美团点评
美团点评
技术年货（中）
~嗝儿~
狗年吃狗粮，代码成就卓越人生

扫码关注美团点评
微信公众号

tech.meituan.com
美团点评技术博客

序

春节已近，年味渐浓。

过去的一年，美团点评作为全球领先的一站式生活服务互联网平台，在吃喝玩乐住行等 200 多个品类，2800 多个城区县，服务了亿万消费者、数百万商家，日订单数超过 2200 万，年度交易总额达到了 3600 亿。

2017 年 10 月最新一轮融资，300 亿美元的估值，也使我们进入全球独角兽的最前列。

在幕后默默支撑和驱动这个平台持续高速发展的，是美团点评技术团队。从最初只有几位工程师的技术组，到如今近 7000 人的大部队，美团点评技术团队已经成为业界一流的研发组织，涵盖了前端（Web、iOS 和 Android）、后台、系统、算法、测试、运维等技术领域。我们已经建成了比较完整的技术体系，包括基于主流开源技术加自研的云计算、大数据、人工智能、基础架构平台，和比较完备的运维、安全、风控等保障系统，更有支持消费者和商家的众多终端软硬件和后台系统。

我们实际上是在为实体经济诸多领域开发全行业的信息基础设施，任重而道远。这些工作当然离不开许许多多技术同行的贡献——开源代码、会议论文、博客、图书……因此我们常怀感恩之心，也希望通过各种方式更多地回馈社区：

- “美团点评技术团队”官方博客诞生于 2013 年 12 月，已累计发表 200 多篇来自一线的技术实践文章；
- 2014 年 9 月，“美团点评技术团队”公众号开始运营，至今有 8 万多业界同行关注；（感谢大家一直的陪伴！）
- “美团点评技术沙龙”自 2015 年举办 33 场，覆盖了北京、上海、厦门和成都四个我们有研发中心的城市。去年我们还举办了 15 次线上沙龙直播。总共超过 1 万人次参与过我们的技术交流；
- 我们还有多个项目开源，在业界大会上发表演讲，在知名技术媒体上发表文章。

不时能听到大家对我们对外分享工作的鼓励，甚至已经有新锐独角兽公司的同学说：“我们是看美团点评技术博客长大的。”（开心醉啦！）

春节放假，技术同学终于有了难得的大块空闲时间。我们特地从过去的技术文章中精选了 58 篇，制作成一本 759 页的电子书。

这本书覆盖了从前端到后台，从技术工程到系统架构，从数据库管理到算法实践，从移动测试到安全运维，并分成前端、后端架构、大数据、数据库、算法与 AI、运维、安全、测试等 8 个大类，作为新春大礼包，送给关注美团点评技术团队公众号的每一位小伙伴。

欢迎大家对书中的技术问题深入探讨，找出 bug，同时能够给我们提供反馈。

也欢迎大家转给有相同兴趣的同事、朋友，让更多同学加入，一起切磋。

祝大家新春快乐，学习成长快乐！

在美团点评，我们信仰耐心和坚持的力量，愿意持续去做一些正确、有积累、可能表面看上去不那么重要实则非常关键的事情。

新的一年，我们将分享更多优质内容，尤其是更加系统全面地展现美团点评技术平台的方方面面。敬请期待！

美团点评技术团队

2018 年春节

目录

算法与 AI 1

深度学习在美团点评的应用 1

深度学习在美团点评推荐平台排序中的运用 10

机器学习中模型优化不得不思考的几个问题 27

人工智能在线特征系统中的生产调度 33

人工智能在线特征系统中的数据存取技术 52

即时配送的 ETA 问题之亿级样本特征构造实践 66

即时配送的订单分配策略：从建模和优化 76

外卖 O2O 的用户画像实践 91

旅游推荐系统的演进 99

美团点评旅游搜索召回策略的演进 120

美团点评联盟广告场景化定向排序机制 137

美团 DSP 广告策略实践 150

大数据 164

美团点评数据平台融合实践 164

美团点评酒旅数据仓库建设实践 186

流计算框架 Flink 与 Storm 的性能对比 194

智能投放系统之场景分析最佳实践 215

智能分析最佳实践——指标逻辑树 225

大圣魔方：美团点评酒旅 BI 报表工具平台开发实践 233

算法与 AI

深度学习在美团点评的应用

文竹 李彪 晓明

前言

近年来，深度学习在语音、图像、自然语言处理等领域取得非常突出的成果，成了最引人注目的技术热点之一。美团点评这两年在深度学习方面也进行了一些探索，其中在自然语言处理领域，我们将深度学习技术应用于文本分析、语义匹配、搜索引擎的排序模型等；在计算机视觉领域，我们将其应用于文字识别、目标检测、图像分类、图像质量排序等。下面我们就以语义匹配、图像质量排序及文字识别这三个应用场景为例，来详细介绍美团点评在深度学习技术及应用方面的经验和方法论。

基于深度学习的语义匹配

语义匹配技术，在信息检索、搜索引擎中有着重要的地位，在结果召回、精准排序等环节发挥着重要作用。

传统意义上讲的语义匹配技术，更加注重文字层面的语义吻合程度，我们暂且称之为语言层的语义匹配；而在美团点评这样典型的 O2O 应用场景下，我们的结果呈现除了和用户表达的语言层语义强相关之外，还和用户意图、用户状态强相关。

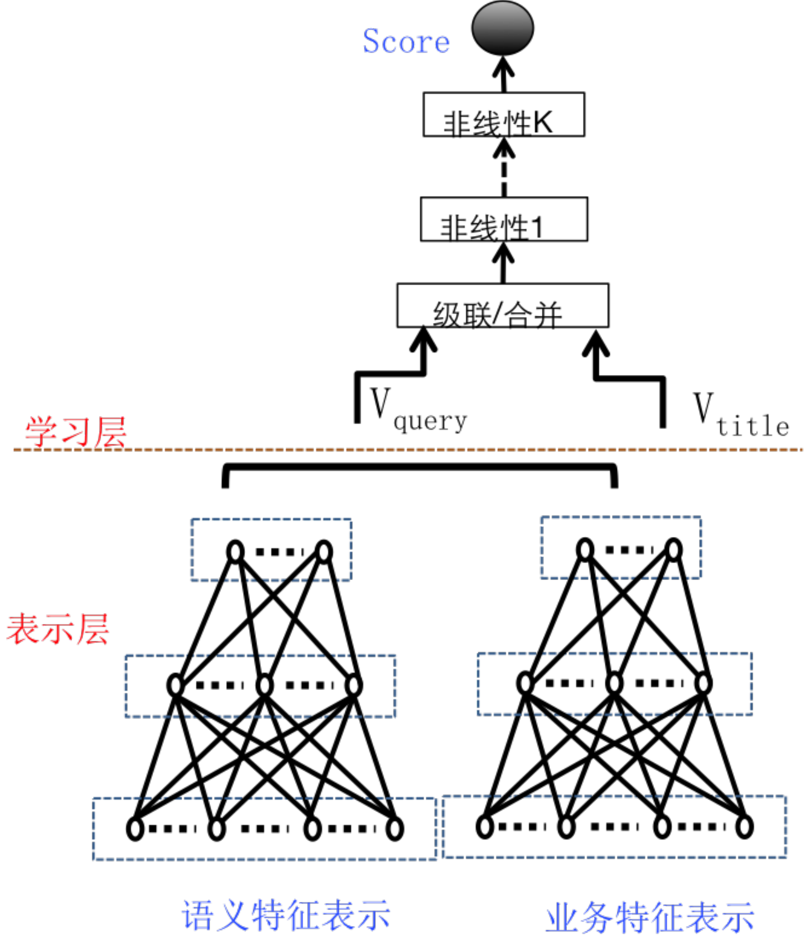
用户意图即用户是来干什么的？比如用户在百度上搜索“关内关外”，他的意图可能是想知道关内和关外代表的地理区域范围，“关内”和“关外”被作为两个词进行检索，而在美团上搜索“关内关外”，用户想找的就是“关内关外”这家饭店，“关内关外”被作为一个词来对待。

再说用户状态，一个在北京和另一个在武汉的用户，在百度或淘宝上搜索任何一

个词条，可能得到的结果不会差太多；但是在美团这样与地理位置强相关的场景下就会完全不一样。比如我在武汉搜“黄鹤楼”，用户找的可能是景点门票，而在北京搜索“黄鹤楼”，用户找的很可能是一家饭店。

如何结合语言层信息和用户意图、状态来做语义匹配呢?

我们的思路是在短文本外引入部分 O2O 业务场景特征，融合到所设计的深度学习语义匹配框架中，通过点击 / 下单数据来指引语义匹配模型的优化方向，最终把训练出的点击相关性模型应用到搜索相关业务中。下图是针对美团点评场景设计的点击相似度框架 ClickNet，是比较轻量级的模型，兼顾了效果和性能两方面，能很好地推广到线上应用。



表示层

对 Query 和商家名分别用语义和业务特征表示，其中语义特征是核心，通过 DNN/CNN/RNN/LSTM/GRU 方法得到短文本的整体向量表示，另外会引入业务相关特征，比如用户或商家的相关信息，比如用户和商家距离、商家评价等，最终结合起来往上传。

学习层

通过多层全连接和非线性变化后，预测匹配得分，根据得分和 Label 来调整网络以学习出 Query 和商家名的点击匹配关系。

在该算法框架上要训练效果很好的语义模型，还需要根据场景做模型调优：首先，我们从训练语料做很多优化，比如考虑样本不均衡、样本重要度、位置 Bias 等方面问题。其次，在模型参数调优时，考虑不同的优化算法、网络大小层次、超参数的调整等问题。经过模型训练优化，我们的语义匹配模型已经在美团点评平台搜索、广告、酒店、旅游等召回和排序系统中上线，有效提升了访购率 / 收入 / 点击率等指标。

小结

深度学习应用在语义匹配上，需要针对业务场景设计合适的算法框架，此外，深度学习算法虽然减少了特征工程工作，但模型调优上难度会增加，因此可以从框架设计、业务语料处理、模型参数调优三方面综合起来考虑，实现一个效果和性能兼优的模型。

基于深度学习的图像质量排序

国内外各大互联网公司（比如腾讯、阿里和 Yelp）的线上广告业务都在关注展示什么样的图像能吸引更多点击。在美团点评，商家的首图是由商家或运营人工指定的，如何选择首图才能更好地吸引用户呢？图像质量排序算法目标就是做到自动选择更优质的首图，以吸引用户点击。

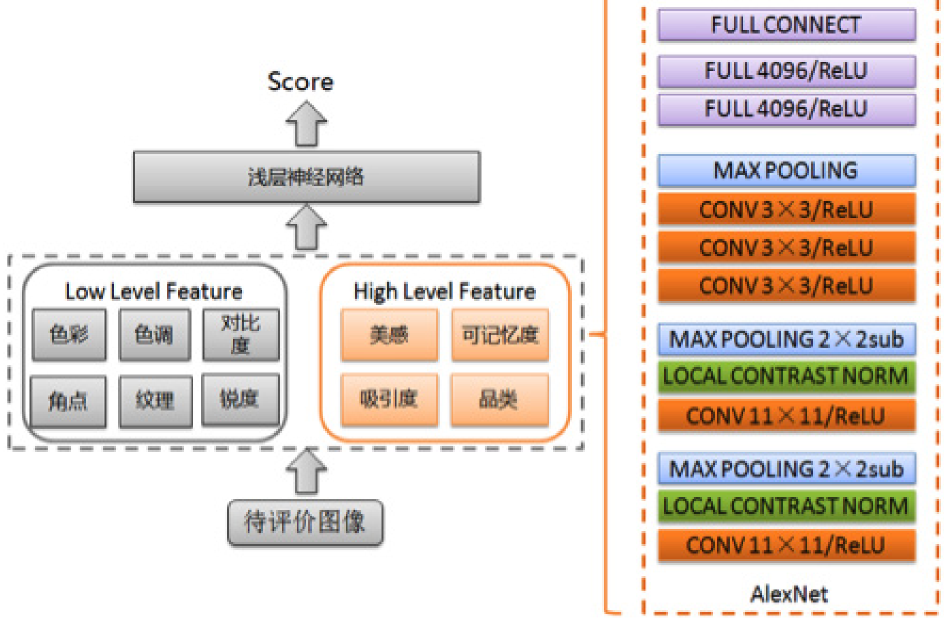
传统的图像质量排序方法主要从美学角度进行质量评价，通过颜色统计、主体分

布、构图等来分析图片的美感。但在实际业务场景中，用户对图片质量优劣的判断主观性很强，难以形成统一的评价标准。比如：

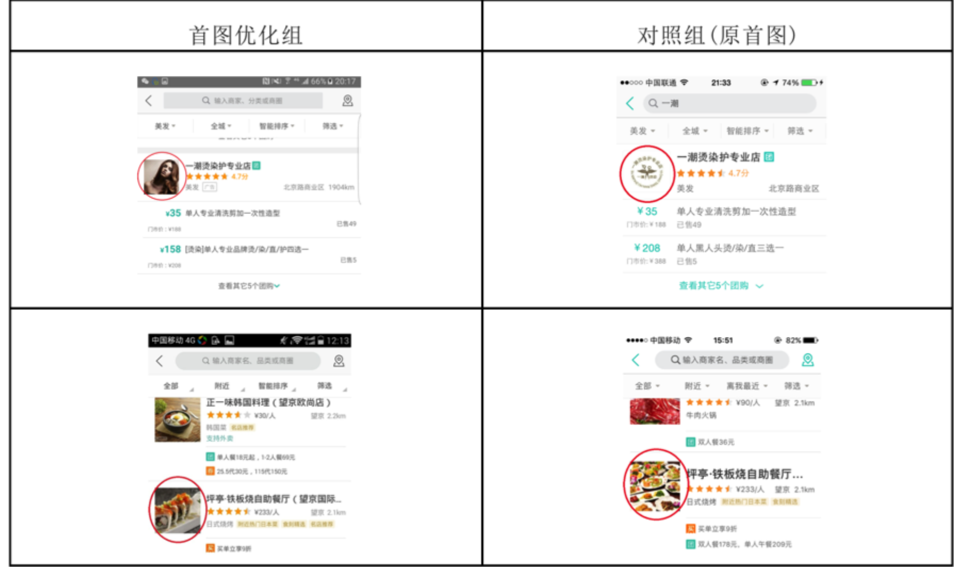
1. 有的用户对清晰度或分辨率更敏感；
2. 有的用户对色彩或构图更敏感；
3. 有的用户偏爱有视觉冲击力的内容而非平淡无奇的环境图。

因此我们使用深度学习方法，去挖掘图片的哪些属性会影响用户的判断，以及如何有效融合这些属性对图片进行评价。

我们使用 AlexNet 去提取图片的高层语义描述，学习美感、可记忆度、吸引度、品类等 High Level 特征，并补充人工设计的 Low Level 特征（比如色彩、锐度、对比度、角点）。在获得这些特征后，训练一个浅层神经网络对图像整体打分。该框架（如图 2 所示）的一个特点是联合了深度学习特征与传统特征，既引入高层语义又保留了低层通用描述，既包括全局特征又有局部特征。



对于每个维度图片属性的学习，都需要大量的标签数据来支撑，但完全通过人工标记代价极大，因此我们借鉴了美团点评的图片来源和 POI 标签体系。关于吸引度属性的学习，我们选取了美团 Deal 相册中点击率高的图片（多数是摄影师通过单反相机拍摄）作为正例，而选取 UGC 相册中点击率低的图片（多数是低端手机拍摄）作为负例。关于品类属性的学习，我们将美团一级品类和常见二级品类作为图片标签。基于上述质量排序模型，我们为广告 POI 挑选最合适的优质首图进行展示，起到吸引用户点击，提高业务指标的目的。图 3 给出了基于质量排序的首图优选结果。



基于深度学习的 OCR

为了提升用户体验，O2O 产品对 OCR 技术的需求已渗透到上单、支付、配送和用户评价等环节。OCR 在美团点评业务中主要起着两方面作用。一方面是辅助录入，比如在移动支付环节通过对银行卡卡号的拍照识别，以实现自动绑卡，又如辅助 BD 录入菜单中菜品信息。另一方面是审核校验，比如在商家资质审核环节对商家上传的身份证、营业执照和餐饮许可证等证件照片进行信息提取和核验以确保该商家的合法性，比如机器过滤商家上单和用户评价环节产生的包含违禁词的图片。相比于传

统 OCR 场景（印刷体、扫描文档），美团的 OCR 场景主要是针对手机拍摄的照片进行文字信息提取和识别，考虑到线下用户的多样性，因此主要面临以下挑战：

- 成像复杂：噪声、模糊、光线变化、形变；
- 文字复杂：字体、字号、色彩、磨损、笔画宽度不固定、方向任意；
- 背景复杂：版面缺失，背景干扰。

对于上述挑战，传统的 OCR 解决方案存在着以下不足：

1. 通过版面分析（二值化，连通域分析）来生成文本行，要求版面结构有较强的规则性且前背景可分性强（例如文档图像、车牌），无法处理前背景复杂的随意文字（例如场景文字、菜单、广告文字等）。
2. 通过人工设计边缘方向特征（例如 HOG）来训练字符识别模型，此类单一的特征在字体变化，模糊或背景干扰时泛化能力迅速下降。
3. 过度依赖字符切分的结果，在字符扭曲、粘连、噪声干扰的情况下，切分的错误传播尤其突出。

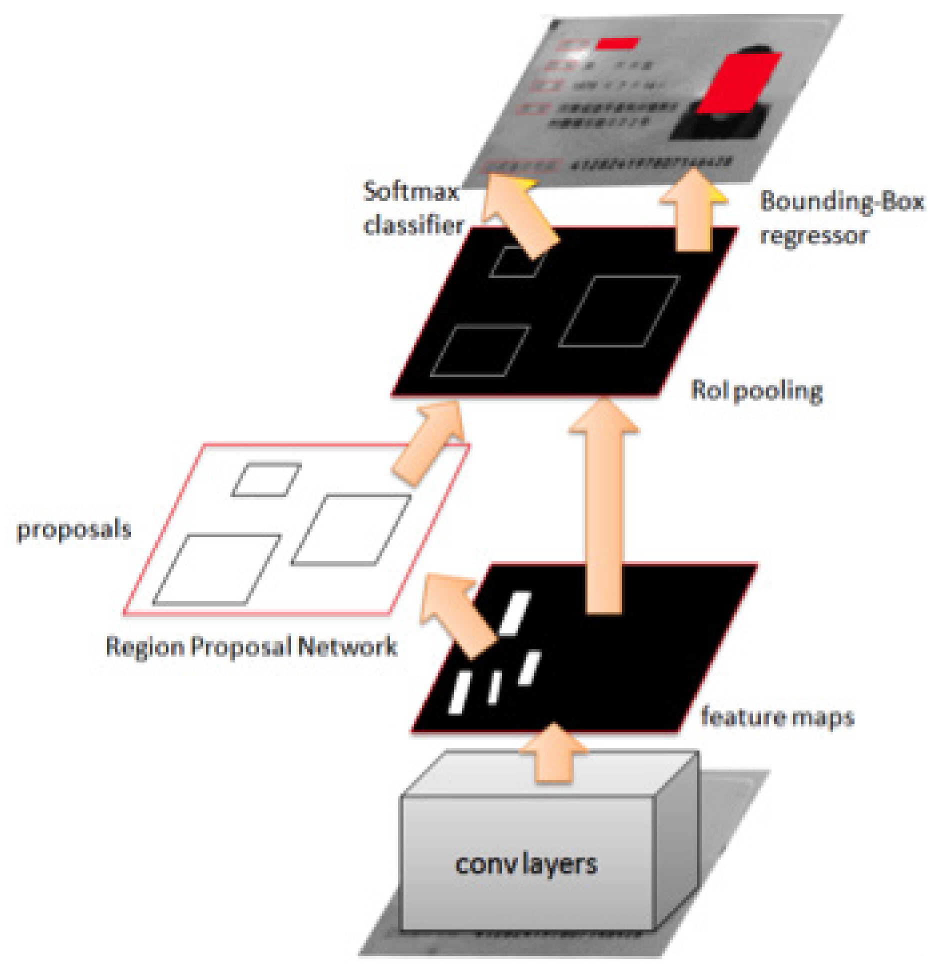
针对传统 OCR 解决方案的不足，我们尝试基于深度学习的 OCR。

1. 基于 Faster R-CNN 和 FCN 的文字定位

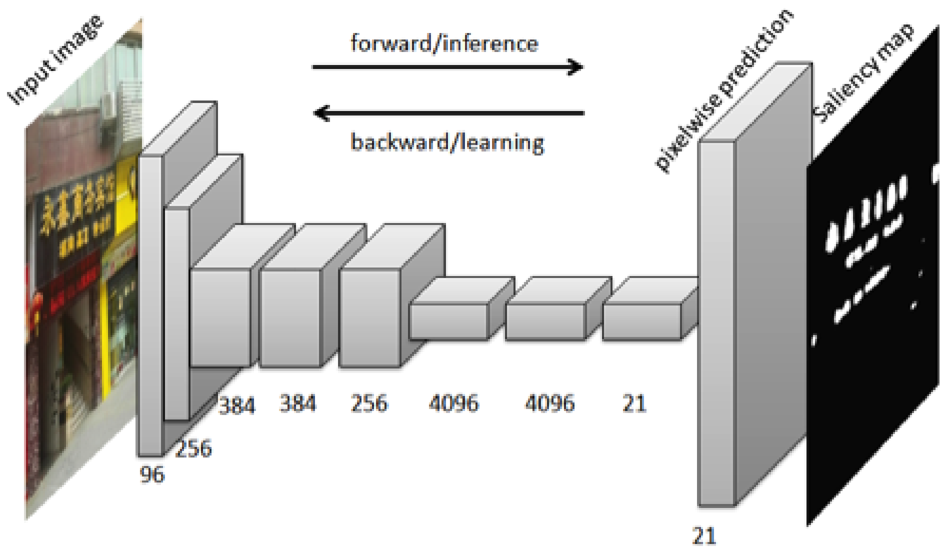
首先，我们根据是否有先验信息将版面划分为受控场景（例如身份证、营业执照、银行卡）和非受控场景（例如菜单、门头图）。

对于受控场景，我们将文字定位转换为对特定关键字目标的检测问题。主要利用 Faster R-CNN 进行检测，如下图所示。为了保证回归框的定位精度同时提升运算速度，我们对原有框架和训练方式进行了微调：

- 考虑到关键字目标的类内变化有限，我们裁剪了 ZF 模型的网络结构，将 5 层卷积减少到 3 层。
- 训练过程中提高正样本的重叠率阈值，并根据业务需求来适配 RPN 层 Anchor 的宽高比。

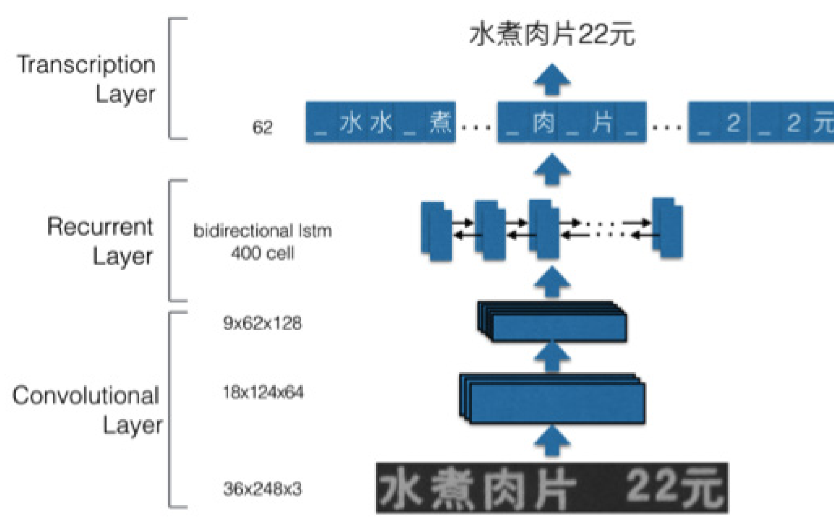


对于非受控场景，由于文字方向和笔画宽度任意变化，目标检测中回归框的定位粒度不够，我们利用语义分割中常用的全卷积网络（FCN）来进行像素级别的文字/背景标注，如下图所示。为了同时保证定位的精度和语义的清晰，我们不仅在最后一层进行反卷积，而且融合了深层 Layer 和浅层 Layer 的反卷积结果

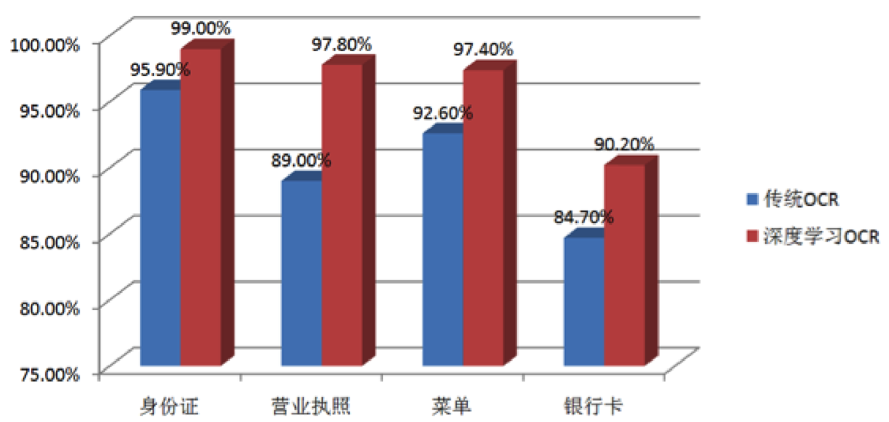


2. 基于序列学习框架的文字识别

为了有效控制字符切分和识别后处理的错误传播效应，实现端到端文字识别的可训练性，我们采用如下图所示的序列学习框架。框架整体分为三层：卷积层，递归层和翻译层。其中卷积层提特征，递归层既学习特征序列中字符特征的先后关系，又学习字符的先后关系，翻译层实现对时间序列分类结果的解码。



由于序列学习框架对训练样本的数量和分布要求较高，我们采用了真实样本 + 合成样本的方式。真实样本以美团点评业务来源（例如菜单、身份证、营业执照）为主，合成样本则考虑了字体、形变、模糊、噪声、背景等因素。基于上述序列学习框架和训练数据，在多种场景的文字识别上都有较大幅度的性能提升，如下图所示。



总结

本文主要以深度学习在自然语言处理、图像处理两个领域的应用为例进行了介绍，但深度学习在美团点评可能发挥的价值远远不限于此。未来，我们将继续在各个场景深入挖掘，比如在智能交互、配送调度、智能运营等，在美团点评产品的智能化道路上贡献一份力量。

作者简介

文竹，美团点评美团平台与酒旅事业群智能技术中心负责人，2010 年从清华硕士毕业后，加入百度，先后从事机器翻译的研发及多个技术团队的管理工作。2015 年 4 月加入美团，负责智能技术中心的管理工作，致力于推动自然语言处理、图像处理、机器学习、用户画像等技术在公司业务上的落地。

李彪，美团点评美团平台及酒旅事业群 NLP 技术负责人，曾就职搜狗、百度。2015 年加入美团点评，致力于 NLP 技术积累和业务的落地，负责的工作包括深度学习平台和模型，文本分析在搜索、广告、推荐等业务上应用，智能客服和交互。

晓明，美团点评平台及酒旅事业群图像技术负责人，曾就职于三星研究院。2015 年加入美团点评，主要致力于图像识别技术的积累和业务落地，作为技术负责人主导了图像机审、首图优选和 OCR 等项目的上线，推进了美团产品的智能化体验和人力成本的节省。

深度学习在美团点评推荐平台排序中的运用

潘晖

美团点评作为国内最大的生活服务平台，业务种类涉及食、住、行、玩、乐等领域，致力于让大家吃得更好，活得更好，有数亿用户以及丰富的用户行为。随着业务的飞速发展，美团点评的用户和商户数在快速增长。在这样的背景下，通过对推荐算法的优化，可以更好的给用户提供感兴趣的内容，帮用户更快速方便的找到所求。我们目标是根据用户的兴趣及行为，向用户推荐感兴趣的内容，打造一个高精准性、高丰富度且让用户感到欣喜的推荐系统。为了达到这个目的，我们在不停的尝试将新的算法、新的技术进引入到现有的框架中。

1. 引言

自 2012 年 ImageNet 大赛技惊四座后，深度学习已经成为近年来机器学习和人工智能领域中关注度最高的技术。在深度学习出现之前，人们借助 SIFT、HOG 等算法提取具有良好区分性的特征，再结合 SVM 等机器学习算法进行图像识别。然而 SIFT 这类算法提取的特征是有局限性的，导致当时比赛的最好结果的错误率也在 26% 以上。卷积神经网络（CNN）的首次亮相就将错误率一下由 26% 降低到 15%，同年微软团队发布的论文中显示，通过深度学习可以将 ImageNet 2012 资料集的错误率降到 4.94%。

随后的几年，深度学习在多个应用领域都取得了令人瞩目的进展，如语音识别、图像识别、自然语言处理等。鉴于深度学习的潜力，各大互联网公司也纷纷投入资源开展科研与运用。因为人们意识到，在大数据时代，更加复杂且强大的深度模型，能深刻揭示海量数据里所承载的复杂而丰富的信息，并对未来或未知事件做更精准的预测。

美团点评作为一直致力于站在科技前沿的互联网公司，也在深度学习方面进行了一些探索，其中在自然语言处理领域，我们将深度学习技术应用于文本分析、语义匹配、搜索引擎的排序模型等；在计算机视觉领域，我们将其应用于文字识别、图像分类、图像质量排序等。本文就是笔者所在团队，在借鉴了 Google 在 2016 年提出的

Wide & Deep Learning 的思想上，基于自身业务的一些特点，在大众点评推荐系统上做出的一些思考和取得的实践经验。

2. 点评推荐系统介绍

与大部分的推荐系统不同，美团点评的场景由于自身业务的多样性，使得我们很难准确捕获用户的兴趣点或用户的实时意图。而且我们推荐的场景也会随着用户兴趣、地点、环境、时间等变化而变化。点评推荐系统主要面临以下几点挑战：

- **业务形态多样性：**除了推荐商户外，我们还根据不同的场景，进行实时判断，从而推出不同形态的业务，如团单、酒店、景点、霸王餐等。
- **用户消费场景多样性：**用户可以选择在家消费：外卖，到店消费：团单、闪惠，或者差旅消费：预定酒店等。

针对上述问题，我们定制了一套完善的推荐系统框架，包括基于机器学习的多选品召回与排序策略，以及从海量大数据的离线计算到高并发在线服务的推荐引擎。推荐系统的策略主要分为召回和排序两个过程，召回主要负责生成推荐的候选集，排序负责将多个算法策略的结果进行个性化排序。

召回层：我们通过用户行为、场景等进行实时判断，通过多个召回策略召回不同候选集。再对召回的候选集进行融合。候选集融合和过滤层有两个功能，一是提高推荐策略的覆盖度和精度；另外还要承担一定的过滤职责，从产品、运营的角度制定一些人工规则，过滤掉不符合条件的 Item。下面是一些我们常用到的召回策略：

- **User-Based 协同过滤：**找出与当前 User X 最相似的 N 个 User，并根据 N 个 User 对某 Item 的打分估计 X 对该 Item 的打分。在相似度算法方面，我们采用了 Jaccard Similarity：

$$sim(x, y) = \frac{r_x \cap r_y}{r_x \cup r_y}$$

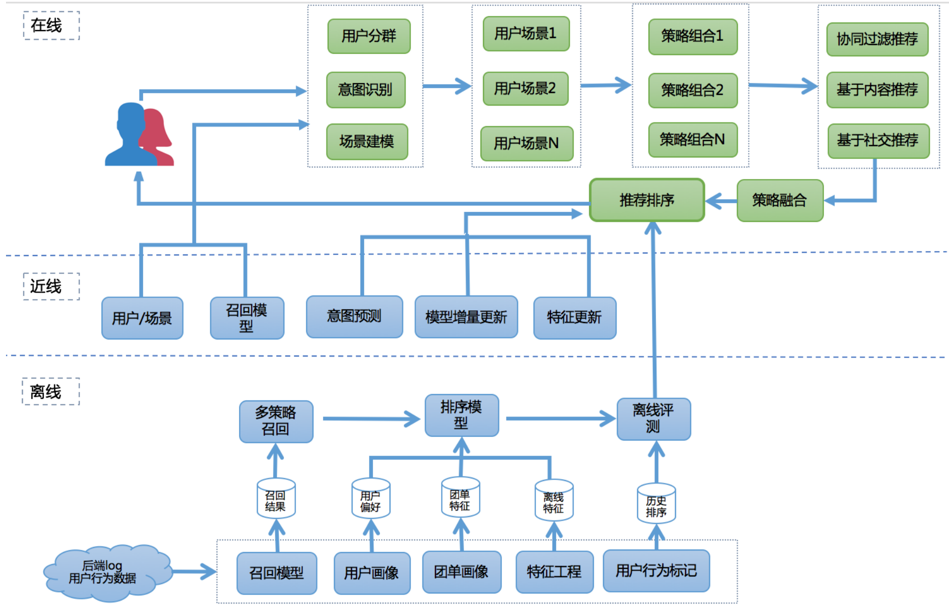
r_x，r_y 分别表示用户对Item集合的打分。

- **Model-Based 协同过滤：**用一组隐含因子来联系用户和商品。其中每个用户、每个商品都用一个向量来表示，用户 u 对商品 i 的评价通过计算这两个向量的内积得到。算法的关键在于根据已知的用户对商品的行为数据来估计用户和商品的隐因子向量。
- **Item-Based 协同过滤：**我们先用 word2vec 对每个 Item 取其隐含空间的向量，然后用 Cosine Similarity 计算用户 u 用过的每一个 Item 与未用过 Item i 之间的相似性。最后对 Top N 的结果进行召回。
- **Query-Based：**是根据 Query 中包含的实时信息（如地理位置信息、WiFi 到店、关键词搜索、导航搜索等）对用户的意图进行抽象，从而触发的策略。
- **Location-Based：**移动设备的位置是经常发生变化的，不同的地理位置反映了不同的用户场景，可以在具体的业务中充分利用。在推荐的候选集召回中，我们也会根据用户的实时地理位置、工作地、居住地等地理位置触发相应的策略。

排序层：每类召回策略都会召回一定的结果，这些结果去重后需要统一做排序。点评推荐排序的框架大致可以分为三块：

- **离线计算层：**离线计算层主要包含了算法集合、算法引擎，负责数据的整合、特征的提取、模型的训练、以及线下的评估。
- **近线实时数据流：**主要是对不同的用户流实施订阅、行为预测，并利用各种数据处理工具对原始日志进行清洗，处理成格式化的数据，落地到不同类型的存储系统中，供下游的算法和模型使用。
- **在线实时打分：**根据用户所处的场景，提取出相对应的特征，并利用多种机器学习算法，对多策略召回的结果进行融合和打分重排。

具体的推荐流程图如下：

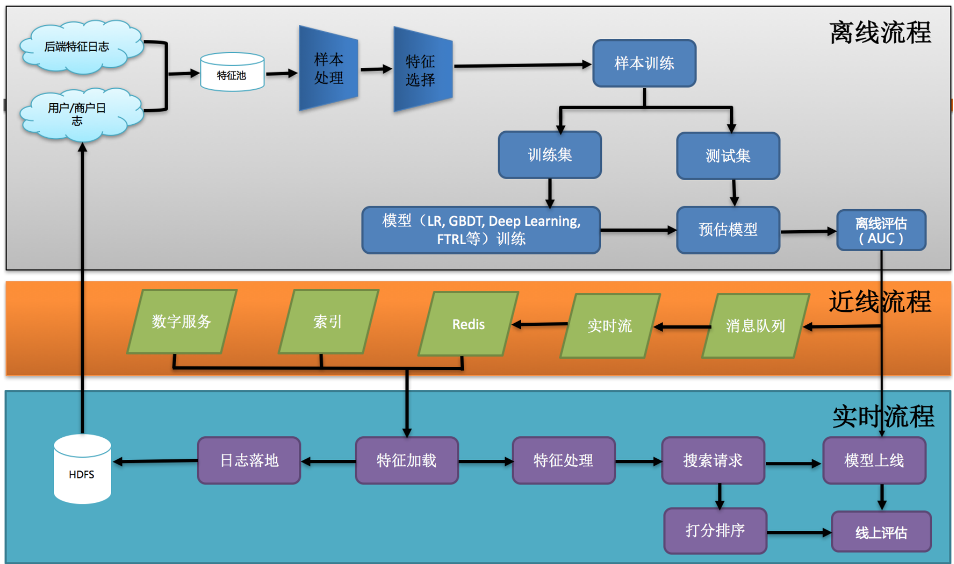


从整体框架的角度看，当用户每次请求时，系统就会将当前请求的数据写入到日志当中，利用各种数据处理工具对原始日志进行清洗，格式化，落地到不同类型的存储系统中。在训练时，我们利用特征工程，从处理过后的数据集中选出训练、测试样本集，并借此进行线下模型的训练和预估。我们采用多种机器学习算法，并通过线下 AUC、NDCG、Precision 等指标来评估他们的表现。线下模型经过训练和评估后，如果在测试集有比较明显的提高，会将其上线进行线上 AB 测试。同时，我们也有多种维度的报表对模型进行数据上的支持。

3. 深度学习在点评推荐排序系统中应用

对于不同召回策略所产生的候选集，如果只是根据算法的历史效果决定算法产生的 Item 的位置显得有些简单粗暴，同时，在每个算法的内部，不同 Item 的顺序也只是简单的由一个或者几个因素决定，这些排序的方法只能用于第一步的初选过程，最终的排序结果需要借助机器学习的方法，使用相关的排序模型，综合多方面的因素来确定。

3.1 现有排序框架介绍



到目前为止，点评推荐排序系统尝试了多种线性、非线性、混合模型等机器学习方法，如逻辑回归、GBDT、GBDT+LR 等。通过线上实验发现，相较于线性模型，传统的非线性模型如 GBDT，并不一定能在线上 AB 测试环节对 CTR 预估有比较明显的提高。而线性模型如逻辑回归，因为自身非线性表现能力比较弱，无法对真实生活中的非线性场景进行区分，会经常对历史数据中出现过的数据过度记忆。下图就是线性模型根据记忆将一些历史点击过的单子排在前面：

从图中我们可以看到，系统在非常靠前的位置推荐了一些远距离的商户，因为这些商户曾经被用户点过，其本身点击率较高，那么就很容易被系统再次推荐出来。但这种推荐并没有结合当前场景给用户推荐出一些有新颖性的 Item。为了解决这个问题，就需要考虑更多、更复杂的特征，比如组合特征来替代简单的“距离”特征。怎么去定义、组合特征，这个过程成本很高，并且更多地依赖于人工经验。

而深度神经网络，可以通过低维密集的特征，学习到以前没出现过的一些 Item 和特征之间的关系，并且相比于线性模型大幅降低了对于特征工程的需求，从而吸引我们进行探索研究。

在实际的运用当中，我们根据 Google 在 2016 年提出的 Wide & Deep Learning 模型，并结合自身业务的需求与特点，将线性模型组件和深度神经网络进

行融合，形成了在一个模型中实现记忆和泛化的宽深度学习框架。在接下来的章节中，将会讨论如何进行样本筛选、特征处理、深度学习算法实现等。

3.2 样本的筛选

数据及特征，是整个机器学习中最重要的两个环节，因为其本身就决定了整个模型的上限。点评推荐由于其自身多业务（包含外卖、商户、团购、酒旅等）、多场景（用户到店、用户在家、异地请求等）的特色，导致我们的样本集相比于其他产品更多元化。我们的目标是预测用户的点击行为。有点击的为正样本，无点击的为负样本，同时，在训练时对于购买过的样本进行一定程度的加权。而且，为了防止过拟合 / 欠拟合，我们将正负样本的比例控制在 10%。最后，我们还要对训练样本进行清洗，去除掉 Noise 样本（特征值近似或相同的情况下，分别对应正负两种样本）。

同时，推荐业务作为整个 App 首页核心模块，对于新颖性以及多样性的需求是很高的。在点评推荐系统的实现中，首先要确定应用场景的数据，美团点评的数据可以分为以下几类：

- **用户画像：**性别、常驻地、价格偏好、Item 偏好等。
- **Item 画像：**包含了商户、外卖、团单等多种 Item。其中商户特征包括：商户价格、商户好评数、商户地理位置等。外卖特征包括：外卖平均价格、外卖配送时间、外卖销量等。团单特征包括：团单适用人数、团单访购率等。
- **场景画像：**用户当前所在地、时间、定位附近商圈、基于用户的上下文场景信息等。

3.3 深度学习中的特征处理

机器学习的另一个核心领域就是特征工程，包括数据预处理，特征提取，特征选择等。

1. **特征提取：**从原始数据出发构造新的特征的过程。方法包括计算各种简单统计量、主成分分析、无监督聚类，在构造方法确定后，可以将其变成一个自动化的数据处理流程，但是特征构造过程的核心还是手动的。

2. **特征选择：**从众多特征中挑选出少许有用特征。与学习目标不相关的特征和冗余特征需要被剔除，如果计算资源不足或者对模型的复杂性有限制的话，还需要选择丢弃一些不重要的特征。特征选择方法常用的有以下几种：

表1:常用的特征选择方法

计算每一个特征与响应变量的相关性	通过计算皮尔逊系数和互信息系数计算相关性，再通过排序选择特征。
构建单个特征的模型	通过模型的准确性为特征排序，借此来选择特征
通过L1正则项来选择特征	因为L1具有稀疏解的特性，因此具备特征选择的特性。同时也可以用L2交叉验证。
训练能够对特征打分的预选模型	Random Forest和Logistic Regression等都能对模型的特征打分，通过打分获得相关性后再训练最终模型
通过特征组合后再来选择特征	对用户id和用户特征最组合来获得较大的特征集再来选择特征
通过深度学习来进行特征选择	通过深度学习具有自动学习特征的能力，从深度学习模型中选择某一神经层的特征后，用来进行最终目标模型的训练。

特征选择开销大、特征构造成本高，在推荐业务开展的初期，我们对于这方面的感觉还不强烈。但是随着业务的发展，对点击率预估模型的要求越来越高，特征工程的巨大投入对于效果的提升已经不能满足我们需求，于是我们想寻求一种新的解决办法。

深度学习能自动对输入的低阶特征进行组合、变换，得到高阶特征的特性，也促使我们转向深度学习进行探索。深度学习“自动提取特征”的优点，在不同的领域有着不同的表现。例如对于图像处理，像素点可以作为低阶特征输入，通过卷积层自动得到的高阶特征有比较好的效果。在自然语言处理方面，有些语义并不来自数据，而是来自人们的先验知识，利用先验知识构造的特征是很有帮助的。

因此，我们希望借助于深度学习来节约特征工程中的巨大投入，更多地让点击率预估模型和各辅助模型自动完成特征构造和特征选择的工作，并始终和业务目标保持一致。下面是一些我们在深度学习中用到的特征处理方式：

3.3.1 组合特征

对于特征的处理，我们沿用了目前业内通用的办法，比如归一化、标准化、离散化等。但值得一提的是，我们将很多组合特征引入到模型训练中。因为不同特征之间的组合是非常有效的，并有很好的可解释性，比如我们将 " 商户是否在用户常驻地 "、" 用户是否在常驻地 " 以及 " 商户与用户当前距离 " 进行组合，再将数据进行离散化，

通过组合特征，我们可以很好的抓住离散特征中的内在联系，为线性模型增加更多的非线性表述。组合特征的定义为：

$$\phi_k(x) = \prod x_i^{c_{ki}}, c_{ki} \in (0, 1)$$

3.3.2 归一化

归一化是依照特征矩阵的行处理数据，其目的在于样本向量在点乘运算或其他核函数计算相似性时，拥有统一的标准，也就是说都转化为“单位向量”。在实际工程中，我们运用了两种归一化方法：

Min-Max:

$$x' = \frac{x - min}{max - min}$$

Min 是这个特征的最小值，Max 是这个特征的最大值。

Cumulative Distribution Function（CDF）: CDF 也称为累积分布函数，数学意义是表示随机变量小于或等于其某一个取值 x 的概率。其公式为：

$$x' = \int_{-\infty}^{x} f(x)\, dx$$

在我们线下实验中，连续特征在经过 CDF 的处理后，相比于 Min-Max，CDF 的线下 AUC 提高不足 0.1%。我们猜想是因为有些连续特征并不满足在（0，1）上均匀分布的随机函数，CDF 在这种情况下，不如 Min-Max 来的直观有效，所以我们在线上采用了 Min-Max 方法。

3.3.3 快速聚合

为了让模型更快的聚合，并且赋予网络更好的表现形式，我们对原始的每一个连续特征设置了它的 super-liner 和 sub-liner，即对于每个特征 x，衍生出 2 个子特征：

$$x^2$$

$$\sqrt{x}$$

实验结果表示，通过对每一个连续变量引入 2 个子特征，会提高线下 AUC 的表现，但考虑到线上计算量的问题，并没有在线上实验中添加这 2 个子特征。

3.4 优化器（Optimizer）的选择

在深度学习中，选择合适的优化器不仅会加速整个神经网络训练过程，并且会避免在训练的过程中困到鞍点。文中会结合自己的使用情况，对使用过的优化器提出一些自己的理解。

3.4.1 Stochastic Gradient Descent (SGD)

SGD 是一种常见的优化方法，即每次迭代计算 Mini-Batch 的梯度，然后对参数进行更新。其公式为：

$$\nu_t = \mu \nabla_\theta J(\theta)$$

缺点是对于损失方程有比较严重的振荡，并且容易收敛到局部最小值。

3.4.2 Momentum

为了克服 SGD 振荡比较严重的问题，Momentum 将物理中的动量概念引入到 SGD 当中，通过积累之前的动量来替代梯度。即：

$$\nu_t = \gamma \nu_{t-1} + \mu \nabla_\theta J(\theta)$$

$$\theta = \theta - \nu_t$$

相较于 SGD，Momentum 就相当于在从山坡上不停的向下走，当没有阻力的话，它的动量会越来越大，但是如果遇到了阻力，速度就会变小。也就是说，在训练

的时候，在梯度方向不变的维度上，训练速度变快，梯度方向有所改变的维度上，更新速度变慢，这样就可以加快收敛并减小振荡。

3.4.3 Adagrad

相较于 SGD，Adagrad 相当于对学习率多加了一个约束，即：

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\mu}{\sqrt{\sum g_{t,i} + \epsilon}}$$

Adagrad 的优点是，在训练初期，由于 gt 较小，所以约束项能够加速训练。而在后期，随着 gt 的变大，会导致分母不断变大，最终训练提前结束。

3.4.4 Adam

Adam 是一个结合了 Momentum 与 Adagrad 的产物，它既考虑到了利用动量项来加速训练过程，又考虑到对于学习率的约束。利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率。Adam 的优点主要在于经过偏置校正后，每一次迭代学习率都有个确定范围，使得参数比较平稳。其公式为：

$$\theta_{t+1} = \theta_t - \frac{\mu}{\sqrt{v_t^1} + \epsilon} m_t^1$$

其中：

$$m_t^1 = \frac{m_t}{1 - \beta_1^t}$$

$$v_t^1 = \frac{v_t}{1 - \beta_2^t}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

小结

通过实践证明，Adam 结合了 Adagrad 善于处理稀疏梯度和 Momentum 善于处理非平稳目标的优点，相较于其他几种优化器效果更好。同时，我们也注意到很多论文中都会引用 SGD，Adagrad 作为优化函数。但相较于其他方法，在实践中，SGD 需要更多的训练时间以及可能会被困到鞍点的缺点，都制约了它在很多真实数据上的表现。

3.5 损失函数的选择

深度学习同样有许多损失函数可供选择，如平方差函数（Mean Squared Error），绝对平方差函数（Mean Absolute Error），交叉熵函数（Cross Entropy）等。而在理论与实践中，我们发现 Cross Entropy 相比于在线性模型中表现比较好的平方差函数有着比较明显的优势。其主要原因是在深度学习通过反向传递更新 W 和 b 的同时，激活函数 Sigmoid 的导数在取大部分值时会落入左、右两个饱和区间，造成参数的更新非常缓慢。具体的推导公式如下：

一般的 MSE 被定义为：

$$C = \frac{1}{2}(a - y)^2$$

其中 y 是我们期望的输出，a 为神经元的实际输出 a= σ (Wx+b)。由于深度学习反向传递的机制，权值 W 与偏移量 b 的修正公式被定义为：

$$\frac{\partial C}{\partial W} = (a - y)\sigma'(a)x^T$$

$$\frac{\partial C}{\partial b} = (a - y)\sigma'(a)$$

因为 Sigmoid 函数的性质，导致 $\sigma'(z)$ 在 z 取大部分值时会造成饱和现象。

Cross Entropy 的公式为：

$$H(y,a) = -\sum y_i log(a_i)$$

如果有多个样本，则整个样本集的平均交叉熵为：

$$H(y,a) = -\frac{1}{n}\sum\sum y_{i,n} log(a_{i,n})$$

其中 n 表示样本编号，i 表示类别编号。 如果用于 Logistic 分类，则上式可以简化成：

$$H(y,a) = -\frac{1}{n}\sum ylog(a) + (1-y)log(1-a)$$

与平方损失函数相比，交叉熵函数有个非常好的特质：

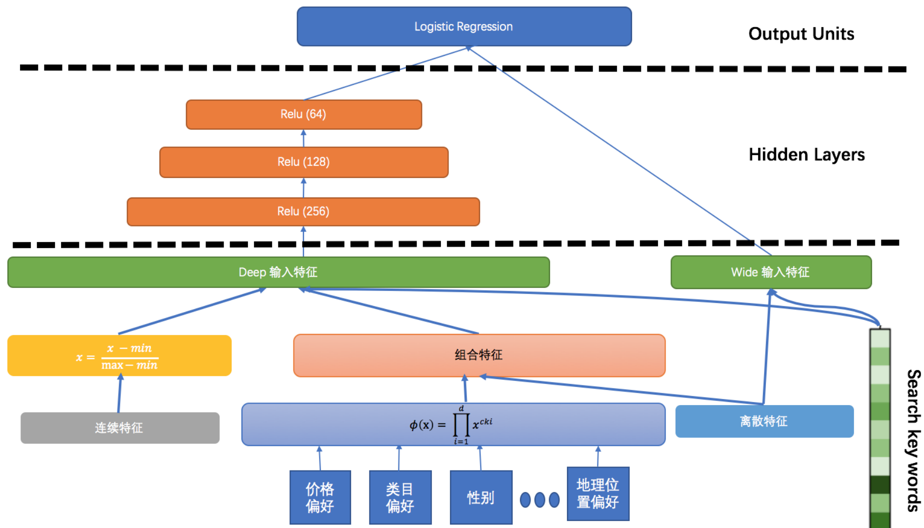
$$H(y,a) = \frac{1}{n}\sum(a_n - y_n) = \frac{1}{n}\sum(\sigma(z_n) - y_n)$$

可以看到，由于没有了 σ' 这一项，这样一来在更新 w 和 b 就不会受到饱和性的影响。当误差大的时候，权重更新就快，当误差小的时候，权重的更新就慢。

3.6 宽深度模型框架

在实验初期，我们只将单独的 5 层 DNN 模型与线性模型进行了比对。通过线下 / 线上 AUC 对比，我们发现单纯的 DNN 模型对于 CTR 的提升并不明显。而且单独的 DNN 模型本身也有一些瓶颈，例如，当用户本身是非活跃用户时，由于其自身与 Item 之间的交互比较少，导致得到的特征向量会非常稀疏，而深度学习模型在处理这种情况时有可能会过度的泛化，导致推荐与该用户本身相关较少的 Item。因此，

我们将广泛线性模型与深度学习模型相结合，同时又包含了一些组合特征，以便更好的抓住 Item-Feature-Label 三者之间的共性关系。我们希望在宽深度模型中的宽线性部分可以利用交叉特征去有效地记忆稀疏特征之间的相互作用，而在深层神经网络部分通过挖掘特征之间的相互作用，提升模型之间的泛化能力。下图就是我们的宽深度学习模型框架：



在离线阶段，我们采用基于 Theano、Tensorflow 的 Keras 作为模型引擎。在训练时，我们分别对样本数据进行清洗和提权。在特征方面，对于连续特征，我们用 Min-Max 方法做归一化。在交叉特征方面，我们结合业务需求，提炼出多个在业务场景意义比较重大的交叉特征。在模型方面我们用 Adam 做为优化器，用 Cross Entropy 做为损失函数。在训练期间，与 Wide & Deep Learning 论文中不同之处在于，我们将组合特征作为输入层分别输入到对应的 Deep 组件和 Wide 组件中。然后在 Deep 部分将全部输入数据送到 3 个 ReLU 层，在最后通过 Sigmoid 层进行打分。我们的 Wide & Deep 模型在超过 7000 万个训练数据中进行了训练，并用超过 3000 万的测试数据进行线下模型预估。我们的 Batch – Size 设为 50000，Epoch 设为 20。

4. 深度学习线下 / 线上效果

在实验阶段，分别将深度学习、宽深度学习以及逻辑回归做了一系列的对比，将表现比较好的宽深度模型放在线上与原本的 Base 模型进行 AB 实验。从结果上来看，宽深度学习模型在线下 / 线上都有比较好的效果。具体结论如下：

表2:不同模型之间AUC对比

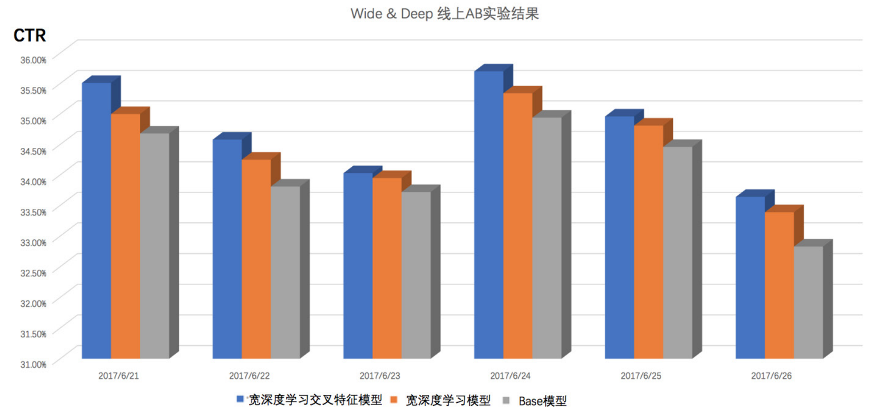
算法版本	AUC
Base Model	68.85%
Deep Learning (256 hidden units)	69.98%
Wide & Deep without Cross feature (256 hidden units)	70.65%
Wide & Deep with Cross feature (256 hidden units)	71.81%

随着隐藏层宽度的增加，线下训练的效果也会随着逐步的提升。但考虑到线上实时预测的性能问题，我们目前采用 256->128->64 的框架结构。

表3:不同隐藏层之间，AUC对比

隐藏层	(Wide & Deep)AUC with cross features
512 ReLU	72.21%
256 ReLU	71.81%
256 ReLU -> 128 ReLU -> 64 ReLU	71.66%

下图是包含了组合特征的宽深度模型与 Base 模型的线上实验效果对比图：



从线上效果来看，宽深度学习模型一定程度上解决了历史点击过的团单在远距离会被召回的问题。同时，宽深度模型也会根据当前的场景推荐一些有新颖性的 Item。

5. 总结

排序是一个非常经典的机器学习问题，实现模型的记忆和泛化功能是推荐系统中的一个挑战。记忆可以被定义为在推荐中将历史数据重现，而泛化是基于数据相关性的传递性，探索过去从未或很少发生的 Item。宽深度模型中的宽线性部分可以利用交叉特征去有效地记忆稀疏特征之间的相互作用，而深层神经网络可以通过挖掘特征之间的相互作用，提升模型之间的泛化能力。在线实验结果表明，宽深度模型对 CTR 有比较明显的提高。同时，我们也在尝试将模型进行一系列的演化：

1. **将 RNN 融入到现有框架。现有的 Deep & Wide 模型只是将 DNN 与线性模型做融合，并没有对时间序列上的变化进行建模。样本出现的时间顺序对于推荐排序同样重要，比如当一个用户按照时间分别浏览了一些异地酒店、景点时，用户再次再请求该异地城市，就应该推出该景点周围的美食。**
2. **引入强化学习，让模型可以根据用户所处的场景，动态地推荐内容。**

深度学习和逻辑回归的融合使得我们可以兼得二者的优点，也为进一步的点击率预估模型设计和优化打下了坚实的基础。

6. 参考文献

1. H. Cheng, L. Koc, J. Harmsen et al, Wide & Deep Learning for Recommender Systems, DLRS 2016 Proceedings of the 1st Workshop on Deep Learning for Recommender Systems.
2. P. Covington, J. Adams, E. Sargin, Deep Neural Networks for YouTube Recommendations, RecSys ‘16 Proceedings of the 10th ACM Conference on Recommender Systems.
3. H. Wang, N. Wang, D. Yeung, Collaborative Deep Learning for Recommender Systems, KDD ‘15 Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

7. 作者简介

潘晖，美团点评高级算法工程师。2015 年博士毕业后加入微软，主要从事自然语言处理的研发。2016 年 12 月加入美团点评，现在负责大众点评的排序业务，致力于用大数据和机器学习技术解决业务问题，提升用户体验。

搜索推荐技术中心：负责点评侧基础检索框架及通用搜索推荐平台的建设；通过大数据及人工智能技术，优化搜索列表的端到端用户体验，提升推荐展位的精准性及新颖性；构建智能技术平台，支持点评侧业务的智能化需求。我们的使命是用搜索推荐技术有效连接人，商家及服务，帮助用户精准高效地发现信息内容，优化用户体验，扩展用户需求，推动业务发展。

机器学习中模型优化不得不思考的几个问题

胡淏

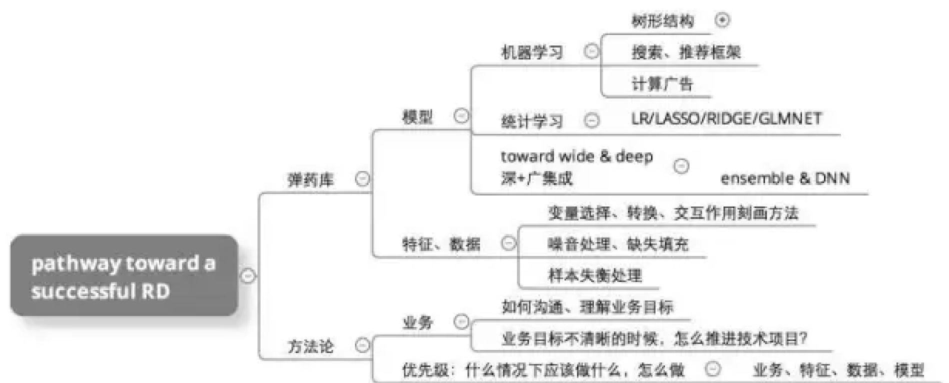


图 1　机器学习工程师的知识图谱

图 1 列出了我认为一个成功的机器学习工程师需要关注和积累的点。机器学习实践中，我们平时都在积累自己的“弹药库”：分类、回归、无监督模型、Kaggle 上面特征变换的黑魔法、样本失衡的处理方法、缺失值填充……这些大概可以归类成模型和特征两个点。我们需要参考成熟的做法、论文，并自己实现，此外还需要多反思自己方法上是否还可以改进。如果模型和特征这两个点都已经做得很好了，你就拥有了一张绿卡，能跨过在数据相关行业发挥模型技术价值的准入门槛。

在这个时候，比较关键的一步，就是高效的**技术变现能力**。所谓高效，就是解决业务核心问题的专业能力。本文将描述这些专业能力，也就是模型优化的四个要素：模型、数据、特征、业务，还有更重要的，就是它们在模型项目中的优先级。

模型项目推进的四要素

项目推进过程中，四个要素相互之间的优先级大致是：业务 > 特征 > 数据 > 模型。

低优先级

高优先级

技术选型：

业务导向：在项目排期、业务预期等多方面约束下，恰到好处的选型
优先级：项目最后一步
模型本身：GBDT、LASSO

特征：

价值：突破项目天花板的有效武器
业务导向：特征源自运营，用特征充分刻画业务理解
方法论：变量体系、研发流程，旧特征精细化、新特征拓展（高维分类变量）

数据：数据是模型性能的上确界

训练数据 **一致对接** 线上预测的业务场景
噪音的过滤 与 **99%准则**

业务：

界定问题：业务核心目标、场景（技术目标要倒推）
评估方案；
追求闭环：项目的输出是什么？如何运营？

图 2　四要素解决问题细分 + 优先级

业务

一个模型项目有好的技术选型、完备的特征体系、高质量的数据一定是很加分的，不过真正决定项目好与坏还有一个大前提，就是这个项目的技术目标是否在解决当下核心业务问题。

业务问题包含两个方面：业务 KPI 和 deadline。举个例子，业务问题是在两周之内降低目前手机丢失带来的支付宝销赃风险。这时如果你的方案是研发手机丢失的核心特征，比如改密是否合理，基本上就死的很惨，因为两周根本完不成，改密合理性也未必是模型优化好的切入点；反之，如果你的方案是和运营同学看 bad case，梳理现阶段的作案通用手段，并通过分析上线一个简单模型或者业务规则的补丁，就明智很多。如果上线后，案件量真掉下来了，就算你的方案准确率很糟、方法很 low，但你解决了业务问题，这才是最重要的。

虽然业务目标很关键，不过一般讲，业务运营同学真的不太懂得如何和技术有效的沟通业务目标，比如：

1. 我们想做一个线下门店风险评级的项目，希望运营通过反作弊模型角度帮我们给门店打个分，这个分数包含的问题有：风险是怎么定义的、为什么要做风险评级、更大的业务目标是什么、怎么排期的、这个风险和我们反作弊模型之间的业务关系你是怎么看的？

2. 做一个区域未来 10min 的配送时间预估模型。我们想通过运营的模型衡量在恶劣天气的时候每个区域的运力是否被击穿（业务现状和排期？运力被击穿可以扫下盲么？运力击穿和配送时间之间是个什么业务逻辑、时间预估是刻画运力紧张度的最有效手段么？项目的关键场景是恶劣天气的话，我们仅仅训练恶劣天气场景的时间预估模型是否就好了？）。

为了保证整个技术项目没有做偏，项目一开始一定要和业务聊清楚三件事情：

1. 业务核心问题、关键场景是什么。
2. 如何评估该项目的成功，指标是什么。
3. 通过项目输出什么关键信息给到业务，业务如何运营这个信息从而达到业务目标。

项目过程中，也要时刻回到业务，检查项目的健康度。

要说正确的业务理解和切入，在为技术项目保驾护航，数据、特征便是一个模型项目性能方面的天花板。garbage in，garbage out 就在说这个问题。

这两天有位听众微信问我一个很难回答的问题，大概意思是，数据是特征拼起来构成的集合嘛，所以这不是两个要素。从逻辑上面讲，数据的确是一列一列的特征，不过数据与特征在概念层面是不同的：数据是已经采集的信息，特征是以兼容模型、最优化为目标对数据进行加工。就比如通过 word2vec 将非结构化数据结构化，就是将数据转化为特征的过程。

所以，我更认为特征工程是基于数据的一个非常精细、刻意的加工过程。从传统的特征转换、交互，到 embedding、word2vec、高维分类变量数值化，最终目的都是更好的去利用现有的数据。之前有聊到的将推荐算法引入有监督学习模型优化中的做法，就是在把两个本不可用的高维 ID 类变量变成可用的数值变量。

观察到自己和童鞋们在特征工程中会遇到一些普遍问题，比如，特征设计不全面，没有耐心把现有特征做得细致……也整理出来一套方法论，仅供参考：

图 3　变量体系、研发流程

在特征设计的时候，有两个点可以帮助我们把特征想的更全面：

1. 现有的基础数据。
2. 业务“二维图”。

这两个方面的整合，就是一个变量的体系。变量（特征），从技术层面是加工数据，而从业务层面实际在反应 RD 的业务理解和数据刻画业务能力。“二维图”，实际上未必是二维的，更重要的是我们需要把业务整个流程抽象成几个核心的维度，举几个例子：

外卖配送时间业务（维度甲：配送的环节，骑手到点、商家出餐、骑手配送、交付用户；维度乙：颗粒度，订单粒度、商家粒度、区域城市粒度；维度丙：配送类型，众包、自营……）。

反作弊变量体系（维度甲：作弊环节，登录、注册、实名、转账、交易、参与营销活动、改密……；维度乙：作弊介质，账户、设备、IP、WiFi、银行卡……）。

通过这些维度，你就可以展开一个“二维图”，把现有你可以想到的特征填上去，你一定会发现很多空白，比如下图，那么哪里还是特征设计的盲点就一目了然：

图 4 账户维度在转账、红包方面的特征很少；没有考虑 WiFi 这个媒介；客满与事件数据没考虑

数据和特征决定了模型性能的天花板。deep learning 当下在图像、语音、机器翻译、自动驾驶等领域非常火，但是 deep learning 在生物信息、基因学这个领域就不是热词：这背后是因为在前者，我们已经知道数据从哪里来，怎么采集，这些数据带来的信息基本满足了模型做非常准确的识别；而后者，即便有了上亿个人体碱基构成的基因编码，技术选型还是不能长驱直入——超高的数据采集成本，人后天的行为数据的获取壁垒等一系列的问题，注定当下这个阶段在生物信息领域，人工智能能发出的声音很微弱，更大的舞台留给了生物学、临床医学、统计学。

