美团数据平台及数仓建设实践

涵盖数据平台、数仓开发、数据治理、数据分析等

本文档整理自美团技术团队博客：https://tech.meituan.com

本文档首发于公众号：五分钟学大数据

微信直接扫码关注

目录

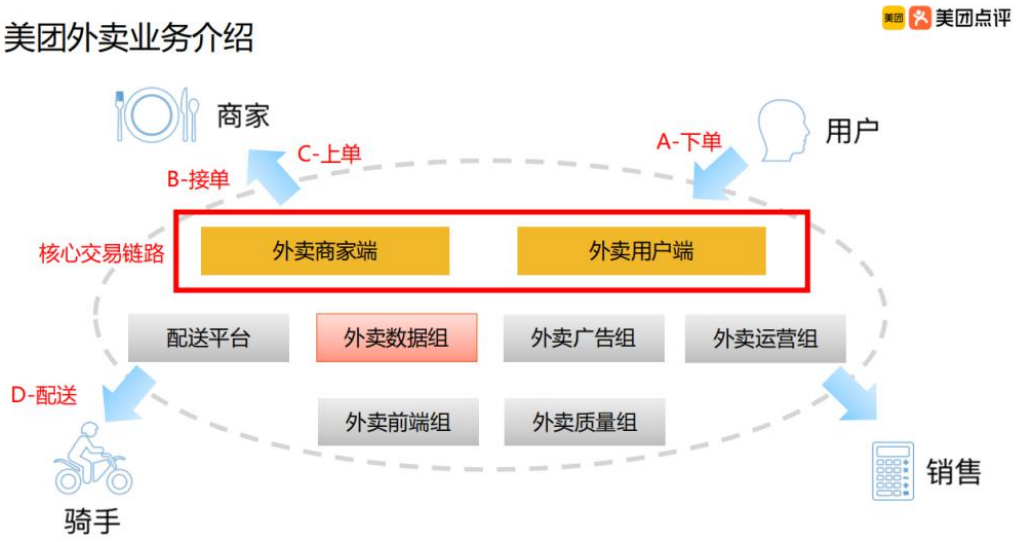
离线数仓

一、美团外卖离线数仓建设实践

导读：美团外卖数据仓库主要是收集各种用户终端业务、行为数据，通过统一口径加工处理，通过多种数据服务支撑主题报表、数据分析等多种方式的应用。数据组作为数据基础部门，支持用户端、商家端、销售、广告、算法等各个团队的数据需求。本文主要介绍美团外卖离线数仓的历史发展历程，在发展过程中碰到的痛点问题，以及针对痛点做的一系列优化解决方案。

01

业务介绍



首先介绍下美团外卖的业务场景， 核心交易链路为：用户可以通过美团的各种用户终端（包括美团外卖的 APP 或者美团 APP、QQ/微信等）下单，然后商家接单、骑手配送，三个阶段完成一笔交易。这一系列交易过程，由包括用户端、商家端、配送平台、数据组、广告组等各个系统协同完成。

这里主要介绍外卖数据组在整个业务中角色。外卖数据组主要是：

- 给用户端、商家端提供业务需求，
- 给前端提供需要展示的数据，
- 给广告、算法团队提供特征等数据，提高算法效率

- 向城市团队提供业务开展所需数据
- 前端提供埋点指标、埋点规范相关数据

1. 整体架构介绍

美团外卖整体分为四层：**数据源层、数据加工层、数据服务层、数据应用层。**

数据源层：包含接入的原始数据，包括客户端日志、服务端日志、业务库、集团数据、外部数据等。

数据加工层：使用 Spark、Hive 构建离线数仓、使用 Storm、 Flink 实时数仓。在数仓之上针对服务对象建设各种数据集市，比如：

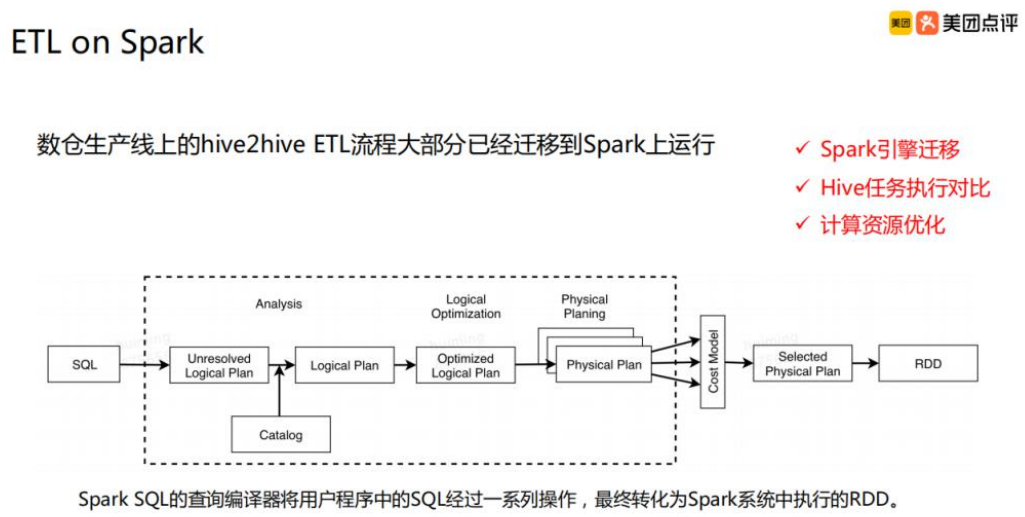
- 面向总部使用的总部数据集市
- 面向行为数据的流量数据集市
- 面向线下城市团队的城市团队集市
- 面向广告的广告集市
- 面向算法的算法特征

数据服务层：主要包括存储介质的使用和数据服务的方式。

- 存储：主要使用开源组件，如 Mysql, HDFS, HBase, Kylin, Doris, Druid, ES, Tair 等
- 数据服务：对外数据查询、接口以及报表服务

数据应用层：主要包括主题报表、自助取数工具、增值产品、数据分析等支撑业务开展，同时依赖公司平台提供的一些工具建设整体数据应用。

2. ETL on Spark



我们离线计算从 17 年开始从 Hive 迁移到 Spark，目前大部分任务已经迁移到 Spark 上运行，任务迁移后，相比之前使用 Hive 整体资源节省超过 20%。相比之下 Spark 的主要优势是：

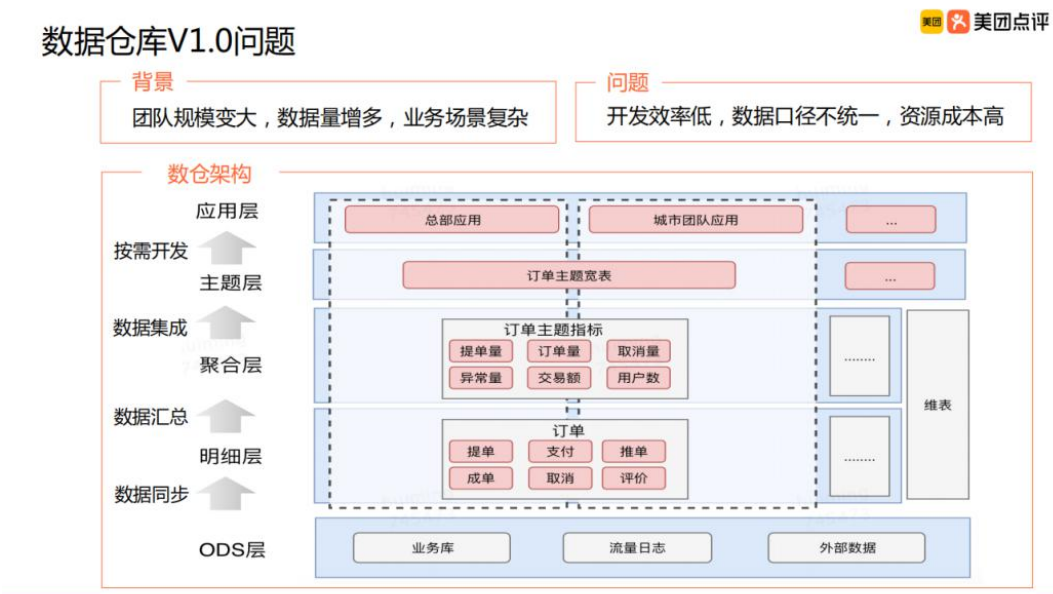
- 算子丰富，支持更复杂的业务逻辑
- 迭代计算，中间结果可以存内存，相比 MR 充分利用了内存，提高计算效率
- 资源复用，申请资源后重复利用

这里简单介绍写 Spark Sql 的任务解析流程：客户端提交任务后，通过 Sql 解析先生成语法树，然后从 Catalog 获取元数据信息，通过分析得到逻辑执行计划，进行优化器规则进行逻辑执行的优化，最后转成物理执行计划提交到 spark 集群执行。

2. 数仓建设

1. 数据仓库 V1.0

2016 年之前。外卖数据组的情况是：团队不大，数据量不多，但是市场竞品较多（饿了么、百度等），竞争激烈， 因此当时数据组的目标是：快速响应业务需求，同时做到灵活多变，支撑业务业务决策，基于这种业务背景和实现目标，当时数仓架构设计如图所示主要分了四层，分别是：ODS 层/明细层/聚合层/主题层/应用层（具体如图示）。



随着数据量、业务复杂度与团队规模的增长， 为更好完成业务需求数据组团队按照应用做拆分，比如面向总部的总部团队、面向城市业务的城市团队，各个团队都做一份自己的明细数据、指标和主题宽表数据，指标和主题宽表很多出现重叠的情况，这时候就像是”烟囱式“开发。这在团队规模较小时，大家相互了解对方做的事情，基本不会有问题；但是在团队规模增长到比较大的时候，多团队“烟囱式”独立作战也暴露出了这种架构的问题，主要是：

- 开发效率低
- 数据口径不统一
- 资源成本高

2. 数据仓库 V2.0

针对上述问题，数据组做了架构的升级，就是数据仓库 V2.0 版本。此次升级优化的目标主要是：

- 简化开发流程，提高开发运维效率
- 明确分层，分主题标准，贯彻执行

完成这个目标的思路如下三个方面：

① 分工标准：

之前面向不同应用建立不同团队完全纵向切分，会导致可以公用的部分明细数据重复开发。为改变这种情况将数据团队改为：数据应用组和数据建模组。各组职责如下：

- 数据应用组：负责应用指标、应用维度、应用模型，这一组的数据建模特点是：自上而下、面向应用。
- 数据建模组：负责基础事实、基础维度、原子指标的数据开发，这一组的数据建模特点是：自下而上、面向业务。

② 分层标准：

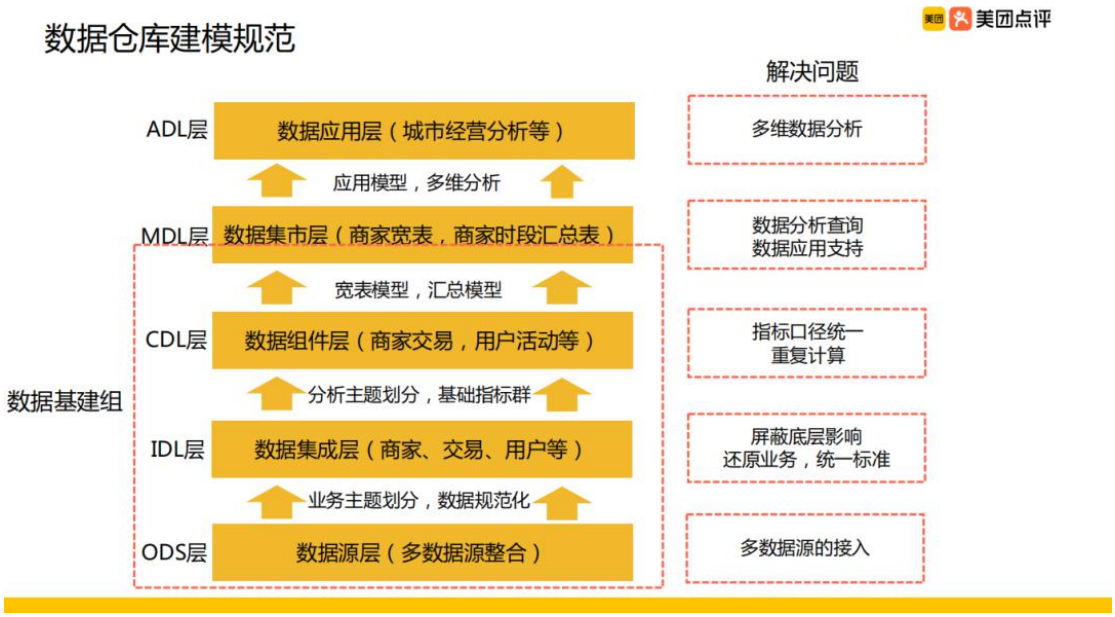
在原有分层的基础上，再次明确各层的职责，比如：明细层用来还原业务过程，轻度汇总曾用来识别分析对象；同时数据加工时考虑数据的共性、个性、时效性、稳定性四个方面的因素，基于以上原则明确数仓各层达到数据本身和应用需求的解耦的目标。具体各层细节在文章接下来的内容会展开来讲。

③ 主题标准：

根据数仓每层的特性使用不同的主题划分方式，总体原则是：主题内部高内聚、不同主题间低耦合。主要有：明细层按照业务过程划分主题，汇总层按照“实体+活动”划分不同分析主题，应用层根据应用需求划分不同应用主题。

2.1 数仓规范

① 数据仓库建模规范



根据前述三个方面的思路，将数仓分为以下几个层次：

- ODS：数据源层，主要职责是接入数据源，并做多数据源的整合
- IDL：数据集成层，主要职责是：业务主题的划分、数据规范化，比如商家、交易、用户等多个主题。这一层主要起到缓冲的作用，屏蔽底层影响，尽量还原业务，统一标准。

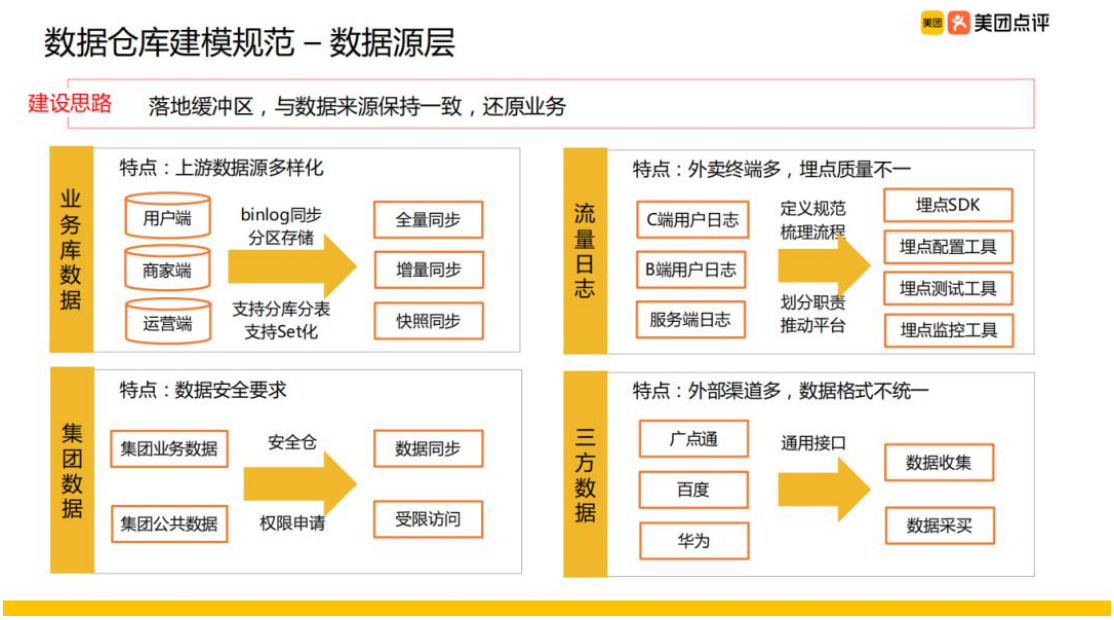
- CDL：数据组件层，主要职责是划分分析主题、建设各个主题的基础指标，比如商家交易、用户活动等多个主题。这样针对同一个分析对象统一了指标口径，同时避免重复计算。

- MDL：数据集市层，主要职责是建设宽表模型、汇总表模型，比如商家宽表、商家时段汇总表等。主要作用是支撑数据分析查询以及支持应用所需数据。

- ADL：数据应用层，主要职责是建设应用分析、支撑多维分析应用，比如城市经营分析等。

其中 ODS/IDL/CDL，以及部分 MDL 集市由数据基建组来做，另外部分数据集市以及 ADL 应用层由数据应用组支撑，分工标准是涉及一些公共的数据集市由数据基建组来完成；数据应用组会围绕应用建设应用数据集市，如流量集市、城市经营集市。

② 数据源层



整体建设思路：从数据源落地到 Hive 表，同时与数据来源保持一致，尽量还原业务。主要由四类数据源：业务库数据、流量日志、集团数据、三方数据。

- 业务库数据：主要是将各个客户端的数据同步 Hive ，主要有用户端、商家端、运营端，同步方法主要采用 binlog 同步，同步方式有全量同步、增量、快照同步三种方式。同时支持业务库分库分表、分集群等不同部署方式下的数据同步。

- 流量日志：特点是外卖终端多，埋点质量不一，比如单 C 端分类就超过十种。为此公司统一了终端埋点 SDK，保证不同终端埋点上报的数据规范一致，同时使用一些配置工具、测试工具、监控工具保证埋点的质量。整理建设思路是：定义埋点规范、梳理埋点流程、完善埋点工具。

- 集团数据：包含集团业务数据、集团公共数据，特点是数据安全要求高。目前公司建立了统一的安全仓，用于存储跨 BU 的数据，同时定义权限申请流程。这样对于需要接入的数据，直接走权限申请流程申请数据然后导入业务数仓即可。

- 三方数据：外部渠道数据，特点是外部渠道多、数据格式不统一，对此我们提供了通用接口对于收集或者采买的三方数据在接入数仓前进行了规范化的清洗。

③ 数据集成层

数据集成层主要是明细数据，与上一层数据源层是有对应关系的。数据集成表的整体建设思路为：

- 抽象业务过程
- 识别实体关系，挂靠业务主题，比如交易过程包括提单、支付等过程，把这些业务行为涉及的事实表进行关联，识别出里面的实体关系
- 兼容数据成本
- 屏蔽业务变化，比如订单状态变化
- 统一数据标准，敏感字段脱敏，字段名称标准化等

如图中示例为提单表建设过程。

④ 数据组件层

数据组件层，主要建设多维明细模型、轻度汇总模型。总体建设思路与建设原则为：

建设思路：

- 识别分析对象，包含分析对象实体以及对象行为
- 圈定分析边界
- 丰富对象属性

建设原则：

数据组件层生成的指标主要是原子指标，原子指标形成数据组件，方便下游的集市层以及应用层拼接数据表。

- 分析对象包括业务实体和业务行为
- 分析对象的原子指标和属性的惟一封装
- 为下一层提供可共享和复用的组件

多维明细模型：

以商家信息表建设过程为例：

- 识别分析对象：首先明确分析对象为商家实体，
- 圈定分析边界：多维明细不需要关联实体行为，只需要识别出实体之后圈定商家属性信息作为分析边界；
- 丰富对象属性：提取商家属性信息，比如商家的品类信息、组织结构信息等

以上信息就形成了一个由商家主键和商家多维信息组成的商家实体的多维明细模型。

轻度汇总模型：

以商家交易表假设过程为例：

- 识别分析对象：分析实体是商家，业务行为是交易，分析对象是商家交易
- 圈定分析边界：圈定提交表、商家信息表、订单状态表、会员表作为商家交易的边界
- 丰富对象属性：将城市、组织结构等维度信息冗余进来，丰富维度属性信息

汇总商家粒度、交易额等原子指标最终建立商家交易表。

⑤ 数据集市层

建设思路：

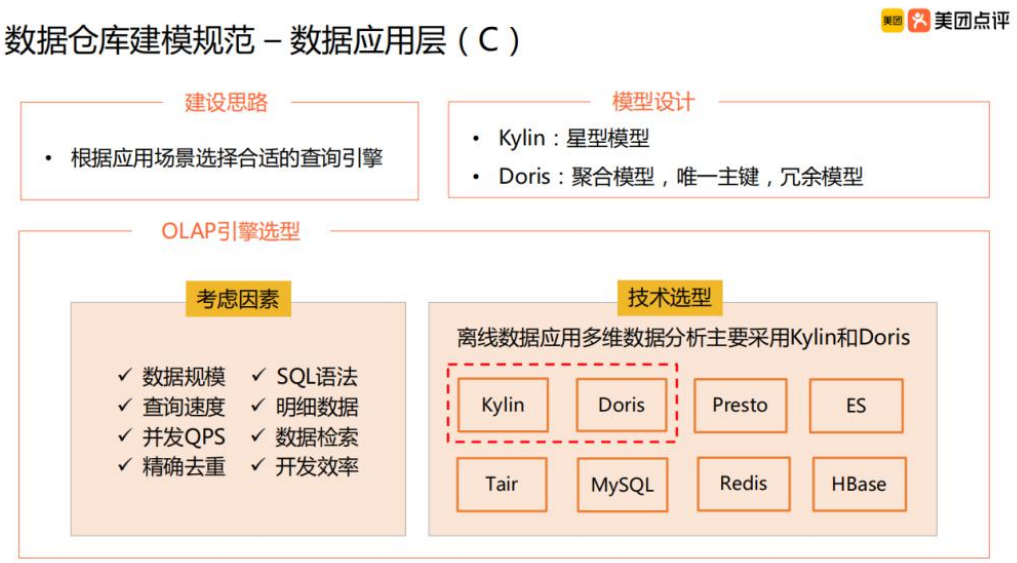
建立宽表模型和汇总模型。两者区别为宽表模型是唯一主键，基于主键拼接各种信息；汇总模型的主键类型为联合主键，根据公共维度关联生成派生指标，丰富信息。

- 宽表模型：订单宽表为例，建设过程为：选定订单实体作为实体对象，然后圈定订单明细、订单状态、订单活动、订单收购等分析对现象通过订单 id 进行关联。这里的宽表模型与数据组件层的多维明细模型的区别在于多维明细模型里的实体对象粒度更细，例如订单宽表中分析对象：订单明细、订单状态、订单活动等都是多维明细模型里的一个个数据组件，这几个数据组件通过订单 id 关联拼接形成了宽表模型。

- 汇总模型：商家时段汇总表为例，建设过程为：选定商家、时段维度作为维度组合，圈定商家和时段维度相关的表，通过公共维度进行关联、维度冗余，支持派生指标、计算指标

的建设。这里区别于组件层的轻度汇总模型，在数据组件层建设的是原子指标，而数据集市层建设的是派生指标。

⑥ 数据应用层



建设思路：根据应用场景选择合适的查询引擎

选型考虑因素：OLAP 引擎选型考虑以下 8 个方面的因素：

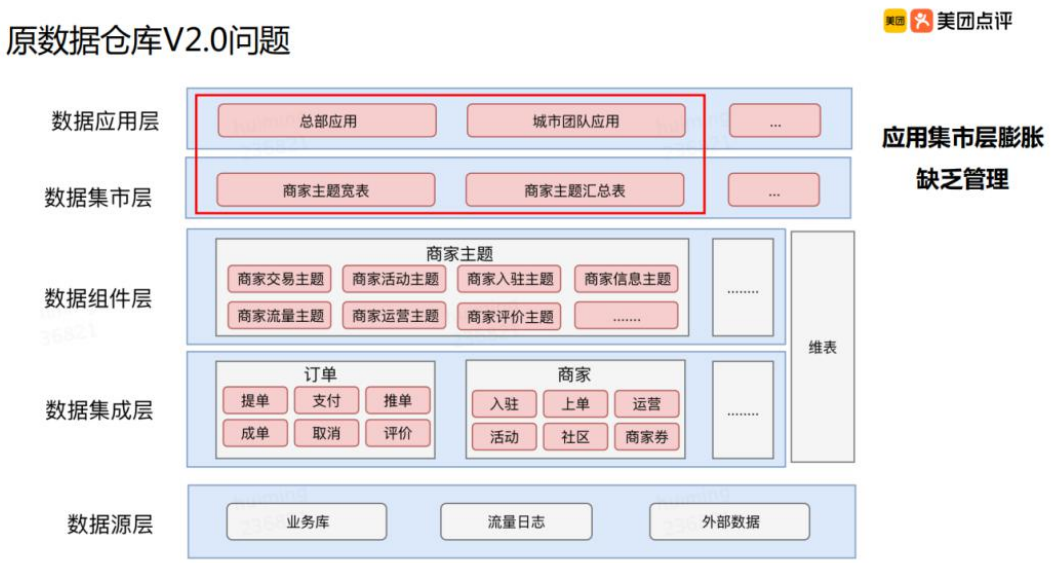
- 数据规模是否适合
- SQL 语法的支持程度如何
- 查询速度怎么样
- 是否支持明细数据
- 是否支持高并发
- 是否支持数据检索
- 是否支持精确去重

- 是否方便使用，开发效率如何

技术选型：早期主要使用 Kylin ，近期部分应用开始迁移 Doris。

模型：根据不同 OLAP 引擎使用不同数据模型来支持数据应用。基于 Kylin 引擎会使用星型模型的方式构建数据模型，在 Doris 支持聚合模型，唯一主键以及冗余模型。

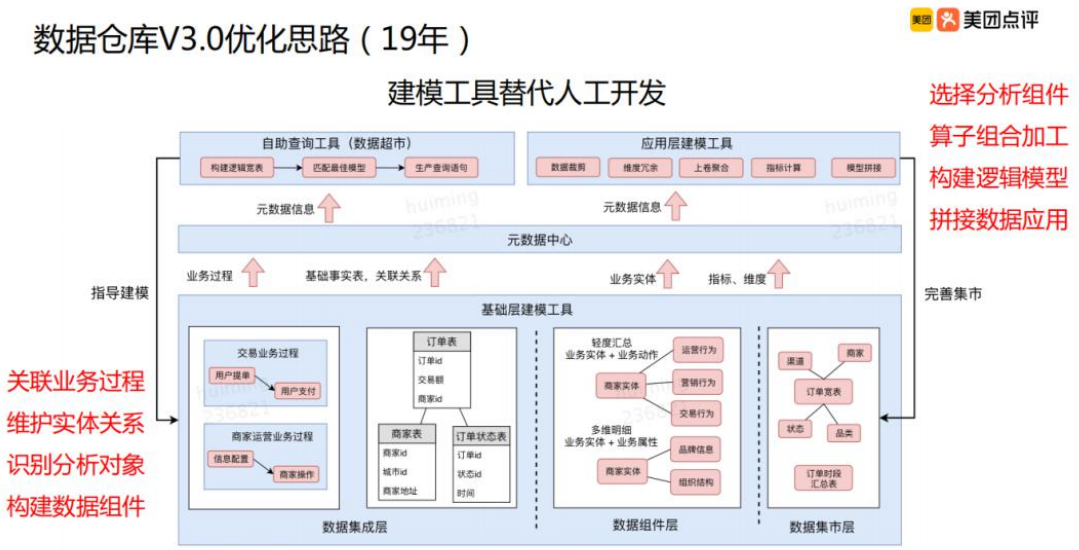
2.2 数仓 V2.0 的缺点



前面几小节对数仓 2.0 做了详细的介绍，在数仓 2.0 版本的建设过程中我们也遇到了一些问题。前面有提到数据集成层与组件层由数据基建组来统一运维，数据应用层是由数据应用组来运维，这样导致虽然在集成层和组件做了收敛但是在应用层和集市层却产生了膨胀，缺乏管理。

面对这个问题，我们在 2019 年对数仓进行了新的迭代，即数仓 V3.0，下面将对此做详细介绍。

3. 数据仓库 V3.0



总体愿景：数仓 3.0 优化思路主要是使用建模工具替代人工开发。

建模工具：分为基础的建模工具和应用层建模工具。

- 基础层建模工具：主要是在元数据中心记录维护业务过程、表的关联关系、实体对象、识别的分析对象，基于维护的信息构建数据组件以供应用层和集市层拼接

- 自助查询工具：根据数据组件和用户选取的需要查询的指标维度信息构建逻辑宽表，根据逻辑宽表匹配最佳模型从而生成查询语句将查询出来的数据反馈给用户。同时根据用户查询情况反过来指导建模，告诉我们需要把哪些指标和哪些维度经常会放在一起查询，根据常用的指标、维度组合建设数据组件

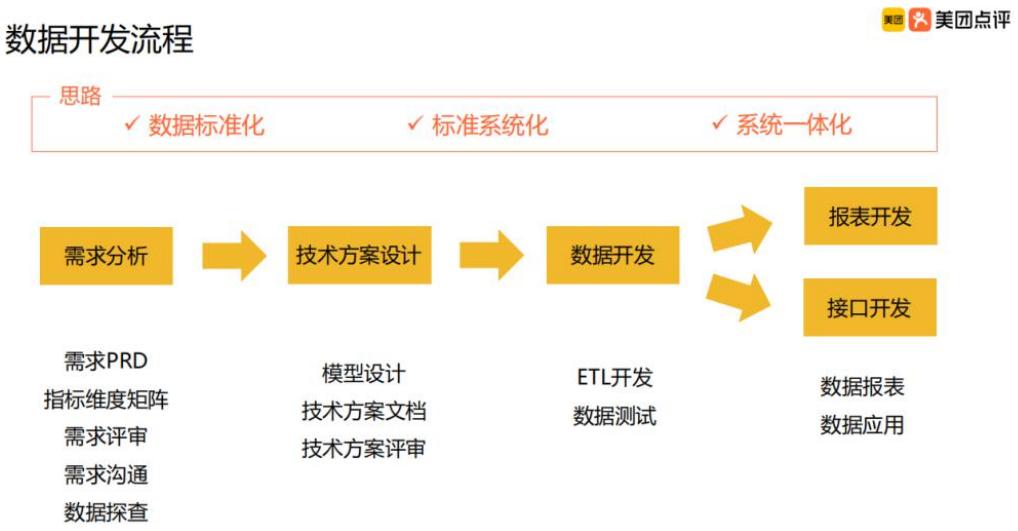
- 应用层建模工具：依赖数据组件，包括数据组件层的多维明细数据、轻度汇总数据以及集市层的宽表等构建构建数据应用。主要过程是获取所需数据组件，进行数据裁剪，与维表

关联后冗余维度属性，按需进行上卷聚合、复合指标的计算，最终把获取到的多个小模型拼接起来构建数据应用

通过整套工具的使得数据组件越来越完善，应用建模越来越简单，以上就是数仓 3.0 的整体思路。

3. 数据治理

1. 数据开发流程



先说下我们数据开发流程，数据开发流程主要分为四个阶段：需求分析、技术方案设计、数据开发、报表开发&接口开发，具体内容如下：

- 需求分析：在需求分析阶段，产品会设计一个需求 PRD 以及列出指标维度矩阵，然后需求评审与需求相关人员进行沟通，做一些数据的探查

- 技术方案设计：完成需求分析之后，形成模型设计的技术方案，同时将方案落地到文档进行技术方案的评审

- 数据开发：完成技术方案的设计与评审就正式进入了数据开发以及测试阶段
- 报表开发&接口开发：数据开发完成之后进入具体应用的开发

在整个数据开发流程中，我们遵循的整体思路是：

- 数据标准化
- 标准系统化
- 系统一体化

即数据符合标准规范，同时将标准规范落地到系统里，最后系统要和周边应用打通，形成一体。下面对各个思路做详细的描述。

① 数据标准化

在数据标准化这块，数据产品团队、数据开发团队、数据分析团队联合建立了数据标准化委员会。数据标准委员会制定了《指标标准规范》、《维度标准规范》以及一些新增指标、维度的流程等一系列规范标准，这样做的好处是：指标维度管理有据可依，指标维度管理有组织保障，保障各业务方指标维度口径清晰统一。

② 标准系统化

明确了数据标准各项规范，需要将这些标准化规范落地到系统，就是我们的数据治理平台，我们的数据治理平台由自建系统 + 集团数据服务构成。这里面主要有四层：**数据生产工具、集团基础平台、元数据层、数据服务层。**

- 数据生产工具：数据生产主要依靠平台的计算能力，包括离线生产平台、实时生产平台、调度管理平台
- 集团基础平台：数据生产工具之上是集团基础平台，包括数据资产管理、元数据管理、数据质量管理、资源管理以及权限管理
- 元数据层：元数据与数据服务都是美团外卖自己业务做的一些工作，元数据层包括数据模型、表/字段、主题/层级、指标/维度、业务过程、词根/词库
- 数据服务层：服务层包含有数据标准化，前面提到的指标流程、维度流程、认证管理都是在这里落地；同时把业务管理起来，包括理业务大盘、业务过程、数据域管理；然后还管

理我们的数据模型包括指标维度矩阵、事实逻辑模型、维度逻辑模型；同时提供元数据服务，包含业务元数据、技术元数据以及维度服务；还有刚才提到的一些在线建模工具

图片右边展示了我们的元数据模型，从下而上，我们首先维护词根组成的词库，同时词根、词库组成我们的指标和维度，其中维度分为维表和码表，指标在确保唯一性的前提下划分业务过程，同时区分原子指标、派生指标、计算指标；然后由维度和指标拼接成字段、字段组成表，表再和业务主题、业务过程相关联，识别出实体、行为，区分事实表、维度表同时确定表所在的层级，最后由一张张的表组成我们的数据模型，整个过程就是我们的元数据模型。

③ 系统一体化

有了前面的数据信息之后，我们和下游对接就比较方便。使用到数据治理平台的数据下游有：

- 报表系统：报表展示所需的指标维度信息、模型信息都是来自于数据治理平台，
- 数据超市：数据超市的自助取数里根据指标、维度、数据组件构建数据查询，这些信息都是来自数据治理平台，

- 海豚数据平台：海豚数据平台是我们的外卖数据门户
- 异常分析平台：对指标维度的波动进行监测分析
- CRM 系统：面向一线城市团队的数据展示
- 算法平台：向算法平台提供标签、特征数据的管理
- 画像平台：管理画像平台的人群标签
- 数据 API 服务：对外提供的 API 模型以及接口的信息
- 到家数据检索：到家业务元数据联通，统一检索元数据信息
- 公司元数据平台：向公司元数据平台提供元数据信息

通过与各个下游不同形式的对接，数据治理平台完成了整个下游数据应用的联通，以及支持数据使用与生产，形成了一体化的系统。

2. 资源优化

资源优化方面，在美团会把核算单元分成若干个租户 ，然后把资源分配给各个租户，在同一个租户里各个项目组协调分割分配到资源，项目组由任务和数据组成。

我们把租户与对应主题进行挂靠，这样就可让租户有对应的接口人管理，比如把外卖核算单元分为：数仓租户、广告租户、算法租户以及其他的业务租户，让每个租户对应一个接口人，与接口人对接资源优化方案、规则，最后由接口人推动。

我们的优化规则主要分为三个方向：

- 流量：流量方面，主要是对无效 ODS 下线从而降低存储与传输成本；同时埋点数据生命周期减少成本浪费，目前用户日活几千万，上报的流量日志是有上百亿，埋点若不再使用对存储与传输成本浪费较大；另外对日志进行序列化压缩处理降低成本

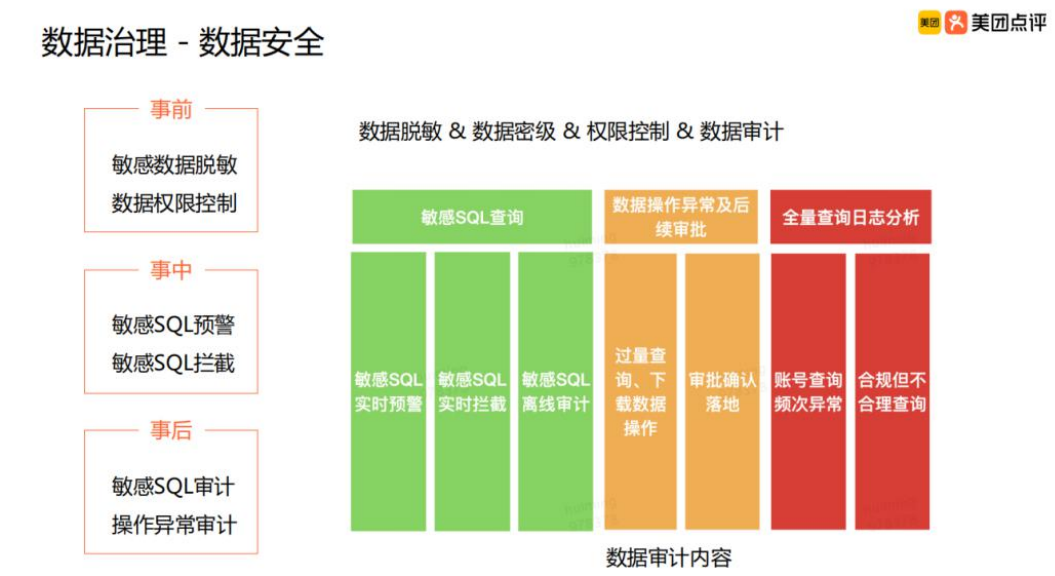
- 存储：存储方面我们使用 ORC 进行压缩，同时对冷热数据做了生命周期管理，另外也通过模型的优化以及文件的优化把存储成本控制住

- 计算：计算方面对无效任务进行下线、 ETL 任务优化；同时对数据开发收敛，把业务中公共部分收敛到数据组

有了优化规则，针对规则的运营监控流程为：首先对账单分析，账单主要有离线/实时计算资源、存储资源、ODS 数据收集使用的资源、日志中心所使用的资源，分析帐单后定义运营规则并将规则落地到数据运维平台，由数据运维平台将任务推送到相关责任人，责任人收到通知后，在数据资产中心做相关处理，同时数据运维平台会做成本监控，对超出配额&预算异常进行报警。

这样我们通过建立统一规则并将规则分发到不同租户落地执行，完成数据资源优化的目标。

3. 数据安全



数据安全方面主要是对数据脱敏，数据保密等级的设定（C1~C4），数据申请做权限控制，审计数据使用的方式，我们分三个阶段完成数据安全的治理：

- 事前：包括敏感数据脱敏、数据权限控制。针对事业部内、事业部外使用不同的权限流程控制。
- 事中：包括敏感 SQL 的预警与拦截，针对敏感 SQL 我们进行拦截并由数据安全人员进行审批
- 事后：包括敏感 SQL 审计，操作异常审计。输出敏感 SQL 审计的月报发到对应的部门负责人，审核内容主要有敏感 SQL 的查询、数据操作异常及后续审批还有全量查询日志分析。

4. 未来规划

1. 未来规划思考

最后介绍下，我们对未来的规划，对未来规划的思考主要是在业务和技术两个方面：

- 业务目标：通过数据赋能业务，帮助完成未来实现业务订单量和收入的高速增长的目标
- 技术目标：提高高效、易用、高质量、低成本的数据服务

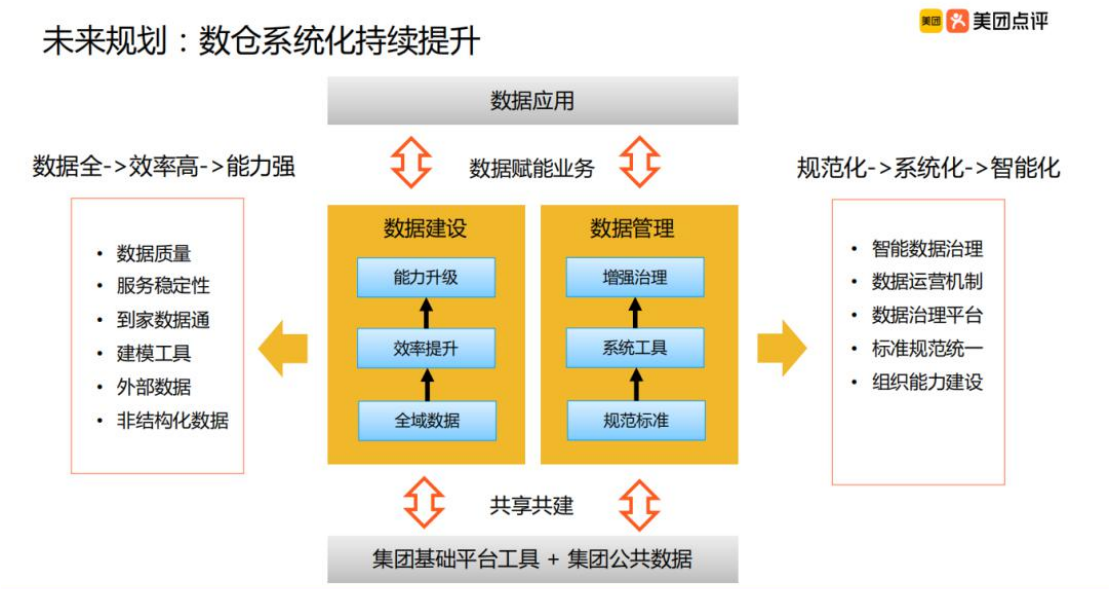
这里面数据价值的具体体现，总结为以下几点：

- 基础的数据能力：保障数据服务的稳定性，以及数据的时效性越来越高，以及数据服务的覆盖度足够广、足够全，扩大数据服务内容
- 运营决策支持：及时洞察业务变化，直到业务完成运营决策
- 数据商业变现：通过增值型数据产品，把数据进行商业变现
- 业务数据支持：更好的支持对接的各个业务系统，从而提高整体数据价值

- 算法效率提升：针对算法加工特征所需的数据提供更好的支持，以提高算法模型效率，完成用户转化率的提高

- 社会影响力：通过我们的 PR 团队做一些对外的分析报告，扩大行业影响力

2. 未来规划实施



针对对未来规划的思考，我们具体实施措施计划是：与集团基础平台工具共享共建，在数据应用方面更好做到数据赋能业务，然后就是具体的数据建设、数据管理。

数据建设：

数据建设主要围绕以下几个方面：

- 数据全：我们希望我们的数据足够全，包括外卖的数据以及团购、点评的线下数据和外部采买的数据等，只要是外卖需要的数据我们都尽量采集过来

- 效率高：效率提升方面，我们刚刚提到我们的使用建模工具替代人工工作从而提高我们的效率
- 能力强：在足够全的数据、提升效率的基础上提高我们的能力，包括服务的稳定性、数据质量

数据管理：

通过完善数据标准规范，并将规范落地到工具以及增强数据治理，另外通过算法的手段发现数据里隐藏的问题完成数据数据治理。这主要需要我们组织能力建设、标准规范的统一、完善数据治理平台与数据运营机制、探索智能数据治理，最终达到数据管理的规范化、系统化、智能化的目的。

二、OneData 建设探索之路：SaaS 收银运营数仓建设

背景

随着业务的发展，频繁迭代和跨部门的垂直业务单元变得越来越多。但由于缺乏前期规划，导致后期数仓出现了严重的数据质量问题，这给数据治理工作带来了很大的挑战。在数据仓库建设过程中，我们总结的问题包括如下几点：

- 缺乏统一的业务和技术标准，如：开发规范、指标口径和交付标准不统一。

- 缺乏有效统一的数据质量监控，如：列值信息不完整和不准确，SLA 时效无法保障等。
- 业务知识体系散乱不集中，导致不同研发人员对业务理解存在较大的偏差，造成产品的开发成本显著增加。
- 数据架构不合理，主要体现在数据层之间的分工不明显，缺乏一致的基础数据层，缺失统一维度和指标管理。

目标

在现有大数据平台的基础上，借鉴业界成熟 OneData 方法论，构建合理的数据体系架构、数据规范、模型标准和开发模式，以保障数据快速支撑不断变化的业务并驱动业务的发展，最终形成我们自己的 OneData 理论体系与实践体系。

OneData 探索

OneData：行业经验

在数据建设方面，阿里巴巴提出了一种 OneData 标准，如图-1 所示：

图 1 OneData 标准

OneData：我们的思考

他山之石，可以攻玉。我们结合实际情况和业界经验，进行了如下思考：

1.对阿里巴巴 OneData 的思考

- 整个 OneData 体系覆盖范围广，包含数据规范定义体系、数据模型规范设计、ETL 规范研发以及支撑整个体系从方法到实施的工具体系。

- 实施周期较长，人力投入成本较高。
- 推广落地的工作比较依赖工具。

2.对现有实际的思考

- 现阶段工具保障方面偏弱，人力比较缺乏。
- 现有开发流程不能全部推翻。

经过综合考量，我们发现直接全盘复用他人经验是不合理的。那我们如何定义自己的 OneData，即能在达到目标的前提下，又能避免上述的难题呢？

OneData：我们的想法

首先，结合行业经验，自身阶段的实践及以往的数仓经验，我们预先定义了 OneData 核心思想与 OneData 核心特点。

OneData 核心思想：从设计、开发、部署和使用层面，避免重复建设和指标冗余建设，从而保障数据口径的规范和统一，最终实现数据资产全链路关联、提供标准数据输出以及建立统一的数据公共层。

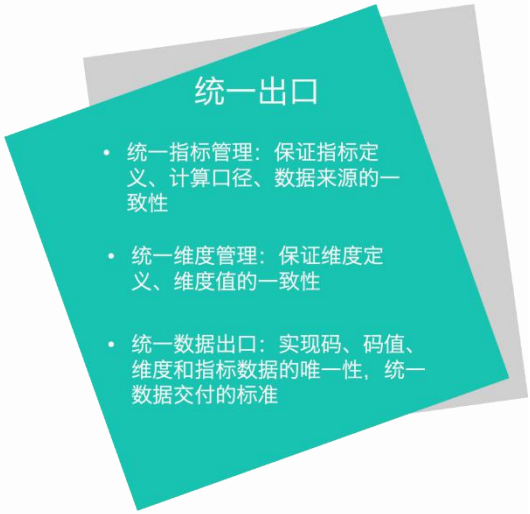
OneData 核心特点：三特性和三效果。

- 三特性：统一性、唯一性、规范性。
- 三效果：高扩展性、强复用性、低成本性。

图 2 OneData 的六个特性

OneData：我们的策略

OneData 即有核心思想又有核心特点，到底怎么来实现核心思想又能满足其核心特点呢？通过以往经验的沉淀，我们提出两个统一方法策略：统一归口、统一出口。



根据两个统一的方法策略，我们开启了 OneData 的实践之路。

OneData 实践

统一业务归口

数据来源于业务并支撑着业务的发展。因此，数仓建设的基石就是对于业务的把控，数仓建设者即是技术专家，也应该是“大半个”业务专家。基于互联网行业的特点，我们基本上采用需求推动数据的建设，这也带来了一些问题，包括：数据对业务存在一定的滞后性；业务知识沉淀在各个需求里，导致业务知识体系分散。针对这些问题，我们提出统一业务归口，构建全局知识库，进而保障对业务认知的一致性。

图 3 支持业务的数据源知识库

设计统一归口

为了解决数据仓库建设过程中出现的各种痛点，我们从模型与规范两个方面进行建设，并提出设计统一归口。

1.模型

规范化模型分层、数据流向和主题划分，从而降低研发成本，增强指标复用性，并提高业务的支撑能力。

(1) 模型分层

优秀可靠的数仓体系，往往需要清晰的数据分层结构，即要保证数据层的稳定又要屏蔽对下游的影响，并且要避免链路过长。结合这些原则及以往的工作经验，我们将分层进行统一定义为四层：

图 4 数据分层架构

(2) 模型数据流向

重构前，存在大量的烟囱式开发、分层应引用不规范性及数据链路混乱、血缘关系很难追溯和 SLA 时效难保障等问题。

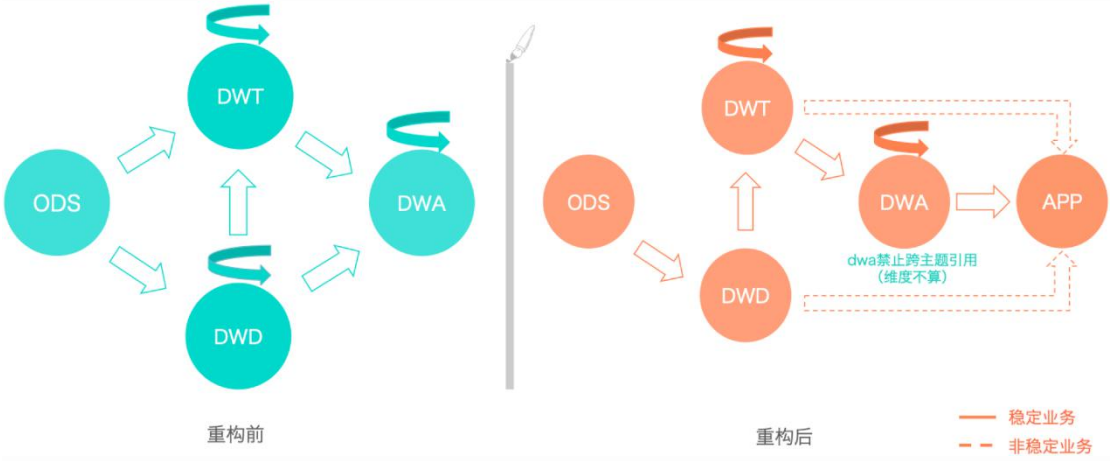


图 5 重构前和重构后的数据流向图

重构之后，稳定业务按照标准的数据流向进行开发，即 ODS–>DWD–>DWA–>APP。非稳定业务或探索性需求，可以遵循 ODS->DWD->APP 或者 ODS->DWD->DWT->APP 两个模型数据流。在保障了数据链路的合理性之后，又在此基础上确认了模型分层引用原则：

- 正常流向：ODS>DWD->DWT->DWA->APP，当出现 ODS >DWD->DWA->APP 这种关系时，说明主题域未覆盖全。应将 DWD 数据落到 DWT 中，对于使用频度非常低的表允许 DWD->DWA。
- 尽量避免出现 DWA 宽表中使用 DWD 又使用（该 DWD 所归属主题域）DWT 的表。
- 同一主题域内对于 DWT 生成 DWT 的表，原则上要尽量避免，否则会影响 ETL 的效率。
- DWT、DWA 和 APP 中禁止直接使用 ODS 的表， ODS 的表只能被 DWD 引用。
- 禁止出现反向依赖，例如 DWT 的表依赖 DWA 的表。

2.主题划分

传统行业如银行、制造业、电信、零售等行业中，都有比较成熟的主题划分，如 BDWM、FS-LDM、MLDM 等等。但从实际调研情况来看，主题划分太抽象会

造成对业务理解和开发成本较高，不适用互联网行业。因此，结合各层的特性，我们提出了两类主题的划分：**面向业务**、**面向分析**。

- 面向业务：按照业务进行聚焦，降低对业务理解的难度，并能解耦复杂的业务。我们将实体关系模型进行变种处理为实体与业务过程模型。实体定义为业务过程的参与体；业务过程定义是由多个实体作用的结果，实体与业务过程都带有自己特有的属性。根据业务的聚合性，我们把业务进行拆分，形成了七大核心主题。
- 面向分析：按照分析聚焦，提升数据易用性，提高数据的共享与一致性。按照分析主体对象不同及分析特征，形成分析域主题在 DWA 进行应用，例如销售分析域、组织分析域。

3.规范

模型是整个数仓建设基石，规范是数仓建设的保障。为了避免出现指标重复建设和数据质量差的情况，我们统一按照最详细、可落地的方法进行规范建设。

(1) 词根

词根是维度和指标管理的基础，划分为普通词根与专有词根，提高词根的易用性和关联性。

- 普通词根：描述事物的最小单元体，如：交易-trade。
- 专有词根：具备约定成俗或行业专属的描述体，如：美元-USD。

(2) 表命名规范

通用规范

- 表名、字段名采用一个下划线分隔词根（示例：clienttype->client_type）。
- 每部分使用小写英文单词，属于通用字段的必须满足通用字段信息的定义。
- 表名、字段名需以字母为开头。
- 表名、字段名最长不超过 64 个英文字符。
- 优先使用词根中已有关键字（数仓标准配置中的词根管理），定期 Review 新增命名的不合理性。
- 在表名自定义部分禁止采用非标准的缩写。

表命名规则

表名称 = 类型 + 业务主题 + 子主题 + 表含义 + 存储格式 + 更新频率 +结尾，如下图所示：

图 6 统一的表命名规范

(3) 指标命名规范

结合指标的特性以及词根管理规范，将指标进行结构化处理。

A. 基础指标词根，即所有指标必须包含以下基础词根：

基础指标词根	英文全称	Hive 数据类型	MySQL 数据类型	长度	精度	词根	样例
数量	count	Bigint	Bigint	10	0	cnt	
金额类	amout	Decimal	Decimal	20	4	amt	
比率/占比	ratio	Decimal	Decimal	10	4	ratio	0.9818
.....	.....	.....	.....			.....	

B.业务修饰词，用于描述业务场景的词汇，例如 trade-交易。

C.日期修饰词，用于修饰业务发生的时间区间。

日期类型	全称	词根	备注
日	daily	d	

日期类型	全称	词根	备注
周	weekly	w	
.....	.....	.....	

D.聚合修饰词，对结果进行聚集操作。

聚合类型	全称	词根	备注
平均	average	avg	
周累计	wtd	wtd	本周一截止到当天累计
.....	.....	.....	

E.基础指标，单一的业务修饰词+基础指标词根构建基础指标 ，例如：交易金额-trade_amt。

F.派生指标，多修饰词+基础指标词根构建派生指标。派生指标继承基础指标的特性，例如：安装门店数量-install_poi_cnt。

G.普通指标命名规范，与字段命名规范一致，由词汇转换即可以。

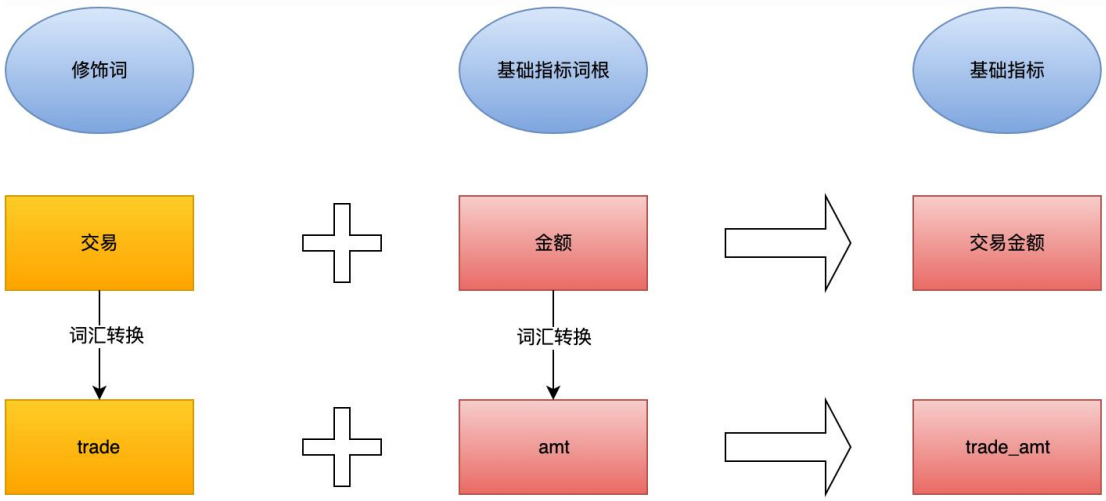


图 7 普通指标规范

H.日期类型指标命名规范，命名时要遵循：业务修饰词+基础指标词根+日期修饰词/聚合修饰词。将日期后缀加到名称后面，如下图所示：

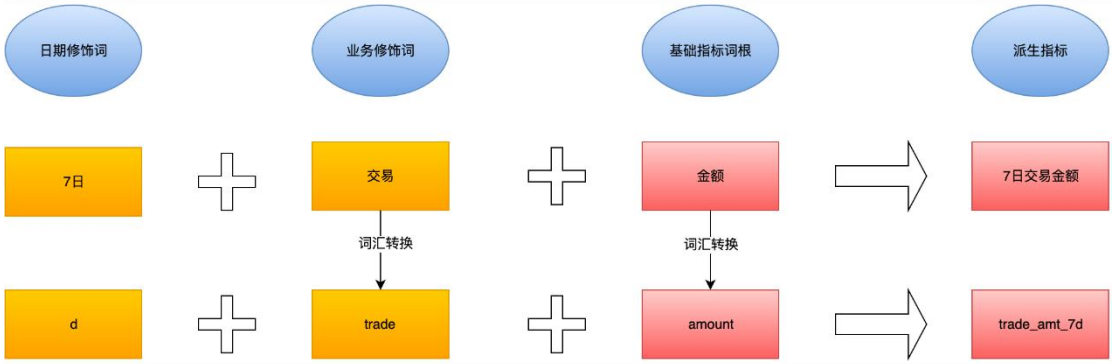


图 8 日期类型指标规范

I.聚合类型指标，命名时要遵循：业务修饰词+基础指标词根+聚合类型+日期修饰词。将累积标记加到名称后面，如下图所示：

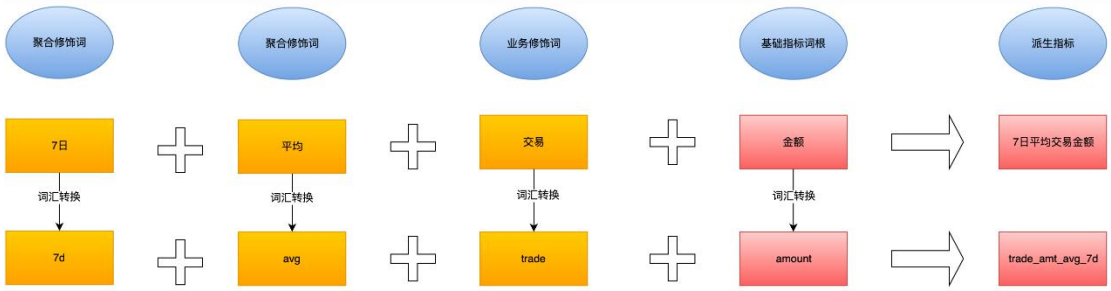


图 9 聚合类指标规范

(4) 清洗规范

确认了字段命名和指标命名之后，根据指标与字段的部分特性，我们整理出了整个数仓可预知的 24 条清洗规范：

数据类型	数据类别	Hive类型	MySQL类型	长度	精度	词根	格式说明	备注
日期类型	字符日期类	string	varchar	10		date	YYYY-MM-DD	日期清洗为相应的格式
数据类型	数量类	bigint	bigint	10	0	cnt	活跃门店数量	
.....	.....	.....	.....	.....	.....	.....	.....	

结合模型与规范，形成模型设计及模型评审两者的工作职责，如下图所示：

图 10 模型设计和审计职责

统一应用归口

在对原有的应用支持流程进行梳理的时候，我们发现整个研发过程是烟囱式。如果不进行改善就会导致前面的建设”毁于一旦“，所以需对原有应用支持流程进行改造，如下图所示：

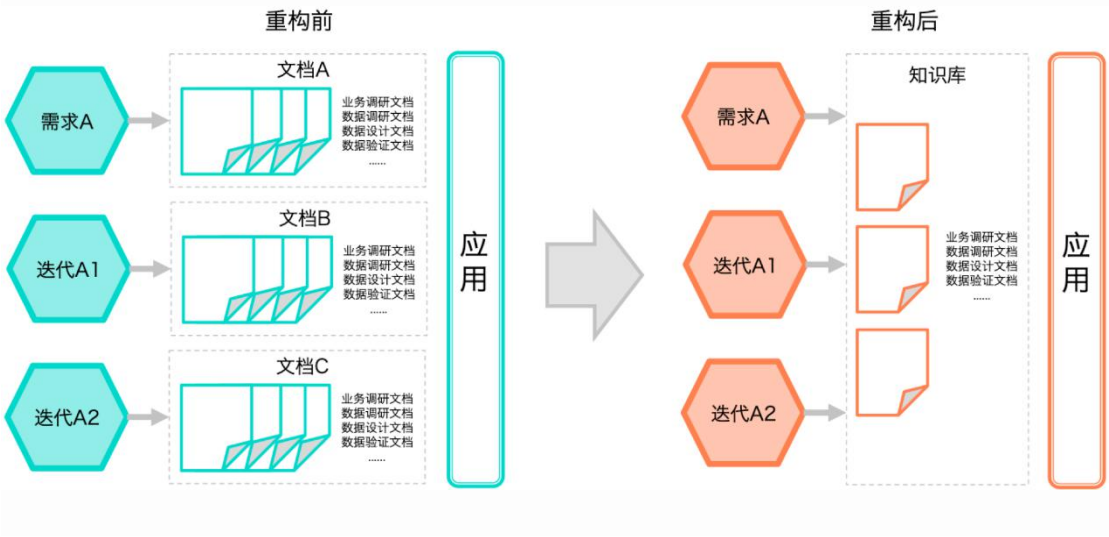


图 11 应用归口

从图中可以看出，重构前一个应用存在多次迭代，每次迭代都各自维护自己的文档。烟囱式开发会导致业务信息混乱、应用无法与文档对齐、知识传递成本、维护成本和迭代成本大大增加等问题。重构后，应用与知识库相对应，保证一个应用只对应一份文档，且应用统一要求在一份文档上进行迭代，从根源上改变应用支持流程。同时，针对核心业务细节和所支撑的数据信息，进行了全局调研并归纳到知识库。综合统一的知识库，降低了知识传递、理解、维护和迭代成本。

统一归口策略包含业务归口统一、设计归口统一和应用归口统一，从底层保证了数仓建设的**三特性**和**三效果**。

统一数据出口

数仓建设不仅仅是为了数据内容而建设，同时也为了提高交付的数据质量与数据使用的便利性。如何保证数据质量以及推广数据的使用，我们提出了统一数据出口策略。在进行数据资产管理和统一数据出口之前，必须高质量地保障输出的数据质量，从而树立 OneData 数据服务体系的权威性。

1.交付标准化

如何保证数据质量，满足什么样的数据才是可交付的，是数据建设者一直探索的问题。为了保证交付的严谨性，在具体化测试方案之前，我们结合数仓的特点明确了数据交付标准的五个特性，如下图所示：

图 12 交付标准化

《交付标准化》完善了整个交付细节，从根本上保证了数据的质量，如：业务测试保障数据的合理性、一致性；技术测试保障数据的唯一性、准确性；数据平台的稳定性和后期人工维护保障数据的及时性。

2.数据资产管理

针对如何解决数据质量中维度与指标一致性以及如何提高数据易用性的问题，我们提出数据资产的概念，借助公司内部平台工具“起源数据平台”实现了整个数据资产管理，它的功能如下图所示：

图 13 起源功能体系

借用起源数据平台，我们实现了：

- 统一指标管理，保证了指标定义、计算口径、数据来源的一致性。
- 统一维度管理，保证了维度定义、维度值的一致性。
- 统一数据出口，实现了维度和指标元数据信息的唯一出口，维值和指标数据的唯一出口。

通过交付标准化和数据资产管理，保证了数据质量与数据的易用性，同时我们也构建出 OneData 数据体系中数据指标管理的核心。

实践的成果

流程改善

我们对开发过程进行梳理，服务于整个 OneData 体系。对需求分析、指标管理、模型设计、数据验证进行了改善，并结合 OneData 模型策略，改善了数仓管理流程。

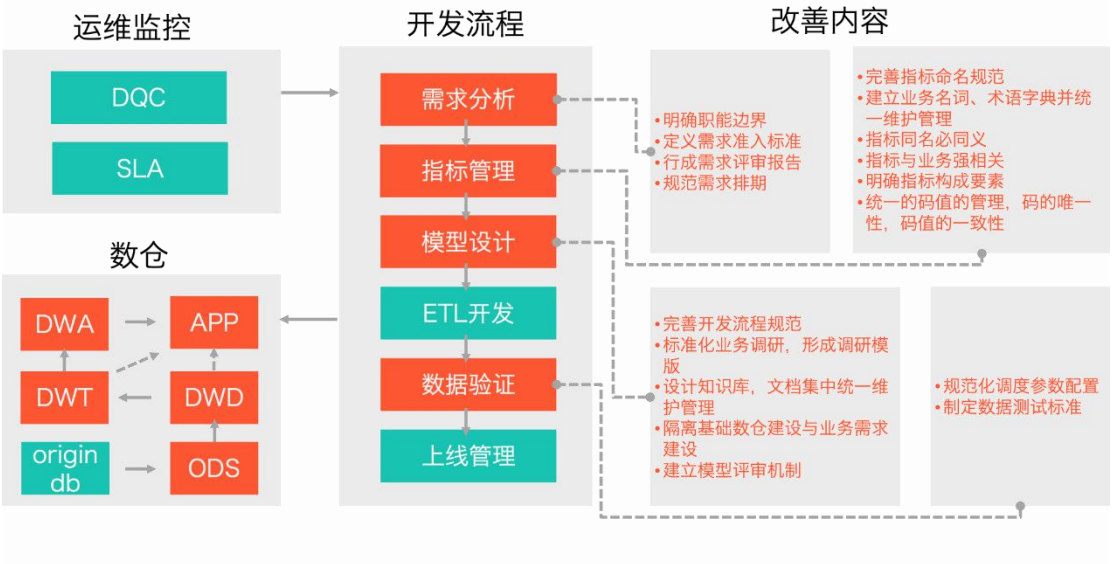


图 14 数仓管理流程

数仓全景图

基于 OneData 主题建设，我们采用**面向业务**、**面向分析**的建设策略，形成数仓全景图，如下图所示：

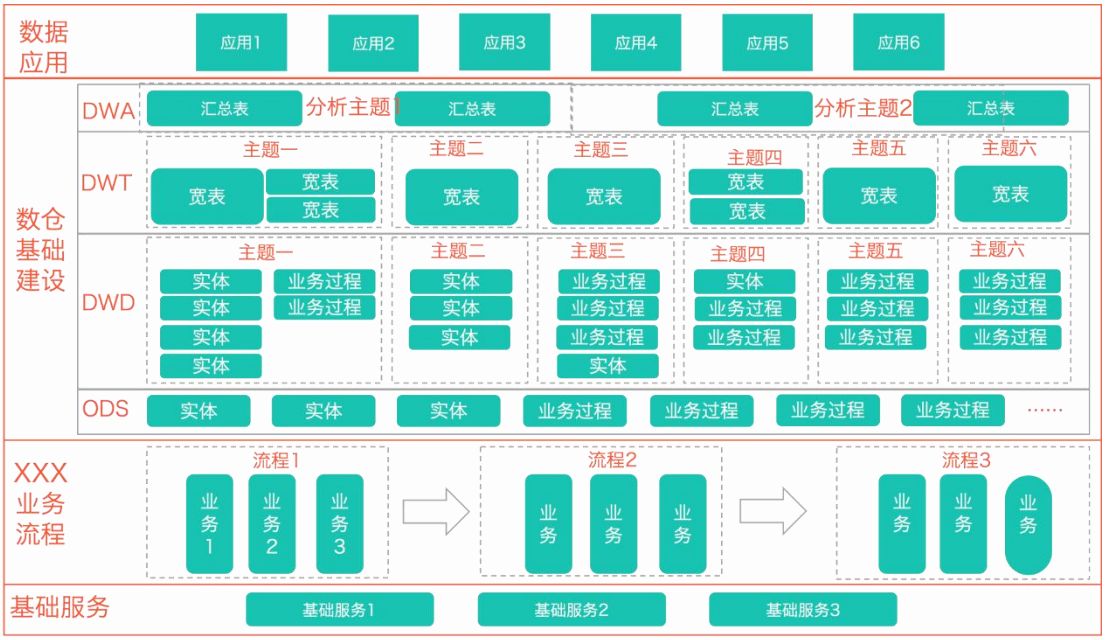


图 15 数仓全景图

资产管理列表

基于起源数据平台形成的资产管理体系，如下图所示：

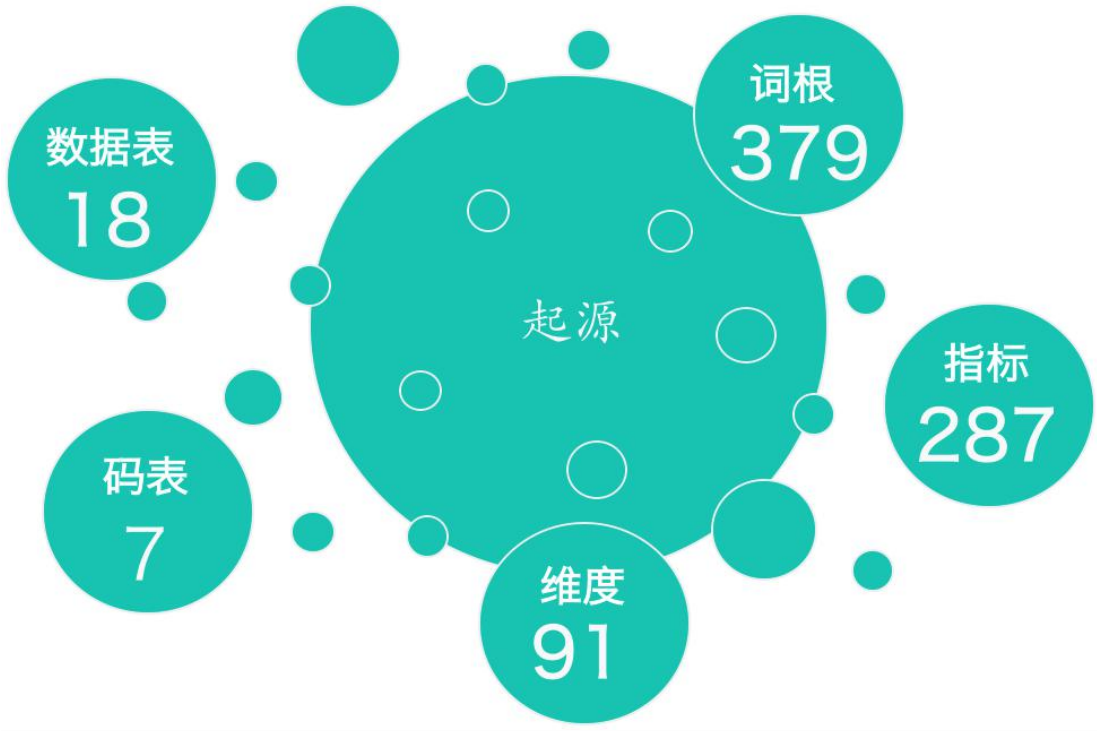


图 16 数据资产管理

项目收益

基于 OneData 建设成果，我们结合实际项目建设样例，对比以前未进行 OneData 建设时的收益。如下图所示：

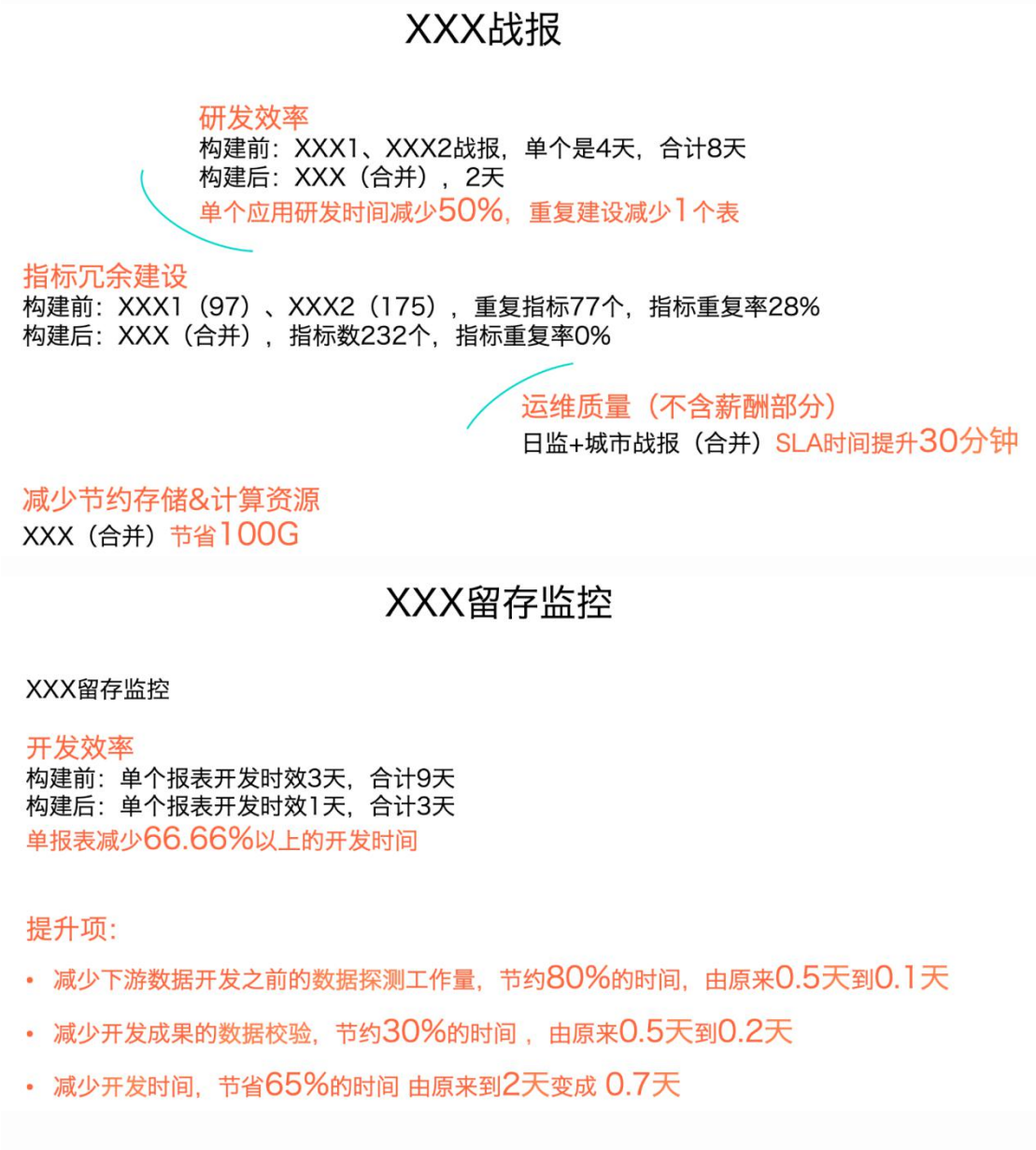


图 17 价值收益

总结和展望

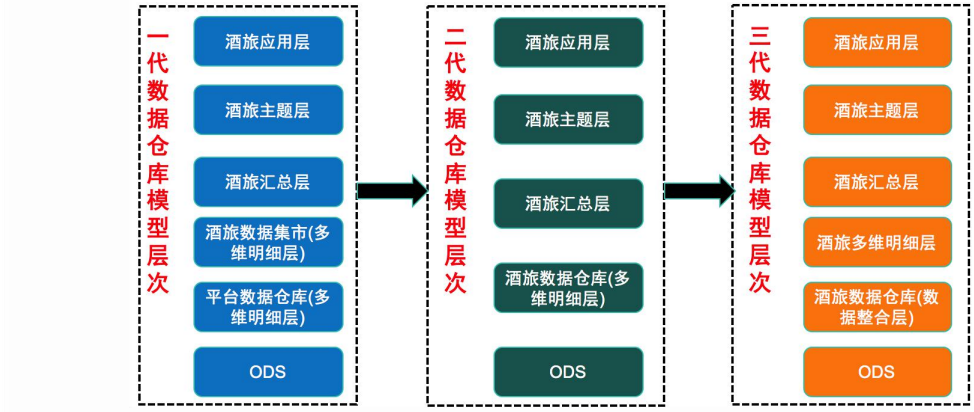
我们结合了 OneData 核心思想与特点，构建一种稳定、可靠的基础数据仓库，从底层保障了数据质量，同时完成 OneData 实践，形成自有的 OneData 理论体系。未来，我们还将在技术上引入实时数据仓库，满足灵活多样、低延时的数据需求；在业务层面会横向拓展其他业务领域，不间断地支撑核心业务的决策与分析。下一步，我们将为企业级 One Entity 数据中台（以 Data As a Service 为理念），提供强有力的数据支撑。在后续数仓维护过程中，不断地发现问题、解决问题和总结问题，保障数据稳定性、一致性和有效性，为核心业务构建价值链，最终形成企业级的数据资产。

