美团点评

美团点评 2018 技术年货

CODE A BETTER LIFE

序

春节已近，年味渐浓。

又到了我们献上技术年货的时候。

不久前，我们已经给大家分享了技术沙龙大套餐，汇集了过去一年我们线上线下技术沙龙 99位讲师，85个演讲，70+小时 分享。

今天出场的，同样重磅——技术博客全年大合集。

2018年，是美团技术团队官方博客第5个年头， 博客网站 全年独立访问用户累计超过300万，微信公众号（meituantech）的关注数也超过了15万。

由衷地感谢大家一直以来对我们的鼓励和陪伴！

在2019年春节到来之际，我们再次精选了114篇技术干货，制作成一本厚达1200多页的电子书呈送给大家。

这本电子书主要包括前端、后台、系统、算法、测试、运维、工程师成长等7个板块。疑义相与析，大家在阅读中如果发现Bug、问题，欢迎扫描文末二维码，通过微信公众号与我们交流。

也欢迎大家转给有相同兴趣的同事、朋友，一起切磋，共同成长。

最后祝大家，新春快乐，阖家幸福。

目录 – 系统篇

DataMan–美团旅行数据质量监管平台实践

作者: 德晓

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：

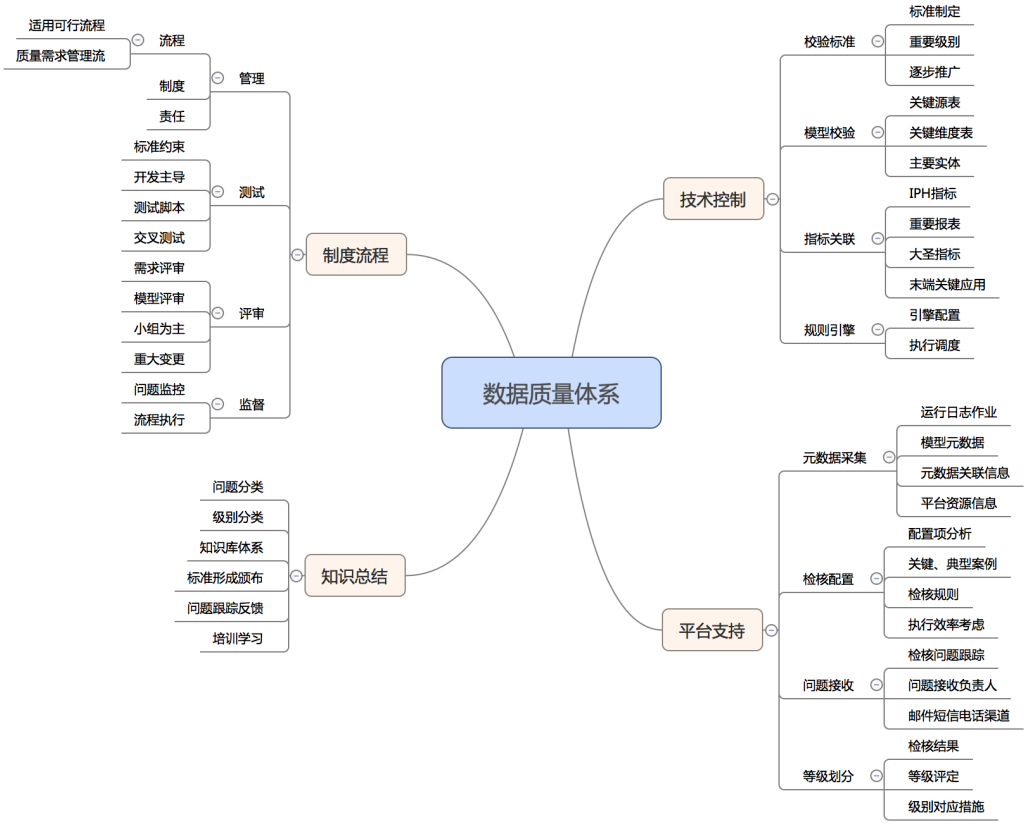


图1 质量监控平台整体框架图

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：

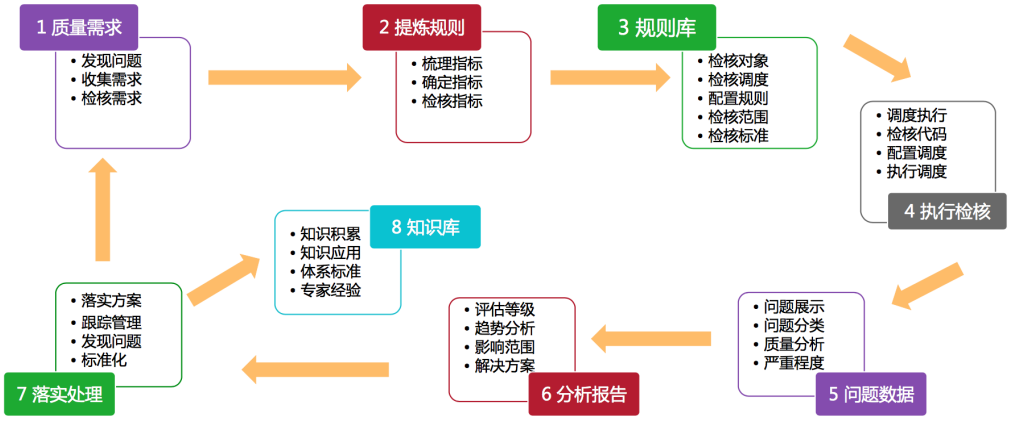


图2 数据质量PDCA流程图

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。

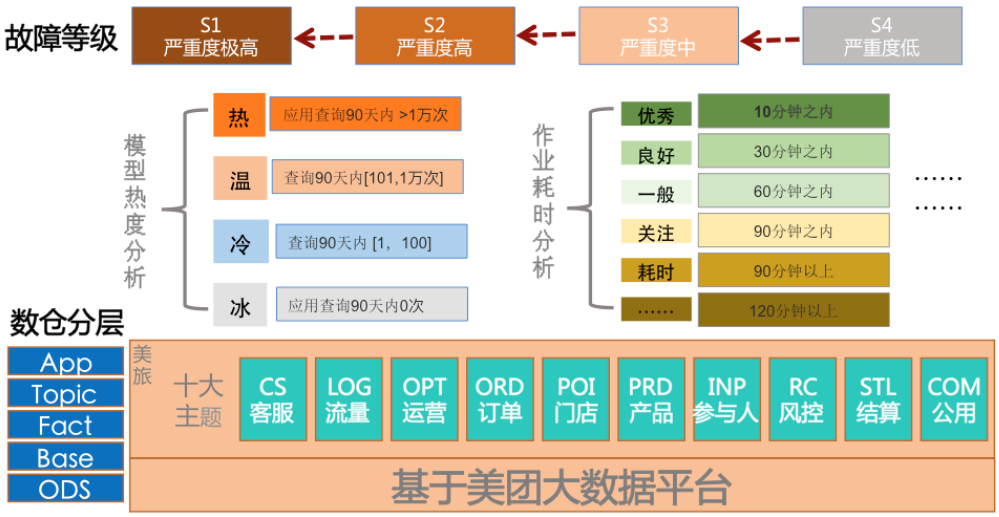


图3 质量检核标准图

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

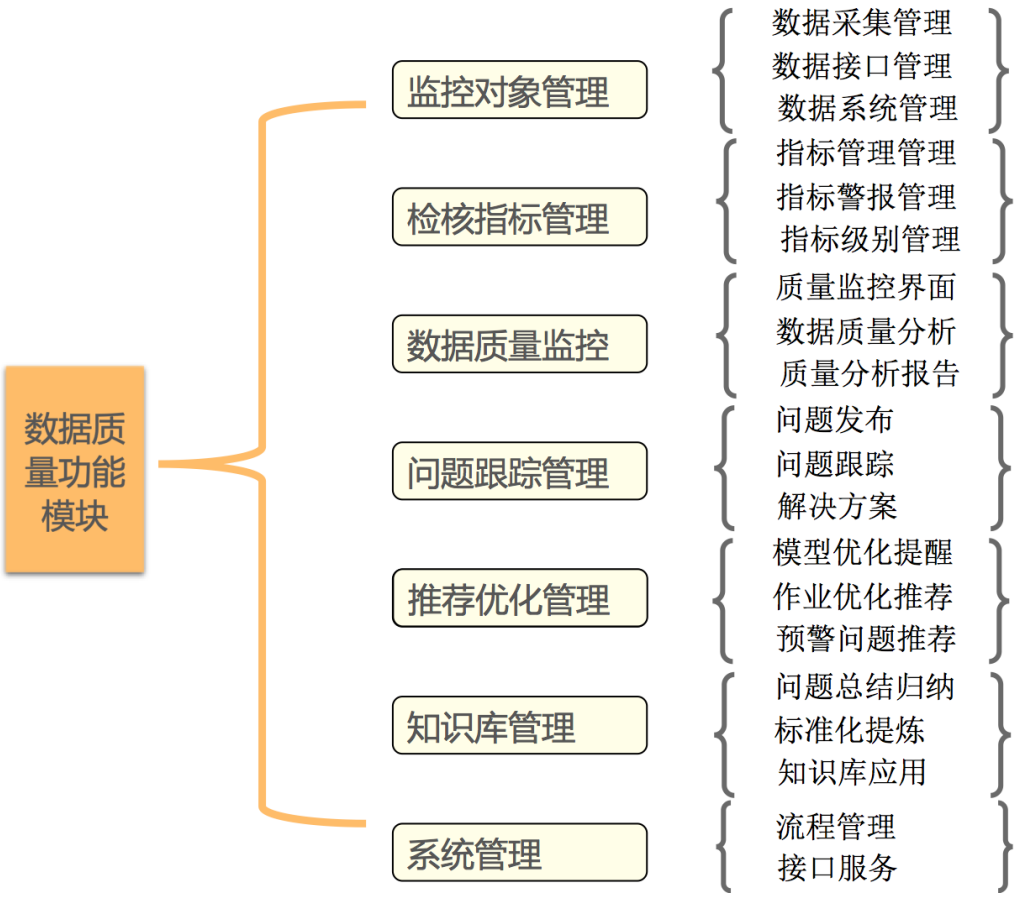


图4 数据质量监管功能图

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：

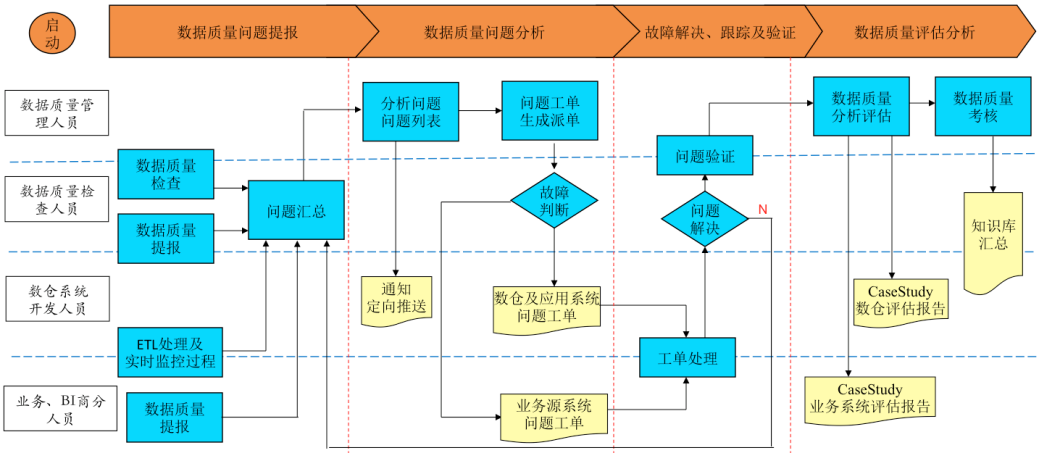


图5 数据质量流程图

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：

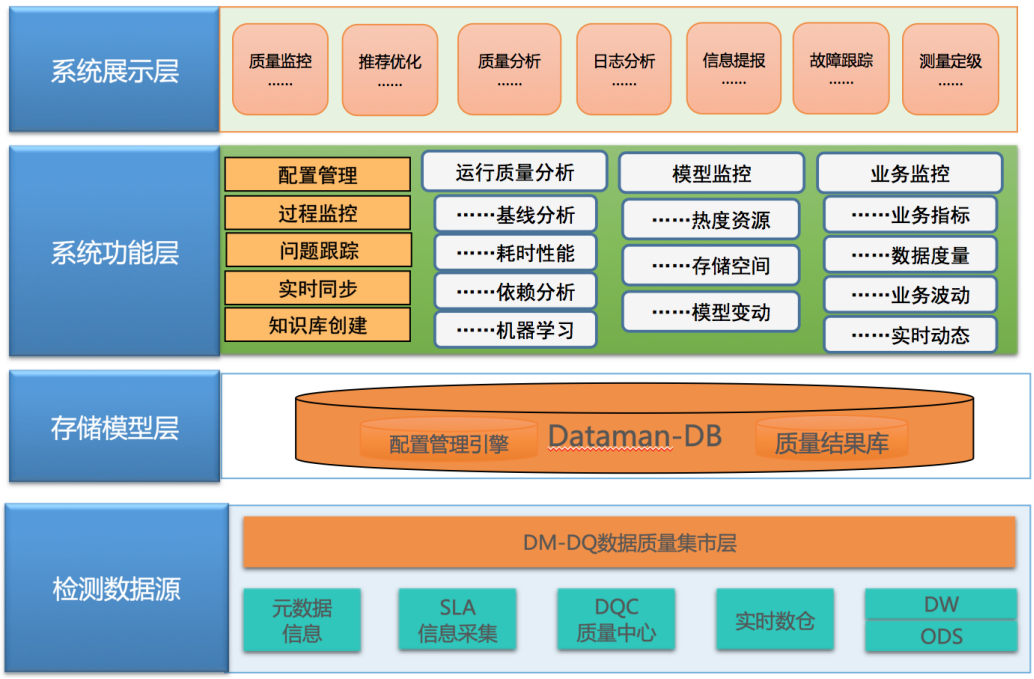


图6 质量监管DataMan总体架构图

- **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

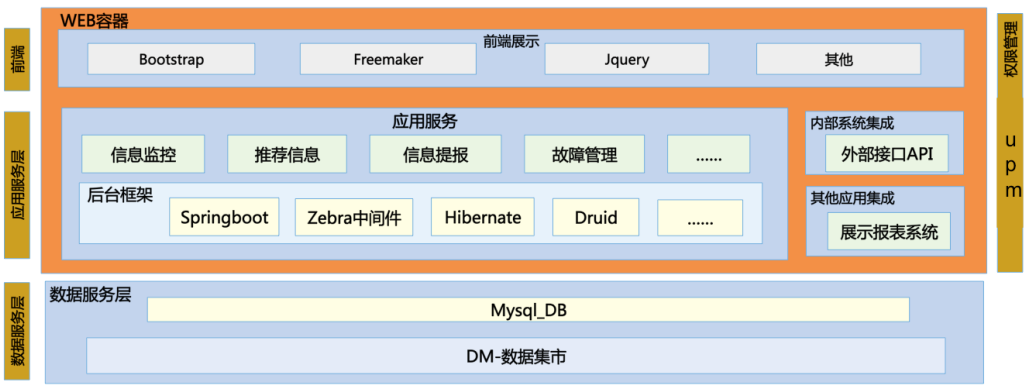


图7 技术架构图

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：

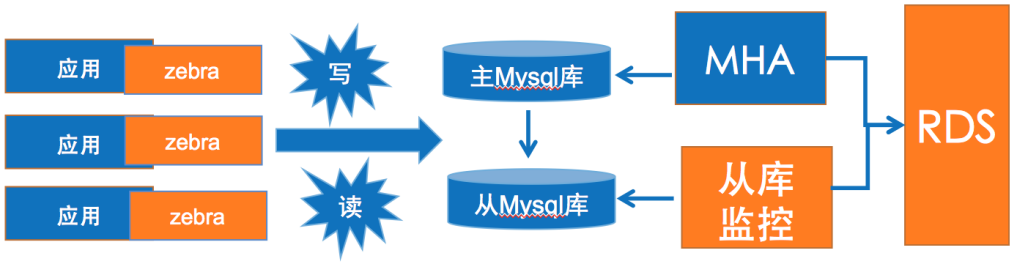


图8 Zebra架构图

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：

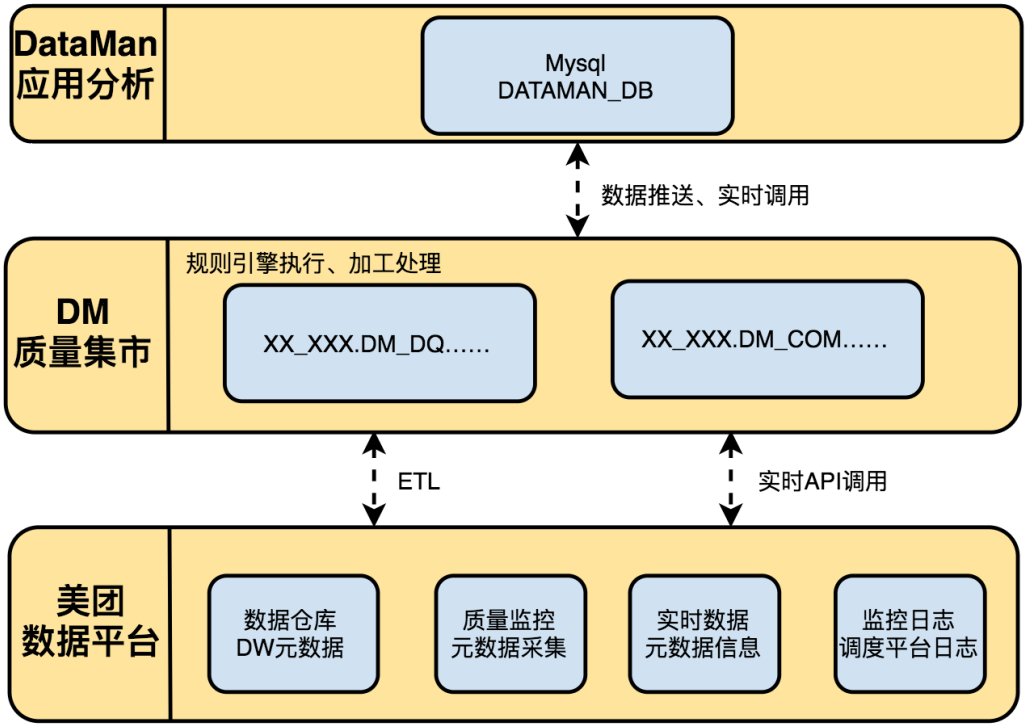


图9 数据流向层级图

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：

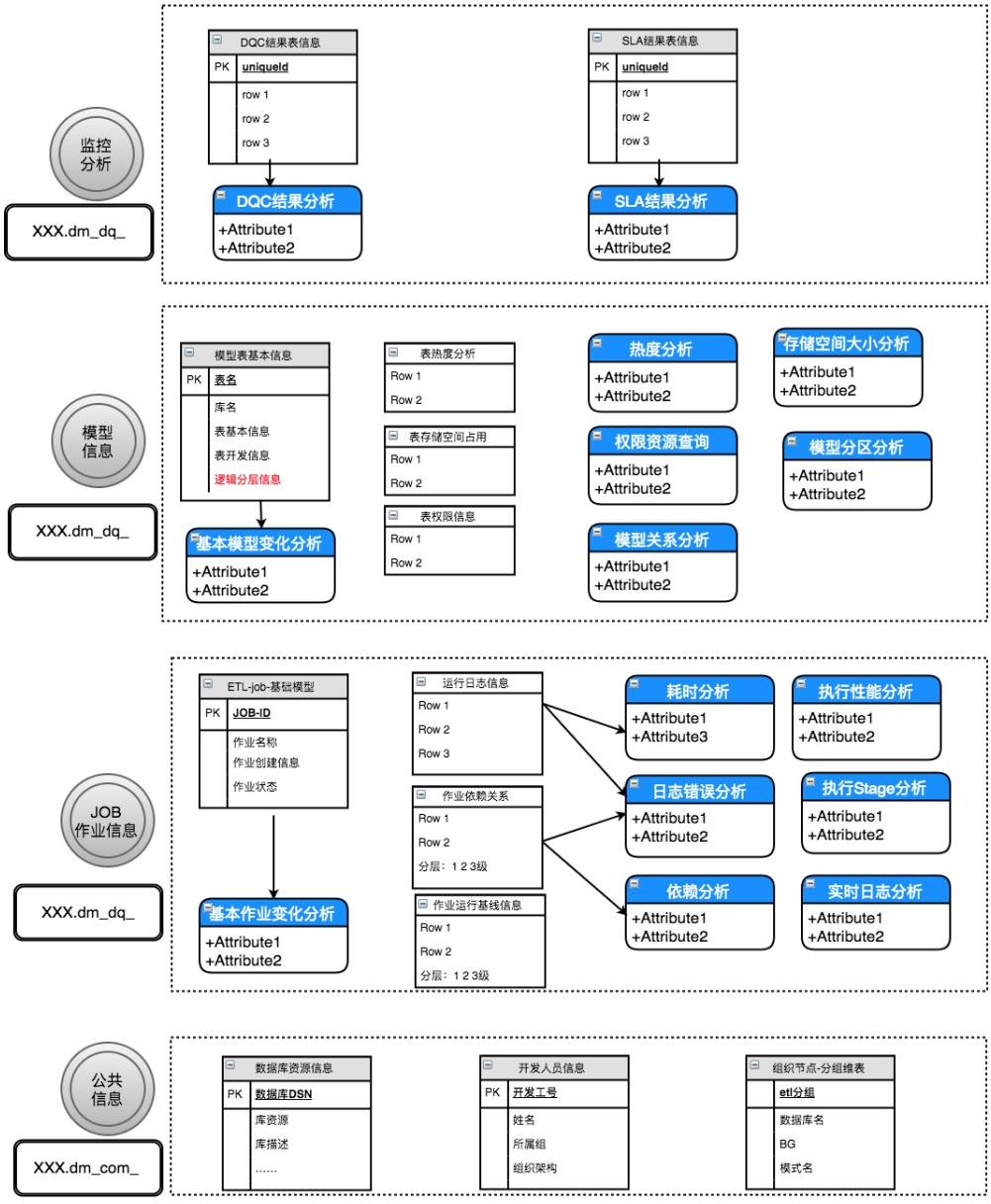


图10 数据质量集市模型图

- 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：

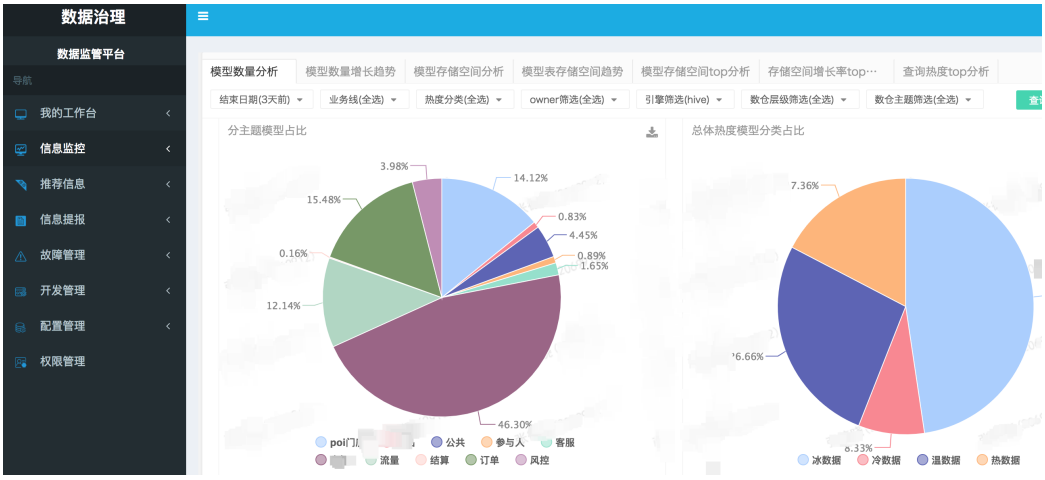


图11 系统效果图

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：

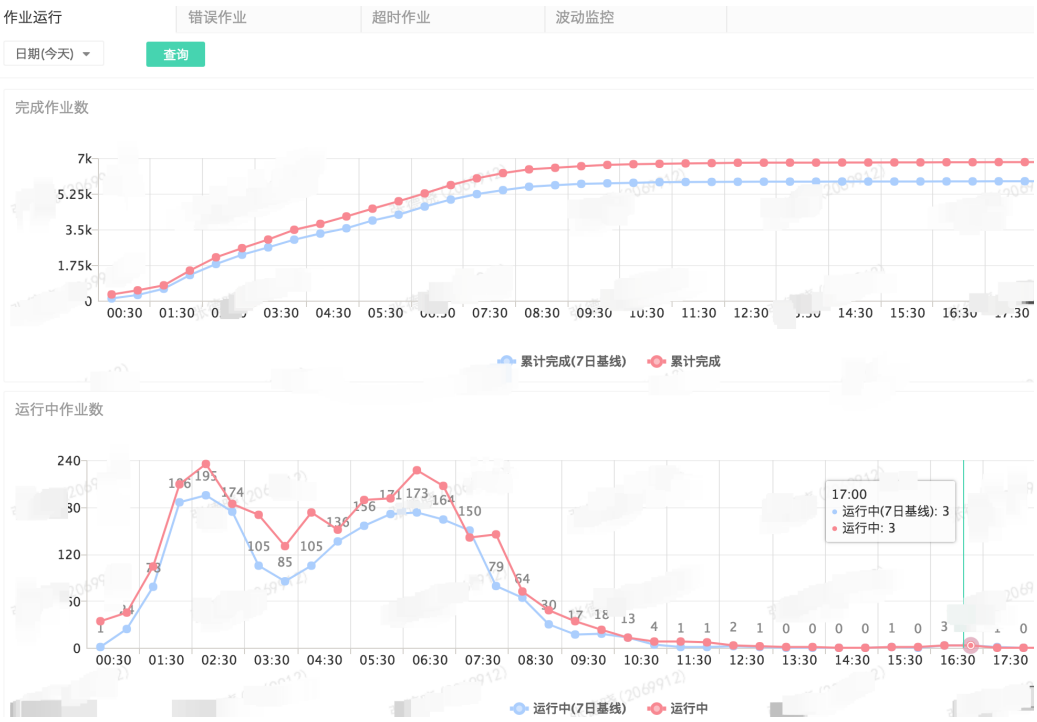


图12 实时作业运行监控图

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。

招聘

最后插播一个招聘广告，我们是一群擅长大数据领域数据建设、数仓建设、数据治理及数据BI应用建设的工程师，期待更多能手加入，有兴趣的同学可以发邮件给yangguang09#meituan.com，zhangdexiao#meituan.com。

作者简介

- 德晓，美团点评数仓专家、大数据高级工程师，长期从事数据仓库、数据建模、数据治理、大数据方向系统实践建设等，现为美团点评大交通数据仓库建设负责人。

美团 R 语言数据运营实战

作者: 喻灿 刘强

一、引言

近年来，随着分布式数据处理技术的不断革新，Hive、Spark、Kylin、Impala、Presto 等工具不断推陈出新，对大数据集合的计算和存储成为现实，数据仓库/商业分析部门日益成为各类企业和机构的标配。在这种背景下，是否能探索和挖掘数据价值，具备精细化数据运营的能力，就成为判定一个数据团队成功与否的关键。

在数据从后台走向前台的过程中，数据展示是最后一步关键环节。与冰冷的表格展示相比，将数据转化成图表并进行适当的内容组织，往往能更快速、更直观的传递信息，进而更好的提供决策支持。从结构化数据到最终的展示，需要通过一系列的探索和分析过程去完成产品思路的沉淀，这个过程也伴随着大量的数据二次处理。

上述这些场合 R 语言有着独特的优势。本文将基于美团到店餐饮技术部的精细化数据运营实践，介绍 R 在数据分析与可视化方面的工程能力，希望能够抛砖引玉，也欢迎业界同行给我们提供更多的建议。

二、数据运营产品分类与 R 的优势

2.1 数据运营产品分类

在企业数据运营过程中，考虑使用场景、产品特点、实施角色以及可利用的工具，大致可以将数据运营需求分为四类，如下表所示，**数据运营需求分类：**

产品	应用场景	产品特点	实施角色	工具
分析报告	对模式不固定的数据进行探索、组织与解释，形成一次性数据分析报告并提供决策支持	基于人对数据的解读；需求发散	数据分析师、数据工程师	Excel、SQL、R、Tableau 等
报表型产品	通过拖拽式或简单代码方式进行开发，对模式固定的数据组装和报表展现	开发效率高，开发门槛低；报表表达能力差	数据分析师	报表工具
定制式分析型产品	对固定模式的数据和分析方法，形成可重复式的数据分析产品并提供决策支持	开发效率较高，支持对数据的深度应用，开发过程可复用、可扩展，对有一定编程能力的开发者开发门槛较低；产品交互能力较弱	数据分析师、数据工程师	Python、R、Tableau 等

定制式展示型产品	对固定模式的数据进行产品的高度定制，通过强化交互和用户体验，满足个性化的数据展示需求	展现样式丰富、交互能力强；仅适合有前端能力的开发者，开发效率较低，数据二次处理能力较差	前端工程师	ECharts、Highcharts 等

2.2 R 在数据运营上的优势

如上节所述，在精细化数据运营过程中，经常需要使用高度定制的数据处理、可视化、分析等手段，这些过程 Excel、Tableau、企业级报表工具都无法面面俱到，而恰好是 R 的强项。一般来说，R 具备的如下特征，让其有了“数据分析领域的瑞士军刀”的名号：

- 免费、开源、可扩展：截至到 2018-08-02，“The CRAN package repository features 12858 available packages.🔗 ”，CRAN 上的软件包涉及贝叶斯分析、运筹学、金融、基因分析、遗传学等方方面面，并在持续新增和迭代。
- 可编程：R 本身是一门解释型语言，可以通过代码控制执行过程，并能通过 rPython、rJava 等软件包实现和 Python、Java 语言的互相调用。
- 强大的数据操控能力：
  - 数据源接入：通过 RMySQL、SparkR、elastic 等软件包，可以实现从 MySQL、Spark、Elasticsearch 等外部数据引擎获取数据。
  - 数据处理：内置 vector、list、matrix、data.frame 等数据结构，并能通过 sqldf、tidyr、dplyr、reshape2 等软件包实现对数据的二次加工。
  - 数据可视化：ggplot2、plotly、dygraph 等可视化包可以实现高度定制化的图表渲染。
  - 数据分析与挖掘：R 本身是一门由统计学家发起的面向统计分析的语言，通过自行编程实现或者第三方软件包调用，可以轻松实现线性回归、方差分析、主成分分析等分析与挖掘功能。
- 初具雏形的服务框架：
  - Web 编程框架：例如不精通前端和系统开发的同学，通过 shiny 软件包开发自己的数据应用。
  - 服务化能力：例如通过 rserve 包，可以实现 R 和其他语言通信的 C/S 架构服务。

对于以数据为中心的应用来说，Python 和 R 都是不错的选择，两门语言在发展过程中也互有借鉴。“越接近统计研究与数据分析，越倾向 R；越接近工程开发工程环境的人，越倾向 Python”，Python 是一个全能型“运动员”，R 则更像是一个统计分析领域的“剑客”，“Python 并未建立起一个能与 CRAN 媲美的巨大的代码库，R 在这方面具有绝对领先优势。统计学并不是 Python 的核心使命”。各技术网站上有大量“Python VS R ”的讨论，感兴趣的读者可以自行了解和作出选择。

三、R 的数据处理、可视化、可重复性数据分析能力

对于具备编程能力的分析师或者具备分析能力的开发人员来说，在进行一系列长期的数据分析工程时，使用 R 既可以满足“一次开发，终身受用”，又可以满足“调整灵活，图形丰富”的要求。下文将分别介绍 R 的数据处理能力、可视化能力和可重复性数据分析能力。

3.1 数据处理

在企业级数据系统中，数据清洗、计算和整合工作会通过数据仓库、Hive、Spark、Kylin 等工具完成。对于数据运营项目，虽然 R 操作的是结果数据集，但也不能避免需要在查询层进行二次数据处理。

在数据查询层，R 生态现成就存在众多的组件支持，例如可以通过 RMySQL 包进行 MySQL 库表的查询，可以使用 Elastic 包对 Elasticsearch 索引文档进行搜索。对于 Kylin 等新技术，在 R 生态的组件支持没有跟上时，可以通过使用 Python、Java 等系统语言进行查询接口封装，在 R 内部使用 rPython、rJava 组件进行第三方查询接口调用。通过查询组件获取的数据一般以 data.frame、list 等类型对象存在。

另外 R 本身也拥有比较完备的二次数据处理能力。例如可以通过 sqldf 使用 sql 对 data.frame 对象进行数据处理，可以使用 reshape2 进行宽格式和窄格式的转化，可以使用 stringr 完成各种字符串处理，其他如排序、分组处理、缺失值填充等功能，也都具备完善的语言本身和生态的支持。

3.2 数据可视化

数据可视化是数据探索过程和结果呈现的关键环节，而“R is a free software environment for statistical computing and graphics.”，绘图（可视化）系统也是 R 的最大优势之一。

目前 R 主流支持的有三套可视化系统：

- 内置系统：包括有 base、grid 和 lattice 三个内置发行包，支持以相对比较朴素的方式完成图形绘制。
- ggplot2：由 RStudio 的首席科学家 Hadley Wickham 开发，ggplot2 通过一套图形语法支持，支持通过图层叠加以组合的方式支持高度定制的可视化。这一理念也逐步影响了包括 Plotly、阿里 AntV 等国内外数据可视化解决方案。截至到 2018-08-02，CRAN 已经落地了 40 个 ggplot2 扩展包，参考链接。
- htmlwidgets for R：这一系统是在 RStudio 支持下于 2016 年开始逐步发展壮大，提供基于 JavaScript 可视化的 R 接口。htmlwidgets for R 作为前端可视化（for 前端工程师）和数据分析可视化（for 数据工程师）的桥梁，发挥了两套技术领域之间的组合优势。截至到 2018-08-02，经过两年多的发展，目前 CRAN 上已经有 101 个基于 htmlwidgets 开发的第三方包，参考链接。

实际数据运营分析过程中，可以固化常规的图表展现和可视化分析过程，实现代码复用，提高开发效率。下图是美团到店餐饮技术部数据团队积累的部分可视化组件示例：

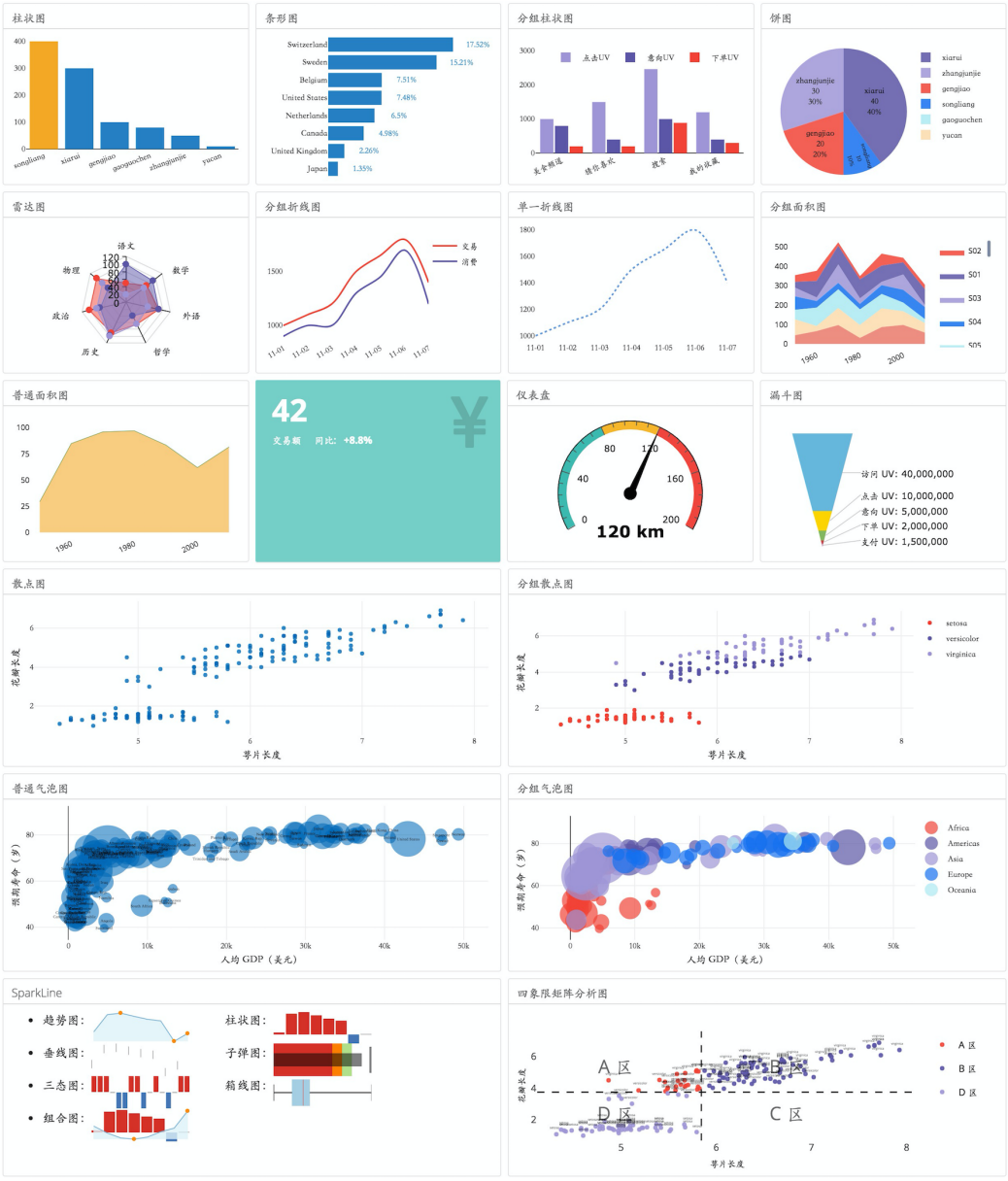


图1 可视化组件示例

基于可视化组件库，一个可视化过程只需要一行代码即可完成，能极大提升开发效率。上图中最后的四象限矩阵分析示例图的代码如下：

```
vis_4quadrant(iris, 'Sepal.Length', 'Petal.Length', label = 'Species', tooltip = 'tooltip', title = '', xtitle = '萼片长度', ytitle = '花瓣长度', pointSize = 1, annotationSize = 1)
```

兹再附四象限矩阵分析可视化组件的函数声明：

```
vis_4quadrant <- function(df, x, y,
  label = '', tooltip = '', title = '', xtitle = '', ytitle = '',
  showLegend = T, jitter = T, centerType = 'mean',
  pointShape = 19, pointSize = 5, pointColors = collocatcolors2,
  lineSize = 0.4, lineType = 'dashed', lineColor = 'black',
  annotationFace = 'sans serif', annotationSize = 5, annotationColor = 'black', annotationDeviationRatio = 15,
  gridAnnotationFace = 'sans serif', gridAnnotationSize = 6, gridAnnotationColor = 'black', gridAnnotationAlpha = 0.6,
  titleFace = 'sans serif', titleSize = 12, titleColor = 'black',
  xyTitleFace = 'sans serif', xyTitleSize = 8, xyTitleColor = 'black',
  gridDesc = c('A 区', 'B 区', 'C 区', 'D 区'), dataMissingInfo = '数据不完整', renderType = 'widget') {

  # 绘制分组散点图
  #
  # Args:
  #   df: 数据框；必要字段；需要进行图形绘制的数据，至少应该有三列
```

```
  #   x: 字符串；必要字段；映射到 X 轴的列名，对应 df 的某一列，此列必须是数值类型或日期类型
  #   y: 字符串；必要字段；映射到 Y 轴的列名，对应 df 的某一列
  #   label: 字符串；映射到点上的文字注释
  #   tooltip: 字符串；映射到点上的悬浮信息
  #   title: 字符串；标题
  #   xtitle: 字符串；X 轴标题
  #   ytitle: 字符串；Y 轴标题
  #   showLegend: bool；定义分区图例是否展示
  #   jitter: bool；定义是否扰动
  #   centerType: 字符串；定义中心点类型，mean 代表平均值，median 代表中位数
  #   pointShape: 整形；定义点型
  #   pointSize: 数值；定义点大小
  #   lineSize: 数值；定义线宽
  #   lineType: 字符串；定义线型
  #   lineColor: 字符串；定义线色
  #   annotationFace: 字符串；定义注释字体
  #   annotationSize: 数值；定义注释字体大小
  #   annotationColor: 字符串；定义注释字体颜色
  #   annotationDeviationRatio: 数值；定义注释文本向上偏移系数
  #   gridAnnotationFace: 字符串；定义网格注释字体
  #   gridAnnotationSize: 数值；定义网格注释字体大小
  #   gridAnnotationColor: 字符串；定义网格注释字体颜色
  #   gridAnnotationAlpha: 数值；定义网格注释文本透明度
  #   titleFace: 字符串；定义标题字体
  #   titleSize: 数值；定义标题字体大小
  #   titleColor: 字符串；定义标题字体颜色
  #   xyTitleFace: 字符串；定义 X、Y 轴标题字体
  #   xyTitleSize: 数值；定义 X、Y 轴标题字体大小
  #   xyTitleColor: 字符串；定义 X、Y 轴标题字体颜色
  #   gridDesc: 长度为 4 的字符串向量
  #   dataMissingInfo: 字符串；数据问题提示文本
  #   renderType: 字符串；定义渲染结果类型，widget 对应 htmlwidget 组件，html 对应 html 内容

  #   代码实现略
}
```

3.3 可重复性数据分析

数据运营分析往往是一个重复性的、重人工参与的过程，最终会落地一套数据分析框架，这套数据分析框架适配具体的数据，用于支持企业数据决策。

RStudio 通过 rmarkdown + knitr 的方式提供了一套基于文学编程的数据分析报告产出方案，开发者可以将 R 代码嵌入 Markdown 文档中执行并得到渲染结果（渲染结果可以是 HTML、PDF、Word 文档格式），实际数据分析过程中，开发者最终能形成一套数据分析模版，每次适配不同的数据，就能产出一份新的数据分析报告。

rmarkdown 本身具备简单的页面布局能力并可以使用 flexdashboard 进行扩展，因此这套方案不仅能实现重复性分析过程，还能实现分析结果的高度定制化展示，可以使用 HTML、CSS、JavaScript 前端三大件对数据分析报告进行展示和交互的细节调整。最终实现人力的节省和数据分析结果的快速、高效产出。

四、R 服务化改造

4.1 R 服务化框架

R 本身既是一门语言、也是一个跨平台的操作环境，具备强大的数据处理、数据分析、和数据可视化能力。除了在个人电脑的 Windows/MacOS 环境中上充当个人统计分析工具外，也可以运行在 Linux 服务环境中，因此可以将 R 作为分析展现引擎，外围通过 Java 等系统开发语言完成缓存、安全检查、权限控制等功能，开发企业报表系统或数据分析（挖掘）框架，而不仅仅只是将 R 作为一个桌面软件。

企业报表系统或数据分析（挖掘）框架设计方案如下图所示：

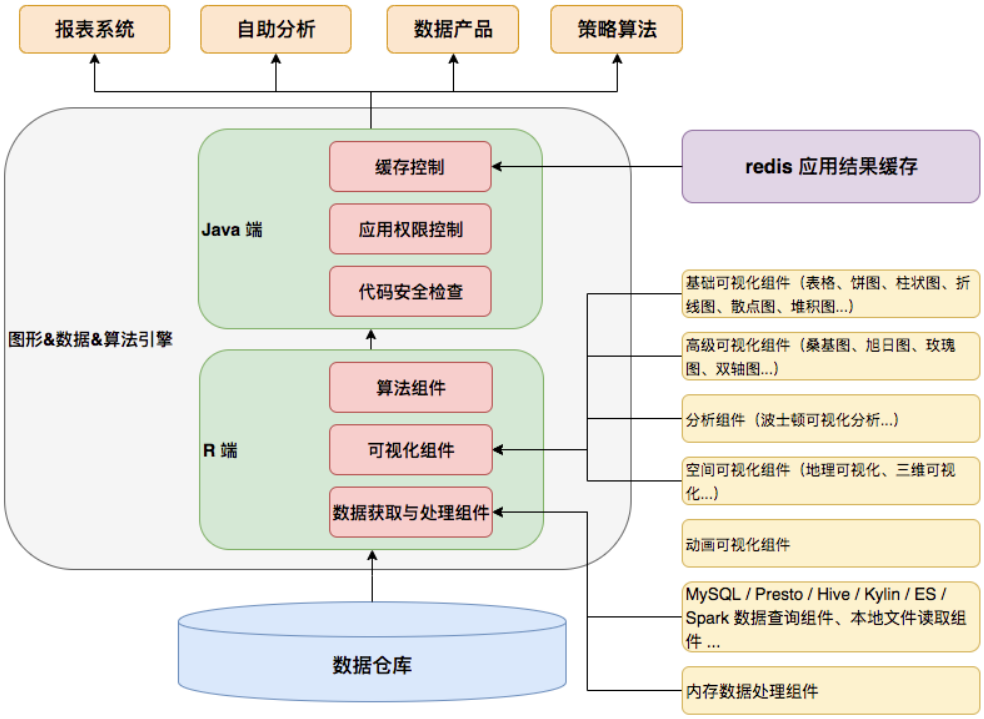


图2 R 服务化框架

4.2 foreach + doParallel 多核并行方案

作为一门统计学家开发的解释性语言，R 运行的是 CPU 单核上的单线程程序、并且需要将全部数据加载到内存进行处理，因此和 Java、Python 等系统语言相比，计算性能是 R 的软肋。对于大数据集合的计算场景，需要尽量将数据计算部分通过 Hive、Kylin 等分布式计算引擎完成，尽量让 R 只处理结果数据集；另外也可以通过 doParallel + foreach 方案，通过多核并行提升计算效率，代码示例如下：

```
library(doParallel)
library(foreach)
registerDoParallel(cores = detectCores())

vis_process1  <- function() {
    # 可视化过程1 ...
}
vis_process2  <- function() {
    # 可视化过程2 ...
}
data_process1 <- function() {
    # 数据处理过程1 ...
}
data_process2 <- function() {
    # 数据处理过程2 ...
}

processes <- c('vis_process1', 'vis_process2', 'data_process1', 'data_process2')
process_res <- foreach(i = 1:length(process), .packages = c('magrittr')) %dopar% {
    do.call(processes[i], list())
}

vis_process1_res  <- process_res[[1]]
vis_process2_res  <- process_res[[2]]
data_process1_res <- process_res[[3]]
data_process2_res <- process_res[[4]]
```

4.3 图形化数据报告渲染性能

在数据分析过程中，R 最重要的是充当图形引擎的角色，因此有必要了解其图形渲染性能。针对主流的基于 rmarkdown + flexdashboard 的数据分析报告渲染方案，其性能测试结果如下：

系统环境： * 4 核 CPU，8 G 内存，2.20GHz 主频。 * Linux version 3.10.0–123.el7.x86_64。

测试方法：

- 测试在不同并发度下、不同复杂度的渲染模式下，重复渲染 100 次的耗时。

测试结果（数据分析报告渲染性能测试）：

渲染模式	并发度 1	并发度 2	并发度 3	并发度 4	并发度 5	并发度 6
rmarkdown + flexdashboard	1m14.087s	0m39.192s	0m28.299s	0m20.795s	0m21.471s	0m19.755s
rmarkdown + flexdashboard + dygraphs	1m48.771s	0m52.716s	0m39.051s	0m27.012s	0m30.224s	0m28.948s
rmarkdown + flexdashboard + ggplot2	2m6.840s	1m1.529s	0m42.351s	0m31.596s	0m35.546s	0m34.992s
rmarkdown + flexdashboard + ggplot2 + dygraph	2m30.586s	1m16.696s	0m51.277s	0m40.651s	0m41.406s	0m41.288s

根据测试结果可知：

- 单应用平均渲染时长在 0.74s 以上，具体的渲染时长视计算复杂度而定（可以通过上节介绍的“foreach + doParallel 多核并行方案 ”加快处理过程）。根据经验，大部分应用能在秒级完成渲染。
- 由于单核单线程模式所限，当并发请求超过 CPU 核数时，渲染吞吐量并不会相应提升。需要根据实际业务场景匹配对应的服务端机器配置，并在请求转发时设置并发执行上限。对于内部运营性质的数据系统，单台 4 核 8 G 机器基本能满足要求。

五、R 在美团数据产品中的落地实践

美团到店餐饮数据团队从 2015 年开始逐步将 R 作为数据产品的辅助开发语言，截至 2018 年 8 月，已经成功应用在面向管理层的日周月数据报告、面向数据仓库治理的分析工具、面向内部运营与分析师的数据 Dashboard、面向大客户销售的品牌商家数据分析系统等多个项目中。目前所有的面向部门内部的定制式分析型产品，都首选使用 R 进行开发。

另外我们也在逐步沉淀 R 可视化与分析组件、开发基于 R 引擎的配置化 BI 产品开发框架，以期进一步降低 R 的使用门槛、提升 R 的普及范围。

下图是美团到店餐饮数据团队在数据治理过程中，使用 R 开发的 ETL 间依赖关系可视化工具：

图3 ETL 间依赖关系可视化工具

六、结语

综上所述，R 可以在企业数据运营实践中扮演关键技术杠杆，但作为一门面向统计分析的领域语言，在很长一段时间，R 的发展主要由统计学家驱动。随着近年的数据爆发式增长与应用浪潮，R 得到越来越多工业界的支持，譬如微软收购基于 R 的企业级数据解决方案提供商 Revolution Analytics、在 SQL Server 2016 集成 R、并从 Visual Studio 2015 开始正式通过 RTVS 集成了 R 开发环境，一系列事件标志着微软在数据分析领域对 R 的高度重视。

在国内，由 统计之都 发起的 中国 R 会议，从 2008 年起已举办了 11 届，推动了 R 用户在国内的发展壮大。截至 2018 年 8 月，美团的 R 开发者大致在 200 人左右。但相比 Java/Python 等系统语言，R 的用户和应用面仍相对狭窄。作者撰写本文的目的，也是希望给从事数据相关工作的同学们一个新的、更具优势的可选项。

关于作者

- 喻灿，美团到店餐饮技术部数据系统与数据产品团队负责人，2015 年加入美团，长期从事数据平台、数据仓库、数据应用方面的开发工作。从 2013 年开始接触 R，在利用 R 快速满足业务需求和节省研发成本上，有一些心得和产出。同时也在美团研发和商业分析团队中积极推动 R 的发展。

招聘信息

对数据工程和将数据通过服务业务释放价值感兴趣的同学，可以发送简历到yucan@meituan.com。我们在数据仓库、数据治理、数据产品开发框架、数据可视化、面向销售和商家侧的数据型创新产品层面，都有很多未知但有意义的领域等你来开拓。

聊聊MyBatis缓存机制

作者: 凯伦

前言

MyBatis是常见的Java数据库访问层框架。在日常工作中，开发人员多数情况下是使用MyBatis的默认缓存配置，但是MyBatis缓存机制有一些不足之处，在使用中容易引起脏数据，形成一些潜在的隐患。个人在业务开发中也处理过一些由于MyBatis缓存引发的开发问题，带着个人的兴趣，希望从应用及源码的角度为读者梳理MyBatis缓存机制。

本次分析中涉及到的代码和数据库表均放在GitHub上，地址： mybatis-cache-demo 。

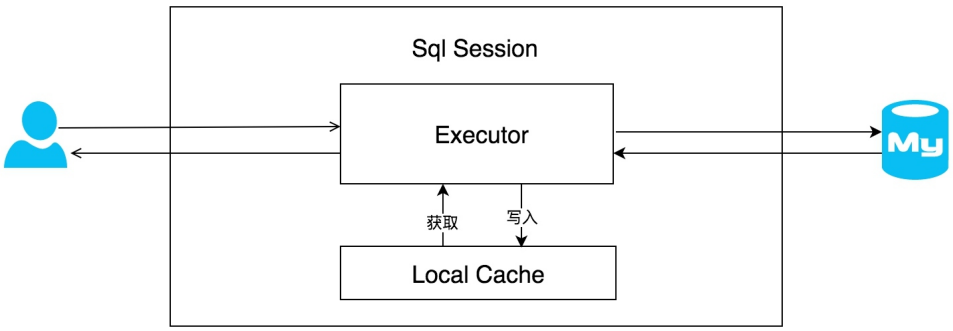
目录

本文按照以下顺序展开。

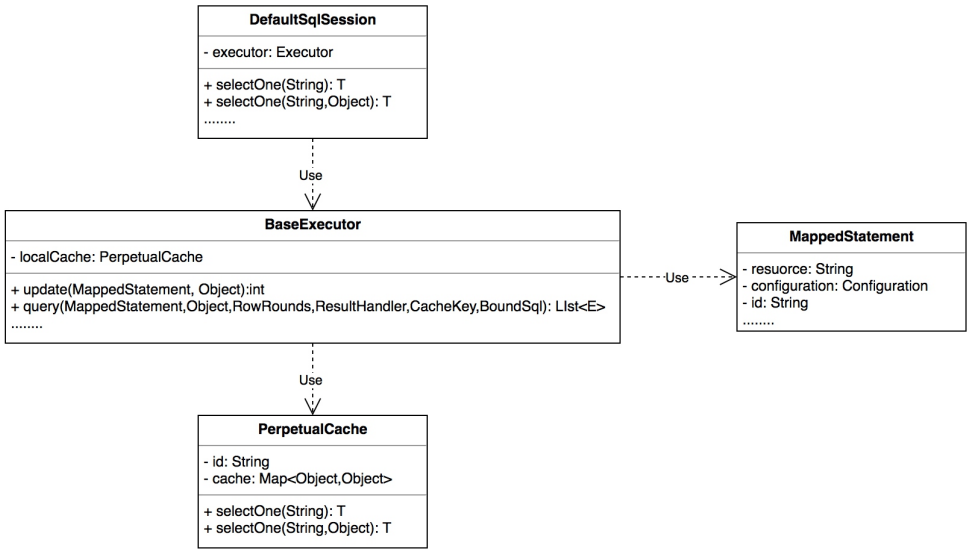
一级缓存

一级缓存介绍

在应用运行过程中，我们有可能在一次数据库会话中，执行多次查询条件完全相同的SQL，MyBatis提供了一级缓存的方案优化这部分场景，如果是相同的SQL语句，会优先命中一级缓存，避免直接对数据库进行查询，提高性能。具体执行过程如下图所示。



每个SqlSession中持有了Executor，每个Executor中有一个LocalCache。当用户发起查询时，MyBatis根据当前执行的语句生成 `MappedStatement` ，在Local Cache进行查询，如果缓存命中的话，直接返回结果给用户，如果缓存没有命中的话，查询数据库，结果写入 `Local Cache` ，最后返回结果给用户。具体实现类的类关系图如下图所示。



一级缓存配置

我们来看看如何使用MyBatis一级缓存。开发者只需在MyBatis的配置文件中，添加如下语句，就可以使用一级缓存。共有两个选项， `SESSION` 或者 `STATEMENT` ，默认是 `SESSION` 级别，即在一个MyBatis会话中执行的所有语句，都会共享这一个缓存。一种是 `STATEMENT` 级别，可以理解为缓存只对当前执行的这一个 `Statement` 有效。

```
<setting name="localCacheScope" value="SESSION"/>
```

一级缓存实验

接下来通过实验，了解MyBatis一级缓存的效果，每个单元测试后都请恢复被修改的数据。

首先是创建示例表student，创建对应的POJO类和增改的方法，具体可以在entity包和mapper包中查看。

```
CREATE TABLE `student` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(200) COLLATE utf8_bin DEFAULT NULL,
  `age` tinyint(3) unsigned DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
```

在以下实验中，id为1的学生名称是凯伦。

实验1

开启一级缓存，范围为会话级别，调用三次 `getStudentById` ，代码如下所示：

```
public void getStudentById() throws Exception {
        SqlSession sqlSession = factory.openSession(true); // 自动提交事务
        StudentMapper studentMapper = sqlSession.getMapper(StudentMapper.class);
        System.out.println(studentMapper.getStudentById(1));
        System.out.println(studentMapper.getStudentById(1));
        System.out.println(studentMapper.getStudentById(1));
    }
```

执行结果：

```
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 凯伦, 25
DEBUG [main] - <==      Total: 1
StudentEntity{id=1, name='凯伦', age=25}
StudentEntity{id=1, name='凯伦', age=25}
StudentEntity{id=1, name='凯伦', age=25}
```

只有第一次真正查询了数据库

我们可以看到，只有第一次真正查询了数据库，后续的查询使用了一级缓存。

实验2

增加了对数据库的修改操作，验证在一次数据库会话中，如果对数据库发生了修改操作，一级缓存是否会失效。

```
@Test
public void addStudent() throws Exception {
        SqlSession sqlSession = factory.openSession(true); // 自动提交事务
        StudentMapper studentMapper = sqlSession.getMapper(StudentMapper.class);
        System.out.println(studentMapper.getStudentById(1));
        System.out.println("增加了" + studentMapper.addStudent(buildStudent()) + "个学生");
        System.out.println(studentMapper.getStudentById(1));
        sqlSession.close();
}
```

执行结果：

```
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 凯伦, 25
DEBUG [main] - <==      Total: 1
StudentEntity{id=1, name='凯伦', age=25}
DEBUG [main] - ==>  Preparing: INSERT INTO student(name,age) VALUES(?, ?)
DEBUG [main] - ==> Parameters: 明明(String), 20(Integer)
DEBUG [main] - <==    Updates: 1
增加了1个学生
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 凯伦, 25
DEBUG [main] - <==      Total: 1
StudentEntity{id=1, name='凯伦', age=25}
```