模型

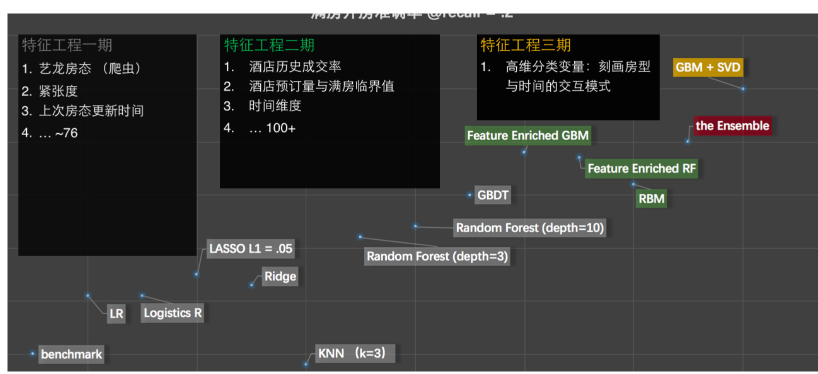


图 5 满房开房的技术选型、特征工程 roadmap

模型这件事儿，许多时候追求的不仅仅是准确率，通常还有业务这一层更大的约束。如果你在做一些需要强业务可解释的模型，比如定价和反作弊，那实在没必要上一个黑箱模型来为难业务。这时候，统计学习模型就很有用，这种情况下，比拼性能的话，我觉得下面这个不等式通常成立：Glmnet>LASSO>=Ridge>LR/Logistic。相比最基本的 LR/Logistic，ridge 通过正则化约束缓解了 LR 在过拟合方面的问题，lasso 更是通过 L1 约束做类似变量选择的工作。

不过两个算法的痛点是很难决定最优的约束强度，Glmnet 是 Stanford 给出的一套非常高效的解决方案。所以目前，我认为线性结构的模型，Glmnet 的痛点是最少的，而且在 R、Python、Spark 上面都开源了。

如果我们开发复杂模型，通常成立第二个不等式 RF（Random Forest，随机森林）<=GBDT<=XGBoost。拿数据说话，29 个 Kaggle 公开的 winner solution 里面，17 个使用了类似 GBDT 这样的 Boosting 框架，其次是 DNN（Deep Neural Network，深度神经网络），RF 的做法在 Kaggle 里面非常少见。

RF 和 GBDT 两个算法的雏形是 CART（Classification And Regression Trees），由 L Breiman 和 J Friedman 两位作者在 1984 年合作推出。但是在 90 年代在发展模型集成思想 the ensemble 的时候，两位作者代表着两个至今也很主流的派系：stacking/ Bagging & Boosting。

一种是把相互独立的 CART（randomized variables，bootstrap samples）水平铺开，一种是深耕的 Boosting，在拟合完整体后更有在局部长尾精细刻画的能力。同时，GBDT 模型相比 RF 更加简单，内存占用小，这都是业界喜欢的性质。XGBoost 在模型的轻量化和快速训练上又做了进一步的工作，也是目前我们比较喜欢尝试的模型。

作者简介

胡淏，美团算法工程师，毕业于哥伦比亚大学。先后在携程、支付宝、美团从事算法开发工作。了解风控、基因、旅游、即时物流相关问题的行业领先算法方案与流程。

人工智能在线特征系统中的生产调度

杨浩 伟彬

前言

在上篇博客《人工智能在线特征系统中的数据存取技术》中，我们围绕着在线特征系统存储与读取这两方面话题，针对具体场景介绍了一些通用技术，此外特征系统还有另一个重要话题：**特征生产调度**。本文将以美团点评酒旅在线特征系统为原型，介绍特征生产调度的架构演进及核心技术。架构演进共包含三个阶段，不同阶段面临的需求痛点和挑战各有不同，包括导入并发控制、特征变更原子切换、实时特征计算框架涉及、实时与离线调度融合等。本文我们将从业务需求角度出发，介绍系统演进的三个阶段所解决的主要问题和技术手段，然后把系统演化过程中的一些常见问题和解决方案抽象出来，放在特征生产技术章节统一讨论。

特征生产调度演进

从离线到在线

在线特征系统最核心的目标是将离线的特征数据通过在线服务的方式，提供给策略系统使用。在线特征系统的出现是为了实现如下的系统目标：

- 将离线的特征数据，以接口访问的形式提供给线上策略系统使用
- 特征数据每日更新一次
- 支撑的数据量在百亿级以上，可以水平扩展
- 每秒特征访问量峰值达到百万，平均响应延迟在 20ms 以内

从整体系统功能上来划分，在线特征系统需要做两件事情：第一，每日将离线更新的特征数据写入到存储引擎，这里我们选用分布式 KV（Key-Value）存储引擎 Tair 作为线上存储引擎，利用公司的 ETL 工具定期将离线数据写入到 Tair；第二，

提供接口服务，我们搭建了一个基于 Thrift 接口协议的 RPC 服务来对外提供特征读取服务。

由于不同特征集查询方式都相同，只是数据不同，因此在 Service 层我们把一组特征集合以及它的查询维度抽象成 Domain。举个例子，Domain=ABC 表示用户基础画像特征，包含性别、年龄、星座等特征，同时它又定义了查询维度为用户 ID。这样对于不同的特征集，只需要调用同一个接口，传入不同的 Domain 即可。

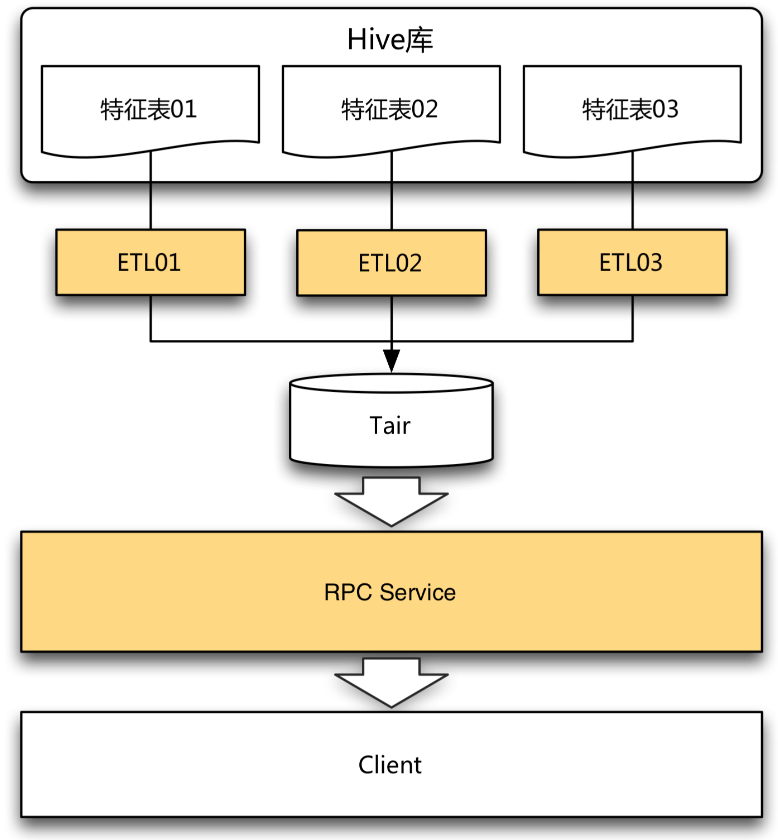


图 1　从离线到在线

在这一阶段，系统的重点是搭建一套特征导入、存储、读取的流程。我们利用公司提供的工具和组件迅速完成了任务。当有新的特征表需要接入时，开发一个导入 ETL，在服务端做相应的配置即可生效。同时，结构上的松散也带来很大的灵活性。

在业务发展初期，团队组织结构单一，需求量少，变化快，种类多，系统保持简单、松耦合，有助于灵活应对不断变化的需求。

从手动到自动

随着每日接入 Domain 数量的增加，接入新 Domain 工作显得繁琐而效率低下：每接入一个新的特征表，需要开发 ETL，而且 ETL 需要测试、上线、配置调度。因此，我们重新设计了数据导入的方案。

元数据驱动，平台化导入

ETL 工具需要开发数据导入脚本，它的灵活性相对较高，写出错的可能性也很大，测试和审核流程难以避免，新入职同学更是需要较大的学习成本。而对于特征导入这个需求，它的模式固化，可以抽取出以下元数据：

- 数据源信息：离线数据库、表名称等。
- 存储引擎信息：引擎类型、机房、IP 等。
- 存储格式信息：Key 字段、Value 字段等。
- 特征更新信息：更新周期、分区字段、分区方式等。

根据这些元数据，将导入流程都固化下来，可以进行平台化的统一调度。用户通过填写或选择少量的表单信息注册任务，出错的可能性大大降低，流程也可以从原来的写 ETL 代码、测试作业、配置调度、上线审核，简化成了填写表单和审核。接入流程从原来的几个小时，缩短到几分钟。同时，存储引擎从原来的仅支持 Tair，到现在 Squirrel（美团点评基于 Redis 的 KV 分布式存储中间件）等多引擎加入，系统调度架构如下。

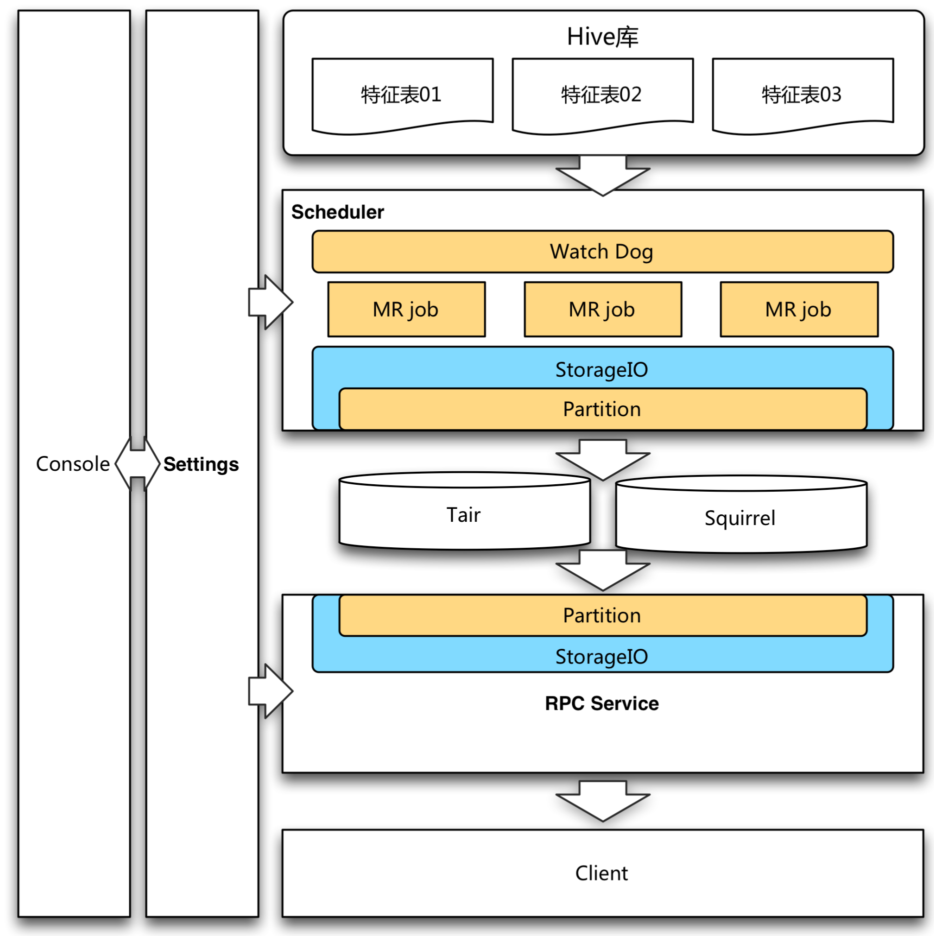


图 2　离线特征生产调度

- 控制台（Console）是元数据的入口，用户在这里完成表单的填写，元数据落入 Settings 模块的 MySQL 库中。
- 调度模块（Scheduler）从 Settings 模块读取元数据，每日扫描需要导入的 Hive 表，待当日离线数据生产完成，便会启动 Map Reduce Job 来执行导入工作。
- 接口服务（Service）接收来自客户端的请求，根据 Domain 名称从 Settings 库中加载 Domain 元数据，然后从存储引擎取到对应的特征信息。由

于调度模块与接口服务模块统一了元数据，因此新特征的接入可以实现服务端工作零成本，新上线的 Domain 可以直接从服务接口取到数据，无需任何人工操作。

阶段二的完成大大简化了离线特征的上线流程，使接入工作从几个小时缩短到几分钟，也降低了出错的可能性。导入平台化的实现，也为通用性优化功能提供了土壤：数据压缩功能使得内存、带宽资源得到了更充分的利用；多引擎存储功能满足了需求方对性能的不同要求；导入调度功能解决了更新流量峰值的问题，提高了系统的整体可用性。

从天级到秒级

迄今为止，原始特征数据都是离线的，且更新周期都是一天，这跟离线数据仓库的 T+1 模式有关。而很多关键的业务指标希望做到实时化，特征工程也是如此。用户近几分钟、近几秒的行为信息往往比很多离线特征更具有价值，实时特征必然会在策略系统中发挥越来越重要的作用。

参考离线特征的计算过程，离线大部分是利用了数据平台的 ETL 工具，它的输入输出都相对固定，只能落地到 Hive，用户大部分的精力只需要关心计算逻辑。因此从离线 Hive 导入到线上存储引擎，成为了特征系统的主要工作，无需操心特征计算。而目前公司没有很完备的、类似 Hive SQL 的计算框架支持实时特征计算，生产计算实时特征需要自己写流式处理作业。因此我们有必要提供一个专用、便捷的特征计算工具来支持常见特征的计算工作，利用简单配置完成实时特征计算。

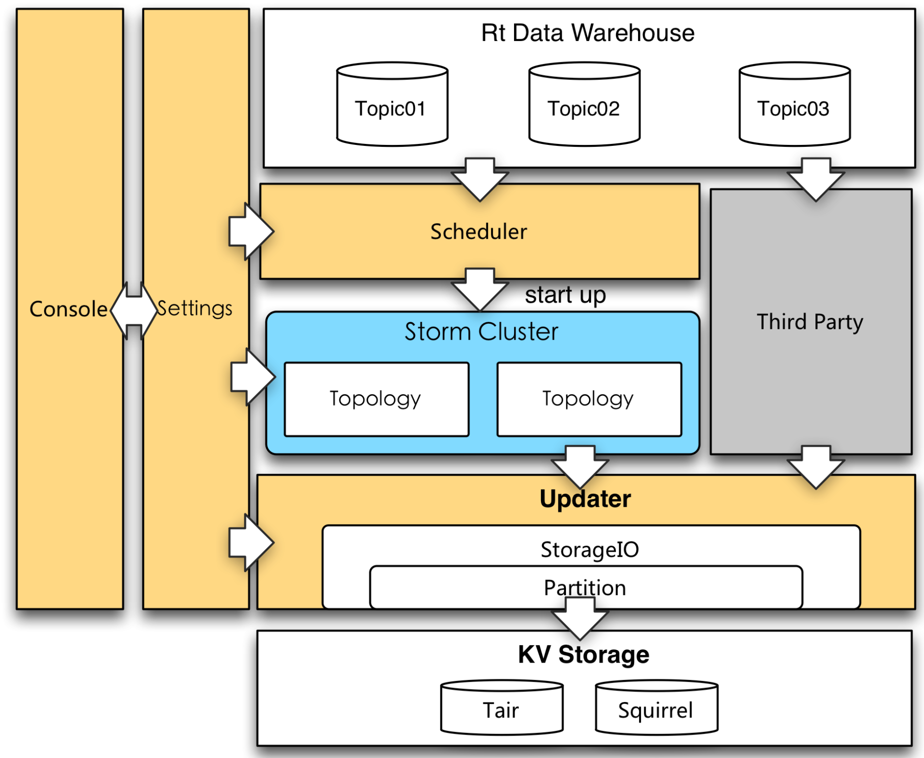


图 3 实时特征生产调度

实时部分的系统架构如上图所示，与离线类似，Console 部分接受用户的表单配置并将元数据写入 Settings 持久化。Scheduler 会负责读取 Settings 的元数据信息调度实时特征生产任务。我们采用 Storm 流式服务计算实时特征，从实时数据仓库的 Kafka Topic 接收流式数据，并按照预先配置好的特征计算逻辑生产、计算实时特征，然后写入到线上存储引擎。

下面详细讨论一下我们对于实时特征计算的平台化以及优化方案。

实时特征计算平台化

算法使用的特征有繁有简，复杂多变，设计一个自动化的实时特征计算系统难度很大。回到业务需求，我们的目的是通过特征生产系统来简化开发工作量，而非完全取代特征开发；因此我们选择一部分常见的实时特征类型，实现自动化生产和导入。

对于更复杂的实时特征，提供了更新接口来支持第三方特征生产程序对接。

以下是系统支持配置化生产的特征类型。首先是不同的**时间跨度**分类：

- 固定时间窗口，时间窗口的起止时间点是固定的，比如某日的销售额。
- 滑动时间窗口，时间窗口的长度是固定的，但起止时间点一直在向前滚动，比如近 2 小时销售额。
- 无限时间窗口，时间窗口的起点是固定的，但终止时间点一直在向前滚动，比如商家历史上销售总额。

销售额这个指标其实是对订单金额做求和（SUM）操作，总结常见的**计算类型**有如下几种：

- 求和（SUM），如销售额。
- 计数（COUNT），如订单量。
- 最大值（MAX），如最大订单金额。
- 最小值（MIN），如最小订单金额。
- 平均数（AVG），如平均订单金额。
- 去重计数（DISTINCT COUNT），如页面的用户浏览量（同一个用户多次浏览算一次）。
- 最新值（LAST），如最后支付时间。
- 列表（LIST），如最近的支付用户 ID 列表。

以上时间窗口与指标的组合，一共支持 24 种常见特征的计算类型。

对于实现上述特征的计算，主要包含如下三个抽象步骤：

1. 读取相关的数据（如上次特征值，或一些中间结果）。
2. 根据收到的业务数据，以及步骤 1 取到的数据进行计算（如累加或求去重数），得到新的特征值（和中间结果）。
3. 将特征（和中间结果）更新到系统。

不同时间窗口的实现方式应该尽量跟**计算类型**解耦，可以抽象出各自的处理方式：

1. 固定时间窗口，这类特征应该将时间窗的标识放在特征的 Key 当中。例如某商户某日销售额这个特征，将 Key 设置成 ${ 商户 ID}_${ 日期 }，这样可以实现时间窗的自然滚动。
2. 滑动时间窗口，常见的做法是缓存时间窗内的所有明细数据作为中间结果，当新的明细数据到来时，删除时间窗内过期的明细数据，并利用缓存的明细数据重新计算特征值。但这种实现方式缺点是当滑动时间窗的跨度较大时，需要缓存大量中间结果，可能成为系统瓶颈。对于这个问题，我们采用了延迟队列的实现方式。

> **延迟队列实现滑动时间窗，当新的明细数据到来时，会直接累计到特征值，同时将明细数据发送到延迟队列。延迟队列的作用是可以将数据延迟指定时间后重新发送回系统。系统接收到延迟消息时，再从特征值中抵消该部分数据（例如计算近 2 小时销售额，收到订单数据后累加销售额，收到延迟订单消息则减去销售额），这样可以只保留特征值，无需缓存明细数据即可实现窗口滑动的逻辑。延迟队列的实现方式只适用于可抵消的计算类型，如求和、计数等，但像最大值、最小值、去重计数等无法满足**

3. 无限时间窗口，简单粗暴的方式是回溯所有历史消息即可。然而这样存在的问题是，第一，流式实时数据本身一般不会持久化保留太长的时间（通常是几天）；第二，这种方式太耗费资源，特征的每一次更新都涉及多次 RPC。较为合适的办法是离线数据计算特征的基准值，实时数据基于离线计算结束的时间点继续累积。详细过程参考下文**数据融合**与**数据恢复**。

为了保证数据可靠性与查询效率，中间结果和特征都存放在分布式 Key-Value 存储引擎中。下图是 Storm 计算框架的拓扑逻辑图，其中 Calc Bolt 承担着不同计算

类型的实现，而 Mafka Delay Topic 则是延迟队列组件，用于实现滑动时间窗口。

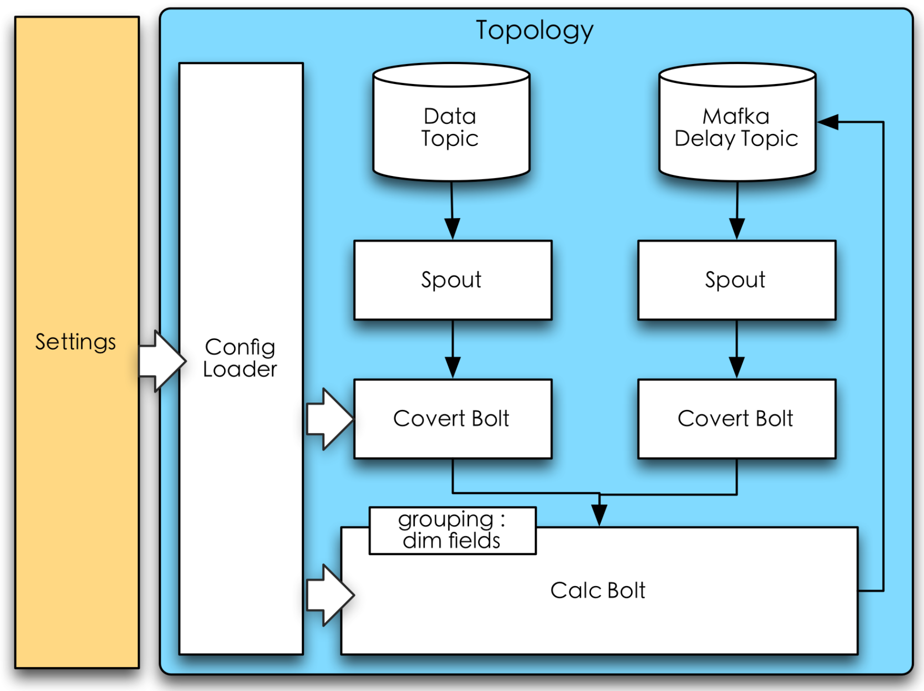
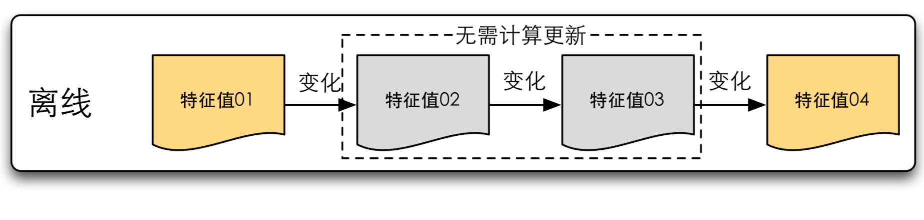


图 4 实时计算框架

上述 24 个特征是常见的一些实时统计类特征，开发者只需要填写表单，选择需要的特征类型即可完成特征开发工作。对于现阶段不支持配置实现的个性化、计算逻辑复杂的特征，开发者可以自己开发 Storm 拓扑实现计算逻辑（对应实时特征生产调度图中灰色的 Third Party 模块），并通过更新接口写入到线上存储引擎。

实时特征计算优化

从上述支持的特征列表中可以看出，实时计算框架目前只支持聚合、明细列表这样的简单特征。即便如此，实时特征计算还是面临很大的挑战。离线特征只需要计算出更新周期内特征的最终值即可，而实时特征需要把每次特征变化都要实时计算出来，它既要计算的快，又要计算的多，因此它无法支持很大量的数据。



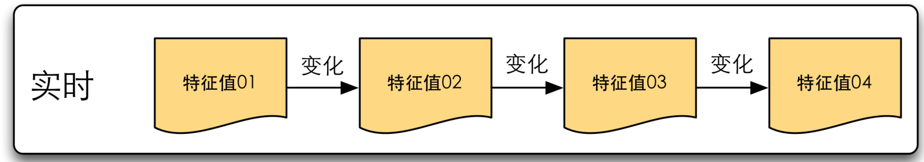


图 5　实时特征与离线特征对比

当面临数据计算量的挑战时，优化思路之一是利用一些中间结果或上次计算结果简化计算量，化全量计算为增量计算。例如求平均数这种特征，你可以存住所有的明细数据，当新的一条明细数据加入进来时，将所有明细数据求和再除以总数。这样需要 O(N) 的时间和空间复杂度，N 是明细数据个数。而你也可以仅保留总和跟总数，每次更新只要做一次加法和除法即可。

另一种优化思路是利用近似计算。比如求去重数（DISTINCT COUNT）这种指标，要精确计算可能很难找到一个时空复杂度都比较低的方案，而如果可以忍受近似计算的误差，HyperLogLog 算法是一个不错的选择。

特征生产调度技术

在生产调度演进过程中，会不断遇到各种系统问题，如可靠性、一致性、性能等等。在这一章节我们把特征生产调度中一些常见的技术手段，以及常见问题的解决方案汇总起来呈现给大家。

逻辑存储层

逻辑存储层的含义是 Domain 的元数据并不直接存放与存储相关的信息，而是将这些信息抽象成 Storage 元数据，如下图所示。其中 Domain 存储了访问控制、离线源信息、Storage ID 等信息，而 Storage 则存储了存储介质、特征元数据、数据存储格式等与存储相关的信息。Domain 与 Storage 是一对多的关系。

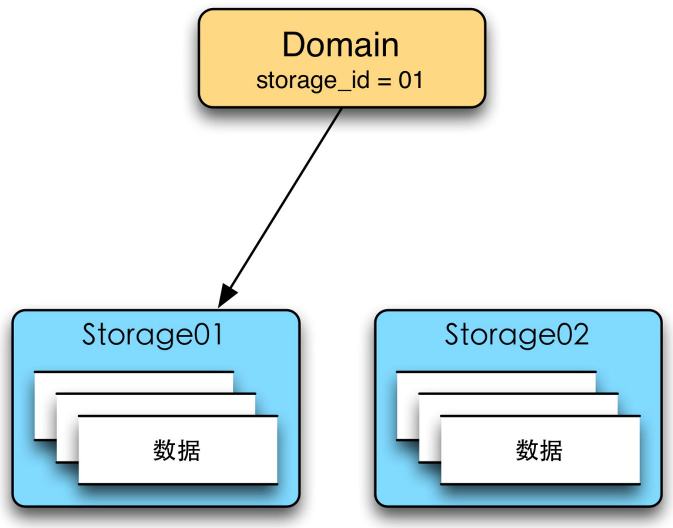


图 6　逻辑存储层

抽象存储层 Storage 有很多好处：

1. 支持数据版本灵活切换。一个 Domain 可能存在多个 Storage 数据版本，而只有一个是生效的。由于线上存储着多份版本数据，Domain 与 Storage 的对应关系自由切换，从而可以实现不同数据版本的自由变更。
2. 可以做到读写分离。这个特性对离线特征更新是一个好消息，因为离线特征对时效性要求不高，因此可以做到读的 Storage 跟写的 Storage 不同，在合适的时机切换读取的 Storage。这样切换表结构、更换存储引擎都可以平滑完成，而对读取方做到完全透明。
3. 实现数据切换的原子性。一次数据导入从 Domain 上看并不是原子操作（更新一个 Key-Value 对是原子操作，但是整个离线表导入到 KV 存储引擎并不是原子的），Storage 的引入可以实现 Domain 导入的原子性，当数据格式、特征元数据发生变化时可以保证数据读取的一致性。

增量更新与数据一致性

对于每日的离线特征更新，我们发现有些虽然总数据量庞大，但每天的变化比较

少。比如用户画像，有很多沉睡的用户他的特征基本不发生变化。如果每天将全量数据刷到线上，其实做了很多无谓的更新操作，对系统资源是一个巨大的浪费。尤其是更新线上存储引擎，写入压力将导致在线服务稳定性的波动。因此考虑在更新前计算出特征的增量变化数据，只更新变化的部分。而计算增量数据需要有线上特征集合的完整离线数据备份——数据镜像。

> 数据镜像（SNAPSHOT）是对线上存储引擎数据的离线备份。由于 KV 存储的特点适用于随机访问，而对顺序访问（如遍历）的支持并不是其强项，因此通过构造离线数据镜像，可以一定程度上帮助我们更为方便的操作线上 KV 存储引擎中的数据。这里主要是为了支持增量更新和数据恢复功能。

如下图所示同一个更新周期（Period）内需要做两次数据处理流程：归档（Archive）和同步（Sync）。Archive 会将上一个更新周期的 SNAPSHOT 和这个更新周期的特征数据表做差集和并集。差集的结果是增量数据（Diff），并集的结果是该更新周期内的 SNAPSHOT。对于数据量大而 Diff 又少的特征集合来说，增量更新会极大的节约线上的资源。

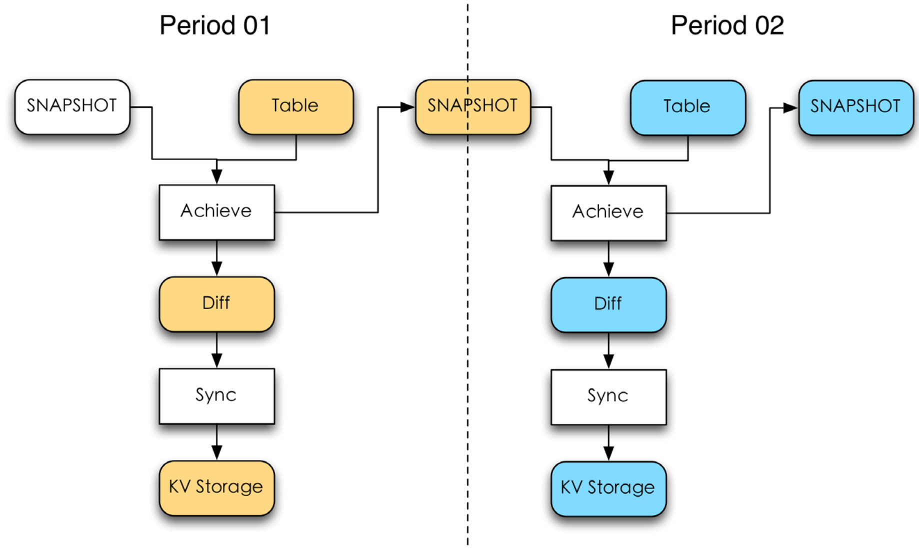


图 7 离线增量更新流程

增量更新可能带来数据一致性的问题。如果 Sync 步骤出现了少量数据更新失败（比如写操作偶然性超时），会导致 SNAPSHOT 与 KV 存储引擎的数据不一致。这种问题在全量更新时并不是什么大问题，当数据在后续更新周期内全量写入时，可以认为总会修复上次的更新失败问题。然而在增量更新时，这种错误是永久性的。因此我们在生成 SNAPSHOT 时为每条数据附上一条租约（Lease），当租约到期时，强制将该条数据加入 Diff 参与当次更新，这样可以保证数据的最终一致性。Lease 的时间我们可以对每一条数据进行随机分布，这样需要更新的数据会平稳的分布到每一天而不出现明显尖峰。Lease 机制其实是全量更新到增量更新的一个平滑过渡，Lease 为 0 时是全量更新，Lease 为无穷大时是增量更新。

写入削峰

随着离线特征表增多，同一时刻进行数据导入的作业相互抢占资源，未加控制的写入速度影响了 KV 存储引擎的正常读取，甚至引起雪崩。实时特征也面临类似问题，实时数据流容易随着集群的状况、业务的特点出现流量峰谷，如果没有消费速度的限制，很容易导致存储引擎压力突增突减，甚至将其打垮。

离线与实时通过不同手段控制并发写入线上存储速度。离线更新的特点是：

1. 更新具有周期性，需要同步时流量很大，同步结束后流量变为 0
2. 对更新延迟性要求不高（往往在小时级别）
3. 写入方完全是特征系统内部模块（每个 Sync 作业）

我们的目标是尽快将这些数据同步到线上存储引擎，同时兼顾写入速度（影响更新延迟）和集群资源（线上存储压力）。鉴于离线更新的特点，且 Sync 作业本来就由调度器管理，因此很容易将并发控制实现在调度器内部。调度器会控制每个存储引擎的最大 Sync 作业并发数量，同时每个 Sync 作业内部并发的写入速度也是固定的。负载限制的关系如下：

> 同步中的作业数 * 作业内部并发度 ≤ 线上存储引擎的最大写入压力

而实时特征更新的特点是：

1. 每时每刻都有写入的流量
2. 流量随着业务时间变化会有波动
3. 对更新延迟要求较高（往往在秒级）
4. 写入方有特征系统内部模块，也有第三方的服务

由于写入方可能来自特征系统外部，难以统一控制写入方速度，因此我们没有像离线一样让写入方直接操作线上存储，而是在两者之间增加了一个 Updater 服务（参考图 5. 实时特征生产调度），由它控制每个写入方的速度。实时特征流量波动大，且对更新延迟要求高，新接入的实时特征需要预估流量峰值并配置到 Updater 服务中。对于超过预设流量的请求予以拒绝或延迟。

原子更新

离线特征与实时特征面临的原子更新问题各有不同。离线更新的粒度为天级别，所有特征一天只更新一次，有的特征集合希望保证天级别的更新是原子的。即不希望任意时刻出现一部分特征是昨天的值，一部分特征是今天的值。这个问题利用上文提到的**逻辑存储层**可以很好的解决，这里不再赘述。

然而实时特征生产更新却面临另一种问题。很多时候需要先读取特征当前值，然后基于当前值做计算得到新值写入 KV 存储引擎，一次更新过程涉及到读取，计算、写入三步。因此如果要保证数据更新的一致性，必须要保证一次更新的读、算、写操作的原子性或者事务性。对于原子更新的需求主要有两类解决方案：

1. 生产方通过数据分组，保证相同 Key 的数据只通过一个线程更新，系统配置化生产的特征都基于 Storm 计算框架，实现起来非常方便。
2. 如果一些第三方（Third Party）不方便做数据分组，我们通过系统内 Updater 服务提供的 CAS（Compare And Swap）接口，生产方调用 CAS 接口进行更新，同样也可以做到原子更新。

数据融合与数据恢复

如果说实时数据是离线数据的延伸，那么离线数据可以说是实时数据的备份，二者是相辅相成的。理论上，利用实时数据可以计算出所有想要的特征，但离线数据可以从不同方面解决实时特征计算中诸多棘手问题：

- 提升效率。可以利用离线计算来提升效率。例如计算每个商家有史以来的营业额，如果全部采用实时数据，那将要实时回放历史上所有订单数据，这样的数据量和计算代价都是巨大的，此时可以利用离线框架计算出历史营业额，在特征初始化时将离线计算好的历史营业额导入线上存储引擎。之后的特征计算更新依赖实时框架，这样可以节省系统开销。
- 提升可靠性。可以利用离线计算和导入校正实时更新可能产生的误差，提升数据可靠性。实时特征计算采用 Storm 框架，可以保证数据记录不漏（At Least Once），但不能保证不重（At Most Once）。从系统设计的角度看，对于实时流式处理要做到确保计算一次（Exactly Once）的代价往往很高，相比于让流式计算绝对可靠，与离线计算结果融合往往是更合适的选择。对于像每日营业额这种固定时间窗的特征，实时更新流程只会更新当前时间窗内的特征（今日营业额），而并不会改动历史时间窗的数据，因此历史时间窗的特征可以利用离线数据重新校正一次，这样可以保证数据的最终正确性。

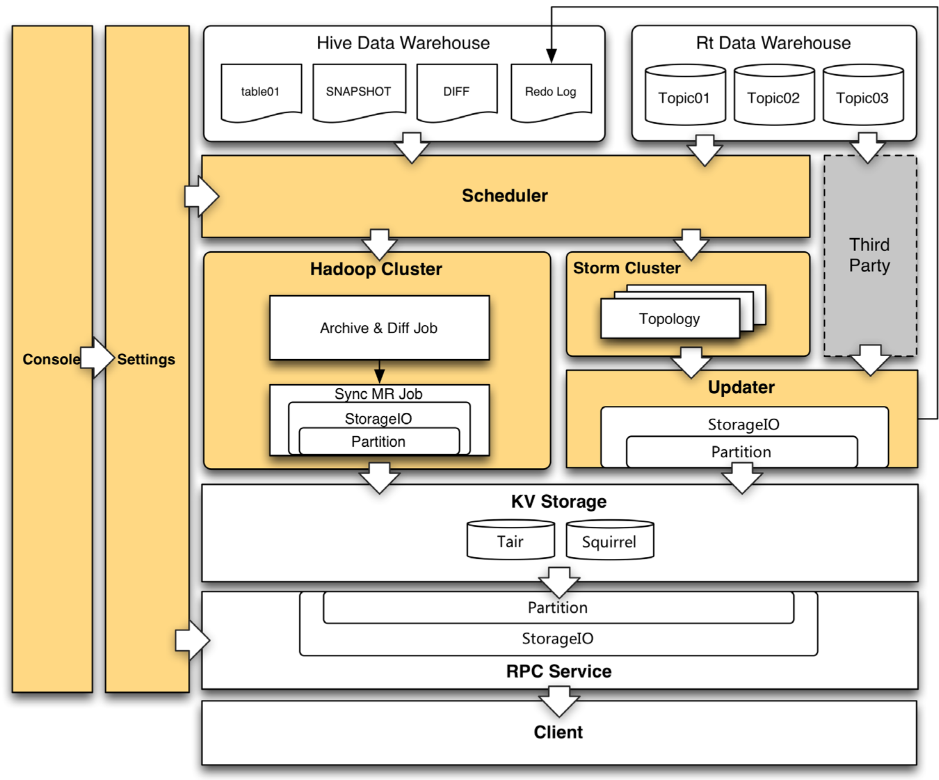


图 8　离线实时特征生产整体架构

上图为离线实时特征生产的整体架构。离线与实时的数据融合，需要一个更强大的调度器，它负责协调离线任务与实时任务的关系，高效、可靠的完成数据导入工作。离线作业与实时作业的调度关系分为两种：

1. 离线只初始化一次，后续只有实时数据从基于离线初始化的值做累积运算。如下图的离线初始化。这种调度类型常见于无限时间窗口的一些计算指标，如商户最后一次订单时间，用户累积消费金额等。
2. 离线与实时作业并存，离线作业定期复写历史数据，实时作业更新最近数据。如下图的离线定期修复。这种调度类型常见于提升固定时间窗特征的可靠性，如商户每日营业额等，这类特征在 Key 中携带时间信息，特征数据天然按时间窗分区，离线与实时作业更新不同分区的数据而互不影响。

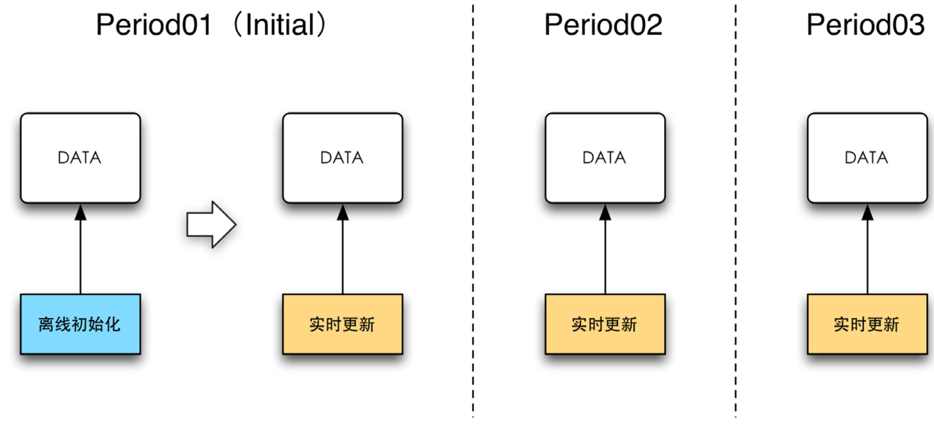


图 9　离线初始化

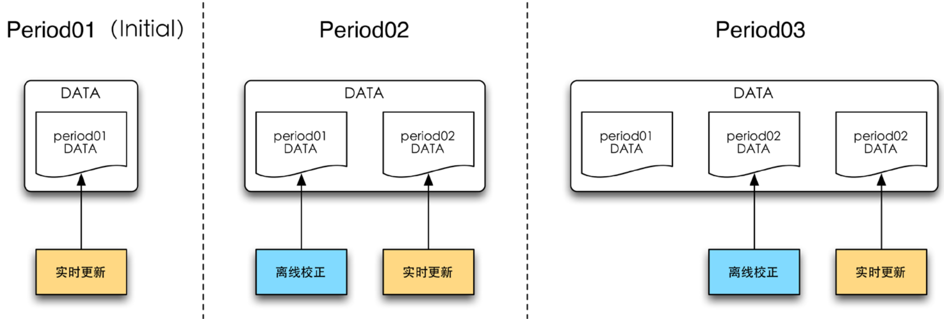


图 10　离线定期修复

数据恢复是指当线上数据发生问题的时候（可能由于数据源问题、线上故障、硬件故障等）如何修复线上数据，使其恢复到正常状态。数据恢复是效率和可靠性的双重考验，越快速的恢复到正常状态，系统的可靠性就越高。离线增量更新的特征与实时特征都是在原有特征基础上累积计算，一旦某一时刻数据出现问题需要重导数据，只能从第一次增量开始重新累积，这无疑是及其低效的。如果能够定期备份线上特征的数据镜像，当实时更新从某一时刻出现故障时，可以用最近一次正确的离线 SNAPSHOT 版本刷新数据。离线数据最新的 SNAPSHOT 应与线上特征数据保持一致，而实时特征的 SNAPSHOT 会有一定延迟，这时只要将上游实时流数据回退到 SNAPSHOT 时间点重新开始消费（如下图所示），这样相比没有 SNAPSHOT 可以较为快速的恢复故障。

数据恢复功能是离线与实时架构融合的产物，只不过它的离线数据不是业务上产出的某张离线表，而是离线镜像数据 SNAPSHOT。

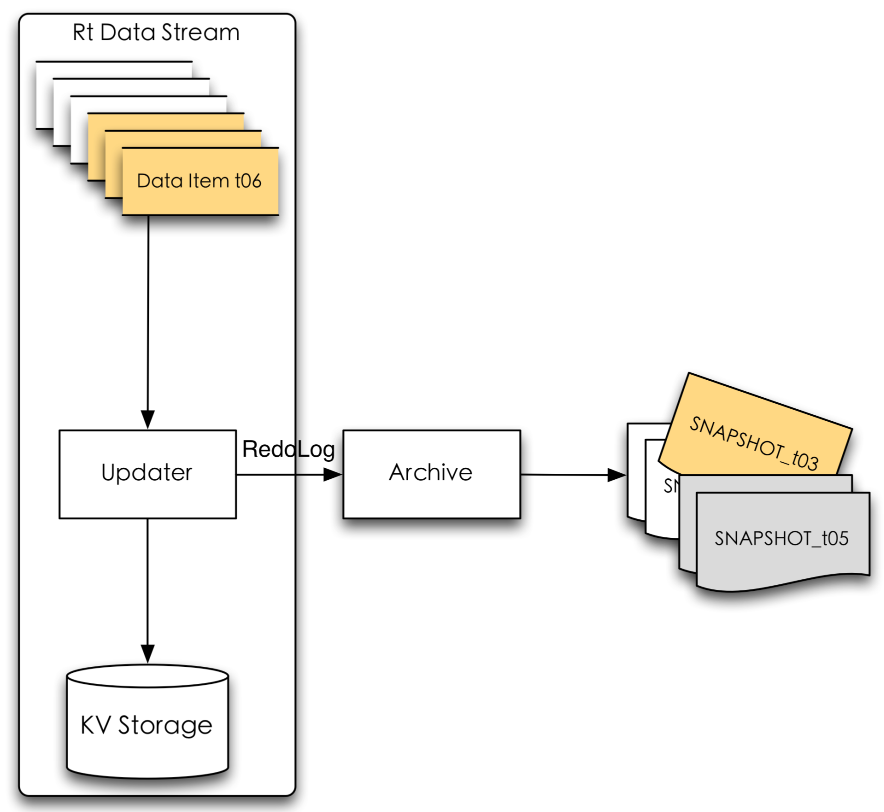


图 11 数据恢复

后记

一个完整的在线特征系统数据流涵盖加载、计算、导入、存储、读取五个步骤。从两类数据的五个步骤来看，在线特征系统截至目前还并不完整；而深入到每一个步骤，还有很多功能特性需要继续完善：支持离线计算框架、支持更多的实时计算类型、实时特征计算高可用、缩短数据恢复时间、特征实时监控等等。在与其他团队交流时，也有将特征系统深入到策略系统内部，实现算法、特征迭代一体化流程。在线特征系统的工作仍任重而道远。

能力所限，难免管中窥豹，挂一漏万。欢迎感兴趣同学一起交流。

作者简介

杨浩，美团平台及酒旅事业群数据挖掘系统负责人，2011 年毕业于北京大学，曾担任 107 间联合创始人兼 CTO，2016 年加入美团点评。长期致力于计算广告、搜索推荐、数据挖掘等系统架构方向。
伟彬，美团平台及酒旅事业群数据挖掘系统工程师，2015 年毕业于大连理工大学，同年加入美团点评，专注于大数据处理技术与高并发服务。

人工智能在线特征系统中的数据存取技术

杨浩　伟彬

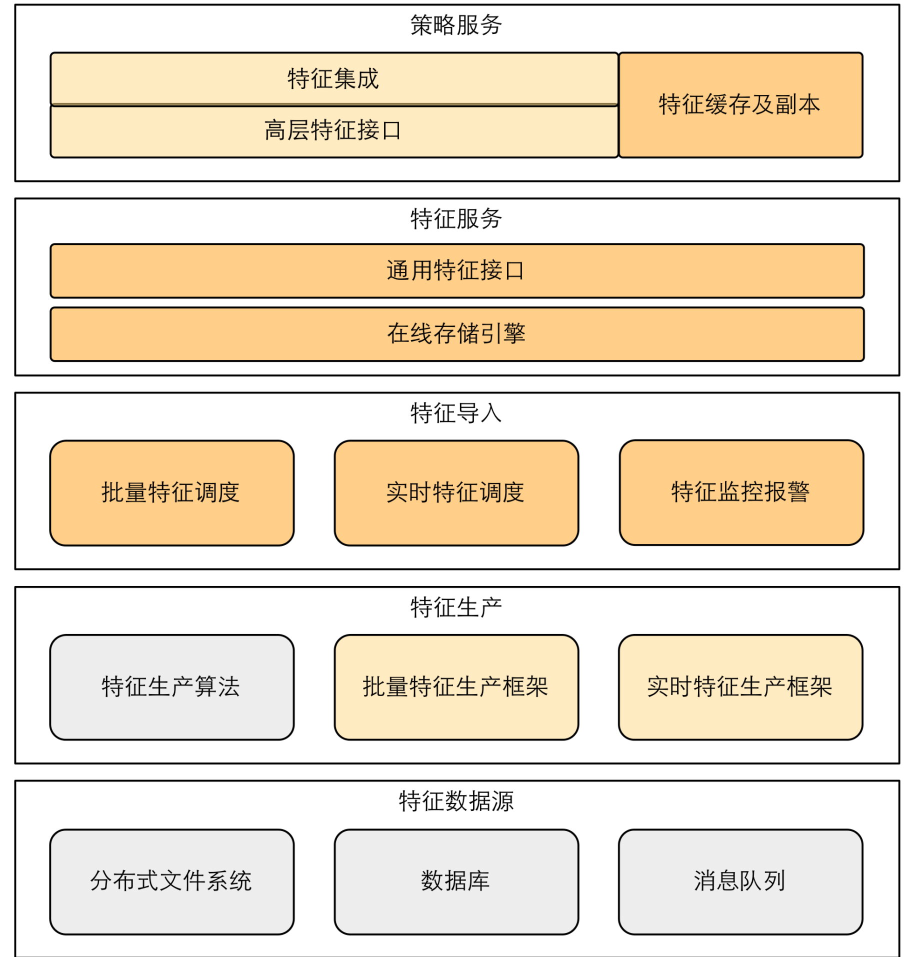
一、在线特征系统

主流互联网产品中，不论是经典的计算广告、搜索、推荐，还是垂直领域的路径规划、司机派单、物料智能设计，建立在人工智能技术之上的策略系统已经深入到了产品功能的方方面面。相应的，每一个策略系统都离不开大量的在线特征，来支撑模型算法或人工规则对请求的精准响应，因此特征系统成为了支持线上策略系统的重要支柱。美团点评技术博客之前推出了多篇关于特征系统的文章，如《机器学习中的数据清洗与特征处理综述》侧重于介绍特征生产过程中的离线数据清洗、挖掘方法，《业务赋能利器之外卖特征档案》侧重于用不同的存储引擎解决不同的特征数据查询需求。而《外卖排序系统特征生产框架》侧重介绍了特征计算、数据同步和线上查询的特征生产 Pipeline。

本文以美团酒旅在线特征系统为原型，重点从线上数据存取角度介绍一些实践中的通用技术点，以解决在线特征系统在高并发情形下面临的问题。

1.1　在线特征系统框架——生产、调度、服务一体化

在线特征系统就是通过系统上下文，获得相关特征数据的在线服务。其功能可以是一个简单的 Key-Value（KV）型存储，对线上提供特征查询服务，也可以辐射到通用特征生产、统一特征调度、实时特征监控等全套特征服务体系。可以说，几个人日就可以完成一个简单能用的特征系统，但在复杂的业务场景中，把这件事做得更方便、快速和稳定，却需要一个团队长期的积累。



以上结构图为一体化特征系统的概貌，自底向上为数据流动的方向，各部分的功能如下：

- **数据源：**用于计算特征的原始数据。根据业务需求，数据来源可能是分布式文件系统（如 Hive），关系型数据库（如 MySQL），消息队列（如 Kafka）等。
- **特征生产：**该部分负责从各种数据源读取数据，提供计算框架用于生产特征。生产框架需要根据数据源的类型、不同的计算需求综合设计，因此会有多套生产框架。
- **特征导入：**该部分负责将计算好的特征写入到线上存储供特征服务读取。该部

分主要关注导入作业之间的依赖、并发写入的速度与一致性等问题。

- **特征服务：**该部分为整个特征系统的核心功能部分，提供在线特征的存取服务，直接服务于上层策略系统。

特征的生命周期按照上述过程，可以抽象为五个步骤：读、算、写、存、取。整个流程于特征系统框架内成为一个整体，作为特征工程的一体化解决方案。本文主要围绕特征服务的核心功能“**存**”、“**取**”，介绍一些通用的实践经验。特征系统的延伸部分，如特征生产、系统框架等主题会在后续文章中做详细介绍。

1.2 特征系统的核心——存与取

简单来说，可以认为特征系统的核心功能是一个大号的 HashMap，用于存储和快速提取每次请求中相关维度的特征集合。然而实际情况并不像 HashMap 那样简单，以我们的通用在线特征系统（Datahub）的系统指标为例，它的核心功能主要需面对**存储**与**读取**方面的挑战：

1. **高并发：**策略系统面向用户端，服务端峰值 QPS 超过 1 万，数据库峰值 QPS 超过 100 万（批量请求造成）。
2. **高吞吐：**每次请求可能包含上千维特征，网络 IO 高。服务端网络出口流量均值 500Mbps，峰值为 1.5Gbps。
3. **大数据：**虽然线上需要使用的特征数据不会像离线 Hive 库那样庞大，但数据条数也会超过 10 亿，字节量会达到 TB 级。
4. **低延迟：**面对用户的请求，为保持用户体验，接口的延迟要尽可能低，服务端 TP99 指标需要在 10ms 以下。

以上指标数字仅是以我们系统作为参考，实际各个部门、公司的特征系统规模可能差别很大，但无论一个特征系统的规模怎样，其系统核心目标必定是考虑：高并发、高吞吐、大数据、低延迟，只不过各有不同的优先级罢了。当系统的优化方向是多目标时，我们不可能独立的用任何一种方式，在有限资源的情况下做到面面俱到。留给我们的是业务最重要的需求特性，以及对应这些特性的解决方案。

二、在线特征存取技术

本节介绍一些在线特征系统上常用的存取技术点，以丰富我们的武器库。主要内容也并非详细的系统设计，而是一些常见问题的通用技术解决方案。但如上节所说，如何根据策略需求，利用合适的技术，制定对应的方案，才是各位架构师的核心价值所在。

2.1 数据分层

特征总数据量达到 TB 级后，单一的存储介质已经很难支撑完整的业务需求了。高性能的在线服务内存或缓存在数据量上成了杯水车薪，分布式 KV 存储能提供更大的存储空间但在某些场景又不够快。开源的分布式 KV 存储或缓存方案很多，比如我们用到的就有 Redis/Memcache，HBase，Tair 等，这些开源方案有大量的贡献者在为它们的功能、性能做出不断努力，本文就不更多着墨了。

对构建一个在线特征系统而言，实际上我们需要理解的是我们的特征数据是怎样的。有的数据非常热，我们通过内存副本或者是缓存能够以极小的内存代价覆盖大量的请求。有的数据不热，但是一旦访问要求稳定而快速的响应速度，这时基于全内存的分布式存储方案就是不错的选择。对于数据量级非常大，或者增长非常快的数据，我们需要选择有磁盘兜底的存储方案——其中又要根据各类不同的读写分布，来选择存储技术。

当业务发展到一定层次后，单一的特征类型将很难覆盖所有的业务需求。所以在存储方案选型上，需要根据特征类型进行数据分层。分层之后，不同的存储引擎统一对策略服务提供特征数据，这是保持系统性能和功能兼得的最佳实践。

2.2 数据压缩

海量的离线特征加载到线上系统并在系统间流转，对内存、网络带宽等资源都是不小的开销。数据压缩是典型的以时间换空间的例子，往往能够成倍减少空间占用，对于线上珍贵的内存、带宽资源来说是莫大的福音。数据压缩本质思想是减少信息冗余，针对特征系统这个应用场景，我们积累了一些实践经验与大家分享。

2.2.1 存储格式

特征数据简单来说即特征名与特征值。以用户画像为例，一个用户有年龄、性别、爱好等特征。存储这样的特征数据通常来说有下面几种方式：

1. JSON 格式，完整保留特征名 – 特征值对，以 JSON 字符串的形式表示。
2. 元数据抽取，如 Hive 一样，特征名（元数据）单独保存，特征数据以 String 格式的特征值列表表示。
3. 元数据固化，同样将元数据单独保存，但是采用强类型定义每个特征，如 Integer、Double 等而非统一的 String 类型。

三种格式各有优劣：

1. JSON 格式的优点在特征数量可以是变长的。以用户画像为例，A 用户可能有年龄、性别标签。B 用户可以有籍贯、爱好标签。不同用户标签种类可以差别很大，都能便捷的存储。但缺点是每组特征都要存储特征名，当特征种类同构性很高时，会包含大量冗余信息。
2. 元数据抽取的特点与 JSON 格式相反，它只保留特征值本身，特征名作为元数据单独存放，这样减少了冗余特征名的存储，但缺点是数据格式必须是同构的，而且如果需要增删特征，需要更改元数据后刷新整个数据集。
3. 元数据固化的优点与元数据抽取相同，而且更加节省空间。然而其存取过程需要实现专有序列化，实现难度和读写速度都有成本。

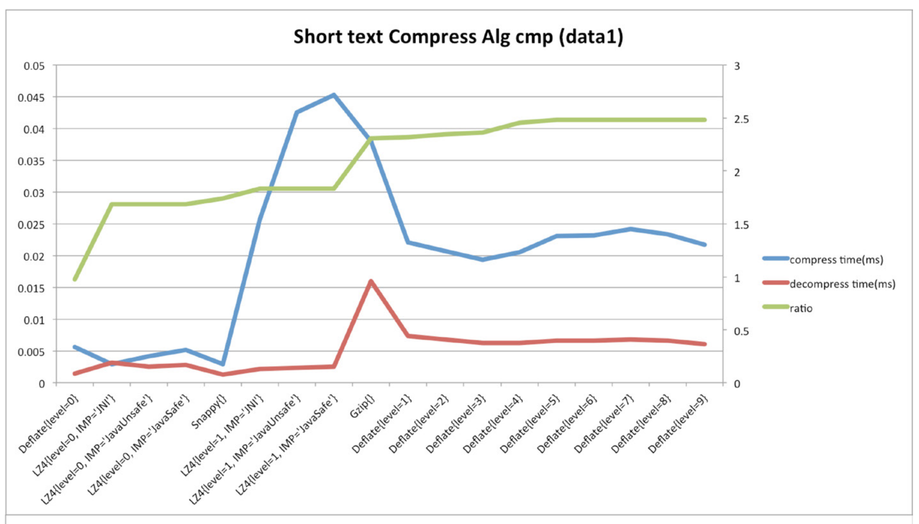
特征系统中，一批特征数据通常来说是完全同构的，同时为了应对高并发下的批量请求，我们在实践中采用了元数据抽取作为存储方案，相比 JSON 格式，有 2~10 倍的空间节约（具体比例取决于特征名的长度、特征个数以及特征值的类型）。

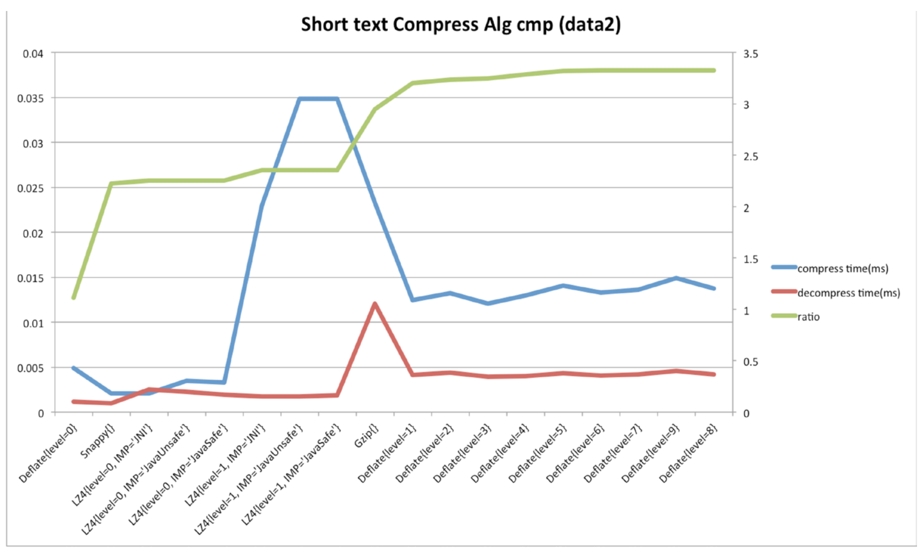
2.2.2 字节压缩

提到数据压缩，很容易就会想到利用无损字节压缩算法。无损压缩的主要思路是将频繁出现的模式（Pattern）用较短的字节码表示。考虑到在线特征系统的读写模式是一次全量写入，多次逐条读取，因此压缩需要针对单条数据，而非全局压缩。目前主流的 Java 实现的短文本压缩算法有 Gzip、Snappy、Deflate、LZ4 等，我们做了两组实验，主要从单条平均压缩速度、单条平均解压速度、压缩率三个指标来对比以上各个算法。

数据集：我们选取了 2 份线上真实的特征数据集，分别取 10 万条特征记录。记录为纯文本格式，平均长度为 300~400 字符（600~800 字节）。

压缩算法：Deflate 算法有 1~9 个压缩级别，级别越高，压缩比越大，操作所需要的时间也越长。而 LZ4 算法有两个压缩级别，我们用 0，1 表示。除此之外，LZ4 有不同的实现版本：JNI、Java Unsafe、Java Safe，详细区别参考 https://github.com/lz4/lz4-java，这里不做过多解释。





实验结果图中的毫秒时间为单条记录的压缩或解压缩时间。压缩比的计算方式为压缩前字节码长度 / 压缩后字节码长度。可以看出，所有压缩算法的压缩 / 解压时间都会随着压缩比的上升而整体呈上升趋势。其中 LZ4 的 Java Unsafe、Java Safe 版由于考虑平台兼容性问题，出现了明显的速度异常。

从使用场景（一次全量写入，多次逐条读取）出发，特征系统主要的服务指标是特征高并发下的响应时间与特征数据存储效率。因此特征压缩关注的指标其实是：快速的解压速度与较高的压缩比，而对压缩速度其实要求不高。因此综合上述实验中各个算法的表现，Snappy 是较为合适我们的需求。

2.2.3 字典压缩

压缩的本质是利用共性，在不影响信息量的情况下进行重新编码，以缩减空间占用。上节中的字节压缩是单行压缩，因此只能运用到同一条记录中的共性，而无法顾及全局共性。举个例子：假设某个用户维度特征所有用户的特征值是完全一样的，字节压缩逐条压缩不能节省任何的存储空间，而我们却知道实际上只有一个重复的值在反复出现。即便是单条记录内部，由于压缩算法窗口大小的限制，长 Pattern 也很难被顾及到。因此，对全局的特征值做一次字典统计，自动或人工的将频繁 Pattern 加入到字典并重新编码，能够解决短文本字节压缩的局限性。

2.3 数据同步

当每次请求，策略计算需要大量的特征数据时（比如一次请求上千条的广告商特征），我们需要非常强悍的在线数据获取能力。而在存储特征的不同方法中，访问本地内存毫无疑问是性能最佳的解决方式。想要在本地内存中访问到特征数据，通常我们有两种有效手段：**内存副本**和**客户端缓存**。

2.3.1 内存副本技术

当数据总量不大时，策略使用方可以在本地完全镜像一份特征数据，这份镜像叫内存副本。使用内存副本和使用本地的数据完全一致，使用者无需关心远端数据源的存在。内存副本需要和数据源通过某些协议进行同步更新，这类同步技术称为内存副本技术。在线特征系统的场景中，数据源可以抽象为一个 KV 类型的数据集，内存副

本技术需要把这样一个数据集完整的同步到内存副本中。

推拉结合——时效性和一致性

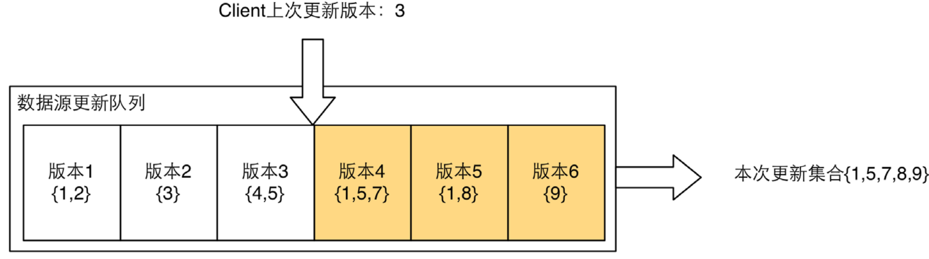
一般来说，数据同步为两种类型：**推**（Push）和**拉**（Pull）。Push 的技术比较简单，依赖目前常见的消息队列中间件，可以根据需求做到将一个数据变化传送到一个内存副本中。但是，即使实现了不重不漏的高可靠性消息队列通知（通常代价很大），也还面临着初始化启动时批量数据同步的问题——所以，Push 只能作为一种提高内存副本时效性的手段，本质上内存副本同步还得依赖 Pull 协议。Pull 类的同步协议有一个非常好的特性就是幂等，一次失败或成功的同步不会影响下一次进行新的同步。

Pull 协议有非常多的选择，最简单的每次将所有数据全量拉走就是一种基础协议。但是在业务需求中需要追求数据同步效率，所以用一些比较高效的 Pull 协议就很重要。为了缩减拉取数据量，这些协议本质上来说都是希望高效的计算出尽量精确的数据差异（Diff），然后同步这些必要的数据变动。这里介绍两种我们曾经在工程实践中应用过的 Pull 型数据同步协议。

基于版本号同步——回放日志（RedoLog）和退化算法

在数据源更新时，对于每一次数据变化，基于版本号的同步算法会为这次变化分配一个唯一的递增版本号，并使用一个更新队列记录所有版本号对应的数据变化。

内存副本发起同步请求时，会携带该副本上一次完成同步时的最大版本号，这意味着所有该版本号之后的数据变化都需要被拉取过来。数据源方收到请求后，从更新队列中找到大于该版本号的所有数据变化，并将数据变化汇总，得到最终需要更新的 Diff，返回给发起方。此时内存副本只需要更新这些 Diff 数据即可。

