美团点评

美团点评
2018 技术年货

CODE A BETTER LIFE

后台

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 – 后台篇

APPKIT打造稳定、灵活、高效的运营配置平台

作者: 国宝 小龙

一、背景

美团App、大众点评App都是重运营的应用。对于App里运营资源、基础配置，需要根据城市、版本、平台、渠道等不同的维度进行运营管理。如何在版本快速迭代过程中，保持运营资源能够被高效、稳定和灵活地配置，是我们团队面临的重大考验。在这种背景下，大众点评移动开发组必须要打造一个稳定、灵活、高效的运营配置平台。本文主要分享我们在建设高效的运营配置平台过程中，积累的一些经验，以及面临的挑战和思考。

运营资源

简单而言，运营资源可以理解为App中经常变动的一些广告、运营活动等等，譬如下图中电影首页顶部的Banner位，就是一个典型的运营资源。对于这类运营资源，它们有如下明显特征：

1. 时效性，只在一定时间范围内显示在C端固定位置。
2. 城市强相关，这类运营资源往往是基于LBS类服务，每个活动、广告都只会出现在固定的某些城市（或区域）。

基础配置

基础配置，常见的有入口资源的配置、网络的配置等。相对运营资源来说，其变更的频繁度相对较低，与时间、城市的关系也没那么强。譬如下面大众点评App–我的页面里的入口。这类配置有如下几个特征：

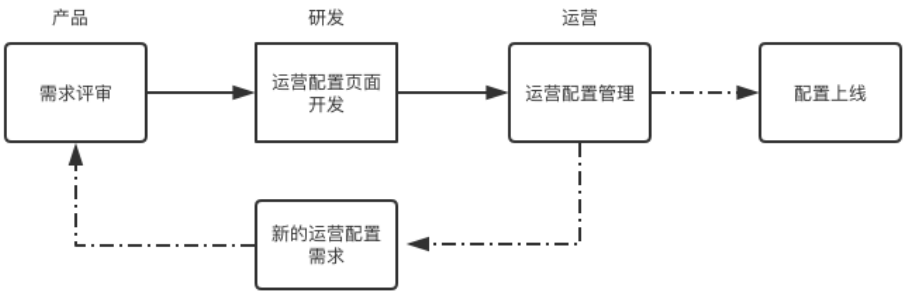
1. 多维度：需要针对不同的版本、平台、渠道，做不同的配置。
2. 长期有效：这种类型的配置一般长期存在，不会存在过期问题。

二、遇到的问题

在从0到1打造运营配置平台的过程，我们遇到了很多“坑”。特别是在早期“刀耕火种”的时代，对于入口的配置，往往是通过“hardcode（硬编码）”的方式写死在代码中。所以必然会遇到很大的问题，这主要体现为以下两方面：

运营效率低

对于新的运营配置需求，研发同学需要开发对应的配置页面，然后转给运营同学进行配置的管理，最后运营人员对资源进行配置上线，其流程如下：

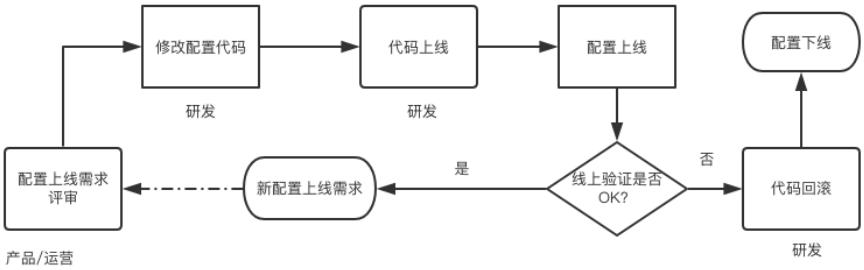


对于每个运营配置需求都要经过需求评审、页面开发、配置管理、上线的流程。同时，对于配置页面的开发，少则需要1到2天的开发工时，研发成本高。问题总结如下：

1. 研发成本高，每个需求要开发新的配置管理页面。
2. 研发周期长，运营效率低，从需求的提出到运营上线周期长。
3. 灵活性差，对不同的运营维度（城市、版本、时间等）都需要事先确定好，无法动态调整。

上线流程“粗糙”

在早期，运营配置上线流程需要研发同学参与。产品提出运营配置需求，研发同学通过修改代码对配置进行变更，然后通过代码上线进行发布。整体流程如下：



这种上线机制存在以下几个问题：

1. 配置上线过多依赖于代码的发布。
2. 整体上线过程无审核机制，无法对配置资源进行合规审核。
3. 配置容易出错，上线前不能提前预览上线后的效果，只有“事后”（上线后）才能验证效果。

三、我们的思考

针对以上问题，我们希望通过设计一个通用的解决方案，去解决上文阐述的各种运营资源管理的问题。我们把这个整体的项目称之为APPKIT，寓意是App的运营配置工具（Kit）。通过不断的实践和总结，我们希望能从三个维度解决上述问题：

数据JSON化

随着业务的不断迭代，无论采用怎样的数据字段组成，都无法满足业务变化的字段（这里是指像标题、副标题、图片、跳转链接等）要求。对底层数据进行JSON化，其对应的数据字段完全可动态扩展，从而满足业务不断迭代的需求。JSON化随之也会带来运营位字段管理的问题，我们通过字段管理的工具来解决这个问题。

运营流程化

设计一套整体的流程管理机制，解决运营的投放、审核、发布和回滚的问题。通过流程化的机制，我们实现了“事前”、“事中”、“事后”的三级管理。

首先，在运营配置上线前，通过测试用户的预览功能，可以预览上线后的实时效果。同时，通过穿越功能可查看将来时段显示的效果。防止出现上线后链接出错、视觉效果达不到预期等问题。

其次，在流程阶段，引入审核机制，通过视觉和内容两方面的审核，保证投放数据的准确性。

最后，在运营配置上线后，如果发现问题，可以通过快速回滚，最大限度地实现“止损”。

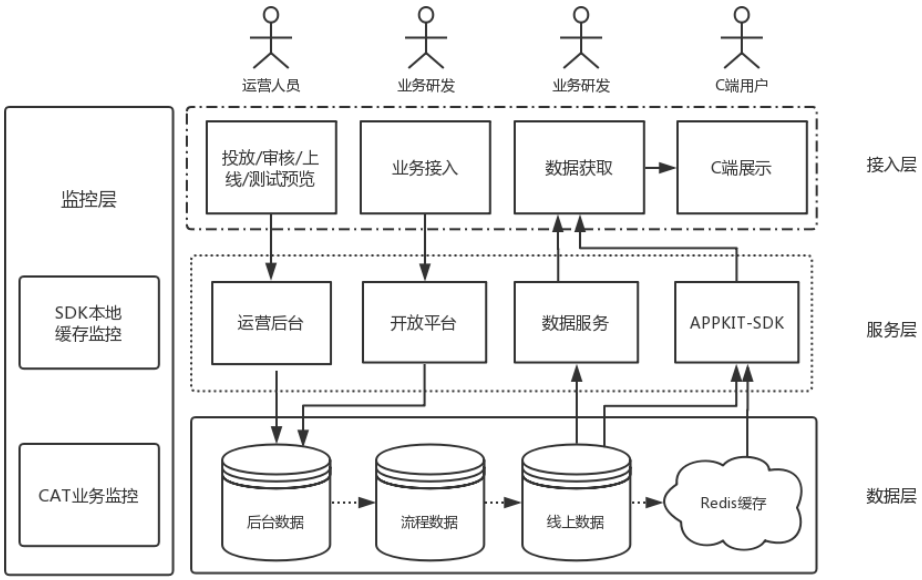
	事前	事中	事后
机制保证	测试预览、穿越预览	多重审核	回滚
解决问题	C端展示问题、链接异常、平台差异	敏感内容过审、图片质量	出错处理、排期问题、最大限度止损

接口SDK化

对于运营数据，无论是通过数据库的落地方案、还是通过分布式缓存的方案，都无法彻底解决服务中心化和服务抖动的问题。通过接入的SDK化，可以做到数据的本地缓存更新机制，解除对中心化服务的依赖，大大提升服务的稳定性和性能。同时整个APPKIT服务变成可水平扩展，在扩展过程中也不会影响中心服务的稳定性。

四、APPKIT架构

APPKIT运营配置系统整体框架如下（数据流向如箭头所示）。从功能角度，大体上分为四层：数据层、服务层、接入层和监控层。



4.1 数据层

数据层作为最底层的数据存储，其保存了最基本的运营后台数据、流程数据和线上数据。对持久化的数据，我们采用MySQL进行存储；对于缓存数据，我们采用了Redis的解决方案。这样数据层形成基本的两级存储结构：MySQL保证了数据的持久性，Redis保证了数据获取的速度。

这里我们对底层数据划分为三个不同域：后台数据，相当于草稿数据，运营人员所有的操作都记录在这里；流程数据，运营人员操作完成后，提供发布流程，预览及审核都在流程数据里进行；线上数据，审核通过后，数据同步到线上数据，最终C端用户获取到的数据都是来源于线上数据。

谈到数据层，这里我们遇到了存储上的一个小问题。按城市运营的每条数据，都需要存储具体的城市ID列表，其在数据库里的存储为“1,2,3,4……”这样字符串。而这种数据存储在业务请求和条件过滤过程中，存在着如下两个问题：

a. 大数据存储对内存的消耗

美团、大众点评运营的城市成千上万，如果每条运营的投放数据都包含大量的城市列表信息，对机器内存势必产生一定消耗。

b. 过滤性能问题

城市的过滤逻辑大体是这样：用户所在城市与从数据库获取到的城市列表（“1,2,3,4……”）进行匹配，在每次匹配过程中都需要做字符串“split”的切割操作。这种操作的特点是流量越大，对机器CPU的消耗越大。

解决方案：基于以上两点考虑，再结合Java语言提供的BitSet机制。我们从数据库里取出城市列表数据后只做一次“split”切割操作，将数据转化为BitSet类型。这样在实际过滤过程中只需要通过BitSet的get机

制就可以判断运营投放的城市是否包含了用户所在的城市。通过BitSet机制，我们既解决了大数据存储对内存的消耗问题，又解决了城市过滤的性能问题。

4.2 服务层

服务层向下对底层数据进行操作；向上为接入层获取数据提供接入能力。其提供四个服务能力：运营后台、开放平台、数据服务、APPKIT-SDK，如下表所列：

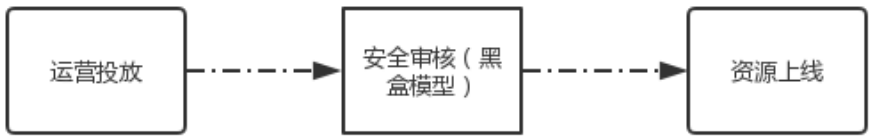
服务类型	涉及人员	能力说明
运营后台	产品、研发、运营	配置运营（投放、审核、上线），运营位管理（创建、字段管理、权限、数据导入导出）
开放平台	产品、研发	业务接入，对运营平台数据进行操作（写）
数据服务	研发、QA	通过RPC服务调用获取运营位相关数据（读）
APPKIT-SDK	研发、QA	通过SDK进行数据获取（读）

服务层

4.3 接入层

接入层主要为运营人员、业务研发提供接入能力。通过运营流程化为事前、事中、事后提供保障。一个运营资源从制作到最后在C端展示，需要经过运营人员的投放、测试预览、审核及发布的中间流程。这里对于一些敏感的运营资源，需要通过安全部门的审查。安全审查主要涉及到敏感词的处理、敏感图片的检测等。对运营配置平台来说，它完全是一个“黑盒模型”。这里主要涉及到两种情况：

1. 资源上线时



2. 资源上线后

4.4 监控层

APPKIT-SDK运行在业务机器上，这里就涉及到多台机器的数据一致性问题。同时，随着业务接入运营数据的增多，SDK对机器内存势必有一定消耗。基于服务的稳定性考虑，我们对SDK运行时的投放内容进行监控，其主要监控两个指标：运营位数及每个运营位的配置总数。这样做可以带来以下几个好处：

1. 对接入的业务数及机器数进行统计。
2. 通过SDK的配置总数监控，防止数量超过最大限制。

同时，对于非SDK的其他性能指标，我们采用统一的监控平台－ CAT 进行监控，其中包括：APPKIT中心服务的调用QPS，机器的性能，网络流量等通用指标。

五、底层模型－灵活性设计

5.1 从一个例子切入

数据模型往往与业务相关。业务越复杂，设计需要越简单，这样方能满足复杂业务的各种变化。因为数据模型太过于抽象，如果直接进行述说会有些乏味，我们可以先从一个具体的业务实例入手。下面是大众点评App顶部金刚位的截图，对于这部分数据，如何做到运营可配?

首先，我们对运营数据做需求拆解。对于这块数据，每个“节点”（对应每个位置：如美食，技术上我们称之为“节点”），其基本的运营诉求如下：

1. 节点内容信息：标题、图片、跳转链接、排序。
2. 节点的过滤维度信息：城市、版本、平台、渠道等。
3. 节点其他信息：角标，如外卖节点，其有一个下午茶这样的角标。值得注意的是，像下午茶这样的角标，除去文案、文案颜色这些基本信息之外，我们也可以按城市、平台、进行过滤（不周的城市对应的文案可能不一样，如上海为“下午茶”，北京因为嘉年华活动可以改成“嘉年华”）。

上面这个运营场景算是非常经典、复杂的一个运营场景了，如果这个问题解决了，其他问题自然就会迎刃而解。

5.2 技术分析

我们做一下进一步的技术分析：

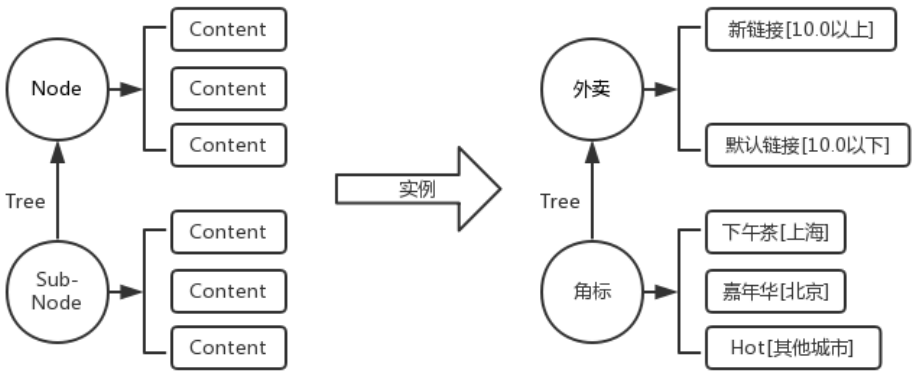
首先，这里有节点，每个节点（Node）有其相应的内容（Content），节点与内容是“一对多”的关系。这里的内容，我们指的是如标题、图片、跳转链接等信息，虽然是“一对多”的关系，但最后在同一个城市、同一个版本下（可选择）只显示一条内容。为什么有这样的需求？举一个简单的业务场景实例，以外卖为例，在新版本10.0的时候做了一个全新的外卖频道页面，其链接信息与老版本的完全不同，这里我们就需要按版本的不同配置两条不同的内容信息。

其次，节点与节点之间有两层关系，其一为“平级关系”，如美食与外卖的关系，这种关系就是一种简单的列表关系；其二为树关系（Tree），如外卖与下午茶之间的关系。这里我们将角标（下午茶）视为一个节点，因为角标也需要按不同维度进行过滤，因此下午茶成了外卖的子节点。其实这里有一些特殊的地方，如果角标不需要按城市、版本等维度进行运营，那很简单它就是一个内容信息（类似标题）。

最后，我们谈一谈排序问题，对于这么多品类，如何排序，他们的优先级是什么？我们需要定一个基本的基调，每个节点都有一个基本的排序值（优先级）。但深入业务分析，对于不同的人（群），每个人关注的点不一样，需要一个“千人千面”的算法，来决定每个人所看到的内容是其真正关心的内容。所以，这种应用场景下的排序应该通过机器学习算法而得到。

5.3 数据建模

针对上面的技术分析，我们提出了一种节点（Node）–内容（Content）–树（Tree）模型，简称为N–C–T模型。如下图所示：左边为抽象的数据模型，右边为以上实例的实现。



N–C–T数据模型设计的非常简单，其中C和T部分都是可选择的，这样使得其灵活性比较强，可以应对业务变化的大部分需求。注意，这里我们只是对业务需求的宏观表现形式进行建模，对于具体Node和Content里的有哪些字段（标题、副标题、图片、跳转链接），这些都是JSON化的存储格式，可以满足任意字段的扩展。

5.4 模型的应用与小结

通过以上经典实例，我们可以很容易通过我们的数据模型解决这个问题。我们再回到文章最开头的背景章节的运营场景，Banner位，如下：

这种Banner位，套用我们上的数据模型，它其实是一种只有一个Node节点、多个Content节点的模型。这也是一种典型的应用场景，为此我们总结了两种应用场景。

	节点(Node)	内容(Content)	树(Tree)
金刚位	n	n	n
Banner位	1	n	0

其实，大部运营场景都可以套用以上两种经典的运营组合。

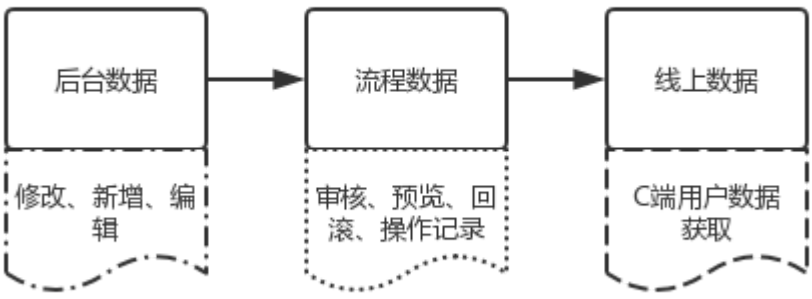
六、运营流程化

将运营资源的管理进行流程化，具有以下几个好处：

1. 资源上线前可进行严格的审核。
2. 出问题时可快速回滚。
3. 记录用户的（上线）操作历史。
4. 上线前可提前预览线上效果。

数据域

对于流程化的实现，我们是将数据域切分成三个不同的部分：后台数据、流程数据和线上数据，如下图所示：



后台数据：我们可以简单理解为草稿数据，这里的数据多用户可同时进行操作，也不会对线上数据有影响。

流程数据：当用户后台数据编辑完成后，对数据提交一个发布流程，数据进入流程数据区域；这时可对数据进行测试预览、审核等操作。

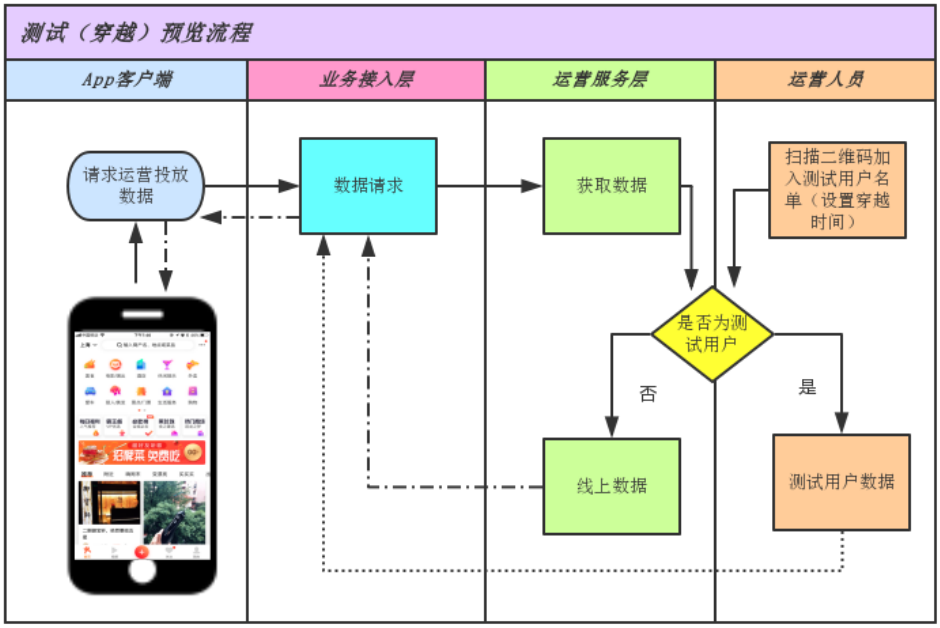
线上数据：这块数据是C端用户真正获取到的数据，当流程数据审核通过后，数据会自动同步到线上数据域，完成上线操作。

上线流程

整个上线流程如下：

1. 运营同学对后台数据进行修改，提交发布流程，同时进行预览测试。
2. 审核同学对提交的发布流程进行审核。
3. 审核通过，自动发布到线上（对审核不通过的流程，数据回退到后台进行再次编辑）。

为了能平稳上线，我们设计了一个测试预览功能。当数据处于流程中时，用户可以通过扫描二维码加入到测试用户名单，可对处于审核流程中的数据进行预览，用美团、大众点评App查看上线后实时效果，其实现的数据流如下：

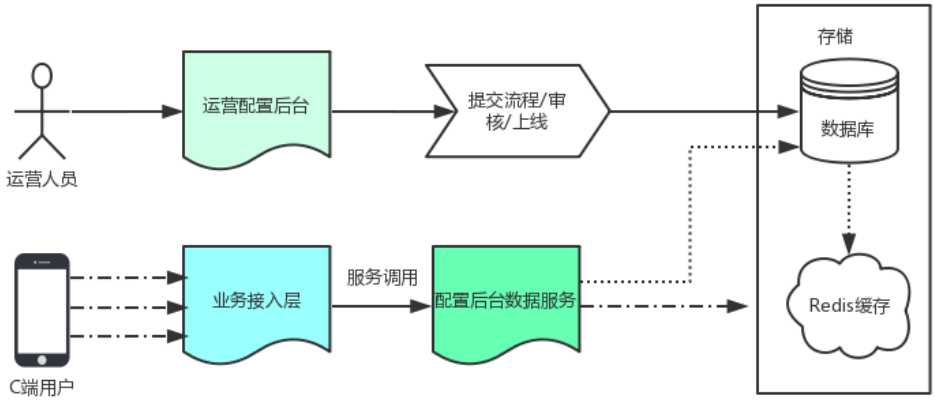


七、稳定性的演进

稳定性是一个运营配置平台最重要的能力，没有稳定性，其他任何功能都会失去实际意义。运营系统的稳定性经历了不同的迭代时期，总结起来，可概括为以下三个阶段：

7.1 经典方案

这是APPKIT最早期的经典方案，它的实现也非常简单，如下图所示：

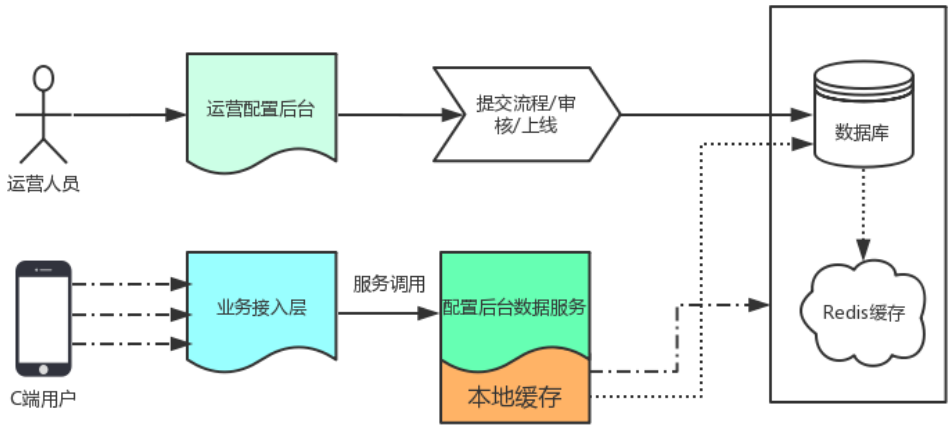


C端用户通过业务接入层获取数据，业务接入层通过服务调用获取配置后台数据（APPKIT服务），配置后台数据服务读取缓存数据。如果缓存数据不存在，则从数据库中读取数据，同时将数据库数据同步到Redis缓存中。这是经典的数据获取模型，但它有以下几个缺点：

1. 数据调用有网络时延。
2. Redis缓存抖动（网络抖动）会对服务的稳定性产生影响。
3. 缓存的单Key存储的容量限制。

7.2 缓存方案

针对以上经典方案的缺点，我们做了进一步的改进，对配置后台数据服务做了一层本地内存缓存，如下图所示：

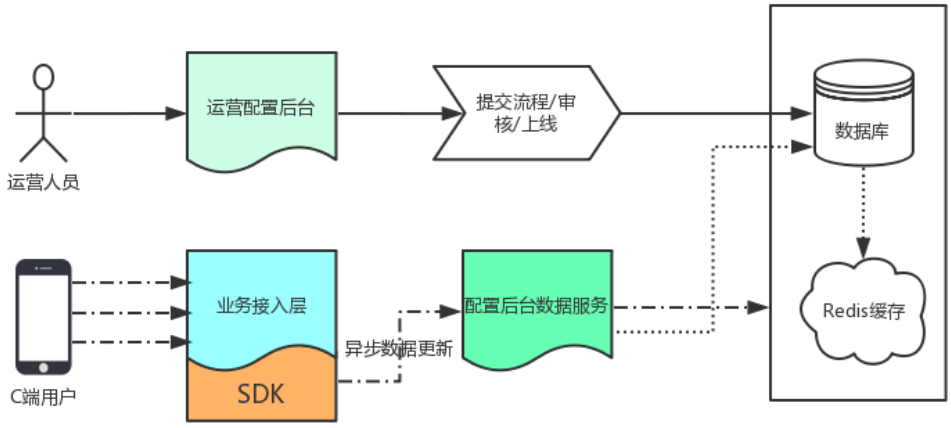


样做可以解决数据调用的部分网络时延问题，同时Redis缓存的抖动也不会影响整体服务的性能。不过，这个方案也有其自身的缺陷。

1. APPKIT服务成了中心节点，业务方对中心节点强依赖，随着业务接入越来越多，中心服务的压力会等比例增加。
2. 业务接入层与配置后台数据服务间调用的网络时延问题仍然存在。
3. 单机的Web缓存容量有限，随着业务接入越多，APPKIT服务器本地缓存的数据量越大。

7.3 SDK方案

为彻底解决缓存方案的问题，尤其是服务中心化带来的流量、容量等问题，我们将运营数据的获取、Web缓存的管理集成进SDK。如下图所示：

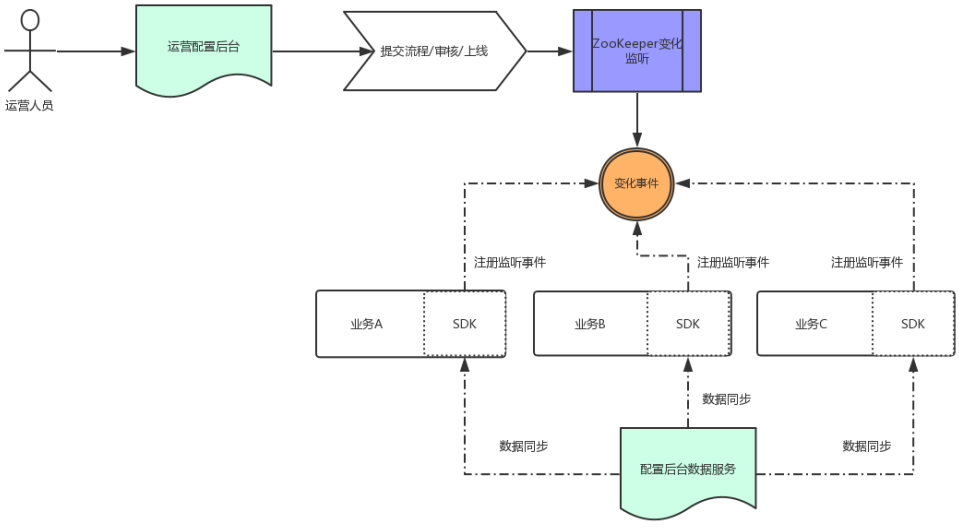


这样的话，无论接入再多的业务，也不会对中心服务产生过大的流量压力和容量压力。SDK同时也解决了服务间调用的网络时延问题。所有同步数据的网络调用都是通过后台线程异步完成，不会影响业务线程的正常处理逻辑。

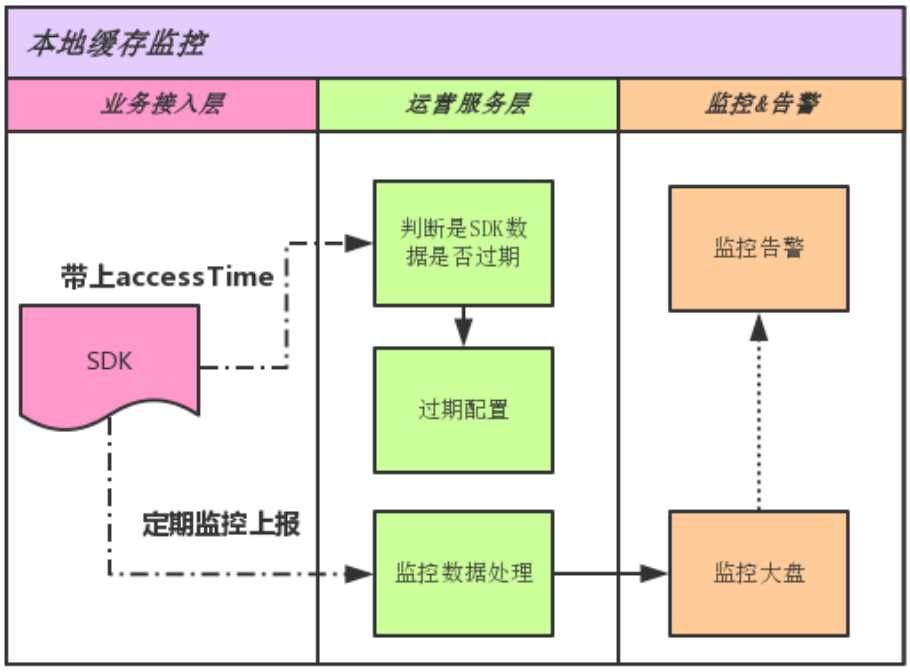
不过，SDK方案也引进了如下的新问题：

1. 数据时效性和一致性如何保证?
2. SDK本地缓存如何监控？过期数据如何删除?
3. SDK版本如何升级?

为了解决数据的时效性和一致性问题，我们引入了监听更新机制，如下图所示：



运营人员在运营后台操作完成后，提交上线流程，流程发布后通过ZooKeeper的变化监控发送一个变化事件；SDK通过监听变化事件，拉取变化后的运营数据更新到本地。这里，为了防止这种监听机制失效，我们也做了一个兜底策略：每分钟定期进行一次数据同步。这样保证数据最迟一分钟内就能实现同步。对于SDK本地缓存，我们设计了监控上报机制，如下图所示：



这里有两条线路，其一为SDK在请求数据时，带上数据的accessTime时间戳，APPKIT服务会根据accessTime时间戳判断SDK本地数据是否过期。当accessTime时间超过24小时，说明这个运营位在一天内都没有使用，可以从本地内存中进行删除。其二为SDK定期进行监控上报，上报SDK本地缓存的数目，这样可以对SDK本地缓存进行监控和告警。对SDK版本升级问题，现有的解决方案，是通过CI构建时对SDK版本升级进行提示（必要时进行强制），不过大部分运营位使用的都是基础功能，在很大程度上不需要进行频繁地升级。

效果对比

缓存方案与SDK方案的效果对比如下：

QPS/性能指标（单位ms）	平均线	90线	95线	99线	99.9线
缓存方案（5.6K）	0.4	2.0	2.0	2.8	2.8
SDK化方案 （5w）	0.0	1.0	1.0	1.1	2.1

注: SDK方案的平均线为0.0是因为统计时舍入引起的，真实值其实非常小。

八、总结与展望

本文通过美团点评移动运营平台的实践，详细介绍了我们在打造稳定、灵活、高效的运营配置平台过程中遇到的问题和挑战，同时本文也提供我们的解决思路：通过数据JSON化，运营流程化，接口SDK化分别解决了运营平台的灵活性、高效性和稳定性。APPKIT帮助产品、运营和研发提升C端的开发和运营效率，加速产品的迭代进程。

目前基于APPKIT的平台化特性，对App的模块化配置、 Picasso 的JS的管理、ABTest、个人中心入口管理、鲁班（面向C端的Key/Value配置系统）等业务提供了底层的数据存储和数据获取的支持，为移动端业务提供了运营配置的基础保障。

同时，为了进一步提升运营效率，我们基于Picasso的多端（Android、iOS、H5、微信小程序）能力，正在构建移动化的运营能力。这样保障用户无论在什么办公环境都能进行运营配置的管理。

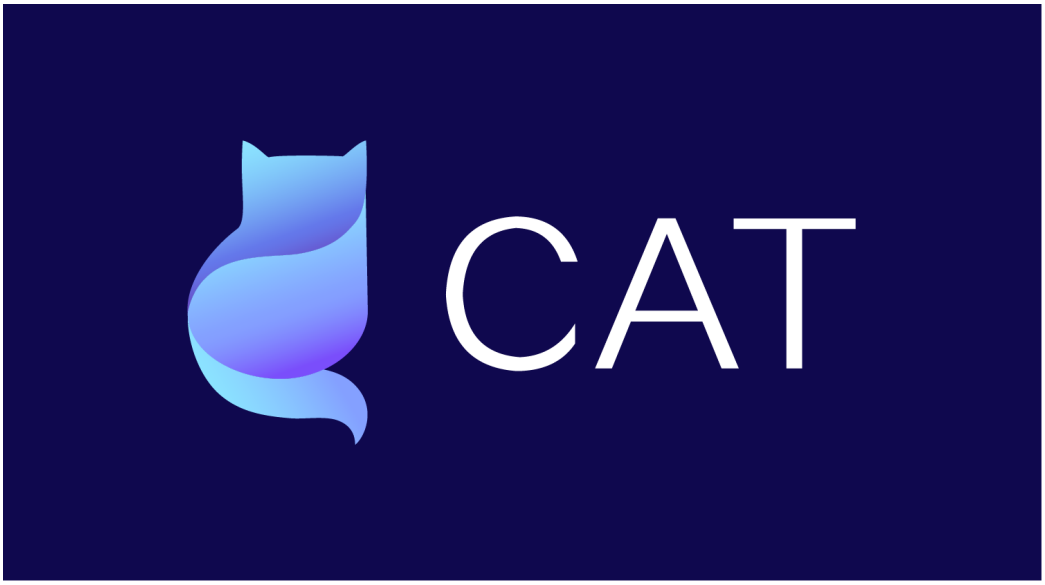
作者简介

- 国宝，美团点评移动运营平台负责人，Java后端架构师，APPKIT项目发起人，负责APPKIT项目的架构设计。专注于高性能、高稳定、大并发系统的设计与应用。
- 小龙，目前为APPKIT项目负责人，主要负责APPKIT项目开发、技术对接和实施、开放平台等。专注于前后端全栈技术开发，喜欢挑战新的技术和业务问题。

CAT 3.0 开源发布，支持多语言客户端及多项性能提升

作者: 尤勇

项目背景



CAT（Central Application Tracking），是美团点评基于 Java 开发的一套开源的分布式实时监控系统。美团点评基础架构部希望在基础存储、高性能通信、大规模在线访问、服务治理、实时监控、容器化及集群智能调度等领域提供业界领先的、统一的解决方案，CAT 目前在美团点评的产品定位是应用层的统一监控组件，在中间件（RPC、数据库、缓存、MQ 等）框架中得到广泛应用，为各业务线提供系统的性能指标、健康状况、实时告警等服务。

本文会对 CAT 的客户端、性能等做详细深入的介绍，前不久我们也发过一篇 CAT 相关的文章，里面详细介绍了 CAT 客户端和服务端的设计思路，欲知更多细节，欢迎阅读 《深度剖析开源分布式监控 CAT》

产品价值

- 减少故障发现时间。
- 降低故障定位成本。
- 辅助应用程序优化。

技术优势

- 实时处理：信息的价值会随时间锐减，尤其是在事故处理过程中。
- 全量数据：全量采集指标数据，便于深度分析故障案例。
- 高可用：故障的还原与问题定位，需要高可用监控来支撑。
- 故障容忍：故障不影响业务正常运转、对业务透明。
- 高吞吐：海量监控数据的收集，需要高吞吐能力做保证。

- 可扩展：支持分布式、跨 IDC 部署，横向扩展的监控系统。

使用现状

目前，CAT 已经覆盖了美团点评的外卖、酒旅、出行、金融等核心业务线，几乎已经接入美团点评的所有核心应用，并在生产环境中大规模地得到使用。

2016 年初至今，CAT 接入的应用增加了400%，机器数增加了 900%，每天处理的消息总量高达 3200 亿，存储消息量近 400TB，高峰期集群 QPS 达 650万/秒。

面对流量的成倍增长，CAT 在通信、计算、存储方面都遇到了前所未有的挑战。整个系统架构也经历了一系列的升级和改造，包括消息采样聚合、消息存储、业务多维度指标监控、统一告警等等，项目最终稳定落地。为公司未来几年内业务流量的稳定增长，打下了坚定的基石。

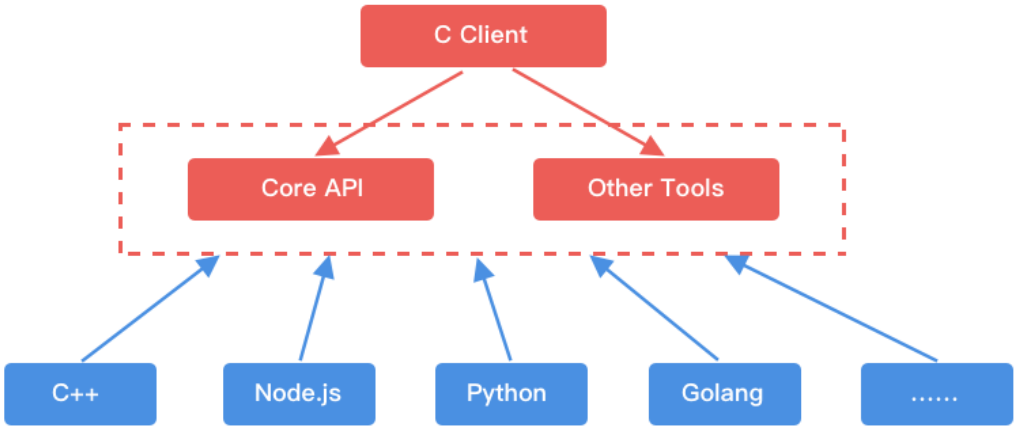
经过 7 年的持续建设，CAT 也在不断发展，我们也希望更好的回馈社区，将 CAT 提供的服务惠及更多的外部公司。我们今年将对开源版本进行较大的迭代与更新，未来也会持续把公司内部一些比较好的实践推广出去，欢迎大家跟我们一起共建这个开源社区。

新版特性

CAT 3.0.0 Release Notes

多语言客户端

随着业务的不断发展，很多产品和应用需要使用不同的语言，CAT 多语言客户端需求日益增多，除 Java 客户端外，目前提供了 C/C++、Python、Node.js、Golang 客户端，基本覆盖了主流的开发语言。对于多语言客户端，核心设计目标是利用 C 客户端提供核心 API 接口作为底层基石，封装其他语言 SDK。



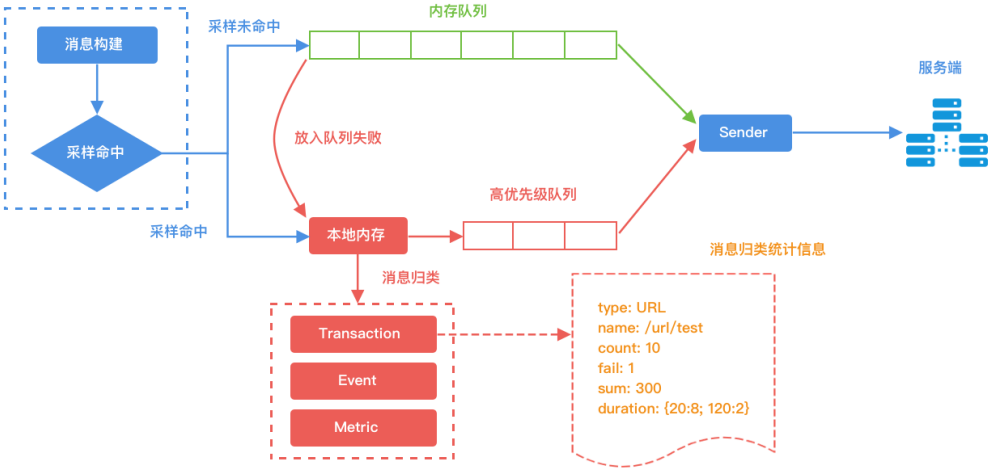
目前支持的主流语言使用指南：

- Java
- C/C++
- Python
- Node.js
- Golang

性能提升

- **消息采样聚合**

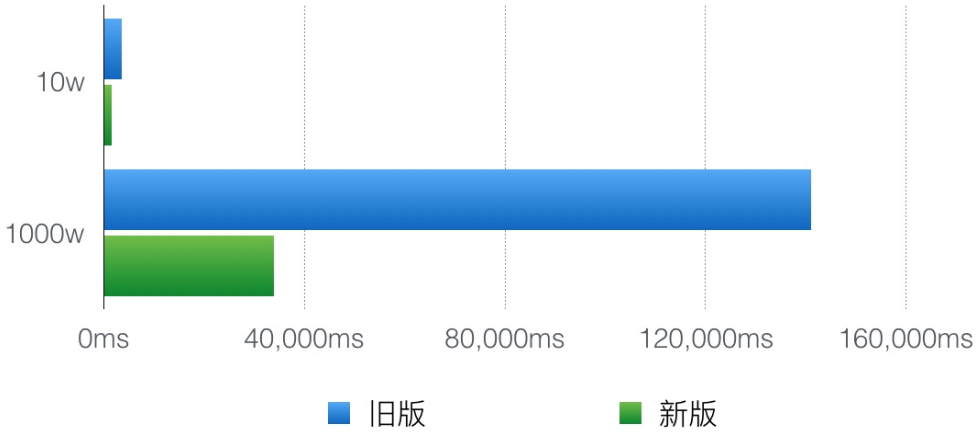
  消息采样聚合在客户端应对大流量时起到了至关重要的作用，当采样命中或者内存队列已满时都会经过采样聚合上报。采样聚合是对消息树拆分归类，利用本地内存做分类统计，将聚合之后的数据进行上报，减少客户端的消息量以及降低网络开销。



- **通信协议优化**

  CAT 客户端与服务端通信协议由自定义文本协议升级为自定义二进制协议，在大规模数据实时处理场景下性能提升显著。目前服务端同时支持两种版本的通信协议，向下兼容旧版客户端。

  - 测试环境：CentOS 6.5，4C8G 虚拟机
  - 测试结果：新版相比旧版，序列化耗时降低约 3 倍

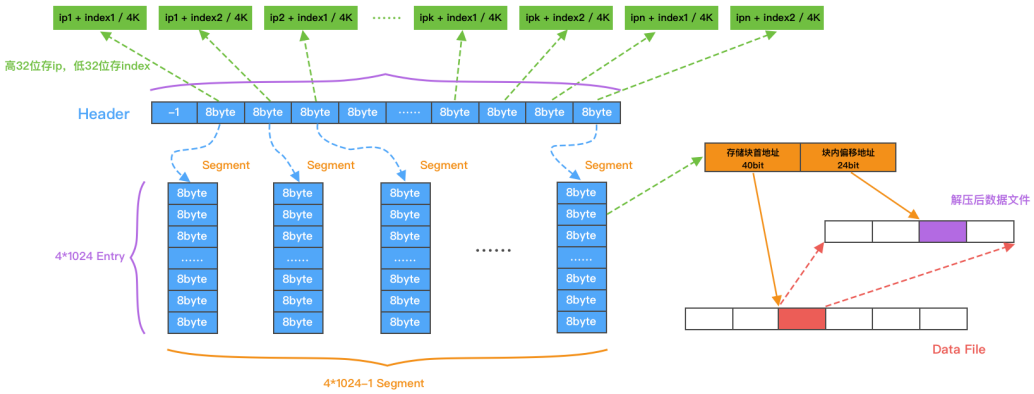


- **消息文件存储**

新版消息文件存储进行了重新设计，解决旧版本的文件存储索引、数据文件节点过多以及随机 IO 恶化的问题。

新版消息文件存储为了同时兼顾读写性能，引入了二级索引存储方案，对同一个应用的 IP 节点进行合并，并且保证一定的顺序存储。下图是索引结构的最小单元，每个索引文件由若干个最小单元组成。每个

单元分为 4×1024 个桶，第一个桶作为我们的一级索引 Header，存储 IP、消息序列号与分桶的映射信息。剩余 4×1024 – 1 个桶作为二级索引，存储消息的地址。



新版消息文件存储文件节点数与应用数量成正比，有效减少随机 IO，消息实时存储的性能提升显著。以下为美团点评内部 CAT 线上环境单机消息存储的数据对比：

每小时单机数据对比	旧版	新版	结果
文件数	3000+	400+	- 650%
消息存储耗时	8min/h	4min/h	- 100%

未来规划

- **技术栈升级**

  拥抱主流技术栈，降低学习和开发成本，使用开源社区主流技术工具（Spring、Mybatis等），建设下一代开源产品。

- **产品体验**

  对产品、交互进行全新设计，提升用户体验。

- **开源社区建设**

  产品官网建设、组织技术交流。

- **更多语言 SDK**

关于开源

https://github.com/dianping/cat

自2014年开源以来，Github 收获 7700+ Star，2800+ Forks，被 100+ 公司企业使用，其中不乏携程、陆金所、猎聘网、平安等业内知 名公司。在每年全球 QCon 大会、全球架构与运维技术峰会等都有持续的技术输出，受到行业内认可，越来越多的企业伙伴加入了 CAT 的开 源建设工作，为 CAT 的成长贡献了巨大的力量。

美团点评基础架构部负责人黄斌强表示，在过去四年中，美团点评在架构中间件领域有比较多的积累沉淀，很多系统服务都经历过大规模线上业务实际运营的检验。我们在使用业界较多开源产品的同时，也希望能把积累的技术开源出去，一方面是回馈社区，贡献给整个行业生态；另一方面，让更多感兴趣的开发工程师也能参与进来，共同加速系统软件的升级与创新。所以，像 CAT 这样的优秀项目，我们将陆续开源输出并长期持续运营，保证开源软件本身的成熟度、支撑度与社区的活跃度，也欢迎大家给我们提出更多的宝贵意见和建议。

结语

这是一场没有终点的长跑，我们整个 CAT 项目组将长期有耐心地不断前行。愿同行的朋友积极参与我们，关注我们，共同打造一款企业级高可用、高可靠的分布式监控中间件产品，共同描绘 CAT 的新未来！

这次开源仅仅是一个新的起点，如果你对 CAT 新版本有一些看法以及建议，欢迎联系我们：cat@dianping.com or Github issues

招聘信息

美团点评基础架构团队诚招 Java 高级、资深技术专家，Base北京、上海。我们是集团致力于研发公司级、业界领先基础架构组件的核心团队，涵盖分布式监控、服务治理、高性能通信、消息中间件、基础存储、容器化、集群调度等技术领域。欢迎有兴趣的同学投送简历到 yong.you@dianping.com。

LruCache在美团DSP系统中的应用演进

作者: 王粲 崔涛 霜霜

背景

DSP系统是互联网广告需求方平台，用于承接媒体流量，投放广告。业务特点是并发度高，平均响应低（百毫秒）。

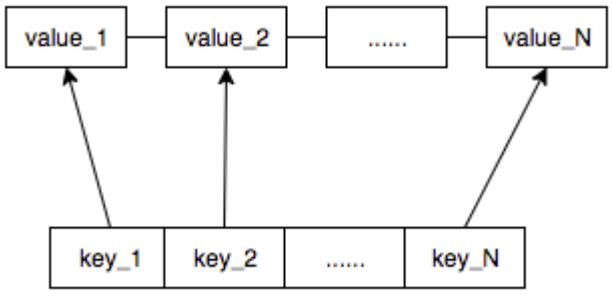
为了能够有效提高DSP系统的性能，美团平台引入了一种带有清退机制的缓存结构LruCache(Least Recently Used Cache)，在目前的DSP系统中，使用LruCache + 键值存储数据库的机制将远端数据变为本地缓存数据，不仅能够降低平均获取信息的耗时，而且通过一定的清退机制，也可以维持服务内存占用在安全区间。

本文将会结合实际应用场景，阐述引入LruCache的原因，并会在高QPS下的挑战与解决方案等方面做详细深入的介绍，希望能对DSP感兴趣的同学有所启发。

LruCache简介

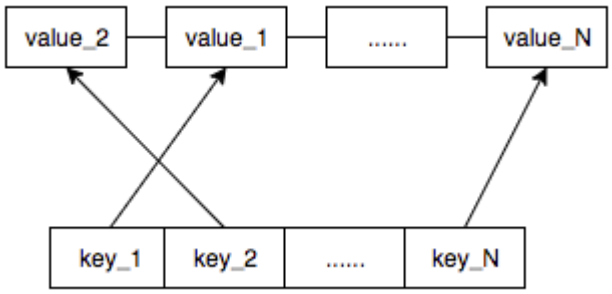
LruCache采用的缓存算法为LRU(Least Recently Used)，即最近最少使用算法。这一算法的核心思想是当缓存数据达到预设上限后，会优先淘汰近期最少使用的缓存对象。

LruCache内部维护一个双向链表和一个映射表。链表按照使用顺序存储缓存数据，越早使用的数据越靠近链表尾部，越晚使用的数据越靠近链表头部；映射表通过Key-Value结构，提供高效的查找操作，通过键值可以判断某一数据是否缓存，如果缓存直接获取缓存数据所属的链表节点，进一步获取缓存数据。LruCache结构图如下所示，上半部分是双向链表，下半部分是映射表（不一定有序）。双向链表中value_1所处位置为链表头部，value_N所处位置为链表尾部。



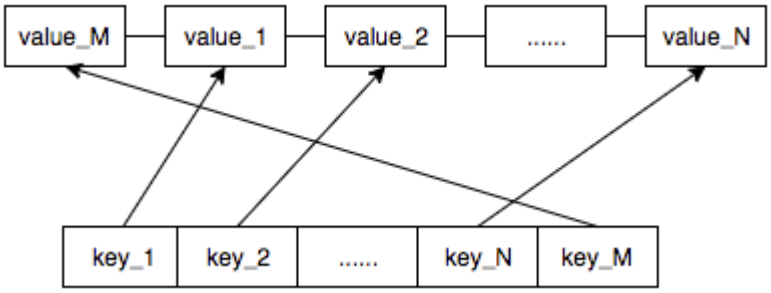
LruCache 初始结构

LruCache读操作，通过键值在映射表中查找缓存数据是否存在。如果数据存在，则将缓存数据所处节点从链表中当前位置取出，移动到链表头部；如果不存在，则返回查找失败，等待新数据写入。下图为通过LruCache查找key_2后LruCache结构的变化。



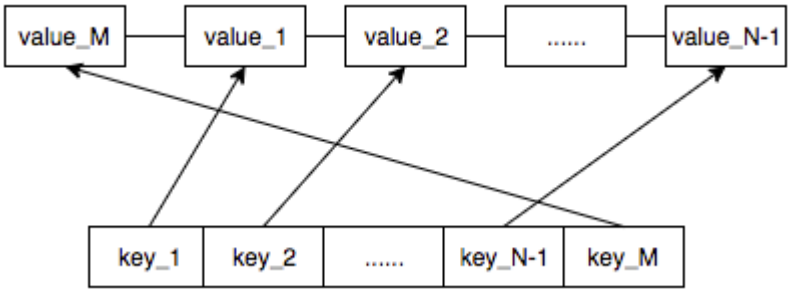
LruCache 查找

LruCache没有达到预设上限情况下的写操作，直接将缓存数据加入到链表头部，同时将缓存数据键值与缓存数据所处的双链表节点作为键值对插入到映射表中。下图是LruCache预设上限大于N时，将数据M写入后的数据结构。



LruCache 未达预设上限，添加数据

LruCache达到预设上限情况下的写操作，首先将链表尾部的缓存数据在映射表中的键值对删除，并删除链表尾部数据，再将新的数据正常写入到缓存中。下图是LruCache预设上限为N时，将数据M写入后的数据结构。



LruCache 达预设上限，添加数据

线程安全的LruCache在读写操作中，全部使用锁做临界区保护，确保缓存使用是线程安全的。

LruCache在美团DSP系统的应用场景

在美团DSP系统中广泛应用键值存储数据库，例如使用Redis存储广告信息，服务可以通过广告ID获取广告信息。每次请求都从远端的键值存储数据库中获取广告信息，请求耗时非常长。随着业务发展，QPS呈现巨大的增长趋势，在这种高并发的应用场景下，将广告信息从远端键值存储数据库中迁移到本地以减少

查询耗时是常见解决方案。另外服务本身的内存占用要稳定在一个安全的区间内。面对持续增长的广告信息，引入LruCache + 键值存储数据库的机制来达到提高系统性能，维持内存占用安全、稳定的目标。

LruCache + Redis机制的应用演进

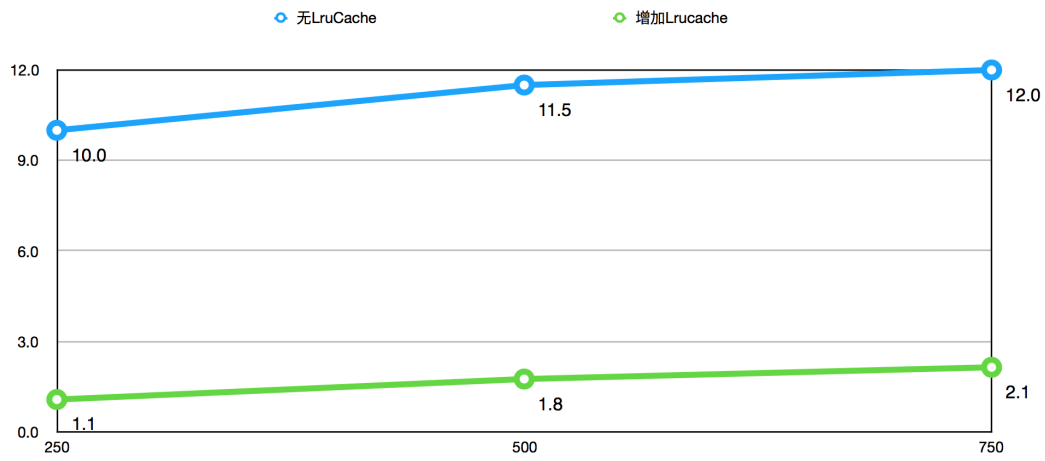
在实际应用中，LruCache + Redis机制实践分别经历了引入LruCache、LruCache增加时效清退机制、HashLruCache满足高QPS应用场景以及零拷贝机制四个阶段。各阶段的测试机器是16核16G机器。

演进一：引入LruCache提高美团DSP系统性能

在较低QPS环境下，直接请求Redis获取广告信息，可以满足场景需求。但是随着单机QPS的增加，直接请求Redis获取广告信息，耗时也会增加，无法满足业务场景的需求。

引入LruCache，将远端存放于Redis的信息本地化存储。LruCache可以预设缓存上限，这个上限可以根据服务所在机器内存与服务本身内存占用来确定，确保增加LruCache后，服务本身内存占用在安全范围内；同时可以根据查询操作统计缓存数据在实际使用中的命中率。

下图是增加LruCache结构前后，且增加LruCache后命中率高于95%的情况下，针对持续增长的QPS得出的数据获取平均耗时(ms)对比图：

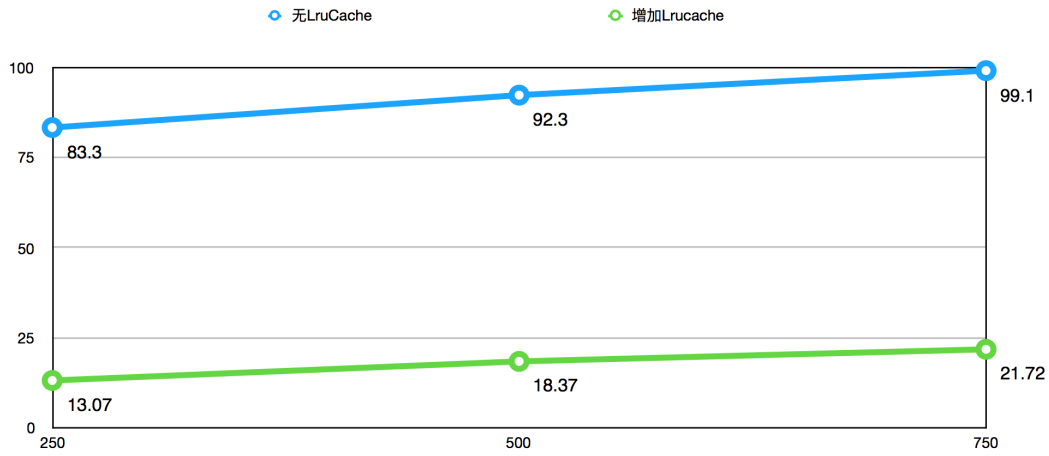


引入LruCache前后平均耗时

根据平均耗时图显示可以得出结论：

1. QPS高于250后，直接请求Redis获取数据的平均耗时达到10ms以上，完全无法满足使用的需求。
2. 增加LruCache结构后，耗时下降一个量级。从平均耗时角度看，QPS不高于500的情况下，耗时低于2ms。

下图是增加LruCache结构前后，且增加LruCache后命中率高于95%的情况下，针对持续增长的QPS得出的数据获取Top999耗时(ms)对比图：



引入LruCache前后TP999耗时

根据Top999耗时图可以得出以下结论：

1. 增加LruCache结构后，Top999耗时比平均耗时增长一个数量级。
2. 即使是较低的QPS下，使用LruCache结构的Top999耗时也是比较高的。

引入LruCache结构，在实际使用中，在一定的QPS范围内，确实可以有效减少数据获取的耗时。但是QPS超出一定范围后，平均耗时和Top999耗时都很高。所以LruCache在更高的QPS下性能还需要进一步优化。

演进二：LruCache增加时效清退机制

在业务场景中，Redis中的广告数据有可能做修改。服务本身作为数据的使用方，无法感知到数据源的变化。当缓存的命中率较高或者部分数据在较长时间内多次命中，可能出现数据失效的情况。即数据源发生了变化，但服务无法及时更新数据。针对这一业务场景，增加了时效清退机制。

时效清退机制的组成部分有三点：设置缓存数据过期时间，缓存数据单元增加时间戳以及查询中的时效性判断。缓存数据单元将数据进入LruCache的时间戳与数据一起缓存下来。缓存过期时间表示缓存单元缓存的时间上限。查询中的时效性判断表示查询时的时间戳与缓存时间戳的差值超过缓存过期时间，则强制将此数据清空，重新请求Redis获取数据做缓存。

在查询中做时效性判断可以最低程度的减少时效判断对服务的中断。当LruCache预设上限较低时，定期做全量数据清理对于服务本身影响较小。但如果LruCache的预设上限非常高，则一次全量数据清理耗时可能达到秒级甚至分钟级，将严重阻断服务本身的运行。所以将时效性判断加入到查询中，只对单一的缓存单元做时效性判断，在服务性能和数据有效性之间做了折中，满足业务需求。

演进三：高QPS下HashLruCache的应用

LruCache引入美团DSP系统后，在一段时间内较好地支持了业务的发展。随着业务的迭代，单机QPS持续上升。在更高QPS下，LruCache的查询耗时有了明显的提高，逐渐无法适应低平响的业务场景。在这种情况下，引入了HashLruCache机制以解决这个问题。

LruCache在高QPS下的耗时增加原因分析：

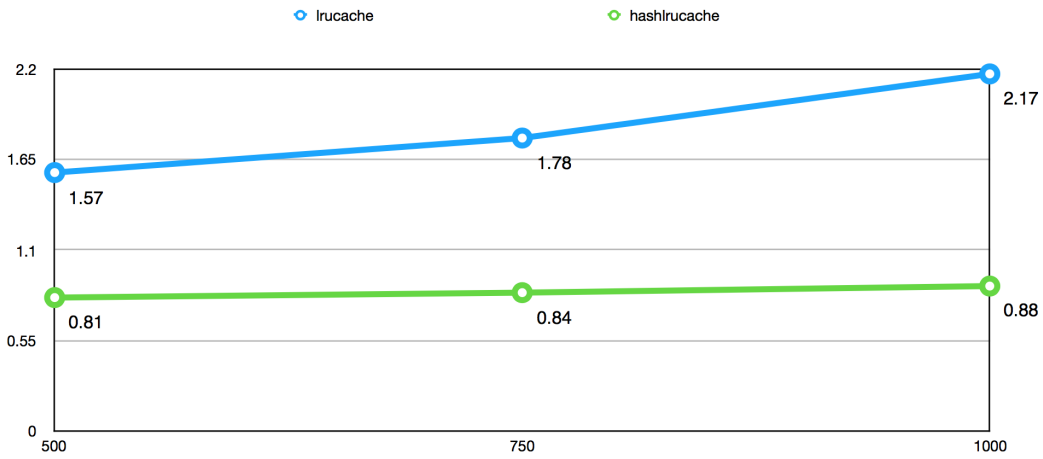
线程安全的LruCache中有锁的存在。每次读写操作之前都有加锁操作，完成读写操作之后还有解锁操作。在低QPS下，锁竞争的耗时基本可以忽略；但是在高QPS下，大量的时间消耗在了等待锁的操作上，导致耗时增长。

HashLruCache适应高QPS场景：

针对大量的同步等待操作导致耗时增加的情况，解决方案就是尽量减小临界区。引入Hash机制，对全量数据做分片处理，在原有LruCache的基础上形成HashLruCache，以降低查询耗时。

HashLruCache引入某种哈希算法，将缓存数据分散到N个LruCache上。最简单的哈希算法即使用取模算法，将广告信息按照其ID取模，分散到N个LruCache上。查询时也按照相同的哈希算法，先获取数据可能存在的分片，然后再去对应的分片上查询数据。这样可以增加LruCache的读写操作的并行度，减小同步等待的耗时。

下图是使用16分片的HashLruCache结构前后，且命中率高于95%的情况下，针对持续增长的QPS得出的数据获取平均耗时(ms)对比图：

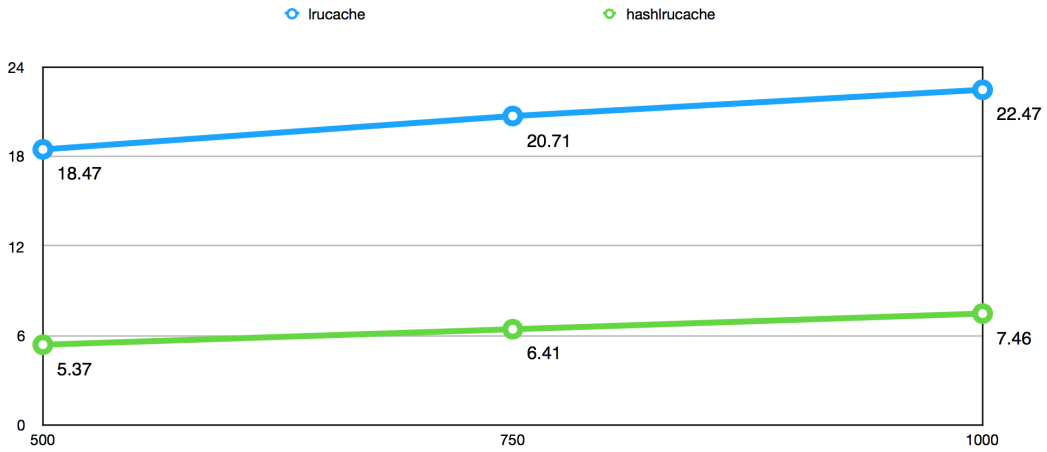


引入HashLruCache前后平均耗时

根据平均耗时图可以得出以下结论：

1. 使用HashLruCache后，平均耗时减少将近一半，效果比较明显。
2. 对比不使用HashLruCache的平均耗时可以发现，使用HashLruCache的平均耗时对QPS的增长不敏感，没有明显增长。

下图是使用16分片的HashLruCache结构前后，且命中率高于95%的情况下，针对持续增长的QPS得出的数据获取Top999耗时(ms)对比图：



引入HashLruCache前后TP999耗时

根据Top999耗时图可以得出以下结论：

1. 使用HashLruCache后，Top999耗时减少为未使用时的三分之一左右，效果非常明显。
2. 使用HashLruCache的Top999耗时随QPS增长明显比不使用的情况慢，相对来说对QPS的增长敏感度更低。

引入HashLruCache结构后，在实际使用中，平均耗时和Top999耗时都有非常明显的下降，效果非常显著。

HashLruCache分片数量确定：

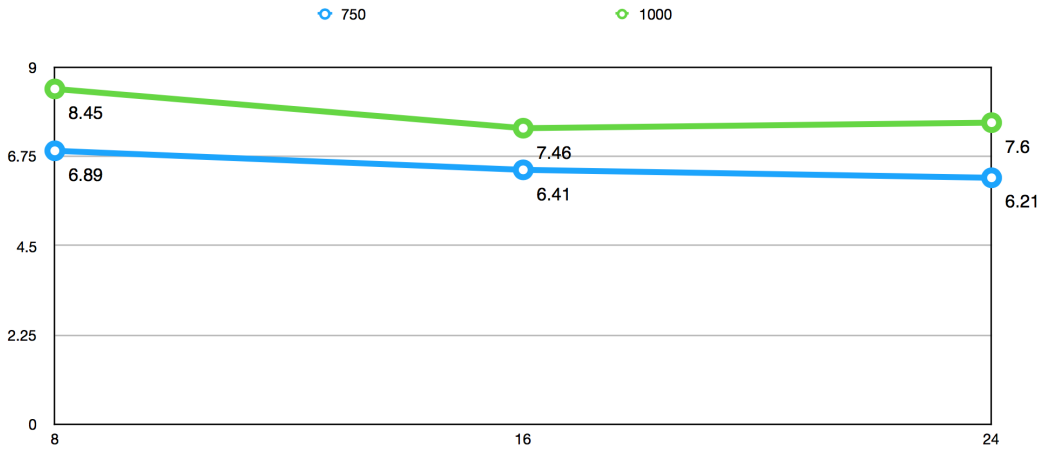
根据以上分析，进一步提高HashLruCache性能的一个方法是确定最合理的分片数量，增加足够的并行度，减少同步等待消耗。所以分片数量可以与CPU数量一致。由于超线程技术的使用，可以将分片数量进一步提高，增加并行性。

下图是使用HashLruCache机制后，命中率高于95%，不同分片数量在不同QPS下得出的数据获取平均耗时(ms)对比图：

HashLruCache分片数量耗时

平均耗时图显示，在较高的QPS下，平均耗时并没有随着分片数量的增加而有明显的减少，基本维持稳定的状态。

下图是使用HashLruCache机制后，命中率高于95%，不同分片数量在不同QPS下得出的数据获取Top999耗时(ms)对比图：



HashLruCache分片TP99耗时

Top999耗时图显示，QPS为750时，分片数量从8增长到16再增长到24时，Top999耗时有一定的下降，并不显著；QPS为1000时，分片数量从8增长到16有明显下降，但是从16增长到24时，基本维持了稳定状态。明显与实际使用的机器CPU数量有较强的相关性。

HashLruCache机制在实际使用中，可以根据机器性能并结合实际场景的QPS来调节分片数量，以达到最好的性能。

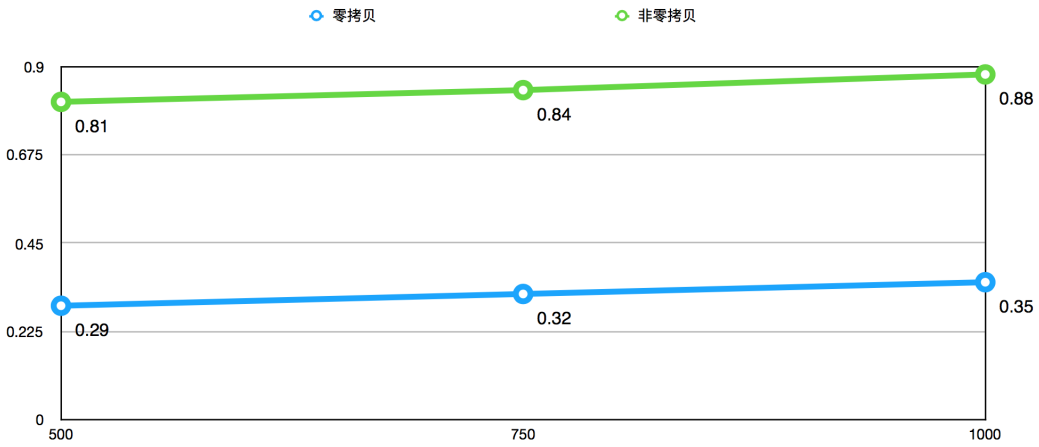
演进四：零拷贝机制

线程安全的LruCache内部维护一套数据。对外提供数据时，将对应的数据完整拷贝一份提供给调用方使用。如果存放结构简单的数据，拷贝操作的代价非常小，这一机制不会成为性能瓶颈。但是美团DSP系统的应用场景中，LruCache中存放的数据结构非常复杂，单次的拷贝操作代价很大，导致这一机制变成了性能瓶颈。

理想的情况是LruCache对外仅仅提供数据地址，即数据指针。使用方在业务需要使用的地方通过数据指针获取数据。这样可以将复杂的数据拷贝操作变为简单的地址拷贝，大量减少拷贝操作的性能消耗，即数据的零拷贝机制。直接的零拷贝机制存在安全隐患，即由于LruCache中的时效清退机制，可能会出现某一数据已经过期被删除，但是使用方仍然通过持有失效的数据指针来获取该数据。

进一步分析可以确定，以上问题的核心是存放于LruCache的数据生命周期对于使用方不透明。解决这一问题的方案是为LruCache中存放的数据添加原子变量的引用计数。使用原子变量不仅确保了引用计数的线程安全，使得各个线程读取的引用计数一致，同时保证了并发状态最小的同步性能开销。不论是LruCache中还是使用方，每次获取数据指针时，即将引用计数加1；同理，不再持有数据指针时，引用计数减1。当引用计数为0时，说明数据没有被任何使用方使用，且数据已经过期从LruCache中被删除。这时删除数据的操作是安全的。

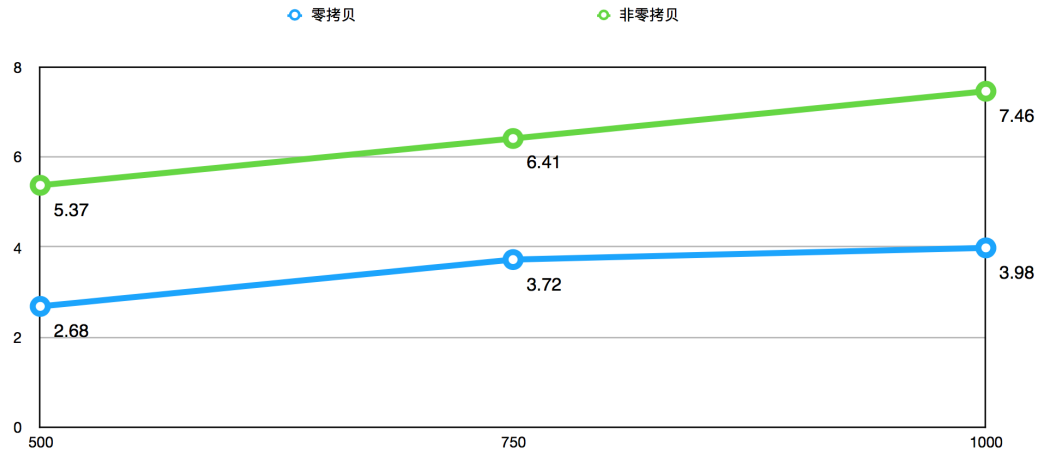
下图是使零拷贝机制后，命中率高于95%，不同QPS下得出的数据获取平均耗时(ms)对比图：



HashLruCache分片数量耗时

平均耗时图显示，使用零拷贝机制后，平均耗时下降幅度超过60%，效果非常显著。

下图是使零拷贝机制后，命中率高于95%，不同QPS下得出的数据获取Top999耗时(ms)对比图：



HashLruCache分片数量耗时

根据Top999耗时图可以得出以下结论：

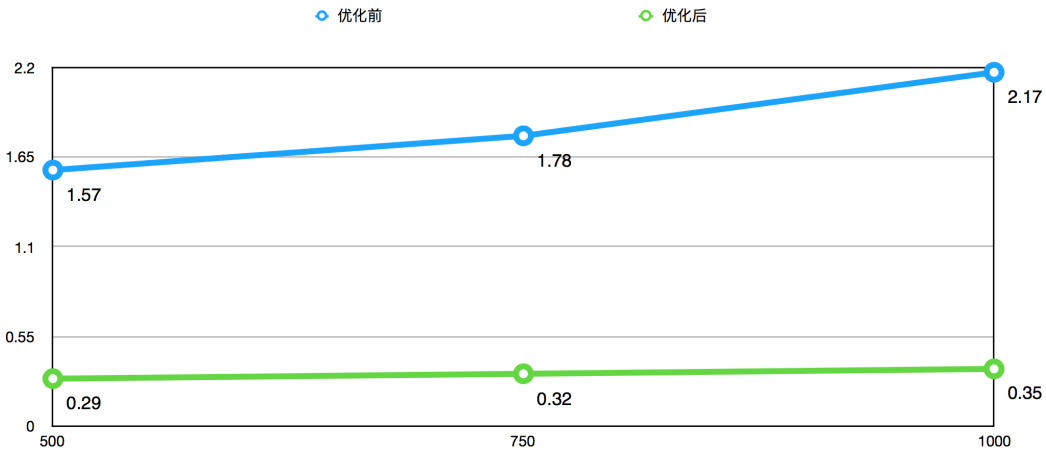
1. 使用零拷贝后，Top999耗时降幅将近50%，效果非常明显。
2. 在高QPS下，使用零拷贝机制的Top999耗时随QPS增长明显比不使用的情况慢，相对来说对QPS的增长敏感度更低。

引入零拷贝机制后，通过拷贝指针替换拷贝数据，大量降低了获取复杂业务数据的耗时，同时将临界区减小到最小。线程安全的原子变量自增与自减操作，目前在多个基础库中都有实现，例如C++11就提供了内置的整型原子变量，实现线程安全的自增与自减操作。

在HashLruCache中引入零拷贝机制，可以进一步有效降低平均耗时和Top999耗时，且在高QPS下对于稳定Top999耗时有非常好的效果。

总结

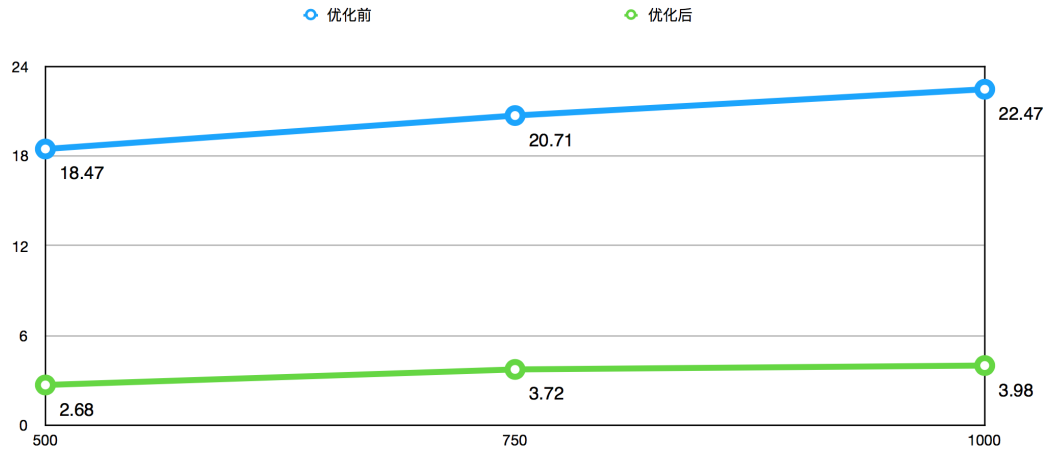
下图是一系列优化措施前后，命中率高于95%，不同QPS下得出的数据获取平均耗时(ms)对比图：



HashLruCache分片数量耗时

平均耗时图显示，优化后的平均耗时仅为优化前的20%以内，性能提升非常明显。优化后平均耗时对于QPS的增长敏感度更低，更好的支持了高QPS的业务场景。

下图是一系列优化措施前后，命中率高于95%，不同QPS下得出的数据获取Top999耗时(ms)对比图：



HashLruCache分片数量耗时

Top999耗时图显示，优化后的Top999耗时仅为优化前的20%以内，对于长尾请求的耗时有非常明显的降低。

LruCache是一个非常常见的数据结构。在美团DSP的高QPS业务场景下，发挥了重要的作用。为了符合业务需要，在原本的清退机制外，补充了时效性强制清退机制。随着业务的发展，针对更高QPS的业务场景，使用HashLruCache机制，降低缓存的查询耗时。针对不同的具体场景，在不同的QPS下，不断尝试更合理的分片数量，不断提高HashLruCache的查询性能。通过引用计数的方案，在HashLruCache中引入零拷贝机制，进一步大幅降低平均耗时和Top999耗时，更好的服务于业务场景的发展。

作者简介

- 王粲，2018年11月加入美团，任职美团高级工程师，负责美团DSP系统后端基础架构的研发工作。
- 崔涛，2015年6月加入美团，任职资深广告技术专家，期间一手指导并从0到1搭建美团DSP投放平台，具备丰富的大规模计算引擎的开发和性能优化经验。

- 霜霜，2015年6月加入美团，任职美团高级工程师，美团DSP系统后端基础架构与机器学习架构负责人，全面负责DSP业务广告召回和排序服务的架构设计与优化。

招聘

美团在线营销DSP团队诚招工程、算法、数据等各方向精英，发送简历至cuitao@meituan.com，共同支持百亿级流量的高可靠系统研发与优化。

Netty堆外内存泄露排查盛宴

作者: 闪电侠

导读

Netty 是一个异步事件驱动的网络通信层框架，用于快速开发高可用高性能的服务端网络框架与客户端程序，它极大地简化了 TCP 和 UDP 套接字服务器等网络编程。

Netty 底层基于 JDK 的 NIO，我们为什么不直接基于 JDK 的 NIO 或者其他NIO框架：

1. 使用 JDK 自带的 NIO 需要了解太多的概念，编程复杂。
2. Netty 底层 IO 模型随意切换，而这一切只需要做微小的改动。
3. Netty自带的拆包解包，异常检测等机制让我们从 NIO 的繁重细节中脱离出来，只需关心业务逻辑即可。
4. Netty解决了JDK 的很多包括空轮训在内的 Bug。
5. Netty底层对线程，Selector 做了很多细小的优化，精心设计的 Reactor 线程做到非常高效的并发处理。
6. 自带各种协议栈，让我们处理任何一种通用协议都几乎不用亲自动手。
7. Netty社区活跃，遇到问题随时邮件列表或者 issue。
8. Netty已经历各大RPC框架（Dubbo），消息中间件（RocketMQ），大数据通信（Hadoop）框架的广泛的线上验证，健壮性无比强大。

背景

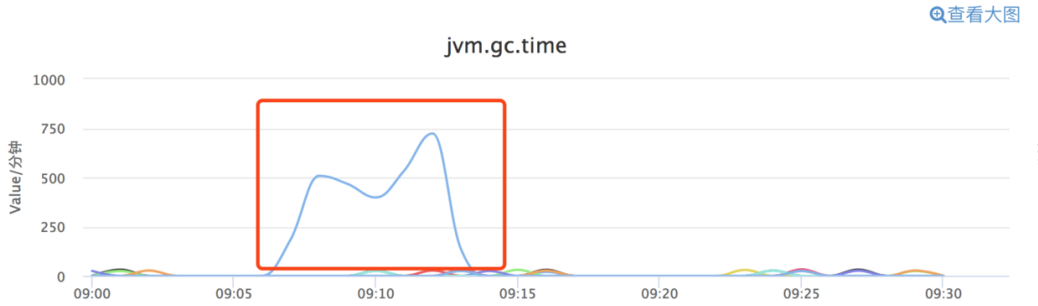
最近在做一个基于 Websocket 的长连中间件，服务端使用实现了 Socket.IO 协议（基于WebSocket协议，提供长轮询降级能力） 的 netty-socketio 框架，该框架为 Netty 实现，鉴于本人对 Netty 比较熟，并且对比同样实现了 Socket.IO 协议的其他框架，Netty 的口碑都要更好一些，因此选择这个框架作为底层核心。

诚然，任何开源框架都避免不了 Bug 的存在，我们在使用这个开源框架时，就遇到一个堆外内存泄露的 Bug。美团的价值观一直都是“追求卓越”，所以我们就想挑战一下，找到那只臭虫（Bug），而本文就是遇到的问题以及排查的过程。当然，想看结论的同学可以直接跳到最后，阅读总结即可。