三、美团点评酒旅数据仓库建设实践

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。

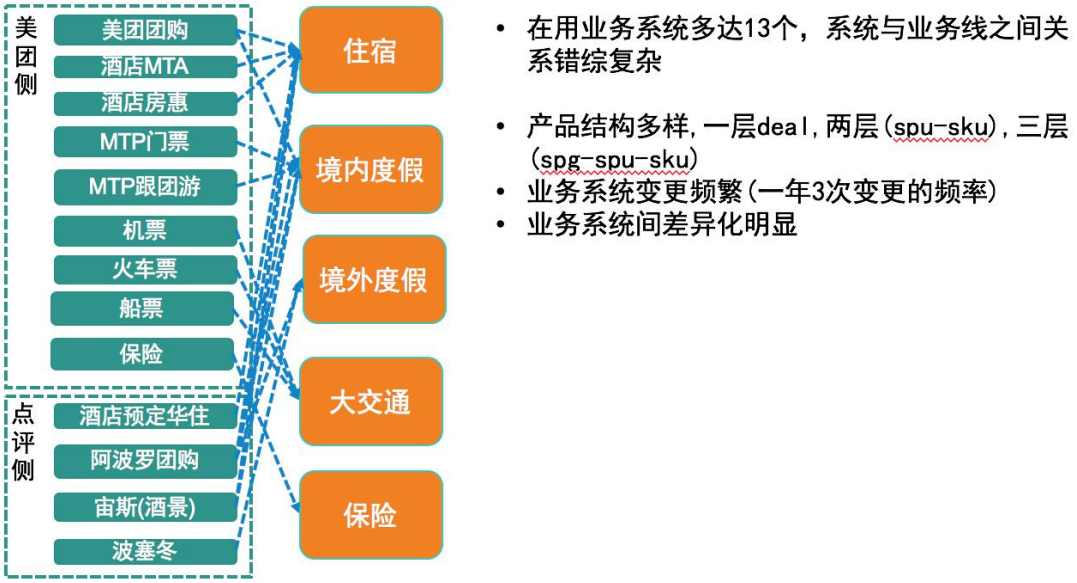


Hotel dw layer

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。

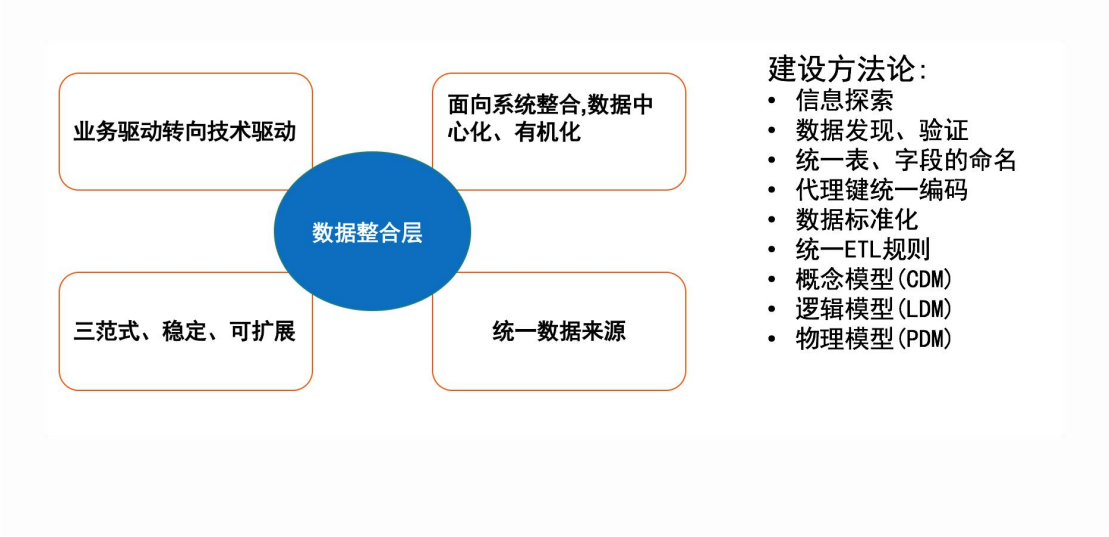
但随着美团原本集中的业务系统不能快速响应各个业务线迅速的发展与业务变化时，酒旅中的酒店业务线开始有了自己的业务系统来支持预订、房惠、团购、直连等产品形式，境内度假业务线也开始有了自己的业务系统来支持门票预订、门票直连、跟团游等复杂业务。我们开始了第二代数仓模型层次的建设，由建设数据集市的形式转变成了直接建设酒旅数据仓库，成为了酒旅自身业务系统数据的唯一加工者。由于系统调整初期给我们带来的重构、修改以及新增等数据处理工作非常大，我们采用了比较短平快的 Kimball 所提的维度建模的方式建设了酒旅数据仓库。

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：



Hotel dw baslayer background

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：

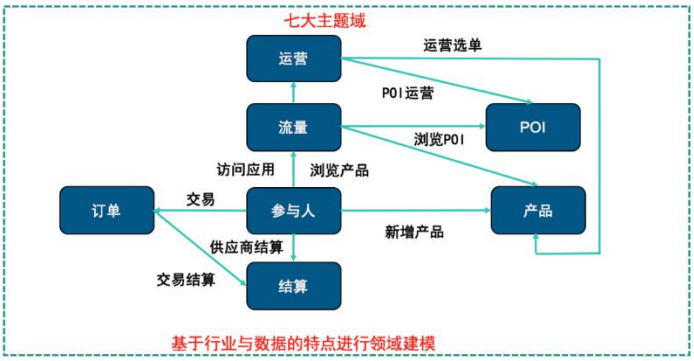


baslayer adventage

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）：



dw topic

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

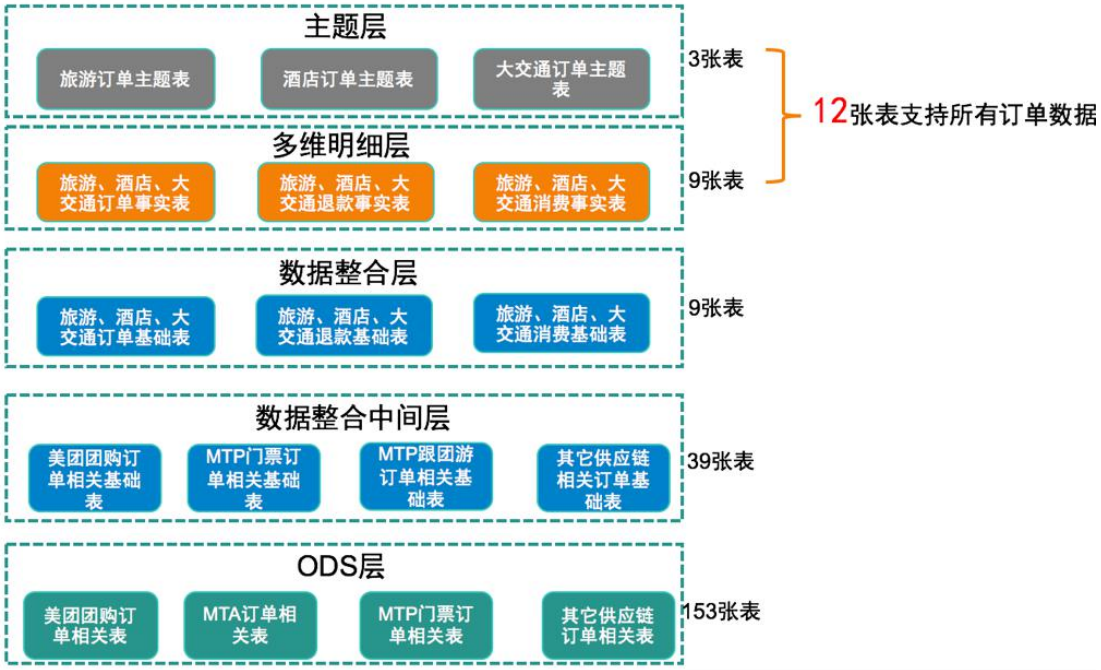
实际上，这个主题在传统企业里面如银行、电信等等都是至关重要的，对我们酒旅而言，建设它的意义能够不断优化商家体验、提高财务结算与管理能力。

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。

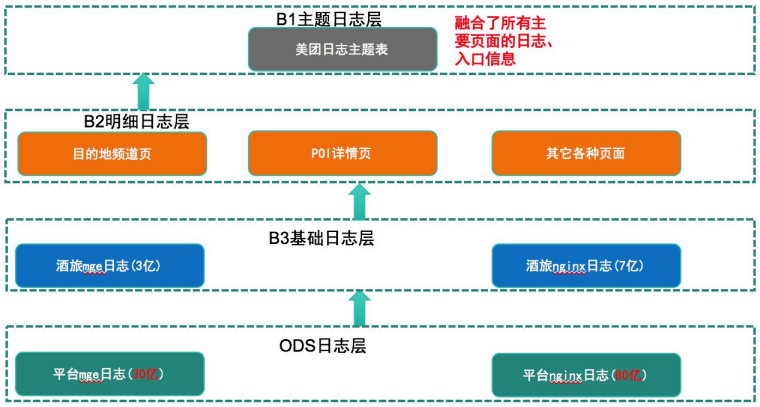


dw architecture

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

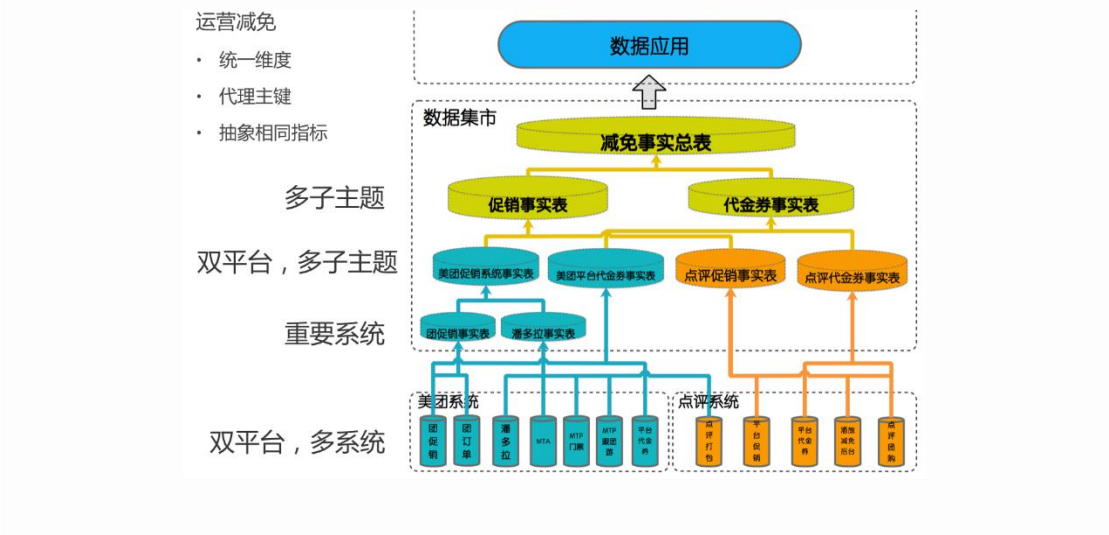
个页面进行合并生成总的日志主题表（最终这张表会满足 80%以上的相关流量统计需求）。



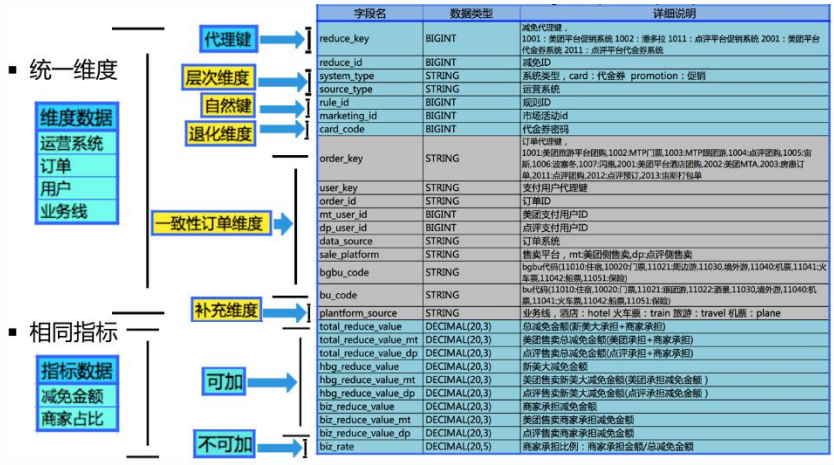
dw architecture

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。



dw architecture



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:雷斯,1006:波塞冬,1007:闪购,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:雷斯打包单
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

dw architecture

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。

实时数仓

四、美团外卖实时数仓建设实践

实时数仓以端到端低延迟、SQL 标准化、快速响应变化、数据统一为目标。美团外卖数据智能组总结的最佳实践是：一个通用的实时生产平台跟一个通用交互式实时分析引擎相互配合，同时满足实时和准实时业务场景。两者合理分工，互相补充，形成易开发、易维护且效率高的流水线，兼顾开发效率与生产成本，以较好的投入产出比满足业务的多样性需求。

01 实时场景

实时数据在美团外卖的场景是非常多的，主要有以下几个方面：

- **运营层面**：比如实时业务变化，实时营销效果，当日营业情况以及当日分时业务趋势分析等。
- **生产层面**：比如实时系统是否可靠，系统是否稳定，实时监控系统的健康状况等。
- **C 端用户**：比如搜索推荐排序，需要实时行为、特点等特征变量的生产，给用户推荐更加合理的内容。
- **风控侧**：实时风险识别、反欺诈、异常交易等，都是大量应用实时数据的场景。

02 实时技术及架构

1. 实时计算技术选型

目前，市面上已经开源的实时技术还是很多的，比较通用的有 Storm、Spark Streaming 以及 Flink，技术同学在做选型时要根据公司的具体业务来进行部署。

美团外卖依托于美团整体的基础数据体系建设，从技术成熟度来讲，公司前几年主要用的是 Storm。当时的 Storm，在性能稳定性、可靠性以及扩展性上也是无可替代的。但随着 Flink 越来越成熟，从技术性能上以及框架设计优势上已经

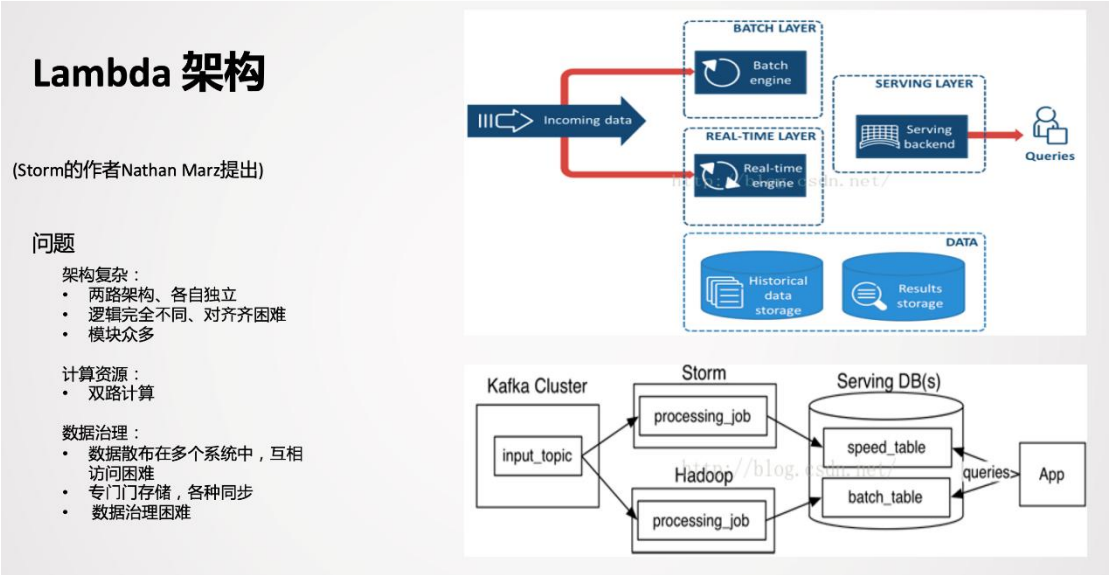
超越了 Storm，从趋势来讲就像 Spark 替代 MR 一样，Storm 也会慢慢被 Flink 替代。当然，从 Storm 迁移到 Flink 会有一个过程，我们目前有一些老的任务仍然运行在 Storm 上，也在不断推进任务迁移。

	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。
性能	稳定，可靠。性能上整体不如Flink	吞吐量是storm 3倍+， 时延明显低于storm
应用现状	公司内应用成熟，老业务大多运行在storm平台上	平台在完善升级，推动业务迁移到Flink平台。大厂应用成熟

具体 Storm 和 Flink 的对比可以参考上图表格。

2. 实时架构

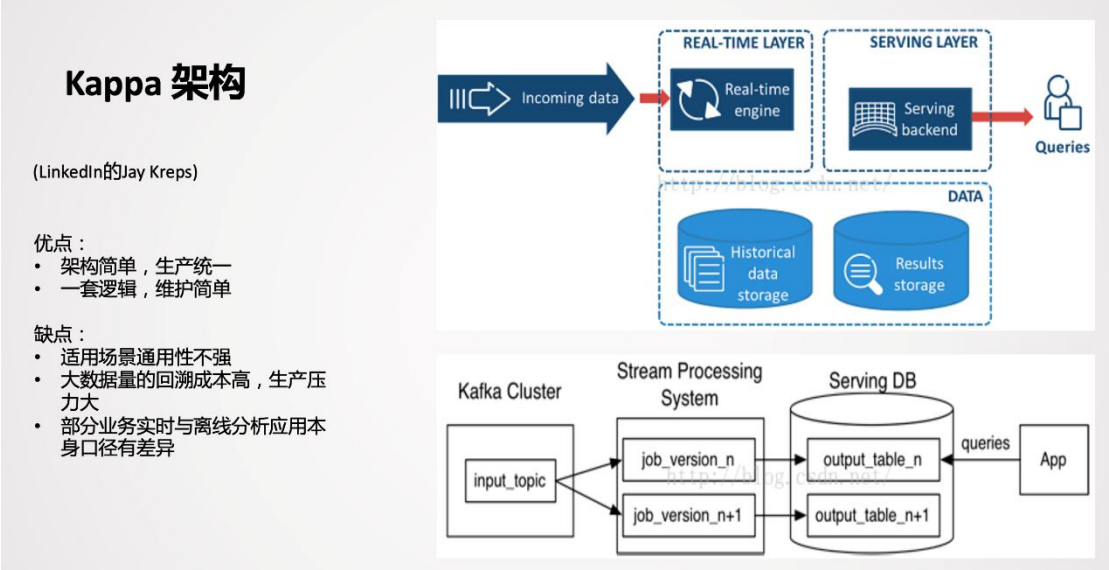
① Lambda 架构



Lambda 是比较经典的一款架构，以前实时的场景不是很多，以离线为主，当附加了实时场景后，由于离线和实时的时效性不同，导致技术生态是不一样的。而 Lambda 架构相当于附加了一条实时生产链路，在应用层面进行一个整合，双路生产，各自独立。在业务应用中，顺理成章成为了一种被采用的方式。

双路生产会存在一些问题，比如加工逻辑 Double，开发运维也会 Double，资源同样会变成两个资源链路。因为存在以上问题，所以又演进了一个 Kappa 架构。

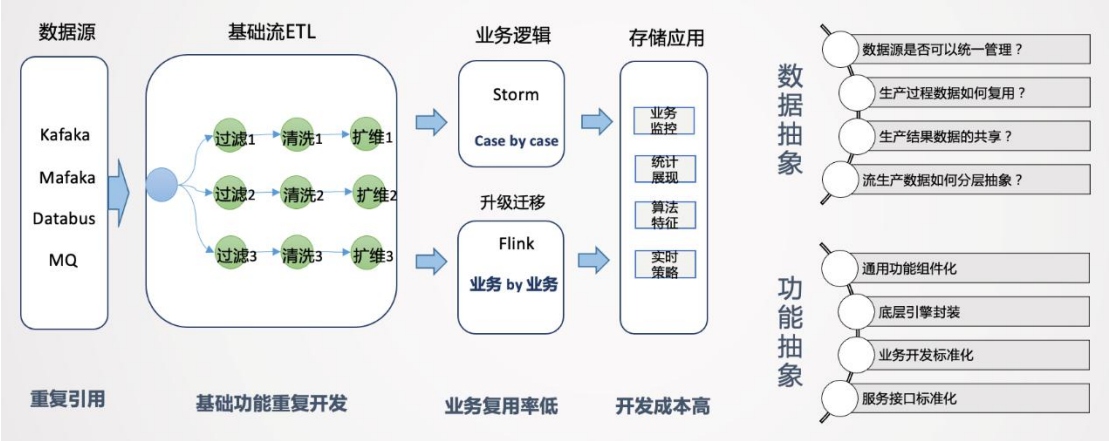
② Kappa 架构



Kappa 从架构设计来讲，比较简单，生产统一，一套逻辑同时生产离线和实时。但是在实际应用场景有比较大的局限性，在业内直接用 Kappa 架构生产落地的案例不多见，且场景比较单一。这些问题在美团外卖这边同样会遇到，我们也会有自己的一些思考，将会在后面的章节进行阐述。

03 业务痛点

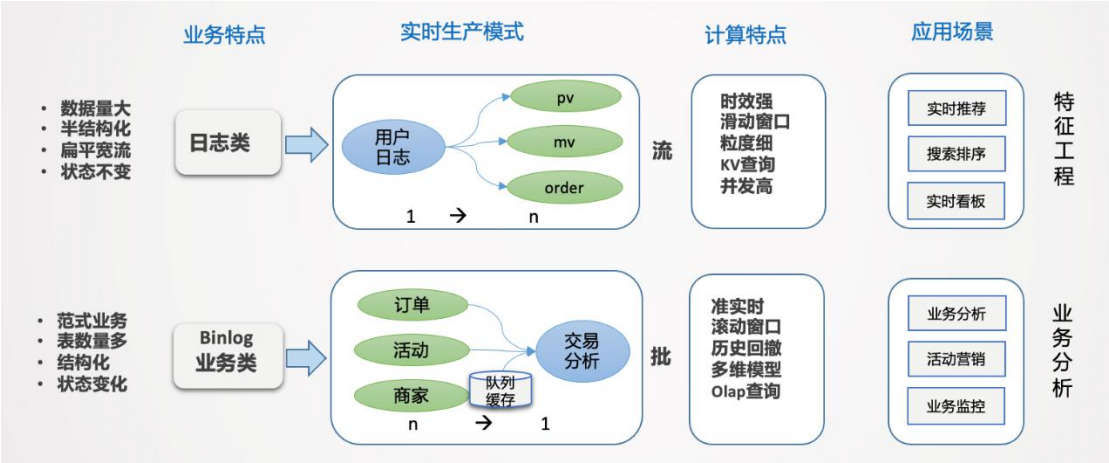
首先，在外卖业务上，我们遇到了一些问题和挑战。在业务早期，为了满足业务需要，一般是 Case By Case 地先把需求完成。业务对于实时性要求是比较高的，从时效性的维度来说，没有进行中间层沉淀的机会。在这种场景下，一般是拿到业务逻辑直接嵌入，这是能想到的简单有效的方法，在业务发展初期这种开发模式也比较常见。



如上图所示，拿到数据源后，我们会经过数据清洗、扩维，通过 Storm 或 Flink 进行业务逻辑处理，最后直接进行业务输出。把这个环节拆开来看，数据源端会重复引用相同的数据源，后面进行清洗、过滤、扩维等操作，都要重复做一遍。唯一不同的是业务的代码逻辑是不一样的，如果业务较少，这种模式还可以接受，但当后续业务量上去后，会出现谁开发谁运维的情况，维护工作量会越来越大，作业无法形成统一管理。而且所有人都在申请资源，导致资源成本急速膨胀，资源不能集约有效利用，因此要思考如何从整体来进行实时数据的建设。

04 数据特点与应用场景

那么如何来构建实时数仓呢？首先要进行拆解，有哪些数据，有哪些场景，这些场景有哪些共同特点，对于外卖场景来说一共有两大类，日志类和业务类。



- **日志类**：数据量特别大，半结构化，嵌套比较深。日志类的数据有个很大的特点，日志流一旦形成是不会变的，通过埋点的方式收集平台所有的日志，统一进行采集分发，就像一颗树，树根非常大，推到前端应用的时候，相当于从树根到树枝分叉的过程（从 1 到 n 的分解过程）。如果所有的业务都从根上找数据，看起来路径最短，但包袱太重，数据检索效率低。日志类数据一般用于生产监控和用户行为分析，时效性要求比较高，时间窗口一般是 5min 或 10min，或截止到当前的一个状态，主要的应用是实时大屏和实时特征，例如用户每一次点击行为都能够立刻感知到等需求。
- **业务类**：主要是业务交易数据，业务系统一般是自成体系的，以 Binlog 日志的形式往下分发，业务系统都是事务型的，主要采用范式建模方式。特点是结构化，主体非常清晰，但数据表较多，需要多表关联才能表达完整业务，因此是一个 n 到 1 的集成加工过程。

而业务类实时处理，主要面临的以下几个难点：

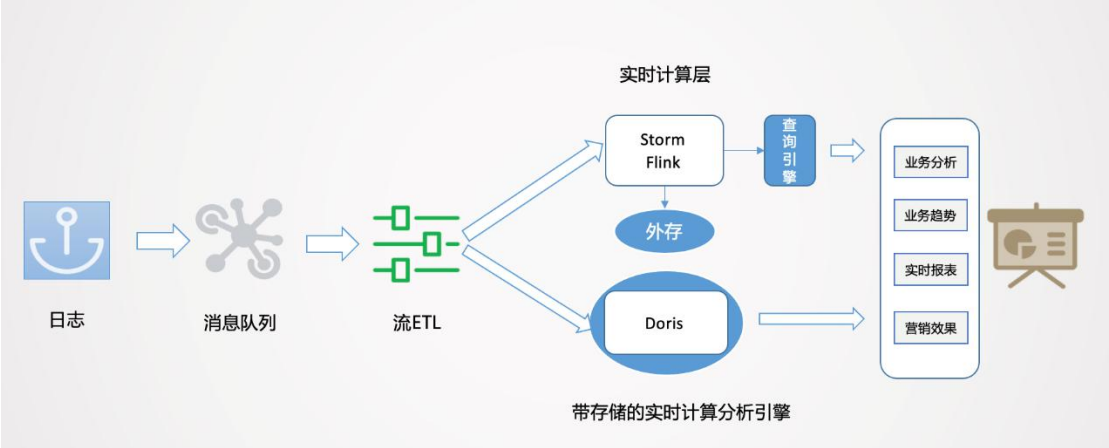
- **业务的多状态性**：业务过程从开始到结束是不断变化的，比如从下单->支付->配送，业务库是在原始基础上进行变更的，Binlog 会产生很多变化的日志。而业务分析更加关注最终状态，由此产生数据回撤计算的问题，例如 10 点下单，13 点取消，但希望在 10 点减掉取消单。
- **业务集成**：业务分析数据一般无法通过单一主体表达，往往是很多表进行关联，才能得到想要的信息，在实时流中进行数据的合流对齐，往往需要较大的缓存处理且复杂。
- **分析是批量的，处理过程是流式的**：对单一数据，无法形成分析，因此分析对象一定是批量的，而数据加工是逐条的。

日志类和业务类的场景一般是同时存在的，交织在一起，无论是 Lambda 架构还是 Kappa 架构，单一的应用都会有一些问题。因此针对场景来选择架构与实践才更有意义。

05 实时数仓架构设计

1. 实时架构：流批结合的探索

基于以上问题，我们有自己的思考。通过流批结合的方式来应对不同的业务场景。



如上图所示，数据从日志统一采集到消息队列，再到数据流的 ETL 过程，作为基础数据流的建设是统一的。之后对于日志类实时特征，实时大屏类应用走实时流计算。对于 Binlog 类业务分析走实时 OLAP 批处理。

流式处理分析业务的痛点是什么？对于范式业务，Storm 和 Flink 都需要很大的外存，来实现数据流之间的业务对齐，需要大量的计算资源。且由于外存的限制，必须进行窗口的限定策略，最终可能放弃一些数据。计算之后，一般是存到 Redis 里做查询支撑，且 KV 存储在应对分析类查询场景中也有较多局限。

实时 OLAP 怎么实现？有没有一种自带存储的实时计算引擎，当实时数据来了之后，可以灵活的在一定范围内自由计算，并且有一定的数据承载能力，同时支持分析查询响应呢？随着技术的发展，目前 MPP 引擎发展非常迅速，性能也在飞快提升，所以在这种场景下就有了一种新的可能。这里我们使用的是 Doris 引擎。

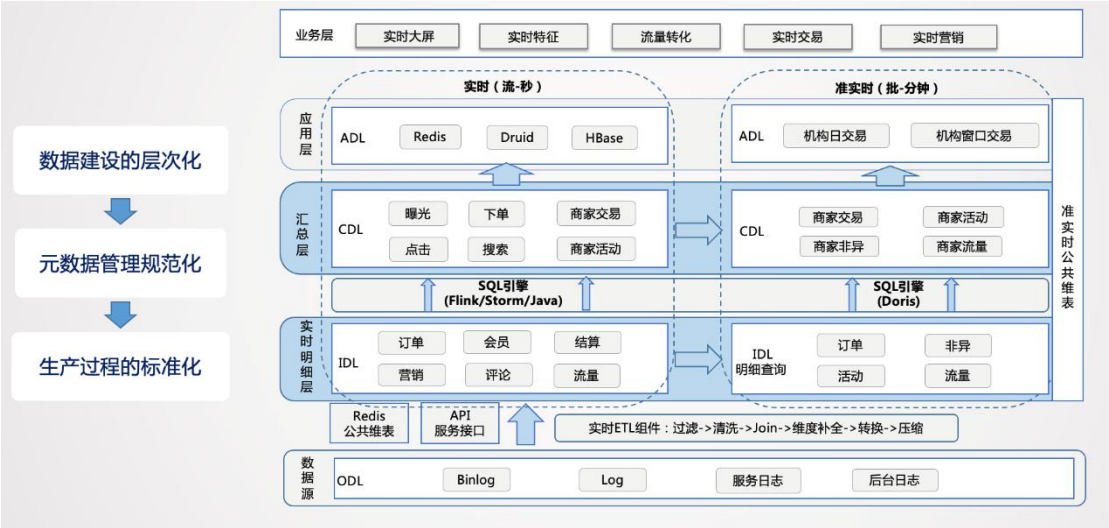
这种想法在业内也已经有实践，且成为一个重要探索方向。阿里基于 ADB 的实时 OLAP 方案等。

2. 实时数仓架构设计

从整个实时数仓架构来看，首先考虑的是如何管理所有的实时数据，资源如何有效整合，数据如何进行建设。

从方法论来讲，实时和离线是非常相似的。离线数仓早期的时候也是 Case By Case，当数据规模涨到一定量的时候才会考虑如何治理。分层是一种非常有效的数据治理方式，所以在实时数仓如何进行管理的问题上，首先考虑的也是分层的处理逻辑，具体内容如下：

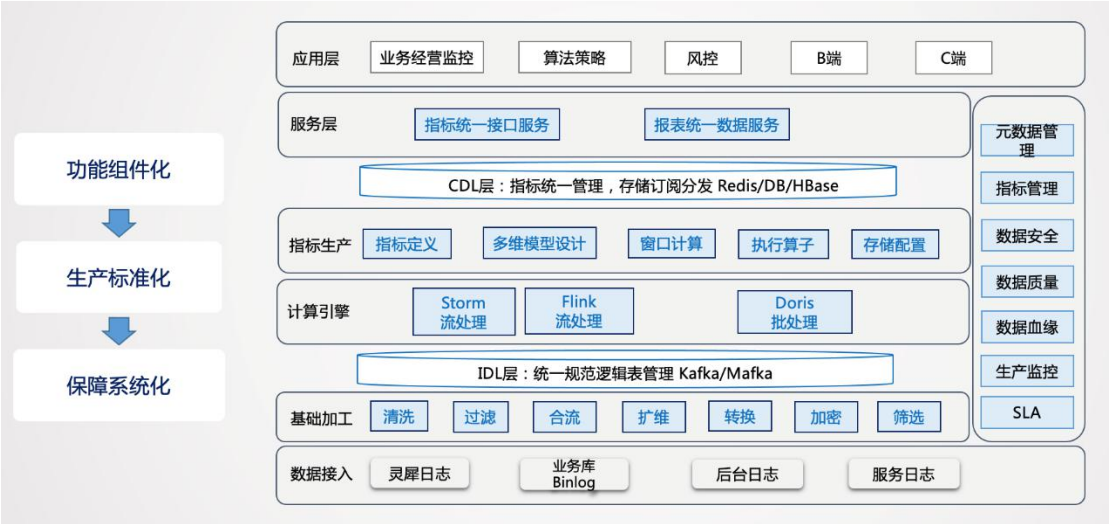
- **数据源**：在数据源的层面，离线和实时在数据源是一致的，主要分为日志类和业务类，日志类又包括用户日志、DB 日志以及服务器日志等。
- **实时明细层**：在明细层，为了解决重复建设的问题，要进行统一构建，利用离线数仓的模式，建设统一的基础明细数据层，按照主题进行管理，明细层的目的是给下游提供直接可用的数据，因此要对基础层进行统一的加工，比如清洗、过滤、扩维等。
- **汇总层**：汇总层通过 Flink 或 Storm 的简洁算子直接可以算出结果，并且形成汇总指标池，所有的指标都统一在汇总层加工，所有人按照统一的规范管理建设，形成可复用的汇总结果。



总结起来，从整个实时数仓的建设角度来讲，首先数据建设的层次化要先建出来，先搭框架，然后定规范，每一层加工到什么程度，每一层用什么样的方式，当规范定义出来后，便于在生产上进行标准化的加工。由于要保证时效性，设计的时候，层次不能太多，对于实时性要求比较高的场景，基本可以走上图左侧的数据流，对于批量处理的需求，可以从实时明细层导入到实时 OLAP 引擎里，基于 OLAP 引擎自身的计算和查询能力进行快速的回撤计算，如上图右侧的数据流。

06 实时平台化建设

架构确定之后，我们后面考虑的是如何进行平台化的建设，实时平台化建设是完全附加于实时数仓管理之上进行的。



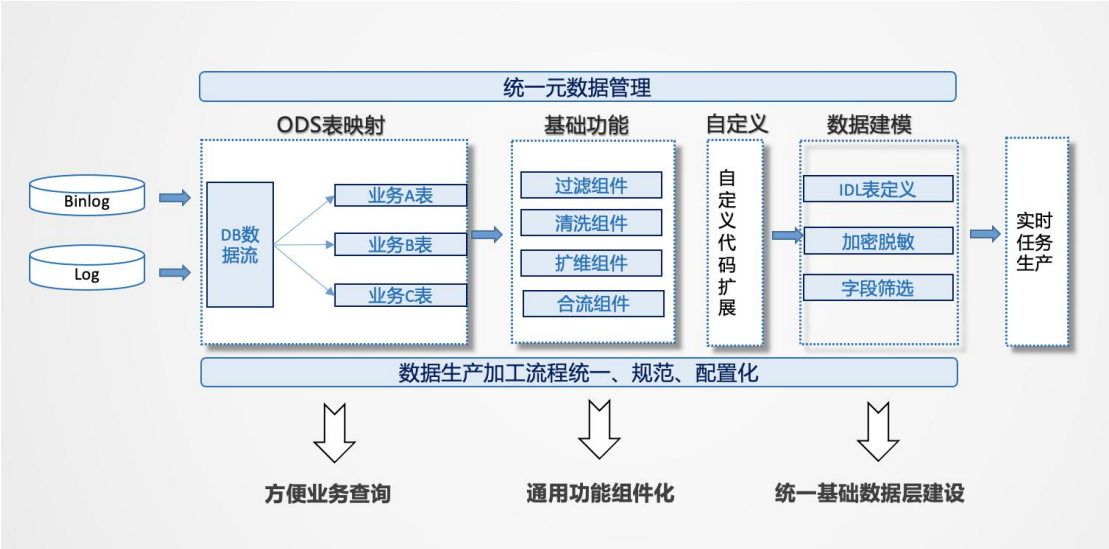
首先进行功能的抽象，把功能抽象成组件，这样就可以达到标准化的生产，系统化的保障就可以更深入的建设，对于基础加工层的清洗、过滤、合流、扩维、转换、加密、筛选等功能都可以抽象出来，基础层通过这种组件化的方式构建直接可用的数据结果流。这会产生一个问题，用户的需求多样，为了满足了这个用户，如何兼容其他的用户，因此可能会出现冗余加工的情况。从存储的维度来讲，实时数据不存历史，不会消耗过多的存储，这种冗余是可以接受的，通过冗余的方式可以提高生产效率，是一种**以空间换时间**思想的应用。

通过基础层的加工，数据全部沉淀到 IDL 层，同时写到 OLAP 引擎的基础层，再往上是实时汇总层计算，基于 Storm、Flink 或 Doris，生产多维度的汇总指标，形成统一的汇总层，进行统一的存储分发。

当这些功能都有了以后，元数据管理，指标管理，数据安全性、SLA、数据质量等系统能力也会逐渐构建起来。

1. 实时基础层功能

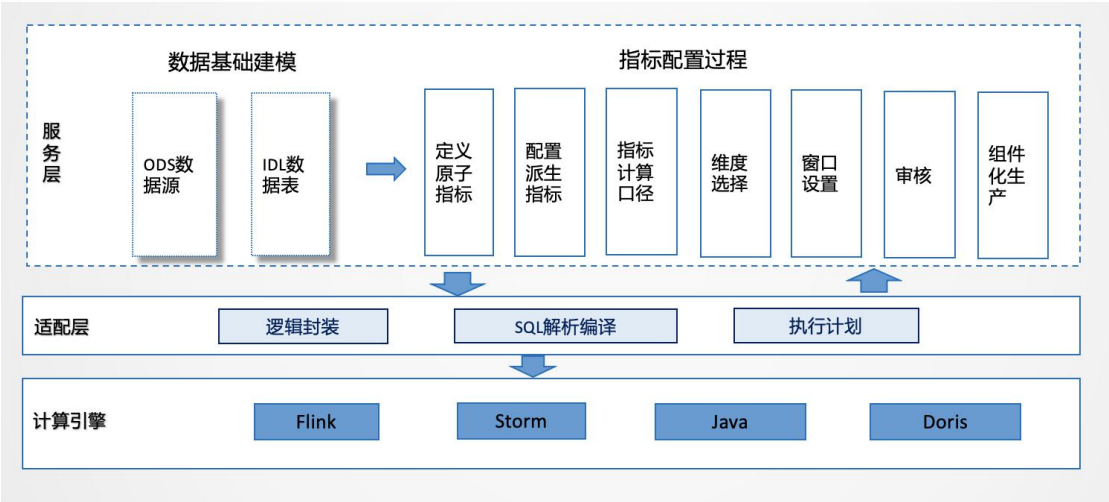
实时基础层的建设要解决一些问题。首先是一条流重复读的问题，一条 Binlog 打过来，是以 DB 包的形式存在的，用户可能只用其中一张表，如果大家都要用，可能存在所有人都要接这个流的问题。解决方案是可以按照不同的业务解构出来，还原到基础数据流层，根据业务的需要做成范式结构，按照数仓的建模方式进行集成化的主题建设。



其次要进行组件的封装，比如基础层的清洗、过滤、扩维等功能，通过一个很简单的表达入口，让用户将逻辑写出来。数据转换环节是比较灵活的，比如从一个值转换成另外一个值，对于这种自定义逻辑表达，我们也开放了自定义组件，可以通过 Java 或 Python 开发自定义脚本，进行数据加工。

2. 实时特征生产功能

特征生产可以通过 SQL 语法进行逻辑表达，底层进行逻辑的适配，透传到计算引擎，屏蔽用户对计算引擎的依赖。就像对于离线场景，目前大公司很少通过代码的方式开发，除非一些特别的 Case，所以基本上可以通过 SQL 化的方式表达。



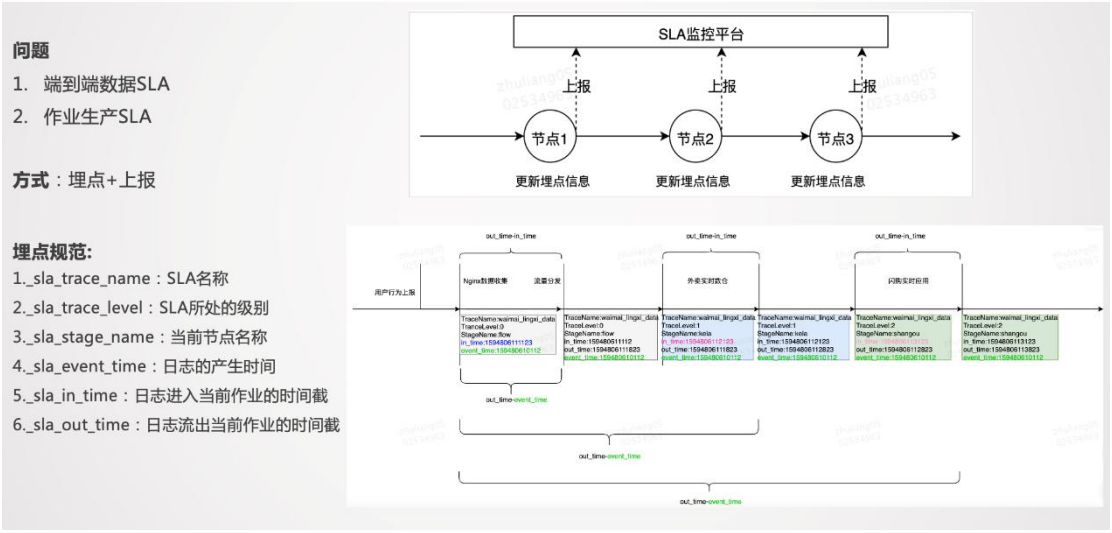
在功能层面，把指标管理的思想融合进去，原子指标、派生指标，标准计算口径，维度选择，窗口设置等操作都可以通过配置化的方式，这样可以统一解析生产逻辑，进行统一封装。

还有一个问题，同一个源，写了很多 SQL，每一次提交都会起一个数据流，比较浪费资源，我们的解决方案是，通过同一条流实现动态指标的生产，在不停服务的情况下可以动态添加指标。

所以在实时平台建设过程中，更多考虑的是如何更有效的利用资源，在哪些环节更能节约化的使用资源，这是在工程方面更多考虑的事情。

3. SLA 建设

SLA 主要解决两个问题，一个是端到端的 SLA，一个是作业生产效率的 SLA，我们采用埋点+上报的方式，由于实时流比较大，埋点要尽量简单，不能埋太多的东西，能表达业务即可，每个作业的输出统一上报到 SLA 监控平台，通过统一接口的形式，在每一个作业点上报所需要的信息，最后能够统计到端到端的 SLA。



在实时生产中，由于链路非常长，无法控制所有链路，但是可以控制自己作业的效率，所以作业 SLA 也是必不可少的。

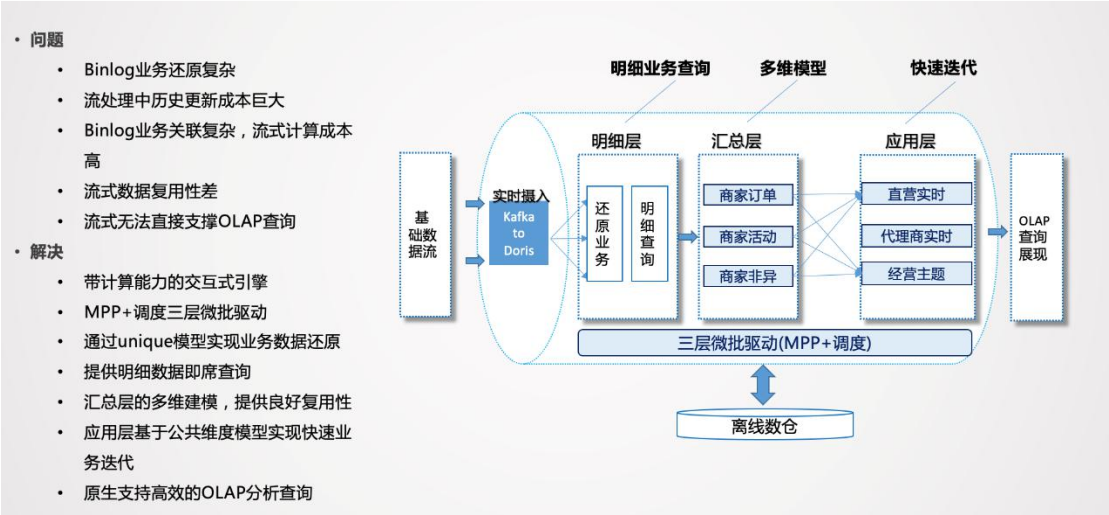
4. 实时 OLAP 方案

问题

- **Binlog 业务还原复杂**：业务变化很多，需要某个时间点的变化，因此需要进行排序，并且数据要存起来，这对于内存和 CPU 的资源消耗都是非常大的。
- **Binlog 业务关联复杂**：流式计算里，流和流之间的关联，对于业务逻辑的表达是非常困难的。

解决方案

通过带计算能力的 OLAP 引擎来解决，不需要把一个流进行逻辑化映射，只需要解决数据实时稳定的入库问题。



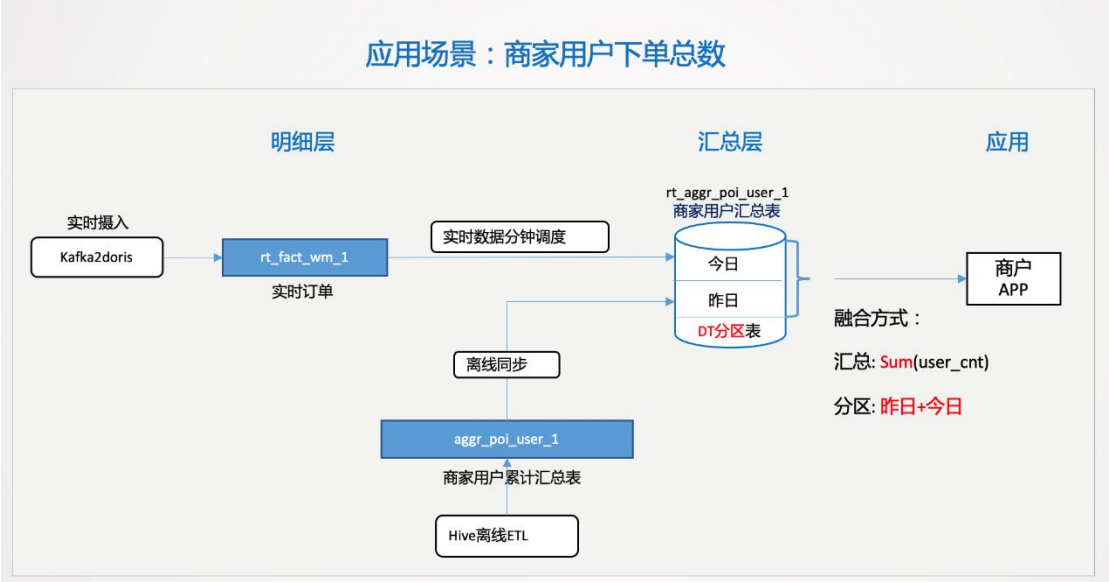
我们这边采用的是 Doris 作为高性能的 OLAP 引擎，由于业务数据产生的结果和结果之间还需要进行衍生计算，Doris 可以利用 Unique 模型或聚合模型快速还原业务，还原业务的同时还可以进行汇总层的聚合，也是为了复用而设计。应用层可以是物理的，也可以是逻辑化视图。

这种模式重在解决业务回撤计算，比如业务状态改变，需要在历史的某个点将值变更，这种场景用流计算的成本非常大，OLAP 模式可以很好的解决这个问题。

07 实时应用案例

最后通过一个案例说明，比如商家要根据用户历史下单数给用户优惠，商家需要看到历史下了多少单，历史 T+1 的数据要有，今天实时的数据也要有，这种场景是典型的 Lambda 架构。我们可以在 Doris 里设计一个分区表，一个是历史

分区，一个是今日分区，历史分区可以通过离线的方式生产，今日指标可以通过实时的方式计算，写到今日分区里，查询的时候进行一个简单的汇总。



这种场景看起来比较简单，难点在于商家的量上来之后，很多简单的问题都会变得复杂。后续，我们也会通过更多的业务输入，沉淀出更多的业务场景，抽象出来形成统一的生产方案和功能，以最小化的实时计算资源支撑多样化的业务需求，这也是未来我们需要达到的目的。

五、美团基于 Flink 的实时数仓建设实践

引言

近些年，企业对数据服务实时化服务的需求日益增多。本文整理了常见实时数据组件的性能特点和适用场景，介绍了美团如何通过 Flink 引擎构建实时数据仓库，从而提供高效、稳健的实时数据服务。此前我们美团技术博客发布过一篇文

章《流计算框架 Flink 与 Storm 的性能对比》，对 Flink 和 Storm 俩个引擎的计算性能进行了比较。本文主要阐述使用 Flink 在实际数据生产上的经验。

实时平台初期架构

在实时数据系统建设初期，由于对实时数据的需求较少，形成不了完整的数据体系。我们采用的是“一路到底”的开发模式：通过在实时计算平台上部署 Storm 作业处理实时数据队列来提取数据指标，直接推送到实时应用服务中。

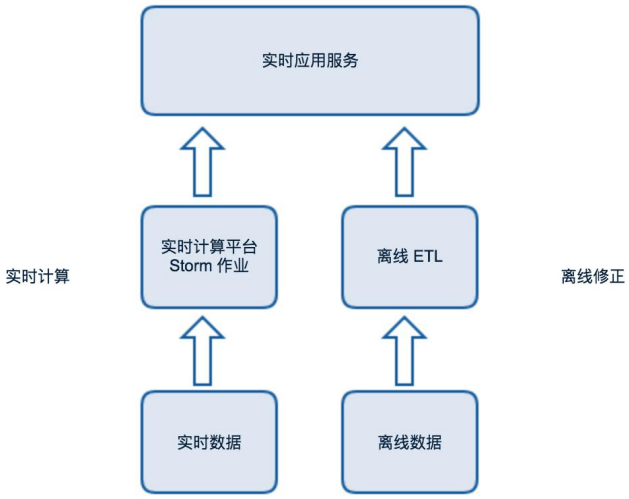


图 1 初期实时数据架构

但是，随着产品和业务人员对实时数据需求的不断增多，新的挑战也随之发生。

1. 数据指标越来越多，“烟囱式”的开发导致代码耦合问题严重。
2. 需求越来越多，有的需要明细数据，有的需要 OLAP 分析。单一的开发模式难以应付多种需求。
3. 缺少完善的监控系统，无法在对业务产生影响之前发现并修复问题。

实时数据仓库的构建

为解决以上问题，我们根据生产离线数据的经验，选择使用分层设计方案来建设实时数据仓库，其分层架构如下图所示：

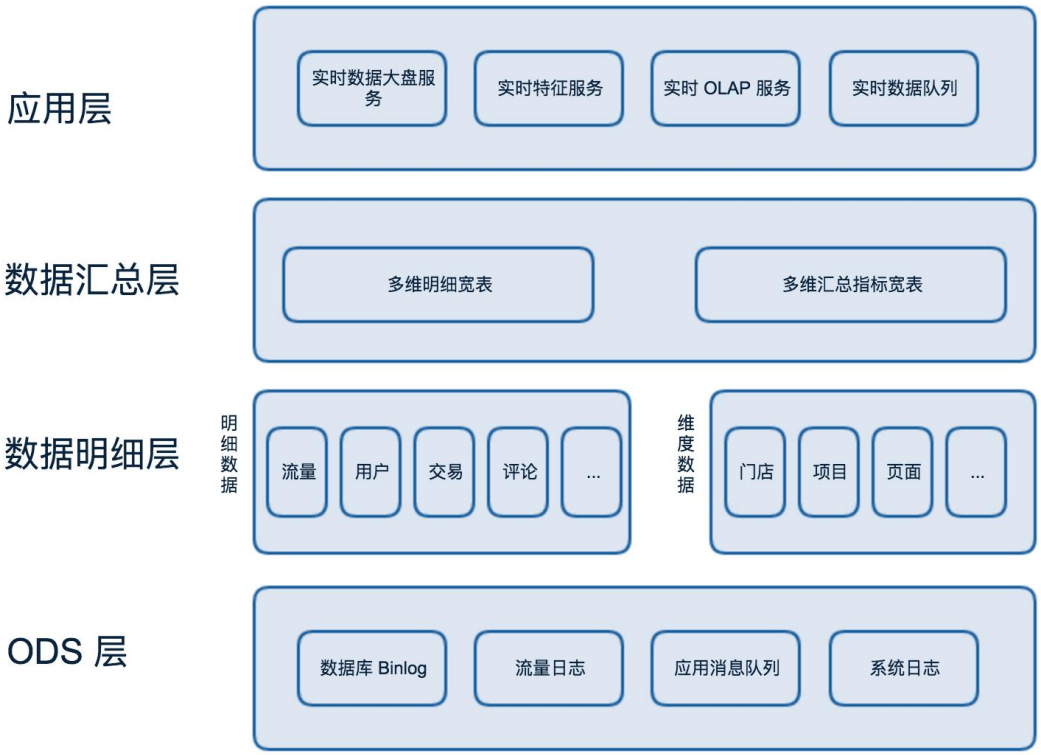


图 2 实时数仓数据分层架构

该方案由以下四层构成：

1. ODS 层：Binlog 和流量日志以及各业务实时队列。
2. 数据明细层：业务领域整合提取事实数据，离线全量和实时变化数据构建实时维度数据。
3. 数据汇总层：使用宽表模型对明细数据补充维度数据，对共性指标进行汇总。
4. App 层：为了具体需求而构建的应用层，通过 RPC 框架对外提供服务。

通过多层设计我们可以将处理数据的流程沉淀在各层完成。比如在数据明细层统一完成数据的过滤、清洗、规范、脱敏流程；在数据汇总层加工共性的多维指标汇总数据。提高了代码的复用率和整体生产效率。同时各层级处理的任务类型相似，可以采用统一的技术方案优化性能，使数仓技术架构更简洁。

技术选型

1.存储引擎的调研

实时数仓在设计中不同于离线数仓在各层级使用同种储存方案，比如都存储在 Hive 、DB 中的策略。首先对中间过程的表，采用将结构化的数据通过消息队列存储和高速 KV 存储混合的方案。实时计算引擎可以通过监听消息消费消息队列内的数据，进行实时计算。而在高速 KV 存储上的数据则可以用于快速关联计算，比如维度数据。 其次在应用层上，针对数据使用特点配置存储方案直接写入。避免了离线数仓应用层同步数据流程带来的处理延迟。 为了解决不同类型的实时数据需求，合理的设计各层级存储方案，我们调研了美团内部使用比较广泛的几种存储方案。

存储方案列表如下：

方案	优势	劣势
MySQL	1. 具有完备的事务功能，可以对数据进行更新。2. 支持 SQL，开发成本低。	1. 横向扩展成本大，存储容易成为瓶颈； 2. 实时数据的更新和查询频率都很高，线上单个实时应用请求就有 1000+ QPS；使用 MySQL 成本太高。
Elasticsearch	1. 吞吐量大，单个机器可以支持 2500+ QPS，并且集群可以快速横向扩展。2. Term 查询时响应速度很快，单个机器在 2000+ QPS 时，查询延迟在 20 ms 以内。	1. 没有原生的 SQL 支持，查询 DSL 有一定的学习门槛；2. 进行聚合运算时性能下降明显。
Druid	1. 支持超大数据量，通过 Kafka 获取实时数据时，单个	1. 预聚合导致无法支持明细的查询；2. 无法支持

方案	优势	劣势
Cellar	作业可支持 6W+ QPS；2.可以在数据导入时通过预计算对数据进行汇总，减少的数据存储。提高了实际处理数据的效率； 3. 有很多开源OLAP 分析框架。实现如Superset。 1. 支持超大数据量，采用内存加分布式存储的架构，存储性价比很高；2. 吞吐性能好，经测试处理 3W+ QPS 读写请求时，平均延迟在 1ms 左右；通过异步读写线上最高支持 10W+ QPS。	Join 操作；3. Append-only 不支持数据的修改。只能以 Segment 为单位进行替换。 1. 接口仅支持 KV，Map，List 以及原子加减等；2. 单个 Key 值不得超过 1KB，而 Value 的值超过 100KB 时则性能下降明显。

根据不同业务场景，实时数仓各个模型层次使用的存储方案大致如下：

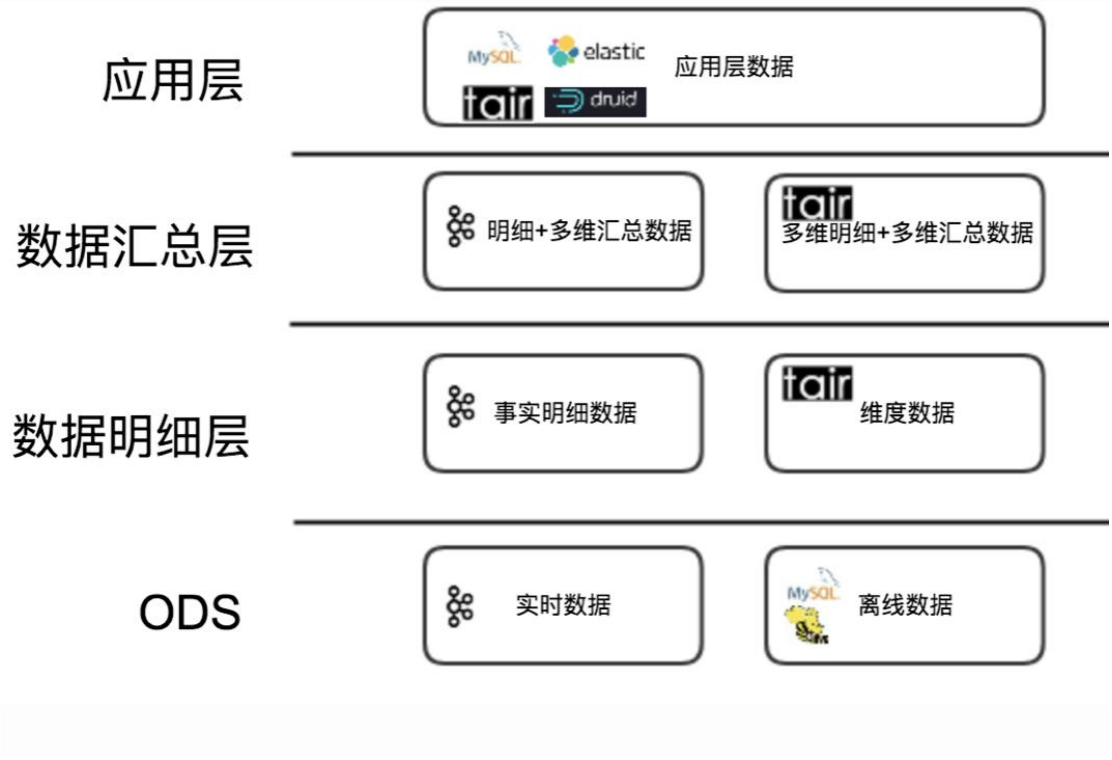


图 3 实时数仓存储分层架构

1. **数据明细层** 对于维度数据部分场景下关联的频率可达 10w+ TPS，我们选择 Cellar（美团内部存储系统） 作为存储，封装维度服务为实时数仓提供维度数据。

2. **数据汇总层** 对于通用的汇总指标，需要进行历史数据关联的数据，采用和维度数据一样的方案通过 Cellar 作为存储，用服务的方式进行关联操作。
3. **数据应用层** 应用层设计相对复杂，再对比了几种不同存储方案后。我们制定了以数据读写频率 1000 QPS 为分界的判断依据。对于读写平均频率高于 1000 QPS 但查询不太复杂的实时应用，比如商户实时的经营数据。采用 Cellar 为存储，提供实时数据服务。对于一些查询复杂的和需要明细列表的应用，使用 Elasticsearch 作为存储则更为合适。而一些查询频率低，比如一些内部运营的数据。 Druid 通过实时处理消息构建索引，并通过预聚合可以快速的提供实时数据 OLAP 分析功能。对于一些历史版本的数据产品进行实时化改造时，也可以使用 MySQL 存储便于产品迭代。

2.计算引擎的调研

在实时平台建设初期我们使用 Storm 引擎来进行实时数据处理。Storm 引擎虽然在灵活性和性能上都表现不错。但是由于 API 过于底层，在数据开发过程中需要对一些常用的数据操作进行功能实现。比如表关联、聚合等，产生了很多额外的开发工作，不仅引入了很多外部依赖比如缓存，而且实际使用时性能也不是很理想。同时 Storm 内的数据对象 Tuple 支持的功能也很简单，通常需要将其转换为 Java 对象来处理。对于这种基于代码定义的数据模型，通常我们只能通过文档来进行维护。不仅需要额外的维护工作，同时在增改字段时也很麻烦。综合来看使用 Storm 引擎构建实时数仓难度较大。我们需要一个新的实时处理方案，要能够实现：

1. 提供高级 API，支持常见的数据操作比如关联聚合，最好是能支持 SQL。
2. 具有状态管理和自动支持久化方案，减少对存储的依赖。
3. 便于接入元数据服务，避免通过代码管理数据结构。
4. 处理性能至少要和 Storm 一致。

我们对主要的实时计算引擎进行了技术调研。总结了各类引擎特性如下表所示：

实时计算方案列表如下：

项目/引擎	Storm	Flink	spark-treaming

项目/引擎	Storm	Flink	spark-treaming
API	灵活的底层 API 和具有事务保证的 Trident API	流 API 和更加适合数据开发的 Table API 和 Flink SQL 支持	流 API 和 Structured-Streaming API 同时也可以使用更适合数据开发的 Spark SQL
容错机制	ACK 机制	State 分布式快照保存点	RDD 保存点
状态管理	Trident State 状态管理	Key State 和 Operator State 两种 State 可以使用，支持多种持久化方案	有 UpdateStateByKey 等 API 进行带状态的变更，支持多种持久化方案
处理模式	单条流式处理	单条流式处理	Mic batch 处理
延迟	毫秒级	毫秒级	秒级
语义保障	At Least Once，Exactly Once	Exactly Once，At Least Once	At Least Once

从调研结果来看，Flink 和 Spark Streaming 的 API 、容错机制与状态持久化机制都可以解决一部分我们目前使用 Storm 中遇到的问题。但 Flink 在数据延迟上和 Storm 更接近，对现有应用影响最小。而且在公司内部的测试中 Flink 的吞吐性能对比 Storm 有十倍左右提升。综合考量我们选定 Flink 引擎作为实时数仓的开发引擎。

更加引起我们注意的是，Flink 的 Table 抽象和 SQL 支持。虽然使用 Strom 引擎也可以处理结构化数据。但毕竟依旧是基于消息的处理 API ，在代码层层

面上不能完全享受操作结构化数据的便利。而 Flink 不仅支持了大量常用的 SQL 语句，基本覆盖了我们的开发场景。而且 Flink 的 Table 可以通过 TableSchema 进行管理，支持丰富的数据类型和数据结构以及数据源。可以很容易的和现有的元数据管理系统或配置管理系统结合。通过下图我们可以清晰的看出 Storm 和 Flink 在开发统过程中的区别。

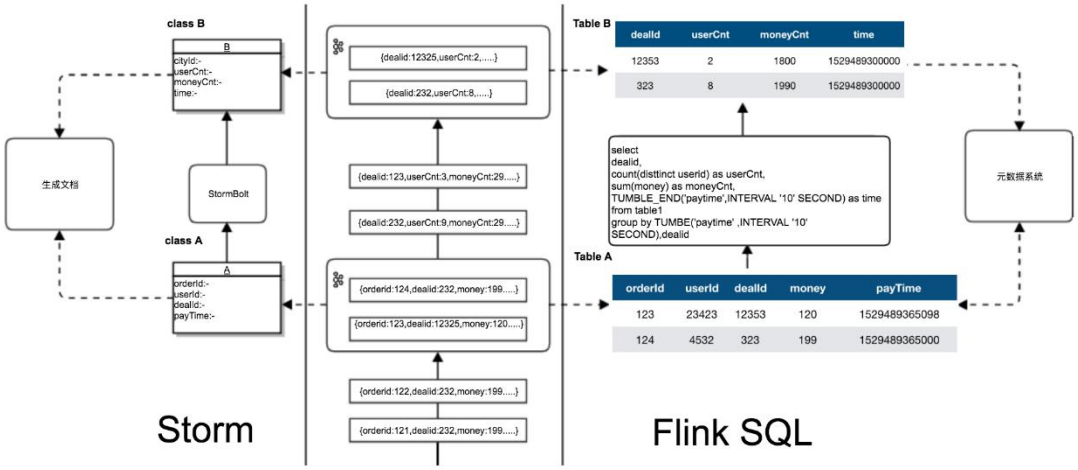


图 4 Flink - Storm 对比图

在使用 Storm 开发时处理逻辑与实现需要固化在 Bolt 的代码。Flink 则可以通过 SQL 进行开发，代码可读性更高，逻辑的实现由开源框架来保证可靠高效，对特定场景的优化只要修改 Flink SQL 优化器功能实现即可，而不影响逻辑代码。使我们可以把更多的精力放到到数据开发中，而不是逻辑的实现。当需要离线数据和实时数据口径统一的场景时，我们只需对离线口径的 SQL 脚本稍加改造即可，极大地提高了开发效率。同时对比图中 Flink 和 Storm 使用的数据模型，Storm 需要通过一个 Java 的 Class 去定义数据结构，Flink Table 则可以通过元数据来定义。可以很好的和数据开发中的元数据，数据治理等系统结合，提高开发效率。

Flink 使用心得

在利用 Flink-Table 构建实时数据仓库过程中。我们针对一些构建数据仓库的常用操作，比如数据指标的维度扩充，数据按主题关联，以及数据的聚合运算通过 Flink 来实现总结了一些使用心得。

1. 维度扩充

数据指标的维度扩充，我们采用的是通过维度服务获取维度信息。虽然基于 Cellar 的维度服务通常的响应延迟可以在 1ms 以下。但是为了进一步优化 Flink 的吞吐，我们对维度数据的关联全部采用了异步接口访问的方式，避免了使用 RPC 调用影响数据吞吐。 对于一些数据量很大的流，比如流量日志数据量在 10W 条/秒这个量级。在关联 UDF 的时候内置了缓存机制，可以根据命中率和时间对缓存进行淘汰，配合用关联的 Key 值进行分区，显著减少了对外部服务的请求次数，有效的减少了处理延迟和对外部系统的压力。

2. 数据关联

数据主题合并，本质上就是多个数据源的关联，简单的来说就是 Join 操作。Flink 的 Table 是建立在无限流这个概念上的。在进行 Join 操作时并不能像离线数据一样对两个完整的表进行关联。采用的是在窗口时间内对数据进行关联的方案，相当于从两个数据流中各自截取一段时间的数据进行 Join 操作。有点类似于离线数据通过限制分区来进行关联。同时需要注意 Flink 关联表时必须有至少一个“等于”关联条件，因为等号两边的值会用来分组。

由于 Flink 会缓存窗口内的全部数据来进行关联，缓存的数据量和关联的窗口大小成正比。因此 Flink 的关联查询，更适合处理一些可以通过业务规则限制关联数据时间范围的场景。比如关联下单用户购买之前 30 分钟内的浏览日志。过大的窗口不仅会消耗更多的内存，同时会产生更大的 Checkpoint ，导致吞吐下降或 Checkpoint 超时。在实际生产中可以使用 RocksDB 和启用增量保存点模式，减少 Checkpoint 过程对吞吐产生影响。对于一些需要关联窗口期很长的场景，比如关联的数据可能是几天以前的数据。对于这些历史数据，我们可以将其理解为是一种已经固定不变的”维度”。可以将需要被关联的历史数据采用和维度数据一致的处理方法：”缓存 + 离线”数据方式存储，用接口的方式进行关联。另外需要注意 Flink 对多表关联是直接顺序链接的，因此需要注意先进行结果集小的关联。

3. 聚合运算

使用聚合运算时，Flink 对常见的聚合运算如求和、极值、均值等都有支持。美中不足的是对于 Distinct 的支持，Flink-1.6 之前的采用的方案是通过先对去重字段进行分组再聚合实现。对于需要对多个字段去重聚合的场景，只能分别计算再进行关联处理效率很低。为此我们开发了自定义的 UDAF，实现了 MapView 精确去重、BloomFilter 非精确去重、 HyperLogLog 超低内存去重方案应对各种实时去重场景。但是在使用自定义的 UDAF 时，需要注意 RocksDBStateBackend 模式对于较大的 Key 进行更新操作时序列化和反序列化耗时很多。可以考虑使用 FsStateBackend 模式替代。另外要注意的一点 Flink 框架在计算比如 Rank 这样的分析函数时，需要缓存每个分组窗口下的全