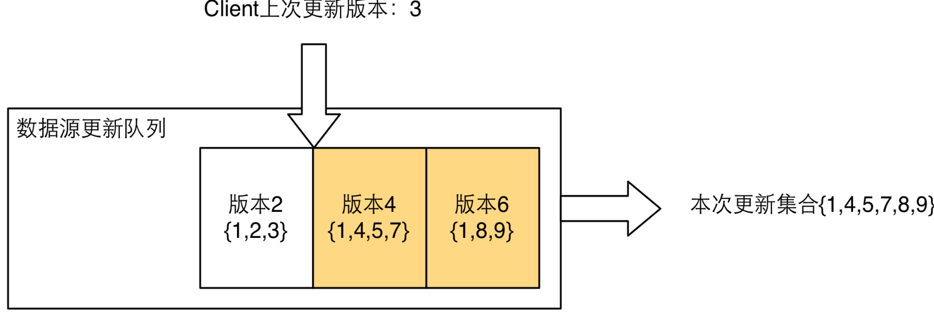


对于大多数的业务场景，特征数据的生成会收口到一个统一的更新服务中，所以

递增版本号可以串行的生成。如果在分布式的数据更新环境中，则需要利用分布式 id 生成器来获取递增版本号。

另一个问题则是更新队列的长度。如果不进行任何优化，更新队列理论上是无限长的，甚至会超过数据集的大小。一个优化方法是我们限制住更新队列的最大长度，一旦长度超过限制，则执行**合并**（Merge）操作。Merge 操作将队列中的数据进行两两合并，合并后的版本号以较大的版本号为准，合并后的更新数据集是两个数据集的并。Merge 后，新的队列长度下降为原更新队列的一半。

Merge 之后的更新队列，我们依然可以使用相同的算法进行同步 Diff 计算：在队列中找到大于上一次更新版本号的所有数据集。可以看到由于版本号的合并，算出的 Diff 不再是完全精准的更新数据，在队列中最早的更新数据集有可能包含部分已经同步过的数据——但这样的退化并不影响同步正确性，仅仅会造成少量的同步冗余，冗余的量取决于 Diff 中最早的数据集经过 Merge 的次数。



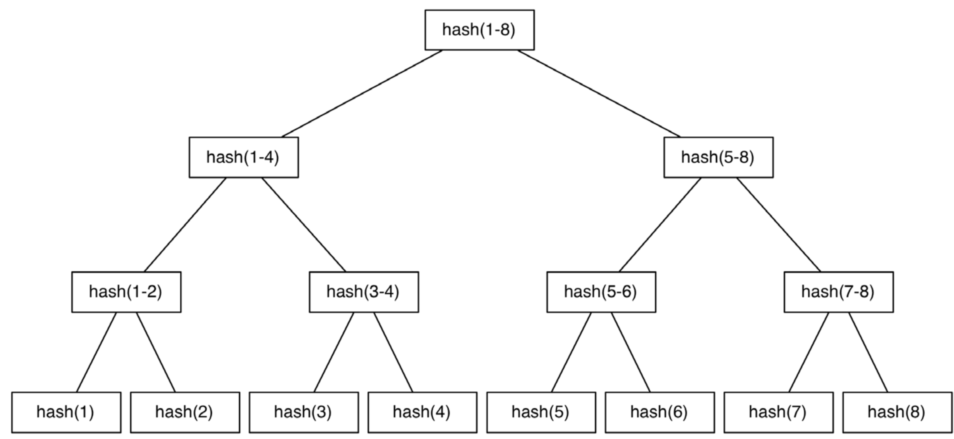
MerkleTree 同步——数据集对比算法

基于版本号的同步使用的是类似 RedoLog 的思想，将业务变动的历史记录下来，并通过回放未同步的历史记录得到 Diff。由于记录不断增长的 RedoLog 需要不小的开销，所以采用了 Merge 策略来退化原始**日志**（Log）。对于批量或者微批量的

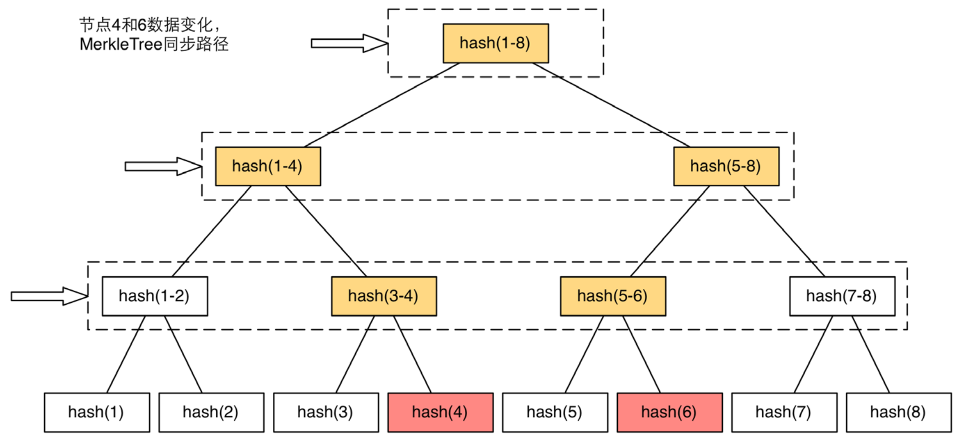
更新来说，基于版本号的同步算法能较好的工作；相反，若数据是实时更新的，将会出现大量的 RedoLog，并快速的退化，影响同步的效率。

Merkle Tree 同步算法走的是另一条路，简单来说就是通过每次直接比较两个数据集的差异来获取 Diff。首先看一个最简单的算法：每次内存副本将所有数据的 Hash 值发送给数据源，数据源比较整个数据集，对于 Hash 值不同的数据执行同步操作——这样就精确计算出了两个数据集之间的 Diff。但显而易见的问题，是每次传输所有数据的 Hash 值可能并不比多传几个数据轻松。Merkle Tree 同步算法就是使用 Merkle Tree 数据结构来优化这一比较过程。

Merkle Tree 简单来说是就是把所有数据集的 hash 值组织成一棵树，这棵树的叶子节点描述一个（或一组）数据的 Hash 值。而中间节点的值由其所有儿子的 Hash 值再次 Hash 得到，描述了以它为根的子树所包含的数据的整体 Hash。显然，在不考虑 Hash 冲突的情况下，如果两颗 Merkle Tree 根节点相同，代表这是两个完全相同的数据集。



Merkle Tree 同步协议由副本发起，将副本根节点值发送给数据源，若与数据源根节点 hash 值一致，则没有数据变动，同步完成。否则数据源将把根结点的所有儿子节点的 hash 发送给副本，进行递归比较。对于不同的 hash 值，一直持续获取直到叶子节点，就可以完全确定已经改变的数据。以二叉树为例，所有的数据同步最多经过 LogN 次交互完成。



2.3.2 客户端缓存技术

当数据规模大，无法完全放入到内存中，冷热数据分明，对于数据时效性要求又不高的时候，通常各类业务都会采用客户端缓存。客户端缓存的集中实现，是特征服务延伸的一部分。通用的缓存协议和使用方式不多说，从在线特征系统的业务角度出发，这里给出几个方向的思考和经验。

接口通用化——缓存逻辑与业务分离

一个特征系统要满足各类业务需求，它的接口肯定是丰富的。从数据含义角度分有用户类、商户类、产品类等等，从数据传输协议分有 Thrift、HTTP，从调用方式角度分有同步、异步，从数据组织形式角度分有单值、List、Map 以及相互嵌套等等……一个良好的架构设计应该尽可能将数据处理与业务剥离开，抽象各个接口的通用部分，一次缓存实现，多处接口同时受益复用。下面以同步异步接口为例介绍客户端接口通用化。

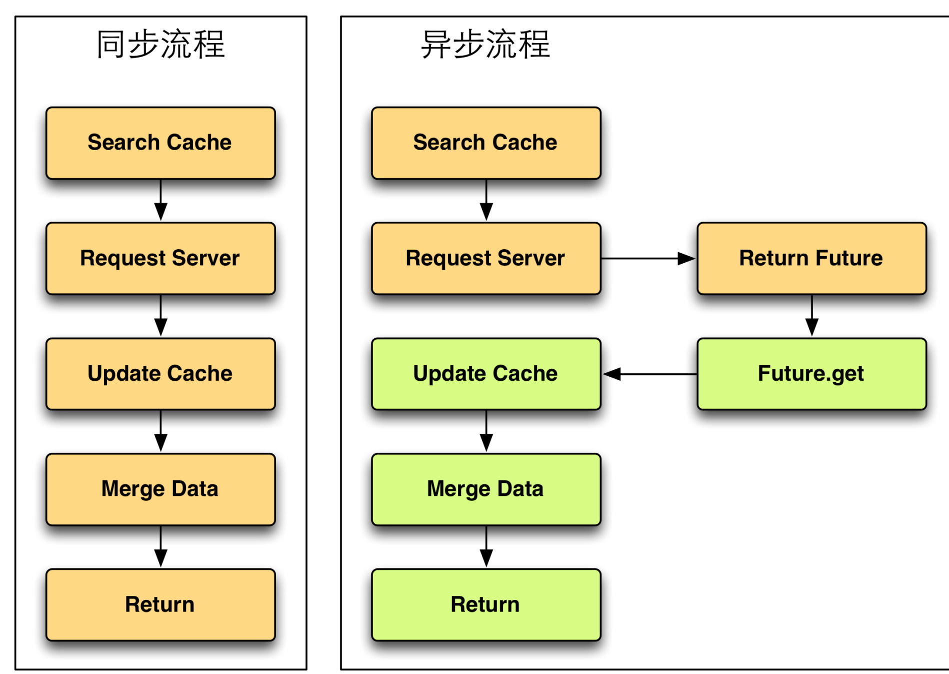
同步接口只有一步：

1. 向服务端发起请求得到结果。

异步接口分为两步：

1. 向服务端发起请求得到 Future 实例。
2. 向 Future 实例发起请求，得到数据。

同步和异步接口的数据处理只有顺序的差别，只需要梳理好各个步骤的执行顺序即可。引入缓存后，数据处理流程对比如下：



不同颜色的处理框表示不同的请求。异步流程需要使用方的两次请求才能获取到数据。像图中“用服务端数据更新缓存”（update cache）、“服务端数据与缓存数据汇总”（merge data）步骤在异步流程里是在第二次请求中完成的，区别于同步流程第一次请求就完成所有步骤。将数据流程拆分为这些子步骤，同步与异步只是这些步骤的不同顺序的组合。因此读写缓存（search cache、update cache）这两个步骤可以抽象出来，与其余逻辑解耦。

数据存储——时间先于空间，客户端与服务端分离

客户端之于服务端，犹如服务端之于数据库，其实数据存储压缩的思路是完全一样的。具体的数据压缩与存储策略在上文数据压缩章节已经做了详细介绍，这里主要想说明两点问题：

客户端压缩与服务端压缩由于应用场景的不同，其目标是有差异的。服务端压缩使用场景是一次性高吞吐写入，逐条高并发低延迟读取，它主要关注的是读取时的解

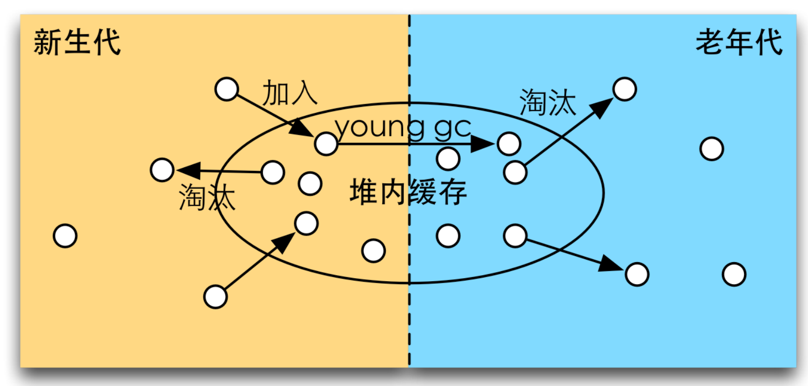
压时间和数据存储时的压缩比。而客户端缓存属于数据存储分层中最顶端的部分，由于读写的场景都是高并发低延迟的本地内存操作，因此对压缩速度、解压速度、数据量大小都有很高要求，它要做的权衡更多。

其次，客户端与服务端是两个完全独立的模块，说白了，虽然我们会编写客户端代码，但它不属于服务的一部分，而是调用方服务的一部分。客户端的数据压缩应该尽量与服务端解耦，切不可为了贪图实现方便，将两者的数据格式耦合在一起，与服务端的数据通信格式应该理解为一种独立的协议，正如服务端与数据库的通信一样，数据通信格式与数据库的存储格式没有任何关系。

内存管理——缓存与分代回收的矛盾

缓存的目标是让热数据（频繁被访问的数据）能够留在内存，以便提高缓存命中率。而 JVM **垃圾回收**（GC）的目标是释放失去引用的对象的内存空间。两者目标看上去相似，但细微的差异让两者在高并发的情景下很难共存。缓存的淘汰会产生大量的内存垃圾，使 Full GC 变得非常频繁。这种矛盾其实不限于客户端，而是所有 JVM 堆内缓存共同面临的问题。下面我们仔细分析一个场景：

随着请求产生的数据会不断加入缓存，QPS 较高的情形下，Young GC 频繁发生，会不断促使缓存所占用的内存从新生代移向老年代。缓存被填满后开始采用 Least Recently Used（LRU）算法淘汰，冷数据被踢出缓存，成为垃圾内存。然而不幸的是，由于频繁的 Young GC，有很多冷数据进入了老年代，淘汰老年代的缓存，就会产生老年代的垃圾，从而引发 Full GC。



可以看到，正是由于缓存的淘汰机制与新生代的 GC 策略目标不一致，导致了缓存淘汰会产生很多老年代的内存垃圾，而且产生垃圾的速度与缓存大小没有太多关系，而与新生代的 GC 频率以及堆缓存的淘汰速度相关。而这两个指标均与 QPS 正相关。因此堆内缓存仿佛成了一个通向老年代的垃圾管道，QPS 越高，垃圾产生越快！

因此，对于高并发的缓存应用，应该避免采用 JVM 的分带管理内存，或者可以说，GC 内存回收机制的开销和效率并不能满足高并发情形下的内存管理的需求。由于 JVM 虚拟机的强制管理内存的限制，此时我们可以将对象序列化存储到**堆外**（Off Heap），来达到绕开 JVM 管理内存的目的，例如 Ehcache，BigMemory 等第三方技术便是如此。或者改动 JVM 底层实现（类似之前淘宝的做法），做到堆内存储，免于 GC。

三、结束语

本文主要介绍了一些在线特征系统的技术点，从系统的高并发、高吞吐、大数据、低延迟的需求出发，并以一些实际特征系统为原型，提出在线特征系统的一些设计思路。正如上文所说，特征系统的边界并不限于数据的存储与读取。像数据导入作业调度、实时特征、特征计算与生产、数据备份、容灾恢复等等，都可看作为特征系统的一部分。本文是在线特征系统系列文章的第一篇，我们的特征系统也在需求与挑战中不断演进，后续会有更多实践的经验与大家分享。一家之言，难免有遗漏和偏颇之处，但是他山之石可以攻玉，若能为各位架构师在面向自己业务时提供一些思路，善莫大焉。

作者简介

杨浩，美团平台及酒旅事业群数据挖掘系统负责人，2011 年毕业于北京大学，曾担任 107 间联合创始人兼 CTO，2016 年加入美团点评。长期致力于计算广告、搜索推荐、数据挖掘等系统架构方向。

伟彬，美团平台及酒旅事业群数据挖掘系统工程师，2015 年毕业于大连理工大学，同年加入美团点评，专注于大数据处理技术与高并发服务。

即时配送的 ETA 问题之亿级样本特征构造实践

超逸

引言

ETA（Estimated time of Arrival，预计送达时间）是外卖配送场景中最重要的变量之一（如图 1）。我们对 ETA 预估的准确度和合理度会对上亿外卖用户的订单体验造成深远影响，这关系到用户的后续行为和留存，是用户后续下单意愿的压舱石。ETA 在配送业务架构中也具有重要地位，是配送运单实时调度系统的关键参数。对 ETA 的准确预估可以提升调度系统的效率，在有限的运力中做到对运单的合理分配。在保障用户体验的同时，对 ETA 的准确预估也可以帮助线下运营构建有效可行的配送考核指标，保障骑手的体验和收益。

图 1　ETA 的业务价值

最近几年，ETA 在互联网行业中的运用取得了令人瞩目的进展，其中以外卖行业和打车行业最令人关注。但与打车行业相比，ETA 在外卖行业中的业务场景更为复杂。如图 2 所示，从业务要素来看，打车涉及到两方——乘客和司机，而外卖行业则涉及了三方——骑手、商家、用户，这使得问题的处理难度提升了一个量级。从业

务的环节来看，打车分为派单、接人、送达三个环节，是一个司机接单到达指定地点接送乘客到达目的地的过程；而外卖则主要分为接单、到店、取餐、送达四个环节，是一个用户、骑手、商家来回交错的场景。业务环节的增加带来了更多的复杂性和不确定性，如骑手操作在各个环节中存在较多的不可控因素，商家可能出餐较慢，此外还有运力规划和天气因素的不确定性等，这就直接导致了外卖 ETA 采取了端到端（下单到接单）的预估方式，相比于拆分成四个环节单独预估具有更强的容错性。无论从业务所涉及的要素还是从业务环节来看，外卖业务的复杂程度远远高于打车业务，对 ETA 预估的难度更大。

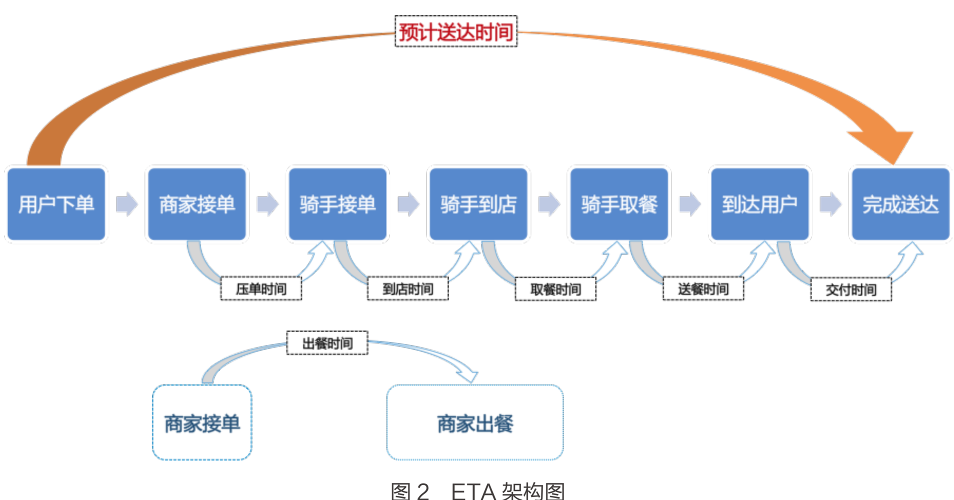


图 2　ETA 架构图

ETA 中比较常用的模型是以 GBDT（Gradient Boost Decision Tree，梯度提升决策树）、RF（RandomForest，随机森林）和线性回归为主的回归预测模型。GBDT 是利用 DT Boosting 的思路，通过梯度求解的方式追踪残差，最终达到利用弱分类器（回归器）构造强分类器（回归器）的目的。RF 在 DT Bagging 的基础之上通过加入样本随机和特征随机的方式引入更多的随机性，解决了决策树泛化能力弱的问题。而线性回归作为线性模型，很容易并行化，处理上亿条训练样本不是问题。但线性模型学习能力有限，需要大量特征工程预先分析出有效的特征、特征组合，从而去间接增强线性回归的线性学习能力。

在回归模型中，特征组合非常重要，但只依靠业务理解和人工经验不一定能带来

效果提升，这导致在实际应用中存在特征匮乏的问题。所以如何发现、构造、组合有效特征，并弥补人工经验的不足，成了 ETA 中重要的一环。

GBDT 构造特征现状

Facebook 2014 年的文章介绍了通过 GBDT 解决 LR 的特征组合问题。[1] GBDT 思想对于发现多种有区分性的特征和组合特征具有天然优势，可以用来构造新的组合特征。在 Facebook 的文章中，会基于样本在 GBDT 中的输出节点索引位置构造 0–1 特征，实现特征的丰富化。新构造的 0–1 特征中，每一个特征对应样本在每棵树中可能的输出位置，它们代表着某些特征的某种逻辑组合。例如一棵树有 n 个叶子节点，当样本在第 k 个叶子节点输出时，则第 k 个特征输出 1，其余 n–1 个特征输出 0，如图 3 所示。

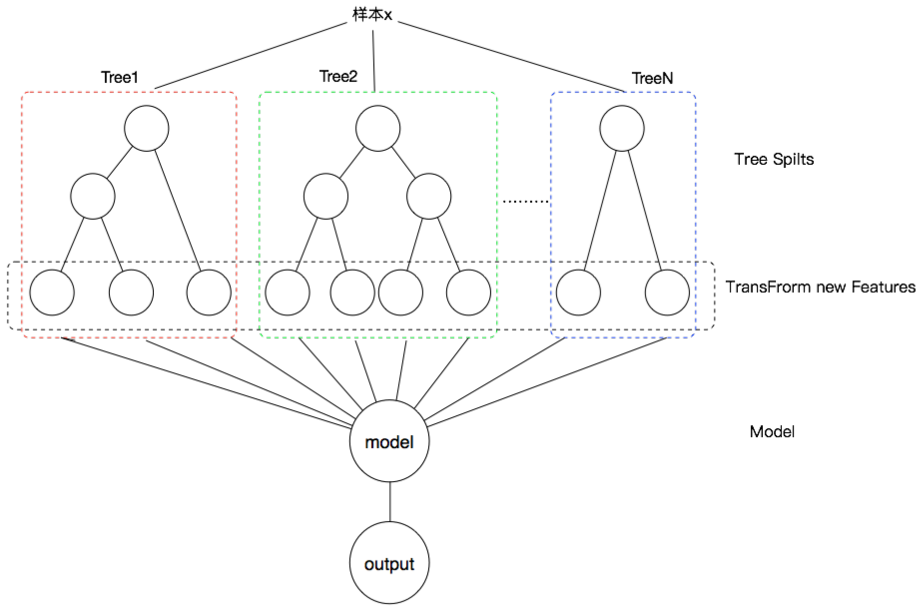


图 3　GBDT(Gradient Boost Decision Tree) 特征构造方法

至于构造新特征的规模，需要由具体业务规模而决定。当 GBDT 中树的数量较多或树深较深时，构造的特征规模也会大幅增加；当业务中所用的数据规模较小时，大规模的构造新特征会导致后续训练模型存在过拟合的可能。所以构造特征的规模需

要足够合理。

GBDT 构造特征在 ETA 场景的中的应用方案

在 ETA 场景中，由于业务场景复杂，所以特征的丰富性决定了 ETA 最终效果的上限。在目前所拥有的特征中，在特征工程的基础阶段，我们依靠业务理解、人工分析和经验总结来构造特征。但从特征层面来看仍然存在欠缺，需要让特征更加丰富化，深度挖掘特征之间的潜在价值。

基础特征构建

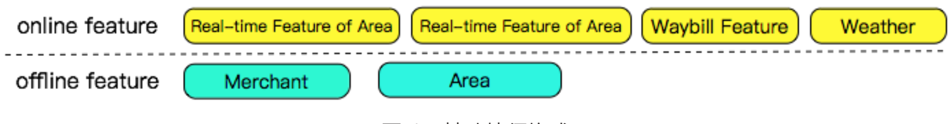


图 4　基础特征构成

特征作为 ETA 中的重要部分，决定了 ETA 的上限。我们基于人工经验和业务理解构建了不同的离线特征和在线特征。

(1) 离线特征

a. 商户画像：商户平均送达时长、到店时长、取餐时长、出餐状况、单量、种类偏好、客单价、平均配送距离。

b. 配送区域画像：区域运力平均水平、骑手规模、单量规模、平均配送距离。

(2) 在线特征

a. 商家实时特征：商家订单挤压状况、过去 N 分钟出单量、过去 N 分钟进单量。

b. 区域实时特征：在岗骑手实时规模、区域挤压（未取餐）单量、运力负载状况。

c. 订单特征：配送距离、价格、种类、时段。

d. 天气数据：温度、气压、降水量。

其中区域实时特征和商家实时特征与配送运力息息相关，运力是决定配送时长和用户体验的重要因素。

GBDT 模型训练和特征构造

利用基础特征，训练用于构造新特征的 GBDT 模型。在 GBDT 中，我们每次训练一个 CART（Classification And Regression Trees）回归树，基于当前轮次 CART 树的损失函数的逆向梯度，拟合下一个 CART 树，直到满足要求为止。

(1) 超参数选择

a. 首先为了节点分裂时质量和随机性，分裂时所使用的最大特征数目为√ n。

b. GBDT 迭代次数（树的数量）。

- 树的数量决定了后续构造特征的规模，与学习速率相互对应。通常学习速率设置较小，但如果过小，会导致迭代次数大幅增加，使得新构造的特征规模过大。
- 通过 GridSearch+CrossValidation 可以寻找到最合适的迭代次数 + 学习速率的超参组合。

c. GBDT 树深度需要足够合理，通常在 4~6 较为合适。

- 虽然增加树的数量和深度都可以增加新构造的特征规模。但树深度过大，会造成模型过拟合以及导致新构造特征过于稀疏。

(2) 训练方案

将训练数据随机抽样 50%，一分为二。前 50% 用于训练 GBDT 模型，后 50% 的数据在通过 GBDT 输出样本在每棵树中输出的叶子节点索引位置，并记录存储，用于后续的新特征的构造和编码，以及后续模型的训练。如样本 x 通过 GBDT 输出后得到的形式如下：x → [25,20,22,....,30,28] ，列表中表示样本在 GBDT 每个树中输出的叶子节点索引位置。

OneHotEncoder（新特征热编码）

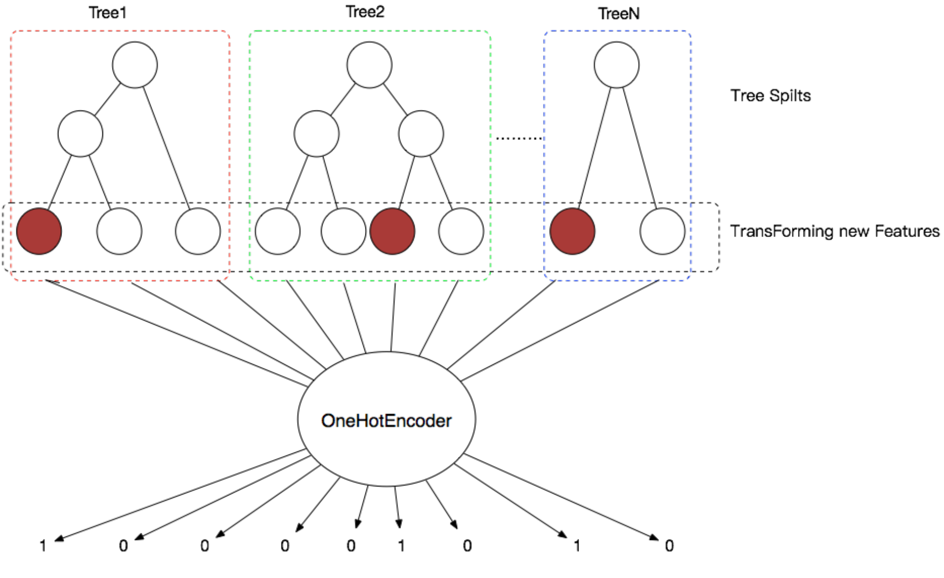


图 5. OneHotEncoder(新特征热编码) 使用方法

由于样本经过 GBDT 输出后得到的 x → [25,20,22,....,30,28] 是一组新特征，但由于这组新特征是叶子节点的 ID，其值不能直接表达任何信息，故不能直接用于 ETA 场景的预估。为了解决上述的问题，避免训练过程中无用信息对模型产生的负面影响，需要通过独热码（OneHotEncoder）的编码方式对新特征进行处理，将新特征转化为可用的 0–1 的特征。

以图 5 中的第一棵树和第二棵树为例，第一棵树共有三个叶子节点，样本会在三个叶子节点的其中之一输出。所以样本在该棵树有会有可能输出三个不同分类的值，需要由 3 个 bit 值来表达样本在该树中输出的含义。图中样本在第一棵树的第一个叶子节点输出，独热码表示为 {100}；而第二棵树有四个叶子节点，且样本在第三个叶子节点输出，则表示为 {0010}。将样本在每棵树的独热码拼接起来，表示为 {1000010}，即通过两棵 CART 树构造了 7 个特征，构造特征的规模与 GBDT 中 CART 树的叶子节点规模直接相关。

基于独热码编码新特征完成后，加上原来的基础特征，特征规模达到 1000+ 以

上，实现特征丰富化。

评估指标与实践效果

评估指标

与传统的回归问题不同，ETA 与实际业务深度耦合，所以需要基于业务因素考虑更多的评估指标，以满足各端（C 端、R 端）用户利益。

① N 分钟准确率：订单实际送达时长与预估时长的绝对误差在 N 分钟内的概率。

1. 业务含义：

- 在 N 分钟准确率中，N 的判定方法来源于绝对误差与用户满意度的关系曲线。通常绝对误差在一定范围内，用户满意度不会有明显波动。如果发现当误差大于 K 分钟时，用户满意度出现明显下滑，则可以将 K 做为 N 分钟准确率中 N 的取值依据。
- 预估时长等同于平台提供给 C 端用户对送餐快慢的心理预期，如果 N 分钟准确率越高，证明预估时长愈发接近用户的心理预期，表示 C 端用户体验越好。

2. 计算方法：

- Xi：样本 i 的绝对误差 =abs(实际送达时长 – 预估时长)。
- 计算每个样本的误差的是否在 N 分钟内，并统计概率 P(Xi <= N)，如图 6、图 7 所示。

$$f(x_i)=\begin{cases}1(x_i<=N)\\0(x_i>N)\end{cases}$$

图 6　判定订单预估是否准确的计算方法

$$P(X_i <= N) = \frac{\sum_{i=1}^{n} f(x_i)}{n}$$

图 7 N 分钟准确率计算方法

② N 分钟业务准时率：实际送达时长与预估时长的差值在 N 分钟内的概率。

1. 业务含义：

- N 分钟业务准时率中 N 的判定方法与 N 分钟准确率类似。
- N 分钟业务准时率是一种合理考核骑手以及保障骑手绩效收益的指标。ETA 场景与其它回归场景相比，在预估准确的同时，预估合理性同样很重要。N 分钟准确率虽然有效地量化 C 端用户体验指标，但无法衡量 R 端骑手体验。所以 N 分钟业务准时率是一个很好的补充指标。

2. 计算方法：

- Xi，样本 i 的有偏差值 =(实际送达时长 − 预估时长)。
- 若 Xi < 0 ，表示骑手提前送达，等同于业务准时。
- 若 0 < Xi <= N ，表示骑手在超时 N 分钟内送达，等同于业务准时；但如果 Xi > N ，表示骑手超时 N 分钟以上送达，从指标层面看，该订单骑手配送业务超时。
- 统计订单配送不超时的概率 P(Xi <= N)，计算方法与 N 分钟准确率 (图 7) 类似。

实践效果对比

我们在此之前已经做了很多特征工程和优化方面的工作，本次我们在不增加任何额外特征的情况下，仅通过模型架构的变化进行优化。在对全量订单进行评估对比的同时，我们对一些高价值和午高峰期间的订单进行重点评估。

① 高价值订单：高价值订单仅占全量订单的 5%。这部分订单用户满意度较低、配送体验较差，属于长尾订单范畴，优化难度高于其它类型订单。

② 午高峰订单：午高峰运单业务占比高达 40%。午高峰期间，商家存在堂食和

外卖资源争抢问题，造成出餐时间不稳定，导致业务中存在更多不确定性，预估难度明显大于非高峰期。

将 GBDT 构造特征 +Ridge 与老版本 base model（GBDT）进行对比。从结果上来看，构造新特征后，可以对 ETA 预估带来更好的效果，其中在高价值订单中，骑手的 N 分钟业务准时率提升显著。具体结论如下：

① 全量订单

平均偏差 (MAE) 减少了 3.4%，误差率减少 1.7 个百分点，N 分钟准确率提升 2.2 个百分点，N 分钟业务准时率持平。

② 高价值订单

平均偏差 (MAE) 减少了 2.56%，误差率减少 1 个百分点，N 分钟准确率提升 1.6 个百分点，N 分钟业务准时率提升 3.46 个百分点。

③ 午高峰订单

平均偏差 (MAE) 减少了 3.1%，误差率减少 1.4 个百分点，N 分钟准确率提升 1.7 个百分点，N 分钟业务准时率持平。

从上述效果来看，GBDT 构造特征可以给 ETA 场景带来更多的提升，在线上使用时，也需要在性能和构造特征的规模上做出取舍。考虑到骑手的主观能动性等因素，通常上线后，线上效果比线下试验效果要更加乐观。

总结

ETA 作为是外卖配送场景中最重要变量之一，是一个复杂程度较高的机器学习问题，其特征的丰富性决定了 ETA 的上限。在业务特征相对匮乏的情况下，GBDT+OneHotEncoder 可以实现特征的丰富化，深度挖掘出特征的潜在价值。实验结果显示，在特征丰富化的情况下，ETA 的准确度明显提高。

与此同时，我们也在尝试进行更多的探索。我们认为时序关系也是 ETA 场景的重要特征，并尝试将该关系特征化加入到目前的模型和策略中，改善特征质量，提高 ETA 的预估能力上限。同时引入深度学习和增强学习，在提高上限的同时，用更好

的模型去接近这个新的预估上限，为 ETA 的场景提升打下坚实的基础。

参考文献

[1] He X, Pan J, Jin O, et al. Practical Lessons from Predicting Clicks on Ads at Facebook[C]. Proceedings of 20th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. ACM, 2014: 1-9.
[2] https://www.csie.ntu.edu.tw/~r01922136/kaggle-2014-criteo.pdf.
[3] GitHub，guestwalk.

即时配送的订单分配策略：从建模和优化

井华

序言

最近两年，外卖的市场规模持续以超常速度发展。近期美团外卖订单量峰值达到1600 万，是全球规模最大的外卖平台。目前各外卖平台正在优质供给、配送体验、软件体验等各维度展开全方位的竞争，其中，配送时效、准时率作为履约环节的重要指标，是外卖平台的核心竞争力之一。

要提升用户的配送时效和准时率，最直接的方法是配备较多的配送员，扩大运力规模，然而这也意味着配送成本会很高。所以，外卖平台一方面要追求好的配送体验，另一方面又被配送的人力成本掣肘。怎么在配送体验和配送成本之间取得最佳的平衡，是即时配送平台生存的根基和关键所在。

随着互联网时代的上半场结束，用户增长红利驱动的粗放式发展模式已经难以适应下半场的角逐。如何通过技术手段，让美团外卖平台超过 40 万的骑手高效工作，在用户满意度持续提升的同时，降低配送成本、提高骑手满意度、驱动配送系统的自动化和智能化，是美团配送技术团队始终致力于解决的难题。

在过去一年多时间里，美团配送团队在机器学习、运筹优化、仿真技术等方面，持续发力，深入研究，并针对即时配送场景特点将上述技术综合运用，推出了用于即时配送的“超级大脑”——O2O 即时配送智能调度系统。

系统首先通过优化设定配送费以及预计送达时间来调整订单结构；在接收订单之后，考虑骑手位置、在途订单情况、骑手能力、商家出餐、交付难度、天气、地理路况、未来单量等因素，在正确的时间将订单分配给最合适的骑手，并在骑手执行过程中随时预判订单超时情况并动态触发改派操作，实现订单和骑手的动态最优匹配；同时，系统派单后，为骑手提示该商家的预计出餐时间和合理的配送线路，并通过语音方式和骑手实现高效交互；在骑手送完订单后，系统根据订单需求预测和运力分布情况，告知骑手不同商圈的运力需求情况，实现闲时的运力调度。通过上述技术和模式

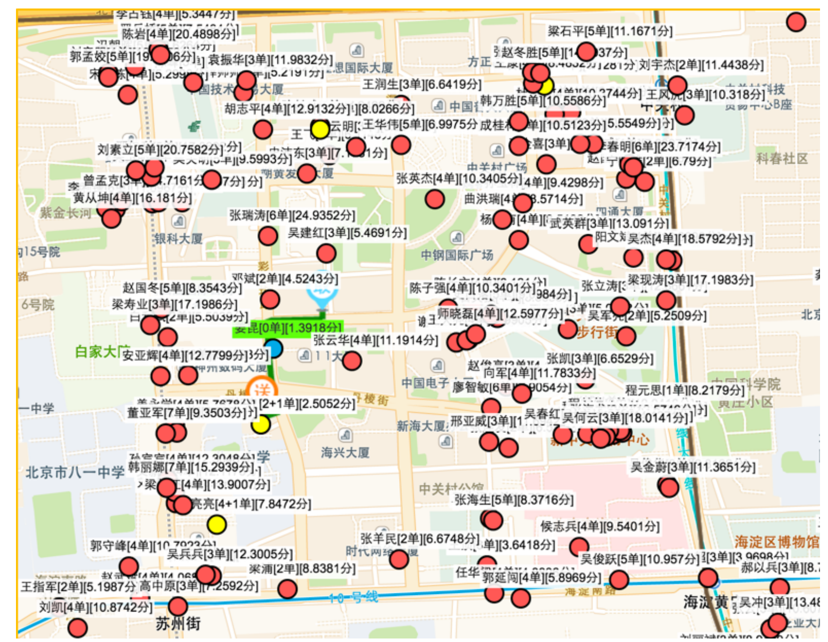
的引入，持续改善了用户体验和配送成本：订单的平均配送时长从 2015 年的 41 分钟，下降到 32 分钟，进一步缩短至 28 分钟，另一方面，在骑手薪资稳步提升的前提下，单均配送成本也有了 20% 以上的缩减。

本文将以外卖场景下上述调度流程中的关键问题之一——订单分配问题为例，阐述该问题的本质特点、模式变迁、方案架构和关键要点，为大家在解决业务优化问题上提供一个案例参考。

外卖订单分配问题描述

外卖订单的分配问题一般可建模为带有若干复杂约束的 DVRP（Dynamic Vehicle Routing Problem）问题。这类问题一般可表述为：有一定数量的骑手，每名骑手身上有若干订单正在配送过程中，在过去一段时间（如 1 分钟）内产生了一批新订单，已知骑手的行驶速度、任意两点间的行驶距离、每个订单的出餐时间和交付时间（骑手到达用户所在地之后将订单交付至用户所需的时间），那么如何将这批新订单在正确的时间分配至正确的骑手，使得用户体验得到保证的同时，骑手的配送效率最高。

下图是外卖配送场景下一个配送区域上众多骑手的分布示意图。



即时配送订单分配模式的演进

在 O2O 领域，订单和服务提供方的匹配问题是一个非常关键的问题。在外卖行业发展初期主要依赖骑手抢单模式和人工派单模式。抢单模式的优势是开发难度低，服务提供者（如司机、骑手）的自由度较高，可以按照自身的需要进行抢单，但其缺点也很明显：骑手 / 司机只考虑自身的场景需求，做出一个局部近优的选择，然而由于每个骑手掌握的信息有限又只从自身利益出发来决策，导致配送整体效率低下，从用户端来看，还存在大量订单无人抢或者抢了之后造成服务质量无法保证（因为部分骑手无法准确预判自己的配送服务能力）的场景，用户体验比较差。

人工派单的方式，从订单分配的结果上来看，一般优于抢单模式。在订单量、骑手数相对比较少的情形下，有经验的调度员可以根据订单的属性特点、骑手的能力、骑手已接单情况、环境因素等，在骑手中逐个比对，根据若干经验规则挑选一个比较合适的骑手来配送。一般而言，人工调度一个订单往往至少需要半分钟左右的时间才能完成。然而，随着外卖订单规模的日益增长，在热门商圈（方圆 3 公里左右）的高峰时段，1 分钟的时间内可能会有 50 单以上，在这种情况下，要求人工调度员每 1–2 秒钟做出一次合理的调度决策，显然是不可能的。另一方面，由于即时配送过程的复杂性，要做出合理的匹配决策，要求调度员对配送范围内各商家的出餐速度、各用户地址的配送难度（例如有的写字楼午高峰要等很长时间的电梯）、各骑手自身的配送工具 / 熟悉的商家和用户范围 / 工作习惯等等要有非常深入的了解，在此基础上具备统筹优化能力，考虑未来进单量、减少空驶等因素，做出全局近优的选择，这对人工调度员而言，又是一项极其艰巨的任务。另外，美团外卖有数千个配送区域，如果采用人工调度方式则每个区域均需要配置调度员，会消耗非常高的人力成本。

该问题虽然复杂，但仍具备一定的规律性。尤其是移动互联网高度发达的今天，我们拥有骑手配送订单过程中的各类大量历史数据，e.g. 骑手的位置、订单状态、天气数据、LBS 数据，利用这些数据辅以相关数学工具使得实现计算机系统的自动派单成为可能。

系统派单具备如下优势：

- 系统可以在全局层面上掌握和配送有关的骑手、商家、用户、订单等各类信息，在此基础上，可以做出全局较优的方案，从而提升配送效率和配送体验，减少配送成本；
- 显著减轻人工调度员的工作，从而降低人工成本，人工调度员只需要在一些意外场景（如配送员出现紧急情况无法继续配送等）发生的时候进行干预即可。

所以，随着数据采集的不断完善和人工智能技术的不断成熟，通过人工智能的方法来进行订单的指派，具有巨大的收益，成为各个配送平台研究的热点之一。

订单智能分配系统的基本架构

美团外卖每天产生巨量的订单配送日志、行驶轨迹数据。通过对配送大数据进行分析、挖掘，会得到每个用户、楼宇、商家、骑手、地理区域的个性化信息，以及有关各地理区块骑行路径的有效数据，那么订单智能分配系统的目标就是基于大数据平台，根据订单的配送需求、地理环境以及每名骑手的个性化特点，实现订单与骑手的高效动态最优匹配，从而为每个用户和商家提供最佳的配送服务，并降低配送成本。

即时配送大数据平台实现对骑手轨迹数据、配送业务数据、特征数据、指标数据的全面管理和监控，并通过模型平台、特征平台支持相关算法策略的快速迭代和优化。机器学习模块负责从数据中寻求规律和知识，例如对商家的出餐时间、到用户所在楼宇上下楼的时间、未来的订单、骑行速度、红绿灯耗时、骑行导航路径等因素进行准确预估，为调度决策提供准确的基础信息；而运筹优化模块则在即时配送大数据

平台以及机器学习的预测数据基础上，采用最优化理论、强化学习等优化策略进行计算，做出全局最优的分配决策，并和骑手高效互动，处理执行过程中的问题，实现动态最优化。

问题分析和建模：高效求解问题的第一步

学术研究领域有很多经典的优化问题（如旅行商问题 TSP、装箱问题 BP、车辆路径问题 VRP 等），它们的决策变量、优化目标和约束条件往往非常明确、简单。这在学术研究中是很必要的，因为它简化了问题，让研究者把精力放在如何设计高效算法上。然而，由于实际工业场景的复杂性，绝大部分实际场景的决策优化问题很难描述的如此简单，此时，如果不仔细分析实际业务过程特点而错误地建立了和实际场景不符的模型，自然会造成我们获得的所谓“最优解”应用于实际后也会“水土不服”，最后被大量抱怨甚至抛弃。所以我们说，准确建模是实际决策优化项目的第一步，也是最关键的一步。

准确建模，包括两个方面的问题：

- 我们正确理解了实际业务场景的优化问题，并且通过某种形式化语言进行了准确描述；
- 我们建立的模型中，涉及的各类参数和数据，能够准确得获取。

在上述两个前提下，采用相应的高效优化算法求解模型所得到的最优解，就是符合实际场景需求的最优决策方案。第一个问题，一般是通过业务调研、分析并结合建模工具来得到；而解决第二个问题，则更多地需要依赖数据分析、机器学习、数据挖掘技术结合领域知识，对模型进行精确的量化表达。

一个决策优化问题的数学模型，一般包括三个要素：

- 决策变量
- 优化目标
- 约束条件

其中，决策变量说明了我们希望算法来帮助我们做哪些决策；优化目标则是指我们通过调整决策变量，使得哪些指标得到优化；而约束条件则是在优化决策的过程中所考虑的各类限制性因素。

为了说明即时配送场景下的订单分配问题，我们先引入若干符号定义：

n：订单数量
m：骑手数量
r_i：订单 i 的下单时刻
d_i：订单 i 的期望送达时刻
p_i：订单 i 的备货消耗时长
v_j：骑手 j 的骑行速度
u_i：订单 i 的交付消耗时长
pos_j：骑手 j 的当前位置
(i,B)：订单 i 的取货任务
(i,C)：订单 i 的送货任务
Ω：所有任务集合
$pos(i,B)$：订单 i 的取货位置
$pos(i,C)$：订单 i 的取货位置
$l_{pos1,pos2}$：pos1到 $pos2$的导航距离

在即时配送调度场景下，决策变量包括各个订单需要分配的骑手，以及骑手的建议行驶路线。

Ω_j：骑手 j 所分配的任务集合
seq_j：骑手 j 所分配任务 执行顺序
$fr_{(i,B)}$：订单 i 的到达商户时刻
$f_{(i,B)}$：订单 i 的取货时刻
$fr_{(i,C)}$：订单 i 的到达用户时刻
$f_{(i,C)}$：订单 i 的送达时刻

即时配送订单分配问题的优化目标一般包括希望用户的单均配送时长尽量短、骑手付出的劳动尽量少、超时率尽量低，等等。一般可表达为：

目标1：最小化超时率

$$\min g_1(\Omega) = \frac{\sum_{i=1}^{n} 1\left(f_{(i,C)} \geq d_i\right)}{n}$$

目标2：最小化单均行驶距离

$$\min g_2(\Omega) = \frac{\sum_{j=1}^{m} \sum_{k1,k2 \in seq_j} l_{k1,k2}}{n}$$

目标3：最小化单均消耗时间

$$\min g_3(\Omega) = \frac{\sum_{j=1}^{m} \max_{(i,x) \in seq_j} T_{(i,x)}}{n}$$

针对实际场景下的配送订单分配问题，设置哪些指标作为目标函数是一个较为复杂的问题。

原因在于两个方面：

1）该优化问题是多目标的，且各个目标在不同时段、不同环境下会有差别。举个例子，经验丰富的调度员希望在负载较低的空闲时段，将订单派给那些不熟悉区域地形的骑手，以锻炼骑手能力；在天气恶劣的情况下，希望能够容忍一定的超时率更多地派顺路单，以提高订单消化速度等。这些考量有其合理性，需要在优化目标中予以体现。

2）缺乏有助于量化优化目标的数据。如果带标签数据足够多，同时假设调度员的能力足够好，那么可以通过数据挖掘的手段获取优化目标的量化表达。不幸的是，这两个前提都不成立。我们针对该难题，首先通过深入调研明确业务痛点和目标，在此基础上，采用机理和数据相结合的办法，由人工设定目标函数的结构，通过仿真系统（下文介绍）和实际数据去设定目标函数的参数，来确定最终采用的目标函数形态。

即时配送调度问题的约束条件至少涵盖如下几种类型：

约束1：一个订单的送货要在取货后进行

$$f_{i,B}+\frac{l_{pos(i,B),pos(i,C)}}{\max\limits_{j} v_j}\le fr_{(i,C)}+u_i=f_{i,C}$$

$$\forall i=1,2,\cdots,n,\ \forall j=1,2,\cdots,m$$

约束2：订单取货需要备货完成和骑手到达都具备后才能进行

$$f_{i,B}\ge r_i+p_i \qquad \forall i=1,2,\cdots,n$$

$$f_{i,B}\ge fr_{(i,C)} \qquad \forall i=1,2,\cdots,n$$

约束3：同一个骑手所分配的多项任务的完成时间限制

$$f_{(i1,x1)}\ge f_{(i2,x2)}+\frac{l_{pos(i1,x1),pos(i2,x2)}}{v_j}$$

$$\forall\ (i1,x1),(i2,x2)\in\Omega_j,\ \forall j=1,2,\cdots,m$$

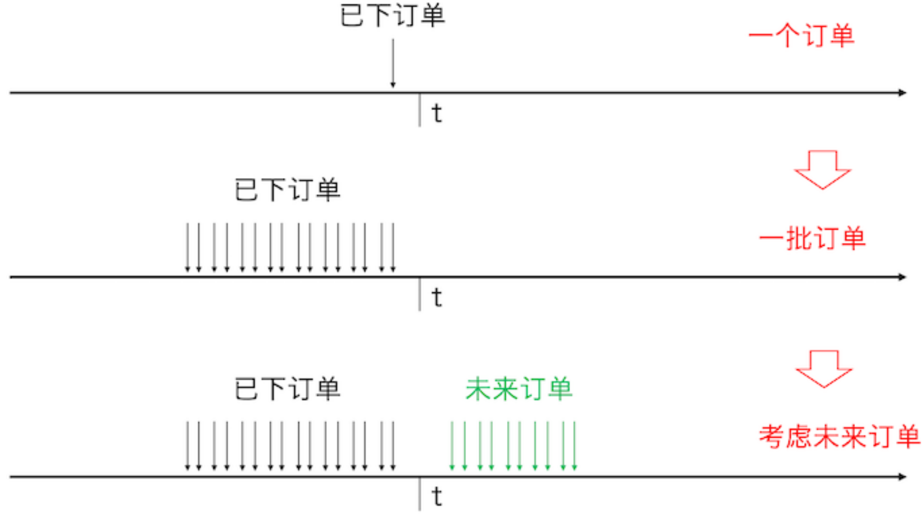
约束4：一个订单的取送，要由同一名骑手完成

$$\forall\ (i,B),(i,C)\in\Omega,$$

$$\exists j\ 满足\ (i,B),(i,C)\in\Omega_j$$

除了以上约束外，有时还需要考虑部分订单只能由具备某些特点的骑手来配送（例如火锅订单只能交给携带专门装备的骑手等）、载具的容量限制等。

以上只是针对给定的一批订单进行匹配决策的优化问题在建模时所需考虑的部分因素。事实上，在外卖配送场景中，我们希望的不是单次决策的最优，而是策略在一段时间应用后的累积收益最大。换句话说，我们不追求某一个订单的指派是最优的，而是希望一天下来，所有的订单指派结果整体上是全局最优的。这进一步加大了问题建模的难度，原因在于算法在做订单指派决策的时候，未来的订单信息是不确定的，如下图所示，在 t 时刻进行决策的时候，既需要考虑已确定的订单，还需要考虑未来的尚未确定的订单。运筹优化领域中的马尔可夫决策过程描述的就是这样的一类在不确定、信息不完备环境下的序贯决策优化问题。



问题建模中的机器学习

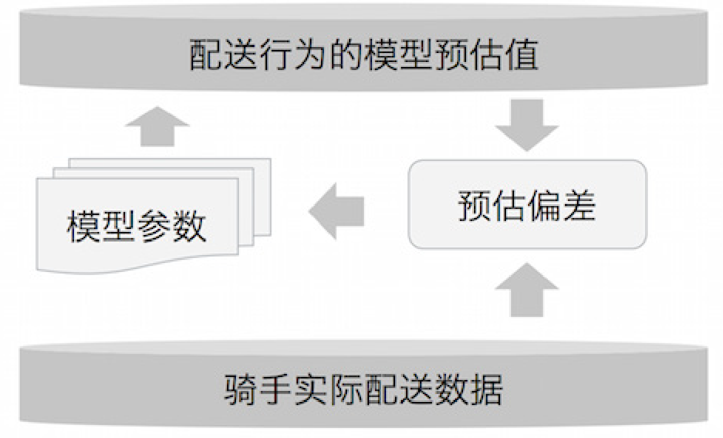
过去，在信息化水平较低的环境下，很多工业运筹优化类的项目不成功，重要原因之一就是缺少足够完备的数据采集基础工具，大量数据由人工根据经验设定，其准确性难以保证，且难以随着环境变化而自适应调整，从而造成模型的优化结果渐渐变得不符合实际。机器学习领域有个谚语“Garbage in，garbage out”，说明了精准的基础数据对于人工智能类项目的重要性。

即时配送订单分配场景下的数据包括两类：

- 直接通过业务系统采集可获取的数据，例如订单数据、骑手负载数据、骑手状态数据等。
- 无法直接采集得到，需要预测或统计才能获取的数据，如商户出餐时间、用户驻留时间（骑手到达用户处将订单交付给用户的时间）、骑手配送能力等。

第一类数据的获取一般由业务系统、骑手端 App 直接给出，其精度通过提升工程质量或操作规范可有效保证；而第二类数据的获取是即时配送调度的关键难点之一。

在订单的配送过程中，骑手在商家、用户处的取餐和交付时间会占到整个订单配送时长的一半以上。准确估计出餐和交付时间，可以减少骑手的额外等待，也能避免“餐等人”的现象。商家出餐时间的长短，跟品类、时段、天气等因素都有关，而交付时间更为复杂，用户在几楼，是否处于午高峰时段，有没有电梯等等，都会影响骑手（到了用户所在地之后）交付订单给用户的时间。对这两类数据，无法单纯通过机理来进行预测，因为相关数据无法采集到（如商家今天有几个厨师值班、用户写字楼的电梯是否开放，等等）。为解决这些问题，我们利用机器学习工具，利用历史的骑手到店、等餐、取餐的数据，并充分考虑天气等外部因素的影响，建立了全面反映出餐能力的预测模型，并通过实时维度的特征进行修正，得到准确的出餐 / 交付时间估计。



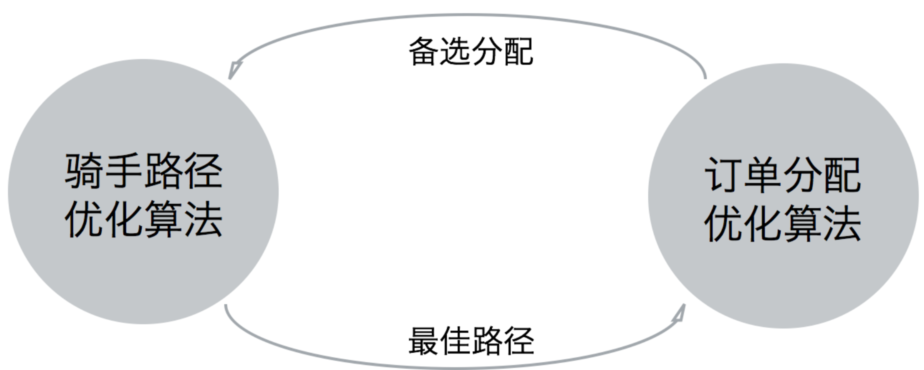
进一步，我们建立了调度模型的自学习机制，借鉴多变量控制理论的思想，不断根据预估偏差调整预估模型中的相关参数。通过以上工作，我们通过调度模型来预估

骑手的配送行为（取餐时间和送达时间），平均偏差小于 4 分钟，10 分钟置信度达到 90% 以上，有效地提升了派单效果和用户满意度。

订单——骑手的匹配优化

如果说上述建模过程的目标是构建和实际业务吻合的解空间，优化算法的作用则是在我们构建的解空间里找到最优的策略。配送调度问题属于典型的 NP-Hard 类离散系统优化问题，解空间巨大。以一段时间内产生 50 个订单，一个区域有 200 骑手，每个骑手身上有 5 个订单为例，那么对应的调度问题解空间规模将达到 pow(200,50)*10（部分为不可行解），这是一个天文数字！所以，如何设计好的优化算法，从庞大的解空间中搜索得到一个满意解（由于问题的 NP-Hard 特性，得到最优解几乎是不可能的），是一个很大的挑战。即时配送对于优化算法的另一个要求是高实时性，算法只允许运行 2~3 秒钟的时间必须给出最终决策，这和传统物流场景的优化完全不同。

针对此难题，我们采用了两个关键思路。一是问题特征分析。运筹优化领域有个说法叫“No Free Lunch Theory”，没有免费的午餐，含义是说如果没有对问题的抽象分析并在算法中加以利用，那么没有算法会比一个随机算法好。换句话说，就是我们必须对问题特点和结构进行深入分析，才能设计出性能优越的算法。在运筹优化领域中的各类基础性算法也是这样的更多思路，如单纯形、梯度下降、遗传算法、模拟退火、动态规划等，它们的本质其实是假定了问题具备某些特征（如动态规划的贝尔曼方程假设，遗传算法的 Building Blocks 假设等），并利用这些假设进行算法设计。那么，针对配送调度的场景，这个问题可以被分解为两个层次：骑手路径优化和订单分配方案的优化。骑手路径优化问题要解决的问题是：在新订单分配至骑手后，确定骑手的最佳配送线路；而订单分配优化问题要解决的问题是：把一批订单分配至相应的骑手，使得我们关注的指标（如配送时长、准时率、骑手的行驶距离等）达到最优。这两个问题的关系是：通过订单分配优化算法进行初始的订单分配，然后通过骑手路径优化算法获取各骑手的最佳行驶路线，进而，订单分配优化算法根据骑手路径优化结果调整分配方案。这两个层次不断反复迭代，最终获得比较满意的解。



第二个思路是跨学科结合。订单分配问题在业内有两类方法，第一类方法是把订单分配问题转换成图论中的二分图匹配问题来解决。但是由于标准的二分图匹配问题中，一个人只能被分配一项任务，所以常用的一个方法是先对订单进行打包，将可以由一个人完成的多个订单组成一个任务，再使用二分图匹配算法（匈牙利算法、KM 算法）来解决。这种做法是一个不错的近似方案，优点是实现简单计算速度快，但它的缺点是会损失一部分满意解。第二类方法是直接采用个性化的算法进行订单分配方案的优化，优点是不损失获得满意解的可能性，但实际做起来难度较大。我们结合领域知识、优化算法、机器学习策略以及相关图论算法，基于分解协调思想，设计了骑手路径优化算法和订单分配优化算法。进一步，我们利用强化学习的思想，引入了离线学习和在线优化相结合的机制，离线学习得到策略模型，在线通过策略迭代，不断寻求更优解。通过不断地改进算法，在耗时下降的同时，算法的优化效果提升 50% 以上。

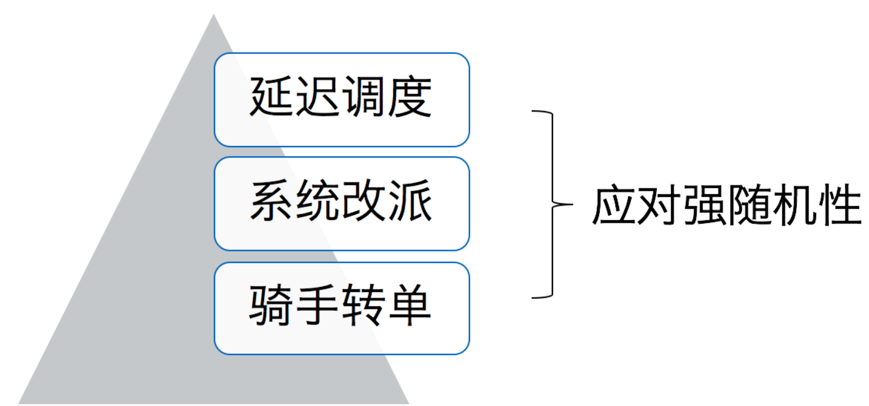
我们在大量的实际数据集上进行评估验证，99% 以上的情况下，骑手路径优化算法能够在 30ms 内给出最优解。为了有效降低算法运行时间，我们对优化算法进行并行化，并利用并行计算集群进行快速处理。一个区域的调度计算会在数百台计算机上同步执行，在 2~3 秒内返回满意结果，每天的路径规划次数超过 50 亿次。

应对强随机性

即时配送过程的一个突出特点是线下的突发因素多、影响大，例如商家出餐异常慢、联系不上用户、车坏了、临时交通管制等等。这些突发事件造成的一个恶劣结果

是，虽然在指派订单的时刻，所指派的骑手是合理的，然而过了一段时间之后，由于骑手、订单等状态发生了变化，会变得不够合理。订单交给不合适的骑手来完成，会造成订单超时，以及骑手需要额外的等待时间来完成订单，影响了配送效率和用户体验的提升。

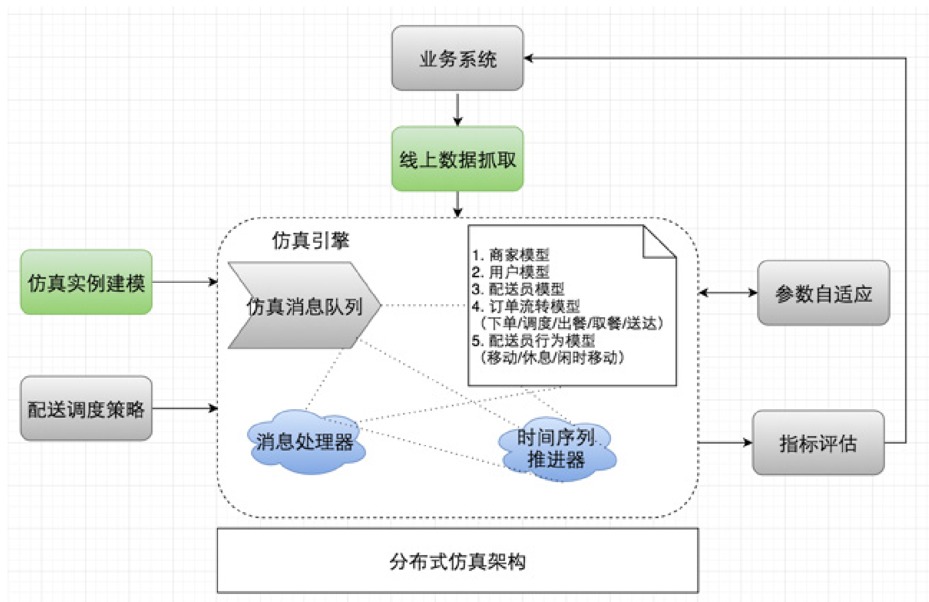
在出现上述不确定因素造成派单方案变得不合理的情况时，现有方法主要通过人工来完成，即：配送站长 / 调度员在配送信息系统里，查看各个骑手的位置、手中订单的状态及商户 / 用户的位置 / 期望送达时间等等信息，同时接听骑手的电话改派请求，在此基础上，分析哪些订单应该改派，以及应该改派给哪位骑手，并执行操作。



我们针对即时配送的强不确定性特点，提出了两点创新：一是延迟调度策略，即在某些场景订单可以不被指派出去，在不影响订单超时的情况下，延迟做出决策；二是系统自动改派策略，即订单即便已经派给了骑手，后台的智能算法仍然会实时评估各个骑手的位置、订单情况，并帮助骑手进行分析，判断是否存在超时风险。如果存在，则系统会评估是否有更优的骑手来配送。延迟调度的好处一方面是在动态多变的不确定环境下，寻求最佳的订单指派时机，以提高效率；另一方面是在订单高峰时段存在大量堆积时，减轻骑手的配送压力。有了这两项策略，订单的调度过程更加立体、全面，覆盖了订单履行过程全生命周期中的主要优化环节，实现订单和骑手的动态最优化匹配。

仿真系统

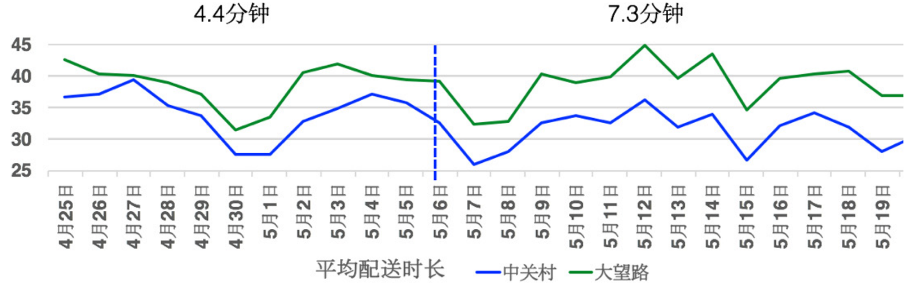
工业系统非常看重监控和评估，“No measurement, No improvement”。在工业优化场景中，如何准确评估算法的好坏，其重要性不亚于设计一个好的算法。然而，由于多个订单在线下可能会由同一名骑手来配送，订单与订单之间存在耦合关系，导致无法做订单维度的 A/B 测试。而区域维度指标受天气、订单结构、骑手水平等外在随机因素影响波动比较大，算法效果容易被随机因素湮没从而无法准确评估。为此，我们针对即时配送场景，建立了相应的仿真模型，开发了配送仿真系统。系统能够模拟真实的配送过程和线上调度逻辑，并给出按照某种配送策略下的最终结果。该模拟过程和线下的实际导航、地理数据完全一致，系统同时能够根据实际配送数据进行模型自学习，不断提升仿真精度。

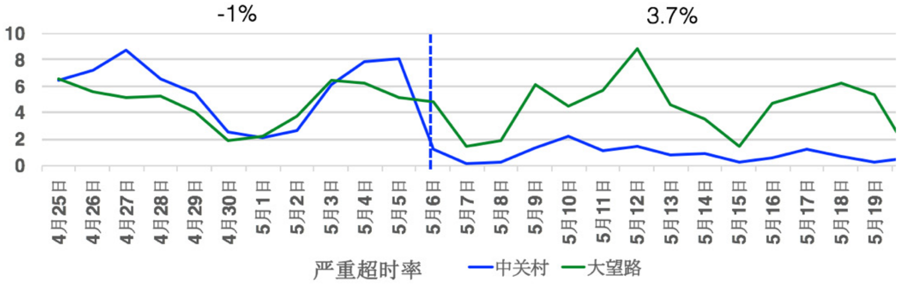


一个高精度的配送仿真系统，除了能够对配送调度算法进行准确评估和优化，从而实现高效的策略准入控制外，另一个巨大的价值在于能够对配送相关的上下游策略进行辅助优化，包括配送范围优化、订单结构优化、运力配置优化、配送成本评估等等，其应用的想象空间非常大。

结语

美团配送智能调度系统在应用之后，取得了非常不错的应用效果。下图说明了在订单结构比较类似的两个白领区域上的 A/B 测试结果。中关村配送站在 5 月 6 日切换了派单模式和相应的算法，大望路配送站的调度策略维持不变。可以看出，在切换后，中关村的平均配送时长有了 2.9 分钟的下降，严重超时率下降了 4.7 个百分点（相比较对比区域）。





同时，在更广泛的区域上进行了测试，结果表明，在体验指标不变的前提下，新策略能够降低 19% 的运力消耗。换言之，原来 5 个人干的活，现在 4 个人就能干好，所以说，智能调度在降低成本上价值是很大的。

美团配送的目标之一是做本地化的物流配送平台，那么，效率、体验和成本将成为平台追求的核心指标。人工智能技术在美团配送的成功应用有很多，通过大数据、人工智能手段打造一个高效、智能化、动态协同优化的本地智慧物流平台，能显著提高本地、同城范围内的物流配送效率，持续提升配送体验，降低配送成本。

外卖 O2O 的用户画像实践

李滔

美团外卖经过 3 年的飞速发展，品类已经从单一的外卖扩展到了美食、夜宵、鲜花、商超等多个品类。用户群体也从早期的学生为主扩展到学生、白领、社区以及商旅，甚至包括在 KTV 等娱乐场所消费的人群。随着供给和消费人群的多样化，如何在供给和用户之间做一个对接，就是用户画像的一个基础工作。所谓千人千面，画像需要刻画不同人群的消费习惯和消费偏好。

外卖 O2O 和传统的电商存在一些差异。可以简单总结为如下几点：

1）新事物，快速发展：这意味很多用户对外卖的认知较少，对平台上的新品类缺乏了解，对自身的需求也没有充分意识。平台需要去发现用户的消费意愿，以便对用户的消费进行引导。

2）高频：外卖是个典型的高频 O2O 应用。一方面消费频次高，用户生命周期相对好判定；另一方面消费单价较低，用户决策时间短、随意性大。

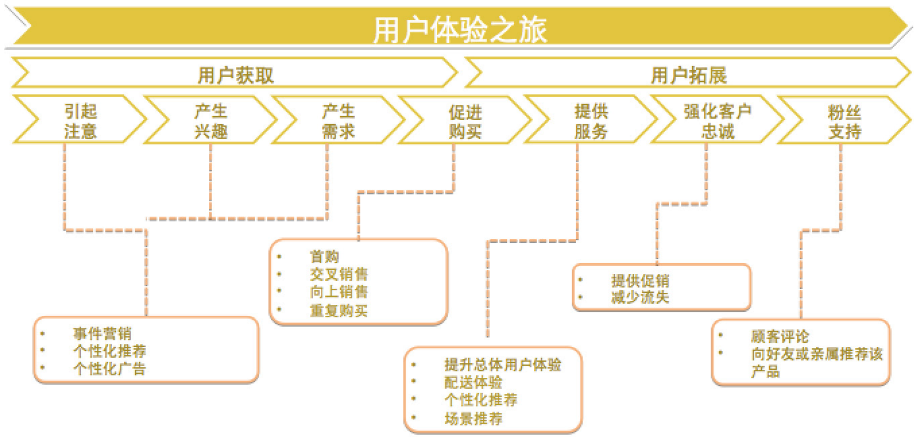
3）场景驱动：场景是特定的时间、地点和人物的组合下的特定的消费意图。不同的时间、地点，不同类型的用户的消费意图会有差异。例如白领在写字楼中午的订单一般是工作餐，通常在营养、品质上有一定的要求，且单价不能太高；而到了周末晚上的订单大多是夜宵，追求口味且价格弹性较大。场景辨识越细致，越能了解用户的消费意图，运营效果就越好。

4）用户消费的地理位置相对固定，结合地理位置判断用户的消费意图是外卖的一个特点。

外卖产品运营对画像技术的要求

如下图所示，我们大致可以把一个产品的运营分为用户获取和用户拓展两个阶段。在用户获取阶段，用户因为自然原因或一些营销事件（例如广告、社交媒体传播）产生对外卖的注意，进而产生了兴趣，并在合适的时机下完成首购，从而成为外卖新

客。在这一阶段，运营的重点是提高效率，通过一些个性化的营销和广告手段，吸引到真正有潜在需求的用户，并刺激其转化。在用户完成转化后，接下来的运营重点是拓展用户价值。这里有两个问题：第一是提升用户价值，具体而言就是提升用户的单均价和消费频次，从而提升用户的 LTV（life-time value）。基本手段包括交叉销售（新品类的推荐）、向上销售（优质高价供给的推荐）以及重复购买（优惠、红包刺激重复下单以及优质供给的推荐带来下单频次的提升）；第二个问题是用户的留存，通过提升用户总体体验以及在用户有流失倾向时通过促销和优惠将用户留在外卖平台。

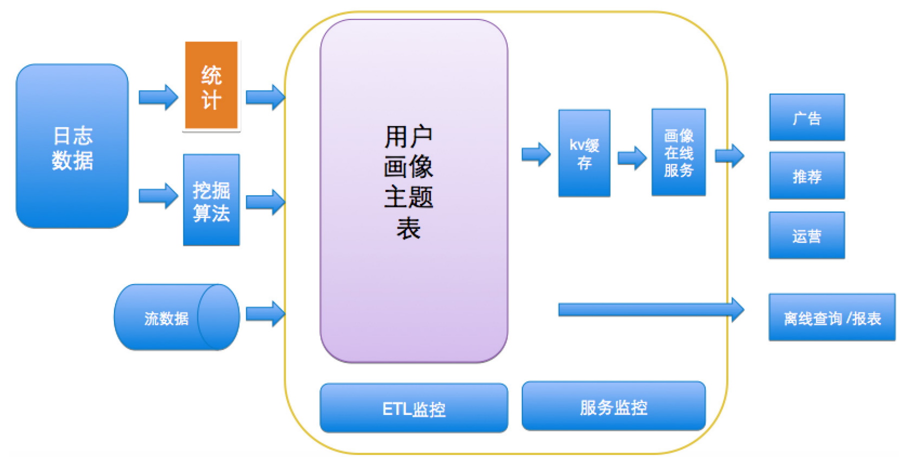


所以用户所处的体验阶段不同，运营的侧重点也需要有所不同。而用户画像作为运营的支撑技术，需要提供相应的用户刻画以满足运营需求。根据上图的营销链条，从支撑运营的角度，除去提供常规的用户基础属性（例如年龄、性别、职业、婚育状况等）以及用户偏好之外，还需要考虑这么几个问题：1）什么样的用户会成为外卖平台的顾客（新客识别）；2）用户所处生命周期的判断，用户是否可能从平台流失（流失预警）；3）用户处于什么样的消费场景（场景识别）。后面“外卖 O2O 的用户画像实践”一节中，我们会介绍针对这三个问题的一些实践。

外卖画像系统架构

下图是我们画像服务的架构：数据源包括基础日志、商家数据和订单数据。数据完成处理后存放在一系列主题表中，再导入 kv 存储，给下游业务端提供在线服务。

同时我们会对整个业务流程实施监控。主要分为两部分，第一部分是对数据处理流程的监控，利用用内部自研的数据治理平台，监控每天各主题表产生的时间、数据量以及数据分布是否有异常。第二部分是对服务的监控。目前画像系统支持的下游服务包括：广告、排序、运营等系统。



外卖 O2O 的用户画像实践

新客运营

新客运营主要需要回答下列三个问题：

1）新客在哪里?