在插入操作后的select操作，重新查询了数据库

我们可以看到，在修改操作后执行的相同查询，查询了数据库，**一级缓存失效**。

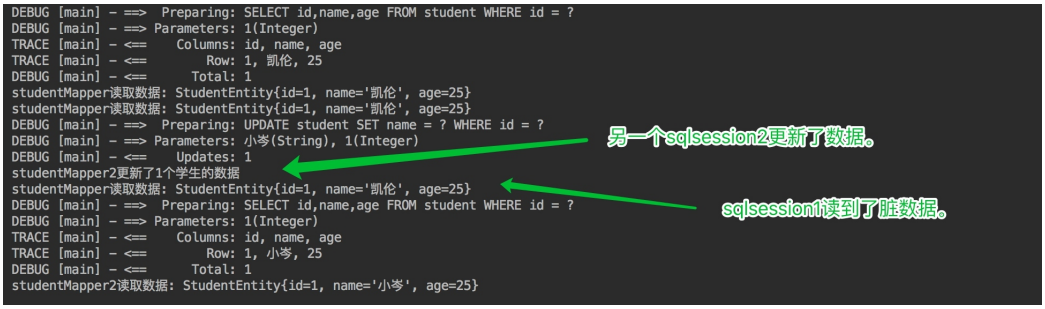
实验3

开启两个 `SqlSession` ，在 `sqlSession1` 中查询数据，使一级缓存生效，在 `sqlSession2` 中更新数据库，验证一级缓存只在数据库会话内部共享。

```
@Test
public void testLocalCacheScope() throws Exception {
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);
```

```
        StudentMapper studentMapper = sqlSession1.getMapper(StudentMapper.class);
        StudentMapper studentMapper2 = sqlSession2.getMapper(StudentMapper.class);

        System.out.println("studentMapper读取数据: " + studentMapper.getStudentById(1));
        System.out.println("studentMapper读取数据: " + studentMapper.getStudentById(1));
        System.out.println("studentMapper2更新了" + studentMapper2.updateStudentName("小岑",1) + "个学生的数据");
        System.out.println("studentMapper读取数据: " + studentMapper.getStudentById(1));
        System.out.println("studentMapper2读取数据: " + studentMapper2.getStudentById(1));
}
```



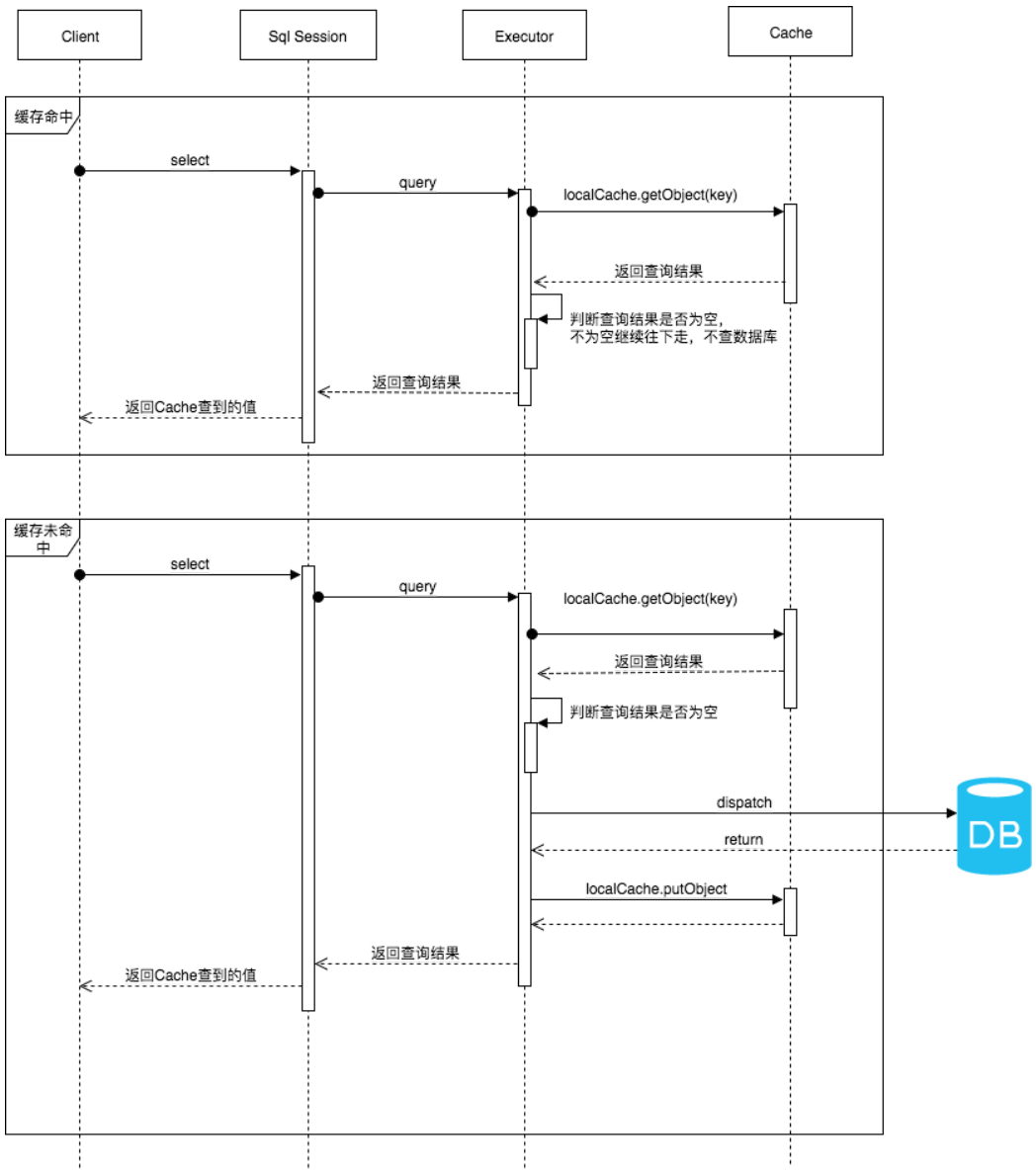
`sqlSession2` 更新了id为1的学生的姓名，从凯伦改为了小岑，但session1之后的查询中，id为1的学生的名字还是凯伦，出现了脏数据，也证明了之前的设想，一级缓存只在数据库会话内部共享。

一级缓存工作流程&源码分析

那么，一级缓存的工作流程是怎样的呢？我们从源码层面来学习一下。

工作流程

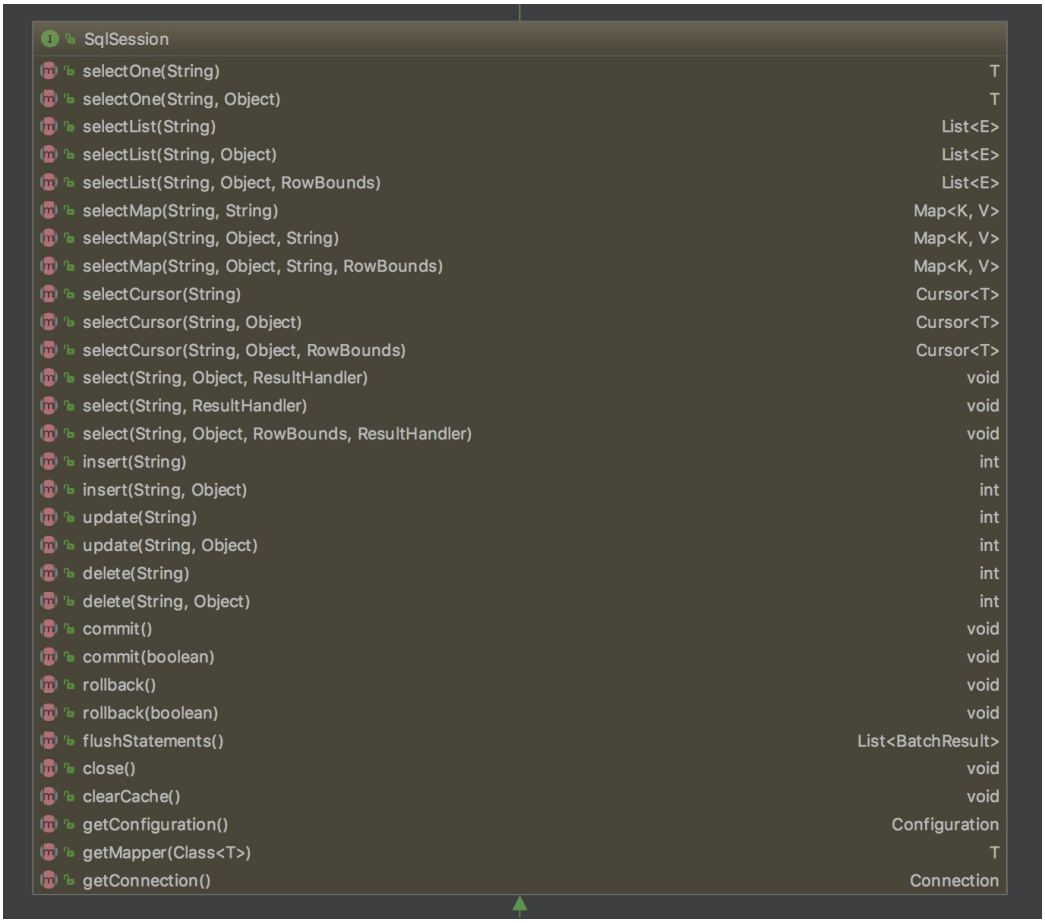
一级缓存执行的时序图，如下图所示。



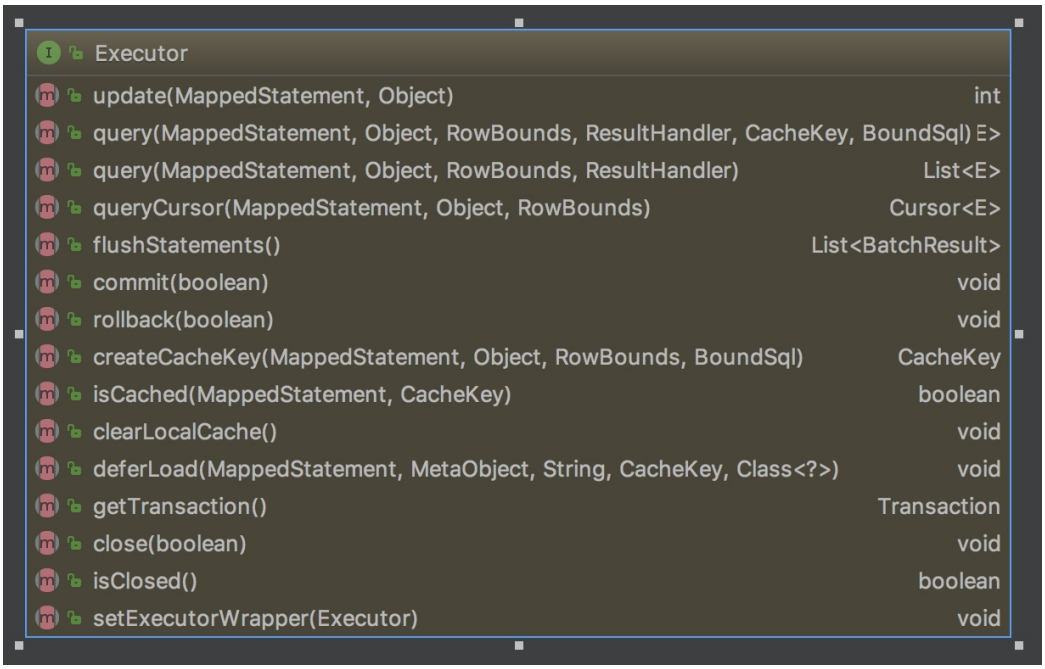
源码分析

接下来将对MyBatis查询相关的核心类和一级缓存的源码进行走读。这对后面学习二级缓存也有帮助。

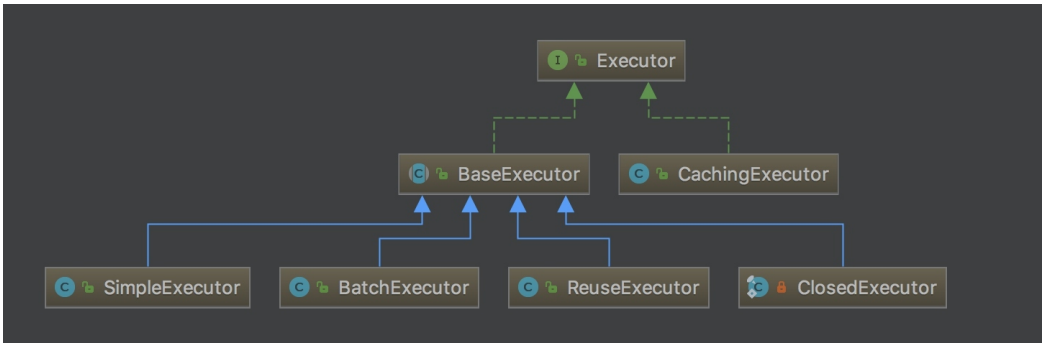
SqlSession： 对外提供了用户和数据库之间交互需要的所有方法，隐藏了底层的细节。默认实现类是 `DefaultSqlSession` 。



Executor： `SqlSession` 向用户提供操作数据库的方法，但和数据库操作有关的职责都会委托给Executor。



如下图所示，Executor有若干个实现类，为Executor赋予了不同的能力，大家可以根据类名，自行学习每个类的基本作用。



在一级缓存的源码分析中，主要学习 `BaseExecutor` 的内部实现。

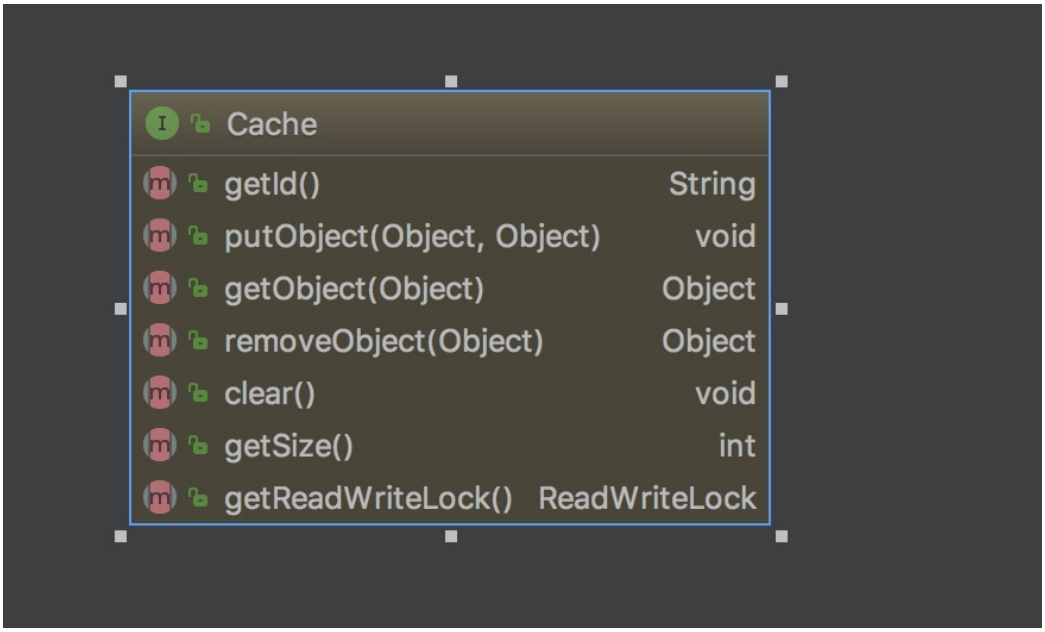
BaseExecutor： `BaseExecutor` 是一个实现了Executor接口的抽象类，定义若干抽象方法，在执行的时候，把具体的操作委托给子类进行执行。

```
protected abstract int doUpdate(MappedStatement ms, Object parameter) throws SQLException;
protected abstract List<BatchResult> doFlushStatements(boolean isRollback) throws SQLException;
protected abstract <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandle
r, BoundSql boundSql) throws SQLException;
protected abstract <E> Cursor<E> doQueryCursor(MappedStatement ms, Object parameter, RowBounds rowBounds, BoundSql boundSql)
throws SQLException;
```

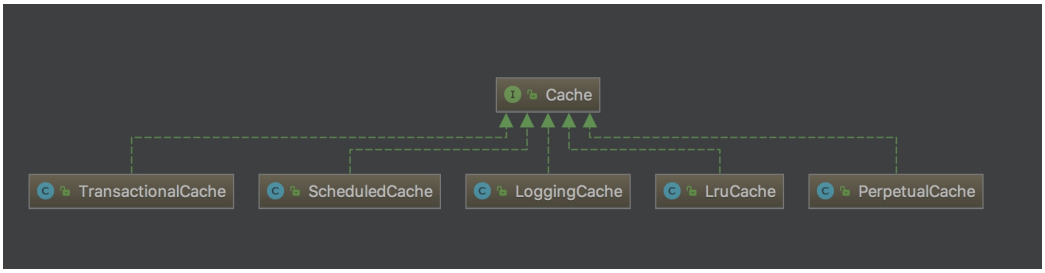
在一级缓存的介绍中提到对 `Local Cache` 的查询和写入是在 `Executor` 内部完成的。在阅读 `BaseExecutor` 的代码后发现 `Local Cache` 是 `BaseExecutor` 内部的一个成员变量，如下代码所示。

```
public abstract class BaseExecutor implements Executor {
protected ConcurrentLinkedQueue<DeferredLoad> deferredLoads;
protected PerpetualCache localCache;
```

Cache： MyBatis中的Cache接口，提供了和缓存相关的最基本的操作，如下图所示：



有若干个实现类，使用装饰器模式互相组装，提供丰富的操控缓存的能力，部分实现类如下图所示：



`BaseExecutor` 成员变量之一的 `PerpetualCache` ，是对Cache接口最基本的实现，其实现非常简单，内部持有HashMap，对一级缓存的操作实则是对HashMap的操作。如下代码所示：

```
public class PerpetualCache implements Cache {
  private String id;
  private Map<Object, Object> cache = new HashMap<Object, Object>();
```

在阅读相关核心类代码后，从源代码层面对一级缓存工作中涉及到的相关代码，出于篇幅的考虑，对源码做适当删减，读者朋友可以结合本文，后续进行更详细的学习。

为执行和数据库的交互，首先需要初始化 `SqlSession` ，通过 `DefaultSqlSessionFactory` 开启 `SqlSession` ：

```
private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level, boolean autoCommit) {
    ............
    final Executor executor = configuration.newExecutor(tx, execType);
    return new DefaultSqlSession(configuration, executor, autoCommit);
}
```

在初始化 `SqlSesion` 时，会使用 `Configuration` 类创建一个全新的 `Executor` ，作为 `DefaultSqlSession` 构造函数的参数，创建Executor代码如下所示：

```
public Executor newExecutor(Transaction transaction, ExecutorType executorType) {
    executorType = executorType == null ? defaultExecutorType : executorType;
    executorType = executorType == null ? ExecutorType.SIMPLE : executorType;
    Executor executor;
    if (ExecutorType.BATCH == executorType) {
      executor = new BatchExecutor(this, transaction);
    } else if (ExecutorType.REUSE == executorType) {
      executor = new ReuseExecutor(this, transaction);
    } else {
      executor = new SimpleExecutor(this, transaction);
    }
    // 尤其可以注意这里，如果二级缓存开关开启的话，是使用CahingExecutor装饰BaseExecutor的子类
    if (cacheEnabled) {
      executor = new CachingExecutor(executor);
    }
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
}
```

`SqlSession` 创建完毕后，根据Statment的不同类型，会进入 `SqlSession` 的不同方法中，如果是 `Select` 语句的话，最后会执行到 `SqlSession` 的 `selectList` ，代码如下所示：

```
@Override
public <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds) {
      MappedStatement ms = configuration.getMappedStatement(statement);
      return executor.query(ms, wrapCollection(parameter), rowBounds, Executor.NO_RESULT_HANDLER);
}
```

`SqlSession` 把具体的查询职责委托给了Executor。如果只开启了一级缓存的话，首先会进入 `BaseExecutor` 的 `query` 方法。代码如下所示：

```
@Override
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler) throws SQLEx
ception {
    BoundSql boundSql = ms.getBoundSql(parameter);
    CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql);
    return query(ms, parameter, rowBounds, resultHandler, key, boundSql);
}
```

在上述代码中，会先根据传入的参数生成CacheKey，进入该方法查看CacheKey是如何生成的，代码如下所示：

```
CacheKey cacheKey = new CacheKey();
cacheKey.update(ms.getId());
cacheKey.update(rowBounds.getOffset());
cacheKey.update(rowBounds.getLimit());
cacheKey.update(boundSql.getSql());
//后面是update了sql中带的参数
cacheKey.update(value);
```

在上述的代码中，将 `MappedStatement` 的Id、SQL的offset、SQL的limit、SQL本身以及SQL中的参数传入了CacheKey这个类，最终构成CacheKey。以下是这个类的内部结构：

```
private static final int DEFAULT_MULTIPLYER = 37;
private static final int DEFAULT_HASHCODE = 17;

private int multiplier;
private int hashcode;
private long checksum;
private int count;
private List<Object> updateList;

public CacheKey() {
    this.hashcode = DEFAULT_HASHCODE;
    this.multiplier = DEFAULT_MULTIPLYER;
    this.count = 0;
    this.updateList = new ArrayList<Object>();
}
```

首先是成员变量和构造函数，有一个初始的 `hachcode` 和乘数，同时维护了一个内部的 `updatelist` 。在 `CacheKey` 的 `update` 方法中，会进行一个 `hashcode` 和 `checksum` 的计算，同时把传入的参数添加进 `updatelist` 中。如下代码所示：

```
public void update(Object object) {
    int baseHashCode = object == null ? 1 : ArrayUtil.hashCode(object);
    count++;
    checksum += baseHashCode;
    baseHashCode *= count;
    hashcode = multiplier * hashcode + baseHashCode;

    updateList.add(object);
}
```

同时重写了 `CacheKey` 的 `equals` 方法，代码如下所示：

```
@Override
public boolean equals(Object object) {
    ............
    for (int i = 0; i < updateList.size(); i++) {
      Object thisObject = updateList.get(i);
      Object thatObject = cacheKey.updateList.get(i);
      if (!ArrayUtil.equals(thisObject, thatObject)) {
        return false;
      }
    }
    return true;
}
```

除去hashcode、checksum和count的比较外，只要updatelist中的元素一一对应相等，那么就可以认为是CacheKey相等。只要两条SQL的下列五个值相同，即可以认为是相同的SQL。

> Statement Id + Offset + Limmit + Sql + Params

BaseExecutor的query方法继续往下走，代码如下所示：

```
list = resultHandler == null ? (List<E>) localCache.getObject(key) : null;
if (list != null) {
    // 这个主要是处理存储过程用的。
    handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
    } else {
    list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key, boundSql);
}
```

如果查不到的话，就从数据库查，在 `queryFromDatabase` 中，会对 `localcache` 进行写入。

在 `query` 方法执行的最后，会判断一级缓存级别是否是 `STATEMENT` 级别，如果是的话，就清空缓存，这也就是 `STATEMENT` 级别的一级缓存无法共享 `localCache` 的原因。代码如下所示：

```
if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
        clearLocalCache();
}
```

在源码分析的最后，我们确认一下，如果是 `insert/delete/update` 方法，缓存就会刷新的原因。

`SqlSession` 的 `insert` 方法和 `delete` 方法，都会统一走 `update` 的流程，代码如下所示：

```
@Override
public int insert(String statement, Object parameter) {
    return update(statement, parameter);
  }
   @Override
  public int delete(String statement) {
    return update(statement, null);
}
```

`update` 方法也是委托给了 `Executor` 执行。 `BaseExecutor` 的执行方法如下所示：

```
@Override
public int update(MappedStatement ms, Object parameter) throws SQLException {
    ErrorContext.instance().resource(ms.getResource()).activity("executing an update").object(ms.getId());
    if (closed) {
      throw new ExecutorException("Executor was closed.");
    }
    clearLocalCache();
    return doUpdate(ms, parameter);
}
```

每次执行 `update` 前都会清空 `localCache` 。

至此，一级缓存的工作流程讲解以及源码分析完毕。

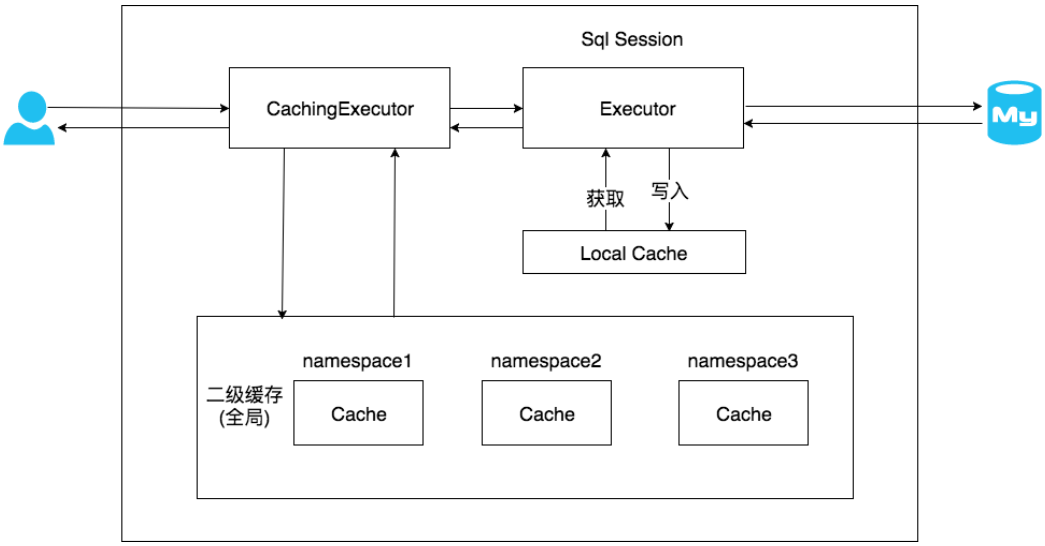
总结

1. MyBatis一级缓存的生命周期和SqlSession一致。
2. MyBatis一级缓存内部设计简单，只是一个没有容量限定的HashMap，在缓存的功能性上有所欠缺。
3. MyBatis的一级缓存最大范围是SqlSession内部，有多个SqlSession或者分布式的环境下，数据库写操作会引起脏数据，建议设定缓存级别为Statement。

二级缓存

二级缓存介绍

在上文中提到的一级缓存中，其最大的共享范围就是一个SqlSession内部，如果多个SqlSession之间需要共享缓存，则需要使用到二级缓存。开启二级缓存后，会使用CachingExecutor装饰Executor，进入一级缓存的查询流程前，先在CachingExecutor进行二级缓存的查询，具体的工作流程如下所示。



二级缓存开启后，同一个namespace下的所有操作语句，都影响着同一个Cache，即二级缓存被多个SqlSession共享，是一个全局的变量。

当开启缓存后，数据的查询执行的流程就是 二级缓存 -> 一级缓存 -> 数据库。

二级缓存配置

要正确的使用二级缓存，需完成如下配置的。

1. 在MyBatis的配置文件中开启二级缓存。

```
<setting name="cacheEnabled" value="true"/>
```

1. 在MyBatis的映射XML中配置cache或者 cache-ref 。

cache标签用于声明这个namespace使用二级缓存，并且可以自定义配置。

```
<cache/>
```

- `type` ： cache使用的类型，默认是 `PerpetualCache` ，这在一级缓存中提到过。
- `eviction` ： 定义回收的策略，常见的有FIFO，LRU。
- `flushInterval` ： 配置一定时间自动刷新缓存，单位是毫秒。
- `size` ： 最多缓存对象的个数。
- `readOnly` ： 是否只读，若配置可读写，则需要对应的实体类能够序列化。
- `blocking` ： 若缓存中找不到对应的key，是否会一直blocking，直到有对应的数据进入缓存。

`cache-ref` 代表引用别的命名空间的Cache配置，两个命名空间的操作使用的是同一个Cache。

```
<cache-ref namespace="mapper.StudentMapper"/>
```

二级缓存实验

接下来我们通过实验，了解MyBatis二级缓存在使用上的一些特点。

在本实验中，id为1的学生名称初始化为点点。

实验1

测试二级缓存效果，不提交事务， `sqlSession1` 查询完数据后， `sqlSession2` 相同的查询是否会从缓存中获取数据。

```
@Test
public void testCacheWithoutCommitOrClose() throws Exception {
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);

        StudentMapper studentMapper = sqlSession1.getMapper(StudentMapper.class);
        StudentMapper studentMapper2 = sqlSession2.getMapper(StudentMapper.class);

        System.out.println("studentMapper读取数据: " + studentMapper.getStudentById(1));
        System.out.println("studentMapper2读取数据: " + studentMapper2.getStudentById(1));
}
```

执行结果：

```
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 点点, 16
DEBUG [main] - <==      Total: 1
studentMapper读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 点点, 16
DEBUG [main] - <==      Total: 1
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
```

我们可以看到，当 `sqlsession` 没有调用 `commit()` 方法时，二级缓存并没有起到作用。

实验2

测试二级缓存效果，当提交事务时， `sqlSession1` 查询完数据后， `sqlSession2` 相同的查询是否会从缓存中获取数据。

```
@Test
public void testCacheWithCommitOrClose() throws Exception {
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);

        StudentMapper studentMapper = sqlSession1.getMapper(StudentMapper.class);
        StudentMapper studentMapper2 = sqlSession2.getMapper(StudentMapper.class);

        System.out.println("studentMapper读取数据: " + studentMapper.getStudentById(1));
        sqlSession1.commit();
        System.out.println("studentMapper2读取数据: " + studentMapper2.getStudentById(1));
}
```

```
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 点点, 16
DEBUG [main] - <==      Total: 1
studentMapper读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.5
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
```

从图上可知， `sqlsession2` 的查询，使用了缓存，缓存的命中率是0.5。

实验3

测试 `update` 操作是否会刷新该 `namespace` 下的二级缓存。

```
@Test
public void testCacheWithUpdate() throws Exception {
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);
        SqlSession sqlSession3 = factory.openSession(true);

        StudentMapper studentMapper = sqlSession1.getMapper(StudentMapper.class);
        StudentMapper studentMapper2 = sqlSession2.getMapper(StudentMapper.class);
        StudentMapper studentMapper3 = sqlSession3.getMapper(StudentMapper.class);

        System.out.println("studentMapper读取数据: " + studentMapper.getStudentById(1));
        sqlSession1.commit();
        System.out.println("studentMapper2读取数据: " + studentMapper2.getStudentById(1));

        studentMapper3.updateStudentName("方方",1);
        sqlSession3.commit();
        System.out.println("studentMapper2读取数据: " + studentMapper2.getStudentById(1));
}
```

```
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.0
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 点点, 16
DEBUG [main] - <==      Total: 1
studentMapper读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.5
studentMapper2读取数据: StudentEntity{id=1, name='点点', age=16, className='null'}
DEBUG [main] - ==>  Preparing: UPDATE student SET name = ? WHERE id = ?
DEBUG [main] - ==> Parameters: 方方(String), 1(Integer)
DEBUG [main] - <==    Updates: 1
DEBUG [main] - Cache Hit Ratio [mapper.StudentMapper]: 0.3333333333333333
DEBUG [main] - ==>  Preparing: SELECT id,name,age FROM student WHERE id = ?
DEBUG [main] - ==> Parameters: 1(Integer)
TRACE [main] - <==    Columns: id, name, age
TRACE [main] - <==        Row: 1, 方方, 16
DEBUG [main] - <==      Total: 1
studentMapper2读取数据: StudentEntity{id=1, name='方方', age=16, className='null'}
```

更新后，缓存被刷新，之后相同的查询走了数据库

我们可以看到，在 `sqlSession3` 更新数据库，并提交事务后， `sqlsession2` 的 `StudentMapper namespace` 下的查询走了数据库，没有走Cache。

实验4

验证MyBatis的二级缓存不适应用于映射文件中存在多表查询的情况。

通常我们会为每个单表创建单独的映射文件，由于MyBatis的二级缓存是基于 `namespace` 的，多表查询语句所在的 `namspace` 无法感应到其他 `namespace` 中的语句对多表查询中涉及的表进行的修改，引发脏数据问题。

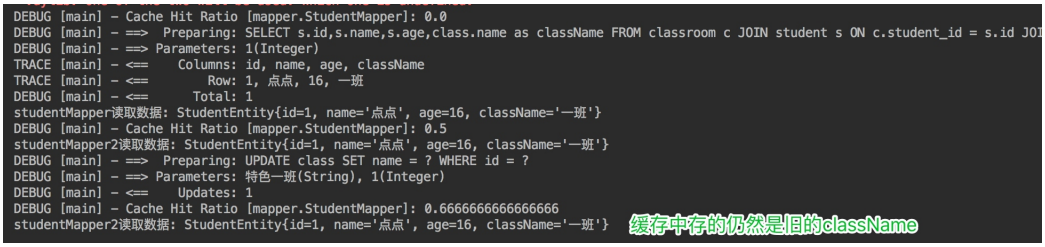
```
@Test
public void testCacheWithDifferentNamespace() throws Exception {
        SqlSession sqlSession1 = factory.openSession(true);
        SqlSession sqlSession2 = factory.openSession(true);
        SqlSession sqlSession3 = factory.openSession(true);
```

```
        StudentMapper studentMapper = sqlSession1.getMapper(StudentMapper.class);
        StudentMapper studentMapper2 = sqlSession2.getMapper(StudentMapper.class);
        ClassMapper classMapper = sqlSession3.getMapper(ClassMapper.class);

        System.out.println("studentMapper读取数据: " + studentMapper.getStudentByIdWithClassInfo(1));
        sqlSession1.close();
        System.out.println("studentMapper2读取数据: " + studentMapper2.getStudentByIdWithClassInfo(1));

        classMapper.updateClassName("特色一班",1);
        sqlSession3.commit();
        System.out.println("studentMapper2读取数据: " + studentMapper2.getStudentByIdWithClassInfo(1));
}
```

执行结果：



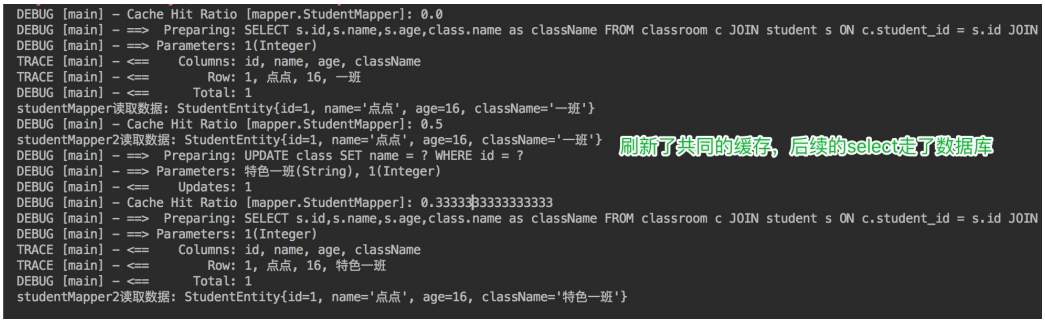
在这个实验中，我们引入了两张新的表，一张class，一张classroom。class中保存了班级的id和班级名，classroom中保存了班级id和学生id。我们在 `StudentMapper` 中增加了一个查询方法 `getStudentByIdWithClassInfo` ，用于查询学生所在的班级，涉及到多表查询。在 `ClassMapper` 中添加了 `updateClassName` ，根据班级id更新班级名的操作。

当 `sqlsession1` 的 `studentmapper` 查询数据后，二级缓存生效。保存在StudentMapper的namespace下的cache中。当 `sqlSession3` 的 `classMapper` 的 `updateClassName` 方法对class表进行更新时， `updateClassName` 不属于 `StudentMapper` 的 `namespace` ，所以 `StudentMapper` 下的cache没有感应到变化，没有刷新缓存。当 `StudentMapper` 中同样的查询再次发起时，从缓存中读取了脏数据。

实验5

为了解决实验4的问题呢，可以使用Cache ref，让 `ClassMapper` 引用 `StudenMapper` 命名空间，这样两个映射文件对应的SQL操作都使用的是同一块缓存了。

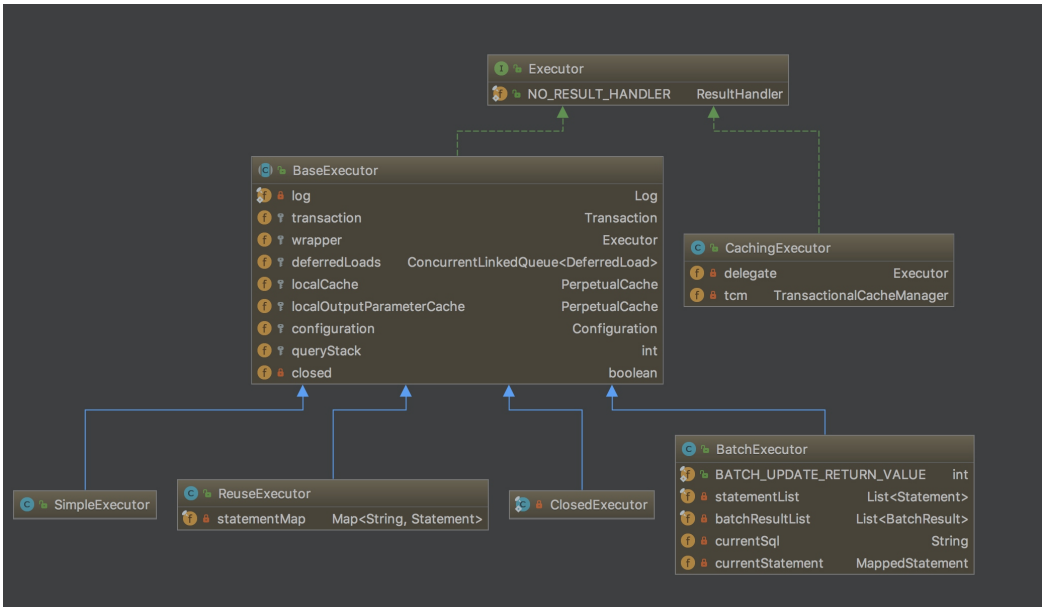
执行结果：



不过这样做的后果是，缓存的粒度变粗了，多个 `Mapper namespace` 下的所有操作都会对缓存使用造成影响。

二级缓存源码分析

MyBatis二级缓存的工作流程和前文提到的一级缓存类似，只是在一级缓存处理前，用 `CachingExecutor` 装饰了 `BaseExecutor` 的子类，在委托具体职责给 `delegate` 之前，实现了二级缓存的查询和写入功能，具体类关系图如下图所示。



源码分析

源码分析从 `CachingExecutor` 的 `query` 方法展开，源代码走读过程中涉及到的知识点较多，不能一一详细讲解，读者朋友可以自行查询相关资料来学习。

`CachingExecutor` 的 `query` 方法，首先会从 `MappedStatement` 中获得在配置初始化时赋予的Cache。

```
Cache cache = ms.getCache();
```

本质上是装饰器模式的使用，具体的装饰链是：

> SynchronizedCache –> LoggingCache –> SerializedCache –> LruCache –> PerpetualCache。

以下是具体这些Cache实现类的介绍，他们的组合为Cache赋予了不同的能力。

- `SynchronizedCache` ：同步Cache，实现比较简单，直接使用synchronized修饰方法。
- `LoggingCache` ：日志功能，装饰类，用于记录缓存的命中率，如果开启了DEBUG模式，则会输出命中率日志。
- `SerializedCache` ：序列化功能，将值序列化后存到缓存中。该功能用于缓存返回一份实例的Copy，用于保存线程安全。

- `LruCache` ： 采用了Lru算法的Cache实现，移除最近最少使用的Key/Value。
- `PerpetualCache` ： 作为为最基础的缓存类，底层实现比较简单，直接使用了HashMap。

然后是判断是否需要刷新缓存，代码如下所示：

```
flushCacheIfRequired(ms);
```

在默认的设置中 `SELECT` 语句不会刷新缓存， `insert/update/delte` 会刷新缓存。进入该方法。代码如下所示：

```
private void flushCacheIfRequired(MappedStatement ms) {
    Cache cache = ms.getCache();
    if (cache != null && ms.isFlushCacheRequired()) {
      tcm.clear(cache);
    }
}
```

MyBatis的 `CachingExecutor` 持有了 `TransactionalCacheManager` ，即上述代码中的tcm。

`TransactionalCacheManager` 中持有了一个Map，代码如下所示：

```
private Map<Cache, TransactionalCache> transactionalCaches = new HashMap<Cache, TransactionalCache>();
```

这个Map保存了Cache和用 `TransactionalCache` 包装后的Cache的映射关系。

`TransactionalCache` 实现了Cache接口， `CachingExecutor` 会默认使用他包装初始生成的Cache，作用是如果事务提交，对缓存的操作才会生效，如果事务回滚或者不提交事务，则不对缓存产生影响。

在 `TransactionalCache` 的clear，有以下两句。清空了需要在提交时加入缓存的列表，同时设定提交时清空缓存，代码如下所示：

```
@Override
public void clear() {
        clearOnCommit = true;
        entriesToAddOnCommit.clear();
}
```

`CachingExecutor` 继续往下走， `ensureNoOutParams` 主要是用来处理存储过程的，暂时不用考虑。

```
if (ms.isUseCache() && resultHandler == null) {
        ensureNoOutParams(ms, parameterObject, boundSql);
```

之后会尝试从tcm中获取缓存的列表。

```
List<E> list = (List<E>) tcm.getObject(cache, key);
```

在 `getObject` 方法中，会把获取值的职责一路传递，最终到 `PerpetualCache` 。如果没有查到，会把key加入Miss集合，这个主要是为了统计命中率。

```
Object object = delegate.getObject(key);
if (object == null) {
        entriesMissedInCache.add(key);
}
```

`CachingExecutor` 继续往下走，如果查询到数据，则调用 `tcm.putObject` 方法，往缓存中放入值。

```
if (list == null) {
        list = delegate.<E> query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
        tcm.putObject(cache, key, list); // issue #578 and #116
}
```

tcm的 `put` 方法也不是直接操作缓存，只是在把这次的数据和key放入待提交的Map中。

```
@Override
public void putObject(Object key, Object object) {
    entriesToAddOnCommit.put(key, object);
}
```

从以上的代码分析中，我们可以明白，如果不调用 `commit` 方法的话，由于 `TranscationalCache` 的作用，并不会对二级缓存造成直接的影响。因此我们看看 `Sqlsession` 的 `commit` 方法中做了什么。代码如下所示：

```
@Override
public void commit(boolean force) {
    try {
      executor.commit(isCommitOrRollbackRequired(force));
```

因为我们使用了CachingExecutor，首先会进入CachingExecutor实现的commit方法。

```
@Override
public void commit(boolean required) throws SQLException {
    delegate.commit(required);
    tcm.commit();
}
```

会把具体commit的职责委托给包装的 `Executor` 。主要是看下 `tcm.commit()` ，tcm最终又会调用到 `TrancationalCache` 。

```
public void commit() {
    if (clearOnCommit) {
      delegate.clear();
    }
    flushPendingEntries();
    reset();
}
```

看到这里的 `clearOnCommit` 就想起刚才 `TrancationalCache` 的 `clear` 方法设置的标志位，真正的清理Cache是放到这里来进行的。具体清理的职责委托给了包装的Cache类。之后进入 `flushPendingEntries` 方法。代码如下所示：

```
private void flushPendingEntries() {
    for (Map.Entry<Object, Object> entry : entriesToAddOnCommit.entrySet()) {
      delegate.putObject(entry.getKey(), entry.getValue());
    }
    ..............
}
```

在 `flushPending` Entries中，将待提交的Map进行循环处理，委托给包装的Cache类，进行 `putObject` 的操作。

后续的查询操作会重复执行这套流程。如果是 `insert|update|delete` 的话，会统一进入 `CachingExecutor` 的 `update` 方法，其中调用了这个函数，代码如下所示：

```
private void flushCacheIfRequired(MappedStatement ms)
```

在二级缓存执行流程后就会进入一级缓存的执行流程，因此不再赘述。

总结

1. MyBatis的二级缓存相对于一级缓存来说，实现了 `SqlSession` 之间缓存数据的共享，同时粒度更加的细，能够到 `namespace` 级别，通过Cache接口实现类不同的组合，对Cache的可控性也更强。
2. MyBatis在多表查询时，极大可能会出现脏数据，有设计上的缺陷，安全使用二级缓存的条件比较苛刻。
3. 在分布式环境下，由于默认的MyBatis Cache实现都是基于本地的，分布式环境下必然会出现读取到脏数据，需要使用集中式缓存将MyBatis的Cache接口实现，有一定的开发成本，直接使用Redis、Memcached等分布式缓存可能成本更低，安全性也更高。

全文总结

本文对介绍了MyBatis一二级缓存的基本概念，并从应用及源码的角度对MyBatis的缓存机制进行了分析。最后对MyBatis缓存机制做了一定的总结，个人建议MyBatis缓存特性在生产环境中进行关闭，单纯作为一个ORM框架使用可能更为合适。

作者简介

- 凯伦，美团点评后端研发工程师，2016年毕业于上海海事大学，现从事美团点评餐饮平台相关的开发工作。

招聘信息

美团点评点餐事业部期待你的加入，上海在招岗位：Java后台，数据开发，前端，QA，产品，产品运营，商业分析等。内推简历邮箱：weiyanping#meituan.com

使用TensorFlow训练WDL模型性能问题定位与调优

作者: 郑坤

简介

TensorFlow是Google研发的第二代人工智能学习系统，能够处理多种深度学习算法模型，以功能强大和高可扩展性而著称。TensorFlow完全开源，所以很多公司都在使用，但是美团点评在使用分布式TensorFlow训练WDL模型时，发现训练速度很慢，难以满足业务需求。

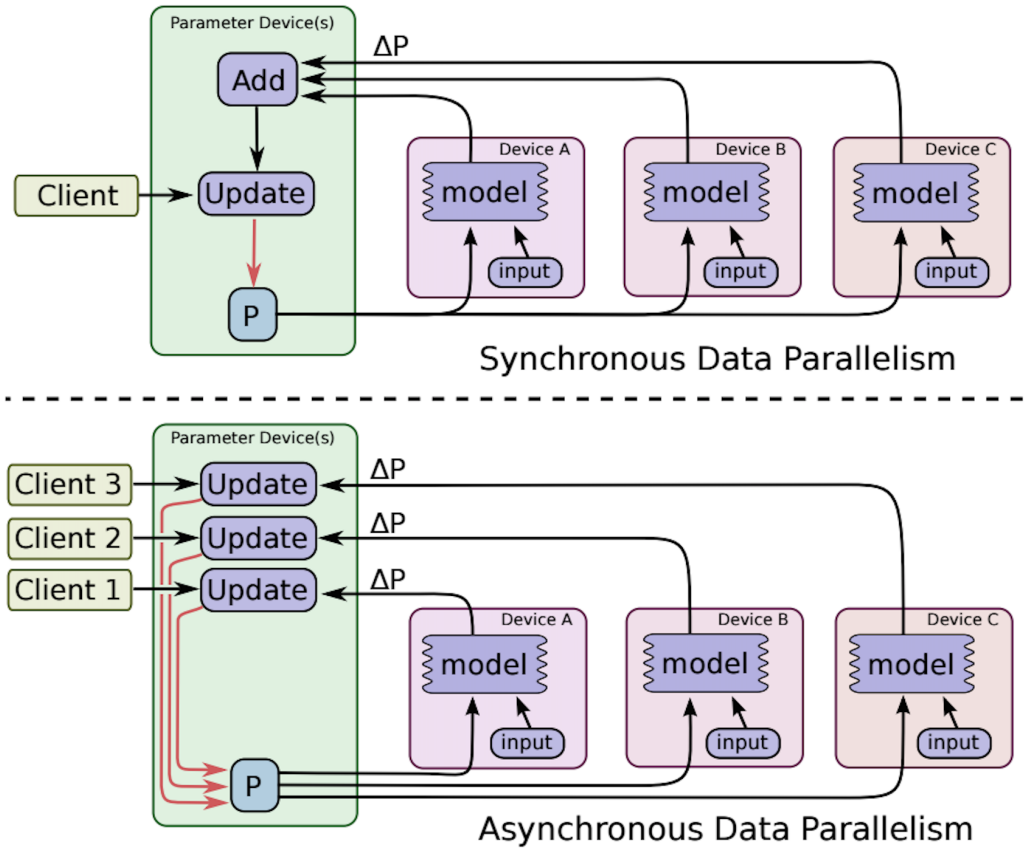
经过对TensorFlow框架和Hadoop的分析定位，发现在数据输入、集群网络和计算内存分配等层面出现性能瓶颈。主要原因包括TensorFlow数据输入接口效率低、PS/Worker算子分配策略不佳以及Hadoop参数配置不合理。我们在调整对TensorFlow接口调用、并且优化系统配置后，WDL模型训练性能提高了10倍，分布式线性加速可达32个Worker，基本满足了美团点评广告和推荐等业务的需求。

术语

- TensorFlow – Google发布的开源深度学习框架
- OP – Operation缩写，TensorFlow算子
- PS – Parameter Server 参数服务器
- WDL – **Wide & Deep Learning**🔗，Google发布的用于推荐场景的深度学习算法模型
- AFO – AI Framework on YARN的简称 – 基于YARN开发的深度学习调度框架，支持TensorFlow，MXNet等深度学习框架

TensorFlow分布式架构简介

为了解决海量参数的模型计算和参数更新问题，TensorFlow支持分布式计算。和其他深度学习框架的做法类似，分布式TensorFlow也引入了参数服务器（Parameter Server，PS），用于保存和更新训练参数，而模型训练放在Worker节点完成。



TensorFlow分布式架构

TensorFlow支持图并行（in-graph）和数据并行（between-graph）模式，也支持同步更新和异步更新。因为in-graph只在一个节点输入并分发数据，严重影响并行训练速度，实际生产环境中一般使用between-graph。

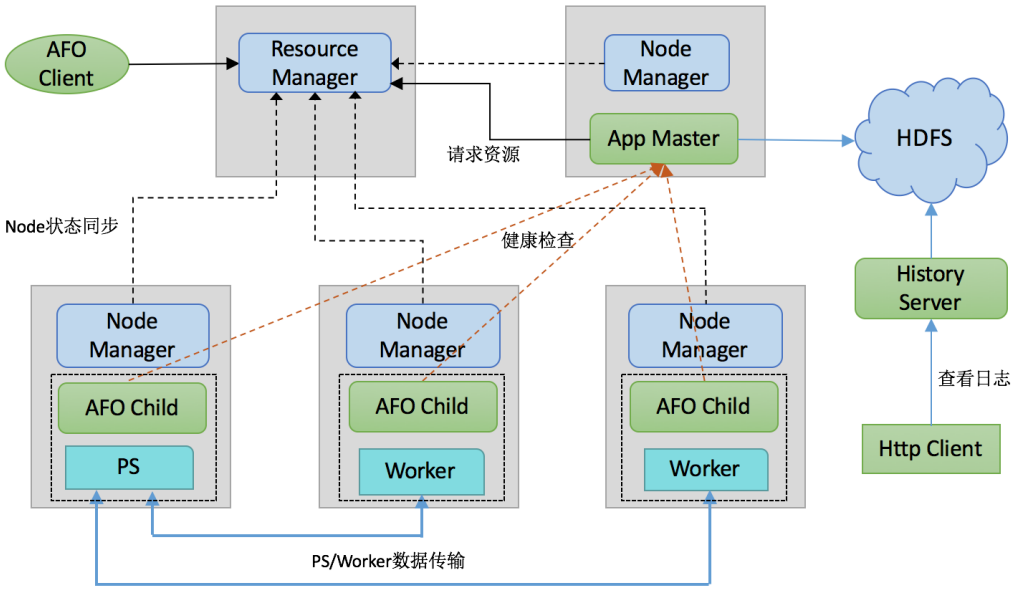
同步更新时，需要一个Woker节点为Chief，来控制所有的Worker是否进入下一轮迭代，并且负责输出checkpoint。异步更新时所有Worker都是对等的，迭代过程不受同步barrier控制，训练过程更快。

AFO架构设计

TensorFlow只是一个计算框架，没有集群资源管理和调度的功能，分布式训练也欠缺集群容错方面的能力。为了解决这些问题，我们在YARN基础上自研了AFO框架解决这个问题。

AFO架构特点：

- 高可扩展，PS、Worker都是任务（Task），角色可配置
- 基于状态机的容错设计
- 提供了日志服务和Tensorboard服务，方便用户定位问题和模型调试



AFO 架构

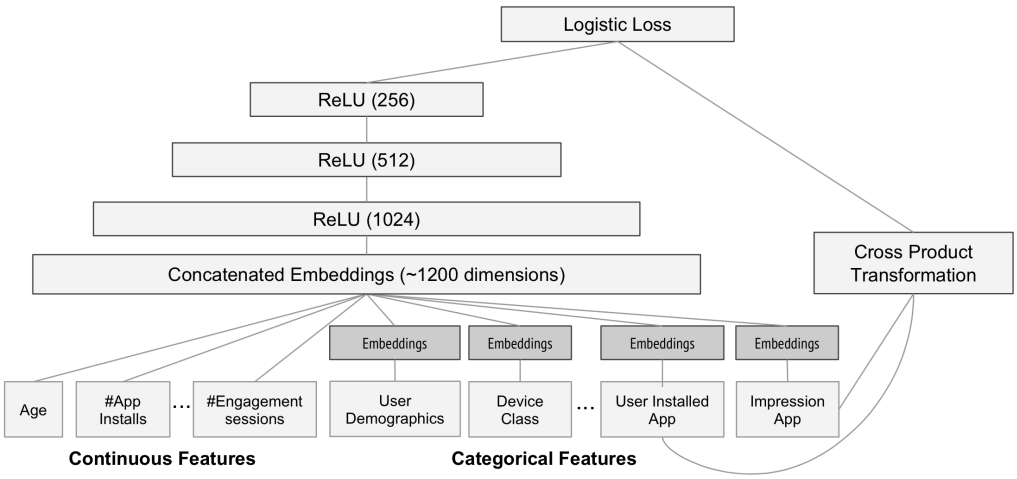
AFO模块说明：

- Application Master：用来管理整个TensorFlow集群的资源申请，对任务进行状态监控
- AFO Child：TensorFlow执行引擎，负责PS、Worker运行时管理和状态同步
- History Server：管理TensorFlow训练生成的日志
- AFO Client：用户客户端

WDL模型

在推荐系统、CTR预估场景中，训练的样本数据一般是查询、用户和上下文信息，系统返回一个排序好的候选列表。推荐系统面临的主要问题是，如何同时可以做到模型的记忆能力和泛化能力，WDL提出的思想是结合线性模型（Wide，用于记忆）和深度神经网络（Deep，用于泛化）。

以论文中用于Google Play Store推荐系统的WDL模型为例，该模型输入用户访问应用商店的日志，用户和设备的信息，给应用App打分，输出一个用户“感兴趣”App列表。



WDL 模型网络

其中，installed apps和impression apps这类特征具有稀疏性（在海量大小的App空间中，用户感兴趣的只有很少一部分），对应模型“宽的部分”，适合使用线性模型；在模型“深的部分”，稀疏特征由于维度太高不适合神经网络处理，需要embedding降维转成稠密特征，再和其他稠密特征串联起来，输入到一个3层ReLU的深度网络。最后Wide和Deep的预估结果加权输入给一个Logistic损失函数（例如Sigmoid）。

WDL模型中包含对稀疏特征的embedding计算，在TensorFlow中对应的接口是tf.embedding_lookup_sparse，但该接口所包含的OP无法使用GPU加速，只能在CPU上计算，因此TensorFlow在处理稀疏特征性能不佳。不仅如此，我们发现分布式TensorFlow在进行embedding计算时会引发大量的网络传输流量，严重影响训练性能。

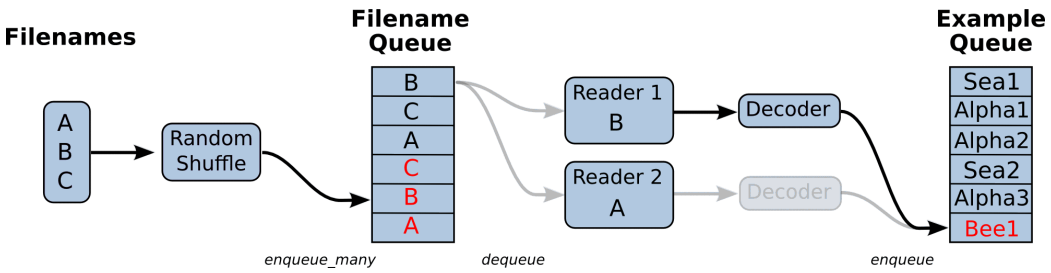
性能瓶颈分析与调优

在使用TensorFlow训练WDL模型时，我们主要发现3个性能问题：

1. 每轮训练时，输入数据环节耗时过多，超过60%的时间用于读取数据。
2. 训练时产生的网络流量高，占用大量集群网络带宽资源，难以实现分布式性能线性加速。
3. Hadoop的默认参数配置导致glibc malloc变慢，一个保护malloc内存池的内核自旋锁成为性能瓶颈。

TensorFlow输入数据瓶颈

TensorFlow支持以流水线（Pipeline）的方式输入训练数据。如下图所示，典型的输入数据流水线包含两个队列：Filename Queue对一组文件做shuffle，多个Reader线程从此队列中拿到文件名，读取训练数据，再经过Decode过程，将数据放入Example Queue，以备训练线程从中读取数据。Pipeline这种多线程、多队列的设计可以使训练线程和读数据线程并行。理想情况下，队列Example Queue总是充满数据的，训练线程完成一轮训练后可以立即读取下一批的数据。如果Example Queue总是处于“饥饿”状态，训练线程将不得不阻塞，等待Reader线程将Example Queue插入足够的数据。使用TensorFlow Timeline工具，可以直观地看到其中的OP调用过程。



