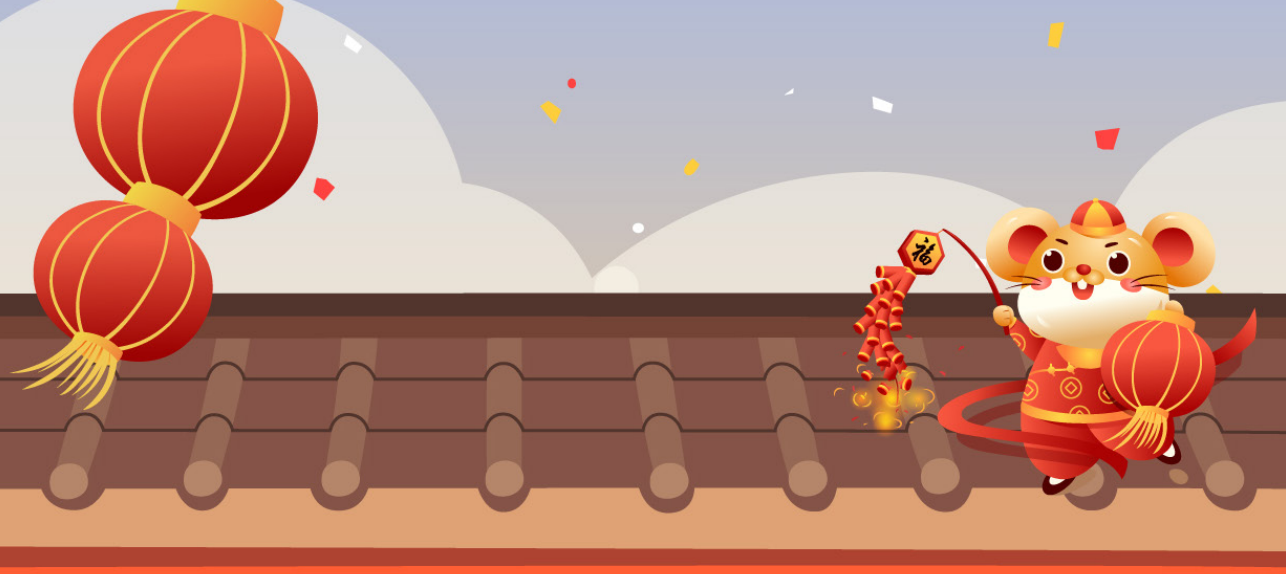


美团点评
2019 技术年货

CODE A BETTER LIFE

【大数据篇】

美团点评

微信扫码关注技术团队公众号

tech.meituan.com
美团技术博客

新年
快乐

目录

大数据

基于大数据，可以更好的赋能业务、优化开发、反馈产品、助力运营。在美团点评，大数据应用于各业务线，以数据驱动的方式帮助公司实现业务目标，持续提高公司的运营效率和核心竞争力。同时，美团点评大数据不只做平台能力建设，更注重平台建设与平台能力在数据领域的实践，从而形成体系化的数据解决方案。

2019 年，美团技术博客发布的大数据相关文章涉及数据治理、资源优化、大数据平台能力建设、非固化的探索式数据分析领域。《Hadoop YARN：调度性能优化实践》和《将军令：数据安全平台建设实践》阐述了美团点评大数据平台能力的建设；《OneData 建设探索之路：SaaS 收银运营数仓建设》、《研发团队资源成本优化实践》、《Jupyter 在美团民宿的应用实践》三篇文章分别从数据治理、资源优化、非固化的探索式数据分析三个方面讲述了美团点评不同业务场景的应用实践。

抛砖引玉，希望美团点评的这些经验能够对数据工程师们有所启迪，未来一起交流、成长。

Hadoop YARN：调度性能优化实践

世龙　廷稳

背景

YARN 作为 Hadoop 的资源管理系统，负责 Hadoop 集群上计算资源的管理和作业调度。

美团的 YARN 以社区 2.7.1 版本为基础构建分支。目前在 YARN 上支撑离线业务、实时业务以及机器学习业务。

- 离线业务主要运行的是 Hive on MapReduce，Spark SQL 为主的数据仓库作业。
- 实时业务主要运行 Spark Streaming，Flink 为主的实时流计算作业。
- 机器学习业务主要运行 TensorFlow，MXNet，MLX（美团点评自研的大规模机器学习系统）等计算作业。

YARN 面临高可用、扩展性、稳定性的问题很多。其中扩展性上遇到的最严重的，是集群和业务规模增长带来的调度器性能问题。从业务角度来看，假设集群 1000 台节点，每个节点提供 100 个 CPU 的计算能力。每个任务使用 1 个 CPU，平均执行时间 1 分钟。集群在高峰期始终有超过 10 万 CPU 的资源需求。集群的调度器平均每分钟只能调度 5 万的任务。从分钟级别观察，集群资源使用率是 50000/(100*1000)=0.5，那么集群就有 50% 的计算资源因为调度能力的问题而无法使用。

随着集群规模扩大以及业务量的增长，集群调度能力会随着压力增加而逐渐下降。假设调度能力依然保持不变，每分钟调度 5 万个任务，按照 5000 台节点的规模计算，如果不做任何优化改进，那么集群资源使用率为：50000/(100*5000) = 10%，剩余的 90% 的机器资源无法被利用起来。

这个问题解决后，集群在有空余资源的情况下，作业资源需求可以快速得到满

足，集群的计算资源得到充分地利用。

下文会逐步将 Hadoop YARN 调度系统的核心模块展开说明，揭开上述性能问题的根本原因，提出系统化的解决方案，最终 Hadoop YARN 达到支撑单集群万级别节点，支持并发运行数万作业的调度能力。

整体架构

YARN 架构

YARN 负责作业资源调度，在集群中找到满足业务的资源，帮助作业启动任务，管理作业的生命周期。

YARN 详细的架构设计请参考 Hadoop 官方文档。

资源抽象

YARN 在 cpu，memory 这两个资源维度对集群资源做了抽象。

```
class Resource{
  int cpu;    //cpu 核心个数
  int memory-mb; // 内存的 MB 数
}
```

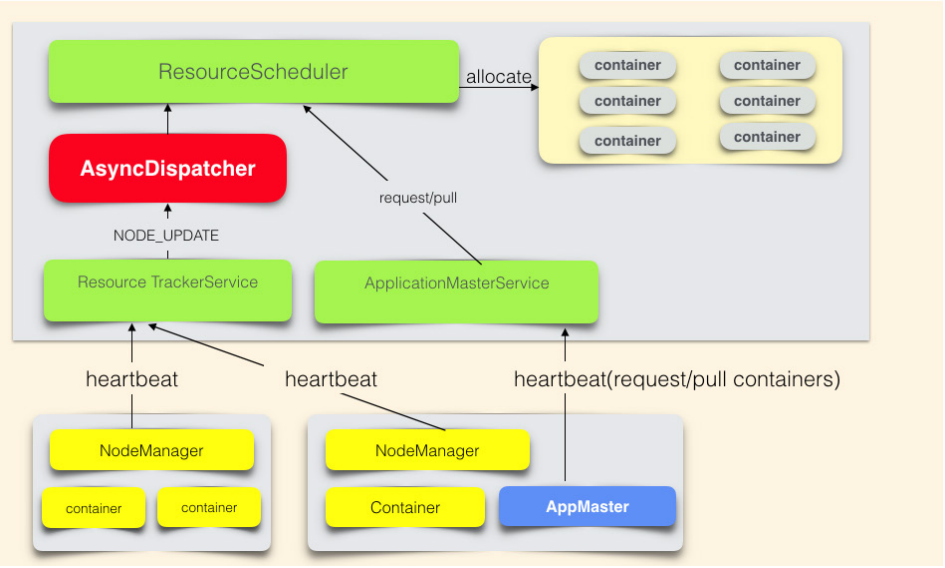
作业向 YARN 申请资源的请求是：List[ResourceRequest]

```
class ResourceRequest{
  int numContainers; // 需要的 container 个数
  Resource capability;// 每个 container 的资源
}
```

YARN 对作业响应是：List[Container]

```
class Container{
  ContainerId containerId; //YARN 全局唯一的 container 标示
  Resource capability;  // 该 container 的资源信息
  String nodeHttpAddress; // 该 container 可以启动的 NodeManager 的 hostname
}
```

YARN 调度架构



YARN 调度器

名词解释

- ResourceScheduler 是 YARN 的调度器，负责 Container 的分配。
- AsyncDispatcher 是单线程的事件分发器，负责向调度器发送调度事件。
- ResourceTrackerService 是资源跟踪服务，主要负责接收处理 NodeManager 的心跳信息。
- ApplicationMasterService 是作业的 RPC 服务，主要负责接收处理作业的心跳信息。
- AppMaster 是作业的程序控制器，负责跟 YARN 交互获取 / 释放资源。

调度流程

1. 作业资源申请过程：AppMaster 通过心跳告知 YARN 资源需求 (List[ResourceRequest])，并取回上次心跳之后，调度器已经分配好的资源 (List[Container])。

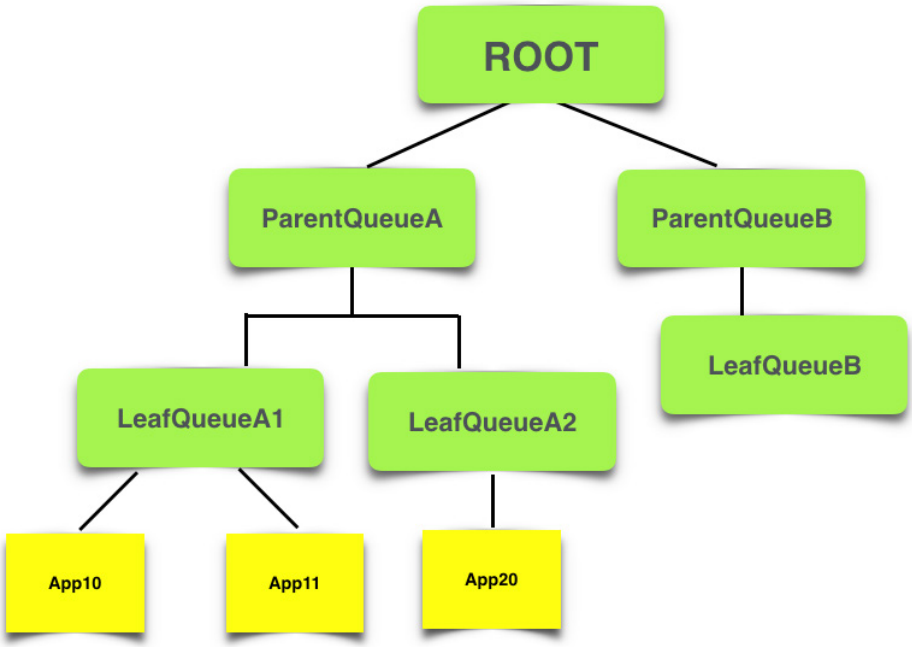
2. 调度器分配资源流程是：Nodemanager 心跳触发调度器为该 NodeManager 分配 Container。

资源申请和分配是异步进行的。ResourceScheduler 是抽象类，需要自行实现。社区实现了公平调度器 (FairScheduler) 和容量调度器 (CapacityScheduler)。美团点评根据自身的业务模式的特点，采用的是公平调度器。

公平调度器

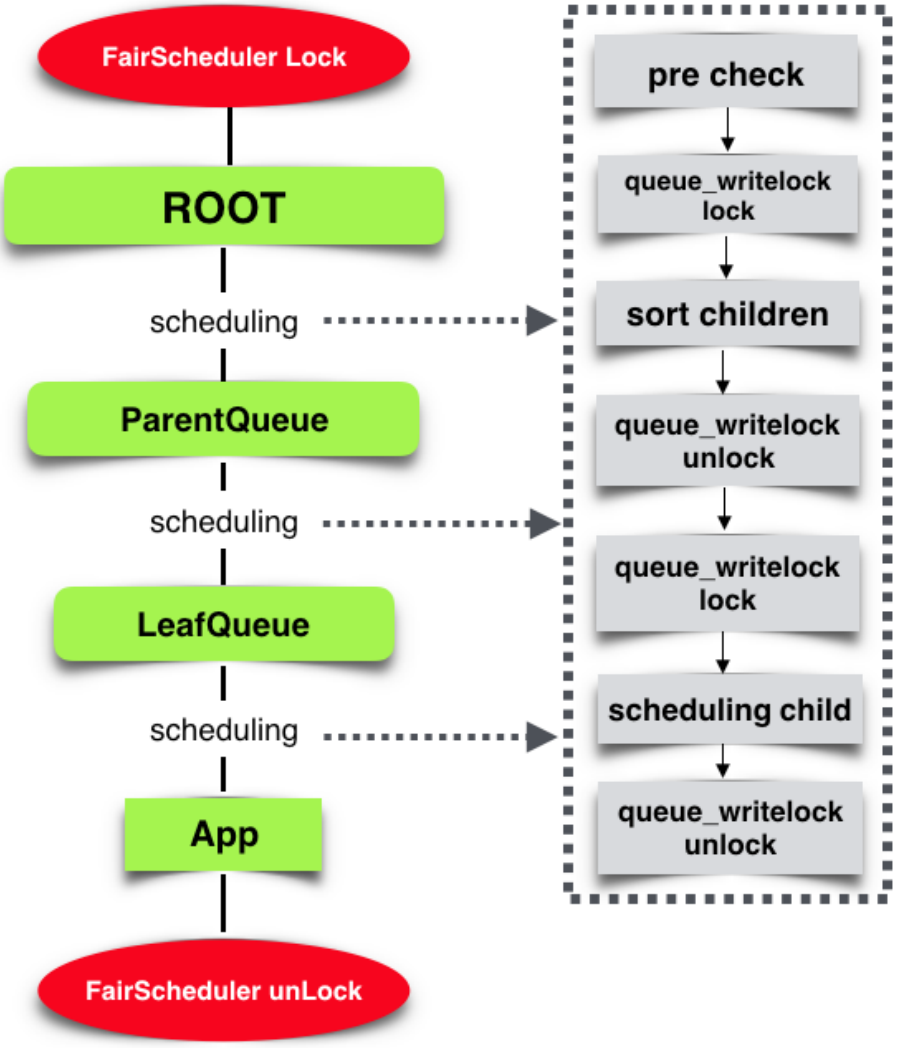
作业的组织方式

在公平调度器中，作业 (App) 是挂载如下图的树形队列的叶子。



队列结构

核心调度流程



核心调度流程

1. 调度器锁住 FairScheduler 对象，避免核心数据结构冲突。
2. 调度器选取集群的一个节点 (node)，从树形队列的根节点 ROOT 开始出发，每层队列都会按照公平策略选择一个子队列，最后在叶子队列按照公平策略选择一个 App，为这个 App 在 node 上找一块适配的资源。

对于每层队列进行如下流程：

1. 队列预先检查：检查队列的资源使用量是否已经超过了队列的 Quota
2. 排序子队列 /App：按照公平调度策略，对子队列 /App 进行排序
3. 递归调度子队列 /App

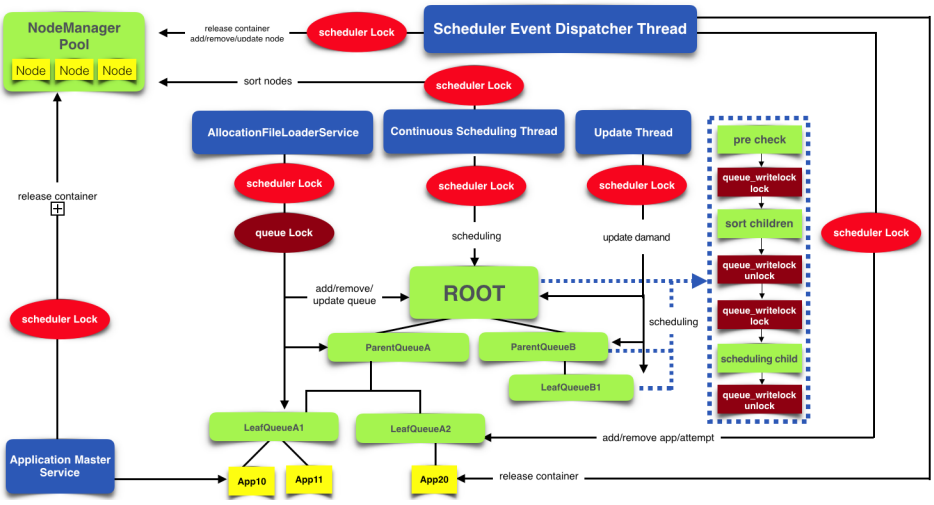
例如，某次调度的路径是 ROOT -> ParentQueueA -> LeafQueueA1 -> App11，这次调度会从 node 上给 App11 分配 Container。

伪代码

```
class FairScheduler{
  /* input：NodeId
   *  output：Resource 表示分配出来的某个 app 的一个 container 的资源量
   *  root 是树形队列 Queue 的根
   */
  synchronized Resource attemptScheduling(NodeId node){
    root.assignContainer(NodeId);
  }
}

class Queue{
  Resource assignContainer(NodeId node){
    if(! preCheck(node) ) return;  // 预先检查
      sort(this.children);  // 排序
    if(this.isParent){
      for(Queue q: this.children)
        q.assignContainer(node);  // 递归调用
    }else{
      for(App app: this.runnableApps)
        app.assignContainer(node);
    }
  }
}

class App{
  Resource assignContainer(NodeId node){
    ......
  }
}
```

公平调度器架构

公平调度器是一个多线程异步协作的架构，而为了保证调度过程中数据的一致性，在主要的流程中加入了 FairScheduler 对象锁。其中核心调度流程是单线程执行的。这意味着 Container 分配是串行的，这是调度器存在性能瓶颈的核心原因。



公平调度器架构

- scheduler Lock：FairScheduler 对象锁
- AllocationFileLoaderService：负责公平策略配置文件的热加载，更新队列数据结构
- Continuous Scheduling Thread：核心调度线程，不停地执行上节的核心调度流程
- Update Thread：更新队列资源需求，执行 Container 抢占流程等
- Scheduler Event Dispatcher Thread: 调度器事件的处理器，处理 App 新增，App 结束，node 新增，node 移除等事件

性能评估

上文介绍了公平调度器的架构，在大规模的业务压力下，这个系统存在性能问题。从应用层的表现看，作业资源需求得不到满足。从系统模块看，多个模块协同工

作，每个模块多多少少都存在性能问题。如何评估系统性能已经可以满足线上业务的需求？如何评估系统的业务承载能力？我们需要找到一个系统的性能目标。因此在谈性能优化方案之前，需要先说一说调度系统性能评估方法。

一般来说，在线业务系统的性能是用该系统能够承载的 QPS 和响应的 TP99 的延迟时间来评估，而调度系统与在线业务系统不同的是：调度系统的性能不能用 RPC（ResourceManager 接收 NodeManager 和 AppMaster 的 RPC 请求）的响应延迟来评估。原因是：这些 RPC 调用过程跟调度系统的调度过程是异步的，因此不论调度性能多么差，RPC 响应几乎不受影响。同理，不论 RPC 响应多么差，调度性能也几乎不受影响。

业务指标 – 有效调度

首先从满足业务需求角度分析调度系统的业务指标。调度系统的业务目标是满足业务资源需求。指标是：有效调度 (validSchedule)。在生产环境，只要 validSchedule 达标，我们就认为目前调度器是满足线上业务需求的。

定义 validSchedulePerMin 表示某一分钟的调度性能达标的情况。达标值为 1，不达标值为 0。

```
validPending = min(queuePending, QueueMaxQuota)
if  (usage / total  > 90% || validPending == 0):   validSchedulePerMin =
1 // 集群资源使用率高于
90%，或者集群有效资源需求为 0，这时调度器性能达标。
if (validPending > 0 &&  usage / total < 90%) : validSchedulePerMin =
0;// 集群资源使用率低于
90%，并且集群存在有效资源需求，这时调度器性能不达标。
```

- validPending 表示集群中作业有效的资源需求量
- queuePending 表示队列中所有作业的资源需求量
- QueueMaxQuota 表示该队列资源最大限额
- usage 表示集群已经使用的资源量
- total 表示集群总体资源

设置 90% 的原因是：资源池中的每个节点可能都有一小部分资源因为无法满足任何的资源需求，出现的资源碎片问题。这个问题类似 linux 内存的碎片问题。由于离线作业的任务执行时间非常短，资源很快可以得到回收。在离线计算场景，调度效率的重要性远远大于更精确地管理集群资源碎片，因此离线调度策略暂时没有考虑资源碎片的问题。

validSchedulePerDay 表示调度性能每天的达标率。 validSchedulePerDay = Σ validSchedulePerMin /1440

目前线上业务规模下，业务指标如下：validSchedulePerMin > 0.9; validSchedulePerDay > 0.99

系统性能指标 – 每秒调度 Container 数

调度系统的本质是为作业分配 Container，因此提出调度系统性能指标 CPS - 每秒调度 Container 数。 在生产环境，只要 validSchedule 达标，表明目前调度器是满足线上业务需求的。而在测试环境，需要关注不同压力条件下的 CPS，找到当前系统承载能力的上限，并进一步指导性能优化工作。

CPS 是与测试压力相关的，测试压力越大，CPS 可能越低。从上文公平调度器的架构可以看到，CPS 跟如下信息相关：

- 集群总体资源数；集群资源越多，集群可以并发运行的的 Container 越多，对调度系统产生越大的调度压力。目前每台物理机的 cpu、memory 资源量差距不大，因此集群总体资源数主要看集群的物理机节点个数。
- 集群中正在运行的 App 数；作业数越多，需要调度的信息越多，调度压力越大。
- 集群中的队列个数；队列数越多，需要调度的信息越多，调度压力越大。
- 集群中每个任务的执行时间；任务执行时间越短会导致资源释放越快，那么动态产生的空闲资源越多，对调度系统产生的压力越大。

例如，集群 1000 个节点，同时运行 1000 个 App，这些 App 分布在 500 个 Queue 上，每个 App 的每个 Container 执行时间是 1 分钟。在这样的压力条件下，