2）新客的偏好如何?

3）新客的消费力如何?

回答这三个问题是比较困难的，因为相对于老客而言，新客的行为记录非常少或者几乎没有。这就需要我们通过一些技术手段作出推断。例如：新客的潜在转化概率，受到新客的人口属性（职业、年龄等）、所处地域（需求的因素）、周围人群（同样反映需求）以及是否有充足供给等因素的影响；而对于新客的偏好和消费力，从新客在到店场景下的消费行为可以做出推测。另外用户的工作和居住地点也能反映他的消费能力。

对新客的预测大量依赖他在到店场景下的行为，而用户的到店行为对于外卖是比较稀疏的，大多数的用户是在少数几个类别上有过一些消费行为。这就意味着我们需要考虑选择什么样的统计量描述：是消费单价，总消费价格，消费品类等等。然后通过大量的试验来验证特征的显著性。另外由于数据比较稀疏，需要考虑合适的平滑处理。

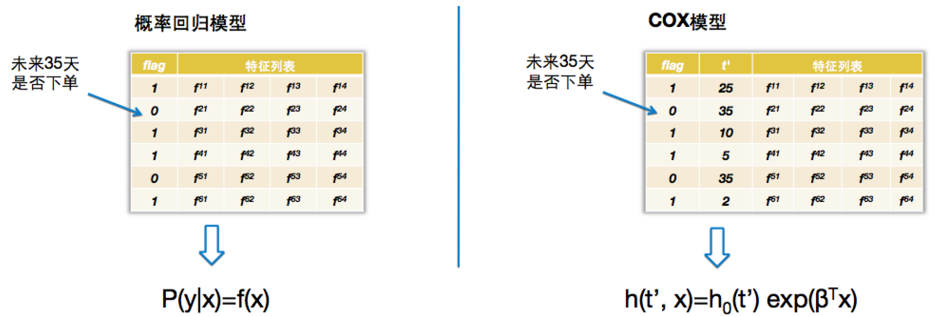
我们在做高潜新客挖掘时，融入了多方特征，通过特征的组合最终作出一个效果比较好的预测模型。我们能够找到一些高转化率的用户，其转化率比普通用户高若干倍。通过对高潜用户有针对性的营销，可以极大提高营销效率。

流失预测

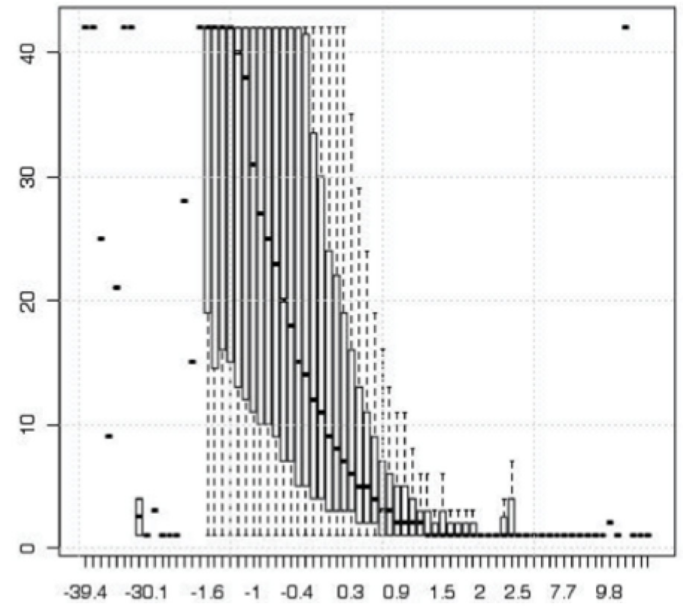
新客来了之后，接下来需要把他留在这个平台上，尽量延长生命周期。营销领域关于用户留存的两个基本观点是（引自菲利普 . 科特勒《营销管理》）：

> 获取一个新顾客的成本是维系现有顾客成本的 5 倍！
>
> 如果将顾客流失率降低 5%，公司利润将增加 25%~85%

用户流失的原因通常包括：竞对的吸引；体验问题；需求变化。我们借助机器学习的方法，构建用户的描述特征，并借助这些特征来预测用户未来流失的概率。这里有两种做法：第一种是预测用户未来若干天是否会下单这一事件发生的概率。这是典型的概率回归问题，可以选择逻辑回归、决策树等算法拟合给定观测下事件发生的概率；第二种是借助于生存模型，例如 COX-PH 模型，做流失的风险预测。下图左边是概率回归的模型，用户未来 T 天内是否有下单做为类别标记 y，然后估计在观察到特征 X 的情况下 y 的后验概率 P(y|X)。右边是用 COX 模型的例子，我们会根据用户在未来 T 天是否下单给样本一个类别，即观测时长记为 T。假设用户的下单的距今时长 t<T，将 t 作为生存时长 t'；否则将生存时长 t' 记为 T。这样一个样本由三部分构成：样本的类别 (flag)，生存时长 (t') 以及特征列表。通过生存模型虽然无法显式得到 P(t'|X) 的概率，但其协变量部分实际反映了用户流失的风险大小。



生存模型中，$\beta^T x$ 反映了用户流失的风险，同时也和用户下次订单的时间间隔成正相关。下面的箱线图中，横轴为 $\beta^T x$，纵轴为用户下单时间的间隔。



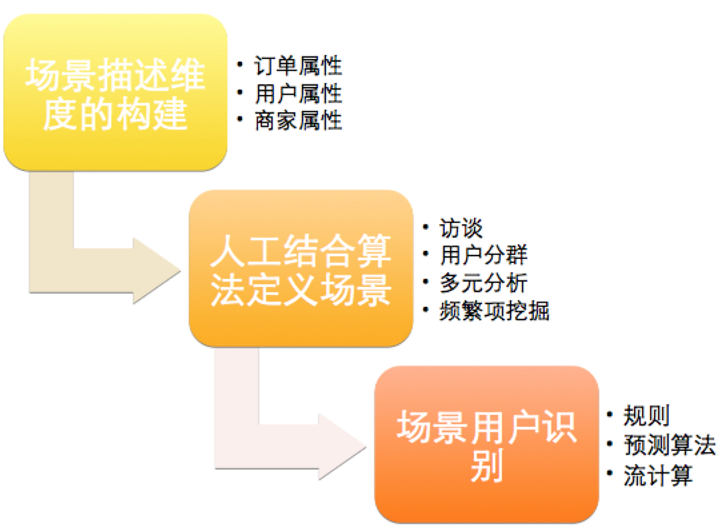
我们做了 COX 模型和概率回归模型的对比。在预测用户 XX 天内是否会下单上面，两者有相近的性能。

美团外卖通过使用了用户流失预警模型，显著降低了用户留存的运营成本。

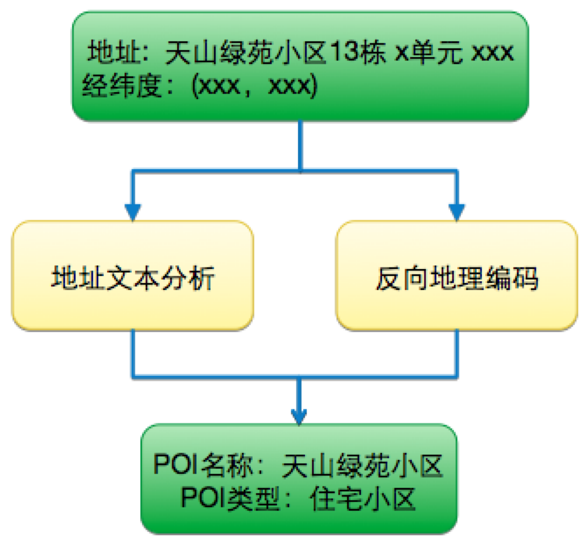
场景运营

拓展用户的体验，最重要的一点是要理解用户下单的场景。了解用户的订餐场景

有助于基于场景的用户运营。对于场景运营而言，通常需要经过如下三个步骤：



场景可以从时间、地点、订单三个维度描述。比如说工作日的下午茶，周末的家庭聚餐，夜里在家点夜宵等等。其中重要的一点是用户订单地址的分析。通过区分用户的订单地址是写字楼、学校或是社区，再结合订单时间、订单内容，可以对用户的下单场景做到大致的了解。



上图是我们订单地址分析的流程。根据订单系统中的用户订单地址文本，基于自然语言处理技术对地址文本分析，可以得到地址的主干名称（指去掉了楼宇、门牌号的地址主干部分）和地址的类型（写字楼、住宅小区等）。在此基础上通过一些地图数据辅助从而判断出最终的地址类型。

另外我们还做了合并订单的识别，即识别一个订单是一个人下单还是拼单。把拼单信息、地址分析以及时间结合在一起，我们可以预测用户的消费场景，进而基于场景做交叉销售和向上销售。

总结

外卖的营销特征，跟其他行业的主要区别在于：

外卖是一个高频的业务。由于用户的消费频次高，用户生命周期的特征体现较显著。运营可以基于用户所处生命周期的阶段制定营销目标，例如用户完成首购后的频次提升、成熟用户的价值提升、衰退用户的挽留以及流失用户的召回等。因此用户的生命周期是一个基础画像，配合用户基本属性、偏好、消费能力、流失预测等其他画像，通过精准的产品推荐或者价格策略实现运营目标。

用户的消费受到时间、地点等场景因素驱动。因此需要对用户在不同的时间、地点下消费行为的差异做深入了解，归纳不同场景下用户需求的差异，针对场景制定相应的营销策略，提升用户活跃度。

另外由于外卖是一个新鲜的事物，在用户对一些新品类和新产品缺乏认知的情况下，需要通过技术手段识别用户的潜在需求，进行精准营销。例如哪些用户可能会对小龙虾、鲜花、蛋糕这样的相对低频、高价值的产品产生购买。可以采用的技术手段包括用户分群、对已产生消费的用户做 look-alike 扩展、迁移学习等。

同时我们在制作外卖的用户画像时还面临如下挑战：

1）数据多样性，存在大量非结构化数据例如用户地址、菜品名称等。需要用到自然语言处理技术，同时结合其他数据进行分析。

2）相对于综合电商而言，外卖是个相对单一的品类，用户在外卖上的行为不足以全方位地描述用户的基本属性。因此需要和用户在其他场合的消费行

为做融合。

3）外卖单价相对较低，用户消费的决策时间短、随意性强。不像传统电商用户在决策前有大量的浏览行为可以用于捕捉用户单次的需求。因此更需要结合用户画像分析用户的历史兴趣、以及用户的消费场景，在消费前对用户做适当的引导、推荐。

面临这些挑战，需要用户画像团队更细致的数据处理、融合多方数据源，同时发展出新的方法论，才能更好地支持外卖业务发展的需要。而外卖的上述挑战，又分别和一些垂直领域电商类似，经验上存在可以相互借鉴之处。因此，外卖的用户画像的实践和经验累积，必将对整个电商领域的大数据应用作出新的贡献。

旅游推荐系统的演进

郑刚

背景

度假业务在整个在线旅游市场中占据着非常重要的位置，如何做好做大这块蛋糕是行业内的焦点。与美食或酒店的用户兴趣点明确（比如找某个确定的餐厅或者找某个目的地附近的酒店）不同，旅游场景中的用户兴趣点（比如周末去哪儿好玩）很难确定，而且会随着季节、天气、用户属性等变化而变化。这些特点导致传统的信息检索并不能很好的满足用户需求，我们迫切需要建设旅游推荐系统（本文中度假 = 旅游）。

旅游推荐系统主要面临以下几点挑战：

1. 本异地差异大。在本地生活场景中用户需求绝大部分集中在本地，而在旅游场景中超过 30% 的订单来自于异地请求，即常驻城市为 A 的用户购买了城市 B 的旅游订单。外地人浏览北京时推荐故宫、长城没有问题，北京人浏览时推荐北京欢乐谷、野生动物园更为合适。
2. 推荐形式多样。除了景点推荐外，还有跟团游、景酒套餐的推荐。景点下有大量重复相似的门票，不适合按 Deal（团购单）样式展示；跟团游、景酒套餐一般会绑定多个景点，又不适合按 POI（门店）样式展现。
3. 季节性明显。比如，冬季温泉、滑雪比较热销，夏季更多人选择水上乐园。
4. 需求个性化。比如，亲子类用户和情侣类用户的需求会不太一样，进一步细分，1~4 岁、6 岁以上亲子类用户的需求也会有所差别。

针对上述问题我们定制了一套完整的推荐系统框架，包括基于机器学习的召回排序策略，以及从海量大数据的离线计算到高并发在线服务的推荐引擎。

策略迭代

推荐系统的策略主要分为召回和排序两类，召回负责生成推荐的候选集，排序负

责将多个召回策略的结果进行个性化排序。下文会分别对召回和排序策略的迭代演进过程进行阐述。

召回策略迭代

我们从 2015 年底启动了旅游推荐系统的建设，此时度假业务有独立的周边游频道首页，其中猜你喜欢展位的推荐策略由平台统一负责，不能很好的解决旅游场景中的诸多问题。下文会按时间顺序来阐述如何利用多种召回策略解决这些问题。

热销策略 1.0

旅游推荐第一版的策略主要基于城市热销，不同于基于 Deal 所在城市统计分城市热销，这一版策略基于用户常驻城市来统计，原因是不同城市的旅游资源分布各异，存在资源缺乏（客源地）、旅游资源丰富（供给地）以及本地人到周边城市游玩的需求。即对于每个城市，都有其对应的“城市圈”Deal 库，比如：廊坊没有滑雪场，但常驻城市为廊坊的用户经常购买北京的滑雪场，因此当廊坊用户在当地浏览周边游频道时会推荐出北京的滑雪场。

在具体实现时考虑旅游产品随季节性变化的特性，销量随时间逐渐衰减，假定 4 周为 1 个变化周期，Deal 得分公式为：deal_score = ∑ ((count(payorder) * α ^ i)，其中 count(payorder) 指该 Deal 相应日期的支付订单数，i 指该日期距今的天数，取从 1 到 28 的整数，α 为衰减系数 (<1)，Deal 得分为一定周期内每日销量得分的总和。

根据上述公式对每个城市都能统计 Top N 热销 Deal，再根据 Deal 关联 POI 过滤离当前浏览城市 200km 以外的 Deal，比如：在浏览北京时推荐上海迪士尼门票不太好，不符合周边游的定位。

这一阶段还尝试了热门单、低价单、新单策略。新单和低价单比较好理解，就是给这些 Deal 一定的曝光机会。热门单跟热销单类似，统计的是 Deal 浏览数据，热门单召回的 Deal 跟热销策略差异不大。但由于推荐的评估指标是访购率（支付 UV/推荐 UV），这些策略的效果不及热销，都没有上线。

另外还初步尝试了分时间上下文的推荐，比如：区分工作日 / 非工作日，周一至

周四过滤周末票、周五至周日过滤平日票，不过随着推荐 POI 化而下线了。

这一阶段的策略主要有两个创新点：

1. 基于用户常驻城市统计热销，突破了 Deal 所在城市的限制，在本地能推荐出周边城市的旅游产品。
2. 通过销量衰减，基本解决了季节性问题。

推荐 POI 化

每个景点下通常会有多个票种，每个票种下通常会有多个 Deal，比如：故宫门票的票种有成人票、学生票和老人票，成人票下由于 Deal 供应商不同会有多个 Deal，这些 Deal 的价格、购买限制可能会有所区别。如果按 Deal 样式展示，可能故宫成人票、学生票都会被推荐出来，一方面大量重复相似 Deal 占据了推荐展位，另一方面 Deal 摘要信息较长，不利于用户决策。因此 2016 年初启动了推荐 POI 化，第一版的 POI 化方案基于 Deal 关联的 POI 做推荐，即故宫成人票是热销单，实际推荐展示的故宫 POI。 这个方案有两个问题：

1. 推荐的 Deal 有可能来自同一个 POI，POI 化需要去重。如果推荐展位有 30 个，候选推荐 Deal 的数量肯定要 >=30，但也可能出现 POI 化后不足 30 个情况。
2. 由 Deal 反推的 POI 销量并不准确，POI 实际销量需要更精确的统计方法。

因此在 2016 年 Q2 上线了基于 F 值的 POI 热销策略，F 值是美团点评内部的一种埋点追踪方法，可以简单理解为：用户在浏览 POI 详情页时会在埋点日志的 F 值记录 POI ID，然后这个标记会一直带到订单中，这样就能相对准确计算每个订单的 POI 归属。

热销策略 2.0

1.0 版热销策略的主要问题是只考虑常驻城市的用户在当地的购买偏好，简而言之，只解决了上海人在浏览上海时的推荐问题，北京人在浏览上海时推荐的结果跟上海人推荐的一样。放大看是本异地场景的问题，本异地场景的定义见下表。

2.0 版热销策略对本异地订单分别统计，当某个用户访问美团时先判断该用户是本地还是异地用户，再分别召回对应的 POI，对于取不到常驻城市的用户默认看做是本地请求。从推荐结果看北京本地人爱去欢乐谷，外地人到北京更爱去长城、故宫。

分类	场景	召回策略
本地需求	浏览城市＝常驻城市 （示例：北京人浏览北京）	当地用户购买的热销POI （POI所在城市不一定等于浏览城市）
异地需求	浏览城市!＝常驻城市 （示例：重庆人浏览北京）	异地用户购买的热销POI （所有非北京人购买的热销POI）

这一版本中继续尝试了分时间上下文的细分推荐，统计一段时间内每天各小时的订单分布，其中有 3 个鞍点，对应将一天分为早、中、晚 3 个时间段，分时间段统计 POI 热销。从召回层面看 POI 排序对比之前变化比较大，但由于下文中 Rerank 的作用，对推荐整体的影响并不大。

用户历史行为强相关策略

热销策略虽然能区分本异地用户的差异，但对具体单个用户缺少个性化推荐，因此引入用户历史行为强相关的推荐策略。取用户最近一个月内浏览、收藏未购买的 POI，按城市分组，按 POI ID 去重，越实时权重越高。

基于地理位置的推荐策略

上文的策略要么是有大量 POI 数据，要么是有用户数据，如果用户或 POI 没有历史行为数据或比较稀疏，上述策略就不能奏效，即所谓的“冷启动”问题。在移动场景下通过设备能实时获取到用户的地理位置，然后根据地理位置做推荐。具体推荐策略分为两类：

1. 查找用户实时位置几公里范围内的 POI 按近期销量衰减排序，取 Top POI 列表。
2. 查找用户实时位置几公里范围内的用户群，基于其近期发生的购买行为推荐 Top POI。比如用户定位在回龙观，回龙观附近没有 POI，但回龙观的用户会购买一些应季热门 POI。

地理位置推荐策略需要过滤用户定位城市跟客户端选择城市不一致的情况，比如：定位北京的用户在浏览上海时推荐北京周边 POI 不太合适。

协同过滤策略

协同过滤是推荐系统中最经典的算法，相对于历史行为强相关策略，对用户兴趣、POI 属性相当于是做了抽象和泛化。协同过滤算法主要分为 ItemCF 和 UserCF 两类，我们首先实现了 ItemCF，主要原因是：

- 性能：美团旅游 POI 数量远小于用户数，协同过滤算法核心的地方是需要维护一个相似度矩阵（Item/User 相似度），维护 POI 相似度矩阵比维护用户相似度矩阵代价小得多；
- 实时性：用户有新行为，一定会导致推荐结果的实时变化；
- 冷启动：新用户只要对一个 POI 产生行为，就可以给他推荐和该 POI 相关的其他 POI；
- 可解释性：利用用户的历史行为给用户做推荐解释，可以令用户比较信服。

基于 POI 浏览行为的协同过滤

根据 UUID 维度的浏览数据来计算 POI 之间的相似度，浏览行为比下单、支付行为更为稠密。时间窗口取一个月的数据，理论上只要计算计算能力不是瓶颈，时间窗口应该尽可能的长。相似度公式定义如下：

$$w_{i,j} = \frac{|N(i) \bigcap N(j)|}{\sqrt{|N(i)||N(j)|}}$$

分母$|N(i)|$是浏览 POI i 的用户数，分子$|N(i)\cap N(j)|$是一个月内同时浏览过 POI i 和 j 的用户数。在计算完 POI 相似度后，再通过如下公式计算用户 u 对 POI j 的兴趣：

$$p_{uj} = \sum_{i \in N(u) \bigcap S(j,K)} w_{ji} r_{ui}$$

这里$N(u)$是用户浏览或购买过的 POI 的集合，$S(j,K)$是和 POI j 最相似的 K 个 POI 的集合，w_{ji}是 POI j 和 i 的相似度，r_{ui}是用户 u 对 POI i 的兴趣。协同过滤可以看做是用户历史行为强相关策略的泛化，最终的推荐结果示例：用户浏览了“北京欢乐谷”，推荐出“北京海洋馆”、“香山公园”。

用户对 POI 的行为表每天离线生产好后更新，相当于只有当天之前的数据，缺少对用户当天实时行为的反馈，因此增加基于用户实时 POI 行为的协同过滤推荐，复用上文中的 POI 相似度计算结果。

基于用户搜索行为的协同过滤

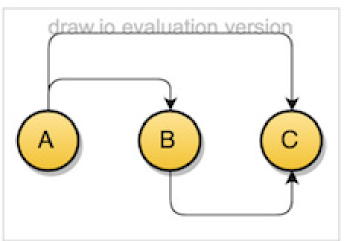
搜索行为是一种强意图行为，旅游较多订单来源于搜索入口，相当比例的搜索用户没有点击任何 POI，基于用户搜索行为的推荐可以作为 POI 浏览推荐的一种补充。首先构造 Query 和 POI 的相似度矩阵，利用用户搜索 Query 后 10 分钟内浏览的 POI 构造对，相似度算法跟 POI 相似度公式一致。

具体实现时以 Query+City 为 Key，原因是旅游场景中存在部分全国连锁 POI，如：欢乐谷、方特，如果只以 Query 为 Key，则跟“欢乐谷”Query 最相关的 POI 可能是“北京欢乐谷”，那用户在深圳搜索“欢乐谷”后会推荐出北京欢乐谷，不符合用户需求。

相似度改进

上述相似度计算公式有两个改进点：一是未考虑用户行为的先后顺序，比如用户先后浏览了 POI ，之前会两两计算相似度，实际只用计算 A 和以及 B 和 C 的相似度即可，因为用户是先浏览了 A 再浏览了 B，所以浏览 A 时可以推荐 B，但浏览 B 时推荐 A 不一定合适。二是未考虑 POI 之间的时间序列跨度，理论上 A 和 B 的相似度

应该高于 A 和 C 的相似度。

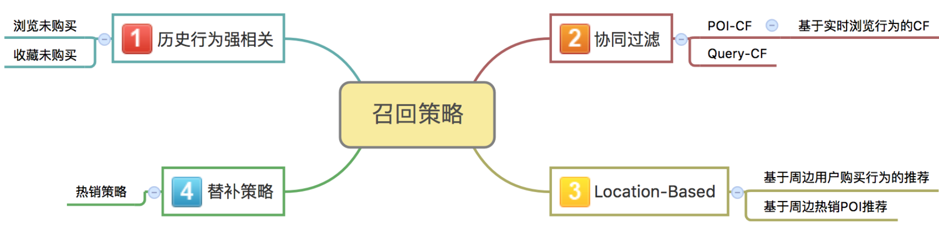


改进后的相似度公式如下，其中 l 表示 POI i 和 j 的序列跨度长度，N_{ijl} 是 POI i 和 j 序列长度为的次数，α 是序列跨度的衰减系数 (<1)：

$$w_{ij} = \frac{\sum_l N_{ijl} * \alpha^l}{\sqrt{|N(i)||N(j)|}}(i < j)$$

召回策略全景视图

经过一年的迭代，目前线上在线的召回策略如下图，此外还尝试了基于 ALS 的矩阵分解，但推荐的结果比较冷门，可解释性较差；另外启动了基于用户标签的推荐，对用户和 POI 都打上相应的属性标签，可以直接单维度标签进行推荐，比如：给亲子类用户推荐亲子类 POI，也可以把标签当做维度，多维度计算用户和 POI 的相关性。



每类召回策略的结果都需要做过滤，过滤策略主要有几类：

1. 黑名单过滤。如源头有脏数据或需要人工干预的 Case。
2. 无售卖 POI 过滤。即过滤没有售卖 Deal 的 POI。

3. POI 距离过滤。过滤据当前浏览城市几百公里外的 POI。

4. 非当前城市过滤。过滤非当前浏览城市的 POI。

5. 已购买 POI 过滤。

其中前 2 类过滤策略对所有召回策略是通用的，都需要做，黑名单过滤考虑到数据更新的实时性，在线上处理，其他过滤策略可以在离线数据层统一处理。后 3 类只有特定召回策略需要，因为依赖用户请求，只能在线上处理，具体规则如下：

召回策略	过滤规则
热销策略	POI距离过滤
历史行为强相关	已购买POI过滤 非当前城市过滤
Location-Based	非当前城市过滤
ItemCF	已购买POI过滤 非当前城市过滤

排序策略迭代

每类召回策略都会召回一定的结果，这些结果去重后需要统一做排序。在早期只有热销策略一个时不需要 Rerank，直接根据热销得分来排序，加入历史行为强相关和 Location-Based 策略后也是按固定展位交叉展示的，比如：第 1、3、5、7 位给历史行为强相关策略，第 2、4、6、8 位给 Location-Based 策略。

在 2016 年 Q1 初尝试了第一版的 Rerank 策略，当时推荐样式还是 Deal，因此排序对象也是 Deal，主要特征是 30/180 天的销量 / 评分数据，因为考虑的特征比较少，上线后效果并不明显。

在 Q2 初由于基本完成了 POI 化展示，排序对象变成 POI，主要特征包括销量、

评分、价格、退款数据，上线后效果仍不明显。

因为推荐列表页跟筛选列表页类似，在 Q2 中期尝试直接接入筛选 Rerank，但效果不太理想。随后基于推荐的数据样本重新进行了训练，并新增了一些特征，特征上大致分为以下几类：

特征维度	特征名称	说明
上下文	HOUR_OF_DAY	一天中的第几小时
	DAY_OF_WEEK	一周中的第几天
	CITY_ID	客户端选择城市id
	DISTANCE	用户和POI的距离
POI	REC_POI_CTR_DAY7 ...	POI 7天的点击率
	POI_ALLCATE_PAY_F_CNT_DAY7 ...	POI 7天的支付数据
	POI_COMMENT_CNT_DAY7 ...	POI 7天的评分数

从上表看在销量和评价基础上主要新增了上下文特征、距离特征和访购相关特征，注意到 HOUR_OF_DAY、DAY_OF_WEEK、CITY_ID 并没有采用 one-hot 编码，在线上实验 one-hot 编码效果并不优于直接使用原始值。可能的解释是 HOUR_OF_DAY 离散值可以用于树模型来分类，比如：0~11 点可以表示上午、12 点 ~18 点可以表示下午、19 点 ~23 点表示夜晚；同理 DAY_OF_WEEK 周一到周四可以认为是平日，周五到周日认为是周末；CITY_ID 可能的解释是 ID 越小，越是开站较早的城市，也是更热门的城市。

模型上取最后一个点击前的样本为候选样本集，以支付为正样本，其他为负样本，正负样本采样比为 1：10。如果不做样本采样，假设每 100 人访问只有 1 个

支付，每次访问列表页假设用户平均能看到 10 个 POI，即正负样本比例大约为 1：1000，样本分布极不均衡，容易导致过拟合。模型训练上采用 XGBoost 算法，上线后点击率和访购率均明显正向，证明了 Rerank 的有效性。

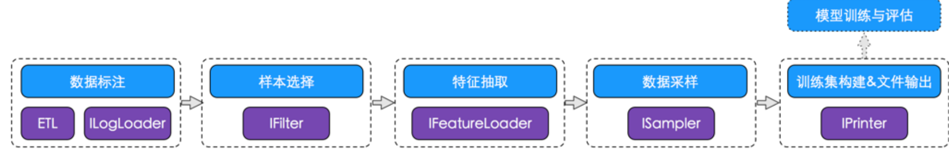
在上述基础上后续又逐步丰富了上下文特征，比如：召回可能触发周边城市圈的 POI，因此增加 POI 是否本城市的特征，另外热销召回策略拆分了本异地，Rerank 也对应增加了用户请求是否本异地特征；增加了 User-POI 组合特征：User 7 天内是否浏览 / 收藏过 POI、实时特征、基于协同过滤的 User-POI 相关性等，跟历史行为强相关、协同过滤的召回策略能相呼应；增加了 POI 静态属性特征，如：星级，另外把 POI 的销量也按本异地进行了拆分。这些特征上线后效果基本都正向，符合预期。

特征维度	特征名称	说明
上下文	SCENE_LR	是本地OR异地用户
	IS_POI_LOCAL	POI是否本城市
User-POI	POI_VIEWED_DAY7	POI 7天内是否被浏览过
	...	
	POI_RT_VIEWED	实时特征：用户最近是否浏览过
	...	
	REC_POI_CF_SCORE	通过POI CF计算出的User和POI的语义相关性
	...	
POI	PLACE_STAR	景区星级
	...	
	POI_SCENE_PAY_F_CNT_DAY7	POI分本异地的销量
	...	（当用户是本地请求时使用本地销量，异地时使用异地销量）

模型上尝试了短周期模型 + 长周期模型的融合，短周期为近期一个月数据，长周期为近期三个月数据。从线上结果看直接用短周期模型效果最好，这可能跟旅游应季变化快有关。除了上述特征外，后续还可以增加 User 个性化特征、天气上下文特征、POI 特征 CTR/CVR 可以拆分本异地等。

排序策略全景视图

推荐的离线训练流程跟搜索、筛选排序保持一致，流程图如下：

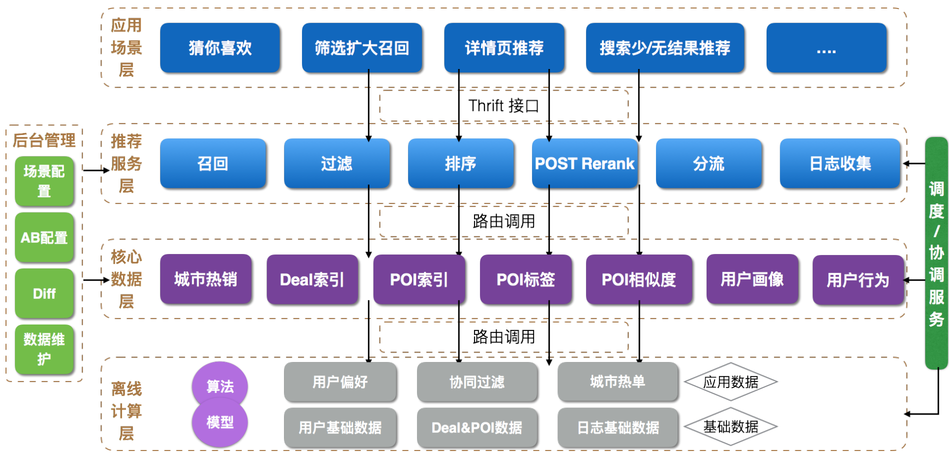


- 首先是数据标注，数据源是原始的样本日志，记录在 Hive 中，输出是 ISample 对象，同时打上 label。另外可能部分特征需要在线上生产并写入样本日志中，比如：实时特征，没办法用离线 ETL 采集；
- 样本选择：对初始样本做过滤，比如：过滤最后一个点击样本之后的数据，输出还是 ISample；
- 特征抽取：在样本中有 POI ID，根据 POI ID 可以抽取 POI 的销量、评价等特征；同理可以根据样本中的 UUID 抽取用户相关特征。这样就生成了带上 Feature 的 Sample；
- 数据采样：按事先定义的正负样本比例对样本进行抽象；
- 训练集构建 & 输出：按 XGBoost 格式输出训练集。

整个训练集的构造过程由 Scala 编写在 Spark 集群上运行，而由于 XGBoost 的 Spark 版本效果不太稳定，在最后的模型训练与评估中使用的 XGBoost 的单机版本，模型的训练参数（迭代次数、树的深度等）一般选取经验值，训练集选一个月的数据，测试集一般选训练集日期后的若干天，离线评估指标主要参考 AUC，离线效果有提升就会上线 ABTest 实验，逐步迭代。

工程架构设计

推荐系统的整体工程架构如下图，从下至上包括离线计算层、核心数据层、推荐服务层和应用场景层，另外是后台配置管理系统和数据调度服务。



离线计算层

离线计算层除了 Rerank 需要的特征和训练日志外，主要包括基础数据和应用数据两类。基础数据中最重要的是 Deal 和 POI 的数据，为了保证数据的准确性和实时性，Deal 和 POI 的数据直接从旅游产品中心去取，通过定时全量拉取并辅以消息队列实时更新。应用数据按生产方式又可以分为三类：

1. Hive ETL 生产的数据：比如 POI 过滤需要用到的离线表（主门店等逻辑），另一大类是统计数据，比如：城市 POI 热销、线路游热销、用户对 POI 的浏览 / 购买行为。
2. Spark 生产的数据：比如：User CF、POI CF、矩阵分解算法等，这类数据生产逻辑复杂，不好直接通过 ETL 计算完成。
3. Storm 生产的数据：用户实时行为在召回、排序都需要用到，目前公司提供统一的实时用户行为数据流 user__action_basic，包括：浏览 / 收藏 POI/Deal、下单、支付、消费、退款，从中过滤出旅游 POI/Deal 的行为即可。

核心数据层

抽象出核心数据层的一个重要原因是需要离线计算工程和线上服务工程复用 DataSet，从供线上使用的存储方式看可以分为三类：

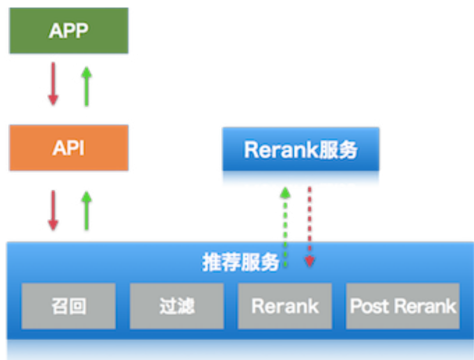
1. 存储在 ElasticSearch（以下简称 ES）中的数据。主要是 POI/Deal 索引，比如：POI 的地理位置、所在城市，当线上需要根据地理位置过滤时可使用 ES 查询，比如：城市圈的距离限制，Location-Based 策略一定距离内的召回。另外对于多维查询场景 ES 也比 KV 存储更为合适。这类数据通过公司统一的任务调度系统来定时调度，通常几小时更新一次。这里为 ES 索引建立一个别名，离线更新索引切别名的指向，保证操作的原子性。
2. 存储在 DataHub 中的数据。DataHub 是酒旅搜索团队开发的一套数据管理系统，集数据存储、管理、使用于一体。目前支持将 Hive 表的数据定期导入，DataHub 内部主要使用 Tair 作为存储，对客户端使用透明，客户端接口支持一维和二维的 Key，接口对应用方基本是一致的，另外应用方也不需要自行维护 Tair 集群配置管理了。DataHub 自带调度功能，通过扫描 HDFS 分区生成后自动写入 Tair。
3. 直接存储在 Tair 中的数据。主要面向 DataHub 还不支持的两类场景，一是实时数据的存储落地，二是 value 直接存储对象，存储为对象的好处是从 Tair 读取出来的对象可直接供线上使用，无需自行序列化和赋值。实时数据无需定时调度，通过 Spark 直接写入 Tair 的数据通常需要依赖上游 Hive 表先 Ready 才能执行，所以通过公司统一的数据协同平台调度。

推荐服务层

服务上下游

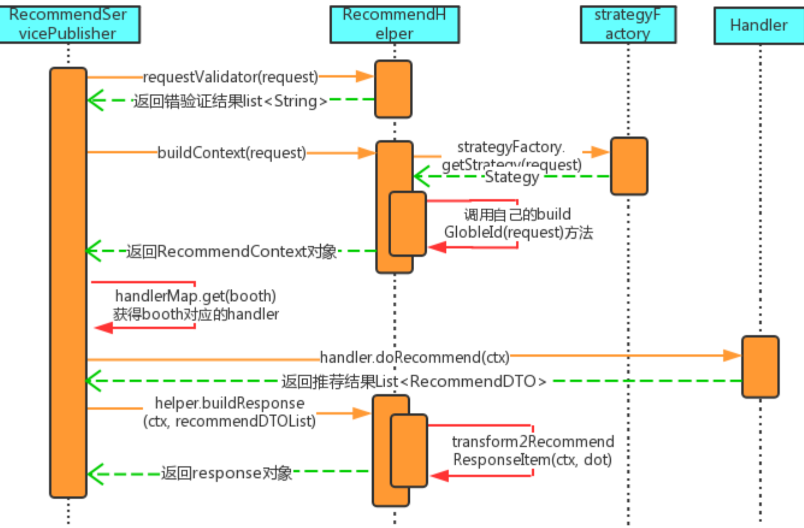
推荐上下游的架构图如下图，客户端向 API 发起调用，API 调用推荐服务拿到推荐的 ID 再添加供 App 展示用的相关字段传会给 App。推荐和搜索没有整合成一个服务的重要原因是推荐的召回策略复杂多样，每次请求可能命中多个召回策略，而搜索单次请求的意图一般比较单一，通常只有一个召回策略。另外推荐服务重点在召回

和过滤，Rerank 调用独立的 rank 服务，原因是推荐 Rerank 和搜索筛选 Rerank 在特征上有很多是可以复用的，比如：用户特征、POI 特征等。



整体流程

推荐服务向下从数十个数据源中获取数据，经过业务逻辑处理后向上支持数十个应用场景，整个调用流程如下：

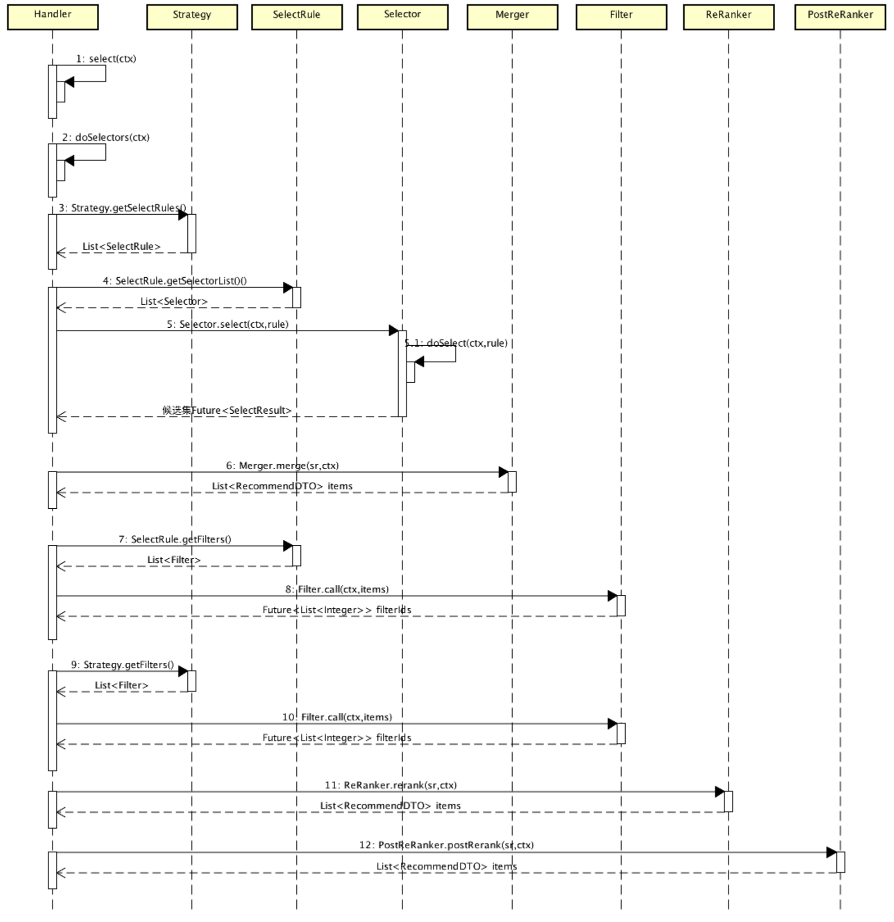


1. RecommendServicePublisher 作为服务的入口，从 Client 接到 Request 请求后首先验证请求是否合法，比如：请求参数中场景 Booth 和 UUID 不能为空。

2. 构造请求上下文 Context，其中会生成唯一的 global ID 标识一次请求，根据 UUID 查询用户画像服务获取常驻城市，根据定位的经纬度查询定位城市，以及根据 ABTest 分流配置获取处理请求的召回排序 Strategy。
3. 根据请求场景的 Booth 获取对应的 Handler，默认使用统一的 AbstractHandler 即可，包括召回、过滤、rerank、post rerank。
4. 对 Handler 返回的结果做包装，增加召回和排序策略名称、得分等，最终返回给 Client。

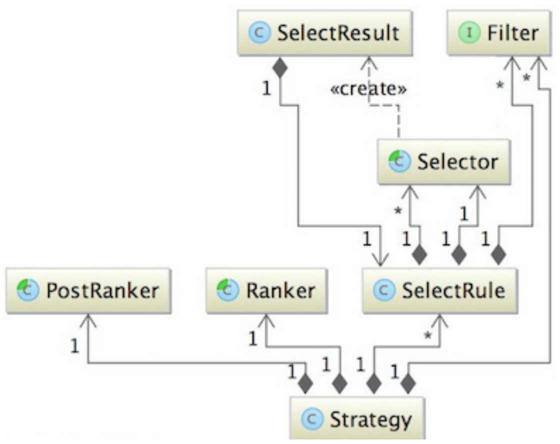
核心流程与模型

Handler 是整个流程的核心，其调用流程如下：



1. Handler 根据不同的 Strategy 获取对应的 SelectRule 集合，一个场景 Booth 可能对应多个 Strategy，跟 ABTest 对应，比如：Baseline 就是一个 Strategy。每个 Strategy 可能有多个 SelectRule，比如：Baseline 策略由历史行为强相关 SelectRule、Location-Based SelectRule、热销 Rule 等组成。
2. 召回：每个 SelectRule 又对应多个 Selector，多个 Selector 通过线程池并发获取结果，比如：Location-Based Rule 可以细分为基于周边热销 POI 召回和基于周边用户购买 POI 召回。Selector 可再做抽象，比如：分本异地场景的城市热销策略，美团和点评双平台都需要，只是数据源稍有不同，另外对于从 ES 和 DataHub 获取的数据可以加 Cache。
3. Merge 去重：多个召回策略的结果需要 Merge 去重，比如早期没有 Rerank 时 Location-Based 策略固定在 2、4、6 位。
4. 过滤：具体有两级过滤，一级是针对 SelectRule 的，比如：针对历史行为强相关策略中基于浏览行为和收藏行为召回的结果都需要过滤用户已购买过的 POI；另一级是针对所有策略通过的过滤，比如黑名单、旅行社代理商。
5. 重排序：对于 POI 列表调用 POI Rerank 服务，对于 Deal 列表调用 Deal Rerank 服务。
6. PostRerank：一般用于处理广告运营的需求和人工干预的 Case。

核心的对象模型如下图：



监控降级

监控分为离线监控和实时监控两部分，离线监控使用 Falcon 来监控以下几类指标：

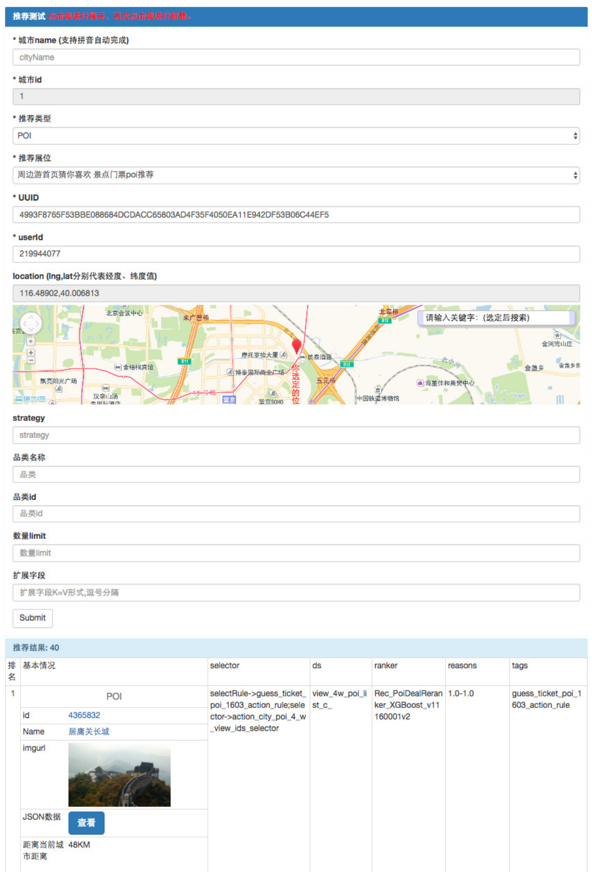
- JVM 监控：比如 FullGC 次数、内存使用情况、Thread block 情况
- ES 监控：ES 查询次数和平均响应时间
- 业务监控：各接口、各策略的请求次数和平均响应时间

实时监控接入公司统一的实时数据统计平台，可以分时、分多粒度统计各 Booth 的请求次数和响应时间。

降级主要通过 Hystrix 来实现，比如：调用 Rerank 服务在一定时间内响应时间超过设定的阈值，则直接熔断不请求 Rerank 服务。

工具化

推荐服务开发了 Debug 工具，输入支持城市、展位、UUID、经纬度等参数，输出展示了 POI/Deal 的头图、标题、和用户的距离、召回排序策略与得分等。方便 PM 和 RD 测试、定位追查 Case。



应用场景

推荐系统支持了美团 / 点评共 20 个应用场景，主要场景是周边游频道首页猜你喜欢，其召回策略在上文中已有阐述，这里重点阐述其他几类推荐场景：

跟团游推荐

跟团游 Deal 一般会绑定多个景点，不适合按 POI 样式展现，因此采用 Deal 形式展现，召回策略跟热销 POI 策略类似，区分本异地，从结果看北京本地人会推荐“古北水镇一日游”，外地人浏览北京时会推荐“故宫、长城一日游”。

筛选异地召回

用户在筛选酒店时会先选择入住城市再筛选该城市的酒店 POI，而周边游存在客源地旅游资源不丰富的问题，筛选时需要突破选择城市限制，能够推荐出周边城市的热门 POI，筛选异地召回上线后增加了一定比例的订单，是对本地召回的有效补充。

筛选主题标签挖掘

即为 POI 打标签，用户可以用这些标签进行筛选，比如：附近热门、近郊周边、周末去哪、亲子同乐、夜场休闲。每个标签都可以定义一套挖掘方法，比如：“亲子同乐”有以下几类方法：

- POI 下有亲子票种
- Deal 标题包含“亲子”
- 同一 POI 下同时包含“成人票”和“儿童票”
- 用户画像为“亲子”的用户最近一个月购买的 POI

上述挖掘方法偏规则，后续希望能通过半 / 无监督方法，挖掘 POI 描述和评论，自动为 POI 打标。

搜索少 / 无结果推荐

搜索少结果推荐是指当搜索结果 POI 类聚结果数 =1 时，为丰富页面内容给用户提供推荐信息。这里重点利用搜索的 POI 结果根据 POI CF 触发推荐，以及利用搜索 POI 的品类进行同城市同品类推荐。

搜索无结果推荐可以直接统计搜索 Query 后一定时间内用户浏览的 POI 做推荐，但这个策略的覆盖面有限，进一步可以计算一段时间内的 Query CF，然后做协同推荐；另一方面可以通过意图识别判断 Query 中是否有品类词，触发同品类推荐。

酒旅交叉推荐

目前只实现了酒店和旅游之间的交叉推荐，当用户在酒店频道搜索时先判断 Query 是否旅游意图，其中重点分析两类意图：一是景点 POI 意图，推荐该景点几公里范围内的 POI；二是品类意图，比如：温泉、滑雪，会推荐用户定位附近该品类的热销 POI。

在酒店 POI 详情页会获取酒店 POI 的地理位置，推荐酒店附近的景点。对于异地用户浏览酒店时都会触发景点推荐，对于本地用户只有在浏览郊区酒店时会触发旅游推荐，这是假设本地用户在浏览市区酒店时旅游度假的意图可能不明显。

除了在各类推荐场景的应用，这些策略在运营上也有应用尝试，比如：用户浏览或购买过 POI 后根据 POI CF 给用户 PUSH 相似的 POI，实验证明推荐策略的 PUSH 点击率要高于平均水平。

未来的挑战

经过一年多的迭代优化，周边游频道内相当比例的订单来自推荐，线上支持了 20 个左右的推荐场景，很多推荐策略被作为特征加入搜索、筛选 Rerank，有明显正向效果，在用户运营上也有了初步的探索。基于目前的推荐系统本身还有不少优化点：

- 召回策略：策略的广度和深度都有不少提升空间，广度方面可以继续探索矩阵分解 FFM、User CF、基于用户画像的推荐、图挖掘；深度方面尝试 LLR 等多种相似度计算方法、以及多时间 / 多用户维度改进召回策略。数据上可以扩大到酒店甚至美团全平台的用户数据，另外对策略的离线实现还要更模块化、抽象化，比如：相似度改进算法在一处场景验证有效，可快速推广上线到其他场景
- 排序策略：特征工程方面可以增加 User 个性化特征、天气 /Listwise 上下文特征等，模型上可以尝试 DNN 等方法，评估指标可以从访购率改进成访消率（消费 UV/ 访问 UV），另外对美团 / 点评双平台可以定制不同的特征数据和排序策略
- 工程架构：搜索少 / 无结果推荐从搜索工程迁移到推荐工程，另外对核心数据层存储方式的边界划分，线上服务层的缓存、Selector/Rerank 降级、Filter/Merge 逻辑梳理等需要做“轻量重构”

- 应用场景：除了在酒店购买前的交叉推荐外还可以增加购买后的推荐，以及和机票、火车票大交通相关的交叉推荐，在旅游内部可以探索更多的场景化建设，比如：亲子游、情侣游

跳出目前单一的以 POI/Deal 列表为主体的推荐形态看，可以从用户、场景、内容、触达方式四个方面看如何做好旅游推荐：

用户需求

首先考虑用户是谁？要满足用户的什么需求？这里可以利用美团 / 点评的数亿用户，打“人群标签”，是一二线城市高端品质女用户、勤俭住宿的中年大叔还是三线城市实惠型年轻妈妈。然后分析这些人群背后的需求，是本地休闲用户、差旅用户还是高频度假用户，不同用户的需求是不一样的。

场景划分

当知道用户后需要知道用户的场景是什么？可以从四个维度定义场景：时间、位置、行为、渠道。

时间很好理解，当用户在周四周五搜索“滑雪场”，会被认为是休闲度假周末用户，可以协同推荐北京郊区的滑雪场。

地理位置是核心要素，要根据用户的常驻城市和客户端选择城市来判断是本地还

是异地需求，对于异地的差旅用户可以推荐商务型的酒店。

行为是用户需求最直接的反应，比如：用户搜索“古北水镇”，不管用户后续是否有浏览行为，都可以推荐古北水镇相关的酒店和景点门票。

渠道包括美团 / 点评双平台 App、i 版、PC 等多个终端，以美团 App 为例，周边游、酒店、机票 / 火车票频道的用户特征都不一样，比如：大交通频道最常见的是差旅用户、周边游频道更多是本地度假休闲的人群。

内容形态

知道了用户是谁以及处于什么场景，要考虑提供什么样的内容产品？对于美团来说核心是交易，内容不是最核心的目标，但内容是一个非常好的引流措施。以本地场景为例，可以加强场景建设，比如：亲子、团建、温泉等；异地行前场景可以加强目的地、点评游记攻略、酒店交通行程安排等内容建设。

触达方式

除了目前的搜索推荐外，还可以增加定向投放、内容引导、广告植入、活动运营等多种触达方式。

总之旅游推荐问题复杂多样，需要从度假出行六要素：吃、住、行、游、购、娱综合考虑和规划，对产品形态、业务策略、技术架构都还有很大的挑战和机遇。

作者简介

郑刚，美团点评高级技术专家。2010 年毕业于中科院计算所，2011 年加入美团，参与美团早期数据平台搭建，先后负责平台、酒旅数据仓库和数据产品建设，目前在酒旅事业群数据研发中心，重点负责酒店旅游场景下的搜索排序推荐、数据挖掘工作，致力于用大数据和机器学习技术解决业务痛点，提升用户体验。

美团点评旅游搜索召回策略的演进

郑刚

背景

美团点评作为最大的生活服务平台，有丰富的品类可供用户选择，因此搜索这个入口对各业务的重要性不言而喻，除了平台搜索外，业务搜索系统的质量和效果对用户体验、商家曝光、平台交易也有着关键作用。

相对美团点评平台的 O2O 检索，旅游搜索系统主要面临以下几点挑战：

- 本异地差异大。在本地生活场景中用户的搜索需求往往集中在本城市内，而在旅游场景特别是行前场景用户会先搜索异地的 POI（门店），比如常驻城市为北京的用户在去上海之前可能会先搜索“东方明珠”、“迪士尼”了解相关信息。
- 搜索意图多样，不同意图的展现形式可能不同。搜“故宫”、“故宫成人票”是景点门票意图，搜“北京”、“云南”是行政区意图，搜“水上乐园”、“滑雪场”是品类意图，搜“上海到南京”、“一日游”是线路游意图。
- 底层脏数据多。旅游早期由于上单审核不严等原因，会出现“真人 CS”Deal（团购单）下挂在“故宫博物馆”POI 的情况，按照平台的检索策略，搜“真人 CS”时会展现“故宫”的 POI，导致大量误召回。

针对上述问题，我们建设了一套相对完整的搜索系统，包括检索召回、查询分析、智能排序和业务应用几部分，本文将重点介绍搜索召回（检索召回、查询分析）的策略演进过程。

评价指标

我们在 2015 年 Q2 启动了旅游搜索系统的建设，此时旅游业务有独立的周边游频道，其中的搜索策略由平台统一负责，不能很好的解决旅游场景中的诸多问题。为了解决这些问题，我们首先需要确定搜索的评价指标：

- 访购率：支付用户数 / 搜索访问 UV，这个是评估搜索效果的主指标。美团点评是一家电商公司，营业收入是核心指标，以搜索为例，用户行为链条包括搜索 Query-> 点击搜索结果列表页中的 POI/Deal 等 -> 下单支付 -> 消费，最后计算消费收入。如果只看点击率，关注的链条太短，没有反映交易属性；如果看最终的收入结果，部分因素（消费受产品的购买限制、退款条件等影响，收入又跟商户拓展人员谈单的毛利等相关）非搜索可控。因此以访购率作为搜索的核心指标跟美团点评的业务特点最为匹配。

$$\text{每搜索用户收入}=\underbrace{\frac{\text{点击用户数}}{\text{搜索用户数}}\times\frac{\text{支付用户数}}{\text{点击用户数}}}_{\text{访购率}}\times\frac{\text{消费用户数}}{\text{支付用户数}}\times\text{每用户消费收入}$$

- 点击率：点击 PV/ 搜索 PV（Page View）。部分景点由于商户拓展人员没有谈单或者是免费景点等原因导致没有门票或线路游 Deal 可售时，访购率为零，但用户可能需要了解景点相关信息，这时点击率是重要的辅助评价指标。一个相关的指标是有点行为比，以搜索请求量为统计口径。
- 无结果率：无结果请求数 / 搜索请求数，衡量搜索召回质量的重要指标。
- 用户满意度：由产品经理定期人工评测，比如取搜索结果的前 20 条，如果是单景点意图，对应的 POI 能排在首位，排序合理，无重复 POI 则为 1 分；搜索结果满足部分用户需求，存在误召回、排序不合理的情况则为 0.5 分；完全不能满足用户旅游需求，搜索结果没有有效信息则为 0 分。

除了可以用指标评估的问题外，还有一些指标外的问题，比如广告运营、直签门票加权等，这些问题可能跟指标负相关或不好量化评估。指标内的问题又分为两类：一类是算法问题，比如查询意图理解、召回检索策略、个性化排序；另一类是产品和业务问题，比如页面改版、源数据清洗，部分产品问题也需要策略协同解决。