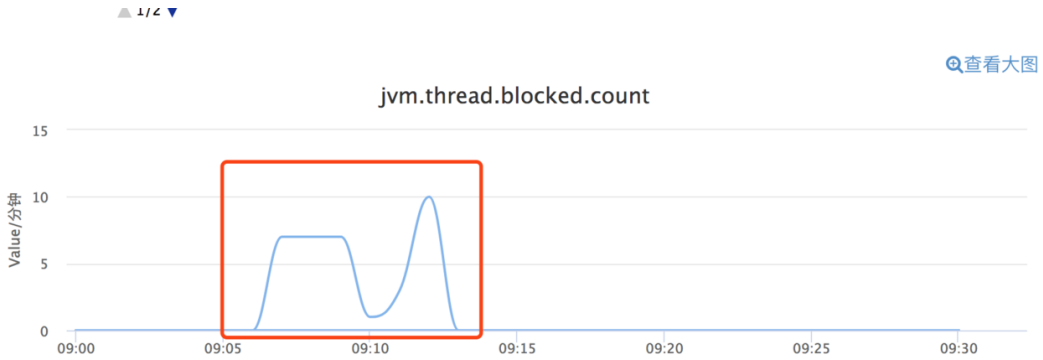
问题

某天早上，我们突然收到告警，Nginx 服务端出现大量5xx。

我们使用 Nginx 作为服务端 WebSocket 的七层负载，5xx的爆发通常表明服务端不可用。由于目前 Nginx 告警没有细分具体哪台机器不可用，接下来，我们就到 CAT（美团点评统一监控平台，目前已经开源）去检查一下整个集群的各项指标，就发现如下两个异常:



某台机器在同一时间点爆发 GC（垃圾回收），而且在同一时间，JVM 线程阻塞。



接下来，我们就就开始了漫长的堆外内存泄露“排查之旅”。

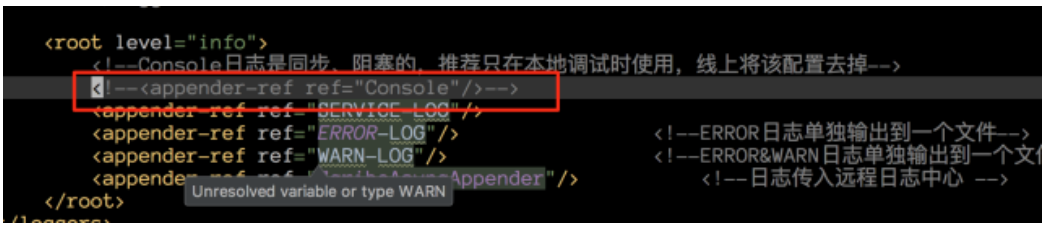
排查过程

阶段1: 怀疑是log4j2

因为线程被大量阻塞，我们首先想到的是定位哪些线程被阻塞，最后查出来是 Log4j2 狂打日志导致 Netty 的 NIO 线程阻塞（由于没有及时保留现场，所以截图缺失）。NIO 线程阻塞之后，因我们的服务

器无法处理客户端的请求，所以对Nginx来说就是5xx。

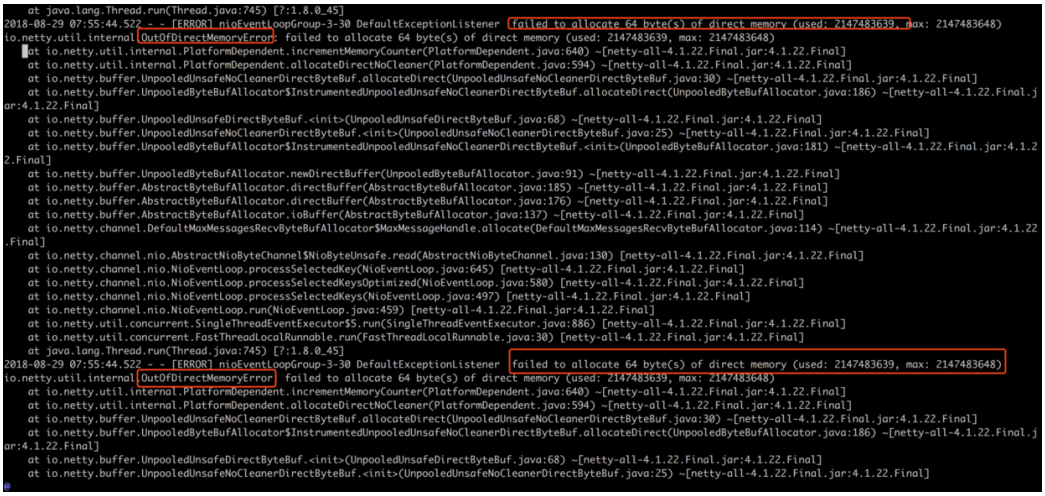
接下来，我们查看了 Log4j2 的配置文件。



我们发现打印到控制台的这个 appender 忘记注释掉了，所以初步猜测：因为这个项目打印的日志过多，而 Log4j2 打印到控制台是同步阻塞打印的，所以就导致了这个问题。那么接下来，我们把线上所有机器的这行注释掉，本以为会“大功告成”，但没想到仅仅过了几天，5xx告警又来“敲门”。看来，这个问题并没我们最初想象的那么简单。

阶段2：可疑日志浮现

接下来，我们只能硬着头皮去查日志，特别是故障发生点前后的日志，于是又发现了一处可疑的地方：



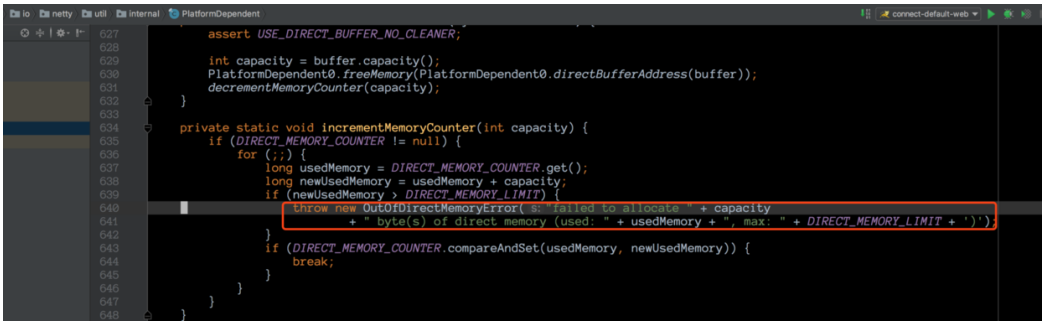
可以看到：在极短的时间内，狂打 `failed to allocate 64(bytes) of direct memory(...)` 日志（瞬间十几个日志文件，每个日志文件几百M），日志里抛出一个 Netty 自己封装的 `OutOfDirectMemoryError` 。说白了，就是堆外内存不够用，Netty 一直在“喊冤”。

堆外内存泄露，听到这个名词就感到很沮丧。因为这个问题的排查就像 C 语言内存泄露一样难以排查，首先能想到的就是，在 OOM 爆发之前，查看有无异常。然后查遍了 CAT 上与机器相关的所有指标，查遍了 OOM 日志之前的所有日志，均未发现任何异常！这个时候心里已经“万马奔腾”了……

阶段3：定位OOM源

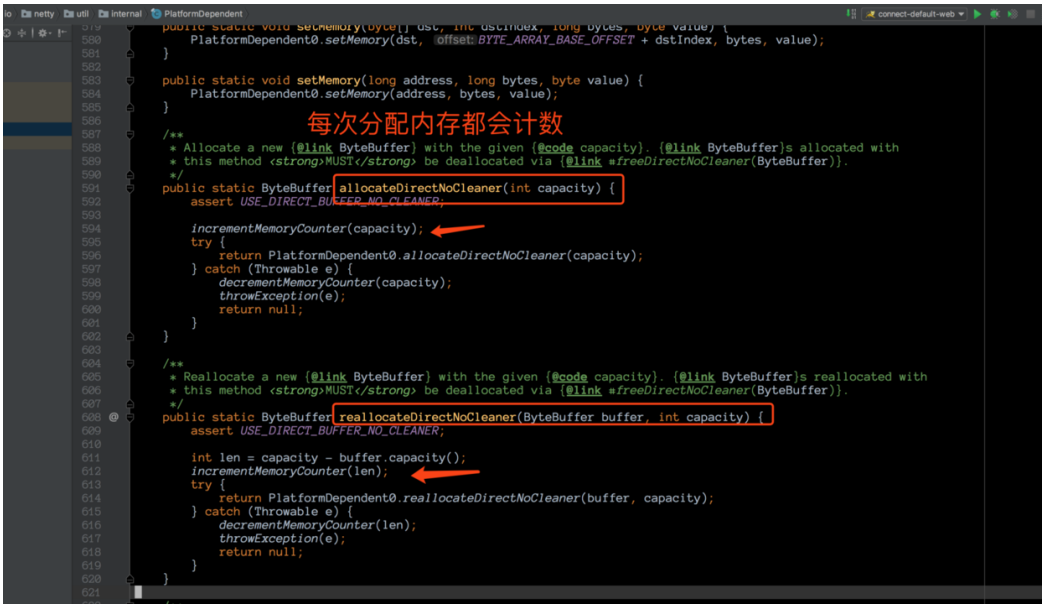
没办法，只能看着这堆讨厌的 OOM 日志发着呆，希望答案能够“蹦到”眼前，但是那只是妄想。一筹莫展之际，突然一道光在眼前一闪而过，在 OOM 下方的几行日志变得耀眼起来（为啥之前就没想认真查看日志？估计是被堆外内存泄露这几个词吓怕了吧 ==！），这几行字是 `....PlatformDepedeng.incrementMemory()...` 。

原来，堆外内存是否够用，是 Netty 这边自己统计的，那么是不是可以找到统计代码，找到统计代码之后我们就可以看到 Netty 里面的对外内存统计逻辑了？于是，接下来翻翻代码，找到这段逻辑，就在 `PlatformDepedent` 这个类里面。



这个地方，是一个对已使用堆外内存计数的操作，计数器为 `DIRECT_MEMORY_COUNTER`，如果发现已使用内存大于堆外内存的上限（用户自行指定），就抛出一个自定义 OOM Error，异常里面的文本内容正是我们在日志里面看到的。

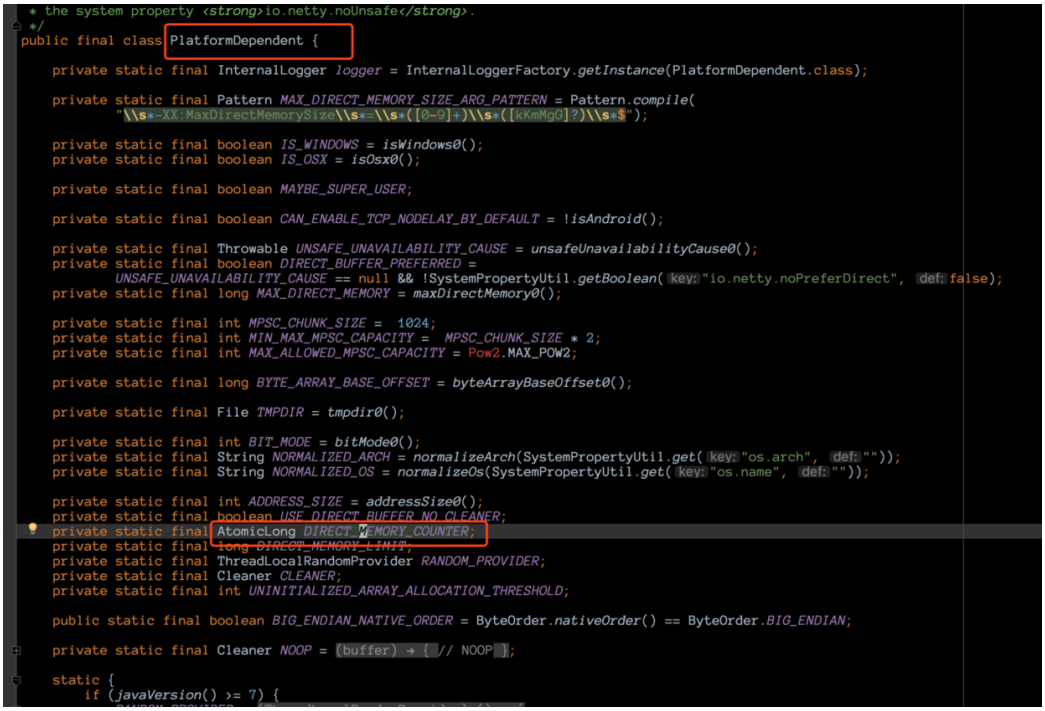
接下来，就验证一下这个方法是否是在堆外内存分配的时候被调用。



果然，在 Netty 每次分配堆外内存之前，都会计数。想到这，思路就开始慢慢清晰，而心情也开始从“秋风瑟瑟”变成“春光明媚”。

阶段4：反射进行堆外内存监控

CAT 上关于堆外内存的监控没有任何异常（应该是没有统计准确，一直维持在 1M），而这边我们又确认堆外内存已快超过上限，并且已经知道 Netty 底层是使用的哪个字段来统计。那么接下来要做的第一件事情，就是反射拿到这个字段，然后我们自己统计 Netty 使用堆外内存的情况。



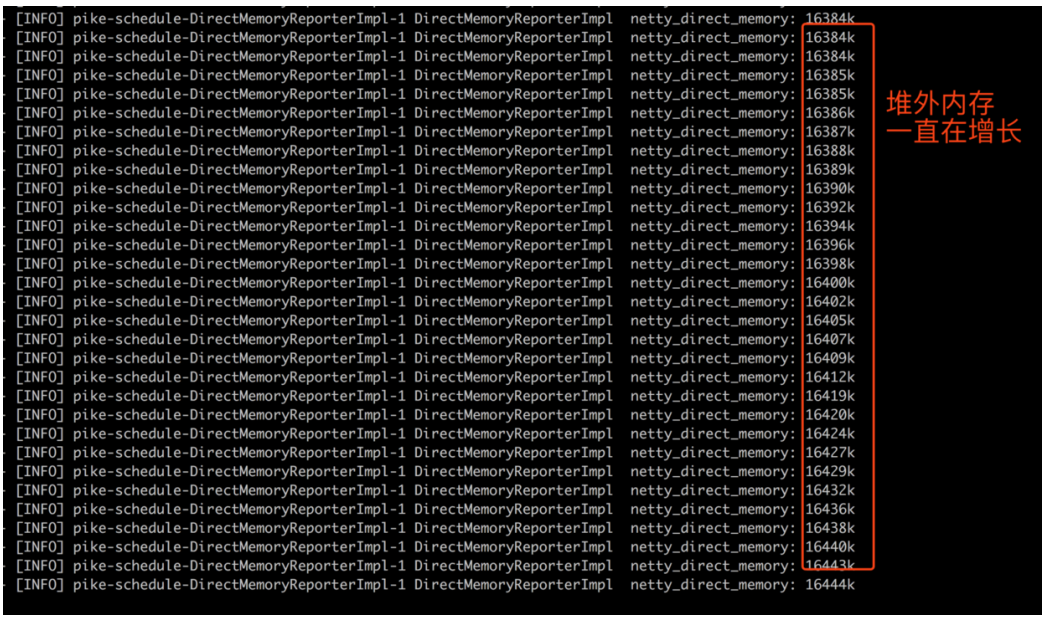
堆外内存统计字段是 `DIRECT_MEMORY_COUNTER`，我们可以通过反射拿到这个字段，然后定期 Check 这个值，就可以监控 Netty 堆外内存的增长情况。

于是我们通过反射拿到这个字段，然后每隔一秒打印，为什么要这样做?

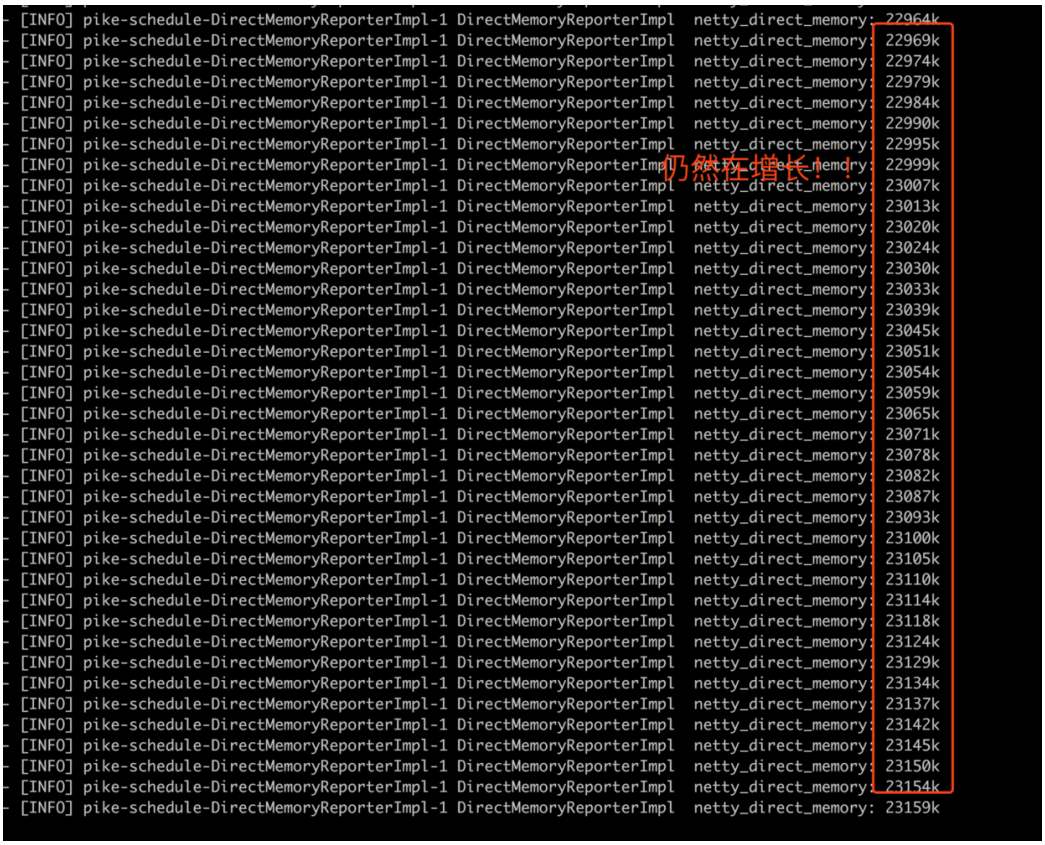
因为，通过我们前面的分析，在爆发大量 OOM 现象之前，没有任何可疑的现象。那么只有两种情况，一种是突然某个瞬间分配了大量的堆外内存导致OOM；一种是堆外内存缓慢增长，到达某个点之后，最后一根稻草将机器压垮。在这段代码加上去之后，我们打包上线。

阶段5：到底是缓慢增长还是瞬间飙升?

代码上线之后，初始内存为 16384k（16M），这是因为线上我们使用了池化堆外内存，默认一个 chunk 为16M，这里不必过于纠结。

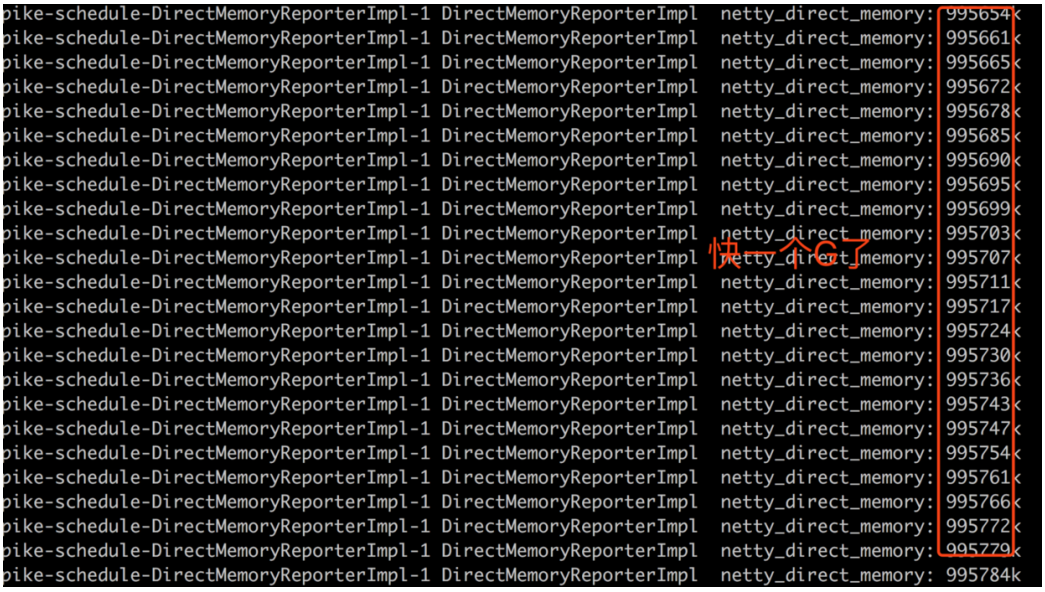


但是没过一会，内存就开始缓慢飙升，并且没有释放的迹象，二十几分钟之后，内存使用情况如下：



走到这里，我们猜测可能是前面提到的第二种情况，也就是内存缓慢增长造成的 OOM，由于内存实在增长太慢，于是调整机器负载权重为其他机器的两倍，但是仍然是以数K级别在持续增长。那天刚好是周五，索性就过一个周末再开看。

周末之后，我们到公司第一时间就连上了跳板机，登录线上机器，开始 tail –f 继续查看日志。在输完命令之后，怀着期待的心情重重的敲下了回车键：

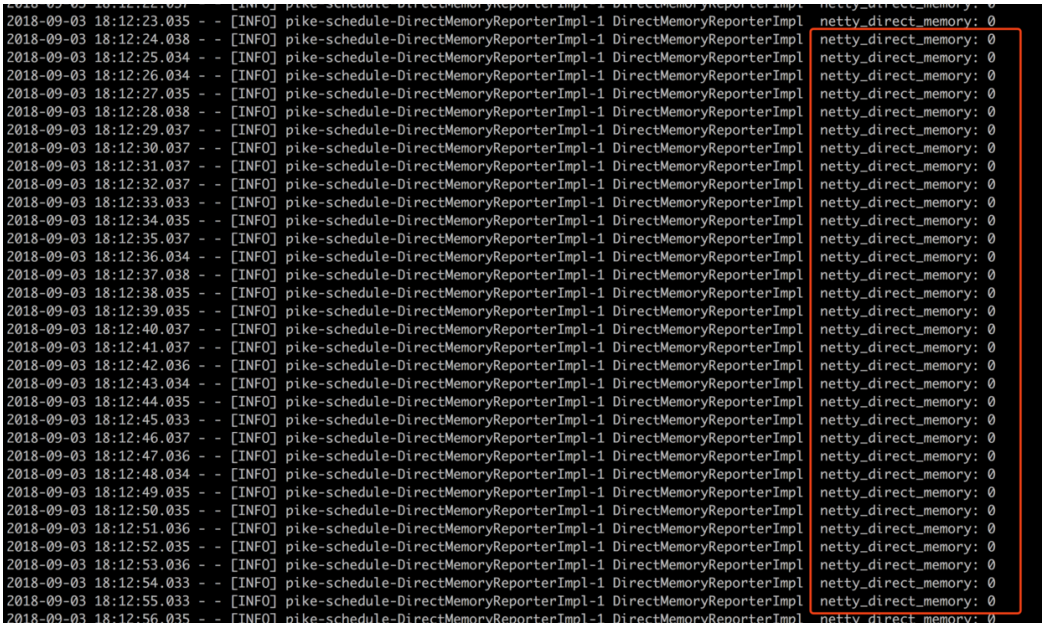


果然不出所料，内存一直在缓慢增长，一个周末的时间，堆外内存已经飙到快一个 G 了。这个时候，我竟然想到了一句成语：“只要功夫深，铁杵磨成针”。虽然堆外内存以几个K的速度在缓慢增长，但是只要一直持续下去，总有把内存打爆的时候（线上堆外内存上限设置的是2G）。

此时，我们开始自问自答环节：内存为啥会缓慢增长，伴随着什么而增长？因为我们的应用是面向用户端的WebSocket，那么，会不会是每一次有用户进来，交互完之后离开，内存都会增长一些，然后不释放呢？带着这个疑问，我们开始了线下模拟过程。

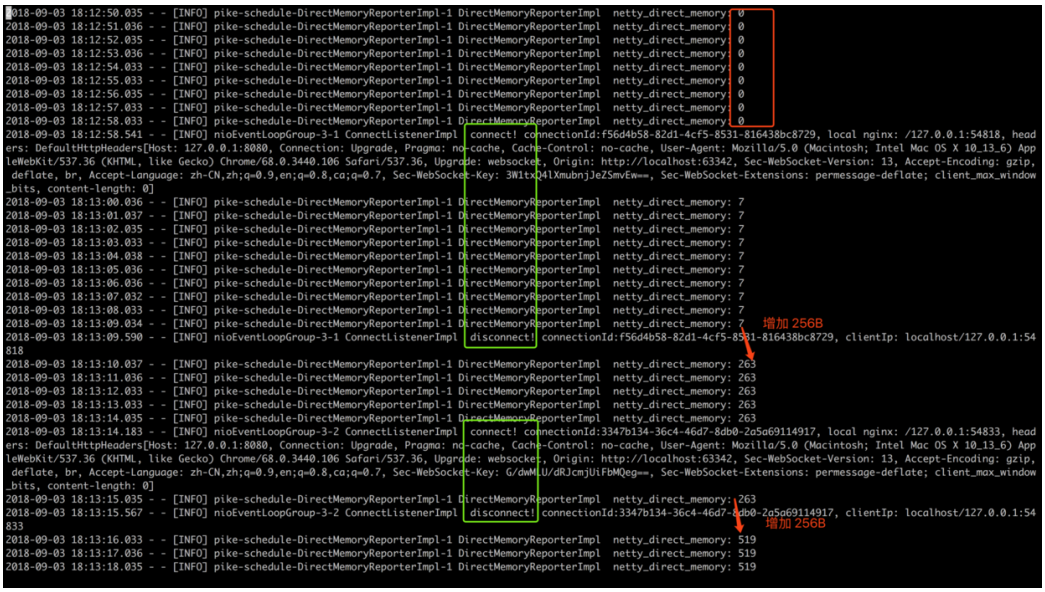
阶段6：线下模拟

本地起好服务，把监控堆外内存的单位改为以B为单位（因为本地流量较小，打算一次一个客户端连接），另外，本地也使用非池化内存（内存数字较小，容易看出问题），在服务端启动之后，控制台打印信息如下



在没有客户端接入的时候，堆外内存一直是0，在意料之中。接下来，怀着着无比激动的心情，打开浏览器，然后输入网址，开始我们的模拟之旅。

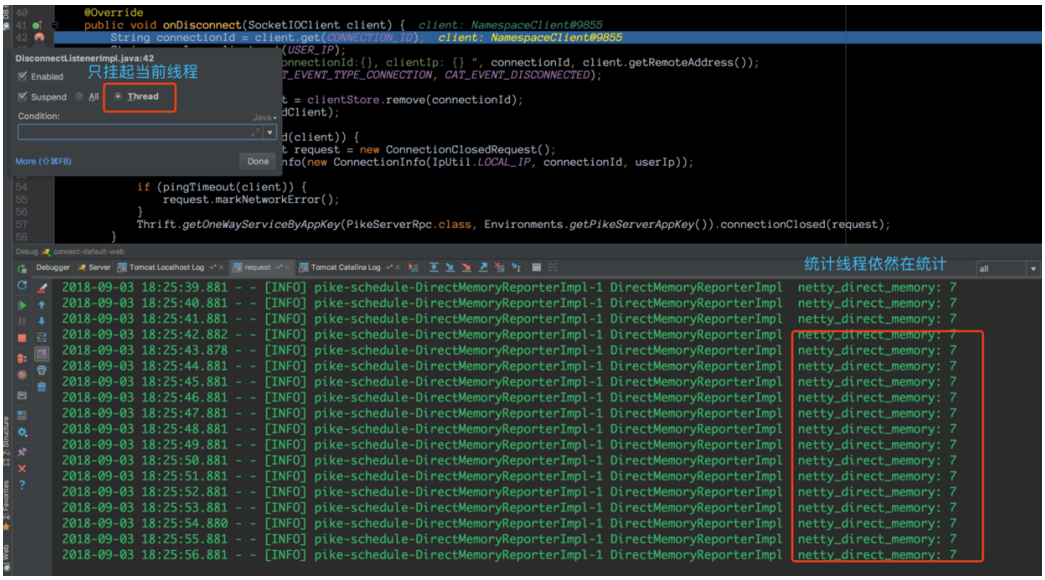
我们的模拟流程是：新建一个客户端链接->断开链接->再新建一个客户端链接->再断开链接。



如上图所示，一次 Connect 和 Disconnect 为一次连接的建立与关闭，上图绿色框框的日志分别是两次连接的生命周期。我们可以看到，内存每次都是在连接被关闭的的时候暴涨 256B，然后也不释放。走到这里，问题进一步缩小，肯定是连接被关闭的时候，触发了框架的一个Bug，而且这个Bug在触发之前分配了 256B 的内存，随着Bug被触发，内存也没有释放。问题缩小之后，接下来开始“撸源码”，捉虫！

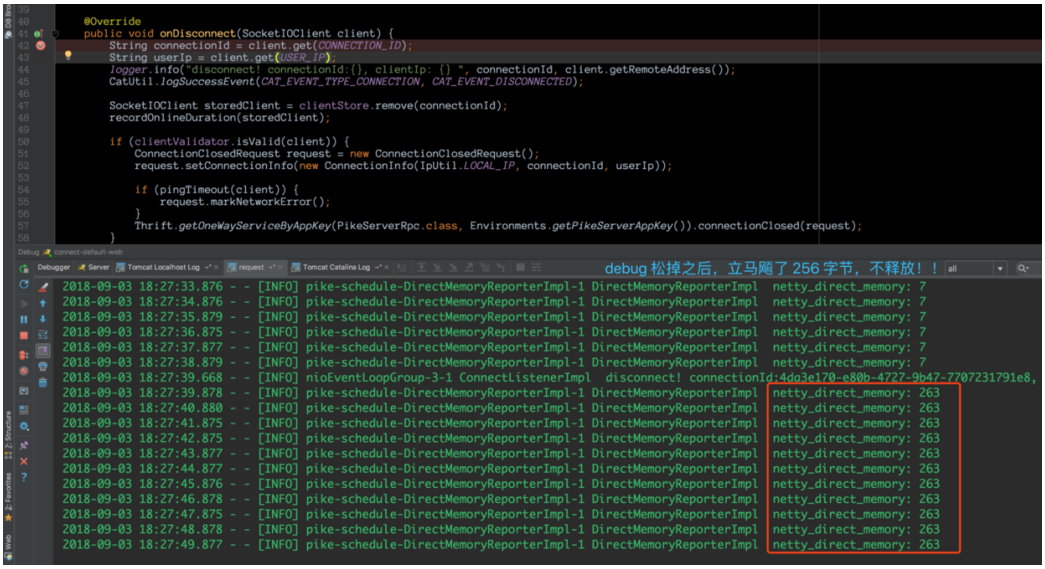
阶段7：线下排查

接下来，我们将本地服务重启，开始完整的线下排查过程。同时将目光定位到 netty-socketio 这个框架的 Disconnect 事件（客户端WebSocket连接关闭时会调用到这里），基本上可以确定，在 Disconnect 事件前后申请的内存并没有释放。



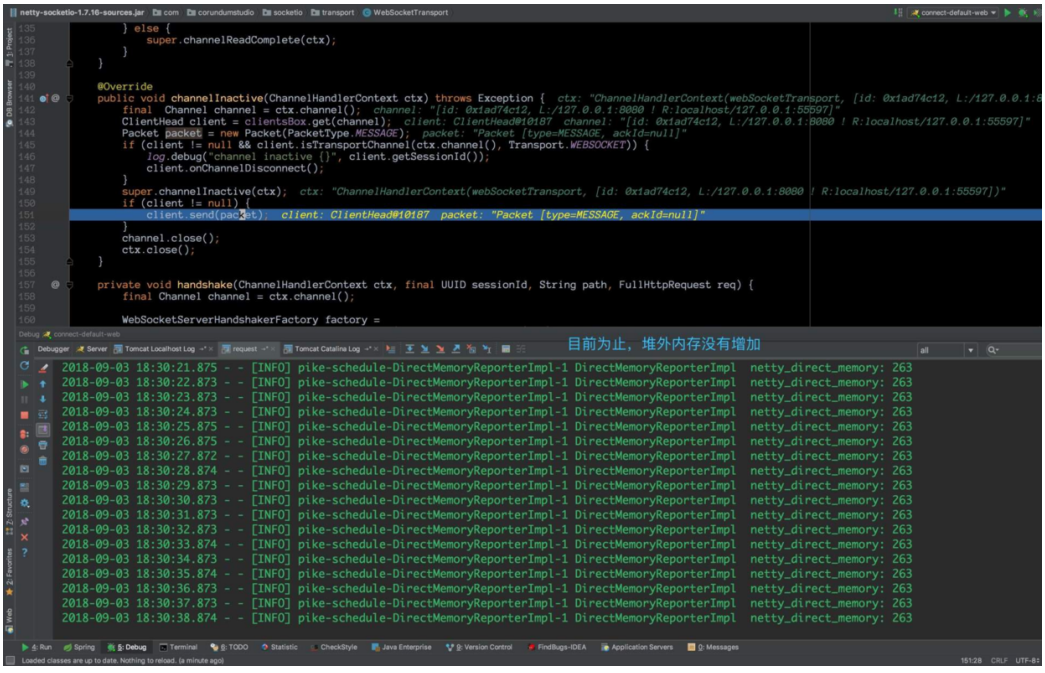
在使用 idea debug 时，要选择只挂起当前线程，这样我们在单步跟踪的时候，控制台仍然可以看到堆外内存统计线程在打印日志。

在客户端连接上之后然后关闭，断点进入到 `onDisconnect` 回调，我们特意在此多停留了一会，发现控制台内存并没有飙升（7B这个内存暂时没有去分析，只需要知道，客户端连接断开之后，我们断点hold住，内存还未开始涨）。接下来，神奇的一幕出现了，我们将断点放开，让程序跑完：

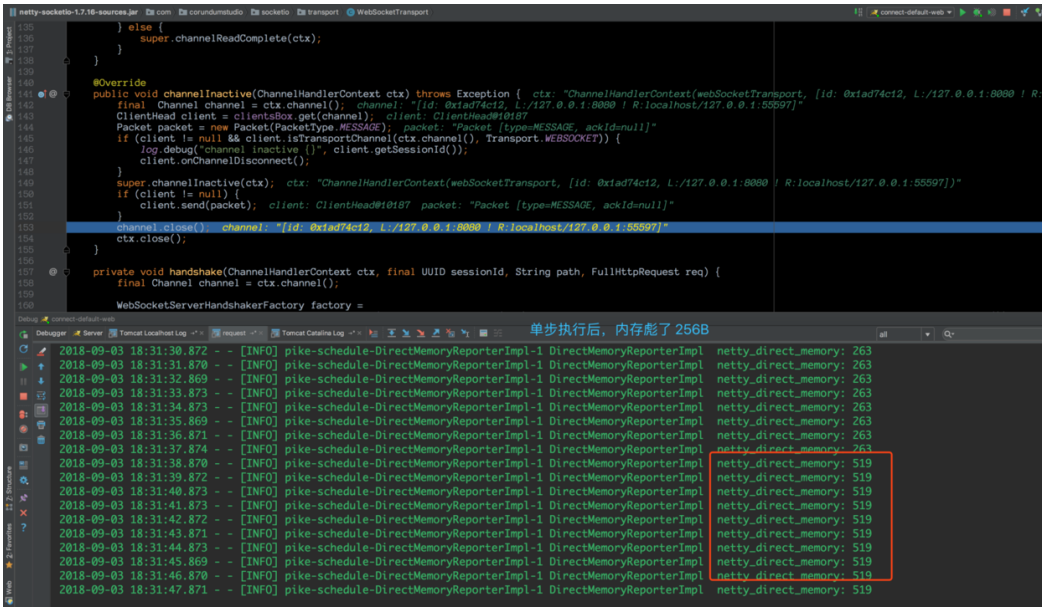


Debug 松掉之后，内存立马飙升了！！此时，我们已经知道，这只“臭虫”飞不了多远了。在 Debug 时，挂起的是当前线程，那么肯定是当前线程某个地方申请了堆外内存，然后没有释放，继续“快马加鞭“，深入源码。

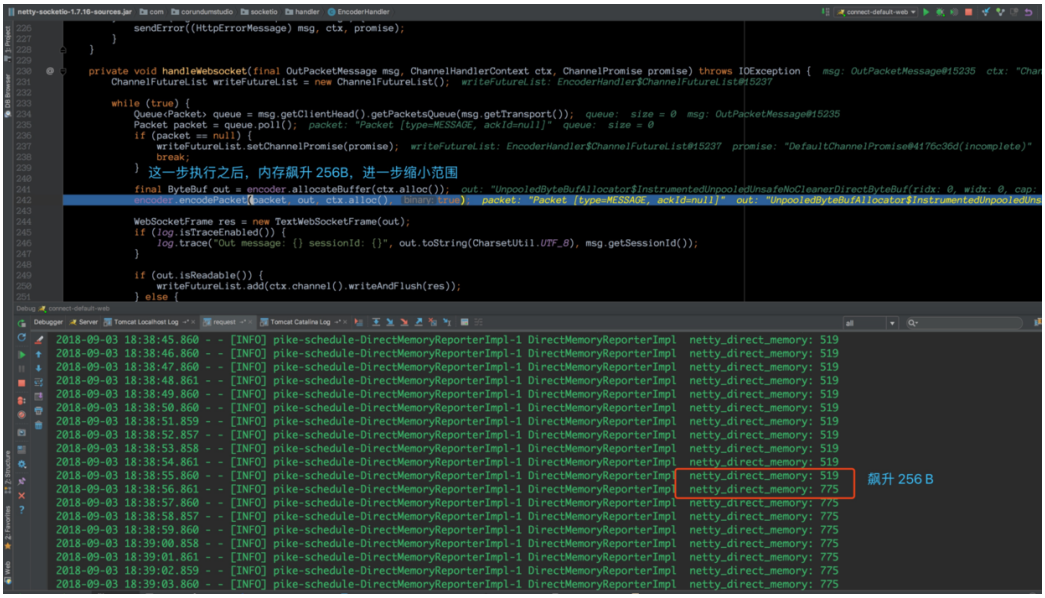
其实，每一次单步调试，我们都会观察控制台的内存飙升的情况。很快，我们来到了这个地方：



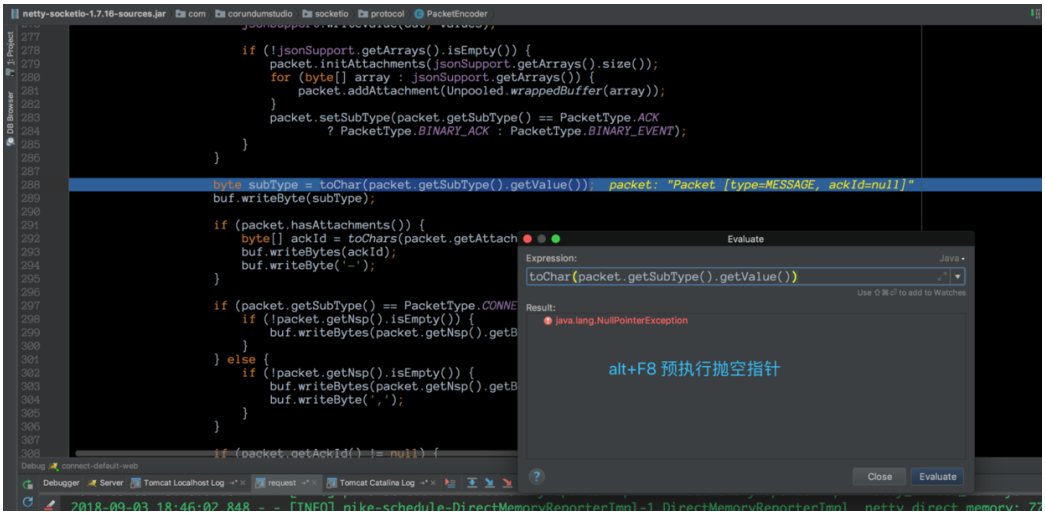
在这一行没执行之前，控制台的内存依然是 263B。然后，当执行完该行之后，立刻从 263B涨到519B（涨了256B）。



于是，Bug 范围进一步缩小。我们将本次程序跑完，释然后客户端再来一次连接，断点打在 `client.send()` 这行， 然后关闭客户端连接，之后直接进入到这个方法，随后的过程有点长，因为与 Netty 的时间传播机制有关，这里就省略了。最后，我们跟踪到了如下代码， `handleWebsocket` ：



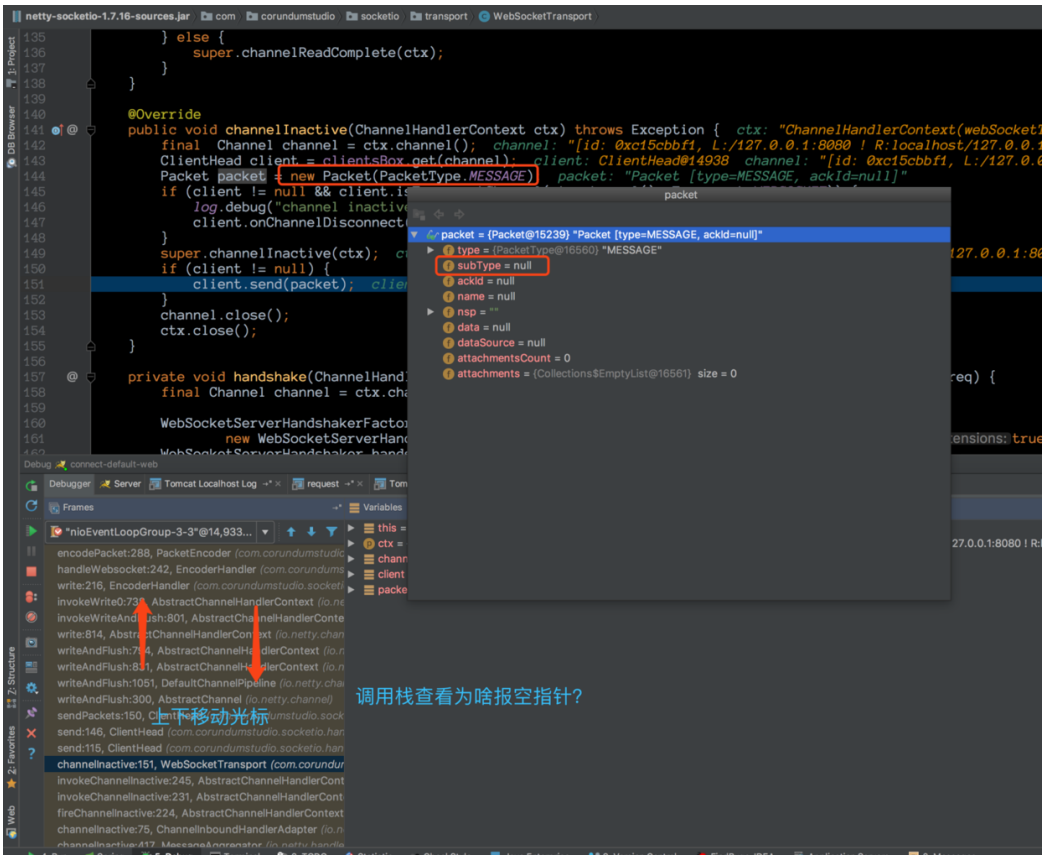
在这个地方，我们看到一处非常可疑的地方，在上图的断点上一行，调用 `encoder` 分配了一段内存，调用完之后，我们的控制台立马就彪了 256B。所以，我们怀疑肯定是这里申请的内存没有释放，它这里接下来调用 `encoder.encodePacket()` 方法，猜想是把数据包的内容以二进制的方式写到这段 256B的内存。接下来，我们追踪到这段 encode 代码，单步执行之后，就定位到这行代码：



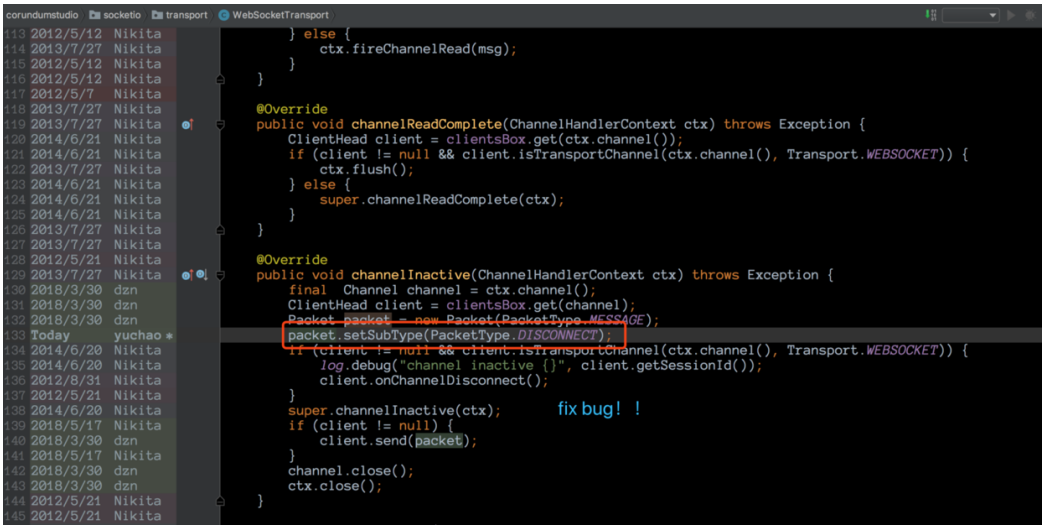
这段代码是把 packet 里面一个字段的值转换为一个 char。然而，当我们使用 idea 预执行的时候，却抛出类一个愤怒的 NPE！！也就是说，框架申请到一段内存之后，在 encoder 的时候，自己 GG 了，还给自己挖了个NPE的深坑，最后导致内存无法释放（最外层有堆外内存释放逻辑，现在无法执行到了）。而且越攒越多，直到被“最后一根稻草”压垮，堆外内存就这样爆了。这里的源码，有兴趣的读者可以自己去分析一下，限于篇幅原因，这里就不再展开叙述了。

阶段8：Bug解决

既然 Bug 已经找到，接下来就要解决问题了。这里只需要解决这个NPE异常，就可以 Fix 掉。我们的目标就是，让这个 `subType` 字段不为空。于是我们先通过 idea 的线程调用栈，定位到这个 packet 是在哪个地方定义的：



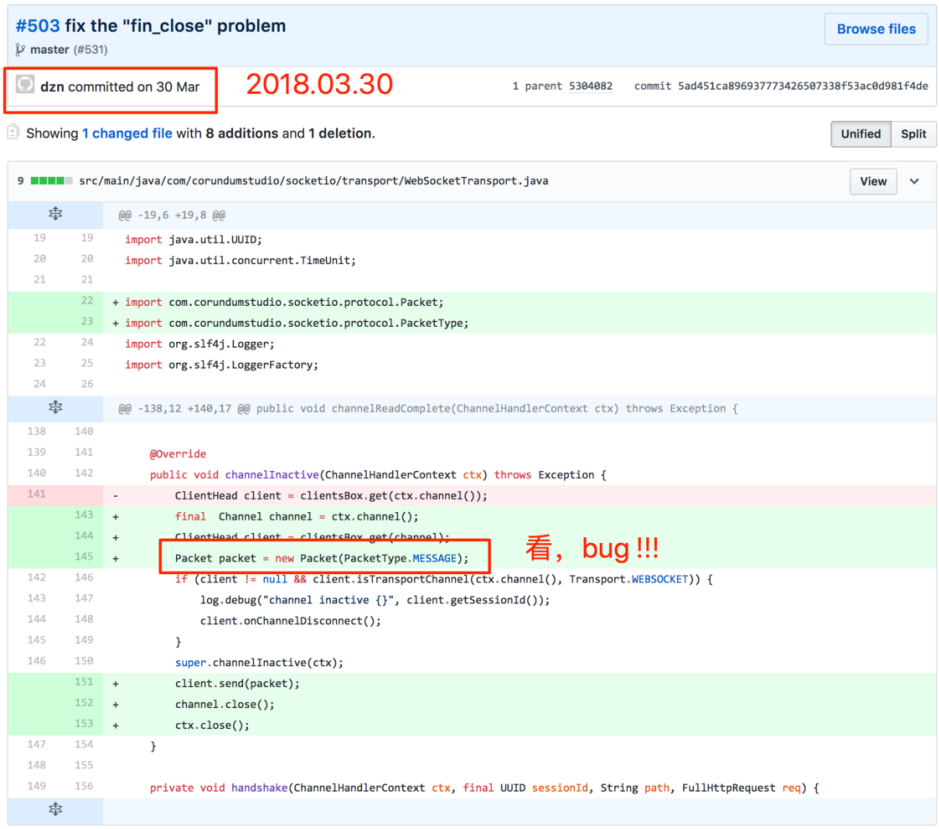
我们找到 idea 的 debugger 面板，眼睛盯着 packet 这个对象不放，然后上线移动光标，便光速定位到。原来，定义 packet 对象这个地方在我们前面的代码其实已经出现过，我们查看了一下 `subType` 这个字段，果然是 null。接下来，解决 Bug 就很容易了。



我们给这个字段赋值即可，由于这里是连接关闭事件，所以我们给他指定了一个名为 DISCONNECT 的字段（可以改天深入去研究 Socket.IO 的协议），反正这个 Bug 是在连接关闭的时候触发的，就粗暴一点了！

解决这个 Bug 的过程是：将这个框架的源码下载到本地，然后加上这一行，最后重新 Build一下，pom 里改了一下名字，推送到我们公司的仓库。这样，项目就可以直接进行使用了。

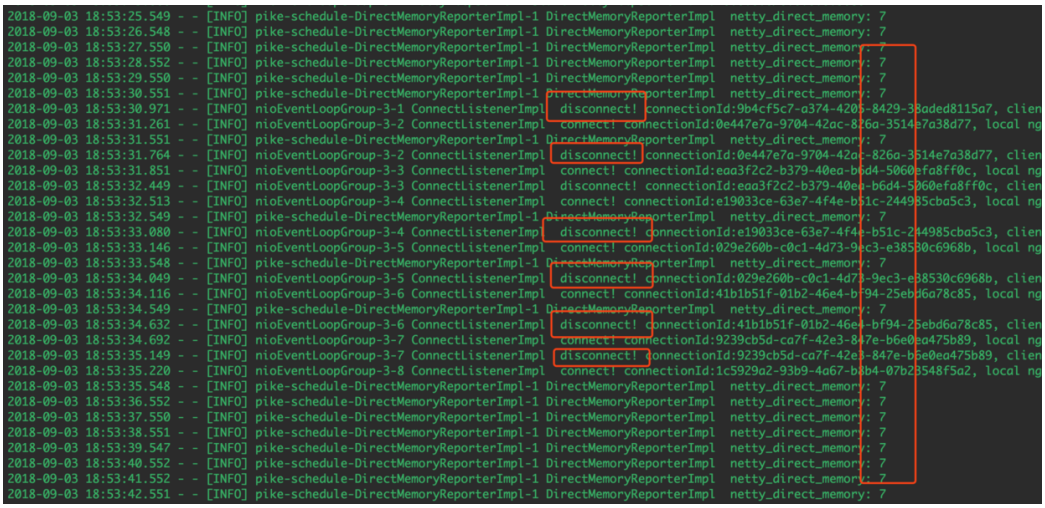
改完 Bug 之后，习惯性地去 GitHub上找到引发这段 Bug 的 Commit：



好奇的是，为啥这位 `dzn` commiter 会写出这么一段如此明显的 Bug，而且时间就在今年3月30号，项目启动的前夕！

阶段9：线下验证

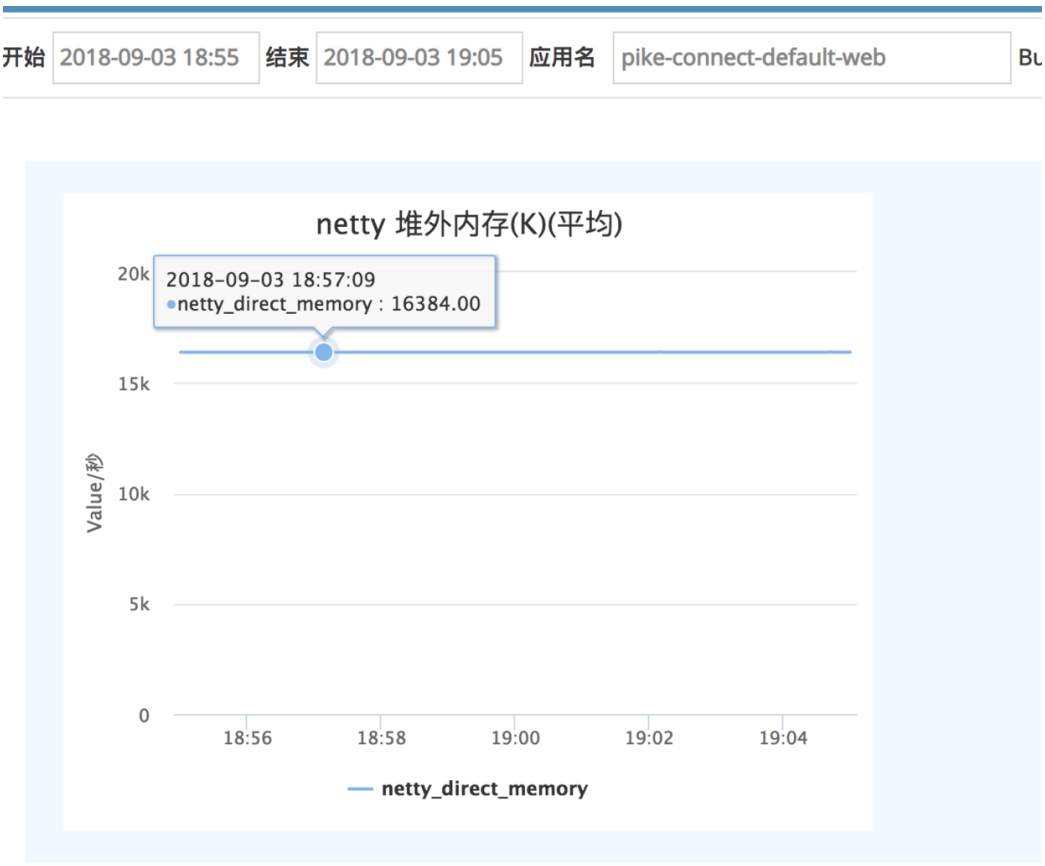
一切准备就绪之后，我们就来进行本地验证，在服务起来之后，我们疯狂地建立连接，疯狂地断开连接，并观察堆外内存的情况：



Bingo！不管我们如何断开连接，堆外内存不涨了。至此，Bug 基本 Fix，当然最后一步，我们把代码推到线上验证。

阶段10：线上验证

这次线上验证，我们避免了比较土的打日志方法，我们把堆外内存的这个指标“喷射”到 CAT 上，然后再来观察一段时间的堆外内存的情况：



过完一段时间，堆外内存已经稳定不涨了。此刻，我们的“捉虫之旅”到此结束。最后，我们还为大家做一个小小的总结，希望对您有所帮助。

总结

1. 遇到堆外内存泄露不要怕，仔细耐心分析，总能找到思路，要多看日志，多分析。
2. 如果使用了 Netty 堆外内存，那么可以自行监控堆外内存的使用情况，不需要借助第三方工具，我们是使用的“反射”拿到的堆外内存的情况。
3. 逐渐缩小范围，直到 Bug 被找到。当我们确认某个线程的执行带来 Bug 时，可单步执行，可二分执行，定位到某行代码之后，跟到这段代码，然后继续单步执行或者二分的方式来定位最终出 Bug 的代码。这个方法屡试不爽，最后总能找到想要的 Bug。
4. 熟练掌握 idea 的调试，让我们的“捉虫”速度快如闪电（“闪电侠”就是这么来的）。这里，最常见的调试方式是预执行表达式，以及通过线程调用栈，死盯某个对象，就能够掌握这个对象的定义、赋值之类。

最后，祝愿大家都能找到自己的“心仪已久”Bug！

作者简介

- 闪电侠，2014年加入美团点评，主要负责美团点评移动端统一长连工作，欢迎同行进行技术交流。

招聘

目前我们团队负责美团点评长连基础设施的建设，支持美团酒旅、外卖、到店、打车、金融等几乎公司所有业务的快速发展。加入我们，你可以亲身体验到千万级在线连接、日吞吐百亿请求的场景，你会直面互联网高并发、高可用的挑战，有机会接触到 Netty 在长连领域的各个场景。我们诚邀有激情、有想法、

有经验、有能力的同学，和我们一起并肩奋斗！欢迎感兴趣的同学投递简历至 chao.yu#dianping.com 咨询。

参考文献

1. Netty 是什么🔗
2. Netty 源码分析之服务端启动全解析🔗

Oceanus：美团HTTP流量定制化路由的实践

作者: 周峰

背景

Oceanus是美团基础架构部研发的统一HTTP服务治理框架，基于Nginx和ngx_lua扩展，主要提供服务注册与发现、动态负载均衡、可视化管理、定制化路由、安全反扒、session ID复用、熔断降级、一键截流和性能统计等功能。本文主要讲述Oceanus如何通过策略抽象、查询、渲染和分组动态更新，实现HTTP请求的定制化路由。

随着公司业务的高速发展，路由场景也越来越复杂。比如：

- 团购秒杀要灵活控制压测流量，实现线上服务单节点、各机房、各地域等多维度的压测。
- 外卖业务要做流量隔离，把北方地域的流量转发到分组a，南方地域的流量转发到分组b。
- 酒旅业务要对App新版本进行灰度，让千分之一的用户试用新版本，其他用户访问老版本。
- QA部门要通过请求的自定义参数指定转发分组，构建稳定且高可用的测试环境。

由于公司早期的业务场景相对比较简单，所以均通过Nginx if指令支持。比如某业务要把来源IP为10.4.242.16的请求转发到后端节点10.4.232.110，其它请求转发到后端节点10.4.232.111和10.4.232.112，就可以进行如下配置：

```
upstream backend_aaa {
    server 10.4.232.110:8080 weight=10;
}
upstream backend_bbb {
    server 10.4.232.111:8080 weight=10;
    server 10.4.232.112:8080 weight=10;
}
location /abc {
    if($remote_ip = "10.4.242.16") {
        proxy_pass http://backend_aaa; #路由到backend_aaa集群
    }
    proxy_pass http://backend_bbb; #路由到backend_bbb集群
}
```

上述方式虽然不需要额外开发，性能方面也接近原生的Nginx框架，但是使用场景比较受限，因为if指令仅支持比较简单的condition类型，官方描述如下：

> A condition may be any of the following:
>
> - a variable name; false if the value of a variable is an empty string or “0”;
>
> > Before version 1.0.1, any string starting with “0” was considered a false value.
>
> - comparison of a variable with a string using the “=” and “!=” operators;
> - matching of a variable against a regular expression using the “~” (for case-sensitive matching) and “~*” (for case-insensitive matching) operators. Regular expressions can contain captures that are made available for later reuse in the $1..$9 variables. Negative operators “!~” and “!~*” are also available. If a regular expression includes the “}” or “;” characters, the whole expressions should be enclosed in single or double quotes.
> - checking of a file existence with the “-f” and “!-f” operators;
> - checking of a directory existence with the “-d” and “!-d” operators;
> - checking of a file, directory, or symbolic link existence with the “-e” and “!-e” operators;
> - checking for an executable file with the “-x” and “!-x” operators.

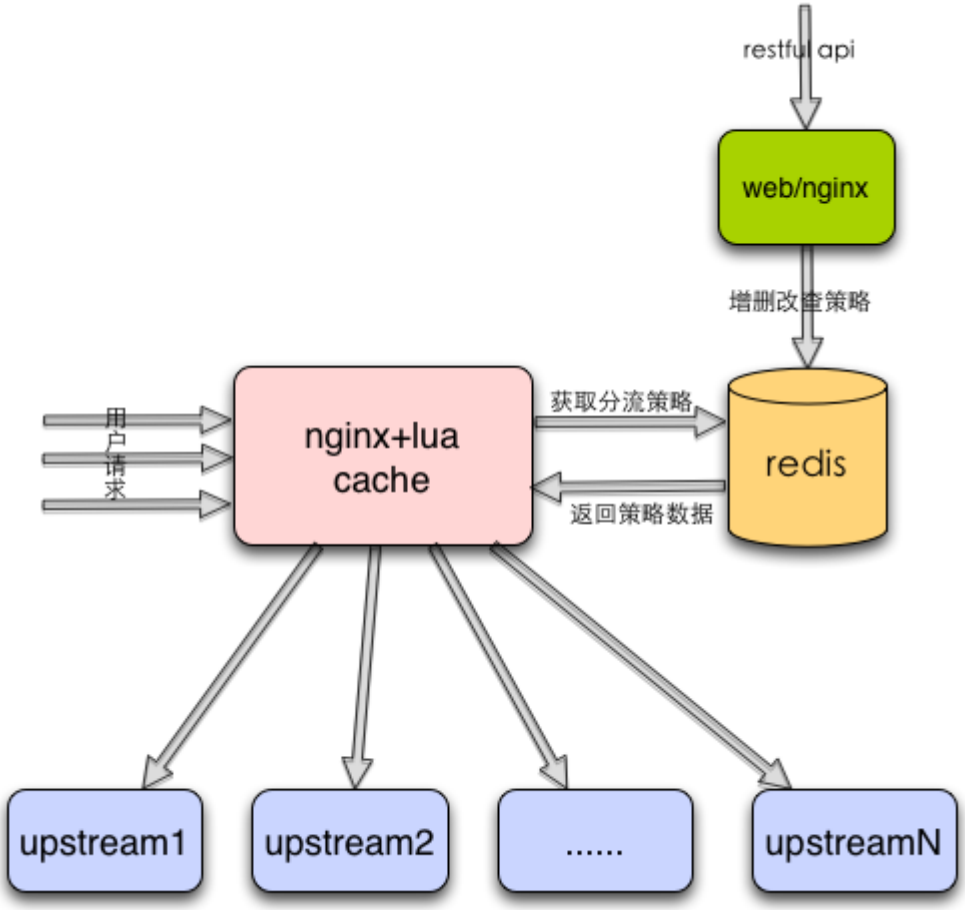
如果该业务要把IP段10.4.242.16/34的请求转发到10.4.232.110时，if指令勉强还可以支持。但对于上述的复杂业务场景，if指令均无法支持。除此之外，这种方式还存在以下两点不足：

- 规则调整不支持动态化：如果要把客户端10.4.242.16调整为10.4.242.17，需要对Nginx进行reload，而reload操作会使Nginx的并发能力下降，业务高峰时甚至会导致请求504或502。
- 指令坑太多：if指令和set、rewrite指令等一起使用时，很多时候会出现不符合预期的行为，严重时甚至会导致段错误，最好的方法就是避免使用。

为了解决上述问题，Oceanus开始探索如何实现HTTP流量的定制化路由。

业界调研

通过初步调研，发现业界有一套开源的ABTestingGateway（以下简称AB）框架：



由上图所示，AB框架使用Redis存储策略数据，key是Host字段，value是策略对象，包括策略类型、匹配区间和要分发的Upstream。策略的增删改查可以通过基于Nginx搭建的Web服务的API实现，运行时根据请求的Host字段从lua-shared-dict或Redis获取关联的策略，根据策略类型（iprange/uidrange/uidsuffix/uidappoint）选择对应的Lua脚本从请求中获取相关参数（IP、UID）查询是否匹配策略，若匹配，就修改请求的Upstream上下文完成分流的目的。

相比if指令的方式，AB框架有下面两个优点：

- 策略调整动态生效：已有策略类型中的策略变更均可以通过HTTP API进行动态管理。
- 分流策略丰富：支持IP段、UID段等策略，也可以通过新增策略类型对策略库进行扩展。

由于AB框架只支持4种策略类型，对于业务要根据请求Cookie、自定义header控制转发的情况，均需要开发新的策略类型和发布上线。另外，策略类型和业务场景紧密相关，导致AB系统的扩展性极差，很难快速支持新业务的路由需求。

无论是Nginx if指令，还是AB框架，要么需要reload重新加载才能生效，要么无法支持某些业务场景下的分流需求，所以都很难作为解决公司级分流框架的有效手段。针对它们所存在的不足，Oceanus开发了一套应用级、高可扩展的动态分流框架，不仅动态支持各种业务场景的分流需求，而且保证了请求转发的性能，下文将阐述我们如何解决分流机制的几个核心问题。

Oceanus定制化路由的核心设计&实现

关于分流机制，我们主要从以下四个方面来讲述：

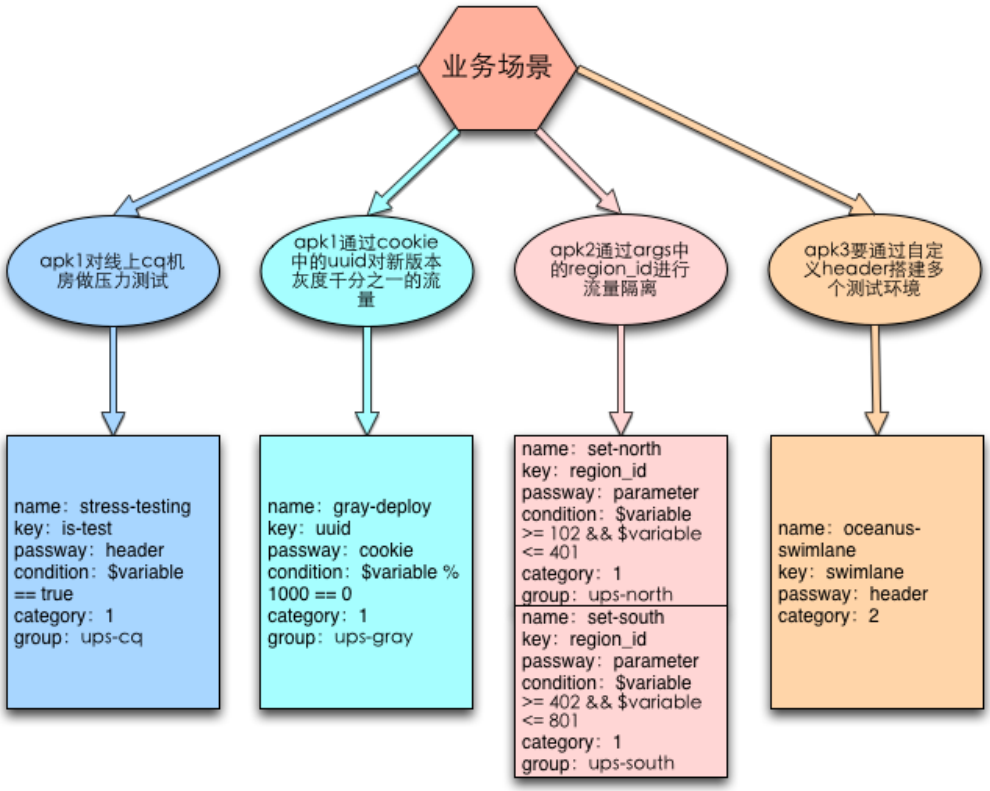
- 策略抽象：合理定义策略结构，适用尽可能多的业务场景。
- 策略的高效查询：接口粒度关联，应用维度管理。
- 运行时策略渲染：渲染策略模板，判断是否匹配策略，实现动态路由。
- 分组动态更新：分组数据增删改，均不需要reload。

策略的结构定义

以AB框架为例，只支持iprange、uidrange、uidsuffix、uidappoint四种场景，对策略类型和匹配方式太具体化，导致无法支持更多普适性的业务场景。从分流的本质出发，即根据请求特征完成流量的定制化路由。结合Nginx if指令的几个组成部分：条件判断依赖的变量、条件判断要匹配的value、条件表达式、匹配后要执行的proxy_pass，一个策略必须要包含请求特征描述、定制化路由描述以及两者的关系描述。其中请求特征描述包含特征关键字、关键字的上下文传输方式，定制化路由描述通过Upstream表示，Upstream可以预先设置，也可以动态指定，两者的关系通过泛型表达式表示。那么一个策略就需要包含下面几个属性：

- name：策略名，没有实际意义，可以根据业务场景进行定义。
- key：分流时依赖的关键字，比如要根据城市地域进行分发路由时，key就是regionid。
- passway：关键字在HTTP协议中的传输方式，可以是Parameter、Cookie、header、body中的一种。
- condition：表达式模板，支持四则运算/取模、关系运算符、逻辑运算符等。
- group：后端服务集群，即匹配策略后，转发请求的目标节点，一般是策略所属应用集群中的部分节点。
- category：策略类型，如果为1，表示某个服务的私有策略；如果为2，表示公共策略，主要用于策略数据管理。
- switch：策略开关，用于控制当前策略是在线还是离线。
- graylist：灰度列表，用于策略变更的线上灰度校验。

其中switch、graylist字段主要用于策略的上下线操作，这里不做过多讨论。下面重点介绍上面的策略定义是如何表述业务场景的：



备注：应用apk1和apk2分别配置2个私有策略，apk3使用公共策略。

如上图所示，无论业务根据请求的哪些特征进行分流，策略结构均可以支持。

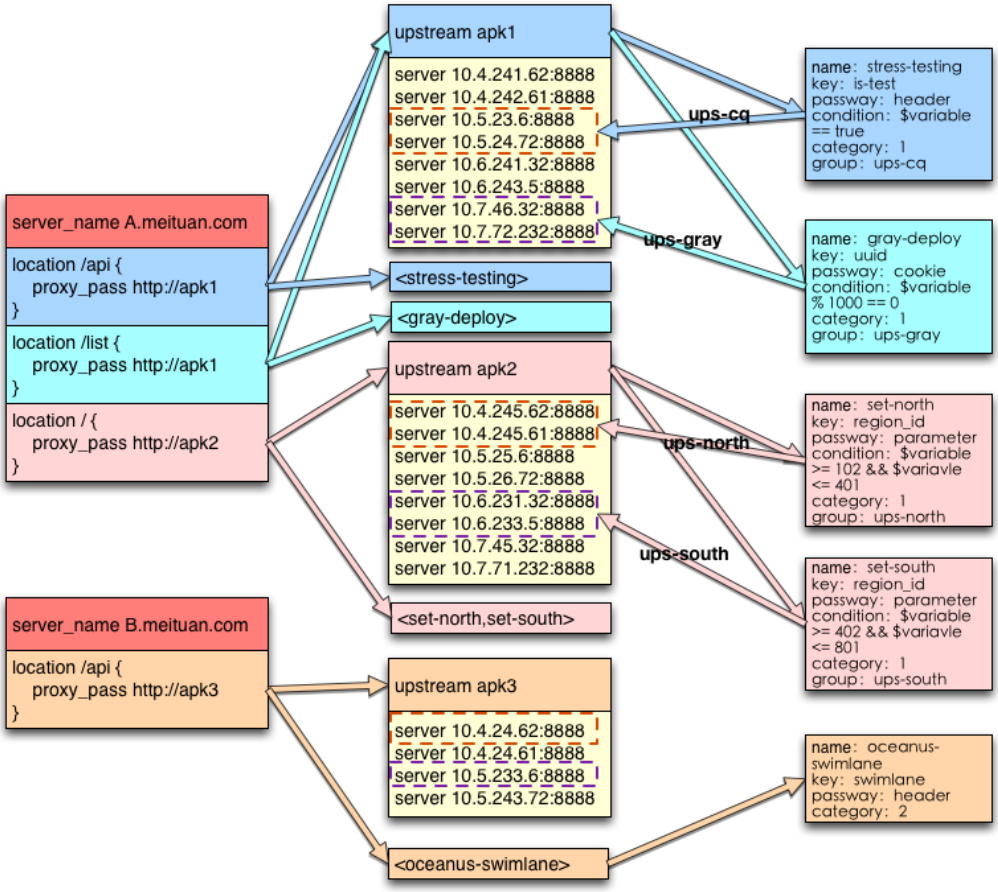
以私有策略gray-deploy为例，在Oceanus管理平台进行添加，如下图所示：

备注：这里省略了策略的非核心字段比如switch、graylist等。

如何实现策略的高效查询?