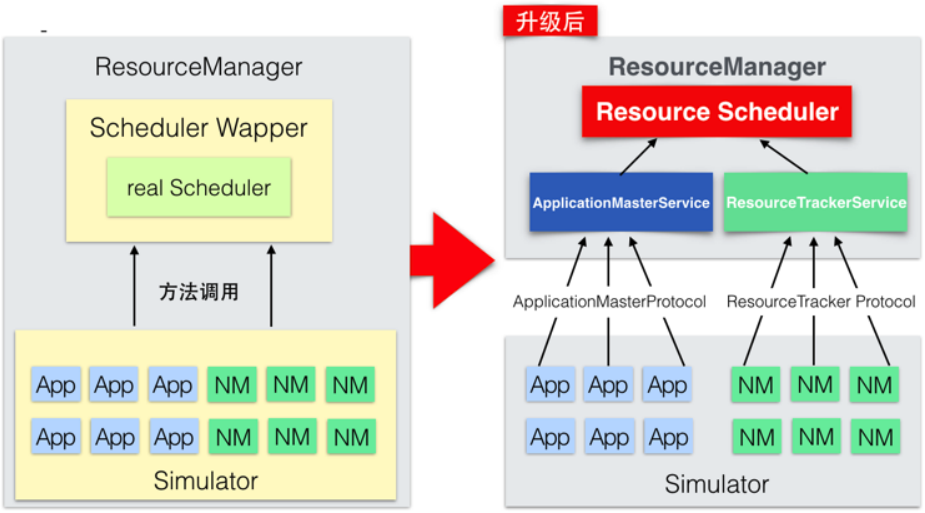
调度系统在有大量资源需求的情况下，每秒可以调度 1000 个 Container。那么在这个条件下，调度系统的 CPS 是 1000/s。

调度压力模拟器

在线上环境中，我们可以通过观察上文提到的调度系统的指标来看当前调度性能是否满足业务需求。但我们做了一个性能优化策略，不能直接到在线上环境去实验，因此我们必须有能力在线下环境验证调度器的性能是满足业务需求的，之后才能把实验有效的优化策略推广到线上环境。

那我们在线下也搭建一套跟线上规模一样的集群，是否就可以进行调度器性能优化的分析和研究呢？理论上是可以的，但这需要大量的物理机资源，对公司来说是个巨大的成本。因此我们需要一个调度器的压力模拟器，在不需要大量物理机资源的条件下，能够模拟 YARN 的调度过程。

社区提供了开源调度器的压力模拟工具 - Scheduler Load Simulater(SLS)。



调度压力模拟器

如上图，左侧是开源 SLS 的架构图，整体都在一个进程中，ResourceManager 模块里面有一个用线程模拟的 Scheduler。App 和 NM(NodeManager) 都是

由线程模拟。作业资源申请和 NM 节点心跳采用方法调用。

开源架构存在的问题有：

- 模拟大规模 APP 和 NM 需要开启大量的线程，导致调度器线程和 NM/App 的模拟线程争抢 cpu 资源，影响调度器的评估。
- SLS 的 Scheduler Wapper 中加入了不合理的逻辑，严重影响调度器的性能。
- SLS 为了通用性考虑，没有侵入 FairScheduler 的调度过程获取性能指标，仅仅从外围获取了 Queue 资源需求，Queue 资源使用量，App 资源需求，App 资源使用量等指标。这些指标都不是性能指标，无法利用这些指标分析系统性能瓶颈。

针对存在的问题，我们进行了架构改造。右侧是改造后的架构图，从 SLS 中剥离 Scheduler Wapper 的模拟逻辑，用真实的 ResourceManager 代替。SLS 仅仅负责模拟作业的资源申请和节点的心跳汇报。ResourceManager 是真实的，线上生产环境和线下压测环境暴露的指标是完全一样的，因此线上线下可以很直观地进行指标对比。详细代码参考：YARN-7672

细粒度监控指标

利用调度压力模拟器进行压测，观察到 validSchedule 不达标，但依然不清楚性能瓶颈到底在哪里。因此需要细粒度指标来确定性能的瓶颈点。由于调度过程是单线程的，因此细粒度指标获取的手段是侵入 FairScheduler，在调度流程中采集关键函数每分钟的时间消耗。目标是找到花费时间占比最多的函数，从而定位系统瓶颈。例如：在 preCheck 函数的前后加入时间统计，就可以收集到调度过程中 preCheck 消耗的时间。

基于以上的思路，我们定义了 10 多个细粒度指标，比较关键的指标有：

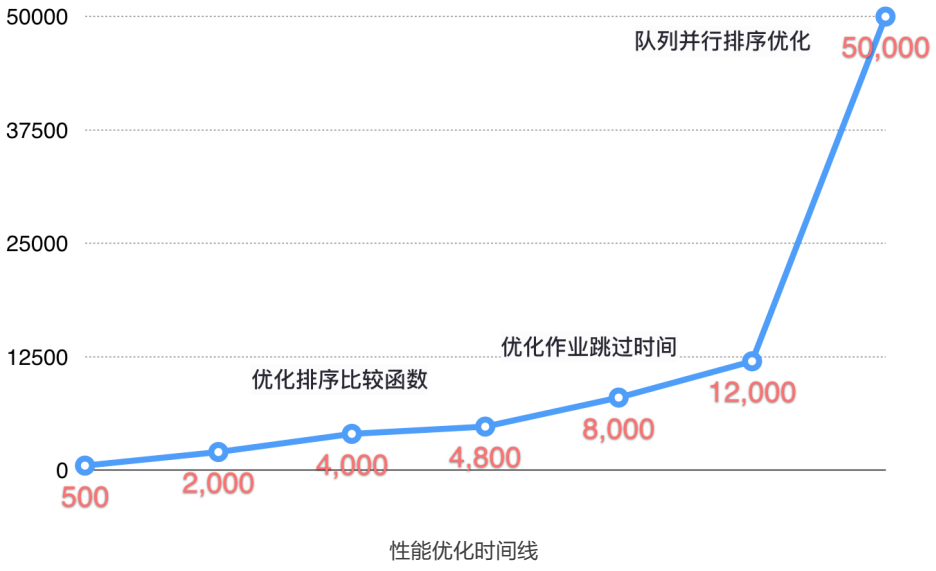
- 每分钟父队列 preCheck 时间
- 每分钟父队列排序时间
- 每分钟子队列 preCheck 时间

- 每分钟子队列排序时间
- 每分钟为作业分配资源的时间
- 每分钟因为作业无资源需求而花费的时间

关键优化点

第一次做压测，给定的压力就是当时线上生产环境峰值的压力情况（1000 节点、1000 作业并发、500 队列、单 Container 执行时间 40 秒）。经过优化后，调度器性能提升，满足业务需求，之后通过预估业务规模增长来调整测试压力，反复迭代地进行优化工作。

下图是性能优化时间线，纵轴为调度性能 CPS。



性能优化时间线

优化排序比较函数

在核心调度流程中，第 2 步是排序子队列。观察细粒度指标，可以很清楚地看到每分钟调度流程总共用时 50 秒，其中排序时间占用了 30 秒，占了最大比例，因此首先考虑优化排序时间。

排序本身用的快速排序算法，已经没有优化空间。进一步分析排序比较函数，发

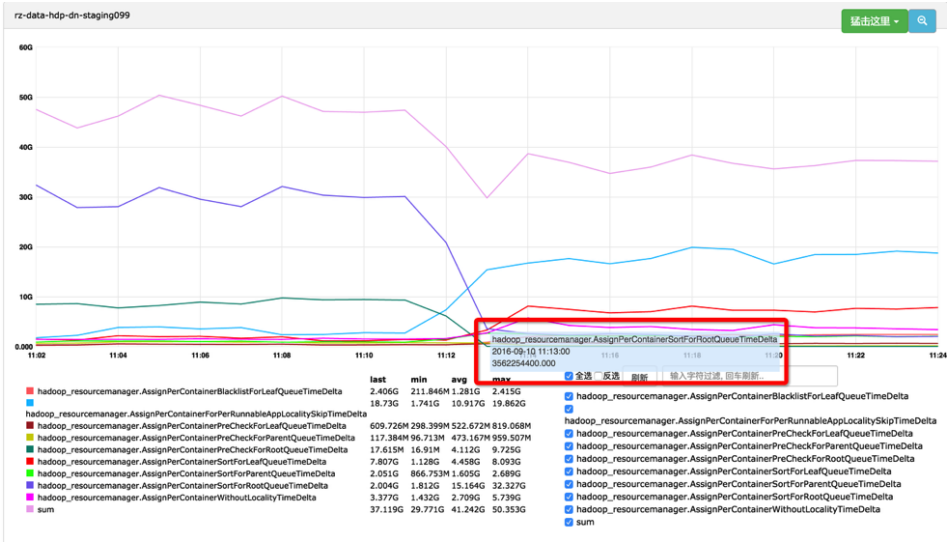
现排序比较函数的时间复杂度非常高。

计算复杂度最高的部分是：需要获取队列 / 作业的资源使用情况 (resourceUsage)。原算法中，每 2 个队列进行比较，需要获取 resourceUsage 的时候，程序都是现场计算。计算方式是递归累加该队列下所有作业的 resourceUsage。这造成了巨大的重复计算量。

优化策略：将现场计算优化为提前计算。

提前计算算法：当为某个 App 分配了一个 Container（资源量定义为 containerResource），那么递归调整父队列的 resourceUsage，让父队列的 resourceUsage += containerResource。当释放某个 App 的一个 Container，同样的道理，让父队列 resourceUsage -= containerResource。 利用提前计算算法，队列 resourceUsage 的统计时间复杂度降低到 O(1)。

优化效果：排序相关的细粒度指标耗时明显下降。



优化排序比较函数效果

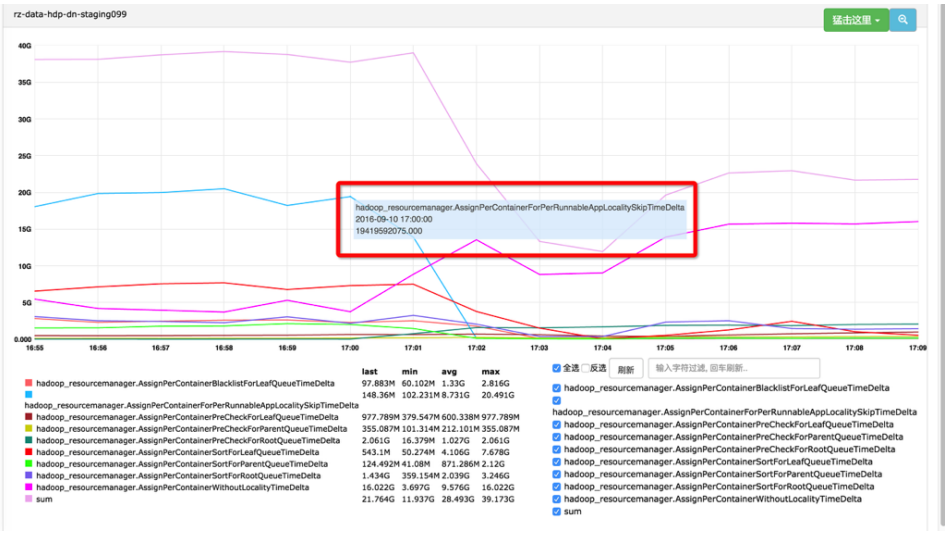
红框中的指标表示每分钟调度器用来做队列 / 作业排序的时间。从图中可以看出，经过优化，排序时间从每分钟 30G（30 秒）下降到 5G（5 秒）以内。 详细代码参考：YARN-5969

优化作业跳过时间

从上图看，优化排序比较函数后，蓝色的线有明显的增加，从 2 秒增加到了 20 秒。这条蓝线指标含义是每分钟调度器跳过没有资源需求的作业花费的时间。从时间占比角度来看，目前优化目标是减少这条蓝线的时间。

分析代码发现，所有队列 / 作业都会参与调度。但其实很多队列 / 作业根本没有资源需求，并不需要参与调度。因此优化策略是：在排序之前，从队列的 Children 中剔除掉没有资源需求的队列 / 作业。

优化效果：这个指标从 20 秒下降到几乎可以忽略不计。详细代码参考：YARN-3547



优化作业跳过时间

这时，从上图中可以明显看到有一条线呈现上升趋势，并且这个指标占了整个调度时间的最大比例。这条线对应的指标含义是确定要调度的作业后，调度器为这个作业分配出一个 Container 花费的时间。这部分逻辑平均执行一次的时间在 0.02ms 以内，并且不会随着集群规模、作业规模的增加而增加 , 因此暂时不做进一步优化。

队列并行排序优化

从核心调度流程可以看出，分配每一个 Container，都需要进行队列的排序。排序的时间会随着业务规模增加（作业数、队列数的增加）而线性增加。

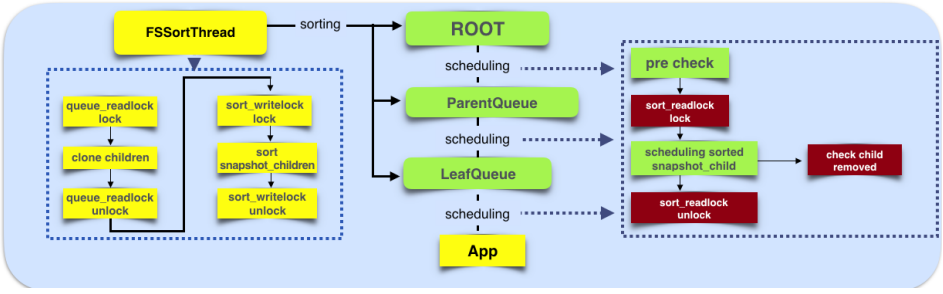
架构思考：对于公平调度器来说，排序是为了实现公平的调度策略，但资源需求是时时刻刻变化的，每次变化，都会引起作业资源使用的不公平。即使分配每一个 Container 时都进行排序，也无法在整个时间轴上达成公平策略。

例如，集群有 10 个 cpu，T1 时刻，集群只有一个作业 App1 在运行，申请了 10 个 cpu，那么集群会把这 10 个 cpu 都分配给 App1。T2 时刻 (T2 > T1)，集群中新来一个作业 App2，这时集群已经没有资源了，因此无法为 App2 分配资源。这时集群中 App1 和 App2 对资源的使用是不公平的。从这个例子看，仅仅通过调度的分配算法是无法在时间轴上实现公平调度。

目前公平调度器的公平策略是保证集群在某一时刻资源调度的公平。在整个时间轴上是需要抢占策略来补充达到公平的目标。 因此从时间轴的角度考虑，没有必要在分配每一个 Container 时都进行排序。

综上分析，优化策略是排序过程与调度过程并行化。要点如下：

1. 调度过程不再进行排序的步骤。
2. 独立的线程池处理所有队列的排序，其中每个线程处理一个队列的排序。
3. 排序之前，通过深度克隆队列 / 作业中用于排序部分的信息，保证排序过程中队列 / 作业的数据结构不变。

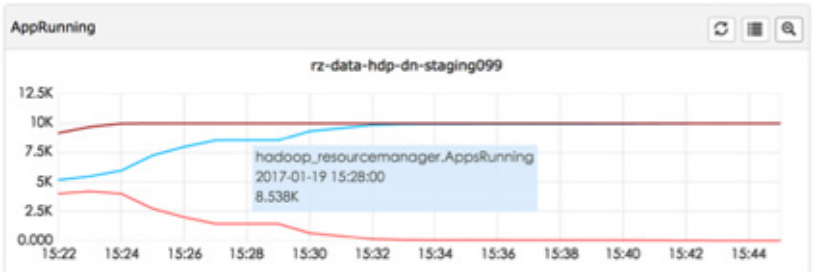


并行排序优化

优化效果如下：

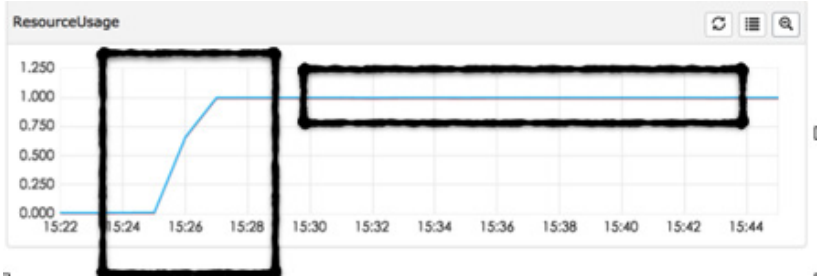
队列排序效率：利用线程池对 2000 个队列进行一次排序只需要 5 毫秒以内(2ms-5ms)，在一秒内至少可以完成 200 次排序，对业务完全没有影响。

在并行运行 1 万作业，集群 1.2 万的节点，队列个数 2000，单 Container 执行时间 40 秒的压力下，调度 CPS 达到 5 万，在一分钟内可以将整个集群资源打满，并持续打满。



并发作业数

作业资源需求量



集群资源使用率

上图中，15:26 分，pending 值是 0，表示这时集群目前所有的资源需求已经被调度完成。15:27 分，resourceUsage 达到 1.0，表示集群资源使用率为 100%，集群没有空闲资源。pending 值达到 4M（400 万 mb 的内存需求）是因为没有空闲资源导致的资源等待。

稳定上线的策略

线下压测的结果非常好，最终要上到线上才能达成业务目标。然而稳定上线是有难度的，原因：

- 线上环境和线下压测环境中的业务差别非常大。线下没问题，上线不一定没问题。
- 当时 YARN 集群只有一个，那么调度器也只有一个，如果调度器出现异常，是整个集群的灾难，导致整个集群不可用。

除了常规的单元测试、功能测试、压力测试、设置报警指标之外，我们根据业务场景提出了针对集群调度系统的上线策略。

在线回滚策略

离线生产的业务高峰在凌晨，因此凌晨服务出现故障的概率是最大的。而凌晨 RD 同学接到报警电话，执行通常的服务回滚流程（回滚代码，重启服务）的效率是很低的。并且重启期间，服务不可用，对业务产生了更长的不可用时间。因此我们针对调度器的每个优化策略都有参数配置。只需要修改参数配置，执行配置更新命令，那么在不重启服务的情况下，就可以改变调度器的执行逻辑，将执行逻辑切换回优化前的流程。

这里的关键问题是：系统通过配置加载线程更新了调度器某个参数的值，而调度线程也同时在按照这个参数值进行工作。在一次调度过程中可能多次查看这个参数的值，并且根据参数值来执行相应的逻辑。调度线程在一次调度过程中观察到的参数值发生变化，就会导致系统异常。

处理办法是通过复制资源的方式，避免多线程共享资源引起数据不一致的问题。调度线程在每次调度开始阶段，先将当前所有性能优化参数进行复制，确保在本次调

度过程中观察到的参数不会变更。

数据自动校验策略

优化算法是为了提升性能，但要注意不能影响算法的输出结果，确保算法正确性。对于复杂的算法优化，确保算法正确性是一个很有难度的工作。

在“优化排序比较时间”的研发中，变更了队列 resourceUsage 的计算方法，从现场计算变更为提前计算。那么如何保证优化后算法计算出来的 resourceUsage 是正确的呢?

即使做了单元策略，功能测试，压力测试，但面对一个复杂系统，依然不能有 100% 的把握。 另外，未来系统升级也可能引起这部分功能的 bug。

算法变更后，如果新的 resourceUsage 计算错误，那么就会导致调度策略一直错误执行下去。从而影响队列的资源分配。会对业务产生巨大的影响。例如，业务拿不到原本的资源量，导致业务延迟。

通过原先现场计算的方式得到的所有队列的 resourceUsage 一定是正确的，定义为 oldResourceUsage。 算法优化后，通过提前计算的方式得到所有队列的 resourceUsage，定义为 newResourceUsage。

在系统中，定期对 oldResourceUsage 和 newResourceUsage 进行比较，如果发现数据不一致，说明优化的算法有 bug，newResourceUsage 计算错误。这时系统会向 RD 发送报警通知，同时自动地将所有计算错误的数据用正确的数据替换，使得错误得到及时自动修正。

总结与未来展望

本文主要介绍了美团点评 Hadoop YARN 集群公平调度器的性能优化实践。

1. 做性能优化，首先要定义宏观的性能指标，从而能够评估系统的性能。
2. 定义压测需要观察的细粒度指标，才能清晰看到系统的瓶颈。
3. 工欲善其事，必先利其器。高效的压力测试工具是性能优化必备的利器。
4. 优化算法的思路主要有：降低算法时间复杂度；减少重复计算和不必要的计

算；并行化。

5. 性能优化是永无止境的，要根据真实业务来合理预估业务压力，逐步开展性能优化的工作。
6. 代码上线需谨慎，做好防御方案。

单个 YARN 集群调度器的性能优化总是有限的，目前我们可以支持 1 万节点的集群规模，那么未来 10 万，100 万的节点我们如何应对?