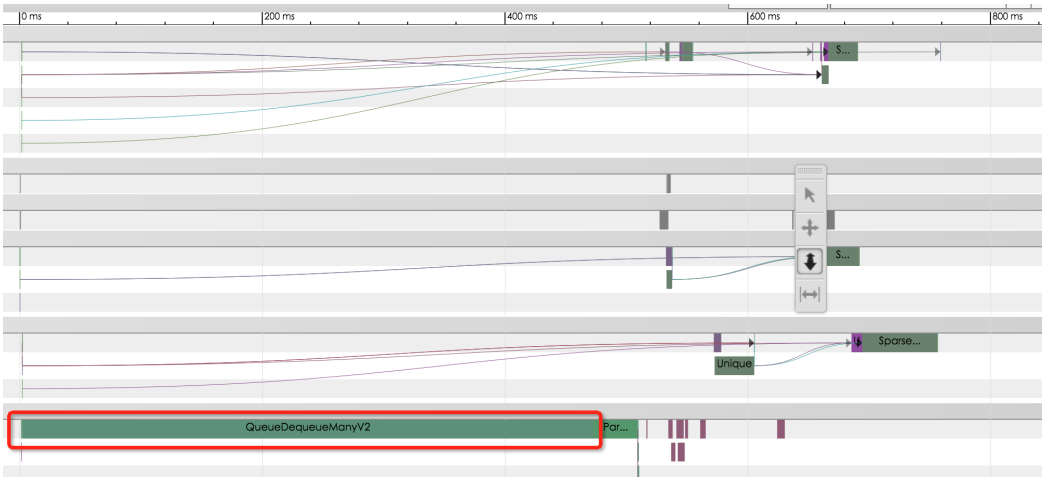
TensorFlow输入数据流水线

使用Timeline，需要对tf.Session.run()增加如下几行代码：

```
with tf.Session as sess:
    ptions = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
    run_metadata = tf.RunMetadata()
    _ = sess.run([train_op, global_step], options=run_options, run_metadata=run_metadata)
    if global_step > 1000 && global_step < 1010:
        from tensorflow.python.client import timeline
        fetched_timeline = timeline.Timeline(run_metadata.step_stats)
        chrome_trace = fetched_timeline.generate_chrome_trace_format()
        with open('/tmp/timeline_01.json', 'w') as f:
            f.write(chrome_trace)
```

这样训练到global step在1000轮左右时，会将该轮训练的Timeline信息保存到timeline_01.json文件中，在Chrome浏览器的地址栏中输入chrome://tracing，然后load该文件，可以看到图像化的Profiling结果。

业务模型的Timeline如图所示：

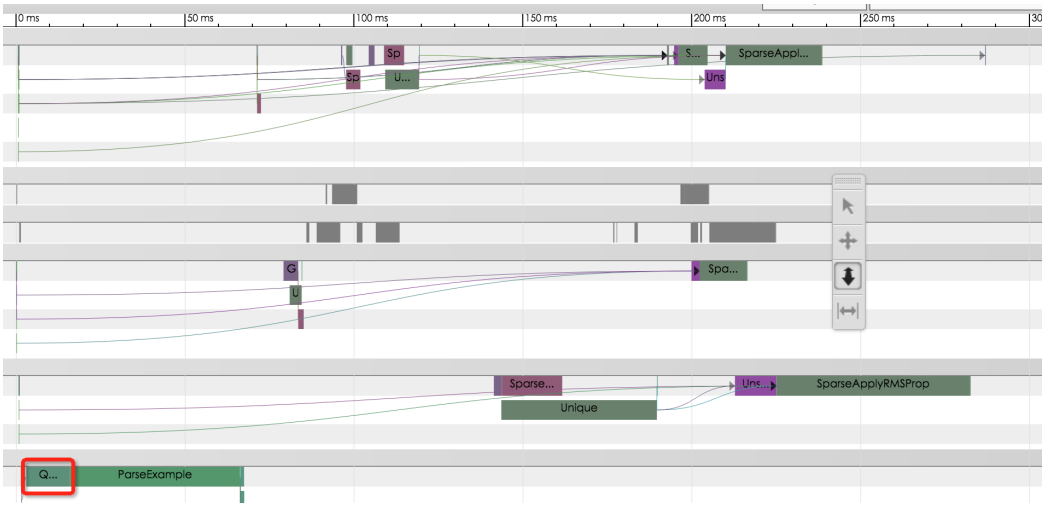


Timeline显示数据输入是性能瓶颈

可以看到QueueDequeueManyV2这个OP耗时最久，约占整体时延的60%以上。通过分析TensorFlow源码，我们判断有两方面的原因：

- （1）Reader线程是Python线程，受制于Python的全局解释锁（GIL），Reader线程在训练时没有获得足够的调度执行；
- （2）Reader默认的接口函数TFRecordReader.read函数每次只读入一条数据，如果Batch Size比较大，读入一个Batch的数据需要频繁调用该接口，系统开销很大；

针对第一个问题，解决办法是使用 **TensorFlow Dataset接口**，该接口不再使用Python线程读数据，而是用C++线程实现，避免了Python GIL问题。 针对第二个问题，社区提供了批量读数据接口TFRecordReader.read_up_to，能够指定每次读数据的数量。我们设置每次读入1000条数据，使读数句接口被调用的频次从10000次降低到10次，每轮训练时延降低2–3倍。



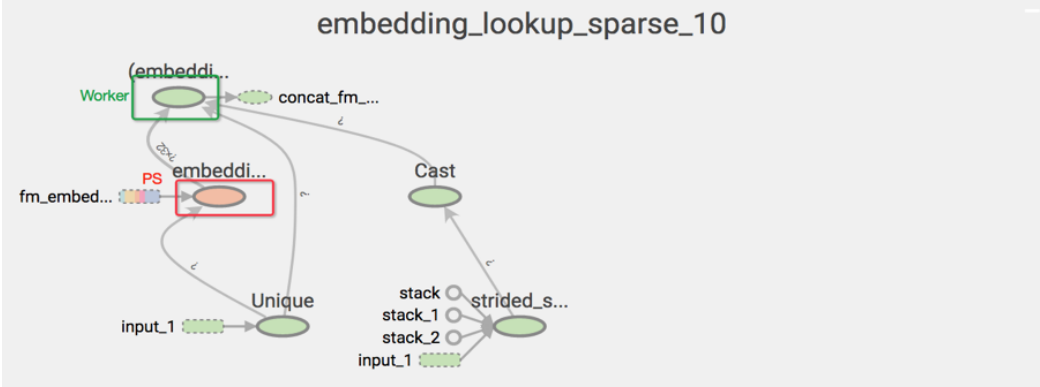
优化数据输入使性能提升2-3倍

可以看到经过调优后，QueueDequeueManyV2耗时只有十几毫秒，每轮训练时延从原来的800多毫秒降低至不到300毫秒。

集群网络瓶颈

虽然使用了Mellanox的25G网卡，但是在WDL训练过程中，我们观察到Worker上的上行和下行网络流量抖动剧烈，幅度2-10Gbps，这是由于打满了PS网络带宽导致丢包。因为分布式训练参数都是保存和更新都是在PS上的，参数过多，加之模型网络较浅，计算很快，很容易形成多个Worker打一个PS的情况，导致PS的网络接口带宽被打满。

在推荐业务的WDL模型中，embedding张量的参数规模是千万级，TensorFlow的tf.embedding_lookup_sparse接口包含了几个OP，默认是分别摆放在PS和Worker上的。如图所示，颜色代表设备，embedding lookup需要在不同设备之前传输整个embedding变量，这意味着每轮Embedding的迭代更新需要将海量的参数在PS和Worker之间来回传输。



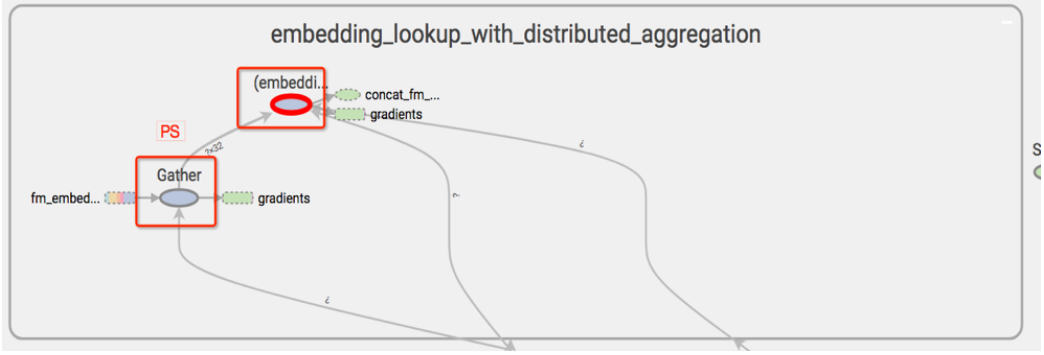
embedding_lookup_sparse的OP拓扑图

有效降低网络流量的方法是尽量让参数更新在一个设备上完成，即

```
with tf.device(PS):
    do embedding computing
```

社区提供了一个接口方法正是按照这个思想实现的：**embedding_lookup_sparse_with_distributed_aggregation接口**，该接口可以将embedding计算的所使用的OP都放在变量所在的PS上，计算后转成稠密张量再传送到Worker上继续网络模型的计算。

从下图可以看到，embedding计算所涉及的OP都是在PS上，测试Worker的上行和下行网络流量也稳定在2-3Gpbs这一正常数值。

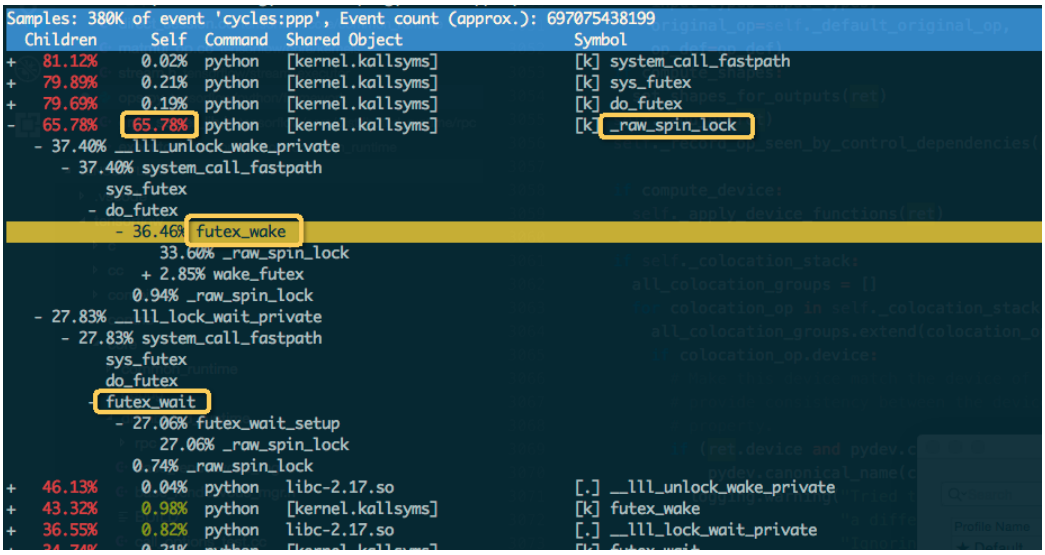


embedding_lookup_sparse_with_distributed_aggregation的OP拓扑图

PS上的UniqueOP性能瓶颈

在使用分布式TensorFlow 跑广告推荐的WDL算法时，发现一个奇怪的现象：WDL算法在AFO上的性能只有手动分布式的1/4。手动分布式是指：不依赖YARN调度，用命令行方式在集群上分别启动PS和Worker作业。

使用Perf诊断PS进程热点，发现PS多线程在竞争一个内核自旋锁，PS整体上有30%-50%的CPU时间耗在malloc的在内核的spin_lock上。



Perf诊断PS计算瓶颈

进一步查看PS进程栈，发现竞争内核自旋锁来自于malloc相关的系统调用。WDL的embedding_lookup_sparse会使用UniqueOp算子，TensorFlow支持OP多线程，UniqueOp计算时会开多线程，线程执行时会调用glibc的malloc申请内存。

经测试排查，发现Hadoop有一项默认的环境变量配置:

```
export MALLOC_ARENA_MAX="4"
```

该配置意思是限制进程所能使用的glibc内存池个数为4个。这意味着当进程开启多线程调用malloc时，最多从4个内存池中竞争申请，这限制了调用malloc的线程并行执行数量最多为4个。

翻查Hadoop社区 **相关讨论**，当初增加这一配置的主要原因是：glibc的升级带来多线程ARENA的特性，可以提高malloc的并发性能，但同时也增加进程的虚拟内存（即top结果中的VIRT）。YARN管理进程树的虚拟内存和物理内存使用量，超过限制的进程树将被杀死。将MALLOC_ARENA_MAX的默认设置改为4之后，可以不至于VIRT增加很多，而且一般作业性能没有明显影响。

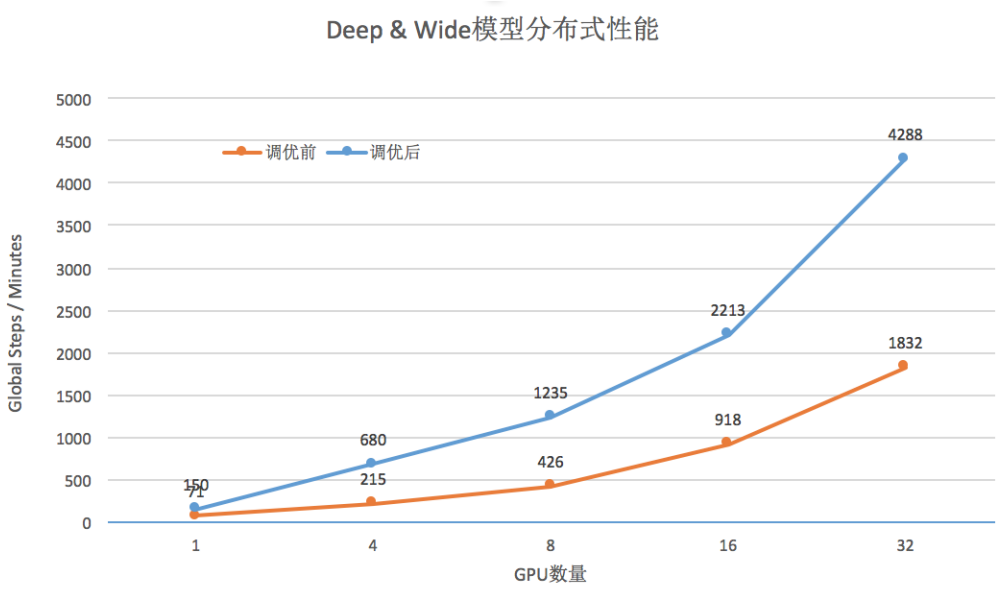
但这个默认配置对于WDL深度学习作业影响很大，我们去掉了这个环境配置，malloc并发性能极大提升。经过测试，WDL模型的平均训练时间性能减少至原来的1/4。

调优结果

注意：以下测试都去掉了Hadoop MALLOC_ARENA_MAX的默认配置

我们在AFO上针对业务的WDL模型做了性能调优前后的比对测试，测试环境参数如下：

```
模型：推荐广告模型WDL
OS：CentOS 7.1
CPU：Xeon E5 2.2G, 40 Cores
GPU：Nvidia P40
磁盘：Local Rotational Disk
网卡：Mellanox 25G（未使用RoCE）
TensorFlow版本：Release 1.4
CUDA/cuDNN：8.0/5.1
```



分布式线性加速效果

可以看到调优后，训练性能提高2-3倍，性能可以达到32个GPU线性加速。这意味着如果使用同样的资源，业务训练时间会更快，或者说在一定的性能要求下，资源节省更多。如果考虑优化MALLOC_ARENA_MAX的因素，调优后的训练性能提升约为10倍左右。

总结

我们使用TensorFlow训练WDL模型发现一些系统上的性能瓶颈点，通过针对性的调优不仅可以大大加速训练过程，而且可以提高GPU、带宽等资源的利用率。在深入挖掘系统热点瓶颈的过程中，我们也加深了对业务算法模型、TensorFlow框架的理解，具有技术储备的意义，有助于我们后续进一步优化深度学习平台性能，更好地为业务提供工程技术支持。

作者简介

- 郑坤，美团点评技术专家，2015年加入美团点评，负责深度学习平台、Docker平台的研发工作。

招聘

美团点评GPU计算团队，致力于打造公司一体化的深度学习基础设施平台，涉及到的技术包括：资源调度、高性能存储、高性能网络、深度学习框架等。目前平台还在建设中期，不论在系统底层、分布式架

构、算法工程优化上都有很大的挑战！诚邀对这个领域感兴趣的同学加盟，不论是工程背景，还是算法背景我们都非常欢迎。有兴趣的同学可以发送简历到zhengkun@meituan.com。

美团点评基于 Flink 的实时数仓建设实践

作者: 伟伦 徐阳 喻灿 刘强

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。

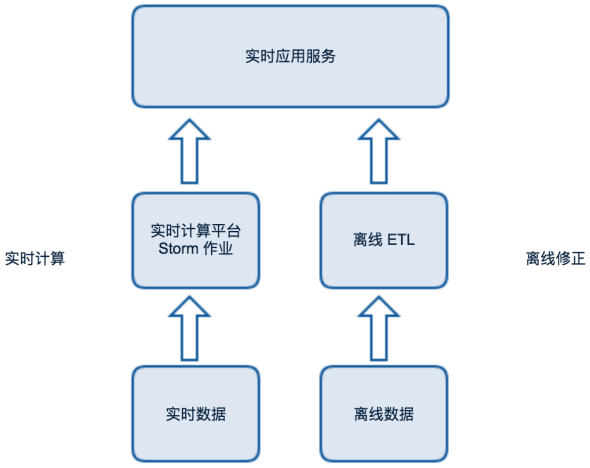


图1 初期实时数据架构

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：

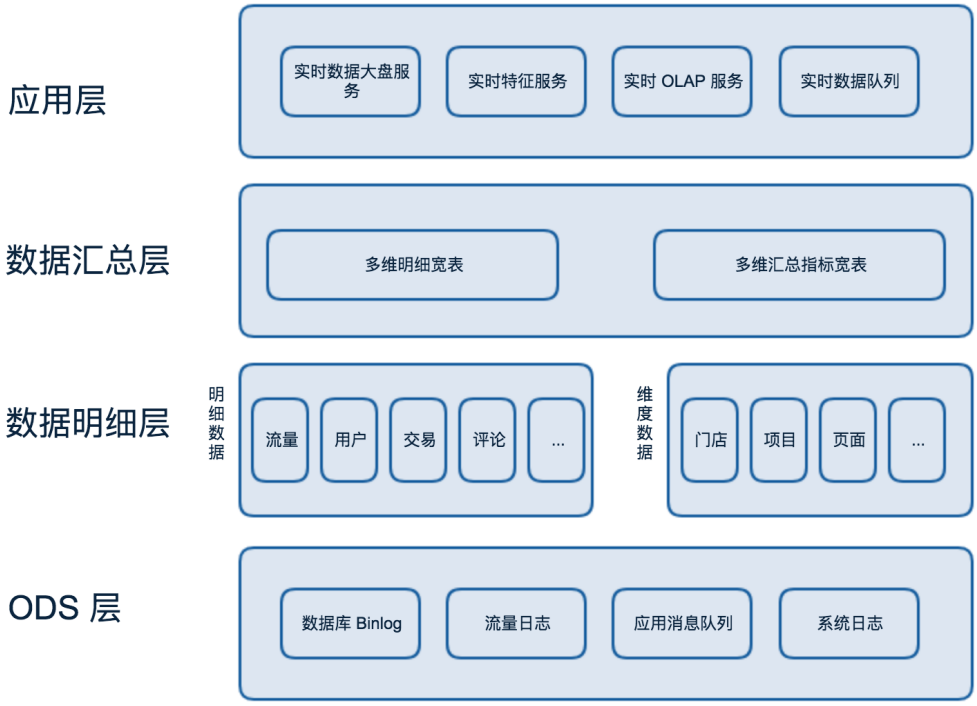


图2 实时数仓数据分层架构

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. 预聚合导致无法支持明细的查询；2. 无法支持 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。
Cellar	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。	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：

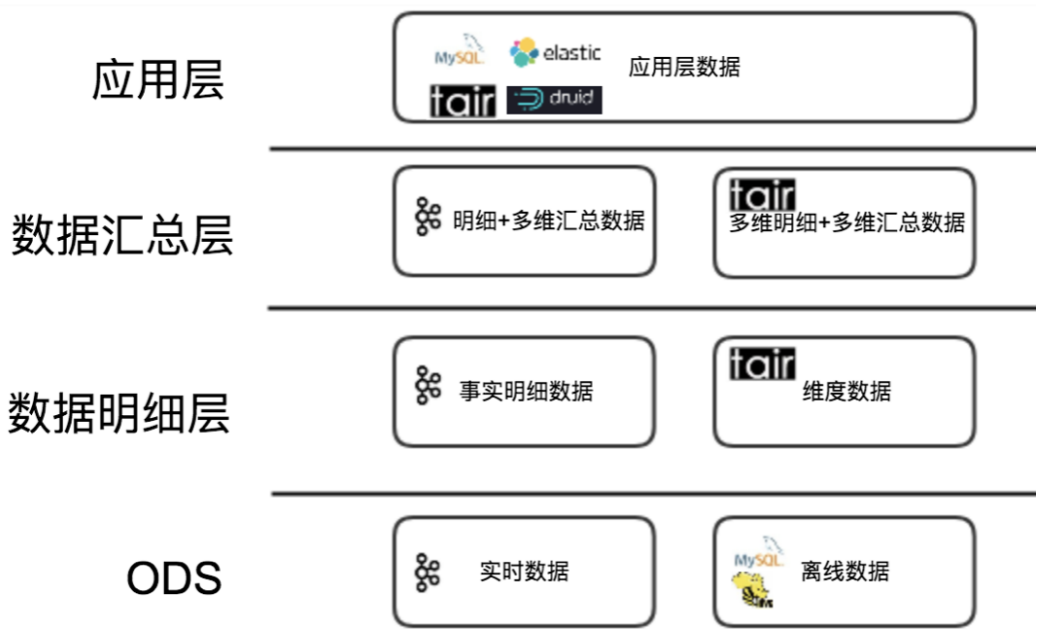


图3 实时数仓存储分层架构

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
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。

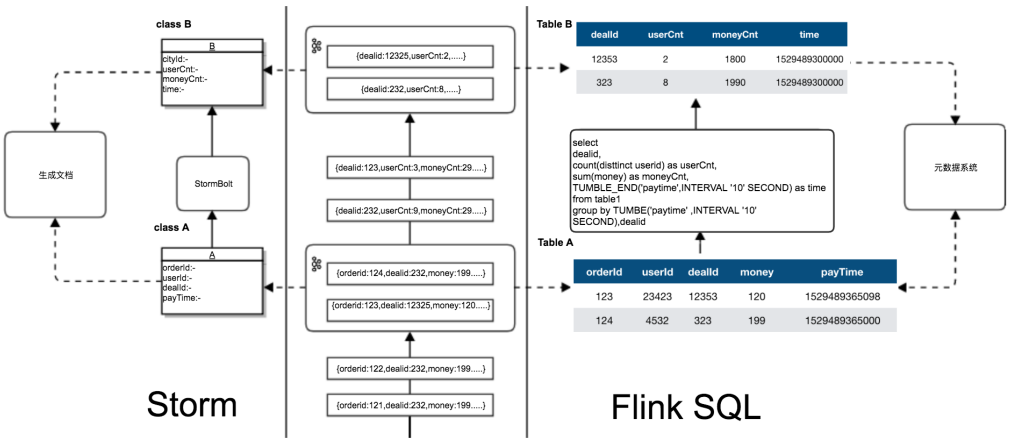


图4 Flink - Storm 对比图

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：

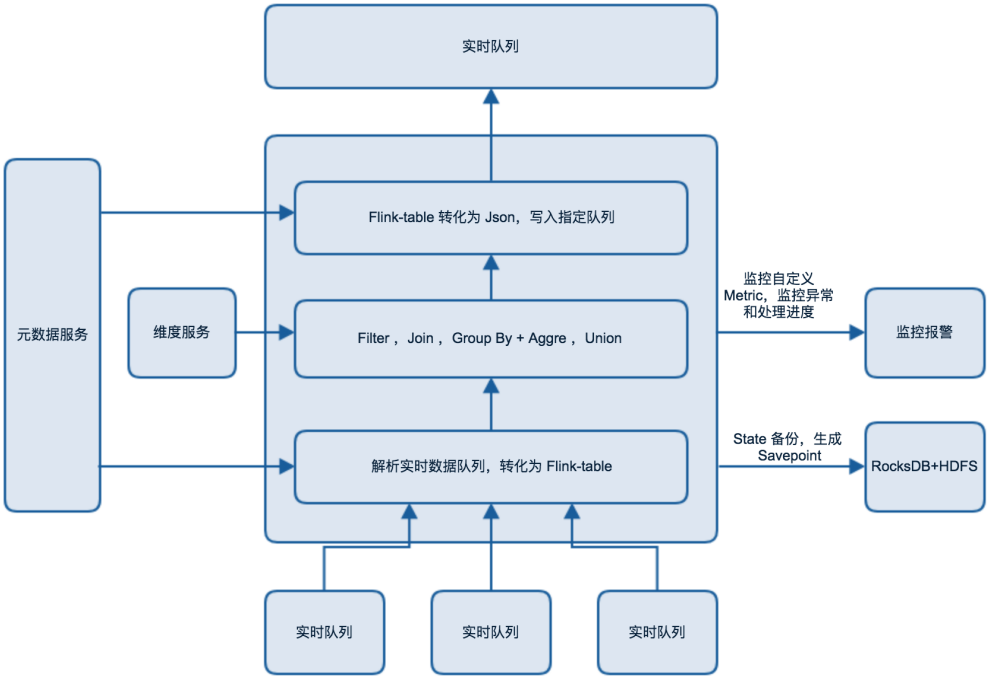


图5 实时计算流程图

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。

参考文献

流计算框架 Flink 与 Storm 的性能对比

作者简介

- 伟伦，美团到店餐饮技术部实时数据负责人，2017年加入美团，长期从事数据平台、实时数据计算、数据架构方面的开发工作。在使用 Flink 进行实时数据生产和提高生产效率上，有一些心得和产出。同时也积极推广 Flink 在实时数据处理中的实战经验。

招聘信息

对数据工程和将数据通过服务业务释放价值感兴趣的同学，可以发送简历到 huangweilun@meituan.com。我们在实时数据仓库、实时数据治理、实时数据产品开发框架、面向销售和商家侧的数据型创新产品层面，都有很多未知但有意义的领域等你来开拓。

美团点评基于Storm的实时数据处理实践

作者: 徐阳 Fat-Carrot

背景

目前美团点评已累计了丰富的线上交易与用户行为数据，为商家赋能需要我们有更强大的专业化数据加工能力，来帮助商家做出正确的决策从而提高用户体验。目前商家端产品在数据应用上主要基于离线数据加工，数据生产调度以“T+1”为主，伴随着越来越深入的精细化运营，实时数据应用诉求逾加强烈。本文将从目前主流实时数据处理引擎的特点和我们面临的问题出发，简单的介绍一下我们是如何搭建实时数据处理系统。

设计框架

目前比较流行的实时处理引擎有 Storm，Spark Streaming，Flink。每个引擎都有各自的特点和应用场景。 下表是对这三个引擎的简单对比：

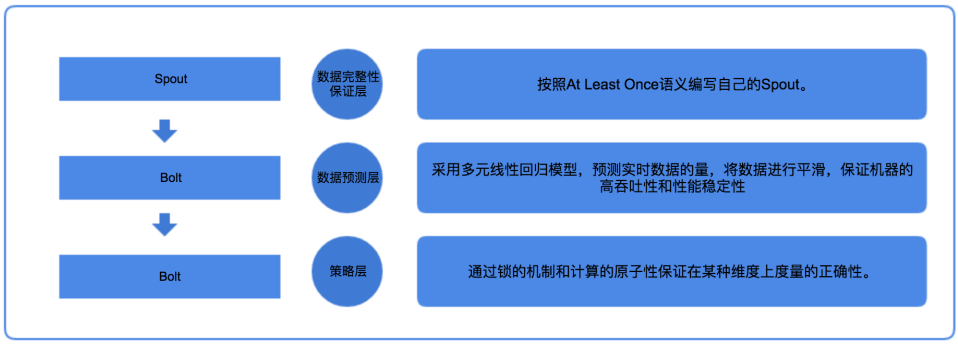
-	Storm	Spark Streaming	Flink
吞吐量	低	高	高
延迟	毫秒级	秒级	毫秒级
语义保障	at least once	exactly once	exactly once/at least once
处理模式	单条数据处理	批量数据处理	单条、批量数据处理
成熟度	成熟	成熟	新兴框架
SQL支持	Beta	成熟	新兴

考虑到每个引擎的特点、商家端应用的特点和系统的高可用性，我们最终选择了 Storm 作为本系统的实时处理引擎。

面临的问题

1. 数据量的不稳定性，导致对机器需求的不确定性。用户的行为数据会受到时间的影响，比如半夜时刻和用餐高峰时段每分钟产生的数据量有两个数量级的差异。
2. 上游数据质量的不确定性。
3. 数据计算时，数据的落地点应该放到哪里来保证计算的高效性。
4. 如何保证数据在多线程处理时数据计算的正确性。
5. 计算好的数据以什么样的方式提供给应用方。

具体的实施方案



实时摄入数据完整性保障

数据完整性保证层：如何保证数据摄入到计算引擎的完整性呢？正如表格中比较的那样，Storm 框架的语义为 At Least Once，至少摄入一次。这个语义的存在正好保证了数据的完整性，所以只需要根据自己的需求编写 Spout 即可。好消息是我们的技术团队已经开发好了一个满足大多数需求的 Spout，可以直接拿来使用。特别需要注意的一点，在数据处理的过程中需要我们自己来剔除已经处理过的数据，因为 Storm 的语义会可能导致同一条数据摄入两次。灰度发布期间（一周）对数据完整性进行验证，数据完整性为100%。

实时数据平滑处理

数据预测层：实时的数据预测可以帮助我们对到达的数据进行有效的平滑，从而可以减少在某一时刻对集群的压力。 在数据预测方面，我们采用了在数学上比较简单的多元线性回归模型（如果此模型不满足业务需求，可以选用一些更高级别的预测模型），预测下一分钟可能到来的数据的量。在数据延迟可接受的范围内，对数据进行平滑，并完成对数据的计算。通过对该方案的使用，减轻了对集群约33%的压力。具体步骤如下：

- **步骤一**：将多个业务的实时数据进行抽象化，转换为(Y_i,X_1*i*,X_2*i*,X_3i,… ,X_ni)，其中Y_i为在(X_1i…X_ni)属性下的数据量，(X_1i…X_ni)为n个不同的属性，比如时间、业务、用户的性别等等。
- **步骤二**：因为考虑到实时数据的特殊性，不同业务的数据量随时间变量基本呈现为M走势，所以为了将非线性走势转换为线性走势，可以将时间段分为4部分，保证在每个时间段内数据的走势为线性走势。同理，如果其他的属性使得走势变为非线性，也可以分段分析。
- **步骤三**：将抽象好的数据代入到多元线性回归模型中，其方程组形式为：

$$Y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \ldots + \beta_k X_{ki} + \mu_i, (i = 1, 2, \ldots, n)$$

即：

$$\begin{cases} Y_1 = \beta_0 + \beta_1 X_{11} + \beta_2 X_{21} + \ldots + \beta_k X_{k1} + \mu_1 \\ Y_2 = \beta_0 + \beta_1 X_{12} + \beta_2 X_{22} + \ldots + \beta_k X_{k2} + \mu_2 \\ \ldots\ldots \\ Y_n = \beta_0 + \beta_1 X_{1n} + \beta_2 X_{2n} + \ldots + \beta_k X_{kn} + \mu_n \end{cases}$$

通过对该模型的求解方式求得估计参数，最后得多元线性回归方程。

$$\hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 X_{1i} + \hat{\beta}_2 X_{2i} + \ldots + \hat{\beta}_k X_{ki}, (i = 1, 2, \ldots, n)$$

- **步骤四**：数据预测完之后通过控制对数据的处理速度，保证在规定的时间内完成对规定数据的计算，减轻对集群的压力。

实时数据计算策略

策略层：Key/Value 模式更适应于实时数据模型，不管是在存储还是计算方面。Cellar（我们内部基于阿里开源的Tair研发的公共KV存储）作为一个分布式的 Key/Value 结构数据的解决方案，可以做到几乎无

延迟的进行 IO 操作，并且可以支持高达千万级别的 QPS，更重要的是 Cellar 支持很多**原子操作**，运用在实时数据计算上是一个不错的选择。所以作为数据的落脚点，本系统选择了Cellar。

但是在数据计算的过程中会遇到一些问题，比如说统计截止到当前时刻入住旅馆的男女比例是多少？很容易就会想到，从 Cellar 中取出截止到当前时刻入住的男生是多少，女生是多少，然后做一个比值就 OK 了。但是本系统是在多线程的环境运行的，如果该时刻有两对夫妇入住了，产生了两笔订单，恰好这两笔订单被两个线程所处理，当线程A将该男士计算到结果中，正要打算将该女士计算到结果中的时候，线程B已经计算完结果了，那么线程B计算出的结果就是2/1，那就出错啦。

所以为了保证数据在多线程处理时数据计算的正确性，我们需要用到**分布式锁**。实现分布式锁的方式有很多，本文就不赘述了。这里给大家介绍一种更简单快捷的方法。Cellar 中有个 setNx 函数，该函数是原子的，并且是（Set If Not Exists），所以用该函数锁住关键的字段就可以。就上面的例子而言，我们可以锁住该旅馆的唯一 ID 字段,计算完之后 delete 该锁，这样就可以保证了计算的正确性。

另外一个重要的问题是 **Cellar 不支持事务，就会导致该计算系统在升级或者重启时会造成少量数据的不准确**。为了解决该问题，运用到一种 getset 原子思想的方法。如下：

```
public void doSomeWork(String input) {
    cellar.mapPut("uniq_ID");
    cellar.add("uniq_ID_1","some data");
    cellar.add("uniq_ID_2","some data again");
    ....
    cellar.mapRemove("uniq_ID");
}
```

如果上述代码执行到[2..5]某一行时系统重启了，导致后续的操作并没有完成，如何将没有完成的操作添加上去呢？如下：

```
public void remedySomething() {
        map = cellar.mapGetAll();
        version = cellar.mapGet("uniq_ID").getVersion();
        for (string str : map) {
                if (cellar.get(str + "_1").getVersion()!= version) {
                        cellar.add(str + "_1", "some data");
                        cellar.mapRemove(str);
                }
                .......
        }
}
```

正如代码里那样，会有一个容器记录了哪些数据正在被操作，当系统重启的时候，从该容器取出上次未执行完的数据，用 Version（版本号）来记录哪些操作还没有完成，将没有完成的操作补上，这样就可以保证了计算结果的准确性。起初 Version（版本号）被设计出来解决的问题是防止由于数据的并发更新导致的问题。

> 比如，系统有一个 value 为“a,b,c”，A和B同时get到这个 value。A执行操作，在后面添加一个d，value 为 “a,b,c,d”。B执行操作添加一个e，value为”a,b,c,e”。如果不加控制，无论A和B谁先更新成功，它的更新都会被后到的更新覆盖。Tair 无法解决这个问题，但是引入了version机制避免这样的问题。还是拿刚才的例子，A和B取到数据，假设版本号为10，A先更新，更新成

> 功后，value 为”a,b,c,d”，与此同时，版本号会变为11。当B更新时，由于其基于的版本号是10，服务器会拒绝更新，从而避免A的更新被覆盖。B可以选择 get 新版本的 value，然后在其基础上修改，也可以选择强行更新。

将 Version 运用到事务的解决上也算是一种新型的使用。为验证该功能的正确性，灰度发布期间每天不同时段对项目进行杀死并重启，并对数据正确性进行校验，数据的正确性为100%。

实时数据存储

为了契合更多的需求，将数据分为三部分存储。

Kafka：存储稍加工之后的明细数据，方便做更多的扩展。 **MySQL**：存储中间的计算结果数据，方便计算过程的可视化。 **Cellar**：存储最终的结果数据，供应用层直接查询使用。

应用案例

1. **美团开店宝的实时经营数据卡片**。

美团开店宝作为美团商家的客户端，支持着众多餐饮商家的辅助经营，而经营数据的实时性对影响商家决策尤为重要。该功能上线之后受到了商家的热烈欢迎。卡片展示如下图：

1. **美团点评金融合作门店的实时热度标签**。

该功能用于与美团点评金融合作商家增加支付标签，用以突出这些商家，增加营销点。另一方面为优质商家吸引更多流量，为平台带来更多收益。展示如下图：

总结与展望

以上就是该系统的设计框架与思路，并且部分功能已应用到系统中。为了商家更好的决策，用户更好的体验，在业务不断增长的情况下，对实时数据的分析就需要做到更全面。所以实时数据分析还有很多东西可以去做。

老生常谈的大数据 4V+1O 特征，即数据量大(Volume)、类型繁多(Variety)、价值密度低(Value)、速度快时效性高(Velocity)、数据在线(Online)，相比离线数据系统，对实时数据的计算和应用挑战尤其艰巨。在技术框架演进层面，对流式数据进行高度抽象，简化开发流程；在应用端，我们后续希望在数据大屏、用户行为分析产品、营销效果跟踪等 DW/BI 产品进行持续应用，通过加快数据流转的速度，更好的发挥数据价值。

参考

- 多元线性回归模型

关于我们

到餐数据团队，用业内最先进的理念建设数据相关的系统和应用，期待更多数据系统开发、数据仓库开发、数据建模好手的加入。 发邮件给liuqiang24@meituan.com 、xuyang14@meituan.com 、xuyang14@meituan.com。

全链路压测平台（Quake）在美团中的实践

作者: 耿杰

背景

在美团的价值观中，“以客户为中心”被放在一个非常重要的位置，所以我们对服务出现故障越来越不能容忍。特别是目前公司业务正在高速增长阶段，每一次故障对公司来说都是一笔非常不小的损失。而整个IT基础设施非常复杂，包括网络、服务器、操作系统以及应用层面都可能出现问题。在这种背景下，我们必须对服务进行一次全方位的“体检”，从而来保障美团多个业务服务的稳定性，提供优质的用户服务体验。真正通过以下技术手段，来帮助大家吃的更好，生活更好：

- 验证峰值流量下服务的稳定性和伸缩性。
- 验证新上线功能的稳定性。
- 进行降级、报警等故障演练。
- 对线上服务进行更准确的容量评估。
- ……

全链路压测是基于线上真实环境和实际业务场景，通过模拟海量的用户请求，来对整个系统进行压力测试。早期，我们在没有全链路压测的情况下，主要的压测方式有：

- 对线上的单机或集群发起服务调用。
- 将线上流量进行录制，然后在单台机器上进行回放。
- 通过修改权重的方式进行引流压测。

但以上方式很难全面的对整个服务集群进行压测，如果以局部结果推算整个集群的健康状况，往往会“以偏概全”，无法评估整个系统的真实性能水平，主要的原因包括：

- 只关注涉及的核心服务，无法覆盖到所有的环节。
- 系统之间都是通过一些基础服务进行串联，如 Nginx、Redis 缓存、数据库、磁盘、网络等等，而基础服务问题在单服务压测中往往不能被暴露出来。

综合多种因素考虑，全链路压测是我们准确评估整个系统性能水平的必经之路。目前，公司内所有核心业务线都已接入全链路压测，月平均压测次数达上万次，帮助业务平稳地度过了大大小小若干场高峰流量的冲击。

解决方案

Quake （雷神之锤）作为公司级的全链路压测平台，它的目标是提供对整条链路进行全方位、安全、真实的压测，来帮助业务做出更精准的容量评估。因此我们对 Quake 提出了如下的要求：

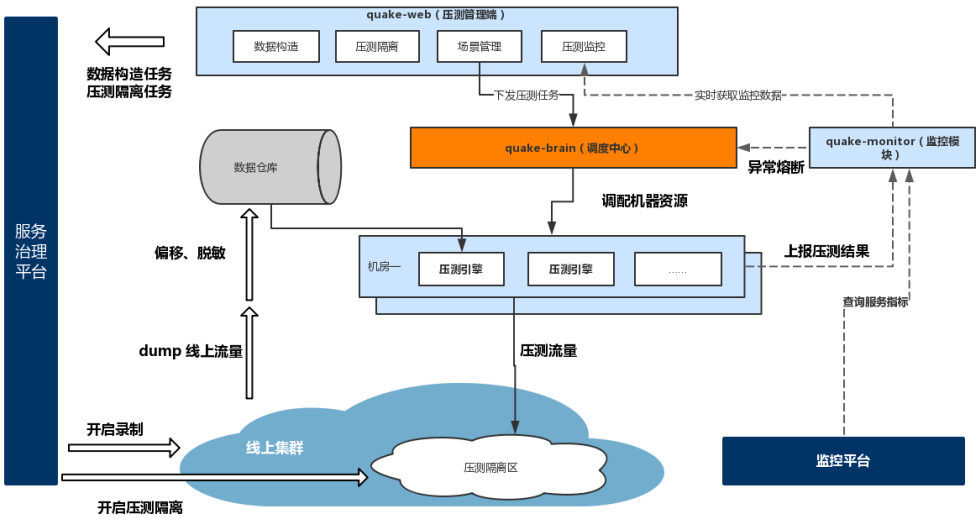
- 提供模拟线上真实流量的能力
  - 压测和 DDoS 攻击不同的是，压测有应用场景，而 DDoS 可能只需要一个请求。为了更真实的还原用户行为，我们需要获取线上的真实流量进行压测。

- 具备快速创建压测环境的能力
  - 这里的环境指的是线上环境，因为如果压测的是线下环境，即使不考虑“机器配置是否相同”这个因素，像集群规模、数据库体量、网络条件等这些因素，在线下环境下都无法进行模拟，这样得出压测结果，其参考价值并不大。
- 支持多种压测类型
  - 压测类型除了支持标准的 HTTP 协议，还需要对美团内部的 RPC 和移动端协议进行支持。
- 提供压测过程的实时监控与过载保护
  - 全链路压测是一个需要实时关注服务状态的过程，尤其在探测极限的时候，需要具备精准调控 QPS 的能力，秒级监控的能力，预设熔断降级的能力，以及快速定位问题的能力。

Quake 整体架构设计

Quake 集数据构造、压测隔离、场景管理、动态调控、过程监控、压测报告为一体，压测流量尽量模拟真实，具备分布式压测能力的全链路压测系统，通过模拟海量用户真实的业务操作场景，提前对业务进行高压力测试，全方位探测业务应用的性能瓶颈，确保平稳地应对业务峰值。

架构图：



Quake 整体架构上分为：

- Quake-Web：压测管理端，负责压测数据构造、压测环境准备、场景管理、压测过程的动态调整以及压测报表展示等。
- Quake-Brain：调度中心，负责施压资源的调度、任务分发与机器资源管理。
- Quake-Agent：压测引擎，负责模拟各种压测流量。
- Quake-Monitor：监控模块，统计压测结果，监控服务各项指标。

管理端核心功能

数据构造

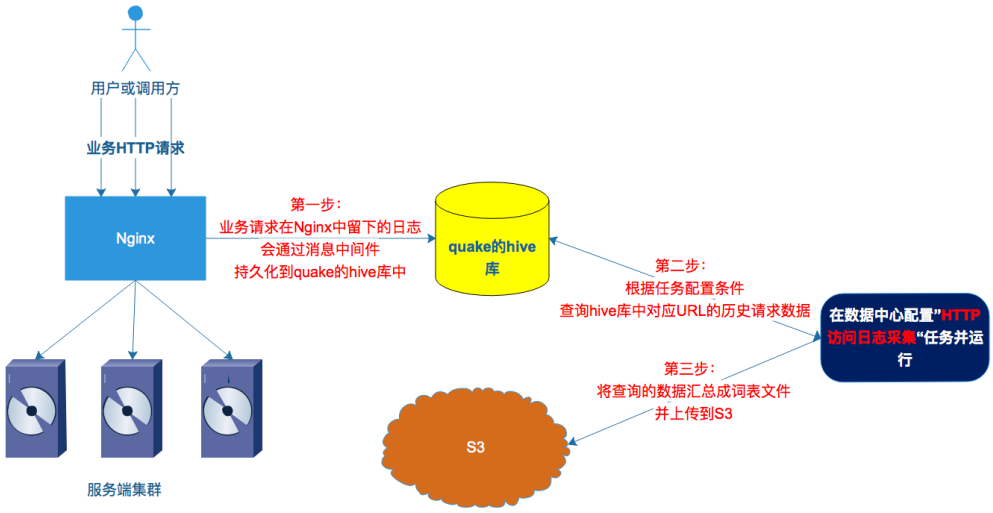
传统的数据构造，一般由测试人员自己维护一批压测数据。但这种方式存在很大的弊端，一方面维护成本相对较高，另一方面，其构造出的数据多样性也不足够。在真实业务场景中，我们需要的是能直接回放业

务高峰期产生的流量，只有面对这样的流量冲击，才能真实的反映系统可能会产生的问题。

Quake 主要提供了 HTTP 和 RPC 的两种数据构造方式：

HTTP 服务的访问日志收集

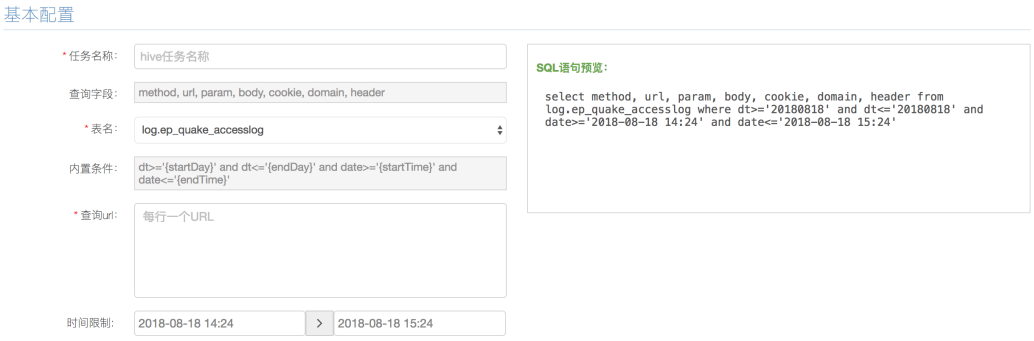
对于 HTTP 服务，在 Nginx 层都会产生请求的访问日志，我们对这些日志进行了统一接入，变成符合压测需要的流量数据。架构图如下：



S3为最终日志存储平台

底层使用了 Hive 作为数仓的工具，使业务在平台上可以通过简单的类 SQL 语言进行数据构造。Quake 会从数仓中筛选出相应的数据，作为压测所需的词表文件，将其存储在 S3 中。

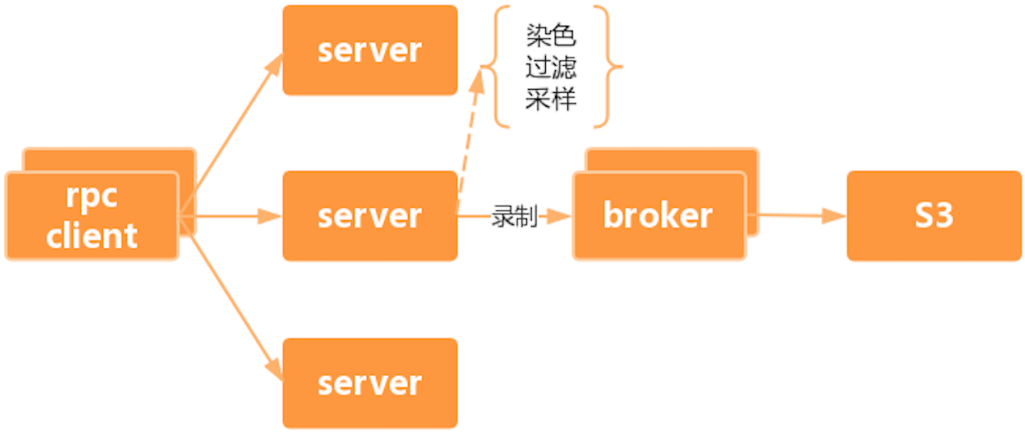
词表：压测所需的元数据，每一行代表一个请求，包含请求的 method、path、params、header、body等等。



RPC 线上流量实时录制

对于 RPC 服务，服务调用量远超 HTTP 的量级，所以在线上环境不太可能去记录相应的日志。这里我们使用对线上服务进行实时流量录制，结合 RPC 框架提供的录制功能，对集群中的某几台机器开启录制，

根据要录制的接口和方法名，将请求数据上报到录制流量的缓冲服务（Broker）中，再由 Broker 生成最终的压测词表，上传到存储平台（S3）。



- RPC Client：服务的调用方
- Server：服务提供方
- Broker：录制后流量缓冲服务器
- S3：流量最终存储平台

其他优化：

流量参数偏移

有些场景下，构造出来的流量是不能直接使用的，我们需要对用户 ID、手机号等信息进行数据偏移。Quake 也是提供了包含四则运算、区间限定、随机数、时间类型等多种替换规则。

词表文件的分片

数据构造产生的词表文件，我们需要进行物理上的分片，以保证每个分片文件大小尽可能均匀，并且控制在一定大小之内。这么做的主要原因是，后续压测肯定是由一个分布式的压测集群进行流量的打入，考虑到单机拉取词表的速度和加载词表的大小限制，如果将词表进行分片的话，可以有助于任务调度更合理的进行分配。

压测隔离

做线上压测与线下压测最大不同在于，线上压测要保证压测行为安全且可控，不会影响用户的正常使用，并且不会对线上环境造成任何的数据污染。要做到这一点，首要解决的是压测流量的识别与透传问题。有了压测标识后，各服务与中间件就可以依据标识来进行压测服务分组与影子表方案的实施。

测试标识透传

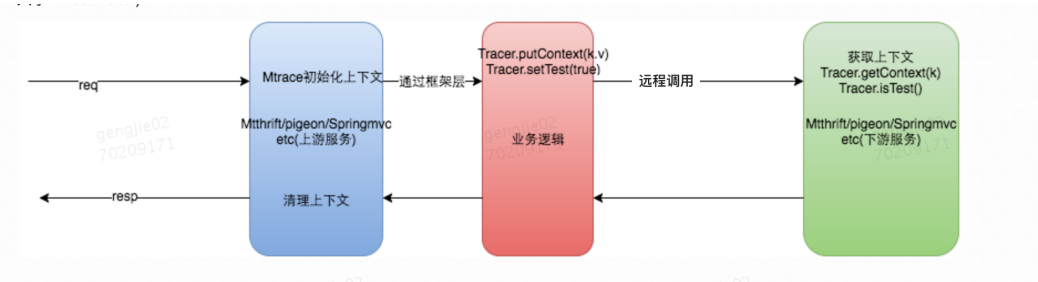
对于单服务来说，识别压测流量很容易，只要在请求头中加个特殊的压测标识即可，HTTP 和 RPC 服务是一样的。但是，要在整条完整的调用链路中要始终保持压测标识，这件事就非常困难。

跨线程间的透传：

对于涉及多线程调用的服务来说，要保证测试标识在跨线程的情况下不丢失。这里以 Java 应用为例，主线程根据压测请求，将测试标识写入当前线程的 ThreadLocal 对象中（**ThreadLocal 会为每个线程创建一个副本，用来保存线程自身的副本变量**），利用 InheritableThreadLocal 的特性，对于父线程 ThreadLocal 中的变量会传递给子线程，保证了压测标识的传递。而对于采用线程池的情况，同样对线程池进行了封装，在往线程池中添加线程任务时，额外保存了 ThreadLocal 中的变量，执行任务时再进行替换 ThreadLocal 中的变量。

跨服务间的透传：

对于跨服务的调用，架构团队对所有涉及到的中间件进行了一一改造。利用 Mtrace （**公司内部统一的分布式会话跟踪系统**）的服务间传递上下文特性，在原有传输上下文的基础上，添加了测试标识的属性，以保证传输中始终带着测试标识。下图是 Mtrace 上下游调用的关系图：



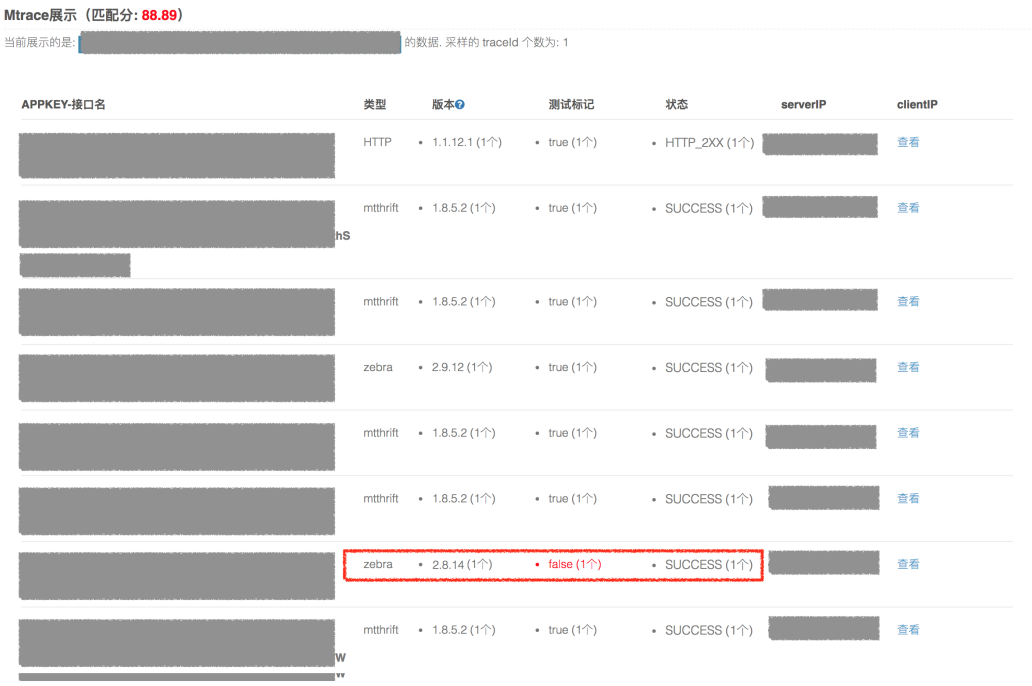
链路诊断

由于链路关系的复杂性，一次压测涉及的链路可能非常复杂。很多时候，我们很难确认间接依赖的服务又依赖了哪些服务，而任何一个环节只要出现问题，比如某个中间件版本不达标，测试标识就不会再往下进行透传。Quake 提供了链路匹配分析的能力，通过平台试探性地发送业务实际需要压测的请求，根据Mtrace提供的数据，帮助业务快速定位到标记透传失败的服务节点。

链路诊断总览

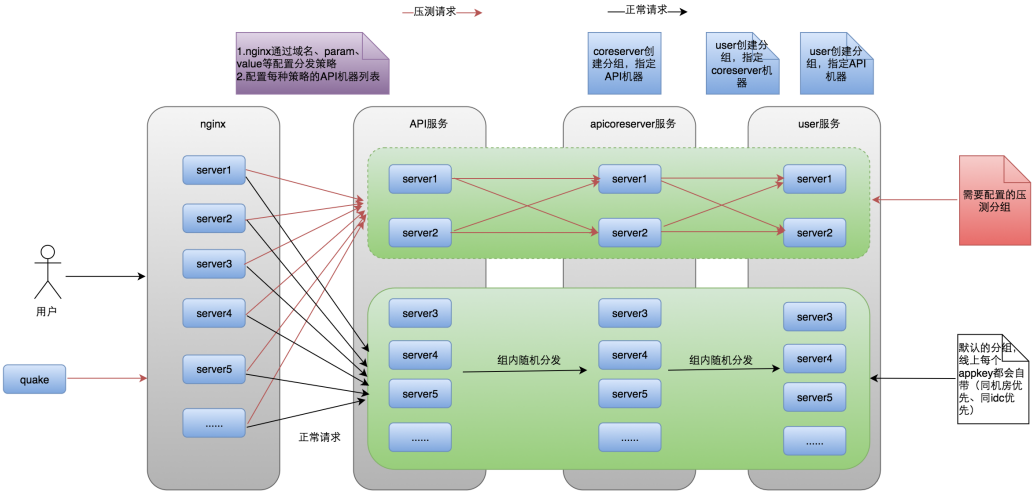
路径	匹配分	最新qps	响应时间(ms)					错误率(%)	请求量	占比
			平均	top50	top90	top99	top999			
[redacted]	100	1	135	135	135	135	135	0	1	0.5
[redacted]od	88.89	1	73	70	82	84	84	0	1	0.5

链路诊断详情定位



压测服务隔离

一些大型的压测通常选择在深夜低峰时期进行，建议相关的人员要时刻关注各自负责的系统指标，以免影响线上的正常使用。而对于一些日常化的压测，Quake 提供了更加安全便捷的方式进行。在低峰期，机器基本都是处于比较空闲的状态。我们将根据业务的需求在线上对整条链路快速创建一个压测分组，隔出一批空闲的机器用于压测。将正常流量与测试流量在机器级别进行隔离，从而降低压测对服务集群带来的影响。