策略拓扑关系

分流策略分为私有策略和公共策略。私有策略是面向服务的，而且和该服务创建的分组紧密相关。不同服务的私有策略完全独立，可以相同，也可以不同。一个服务可以配置多个私有策略，也可以关联多个Host的Location，Location之间的策略使用完全独立，一个Location可以启用该服务的一个或者多个私有策略。如果通过Host+location_path直接关联策略数据，不同Location关联同一个私有策略时，会存在大量的数据冗余。所以我们通过服务标识（appkey，唯一标识一个应用服务）关联具体的策略数据，Host+location_path只关联当前Location使用的策略名列表，策略之间支持指定顺序。 公共策略与具体服务无关，策略名全局唯一，可以使用策略名关联策略数据即可。综上，策略的拓扑关系描述如下：



StrategyTopology

如上图所示，以应用apk1为例，关联了两个Location接口，分别为/api和/list，总共部署了8个节点，创建了2个分组ups-cq和ups-gray，其中节点10.5.23.6和10.5.24.72属于分组ups-cq，节点10.7.46.32和10.7.72.232属于分组ups-gray。应用配置了两个私有策略stress-testing和gray-deploy，其中策略stress-testing被接口/api启用，匹配策略的流量路由到分组ups-cq，策略gray-deploy被接口/list启用，匹配策略的流量路由到ups-gray。

运行时获取Location path

Nginx在解析Location配置时，通过不同的字段区分不同类型的Location，没有记录配置中的Location path。如果要运行时获取，一般有两种方式：一种是根据相关字段逆向还原path，另一种是为框架新增变量。由于Nginx在处理正则Location时，对于是否忽略大小写的情况，并没有做标记，即解析的过程是不可逆的，所以我们选择了第二种方式。在核心模块的变量数组ngx_http_core_variables中新增了内置变量，记录下原始的Location path，变量属性定义如下：

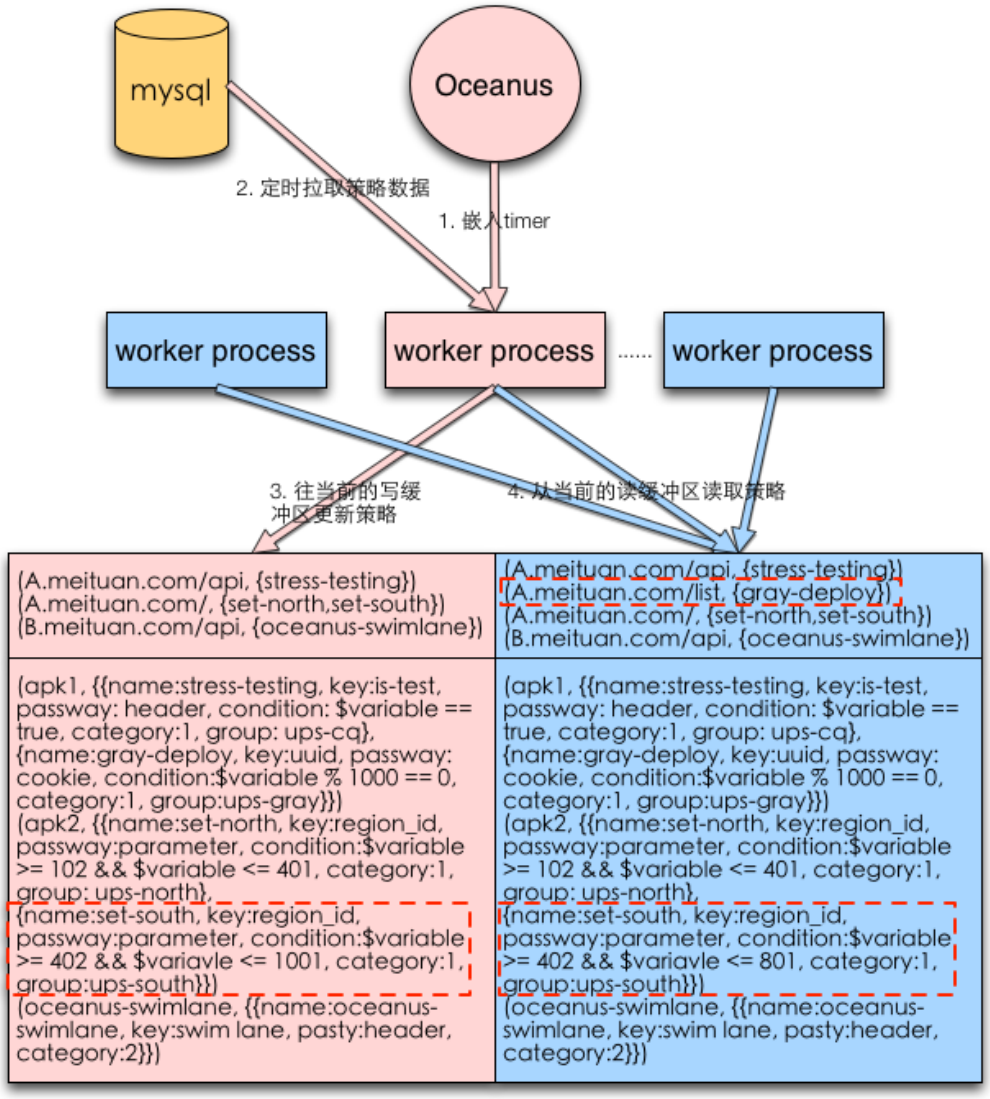
```
{ngx_string("loc_mod"), NULL, ngx_http_variable_loc_mod,
  0, NGX_HTTP_VAR_NOCACHEABLE, 0},
{ngx_string("loc_name"), NULL, ngx_http_variable_loc_name,
  0, NGX_HTTP_VAR_NOCACHEABLE, 0}
```

loc_mod和loc_name之间用一个空格符连接，格式和Oceanus管理平台保持一致。

异步更新机制

为了保证运行时获取策略数据的高效性，我们通过异步定时拉取，把策略数据全量同步到本地的共享内存中。基于稳定性和灵活性的考虑，我们采用了关系型数据库MySQL存储策略。

更新机制如下图所示：

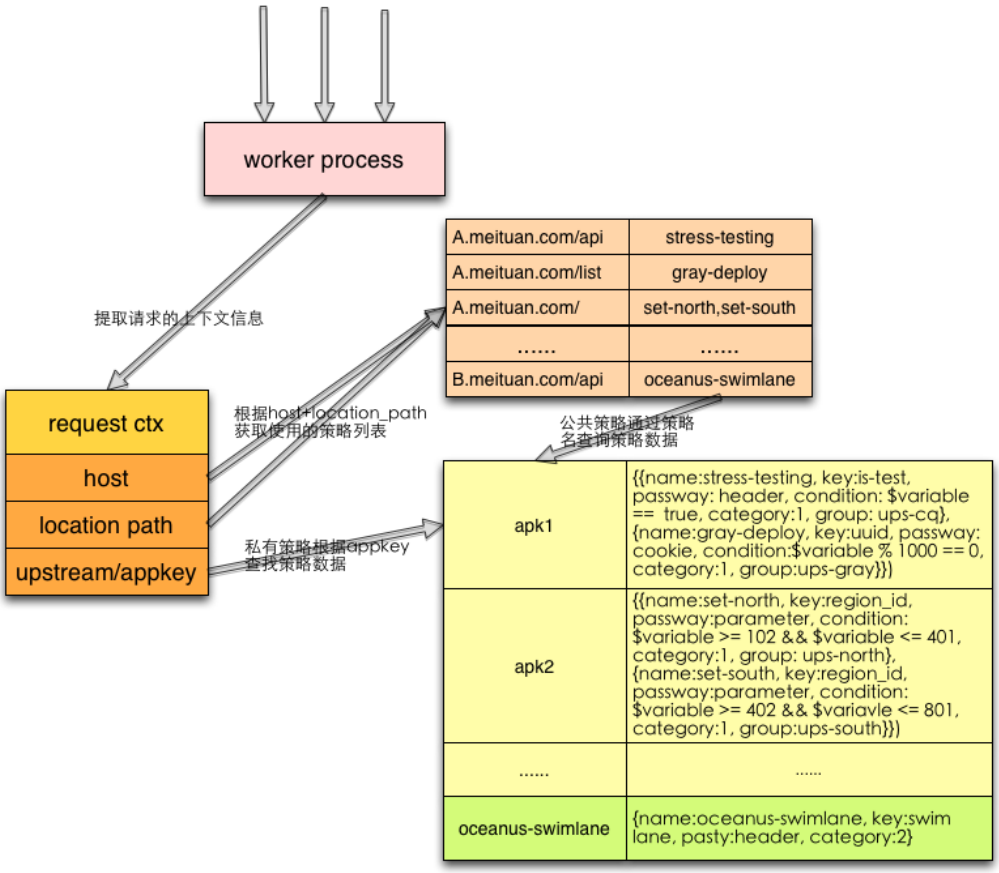


1. Oceanus在init_worker阶段随机选择某个worker进程，嵌入timer。
2. 被选中的worker会异步非阻塞地从MySQL定时拉取策略数据。
3. timer worker把拉取到的策略数据解析，按照策略的拓扑关系，更新到当前共享内存中的写缓存区，完成更新后，切换读写缓存区，保证最新的策略立即生效。
4. worker进程在处理请求时，从当前共享内存中的读缓存区获取策略数据。

为了解决timer worker和其它worker在读写策略数据时的竞态关系，我们采用了双buffer机制，实现了业务层策略数据的无锁读写。另外，通过设置timer的时间为0，保证在所有worker处理请求前，策略数据已经在共享内存中完成初始化。

策略查询机制

查询算法如下图所示：

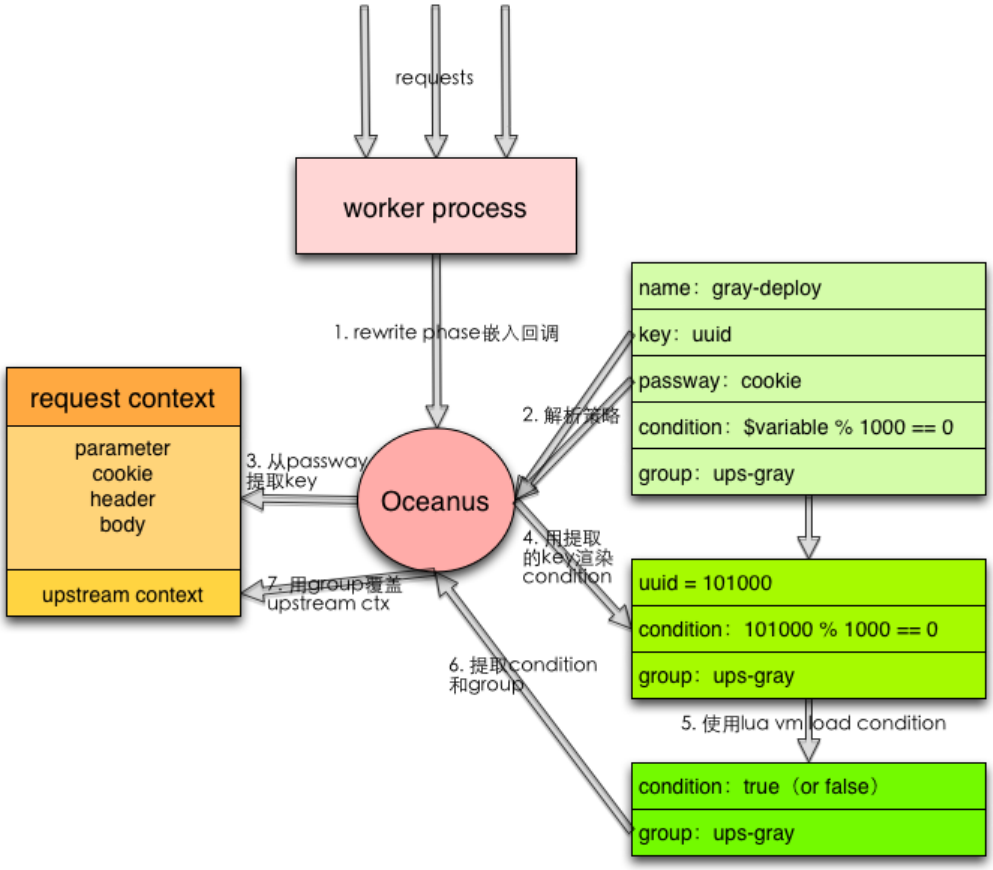


1. worker进程从request上下文中获取请求的Host，以及所匹配Location的location_path。
2. 根据Host+location_path，到共享内存中查询所开启的策略名。
3. 如果是公共策略，直接根据策略名去查询策略数据。
4. 如果是私有策略，从request上下文获取Location关联的Upstream，即应用标识appkey，到共享内存读缓存区获取具体的策略数据。

备注：公共策略以”oceanus”开头，区别于私有策略的命名。

运行时策略渲染

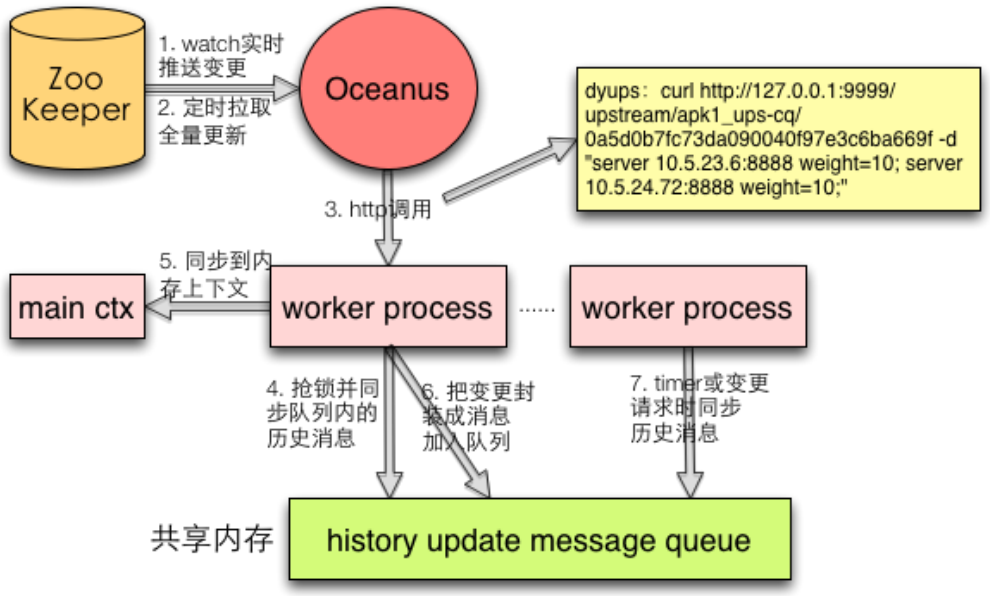
查询到请求开启的策略后，Oceanus需要运行时判断是否匹配，以私有策略为例，执行流如下图所示：



1. 在rewrite phase，Oceanus通过rewrite_by_lua_file嵌入回调，触发请求处理，进入分流框架的主流程。
2. 通过上面的策略查询机制获取请求的策略，进行解析，获取策略的key和passway。
3. 根据passway从请求对应的上下文获取key的value。
4. 用3获取到的value渲染策略的condition，把condition中的占位符替换为value。
5. 基于Lua VM，通过load计算condition的结果，即true或false。
6. 从策略中获取condition的value和group数据。
7. 如果condition为true，就用group覆盖请求的Upstream上下文，否则，不做处理。

分组动态更新

分组列表的动态化是分流框架的重要一环。更新机制如下图所示：



1. 分组数据使用ZooKeeper存储，变更通过watcher机制实现增量同步。
2. Oceanus也会定时拉取，进行全量同步。
3. Oceanus把所有变更都通过本地的HTTP调用同步到Nginx内存。
4. worker处理变更请求前，会先抢锁，读取共享内存中的消息队列，同步其它worker进行的历史变更。
5. 把这次变更同步到当前worker的Upstream main上下文中，完成当前worker的更新。
6. 把变更封装成消息，加入到共享内存中的队列。
7. 其它worker通过timer或者自己处理变更消息前读取消息队列，完成更新。

总结

通过Oceanus分流机制在美团外卖、酒旅、到店餐饮等多个业务线的广泛使用，基础架构部帮助业务同胞解决了多个定制化路由的需求，比如服务set化、链路压测、灰度发布、泳道环境建设等等。目前，Oceanus分流机制只关注了流量转发方向，还不支持更复杂的转发动作，比如根据策略调整请求的Parameter、header、Cookie，也不支持根据请求的URL实现动态路由等，未来我们还将逐一完善这些问题，当然也欢迎大家跟我们一起交流，共同进步。

作者简介

- 周峰，美团高级工程师，2015年7月加入美团基础架构部，先后负责统一密钥管理服务、智能反爬服务和HTTP负载均衡，目前主要负责HTTP服务治理Oceanus的相关工作，致力于探索和研究服务的自动化、智能化、和高性能等方向。

招聘信息

如果你对大规模分布式环境下的HTTP服务治理、分布式会话链路追踪等系统感兴趣，诚挚欢迎投递简历至：zhangzhitong#meituan.com。

参考文献

1. ngx_http_rewrite_module：https://nginx.org/en/docs/http/ngx_http_rewrite_module.html

2. AB框架：https://github.com/CNSRE/ABTestingGateway

UAS–点评侧用户行为检索系统

作者: 朱凯

背景

随着整个中国互联网下半场的到来，用户红利所剩无几，原来粗放式的发展模式已经行不通，企业的发展越来越趋向于精耕细作。美团的价值观提倡以客户为中心，面对海量的用户行为数据，如何利用好这些数据，并通过技术手段发挥出数据的价值，提高用户的使用体验，是我们技术团队未来工作的重点。

大众点评在精细化运营层面进行了很多深度的思考，我们根据用户在App内的操作行为的频次和周期等数据，给用户划分了不同的生命周期，并且针对用户所处生命周期，制定了不同的运营策略，比如针对成长期的用户，主要运营方向是让其了解平台的核心功能，提高认知，比如写点评、分享、收藏等。同时，我们还需要为新激活用户提供即时激励，这对时效性的要求很高，从用户的行为发生到激励的下发，需要在毫秒级别完成，才能有效提升新用户的留存率。

所以，针对这些精细化的运营场景，我们需要能够实时感知用户的行为，构建用户的实时画像。此外，面对大众点评超大数据流量的冲击，我们还要保证时效性和稳定性，这对系统也提出了非常高的要求。在这样的背景下，我们搭建了一套用户行为系统（User Action System，以下简称UAS）。

面临的问题

如何实时加工处理海量的用户行为数据，我们面临以下几个问题：

1. **上报不规范**：点评平台业务繁多，用户在业务上产生的行为分散在四处，格式不统一，有些行为消息是基于自研消息中间件Mafka/Swallow，有些行为消息是基于流量打点的Kafka消息，还有一些行为没有对应的业务消息，收集处理工作是一个难点。
2. **上报时效性差**：目前大部分行为，我们通过后台业务消息方式进行收集，但是部分行为我们通过公司统一的流量打点体系进行收集，但是流量打点收集在一些场景下，无法满足我们的时效性要求，如何保证收集处理的时效性，我们需要格外关注。
3. **查询多样化**：收集好行为数据之后，各个业务对用户行为的查询存在差异化，比如对行为次数的统计，不同业务有自己的统计逻辑。无法满足现有业务系统的查询需求，如何让系统既统一又灵活？这对我们的业务架构能力提出了新要求。

针对问题模型，方案思考

格式统一

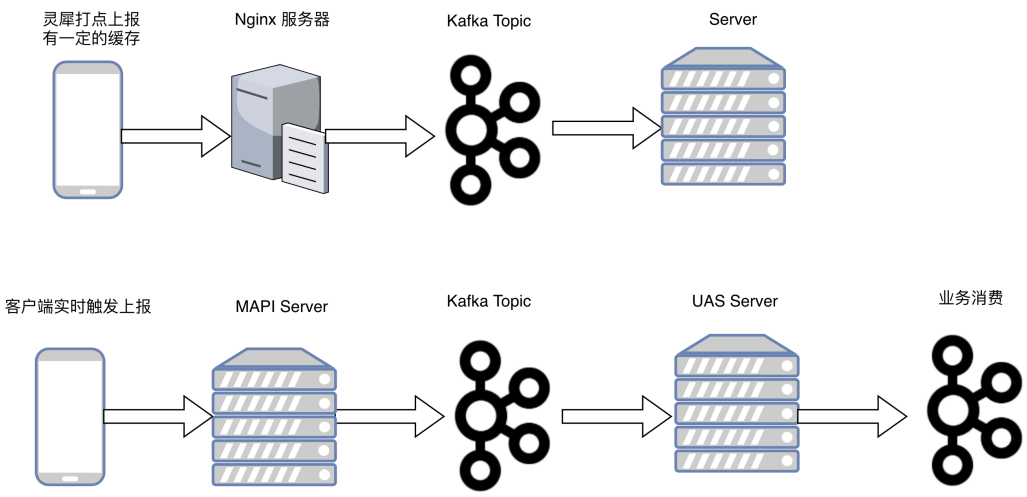
面对繁杂的格式，我们如何进行统一？在这里我们参考了5W1H模型，将用户的行为抽象为以下几大要素：

	who	when	what	where	why	how
5W1H模型	人员	时间	对象	地点	原因	方法
用户行为	行为的触发者	行为发送的时间	行为的类型	行为作用的地方	无	行为的属性

其中行为作用的地方，这里一般都是作用对象的ID，比如商户ID，评论ID等等。 行为的属性，代表的是行为发生的一些额外属性，比如浏览商户的商户品类、签到商家的城市等。

上报统一

对于用户行为的上报，之前的状态基本只有基于流量打点的上报，虽然上报的格式较为标准化，但是存在上报延时，数据丢失的情况，不能作为主要的上报渠道，因此我们自建了其他的上报渠道，通过维护一个通用的MAPI上报通道，直接从客户端通过专有的长连接通道进行上报，保证数据的时效性，上报后的数据处理之后，进行了标准化，再以消息的形式传播出去，并且按照一定的维度，进行了TOPIC的拆分。目前我们是两个上报通道在不同场景使用，对外是无感知的。



服务统一

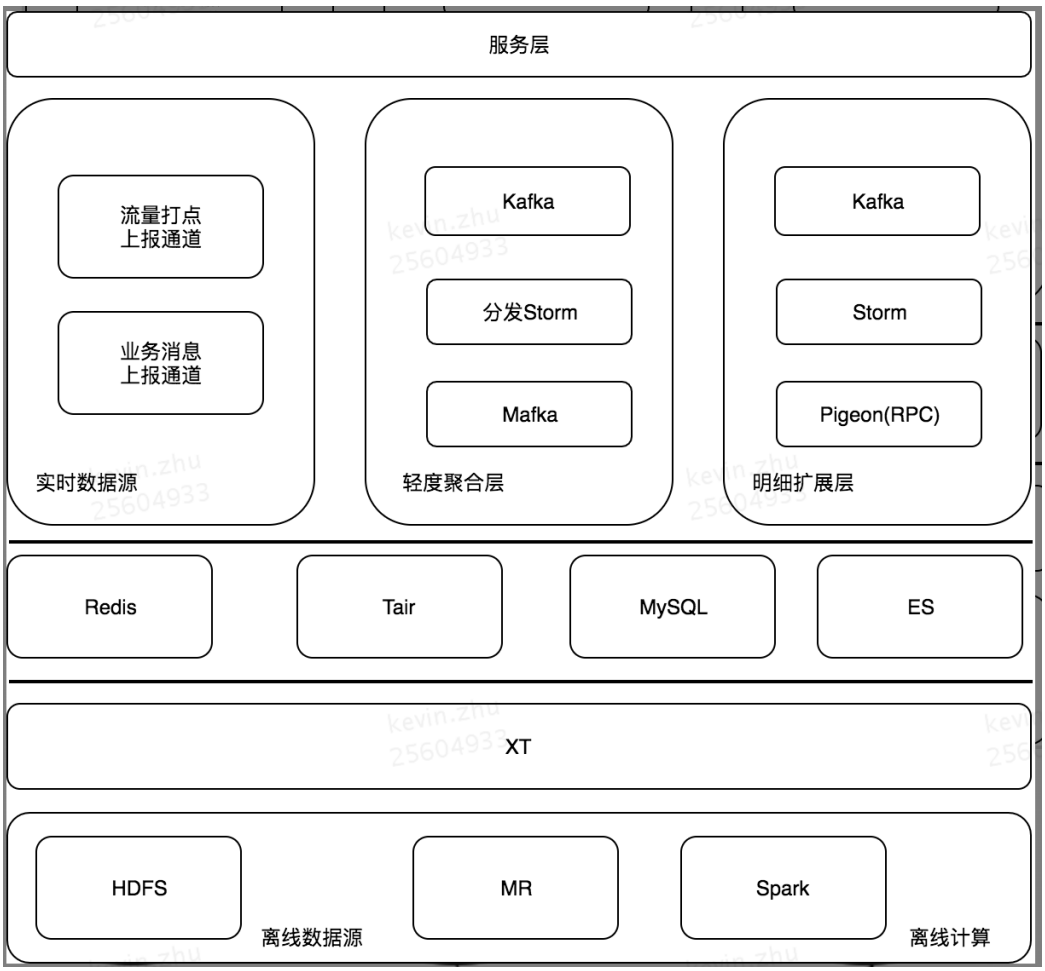
不同场景下，对用户行为处理的数据规模要求，时效性要求也是不一样的，比如有些场景需要在用户行为上报之后，立刻做相关的查询，因此写入和查询的性能要求很高，有些场景下，只需要进行行为的写入，就可以采取异步的方式写入，针对这样不同的场景，我们有不同的解决方案，但是我们统一对外提供的还是UAS服务。

架构统一

从数据的收集上报，到处理分发，到业务加工，到持久化，UAS系统架构需要做到有机的统一，既要能满足日益增长的数据需求，同时也要能够给业务充分的灵活性，起到数据中台的作用，方便各个业务基于现有的架构上，进行快速灵活的开发，满足高速发展的业务。

系统整体架构

针对这样一些想法，开始搭建我们的UAS系统，下图是UAS系统目前的整体架构：



数据源简介

我们处理的数据源分为实时数据源和离线数据源：

1. 实时数据源主要分两块，一块是基于客户端打点上报，另外一块是我们的后台消息，这两部分是基于公司的消息中间件Mafka和开源消息中间件Kafka，以消息的形式上报上来，方便我们后续的处理，MQ的方式能够让系统更好的解耦，并且具备更高的吞吐量，还可以指定消费的起始时间点，做到消息的回溯。
2. 历史数据的来源主要是我们的Hive和HDFS，可以方便的做到大数据量的存储和并行计算。

离线计算简介

在离线处理这块，主要包含了MR模块和Spark模块，我们的一些ETL操作，就是基于MR模块的，一些用户行为数据的深度分析，会基于Spark去做，其中我们还有一个XT平台，是美团点评内部基于Hive搭建的ETL平台，它主要用来开发数据处理任务和数据传输任务，并且可以配置相关的任务调度信息。

实时计算简介

对于用户行为的实时数据处理，我们使用的是Storm实时大数据处理框架，Storm中的Spout可以方便的对接我们的实时消息队列，在Bolt中处理我们的业务逻辑，通过流的形式，可以方便的做到业务数据的分流、处理、汇聚，并且保持它的时效性。而且Storm也有比较好的心跳检测机制，在Worker挂了之后，可以做到自动重启，保证任务不挂，同时Storm的Acker机制，可以保持我们实时处理的可靠性。

接下来，我们按照用户行为数据的处理和存储来详细介绍我们的系统。

数据的处理

离线处理

离线数据的处理，主要依赖的是我们的数据开发同学，在构建用户行为的数据仓库时，我们会遵循一套美团点评的数据仓库分层体系。

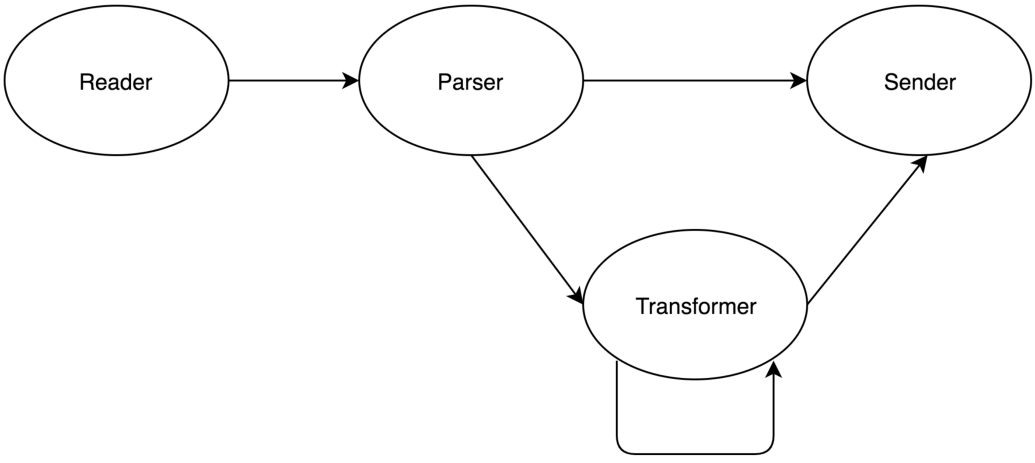
同时我们会出一些比较通用的数据，方便线上用户使用，比如我们会根据用户的行为，发放勋章奖励，其中一个勋章的发放条件是用户过去30天的浏览商户数量，我们不会直接出一个30天的聚合数据，而是以天为周期，做一次聚合，然后再把30天的数据聚合，这样比较通用灵活一些，上层应用可以按照自己的业务需求，进行一些其他时间段的聚合。

在数据的导入中，我们也有不同的策略：

1. 比如用户的行为路径分析中，我们在Hive中计算好的结果，数据量是非常庞大的，但是Hive本身的设计无法满足我们的查询时效性要求，为了后台系统有比较好的体验，我们会把数据导入到ES中，这里我们无需全量导入，只要抽样导入即可，这样在满足我们的查询要求的同时也能提高我们的查询效率。
2. 在导入到一些其他存储介质中，传输的效率有时候会成为我们的瓶颈，比如我们导入到Cellar中，数据量大，写入效率也不高，针对这种情况，我们会采用增量导入的方式，每次导入的数据都是有发生变化的，这样我们的导入数据量会减少，从而减小我们的传输耗时。

实时处理

实时处理这块，我们构建了基于点评全网的流量网关，所有用户产生的行为数据，都会通过实时上报通道进行上报，并且会在我们的网关中流转，我们在这里对行为数据，做一些加工。



实时处理

Reader

我们目前使用的是Storm的Spout组件对接我们的实时消息，基于抽象的接口，未来可以扩展更多的数据来源，比如数据库、文件系统等。

Parser

Parser是我们的解析模块，主要具备以下功能：

1. 我们会对字段做一些兼容，不同版本的打点数据可能会有差异。
2. JSON串的处理，对于多层的JSON串进行处理，使得后续可以正常解析。
3. 时间解析，对于不同格式的的上报时间进行兼容统一。

Transformer

Transformer是我们的转换模块，它是一种更加高级的处理过程，能够提供给业务进行灵活的行为属性扩展： 1. 比如需要根据商户ID转换出商户的星级、品类等其他信息，我们可以在我们的明细扩展层配置一个Transformer。 2. 或者业务有自己的转换规则，比如他需要把一些字段进行合并、拆分、转换，都可以通过一个Transformer模块，解决这个问题。

Sender

Sender是我们的发送模块，将处理好的数据，按照不同的业务数据流，进行转发，一般我们是发送到消息队列中，Sender模块，可以指定发送的格式、字段名称等。

目前我们的实时处理，基本上已经做到可视化的配置，之前需要几人日才能做到的用户行为数据分发和处理，现在从配置到验证上线只需要几分钟左右。

近实时处理

在近线计算中，我们会把经过流量网关的数据，通过Kafka2Hive的流程，写入到我们的Hive中，整个过程的时延不超过15分钟，我们的算法同学，可以利用这样一些近实时的数据，再结合其他的海量数据，进行整体的加工、存储，主要针对的是一些时效性要求不高的场景。

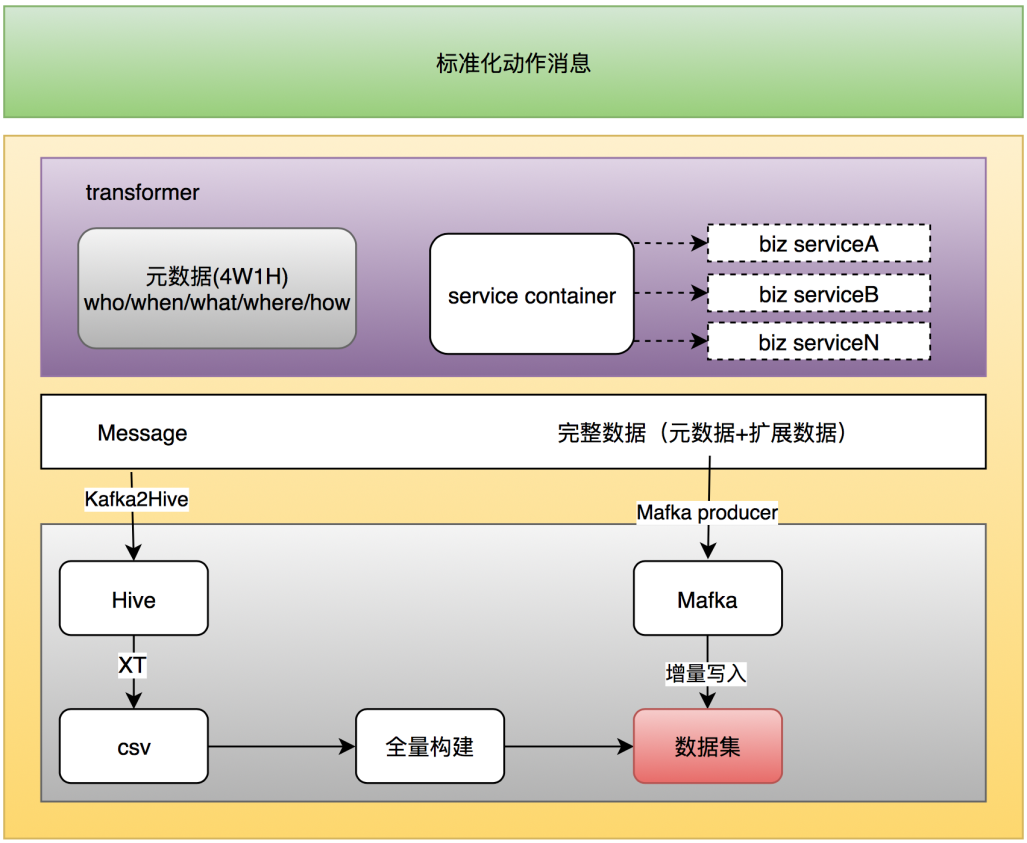
通过上面三套处理方法，离线、实时、近实时，我们可以很好的满足业务不同的时效性需求。

数据的存储

经过实时处理之后，基本上已经是我们认为的标准化数据，我们会对这些数据进行明细存储和聚合存储。

明细存储

明细的存储，是为了保证我们的数据存储，能够满足业务的查询需求，这些明细数据，主要是用户的一些核心操作行为，比如分享、浏览、点击、签到等，这些数据我们会按照一定的粒度拆分，存储在不同的搜索集群中，并且有一定的过期机制。



搜索

上图是我们的处理方式:

1. 通过Transformer，业务方可以通过自己的服务，对数据的维度进行扩展，从而Sender发出的Message就是满足业务需求的数据。
2. 然后在Kafka2Hive这一步，会去更新对应的Hive表结构，支持新的扩展数据字段，同时在XT作业中，可以通过表的关联，把新扩展的字段进行补齐。
3. 重跑我们的历史之后，我们的全量数据就是已经扩展好的字段。同时，我们的实时数据的写入，也是扩展之后的字段，至此完成了字段的扩展。

NoSQL存储

通过明细数据的存储，我们可以解决大部分问题。虽然搜索支持的查询方式比较灵活，但是某些情况下，查询效率会较慢，平均响应时间在20ms左右，对一些高性能的场景，或者一些基础的用户行为画像，这个响应时间显然是偏高的。因此我们引入了NoSQL的存储，使用公司的存储中间件Squirrel和Cellar，其中Cellar是基于淘宝开源的Tair进行开发的，而Squirrel是基于Redis-cluster进行开发的，两者的差异就不在此赘述，简单讲一下我们的使用场景：

1. 对于冷热比较分明，单个数据不是很大（小于20KB，过大会影响查询效率），并且value不是复杂的，我们会使用Cellar，比如一些低频次的用户行为数据。
2. 在大并发下，对于延迟要求极为敏感，我们会使用Redis。
3. 对于一些复杂的数据结构，我们会使用到Redis，比如会用到Redis封装好的HyperLogLog算法，进行数据的统计处理。

系统特性

灵活性

构建系统的灵活性，可以从以下几个方面入手：

1. 对用户的行为数据，可以通过Transformer组件进行数据扩展，从而满足业务的需求，业务只需要开发一个扩展接口即可。
2. 第二个方面就是查询，我们支持业务方以服务注册的方式，去编写自己的查询逻辑，或者以插件的形式，托管在UAS平台，去实现自己负责的业务逻辑，比如同样一个浏览商户行为，有些业务的逻辑是需要看某批用户最近7天看了多少家3星商户，并且按照shopID去重，有些业务逻辑可能是需要看某批用户最近7天浏览了多少个品类的商户。因此这些业务复杂的逻辑可以直接托管在我们这里，对外的接口吐出基本是一致的，做到服务的统一。
3. 我们系统目前从实时分发/计算/统计/存储/服务提供，是一套比较完备的系统，在不同的处理阶段，都可以有不同的组件/技术选型，根据业务的需求，我们可以做到灵活的组合、搭配。

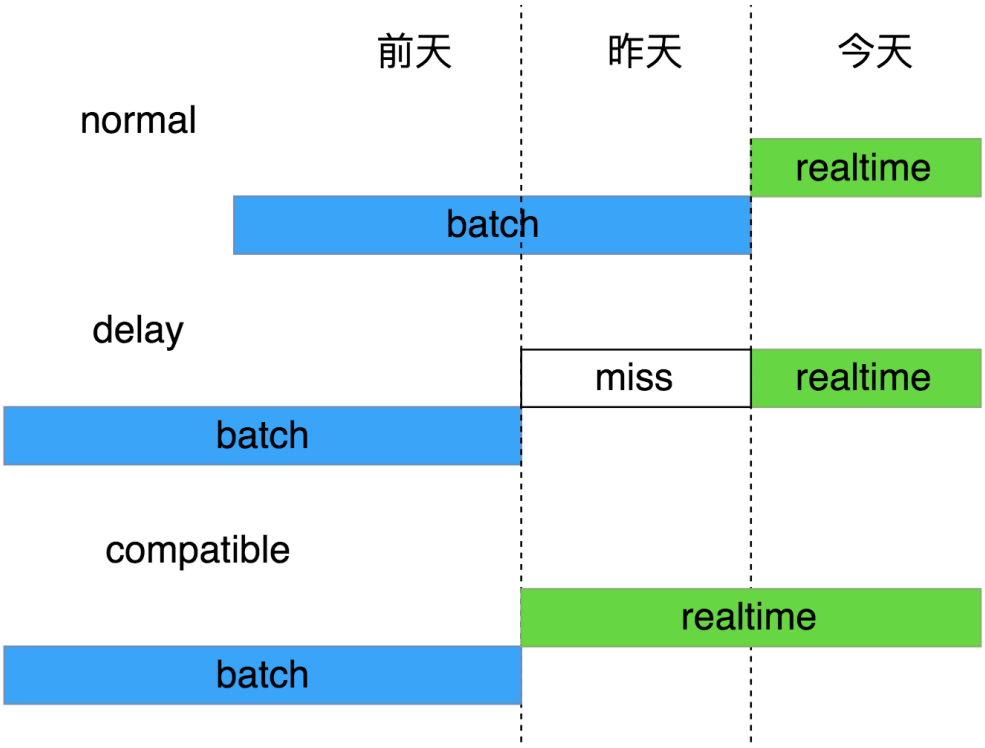
低延时

对于一些跨周期非常长，存储非常大的数据，我们采用了Lambda架构，既保证了数据的完备性又做到了数据的时效性。其中Batch Layer为批处理层，会有固定的计算视图，对历史数据进行预计算，生成离线结果；Speed Layer为实时计算层，对实时数据进行计算，生成增量的结果，最终Server Layer合并两个视图的数据集，从而来提供服务。

可用性

数据可用性

前面提到了我们采用Lambda架构处理一些数据，但是离线数据有时候会因为上游的一些原因，处理不稳定，导致产出延迟，这个时候为了保证数据的准确性，我们在Speed Layer会多保留两天的数据 ，保证覆盖到全量数据。如图所示：

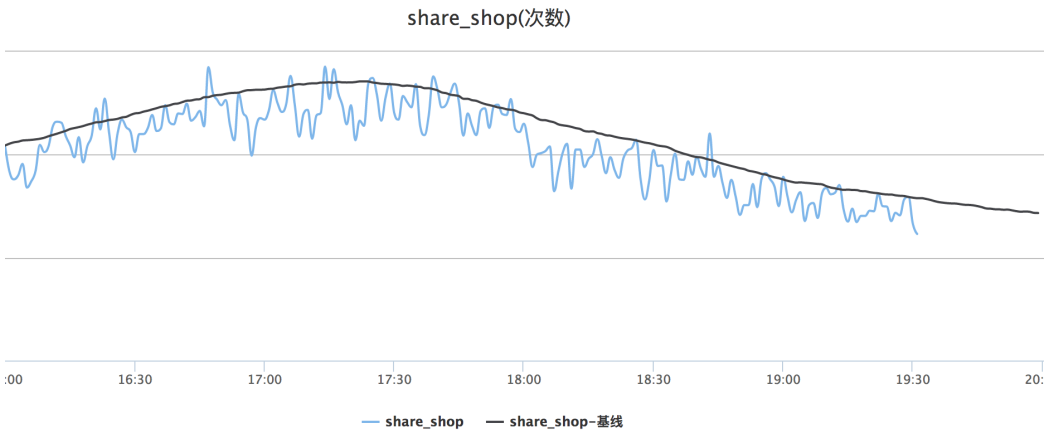


服务的可用性

在服务的可用性方面，我们对接入的服务进行了鉴权，保证服务的安全可靠，部分核心行为，我们做了物理上的隔离，保证行为数据之间不会相互影响，同时接入了公司内部基于Docker的容器管理和可伸缩平台HULK，能做到自动扩容。对于数据使用有严格权限审计，并且做了相关数据脱敏工作。

监控

从用户行为数据的产生，到收集分发，到最后的处理，我们都做到了相关的监控，比如因为我们的代码改动，发生处理时长变长，我们可以立马收到相关的报警，检查是不是代码出问题了。或者监控到的行为产生次数和历史基线比，发生较大变化，我们也会去追踪定位问题，甚至可以早于业务先发现相关问题。下图是分享商户行为的一个监控：



结语

用户行为系统搭建之后，目前：

1. 处理的上报数据量日均在45+亿。
2. 核心行为的上报延迟从秒级降低到毫秒级。
3. 收录用户行为数十项，提供用户行为实时流。
4. 提供多维度下的实时服务，日均调用量在15亿左右，平均响应时间在3ms，99线在10ms。

目前系统承载的业务还在不断增长中，相比以前的T+1服务延时，大大提升了用户体验。我们希望构建用户行为的中台系统，通过我们已经抽象出的基础能力，解决业务80%的问题，业务可以通过插件或者接口的形式，在我们的中台上解决自己个性化的问题。

未来展望

目前我们的实时计算视图，比较简单，做的是相对比较通用的聚合计算，但是业务的聚合规则可能是比较复杂且多变的，不一定是直接累加，未来我们希望在聚合计算这块，也能直接通过配置的方式，得到业务自定义的聚合数据，快速满足线上业务需求。

同时，用户的实时行为会流经我们的网关，我们对用户行为进行一些特征处理之后，结合用户过去的一些画像数据，进行用户意图的猜测，这种猜测是可以更加贴近业务的。

作者简介

- 朱凯，资深工程师，2014年加入大众点评，先后从事过账号端/商家端的开发，有着丰富的后台开发架构经验，同时对实时数据处理领域方法有较深入的理解，目前在点评平台负责运营业务相关的研发工作，构建精细化运营的底层数据驱动能力，着力提升用户运营效率。重点打造点评平台数据中台产品——灯塔。

美团DB数据同步到数据仓库的架构与实践

作者: 萌萌 心序 成聪

背景

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

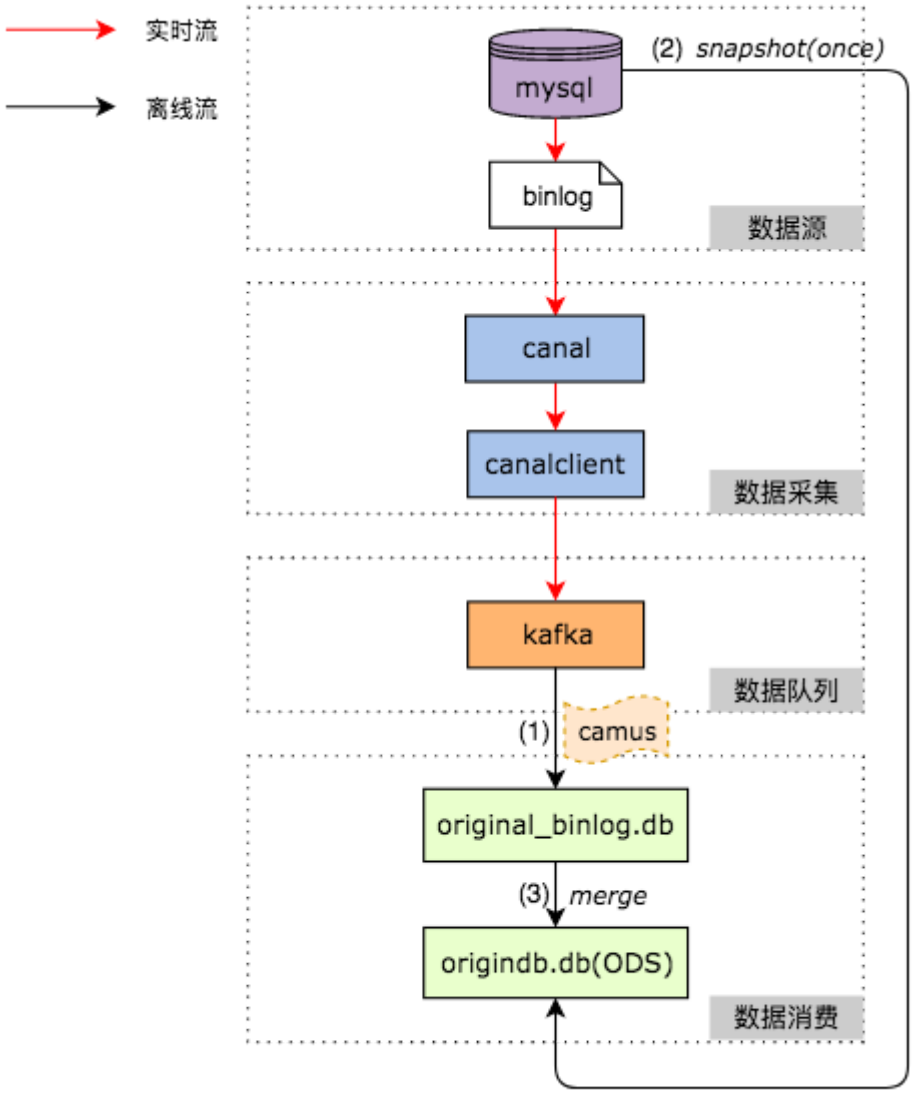
如何准确、高效地把MySQL数据同步到Hive中？一般常用的解决方案是批量取数并Load：直连MySQL去Select表中的数据，然后存到本地文件作为中间存储，最后把文件Load到Hive表中。这种方案的优点是实现简单，但是随着业务的发展，缺点也逐渐暴露出来：

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

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

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

整体架构



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

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

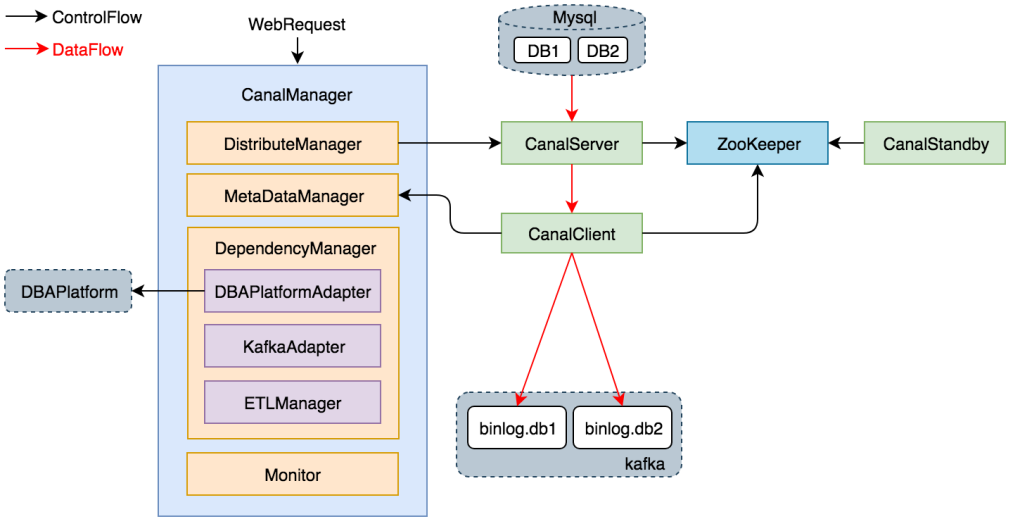
1. 采用Linkedin的开源项目Camus，负责每小时把Kafka上的Binlog数据拉取到Hive上。
2. 对每张ODS表，首先需要一次性制作快照（Snapshot），把MySQL里的存量数据读取到Hive上，这一过程底层采用直连MySQL去Select数据的方式。
3. 对每张ODS表，每天基于存量数据和当天增量产生的Binlog做Merge，从而还原出业务数据。

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

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

Binlog实时采集

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



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

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

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

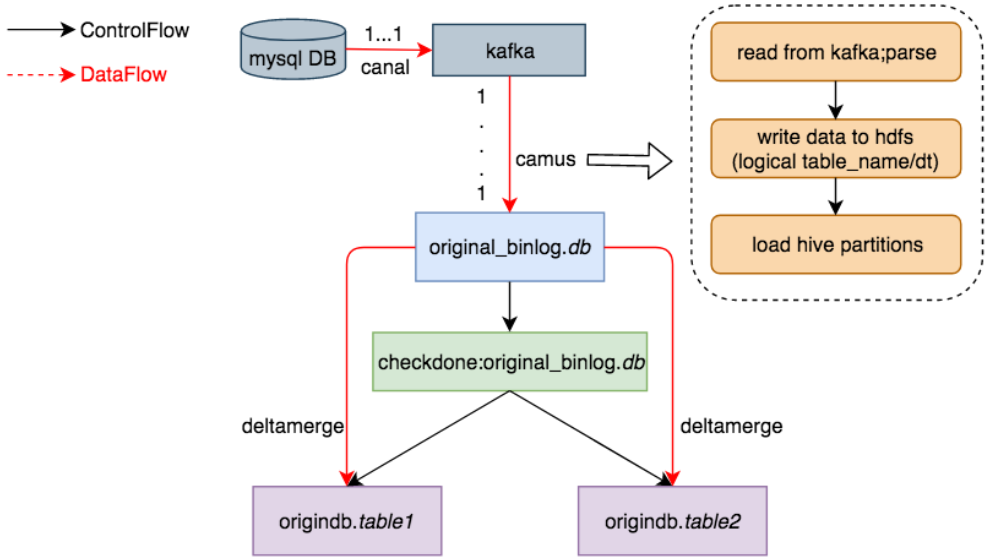
这样做的目的有两个：

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

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

离线还原MySQL数据

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



Kafka2Hive

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

对Camus的二次开发

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

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

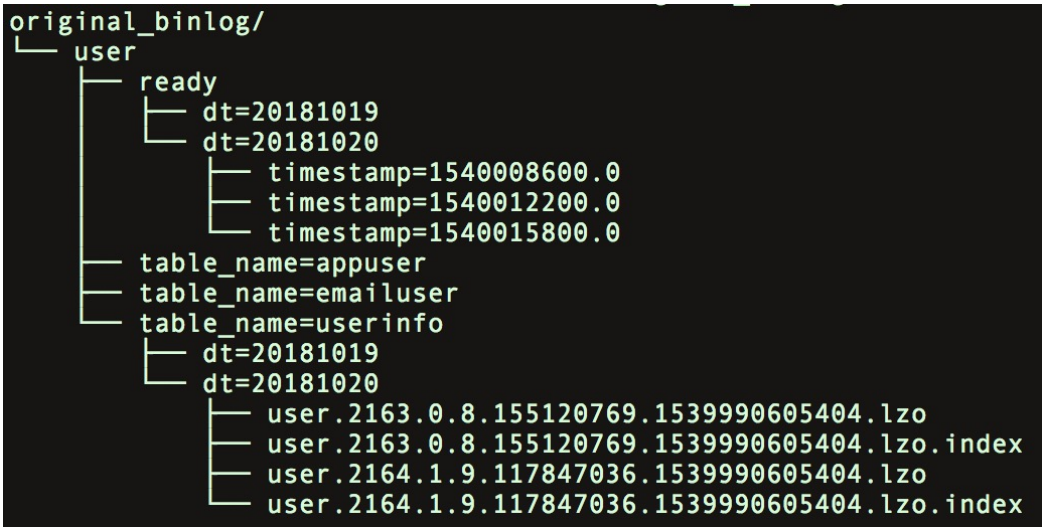
为了解决这一问题，我们引入了Checkdone任务。Checkdone任务是天任务，主要负责检测前一天的Kafka2Hive是否成功完成。如果成功完成了，则Checkdone任务执行成功，这样下游的Merge任务就可以正确启动了。

Checkdone的检测逻辑

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

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

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



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

Merge

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

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

Merge流程举例

下面用一个例子来具体说明Merge的流程。

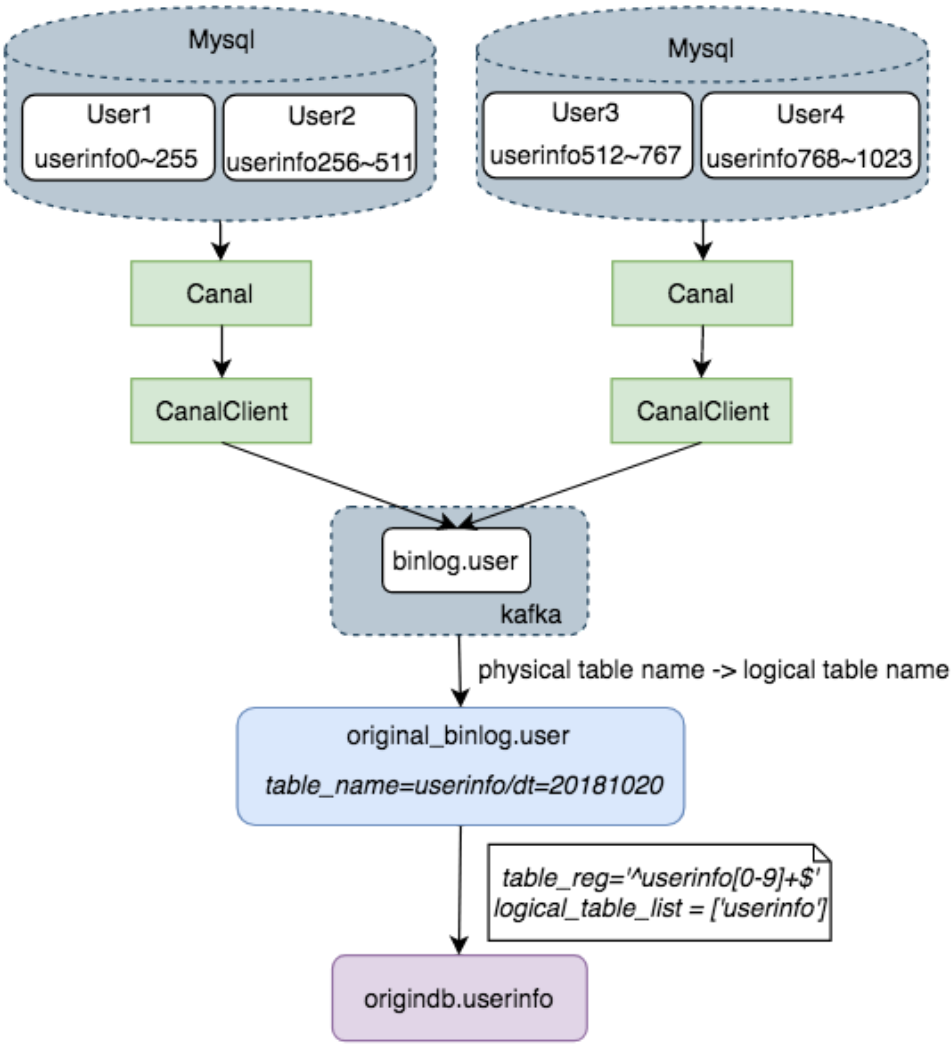
数据表共id、value两列，其中id是主键。在提取Delta数据时，对同一条数据的多次更新，只选择最后更新的一条。所以对id=1的数据，Delta表中记录最后一条更新后的值value=120。Delta数据和存量数据做Merge后，最终结果中，新插入一条数据（id=4），两条数据发生了更新（id=1和id=2），一条数据未变（id=3）。

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

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

实践一：分库分表的支持

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



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

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

这样通过两个层面的工作，就完成了分库分表在ODS层的合并。

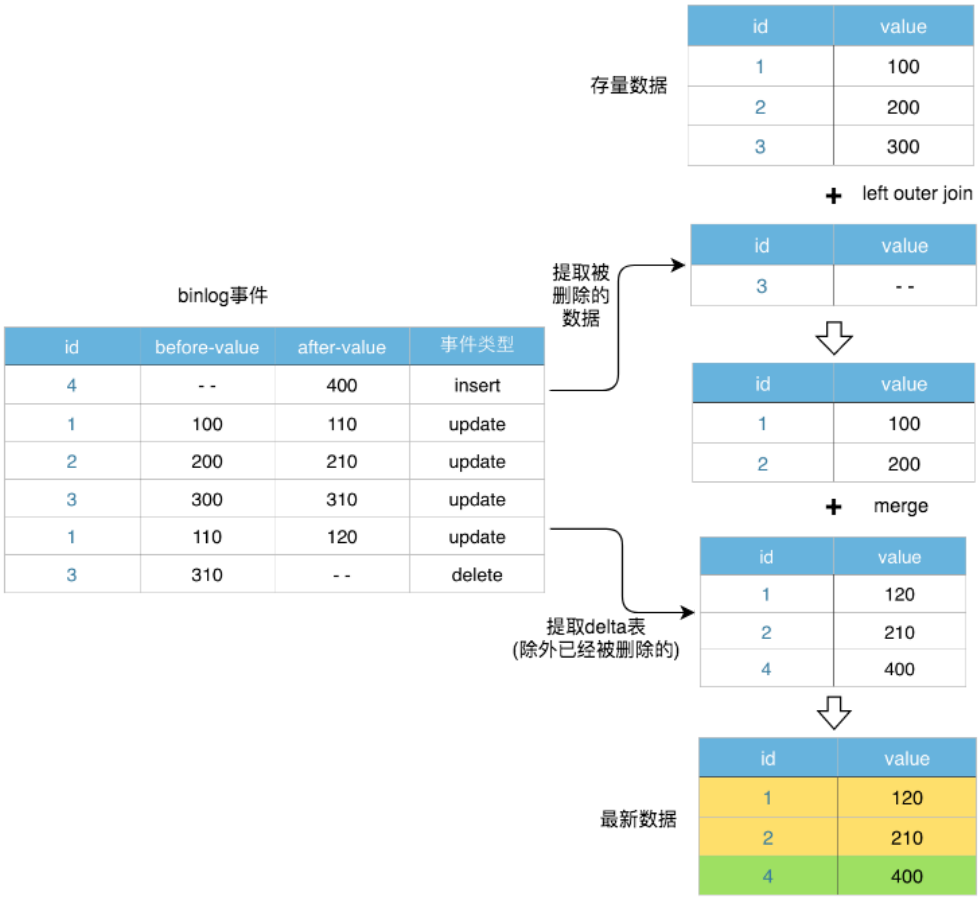
这里面有一个技术上的优化，在进行Kafka2Hive时，我们按业务分表规则对表名进行了处理，把物理表名转换成了逻辑表名。例如userinfo123这张表名会被转换为userinfo，其Binlog数据存储在original_binlog.user表的table_name=userinfo分区中。这样做的目的是防止过多的HDFS小文件和Hive分区造成的底层压力。

实践二：删除事件的支持

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

对需要处理Delete事件的Merge流程，采用如下两个步骤：

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



总结与展望

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

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

招聘

如果你对我们的工作内容比较感兴趣，欢迎发送简历给 wangmengmeng05@meituan.com，和我们一起致力于解决海量数据采集和传输的问题中来吧！

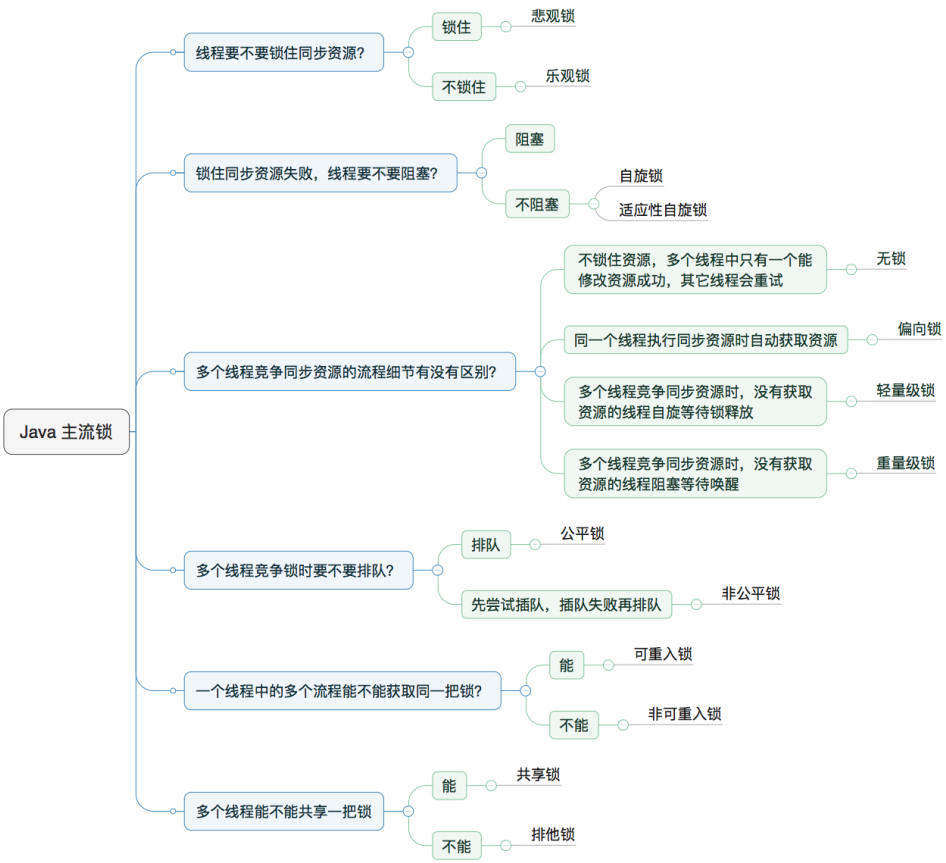
不可不说的Java“锁”事

作者: 家琪

前言

Java提供了种类丰富的锁，每种锁因其特性的不同，在适当的场景下能够展现出非常高的效率。本文旨在对锁相关源码（本文中的源码来自JDK 8和Netty 3.10.6）、使用场景进行举例，为读者介绍主流锁的知识点，以及不同的锁的适用场景。

Java中往往是按照是否含有某一特性来定义锁，我们通过特性将锁进行分组归类，再使用对比的方式进行介绍，帮助大家更快捷的理解相关知识。下面给出本文内容的总体分类目录：



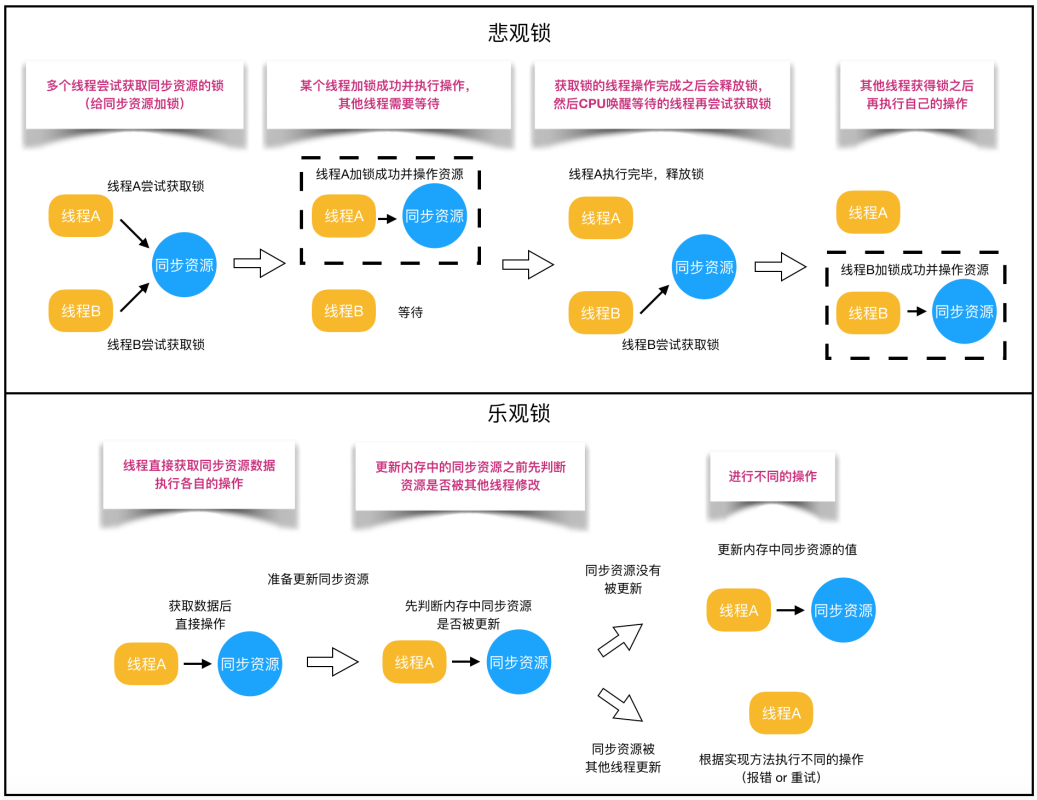
1. 乐观锁 VS 悲观锁

乐观锁与悲观锁是一种广义上的概念，体现了看待线程同步的不同角度。在Java和数据库中都有此概念对应的实际应用。

先说概念。对于同一个数据的并发操作，悲观锁认为自己在使用数据的时候一定有别的线程来修改数据，因此在获取数据的时候会先加锁，确保数据不会被别的线程修改。Java中，synchronized关键字和Lock的实现类都是悲观锁。

而乐观锁认为自己在使用数据时不会有别的线程修改数据，所以不会添加锁，只是在更新数据的时候去判断之前有没有别的线程更新了这个数据。如果这个数据没有被更新，当前线程将自己修改的数据成功写入。如果数据已经被其他线程更新，则根据不同的实现方式执行不同的操作（例如报错或者自动重试）。

乐观锁在Java中是通过使用无锁编程来实现，最常采用的是CAS算法，Java原子类中的递增操作就通过CAS自旋实现的。



根据从上面的概念描述我们可以发现：

- 悲观锁适合写操作多的场景，先加锁可以保证写操作时数据正确。
- 乐观锁适合读操作多的场景，不加锁的特点能够使其读操作的性能大幅提升。

光说概念有些抽象，我们来看下乐观锁和悲观锁的调用方式示例：

```
// ------------------------- 悲观锁的调用方式 -------------------------
// synchronized
public synchronized void testMethod() {
        // 操作同步资源
}
// ReentrantLock
private ReentrantLock lock = new ReentrantLock(); // 需要保证多个线程使用的是同一个锁
public void modifyPublicResources() {
        lock.lock();
        // 操作同步资源
        lock.unlock();
}

// ------------------------- 乐观锁的调用方式 -------------------------
private AtomicInteger atomicInteger = new AtomicInteger();  // 需要保证多个线程使用的是同一个AtomicInteger
atomicInteger.incrementAndGet(); //执行自增1
```

通过调用方式示例，我们可以发现悲观锁基本都是在显式的锁定之后再操作同步资源，而乐观锁则直接去操作同步资源。那么，为何乐观锁能够做到不锁定同步资源也可以正确的实现线程同步呢？我们通过介绍乐观锁的主要实现方式“CAS”的技术原理来为大家解惑。

CAS全称 Compare And Swap（比较与交换），是一种无锁算法。在不使用锁（没有线程被阻塞）的情况下实现多线程之间的变量同步。java.util.concurrent包中的原子类就是通过CAS来实现了乐观锁。

CAS算法涉及到三个操作数：

- 需要读写的内存值 V。
- 进行比较的值 A。
- 要写入的新值 B。

当且仅当 V 的值等于 A 时，CAS通过原子方式用新值B来更新V的值（“比较+更新”整体是一个原子操作），否则不会执行任何操作。一般情况下，“更新”是一个不断重试的操作。

之前提到java.util.concurrent包中的原子类，就是通过CAS来实现了乐观锁，那么我们进入原子类AtomicInteger的源码，看一下AtomicInteger的定义：

```
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField( name: "value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;
```

根据定义我们可以看出各属性的作用：

- unsafe： 获取并操作内存的数据。
- valueOffset： 存储value在AtomicInteger中的偏移量。
- value： 存储AtomicInteger的int值，该属性需要借助volatile关键字保证其在线程间是可见的。

接下来，我们查看AtomicInteger的自增函数incrementAndGet()的源码时，发现自增函数底层调用的是unsafe.getAndAddInt()。但是由于JDK本身只有Unsafe.class，只通过class文件中的参数名，并不能很好的了解方法的作用，所以我们通过OpenJDK 8 来查看Unsafe的源码：

```
// ------------------------ JDK 8 ------------------------
// AtomicInteger 自增方法
public final int incrementAndGet() {
  return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
}

// Unsafe.class
public final int getAndAddInt(Object var1, long var2, int var4) {
  int var5;
  do {
      var5 = this.getIntVolatile(var1, var2);
  } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));
  return var5;
}

// ------------------------ OpenJDK 8 ------------------------
```

```
// Unsafe.java
public final int getAndAddInt(Object o, long offset, int delta) {
    int v;
    do {
        v = getIntVolatile(o, offset);
    } while (!compareAndSwapInt(o, offset, v, v + delta));
    return v;
}
```

根据OpenJDK 8的源码我们可以看出，getAndAddInt()循环获取给定对象o中的偏移量处的值v，然后判断内存值是否等于v。如果相等则将内存值设置为 v + delta，否则返回false，继续循环进行重试，直到设置成功才能退出循环，并且将旧值返回。整个“比较+更新”操作封装在compareAndSwapInt()中，在JNI里是借助于一个CPU指令完成的，属于原子操作，可以保证多个线程都能够看到同一个变量的修改值。

后续JDK通过CPU的cmpxchg指令，去比较寄存器中的 A 和 内存中的值 V。如果相等，就把要写入的新值 B 存入内存中。如果不相等，就将内存值 V 赋值给寄存器中的值 A。然后通过Java代码中的while循环再次调用cmpxchg指令进行重试，直到设置成功为止。

CAS虽然很高效，但是它也存在三大问题，这里也简单说一下：

1. **ABA问题**。CAS需要在操作值的时候检查内存值是否发生变化，没有发生变化才会更新内存值。但是如果内存值原来是A，后来变成了B，然后又变成了A，那么CAS进行检查时会发现值没有发生变化，但是实际上是有变化的。ABA问题的解决思路就是在变量前面添加版本号，每次变量更新的时候都把版本号加一，这样变化过程就从“A－B－A”变成了“1A－2B－3A”。
    - JDK从1.5开始提供了AtomicStampedReference类来解决ABA问题，具体操作封装在compareAndSet()中。compareAndSet()首先检查当前引用和当前标志与预期引用和预期标志是否相等，如果都相等，则以原子方式将引用值和标志的值设置为给定的更新值。
2. **循环时间长开销大**。CAS操作如果长时间不成功，会导致其一直自旋，给CPU带来非常大的开销。
3. **只能保证一个共享变量的原子操作**。对一个共享变量执行操作时，CAS能够保证原子操作，但是对多个共享变量操作时，CAS是无法保证操作的原子性的。
    - Java从1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，可以把多个变量放在一个对象里来进行CAS操作。

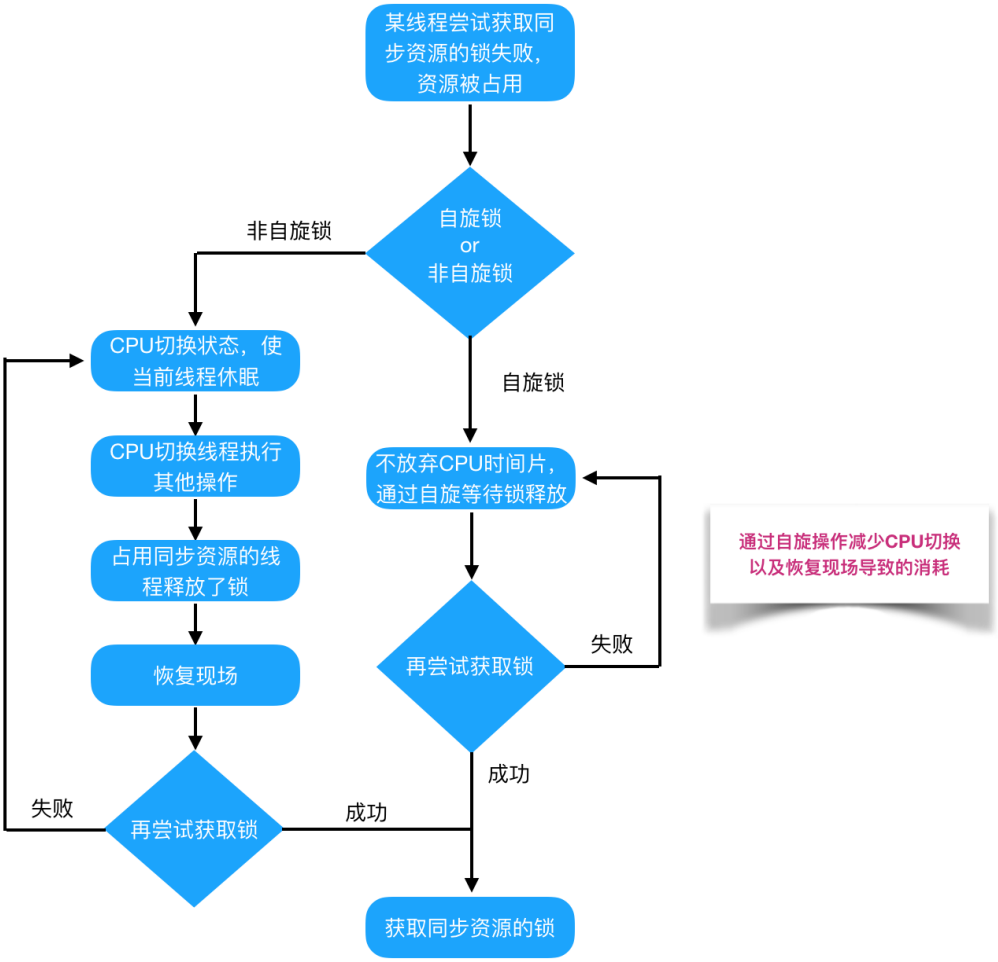
2. 自旋锁 VS 适应性自旋锁

在介绍自旋锁前，我们需要介绍一些前提知识来帮助大家明白自旋锁的概念。

阻塞或唤醒一个Java线程需要操作系统切换CPU状态来完成，这种状态转换需要耗费处理器时间。如果同步代码块中的内容过于简单，状态转换消耗的时间有可能比用户代码执行的时间还要长。

在许多场景中，同步资源的锁定时间很短，为了这一小段时间去切换线程，线程挂起和恢复现场的花费可能会让系统得不偿失。如果物理机器有多个处理器，能够让两个或以上的线程同时并行执行，我们就可以让后面那个请求锁的线程不放弃CPU的执行时间，看看持有锁的线程是否很快就会释放锁。

而为了让当前线程“稍等一下”，我们需让当前线程进行自旋，如果在自旋完成后前面锁定同步资源的线程已经释放了锁，那么当前线程就可以不必阻塞而是直接获取同步资源，从而避免切换线程的开销。这就是自旋锁。



自旋锁本身是有缺点的，它不能代替阻塞。自旋等待虽然避免了线程切换的开销，但它要占用处理器时间。如果锁被占用的时间很短，自旋等待的效果就会非常好。反之，如果锁被占用的时间很长，那么自旋的线程只会白浪费处理器资源。所以，自旋等待的时间必须要有一定的限度，如果自旋超过了限定次数（默认是10次，可以使用-XX:PreBlockSpin来更改）没有成功获得锁，就应当挂起线程。

自旋锁的实现原理同样也是CAS，AtomicInteger中调用unsafe进行自增操作的源码中的do-while循环就是一个自旋操作，如果修改数值失败则通过循环来执行自旋，直至修改成功。

```
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5,  var5: var5 + var4));

    return var5;
}
```

自旋锁在JDK1.4.2中引入，使用-XX:+UseSpinning来开启。JDK 6中变为默认开启，并且引入了自适应的自旋锁（适应性自旋锁）。

自适应意味着自旋的时间（次数）不再固定，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也是很有可能再次成功，进而它将允许自旋等待持续相对更长的时间。如果对于某个

锁，自旋很少成功获得过，那在以后尝试获取这个锁时将可能省略掉自旋过程，直接阻塞线程，避免浪费处理器资源。

在自旋锁中 另有三种常见的锁形式:TicketLock、CLHlock和MCSlock，本文中仅做名词介绍，不做深入讲解，感兴趣的同学可以自行查阅相关资料。

3. 无锁 VS 偏向锁 VS 轻量级锁 VS 重量级锁

这四种锁是指锁的状态，专门针对synchronized的。在介绍这四种锁状态之前还需要介绍一些额外的知识。

首先为什么Synchronized能实现线程同步?

在回答这个问题之前我们需要了解两个重要的概念：“Java对象头”、“Monitor”。

Java对象头

synchronized是悲观锁，在操作同步资源之前需要给同步资源先加锁，这把锁就是存在Java对象头里的，而Java对象头又是什么呢?

我们以Hotspot虚拟机为例，Hotspot的对象头主要包括两部分数据：Mark Word（标记字段）、Klass Pointer（类型指针）。

Mark Word：默认存储对象的HashCode，分代年龄和锁标志位信息。这些信息都是与对象自身定义无关的数据，所以Mark Word被设计成一个非固定的数据结构以便在极小的空间内存存储尽量多的数据。它会根据对象的状态复用自己的存储空间，也就是说在运行期间Mark Word里存储的数据会随着锁标志位的变化而变化。

Klass Point：对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

Monitor

Monitor可以理解为一个同步工具或一种同步机制，通常被描述为一个对象。每一个Java对象就有一把看不见的锁，称为内部锁或者Monitor锁。

Monitor是线程私有的数据结构，每一个线程都有一个可用monitor record列表，同时还有一个全局的可用列表。每一个被锁住的对象都会和一个monitor关联，同时monitor中有一个Owner字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用。

现在话题回到synchronized，synchronized通过Monitor来实现线程同步，Monitor是依赖于底层的操作系统的Mutex Lock（互斥锁）来实现的线程同步。

如同我们在自旋锁中提到的“阻塞或唤醒一个Java线程需要操作系统切换CPU状态来完成，这种状态转换需要耗费处理器时间。如果同步代码块中的内容过于简单，状态转换消耗的时间有可能比用户代码执行的时间还要长”。这种方式就是synchronized最初实现同步的方式，这就是JDK 6之前synchronized效率低

的原因。这种依赖于操作系统Mutex Lock所实现的锁我们称之为“重量级锁”，JDK 6中为了减少获得锁和释放锁带来的性能消耗，引入了“偏向锁”和“轻量级锁”。

所以目前锁一共有4种状态，级别从低到高依次是：无锁、偏向锁、轻量级锁和重量级锁。锁状态只能升级不能降级。

通过上面的介绍，我们对synchronized的加锁机制以及相关知识有了一个了解，那么下面我们给出四种锁状态对应的的Mark Word内容，然后再分别讲解四种锁状态的思路以及特点：

锁状态	存储内容	存储内容
无锁	对象的hashCode、对象分代年龄、是否是偏向锁（0）	01
偏向锁	偏向线程ID、偏向时间戳、对象分代年龄、是否是偏向锁（1）	01
轻量级锁	指向栈中锁记录的指针	00
重量级锁	指向互斥量（重量级锁）的指针	10

无锁

无锁没有对资源进行锁定，所有的线程都能访问并修改同一个资源，但同时只有一个线程能修改成功。

无锁的特点就是修改操作在循环内进行，线程会不断的尝试修改共享资源。如果没有冲突就修改成功并退出，否则就会继续循环尝试。如果有多个线程修改同一个值，必定会有一个线程能修改成功，而其他修改失败的线程会不断重试直到修改成功。上面我们介绍的CAS原理及应用即是无锁的实现。无锁无法全面代替有锁，但无锁在某些场合下的性能是非常高的。

偏向锁

偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁，降低获取锁的代价。

在大多数情况下，锁总是由同一线程多次获得，不存在多线程竞争，所以出现了偏向锁。其目标就是在只有一个线程执行同步代码块时能够提高性能。

当一个线程访问同步代码块并获取锁时，会在Mark Word里存储锁偏向的线程ID。在线程进入和退出同步块时不再通过CAS操作来加锁和解锁，而是检测Mark Word里是否存储着指向当前线程的偏向锁。引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令即可。

偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，线程不会主动释放偏向锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有字节码正在执行），它会首先暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态。撤销偏向锁后恢复到无锁（标志位为“01”）或轻量级锁（标志位为“00”）的状态。

偏向锁在JDK 6及以后的JVM里是默认启用的。可以通过JVM参数关闭偏向锁：–XX:–UseBiasedLocking=false，关闭之后程序默认会进入轻量级锁状态。

轻量级锁

是指当锁是偏向锁的时候，被另外的线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，从而提高性能。

在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，然后拷贝对象头中的Mark Word复制到锁记录中。

拷贝成功后，虚拟机将使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针，并将Lock Record里的owner指针指向对象的Mark Word。

如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为“00”，表示此对象处于轻量级锁定状态。

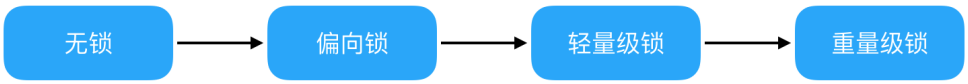
如果轻量级锁的更新操作失败了，虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行，否则说明多个线程竞争锁。

若当前只有一个等待线程，则该线程通过自旋进行等待。但是当自旋超过一定的次数，或者一个线程在持有锁，一个在自旋，又有第三个来访时，轻量级锁升级为重量级锁。

重量级锁

升级为重量级锁时，锁标志的状态值变为“10”，此时Mark Word中存储的是指向重量级锁的指针，此时等待锁的线程都会进入阻塞状态。

整体的锁状态升级流程如下：



综上，偏向锁通过对比Mark Word解决加锁问题，避免执行CAS操作。而轻量级锁是通过用CAS操作和自旋来解决加锁问题，避免线程阻塞和唤醒而影响性能。重量级锁是将除了拥有锁的线程以外的线程都阻塞。

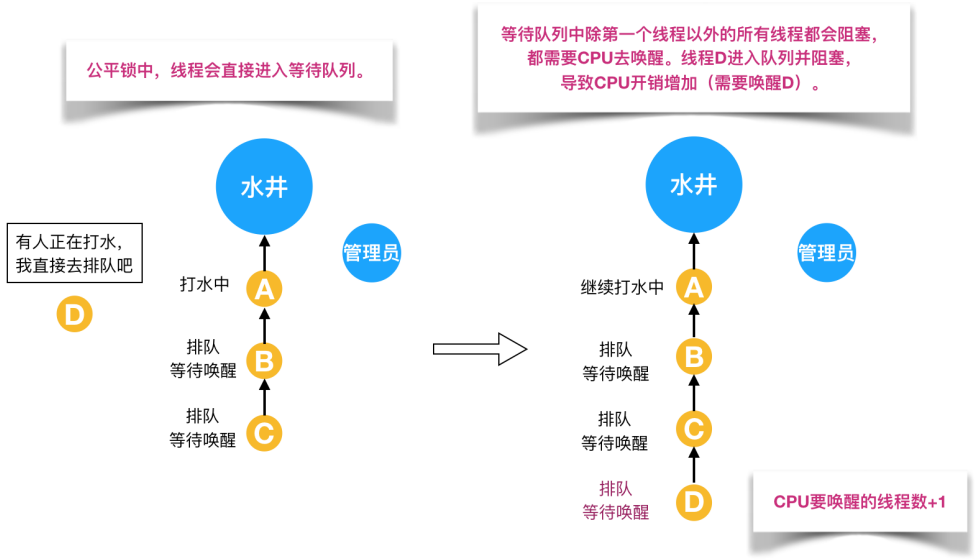
4. 公平锁 VS 非公平锁

公平锁是指多个线程按照申请锁的顺序来获取锁，线程直接进入队列中排队，队列中的第一个线程才能获得锁。公平锁的优点是等待锁的线程不会饿死。缺点是整体吞吐效率相对非公平锁要低，等待队列中除第一个线程以外的所有线程都会阻塞，CPU唤醒阻塞线程的开销比非公平锁大。

非公平锁是多个线程加锁时直接尝试获取锁，获取不到才会到等待队列的队尾等待。但如果此时锁刚好可用，那么这个线程可以无需阻塞直接获取到锁，所以非公平锁有可能出现后申请锁的线程先获取锁的场

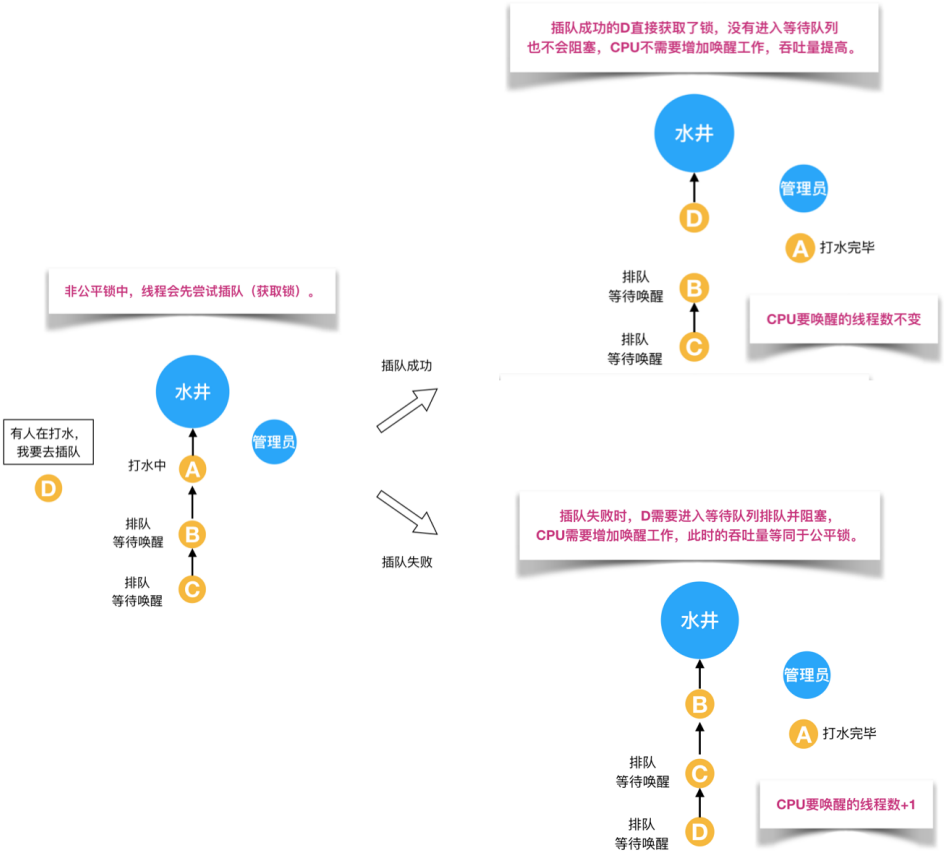
景。非公平锁的优点是可以减少唤起线程的开销，整体的吞吐效率高，因为线程有几率不阻塞直接获得锁，CPU不必唤醒所有线程。缺点是处于等待队列中的线程可能会饿死，或者等很久才会获得锁。

直接用语言描述可能有点抽象，这里作者用从别处看到的一个例子来讲述一下公平锁和非公平锁。



如上图所示，假设有一口水井，有管理员看守，管理员有一把锁，只有拿到锁的人才能够打水，打完水要把锁还给管理员。每个过来打水的人都要管理员的允许并拿到锁之后才能去打水，如果前面有人正在打水，那么这个想要打水的人就必须排队。管理员会查看下一个要去打水的人是不是队伍里排最前面的人，如果是的话，才会给你锁让你去打水；如果你不是排第一的人，就必须去队尾排队，这就是公平锁。

但是对于非公平锁，管理员对打水的人没有要求。即使等待队伍里有排队等待的人，但如果在上一个人刚打完水把锁还给管理员而且管理员还没有允许等待队伍里下一个人去打水时，刚好来了一个插队的人，这个插队的人是可以直接从管理员那里拿到锁去打水，不需要排队，原本排队等待的人只能继续等待。如下图所示：



接下来我们通过ReentrantLock的源码来讲解公平锁和非公平锁。

```
public class ReentrantLock implements Lock, java.io.Serializable {
    private static final long serialVersionUID = 7373984872572414699L;
    /** Synchronizer providing all implementation mechanics */
    private final Sync sync;

    /**...*/
    abstract static class Sync extends AbstractQueuedSynchronizer {...}

    /**...*/
    static final class NonfairSync extends Sync {...}

    /**...*/
    static final class FairSync extends Sync {...}

    /**...*/
    public ReentrantLock() { sync = new NonfairSync(); }

    /**...*/
    public ReentrantLock(boolean fair) { sync = fair ? new FairSync() : new NonfairSync(); }
```

根据代码可知，ReentrantLock里面有一个内部类Sync，Sync继承AQS（AbstractQueuedSynchronizer），添加锁和释放锁的大部分操作实际上都是在Sync中实现的。它有公平锁FairSync和非公平锁NonfairSync两个子类。ReentrantLock默认使用非公平锁，也可以通过构造器来显示的指定使用公平锁。

下面我们来看一下公平锁与非公平锁的加锁方法的源码:

公平锁

```
/**...*/
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (!hasQueuedPredecessors() &&
            compareAndSetState( expect: 0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

非公平锁

```
/**...*/
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState( expect: 0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

通过上图中的源代码对比，我们可以明显的看出公平锁与非公平锁的lock()方法唯一的区别就在于公平锁在获取同步状态时多了一个限制条件：hasQueuedPredecessors()。

```
/**...*/
public final boolean hasQueuedPredecessors() {
    //...
    Node t = tail; // Read fields in reverse initialization order
    Node h = head;
    Node s;
    return h != t &&
        ((s = h.next) == null || s.thread != Thread.currentThread());
}
```