我们的解决思路是：基于社区的思路，设计适合美团点评的业务场景的技术方案。社区 Hadoop 3.0 研发了 Global Scheduling，完全颠覆了目前 YARN 调度器的架构，可以极大提高单集群调度性能。我们正在跟进这个 Feature。社区的 YARN Federation 已经逐步完善。该架构可以支撑多个 YARN 集群对外提供统一的集群计算服务，由于每个 YARN 集群都有自己的调度器，这相当于横向扩展了调度器的个数，从而提高集群整体的调度能力。我们基于社区的架构，结合美团点评的业务场景，正在不断地完善美团点评的 YARN Federation。

作者简介

世龙、廷稳，美团用户平台大数据与算法部研发工程师。

团队介绍

数据平台资源调度团队，目标是建设超大规模、高性能、支持异构计算资源和多场景的资源调度系统。目前管理的计算节点接近 3 万台，在单集群节点过万的规模下实现了单日数十万离线计算作业的高效调度，资源利用率超过 90%。资源调度系统同时实现了对实时计算作业、机器学习模型 Serving 服务等高可用场景的支持，可用性超过 99.9%。系统也提供了对 CPU/GPU 等异构资源的调度支持，实现了数千张 GPU 卡的高效调度，以及 CPU 资源的离线与训练混合调度，目前正在引入 NPU/FPGA 等更多异构资源，针对机器学习场景的特点实现更高效合理的调度策略。

团队有多个岗位正在招聘，如果你对超大规模系统的挑战感到兴奋，如果你对异构计算资源的调度策略感到好奇，欢迎加入我们，联系邮箱 tech@meituan.com，注明“用户平台大数据”。

将军令：数据安全平台建设实践

夷山　中华

背景

在大数据时代，数据已经成为公司的核心竞争力。此前，我们介绍了美团酒旅起源数据治理平台的建设与实践，主要是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。

近期，业内数据安全事件频发，给相关企业造成了无可挽回的损失，更为数据安全防护意识薄弱的企业敲响了警钟。如何对公司内部数据最为集中的数据分析、数据服务、数据治理等各种数据类产品进行权限管控，已经成为数据安全建设中最为重要的任务。

如果从控制力的角度来进行划分的话，权限管控可以分为功能级权限管控和数据级权限管控。早期的数据安全产品大多使用传统的权限模型，只能实现功能级权限管控，无法进行数据级权限管控。基于数据类产品更高的安全要求，我们需要构建一个同时满足各类产品数据安全的我们需要构建一个同时满足各类产品数据安全的平台。

为此，美团用户平台应用研发组不仅设计了能表达和管控各种复杂关系的权限模型，还针对事前、事中、事后等三个场景，分别设计了审批、权限、审计三个子系统以保障数据安全的完整闭环，进而满足数据安全的各种要求。

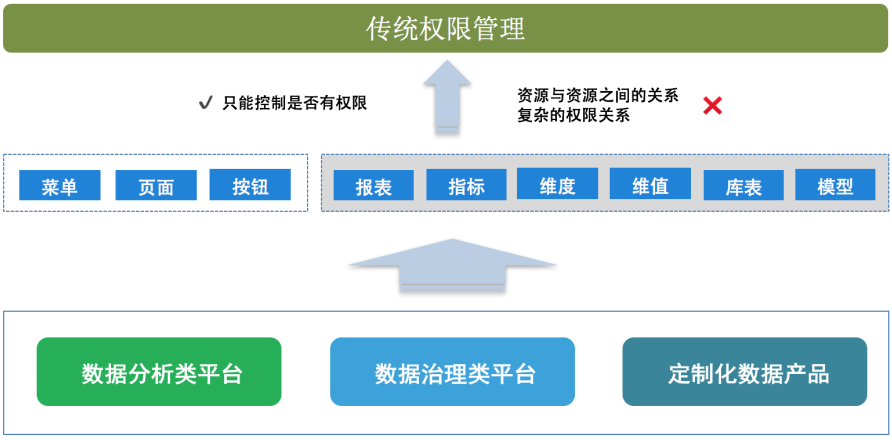


图 1　权限背景

功能应用类产品的权限表达，一般为“是否有权限”，而数据类产品权限表达的关系更加复杂。例如数据类产品的报表，不仅需要表达出用户能否访问这个报表，而且需要表达出用户能访问报表中的哪些维度、指标以及维值的范围。还需要告知这些维度指标来自于哪些库表模型，是否有权限访问以及创建报表。

权限模型

传统的权限模型有 ACL（Access Control List）访问控制列表，RBAC（Role-Based Access Control）基于角色的访问控制等。以上模型比较适用于应用类型产品的权限管控，而数据类型的产品对信息安全的要求更高，而且各类资源间的关系也更复杂，使用传统的模型难以将内部关系进行清晰的表达，所以我们在 RBAC 权限模型的基础上，扩展设计了新的权限模型。

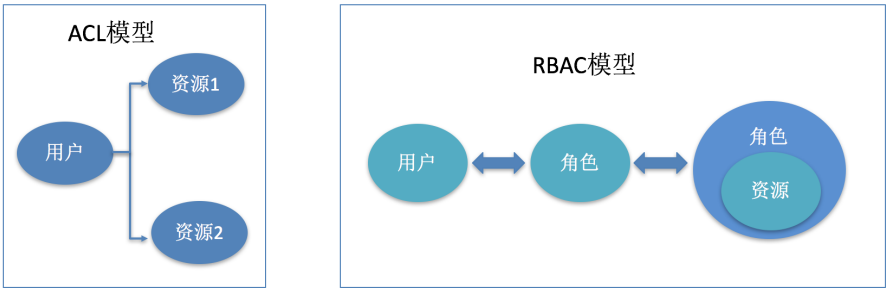


图 2　传统权限模型

如图 2 所示，传统的权限模型：

- ACL 访问控制列表，用户与权限直接关联，直接维护用户与列表中资源的关系从而达到权限管控的目的。
- RBAC 模型则是角色与权限进行关联，用户成为相应的角色而获得对应的权限。

为什么要设计新的权限模型?

1. ACL 模型是用户与资源直接建立关系，没有角色的概念。当某些用户需要一批同样资源的权限时，赋权操作就变得很复杂，此时这种模型就不太适应了。
2. RBAC 模型引入了角色的概念，角色与资源建立关系。当某些用户需要一批同样资源的权限时，只需要构建一个角色并赋予使用这批资源的权限。当用户加入这个角色时，则拥有该角色的所有权限。解决了赋权操作复杂的问题。

不过 ACL 模型和 RBAC 模型，都存在以下几个问题：

1. 数据类产品资源之间关系复杂，不能很好地表达这种复杂的关系。例如：一个报表下有多个标签页，一个标签页下有多个组件，一个组件下有多个维度、指标等。同时维度、指标又来自不同的数据模型、库表等。资源与资源之间存在关系，当管理员给一个用户赋予报表的全部或部分权限时，报表下的子资源需要同时获得对应的权限。
2. RBAC 模型中角色与角色之间没有对应的关系。例如：组织架构中，员工所

在的组织架构如下：华东区 / 销售一区 / 销售一组，员工拥有的角色是销售一组的角色。当角色之间没有关系时，员工如果需要华东区角色的权限时，需要添加到华东区的角色中。而如果角色与角色之间具有从属关系时，则能很好地解决这个问题。

新的权限模型是如何解决上面这些问题的：

1. 设计资源模型时，资源与资源之间具有从属关系，并且资源允许多层级，以树形结构展示。例如报表是一个父资源，标签、组件、维度指标都是报表下的子资源，这样赋权时能清晰地展示出报表资源与下面的子资源的关系，赋权和鉴权时才能满足各种权限控制的要求。
2. 角色与角色之间具有从属关系，例如员工在华东区 / 销售一区 / 销售一组的组织架构中，华东区 / 销售一区 / 销售一组这 3 个角色之间分别具有父子级的从属关系，当员工在销售一组部门下，则拥有华东区、销售一区、销售一组的所有权限。当权限不冲突时则直接合并所有权限，冲突时则以“就近原则”进行覆盖。

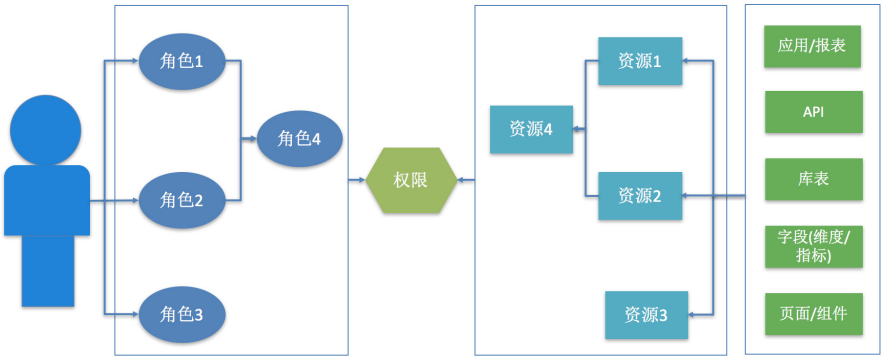


图 3　新的权限模型

如图 3 所示，新的权限模型包含 3 个部分，用户中心、资源中心、权限中心。

用户中心：用户管理、角色管理

- 角色分为个人、组织、自定义 3 种，一个用户可以同时拥有多个角色，比如用户默认对应一个个人角色，又可同时拥有在公司组织架构中组织角色、在自定义组织的自定义角色。
- 角色支持多层级，满足角色间权限继承的表达方式。
- 用户、部门信息 Mafka（美团基于 Kafka 开发的一个分布式消息中间件综合解决方案）实时更新，每天 ETL 定时同步，保证人员入职、转岗、调离权限实时同步。

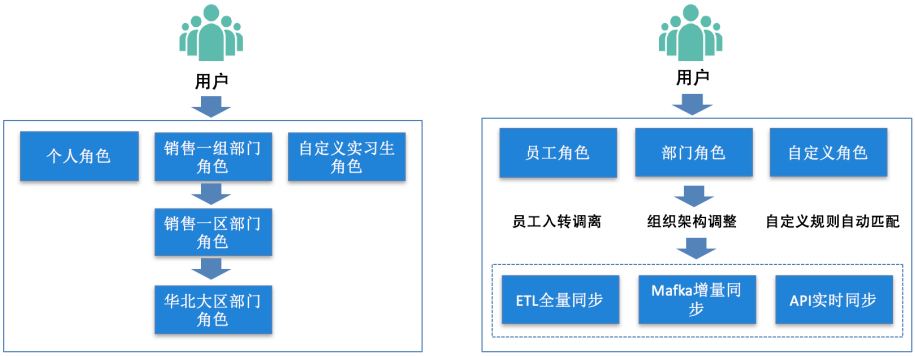


图 4　用户中心

资源中心：资源管理

- 资源类型支持自定义，在通用资源类型的基础上支持自定义的资源接入，满足各个系统不同资源的统一管控。
- 资源支持多层级，树形结构的资源展示方式便于资源的统一赋权鉴权；给一个报表资源赋权时，挂在报表下的维度、指标等资源能统一获得权限。
- 支持资源打包简化赋权流程。
- 资源安全密级、资源负责人，支持按照资源配置不同的审批模板进行权限自助申请。

图 5　资源中心

权限中心：角色与资源的关系的多种策略表达

- 范围策略：例如报表中的平台维度的维值包括美团和大众点评，赋权时，支持按要求给用户赋予部分或全部权限；鉴权时，按照规则解析为某人拥有某维度的部分或全部权限。
- 表达式策略：当把报表给用户赋权时，设置表达式为 limit 10，表示当前用户在该报表其他权限的基础上再进行限制，只能返回前 10 条记录。
- 权限自动合并：一个用户拥有多个角色，多角色的同一资源的权限鉴权时按照规则自动合并；规则解析时，权限数据不冲突时取合集，冲突时按照优先级取对应的值。
- 黑白名单：支持按照特定的规则，对某人针对某资源全面开发和封禁，黑白名单策略的优先级最高，其中黑名单高于白名单。

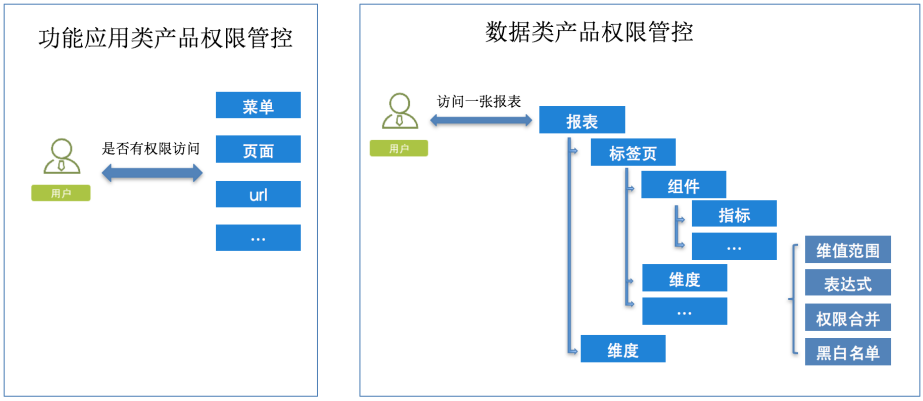


图 6　权限中心

挑战

在建设数据安全平台的过程中，主要面临以下几点挑战：

- 随着支持的业务线增加，通用平台的不能满足各个业务线的定制需求时，需要保证系统的灵活可扩展。
- 提供一个通用的数据安全平台，满足大部分的数据安全的要求，保证系统的通用性。
- 权限系统作为一个高 QPS 访问的系统，如何保证系统的高可用。

解决思路

1. 提供灵活可插拔的 Plugin 服务，在通用权限基础上，满足各个业务线灵活的权限管控要求。
2. 提供一个通用的数据安全平台，满足基本的权限、审批、审计的基础功能。
3. 微服务架构、核心与非核心服务分离、数据缓存降级满足系统高可用。

解决方案

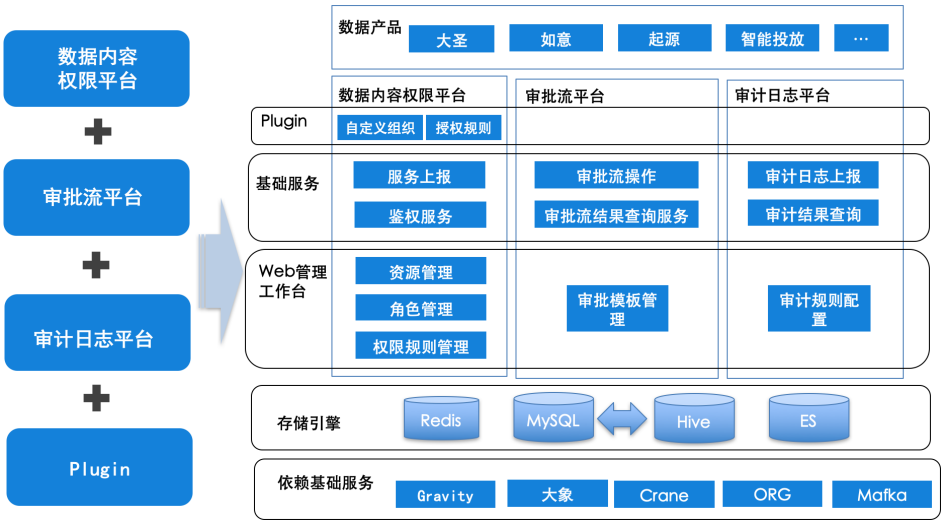


图 7　将军令整体架构

如图 7 所示，将军令分 3 块，数据内容权限平台、审批流平台、审计日志平台：

- 提供各种灵活可插拔的 Plugin 服务，支持在通用服务的基础基础上进行定制开发。
- 提供基础服务，满足各种通用的数据安全要求。
- 提供管理工作台，支持管理员对各种数据和规则进行页面管理和配置。

具体方案

Plugin 服务层，保证系统灵活可扩展

在满足通用权限的基础上，各个业务线难免会有定制的权限管控需求，于是设计了权限 Plugin 模块。

通用服务提供用户管理、资源管理、鉴权授权的服务，Plugin 调用基础服务实现特殊的权限管控。Plugin 模块的应用和数据各自单独管理，通过 RPC 方式调用通用服务实现灵活可插拔。后续 Plugin 模块的服务支持各个接入的应用单独定制开发。

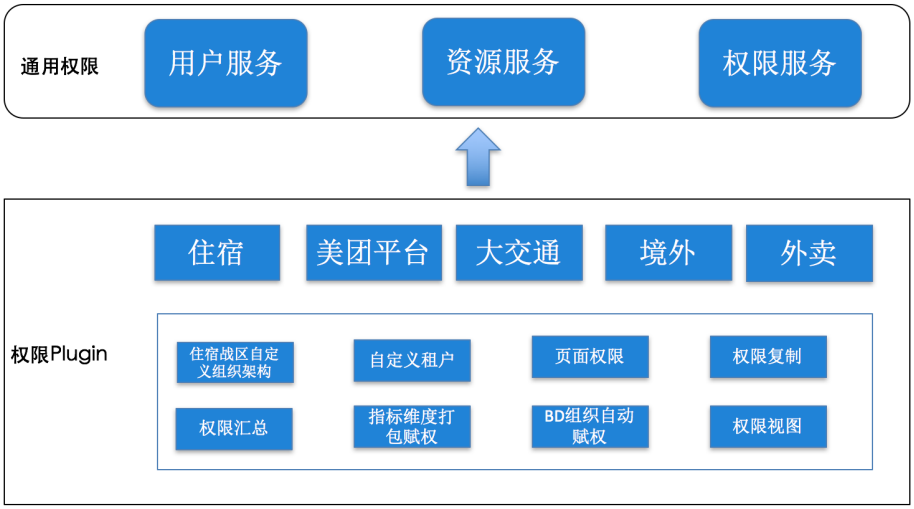


图 8　Plugin 服务

如图 8 所示，通用权限的服务与 Plugin 的服务是分离的，支持多个 Plugin 服务灵活可插拔：

- 通用服务提供用户、资源、鉴权授权等通用服务，大部分的系统基于通用服务即可实现权限管控要求。
- Plugin 服务基于通用服务对外提供的 SDK 进行拓展，各个 Plugin 服务单独部署，保证系统之间互相独立。

最终的权限实现分层管控，分为核心数据层（用户、资源、权限数据）和应用层。核心数据层的数据由通用服务进行管理，达到权限数据统一管控的要求。应用层以 Plugin 服务方式接入，Plugin 通过通用服务层对外的 SDK 进行权限数据读写，达到定制的管控要求。应用层的数据各自存储，可以自定义管控规则。接口之间的调用通过 BA 认证鉴权，保证服务之间调用的安全性。

基础服务层，保证系统通用性

通用权限系统架构

使用微服务架构设计，系统分为接入层、服务层、数据库层、以及外部服务层。主要包含以下几个核心服务：

- 用户服务：主要包含用户和部门信息同步、角色管理。
- 资源服务：包含资源注册、资源定时同步、资源密级及管理员管理、资源包管理。
- 赋权服务：权限自助申请、管理员赋权。
- 鉴权服务：提供各种鉴权的 SDK 供使用方调用。

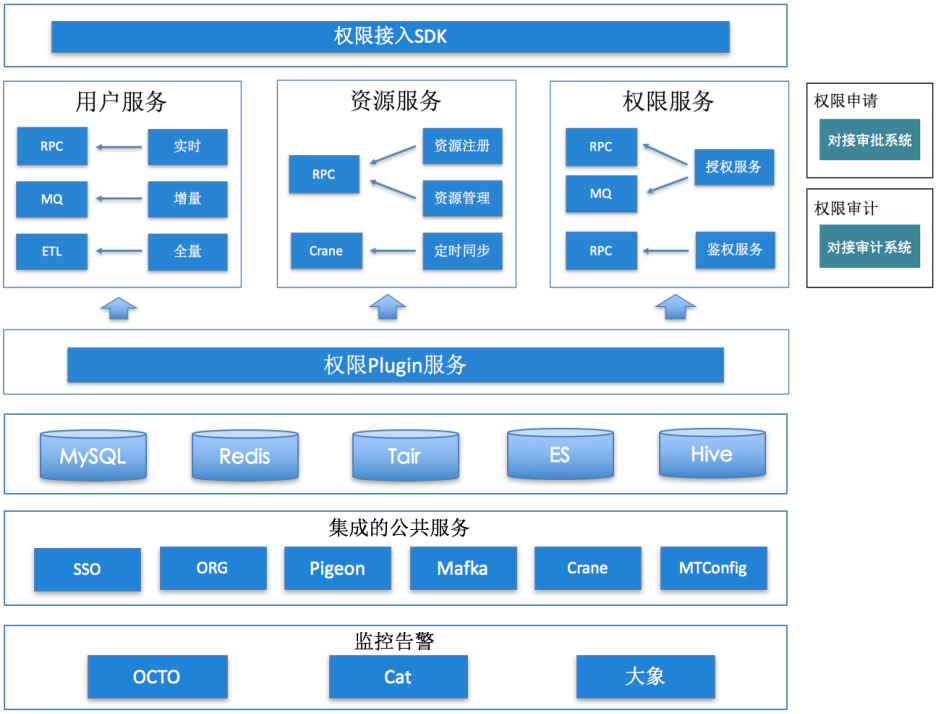


图 9　权限系统架构

如图 9 所示：

- 接入层：对外所有系统通过统一的 SDK 调用服务。
- 服务层：微服务架构，各个服务之间互相之间提供服务。
- 数据库层：合理利用缓存、数据降级，保证服务高可用。
- 集成公司公共服务，保证系统稳健运行。

审批系统架构

提供通用的审批服务，提供多级审批模板，使用时选择模板启动审批流，审批系统按照启动的参数进行规则解析，自动适配对应的审批流程。缩减接入流程支持一键接入。

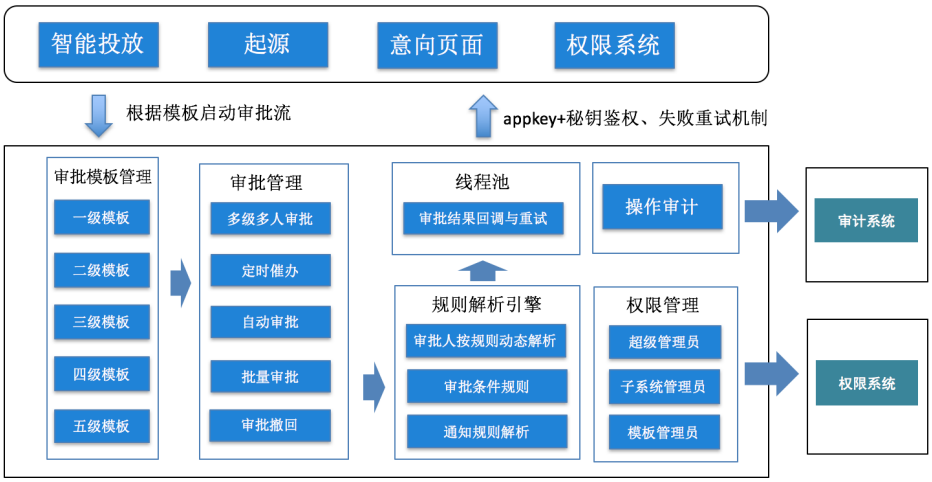


图 10　审批系统架构

如图 10 所示：优化审批接入流程，提供通用的审批服务，减少系统接入开发成本：

- 前期开发一个审批功能需要 6 个步骤，绘制流程图，配置审批的组和成员，配置通知的消息，配置事件映射，启动审批流，开发回调接口改状态。
- 而我们在平台的审批服务基础上进行封装，提供通用的审批模板，接入审批系统只需要选择模板启动审批流，并提供回调接口即可。能满足大部分的审批功能。