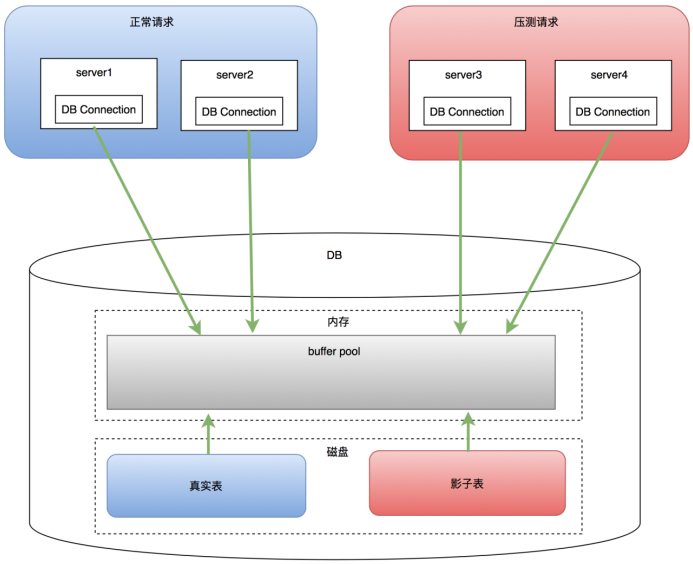


依赖标识透传的机制，在 Quake 平台上提供了基于 IP、机器数、百分比不同方式的隔离策略，业务只需提供所需隔离的服务名，由 Quake 进行一键化的开启与关闭。

压测数据隔离

还有一个比较棘手的问题是针对写请求的压测，因为它会向真实的数据库中写入大量的脏数据。我们借鉴了阿里最早提出的“影子表”隔离的方案。“影子表”的核心思想是，使用线上同一个数据库，包括共享数

据库中的内存资源，因为这样才能更接近真实场景，只是在写入数据时会写在了另一张“影子表”中。



对于 KV 存储，也是类似的思路。这里讲一下 MQ（消息队列）的实现，MQ 包括生产和消费两端，业务可以根据实际的需要选择在生产端忽略带测试标识的消息，或者在消费端接收消息后再忽略两种选择。

调度中心核心设计

调度中心作为整个压测系统的大脑，它管理了所有的压测任务和压测引擎。基于自身的调度算法，调度中心将每个压测任务拆分成若干个可在单台压测引擎上执行的计划，并将计划以指令的方式下发给不同的引擎，从而执行压测任务。

资源计算

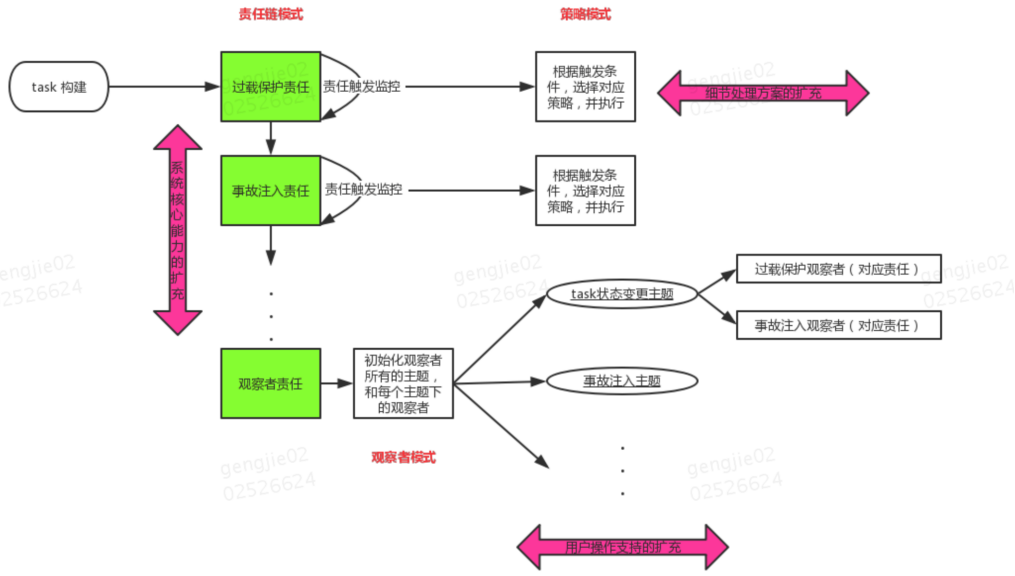
不同的压测场景，需要的机器资源不一样。以 HTTP 服务为例，在请求/响应体都在 1K 以内，响应时间在 50ms 以内和 1s 左右的两个请求，单个施压机能达到的极限值完全不同。影响压测能力的因素有很多，计算中心会依据压测模型的不同参数，进行资源的计算。

主要参考的数据包括：

- 压测期望到达的 QPS。
- 压测请求的平均响应时间和请求/响应体大小。
- 压测的词表大小、分片数。
- 压测类型。
- 所需压测的机房。

事件注入机制

因为整个压测过程一直处在动态变化之中，业务会根据系统的实际情况对压力进行相应的调整。在整个过程中产生的事件类型比较多，包括调整 QPS 的事件、触发熔断的事件、开启事故注入、开启代码级性能分析的事件等等，同时触发事件的情况也有很多种，包括用户手动触发、由于系统保护机制触等等。所以，我们在架构上也做了相应的优化，其大致架构如下：



在代码设计层面，我们采用了观察者和责任链模式，将会触发事件的具体情况作为观察主题，主题的订阅者会视情况类型产生一连串执行事件。而在执行事件中又引入责任链模式，将各自的处理逻辑进行有效的拆分，以便后期进行维护和能力扩充。

机器管理

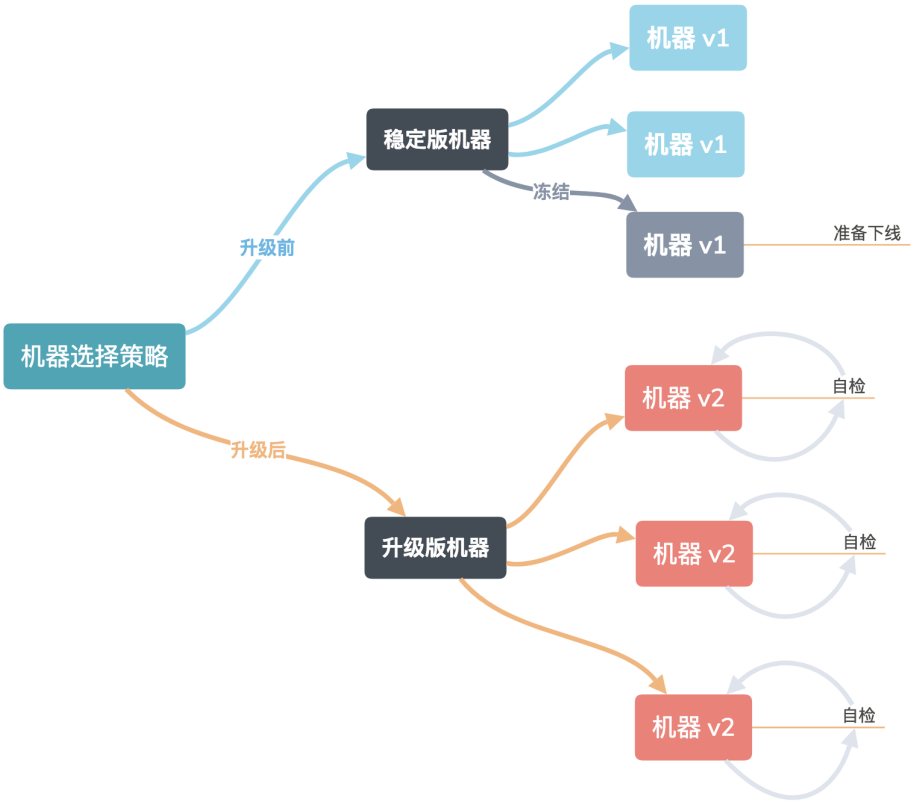
调度中心管理了所有的施压机资源，这些施压机分布在北京、上海的多个机房，施压机采用容器化方式进行部署，为后续的动态扩容、施压机灰度升级以及异常摘除的提供了基础保障。

动态扩容

业务对压测的需求有高低峰之分，所以平台也需要事先部署一部分机器用于日常的业务压测。当业务申请资源不足时，平台会按需通过容器化方式动态的进行扩容。这样做的好处，一方面是节省机器资源，另一方面就是便于升级。不难想象，升级50台机器相对升级200台机器，前者付出的代价肯定更小一些。

灰度升级

整个机器池维护着几百台机器，如果需要对这些机器进行升级操作，难度系数也比较高。我们以前的做法是，在没有业务压测的时候，将机器全部下线，然后再批量部署，整个升级过程既耗时又痛苦。为此，我们引入了灰度升级的概念，对每台施压机提供了版本的概念，机器选择时，优先使用稳定版的机器。根据机器目前使用的状态，分批替换未使用的机器，待新版本的机器跑完基准和回归测试后，将机器选择的策略改为最新版。通过这种方式，我们可以让整个升级过程，相对平顺、稳定，且能够让业务无感知。



异常摘除

调度中心维持了与所有施压机的心跳检测，对于异常节点提供了摘除替换的能力。机器摘除能力在压测过程中非常有必要，因为压测期间，我们需要保证所有的机器行为可控。不然在需要降低压力或停止压测时，如果施压机不能正常做出响应，其导致的后果将会非常严重。

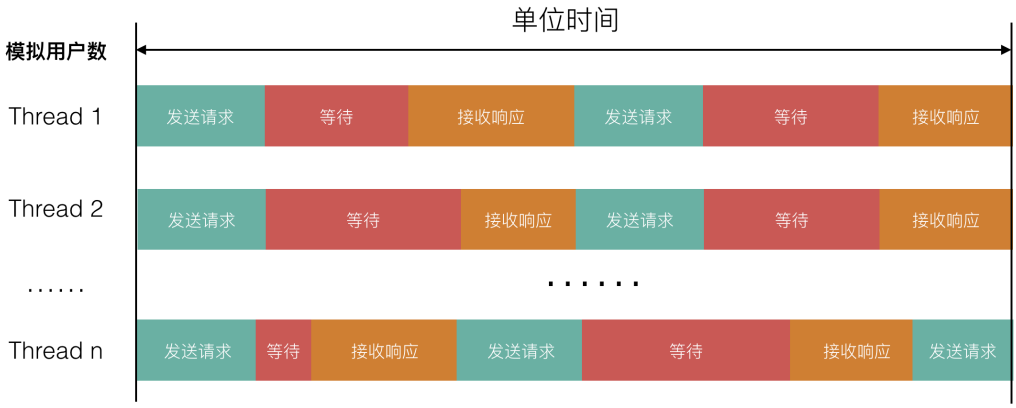
压测引擎优化

在压测引擎的选择上，Quake 选择了自研压测引擎。这也是出于扩展性和性能层面的考虑，特别在扩展性层面，主要是对各种协议的支持，这里不展开进行阐述。性能方面，为了保证引擎每秒能产生足够多的请求，我们对引擎做了很多性能优化的工作。

性能问题

通常的压测引擎，采用的是 BIO 的方式，利用多线程来模拟并发的用户数，每个线程的工作方式是：请求–等待–响应。

通信图：



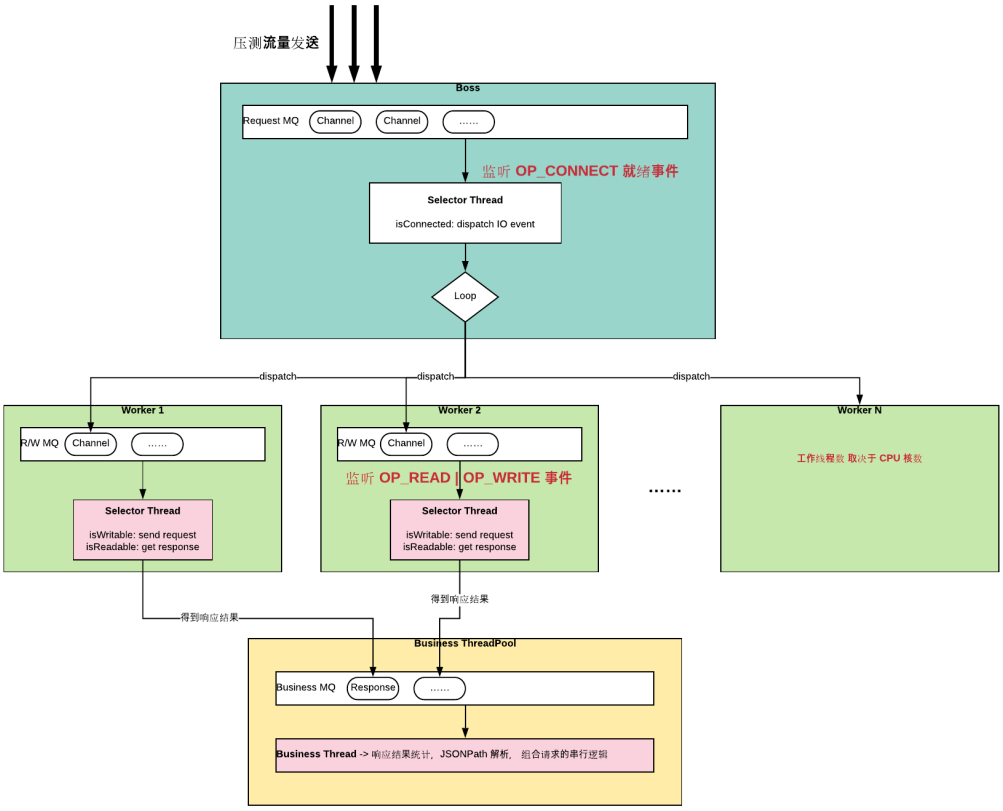
这种方式主要的问题是，中间的等待过程，线程资源完全被浪费。这种组合模式下，性能问题也会更严重（**组合模式：即模拟用户一连串的用户行为，以下单为例，请求组中会包含用户登录、加入购物车、创建订单、支付订单、查看支付状态。这些请求彼此间是存在先后关系的，下一个请求会依赖于上一个请求的结果。**），若请求组中有5个串联请求，每个请求的时长是200ms，那完成一组请求就需要 1s 。这样的话，单机的最大 QPS 就是能创建的最大线程数。我们知道机器能创建的线程数有限，同时线程间频繁切换也有成本开销，致使这种通信方式能达到的单机最大 QPS 也很有限。

这种模型第二个问题是，线程数控制的粒度太粗，如果请求响应很快，仅几十毫秒，如果增加一个线程，可能 QPS 就上涨了将近100，通过增加线程数的方式无法精准的控制 QPS，这对探测系统的极限来说，十分危险。

IO 模型优化

我们先看下 NIO 的实现机制，从客户端发起请求的角度看，存在的 IO 事件分别是建立连接就绪事件（OP_CONNECT）、IO 就绪的可读事件 （OP_READ） 和 IO 就绪的可写事件（OP_WRITE），所有 IO 事件会向事件选择器（Selector）进行注册，并由它进行统一的监听和处理，Selector 这里采用的是 IO 多路复用的方式。

在了解 NIO 的处理机制后，我们再考虑看如何进行优化。整个核心思想就是根据预设的 QPS，保证每秒发出指定数量的请求，再以 IO 非阻塞的方式进行后续的读写操作，取消了 BIO 中请求等待的时间。优化后的逻辑如下：



优化一：采用 Reactor 多线程模型

这里主要耗时都在 IO 的读写事件上，为了达到单位时间内尽可能多的发起压测请求，我们将连接事件与读写事件分离。连接事件采用单线程 Selector 的方式来处理，读写事件分别由多个 Worker 线程处理，每个 Worker 线程也是以 NIO 方式进行处理，由各自的 Selector 处理 IO 事件的读写操作。这里每个 Worker 线程都有自己的事件队列，数据彼此隔离，这样做主要是为了避免数据同步带来的性能开销。

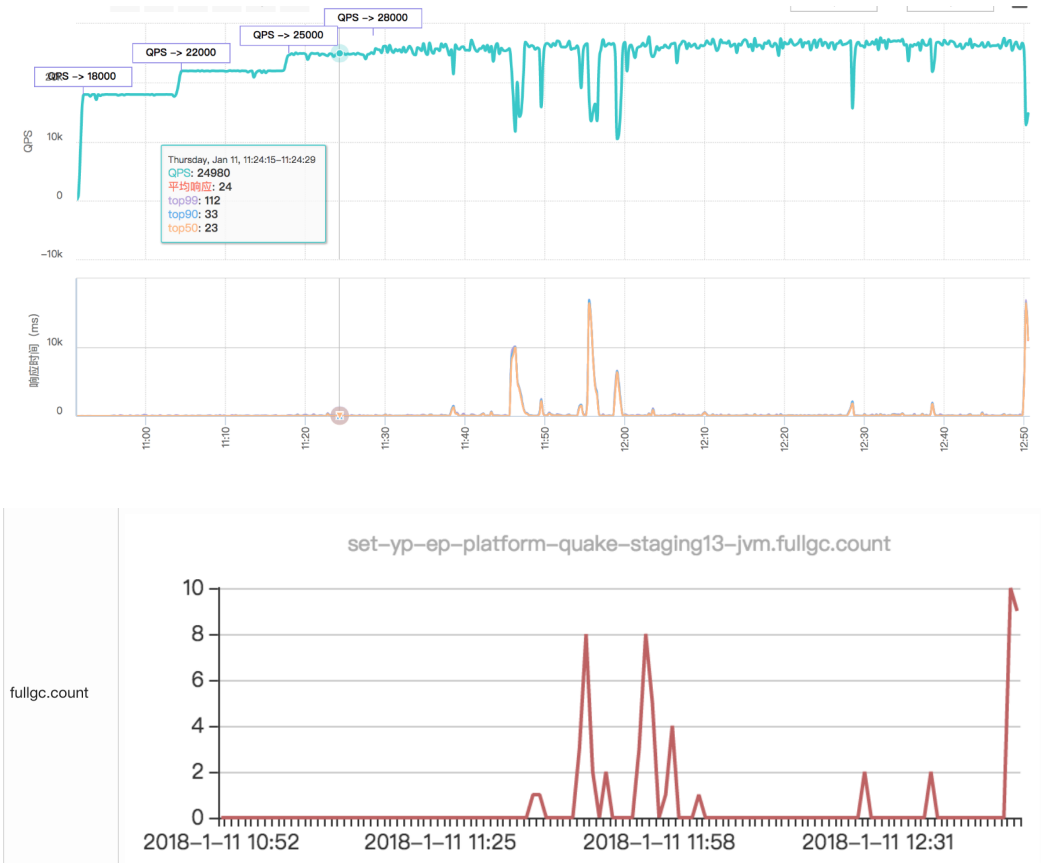
优化二：业务逻辑与 IO 读写事件分离

这里说的业务逻辑主要是针对请求结果的处理，包括对请求数据的采样上报，对压测结果的解析校验，对请求转换率的匹配等。如果将这些逻辑放在 Worker 线程中处理，必然会影响 IO 读取的速度。因为 Selector 在监听到 IO 就绪事件后，会进行单线程处理，所以它的处理要尽可能的简单和快速，不然会影响其他就绪事件的处理，甚至造成队列积压和内存问题。

内存优化

压测引擎另一个重要的指标是 Full GC 的时间，因为如果引擎频繁出现 Full GC，那会造成实际压测曲线（QPS）的抖动，这种抖动会放大被压服务真实的响应时间，造成真实 QPS 在预设值的上下波动。严重的情况，如果是长时间出现 Full GC，直接就导致预压的 QPS 压不上去的问题。

下面看一组 Full GC 产生的压测曲线：



为了解决 GC 的问题，主要从应用自身的内存管理和 JVM 参数两个维度来进行优化。

合理分配内存对象

请求对象加载机制优化

引擎首先加载词表数据到内存中，然后根据词表数据生成请求对象进行发送。对于词表数据的加载，需要设置一个大小上限，这些数据是会进入“老年代”，如果“老年代”占用的比例过高，那就会频发出现 Full GC 的情况。这里对于词表数据过大的情况，可以考虑采用流式加载的方式，在队列中维持一定数量的请求，通过边回放边加载的方式来控制内存大小。

请求对象的快用快销

引擎在实际压测过程中，假设单机是 1W 的 QPS，那它每秒就会创建 1W 个请求对象，这些对象可能在下一秒处理完后就会进行销毁。如果销毁过慢，就会造成大量无效对象晋升老年代，所以在对响应结果的处理中，不要有耗时的操作，保证请求对象的快速释放。

这里放弃对象复用的原因是，请求的基本信息占用的内存空间比较小。可一旦转换成了待发送对象后，占用的内存空间会比原始数据大很多，在 HTTP 和 RPC 服务中都存在同样的问题。而且之前使用 Apache HttpAsyncClient 作为 HTTP 请求的异步框架时，发现实际请求的 Response 对象挂在请求对象身上。也就是说一个请求对象在接收到结果后，该对象内存增加了响应结果的空间占用，如果采用复用请求对象的方式，很容易造成内存泄露的问题。

JVM 参数调优

这里以 JVM 的 CMS 收集器为例，对于高并发的场景，瞬间产生大量的对象，这些对象的存活时间又非常短，我们需要：

- 适当增大新生代的大小，保证新生代有足够的空间来容纳新产生的对象。当然如果老年代设置的过小，会导致频繁的 Full GC。
- 适当调大新生代向晋升老年代的存活次数，减少无效对象晋升老年代的机率；同时控制新生代存活区的大小，如果设置的过小，很容易造成那些无法容纳的新生代对象提前晋升。
- 提前触发老年代的 Full GC，因为如果等待老年代满了再开始回收，可能会太晚，这样很容易造成长时间的 Full GC。一般设在 70% 的安全水位进行回收。而且回收的时候，需要触发一次 Young GC，这可以减少重新标记阶段应用暂停的时间，另一方面，也防止在回收结束后，有大量无效的对象进入老年代中。
- 设置需要进行内存压缩整理的 GC 次数，内存整理，很多时候是造成长时间 GC 的主要原因。因为内存整理是采用 Serial Old 算法，以单线程的方式进行处理，这个过程会非常慢。尤其是在老年代空间不足的情况下，GC 的时间会变得更长。

监控模块

压测肯定会对线上服务产生一定的影响，特别是一些探测系统极限的压测，我们需要具备秒级监控的能力，以及可靠的熔断降级机制。

客户端监控

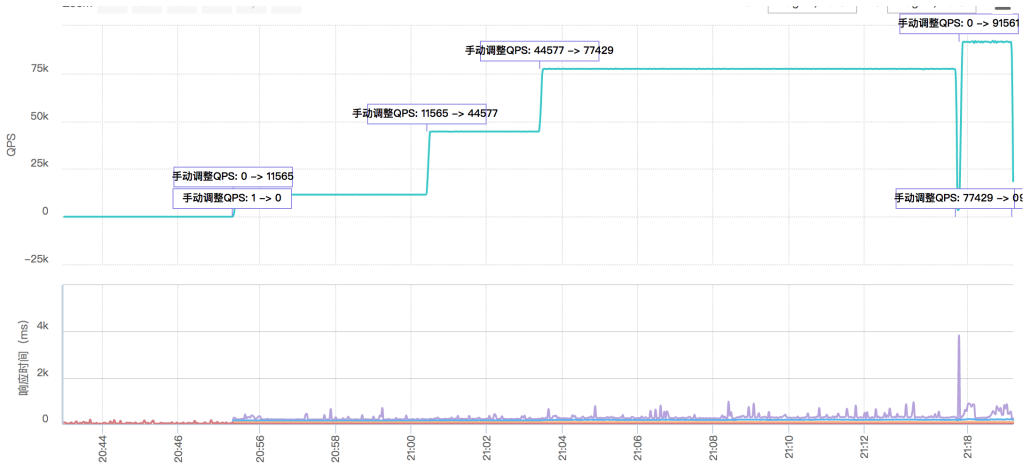
压测引擎会将每秒的数据汇总后上报给监控模块，监控模块基于所有上报来的数据进行统计分析。这里的分析需要实时进行处理，这样才能做到客户端的秒级监控。监控的数据包括各 TP 线的响应情况、QPS 曲线波动、错误率情况以及采样日志分析等等。

实时 QPS 曲线

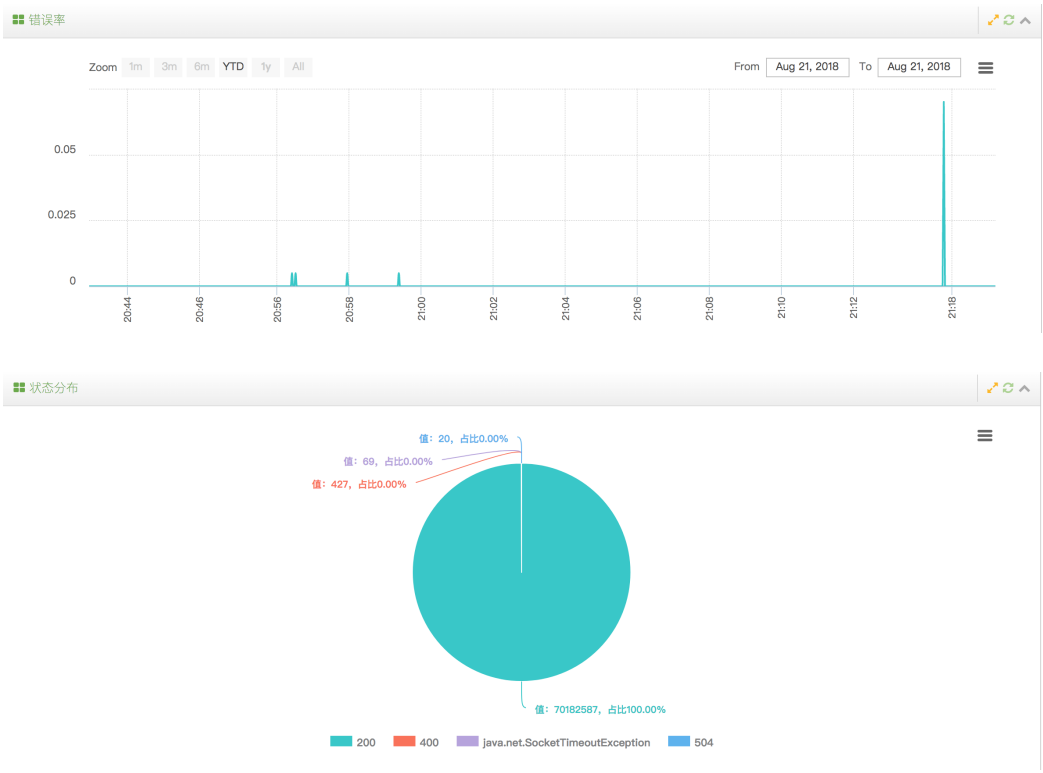
Display 10 records　　Search:

路径	最新qps	响应时间(ms)					错误率(%)	请求量	占比
		平均	top50	top90	top99	top999			
	10,940	20	19	25	60	220	0	8,443,422	0.12
	8,267	12	11	15	40	200	0	6,337,288	0.09
	6,017	15	13	19	55	220	0	4,510,927	0.06
	4,753	96	90	135	200	500	0	3,658,306	0.05
	4,515	21	20	25	65	220	0	3,543,791	0.05
	4,024	30	30	50	85	220	0.01	3,077,333	0.04
	3,936	12	11	15	40	200	0	2,994,137	0.04
	3,541	8	8	10	35	200	0	2,706,864	0.04
	3,105	135	130	185	260	440	0	2,358,786	0.03
	2,611	12	10	15	55	200	0	2,045,277	0.03

Showing 1 to 10 of 83 entries　　Previous 1 2 3 4 5 … 9 Next



错误率统计



采样日志

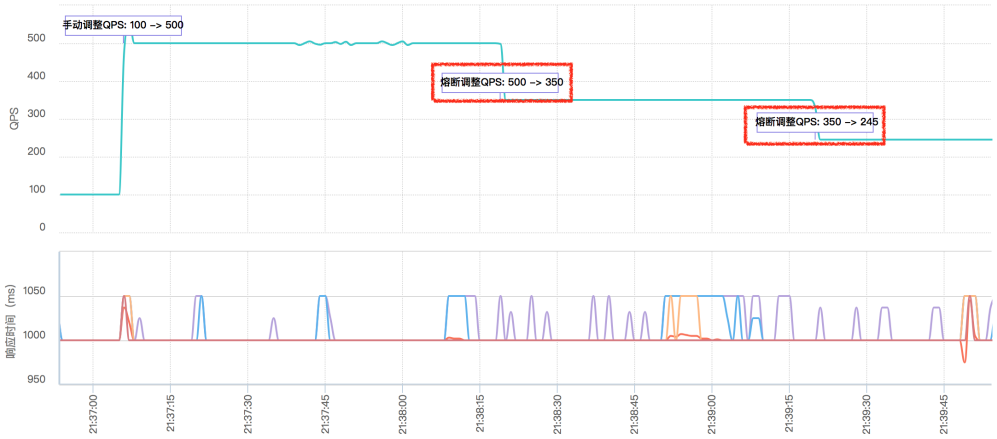
服务端监控

除了通过引擎上报的压测结果来进行相应的监控分析之外，Quake 还集成了公司内部统一的监控组件，有监控机器指标的 Falcon 系统（小米开源），还有监控服务性能的 CAT系统 （美团已经开源）。Quake 提供了统一的管理配置服务，让业务能在 Quake 上方便观察整个系统的健康状况。

熔断保护机制

Quake 提供了客户端和服务端两方面的熔断保护措施。

首先是客户端熔断，根据业务自定义的熔断阈值，Quake 会实时分析监控数据，当达到熔断阈值时，任务调度器会向压测引擎发送降低 QPS 或者直接中断压测的指令，防止系统被压挂。



被压服务同样也提供了熔断机制，Quake 集成了公司内部的熔断组件（Rhino），提供了压测过程中的熔断降级和限流能力。与此同时，Quake 还提供了压测故障演练的能力，在压测过程中进行人为的故障注入，来验证整个系统的降级预案。

项目总结

最后，总结一下做 Quake 这个项目的一些心得。

小步快跑

其实在 Quake 出来之前，美团公司内部已有一个压测平台（Ptest ），它的定位是针对单服务的性能压测。我们分析了 Ptest 平台存在的一些问题，其压测引擎能力也非常有限。在美团发展早期，如果有两个大业务线要进行压测的话，机器资源往往会不足，这需要业务方彼此协调。因为准备一次压测，前期投入成本太高，用户需要自己构造词表，尤其是 RPC 服务，用户还需要自己上传 IDL 文件等等，非常繁琐。

Quake 针对业务的这些痛点，整个团队大概花费一个多月的时间开发出了第一个版本，并且快速实现了上线。当时，正面临猫眼十一节前的一次压测，那也是 Quake 的第一次亮相，而且取得了不错的成绩。后续，我们基本平均两周实现一次迭代，然后逐步加入了机器隔离、影子表隔离、数据偏移规则、熔断保护机制、代码级别的性能分析等功能。

快速响应

项目刚线上时，客服面临问题非常多，不仅有使用层面的问题，系统自身也存在一些 Bug 缺陷。当时，一旦遇到业务线大规模的压测，我们团队都是全员待命，直接在现场解决问题。后续系统稳定后，我们组内采用了客服轮班制度，每个迭代由一位同学专门负责客服工作，保障当业务遇到的问题能够做到快速响

应。尤其是在项目上线初期，这点非常有必要。如果业务部门使用体验欠佳，项目口碑也会变差，就会对后续的推广造成很大的问题。

项目推广

这应该是所有内部项目都会遇到的问题，很多时候，推广成果决定项目的生死。前期我们先在一些比较有代表性的业务线进行试点。如果在试点过程中遇到的问题，或者业务同学提供的一些好的想法和建议，我们能够快速地进行迭代与落地。然后再不断地扩大试点范围，包括美团外卖、猫眼、酒旅、金融等几个大的 BG 都在 Quake 上进行了几轮全流程、大规模的全链路压测。

随着 Quake 整体功能趋于完善，同时解决了 Ptest（先前的压测系统）上的多个痛点，我们逐步在各个业务线进行了全面推广和内部培训。从目前收集的数据看，美团超过 90% 的业务已从 Ptest 迁移到了 Quake 。而且整体的统计数据，也比 Ptest 有了明显的提升。

开放生态

Quake 目标是打造全链路的压测平台，但是在平台建设这件事上，我们并没有刻意去追求。公司内部也有部分团队走的比较靠前，他们也做一些很多“试水性”的工作。这其实也是一件好事，如果所有事情都依托平台来完成，就会面临做不完的需求，而且很多事情放在平台层面，也可能无解。

同时，Quake 也提供了很多 API 供其他平台进行接入，一些业务高度定制化的工作，就由业务平台独自去完成。平台仅提供基础的能力和数据支持，我们团队把核心精力聚焦在对平台发展更有价值的事情上。

跨团队合作

其实，全链路压测整个项目涉及的团队非常之多，上述提到的很多组件都需要架构团队的支持。在跨团队的合作层面，我们应该有“双赢”的心态。像 Quake 平台使用的很多监控组件、熔断组件以及性能分析工具，有一些也是兄弟团队刚线上没多久的产品。 Quake 将其集成到平台中，一方面是减少自身重复造轮子；另一方面也可以帮助兄弟团队推动产品的研发工作。

作者简介

- 耿杰，美团点评高级工程师。2017年加入美团点评，先后负责全链路压测项目和 MagicDB 数据库代理项目，目前主要负责这两个项目的整体研发和推广工作，致力于提升公司的整体研发效率与研发质量。

招聘

团队长期招聘 Java、Go、算法、AI 等技术方向的工程师，Base 北京、上海，欢迎有兴趣的同学投递简历到gengjie02@meituan.com。

美团配送系统架构演进实践

作者: 永俊

写在前面

美团配送自成立以来，业务经历了多次跨越式的发展。业务的飞速增长，对系统的整体架构和基础设施提出了越来越高的要求，同时也不断驱动着技术团队深刻理解业务、准确定位领域模型、高效支撑系统扩展。如何在业务高速增长、可用性越来越高的背景下实现系统架构的快速有效升级？如何保证复杂业务下的研发效率与质量？本文将为大家介绍美团配送的一些思考与实践。

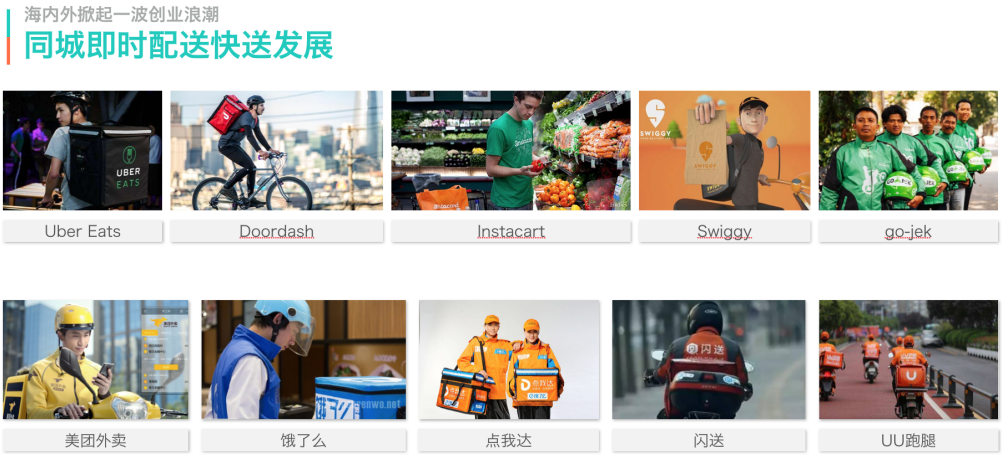
配送业务

从物流到同城即时配送

物流行业的发展离不开商业的发展，近些年，商业的变革为物流发展创造了新的机会。电商的兴起有效带动了快递行业的飞速发展，直接造就了顺丰、四通一达这样的快递公司。而近年来O2O商业模式的兴起，尤其是外卖、生鲜等到家场景的发展促进了同城即时配送的快速发展。

与物流领域下的其他分支不同，同城即时配送具有如下特点:

- **时效快**：美团外卖平均送达时间28min。
- **距离短**：配送距离多数为3~5km范围，较大的扩展到同城范围。
- **随机性强**：取货点、交付点具有时间与空间的随机性，预测与规划难度相对较高。



同城即时配送业务的发展契机

行业的流程再造一般离不开两个因素：

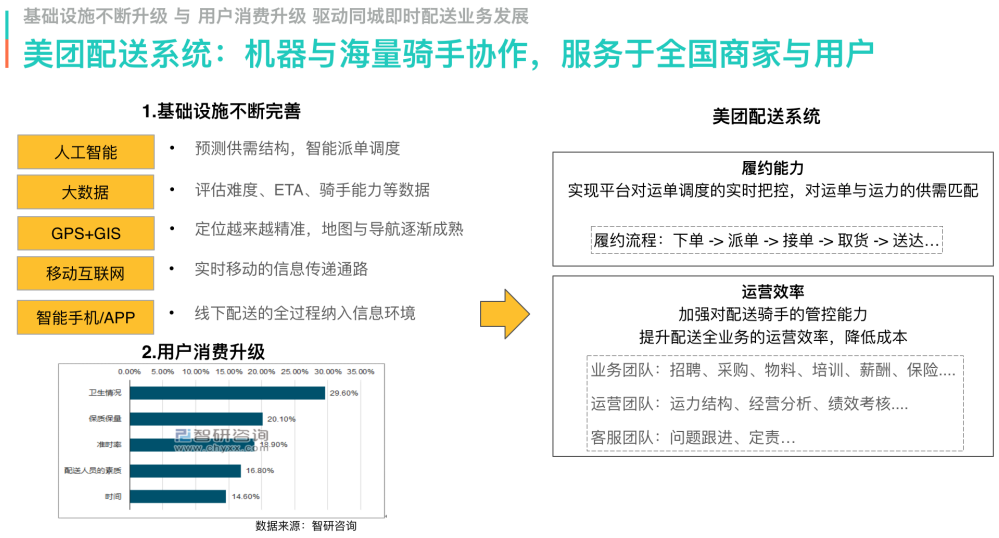
- 内因：技术或基础设施取得重大突破
- 外因：用户消费升级或市场发生重大变化

技术方面，AI与大数据的应用逐步普及，基于人工智能可以对配送难度、ETA、骑手能力精确评估。GPS的快速发展与GIS厂商能力的不断开放，使得基于LBS的应用大大降低了开发成本。基础设施方面，得益于国家的持续投入，移动网络的质量不断提升，成本逐年下降，也间接促使智能手机几乎实现了全民覆盖。

市场方面，由于中国人口具有超大规模性的特点，人群聚集度高，外卖等到家场景在各大城市尤其是一线城市的需求持续增强。用户对于外卖的安全、时效、配送员的服装、礼貌用语等都有更高的要求。

在这两个因素的共同作用下，促成了同城即时配送行业的发展。而对于同城即时配送业务而言，履约能力与运营效率是研发团队要重点解决的两个问题：

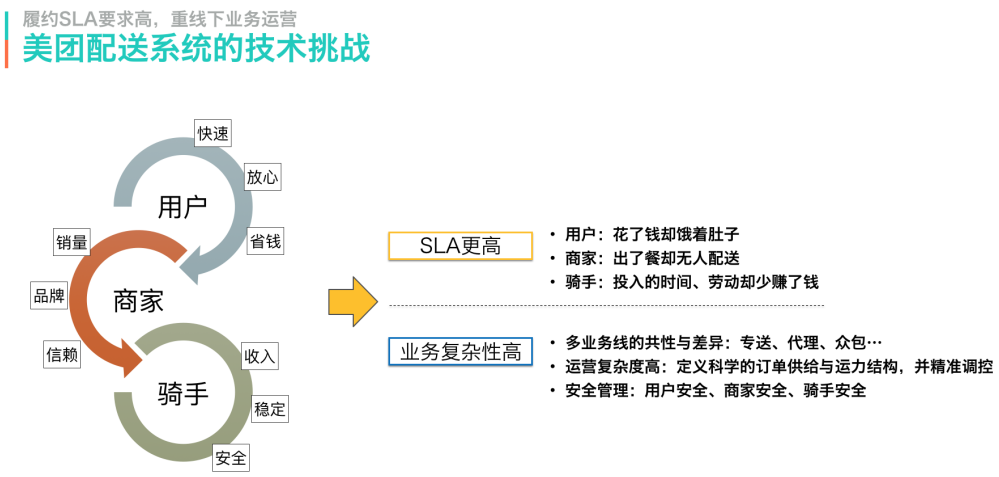
1. 履约能力保证：实现平台对运单调度的实时把控，具备供需调控能力。
2. 运营效率提升：加强对配送骑手的管控能力，提升配送全业务的运营效率，持续降低成本。



技术挑战

美团配送系统的本质——机器与海量骑手协作，服务于全国用户和商家的大规模协作系统。技术的挑战本质上源于业务的痛点，具体体现为**线上的强履约能力**要求与**线下的强运营能力**要求。技术上的挑战也同样来源于线上和线下两个方面：

- 线上履约的SLA要求更高。配送业务需要兼顾用户、商家、骑手三端利益，任何一次宕机的影响都可能是灾难性的。如果体验不好，用户会说，为什么我付了钱，却还饿肚子？商家会说，这是因为出了餐没人取；但是对骑手来说，会觉得自己付出了时间与劳动，却没有获得足够的收益。
- 线下的业务复杂性更高。多条业务线管理模式不同，对于如何兼顾系统在共性和差异化上有很大挑战。



系统架构演进

美团配送系统架构的演进过程可以分为三个阶段：

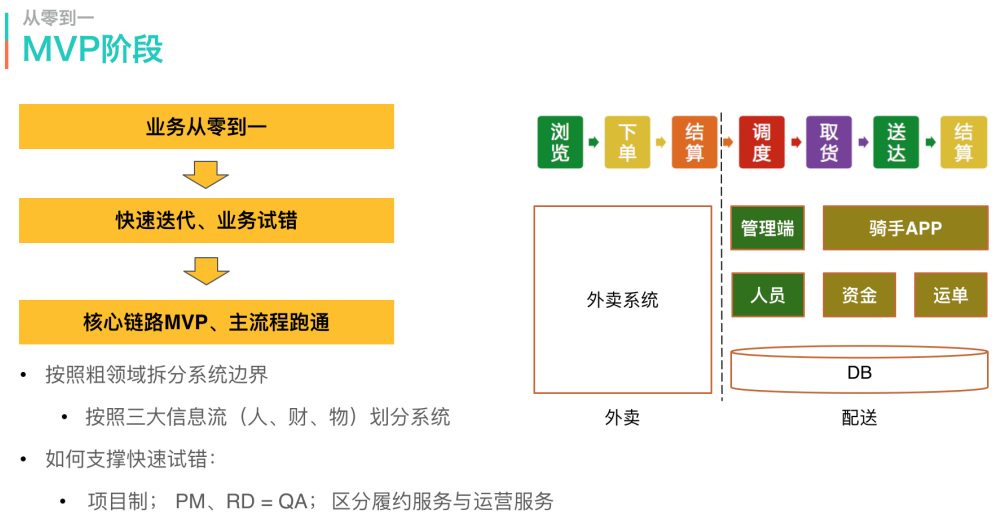
1. MVP阶段：业务模式探索，快速试错，如何具备快速迭代能力。
2. 规模化阶段：业务成指数级增长，如何既保证业务发展，又解决系统可用性、扩展性、研发效率等问题。
3. 精细化阶段：业务模式逐步成熟，运营逐步精细化，如何通过产品技术创新驱动业务发展。

MVP阶段

试错阶段，需要快速探索业务模式到底是不是一个方向，这个阶段不要期望很多事情都想得很清楚，用户和市场会快速反馈结果。所以，对于技术团队而言，这个阶段最主要的能力是快。抢夺市场，唯快不破。

从系统架构角度，MVP阶段只需要做粗粒拆解，我们按照**人、财、物**三大领域将系统做了初步服务划分，以保证后续的业务领域都可以从这三个主领域中分离、继承。

顺便提一下当时团队的组织形式，研发团队按项目制组织，大家共同维护一套系统。当时团队中无QA岗位，由PM、RD共同保证开发质量，一天发布二十几次是常态。



规模化阶段

进入这个阶段，业务和产品已经得到了市场的初步验证，的确找到了正确的方向。同时，业务发展增速也对研发团队的能力提出了更高的要求，因为这个阶段会有大量紧急且重要的事情涌现，且系统可用性、扩展性方面的问题会逐步凸显，如果处理不当，就会导致系统故障频发、研发效率低下等问题，使研发疲于奔命。

这个阶段从架构层面我们重点在思考三个方面的问题：

- 整体架构应该如何演化？履约系统与运营系统的边界在哪里？
- 履约系统的可用性如何保证？系统容量如何规划？
- 运营系统如何解决业务的真正痛点？如何在大量“琐碎需求”下提升研发效率？

解决以上问题的整体思路为**化繁为简**（理清逻辑关系）、**分而治之**（专业的人做专业的事）、**逐步演进**（考虑ROI）。

整体架构设计

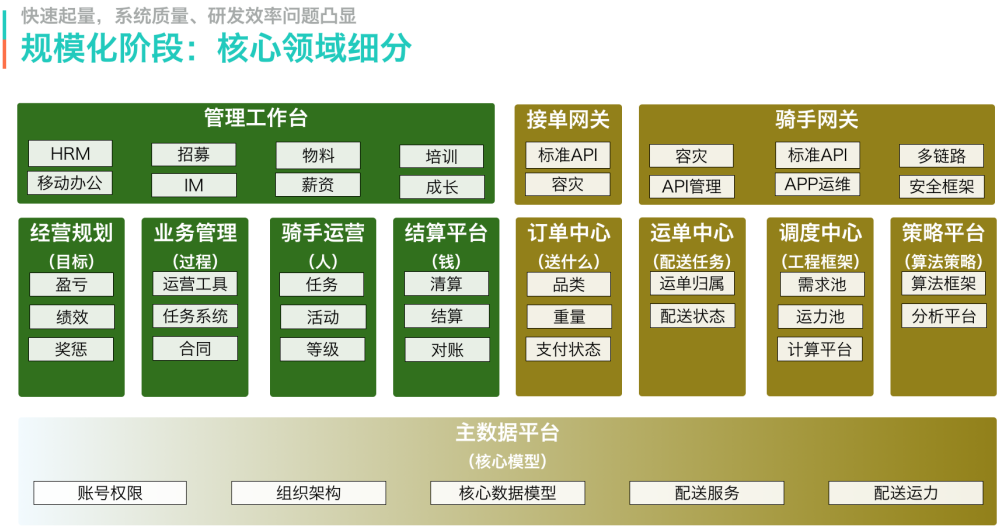
在整体架构上，我们将配送系统拆解为履约系统、运营系统和主数据平台。

履约系统（图右上侧）的设计上，首先按照用户侧与骑手侧做了初步划分，这样拆分兼顾了双端角色和调度流程的统一。例如：用户侧更关注发单的成功率与订单状态的一致性，骑手侧则更关注派单效果、推单成功率等，整体上解耦了发单、支付、调度等模块。

运营系统（图左上侧）方面，需求长期多而杂，架构设计上需要先想清楚配送的运营系统应该管什么、不应该管什么。在长期的项目开发中，我们从业务战略与组织架构出发，在明确业务战略目标和阶段策略下，梳理每个业务团队/岗位的核心职责、考核目标、组织之间的协作流程，最终整理出现阶段配送运营管理的中心为四个领域：

- 经营规划：如何科学地定义目标，并保证目标能够有效达成。
- 业务管理：如何提升每一个业务管理过程的效率与质量。
- 骑手运营：骑手是核心资源，一个城市需要多少骑手、骑手分级是否科学、如何调控需要系统性方案。
- 结算平台：提高钱的效能，是能否做到成本领先的关键。如何把钱用得对、用得准需要长期思考。

除了履约、运营两个系统的架构设计外，架构设计层面还有一个非常关键的问题，即履约、运营系统的边界与职责如何划分的问题。个人理解这个问题可能是O2O类业务在规模化阶段最关键的架构设计问题，如果不能有效解决将为系统的可用性、扩展性埋下巨大隐患。履约、运营两个方向的业务需求和技术职责有较大差异，且多数数据的生产都在运营系统，最核心最关键的应用在履约系统。虽然各自的领域职责是清晰的，但对于具体的需求边界上不见得简单明了。对此，我们借鉴了MDM思路，提出了**主数据平台**（图下侧）的概念，重点解决履约系统与运营系统的合作与边界问题。



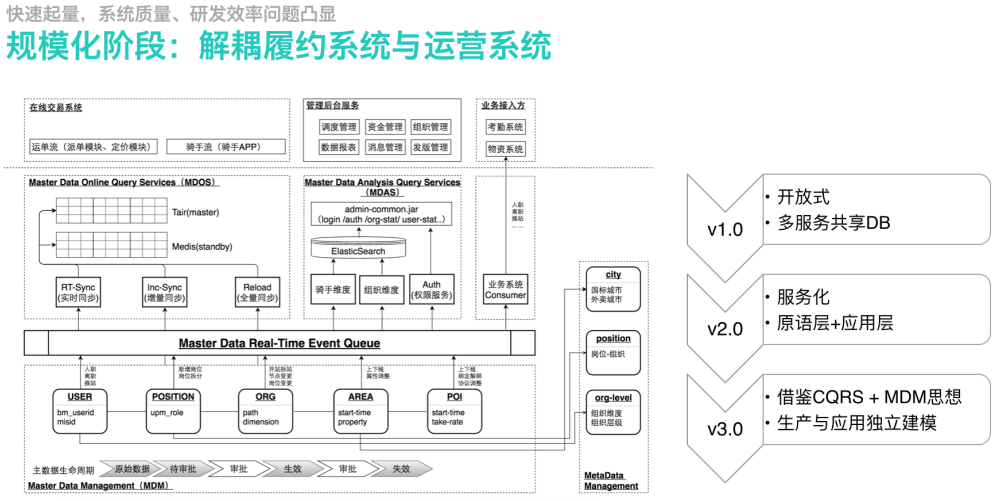
主数据平台

主数据是企业信息系统中最基础的业务单位数据，对于配送而言是组织、岗位、人员、商家、用户、城市等数据。与之对应的是业务数据，例如：订单、考勤、薪资等。主数据有两个最关键的特征：

1. 基础性：业务数据生长在主数据的维度上，例如：订单数据是用户、商家两个主数据实体下的交易数据
2. 共享性：各类系统都强依赖于主数据，主数据的变化上游各业务系统需要感知与联动

主数据管理并非一蹴而就，是伴随业务发展逐步迭代的。早期系统较简单，上游系统直接从DB中读取数据并应用。这种方案在系统逐步复杂之后，容易出现多个团队开发互相影响，不利于系统扩展，并且在可用性上有很大风险。为此我们专门成立的主数据的团队，独立拆分了主数据服务，并把所有对于数据的访问收回到服务上。在此基础上，经过不断的迭代和演进，最终我们吸收了CQRS（Command Query Responsibility Segregation）和MDM（Master Data Management）的思想，将整个主数据平台逐步划分成四个部分：

- 生产系统：负责对数据生产的建模，隔离数据生产对核心模型的影响。例如：骑手入职、组织拆分流程等。
- 核心模型：挖掘数据实体关系，提升模型能力。例如：一人多岗、双线汇报等。
- 运力中心：面向履约系统的应用场景支持，将骑手诸多属性抽象为运力模型，并对可用性、吞吐能力着重建设。
- 管理中心：面向运营系统提供标准化框架，提供信息检索、流程审批、权限控制等场景的统一解决方案。



美团配送主数据平台架构图

系统可用性

业务的快速增长对系统的可用性提出越来越高的要求，在方法论层面，我们按照事故发生的时间序列（事前、事中、事后）提出了四大能力建设，即：**预防能力**、**诊断能力**、**解决能力**、**规避能力**。同时，在具体工作上，我们划分为**流程**和**系统**两个方面。

可用性建设是一个长期项目。考虑到ROI，起步阶段重点完成事前的流程建设，即上线规范等一系列线上操作流程，这个工作在早期能够规避80%的线上故障。在流程规范跑通并证实有效之后，再逐步通过系统建设提升人效。

容灾能力

容灾能力建设上，首先思考的问题是系统最大的风险点是什么。从管理的角度来看，职责的“灰色地带”通常是系统质量容易出现风险的地方。因此，早期最先做的容灾处理是核心依赖、第三方依赖的降级，优先保证一旦依赖的服务、中间件出现问题，系统自身具备最基本的降级能力。

第二阶段我们提出了端到端的容灾能力。首先，我们建设了业务大盘，定义了实时监控核心业务指标（单量、在线骑手数等），通过这些指标能够快速判断**系统是不是出了问题**。其次，我们在核心指标上扩展了关键维度（城市、App版本、运营商等），以快速评估**问题有多大影响**。最后，我们通过Trace系统，将服务间的调用关系与链路级成功率可视化展现，具备了快速定位**问题的根因在哪**的能力。

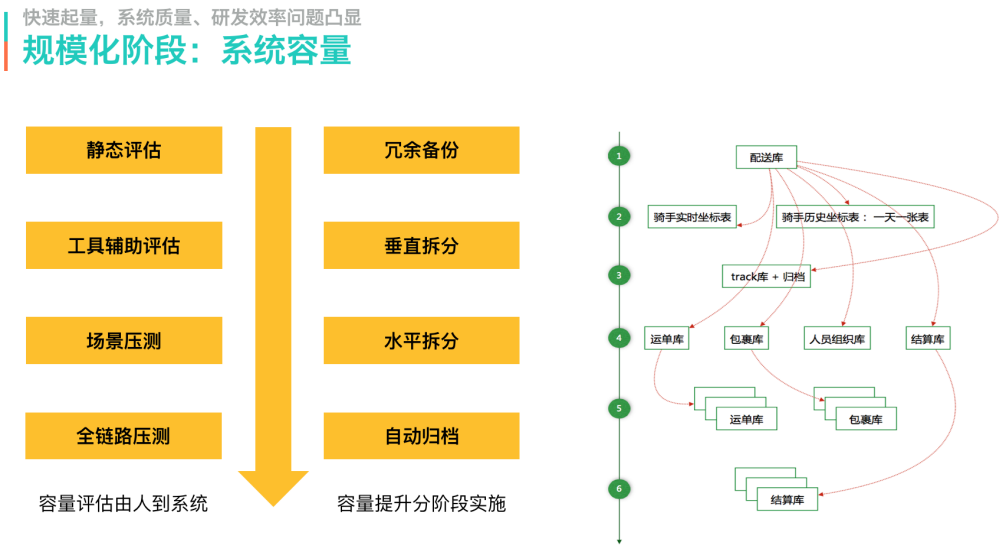
第三阶段，我们期望将容灾预案集成到系统中，基于各类事故场景打造定制化、一体化的容灾工具，这样可以进一步缩短故障的响应、处理时间以及研发学习成本。例如，为了进一步提升配送系统的SLA，我们在端到端的容灾能力上深度优化，重点解决了骑手弱网、无网的情况下的端到端交互问题。中国某些地区人群非常密集但移动运营商网络质量较差，会导致骑手到了这个区域后操作App延迟较大甚至无法操作，这对骑手的正常工作有非常大的影响。因此，我们在移动网络链路层面不断加强长连接、多路互备的能力，并将网络的诊断、处理、验证工具一体化，使骑手App的端到端到达率有了进一步的提升。

系统容量

对于一个规模快速增长的业务，系统的容量规划是一个长期命题。容量规划的关键点是评估与扩容。

评估方面，在业务发展早期我们一个架构师就能够完全掌控整个系统，采用静态评估的方式基本可以衡量系统容量。随着系统复杂度逐渐提升，我们逐步引入了Trace、中间件容量监控等工具辅助评估容量，由架构师团队定义容量评估主框架，由各团队细化评估每个子系统的容量。当业务已经变得非常复杂时，没有任何一个人或团队能够保证精确完成容量评估，这时我们启动了场景压测、引流压测、全链路压测等项目，通过 **流量标记** + **影子表** + **流量偏移** + **场景回放** 等手段，实现了通过线上流量按比例回放压测的能力，通过系统报告精确评估容量与瓶颈点。

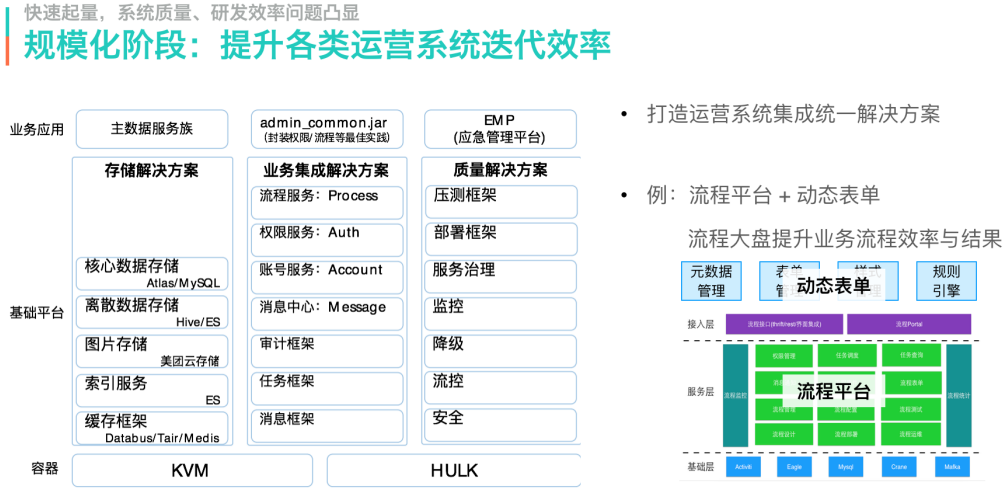
扩容方面，我们分阶段依次实施了**冗余备份**（主从分离）、**垂直拆分**（拆分核心属性与非核心属性）、**水平拆分**（分库分表）、**自动归档**。



运营系统迭代效率

运营系统涉及一个业务运营管理的方方面面，我们在业务领域上除了明确**目标**、**过程**、**运力**、**资金**四个领域外，打造了一套运营系统集成解决方案集合。研发通过持续投入精力在平台化服务或组件的长期建设上，使每个垂直的运营系统扩展性得到保证，从而不断提升研发效率。以工作流场景为例，通过**动态表单**