再进入hasQueuedPredecessors()，可以看到该方法主要做一件事情：主要是判断当前线程是否位于同步队列中的第一个。如果是则返回true，否则返回false。

综上，公平锁就是通过同步队列来实现多个线程按照申请锁的顺序来获取锁，从而实现公平的特性。非公平锁加锁时不考虑排队等待问题，直接尝试获取锁，所以存在后申请却先获得锁的情况。

5. 可重入锁 VS 非可重入锁

可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，再进入该线程的内层方法会自动获取锁（前提锁对象得是同一个对象或者class），不会因为之前已经获取过还没释放而阻塞。Java中ReentrantLock和synchronized都是可重入锁，可重入锁的一个优点是可一定程度避免死锁。下面用示例代码来进行分析：

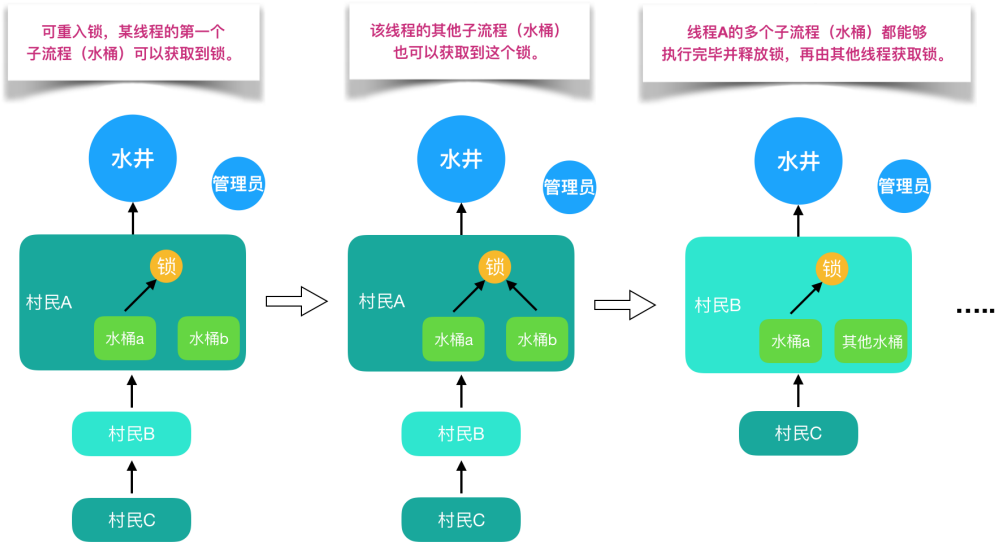
```
public class Widget {
    public synchronized void doSomething() {
        System.out.println("方法1执行...");
        doOthers();
    }

    public synchronized void doOthers() {
        System.out.println("方法2执行...");
    }
}
```

在上面的代码中，类中的两个方法都是被内置锁synchronized修饰的，doSomething()方法中调用doOthers()方法。因为内置锁是可重入的，所以同一个线程在调用doOthers()时可以直接获得当前对象的锁，进入doOthers()进行操作。

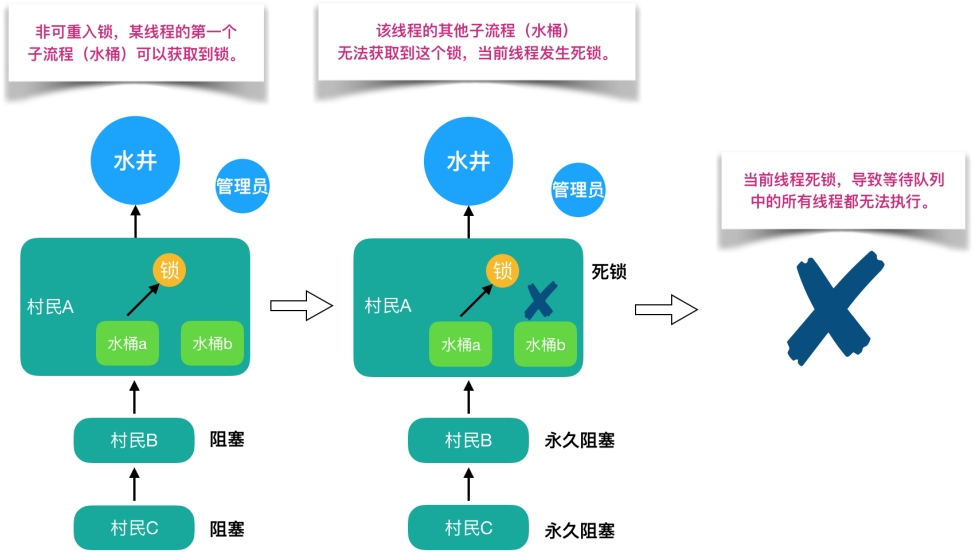
如果是一个不可重入锁，那么当前线程在调用doOthers()之前需要将执行doSomething()时获取当前对象的锁释放掉，实际上该对象锁已被当前线程所持有，且无法释放。所以此时会出现死锁。

而为什么可重入锁就可以在嵌套调用时可以自动获得锁呢？我们通过图示和源码来分别解析一下。

还是打水的例子，有多个人在排队打水，此时管理员允许锁和同一个人的多个水桶绑定。这个人用多个水桶打水时，第一个水桶和锁绑定并打完水之后，第二个水桶也可以直接和锁绑定并开始打水，所有的水桶都打完水之后打水人才会将锁还给管理员。这个人的所有打水流程都能够成功执行，后续等待的人也能够打到水。这就是可重入锁。



但如果是非可重入锁的话，此时管理员只允许锁和同一个人的一个水桶绑定。第一个水桶和锁绑定打完水之后并不会释放锁，导致第二个水桶不能和锁绑定也无法打水。当前线程出现死锁，整个等待队列中的所有线程都无法被唤醒。



之前我们说过ReentrantLock和synchronized都是重入锁，那么我们通过重入锁ReentrantLock以及非可重入锁NonReentrantLock的源码来对比分析一下为什么非可重入锁在重复调用同步资源时会出现死锁。

首先ReentrantLock和NonReentrantLock都继承父类AQS，其父类AQS中维护了一个同步状态status来计数重入次数，status初始值为0。

当线程尝试获取锁时，可重入锁先尝试获取并更新status值，如果status == 0表示没有其他线程在执行同步代码，则把status置为1，当前线程开始执行。如果status != 0，则判断当前线程是否是获取到这个锁的线程，如果是的话执行status+1，且当前线程可以再次获取锁。而非可重入锁是直接去获取并尝试更新当前status的值，如果status != 0的话会导致其获取锁失败，当前线程阻塞。

释放锁时，可重入锁同样先获取当前status的值，在当前线程是持有锁的线程的前提下。如果status-1 == 0，则表示当前线程所有重复获取锁的操作都已经执行完毕，然后该线程才会真正释放锁。而非可重入锁则是在确定当前线程是持有锁的线程之后，直接将status置为0，将锁释放。

可重入锁

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState( expect: 0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true; 获取锁时先判断，如果当前线程就是已经
    }                占有锁的线程，则status值+1，并返回true。
    return false;
}

protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

释放锁时也是先判断当前线程是否是已占有锁的线程，然后在判断status。如果status等于0，才真正的释放锁

非可重入锁

```
protected boolean tryAcquire(int acquires) {
    if (this.compareAndSetState( expect: 0,  update: 1)) {
        this.owner = Thread.currentThread();
        return true;
    } else {
        return false;
    }
}

protected boolean tryRelease(int releases) {
    if (Thread.currentThread() != this.owner) {
        throw new IllegalMonitorStateException();
    } else {
        this.owner = null;
        this.setState(0);
        return true;
    }
}
```

非重入锁是直接尝试获取锁

释放锁时也是直接将status置为0

6. 独享锁 VS 共享锁

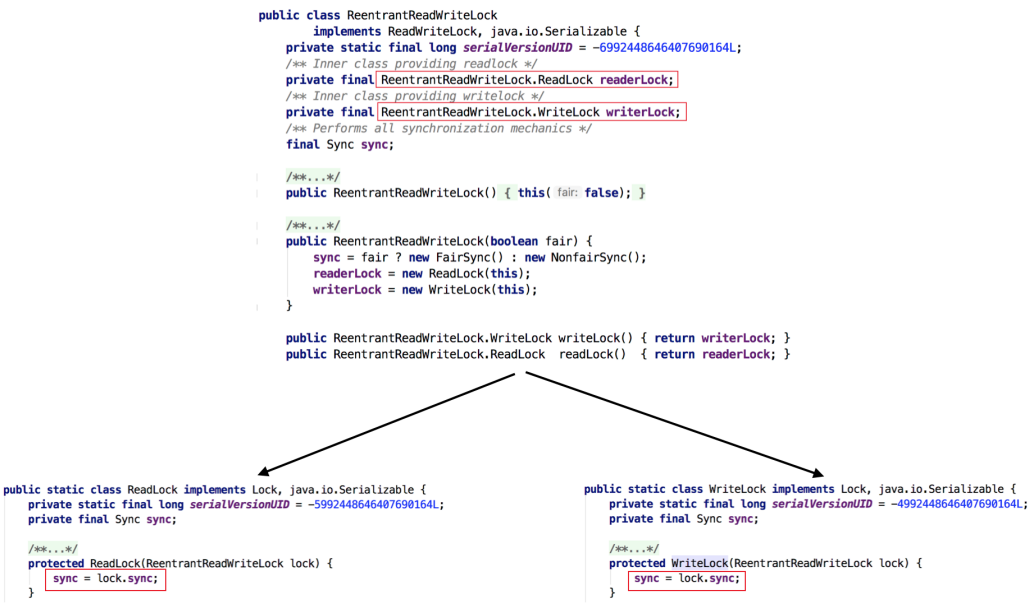
独享锁和共享锁同样是一种概念。我们先介绍一下具体的概念，然后通过ReentrantLock和ReentrantReadWriteLock的源码来介绍独享锁和共享锁。

独享锁也叫排他锁，是指该锁一次只能被一个线程所持有。如果线程T对数据A加上排它锁后，则其他线程不能再对A加任何类型的锁。获得排它锁的线程即能读数据又能修改数据。JDK中的synchronized和JUC中Lock的实现类就是互斥锁。

共享锁是指该锁可被多个线程所持有。如果线程T对数据A加上共享锁后，则其他线程只能对A再加共享锁，不能加排它锁。获得共享锁的线程只能读数据，不能修改数据。

独享锁与共享锁也是通过AQS来实现的，通过实现不同的方法，来实现独享或者共享。

下图为ReentrantReadWriteLock的部分源码：



我们看到ReentrantReadWriteLock有两把锁：ReadLock和WriteLock，由词知意，一个读锁一个写锁，合称“读写锁”。再进一步观察可以发现ReadLock和WriteLock是靠内部类Sync实现的锁。Sync是AQS的一个子类，这种结构在CountDownLatch、ReentrantLock、Semaphore里面也都存在。

在ReentrantReadWriteLock里面，读锁和写锁的锁主体都是Sync，但读锁和写锁的加锁方式不一样。读锁是共享锁，写锁是独享锁。读锁的共享锁可保证并发读非常高效，而读写、写读、写写的过程互斥，因为读锁和写锁是分离的。所以ReentrantReadWriteLock的并发性相比一般的互斥锁有了很大提升。

那读锁和写锁的具体加锁方式有什么区别呢？在了解源码之前我们需要回顾一下其他知识。 在最开始提及AQS的时候我们也提到了state字段（int类型，32位），该字段用来描述有多少线程获持有锁。

在独享锁中这个值通常是0或者1（如果是重入锁的话state值就是重入的次数），在共享锁中state就是持有锁的数量。但是在ReentrantReadWriteLock中有读、写两把锁，所以需要在一个整型变量state上分别描述读锁和写锁的数量（或者也可以叫状态）。于是将state变量“按位切割”切分成了两个部分，高16位表示读锁状态（读锁个数），低16位表示写锁状态（写锁个数）。如下图所示：

32位																													
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
高十六位，读状态																低十六位，写状态													

了解了概念之后我们再来看代码，先看写锁的加锁源码：

```
protected final boolean tryAcquire(int acquires) {
        Thread current = Thread.currentThread();
        int c = getState(); // 取到当前锁的个数
        int w = exclusiveCount(c); // 取写锁的个数w
        if (c != 0) { // 如果已经有线程持有了锁(c!=0)
    // (Note: if c != 0 and w == 0 then shared count != 0)
                if (w == 0 || current != getExclusiveOwnerThread()) // 如果写线程数（w）为0（换言之存在读锁） 或者持有锁的线程不是当前线程就返回失败
                        return false;
                if (w + exclusiveCount(acquires) > MAX_COUNT)     // 如果写入锁的数量大于最大数（65535，2的16次方-1）就抛出一个Error。
      throw new Error("Maximum lock count exceeded");
                // Reentrant acquire
    setState(c + acquires);
    return true;
  }
  if (writerShouldBlock() || !compareAndSetState(c, c + acquires)) // 如果当且写线程数为0，并且当前线程需要阻塞那么就返回失败；或者如果通过CAS增加写线程数失败也返回失败。
                return false;
```

```
        setExclusiveOwnerThread(current); // 如果c=0, w=0或者c>0, w>0（重入），则设置当前线程或锁的拥有者
        return true;
}
```

- 这段代码首先取到当前锁的个数c，然后再通过c来获取写锁的个数w。因为写锁是低16位，所以取低16位的最大值与当前的c做与运算（ int w = exclusiveCount©; ），高16位和0与运算后是0，剩下的就是低位运算的值，同时也是持有写锁的线程数目。
- 在取到写锁线程的数目后，首先判断是否已经有线程持有了锁。如果已经有线程持有了锁(c!=0)，则查看当前写锁线程的数目，如果写线程数为0（即此时存在读锁）或者持有锁的线程不是当前线程就返回失败（涉及到公平锁和非公平锁的实现）。
- 如果写入锁的数量大于最大数（65535，2的16次方-1）就抛出一个Error。
- 如果当且写线程数为0（那么读线程也应该为0，因为上面已经处理c!=0的情况），并且当前线程需要阻塞那么就返回失败；如果通过CAS增加写线程数失败也返回失败。
- 如果c=0,w=0或者c>0,w>0（重入），则设置当前线程或锁的拥有者，返回成功！

tryAcquire()除了重入条件（当前线程为获取了写锁的线程）之外，增加了一个读锁是否存在的判断。如果存在读锁，则写锁不能被获取，原因在于：必须确保写锁的操作对读锁可见，如果允许读锁在已被获取的情况下对写锁的获取，那么正在运行的其他读线程就无法感知到当前写线程的操作。

因此，只有等待其他读线程都释放了读锁，写锁才能被当前线程获取，而写锁一旦被获取，则其他读写线程的后续访问均被阻塞。写锁的释放与ReentrantLock的释放过程基本类似，每次释放均减少写状态，当写状态为0时表示写锁已被释放，然后等待的读写线程才能够继续访问读写锁，同时前次写线程的修改对后续的读写线程可见。

接着是读锁的代码：

```
protected final int tryAcquireShared(int unused) {
    Thread current = Thread.currentThread();
    int c = getState();
    if (exclusiveCount(c) != 0 &&
        getExclusiveOwnerThread() != current)
        return -1;                                  // 如果其他线程已经获取了写锁，则当前线程获取读锁失败，进入等待状态
    int r = sharedCount(c);
    if (!readerShouldBlock() &&
        r < MAX_COUNT &&
        compareAndSetState(c, c + SHARED_UNIT)) {
        if (r == 0) {
            firstReader = current;
            firstReaderHoldCount = 1;
        } else if (firstReader == current) {
            firstReaderHoldCount++;
        } else {
            HoldCounter rh = cachedHoldCounter;
            if (rh == null || rh.tid != getThreadId(current))
                cachedHoldCounter = rh = readHolds.get();
            else if (rh.count == 0)
                readHolds.set(rh);
            rh.count++;
        }
        return 1;
    }
    return fullTryAcquireShared(current);
}
```

可以看到在tryAcquireShared(int unused)方法中，如果其他线程已经获取了写锁，则当前线程获取读锁失败，进入等待状态。如果当前线程获取了写锁或者写锁未被获取，则当前线程（线程安全，依靠CAS保证）增加读状态，成功获取读锁。读锁的每次释放（线程安全的，可能有多个读线程同时释放读锁）均减

少读状态，减少的值是“1<<16”。所以读写锁才能实现读读的过程共享，而读写、写读、写写的过程互斥。

此时，我们再回头看一下互斥锁ReentrantLock中公平锁和非公平锁的加锁源码：

公平锁

```
/**...*/
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (!hasQueuedPredecessors() &&
            compareAndSetState( expect: 0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

非公平锁

```
/**...*/
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState( expect: 0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

我们发现公平锁和非公平锁如果在当前线程不是拥有锁的线程时，就不能添加锁。
所以它们添加的都是独享锁！

我们发现在ReentrantLock虽然有公平锁和非公平锁两种，但是它们添加的都是独享锁。根据源码所示，当某一个线程调用lock方法获取锁时，如果同步资源没有被其他线程锁住，那么当前线程在使用CAS更新state成功后就会成功抢占该资源。而如果公共资源被占用且不是被当前线程占用，那么就会加锁失败。所以可以确定ReentrantLock无论读操作还是写操作，添加的锁都是都是独享锁。

结语

本文Java中常用的锁以及常见的锁的概念进行了基本介绍，并从源码以及实际应用的角度进行了对比分析。限于篇幅以及个人水平，没有在本篇文章中对所有内容进行深层次的讲解。

其实Java本身已经对锁本身进行了良好的封装，降低了研发同学在平时工作中的使用难度。但是研发同学也需要熟悉锁的底层原理，不同场景下选择最适合的锁。而且源码中的思路都是非常好的思路，也是值得大家去学习和借鉴的。

参考资料

1. 《Java并发编程艺术》
2. Java中的锁🔗
3. Java CAS 原理剖析🔗
4. Java并发——关键字synchronized解析🔗
5. Java synchronized原理总结🔗
6. 聊聊并发（二）——Java SE1.6中的Synchronized🔗
7. 深入理解读写锁—ReadWriteLock源码分析🔗
8. 【JUC】JDK1.8源码分析之ReentrantReadWriteLock🔗
9. Java多线程（十）之ReentrantReadWriteLock深入分析🔗
10. Java—读写锁的实现原理🔗

作者简介

- 家琪，美团点评后端工程师。2017 年加入美团点评，负责美团点评境内度假的业务开发。

境外业务性能优化实践

作者: 云霜

> 本文根据第16期美团技术线上沙龙OnLine演讲内容整理而成。

前言

性能问题简介

应用性能是产品用户体验的基石，性能优化的终极目标是优化用户体验。当我们谈及性能，最直观能想到的一个词是“快”，Strangeloop在对众多的网站做性能分析之后得出了一个著名的3s定律“页面加载速度超过3s，57%的访客会离开”,可见页面加载速度对于互联网产品的重要性。速度在Google、百度等搜索引擎的PR评分中也占有一定的比例，会影响到网站的SEO排名。“天下武功，唯快不破”，套在性能上面也非常适用。

性能指标

性能优化是个系统性工程，涉及到后端、前端、移动端、系统网络及各种基础设施，每一块都需要做各自的性能优化。当我们系统的分析性能问题时，可以通过以下指标来衡量：

- Web端：首屏时间、白屏时间、可交互时间、完全加载时间等。

  首屏时间是指从用户打开网页开始到浏览器第一屏渲染完成的时间，是最直接的用户感知体验指标，也是性能领域公认的最重要的核心指标。

  首屏时间 = DNS时间 + 建立连接时间 + 后端响应时间 + 网络传输时间 + 首屏页面渲染时间

- 移动端：Crash率、内存使用率、FPS（Frames Per Second, 每秒传输帧数）、端到端响应时间等。

  Native相比于H5在交互体验方面有更多的优势，FPS是体现页面顺畅程度的一个重要指标，另外移动端开发同学还需要关注App进程的CPU使用率、内存使用率等系统性能指标。端到端响应时间是衡量一个API性能的关键指标，比纯后端响应时间更全面，它会受到DNS、网络带宽、网络链路、HTTP Payload等多个因素的影响。端到端响应时间是DNS解析时间、网络传输时间及后端响应时间的总和。

- 后端：响应时间（RT）、吞吐量（TPS）、并发数等。

  后端系统响应时间是指系统对请求做出响应的时间（应用延迟时间），对于面向用户的Web服务，响应时间能很好度量应用性能，会受到数据库查询、RPC调用、网络IO、逻辑计算复杂度、JVM垃圾回收等多方面因素影响。对于高并发的应用和系统，吞吐量是个非常重要的指标，它与request对CPU、内存资源的消耗，调用的外部接口及IO等紧密关联。

影响性能的因素

互联网产品是创意、设计、研发、系统、网络、硬件、运维等众多资源相互交织的集合体，性能受多方面因素影响，犹如一只木桶，木桶能盛多少水，取决于最短的那块木板，也可称之为短板效应。影响产品性能的因素有：

1. 产品逻辑与用户行为

产品逻辑过于复杂、功能交互过于丰富、产品设计过于绚丽、页面元素素材过多等都会影响产品性能。

2. 基础网络

中国的基础网络是世界上最复杂的基础网络，国内的网络运营商众多且各自为政，互联互通成本很高。对于境外业务来说更是要面对国内国际网络交互的情况，再加上GFW的存在，网络延迟、丢包现象非常严重。

3. 代码及应用

开发语言瓶颈、代码质量及系统架构等都会影响系统性能，常见的代码及应用问题有：

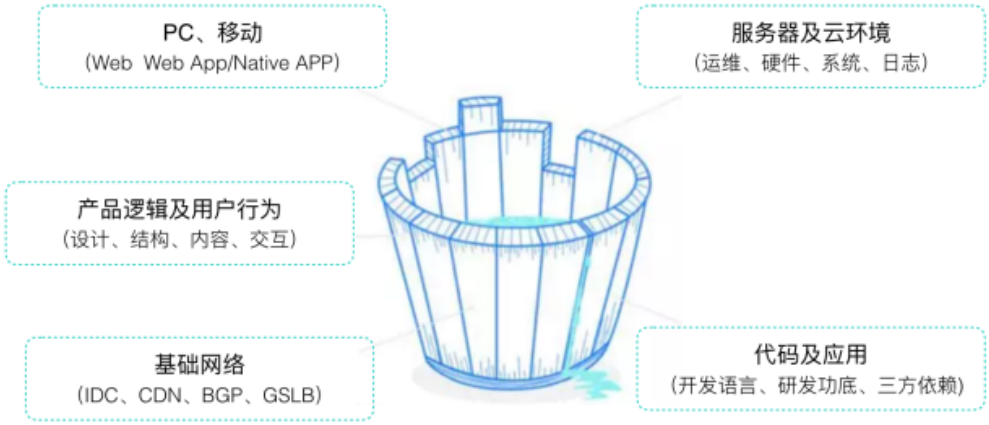
- 架构不合理。业务发展超越架构支撑能力而导致系统负荷过载，进而导致出现系统奔溃、响应超时等现象。另外不合理的架构如：单点、无cache、应用混部署、没有考虑分布式、集群化等也都会影响性能。
  - 研发功底和经验不足。开发的App、Server效率和性能较低、不稳定也是常见的事情。
  - 没有性能意识，只实现了业务功能不注意代码性能，新功能上线后整体性能下降，或当业务上量后系统出现连锁反应，导致性能问题叠加，直接影响用户体验。
  - 多数的性能问题发生在数据库上。由慢SQL（详情可以参考美团技术团队博客的经典文章《MySQL索引原理及慢查询优化》🔗）、过多查询等原因造成的数据库瓶颈，没有做读写分离、分库分表等。

4. 移动端环境

移动互联网时代，移动端环境的复杂性对产品的性能影响也很大，比如用户的设备类型、设备性能、操作系统类型、系统版本及网络类型（电信/移动/联通，Wi-Fi/4G/3G/2G）等。

5. 硬件及云环境

硬件的发展遵循着摩尔定律，生命周期一般都很短，服务器老化或其他硬件问题经常会导致应用故障。IDC、机架、服务器、内存、磁盘、网卡等不同硬件和操作系统上运行的应用性能差距可以达到数十倍之多。



关于性能优化这个主题，美团技术团队博客之前也发表过不少系统阐述的文章，比如《常见性能优化策略的总结》和《性能优化模式》，可以参考。

境外业务的特点

美团旅行境外业务主要包含商户信息、用户评论、特色美食、出境线路、境外当地玩乐、购物优惠券等6大板块，满足用户行前到行中的出境场景。在覆盖城市上，目前已经100%覆盖全球热门旅游目的地，大家出境游玩时都可以体验使用。根据官方提供的数据，超过30%的自由行游客在出境时会选择使用美团，当前美团已经成为除微信之外，国人在境外最喜欢打开的移动App之一。

境外业务与其他境内业务相比，区别主要表现在以下及方面：

用户在境外访问

境外业务很大一部分流量来自境外访问，国外网络情况十分复杂，一些国家的网络基础设施很差，4G覆盖率很低，从国外访问国内机房，不仅网络链路长，还涉及到跨网跨运营商跨GFW的访问情况，访问延迟、网络丢包等情况非常严重。

大量使用Hybrid实现

由于业务发展很快，业务新增及变更也相对频繁，为适应业务的快速发展，我们大量采用的是H5方式实现，大量使用Hybrid模式。H5相比Native页面需要加载的内容更多，对网络环境的要求更高。

与境外商家对接

除了用户在境外访问，境外业务还会和很多境外的商家、供应商或代理商有合作对接，同样面临着跨国网络访问的问题。

基于以上背景，如何提升产品性能，做到像国内业务一样，其中面临了很多的技术挑战。本文将从网络优化、前端优化、后端优化几个方面来介绍境外业务在性能优化方面的做过的一些事情。

网络优化

网络问题

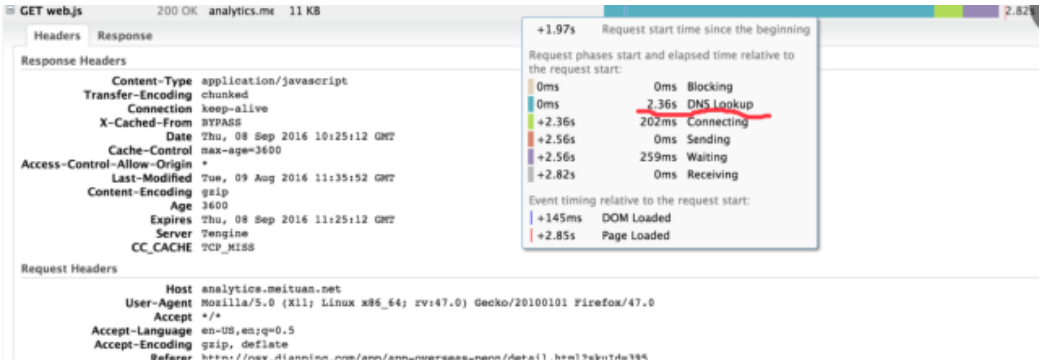
影响网络性能的问题有很多，常见的网络问题有以下几类：

问题一：DNS问题

DNS问题最容易被大家所忽视，而实际上DNS出问题的概率非常大，DNS问题主要有2类：

一类是DNS解析慢或解析失败，我们统计过一些数据，我们的域名在国内DNS解析耗时大概在30–120ms之间，而国外网络下耗时达到200–300ms左右。在2G/3G等弱网环境下，DNS解析失败非常常见。DNS对于首次网络访问的耗时及网络成功率会有很大的影响。

下图是我们利用页面测速工具(GTmatrix)在加拿大温哥华节点测试的一个页面首次访问时的网络请求情况，可以看出当用户在加拿大第一次访问且运营商LocalDNS无NS缓存记录时，DNS解析耗时2.36s。



另一类常见问题是DNS劫持或失效，乌云（WooYun）上曾报过多起因域名服务商安全漏洞被黑客利用导致网站NS记录被篡改的case。而更多的DNS劫持问题则是来自于网络运营商的作恶，主要有以下几种：

1. 本地缓存，各运营商确保网内访问，同时减少跨网结算，运营商在网内搭建了内容缓存服务器，把用户域名强行指向内容缓存服务器。
2. 内容劫持，有部分LocalDNS会把部分域名解析指向内容缓存，并替换成第三方广告联盟的广告。
3. 解析转发，运营商的LocalDNS还存在解析转发的形象。指运营商自身不进行域名递归解析，而是把域名解析请求转发到其他运营商的递归DNS上，解析请求的来源IP成了其他运营商的IP，从而导致用户请求跨网访问，性能变差。

问题二：网络链路问题

链路过长、请求经过的路由转发跳数过多、跨网访问等都是影响网络传输性能的关键因素。另外网络攻击（主要是DDoS、CC攻击等洪水攻击）流量也影响着网络链路的稳定性。据统计，骨干网上每天有数百G的流量为攻击流量。

问题三：传输Payload大小

移动设备的网络在非Wi–Fi环境下时通常不太稳定，再加上有TCP拥塞策略的限制，数据传输量越大，传的就越慢。我们需要尽量的减少数据传输量。通常的做法有：数据压缩、图片压缩、选择更高效的序列化算法（比如Protocol Buffers）等。

我们在网络优化方面主要做了以下几件事情：

1. CDN优化：海外CDN加速、CDN缓存预热。
2. DNS Prefetch：DNS预热，刷新移动设备系统/VM的DNS缓存。
3. 长连接：“代理长连接”Shark，专线链路优化，并且有效解决了DNS的瓶颈问题。

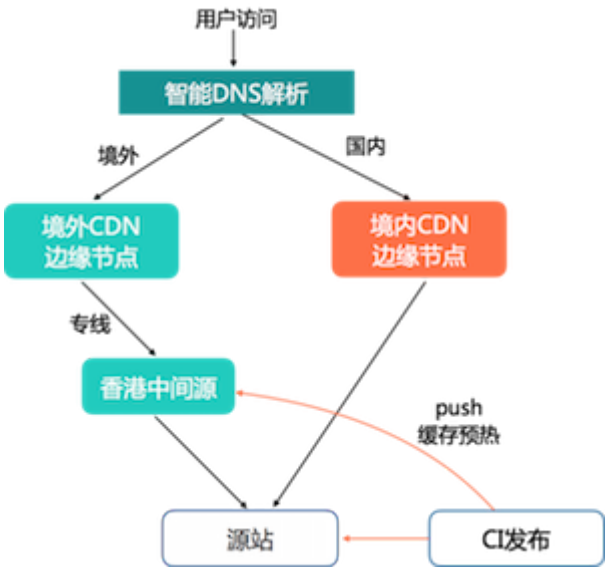
4. 网络链路优化：通过专线和代理，解决公网链路长及网络抖动不稳定的问题。

接下来，我们分别详细阐述一下。

CDN优化

CDN服务的好坏主要取决于节点部署的覆盖程度、带宽以及调度算法。对国内业务而言，使用蓝汛、网宿、帝联等老牌CDN服务商或腾讯云、阿里云、UPYUN等云CDN服务商性能都很好，但这些CDN服务商在境外的节点很少。为了提升境外的静态资源加速效果，我们尝试对接了很多国外的知名CDN服务商。通过智能DNS解析用户的IP来源，如果是境外访问则CNAME到国外CDN，国内访问时仍然走的是国内CDN。

CDN也是一种缓存，是缓存就不得不谈命中率的问题。 如果用户在境外访问时CDN未命中，静态资源从境外回源到国内源站获取，成本非常高。为了提升缓存命中率，我们的做法是在香港搭了一个CDN中间源，在前端资源发布时会调用CDN的push接口把资源预热到中间源，保证当境外边缘节点缓存未命中时无需再回源到国内IDC，只需从中间源获取。



DNS Prefetch

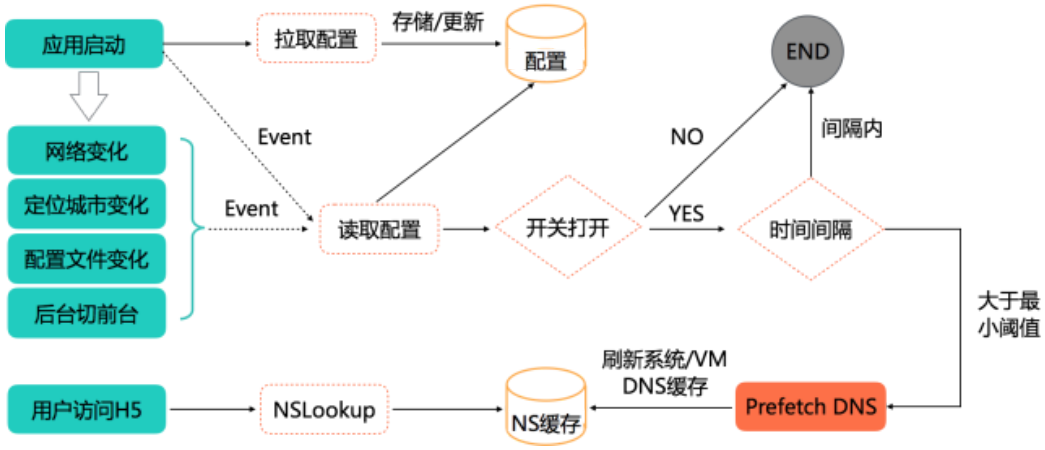
由于DNS的种种问题，腾讯推出了HttpDNS服务，使用HTTP协议向DNS服务器的80端口进行请求，代替传统的DNS协议向DNS服务器的53端口进行请求，绕开Local DNS，避免网络劫持和跨网访问等问题。但HttpDNS需要能够获取CDN边缘节点的IP，这就限制了只有像腾讯、阿里这种有自建CDN的大厂才能实现。

另外W3C也提供了DNS预读的方案，可以通过在服务器端发送 X-DNS-Prefetch-Control 报头，或是在文档中使用值为 http-equiv 的 <meta> 标签：

<meta http-equiv="x-dns-prefetch-control" content="on"> 的方式来打开浏览器的DNS预读取功能，但是该API功能目前在移动端浏览器内核中实现支持的较少。

我们采取的是一种轻量级的方案，如下：

1. 利用App启动时的config接口，下发DNS Prefetch的配置参数：开关、时间间隔、需要进行prefetch的域名列表等。
2. 监听App启动、网络变化、定位城市变化、配置文件变化、前后台切换等事件，在独立的线程中执行DNS Prefetch的逻辑。
3. 如果开关打开，且上次Prefetch的时间距离当前的时间大于阈值，则刷新DNS，触发操作系统/VM层的缓存功能。



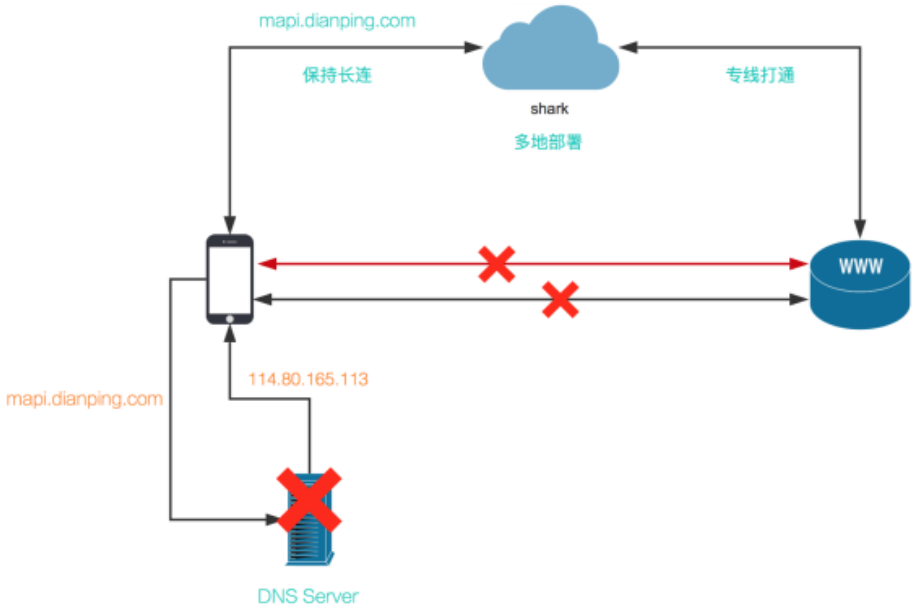
DNS Prefetch上线后境外域名解析时间RT90从350ms下降至250ms左右。

长连接

HTTP请求重度依赖DNS，DNS劫持、移动端网络不稳定使建连失败，以及公网链路质量差等因素，导致移动端的网络成功率一直不高。HTTP 2.0可以通过SSE、WebSocket等方式与服务端保持长连接，并且可以做到请求多路复用，但HTTP 2.0对运维、前端、后端的改造成本非常高。基于此背景美团自研了Shark服务。一种“代理长连接”的模式，主要用于解决移动设备网络通信质量差的问题。

- Shark在国内和境外部署了多个接入点，类似于CDN的就近访问，用户可以就近连接到Shark节点。
- Shark各节点的IP会在App启动时加载到设备，客户端通过“跑马测试”（ping各节点）的方式选择最优节点，建立并保持TCP长连接。
- Shark节点和IDC之间通过专线连接，从而保证了网络链路的质量。
- 客户端Shark的网络层会拦截App内的HTTP请求，通过特定的协议格式将HTTP请求信息转化成TCP包传到Shark节点，Shark节点再将TCP请求“还原”成HTTP，再通过专线请求后端服务。
- 当Shark通道出问题的时候，可以failover到普通的HTTP模式，从而实现高可用。

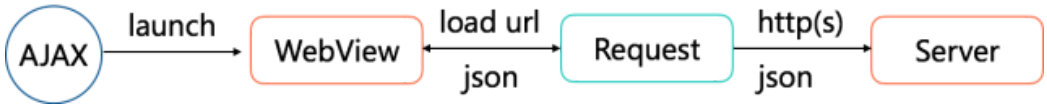
这种“代理长连接”的模式，对后端业务是无感知的，业务无需做任何改造。另外也巧妙的绕开了DNS、公网质量差等问题，极大的提升了Native API请求的网络成功率。



关于Shark的详情，可以参考之前发表的文章《美团移动网络优化实践》。

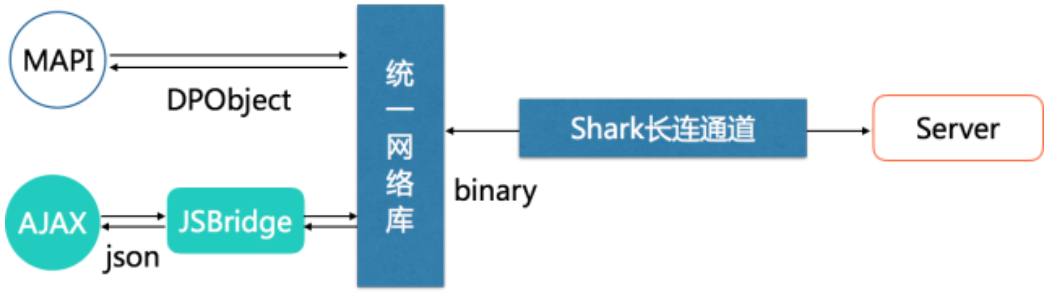
Ajax接长连

目前美团大部分的App都接入了Shark SDK基础网络库，Native API（我们内部叫Mobile API，MAPI）的网路请求由Shark SDK统一解决，使用的是自定义的序列化方式（内部称DPObject，比JSON效率高）。但对于H5页面中的Ajax请求，是没法直接享受到Shark带来的“福利”的。先看一下Hybrid模式下一次Ajax请求的过程：



上图可以看出，一次普通的Ajax请求会由WebView的内置浏览器内核来发送接受请求，一般是JSON格式的数据，和PC浏览器的一次HTTP请求过程差别不大，都是要经过DNS、TCP建连以及要面对公网链路差等问题，另外Ajax请求没法复用TCP连接，意味着每次请求都要重新建连。有了MAPI的经验，我们很容易想到，能否像MAPI一样利用长连通道来提升性能呢?

一种方式是在WebView中拦截页面的HTTP请求，在容器层做请求代理并处理序列化反序列化等事情。这种方式对业务比较友好，业务方几乎不需要做什么事情。但WKWebView的限制比较多，所以该方案目前很难推行。另一种方式是通过JsBridge来实现，缺点是对业务侵入性较高，业务方需要手动控制桥API的调用，一期我们选择的是“较笨拙”的方案二。

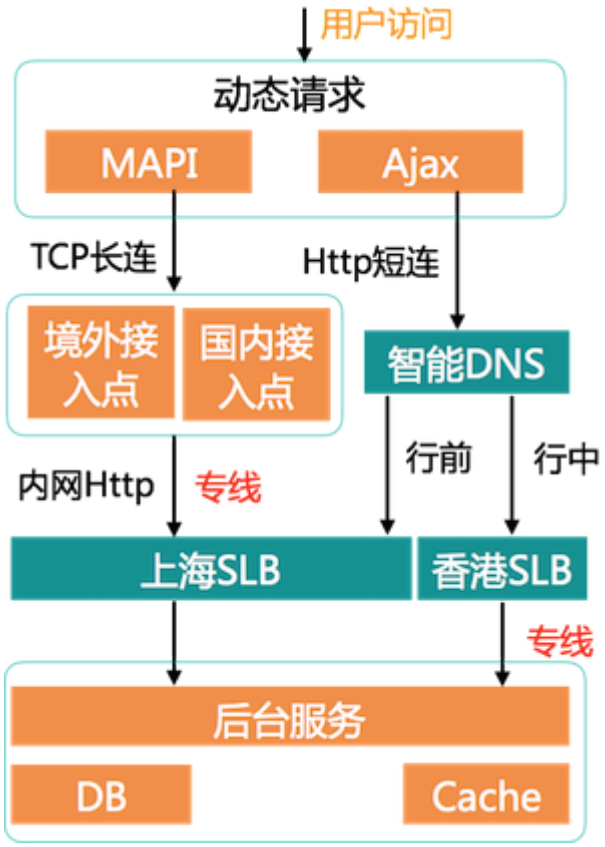


网络链路优化

用户境外访问

Ajax接长连解决了Hybrid模式下App内的H5场景，而对于App外的入口场景，如M站（针对手机的网站）、微信朋友圈分享等，我们没法使用到Shark长连接，跨网跨国访问的问题依然存在。这种情况下我们的优化主要是“利用专线”：

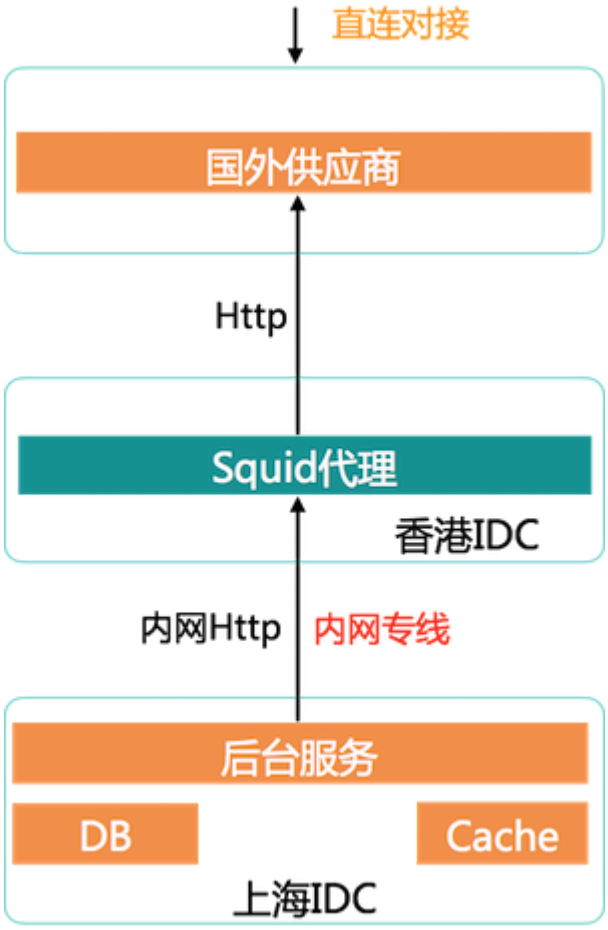
- 我们的后台应用和数据部署在上海机房，在香港机房（香港IDC是个租用机房，未部署数据和应用，非“异地多活”）内部署了一组SLB（反向代理和七层负载均衡，基于tenginx实现）。
- 利用专线连接上海和香港的机房，解决了GFW拦截过滤、跨境网络访问及公网链路差的问题。
- 当用户在境外访问时，智能DNS会解析出香港机房的IP，请求经香港SLB走专线转发到境内服务器；而当用户在境内访问时，则直接请求到上海的机房。



境外直连对接

另一个场景是，我们和很多的境外供应商有直连对接，通过HttpClient的方式后端发起调用对方的Open API接口，这种场景优化前接口延迟及网络成功率都非常不理想（很多和国外对接的业务应该都遇到过类似的问题），我们的优化方案是：

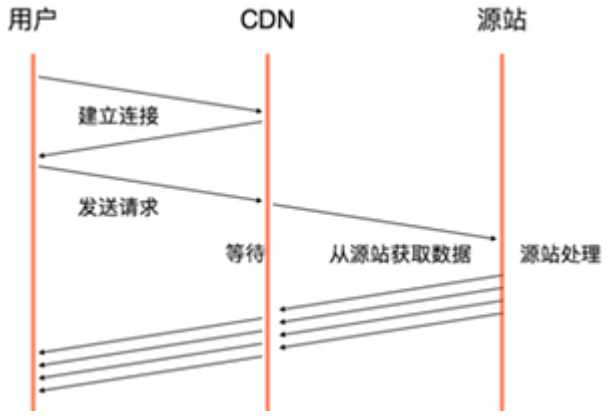
在香港部署一个正向代理Squid，请求先由内网专线转发到香港的Squid服务器，再从香港机房的网络出口出去。以与香港迪士尼的对接为例，优化前的API接口RT95：9s+（迪士尼接口传输的数据非常多），优化后降到2.3s，效果非常明显。



CDN动态加速

除了专线方案，我们还测试CDN动态加速。

CDN不仅可以用来对静态资源做缓存加速，也可以对动态数据接口起加速作用，原理如下：



用户请求耗时=用户和边缘交互的时间 + 1*RTT（边缘到源站）+ 源站处理时间

HTTP请求中，建连是个非常耗时的事，普通HTTP请求TCP 3次握手要消耗2个RTT时间，如果用户和服务器位置很远，比方说用户在美国，服务在中国，请求走海底光缆的话一次RTT理论值是150ms左右（光波信号传输速度结合物理距离计算得出），实际肯定大于理论值。

CDN动态加速主要在以下几方面起到优化效果：

1. 用户与服务器的建连改成与CDN边缘节点建连（就近访问），缩短了建连时间，同时也提升了建连成功率。
2. CDN与源站之间通信相比公网网络链路质量有保证。
3. CDN节点和源站的连接可复用。

我们实测下来CDN动态加速在部分国家和地区有明显的加速效果，但整体的效果不够明显，所以最终未投入规模使用。

前端优化

前端优化我们主要做了下面几件事情：

- 前后端分离
- 图片优化
- 域名收敛、减少请求
- 离线化
- 首屏Node后端同构渲染

前后端分离

在之前的项目中，页面是“Java直出”的方式，由Java后端项目中通过FTL模板引擎拼装，前端团队会维护另外一个前端的项目，存放相应的CSS和JS文件，最后通过公司内部的Cortex系统打包发布。

这个流程的问题在于前端对于整个页面入口没有控制力，需要依赖后端的FTL拼装，页面的内容需要更改时，前后端同学就要反复沟通协调，整体效率比较差，容易出错，也不方便实现前端相关的优化，前端做模块组件化的成本较高。

前后端分离的关键点在于前端拥有完整独立的开发、测试、部署的流程，与后端完全分离。我们把页面的组装完全放置到了前端项目，后端只提供Ajax的接口用于获取和提交数据。前端页面完全静态化，构建完毕之后连同相应的静态资源通过CI直接发布到CDN。这样的好处有：

1. 前后端同学的开发工作解耦，只需要约定好API，两边即可并行开发。
2. 后端API可以做到多端复用，比如PC、H5、M站、小程序等。
3. 前端主文档HTML页面可以利用CDN加速。

前后端分离架构有诸多的优点，但有一个坑需要注意：SEO的问题，无法提供给搜索引擎可收录的页面，因为主文档HTML基本为空页面，需要搜索引擎蜘蛛拥有执行JavaScript的能力才行，现实是大部分的搜索引擎都不支持。所以对于一些需要搜索引擎引流的页面不推荐用前后端分离。

图片优化

在一些重体验的网页上，图片资源的占比通常较大，一些高清大图动则几十上百K大小。 针对图片这块我们主要做了以下几点优化：

- 图片尺寸按屏幕大小自适应。原先我们图片的尺寸都是由后端控制，由服务在代码中写死下发给前端，这样带来问题是：
    1. 不同设备不同大小的屏幕上使用的图片尺寸一样，无法满足所有设备。

2. 一些没经验的开发同学经常乱用尺寸，比如我们POI列表页的商户头图只需要200 * 200就够了。而新手工程师可能使用的是800 * 800的图片，导致页面加载慢、用户流量白白被浪费且客户端还需要做图片压缩剪裁。

   美团云的图片服务提供了实时剪裁功能，后端在下发图片URL时不需要指定尺寸，由客户端根据屏幕尺寸做自适应计算，这样可保证每台设备上的图片都“刚好合适”。

- CDN加速：前面CDN优化章节已介绍，通过接入境外CDN服务商及CDN预热的方式做CDN加速。
- 图片压缩：境外业务内部已在全面使用WebP，经测试WebP格式能够优化图片大小25%–50%，清晰度基本没有影响。
- 懒加载：图片资源通常比较大，选用懒加载可有效缩短页面可交互时间。

域名收敛、减少请求

域名和请求数过多会带来以下问题：

1. DNS解析成本高。
2. CDN加速一般都是按域名来做配置，过多的域名无形增加了CDN接入的成本。
3. 浏览器的并发加载数限制，浏览器对单域名的并发度是有限的，超过限制的请求需要等待串行加载，页面加载速度会变慢。

我们做了以下几件事：

1. 去掉一些直接引入的第三方脚本，将脚本直接打包到我们的代码中（可以利用CDN）。
2. 所有静态资源共用一个域名。
3. 将一些尺寸较小的脚本和CSS构建过程中内联到主文档中。

通过域名收敛/减少请求数，我们商品详情页的页面请求数从8个减少到4个、域名数也减少一半，页面完全加载时间下降了约1000ms。

离线化

离线化可以减少网络请求，加速页面加载速度。另外当用户网络不稳定或断网时也可以访问已被缓存的页面和资源，我们先后使用了2种离线化方案：

1. AppCache（HTML Application Cache API），在前端项目构建流程中，通过分析页面资源依赖关系，自动生成资源manifest文件，这样就能够确保页面及资源发生变更时，manifest文件内容同步更新。当浏览器监测到manifest文件有更新时，会自动重新下载manifest里面的文件。AppCache的一个缺点是缓存文件会越来越多，缓存不容易清理。AppCache未来会逐步被Service Worker所取代，无论从灵活性还是可扩展性而言，SW都更胜一筹。
2. 目前在使用的是公司平台自研的离线包框架，相比于AppCache，离线包框架在资源更新，离线配置，内存管理等方面都做了很大的改善。另外AppCache对于用户第一次加载页面是没有加速效果的，因为只有第一次访问之后才会把资源缓存下来。而离线包框架则可以做到真正的预加载，它会监听APP正常启动事件，当APP启动后即可开始加载更新离线资源。

Node服务端同构渲染

前面介绍了前后端分离的架构，HTML主文档可以利用CDN加速，另外前后端同学很好的解耦开了，前端可以更方便的做组件化沉淀。但这种架构除了SEO时的问题还有另外一个问题，先看一下前后端分离下的一个页面加载过程：

当遇到页面引入的外部依赖很多时，这种架构性能可能还不如”Java直出”：

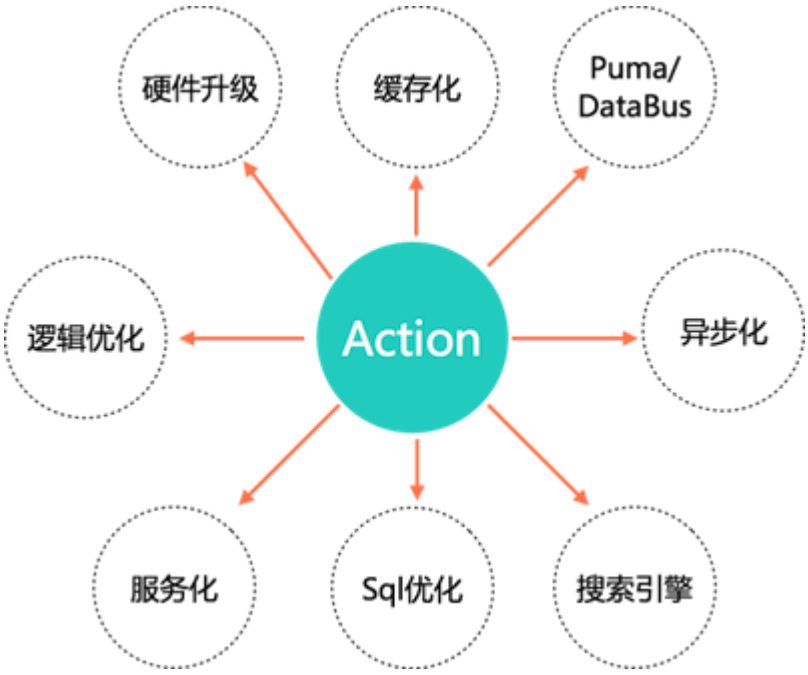
同构渲染，结合了Java直出和前后端分离的优势：

1. 通过在Java API之前加入Node层，Node由前端同学来维护，这样仍然可以做到前后端解耦，后端同学只需要写Ajax返回JSON，甚至只需要通过RPC接口。
2. 缩短并优化了前端串行请求的链路，首屏数据可以更早的展示。
3. 相比于服务端，移动端设备的性能较弱，页面在服务端渲染比在前端渲染会快很多。配合上Ajax长连等网络优化技术，Node同构首屏后端渲染提升了首屏加载速度。

Node同构和一开始的Java freemark后端渲染Java直出的方式对比，最大的区别在于：Node项目可由前端同学来维护，用的是前端工程师熟悉的JS语言。另外前端生态较好，React、Vue等框架都提供了丰富的渲染模板供前端工程师选择。

后端优化

后端优化的思路相对比较比较通用，和境外业务的特点关系性并不大，文中的前言部分“影响性能的因素”章节有简单描述，本文将不对各种后端优化手段做详细介绍，只挑几件我们做过的事情做下简单介绍：



1. 硬件升级

硬件问题对性能的影响不容忽视，早期的时候，我们的一个DB集群经常有慢SQL报警，业务排查下来发现SQL都很简单，该做的索引优化也都做了。后来DBA同学帮忙定位到问题是硬件过旧导致，将机械硬盘升级成固态硬盘之后报警立马消失了，效果立竿见影！

2. 缓存化

缓存可以称的上是性能优化的利器，使用缓存时需要考虑缓存命中率、缓存更新、数据一致性、缓存穿透及雪崩、Value过大等问题，可以通过mutiGet将多次请求合并一次、异步访问等方式来提升缓存读取的性能。

3. 产品逻辑优化

业务逻辑优化经常会容易被忽略，但效果却往往比数据库调优、JVM调优之类的来的更明显。

举个例子，我们的商家系统有一个商家导出历史订单的功能，接口经常超时。排查下来是由于功能上默认是导出商家的全部历史订单，数据量过大导致。而实际业务上几个月之前已完成的订单对商家来说并没有什么意义，如果是要看统计数据我们有专门的统计功能。后来和PM商量后在导出时增加了时间段选择项，默认只会导出最近3个月的订单，超时问题顺利解决。

再比如，12306春运抢火车票的场景，由于访问的人多，用户点击“查票”之后系统会非常卡，进度条非常慢，作为用户，我们会习惯性的再去点“查票”，可能会连续点个好几次。假设平均一个用户点5次，则后端系统负载就增加了5倍！而其中80%的请求是重复请求。这个时候我们可以通过产品逻辑的方式来优化，比如，在用户点击查询之后将“按钮置灰”，或者通过JS控制xx秒只能只能提交一次请求等，有效的拦截了80%的无效流量。

4. 服务化

我们做服务化最基础的是按业务做服务拆分，避免跨业务间的互相影响，数据和服务同时拆分。同一个业务内部我们还按计算密集型/IO密集型的服务拆分、C端/B端服务拆分、核心/非核心服务拆分、高频服务单独部署等原则做拆分。

5. 异步化

异步化可以利用线程池、消息队列等方式实现。使用线程池的时候一定要注意核心参数的设置，可以通过监控工具去观测实际创建、活跃、空闲的线程数，结合CPU、内存的使用率情况来做线程池调优。

另一种是通过NIO实现异步化，一切网络IO皆可异步：RPC框架、Servlet 3.0提供的异步技术、Apache HttpAsyncClient、缓存异步接口等等。

6. 搜索引擎

复杂查询以及一些聚合计算不适合在数据库中做，可以利用搜索引擎来实现，另外搜索引擎还可以帮我们很好的解决跨库、跨数据源检索的场景。

服务架构

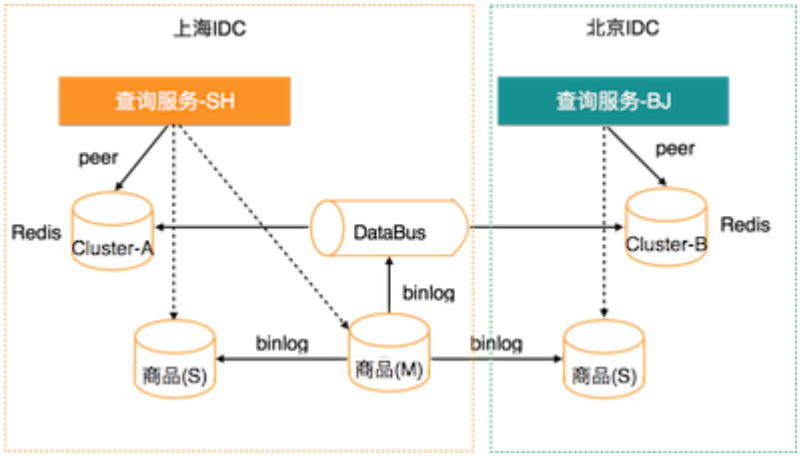
问题背景

美团在上海、北京两地都有机房，以我们的商品服务为例，商品服务会同时被上海和北京的两侧的应用依赖调用。一开始我们的服务只部署在上海机房，由于上海、北京两地机房间地理间隔较远（北京–上海专线ping延迟30ms左右），当北京的应用调服务时就会有较高的延迟，如果一个API内多次跨地域调用则性能会非常差。

为此我们做了如下的缓存双集群加异地模式部署：

1. 商品静态信息全缓存，缓存未命中时再查DB。
2. DB以主从的方式部署，北京机房也部署了一套从库。
3. 缓存数据更新使用Databus，基于binglog数据同步的方式，保障DB和Redis数据的“准实时”同步。
4. 服务、缓存、DB均两地部署，调用方就近访问（RPC、Cache、DAL等基础组件支持就近路由）。
5. 缓存更新利用Client提供的双写模式。当DB数据有变更，Databus监听到数据更新后，消费程序通过缓存Client提供的双写功能往两个缓存集群同时写入，服务层读数据时会选择就近的缓存集群。

通过这种双缓存加异地部署的“异地多活”模式（实际是异地只读），提升了我们服务在跨地域场景下调用时的性能。



总结

结合境外业务特点，本文从网络优化、前端优化、后端优化几个角度介绍了境外业务在性能优化上的一些实践，重点篇幅放在了网络优化部分。性能优化是一个系统性工程，需要前端、后端和SRE一起协作才能做好。得益于公司强大的高性能前端框架、BGP网络、高性能应用组件、云平台等基础设施，以及在靠谱的运维保障SRE团队、基础架构团队以及平台团队支持下，我们境外业务的性能优化取得了阶段性的成果，后续还要继续努力。

作者简介

- 云霜，美团后台开发工程师，美团旅行境外度假业务交易后台组技术负责人，2012年加入原大众点评。

招聘信息

最后，打一波广告，美团旅行境外度假业务正在招人，欢迎志同道合的朋友加入我们（简历传送门：yunshuang.tao#dianping.com），加入美团旅行，一起来做改变世界的事。

美团广告实时索引的设计与实现

作者: 仓魁 李晓晖 刘铮 蔡平

背景

在线广告是互联网行业常见的商业变现方式。从工程角度看，广告索引的结构和实现方式直接决定了整个系统的服务性能。本文以美团的搜索广告系统为蓝本，与读者一起探讨广告系统的工程奥秘。

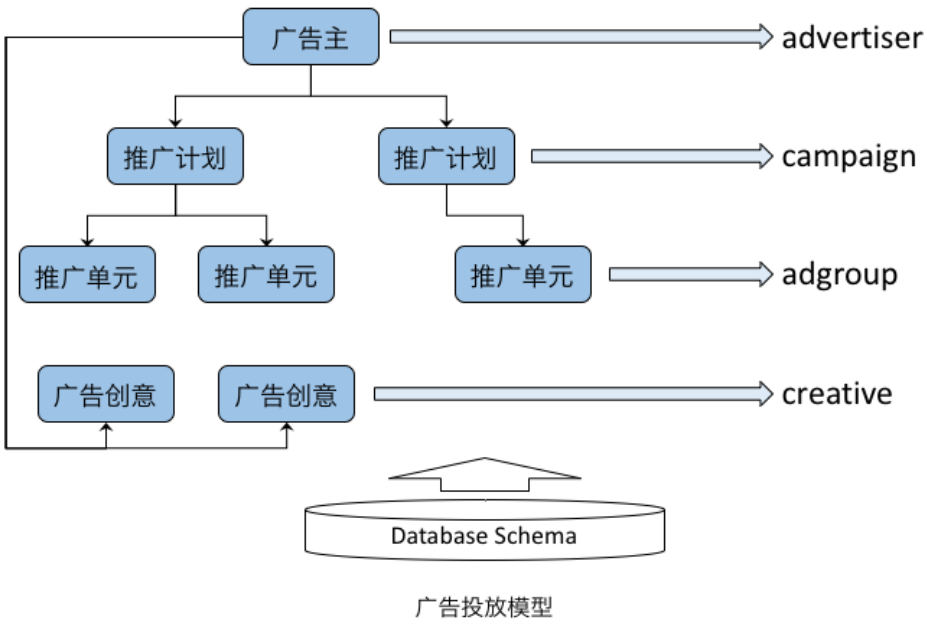
领域问题

广告索引需具备以下基本特性：

1. 层次化的索引结构。
2. 实时化的索引更新。

层次投放模型

一般地，广告系统可抽象为如下投放模型，并实现检索、过滤等处理逻辑。



广告投放模型

广告投放模型

该层次结构的上下层之间是一对多的关系。一个广告主通常创建若干个推广计划，每个计划对应一个较长周期的KPI，比如一个月的预算和投放地域。一个推广计划中的多个推广单元分别用于更精细的投放控

制，比如一次点击的最高出价、每日预算、定向条件等。广告创意是广告曝光使用的素材，根据业务特点，它可以从属于广告主或推广计划层级。

实时更新机制

层次结构可以更准确、更及时地反应广告主的投放控制需求。投放模型的每一层都会定义若干字段，用于实现各类投放控制。广告系统的大部分字段需要支持实时更新，比如审核状态、预算上下线状态等。例如，当一个推广单元由可投放状态变为暂停状态时，若该变更没有在索引中及时生效，就会造成大量的无效投放。

业界调研

目前，生产化的开源索引系统大部分为通用搜索引擎设计，基本无法同时满足上述条件。

- Apache Lucene
  - 全文检索、支持动态脚本；实现为一个Library。
  - 支持实时索引，但不支持层次结构。
- Sphinx
  - 全文检索；实现为一个完整的Binary，二次开发难度大。
  - 支持实时索引，但不支持层次结构。

因此，广告业界要么基于开源方案进行定制，要么从头开发自己的闭源系统。在经过再三考虑成本收益后，我们决定自行设计广告系统的索引系统。

索引设计

工程实践重点关注稳定性、扩展性、高性能等指标。

设计分解

设计阶段可分解为以下子需求。

实时索引

广告场景的更新流，涉及索引字段和各类属性的实时更新。特别是与上下线状态相关的属性字段，需要在若干毫秒内完成更新，对实时性有较高要求。

用于召回条件的索引字段，其更新可以滞后一些，如在几秒钟之内完成更新。采用分而治之的策略，可极大降低系统复杂度。

- 属性字段的更新：直接修改正排表的字段值，可以保证毫秒级完成。
- 索引字段的更新：涉及更新流实时计算、倒排索引等的处理过程，只需保证秒级完成。

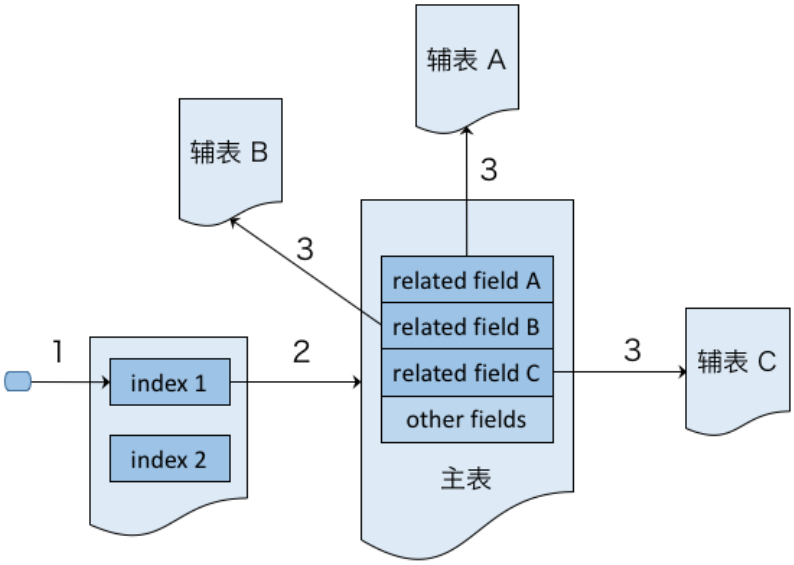
此外，通过定期切换全量索引并追加增量索引，由索引快照确保数据的正确性。

层次结构

投放模型的主要实体是广告主（Advertiser）、推广计划（Campaign）、广告组（Adgroup）、创意（Creative）等。其中：

- 广告主和推广计划：定义用于控制广告投放的各类状态字段。
- 广告组：描述广告相关属性，例如竞价关键词、最高出价等。
- 创意：与广告的呈现、点击等相关的字段，如标题、创意地址、点击地址等。

一般地，广告检索、排序等均基于广告组粒度，广告的倒排索引也是建立在广告组层面。借鉴关系数据库的概念，可以把广告组作为正排主表（即一个Adgroup是一个doc），并对其建立倒排索引；把广告主、推广计划等作为辅表。主表与辅表之间通过外键关联。



广告检索流程

广告检索流程

1. 通过查询条件，从倒排索引中查找相关docID列表。
2. 对每个docID，可从主表获取相关字段信息。
3. 使用外键字段，分别获取对应辅表的字段信息。

检索流程中实现对各类字段值的同步过滤。

可靠高效

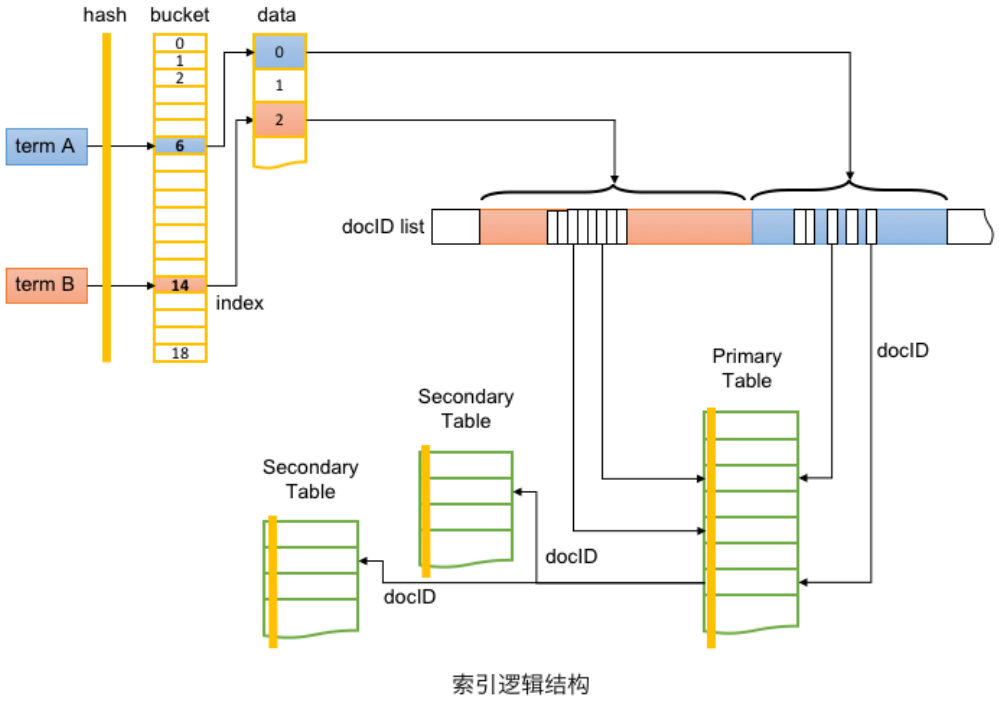
广告索引结构相对稳定且与具体业务场景耦合较弱，为避免Java虚拟机由于动态内存管理和垃圾回收机制带来的性能抖动，最终采用C++11作为开发语言。虽然Java可使用堆外内存，但是堆外堆内的数据拷贝对高并发访问仍是较大开销。项目严格遵循《Google C++ Style》，大幅降低了编程门槛。

在“读多写少”的业务场景，需要优先保证“读”的性能。检索是内存查找过程，属于计算密集型服务，为保证CPU的高并发，一般设计为无锁结构。可采用“一写多读”和延迟删除等技术，确保系统高效稳定运转。此外，巧妙利用数组结构，也进一步优化了读取性能。

灵活扩展

正排表、主辅表间的关系等是相对稳定的，而表内的字段类型需要支持扩展，比如用户自定义数据类型。甚至，倒排表类型也需要支持扩展，例如地理位置索引、关键词索引、携带负载信息的倒排索引等。通过继承接口，实现更多的定制化功能。

逻辑结构



索引逻辑结构

广告检索流程

从功能角度，索引由Table和Index两部分组成。如上图所示，Index实现由Term到主表docID的转换；Table实现正排数据的存储，并通过docID实现主表与辅表的关联。

分层架构

索引库分为三层：

1. 接口层：以API方式对外提供索引的构建、更新、检索、过滤等功能。
2. 能力层：实现基于倒排表和正排表的索引功能，是系统的核心。
3. 存储层：索引数据的内存布局和到文件的持久化存储。

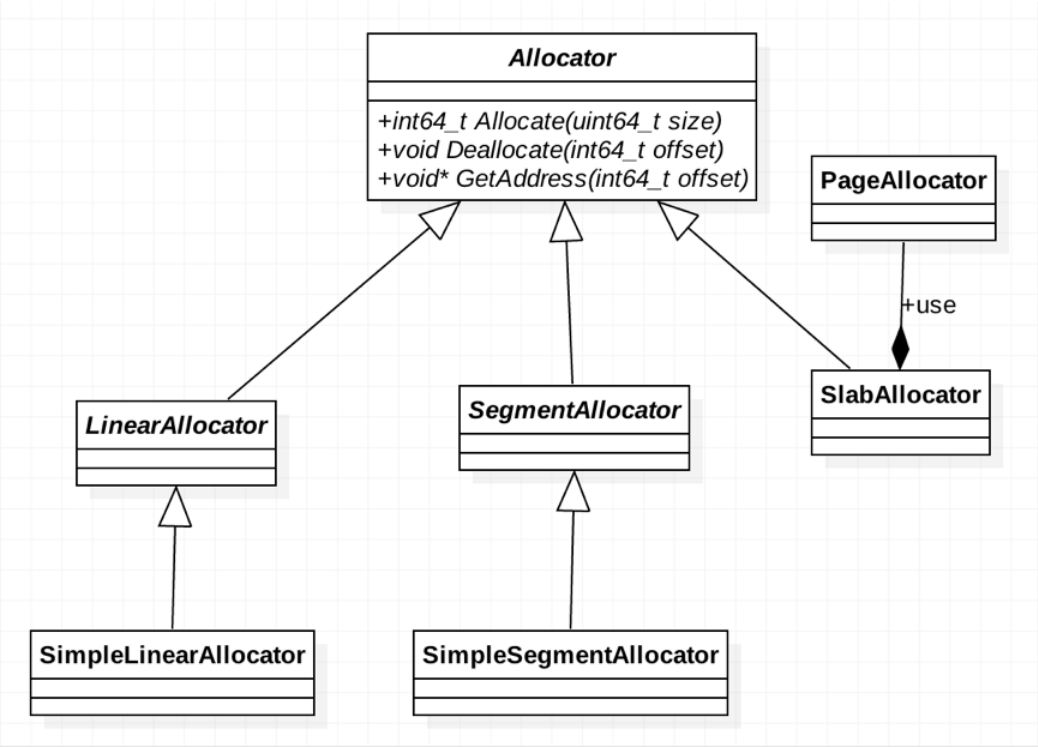
索引实现

本节将自底向上，从存储层开始，逐一描述各层的设计细节和挑战点。

存储层

存储层负责内存分配以及数据的持久化，可使用mmap实现到虚拟内存空间的映射，由操作系统实现内存与文件的同步。此外，mmap也便于外部工具访问或校验数据的正确性。

将存储层抽象为分配器（Allocator）。针对不同的内存使用场景，如对内存连续性的要求、内存是否需要回收等，可定制实现不同的分配器。



内存分配器

以下均为基于mmap的各类分配器，这里的“内存”是指调用进程的虚拟地址空间。实际的代码逻辑还涉及复杂的Metadata管理，下文并未提及。

简单的分配策略

- LinearAllocator
  - 分配连续地址空间的内存，即一整块大内存；当空间需要扩展时，会采用新的mmap文件映射，并延迟卸载旧的文件映射。
  - 新映射会导致页表重新装载，大块内存映射会导致由物理内存装载带来的性能抖动。
  - 一般用于空间需求相对固定的场景，如HashMap的bucket数组。
- SegmentAllocator
  - 为解决LinearAllocator在扩展时的性能抖动问题，可将内存区分段存储，即每次扩展只涉及一段，保证性能稳定。
  - 分段导致内存空间不连续，但一般应用场景，如倒排索引的存储，很适合此法。

- 默认的段大小为64MB。

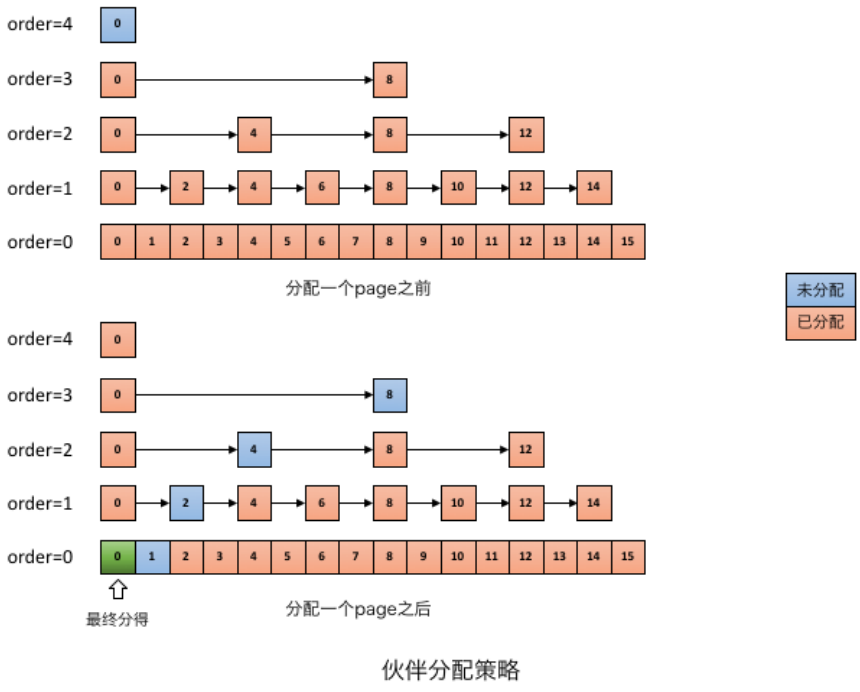
集约的分配策略

频繁的增加、删除、修改等数据操作将导致大量的外部碎片。采用压缩操作，可以使占用的内存更紧凑，但带来的对象移动成本却很难在性能和复杂度之间找到平衡点。在工程实践中，借鉴Linux物理内存的分配策略，自主实现了更适于业务场景的多个分配器。

- PageAllocator
  - 页的大小为4KB，使用伙伴系统（Buddy System）的思想实现页的分配和回收。
  - 页的分配基于SegmentAllocator，即先分段再分页。

在此简要阐述伙伴分配器的处理过程，为有效管理空闲块，每一级order持有一个空闲块的FreeList。设定最大级别order=4，即从order=0开始，由低到高，每级order块内页数分别为1、2、4、8、16等。分配时先找满足条件的最小块；若找不到则在上一级查找更大的块，并将该块分为两个“伙伴”，其中一个分配使用，另一个置于低一级的FreeList。

下图呈现了分配一个页大小的内存块前后的状态变化，分配前，分配器由order=0开始查找FreeList，直到order=4才找到空闲块。



伙伴分配器策略

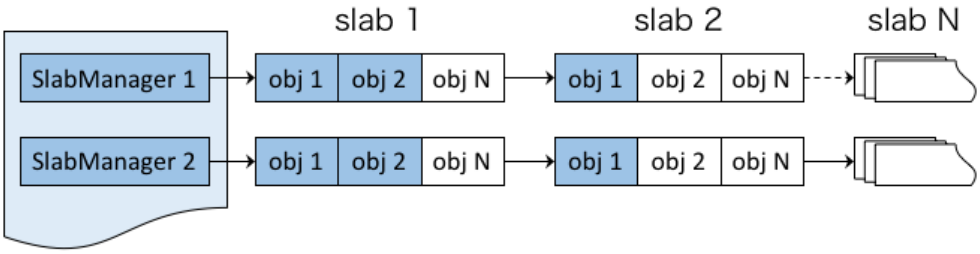
将该空闲块分为页数为8的2个伙伴，使用前一半，并将后一半挂载到order=3的FreeList；逐级重复此过程，直到返回所需的内存块，并将页数为1的空闲块挂在到order=0的FreeList。

当块释放时，会及时查看其伙伴是否空闲，并尽可能将两个空闲伙伴合并为更大的空闲块。这是分配过程的逆过程，不再赘述。

虽然PageAllocator有效地避免了外部碎片，却无法解决内部碎片的问题。为解决这类小对象的分配问题，实现了对象缓存分配器（SlabAllocator）。

- SlabAllocator
  - 基于PageAllocator分配对象缓存，slab大小以页为单位。
  - 空闲对象按内存大小定义为多个SlabManager，每个SlabManager持有一个PartialFreeList，用于放置含有空闲对象的slab。

对象的内存分配过程，即从对应的PartialFreeList获取含有空闲对象的slab，并从该slab分配对象。反之，释放过程为分配的逆过程。



对象缓存分配器

对象缓存分配器

综上，实时索引存储结合了PageAllocator和SlabAllocator，有效地解决了内存管理的外部碎片和内部碎片问题，可确保系统高效稳定地长期运行。

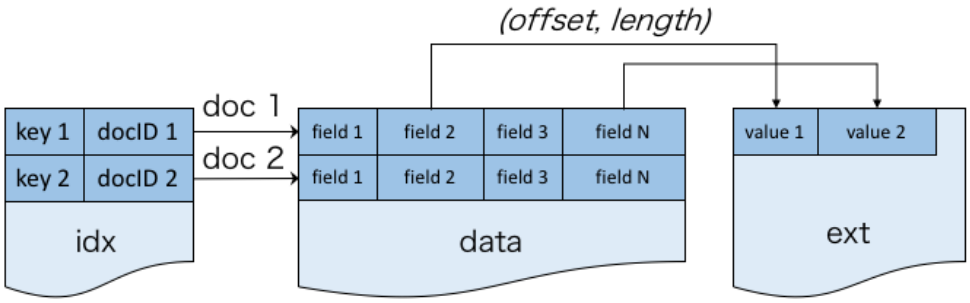
能力层

能力层实现了正排表、倒排表等基础的存储能力，并支持索引能力的灵活扩展。

正向索引

也称为正排索引（Forward Index），即通过主键（Key）检索到文档（Doc）内容，以下简称正排表或Table。不同于搜索引擎的正排表数据结构，Table也可以单独用于NoSQL场景，类似于Kyoto Cabinet的哈希表。

Table不仅提供按主键的增加、删除、修改、查询等操作，也配合倒排表实现检索、过滤、读取等功能。作为核心数据结构，Table必须支持频繁的字段读取和各类型的正排过滤，需要高效和紧凑的实现。



正排存储结构

正排存储结构

为支持按docID的随机访问，把Table设计为一个大数组结构（data区）。每个doc是数组的一个元素且长度固定。变长字段存储在扩展区（ext区），仅在doc中存储其在扩展区的偏移量和长度。与大部分搜索引擎的列存储不同，将data区按行存储，这样可针对业务场景，尽可能利用CPU与内存之间的缓存来提高访问效率。

此外，针对NoSQL场景，可通过HashMap实现主键到docID的映射（idx文件），这样就可支持主键到文档的随机访问。由于倒排索引的docID列表可以直接访问正排表，因此倒排检索并不会使用该idx。