部数据才能进行排序，会消耗大量内存。建议在这种场景下优先转换为 TopN 的逻辑，看是否可以解决需求。

下图展示一个完整的使用 Flink 引擎生产一张实时数据表的过程：

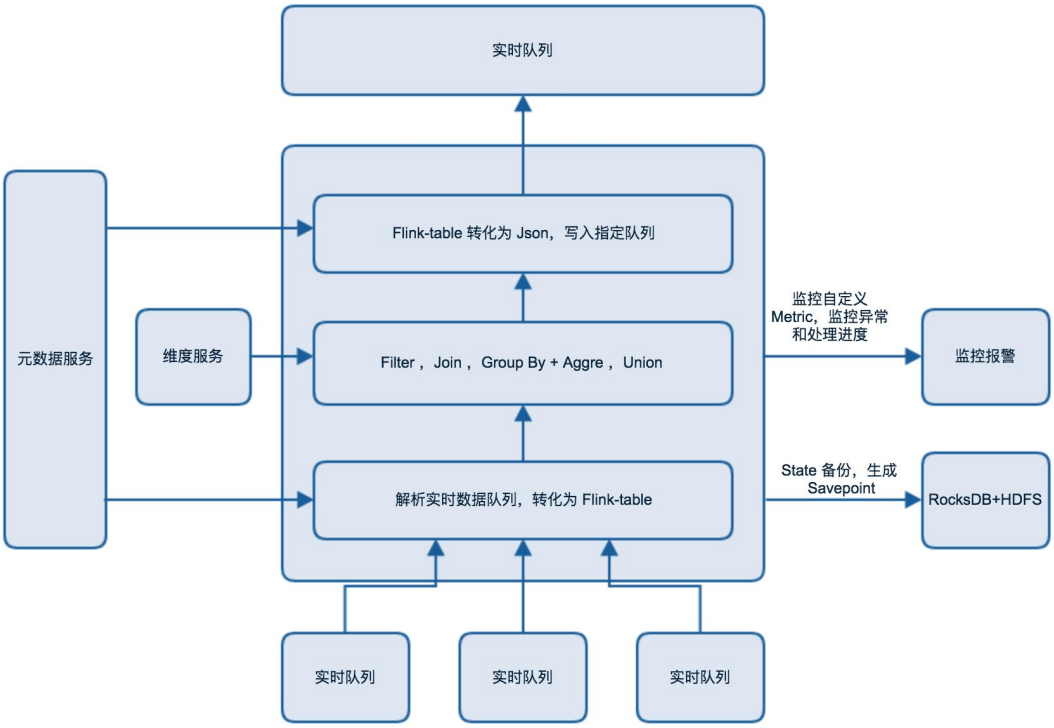


图 5 实时计算流程图

实时数仓成果

通过使用实时数仓代替原有流程，我们将数据生产中的各个流程抽象到实时数仓的各层当中。实现了全部实时数据应用的数据源统一，保证了应用数据指标、维度的口径的一致。在几次数据口径发生修改的场景中，我们通过对仓库明细和汇总进行改造，在完全不用修改应用代码的情况下就完成全部应用的口径切换。在开发过程中通过严格的把控数据分层、主题域划分、内容组织标准规范和命名规则。使数据开发的链路更为清晰，减少了代码的耦合。再配合上使用 Flink SQL

进行开发，代码加简洁。单个作业的代码量从平均 300+ 行的 JAVA 代码 ，缩减到几十行的 SQL 脚本。项目的开发时长也大幅减短，一人日开发多个实时数据指标情况也不少见。

除此以外我们通过针对数仓各层级工作内容的不同特点，可以进行针对性的性能优化和参数配置。比如 ODS 层主要进行数据的解析、过滤等操作，不需要 RPC 调用和聚合运算。 我们针对数据解析过程进行优化，减少不必要的 JSON 字段解析，并使用更高效的 JSON 包。在资源分配上，单个 CPU 只配置 1GB 的内存即可满需求。而汇总层主要则主要进行聚合与关联运算，可以通过优化聚合算法、内外存共同运算来提高性能、减少成本。资源配置上也会分配更多的内存，避免内存溢出。通过这些优化手段，虽然相比原有流程实时数仓的生产链路更长，但数据延迟并没有明显增加。同时实时数据应用所使用的计算资源也有明显减少。

展望

我们的目标是将实时仓库建设成可以和离线仓库数据准确性，一致性媲美的数据系统。为商家，业务人员以及美团用户提供及时可靠的数据服务。同时作为到餐实时数据的统一出口，为集团其他业务部门助力。未来我们将更加关注在数据可靠性和实时数据指标管理。建立完善的数据监控，数据血缘检测，交叉检查机制。及时对异常数据或数据延迟进行监控和预警。同时优化开发流程，降低开发实时数据学习成本。让更多有实时数据需求的人，可以自己动手解决问题。

数据平台

六、美团数据平台融合实践

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”，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。

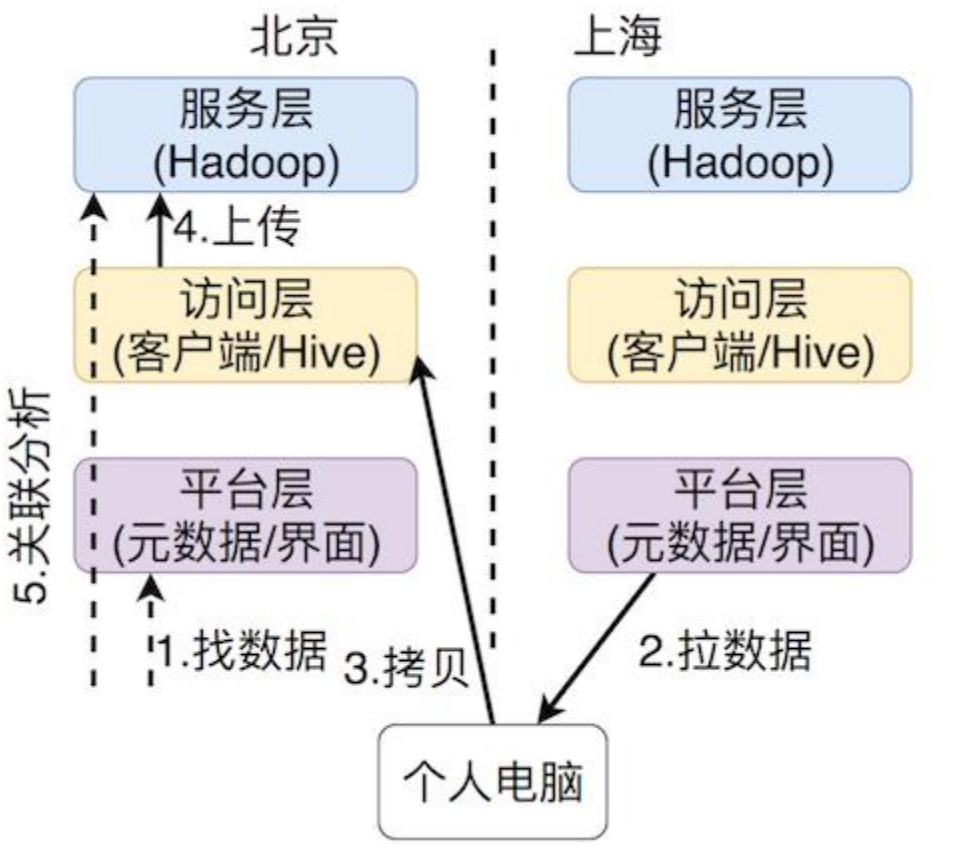
本文将重点讲述数据平台融合项目的实践思路和经验，并深入地讨论 Hadoop 多机房架构的一种实现方案，以及大面积 SQL 任务重构的一种平滑化方法。最后介绍这种复杂的平台系统如何保证平稳平滑地融合。

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。



1

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。

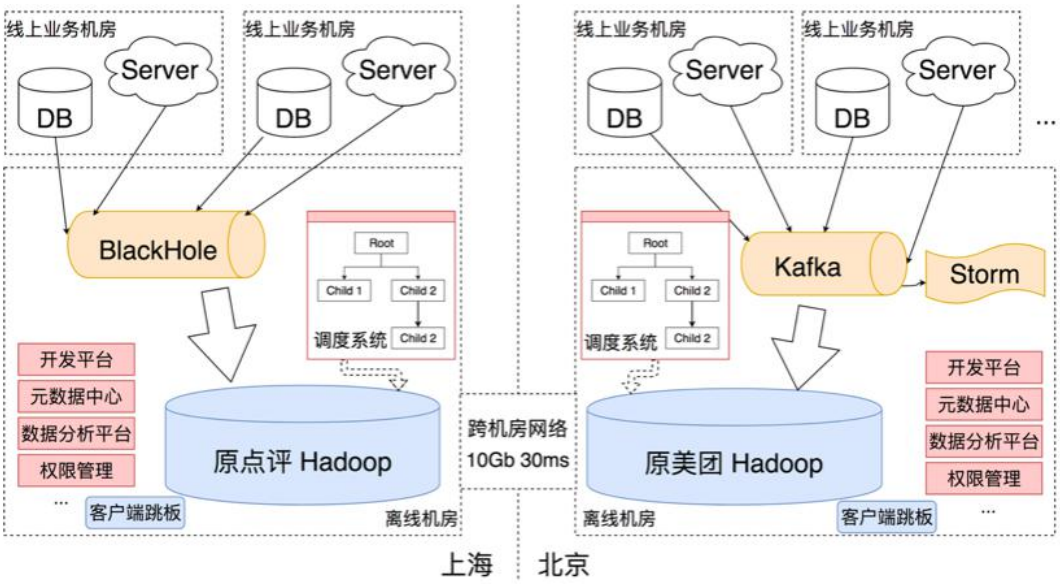
确立目标

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



2

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+

4

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。

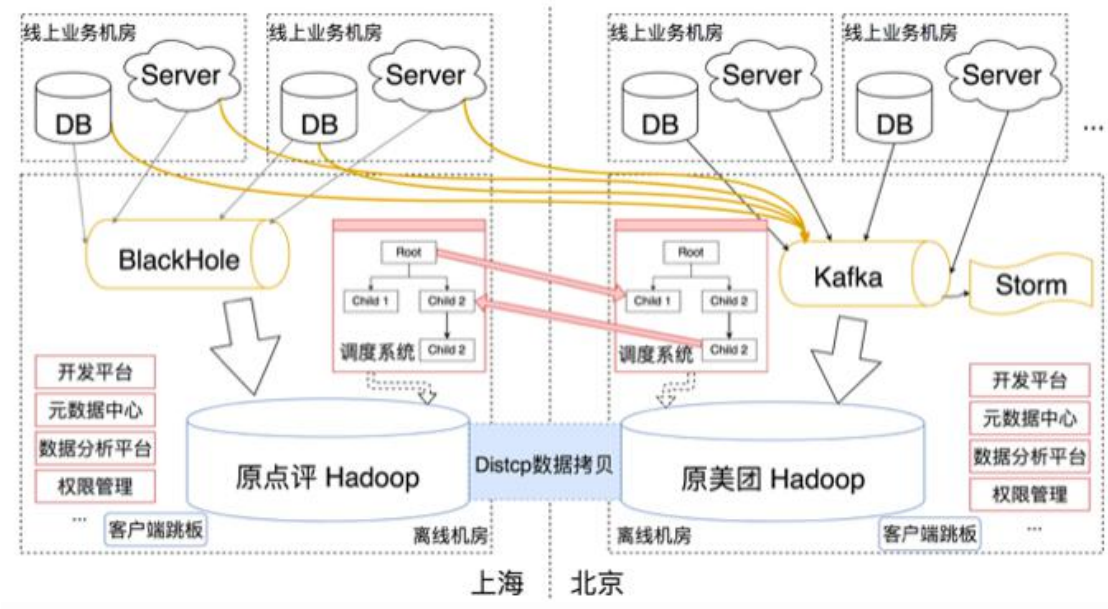
我们想要改一个表的内容、一个数据任务的逻辑，都是不被允许的，都必须是由业务侧的同学们来做。两侧的平台融合难免存在功能性的差异，数据开发平台的日活跃就有 100 和 240，如果查询就是每天作分析的日活跃的话，原点评和美团加起来有 1000 多。所以在平台融合过程中，能让这么多用户觉得毫无违和感是非常有挑战的。综上，我们做了一个项目拆解。

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。

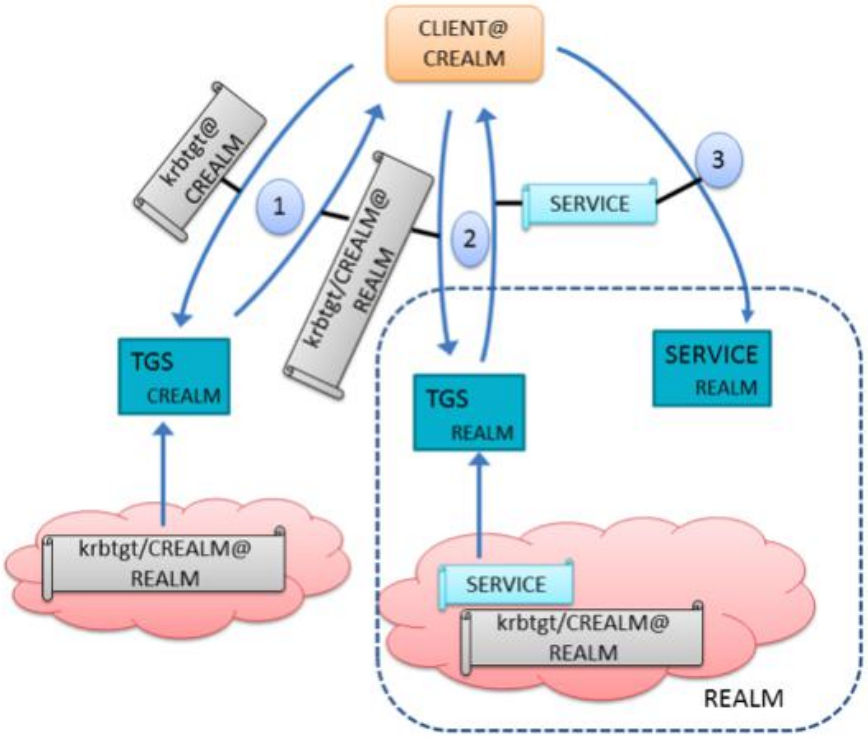
然后我们两侧集群分别建立一个用于同步的账号，原则是在读的那一端提交任务。什么叫“读的一端”？比如说我想把一个原点评侧的数据同步到原美团侧，原美团侧就是要读的那端，我在原美团侧起这个任务去读原点评侧的数据，然后写到原美团侧。这里的主要考虑是读端更多是需求端，所以，他要在他的资源池里去跑。另外，对集群的影响读小于写，我们希望对于被读集群的影响能尽量减少。

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。

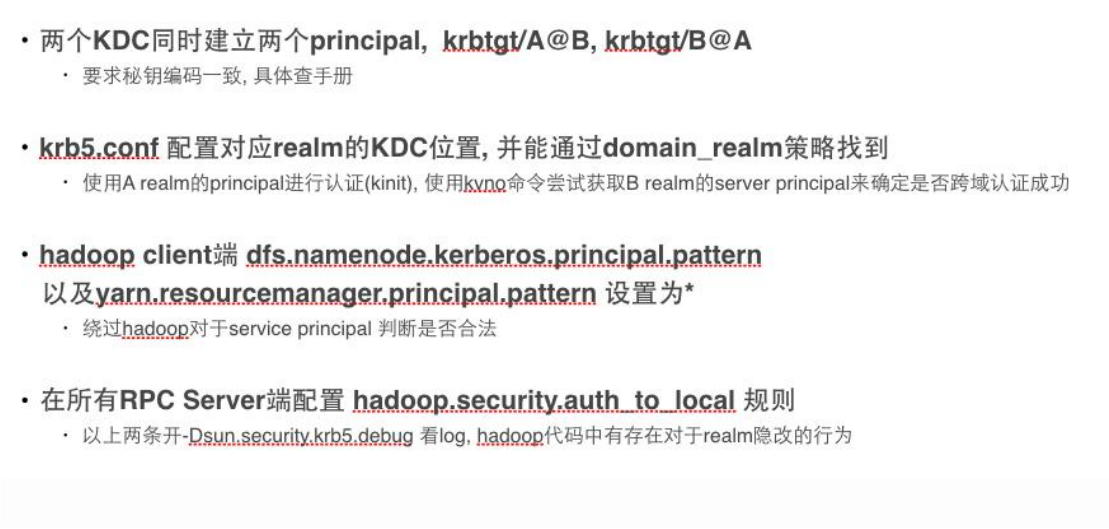


6

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。



7

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。

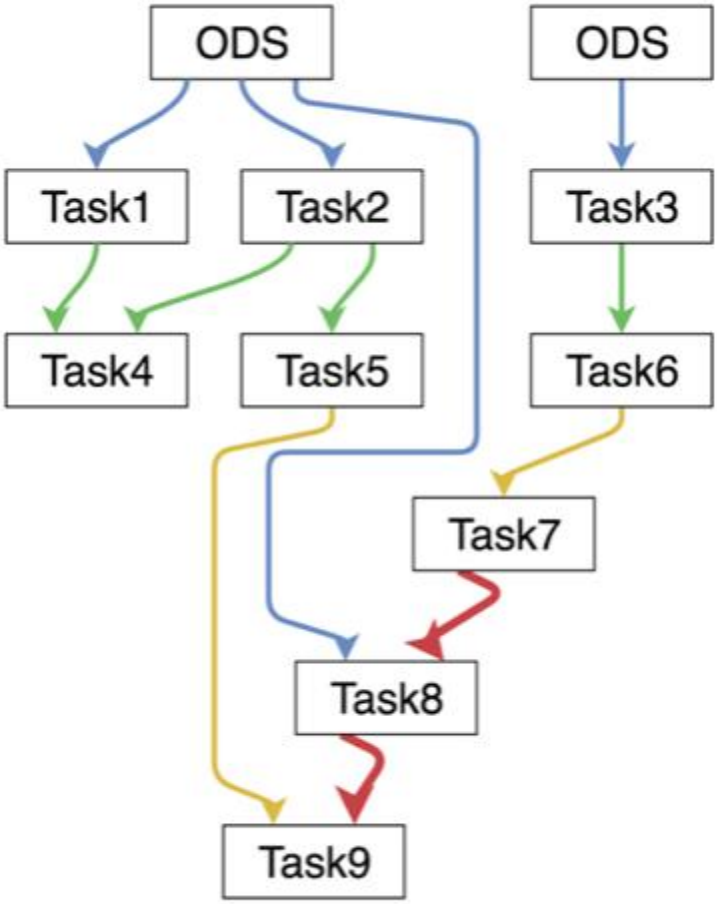
我们做完这个项目之后，分析师就可以愉快地配置一些调度任务去同步数据，然后在对应的集群上去关联他们的数据进行分析了。做完这个项目之后，我们两边的团队也相互磨合，相互形成了一定的认可。因为这个小项目涉及到了数据平台的每一个领域，包括工具链、实时计算、离线的团队都做了一些磨合。

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？



8

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
……	……	…….

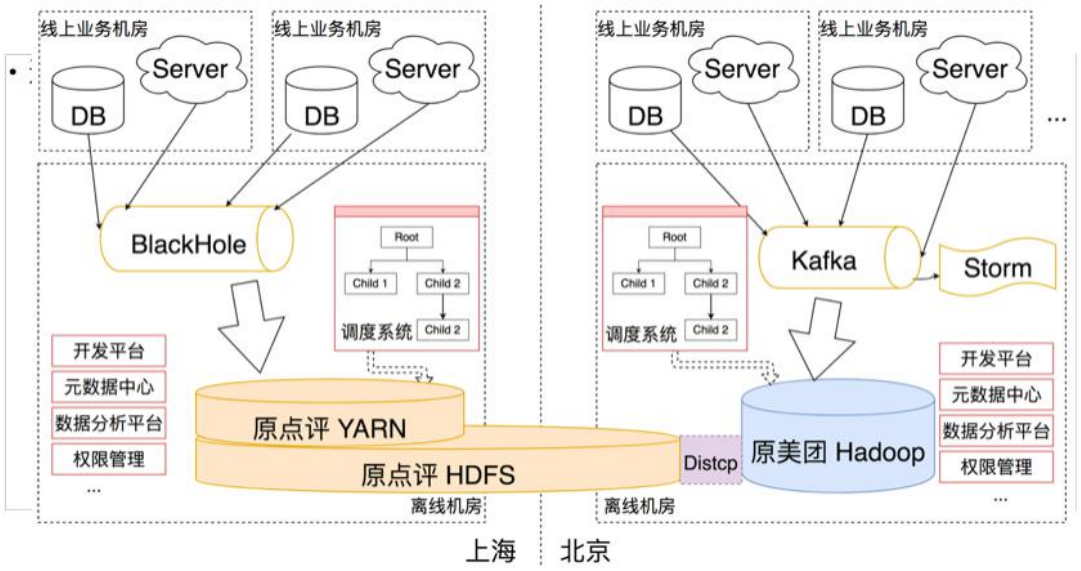
9

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。



10

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。

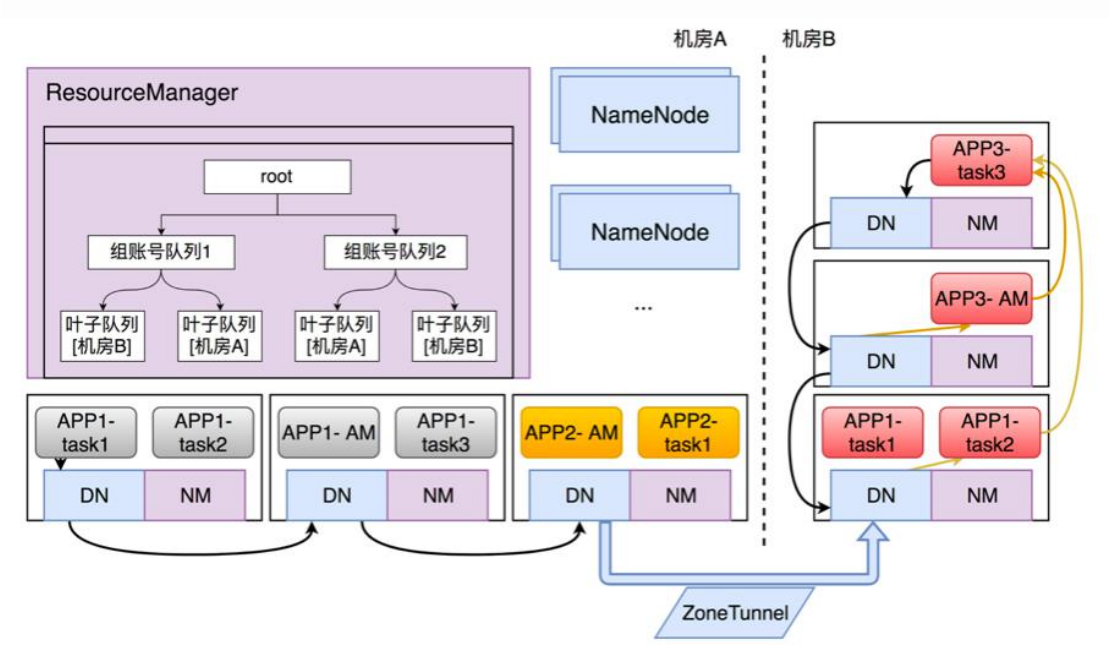
那整个集群的拷贝和切换是怎么做的呢？其实就是用我们自研的一套 Hadoop 多机房架构。可能做 Hadoop 集群维护管理的同学们对这个有深刻的体会，就是不时地就要从一个机房搬到另一个机房。设计目标是说我们一个 Hadoop 集群可以跨机房去部署，然后在块的力度上能控制数据副本的放置策略，甚至是进行主动迁移。

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。



11

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。

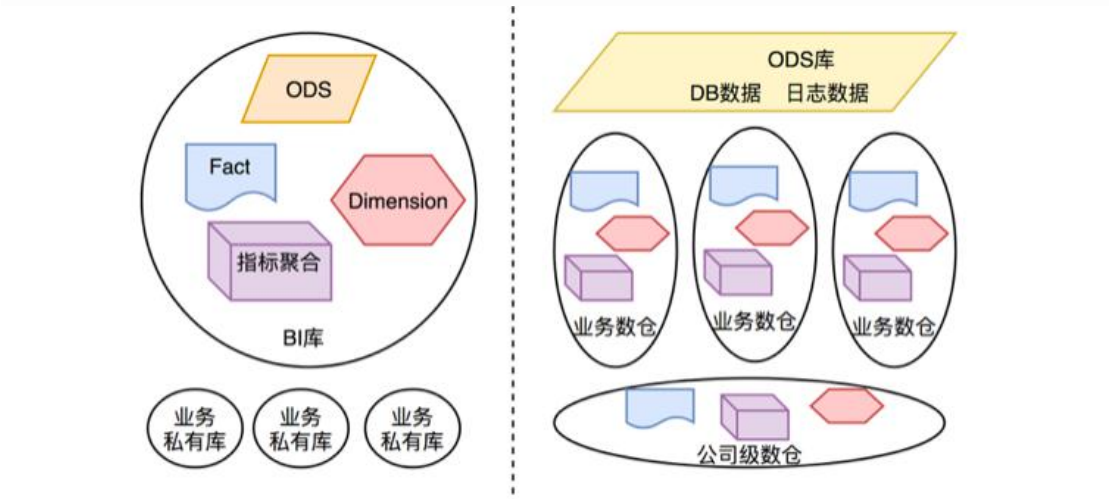
前面说的那个做了跨域认证的配置我们还是要拆掉的，必须进行服务认证的统一，不然的话以后没法看起来像一个集群，这个时候把原来的 KDC 里面的账号进行导出，之后逐步地去切换每一个配置，让它慢慢切到新的 KDC。切的过程当中，我们各种请求还是有跨域情况的，我们认为两个域是一体的，是一样的。等切干净之后，也就是原来的 KDC 没有请求了之后，我们再把它干掉。

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



12

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?

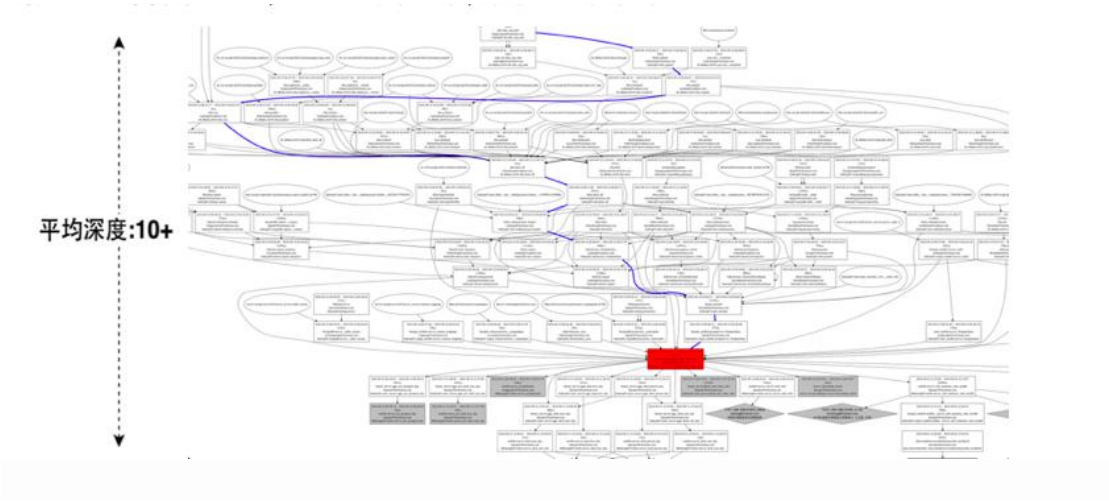
原来任务的内容大概是 insert into 一个 BI 库里面的一张表，接着 select from BI 库里面的某两张表，然后 where group by。像这样的任务我们有七八千个，它们在我们平台上配置着每天的依赖调度。我们希望把它都改成下图中的样子。所有涉及到的这些表都需要改名字，说白了就是一个批量改名字的事儿。

```
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 ***
```

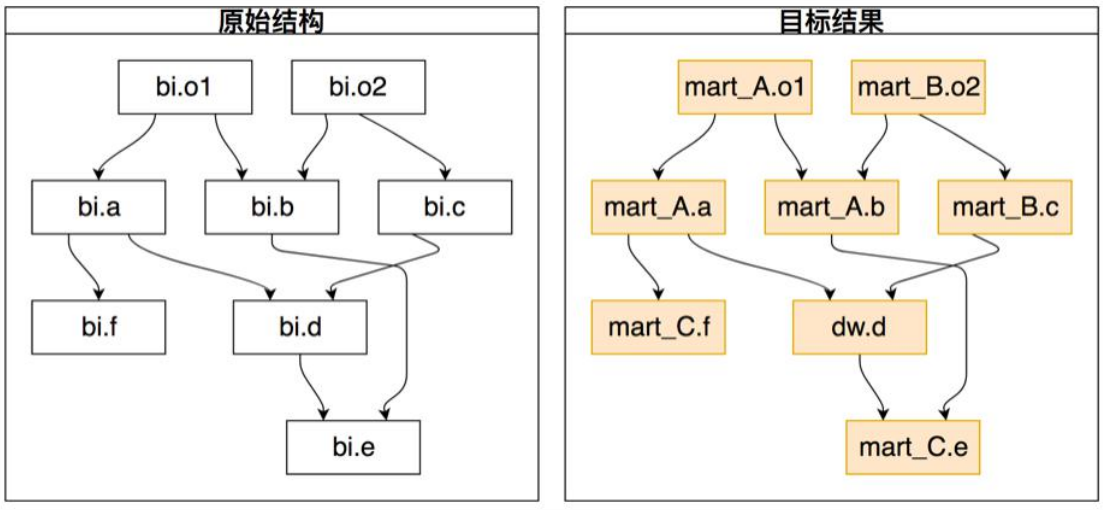
20

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？



13

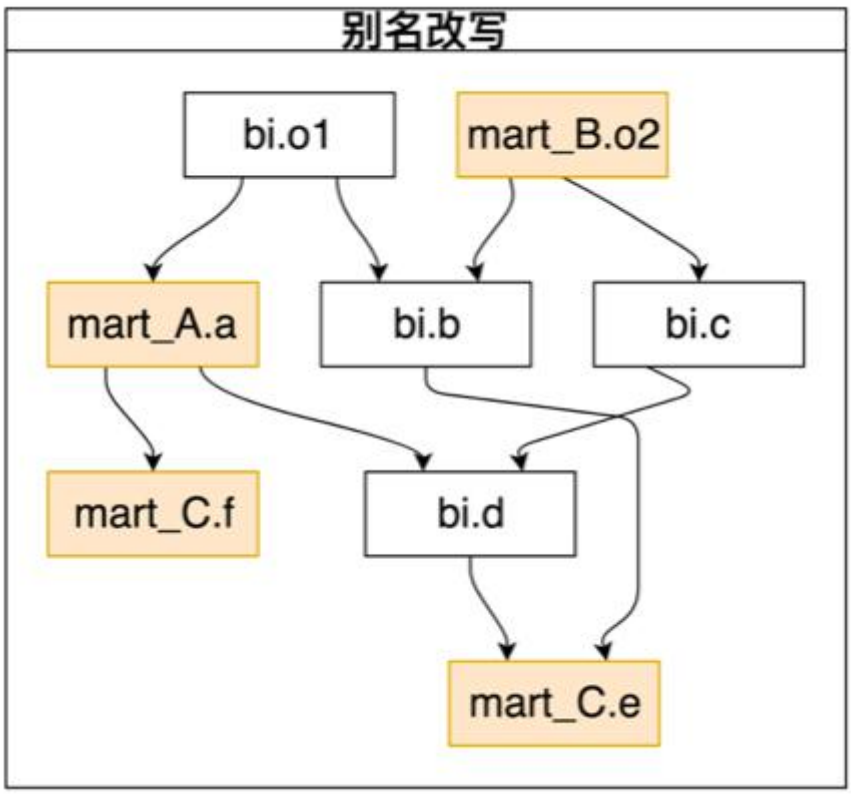
下图是我们的原始结构，首先这里有一个大前提是每一个任务只对一个结果表。原始的结构中，a 表只依赖 o1 表，b 表依赖 o1、o2，然后 c 表只依赖 o2，它们之间相互关联。这时候我希望可以对库名和表名进行一次性的修改。那如果我们逐层地去改写怎么办呢？首先要先把最上层的 mart 表改了，而我一旦改上游的某一个表，所有跟对它有依赖的表都必须改任务内容。每推动一层改动，下面一层都要变动多次，这样一来，我们这个流程就非常受限。



14

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



15

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。

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。

七、研发团队资源成本优化实践

背景

工程师主要面对的是技术挑战，更关注技术层面的目标。研发团队的管理者则会把实现项目成果和业务需求作为核心目标。实际项目中，研发团队所需资源（比如物理机器、内存、硬盘、网络带宽等）的成本，很容易被忽略，或者在很晚才考虑。

在一般情况下，如果要满足更多的技术指标如并发量和复杂度等，或者满足峰值业务的压力，最直接有效的方法就是投入更多的资源。然而，从全局来看，如果资源成本缺乏优化，最终会出现如下图所示的“边际效用递减”现象——技术能力的提升和资源的增幅并不匹配，甚至资源的膨胀速度会超过技术能力的提升，从而使技术项目本身的 ROI 大打折扣。

从笔者十余年的工作经验来看，资源成本优化宜早不宜迟。很多管理者在业务发展的较前期阶段，可能并没有意识到资源成本膨胀所带来的压力。等到业务到了一定规模，拿到机器账单的时候，惊呼“机器怎么这么费钱”，再想立即降低成本，可能已经错过了最佳时机，因为技术本身是一个相对长期的改造过程。

业务阶段	业务特点	成本管理意识
探索期	验证模式，从0到1	粗放式管理，只控制上限
进攻期	市场占有率是唯一目标	不需要控制成本，要多少给多少
发展期	稳居市场TOP2，业务成熟	看财报，发现设备这么花钱？
变革期	增速放缓，转型或变革	机器成本要控制了！

所以，正在阅读此文的读者，假如你已经感觉到了成本膨胀的压力，或者正在做成本控制相关的工作，恭喜，这是幸福的烦恼，贵公司的业务体量应该已经达到一定规模，同时也说明你的管理意识可能已经超前于业务发展了（握手）。

本文我们将分享美团到餐研发团队的资源成本优化实践。

实践

像美团这种体量的公司，资源的提供方包括多个团队，除了到店餐饮研发团队自用的资源外，还有多个兄弟团队也提供了资源支持或者联合共建，比如 SRE、大数据团队、风控团队、广告团队等等。在每个月拿到成本账单之后，我们都需要抽丝剥茧，对每一项进行拆解，才能制定对应的解决策略，具体流程如下图所示。

1. 确定方法论

“凡事预则立，不预则废”。在做一件事之前，要充分评估整个工作完整生命周期的要素，并进行整体工作框架的设计，一个科学的方法论是十分有必要的。成本优化遵循的是一个行业内成熟的 P（Plan）D（Do）C（Check）A（Act）方法论，在每个阶段都又有对应的二次迭代和微循环。具体方法论如下：

- **在 Plan 或 Standard 阶段要做的事**：建立意识->确定目标->分析现状->确定评价指标。
- **在 Do 执行阶段要做的事**：分解原子项目->确定方案->落实到人->优化原子指标。
- **在 Check 检查阶段要做的事**：规定动作检查->行动结果评估->系统问题定位->修正标准动作。
- **在 Act 优化处理阶段要做的事**：定期复盘->形成报告->迭代认知->升级方法论->下阶段目标。

2. 计划规划阶段（Plan&Standard）

在这个阶段的核心目标是：**用 2-3 个指标来衡量自己的工作**。很多工作之所以最后失败，很多时候是相关人员根本没有办法用具体可衡量的指标来衡量自己的工作，这样对于工作结果，只能有一个“定性”的认识（比如很好，很不错，不好，较差），而无法做到“定量”。

对于研发人员来讲，不能定量的结果是不够科学的，具体如何确定指标，或者确定哪些指标作为工作目标，其实也是一门学问（有机会另外发文章讨论）。这个阶段的几个建议步骤为：

- **建立意识。**这个是团队 Leader 的首要责任，要让团队成员明白自己在资源上花了多少钱，成本控制是不是一件真正有意义和价值的事，要做到大家认知一致。虽然见到过一些团队在提倡成本控制，但是落实到具体行动时，却流于形式或者无从下手，最后只能停留在口头上，并没有产生实际的效果。
- **确定目标。**这个过程相对宏观，也可以认为是“定性”的阶段。在这个阶段要明确的就是，在成本控制这件事上，后续动作要解决的问题是什么？比如有些团队是总体成本偏高，但有些团队总成本并不高，而是应该增加成本，有些团队是非核心服务消耗的成本偏高，这些目标都需要经过团队成员讨论后得到一致的结果。在后续阶段的迭代中，也可以进行不断地修正。就像“客户永远不知道自己的需求”一样，很多人是不清楚自己的目标的，可以使用 SMART 原则来明确目标。
- **分析现状。**对成本这件事，罗列相关的数据，尽可能多地帮助自己做判断。自己团队在成本优化这件事上，处在哪一个阶段，哪些工作有可能被进一步优化，在此阶段要明确出来。
- **确定评价指标。**对于不同的专业序列，甚至对于同一专业序列的不同人员，大家对于成本的评价指标都不一样。这个阶段要做到最终的收敛，把团队未来成本优化的结果，用明确的数据表示出来。具体在到餐研发团队，我们确认了 2 个优化的核心指标：总成本、总订单成本。后续大家所有努力的目标，如果跟这两个指标没有关系或者弱相关，都可以忽略。

本阶段最大的经验是“知易行难”，虽然拍脑袋想出来一两个方向和目标很容易，但是最后用数据论证现状时，如何判断自己这个指标是“优秀”、“良好”还是“不及格”？对标的团队是谁？为什么对标的对象是 TA？都是需要从人员规模、业务阶段、业务量、行业特点等方面考虑仔细，也需要想清楚，其工作量甚至不比实际干活阶段小。

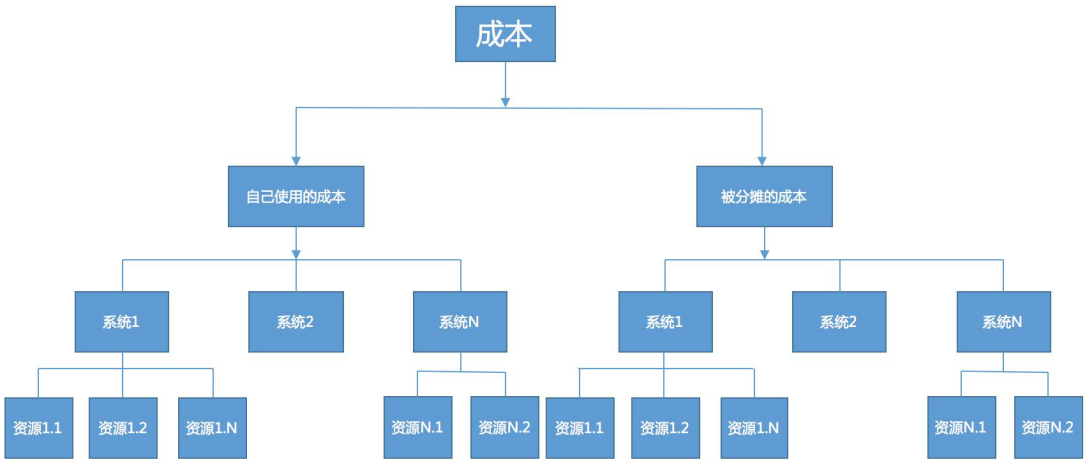
3. 执行阶段（Do）

3.1 建立思考流程

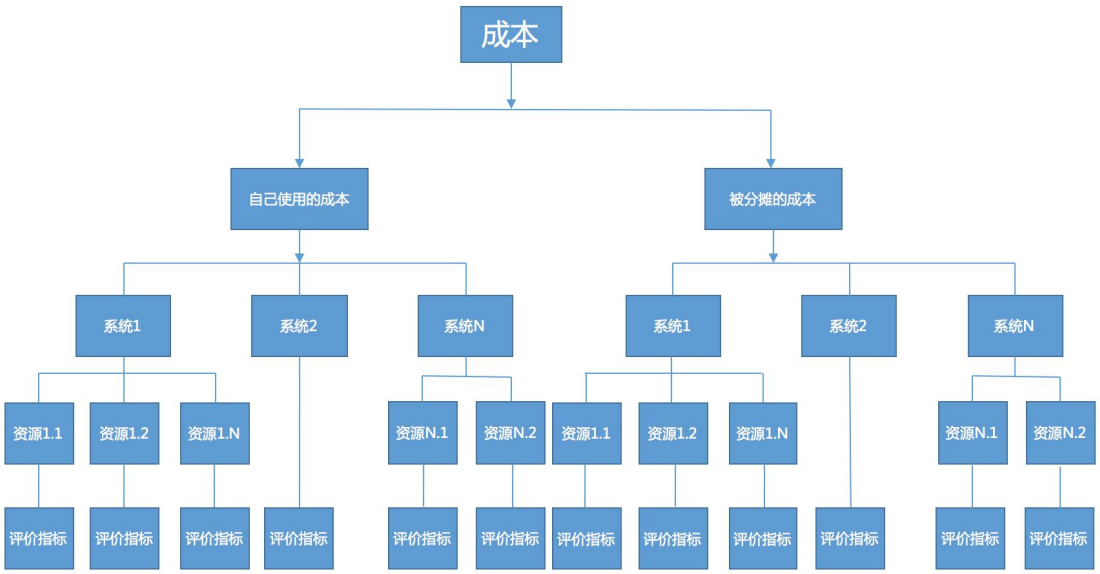
在执行阶段的流程是：分解原子项目->确定方案->落实到人->优化原子指标。在这里包括两个核心要素：1）把核心指标相关的工作向下一层分解；2）在下一层，找到具体的人来执行，这个人要具备将自己负责的指标继续分解到更细的

能力，类似于我们说的树状结构。这样层层地分解下去，每一层的叶子节点都可以找到对应的负责人。这种“总分”结构，在一本经典教程《金字塔原理》中也有详细的阐述。

- **分解原子项目**。在本阶段要建立一个完全细化的分级结构，用金字塔原理中的”MECE 不重不漏”原则，将工作内容分解到最细的可控粒度。至于按哪个维度进行拆分，不同的团队或者业务可能会有不同的原则，比如有些团队直接按子团队进行拆分，有些团队按业务进行拆分，有些团队按流程进行拆分。从较多团队通用的角度，成本控制这件事，可以简单的将指标分解到二级指标，包括“自身使用的成本”和“被分摊的成本”。其中，“自身使用的成本”是指，为了满足自己业务的需要，由本技术团队申请或者使用资源产生的成本；“被分摊的成本”是指，由于根据某种计算逻辑，间接使用了其他团队的资源，为其他技术团队承担一部分成本费用，比如常见的资源包括公司其他团队开发的广告、投放、风控、安全等系统。如果可以分拆到具体的系统，则每个系统又可以继续向下拆分到更细粒度的构成项目，每个节点都是一个小的“总分”结构，按这个逻辑继续向下分解，可以分为“可落地的最细粒度的成本”和“可落地的最细粒度的分摊成本”。



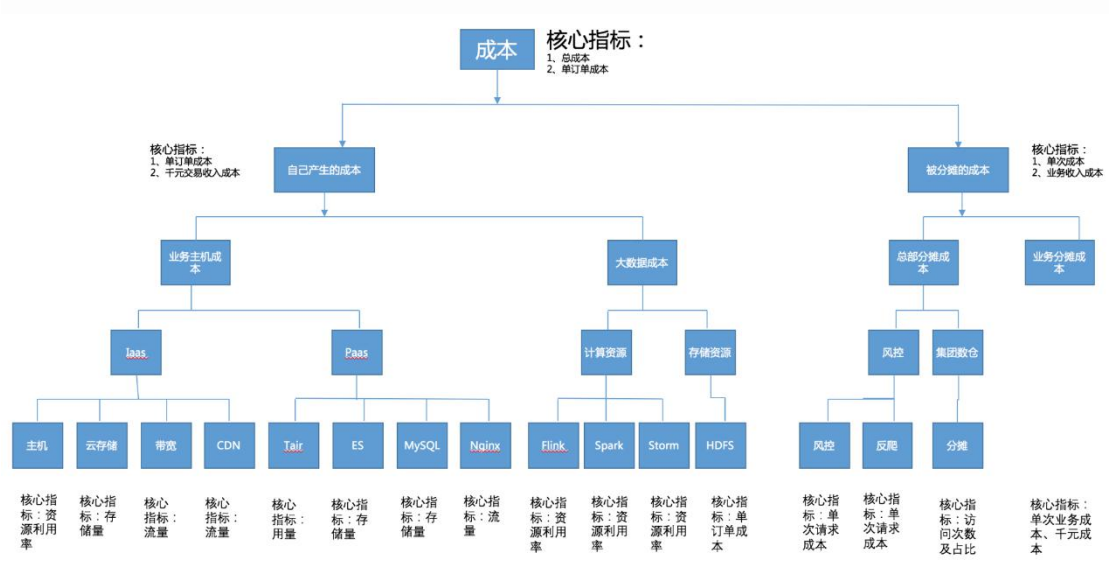
再根据开篇描述的方法，确定每个原子的评价指标，无法量化的项目都是“耍流氓”。这样就形成了一个更完整的金字塔结构，如下图所示：



- **确定方案。**根据上面的金字塔结构，每个原子指标，都需要专业的同学来评价分析，确定如何进行优化。比如，系统主机的成本，主要集中在虚拟机+存储这样的资源上，衡量的指标可以确定为“资源利用率”和“单订单成本”，为了解决“资源利用率”这个原子指标，就需要考虑目前的空闲机器是否可以下线，在线的服务是否可以优化或者合并；为了解决“单订单成本”这个指标，可以考虑分析下系统架构，跟核心流程处理有关的服务是否可以更加高效或者抽象出来成为服务中台，这样就可以释放一些”烟囱式”的建设资源，使得核心处理能力更加集中、高效。类似这样将所有的解决方案整合起来，就形成了最后的解决方案。
- **落实到人。**有了方案之后，一定要确定唯一的 Owner (主 R)，根据经验，主 R 只有一个会比较好，否则会造成“责”、“权”、“利”分割不清。在这个过程中，也是培养团队技术能力和架构能力的好机会。
- **优化指标。**不同的方案，实施的周期和代价不同，各个主 R 深入到不同专业后，会对目前的资源指标有分析和反馈，有可能理论上所有的指标都需要优化，也有可能一些指标已经很好了，这时候要甄别出来哪些资源指标的实施“杠杆率”比较高，建议应用 80/20 原则进行分析，即某些指标投入 20%的资源和精力可以解决最后 80%的核心问题，保证投入适合的工作量带来较高的产出。对于没有解决方案的资源或者实施难度过大的资源，建议果断放弃或者搁置。

3.2 实践分析框架

在具体实践中，我们可以把以上的过程，再次用一个金字塔结构来表述，如下图所示：



建立了以上的结构，就可以根据各个专业的不同，对各自的指标进行优化了，如果最细一级的指标被成功优化之后，最上层的指标一定会有下降。因为上述指标都有其各自深层次的业务、技术，甚至是财务上的逻辑，故在此把一些需要关注的概念再赘述一下.

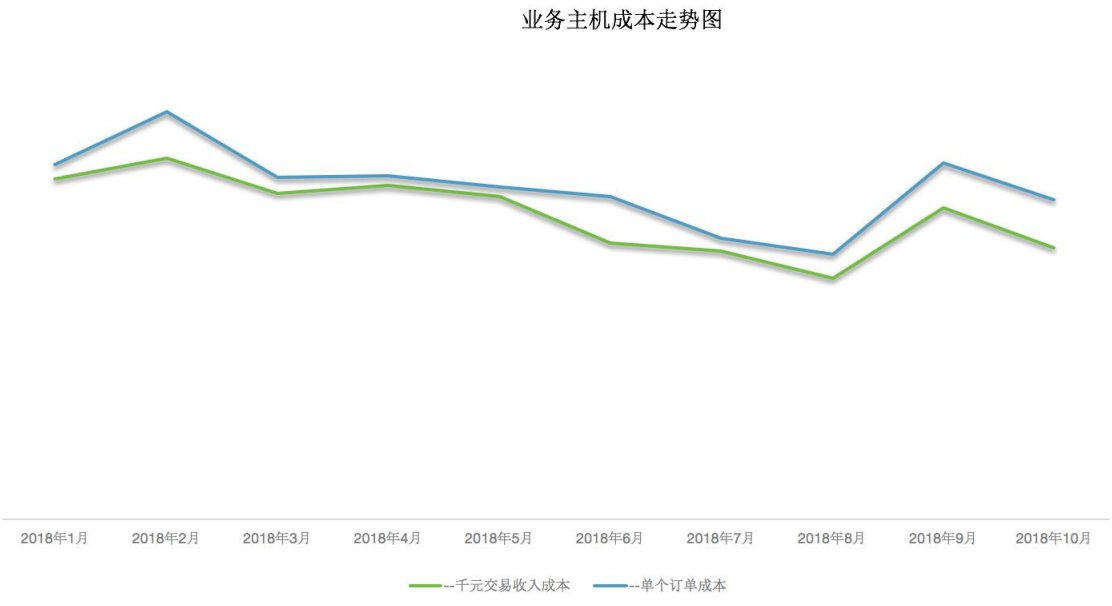
很多公司每个技术团队的机器成本，在财务上叫做“网站运维成本”（网站？听起来还像 PC 时代的概念对不对），从顶层可以分为两类构成因素，就是“自己产生的成本”（自己用的）和“被分摊的成本”（别人替你用的）两大类。跟自己有关的继续向下钻取，可以分为交易相关的资源成本（跟业务流程相关的）以及跟分析有关的大数据成本（分析、算法、决策相关）。

3.2.1 业务主机成本

大部分业务系统的团队，使用的资源成本都包含在这个部分，比如商户研发团队、订单系统研发团队、前端研发团队、供应链研发团队、营销系统研发团队、CRM 研发团队等。这些资源典型的物理载体就是物理机、虚拟机、容器资源以及对应

的机器连接的存储（DB、缓存、K-V 数据库等）资源，还会包含由于交换、存储以上资源之间的数据产生的带宽、云资源、CDN 等。

这部分资源，我们从控制成本的角度，最浅的层次，建议关注服务组（OWT）所消耗主机的资源利用率，如果资源利用率较低的主机数量较多，建议及时下线。同时，从技术方案本身来说，任何一个服务承载的业务能力和消耗资源之间，会有相对的一个“比例”或者权重。某些高利用率的服务从架构上是否可以重构、解耦或者改造，也非常有利于节省资源。这块内容到餐技术部在过去一年的工作中，对于核心、非核心的服务都进行了梳理，对于其中可以优化的服务也进行了部分重构。相比年初，很好的降低了资源的成本，业务主机成本的两个主要指标的变化情况如下（备注，后续由于新增其他业务导致成本略有上升）：



3.2.2 大数据成本

数据行业在互联网的应用目前已经较为成熟，行业主流的数据处理架构都是Yarn 2.0 或者类似框架，核心的资源消耗主要基于 Container（Vcore+Mem）的计算资源+基于 HDFS 的存储资源消耗这两部分：

第一部分，是存储资源的消耗，行业通用的模型是基于物理 HDFS 或自研的类似存储引擎，这部分主要是指离线 ETL 用来按分区（一般是按时间戳）进行存储的资源，由于数据仓库的核心理念之一是保存“所有”的数据，并在此基础上按照维度建模理论对数据进行预汇总、加和。但是，由于对于模型建设本身的理解深度不同，故在基础数据之上的数据冗余，在很多数据研发人员看来是理所应当的，进而导致存储资源的快速膨胀，这是每个数据团队在管理过程中面临的难题。

在此，到餐研发团队主要采取了两种手段：

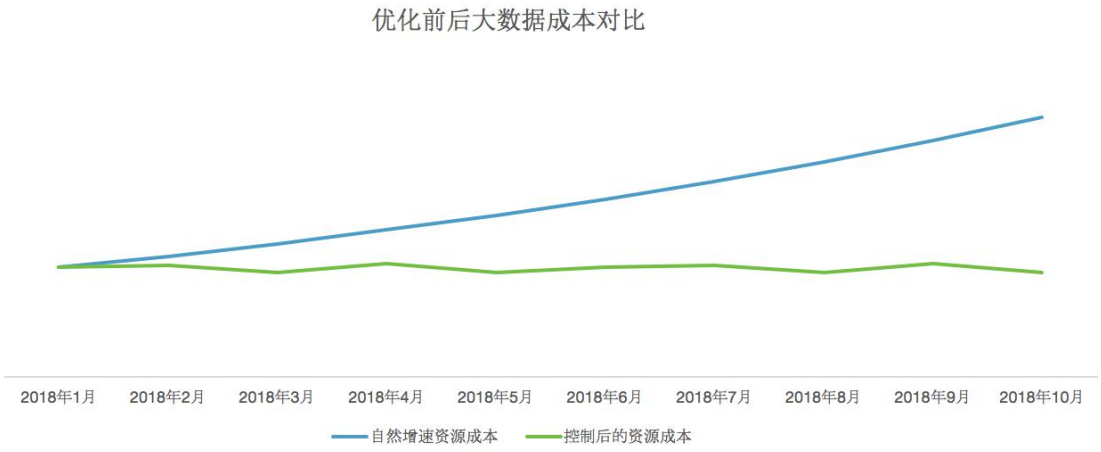
1. 对于数据模型的热度进行了分级，把数据分为冷、温、热数据，对于需要保留的数据才保存在生产环境的 HDD、SSD 中，对于不重要的冷数据，通过异构的方式存入其他介质中。
2. 对于数据模型本身，需要重新思考数据的价值和存储，在数据的中间层（汇聚层），对数据模型进行重构，这也是很多数据团队忽略的基本功部分。

到餐数据团队对于数据仓库进行了二次迭代，每次都基于新的业务模式，重新构建中间层以及之上的集市、宽表层，有效节省了空间。还有一种技术手段是压缩，比如流量的数据往往是存储大户，但是流量数据相对的格式比较固定，所以很多流量数据可以进行压缩或者改变其存储格式（如 map 型），根据实测可以节省20%以上的流量数据空间。

另外需要补充的，还有一部分 OLAP 存储资源，也会消耗大量资源，比如 Kylin、Elasticsearch、Druid、MySQL 等，这些数据库主要用来将基于 HDFS 上的文

件，同步到前端可以直接访问的介质上，供系统访问。这部分资源有些也是基于 HDFS 的（如 Kylin、HBase），有些需要单独的存储介质，也需要关注其膨胀速度以及存储周期。

第二部分，是计算资源的消耗，主要满足基于复杂规则的分析或者机器学习算法中的计算，也就是实时 ETL 计算和离线 ETL 计算的场景（代表性的引擎如 Storm、Flink 的计算还有 MapReduce 的计算）。这部分计算消耗的资源类似于业务系统，可以参照业务系统的“资源利用率”确定几个指标，进行机器优化或者算法逻辑优化。



3.2.3 分摊成本（一）风控及反爬

在某些公司里，某个技术团队开发的内容，有可能为了服务其他团队业务，比如前文中提到的风控、反爬、广告等，会为各种业务提供基础的技术能力。这时候就涉及到一个重要的概念“分摊”。分摊有两种规则，一种是按“实际用量进行”，另外一种是按照“使用比例”进行，这两种模式之上，可能还有混合计费模式，

即“按照实际发生的比例进行整体费用的分摊”，做成本控制时，就要清楚地知道这部分成本是按哪种逻辑来进行计算的。

在风控及反爬的实践中，美团的风控及反爬按照整体风控技术团队的总体成本，按比例分摊给业务团队。所以作为业务团队，如果试图降低这部分成本，也要关注两个组成项：一是自己使用的风控及反爬的原子业务数量的绝对值，对每天风控及反爬的总体请求次数是否合理需要进行判断，以保证自己的业务请求量不增加；二是自己业务使用的比例。需要跟相关技术团队一起进行分析，以防止某些场景下，自身业务使用的绝对值下降了，但是因为其他业务绝对值下降的更快，导致自己比例反而上升，进而导致成本上升。

3.2.4 分摊成本（二）安全数仓成本

为了保证各个业务团队之间的离线数据交换，美团集团层面建设了安全数据仓库，用来满足跨团队之间的数据交换。这部分的费用也按照实际发生的资源占比进行统计，所以同理，为了降低成本，需要关注两个组成项目：一是自己使用的数量，从架构设计上能否将相关数据模型的效率提升、降低空间是关键因素；二是自己

的使用资源在整体资源的占比，这时候也需要跟相关团队一起努力降低总成本。很多公司的技术团队，也有类似的数据共享仓库或者共建仓库的概念。

3.2.5 分摊成本（三）广告成本

很多互联网公司都有做广告业务的技术团队，广告的形式主要有按点击收费 CPC，按时长收费 CPT 等等，这部分分摊的逻辑同上述两者，也是按最终的总费用中的占比进行分摊。但是这块有一个需要关注的点是，由于广告的业务逻辑并不在到餐自己的业务方，也就是说归到餐研发团队可以控制的部分较小，故在这个过程中需要建立有效的评价体系，来衡量广告分摊的费用，在此采用的指标是“千次曝光成本”和“千元广告收入成本”，这里仅供大家参考。

3.2.6 其他成本

除了以上梳理的项目之外，每月还会有一些新增的成本项目加入进来，团队要保持足够的关注。在实践中会发现某项成本在个别月份突然升高，这时候就要找到是新增加了项目，还是某个指标在业务或者算法上有所调整。

4. 检查（Check）

在这个阶段，建议关注以下结果：

- **规定动作检查。**规定的方案是否执行？相关的同学是否按照规定的动作进行了相对应的行动？这个阶段只关注过程不关注结果，而且更多的是关注执行人、配合方、时间点，用项目管理的思路来运营。
- **结果评估。**之前梳理出来的指标是否得到了优化？这个过程是在验证结果，各项指标中得到优化和未优化的都要整理出详细的 List，有些指标如“资源利用率”是立即可以查看结果的，有些结果是需要周期性的时间才能获得。在这个基础上可以继续深入反向思考，按“指标定义是否有问题->方案制定是否有问题->执行人是否有问题->配合方是否有问题”这个流程来进行评估。
- **系统问题定位。**在这个过程中，可以做到小范围闭环，建议针对某个指标的优化方案可以设计多套，方案 A 不行马上迭代成方案 B，快速试错，找到合理的方案。
- **修正标准动作。**在执行的过程中，很多方案和动作，都是在一线现场发现和修正的，不需要等待大规模复盘的时候再提出问题和总结，主 R 要具备这样的意识，在执行过程中多说多问，找到关键要素，相信每个同学都有过这样的经历。经历过某个完整项目生命周期的同学，往往也是团队内成长最快的骨干。

在到餐研发团队的实践中，业务系统的指标定义上也有类似的经验可以分享。开始进行优化工作的时候，定义了非常多的的项目和指标，比如业务主机分为云存储、带宽、CDN、Tair、Redis 等等，关注到每一项对于 RD 投入的时间和精力都是巨大的损耗，后来经过反复跟相关兄弟团队确认，向上抽象了一层“服务组的资源利用率”，这时候就不需要关注太多细碎的项目，而只关注与这些服务有关机器的使用情况，因为机器会自然的消耗 CPU、内存、带宽、CDN 等，这样可以有效节省运营的时间成本，把精力集中在优化机器和优化服务架构设计层面。

5. 复盘总结，继续迭代(Act)

- **定期复盘。**复盘是一个非常重要的能力，个人以为，复盘总结的能力在某种程度上也代表了自己的“抽象能力+思考能力+管理能力”，关于复盘的方法论书籍很多，这里不再进行赘述。在这个阶段，个人建议关注的点在于两个“知道”：“知道自己不知道”，通过复盘掌握了成本优化的方法、框架、方案、团队素质、结果；“不知道自己原来知道”，通过一些结果，知道了自己原来一直是在正确的道路上还是在错误的道路上前进，把带有“运气”成分的成功，升华成为一种未来的“习惯性成功”。
- **形成报告。**让第一次看到这个报告的人，也能通过 1-2 次实践，学会成本优化这件事。
- **迭代认知。**将之前的过程开始深化和迭代，也是再次进行 PDCA 的过程，反复打磨自己的抽象能力、思考能力、管理能力，使自己工作深度、广度的 ROI 继续提升。在迭代过程中，总会有一些惊喜和收获。从个人来说，原来以为成本项目仅仅是个管理项目，在不断通过技术手段取得成本优化的过程中，收获了对架构、技术的理解，并且很多时

候需要用创新的手段来解决前人未曾突破的问题，另外还收获了 7 项跟架构升级、数据压缩、技术处理有关技术专利，也是技术能力提升的一个佐证。

总结

成本优化这件事，有可能被阶段性忽略，但是重要性一直存在。到餐研发团队通过将近一年时间的运营，帮助公司节省了几千万的成本。这个过程有时候枯燥，有时候让人兴奋，有时候又让人懊恼和沮丧，某些时候其实是在拷问自己一个问题：“保证业务不停的前提下，敢砍掉多余的机器吗？”在管理越来越精细化的今天，相信更多的有识之士也有一些需求或者进行了一些实践。期待跟行业同侪一起，在保证技术能力和满足业务的前提下，更加合理使用资源，节约公司成本，不断提升研发团队的效率，希望本文能给大家带来一些启发。

数据分析

八、美团点评运营数据产品化应用与实践

背景

美团点评作为全球最大的生活服务平台，承接超过千万的 POI，服务于数量庞大的活跃用户。在海量数据的前提下，定位运营业务、准确找到需要数据的位置，并快速提供正确、一致、易读的数据就变得异常困难，这些困难主要体现在以下方面：

- 取数门槛高，找不到切合的数据，口径复杂不易计算，对运营人员有一定的技能要求，人力成本增大；
- 数据处理非常耗时，缺少底层离线数仓模型建设和预计算支撑，Ad-hoc 平台查询缓慢；
- 数据不一致，不同渠道口径不一致，缺少对杂乱指标的统一管理；
- 数据反馈形式不友好，缺少数据可视化的形式，无法呈现趋势，继而影响业务人员对多维、降维、对比等情况的进一步分析操作。

因此团队提出将运营专题数据产品化，首先分析面临的一些问题和挑战。

挑战

① 服务业务能力

数据模式是需求驱动导向，这就导致数据最初只支持了少数团队，而更多有个性化需求的业务团队就无法被支持。

② 存储、计算、研发成本

没有统一的规范标准管理，造成了重复计算的资源浪费；数据的层次和粒度不清晰，使得重复存储严重；同时，工程师需要了解研发流程的整个细节，对研发的时间和精力成本造成浪费。

③ 数据标准不统一

业务指标繁杂，即使同样的命名，但定义口径也会不一致。例如，支付用户数就有多种定义，由此带来的问题是，都是支付用户数，应该用哪个？为什么数据都不一样？

④ 业务分析响应能力

即使拥有健壮的数仓模型支撑，但最终能否快速响应多维计算，进行对比分析，同时做到数据可读，都是对产品交互和服务能力的一种挑战。

针对以上的问题和挑战，开始制定建设方案。

方案

首先，构建了一个针对境内旅游运营侧全域的公共底层数据，将不同平台促销系统的数据按业务整合到一起，同时划分不同活动主题，按事件再向上聚合，做专题的数据支撑，统一数据出口。然后通过多维预计算引擎对事实数据进行预计算，构建数仓与应用的管道，从而节省计算成本，并且提升了数据互通和消费的效率，最后建设统一的数据服务中台，搭配不同端的 Web 应用。通过丰富的可视化效果，及多样的分析对比操作，快速、全面地支撑运营业务。

以下为整个产品的功能模块图：

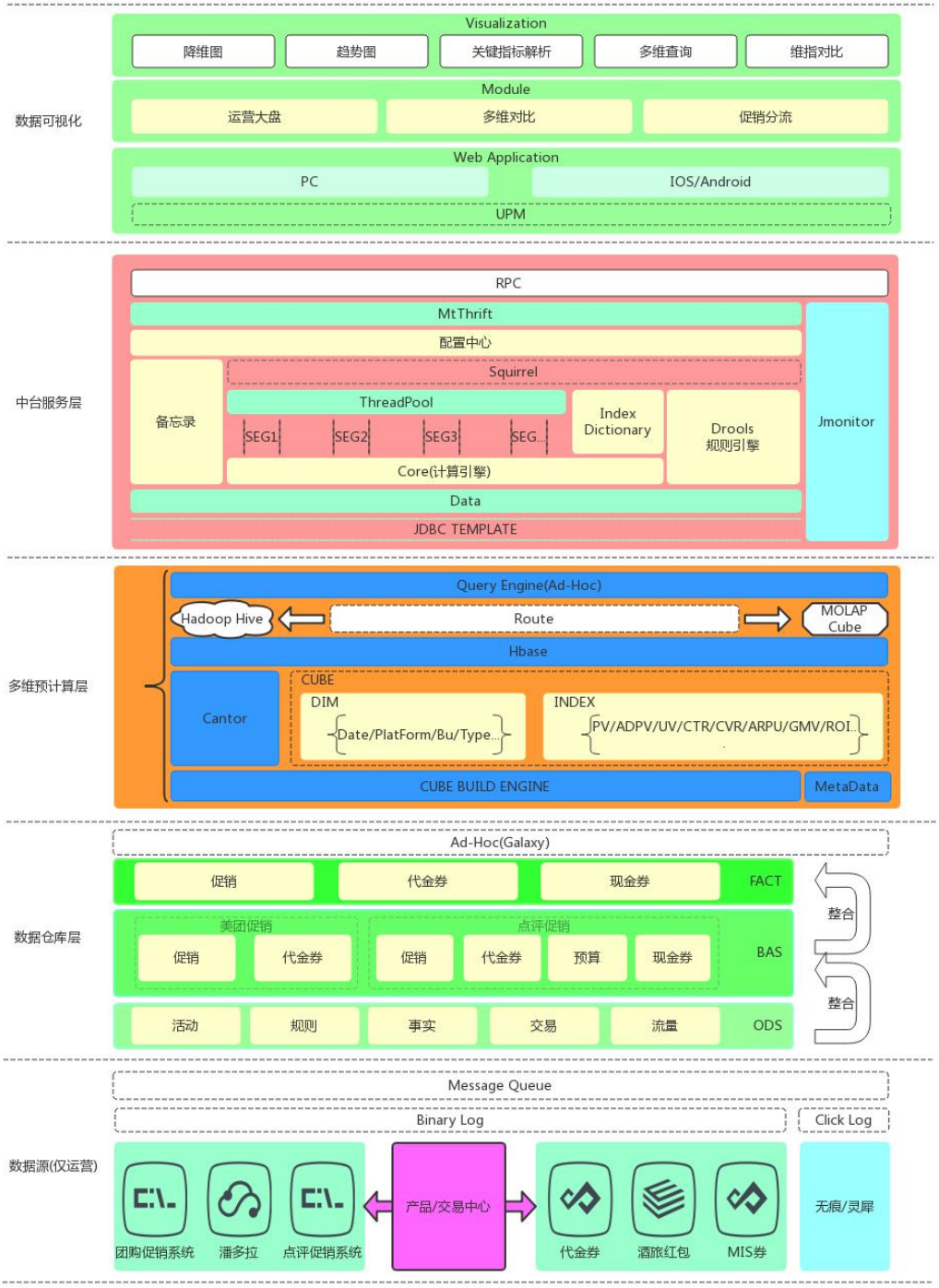


图 1 运营专题整体功能模块图

如图所示，运营专题数据的产品化，根据需要解决的问题划分了多个不同的层次，每一层除其需要面对的核心问题外，还有其领域内其它功能模块的抽象和扩展，下面将会按照层次划分逐一介绍各个模块。

数据仓库层

数据生产和消费的基础平台，是整个数据产品化过程中最核心的角色。数据仓库的模型建设，不但影响产品化的难易程度及可行性，更是数据一致性等关键问题的直接因素，所以为降低使用门槛、统一数据标准、支撑上层更合理的架构，模型的选取就变得尤为重要。

领域内常见的建模方法

① 3NF 模型

3NF 模型（又叫“范式模型”）是数据仓库之父 Inmon 提出的，它用实体加关系的数据模型描述业务架构，在范式理论上符合 3NF，是站在全局角度面向主题的抽象。它更多的是面向数据的一致性治理。

3NF 模型最基本的要素是实体、属性和关系：

- 实体：相同特征和性质的属性抽象，用抽象的实体名和属性名集合共同刻画的逻辑实体；
- 关系：实体之间的关系；
- 属性：实体的某种特性，一般实体具有多个属性。

② 维度模型

维度模型是 Kimball 提出的。维度模型多为分析和决策提供服务，因此它重点解决快速完成分析，同时提供大规模复杂查询的响应性能（预聚合），更直接地面向业务。例如熟知的星形模型，以及特殊场景的雪花模型。

维度建模最基本的要素是事实和维度：

- 事实表：一般由两部分组成，纬度和指标，通常理解为“某人在某时间某地点通过某手段做了什么事情”的事实记录，它体现了业务流程的核心内容；
- 维度表：看待事实的角度，用以描述和还原事实发生的场景，比如通过地址、时间等维度还原业务场景。

③ DV 模型（DataVault）

DataVault 是 Dan Linstedt 发起的，是一种介于 3NF 和维度建模之间的建模方法。它的设计主要是满足灵活性、可扩展性、一致性和对需求的适应性。它强调建立一个可审计的基础数据层，主要包括 Hub（核心实体）、Link（关系）、Satellite（实体属性）三个要素。

④ Anchor 模型

Anchor 模型由 Lars. Rönnbäck 提出，是 DataVault 模型的进一步范式化处理，核心思想是只添加、不修改的可扩展模型，Anchor 模型构建的表极窄，类似于 K-V 结构化模型。它主要包括 Anchors（实体且只有主键），Atributes（属性），Ties（关系），Knots（公用枚举属性））。Anchor 是应用中比较少的建模方法，只有传统企业和少数几家互联网公司有应用，例如：蚂蜂窝等。

运营专题数据如何构建

随着促销系统不断发展，平台趋于稳定，再结合各活动类型，及对需求的整理和进一步产品化，选择了 3NF+维度建模为基础的模型方法论，对数据进行合理划分和整合，构建了运营专题数据体系。

① 数据规范制定

数据规范的制定也是指标字典和服务层规则引擎抽象的基础。首先同业务达成共识，制定数据一致性标准，统一口径。同时将核心指标和个性化指标进行抽象，抽取统一规范定义，例如：月初到月末的整体交易类 GMV 和补贴类 GMV，其原子指标是 GMV，其它要素都属于指标的修饰。

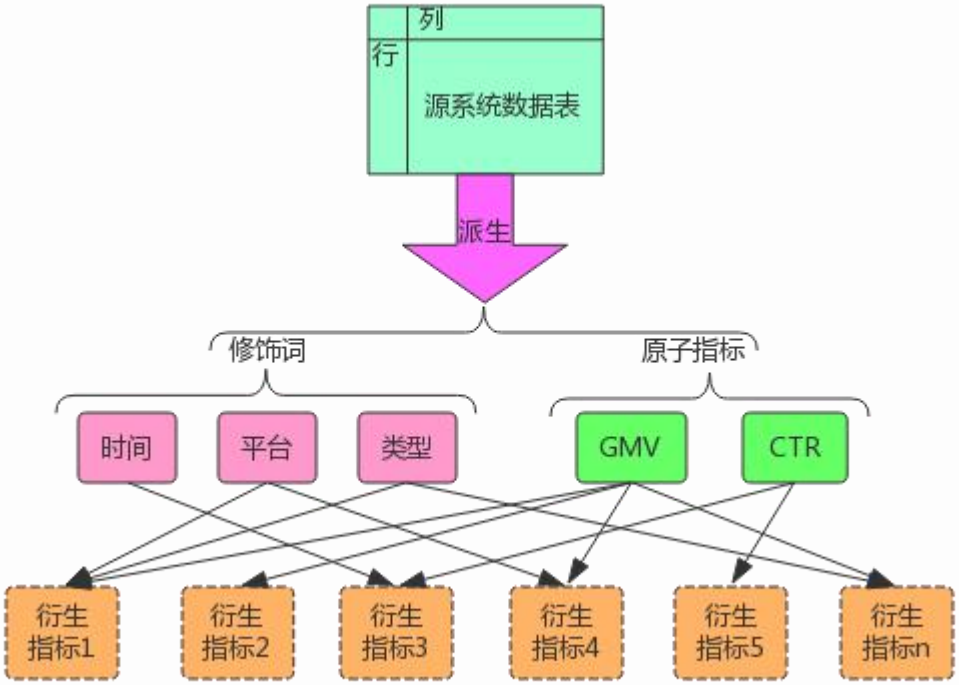


图 2 数据规范抽象示意图

② 数仓模型架构

将数据分为 ODS 层、BAS 层、FACT 层、TOPIC 层。

ODS 层主要功能

从分布式消息队列中消费 Binlog 和 Click-log，并对埋点数据进行清洗和业务库数据还原，并根据需要增量或全量同步到 Hive，同时积累历史数据并保存。

BAS 层主要功能

采用 3NF 建模方法，对整体业务进行概念抽象及适当冗余，在保证数据一致的同时将同属性实体归纳整合到同一逻辑域。BAS 层主要是为了减少数据的不一致，减少存储空间，响应业务系统的变化，避免更新异常。

FACT 层主要功能

采用维度建模方法，根据活动特点及事实场景，对代金券、现金券、促销等的事件进一步整合。经过对维度的预处理，在使用信息的时候，不但减少时间成本、提高数据的提取效率，又为用户在 Ad-Hoc 平台查询提供很好的支撑，同时它成为了上层数据应用的关键出口。