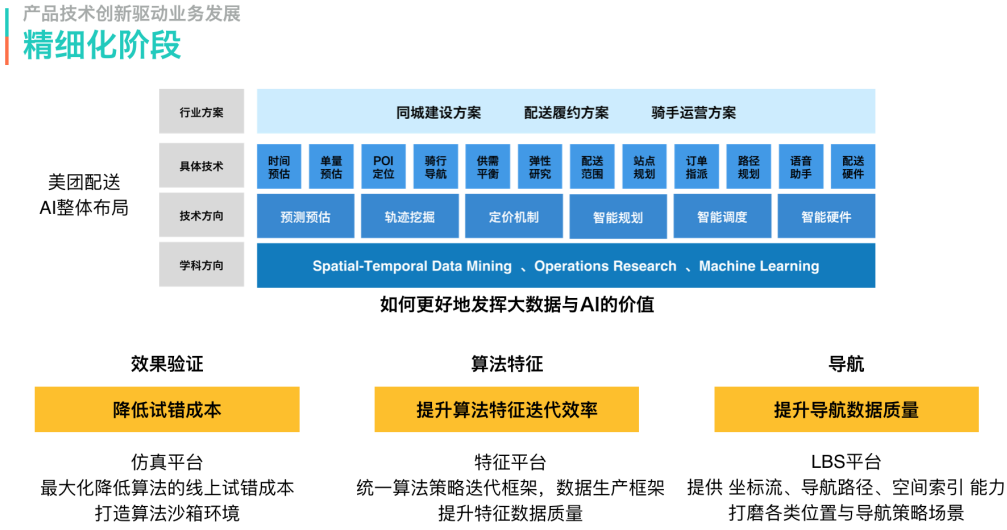
+ **流程平台**的方式，统一各类业务流、审批流的工程实现，各类管理动作的效率与质量可量化，找到流程阻塞节点，自动化部分流程环节，通过技术手段不断降低人工成本。



精细化阶段

业务发展不断成熟之后，业务的各类运营管理动作会趋于精细化。这个阶段，业务对于产品技术有更高要求，期望通过产品技术创新不断打造技术壁垒，保持领先优势。配送的业务特点天然对AI应用有很强的需求，大到供给调整，小到资源配置，都是AI发挥效力的主战场。对于工程层面，需要持续思考的问题是如何更好地实现AI的业务应用。为此我们重点提升了几方面的能力：

- 降低试错成本：构建仿真平台，打造算法的“沙箱环境”，在线下环境快速评估算法效果。
- 提升算法特征迭代效率：构建特征平台，统一算法策略迭代框架与特征数据生产框架，提升特征数据质量。
- 提升导航数据质量：持续深耕LBS平台，提升基础数据质量，提供位置、导航、空间的应用能力。



仿真平台

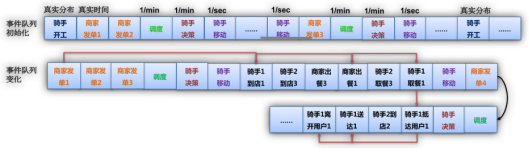
仿真平台的核心是打造“沙箱环境”，配送的服务业属性要求用户、商家、骑手深度参与服务过程，因此算法的线上试错成本极高。对于仿真平台的建设上，我们删减掉调度系统的细枝末节，粗粒度的构建了一

套微型调度系统，并通过**发单回放**、**用户**、**商家**、**骑手实体建模**、**骑手行为模拟**等方法模拟线上场景。每次仿真会产出算法的KPI报告，实现算法效果的离线预估。

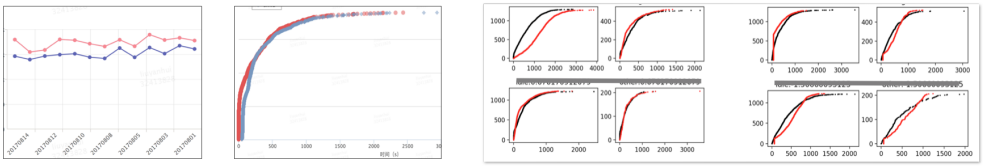
产品技术创新驱动业务发展

精细化阶段：仿真平台

目标：构建线下模拟沙盘



思路：基于线上真实数据对配送全流程构建模拟场景，并对事件、数据模拟



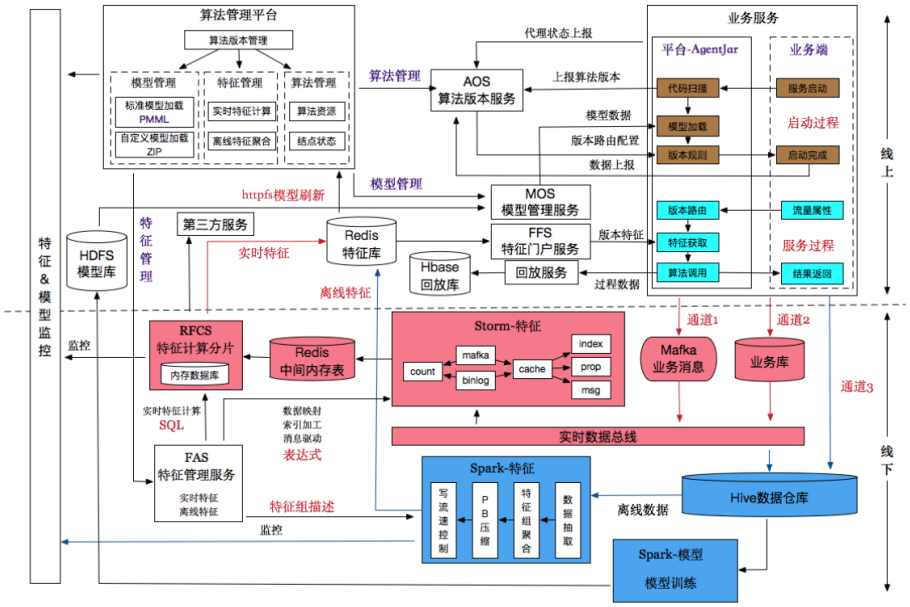
仿真报告可视化

算法数据平台

算法策略的效果，主要依赖于算法模型和特征数据的质量。为此我们围绕模型和特征，打造了一站式算法数据平台，提供从数据清洗、特征提取，模型训练、线上预测到算法效果评估的全方位数据闭环解决方案，为机器学习和深度学习算法模型在配送各个业务线落地提供支撑。

产品技术创新驱动业务发展

精细化阶段：算法数据平台



LBS平台

LBS平台早在配送业务的起步阶段就开始实施，随着算法场景的不断发展，LBS不断深化点线面空间能力，为配送调度、时间预估、定价等业务场景提供支撑，打造了任务地图、路径规划、语音导航、热力图

等产品。

结语

美团配送系统架构的演进过程，架构师团队长期关注技术驱动业务、明确领域职责与边界等关键问题，同时架构的演进过程也是不断考虑ROI的权衡取舍过程。技术的持续发展不断提升体验、规模，降低运营成本，而架构在里面解决的问题是化繁为简，将复杂问题拆解为简单的问题并通过领域专家逐级各个击破。随着规模的持续增长，业务的持续创新会给系统架构提出越来越高的挑战，系统架构设计将是我们长期研究的一个课题。

作者简介

- 永俊，美团资深技术专家，配送业务系统团队负责人。长期从事配送系统质量保证、运营体系建设、系统架构升级等方向。

招聘

本文为美团配送技术团队的集体智慧结晶，感谢团队每一名成员的努力付出。如果你对业务分析、领域模型感兴趣，欢迎联系yinyongjun@meituan.com。

美团旅行销售绩效系统研发实践

作者: 夷山 龙龙 张乐

背景

O2O是目前互联网竞争最激烈的领域之一，其重要的业务特征是有大规模的线下业务团队，他们分布在五湖四海，直接服务着数以百万的商家，责任很重，管理的难度巨大。能否通过技术手段，打造高效的线下团队，是O2O公司的核心竞争力。随着美团酒旅业务的快速增长，业务人员绩效的考核方案和激励政策呈现出多样化、复杂化的特点，对绩效计算实时性和准确性的要求也越来越高。

原始的绩效计算流程，包括各战区目标下发、激励调整、数据校验等繁琐工作，需要投入大量的人力，准确性也无法得到保障。鉴于这种情况，以优化体验、提升效率、降低成本为目标，我们设计并实现了一套在准确性、实时性、灵活性等方面表现都比较优秀的绩效系统来解决上述问题。

挑战

在开发Hermes（新的绩效系统的名称，是希腊神话中的财神）的过程中，主要面临的挑战有如下几个：

1. 数据量大，来源广且高度复杂。业务人员绩效相关的数据每天以百万级的速度在增长，月末产生的数据至少是千万级别的，而且所产生的数据由于没有统一的规范，所以复杂度特别高。由于数据没有统一的来源，而且将来可能还会增加新的数据源，那么就需要我们对各方的数据进行预先的组织和处理。为了实现绩效计算的准确性和实时性，我们至少要保证每天都能更新绩效数据，否则月底的压力非常大，一是研发人员面对如此海量的数据计算压力十分大，二是业务人员数据校验的压力也会非常大。
2. 绩效考核方案配置灵活，指标规则千变万化。为了更好地支撑业务的增长，每个月的考核方案和考核规则变化非常大，那么我们必须保证方案的配置灵活，指标规则的配置通用，能够跟得上业务发展的脚步。
3. 绩效目标设置困难，难以有效、快速地提高业务人员的效率和积极性。绩效目标的设置往往是根据以往的经验决定的，但是随着酒旅业务的高速增长，经验可能往往不够，设置高了影响业务人员的积极性，设置低了，影响业务人员的效率，更是不利于公司业务长久高速地增长。所以最终我们要实现智能化、差异化的生成绩效考核方案。

解决思路

为了解决这一系列的问题，我们设计了如图1所示的整体流程。

- 数据引擎完成了数据的清洗、加载、转换、聚合的过程，为系统提供了稳定的外部数据支持
- 激励管理模块完成指标数据的申诉、主观考核以及激励调整等内部的数据准备工作，进一步保障了数据的可靠性和完整性
- 指标配置模块供商业分析的同事创建考核方案时使用，考核粒度可细化到具体的业务人员，该模块实现了考核指标的动态创建以及指标的多样性考核等功能<
- 规则引擎完成了指标考核规则的配置、解析和匹配功能，使指标数据的计算更加准确、高效，同时，计算过程可追溯、可解释
- 计算引擎包括指标依赖解析、SQL执行器、计算调度器三个子部分，实现了根据指标的依赖关系完成指标的下钻、回溯计算流程，保障了系统的正常调度、准确计算及稳定输出功能

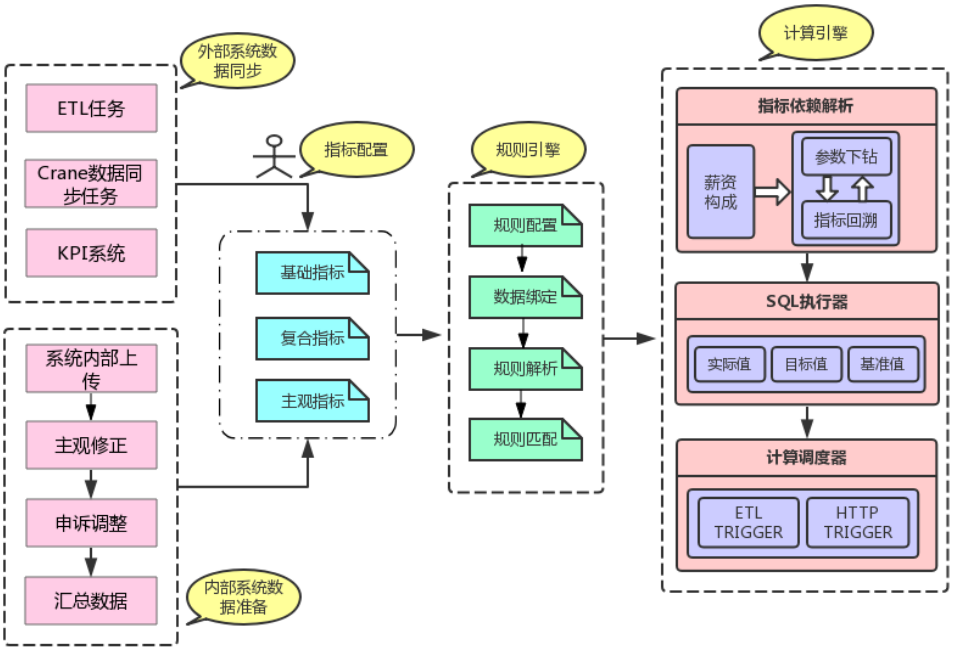


图1 系统整体流程图

解决方案

在明确解决思路之后，我们就开始了Hermes系统的设计，具体的系统架构图如图2所示。我们在明确整体功能和架构之后，设计了六个模块和四个引擎，双重缓存结构，下面我们开始逐层的介绍。

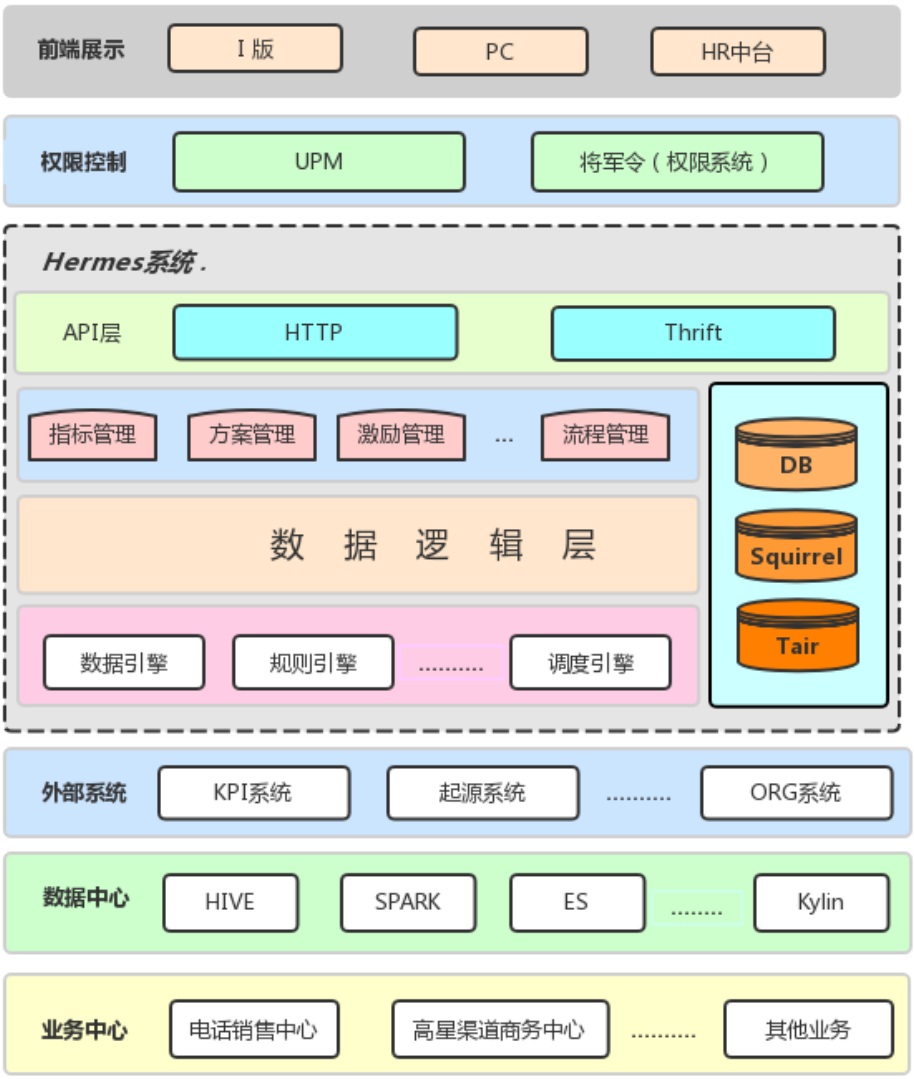


图2 Hermes系统架构图

在API层我们会通过HTPP和Thrift两种方式分别为前端和HR中台提供数据，为了保证数据的安全性，在登录和流程管理时我们会使用UPM（美团点评用户权限管理系统）控制当前用户的权限以及展示的模块，在具体到指标管理、方案管理和激励管理等模块时，我们会依赖于将军令系统（美团点评数据管理权限系统）做数据权限控制。

指标管理主要用于管理指标的信息，指标是组成方案的最基本的元素，分为基础指标、主观指标和复合指标。基础指标是从外部系统获取，而复合指标和主观指标是在Hermes系统内创建。

如图3所示，复合指标是由基础指标和数字组成，并且在创建之初就需要确定该复合指标的数据口径以及数据来源。数据口径主要分为本人的、下属的和直接下属的三种，主要是为了标明所考核的指标是使用本人的值还是下属的汇总值。我们选择将数据口径放在指标维度是基于两点考虑，一是结构清晰，二是可以针对业务经理所考核的指标做预计算，减少后期的复杂度。数据来源是指所考核的指标的目标值、基准值等等数据的来源。如果是本系统上传，那么可以在创建方案的时候强制用户上传，起到一个预判断的作用。

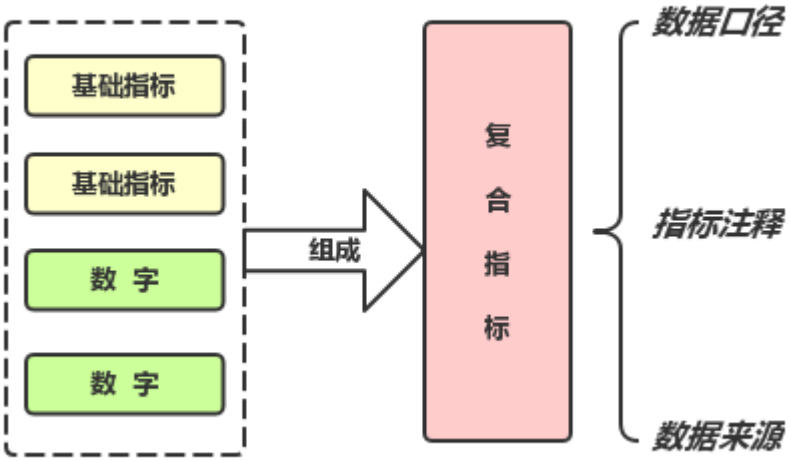


图3 复合指标

主观指标主要应用于主观考核，即领导对于员工个人表现的一个打分，一般会作为最后工资计算的系数，在创建主观指标的时候也会有一个预判断，防止主观考核的系数过高或者过低。

方案管理主要用于管理方案的相关信息，如图4所示，方案主要是由考核对象、考核指标、指标规则和方案的封顶值等元素组成。方案的创建和修改在此模块发起，但是并不是真正的创建和修改方案，只是将修改或创建方案的任务插入到任务队列中去，扮演着生产者的角色，然后由调度引擎即消费者来真正完成方案的创建和更改操作。这里我们使用了Producer–Consumer模式，并在此基础上进行了改良，使用工作窃取算法完成方案的快速创建与修改。

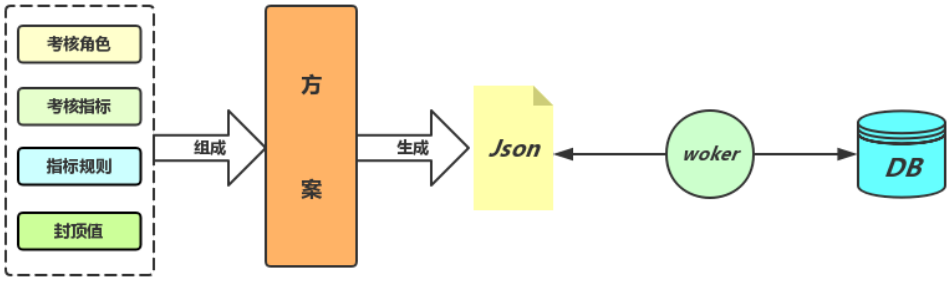


图4 方案创建

由于Hermes系统负责美团旅行所有战区的所有业务人员绩效方案的创建与修改，任务量有时会非常的大。如果使用简单的Producer–Consumer模式，那么会使众多的消费者共享一个队列实例，很容易产生锁的竞争，即修改队列的头指针时所需要获取的锁而导致的竞争。如果一个通道实例对应多个队列实例，那么就可以实现多个消费者线程同时从通道中获取任务的时候访问的是各自的队列实例。

此时，各个消费者线程修改队列的头指针并不会导致锁的竞争。同时，当一个通道对应于多个队列实例时，当一个消费者处理完该线程所对应的队列中的任务后，它可以继续从其他的队列中取出任务进行处

理，这样既不会导致该消费者线程闲置，又减轻了其他消费者的负担，加快了任务的执行进度。当任务执行失败之后，我们会使用公司的组件Mafka（Mafka是美团点评内部的消息中间件，在Kafka的基础上进行了优化）将任务做一个标识重新放入队列中去，当重试多次后仍然失败，会使用大象（是美团点评内部的一个通信软件）通知方案创建者和相关的研发人员。

激励管理模块主要是用于对业务人员的激励或是惩罚以及业务人员的申诉。主要通过主观值和调整值对业务人员进行激励和惩罚，调整值是直接在方案计算业务人员工资的基础上加或减工资，而主观值主要用于上级对下属的主观评价，然后作为一个系数直接作用于绩效考核方案的结果。如果业务人员对于某个指标的相关数据产生疑问，可以在此模块进行申诉。

如图5所示，申诉、激励调整和主观考核都会发起一个审批流程。由于审批流比较复杂，我们使用的是公司内部的组件Gravity（美团点评内部的流程管理组件）。它是基于activiti引擎构建的流程平台，能够轻松管理业务流程事项，并且提供了统一的流程规划、流程建模、流程部署、流程管理、任务管理、流程监控、流程分析、流程运维等服务，这样我们就节约了一定的开发成本。

审批如果被驳回，我们会通过大象通知流程发起者；审批通过时，我们一方面会通知流程发起者，另一方面我们会把审批的详细信息存入到一张中间表中。然后通过定时调度引擎来定时扫描该表中的数据，目前是10分钟一扫描，然后依据审批信息重新计算相关人员的绩效数据，最终更新数据库中的信息，完成申诉、激励调整和主观考核的流程。

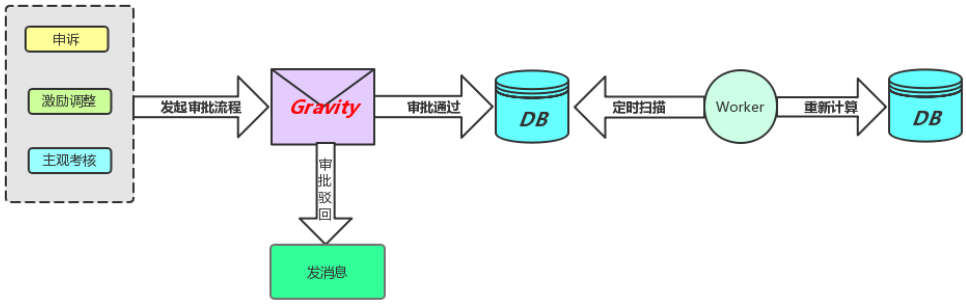


图5 激励管理

激励查询主要分为业务人员查询和管理人员查询两个部分。业务人员可以查询到自己的工资详情，还有每个指标的完成情况以及每个指标对应的奖励详情；管理人员可以查询自己的工资详情，还有其所属下属指标的完成情况，以及下属之间竞争力的比较。

当主管查询下属的各个指标具体完成情况或是历史详情时，数据量会非常大。如图6所示，我们会先count一下，拿到具体的数据量，当数据量超过5万的时候前端渲染就会有问题，这时我们会提示用户数据量过大，只能将数据下载到Excel中。如果数据量没有超过5万，我们会走两级缓存结构。当一级缓存中有数据时直接返回，整个请求结束；如果一级缓存异常，我们会走二级缓存，二级缓存有数据直接返回，整个请求结束；如果二级缓存也发生异常（几乎不可能），我们会查询数据库。

为了防止整个查询超时，我们会在用户访问时返回给前端一个query_id作为唯一的查询标识符。前端会每隔5分钟轮询调用后台接口，直到有数据返回，整个查询结束。这样就很好地解决了当数据量过大，无法

正常返回的场景。

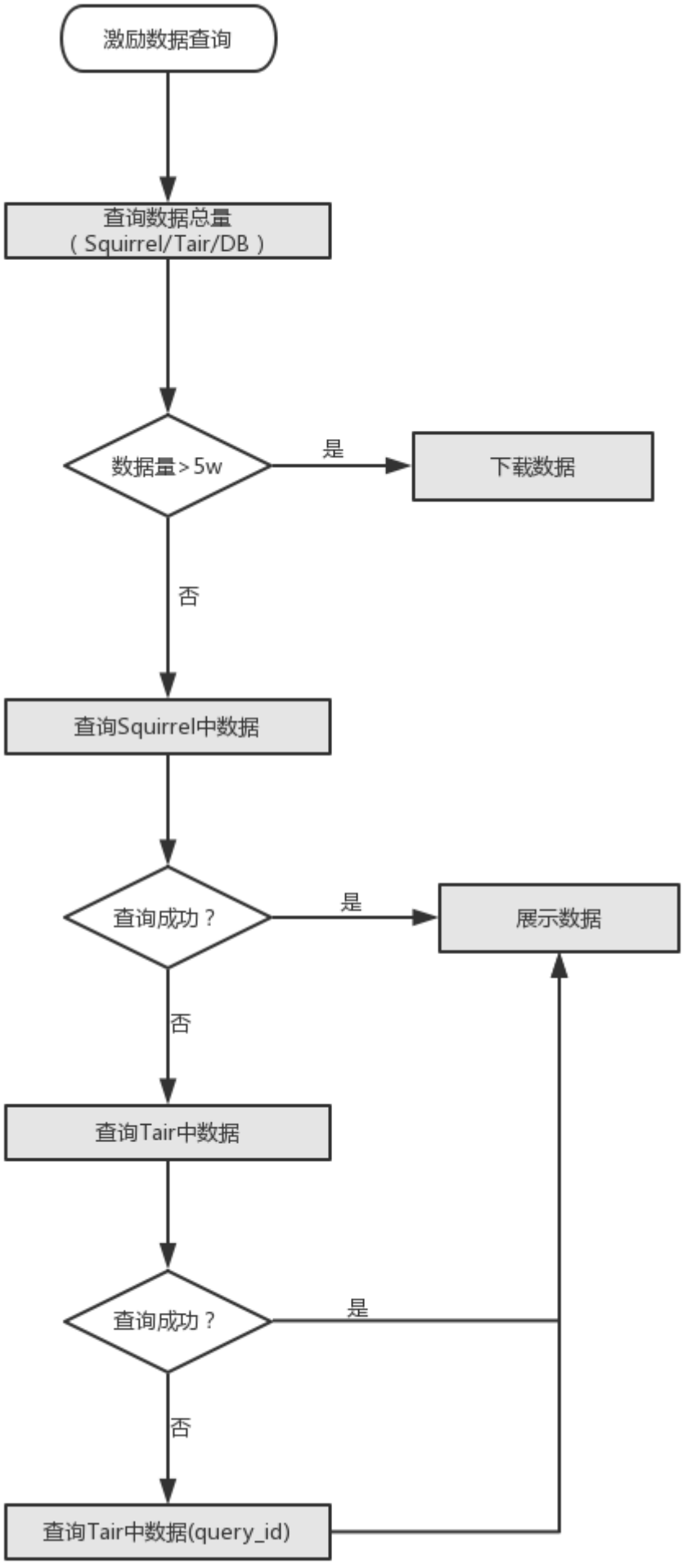

激励数据查询
查询数据总量
（Squirrel/Tair/DB）
数据量>5w
是
下载数据
否
查询Squirrel中数据
查询成功？
是
展示数据
否
查询Tair中数据
查询成功？
是
否
查询Tair中数据(query_id)

图6 激励信息查询流程图

流程管理，主要是对于审批流程的管理。审批流使用的是公司内部的组件Gravity，我们在此基础上进行了一层封装，保存了审批相关的所有信息，这样我们就可以依据自身的场景做一些业务逻辑。现在只有流程查询和审批内容查询两部分，将来还会加入审计和权限管理等功能。

绩效管理，主要是用于和总部的HR中台打通。他们可以在这里查询员工的各种绩效信息，然后对比分析以核查方案创建的有效性，即考核方案是否合理，是否有效地鼓舞了员工，提升效率等等。

数据引擎是最基础的部分，是各种运算的数据来源和对外交互的数据来源。如图7所示，数据来源十分杂乱，有的来自于ETL任务，有的来自于起源系统等等，而且它们各有各的数据结构。如果我们为每一个来源都提供一种接入方式，那么将来如果有新的数据源，我们也需要新增一种新的接入方式，这样是不满足系统的可扩展性的。所以我们要做的是提供一种统一的接入方式，一个统一的数据规范，为此我们设计了一个适配层。

无论是哪里来的数据必须符合适配层的规范，否则不予录入。这样就很好的解决了数据来源广、不规范的问题，无论将来新增多少种数据源都不需要我们做新的开发工作，节省了大量的无用功。在适配层我们会做一些简单的数据组装工作，然后分别存入到对应的表中。而且在适配层我们还会提前做一些缓存的逻辑，为后面的数据计算做好铺垫。

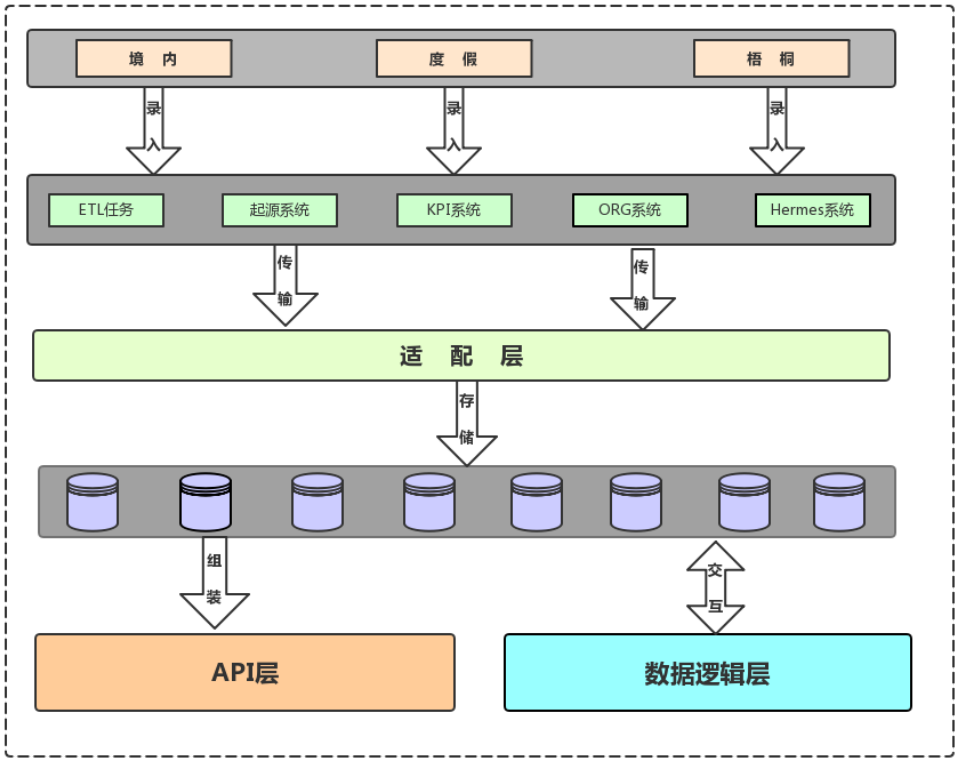


图7 数据引擎

API层是我们对外提供数据的核心层，我们会为总部和HR中台提供一些绩效数据。而数据逻辑层则负责Hermes系统内部的逻辑计算，会将计算结果回写到数据库和缓存中。

规则引擎是整个绩效系统的核心部分，主要包含两个方面，一是建立合理有效的激励考核方案，二是针对考核方案所涉及的指标进行规则的配置。方案的配置方式特别灵活，我们需要做的是能够兼容所有战区之间的差异。必须做到通用性强，能够满足战区的大部分需求，而不是每个月为了新的需求忙得焦头烂额。只有这样才能有效地解放人力，将一些重复的工作屏蔽掉，实现智能化方案管理。

如图8所示，方案是由参数组成，而参数是由指标组成，然后针对每一个指标配置一套规则。规则是由计算规则和计算表达式两部分组成。参数是对业务人员某项业务能力的考量。当配置方案时，我们只需根据当前月的主要目的就可以进行参数的筛选，而不必每个月都从头开始创建新的方案。

指标是组成方案的最基本的元素，每个指标会对应一套规则。针对不同的场景可以选择不同的规则，这样完全能够满足大部分的业务需求，每个月只需新增不满足需求的指标或指标规则即可，节省了大量的配置工作。计算规则是一个通用的逻辑表达式，计算表达式和计算规则成对出现，是一对一的关系，类似于数学的计算公式。

配置方案的时候需要指明该方案所要考核的对象，即图8中的Role。考核对象可能是一个组织，可能是一个个人，也可能是一个组织和多个人等等。同时还需要对这些角色进行职位的筛选，并且还要考虑一个人的离职、转岗以及升职等等情况。正是基于这种场景，我们选择使用empId_positionId_orgId作为一个唯一键，来唯一确定一个人。

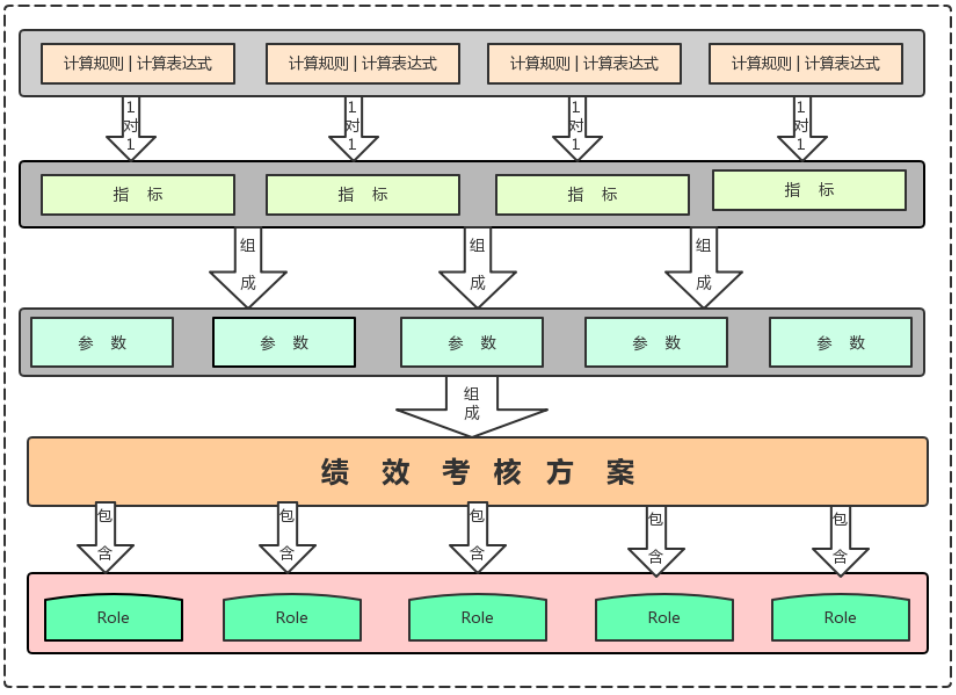


图8 规则引擎

这里还有一个关键问题，方案的流转状态包括审批中、通过、驳回和拒绝，而审批状态的变化必然涉及到方案的有效性。方案的有效性会引起Role的变化，因为一个自然月内一个人（empId_positionId_orgId）

有且仅能有一个方案。那么我们怎么保持数据库、Squirrel缓存和Tair缓存的一致性呢？我们采用的依旧是任务队列的方式，即方案的状态发生变动时，我们会把它作为一个任务放到任务表中，然后使用一个任务以事务的方式来保证缓存和数据库的一致性。

调度引擎主要就是定时的完成一些任务，主要包含两个方面，一是数据的预加载，二是方案相关操作的异步化。

数据的预加载主要涉及到战区人员信息的提前缓存，人员组织信息的提前缓存和基础数据预计算结果的缓存等等，所有涉及到数据量大并且频繁使用的数据，我们都会进行预处理和存储在两个缓存当中，这样能够有效地提高系统的响应速度和降低出错率。人员信息和人员的组织信息主要是调用公司的ORG接口，如果选择实时调用接口，那么性能将会有一定的影响，而且这样我们将严重依赖于第三方，一旦ORG出错我们也会受到波及，所以我们选择使用两级缓存结构来降低这种风险。第一，我们是提前缓存数据，一旦出错我们能在业务人员使用前解决掉（数据同步时间是每天凌晨）；第二，我们使用了两级缓存结构，二者同时出错的概率极小，而且响应性能肯定是比接口好的，还有可以预先根据业务场景组织数据。其他基础数据的预加载和这个是一个思路。

方案相关操作的异步化是指创建方案和编辑方案时，由于涉及的表操作和缓存操作较多，如果同步构建缓存和同步更新数据库很有可能造成超时，即使不超时，响应时间也是非常长的，影响用户体验。所以当创建或是修改方案时只是将前端的入参合理组装之后便插入到任务表中，然后通过调度引擎扫描任务表更新方案信息和构建缓存，同时更新任务表状态。

计算引擎的设计我们借鉴了Hadoop中的计算模型（Map-reduce）的设计理念。如图9所示，在Map阶段，我们对薪资表达式先进行参数级别的下钻。图中的参数1指的是业务人员的指标奖金考核项，可依赖于指标类数据项（包括基础指标、复合指标、主观指标及在职奖金基数类统计指标）。此处实现了参数粒度的并行计算，接下来再进行指标级别的下钻。若该处的指标为复合指标时，计算引擎会继续下钻到基础指标级别；待所有的指标计算结果就绪后，会触发计算引擎的回溯计算流程(类似Hadoop中的reduce过程)，依赖于各指标的参数因子会被计算并保存。当各参数因子准备就绪时，直接生成绩效最终结果。

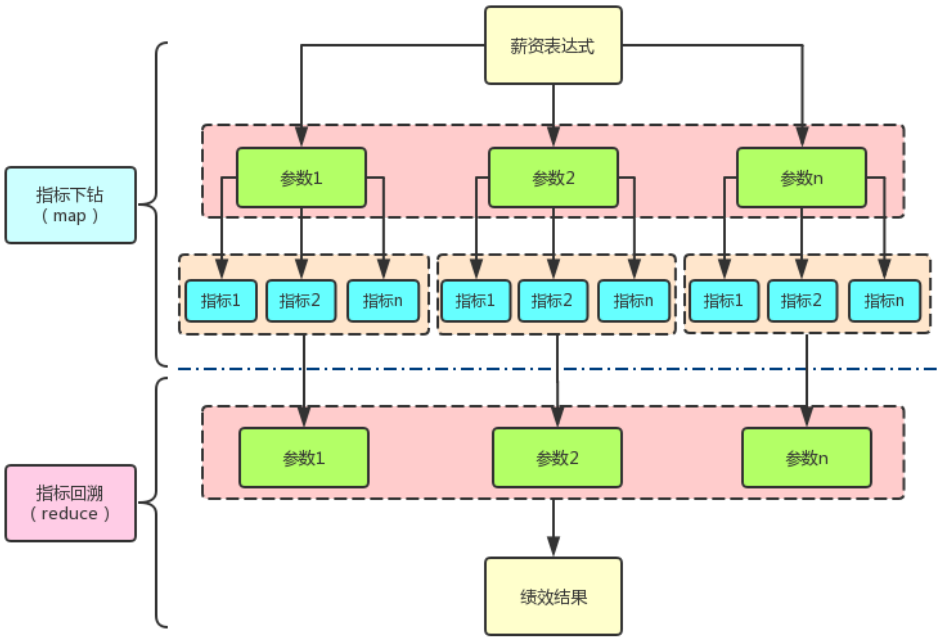


图9 计算引擎

展望

我们最终的目标是将Hermes系统工具化、平台化，增强系统数据的可视化能力和可分析能力，指导战区制定更加合理化、人性化的激励性更强的考核方案；对于业务人员来说，提供指导性和针对性更强的指标业务导向，提升工作效率，进一步增强美团酒旅业务在行业中的竞争能力。后续，我们计划使用数据挖掘和机器学习等手段提升系统的智能化水平，以数据为依托，业务人员业绩产出为目标，指导修正已有的绩效考核方案。

针对现有系统的不足，我们也将努力修正，例如数据源管理方面，针对指标来源分散导致的管理难的问题，我们将引入起源，统一管理指标数据，降低系统复杂度。总而言之，我们的目标就是要做一个稳定的，高效的绩效系统，欢迎加入我们！

作者简介

- 夷山，美团点评技术专家，现任TechClub–Java俱乐部主席，2006年毕业于武汉大学，先后就职于IBM、用友、风行以及阿里。2014年加入美团，长期致力于BI工具、数据安全与数据质量工作等方向。
- 龙龙，美团平台数据中心后台开发工程师，2016年毕业于哈尔滨工业大学，2016年加入美团点评数据中心，长期从事BI工具开发工作。
- 张乐，美团平台数据中心后台开发工程师，2015年毕业于吉林大学，2017年加入美团点评数据中心，长期从事BI工具开发工作。

招聘信息

最后插播一个招聘广告，有对数据产品工具开发感兴趣的可以发邮件给 fuyishan#meituan.com。我们是一群擅长大数据领域数据工具，数据治理，智能数据应用架构设计及产品研发的工程师。

美团酒旅实时数据规则引擎应用实践

作者: 晓星 伟彬

背景

美团点评酒旅运营需求在离线场景下，已经得到了较为系统化的支持，通过对离线数据收集、挖掘，可对目标用户进行T+1触达，通过向目标用户发送Push等多种方式，在一定程度上提高转化率。但T+1本身的延迟性会导致用户在产生特定行为时不能被实时触达，无法充分发挥数据的价值，取得更优的运营效果。

在此背景下，运营业务需要着手挖掘用户行为实时数据，如实时浏览、下单、退款、搜索等，对满足运营需求用户进行实时触达，最大化运营活动效果。

业务场景

在运营实时触达需求中，存在如下具有代表性的业务场景：

> 1. 用户在30分钟内发生A行为次数大于等于3次
> 2. 用户为美团酒店老客，即用户曾购买过美团酒店产品
> 3. 用户在A行为前24小时内未发生B行为
> 4. 用户在A行为后30分钟内未发生B行为（排除30分钟内用户自发产生B行为的影响，降低对结果造成的偏差）

本文以该典型实时运营场景为例，围绕如何设计出可支撑业务需求高效、稳定运行的系统进行展开。

早期方案

运营实时触达需求早期活动数量较少，我们通过为每个需求开发一套Storm拓扑相关代码、将运营活动规则硬编码这一“短平快”的方式，对运营实时触达需求进行快速支持，如图1所示：

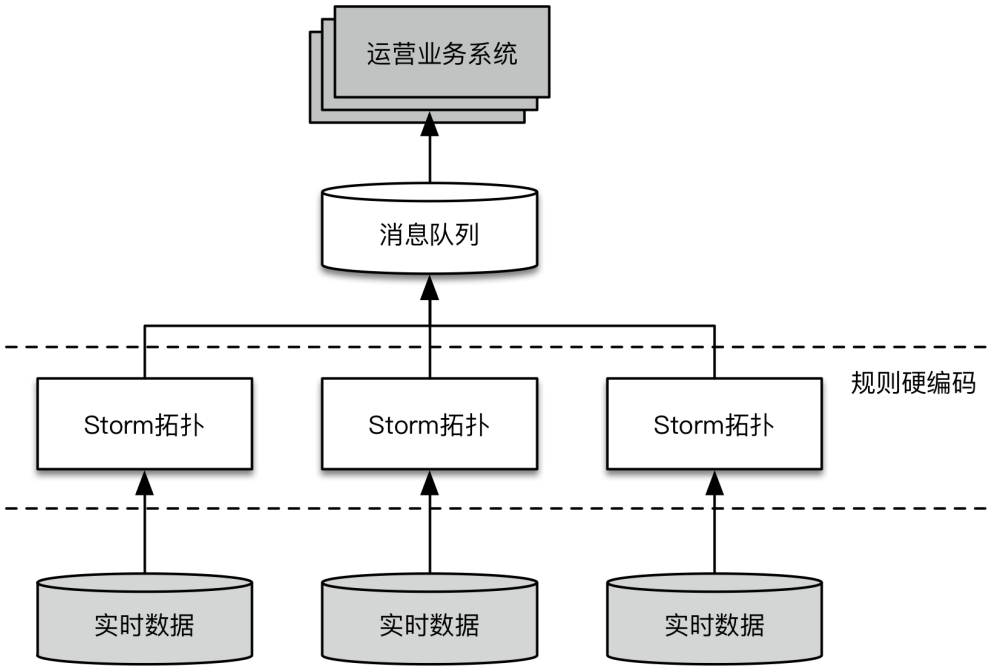


图1 早期方案示意图

早期方案的问题

早期方案是一种Case By Case的解决方式，不能形成一个完整的系统。随着实时运营业务开展，相关运营活动数量激增，线上维护着多套相似代码，一方面破坏了 DRY（Don't Repeat Yourself）原则，另一方面线上维护成本也呈线性增长，系统逐渐无法支撑当前的需求。

挑战

为解决早期方案中出现的问题，对系统建设提出了以下挑战：

- 硬编码活动规则的方式产生了大量重复代码，开发成本较高，需求响应时间较长。
- 业务规则修改困难，调整运营活动条件需要修改代码并重启线上拓扑。
- 线上Storm拓扑较多，资源利用率、系统吞吐量低，统一维护成本较高。
- 缺乏完善的监控报警机制，很难早于业务发现系统及数据中存在的稳定性问题。

针对以上挑战，结合业务规则特点，美团点评数据智能团队调研并设计了酒旅运营实时触达系统。

技术调研

规则引擎的必要性

提高灵活度需要从业务规则和系统代码解耦和入手，规则和代码耦合直接导致了重复代码增多、业务规则修改困难等问题。那如何将业务规则和系统代码解耦和呢？我们想到使用规则引擎解决这一问题。

规则引擎是处理复杂规则集合的引擎。通过输入一些基础事件，以推演或者归纳等方式，得到最终的执行结果。规则引擎的核心作用在于将复杂、易变的规则从系统中抽离出来，由灵活可变的规则来描述业务需求。由于很多业务场景，包括酒旅运营实时触达场景，规则处理的输入或触发条件是事件，且事件间有依赖或时序的关系，所以规则引擎经常和 CEP（复合事件处理）结合起来使用。

CEP通过对多个简单事件进行组合分析、处理，利用事件的相互关系，找出有意义的事件，从而得出结论。文章最前面背景中提到的业务场景，通过多次规则处理，将单一事件组合成具有业务含义的复合事件，进而提高该类仅浏览未下单的用户的下单概率。可以看出，规则引擎及CEP可以满足业务场景的具体需求，将其引入可以提高系统面对需求变化的灵活度。

规则引擎调研

在设计规则引擎前，我们对业界已有的规则引擎，主要包括 Esper 和 Drools，进行了调研。

Esper

Esper设计目标为CEP的轻量级解决方案，可以方便的嵌入服务中，提供CEP功能。

优势

- 轻量级可嵌入开发，常用的CEP功能简单好用。
- EPL语法与SQL类似，学习成本较低。

劣势

- 单机全内存方案，需要整合其他分布式和存储。
- 以内存实现时间窗功能，无法支持较长跨度的时间窗。
- 无法有效支持定时触达（如用户在浏览发生后30分钟触达支付条件判断）。

Drools

Drools开始于规则引擎，后引入Drools Fusion模块提供CEP的功能。

优势

- 功能较为完善，具有如系统监控、操作平台等功能。

劣势

- 学习曲线陡峭，其引入的DRL语言较复杂，独立的系统很难进行二次开发。
- 以内存实现时间窗功能，无法支持较长跨度的时间窗。
- 无法有效支持定时触达（如用户在浏览发生后30分钟触达支付条件判断）。

由于业务规则对时间窗功能及定时触达功能有较强的依赖，综合以上两种规则引擎的优劣势，我们选用了相对SpEL更为轻量的表达式引擎 Aviator，将流式数据处理及规则引擎集成至Storm中，由Storm保证系统在数据处理时的吞吐量，在系统处理资源出现瓶颈时，可在公司托管平台上调整Worker及Executor数量，降低系统水平扩展所需成本。

技术方案

确定引入规则引擎后，围绕规则引擎的设计开发成为了系统建设的主要着力点。通过使用实时数据仓库中的用户实时行为数据，按业务运营活动规则，组合成有意义的复合事件，交由下游运营业务系统对事件的

主体，也就是用户进行触达。将系统抽象为以下功能模块，如图2所示：

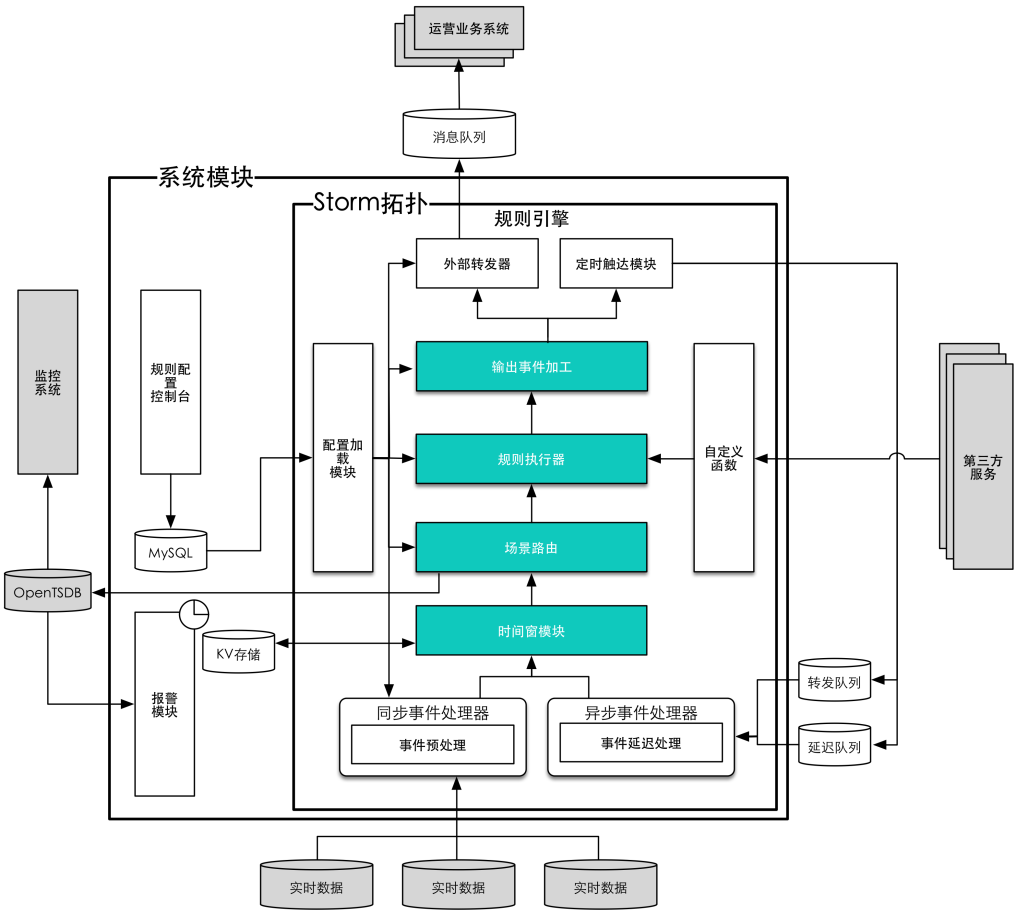


图2 系统模块图

总体来看，系统组成模块及功能如下：

- 规则引擎：集成于Storm拓扑中，执行运营活动条件转换成为的具体规则，作出对应响应。
- 时间窗模块：具有可选时间跨度的滑动时间窗功能，为规则判定提供时间窗因子。
- 定时触达模块：设定规则判定的执行时间，达到设定时间后，执行后续规则。
- 自定义函数：在Aviator表达式引擎基础函数之上，扩展规则引擎功能。
- 报警模块：定时检查系统处理的消息量，出现异常时为负责人发送报警信息。
- 规则配置控制台：提供配置页面，通过控制台新增场景及规则配置。
- 配置加载模块：定时加载活动规则等配置信息，供规则引擎使用。

其中，规则引擎由核心组件构成的最小功能集及扩展组件提供的扩展功能组成。由于规则引擎解耦了业务规则和系统代码，使得实时数据在处理时变的抽象，对数据监控、报警提出了更高的要求。下面我们将从规则引擎核心组件、规则引擎扩展组件、监控与报警三个方面分别进行介绍。

规则引擎核心组件

规则引擎核心组件为构成规则引擎的最小集合，用以支持完成基础规则判断。

规则引擎核心流程

引入规则引擎后，业务需求被转换为具体场景和规则进行执行，如图3所示：

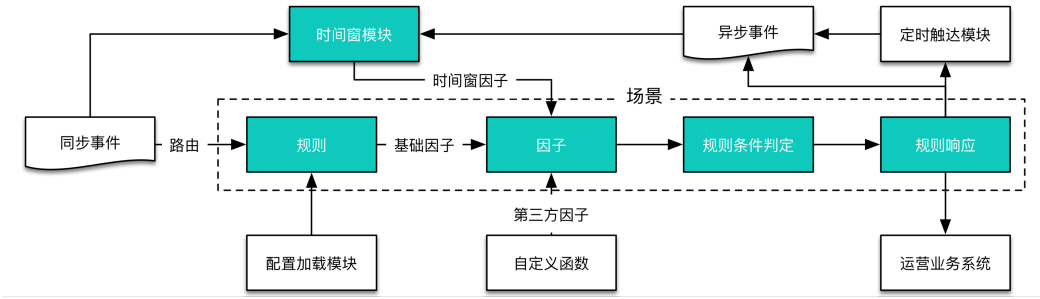


图3 规则引擎处理流程图

规则引擎在执行规则过程中，涉及以下数据模型：

- 场景：业务需求的抽象，一个业务需求对应一个场景，一个场景由若干规则组成。用不同的规则组成时序和依赖关系以实现完整的业务需求。
- 规则：规则由规则条件及因子组成，由路由至所属场景的事件触发，规则由规则条件、因子及规则响应组成。
- 规则条件：规则条件由因子构成，为一个布尔表达式。规则条件的执行结果直接决定是否执行规则响应。
- 因子：因子是规则条件的基础组成部分，按不同来源，划分为基础因子、时间窗因子和第三方因子。基础因子来源于事件，时间窗因子来源于时间窗模块获取的时间窗数据，第三方因子来源于第三方服务，如用户画像服务等。
- 规则响应：规则执行成功后的动作，如将复合事件下发给运营业务系统，或发送异步事件进行后续规则判断等。
- 事件：事件为系统的基础数据单元，划分为同步事件和异步事件两种类型。同步事件按规则路由后，不调用定时触达模块，顺序执行；异步事件调用定时触达模块，延后执行。

时间窗模块

时间窗模块是酒旅运营实时触达系统规则引擎中的重要构成部分，为规则引擎提供时间窗因子。时间窗因子可用于统计时间窗口内浏览行为发生的次数、查询首次下单时间等，表1中列举了在运营实时触达活动中需要支持的时间窗因子类型（表1 时间窗因子类型）：

类型	示例	因子构成
count	近X分钟浏览POI大于Y次	count(timeWindow(event.id, event.userId, X * 60))
distinct count	近X分钟浏览不同POI大于Y次	count(distinct(timeWindow(event.id, event.userId, X * 60)))
first	近X天支付的首单酒店	first(timeWindow(event.id, event.userId, X * 60))
last	近X天最后一次搜索的酒店	last(timeWindow(event.id, event.userId, X * 60))

根据时间窗因子类型可以看出，时间窗因子有以下特点：

1. 时间窗存储中需要以List形式保存时间窗详情数据，以分别支持聚合及详情需求。
2. 时间窗因子需要天粒度持久化，并支持EXPIRE。
3. 时间窗因子应用场景多，是许多规则的重要组成因子，服务承受的压力较大，响应时间需要在ms级别。

对于以上特点，在评估使用场景和接入数据量级的基础上，我们选择公司基于Tair研发的KV的存储服务Cellar存储时间窗数据，经测试其在20K QPS请求下，TP99能保证在2ms左右，且存储方面性价比较高，可以满足系统需求。

在实际运营活动中，对时间窗内用户某种行为次数的判断往往在5次以内，结合此业务场景，同时为避免Value过大影响读写响应时间，在更新时间窗数据时设置阈值，对超出阈值部分进行截断。时间窗数据更新及截断流程如图4所示：