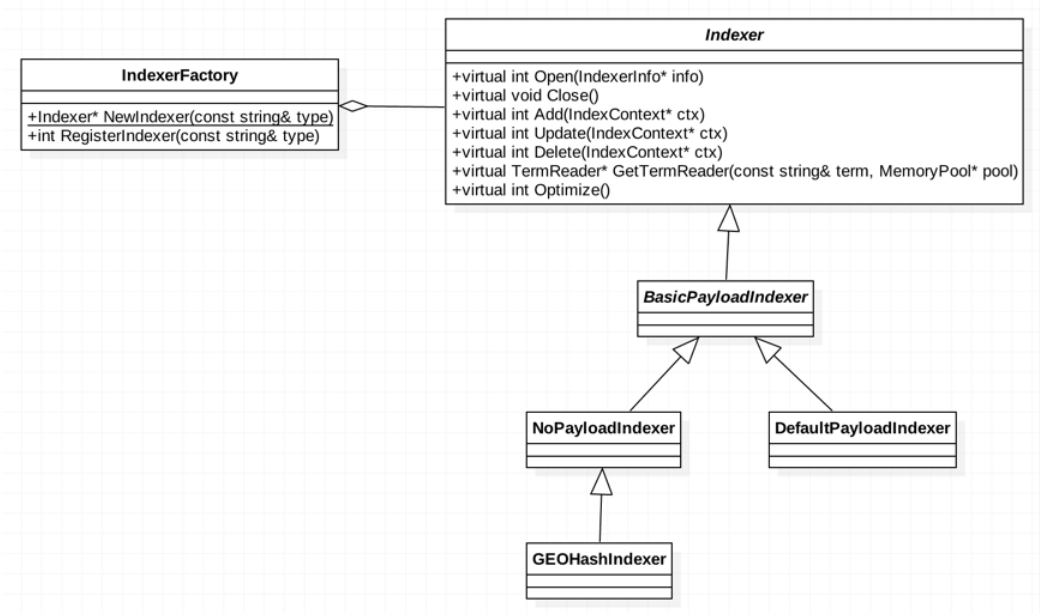
反向索引

也称作倒排索引（Inverted Index），即通过关键词（Keyword）检索到文档内容。为支持复杂的业务场景，如遍历索引表时的算法粗排逻辑，在此抽象了索引器接口Indexer。

```
class Indexer {
 public:
  virtual ~Indexer();

  virtual int Open(IndexerInfo* info);
  virtual int Close();

  virtual int Add(IndexContext* ctx) = 0;
  virtual int Update(IndexContext* ctx) = 0;
  virtual int Delete(IndexContext* ctx) = 0;

  virtual TermReader* GetTermReader(const std::string& term, MemoryPool* pool) = 0;
  virtual ExecutorFactory* GetExecutorFactory(const Query& query, MemoryPool* pool) = 0;

  virtual int Optimize() = 0;
};
```

索引器接口定义

具体的Indexer仅需实现各接口方法，并将该类型注册到IndexerFactory，可通过工厂的NewIndexer方法获取Indexer实例，类图如下：



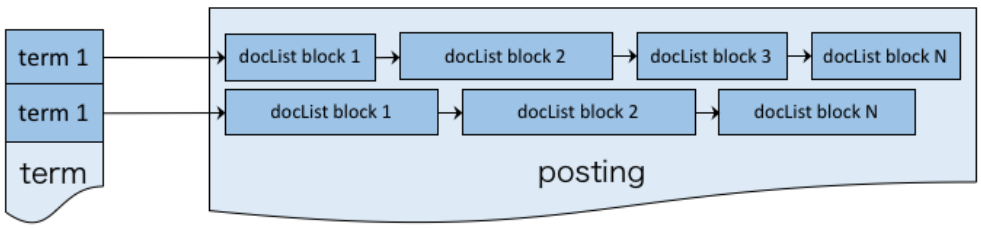
索引器接口类图

当前实现了三种常用的索引器：

- NoPayloadIndexer：最简单的倒排索引，倒排表为单纯的docID列表。
- DefaultPayloadIndexer：除docID外，倒排表还存储keyword在每个doc的负载信息。针对业务场景，可存储POI在每个Node粒度的静态质量分或最高出价。这样在访问正排表之前，就可完成一定的倒排优选过滤。
- GEOHashIndexer：即基于地理位置的Hash索引。

上述索引器的设计思路类似，仅阐述其共性的两个特征：

- 词典文件term：存储关键词、签名哈希、posting文件的偏移量和长度等。与Lucene采用的前缀压缩的树结构不同，在此实现为哈希表，虽然空间有所浪费，但可保证稳定的访问性能。
- 倒排表文件posting：存储docID列表、Payload等信息。检索操作是顺序扫描倒排列表，并在扫描过程中做一些基于Payload的过滤或倒排链间的布尔运算，如何充分利用高速缓存实现高性能的索引读取是设计和实现需要考虑的重要因素。在此基于segmentAllocator实现分段的内存分配，达到了效率和复杂度之间的微妙平衡。



倒排存储结构

倒排存储结构

出于业务考虑，没有采用Lucene的Skip list结构，因为广告场景的doc数量没有搜索引擎多，且通常为单个倒排列表的操作。此外，若后续doc数量增长过快且索引变更频繁，可考虑对倒排列表的元素构建B+树

结构，实现倒排元素的快速定位和修改。

接口层

接口层通过API与外界交互，并屏蔽内部的处理细节，其核心功能是提供检索和更新服务。

配置文件

配置文件用于描述整套索引的Schema，包括Metadata、Table、Index的定义，格式和内容如下：

```
<database>
  <global>
    <data path="/full/path/to/data/directory/">
  </global>

  <table name="adgroup" key="adgroupID" type="primary">
    <meta>
      <doc_size value="4000" />
      <extend_doc_size value="40" />
      <ext_file_extend_size_byte value="4096" />
      <ext_file_init_max_size_byte value="8192" />
    </meta>

    <attr>
      <field name="adgroupID" type="UINT64" num="1" />
      <field name="adgroupID" type="HASH32" num="1" />
      <field name="adgroupID" type="HASH64" num="1" />
      <field name="campaignID" type="UINT64" related="campaign" />
      <field name="maxPrice" type="UINT32" num="1" />
    </attr>

    <index name="catID" type="NoPayloadIndexer" />

    <index name="keywords" type="DefaultPayloadIndexer">
      <payload>
        <field name="price" type="UINT32" />
        <field name="score" type="UINT32" />
      </payload>
    </index>
  </table>

  <table name="campaign" key="campaignID" type="secondary">
    <attr>
      <field name="campaignID" type="UINT64" num="1" />
      <field name="campaignStatus" type="UINT8" num="1" />
    </attr>
  </table>
</database>
```

索引配置文件

可见，Index是构建在Table中的，但不是必选项；Table中各个字段的定义是Schema的核心。当Schema变化时，如增加字段、增加索引等，需要重新构建索引。篇幅有限，此处不展开定义的细节。

检索接口

检索由查找和过滤组成，前者产出查找到的docID集合，后者逐个对doc做各类基础过滤和业务过滤。

```
int Search(const std::string& query, SearchContext* search_context, ResultSet** rs);

int DoSearch(const Query& query, SearchContext* search_context, ResultSet** rs);
int DoFilter(const Query& query, SearchContext* search_context, ResultSet** rs);
```

检索接口定义

- Search：返回正排过滤后的ResultSet，内部组合了对DoSearch和DoFilter的调用。
- DoSearch：查询doc，返回原始的ResultSet，但并未对结果进行正排过滤。
- DoFilter：对DoSearch返回的ResultSet做正排过滤。

一般仅需调用Search就可实现全部功能；DoSearch和DoFilter可用于实现更复杂的业务逻辑。

以下为检索的语法描述：

```
/{table}/{indexer|keyfield}?query=xxxxxx&filter=xxxxx
```

第一部分为路径，用于指定表和索引。第二部分为参数，多个参数由&分隔，与URI参数格式一致，支持query、filter、Payload_filter、index_filter等。

由query参数定义对倒排索引的检索规则。目前仅支持单类型索引的检索，可通过index_filter实现组合索引的检索。可支持AND、OR、NOT等布尔运算，如下所示：

```
query=(A&B|C|D)!E
```

查询语法树基于Bison生成代码。针对业务场景常用的多个term求docID并集操作，通过修改Bison文法规则，消除了用于存储相邻两个term的doc合并结果的临时存储，直接将前一个term的doc并入当前结果集。该优化极大地减少了临时对象开销。

由filter参数定义各类正排表字段值过滤，多个键值对由“;”分割，支持单值字段的关系运算和多值字段的集合运算。

由Payload_filter参数定义Payload索引的过滤，目前仅支持单值字段的关系运算，多个键值对由“;”分割。

详细的过滤语法如下：

FILTER 语法

运算符	描述	示例
FIELD EQ NUMBER	等于	@field_name==$value
FIELD NE NUMBER	不等于	@field_name!=$value
FIELD GT NUMBER	大于	@field_name>$value
FIELD GE NUMBER	大于等于	@field_name>=$value
FIELD LT NUMBER	小于	@field_name<$value
FIELD LE NUMBER	小于等于	@field_name<=$value
SET ANY IN FIELD	集合运算any in	(x,y,z) ANY IN @field_name
SET ALL IN FIELD	集合运算all in	(x,y,x) ALL IN @field_name
SET NOT IN FIELD	集合运算not in	(x,y,z) NOT IN @field_name
FIELD ANY IN SET	集合运算any in	@field_name ANY IN (x,y,z)
FIELD ALL IN SET	集合运算all in	@field_name ALL IN (x,y,z)
FIELD NOT IN SET	集合运算not in	@field_name NOT IN (x,y,z)

PAYLOAD FILTER 语法

运算符	描述	示例
FIELD EQ NUMBER	等于	@field_name==$value
FIELD NE NUMBER	不等于	@field_name!=$value
FIELD GT NUMBER	大于	@field_name>$value
FIELD GE NUMBER	大于等于	@field_name>=$value
FIELD LT NUMBER	小于	@field_name<$value
FIELD LE NUMBER	小于等于	@field_name<=$value

过滤语法格式

此外，由index_filter参数定义的索引过滤将直接操作倒排链。由于构造检索数据结构比正排过滤更复杂，此参数仅适用于召回的docList特别长但通过索引过滤的docList很短的场景。

结果集

结果集ResultSet的实现，参考了java.sql.ResultSet接口。通过cursor遍历结果集，采用inline函数频繁调用的开销。

实现为C++模板类，主要接口定义如下：

```
// move cursor to the next doc in the result set
inline bool Next();

template<typename T>
auto GetValue(FieldDescriptor<T> fd, T def_value = T()) -> T {
  auto table_reader = GET_TABLE_READER(fd.id, def_value);
  return table_reader->GetValue(fd);
}

template<typename T>
auto GetMultiValue(FieldDescriptor<T> fd, uint32_t* size) -> T* {
  *size = 0;
  auto table_reader = GET_TABLE_READER(fd.id, nullptr);
  return table_reader->GetMultiValue(fd, size);
}
```

结果集接口定义

- Next：移动cursor到下一个doc，成功返回true，否则返回false。若已经是集合的最后一条记录，则返回false。
- GetValue：读取单值字段的值，字段类型由泛型参数T指定。如果获取失败返回默认值def_value。

- GetMultiValue：读取多值字段的值，返回指向值数组的指针，数组大小由size参数返回。读取失败返回null，size等于0。

更新接口

更新包括对doc的增加、修改、删除等操作。参数类型Document，表示一条doc记录，内容为待更新的doc的字段内容，key为字段名，value为对应的字段值。操作成功返回0，失败返回非0，可通过GetErrorString接口获取错误信息。

```
int Add(const std::string& table_name, const Document& document);
int Update(const std::string& table_name, const Document& document);
int Delete(const std::string& table_name, const Document& document);
```

更新接口定义

- 增加接口Add：将新的doc添加到Table和Index中。
- 修改接口Update：修改已存在的doc内容，涉及Table和Index的变更。
- 删除接口Delete：删除已存在的doc，涉及从Table和Index删除数据。

更新服务对接实时更新流，实现真正的广告实时索引。

更新系统

除以上描述的索引实现机制，生产系统还需要打通在线投放引擎与商家端、预算控制、反作弊等的更新流。

挑战与目标

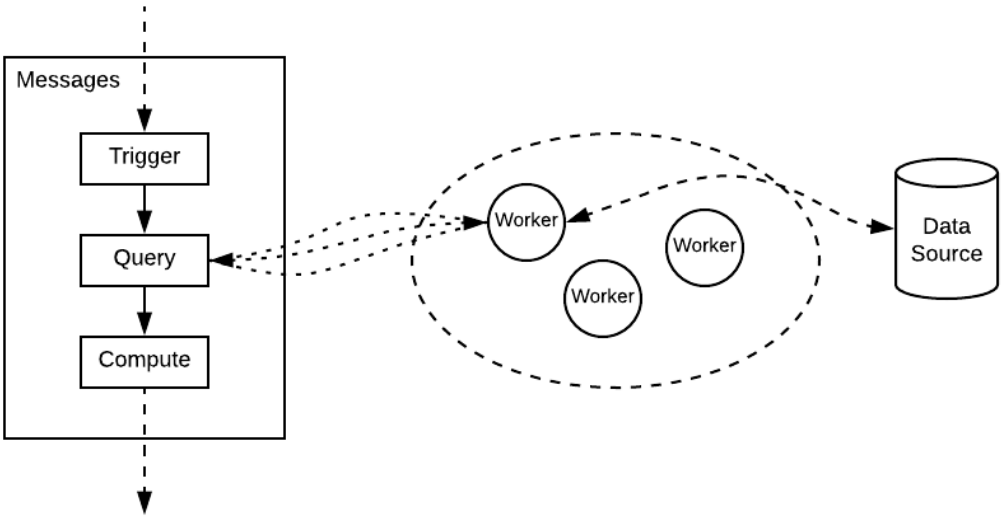
数据更新系统的主要工作是将原始多个维度的信息进行聚合、平铺、计算后，最终输出线上检索引擎需要的维度和内容。

业务场景导致上游触发可能极不规律。为避免更新流出现的抖动，必须对实时更新的吞吐量做优化，留出充足的性能余量来应对触发的尖峰。此外，更新系统涉及多对多的维度转换，保持计算、更新触发等逻辑的可维护性是系统面临的主要挑战。

吞吐设计

虽然更新系统需要大量的计算资源，但由于需要对几十种外部数据源进行查询，因此仍属于IO密集型应用。优化外部数据源访问机制，是吞吐量优化的主要目标。

在此，采取经典的批量化方法，即集群内部，对于可以批量查询的一类数据源，全部收拢到一类特定的worker上来处理。在短时间内，worker聚合数据源并逐次返回给各个需要数据的数据流。处理一种数据源的worker可以有多个，根据同类型的查询汇集到同一个worker批量查询后返回。在这个划分后，就可以做一系列的逻辑优化来提升吞吐量。

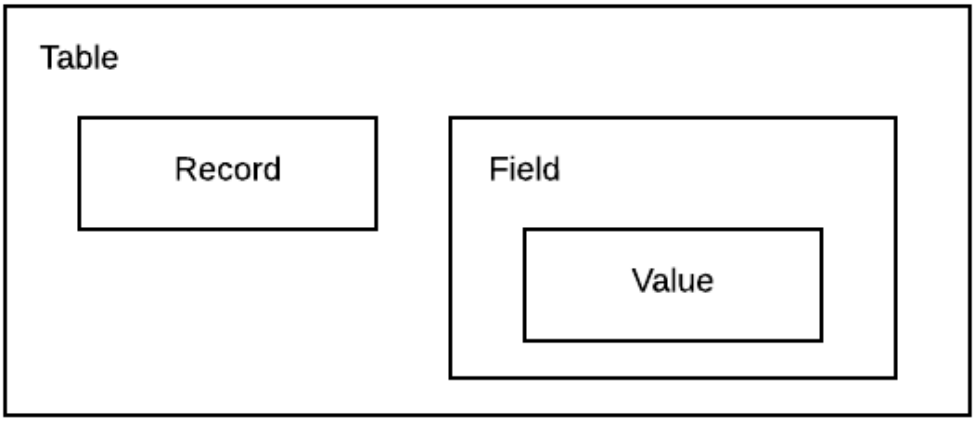


分层抽象

除生成商家端的投放模型数据，更新系统还需处理针对各种业务场景的过滤，以及广告呈现的各类专属信息。业务变更可能涉及多个数据源的逻辑调整，只有简洁清晰的分成抽象，才能应对业务迭代的复杂度。

工程实践中，将外部数据源抽象为统一的Schema，既做到了数据源对业务逻辑透明，也可借助编译器和类型系统来实现完整的校验，将更多问题提前到编译期解决。

将数据定义为表（Table）、记录（Record）、字段（Field）、值（Value）等抽象类型，并将其定义为Scala Path Dependent Type，方便编译器对程序内部的逻辑进行校验。



可复用设计

多对多维度的计算场景中，每个字段的处理函数（DFP）应该尽可能地简单、可复用。例如，每个输出字段（DF）的DFP只描述需要的源数据字段（SF）和该字段计算逻辑，并不描述所需的SF(1)到SF(n)之间的查询或路由关系。

此外，DFP也不与最终输出的层级绑定。层级绑定在定义输出消息包含的字段时完成，即定义消息的时候需要定义这个消息的主键在哪一个层级上，同时绑定一系列的DFP到消息上。

这样，DFP只需单纯地描述字段内容的生成逻辑。如果业务场景需要将同一个DF平铺到不同层级，只要在该层级的输出消息上引用同一个DFP即可。

触发机制

更新系统需要接收数据源的状态变动，判断是否触发更新，并需要更新哪些索引字段、最终生成更新消息。

为实现数据源变动的自动触发机制，需要描述以下信息：

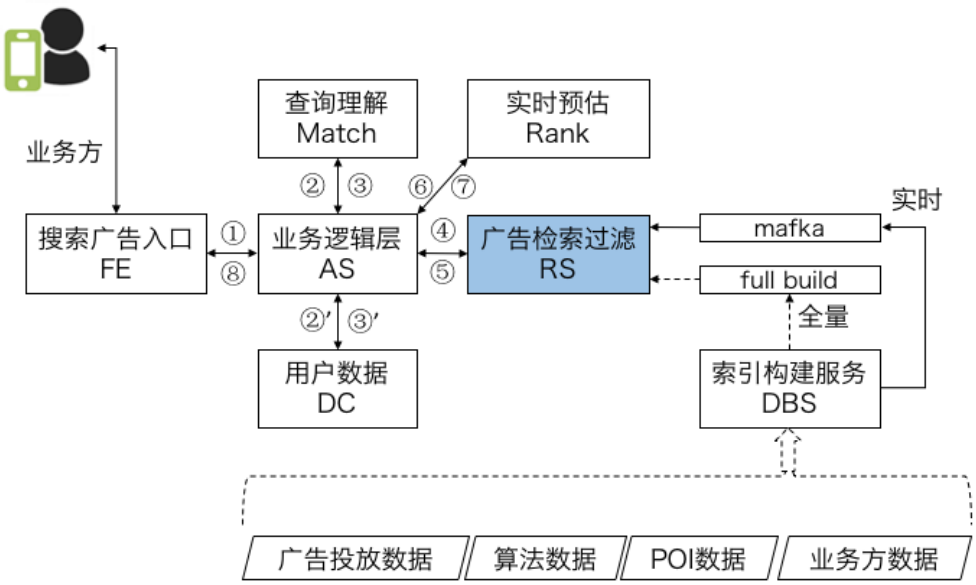
- 数据间的关联关系：实现描述关联关系的语法，即在描述外部数据源的同时就描述关联关系，后续字段查询时的路由将由框架处理。
- DFP依赖的SF信息：仅对单子段处理的简单DFP，可通过配置化方式，将依赖的SF固化在编译期；对多种数据源的复杂DFP，可通过源码分析来获取该DFP依赖的SF，无需用户维护依赖关系。

生产实践

早期的搜索广告是基于自然搜索的系统架构建的，随着业务的发展，需要根据广告特点进行系统改造。新的广告索引实现了纯粹的实时更新和层次化结构，已经在美团搜索广告上线。该架构也适用于各类非搜索的业务场景。

系统架构

作为整个系统的核心，基于实时索引构建的广告检索过滤服务（RS），承担了广告检索和各类业务过滤功能。日常的业务迭代，均可通过升级索引配置完成。



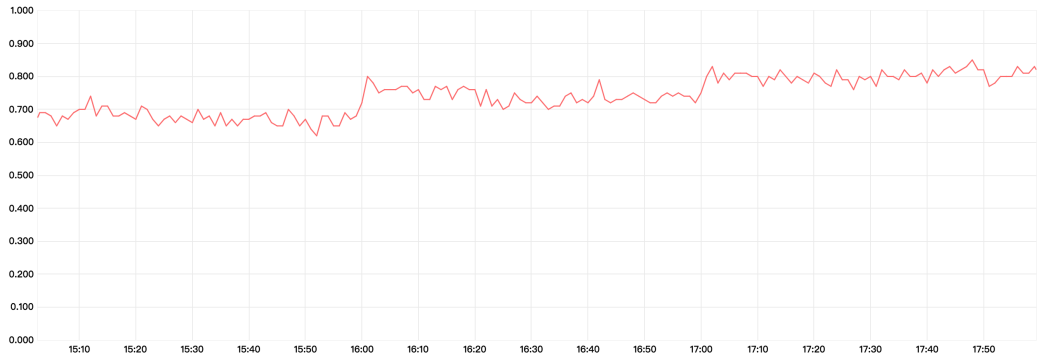
系统架构

系统架构

此外，为提升系统的吞吐量，多个模块已实现服务端异步化。

性能优化

以下为监控系统的性能曲线，索引中的doc数量为百万级别，时延的单位是毫秒。



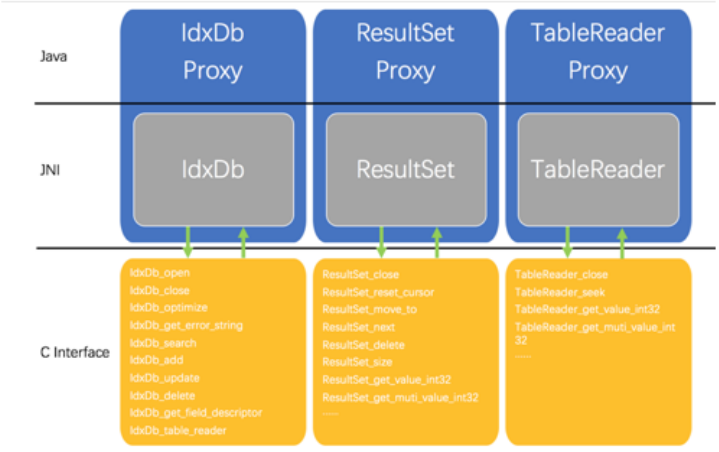
索引查询性能

后续规划

为便于实时索引与其他生产系统的结合，除进一步的性能优化和功能扩展外，我们还计划完成多项功能支持。

JNI

通过JNI，将Table作为单独的NoSQL，为Java提供本地缓存。如广告系统的实时预估模块，可使用Table存储模型使用的广告特征。



JNI

索引库JNI

SQL

提供SQL语法，提供简单的SQL支持，进一步降低使用门槛。提供JDBC，进一步简化Java的调用。

参考资料

- Apache Lucene http://lucene.apache.org/
- Sphinx http://sphinxsearch.com/
- “Understanding the Linux Virtual Memory Manager” https://www.kernel.org/doc/gorman/html/understand/
- Kyoto Cabinet http://fallabs.com/kyotocabinet/
- GNU Bison https://www.gnu.org/software/bison/

作者简介

- 仓魁：广告平台搜索广告引擎组架构师，主导实时广告索引系统的设计与实现。擅长C++、Java等多种编程语言，对异步化系统、后台服务调优等有深入研究。
- 晓晖：广告平台搜索广告引擎组核心开发，负责实时更新流的设计与实现。在广告平台率先尝试Scala语言，并将其用于大规模工程实践。
- 刘铮：广告平台搜索广告引擎组负责人，具有多年互联网后台开发经验，曾领导多次系统重构。
- 蔡平：广告平台搜索广告引擎组点评侧负责人，全面负责点评侧系统的架构和优化。

招聘信息

有志于从事Linux后台开发，对计算广告、高性能计算、分布式系统等有兴趣的朋友，请通过邮箱与我们联系。联系邮箱：liuzheng04@meituan.com 。

大众点评账号业务高可用进阶之路

作者: 沙堂堂 孟德鑫 杨正 谢可 徐升

引言

在任何一家互联网公司，不管其主营业务是什么，都会有一套自己的账号体系。账号既是公司所有业务发展留下的最宝贵资产，它可以用来衡量业务指标，例如日活、月活、留存等，同时也给不同业务线提供了大量潜在用户，业务可以基于账号来做用户画像，制定各自的发展路径。因此，账号服务的重要性不言而喻，同时美团业务飞速发展，对账号业务的可用性要求也越来越高。本文将分享一些我们在高可用探索中的实践。

衡量一个系统的可用性有两个指标：

1. MTBF (Mean Time Between Failure) 即平均多长时间不出故障；
2. MTTR (Mean Time To Recovery) 即出故障后的平均恢复时间。通过这两个指标可以计算出可用性，也就是我们大家比较熟悉的“几个9”。

$$Availability = \frac{MTBF}{MTBF + MTTR}$$

可用性公式

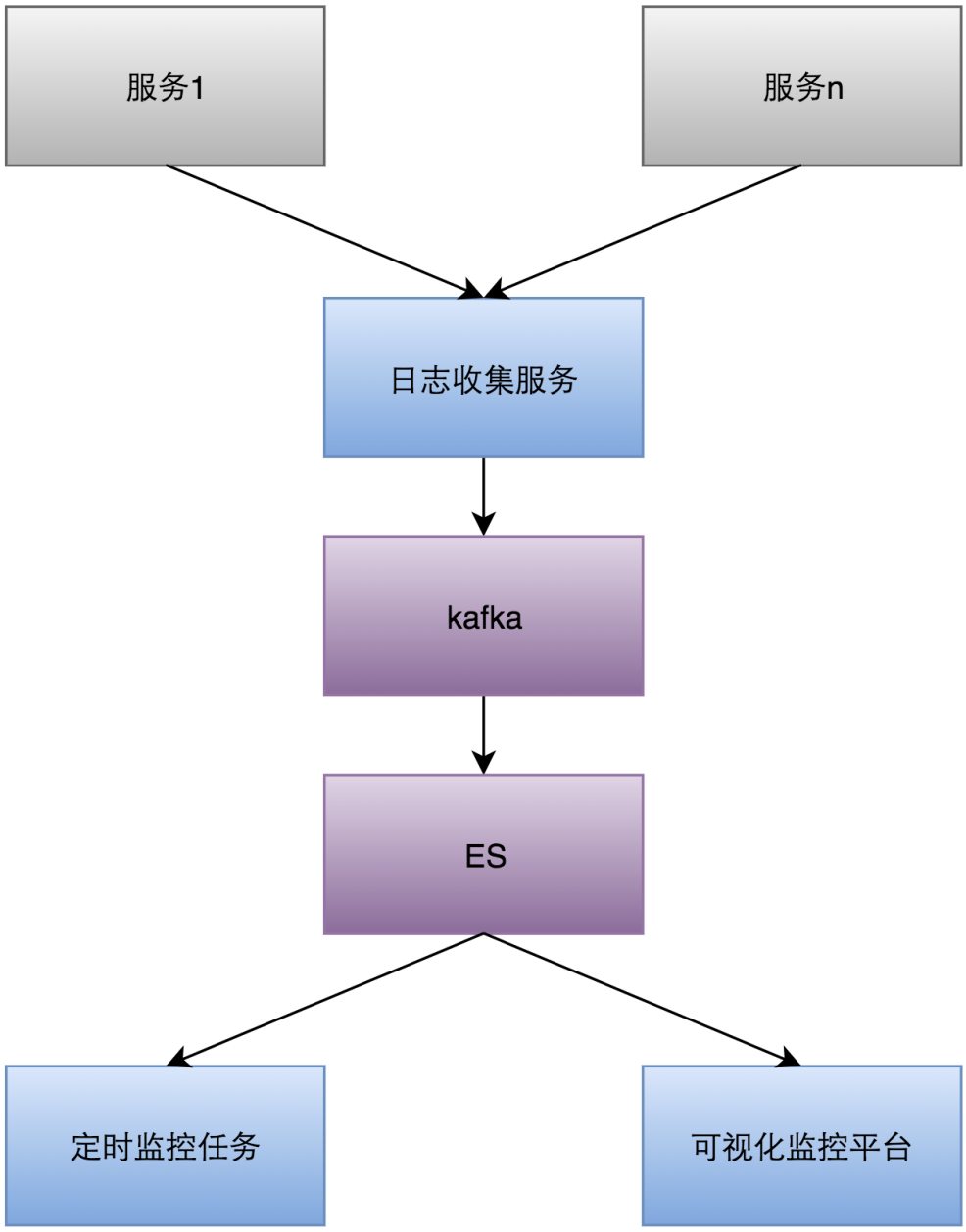
因此提升系统的可用性，就得从这两个指标入手，要么降低故障恢复的时间，要么延长不出故障的时间。

1. 业务监控

要降低故障恢复的时间，首先得尽早的发现故障，然后才能解决故障，这些故障包括系统内和系统外的，这就需要依赖业务监控系统。

业务监控不同于其他监控系统，业务监控关注的是各个业务指标是否正常，比如账号的登录曲线。大众点评登录入口有很多，从终端上分有App、PC、M站，从登录类型上分有密码登录、快捷登录、第三方登录（微信/QQ/微博）、小程序登录等。需要监控的维度有登录总数、成功数、失败分类、用户地区、App版本号、浏览器类型、登录来源Referer、服务所在机房等等。业务监控最能从直观上告诉我们系统的运行状况。

由于业务监控的维度很多很杂，有时还要增加新的监控维度，并且告警分析需要频繁聚合不同维度的数据，因此我们采用ElasticSearch作为日志存储。整体架构如下图：



每条监控都会根据过去的业务曲线计算出一条基线（见下图），用来跟当前数据做对比，超出设定的阈值后就会触发告警。

监控曲线图

每次收到告警，我们都要去找出背后的原因，如果是流量涨了，是有活动了还是被刷了？如果流量跌了，是日志延时了还是服务出问题了？另外值得重视的是告警的频次，如果告警太多就会稀释大家的警惕性。我们曾经踩过一次坑，因为告警太多就把告警关了，结果就在关告警的这段时间业务出问题了，我们没有及时发现。为了提高每条告警的定位速度，我们在每条告警后面加上维度分析。如下图（非真实数据），告警里直接给出分析结果。

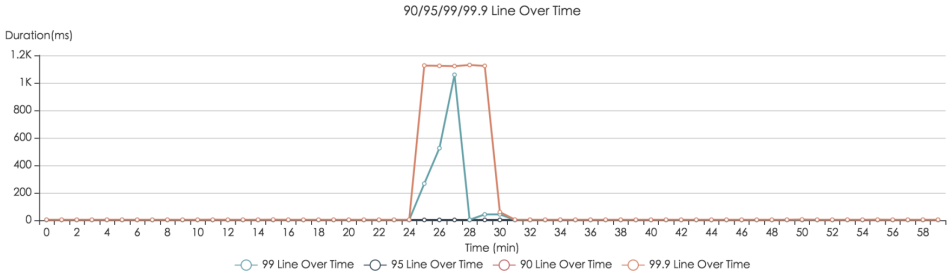
监控告警分析图

其实业务监控也从侧面反映出一个系统的可用性，所谓服务未动，监控先行。

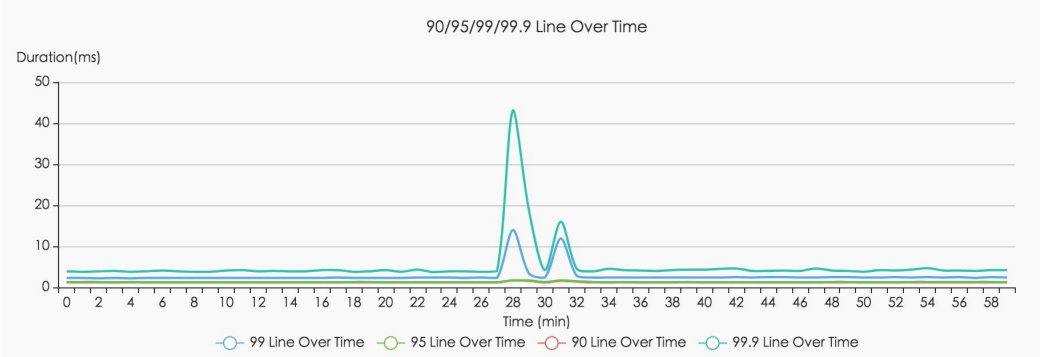
2. 柔性可用

柔性可用的目的是延长不出故障的时间，当业务依赖的下游服务出故障时不影响自身的核心功能或服务。账号对上层业务提供的鉴权和查询服务即核心服务，这些服务的QPS非常高，业务方对它们的可用性要求也很高，别说是服务故障，就连任何一点抖动都是不能接受的。对此我们先从整体架构上把服务拆分，其次在服务内对下游依赖做资源隔离，都尽可能的缩小故障发生时的影响范围。

另外对非关键路径上的服务故障做了降级。例如账号的一个查询服务依赖Redis，当Redis抖动的时候服务的可用性也随之降低，我们通过公司内部另外一套缓存中间件Cellar来做Redis的备用存储，当检测到Redis已经非常不可用时就切到Cellar上。通过开源组件Hystrix或者我们公司自研的中间件Rhino就能非常方便地解决这类问题，其原理是根据最近一个时间窗口内的失败率来预测下一个请求需不需要快速失败，从而自动降级，这些步骤都能在毫秒级完成，相比人工干预的情况提升几个数量级，因此系统的可用性也会大幅提高。下图是优化前后的对比图，可以非常明显的看到，系统的容错能力提升了，TP999也能控制在合理范围内。



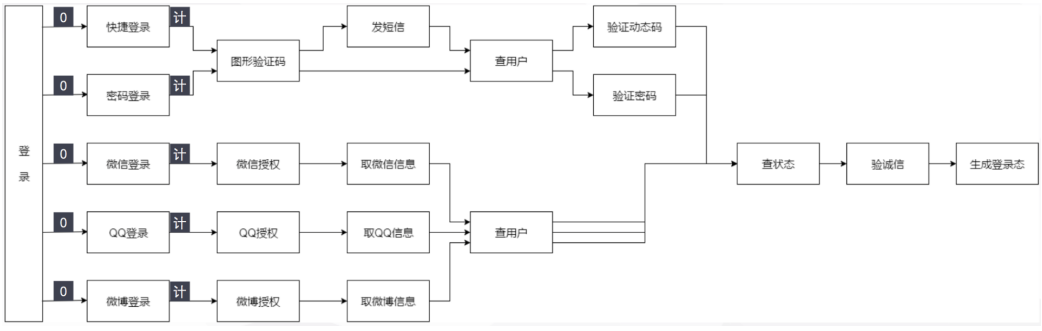
性能对比图前



性能对比图后

对于关键路径上的服务故障我们可以减少其影响的用户数。比如手机快捷登录流程里的某个关键服务挂了，我们可以在返回的失败文案上做优化，并且在登录入口挂小黄条提示，让用户主动去其他登录途径，这样对于那些设置过密码或者绑定了第三方的用户还有其他选择。

具体的做法是我们在每个登录入口都关联了一个计数器，一旦其中的关键节点不可用，就会在受影响的计数器上加1，如果节点恢复，则会减1，每个计数器还分别对应一个标志位，当计数器大于0时，标志位为1，否则标志位为0。我们可以根据当前标志位的值得知登录入口的可用情况，从而在登录页展示不同的提示文案，这些提示文案一共有2^5=32种。



登录降级流程图

下图是我们在做故障模拟时的降级提示文案：


下午5:18
密码登录
短信验证码登录暂不可用，请使用其他登录方式
+86
158
密码
请输入密码
登录
忘记密码?
手机号快捷登录
Debug
第三方账号登录
微信
QQ
更多


下午5:19
快捷登录
短信验证码登录暂不可用，请使用其他登录方式
+86
158
获取验证码
验证码
请输入验证码
未注册的手机号码验证后自动创建点评账户
登录
登录遇到问题
短信验证码暂时不可用，请使用其他登录方式！
Debug

3. 异地多活

除了柔性可用，还有一种思路可以来延长不出故障的时间，那就是做冗余，冗余的越多，系统的故障率就越低，并且是呈指数级降低。不管是机房故障，还是存储故障，甚至是网络故障，都能依赖冗余去解决，比如数据库可以通过增加从库的方式做冗余，服务层可以通过分布式架构做冗余，但是冗余也会带来新的问题，比如成本翻倍，复杂性增加，这就要衡量投入产出比。

目前美团的数据中心机房主要在北京上海，各个业务都直接或间接的依赖账号服务，尽管公司内已有北上专线，但因为专线故障或抖动引发的账号服务不可用，间接导致的业务损失也不容忽视，我们就开始考虑做跨城的异地冗余，即异地多活。

3.1 方案设计

首先我们调研了业界比较成熟的做法，主流思路是分set化，优点是非常利于扩展，缺点是只能按一个维度划分。比如按用户ID取模划分set，其他的像手机号和邮箱的维度就要做出妥协，尤其是这些维度还有唯一性要求，这就使得数据同步或者修改都增加了复杂度，而且极易出错，给后续维护带来困难。考虑到账号读多写少的特性（读写比是350:1），我们采用了一主多从的数据库部署方案，优先解决读多活的问题。

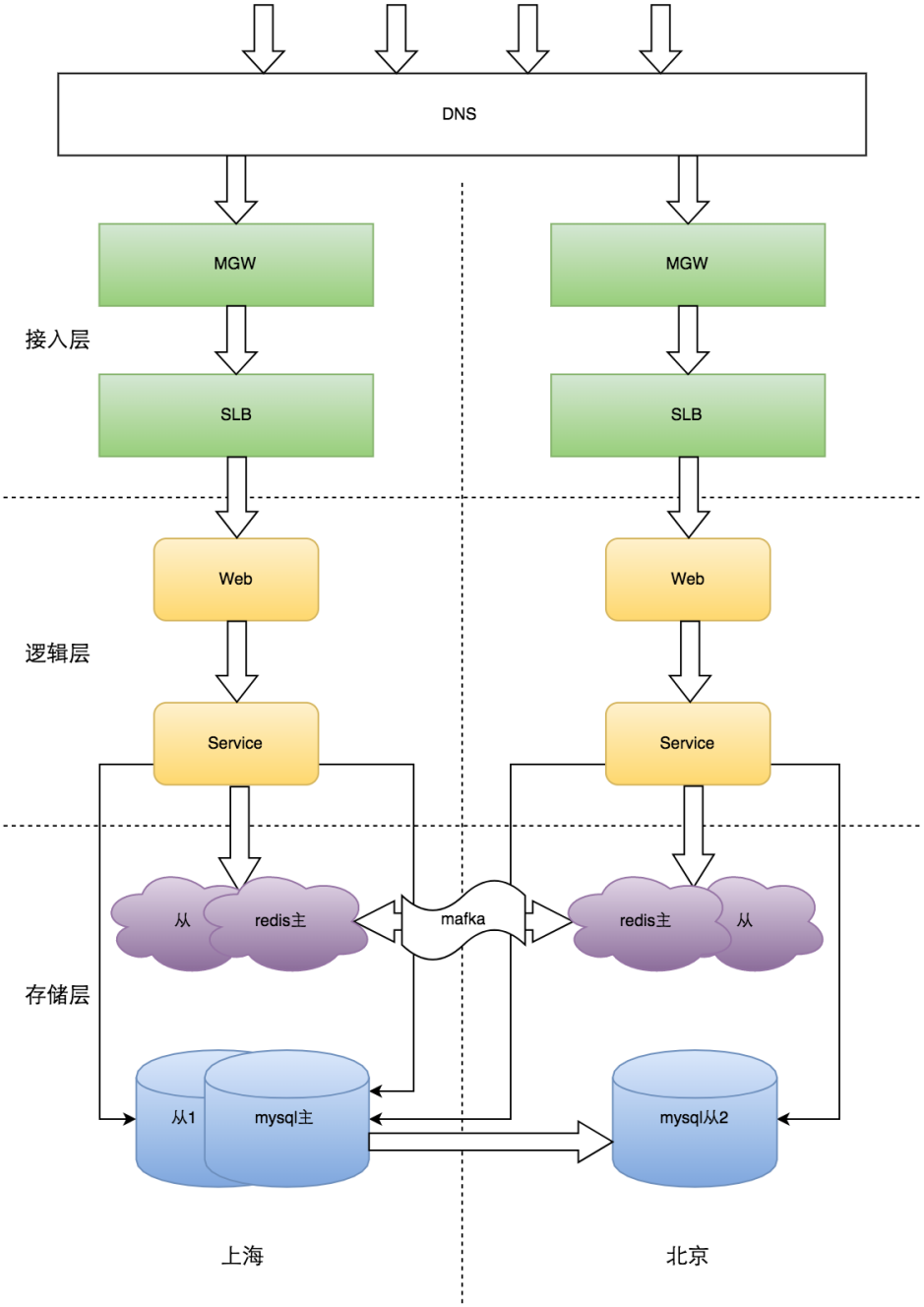
Redis如果也用一主多从的模式可行吗？答案是不行，因为Redis主从同步机制会优先尝试增量同步，当增量同步不成功时，再去尝试全量同步，一旦专线发生抖动就会把主库拖垮，并进一步阻塞专线，形成“雪崩效应”。因此两地的Redis只能是双主模式，但是这种架构有一个问题，就是我们得自己去解决数据同步的问题，除了保证数据不丢，还要保证数据一致。

另外从用户进来的每一层路由都要是就近的，因此DNS需要开启智能解析，SLB要开启同城策略，RPC已默认就近访问。

总体上账号的异地多活遵循以下三个原则：

1. 北上任何一地故障，另一地都可提供完整服务。
2. 北上两地同时对外提供服务，确保服务随时可用。
3. 两地服务都遵循BASE原则，确保数据最终一致。

最终设计方案如下：



异地多活架构图

3.2 数据同步

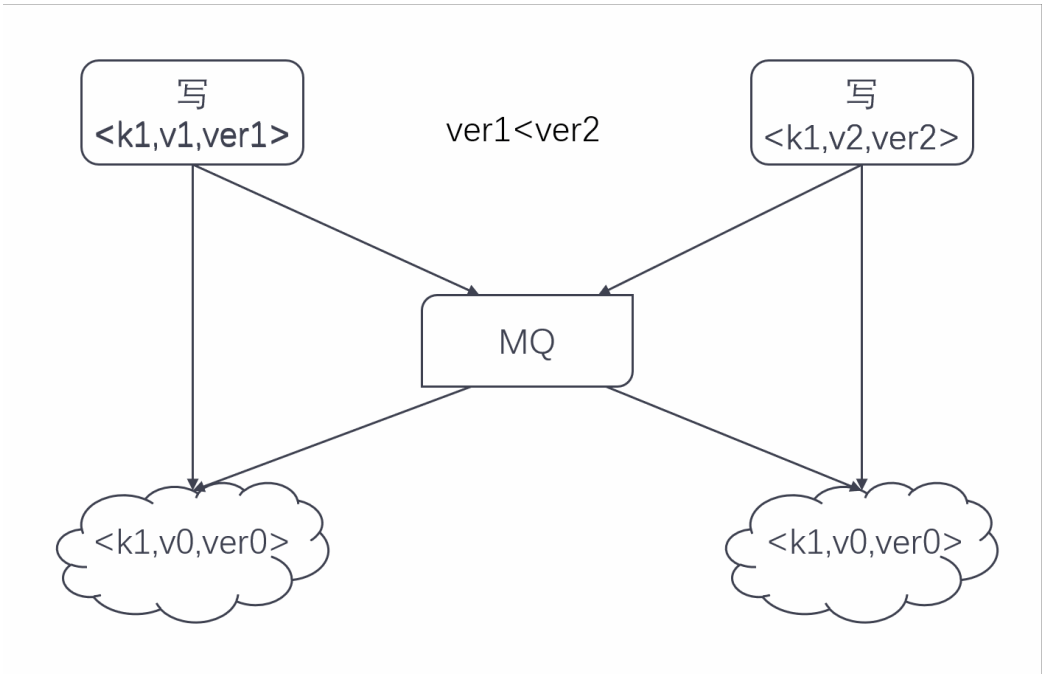
首先要保证数据在传输的过程中不能丢，因此需要一个可靠接收数据的地方，于是我们采用了公司内部的MQ平台Mafka（类Kafka）做数据中转站。可是消息在经过Mafka传递之后可能是乱序的，这导致对同一个key的一串操作序列可能导致不一致的结果，这是不可忍受的。但Mafka只是不保证全局有序，在单个partition内却是有序的，于是我们只要对每个key做一遍一致性散列算法对应一个partitionId，这样就能保证每个key的操作是有序的。

但仅仅有序还不够，两地的并发写仍然会造成数据的不一致。这里涉及到分布式数据的一致性问题，业界有两种普遍的认知，一种是Paxos协议，一种是Raft协议，我们吸取了对实现更为友好的Raft协议，它主张有一个主节点，其余是从节点，并且在主节点不可用时，从节点可晋升为主节点。简单来说就是把这些节点排个序，当写入有冲突时，以排在最前面的那个节点为准，其余节点都去follow那个主节点的值。在

技术实现上，我们设计出一个版本号（见下图），实际上是一个long型整数，其中数据源大小即表示节点的顺序，把版本号存入value里面，当两个写入发生冲突的时候只要比较这个版本号的大小即可，版本号大的覆盖小的，这样能保证写冲突时的数据一致性。

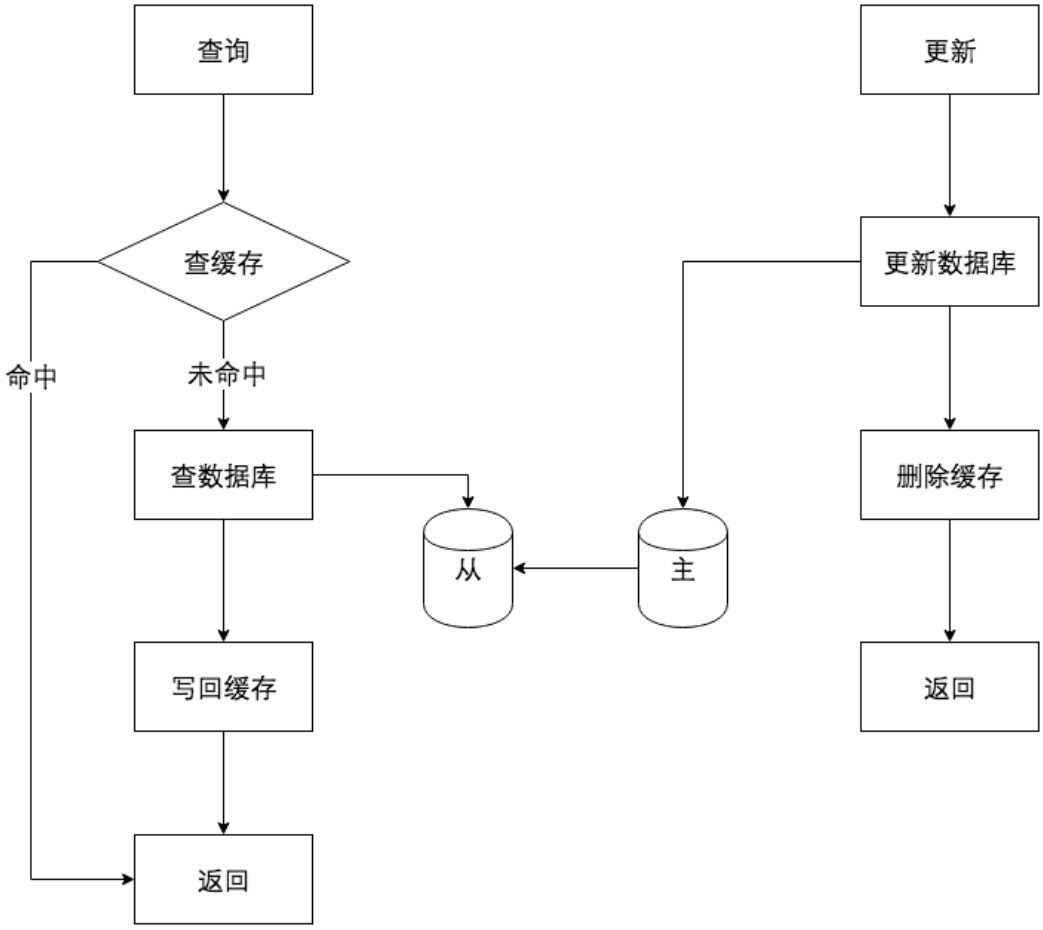
1位	38位	20位	5位
高位置0	时间戳(秒)	自增序列	数据源

写并发时数据同步过程如下图：



这种同步方式的好处显而易见，可以适用于所有的Redis操作且能保证数据的最终一致性。但这也有一些弊端，由于多存了版本号导致Redis存储会增加，另外在该机制下两地的数据其实是全量同步的，这对于那些仅用做缓存的存储来说是非常浪费资源的，因为缓存有数据库可以回源。而账号服务几乎一半的Redis存储都是缓存，因此我们需要对缓存同步做优化。

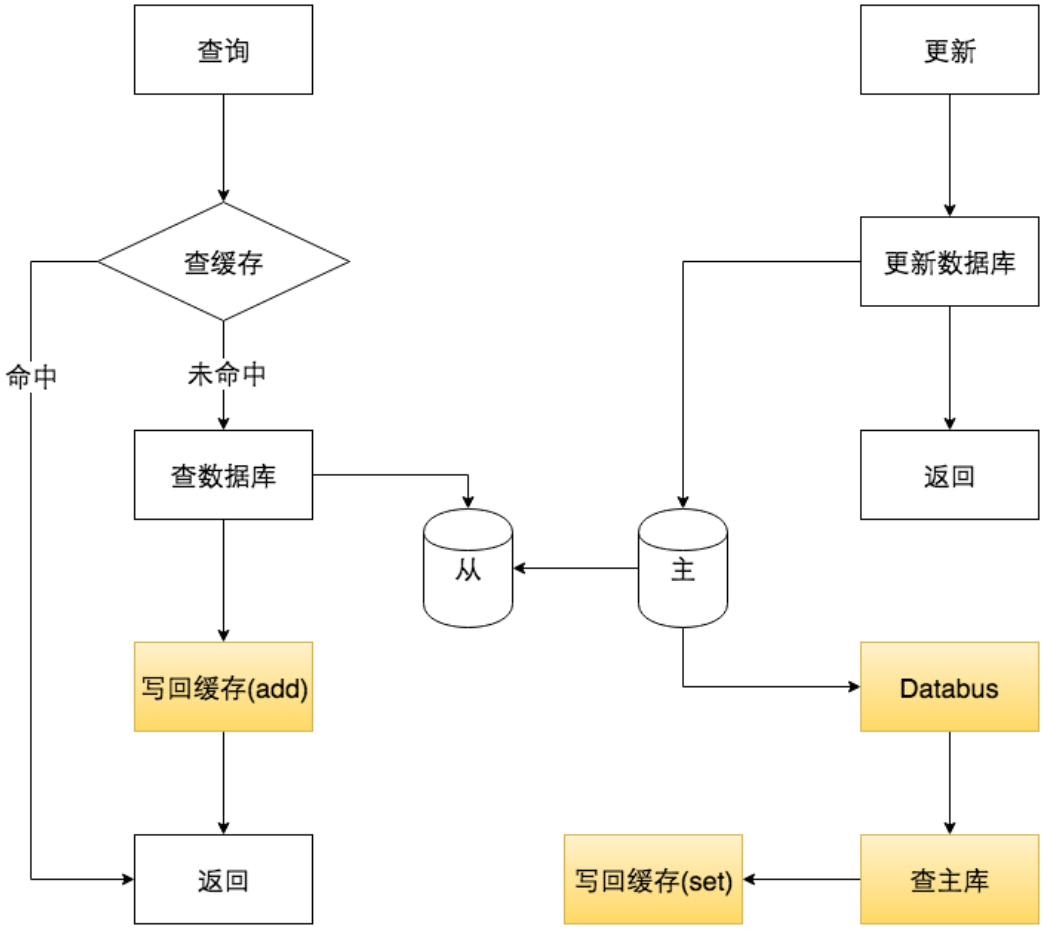
账号服务的缓存加载与更新模式如下图：



我们优化的方向是在缓存加载时不同步，只有在数据库有更新时才去同步。但是数据更新这个流程里不能再使用delete操作，这样做有可能使缓存出现脏数据，比如下面这个例子：

线程1（查询）	线程2（修改）
查缓存未命中	
查数据库	
	更新数据库
	删缓存
写回缓存	

我们对这个问题的解决办法是用set（若key不存在则添加，否则覆盖）代替delete，而缓存的加载用add（若key不存在则添加，否则不修改），这样能保证缓存更新时的强一致性却不需要增加额外存储。考虑到账号修改的入口比较多，我们希望缓存更新的逻辑能拎出来单独处理减少耦合，最后发现公司内部数据同步组件Databus非常适用于该场景，其主要功能是把数据库的变更日志以消息的形式发出来。于是优化后的缓存模式如下图：



从理论变为工程实现的时候还有些需要注意的地方，比如同步消息没发出去、数据收到后写失败了。因此我们还需要一个方法来检测数据不一致的数量，为了做到这点，我们新建了一个定时任务去scan两地的数据做对比统计，如果发现有不一致的还能及时修复掉。

项目上线后，我们也取得一些成果，首先性能提升非常明显，异地的调用平均耗时和TP99、TP999均至少下降80%，并且在一次线上专线故障期间，账号读服务对外的可用性并没有受影响，避免了更大范围的损失。

总结

服务的高可用需要持续性的投入与维护，比如我们会每月做一次容灾演练。高可用也不止体现在某一两个重点项目上，更多的体现在每个业务开发同学的日常工作里。任何一个小Bug都可能引起一次大的故障，让你前期所有的努力都付之东流，因此我们的每一行代码，每一个方案，每一次线上改动都应该是仔细推敲过的。高可用应该成为一种思维方式。最后希望我们能在服务高可用的道路上越走越远。

团队简介

账号团队拥有一群朝气蓬勃的成员：堂堂、德鑫、杨正、可可、徐升、艳豪，虽然他们之中有些刚毕业不久，但技术上都锐意进取，在讨论技术方案时观点鲜明，大家都充分地享受着思想碰撞的火花，这个年轻的团队在一起推进着高可用项目的进行，共同撑起了账号服务的平稳运行及业务发展。

招聘信息

- 如果你觉得我们的高可用仍有提升空间，欢迎来大众点评基础平台研发组。
- 如果你想更深入学习高可用的技术细节，欢迎来大众点评基础平台研发组。
- 如果你想遇到一群志同道合的技术开发，欢迎来大众点评基础平台研发组。
- 简历传送门：tangtang.sha#dianping.com。

美团容器平台架构及容器技术实践

作者: 欧阳坚

> **本文根据美团基础架构部/容器研发中心技术总监欧阳坚在2018 QCon（全球软件开发大会）上的演讲内容整理而成。**

背景

美团的容器集群管理平台叫做HULK。漫威动画里的HULK在发怒时会变成“绿巨人”，它的这个特性和容器的“弹性伸缩”很像，所以我们给这个平台起名为HULK。貌似有一些公司的容器平台也叫这个名字，纯属巧合。

2016年，美团开始使用容器，当时美团已经具备一定的规模，在使用容器之前就已经存在的各种系统，包括CMDB、服务治理、监控告警、发布平台等等。我们在探索容器技术时，很难放弃原有的资产。所以容器化的第一步，就是打通容器的生命周期和这些平台的交互，例如容器的申请/创建、删除/释放、发布、迁移等等。然后我们又验证了容器的可行性，证实容器可以作为线上核心业务的运行环境。

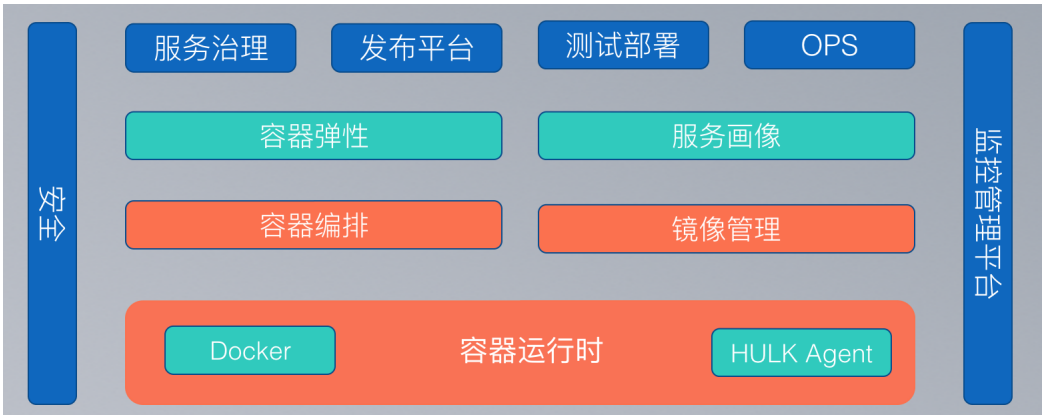
2018年，经过两年的运营和实践探索，我们对容器平台进行了一次升级，这就是容器集群管理平台HULK 2.0。

- 把基于OpenStack的调度系统升级成容器编排领域的事实标准Kubernetes（以后简称K8s）。
- 提供了更丰富可靠的容器弹性策略。
- 针对之前在基础系统上碰到的一些问题，进行了优化和打磨。

美团的容器使用状况是：目前线上业务已经超过3000个服务，容器实例数超过30000个，很多大并发、低延时要求的核心链路服务，已经稳定地运行在HULK之上。本文主要介绍我们在容器技术上的一些实践，属于基础系统优化和打磨。

美团容器平台的基本架构

首先介绍一下美团容器平台的基础架构，相信各家的容器平台架构大体都差不多。



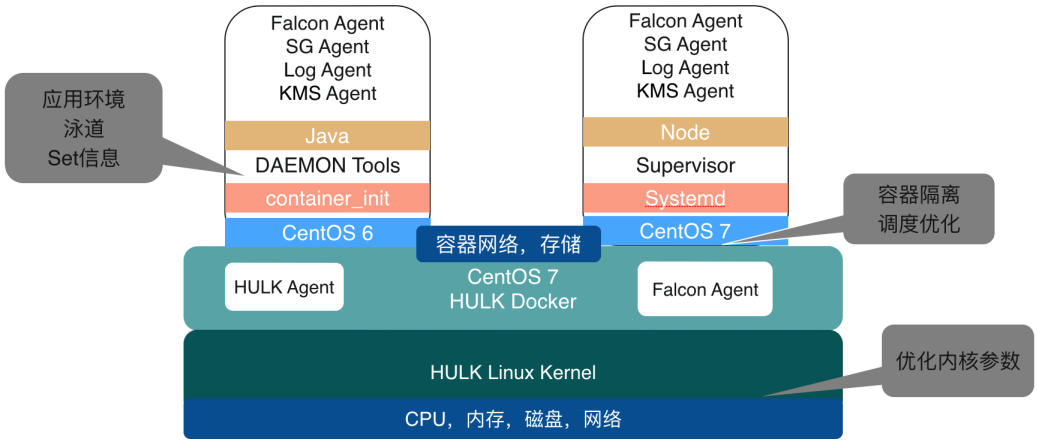
首先，容器平台对外对接服务治理、发布平台、CMDB、监控告警等等系统。通过和这些系统打通，容器实现了和虚拟机基本一致的使用体验。研发人员在使用容器时，可以和使用VM一样，不需要改变原来的使用习惯。

此外，容器提供弹性扩容能力，能根据一定的弹性策略动态增加和减少服务的容器节点数，从而动态地调整服务处理能力。这里还有个特殊的模块——“服务画像”，它的主要功能是通过对服务容器实例运行指标的搜集和统计，更好的完成调度容器、优化资源分配。比如可以根据某服务的容器实例的CPU、内存、IO等使用情况，来分辨这个服务属于计算密集型还是IO密集型服务，在调度时尽量把互补的容器放在一起。再比如，我们可以知道某个服务的每个容器实例在运行时会有大概500个进程，我们就会在创建容器时，给该容器加上一个合理的进程数限制（比如最大1000个进程），从而避免容器在出现问题时，占用过多的系统资源。如果这个服务的容器在运行时，突然申请创建20000个进程，我们有理由相信是业务容器遇到了Bug，通过之前的资源约束对容器进行限制，并发出告警，通知业务及时进行处理。

往下一层是“容器编排”和“镜像管理”。容器编排解决容器动态实例的问题，包括容器何时被创建、创建到哪个位置、何时被删除等等。镜像管理解决容器静态实例的问题，包括容器镜像应该如何构建、如何分发、分发的位置等等。

最下层是我们的容器运行时，美团使用主流的Linux+Docker容器方案，HULK Agent是我们在服务器上的管理代理程序。

把前面的“容器运行时”具体展开，可以看到这张架构图，按照从下到上的顺序介绍：



- 最下层是CPU、内存、磁盘、网络这些基础物理资源。

- 往上一层，我们使用的是CentOS7作为宿主机操作系统，Linux内核的版本是3.10。我们在CentOS发行版默认内核的基础上，加入一些美团为容器场景研发的新特性，同时为高并发、低延时的服务型业务做了一些内核参数的优化。
- 再往上一层，我们使用的是CentOS发行版里自带的Docker，当前的版本是1.13，同样，加入了一些我们自己的特性和增强。HULK Agent是我们自己开发的主机管理Agent，在宿主机上管理Agent。Falcon Agent同时存在于宿主机和容器内部，它的作用是收集宿主机和容器的各种基础监控指标，上报给后台和监控平台。
- 最上一层是容器本身。我们现在主要支持CentOS 6和CentOS 7两种容器。在CentOS 6中有一个container init进程，它是我们开发容器内部的1号进程，作用是初始化容器和拉起业务进程。在CentOS 7中，我们使用了系统自带的systemd作为容器中的1号进程。我们的容器支持各种主流编程语言，包括Java、Python、Node.js、C/C++等等。在语言层之上是各种代理服务，包括服务治理的Agent、日志Agent、加密Agent等等。同时，我们的容器也支持美团内部的一些业务环境，例如set信息、泳道信息等，配合服务治理体系，可以实现服务调用的智能路由。

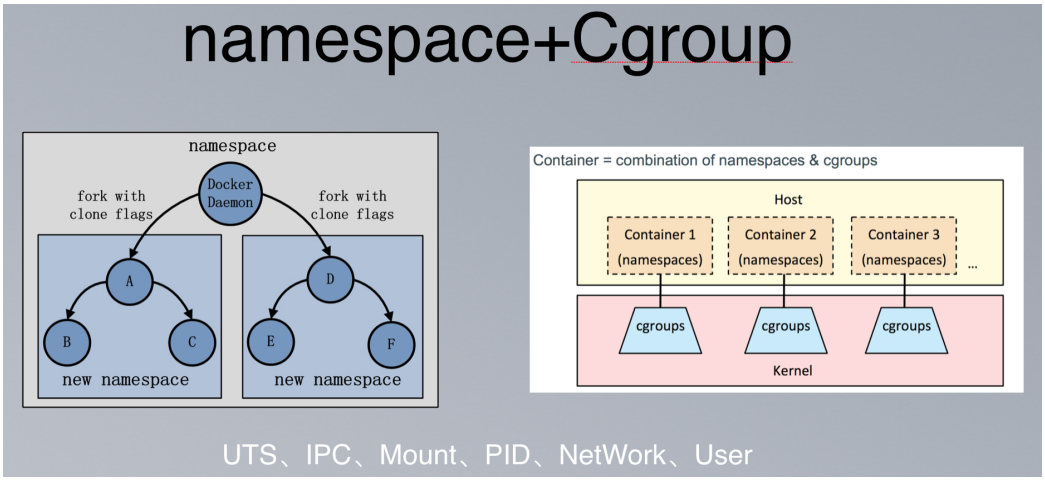
美团主要使用了CentOS系列的开源组件，因为我们认为Red Hat有很强的开源技术实力，比起直接使用开源社区的版本，我们希望Red Hat的开源版本能够帮助解决大部分的系统问题。我们也发现，即使部署了CentOS的开源组件，仍然有可能会碰到社区和Red Hat没有解决的问题。从某种程度上也说明，国内大型互联公司在技术应用的场景、规模、复杂度层面已经达到了世界领先的水平，所以才会先于社区、先于Red Hat的客户遇到这些问题。

容器遇到的一些问题

在容器技术本身，我们主要遇到了4个问题：隔离、稳定性、性能和推广。

- 隔离包含两个层面：第一个问题是，容器能不能正确认识自身资源配置；第二个问题是，运行在同一台服务器上的容器会不会互相影响。比如某一台容器的IO很高，就会导致同主机上的其他容器服务延时增加。
- 稳定性：这是指在高压力、大规模、长时间运行以后，系统功能可能会出现不稳定的问题，比如容器无法创建、删除，因为软件问题发生卡死、宕机等问题。
- 性能：在虚拟化技术和容器技术比较时，大家普遍都认为容器的执行效率会更高，但是在实践中，我们遇到了一些特例：同样的代码在同样配置的容器上，服务的吞吐量、响应时延反而不如虚拟机。
- 推广：当我们把前面几个问题基本上都解决以后，仍然可能会碰到业务不愿意使用容器的情况，其中原因一部分是技术因素，例如容器接入难易程度、周边工具、生态等都会影响使用容器的成本。推广也不是一个纯技术问题，跟公司内部的业务发展阶段、技术文化、组织设置和KPI等因素都密切相关。

容器的实现



容器本质上是把系统中为同一个业务目标服务的相关进程合成一组，放在一个叫做namespace的空间中，同一个namespace中的进程能够互相通信，但看不见其他namespace中的进程。每个namespace可以拥有自己独立的主机名、进程ID系统、IPC、网络、文件系统、用户等等资源。在某种程度上，实现了一个简单的虚拟：让一个主机上可以同时运行多个互不感知的系统。

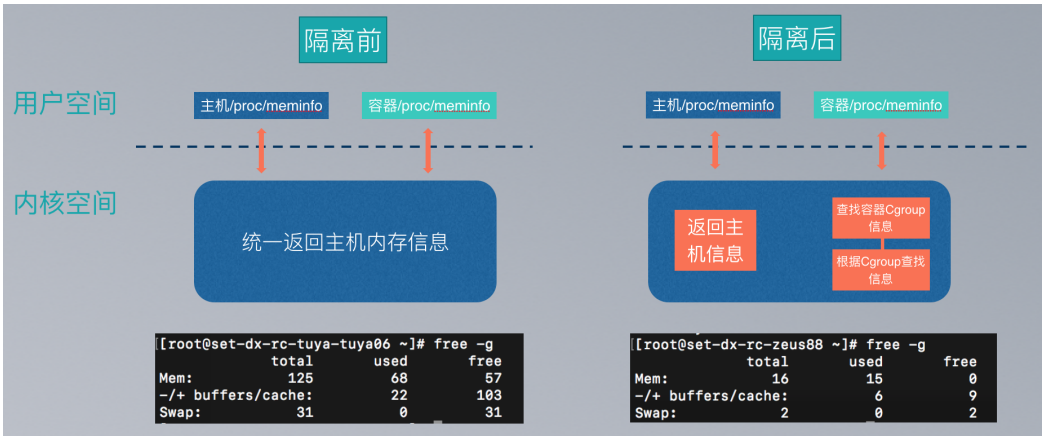
此外，为了限制namespace对物理资源的使用，对进程能使用的CPU、内存等资源需要做一定的限制。这就是Cgroup技术，Cgroup是Control group的意思。比如我们常说的4c4g的容器，实际上是限制这个容器namespace中所用的进程，最多能够使用4核的计算资源和4GB的内存。

简而言之，Linux内核提供namespace完成隔离，Cgroup完成资源限制。namespace+Cgroup构成了容器的底层技术（rootfs是容器文件系统层技术）。

美团的解法、改进和优化

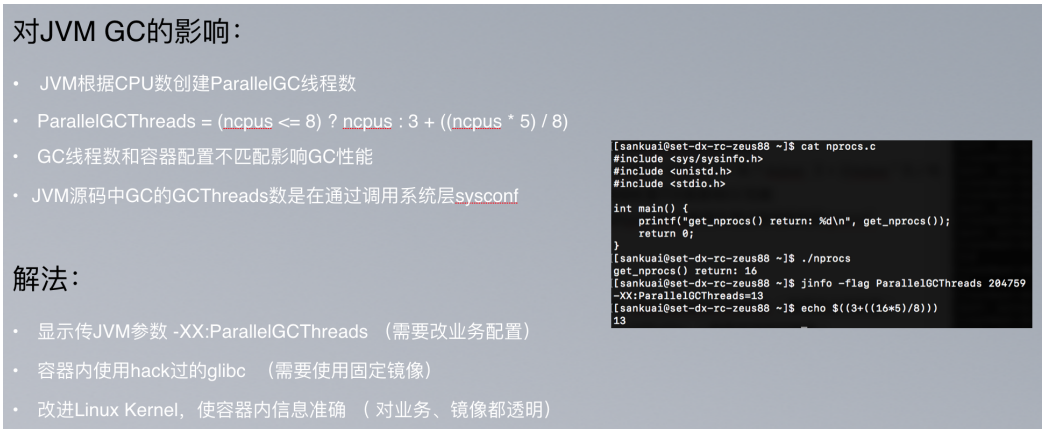
隔离

之前一直和虚拟机打交道，但直到用上容器，才发现在容器里面看到的CPU、Memory的信息都是服务器主机的信息，而不是容器自身的配置信息。直到现在，社区版的容器还是这样，比如一个4c4g的容器，在容器内部可以看到有40颗CPU、196GB内存的资源，这些资源其实是容器所在宿主机的信息。这给人的感觉，就像是容器的“自我膨胀”，觉得自己能力很强，但实际上并没有，还会带来很多问题。



上图是一个内存信息隔离的例子。获取系统内存信息时，社区Linux无论在主机上还是在容器中，内核都是统一返回主机的内存信息，如果容器内的应用，按照它发现的宿主机内存来进行配置的话，实际资源是远远不够的，导致的结果就是：系统很快会发生OOM异常。

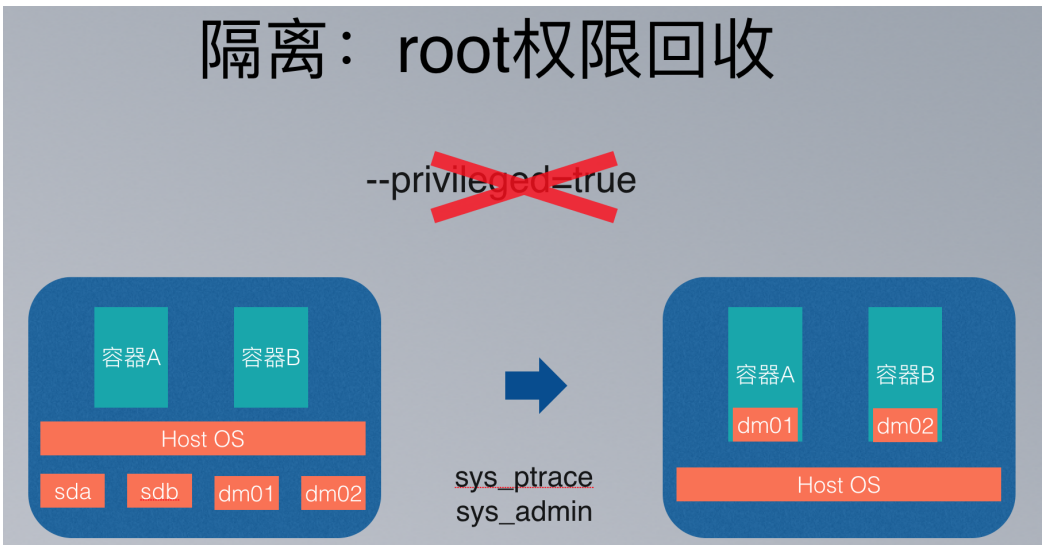
我们做的隔离工作，是在容器中获取内存信息时，内核根据容器的Cgroup信息，返回容器的内存信息（类似LXCFS的工作）。



CPU信息隔离的实现和内存的类似，不再赘述，这里举一个CPU数目影响应用性能例子。

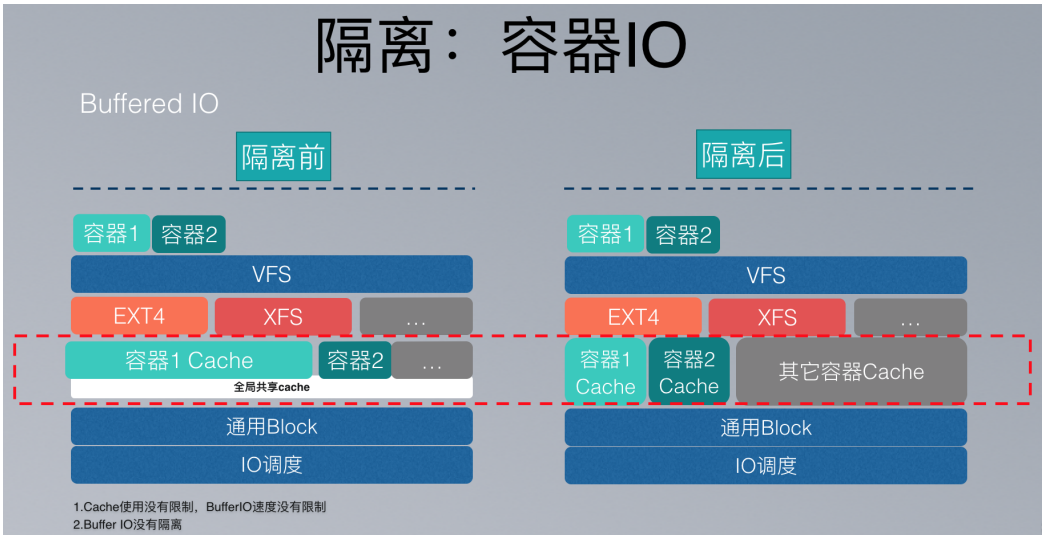
大家都知道，JVM GC（垃圾对象回收）对Java程序执行性能有一定的影响。默认的JVM使用公式“ParallelGCThreads = (ncpus <= 8) ? ncpus : 3 + ((ncpus * 5) / 8)” 来计算做并行GC的线程数，其中ncpus是JVM发现的系统CPU个数。一旦容器中JVM发现了宿主机的CPU个数（通常比容器实际CPU限制多很多），这就会导致JVM启动过多的GC线程，直接的结果就导致GC性能下降。Java服务的感受就是延时增加，TP监控曲线突刺增加，吞吐量下降。针对这个问题有各种解法：

- 显式的传递JVM启动参数“–XX:ParallelGCThreads”告诉JVM应该启动几个并行GC线程。它的缺点是需要业务感知，为不同配置的容器传不同的JVM参数。
- 在容器内使用Hack过的glibc，使JVM（通过sysconf系统调用）能正确获取容器的CPU资源数。我们在一段时间内使用的就是这种方法。其优点是业务不需要感知，并且能自动适配不同配置的容器。缺点是必须使用改过的glibc，有一定的升级维护成本，如果使用的镜像是原生的glibc，问题也仍然存在。
- 我们在新平台上通过对内核的改进，实现了容器中能获取正确CPU资源数，做到了对业务、镜像和编程语言都透明（类似问题也可能影响OpenMP、Node.js等应用的性能）。



有一段时间，我们的容器是使用root权限进行运行，实现的方法是在docker run的时候加入‘privileged=true’参数。这种粗放的使用方式，使容器能够看到所在服务器上所有容器的磁盘，导致了安全问题和性能问题。安全问题很好理解，为什么会导致性能问题呢？可以试想一下，每个容器都做一次磁盘状态扫描的场景。当然，权限过大的问题还体现在可以随意进行mount操作，可以随意的修改NTP时间等等。

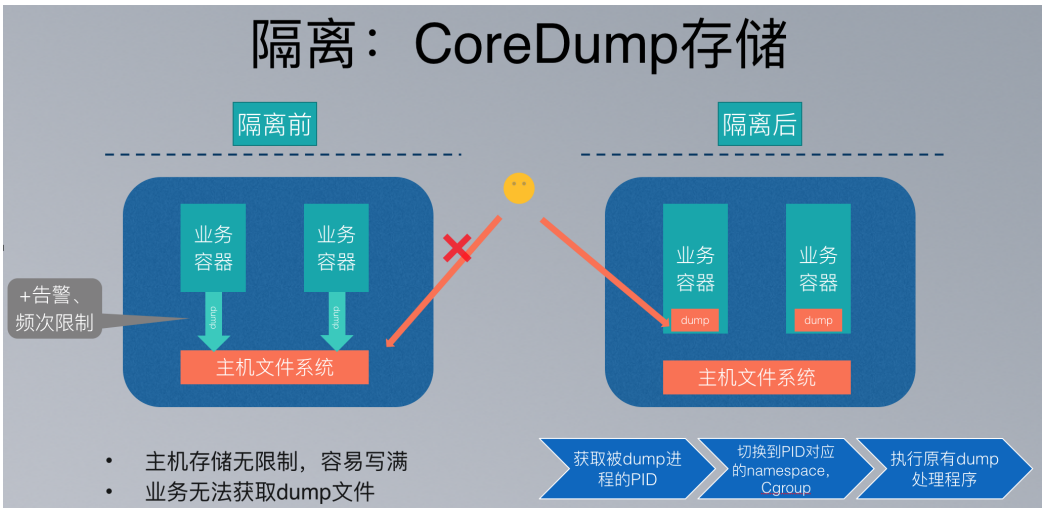
在新版本中，我们去掉了容器的root权限，发现有一些副作用，比如导致一些系统调用失败。我们默认给容器额外增加了sys_ptrace和sys_admin两个权限，让容器可以运行GDB和更改主机名。如果有特例容器需要更多的权限，可以在我们的平台上按服务粒度进行配置。



Linux有两种IO：Direct IO和Buffered IO。Direct IO直接写磁盘，Buffered IO会先写到缓存再写磁盘，大部分场景下都是Buffered IO。

我们使用的Linux内核3.X，社区版本中所有容器Buffer IO共享一个内核缓存，并且缓存不隔离，没有速率限制，导致高IO容器很容易影响同主机上的其他容器。Buffer IO缓存隔离和限速在Linux 4.X里通过Cgroup V2实现，有了明显的改进，我们还借鉴了Cgroup V2的思想，在我们的Linux 3.10内核实现了相同的功能：每个容器根据自己的内存配置有对应比例的IO Cache，Cache的数据写到磁盘的速率受容器Cgroup IO配置的限制。

Docker本身支持较多对容器的Cgroup资源限制，但是K8s调用Docker时可以传递的参数较少，为了降低容器间的互相影响，我们基于服务画像的资源分配，对不同服务的容器设定不同的资源限制，除了常见的CPU、内存外，还有IO的限制、ulimit限制、PID限制等等。所以我们扩展了K8s来完成这些工作。



业务在使用容器的过程中产生core dump文件是常见的事，比如C/C++程序内存访问越界，或者系统OOM的时候，系统选择占用内存多的进程杀死，默认都会生成一个core dump文件。

社区容器系统默认的core dump文件会生成在宿主机上，由于一些core dump文件比较大，比如JVM的core dump通常是几个GB，或者有些存在Bug的程序，其频发的core dump很容易快速写满宿主机的存储，并且会导致高磁盘IO，也会影响到其他容器。还有一个问题是：业务容器的使用者没有权限访问宿主机，从而拿不到dump文件进行下一步的分析。

为此，我们对core dump的流程进行了修改，让dump文件写到容器自身的文件系统中，并且使用容器自己的Cgroup IO吞吐限制。

稳定性

我们在实践中发现，影响系统稳定性的主要是Linux Kernel和Docker。虽然它们本身是很可靠的系统软件，但是在大规模、高强度的场景中，还是会存在一些Bug。这也从侧面说明，我们国内互联网公司在应用规模和应用复杂度层面也属于全球领先。

在内核方面，美团发现了Kernel 4.x Buffer IO限制的实现问题，得到了社区的确认和修复。我们还跟进了一系列CentOS的Ext4补丁，解决了一段时间内进程频繁卡死的问题。

我们碰到了两个比较关键的Red Hat版Docker稳定性问题：

- 在Docker服务重启以后，Docker exec无法进入容器，这个问题比较复杂。在解决之前我们用nsenter来代替Docker exec并积极反馈给RedHat。后来Red Hat在今年初的一个更新解决了这个问题。 https://access.redhat.com/errata/RHBA-2017:1620
- 是在特定条件下Docker Daemon会Panic，导致容器无法删除。经过我们自己Debug，并对比最新的代码，发现问题已经在Docker upstream中得到解决，反馈给Red Hat也很快得到了解决。 https://github.com/projectatomic/containerd/issues/2

面对系统内核、Docker、K8s这些开源社区的系统软件，存在一种观点是：我们不需要自己分析问题，只需要拿社区的最新更新就行了。但是我们并不认同，我们认为技术团队自身的能力很重要，主要是如下原因：

- 美团的应用规模大、场景复杂，很多问题也许很多企业都没有遇到过，不能被动的等别人来解答。
- 对于一些实际的业务问题或者需求（例如容器内正确返回CPU数目），社区也许觉得不重要，或者不是正确的理念，可能就不会解决。