策略迭代方法

明确评价指标后需要找到策略优化的方向和思路，不同于推荐（可以参考作者之

前的文章《旅游推荐系统的演进》)，搜索的 bad case 往往非常明确，因此我们确立了以 case 驱动为主的策略迭代方法。

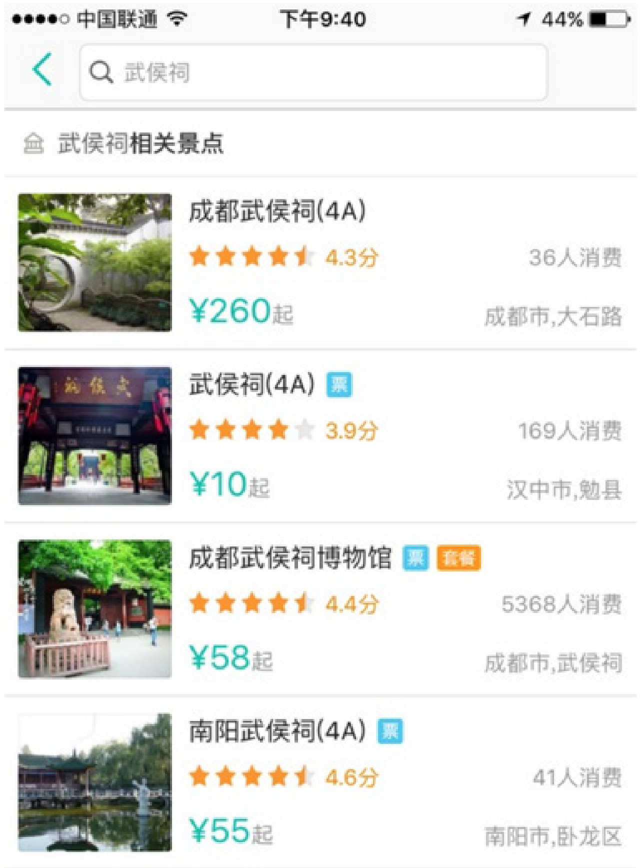
1. 质量评估：定义满意度标准和评估体系，定期（月 / 季度）评估搜索满意度，确定评估样本，了解 Query 需求分布、意图识别准召率、召回及排序情况。
2. 问题分析：对问题进行梳理分类，比如无供给问题、误召回问题、意图识别问题、POI 排序问题、展示问题等，找出主要问题并明确优化方向。
3. 项目开发：评估项目实施的可行性，制定相应的技术方案，配合产品、客户端等其他技术团队联调、测试。
4. 实验迭代：上线 A/B Testing 验证优化效果，根据指标评估项目收益，效果正向则扩量，负向则分析调整或下线，并继续迭代优化。

召回策略迭代

全国召回

2015 年 Q2 启动了第一次周边游频道内的搜索质量评估，其中 Query 搜索无结果影响面非常大，除无供给问题外最重要的一个原因是不支持异地搜索。比如在德州搜索“北京故宫”无结果，进一步分析发现在旅游场景中超过 30% 的订单来自于异地请求，即常驻城市为 A 的用户购买了城市 B 的旅游订单。因此在周边游频道内先

放开了上单城市的召回限制，当用户搜索 Query 时根据 POI 和 Deal 字段匹配召回全国范围内的结果，比如在北京搜“武侯祠”能召回多个城市的结果，全国召回策略上线后无结果率大幅下降。



模块化展示

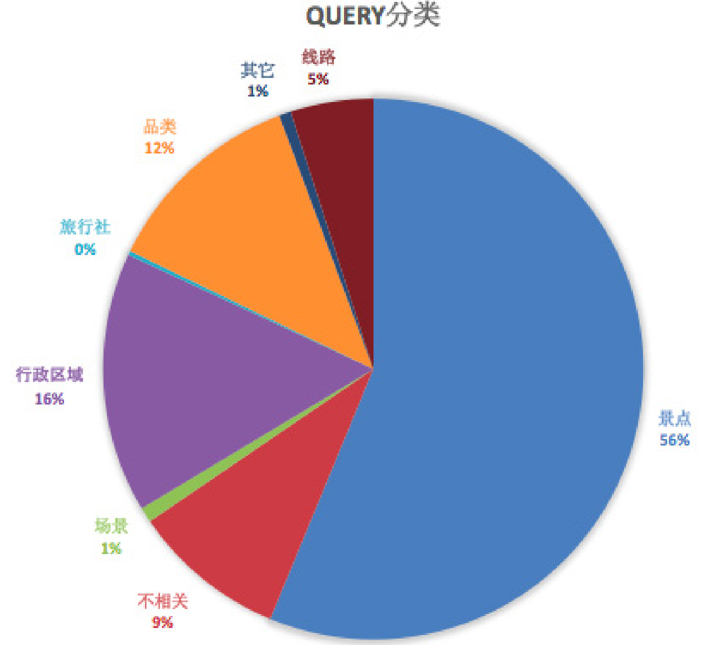
除全国召回外，周边游频道搜索当时仍沿用了美团点评平台的展示及召回机制：

- POI 下挂 Deal 形式展示。
- 通过 POI 及 POI 下挂的 Deal 信息进行召回。

这些机制主要有两个问题：

- 以 POI 为主的展现形式不能很好满足用户的线路游需求，比如用户搜“北京一日游”，返回的是故宫、长城等 POI 结果，用户不能方便找到线路游 Deal。
- 旅游不同于其他品类，Deal 与 POI 不是一一对应关系，尤其是线路游、年票等 Deal 往往关联多个景点，按照平台现有召回策略，会导致大量 POI 被误召回，比如搜“长城”返回“故宫”POI 结果；同时由于上单审核不严等原因，会出现“真人 CS”Deal 下挂在“故宫”POI 的脏数据，也会被误召回。

针对这些问题，在 2015 年 Q3 启动了旅游搜索结果分模块展示的开发，即对用户 Query 进行意图分类，每类意图定制召回策略和展现样式，Query 意图分类如下：



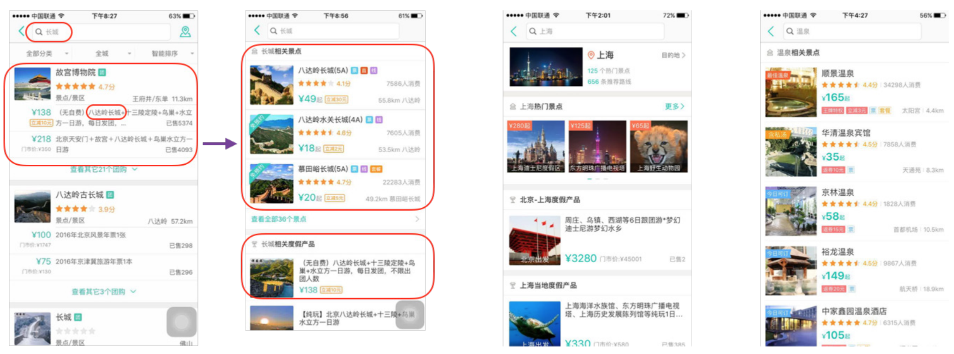
以意图占比为 56% 的景点 POI 为例，当用户搜索“长城”时会展现“长城相关景点”和“长城相关度假产品”两个类聚，景点类聚只在 POI 字段域搜索“长城”，比如 POI 所在城市、名称，这些字段中不包含“故宫”Term，因此不会返回“故

宫”POI。度假产品类聚只限定在非门票 Deal 集合内检索 Deal 标题、品类、商圈等字段，返回的都是跟团游、酒景套餐自由行等线路游信息，方便用户决策。

当用户在北京搜“上海”时是行政区意图，会展示“上海目的地”、“上海热门景点”、“北京 - 上海度假产品”、“上海当地度假产品”4 个类聚，其中“目的地”是为城市专门定制的落地页，“北京 - 上海度假产品”是根据出发地为北京、目的地为上海这两个线路游字段来进行检索。

当用户搜索“温泉”时是品类意图，检索策略跟 POI 景点搜索类似，但会增加品类检索字段。

分模块展示上线后一方面改善了用户体验，另一方面打压了旅游 POI 和 Deal 关联的脏数据，访购率和点击率也大幅提升。

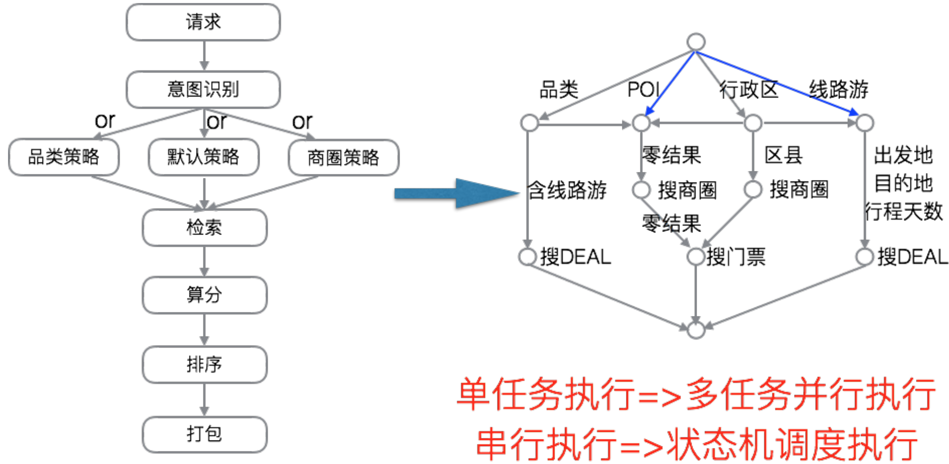


同时为了降低无结果率，在一次召回无结果的基础上增加了二次、三次召回，比如增加 POI 商圈字段。如果二次召回也没有结果，会增加门票 Deal 字段进行三次召回，返回门票结果。

综上可知用户 Query 主要包含景点、行政区、品类、线路游 4 类意图，每类意图又可能展现多个类聚，每个类聚的召回检索策略不同。而早期的技术架构在单次请求下只支持单策略检索，同时在多次召回时只能串行执行，因此需要对检索架构进行升级：

- 由单任务执行变成多任务并行执行，比如搜索“故宫”时需要并发执行 POI 和线路游两个检索策略。

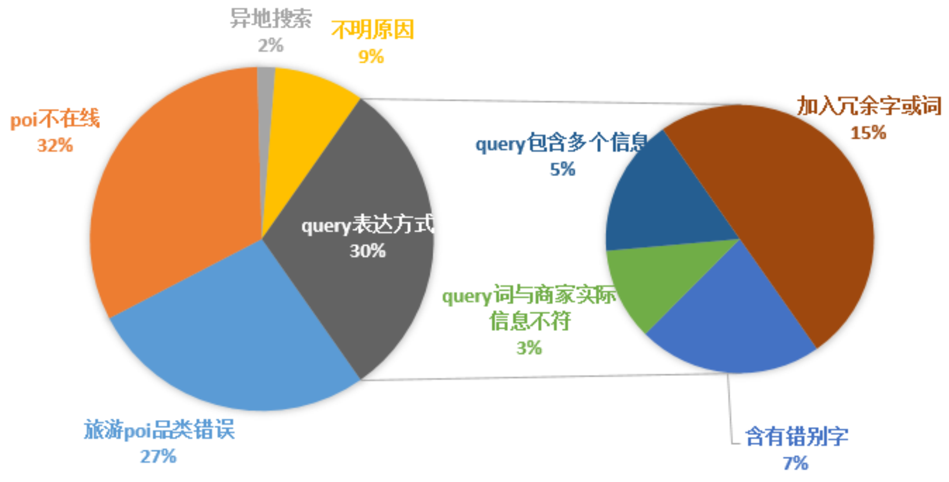
- 由串行执行变成基于状态机的调度执行，比如 POI 策略下一次召回无结果，会增加商圈字段二次召回，再无结果时会基于门票 Deal 字段进行三次召回。



无结果优化

为了进一步降低无结果率，在 2015 年 Q4 对线上 Query 做了一次无结果分析，其中 32% 原因是 POI 不在线（无供给，POI 没有可售 Deal），27% 是 POI 品类错误（即 POI 品类标签不是旅游），这两类问题策略不好解决，剩下 30% 是由于 Query 表达方式多样导致搜索无结果，这些 case 细分原因如下：

- 15% 是 Query 包含冗余词，比如搜“东莞的隐贤山庄”无结果，去掉“的”有结果。
- 7% 是 Query 含有错别字或同义词，比如在北京搜“雁西湖”无结果，用户实际需求是“雁栖湖”。
- 5% 是 Query 包含多个信息，比如搜“北京动物园海洋馆门票”无结果，分别搜“北京动物园”和“北京海洋馆”有结果。
- 3% 是 Query 词与商家实际信息不符，比如在北京搜“798 艺术 3D 体验馆”，搜“活的 3D 博物馆”有结果。



A策略未召回原因分析

丢词 & 查询改写

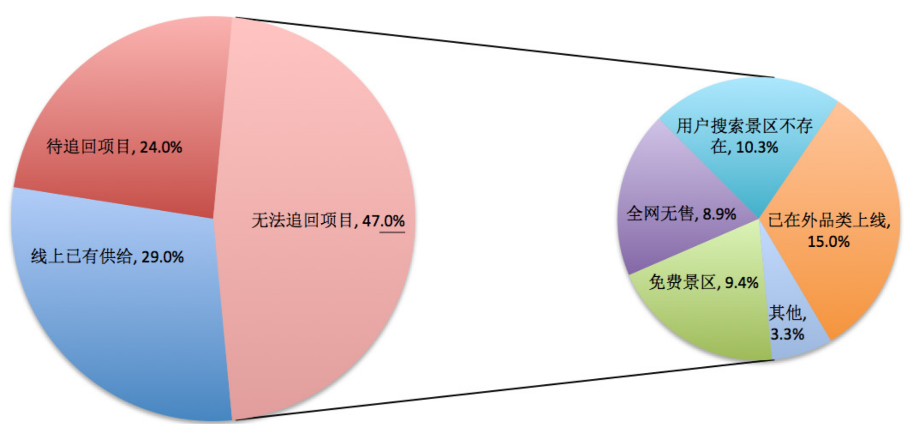
针对上述问题分别定制了以下几类策略：

- 丢词策略：通过挖掘 Query 日志，统计其中的高频停用词，比如的、一张、价格、团购、去哪等，对用户输入 Query 直接丢弃其中的停用词，再进行检索召回。
- Query 纠错 & 同义词改写：统计同一 Session（比如一个小时内）内用户的查询对，选择词频共现比较高的查询对作为候选，再人工审核加入到同义词词典。用户查询，同时用原词和同义词去检索，最后对两者返回的结果取并集。
- 二次召回：在上文中已有提及，即一次召回无结果时扩大检索字段和检索范围。
- 无结果推荐：推荐本身并不能降低无结果率，但在无结果时给用户提供了另外的选择。

无合作 POI 召回

上述策略上线后搜索无结果率又有了大幅下降，但仍有一定的优化空间，2016 年 Q2 启动了新一轮的无结果分析，无结果 case 大致可以分为 3 类：

- 无法追回项目：比如免费景区或全网无售（商户拓展人员无法谈单），这类 case 早期由于评价指标是访购率，搜索并不能召回，但其实对用户体验伤害较大，容易导致用户流失。因此放开一次召回无结果时二次召回无合作 POI，比如搜索“潭柘寺”会返回结果，虽然暂无可售的 Deal，但用户可以浏览 POI 详情页的景区简介、预订须知等。
- 待追回项目：即目前无供给，但可以反馈给商户拓展人员谈单，针对这类 case 建立了搜索反馈商户拓展人员上单的流程，自动生成任务工单并分派商户拓展人员处理，形成无结果反馈的整体闭环。
- 线上已有供给：搜索召回策略问题导致的无结果，分析发现通过丢词可以解决大部分 case。之前的丢词是词表丢词，丢词的范围有限，需要在一次词表丢词的基础上增加基于模型的二次丢词，主要方法是对 Query 做 Chunk 分析，为每个 Term 打上 Chunk 标签，人工定义哪些 Chunk 可以丢弃。



上述策略上线后搜索无结果率横向对比美团点评平台和其他业务基本达到了合理正常的水平。

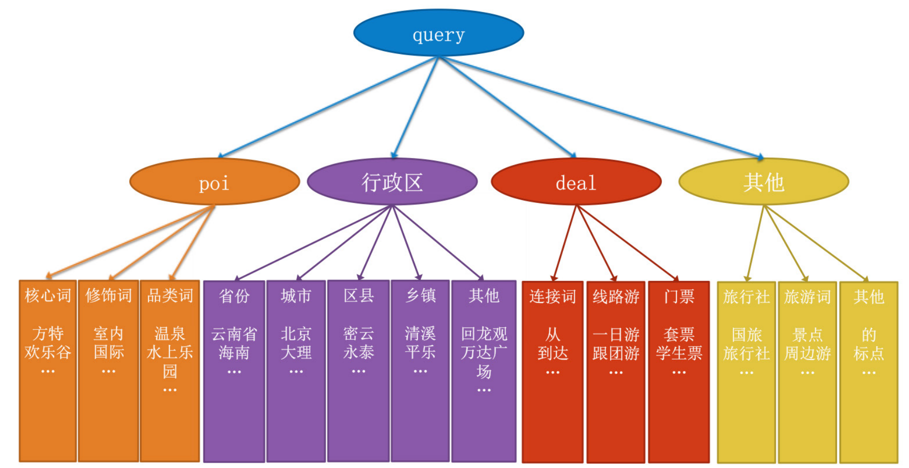
分类意图识别

模块化展示中用到了 Query 意图分类，早期的意图分类使用词表精确匹配的方法，比如搜“大理”和“云南大理”都是行政区意图，其中“云南大理”被切分成

“云南”和“大理”，然后分别和省份、城市词表匹配。词表精确匹配的准确率较高，但召回率不高，比如“大理旅游”、“去大理”跟“大理”都是同一个意图，但无法通过词表精确匹配。如果采用宽泛匹配准确率又不会太高，比如“北海公园”、“中山公园”中都包含行政区，但其实是景点意图。基于此，2015 年 Q4 启动了分类意图识别的优化，首先根据 Query 分布定义了 8 类意图：

- POI：景点、游乐场、度假村等。
- 行政区：国家、省、市、县、区、镇。
- 品类：POI 品类体系中的品类词，以及公园、体验馆等指代词。
- 线路游：一日游、跟团游等。
- 旅游关键词：旅游同义词如旅行、游玩等。
- 旅行社。
- 门票词：门票、套票、成人票等。
- 非旅游：美食、住宿等外品类词，杂质词（的、一张等）。

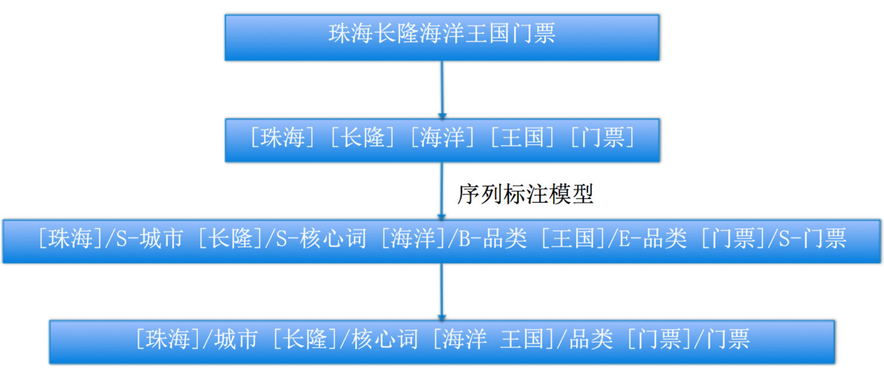
可以通过识别 Query 中 Term 的意图来判定整个 Query 的意图，但上述意图分类对 Term 而言粒度较粗，比如“珠海长隆海洋王国门票”会被切分成“珠海 长隆 海洋 王国 门票”，“珠海”是行政区，“门票”是门票词，“长隆 海洋 王国”整体是 POI，但每个 Term 无法对应到上述分类体系，因此需要设计一套更精细的 tag 体系。



其中将行政区细化到国家、省、市、区县、乡镇和地标商圈等 tag，POI 细化为 POI 核心词、品类词、品类修饰词 tag。在上例中“长隆”是 POI 核心词，“海洋 王国”两个 Term 合并是 POI 品类词，“海洋王国”即是一个 Chunk，Chunk 可以认为是一个语义单元，粒度要大于等于 Term 分词粒度。

基于模型的 Chunk 分析

对于 Query 分词后的 Term，问题转化为识别 Chunk 的边界以及为 Chunk 打上何种 tag，即序列标注问题。Chunk 边界可以采用 BMES（Begin、Middle、End、Single）标记方式，比如“海洋 王国”Chunk 中“海洋”标记是 B，“王国”标记是 E。“珠海”是一个单独的 Chunk，所以整体标记为 S- 城市，同理“长隆”整体标记为 S-POI 核心词，“海洋”标记为 B- 品类词。



Chunk 分析转化为序列标注问题后跟其他机器学习问题类似，需要考虑三方面因素：1）算法模型；2）标记语料；3）特征选取。算法模型方面采用 CRF（条件随机场）模型，其结合了最大熵模型和隐马尔可夫模型的特点，近年来在分词、词性标注和命名实体识别等序列标注任务中取得了很好的效果。

标记语料方面采用一段时间内的搜索日志，分词后对每个 Term 进行标注，但全部采用人工标注费时费力，因此采用词表规则标注然后人工校验，其中重点是收集各 tag 的词表，其中行政区、Deal、旅行社等词表比较好收集，POI 核心词、品类词、修饰词可以通过挖掘和模板匹配来实现，这里以 POI 名称为候选词集合，分词后从后向前匹配，定义模板规则，迭代挖掘品类词、修饰词和核心词。

候选词集合选择 → 品类/后缀词挖掘 → 品类修饰词挖掘 → 核心词挖掘

特征选取方面主要包括三类：

- 边界特征：即可以用于确定 Chunk 边界的特征，包括左右熵、互信息等。
- tag 特征：即可以用于确定 Chunk tag 类别的特征，包括词长度、Term 的 tag 类别等。
- 组合特征：左右熵组合，词的组合等。

模型训练时采用 CRF++，需要将标注语料转成 CRF++ 的训练格式，以 Query“珠海 长隆 海洋 王国 门票”为例，训练语料格式如下：

term	term length	是否有城市Tag	是否有品类Tag		左熵	右熵	label
海洋	6	0	1	...	5	3	B-品类词
王国	6	0	1	...	4	4	E-品类词
珠海	6	1	0	...	3	6	S-城市
长隆	6	0	0	...	3	4	S-核心词
门票	6	0	0	...	7	4	S-门票

最后通过离线训练生成模型供线上使用，对用户输入的 Query，模型会输出分词后每个 Term 的 tag。Chunk 分析是一项非常基础的工作，基于分析的结果可以应用于丢词、Term 重要度、意图识别、Query 改写等。

从 Chunk 分析到意图识别

得到 Chunk 的 tag 后可以制定规则输出整个 Query 的意图，意图之间有优先级顺序：线路游 >POI> 品类 > 门票，比如“北京故宫一日游”是线路游意图，“北京故宫”是 POI 意图，“北京动物园”是 POI 意图，“动物园”是品类意图。

分类意图识别对搜索整个流程都意义重大，召回层面可以分意图定制检索字段、相关性计算等检索策略，Rerank 层面可以分意图优化特征，展示层面可以控制不同

的展现样式。

粗排序改进

除了 Query 分析、检索策略外，粗排序是搜索召回的另一个核心功能。当搜索结果较多时，如果粗排序不合理，会导致部分优质 POI 或 Deal 无法召回，并且这些 case 不好人工干预。因此我们在 2016 年 Q3 启动了粗排序的改进工作，主要包括：

- 距离分分段：计算客户端选择城市中心和 POI 的距离，若距离 >=300KM，则距离分为 0，300KM 以内距离越近，得分越高。另外当搜索品类意图时，加大距离分的权重，比如东莞用户更希望去东莞附近的温泉（东莞本地温泉较少），而不是北京的。
- 综合评价数和评分：早期评价数和评分是线性加权，会出现部分冷门 POI 评价人数较少但评分较高的情况，因此考虑评分的置信度，评价数越多，置信度越高，总体评分越高。
- 新单销量平滑：新单或新 POI 由于上线时间较短销量一般不高，因此对据当前日期一段时间内上线的产品会赋予默认销量，并考虑时间衰减。
- 各因子相乘：文本相关性、距离、评价、销量这些因子维度差异较大，线性加权的权重不好设定，改成相乘，会使各因子的影响更为显著。

文本相关性改进

除了数值类因子优化外，我们对文本相关性也进行了一些改进，早期的文本相关性计算基于 TF-IDF，公式可以简化如下：

$$R_{Q,D} = \sum_{t \in Q} (\sum_{f \in H} \frac{tf_{t,f}}{l_f} * w_f) * idf_t$$

$R_{Q,D}$ 是搜索词和文档的相关性，t 是 Q 分词后的 Term，H 是 t 在文档中命中的文本域集合，$tf_{t,f}$ 是 t 在某个命中文本域 f 中的出现次数，l_f 是文本域 f 的长度，w_f 是 f 的权重，比如 POI 名称域的权重一般会高于 Deal 标题域，idf_t 是 Term t 的倒排文档频率。上述公式主要存在如下问题：

- 文本域长度影响过大，比如搜“庐山”，官方 POI 是“庐山风景名胜区”，分词后包含“庐山”、“风景”、“名胜区”3 个 Term，而“庐山植物园”只包含“庐山”、“植物园”2 个 Term，权重是官方 POI 的 1.5 倍。
- 多个域计算结果求和，对部分文本域缺失的 POI 不公平，比如搜“欢乐谷”，“天津欢乐谷”POI 的品牌名 (Brand Name) 字段是“欢乐谷”，“北京欢乐谷”POI 的品牌名字段为空，导致“北京欢乐谷”的权重不如“天津欢乐谷”。
- 没有考虑字段域的动态权重，比如搜“动物园”，细粒度分词会分成“动物”、“园”，“苏州文化园”POI（包含“动物园、文化园一日游”的 Deal）命中了 Term“园”，“万鸟林”POI 的品类字段是“动物园”，由于 POI 名称域的权重高于品类域，导致“苏州文化园”的权重更高。
- IDF 只体现了 Term 自身的重要程度，不能体现 Term 在 Query 中的重要程度。

基于上述问题对文本相关性计算公式做了如下改进：

$$R_{Q,D} = \sum_{t \in Q} \max_{f \in H} \{ \frac{tf_{t,f} * (k_1 + 1)}{tf_{t,f} + K} * w_f * i_f \} * idf_t'$$

$$K = k_1 * (1 - b + b * \frac{l_f}{avgl_f})$$

其中 k_1 和 b 是调节因子，这部分参考了 BM25 的相关性计算，可以降低文本域长度的影响；另外对多个域的计算结果求 max，减小部分字段缺失的影响；i_f 是命中域的动态权重，可以根据命中 Term 在 Query 中的比例或权重来设置；idf_t' 使用的是 Term 在 Query 中的动态权重。

Term 重要度

如何计算 Term 在 Query 中的动态权重呢？实现时采用模型打分方法，以搜索 Query 为原始语料，人工进行标注，重要度共分 4 级：

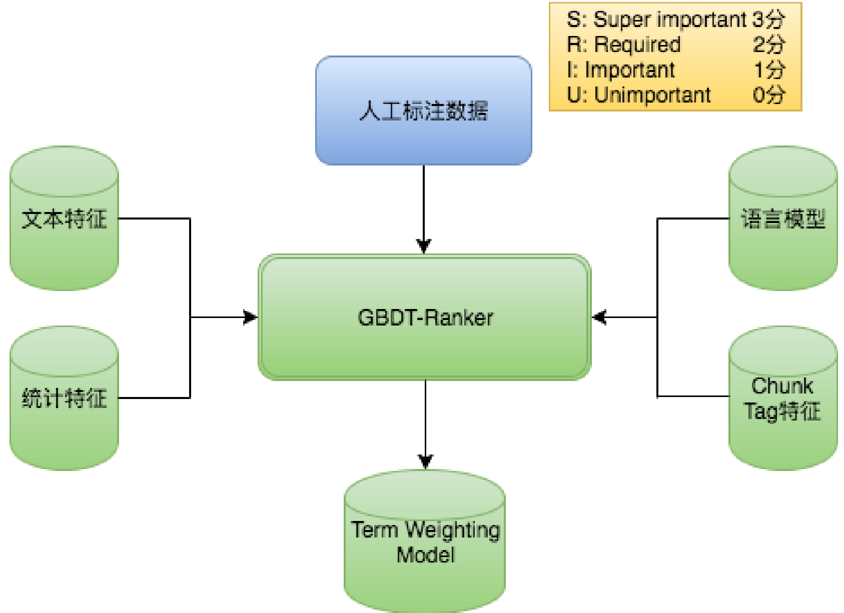
- Super important：主要包括 POI 核心词，比如方特、欢乐谷。

- Required：包括行政区词、品类词等，比如“北京 温泉”中“北京”和“温泉”都很重要。
- Important：包括品类词、门票等，比如“顺景 温泉”中“温泉”相对没有那么重要，用户搜“顺景”大部分都是温泉的需求。
- Unimportant：包括线路游、泛需求词、停用词等。

上例中可见“温泉”在不同的 Query 中重要度是不同的，在特征选取方面有 4 类：

- 文本特征：包括 Query 长度、Term 长度，Term 在 Query 中的偏移量等。
- 统计特征：包括 PMI、IDF 等。
- 语言模型特征：整个 query 的语言模型概率 / 去掉该 Term 后的 Query 的语言模型概率。
- Chunk tag 特征。

模型方面采用 XGBoost 进行训练，离线生成模型后供线上使用。



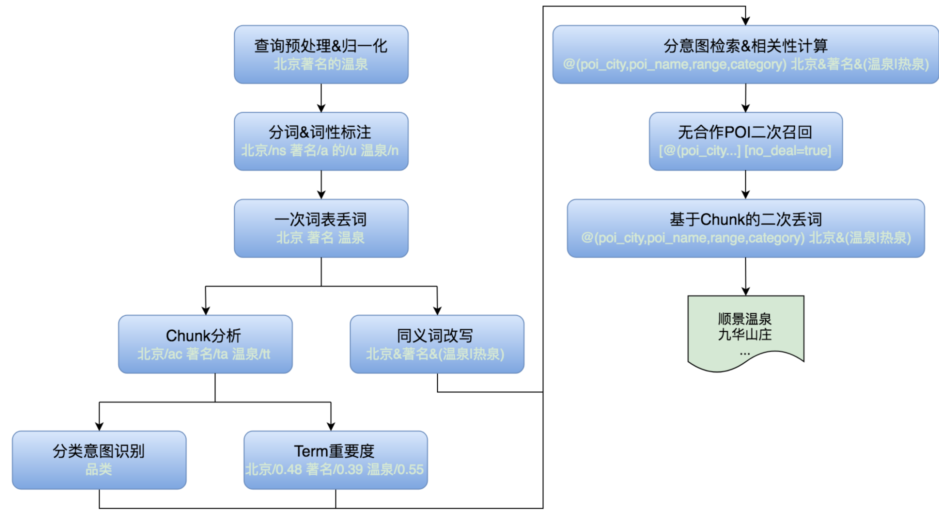
全字段召回

随着粗排序和 Rerank 的改进优化上线，我们放开了 POI 类聚检索字段的限制，改为使用所有字段做文本匹配，包括 POI 城市、名称、品类、商圈，简化了二次召回的逻辑。

召回策略流程示例

经过一年多的迭代，整个搜索召回的流程大致如下，以搜索“北京著名的温泉”为例：

1. 对输入的查询进行预处理，比如特殊字符处理、全半角转换。
2. 查询分词和词性标注，“北京”是地名、“著名”是形容词、“的”是助词、“温泉”是名词。
3. 基于词表的一次丢词，“的”作为停用词被丢弃。
4. 同义词改写，对分词的 Term 匹配同义词，如“温泉”和“热泉”是同义词。
5. 在同义词改写的同时分析 chunk tag，“北京”是城市、“著名”是品类修饰词、“温泉”是品类词。
6. 基于 Chunk 分析的结果识别 Query 整体为品类意图。
7. 同时计算 Term 在 Query 中的重要度，“北京”为 0.48、“著名”为 0.39、“温泉”为 0.55。
8. 基于品类意图确定检索字段和相关性计算的逻辑，比如距离加权。
9. 由于所有 POI 的文本字段中都不包含“著名”，一次召回无结果，因此扩大 POI 范围，在无合作 POI 集合中进行二次检索。
10. 由于无合作 POI 的文本字段也不包含“著名”，二次召回也无结果，因此基于 Chunk 丢弃品类修饰词“著名”，然后进行三次检索。
11. 最终返回搜索结果列表，“顺景温泉”、“九华山庄”等北京著名温泉。



总结

在旅游搜索召回策略的迭代过程中我们并没有采用大开大合的做法，而是参照策略迭代的四步方法论，定期评估搜索质量，对问题分类分析，集中解决主要核心问题，上线实验验证效果，在避免“误召回”和“无召回”之间保持平衡，逐步迭代，为实现更全更准的搜索目标不断改进。

作者简介

郑刚，美团点评高级技术专家。2010 年毕业于中科院计算所，2011 年加入美团，参与美团早期数据平台搭建，先后负责平台、酒旅数据仓库和数据产品建设，目前在酒旅事业群数据研发中心，重点负责酒店旅游场景下的搜索排序推荐、数据挖掘工作，致力于用大数据和机器学习技术解决业务痛点，提升用户体验。

美团点评联盟广告场景化定向排序机制

马莹　一凡

前言

在美团点评的联盟广告投放系统（DSP）中，广告从召回到曝光的过程需要经历粗排、精排和竞价及反作弊等阶段。其中精排是使用 CTR 预估模型进行排序，由于召回的候选集合较多，出于工程性能上的考虑，不能一次性在精排过程中完成候选集的全排序，因此在精排之前，需要对候选广告进行粗排，来过滤、筛选出相关性较高的广告集合，供精排使用。

本文首先会对美团点评的广告粗排机制进行概要介绍，之后会详细阐述基于用户、天气、关键词等场景特征的广告粗排策略。

广告粗排机制简介

广告粗排框架对引擎端召回的若干广告进行排序，并将排序的结果进行截断，截断后的候选集会被传递给广告精排模块处理。

粗排是为了尽可能在候选广告集合里找到与流量相关性较高的广告，一般以有效转化（通常包括点击后发生后续行为、电话、预约、购买等）为目标。流量因素通常包含媒体、用户（用户包含用户画像、历史搜索、历史点击等各类用户行为）因素。此外，还可包含流量外因素，如 LBS、当前实时天气特征等。

考虑到不同流量所覆盖的特征不尽相同：如有的流量包含大量丰富的用户画像，而有的流量无用户画像，但有标识性较为明显的媒体特征，如 P2P、母婴类媒体等，因此对于不同流量，会使用不同的粗排策略，以更好地应用流量特征。

下面将根据不同场景详细介绍各类粗排机制的离线模型，以及线上应用方案。

基于用户画像的广告粗排

用户兴趣对于广告转化有着显著影响。在用户画像基础上，向不同兴趣的用户推

荐不同类型的广告，对广告的点击和转化皆能带来较大提升。下面先对用户画像进行简要介绍，之后阐述我们是如何应用用户画像完成广告定向的。

用户画像

用户画像也称用户标签。美团点评的用户画像与用户的站内行为息息相关。用户的原始行为多为用户对店铺的浏览、点击、购买、点评、收藏等，用户画像主要是基于这些用户行为产出的统计型画像。除了原始行为，我们还整合了其他团队的数据，包含通过频繁集挖掘出的用户预估标签，以及一些产品上自定义标签等。此外，联盟广告独有的 ADX 数据源也被使用进来，ADX 日均约数十亿次请求，涵盖了文学、金融、教育、母婴等各类媒体，我们将媒体带来的部分用户信息进行清洗，并整合到用户画像，从而提升用户画像的覆盖率和丰富性。融合以上所有数据源信息，我们产出了针对联盟广告的用户画像。

在策略应用的同时，考虑到产品投放，用户标签体系的设计采用了树状结构，以便于投放选择。

标签体系分为五大类：

1. 与目前美团点评的商户分类体系强相关（因为广告主都来自于这些产品分类）的兴趣体系，如“美食 / 火锅”兴趣人群，“亲子 / 乐园”兴趣人群等。
2. 自然属性，如用户的年龄、性别、常驻城市等。
3. 社会属性，如职业、婚恋状态、受教育程度等。
4. 心理认知，消费水平、时尚偏好等。
5. 根据某些需求衍生的自定义标签，标签可以根据后续需求不断新增。

在工程方面，用户画像工程每日例行化运行一次，离线处理各数据源，并进行合并，产出设备 ID 粒度（IMEI 或 IDFA）的标准化用户标签。用户标签结果会从 Hive 表导入 Redis 缓存，以供线上加载使用。

离线建模

用户定向包含了两层含义：

1. 策略应用，针对用户的不同兴趣，对召回的广告进行权重调整，以筛选出最适合用户的广告。
2. 产品投放，广告精准定向，即某些广告只投放具有某些兴趣的人群，如幼儿教育的广告，只投放给家里有小孩的人群，以获取更好的投放效果。

本文所述的用户定向，仅指策略应用。

在用户定向上，我们使用了频繁集挖掘方案，因为用户标签与商户分类较为相似，直观上讲，规则的收益可能好于模型。使用频繁集方案，一方面可以挖掘出规则考虑不到的关联关系；另一方面，它的可解释性较强，且后期可以方便地进行人工干预。

我们抽取一段时间的用户点击历史数据，来挖掘用户兴趣与广告商户分类的关联关系。同时，考虑到转化（包含电话、预约、购买等种种行为）是一种强行为，我们将其等同于多次点击行为，进行升采样（如一次电话等于两次点击，等等）。之后使用 Spark 的 ML 库来进行频繁集挖掘。数据处理上，每条点击纪录会将广告商户的一级、二级、三级分类分别与用户标签关联，即一条记录会拆分成多份，分别使用不同级别的分类与用户标签关联，这样保证在计算频繁集的时候，各个层级分类都不会被遗漏。通过 Spark 的 ML 库找出大量频繁集后，剔除掉仅包含广告分类或仅包含用户标签的，仅保留两者共存集合。同时我们限制了用户标签在频繁项中的数量，使其不超过两个，以保证规则可以覆盖较多线上用户。接着，我们对标签与广告分类的关系进行打分，广告分类 A 与兴趣标签 B 的打分为：$\frac{\sum(A \cap B)}{A}$ 。

举例见下表：

广告二级分类	用户兴趣tag	共生次数	该二级分类下广告展现次数	用户tag与广告关联的打分
亲子/儿童乐园	亲子/幼儿教育	100	1000	0.1
亲子/少儿才艺	亲子/幼儿教育	150	1000	0.15
美食/火锅	亲子/幼儿教育	150	2000	0.075

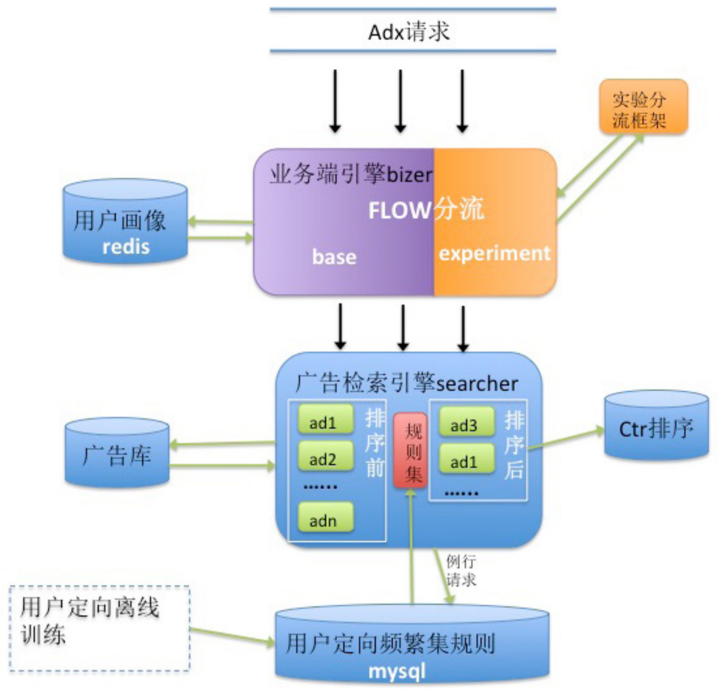
“美食 / 火锅”更为大众化，因此访问量远大于“亲子 / 儿童乐园”，上列访问量虽为虚构，但与实际认知相符。上列表格想要说明的是，像“美食 / 火锅”这样的广告分类，可能与很多用户标签的共生次数都很高，但由于它本身是大众化的分类，因

此分母取值也很大，则实际关联打分就会很小（除非是强相关兴趣）。用这种方式打分，可以筛选出指定标签下关联度较高的广告二级分类。

得到全部的频繁集及相应打分后，可线下进行人工筛选，剔除掉明显不符合认知的频繁项集。最终结果作为离线模型产出，写入 MySQL。

检索端加载

检索端每天定时加载一次离线结果到内存中。对于实时广告请求，先从 Redis 读取当前用户设备的用户画像，然后根据用户实时位置等条件召回几百条广告后，对于当前流量中有用户画像的，根据离线训练结果，对广告进行打分及排序，排序后会根据线上的设置进行截断。用户定向的整体应用框架如下：



在业务引擎端，我们会进行 A/B 分流实验，随机划分一定百分比的流量作为实验流量，用于进行用户定向实验。之后，分析一段时间的累积实验结果，与 base 分流作对比，以检测用户定向策略带来的转化率提升，同时反馈给上游频繁集模型，用于干预调整离线产出模型。

基于天气场景特征的广告粗排

实时天气情况对用户有一定影响。直观上来说，炎热的天气下，用户会更倾向于点击游泳馆的广告；而寒冷的天气下，飘着热气的星巴克广告会更受欢迎。有些广告行业则跟天气的关系不大，如非实时消费的结婚、摄影、亲子类广告。下面将介绍我们基于天气场景的离线模型及在线打分方案。

数据准备

天气基础数据包括温度、雨量、雪量、天气现象（大雨、雾霾等）、风力等级等。我们需要的天气数据，并不需要实时更新，原因如下：

① 从应用角度讲，由于天气情况在短时间内比较稳定，并不需要每时每刻都抓取天气数据；

② 第三方媒体对 DSP（Demand-Side Platform）的响应时间有严格限制，如果每次广告请求都去请求天气数据，对广告引擎的性能将造成较大影响。因此我们以小时粒度来保存天气数据，即在确定的某个城市、某个小时内，天气情况是固定的。

天气数据包含两种：

① 线下模型使用的历史天气数据；

② 线上检索使用的当前天气数据。

两者在相同特征上的数据取值涵义要保证一致，即特征的一致性。美团配送团队对基础天气数据进行清洗和加工，每日提供未来 72 小时内的天气预报数据，同时保存了稳定的历史数据，与我们的需求完全吻合，因为我们使用了该数据。

离线建模

天气特征的离线模型选用了 AdaBoost 模型，该模型可使用若干简单的弱分类器训练出一个强大的分类器，且较少出现过拟合现象。考虑到要在线上检索端加载弱分类器进行计算，基础的弱分类器不能太过复杂（以免影响线上性能），基于以上考虑

我们选用了 AdaBoost 常用的树桩模型（即深度为 1 的决策树）。

选定模型后，首先需要对原始数据进行处理，将其处理成适合决策树分类的特征。我们选定温度、湿度、降水量、降雪量、天气情况等特征。以温度举例，将其作为连续变量处理，对于特征为温度的决策树，训练合适的分割点，将温度归类到合适的叶子节点上；而对于天气情况 icon-code，由于其仅有几个取值（正常、一般恶劣、非常恶劣等），因此当做离散值对待，进行 one-hot 编码，散列到有限的几个数值上（如 1、2、3）。在树桩模型中，左右叶子节点分别对离散值进行排列组合（如左子树取 1、3，右子树取 2 等），直到左右子树的均方误差值之和为最小。当然，对于离散值较多的情况，出于性能上考虑，多以连续值对待，并训练合适的分割点分离左右子树。

对特征进行处理后，可以应用模型对特征进行迭代处理。我们以转化为目标，搜集一段时间的历史点击数据，对数据进行特征化处理，最终训练出合适的离线模型。直观上来讲，天气对不同行业的转化影响有显著差异。某些行业对天气更为敏感，如餐饮、运动等，而有些行业则对天气不敏感，如亲子、结婚（因为转化一般不发生在当下）等，因此我们对不同的广告分类分别做训练，每个行业训练一个模型。考虑到在检索端需要对广告和天气的特征关系进行打分，因此分类模型不能完全满足我们的需求。最终我们选取了 AdaBoost 的改进版 Gentle AdaBoost，有关该模型的论述网上有很多（根据文献 1，在采用树桩模型时，该模型效果好于传统的离散 AdaBoost），本文不再对其进行数学说明。算法大致流程如下图所示：

Gentle AdaBoost

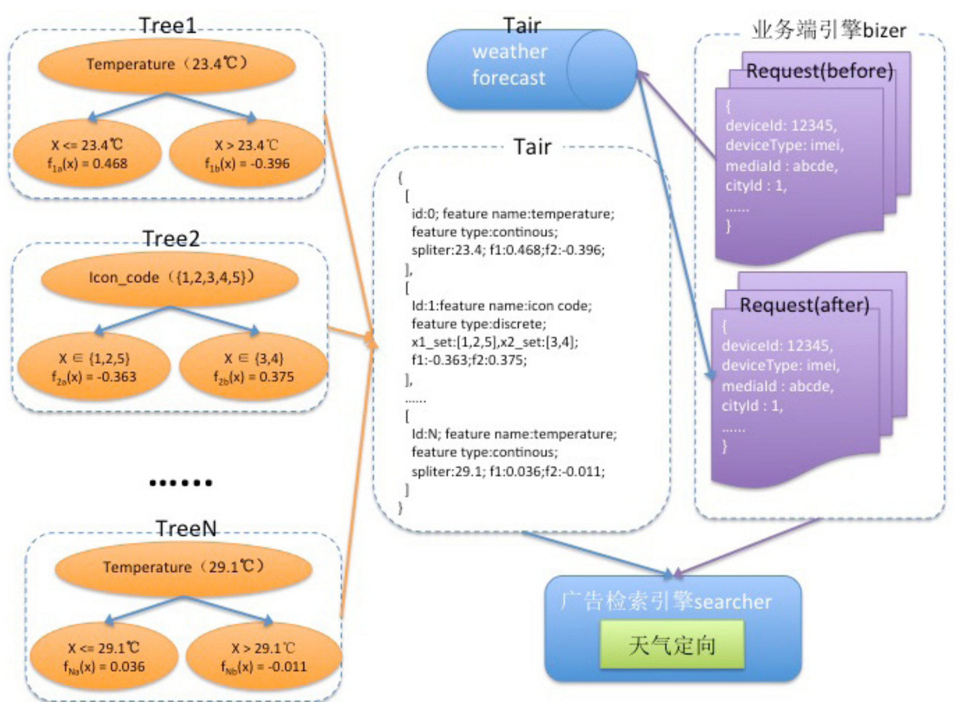
1. Start with weights $w_i = 1/N, i = 1, 2, \ldots, N, F(x) = 0$.
2. Repeat for $m = 1, 2, \ldots, M$:
   (a) Fit the regression function $f_m(x)$ by weighted least-squares of y_i to x_i with weights w_i.
   (b) Update $F(x) \leftarrow F(x) + f_m(x)$.
   (c) Update $w_i \leftarrow w_i \exp(-y_i f_m(x_i))$ and renormalize.
3. Output the classifier $\text{sign}[F(x)] = \text{sign}[\sum_{m=1}^{M} f_m(x)]$.

另外说明一下我们对该模型的一些工程处理，我们去掉了最后的感知器模型，直

接使用回归函数的和作为打分。在每次迭代过程中，我们会保留当前错误率，当迭代达到一定次数，而错误率仍大于给定阈值时，则直接舍弃对该行业的训练，即在天气场景定向中，不对该行业的广告打分。

工程上，我们使用 Spark 进行特征处理和模型训练，并将最终结果写入在线缓存 Tair 中。其中 key 为一级及二级行业，value 即为 AdaBoost 模型的多轮迭代结果，同时保留了最后一轮迭代的错误率。保留错误率的原因是在线上检索端加载模型时，可以动态配置错误率阈值，根据模型的错误率超过阈值与否来决定是否对广告打分。另外，考虑到线上加载迭代模型会牺牲性能，我们将迭代轮次控制在 100 次以内。

天气场景的离线数据和线上数据模型如下图所示：



经过 Gentle AdaBoost 训练出的多棵树模型，以 JSON 格式写入 Tair 中；线上在获取 ADX 请求后，根据 Tair 中已经写好的天气预报信息，加载当前小时当前城市的天气情况，检索端根据当前天气和模型，对广告进行打分和排序。

线上优化

广告检索端需要从 Tair 读取离线模型，来完成广告打分。由于第三方媒体对 DSP 响应时间要求较高，在线上做迭代模型的加载，是一个较为耗时的操作，因此需要做一些优化处理。我们一共做了三层优化处理：

1. 模型缓存：对于检索端召回的几百条广告，对广告一级 / 二级分类进行缓存处理，对某条广告打分后，其对应的二级分类及相应模型加入缓存，而后续遇到来自同样二级行业的广告，将直接使用缓存模型。
2. 打分缓存：由于我们使用了小时级的天气数据，因此对于指定的城市，在指定的小时内，天气状况完全一致，当广告一级 / 二级行业确定后，模型对于该广告的打分是确定的。因此我们对广告行业 + 城市的打分进行缓存处理，且每个整点清空一次缓存。对于已经打分过的城市 + 广告行业，可以直接从缓存中读取并使用打分结果。
3. 性能与打分的折中：使用了前面两种缓存方案后，性能仍无法得到足够保证，此时我们需要考虑一个折中方案，牺牲一部分广告打分，以换取性能的提升。即我们使用动态配置的阈值来控制每次检索请求中模型迭代的轮次。举个例子，阈值设为 200 次，在整点时刻，前五个召回的广告的行业各不相同，且使用的模型分别迭代 80、90、60、60、30 次结束，则本次请求中我们只对前三个广告打分（80+90+60<200），并将广告打分进行缓存。后续召回广告，其二级分类若能命中缓存，则打分，否则不打分。在第二次广告请求过来时，同样沿用这个策略，对已经缓存的广告打分直接加载，否则迭代模型进行打分，直到达到迭代阈值 200 为止。通过打分缓存机制，可以保证前面牺牲掉的广告行业被逐步打分。使用该优化策略，可以完全确保上线后的性能，通过调整迭代轮次的阈值，控制打分与性能的折中关系。

通过以上三层优化处理机制，保证了 AdaBoost 这样的迭代模型可以在线上被加载使用。另外，我们还会考察离线模型中的错误率，通过线上动态调整阈值，舍弃一些错误率较高的模型，以达到效果的最优化。模型上线后，我们进行 A/B testing，

以检测使用天气场景模型带来的转化率提升。

基于关键词特征的广告粗排

用户在美团点评站内搜索的关键词，强烈地表达了用户的短期偏好。基于关键词定向（Query Targeting）是许多广告精准定向的利器。尤其对于闭环条件下的应用，如百度凤巢，淘宝的直通车，当用户在站内进行搜索，可以直接根据搜索词展示相关广告，引导用户在站内转化。一般来说，关键词定向的效果都非常出色。联盟的关键词定向，是通过对用户近期搜索词的分析，识别出用户感兴趣的店铺及店铺分类，进而在站外 App 为用户投放相关广告。下面将介绍联盟广告基于关键词特征的广告排序机制。

离线模型

离线模型需要根据用户搜索词分析出用户的偏好，对于大多数搜索引擎来说，需要使用 NLP 的相关技术和复杂的基础特征建设工作。对于美团点评的关键词搜索场景，由于大部分搜索词与美团点评店铺及店铺分类强相关（大部分搜索词甚至直接是店铺名称），且新词搜索量增长幅度不大，同时考虑到开发成本，我们的关键词定向舍弃了基础特征构建的方案，而是直接采用一套合理的离线分析模型，来构建搜索词和店铺分类的关系。

我们选用了 TF/IDF 模型，来构建关键词和用户偏好的关系。用这个方案原因是文章 – 词模型与词 – 店铺模型非常相似。该方案主要用于计算店铺分类（或商圈）与关键词的相关程度，也是对其打分的依据。该模型相关资料很多，不再做原理性阐述，此处仅举一例如下：

用户搜索词为“潮汕火锅”，计算“美食 / 火锅”的商家分类与该关键词的相似程度。

假设：

c1 = 搜索“潮汕火锅”后的全部点击数，

c2 = 搜索“潮汕火锅”后点击“美食 / 火锅”类目店铺的全部点击数，

c3 = 搜索词总数，

c4 = 搜索点击“美食 / 火锅”类目的词总数。则

$$tfidf=(c2/c1)\times log(c3/(c4+1))$$

由此计算出店铺分类与关键词的关系，取 topN（根据存储大小及不同店铺对同一词的 TF-IDF 差距拟定）个店铺分类。

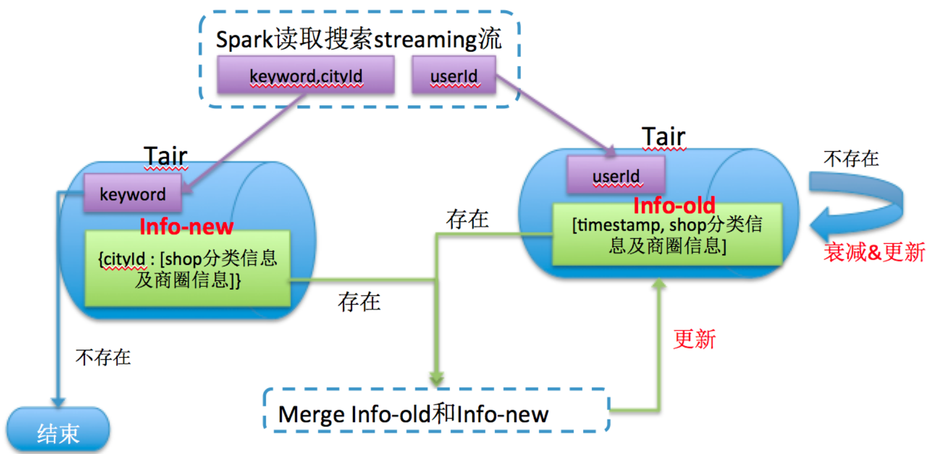
单店及商圈计算方法与此类似，它们的计算值会同时与店铺分类的 TF-IDF 进行比较，不作区分。（此处有一点需注意：如果用户搜索“中山公园 火锅”，可以预见店铺分类与商圈会同等重要，则最终产出两条独立打分规则，分别挂在店铺分类和商圈下面）。

我们使用 Spark 来构建离线模型，提取用户的搜索词和搜索后点击的店铺及店铺分类，运用上述方案来计算每个搜索词的关联店铺及店铺分类，设置阈值，保留分数较大的分类结果。

为了提升线上命中率，我们使用了点评分词系统，对长度较长的搜索词进行分词，同时保存原始词和切分后基础词的 TF-IDF 结果。为了方便线上快速检索，结果同样保存在 Tair 中。以检索词为 key，关联店铺分类和店铺的 TF-IDF 打分作为 value 进行保存。

实时流计算

对于关键词定向，与用户定向的一个区别在于前者时效性要求很高，因此需要使用实时计算系统来处理用户行为，并将最后的结果保存在 Tair 集群。首先通过 Kafka 订阅用户行为实时流，以五分钟为时间片处理用户行为，查找用户 ID 和搜索词，如果搜索词过长，则进行分词，接着从 Tair 中查找出与该搜索词相关的店铺及店铺分类和打分（离线模型给出）。接下来会 Tair 中查找该用户 ID 是否有历史结果，若有，则读出，对之前的打分进行衰减（衰减方案见下文），并与当前新的打分进行合并；否则，将新的数据及时间戳写入 Tair。该方案的流程图如下：



比较重要的部分是合并新来的数据与 Tair 里的老数据，合并时，如果新数据包含老数据中某些店铺（店铺分类），就直接使用新数据中的店铺（店铺分类）权重；否则，对老数据中的店铺（店铺分类）权重进行衰减，若衰减后权重小于给定的阈值，则直接将这个店铺（店铺分类）从合并数据中剔除。

衰减方案根据时间进行衰减。默认半衰期（即衰减权重从 1 衰减到 0.5 的时间长度）为 72 小时（不同的店铺分类给予不同的半衰期），使用牛顿冷却定律，参数计算公式为：0.5 = 1 × e− α * 时间间隔 ，解出 α，并带入下面公式得到实际权重为：

$$w' = w \times e^{-\alpha \times 时间间隔}$$

其中，w 为老权重，w' 为新权重。

检索端排序

检索端接收到广告请求，根据当前获取的用户 ID，从 Tair 中读取用户偏好的店铺分类，与召回的广告进行匹配，当广告分类与召回广告匹配成功，则可将 Tair 读出的分数进行时间衰减后，作为该广告的最终打分。检索端采用与实时流同样的时间衰减方案，以保证一致性。举例如下：

用户 A 在早上 8:30 有火锅类搜索行为，Spark Streaming 处理后进入 Redis，假设此时最新时间戳为 8:30，而该用户在 11：00 搜索亲子类商铺，Spark

Streaming 处理该条记录后，之前的火锅权重需要衰减，同时时间戳更新为 11:00，假设此时立即有广告检索请求命中该用户，则此时用户火锅类偏好权重为 11:00 时权重；假设下午 16:00 有 ADX 请求命中该用户，则用户火锅类权重需要根据 16:00 到 11:00 的时间间隔继续衰减。

除了上述三种定向策略，还有其他基于上下文的定向，重定向等，它们方案各异，但大致思路与前述方案类似，本文不再详述。

定向汇总

在经过上述各类定向场景分别打分之后，需要对每个场景打分进行综合，因为不同的广告行业在不同场景下的重要性是不同的。如餐饮行业更注重距离和天气，而丽人、亲子等行业较注重媒体和用户画像。因此，不同行业下，各个定向打分的权重是不同的。我们使用模型的方式对各个场景打分进行权重的训练和预测。

综合打分我们采用了 LR 模型，分不同广告行业，以点击为样本，转化为模型，以各个场景下的前期打分为特征，进行混合打分权重的离线建模。

$$h_\theta = g(\theta^T) = \frac{1}{1 + e^{-\theta^T x}}$$

其中 θ 是向量， $\theta_0 x_0 + \theta_1 x_1 + ..., + \theta_n x_n = \sum_{i=0}^{n} \theta_i x_i = \theta^T x$, 其中， $x_1,...,x_n$ 是各个场景定向下的具体打分，打分分布在 [0,1] 之间。

冷启动时，对每个场景打分给予一个默认权重，积累一定量数据后，使用离线模型训练出各个广告行业下的 θ 向量，并在引擎端加载使用。引擎端加载各个场景的广告打分，并根据广告行业加载打分权重，最终完成每个广告与当前的流量综合打分。

总结

场景化定向综合考虑了当前流量的种种场景因素（用户、天气、媒体等），分别根据业务需求设计了不同的模型来构建广告打分机制，并对单个场景的广告打分进行综合。通过这种场景化的广告粗排机制，对召回的广告进行排序和筛选，可以保留相关

性较强的广告，以用于后续的 CTR 排序和处理。从实际的 A/B testing 来看，使用了场景化排序机制的流量，其点击率和转化率的提升效果都较为显著。

展望未来，如何丰富各类场景特征（如天气、媒体的更多特征），引入更多的场景因素（如所处环境周边店铺、当前时间用户行为等），尝试不同的模型方案，都是下一步的可探索方向。

参考文献

Friedman J, Hastie T, Tibshirani R.(2000), Additive Logistic Regression: A Statistical View of Boosting, Annals of Statistics, 28, 337−307.