提供通用的规则解析引擎，支持审批人、审批条件、审批通知按照规则动态解析匹配。灵活实现自动审批、多人多级审批、定时催办等多种通用功能。

- 对接权限和审计系统，保证审批系统数据安全；
- 对接权限系统，提供管理员权限管控。

对接审计系统，操作数据落到审计系统便于后续的数据审计。

审计系统架构

提供通用的数据审计服务，客户端日志埋点上报，审计日志按类型落到 Elastic-search 中存储。对接如意可视化报表出审计报告，对接权限系统管控数据权限。

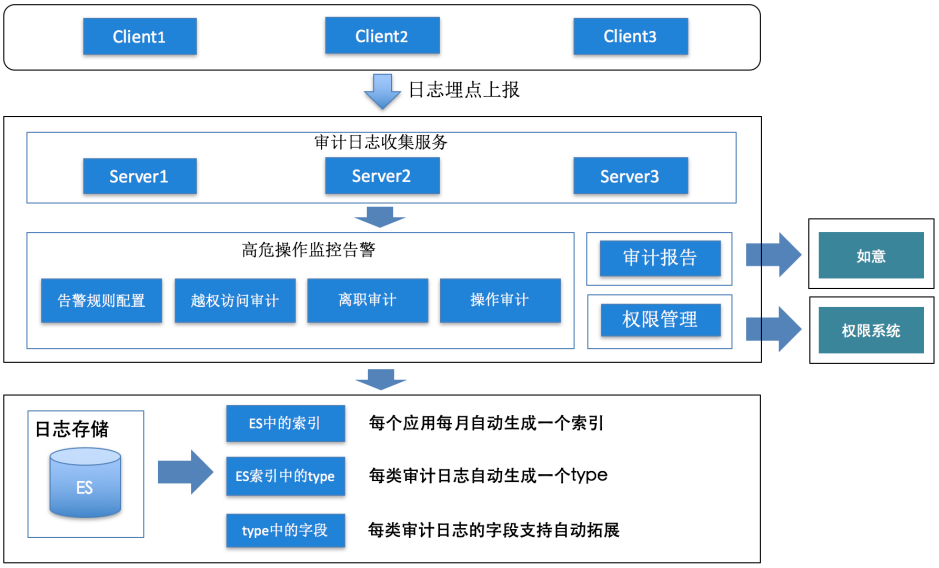


图 11　审计系统架构

如图 11 所示：审计数据模型层支持自动扩展：

- 每个应用对应一个 appkey，每个 appkey 按照模板分日期自动创建一个索引，支持自动扩展。
- 每种类型的审计日志对应 Elasticsearch 索引中的一个 type，新增一种操作日志时，type 自动创建。

- 审计日志中的字段对应 type 中的字段，新增字段时自动扩展。

保证系统高可用

微服务架构服务分离

随着系统的模块功能越来越多，单一架构模式已不再适合敏捷开发，模块越来越大系统启动则越慢，任一模块出错则整个系统的服务都不可用。

为了保证服务的高可用和扩展性，于是以微服务架构把模块进行拆分，并把核心与非核心服务进行分离。

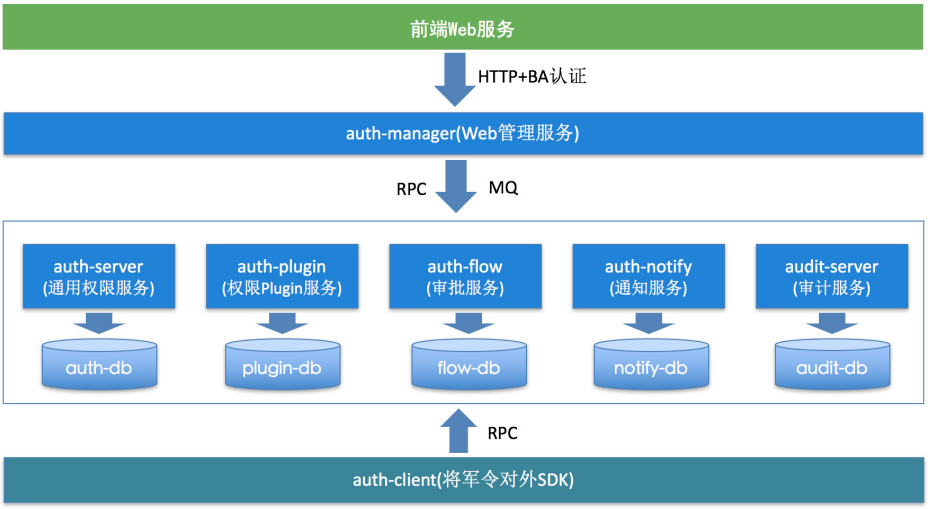


图 12　微服务架构

如图 12 所示：

- 前端接入层通过 HTTP 接入，BA 认证校验请求合法性，通过 Nginx 负载均衡。
- 管理控制台，通过调用服务层的各个服务实现统一管理。
- 服务层，抽象系统各个模块，每个模块都是一个微服务，每一个微服务都独立部署，可以根据每个服务的规模按需部署。
- Client 层，对外提供统一的 Pigeon（美团内部分布式服务 RPC 通信框架）接口，通过 POM 引入调用服务层各个服务。

权限继承

由于资源支持多层级，设计权限模型时支持权限继承，当赋权时开启继承，则用户默认拥有该资源以及下面所有资源的全部权限，数据存储时只需要存储祖先资源与用户之间的关系。大大减少了权限矩阵大小。

图 13　权限继承

权限数据存储

接入的系统越多，则资源和用户就越多。随着系统运行越久，对应的权限数据也会随之快速增长。如何在数据增长的同时保证接口的性能和高可用。

权限备份与恢复

参照 HBase 的版本号和 MySQL 的 Binlog 的设计思路，赋权时权限只存储当前用户最新权限数据，历史权限数据和操作记录用版本号的方式存储到 Elasticsearch 中。用户鉴权时只需要查询 MySQL 的权限数据即可，保证鉴权接

口的高效性。

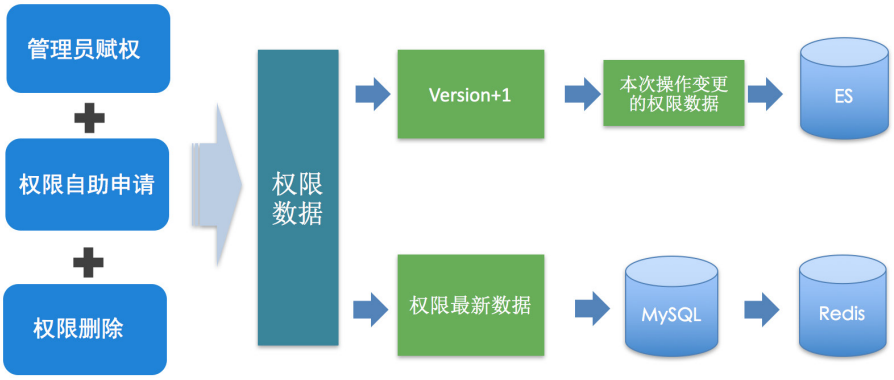


图 14　权限备份与恢复

如图 14 所示：

- 赋权操作时，通过版本号管理权限数据，每次操作后版本号加 1，MySQL 和 Redis 中只存储最新的权限数据。
- 历史权限数据通过版本号的方式存储到 Elasticsearch 中，每次查看历史操作记录或恢复权限数据时，根据版本号回溯即可。

权限过期清理

- 通过 Crane 定时调度，根据配置的通知规则，扫描即将过期的权限数据，发送消息通知用户进行权限续期。
- 扫描已过期的权限数据，清理 MySQL 和 Redis 中的过期权限数据，并转储到 Elasticsearch 中保存，已备后续的权限审计。

数据读写分离、缓存、备份以及服务熔断降级

各个服务使用 MySQL 分库存储，使用 Zebra（美团数据库访问层中间件）进行读写分离；合理使用数据缓存与备份，并支持服务的熔断降级，以保证服务的高可用。

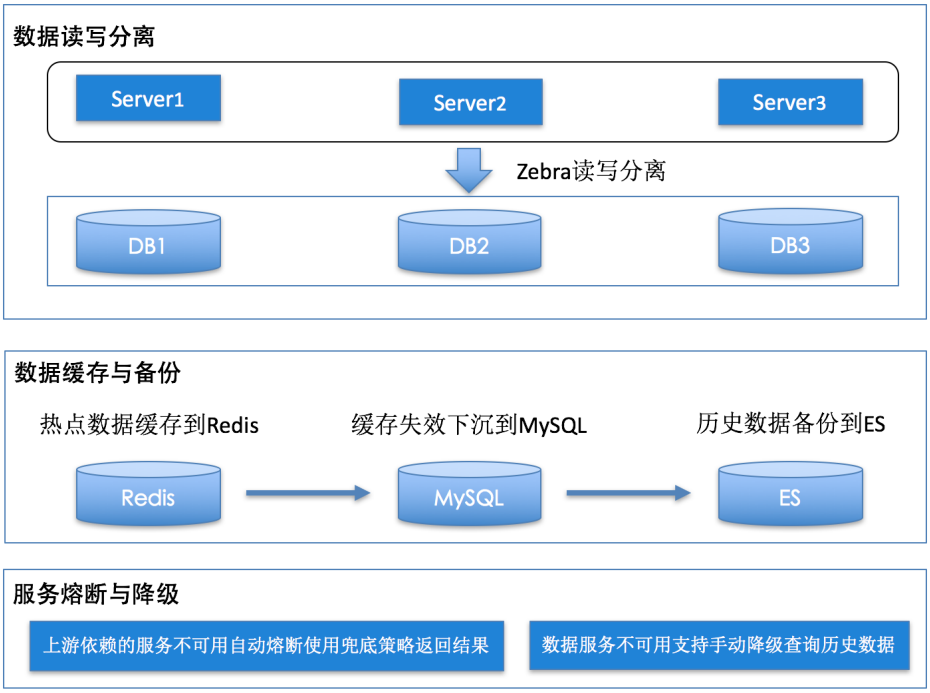


图 15　数据读写分离、缓存、备份以及服务熔断降级

如图 15 所示：

- 各个服务使用 MySQL 分库存储；核心服务与非核心服务分离，服务和数据库支持按需弹性拓展。
- 角色、资源等热点数据使用 Redis 做缓存，并在 Redis 缓存不可用时自动下沉到 MySQL 进行查询。
- 操作记录和历史数据等不活跃数据落地到 Elasticsearch，以便审计和数据恢复。
- 服务不可用时支持熔断降级，以保证核心服务的可用性。

合理使用消息队列、任务调度、线程池、分布式锁

使用消息队列、任务调度、线程池进行异步、削峰、解耦合，减少服务响应时间，提升用户体验。并使用分布式锁保证数据一致性。

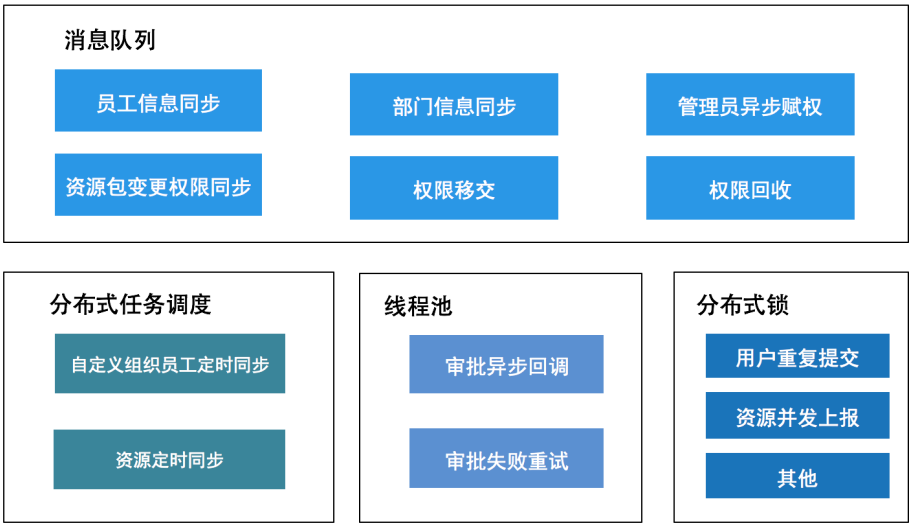


图 16　提高服务响应速度

如图 16 所示：

- 使用消息队列处理用户请求，实时返回操作成功，后台根据接受到的 MQ 消息异步进行处理并修改状态，页面轮询状态展示最终结果或发送大象（美团内部通讯工具）消息进行最终结果推送。
- 需要定时同步的任务通过 Crane 分布式任务调度平台进行定时调度执行。
- 审批回调时使用线程池处理审批结果回调与失败重试，较少创建销毁线程的开销。
- 分布式锁，保证同一个方法在同一操作上只能被一台机器上的一个线程执行，避免用户重复提交或者多机器重复处理导致的数据不一致。

展望

作为一个通用的数据安全平台，各个业务线的各种定制需求不可能都满足。目前在系统架构上已支持提供多个可插拔的 Plugin 服务，在通用服务的基础上实现定制的权限管控。后续将军令将针对权限、审批、审计提供 Plugin 开发规范，支持接入的系统在现有的基础上进行定制开发。

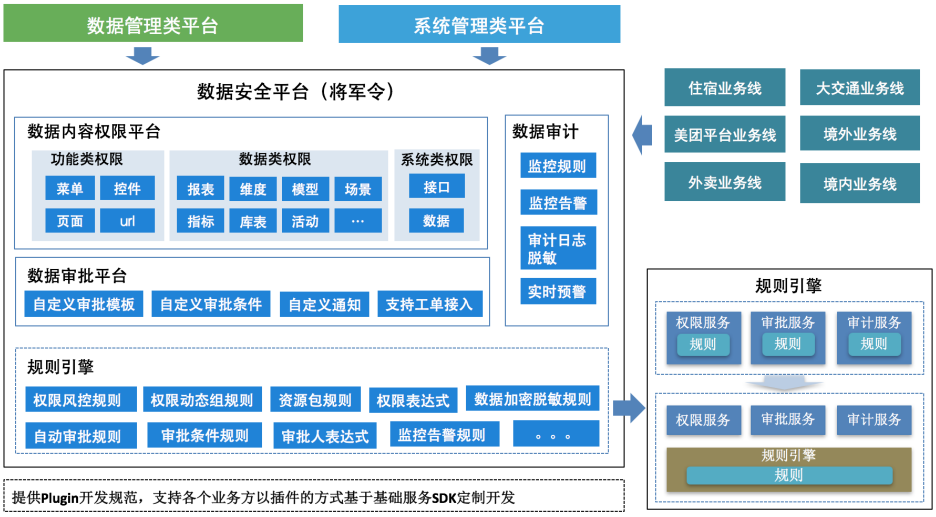


图 17　总体架构与展望

如图 17 所示：

- 后续将对外提供统一的 Plugin 开发规范，支持各个接入方系统以 Plugin 服务的形式在平台基础服务之上进行定制开发，以满足各自的特殊权限管控要求。从而实现数据产品权限集中管控确保数据安全。
- 把将军令中的规则从现有的服务中分离出来，抽象出一个通用的规则引擎服务，实现规则灵活可配置。

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。
中华，美团点评数据系统研发工程师，2017 年加入美团点评数据中心，长期从事于 BI 工具、数据安全相关工作。

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。

OneData 建设探索之路：SaaS 收银运营数仓建设

周成

背景

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

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

目标

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

OneData 探索

OneData：行业经验

在数据建设方面，阿里巴巴提出了一种 OneData 标准，如图 1 所示：

图 1　OneData 标准

OneData：我们的思考

他山之石，可以攻玉。我们结合实际情况和业界经验，进行了如下思考：

1. 对阿里巴巴 OneData 的思考

- 整个 OneData 体系覆盖范围广，包含数据规范定义体系、数据模型规范设计、ETL 规范研发以及支撑整个体系从方法到实施的工具体系。
- 实施周期较长，人力投入成本较高。
- 推广落地的工作比较依赖工具。

2. 对现有实际的思考

- 现阶段工具保障方面偏弱，人力比较缺乏。
- 现有开发流程不能全部推翻。

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

OneData：我们的想法

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

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

OneData 核心特点：三特性和三效果。

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

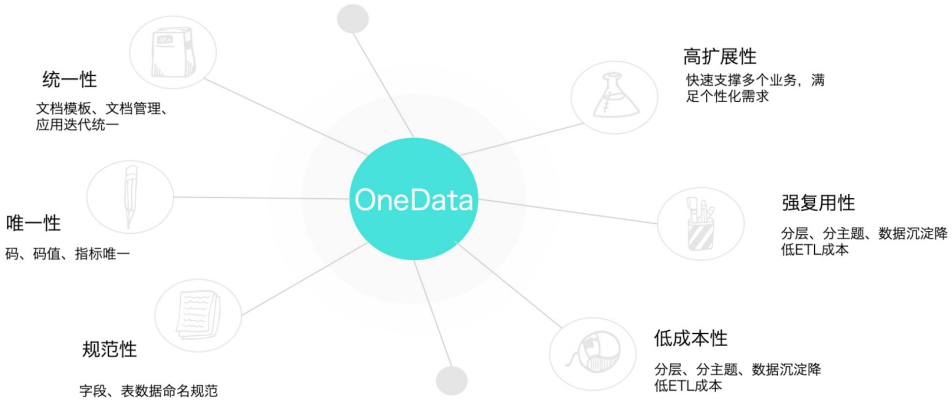
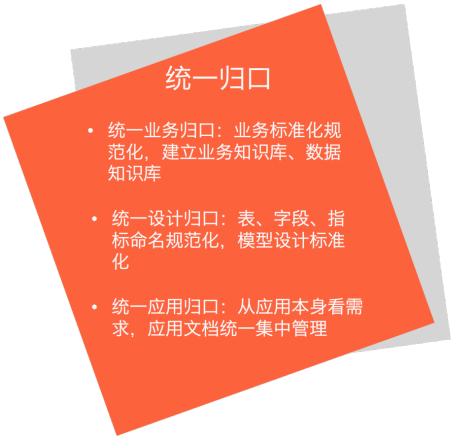
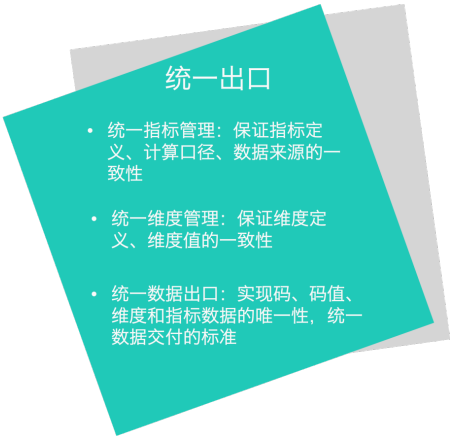


图 2　OneData 的六个特性

OneData：我们的策略

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





根据两个统一的方法策略，我们开启了 OneData 的实践之路。

OneData 实践

统一业务归口

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

图 3　支持业务的数据源知识库

设计统一归口

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

1. 模型

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

（1）模型分层

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

图 4　数据分层架构

（2）模型数据流向

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

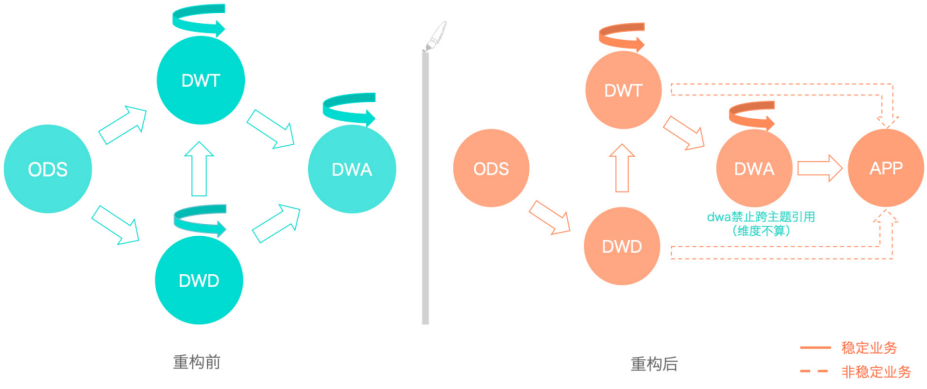


图 5　重构前和重构后的数据流向图

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

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

2. 主题划分

传统行业如银行、制造业、电信、零售等行业中，都有比较成熟的主题划分，如 BDWM、FS-LDM、MLDM 等等。但从实际调研情况来看，主题划分太抽象会造成对业务理解和开发成本较高，不适用互联网行业。因此，结合各层的特性，我们提出了两类主题的划分：**面向业务**、**面向分析**。

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

分析域、组织分析域。

3. 规范

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

（1）词根

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

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

（2）表命名规范

通用规范

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

表命名规则

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

存储层-ods	业务库表名			全量-不加 快照-Snapshot-ss 增量-Increment-inc 拉链、缓慢变化维-SlowlyChangingDimensions-scd	按小时更新-hourly 按天更新-daily 按周更新-weekly 按月更新-monthly 每年-yearly	[_his]
明细层-dwd	业务主题-m	二级主题简称	自定义表名[_(detail|ext)]			
主题宽表层-dwt						
轻度汇总层-dwa						
应用层-app						
维度表-dim			维度名[_(detail|ext)]			
临时表-temp	业务表名					序号:01-100
视图	业务表名					view
外部表	原表名					extnl