TOPIC 层主要功能

该层建设不是必须的，是针对业务中个性化诉求，根据需要建设专题数据。服务小范围业务群体和用户，用来支撑核心业务指标外的某一块个性化指标和应用。

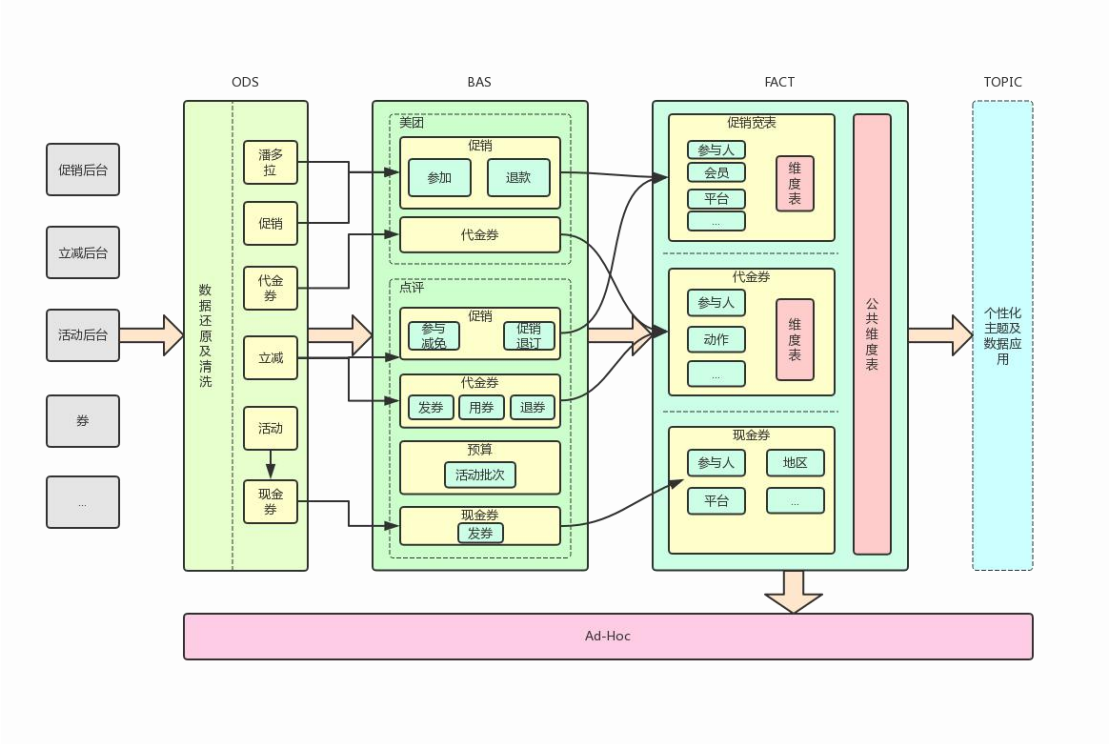


图 3 数据仓库模型图

如图所示，数仓模型整体架构图。通过构建运营专题的底层数据，针对数据一致性等问题，在数仓层面上得到了很好的解决，同时在数据提取效率上有很大的提升。数仓建设为接下来的业务支撑打好了充分的基础。

多维预计算层

预计算层是连接数据和应用之间的管道，是应用层垂直模块的专项支持。它是在 Fact 层数据之上的预聚合，强依赖于数仓模型中事实和维度的构建以及预关联。预计算采用 Kylin 引擎构建 Cube 聚合组，来解决取数门槛和数据处理耗时等问题，同是提供多维分析的能力，不但提供了新的 Ad-Hoc(Query Engine)平台，在提高查询响应的同时，又能为产品带来更流畅的交互，增强用户体验。例如：创建一个交易数据 cube，它包含日期（datakey）、用户（userid）、付款方式（paytype）、购买城市（city）。为满足不同消费方式在不同城市的应用情

况和查看用户在不同城市的消费行为，建立以下两个聚合组，包含的维度和方式如图所示：

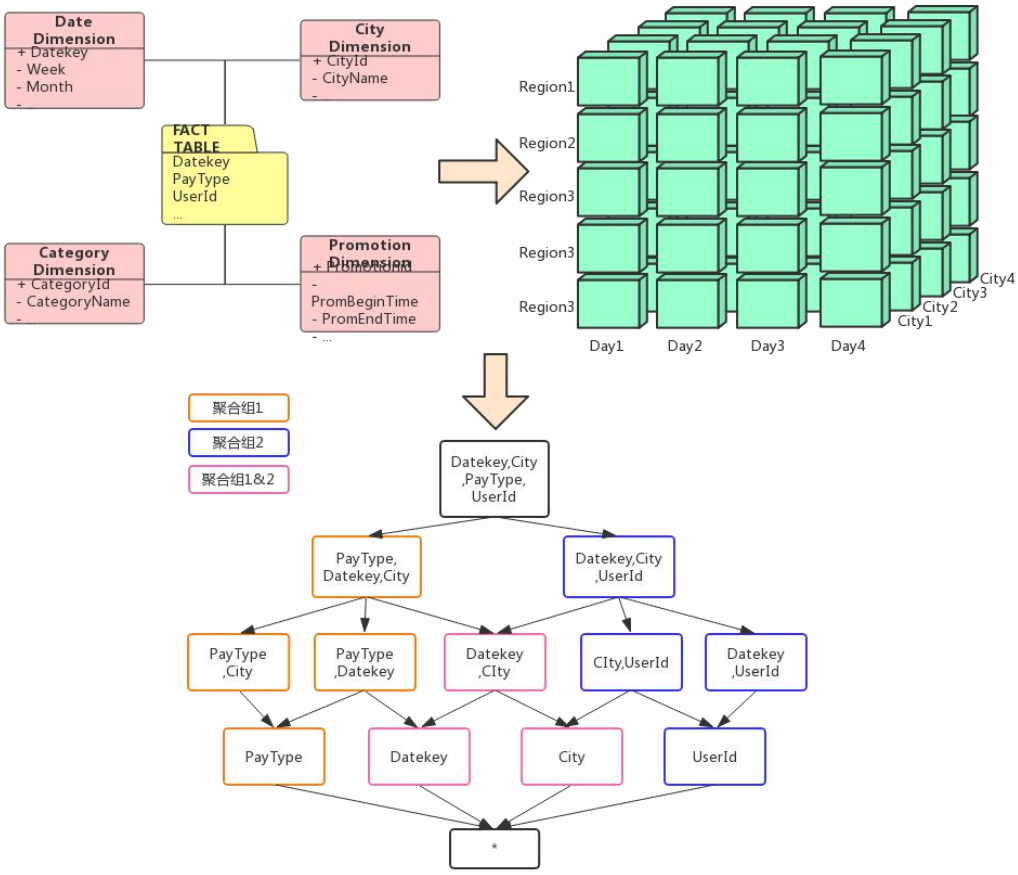


图 4 构建 cube 示例图

中台服务层

数据预计算之后，需要分别对 PC 和移动端提供计算和装载，并且要针对不同端的特定模块做特定的开发，为了应对多变的业务逻辑，以及未来的可扩展能力，需要提供可插拔的、统一的服务层，该层主要可以解决如下问题：

- 服务与预计算数据同步，数据模型的修改只影响到预计算层，同时服务层还可以完全感知预计算数据的变化，不需要对服务做开发调整，实现数据变更的同步响应；
- 服务与端解耦，针对不同端产品提供统一数据服务，避免重复开发，同时产品的迭代升级与服务层隔离，应对多变的业务发展和增长；

- 服务扩展能力增强，支持服务的横向扩展，不影响正常业务的同时提高服务能力，同时在该层实现可抽象通用操作以及规范管理。

总体架构

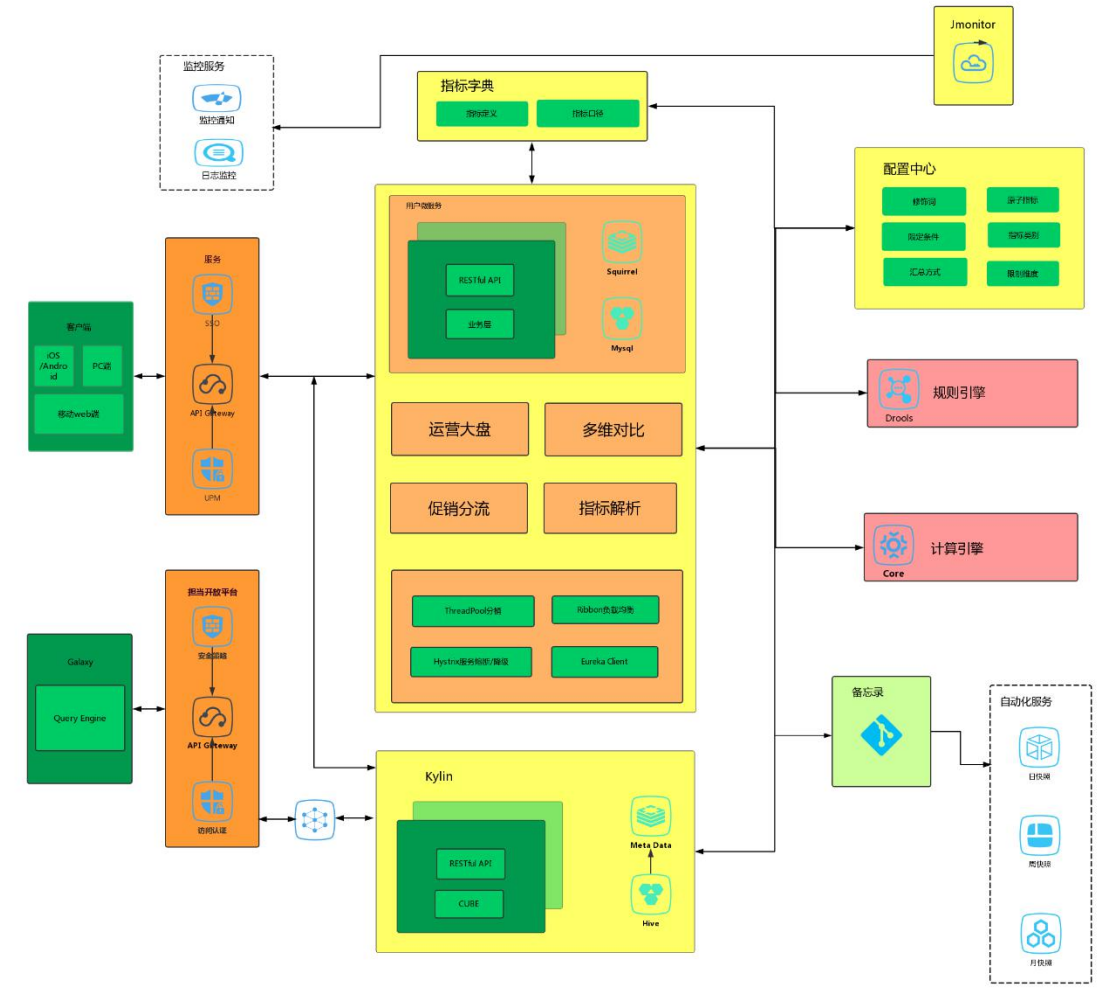


图 5 运营专题产品架构图

整个服务由独立的 Web 应用端发起请求，通过权限验证后对中台发起调用，然后读取配置中心的配置，由计算引擎对数据进行并行计算，同时规则引擎按业务线和指标修饰词等生产衍生指标，然后将引擎完成的数据按周期进行快照，存入备忘录，同时关联指标字典将数据与文案返回前台，最后按功能再对数据做可视化处理。下面分别对服务中交互的几个模块做简单的介绍。

配置中心

把系统的各类资源（比如：数据库、服务地址、缓存等）以及多个环境和具体业务逻辑（比如：业务线、平台、指标类型），按功能特性抽取出公共的控制的线头，在需要调整的时候，人为的控制系统。

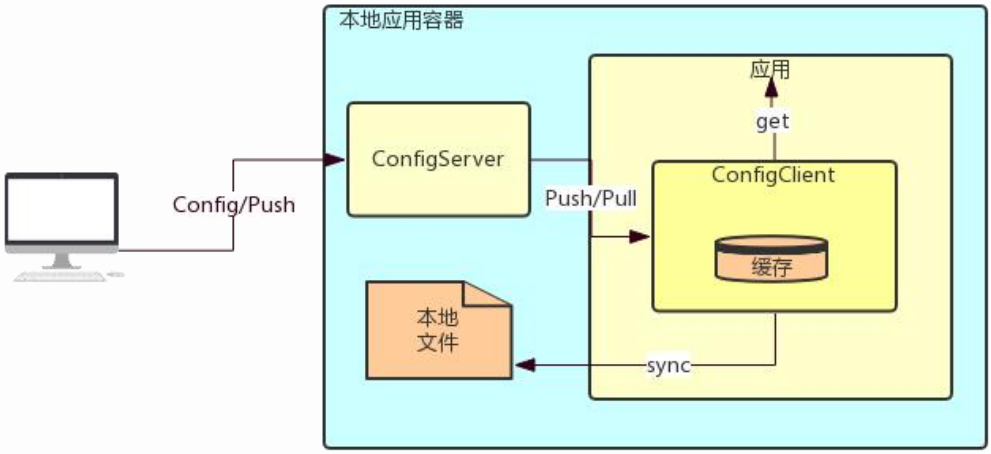


图 6 配置中心示意图

如图所示，用户通过单独的配置入口，将系统配置、优化条件判断、业务线个性化指标配置等信息提交到 Server，运行时 Client 会到 Server 拉取配置，放入缓存，并定时持久化到本地文件，方便异常中断或重启时手动或自动重新加载配置。

指标字典

公司中的很多运营部门指标定义不清晰或不尽相同，但叫法相同（文案），又或者叫法相同指标口径不同，出现一些对指标的理解不一致，含义不清等问题。基于指标字典，不但是指标命名的规范和明确，也是统一计算口径的落地，接入规

则引擎后生成关联衍生指标，即可自助完成查询和分析。可见，指标字典的建立，是数据服务平台的基础。

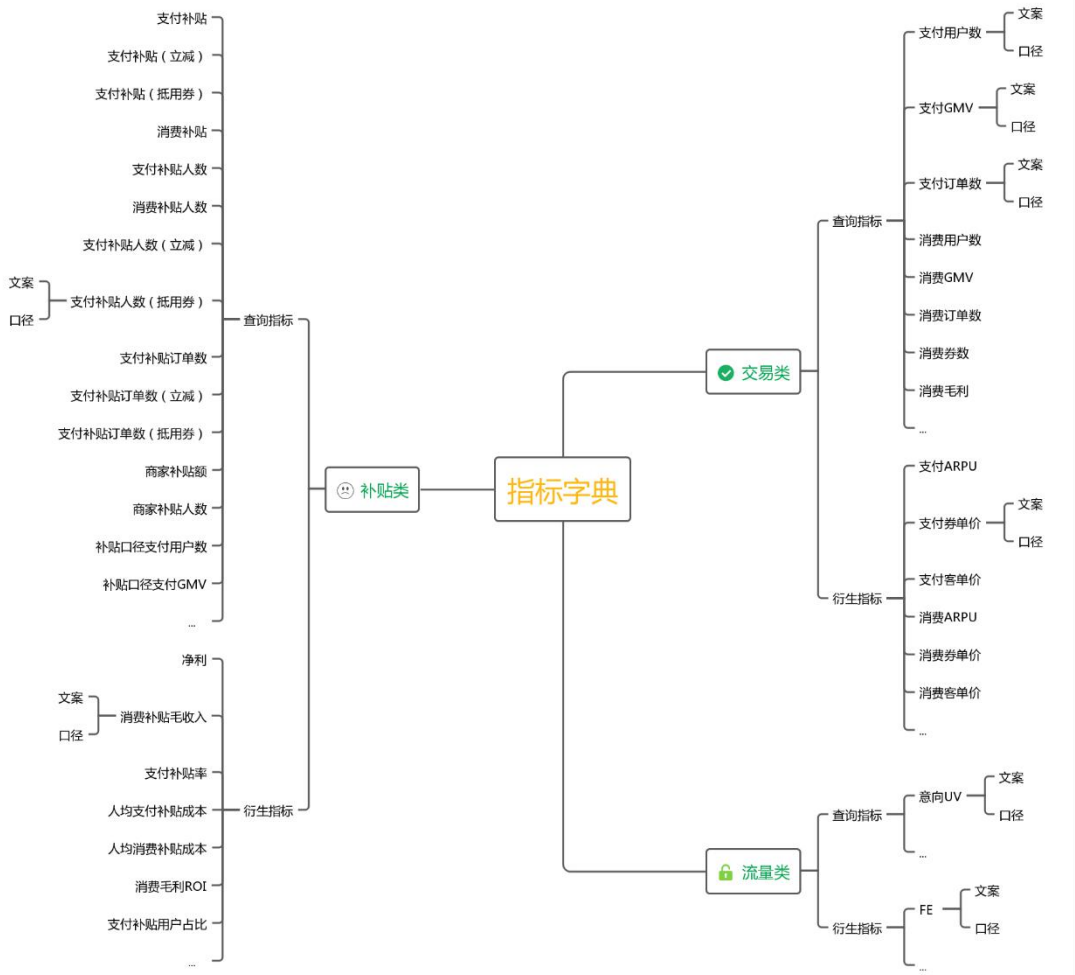


图 7 指标字典思维导图

如图所示，基于数仓中对数据规范的制定，将指标按业务线、类型、基础、衍生等划分为不同类别，并对指标名称、别名、口径等信息落地入库，进行持久化存储。

规则引擎

运营业务的特点是运营活动规则的多变，需要很多个性化配置。为解决复杂和复合的计算问题（维度和事实的交叉）并降低维护成本，将逻辑从“硬编码”中将规则抽离，然后根据不同业务线特点按修饰词进行隔离，提高应用灵活性，简化架构。

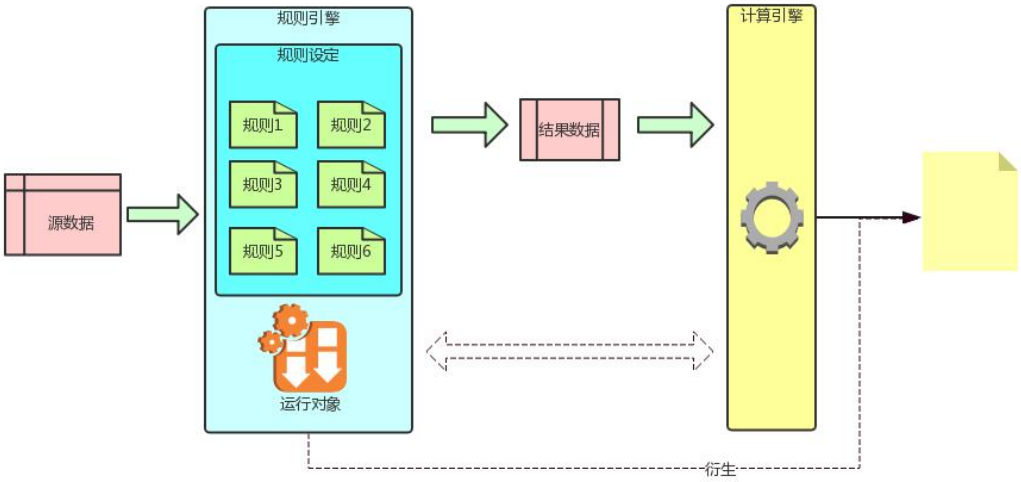


图 8 规则引擎示意图

① 数据准备规则

在应用数据计算之前把外部数据引入作为规则匹配运算的算子或数据集，例如某活动针对全部用户做发红包活动，而在活动中针对新用户发 x 面额的红包，而针对老用户发 y 面值的红包。其规则条件为：红包金额 请输入金额 ，且使用地点为 ；

② 数据计算规则

实现对业务规则的变量和参数化，按指标字典中的指标定义，转化为计算表达式，使得规则执行的结果作为其他规则条件的计算因子，生成衍生指标。

计算引擎

计算引擎（core 模块）在对数据进行处理时对数据进行了分片，分桶等优化操作，在面对多维度大范围数据查询时一定程度上提升了查询性能，计算模块的抽取实现了与业务逻辑的解耦，它只负责任务的处理和执行，可仅对性能进行维护和升级，甚至可以维护不同处理方式的多个计算引擎，无需关心业务逻辑。

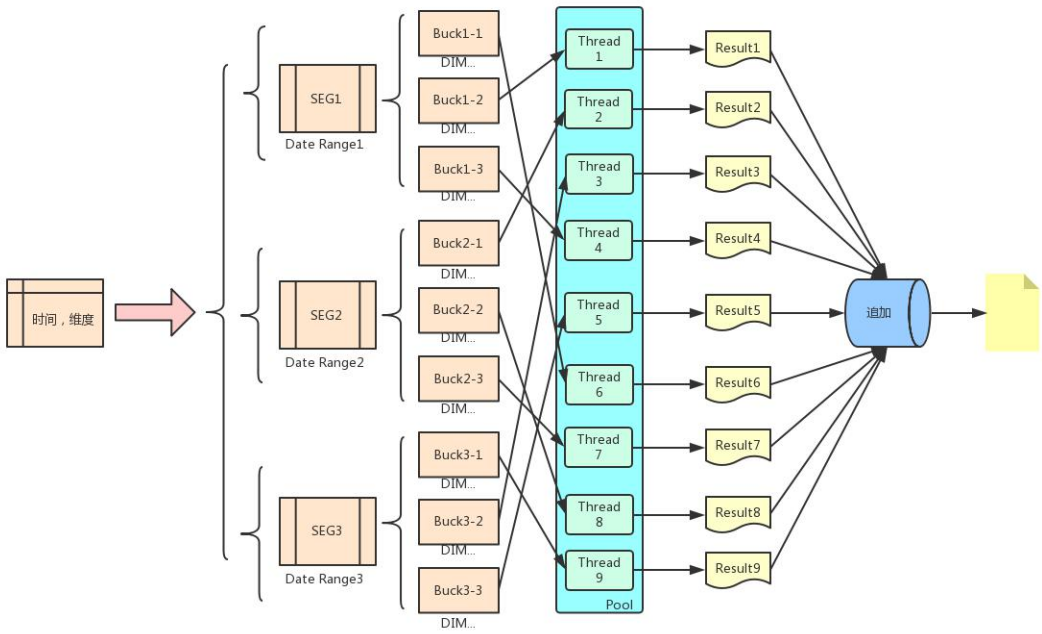


图 9 计算引擎示意图

如图所示，当引擎接收一个时间跨度较大，维度较多的数据时，会先按照时间进行横向切分，然后将切分的数据按维度组合进行纵向切割，每一组都交由一个线程进行处理，并对该结果数据进行 tag 标记，然后根据标记在前台进行整合。

备忘录

备忘录是按时间周期对数据计算完成装载后状态的快照历史，是对值和计算规则的持久化。通过备忘录可以为用户提供横向，纵向等对比分析功能，帮助用户分析趋势。简化示意图如下：

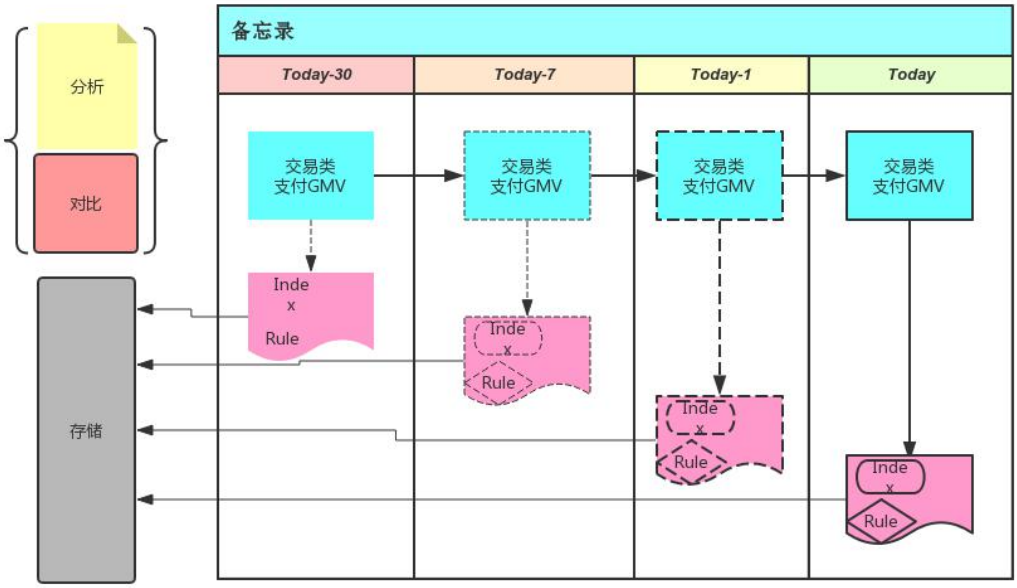


图 10 备忘录示意图

数据可视化

面对冰冷的数字，如何将数据组织起来，使其既有吸引力又易于理解？ 可视化是解决问题的一种高效的手段，数据是强大的，如果能真正理解其中的内容。 运营专题产品采用了开源的 Echarts，通过定制化开发的可视化数据，帮助用户将数据转化为可以付诸行动的见解，在提供可视化数据的同时，又为专题数据特定模块提供特定的降维，对比等线上分析操作。

趋势对比

通过维度的筛选切换，业务不同视角的核心指标趋势一目了然，不仅提供不同时间粒度同环比的纵向比对，还提供同级指标的横向比对，努力做到多角度、全方位的数据呈现。

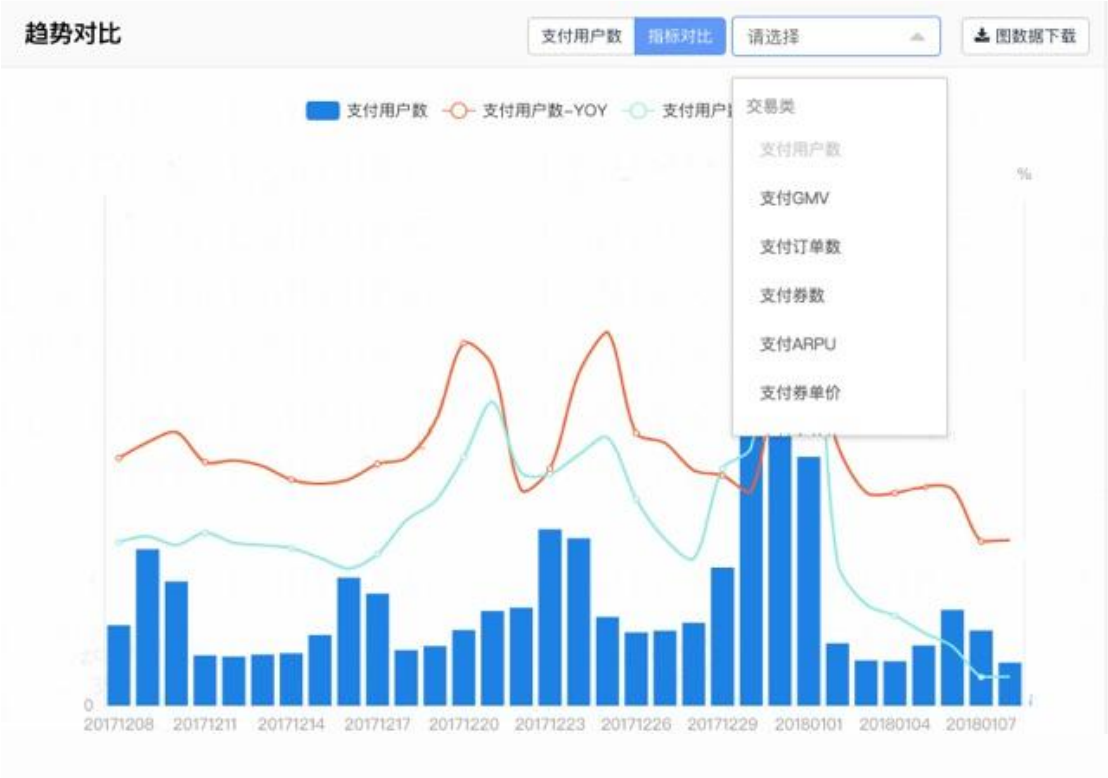


图 11 产品趋势对比图

降维操作

为更好的认识和理解数据，降低复杂度，缓解“信息丰富、知识贫乏”现状，提供了降维操作，让原本稀疏分布在各维度的特征聚敛，将某类特性更直接的表现出来。

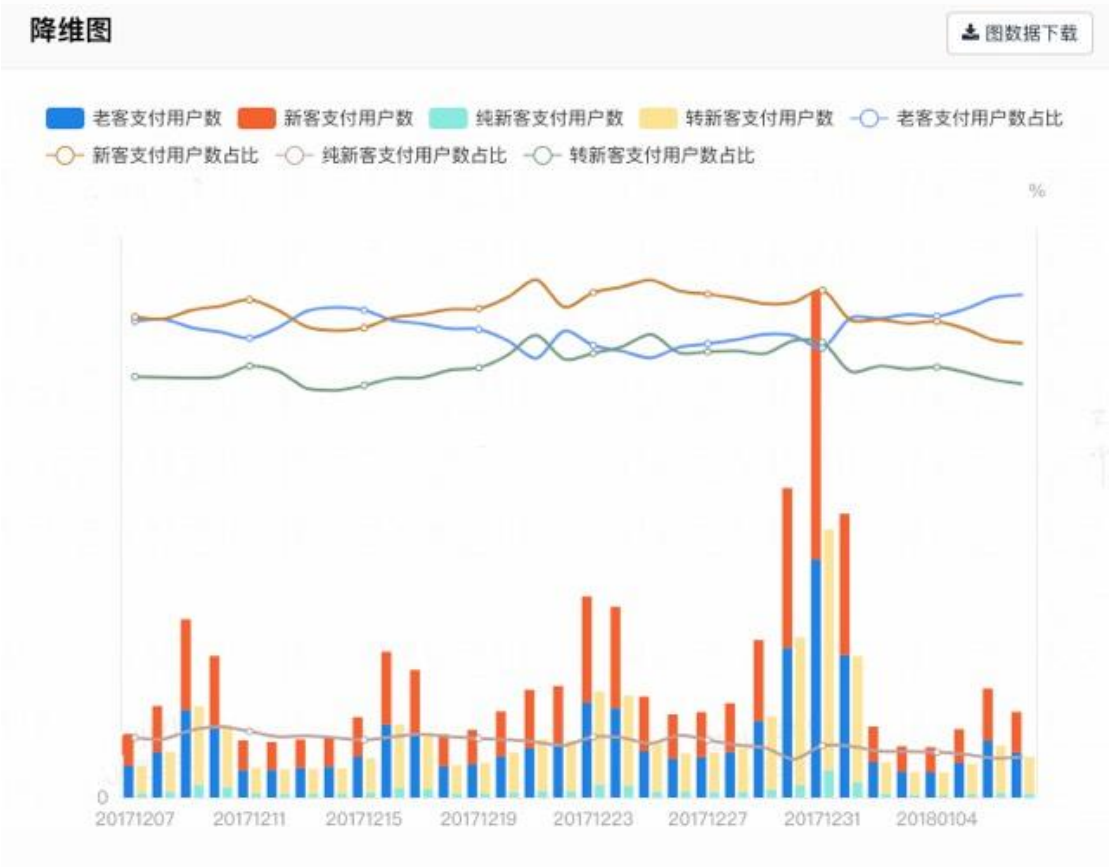


图 12 产品降维操作图

指标对比

将业务人员的线下热操作简移到线上，通过将维度压扁拉伸成纵向指标，进行多维指标的对比，并提供明细。

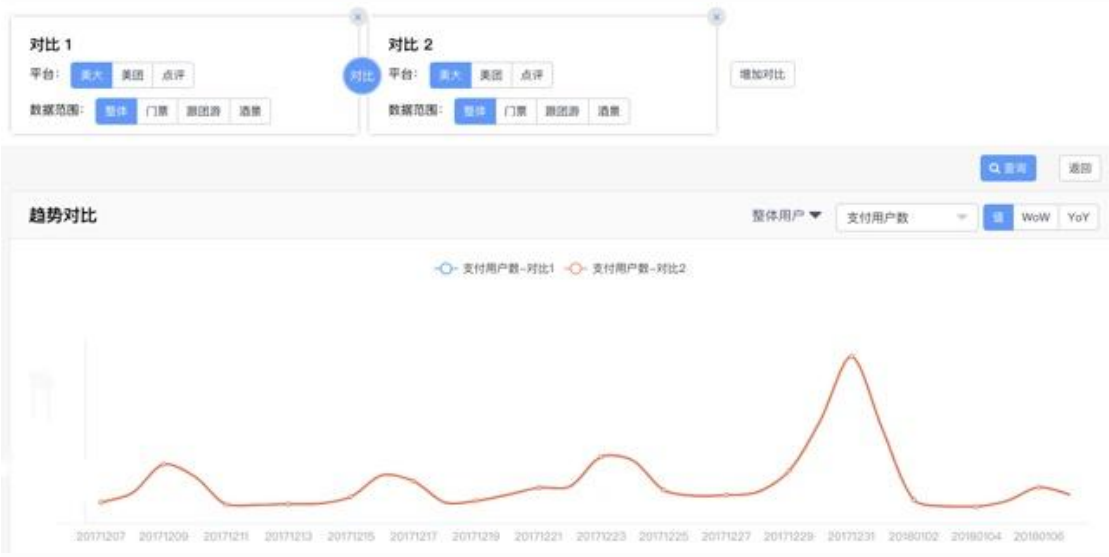


图 13 产品指标对比图

多维查询

为支持更好的 OLAP 分析，发挥预计算层的作用，还提供了关键指标解析和多维查询的功能，是产品对常规性分析的功能补充。

图 14 产品多维查询图

总结

在运营专题数据产品化的过程中，将技术转化为价值，提炼数据内容、为业务赋能是真正的发力点，为发挥数据的最大价值以及带给用户更好的体验，投入了大量的思考与实践，最终产品的生产投入为现阶段带来了以下收益：

- 数据标准统一：数据指标口径一致，各种场景下看到的数据一致性得到保障，支撑多个团队；
- 极大扩展性：服务了内部全运营业务团队，满足不同团队的个性化需求；
- 统一服务：建立了统一的数据服务和中台服务，支持灵活配置；
- 计算、存储、研发成本：增强了指标的复用、模型分层、粒度清晰，精简了数据表的落地量，通过数据分域、模型分层，节省了研发的时间和精力；
- 业务支持：丰富的可视化数据，提供多维、降维、对比等多样的分析操作，多方位全角度支撑业务。

九、流量运营数据产品最佳实践——美团旅行流量罗盘

背景

互联网进入“下半场”后，美团点评作为全球最大的生活服务平台，拥有海量的活跃用户，这对技术来说，是一个巨大的宝藏。此时，我们需要一个利器，来最大程度发挥这份流量巨矿的价值，为酒旅的业务增长提供源源不断的动力。这个利器，我们叫它“流量罗盘”。

我们首先要思考几个问题：

1. 流量都来自哪些入口；
2. 本地场景、异地场景的流量差异如何运用好；
3. 如何挖掘出适合不同品类的流量场景；
4. 是否能让不同群体的用户得到合理的引导。

所以，我们先要给流量罗盘做一个能够快速对比和衡量流量价值的来源分析功能，来覆盖流量的灵活细分及组合方式，继而找到酒旅流量增长的契机 ，为优化流量应用场景提供建议。

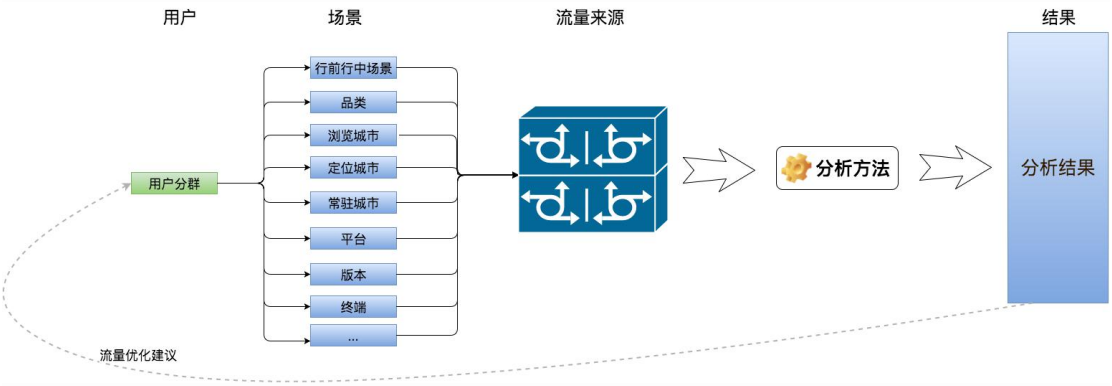


图 1 产品结构图

从图 1 可以看出，流量罗盘就是将用户、场景、流量来源进行充分组合，封装了流量来源分析方法，这样一来所有 C 端的 PM（产品经理）和运营都可以随时随地完成对流量来源的分析，继而迅速落实到流量优化的实际工作中，大大提高了工作效率。

以上数据组合每个环节的需求关键点在于：

1. 满足丰富的场景组合、灵活且能够随时满足酒旅业务的场景扩展；
2. 流量来源可以是任何一个页面或控件，甚至是组合，来源的组合要高效易用。

挑战

在建设流量罗盘过程中，主要面临以下几点挑战：

1. 维度种类较多且元素丰富。清洗后所到达主题层的与业务直接相关的 UV 流量，每天增量在千万级别。同时数量众多的维度和丰富的元素，使得总体数据量呈指数级别膨胀。考虑用户使用体验，分析结果反应时间需要控制在<=3s（天粒度）和 <=5s（月粒度）。
2. 维度的可扩展性。现有基础维度是 12 个，在此基础上做维度的扩展主要从三个方面着手。第一，添加新的正常维度；第二，新增已有维度的衍生维度，例如城市等级；第三，在已有维度中增加新元素，该元素可以与已有元素存在交集，例如酒店业务的两种不同品类共享一部分门店，这时候流量和交易需要双算。业务发展较快，要在不修改已有数据模型的前提下，快速迭代添加新的维度。
3. 入口来源的灵活性。其实入口来源属于上文中维度里面的一种，之所以在这里单独说明，是因为它非常特殊。它要求我们做到在每个版本中所有的新加入口，都可以立即分析其效果。
4. 在查询引擎中，我们在选择时间维度类型时，选择按周或按月，各个指标的值都是计算日均值（单日数据去重，跨天不去重），单日的指标值数据都是针对用户去重的，直接按周按月查询是按周去重和按月去重的，这就不符合按周按月指标的计算逻辑。

解决思路

建设流量罗盘的时候，我们既要满足新的流量分析应用需求，解决以往的短板和痛点，也要有一定的业务和技术前瞻性，给未来留有改进的空间。针对前文描述的问题，有以下几点思考：

1. 在应用接入选型 Kylin 的基础上，考虑到其维度数量的限制，数据输入的时候，可以提前对维度进行剪枝。例如酒店常有的本异地场景，观察角度具有多个层次，我们可以只采用最低层次粒度输入，然后通过后台的关系规则匹配关联，间接得到更多丰富场景的计算数据。
2. 同样是为了解决之前低扩展的问题，尽量将整个数据链条层次化，功能模块要避免高耦合的数据逻辑。
3. 想要满足入口来源的灵活配置，首先埋点要规则统一，然后抽象该规则至入口维度中，最后搭配指标的联合计算得到。

解决方案

明确思路之后，我们开始了流量罗盘制作的实践，包括流量日志采集和抽取、来源归因、主题模型建设、构建多维应用层，以及应用后台和前端的开发。

体系架构

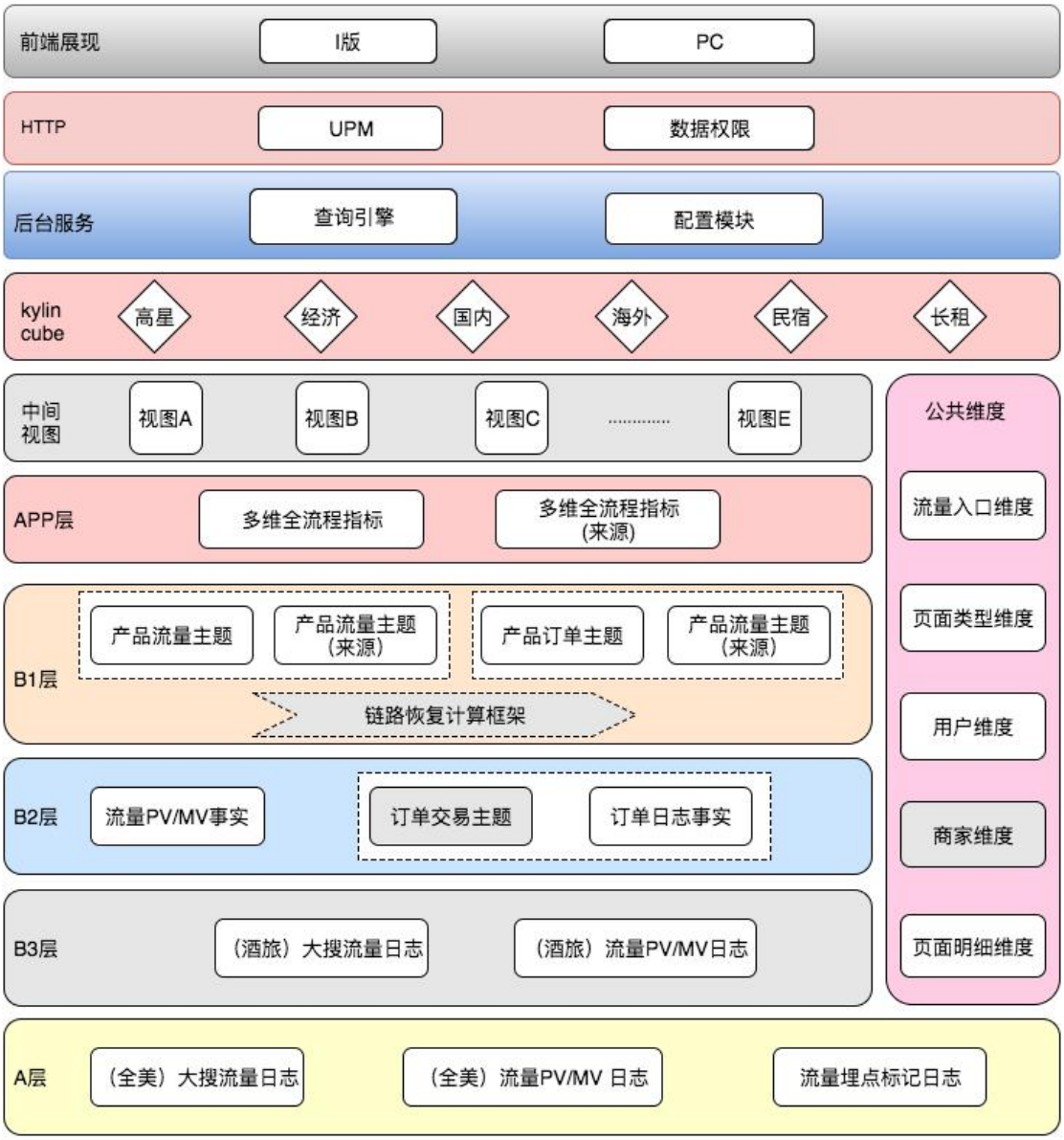


图 2 流量罗盘体系架构

如图 2 所示：

- A 层（也称 ODS 层），包含美团 App 的大搜日志、页面流量日志、模块事件日志以及描述埋点内容的信息日志。
- 公共维度，其中重要的流量入口维度、页面维度都是从具有统一规则的埋点标记日志中，抽象形成的维度。
- B3 层（酒旅基础明细层），通过对 A 层的抽取转换，初步形成只含酒旅业务所需的基础流量日志。
- B2 层（酒旅多维模型层），对已有的基础层数据和公共维度的轻加工，扩展出业务常用的维度信息，例如页面类型、商家门店、产品、城市，以及平台等。
- B1 层（主题宽表层），主题宽表层主要是对多维模型层的聚合计算，包括多个复杂业务口径的输出、少数维度的深加工，以及来源入口的增加，保证数据的一致性。

- App 层，该层是针对各自的流量应用（流量罗盘）设计的，满足该产品应用所需且具有一定扩展容量的聚合模型结构。
- 视图层，作为 App 层与 Kylin cube 的缓冲层，依靠其本身视图的特性，能够很好地解决顶层扩展、查询延时、资源分配，以及表意理解等多个问题。
- cube 层，每个 Kylin cube 是由单个视图与多个维度的雪花组合，输出计算数据给罗盘后台服务。
- 后台服务层，包含查询引擎和配置模块两部分的内容。处理前端的查询请求。
- 权限层，对各个业务线分平台和终端控制权限。
- 前端展示层，与用户交互并提交用户的查询请求。

具体方案

公共维度

从图 2 可以了解到，公共维度从 B3 层一直贯穿到视图层，最终形成顶层 Cube。公共维度的主要作用是将抽象的埋点规则、业务规则，以及各项标签模块化，能够被各层数据直接或间接调用，从而保证数据的一致性。

图 3 举例说明的是，页面类型维度、页面明细维度，以及流量入口维度的来源。

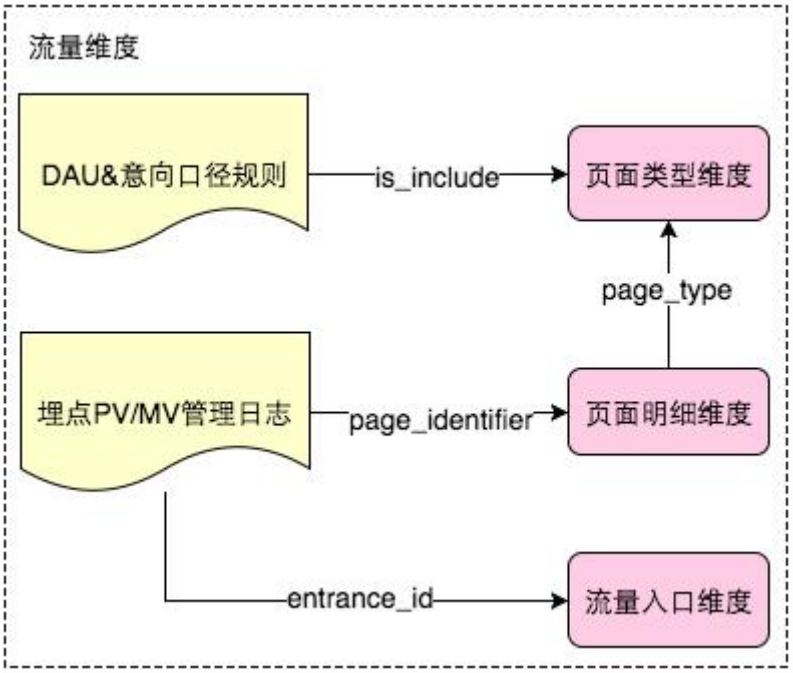


图 3 公共维度来源图

如图 3 所示：

- 页面类型维度，抽象于业务方对页面的定义、对 DAU 和意向等口径的定义（文档日志）；
- 页面明细维度和流量入口维度，来源于平时规范化的埋点方案文档日志，明确标示业务页面范畴和业务入口的定义。

主题宽表层

主题宽表层的主要作用是在满足一定就绪时效和查询效率的前提下，尽可能地向下游输出丰富维度、标准业务口径、具有强扩展性的数据模型。

这里举其中一个例子来回应下前文提出的维度扩展性问题。

场景：如何在不更改主题模型结构和让下游无感知的前提下，满足多品类（相互重叠）的添加和扩展？

针对该场景，我们采用具有强扩展性的 Json 串作为该维度内容，同时封装该复杂的口径处理逻辑，使其具有全局复用性和扩展性，并保证口径的唯一。

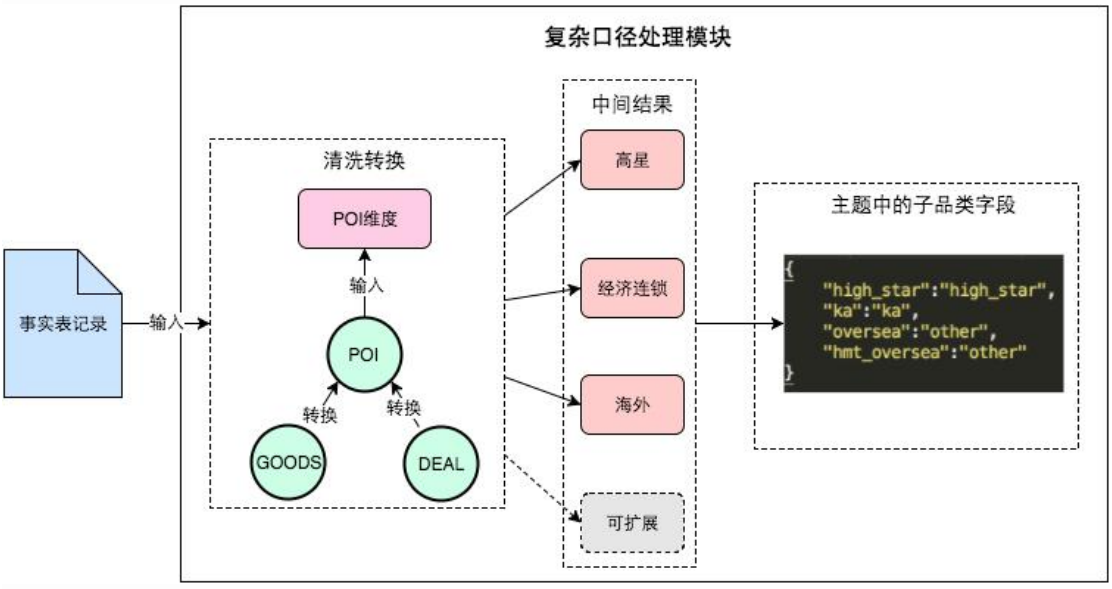


图 4 口径处理模块

上面的部分讲解了主题层是如何设计的，接下来将具体描述下为了达到所设计的模型，我们的 ETL 从（酒旅）流量日志开到上层主题过程中，是如何流转处理的。

因为流量产品主题和订单产品主题的较为类似，故以流量产品主题为例，表示基础层到事实层再到主题层的一般流程。

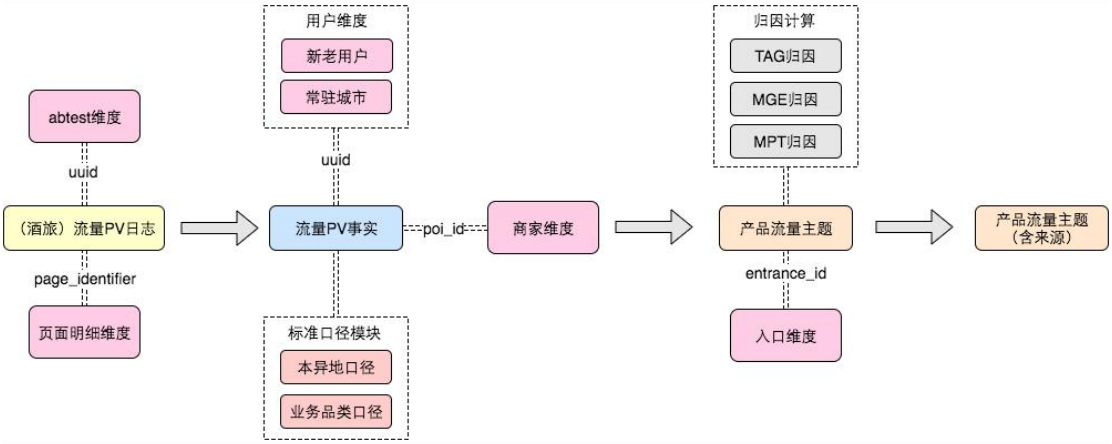


图 5 主题模型计算流程

如图 5 所示，数据链路中的各个节点功能之间相互独立：

- 日志到事实是保留基础流量信息的前提下，提取和分区主要流量页面，同时附加 A/B Testing 策略维度；
- 用户维度的输入是用户，输出是包含但不限于新老、常驻等具有用户标签属性的扩展维度内容；
- 标准口径模块的功能是统一管理口径处理逻辑、统一输出口径标签维度，具有全局适配性；
- 从浏览事实到产品流量主题，除了保留了原有的信息外，就是增加了很多需要二次计算的补充扩展维度；
- 归因计算模块，包括 tag 标记归因、页面归因和模块事件归因，根据不同的场景匹配不同的归因方式，得到不同的入口来源；
- 对产品流量主题进行流量入口的加工，产出具有易于分析的入口主题，两者的就绪时间相差较大，所以分步进行。

数据应用层

应用层的功能是向系统后台提供方便的、直接满足用户需求的查询通道，本文采用的是 Kylin cube。这样系统后台可以根据约定好的查询逻辑，直接调用 kylin 接口，得到数据。

所以，本文认为衡量应用层和后台系统对接的效果有三个方面：

- 后台查询数据的逻辑简单。
- 屏蔽业务逻辑。
- 业务扩展，查询逻辑不变或者较少更改。

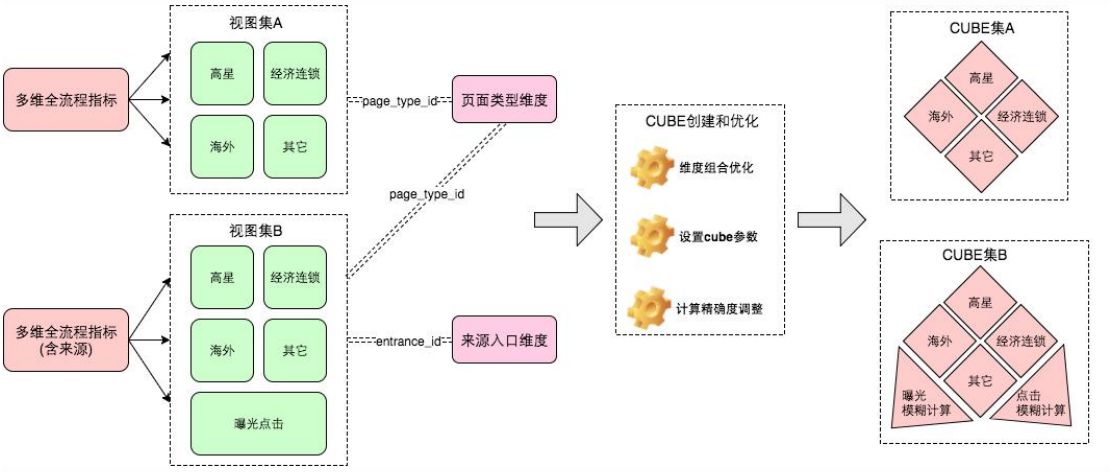


图 6 应用层计算流程

如图 6 所示，为了满足业务需求、用户查询体验，以及较好地与下游对接，做了几方面工作：

1. App 层分为两部分，不分来源和区分来源，因为两者就绪时间不同，在数据一致性的前提下，采用先到先展示的方案；
2. 在 App 层和 Cube 层之间，加入中间视图层，是为了隔离业务变动、口径变动，以及数据更改对下游使用者的影响，同时也整体增强了扩展能力；
3. Cube 创建过程中的维度优化和参数设置，目的主要是为了提高 cube 查询和 cube 生产的效率；
4. 因为 Kylin 本身的原因，Cube 也与视图一一对应，其中曝光和点击采用模糊计算，经过研究对该类指标采用模糊方案，性价比最大。

后台服务层

后台服务提供查询引擎和配置模块。

由于酒旅各个业务线关注的来源入口的不统一，各个入口支持的本异地、品类等维度的不同以及各个入口能支持的指标不一致，造成了各业务线的差异性，流量罗盘针对这种差异性，设置了针对不同的业务线和平台的各个入口分别进行配置。包含入口的指标计算都会基于来源配置的内容生成查询服务。来源入口的配置包含（不限于）以下几方面的内容：

- 来源入口对指标的支持。
- 来源入口各个指标对品类（其中酒店包含高星、经济连锁、海外等）的支持。
- 来源入口各个指标对本异地的支持。

配置模块也是基于权限控制的，配置模块是分业务线和平台配置的，只有具有对应权限的用户才能增加和更改来源入口的配置。只有在配置模块配置的入口信息才会展示在对应业务线对应平台的入口维度选择中。其中，配置模块交互如图 7 所示：

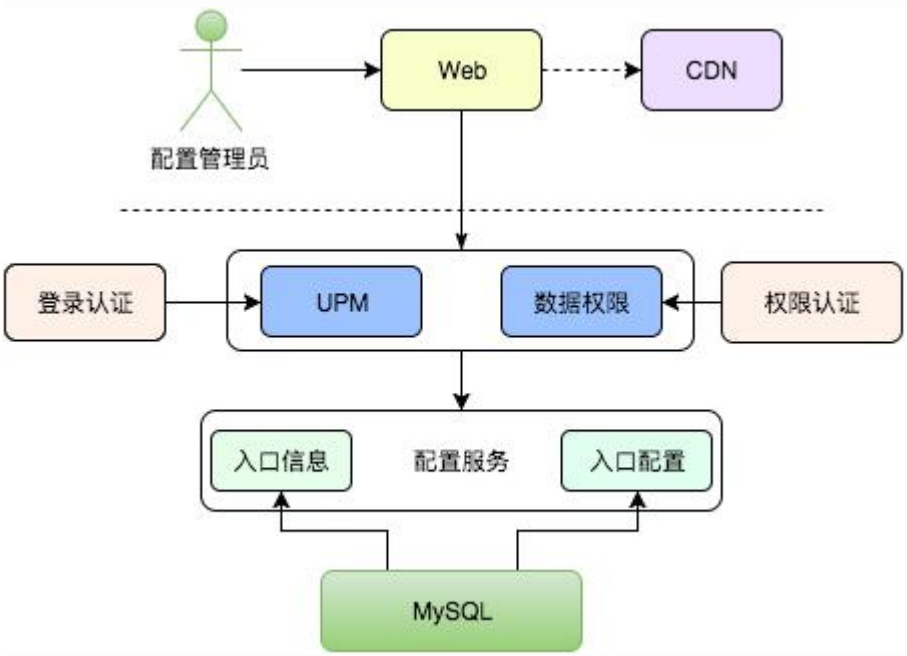


图 7 配置模块交互

前端展示登录用户具有权限的业务线和平台的入口配置信息。当增加一条配置信息时，配置服务会从入口信息维表中读取对应业务线和平台下的所有可配置的入口信息，配置完入口对应支持的指标及各指标支持的基本维度信息后会将配置信息存入入口配置表。当入口的状态修改为在线状态后，在查询引擎的入口维度中才可以展示此入口维度。

查询模块负责根据用户提交的查询请求中的维度信息，执行查询，返回前端结果。用户提交的请求中如果包含入口信息，会查询配置模块中对应入口配置的指标中是否支持对应维度的查询，若不支持将不对此维度进行限制。如果前端提交的查询请求中不包含入口信息，查询的指标为各业务线设定的默认的指标信息。

查询引擎也会受权限的控制。此模块根据业务线、平台和终端三部分共同配置权限，只有具有某业务线下某个平台和某个终端的权限时，用户方可进行查询服务。其中查询请求流程如图 8 所示：

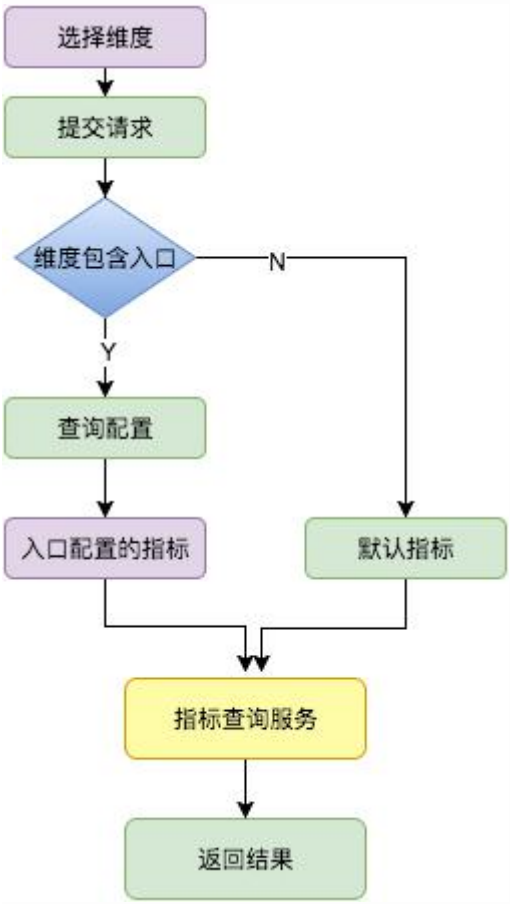


图 8 查询服务流程图

当用户选择的时间维度是按周或按月的查询时，各个指标的值是计算日均值（对于单日数据去重，跨天不去重的逻辑），单日的指标值数据都是针对用户去重的，直接按周按月查询是周去重和月去重的，这就不符合按周按月指标的计算逻辑导致数据查询结果存在差异性。为了解决数据准确性和按周按月查询数据量过大导致的查询效率的问题，将 Master-Worker 的多线程的设计模式应用于按周和按月的指标查询中。其中任务拆分指标计算的过程如图 9 所示：

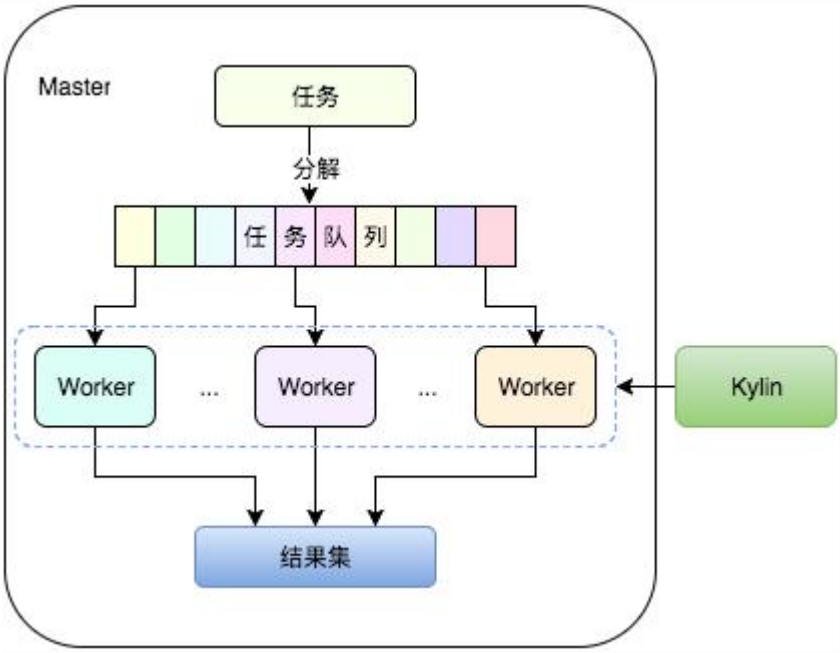


图 9 任务拆分指标计算

如图 9 所示：

- 用户在选择维度之后提交每个指标计算的总任务。
- Master 将总任务简单的按时间维度拆分成对每个周或是每个月为维度求日均值的查询任务放到任务队列中。
- Worker 进程队列从任务队列中获取任务、执行任务并将任务结果提交给 Master 的结果集。
- Master 将各个子任务的指标计算结果进行汇总返回。

目前，涉及到的指标都相对简单，之后如果涉及到较复杂的衍生指标，也可以将指标的计算拆分成对基础指标的计算，计算完成后再将基础指标的计算结果合并计算衍生指标的值。

评价指标

整套数仓设计和实现是一个需要长期持续优化迭代的过程，所以需要一些具有客观评价系统好坏的指标。

评价指标可以分为两类，一类是，对业务支撑的深度、广度，以及响应的快慢程度。

- 业务的深度和广度，可以从业务对模型的需求总量来侧面衡量；
- 响应的快慢，可以从需求发起到需求接受，再到需求完成的各阶段时间来衡量。

另一类，需要从数据模型本身出发，考量模型的稳定性、生产查询效率、数据质量，以及可扩展能力。

- 数据效率（生产和查询），包含数据最晚（平均）就绪时间、数据最大（平均）执行时长，以及最大（平均）多维查询反馈时间；
- 数据质量，包含每月平均数据问题产生数，细分可以有数据缺失、数据合理性问题、数据一致性问题等；
- 数据占用资源，例如整体数据占用存储资源多少、每天占用计算资源多少。

总结

流量罗盘在美团点评酒店旅游各业务线已经得到了全面的应用，并收获了大量好评和建议。本文从底层数仓总纲和产品层面对流量罗盘进行分析，目前流量罗盘已经在线运营了 2 个季度，后续将会持续优化和迭代。

展望

由于魔方的查询引擎、统一建模工具和起源已经相对成熟，后面产品的查询引擎方面会接入魔方（关于“魔方”的介绍可以参考之前的博客文章《大圣魔方——美团点评酒旅 BI 报表工具平台开发实践》）的查询引擎（筋斗云），配置模块会接入魔方的统一建模工具，将起源应用于统一指标口径，关于筋斗云、魔方

建模工具、起源以及改版后的流量罗盘产品敬请期待！改版后流量罗盘产品架构如图 10 所示：

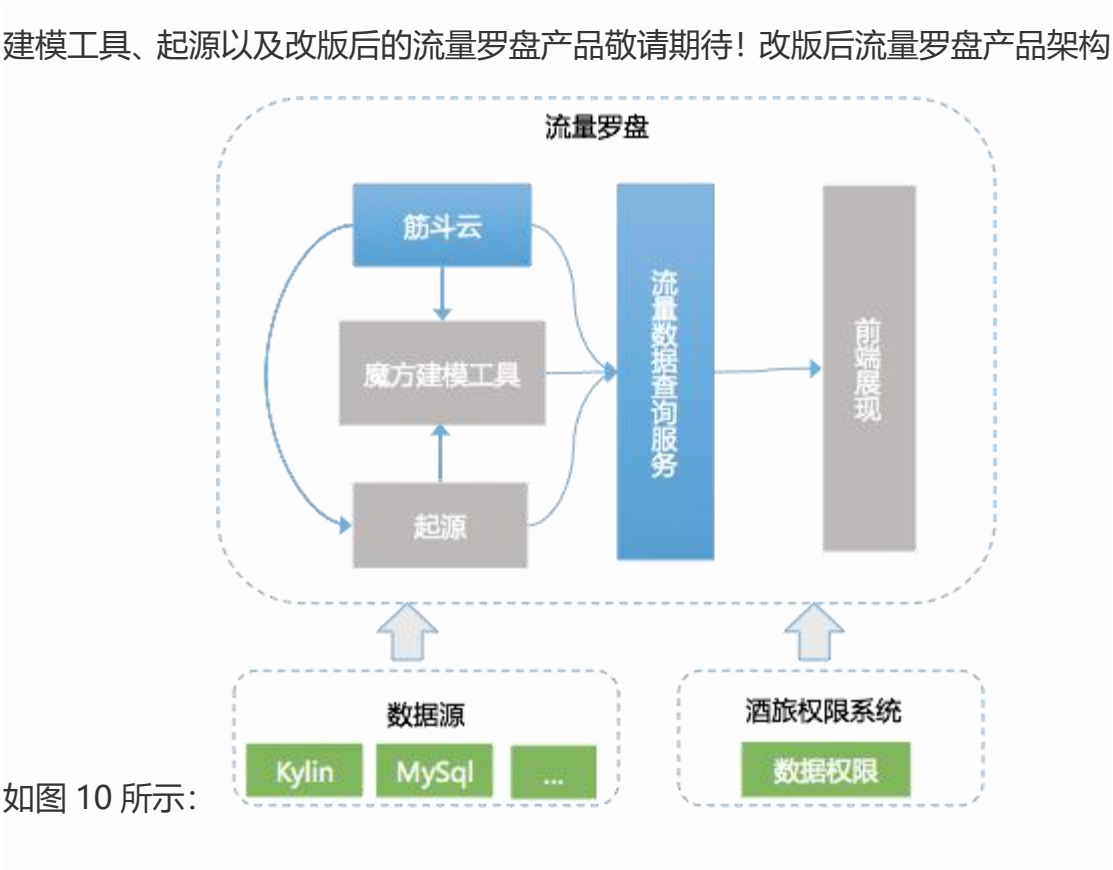


图 10 产品架构图

数据治理

十、美团配送数据治理实践

大数据时代的到来，让越来越多的企业看到了数据资产的价值。将数据视为企业的重要资产，已经成为业界的一种共识，企业也在快速探索应用场景和商业模式，并开始建设技术平台。

但这里要特别强调一下，如果在大数据“拼图”中遗忘了数据治理，可能再多的技术投入也是一种徒劳。因为没有数据治理这一环节，其带来后果往往是：随处可见的数据不统一，难以提升的数据质量，难以完成的模型梳理，难以保障的数据安全等等，源源不断的基础性数据问题会进一步产生，进而导致数据建设难以真正发挥其商业价值。

因此，消除数据的不一致性，建立规范的数据标准，提高数据治理能力，实现数据安全共享，并能够将数据作为企业的宝贵资产应用于业务、管理、战略决策中，发挥数据资产价值变得尤为迫切和重要，数据治理呼之欲出。本文将介绍美团配送技术团队在数据治理方面的一些探索和实践，希望能够对大家有所启发和帮助。

1. 如何理解数据治理

数据治理，从严格的定义来讲是对组织的大数据管理并利用其进行评估、指导和监督的体系框架。企业通过制定战略方针、建立组织架构、明确职责分工等，实现数据的风险可控、安全合规、绩效提升和价值创造，并提供创新的大数据服务。从个人实践的层面来讲，数据治理是对存量数据治理和增量数据管控的一个过程，

对存量数据实现由乱到治、建章立制，对增量数据实现严格把控、行不逾矩的约束。

2. 要达成的目标

数据治理本身并不是目的，它只是实现组织战略目标的一个手段而已。从组织职能和体量大小方面来看，不同类型组织的数据治理目标大不相同，而基于目前美团配送数据团队所处的组织职能和发展阶段来说，我们希望通过数据治理解决数据生产、管理和使用过程中遇到的问题，完善已有的生产管理流程规范，保障数据安全和数据一致性，从而促进数据在组织内无障碍地进行共享。

3. 何时进行数据治理

找准数据治理的切入点，是关乎数据治理成败的关键。很多同学会问，如果将数仓建设分为数仓雏形阶段、数仓迭代阶段和能力沉淀阶段，数据治理应该在哪个阶段切入为宜呢？其实，我们不该把数据治理看作是一个阶段性的项目，它应该是一个贯彻数据建设各阶段的长期工程，只是在不同阶段根据业务特点和技术特点其覆盖的范围和关注的目标有所不同而已。

在数仓雏形阶段，也就是美团配送业务刚成立时，在该阶段中业务有两个特点：第一，重规模、快扩张；第二，业务变化快，数据需求多。为了快速响应业务的需求，并能够保障数据交付结果的准确性，我们主要进行技术规范和指标口径的治理，在规范治理方面，通过制定一系列研发规范来保障研发质量，并在实际建模过程中不断迭代和完善我们的研发质量。在指标治理方面，我们对存量指标口径进行梳理，从而确保指标口径对外输出一致。

在数仓迭代阶段，我们希望通过架构治理改变前期开发的“烟囱式”模型，消除冗余，提升数据一致性。并且随着数仓中管理的数据越多，数据安全和成本问题也变得越发重要。所以在该阶段，我们在产研层面逐步开展架构治理、资源治理和安全治理。在架构治理方面，我们明确了数仓中各层和各主题的职责和边界，构建一致的基础数据核心模型，并制定一系列的指标定义规范来确保指标的清晰定义，并基于业务迭代来不断完善和迭代相应的模型和规范。在资源治理方面，我们通过对不同层级的数据采用不同生命周期管理策略，确保用最少的存储成本来满足最大的业务需求。在安全治理方面，我们通过制定一系列的数据安全规范来确保数据的使用安全。

在能力沉淀阶段，我们基于前两个阶段所做的业务和技术沉淀，将前期一系列规范形成标准，从业务到产研，自上而下地推动数据治理，并通过建立相应的组织、流程和制度来保障标准在该阶段的全面落地实施，并通过建设数据治理平台来辅助更高质量地执行标准。

4.如何开展数据治理

从大的阶段来看，数据治理主要分为存量数据“由乱到治”的阶段，以及增量数据严格按照规章制度实施确保“行不逾矩”的运营阶段。在“由乱到治”的过程中，我们需要沉淀出规章制度、标准规范，以及辅以规章制度标准规范实施的工具和组织。在增量数据的运营阶段，我们主要靠对应的组织确保规章制度的落实，通过审计定期考察实施效果，并在长期的运营中不断完善规章制度。在实现存量数据“由乱到治”的阶段，我们主要采取了“两步走”策略，具体执行策略如下所示。