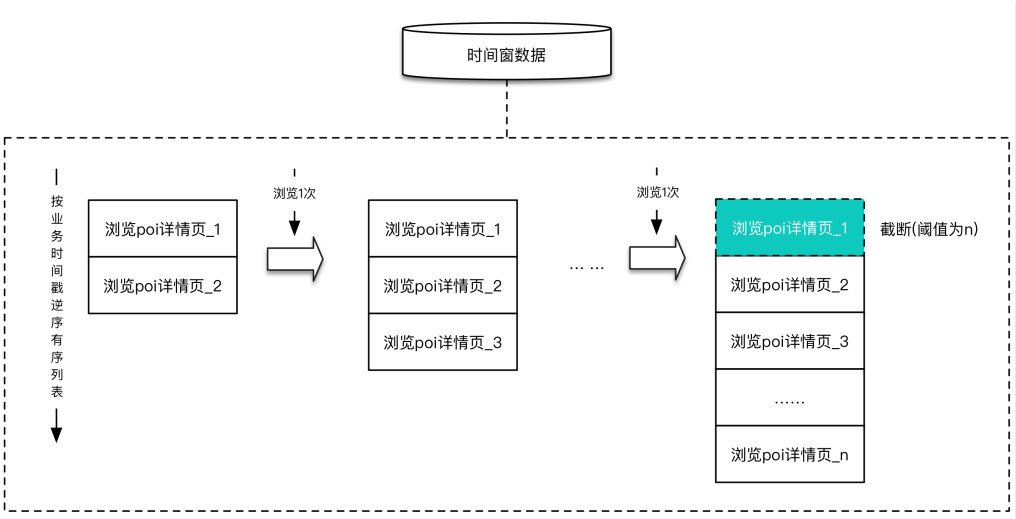


图4 时间窗数据更新示意图

文章最前面背景中提到的业务场景，在 1．用户在30分钟内发生A行为次数大于等于3次 、 3．用户在A行为前24小时内未发生B行为 、 4．用户在A行为后30分钟内未发生B行为（排除30分钟内用户自发产生B行为的影响，降低对结果造成的偏差） 中，均使用了时间窗模块对滑动时间窗内的用户行为进行了统计，以时间窗因子作为规则执行判断的依据。

规则引擎扩展组件

规则引擎扩展组件在核心组件的基础上，增强规则引擎功能。

自定义函数

自定义函数可以扩充Aviator功能，规则引擎可通过自定义函数执行因子及规则条件，如调用用户画像等第三方服务。系统内为支持运营需求扩展的部分自定义函数如表2（自定义函数示例）所示：

名称	示例	含义
equals	equals(message.orderType, 0)	判断订单类型是否为0
filter	filter(browseList, ‘source’, ‘dp’)	过滤点评侧浏览列表数据
poiPortrait	poiPortrait(message.poiId)	根据poiId获取商户画像数据，如商户星级属性

userPortrait	userPortrait(message.userId)	根据userId获取用户画像数据，如用户常住地城市、用户新老客属性
userBlackList	userBlackList(message.userId)	根据userId判断用户是否为黑名单用户

文章最前面背景中提到的业务场景，在 2．用户为美团酒店老客，即用户曾购买过美团酒店产品 中，判断用户是否为美团酒店老客，就用到了自定义函数，调用用户画像服务，通过用户画像标签进行判定。

定时触达模块

定时触达模块支持为规则设定定时执行时间，延后某些规则的执行以满足运营活动规则。文章最前面背景中提到的业务场景，在 4．用户在A行为后30分钟内未发生B行为（排除30分钟内用户自发产生B行为的影响，降低对结果造成的偏差） 条件中，需要在A行为发生30分钟后，对用户是否发生B行为进行判定，以排除用户自发产生B行为对活动效果造成的影响。

定时触达模块涉及的数据流图如图5所示：

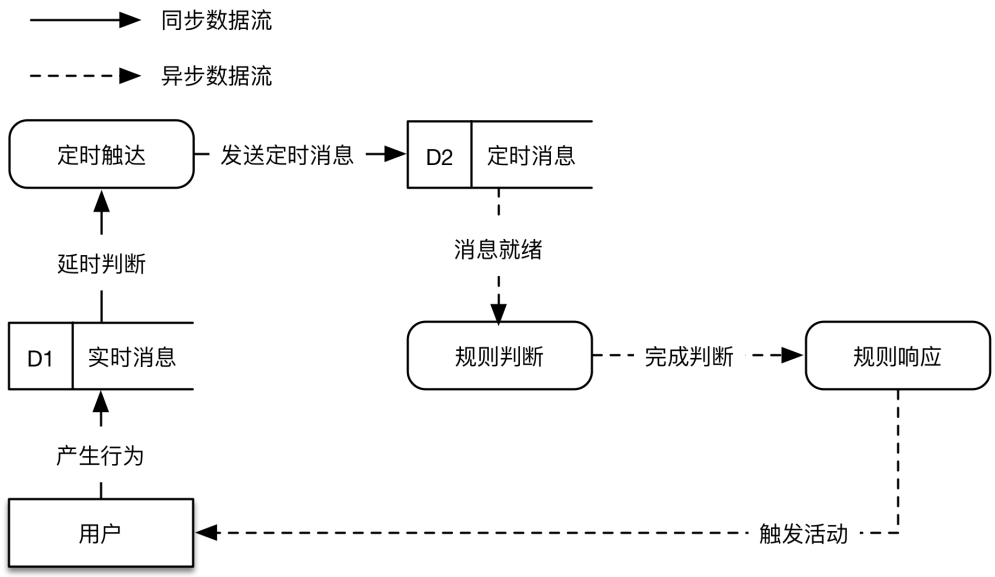


图5 定时触达模块数据流图

早期的业务需求对延迟时间要求较短，且活动总数量较小，通过维护纯内存DelayQueue的方式，支持定时触达需求。随着相关运营活动数量增多及定时触达时间的延长，纯内存方式对内存的占用量越来越大，且在系统重启后定时数据会全部丢失。在对解决方案进行优化时，了解到公司消息中间件组在Mafka消息队列中支持消息粒度延迟，非常贴合我们的使用场景，因此采用此特性，代替纯内存方式，实现定时触达模块。

监控与报警

对比离线数据，实时数据在使用过程中出现问题不易感知。由于系统针对的运营活动直接面向C端，在出现系统异常或数据质量异常时，如果没有及时发现，将会直接造成运营成本浪费，严重影响活动转化率等

活动效果评估指标。针对系统稳定性问题，我们从监控与报警两个角度入手解决。

监控

利用公司数据平台现有产品，对系统处理的实时事件按其事件ID上报，以时间粒度聚合，数据上报后可实时查看各类事件量，通过消息量评估活动规则和活动效果是否正常，上报数据展示效果如图6所示：

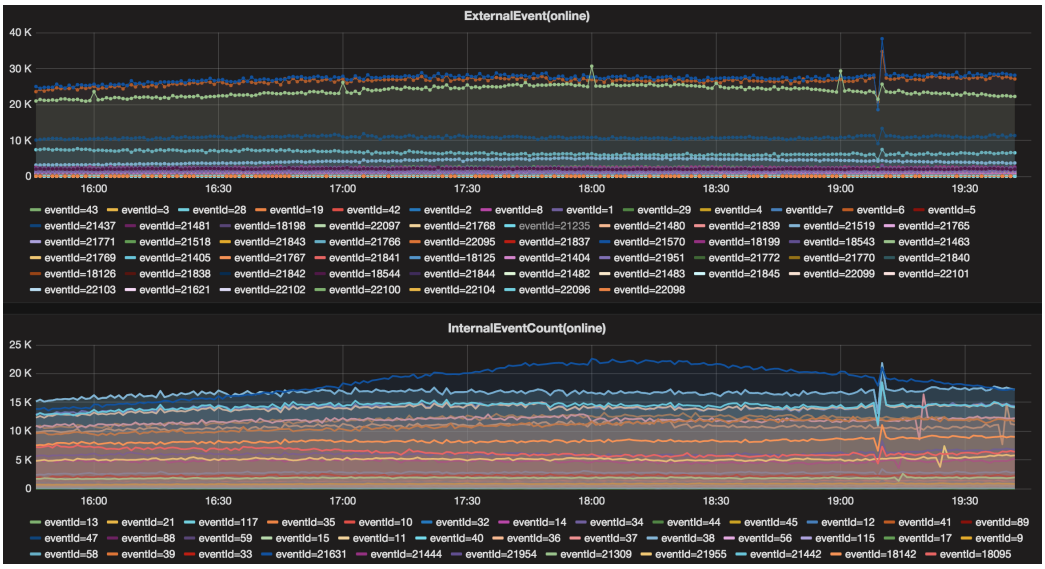


图6 实时事件监控图

报警

监控只能作为Dashboard供展示及查看，无法实现自动化报警。由于用于监控所上报的聚合数据存储于时序数据库OpenTSDB中，我们基于OpenTSDB开放的HTTP API，定制报警模块，定时调度、拉取数据，对不同事件，按事件量级、活动重要性等指标，应用环比、绝对值等不同报警规则及阈值。超出设定阈值后，通过公司IM及时发送报警信息。如图7所示，该事件环比出现数据量级下降，收到报警后相关负责人可及时跟踪问题：

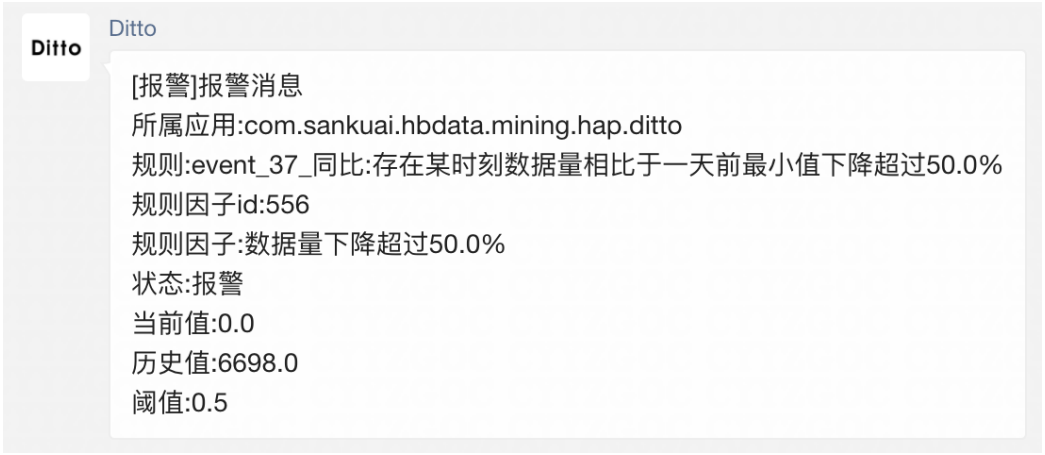


图7 报警信息示意图

总结与展望

酒旅运营实时触达系统已上线稳定运行一年多时间，是运营业务中十分重要的环节，起到承上启下的作用，在系统处理能力及对业务贡献方面取得了较好的效果：

- 平均日处理实时消息量近10亿。
- 峰值事件QPS 1.4万。
- 帮助酒店、旅游、大交通等业务线开展了丰富的运营活动。
- 对转化率、GMV、拉新等指标促进显著。

当前系统虽然已解决了业务需求，但仍存在一些实际痛点：

- 实时数据接入非自动化。
- 规则引擎能力需要推广、泛化。
- 场景及规则注册未对运营PM开放，只能由RD完成。

展望未来，在解决痛点方面我们还有很多路要走，未来会继续从技术及业务两方面入手，将系统建设的更加易用、高效。

作者简介

- 晓星，美团平台技术部－数据中心－数据智能组系统工程师，2014年毕业于北京理工大学，从事Java后台系统及数据服务建设。2017年加入美团点评，从事大数据处理相关工作。
- 伟彬，美团平台技术部－数据中心－数据智能组系统工程师，2015年毕业于大连理工大学，同年加入美团点评，专注于大数据处理技术与高并发服务。

招聘

最后发个广告，美团平台技术部－数据中心－数据智能组长期招聘数据挖掘算法、大数据系统开发、Java后台开发方面的人才，有兴趣的同学可以发送简历到**lishangqiang#meituan.com**。

美团配送资金安全治理之对账体系建设

作者: 甄超 宏伟

前言

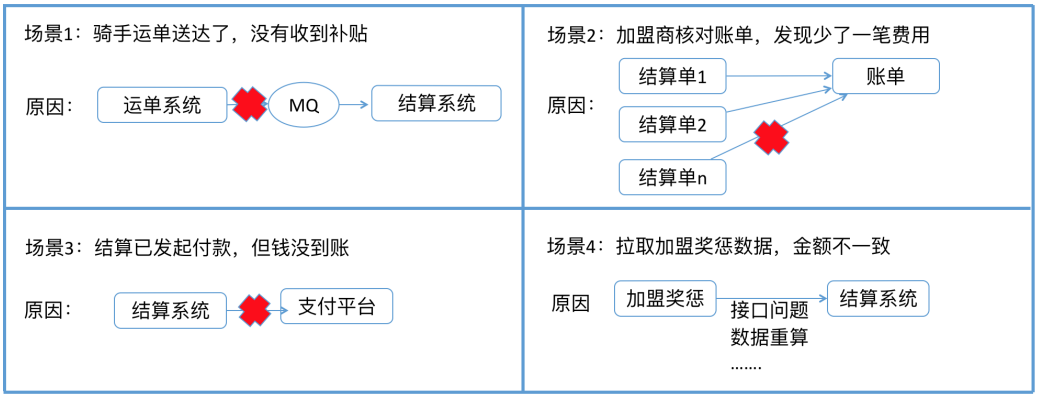
随着美团配送业务的飞速发展，单量已经达到千万级别，同时每天产生的资金额已经超过几千万，清结算系统在保证线上服务稳定可靠的前提下，如何系统化的保障资金安全是非常核心且重要的课题，配送清结算系统经过近3年的建设和打磨，在资金安全保障的多个方面均有一些总结和实践，保障资金安全是值得系统思考的课题，只言片语难以全面概括，需要更多的着墨才能较完整阐述，本文侧重点会阐述“对账”的概念，在支付&清结算领域，这是一个非常重要的专业名词，下文将介绍“对账”在分布式系统建设中的实践和解决方案，力求在系统覆盖度、资金准确性、时效等多个维度为系统资金安全保驾护航，实现更健壮可靠的资金履约。

背景&问题

随着美团外卖配送事业的蓬勃发展，配送清结算业务的复杂性也在不断的增高，总结起来，主要有以下几个特点：

- 场景多：包括专送、众包、快送、跑腿、外部单等多条业务线；订单补贴、活动发放、奖惩、餐损、打赏、保险等多种结算场景；对接外部十多个系统。
- 链路长：清结算内部经历定价、记账、汇总账单、付款等多个流程。
- 单量大：目前日单量已达到千万级别。

在这样的业务背景下，我们的系统可谓险象环生。因业务高度复杂，稍有不慎就会出现问题，面对千万级的日单量，同时还要确保结算金额的准确，这就让我们对问题的容忍度变得极低。这也给我们的资金安全保障造成了巨大的挑战。下面我们列举了一些系统日常运行过程中出现的问题。



场景

可以看出，这些都是一些上下游交互的边界场景，以及遇到的问题。当然不仅限于这些场景，凡是有系统交互、数据交互边界的场景，都会出现此类问题，我们称之为“一致性问题”。经粗略统计，我们清结算系统建立以来有70%左右的问题都属于一致性问题。

导致一致性问题的原因有很多，诸如：

- 幂等、并发控制不当。
- 基础环境故障：比如网络、数据库、消息中间将发生故障。
- 其他代码bug。

目前配送的日结算金额已达到千万级别，每个一致性问题都有可能给我们整个美团造成巨大的损失。因此如何解决系统的一致性问题成为我们保障资金安全的重中之重。关于一致性问题，业内已经论述的非常成熟了，搜索引擎中搜索“一致性问题”，随处可见此概念的定义、问题阐述、意义以及解决思路，诸如：

> “
>
> - **强一致性协议: 两阶段提交、三阶段提交、TCC (Try-Confirm-Cancel)等**
> - **最终一致性: 主动轮询、异步确保、可靠消息、消息事务等**

这些手段的目标都是在事中避免问题的发生。但是在实际场景中，无论是系统的内部逻辑还是外部环境都十分复杂多变、不可预知，我们很难完全避免问题。因此事后对于问题数据的发现以及修复就显得尤为重要。这些也正是我们这篇文章要论述的“对账”的核心使命。我们力求总结对账领域内最专业的思路和方法，并结合自身的业务特点，建设配送清结算的对账体系，构筑配送资金安全的坚固防线。在系统的整个构建过程中我们主要围绕以下几个目标：

- 场景覆盖的完整性：无死角覆盖清结算业务涉及的各个场景。
- 问题发现的准确性：能够准确的发现问题，保证不漏报，不误报。
- 问题处理的实时性：尽可能缩短问题处理的周期，极力避免可能造成的损失。

下面开始正式介绍美团配送清结算对账体系的构建经验。

对账的定义

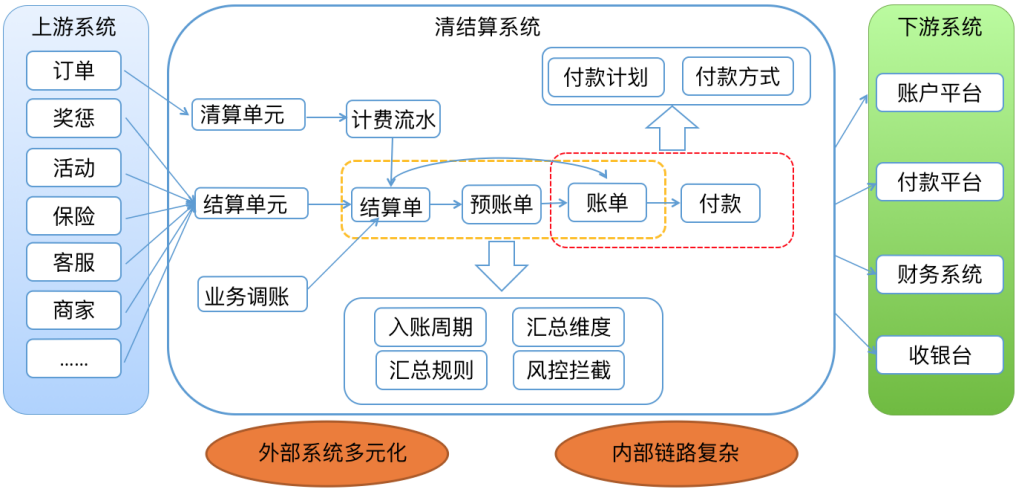
对账的概念随着金融、互联网行业的发展，定义上也经历了几个阶段的变化，如下：

- stage 1： 对账最初来源于**会计核算**，是为保证账簿记录正确可靠，对账簿中的相关数据进行检查和核对的工作。
- stage 2： 随着**互联网金融或电商**行业的发展，对账也扩大了应用范围，这一时期，对账是指在固定周期内，支付使用方和支付提供方（银行和第三方支付）相互确认交易、资金的正确性，保证双方的交易、资金一致正确。
- stage 3： 从广义来看，**所有的跨端系统**之间的数据核对都应该叫对账，主要是检查和发现数据在流转过程中的不一致问题。通常分为信息流的核对和资金流的核对。信息流核对主要是对业务数据之间的核对，资金流是对资金交易数据进行核对。

对账系统的构建思路

系统概况

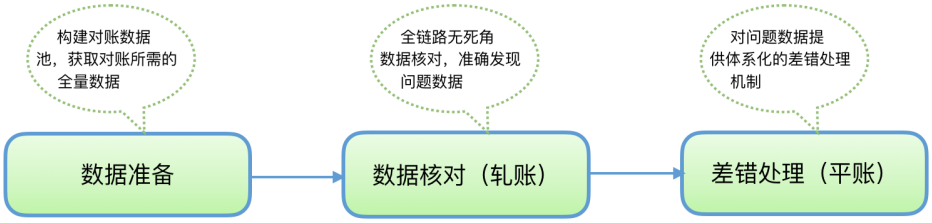
配送结算做为核心交易履约系统，上游对接了订单、奖惩、活动等十多个外部系统，下游又承担了对接支付平台、财务系统的职责，不仅“承上启下”，而且涉及业务复杂。而系统内部又历经定价、计费（清算）、记账、汇总账单、付款等多个环节，系统的高度复杂性给对账的全面性和准确性造成了极大的困难，如图：



为了系统更加专业化的实现对账、做好对账，我们对支付、清结算等资金领域进行了体系化的调研和学习，并结合业务的自身的特点，总结了一套对账系统构建的思路方法，并基于该思路进行了较完整的系统化实现。

设计思路

从整体来看，按照时序维度的先后，系统对账主要分为三阶段的工作。分别是**数据准备**、**数据核对**和**差错处理**。在对账专业概念中，数据核对和差错处理又叫**轧账**和**平账**。三个环节紧密相连，从前期准备、问题发现、问题处理三个角度展开对账工作。



数据准备

数据准备，顾名思义，我们需要把对账所需的全部数据，接入到我们的对账系统。该模块主要实现两个目标：

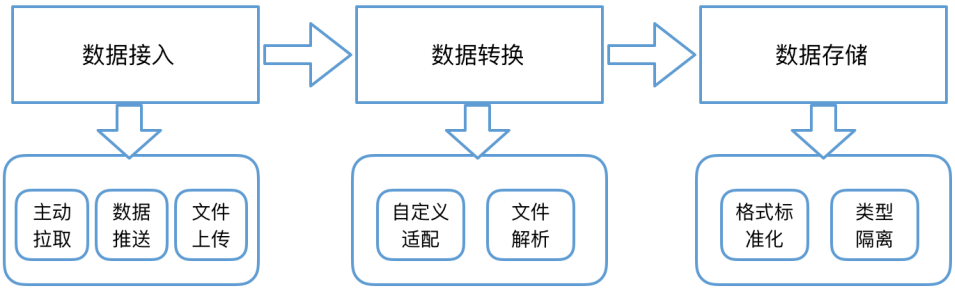
- 为不同的外部系统提供多元化的接入机制。
- 通过数据适配的手段把外部数据以统一的格式进行转换和存储。

在数据接入层，我们会针对不同的数据接入方提供三种不同的数据接入模式。

- 数据拉取：我们主动拉取数据，并通过数据适配的方式，将数据存储到对账数据池中。
- 数据推送：由数据接入方将数据通过ETL（Extract-Transform-Load）等方式直接推送到我们的对账数据池中，数据格式由数据接入方自行适配。
- 文件上传：我们会提供标准的文件模板，由数据接入方填充数据，通过文件上传的方式将数据接入到我们的数据池。

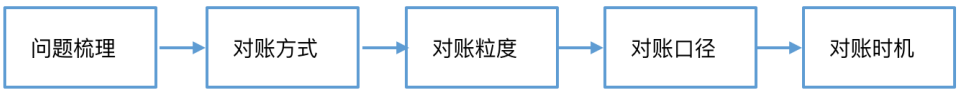
其中第二种方式是我们最优选择的，因为数据推送这种形式对于数据接入方来说只需要一次性编写相关的代码，定期运行，一劳永逸，减少了人工上传的成本。对于我们结算来说，也不需要感知对方的数据格式

以及业务逻辑。



数据核对（轧账）

数据核对是对账中最核心的一个阶段。其目标是发现问题数据。数据核对阶段我们的两个目标是保障数据核对的 **覆盖度** 和 **准确性** 。经过总结和梳理，数据核对过程可以分为以下5个环节。



1. 问题梳理

由于数据核对的目标是发现问题，那么我们进行数据核对就要从问题出发，首先明确我们要通过对账发现哪些问题，只有这样才能保证数据核对的覆盖度。经过梳理，我们发现在数据流转中过程中数据的不一致问题可以统一归结为三类，分别是 **漏结** 、 **重复结** 、 **错结** 。我们可以从这三个角度去统一进行问题梳理。下面介绍一下这三种错误类型的具体含义。

- 漏结：发起方有数据，而接收方没有数据。举个例子，目前清结算系统会在订单送达时给骑手结算。如果订单的状态是送达，而没有给骑手生成对应的结算数据。就是一种典型的漏结算场景。
- 重复结：接收方重复处理。还是上面的例子，如果订单送达，给骑手结算了两次，产生了重复的结算数据，就是重复结算。
- 错结：发起方和接受方数据不一致。一般会发生在金额和状态两个字段。比如说订单上的数据是用户加小费3元。结算这边只产生了2元的小费结算数据，就是错结。

2. 对账方式

对账方式主要分为两种， **单向对账** 和 **双向对账** 。

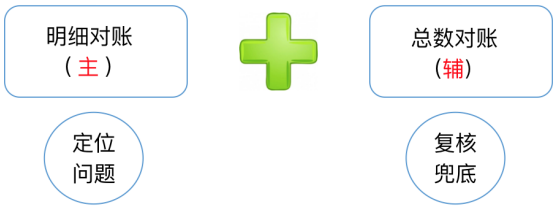
- 单向对账：以一方数据为基准进行对账。比如结算跟支付平台，以结算数据为基准和支付平台核对，用来发现结算数据为支付成功，支付平台支付失败等问题。
- 双向对账：以双方的数据互为基准对账。既要保证结算数据为成功的，支付平台也要成功，又要保证支付平台数据为成功的，结算数据也要成功。

显而易见，双向对账更能够全面的发现问题。因此在条件允许的情况下，我们会优先选择双向对账。

3. 对账粒度

对账粒度也分为两种，分别是 **明细对账** 和 **总数对账** 。

- 明细对账：对双方的每条数据依次进行比对。它的优点是可以准确定位问题数据。缺点是对账口径的设计比较复杂。因为我们需要同时针对漏、重、错三种错误类别设计不同的对账口径，同时还要考虑到业务的边缘场景。稍有不慎，就会影响对账的准确性。
- 总数对账：选择一个维度，进行总数级别的对账。总数级别的对账好处是对账口径的设计比较简单，可以快速实现，不易出错。缺点就是无法定位问题数据，一旦对账发现问题。还需要进一步寻找问题数据。

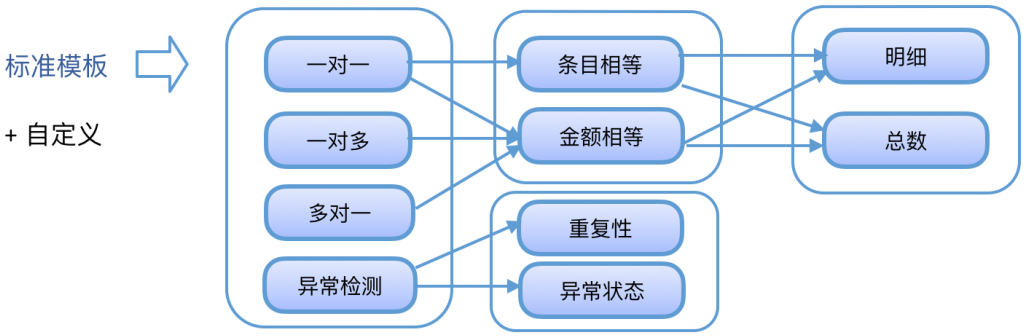


因此，推荐的做法应该是以明细对账为主，定位具体问题。以总数对账为辅，对明细对账的结果进行复核兜底。

4. 对账口径

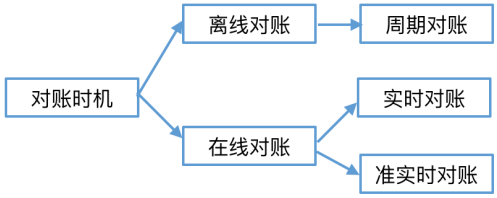
对账口径，也就是具体的对账逻辑的设计。我们会提供固定的 **对账模板** ，供不同的对账场景选取。如果某些特殊场景对账模板不能覆盖，也可以采取对账逻辑 **自定义** 的方式进行对账。

经过总结我们发现，对账的形式无非就是两方比对和自身异常检测两种。两方比对又可以细分为一对一、多对一、一对多。比对方法也主要是分为条目匹配和金额匹配。自身异常检测主要是重复性和异常状态的检测。我们把这些通用的对账逻辑模板化，减少重复的开发工作。



5. 对账时机

数据核对的最后一步就是对账时机的选择。分为 **离线对账** 和 **在线对账** 。离线对账主要是通过固定的周期进行对账。最短周期为 T + 1 。它的好处是适用性较强，基本可以覆盖所有的对账场景。而在线对账又分为 **实时对账** 和 **准实时对账** 。实时对账和准实时对账的区别主要是实时对账耦合在结算链路中，可以在发现问题数据时，对结算流程进行拦截，而准实时对账是异步进行的，不具备拦截能力。在线对账有一定的局限性，一方面它依赖于对账数据是否能实时的准备好，另一方面也比较占用系统资源。因此我们的做法应该是以周期对账为主，在某些实时性要求比较高，且条件满足的场景使用在线对账。



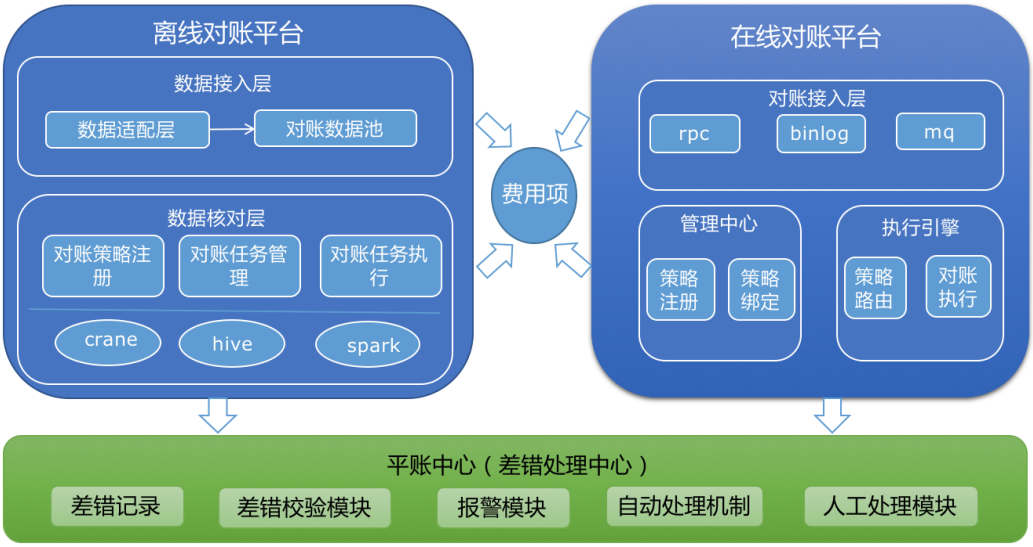
差错处理（平账）

差错处理主要是对数据核对过程中发现的问题数据进行处理。我们会建立一个统一结构的差错记录，将数据核对发现的问题进行统一存储。差错记录中的数据会进行二次核对，避免由于日切等原因造成的问题错报。对于那些真实存在问题的数据我们会提供两种解决模式，如果是常见的问题，且有一套标准的解决方案的话，我们会把它系统化，采取系统自动修复的方式；如果系统无法自动修复，那么我们会进行系统报警，并进行人工处理。

对账系统设计实现

总体架构

综上所述，对账体系的整体架构，分为三个模块，分别是离线对账平台，在线对账平台和平账中心。完全是按照我们上面的对账思路设计的。三个模块互相协作，一体化的完成数据准备、数据核对、问题处理三部分工作。由于我们整个清结算系统是围绕不同的费用项建立的，因此费用项也是我们设计对账、执行的对账一个最小粒度的单元。



具体介绍下三个模块。

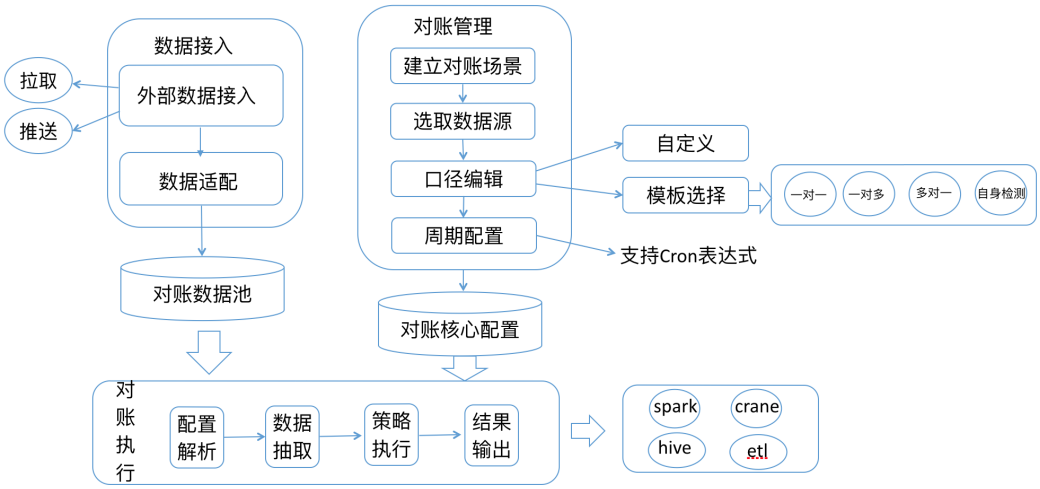
离线对账模块

离线对账分为三个子模块，分别是数据接入层、对账管理层和对账执行层。

在数据接入层我们提供拉取和推送两种模式，经历一个数据适配的过程，将数据存储到我们统一的对账数据池当中。

在对账管理层当中，我们抽象出了一个对账场景的概念，我们基于对账场景进行对账属性的配置：首先要选取对账双方的数据源；然后进行对账口径编辑，这里提供了自定义和模板选择两种方式；最后配置对账的周期。这里我们是通过cron表达式来进行周期配置的。

在对账执行层，我们会拉取对账数据池和对账核心配置中的相关数据，经历配置解析，数据抽取，策略执行的过程，最终输出对账结果。



在线对账模块

下图左边是在线对账平台的架构图，右边是在线对账的实例。我们通过RPC、监听消息队列(MQ)、监听数据库binlog三种方式进行对账接入。在线对账平台分为管理层和执行层。管理层主要是承担策略编辑、策略绑定和拦截管理的相关工作。而执行层分为异步（准实时）对账和同步（实时）对账两个模块。

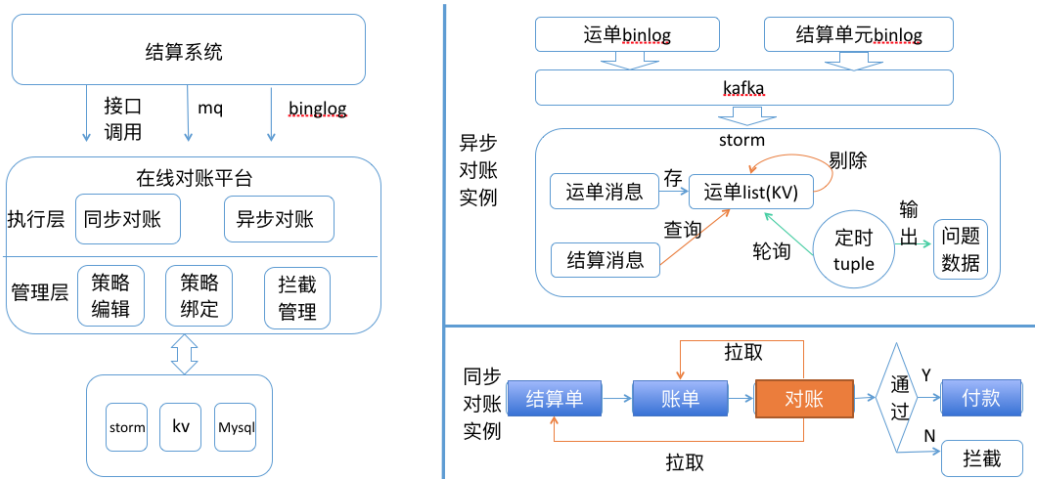
右边两图分别是分别是异步对账和同步对账的实例。在异步对账的实例中，是运单和结算单元的对账。

> - **运单是什么？对应外卖订单，作为配送内部的基础交易数据。**
> - **结算单元是什么？ 清结算系统内部的模型，和运单是一对一关系，记录运单各个节点的结算状态。**

①异步对账：我们分别监听运单和结算单元的Binlog，通过Kafka->Storm的经典架构，进行对账策略的执行。实际的流程比较复杂，这里只是一张简图，大概就是：（细节可以忽略）

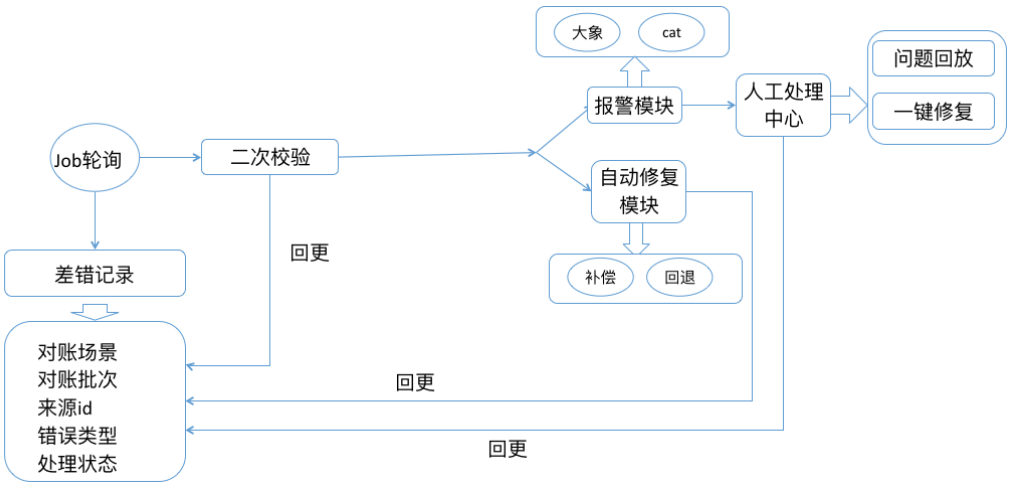
> 收单运单消息后，我们会把对于的运单以List的形式存储到Squirrel(Redis)中，当结算消息来了以后，就把对应运单记录Delete掉。如果有运单记录一直停留在List当中，也就是说明结算消息没有来，应该是发生了漏结算。我们通过过定时任务轮询运单List将问题数据输出。

②同步对账：示例中是结算内部的流程，经历结算单、账单、付款几个流程。因为付款是最后一个流程，如果这个时候数据存在问题，那么就会造成实际的资金损失。因此我们会在付款环节之前，对前面的数据进行对账。如果发现账单和结算单的数据不一致，我们就会进行数据拦截。



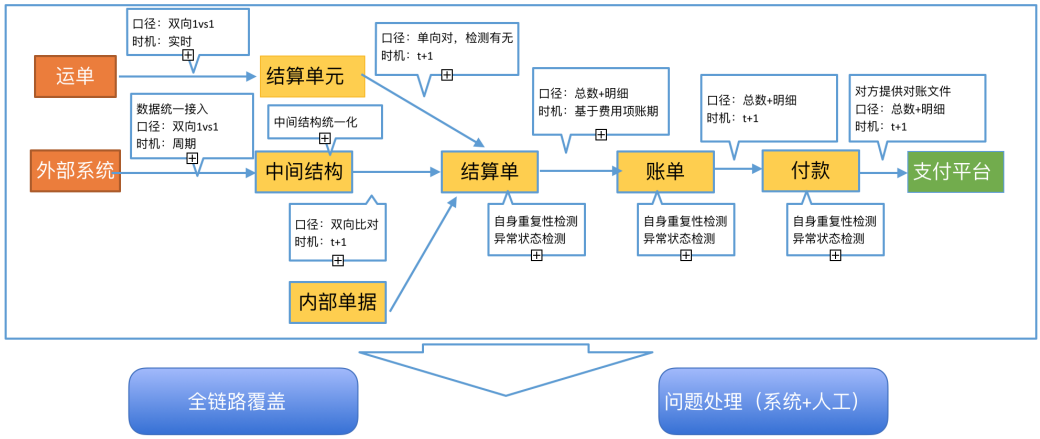
差错处理模块

在差错处理阶段，我们会建立一个统一的差错记录模型，核心字段包括 **对账场景、对账批次，数据来源，错误类型编码和数据处理状态** 等。通过定时任务定期轮询差错记录的方式发起差错处理流程：首先对差错记录的数据进行二次核对，如果二次核对确认这条数据并没有问题，我们就会回更差错记录的处理状态。如果二次核对发现数据确实有问题。我们会提供两种处理模式。一种是通过系统的手段自动修复。另外一种是通过报警的方式，人为介入。此外我们还建立了一个问题的人工处理模块。可以对一次结算流程的整个生命周期进行回放，并针对特定场景提供一键修复的能力。



小结与展望

按照计划实施后，系统的各个节点都会有行之有效的对账手段覆盖，实现资金安全、数据一致性的保障，示意图：



本篇文章的内容是我们根据业务的特点，经过长期的思考和外部调研，总结的一套关于对账的思路以及实施落地方法。目前我们对账体系还在分布实施阶段，我们最终的目标是：

- 覆盖度：实现全链路无死角的对账。
- 处理效率：对于问题的处理尽可能的去人工化，实现自动化或者工具化。
- 接入成本：后续新的业务场景实现对账尽可能的降低成本。

目前外卖配送的单量与日剧增，资金安全所面临的挑战越来越大。需多次强调的是：资金安全是一个很大的课题，需要投入大量的时间和精力去系统思考，对账只是其中一环。我们目前围绕资金安全进行了一系列的治理动作，未来还将会继续加强我们对于资金安全的理解深度，通过更多的对外交流和学习丰富我们保障资金安全的手段。

作者简介

- 甄超，2015年9月加入美团，配送清结算系统核心成员，专注于清结算架构建设、资金安全治理工作。
- 宏伟，2015年4月加入美团，配送清结算系统负责人，参与了美团配送系统建设的全过程。

招聘信息

外卖配送是个挑战与机遇并存、荣誉与艰辛同在的业务，优秀的项目也同样需求优秀的同学，配送订单调度、清结算系统长期招聘资深工程师、技术专家、架构师等岗位，欢迎各位志同道合、能力优秀的人加入。如果有意向，请发简历至 zhanghongwei#meituan.com

美团酒旅起源数据治理平台的建设与实践

作者: 李鹏 夷山 永超

背景

作为一家高度数字化和技术驱动的公司，美团非常重视数据价值的挖掘。在公司日常运行中，通过各种数据分析挖掘手段，为公司发展决策和业务开展提供数据支持。

经过多年的发展，美团酒旅内部形成了一套完整的解决方案，核心由数据仓库+各种数据平台的方式实现。其中数据仓库整合各业务线的数据，消灭数据孤岛；各种数据平台拥有不同的特色和定位，例如：自助报表平台、专业数据分析平台、CRM数据平台、各业务方向绩效考核平台等，满足各类数据分析挖掘需求。早期数据仓库与各种数据平台的体系架构如图1所示：

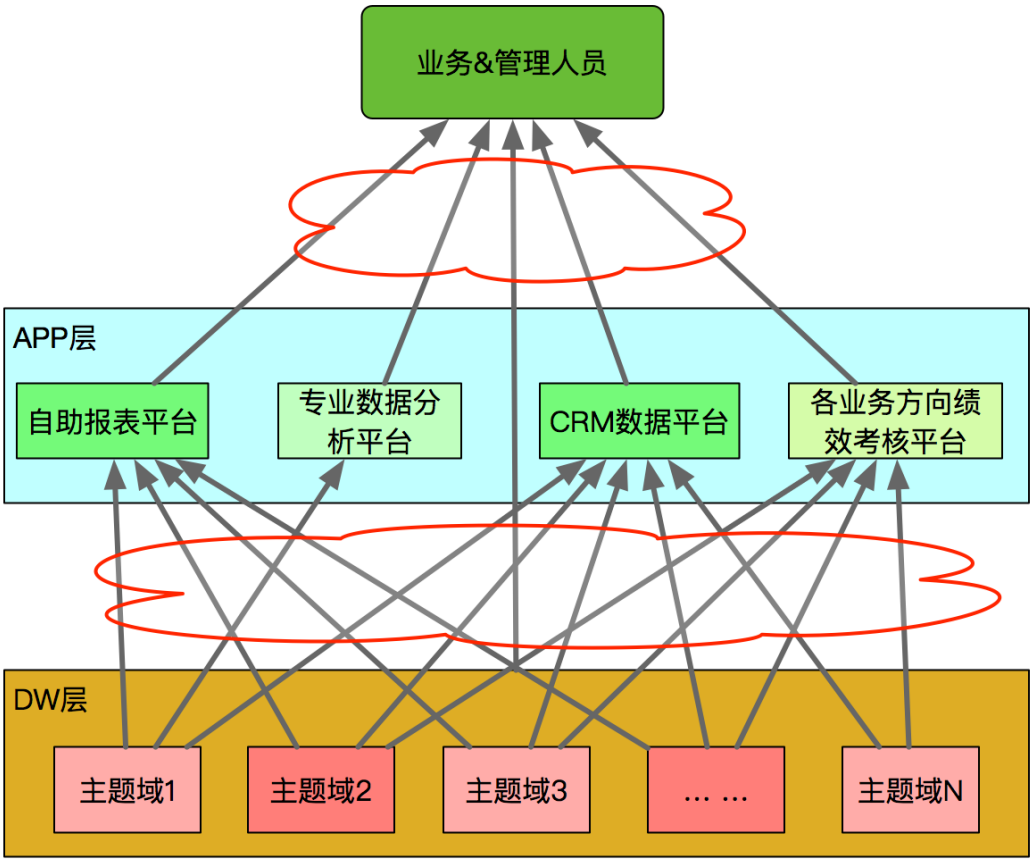


图1 酒旅早期各数据平台和数据仓库体系架构图

图1所示的体系架构，在业务需求的满足上非常高效，但在长时间的使用过程中，也产生了如下一些问题：

- 各数据平台或平台内不同模块的指标定义不一致。
- 各数据平台或平台内不同模块指标计算口径不一致。
- 各数据平台或平台内不同模块指标数据来源不一致。

上述这些问题总结归纳起来，就是指标数据不一致的问题，最终带来的后果是指标数据可信度底，严重影响分析决策。通过后续追踪分析，上述问题的由来，主要是不同业务线的数据分析人员、数据开发人员，以及不同的产品之间，缺乏有效的沟通，也没有一个统一的入口，来记录业务的发生和加工过程。在加上人员的流动，长时间积累之后就产生了这些问题。针对这些问题，酒旅内部启动了数据治理项目，通过建设一个专业数据治理平台，实现指标维度及数据的统一管理，也探索一套高效的数据治理流程。

挑战

在建设起源数据治理平台的过程中，主要面临的挑战如下：

- 起源数据治理平台应该在架构中的哪个位置切入，减少对原有系统的侵入，并实现数据治理目标。
- 探索一套简洁高效的管理流程，实现指标维度信息统一管理，保证信息的唯一性、正确性。
- 整合各种存储引擎，实现一套高并发、高可用的数据唯一出口。
- 做好各业务线间的信息隔离和管理，确保数据安全。

解决思路

为了达成数据治理的目标，起源数据治理平台就必须记录下业务发展过程，并映射到数据加工和数据提取，规范约束这些过程。因此起源数据治理平台归纳到数据治理层，该层就位于数据仓库层（或数据集市层）之上，数据应用层之下起到桥梁的作用，而且提供一系列规则，改变原来无序交互方式，将数据仓库层和数据应用层的交互变为有序的、可查询、可监控。新的体系架构如图2所示：

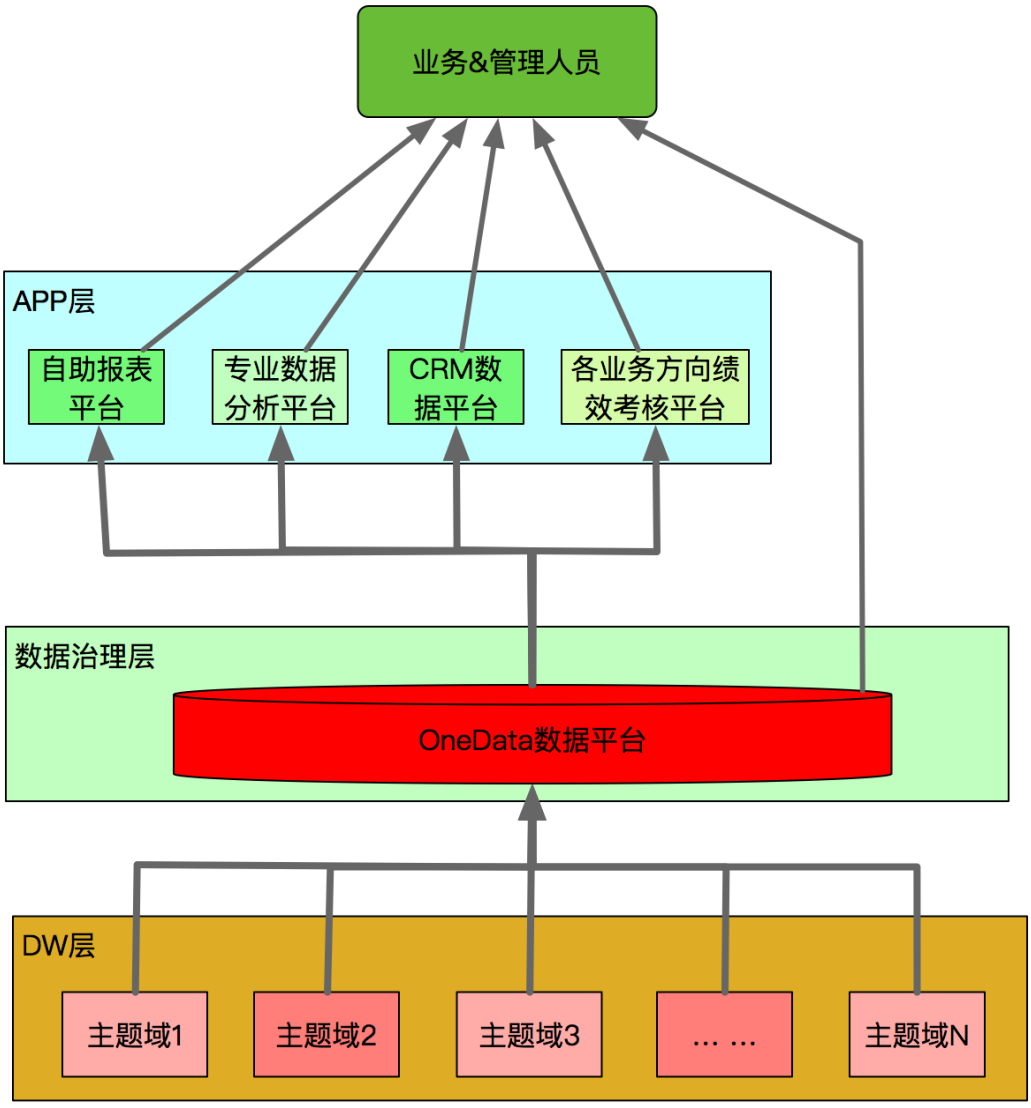


图2 数据治理后的新体系架构图

如上图所示，在新的体系架构下：对于数据仓库层，起源数据治理平台综合业务组织形式、指标数据来源、上层产品的使用及查询的效率，指导数据仓库模型的建设；对于应用层的产品，业务元数据信息及数据信息都是由起源数据治理平台提供，保证了各数据产品获取到的信息一致，而且还简化了应用层产品数据获取成本，也降低了对原有系统的侵入。

平台架构

起源数据治理平台核心是保证数据一致，在数据安全的前提下，尽可能提升数据分发能力。因此平台内部有着极其复杂的关系，需要在建设过程中进行抽象，形成具有相对单一功能的模块；合理地组织模块的层级和连接关系，降低平台的开发难度，并提升平台的可维护性。平台架构如图3所示，展示了平台的内部模块组织方式。

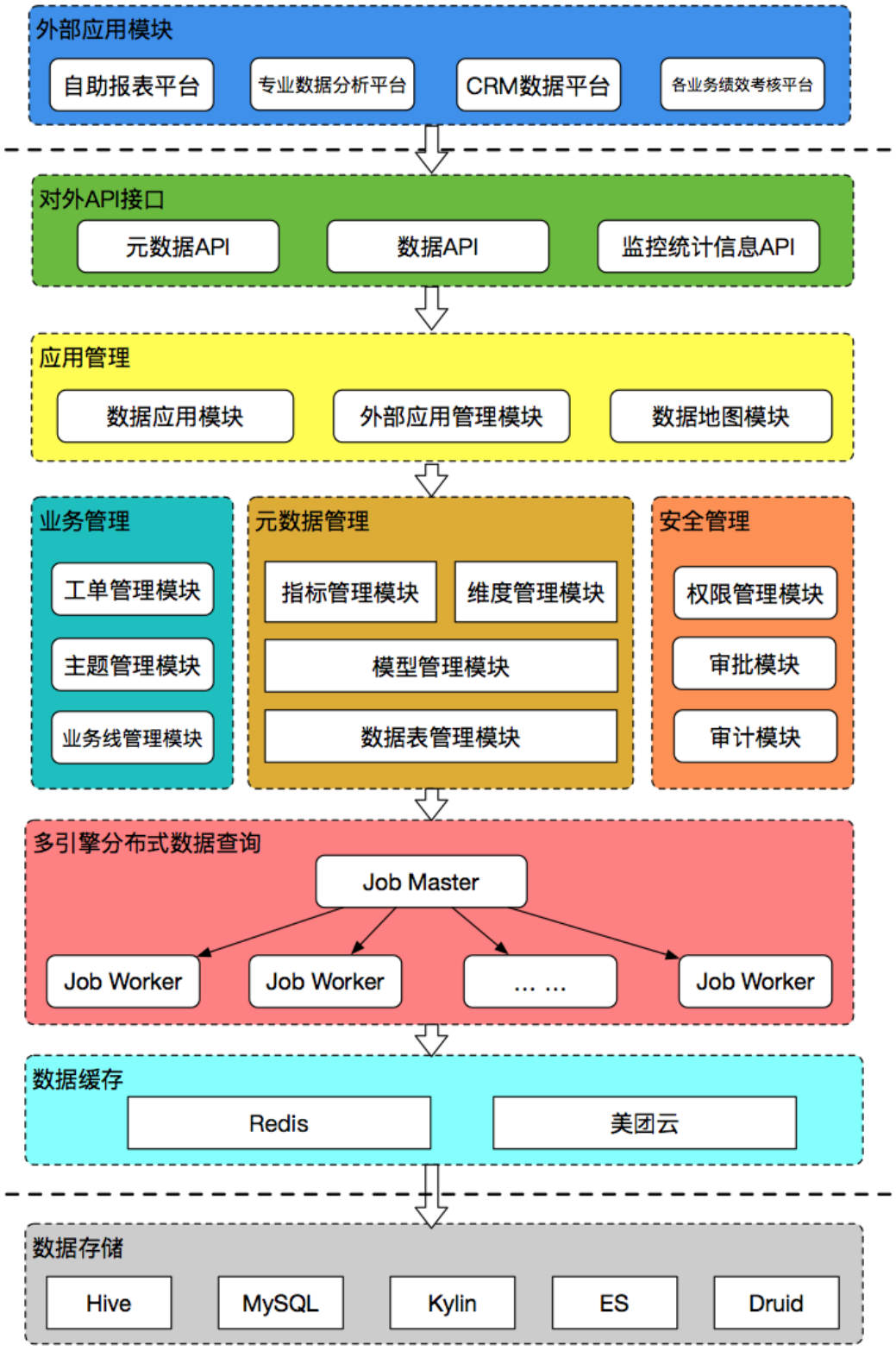


图3 起源数据治理平台架构图

如上图所示起源数据治理平台在功能模块上由数据存储、数据查询、数据缓存、元数据管理、业务管理、安全管理、应用管理、对外API接口构成，各模块的功能介绍如下。

数据存储

起源数据治理平台管理的数据存储范围包括：数据仓库中的Topic层和数据应用层，存储方式包括：Hive、MySQL、Kylin、Palo、ES、Druid。如下图4所示：

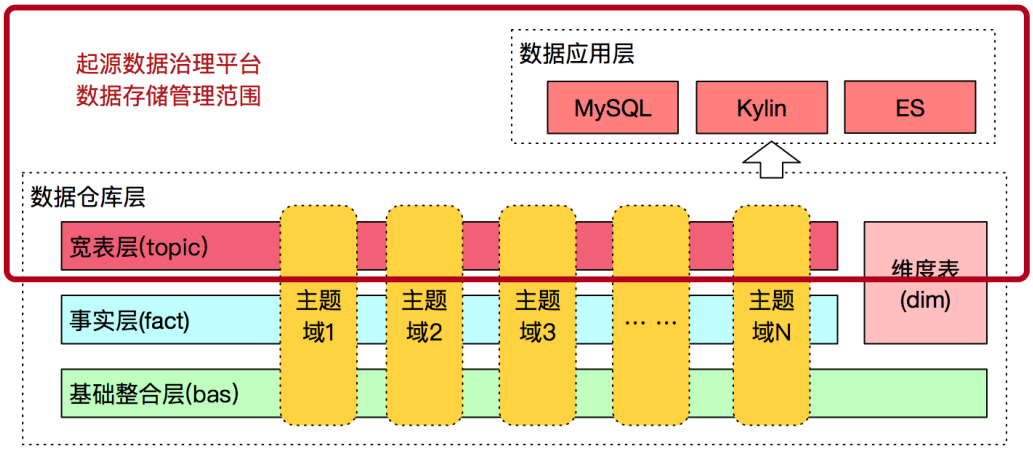


图4 起源数据治理平台管理的数据存储

上图所示的这些数据存储中的数据的加工过程，由数据开发工程师负责，具体采用哪种存储介质，由数据开发工程师综合所需数据存储空间、查询效率、模型的组织形式等因素决定。但后续的使用维护都由起源数据治理平台管理，管理方式是通过管理这些数据表的元数据信息和查询实现，具体实现细节会在下面章节中详解。

数据存储托管之后，数据表元数据信息变更监控、表数据生产（存储空间、生产状态及完成时间）监控、表数据波动（同环比等）监控以及表的使用（模型的构建及查询效率等）监控及评估，都由起源数据治理平台自动完成，所有信息的变动都会自动周知对应的负责人，保证数据应用的安全和稳定。

元数据管理

元数据信息宏观上包括两大部分：业务元数据信息和数据元数据信息。其中业务元数据信息包括：指标业务定义、维度的业务定义等；数据元数据信息包括：数据表元数据信息、模型元数据信息、维表与维度的绑定关系、数据模型字段与指标的绑定关系。