例如：dwt_m_ord_order_ss_daily

[]：可选项，可以省略
()：多选项，以 | 分隔，必须选填一项

图 6　统一的表命名规范

（3）指标命名规范

结合指标的特性以及词根管理规范，将指标进行结构化处理。

A. 基础指标词根，即所有指标必须包含以下基础词根：

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

B. 业务修饰词，用于描述业务场景的词汇，例如 trade- 交易。

C. 日期修饰词，用于修饰业务发生的时间区间。

日期类型	全称	词根	备注
日	daily	d	
周	weekly	w	
……	……	……	

D. 聚合修饰词，对结果进行聚集操作。

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

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

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

G. 普通指标命名规范，与字段命名规范一致，由词汇转换即可以。

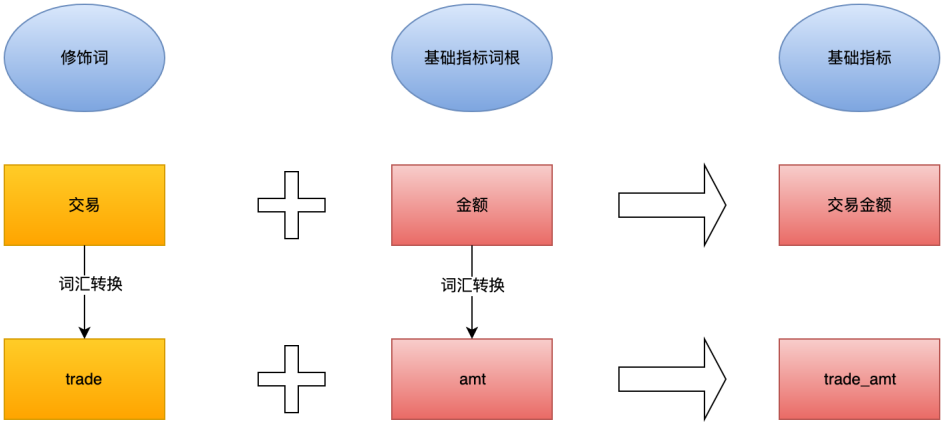


图 7　普通指标规范

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

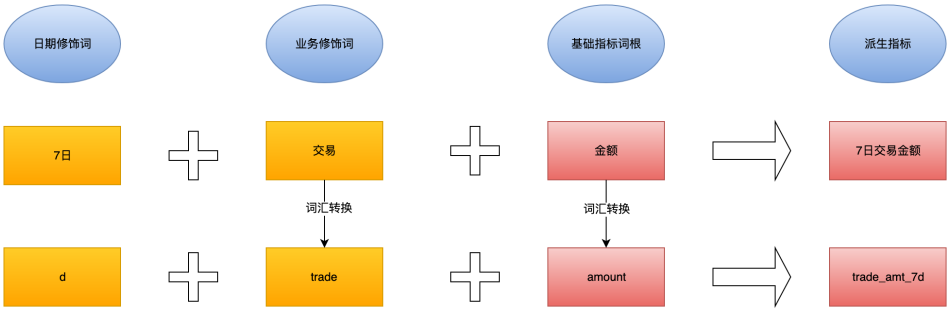


图 8　日期类型指标规范

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

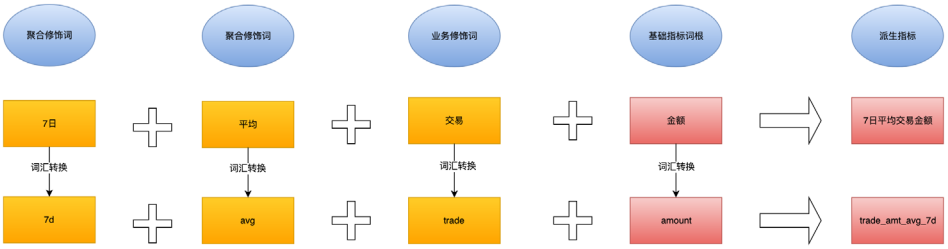


图 9　聚合类指标规范

（4）清洗规范

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

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

结合模型与规范，形成模型设计及模型评审两者的工作职责，如下图所示：

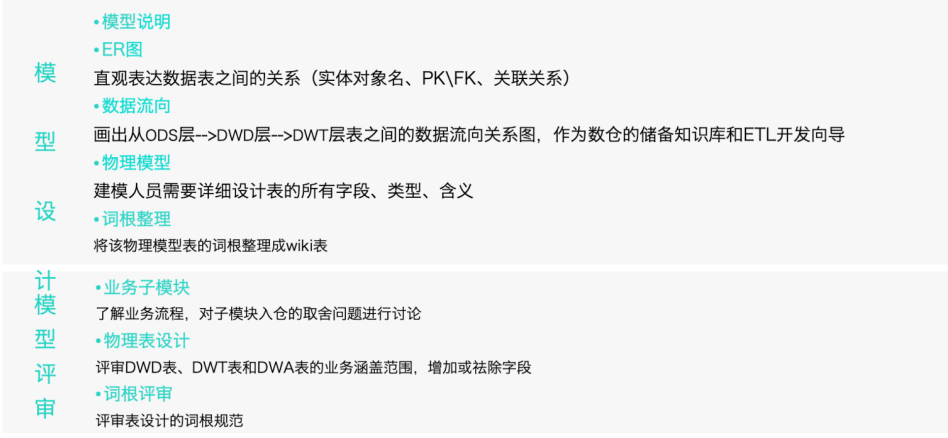


图 10　模型设计和审计职责

统一应用归口

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

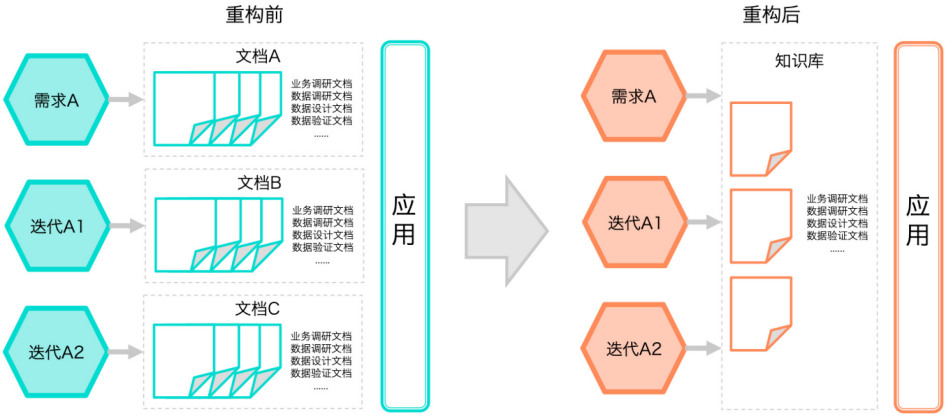


图 11　应用归口

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

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

统一数据出口

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

1. 交付标准化

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

图 12　交付标准化

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

2. 数据资产管理

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

图 13　起源功能体系

借用起源数据平台，我们实现了：

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

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

实践的成果

流程改善

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

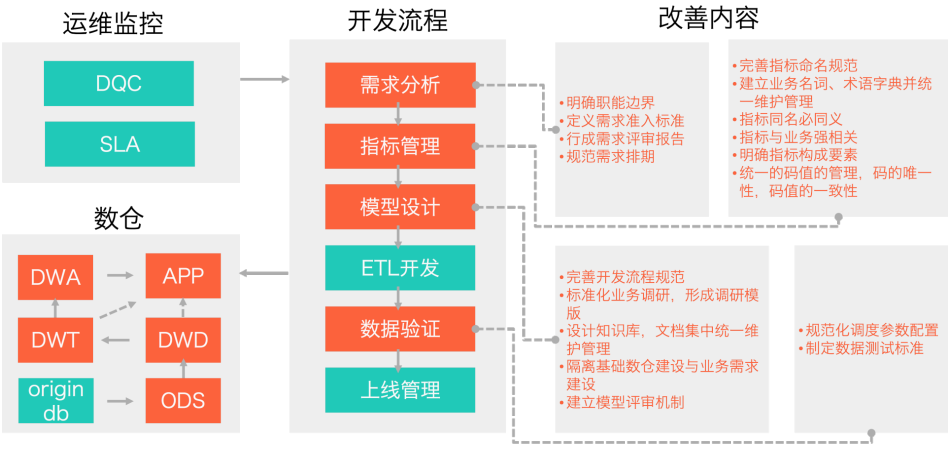


图 14　数仓管理流程

数仓全景图

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

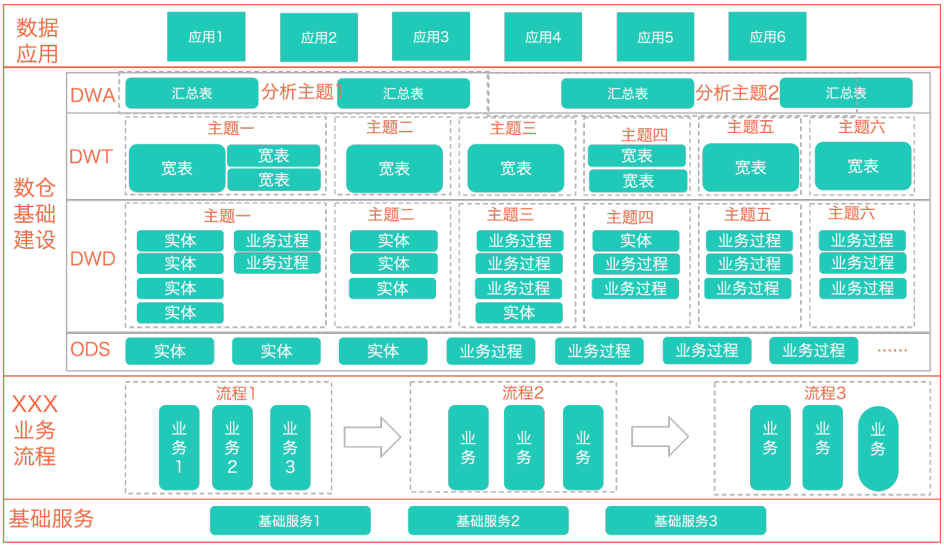


图 15　数仓全景图

资产管理列表

基于起源数据平台形成的资产管理体系，如下图所示：

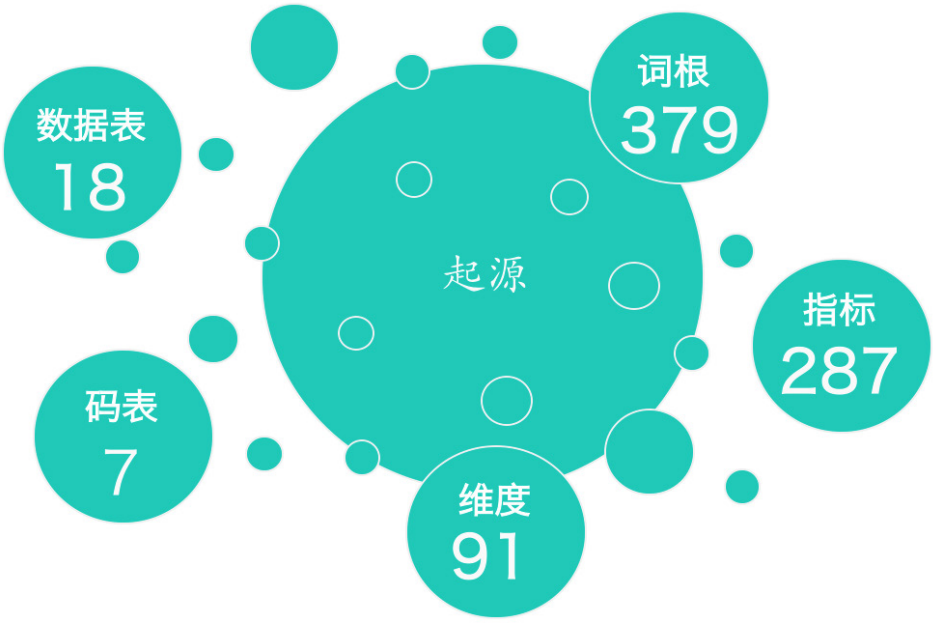


图 16　数据资产管理

项目收益

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

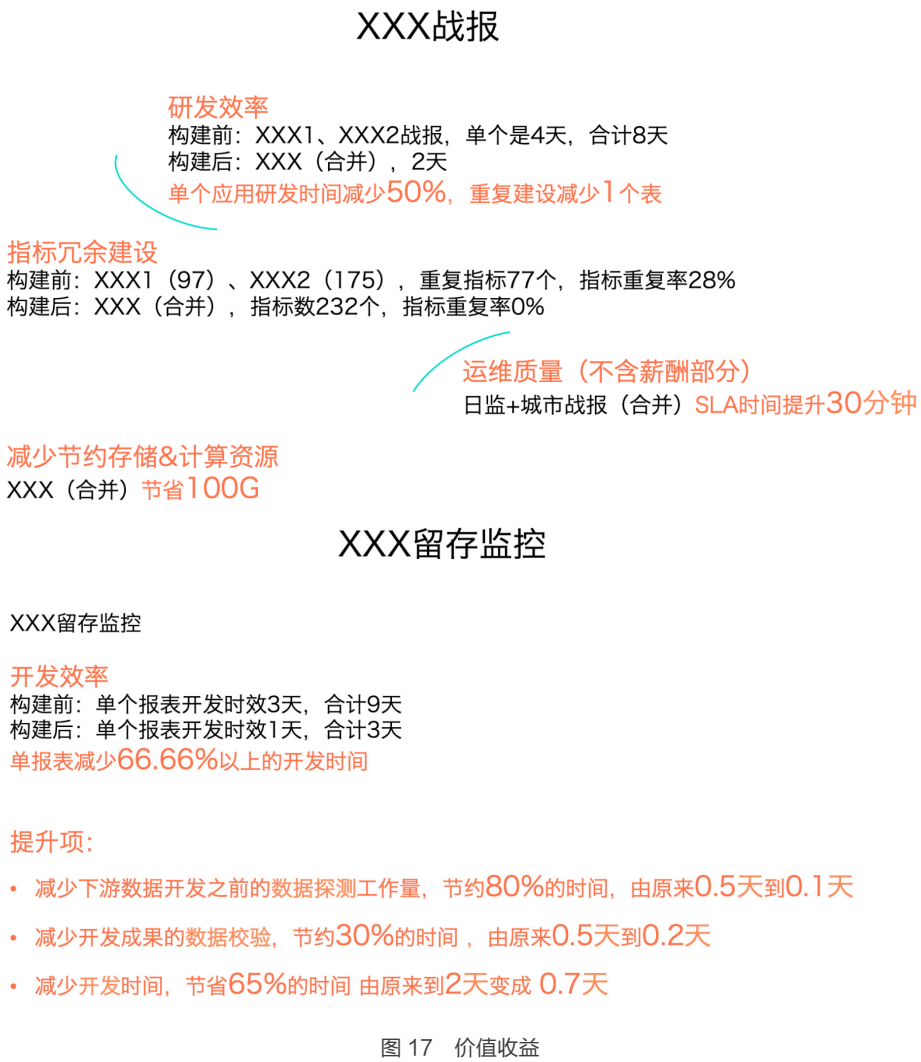


图 17　价值收益

总结和展望

我们结合了 OneData 核心思想与特点，构建一种稳定、可靠的基础数据仓库，从底层保障了数据质量，同时完成 OneData 实践，形成自有的 OneData 理论体系。未来，我们还将在技术上引入实时数据仓库，满足灵活多样、低延时的数据需求；在业务层面会横向拓展其他业务领域，不间断地支撑核心业务的决策与分析。下一步，我们将为企业级 One Entity 数据中台（以 Data As a Service 为理念），

提供强有力的数据支撑。在后续数仓维护过程中，不断地发现问题、解决问题和总结问题，保障数据稳定性、一致性和有效性，为核心业务构建价值链，最终形成企业级的数据资产。

作者

禄平，周成，黄浪，健平，高谦，美团数据研发工程师。

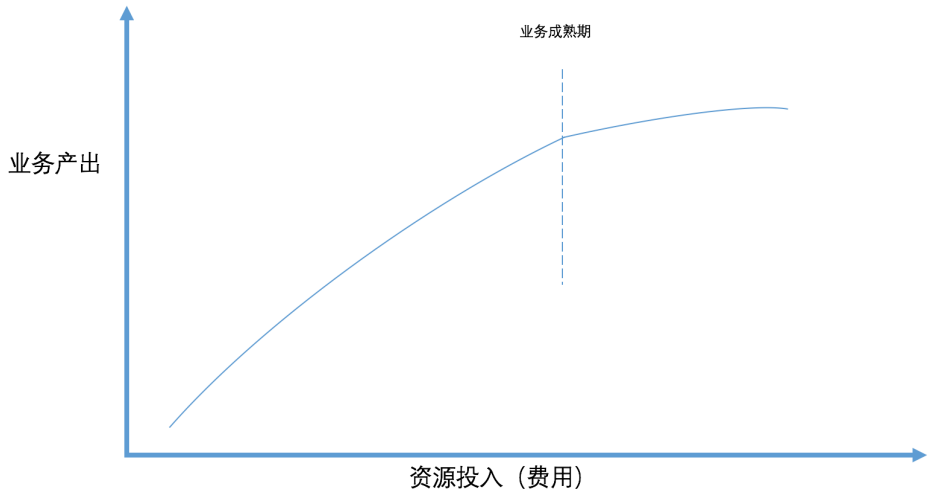
研发团队资源成本优化实践

刘强　建钟　小英　杨轩　云杰　方旭　鹏文

背景

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

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



从笔者十余年的工作经验来看，资源成本优化宜早不宜迟。很多管理者在业务发展的较前期阶段，可能并没有意识到资源成本膨胀所带来的压力。等到业务到了一定规模，拿到机器账单的时候，惊呼“机器怎么这么费钱”，再想立即降低成本，可能

已经错过了最佳时机，因为技术本身是一个相对长期的改造过程。

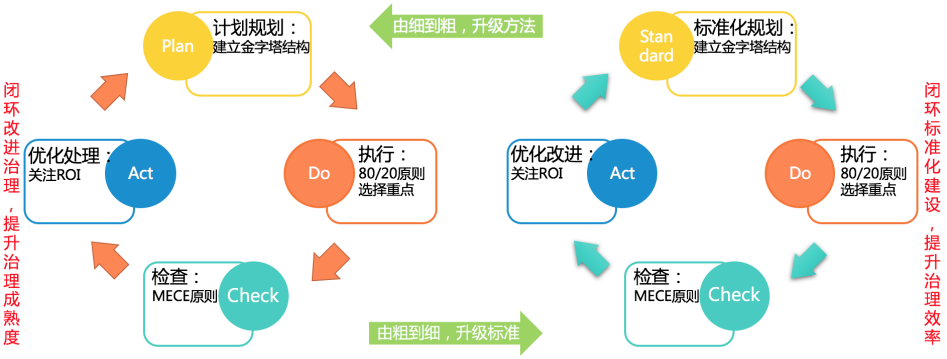
业务阶段	业务特点	成本管理意识
探索期	验证模式，从0到1	粗放式管理，只控制上限
进攻期	市场占有率是唯一目标	不需要控制成本，要多少给多少
发展期	稳居市场TOP2，业务成熟	看财报，发现设备这么花钱？
变革期	增速放缓，转型或变革	机器成本要控制了！

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

本文我们将分享美团到餐研发团队的资源成本优化实践。

实践

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



1. 确定方法论

“凡事预则立，不预则废”。在做一件事之前，要充分评估整个工作完整生命周期的要素，并进行整体工作框架的设计，一个科学的方法论是十分有必要的。成本优化

遵循的是一个行业内成熟的 P（Plan）D（Do）C（Check）A（Act）方法论，在每个阶段都又有对应的二次迭代和微循环。具体方法论如下：

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

2. 计划规划阶段（Plan&Standard）

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

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

- **建立意识**。这个是团队 Leader 的首要责任，要让团队成员明白自己在资源上花了多少钱，成本控制是不是一件真正有意义和价值的事，要做到大家认知一致。虽然见到过一些团队在提倡成本控制，但是落实到具体行动时，却流于形式或者无从下手，最后只能停留在口头上，并没有产生实际的效果。
- **确定目标**。这个过程相对宏观，也可以认为是“定性”的阶段。在这个阶段要明确的就是，在成本控制这件事上，后续动作要解决的问题是什么？比如有些团队是总体成本偏高，但有些团队总成本并不高，而是应该增加成本，有些团

队是非核心服务消耗的成本偏高，这些目标都需要经过团队成员讨论后得到一致的结果。在后续阶段的迭代中，也可以进行不断地修正。就像“客户永远不知道自己的需求”一样，很多人是不清楚自己的目标的，可以使用 SMART 原则来明确目标。

- **分析现状**。对成本这件事，罗列相关的数据，尽可能多地帮助自己做判断。自己团队在成本优化这件事上，处在哪一个阶段，哪些工作有可能被进一步优化，在此阶段要明确出来。
- **确定评价指标**。对于不同的专业序列，甚至对于同一专业序列的不同人员，大家对于成本的评价指标都不一样。这个阶段要做到最终的收敛，把团队未来成本优化的结果，用明确的数据表示出来。具体在到餐研发团队，我们确认了 2 个优化的核心指标：总成本、总订单成本。后续大家所有努力的目标，如果跟这两个指标没有关系或者弱相关，都可以忽略。