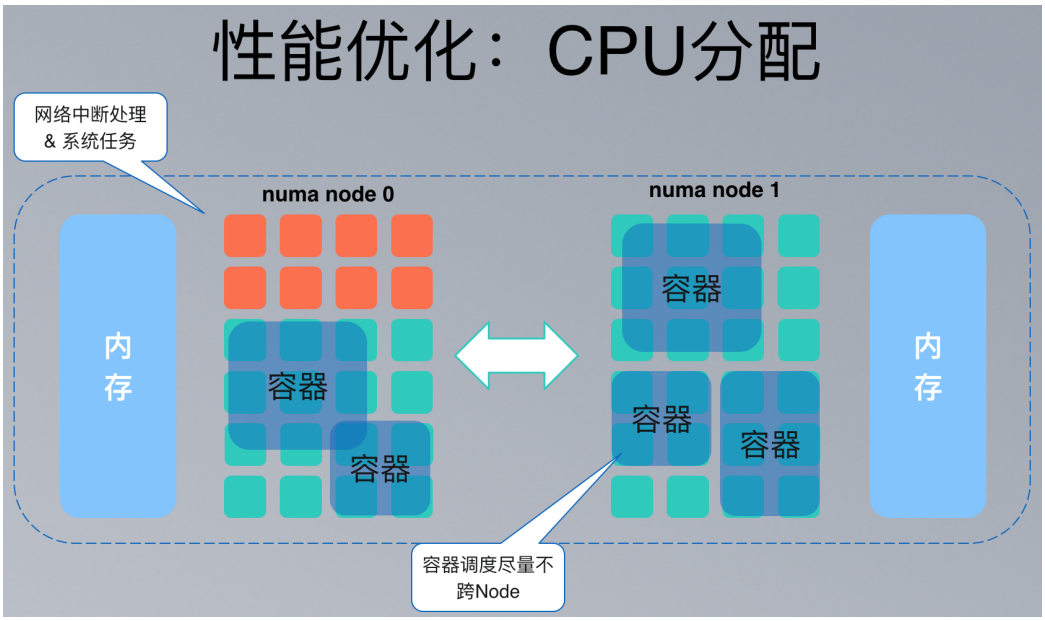
- 社区很多时候只在Upstream解决问题，而Upstream通常不稳定，即使有Backport到我们正在使用的版本，排期也很难进行保障。
- 社区会发布很多补丁，通常描述都比较晦涩难懂。如果没有对问题的深刻理解，很难把遇到的实际问题和一系列补丁联系起来。
- 对于一些复杂问题，社区的解决方案不一定适用于我们自身的实际场景，我们需要自身有能力进行判断和取舍。

美团在解决开源系统问题时，一般会经历五个阶段：自己深挖、研发解决、关注社区、和社区交互，最后贡献给社区。

性能

容器平台性能，主要包括两个方面性能：

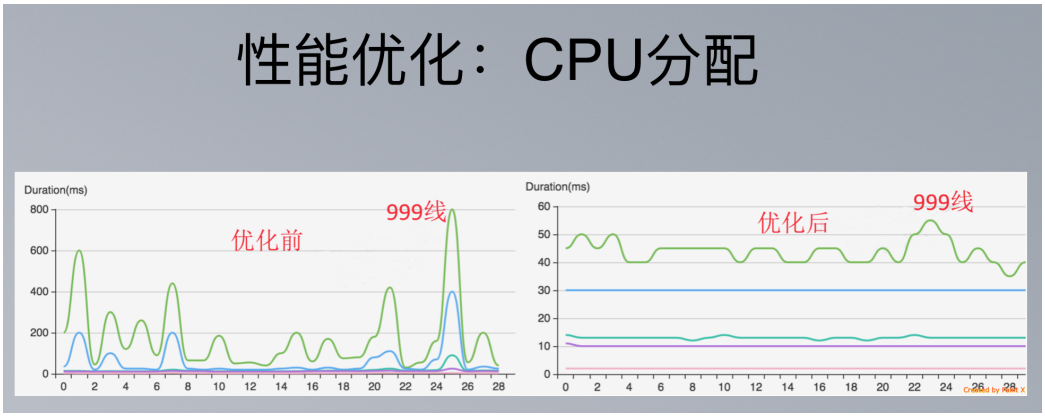
- 业务服务运行在容器上的性能。
- 容器操作（创建、删除等等）的性能。



上图是我们CPU分配的一个例子，我们采用的主流服务器是两路24核服务器，包含两个Node，每个12核，算上超线程共48颗逻辑CPU。属于典型的NUMA（非一致访存）架构：系统中每个Node有自己的内存，Node内的CPU访问自己的内存的速度，比访问另一个Node内存的速度快很多（差一倍左右）。

过去我们曾经遇到过网络中断集中到CPU0上的问题，在大流量下可能导致网络延时增加甚至丢包。为了保证网络处理能力，我们从Node0上划出了8颗逻辑CPU用来专门处理网络中断和宿主机系统上的任务，例如镜像解压这类高CPU的工作，这8颗逻辑CPU不运行任何容器的Workload。

在容器调度方面，我们的容器CPU分配尽量不跨Node，实践证明跨Node访问内存对应用性能的影响比较大。在一些计算密集型的场景下，容器分配在Node内部会提升30%以上的吞吐量。按Node的分配方案也存在一定的弊端：会导致CPU的碎片增加，为了更高效地利用CPU资源。在实际系统中，我们会根据服务画像的信息，分配一些对CPU不敏感的服务容器跨Node使用CPU资源。



上图是一个真实的服务在CPU分配优化前后，响应延时的TP指标线对比。可以看到TP999线下降了一个数量级，所有的指标都更加平稳。

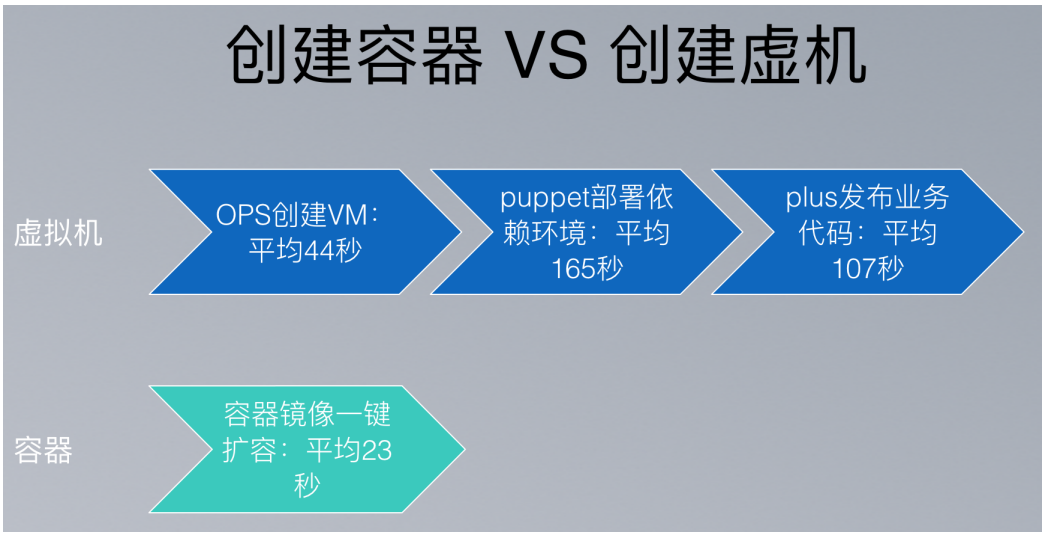
性能优化：文件系统

针对文件系统的性能优化，第一步是选型，根据统计到的应用读写特征，我们选择了Ext4文件系统（超过85%的文件读写是对小于1M文件的操作）。

Ext4文件系统有三种日志模式：

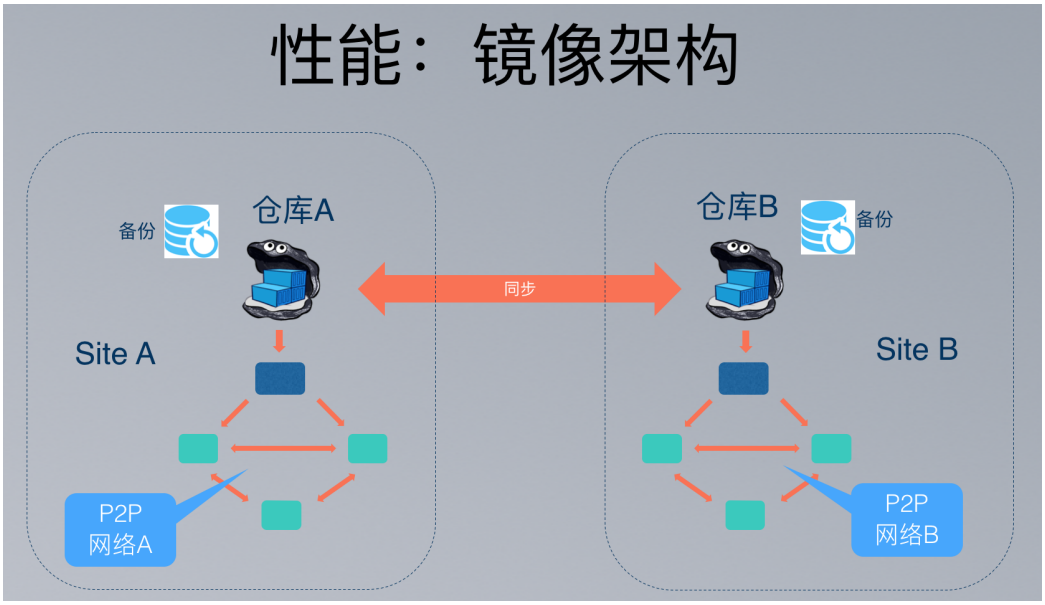
- Journal：写数据前等待Metadata和数据的日志落盘。
- Ordered：只记录Metadata的日志，写Metadata日志前确保数据已经落盘。
- Writeback：仅记录Metadata日志，不保证数据比Metadata先落盘。

我们选择了Writeback模式（默认是oderded），它在几种挂载模式中速度最快，缺点是：发生故障时数据不好恢复。我们大部分容器处于无状态，故障时在别的机器上再拉起一台即可。因此我们在性能和稳定性中，选择了性能。容器内部给应用提供可选的基于内存的文件系统tmpfs，可以提升有大量临时文件读写的服务性能。



如上图所示，在美团内部创建一个虚拟机至少经历三步，平均时间超过300秒。使用镜像创建容器平均时间23秒。容器的灵活、快速得到了显著的体现。

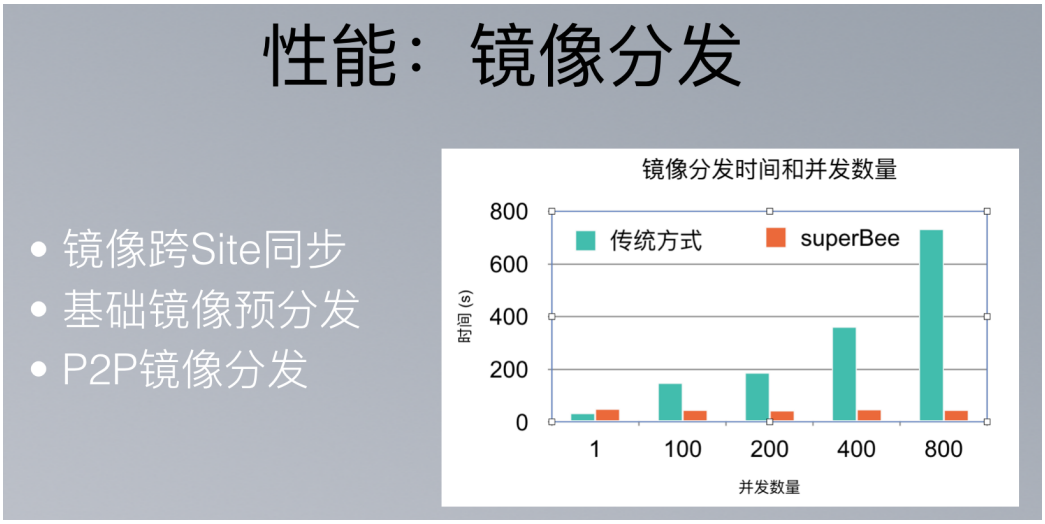
容器扩容23秒的平均时间包含了各个部分的优化，如扩容链路优化、镜像分发优化、初始化和业务拉起优化等等。接下来，本文主要介绍一下我们做的镜像分发和解压相关的优化。



上图是美团容器镜像管理的总体架构，其特点如下：

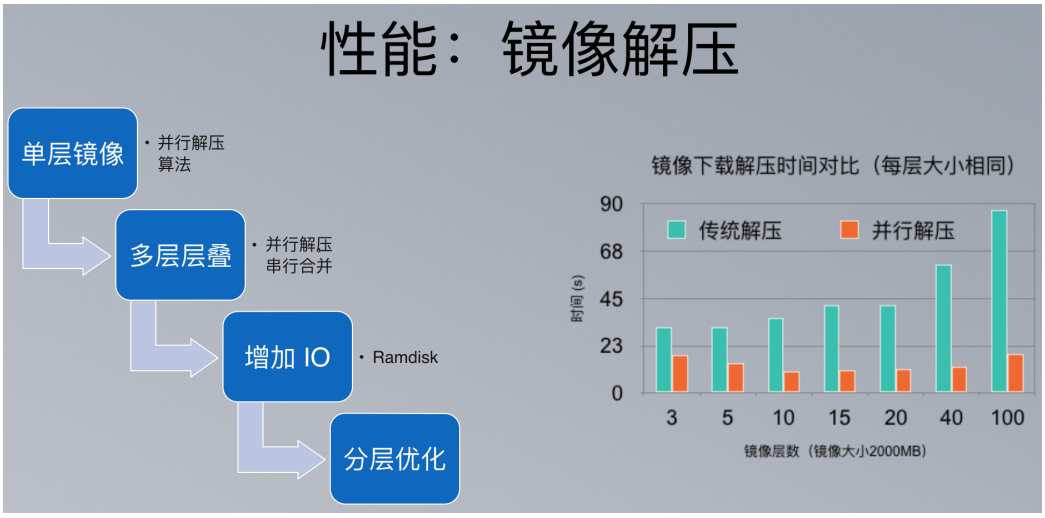
- 存在多个Site。
- 支持跨Site的镜像同步，根据镜像的标签确定是否需要跨Site同步。
- 每个Site有镜像备份。
- 每个Site内部有实现镜像分发的P2P网络。

镜像分发是影响容器扩容时长的一个重要环节。



- 跨Site同步：保证服务器总能从就近的镜像仓库拉取到扩容用的镜像，减少拉取时间，降低跨Site带宽消耗。
- 基础镜像预分发：美团的基础镜像是构建业务镜像的公共镜像，通常有几百兆的大小。业务镜像层是业务的应用代码，通常比基础镜像小很多。在容器扩容的时候如果基础镜像已经在本地，就只需要拉取业务镜像的部分，可以明显的加快扩容速度。为达到这样的效果，我们会把基础镜像事先分发到所有的服务器上。
- P2P镜像分发：基础镜像预分发在有些场景会导致上千个服务器同时从镜像仓库拉取镜像，对镜像仓库服务和带宽带来很大的压力。因此我们开发了镜像P2P分发的功能，服务器不仅能从镜像仓库中拉取镜像，还能从其他服务器上获取镜像的分片。

从上图可以看出，随着分发服务器数目的增加，原有分发时间也快速增加，而P2P镜像分发时间基本上保持稳定。



Docker的镜像拉取是一个并行下载，串行解压的过程，为了提升解压的速度，我们美团也做了一些优化工作。

对于单个层的解压，我们使用并行解压算法替换Docker默认的串行解压算法，实现上是使用pgzip替换gzip。

Docker的镜像具有分层结构，对镜像层的合并是一个“解压一层合并一层，再解压一层，再合并一层”的串行操作。实际上只有合并是需要串行的，解压可以并行起来。我们把多层的解压改成并行，解压出的数据先放在临时存储空间，最后根据层之间的依赖进行串行合并。前面的改动（并行解压所有的层到临时空间）导致磁盘IO的次数增加了近一倍，也会导致解压过程不够快。于是，我们使用基于内存的Ramdisk来存储解压出来的临时文件，减轻了额外文件写带来的开销。做了上面这些工作以后，我们又发现，容器的分层也会影响下载加解压的时间。上图是我们简单测试的结果：无论对于怎么分层的镜像并行解压，都能大幅提升解压时间，对于层数多的镜像提升更加明显。

推广

推广容器的第一步是能说出容器的优势，我们认为容器有如下优势：

- 轻量级：容器小、快，能够实现秒级启动。
- 应用分发：容器使用镜像分发，开发测试容器和部署容器配置完全一致。
- 弹性：可以根据CPU、内存等资源使用或者QPS、延时等业务指标快速扩容容器，提升服务能力。

这三个特性的组合，可以给业务带来更大的灵活度和更低的计算成本。

因为容器平台本身是一个技术产品，它的客户是各个业务的RD团队，因此我们需要考虑下面一些因素：

- **产品优势**：推广容器平台从某种程度上讲，自身是一个ToB的业务，首先要有好的产品，它相对于以前的解决方案（虚拟机）存在很多优势。
- **和已有系统打通**：这个产品要能和客户现有的系统很好的进行集成，而不是让客户推翻所有的系统重新再来。
- **原生应用的开发平台、工具**：这个产品要易于使用，要有配合工作的工具链。
- **虚拟机到容器的平滑迁移**：最好能提供从原有方案到新产品的迁移方案，并且容易实施。
- **与应用RD紧密配合**：要提供良好的客户支持，（即使有些问题不是这个产品导致的也要积极帮忙解决）。
- **资源倾斜**：从战略层面支持颠覆性新技术：资源上向容器平台倾斜，没有足够的理由，尽量不给配置虚拟机资源。

总结

Docker容器加Kubernetes编排是当前容器云的主流实践之一，美团容器集群管理平台HULK也采用了这样的方案。本文主要分享了美团在容器技术上做的一些探索和实践。内容主要涵盖美团容器云在Linux Kernel、Docker和Kubernetes层面做的一些优化工作，以及美团内部推动容器化进程的一些思考，欢迎大家跟我们交流、探讨。

作者简介

- 欧阳坚，2006年毕业于清华大学计算机系，拥有12年数据中心开发管理经验。曾任VMware中国Staff Engineer，无双科技CTO，中科睿光首席架构师。现任美团基础架构部/容器研发中心技术总监，负责美团容器化的相关工作。

招聘信息

美团点评基础架构团队诚招Java高级、资深技术专家，Base北京、上海。我们是集团致力于研发公司级、业界领先基础架构组件的核心团队，涵盖分布式监控、服务治理、高性能通信、消息中间件、基础存储、容器化、集群调度等技术领域。欢迎有兴趣的同学投送简历到 liuxing14@meituan.com。

美团即时物流的分布式系统架构设计

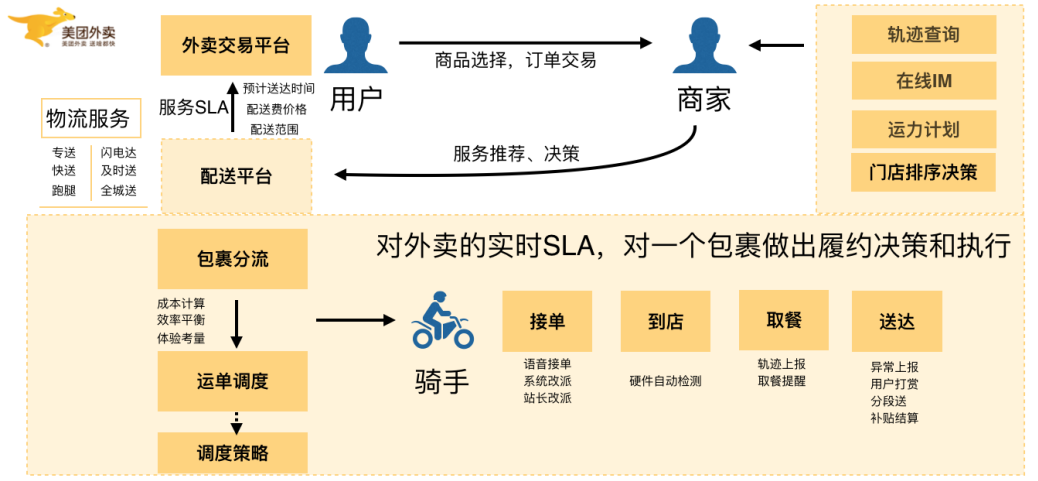
作者: 宋斌

> 本文根据美团资深技术专家宋斌在ArchSummit架构师峰会上的演讲整理而成。

背景

美团外卖已经发展了五年，即时物流探索也经历了3年多的时间，业务从零孵化到初具规模，在整个过程中积累了一些分布式高并发系统的建设经验。最主要的收获包括两点：

1. 即时物流业务对故障和高延迟的容忍度极低，在业务复杂度提升的同时也要求系统具备分布式、可扩展、可容灾的能力。即时物流系统阶段性的逐步实施分布式系统的架构升级，最终解决了系统宕机的风险。
2. 围绕成本、效率、体验核心三要素，即时物流体系大量结合AI技术，从定价、ETA、调度、运力规划、运力干预、补贴、核算、语音交互、LBS挖掘、业务运维、指标监控等方面，业务突破结合架构升级，达到促规模、保体验、降成本的效果。



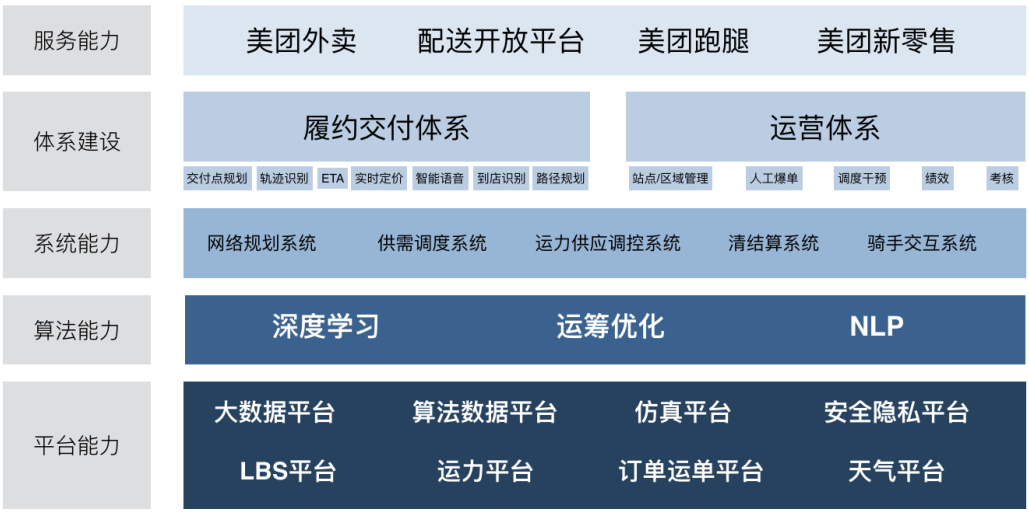
本文主要介绍在美团即时物流分布式系统架构逐层演变的进展中，遇到的技术障碍和挑战：

- 订单、骑手规模大，供需匹配过程的超大规模计算问题。
- 遇到节假日或者恶劣天气，订单聚集效应，流量高峰是平常的十几倍。
- 物流履约是线上连接线下的关键环节，故障容忍度极低，不能宕机，不能丢单，可用性要求极高。
- 数据实时性、准确性要求高，对延迟、异常非常敏感。

美团即时物流架构

美团即时物流配送平台主要围绕三件事展开：一是面向用户提供履约的SLA，包括计算送达时间ETA、配送费定价等；二是在多目标（成本、效率、体验）优化的背景下，匹配最合适的骑手；三是提供骑手完整

履约过程中的辅助决策，包括智能语音、路径推荐、到店提醒等。

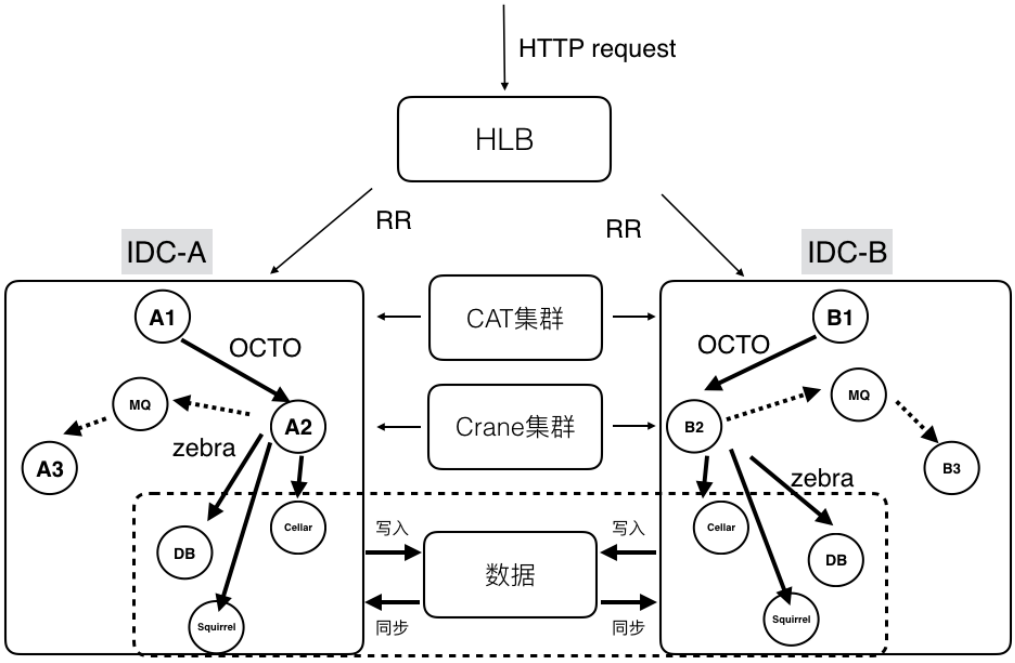


在一系列服务背后，是美团强大的技术体系的支持，并由此沉淀出的配送业务架构体系，基于架构构建的平台、算法、系统和服务。庞大的物流系统背后离不开分布式系统架构的支撑，而且这个架构更要保证高可用和高并发。

分布式架构，是相对于集中式架构而言的一种架构体系。分布式架构适用CAP理论（Consistency 一致性，Availability 可用性，Partition Tolerance 分区容忍性）。在分布式架构中，一个服务部署在多个对等节点中，节点之间通过网络进行通信，多个节点共同组成服务集群来提供高可用、一致性的服务。

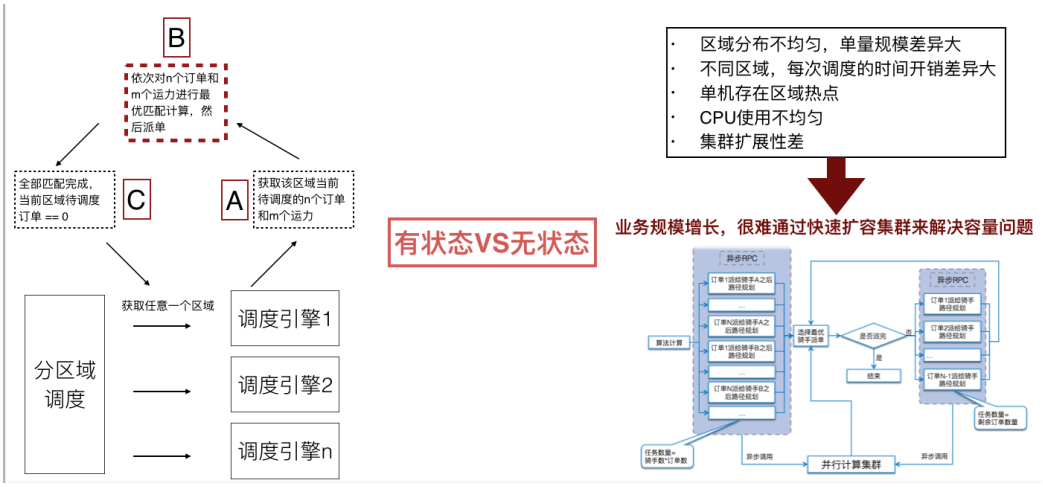
早期，美团按照业务领域划分成多个垂直服务架构；随着业务的发展，从可用性的角度考虑做了分层服务架构。后来，业务发展越发复杂，从运维、质量等多个角度考量后，逐步演进到微服务架构。这里主要遵循了两个原则：不宜过早的进入到微服务架构的设计中，好的架构是演进出来的不是提前设计出来的。

分布式系统实践



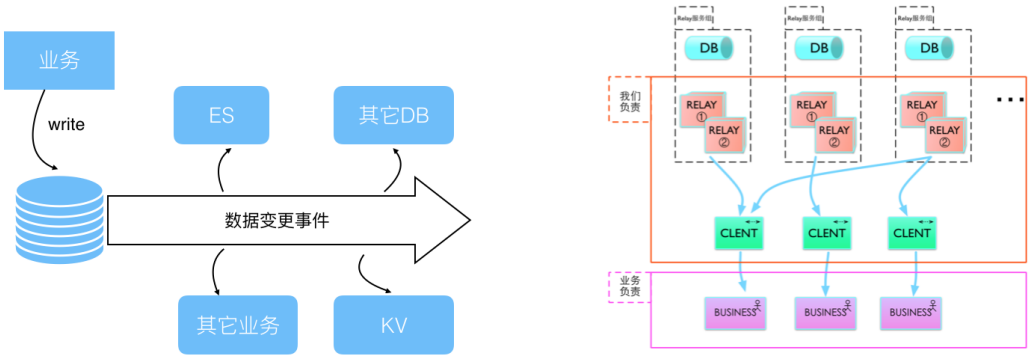
上图是比较典型的美团技术体系下的分布式系统结构：依托了美团公共组件和服务，完成了分区扩容、容灾和监控的能力。前端流量会通过HLB来分发和负载均衡；在分区内，服务与服务会通过OCTO进行通信，提供服务注册、自动发现、负载均衡、容错、灰度发布等等服务。当然也可以通过消息队列进行通信，例如Kafka、RabbitMQ。在存储层使用Zebra来访问分布式数据库进行读写操作。利用 CAT （美团开源的分布式监控系统）进行分布式业务及系统日志的采集、上报和监控。分布式缓存使用Squirrel+Cellar的组合。分布式任务调度则是通过Crane。

在实践过程还要解决几个问题，比较典型的是集群的扩展性，有状态的集群可扩展性相对较差，无法快速扩容机器，无法缓解流量压力。同时，也会出现节点热点的问题，包括资源不均匀、CPU使用不均匀等等。

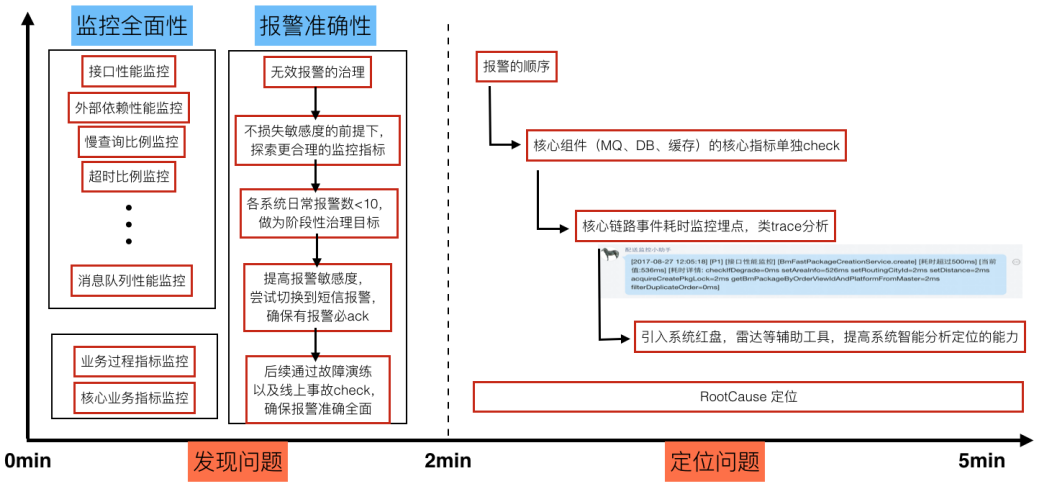


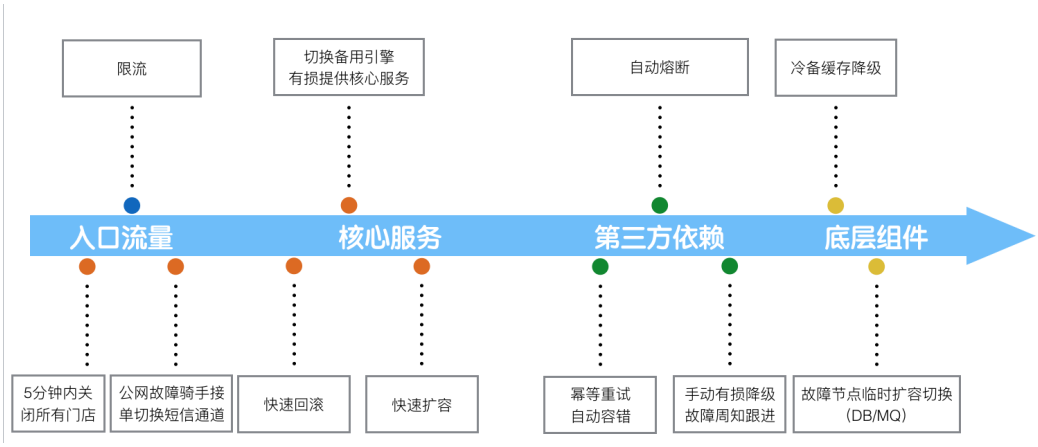
首先，配送后台技术团队通过架构升级，将有状态节点变成无状态节点，通过并行计算的能力，让小的业务节点去分担计算压力，以此实现快速扩容。

第二是要解决一致性的问题，对于既要写DB也要写缓存的场景，业务写缓存无法保障数据一致性，美团内部主要通过Databus来解决，Databus是一个高可用、低延时、高并发、保证数据一致性的数据库变更实时传输系统。通过Databus上游可以监控业务Binlog变更，通过管道将变更信息传递给ES和其他DB，或者是其他KV系统，利用Databus的高可用特性来保证数据最终是可以同步到其他系统中。



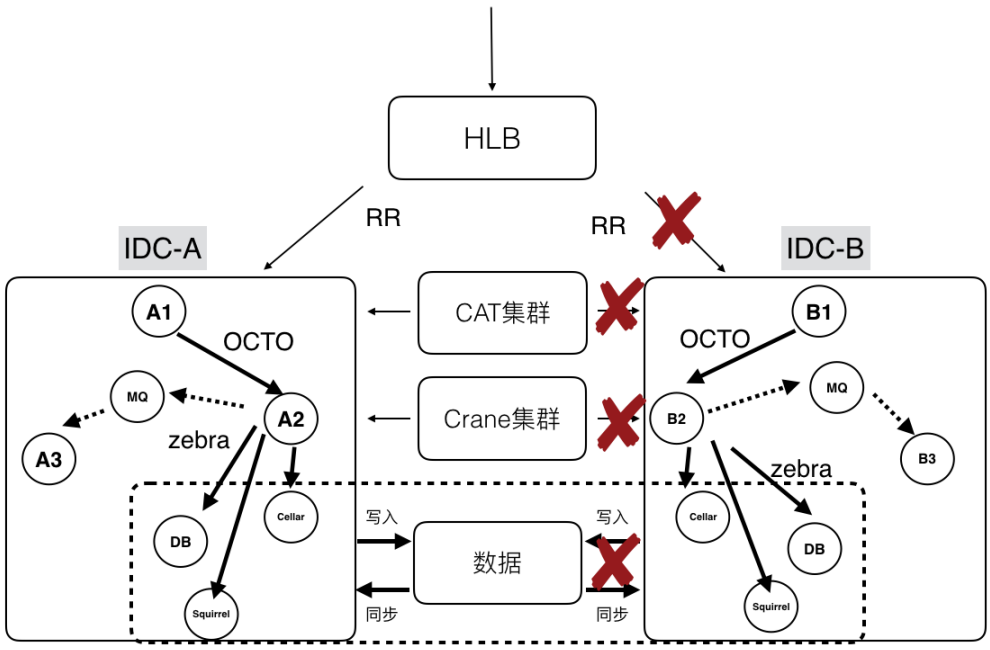
第三是我们一直在花精力解决的事情，就是保障集群高可用，主要从三个方面来入手，事前较多的是做全链路压测评，估峰值容量；周期性的集群健康性检查；随机故障演练（服务、机器、组件）。事中做异常报警（性能、业务指标、可用性）；快速的故障定位（单机故障、集群故障、IDC故障、组件异常、服务异常）；故障前后的系统变更收集。事后重点做系统回滚；扩容、限流、熔断、降级；核武器兜底。





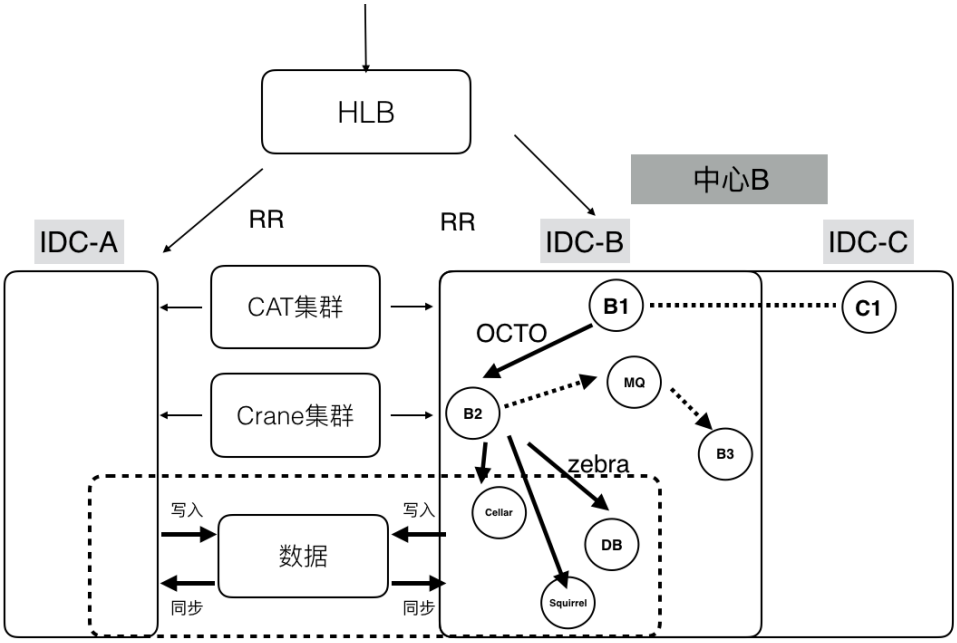
单IDC的快速部署&容灾

单IDC故障之后，入口服务做到故障识别，自动流量切换；单IDC的快速扩容，数据提前同步，服务提前部署，Ready之后打开入口流量；要求所有做数据同步、流量分发的服务，都具备自动故障检测、故障服务自动摘除；按照IDC为单位扩缩容的能力。



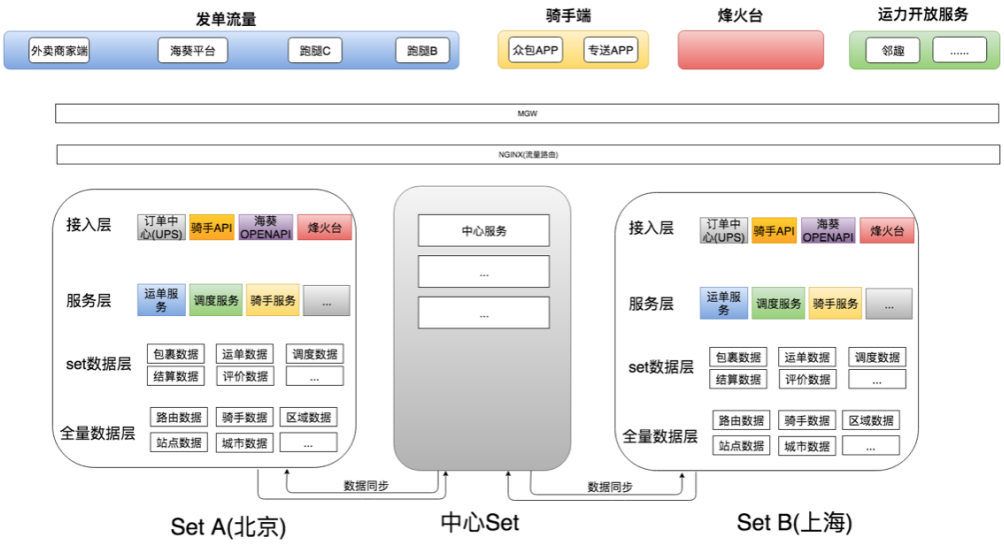
多中心尝试

美团IDC以分区为单位，存在资源满排，分区无法扩容。美团的方案是多个IDC组成虚拟中心，以中心为分区的单位；服务无差别的部署在中心内；中心容量不够，直接增加新的IDC来扩容容量。



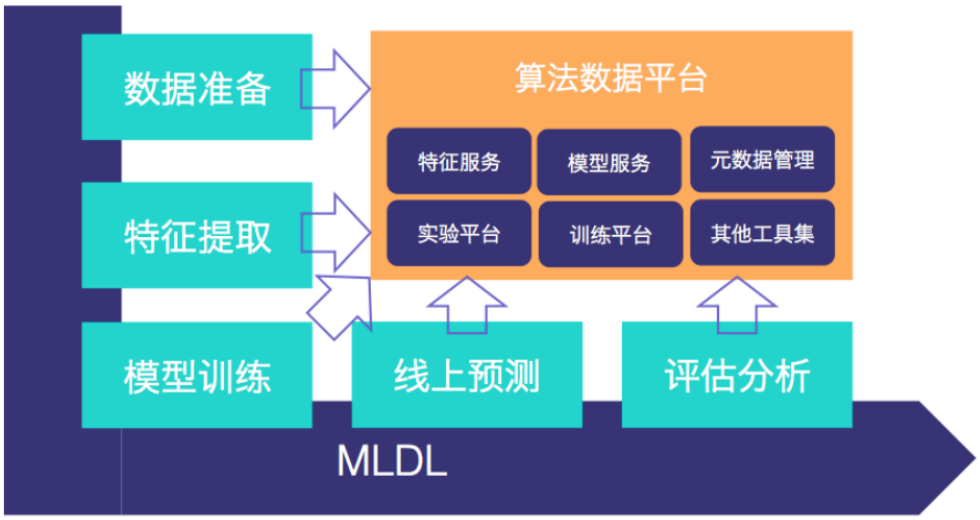
单元化尝试

相比多中心来说，单元化是进行分区容灾和扩容的更优方案。关于流量路由，美团主要是根据业务特点，采用区域或城市进行路由。数据同步上，异地会出现延迟状况。SET容灾上要保证同本地或异地SET出现问题时，可以快速把SET切换到其他SET上来承担流量。

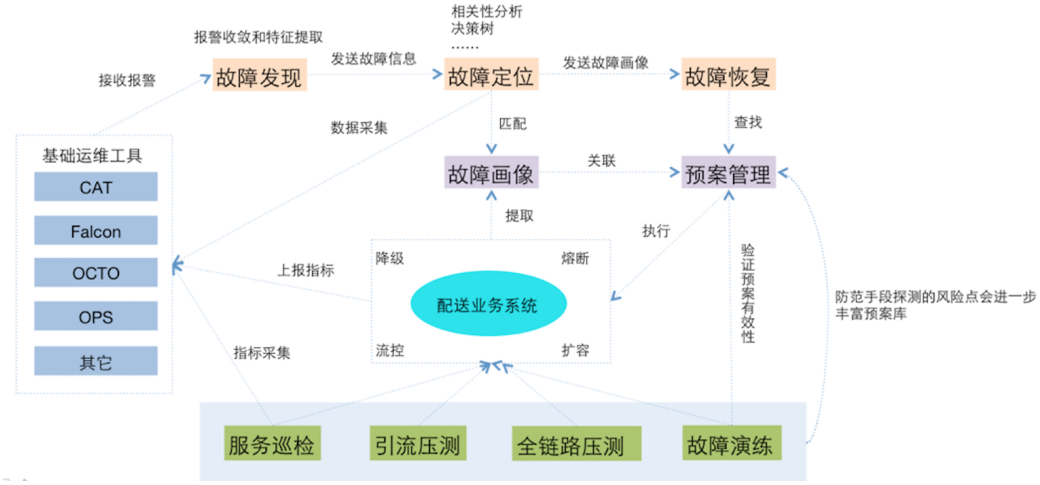


智能物流的核心技术能力和平台沉淀

机器学习平台，是一站式线下到线上的模型训练和算法应用平台。之所以构建这个平台，目的是要解决算法应用场景多，重复造轮子的矛盾问题，以及线上、线下数据质量不一致。如果流程不明确不连贯，会出现迭代效率低，特征、模型的应用上线部署出现数据质量等障碍问题。



JARVIS是一个以稳定性保障为目标的智能化业务运维AIOps平台。主要用于处理系统故障时报警源很多，会有大量的重复报警，有效信息很容易被淹没等各种问题。此外，过往小规模分布式集群的运维故障主要靠人和经验来分析和定位，效率低下，处理速度慢，每次故障处理得到的预期不稳定，在有效性和及时性方面无法保证。所以需要AIOps平台来解决这些问题。



未来的挑战

经过复盘和Review之后，我们发现未来的挑战很大，微服务不再“微”了，业务复杂度提升之后，服务就会变得膨胀。其次，网状结构的服务集群，任何轻微的延迟，都可能导致的网络放大效应。另外复杂的服务拓扑，如何做到故障的快速定位和处理，这也是AIOps需要重点解决的难题。最后，就是单元化之后，从集群为单位的运维到以单元为单位的运维，业给美团业务部署能力带来很大的挑战。

作者简介

- 宋斌，美团资深技术专家，长期参与分布式系统架构、高并发系统稳定性保障相关工作。目前担任即时物流团队后台技术负责人。2013年加入美团，参与过美团外卖C端、即时物流体系从零搭建。现在带领团队负责调度、清结算、LBS、定价等业务系统、算法数据平台、稳定性保障平台等技术平台的研发和运维。最近重点关注AIOps方向，探索在高并发、分布式系统架构下，如何更好的做好系统稳定性保障。

招聘信息

美团配送技术团队诚招LBS领域、调度履约平台、结算平台、AIOps方向、机器学习平台、算法工程方向的资深技术专家和架构师。共建全行业最大的单一即时配送网络和平台，共同面对复杂业务和高并发流量的挑战，迎接配送业务全面智能化的时代。欢迎感兴趣的同学投送简历至 songbin@meituan.com，chencheng13@meituan.com。

美团点评运营数据产品化应用与实践

作者: 吉喆

背景

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

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

因此团队提出将运营专题数据产品化，首先分析面临的一些问题和挑战。

挑战

① 服务业务能力

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

② 存储、计算、研发成本

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

③ 数据标准不统一

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

④ 业务分析响应能力

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

针对以上的问题和挑战，开始制定建设方案。

方案

首先，构建了一个针对境内旅游运营侧全域的公共底层数据，将不同平台促销系统的数据按业务整合到一起，同时划分不同活动主题，按事件再向上聚合，做专题的数据支撑，统一数据出口。然后通过多维预计

算引擎对事实数据进行预计算，构建数仓与应用的管道，从而节省计算成本，并且提升了数据互通和消费的效率，最后建设统一的数据服务中台，搭配不同端的Web应用。通过丰富的可视化效果，及多样的分析对比操作，快速、全面地支撑运营业务。

以下为整个产品的功能模块图：

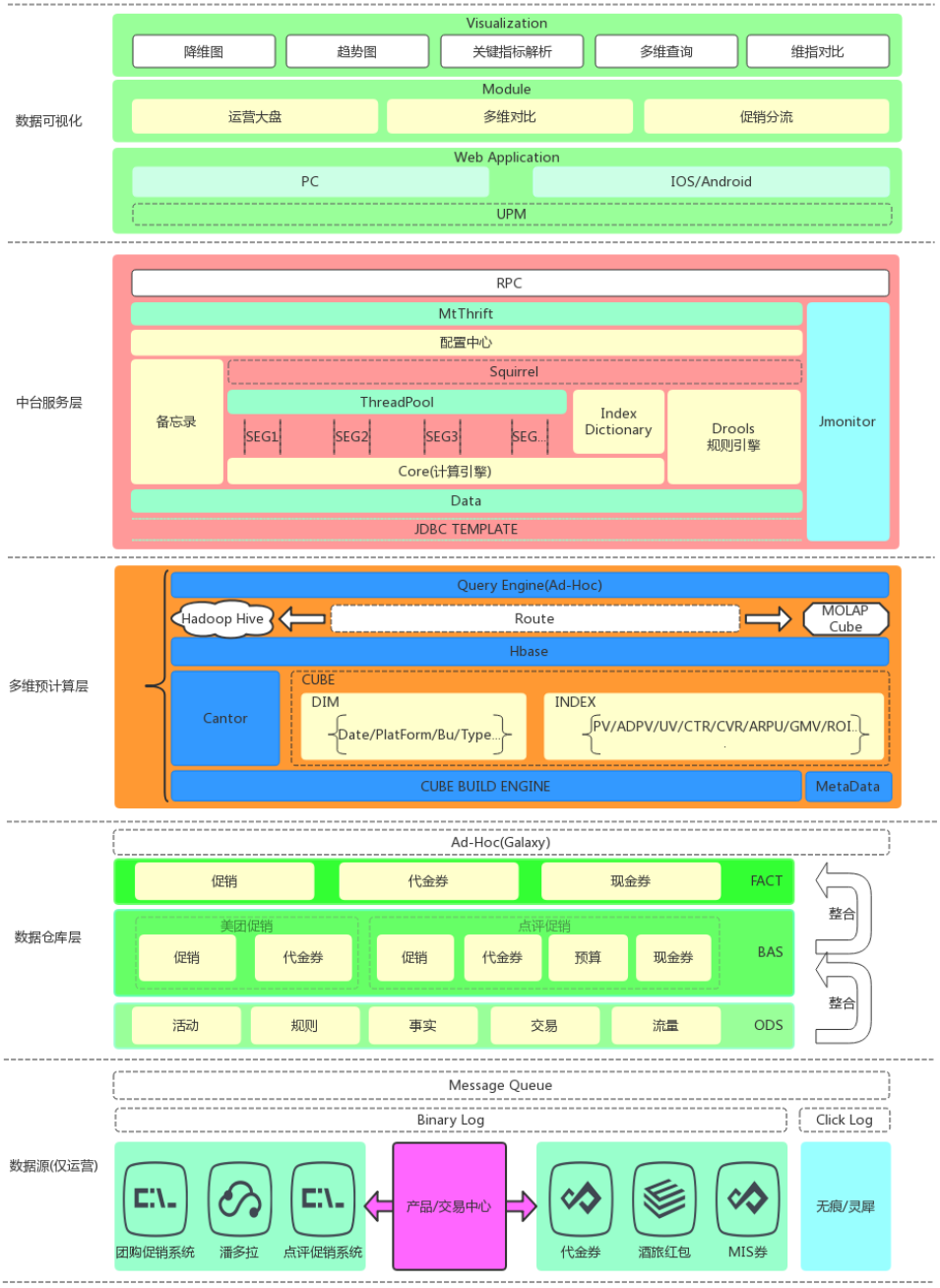


图1 运营专题整体功能模块图

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

数据仓库层

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

领域内常见的建模方法

① 3NF模型

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

3NF模型最基本的要素是实体、属性和关系：

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

② 维度模型

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

维度建模最基本的要素是事实和维度：

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

③ DV模型（DataVault）

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

④ Anchor模型

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

运营专题数据如何构建

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

① 数据规范制定

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

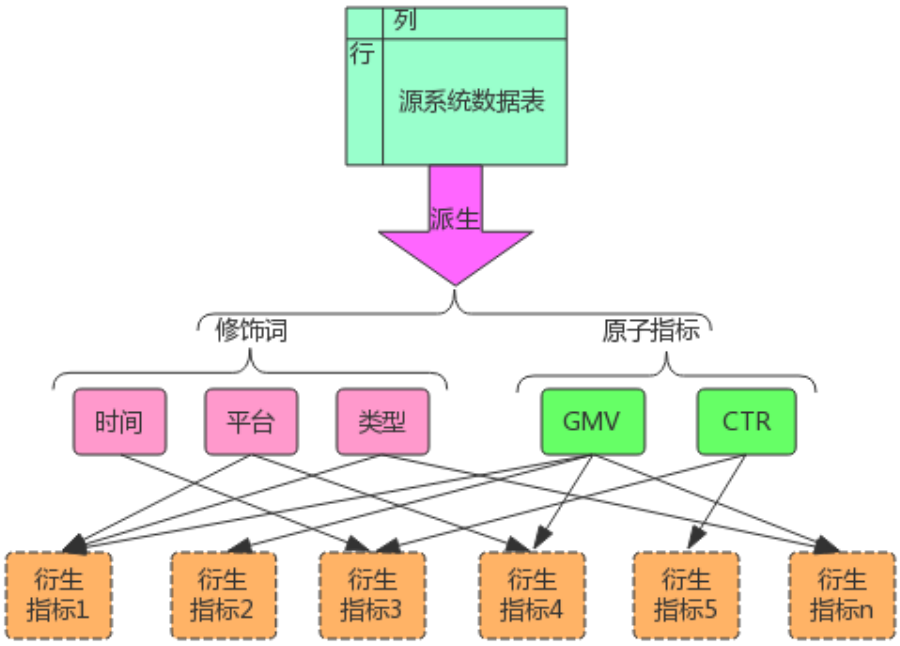


图2 数据规范抽象示意图

② 数仓模型架构

将数据分为ODS层、BAS层、FACT层、TOPIC层。

ODS层主要功能

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

BAS层主要功能

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

FACT层主要功能

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

TOPIC层主要功能

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

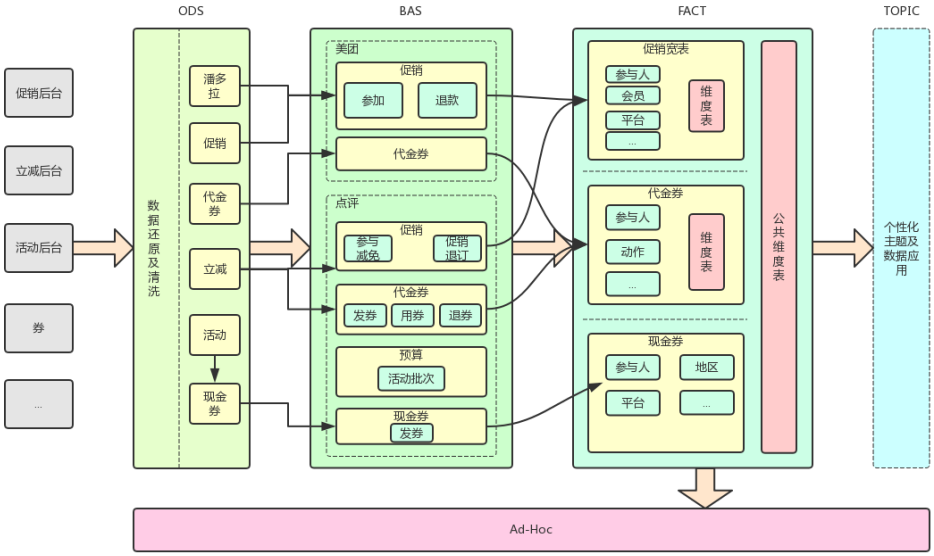


图3 数据仓库模型图

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

多维预计算层

预计算层是连接数据和应用之间的管道，是应用层垂直模块的专项支持。它是在Fact层数据之上的预聚合，强依赖于数仓模型中事实和维度的构建以及预关联。预计算采用Kylin引擎构建Cube聚合组，来解决取数门槛和数据处理耗时等问题，同是提供多维分析的能力，不但提供了新的Ad-Hoc(Query Engine)平台，在提高查询响应的同时，又能为产品带来更流畅的交互，增强用户体验。例如：创建一个交易数据cube，它包含日期（datakey）、用户（userid）、付款方式（paytype）、购买城市（city）。为满足不同消费方式在不同城市的应用情况和查看用户在不同城市的消费行为，建立以下两个聚合组，包含的维度和方式如图所示：

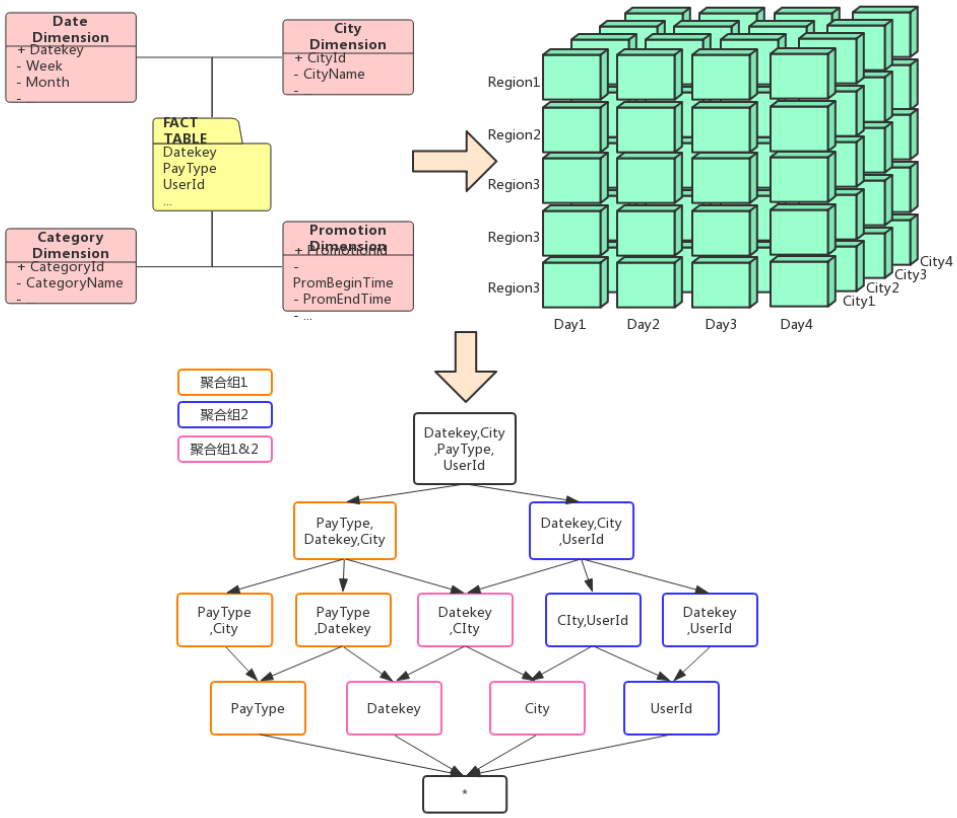


图4 构建cube示例图

中台服务层

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

- 服务与预计算数据同步，数据模型的修改只影响到预计算层，同时服务层还可以完全感知预计算数据的变化，不需要对服务做开发调整，实现数据变更的同步响应；
- 服务与端解耦，针对不同端产品提供统一数据服务，避免重复开发，同时产品的迭代升级与服务层隔离，应对多变的业务发展和增长；
- 服务扩展能力增强，支持服务的横向扩展，不影响正常业务的同时提高服务能力，同时在该层实现可抽象通用操作以及规范管理。

总体架构

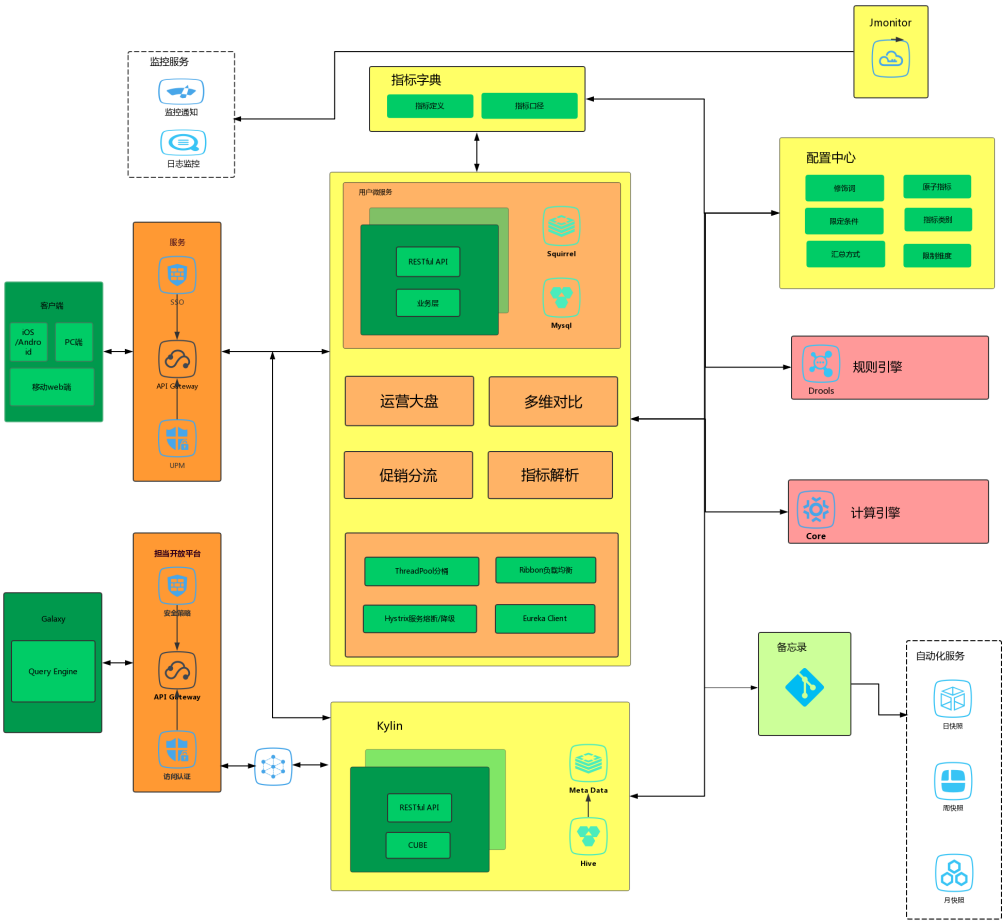


图5 运营专题产品架构图

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

配置中心

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

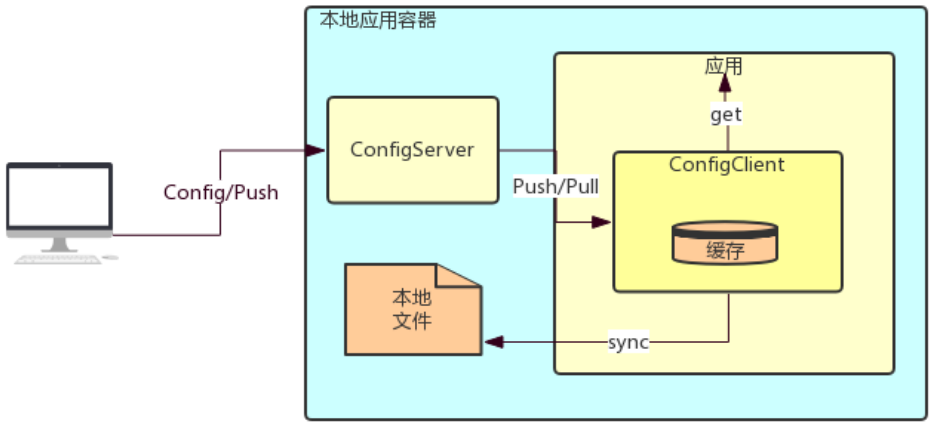


图6 配置中心示意图

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

指标字典

公司中的很多运营部门指标定义不清晰或不尽相同，但叫法相同（文案），又或者叫法相同指标口径不同，出现一些对指标的理解不一致，含义不清等问题。基于指标字典，不但是指标命名的规范和明确，也是统一计算口径的落地，接入规则引擎后生成关联衍生指标，即可自助完成查询和分析。可见，指标字典的建立，是数据服务平台的基础。

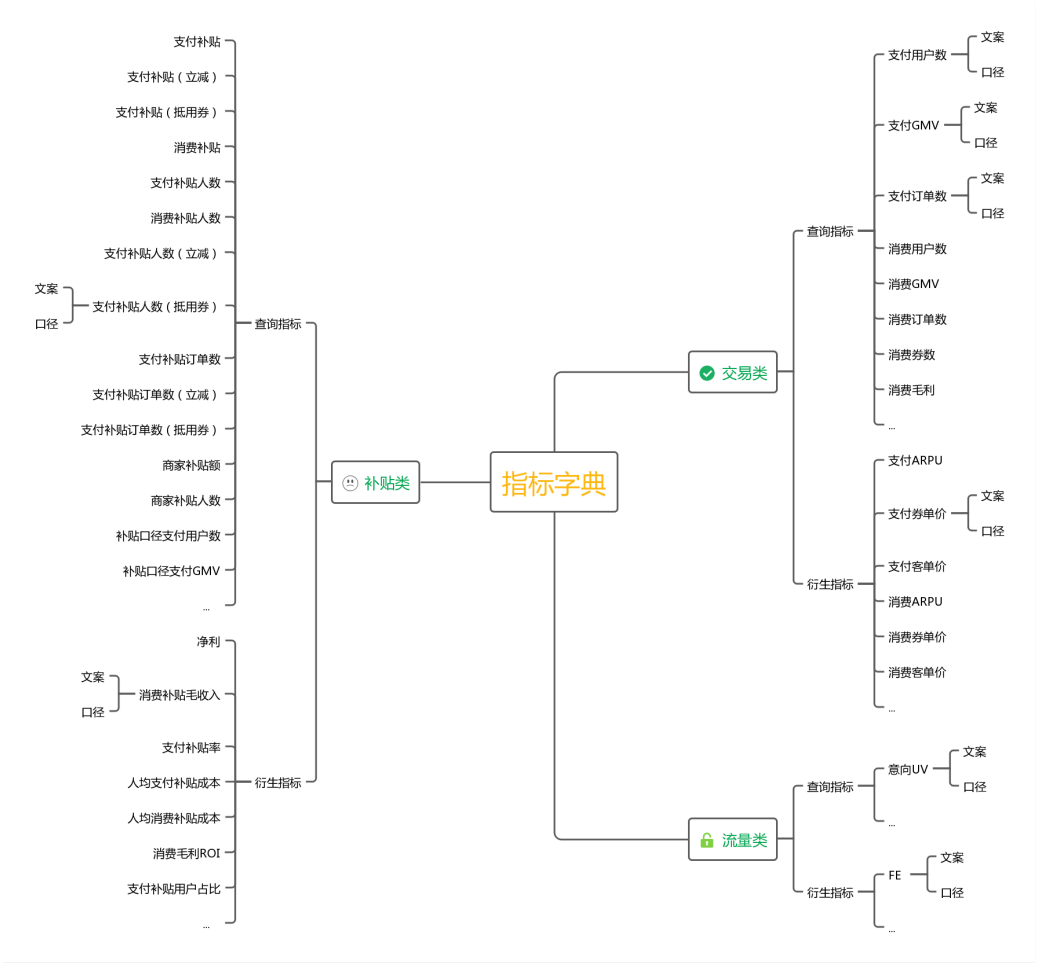


图7 指标字典思维导图

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

规则引擎

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

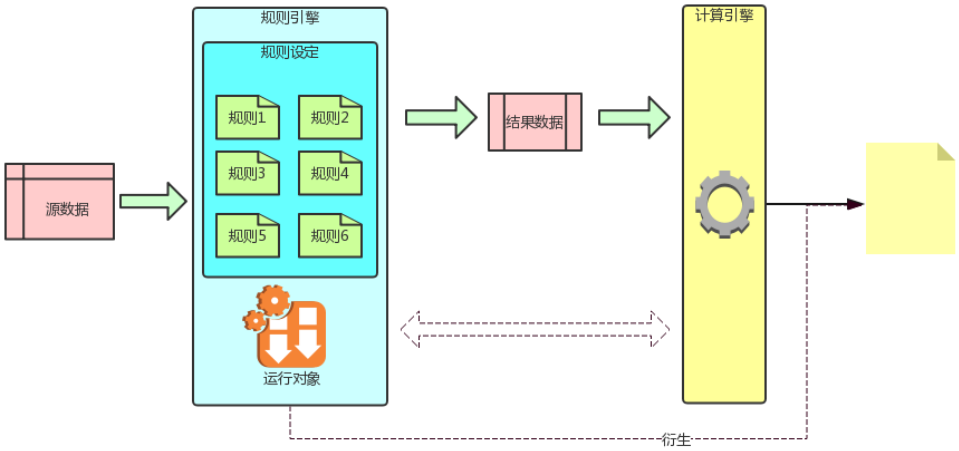


图8 规则引擎示意图

① 数据准备规则

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

② 数据计算规则

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

计算引擎

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

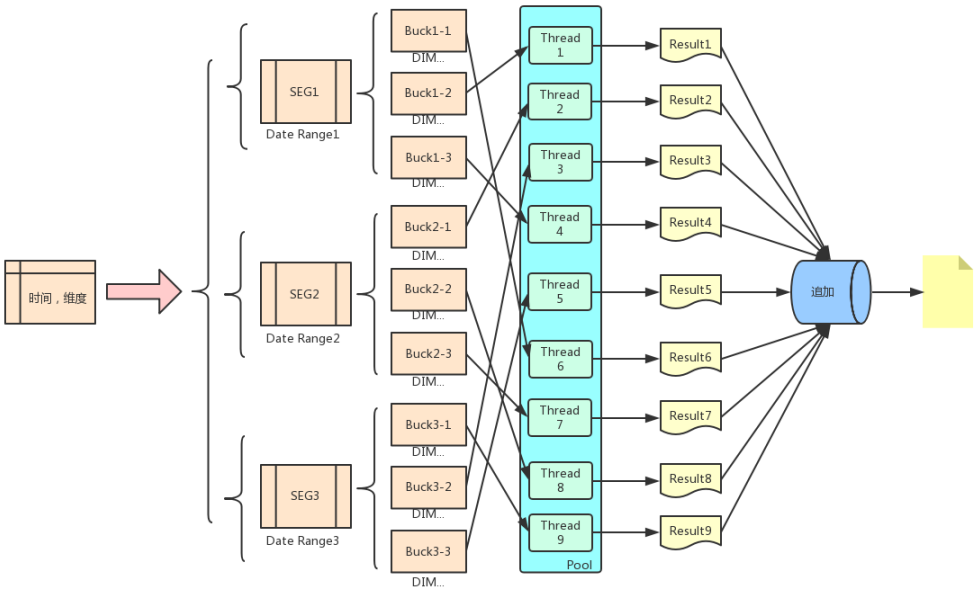


图9 计算引擎示意图

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

备忘录

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

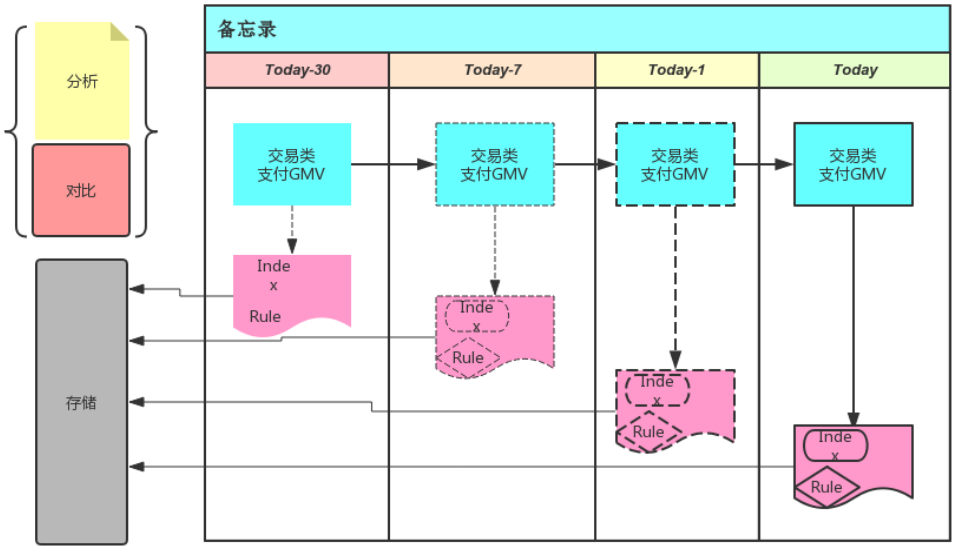


图10 备忘录示意图

数据可视化

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

趋势对比

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

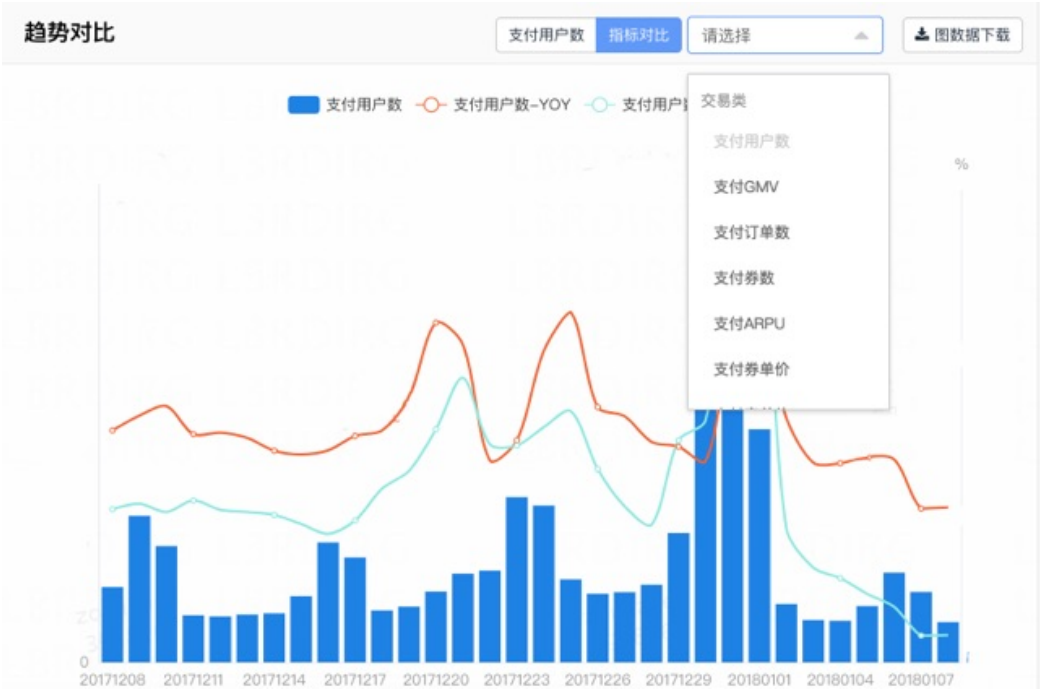


图11 产品趋势对比图

降维操作

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

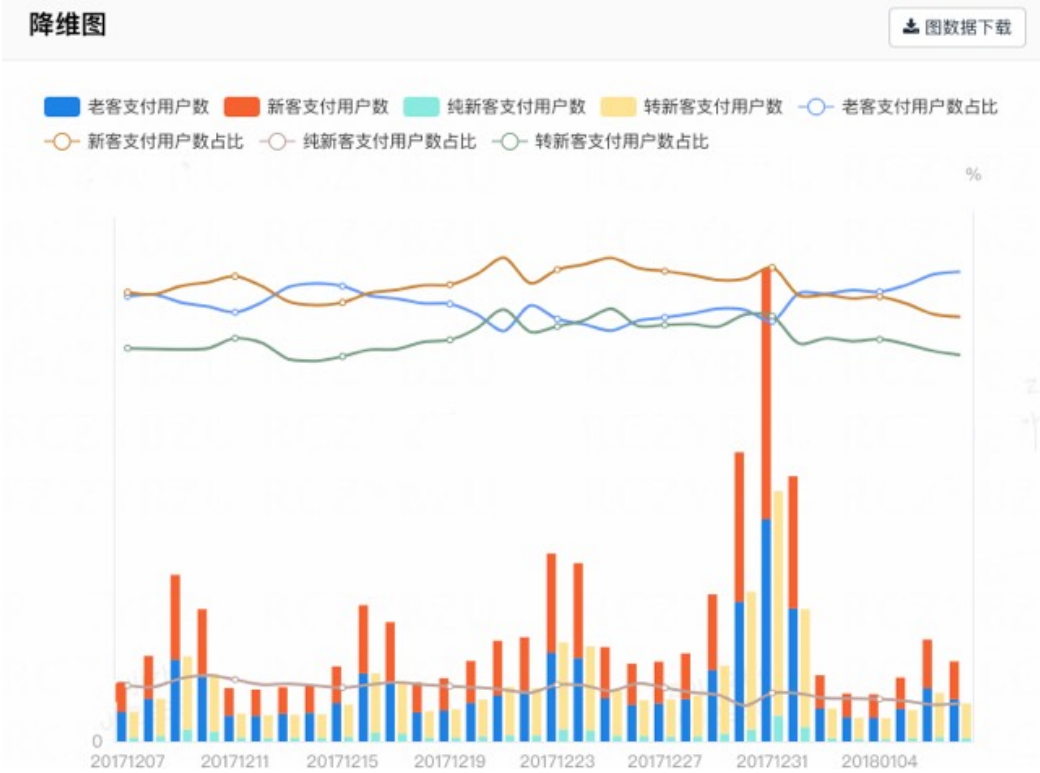


图12 产品降维操作图

指标对比

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

图13 产品指标对比图

多维查询

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

图14 产品多维查询图

总结

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

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

作者简介

- 吉喆，美团点评系统开发工程师，曾就职于新浪，美团，阿里巴巴从事系统开发及数据开发工作，2017年加入美团点评，负责数据仓库建设和产品开发相关工作。

招聘信息

团队长期招聘数据仓库、数据开发工程师，欢迎对大数据领域感兴趣的同学投递简历到yangguang09#meituan.com，期待您的加入。

美团服务体验平台对接业务数据的最佳实践–海盗中间件

作者: 王彬 陈胜

背景

移动互联网时代，用户体验为王。美团服务体验平台希望能够帮助客户解决在选、购、用美团产品过程中遇到的各种问题，真正做到“以客户为中心”，为客户排忧解难。但服务体验平台内部只维护客户的客诉数据，为了精准地预判和更好地解决客户遇到的问题，系统必须依赖业务部门提供的一些业务数据，包括但不限于订单数据、退款数据、产品数据等等。 本文会着重讲一下在整个系统交互过程中遇到的一些问题，然后分享一下在实践中探索出来的经验和方法论，希望能够给大家带来一些启发。

问题

对接场景广而杂

首先，需要接入服务体验平台服务（包括直接面向用户的C端服务、面向客服的工单服务等等）的业务方非常多且杂，而且在不断拓展。美团有非常多的业务线，比如外卖、酒店、旅游、打车、交通、到店餐饮、到店综合、猫眼等等。其中部分业务又延展出多条子业务线，比如大交通部门包含火车票、汽车票、国内机票、国际机票、船票等等。具体到每一条子业务线的每一个业务场景，客户都有可能会遇到问题。对于这些场景，服务体验平台服务都需要调用对应的业务数据接口，来帮助用户自助或者客服协助解决这些问题。就美团现有的业务而言，这样的场景数量会达到万级。而且业务形态在不断迭代，还会有更多的场景被挖掘出来，这些都需要持续对接更多的业务数据来进行支撑。

接入场景定制化要求高

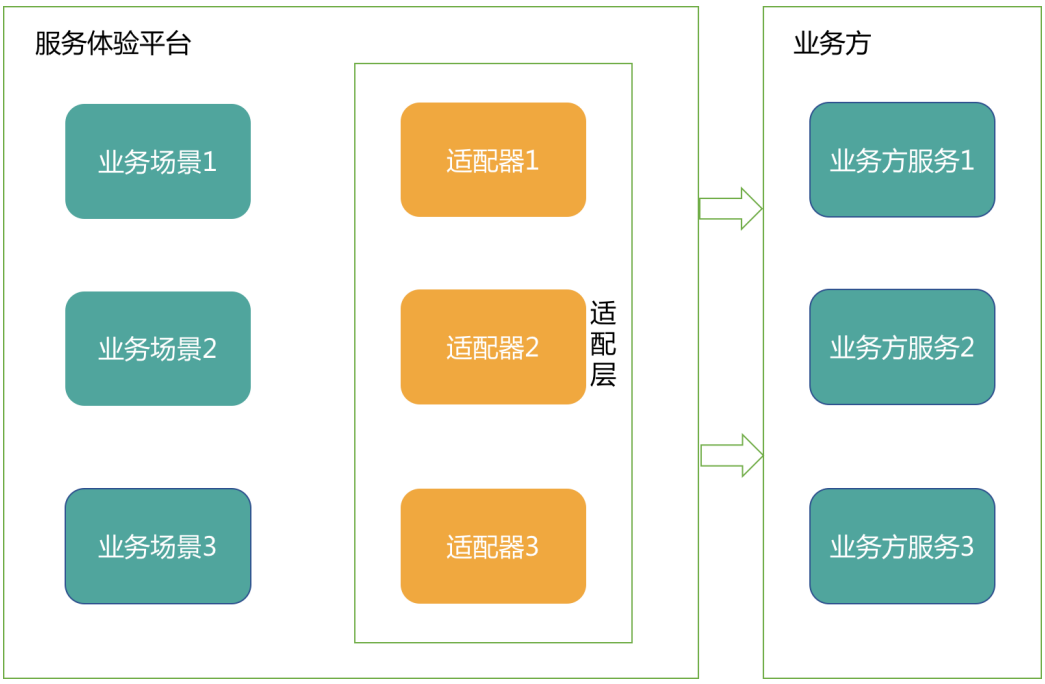
其次，接入服务体验平台服务的业务方定制化要求很高。因为业务场景的差异化非常大，不同的接入方都希望能够定制特殊复杂逻辑，需要服务体验平台提供的服务解决方案与业务深度耦合。这就需要服务体验平台侧对接入方业务逻辑和数据接口深入了解，并对这些业务数据进行组装，针对每个场景进行定制开发。

方案

早期方案

为了解决上述问题，初期在做系统设计时候，考虑业务方多是既有系统，所以服务体验平台服务趋向平台化设计，并引入了适配层。服务体验平台内部对所有的业务数据和逻辑进行统一抽象，对内标准化接口，屏蔽掉业务逻辑和接口的差异。所有的定制化逻辑都在适配层中封装。但这需要客服侧RD对所有的场景去编写适配器代码，将从一个或者多个业务部门接口中拿到的业务数据，转成内部实际场景需要的数据。

其系统交互如下图所示：



缺点

虽然上述系统设计能满足业务上的要求，但是存在两个比较明显的缺点。

编码工作量繁重

如上图所示，每个业务场景都需要编写适配器来满足需求，如果依赖的外部接口比较少，场景也比较单一，按照上述方案实施还可以接受。但业务接入非常多且杂，给客服侧RD带来了非常繁重的工作量，包括适配器编写以及后续维护过程中对下游业务接口的持续跟踪和监控。

客服侧RD需要深入了解业务方逻辑

另外，由于客服侧RD对于业务模型的不熟悉，解析业务模型然后组装最终展示给客户的数据，需要比业务方RD花更多的时间来梳理和实现，并且花费更多的时间来验证正确性。比如下面是一个真实的组装业务接口并对业务数据进行处理的案例：

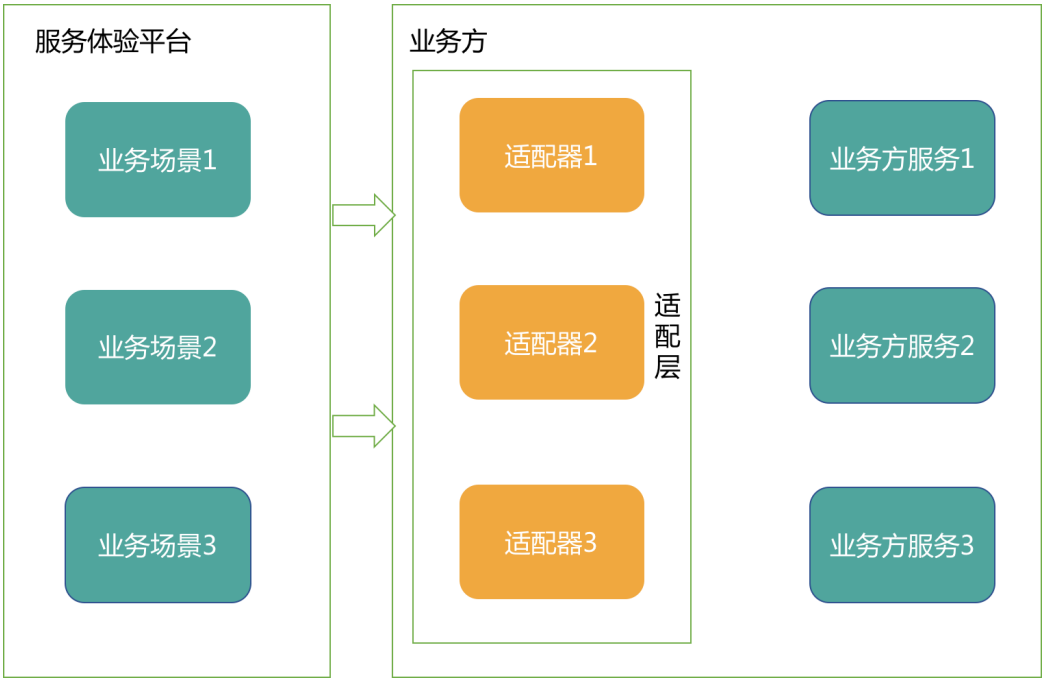
```
public class TicketAdapterServiceImpl implements OrderAdapterService {

    @Resource(name = "tradeQueryClient")
    private TradeTicketQueryClient tradeTicketQueryClient;
    @Resource
    private ColumbusTicketService columbusTicketService;

    /**
    * 根据订单ID获取门票相关的订单数据、门票数据、退款数据等
    **/
    @Override
    public OrderInfoDTO handle(OrderRequestDTO orderRequestDTO) {
        List<ITradeTicketQueryService.TradeDetailField> tradeDetailFieldList = new ArrayList<ITradeTicketQueryService.TradeDe
tailField>();
        tradeDetailFieldList.add(ITradeTicketQueryService.TradeDetailField.ORDER);
        tradeDetailFieldList.add(ITradeTicketQueryService.TradeDetailField.TICKET);
        tradeDetailFieldList.add(ITradeTicketQueryService.TradeDetailField.REFUND_REQUEST);
        try {
            //通过接口A得到部分订单数据、门票数据和退款数据
            RichOrderDetail richOrderDetail = tradeTicketQueryClient.getRichOrderDetailById(orderRequestDTO.getOrderId(), tra
deDetailFieldList);
            if (richOrderDetail == null) {
                return null;
            }
            if (richOrderDetail.getOrderDetail() == null) {
                return null;
            }
```

```
            OrderDetail orderDetail = richOrderDetail.getOrderDetail();
            RefundDetail refundDetail = richOrderDetail.getRefundDetail();
            OrderInfoDTO orderInfoDTO = new OrderInfoDTO();

            //解析和处理接口A返回的字段，得到客服侧场景真正需要的数据
            orderInfoDTO.put("dealId", orderDetail.getMtDealId());
            orderInfoDTO.put(DomesticTicketField.VOUCHER_CODE.getValue(), getVoucherCode(richOrderDetail));
            orderInfoDTO.put(DomesticTicketField.REFUND_CHECK_DUE.getValue(), getRefundCheckDueDate(richOrderDetail));
            orderInfoDTO.put(DomesticTicketField.REFUND_RECEIVED_DUE.getValue(), getRefundReceivedDueDate(richOrderDetail));

            //根据接口B获取另外一些订单数据、门票详情数据、退款数据
            ColumbusTicketDTO columbusTicketDTO = columbusTicketService.getByDealId((int) richOrderDetail.getOrderDetail().ge
tMtDealId());
            if (columbusTicketDTO == null) {
                return orderInfoDTO;
            }
            //解析和处理接口B返回的字段，得到客服侧场景真正需要的数据
            orderInfoDTO.put(DomesticTicketField.REFUND_INFO.getValue(), columbusTicketDTO.getRefundInfo());
            orderInfoDTO.put(DomesticTicketField.USE_METHODS.getValue(), columbusTicketDTO.getUseMethods());
            orderInfoDTO.put(DomesticTicketField.BOOK_INFO.getValue(), columbusTicketDTO.getBookInfo());
            orderInfoDTO.put(DomesticTicketField.INTO_METHOD.getValue(), columbusTicketDTO.getIntoMethod());

            return orderInfoDTO;
        } catch (TException e) {
            Cat.logError("查询不到对应的订单详情", e);
            return null;
        }
    }
}
```

探索

将适配层交由业务方实现

为了克服早期方案的两个缺点，最初，我们希望能够把场景数据的准备和业务模型的解析工作，都交给对业务比较熟悉的团队来处理，即将适配层交由业务方来实现。



这样做的话优势和劣势也比较明显。

优势

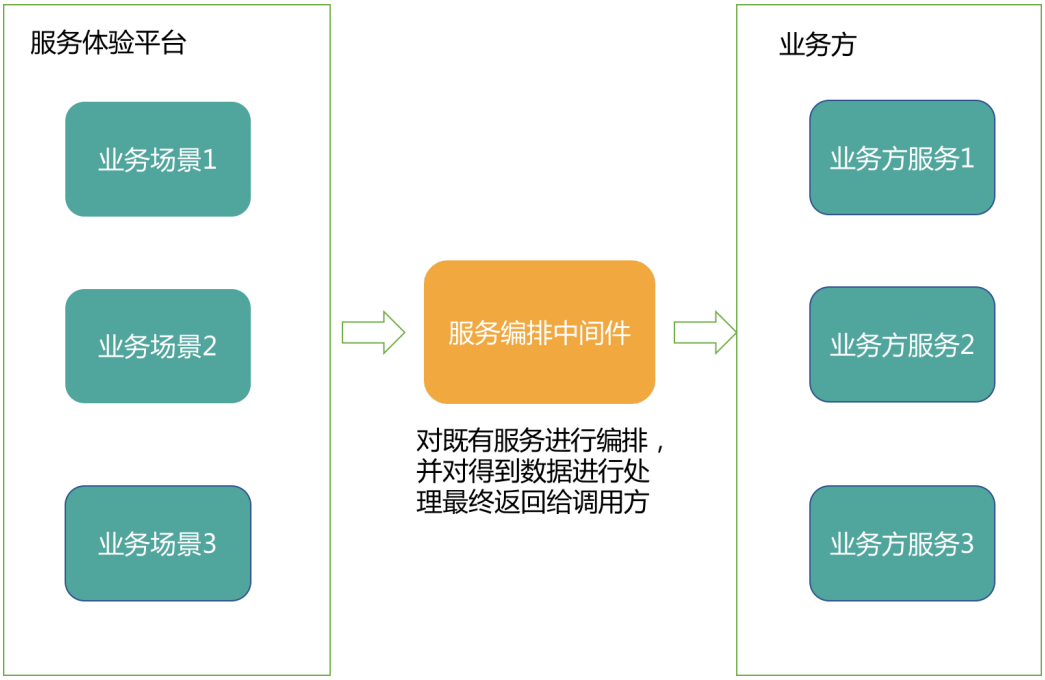
客服这边关注自己的领域服务就好，做好平台化，数据提供都交给业务团队，解放了客服侧RD。

劣势

但对业务方来说带来了比较大的工作量，业务方既有服务的复用性很低，对客服侧每一个需要数据的场景，都要重新封装新的服务。

更好的解决方案?