起源平台为了实现元数据信息的管理，设计了四个模块实现，分别是：数据表管理模块、模型管理模块、指标管理模块、维度管理模块。元数据管理是起源数据治理平台的核心，起源平台就是通过控制好元数据，来驱动数据的生产和消费。

数据表管理模块

数据表管理模块管理了数据库信息和数据表信息。其中数据库信息包括数据库链接信息，数据库信息维护后，起源数据治理平台自动获取对应库中表的元数据信息。

数据表信息包括：表的元数据信息（引擎、字段等）、表类型（维表或事实表）、表的使用情况（是否被模型使用）、表对应的ETL、表的负责人、表的推荐度、描述信息、表的监控配置及报警历史、以及样例数据等。上述这些信息为业务用户提供指导，为模型管理提供数据支持，为数据表和数据的稳定提供监控和预警。

模型管理模块

模型管理模块能够还原业务落地后数据表的组织关系，包括：数据表的关联方式（join、left join、semi join等）、数据表的关联限制、模型ER图、模型包含字段、模型字段与维度的绑定关系、模型与指标的绑定关系。

不过在实际使用过程中，面向业务和面向分析的模型有所不同，起源数据治理平台是面向分析的，所以主要的模型包括维度建模中的星型模型或雪花模型，再就是OLAP多维分析的MOLAP或ROLAP。模型管理如下图5、图6所示：

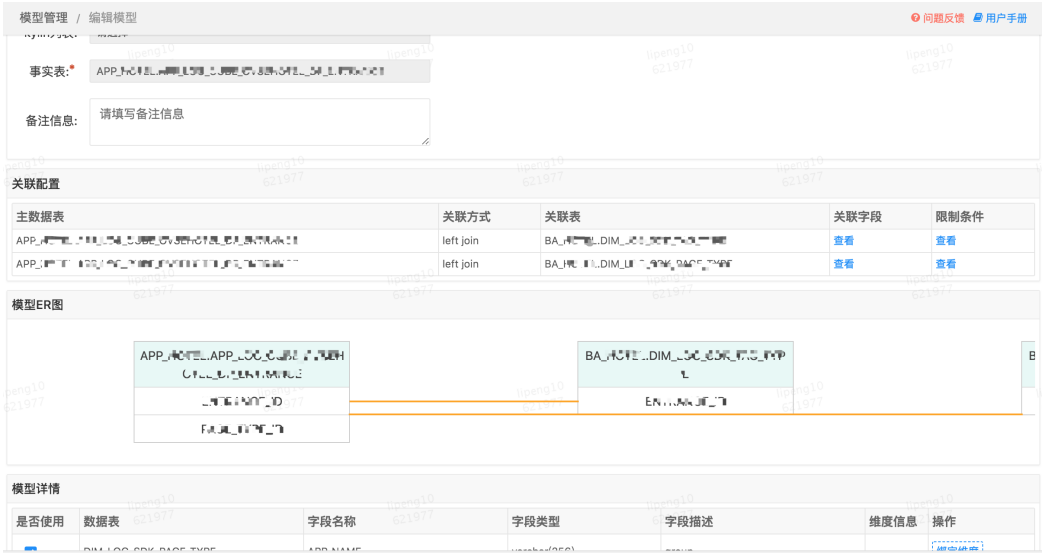


图5 起源数据治理平台数据表模型

图6 起源数据治理平台SQL模型

维度管理模块

维度管理模块包括基础信息和技术信息，对应着不同人员维护。其中基础信息对应维度的业务信息，由业务管理人员维护，包括维度名称、业务定义、业务分类。技术信息对应维度的数据信息，由数据开发工程师维护，包括是否有维表（是枚举维度还是有独立的维表）、是否是日期维、对应code英文名称和中文名称、对应name英文名称和中文名称。如果维度有维表，则需要和对应的维度表绑定，设置code和name对应的字段；如果维度是枚举维，则需要填写对应的code和name。维度的统一管理，有利于以后数据表的标准化，也方便用户的查看。

指标管理模块

指标管理模块核心包括基础信息和技术信息管理，衍生信息包括关联指标、关联应用管理。基础信息对应的就是指标的业务信息，由业务人员填写，主要包括指标名称、业务分类、统计频率、精度、单位、指标类型、指标定义、计算逻辑、分析方法、影响因素、分析维度等信息；基础信息中还有一个比较重要的部分是监控配置，主要是配置指标的有效波动范围区间、同环比波动区间等，监控指标数据的正常运行。

技术信息构成比较复杂，包括数据类型、指标代码，但是核心部分是指标与模型的绑定关系，通过使用演进形成了当前系统两类绑定关系：绑定物理模型和构建虚拟模型。绑定物理模型是指标与模型管理中的物理模型字段绑定，并配置对应的计算公式，或还包含一些额外的高级配置，如二次计算、模型过滤条件等；创建虚拟模型是通过已有指标和其对应的物理模型，具体步骤首先配置已有指标的计算方式或指标维度的过滤，然后选择指标已绑定的物理模型，形成一个虚拟模型，虚拟模型的分析维度就是所选指标基础模型的公共维度。

衍生信息中的关联指标、关联应用管理，是为了方便观察指标被那些其他指标和数据应用使用，这是因为指标技术信息采用了严格权限控制，一旦被使用为了保证线上的运行安全是禁止变更的，只有解绑并审核通过后才可以编辑，所以这些衍生信息就是方便管理人员使用。指标技术信息如图7所示：

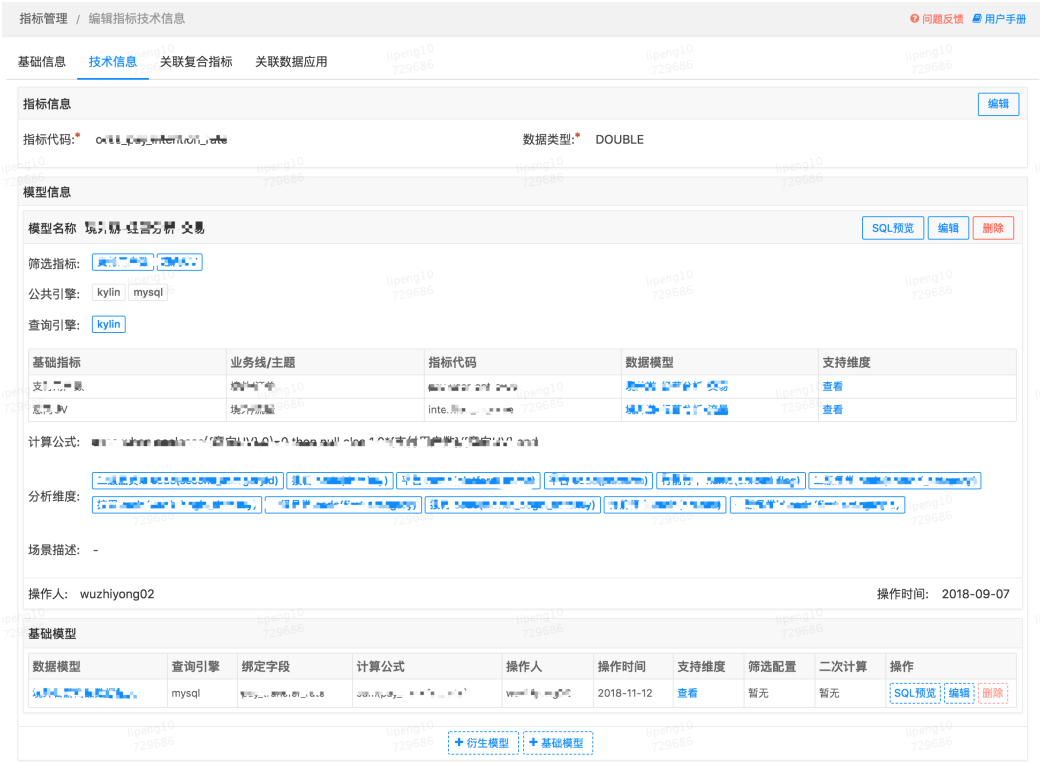


图7 起源数据治理平台指标技术信息

业务管理

业务管理按照功能划分为业务线管理、主题管理和工单管理三部分，在系统的实际建设中是拆分为业务主题管理、数据主题管理和工单管理三大模块实现的。相关模块的建设主要保证业务人员和数据人员业务主题建设，相关模块的权限控制，业务流程审核，对应资源的隔离以及业务资源加工申请和加工过程的记录追踪。具体实现和功能如下：

业务主题管理

实现业务业务线管理和业务主题管理，实现不同业务线的管理以及业务线下的业务主题管理。业务线的拆分还隐藏着其他模块的权限管控和资源隔离的功能，不同业务线的用户只能看到有权业务线的指标和维度；而且业务线的用户划分为普通用户和管理员，分别查看或编辑维度和指标的业务信息。而且业务线和业务主题中分别维护的商分负责人对指标进行二级审核，因为新创建的指标仅仅是普通指标，如果想要全网都能查看，则需要发起认证，由这些人员审核。

数据主题管理

数据主题管理实现数据业务线和数据主题管理，实现不同数据线的管理以及数据线下的数据主题管理。数据线的拆分也隐藏着对数据表、模型、指标、维度的资源隔离和权限管控的功能，不同数据线的用户只能查看有权数据线的资源；而且数据线的用户分为普通用户和管理员，对有权资源进行查看或编辑。数据线的接口人在工单模块中具有审核工单的权限功能。数据主题的负责人拥有审核模型和指标技术信息的权限功能。

工单模块管理

工单模块管理实现了指标维度和对应模型加工线上申请、审核、加工、审批的流程。整个模块也是围绕着这四个流程实现的，具体是业务人员发起指标和维度集合的加工申请，然后由数据线接口人审核工单的合理性并分配对应的数据开发工程师，数据开发工程师加工模型并与对应的维度指标绑定，然后在工单中提交由数据接口人审核是否合理，最终由工单发起人验收。

这个流程是一个标准的工单流程，每个节点的业务流程可能会反复，但是每次操作都进行记录，方便业务人员后期追踪。工单管理如下图8所示：

图8 起源数据治理平台工单管理

安全管理

安全管理是起源数据治理平台核心功能之一，分为平台操作权限管理和接口调用权限管理两大部分。其中平台操作权限管理是通过与公司将军令权限管理系统打通，并配合平台其他模块中权限控制代码，实现了权限管理、审批、审计三大功能模块；接口权限管理是通过平台内的数据应用管理和外部应用管理模块的映射关系，并在接口调用时鉴权实现，这部分会在下面的应用管理章节中介绍。

权限管理模块

权限管理模块是将平台的资源分划分为页面权限、业务线&数据线用户权限、数据应用权限来实现的。页面权限实现平台内页面访问控制。业务线&数据线用户权限是将用户分类为普通用户和管理员，普通用户只能查看业务线和数据线内资源，管理员可以操作业务线和数据线内的资源；并且通过业务线和数据线的独立管理实现资源隔离，业务线实现了所属维度和指标的隔离；数据线实现了所属数据表和模型的隔离，并且通过建立业务线和数据线的关联关系，也保证了指标和维度的技术信息操作隔离。数据应用中每个应用都是独立管理的，每个应用权限都拆分普通用户和管理员，普通用户可以访问查询应用，管理员可以操作应用。

审批模块

审批模块包含审批工作流、我的申请、我的审批构成。审批工作流是根据不同的应用场景实现不同层级的审批，例如：在指标管理中服务于个人的普通指标变更为服务于整个业务线的认证指标，就需要发起两级审批，由业务主题负责人和业务商分审核通过才可以；模型管理中新增或修改模型上线，都需要数据主题负责人审批；数据应用的变更，都需要下游所有依赖外部应用负责人审批才生效。我的申请和我的审批是

平台页面方便用户查看流程进度和操作审核。审批模块目标是保证发布信息的正确性、系统服务的稳定性。

审计模块

审计模块包括用户操作记录和记录查看追踪。用户操作记录是平台各模块调用接口记录用户每次操作前后的数据变更；记录查看追踪是检索查询页面，查看对应的变更。审计模块保证了用户操作追踪追责，也保证误操作的信息恢复。

应用管理

应用管理由数据应用、外部应用、数据地图三大模块组成，它们构成了对外服务的主体，记录了外部应用与平台内管理的指标、维度、模型和表的关联关系，也提供数据查询展示、应用层ETL生产的能力。而且数据开发人员从底层向上观察，可以追踪数据最终的所有流向；业务分析人员从顶层向下观察，可以看到构成服务的所有数据来源。

数据应用模块

数据应用模块是记录生成每个服务所需的指标、维度和数据模型的关系。每次服务中可以包含多个指标，这些指标可以来源于多个数据模型，不过不同的数据模型中需要包含公共维度，因为是通过这些公共维度将不同模型关联起来。

数据应用中构建的服务可以发布成查询服务、应用层ETL生产服务、对外API数据接口服务、通用报表配置服务，来满足业务的不同需求。数据应用管理如下图9所示：

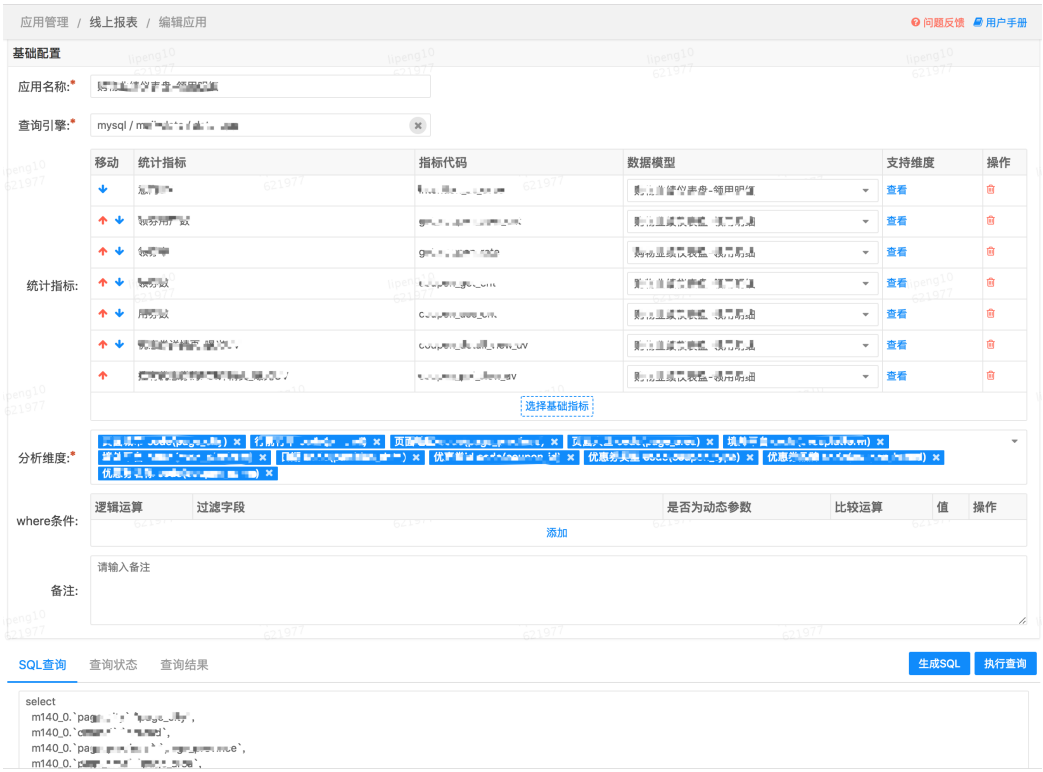


图9 起源数据治理平台数据应用

外部应用模块

外部应用模块管理外部应用和应用内的模块，以及这些模块订阅的对应数据应用，目标是实现API接口调用的权限管理和数据最终流向的记录。具体的实现上模块首先创建对应的外部应用，记录外部应用的名称、URL、APPKEY等信息，然后由对应应用的负责人创建模块，记录模块名称、URL、moduleKey等信息。这些信息完善后，由对应的数据应用赋权给对应的模块，建立起数据应用与外部应用的联系。最后在外部应用调用平台对外API接口时，进行权限管理。

数据地图

数据地图功能是追查数据的流向，可以从数据表、模型、指标、数据应用、外部应用任意节点查看上游数据来源和下游数据去向。起源数据治理平台核心功能也是组织这些节点间的关系，形成完整的服务，数据地图就是通过上面介绍模块记录的关系，追踪数据流向，方便数据开发人员和业务分析人员了解数据消费和数据来源。数据地图如下图10所示：

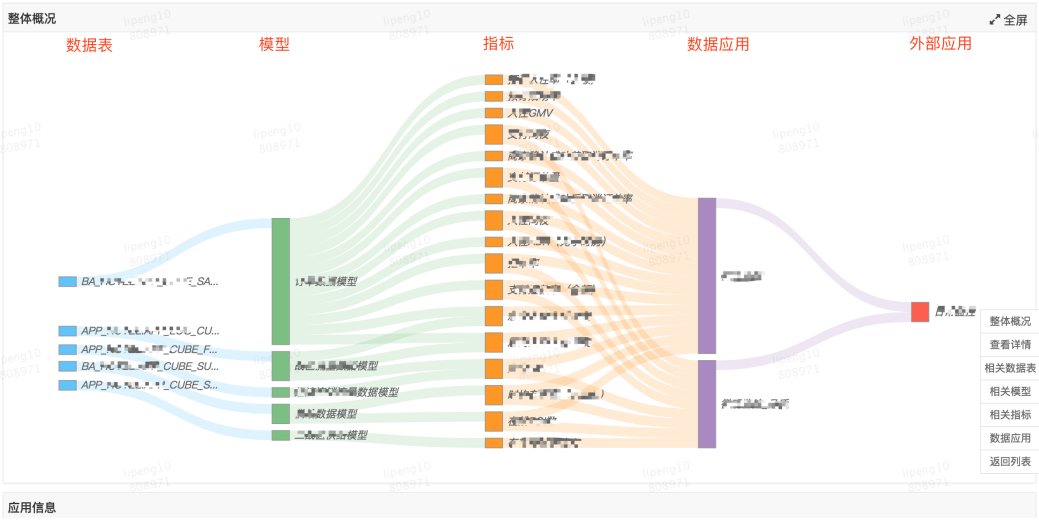


图10 起源数据治理平台数据地图

对外API

对外API接口是一套完整的对外信息提供接口，提供的功能分为元数据信息类的接口、数据类接口、监控统计类接口，分别满足外部平台和分析人员的对应需求。外部系统通过起源数据治理平台获取到的元数据和数据是经过认证并由平台自动校验后的，可以保证信息的一致性、正确性。

元数据信息接口

元数据信息接口提供的包括指标、维度业务元数据信息和数据表、模型、指标计算、维度维表相关的数据元数据信息，实现与上游系统信息共享，达到信息一致性的目标。

数据类接口

数据类接口提供指标维度数据查询服务，不单单满足常见的单条SQL查询，而且可以实现多次查询聚合运算（例如：同环比等）以及跨引擎查询，并通过并发处理，可以有效提升查询效率，满足更多的业务场景。接口具有监控功能，能够评估每次查询效率，提供查询指导或预警的能力。

监控统计类接口

监控统计类接口提供指标数据监控信息、指标维度使用统计、数据接口的调用效率统计等服务，帮助下游服务平台了解服务质量。

内部工作原理

起源数据治理平台内部工作原理就是实现指标、维度业务信息与数据模型计算关系的映射管理，并根据外部应用所需的指标、维度以及查询条件选择最优的模型动态的实现查询SQL或查询Query的拼接，然后通过分布式查询引擎实现数据的高效查询，具体过程如下图11所示：

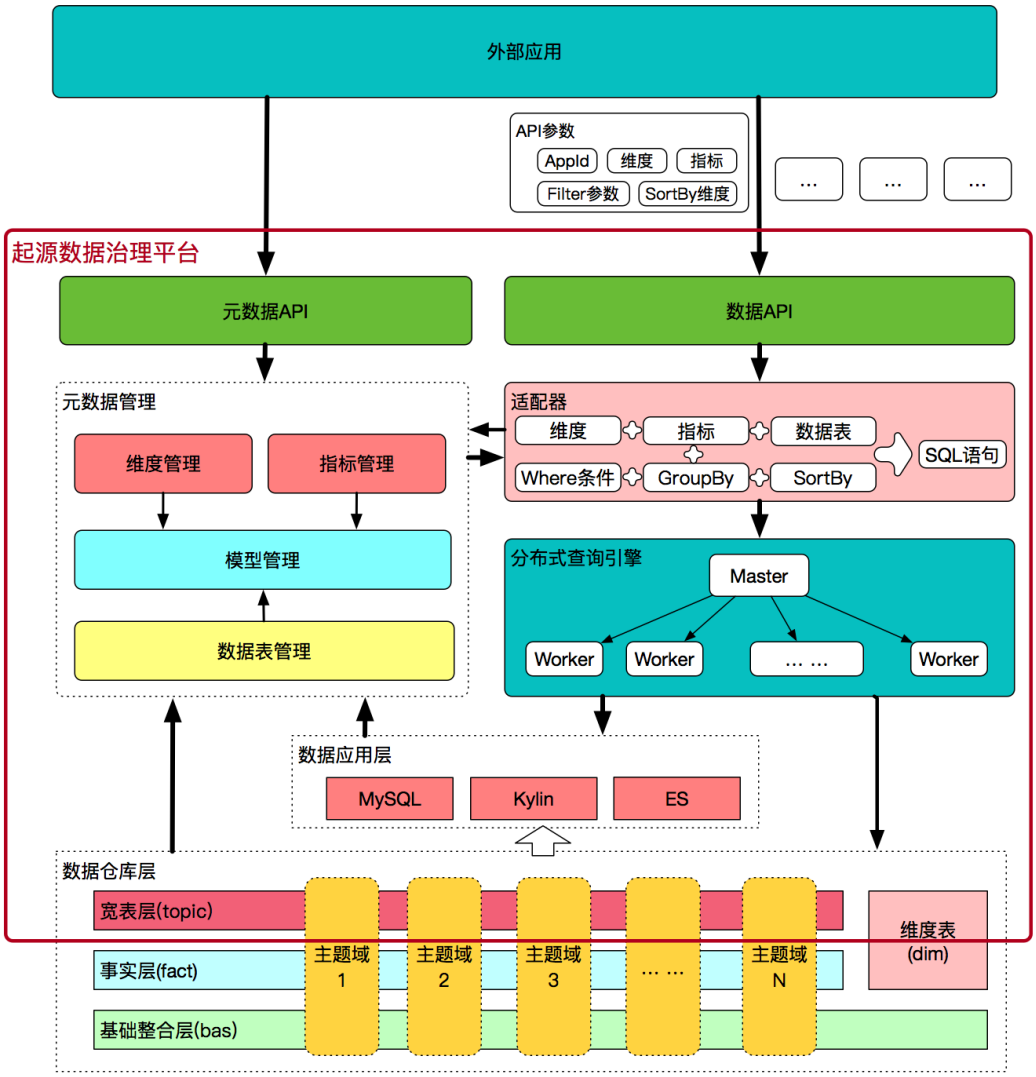


图11 起源数据治理平台内部工作原理

上图所示的分布式查询引擎，整合了大数据分析常见的各种存储，通过封装的接口提供服务。而且分布式是通过Akka Cluster自主实现，通过Cluster Singleton解决单点故障的问题，通过Redis实现了任务队列

的持久化，通过平衡子节点任务量实现任务的合理调度，通过查询状态监控自动实现查询降级和任务队列的拆解，并且也完善了整个调度的监控，可以实时查看任务和节点的运行情况。

管理流程

起源数据治理平台生产所需参与的角色包括：业务人员和数据开发人员（RD）。为了保证信息的正确性，平台内有着严格的管理流程，需要不同的角色在对应的节点进行维护管理，平台的管理流程如下图12所示：

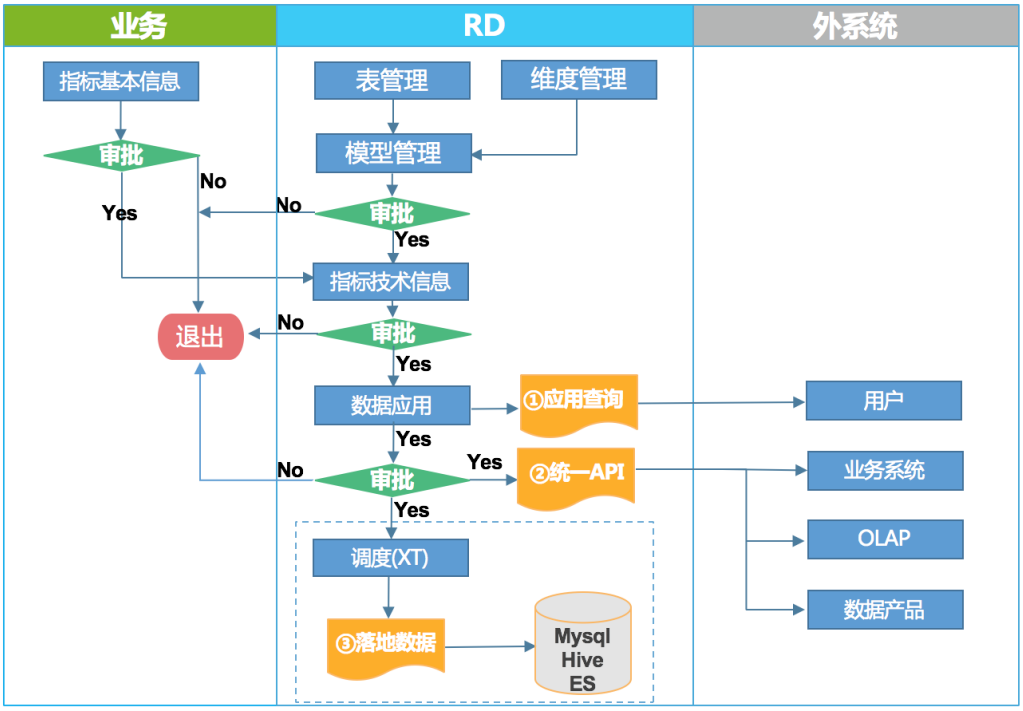


图12 起源数据治理平台管理流程

所上图所示，指标的业务信息需要业务人员首先进行维护，然后数据RD同学进行相应的数据表的建设，维护对应的数据表和模型的元数据信息，并完成指标与模型的绑定，最后由数据RD同学构建数据应用为用户、业务系统及数据产品等提供服务。

建设成果

经过长时间的探索开发，完成了起源数据治理平台的建设，成功的解决了上面提到的问题，并且已经完成了酒旅内部10+个数据平台（包括定制化产品和通用报表服务平台）的数据治理支持。起源数据治理平台还带来了一些额外的收获，总结归纳起来实现了3个目标，提供了4种能力，如下：

- **统一指标管理的目标。保证指标定义、计算口径、数据来源的一致性。**
- **统一维度管理的目标。保证维度定义、维度值的一致性。**
- **统一数据出口的目标。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。**
- 提供维度和指标数据统一监控及预警能力。
- 提供灵活可配的数据查询分析能力。
- 提标数据地图展示表、模型、指标、应用上下游关系及分布的能力。
- 提供血缘分析追查数据来源的能力。

如果换位到指标的角色，以辩证的角度分析，起源数据治理平台解决了一个终极哲学问题：**我是谁，我从哪里来，我到哪里去**。

未来展望

起源数据治理平台是天工体系（从数据管理、查询到展示的一个完整生态）的一部分，整个天工体系还包括如意通用报表系统、筋斗云数据查询系统。通过对天工体系的建设，直接目标是为业务提供一整套高效、高质量的数据服务平台；但是在天工体系的建设中，进行微服务治理，抽象形出一套统一标准，吸纳更多的业务参与建设，为业务提供开发降级，避免服务的重复建设，提升服务建设速度。如下图13所示：

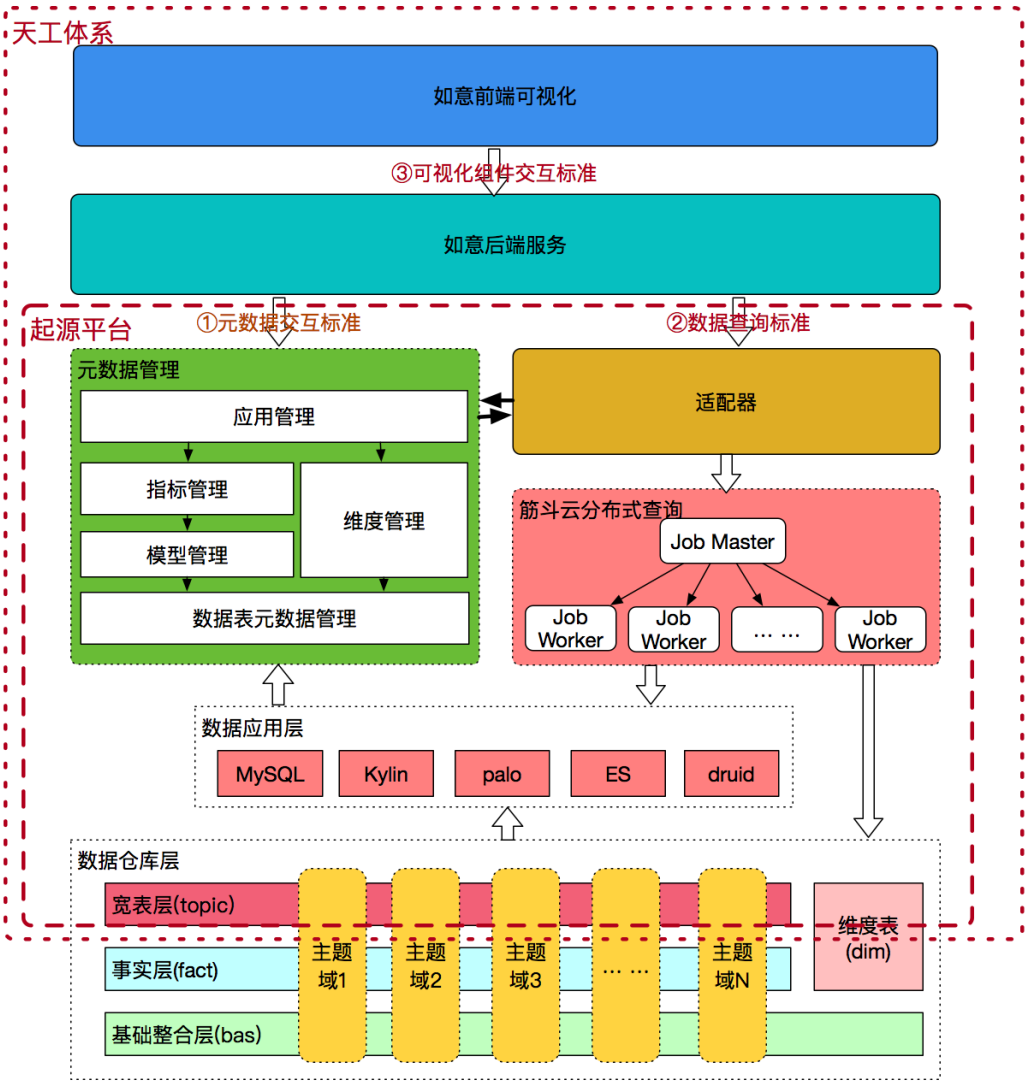


图13 天工体系架构图

如上图所示，天工体系开放三套交互标准，实现模块的可插拔和自由扩展，分别是：

- 元数据交互标准，实现元数据管理的可插拔。
- 数据查询标准，实现数据查询引擎的可插拔。
- 可视化组件数据交互标准，实现可视化组件的可插拔。

作者简介

- 夷山，美团点评技术专家，现任TechClub–Java俱乐部主席，2006年毕业于武汉大学，先后就职于IBM、用友、风行以及阿里巴巴。2014年加入美团，长期致力于BI工具、数据安全与数据质量工作等方向。
- 李鹏，美团点评技术专家，曾就职于搜狐畅游、网易，2018年加入美团点评，长期致力于数据治理、数据仓库建设、数据平台研发等工作方向。

招聘

最后插播一个招聘广告，有对数据产品工具开发感兴趣的可以发邮件给 fuyishan#meituan.com。我们是一群擅长大数据领域数据工具，数据治理，智能数据应用架构设计及产品研发的工程师。

高性能平台设计—美团旅行结算平台实践

作者: 子鑫

> 本文根据第23期美团技术沙龙演讲内容整理而成。

背景

美团酒旅有很多条业务线，例如酒店、门票、火车票等等，每种业务都有结算诉求，而结算处于整个交易的最后一环不可缺少，因此我们将结算平台化，来满足业务的结算诉求。结算平台通过业务需求以及我们对业务的理解，沉淀了各种能力并构建了丰富的能力地图。

我们将业务的发展归纳为几个阶段，例如业务孵化阶段，快速抢占市场扩大覆盖阶段，市场稳定后急需盈利阶段，国内业务稳定后的国际化阶段，业务发展的各个阶段都能在结算平台找到相应的能力支持。业务孵化阶段讲究的是低成本试错，我们将结算的核心流程，账单–>付款–>发票等模块平台化，新业务接入只需要5~10天。我们的预付款结算、分销结算、阶梯返佣结算、地推结算能力能快速的帮业务抢占市场实现盈利。

我们的汇率管理，多币种，多时区结算能力可以助力业务开展海外市场。当前结算平台支持美团酒旅4个事业部、17条业务线，涵盖境内、境外等业务的线上结算，后边我主要介绍下对账平台的实践即账单的实践。

对账平台的重要性

对账是平台化的第一环，它需要算清楚商家和美团的收益明细，后续的付款，发票等，都是基于对账进行开展的。同时它需要对接酒旅的各个订单中心，不同业务的订单具体实现和业务流程不同，不同的业务对账规则和时机也不相同，同时对账每天需要处理数百万条交易明细。

如何解决订单、对账层面的差异？每天处理数百万交易明细，如何高效准确的处理这些数据？17条业务线如何做隔离、定制、个性化扩容？其中面临了很多挑战，这也是为什么要讲对账的原因。

早期的系统实践及优化

早期系统实践

账单的生成逻辑主要分两部分：

1. 通过订单中心，供应链分别获取交易数据、商家基本信息、结算规则。
2. 基于交易数据、商家信息、结算规则生成对账单。

早期结算系统基于酒店业务实现，有很重的业务特性，模式是期末模式。比如账期是6月1日到6月30日，会在7月1日时执行以上两步操作，结算需要关注业务逻辑。这种模式存在很多问题，比如业务是灵活多变

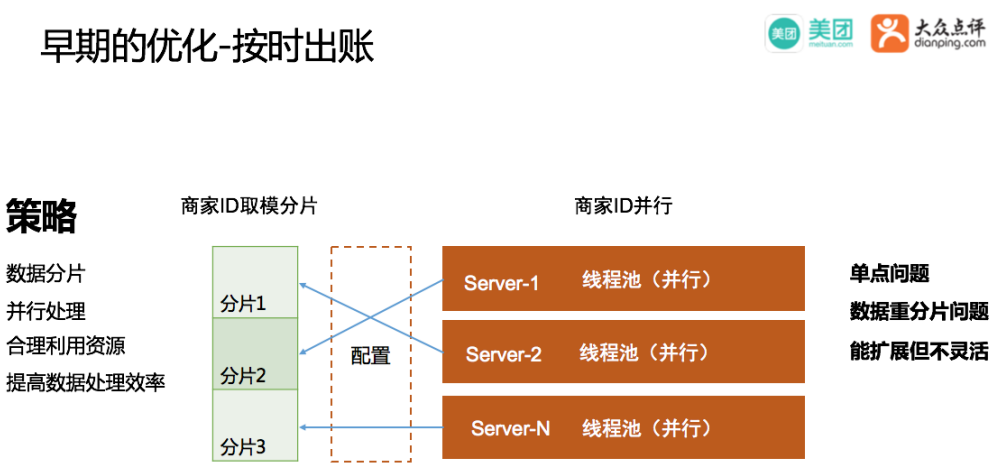
的，结算往往7月1日时才能感知到业务的变化，同时需要处理整个月的交易数据，这样就需要结算有很强的机动能力应对变化，很强的数据处理能力。

早期的系统架构和实现有一些问题，比如采用单线程，Pull的模式去获取数据，生成账单，虽然部署了很多台Server，但其实同一时刻只有一台Server在工作，有严重的资源浪费，同时数据处理效率较低。本应7月1日进行结账，因为业务变更，数据处理效率低，出账往往会拖到7月底，商家体验非常差。当时除了酒店业务之外，门票、火车票业务都处于孵化阶段也有结算需求，比较紧迫的任务有两个，一个是快速的解决酒店结账不及时的问题，另外一个是支持新业务的结算需求。所以当时的策略是，优化老系统让酒店业务平稳运行，搭建新的结算平台支持新业务的孵化。

按时结账优化

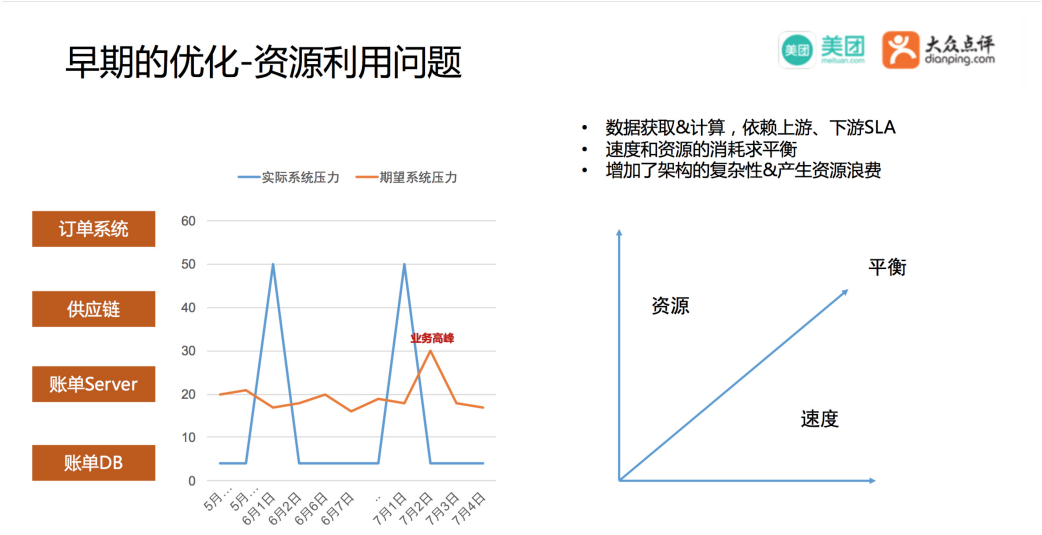
优化的策略有两点：

1. 和上游约定业务变更流程，需要提前告知结算，结算提前做系统升级。
2. 结算提高数据处理能力。



提高数据处理能力思路是：单线程改多线程，串行改并行，单机计算改分布式计算。具体做法是：商家ID维度用取模的方式分片，通过配置将不同的分片配置到不同的Server，同时每个Server上有单独的线程池，可以在商家ID的维度并行获取数据，处理账单。这样虽然能快了起来，但是它又引入了一些新的问题。

比如单点问题，如果Server2宕机了，配置不具备自动Rebalance的能力，就需要结合报警来人工处理，运维压力会变大。如果继续提升处理能力，就需要更多的数据分片，更多的Server资源，这时需要去重新处理分片和配置的逻辑，虽然它是可扩展的，但是并不灵活。



资源利用率波动也比较大，在7月1日处理整个月的数据，资源利用率（蓝线），在月初的时候最高，平时的时候基本上接近于0。比较健康的资源利用率，应该像黄线一样，虽然会有业务的峰值，但是整体上来看是趋于平稳的。如果是蓝线这样的资源利用，就会导致订单系统、结算系统、DB，都需要按照业务的峰值来部署，会产生资源浪费，最后我们在速度和资源的使用上求一个平衡，通过这些优化做到了及时结账。

但还遗留了如下问题：

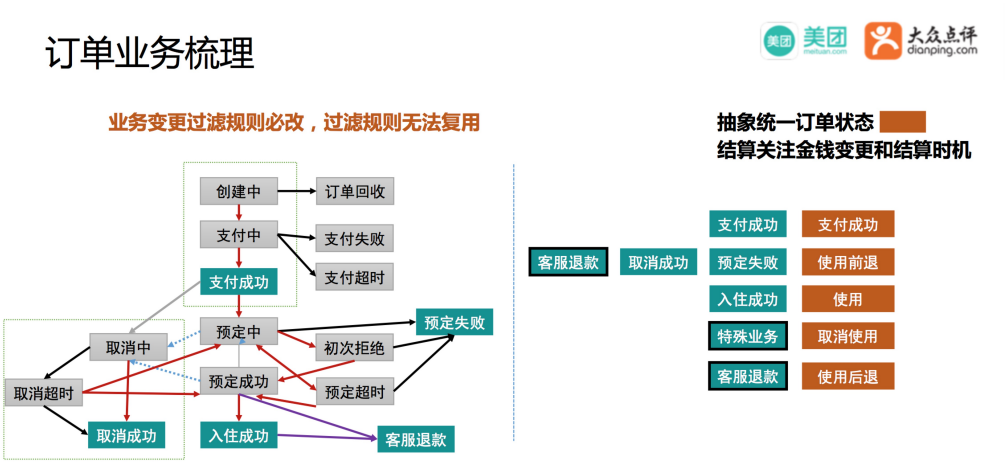
1. 单点问题。
2. 提升数据处理效率产生资源浪费。
3. 系统处理能力可扩展，但是不够灵活优雅。
4. 业务逻辑和结算逻辑严重耦合。

对账平台化及高性能实践

上边这些问题在对账平台化时都需要解决，尤其是逻辑耦合问题，如果不解决，后续对接的业务越多，步伐越沉重，最终会拖慢业务的发展。解决这个问题需要优化结算系统的业务架构，需要抽象一套订单模型，商家模型，计算规则模型，这些模型要足够通用，在兼容各个业务差异同时又要有一定的扩展和定制能力，模型的抽象是平台化的关键。

业务架构优化

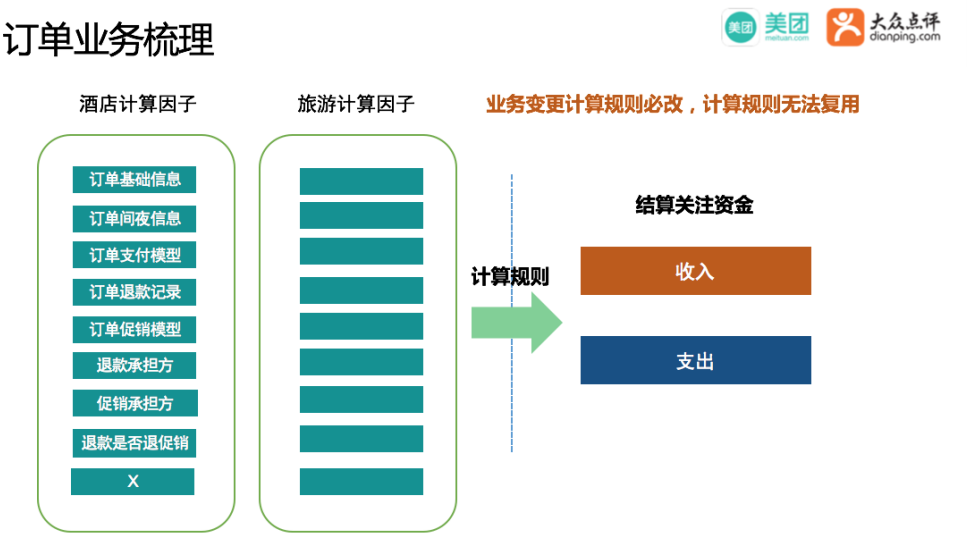
订单模型抽象



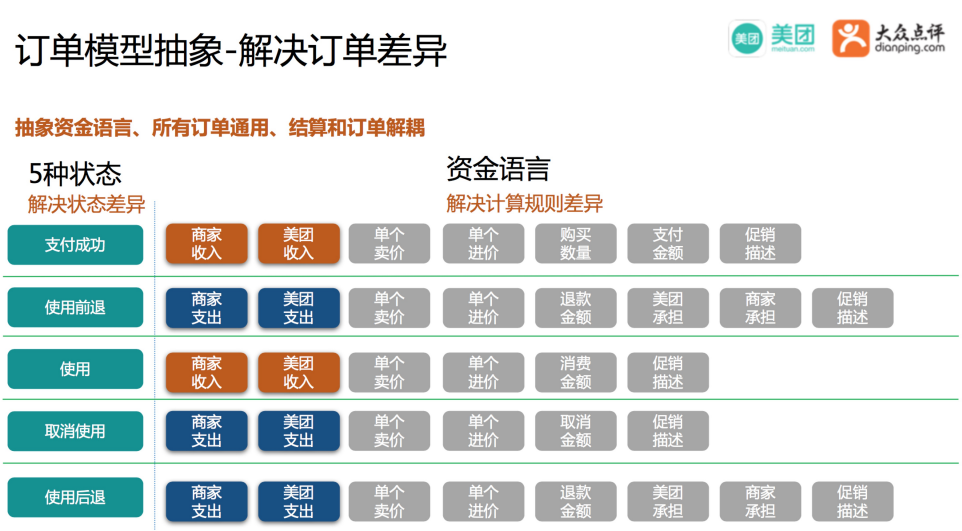
如上图订单状态机所示业务非常复杂，状态变更的轨迹订单如果不记录清楚，事后结算时很难回溯出来当时的交易情况。结算关注金钱变更和结算时机因此需要分析订单的状态机，过滤出结算要关注的状态，在上图中我用蓝色表示，业务变更会导致状态机的设计变更同时会影响过滤逻辑，这个是业务和结算耦合的关键点，不同的业务玩法不一样，状态机也不一样，过滤逻辑完全无法复用。

结算抽象了一组统一的订单状态来描述订单的整个生命周期：支付成功，使用前退、使用、取消使用、使用后退。它们所代表的含义如字面上一样，简单便于理解，每来一条业务线，将订单的原始状态和结算抽象的状态做映射，就能解决订单层面的差异。以酒店为例：预定周日晚上的酒店，周四行程变更无法入住，周五找客服退款，客服退款成功，这个对应使用前退，客服退款在一些特殊的场景也会对应到使用后退，最终解决了订单状态层面的差异。

计算规则抽象



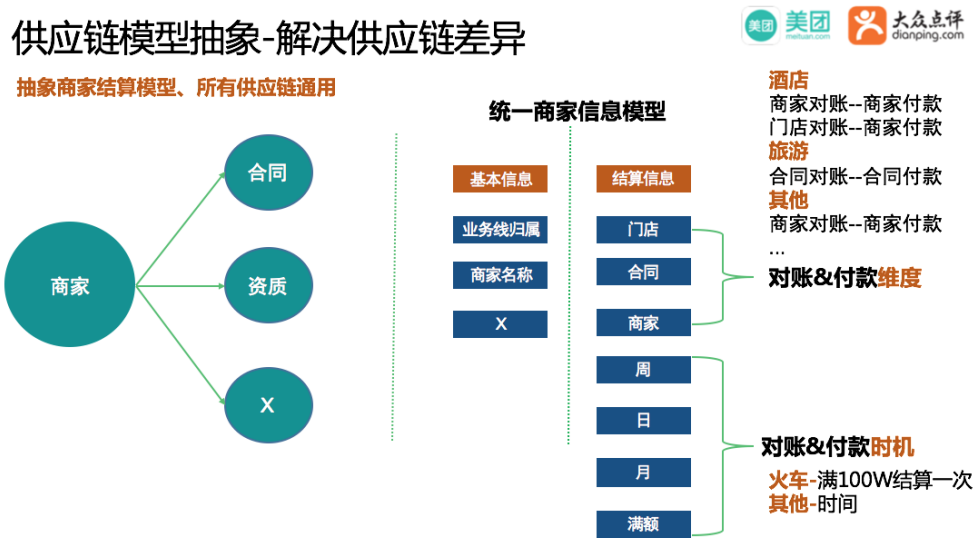
状态过滤出来后要计算这笔交易商家和美团分别的收入和支出是多少。如上图所示酒店要计算收入和支出需要订单的基础信息、间夜信息等因子，这些因子可能散落在各个数据表中，收集和计算非常复杂，业务变更会导致计算因子和计算逻辑变更，不同业务计算因子不相同，所以它很难复用。



结合前边抽象的订单状态，以及结算关注的收入和支出，我们抽象出来一套资金语言，它由5种状态+资金描述组成，所有订单通用，从而做到结算和订单的彻底解耦。例如“支付成功状态”、“使用状态”对应收入；“使用前退状态”、“取消使用状态”、“使用后退状态”对应支出。资金描述包含具体的收入、支出金额，以及收入、支出的具体描述。 商家收入= 单个进价 * 购买数量 - 商家承担促销 ； 美团收入=总卖价 - 总进价 - 美团承担促销 ， 最终商家的收入=总收入-总支出 ， 通过资金语言能推算出收入和支出的具体由来。

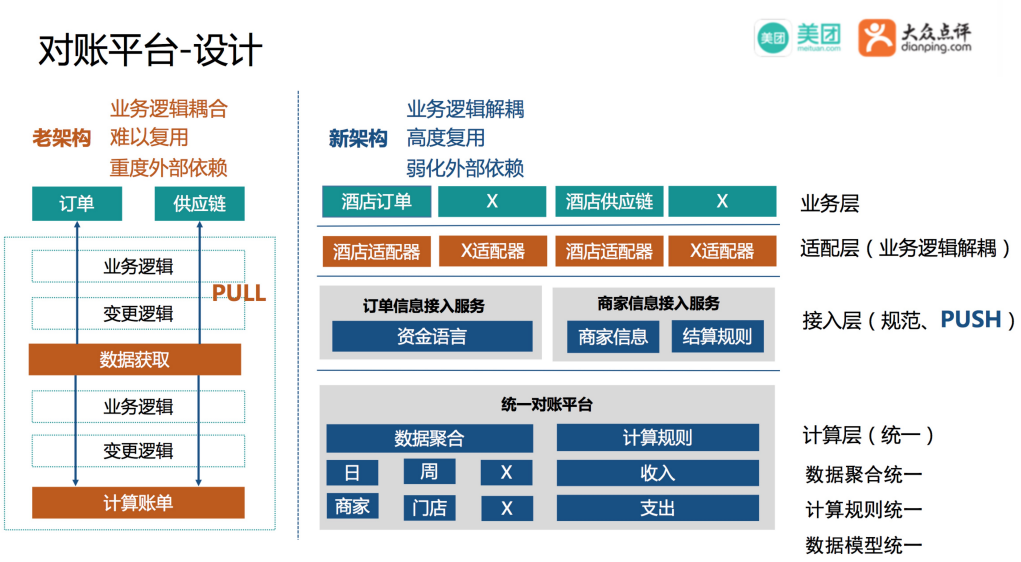
资金语言是结算和订单的标准协议，不管什么样的业务，数据结构和业务流程怎么设计，都按照标准协议来，只要协议不变，不管是订单变化还是结算变化都不会相互影响，订单和结算也就具备了独立演进的可能。

商家信息抽象



商家关联的信息有很多，例如合同、资质等等，但是结算关注的是这个商家的身份信息，我们将这类信息抽象为商家的基本信息，同时也需要关注商家的对账和付款规则，我们将这类信息抽象为结算信息，结算信息包含对账和付款的维度和时机。对账和付款不同业务维度和时机都不相同，我们将这些信息抽象出来，只要新业务维度和时机在这些规则之内，就能很好的复用，最终解决了供应链层面的差异。

对账平台设计



如上图所示。

老架构

数据获取采用PULL模式，如果上游故障会导致无法获取数据，从而影响账单计算，重度外部依赖。不管是数据获取还是计算账单都需要穿透两层业务逻辑，业务逻辑严重耦合难以复用。

新架构

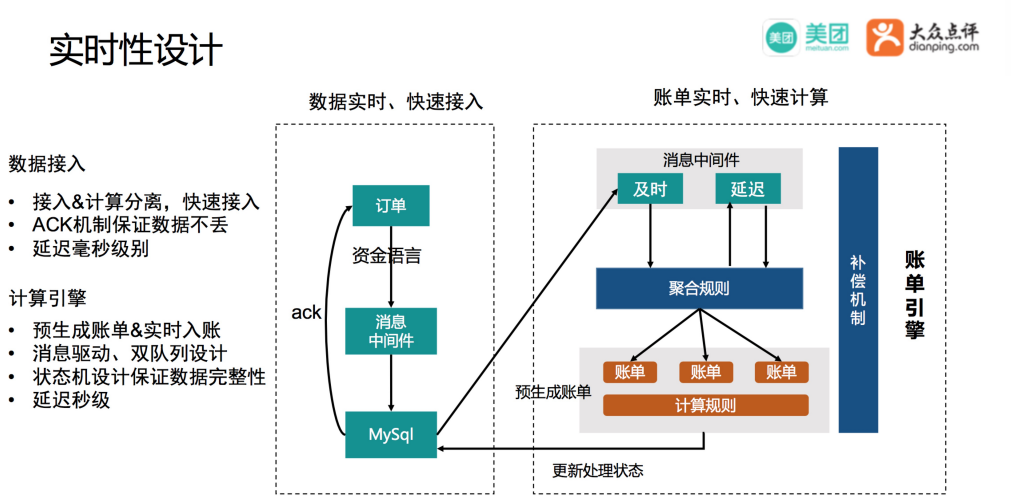
抽象了资金语言，商家信息，结算规则，标准化了接入规范，数据接入采用PUSH的模式，业务产生了交易，新签约了商家只要能将数据按时的PUSH过来就能按时结账，结算被动接收数据，轻度外部依赖。新架构设计了适配层，用来做业务数据和标准协议的适配，适配层逻辑结算和业务都可以做，但是考虑到业务侧的同学更了解业务，业务变更自己修改适配层不需要找结算排期更灵活，我们将适配层交给业务团队来做，每来一个新业务只需要一个小的适配块，就能快速接入。通过抽象的模型和标准协议，对账平台做到了数据聚合统一，计算规则统一，数据模型统一，从而达到了高度复用，结算和业务解耦。历史上酒店和旅游订单发生过多次重构，对结算基本无影响。人员也得到了高度复用，当前对账模块只有4名RD对接了17条业务线。

对账平台具体实践

对账平台实践主要关注如下四点：

1. 实时性设计，我们希望订单产生交易商家立刻能看到自己的收入，同时希望解决资源利用率的问题。
2. 高性能设计，每天数百万交易明细，每个账期需要处理数百万账单，怎么保证这些数据的生产和计算准确、高效?
3. 隔离和定制，结算对接了17条业务线，每个业务体量不同，业务逻辑也会有一定的差异，多个业务之间怎么隔离？怎么做到业务之间相互不影响?
4. 可扩展性设计，业务快速规模化以后，系统的处理能力可扩展，满足业务的发展预期。

实时性设计



数据接入

订单产生交易，将交易转为资金语言，通过消息中间件（Mafka）实时的推送给结算，结算只做必要的校验，完成后数据落到MySQL中，此时数据的状态是未处理，这一步设计了ACK机制保证数据不丢。落库后会给账单引擎发一个消息，说有一条数据要处理了。数据的接入和账单引擎的计算做了分离，数据的接入非常快，基本延迟在毫秒级别。

账单引擎

账单引擎会在6月1日生成6月1日到6月30日的空账单，数据经过聚合规则，进入到相应的账单，同时触发账单计算规则，完成账单的实时计算。一些特殊的情况例如数据产生在6月1日，结算日期是6月2日，会将这个消息转发到Mafka的延迟队列，在6月2日重新消费这条消息，数据处理成功后，数据的状态变更为已处理。

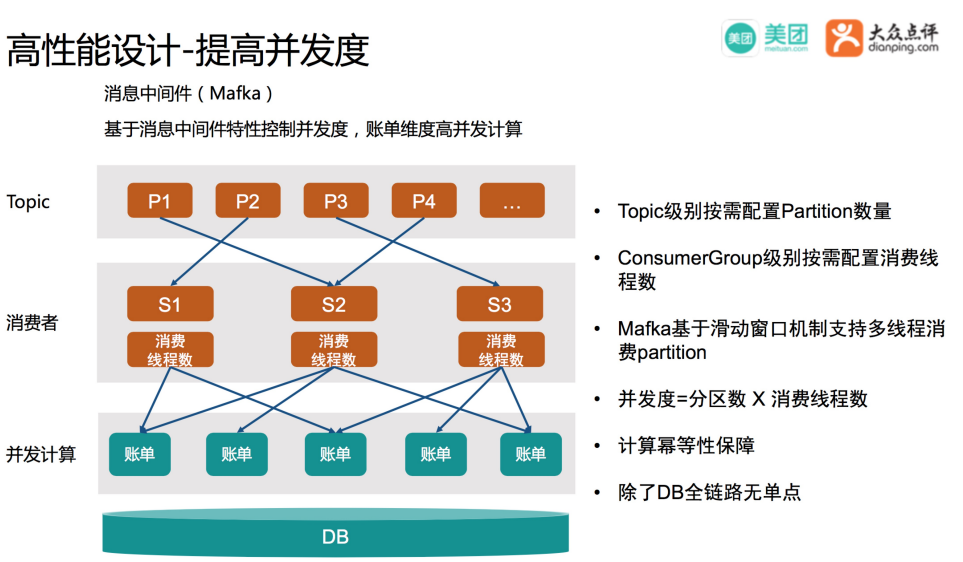
账单引擎的补偿和监控机制是通过对数据的状态控制来实现的，例如6月1日到30日，商家A一共产生了1万条交易明细，已处理的数据只有8千条，那剩下的2千条要么就是在途，要么就是消息丢了，或者系统Bug，会有一定的监控和补偿策略来保证数据的完整性，账单引擎处理具体的交易明细延迟在秒级。

高性能设计

提高账单的处理效率需要从两个维度出发：

1. 每天要处理对账单数量大概在百万级别，一个一个处理是非常慢的，所以要尽可能在对账单维度提高并发度。
2. 每个对账单要接收的流水数量也不相同，可能单对账单同一时刻的并发度会很高，所以要尽可能提高单对账单的并行度。