作者简介

马莹，美团点评高级算法工程师，2012 年毕业于浙江大学，同年加入百度联盟研发部。2016 年加入美团点评联盟广告部门，长期从事广告行业策略算法研究开发工作。专注于计算广告、用户画像、数据挖掘等方向。

一凡，美团点评高级算法工程师，现负责美团点评广告平台联盟广告网盟方向。2011 年毕业于华中科技大学，毕业后先后就职于百度、腾讯，2014 年加入点评平台，2016 年加入美团点评联盟广告部，长期致力于计算广告算法优化、推荐算法、大数据挖掘等方向。

美团 DSP 广告策略实践

鸿杰　大龙　李乐

前言

近年来，在线广告在整个广告行业的比重越来越高。在线广告中实时竞价的广告由于其良好的转化效果，占有的比重逐年升高。DSP（Demand-Side Platform）[1] 作为需求方平台，通过广告交易平台（AdExchange）[2] 对每次曝光进行竞价尝试。对于 AdExchange 的每次竞价请求，DSP 根据 Cookie Mapping [3] 或者设备信息，尝试把正在浏览媒体网站、App 的用户映射到 DSP 能够识别的用户，然后根据 DSP 从用户历史行为中挖掘的用户画像，进行流量筛选、点击率 / 转化率预估等，致力于 ROI [4] 的最大化。

美团点评的用户量越来越大，积累了大量的用户在站内的行为信息，我们基于这些行为构造了精准的用户画像，并在此基础上针对美团 App 和网站的用户搭建了美团 DSP 平台，致力于获取站外优质的流量，为公司带来效益。本文从策略角度描述一下在搭建 DSP 过程中的考虑、权衡及对未来的思考。

- 在 DSP 实时竞价过程中，策略端都在哪些步骤起作用；
- 对每一个步骤的尝试和优化方向做出详细介绍；
- 总结 DSP 如何通过 AB 测试、用户行为反馈收集、模型迭代、指导出价 / 排序等步骤来打通整个 DSP 实时竞价广告闭环。

竞价展示流程

美团 DSP 在一次完整的竞价展示过程中可能涉及到两个大的步骤：

1. 对 AdExchange 的竞价请求实时竞价；
2. 竞价成功之后用户点击进入二跳页、浏览、点击、最后转化。

我们分别看一下这两个步骤中策略的支持。

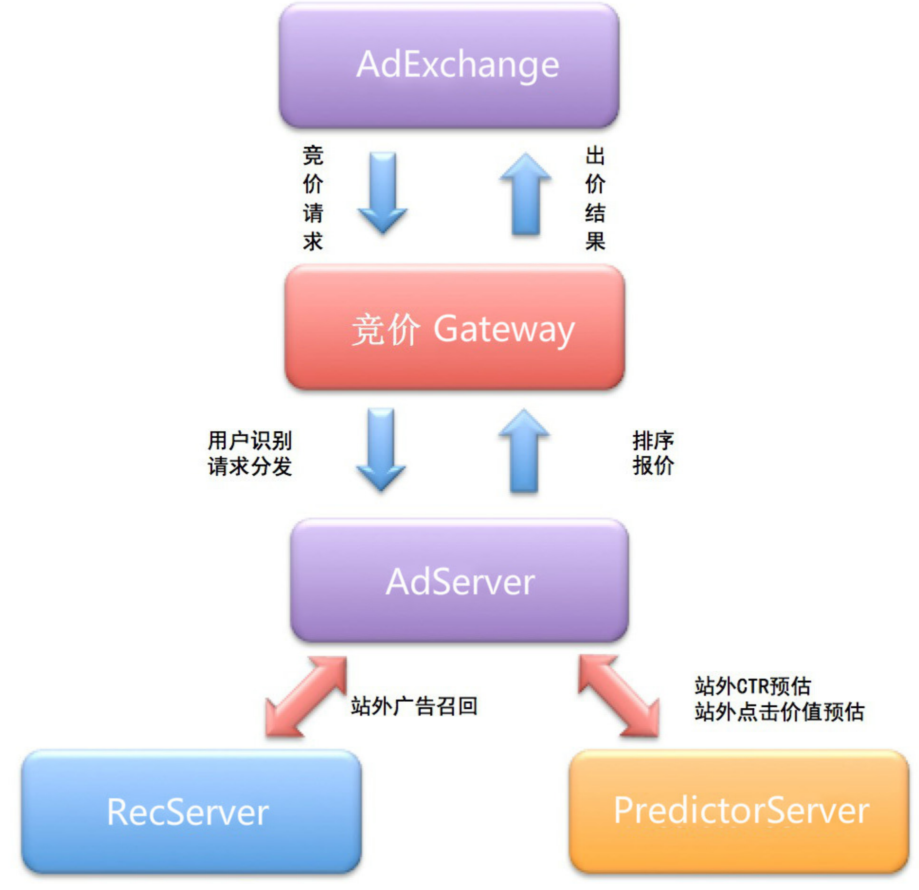


图 1　竞价广告示意图

图 1 给出了每一次竞价广告的粗略示意图，竞价 Gateway 在收到竞价请求之后，会识别出美团点评用户的流量，根据网站历史 CTR、网站品类属性等因素进行简单的流量过滤，把流量分发到后端的 AdServer。AdServer 作为后端广告的总控模块，首先向 RecServer（定向召回服务）获取站外展示广告召回结果，然后根据获取的广告结果向 PredictorServer（CTR/ 点击价值预测服务）请求每个广告的站外点击率和点击价值。最后 AdServer 根据获取的点击价值 v 和 ctr，根据 $v^{*}ctr^{t}$ 进行排序，从而挑选出 top 的广告进行展示。

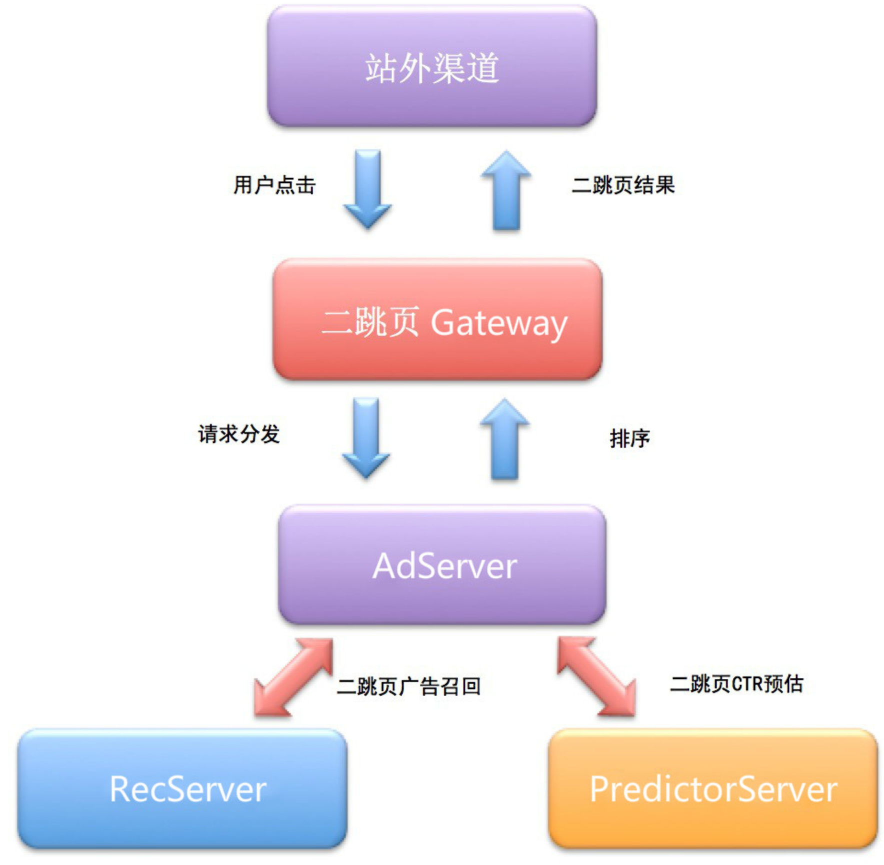


图 2 二跳页广告流程图

图 2 给出了竞价成功后，用户从站外展示的广告点击后，所经历的流程示意图。用户点击站外广告后，到达二跳页 Gateway，二跳页 Gateway 向 AdServer 请求广告列表。AdServer 从 RecServer 获取站内二跳页广告召回结果，然后根据获取的广告结果向 PredictorServer 请求每个广告的二跳页点击率并进行排序。排序后的结果返回给二跳页 Gateway 进行广告填充。

在上述两个步骤中，美团 DSP 策略端的支持由 RecServer 和 PredictorServer 提供，在图 1 和图 2 分别用红色的箭头和 AdServer 交互。其中 RecServer 主要负责站外广告和二跳页的广告召回策略，而 PredictorServer 主要负责站外流量的 CTR 预估，点击价值预估和二跳页内的 CTR 预估。整个策略的闭环如下图：

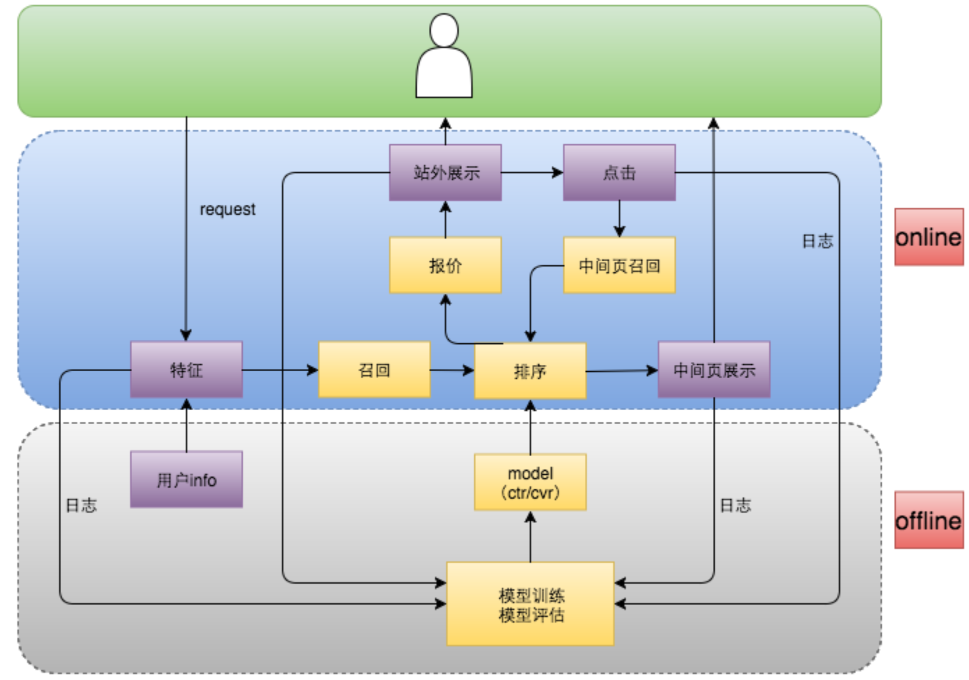


图 3　策略闭环图

接下来详细介绍下美团 DSP 的召回、CTR 预估、点击价值预估相关的策略。

定向召回

从上一小节的介绍可以看到，定向召回服务分别在实时竞价过程中提供了站外广告的召回服务，在竞价完成之后提供了二跳页的广告召回服务。站外召回和站内二跳页召回没有本质的区别，比较常见的做法是二跳页会根据用户点击商品的品类进行品类过滤。下面我们具体看一下目前定向召回相关的具体策略。

基于实时行为召回

通过实时日志流平台准确的跟踪用户的实时点击浏览 / 收藏 / 购买行为，对于相应的用户重新投放用户近一段时间内发生过浏览 / 收藏 / 购买行为的商品。需要注意的是这个策略需要考虑召回概率按时间进行衰减，用户的实时行为能够比较强反映用户的近期兴趣，距离当前时间比较长的用户行为对于用户近期兴趣的定向偏弱。

基于位置召回

O2O 的业务特点与传统的电商有明显的区别，传统电商是在线上达成交易意向，然后通过快递送货的方式完成交易。O2O 业务绝大部分消费者是在线上买入电子券，然后要到店进行消费，所以用户的位置信息在广告召回中起着举足轻重的作用。我们在基于位置的广告召回中尝试了以下三种策略：

1. 实时地理位置召回

根据用户所在的实时地理位置召回距离比较近的广告。

- 对于移动端的广告流量，可以比较准确的获得用户的实时地理位置，从而进行比较精准的投放；
- 对于 PC 端的流量，地理位置是通过用户访问的 IP 地址进行推算的，所以地理位置是有偏移的，但是考虑到 PC 端浏览广告的流量用户位置一般都比较固定，比如用户一般是在上班或者在家休息，我们仍然使用了这个策略。

2. 实时商圈热单召回

根据用户所在的实时地理位置推断出用户目前所在商圈，给用户投放当前商圈的热门消费单。商圈的范围一般在几公里范围之内，对于用户到店消费是一个合理的距离范围，所以我们离线挖掘出每一个商圈的热门消费单，作为用户召回的候选。

> 可以看到策略 1 和策略 2 是不需要 userid 的，所以这两个策略也是我们在识别不到 userid 的时候一个比较好的冷启动召回策略。

3. 偏好商圈热单召回

通过离线分析用户历史的浏览 / 点击 / 购买行为，分析出用户的历史商圈偏好，召回用户偏好的商圈消费热单作为广告候选集。这个策略需要用户的 userid，仅对于能够识别并能映射到 userid 的用户适用。

基于协同过滤召回

基于协同过滤的召回策略我们融合了 user-based 和 item-based 两种。

基于 item-based 的协同过滤，我们首先通过用户的购买行为计算 item 之间的相似度，比如通过计算发现 item A 和 item B 之间的相似度比较高，我们把 item A 作为候选推荐给购买 item B 的用户，作为 item B 的用户的召回候选集之一；同样也把 item B 作为候选推荐给购买 item A 的用户，作为购买 item A 的用户的召回候选集之一。因为 item-based 协同过滤的特征，这一部分召回基本能够把热门爆款单都拉到候选集中。

基于 user-based 的协同过滤，我们同样需要先计算用户之间的相似度。计算用户相似度时，除了考虑用户购买的商品，还可以把用户所消费过的商家及商家所在的商圈作为相似度权重考虑进来。这么做是因为，很多商品是在全国多个城市都可以购买的，如果只采用用户购买的商品来计算相似度，可能把两个不同城市用户的相似度计算的比较高，加入商家和商圈的权重，可以大大降低这种情况的可能性。

基于矩阵分解的场景化召回

对于 O2O 消费的某些场景，比如美食和外卖，用户是否发生购买与用户目前所处的场景有很大关系，这里的场景包含时间、地点、季节、天气等。举个例子来说，工作日的中午，如果还在下雨，这个时候外卖的购买概率一般是比其他商品高的。

基于此，我们开发了基于矩阵分解的场景化召回策略。我们采用了 FM 模型来进行建模，建模的特征包括季节、时间（工作日 / 周末，一天之内的时段）、地点、天气等。这个策略的目的是希望召回用户实时的基于场景化的需求。

CTR 预估

上文提到在实时竞价阶段，AdServer 会跟 PredictorServer 请求每个广告的站外点击率和点击价值，最后 AdServer 根据获取的点击价值 v 和 ctr，根据 $v^{*}ctr^{t}$ 进行站外广告排序，挑选 top 的广告。最终的报价公式如下：

$$a*\sum_{i=1}^{k}v_i*ctr_i^t+b \quad (1)$$

k 是本次竞价要展示的广告数，t,a,b 都是根据实际流量情况进行调整。其中 t 为

挤压因子，为了控制 ctr 在排序和报价中起作用的比重，t 越大，ctr 在排序和报价中的比重越高；a,b 需要根据 DSP 需要获取的流量和需要达到的 ROI 之间的权衡进行调整，a,b 越大，出价越高，获取的流量越多，成本越高，ROI 就减少。

公式 1 中 CTR 直接作为一个引子进行出价计算，所以这里的 CTR 必须是一个真实的点击率。因为在站外广告点击日志中，正样本是非常稀疏的，为了保证模型的准确度，我们一般都会采用负样本抽样。这样模型估计出来的 CTR 相对大小是没有问题的，可以作为排序依据，但是用来计算出价的时候，必须把负样本采样过程还原回去，我们在下面的小节中详细解释。

站外 CTR 预估

该模型目标是，对于 RecServer 召回的广告，预测出广告的相对点击率和真实点击率，相对点击率用于排序，真实点击率用于流量报价。对于每个流量，AdExchange 会下发给多个 DSP，报价最高的 DSP 会胜出，获取在这个流量上展示广告的机会。为了能够引入更多的优质流量，减少流量成本，提高 ROI、CTR 预估模型需要充分考虑站点、广告、用户等维度的信息。

广告的点击与转化主要与用户、广告、媒体（user，ad，publisher）这三个因素相关。我们的特征也主要从这三个方向去构建，并衍生出一些特征 [5]。

特征选择

1. 用户特征

用户浏览，购买的品类，用户画像，浏览器，操作系统等特征。

2. 广告特征

- 广告 deal 的属性特征，如商家、品类、价格、创意类型等特征。
- 广告 deal 的统计特征，如历史 CTR、CVR、PV、UV、订单量、评分等。

3. 媒体特征

网站类别，网站域名，广告位，尺寸等特征。

4. 匹配特征（主要是用户与广告维度的匹配）

- 用户浏览、购买的品类与广告品类的 match，商家的 match。
- 用户浏览广告的不同时间粒度的频次特征，比如用户浏览当前广告的次数、用户上次点击广告距离当前的时间差。

5. 组合特征

在 LR+ 人工特征的实现过程中，需要人工构造一些组合特征，比如，网站 + 广告、用户消费水平 + 价格、广告主 + 广告品类等，对于 FM 和 FFM 能都自动进行特征的组合。

6. 环境特征

广告的效果往往与用户所处的外部环境相关。比如 时段、工作日 / 节假日、移动端的经纬度等。

特征处理

最后再看我们具体如何构建模型。

1. 模型选择

由于站外的站点数量巨大、广告位较多、广告的品类较多，造成训练样本的特征数较大，需要选择合适的模型来处理，这里我们选用了 LR+ 人工特征的方式，确保训练的性能。

2. 特征降维

点击率模型需要考虑用户维度的数据，由于美团的用户量巨大，如果直接用用户 id 作为特征会造成特征数急剧增大，而且 one-hot encoding 后的样本会非常稀疏，从而影响模型的性能和效果。所以我们这里采用了用户的行为和画像数据来表征一个用户，从而降低用户维度的大小。

3. 负样本选择

- 对于站外广告，有很多广告位比较靠近页面的下方，没有被用户看到，这样的广告作为负样本是不合理的。我们在负样本选择的时候需要考虑广告的位置信

息，由于我们作为 DSP 无法获取广告是否真实被用户看到的信息。这里通过适当减少点击率较低的展位负样本数量，来减轻不合理的负样本的情况。

- 对于二跳页广告，只取点击的位置之前的负样本，而未点击的则只取 top20 的广告作为负样本。

4. 负样本采样

由于广告点击的正负样本分布极其不均，站外广告的点击率普遍较低，绝大多数样本是负样本，为了保证模型对正样本的召回，需要对负样本按照一定比例抽样。

5. 真实 CTR 校准

由于负样本抽样后，会造成点击率偏高的假象，需要将预测值还原成真实的值。调整的公式如下：

$$q = \frac{p}{\left(p + \frac{1-p}{w}\right)} \tag{2}$$

> q: 调整后的实际点击率。
>
> p: 负样本抽样下预估的点击率。
>
> w: 负样本抽样的比例。

二跳页 CTR 预估

当用户点击了广告后，会跳转到广告中间页，因为站外流量转化非常不容易，所以对于吸引进来的流量，我们希望通过比较精细化的排序给用户投放尽可能感兴趣的广告。

由于进入二跳页的流量大概比站外流量少两个数量级，我们可以使用比较复杂的模型，同时因为使用比较多的用户 / 广告特征，所以这里我们选择了效果比较好的 FFM[6] 模型（详情可以参考之前的博客文章《深入 FFM 原理与实践》）。

特征和样本处理方面的流程基本类似 CTR 预估模块中的样本处理流程。差别在

于广告在展示列表中的位置，对广告的点击概率和下单概率是有非常大影响的，排名越靠前的广告，越容易被点击和下单，这就是 position bias 的含义。在抽取特征和训练模型的时候，就需要很好去除这种 position bias。

我们在两个地方做这种处理：

- 在计算广告的历史 CTR 和历史 CVR 的时候，首先要计算出每个位置的历史平均点击率 ctr_p，和历史平均下单率 cvr_p，然后再计算 i 广告的每次点击和下单的时候，都根据这个 item 被展示的位置，计算为 ctr_0/ctr_p 及 cvr_0/ctr_p。
- 在产生训练样本的时候，把展示位置作为特征放在样本里面，并且在使用模型的时候，把展示位置特征统一置为 0。

点击价值预估

上文提到广告是根据 v^*ctr^t 进行排序，并通过公式 1 进行报价。这里面的 v 就是点击价值（点击价值是指用户发生一次点击之后会带来的转化价值）。

广告业务的根本在于提高展示广告的 eCPM[7]，eCPM 的公式可以写为 v^*ctr^*1000，准确的预估点击价值是为了准确预估当前流量对于每一个广告的 eCPM。刘鹏在《计算广告》[8] 中提到，只要准确的估计出点击价值，通过点击价值计算和 CTR 计算得到的 eCPM 进行报价，就始终会有利润，这是因为 AdExchange 是按照广义第二出价进行收费的。

在实际投放过程中，出价公式可以随着业务目标的不同进行适当的调整，比如我们的出价公式中包含了挤压因子 t，和 a ，b 两个参数。出价越高带回来的流量越大，可能带来质量参差不齐的流量，一般在一段时间之内会引起 CTR 的降低，这样会带来 CPC 点击成本的提高，所以 ROI 会降低。反之出价比较低的情况下，带来的流量越少，经过比较细致的流量过滤，CTR 能长期保持在一个较高的水平，点击成本 CPC 比较低，ROI 就会比较高。

美团 DSP 在点击价值预估上经历了两个阶段：

- 第一阶段是站外广告的落地页是广告的详情页面时，广告的点击价值预估比较简单，只需要预估出站外流量到达广告详情页之后的 CVR 即可。正负样本的选择也比较简单，采集转化样本为正样本，采集浏览未转化样本作为负样本。我们也进行了适当的负样本采样和真实 CVR 校准，这里采用的方法跟上一节类似，不再赘述。
    - 模型方面，在控制特征复杂度的基础上，我们选择了效果不错的二次模型 FFM，复杂度和性能都能够满足线上的性能。
    - 特征方面，我们使用了站外实时特征 + 部分离线挖掘特征，由于 FFM 预测复杂度是 (k*n*n)，k 是隐向量长度，n 是特征的个数，特征的选择上需要挑选贡献度比较大的特征。
- 第一阶段投放之后，经过统计，详情页的用户流失率非常高，为了降低流失率，我们开发了广告二跳页，在二跳页里面，用户在站外点击的广告排在第一位，剩下的是根据我们的召回策略和排序策略决定的。根据公式 1，点击价值是由二跳页的 k 个广告共同决定的。但是在站外广告排序和报价的过程中，我们无法获取中间页的召回结果，所以在实际情况中是无法适用的。目前我们的策略是直接对当前用户和当前商品的特征建立一个回归模型，使用用户在二跳页上成交的金额作为 label 进行训练，模型分别尝试了 GBDT 和 FM，最终采用了效果稍好些的 GBDT 模型。

效果评估和监控

离线评估

业内常用的量化指标是 AUC，就是 ROC 曲线下的面积。AUC 数值越大，模型的分别能力越强。

Facebook 提出了 NE（Normalized Entropy）[9] 来衡量模型，NE 越小，模型越好。

$$NE = \frac{-\frac{1}{N}\sum_{i=1}^{n}\left(\frac{1+y_i}{2}\log(p_i) + \frac{1-y_i}{2}\log(1-p_i)\right)}{-(p * \log(p) + (1-p) * \log(1-p))} \quad (3)$$

> N：训练的样本的数量。
>
> yi：第 i 个样本的 lable，点击为 +1，未点击为 −1。
>
> pi：第 i 个样本预估的点击率。
>
> P：所有样本的实际点击率。

离线我们主要使用的是 AUC 和 NE 的评估方法。

在线 AB 测试

通过在线 ABtest，确保每次上线的效果都是正向的，多次迭代后，站外 CTR 提升 30%，广告二跳页 CTR 提升 13%，二跳页 CVR 提升 10%。

在线监控

1. 在线 AUC 监控

在线预估的 CTR 和 CVR，建立小时级流程，计算每个小时的在线 AUC。发现 AUC 异常的情况，会报警，确保模型在线应用是正常的。

2. 在线预估均值监控

在线预估的值会计算出平均值，确保均值在合理的范围之内。均值过高会导致报价偏高，获取流量的成本增加。均值过低，造成报价偏低，获取的流量就偏少，对于估值异常的情况能及时响应。

结束语

本文介绍了美团 DSP 在站外投放过程中的策略实践。很多细节都是在业务摸索过程中摸索出来的。后续有些工作还可以更细致深入下去：

1. 流量筛选

流量筛选目前还是比较粗暴的根据网站历史的 CTR 等直接进行过滤，后续会基于用户的站内外的行为，对流量进行精细化的筛选，提升有效流量，提高转换。

2. 动态调整报价

- 在 DSP 的报价环节，点击率预估模型会对每一个流量预估出一个 CTR，为了适应 adx 市场的需要，会加上指数和系数项进行调整。但是通过这种报价方式获取的流量，由于外部竞争环境的变化，流量天然在不同时段的差异，经常会出现 CPC 不稳定。该报价的系数对于所有的媒体都是一致的，而一般的优质媒体都是有底价的，且不同媒体的底价不一致，造成该报价方式无法适用所有的媒体，出现部分优质媒体无法获取足够的流量。
- 我们的目标是在 CPC 一定的情况下，在优质媒体、优质时段尽可能多的获取流量，这里我们需要根据实时的反馈和期望稳定的 CPC 来动态调整线上的报价 [10]。从而在竞价环境、时段、媒体变化时，CPC 保持稳定，进一步保证我们的收益最大化（同样的营销费用，获取的流量最多）。

3. 位置召回

基于位置的召回策略中，我们对用户的商圈属性没有作区分，比较粗粒度的统一召回，这样其实容易把用户当前时间 / 位置真正有兴趣的商品拍的比较靠后；比较好的办法是通过精准的用户画像和用户消费时间 / 位置上下文挖掘，根据用户竞价时的位置和时间，分析出用户转化率高的商圈，从而进行更加精准的投放。

在业务上，美团 DSP 会逐步接入市场上主流的 AdExchange 和自有媒体的流量。技术上，会持续探索机器学习、深度学习在 DSP 业务上的应用，从而提升美团 DSP 的效果。

参考文献

1. https://en.wikipedia.org/wiki/Demand-side_platform
2. https://en.wikipedia.org/wiki/Ad_exchange
3. https://developers.google.com/ad-exchange/rtb/cookie-guide?hl=en

4. https://en.wikipedia.org/wiki/Return_on_investment
5. http://www.xiutx.cn/archives/263
6. https://www.csie.ntu.edu.tw/~cjlin/libffm/
7. http://baike.baidu.com/view/1666309.htm
8. http://book.douban.com/subject/26596778/
9. http://www.herbrich.me/papers/adclicksfacebook.pdf
10. https://en.wikipedia.org/wiki/PID_controller

作者简介

鸿杰，美团平台与酒旅事业群用户增长策略负责人，曾就职于阿里，2015 年加入美团点评。主要致力于通过机器学习提升美团点评平台的活跃用户数，作为技术负责人，主导了站外渠道投放、站内新客运营等项目的算法工作，提升营销效率，有效降低营销成本。

李乐，美团点评美团平台与酒旅事业群用户增长组 DSP 业务基础召回和设备下载的负责人，2014 年 7 月从浙江大学硕士毕业后加入美团。负责过 CPS 搜索广告、新客运营、DSP 基础召回、DSP 设备下载等业务，致力于推动全网设备的精准触达。

大数据

美团点评数据平台融合实践

语宸

背景

互联网格局复杂多变，大规模的企业合并重组不时发生。原来完全独立甚至相互竞争的两家公司，有着独立的技术体系、平台和团队，如何整合，技术和管理上的难度都很大。2015 年 10 月，美团与大众点评合并为今天的“美团点评”，成为全球规模最大的生活服务平台。主要分布在北京和上海两地的两支技术团队和两套技术平台，为业界提供了一个很好的整合案例。

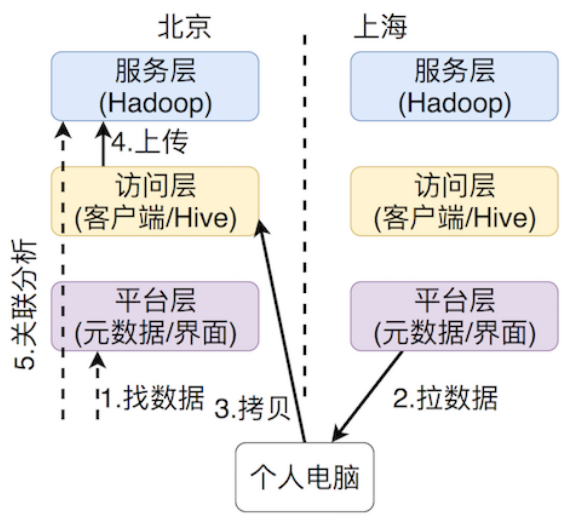
本文将重点讲述数据平台融合项目的实践思路和经验，并深入地讨论 Hadoop 多机房架构的一种实现方案，以及大面积 SQL 任务重构的一种平滑化方法。最后介绍这种复杂的平台系统如何保证平稳平滑地融合。

两家公司融合之后，从业务层面上，公司希望能做到“1+1>2”，所以决定将美团和大众点评两个 App 的入口同时保留，分别做出各自的特色，但业务要跨团队划分，形成真正的合力。比如丽人、亲子、结婚和休闲娱乐等综合业务以及广告、评价 UGC 等，都集中到上海团队；而餐饮、酒店旅游等业务集中到北京团队。为了支撑这种整合，后台服务和底层平台也必须相应融合。

点评 App 和美团 App 的数据，原来会分别打到上海和北京两地的机房，业务整合之后，数据的生产地和数据分析的使用地可能是不一样的。同时，随着公司的融合，我们跨团队、跨业务线的分析会越来越多，并且还需要一些常态化的集团级报表，包括流量的分析表、交易的数据表，而这些在原来都是独立的。

举个例子，原点评侧的分析师想要分析最近一年访问过美团和大众点评两个 App 的重合用户数，他需要经过这样一系列的过程：如下图所示，首先他要想办法找到数

据，这样就需要学习原美团侧数据平台元数据的服务是怎么用的，然后在元数据服务上去找到数据，才能开始做分析。而做分析其实是一个人工去做 SQL 分解的过程，他需要把原点评侧的去重购买用户数拉下来，然后发到原美团侧的数据平台，这个环节需要经历一系列的操作，包括申请账号、下载数据、上传数据，可能还会踩到各种上传数据限制的坑等等。最终，如果在这些都走完之后想做一个定期报表，那他可能每天都要去人工处理一回。如果他的分析条件变了怎么办？可能还要再重新走一遍这个流程。



所以他们特别痛苦，最终的结果是，分析师说：“算了，我们不做明细分析了，我们做个抽样分析吧！”最后他做了一个在 Excel 里就能做的去重数据量的分析。我们作为平台开发的同学来说，看到这个事情是非常羞愧的。那怎么办呢?

在经过一些磨合后，我们得出一个结论，就是必须进行数据口整合。

融合实践

确立目标

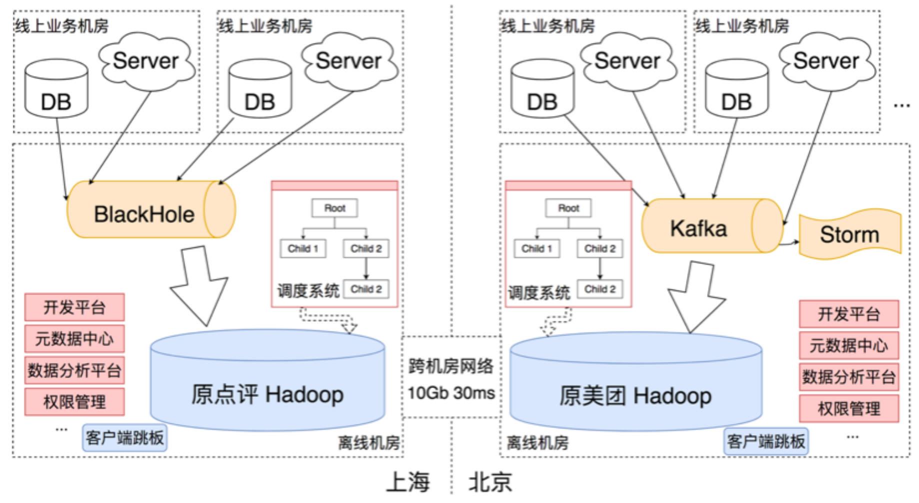
我们定了一个整体的目标，希望最终是能做到一个集群、一套数据平台的工具、一套开发规范。但是这个目标有点大，怎么把它变的可控起来呢？首先至少看来是一

个集群，也就是说从用户访问的角度上来讲，他通过一个 Client 或一套用户视图就能访问。工具方面至少明确已有的两套，哪些是新的员工进来之后还需要学，哪些是未来会抛弃掉的。最终，让大家认同我们有了一套数据平台规范，虽然这套规范短期内还没有办法做到完美。我们做的这些权衡其实是为了从整体上能将问题收敛。

但即使我们把这个目标缩小了，想要达到也是很难的。难点在哪呢?

难点

架构复杂，基础设施限制



如上图所示，整个数据平台基本上分为数据接入、数据开发、数据分析、数据输出等等几个阶段。我这里只列了其中涉及到跨机房、跨地域的部分，还有很多数据平台产品的融合，在这里就不赘述了。在两个公司融合之前，原点评侧和美团侧都已经在同地域进行多机房的部署了，也都很 " 默契 " 地抽象出了离线的机房是相对独立的。在线的业务机房不管是通过消息队列还是原点评自己当时做的 Blackhole（一个类似 DataX 的产品），都会有一系列数据收集的过程、对应任务的调度系统和对应的开发工具，也会有一些不在数据开发体系内的、裸的开源客户端的跳板机。虽然架构大体一致，但是融合项目会牵扯整套系统，同时我们有物理上的限制，就是当时跨机房带宽只有 10Gb。

可靠性要求

由于团购网站竞争激烈，两家公司对于用数据去优化线上的一些运营策略以控制运营成本，以及用数据指导销售团队的管理与支撑等场景，都有极强的数据驱动意识，管理层对于数据质量的要求是特别高的。我们每天从零点开始进行按天的数据生产，工作日 9 点，老板们就坐在一起去开会，要看到昨天刚刚发生过什么、昨天的运营数据怎么样、昨天的销售数据怎么样、昨天的流量数据怎么样；工作日 10 点，分析师们开始写临时查询，写 SQL 去查数据，包括使用 Presto、Hive，一直到 22 点；同时数据科学家开始去调模型。如果我们集群不能 work，几千人每天的工作就只能坐在电脑面前看着 Excel……

当时的分析是这样，如果考虑回滚的情况下，我们运维的时间窗口在平日只有一个小时，而且要对全公司所有用数据的同学进行通告，这一个小时就是他们下班之后，晚上 6 点至 7 点的时候开始，做一个小时，如果一个小时搞不定，回滚还有一个小时。周末的话好一点，可以做 4 小时之内，然后做全面的通告，相当于整个周末大家都没法加班了，他们是非常不开心的。

融合前	节点数	数据量	日生成数据量
上海（原点评）	500+	11P	120T
北京（原美团）	3000+	75P	800T

融合前	仓库任务数	业务团队数	开发平台日活	查询平台日活
上海（原点评）	7000+	28	100+	400+
北京（原美团）	14000+	50+	240+	900+

体量

虽然没有到 BAT 几万台节点的规模，但是也不算小了，融合时原点评的节点数

是 500 个，数据量是 11 个 P；原美团的节点数是 3000 个，现在整体已经上 6000 了。这里有一个比较关键的数据就是每天生成的数据量，由于我们的集群上面以数仓的场景为主，会有很多重新计算，比如说我要看去年每月的去重，这些都是经过一些时间变化之后会进行重算的。它对于分析数据的迭代速度要求很高，我每天可能都会有新的需求，如果原来的数据表里面要加一个字段，这个字段是一个新的统计指标，这个时候我就要看历史上所有的数据，就得把这些数据重新跑一遍。这里的生成数据量其中有 50% 是对历史的替换，50% 是今天新增的。这对于后面我们拷数据、挪数据是比较大的挑战。

平台化与复杂度

两家公司其实都已经慢慢变成一个平台，也就是说数据平台团队是平台化的，没法对数据的结果分析负责，数据平台团队其实对外暴露了数据表和计算任务这两种概念。平台化以后，这些数据表的 owner 和这些数据任务的 owner 都是业务线的同学们，我们对他们的掌控力其实是非常差的。我们想要改一个表的内容、一个数据任务的逻辑，都是不被允许的，都必须是由业务侧的同学们来做。两侧的平台融合难免存在功能性的差异，数据开发平台的日活跃就有 100 和 240，如果查询就是每天作分析的日活跃的话，原点评和美团加起来有 1000 多。所以在平台融合过程中，能让这么多用户觉得毫无违和感是非常有挑战的。

综上，我们做了一个项目拆解。

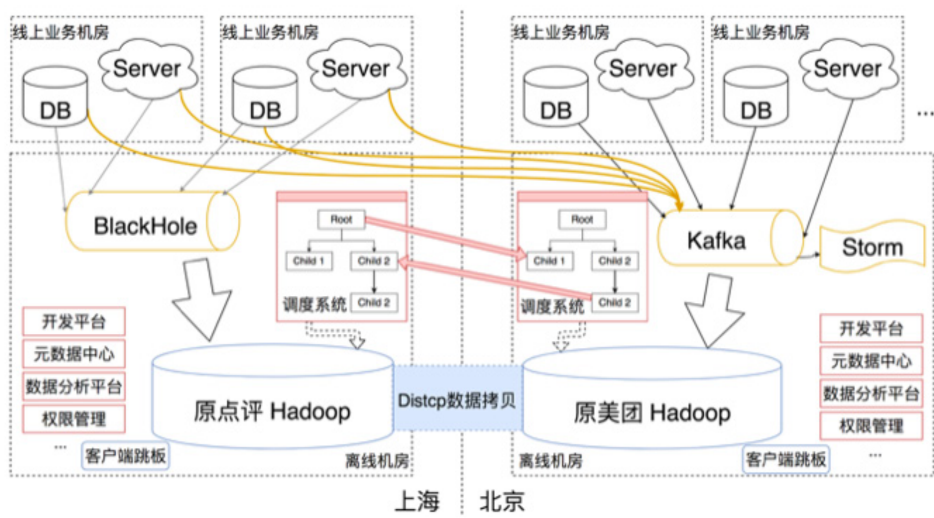
项目拆解

数据互访打通

数据互访打通其实是最早开始的，早在公司宣布融合以后，我们两侧平台团队坐在一起讨论先做什么，当时做了一个投入产出比的权衡，首要任务是用相对少的开发，先保障两边分析师至少有能在我们平台上进行分析的能力。接着是让用户可以去配置一些定时任务，通过配置一些数据拷贝任务把两地数据关联起来。

在这方面我们总共做了三件事。

原始层数据收集



在原美团侧把原点评侧线上业务机房一些 DB 数据以及 Server 的 log 数据同步过来。这个时候流式数据是双跑的，已经可以提供两边数据合在一起的分析能力了。

集群数据互拷

集群数据互拷，也就是 DistCp。这里稍微有一点挑战的是两边的调度系统分别开了接口，去做互相回调。如果我们有一份数据，我想它 ready 之后就立即拷到另外一边，比如原点评侧有个表，我要等它 ready 了之后拷到原美团侧，这个时候我需要在原美团侧这边配一个任务去依赖原点评侧某一个任务的完成，就需要做调度系统的打通。本文主要讨论大数据框架的部分，所以上面的调度系统还有开发平台的部分都是我们工具链团队去做的，就不多说了，下文重点描述 DistCp。

其实 Hadoop 原生支持 DistCp，就是我起一个 MapReduce 在 A 集群，然后并行地去从 B 集群拖数据到 A 集群，就这么简单。只要你网络是通的，账号能认（比如说你在 A 集群跑的任务账号能被 B 集群认），并且有对应的读权限，执行端有计算资源，用开源版本的 DistCp 就可以搞定。

这方面我们做了一些权衡：

首先是因为涉及到带宽把控的问题，所以同步任务是由平台团队来统一管理，业

务侧的同学们提需求。

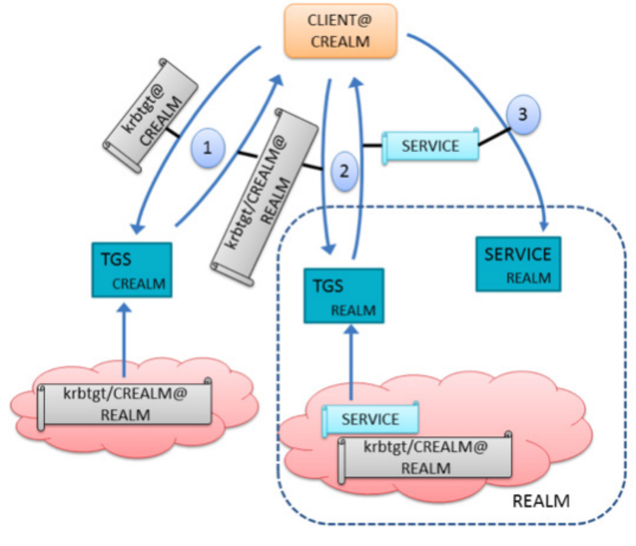
然后我们两侧集群分别建立一个用于同步的账号，原则是在读的那一端提交任务。什么叫“读的一端”？比如说我想把一个原点评侧的数据同步到原美团侧，原美团侧就是要读的那端，我在原美团侧起这个任务去读原点评侧的数据，然后写到原美团侧。这里的主要考虑是读端更多是需求端，所以，他要在他的资源池里去跑。另外，对集群的影响读小于写，我们希望对于被读集群的影响能尽量减少。

当然，这都是一些临时的项目，投入较小，但收益是磨合了两地团队。

Kerberos 跨域认证架构

接着介绍一下认证部分是怎么打通的。原美团侧和点评侧恰好都用了 Kerberos 去做认证服务，这个 Kerberos 在这我不去详细展开，只是简单介绍一下。首先是 KDC 会拥有所有的 Client 和 Server，Client 就是 HDFS Client，Server 就是 Name Node，KDC 会有 Client 和 Server 的密钥，然后 Client 和 Server 端都会保有自己的密钥，这两个甚至都是明文的。所有的密钥都不在传输过程中参与，只拿这个密钥来进行加密。基于你能把我用你知道的密钥加密的信息解出来，这一假设去做认证。这也是 Kerberos 架构设计上比较安全的一点。

Kerberos 不细讲了，下面详细讲一下 Kerberos 跨域认证架构。



一般公司都不会需要这个，只有像我们这种两地原来是两套集群的公司合并了才需要这种东西。我们当时做了一些调研，原来的认证过程是 Client 和 KDC 去发一个请求拿到对应 Server 的 ticket，然后去访问 Server，就结束了。但是如上图所示，在这里它需要走 3 次，原来是请求 2 次。大前提是两边的 Kerberos 服务，KDC 其中的 TGS 部分，下面存储的内容部分分别要有一个配置叫 krbtgt，它有 A realm 依赖 @ B realm 这样的一个配置。两边的 KDC 基于这个配置是要一致的，包括其中的密码，甚至是包括其中的加密方式。那这个时候我们认为这两个 KDC 之间实际上是相互信任的。

流程是 Client 发现要请求的 Server 是在另外一个域，然后需要先去跟 Client 所属的 KDC 发请求，拿一个跨域的 ticket，就是上图中 1 右边那个回来的部分，他拿到了这个 krbtgt CREALM @ REALM。然后 Client 拿着跨域的 ticket 去请求对应它要访问 Service 那一个域的 KDC，再去拿对应那个域的 Service 的 ticket，之后再去访问这个 Service。这个流程上看文档相对简单，实则坑很多，下面就讲一下这个。

- **两个KDC同时建立两个principal，krbtgt/A@B, krbtgt/B@A**
  - 要求秘钥编码一致，具体查手册
- **krb5.conf 配置对应realm的KDC位置，并能通过domain_realm策略找到**
  - 使用A realm的principal进行认证(kinit)，使用kvno命令尝试获取B realm的server principal来确定是否跨域认证成功
- **hadoop client端 dfs.namenode.kerberos.principal.pattern 以及yarn.resourcemanager.principal.pattern 设置为***
  - 绕过hadoop对于service principal 判断是否合法
- **在所有RPC Server端配置 hadoop.security.auth_to_local 规则**
  - 以上两条开-Dsun.security.krb5.debug 看log，hadoop代码中有存在对于realm隐改的行为

上图是 Kerberos 跨域认证的一些要求。

首先第一个比较大的要求就是密钥的编码一致，这有一个大坑，就是你必须让两个 KDC 拿到的信息是一样的，它们基于这个信息去互信，去互相访问。然后 krb5.conf 里面有一些比较诡异的 domain_realm 策略，这个在你网络环境不一致的时候会有一定的影响，包括 DNS 也会影响这个。在你的网络环境比较不可知的时

候，你需要做做测试，尝试去怎么配，然后在 Hadoop 端有两个配置需要做，分别在 Server 端和 Client 端配置即可。其中比较恶心的是说，在测试的过程当中，需要去看 Hadoop 的详细日志，需要开一下它的 Debug，然后去看一下它真正请求的那个域是什么样的。因为我们翻代码发现，Hadoop 底层有对 log，Client 去请求 realm 的隐改，就是说我认为我应该是这个 realm 啊，它为什么传出来的是另外一个 realm？这个是比较坑的一点。

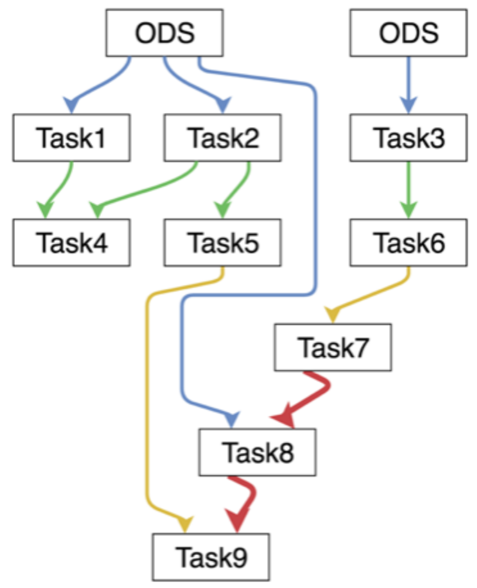
我们做完这个项目之后，分析师就可以愉快地配置一些调度任务去同步数据，然后在对应的集群上去关联他们的数据进行分析了。做完这个项目之后，我们两边的团队也相互磨合，相互形成了一定的认可。因为这个小项目涉及到了数据平台的每一个领域，包括工具链、实时计算、离线的团队都做了一些磨合。

集群融合

粗看起来，打通了数据平台，我们的大目标似乎已经完成了：一个集群、一套数据平台的工具、一套开发规范。把数据拷过来，然后重新改它的任务，就可以形成在统一的一套工具和规范里面用一个集群，然后慢慢把原来团队维护的服务都下掉就好了。事实上不是这样的，这里面有大量的坑。如果接下来我们什么都不做的话，会发生什么情况呢?

数据 RD 会需要在迁移的目标平台重建数据，比如说我们都定了，以后把原美团侧平台砍掉，那么好，以后都在原点评侧的平台，包括平台的上传工具、平台的集群去使用、去开发。这个时候，至少原美团侧的同学会说："原点评那边平台的那些概念、流程，可能都跟我不一样啊，我还需要有学习的时间，这都还好"。但他们怎么迁移数据呢? 只能从源头开始迁移，因为对端什么都没有，所以要先做数据的拷贝，把上游所有的表都拷贝过去。然后一层一层地去改，一整套任务都要完全重新构建一遍。

那我们有多少任务呢?



我们当时有 7000 个以上，后来超过 8000 个任务，然后我们平均深度有 10 层。也就是说上游先迁过来，然后下游才能迁。整个流程会变成数据表的拷贝，然后上线任务进行双跑。因为必须得有数据的校验，我才能放心地切过来，花的时间大概是拷贝数据 1~4 天，然后改代码加测试再加双跑，可能要 3~5 天。这里我们有一个流水线的问题，如上图所示，蓝色的部分只有一层依赖的，当然我把这个左边的 ODS 都迁完了之后，1 层依赖的 Task 1、Task 2、Task 3、Task 8 中，Task 1、2、3 就可以迁了，但是 Task 8 还是不能迁的，因为 Task 8 依赖的 Task 7 还没过来。我再走一层，Task 4 的负责人要等上游相关任务都迁完了之后才能干活，那整个这个迁移就纯线性化，我们大概估了一下，并行度不会超过 50。如果是两地两份数据，这个项目的周期会变成特别长，会有长期的两份数据、两份任务。这个时候，第一是我们真存的下吗？第二是如果我要迁移出来那个方向的业务有需求的变更，我怎么改？我要两边都再改一遍？所以这个是非常不可控的。

那这个时候怎么办？

集群融合的问题本质

反思一下这个问题的本质，首先我们是不能双跑的，因为一旦双跑，我们必须

有常态化的两份数据，然后衍生一系列的校验、存储量、切换策略等问题。所以我们必须得有一套数据，一套任务执行机制。后续任务的改变，不管是替换工具链上的东西，替换计算引擎，比如说让两边 Hive、Spark 和 UDF 去做一致化的时候，其实本质上是说对单个任务的修改，对每个任务灰度的修改就好了。

所以我们推断出，必须自底向上地去进行融合，先合集群，然后后续再推动上游平台和引擎的融合。

集群融合的解决思路

整体我们融合的思路是这样的，集群融合先行，两边的 Hadoop 的服务架构和代码先进行统一，其次拷贝原点评侧集群的 Block，同步到原美团侧机房的两个副本。这里有一个大的前提，第一个是原点评侧的集群节点数相对来讲确实小，再一个就是原点评侧的机房确实放不下了，它当时只能扩容到 10 月，再往后扩就装不下机器了。

所以我们将原点评侧的集群，合并到原美团侧机房，然后进行拷贝和切换。我们让整个这个集群变成在原美团侧机房一样的样子，然后进行融合。我们会把上面的客户端和元数据统一，使得访问任何一个集群的时候，都可以用一套客户端来做。一旦我们做到这个样子之后，基于统一的数据、集群的元数据和访问入口之后，我们上面的工具链就可以慢慢地去做一个一个机制，一个一个模块的融合了。

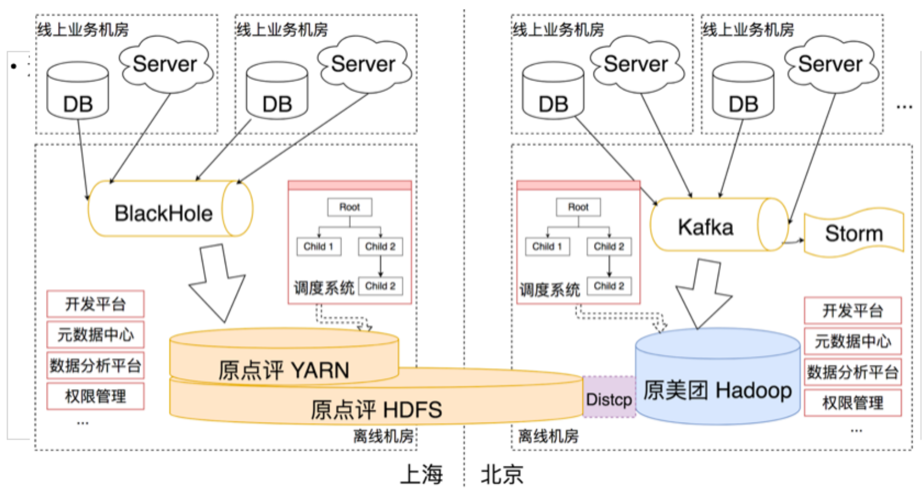
简单总结下来就是四步：统一、拷贝、切换、融合，下面我们来展开说一下这四步。

环境对比	北京侧	上海侧
JDK	1.7.0_76	1.6.0_43
Hadoop	2.7.1	2.4.1
HDFS Arch.	HA using QJM, Federation	无
Hive	0.13, 1.2	0.11
Kerberos	keytab	keytab, token, password
日志收集	自研Agent, Kafka, camus	自研收集服务 BlackHole
任务调度	自建	自建
……	……	…….

统一

第一优先级要解决的是上图中标红的部分，两边的 Hadoop 版本是不一样的，我们需要将原上海侧的版本变成我们的 2.7.1 带着跨机房架构的版本。同时因为我们后面要持续地去折腾 Hadoop 集群，所以必须先把原上海侧的 HDFS 架构改全，改成高可用的。

这里有一个小经验就是，我们自研的 patch 对改的 bug 或者是加的 feature，一定要有一个机制能够管理起来，我们内部是用 Git 去管理的，然后我们自研的部分会有特殊的标签，能一下拉出来。我们当时其实互相 review 了上百个 patch，因为当时两个团队都有对集群，包括 Hive 等等这些开源软件的修改。这是统一的阶段，相对容易，就是一个梳理和上线的过程。接下来是拷贝的阶段。



拷贝

上图是最终的效果图，同步在运行的打通任务还是用 DistCp，然后先把原点评侧的 HDFS 跨机房部署。但是这个时候原点评侧的 YARN 还是在上海机房。在这个过程当中，因为 HDFS 跨机房部署了，所以原新上线的 DataNode 可以承载更多在原点评侧集群的冷数据。这个过程是慢慢进行拷贝的，大概持续了 4 个月，中间长期都是 10Gbps 的小管子。

切换

这个相当于把原点评侧的 NameNode（这个时候还没有彻底下线）切换到原美团侧机房，然后把对应的 YARN 重新启动起来。这里有一个小 trick 就是原美团侧机房的承载能力，大概是 1000 多台节点，是原点评侧的两倍，所以我们才能做这个事，最近我们刚刚把上海机房的节点迁完。

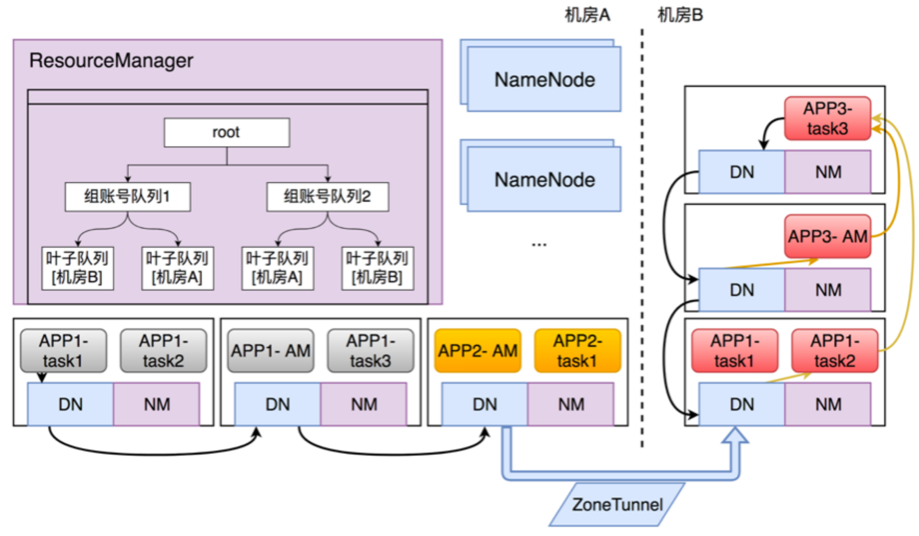
那整个集群的拷贝和切换是怎么做的呢？其实就是用我们自研的一套 Hadoop 多机房架构。可能做 Hadoop 集群维护管理的同学们对这个有深刻的体会，就是不时地就要从一个机房搬到另一个机房。设计目标是说我们一个 Hadoop 集群可以跨机房去部署，然后在块的力度上能控制数据副本的放置策略，甚至是进行主动迁移。

设计是怎么做的呢？整个 Hadoop 原生的架构其实没有机房这个概念，只支持 Rack 也就是机架，所有服务器都被认为是在同一个机房的。这个时候不可避免地就会有很多跨机房的流量，就如果你真的什么都不干，就把 Hadoop 跨机房去部署的话，那么不好意思，你中间有好多的调用和带宽都会往这儿走，最大的瓶颈是中间机房网络带宽的资源受限。

我们梳理了一下跨机房部署的时候大概都有哪些场景会真正引发跨机房流量，基本上就这 3~4 个。首先是写数据的时候，大家知道会 3 副本，3 个 DataNode 去建 pipeline，这个时候由于是机器和机器之间建连接，然后发数据的，如果我要分机房部署的话，肯定会跨机房。那我要怎么应对呢？我们在 NameNode 专门增加 zone 的概念，相当于在 Rack 上面又加了一层概念，简单改了一些代码。然后修改了一下 NameNode 逻辑。当它去建立 pipeline 的时候，在那个调用里面 hack 了一下。建 pipeline 的时候，我只允许你选当前这个 Client 所属的 zone，这样写数据时就不会跨机房了。

这些 Application 在调度的时候有可能会在两个机房上，比如说 mapper 在 A 机房，reducer 在 B 机房，那么中间的带宽会非常大。我们怎么做的呢？在 YARN 的队列里面，也增加 zone 的概念，我们用的是 Fair Scheduler。在队列配置里面，对于每一个叶子队列，都增加了一个 zone 的概念。一个叶子队列，其实就是对应了这个叶子队列下面的所有任务，它在分配资源的时候就只能拿到这个 zone 的节点。读

取数据的时候有可能是跨机房的，那这个时候没有办法，我们只有在读取块选择的时候本地优先。我们有一些跨机房提交 job 的情况，提交 job 的时候会把一些 job 里面的数据进行上传，这个时候加了一些任务的临时文件上传的是任务所在的目标机房。这里做一些简单的改动，最重要的是提供了一个功能，就是我们在拷贝数据的时候，其实用 balancer 所用的那一套接口，我们在此基础之上做了一层 Hack，一层封装。形成了一个工具，我们叫 ZoneTransfer，又由它来按照我们一系列的策略配置去驱动 DataNode 之间的跨机房的 block 粒度的拷贝。



上图是我们跨机房架构的架构图，下面的 Slave 里面有 DN(DataNode) 和 NM(NodeManager)，上面跑的同颜色的是一个 App。我们在 RM(ResourceManager) 里面的叶子队列里配置了 zone 的概念，然后在调度的时候如大家所见，一个 App 只会在一个机房。然后下面黑色的线条都是写数据流程，DN 之间建立的 pipeline 也会在一个机房，只有通过 root 去做的，DN 之间做数据 transfer 的时候才会跨机房进行，这里我们基本上都卡住了这个跨机房的带宽，它会使用多少都是在我们掌控之内的。

在上线和应用这个多机房架构的时候，我们有一些应用经验。

首先在迁移的过程当中我们需要评估一点就是带宽到底用多少，或者说到底多长

时间之内能完成这个整体数据的拷贝。这里需要面对的一个现实就是，我们有很多数据是会被持续更新的。比如我昨天看到这个块还在呢，今天可能由于更新被删，那昨天已经同步过来的数据就白费了。那我昨天已经同步过来的数据就白费了。所以我们定义了一个概念叫拷贝留存率。经过 4 个月的整体拷贝，拷贝留存率大概是 70% 多，也就是说我们只有 70% 的带宽是有效的，剩下的 30% 拷过去的数据，后面都被删了。

第二个是我们必须得有元数据的分析能力，比如说有一个方法能抓到每一个块，我要拷的块当前分布是什么样子。我们最开始是用 RPC 直接裸抓 Active NameNode，其实对线上的影响还是蛮大的。后面变成了我们通过 FsImage 去拉文件的列表，形成文件和块的列表，然后再到把请求发到 standby，那边开了一个小口子，允许它去读。因为 FsImage 里面是没有 block 在哪一个 DataNode 的元信息的。

这里需要注意的一点就是，我们每天都会有一个按天的数据生产，为了保证它的一致性，必须在当天完成。在切换之前，让被切换集群的 NN（NameNode）进入 SafeMode 的状态，然后就不允许写了，所有的写请求停止，所有的任务停止。我们当时上线大概花了 5~6 个小时吧，先停，然后再去拷贝数据，把当天的所有新生产的数据都拷过来，然后再去做操作。这里最基本的要做到一点就是，我们离线的大数据带宽不能跟线上的服务的带宽抢资源，所以一定要跟基础设施团队去商量，让他们做一些基于打标签的带宽隔离策略。

融合

当我们把集群搬到了原美团侧的机房之后，又做了一层融合。想让它看起来像一个集群的样子，基本上只需要 3 步。首先是“把冰箱门打开”，把原点评侧集群的那个 NN 作为一个 federation 合到原美团侧的集群，只需要改 cluster ID，去客户端改 mount table 配置，cluster ID 是在元数据里面。第二个是对 Hive 进行元数据的融合。我们恰好两侧元数据存储都是用 MySQL 的，把对应的表导出来，灌到这边，然后持续建一个同步的 pipeline。它是长期活动的，到时候把上传的服务一切就可以。

前面说的那个做了跨域认证的配置我们还是要拆掉的，必须进行服务认证的统一，不然的话以后没法看起来像一个集群，这个时候把原来的 KDC 里面的账号进行

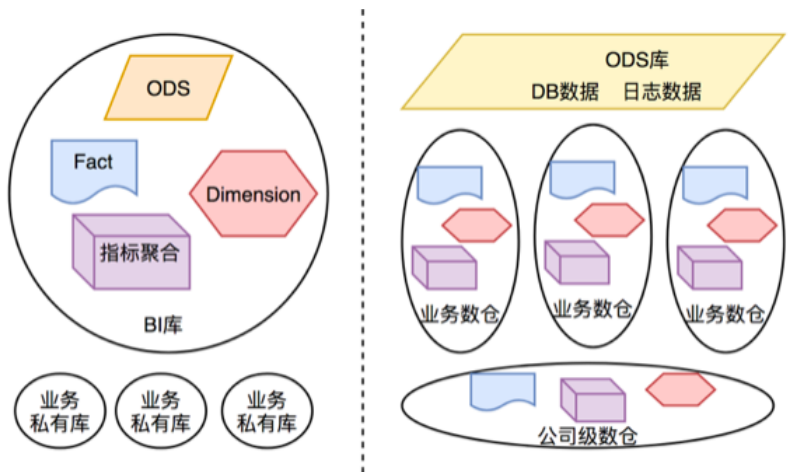
导出，之后逐步地去切换每一个配置，让它慢慢切到新的 KDC。切的过程当中，我们各种请求还是有跨域情况的，我们认为两个域是一体的，是一样的。等切干净之后，也就是原来的 KDC 没有请求了之后，我们再把它干掉。

开发工具融合

集群融合结束后，我们就做了开发工具的融合。由于这个跟大数据基础架构这个主题关系不是特别大，开发工具都是我们内部自研的，涉及的程序也很复杂，是一个特别大的项目，涉及一系列复杂的工具，每个模块的融合、打通。所以这个暂时不讲了。另外我觉得比较有意思的是下面这一点，就是原点评侧的一个拆库，这个在很多公司的数据平台慢慢扩大的过程当中可能会用到。

原点评侧拆库

难点



先说一下背景，由于原点评和原美团整体历史上发展经验、周期和阶段不同，如上图所示，原点评侧的数据仓库是先有的 Hadoop 集群，后有的数据仓库平台，因此有很多平台完全没法掌控的私有库，但是他们对于数仓所在库的掌控是非常强的，所有的任务都在这一个大的 Hive 库里面，里面有七八千张表。而原美团侧是先有的数

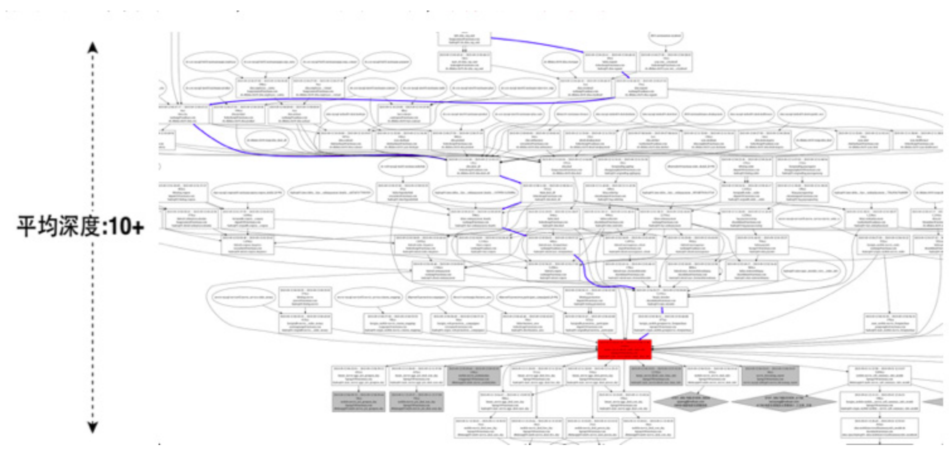
据平台，后来因为数据平台整个体量撑不住了，底层改成了 Hadoop。同时在平台化的演进过程中，已经慢慢把各个业务进行独立拆分了，每个业务都有一个独立的私有库，简单来说就是库名和库名的规范不一样。我们希望能让这两套规范进行统一。

我们如何去做呢?

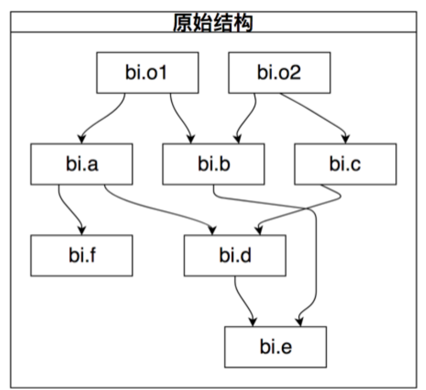
原来任务的内容大概是 insert into 一个 BI 库里面的一张表，接着 select from BI 库里面的某两张表，然后 where group by。像这样的任务我们有七八千个，它们在我们平台上配置着每天的依赖调度。我们希望把它都改成下图中的样子。所有涉及到的这些表都需要改名字，说白了就是一个批量改名字的事儿。

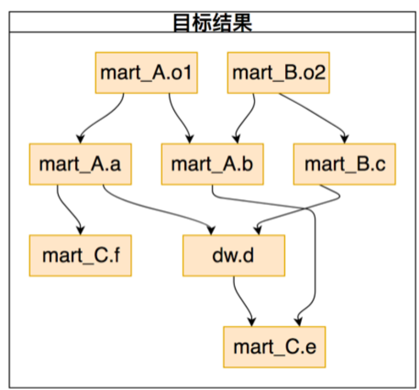
```
原任务内容:
insert into bi.table_a
select x, y, z
from bi.table_b join bi.table_c on ***
where *** group by ***

希望改写成
insert into mart_xxx.table_a
select x, y, z
from mart_yyy.table_b join mart_zzz.table_c on ***
where *** group by ***
```

改名字听起来很简单，实际上并不是，我们有近 8000 个这样的任务需要改，同时这些任务相互之间都有非常复杂的依赖。下图是我随便找的一个，原美团侧某一个任务所有上游和下游的依赖关系图，如此复杂，任务的平均深度大概有 10 层，这还是平均数，最严重的可能要有大几十层。如果我们改这里面的任务表达，就只能分层推动。但是，当我们每改其中一个的时候，可能上下游都得跟着改，具体是什么样子的呢?



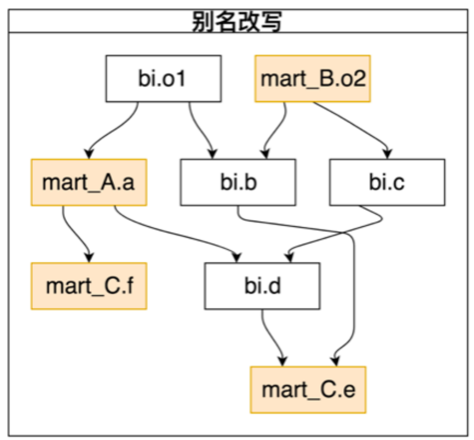
下图是我们的原始结构，首先这里有一个大前提是每一个任务只对一个结果表。原始的结构中，a 表只依赖 o1 表，b 表依赖 o1、o2，然后 c 表只依赖 o2，它们之间相互关联。这时候我希望可以对库名和表名进行一次性的修改。那如果我们逐层地去改写怎么办呢？首先要先把最上层的 mart 表改了，而我一旦改上游的某一个表，所有跟对它有依赖的表都必须改任务内容。每推动一层改动，下面一层都要变动多次，这样一来，我们这个流程就非常受限。





刚刚那个情况基本上是类似的，就是说我们对它们的改动没法批量化、信息化、流水线化，所有的用户和数据开发们，需要跟我们去聊，最近改了多少，然后谁谁谁没改完，谁谁谁又说要依赖他，整个依赖图是非常大的，我们整个项目又不可控了。那怎么办呢?