4.1 定标准，提质量

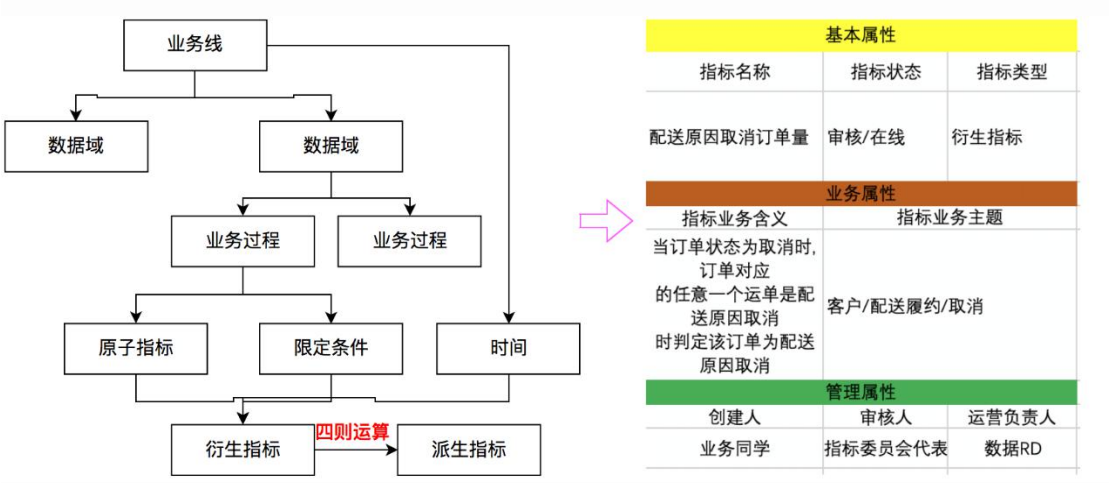
第一步，主要围绕着业务标准、技术标准、数据安全标准和资源管理标准进行展开。通过业务标准，指导一线团队完成指标的规范定义，最终达成业务对指标认知一致性这一目标；然后通过技术标准来指导研发同学规范建模，从技术层面解决模型扩展性差、冗余多等问题并保障数据一致性；通过安全标准来指导我们加强数据的安全管控，确保数据拿不走、走不脱，针对敏感数据，用户看不懂；通过资源管理标准的制定，帮助我们在事前做好资源预算，在事中做好资源管理，在事后做好账单管理。

4.1.1 业务标准

业务标准主要是指标的管理和运营标准，我们主要解决三个问题：指标由谁来定义，指标该如何定义，指标该如何运营。基于这三个问题，我们同时提出了三条原则：

- 业务团队负责指标的定义。
- 产研商分负责给出指标定义标准和辅助工具，辅助业务团队完成指标的规范定义，达成指标认知一致性这一目标。
- 最后由指标管理委员会负责指标的管理与运营，保障指标从创建、审核、上线以及到最后消亡的整个生命周期的运营。

为统一指标的定义，我们将指标分为原子指标、衍生指标和派生指标，原子指标通过限定条件和时间的限定生成衍生指标。衍生指标间的“四则混合运算”构成了派生指标。我们不但制定了指标的标准定义，还对其做了准确的资产归属，一个指标出自一个具体的业务过程，一个业务过程归属于不同的数据域，多个数据域构成了美团配送业务线下的分析场景，如下图所示：



指标定义标准

4.1.2 技术标准

这里所说的技术标准，主要是针对数据 RD 提出的建模标准和数据生产规范，通过建模标准来明确数仓分层架构，并清晰定义每一层的边界与职责，采用维度建模的设计理念。我们的整个仓库架构分为四层：操作层、基础事实层、中间层和应用层，并在每一层同步制定对应的建模规范，如下图所示：

数仓架构以及建模标准

除了建模标准外，我们还制定了涵盖从生产到运维环节的生产规范以保障模型的质量，主要包括上线前的模型评审、生产过程中的完成元数据配置、DQC、SLA

和生命周期设置以及上线后的日常运维机制等等。尤其针对元数据管理和生命周期管理，我们分别制定了仓库每一层元数据维护规范和生命周期管理规范，其中元数据管理规范，是依据数仓各层级中各种类型表的建模标准来制定，需要做到规范命名，明确数据归属，并打通业务元数据和技术元数据之间的关系。而生命周期管理规范，是依据配送业务特点和数仓各层级现状来制定的，如下表所示：

数据仓库表分类	主题/实体	业务过程	主键	维度	维度属性	指标	是否核心模型
分析宽表	需要(跨主题)	X	需要	需要 指定对应的维度表	需要 有表的对应Code表，没有表的枚举值及说明，采用键值对方式	需要	是
汇总聚合表	需要(单一主题)	X		需要 指定对应的维度表		需要	
事实表(主题核心表)	需要(单一主题)	需要	需要	需要 指定对应的维度表	需要 有表的对应Code表，没有表的枚举值及说明，采用键值对方式	不需要	是
事实表(非核心表)	需要(单一主题)	需要	需要	需要 指定对应的维度表	需要 有表的对应Code表，没有表的枚举值及说明，采用键值对方式	不需要	否
维度表			需要	需要 指定对应的维度表	需要 有表的对应Code表，没有表的枚举值及说明，采用键值对方式	X	是
快照表	需要(单一主题)	需要	需要	X	X	X	
三方解耦表	需要(有可能跨主题)			需要 指定对应的维度表	需要	需要	

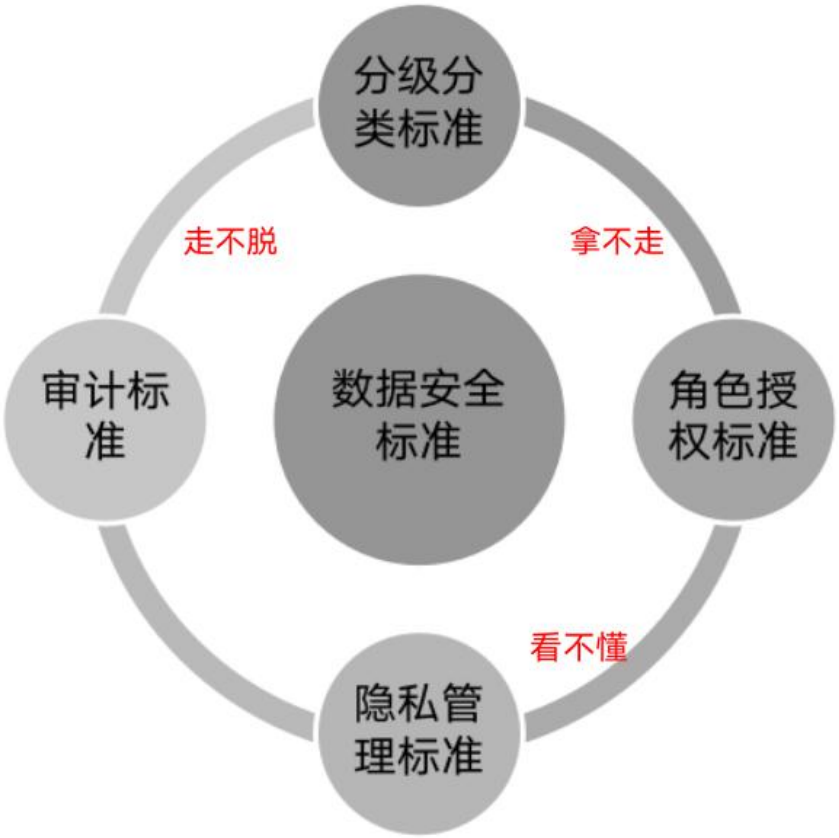
仓库各层元数据管理标准

仓库层级	数据类别	生命周期管理策略	存储方式
A	原始业务数据	永久保留策略	HDFS(ORC格式存储)
	原始LOG数据	周期性删除策略（保留1.5年）	RAID存储
B3/B1	基础明细数据/单维宽表模型	永久保留策略	HDFS(ORC格式存储)
B2/C	汇总表数据/应用表数据	周期性删除策略（2年）	HDFS(ORC格式存储)

仓库各层生命周期管理策略

4.1.3 安全标准

围绕数据安全标准，首先要有数据的分级、分类标准，确保数据在上线前有着准确的密级。第二，针对数据使用方，要有明确的角色授权标准，通过分级分类和角色授权，来保障重要数据拿不走。第三，针对敏感数据，要有隐私管理标准，保障敏感数据的安全存储，即使未授权用户绕过权限管理拿到敏感数据，也要确保其看不懂。第四，通过制定审计标准，为后续的审计提供审计依据，确保数据走不脱。



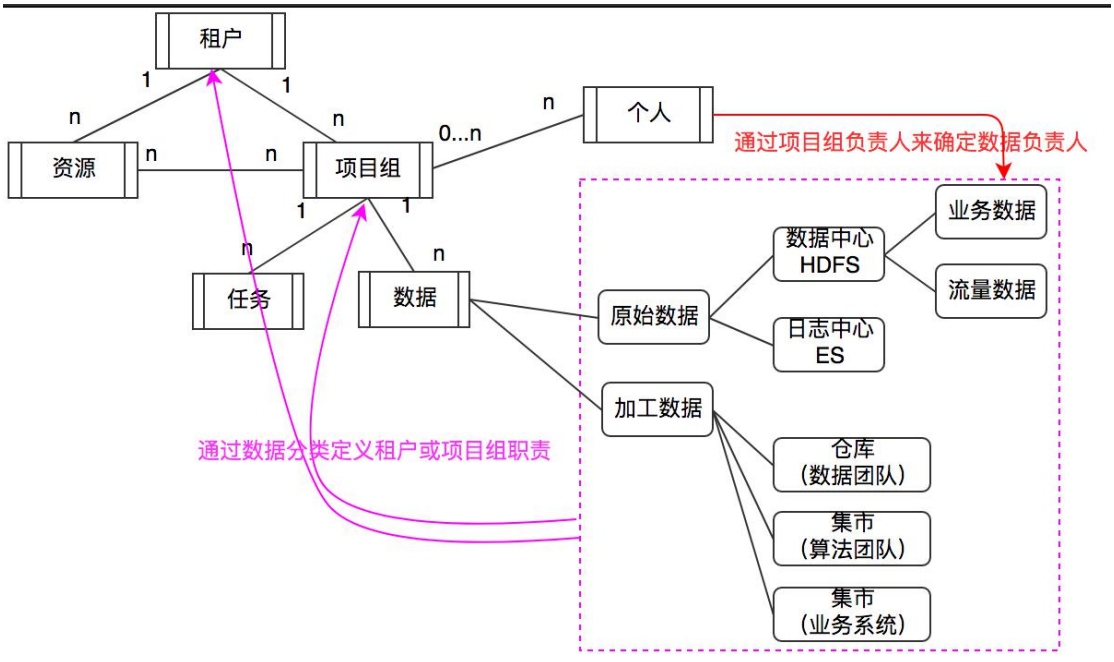
安全标准建设

4.1.4 资源管理标准

在资源管理方面，配送技术工程部已经对资源管理涉及的内容进行了合理抽象和准确定义，抽象出租户、资源和项目组等概念。不管是后续的资源预算还是资源

管理，我们都需要基于租户和项目组来进行运营，因此，对于业务团队而言，我们只需要将租户和项目组特定职能划分清楚，然后根据不同的职能归属我们的资产，并分配生产该资产所需要的资源。为了方便后续的运营，我们对每个租户和项目组分配确定了责任人，由责任人对运营结果负责。

对业务部门来说，资源管理的关键是对数据资产做清晰的分类，基于数据的分类划分不同的租户和项目组，将数据和租户、项目组实现一一映射。由于租户和项目组都有特定的责任人对其负责，因此，我们通过这种映射关系，不仅实现了资产的隔离，还实现了资产确权（项目组负责人同时对资产负责和运营）。我们整体将数据分为两大类，一是原始数据，包括流到数据中心的数据和日志中心的数据，针对流入数据中心的数据，根据其产生的方式不同，又进一步分为业务数据和流量数据。二是加工数据，对应着数据团队的仓库建设和其他团队的集市建设。基于上述的描述，针对资源管理，我们做了如下划分和确权：



资源划分与管理

4.2 重实施，保落实

第二步，落实第一步的标准，完成数据治理第一阶段的目标，实现存量数据“由乱到治”，并完成相应组织和工具的建设，为实现第二阶段“行不逾矩”这一目标提供工具和组织能力。在此过程中，主要分成三个方面的治理工作：第一，架构模型“由乱到治”的治理，消除模型冗余、跨层引用和链路过长等问题，在架构上保证模型的稳定性和数据一致性；第二，元数据“由乱到治”的治理，实现指标的标准定义、技术元数据的完整采集并建立指标与表、字段的映射关系，彻底解决指标认知一致性，以及用户在使用数据过程中的“找数难”等问题；第三，围绕着隐私安全和共享安全加强数据的安全管控来实现数据走不脱、拿不走，以及隐私数据看不懂这一目标。

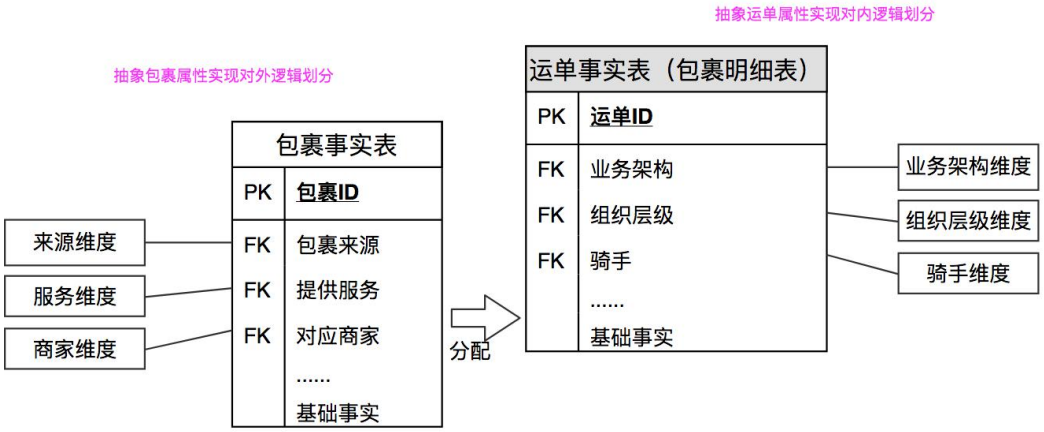
4.2.1 架构治理

总结起来，架构方面的治理主要是解决两个问题：第一，模型的灵活性，避免需求变更和业务迭代对核心模型带来的冲击，让 RD 深陷无休止的需求迭代中；第二，数据一致性，消除因模型冗余、跨层引用等问题带来的数据一致性问题。

模型灵活性

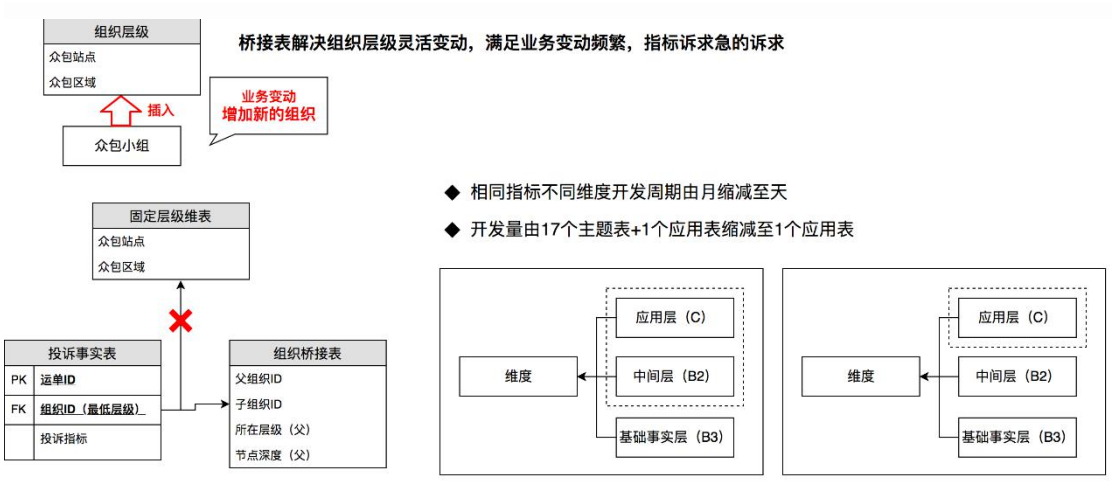
配送解决的是效率、成本和体验三者之间的平衡问题，即在满足一定用户体验的条件下，如何提升骑手配送效率，服务更多的商家，以及如何管控骑手，降低配送成本。抽象到数据层面，基本上反映为上游包裹来源的变化、配送对外提供服务的变化以及对内业务管控的变化。为屏蔽业务迭代给核心模型带来的冲击，我们通过对外封装包裹属性和对内封装运单属性，抽象出包裹来源、提供服务、业

务架构等一致性维度，任何业务迭代在数据层面只涉及维度的调整，大大降低了对核心模型冲击和“烟囱式”数据建设问题（新来一个业务，就拉起一个分支进行建设）。



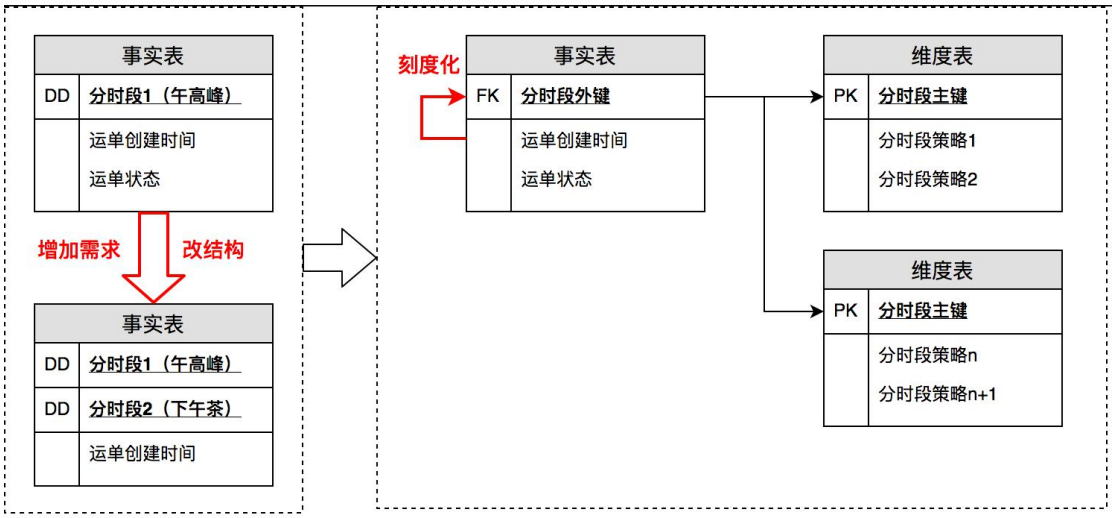
包裹事实分配到运单明细构造单一运单模型

配送指标体系建设的一个重点就是要输出各组织层级的规模、体验和效率指标，实现对运力的有效管控，运力所属组织的层级关系会随业务的迭代而不断变化。为了适应这种变化，避免仅仅因增加维度带来中间层数据的重复建设，我们将组织层级维表由固定层级建模方式调整为桥接表的方式来自适配组织层级变化，从而实现了中间层模型可以自动适配组织层级的变化，能自动产生新维度的指标。如下图所示：



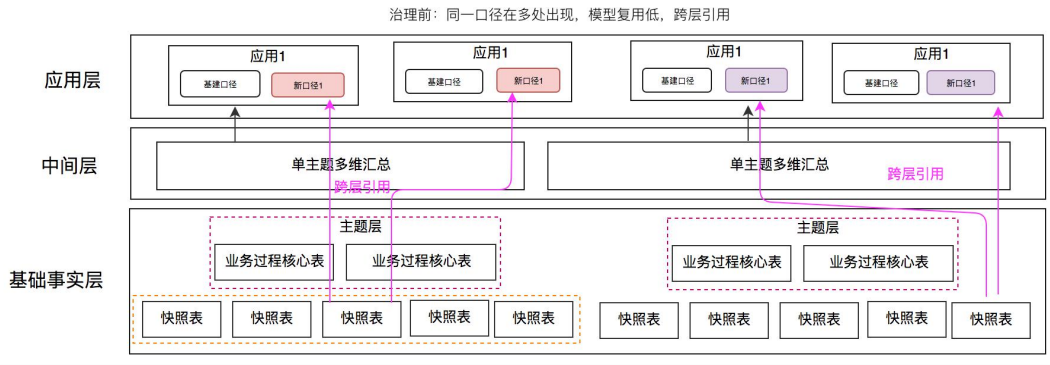
桥接表自适配组织层级灵活变动

在精细化分析的场景下，业务会有分时段、分距离段以及分价格段的数据分析诉求。我们以分时段为例，有晚高峰、午高峰、下午茶等不同的分时段，不同的业务方对同一个时段的定义口径不同，即不同的业务方会有不同的分时段策略。为解决该场景下的分析诉求，我们在事实表中消除退化维度，将原来封装到事实表的时段逻辑迁移到维度表中，并将事实表中的时间进行按特定的间隔进行刻度化作为维表中的主键，将该主键作为事实表的外键。这样，针对业务不同的时间策略需要，我们就可以在维表中进行配置，避免了重复调整事实表和反复刷数的问题。即通过将时间、价格、距离事实刻度化，实现灵活维度分析。如下图所示：



数据一致性

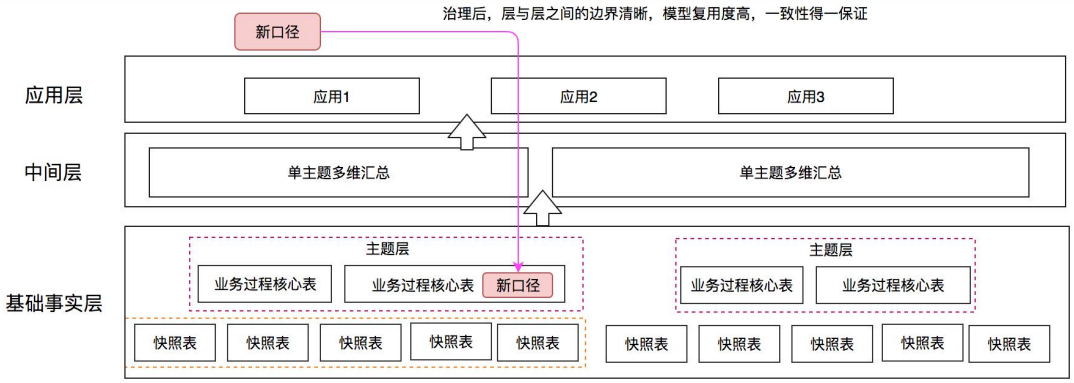
数据一致性得不到保障的一个根本原因，是在建模的过程中没有实现业务口径标签化，并将业务口径下沉到主题层。很多同学在基于需求进行开发时，为实现方便，将新指标口径通过“Case When”的方式在应用层和中间层进行封装开发，主题层建设不能随着业务的迭代不断完善，RD 在开发过程中会直接引用仓库的快照表在中间层或应用层完成需求开发。久而久之，就会造成数据复用性低下，相同指标的口径封装在不同的应用表来满足不同报表的需求，但随着应用的增多，很难保障相同指标在不用应用表封装逻辑的一致性，数据一致性难以得到保障，同时这种方式还带来两个严重后果：第一，跨层引用增多，数据复用性低下，造成计算和存储成本的浪费；第二，一旦指标口径发生变化，将是一个“灾难”，不仅影响评估是一个问题，而且涉及该指标的应用层逻辑调整对 RD 来说也是一个巨大的挑战。



治理前模型架构

因此，我们在“由乱到治”的治理过程中，以衍生事实的方式实现业务口径标签化，将业务逻辑下沉到主题层，消除跨层引用和模型冗余等问题，从技术层面保

障数据一致性是该阶段架构治理的重点。我们在业务上，已经划分了严格的数据域和业务过程，在主题建设层面，将业务划分的数据域作为我们的主题，并基于业务过程进行维度建模，将属于该业务过程的指标口径封装在对应业务过程下的衍生事实中。

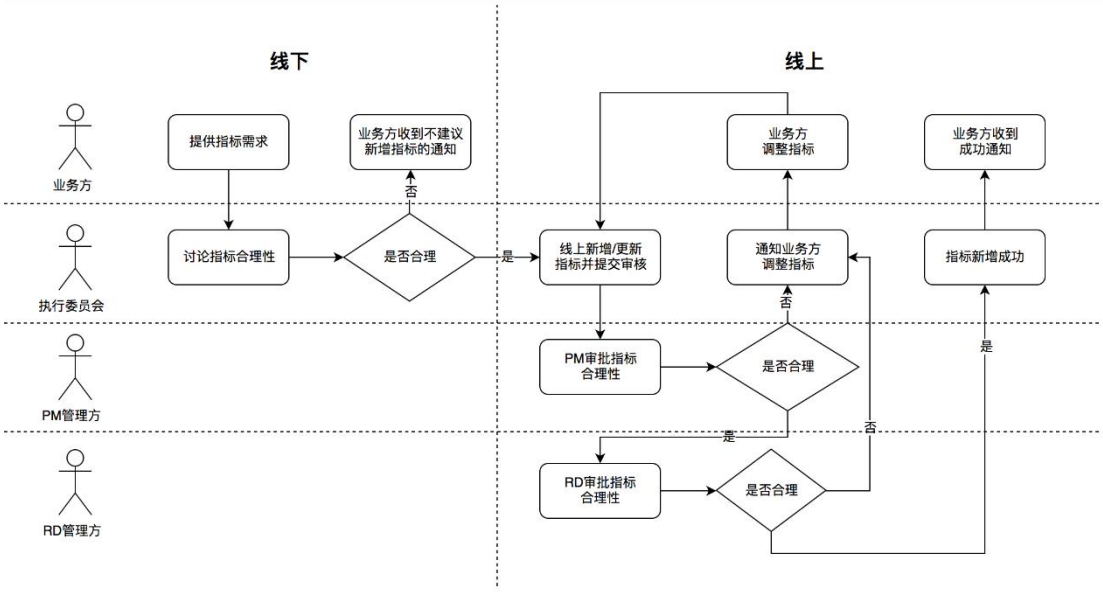


治理后模型架构

4.2.2 元数据治理

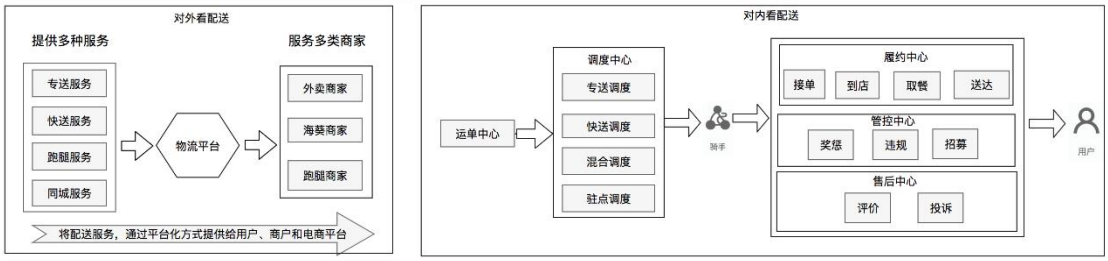
元数据治理主要解决三个问题：首先，通过建立相应的组织、流程和工具，推动业务标准的落地实施，实现指标的规范定义，消除指标认知的歧义；其次，基于业务现状和未来的演进方式，对业务模型进行抽象，制定清晰的主题、业务过程和分析方向，构建完备的技术元数据，对物理模型进行准确完善的描述，并打通技术元数据与业务元数据的关系，对物理模型进行完备的刻画；第三，通过元数据建设，为使用数据提效，解决“找数、理解数、评估”难题以及“取数、数据可视化”等难题。

首先，为保障业务标准的顺利实施，实现业务对指标认知一致性这一目标。我们协同产研、商分、业务部门推动成立了度量衡委员会，并建立起指标运营机制，通过组织保障来实现指标运营按照规范的标准和流程实施。如下图所示：



指标注册流程

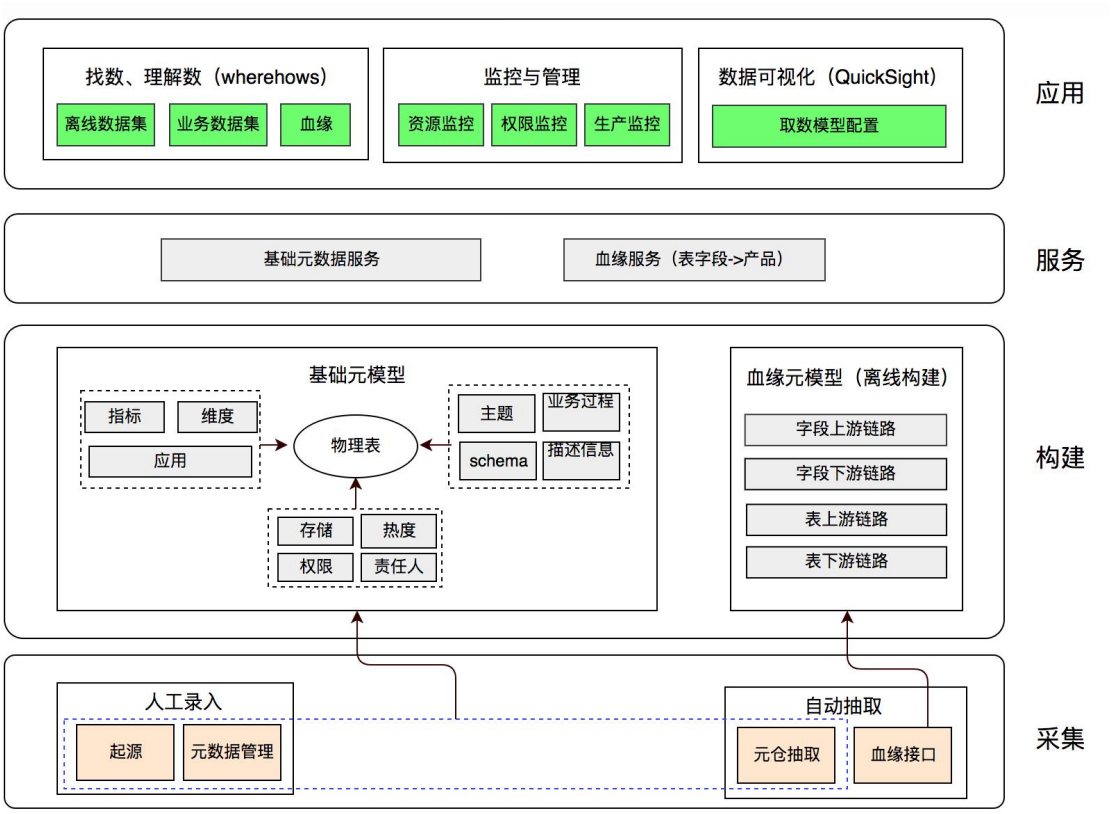
其次，基于配送业务的现状和未来演进方式，我们进行了高度的业务抽象，完成了主题、业务过程和分析方向等元数据内容的建设。配送即物流，通过线上系统和线下运营，我们将用户的配送需求和美团的运力进行有效的资源配置，实现高服务体验、低成本的配送服务。对外，我们将配送服务通过平台化的方式，提供给用户、商户和电商平台，以满足不同用户在不同业务场景下的配送需求。 对内，我们通过不同的调度模式将运单池中的运单调度给合适的骑手来完成履约，平衡规模、成本和体验之间的关系。如下图所示：



配送业务模式抽象

基于以上的业务模式，我们划分了运单主题（对履约数据域下的数据进行构建，支撑规模和体验的数据分析需求）、调度主题（调度数据域下产生的数据，用于支撑调度策略的分析）、结算、评价、投诉、取消主题（用于支撑体验、成本数据分析需求）和管控主题（用于支撑运力奖惩、违规和招募分析需求）等各种主题，并在每个主题下划分对应的业务过程，在应用层制定分析方向的分析标签，通过对元数据内容的建设完成对业务的抽象，为物理模型的刻画准备了基础数据。

第三，元数据服务建设，我们打通了元数据从采集到构建再到应用的整条链路，为使用数据提效，解决“找数、理解数、评估”难题以及“取数、数据可视化”难题。在整个建设过程中，我们围绕着元数据采集、元模型构建、元数据服务以及最后的产品应用进行展开，整体架构如下图所示：



元数据建设架构图

元数据采集

元数据采集分为人工录入和自动抽取，通过人工录入的方式实现物理表的准确归属（包括该表属于仓库哪一层、对应的主题、业务过程、星型模型关系等）以及指标的采集，从而完成技术元数据和业务元数据的采集，通过自动抽取的方式完成生产元数据的采集和使用元数据的采集，主要包括：物理模型的依赖关系、存储占用、热度、等信息。

元模型构建

分为以物理表为核心的基础元模型构建，以及以血缘为中心的血缘元模型。基础元模型构建以物理表为中心，打通其与技术元数据（主题、业务过程、Schema）

的关系，实现了物理表的清晰归属，打通其与生产元数据的关系，为其加上了物理表查询热度、资源消耗、查询密级等生产使用信息，打通其与指标、维度和应用的对应关系，为上层的取数应用建立了完备的元数据。血缘元模型以血缘为中心，不仅构建了从上游业务表到仓库离线表的物理血缘，而且打通了仓库离线表到下游对应报表的血缘，为后续的影响评估构建了完备的元数据基础。

元数据服务

统一元数据服务（OneService），主要提供两类元数据服务，提供查询表、指标、维度基本信息的基础元数据服务以及查询表级血缘、字段级血缘的血缘服务。

元数据应用

主要孵化出了三个产品，以“找数、理解数、影响评估”为应用场景的数据地图（Wherehows），以“取数、数据可视化”为应用场景的数据可视化（QuickSight），以及以管理审计为目的的管理审计报表。

4.2.3 安全治理

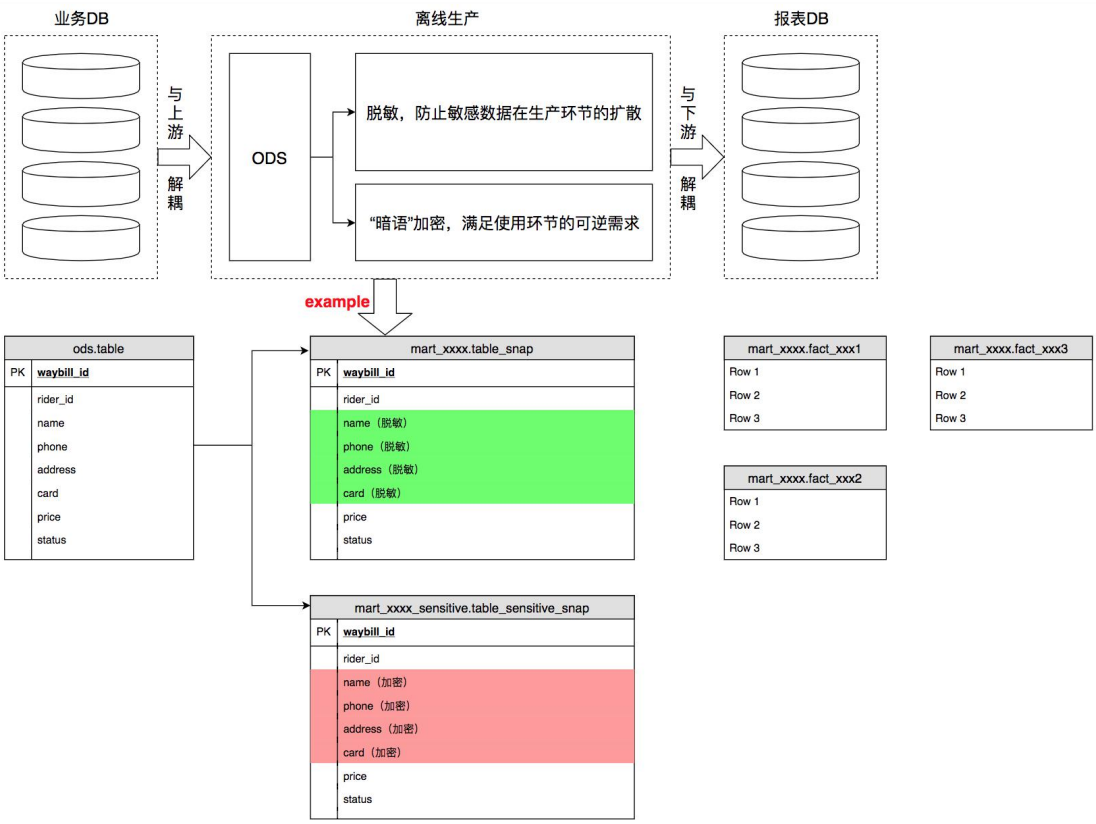
安全治理主要加强了敏感数据的安全治理和数据共享环节的安全治理。通过对隐私数据的安全治理，不仅要保证其在存储环节的不可见性，而且还要保证在其使用环节对用户进行双重鉴权，字段的密级鉴权和解密的密钥鉴权；通过对数据共享环节的安全治理，我们在数据分级分类的基础上，使数据的权限控制从表级权限控制扩展到行级权限控制。

敏感数据安全治理

敏感数据的安全治理，主要是解决敏感数据的存储安全和使用安全。离线场景下，敏感数据存储安全要解决两大挑战：

- 确保仓库侧处理方案既要屏蔽上游业务系统变动带来的影响，又要屏蔽自身策略对下游 BI 系统的影响。
- 要避免敏感数据在整个加工链路中的扩散。

因此，为解决仓库处理方案与上游业务系统和下游 BI 系统的解耦问题，我们在上游敏感数据落到 ODS 环节，确保落到 ODS 层的敏感数据必须是明文，为保障其安全，对 ODS 层的所有数据进行文件加密，但是在使用层面，对下游链路透明保障下游链路的正常生产，并限制 ODS 层数据权限的开放。ODS 层数据只用于安全生产，通过此方案既屏蔽了上游处理方案对仓库的影响，又解决了敏感数据的安全问题。当数据从离开仓库时，在传输环节对敏感数据进行可逆操作，将敏感数据以明文的形式推入 BI 库，实现与下游 BI 系统的解耦。为解决敏感数据在整个生产链路的扩散，我们在快照层对敏感数据进行脱敏处理，从快照层开始消除敏感数据，为保障敏感数据的可逆性，将 ODS 层的敏感数据抽取到安全库中并进行加密存储，实现安全独立管理。具体执行如下图所示：



针对敏感数据的使用安全，我们通过对敏感字段的权限控制和对解密密钥的权限控制，来实现敏感数据使用安全这一目标。针对单独抽取的敏感数据，我们除了针对敏感数据设置其相应的密级确保敏感数据的权限管控外，还基于“暗语”的加密方式为每个项目组分配一个相同的密钥，并且将该密钥存放到与 Hadoop 集群集成的 KMS 进行管理（确保支撑离线计算的高并发），确保解密时实现密钥的权限管控。

共享环节安全治理

针对共享环节的安全治理，我们主要是在数据生产环节完成数据的分级分类和数据确权，在数据的使用环节完成数据的表级权限控制和行级权限控制。确保数据在使用环节规范的审批流转，权限开放以后的安全审计，保证数据走不脱。

首先，我们在生产环节 B3、B2、B1 层数据按照主题或实体 C 层数据按照应用方向进行逻辑划分，并设定资源的密级和权限负责人。特别地为实现 B3 层数据在查询环节可按照业务线进行权限管控这一目标（即行级鉴权），针对 B3 层数据，我们标记该数据需要在查询环节进行行级权限管控，标记使用行级鉴权所需的字段和该字段对应的枚举值。

其次，在使用环节，我们按照资产密级和使用人角色完成数据的审批流转，实现数据的安全共享。

第三，针对 B3 层数据，审计是否设置了行级权限管控。在数据开放时是否存在越权使用的情况，以及针对即将离职员工加强数据的使用审计，保证数据走不脱。

在数据“由乱到治”的治理过程中，我们不仅实现了存量数据的“由乱到治”，并且在此过程中沉淀出了一系列的建模方法论、工具，并建立了相应的安全小组和指标运营组织。同时，我们为后续增量数据治理确保数据建设“行不逾矩”，提供了强有力的组织保障、稳定的辅助工具和严格的执行标准。在数据治理的第二阶段实现增量数据的“行不逾矩”的过程中，我们主要围绕大数据架构审计、大数据安全与隐私管理审计、大数据质量管理审计和大数据生命周期管理审计这四方面的工作展开，保障治理工作的持续进行，不断提高了组织的治理水平。

5. 工具简介

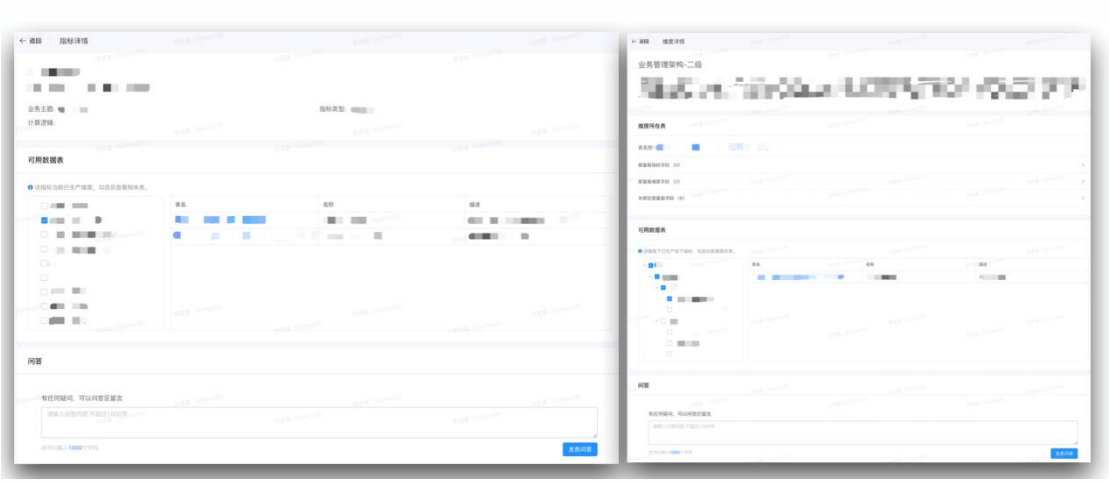
5.1 数据地图（Wherehows）

数据地图作为元数据应用的一个产品，聚焦于数据使用者的“找数”场景，实现检索数据和理解数据的“找数”诉求。我们通过对离线数据集和在线数据集的元数据刻画，满足了用户找数和理解数的诉求，通过血缘图谱，完成物理表到产品的血缘建设，消除用户人肉评估的痛苦。

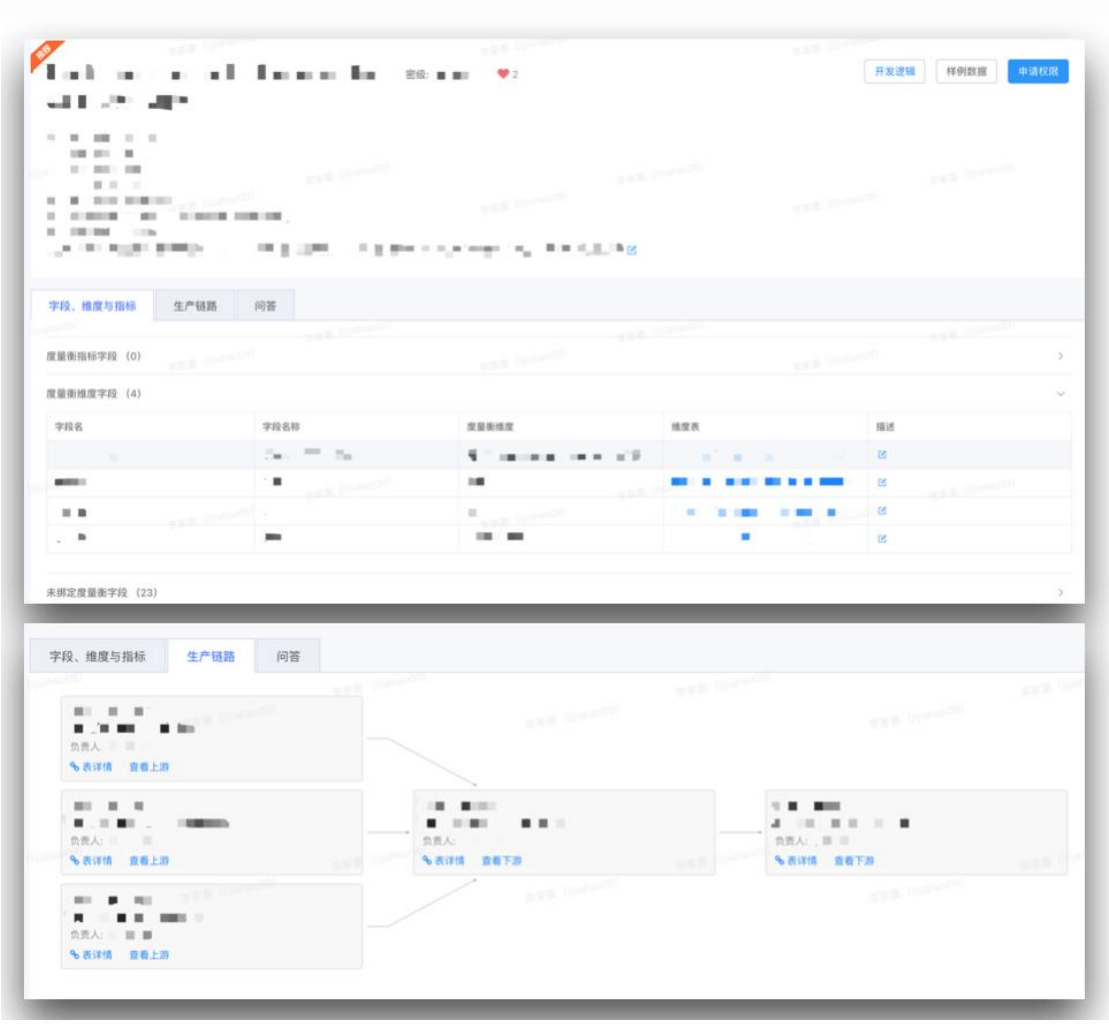
离线数据场景

1.关键字检索和向导查询共同解决了“找数据”的问题：大部分的检索数据场景下，数据使用者都可以通过关键字检索来得到匹配结果。剩下的一小部分场景，例如，对于新人入职后如何了解整个数仓和指标的体系（数仓分几层，每层解决什么问题，都孵化出什么模型；整个指标、维度体系都是怎么分类，有哪些指标和维度），这部分场景可以使用向导查询功能。向导查询相当于分类查询，将表和指标按照业务过程进行分类，用户可以按照分类逐步找到想要的表或指标。

2.我们打通了业务元数据和技术元数据之间的关系，提高了“找数据”的能力：通过“Wherehows”查找到指标后，不仅不可查看指标的业务定义，还能查看指标的技术实现逻辑，指标在哪些维度或维度组合中已经实现，并且能够在哪张表里找到这些维度，或维度组合的指标数据。反之，也可以知道在某个维度下已经实现了哪些指标，对应的指标在哪些表里。这些功能能让用户更加方便地找到想要的数据。



3.我们提供了较为完善的数据信息，帮助用户更好理解数据：对于表的信息，“Wherehows”除了提供表和字段的中英文名称、描述信息等基础信息外，为了帮助用户更好地理解表的建设思路，我们还提供了表的星型模型（可以关联的一致性维度及对应的维度表）、表的血缘关系等信息。



4.我们通过评论问答功能，帮助用户可以快速得到问题反馈：如果用户看了信息后还是感到有问题，“Wherehows”提供评论问答的功能，用户通过这个功能可以进行提问，会有相应的负责人进行回复。对于重复问反复问的问题，用户通过查看其它人的提问和回复就能找到答案。并且负责人还会定期的将问答信息沉淀到对应的元数据里，不断地对元数据进行补充和完善。

业务数据场景

业务数据场景主要想解决的一个问题是，如何知道一个业务表（MySQL 表）有没有同步到数仓。如果没有同步，能够找谁进行同步。因为已经打通“业务表 -> 数仓表 -> 产品”三者之间的血缘关系，我们能够轻松解决业务数据场景的问题。

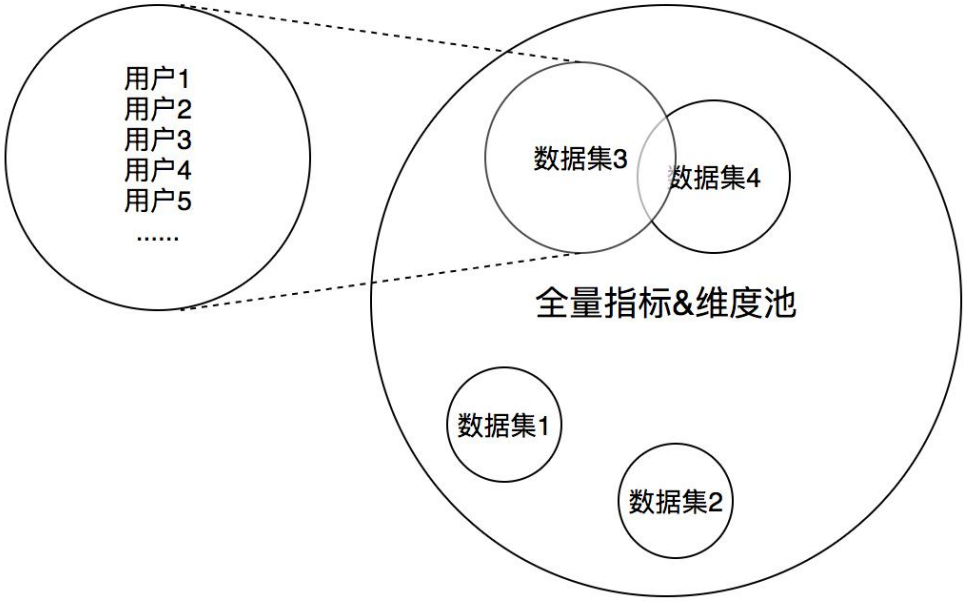
生产评估场景

在日常数据生产工作中，我们经常需要对表进行影响评估、故障排查、链路分析等工作，这些工作如果靠纯人工去做，费时费力。但现在我们已经打通了“业务表/字段 -> 数仓表/字段 -> 产品”三者之间的血缘关系，就能够在 10 分钟内完成评估工作。对于不同的场景，血缘链路提供了两个便捷的功能：过滤和剪枝。例如，某个表逻辑需要修改，需要看影响哪些下游表或产品？应该要通知哪些 RD 和 PM？这种情况下，血缘工具直观地显示影响了哪些负责人和产品，以及这个表的下游链路。

有些表的链路很长，整个血缘关系图很大，这样会导致用户定位信息或问题。所以血缘工具提供了剪枝的功能，对于没用的、不想看到的分支可以剪掉，从而让整个链路变得更加直观。

5.2 数据可视化（QuickSight）

聚焦于数据使用者“取数”场景，使用 QuickSight，用户可以不再关心数据的来源，不再担心数据的一致性，不再依赖 RD 的排期开发。通过所选即所得的方式，满足了用户对业务核心指标的二次加工、报表和取数诉求。首先，我们通过指标池、数据集等概念对离线生产的指标进行逻辑隔离，针对不同用户开发不同的数据集以达到权限控制的目的，如下图所示：



用户、指标池与数据集间的关系

其次，我们为用户提供一系列的组件，帮助用户基于为其开放的数据集实现指标的二次加工和数据可视化功能，满足其在不同业务场景下的取数和可视化应用。如下图所示：

指标加工组件

6.总结与展望

经过三个阶段的治理工作，我们在各个方面都取得了较好的效果：

- 在数据标准方面，我们制定了业务标准、技术标准、安全标准、资源管理标准，从而保障了数据生产、管理、使用合规。
- 在数据架构方面，我们通过桥接表、时间刻度化、业务口径下沉等手段提升模型灵活性，并保障数据一致性，消除跨层引用和模型冗余等问题。
- 在数据安全方面，我们加强了对敏感数据和数据共享环节的安全治理，保证数据拿不走、走不脱，隐私数据看不懂。
- 在元数据建设方面，我们打通了从采集到构建再到应用的整条链路，并为数据使用人员提供数据地图、数据可视化等元数据应用产品，帮助他们解决了“找数”、“取数”、“影响评估”等难题。

未来，我们还会继续通过组织、规范、流程等手段持续对数据安全、资源利用、数据质量等各方面进行治理，并在数据易用性上下功夫，持续降低用户的数据使用成本。

- 在数据架构方面，随着数据库技术的飞速进步，现在已经有很多数据库能够支持千万级乃至亿级数据的现算先用，我们也在尝试使用这些数据库帮助提升数据开发效率，改善数仓分层管理和应用支撑效率。
- 在数据产品方面，我们将持续完善数据地图、数据可视化等数据应用产品，帮助用户快速探查、高效分析，真正发挥数据的业务价值。

十一、美团酒旅数据治理实践

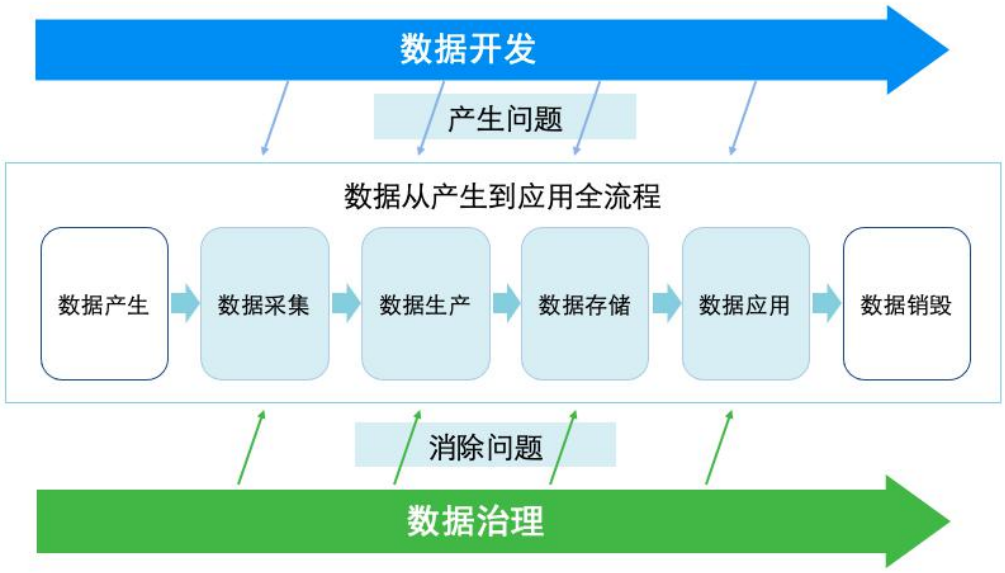
一、背景

1. 为什么要做数据治理

随着移动互联网的兴起，线下商业活动逐渐开始向线上化发展，数据的产生速度有了极大的提升。越来越多的公司开始认识到数据的重要性，并将其打造成为公司的核心资产，从而驱动业务的发展。在数据相关的领域中，“数据治理”这个话题近两年尤为火热，很多公司特别是大型互联网公司都在做一些数据治理的规划和动作。

为什么要做数据治理？因为在数据产生、采集、加工、存储、应用到销毁的全过程中，每个环节都可能会引入各种质量、效率或安全相关的问题。在公司早期的发展阶段，这些数据问题对公司发展的影响并不是很大，公司对问题的容忍度相对也比较高。但是，随着业务的发展，公司在利用数据资产创造价值的同时，对数据质量和稳定性要求也有所提升。此外，当数据积累得越来越多，公司对数据精细化运营程度的要求也随之提高，会逐渐发现有很多问题需要治理。

同时，在数据开发的过程中也会不断引入一些问题，而数据治理就是要不断消除引入的这些问题，保障数据准确、全面和完整，为业务创造价值，同时严格管理数据的权限，避免数据泄露带来的业务风险。因此，数据治理是数字时代很多公司一项非常重要的核心能力。



2. 需要治理哪些问题

数据治理是一项需要长期被关注的复杂工程，这项工程通过建立一个满足企业需求的数据决策体系，在数据资产管理过程中行使权力、管控和决策等活动，并涉及到组织、流程、管理制度和技术体系等多个方面。一般而言，数据治理的治理内容主要包括下面几个部分：

- **质量问题**：这是最重要的问题，很多公司的数据部门启动数据治理的大背景就是数据质量存在问题，比如数仓的及时性、准确性、规范性，以及数据应用指标的逻辑一致性问题等。
- **成本问题**：互联网行业数据膨胀速度非常快，大型互联网公司在大数据基础设施上的成本投入占比非常高，而且随着数据量的增加，成本也将继续攀升。
- **效率问题**：在数据开发和数据管理过程中都会遇到一些影响效率的问题，很多时候是靠“盲目”地堆人力在做。
- **安全问题**：业务部门特别关注用户数据，一旦泄露，对业务的影响非常之大，甚至能左右整个业务的生死。
- **标准问题**：当公司业务部门比较多的时候，各业务部门、开发团队的数据标准不一致，数据打通和整合过程中都会出现很多问题。

数据常见问题

- 数据质量
- 成本控制
- 研发效率
- 数据安全
- 标准规范

3. 美团酒旅数据现状

2014 年，美团酒旅业务成为独立的业务部门，到 2018 年，酒旅平台已经成为国内酒旅业务重要的在线预订平台之一。业务发展速度较快，数据增长速度也很快。在 2017 到 2018 两年里，生产任务数以每年超过一倍的速度在增长，数据量以每年两倍多的速度在增长。如果不做治理的话，根据这种接近指数级的数据增长趋势来预测，未来数据生产任务的复杂性及成本负担都会变得非常之高。在 2019 年初，我们面临着下面五种问题：

- **数据质量问题严重**：一是数据冗余严重，从数据任务增长的速度来看，新上线任务多，下线任务少，对数据表生命周期的控制较少；二是在数据建设过程中，很多应用层数据都属于“烟囱式”建设，很多指标口径没有统一的管理规范，数据一致性无法进行保证，同名不同义、同义不同名的现象频发。
- **数据成本增长过快**：某些业务线大数据存储和计算资源的机器费用占比已经超过了 35%，如果不加以控制，大数据成本费用只会变得越来越高。
- **数据运营效率低下**：数据使用和咨询多，数据开发工程师需要花费大量时间一对一解答业务用户的各种问题。但是这种方式对于用户来说，并没有提升数据的易用性，无法有效地积累和沉淀数据知识，还降低了研发人员的工作效率。
- **数据安全缺乏控制**：各业务线之间可以共用的数据比较多，而且每个业务线没有统一的数据权限管控标准。
- **开发标准规范缺失**：早期为快速响应业务需求，研发人员通常采用“烟囱式”的开发模式，由于缺乏相应的开发规范约束，且数据工程师的工作思路和方式差异性都非常大，导致数据仓库内的重复数据多，规范性较差。当发生数据问题时，问题的排查难度也非常大，且耗时较长。

4. 治理目标

2019 年，美团酒旅数据团队开始主动启动数据治理工作，对数据生命周期全链路进行体系化数据治理，期望保障数据的长期向好，解决数据各个链路的问题，并保持数据体系的长期稳定。具体的目标包含以下几个方面：

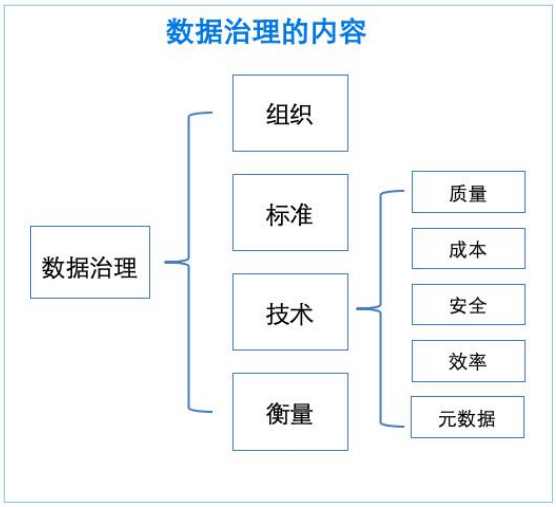
1. 建立数据开发全链路的标准规范，提高数据质量，通过系统化手段管理指标口径，保障数据一致性。
2. 控制大数据成本，避免大数据机器成本膨胀对业务营收带来的影响，合理控制数据的生命周期，避免数据重复建设，减少数据冗余，及时归档和清理冷数据。
3. 管理数据的使用安全，建立完善的数据安全审批流程和使用规范，确保数据被合理地使用，避免因用户数据泄露带来的安全风险和商业损失。
4. 提高数据工程师的开发和运维效率，减少他们数据运营时间的投入，提高数据运营的自动化和系统化程度。

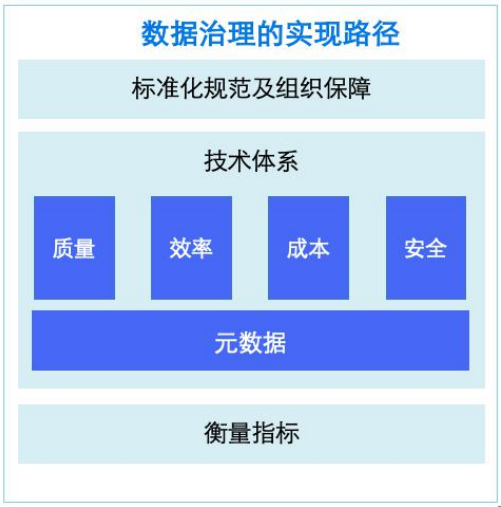
二、数据治理实践

其实早在 2018 年以前，酒旅数据组就做过数据治理，当时只是从数仓建模、指标管理和应用上单点做了优化和流程规范。之后，基于上面提到的五个问题，我们又做了一个体系化的数据治理工作。下面将介绍一下美团酒旅数据团队在数据治理各个方向上的具体实践。

1. 数据治理策略

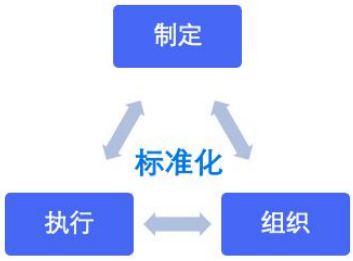
数据治理方案需要覆盖数据生命周期的全链路，我们把数据治理的内容划分为几大部分：组织、标准规范、技术、衡量指标。整体数据治理的实现路径是以标准化的规范和组织保障为前提，通过做技术体系整体保证数据治理策略的实现。同时，搭建数据治理的衡量体系，随时观测和监控数据治理的效果，保障数据治理长期向好的方向发展。





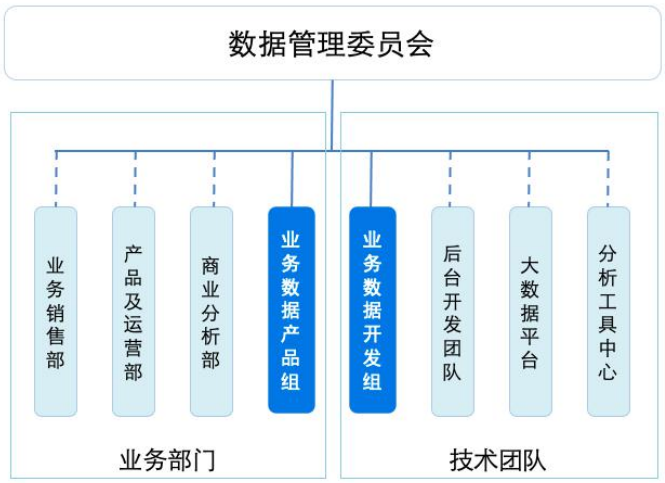
2. 标准化和组织保障

我们制定了一个全链路的数据标准，从数据采集、数仓开发、指标管理到数据生命周期管理，全链路建立标准，在标准化建立过程中联合组建了业务部门的数据管理委员会。



全链路数据标准化建设

- 数据采集
- 数仓开发
- 指标管理
- 数据应用
- 数据生命周期管理



2.1 标准化

数据标准化包括三个方面：一是标准制定；二是标准执行；三是在标准制定和执行过程中的组织保障，比如怎么让标准能在数据技术部门、业务部门和相关商业分析部门达成统一。

从标准制定上，我们制定了一套覆盖数据生产到使用全链路的数据标准方法，从数据采集、数仓开发、指标管理到数据生命周期管理都建立了相应环节的标准化的研发规范，数据从接入到消亡整个生命周期全部实现了标准化。

2.2 组织保障

根据美团数据管理分散的现状，专门建立一个职能全面的治理组织去监督执行数据治理工作的成本有点太高，在推动和执行上，阻力也会比较大。所以，在组织保障上，我们建立了委员会机制，通过联合业务部门和技术部门中与数据最相关的团队成立了数据管理委员会，再通过委员会去推动相关各方去协同数据治理的相关工作。

业务部门的数据接口团队是数据产品组，数据技术体系是由数据开发组负责建设，所以我们以这两个团队作为核心建立了业务数据管理委员会，并由这两个团队负责联合业务部门和技术部门的相关团队，一起完成数据治理各个环节工作和流程的保障。组织中各个团队的职责分工如下：

数据管理委员会：负责数据治理策略、目标、流程和标准的制定，并推动所有相关团队达成认知一致。 **业务数据产品组**：负责数据标准、需求对接流程、指标统一管理、数据安全控制以及业务方各部门的协调推动工作。 **技术数据开发组**：

负责数据仓库、数据产品、数据质量、数据安全和数据工具的技术实现，以及技术团队各个部门的协调推动工作。

3. 技术系统

数据治理涉及的范围非常广，需要协作的团队也很多，除了需要通过组织和流程来保障治理行动正常开展，我们也考虑通过技术系统化和自动化的方式进一步提效，让系统代替人工。下面我们将从数据质量、数据成本、数据安全和运营效率等几个方向，来逐一介绍技术实现方案。

3.1 数据质量

数据质量是影响数据价值最重要的因素，高质量的数据给带来准确的数据分析，错误的数据会把业务引导到错误的方向。数据质量涉及范围较广，在数据链路的每一个环节都有可能出现数据质量问题，酒旅业务现阶段的主要质量问题包括：

- 数仓规范性差，数仓架构无统一的强制规范执行约束，数仓历史冗余数据严重。
- 应用层数据属于“烟囱式”建设，指标在多个任务中生产，无法保证数据的一致性。
- 数据下游应用的数据使用无法把控，数据准确较差，接口稳定性无法得到保障。
- 业务方对多个数据产品的指标逻辑无统一的定义，各个产品中数据不能直接对标。

数据组的治理数据质量方案覆盖了数据生命周期的各个环节，下面将介绍一下整体的技术架构。

- **统一数仓规范建模（One Model）**：通过统一数仓规范建模系统化保障数仓规范执行，做到业务数仓规范标准化，并及时监控和删除重复和过期的数据。
- **统一指标逻辑管理（One Logic）**：通过业务内统一的指标定义和使用，并系统化管理指标逻辑，数据应用层的数据指标逻辑都从指标管理系统中获取，保障所有产品中的指标逻辑一致。
- **统一数据服务（One Service）**：通过建设统一的数据服务接口层，解耦数据逻辑和接口服务，当数据逻辑发生变化后不影响接口数据准确性，同时监控接口的调用，掌握数据的使用情况。
- **统一用户产品入口（One Portal）**：分用户整合数据产品入口，使同一场景下数据逻辑和使用方式相同，用户没有数据不一致的困惑。

3.1.1 统一数仓规范建模（One Model）

在业务发展初期，数据团队集中精力在快速建设数仓来支持业务，数仓建模规范疏于管理。随着业务的发展，数仓中的数据急剧增多，数据产品和下游应用快速增加，数据工程师和数据使用方也变得越来越多，数仓的问题日益突显。业务数据仓库从初期发展到现在主要暴露了 3 方面的问题：

- 数据规范性较差，不同时间的数仓规范不同，数仓规范的执行审核需要较多的人力。
- 数据不一致问题多，同一指标在多个 ETL 中生产，数据更新同步也不及时。
- 历史数据冗余严重，数据存储方式较多，业务方查询不知道该用哪个数据。

数据团队主要通过数仓规范化制定、数仓分层架构和数仓规范化系统来解决上述问题，下面是我们的具体解决方案。

制定标准-数仓规范

做好数仓规范化最基本的前提是要制定一系列标准化的规范，并推动组内同学执行。标准化的适用性、全面性和可执行性直接影响到规范的执行效果。数仓规范主要从 3 个方面制定数据标准化：

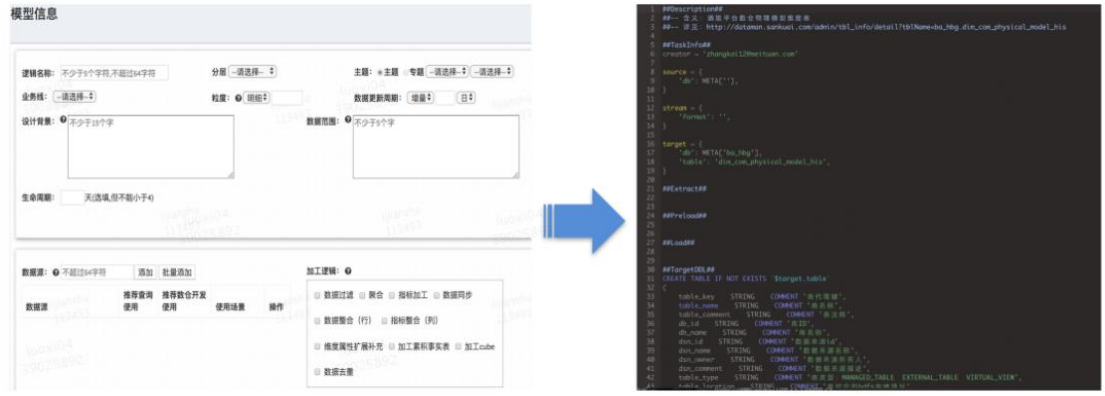
- 数仓建模规范，数仓建设最基础的规范，包括分层、命名、码值、指标定义、分层依赖等维度。
- 主数据管理规范，数仓各个主题的数据只有一份，团队共建复用，不能重复开发。
- 数据使用规范，在查询数据时优先查询主题层，不再提供明细层和 ODS 层的查询访问入口。

工具保障-数仓规范化开发系统-Dataman

在执行数据规范化的过程中，我们发现团队中每个人对规范的理解不一致，很可能造成数据规范不统一，审核人在审核上线任务时需要考虑规范的全部规则，审批需要投入的人力较多。在这样的流程下，数据规范性无法从根源上进行控制，因此需要建设数据规范化的工具，通过系统保障规范的一致性。数据组使用的数据层规范化工具-Dataman，主要包括 3 个功能模块：标准化规范、配置化开发和规则化验证。

- **标准化规范**：制定业务数据仓库的标准规范并配置在系统中，包括架构分层、字段管理、词根管理、公共维度和码值管理等，在 ETL 开发时通过统一的数仓规范开发，通过配置化实现数仓的命名、分层和码值，保障数仓长期的规范性。

- **配置化开发**：系统化保障工程师在开发 ETL 过程中遵守数仓规范，Dataman 可以用配置化的方式生成 XT 任务模板，模板中包含数据模型的基础信息，研发同学只需要在任务模板中开发数据生产逻辑。



- **规则化验证**：跟进数据仓库底层元数据和标准化配置信息，定期扫描数仓的规范性情况，判断出不符合数仓规范的任务和高相似度的数据表。

3.1.2 统一指标逻辑管理（One Logic）

业务使用数据的第一步是搭建业务指标体系，业务的目标和策略的执行情况需要通过指标来分析，指标体系的合理性和指标数据的质量直接影响到业务决策，指标的重要性不言而喻。我们通过系统化地管理数据指标，从根源上解决指标口径一致性问题，主要从以下 3 个方向入手：

- **指标定义规范化**
- **指标管理系统化**
- **数据查询智能化**

指标定义规范化

此处主要从指标的生成和管理上做好规范，确保业务同学和研发人员对指标体系管理的认知一致，确保指标的新建、更改和使用都按照规范执行。我们通过下面 2 个方向来实现指标定义的规范统一。

- **业务指标体系的规范化**：我们在业务线内统一了指标体系规范，指标分为原子指标、计算指标和复合指标，通过使用这 3 类指标支持业务的数据分析需求，业务未来新增指标也要按照这个标准分类。
- **指标的管理规范化**：我们与商业分析团队一起梳理业务指标逻辑标准和录入流程，通过制定指标的新增和变更规范 SOP，解决由指标管理流程引起的质量问题，使得指标定义、系统录入、指标认证和使用各个环节都有严格的流程管控，经由业务侧数据产品经理、业务侧数据治数据管理员和数据工程师共同审批，确保标准规范的落地执行。

指标管理系统化

物理数据表管理：数据表管理的信息主要包括表的基础元数据信息、表类型（维表或事实表）、表的推荐度、描述信息和样例数据等。数据表管理主要是面向数据开发同学，通过维护数据表信息，为数据模型和指标管理提供数据基础支持。