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

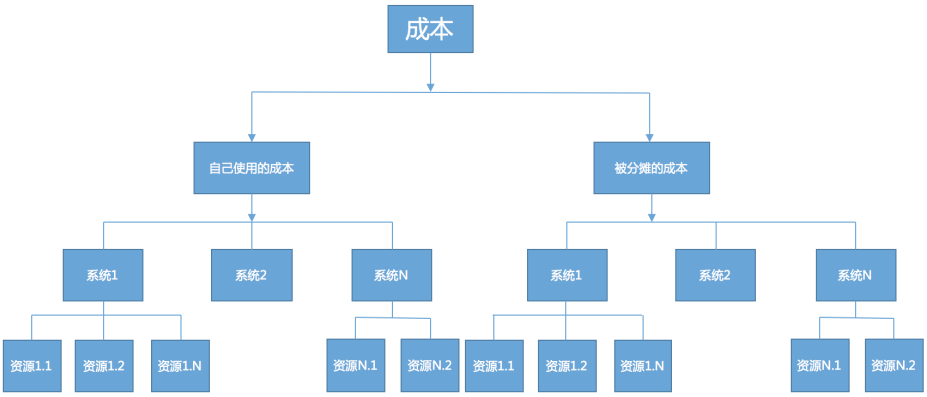
3. 执行阶段（Do）

3.1 建立思考流程

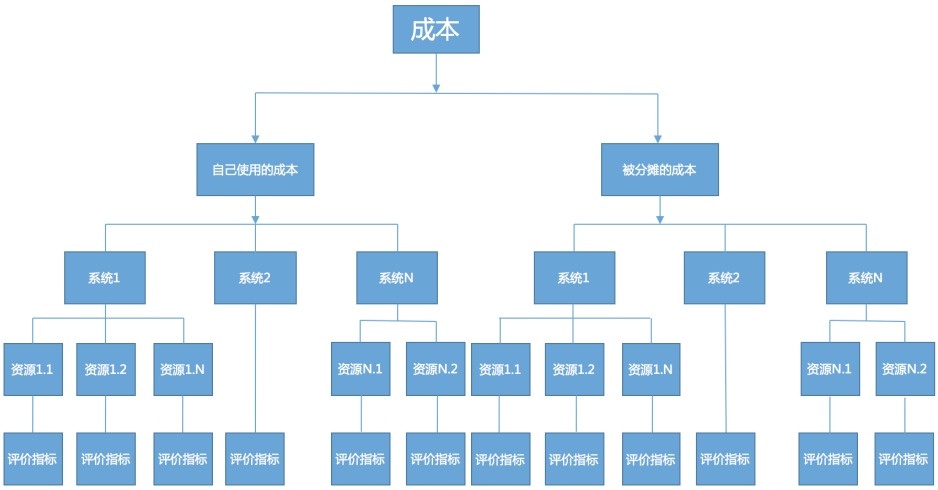
在执行阶段的流程是：分解原子项目 -> 确定方案 -> 落实到人 -> 优化原子指标。在这里包括两个核心要素：1）把核心指标相关的工作向下一层分解；2）在下一层，找到具体的人来执行，这个人要具备将自己负责的指标继续分解到更细的能力，类似于我们说的树状结构。这样层层地分解下去，每一层的叶子节点都可以找到对应的负责人。这种“总分”结构，在一本经典教程《金字塔原理》中也有详细的阐述。

- **分解原子项目**。在本阶段要建立一个完全细化的分级结构，用金字塔原理中的”MECE 不重不漏”原则，将工作内容分解到最细的可控粒度。至于按哪个

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



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

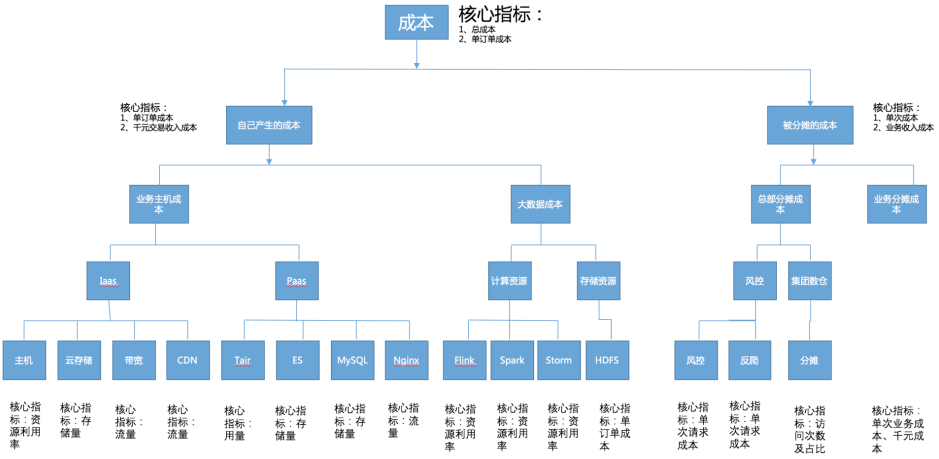


- **确定方案**。根据上面的金字塔结构，每个原子指标，都需要专业的同学来评价分析，确定如何进行优化。比如，系统主机的成本，主要集中在虚拟机 + 存储这样的资源上，衡量的指标可以确定为“资源利用率”和“单订单成本”，为了解决“资源利用率”这个原子指标，就需要考虑目前的空闲机器是否可以下线，在线的服务是否可以优化或者合并；为了解决“单订单成本”这个指标，可以考虑分析下系统架构，跟核心流程处理有关的服务是否可以更加高效或者抽象出来成为服务中台，这样就可以释放一些”烟囱式”的建设资源，使得核心处理能力更加集中、高效。类似这样将所有的解决方案整合起来，就形成了最后的解决方案。
- **落实到人**。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

的产出。对于没有解决方案的资源或者实施难度过大的资源，建议果断放弃或者搁置。

3.2 实践分析框架

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



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

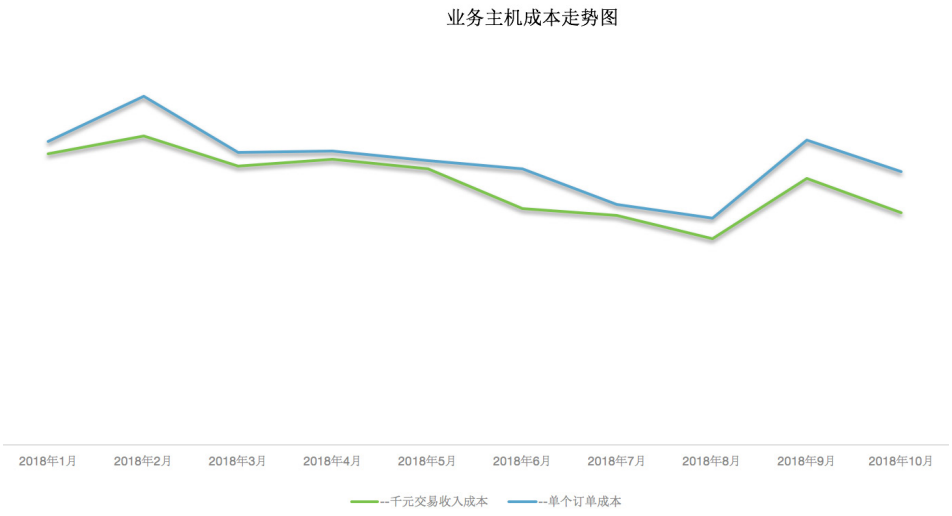
很多公司每个技术团队的机器成本，在财务上叫做“网站运维成本”（网站？听起来还像 PC 时代的概念对不对），从顶层可以分为两类构成因素，就是“自己产生的成本”（自己用的）和“被分摊的成本”（别人替你用的）两大类。跟自己有关的继续向下钻取，可以分为交易相关的资源成本（跟业务流程相关的）以及跟分析有关的大数据成本（分析、算法、决策相关）。

3.2.1 业务主机成本

大部分业务系统的团队，使用的资源成本都包含在这个部分，比如商户研发团队、订单系统研发团队、前端研发团队、供应链研发团队、营销系统研发团队、

CRM 研发团队等。这些资源典型的物理载体就是物理机、虚拟机、容器资源以及对应的机器连接的存储（DB、缓存、K-V 数据库等）资源，还会包含由于交换、存储以上资源之间的数据产生的带宽、云资源、CDN 等。

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



3.2.2 大数据成本

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

第一部分，是存储资源的消耗，行业通用的模型是基于物理 HDFS 或自研的类

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

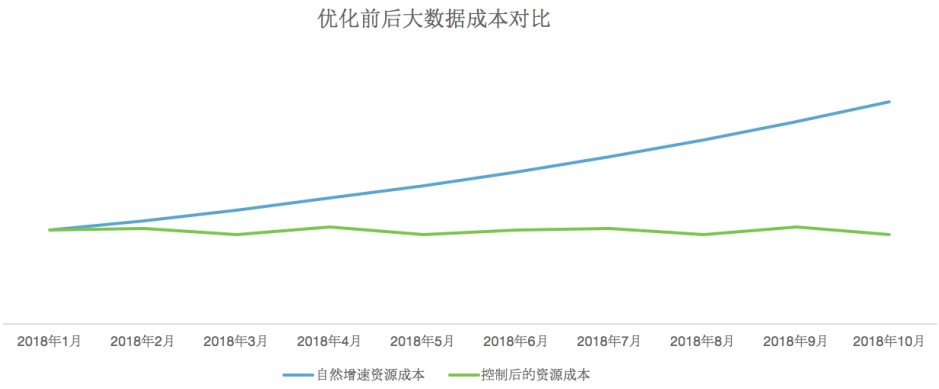
在此，到餐研发团队主要采取了两种手段：

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

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

另外需要补充的，还有一部分 OLAP 存储资源，也会消耗大量资源，比如 Kylin、Elasticsearch、Druid、MySQL 等，这些数据库主要用来将基于 HDFS 上的文件，同步到前端可以直接访问的介质上，供系统访问。这部分资源有些也是基于 HDFS 的（如 Kylin、HBase），有些需要单独的存储介质，也需要关注其膨胀速度以及存储周期。

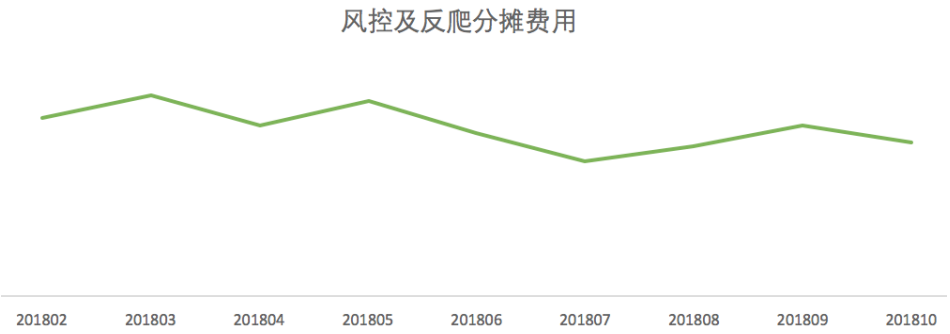
第二部分，是计算资源的消耗，主要满足基于复杂规则的分析或者机器学习算法中的计算，也就是实时 ETL 计算和离线 ETL 计算的场景（代表性的引擎如 Storm、Flink 的计算还有 MapReduce 的计算）。这部分计算消耗的资源类似于业务系统，可以参照业务系统的“资源利用率”确定几个指标，进行机器优化或者算法逻辑优化。



3.2.3 分摊成本（一）风控及反爬

在某些公司里，某个技术团队开发的内容，有可能为了服务其他团队业务，比如前文中提到的风控、反爬、广告等，会为各种业务提供基础的技术能力。这时候就涉及到一个重要的概念“分摊”。分摊有两种规则，一种是按“实际用量进行”，另外一种是按照“使用比例”进行，这两种模式之上，可能还有混合计费模式，即“按照实际发生的比例进行整体费用的分摊”，做成本控制时，就要清楚地知道这部分成本是按哪种逻辑来进行计算的。

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

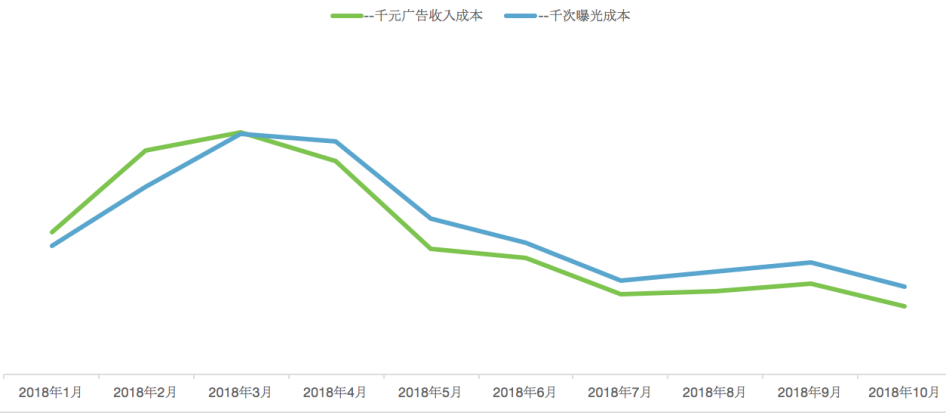


3.2.4 分摊成本（二）安全数仓成本

为了保证各个业务团队之间的离线数据交换，美团集团层面建设了安全数据仓库，用来满足跨团队之间的数据交换。这部分的费用也按照实际发生的资源占比进行统计，所以同理，为了降低成本，需要关注两个组成项目：一是自己使用的数量，从架构设计上能否将相关数据模型的效率提升、降低空间是关键因素；二是自己的使用资源在整体资源的占比，这时候也需要跟相关团队一起努力降低总成本。很多公司的技术团队，也有类似的数据共享仓库或者共建仓库的概念。

3.2.5 分摊成本（三）广告成本

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



3.2.6 其他成本

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

4. 检查（Check）

在这个阶段，建议关注以下结果：

- **规定动作检查**。规定的方案是否执行？相关的同学是否按照规定的动作进行了相对应的行动？这个阶段只关注过程不关注结果，而且更多的是关注执行人、配合方、时间点，用项目管理的思路来运营。
- **结果评估**。之前梳理出来的指标是否得到了优化？这个过程是在验证结果，各项指标中得到优化和未优化的都要整理出详细的 List，有些指标如“资源利用率”是立即可以查看结果的，有些结果是需要周期性的时间才能获得。在这个基础上可以继续深入反向思考，按“指标定义是否有问题 -> 方案制定是否有问题 -> 执行人是否有问题 -> 配合方是否有问题”这个流程来进行评估。
- **系统问题定位**。在这个过程中，可以做到小范围闭环，建议针对某个指标的优化方案可以设计多套，方案 A 不行马上迭代成方案 B，快速试错，找到合理的方案。

- **修正标准动作**。在执行的过程中，很多方案和动作，都是在一线现场发现和修正的，不需要等待大规模复盘的时候再提出问题和总结，主 R 要具备这样的意识，在执行过程中多说多问，找到关键要素，相信每个同学都有过这样的经历。经历过某个完整项目生命周期的同学，往往也是团队内成长最快的骨干。

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

5. 复盘总结，继续迭代 (Act)

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

的问题，另外还收获了 7 项跟架构升级、数据压缩、技术处理有关技术专利，也是技术能力提升的一个佐证。

总结

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

作者简介

刘强，美团到店餐饮研发中心数据方向负责人，美团数据技术通道委员，2017 年入职美团点评，就职于到店餐饮研发中心，负责到餐数据仓库、数据产品、数据系统的研发工作。之前曾任多家公司的数据方向负责人。
建钟、小英、杨轩、云杰、方旭、鹏文，均为美团到餐研发团队工程师，对本文均有贡献。

特别感谢

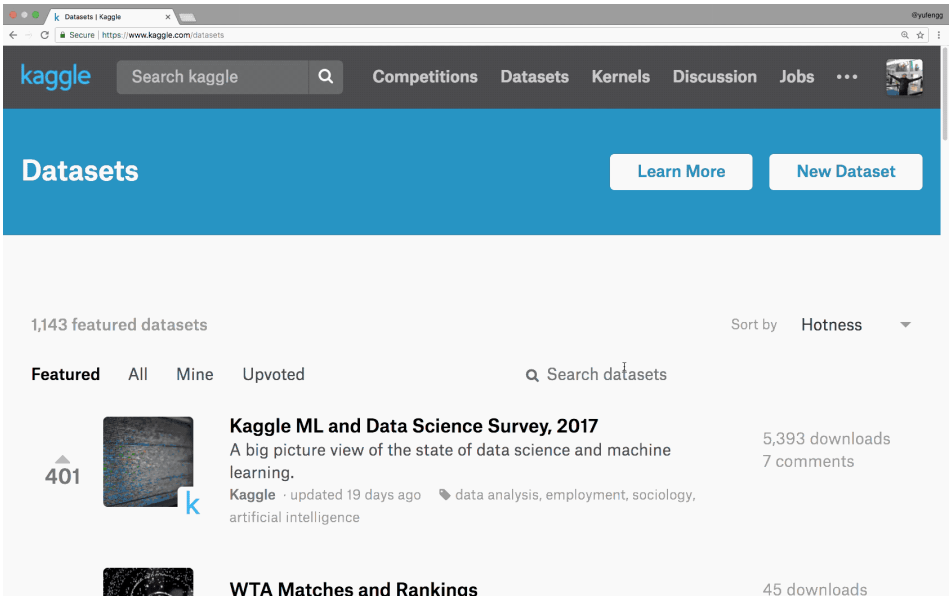
在本文及成本优化过程中，得到了美团技术团队李伟、任登君、李闻、谢语宸、洪丹、左普存、郭树熠、刁士涵等人的支持和帮助，在此表示感谢！

Jupyter 在美团民宿的应用实践

文龙　颖艺

前言

做算法的同学对于 Kaggle 应该都不陌生，除了举办算法挑战赛以外，它还提供了一个学习、练习数据分析和算法开发的平台。Kaggle 提供了 Kaggle Kernels，方便用户进行数据分析以及经验分享。在 Kaggle Kernels 中，你可以 Fork 别人分享的结果进行复现或者进一步分析，也可以新建一个 Kernel 进行数据分析和算法开发。Kaggle Kernels 还提供了一个配置好的环境，以及比赛的数据集，帮你从配置本地环境中解放出来。Kaggle Kernels 提供给你的是一个运行在浏览器中的 Jupyter，你可以在上面进行交互式的执行代码、探索数据、训练模型等等。更多关于 Kaggle Kernels 的使用方法可以参考 Introduction to Kaggle Kernels，这里不再多做阐述。



对于比赛类的任务，使用 Kaggle Kernels 非常方便，但我们平时的主要任务还

是集中在分析、处理业务数据的层面，这些数据通常比较机密并且数量巨大，所以就不能在 Kaggle Kernels 上进行此类分析。因此，大型的互联网公司非常有必要开发并维护集团内部的一套「Kaggle Kernels」服务，从而有效地提升算法同学的日常开发效率。

本文我们将分享美团民宿团队是如何搭建自己的「Kaggle Kernels」—— 一个平台化的 Jupyter，接入了大数据和分布式计算集群，用于业务数据分析和算法开发。希望能为有同样需求的读者带来一些启发。

美团内部数据系统现状

现有系统与问题

算法同学在离线阶段主要包含三类任务：数据分析、数据生产、模型训练。为满足这些任务的要求，美团内部也开发了相应的系统：

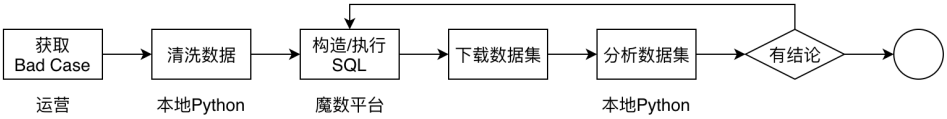
- 魔数平台：用于执行 SQL 查询，下载结果集的系统。通常在数据分析阶段使用。
- 协同平台：用于使用 SQL 开发 ETL 的平台。通常用于数据生产。
- 托管平台：用于管理和运行 Spark 任务，用户提供任务的代码仓库，系统管理和运行任务。通常用于逻辑较复杂的 ETL、基于 Spark 的离线模型训练 / 预测任务等。
- 调度平台：用于管理任务的依赖关系，周期性按依赖执行调度任务。

这些系统对于确定的任务完成的比较好。例如：当取数任务确定时，适合在魔数平台执行查询；当 Spark 任务开发就绪后，适合在托管平台托管该任务。但对于探索性、分析性的任务没有比较好的工具支持。探索性的任务有程序开发时的调试和对陌生数据的探查，分析性的任务有特征分析、Bad Case 分析等等。

以数据探索为例，我们经常需要对数据进行统计与可视化，现有的做法通常是：魔数执行 SQL -> 下载 Excel -> 可视化。这种方式存在的问题是：

- 分析和取数工具割裂。
- 大数据分析可视化困难。

以 Bad Case 分析为例，现有的做法通常是：



这种方式存在的问题是：

- 分析与取数割裂，整个过程需要较多的手工操作。
- 分析过程不容易复现，对于多人协作式的验证以及进一步分析不利。
- 本地 Python 环境可能与分析对象的依赖有冲突，需要付出额外精力管理 Python 环境。

离线数据相关任务的模式通常是取数（小数据 / 大数据）-> Python 处理（单机 / 分布式）-> 查看结果（表格 / 可视化）这样的循环。我们希望支持这一类任务的工具具有如下特质：

- 体验流畅：数据任务可以在统一的工具中完成，或者在可组合的工具链中完成。
- 体验一致：数据任务所用工具应该是一致的，不需要根据任务切换不同工具。
- 使用便捷：工具应是开箱即用，不需要繁琐的前置配置。
- 结果可复现：分析过程能够作为可执行代码保存下来，需要复现时执行即可，也应支持修改。

探索和分析类任务往往会带来可以沉淀的结果，如产生新的特征、模型、例行报告，希望可以建立起分析任务和调度任务的桥梁。

我们需要怎样的 Jupyter

参考 Kaggle Kernels 的体验和开源 Jupyter 的功能，Notebook 方式进行探索分析具有良好的体验。我们计划定制 Jupyter，使其成为完成数据任务的统一工具。

这个定制的 Jupyter 应具备以下功能：

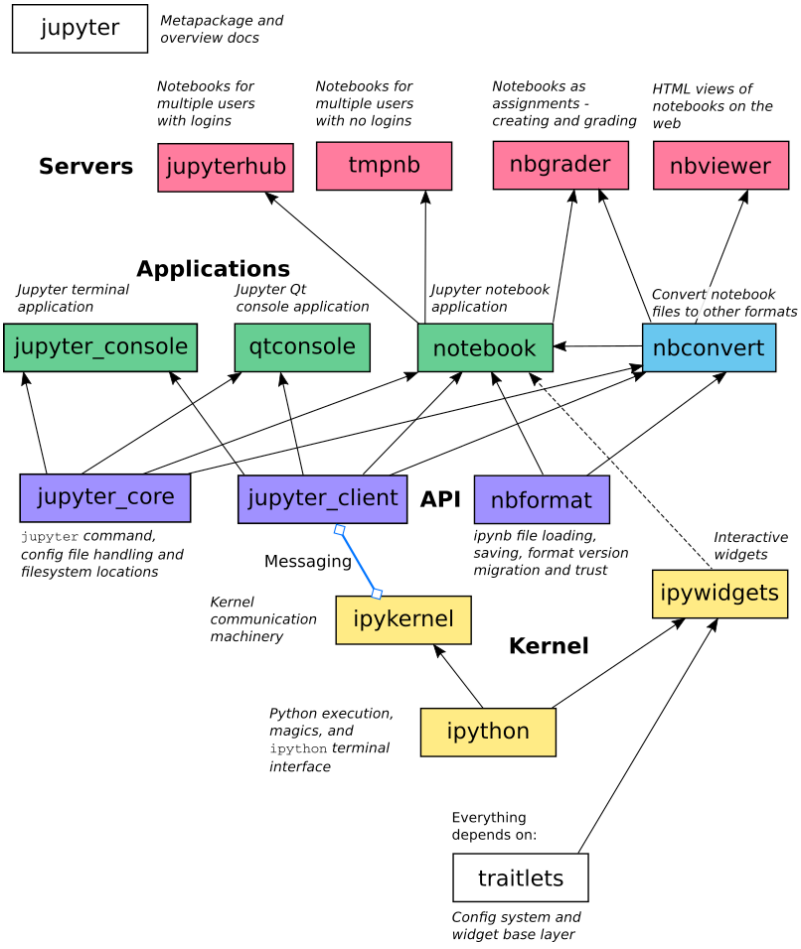
- 接入 Spark：取数与分析均在 Jupyter 中完成，达到流畅、一致的体验。