解决方案



很简单，我们只干了一件事情，就是在 Hive 层面上进行了一波 Hack。比如说我要让原来叫 bi.o2 的表未来会变成 mart_b.o2，我就同时允许你以 mart_b.o2 和 bi.o2 这两种方式去访问 bi.o2 这张表就好了。不管是写入还是读取，我们只需要在 Hive 的元数据层面去做一层 Hack，然后做一个对应表，这个对应表我们是有规范的、能梳理出来的。在这之后，任何一个人都可以把他的任务改写成他希望的样子而不受任何影响，他写的那些表还是原来的那些表，真正在物理上的存在还是 bi. 什么什么这样的表，我们整个项目就 run 起来了。

具体的实施流程是这样，首先先梳理业务，确定整体的映射关系。然后 Hive 元数据入口上去做别名能力，我们是在 Hive metaserver 里面去改的，大部分请求都在这里面，包括 Spark 的、Presto 的、Hive 的等，都能兼容掉，推动分批次改写，单任务内以及任务链条内完全不需要做依赖关系的约束，最终真正实现的是自动化地把 SQL 文本替换掉了。业务的同学们只需要批量看一个检测报告，比如说数据对应上有没有什么问题，然后一键就切了。

我们用了一个季度业务侧来磨合、尝试练习和熟练，同时做工具的开发。然后第二个季度结束后，我们就完成了 7000 多个任务中 90%SQL 任务批量的改写。当任务都切完了之后，我们还有手段，因为所有的请求都是从 Hive 的 metaserver 去访问的，当你还有原有的访问模式的时候，我就可以找到你，你是哪一个任务来的，然

后你什么东西改没改，改完了之后我们可以去进行物理上的真正切分，干掉这种元数据对应关系。

物理上的真正切分其实就是把原来都统一的库，按照配置去散到真实的物理上对应的库上，本质还是改 NN 一个事情。

总结与展望

未来——常态化多机房方案

我们目前正在做的一个项目，就是常态化地把集群跨机房去跑，其中最核心的就是我们需要对跨机房的数据进行非常强的管理能力，本质上是一个 Block 粒度 Cache 的事情，比如说 Cache 的击穿、Cache 的预热或者 Cache 的等待等等，都是一个 Cache 管理的事情。我们会引入一个新的 server，叫 zone Server，所有的 Client 请求，NameNode 进行块分布的时候，调整和修改。之后大家会在美团点评技术博客上看到我们的方案。

反思——技术换运营

数据平台做起来是很痛苦的，痛苦在哪儿呢？第一，数据平台对上层提供的不只是 RPC 接口，它要管的是数据表和计算任务。所以我们做 SLA 很难，但是我们还在努力去做。第二，就是最开始的时候一定是基于开源系统拼接出来的，然后再到平台化，这一定是一个规范的收敛，也是限制增多的过程。在这个过程中，我们必须去推动上面应用的、不符合规范的部分，推动他们去符合新的规范。平台的变更即使做到兼容，我们的整体收尾还是要尽快扫清的，不然整个平台就会出现同时进行大量灰度、每一个模块当前都有多种状态的情况，这是不可维护的。

综上，我们定义了一个概念叫“可运营性”，推动用户去做迁移、做改动是一个 " 运营的事情 "。可运营性基本上的要求如下。

- 可灰度。任务的改动是可灰度的。
- 可关门。当某一刻，我不允许你再新增不符合新规范的任务、表或者配置，我们内部叫“关门打狗”，就是说先把新增的部分限制住，然后再去慢慢清理老的。

- 进度可知。清理老的我们需要有一个进度可知，需要有手段去抓到还有哪些任务不符合我们新的规范。
- 分工可知。抓到任务的分工是谁，推动相关团队去改动。
- 变更兼容 / 替代方案。我们肯定过程中会遇到一些人说：不行，我改不动了，你 deadline 太早了，我搞不定。这时候得有一些降级或者兼容变更的一些方案。

那我们什么时候去使用技术降低运营成本呢？前面已经有两个例子，就集群的迁移和融合，还有 Hive 表别名去帮助他们改任务名，这都是用技术手段去降低运营成本的。

怎么做到呢？

第一是找核心问题，我们能否彻底规避运营、能不能自动化？在集群融合的过程当中，其实已经彻底避免了运营的问题，用户根本都不需要感知，相当于在这一层面都抽象掉了。第二，是即使我没法规避，那我能不能让运营变得批量化、并行化、流水线化、自动化？然后当你抓核心问题有了一个方案之后，就小范围去迭代、去测试。最后还有一点，引入架构变更的复杂度最终要能清理掉，新增的临时功能最后是可被下线的。

体会——复杂系统重构与融合

最后稍微聊一下复杂系统的重构与融合。从项目管理的角度上来讲，怎么去管控？复杂系统的重构还有融合本质上最大的挑战其实是一个复杂度管理的事情，我们不可能不出问题，关键是出问题后，对影响的范围可控。

从两个层面去拆分，第一个层面是，先明确定义目标，这个目标是能拆到一个独立团队里去做的，比如说我们最开始那四个大的目标，这样保证团队间能并行地进行推动，其实是一点流水线的思路。第二，我们在团队内进行目标的拆分，拆分就相对清晰了，先确定我要变更什么，然后内部 brainstorming，翻代码去查找、测试、分析到底会对什么东西产生影响，然后去改动、测试、制定上线计划。

内部要制定明确的上线流程，我记得当时在做的时候从 11 月到 12 月我们拆分了应该是有 11 次上线，基本上每次大的上线都是在周末做的，10、11、12 月总共

有 12 个周末，一共上线 11 次，大的上线应该是占了 7 到 8 个周末吧。要提前准备好如何管理依赖，如何串行化，然后准备上线，上线完怎么管理，这些都是在整个项目管理过程当中需要考虑的。

其中，两个可能大家都持续提的东西，第一个是监控，要知道改完了之后发生了什么，在改的时候就像加测试用例一样把改动部分的监控加好。第二要有抓手，如果我线上垮了，这个时候重复恢复的成本太高，也就是完全重启、完全回滚的成本太高，我能不能线上进行一些改动?

最后这张图，献给大家，希望大家在对自己系统改动的时候，都能像这哥们一样从容。

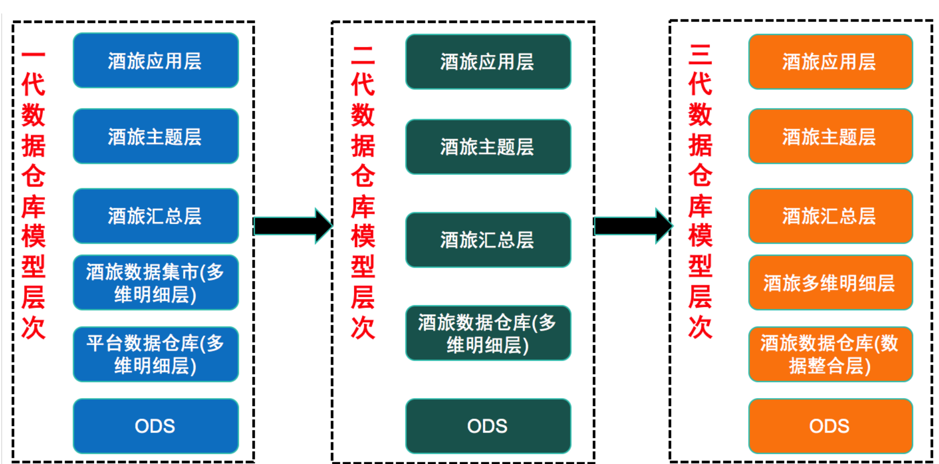
美团点评酒旅数据仓库建设实践

德臣

在美团点评酒旅事业群内，业务由传统的团购形式转向预订、直连等更加丰富的产品形式，业务系统也在迅速的迭代变化，这些都对数据仓库的扩展性、稳定性、易用性提出了更高要求。对此，我们采取了分层次、分主题的方式，本文将分享这一过程中的一些经验。

技术架构

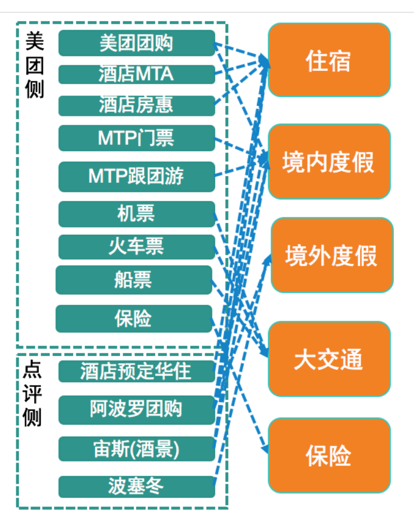
随着美团点评整体的系统架构调整，我们在分层次建设数据仓库的过程中，不断优化并调整我们的层次结构，下图展示了技术架构的变迁。



我们把它们简称为三代数仓模型层次。在第一代数仓模型层次中，由于当时美团整体的业务系统所支持的产品形式比较单一（团购），业务系统中包含了所有业务品类的数据，所以由平台的角色来加工数据仓库基础层是非常合适的，平台统一建设，支持各个业务线使用，所以在本阶段中我们酒旅只是建立了一个相对比较简单的数据集市。

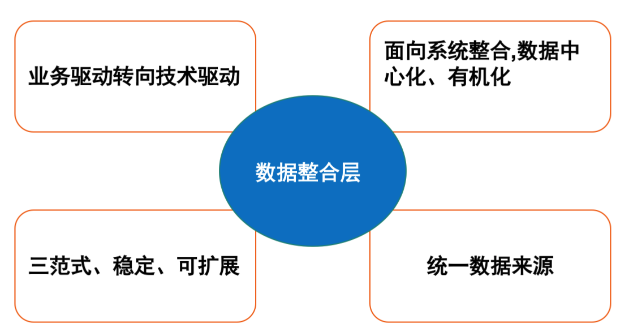
但随着美团原本集中的业务系统不能快速响应各个业务线迅速的发展与业务变化时，酒旅中的酒店业务线开始有了自己的业务系统来支持预订、房惠、团购、直连等产品形式，境内度假业务线也开始有了自己的业务系统来支持门票预订、门票直连、跟团游等复杂业务。我们开始了第二代数仓模型层次的建设，由建设数据集市的形式转变成了直接建设酒旅数据仓库，成为了酒旅自身业务系统数据的唯一加工者。由于系统调整初期给我们带来的重构、修改以及新增等数据处理工作非常大，我们采用了比较短平快的 Kimball 所提的维度建模的方式建设了酒旅数据仓库。

在第二代数仓模型层次运转一段时间后，我们的业务又迎来了一个巨大的变化，上海团队和我们融合了，同时我们酒旅自身的业务系统重构的频率相对较高，对我们的数仓模型稳定性造成了非常大的影响，原本的维度模型非常难适配这么迅速的变化。下图就是我们数仓模型当时所面临的挑战：



- 在用业务系统多达13个，系统与业务线之间关系错综复杂
- 产品结构多样，一层deal，两层(spu-sku)，三层(spg-spu-sku)
- 业务系统变更频繁(一年3次变更的频率)
- 业务系统间差异化明显

于是我们在 ODS 与多维明细层中间加入了数据整合层，参照 Bill Inmon 所提出的企业信息工厂建设的模式，基本按照三范式的原则来进行数据整合，由业务驱动调整成了由技术驱动的方式来建设数据仓库基础层。下图是该层次的一些描述：



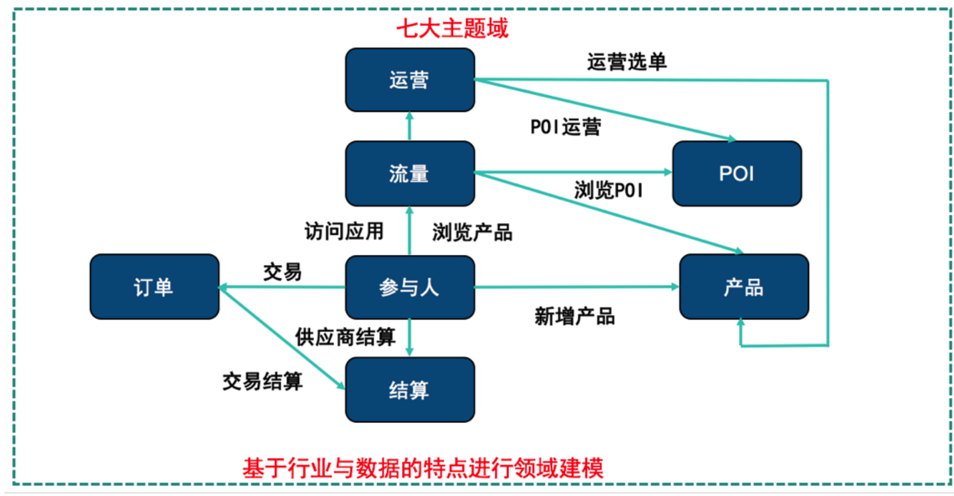
建设方法论：

- 信息探索
- 数据发现、验证
- 统一表、字段的命名
- 代理键统一编码
- 数据标准化
- 统一ETL规则
- 概念模型(CDM)
- 逻辑模型(LDM)
- 物理模型(PDM)

使用本基础层的最根本出发点还是在于我们的供应链、业务、数据它们本身的多样性，如果业务、数据相对比较单一、简单，本层次的架构方案很可能将不再适用。

业务架构

下面介绍我们的主题建设，实际上在传统的一些如银行、制造业、电信、零售等行业里，都有一些比较成熟的模型，如耳熟能详的 BDWM、FS-LDM、MLDM 等等模型，它们都是经过一些具有相类似行业的企业在二三十年数据仓库建设中所积累的行业经验，不断的优化并通用化。但我们所处的 O2O 行业本身就没有可借鉴的成熟的数据仓库主题以及模型，所以，我们在摸索建设两年的时间里，我们目前总结了下面比较适合我们现状的七大主题（后续可能还会新增）：



参与人主题

用户子主题：使用我们服务的所有人都是我们的用户，这是我们数据中至关重要的实体，也是我们数仓中非常重要的一个主题，对用户数据的系统化建设能够很好的帮助我们企业快速的发展，不断提高用户的体验、扩大我们的用户群。

BD 子主题：通过 BD 的业务扩展，建立我们与商户之间的关系，让用户通过我们的服务访问到商户所发布的信息，对 BD 数据的建设，能够让我们的商户覆盖更加迅速、让我们和商户之间的关系更加紧密。

供应商子主题：供应商无论作为直签还是作为三方签约对象，对我们的业务发展都非常重要，通过对其数据的建设，可以让我们彼此双赢，通过我们的平台让双方的业务迅速发展。

流量主题

用户通过 App 或 PC 或 I 版、微信等等形式访问我们的服务，形成了对我们企业至关重要的流量，本主题也是比较具有互联网特色的主题，对于流量的数据建设能够让我们不断优化我们的产品、服务，给我们带来更多的流量、更快的扩张。

订单主题

当用户给我们带来流量的同时，他们也会产生交易，订单主题的独立建设以及其重要性我这里就不再赘述了，在所有的互联网以及传统公司里，该主题都是至关重要的。

POI 主题

这个主题也具有我们自身的 O2O 特色，实际上这个主题与阿里的商家主题比较类似但又具备自己的特点，对于 POI 自身的重要性就不再过多介绍，通过对 POI 的数据集中建设能够让我们给 POI 带去更好的服务与回报。

产品主题

与 POI 强相关的就是产品了，如何让产品能够更加的贴近用户的需求以及产生更多的交易、流量，产品数据主题的建设及目的的意义就在于此。

运营主题

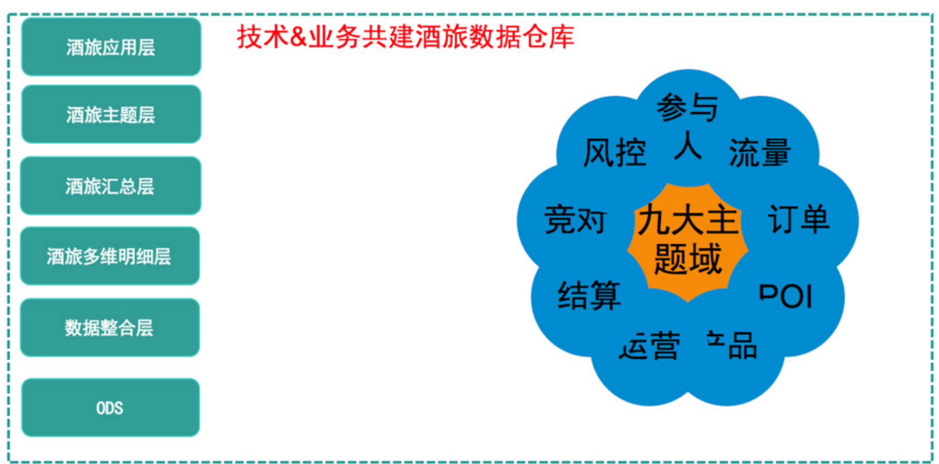
我们的业务发展将不再依靠粗暴的补贴式的扩张发展模式，需要依赖现在的精细化运营方式，运营数据主题的建设就有了非常强的必要性，通过数据进行精细化运营已经成为我们运营的主要发展趋势。

结算主题

实际上，这个主题在传统企业里面如银行、电信等等都是至关重要的，对我们酒旅而言，建设它的意义能够不断优化商家体验、提高财务结算与管理能力。

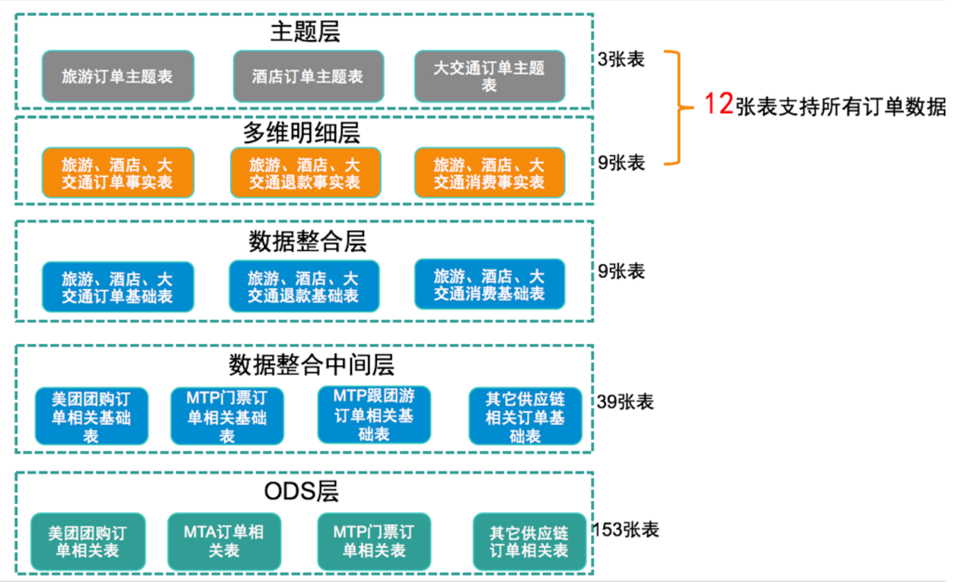
整体架构

我们的七个主题基本上都采用 6 层结构的方式来建设，划分主题更多是从业务的角度出发，而层次划分则是基于技术，实质上我们就是基于业务与技术的结合完成了整体的数据仓库架构。下面介绍一下具体的一些主题案例：



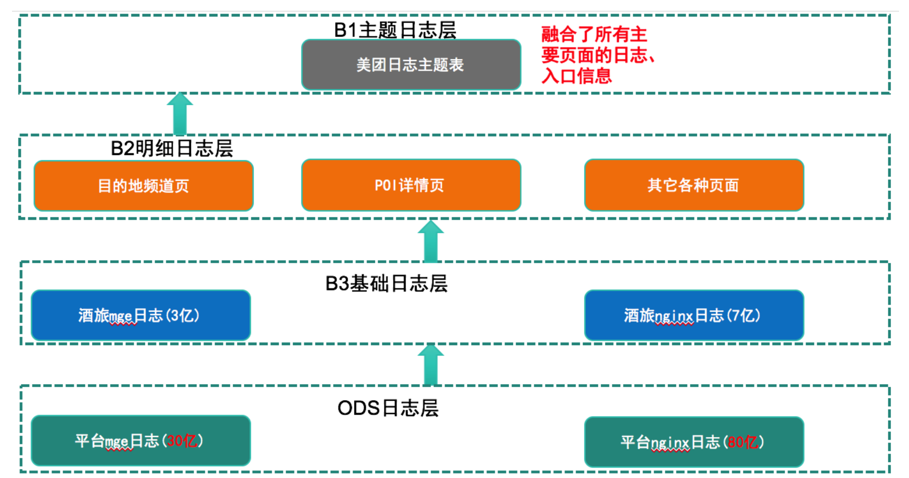
订单主题

在订单主题的建设过程中，我们是按照由分到总的结构思路来进行建设，首先分供应链建设订单相关实体（数据整合中间层 3NF），然后再进行适度抽象把分供应链的相关订单实体进行合并后生成订单实体（数据整合层 3NF），后续在数据整合层的订单实体基础上再扩展部分维度信息来完成后续层次的建设。



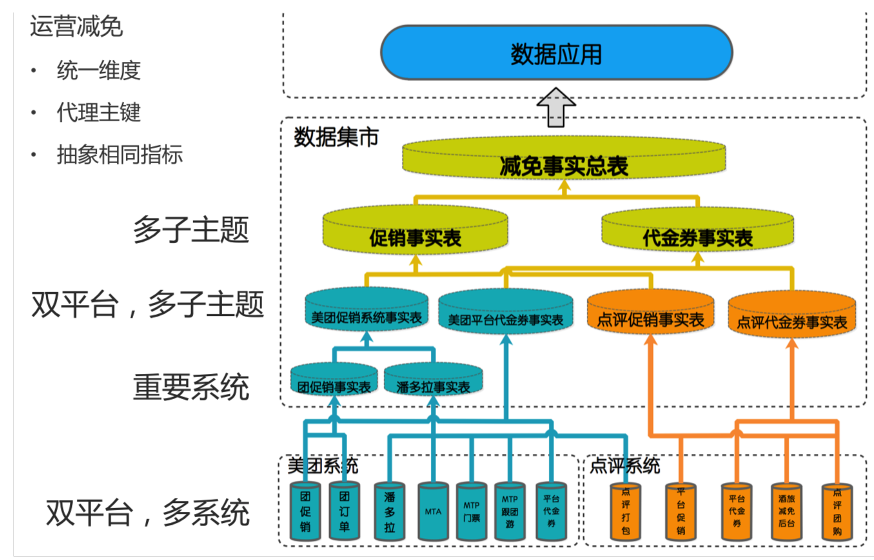
流量主题

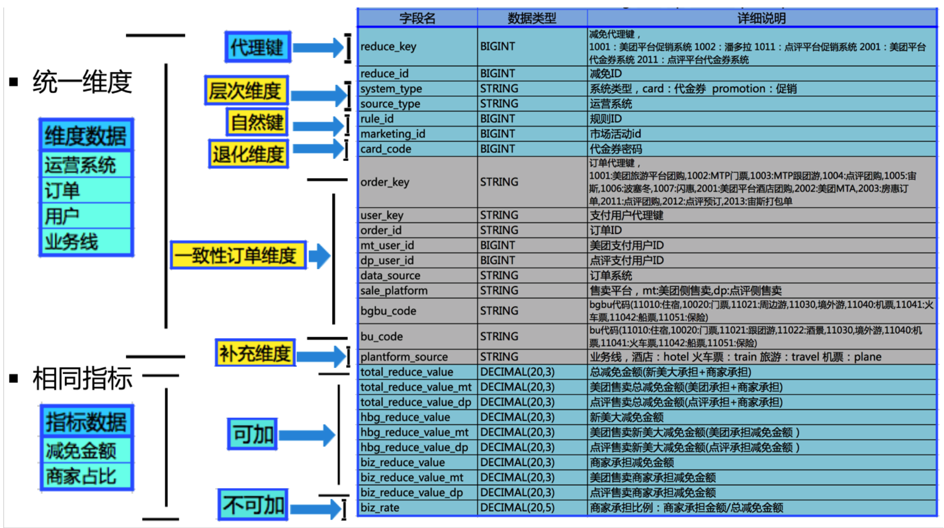
流量主题与订单主题的区别是非常大的，它的数据来源具有一定的特殊性，我们的总体建设思路是总－分－总的思路，首先从总的日志数据中剥离出来属于酒旅事业群的数据，后续再从这些数据中分拆到各个具体的页面（可以适当补充些各个页面中所具有的 B 端信息，如 POI 详情页中增加 POI 品类信息），最后再把各个页面进行合并生成总的日志主题表（最终这张表会满足 80% 以上的相关流量统计需求）。



运营主题

运营主题与订单、流量主题相比也具有自身的特殊性，主要原因也在于其数据来源本身的特殊性，关于它的建设思路总体也是总 – 分 – 总，但我们本身的数据来源大多已经不是最底层的 ODS 数据，而是一些已经加工过的事实表或维度表，所以我们整体的建模原则基本上都是维度建模。





字段名	数据类型	详细说明
reduce_key	BIGINT	减免代理键，1001：美团平台促销系统 1002：潘多拉 1011：点评平台促销系统 2001：美团平台代金券系统 2011：点评平台代金券系统
reduce_id	BIGINT	减免ID
system_type	STRING	系统类型，card：代金券 promotion：促销
source_type	STRING	运营系统
rule_id	BIGINT	规则ID
marketing_id	BIGINT	市场活动id
card_code	BIGINT	代金券密码
order_key	STRING	订单代理键，1001:美团旅游平台团购,1002:MTP门票,1003:MTP跟团游,1004:点评团购,1005:宙斯,1006:波塞冬,1007:闪惠,2001:美团平台酒店团购,2002:美团MTA,2003:房惠订单,2011:点评团购,2012:点评预订,2013:宙斯打包单
user_key	STRING	支付用户代理键
order_id	STRING	订单ID
mt_user_id	BIGINT	美团支付用户ID
dp_user_id	BIGINT	点评支付用户ID
data_source	STRING	订单系统
sale_platform	STRING	售卖平台，mt:美团侧售卖,dp:点评侧售卖
bgbu_code	STRING	bgbu代码(11010:住宿,10020:门票,11021:周边游,11030,境外游,11040:机票,11041:火车票,11042:船票,11051:保险)
bu_code	STRING	bu代码(11010:住宿,10020:门票,11021:跟团游,11022:酒景,11030,境外游,11040:机票,11041:火车票,11042:船票,11051:保险)
plantform_source	STRING	业务线，酒店：hotel 火车票：train 旅游：travel 机票：plane
total_reduce_value	DECIMAL(20,3)	总减免金额(新美大承担+商家承担)
total_reduce_value_mt	DECIMAL(20,3)	美团售卖总减免金额(美团承担+商家承担)
total_reduce_value_dp	DECIMAL(20,3)	点评售卖总减免金额(点评承担+商家承担)
hbg_reduce_value	DECIMAL(20,3)	新美大减免金额
hbg_reduce_value_mt	DECIMAL(20,3)	美团售卖新美大减免金额(美团承担减免金额）
hbg_reduce_value_dp	DECIMAL(20,3)	点评售卖新美大减免金额(点评承担减免金额）
biz_reduce_value	DECIMAL(20,3)	商家承担减免金额
biz_reduce_value_mt	DECIMAL(20,3)	美团售卖商家承担减免金额
biz_reduce_value_dp	DECIMAL(20,3)	点评售卖商家承担减免金额
biz_rate	DECIMAL(20,5)	商家承担比例：商家承担金额/总减免金额

基于上面介绍的几个主题，我们实际上在做分主题的层次架构时也是基于本主题的业务、数据特点作为最终的判断条件，没有绝对的一种层次架构适用于所有的主题，需要综合各项要素来进行综合判断才能设计比较合适的层次架构。

作者简介

德臣，美团点评酒旅事业群数据仓库专家，2003 年毕业于湖南大学，2015 年加入美团，整体负责酒旅事业群的离线数据仓库、实时数据仓库建设。

酒旅数据仓库团队，结合酒旅业务的发展，灵活利用大数据生态链的相关技术，致力于离线数据仓库与实时数据仓库的建设，为业务提供多样化的数据服务。

流计算框架 Flink 与 Storm 的性能对比

梦瑶

1. 背景

Apache Flink 和 Apache Storm 是当前业界广泛使用的两个分布式实时计算框架。其中 Apache Storm（以下简称“Storm”）在美团点评实时计算业务中已有较为成熟的运用（可参考 Storm 的可靠性保证测试），有管理平台、常用 API 和相应的文档，大量实时作业基于 Storm 构建。而 Apache Flink（以下简称“Flink”）在近期倍受关注，具有高吞吐、低延迟、高可靠和精确计算等特性，对事件窗口有很好的支持，目前在美团点评实时计算业务中也已有一定应用。

为深入熟悉了解 Flink 框架，验证其稳定性和可靠性，评估其实时处理性能，识别该体系中的缺点，找到其性能瓶颈并进行优化，给用户提供最适合的实时计算引擎，我们以实践经验丰富的 Storm 框架作为对照，进行了一系列实验测试 Flink 框架的性能，计算 Flink 作为确保“至少一次”和“恰好一次”语义的实时计算框架时对资源的消耗，为实时计算平台资源规划、框架选择、性能调优等决策及 Flink 平台的建设提出建议并提供数据支持，为后续的 SLA 建设提供一定参考。

Flink 与 Storm 两个框架对比：

	Storm	Flink
状态管理	无状态，需用户自行进行状态管理	有状态
窗口支持	对事件窗口支持较弱，缓存整个窗口的所有数据，窗口结束时一起计算	窗口支持较为完善，自带一些窗口聚合方法，并且会自动管理窗口状态。
消息投递	At Most Once At Least Once	At Most Once At Least Once **Exactly Once**
容错方式	ACK 机制：对每个消息进行全链路跟踪，失败或超时进行重发。	检查点机制：通过分布式一致性快照机制，对数据流和算子状态进行保存。在发生错误时，使系统能够进行回滚。
应用现状	在美团点评实时计算业务中已有较为成熟的运用，有管理平台、常用 API 和相应的文档，大量实时作业基于 Storm 构建。	在美团点评实时计算业务中已有一定应用，但是管理平台、API 及文档等仍需进一步完善。

2. 测试目标

评估不同场景、不同数据压力下 Flink 和 Storm 两个实时计算框架目前的性能表现，获取其详细性能数据并找到处理性能的极限；了解不同配置对 Flink 性能影响的程度，分析各种配置的适用场景，从而得出调优建议。

2.1 测试场景

“输入 – 输出”简单处理场景

通过对“输入 – 输出”这样简单处理逻辑场景的测试，尽可能减少其它因素的干扰，反映两个框架本身的性能。

同时测算框架处理能力的极限，处理更加复杂的逻辑的性能不会比纯粹“输入 – 输出”更高。

用户作业耗时较长的场景

如果用户的处理逻辑较为复杂，或是访问了数据库等外部组件，其执行时间会增大，作业的性能会受到影响。因此，我们测试了用户作业耗时较长的场景下两个框架的调度性能。

窗口统计场景

实时计算中常有对时间窗口或计数窗口进行统计的需求，例如一天中每五分钟的访问量，每 100 个订单中有多少个使用了优惠等。Flink 在窗口支持上的功能比 Storm 更加强大，API 更加完善，但是我们同时也想了解在窗口统计这个常用场景下两个框架的性能。

精确计算场景（即消息投递语义为“恰好一次”）

Storm 仅能保证“至多一次”(At Most Once) 和“至少一次”(At Least Once) 的消息投递语义，即可能存在重复发送的情况。有很多业务场景对数据的精确性要求较高，希望消息投递不重不漏。Flink 支持“恰好一次”(Exactly Once) 的语义，但是在限定的资源条件下，更加严格的精确度要求可能带来更高的代价，从而影响性能。因此，我们测试了在不同消息投递语义下两个框架的性能，希望为精确计算场景

的资源规划提供数据参考。

2.2 性能指标

吞吐量（Throughput）

- 单位时间内由计算框架成功地传送数据的数量，本次测试吞吐量的单位为：条 / 秒。
- 反映了系统的负载能力，在相应的资源条件下，单位时间内系统能处理多少数据。
- 吞吐量常用于资源规划，同时也用于协助分析系统性能瓶颈，从而进行相应的资源调整以保证系统能达到用户所要求的处理能力。假设商家每小时能做二十份午餐（吞吐量 20 份 / 小时），一个外卖小哥每小时只能送两份（吞吐量 2 份 / 小时），这个系统的瓶颈就在小哥配送这个环节，可以给该商家安排十个外卖小哥配送。

延迟（Latency）

- 数据从进入系统到流出系统所用的时间，本次测试延迟的单位为：毫秒。
- 反映了系统处理的实时性。
- 金融交易分析等大量实时计算业务对延迟有较高要求，延迟越低，数据实时性越强。
- 假设商家做一份午餐需要 5 分钟，小哥配送需要 25 分钟，这个流程中用户感受到了 30 分钟的延迟。如果更换配送方案后延迟变成了 60 分钟，等送到了饭菜都凉了，这个新的方案就是无法接受的。

3. 测试环境

为 Storm 和 Flink 分别搭建由 1 台主节点和 2 台从节点构成的 Standalone 集群进行本次测试。其中为了观察 Flink 在实际生产环境中的性能，对于部分测内容也进行了 on Yarn 环境的测试。

3.1 集群参数

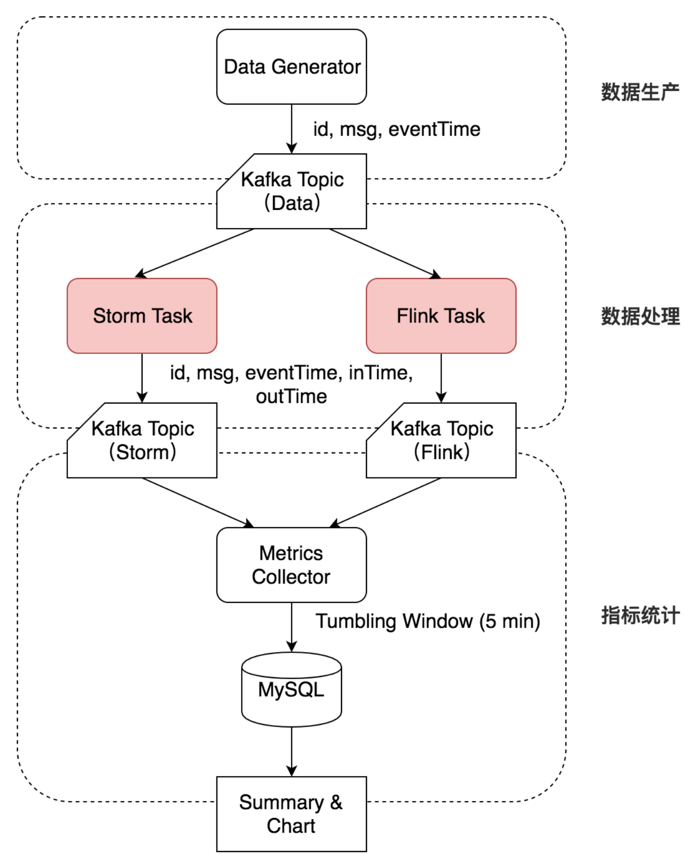
参数项	参数值
CPU	QEMU Virtual CPU version 1.1.2 2.6GHz
Core	8
Memory	16GB
Disk	500G
OS	CentOS release 6.5 (Final)

3.2 框架参数

参数项	Storm 配置	Flink 配置
Version	Storm 1.1.0-mt002	Flink 1.3.0
Master Memory	2600M	2600M
Slave Memory	1600M * 16	12800M * 2
Parallelism	2 supervisor	2 Task Manager
	16 worker	16 Task slots

4. 测试方法

4.1 测试流程



数据生产

Data Generator 按特定速率生成数据，带上自增的 id 和 eventTime 时间戳写入 Kafka 的一个 Topic（Topic Data）。

数据处理

Storm Task 和 Flink Task（每个测试用例不同）从 Kafka Topic Data 相同的 Offset 开始消费，并将结果及相应 inTime、outTime 时间戳分别写入两个 Topic（Topic Storm 和 Topic Flink）中。

指标统计

Metrics Collector 按 outTime 的时间窗口从这两个 Topic 中统计测试指标，每五分钟将相应的指标写入 MySQL 表中。

Metrics Collector 按 outTime 取五分钟的滚动时间窗口，计算五分钟的平均吞吐（输出数据的条数）、五分钟内的延迟（outTime – eventTime 或 outTime – inTime）的中位数及 99 线等指标，写入 MySQL 相应的数据表中。最后对 MySQL 表中的吞吐计算均值，延迟中位数及延迟 99 线选取中位数，绘制图像并分析。

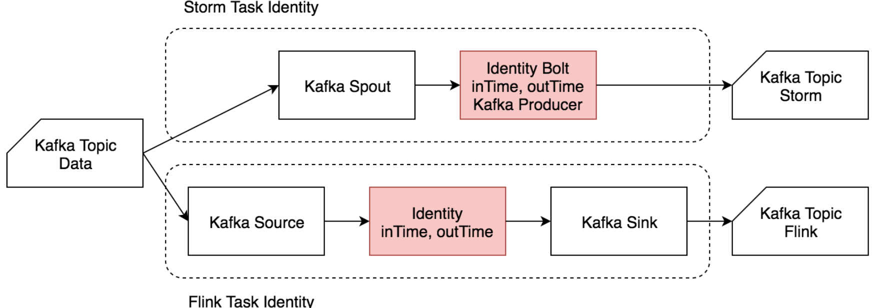
4.2 默认参数

- Storm 和 Flink 默认均为 At Least Once 语义。
  - Storm 开启 ACK，ACKer 数量为 1。
  - Flink 的 Checkpoint 时间间隔为 30 秒，默认 StateBackend 为 Memory。
- 保证 Kafka 不是性能瓶颈，尽可能排除 Kafka 对测试结果的影响。
- 测试延迟时数据生产速率小于数据处理能力，假设数据被写入 Kafka 后立刻被读取，即 eventTime 等于数据进入系统的时间。
- 测试吞吐量时从 Kafka Topic 的最旧开始读取，假设该 Topic 中的测试数据量充足。

4.3 测试用例

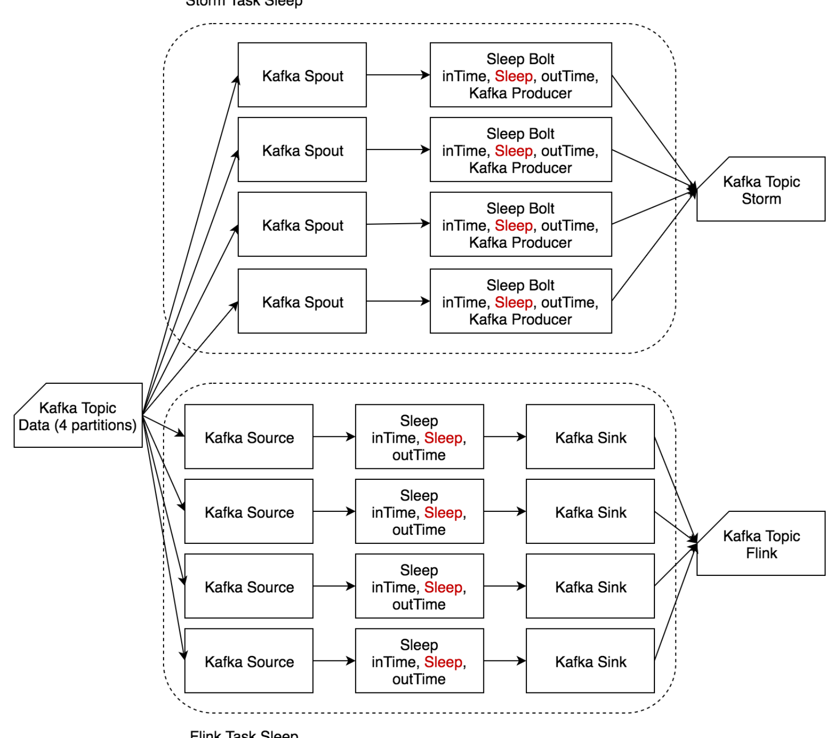
Identity

- Identity 用例主要模拟“输入 – 输出”简单处理场景，反映**两个框架本身的性能**。
- 输入数据为“msgId, eventTime”，其中 eventTime 视为数据生成时间。单条输入数据约 20 B。
- 进入作业处理流程时记录 inTime，作业处理完成后（准备输出时）记录 outTime。
- 作业从 Kafka Topic Data 中读取数据后，在字符串末尾追加时间戳，然后直接输出到 Kafka。
- 输出数据为“msgId, eventTime, inTime, outTime”。单条输出数据约 50 B。



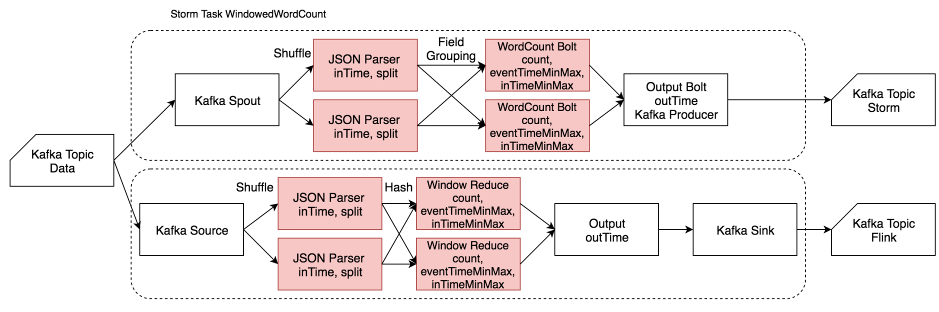
Sleep

- Sleep 用例主要模拟用户作业耗时较长的场景，反映**复杂用户逻辑对框架差异的削弱**，比较两个**框架的调度性能**。
- 输入数据和输出数据均与 Identity 相同。
- 读入数据后，等待一定时长（1 ms）后在字符串末尾追加时间戳后输出



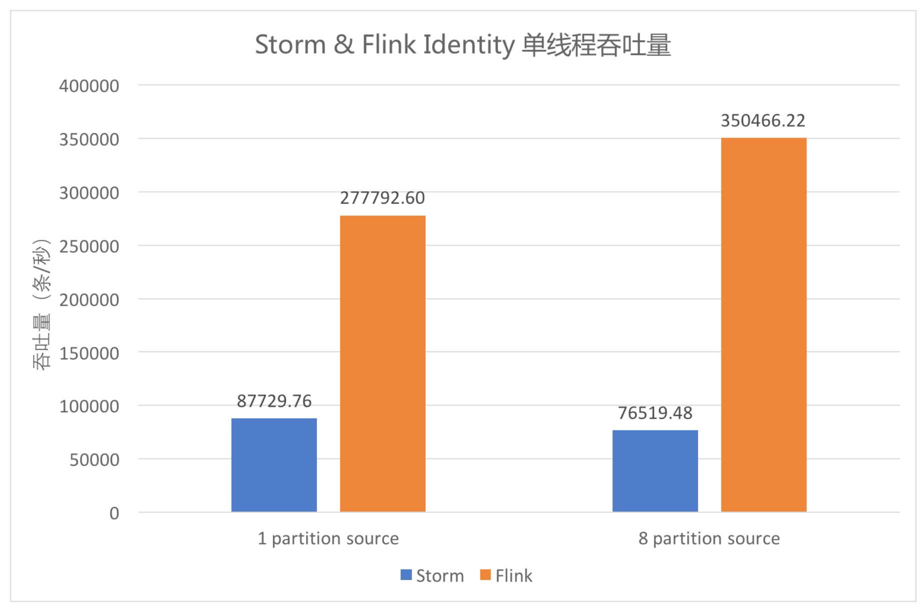
Windowed Word Count

- Windowed Word Count 用例主要模拟窗口统计场景，反映两个**框架在进行窗口统计时性能的差异**。
- 此外，还用其进行了精确计算场景的测试，反映 Flink **恰好一次投递的性能**。
- 输入为 JSON 格式，包含 msgId、eventTime 和一个由若干单词组成的句子，单词之间由空格分隔。单条输入数据约 150 B。
- 读入数据后解析 JSON，然后将句子分割为相应单词，带 eventTime 和 inTime 时间戳发给 CountWindow 进行单词计数，同时记录一个窗口中最大最小的 eventTime 和 inTime，最后带 outTime 时间戳输出到 Kafka 相应的 Topic。
- Spout/Source 及 OutputBolt/Output/Sink 并发度恒为 1，增大并发度时仅增大 JSONParser、CountWindow 的并发度。
- 由于 Storm 对 window 的支持较弱，CountWindow 使用一个 HashMap 手动实现，Flink 用了原生的 CountWindow 和相应的 Reduce 函数。



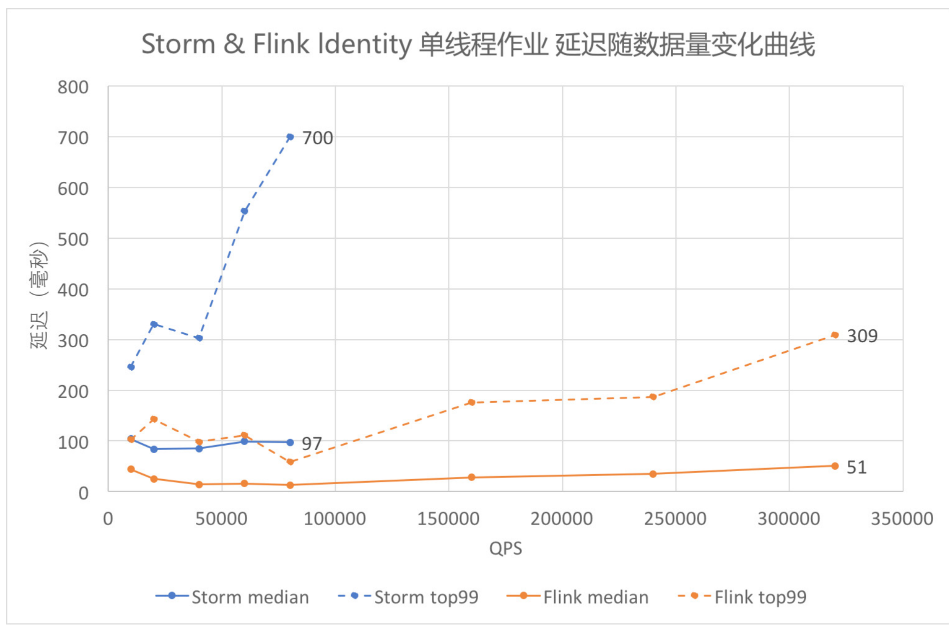
5. 测试结果

5.1 Identity 单线程吞吐量



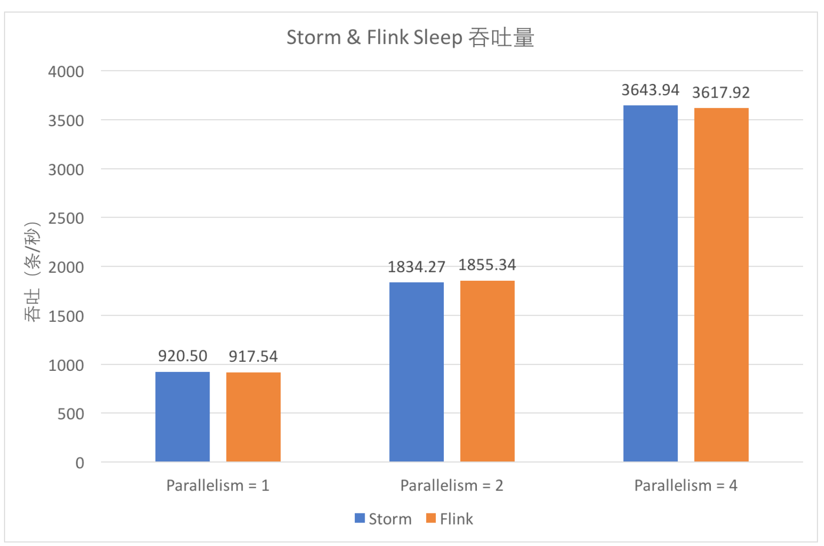
- 上图中蓝色柱形为单线程 Storm 作业的吞吐，橙色柱形为单线程 Flink 作业的吞吐。
- Identity 逻辑下，Storm 单线程吞吐为 **8.7** 万条 / 秒，Flink 单线程吞吐可达 **35** 万条 / 秒。
- 当 Kafka Data 的 Partition 数为 1 时，Flink 的吞吐约为 Storm 的 3.2 倍；当其 Partition 数为 8 时，Flink 的吞吐约为 Storm 的 4.6 倍。
- 由此可以看出，**Flink 吞吐约为 Storm 的 3-5 倍**。

5.2 Identity 单线程作业延迟



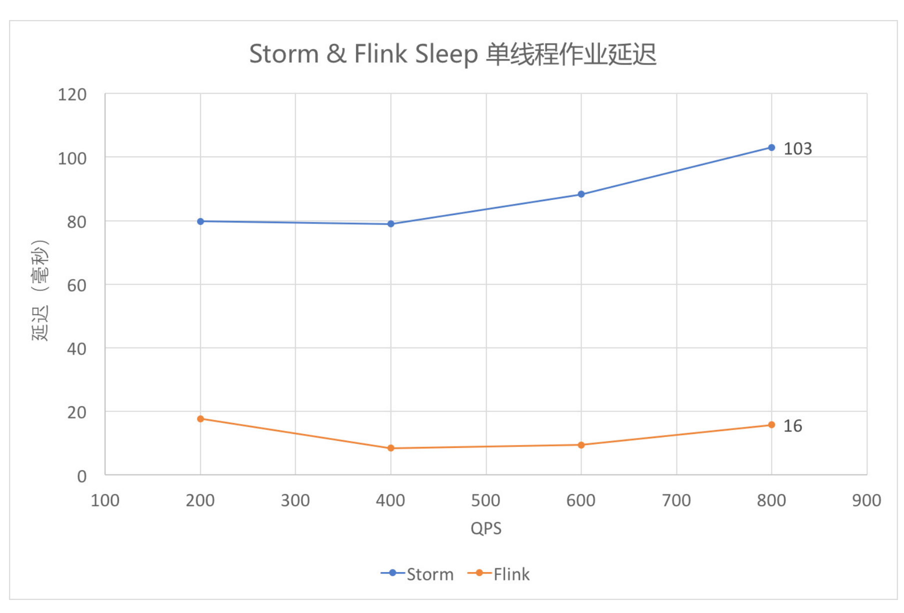
- 采用 outTime – eventTime 作为延迟，图中蓝色折线为 Storm，橙色折线为 Flink。虚线为 99 线，实线为中位数。
- 从图中可以看出随着数据量逐渐增大，Identity 的延迟逐渐增大。其中 99 线的增大速度比中位数快，Storm 的 增大速度比 Flink 快。
- 其中 QPS 在 80000 以上的测试数据超过了 Storm 单线程的吞吐能力，无法对 Storm 进行测试，只有 Flink 的曲线。
- 对比折线最右端的数据可以看出，Storm QPS 接近吞吐时延迟中位数约 100 毫秒，99 线约 700 毫秒，Flink 中位数约 50 毫秒，99 线约 300 毫秒。**Flink 在满吞吐时的延迟约为 Storm 的一半**。

5.3 Sleep 吞吐量



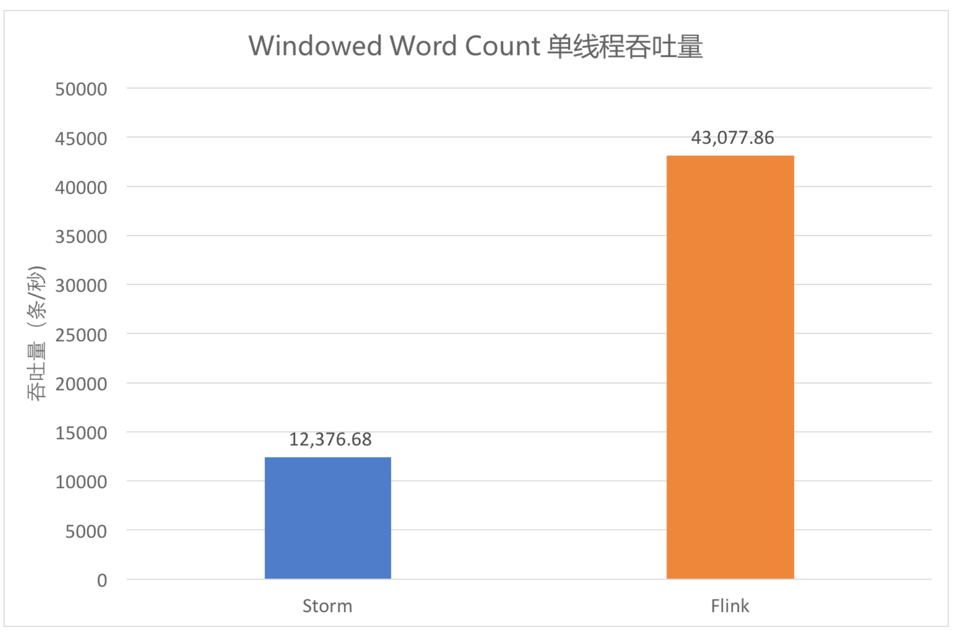
- 从图中可以看出，Sleep 1 毫秒时，Storm 和 Flink 单线程的吞吐均在 900 条 / 秒左右，且随着并发增大基本呈线性增大。
- 对比蓝色和橙色的柱形可以发现，**此时两个框架的吞吐能力基本一致**。

5.4 Sleep 单线程作业延迟（中位数）



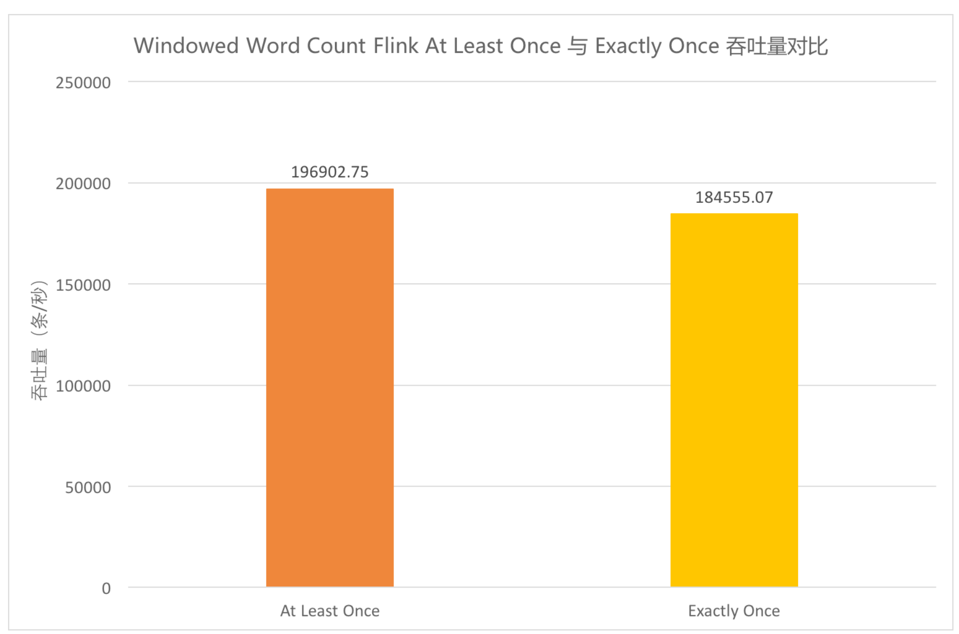
- 依然采用 outTime – eventTime 作为延迟，从图中可以看出，Sleep 1 毫秒时，Flink 的延迟仍低于 Storm。

5.5 Windowed Word Count 单线程吞吐量



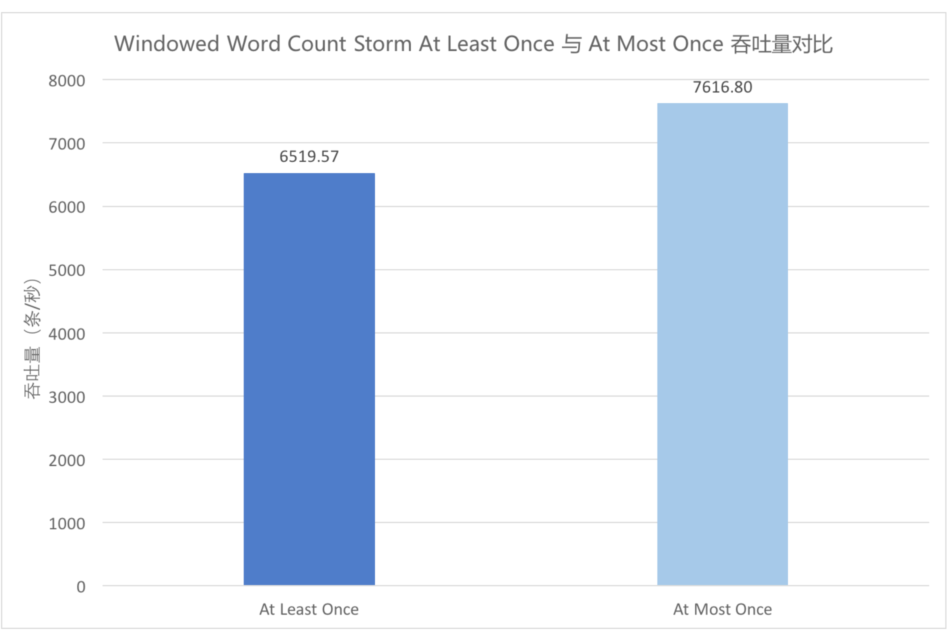
- 单线程执行大小为 10 的计数窗口，吞吐量统计如图。
- 从图中可以看出，Storm 吞吐约为 1.2 万条 / 秒，Flink Standalone 约为 4.3 万条 / 秒。**Flink 吞吐依然为 Storm 的 3 倍以上**。

5.6 Windowed Word Count Flink At Least Once 与 Exactly Once 吞吐量对比



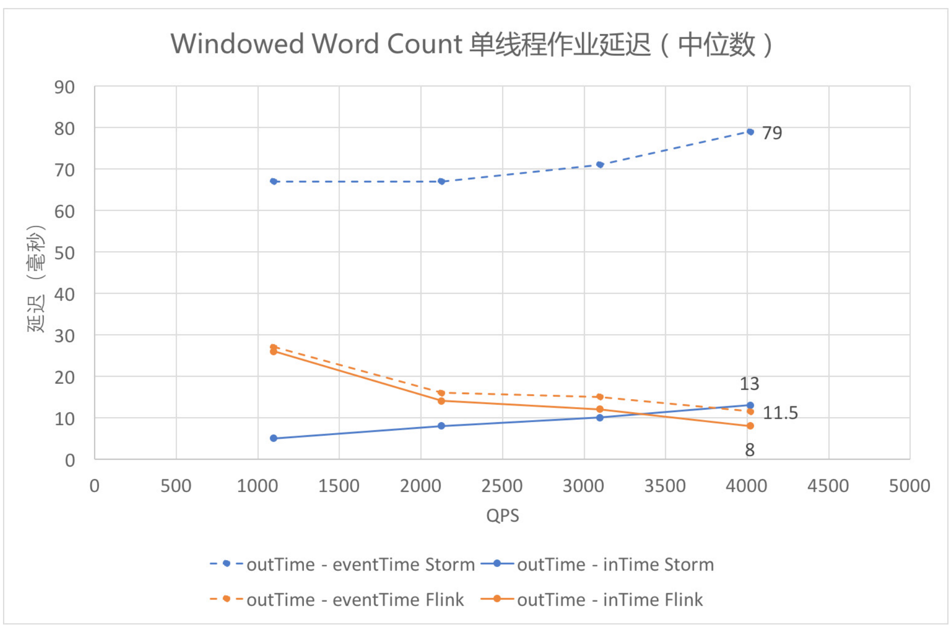
- 由于同一算子的多个并行任务处理速度可能不同，在上游算子中不同快照里的内容，经过中间并行算子的处理，到达下游算子时可能被计入同一个快照中。这样一来，这部分数据会被重复处理。因此，Flink 在 Exactly Once 语义下需要进行对齐，即当前最早的快照中所有数据处理完之前，属于下一个快照的数据不进行处理，而是在缓存区等待。当前测试用例中，在 JSON Parser 和 CountWindow、CountWindow 和 Output 之间均需要进行对齐，有一定消耗。为体现出对齐场景，Source/Output/Sink 并发度的并发度仍为 1，提高了 JSONParser/CountWindow 的并发度。具体流程细节参见前文 Windowed Word Count 流程图。
- 上图中橙色柱形为 At Least Once 的吞吐量，黄色柱形为 Exactly Once 的吞吐量。对比两者可以看出，在当前并发条件下，**Exactly Once 的吞吐较 At Least Once 而言下降了 6.3%**。

5.7 Windowed Word Count Storm At Least Once 与 At Most Once 吞吐量对比



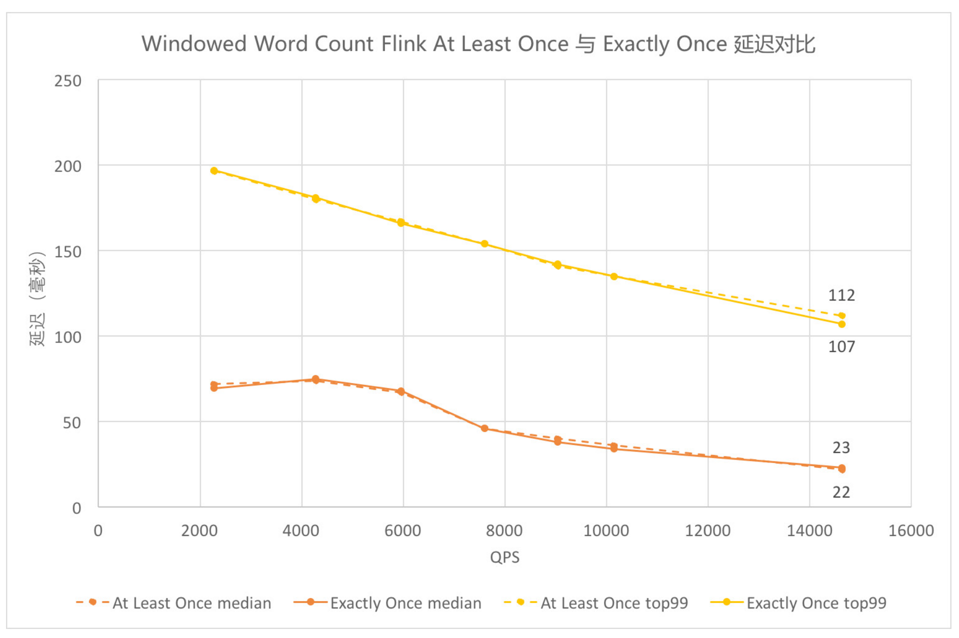
- Storm 将 ACKer 数量设置为零后，每条消息在发送时就自动 ACK，不再等待 Bolt 的 ACK，也不再重发消息，为 At Most Once 语义。
- 上图中蓝色柱形为 At Least Once 的吞吐量，浅蓝色柱形为 At Most Once 的吞吐量。对比两者可以看出，在当前并发条件下，**At Most Once 语义下的吞吐较 At Least Once 而言提高了 16.8%**。

5.8 Windowed Word Count 单线程作业延迟



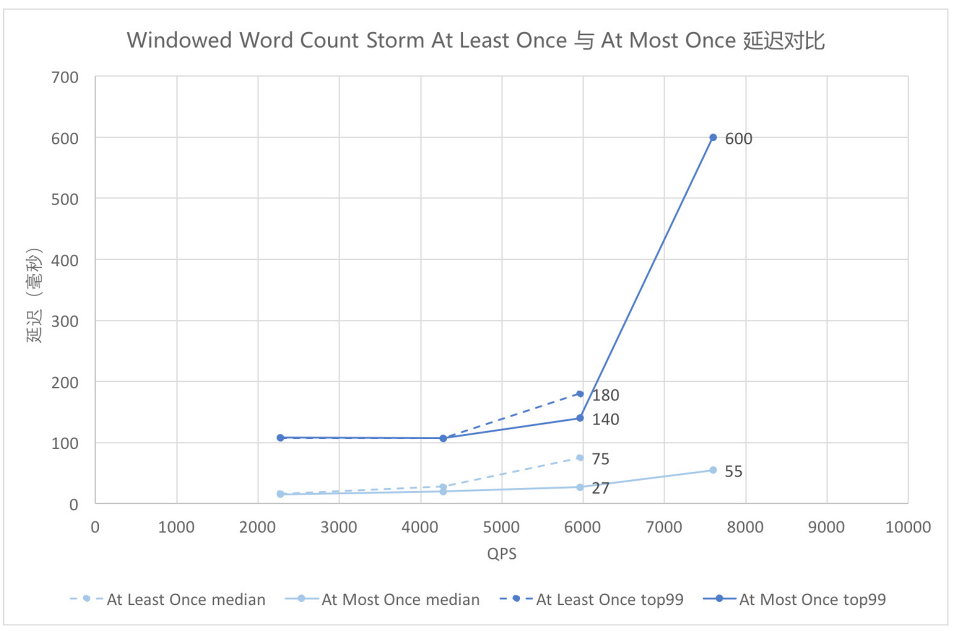
- Identity 和 Sleep 观测的都是 outTime – eventTime，因为作业处理时间较短或 Thread.sleep() 精度不高，outTime – inTime 为零或没有比较意义；Windowed Word Count 中可以有效测得 outTime – inTime 的数值，将其与 outTime – eventTime 画在同一张图上，其中 outTime – eventTime 为虚线，outTime – InTime 为实线。
- 观察橙色的两条折线可以发现，Flink 用两种方式统计的延迟都维持在较低水平；观察两条蓝色的曲线可以发现，Storm 的 outTime – inTime 较低，outTime – eventTime 一直较高，即 inTime 和 eventTime 之间的差值一直较大，可能与 Storm 和 Flink 的数据读入方式有关。
- 蓝色折线表明 Storm 的延迟随数据量的增大而增大，而橙色折线表明 Flink 的延迟随着数据量的增大而减小（此处未测至 Flink 吞吐量，接近吞吐时 Flink 延迟依然会上升）。
- 即使仅关注 outTime – inTime（即图中实线部分），依然可以发现，**当 QPS 逐渐增大的时候，Flink 在延迟上的优势开始体现出来**。

5.9 Windowed Word Count Flink At Least Once 与 Exactly Once 延迟对比



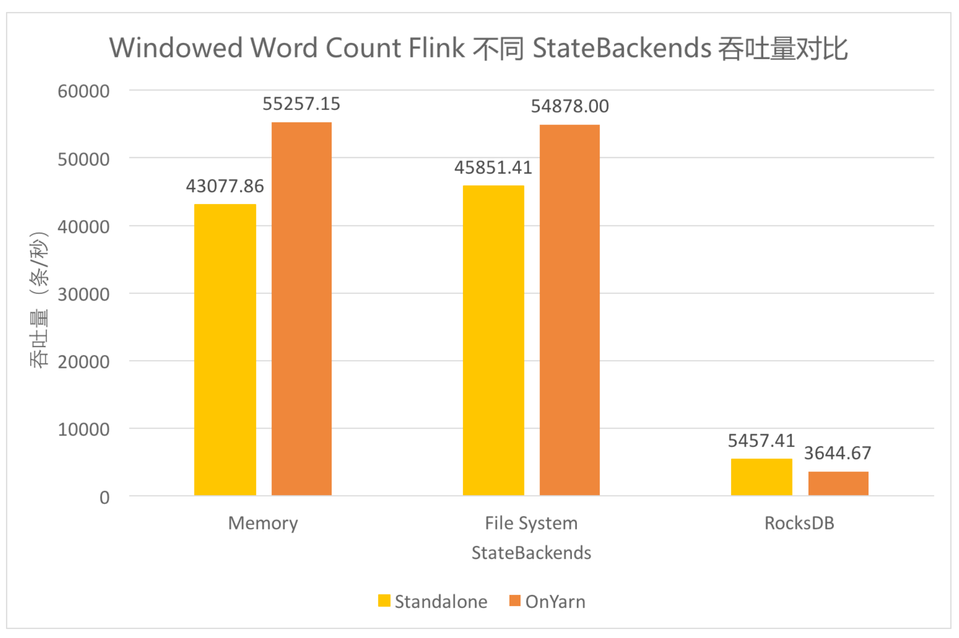
- 图中黄色为 99 线，橙色为中位数，虚线为 At Least Once，实线为 Exactly Once。图中相应颜色的虚实曲线都基本重合，可以看出 **Flink Exactly Once 的延迟中位数曲线与 At Least Once 基本贴合，在延迟上性能没有太大差异。**

5.10 Windowed Word Count Storm At Least Once 与 At Most Once 延迟对比



- 图中蓝色为 99 线，浅蓝色为中位数，虚线为 At Least Once，实线为 At Most Once。QPS 在 4000 及以前的时候，虚线实线基本重合；QPS 在 6000 时两者已有差异，虚线略高；QPS 接近 8000 时，已超过 At Least Once 语义下 Storm 的吞吐，因此只有实线上的点。
- 可以看出，QPS 较低时 Storm At Most Once 与 At Least Once 的延迟观察不到差异，**随着 QPS 增大差异开始增大，At Most Once 的延迟较低**。

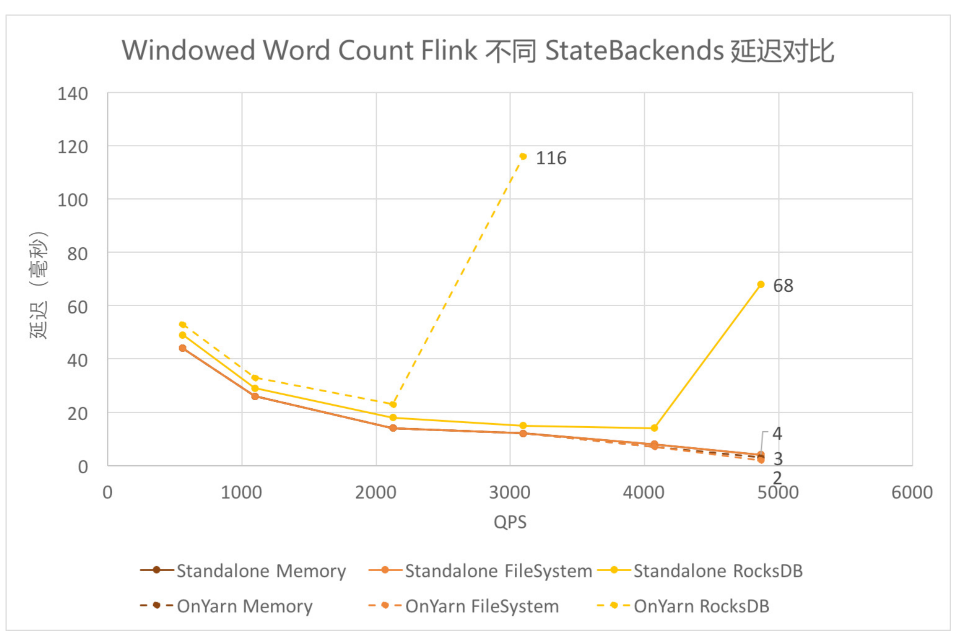
5.11 Windowed Word Count Flink 不同 StateBackends 吞吐量对比



- Flink 支持 Standalone 和 on Yarn 的集群部署模式，同时支持 Memory、FileSystem、RocksDB 三种状态存储后端（StateBackends）。由于线上作业需要，测试了这三种 StateBackends 在两种集群部署模式上的性能差异。其中，Standalone 时的存储路径为 JobManager 上的一个文件目录，on Yarn 时存储路径为 HDFS 上一个文件目录。
- 对比三组柱形可以发现，**使用 FileSystem 和 Memory 的吞吐差异不大，使用 RocksDB 的吞吐仅其余两者的十分之一左右**。
- 对比两种颜色可以发现，**Standalone 和 on Yarn 的总体差异不大**，使用 FileSystem 和 Memory 时 on Yarn 模式下吞吐稍高，使用 RocksDB 时 Standalone 模式下的吞吐稍高。

5.12 Windowed Word Count Flink 不同 StateBackends 延迟对比

- 使用 FileSystem 和 Memory 作为 Backends 时，延迟基本一致且较低。
- 使用 RocksDB 作为 Backends 时，延迟稍高，且由于吞吐较低，在达到吞吐瓶颈前的延迟陡增。其中 on Yarn 模式下吞吐更低，接近吞吐时的延迟更高。



6. 结论及建议

6.1 框架本身性能

- 由 5.1、5.5 的测试结果可以看出，Storm 单线程吞吐约为 8.7 万条 / 秒，Flink 单线程吞吐可达 35 万条 / 秒。Flink 吞吐约为 Storm 的 3-5 倍。
- 由 5.2、5.8 的测试结果可以看出，Storm QPS 接近吞吐时延迟（含 Kafka 读写时间）中位数约 100 毫秒，99 线约 700 毫秒，Flink 中位数约 50 毫秒，99 线约 300 毫秒。Flink 在满吞吐时的延迟约为 Storm 的一半，且随着 QPS 逐渐增大，Flink 在延迟上的优势开始体现出来。

- 综上可得，**Flink 框架本身性能优于 Storm**。

6.2 复杂用户逻辑对框架差异的削弱

- 对比 5.1 和 5.3、5.2 和 5.4 的测试结果可以发现，单个 Bolt Sleep 时长达到 1 毫秒时，Flink 的延迟仍低于 Storm，但吞吐优势已基本无法体现。
- 因此，用户逻辑越复杂，本身耗时越长，针对该逻辑的测试体现出来的框架的差异越小。

6.3 不同消息投递语义的差异

- 由 5.6、5.7、5.9、5.10 的测试结果可以看出，Flink Exactly Once 的吞吐较 At Least Once 而言下降 6.3%，延迟差异不大；Storm At Most Once 语义下的吞吐较 At Least Once 提升 16.8%，延迟稍有下降。
- 由于 Storm 会对每条消息进行 ACK，Flink 是基于一批消息做的检查点，不同的实现原理导致两者在 At Least Once 语义的花费差异较大，从而影响了性能。而 Flink 实现 Exactly Once 语义仅增加了对齐操作，因此在**算子并发量不大、没有出现慢节点的情况下对 Flink 性能的影响不大**。Storm At Most Once 语义下的性能仍然低于 Flink。

6.4 Flink 状态存储后端选择

Flink 提供了内存、文件系统、RocksDB 三种 StateBackends，结合 5.11、5.12 的测试结果，三者的对比如下：

StateBackend	过程状态存储	检查点存储	吞吐	推荐使用场景
Memory	TM Memory	JM Memory	高（3-5 倍 Storm）	调试、无状态或对数据是否丢失重复无要求
FileSystem	TM Memory	FS/HDFS	高（3-5 倍 Storm）	普通状态、窗口、KV 结构（建议作为默认 Backend）
RocksDB	RocksDB on TM	FS/HDFS	低（0.3-0.5 倍 Storm）	超大状态、超长窗口、大型 KV 结构

6.5 推荐使用 Flink 的场景

综合上述测试结果，以下实时计算场景建议考虑使用 Flink 框架进行计算：

- 要求消息投递语义为 **Exactly Once** 的场景；
- 数据量较大，要求**高吞吐低延迟**的场景；
- 需要进行**状态管理**或**窗口统计**的场景。

7. 展望

- 本次测试中尚有一些内容没有进行更加深入的测试，有待后续测试补充。例如：
    - Exactly Once 在并发量增大的时候是否吞吐会明显下降?
    - 用户耗时到 1ms 时框架的差异已经不再明显（Thread.sleep() 的精度只能到毫秒），用户耗时在什么范围内 Flink 的优势依然能体现出来?
- 本次测试仅观察了吞吐量和延迟两项指标，对于系统的可靠性、可扩展性等重要的性能指标没有在统计数据层面进行关注，有待后续补充。
- Flink 使用 RocksDBStateBackend 时的吞吐较低，有待进一步探索和优化。
- 关于 Flink 的更高级 API，如 Table API & SQL 及 CEP 等，需要进一步了解和完善。

8. 参考内容

1. 分布式流处理框架——功能对比和性能评估 .
2. intel-hadoop/HiBench: HiBench is a big data benchmark suite.
3. Yahoo 的流计算引擎基准测试 .
4. Extending the Yahoo! Streaming Benchmark.