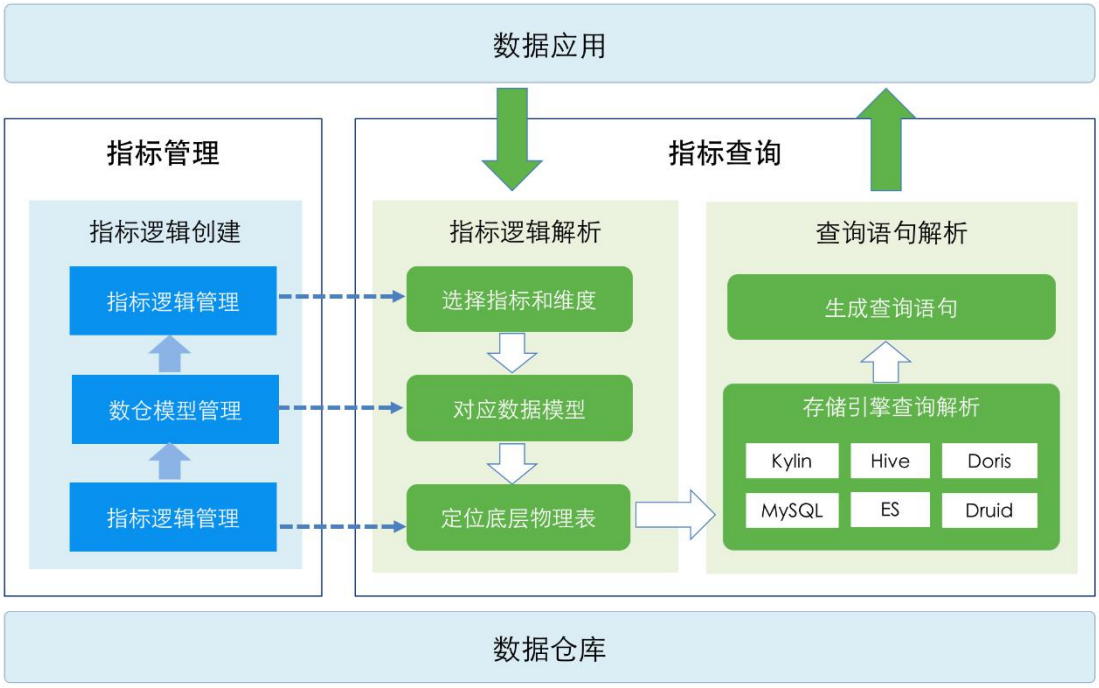
数据模型管理：是对物理数据表的模型构建，通过一个物理模型可以查询到指标和相关的维度数据。数据模型可以是星型模型或宽表，星型模型中维护多个数据表的关联方式、关联字段、维度表包含字段和模型的 ER 图等信息。

指标管理：主要包括 2 部分的内容，指标的业务信息和技术信息。

- **业务信息**：为了保障业务的指标信息准确且统一，指标的业务信息需要数据产品经理与商业分析团队讨论确定后录入，录入后需要指标所属数据主题的负责人审批后才能上线。
- **技术信息**：技术信息主要包括指标对应的物理模型以及指标的计算逻辑，技术信息的填写需要数据工程师配置。技术信息配置后会在系统里生成技术元数据，指标管理系统通过技术元数据生成数据查询语句，提供给下游应用。

指标查询智能化

在指标管理系统中创建指标时，我们系统化管理了指标与数仓物理模型的关联关系和取数逻辑，通过数据物理模型获得指标对应的字段和可以关联的维度，以此把指标解析为数据查询 SQL 语句，通过数据查询引擎执行生产的 SQL，智能化获得指标数据。



在查询解析过程中，经常出现指标绑定了多个底层数据表的情况，此时需要我们手动的选一个物理模型作为指标生产的底层数据。但问题是，如果一个指标对应的模型太多，每次解析都需要手动指定，研发人员不确定选择哪个模型的性能最好。另外，随着物理模型的增多，大量旧的指标配置的关联模型不是最优解，就需要手动优化更改。为了解决这个问题，指标管理系统增加了智能解析模块，在选择智能模式查询时，系统会根据指标管理模型的数据量、存储性能和查询次数等信息自动选取最优的物理模型。

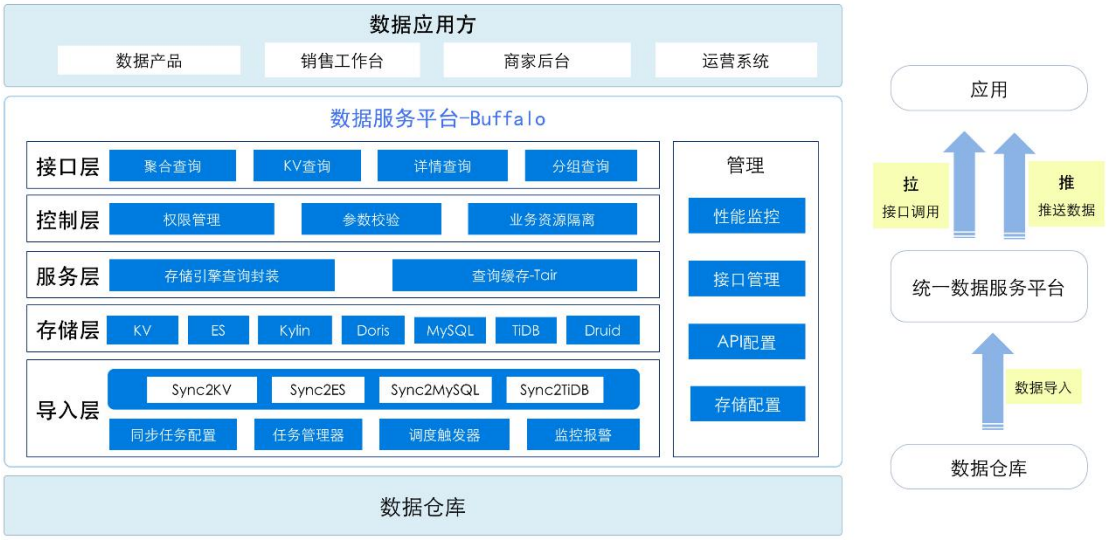
3.1.3 统一数据服务（One Service）

数据仓库对外提供数据的需求越来越多，除了管理层、分析师和产品运营同学使用数据产品和报表外，数据还需要提供到各个业务系统中使用。常用的提供数据的方式主要包括同步数据表、提供 SQL 和为下游服务开发定制化 API 接口等方式，但存在以下几个方面的问题：

- 数据一致性无法保障，当数据指标逻辑更改时，业务系统不能及时调整，导致不同业务系统的数据不一致。
- 数据同步到业务系统后，我们就无法管控数据的使用方式，也不能监控到数据是否被其他下游使用的情况。
- 数据开发效率比较低，数据服务稳定性比较差，数据工程师开发一个定制化 API 接口需要几天时间，各个接口服务单独维护，服务稳定性也比较差。

从 2018 年开始，数据 BP 中心与分析系统中心合作建设了统一数据 API 服务平台（Buffalo），通过开发可配置的数据接口服务平台实现数据对外的灵活提供，并实现对数据服务的下游使用及性能的可监控。统一的数据服务平台解决了几个比较关键的问题：

- **数据逻辑统一收口**：数据服务接口和数据逻辑解耦，当数仓更改和数据指标逻辑变更后下游无感知。
- **数据服务的更好管控**：研发同学能够了解到数据被哪些下游使用、调用了多少次和数据服务是否稳定等信息。
- **开发效率大幅提升，服务稳定性大幅提高**：通过统一服务平台可以在 1 小时内完成一个接口的配置化开发，与此同时，接口稳定性统一运维，服务稳定性有了很好的保障。



3.1.4 统一用户产品入口（One Portal）

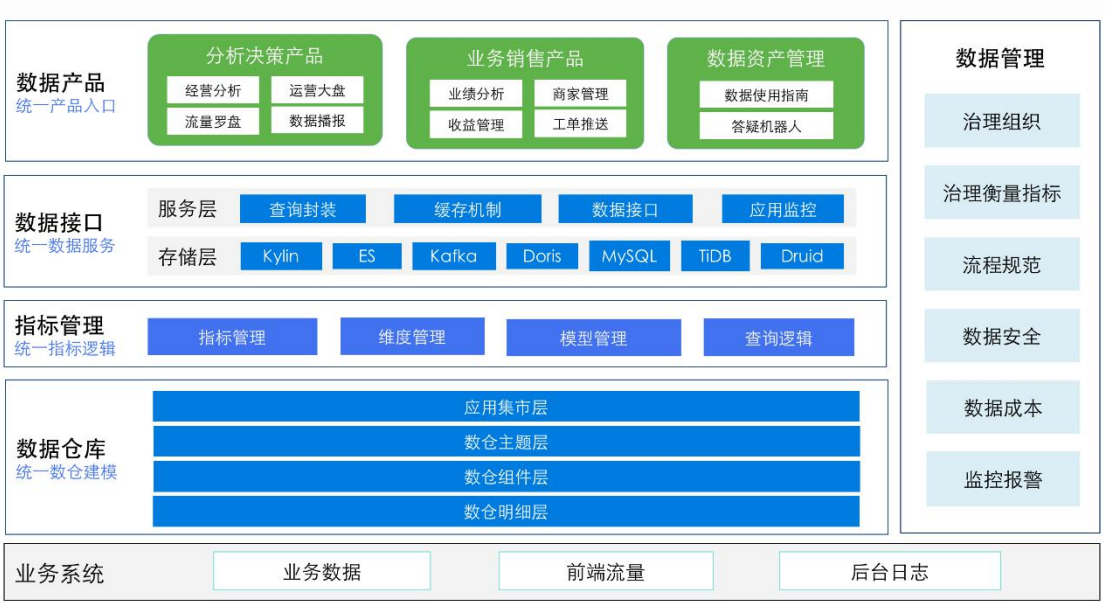
如果不加控制，数据产品就会建设得越来越多。酒旅业务在 2018 年有超过 10 个数据相关产品的入口，用户很难快速地找到自己想要查的数据产品和报表。不同产品面对的用户不一样，数据的使用场景和展示方式也各不相同，业务方在使用数据时不知道从哪里能看到最全面的数据产品。

此外，也存在因为适用场景不一样，导致面向不同用户的数据逻辑不同的情况，比如某些业务同学查看的 GMV 不包含民宿数据，但是商业分析团队要看的 GMV 是包含民宿数据的。为了能够让业务方能够在一个数据产品门户中找到更全面的数据，且这个产品门户中多个产品的数据逻辑是一致的，我们将数据门户按照使用用户和应用场景划分为 3 类：

- 决策分析使用“大圣”（美团内部的数据平台），面向管理者和商业分析团队，所有业务管理者和商业分析团队成员需要的数据都可以从大圣数据产品里查看。
- 业务数据查询使用“天狼” （美团内部的数据平台），用户主要是销售，在天狼里能查看销售所需的各种数据。
- 数据资产信息查询使用“大禹”（美团内部的数据平台），用户是研发人员和检索数据信息的业务方，在大禹数据门户里可以找到数据资产的信息，能更快地找到想要的数据，更全面地了解相关的元数据。

3.1.5 整体系统架构

整体的技术架构分为三层，从统一数据建模到统一指标逻辑、统一数据服务和统一产品入口，整体保障了数据的质量，同时配合数据管理的组织保障体系和流程规范，将整体数据质量相关的架构搭建起来。



3.2 数据运营效率

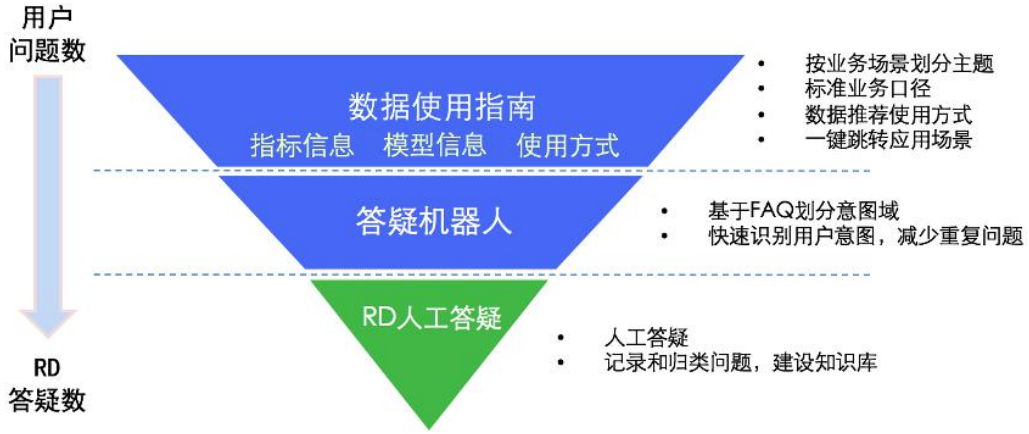
数据工程师在日常工作中的主要工作包括两大部分：数据开发和数据运营。我们在前面介绍了通过数据开发和指标管理相关的工具系统建设，开发效率得到了大幅提升。而数据运营是另一大类工作，他们的主要时间投入在数据使用咨询和数据问题答疑，大概占数据工程师日常工作 5%～10%的时间。

数据工程师日常投入到运营的人力多的主要原因是信息不对称和信息检索能力弱，数据团队建设了很多数据模型和数据产品，但是用户不知道怎么快速地找到和使用这些数据，问题主要体现在下面 3 个方面：

- **找数难**：所需要的数据有没有？在哪里能找到？
- **看不懂**：数据仓库是以数据表和报表等方式提供，数据的逻辑和含义不够清晰易懂。
- **不会用**：数据指标的查询逻辑是什么？多个表怎么关联使用？

3.2.1 方案思路

数据团队通过数据资产信息的系统化的方式建设易用的数据检索产品，帮助用户更快捷、更方便地找到数据，并指导用户正确地使用数据，提高数据信息的易用性，以此减少数据工程师的数据答疑和运维时间。实现策略是通过用户的问题分类，通过数据信息系统化的方式分类解答 80%的问题，最后少量的问题透传到研发人员再进行人工答疑。系统化方式主要分两层，数据使用智能和数据答疑机器人。

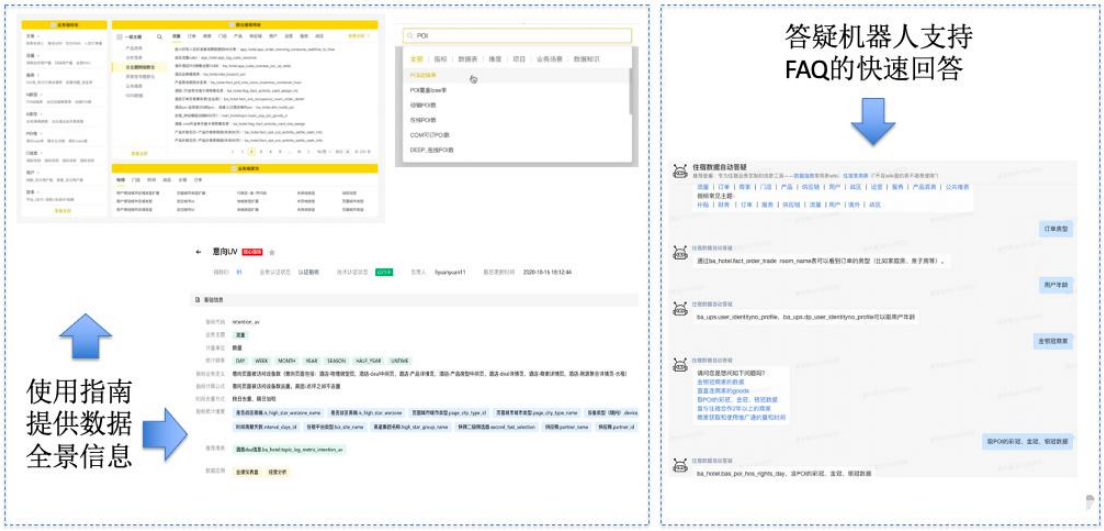


3.2.2 数据使用指南系统

数据使用指南的定位是业务数据信息的知识白皮书，提供最新、最全、最准确的指标口径、项目指标体系、数据表用法等信息，以简洁、流畅的操作支持数据指南中的内容及时更新，降低业务方的数据答疑和数据使用成本。

数据使用指南通过把业务场景和数据使用场景打通，从业务场景分析到使用到的数据表、指标和数据产品打通，在系统中能够快速找到数据表、指标定义、数据查询 SQL、指标所在数据产品等信息，一站式解决数据查找、使用和分析的全部场景。主要功能包括指标信息和数据表信息及使用。

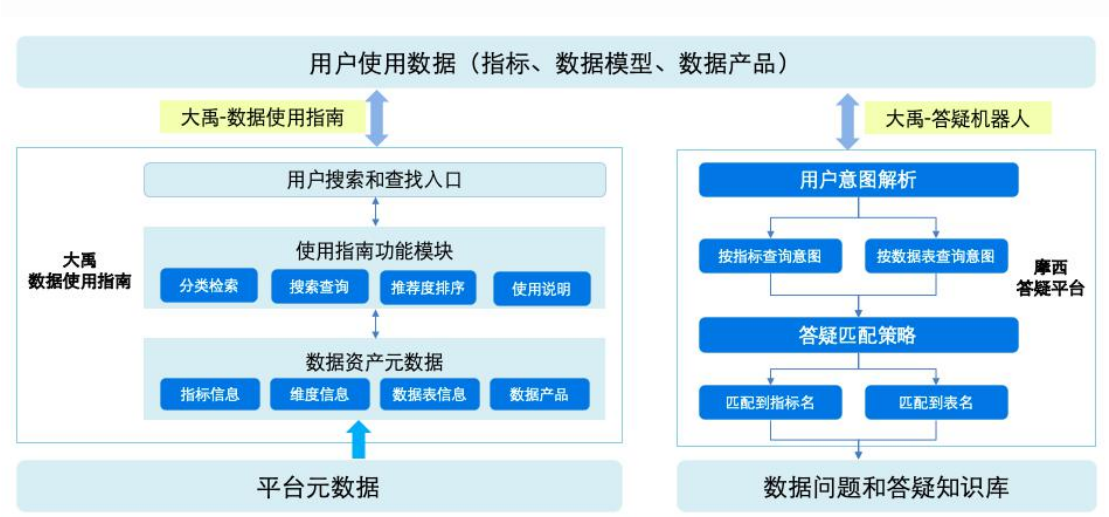
- **指标信息**：包括业务分类指标和指标的详细信息，在指标详细信息页面可以查看指标定义、指标使用场景、指标统计维度、指标对应数据表、指标所在数据产品和指标的 SQL 查询示例等信息，把指标信息与数据表和数据产品关联，方便用户快速根据指标信息查找到数据。
- **数据表信息及使用方式**：包括数据表的基础信息、表的使用推荐度、SQL 查询样例、数据更新时间和数据就绪时间等信息，帮助使用者快速定位需要的数据表和数据 SQL 的查询使用。



3.2.3 数据答疑机器人

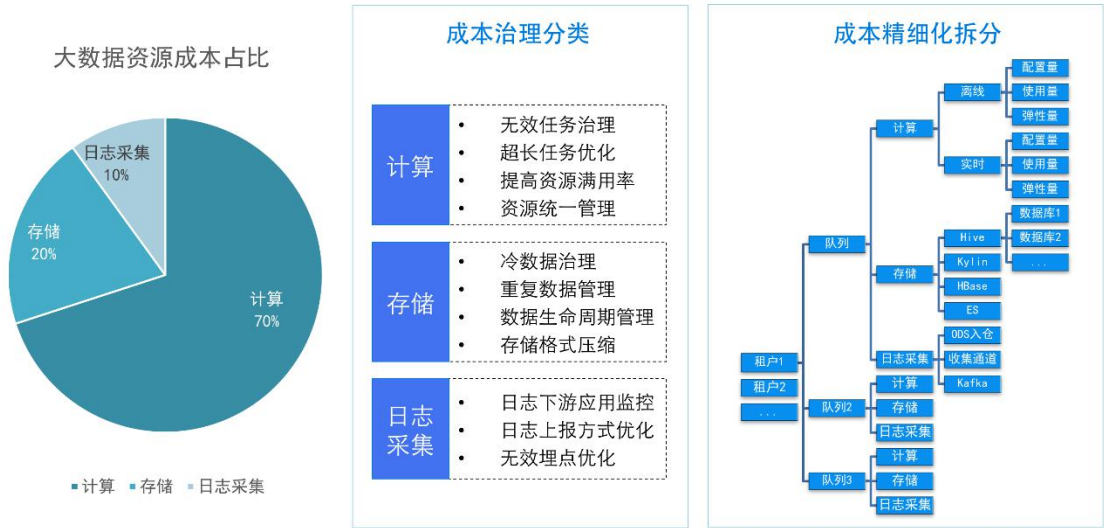
用户在使用数据时，经常咨询数据工程师一些问题，比如想找的数据在哪个表？指标怎么取？业务系统的一个字段怎么在数仓里面取到？很多问题会被重复问到，每次解答都需要研发人员花费一定的时间，而通过 Wiki 的方式维护效果较差，于是我们考虑用自动化答疑的方式，把数据工程师在日常答疑过程中积累问题和答案，通过一定的规则匹配，当再次被问到时系统可以自动地给出解答。

使用日常答疑中积累的咨询问题和答案作为基础答疑知识库，数据答疑机器人使用美团 AI 平台的摩西机器人搭建，配合问题答疑的策略，实现对历史已有问题和答案通过搜索匹配后发送给用户，具体实现方式如下：



3.3 数据成本

大数据的主要成本构成有 3 大部分，计算资源、存储资源和日志采集资源，其中计算资源和存储占总成本超过 90%，我们的数据成本治理主要是针对大数据计算和存储这两个部分。



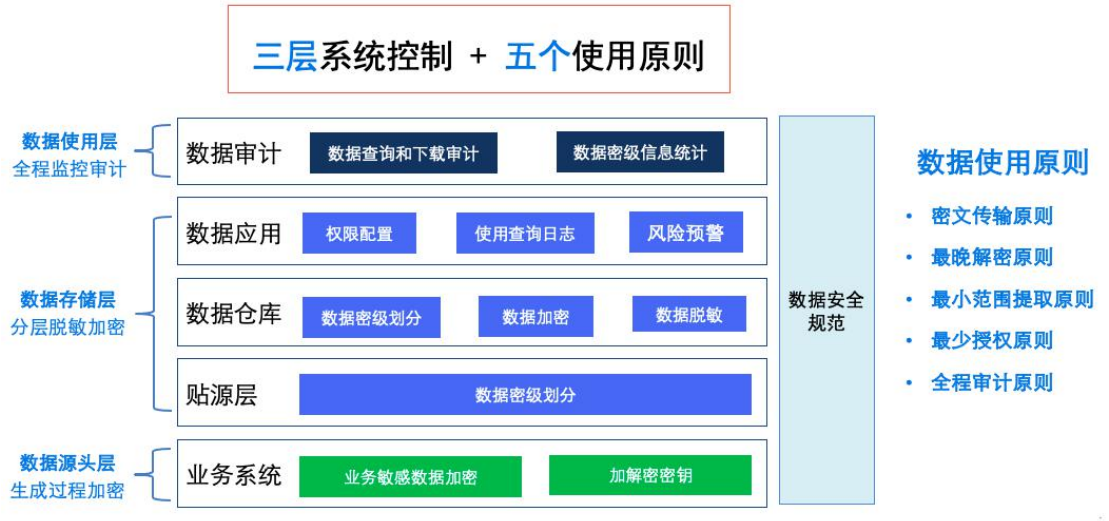
大数据成本优化方案

- **计算资源**
  - 无效任务清理，通过任务生产出来数据的使用情况判断是否为无效任务，通过下线无效任务，减少任务执行使用的计算资源。

    - 超长任务优化，经过任务的计算资源使用数据可以发现，某几个大任务在执行时会占用大部分的计算资源，导致其他任务执行时间变长，或者占用配置外的弹性计算资源，导致计算成本增加。数据组会统计和监控每天任务的执行情况，发现执行时间长（超过 2 个小时）或者占用资源多的任务会及时进行优化。
    - 分散利用计算资源，数仓的夜间批处理任务使用计算资源的实际一般都集中在早晨 2 点到上午 10 点前，这就导致在一天中只有三分之一的资源被充分利用，而且这段时间内通常资源都是不够用的，需要使用平台提供的配置外弹性资源。而其他时间段的计算资源闲置，对资源有较大的浪费。为了把全天的资源都有效地利用起来，我们会把一些对就绪时间不敏感的任务（比如算法挖掘、用户标签、数据回刷等）放到 10 点之后，把配置的计算资源充分利用起来。
    - 租户拆分和整合统一管理，提高资源池总量和资源总体的使用率。
- **存储资源**
    - 数仓架构优化和重构：通过统一数仓建模规范，把相似或相同模型进行整合和去重，确保每个主题数据只保留一份。
    - 数据存储压缩：在数据仓库建设初期，很多 Hive 表的存储格式是 txt，通过压缩为 ORC 格式可以减少大量的存储空间。
    - 冷数据处理：把数据分为冷、热两大类数据，通过每天对全部数仓表扫描识别出冷数据，发给数据负责人及时处理。
    - 数据生命周期控制：按照数仓分层的应用场景配置数据的生命周期，明细数仓层保留的全部历史数据，主题层保留 5 年数据，应用层保留 1～3 年数据。通过数据生命周期控制，极大地减少了数据存储成本。
- **日志采集资源**
    - 下线冷数据的上游日志数据收集任务，数据收集费用主要来自两类数据，业务系统数据库的 Log 同步和后台日志数据收集，通过对收集数据的使用情况监控，及时下线下游无应用的数据收集任务。

3.4 数据安全

数据资产对业务来说既是价值，也是风险。数据安全作为业务部门“事关生死”的核心工作，在技术架构上会从数据产生到数据应用各个环节进行控制，保障数据应用事前有控制、事中有监控和事后有审计。数据安全控制从业务系统开始对用户高敏感数据加密，在数仓进行分级和脱敏，在应用层做密文数据权限和密钥权限的双重保障，管控用户相关的高敏感数据，按照三层系统控制加五个使用原则实现如下：



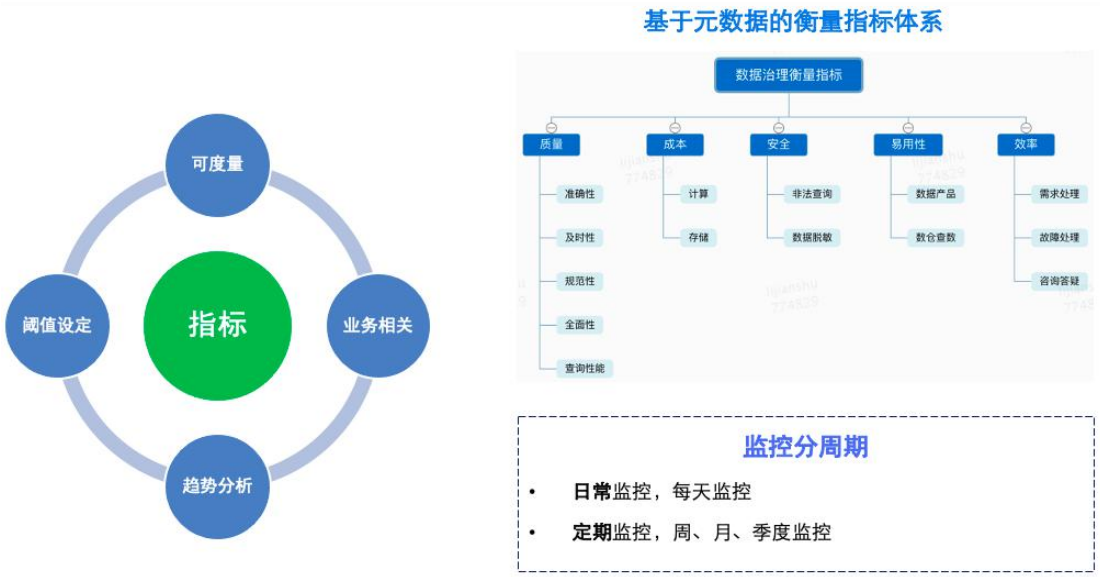
4. 衡量指标

业务部门在业务发展初级就会建立指标体系，并使用数据指标对各个业务过程做精细化的分析，衡量业务目标的达成情况和行动的执行程度。数据治理也需要一套成熟稳定的衡量指标体系，对数据体系做到长期、稳定和可量化的衡量。我们通过制定体系化的数据衡量指标体系，来及时监测数据治理过程中哪些部分做的好，哪些部分还有问题。

4.1 衡量指标建设

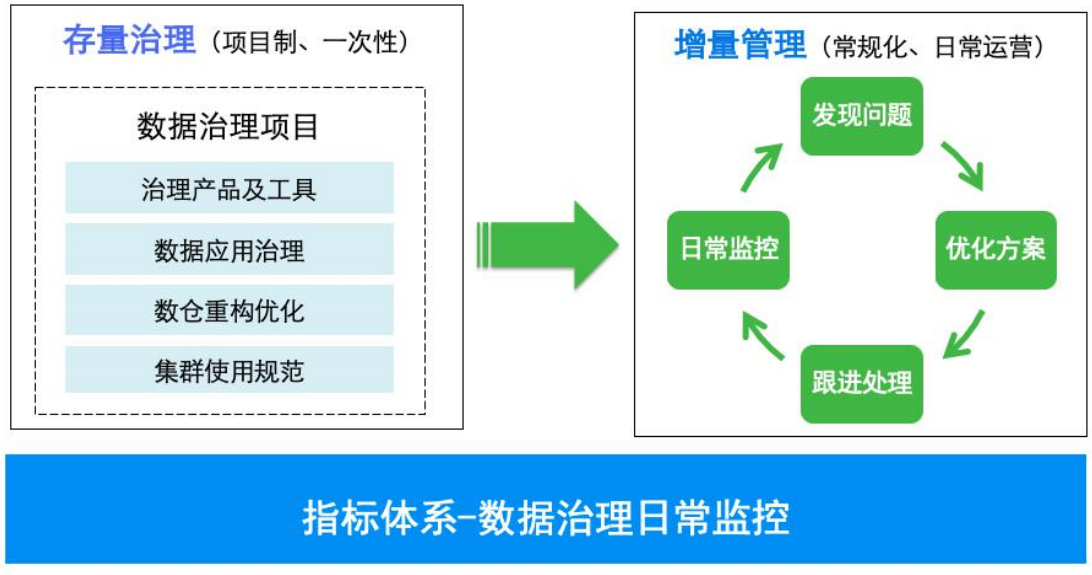
为了能够不重不漏地把指标都建立起来，我们从 2 个方面进行考虑：

- 技术分类，按照数据团队关注的问题和目标，把数据治理的指标体系分成质量、成本、安全、易用性和效率这 5 大类。
- 数据流环节，分别从数据的采集、生产、存储、指标管理、应用和销毁等环节监控关注的指标。



4.2 衡量指标保障数据治理

根据 PDCA 原则，将数据治理作为日常的运营项目做起来，底层依赖数据指标体系进行监控，之上从发现问题到提出优化方案，然后跟进处理，再到日常监控，构成一个完整的循环。



5. 治理效果总结

数据治理覆盖了数据生命周期全链路，通过围绕数据从产生到价值消亡全部生命周期，建立数据治理组织、制定治理衡量体系和建设治理技术系统来达到数据治理目标。经过体系化的数据治理，数据系统的治理、成本、安全和运营效率都有了比较大的改善。

- **数据质量**：技术架构优化后，通过标准化规范和系统保障数据的准确性，并在治理过程中清除和整合了历史冗余数据，数据质量问题有很大的改善。2019 年数据生产任务的增长率比 2018 年减少了 60%左右。
- **数据成本**：经过数据成本优化后，在支持 2019 年酒旅业务高速增长的同时，大数据的单均成本费用降低了 40%左右。
- **数据安全**：通过业务系统数据加密和数据仓库数据脱敏，双重保障高敏感数据安全，避免数据泄露。通过数据安全规范和数据敏感性的宣导，加强业务同学的数据安全意识，业务没有严重数据安全问题的发生。
- **运营效率**：运营工具化减少了研发同学超过 60%的日常答疑时间，极大地减少了研发同学工作被打扰的次数，提高了开发效率。

三、未来规划

数据治理分为三个大阶段：被动治理、主动治理、自动治理。

- 第一阶段我们做的是**被动治理**，也就是阶段性治理，缺少统筹考虑，主要是基于单个问题的治理，而且治理之后过一段时间可能还要做重复治理。这个阶段更多是人治，一个项目成立，协调几个人按照项目制完成，没有体系规划，也没有组织保障。
- 第二阶段是**主动治理**，有长期的统筹规划，能覆盖到数据生命周期的各个链路，在治理过程中把一些手段和经验流程化、标准化、系统化，长期解决一些数据问题，让数据治理长期可控。

- 第三阶段是**自动治理**，也是智能治理，在长期规划和数据生命周期各环节链路确定好之后，把已经有的经验、流程和标准做成策略。一旦出现问题，自动监控，通过一些系统化的方式解决。自动治理的第一步还是治理方案的落地和策略化，这非常依赖于元数据，把数据治理各个过程中的一些经验技术都沉淀起来。做完策略沉淀之后做自动化，把策略用工具的方式实现，当系统发现数据有问题时，自动就去处理。

目前，美团酒旅业务数据治理处在第二阶段和第三阶段之间，虽然有整体治理计划、技术架构和组织保障，但仍需要投入一定的人力去做。未来，数据治理会继续朝着智能化的方向进行探索，真正把自动化治理工作做得更好。

十二、DataMan-美团旅行数据质量监管平台实践

背景

数据，已经成为互联网企业非常依赖的新型重要资产。数据质量的好坏直接关系到信息的精准度，也影响到企业的生存和竞争力。Michael Hammer（《Reengineering the Corporation》一书的作者）曾说过，看起来不起眼的数据质量问题，实际上是拆散业务流程的重要标志。 数据质量管理是测度、提高和验证质量，以及整合组织数据的方法等一套处理准则，而体量大、速度快和多样性的特点，决定了大数据质量所需的处理，有别于传统信息治理计划的质量管理方式。

本文将基于美团点评大数据平台，通过对数据流转过程中各阶段数据质量检测结果的采集分析、规则引擎、评估反馈和再监测的闭环管理过程出发，从面临挑战、建设思路、技术方案、呈现效果及总结等方面，介绍美团平台酒旅事业群（以下简称美旅）数据质量监管平台 DataMan 的搭建思路和建设实践。

挑战

美旅数据中心日均处理的离线和实时作业高达数万量级， 如何更加合理、高效的监控每类作业的运行状态，并将原本分散、孤岛式的监控日志信息通过规则引擎集中共享、关联、处理；洞察关键信息，形成事前预判、事中监控、事后跟踪的质量管理闭环流程；沉淀故障问题，搭建解决方案的知识库体系。在数据质量监管平台的规划建设中，面临如下挑战：

- 缺乏统一监控视图，离线和实时作业监控分散，影响性、关联性不足。
- 数据质量的衡量标准缺失，数据校验滞后，数据口径不统一。
- 问题故障处理流程未闭环，“点”式解决现象常在；缺乏统一归档，没有形成体系的知识库。
- 数据模型质量监控缺失，模型重复，基础模型与应用模型的关联度不足，形成信息孤岛。
- 数据存储资源增长过快，不能监控细粒度资源内容。

DataMan 质量监管平台研发正基于此，以下为具体建设方案。

解决思路

整体框架

构建美旅大数据质量监控平台，从可实践运用的视角出发，整合平台资源、技术流程核心要点，重点着力平台支持、技术控制、流程制度、知识体系形成等方向建设，确保质量监控平台敏捷推进落地的可行性。数据质量监控平台整体框架如图 1 所示：

图 1 质量监控平台整体框架图

建设方法

以数据质量检核管理 PDCA 方法论，基于美团大数据平台，对数据质量需求和问题进行全质量生命周期的管理，包括质量问题的定义、检核监控、发现分析、跟踪反馈及知识库沉淀。数据质量 PDCA 流程图如图 2 所示：

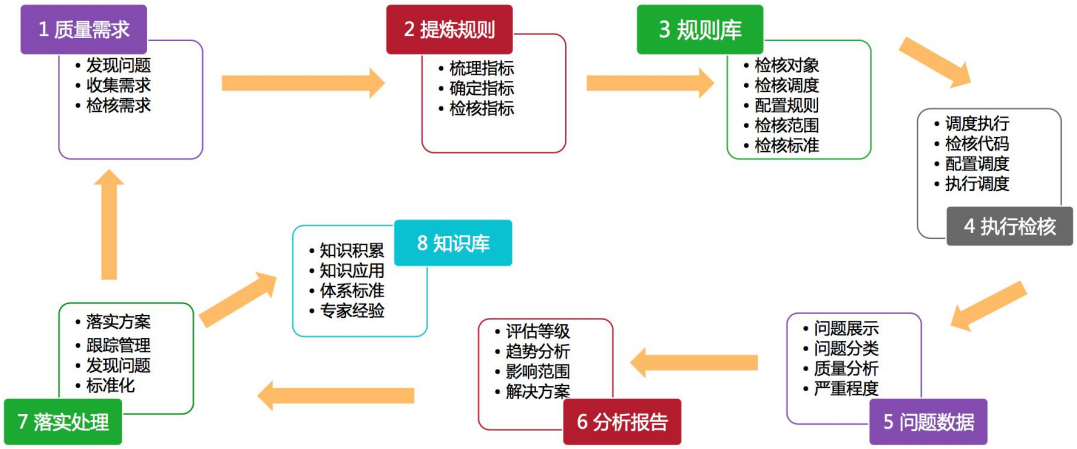


图 2 数据质量 PDCA 流程图

关键流程：

质量监管平台建设实践应用及价值体现，离不开管理流程、技术实现和组织人员的紧密结合，主要包含如下 8 大流程步骤：

1. 质量需求：发现数据问题；信息提报、收集需求；检核规则的需求等。
2. 提炼规则：梳理规则指标、确定有效指标、检核指标准确度和衡量标准。
3. 规则库构建：检核对象配置、调度配置、规则配置、检核范围确认、检核标准确定等。
4. 执行检核：调度配置、调度执行、检核代码。
5. 问题检核：检核问题展示、分类、质量分析、质量严重等级分类等。
6. 分析报告：数据质量报告、质量问题趋势分析，影响度分析，解决方案达成共识。
7. 落实处理：方案落实执行、跟踪管理、解决方案 Review 及标准化提炼。
8. 知识库体系形成：知识经验总结、标准方案沉淀、知识库体系建设。

质量检核标准

- 完整性：主要包括实体缺失、属性缺失、记录缺失和字段值缺失四个方面；
- 准确性：一个数据值与设定为准确的值之间的一致程度，或与可接受程度之间的差异；
- 合理性：主要包括格式、类型、值域和业务规则的合理有效；
- 一致性：系统之间的数据差异和相互矛盾的一致性，业务指标统一定义，数据逻辑加工结果一致性；
- 及时性：数据仓库 ETL、应用展现的及时和快速性，Jobs 运行耗时、运行质量、依赖运行及时性。

大数据平台下的质量检核标准更需考虑到大数据的快变化、多维度、定制化及资源量大等特性，如数仓及应用 BI 系统的质量故障等级分类、数据模型热度标准

定义、作业运行耗时标准分类等和数仓模型逻辑分层及主题划分组合如下图 3 所示。

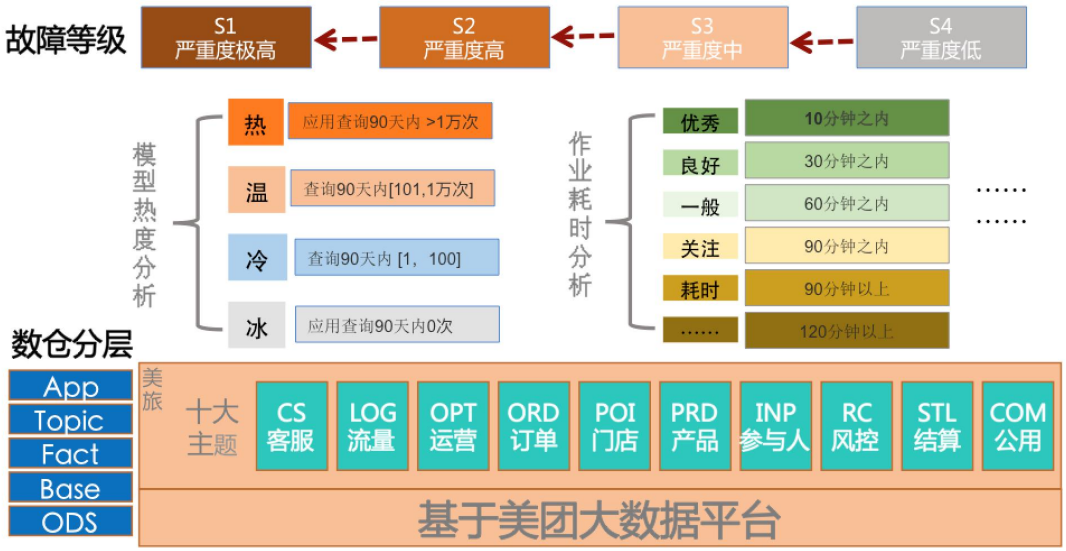


图 3 质量检核标准图

美旅数仓划分为客服、流量、运营、订单、门店、产品、参与人、风控、结算和公用等十大主题，按 Base、Fact、Topic、App 逻辑分层，形成体系化的物理模型。从数据价值量化、存储资源优化等指标评估，划分物理模型为热、温、冷、冰等四类标准，结合应用自定义其具体标准范围，实现其灵活性配置；作业运行耗时分为：优、良、一般、关注、耗时等，每类耗时定义的标准范围既符合大数据的特性又可满足具体分析需要，且作业耗时与数仓主题和逻辑分层深度整合，实现多角度质量洞察评估；针对数万的作业信息从数据时效性、作业运行等级、服务对象范围等视角，将其故障等级分为：

- S1：严重度极高；
- S2：严重度高；
- S3：严重度中；
- S4：严重度低等四项标准。

各项均对应具体的实施策略。整体数据质量的检核对象包括离线数仓和实时数据。

监管核心点

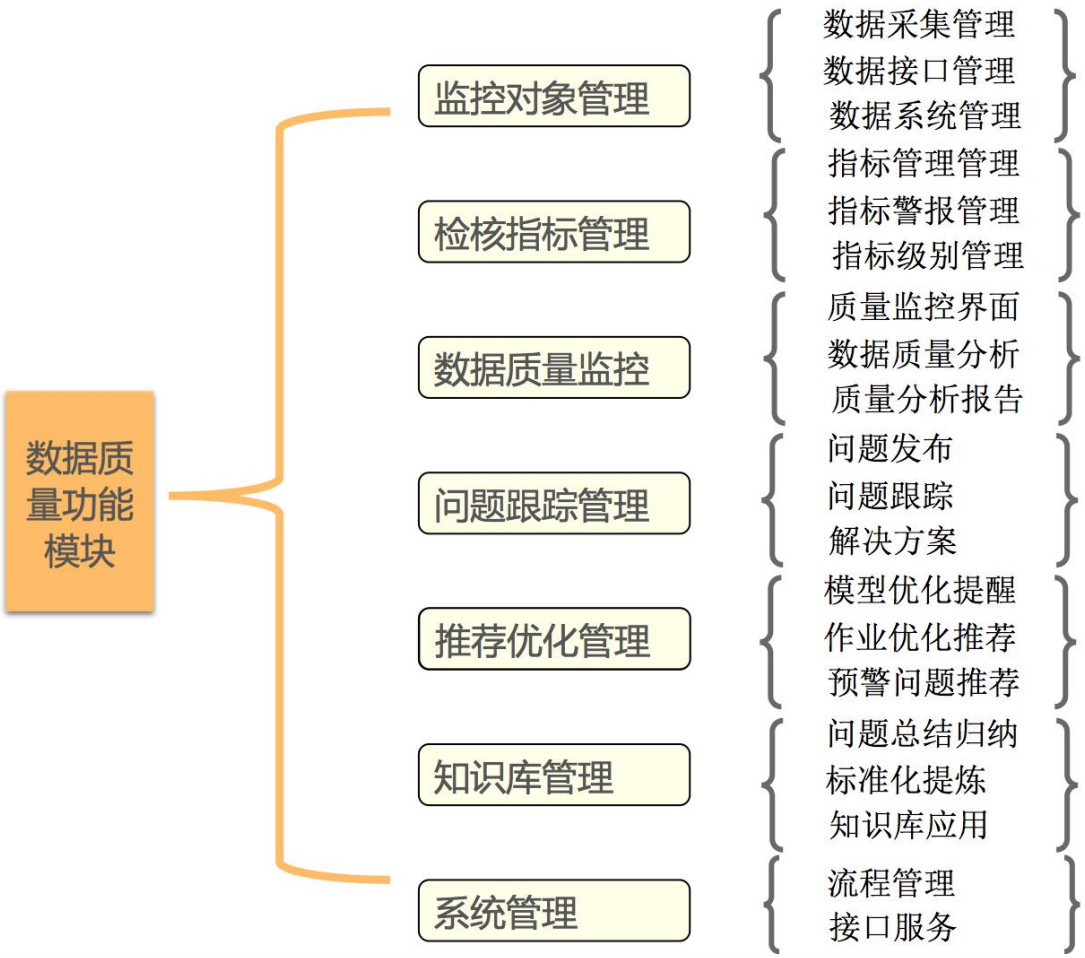


图 4 数据质量监管功能图

数据质量功能模块设计的主要功能如上图 4 所示，包括：监控对象管理、检核指标管理、数据质量过程监控、问题跟踪管理、推荐优化管理、知识库管理及系统管理等。其中过程监控包括离线数据监控、实时数据监控；问题跟踪处理由问题发现（支持自动检核、人工录入）、问题提报、任务推送、故障定级、故障处理、知识库沉淀等形成闭环流程。

管理流程

流程化管理是推进数据问题从发现、跟踪、解决到总结提炼的合理有效工具。质量管理流程包括：数据质量问题提报、数据质量问题分析、故障跟踪、解决验证、数据质量评估分析等主要环节步骤；从干系人员的角度分析包括数据质量管理人员、数据质量检查人员、数据平台开发人员、业务及 BI 商分人员等，从流程步骤到管理人员形成职责和角色的矩阵图。如图 5 所示：

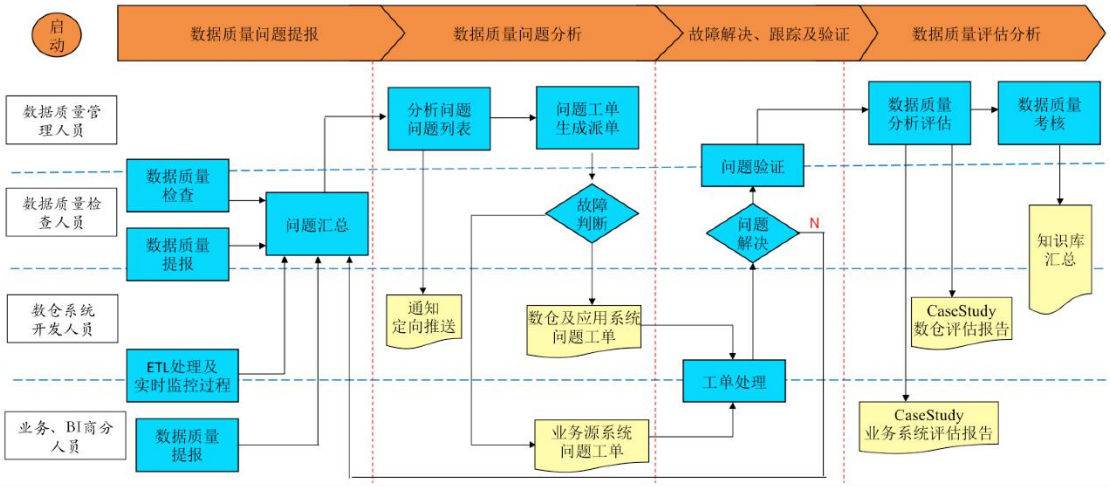


图 5 数据质量流程图

问题汇总：数据质量提报、ETL 处理及监控过程上报、数据质量检查点等多方来源，其中 ETL 处理部分为程序自动化上报，减少人为干预。 **问题分析**：通过规定的角色和岗位的人员对汇总问题分析和评估，由统一公共账号自动推送提醒消息至责任人。 **问题工单**：对采集的问题经过分析归类，主要划为信息提示和故障问题两大类，信息提示无需工单生成，故障问题将产生对应的工单，后推送至工单处理人。 **故障定级**： 针对生成的问题工单判断其故障级别，其级别分为：S1、S2、S3、S4 等四类（如图 3 所述），针对尤为严重的故障问题需 Review 机制并持续跟踪 CaseStudy 总结。 **知识库体系**：从由数据问题、解决方案、典

型案例等内容中，提炼总结形成标准化、完备知识库体系，以质量问题中提炼价值，形成标准，更加有效的指导业务、规范业务，提高源头数据质量，提升业务服务水平。

质量流程管理：

- **流程原则**：统一流程、步骤稳定。
- **权限控制**：流程节点与人员账户号绑定，若节点未设置人员账户即面向所有人员，否则为规定范围的人员。
- **权限管理**：可结合美团平台的 UPM 系统权限管理机制。

技术方案

总体架构

DataMan 系统建设总体方案基于美团的大数据技术平台。自底向上包括：检测数据采集、质量集市处理层；质量规则引擎模型存储层；系统功能层及系统应用展示层等。整个数据质量检核点基于技术性、业务性检测，形成完整的数据质量报告与问题跟踪机制，创建质量知识库，确保数据质量的完整性（Completeness）、正确性（Correctness）、当前性（Currency）、一致性（Consistency）。

总体架构图如图 6 所示：

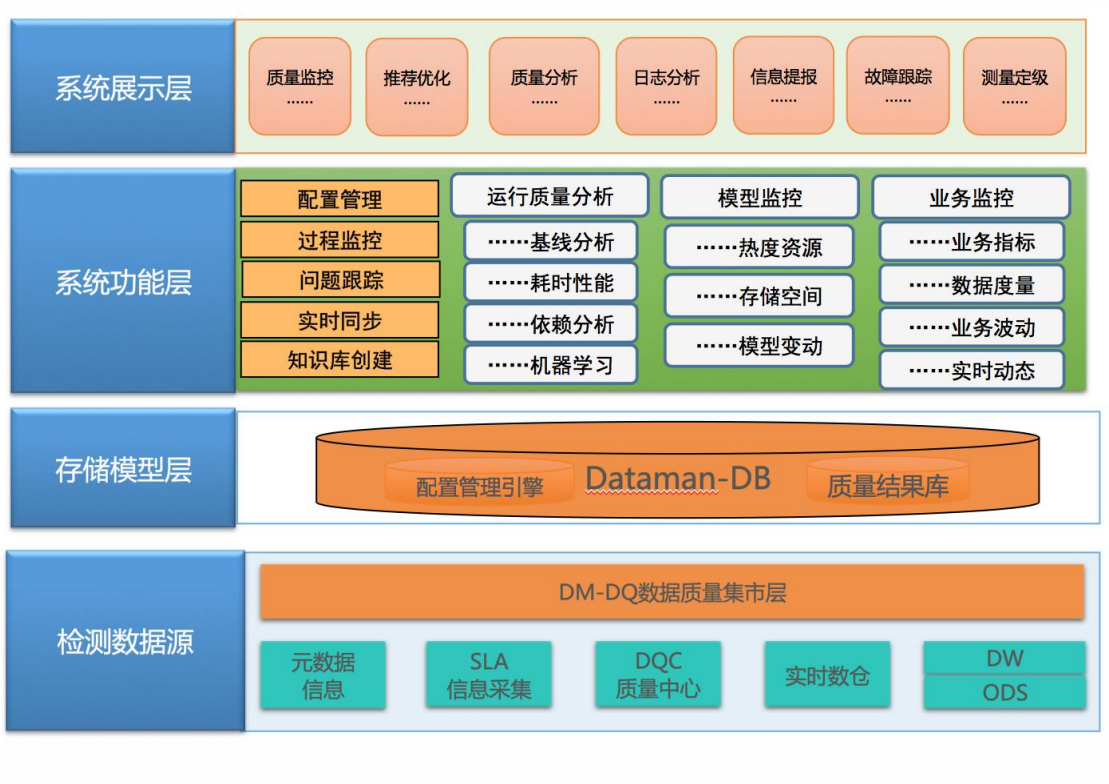


图 6 质量监管 DataMan 总体架构图

- **数据源及集市层**：首先采集数据平台质量相关的元数据信息、监控日志信息、实时日志、检测配置中心日志、作业日志及调度平台日志等关键的质量元数据；经数据质量集市的模型设计、监控对象的分类，加工形成完整、紧关联、多维度、易分析的数据质量基础数据模型，为上层质量应用分析奠定数据基础。数据来源自大数据平台、实时数仓、调度平台等，涉及到 Hive、 Spark、Storm、 Kafka、MySQL 及 BI 应用等相关平台数据源；
- **存储模型层**：主要功能包括规则引擎数据配置、质量模型结果存储；以数据质量监控、影响关联、全方位监控等目标规则引擎的推动方式，将加工结果数据存储至关系型数据库中，构成精简高质数据层；
- **系统功能层**：包括配置管理、过程监控、问题跟踪、故障流程管理、实时数据监控、知识库体系的创建等；处理的对象包括日志运行作业、物理监控模型、业务监控模型等主要实体；
- **系统展示层**：通过界面化方式管理、展示数据质量状态，包括质量监控界面、推荐优化模块、质量分析、信息展示、问题提报、故障跟踪及测量定级、系统权限管理等功能。

技术框架

前后端技术

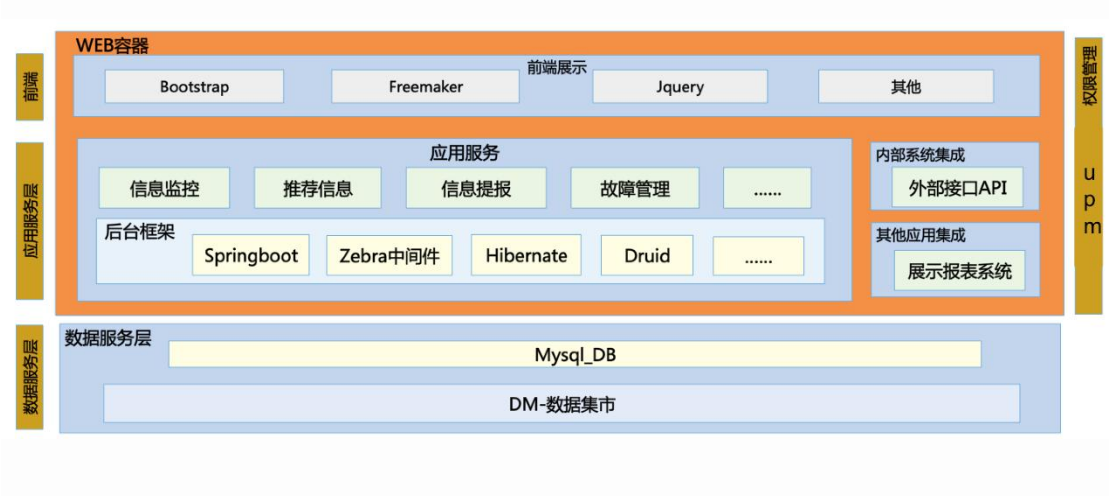


图 7 技术架构图

DataMan 应用系统其前端框架（如上图 7）基于 Bootstrap 开发，模板引擎为 FreeMarker，Tomcat（开发环境）作为默认 Web 容器，通过 MVC 的方式实现与应用服务层对接。Bootstrap 的优势基于 jQuery，丰富的 CSS、JS 组件，兼容多种浏览器，界面风格统一等；FreeMarker 为基于模板用来生成输出文本的引擎。后台基于开源框架 Spring4，Spring Boot，Hibernate 搭建，其集成了 Druid，Apache 系列和 Zebra 等数据库访问中间件等，为系统的功能开发带来更多选择和便利。

Zebra 中间件

系统数据库连接采用中间件 Zebra，这是美团点评 DBA 团队推荐的官方数据源组件，基于 JDBC、API 协议上开发出的高可用、高性能的数据库访问层解决方

案；提供如动态配置、监控、读写分离、分库分表等功能。Zebra 整体架构如图 8 所示：

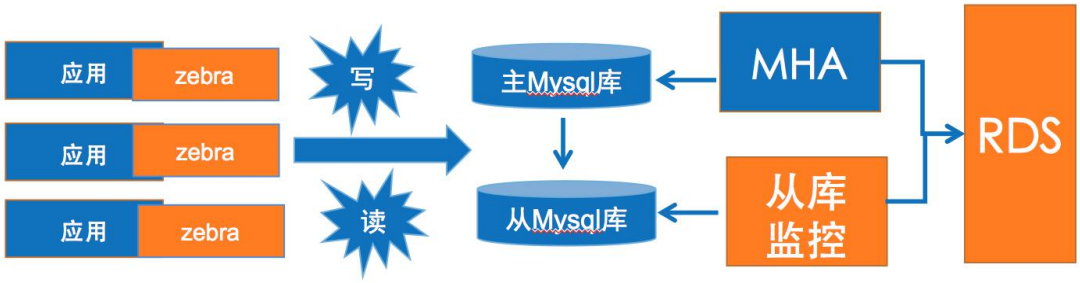


图 8 Zebra 架构图

Zebra 客户端会据路由配置直连到 MySQL 数据库进行读写分离和负载均衡。RDS 是一站式的数据库管理平台，提供 Zebra 的路由配置信息的维护；MHA 组件和从库监控服务分别负责主库和从库的高可用。Zebra 支持丰富的底层连接池；统一源数据配置管理；读写分离和分库分表；数据库的高可用。

数据模型

整个质量监管平台数据流向为数据质量元数据信息采集于美团平台，包括数据仓库元数据信息、质量检测元数据、调度平台日志信息、监控日志及实时元数据信息等，加工形成独立数据质量的集市模型，以此支撑应用层系统的数据需求。应用层系统数据库采用关系型数据库存储的方式，主要包含了规则配置管理信息、数据质量结果库等信息内容。数据流向层级关系图如下：

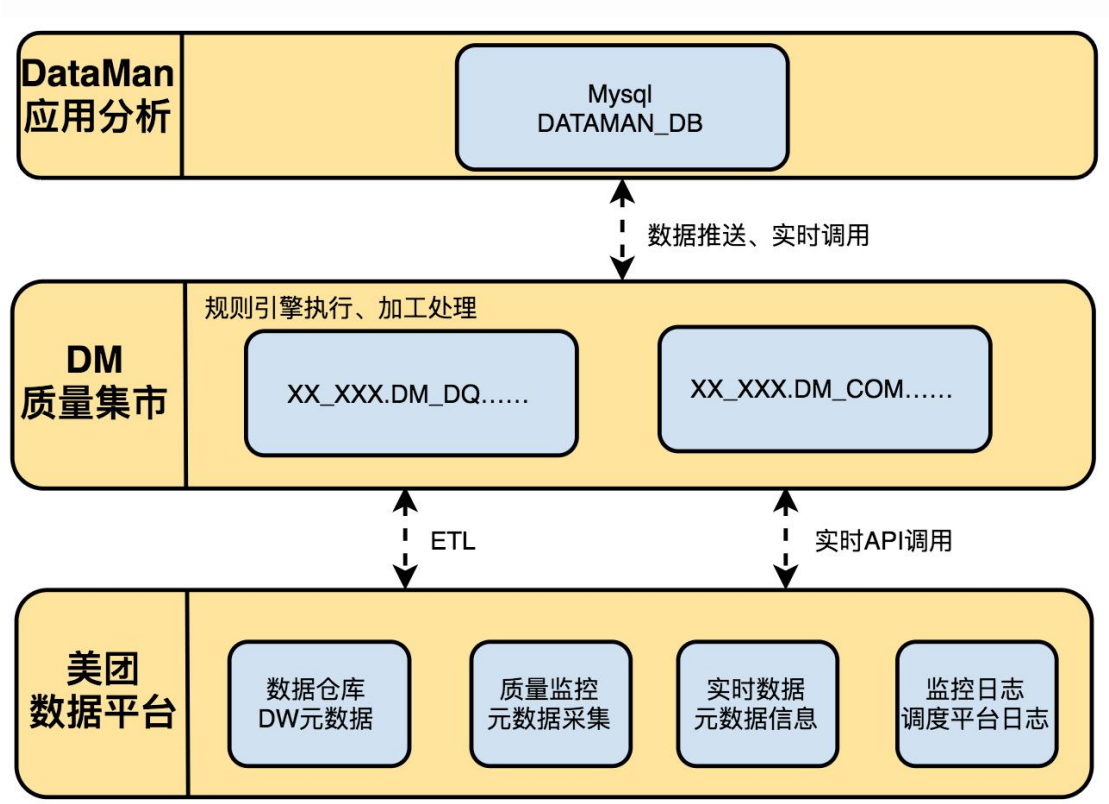


图 9 数据流向层级图

数据平台层：基于美团大数据平台的数据质量元数据是质量分析和监管的来源，是整个系统最基础重要资源信息，此数据主要包括：数仓元数据信息，如数仓模型表基本信息、表存储空间资源信息、表分区信息、节点信息、数据库 meta 信息、数据库资源信息等；运行作业调度日志信息，如作业基本信息、作业运行资源信息、作业调度状态信息、作业依赖关系信息及作业调度日志监控信息等；质量检测元数据信息主要来源于 SLA、DQC（美团内部系统）检测结果的信息。实时元数据采集于调度平台实时作业运行的 API 接口调用分析。

质量集市层：DM 数据质量集市的独立创建是依托基础元数据信息，根据质量监管平台配置的引擎规则 ETL 加工形成。规则库引擎如数仓应用主题的划分规则、数仓逻辑分层约束、数据库引擎分类、模型使用热度等级、模型存储空间分类、

资源增长等级、历史周期分类、作业重要级别、作业运行耗时等级、作业故障分类、及数据质量标准化定义等；在管理方向上，如模型或作业所属的业务条线、组织架构、开发人员等；在时效上分为离线监控数据、实时数据集市等。从多个维度交叉组合分析形成模型类、作业类、监控日志类、实时类等主题的等易理解、简单、快捷的数据质量集市层，强有力的支撑上层应用层功能的数据需求。数据质量集市 DM 主要模型如图 10 所示：

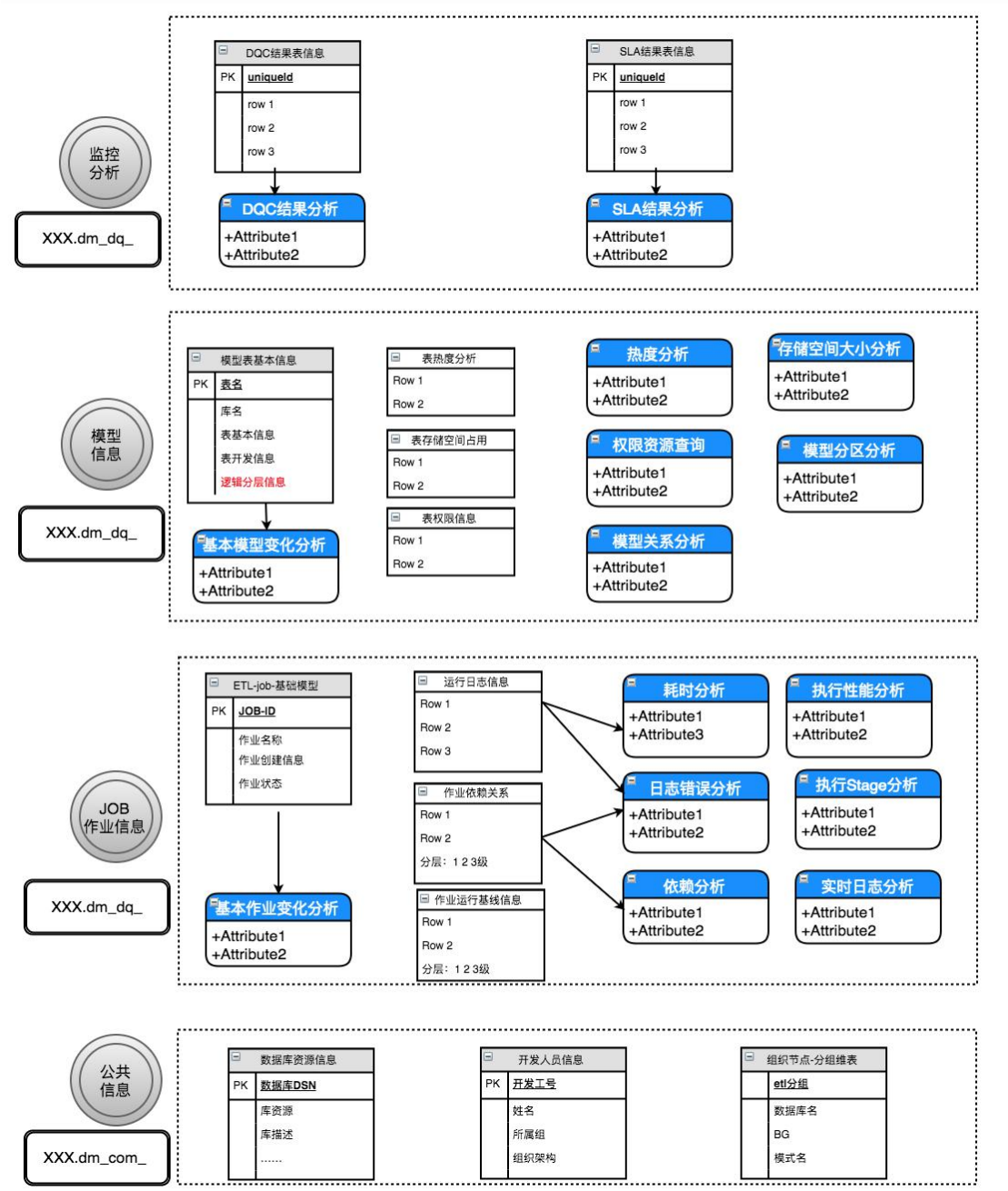


图 10 数据质量集市模型图

- 模型设计："统一规范、简单快捷、快速迭代、保障质量"，基于美团平台元数据、平台日志、实时数据接口等来源，通过制定的规则、标准，形成可衡量、可评估的数据质量集市层，主要包含公共维度类、模型分析类、作业监控类、平台监控类等主要内容；
- 实时数据：针对实时作业的监控通过 API 接口调用，后落地数据，实时监控作业运行日志状态；
- 数据加工：基于美团平台离线 Hive、Spark 引擎执行调度，以数仓模型分层、数仓十大主题规则和数据质量规则库等为约束条件，加工形成独立的数据集市层。

应用分析层：应用层系统数据采用关系型数据库（MySQL）存储的方式，主要包含了规则配置管理信息、数据质量分析结果、实时 API 落地数据、故障问题数

据、知识库信息、流程管理及系统管理类等信息内容，直接面对前端界面的展示和管理。

系统展示

数据质量 DataMan 监控系统一期建设主要实现的功能包括：个人工作台、信息监控、推荐信息、信息提报、故障管理、配置管理及权限系统管理等。系统效果如图 11 所示：

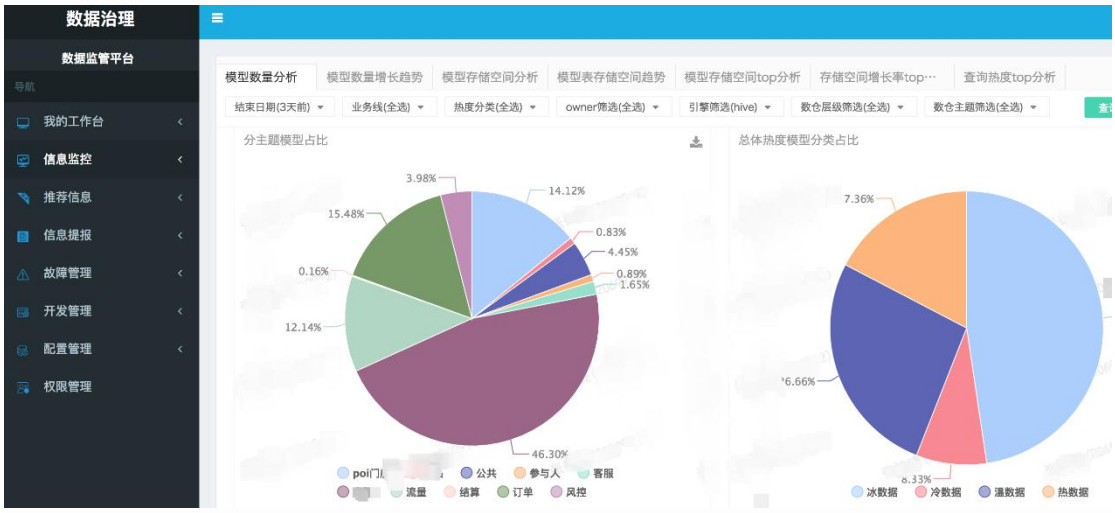


图 11 系统效果图

个人工作台

在系统中将个人待关注、待处理、待优化、待总结等与个人相关的问题和任务形成统一的工作平台入口，通过公共账号推送的方式，第一时间提醒个人，通知反馈问题的提出者，保障问题可跟踪，进度可查询，责任到人的工作流程机制。

离线监控

系统可定时执行模型监控、作业监控、平台日志监控等元数据质量规则引擎，开展数据仓库主题模型、逻辑层级作业、存储资源空间、作业耗时、CPU 及内存资源等细化深度分析和洞察；按照质量分析模型，以时间、增长趋势、同环比、历史基线点等多维度、全面整合打造统一监控平台。

实时监控

从应用角度将作业按照业务条线、数仓分层、数仓主题、组织结构和人员等维度划分，结合作业基线信息，实时监控正在运行的作业质量，并与作业基线形成对比参照，预警不符合标准的指标信息，第一时间通知责任人。实时作业运行与基线对比监控效果如图 12 所示：

图 12 实时作业运行监控图

推荐信息

系统通过规则引擎的设置和自动调度的执行，从存储资源配置、数据模型优化、作业优化、日志错误超时、预警通知等方面考虑，以制定的质量标准为评估依据，自动检测评估，汇总问题，形成可靠的推荐优化内容，并在达到阈值条件后主动推送消息，触发后续任务开展。

公共账号

通过“数据治理公共账号”机器人发送消息模式，将预判触发的预警通知、任务分配、任务提醒和风险评估等信息第一时间通知相应的负责人员，开启工作流程。

故障处理

支持自动提报和人工填报两种模式，以闭环工作流方式开展工作，确保问题故障可跟踪、可查询、可定级、可考核、可量化，以责任到人、落地可行的处理模式，严控数据质量，从根本上提高数据质量，提升业务服务水平。

DataMan 质量监管系统的投入运营，优化数据存储资源、提高作业性能、降低任务耗时、推进了管理工作的规范化和精细化。信息推荐功能以推送通知的形式将待优化、存风险和超时故障信息第一时间发送个人工作台，以工作流机制推动开展；模型监控、作业监控功能在数据存储、模型建设、作业耗时等场景合理的控制资源，节省了投资成本。 问题提报和故障管理功能的有效结合，将问题发现、提报、任务分配、处理完成及 Review 总结沉淀等形成了责任到人、问题可询的闭环流程。随着系统的深入运行，将在实时数据监控、质量故障统计管理、

数据质量考核机制、数据资产质量权威报告、知识库体系标准化及流程深化管理等功能方面持续推进和发挥价值。

总结

数据质量是数据治理建设的重要一环，与元数据管理、数据标准化及数据服务管理等共同构建了数据治理的体系框架。建设一个完整 DataMan 质量监管平台，将从监控、标准、流程制度等方面提升信息管理能力，优先解决所面临的数据质量和数据服务问题，其效果体现以下几个方面：

- 监控数据资产质量状态，为优化数据平台和数据仓库性能、合理配置数据存储资源提供决策支持；
- 持续推动数据质量监控优化预警、实时监控的机制；
- 重点优先监控关键核心数据资产，管控优化 20%核心资源，可提升 80%需求应用性能；
- 规范了问题故障的跟踪、Review、优化方案。从数据中提炼价值，从方案中形成标准化的知识体系；
- 由技术检测到业务监督，形成闭环工作流机制，提高整体数据质量，全面提升服务业务水平。

数据质量是数据仓库建设、数据应用建设和决策支持的关键因素，可通过完善组织架构和管理流程，加强部门间衔接和协调，严格按照标准或考核指标执行落地，确保数据质量方能将数据的商业价值最大化，进而提升企业的核心竞争力和保持企业的可持续发展。

数据同步

十三、美团 DB 数据同步到数据仓库的架构与实践

背景

在数据仓库建模中，未经任何加工处理的原始业务层数据，我们称之为 ODS(Operational Data Store)数据。在互联网企业中，常见的 ODS 数据有业务日志数据（Log）和业务 DB 数据（DB）两类。对于业务 DB 数据来说，从 MySQL 等关系型数据库的业务数据进行采集，然后导入到 Hive 中，是进行数据仓库生产的重要环节。