这个时候我们思考：是否可以既能让业务方解析自己的业务数据，又能够尽量利用既有服务呢？我们考虑把既有服务的组装过程以及模型的转换都让一个服务编排的中间件来实现。但是使用这个中间件有一个前提，就是业务方提供出来的既有服务必须支持泛化调用，避免调用方直接依赖服务方客户端（文章下一个小节也会补充下对于泛化调用的解释）。其交互模型如下图所示：



结果–海盗中间件

简介

什么是海盗?

海盗就是一个用来对支持泛化调用（上述所说）的服务进行编排，然后获取预期结果的一个中间件。使用该中间件调用方可以根据场景来对目标服务进行编排，按需调用。

何为泛化调用?

通常服务提供方提供的服务都会有自己的接口协议，比如一个获取订单数据的服务：

```
package com.dianping.demo;
public interface DemoService{
      OrderDTO  getById(String orderId);
}
```

而调用方调用该服务需要引入该接口协议，即依赖该服务提供的JAR包。如果调用方需要集成多方数据，那就需要依赖非常多的API，同时服务方接口升级客户端也需要随之进行升级。而泛化调用就可以解决这个问题，通过泛化调用客户端可以在服务方没有提供接口协议和不依赖服务方API的情况下对服务进行调用，通过类似 `GenericService` 这样一个接口来处理所有的服务请求。

如下是一个泛化调用的Demo：

```
    public class DemoInvoke{
        public void genericInvoke(){
             /** 调用方配置  **/
             InvokerConfig<GenericService> invokerConfig = new InvokerConfig("com.dianping.demo.DemoService", com.dianping.pi
geon.remoting.common.service.GenericService.class);
             invokerConfig.setTimeout(1000);
             invokerConfig.setGeneric(GenericType.JSON.getName());
             invokerConfig.setCallType("sync");

             /** 泛化调用 **/
             final GenericService genericService = ServiceFactory.getService(invokerConfig);
             List<String> paramTypes = new ArrayList<String>();
             paramTypes.add("java.lang.String");
             List<String> paramValues = new ArrayList<String>();
             paramValues.add("0000000001");
             String result = genericService.$invoke("getById", paramTypes, paramValues);
        }
    }
```

有了这个泛化调用的前提，我们就可以重点去思考如何对服务进行编排，然后对取得的结果进行处理了。

DSL设计

首先重新梳理一下海盗的设计目标：

> - **对既有服务进行编排调用**
> - **对获取的数据进行处理**

而为了实现服务编排，需要定义一个数据结构来描述服务之间的依赖关系、调用顺序、调用服务的入参和出参等等。之后对获取的结果进行处理，也需要在这个数据结构中具体描述对什么样的数据进行怎么样的处理等等。

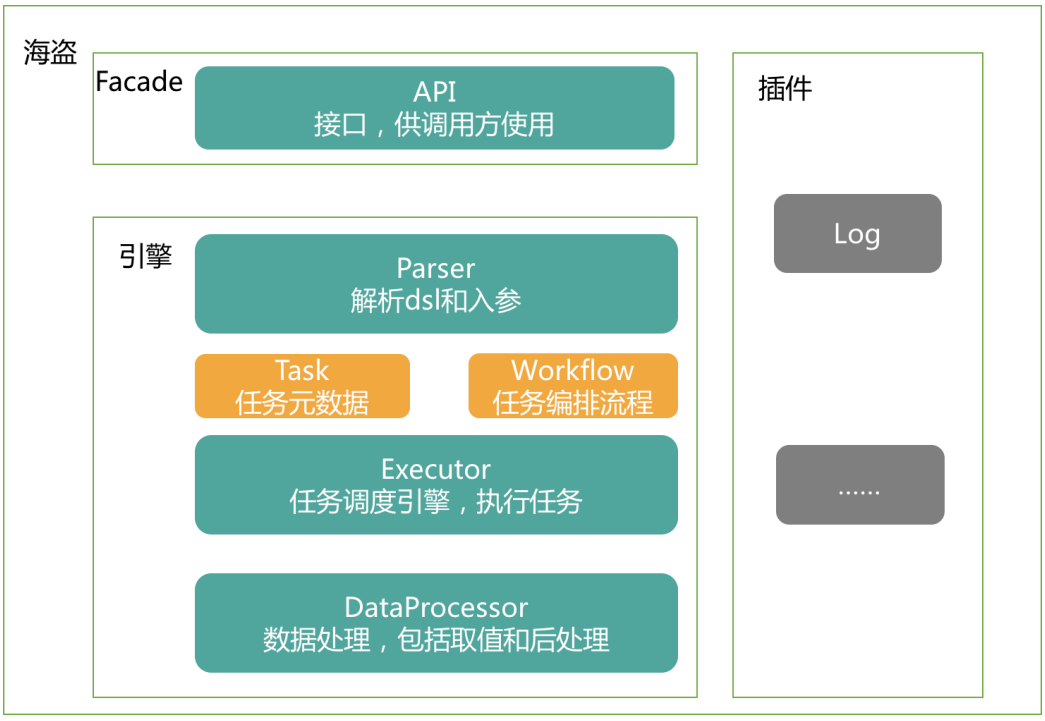
所以我们需要定义一套DSL（领域特定语言）来描述整个服务编排的蓝图，其语法如下：

```
{
    //定义好需要调用的接口以及接口之间的依赖关系，一个接口调用即为一个task
    "tasks": [
        //第一个task
        {
            "url": "http://helloWorld.test.hello",      //url 为pigeon发布的远程服务地址：
            "alias": "d1",    //别名，结果取值的时候可以通过别名引用
            "taskType": "PigeonGeneric",  //task的类别一般可以设置为PigeonGeneric，默认是pigeonAgent方式。
            "method": "getByDoubleRequest", //要调用的pigeon接口的方法名
            "timeout": 3000,  //task的超时时间
            "inputs": {       //入参情况，多个入参通过key:value的结构书写，key的类别通过下面的inputsExtra定义。
                "helloWorld": {
                    "name": "csophys",     //可以通过#orderId，从上下文中获取值，可以通过$d1.orderId的形式从其他的task中获取值
                    "sex": "boy"
                },
                "name": "winnie"
            },
           "inputsExtra": {     //入参key的类别定义
                "helloWorld": "com.dianping.csc.pirate.remoting.pigeon.pigeon_generic_demo_service.HelloWorld",
                "name": "java.lang.String"
            }
        },
```

```
        //另一个task
        {
            "url": "http://helloWorld.test.hello",
            "alias": "d2",
            "taskType": "PigeonGeneric",
            "method": "getByDoubleRequest",
            "inputsExtra": {
                "helloWorld": "com.dianping.csc.pirate.remoting.pigeon.pigeon_generic_demo_service.HelloWorld",
                "name": "java.lang.String"
            },
            "timeout": 3000,
            "inputs": {
                "helloWorld": {
                    "name": "csophys",
                    "sex": "boy"
                },
                "name": "winnie"
            }
        }
    ],
    "name": "pigeonGenericUnitDemo",  //DSL的名称定义，暂时没有特别含义
    "description": "pigeon泛型调用测试",  //DSL的描述
    "outputs": {            //定义好最后输出的数据模型
        "d1name": "$d1.name",
        "languages": "$d2.languages",
        "language1": "$d2.languages[0]",
        "name": "csophys"
    }
}
```

架构设计

有了DSL来描述整个编排蓝图之后，海盗自然要对该DSL进行解析，然后对服务进行具体调用。其整体架构如下所示：



其中涉及到几个重点概念：

- Facade：对外提供统一接口，供客户端调用。
- Parser：对于输入的DSL进行解析，解析成内部流转的数据结构，同时得到所有的task，并且构建task调用逻辑树。
- Executor：真实发起调用的模块，目前支持平台内部的Pigeon和MTThrift调用方式，同时对HTTP等其他协议有良好的扩展性。

- DataProcessor：数据后处理。这边会把所有接口拿到的数据转换层客服场景这边需要的数据，并且通过设计的一些内部函数，可以支持一些如数据半脱敏等功能。
- 组件插件化：对日志等功能实现可插拔，调用方可以自定义这些组件，即插即用。

主要Feature

海盗具有如下主要特点：

- 采用去中心化的设计思路，引擎集成在SDK中。方案通用化，每一个需要业务数据的场景都可以通过海盗直接调用数据提供方。
- 服务编排支持并行和串行调用，使用方可以根据实际场景自己构造服务调用树。通过DSL的方式把之前硬编码组装的逻辑实现了配置化，然后通过海盗引擎把能并行调用的服务都执行了并行调用，数据使用方不用再自己处理性能优化。
- 使用JSON DSL 描述整个工作蓝图，简单易学。
- 支持JSONPath语法对服务返回的结果进行取值。
- 支持内置函数和自定义指令（语法参考ftl）对取到的元数据进行处理，得到需要的最终结果。
- 编排服务树可视化。
- 目前集团内部RPC中间件包括Pigeon、MTThrift，已进行了泛化调用支持，可以通过海盗实现Pigeon服务和MTThrift的服务编排。不需要限制业务团队的服务提供方式，但需要升级中间件版本。这里特别感谢服务治理团队的大力支持。

Tutorial

场景：需要根据订单ID查询订单状态和支付状态，但目前没有现成的接口支持该功能，但有两个既有接口分别是：

- 接口1：根据订单ID，获取到订单状态和支付流水号
- 接口2：根据支付流水号获取支付状态

那我们可以对这两个接口进行编排，编写DSL如下：

```
{
  "tasks": [
    {
      "url": "http://test.service",
      "alias": "d1",
      "taskType": "PigeonGeneric",
      "method": "getByOrderId",
      "timeout": 3000,
      "inputs": {
        "orderId": "#orderId"
      },
      "inputsExtra": {
        "name": "java.lang.String"
      }
    },
    {
      "url": "http://test.service",
      "alias": "d2",
      "taskType": "PigeonGeneric",
      "method": "getPayStatus",
      "timeout": 3000,
      "inputs": {
        "paySerialNo": "$d1.paySerialNo"
      },
      "inputsExtra": {
        "time": "java.lang.String"
      }
    }
  ],
```

```
  "name": "test",
  "description": "组装上述接口获取订单状态和支付状态",
  "outputs": {
    "orderStatus": "$d1.orderStatus",
    "payStatus": "$d2.payStatus"
  }
}
```

然后客户端进行调用：

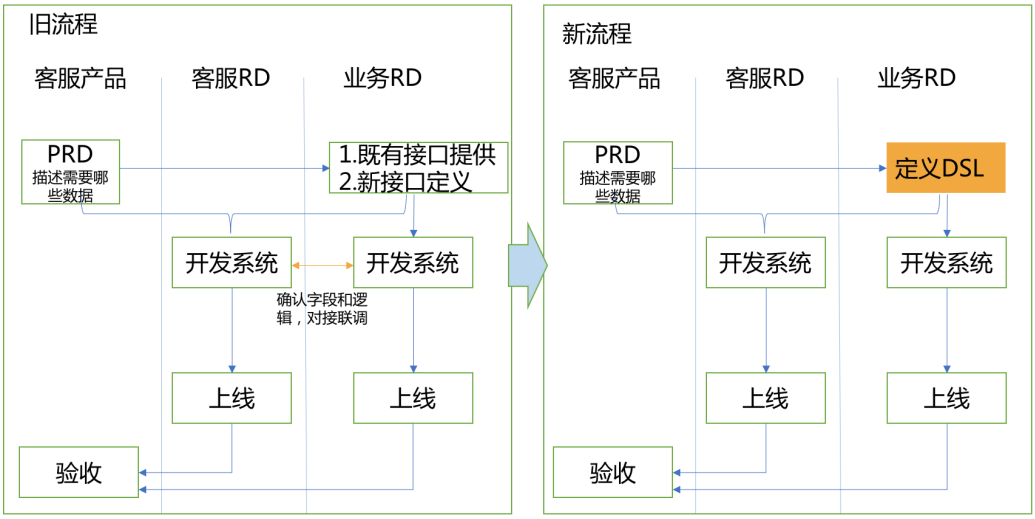
```
    String DSL = "上述DSL文件";
    String params  = "{\"orderId\":\"000000001\"}";
    Response resp = PirateEngine.invoke(DSL, params);
```

最后得到的数据即为调用场景真正需要的数据：

```
{
   "orderStatus":1,
   "payStatus":2
}
```

开发流程变化

因为获取数据的架构产生了变化，开发流程也随之发生改变。



客服RD和业务RD在开发过程中沟通、联调，信息确认时间占比较多。

业务方根据PRD信息提供可靠的DSL，然后提供相关所有服务，客服侧RD不用再熟悉业务方数据模型和确认字段含义，如何数据转换。

如图所示，因为减少了客服侧RD不断去向业务方RD确认返回的数据含义和逻辑，双方RD各自专注各自熟悉的领域，开发效率和最终结果准确性都有显著提升。

总结和展望

最后总结一下使用海盗之后的优势

- 去中心化的设计，可用性得到保证。
- 服务复用性高，领域划分更加清晰，让RD专注在自己熟悉的领域，降低研发成本。
- 因为流程变化后，业务方可以提前验证提供的数据，高质量交付。
- 客服侧对数据获取进行统一收口，可以对所有调用服务统一监控并对数据统一处理。

展望

海盗的技术规划：

- **丰富内部函数和运算表达式**：目前海盗提供了一部分简单的内部函数用来对取到的值进行简单处理，同时正在实现支持调用方自定义运算表达式来支持复杂场景的数据处理，这部分需要持续完善。
- **屏蔽远程调用协议异构性**：目前海盗只支持对美团Pigeon和MTThrift服务进行编排，这里要对协议进行扩展，支持类似HTTP等通用协议，同时支持调用方自定义协议和调用实现。
- **运营工具完善**：提供一个比较完整的运营工具，调用方可以自行配置DSL并进行校验，然后一键调用查询最终结果。同时调用方可以通过该工具进行日志、报表等相关数据查询。
- **自动生成单元测试**：能够把经过验证的DSL生成相应的单元测试用例给到数据提供方，持续保障提供的DSL的可用性和正确性。

作者简介

- 王彬，美团资深研发工程师，毕业于南京大学，2017年2月加入美团。目前主要专注于智能客服领域，从事后端工作。
- 陈胜，海盗项目负责人，智能客服技术负责人，2013年加入大众点评。在未来智能客服组会持续在平台化和垂直领域方向深入下去，为消费者、商家、企业提供更加智能的客户服务体验。

招聘广告

服务体验平台可以深入接触到公司的所有业务，推进业务改善产品。提升客户的服务体验。打造一个客户贴身的智能服务助手。通过技术的手段更快地解决客户的问题，并且最大程度地节省客服的人力成本。欢迎有意向的同学加入服务体验平台，上海、北京都有需求。简历请投递至：sheng.chen#dianping.com

美团点评智能支付核心交易系统的可用性实践

作者: 静儿

背景

每个系统都有它最核心的指标。比如在收单领域：进件系统第一重要的是保证入件准确，第二重要的是保证上单效率。清结算系统第一重要的是保证准确打款，第二重要的是保证及时打款。我们负责的系统是美团点评智能支付的核心链路，承担着智能支付100%的流量，内部习惯称为核心交易。因为涉及美团点评所有线下交易商家、用户之间的资金流转，对于核心交易来说：第一重要的是稳定性，第二重要的还是稳定性。

稳定重要性

问题引发

作为一个平台部门，我们的理想是第一阶段快速支持业务；第二阶段把控好一个方向；第三阶段观察好市场的方向，自己去引领一个大方向。

理想很丰满，现实是自从2017年初的每日几十万订单，到年底时，单日订单已经突破700万，系统面临着巨大的挑战。支付通道在增多；链路在加长；系统复杂性也相应增加。从最初的POS机到后来的二维码产品，小白盒、小黑盒、秒付……产品的多元化，系统的定位也在时刻的发生着变化。而系统对于变化的应对速度像是一个在和兔子赛跑的乌龟。

由于业务的快速增长，就算系统没有任何发版升级，也会突然出现一些事故。事故出现的频率越来越高，系统自身的升级，也经常是困难重重。基础设施升级、上下游升级，经常会发生“蝴蝶效应”，毫无征兆的受到影响。

问题分析

核心交易的稳定性问题根本上是怎么实现高可用的问题。

可用性指标

业界高可用的标准是按照系统宕机时间来衡量的：

	通俗叫法	可用性级别	年度宕机时间
基本可用	2个9	99%	87.6小时
较高可用	3个9	99.9%	8.8小时
具有故障自动恢复能力的可用性	4个9	99.99%	53分钟
极高可用性	5个9	99.999%	5分钟
容错可用性	6个9	99.9999%	31秒

可用性标准

因为业界的标准是后验的指标，考虑到对于平时工作的指导意义，我们通常采用服务治理平台OCTO来统计可用性。计算方法是：

➡服务可用性 = 成功调用数 / 调用总数

➡成功调用数 = 调用总数 - 失败调用数

➡平均可用性 = 选择时间段内成功调用总次数 / 选择时间段内调用总数

➡日平均调用数 = 选择时间段内调用总数 / 时间段的天数

✳说明：

调用总数是从服务端统计得到的值，失败数是从客户端统计得到的值。

根据服务中接口的类型，失败调用数可分为thrift和http两种类型。

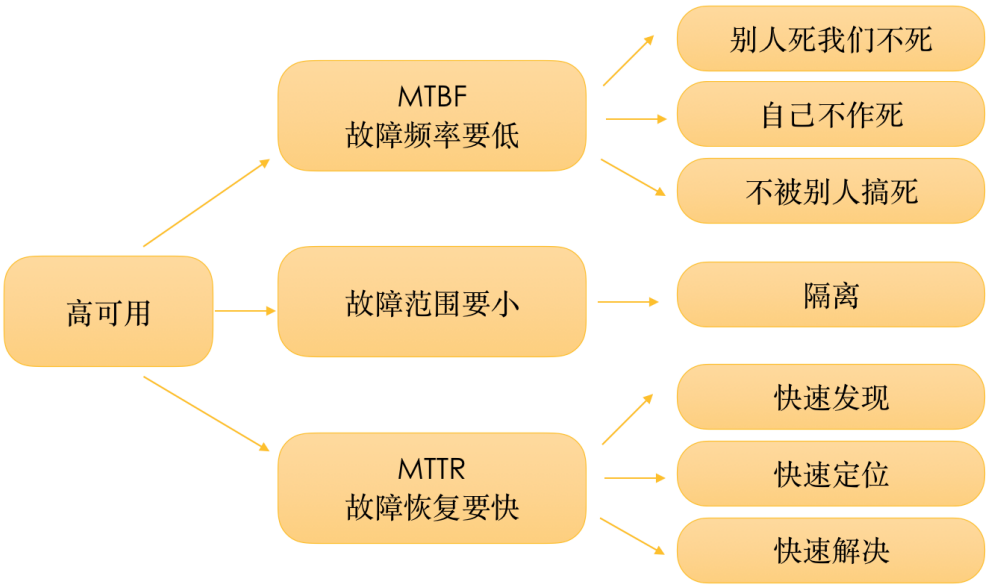
美团可用性计算

可用性分解

业界系统可靠性还有两个比较常用的关键指标：

- 平均无故障时间(Mean Time Between Failures，简称MTBF)：即系统平均能够正常运行多长时间，才发生一次故障。
- 平均修复时间(Mean Time To Repair，简称MTTR)：即系统由故障状态转为工作状态时修理时间的平均值。

对于核心交易来说，可用性最好是无故障。在有故障的时候，判定影响的因素除了时间外，还有范围。将核心交易的可用性问题分解则为：



可用性分解

问题解决

1. 发生频率要低之别人死我们不死

1.1 消除依赖、弱化依赖和控制依赖

用STAR法则举一个场景：

情境(situation)

我们要设计一个系统A，完成：使用我们美团点评的POS机，通过系统A连接银行进行付款，我们会有一些满减，使用积分等优惠活动。

任务(task)

分析一下对于系统A的显性需求和隐性需求：

1. 需要接收上游传过来的参数，参数里包含商家信息、用户信息、设备信息、优惠信息。
2. 生成单号，将交易的订单信息落库。
3. 敏感信息要加密。
4. 要调用下游银行的接口。
5. 要支持退款。
6. 要把订单信息同步给积分核销等部门。
7. 要能给商家一个查看订单的界面。
8. 要能给商家进行收款的结算。

基于以上需求，分析一下怎样才能让里面的最核心链路“使用POS机付款”稳定。

行动(action)

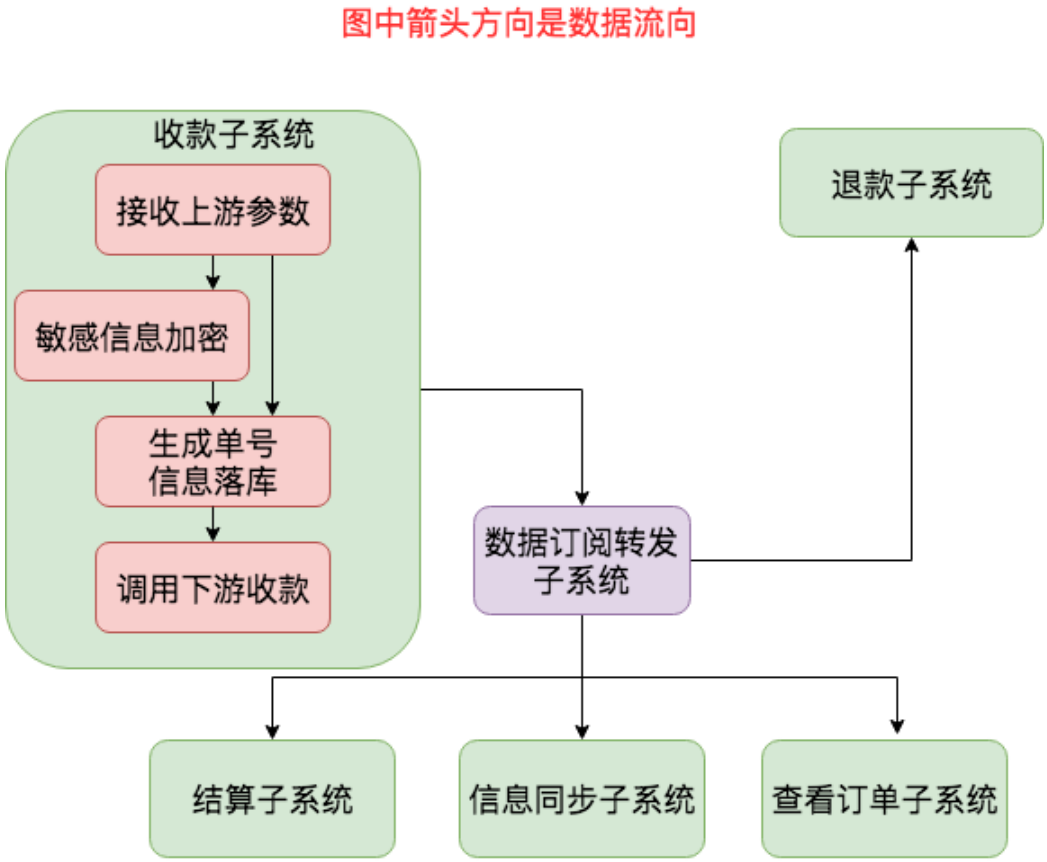
分析一下：需求1到4是付款必需链路，可以做在一个子系统里，姑且称之为收款子系统。5到8各自独立，每个都可以作为一个子系统来做，具体情况和开发人员数量、维护成本等有关系。

值得注意的是需求5–8和收款子系统的依赖关系并没有功能上的依赖，只有数据上的依赖。即他们都要依赖生成的订单数据。

收款子系统是整个系统的核心，对稳定性要求非常高。其他子系统出了问题，收款子系统不能受到影响。

基于上面分析，我们需要做一个收款子系统和其他子系统之间的一个解耦，统一管理给其他系统的数据。这里称为“订阅转发子系统”，只要保证这个系统不影响收款子系统的稳定即可。

粗略架构图如下：



架构图

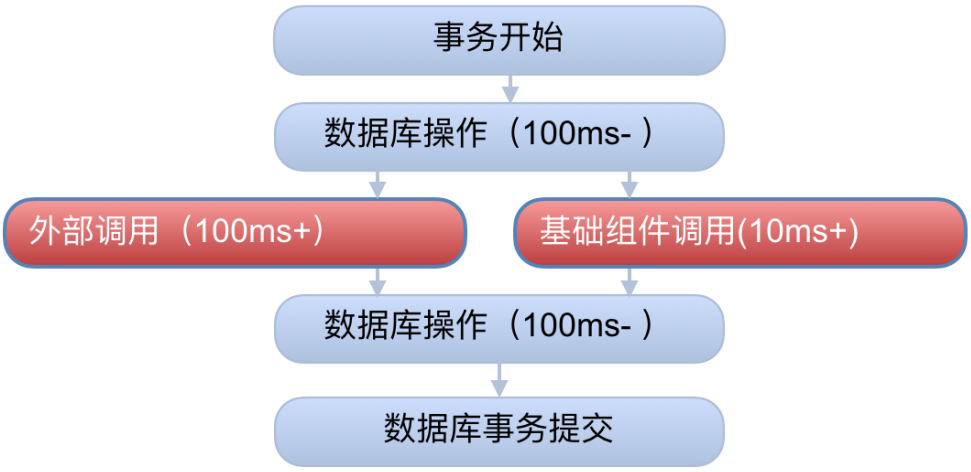
结果(result)

从上图可以看到，收款子系统和退款子系统、结算子系统、信息同步子系统、查看订单子系统之间没有直接依赖关系。这个架构达到了消除依赖的效果。收款子系统不需要依赖数据订阅转发子系统，数据订阅转发子系统需要依赖收款子系统的数据。我们控制依赖，数据订阅转发子系统从收款子系统拉取数据，而不需要收款子系统给数据订阅转发子系统推送数据。这样，数据订阅转发子系统挂了，收款子系统不受影响。

再说数据订阅转发子系统拉取数据的方式。比如数据存在MySQL数据库中，通过同步Binlog来拉取数据。如果采用消息队列来进行数据传输，对消息队列的中间件就有依赖关系了。如果我们设计一个灾备方案：消息队列挂了，直接RPC调用传输数据。对于这个消息队列，就达到了弱化依赖的效果。

1.2 事务中不包含外部调用

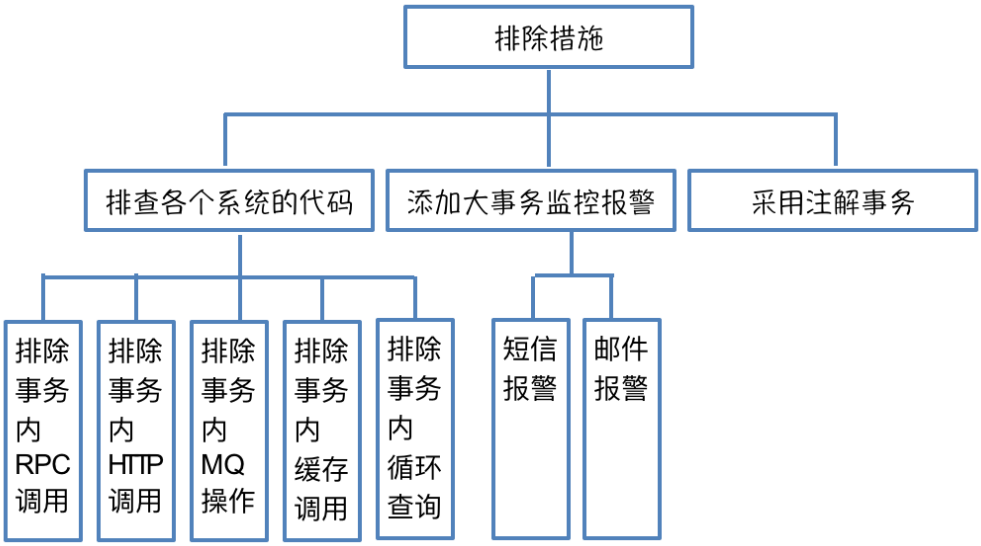
外部调用包括对外部系统的调用和基础组件的调用。外部调用具有返回时间不确定性的特征，如果包含在了事务里必然会造成大事务。数据库大事务会造成其它对数据库连接的请求获取不到，从而导致和这个数据库相关的所有服务处于等待状态，造成连接池被打满，多个服务直接宕掉。如果这个没做好，危险指数五颗星。下面的图显示出外部调用时间的不可控：



大事务问题

解决方法：

- 排查各个系统的代码，检查在事务中是否存在RPC调用、HTTP调用、消息队列操作、缓存、循环查询等耗时的操作，这个操作应该移到事务之外，理想的情况是事务内只处理数据库操作。
- 对大事务添加监控报警。大事务发生时，会收到邮件和短信提醒。针对数据库事务，一般分为1s以上、500ms以上、100ms以上三种级别的事务报警。
- 建议不要用XML配置事务，而采用注解的方式。原因是XML配置事务，第一可读性不强，第二切面通常配置的比较泛滥，容易造成事务过大，第三对于嵌套情况的规则不好处理。

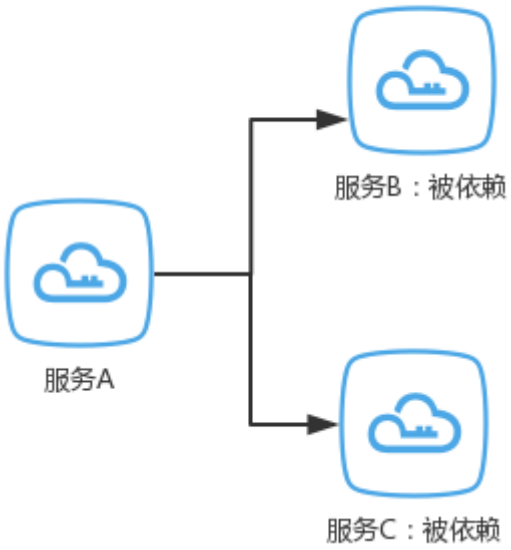


大事务排除措施

1.3 设置合理的超时和重试

对外部系统和缓存、消息队列等基础组件的依赖。假设这些被依赖方突然发生了问题，我们系统的响应时间是：内部耗时+依赖方超时时间*重试次数。如果超时时间设置过长、重试过多，系统长时间不返回，可能会导致连接池被打满，系统死掉；如果超时时间设置过短，499错误会增多，系统的可用性会降低。

举个例子：



依赖例子

服务A依赖于两个服务的数据完成此次操作。平时没有问题，假如服务B在你不知道的情况下，响应时间变长，甚至停止服务，而你的客户端超时时间设置过长，则你完成此次请求的响应时间就会变长，此时如果发生意外，后果会很严重。

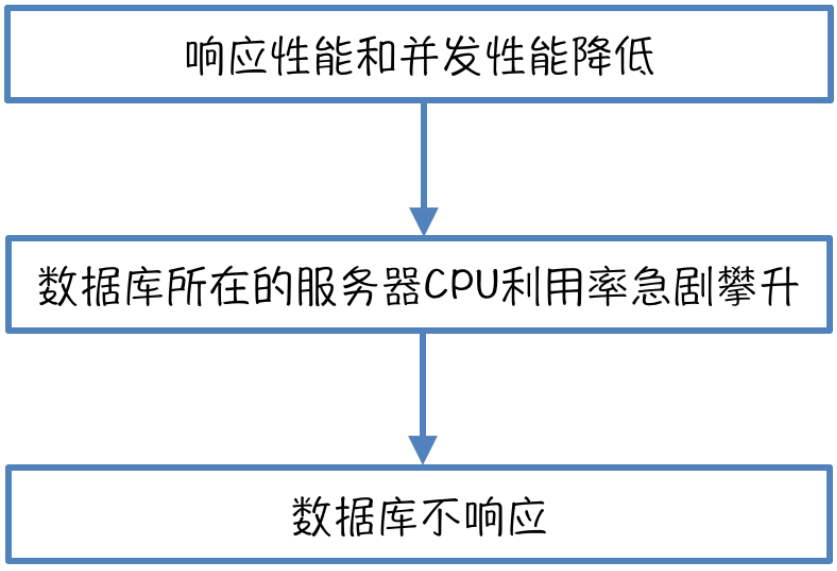
Java的Servlet容器，无论是Tomcat还是Jetty都是多线程模型，都用Worker线程来处理请求。这个可配置有上限，当你的请求打满Worker线程的最大值之后，剩余请求会被放到等待队列。等待队列也有上限，一旦等待队列都满了，那这台Web Server就会拒绝服务，对应到Nginx上返回就是502。如果你的服务是QPS较高的服务，那基本上这种场景下，你的服务也会跟着被拖垮。如果你的上游也没有合理的设置超时时间，那故障会继续向上扩散。这种故障逐级放大的过程，就是服务雪崩效应。

解决方法：

- 首先要调研被依赖服务自己调用下游的超时时间是多少。调用方的超时时间要大于被依赖方调用下游的时间。
- 统计这个接口99%的响应时间是多少，设置的超时时间在这个基础上加50%。如果接口依赖第三方，而第三方的波动比较大，也可以按照95%的响应时间。
- 重试次数如果系统服务重要性高，则按照默认，一般是重试三次。否则，可以不重试。

1.4 解决慢查询

慢查询会降低应用的响应性能和并发性能。在业务量增加的情况下造成数据库所在的服务器CPU利用率急剧攀升，严重的会导致数据库不响应，只能重启解决。关于慢查询，可以参考我们技术博客之前的文章《MySQL索引原理及慢查询优化》。



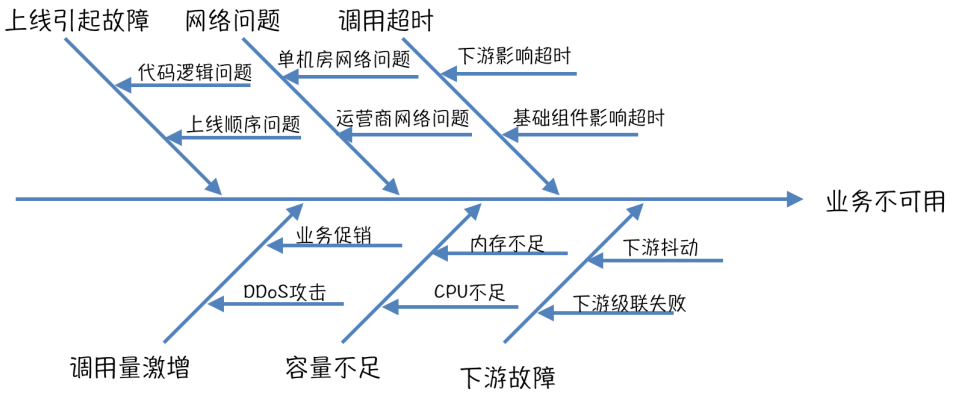
慢查询

解决方法：

- 将查询分成实时查询、近实时查询和离线查询。实时查询可穿透数据库，其它的不走数据库，可以用Elasticsearch来实现一个查询中心，处理近实时查询和离线查询。
- 读写分离。写走主库，读走从库。
- 索引优化。索引过多会影响数据库写性能。索引不够查询会慢。DBA建议一个数据表的索引数不超过4个。
- 不允许出现大表。MySQL数据库的一张数据表当数据量达到千万级，效率开始急剧下降。

1.5 熔断

在依赖的服务不可用时，服务调用方应该通过一些技术手段，向上提供有损服务，保证业务柔性可用。而系统没有熔断，如果由于代码逻辑问题上线引起故障、网络问题、调用超时、业务促销调用量激增、服务容量不足等原因，服务调用链路上有一个下游服务出现故障，就可能导致接入层其它的业务不可用。下图是对无熔断影响的鱼骨图分析：



无熔断

解决方法：

- 自动熔断：可以使用Netflix的Hystrix或者美团点评自己研发的Rhino来做快速失败。
- 手动熔断：确认下游支付通道抖动或不可用，可以手动关闭通道。

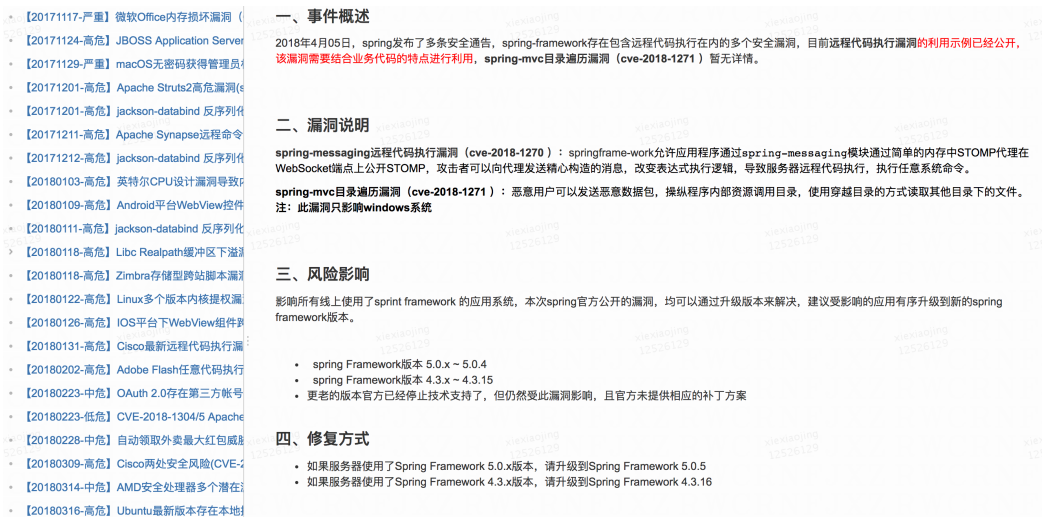
2. 发生频率要低之自己不作死

自己不作死要做到两点：第一自己不作，第二自己不死。

2.1 不作

关于不作，我总结了以下7点：

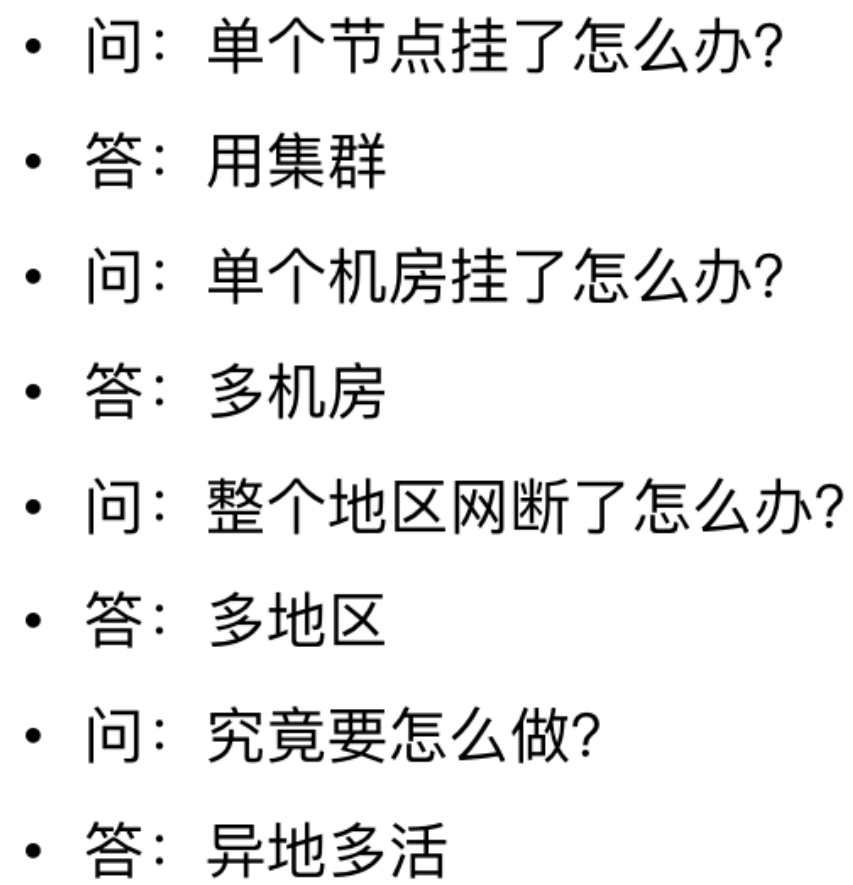
1. 不当小白鼠：只用成熟的技术，不因技术本身的问题影响系统的稳定。
2. 职责单一化：不因职责耦合而削弱或抑制它完成最重要职责的能力。
3. 流程规范化：降低人为因素带来的影响。
4. 过程自动化：让系统更高效、更安全的运营。
5. 容量有冗余：为了应对竞对系统不可用用户转而访问我们的系统、大促来临等情况，和出于容灾考虑，至少要保证系统2倍以上的冗余。
6. 持续的重构：持续重构是确保代码长期没人动，一动就出问题的有效手段。
7. 漏洞及时补：美团点评有安全漏洞运维机制，提醒督促各个部门修复安全漏洞。



安全漏洞

2.2 不死

关于不死，地球上有五大不死神兽：能在恶劣环境下停止新陈代谢的“水熊虫”；可以返老还童的“灯塔水母”；在硬壳里休养生息的“蛤蜊”；水、陆、寄生样样都成的“涡虫”；有隐生能力的“轮虫”。它们的共通特征用在系统设计领域上就是自身容错能力强。这里“容错”的概念是：使系统具有容忍故障的能力，即在产生故障的情况下，仍有能力将指定的过程继续完成。容错即是Fault Tolerance，确切地说是容故障(Fault)，而并非容错误(Error)。

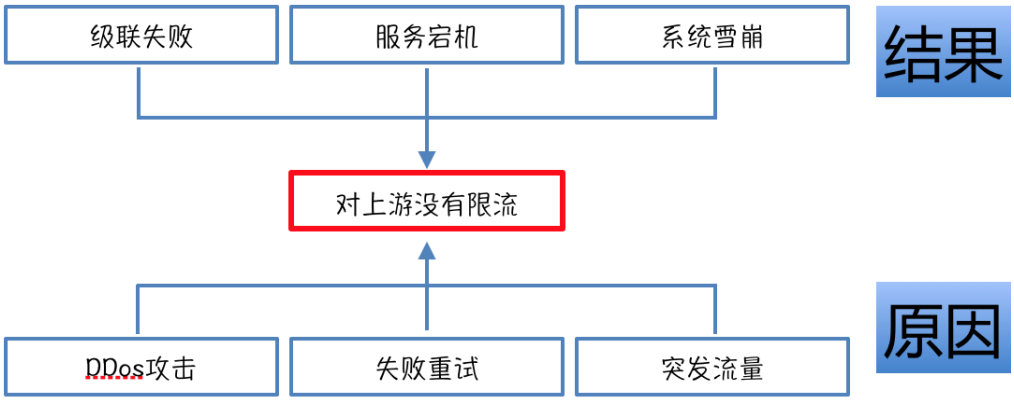


容错

3. 发生频率要低之不被别人搞死

3.1 限流

在开放式的网络环境下，对外系统往往会收到很多有意无意的恶意攻击，如DDoS攻击、用户失败重刷。虽然我们的队友各个是精英，但还是要做好保障，不被上游的疏忽影响，毕竟，谁也无法保证其他同学哪天会写一个如果下游返回不符合预期就无限次重试的代码。这些内部和外部的巨量调用，如果不加以保护，往往会扩散到后台服务，最终可能引起后台基础服务宕机。下图是对无限流影响的问题树分析：



无限流

解决方法：

- 通过对服务端的业务性能压测，可以分析出一个相对合理的最大QPS。
- 流量控制中用的比较多的三个算法是令牌桶、漏桶、计数器。可以使用Guava的RateLimiter来实现。其中SmoothBurstry是基于令牌桶算法的，SmoothWarmingUp是基于漏桶算法的。
- 核心交易这边采用美团服务治理平台OCTO做thrift截流。可支持接口粒度配额、支持单机/集群配额、支持指定消费者配额、支持测试模式工作、及时的报警通知。其中测试模式是只报警并不真正节流。关闭测试模式则超过限流阈值系统做异常抛出处理。限流策略可以随时关闭。
- 可以使用Netflix的Hystrix或者美团点评自己研发的Rhino来做特殊的针对性限流。

4. 故障范围要小之隔离

隔离是指将系统或资源分割开，在系统发生故障时能限定传播范围和影响范围。

服务器物理隔离原则

- ① 内外有别：内部系统与对外开放平台区分对待。
- ② 内部隔离：从上游到下游按通道从物理服务器上进行隔离，低流量服务合并。
- ③ 外部隔离：按渠道隔离，渠道之间互不影响。

线程池资源隔离

- Hystrix通过命令模式，将每个类型的业务请求封装成对应的命令请求。每个命令请求对应一个线程池，创建好的线程池是被放入到ConcurrentHashMap中。

注意：尽管线程池提供了线程隔离，客户端底层代码也必须要有超时设置，不能无限制的阻塞以致于线程池一直饱和。

信号量资源隔离

- 开发者可以使用Hystrix限制系统对某一个依赖的最高并发数，这个基本上就是一个限流策略。每次调用依赖时都会检查一下是否到达信号量的限制值，如达到，则拒绝。

5. 故障恢复要快之快速发现

发现分为事前发现、事中发现和事后发现。事前发现的主要手段是压测和故障演练；事中发现的主要手段是监控报警；事后发现的主要手段是数据分析。

5.1 全链路线上压测

你的系统是否适合全链路线上压测呢?一般来说，全链路压测适用于以下场景：

- ① 针对链路长、环节多、服务依赖错综复杂的系统，全链路线上压测可以更快更准确的定位问题。
- ② 有完备的监控报警，出现问题可以随时终止操作。
- ③ 有明显的业务峰值和低谷。低谷期就算出现问题对用户影响也比较小。

全链路线上压测的目的主要有：

- ① 了解整个系统的处理能力
- ② 排查性能瓶颈
- ③ 验证限流、降级、熔断、报警等机制是否符合预期并分析数据反过来调整这些阈值等信息
- ④ 发布的版本在业务高峰的时候是否符合预期
- ⑤ 验证系统的依赖是否符合预期

全链路压测的简单实现：

- ① 采集线上日志数据来做流量回放，为了和实际数据进行流量隔离，需要对部分字段进行偏移处理。
- ② 数据着色处理。可以用中间件来获取和传递流量标签。
- ③ 可以用影子数据表来隔离流量，但是需要注意磁盘空间，建议如果磁盘剩余空间不足70%采用其他的方式隔离流量。
- ④ 外部调用可能需要Mock。实现上可以采用一个Mock服务随机产生和线上外部调用返回时间分布的时延。

压测工具上，核心交易这边使用美团点评开发的pTest。

	功能点	pTest	JMeter	PTS（阿里云）
易用性	平台化	😊	☹	😊
	HTTP快速支持	😊	😊	😊
	Thrift快速支持	😊	☹	☹
	聚合测试结果	😊	☹	😊
	易扩展	☹	😊	😊
稳定性	万级大并发	😊	☹	😊
	大流量回放	😊	☹	☹

压测工具对比

6. 故障恢复要快之快速定位

定位需要靠谱的数据。所谓靠谱就是和要发现的问题紧密相关的，无关的数据会造成视觉盲点，影响定位。所以对于日志，要制定一个简明日志规范。另外系统监控、业务监控、组件监控、实时分析诊断工具也是定位的有效抓手。

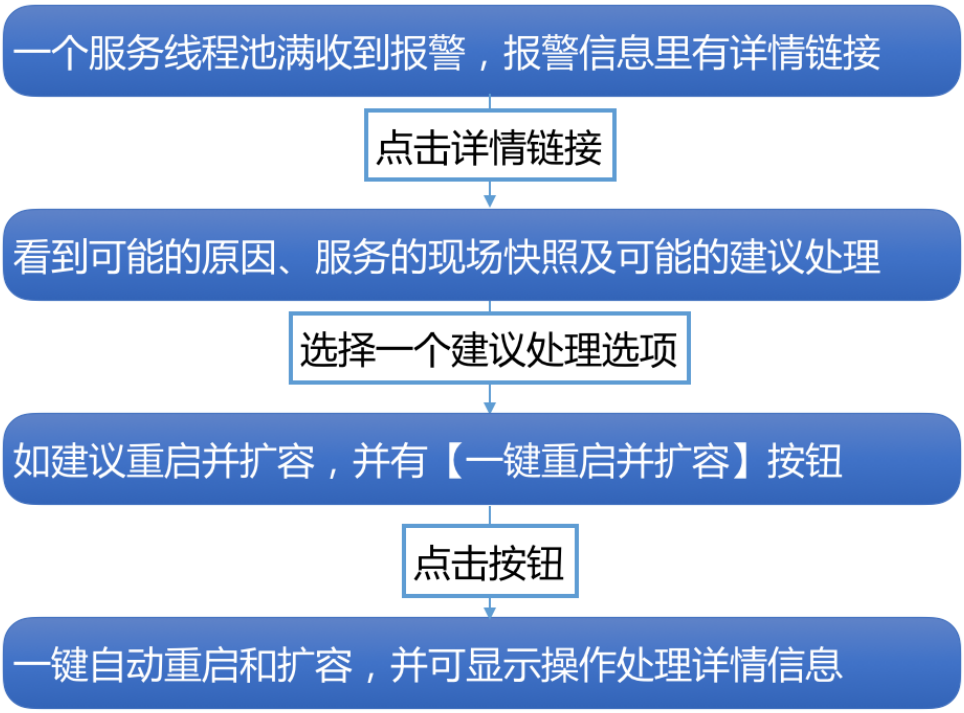
简明日志规范

7. 故障恢复要快之快速解决

要解决，提前是发现和定位。解决的速度还取决于是自动化的、半自动化的还是手工的。核心交易有意向搭建一个高可用系统。我们的口号是：“不重复造轮子，用好轮子。”这是一个集成平台，职责是：“聚焦

核心交易高可用，更好、更快、更高效。”

美团点评内部可以使用的用于发现、定位、处理的系统和平台非常多，但是如果一个个打开链接或者登陆系统，势必影响解决速度。所以我们要做集成，让问题一站式解决。希望达到的效果举例如下：



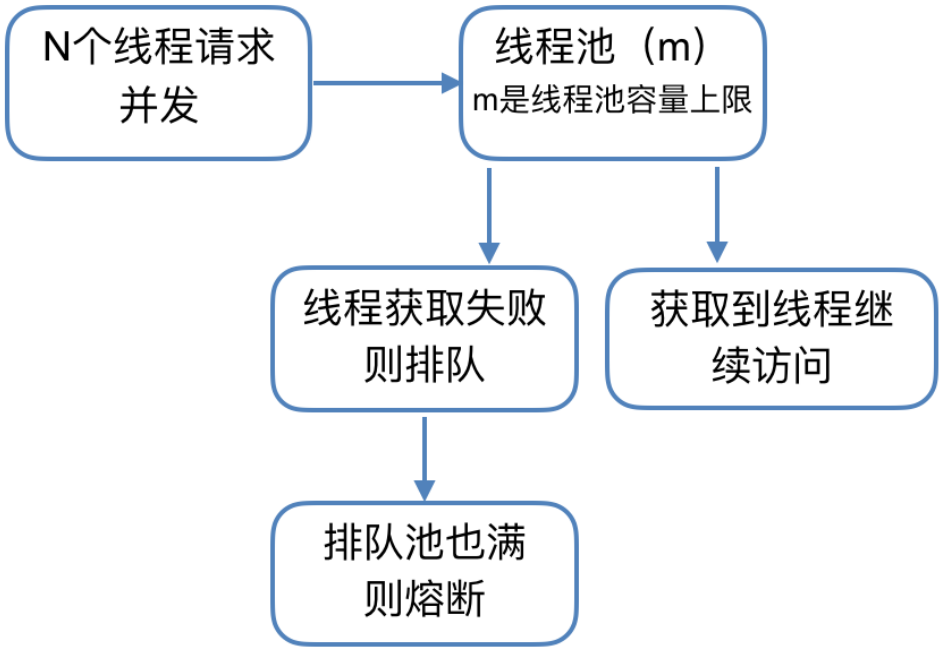
解决问题

工具介绍

Hystrix

Hystrix实现了断路器模式来对故障进行监控，当断路器发现调用接口发生了长时间等待，就使用快速失败策略，向上返回一个错误响应，这样达到防止阻塞的目的。这里重点介绍一下Hystrix的线程池资源隔离和信号量资源隔离。

线程池资源隔离



线程池资源隔离

优点

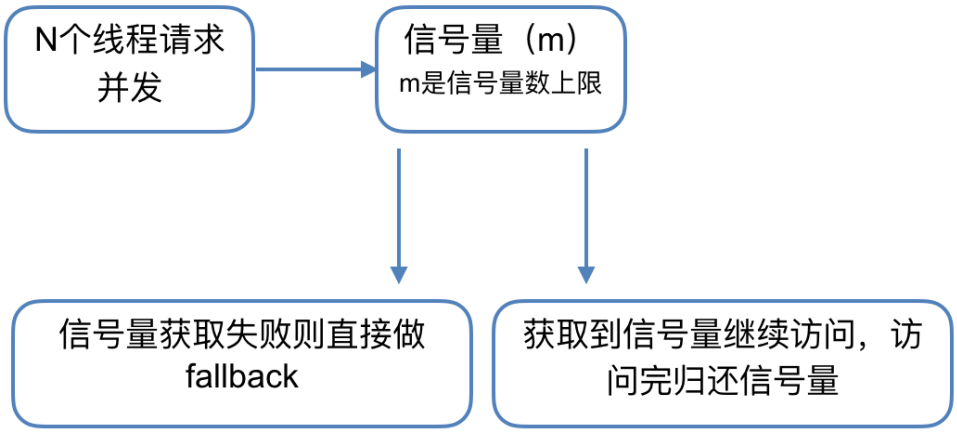
- 使用线程可以完全隔离第三方代码，请求线程可以快速放回。
- 当一个失败的依赖再次变成可用时，线程池将清理，并立即恢复可用，而不是一个长时间的恢复。
- 可以完全模拟异步调用，方便异步编程。

缺点

- 线程池的主要缺点是它增加了CPU，因为每个命令的执行涉及到排队（默认使用SynchronousQueue避免排队），调度和上下文切换。
- 对使用ThreadLocal等依赖线程状态的代码增加复杂性，需要手动传递和清理线程状态（Netflix公司内部认为线程隔离开销足够小，不会造成重大的成本或性能的影响）。

信号量资源隔离

开发者可以使用Hystrix限制系统对某一个依赖的最高并发数。这个基本上就是一个限流策略，每次调用依赖时都会检查一下是否到达信号量的限制值，如达到，则拒绝。



信号量资源隔离

优点

- 不新起线程执行命令，减少上下文切换。

缺点

- 无法配置断路，每次都一定会去尝试获取信号量。

比较一下线程池资源隔离和信号量资源隔离

- 线程隔离是和主线程无关的其他线程来运行的；而信号量隔离是和主线程在同一个线程上做的操作。
- 信号量隔离也可以用于限制并发访问，防止阻塞扩散，与线程隔离的最大不同在于执行依赖代码的线程依然是请求线程。
- 线程池隔离适用于第三方应用或者接口、并发量大的隔离；信号量隔离适用于内部应用或者中间件；并发需求不是很大的场景。

	是否有线程切换	是否支持异步	是否支持超时	是否支持熔断	开销大小	是否支持限流
信号量	否	否	否	是	小	是
线程池	是	是	是	是	大	是

隔离对比

Rhino

Rhino是美团点评基础架构团队研发并维护的一个稳定性保障组件，提供故障模拟、降级演练、服务熔断、服务限流等功能。和Hystrix对比：

- 内部通过CAT（美团点评开源的监控系统，参见之前的博客“深度剖析开源分布式监控CAT🔗”）进行了一系列埋点，方便进行服务异常报警。
- 接入配置中心，能提供动态参数修改，比如强制熔断、修改失败率等。

总结思考

王国维 在《人间词话》里谈到了治学经验，他说：古今之成大事业、大学问者，必经过三种之境界：

> **第一种境界 昨夜西风凋碧树。独上高楼，望尽天涯路。**
>
> **第二种境界 衣带渐宽终不悔，为伊消得人憔悴。**
>
> **第三种境界 众里寻他千百度，蓦然回首，那人却在，灯火阑珊处。**

核心交易的高可用目前正在经历第一种：高瞻远瞩认清前人所走的路，以总结和学习前人的经验做为起点。

下一阶段，既然认定了目标，我们会呕心沥血孜孜以求，持续发展高可用。最终，当我们做了很多的事情，回过头来看，相信会对高可用有更清晰和深入的认识。

关于作者

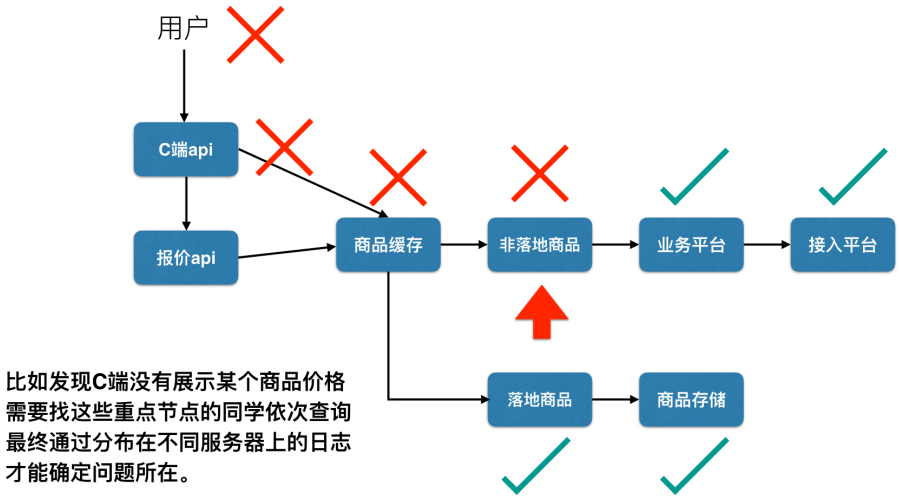
- 晓静，20岁时毕业于东北大学计算机系。在毕业后的第一家公司由于出众的语言天赋，在1年的时间里从零开始学日语并以超高分通过了国际日语一级考试，担当两年日语翻译的工作。后就职于人人网，转型做互联网开发。中国科学院心理学研究生。有近百个技术发明专利，创业公司合伙人。有日本东京，美国硅谷技术支持经验。目前任美团点评技术专家，负责核心交易。

卫星系统——酒店后端全链路日志收集工具介绍

作者: 亚辉 曾鋆

背景

随着酒店业务的高速发展，我们为用户、商家提供的服务越来越精细，系统服务化程度、复杂度也逐渐上升。微服务化虽然能够很好地解决问题，但也有副作用，比如，问题定位。



每次问题定位都需要从源头开始找同事帮我人肉查日志，举一个简单的例子：

“这个详情页的价格是怎么算出来的?”

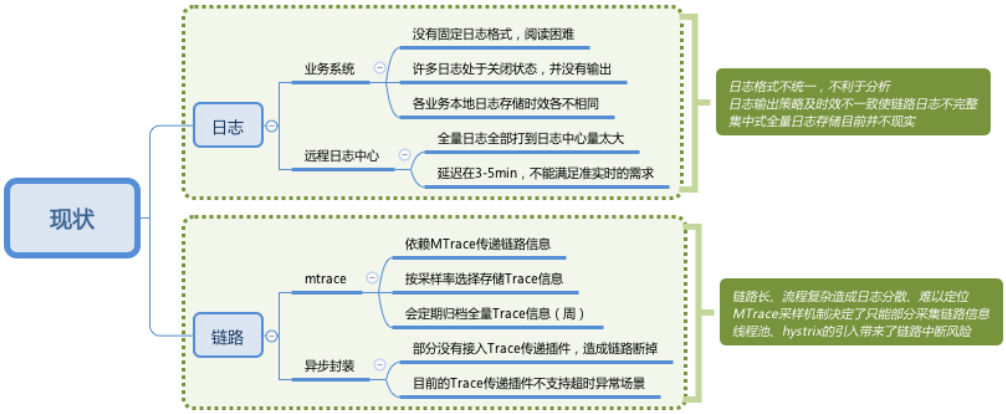
一次用户酒店可订空房页（POI详情页）访问，流量最多需要经过73个服务节点。查问题的时候需要先后找4~5个关键节点的同学帮我们登录十多个不同节点的机器，查询具体的日志，沟通成本极大，效率很低。

为了解决这个问题，基础架构的同学提供了MTrace（详情可以参考技术博客：《分布式会话跟踪系统架构设计与实践》）协助业务方排查长链路问题。

但是与此同时，还有许多不确定性因素，使问题排查过程更加艰难，甚至无果而终：

1. 各服务化节点存储日志的时间长度不一致；
2. 有的服务节点，因为QPS过高，只能不打或者随机打印日志，会导致最终查问题的时候，线索因为没日志断掉；
3. 有的服务节点，使用了异步逻辑（线程池、Hystrix、异步RPC等）导致日志中缺失Trace ID，无法关联在一起；
4. 各服务节点的采样率不一样，链路数据的上报率也是随机性的，线索容易断掉；
5. MTrace上只有链路信息，没有关联各服务节点的日志信息；
6. 动态扩容节点上的日志，缩容后无法找到。

总结起来如图所示：



目标

我们的核心诉求有两个：

1. 根据用户行为快速定位到具体的Trace ID，继而查询到这次服务调用链路上所有节点的日志；
2. 查询的实时性要能做到准实时（秒级输出），相关链路日志要在独立外部存储系统中保存半年以上。

然后我们对诉求做了进一步的拆分：

1. 全量打日志不现实，需要选择性打，打价值最高部分的日志；
2. 链路数据需要全服务节点都上传，避免因为异步化等原因造成链路数据中断不上传；
3. 接入方式尽量简单，避免所有服务节点都需要修改具体业务代码来接入。最好是能通过日志拦截的方式，其它保持透明；
4. 日志格式化，该有的字段（AppKey、hostname、IP、timestamp等）不需要业务RD反复输入，自动补齐；
5. 在不阻塞原有业务操作的情况下，做到准实时展示链路、日志；
6. 链路数据和日志数据存储，不依赖各服务节点，需要在独立的存储系统上存储半年以上。

系统

搞清了核心诉求后，我们针对性地做了许多调研，最终定了一个系统的整体设计方案，这就是目前已经上线并实践已久的美团点评酒店「**卫星系统**」。

下面，我们就来对系统做详细介绍，包括一些核心细节点。

架构

如下图所示，卫星系统从横向分为链路和日志两个部分。

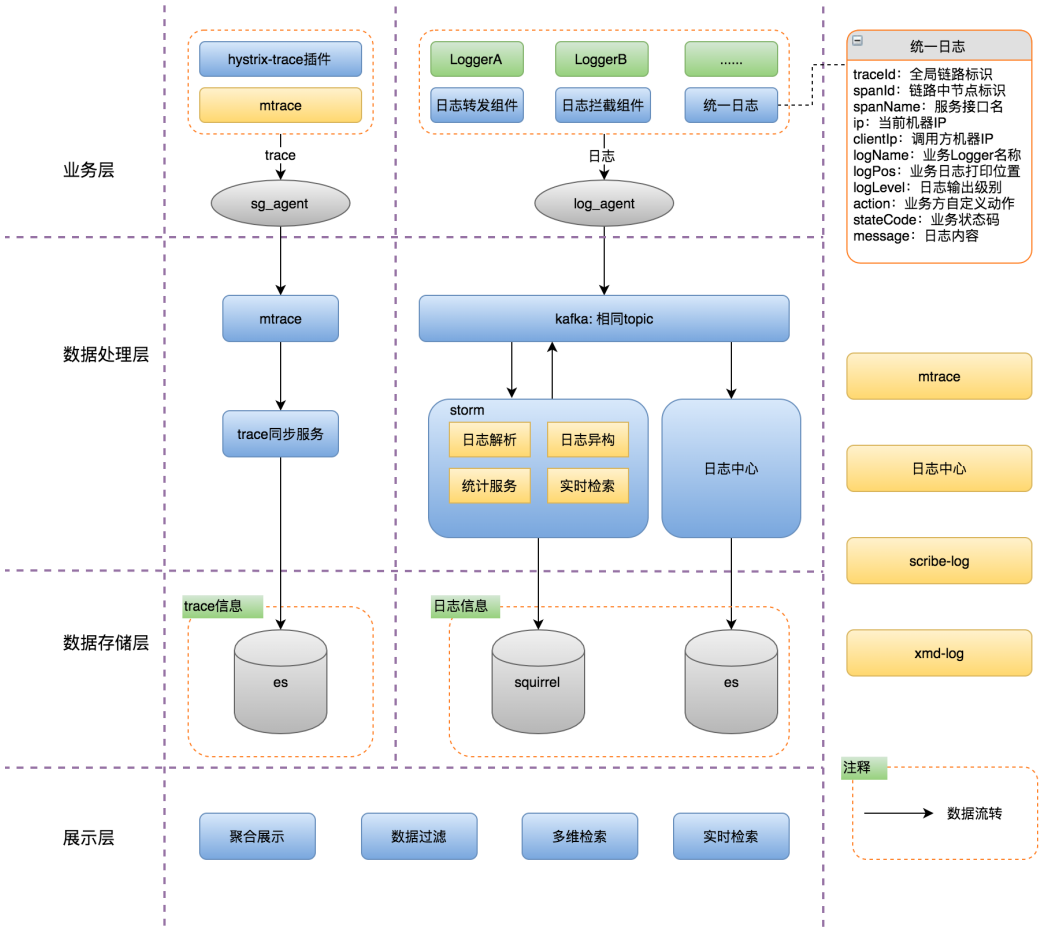


图2 全链路日志系统整体架构

链路部分是以MTrace为基础，用支持超时fallback下Trace信息传递的Hystrix-Trace插件来覆盖全部场景，保证链路被完整采集。

日志部分在接入端有三大核心步骤，首先是依托于日志拦截组件实现对业务代码零侵入的情况下收集系统中所有日志内容，然后根据统一日志规范对日志进行格式化处理，最后通过基于logcenter日志传输机制实现日志到Kafka的传输。

从纵向又分为：

1. 业务接入层，根据策略采集Trace与业务日志；
2. 数据处理层，通过storm流式处理日志信息；
3. 数据存储层，用于支持实时查询的Squirrel（美团点评Redis集群）与持久存储的ES（ElasticSearch），以及面向用户的展示层。

日志采样方案

接入端是所有数据之源，所以方案设计极为关键。要解决的问题有：采集策略、链路完整性保障、日志拦截、日志格式化、日志传输。

有的业务单台机器每天日志总量就有百G以上，更有不少业务因为QPS过高而选择平时不打印日志，只在排查问题时通过动态日志级别调整来临时输出。所以，我们在最初收集日志时必须做出取舍。经过分析，发现在排查问题的时候，绝大多数情况下发起人都是自己人（RD、PM、运营），如果我们只将这些人发

起的链路日志记下来，那么目标日志量将会极大减少，由日志量过大而造成的存储时间短、查询时效性差等问题自然得到解决。

所以我们制定了这样的采集策略：

通过在链路入口服务判断发起人是否满足特定人群（住宿事业部员工）来决定是否进行日志采集，将采集标志通过MTrace进行全链路传递。这样就能保证链路上所有节点都能行为一致地去选择是否进行日志上报，保证链路上日志的完整性。

日志拦截

作为核心要素的日志，如何进行收集是一个比较棘手的问题。让业务方强制使用我们的接口进行日志输出会带来许多麻烦，一方面会影响业务方原有的日志输出策略；另一方面，系统原有的日志输出点众多，涉及的业务也五花八门，改动一个点很简单，但是全面进行改动难保不会出现未知影响。所以，需要尽可能降低对接入方代码的侵入。

由于目前酒店核心业务已全面接入log4j2，通过研究，发现我们可以注册全局Filter来遍历系统所有日志，这一发现，使我们实现了代码零改动的情况下收集到系统所有日志。

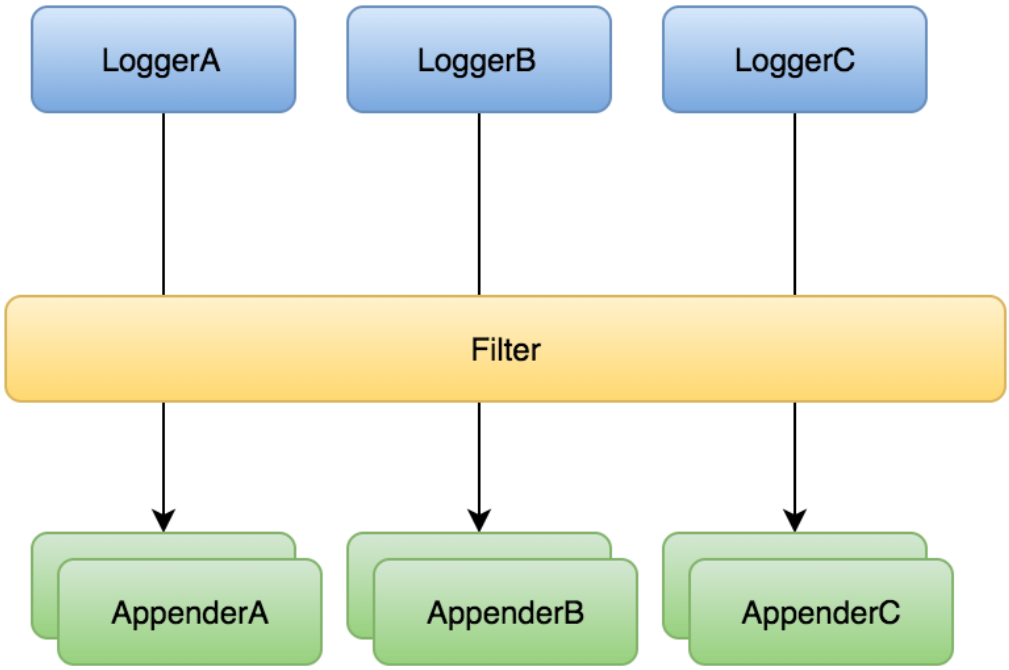


图3 基于log4j2 filter机制的日志收集策略

日志格式化

业务系统输出的日志格式不一，比如有的没有打印TraceID信息，有的没有打印日志位置信息从而很难进行定位。这主要带来两方面的问题，一方面不利于由人主导的排查分析工作，另一方面也不利于后续的系统优化升级，比如对日志的自动化分析报警等等。

针对这些问题，我们设计了统一日志规范，并由框架完成缺失内容的填充，同时给业务方提供了标准化的日志接口，业务方可以通过该接口定义日志的元数据，为后续支持自动化分析打下基础。

由框架填充统一日志信息这一过程利用到了log4j2的Plugins机制，通过Properties、Lookups、ContextMap实现业务无感知的操作。

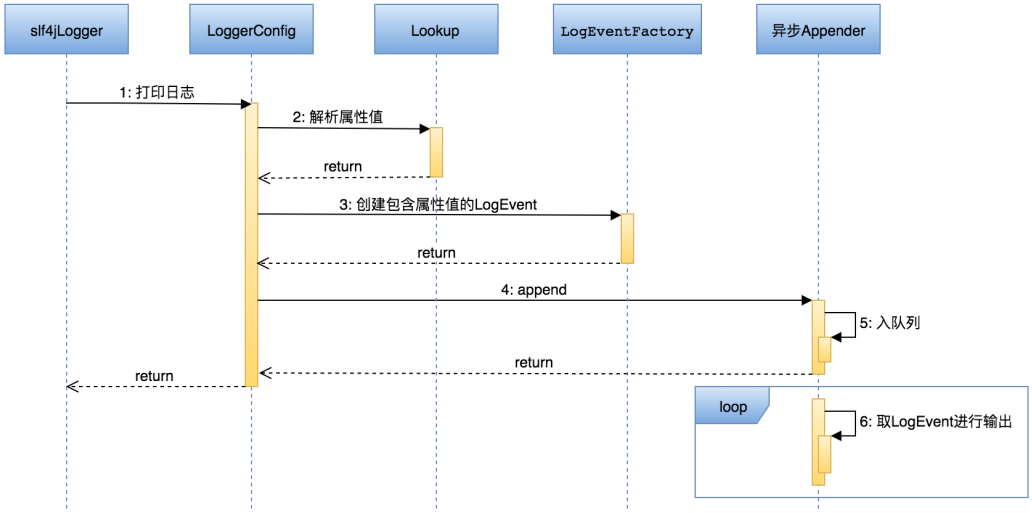


图4 通过Plugins机制支持格式化日志属性传递

日志处理

我们在最终的日志传输环节利用了日志中心的传输机制，使用日志中心的ScribeAppender实现日志传输至本地agent，然后上报到远端Kafka，这样设计有几点好处：