智能投放系统之场景分析最佳实践

张腾

背景

新美大平台作为业内最大的 O2O 的平台，以短信 /push 作为运营手段触达用户的量级巨大，每日数以千万计。

美团点评线上存在超过千万的 POI，覆盖超过 2000 城市、2.5 万个后台商圈。在海量数据存在的前提下，实时投放的用户在场景的选择上存在一些困难，所以我们提供对场景的颗粒化查询和智能建议，为用户解决三大难题：

- 我要投放的区域在哪，实时和历史的客流量是什么样的?
- 在我希望投放的区域历史和现在都发生过什么活动，效果是什么样的?
- 这个区域是不是适合我投放，系统建议我投放哪里?

如图 1 所示，整个产品致力于解决以上三大问题，能够为运营在活动投放前期，提供有效的参考决策依据。

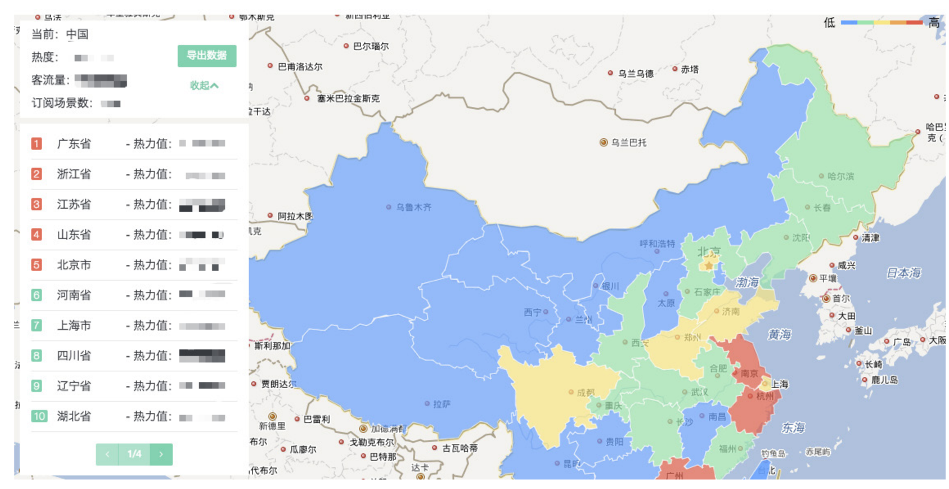


图 1 场景查询器模块效果图

挑战

- 场景查询器需要展示的数据分为多种，所以数据过滤和组装的时间，严重依赖于基础数据量。但是随着维度的下钻，基础数据量巨大，所以导致实时计算数据的响应时间无法忍受。
- 数据来源均是 RPC 服务，需要调用的服务多种多样，每一项服务的响应时间都会影响最终的结果返回，难以提供前端接口的响应时间。
- 需要组装的数据各种各样，没有统一的数据模型，造成代码耦合度高，后期难以维护针对上面的挑战，我们给出如下的解决方案。

总体方案

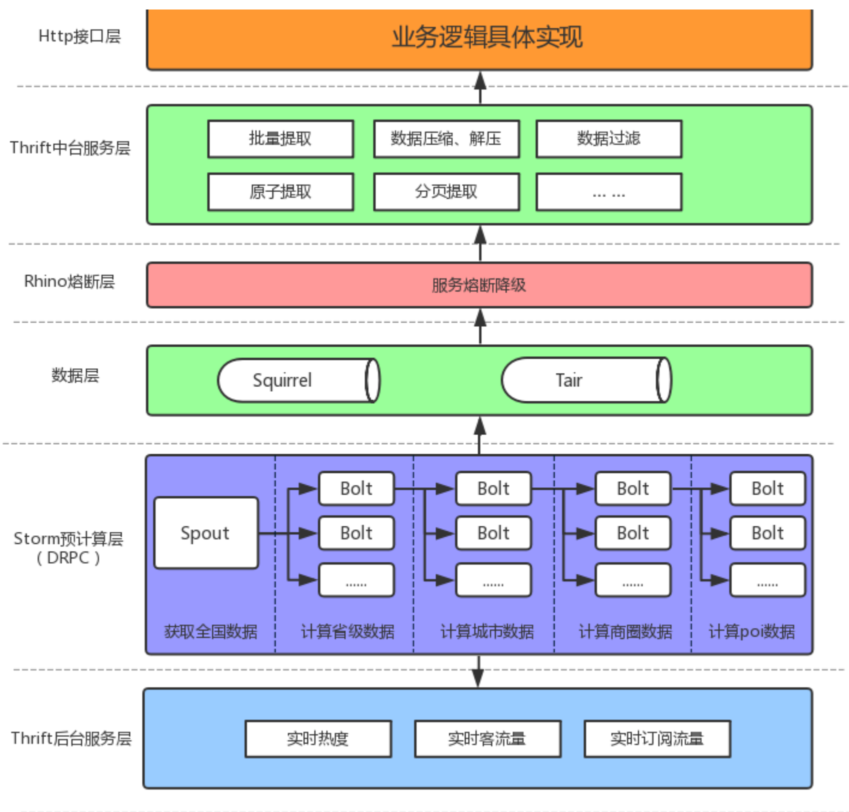


图 2　场景查询器体系架构

如图 2 所示，我们的总体架构是分层设计的，最底层都是各类服务，再上层是预计算层和数据层，预计算层的作用很明显，是连接服务和数据的核心层，通过拉取后台服务的各类数据然后预计算形成数据层。再往上是中台服务层，包含有核心功能是服务熔断降级，以及通用服务，为具体业务逻辑提供统一的服务，最上层便是具体的业务逻辑了，对应具体场景和需求。

后台服务层

该层均是 Thrift 的 RPC 服务，提供各种投放的反馈数据。

数据组装

后台服务层数据特点是数据分散，结果多样。数据组装是对多个服务调用返回的结果，进行过滤组合。

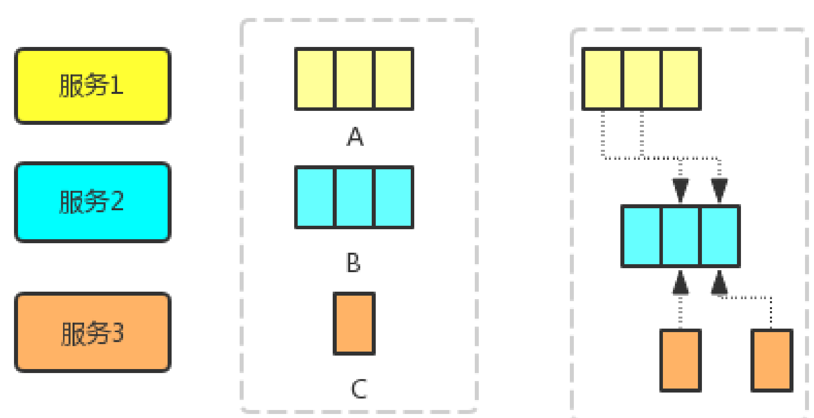


图 3　后台服务结果数据组装样例

如图 3 所示，服务 1、2、3 分别用黄色、蓝色、棕色表示，A 、B、C 均是调用对应服务返回的数据，并且 A、B 的数据格式是列表，C 是单个数据。最后一个虚线框，代表数据组装算法，A 和 B 的列表取交集，结果是长度为 2 的列表，然后再依次调用服务 3，单个获取数据 C。

数据组装痛点

- 过程繁琐，如取交集，单个组装等等，组装时间受数据量影响较大。
- 组装过程中，混合着大量的服务调用，组装时间受服务响应时间影响较大。

后台服务层重点在于提供数据，保证服务的可用性。但是在组装过程中遇到以上痛点，导致出现请求响应时间长，用户体验差等问题。规避此类问题的主要方法是将服务调用数据提前组合计算好进行存储，即数据预计算。

预计算层

主要作用在于提前计算数据，快速响应请求，构建过程依次为数据建模、构建计算模式。该层主要包含以下核心功能。

- 构建通用的数据模型，使上层控制和处理，更加高效。
- 保证计算速度的同时，计算大量基础数据。
- 为了保证数据的实时性，实现高密度并行计算。

数据模型

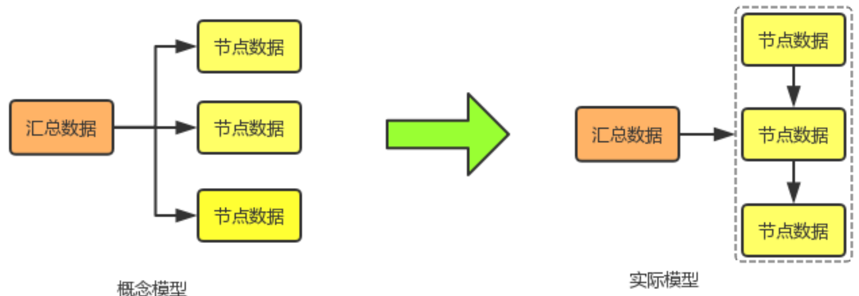


图 4　预计算数据模型

在场景查询器中，为前端提供的数据普遍都是上下级数据，比如页面要展示全国汇总数据，同时会级联展示下属各省数据，如果展示省级汇总数据，那么同时级联展示下属各地级市数据。通过分析业务需求，发现需要的数据，大多是分上下级的这种级联数据。经过抽象，数据模型设计为树形结构，如图 4，左侧为概念模型，树的高

度只有两层，根节点为汇总数据，叶子节点为地理等级维度下钻的数据；右侧为实际使用的模型，因为底层维度的基数比较大，不利于下级数据的遍历、筛选和分页，所以实际使用中，下级节点数据以一个列表存储。节点可以存储若干指标，具体类型根据地理维度而定。该模型的特点如下：首先支持地理维度继续下钻，其次在后台服务支持的情况下，可以对历史数据做预计算。

数据存储和获取

有了数据模型，需要确定一个高效的数据存储和数据定位的方式，因为结果数据大多是非半结构化数据，而且低维度的的数据量数据量较大，所以采用 NoSQL 来存储数据。

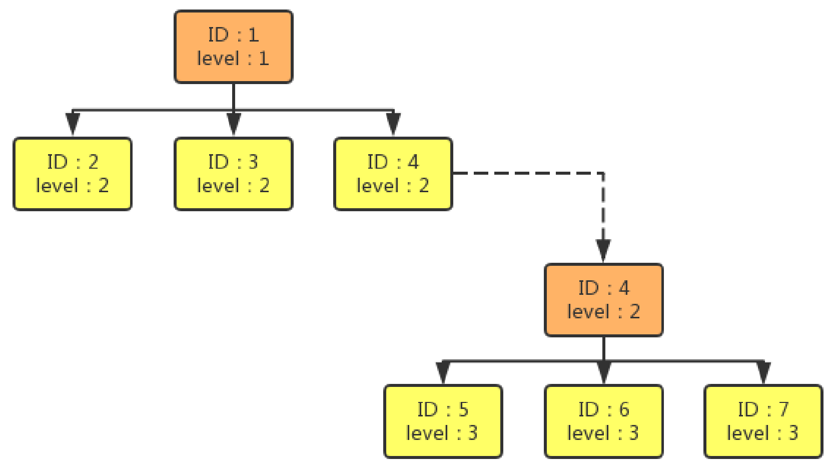


图 5　数据存储和提取方式示意图

如图 5 所示，ID 表示地理维度值，level 表示地理维度等级，数据节点（包含根和叶子节点）以 ID+level 为 Key，转化为树形 JSON 格式数据存储，通过 ID+level 可以唯一获取到一个数据，在数据量不大的情况下，还可以通过级联获取下级模型，即图中虚线代表级联获取下级数据。

计算模式

在构建的数据模型基础上，该层最核心便是预计算模式，从业务需求出发，数据需要在地理等级这个维度不断下钻，从全国开始，一直下钻到 POI 级别，每个级别

单独分层计算，然后存储计算结果。

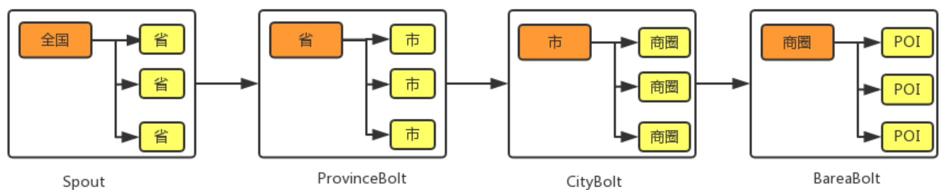


图 6　计算模式示意图

如图 6 所示，每一个矩形代表一级维度的计算，从左到右依次进行维度下钻，从全国的数据依次计算到商圈，计算分层每层单独计算。

实现方式

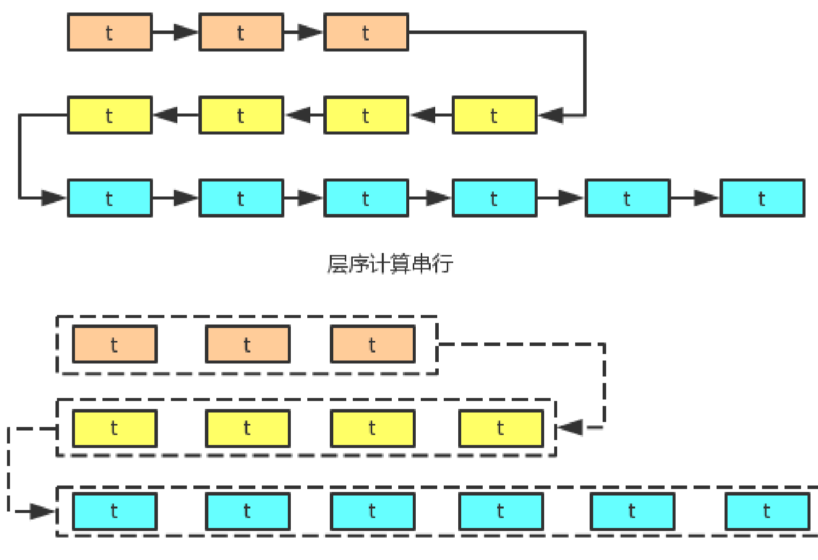


图 7　计算示意图

如图 7 所示，是计算模式的两种实现方式，上半部分是串行层序计算，下半部分是并行层序计算。每部分从上到下分不同颜色区分不同的计算层次。每个矩形对应一个具体 ID 的计算，t 代表计算时间，这里假设所有计算单元的计算时间都相同，方便对比计算时间。

1）串行层序计算

如图 7，普通的串行计算，使用单线程计算，从上到下一层一层计算，这类计算的的痛点有两处，第一，是时间复杂度的，每个计算单元的计算时间都会累计，如计算第一层的时间为 3t，第二层为 4t，第三层为 6*t，总计耗时 13t。第二，是空间复杂度的，因为数据均是调用后台服务获取，计算一层的同时，需要把下级的数据都存储起来，在计算下层时候，再遍历数据计算。

2）并行层序计算

依赖于 Apache Storm 计算框架，将数据抽象成为流，然后通过不同的 Bolt，分别计算不同维度的数据。每一级 Bolt 首先处理数据，然后将下级数据流入下一级 Bolt。同时随着维度的下钻，计算的数据量变得越来越大，通过增加 Bolt 的并发度，加速计算。在预计算的过程中，主要利用了 Storm 高速数据分发和高密度并行计算的特性，规避了串行计算的痛点，首先时间复杂度大幅度降低，如图 7 所示，因为可以并行计算，所以每一层的时间只花费 t，那么总耗时为 3*t，当然这样估算是不准确的，因为没必要在一层所有数据都计算完，才发射数据，可以在每一个计算单元运行完毕，就发射数据。这样就可以形成上下级数据计算流水线，进一步压缩计算时间。其次，空间复杂度大幅度降低，在 Storm 中，不需要保存下级数据，因为数据是不断流动的，计算完毕就会被发射到下级 Bolt。为此，本文采用 Storm 做预计算。计算拓扑结构如图 8。

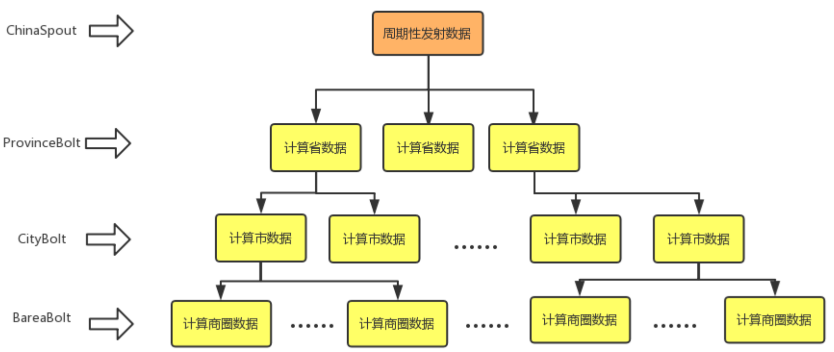


图 8　Srm 计算拓扑示意图

如图 8 所示，数据源头（ChinaSpout）只有一个，该 Spout 内首先计算全国到省的数据，包括全国汇总数据以及省一级的数据，然后立刻将所有省级数据流入下层的 ProvinceBolt，这一层应该考虑增加并发度，因为省到市一级的数据量级开始扩大，设置并发度为 40，在计算完省到市级数据之后，数据开始流入 CityBolt，这一层到市级数据，并发度可以再扩大，目前配置为 300，计算完毕之后，数据流入最后一层 BareaBolt，计算商圈到 POI 级别的数据。各级 Bolt 预计算产生的结果数据，都会存入数据层。存储时遇到一个问题，在计算商圈到 POI 级别的数据时候，发现 POI 的量级比较大，不能直接存储。为了不影响数据模型的通用性，我们队 POI 级别的做了压缩，然后再做存储。为了保障数据的实时性，数据源会周期性产生数据流，更新预计算数据，其实这是 Storm 一类计算模式——DRPC，数据源头就是发射的参数，Storm 的各级 Bolt 承担运算。

该层解决的最大问题，是计算速度慢的问题，通过高密度的并发计算，降低重复数据过滤，大量数据组合，以及批量数据获取慢对响应时间的影响。

数据层

预计算之后的数据需要存储，供业务逻辑使用，存储选型需要满足以下几点：

- 预计算产生的数据模型是树形结构，所以不适合关系型数据库
- 数据具有时效性，数据过期会带来脏数据
- 高密度并行计算，写入并发量大，需要保证写入速度
- 实现灾备，存储不可用时候，需要服务降级

为了满足以上几点要求，选用美团点评内部研发的公共 KV 存储组件 Squirrel 和 Tair 分别来做存储和灾备。其中，Squirrel 是基于 Redis Cluster 的纯内存存储，squirrel 属于 KV 存储，具有写入、查询速度快，并发度高，支持数据丰富，时效好的特点。而 Tair 支持持久化，性价比更高，适合用来做灾备，当 Squirrel 不可用时，使用 Tair 提供服务。

熔断层

预计算过程中，为了实现灾备，还需要使用熔断技术实现服务降级。熔断虽然在上层控制，严格来说应该属于数据层。

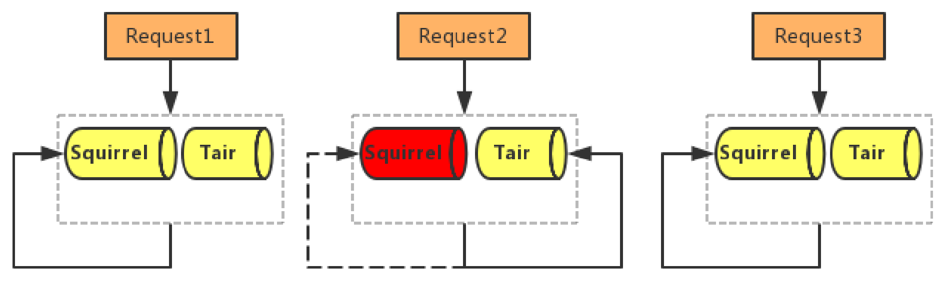


图 9　服务、降级工作原理

熔断技术选用公共组件 Rhino（美团点评自研的稳定性保障平台，比 Hystrix 更加轻量、易用及可控，提供故障模拟、降级演练、服务熔断、服务限流等功能），主要作用是：

a. 保护服务，防止服务雪崩

b. 及时熔断，保障服务稳定

c. 提供多种降级策略，灵活适配服务场景

如图 9 所示，虚线框，代表 Rhino 控制的区域，request1 到来的时候，squirrel 没有问题，正常提供服务。request2 到来的时候，访问 squirrel 发生异常（超时、异常等），请求被切换到 tair。在 squirrel 恢复的过程中，Rhino 会心跳请求 squirrel，验证服务可用性。request3 请到来 squirrel 此刻已经恢复正常，由于 Rhino 会周期检测，所以请求再次被切换到 squirrel 上恢复正常。

中台服务层

数据准备好之后，还不能被业务逻辑直接使用，需要提供统一的服务，应对多变的业务逻辑。该层主要解决如下问题：

- 数据模型修改对业务逻辑有影响，数据服务需对上层具体业务逻辑透明
- 业务逻辑对数据有部分通用操作，需要抽象通用操作，防止数据业务紧耦合
- 存储不可用的时候，需要服务熔断和降级，降低对业务逻辑的影响
- 服务扩展增强能力，不能影响正常业务逻辑

该层对外提供 RPC 服务，直接处理数据模型，提供数据分页、数据压缩、数据解压、数据筛选、数据批量提取以及数据原子提取 等功能，基本覆盖了大部分对数据的操作，使得业务逻辑更加简单。提取数据的时候，加入 Rhino 实现服务熔断和降级，为业务逻辑层提供稳定可靠的服务。因为该层直接操作模型数据，所以即使数据模型有改动，也不会对业务逻辑造成影响，大大降低数据和业务的耦合。另外该层支持服务横向扩展，在消费者大量增加的情况下，仍然能保证服务可靠运转。

通过一系列的抽象和分层，最终业务逻辑直接使用简单的服务接口就可以实现，客户端的响应从最开始的十几秒，提升到 1 秒以内，并且数据和代码之间的耦合大幅度降低，对于后面的业务变化，只需要修改数据模型，增量提供若干中台服务接口，即可满足需求，大大降低了开发难度。

作者简介

张腾，美团点评系统开发工程师，2016 年毕业于西安电子科技大学，同年加入招银网络科技，从事系统开发以及数据开发工作。2017 年加入美团点评数据中心，长期从事 BI 工具开发工作。

智能分析最佳实践——指标逻辑树

萍丽　夷山

背景

所有业务都会面对“为什么涨、为什么降、原因是什么?”这种简单粗暴又不易定位的业务问题。为了找出数据发生异动的原因，业务人员会通过使用多维查 询、dashboard 等数据产品锁定问题，再辅助人工分析查找问题原因，这个过程通常需要一天时间。几乎每种业务角色的用户都在做相似的分析，但在业务 方分析人员发生工作变动时，分析方法难以得到较好传承。因此我们需要一款自动给出分析结论的智能化数据产品来解决上面的问题，产品的基本功能如图 1 所示。

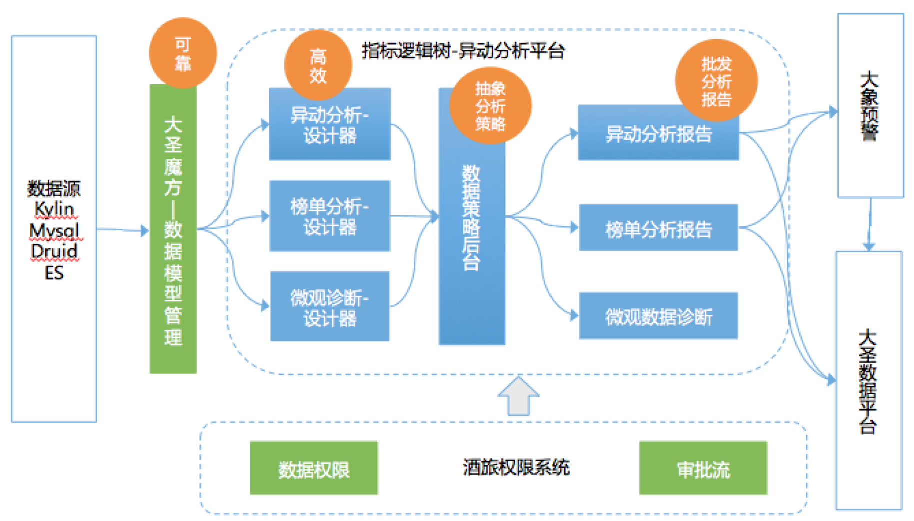


图 1　产品架构图

由上图可知，指标逻辑树就是我们抽象出来的智能异动分析数据产品的最佳实践。它将固定的分析方法和业务场景抽象出来，套用灵活的数据源（包含 Kylin、MySQL、Elasticsearch、Druid 等），自动生成符合各类用户的异动分析报告；它能够直接给出分析结论进而快速落实业务行 动，降低分析成本和决策周期。选定两个

时间周期，指定指标顺序，通过指标逻辑树就可找出导致核心指标发生异动的关键指标，同时可对单一指标进行细分维度拆 分，锁定细分维度对整体的影响。

挑战

指标逻辑树作为一款支持酒旅各业务线的异动分析数据产品，面临的挑战如下：

- 基础指标多、维度多，且来自于不同的数据源。
- 支持多种异动分析算法。
- 自定义计算指标。

针对上面的挑战，我们给出如下的解决方案。

解决方案：指标逻辑树

体系架构

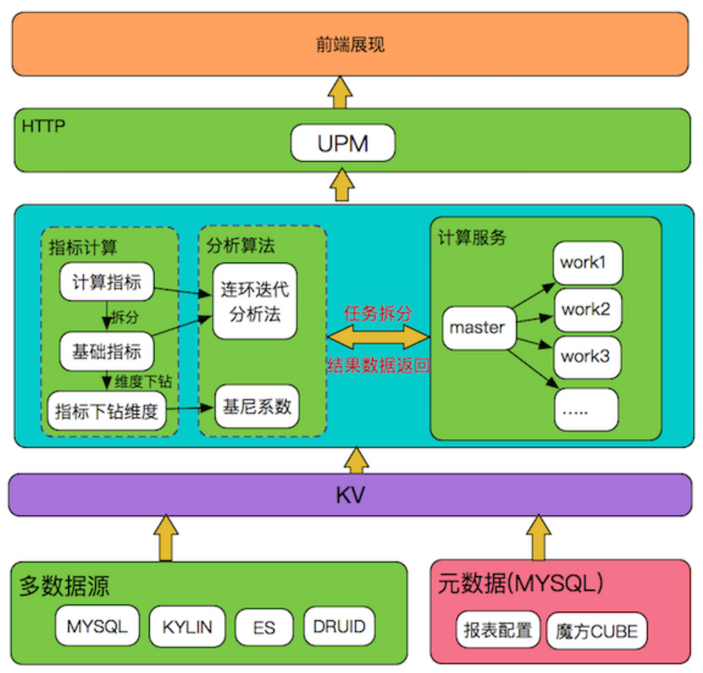


图 2　指标逻辑树体系架构

如图 2 所示：

1. 指标计算，用于解决基础指标多、维度多，且来自于不同数据源的问题以及自定义计算指标的问题；
2. 分析算法，用于支持多种异动分析算法；
3. 计算服务，采用 master-work 的方式解决查询性能的问题。

具体方案

指标计算

指标计算包含指标漏斗、基础指标序列、指标分类，它们之间的关系如图 3 所示。

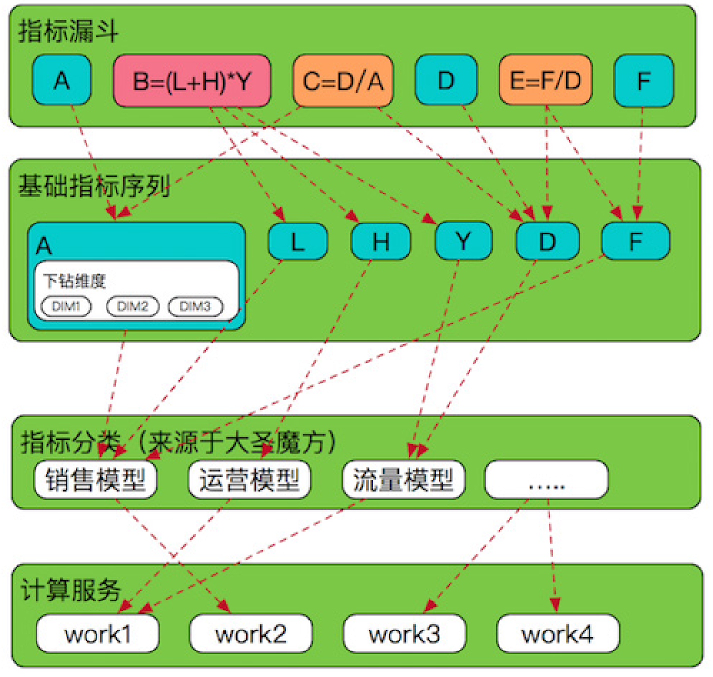


图 3　指标计算

如图 3 所示，指标漏斗为用户自定义的有序指标序列，包含基础指标和计算指标（如，B=(L+H)*Y）；基础指标序列，是将指标漏斗中的计算指标按照顺序拆分之后的指标序列；指标分类采取大圣魔方（可以参考大圣魔方：https://tech.meituan.com/dsmf.html ）配置的规则对基础指标进行分类。

分析算法

目前指标逻辑树支持两种异动分析算法，后续可以根据需要进行扩展。

- 生成瀑布分析图的连环迭代分析法。
- 根据指标下钻维度方案，生成单个指标解释度的基尼系数算法。

下面分别介绍这两种算法在指标逻辑树中的运用。

连环迭代分析法

连环迭代分析法，用于从用户自定义的有序指标列表中找出导致核心指标发生异动的关键指标，如图 4 可知，本期结果指标 E 产生的波动 , 主要由于 A 指标的波动影响。

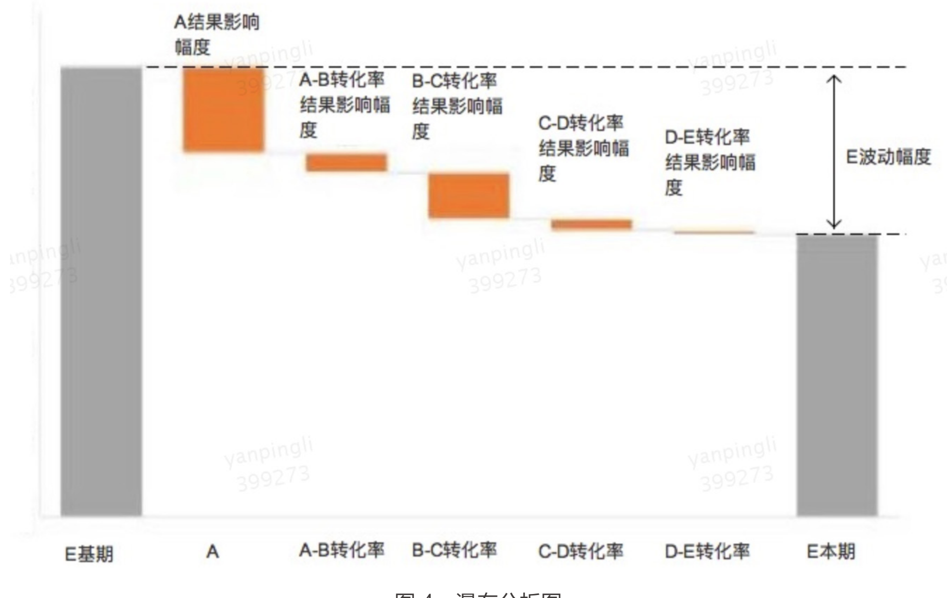


图 4 瀑布分析图

如图 5 所示，意向 UV、访购率、人均单量、连带率、SKU 单价等几个指标中的任意一个发生数据波动，都可能引起支付 GMV 的波动。采用连环迭代分析法，可以确定某个具体指标在本期支付 GMV 的波动中产生的影响最大。算法公式，支付 GMV= 意向 UV* 访购率 * 人均单量 * 连带率 *SKU 单价。

图 5 指标漏斗

基尼系数

基尼系数 A/(A+B)，用于计算各下钻维度方案对单个指标波动的影响程度，横轴用特征分组基期累计占比，纵轴用波动值累计占比（可以为负值），基尼系数越大说明该特征对波动的解释效果越好。

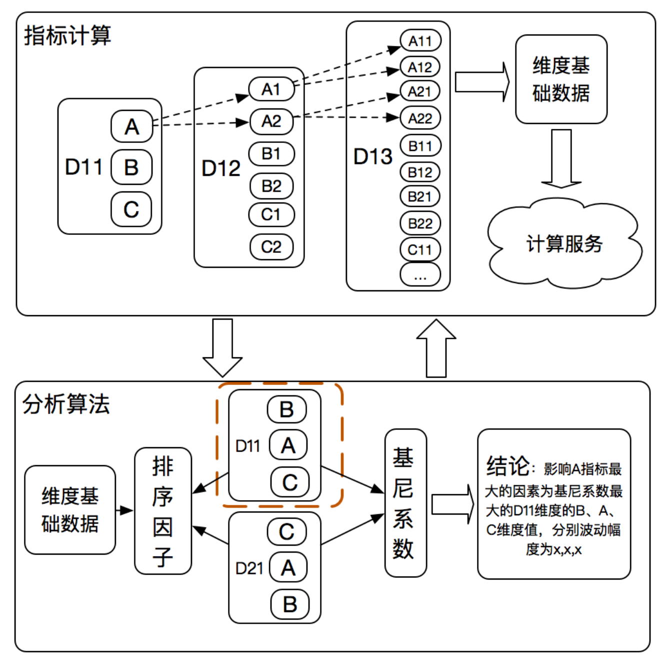


图 6 基尼系数计算

如图 6 所示，指标计算，用于获取层级下钻维度中各个维度的基础数据，如各个城市等级的本期、基期值等信息；分析算法，根据维度基础数据计算出排序因 子，利用排序之后的排序因子计算各特征分组的基期累积占比及波动值累计占比，进而获取到基尼系数；最终选取基尼系数最大的特征作为最终解释。

计算服务

随着业务分析需求的增加，分析用户自行配置的指标序列以及针对单个指标的下钻维度方案将会急剧增加，随之带来的影响就是单个请求需要支持大量的查询任务，因而提升并行计算能力是提升系统性能的一个关键因素。如图 7 所示，计算服务包括任务拆分、并行计算和结果合并。

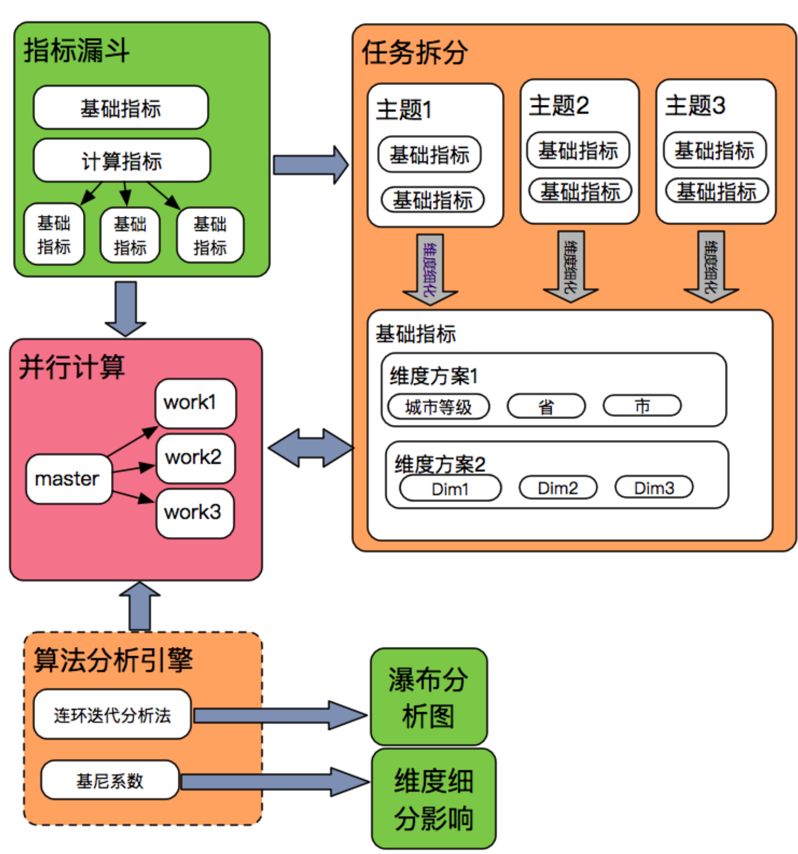


图 7　计算服务

任务拆分

任务拆分分为如下几个步骤：

- 将指标漏斗中的计算指标拆分成基础指标。
- 填充基础指标的细化维度方案，记录指标的各个维度方案及各方案下的层级下钻维度。
- 对基础指标按照数据模型和维度方案进行分类。

并行计算

并行计算提供分布式计算功能，主要处理的是任务拆分之后的细粒度查询任务。

查询任务主要有以下两类：

- 按照数据模型分类之后的指标序列查询任务，需要分别查询本期和基期值，查询量相对较少。
- 按照数据模型和维度方案分类之后的查询任务，需要分别查询本期和基期值，涉及到细化维度，查询量比较大。

结果合并

结果合并主要是针对计算指标来说的，计算指标是分析用户自定义的针对基础指标的一组计算公式。并行查询的结果是针对基础指标的，需要合并基础指标的 查询结果数据，生成符合计算公式的指标数据。结果合并模块需要做两部分的工作，一是解析计算公式，二是根据已有的数据，按照计算公式生成新的数据。

系统中用到数据组装的模块主要有如下：

- 如图 8 所示，根据拆分之后的基础指标数据，生成满足计算公式的计算指标数据。
- 如图 9 所示，根据拆分之后的下钻维度基础数据，分别计算出各个维度的数据，生成符合计算公式的下钻维度数据。

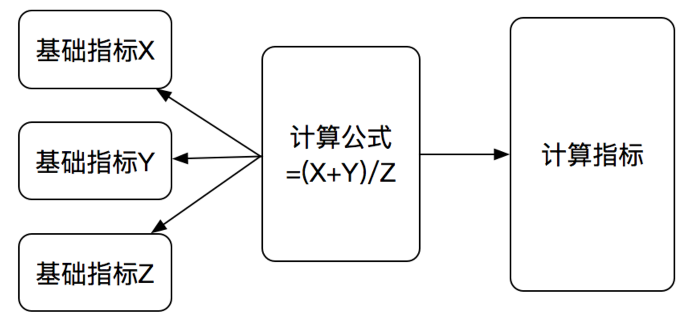


图 8 计算指标数据组装

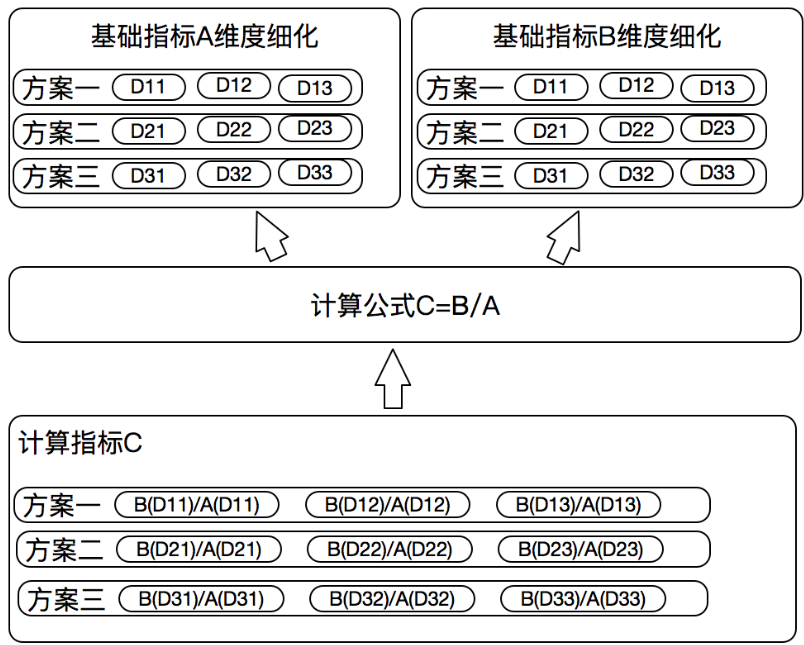


图 9 指标下钻维度数据组装

总结

指标逻辑树在美团点评酒店旅游各业务线中已经得到了一定的应用，并收获了大量好评。本文只是指标逻辑树的一个总纲，目前产品尚处于初级阶段，后续还有很多功能需要完善。

大圣魔方：美团点评酒旅 BI 报表工具平台开发实践

振兴　夷山　米彤

背景

当前的互联网数据仓库系统里，数据中心往往存放了大量 Cube 化或者半 Cube 化的数据。如果需要将这些数据的内在关系体现出来，需要写大量的程序和 SQL 来发现数据之间的内在规律，往往会造成用户做非常多的重复性工作；而且由于没有数据校验的机制，还容易出错，无法直观查看各种数据（没有可视化的 UI 图表）。这时就急需一款基于 Cube 的报表工具快速为用户提供报表服务，可以完成多维查询、上卷、下钻等各种功能。为此，美团点评酒旅技术团队开发了大圣魔方。

难点

一款好的 BI 报表工具，需要考虑并能够解决如下问题：

- 统一数据源
- SQL 生成
- 跨数据源数据聚合
- 自定义计算指标
- 数据权限
- 标准化 UI 组件，自助生成可视化报表

解决方案——大圣魔方

体系架构

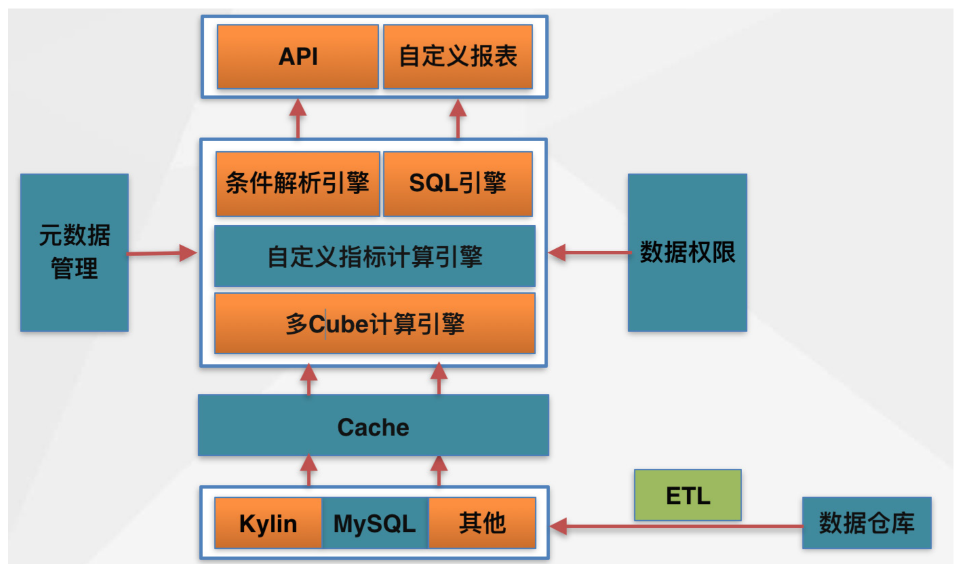


图 1　大圣魔方体系架构

具体方案

1. 统一数据源

提供多数据源查询服务，需要解决的问题主要是两个：

1. 以什么样的统一方式从数据源获取数据。
2. 不是所有的数据源引擎都能提供 OLAP 服务和数据聚合的能力，我们需要从上层考虑，去实现数据的聚合、上卷、下钻、切割、自定义计算等功能。

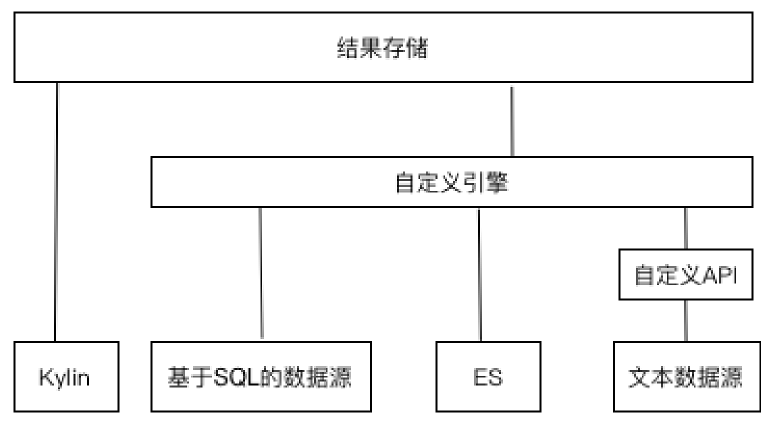


图 2　大圣魔方多数据源

大圣魔方上对能够通过 SQL 查询的数据源，例如 MySQL 和 Kylin 都通过统一 SQL 查询来获取数据；对于 ES（Elasticsearch）采用 ES 提供的 API 来查询；对于普通文本格式的数据采用自定义 API 从数据源获取数据。

如图 2 所示，大圣魔方只是从数据源里面获取基础的数据，之后通过实现自己的计算引擎对数据进行聚合、切割等操作，对此，魔方中设置了四个引擎，用于实现不同的功能。

2. SQL 生成

对于 SQL 的生成也存在两个问题：

1. 不是所有的支持 SQL 的数据源，都支持标准的 SQL，同时，支持标准 SQL 的数据源也会支持带有自身特征的 SQL。
2. 根据用户选择的条件、维度和指标，动态生成 SQL 的核心内容。

针对第一个问题，我们对 SQL 模板进行了定义，当选择不同的数据源时，根据数据源的 Dialect 选择不同的 SQL 模板，而这就决定了 SQL 的组成部分（骨架）。

为了解决第二个问题，我们在 SQL 模板的基础之上做了内容填充和替换操作，选择具体的维度、指标和筛选项的值，再填充到 SQL 模板的不同地方，最终就会生成能够被数据源执行的 SQL。

在 SQL 生成的时候也考虑过其它的框架，如 Apache Calcite Avatica、Alibaba 的 Druid，但是最终都放弃了，原因也是基于两个方面：

1. 这些框架庞大且功能多，适用于我们场景的 SQL 生成的部分 API 使用起来过于复杂。
2. 大都是基于标准的 ANSI SQL-92，很难个性化地生成我们所需要的 SQL。

最终，我们采用了 SQL 模板和字符串填充替换操作来完成。为此我们在 Java 的正则表达式基础之上做了一个功能很多的字符串操作类库。

3. 跨数据源数据聚合

一般情况下，同一个数据源的大部分数据源引擎都能够支持多表的 join 操作，但是也存在不支持的，例如老版本的 Kylin 就不支持多 Cube 的 join 操作，还有一个更重要的问题是数据源引擎无法解决跨数据源的数据聚合问题，必须要自己实现数据的聚合操作，一般的情况下需要自己去实现 inner join、left outer join 和 full outer join 的逻辑。

大圣魔方实现了 inner join 和 left outer join 两个逻辑，因为 full outer join 的需求场景不是很多，所以没有实现。下面是大圣魔方的实现代码：

inner join 核心代码

```
private void join(List<Map<String,String>>[] contents,List<Project>
sharedList,final int n,int[] rowsStatus,LinkedList<MatchRow> result){
        if(this.cubeJoin==1){
            throw new java.lang.IllegalArgumentException("left join call
leftJoin method,not call join method");
        }

        if(n<contents.length){
            List<Map<String,String>> list = contents[n];
            for(int k=0;k<list.size();k++) {
                boolean equal = true;
                if(n!=0) {
                    Map<String, String> prev = contents[n - 1].get
(rowsStatus[n - 1]);
                    Map<String, String> cur = list.get(k);
```

```
                    for (Project proj : sharedList) {
                        String key = proj.fieldName.toUpperCase();
                        if (key.matches("^\\d+$") || key.equals("*")) {
                            key = "_";
                        }
                        key = proj.isCompanion() ? key + proj.getFactId() : key;
                        String prevValue = prev.get(key);
                        String curValue = cur.get(key);
                        if (prevValue == curValue) {
                            continue;
                        }

                        if (prevValue == null || curValue == null ||
!prevValue.equals(curValue)) {
                            equal = false;
                            break;
                        }
                    }
                }
                if (equal) {
                    rowsStatus[n] = k;
                    if(n==contents.length-1){//last dataset match
                        MatchRow mr = new MatchRow();

                        List<MatchRow.DatasetRow> tmp = new ArrayList<>();
                        for(int i=0;i<rowsStatus.length;i++){
                            MatchRow.DatasetRow dr = new MatchRow.DatasetRow();
                            dr.setDatasetIndex(i);
                            dr.setRowIndex(rowsStatus[i]);
                            tmp.add(dr);
                        }

                        mr.addMatchRow(tmp);
                        result.add(mr);
                    }else{
                        join(contents,sharedList,n+1,rowsStatus,result);
                    }
                }
            }
        }
    }
```

上述代码就是通过回溯算法实现 inner join 的核心逻辑，具体解析如下：

- contents 参数表示每个数据源里面的结果集。
- sharedList 表示关联的字段。

- n 和 rowsStatus 是回溯算法记录状态用的。
- result 里面包含的是符合 join 条件的记录。
- MatchRow 里面记录的是一个数据源里面的某一行与其余的数据源里面的那一行是相等的，记录的是下标号。

只有当 sharedList 里面的每个字段都相等的时候，两条记录才满足 inner join 的条件。这个算法是一个通用算法，因为是通过回溯算法实现的，所以要 join 的数据源理论上可以有无限个。

left outer join 核心代码

```
private boolean leftJoin(List<Map<String,String>>[] contents,List<Project>
sharedList,final int n,int[] rowsStatus,LinkedList<MatchRow> result){
        boolean leftJoinMatch = false;
        if(n<contents.length){
            List<Map<String,String>> list = contents[n];

            for(int k=0;k<list.size();k++) {
                boolean equal = true;
                if(n!=0) {
                    //in left join,compare with the first dataset.
                    Map<String, String> prev = contents[0].get(rowsStatus[0]);
                    Map<String, String> cur = list.get(k);

                    for (Project proj : sharedList) {
                        String key = proj.fieldName.toUpperCase();
                        if (key.matches("^\\d+$") || key.equals("*")) {
                            key = "_";
                        }
                        key = proj.isCompanion() ? key + proj.getFactId() : key;
                        String prevValue = prev.get(key);
                        String curValue = cur.get(key);
                        if (prevValue == curValue) {
                            continue;
                        }

                        if (prevValue == null || curValue == null ||
!prevValue.equals(curValue)) {
                            equal = false;
                            break;
                        }
                    }
                }
                if (equal) {
```

```
                    leftJoinMatch = true;
                    rowsStatus[n] = k;
                    if(n==contents.length-1){//last dataset match
                        MatchRow mr = new MatchRow();

                        List<MatchRow.DatasetRow> tmp = new ArrayList<>();
                        for(int i=0;i<rowsStatus.length;i++){
                            MatchRow.DatasetRow dr = new MatchRow.DatasetRow();
                            dr.setDatasetIndex(i);
                            dr.setRowIndex(rowsStatus[i]);
                            tmp.add(dr);
                        }
                        mr.addMatchRow(tmp);
                        result.add(mr);
                    }else{
                        //if next dataset is not match,use the next's next...
                        for(int loopFlag=n+1;loopFlag<rowsStatus.
length;loopFlag++){
                            boolean match = leftJoin(contents,sharedList,
loopFlag,rowsStatus,result);
                            if(match){
                                break;
                            }
                            rowsStatus[loopFlag]=-1;
                            if(loopFlag==contents.length-1){
                                MatchRow mr = new MatchRow();

                                List<MatchRow.DatasetRow> tmp = new ArrayList<>();
                                for(int i=0;i<rowsStatus.length;i++){
                                    MatchRow.DatasetRow dr = new
MatchRow.DatasetRow();
                                    dr.setDatasetIndex(i);
                                    dr.setRowIndex(rowsStatus[i]);
                                    tmp.add(dr);
                                }
                                mr.addMatchRow(tmp);
                                result.add(mr);
                            }
                        }
                    }
                }
            }

        }
        return leftJoinMatch;
    }
```

上面的代码是 left outer join 的实现逻辑，同样也是用的回溯算法，它与 inner join 有 2 个不同之处：

1. left outer join 的数据源匹配逻辑是当第一个数据源与第二个数据源没有匹配的时候，会继续与第三个数据源进行匹配操作，原因是数据源的顺序导致了不匹配，继续往下匹配就可以避免这个问题。
2. 行与行做相等操作的时候，右边没有匹配行的时候，左边的行继续保留，这个是 left outer join 的逻辑决定的。

4. 自定义计算指标

使用自定义计算的原因，主要是基于下面的两个方面：

1. 数据源引擎不支持数据的混合运算或有特殊逻辑的数据处理。
2. 结果数据跨数据源。

对此，我们对大圣魔方做了如下操作：

1. 通过 Java 里面的 ScriptEngine 进行封装来实现数据列的混合运算，不需要自己再去写编译程序解析。对于特殊的数据处理，例如同环比这样的特殊指标，需要单独定义接口，让实现类继承改特定接口，实现类是一个特殊的指标，它需要进行多次数据查询，将最终的结果通过 ScriptEngine 进行运算。
2. 第二个问题是在上文中“跨数据源数据聚合”的基础上实现的，数据聚合后通过 ScriptEngine 做最后的处理。

5. 数据权限的问题

只要是有数据展示，数据权限问题就无法避免，权限主要是分为报表的可查看权限和维度、指标权限。权限遇到的最主要的问题是构成权限矩阵的数据量太大，参与者有用户和组织，权限的实体有维度和指标，这样大的数据维护起来的成本很高；其次是权限数据配置会占用很多的人力。

对此，我们做了如下操作：

1. 使用 UPM 控制报表的可查看权限。公司推荐使用 UPM 来控制权限，不过 UPM 也具有一定的局限性，即只能够判断用户或者组织是否满足某个权限，而不能满足获取部分权限数据的需求，例如某个用户对某个权限只拥有一部分权限，那他就无法提供具体数据。但是 UPM 可以提供是否的权限，所以报表的可查看权限可以使用 UPM 来控制，这样可以节约大量的工作。
2. 使用默认任何人都有权限的机制。通过使用默认有权限的这个机制可以大大减少权限数据。需要鉴权的那些维度和指标采用默认无权限的机制，这样两种方案结合，可以最大限度地减少权限数据。
3. 通过走审批流机制自助申请。通过审批流机制可以让用户走自助申请，大大节约权限数据的维护人力成本。

6. 标准化 UI 组件，自助生成可视化报表

报表上展示数据需要有各种各样的图表，没法为用户只做一个统一的报表，这个时候需要用户能够创建自己想要的报表，这时需要提供一个标准的组件库、布局库和一些常用的模板。用户选择好想要的模板，然后选择布局对报表页面进行布局，接着在每个布局里面填充不同的组件，这样就可以构建一张报表了，也就是我们常说的所见即所得的方式。

大圣魔方就是采用上述的机制提供了一套可视化报表编辑工具。使用它可以快速地创建一个报表，管理人员只需要维护对应的组件、布局和模板就行了。

总结

上述几点就是大圣魔方的一个总纲，其中大部分功能已经实现了，还有一小部分处于开发之中（标准化 UI 组件、自助生成可视化报表）。目前大圣魔方已经上线将近一年了，支持了内部众多业务，后续我们还会在 UI 易用性、星型模型、配置简化、元数据同步等方面做一些提高。

美团点评
技术团队
扫码关注美团点评
微信公众号
tech.meituan.com
美团点评技术博客