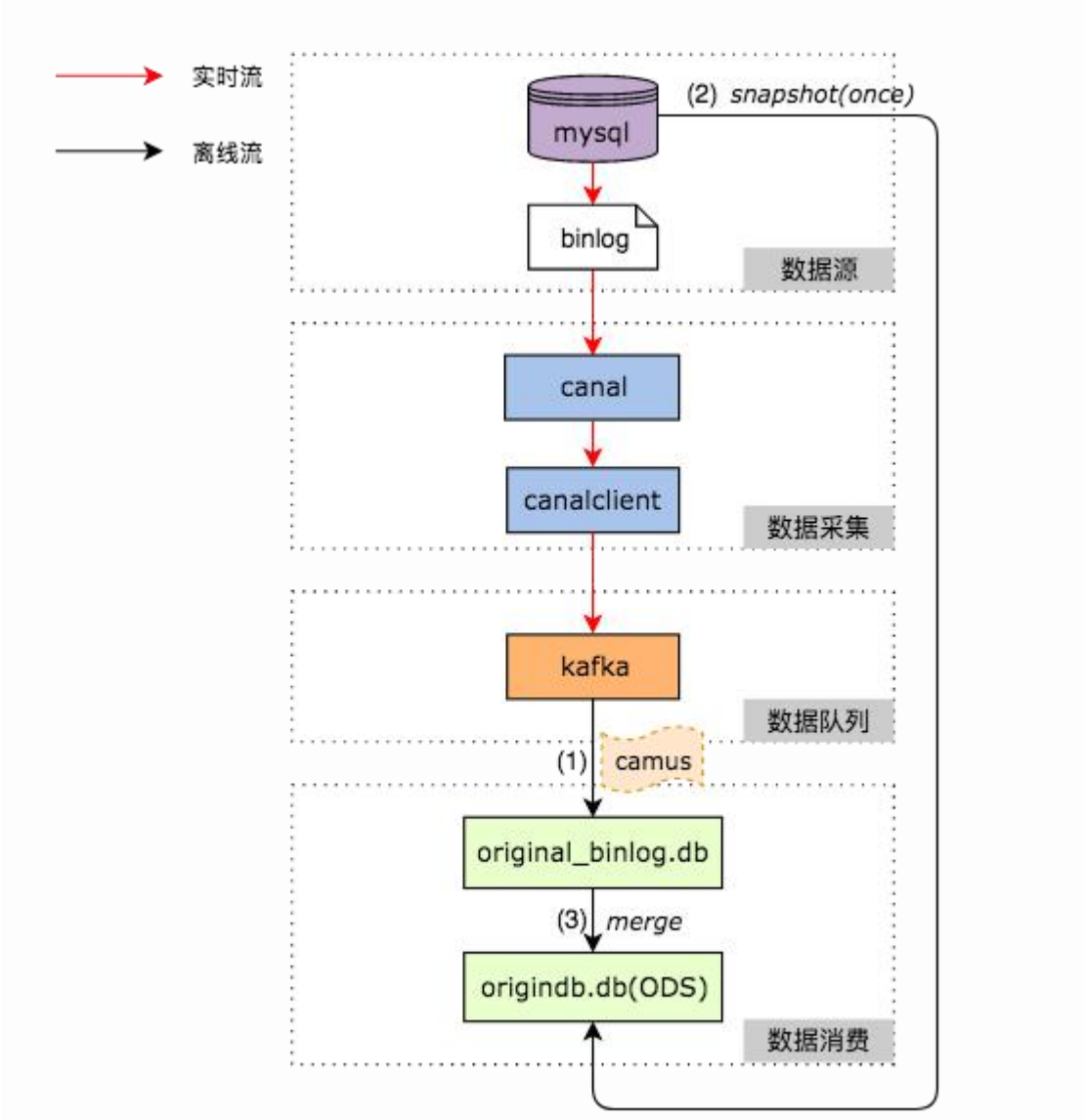
如何准确、高效地把 MySQL 数据同步到 Hive 中？一般常用的解决方案是批量取数并 Load：直连 MySQL 去 Select 表中的数据，然后存到本地文件作为中间存储，最后把文件 Load 到 Hive 表中。这种方案的优点是实现简单，但是随着业务的发展，缺点也逐渐暴露出来：

- 性能瓶颈：随着业务规模的增长，Select From MySQL -> Save to Localfile -> Load to Hive 这种数据流花费的时间越来越长，无法满足下游数仓生产的时间要求。
- 直接从 MySQL 中 Select 大量数据，对 MySQL 的影响非常大，容易造成慢查询，影响业务线上的正常服务。
- 由于 Hive 本身的语法不支持更新、删除等 SQL 原语，对于 MySQL 中发生 Update/Delete 的数据无法很好地进行支持。

为了彻底解决这些问题，我们逐步转向 CDC (Change Data Capture) + Merge 的技术方案，即实时 Binlog 采集 + 离线处理 Binlog 还原业务数据这样一套解决方案。Binlog 是 MySQL 的二进制日志，记录了 MySQL 中发生的所有数据变更，MySQL 集群自身的主从同步就是基于 Binlog 做的。

本文主要从 Binlog 实时采集和离线处理 Binlog 还原业务数据两个方面，来介绍如何实现 DB 数据准确、高效地进入数仓。

整体架构



整体的架构如上图所示。在 Binlog 实时采集方面，我们采用了阿里巴巴的开源项目 Canal，负责从 MySQL 实时拉取 Binlog 并完成适当解析。Binlog 采集后会暂存到 Kafka 上供下游消费。整体实时采集部分如图中红色箭头所示。

离线处理 Binlog 的部分，如图中黑色箭头所示，通过下面的步骤在 Hive 上还原一张 MySQL 表：

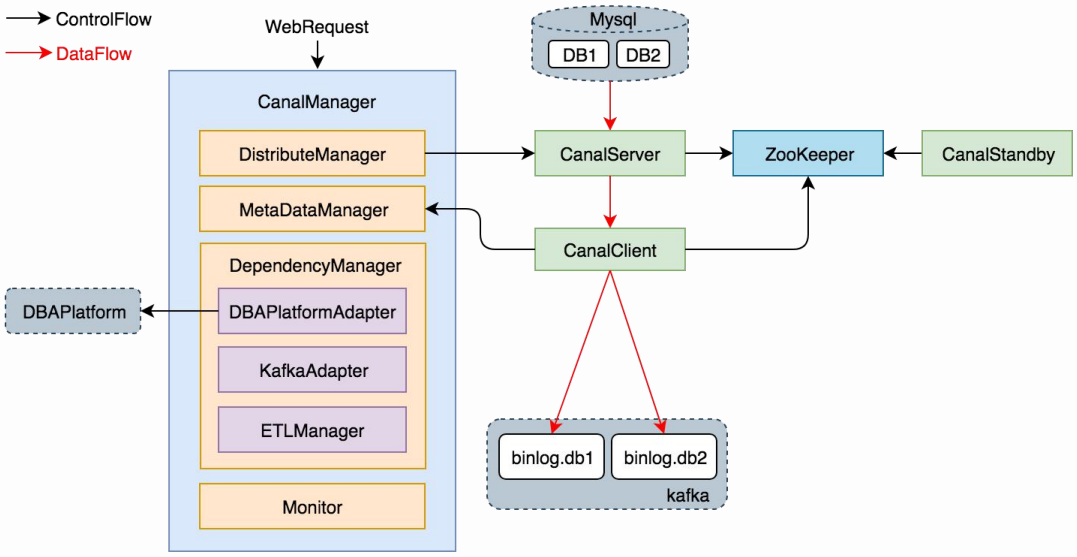
1. 采用 Linkedin 的开源项目 Camus，负责每小时把 Kafka 上的 Binlog 数据拉取到 Hive 上。
2. 对每张 ODS 表，首先需要一次性制作快照（Snapshot），把 MySQL 里的存量数据读取到 Hive 上，这一过程底层采用直连 MySQL 去 Select 数据的方式。
3. 对每张 ODS 表，每天基于存量数据和当天增量产生的 Binlog 做 Merge，从而还原出业务数据。

我们回过头来看看，背景中介绍的批量取数并 Load 方案遇到的各种问题，为什么用这种方案能解决上面的问题呢？

- 首先，Binlog 是流式产生的，通过对 Binlog 的实时采集，把部分数据处理需求由每天一次的批处理分摊到实时流上。无论从性能上还是对 MySQL 的访问压力上，都会有明显地改善。
- 第二，Binlog 本身记录了数据变更的类型（Insert/Update/Delete），通过一些语义方面的处理，完全能够做到精准的数据还原。

Binlog 实时采集

对 Binlog 的实时采集包含两个主要模块：一是 CanalManager，主要负责采集任务的分配、监控报警、元数据管理以及和外部依赖系统的对接；二是真正执行采集任务的 Canal 和 CanalClient。



当用户提交某个 DB 的 Binlog 采集请求时，CanalManager 首先会调用 DBA 平台的相关接口，获取这一 DB 所在 MySQL 实例的相关信息，目的是从中选出最适合 Binlog 采集的机器。然后把采集实例（Canal Instance）分发到合适的 Canal 服务器上，即 CanalServer 上。在选择具体的 CanalServer 时，CanalManager 会考虑负载均衡、跨机房传输等因素，优先选择负载较低且同地域传输的机器。

CanalServer 收到采集请求后，会在 ZooKeeper 上对收集信息进行注册。注册的内容包括：

- 以 Instance 名称命名的永久节点。
- 在该永久节点下注册以自身 ip:port 命名的临时节点。

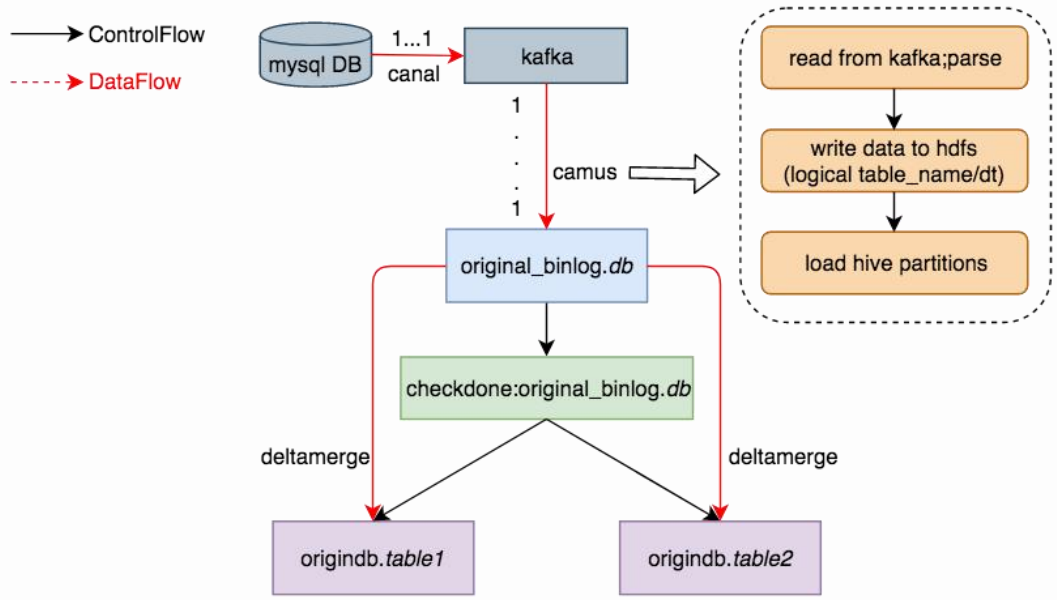
这样做的目的有两个：

- 高可用：CanalManager 对 Instance 进行分发时，会选择两台 CanalServer，一台是 Running 节点，另一台作为 Standby 节点。Standby 节点会对该 Instance 进行监听，当 Running 节点出现故障后，临时节点消失，然后 Standby 节点进行抢占。这样就达到了容灾的目的。
- 与 CanalClient 交互：CanalClient 检测到自己负责的 Instance 所在的 Running CanalServer 后，便会进行连接，从而接收到 CanalServer 发来的 Binlog 数据。

对 Binlog 的订阅以 MySQL 的 DB 为粒度，一个 DB 的 Binlog 对应了一个 Kafka Topic。底层实现时，一个 MySQL 实例下所有订阅的 DB，都由同一个 Canal Instance 进行处理。这是因为 Binlog 的产生是以 MySQL 实例为粒度的。CanalServer 会抛弃掉未订阅的 Binlog 数据，然后 CanalClient 将接收到的 Binlog 按 DB 粒度分发到 Kafka 上。

离线还原 MySQL 数据

完成 Binlog 采集后，下一步就是利用 Binlog 来还原业务数据。首先要解决的第一个问题是把 Binlog 从 Kafka 同步到 Hive 上。



Kafka2Hive

整个 Kafka2Hive 任务的管理，在美团数据平台的 ETL 框架下进行，包括任务原语的表达和调度机制等，都同其他 ETL 类似。而底层采用 LinkedIn 的开源项目 Camus，并进行了有针对性的二次开发，来完成真正的 Kafka2Hive 数据传输工作。

对 Camus 的二次开发

Kafka 上存储的 Binlog 未带 Schema，而 Hive 表必须有 Schema，并且其分区、字段等的设计，都要便于下游的高效消费。对 Camus 做的第一个改造，便是将 Kafka 上的 Binlog 解析成符合目标 Schema 的格式。

对 Camus 做的第二个改造，由美团的 ETL 框架所决定。在我们的任务调度系统中，目前只对同调度队列的任务做上下游依赖关系的解析，跨调度队列是不能建立依赖关系的。而在 MySQL2Hive 的整个流程中，Kafka2Hive 的任务需要每小时执行一次（小时队列），Merge 任务每天执行一次（天队列）。而 Merge 任务的启动必须要严格依赖小时 Kafka2Hive 任务的完成。

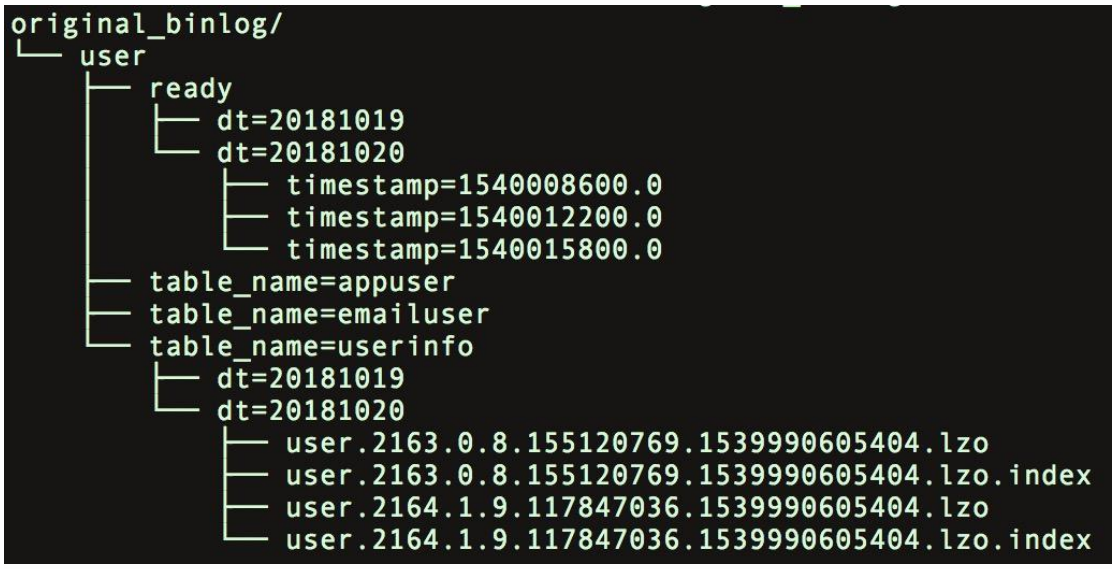
为了解决这一问题，我们引入了 Checkdone 任务。Checkdone 任务是天任务，主要负责检测前一天的 Kafka2Hive 是否成功完成。如果成功完成了，则 Checkdone 任务执行成功，这样下游的 Merge 任务就可以正确启动了。

Checkdone 的检测逻辑

Checkdone 是怎样检测的呢？每个 Kafka2Hive 任务成功完成数据传输后，由 Camus 负责在相应的 HDFS 目录下记录该任务的启动时间。Checkdone 会扫描前一天的所有时间戳，如果最大的时间戳已经超过了 0 点，就说明前一天的 Kafka2Hive 任务都成功完成了，这样 Checkdone 就完成了检测。

此外，由于 Camus 本身只是完成了读 Kafka 然后写 HDFS 文件的过程，还必须完成对 Hive 分区的加载才能使下游查询到。因此，整个 Kafka2Hive 任务的最后一步是加载 Hive 分区。这样，整个任务才算成功执行。

每个 Kafka2Hive 任务负责读取一个特定的 Topic，把 Binlog 数据写入 original_binlog 库下的一张表中，即前面图中的 original_binlog.*db*，其中存储的是对应到一个 MySQL DB 的全部 Binlog。



上图说明了一个 Kafka2Hive 完成后，文件在 HDFS 上的目录结构。假如一个 MySQL DB 叫做 user，对应的 Binlog 存储在 original_binlog.user 表中。ready 目录中，按天存储了当天所有成功执行的 Kafka2Hive 任务的启动时间，供 Checkdone 使用。每张表的 Binlog，被组织到一个分区中，例如 userinfo 表的 Binlog，存储在 table_name=userinfo 这一分区中。每个 table_name 一级分区下，按 dt 组织二级分区。图中的 xxx.lzo 和 xxx.lzo.index 文件，存储的是经过 lzo 压缩的 Binlog 数据。

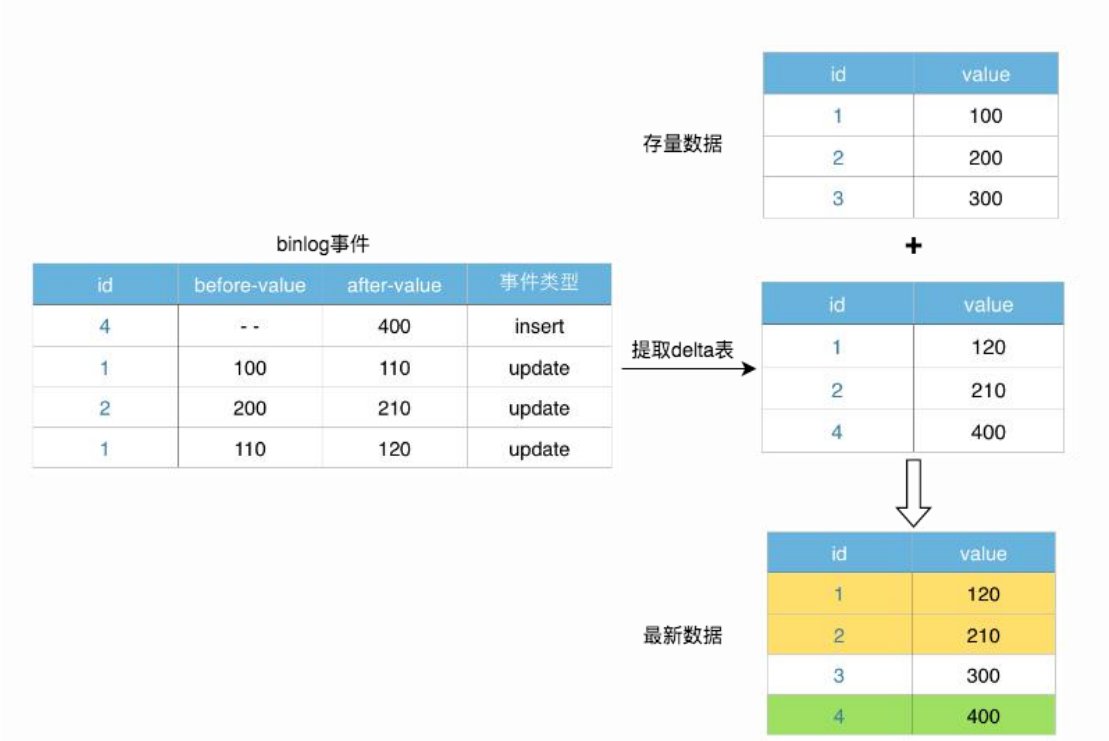
Merge

Binlog 成功入仓后，下一步要做的就是基于 Binlog 对 MySQL 数据进行还原。Merge 流程做了两件事，首先把当天生成的 Binlog 数据存放到 Delta 表中，然后和已有的存量数据做一个基于主键的 Merge。Delta 表中的数据是当天的最新数据，当一条数据在一天内发生多次变更时，Delta 表中只存储最后一次变更后的数据。

把 Delta 数据和存量数据进行 Merge 的过程中，需要有唯一键来判定是否是同一条数据。如果同一条数据既出现在存量表中，又出现在 Delta 表中，说明这一条数据发生了更新，则选取 Delta 表的数据作为最终结果；否则说明没有发生任何变动，保留原来存量表中的数据作为最终结果。Merge 的结果数据会 Insert Overwrite 到原表中，即图中的 origindb.*table*。

Merge 流程举例

下面用一个例子来具体说明 Merge 的流程。



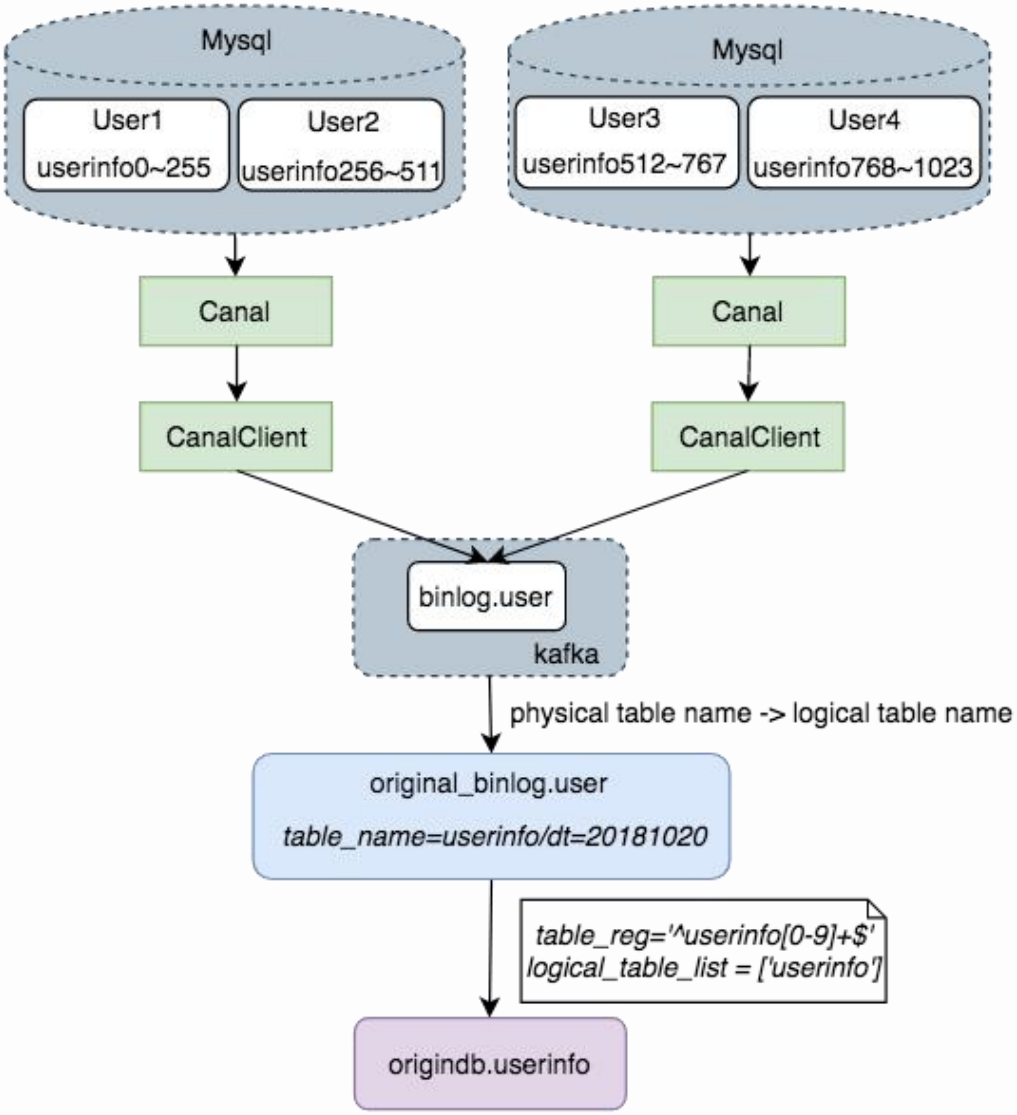
数据表共 id、value 两列，其中 id 是主键。在提取 Delta 数据时，对同一条数据的多次更新，只选择最后更新的一条。所以对 id=1 的数据，Delta 表中记录最后一条更新后的值 value=120。Delta 数据和存量数据做 Merge 后，最终结果中，新插入一条数据（id=4），两条数据发生了更新（id=1 和 id=2），一条数据未变（id=3）。

默认情况下，我们采用 MySQL 表的主键作为这一判重的唯一键，业务也可以根据实际情况配置不同于 MySQL 的唯一键。

上面介绍了基于 Binlog 的数据采集和 ODS 数据还原的整体架构。下面主要从两个方面介绍我们解决的实际业务问题。

实践一：分库分表的支持

随着业务规模的扩大，MySQL 的分库分表情况越来越多，很多业务的分表数目都在几千个这样的量级。而一般数据开发同学需要把这些数据聚合到一起进行分析。如果对每个分表都进行手动同步，再在 Hive 上进行聚合，这个成本很难被我们接受。因此，我们需要在 ODS 层就完成分表的聚合。



首先，在 Binlog 实时采集时，我们支持把不同 DB 的 Binlog 写入到同一个 Kafka Topic。用户可以在申请 Binlog 采集时，同时勾选同一个业务逻辑下的多个物理 DB。通过在 Binlog 采集层的汇集，所有分库的 Binlog 会写入到同一张 Hive 表中，这样下游在进行 Merge 时，依然只需要读取一张 Hive 表。

第二，Merge 任务的配置支持正则匹配。通过配置符合业务分表命名规则的正则表达式，Merge 任务就能了解自己需要聚合哪些 MySQL 表的 Binlog，从而选取相应分区的数据来执行。

这样通过两个层面的工作，就完成了分库分表在 ODS 层的合并。

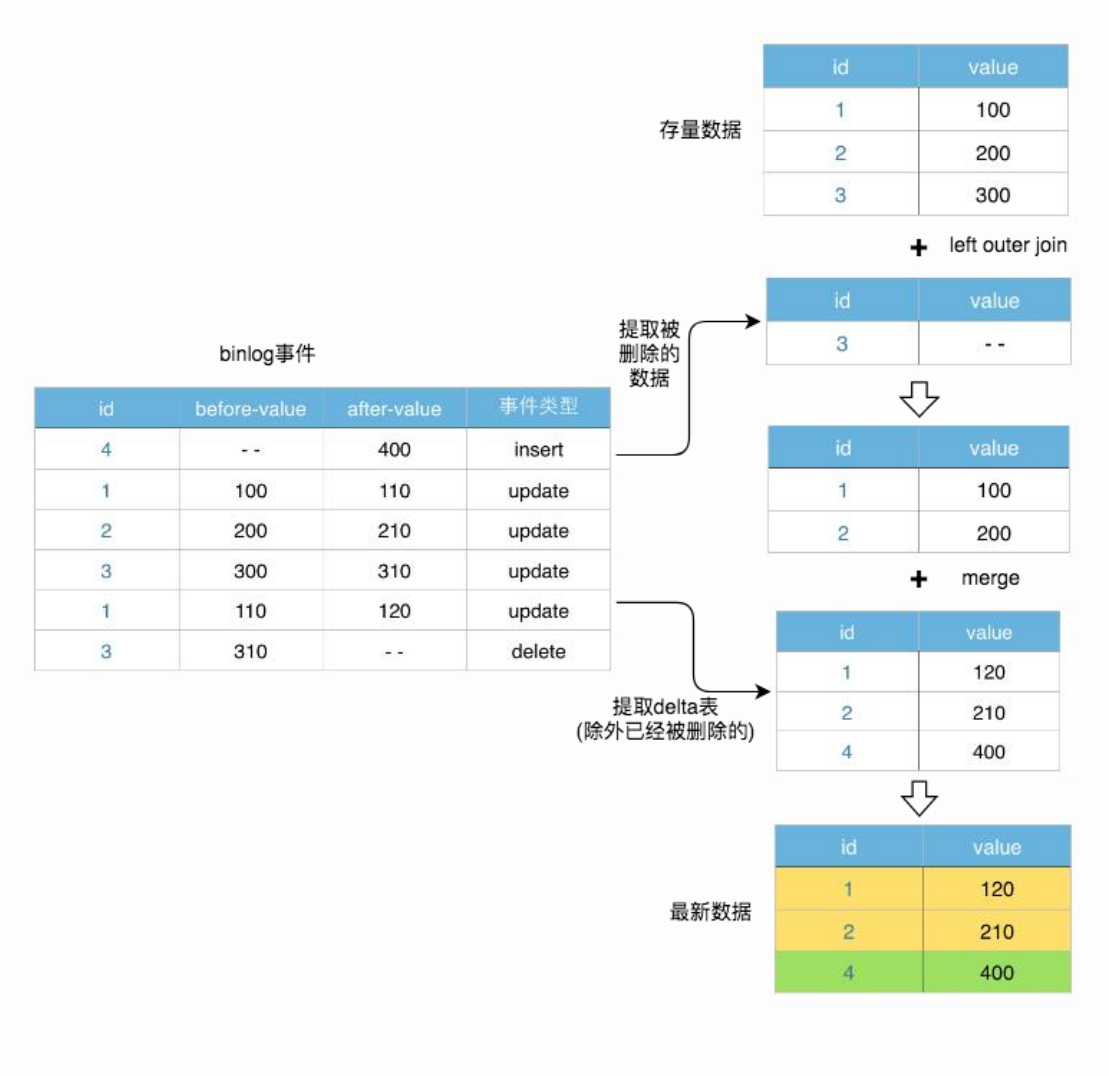
这里面有一个技术上的优化，在进行 Kafka2Hive 时，我们按业务分表规则对表名进行了处理，把物理表名转换成了逻辑表名。例如 userinfo123 这张表名会被转换为 userinfo，其 Binlog 数据存储在 original_binlog.user 表的 table_name=userinfo 分区中。这样做的目的是防止过多的 HDFS 小文件和 Hive 分区造成的底层压力。

实践二：删除事件的支持

Delete 操作在 MySQL 中非常常见，由于 Hive 不支持 Delete，如果想把 MySQL 中删除的数据在 Hive 中删掉，需要采用“迂回”的方式进行。

对需要处理 Delete 事件的 Merge 流程，采用如下两个步骤：

- 首先，提取出发生了 Delete 事件的数据，由于 Binlog 本身记录了事件类型，这一步很容易做到。将存量数据（表 A）与被删掉的数据（表 B）在主键上做左外连接(Left outer join)，如果能够全部 join 到双方的数据，说明该条数据被删掉了。因此，选择结果中表 B 对应的记录为 NULL 的数据，即是应当被保留的数据。
- 然后，对上面得到的被保留下来的数据，按照前面描述的流程做常规的 Merge。



总结与展望

作为数据仓库生产的基础，美团数据平台提供的基于 Binlog 的 MySQL2Hive 服务，基本覆盖了美团内部的各个业务线，目前已经能够满足绝大部分业务的数据同步需求，实现 DB 数据准确、高效地入仓。在后面的发展中，我们会集中解决 CanalManager 的单点问题，并构建跨机房容灾的架构，从而更加稳定地支撑业务的发展。

本文主要从 Binlog 流式采集和基于 Binlog 的 ODS 数据还原两方面，介绍了这一服务的架构，并介绍了我们在实践中遇到的一些典型问题和解决方案。希望能够给其他开发者一些参考价值，同时也欢迎大家和我们一起交流。

Doris 和 Kylin 实践

十四、Apache Doris 在美团外卖数仓中的应用实践

序言

美团外卖数据仓库技术团队负责支撑日常业务运营及分析师的日常分析，由于外卖业务特点带来的数据生产成本较高和查询效率偏低的问题，他们通过引入 Apache Doris 引擎优化生产方案，实现了低成本生产与高效查询的平衡。并以此分析不同业务场景下，基于 Kylin 的 MOLAP 模式与基于 Doris 引擎的 ROLAP 模式的适用性问题。希望能对大家有所启发或者帮助。

本文侧重于以 Doris 引擎为“发动机”的数仓生产架构的改进与思考。在开源的大环境下，各种数据引擎百花齐放，但由于业务的复杂性与多样性，目前并没有哪个引擎能够适配所有业务场景，因此希望通过我们的业务实践与思考为大家提供一些经验参考。美团外卖数仓技术团队致力于将数据应用效率最大化，同时兼顾研发、生产与运维成本的最小化，建设持续进步的数仓能力，也欢迎大家多给我们提出建议。

数仓交互层引擎的应用现状

目前，互联网业务规模变得越来越大，不论是业务生产系统还是日志系统，基本上都是基于 Hadoop/Spark 分布式大数据技术生态来构建数据仓库，然后对数据进行适当的分层、加工、管理。而在数据应用交互层面，由于时效性的要求，数据最终的展现查询还是需要通过 DBMS（MySQL）、MOLAP（Kylin）引擎来进行支撑。如下图所示：

汇总数据的交互

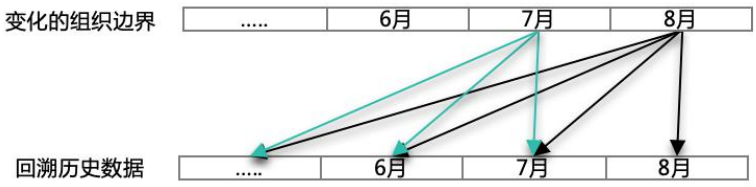
业务团队日常经营分析最典型的场景就是各种维度下的自定义查询，面对如此灵活可变、所见即所得的应用场景，美团平台使用 Kylin 作为公司的主要 MOLAP 引擎。MOLAP 是预计算生产，在增量业务，预设维度分析场景下表现良好，但在变化维的场景下生产成本巨大。例如，如果使用最新商家类型回溯商家近三个月的表现，需要重新计算三个月的 Cube，需花费几个小时，来计算近 TB 的历史数据。另外，应对非预设维度分析，MOLAP 模型需要重新进行适配计算，也需要一定的迭代工作。

明细数据的交互

业务分析除了宏观数据之外，对明细数据查询也是一种刚需。通常大家会选择 MySQL 等关系型 DB 作为明细数据的快速检索查询，但当业务成长较快时，很快就会遇到性能瓶颈，并且运维成本也很高。例如，大数据量的同步、新增字段、历史数据更新等操作，它们的维护成本都非常高。

外卖运营业务特点

美团的使命是“帮大家吃得更好，生活更好”。外卖业务为大家提供送餐服务，连接商家与用户，这是一个劳动密集型的业务，外卖业务有上万人的运营团队来服务全国几百万的商家，并以“商圈”为单元，服务于“商圈”内的商家。“商圈”是一个组织机构维度中的最小层级，源于外卖组织的特点，“商圈”及其上层组织机构是一个变化维度，当“商圈”边界发生变化时，就导致在往常日增量的业务生产方式中，历史数据的回溯失去了参考意义。在所有展现组织机构数据的业务场景中，组织机构的变化是一个绕不开的技术问题。此外，商家品类、类型等其它维度也存在变化维的问题。如下图所示：



数据生产面临的挑战

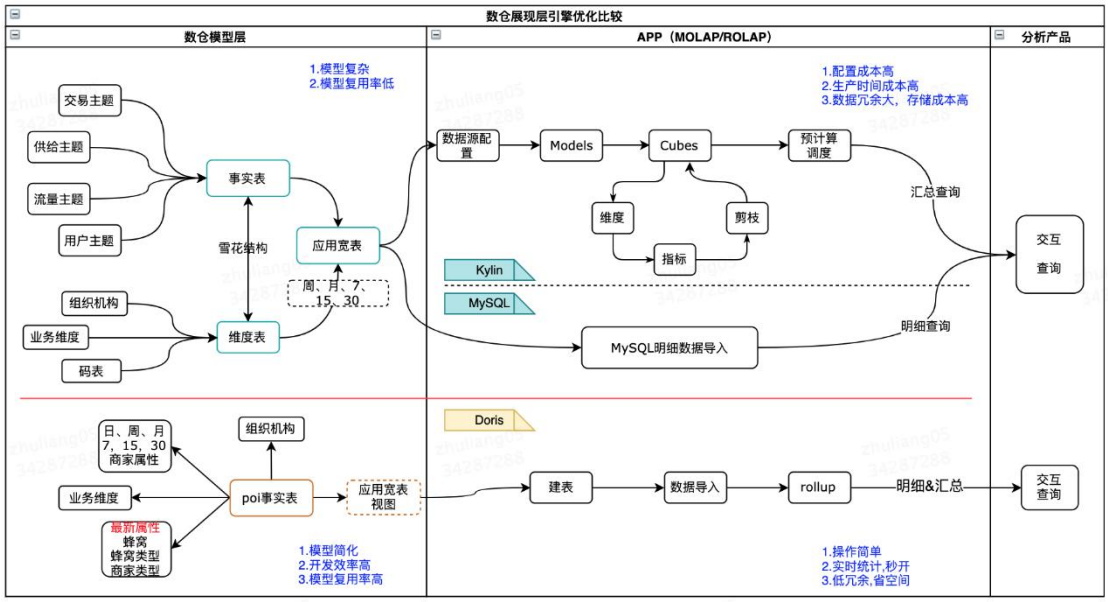
数据爆炸，每日使用最新维度对历史数据进行回溯计算。在 Kylin 的 MOLAP 模式下存在如下问题：

- 历史数据每日刷新，失去了增量的意义。
- 每日回溯历史数据量大，10 亿+的历史数据回溯。
- 数据计算耗时 3 小时+，存储 1TB+，消耗大量计算存储资源，同时严重影响 SLA 的稳定性。
- 预计算的大量历史数据实际使用率低下，实际工作中对历史的回溯 80%集中在近 1 个月左右，但为了应对所有需求场景，业务要求计算近半年以上的历史。
- 不支持明细数据的查询。

解决方案：引入 MPP 引擎，数据现用现算

既然变化维的历史数据预计算成本巨大，最好的办法就是现用现算，但现用现算需要强大的并行计算能力。OLAP 的实现有 MOLAP、ROLAP、HOLAP 三种形式，MOLAP 以 Cube 为表现形式，但计算与管理成本较高。ROLAP 需要强大的关系型 DB 引擎支撑。长期以来，由于传统关系型 DBMS 的数据处理能力有限，所以 ROLAP 模式受到很大的局限性。随着分布式、并行化技术成熟应用，MPP 引擎逐渐表现出强大的高吞吐、低时延计算能力，号称“亿级秒开”的引擎不在少数，ROLAP 模式可以得到更好的延伸。单从业务实际应用考虑，性能在千万量级关联查询现场计算秒开的情况下，已经可以覆盖到很多应用场景，具备应用的可能性。例如：日数据量的 ROLAP 现场计算，周、月趋势的计算，以及明细数据的浏览都可以较好的应对。

下图是 MOLAP 模式与 ROLAP 模式下应用方案的比较：



MOLAP 模式的劣势

1. 应用层模型复杂，根据业务需要以及 Kylin 生产需要，还要做较多模型预处理。这样在不同的业务场景中，模型的利用率也比较低。
2. Kylin 配置过程繁琐，需要配置模型设计，并配合适当的“剪枝”策略，以实现计算成本与查询效率的平衡。
3. 由于 MOLAP 不支持明细数据的查询，在“汇总+明细”的应用场景中，明细数据需要同步到 DBMS 引擎来响应交互，增加了生产的运维成本。
4. 较多的预处理伴随着较高的生产成本。

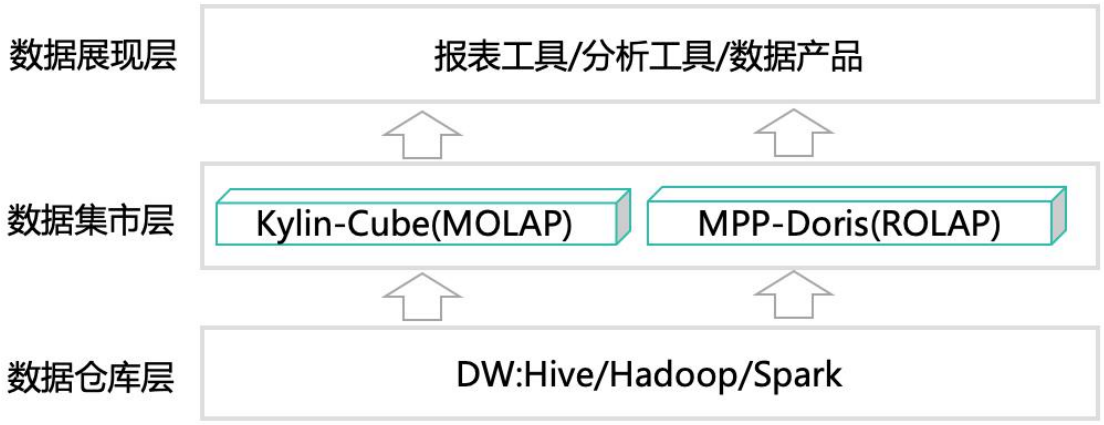
ROLAP 模式的优势

1. 应用层模型设计简化，将数据固定在一个稳定的数据粒度即可。比如商家粒度的星形模型，同时复用率也比较高。
2. App 层的业务表达可以通过视图进行封装，减少了数据冗余，同时提高了应用的灵活性，降低了运维成本。
3. 同时支持“汇总+明细”。
4. 模型轻量标准化，极大的降低了生产成本。

综上所述，在变化维、非预设维、细粒度统计的应用场景下，使用 MPP 引擎驱动的 ROLAP 模式，可以简化模型设计，减少预计算的代价，并通过强大的实时计算能力，可以支撑良好的实时交互体验。

双引擎下的应用场景适配问题

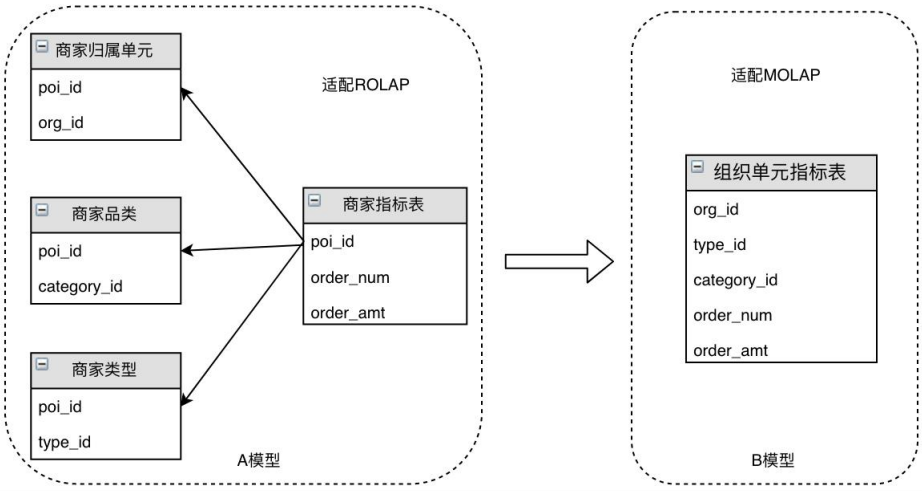
架构上通过 MOLAP+ROLAP 双引擎模式来适配不同应用场景，如下图所示：



技术权衡

MOLAP：通过预计算，提供稳定的切片数据，实现多次查询一次计算，减轻了查询时的计算压力，保证了查询的稳定性，是“空间换时间”的最佳路径。实现了基于 Bitmap 的去重算法，支持在不同维度下去重指标的实时统计，效率较高。 **ROLAP**：基于实时的大规模并行计算，对集群的要求较高。MPP 引擎的核心是通过将数据分散，以实现 CPU、IO、内存资源的分布，来提升并行计算能力。在当前数据存储以磁盘为主的情况下，数据 Scan 需要的较大的磁盘 IO，以及并行导致的高 CPU，仍然是资源的短板。因此，高频的大规模汇总统计，并发能力将面临较大挑战，这取决于集群硬件方面的并行计算能力。传统去重算法需要大量计算资源，实时的大规模去重指标对 CPU、内存都是一个巨大挑战。目前 Doris 最新版本已经支持 Bitmap 算法，配合预计算可以很好地解决去重应用场景。

业务模型适配



MOLAP： 当业务分析维度相对固化，并在可以使用历史状态时，按照时间进行增量生产，加工成本呈线性增长状态，数据加工到更粗的粒度（如组织单元），减少结果数据量，提高交互效率。如上图所示，由 A 模型预计算到 B 模型，使用 Kylin 是一个不错的选择。

ROLAP： 当业务分析维度灵活多变或者特定到最新的状态时（如上图 A 模型中，始终使用最新的商家组织归属查看历史），预计算回溯历史数据成本巨大。在这种场景下，将数据稳定在商家的粒度，通过现场计算进行历史数据的回溯分析，实现现用现算，可以节省掉预计算的巨大成本，并带来较大的应用灵活性。这种情况下适合 MPP 引擎支撑下的 ROLAP 生产模式。

MPP 引擎的选型

目前开源的比较受关注的 OLAP 引擎很多，比如 Greenplum、Apache Impala、Presto、Doris、ClickHouse、Druid、TiDB 等等，但缺乏实践案例的介绍，所以我们也没有太多的经验可以借鉴。于是，我们就结合自身业务的需求，从引擎

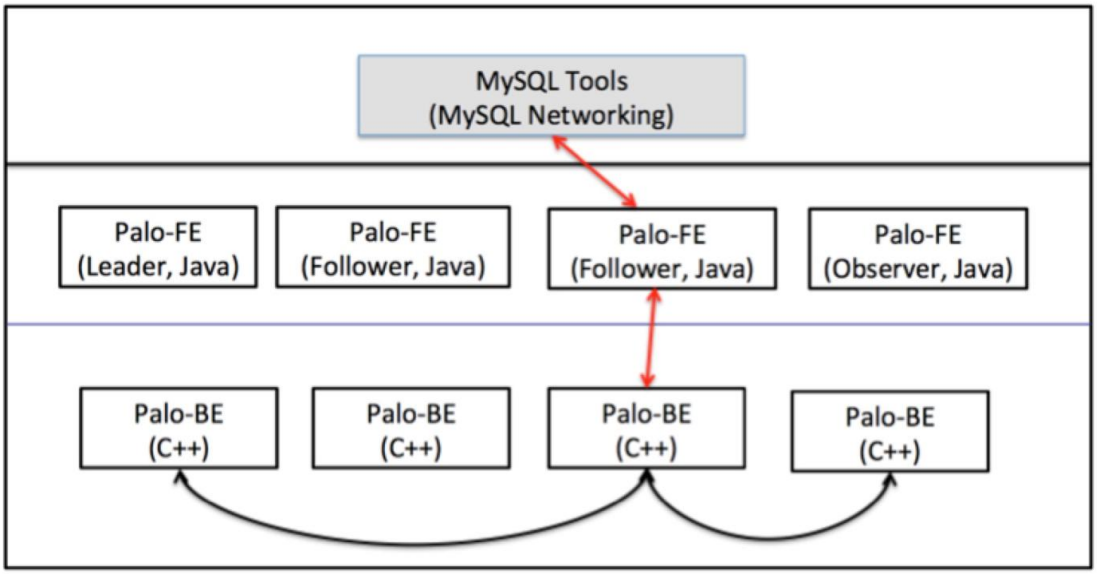
建设成本出发，并立足于公司技术生态融合、集成、易用性等维度进行综合考虑，作为选型依据，最终我们平台部门选择了 2018 年刚进入 Apache 社区的 Doris。

开源OLAP引擎	优点	缺点	技术融合成本	易用性	性能	运维成本
ClickHouse	• 列存储 • 单机性能剽悍 • 保留明细 • 向量化引擎	• 分布集群在线扩展支持不佳 • 运维成本极高	高	非标协议接口	满足	高
Druid	• 实时数据摄入 • 列式存储和位图索引 • 多租户和高并发	• OLAP性能分场景表现差异大 • 使用门槛高	高	非标协议接口	分场景	高
Doris	• Google Mesa +Apache Impala +ORCFile / Parquet • 主键更新 • 支持Rollup Table • 高并发和高吞吐的Ad-hoc查询	• 成熟度不足 • 应用不广泛	低	兼容MySQL 访问协议	满足	低
TIDB	• 100%OLTP+80%OLAP • 同时明细和聚合查询 • 支持更新	• 非列存储 • OLAP能力不足	低	SQL标准	不满足	低

Doris 简介及特点

Doris 是基于 MPP 架构的 OLAP 引擎，主要整合了 Google Mesa (数据模型)、Apache Impala（MPP Query Engine）和 Apache ORCFile （存储格式，编码和压缩）的技术。

Doris 的系统架构如下，主要分为 FE 和 BE 两个组件，FE 主要负责查询的解析、编译、优化、调度和元数据管理；BE 主要负责查询的执行和数据存储。关于 Doris 的更多技术细节，可参考其官方文档。

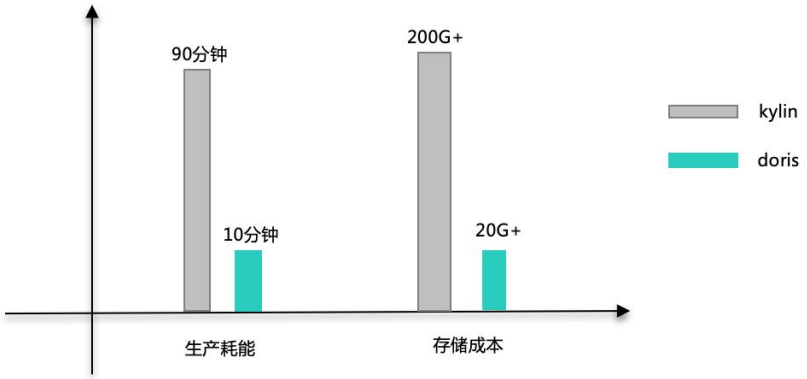


整体架构

Doris 的特点：

- 同时支持高并发点查询和高吞吐的 Ad-hoc 查询。
- 同时支持离线批量导入和实时数据导入。
- 同时支持明细和聚合查询。
- 兼容 MySQL 协议和标准 SQL。
- 支持 Rollup Table 和 Rollup Table 的智能查询路由。
- 支持较好的多表 Join 策略和灵活的表达式查询。
- 支持 Schema 在线变更。
- 支持 Range 和 Hash 二级分区。

Doris 在外卖数仓中的应用效率



上图是我们在一个分析项目改造中的评估项目收益，整体在查询效率不变的情况下，生产耗能及存储成本都有较大收益。

以 20 台 BE+3FE 的 Doris 环境，效率、性能表现情况如下：

- 支撑数据分析产品数十个以上，整体响应达到 ms 级。
- 支持百万、千万级大表关联查询，同时进行维表关联的雪花模型，经过 Colocate Join 特性优化，可以实现秒级响应。
- 日级别，基于商家明细现场计算，同时满足汇总及下钻明细查询，查询时效基本都可以控制在秒级。
- 7 日趋势分析，2~3 秒。由于数据量较大，根据集群规模不同查询性能有所区别，但数据量较大时，调动的集群资源较多，因此 MPP 的并发性能受限于集群的性能。一般原则是并发较高的业务，需要严格控制查询时效（基本在毫秒级），对于并发不高的业务，允许进行较大的查询，但也要考虑集群的承受能力。
- 通过一年来的应用以及 Doris 的不断改进升级，Doris 的高可靠、高可用、高可扩展性也得到进一步验证，服务稳定可靠。

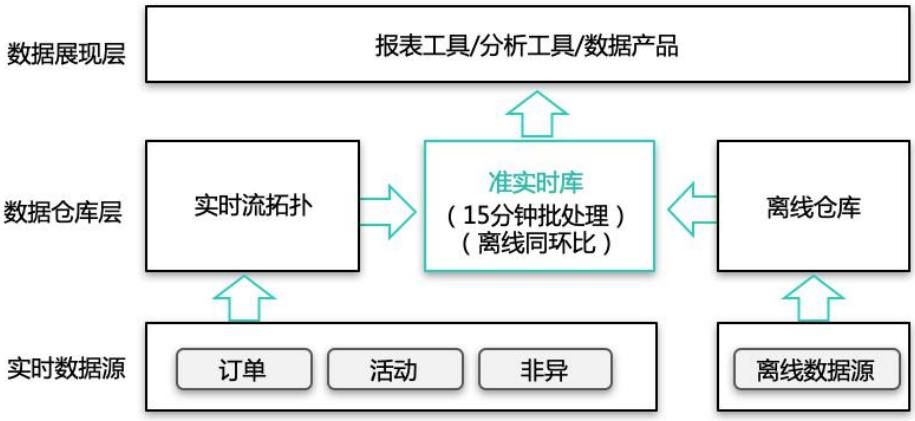
准实时场景下的应用

离线业务分析大多基于 T+1 的离线数据，但在营销活动场景下，外卖团队往往需要当日的实时数据进行业务变化的监控与分析，通常情况下会采用实时流计算来实现。

外卖实时业务监控有如下特点：

- 避免分钟级的生产波动影响，业务上 10、15 分钟准实时数据可以满足分析需要。
- 实时数据需要与离线数据进行日环比与周同比的比对。
- 订单业务需要事件时间，体验业务需要生产时间，业务对齐逻辑复杂。
- 不同业务线需求差异大，指标需要良好扩展性。

由于业务上的复杂性，实时流计算中，需要考虑诸多业务口径的对齐，业务 ER 模型在合流处理中开发成本较高，资源占用较大，通过设计基于 Doris 的准实时生产数仓，可以灵活地实现业务微批处理，且开发生产成本都比较低。以下为基于 Doris 的准实时数仓架构设计，是典型的实时 Lambda 生产架构：



实现准实时计算方案，需要以下能力的支撑：

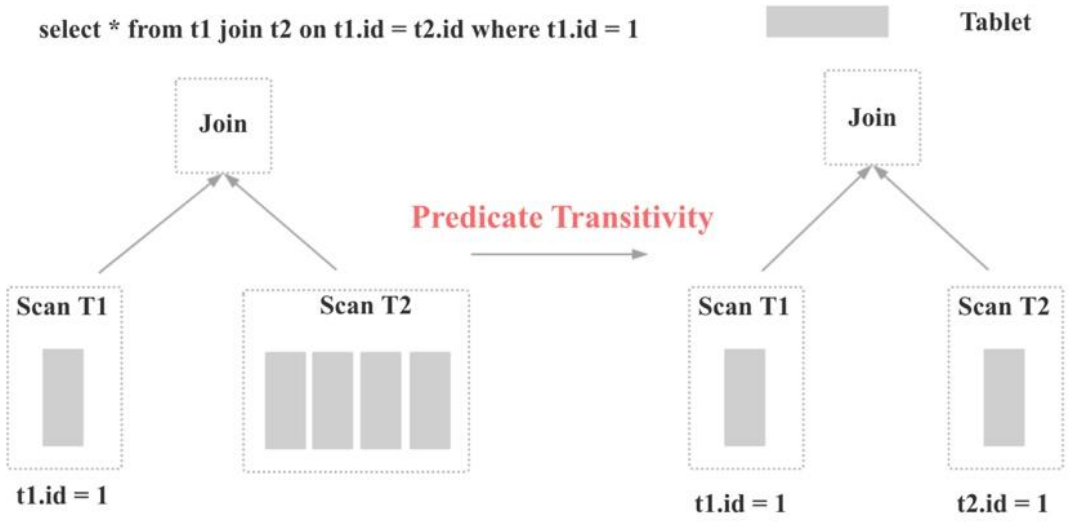
实时的写入能力：目前支持 Kafka To Doris 秒级延迟。在可靠性、稳定性建设方面仍需进一步提升。 **引擎建设**：短平快的计算+高效的存储性能。目前 Doris 引擎性能仍有进步空间，2020 年将有较大改进提升，随着后续 Page Cache，内存表等能力的上线，IO 将不再拖后腿，并发能力将有较大提升。 **可靠的调度能力**：提供 5、10、15、30 分钟的调度保障能力。 **Lambda 架构简化**：实时数据与离线数据更好的在 Doris 中进行融合，灵活支撑应用。 **高效的 OLAP 交互**：支撑业务的灵活查询访问，业务层通过视图进行逻辑封装直接复用汇总层多维模型，提高了开发效率，减少了运维成本。

相比 Storm、Flink 中的窗口计算，准实时 DB 微批的优势：

计算模型	概念	存储	计算	资源	查询
流式窗口	将流式数据抽象成表的逻辑概念，技术上需要大量的兼容改造。	根据窗口大小开设缓存空间，如果窗口较大，或者当日累计，缓存成本较高。	多流合并，处理复杂的关联对齐逻辑。	以业务最高峰时的需求量做资源储备，其他时间资源浪费较多。	结果转储到第三方查询引擎。
DB微批	原生	实时写入，缓存当日或近日所有数据。	原生Join操作	与OLAP分析共用资源，低频的计算对系统负担较小。	原生OLAP查询支持。

Doris 引擎在美团的重要改进

Join 谓词下推的传递性优化



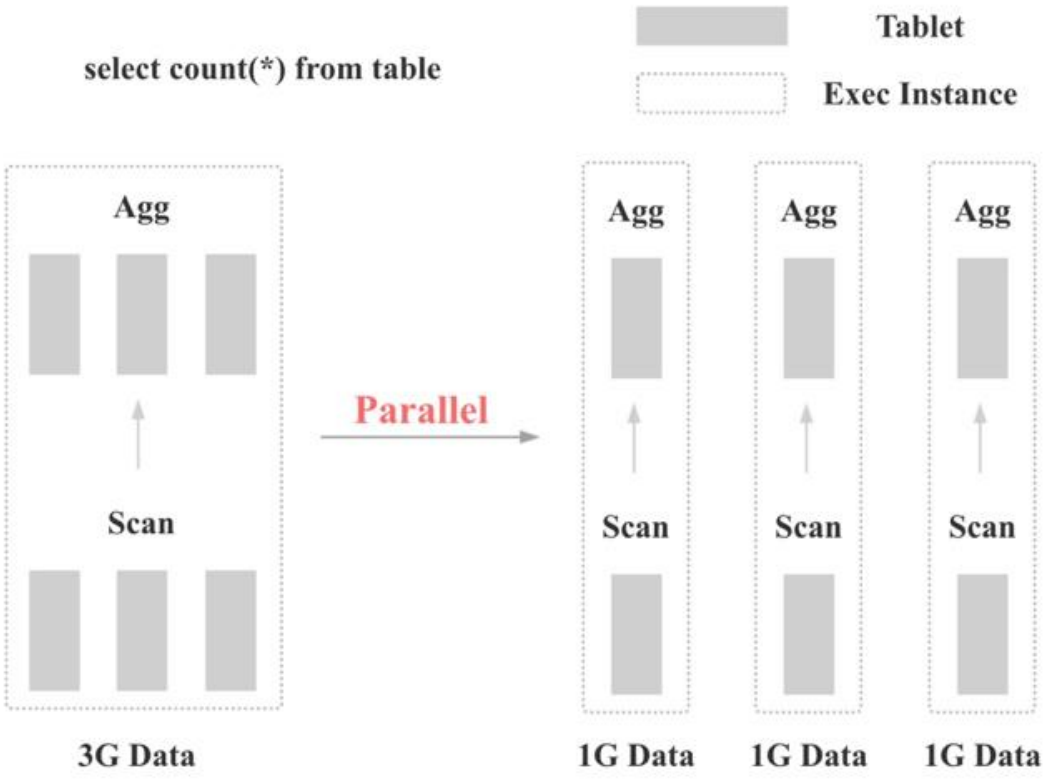
如上图所示，对于下面的 SQL：

```
select * from t1 join t2 on t1.id = t2.id where t1.id = 1
```

Doris 开源版本默认会对 t2 表进行全表 Scan，这样会导致上面的查询超时，进而导致外卖业务在 Doris 上的第一批应用无法上线。

于是我们在 Doris 中实现了第一个优化：Join 谓词下推的传递性优化（MySQL 和 TiDB 中称之为 Constant Propagation）。Join 谓词下推的传递性优化是指：基于谓词 t1.id = t2.id 和 t1.id = 1，我们可以推断出新的谓词 t2.id = 1，并将谓词 t2.id = 1 下推到 t2 的 Scan 节点。 这样假如 t2 表有数百个分区的话，查询性能就会有数十倍甚至上百倍的提升，因为 t2 表参与 Scan 和 Join 的数据量会显著减少。

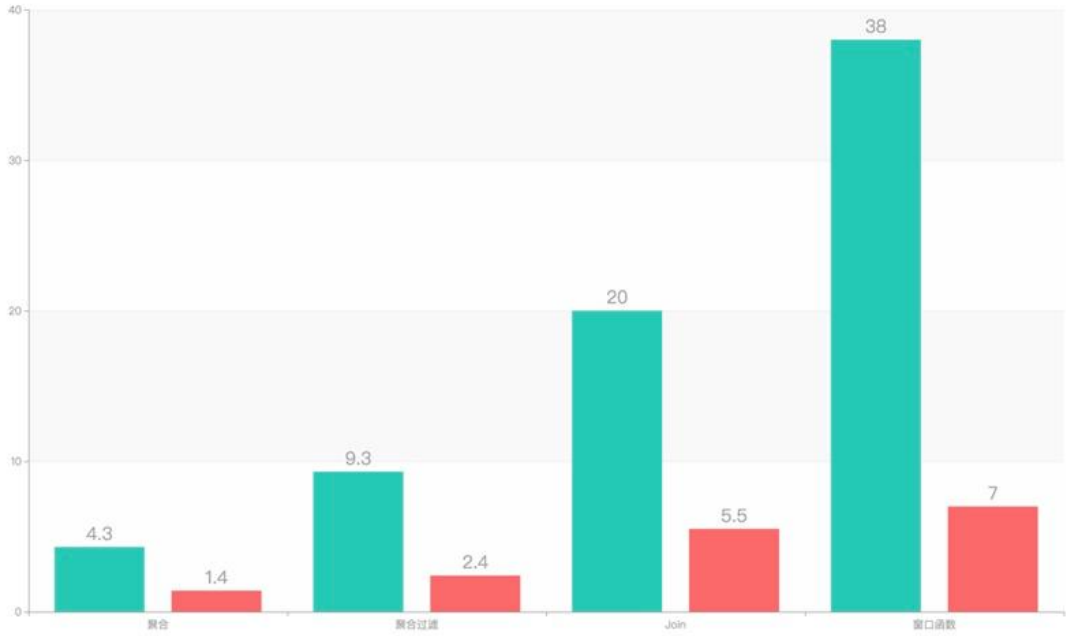
查询执行多实例并发优化



如上图所示，Doris 默认在每个节点上为每个算子只会生成 1 个执行实例。这样的话，如果数据量很大，每个执行实例的算子就需要处理大量的数据，而且无法充分利用集群的 CPU、IO、内存等资源。

一个比较容易想到的优化手段是，我们可以在每个节点上为每个算子生成多个执行实例。这样每个算子只需要处理少量数据，而且多个执行实例可以并行执行。

下图是并发度设置为 5 的优化效果，可以看到对于多种类型的查询，会有 3 到 5 倍的查询性能提升：



Colocate Join

Colocate Join（Local Join）是和 Shuffle Join、Broadcast Join 相对的概念，即将两表的数据提前按照 Join Key Shard，这样在 Join 执行时就没有数据网络传输的开销，两表可以直接在本地进行 Join。

整个 Colocate Join 在 Doris 中实现的关键点如下：

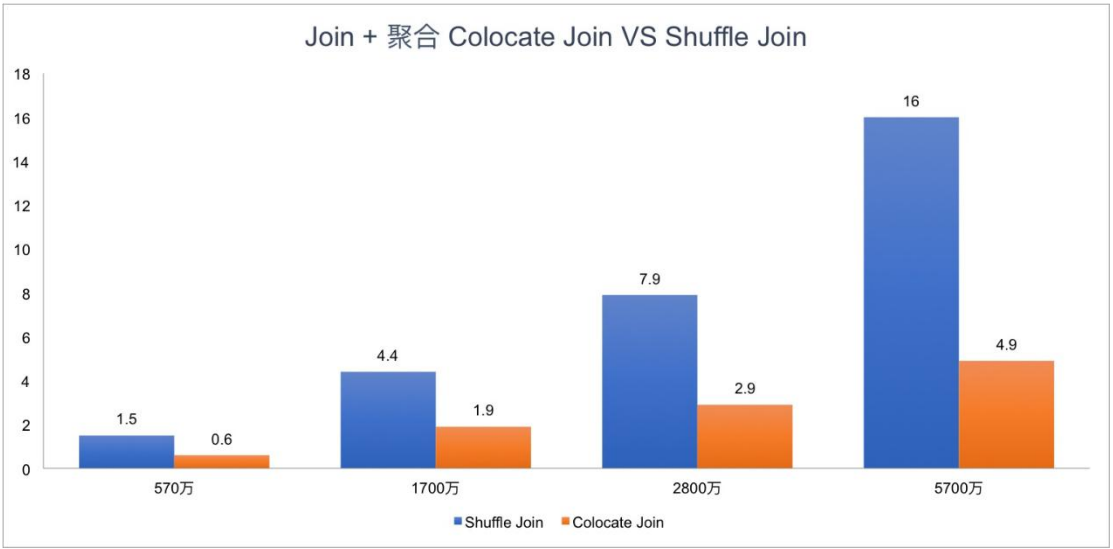
- 数据导入时保证数据本地性。
- 查询调度时保证数据本地性。
- 数据 Balance 后保证数据本地性。
- 查询 Plan 的修改。
- Colocate Table 元数据的持久化和一致性。

- Hash Join 的粒度从 Server 粒度变为 Bucket 粒度。
- Colocate Join 的条件判定。

关于 Doris Colocate Join 的更多实现细节，可以参考《Apache Doris Colocate Join 原理与实践》。

对于下面的 SQL，Doris Colocate Join 和 Shuffle Join 在不同数据量下的性能对比如下：

```
select count(*) FROM A t1 INNER JOIN [shuffle] B t5    ON ((t1.dt = t5.dt) AND (t1.id = t5.id)) INNER JOIN [shuffle] C t6    ON ((t1.dt = t6.dt) AND (t1.id = t6.id)) where t1.dt in (xxx days);
```



Bitmap 精确去重

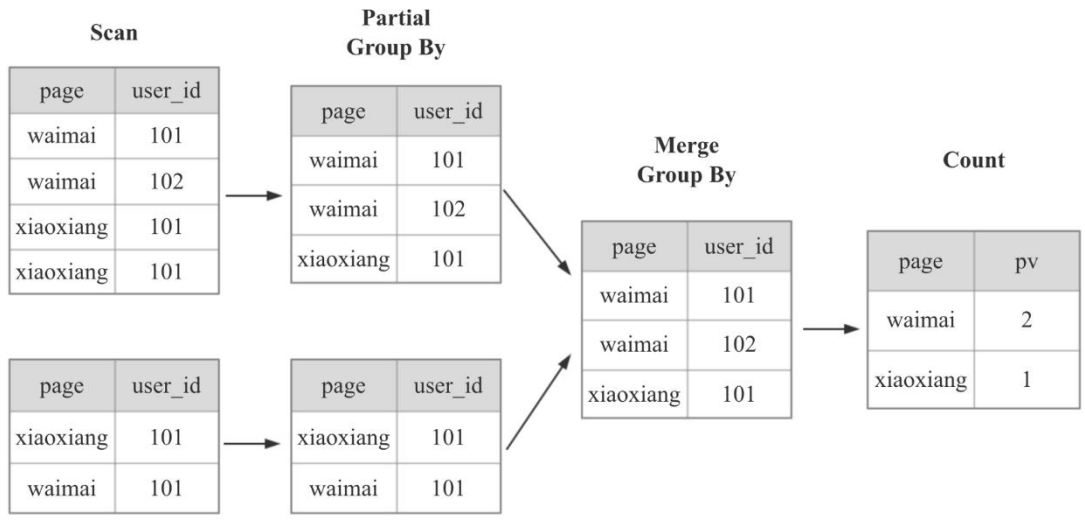
Doris 之前实现精确去重的方式是现场计算的，实现方法和 Spark、MapReduce 类似：

```
+----------+----------+---------+
| dt       | page     | user_id |
+----------+----------+---------+
| 20191206 | xiaoxiang |     101 |
| 20191206 | waimai    |     101 |
| 20191207 | xiaoxiang |     101 |
| 20191207 | waimai    |     102 |
| 20191208 | xiaoxiang |     101 |
| 20191208 | waimai    |     101 |
+----------+----------+---------+
```

```
+-----------+------+
| page      | pv   |
+-----------+------+
| xiaoxiang |    1 |
| waimai    |    2 |
+-----------+------+
```

```
select page, count(distinct user_id) as pv
from table group by page;
```

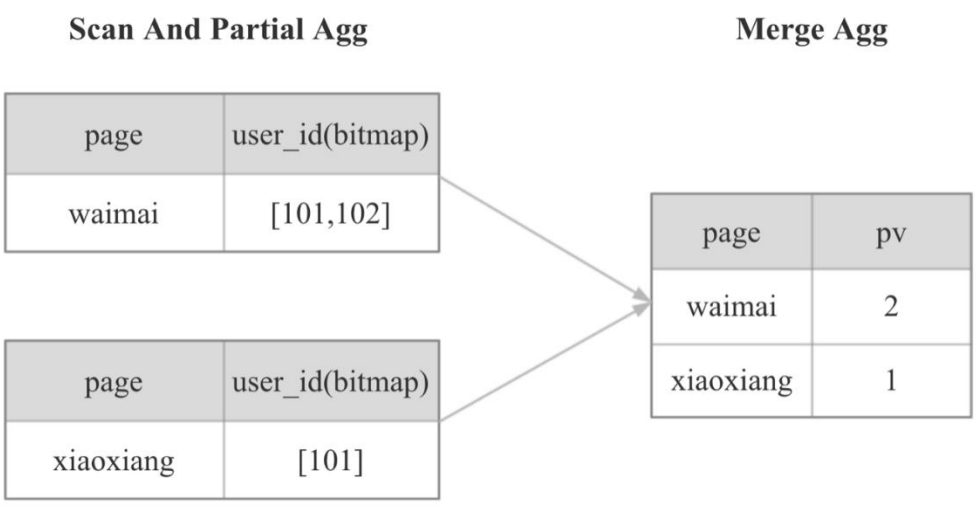
对于上图计算 PV 的 SQL，Doris 在计算时，会按照下图的方式进行计算，先根据 page 列和 user_id 列 group by，最后再 Count：



图中是 6 行数据在 2 个 BE 节点上计算的示意图

显然，上面的计算方式，当数据量越来越大，到几十亿几百亿时，使用的 IO 资源、CPU 资源、内存资源、网络资源会变得越来越多，查询也会变得越来越慢。

于是我们在 Doris 中新增了一种 Bitmap 聚合指标，数据导入时，相同维度列的数据会使用 Bitmap 聚合。有了 Bitmap 后，Doris 中计算精确去重的方式如下：



可以看到，当使用 Bitmap 之后，之前的 PV 计算过程会大幅简化，现场查询时的 IO、CPU、内存，网络资源也会显著减少，并且不再会随着数据规模而线性增加。

总结与思考

在外卖运营分析的业务实践中，由于业务的复杂及应用场景的不同，没有哪一种数据生产方案能够解决所有业务问题。数据库引擎技术的发展，为我们提供更多手段提升数据建设方案。实践证明，以 Doris 引擎为驱动的 ROLAP 模式可以较好地处理汇总与明细、变化维的历史回溯、非预设维的灵活应用、准实时的批处理等场景。而以 Kylin 为基础的 MOLAP 模式在处理增量业务分析，固化维度场景，通过预计算以空间换时间方面依然重要。

业务方面，通过外卖数仓 Doris 的成功实践以及跨 BG 的交流，美团已经有更多的团队了解并尝试使用 Doris 方案。而且在平台同学的共同努力下，引擎性能还有较大提升空间，相信以 Doris 引擎为驱动的 ROLAP 模式会为美团的业务团队

带来更大的收益。从目前实践效果看，其完全有替代 Kylin、Druid、ES 等引擎的趋势。

目前，数据库技术进步飞速，近期柏睿数据发布全内存分布式数据库 RapidsDB v4.0 支持 TB 级毫秒响应（处理千亿数据可实现毫秒级响应）。可以预见，数据库技术的进步将大大改善数仓的分层管理与应用支撑效率，业务将变得“定义即可见”，也将极大地提升数据的价值。

十五、Apache Kylin 的实践与优化

背景

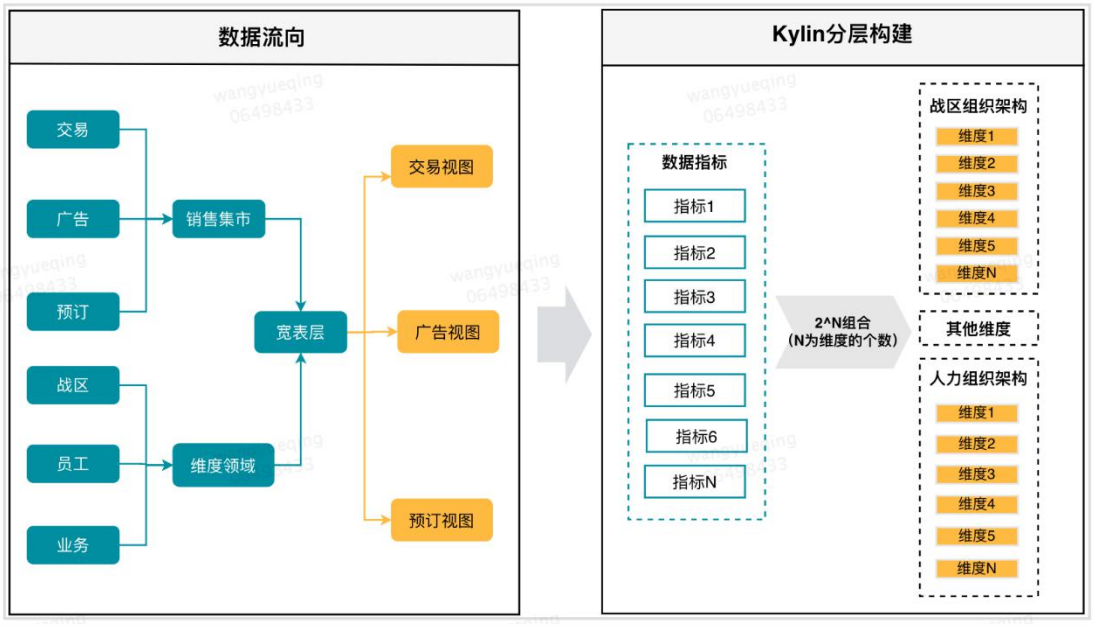
销售业务的特点是规模大、领域多、需求密。美团到店餐饮擎天销售系统（以下简称“擎天”）作为销售数据支持的主要载体，不仅涉及的范围较广，而且面临的技术场景也非常复杂（多组织层级数据展示及鉴权、超过 1/3 的指标需要精准去重，峰值查询已经达到数万级别）。在这样的业务背景下，建设稳定高效的 OLAP 引擎，协助分析人员快速决策，已经成为到餐擎天的核心目标。