提高高并发设计



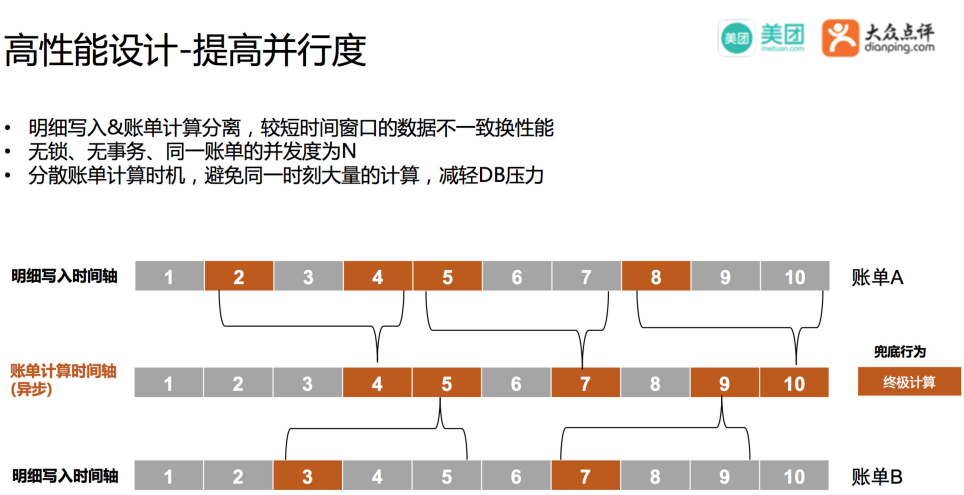
我们通过消息中间件Mafka来提高并发度，在Topic的维度拆分多个Partition，不同的ConsumerGroup配置不同的消费线程数，Mafka基于滑动窗口机制实现了多线程消费同一个Partition的数据，所以可以做到 并发度= Partition数量 * 消费线程数 ，在数据处理上做了幂等性保障，能确保数据的正确处理。从架构上来看除了DB之外全链路无单点，比如某个Server挂了以后，通过Mafka的Rebalance机制，将Partition自动分配给别的Server，消除了Server层面的单点问题。

提高并行度设计

单个对账单包含了多条交易明细，账单的总金额是多条交易明细金额的累加，并发度上来以后，会产生线程安全性问题，怎么保证账单总金额的数据准确性？常规实现：

```
//事务保证原子性
try {
    Lock lock = distributedLockManager
            .getReentrantLock(statementId);//账单维度锁
    lock.lock(); //获取锁，存在阻塞
    insert(detail);//写入明细
    compute(statementId);//计算账单
} finally {
    lock.unlock();// 释放锁
}
```

如上边的伪代码所示，要有一个事务保证明细写入和账单金额计算的原子性，其次是获取分布式锁，写入明细，计算账单，因为锁的缘故单账单的并发度是N，并行度是1，并行度低的结果会导致消息出列变慢，单个账单的处理效率变低，有了事务也会有一定的性能开销。怎么提高并行度，怎么减少事务的粒度成为单账单维度高性能计算的关键问题。



我们对商家的操作习惯进行了分析，商家更关注账单的总金额，不太关注交易明细，只要能保证账单总金额的计算相对实时，最终是正确的就好了，所以我们将账单明细的写入和账单的计算做了分离，不保证原子性，这样就省去了事务的性能开销，账单明细只管写入，不用关注账单的计算，全程不显式加锁，提高了明细的写入效率，Mafka的消息出列也变的更快。

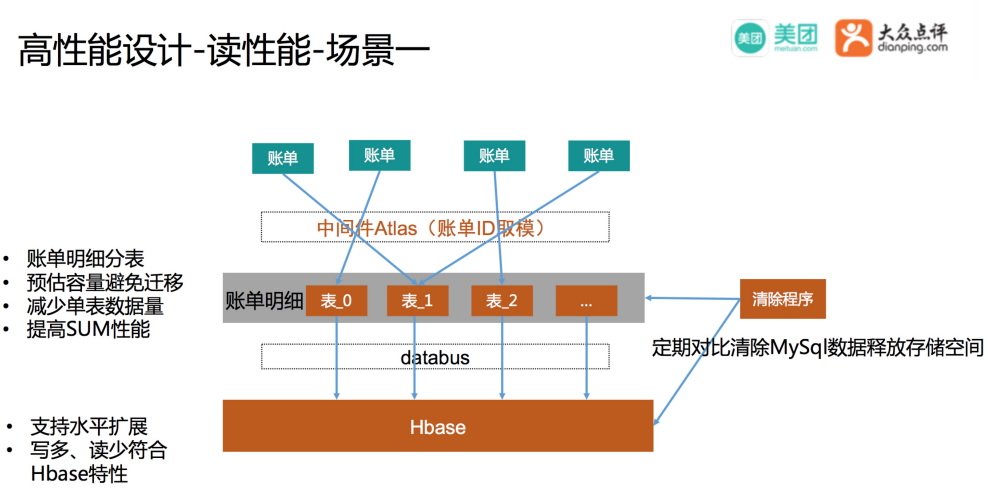
另外一方面，我们为每个账单抽象了写入时间轴，通过写入时间推算账单的计算时间。假设我们配置账单的延迟计算窗口为3秒，账单A在第2秒和第4秒的写入都会在第4秒合并成一次计算，账单B在第3秒和第5秒的写入都会在第5秒合并成一次计算，账单没有明细写入不触发计算。这样在商家的视角账单总金额会有最多3秒钟的延迟，从体验上来看也是完全可以接受的。

每个账单的写入时机不同，每个账单的计算时机也不同，所以可以将多个账单的计算时机随机相对均匀的分散开，减轻DB的压力，每个账单在最终结账时都会有一次终极的兜底计算，防止计算异常。最终做到了无事务、无锁，单账单并行度也从1变成了N，提高了单账单的处理效率。

提高读性能

场景一

以前边所提账单的计算为例，核心语句如下： `select sum(金额) from 账单明细表 where 账单ID = X` ，当数据规模在亿级怎么保证效率？我们除了在账单ID维度增加索引以外，还通过分表，数据备份的方式尽可能保证单表里数据量不至于太大，来提高处理效率。

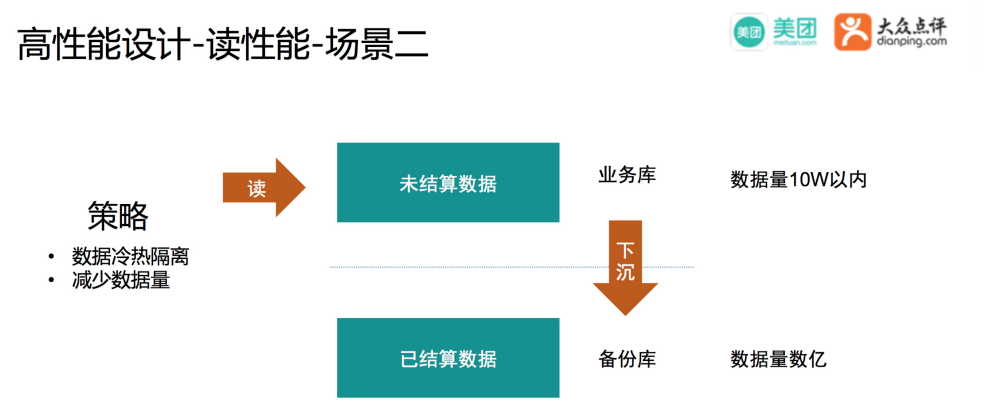


通过中间件atlas，账单明细按照账单ID取模的方式进行分表，保证同一个账单明细在一张表中，虽然会有一定的数据分布不均匀的问题，但是在很大程度上也能避免同一个表过于庞大，从而提高的SUM操作的性能。

账单明细有一个特性是不可变更，我们的业务场景是写多\读少，HBase也支持水平扩展，很适合做数据备份，所以我们通过Databus将数据同步到HBase中一份，还设计了一个程序定期的去对比MySQL和Hase的中的数据，账单结账后就不需要SUM操作了，经过一定的周期以后可以将MySQL中的明细做清除，释放MySQL的存储空间，只要一开始账单明细容量预留的足够大，通过这样的策略基本上能避免后续业务规模化后扩容产生的数据迁移问题，但是会有另外一个问题是需要定期处理数据库因为删除操作而产生的表空间空洞问题。

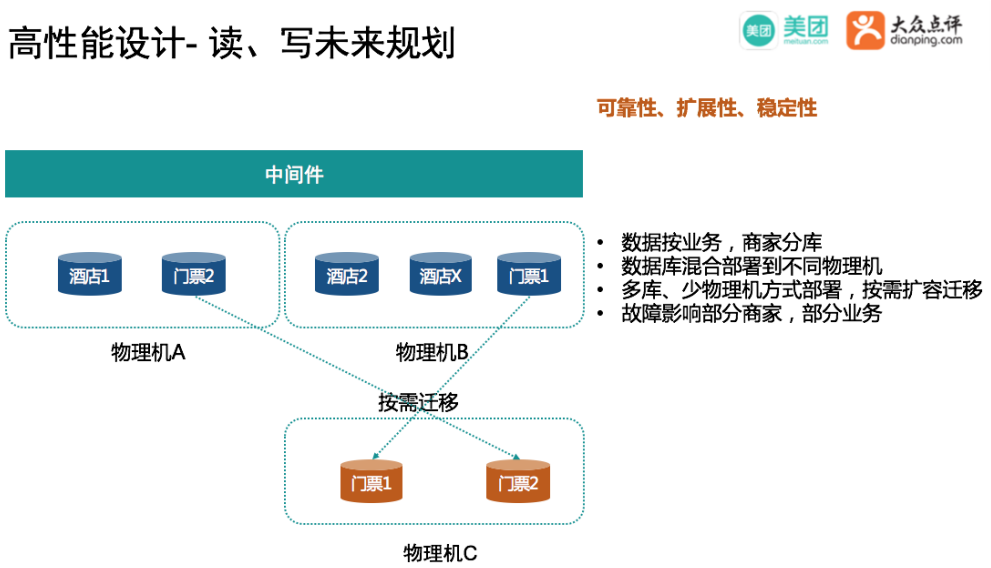
场景二

数据完整性校验，保证一个商家的一个账单数据是完整的，核心语句如下： `select count(1) from 流水表 where 业务线=酒店 and 商家ID=1 and 流水时间 > X and 流水时间 < Y and 数据状态 = 未处理` 。数据量在亿级如何保证效率?



我们将交易明细做了冷热隔离，将已结算的数据备份到备库中，未处理的数据整体量级在10万条以内，主要是不到结算时机或者在途的一些数据，在配合一些索引，整体的处理效率是非常高的。

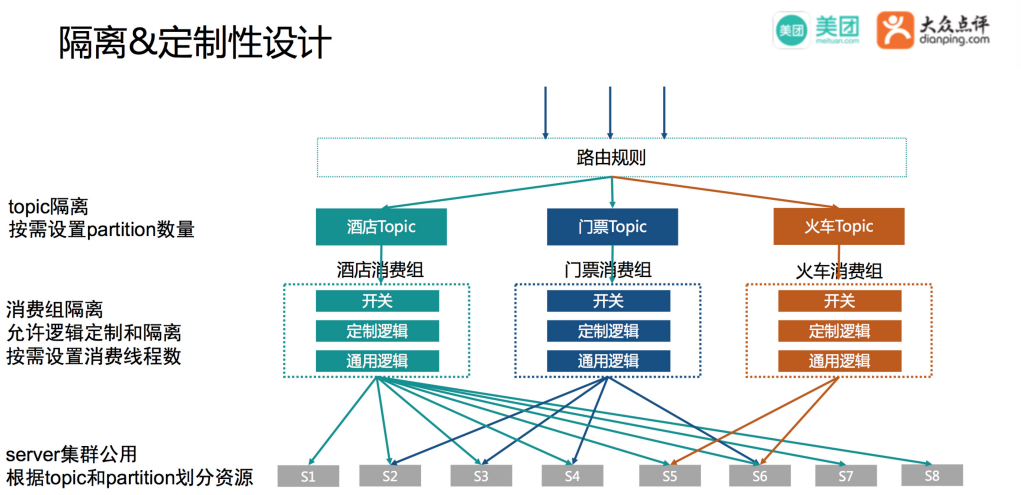
DB读写未来的规划



未来会根据业务和商家分库，前期采用多库，少物理机的模式部署，避免资源浪费。当物理机容量不足时按需迁移到新的物理机，整个过程只会较短时间的block写，对读无影响，迁移较为方便，目的是为了消除DB单点问题，故障只影响部分商家，部分业务。另一方面通过分库，多物理的机的形式也能提高DB层面的读、写能力。

隔离和定制

结算平台对接了17条业务线，不同业务体量不同，所需要的系统容量也不同，怎么定制化扩容？业务也会有一些特殊流程和逻辑怎么做逻辑定制？怎么做到A业务线故障不影响B业务线？



如上图，我们根据一定的路由规则，将数据灌到不同的Topic中，酒店体量比较大可以针对酒店Topic配置较多的Partition，火车票体量没有酒店大可以配置相对少的Partition，针对不同的消费组配置不同的消

费线程数，从而做到给不同业务分配不同的系统容量。账单引擎由通用逻辑+定制逻辑实现，定制逻辑以消费组维度做隔离，每个消费组配置单独的开关，当业务产生异常需要暂停结算时，关掉开关，对其他业务无影响。Server集群公用，通过Mafka的机制根据Topic和Partition自动划分资源，最终做到个性化扩容，多业务隔离相互不影响。

整体扩展性设计

主要分三层：

1. 第一层是消息中间件，可以通过拆Topic、Partition，增加消费线程数，来提高整体的并发度。
2. 第二层是通过程序优化，增加机器，从而提高并行度和整体的处理能力。优化程序如前边提到的减少锁和事务，分散数据压力等手段。
3. 第三层DB这块可以通过分库、分表、冷热隔离，提升整体的读写能力。

整体的扩展是需要多层进行配合的，比如Mafka的并发度特别高，Server数也特别多，但是只有一个DB，DB可能就是一个瓶颈，那就需要在DB的层面做一些扩展和优化。

最后呈现出的效果是，数据接入基本上延迟是在毫秒级别，对账单是实时对账单，产生交易后商家立刻就能在账单上看到收入明细，对商家来说体验非常好，对接了17条业务线，从不交叉影响。

作者简介

- 子鑫，2015年7月加入美团，目前是酒旅结算平台技术负责人。之前在京东和去哪儿从事订单交易相关的一些工作。

招聘信息

结算平台目前在招Java后台研发工程师，感兴趣的可以投递个人简历到邮箱：zhangzixin03#meituan.com，欢迎您的加入。

实时数据产品实践——美团大交通战场沙盘

作者: 娣娣 晓磊

背景

大数据时代，数据的重要性不言而喻，尤其对于互联网公司，随着业务的快速变化，商业模式的不断创新、用户体验个性化、实时化需求日益突出，海量数据实时处理在商业方面的需求越来越大。如何通过数据快速分析出用户的行为，以便做出准确的决策，越来越体现一个公司的价值。现阶段对于实时数据的建设比较单一，主要存在以下问题：

1. 实时仓库建设不足，维度及指标不够丰富，无法快速满足不同业务需求。
2. 实时数据和离线数据对比不灵活，无法自动化新增对比基期数据，且对比数据无法预先生产。
3. 数据监控不及时，一旦数据出现问题而无法及时监控到，就会影响业务分析决策。

因此，本文将基于美团大交通实时数据产品，从面临的挑战、总体解决方案、数据设计架构、后台设计架构等几个方面，详细介绍实时数据系统的整体建设思路。

挑战

实时流数据来源系统较多，处理非常复杂，并且不同业务场景对实时数据的要求不同，因此在建设过程主要有以下挑战：

1. 如何在保证数据准确性的前提下实现多实时流关联；实时流出现延迟、乱序、重复时如何解决。 流式计算中通常需要将多个实时流按某些主键进行关联得到特定的实时数据，但不同于离线数据表关联，实时流的到达是一个增量的过程，无法获取实时流的全量数据，并且实时流的达到次序无法确定，因此在进行关联时需要考虑存储一些中间状态及下发策略问题。
2. 实时流可复用性，实时流的处理不能只为解决一个问题，而是一类甚至几类问题，需要从业务角度对数据进行抽象，分层建设，以快速满足不同场景下对数据的要求。
3. 中台服务如何保证查询性能、数据预警及数据安全。 实时数据指标维度较为丰富，多维度聚合查询场景对服务层的性能要求较高，需要服务层能够支持较快的计算能力和响应能力；同时数据出现问题后，需要做好及时监控并快速修复。
4. 如何保证产品应用需求个性化。 实时数据与离线数据对比不灵活，需要提供可配置方案，并能够及时生产离线数据。

解决思路

我们在充分梳理业务需求的基础上，重新对实时流进行了建设，将实时数据分层建模，并对外提供统一的接口，保证数据同源同口径；同时，在数据服务层，增加可配置信息模块解决了配置信息不能自动化的问题，在数据处理策略上做了多线程处理、预计算、数据降级等优化，在数据安全方面增加数据审计功能，更好地提升了产品的用户体验。

总体方案

产品整体建设方案基于美团技术平台，总共分为源数据层、存储层、服务层及WEB层，整体架构如下所示：

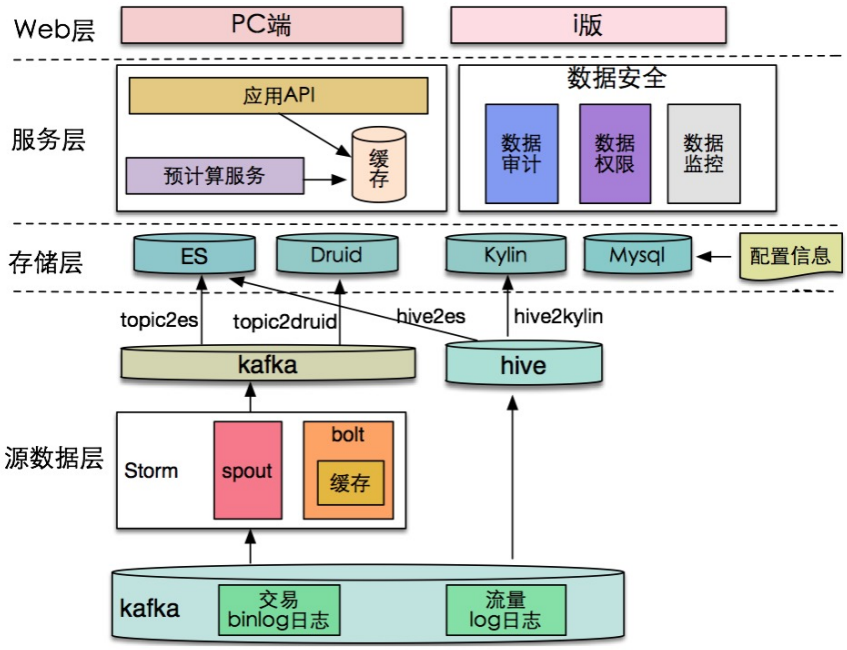


图1 整体架构图

- 源数据层：主要提供三部分数据，实时数据、离线数据、配置信息、维度信息。
- 存储层：源数据清洗后放入相应的存储引擎中，为服务层提供数据服务。
- 服务层：提供三部分功能，数据API服务、预计算服务、权限服务、数据审计服务。
- Web层：使用Echarts可视化数据。

数据层

数据架构

依托于美团提供的公共资源平台，数据架构按功能分为数据采集、数据处理、数据存储、数据服务四层，如下所示：

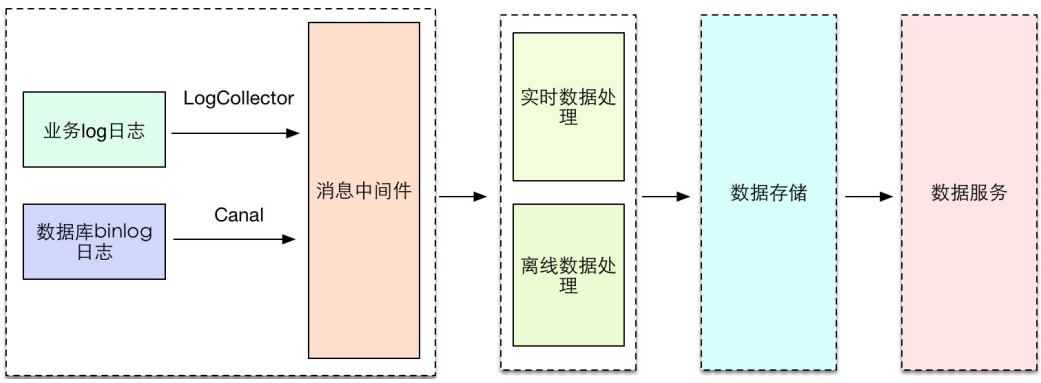


图2 数据架构图

数据采集

数据来源主要有两种：业务上报的Log日志及数据库Binlog日志。这些日志通过美团日志中心进行采集后存储在消息中间件Kafka中，并按照不同的分类存储在不同的Topic中，供下游订阅。

数据处理

数据处理顾名思义，就是对采集的实时流进行逻辑处理，按业务需求输出对应的实时数据，因此这一步骤是流式计算的关键，分两步进行：数据加工、数据推送。

数据加工：数据加工通常需要在流式计算系统中进行，目前流行的流式处理系统主要有Storm、Spark Streaming系统及Flink系统，这些系统都能在不同的应用场景下发挥很好处理能力，并各有优缺点，如下图所示：

|计算框架|吞吐量|延迟|传输保障|处理模式|成熟度| |:—:|:—:|:—:|:—:|:—:|:—:|:—:| |Storm |低|毫秒级|At least once|单条处理|成熟| |Spark Streaming |高|秒级|Exactly once|微批处理|成熟| |Flink |高|毫秒级|Exactly once|单条处理/微批处理|新兴|

最终我们选择Storm作为实时数据处理框架，并借助公司提供的通用组件来简化拓扑开发流程和重复代码编码。例如，组件MTSimpleLogBolt的主要功能是将Kafka中读取的数据（Log or Binlog）解析成Java实体对象；组件StormConfHelper的功能是获取Storm作业应用配置信息。

数据推送：将处理好的数据推送到存储引擎中。

数据存储

数据加工完成后会被存储到实时存储引擎中，以提供给下游使用。目前常用的存储引擎主要有MySQL、Druid、Elasticsearch、Redis、Tair比较如下：

存储引擎	优点	缺点
MySQL	使用简单，支持数据量小	数据量大，对MySQL的压力大,查询性能慢
Druid	数据预计算	不支持精确查询
Elasticsearch	查询效率快，支持常用聚合操作；可以做到精确去重	查询条件受限
Redis	内存存储KV，查询效率高	写入资源有限，不支持大数据量写入
Tair	持久化和非持久化两种缓存，查询效率高	单节点性能比Redis较弱
Kylin	多维查询预计算	不支持实时

综上比较，由于实时数据量较大，且数据精度要求较高，因此我们最终选择交易存储使用ES，流量存储使用Druid，维度存储使用Tair，中间数据存储使用Redis；而离线数据，我们采用Hive和Kylin存储。

数据服务

将存储引擎数据统一对外提供查询服务，支持不同业务应用场景。

具体实现

实时流处理流程

整个数据层架构上主要分为实时数据和离线数据两部分：实时数据分为交易的Binlog日志和流量的Log日志，经过Strom框架处理后写入Kafka，再经过DataLinkStreaming分别写入ES和Druid；离线数据通过Hive处理写入Kylin。

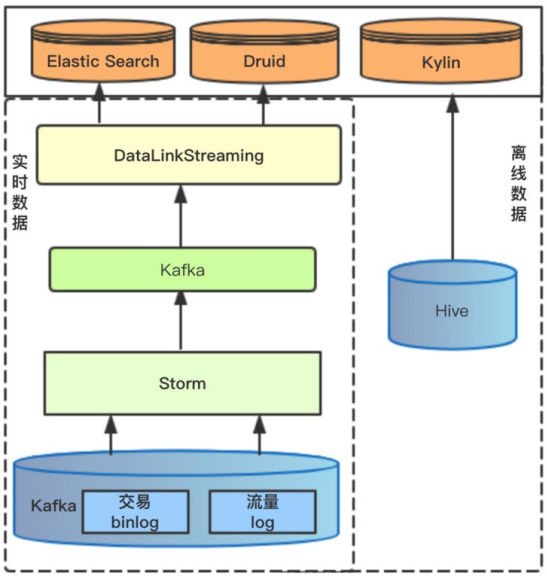


图3 产品数据架构

下图所示为一条消息的处理流程：

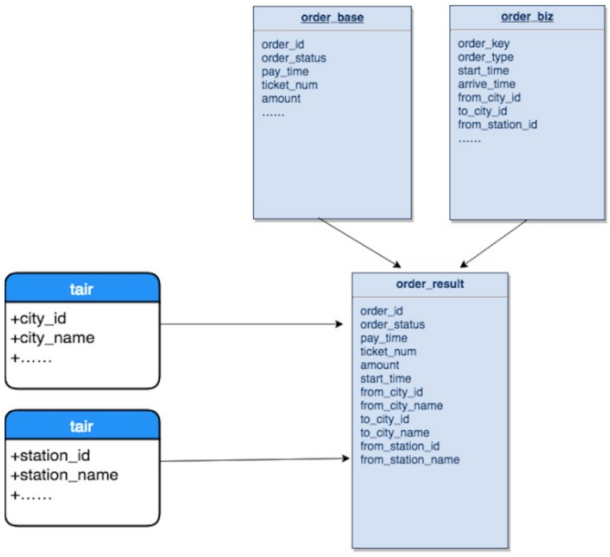


图4 数据关系

两个Topic分别是order_base（主要存放订单基本信息：订单id、订单状态、支付时间、票量、金额等）；order_biz（主要存放订单的扩展信息：订单id、订单类型、出发时间、到达时间、出发城市、到

达城市）。我们最终要拿到一条包括上述全部内容的一条记录。

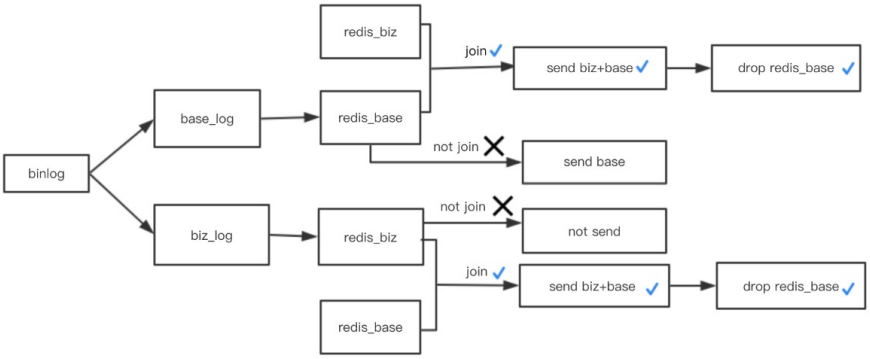


图5 数据处理流程

具体例子：Bolt在处理一条记录时，首先判断这条记录是base还是biz，如果是base则写入缓存中base的Category中，如果是biz则写入biz的Category中。以order_id为Key，如果是base则去和biz关联，如果biz存在则代表能够关联上，这时发送关联后的完整数据，同时删除该主键（order_key）记录；如果biz中不存在，则说明没关联上，这时可能biz的数据延迟或者是丢失，为了保证主数据的准确性，这时我们只发送base的数据，缓存中的数据保留不被删除。如果这条消息是biz，则首先会更新缓存中该主键的biz记录，然后去和base关联，关联上则发送同时删除base中数据，否则不发送。此时我们会根据ES的Update特性去更新之前的数据。从现实效果来看保证了99.2%的数据完整性，符合预期。

数据写入

在Topic2es的数据推送中，通过DataLinkString工具（底层Spark Streaming）实现了Kafka2es的微批次同步，一方面通过多并发batch写入ES获得了良好的吞吐，另一方面提供了5秒的实时写入效率，保证了ES查询的实时可见。同时我们也维护了Kafka的Offset，可以提供At lease once的同步服务，并结合ES的主键，可以做到Exactly once，有效解决了数据重复问题。

ES索引设计及优化

在数据写入ES过程中，由于数据量大，索引时间区间长，在建设索引时需要考虑合理设计保证查询效率，因此主要有以下三点优化：

- **写入优化** 在通过DataLinkString写入ES时，在集群可接受的范围内，数据Shuffle后再分组，增加Client并发数，提升写入效率。
- **数据结构化** 根据需要设计了索引的模版，使用了最小的足够用的数据类型。
- **按天建索引** 通过模版按天建索引，避免影响磁盘IO效率，同时通过别名兼容搜索一致性。
- **设置合理的分片和副本数** 如果分片数过少或过多都会导致检索比较慢。分片数过多会导致检索时打开比较多的文件，另外也会影响多台服务器之间通讯。而分片数过少为导至单个分片索引过大，所以检索速度慢。在确定分片数之前需要进行单服务单索引单分片的测试。 我们根据 索引分片数=数据总量/单分片数 设置了合理的分片数。

实时数据仓库模型

整个实时数据开发遵循大交通实时数仓的分层设计，在此也做一下简单介绍，实时数仓架构如下：

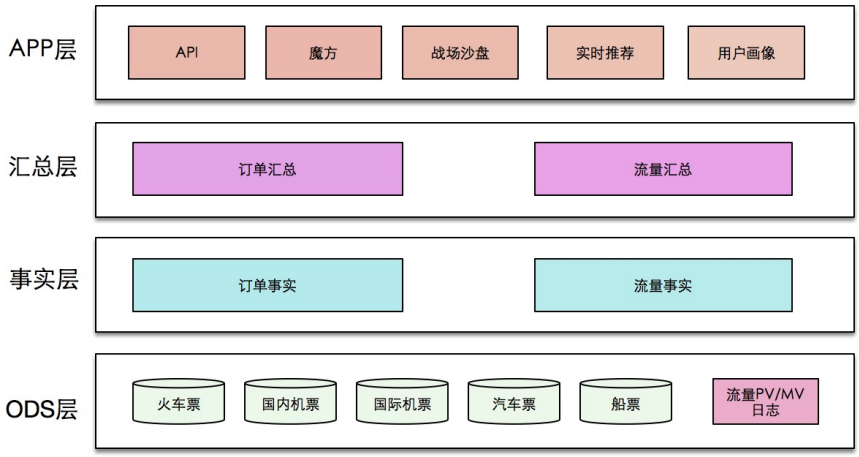


图6 实时数仓架构

- ODS层：包含美团页面流量日志、模块事件日志以及用户操作的Binlog信息日志，是直接从业务系统采集过来的原始数据。
- 事实明细层：根据主题和业务过程，生成订单事实和流量事实。
- 汇总层：对明细层的数据扩展业务常用的维度信息，形成主题宽表。
- App层：针对不同应用在汇总层基础上加工扩展的聚合数据，如火车票在抢票业务下的交易数据汇总信息。

规范建模后，业务需求来临时，只需要在App层建模即可，底层数据统一维护。

中台服务层

后台服务主要实现 登陆验证和权限验证(UPM)、指标逻辑计算和API、预计算服务、数据质量监控、数据审计功能。由于数据量大且实时性要求较高，在实现过程遇到如下挑战：

- 如何保证查询响应性能。
- 服务发生故障后，数据降级方案。
- 数据监控预警方案及数据出现问题解决方案。

针对以上问题，下面进行一一详述:

性能响应优化

服务层处理数据过程中，由于数据量大，在查询时需要一定的响应时间，所以在保证响应性能方面，主要做了以下优化：

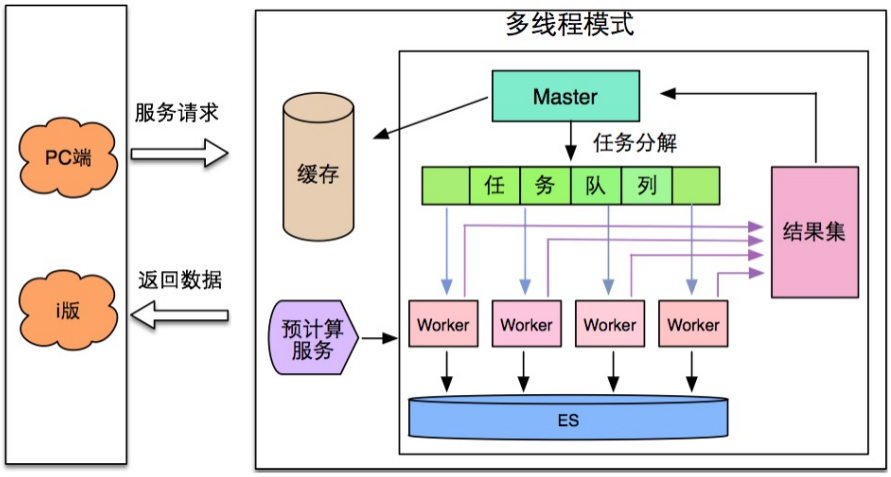


图7 性能响应优化

1. 项目初始由于数据量不是很大，采用单线程直接查询ES，能够满足需求。
2. 随着节假日来临，数据量大增，并行查询人次增多，查询响应变慢，无法快速响应结果，因此引入缓存技术，将中间结果进行缓存，并在缓存有效期内，直接读取缓存数据大大提高了时间效率；并且在此基础上，引入Master-Worker多线程模式，将多指标查询拆分，并行查询ES，使得查询响应大大提高。
3. 虽然问题得到解决，但仍存在一个问题，就是每次都是现查ES及部分中间缓存结果，尤其是第一次查询，需要完全走ES，这样就会让第一个查询数据的用户体验较差，因此引入预计算服务，通过定时调度任务，将部分重要维度下的指标进行预计算放入缓存，用户查询时直接读取缓存数据。而一些不常用的维度下的数据，采用的策略是，第一个用户查询时现查ES，并将结果数据预加载到缓存，后续所有用户再次查询直接读缓存数据，这样既能保证用户体验，也不至于占用太多缓存空间。

数据降级方案

使用缓存避免不了出现一些问题，比如缓存失效、缓存雪崩等问题，针对缓存雪崩问题，通过设置不同Key的过期时间能够很好的解决；而对于缓存数据失效，我们有自己的数据降级方案，具体方案如下：

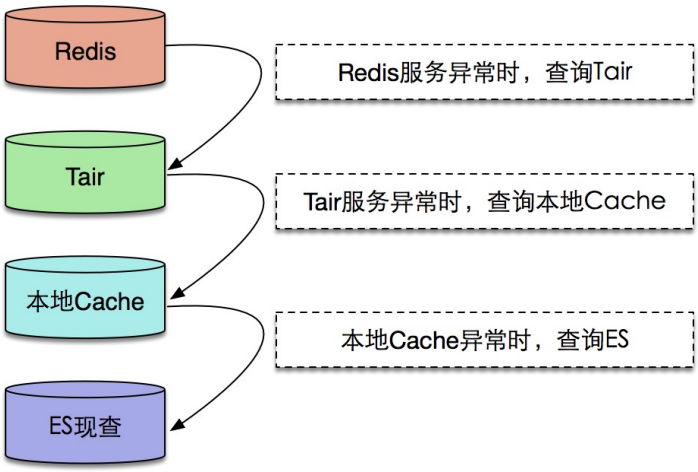


图8 数据降级方案

预计算数据会分别在Redis、Tair和本地缓存中存储一份以保证查询效率，当查询Redis数据不存在时，会去Tair中读取数据，Tair也为空时，会读取本地缓存，只有当本地缓存数据也为空时，才会现查ES做聚合

计算，这样也会降低ES的查询压力。

数据监控

实时监控预警非常重要，在数据出现问题时，一方面能够及时通知我们快速定位修复数据，另一方面也能够及时周知业务同学，避免做出错误分析。基于此，我们做了两方面的实时监控，其一是对源实时流在Storm处理层面的监控，确保源实时流正确生产；其二是对展示的汇总数据进行监控，确保产品展示指标数据正常。 针对数据出现问题预警，我们在解决方案上规范了流程：

1. 监控报警机制及时周知相关同学。
2. 第一时间通过产品上方的黄条提示用户哪些数据异常。
3. 快速定位问题，给出修复方案。

目前对于实时异常数据的修补，主要有两种方法：

1. 针对特殊情况的数据修补方案第一灵活指定Offset，重新消费Kafka数据。
2. 预留了Hive2es的准实时重导功能,确保生产数据的准确和完整。

数据安全

在以数据取胜的时代，数据的安全不言而喻，我们采用公司提供的UPM权限接口进行二级权限管理并加入审计功能及水印功能，能够准确记录用户的所有访问以及操作记录，并且将日志数据格式化到数据库中，进行实时监控分析。

总结

实时数据可以为业务特定场景分析决策提供巨大支持，尤其对于大交通节假日及春运期间。在大交通实时战场沙盘产品化过程中，我们投入了大量的思考和实践，主要取得以下收益：

1. 可视化的产品，为业务方实时分析提供极大便利，取得较好的反馈。
2. 优化实时数据仓库建设，合理分层建模，规范命名设计，统一维度建设和指标口径，对外提供统一接口，保证数据规范准确。
3. 在Storm框架下实时开发和数据写入方面积累了一定的经验。
4. 服务层支持可配置信息，可以灵活配置个性化信息。
5. 服务层性能及获取数据策略的优化，为用户带来更好的产品体验。

加入我们

最后插播一个招聘广告，我们是一群擅长大数据领域数据建设、数仓建设、数据治理及数据BI应用建设的工程师，期待更多能手加入，感兴趣的可以投递个人简历到邮箱：yangguang09#meituan.com，欢迎您的加入。

作者介绍

- 娣娣，美团数据开发工程师，2015年加入美团，从事数据仓库建设、大数据产品开发工作。
- 晓磊，美团数据开发工程师，2017年加入美团，从事数据仓库建设、大数据产品开发工作。

流量运营数据产品最佳实践——美团旅行流量罗盘

作者: 冰 瑞芳 夷山

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。

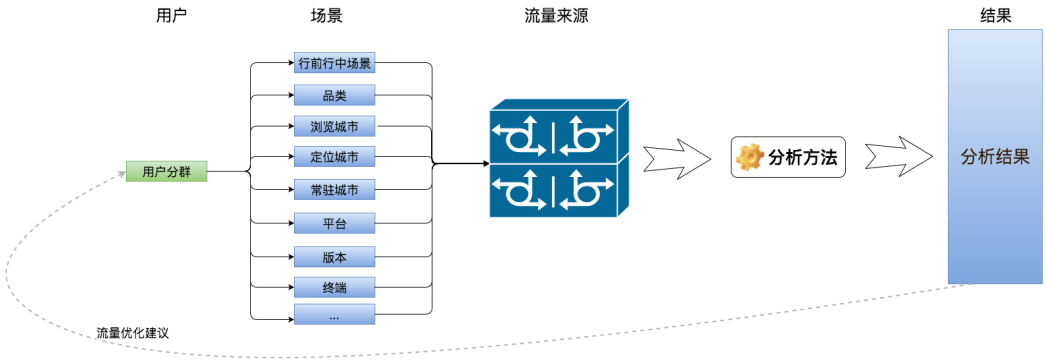


图1 产品结构图

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

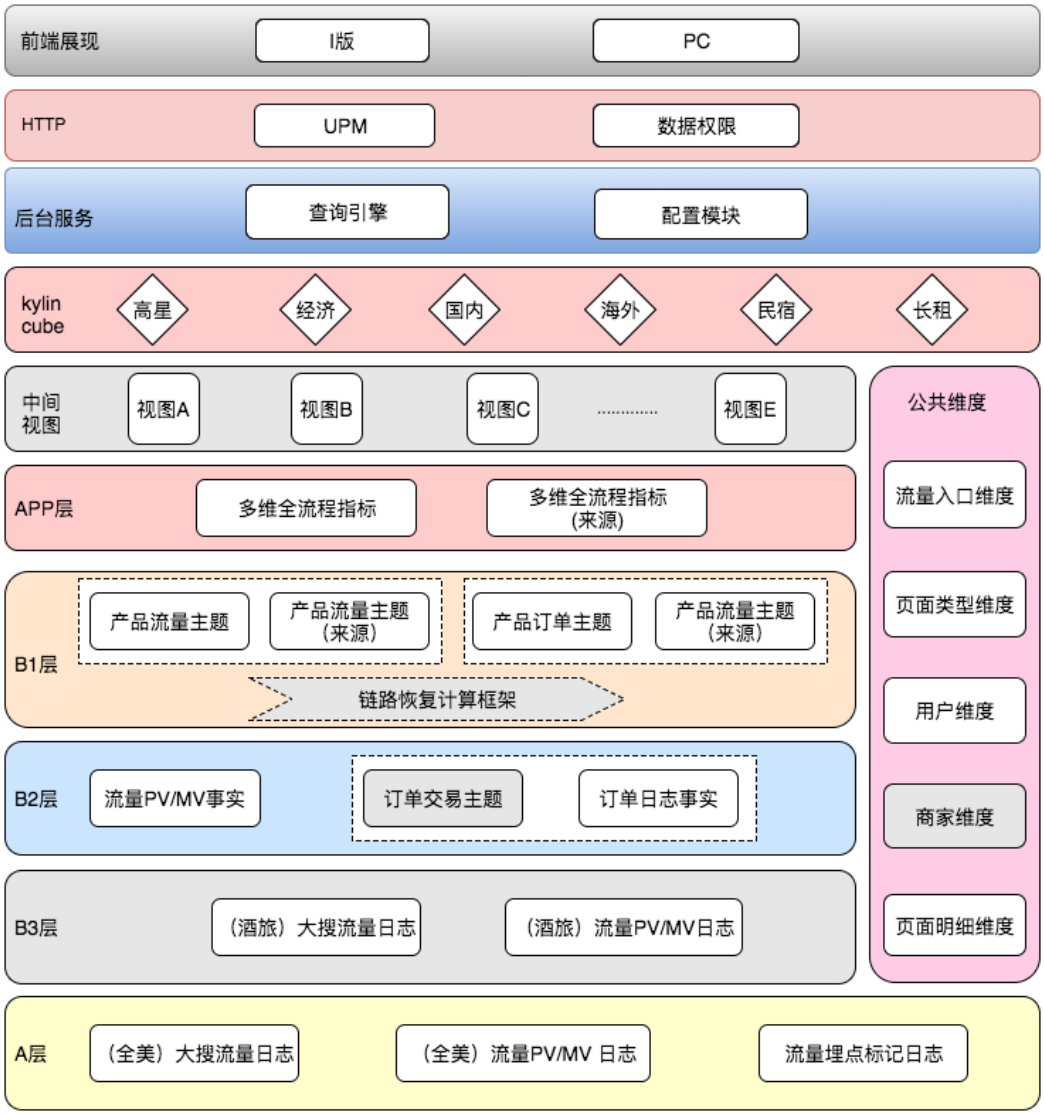


图2 流量罗盘体系架构

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。

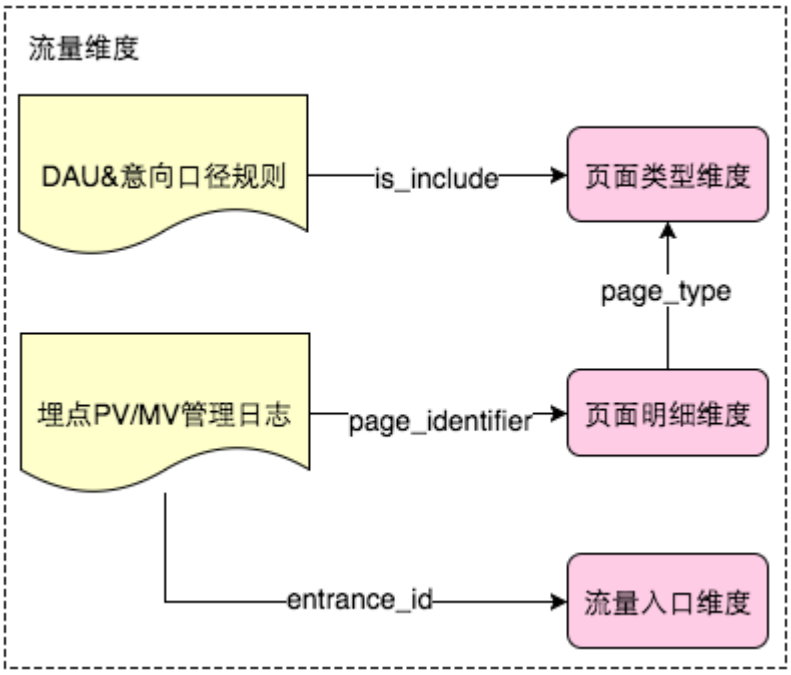


图3 公共维度来源图

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。

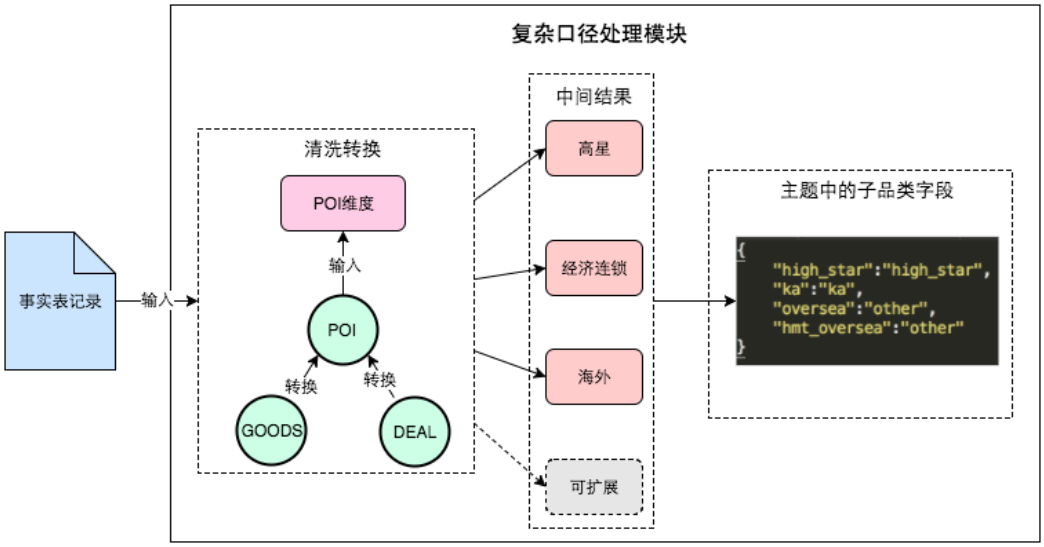


图4 口径处理模块

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。

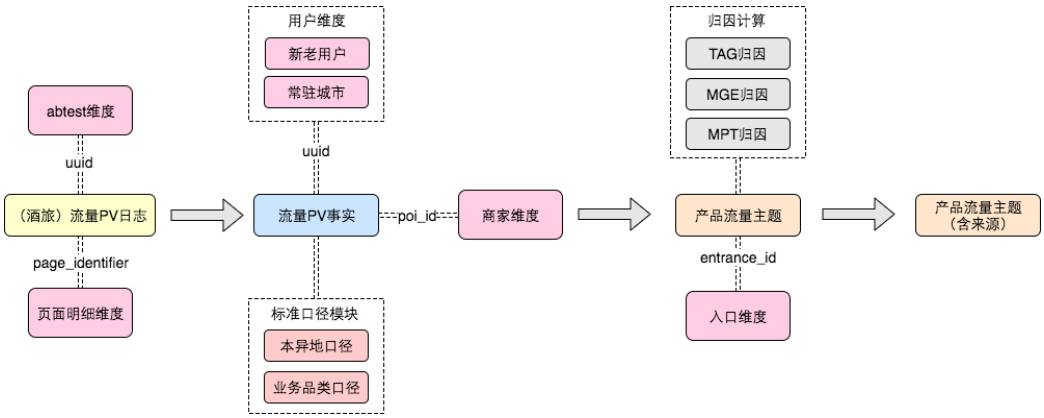


图5 主题模型计算流程

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。

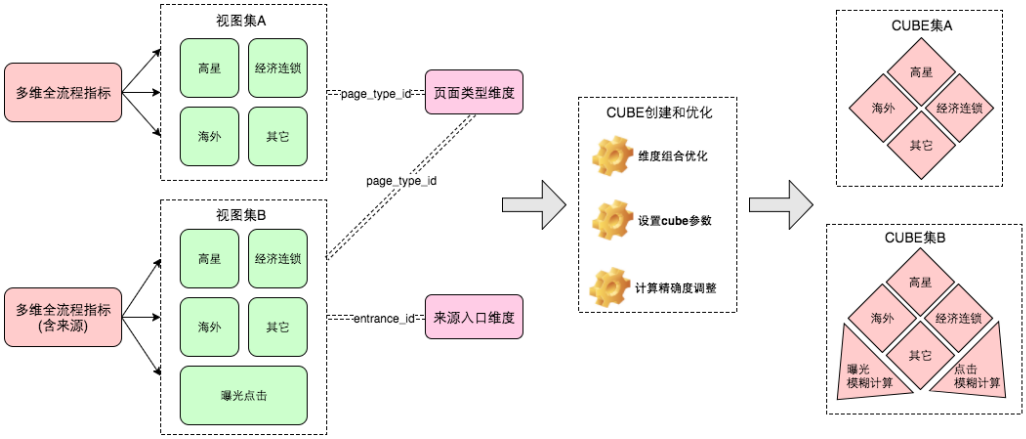


图6 应用层计算流程

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：

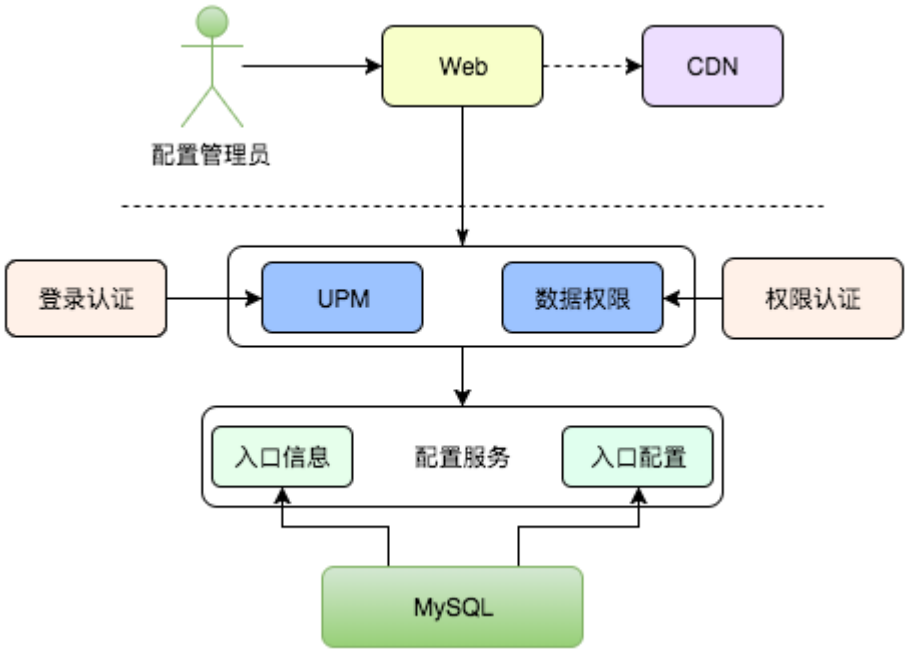


图7 配置模块交互

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：

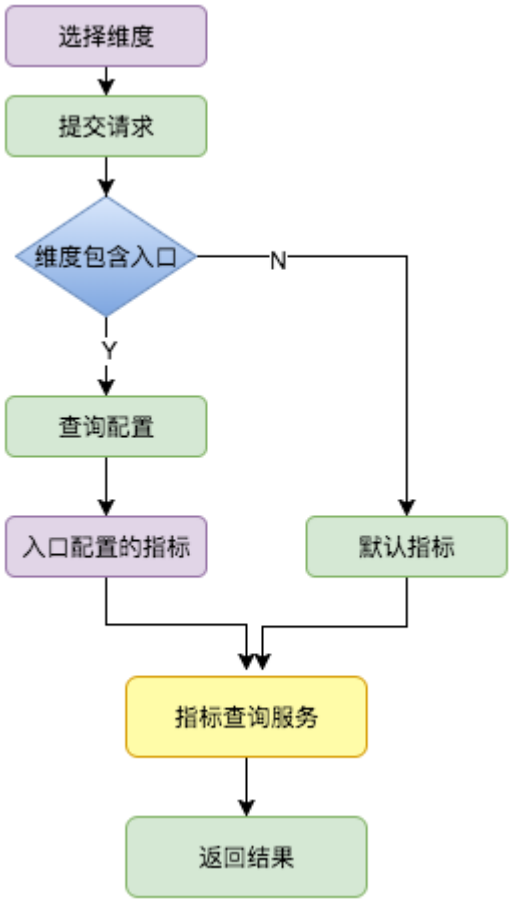


图8 查询服务流程图

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：

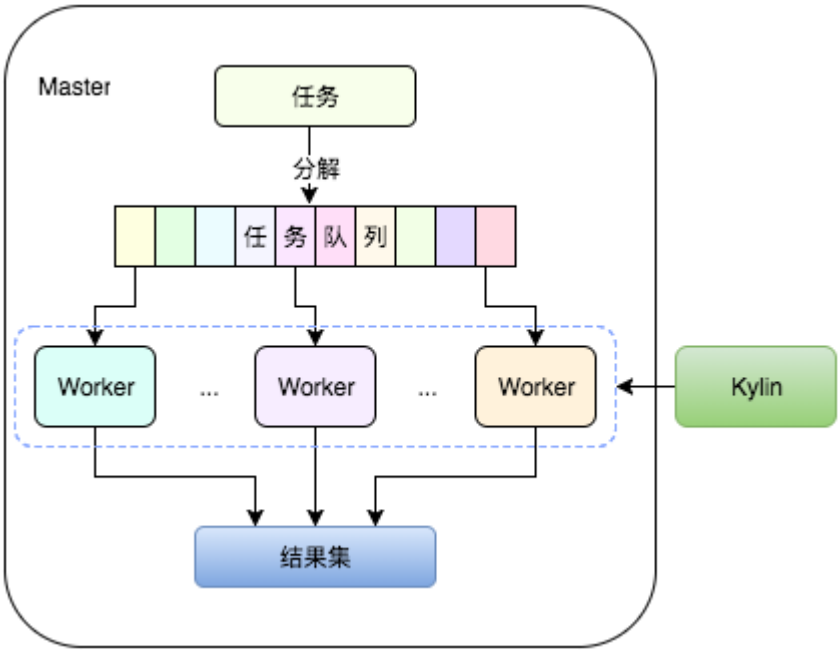


图9 任务拆分指标计算

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：

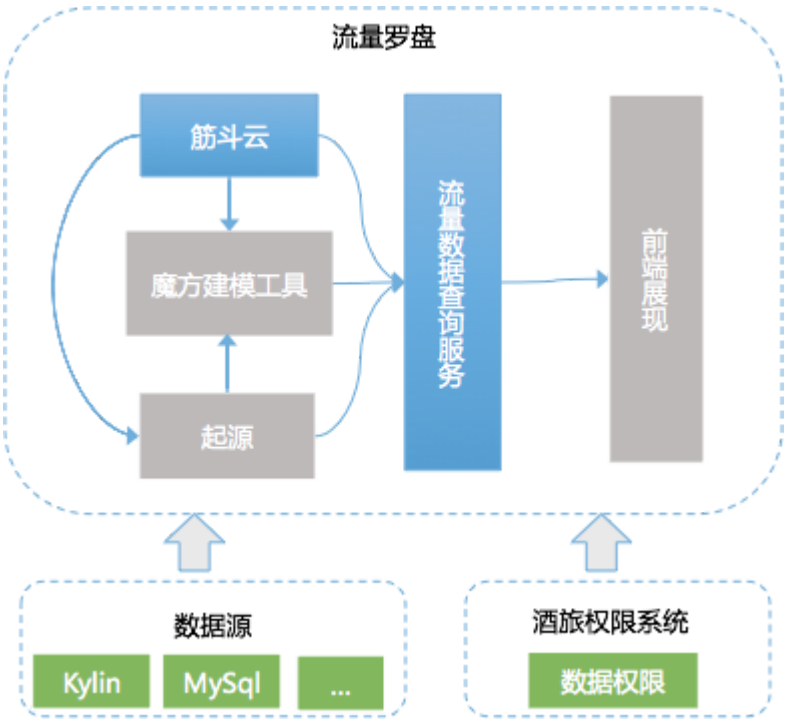


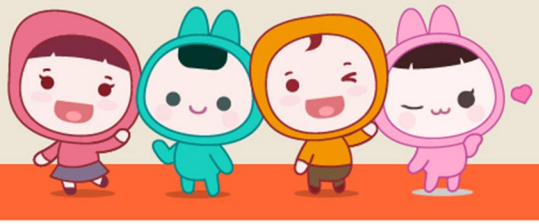
图10 产品架构图

作者简介

- 冰，美团平台数据中心高级研发工程师，2014年毕业于北京航空航天大学，2015年加入美团，负责酒旅事业群的数据仓库建设。
- 瑞芳，美团平台数据中心BI工具后台开发工程师，2016年毕业于北京工业大学，2017年加入美团点评数据中心，长期从事BI工具开发工作。
- 夷山，美团点评技术专家，现任TechClub–Java俱乐部主席，2006年毕业于武汉大学，先后就职于IBM、用友、风行以及阿里。2014年加入美团，长期致力于BI工具、数据安全与数据质量工作等方向。

招聘信息

最后插播一个招聘广告，有对数据产品工具开发感兴趣的可以发邮件给 fuyishan#meituan.com。我们是一群擅长大数据领域数据工具，数据治理，智能数据应用架构设计及产品研发的工程师。



扫码关注技术团队
微信公众号

tech.meituan.com
美团技术博客

CODE A BETTER LIFE
一 行 代 码　亿 万 生 活