- 接入调度系统：方便沉淀分析结果。
- 接入学城系统（内部 WiKi）：方便分享和复现。
- 预配置环境：提供给用户开箱即用的环境。
- 用户隔离环境：避免用户间互相污染环境。

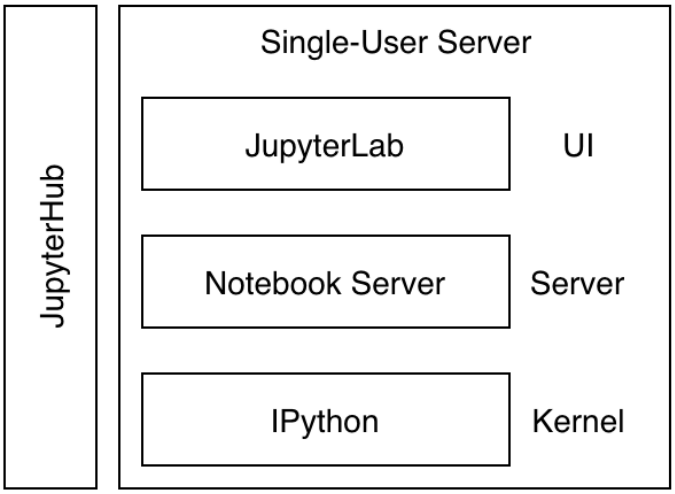
如何搭建 Jupyter 平台

Jupyter 项目架构

Project Jupyter 由多个子项目组成，通过这些子项目可以自由组合出不同的应用。子项目的依赖关系如下图所示：



这个案例中，Jupyter 应用是一个 Web 服务，我们可以从这个维度来看 Jupyter 架构：

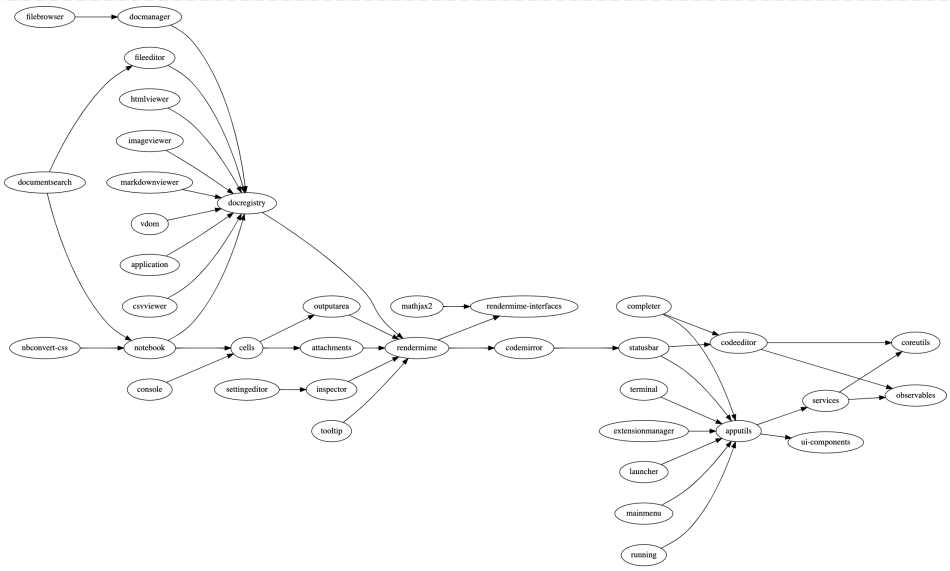


Jupyter 扩展方式

整个 Jupyter 项目的模块化和扩展性上都非常优秀。上图中的 JupyterLab、Notebook Server、IPython、JupyterHub 都是可扩展的。

JupyterLab 扩展（labextension）

JupyterLab 是 Jupyter 全新的前端项目，这个项目有非常明确的扩展规范以及丰富的扩展方式。通过开发 JupyterLab 扩展，可以为前端界面增加新功能，例如新的文件类型打开 / 编辑支持、Notebook 工具栏增加新的按钮、菜单栏增加新的菜单项等等。JupyterLab 上的前端模块具有非常清楚的定义和文档，每个模块都可以通过插件获取，进行方法调用，获取必要的信息以及执行必要的动作。我们在提供分享功能、调度功能时，均开发了 JupyterLab 扩展。JupyterLab 扩展通常采用 TypeScript 开发，开发文档可参考：https://jupyterlab.readthedocs.io/en/stable/developer/extension_dev.html。



JupyterLab 核心组件依赖图

Notebook Server 扩展（serverextension）

Notebook Server 是用 Python 写的一个基于 Tornado 的 Web 服务。通过 Notebook Server 扩展，可以为这个 Web 服务增加新的 Handler。增加新的 Handler 通常有两种用途：

1. 为 JupyterLab 扩展提供对应的后端接口，用于响应一些需要由服务端处理的事件。例如调度任务的注册需要通过 JupyterLab 扩展发起请求，由 Notebook Server 扩展执行。
2. 提供一个前端界面以及对应的后端处理服务。例如 jupyter-rsession-proxy，用于在 JupyterHub 中使用 RStudio。

Notebook Server 扩展开发文档可参考：https://jupyter-notebook.readthedocs.io/en/stable/extending/handlers.html。

##3# Jupyter Kernels

Jupyter 用于执行代码的模块叫 Kernel，除了默认的 ipykernel 以外，还可以有

其他的 Kernel 用于支持其他编程语言。例如支持 Scala 语言的 almond、支持 R 语言的 irkernel，更多详见语言支持列表。

IPython Magics

IPython Magics 就是那些 %、%% 开头的命令。常见的 Magics 有 %matplotlib inline，设置 Notebook 中调用 matplotlib 的绘图函数时，直接展示图表在 Notebook 中。执行 Magics 时，事实上是调用了该 Magics 定义的一个函数。对于 Line Magics（一个 %），传入函数的是当前行的代码；对于 Cell Magics（两个 %），传入的是整个 Cell 的内容。定义一个新的 IPython Magics 仅需定义一个函数，这个函数的入参有两个，一个是当前会话实例，可以用来遍历当前会话的所有变量，可以为当前会话增加新的变量；另一个是用户输入，对于 Line Magics 是当前行，对于 Cell Magcis 是当前 Cell。

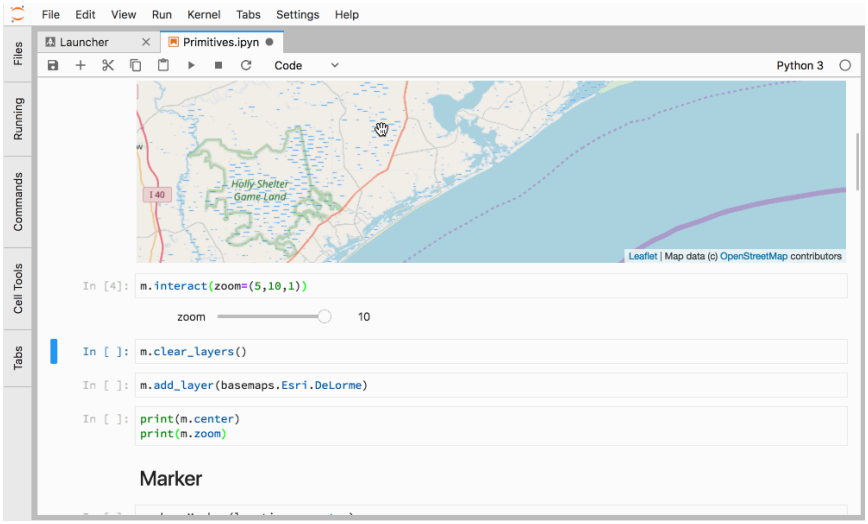
IPython Magics 在简化代码方面非常有效，我们开发了 %%spark、%%sql 用于创建 Spark 会话以及 SQL 查询。另外很多第三方的 Magics 可以用来提高我们的开发效率，例如在开发 Word2Vec 变种时，使用 %%cython 来进行 Cython 和 Python 混合编程，省去编译加载模块的工作。

IPython Magics 开发文档可参考：https://ipython.readthedocs.io/en/stable/config/custommagics.html。

IPython Widgets（ipywidgets）

IPython Widgets 是一种基于 Jupyter Notebook 和 IPython 的可交互控件。与普通可视化不同的是，在控件上的交互会触发和 Python 的通信并执行相应的代码，Python 上相应的动作也会触发界面实时变化。

IPython Widgets 在提供工具类型的功能增强上非常有用，基于它，我们实现了一个线上排序服务的调试和复现工具，用于展示排序结果以及指定房源在排序过程中的各种特征以及中间变量的值。IPython Widgets 的开发可以通过组合现有的 Widgets 实现，也可以完全自定义一个，IPython Widgets 开发文档可参考：https://ipywidgets.readthedocs.io/en/stable/examples/Widget%20Custom.html。



ipyleaflet

扩展 JupyterHub

Authenticators

JupyterHub 是一个多用户系统，登录模块可替换，通过实现新的 Authenticator 类并在配置文件中指定即可。通过这个扩展点，我们实现了使用内部 SSO 系统登录 JupyterHub。Authenticator 开发文档可参考：https://jupyterhub.readthedocs.io/en/stable/reference/authenticators.html。

Spawners

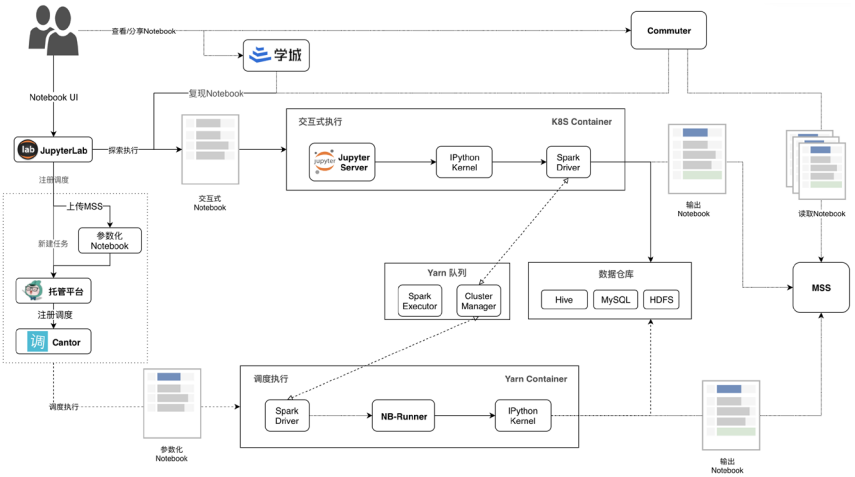
当用户登录时，JupyterHub 需要为用户启动一个用户专用 Notebook Server。启动这个 Notebook Server 有多种方式：本机新的 Notebook Server 进程、本机启动 Docker 实例、K8s 系统中启动新的 Pod、YARN 中启动新的实例等等。每一种启动方式都对应一个 Spawner，官方提供了多种 Spawner 的实现，这些实现本身是可配置的。如果不符合需求，也可以自己开发全新的 Spawner。由于我们需要实现 Spark 接入，对 K8s 的 Pod 有新的要求，所以基于 KubeSpawner 定制了一个 Spawner 来解决 Spark 连接集群的网络问题。Spawner 开发文档可参考：https://jupyterhub.readthedocs.io/en/stable/reference/spawners.html。

我们的定制

回顾我们的需求，这个定制的 Jupyter 应具备以下功能：

- 接入 Spark：可以通过配置容器环境以及 Spawner 完成。
- 接入调度系统：需要开发 JupyterLab 扩展以及 Notebook Server 扩展。
- 接入学城系统：需要开发 JupyterLab 扩展以及 Notebook Server 扩展。
- 预配置环境：镜像配置。
- 用户隔离环境：通过定制 Authenticators + K8s Spawner 实现容器级别环境隔离。

我们的方案是基于 JupyterHub on K8s。下图是平台化 Jupyter 的架构图，从上到下可以看到三条主线：1. 分享复现、2. 探索执行、3. 调度执行。



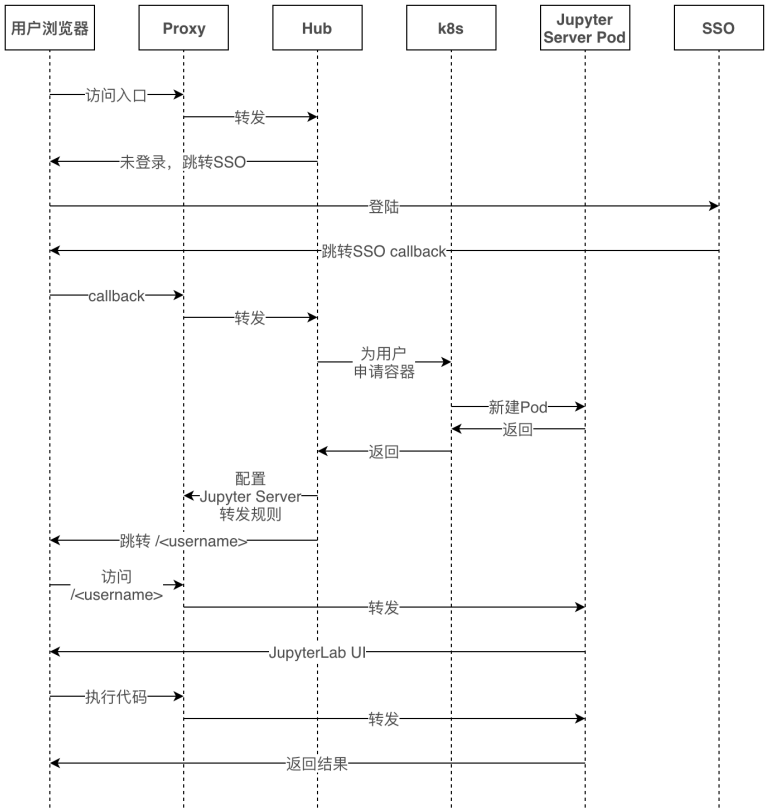
几个关键组件介绍：

- JupyterLab：交互式执行的前端，开源项目。
- Jupyter Server：交互式执行的后端，开源项目。
- Commuter：浏览 Notebook 的工具，开源项目。
- K8s：容器编排系统，开源项目。
- Cantor：美团调度系统，同类开源项目有 AirFlow。

- 托管平台：美团离线任务托管平台，给定代码仓库和任务参数，为我们执行 Spark-Submit 的平台。
- 学城：美团文档系统。
- MSS：美团对象存储。
- NB-Runner：Notebook Runner，在 nbconvert 的基础上增加了参数化和 Spark 支持。

在定制 Jupyter 中，最为关键的两个是接入 Spark 以及接入调度系统，下文中将详细介绍这两部分的原理。

JupyterHub on K8s 包括几个重要组成部分：Proxy、Hub、Kubernetes、用户容器（Jupyter Server Pod）、单点登录系统（SSO）。一个用户在登录后新建容器实例的过程中，这几个模块的交互如下图所示：



可以看到，新建容器实例后，用户的交互都是经过 Proxy 后与 Jupyter Server Pod 进行通信。因此，扩展功能的工作主要是定制 Jupyter Server Pod 对应的容器镜像。

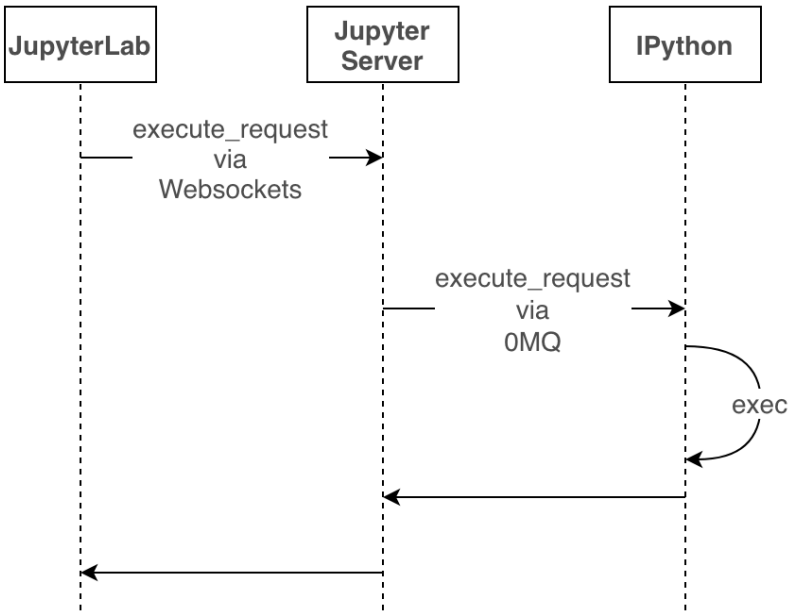
让 Jupyter 支持 Spark

Jupyter 平台化后，我们得到一个接近 Kaggle Kernel 的环境，但是还不能够使用大数据集群。接下来，就是让 Jupyter 支持 Spark，Jupyter 支持 Spark 的方案有 Toree，出于灵活性考虑，我们没有使用。我们希望让普通的 Python Kernel 能支持 PySpark。

为了能让 Jupyter 支持 Spark，我们需要了解两方面原理：Jupyter 代码执行原理和 PySpark 原理。

#3## Jupyter 代码执行原理

所用到的 Jupyter 分三部分：前端 JupyterLab、服务端 Jupyter Server、语言 Kernel IPython。这三个模块的通信如下图所示：



Jupyter 执行代码时序图

这里，需要在 IPython 的 exec 阶段支持 PySpark。

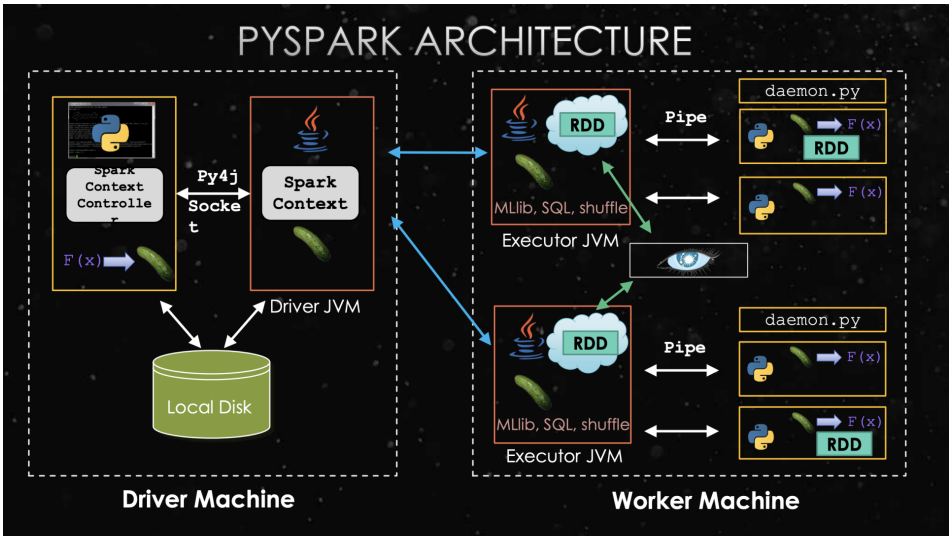
##3# PySpark 原理

启动 PySpark 有两种方式：

- 方案一：PySpark 命令启动，内部执行了 spark-submit 命令。
- 方案二：任意 Python shell（Python、IPython）中执行 Spark 会话创建语句。

这两种启动方式有什么区别呢?