Apache Kylin 是一个基于 Hadoop 大数据平台打造的开源 OLAP 引擎，它采用了多维立方体预计算技术，利用空间换时间的方法，将查询速度提升至亚秒级别，极大地提高了数据分析的效率，并带来了便捷、灵活的查询功能。基于技术与业务匹配度，擎天于 2016 年采用 Kylin 作为 OLAP 引擎，接下来的几年里，这套系统高效地支撑了我们的数据分析体系。

2020 年，美团到餐业务发展较快，数据指标也迅速增加。基于 Kylin 的这套系统，在构建和查询上均出现了严重的效率问题，从而影响到数据的分析决策，并给用户体验优化带来了很大的阻碍。技术团队经过半年左右的时间，对 Kylin 进行一系列的优化迭代，包括维度裁剪、模型设计以及资源适配等等等，帮助销售业绩数据 SLA 从 90%提升至 99.99%。基于这次实战，我们沉淀了一套涵盖了“原理解读”、“过程拆解”、“实施路线”的技术方案。希望这些经验与总结，能够帮助业界更多的技术团队提高数据产出与业务决策的效率。

问题与目标

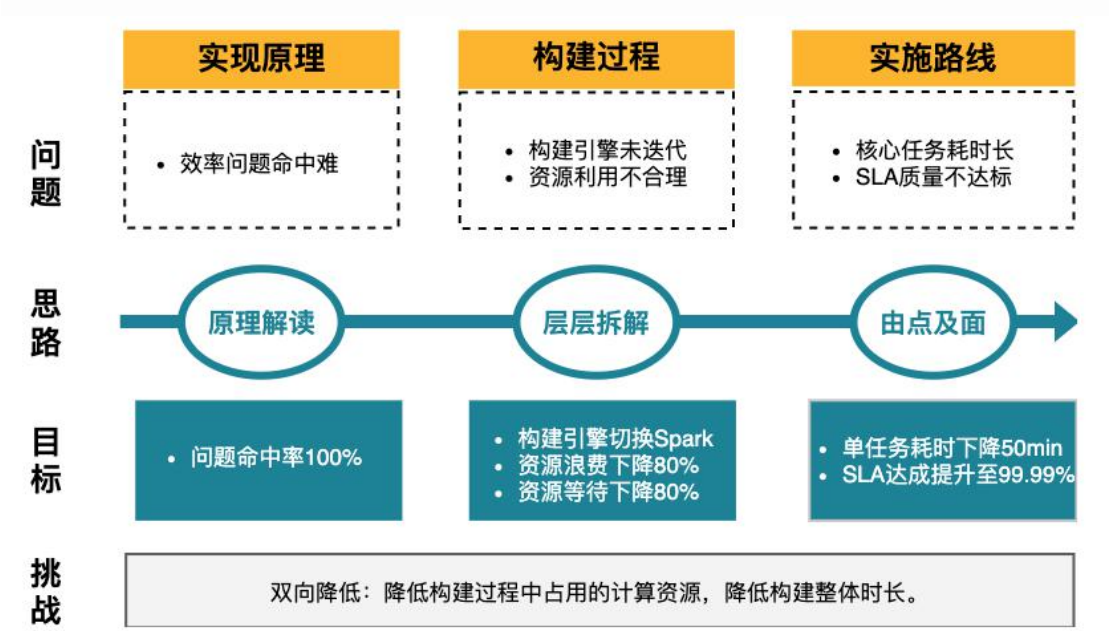
销售作为衔接平台和商家的桥梁，包含销售到店和电话拜访两种业务模式，以战区、人力组织架构逐级管理，所有分析均需要按 2 套组织层级查看。在指标口径一致、数据产出及时等要求下，我们结合 Kylin 的预计算思想，进行了数据的架构设计。如下图所示：



而 Kylin 计算维度组合的公式是 2^N（N 为维度个数），官方提供维度剪枝的方式，减少维度组合个数。但由于到餐业务的特殊性，单任务不可裁剪的组合个数仍高达 1000+。在需求迭代以及人力、战区组织变动的场景下，需要回溯全部历史数据，会耗费大量的资源以及超高的构建时长。而基于业务划分的架构设计，虽能够极大地保证数据产出的解耦，保证指标口径的一致性，但是对 Kylin 构建产生了很大的压力，进而导致资源占用大、耗时长。基于以上业务现状，我们归纳了 Kylin 的 MOLAP 模式下存在的问题，具体如下：

- **效率问题命中难（实现原理）**：构建过程步骤多，各步骤之间强关联，仅从问题的表象很难发现问题的根本原因，无法行之有效地解决问题。
- **构建引擎未迭代（构建过程）**：历史任务仍采用 MapReduce 作为构建引擎，没有切换到构建效率更高的 Spark。
- **资源利用不合理（构建过程）**：资源浪费、资源等待，默认平台动态资源适配方式，导致小任务申请了大量资源，数据切分不合理，产生了大量的小文件，从而造成资源浪费、大量任务等待。
- **核心任务耗时长（实施路线）**：擎天销售交易业绩数据指标的源表数据量大、维度组合多、膨胀率高，导致每天构建的时长超过 2 个小时。
- **SLA 质量不达标（实施路线）**：SLA 的整体达成率未能达到预期目标。

在认真分析完问题，并确定提效的大目标后，我们对 Kylin 的构建过程进行了分类，拆解出在构建过程中能提升效率的核心环节，通过“原理解读”、“层层拆解”、“由点及面”的手段，达成双向降低的目标。具体量化目标如下图所示：



优化前提-原理解读

为了解决效率提升定位难、归因难的问题，我们解读了 Kylin 构建原理，包含了预计算思想以及 By-layer 逐层算法。

预计算

根据维度组合出所有可能的维度，对多维分析可能用到的指标进行预计算，将计算好的结果保存成 Cube。假设我们有 4 个维度，这个 Cube 中每个节点（称作 Cuboid）都是这 4 个维度的不同组合，每个组合定义了一组分析的维度（如 group by），指标的聚合结果就保存在每个 Cuboid 上。查询时，我们根据 SQL 找到对应的 Cuboid，读取指标的值，即可返回。如下图所示：

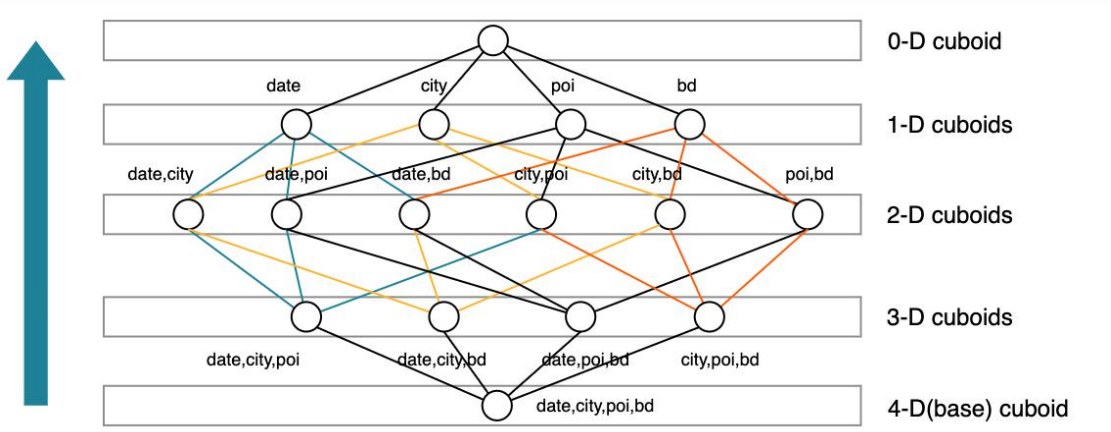
date	city	poi	bd	gtv
2010	北京	poi1	tom	10
2010	上海	poi2	ww	100
2010	上海	poi2	jo	20

Tag	Key	Value
row1	2010,北京,poi1,tom	10
row1	2010,北京,poi1,*	10
row1	2010,北京,*,tom	10
row1	2010,*,poi1,tom	10
row1	*,北京,poi1,tom	10
row1	2010,北京,*,*	10
row1	2010,*,*,tom	10
row1	2010,*,poi1,*	10
row1	*,北京,poi1,*	10
row1	*,北京,*,tom	10
row1	*,*,poi1,tom	10
row1	2010,*,*,*	10
row1	*,北京,*,*	10
row1	*,*,poi1,*	10
row1	*,*,*,tom	10
row1	*,*,*,*	10
row2	2010,上海,poi2, ww	100
row2	2010,上海,poi2,*	100
row2	2010,上海,*, ww	100
row2	2010,*,poi2, ww	100
row2	*,上海,poi2, ww	100
row2	2010,上海,*,*	100
row2	2010,*,*, ww	100
row2	2010,*,poi2,*	100
row2	*,上海,poi2,*	100
row2	*,上海,*, ww	100
row2	*,*,poi2, ww	100
row2	2010,*,*,*	100
row2	*,上海,*,*	100
row2	*,*,poi2,*	100
row2	*,*,*, ww	100
row2	*,*,*,*	100
row3	2010,上海,poi2,jo	20
row3	2010,上海,poi2,*	20
row3	2010,上海,*,jo	20
row3	2010,*,poi2,jo	20
row3	*,上海,poi2,jo	20
row3	2010,上海,*,*	20
row3	2010,*,*, jo	20
row3	2010,*,poi2,*	20
row3	*,上海,poi2,*	20
row3	*,上海,*, jo	20
row3	*,*,poi2, jo	20
row3	2010,*,*,*	20
row3	*,上海,*,*	20
row3	*,*,poi2,*	20
row3	*,*,*, jo	20
row3	*,*,*,*	20

Key	Row1	Row2	Row3
2010,北京,poi1,tom	10		
2010,北京,poi1,*	10		
2010,北京,*,tom	10		
2010,*,poi1,tom	10		
*,北京,poi1,tom	10		
2010,北京,*,*	10		
2010,*,*,tom	10		
2010,*,poi1,*	10		
,北京,poi1,	10		
,北京,,tom	10		
,,poi1,tom	10		
2010,*,*,*	10	20	100
,北京,,*	10		
,,poi1,*	10		
,,*,tom	10		
,,*,*	10	20	100
2010,上海,poi2, ww		20	
2010,上海,poi2,*		20	100
2010,上海,*, ww		20	
2010,*,poi2, ww		20	
*,上海,poi2, ww		20	
2010,上海,*,*		20	100
2010,*,*, ww		20	
2010,*,poi2,*		20	100
,上海,poi2,		20	
,上海,, ww		20	
,,poi2, ww		20	
,上海,,*		20	100
,,poi2,*		20	100
,,*, ww		20	
2010,上海,poi2,jo			100
2010,上海,*,jo			100
2010,*,poi2,jo			100
*,上海,poi2,jo			100
2010,*,*, jo			100
,上海,, jo			100
,,poi2, jo			100
,,*, jo			100

By-layer 逐层算法

一个 N 维的 Cube，是由 1 个 N 维子立方体、N 个(N-1)维子立方体、N*(N-1)/2 个(N-2)维子立方体、……N 个 1 维子立方体和 1 个 0 维子立方体构成，总共有 2^N 个子立方体。在逐层算法中，按照维度数逐层减少来计算，每个层级的计算（除了第一层，由原始数据聚合而来），是基于上一层级的计算结果来计算的。例如：group by [A,B]的结果，可以基于 group by [A,B,C]的结果，通过去掉 C 后聚合得来的，这样可以减少重复计算，当 0 维 Cuboid 计算出来的时候，整个 Cube 的计算也就完成了。如下图所示：



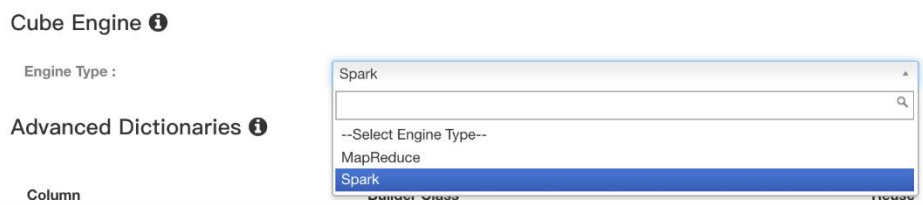
过程分析-层层拆解

在了解完 Kylin 的底层原理后，我们将优化的方向锁定在“引擎选择”、“数据读取”、“构建字典”、“分层构建”、“文件转换”五个环节，再细化各阶段的问题、思路及目标后，我们终于做到了在降低计算资源的同时降低了耗时。详情如下表所示：

分类	引擎选择	数据读取	构建字典	分层构建	文件转换
问题	• 历史任务仍采用MapReduce作为构建引擎	• 按月回刷数据，存在小文件问题	• 精确去重指标多，单任务构建字最大耗时20min	• 率问题定位难、归因难 • 单任务最大申请内存7T+ • 输出3万+的小文件 • 单任务最大耗时40min	• 单任务最大申请内存6T • 单任务最大耗时35min
思路	• 结合官网以及平台测试文档，比对MapReduce和Spark的优劣势	• Hive中源表文件合并 • Kylin参数配置	• 全局字典依赖配置，子集依赖全集 • 高基维度增加计算资源，提升效率	• 原理解读，找到问题本质 • 参数调整，合理适配资源	• 参数调整，合理适配资源 • 降低Map任务申请个数
目标	• 切换Spark	• 耗时下降10%	• 耗时下降20%	• 问题命中率100% • 资源申请下降80% • 小文件数量下降90% • 单任务最大耗时下降20min	• 资源申请下降80% • 单任务最大耗时下降20min

构建引擎选择

目前，我们已经将构建引擎已逐步切换为 Spark。擎天早在 2016 年就使用 Kylin 作为 OLAP 引擎，历史任务没有切换，仅仅针对 MapReduce 做了参数优化。其实在 2017 年，Kylin 官网已启用 Spark 作为构建引擎（官网启用 Spark 构建引擎），构建效率相较 MapReduce 提升 1 至 3 倍，还可通过 Cube 设计选择切换，如下图所示：



读取源数据

Kylin 以外部表的方式读取 Hive 中的源数据，表中的数据文件（存储在 HDFS）作为下一个子任务的输入，此过程可能存在小文件问题。当前，Kylin 上游数据宽表文件数分布比较合理，无需在上游设置合并，如果强行合并反而会增加上游源表数据加工时间。

对于项目需求，要回刷历史数据或增加维度组合，需要重新构建全部的数据，通常采用按月构建的方式回刷历史，加载的分区过多出现小文件问题，导致此过程执行缓慢。在 Kylin 级别重写配置文件，对小文件进行合并，减少 Map 数量，可有效地提升读取效率。

合并源表小文件：合并 Hive 源表中小文件个数，控制每个 Job 并行的 Task 个数。调整参数如下表所示：

参数	值	说明
spark.sql.mergeSmallFileSize	67108864（64MB）	触发小文件合并的阈值
spark.sql.targetBytesInPartitionWhenMerge	67108864（64MB）	设置额外的合并job的map端输入size

Kylin 级别参数重写：设置 Map 读取过程的文件大小。调整参数如下表所示：

参数	值	说明
kylin.engine.spark-conf.spark.hadoopRDD.targetBytesInPartition	67108864（64MB）	Map端输入size，平台默认35M

构建字典

Kylin 通过计算 Hive 表出现的维度值，创建维度字典，将维度值映射成编码，并保存保存统计信息，节约 HBase 存储资源。每一种维度组合，称为一个 Cuboid。理论上来说，一个 N 维的 Cube，便有 2^N 种维度组合。

组合数量查看

在对维度组合剪枝后，实际计算维度组合难以计算，可通过执行日志（截图为提取事实表唯一列的步骤中，最后一个 Reduce 的日志），查看具体的维度组合数量。如下图所示：

```
INFO [main] org.apache.kylin.cube.CubeManager: Loaded 1 cubes, fail on 0 cubes
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Reducer 38 handling stats
INFO [main] org.apache.kylin.engine.mr.KylinReducer: Accepting Reducer Key with ordinal: 1
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Total cuboid number:   718
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Samping percentage:   100
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: The following statistics are collected based on sampling data.
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Number of Mappers: 0
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Cuboid 1015810 row count is:    42
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Cuboid 1016064 row count is:    109
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Cuboid 1016066 row count is:    136
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Cuboid 1016192 row count is:    473
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Cuboid 1016194 row count is:    605
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Cuboid 1016256 row count is:    2882
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Cuboid 1016258 row count is:    3816
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Cuboid 1016288 row count is:    2875
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Cuboid 1016290 row count is:    3788
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Cuboid 1016304 row count is:    4257
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Cuboid 1016306 row count is:    5604
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Cuboid 1016832 row count is:    489613
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Cuboid 1016834 row count is:    479490
INFO [main] org.apache.kylin.engine.mr.steps.FactDistinctColumnsReducer: Cuboid 1017088 row count is:    481652
```

全局字典依赖

擎天有很多业务场景需要精确去重，当存在多个全局字典列时，可设置列依赖，例如：当同时存在“门店数量”、“在线门店数量”数据指标，可设置列依赖，减少对超高基维度的计算。如下图所示：

Advanced Dictionaries

Column	Builder Class	Reuse
POI_PK	org.apache.kylin.dict.GlobalDictionaryBuilder	
B_OBJECT_PK	org.apache.kylin.dict.GlobalDictionaryBuilder	
B_OBJECT_POI_PK	org.apache.kylin.dict.GlobalDictionaryBuilder	
ONLINE_POI		POI_PK
OFFLINE_POI		POI_PK
NEW_POI		POI_PK
DX_POI		POI_PK
EFFECT_DX_POI		POI_PK
B_ONLINE_OBJECT		B_OBJECT_PK
B_NEW_OBJECT		B_OBJECT_PK
B_DX_OBJECT		B_OBJECT_PK
B_EFFECT_DX_OBJECT		B_OBJECT_PK

计算资源配置

当指标中存在多个精准去重指标时，可适当增加计算资源，提升对高基维度构建的效率。参数设置如下表所示：

参数	值	说明
kylin.engine.mr.build-uhc-dict-in-additional-step	TRUE	默认值为 FALSE，设置为 TRUE
kylin.engine.mr.uhc-reducer-count	5	默认值为 1，可以设置为 5，即为每个超高基的列分配 5 个Reduce

分层构建

此过程为 Kylin 构建的核心，切换 Spark 引擎后，默认只采用 By-layer 逐层算法，不再自动选择（By-layer 逐层算法、快速算法）。Spark 在实现 By-layer 逐层算法的过程中，从最底层的 Cuboid 一层一层地向上计算，直到计算出最顶层的 Cuboid（相当于执行了一个不带 group by 的查询），将各层的结果数据缓存到内存中，跳过每次数据的读取过程，直接依赖上层的缓存数据，大大提高了执行效率。Spark 执行过程具体内容如下。

Job 阶段

Job 个数为 By-layer 算法树的层数，Spark 将每层结果数据的输出，作为一个 Job。如下图所示：

Stage 阶段

每个 Job 对应两个 Stage 阶段，分为读取上层缓存数据和缓存该层计算后的结果数据。如下图所示：

Task 并行度设置

Kylin 根据预估每层构建 Cuboid 组合数据的大小（可通过维度剪枝的方式，减少维度组合的数量，降低 Cuboid 组合数据的大小，提升构建效率，本文暂不详细介绍）和分割数据的参数值计算出任务并行度。计算公式如下：

- Task 个数计算公式：Min(MapSize/cut-mb ，MaxPartition) ；Max(MapSize/cut-mb ，MinPartition)
  - MapSize：每层构建的 Cuboid 组合大小，即：Kylin 对各层级维度组合大小的预估值。
  - cut-mb：分割数据大小，控制 Task 任务并行个数，可通过 kylin.engine.spark.rdd-partition-cut-mb 参数设置。
  - MaxPartition：最大分区，可通过 kylin.engine.spark.max-partition 参数设置。
  - MinPartition：最小分区，可通过 kylin.engine.spark.min-partition 参数设置。
- 输出文件个数计算：每个 Task 任务将执行完成后的结果数据压缩，写入 HDFS，作为文件转换过程的输入。文件个数即为：Task 任务输出文件个数的汇总。

资源申请计算

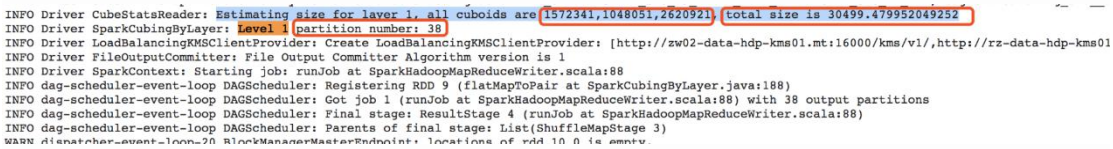
平台默认采用动态方式申请计算资源，单个 Executor 的计算能力包含：1 个逻辑 CPU（以下简称 CPU）、6GB 堆内内存、1GB 的堆外内存。计算公式如下：

- CPU = kylin.engine.spark-conf.spark.executor.cores * 实际申请的 Executors 个数。
- 内存 =（kylin.engine.spark-conf.spark.executor.memory + spark.yarn.executor.memoryOverhead）* 实际申请的 Executors 个数。
- 单个 Executor 的执行能力 = kylin.engine.spark-conf.spark.executor.memory / kylin.engine.spark-conf.spark.executor.cores，即：1 个 CPU 执行过程中申请的内存大小。
- 最大 Executors 个数 = kylin.engine.spark-conf.spark.dynamicAllocation.maxExecutors，平台默认动态申请，该参数限制最大申请个数。

在资源充足的情况下，若单个 Stage 阶段申请 1000 个并行任务，则需要申请资源达到 7000GB 内存和 1000 个 CPU，即：`CPU：1*1000=1000；内存：（6+1）*1000=7000GB`。

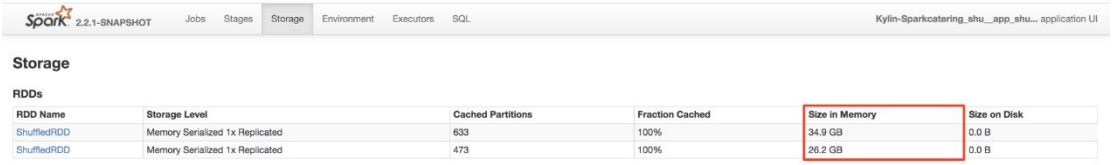
资源合理化适配

由于 By-layer 逐层算法的特性，以及 Spark 在实际执行过程中的压缩机制，实际执行的 Task 任务加载的分区数据远远小于参数设置值，从而导致任务超高并行，占用大量资源，同时产生大量的小文件，影响下游文件转换过程。因此，合理的切分数据成为优化的关键点。通过 Kylin 构建日志，可查看各层级的 Cuboid 组合数据的预估大小，以及切分的分区个数（等于 Stage 阶段实际生成的 Task 个数）。如下图所示：



结合 Spark UI 可查看实执行情况，调整内存的申请，满足执行所需要的资源即可，减少资源浪费。

1.整体资源申请最小值大于 Stage 阶段 Top1、Top2 层级的缓存数据之和，保证缓存数据全部在内存。如下图所示：



计算公式：Stage 阶段 Top1、Top2 层级的缓存数据之和 < kylin.engine.spark-conf.spark.executor.memory * kylin.engine.spark-conf.spark.memory.fraction * spark.memory.storageFraction *最大 Executors 个数

2.单个 Task 实际所需要的内存和 CPU（1 个 Task 执行使用 1 个 CPU）小于单个 Executor 的执行能力。如下图所示：

计算公式：单个 Task 实际所需要的内存 <

kylin.engine.spark-conf.spark.executor.memory *

kylin.engine.spark-conf.spark.memory.fraction *

spark.memory.st·orageFraction /

kylin.engine.spark-conf.spark.executor.cores。参数说明如下表所示：

文件转换

Kylin 将构建之后的 Cuboid 文件转换成 HTable 格式的 Hfile 文件，通过 BulkLoad 的方式将文件和 HTable 进行关联，大大降低了 HBase 的负载。此过

程通过一个 MapReduce 任务完成，Map 个数为分层构建阶段输出文件个数。日志如下：

此阶段可根据实际输入的数据文件大小（可通过 MapReduce 日志查看），合理申请计算资源，避免资源浪费。

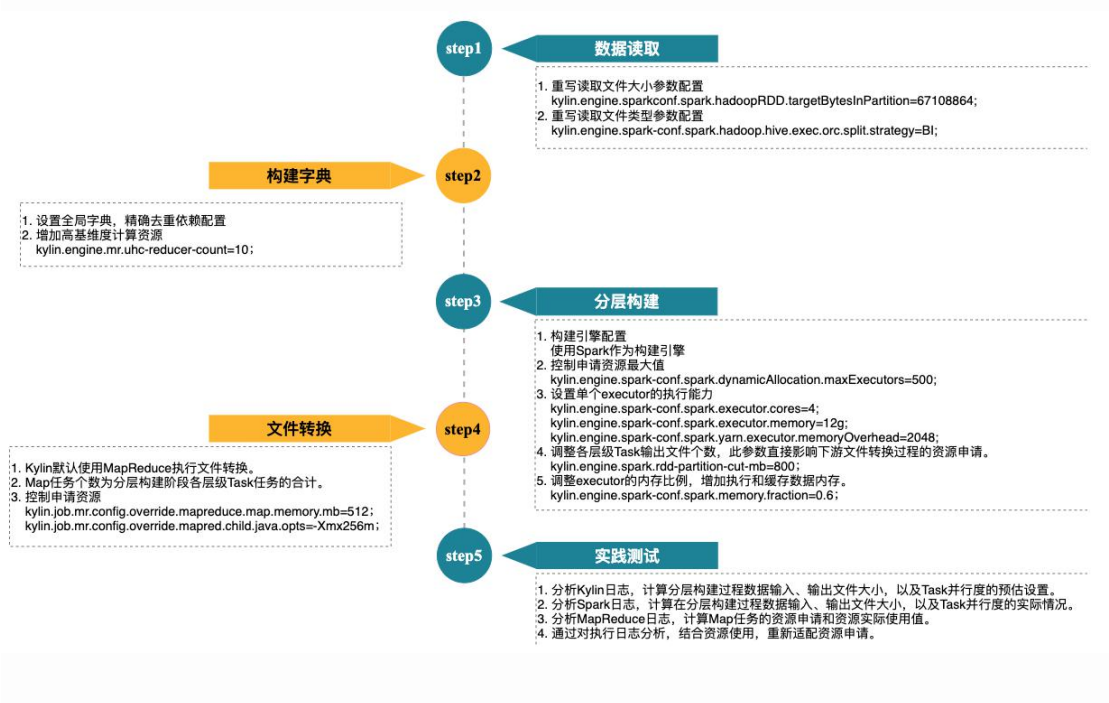
计算公式：Map 阶段资源申请 = kylin.job.mr.config.override.mapreduce.map.memory.mb * 分层构建阶段输出文件个数。具体参数如下表所示：

参数	值	说明
kylin.job.mr.config.override.mapreduce.map.memory.mb	3000	平台默认3000mb，Container 的内存
kylin.job.mr.config.override.mapred.child.java.opts	-Xmx2048m	平台默认2048mb，jvm进程占用的内存

实施路线-由点及面

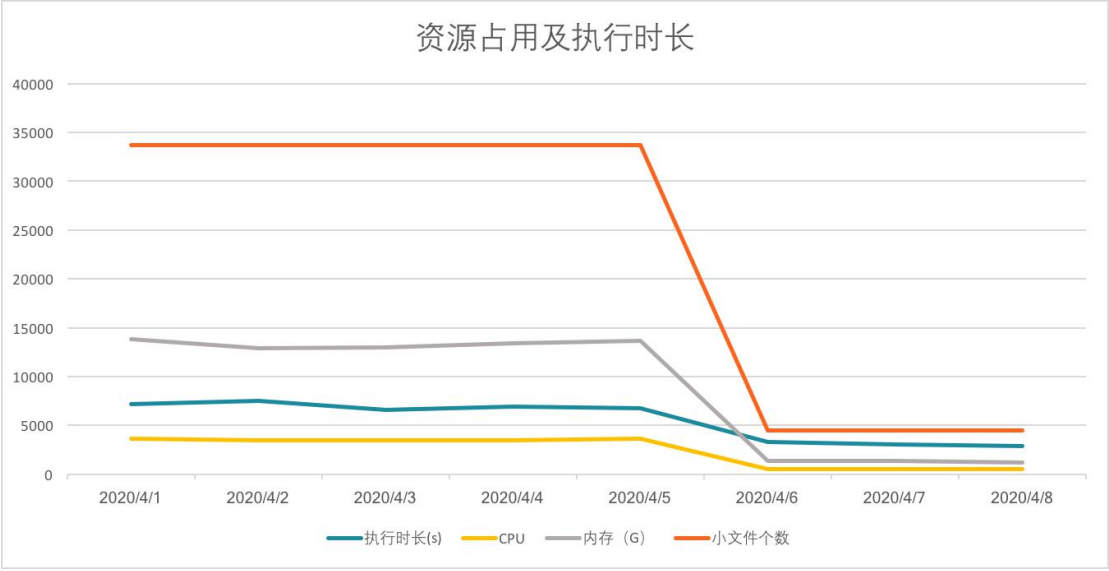
交易试点实践

我们通过对 Kylin 原理的解读以及构建过程的层层拆解，选取销售交易核心任务进行试点实践。如下图所示：



实践结果对比

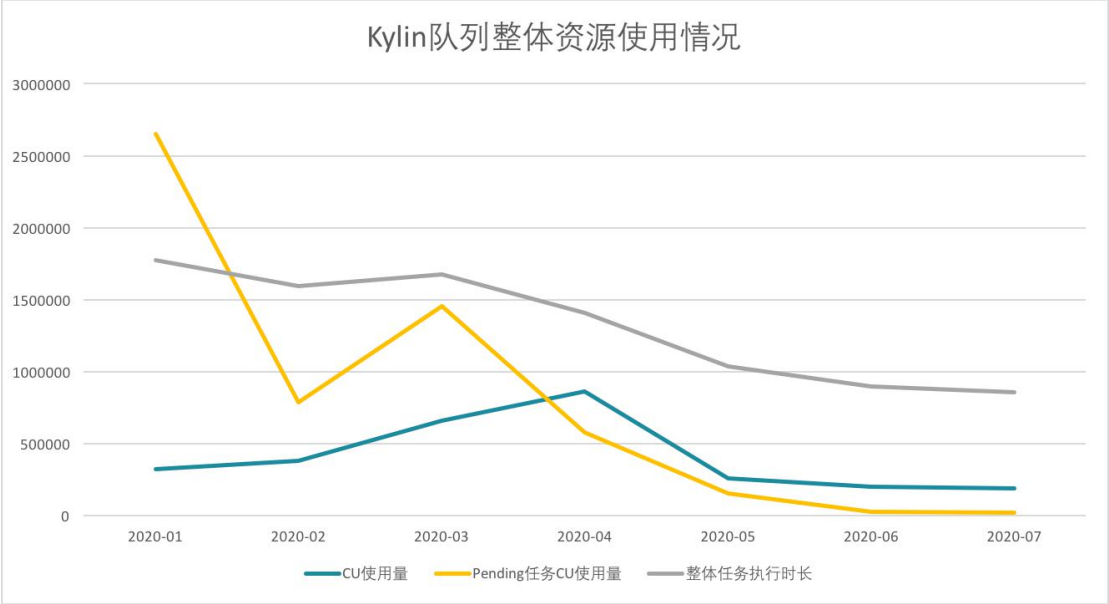
针对销售交易核心任务进行实践优化，对比调整前后资源实际使用情况和执行时长，最终达到双向降低的目标。如下图所示：



成果展示

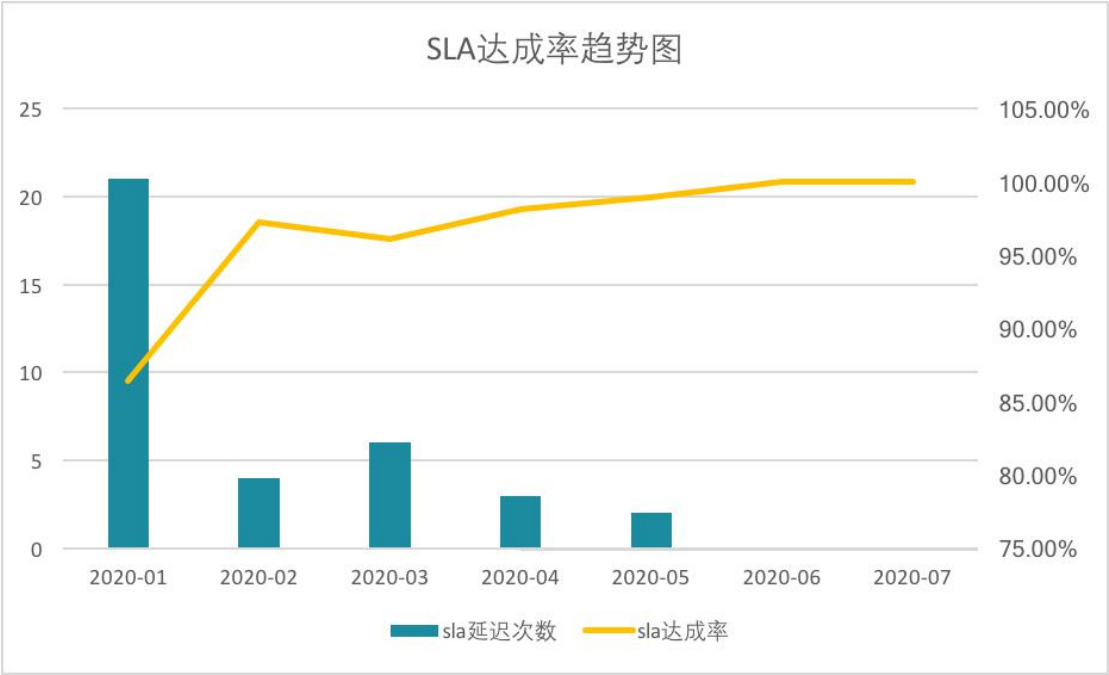
资源整体情况

擎天现有 20+的 Kylin 任务，经过半年时间持续优化迭代，对比 Kylin 资源队列月均 CU 使用量和 Pending 任务 CU 使用量，在同等任务下资源消耗已明显降低。如下图所示：



SLA 整体达成率

经过了由点及面的整体优化，擎天于 2020 年 6 月 SLA 达成率达到 100%。如下图所示：



展望

Apache Kylin 在 2015 年 11 月正式成为 Apache 基金会的顶级项目。从开源到成为 Apache 顶级项目，只花了 13 个月的时间，而且它也是第一个由中国团队完整贡献到 Apache 的顶级项目。目前，美团采用比较稳定的 V2.0 版本，经过近 4 年的使用与积累，到餐技术团队在优化查询性能以及构建效率层面都积累了大量经验，本文主要阐述了在 Spark 构建过程的资源适配方法。值得一提的是，Kylin 官方在 2020 年 7 月发布了 V3.1 版本，引入了 Flink 作为构建引擎，统一使用 Flink 构建核心过程，包含数据读取阶段、构建字典阶段、分层构建阶段、文件转换阶段，以上四部分占整体构建耗时的 95%以上。此次版本的升级也大幅度提高了 Kylin 的构建效率。详情可查看：Flink Cube Build Engine。

回顾 Kylin 构建引擎的升级过程，从 MapReduce 到 Spark，再到如今的 Flink，构建工具的迭代始终向更加优秀的主流引擎在靠拢，而且 Kylin 社区有很多活跃

的优秀代码贡献者，他们也在帮助扩大 Kylin 的生态，增加更多的新功能，非常值得大家学习。最后，美团到店餐饮技术团队再次表达对 Apache Kylin 项目团队的感谢。

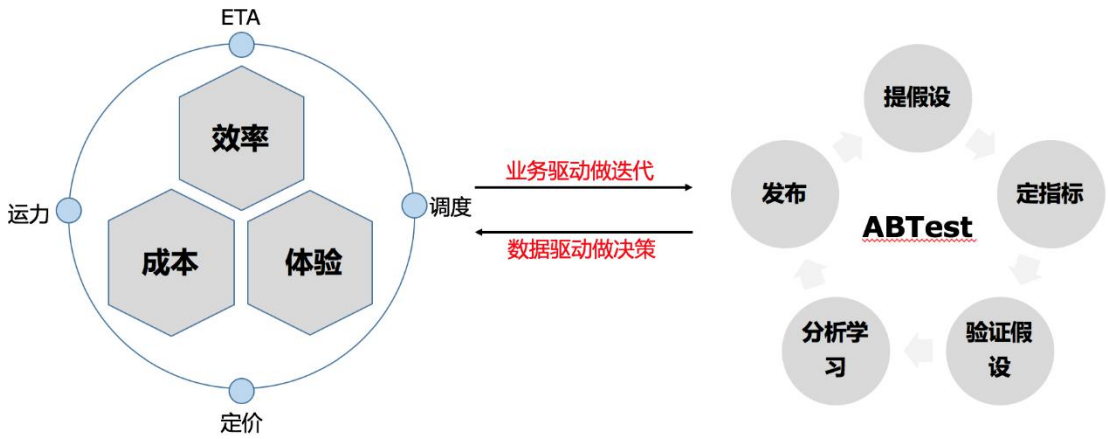
A/B 测试

十六、美团配送 A/B 评估体系建设实践

2019 年 5 月 6 日，美团点评正式推出新品牌“美团配送”，发布了美团配送新愿景：“每天完成一亿次值得信赖的配送服务，成为不可或缺的生活基础设施。”现在，美团配送已经服务于全国 400 多万商家和 4 亿多用户，覆盖 2800 余座市县，日活跃骑手超过 70 万人，成为全球领先的分钟级配送网络。

即时配送的三要素是“效率”、“成本”、“体验”，通过精细化的策略迭代来提升效率，降低成本，提高体验，不断地扩大规模优势，从而实现正向循环。但是，策略的改变，不是由我们随便“拍脑袋”得出，而是一种建立在数据基础上的思维方式，数据反馈会告诉我们做的好不好，哪里有问题，以及衡量可以带来多少确定性的增长。而 A/B-test 就是我们精细化迭代的一个“利器”，通过为同一个迭代目标制定两个或多个版本的方案，在同一时间维度，让组成成分相同（或相似）的 A/B 群组分别采用这些版本，然后收集各群组的体验数据和业务数据，最后分析、评估出最好的版本，帮助我们作出正确的决策，使迭代朝着更好的方向去演进。基于此，构建一个适用于配送业务的 A/B 平台就应运而生了。

1. A/B 平台简介

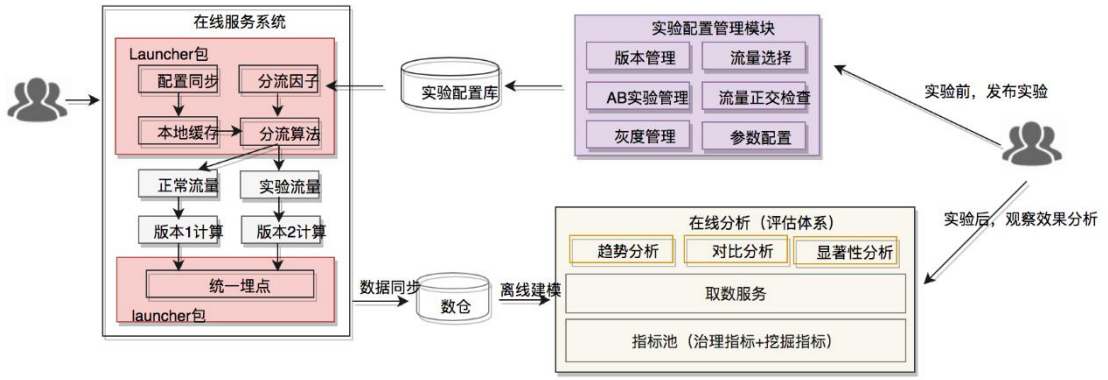


如上图所示，A/B 实验可以看作一个“无尽”的学习环，我们通过提出假设、定义成功指标、检验假设（A/B 实验）、分析学习、发布、建立另一个假设，这就形成一个完整的闭环，通过多轮实验迭代，使策略趋于更优。基于上述对 A/B 实验划分的 5 个步骤，我们将 A/B 实验的完整生命周期分为三个阶段：

- 实验前，提出该实验假设，定义实验成功的指标，确定分流策略；
- 实验中，即验证假设的阶段，根据配置阶段的分流策略进行分流和埋点上报；
- 实验后，进行实验分析与学习，并基于实验报告决定是否发布。

按照功能划分，我们将 A/B 平台分为三个模块，实验配置管理模块、分流以及埋点上报模块和在线分析模块，分别对应于 A/B 实验生命周期的实验前、实验中和实验后三个阶段。在实验配置模块，用户可以基于实验前提出的假设、定义

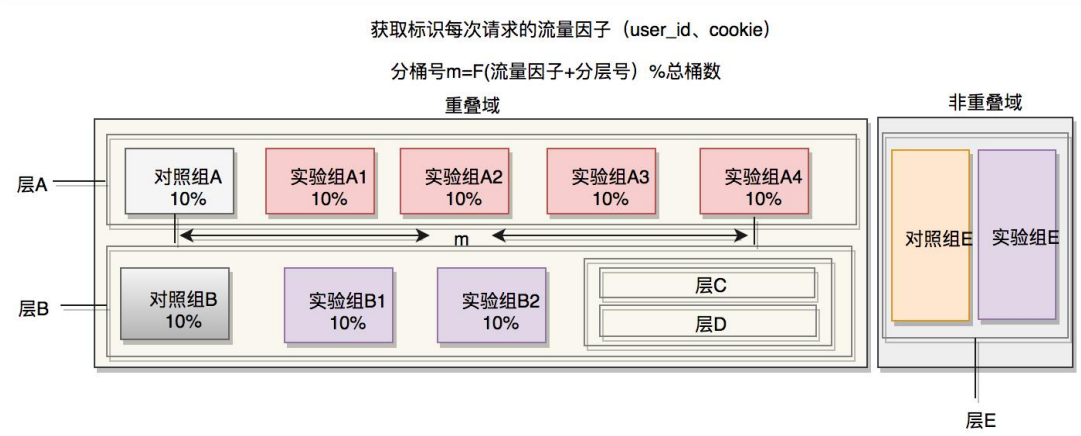
的成功指标快速创建实验，并基于特定的分流策略完成分流配置；分流以及埋点上报模块，提供 JAR 包接入的形式，异步获取实验配置进行本地分流计算和埋点上报；在线分析模块，依据用户在实验配置管理模块选取的用于说明实验效果的指标、分流埋点上报模块记录的日志，自动地产生各实验的实验报告，供实验观察者使用，然后根据实验效果帮助他们作出正确的决策。具体流程如下图所示：



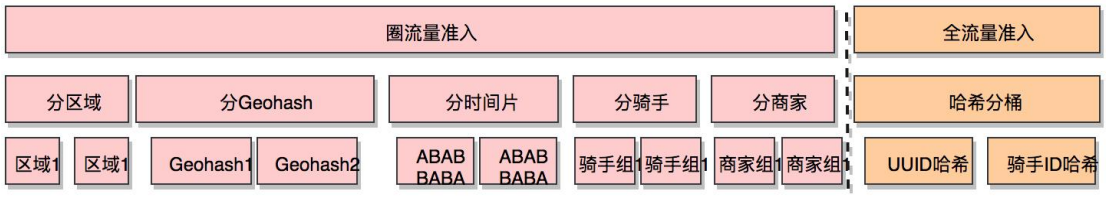
2.为什么要强调评估体系建设

2.1 分流业务场景需要

业界的 A/B 平台建设基本以《Overlapping Experiment Infrastructure: More, Better, Faster Experimentation》这篇论文为蓝本进行展开， 引入分层模型以及在分流算法中加入层编号因子来解决“流量饥饿”和“正交”问题，并且通过引入域的概念，支持域和层之间的相互嵌套，使分层实验模型更加灵活，进而满足多种场景下的 A/B 诉求。如下图所示，将流量通过 Hash 取模的方式即可实现流量的均匀划分。



这种是面向 C 端用户进行流量选择的传统 A/B 实验，采用上述的分流方式基于这样的假设：参与实验的流量因子是相互独立的、随机的，服从独立同分布。但是，配送业务场景下的 A/B 实验，涉及到用户、骑手、商家三端，请求不独立，策略之间相互影响并且受线下因素影响较大。传统 A/B 实验的分流方式，无法保证分出的两个群组实验组和对照组的流量都是无差别的，无法避免因流量分配不平衡而导致的 A/B 群组差异过大问题，很容易造成对实验结果的误判。为满足不同业务场景的诉求，我们的 A/B 平台建设采取了多种分流策略，如下图所示：



针对策略之间的相互影响、请求不独立场景下的 A/B 实验，我们采取限流准入的分流方式，针对不同的实验，选取不同的分流因子。在实验前，我们通过 AA 分组，找出无差别的实验组和对照组，作为我们实验分流配置的依据，这种分流方式要求我们要有一套完整刻画流量因子的指标体系，只要刻画流量因子的指标间无统计显著性，我们就认为分出的实验组和对照组无差别。

2.2 业务决策的重要依据

在实验后的效果评估环节，通常允许实验者用自定义的指标来衡量不同策略带来的影响。但这样做会带来如下两个问题：

- 首先，由实验者来负责实验效果的评估，很难做到客观。同时也无法避免实验者仅仅选择支持自己假设的指标，来证明自己的实验结论；
- 其次，所有的策略迭代都是为业务服务，如果实验者用自定义的、与业务认知不一致的指标，来说明实验效果、推动业务灰度，这种方式往往难以被采纳。

因此，权威的评估体系对于对齐大家认知，并帮助我们在策略迭代方面作出正确的决策，尤为重要。

3.A/B 评估体系构建

A/B 评估体系的构建，要解决 A/B 平台两个核心问题：

- 第一，要有一套用于刻画流量因子（区域、骑手、商家）的权威的、完备的指标体系，帮助实验者完成实验前的 AA 分组和实验后的效果评估；
- 第二，要建立一套科学的评估方法，帮助实验者作出正确的决策。

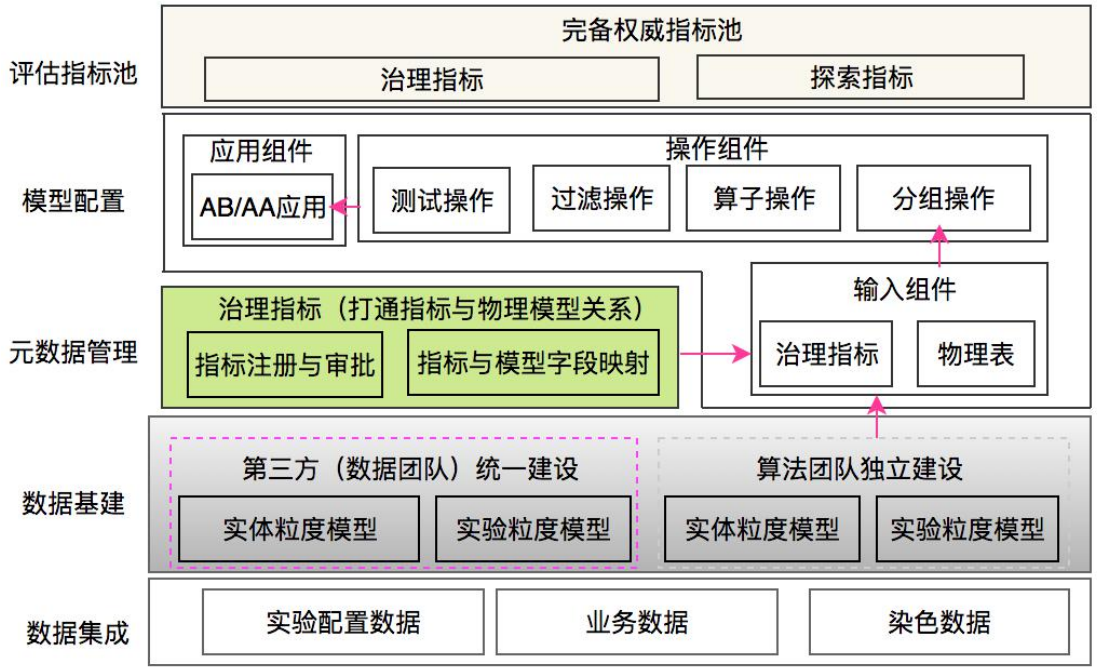
3.1 权威完备的指标体系

指标的权威性体现在：刻画分流因子，用于实验前 AA 分组和证明实验假设的指标，必须经过治理且业务认知一致，这样才能对齐认知，使得实验结果更具说服力；指标的完备性体现在：评估体系中的指标，不仅要有经过第三方独立生产治理且各业务方认知一致的治理指标，而且还要有实验者为了更全面的分析，描述实验过程，自定义的探索指标。

3.1.1 整体架构

治理指标强调的是指标的权威性和生产的规范性，而探索性指标强调的是指标的多样性和生产的灵活性。在评估体系中要实现这两类指标的统一，既要包含用于说明实验效果的治理指标，又要包含帮助实验者更好迭代实验所需的探索指标。

为实现上述的统一，指标层面要有分级运营的策略：治理指标按照业务认知一致性和算法内部认知一致性分别定级为 P0、P1，这一类指标在生产前必须要有严格的注册、评审，生产环节需要交给独立的第三方团队（数据团队）生产，保证指标的权威性，产出后打通指标与字段的映射关系，对用户屏蔽底层实现逻辑；对于探索性指标，定级为 P2，强调的是生产的灵活性和快速实现，因此，它的生产就不宜带有指标注册和评审等环节。为保证其快速实现，希望基于物理表和简单的算子配置就可以实现效果分析时即席查询使用。基于如上的问题拆解，我们进行了如下的架构设计：



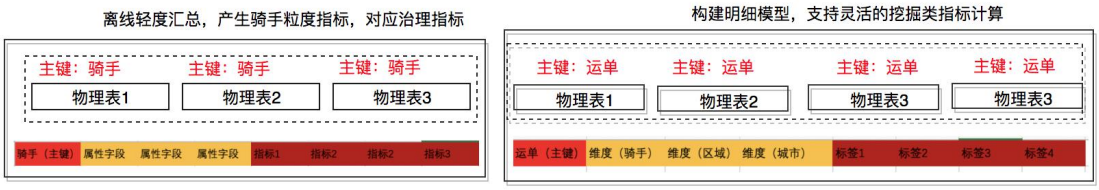
3.1.2 数据集成

为了支持监控和分析，在数据集成环节，我们集成了实验配置数据、业务数据和染色数据，以便实验者在效果评估环节不仅可以查看流量指标（PV、UV 和转化率），也可以深入探索策略变动对业务带来的影响。对于那些在实验配置环节不能确定流量是否真正参加实验的场景（例如：选择了特定区域进行实验，该区域产生的单只有满足特定条件时才能触发实验），我们不能直接通过限制确定的区域来查看业务指标。因为，此时查看的指标并不是真正参与实验的流量所对应的指标。因此在数据集成环节，我们同时将实验前的实验配置数据和实验中的染色数据（针对每个参与实验的流量，每次操作所产生的数据，都会打上实验场景、实验组以及具体的分组标记，我们该数据为染色数据）同步到数仓。在数据基建环节，将业务数据模型和染色数据模型通过流量实体作为关联条件进行关联，构建实验粒度模型。

3.1.3 数据基建

在数据基建层，我们基于指标分级运营的思路，由数据团队和算法团队分别构建实体粒度（区域、骑手、GeoHash）和实验粒度的实体宽表模型，以满足 P0/P1 指标和 P2 指标的诉求；为实现指标的规范化建设和灵活建设的统一，在物理模型和对外提供应用的指标池之间，我们提供了元数据管理工具和模型配置工具，从而实现离线数据快速接入评估体系的指标池。由数据团队建设的实体宽表模型，对应着治理指标（P0/P1 指标），必须在生产后通过元数据管理工具完成指标与物理字段的映射，将指标的加工口径封装在数据层，对用户屏蔽物理实现，确保治理指标的一致性。由算法团队独立建设的实体宽表模型，对应着挖掘指标（P2 指标），为确保其接入评估体系指标池的灵活性和方便性，我们在数据基建环节，

通过标签的形式对指标口径做部分封装，在模型配置环节完成指标逻辑的最终加工。

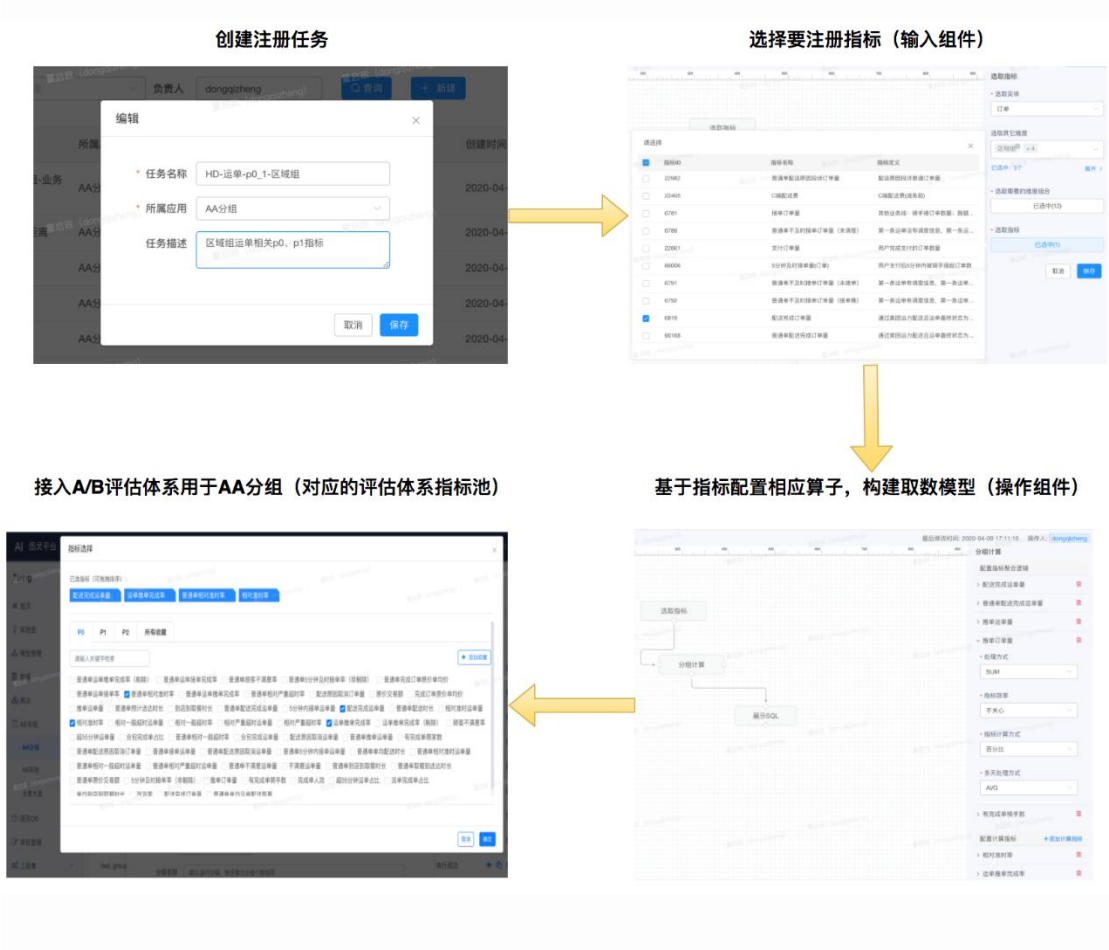


3.1.4 元数据管理

元数据管理层，是实现指标权威性的关键。治理指标在本层实现注册、评审，达到业务认知一致性和算法内部认知一致性的目的。同时，本层还完成了治理指标与数据基建层物理模型之间的绑定，为后续的模型配置建立基础。

3.1.5 模型配置

模型配置工具，是打通物理模型与评估指标池的桥梁，它通过输入组件、操作组件和应用组件，将离线数据接入到评估体系中，满足实验前 AA 分组和实验后 AB 评估的需求。首先，输入组件可以对应不同的数据源，既可以接入治理的离线指标，也可以接入特定库下的物理表。其次，操作组件提供了分组操作、算子操作、过滤操作和测试操作，通过分组操作，确定模型包含的维度；通过算子操作，将算子作用在指标或标签字段上，在取数环节实现指标的二次计算；通过过滤操作，实现数据的过滤；通过测试操作，保证模型配置质量。最后，应用组件可以将配置的模型注册到不同的应用上，针对 A/B 场景主要是 AA 分组和 AB 评估。具体接入流程如下图所示：



3.2 科学权威的评估方式

评估报告的可靠和权威性主要体现在两个方面：一是评估指标的可靠性和权威性；二是评估方式的科学性。在上一节中，我们重点讨论了如何构建可靠权威的指标体系。在这一节，我们重点讨论如何进行科学的评估。

在讨论科学评估之前，我们再重温一下 A/B 实验的定义：A/B 实验，简单来说，就是为同一个目标制定两个版本或多个版本的方案，在同一时间维度，分别让组成成分相同（相似）的 A/B 群组分别采用这些版本，收集各群组的体验数据和业务数据，最后分析、评估出最好版本，正式采用。其中 A 方案为现行的设计（称为控制组）， B 方案是新的设计（称为实验组）。分析 A/B 实验的定义，要实现科学权威的评估，最重要的两点在于：

- 第一，确保在实验前分出无差别的实验组和对照组，避免因流量分配不平衡导致的 AB 群组差异过大，最终造成对于实验结果的误判；
- 第二，确保对实验结果作出准确的判断，能够准确的判断新策略相对于旧策略的优势是不是由自然波动引起的，它的这一优势能否在大规模的推广中反映出来。

无论是实验前确保实验组和对照组流量无显著性差异，还是实验后新策略较旧策略的指标变动是否具有统计上的显著性，无一例外，它们都蕴含着统计学的知识。接下来，我们重点论述一下 A/B 实验所依赖的统计学基础以及如何依据统计学理论做出科学评估。

3.2.1 假设检验

3.2.1.1 两个假设

A/B 测试是一种对比试验，我们圈定一定的流量进行实验，实验结束后，我们基于实验样本进行数据统计，进而验证实验前假设的正确性，我们得出这一有效结论的科学依据便是假设检验。假设检验是利用样本统计量估计总体参数的方法，在假设检验中，先对总体均值提出一个假设，然后用样本信息去检验这个假设是否成立。我们把提出的这个假设叫做原假设，与原假设对立的结论叫做备择假设，如果原假设不成立，就要拒绝原假设，进而接受备择假设。

3.2.1.2 两类错误

对于原假设提出的命题，我们需要作出判断，要么原假设成立，要么原假设不成立。因为基于样本对总体的推断，会面临着犯两种错误的可能：第一类错误，原假设为真，我们却拒绝了；第二类错误，原假设为伪，我们却接受了。显然，我

们希望犯这两类错误的概率越小越好，但对于一定的样本量 n，不能同时做到犯这两类错误的概率很小。

在假设检验中，就有一个对两类错误进行控制的问题。一般来说，哪一类错误所带来的后果严重、危害越大，在假设检验中就应该把哪一类错误作为首要的控制目标。在假设检验中，我们都执行这样一个原则，首先控制犯第一类错误的概率。这也是为什么我们在实际应用中会把要推翻的假设作为原假设，这样得出的结论更具说服力（我们有足够充分的证据证明原来确定的结论是错误的），所以通常会看到，我们把要证明的结论作为备择假设。

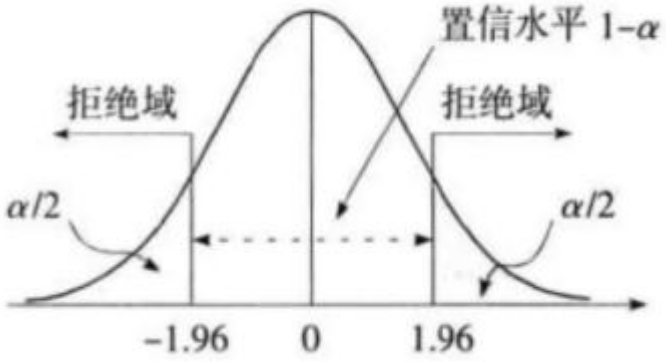
3.2.1.3 T 检验

常见的假设检验方法有 Z 检验、T 检验和卡方检验等，不同的方法有不同的适用条件和检验目标。Z 检验和 T 检验都是用来推断两个总体均值差异的显著性水平，具体选择哪种检验由样本量的大小、总体的方差是否已知决定。在样本量较小且总体的方差未知的情况下，这时只能使用样本方差代替总体方差，样本统计量服从 T 分布，应该采用 T 统计量进行检验。T 统计量具体构造公式如下图所示，其中 f 是 T 统计量的自由度，S1、S2 是样本标准差。

$$t=\frac{(\bar{x}_1-\bar{x}_2)-(\mu_1-\mu_2)}{\sqrt{\frac{s_1^2}{n_1}+\frac{s_2^2}{n_2}}}$$

$$f=\frac{\left(\frac{s_1^2}{n_1}+\frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1-1}+\frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2-1}}$$

T 检验的流程是，在给定的弃真错误概率下（一般取 0.05），依据样本统计量 T 是否落在拒绝域来判断接受还是拒绝原假设。实际上在确定弃真错误概率以后，拒绝域的位置也就相应地确定了。使用 T 统计量进行判断的好处是，进行决策的界限清晰，但缺陷是决策面临的风险是笼统的。例如 T=3 落入拒绝域，我们拒绝原假设，犯弃真错误的概率为 0.05；T=2 也落入拒绝域，我们拒绝原假设，犯弃真错误的概率也是 0.05。事实上，依据不同的统计量进行决策，面临的风险也是有差别的。为了精确地反映决策的风险度，我们仍然需要 P 值来帮助业务来做决策。



3.2.1.4 利用 P 值决策

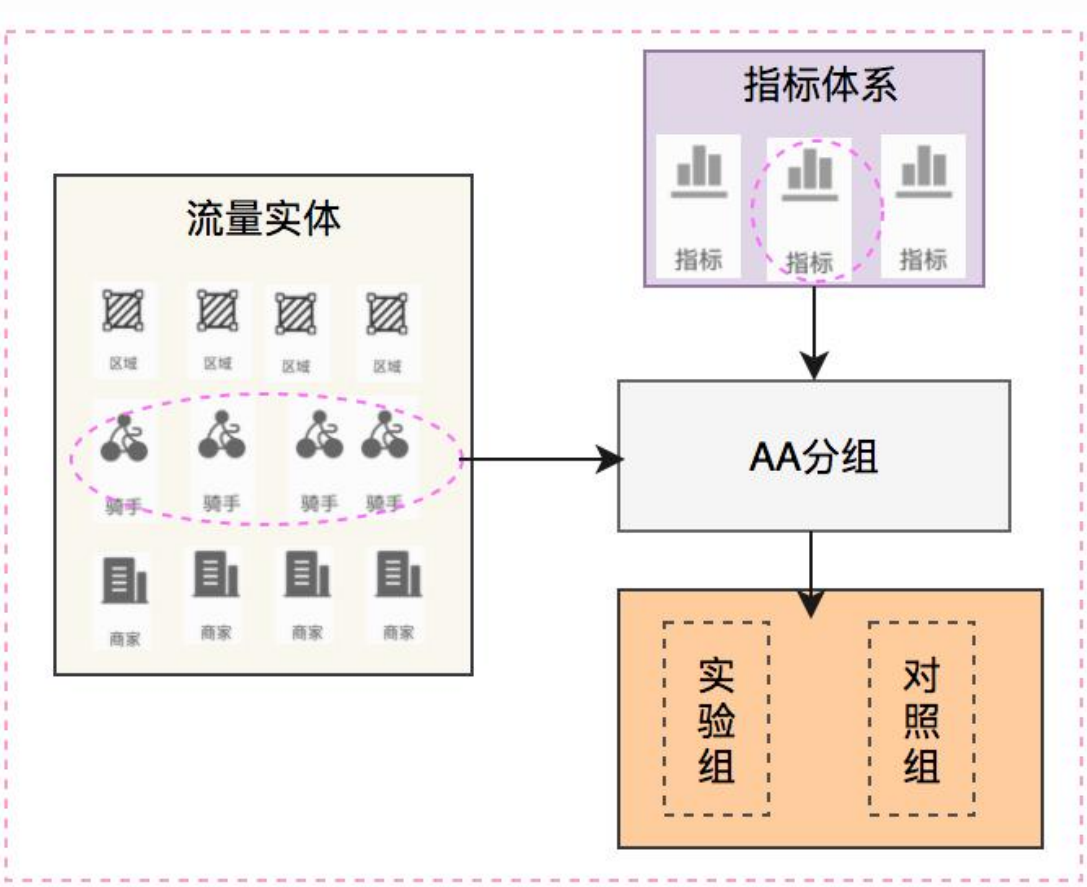
P 值是当原假设为真时，所得到的样本观察结果或更极端的结果出现的概率。如果 P 值很小，说明这种情况发生的概率很小，但是在这次试验中却出现了，根据小概率原理，我们有理由拒绝原假设，P 值越小，我们拒绝原假设的理由越充分。P 值可以理解为犯弃真错误的概率，在确定的显著性水平下（一般取 0.05），P 值小于显著性水平，则拒绝原假设。

3.2.2 基于假设检验的科学评估

围绕着科学评估要解决的两个问题，实验前，针对圈定的流量使用假设检验加上动态规划算法，确保分出无差别的实验组和对照组；实验后，基于实验前选定的用于验证假设结论的指标，构造 T 统计量并计算其对应的 P 值，依据 P 值帮我们做决策。

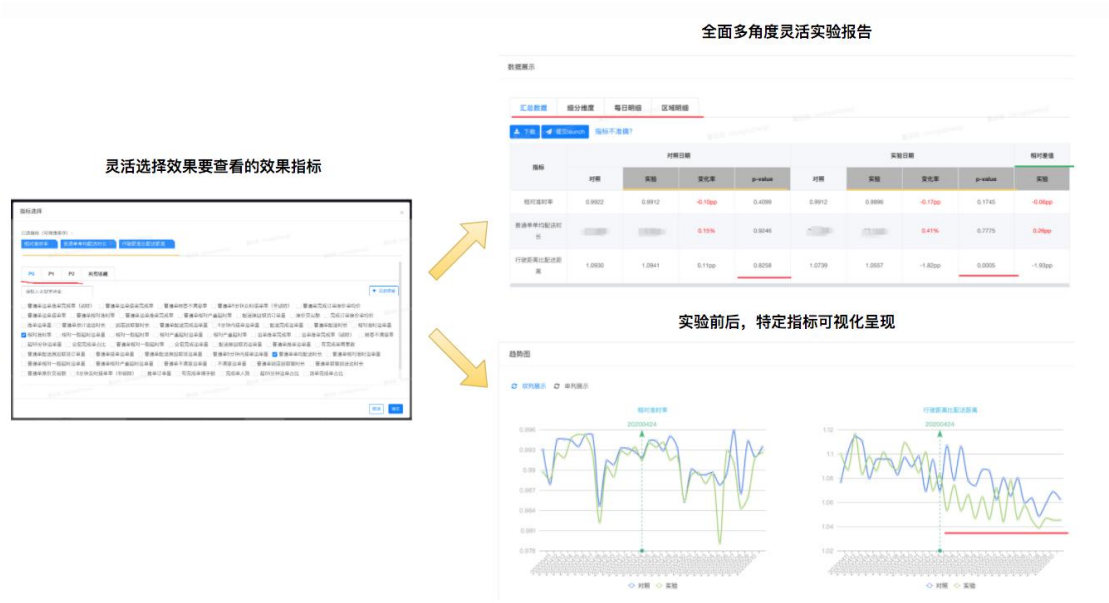
3.2.2.1 AA 分组

首先看如何解决第一个问题：避免因流量分配不平衡，A/B 组本身差异过大造成对实验结果的误判。为解决该问题，我们引入了 AA 分组：基于实验者圈定的流量，通过 AA 分组将该流量分为无显著性差异的实验组和对照组。我们这样定义无显著性差异这一约束：首先，实验者选取的用于刻画实验流量的指标，在实验组和对照组之间无统计上的显著性（即上节所描述的基于均值的假设检验）；其次，在所分出的实验组和对照组之间，这些指标的差值最小，即一个寻找最优解的过程。从实验者的实验流程看，在实验前，圈定进入该实验的流量，然后确定用于刻画实验流量的指标，最后调用 AA 分组，为其将流量分成合理的实验组和对照组。



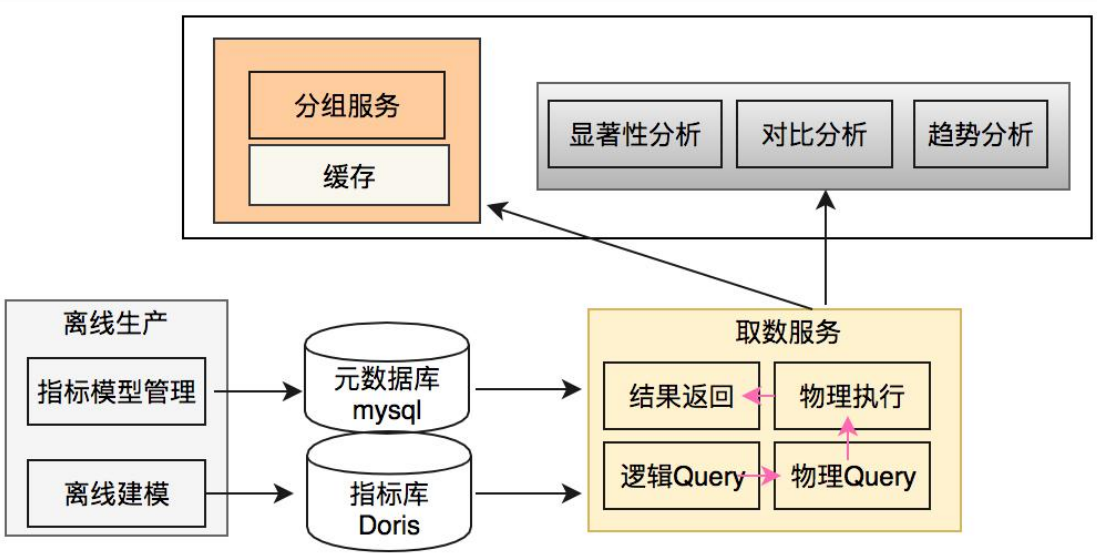
3.2.2.2 A/B 效果评估

A/B 效果评估是实验者在实验后，依据评估报告进行决策的重要依据。 因此，我们在实验后的效果评估环节，效果评估要达成三个目标即权威、灵活性和方便。首先，权威性体现在用于作出实验结论所依赖的指标都是经过治理、各方达成一致的指标，并且确保数据一致性，最终通过假设检验给出科学的实验结论，帮助实验者作出正确的判断。其次，灵活性主要体现在采用列转行的形式，按需自动生成报表告别“烟囱式”的报表开发方式。第三，方便主要体现在不仅可以查看用于说明实验效果的指标，还可以选择查看接入到评估体系里的任意指标；不仅可以查看其实验前后对比以及趋势变化，还可以做到从实验粒度到流量实体粒度的下钻。效果如下图所示：



3.2.2.3 技术实现

不管是实验前的 AA 分组，还是实验后的效果评估，我们要解决的一个核心问题就是如何灵活地“取数”，为我们的 AA 分组和 AB 效果分析提供一个灵活稳定的取数服务。因此，我们整个架构的核心就是构建稳定、灵活的取数服务，具体架构如下图所示。离线建模和指标模型管理完成数据和元数据建设，建立权威完备的指标体系；中间的取数服务作为上层各应用服务和指标体系的“桥梁”，为上层各应用服务提供其所依赖的指标。



4. 总结与展望

目前，A/B 测试已成为许多互联网公司评估其新产品策略和方法的“金标准”，在美团配送业务场景下，它被广泛应用于调度策略、定价策略、运力优化、ETA 时间预估等业务场景，为我们的策略迭代制定数据驱动型决策。特别是针对配送场景下这种策略之间相互影响，请求不独立场景下的 A/B 实验，结合配送技术团队的具体实践，跟大家分享了我们目前的解决思路。

最后再补充一点，在 A/B 测试领域，实验的流量规模应该有足够的统计能力，才能确保指标的变化有统计意义的，为了更好地达到这个目标，未来我们将通过辅助工具建设，在实验前，依据实验者所关注的指标以及敏感度给出流量规模的建议，方便实验者在实验前快速地圈定其实验所需的流量。