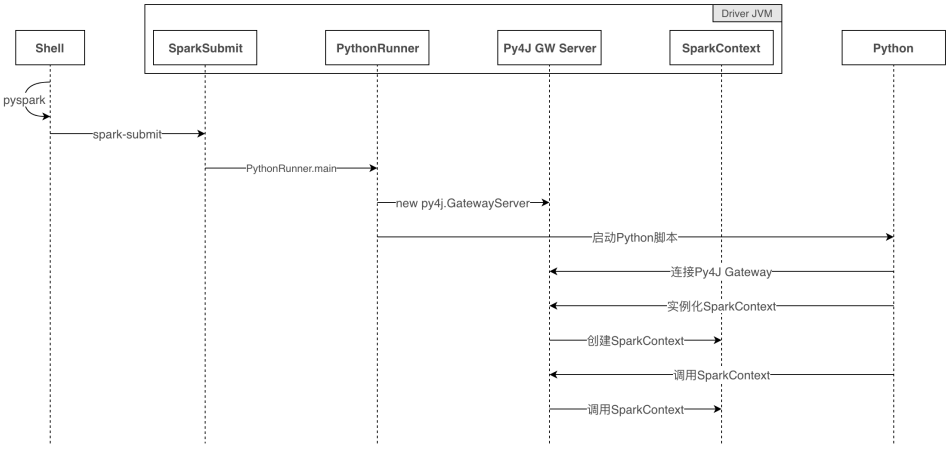
看一下 PySpark 架构图：



PySpark 架构图，来自 SlideShare

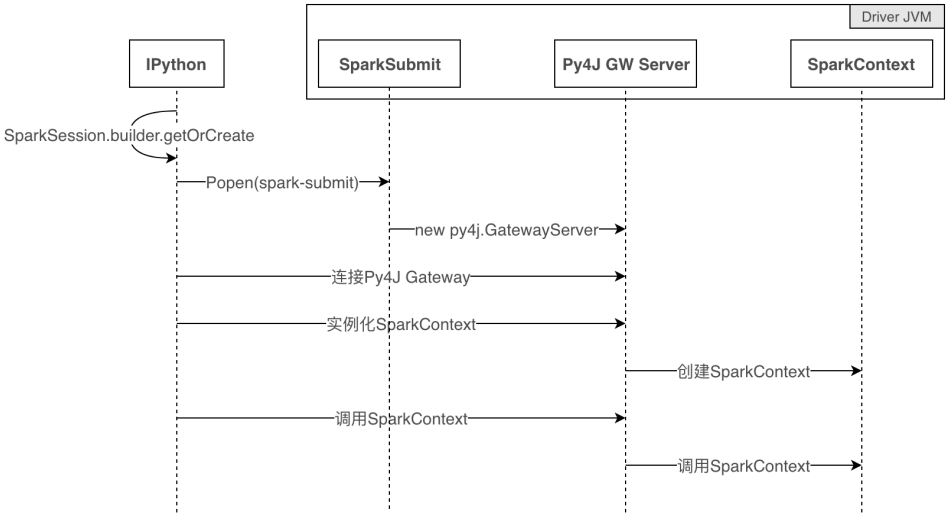
与 Spark 的区别是，多了一个 Python 进程，通过 Py4J 与 Driver JVM 进行通信。

PySpark 方案启动流程



PySpark 启动时序图

IPython 方案启动流程



实际的 IPython 中启动 Spark 时序图

Toree 采用的是类似方案一的方式，脚本中调用 spark-submit 执行特殊版本的 Shell，内置了 Spark 会话。我们不希望这么做，是因为如果这样做的话就会：

- 多了一个 PySpark 专供的 Kernel，我们希望 Kernel 应该是统一的 IPython。
- PySpark 启动参数是固定的，配置在 kernel.json 里。希望 PySpark 任务是可以按需启动，可以灵活配置所需的参数，如 Queue、Memory、Cores。

因此我们采用方案二，只需要一些环境配置，就能顺利启动 PySpark。另外为了简化 Spark 启动工作，我们还开发了 IPython 的 Magics，%spark 和 %sql。

环境配置

为了让 IPython 中能够顺利启动起 Spark 会话，需要正确配置如下环境变量：

- JAVA_HOME：Java 安装路径，如 /usr/local/jdk1.8.0_201。
- HADOOP_HOME：Hadoop 安装路径，如 /opt/hadoop。
- SPARK_HOME：Spark 安装路径，如 /opt/spark-2.2。
- PYTHONPATH：额外的 Python 库路径，如 $SPARK_HOME/python:$SPARK_HOME/python/lib/py4j-0.10.4-src.zip。
- PYSPARK_PYTHON：集群中使用的 Python 路径，如 ./ARCHIVE/notebook/bin/python。集群中使用 Python 通常需要虚拟环境，通过 spark.yarn.dist.archives 带上去。
- PYSPARK_DRIVER_PYTHON：Spark Driver 所用的 Python 路径，如果你用 Conda 管理 Python 环境，那这个变量应为类似 /opt/conda/envs/notebook/bin/python 的路径。

为了方便，建议设置各 bin 路径到 PATH 环境变量中：$SPARK_HOME/sbin:$SPARK_HOME/bin:$HADOOP_HOME/sbin:$HADOOP_HOME/bin:$JAVA_HOME/bin:$PATH。

完成这些之后，可以在 IPython 中执行创建 Spark 会话代码验证：

```
import pyspark
spark = pyspark.sql.SparkSession.builder.appName("MyApp").getOrCreate()
```

在 Spark 任务中执行 Notebook

执行 Notebook 的方案目前有 nbconvert，Python API 方式执行样例如下所示，暂时称这段代码为 NB-Runner.py：

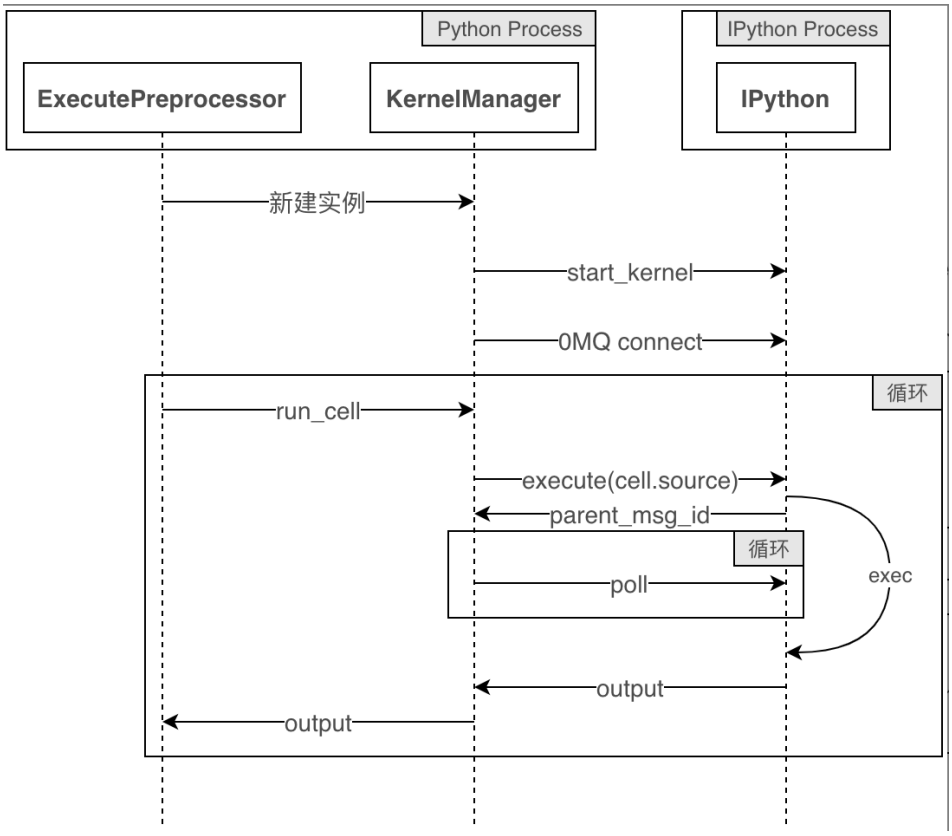
```
# Import：首先我们 import nbconvert 和 ExecutePreprocessor 类：
import nbformat
from nbconvert.preprocessors import ExecutePreprocessor

# 加载：假设 notebook_filename 是 notebook 的路径，我们可以这样加载：
with open(notebook_filename) as f:
    nb = nbformat.read(f, as_version=4)

# 配置：接下来，我们配置 notebook 执行模式：
ep = ExecutePreprocessor(timeout=600, kernel_name='python')

# 执行（preprocess）：真正执行 notebook 的地方是调用函数 preprocess：
ep.preprocess(nb, {'metadata': {'path': 'notebooks/'}})

# 保存：最后，我们保存 notebook 执行结果：
with open('executed_notebook.ipynb', 'w', encoding='utf-8') as f:
    nbformat.write(nb, f)
```

现在有两个问题需要确认：

- 当 Notebook 中存在 Spark 相关代码时，Python NB-Runner.py 能否正常执行?
- 当 Notebook 中存在 Spark 相关代码时，Spark-Submit NB-Runner.py 能否正常执行?

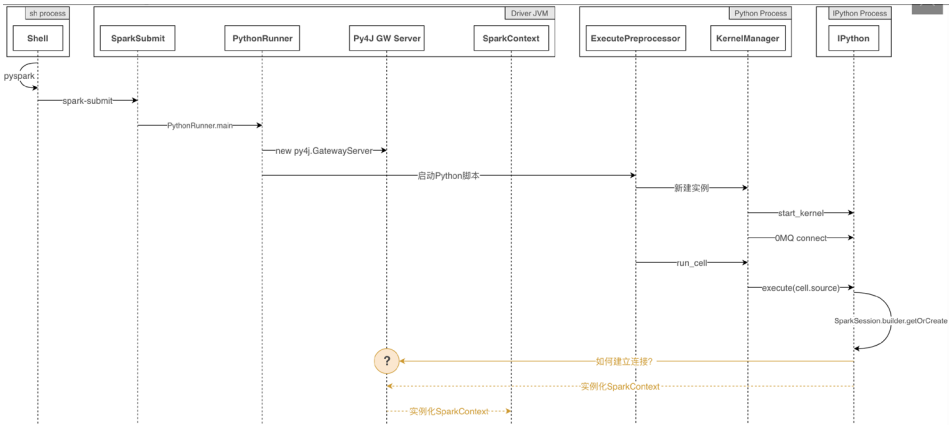
之所以会出现问题 2，是因为我们的调度系统只能调度 Spark 任务，所以必须使用 Spark-Submit 的方式来启动 NB-Runner.py。

为了回答这两个问题，需要了解 nbconvert 是如何执行 Notebook 的。



nbconvert 执行时序图

问题 1 从原理上看，是可以正常执行的。实际测试也是如此。对于问题 2，答案似乎并不明显。结合“PySpark 启动时序图”、“实际的 IPython 中启动 Spark 时序图”与“nbconvert 执行时序图”：



Spark-Submit NB-Runner.py 的方式存在问题的点可能在于，IPython 中执行 Spark.builder.getOrCreate 时，Driver JVM 已经启动并且 Py4J Gateway Server 已经实例化完成。如何让 Spark.builder.getOrCreate 执行时跳过上图“实际的 IPython 中启动 Spark 时序图”的 Popen(spark-submit) 以及后续的启动 Py4J Gateway Server 部分，直接与 Py4J Gateway Server 建立连接?

在 PySpark 代码中，看到如下这段代码：

```
def launch_gateway(conf=None):
    """
    launch jvm gateway
    :param conf: spark configuration passed to spark-submit
    :return:
    """
    if "PYSPARK_GATEWAY_PORT" in os.environ:
        gateway_port = int(os.environ["PYSPARK_GATEWAY_PORT"])
    else:
        SPARK_HOME = _find_spark_home()
        # Launch the Py4j gateway using Spark's run command so that we
pick up the
        # proper classpath and settings from spark-env.sh
        on_windows = platform.system() == "Windows"
        script = "./bin/spark-submit.cmd" if on_windows else "./bin/
spark-submit"
...
```

如果我们能在 IPython 进程中设置环境变量 PYSPARK_GATEWAY_PORT 为真实的 Py4J Gateway Server 监听的端口，就会跳过 Spark-Submit 以及启动

Py4J Gateway Server 部分。那么 PYSPARK_GATEWAY_PORT 从哪来呢？我们发现在 Python 进程中存在这个环境变量，只需要通过 ExecutorPreprocessor 将它传递给 IPython 进程即可。

使用案例

数据分析与可视化

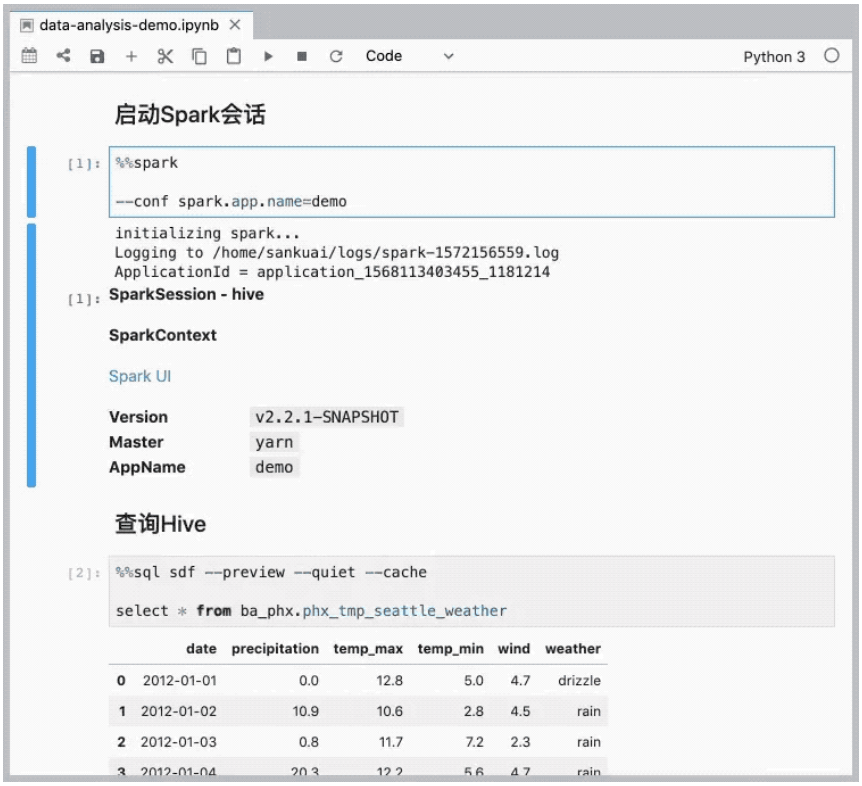
数据探查和数据分析在这里都是同样的流程。用户要分析的数据通常存储在 MySQL 和 Hive 中。为了方便用户在 Notebook 中交互式的执行 SQL，我们开发了 IPython Magics %%sql 用来执行 SQL。

SQL Magics 的用法如下：

```
%%sql <var> [--preview] [--cache] [--quiet]
SELECT field1, field2
  FROM table1
 WHERE field3 == field4
```

SQL 查询的结果暂存在指定的变量名中，对于 MySQL 数据源的类型是 Pandas DataFrame，对于 Hive 数据源的类型是 Spark DataFrame。可用于需要对结果集进行操作的场合，如多维分析、数据可视化。目前，我们支持几乎所有的 Python 数据可视化库。

下图是一个数据分析和可视化的例子：



数据分析与可视化

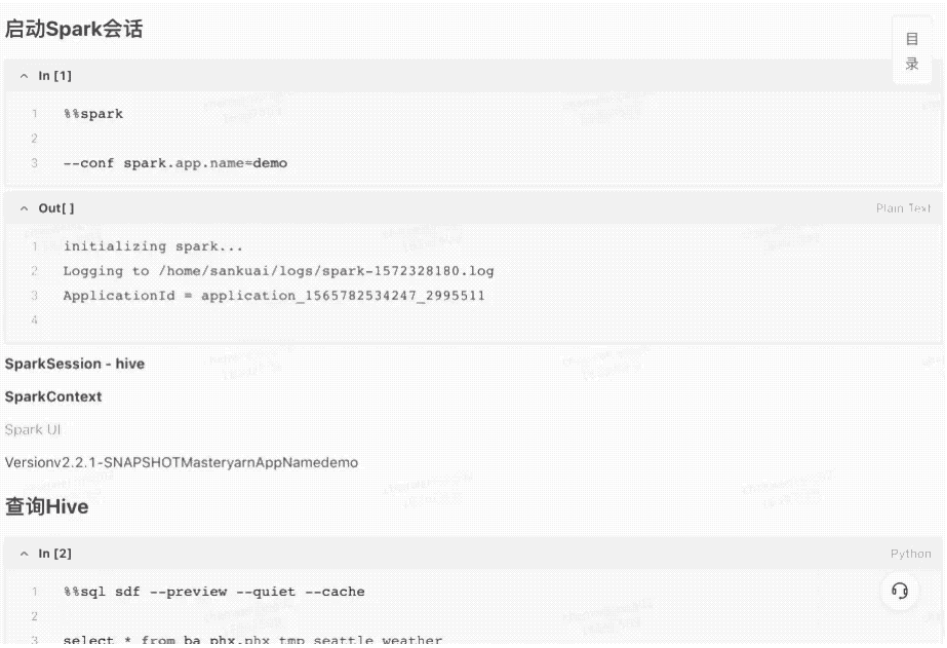
Notebook 分享

Notebook 不仅支持交互式的执行代码，对于文档编辑也有不错的支持。数据分析过程中的数据、表格、图表加上文字描述就是一个很好的报告。Jupyter 服务还支持用户一键将 Notebook 分享到美团内部的学城中。

一键分享：

一键分享

上述数据分析分享到内部学城的效果如下图所示：



Notebook 分享效果

模型训练

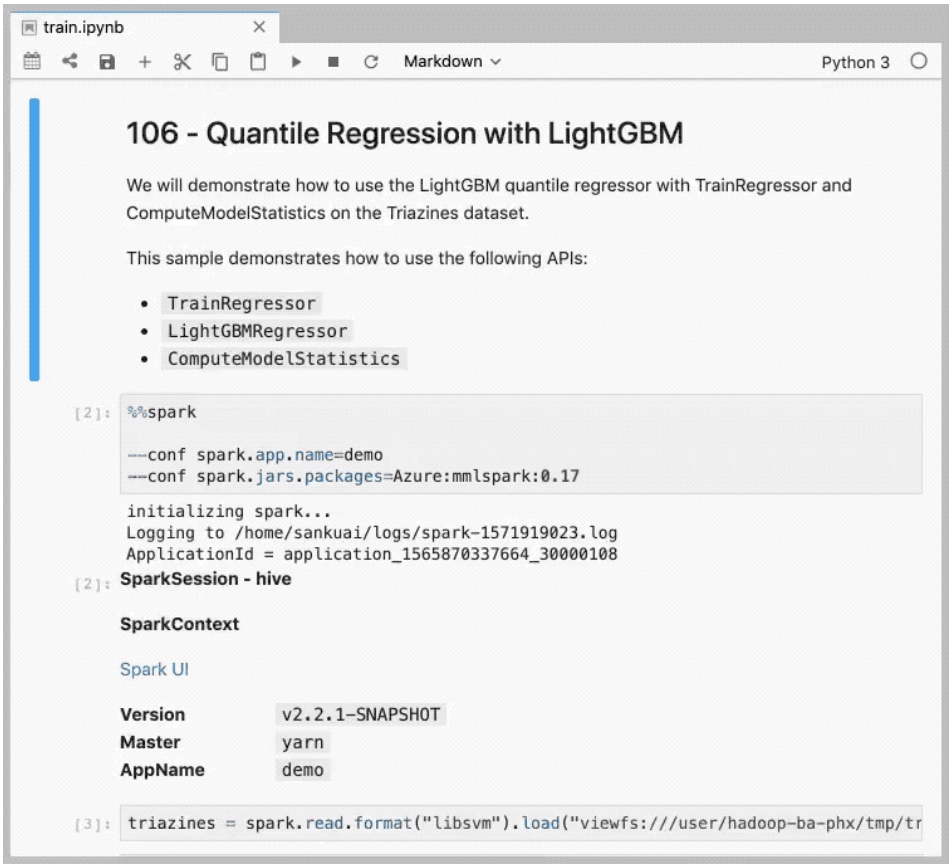
基于大数据的模型训练通常使用 PySpark 来完成。除了 Spark 内置的 Spark ML 可以使用以外，Jupyter 服务上还支持使用第三方 X-on-Spark 的算法，如 XGBoost-on-Spark、LightGBM-on-Spark。我们开发了 IPython Magics %%spark 来简化这个过程。

Spark Magics 的用法如下：

```
%%spark
[--conf <property-name>=<property-value>]
[--conf <property-name>=<property-value>]
...
```

执行 %%spark 后，会启动 Spark 会话，启动后 Notebook 会话中会新建两个变量 spark 和 sc，分别对应当前 Spark 会话的 SparkSession 和 SparkContext。

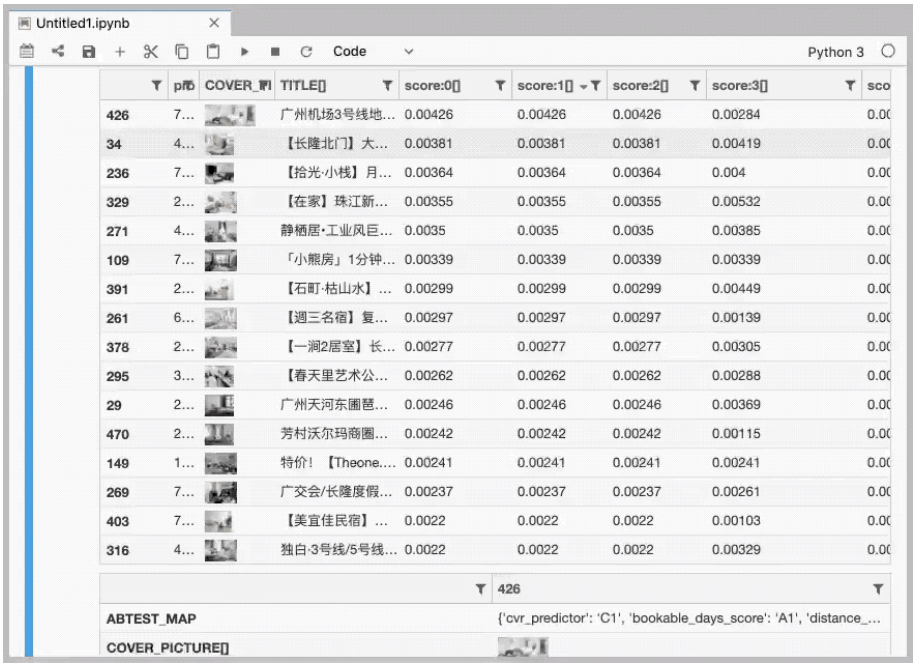
下图是一个使用 LightGBM-on-Yarn 训练模型的例子，基于 Azure/mml-spark 官方 Notebook 例子，仅需添加启动 Spark 语句以及修改数据集路径。



LightGBM on Spark Demo

排序策略调试

通过开发 ipywidgets 实现了一个线上排序策略的调试工具，可以用于查看排序结果以及排序原因（通过查看变量值）。



总结与展望

通过平台化 Jupyter 的定制与部署，我们实现了数据分析、数据生产、模型训练的统一开发环境。在此基础上，还集成了内部公共服务和业务服务，从而实现了从数据分析到策略上线到结果分析的全链路支持。

我们对这个项目未来的定位是数据科学的云端集成开发环境，而 Jupyter 项目所具有的极强扩展性，也能够支持我们朝着这个方向不断进行演进。

作者

文龙，美团民宿研发团队工程师。
颖艺，美团民宿研发团队工程师。

招聘

美团民宿研发团队诚招数据系统研发工程师，Base 厦门，欢迎有兴趣的同学投递简历到 tech@meituan.com 。