1. 依赖公司成熟的基础服务相对而言更可靠、更稳定，同时也省去了自己搭建服务、保证服务安全这一过程；
2. 可以将日志直接转存至日志中心ES做持久化存储，同时支持快速灵活的数据检索；
3. 可以通过Storm对日志进行流式处理，支持灵活的系统扩展，比如：实时检索、基于日志的实时业务检查、报警等等，为后续系统扩展升级打下基础。

我们的数据处理逻辑全部在Storm进行处理，主要包含日志存储Squirrel（美团点评内部基于Redis Cluster研发的纯内存存储）、实时检索与Trace同步。

目前日志中心ES能保证分钟级别实时性，但是对于RD排查问题还是不够，必须支持秒级别实时性。所以我们选择将特定目标用户的日志直接存入Squirrel，失效时间只有半小时，查询日志时结合ES与Squirrel，这样既满足了秒级别实时性，又保证了日志量不会太大，对Squirrel的压力可以忽略不计。

我们的系统核心数据有链路与日志，链路信息的获取是通过MTrace服务获得，但是MTrace服务对链路数据的保存时间有限，无法满足我们的需求。所以，我们通过延时队列从MTrace获取近期的链路信息进行落地存储，这样就实现了数据的闭环，保证了数据完整性。

链路完整性保障

MTrace组件的Trace传递功能基于ThreadLocal，而酒店业务大量使用异步化逻辑（线程池、Hystrix），这样会造成传递信息的损失，破坏链路完整性。

一方面，通过Sonar检查和梳理关键链路，来确保业务方使用类似 transmittable-thread-local 中的 `ExecutorServiceTtlWrapper.java` 、 `ExecutorTtlWrapper.java` 的封装，来将ThreadLocal里的Trace信息，也传递到异步线程中（前文提到的MTrace也提供这样的封装）。

另一方面，Hystrix的线程池模式会造成线程变量丢失。为了解决这个问题，MTrace提供了Mtrace Hystrix Support Plugin插件实现跨线程调用时的线程变量传递，但是由于Hystrix有专门的timer线程池来进行超时fallback调用，使得在超时情况下进入fallback逻辑以后的链路信息丢失。

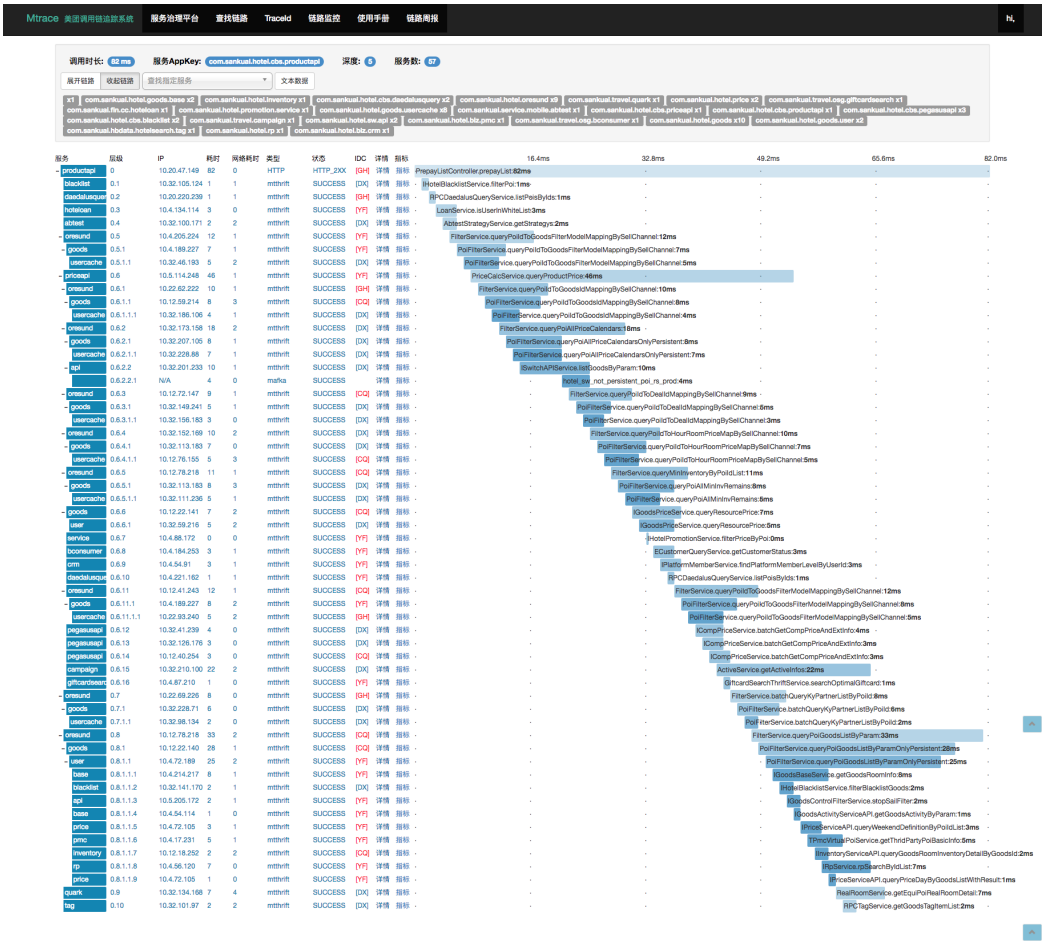
针对这个问题，我们深入研究了Hystrix机制，最终结合Hystrix Command Execution Hook、Hystrix ConcurrencyStrategy、Hystrix Request Context实现了覆盖全场景的Hystrix-Trace插件，保障了链路的完整性。

```
HystrixPlugins.getInstance().registerCommandExecutionHook(new HystrixCommandExecutionHook() {
    @Override
    public <T> void onStart(HystrixInvokable<T> commandInstance) {
        // 执行command之前将trace信息保存至hystrix上下文，实现超时子线程的trace传递
        if (!HystrixRequestContext.isCurrentThreadInitialized()) {
            HystrixRequestContext.initializeContext();
        }
        spanVariable.set(Tracer.getServerSpan());
    }

    @Override
    public <T> Exception onError(HystrixInvokable<T> commandInstance, HystrixRuntimeException.FailureType failureType, Except
ion e) {
        // 执行结束后清空hystrix上下文信息
        HystrixRequestContext context = HystrixRequestContext.getContextForCurrentThread();
        if (context != null) {
            context.shutdown();
        }
        return e;
    }

    @Override
    public <T> void onSuccess(HystrixInvokable<T> commandInstance) {
        // 执行结束后清空hystrix上下文信息
        HystrixRequestContext context = HystrixRequestContext.getContextForCurrentThread();
        if (context != null) {
            context.shutdown();
        }
    }
});

HystrixPlugins.getInstance().registerConcurrencyStrategy(new HystrixConcurrencyStrategy() {
    @Override
    public <T> Callable<T> wrapCallable(Callable<T> callable) {
        // 通过自定义callable保存trace信息
        return WithTraceCallable.get(callable);
    }
});
```

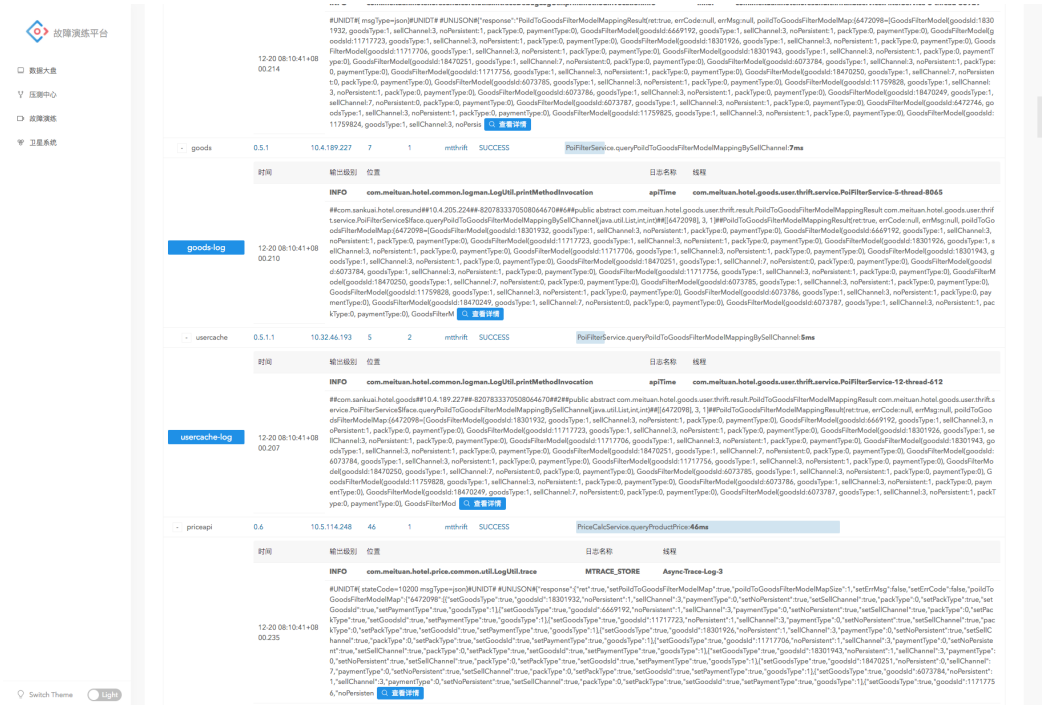
效果展示

比如以排查一次用户点击某POI详情页的TraceID为例子：

我们可以看到他在MTrace中的调用链路是这样的：



在卫星系统中，展示为如下效果：



可见，在保留了链路数据的基础上，系统还将全链路节点日志聚合到了一起，提升了排查效率。

后续规划

目前，系统还处于初级阶段，主要用来解决RD在排查问题时的两大痛点：日志信息的不完整与太分散，现在已经满足了这一需求。但是，全链路日志系统能做的不止这些，后续的主要规划有如下几方面：

1. 支持多链路日志关联搜索，比如一次列表页刷新与后续的详情页展示，虽然链路是多个但是实际处在一个关联的场景中。支持关联搜索就可以可以将日志排查目标从单个动作维度扩展到多动作组成的场景维度。
2. 支持业务方自定义策略规则进行基于日志的自动化业务正确性检查，如果检查出问题可以直接将详细信息通知相关人员，实现实时监测日志、实时发现问题、实时通知到位，免去人工费时费力的低效劳动。

作者简介

- 亚辉，2015年加入美团点评，就职于美团点评酒旅事业群技术研发部酒店后台研发组。
- 曾鋆，2013年加入美团点评，就职于美团点评酒旅事业群技术研发部酒店后台研发组。

招聘信息

最后发个广告，美团点评酒旅事业群技术研发部酒店后台研发组长期招聘Java后台、架构方面的人才，有兴趣的同学可以发送简历到xuguanfei#meituan.com。

深入浅出排序学习：写给程序员的算法系统开发实践

作者: 刘丁

引言

我们正处在一个知识爆炸的时代，伴随着信息量的剧增和人工智能的蓬勃发展，互联网公司越发具有强烈的个性化、智能化信息展示的需求。而信息展示个性化的典型应用主要包括搜索列表、推荐列表、广告展示等等。

很多人不知道的是，看似简单的个性化信息展示背后，涉及大量的数据、算法以及工程架构技术，这些足以让大部分互联网公司望而却步。究其根本原因，个性化信息展示背后的技术是排序学习问题（Learning to Rank）。市面上大部分关于排序学习的文章，要么偏算法、要么偏工程。虽然算法方面有一些系统性的介绍文章，但往往对读者的数学能力要求比较高，也比较偏学术，对于非算法同学来说门槛非常高。而大部分工程方面的文章又比较粗糙，基本上停留在Google的Two-Phase Scheme阶段，从工程实施的角度来说，远远还不够具体。

对于那些由系统开发工程师负责在线排序架构的团队来说，本文会采用通俗的例子和类比方式来阐述算法部分，希望能够帮助大家更好地理解和掌握排序学习的核心概念。如果是算法工程师团队的同学，可以忽略算法部分的内容。本文的架构部分阐述了美团点评到店餐饮业务线上运行的系统，可以作为在线排序系统架构设计的参考原型直接使用。该架构在服务治理、分层设计的理念，对于保障在线排序架构的高性能、高可用性、易维护性也具有一定的参考价值。包括很多具体环节的实施方案也可以直接进行借鉴，例如流量分桶、流量分级、特征模型、级联模型等等。

总之，让开发工程师能够理解排序学习算法方面的核心概念，并为在线架构实施提供细颗粒度的参考架构，是本文的重要目标。

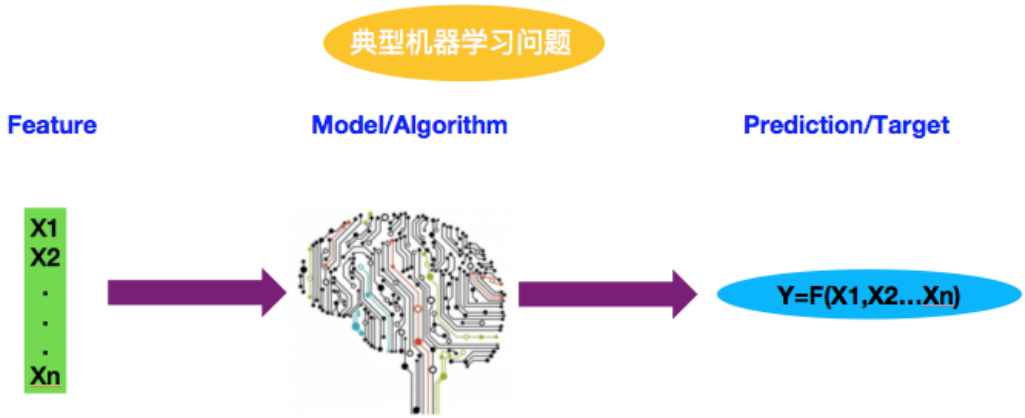
算法部分

机器学习涉及优化理论、统计学、数值计算等多个领域。这给那些希望学习机器学习核心概念的系统开发工程师带来了很大的障碍。不过，复杂的概念背后往往蕴藏着朴素的道理。本节将尝试采用通俗的例子和类比方式，来对机器学习和排序学习的一些核心概念进行揭秘。

机器学习

什么是机器学习?

典型的机器学习问题，如下图所示：

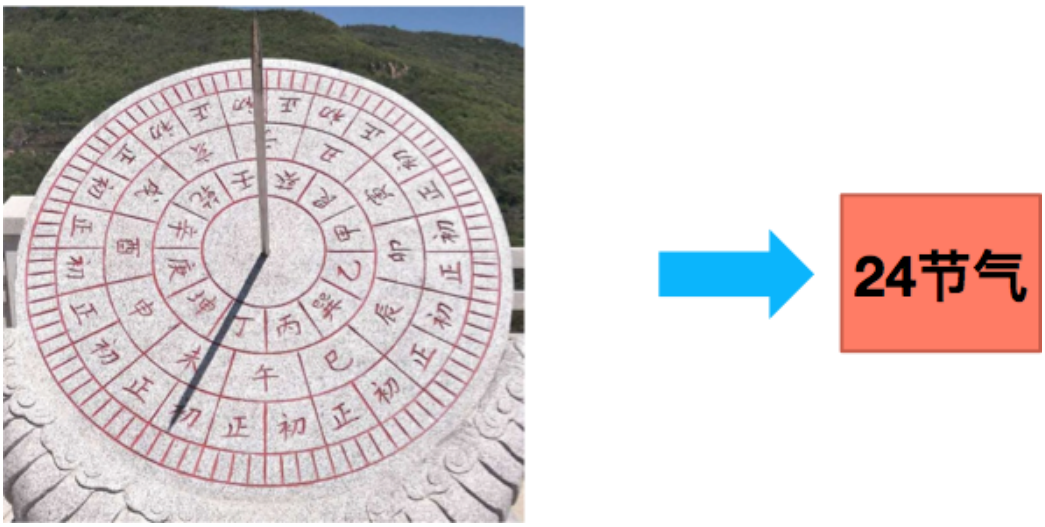


机器学习模型或算法（Model/Algorithm）会根据观察到的特征值（Feature）进行预测，给出预测结果或者目标（Prediction/Target）。这就像是一个函数计算过程，对于特定X值（Feature），算法模型就像是函数，最终的预测结果是Y值。不难理解，机器学习的核心问题就是如何得到预测函数。

Wikipedia的对机器学习定义如下：

“Machine learning is a subset of artificial intelligence in the field of computer science that often uses statistical techniques to give computers the ability to learn with data, without being explicitly programmed.”

机器学习的最重要本质是从数据中学习，得到预测函数。人类的思考过程以及判断能力本质上也是一种函数处理。从数据或者经验中学习，对于人类来说是一件再平常不过的事情了。例如人们通过观察太阳照射物体影子的长短而发明了日晷，从而具备了计时和制定节气的能力。古埃及人通过尼罗河水的涨落发明了古埃及历法。



又比如人们通过观察月亮形状的变化而发明了阴历。

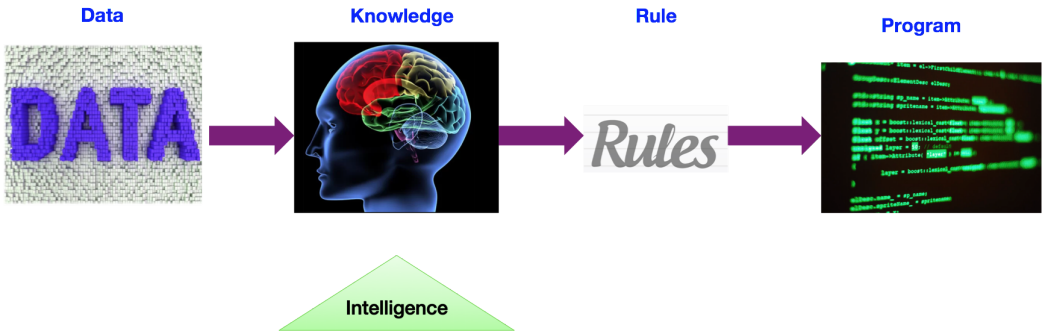
如果机器能够像人一样具备从数据中学习的能力，从某种意义上讲，就具备了一定的“智能”。现在需要回答的两个问题就是：

- 到底什么是“智能”？
- 如何让机器具备智能？

什么是智能？

在回答这个问题之前，我们先看看传统的编程模式为什么不能称之为“智能”。传统的编程模式如下图所示，它一般要经历如下几个阶段：

- 人类通过观察数据（Data）总结经验，转化成知识（Knowledge）。
- 人类将知识转化成规则（Rule）。
- 工程师将规则转化成计算机程序（Program）。



在这种编程模式下，如果一个问题被规则覆盖，那么计算机程序就能处理。对于规则不能覆盖的问题，只能由人类来重新思考，制定新规则来解决。所以在这里“智能”角色主要由人类来承担。人类负责解决新问题，所以传统的程序本身不能称之为“智能”。

所以，“智能”的一个核心要素就是“举一反三”。

如何让机器具备智能？

在讨论这个问题之前，可以先回顾一下人类是怎么掌握“举一反三”的能力的？基本流程如下：

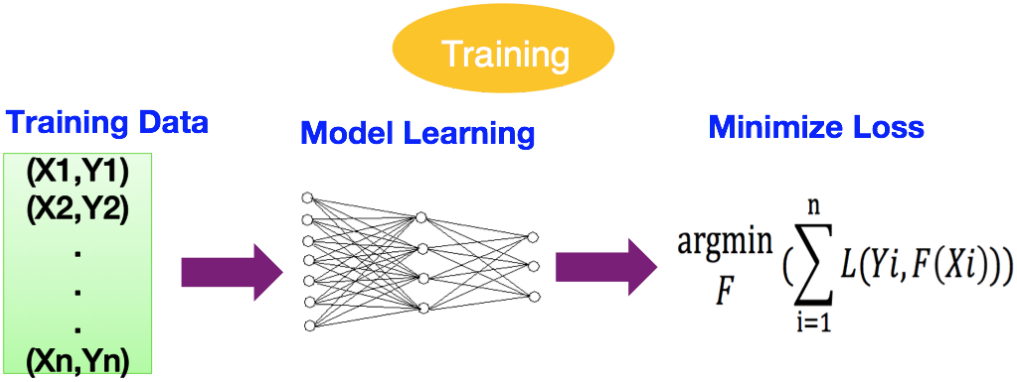
- 老师给学生一些题目，指导学生如何解题。学生努力掌握解题思路，尽可能让自己的答案和老师给出的答案一致。
- 学生需要通过一些考试来证明自己具备“举一反三”的能力。如果通过了这些考试，学生将获得毕业证书或者资格从业证书。
- 学生变成一个从业者之后将会面临并且处理很多之前没有碰到过的新问题。

机器学习专家从人类的学习过程中获得灵感，通过三个阶段让机器具备“举一反三”的能力。这三个阶段是：训练阶段（Training）、测试阶段（Testing）、推导阶段（Inference）。下面逐一进行介绍。

训练阶段

训练阶段如下图所示：

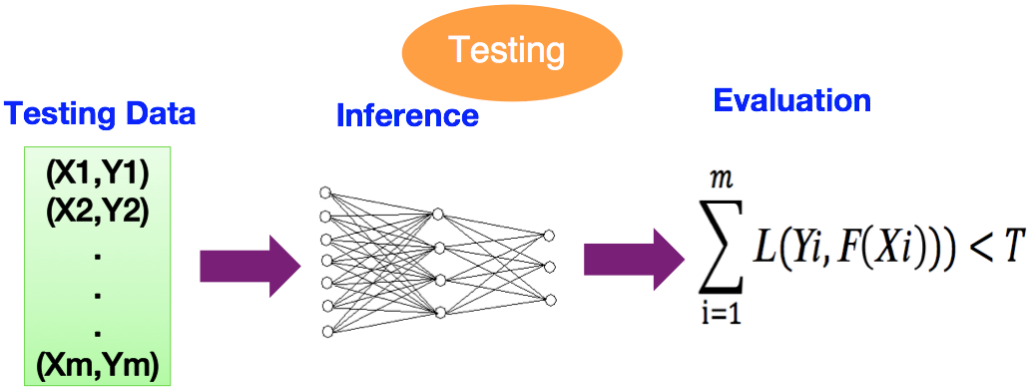
- 人类给机器学习模型一些训练样本（X，Y），X代表特征，Y代表目标值。这就好比老师教学生解题，X代表题目，Y代表标准答案。
- 机器学习模型尝试想出一种方法解题。
- 在训练阶段，机器学习的目标就是让损失函数值最小。类比学生尝试让自己解题的答案和老师给的标准答案差别最小，思路如出一辙。



测试阶段

测试阶段如下图所示：

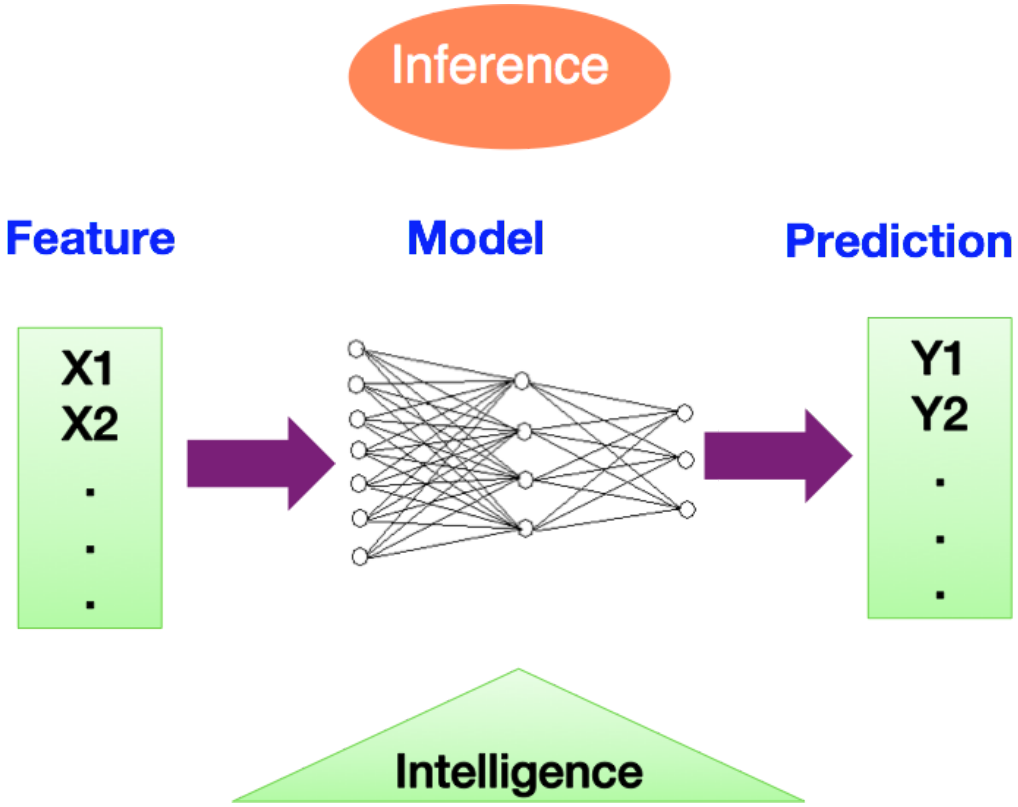
- 人类给训练好的模型一批完全不同的测试样本（X，Y）。这就好比学生拿到考试试卷。
- 模型进行推导。这个过程就像学生正在考试答题。
- 人类要求测试样本的总损失函数值低于设定的最低目标值。这就像学校要求学生的考试成绩必须及格一样。



推导阶段

推导阶段如下图所示：

- 在这个阶段机器学习模型只能拿到特征值X，而没有目标值。这就像工作中，人们只是在解决一个个的问题，但不知道正确的结果到底是什么。
- 在推导阶段，机器学习的目标就是预测，给出目标值。



排序学习

什么是排序学习?

Wikipedia的对排序学习的定义如下：

“Learning to rank is the application of machine learning, typically supervised, semi–supervised or reinforcement learning, in the construction of ranking models for information retrieval systems. Training data consists of lists of items with some partial order specified between items in each list. This order is typically induced by giving a numerical or ordinal score or a binary judgment (e.g. “relevant” or “not relevant”) for each item. The ranking model’s purpose is to rank, i.e. produce a permutation of items in new, unseen lists in a way which is “similar” to rankings in the training data in some sense.”

排序学习是机器学习在信息检索系统里的应用，其目标是构建一个排序模型用于对列表进行排序。排序学习的典型应用包括搜索列表、推荐列表和广告列表等等。

列表排序的目标是对多个条目进行排序，这就意味着它的目标值是有结构的。与单值回归和单值分类相比，结构化目标要求解决两个被广泛提起的概念：

- 列表评价指标
- 列表训练算法

列表评价指标

以关键词搜索返回文章列表为例，这里先分析一下列表评价指标要解决什么挑战。

- 第一个挑战就是定义文章与关键词之间的相关度，这决定了一篇文章在列表中的位置，相关度越高排序就应该越靠前。
- 第二个挑战是当列表中某些文章没有排在正确的位置时候，如何给整个列表打分。举个例子来说，假如对于某个关键词，按照相关性的高低正确排序，文档1、2、3、4、5应该依次排在前5位。现在的挑战就是，如何评估“2，1，3，4，5”和“1，2，5，4，3”这两个列表的优劣呢?

列表排序的评价指标体系总来的来说经历了三个阶段，分别是Precision and Recall、Discounted Cumulative Gain(DCG)和Expected Reciprocal Rank(ERR)。我们逐一进行讲解。

Precision and Recall(P-R)

本评价体系通过准确率（Precision）和召回率（Recall）两个指标来共同衡量列表的排序质量。对于一个请求关键词，所有的文档被标记为相关和不相关两种。

Precision的定义如下：

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

Recall的定义如下：

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

举个列子来说，对于某个请求关键词，有200篇文章实际相关。某个排序算法只认为100篇文章是相关的，而这100篇文章里面，真正相关的文章只有80篇。按照以上定义：

- 准确率=$^{80}/_{100}$=0.8
- 召回率=$^{80}/_{200}$=0.4。

Discounted Cumulative Gain(DCG)

P-R的有两个明显缺点：

- 所有文章只被分为相关和不相关两档，分类显然太粗糙。
- 没有考虑位置因素。

DCG解决了这两个问题。对于一个关键词，所有的文档可以分为多个相关性级别，这里以rel1，rel2…来表示。文章相关性对整个列表评价指标的贡献随着位置的增加而对数衰减，位置越靠后，衰减越严重。基于DCG评价指标，列表前p个文档的评价指标定义如下：

$$\mathrm{DCG_p} = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

对于排序引擎而言，不同请求的结果列表长度往往不相同。当比较不同排序引擎的综合排序性能时，不同长度请求之间的DCG指标的可比性不高。目前在工业界常用的是Normalized DCG(nDCG)，它假定能够获取到某个请求的前p个位置的完美排序列表，这个完美列表的分值称为Ideal DCG(IDCG)，nDCG等于DCG与IDCG比值。所以nDCG是一个在0到1之间的值。

nDCG的定义如下：

$$\mathrm{nDCG_p} = \frac{DCG_p}{IDCG_p}$$

IDCG的定义如下：

$$\mathrm{IDCG_p} = \sum_{i=1}^{|REL|} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

|REL|代表按照相关性排序好的最多到位置p的结果列表。

Expected Reciprocal Rank(ERR)

与DCG相比，除了考虑位置衰减和允许多种相关级别（以R1，R2，R3…来表示）以外，ERR更进了一步，还考虑了排在文档之前所有文档的相关性。举个例子来说，文档A非常相关，排在第5位。如果排在前面的4个文档相关度都不高，那么文档A对列表的贡献就很大。反过来，如果前面4个文档相关度很大，已经完全解决了用户的搜索需求，用户根本就不会点击第5个位置的文档，那么文档A对列表的贡献就不大。

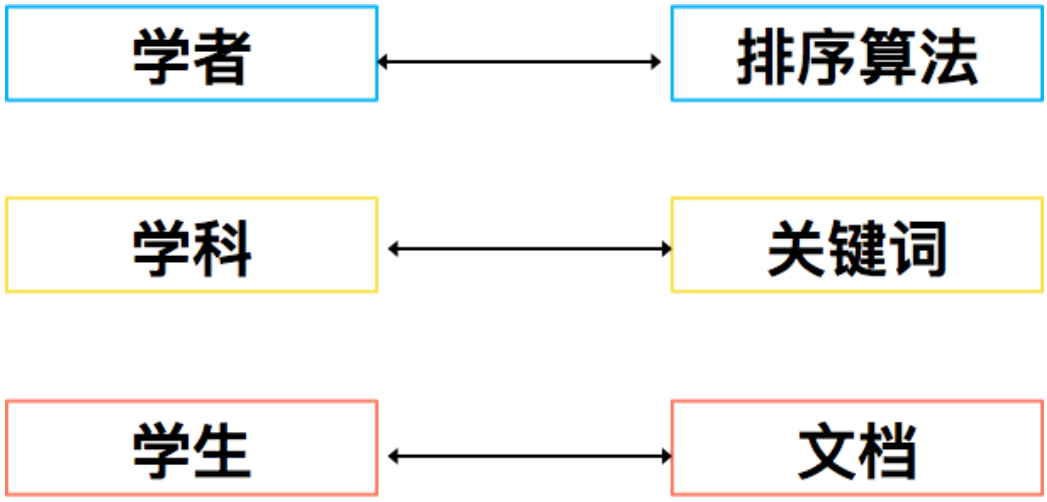
ERR的定义如下：

$$ERR = \sum_{r=1}^{n} \frac{1}{r} \prod_{i=1}^{r-1} (1-R_i) R_r$$

列表训练算法

做列表排序的工程师们经常听到诸如Pointwise、Pairwise和Listwise的概念。这些是什么东西呢，背后的原理又是什么呢？这里将逐一解密。

仍然以关键词搜索文章为例，排序学习算法的目标是为给定的关键词对文章列表进行排序。做为类比，假定有一个学者想要对各科学生排名进行预测。各种角色的对应关系如下：



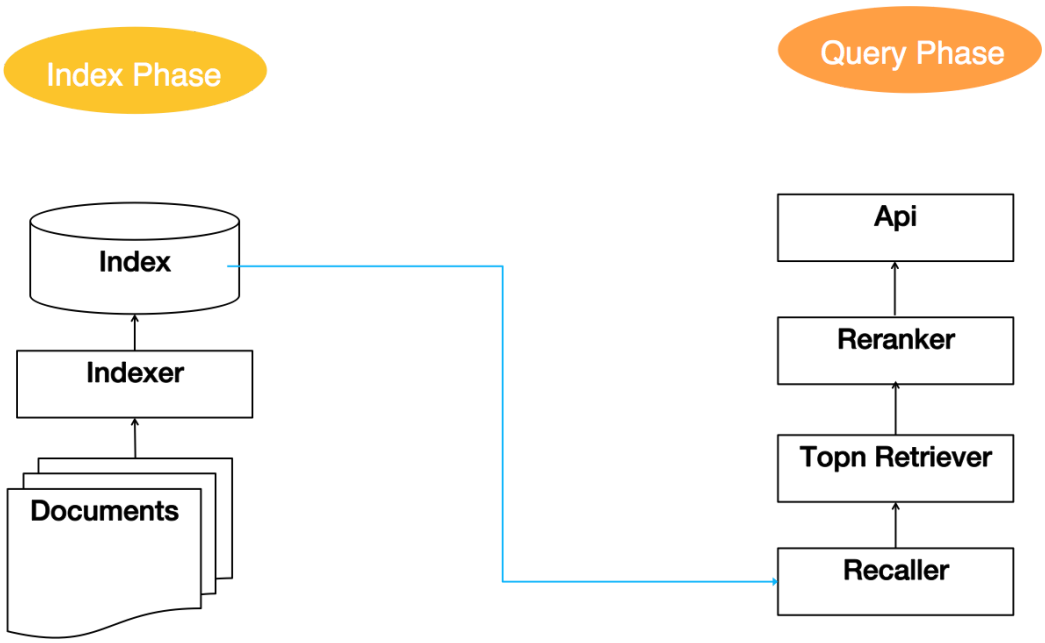
首先我们要告诉学者每个学生的各种属性，这就像我们要告诉排序算法文档特征。对于目标值，我们却有三种方式来告诉学者：

- 对于每个学科，我们可以告诉学者每个学生的成绩。比较每个学生的成绩，学者当然可以算出每个学生的最终排名。这种训练方法被称为Pointwise。对于Pointwise算法，如果最终预测目标是一个实数值，就是一个回归问题。如果目标是概率预测，这就是一个分类问题，例如CTR预估。
- 对于每个学科，我们可以告诉学者任意两个学生的相互排名。根据学生之间排名的情况，学者也可以算出每个学生的最终排名。这种训练方法被称为Pairwise。Pairwise算法的目标是减少逆序的数量，所以是个二分类问题。
- 对于每个学科，我们可以直接告诉学者所有学生的整体排名。这种训练方法被称为Listwise。Listwise算法的目标往往是直接优化nDCG、ERR等评价指标。

这三种方法表面看上去有点像玩文字游戏，但是背后却是工程师和科学家们不断探索的结果。最直观的方案是Pointwise算法，例如对于广告CTR预估，在训练阶段需要标注某个文档的点击概率，这相对来说容易。Pairwise算法一个重要分支是Lambda系列，包括LambdaRank、LambdaMart等，它的核心思想是：很多时候我们很难直接计算损失函数的值，但却很容易计算损失函数梯度（Gradient）。这意味着我们很难计算整个列表的nDCG和ERR等指标，但却很容易知道某个文档应该排的更靠前还是靠后。Listwise算法往往效果最好，但是如何为每个请求对所有文档进行标注是一个巨大的挑战。

在线排序架构

典型的信息检索包含两个阶段：索引阶段和查询阶段。这两个阶段的流程以及相互关系可以用下图来表示：



索引阶段的工作是由索引器（Indexer）读取文档（Documents）构建索引（Index）。

查询阶段读取索引做为召回，然后交给Topn Retriever进行粗排，在粗排后的结果里面将前n个文档传给Reranker进行精排。这样一个召回、粗排、精排的架构最初是由Google提出来的，也被称为“Two-Phase Scheme”。

索引部分属于离线阶段，这里重点讲述在线排序阶段，即查询阶段。

三大挑战

在线排序架构主要面临三方面的挑战：特征、模型和召回。

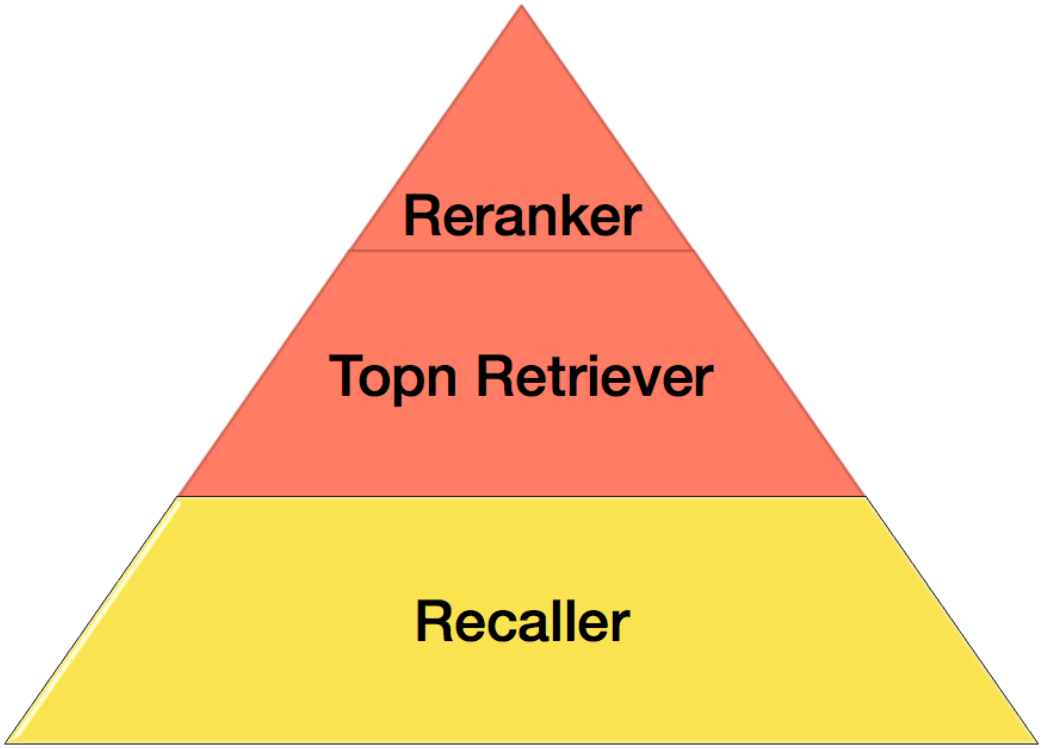
- 特征挑战包括特征添加、特征算子、特征归一化、特征离散化、特征获取、特征服务治理等。
- 模型挑战包括基础模型完备性、级联模型、复合目标、A/B实验支持、模型热加载等。
- 召回挑战包括关键词召回、LBS召回、推荐召回、粗排召回等。

三大挑战内部包含了非常多更细粒度的挑战，孤立地解决每个挑战显然不是好思路。在线排序作为一个被广泛使用的架构值得采用领域模型进行统一解决。Domain-driven design(DDD)的三个原则分别是：领域聚焦、边界清晰、持续集成。

基于以上分析，我们构建了三个在线排序领域模型：召回治理、特征服务治理和在线排序分层模型。

召回治理

经典的Two-Phase Scheme架构如下图所示，查询阶段应该包含：召回、粗排和精排。但从领域架构设计的角度来讲，粗排对于精排而言也是一种召回。和基于传统的文本搜索不同，美团点评这样的O2O公司需要考虑地理位置和距离等因素，所以基于LBS的召回也是一种召回。与搜索不同，推荐召回往往基于协同过滤来完成的，例如User-Based CF和Item-Based CF。

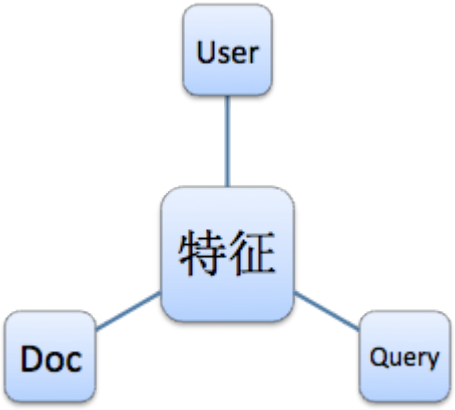


综上所述，召回总体而言分成四大类：

- 关键词召回，我们采用Elasticsearch解决方案。
- 距离召回，我们采用K-D tree的解决方案。
- 粗排召回。
- 推荐类召回。

特征服务治理

传统的视角认为特征服务应该分为用户特征（User）、查询特征（Query）和文档特征（Doc），如下图：

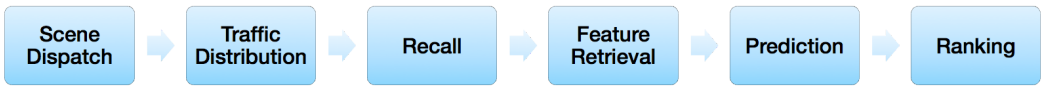


这是比较纯粹的业务视角，并不满足DDD的领域架构设计思路。由于特征数量巨大，我们没有办法为每个特征设计一套方案，但是我们可以把特征进行归类，为几类特征分别设计解决方案。每类技术方案需要统一考虑性能、可用性、存储等因素。从领域视角，特征服务包含四大类：

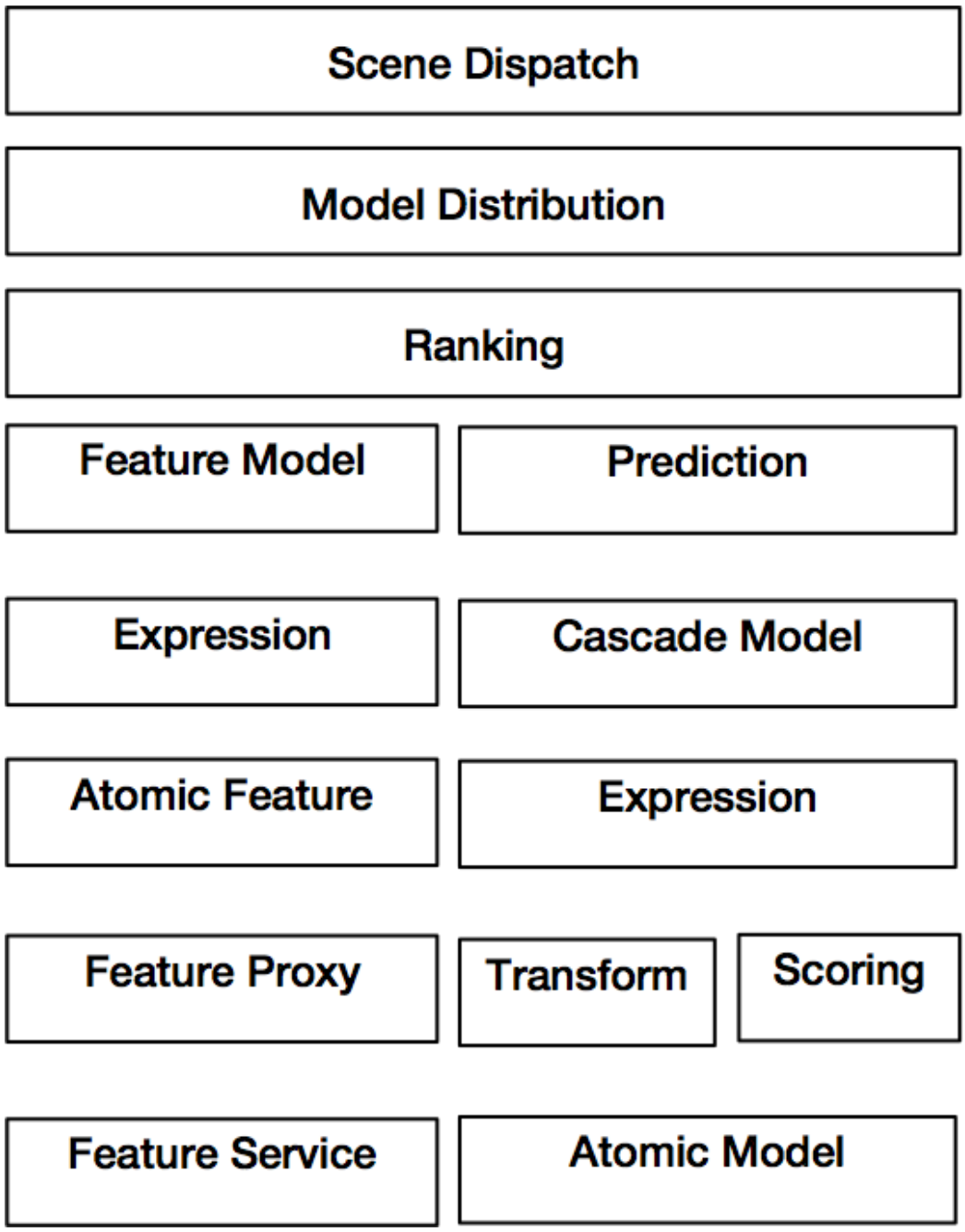
- 列表类特征。一次请求要求返回实体列表特征，即多个实体，每个实体多个特征。这种特征建议采用内存列表服务，一次性返回所有请求实体的所有特征。避免多次请求，从而导致数量暴增，造成系统雪崩。
- 实体特征。一次请求返回单实体的多个特征。建议采用Redis、Tair等支持多级的Key-Value服务中间件提供服务。
- 上下文特征。包括召回静态分、城市、Query特征等。这些特征直接放在请求内存里面。
- 相似性特征。有些特征是通过计算个体和列表、列表和列表的相似度而得到的。建议提供单独的内存计算服务，避免这类特征的计算影响在线排序性能。本质上这是一种计算转移的设计。

在线排序分层模型

如下图所示，典型的排序流程包含六个步骤：场景分发（Scene Dispatch）、流量分配（Traffic Distribution）、召回（Recall）、特征抽取（Feature Retrieval）、预测（Prediction）、排序（Ranking）等等。

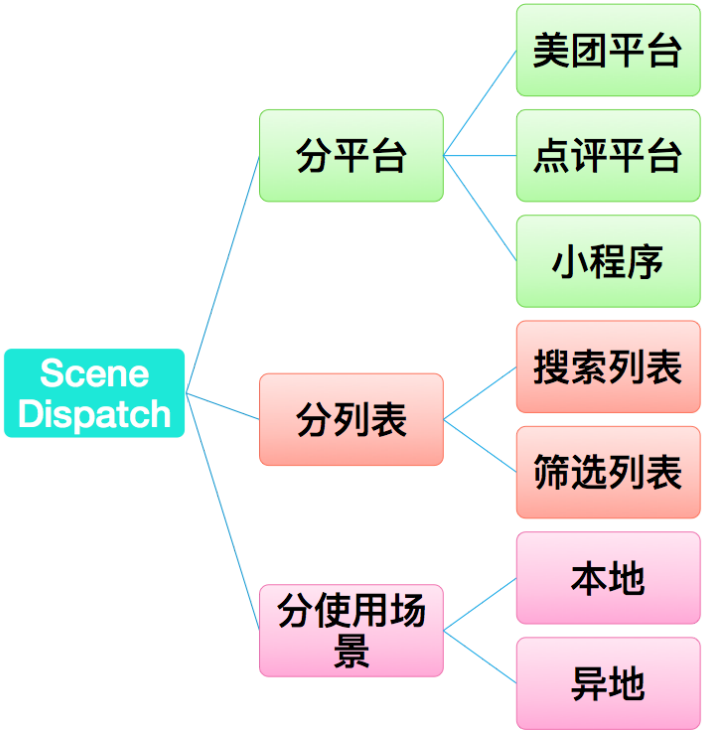


按照DDD的设计原则，我们设计了如下在线排序分层模型，包括：场景分发（Scene Dispatch）、模型分发（Model Distribution）、排序（Ranking）、特征管道（Feature Pipeline）、预测管道（Prediction Pipeline）。我们将逐一进行介绍。



场景分发（Scene Dispatch）

场景分发一般是指业务类型的分发。对于美团点评而言包括：分平台、分列表、分使用场景等。如下图所示：



模型分发（Model Distribution）

模型分发的目标是把在线流量分配给不同的实验模型，具体而言要实现三个功能：

- 为模型迭代提供在线流量，负责线上效果收集、验证等。
- A/B测试，确保不同模型之间流量的稳定、独立和互斥、确保效果归属唯一。
- 确保与其他层的实验流量的正交性。

流量的定义是模型分发的一个基础问题。典型的流量包括：访问、用户和设备。

如何让一个流量稳定地映射到特定模型上面，流量之间是否有级别？这些是模型分发需要重点解决的问题。

流量分桶原理

采用如下步骤将流量分配到具体模型上面去：

- 把所有流量分成N个桶。
- 每个具体的流量Hash到某个桶里面去。
- 给每个模型一定的配额，也就是每个策略模型占据对应比例的流量桶。
- 所有策略模型流量配额总和为100%。
- 当流量和模型落到同一个桶的时候，该模型拥有该流量。

举个例子来说，如上图所示，所有流量分为32个桶，A、B、C三个模型分别拥有37.5%、25%和37.5%的配额。对应的，A、B、C应该占据12、8和12个桶。

为了确保模型和流量的正交性，模型和流量的Hash Key采用不同的前缀。

流量分级

每个团队的模型分级策略并不相同，这里只给出一个建议模型流量分级：

- 基线流量。本流量用于与其他流量进行对比，以确定新模型的效果是否高于基准线，低于基准线的模型要快速下线。另外，主力流量相对基线流量的效果提升也是衡量算法团队贡献的重要指标。
- 实验流量。该流量主要用于新实验模型。该流量大小设计要注意两点：第一不能太大而伤害线上效果；第二不能太小，流量太小会导致方差太大，不利于做正确的效果判断。
- 潜力流量。如果实验流量在一定周期内效果比较好，可以升级到潜力流量。潜力流量主要是要解决实验流量方差大带来的问题。
- 主力流量。主力流量只有一个，即稳定运行效果最好的流量。如果某个潜力流量长期好于其他潜力流量和主力流量，就可以考虑把这个潜力流量升级为主力流量。

做实验的过程中，需要避免新实验流量对老模型流量的冲击。流量群体对于新模型会有一定的适应期，而适应期相对于稳定期的效果一般会差一点。如果因为新实验的上线而导致整个流量群体的模型都更改了，从统计学的角度讲，模型之间的对比关系没有变化。但这可能会影响整个大盘的效果，成本很高。

为了解决这个问题，我们的流量分桶模型优先为模型列表前面的模型分配流量，实验模型尽量放在列表尾端。这样实验模型的频繁上下线不影响主力和潜力流量的用户群体。当然当发生模型流量升级的时候，很多流量用户的服务模型都会更改。这种情况并不是问题，因为一方面我们在尝试让更多用户使用更好的模型，另一方面固定让一部分用户长期使用实验流量也是不公平的事情。

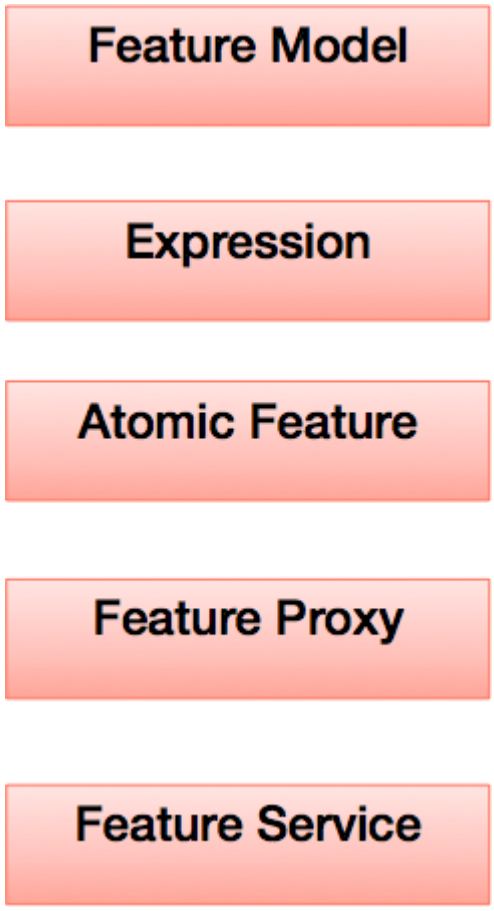
排序（Ranking）

排序模块是特征模块和预测模块的容器，它的主要职责如下：

- 获取所有列表实体进行预测所需特征。
- 将特征交给预测模块进行预测。
- 对所有列表实体按照预测值进行排序。

特征管道（Feature Pipeline）

特征管道包含特征模型（Feature Model）、表达式（Expression）、原子特征（Atomic Feature）、特征服务代理（Feature Proxy）、特征服务（Feature Service）。如下图所示：

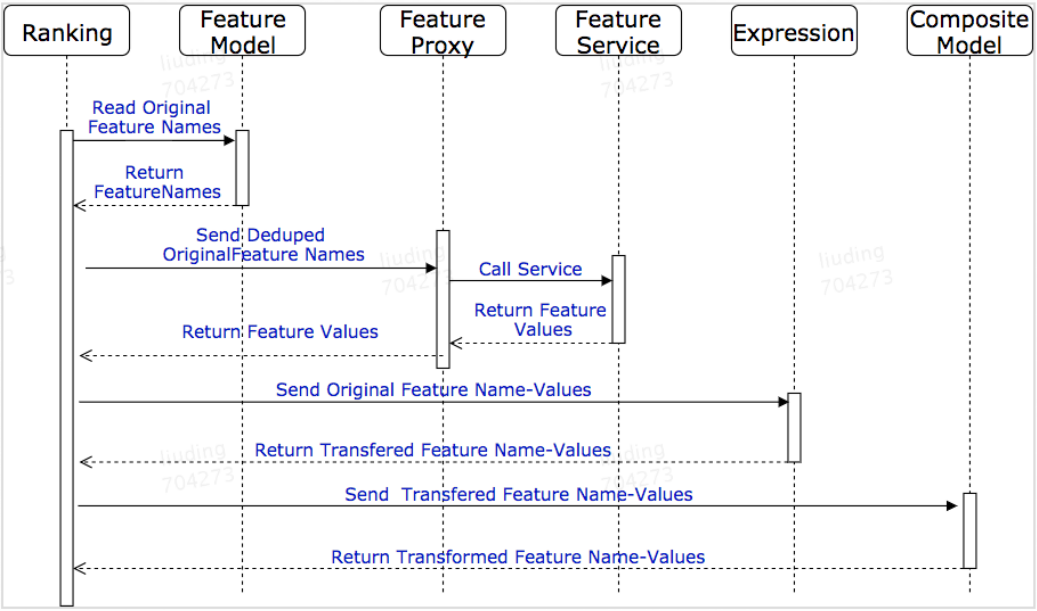


特征管道要解决两个核心问题：

- 给定特征名获取对应特征值。这个过程很复杂，要完成特征名->特征服务->特征类->特征值的转化过程。
- 特征算子问题。模型使用的特征往往是对多个原子特征进行复合运算后的结果。另外，特征离散化、归一化也是特征算子需要解决的问题。

完整的特征获取流程如下图所示，具体的流程如下：

- Ranking模块从FeatureModel里面读取所有的原始特征名。
- Ranking将所有的原始特征名交给Feature Proxy。
- Feature Proxy根据特征名的标识去调用对应的Feature Service，并将原始特征值返回给Ranking模块。
- Ranking模块通过Expression将原始特征转化成复合特征。
- Ranking模块将所有特征交给级联模型做进一步的转换。



特征模型（Feature Model）

我们把所有与特征获取和特征算子的相关信息都放在一个类里面，这个类就是FeatureModel，定义如下：

```
//包含特征获取和特征算子计算所需的meta信息
public class FeatureModel {
    //这是真正用在Prediction里面的特征名
    private String featureName;
    //通过表达式将多种原子特征组合成复合特征。
    private IExpression expression;
    //这些特征名是真正交给特征服务代理(Feature Proxy)去从服务端获取特征值的特征名集合。
    private Set<String> originalFeatureNames;
    //用于指示特征是否需要被级联模型转换
    private boolean isTransformedFeature;
    //是否为one-hot特征
    private boolean isOneHotIdFeature;
    //不同one-hot特征之间往往共享相同的原始特征，这个变量>用于标识原始特征名。
    private String oneHotIdKey;
    //表明本特征是否需要归一化
    private boolean isNormalized;
}
```

表达式（Expression）

表达式的目的是为了将多种原始特征转换成一个新特征，或者对单个原始特征进行运算符转换。我们采用前缀法表达式（Polish Notation）来表示特征算子运算。例如表达式(5-6)*7的前缀表达式为* – 5 6 7。

复合特征需要指定如下分隔符：

- 复合特征前缀。区别于其他类型特征，我们以“$”表示该特征为复合特征。
- 表达式各元素之间的分隔符，采用“_”来标识。
- 用“O”表示运算符前缀。
- 用“C”表示常数前缀。
- 用“V”表示变量前缀。

例如：表达式v1 + 14.2 + (2*(v2+v3)) 将被表示为 $O+_O+_Vv1_C14.2_O*_C2_O+_Vv2_Vv3

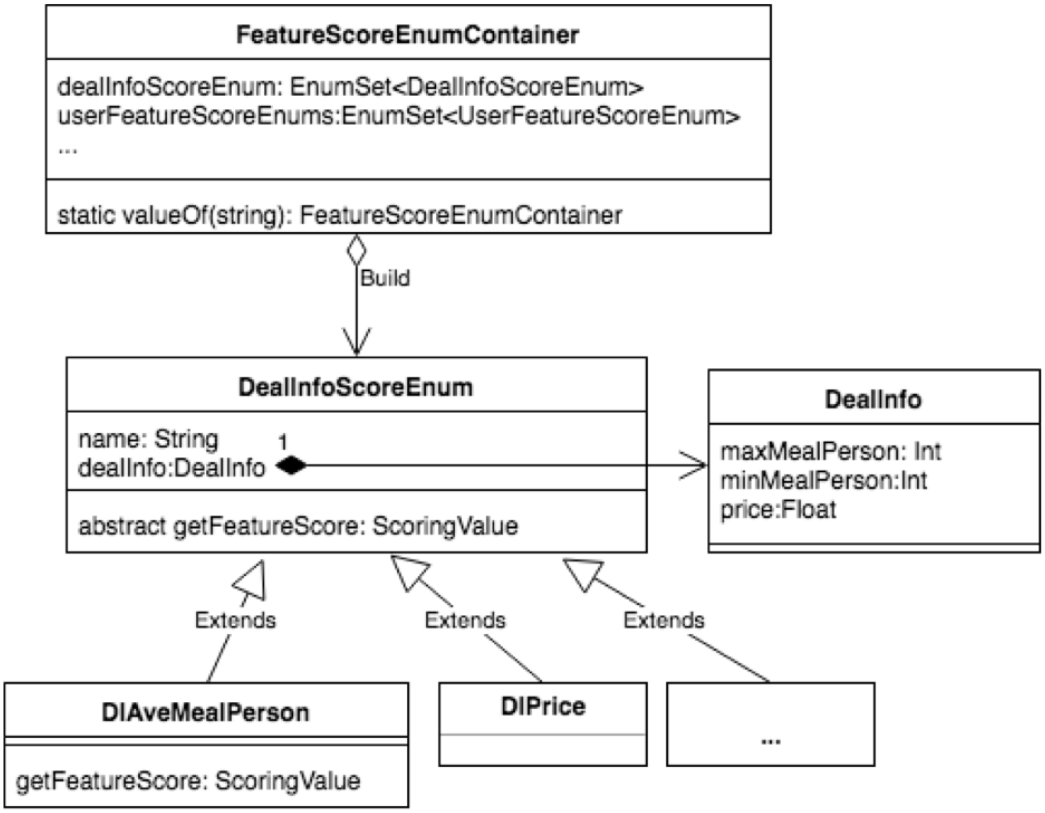
原子特征（Atomic Feature）

原子特征（或者说原始特征）包含特征名和特征值两个部分。原子特征的读取需要由4种实体类共同完成：

- POJO用于存储特征原始值。例如DealInfo保存了所有与Deal实体相关的特征值，包括Price、maxMealPerson、minMealPerson等等。
- ScoringValue用于存储从POJO中返回的特征值。特征值包含三种基本类型，即数量型（Quantity）、序数型（Ordinal）、类别型（Categorical）。
- ScoreEnum实现特征名到特征值的映射。每类原子特征对应于一个ScoreEnum类，特征名通过反射（Reflection）的方式来构建对应的ScoreEnum类。ScoreEnum类和POJO一起共同实现特征值的读取。
- FeatureScoreEnumContainer用于保存一个算法模型所需所有特征的ScoreEnum。

一个典型的例子如下图所示：

- DealInfo是POJO类。
- DealInfoScoreEnum是一个ScoreEnum基类。对应于平均用餐人数特征、价格等特征，我们定义了DlAveMealPerson和DlPrice的具体ScoreEnum类。
- FeatureScoreEnumContainer用于存储某个模型的所有特征的ScoreEnum。



复杂系统设计需要充分的利用语言特性和设计模式。建议的优化点有三个：

- 为每个原子特征定义一个ScoreEnum类会导致类数量暴增。优化方式是ScoreEnum基类定义为Enum类型，每个具体特征类为一个枚举值，枚举值继承并实现枚举类的方法。
- FeatureScoreEnumContainer采用Build设计模式将一个算法模型的所需特征转换成ScoreEnum集合。
- ScoreEnum从POJO类中读取具体特征值采用的是Command模式。

这里稍微介绍一下Command设计模式。Command模式的核心思想是需求方只要求拿到相关信息，不关心谁提供以及怎么提供。具体的提供方接受需求方的需求，负责将结果交给需求方。

在特征读取里面，需求方是模型，模型仅仅提供一个特征名（FeatureName），不关心怎么读取对应的特征值。具体的ScoreEnum类是具体的提供方，具体的ScoreEnum从POJO里面读取特定的特征值，并转换成ScoringValue交给模型。

特征服务代理（Feature Proxy）

特征服务代理负责远程特征获取实施，具体的过程包括：

- 每一大类特征或者一种特征服务有一个FeatureProxy，具体的FeatureProxy负责向特征服务发起请求并获取POJO类。
- 所有的FeatureProxy注册到FeatureServiceContainer类。
- 在具体的一次特征获取中，FeatureServiceContainer根据FeatureName的前缀负责将特征获取分配到不同的FeatureProxy类里面。
- FeatureProxy根据指定的实体ID列表和特征名从特征服务中读取POJO列表。只有对应ID的指定特征名（keys）的特征值才会被赋值给POJO。这就最大限度地降低了网络读取的成本。

预测管道（Prediction Pipeline）

预测管道包含：预测（Prediction）、级联模型（Cascade Model）、表达式（Expression）、特征转换（Transform）、计分（Scoring）和原子模型（Atomic Model）。

预测（Prediction）

预测本质上是对模型的封装。它负责将每个列表实体的特征转化成模型需要的输入格式，让模型进行预测。

级联模型（Cascade Model）

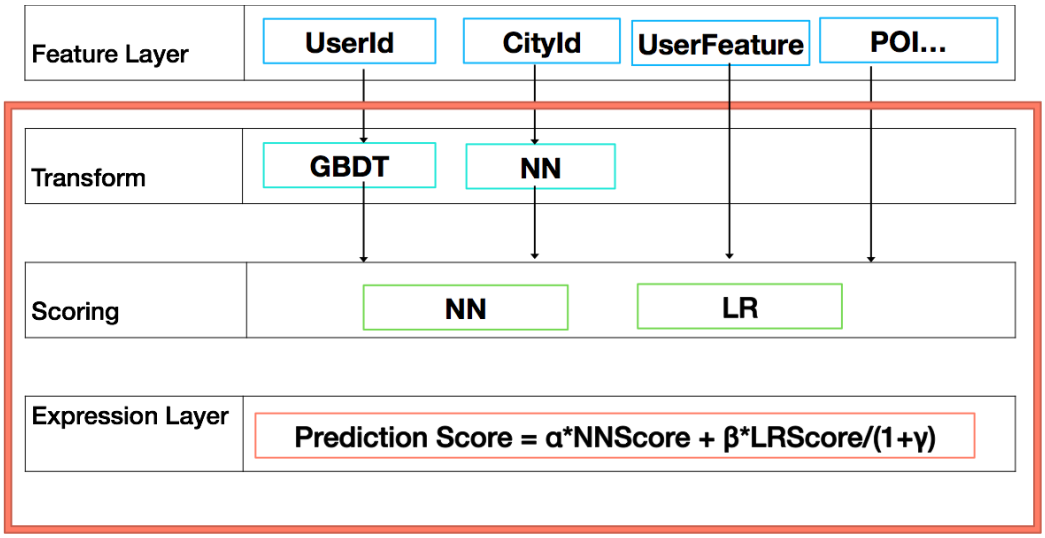
我们构建级联模型主要是基于两方面的观察：

- 基于Facebook的《Practical Lessons from Predicting Clicks on Ads at Facebook》的Xgboost+LR以及最近很热门的Wide&Deep表明，对一些特征，特别是ID类特征通过树模型或者NN模型进行转化，把转化后的值做为特征值交给预测模型进行预测，往往会能实现更好的效果。
- 有些训练目标是复合目标，每个子目标需要通过不同的模型进行预测。这些子目标之间通过一个简单的表达式计算出最终的预测结果。

举例如下图所示，我们自上而下进行讲解：

- 该模型有UserId、CityId、UserFeature、POI等特征。
- UserId和CityId特征分别通过GBDT和NN模型进行转换（Transform）。
- 转换后的特征和UserFeature、POI等原始特征一起交给NN和LR模型进行算分（Scoring）。

- 最终的预测分值通过表达式Prediction Score = α*NNScore + β*LRScore/(1+γ)来完成。表达式中的 α 、β和γ是事先设置好的值。



原子模型（Atomic Model）

在这里原子模型指的是一种原子计算拓扑结构，比如线性模型、树模型和网络模型。

常用的模型像Logistic Regression和Linear Regression都是线性模型。GBDT、Random Forest都是树模型。MLP、CNN、RNN都是网络模型。

这里定义的原子模型主要的目的是为了工程实施的便利。一个模型被认定为原子模型有如下两个原因：

- 该模型经常做为独立预测模型被使用。
- 该模型有比较完整的实现代码。

总结

本文总结了作者在美团点评解决到店餐饮个性化信息展示的相关经验，从算法和架构两方面进行阐述。在算法部分，文章采用通俗的例子和类比方式进行讲解，希望非算法工程师能够理解关键的算法概念。架构部分比较详细地阐述了到店餐饮的排序架构。

根据我们所掌握的知识，特征治理和召回治理的思路是一种全新的视角，这对于架构排序系统设计有很大的帮助。这种思考方式同样也适用于其他领域模型的构建。与Google提供的经典Two-Phase Scheme架构相比，在线排序分层模型提供了更细颗粒度的抽象原型。该原型细致的阐述了包括分流、A/B测试、特征获取、特征算子、级联模型等一系列经典排序架构问题。同时该原型模型由于采用了分层和层内功能聚焦的思路，所以它比较完美地体现了DDD的三大设计原则，即领域聚焦、边界清晰、持续集成。

作者简介

- 刘丁，目前负责到店餐饮算法策略方向，推进AI在到店餐饮各领域的应用。2014年加入美团，先后负责美团推荐系统、智能筛选系统架构、美团广告系统的架构和上线、完成美团广告运营平台的搭建。曾就职于Amazon、TripAdvisor等公司。

参考文章：

- [1]Gamma E, Helm R, Johnson R, et al. Design Patterns-Elements of Reusable Object-Oriented Software. Machinery Industry, 2003
- [2]Wikipedia,Learning to rank🔗.
- [3]Wikipedia,Machine learning🔗.
- [4]Wikipedia,Precision and recall🔗.
- [5]Wikipedia,Discounted cumulative gain🔗.
- [6]Wikipedia,Domain-driven design🔗.
- [7]Wikipedia,Elasticsearch🔗.
- [8]Wikipedia,k-d tree🔗.
- [9]百度百科,太阳历🔗.
- [10]百度百科,阴历🔗.
- [11]Xinran H, Junfeng P, Ou J, et al. Practical Lessons from Predicting Clicks on Ads at Facebook🔗
- [12]Olivier C, Donald M, Ya Z, Pierre G. Expected Reciprocal Rank for Graded Relevance🔗
- [13]Heng-Tze C, Levent K, et al. Wide & Deep Learning for Recommender Systems🔗

每天数百亿用户行为数据，美团点评怎么实现秒级转化分析?

作者: 业锐

背景

用户行为分析是数据分析中非常重要的一项内容，在统计活跃用户，分析留存和转化率，改进产品体验、推动用户增长等领域有重要作用。美团点评每天收集的用户行为日志达到数百亿条，如何在海量数据集上实现对用户行为的快速灵活分析，成为一个巨大的挑战。为此，我们提出并实现了一套面向海量数据的用户行为分析解决方案，将单次分析的耗时从小时级降低到秒级，极大的改善了分析体验，提升了分析人员的工作效率。

本文以有序漏斗的需求为例，详细介绍了问题分析和思路设计，以及工程实现和优化的全过程。本文根据2017年12月ArchSummit北京站演讲整理而成，略有删改。

问题分析

下图描述了转化率分析中一个常见场景，对访问路径“首页–搜索–菜品–下单–支付”做分析，统计按照顺序访问每层节点的用户数，得到访问过程的转化率。

统计上有一些维度约束，比如日期，时间窗口（整个访问过程在规定时间内完成，否则统计无效），城市或操作系统等，因此这也是一个典型的OLAP分析需求。此外，每个访问节点可能还有埋点属性，比如搜索页上的关键词属性，支付页的价格属性等。从结果上看，用户数是逐层收敛的，在可视化上构成了一个漏斗的形状，因此这一类需求又称之为“有序漏斗”。

问题

这类分析通常是基于用户行为的日志表上进行的，其中每行数据记录了某个用户的一次事件的相关信息，包括发生时间、用户ID、事件类型以及相关属性和维度信息等。现在业界流行的通常有两种解决思路。

1. 基于Join的SQL

```
select count (distinct t1.id1), count (distinct t2.id2), count (distinct t3.id3)
from (select uuid id1, timestamp ts1 from data where timestamp >= 1510329600 and ti
mestamp < 1510416000 and page = '首页') t1
left join
(select uuid id2, timestamp ts2 from data where timestamp >= 1510329600 and timesta
mp < 1510416000 and page = '搜索' and keyword = '中餐') t2
on t1.id1 = t2.id2 and t1.ts1 < t2.ts2 and t2.ts2 - t1.ts1 < 3600
left join
(select uuid id3, timestamp ts3 from data where timestamp >= 1510329600 and timesta
mp < 1510416000 and page = '菜品') t3
on t1.id1 = t3.id3 and t2.ts2 < t3.ts3 and t1.ts1 < t3.ts3 and t3.ts3 - t1.ts1 < 36
00
```

2. 基于UDAF(User Defined Aggregate Function)的SQL

```
select
funnel(timestamp, 3600, '首页') stage0,
funnel(timestamp, 3600, '首页', '搜索', keyword = '中餐') stage1, funnel(timestamp, 3600, '首页', '搜索', '菜品') stage2
from data
where timestamp >= 1510329600 and timestamp < 1510416000 group by uuid
```

对于第一种解法，最大的问题是需要做大量join操作，而且关联条件除了ID的等值连接之外，还有时间戳的非等值连接。当数据规模不大时，这种用法没有什么问题。但随着数据规模越来越大，在几百亿的数据集上做join操作的代价非常高，甚至已经不可行。

第二种解法有了改进，通过聚合的方式避免了join操作，改为对聚合后的数据通过UDAF做数据匹配。这种解法的问题是没有足够的筛选手段，这意味着几亿用户对应的几亿条数据都需要遍历筛选，在性能上也难以接受。

那么这个问题的难点在哪里？为什么上述两个解法在实际应用中变得越来越不可行？主要问题有这么几点。

1. **事件匹配有序列关系**。如果没有序列关系就非常容易，通过集合的交集并集运算即可。
2. **时间窗口约束**。这意味着事件匹配的时候还有最大长度的约束，所以匹配算法的复杂度会进一步提升。
3. **属性和维度的需求**。埋点SDK提供给各个业务线，每个页面具体埋什么内容，完全由业务决定，而且取值是完全开放的，因此目前属性基数已经达到了百万量级。同时还有几十个维度用于筛选，有些维度的基数也很高。
4. **数据规模**。目前每天收集到的用户行为日志有几百亿条，对资源和效率都是很大的挑战。

基于上述难点和实际需求的分析，可以总结出几个实际困难，称之为“坏消息”。

1. **漏斗定义完全随机**。不同分析需求对应的漏斗定义完全不同，包括具体包含哪些事件，这些事件的顺序等，这意味着完全的预计算是不可能的。
2. **附加OLAP需求**。除了路径匹配之外，还需要满足属性和维度上一些OLAP的上卷下钻的需求。
3. **规模和性能的矛盾**。一方面有几百亿条数据的超大规模，另一方面又追求秒级响应的交互式分析效率，这是一个非常激烈的矛盾冲突。

另一方面，还是能够从问题的分析中得到一些“好消息”，这些也是在设计和优化中可以利用的点。

1. **计算需求非常单一**。这个需求最终需要的就是去重计数的结果，这意味着不需要一个大而全的数据引擎，在设计上有很大的优化空间。

2. **并发需求不高**。漏斗分析这类需求一般由运营或者产品同学手动提交，查询结果用于辅助决策，因此并发度不会很高，这样可以在一次查询时充分调动整个集群的资源。
3. **数据不可变**。所谓日志即事实，用户行为的日志一旦收集进来，除非bug等原因一般不会再更新，基于此可以考虑一些索引类的手段来加速查询。
4. **实际业务特点**。最后是对实际业务观察得出的结论，整个漏斗收敛非常快，比如首页是几千万甚至上亿的结果，到了最下层节点可能只有几千，因此可以考虑一些快速过滤的方法来降低查询计算和数据IO的压力。

如果用一句话总结这个问题的核心本质，那就是“**多维分析**和**序列匹配**基础上的**去重计数**”。具体来说，最终结果就是每层节点符合条件的UUID有多少个，也就是去重后的计数值。这里UUID要符合两个条件，一是符合维度的筛选，二是事件序列能匹配漏斗的定义。去重计数是相对好解的问题，那么问题的重点就是如果快速有效的做维度筛选和序列匹配。

算法设计

下图是部分行为日志的数据，前面已经提到，直接在这样的数据上做维度筛选和序列匹配都是很困难的，因此考虑如何对数据做预处理，以提高执行效率。

UUID	timestamp	page	city	keyword
AAA	100	首页	北京	
AAA	102	搜索页	北京	中餐
AAA	130	菜品页	北京	
BBB	102	首页	北京	
BBB	103	首页	北京	
BBB	140	搜索页	北京	西餐
CCC	101	首页	上海	
CCC	110	菜品页	上海	
CCC	151	搜索页	上海	中餐

数据1

很自然的想法是基于UUID做聚合，根据时间排序，这也是前面提到的UDAF的思路，如下图所示。这里的问题是没有过滤的手段，每个UUID都需要遍历，成本很高。

UUID	event1	event2	event3	..... event n
AAA	ts = 100 page = 首页 city = 北京	ts = 102 page = 搜索页 city = 北京 keyword = 中餐	ts = 130 page = 菜品页 city = 北京	
BBB	ts = 102 page = 首页 city = 北京	ts = 103 page = 首页 city = 北京	ts = 140 page = 搜索页 city = 北京 keyword = 西餐	
CCC	ts = 101 page = 首页 city = 上海	ts = 110 page = 菜品页 city = 上海	ts = 151 page = 搜索页 city = 上海 keyword = 中餐	

数据2

再进一步，为了更快更方便的做过滤，考虑把维度和属性抽出来构成Key，把对应的UUID和时间戳组织起来构成value。如果有搜索引擎经验的话，很容易看出来这非常像倒排的思路。

key	value1	value2	value3	..... value n
page = 首页	UUID = AAA ts = 100	UUID = BBB ts = 102	UUID = CCC ts = 101	UUID = BBB ts = 103
page = 搜索页	UUID = AAA ts = 102	UUID = BBB ts = 140	UUID = CCC ts = 151	
page = 菜品页	UUID = CCC ts = 110	UUID = AAA ts = 130		
city = 北京	UUID = AAA ts = 100	UUID = AAA ts = 102	UUID = BBB ts = 102	UUID = BBB ts = 103
city = 上海	UUID = CCC ts = 101	UUID = CCC ts = 110	UUID = CCC ts = 151	
keyword = ...				

数据3

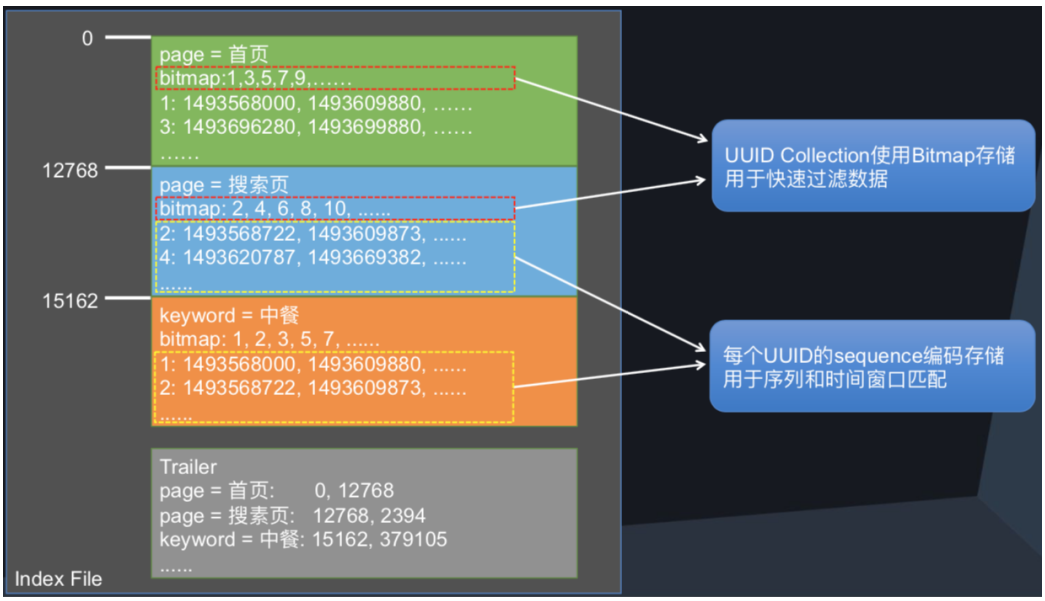
这个数据结构还是存在问题。比如说要拿到某个Key对应的UUID列表时，需要遍历所有的value才可以。再比如做时间序列的匹配，这里的时间戳信息被打散了，实际处理起来更困难。因此还可以在此基础上再优化。

可以看到优化后的Key内容保持不变，value被拆成了UUID集合和时间戳序列集合这两部分，这样的好处有两点：一是可以做快速的UUID筛选，通过Key对应的UUID集合运算就可以达成；二是在做时间序列匹配时，对于匹配算法和IO效率都是很友好的，因为时间戳是统一连续存放的，在处理时很方便。

key	UUID collection	sequence
page = 首页	AAA,BBB,CCC	AAA(100), BBB(102,103), CCC(101)
page = 搜索页	AAA,BBB,CCC	AAA(102), BBB(140), CCC(151)
page = 菜品页	AAA,CCC	AAA(130),CCC(110)
city = 北京	AAA,BBB	AAA(100,102,130), BBB(102,103,140)
city = 上海	CCC	CCC(101,110,151)
keyword = ...		

数据4

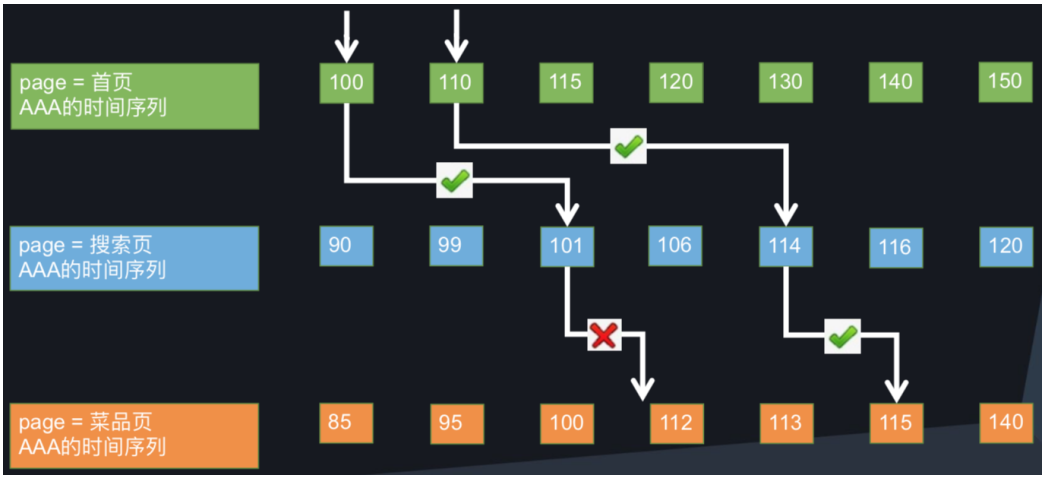
基于上述的思路，最终的索引格式如下图所示。这里每个色块对应了一个索引的block，其中包括三部分内容，一是属性名和取值；二是对应的UUID集合，数据通过bitmap格式存储，在快速筛选时效率很高；三是每个UUID对应的时间戳序列，用于序列匹配，在存储时使用差值或变长编码等一些编码压缩手段提高存储效率。



索引格式

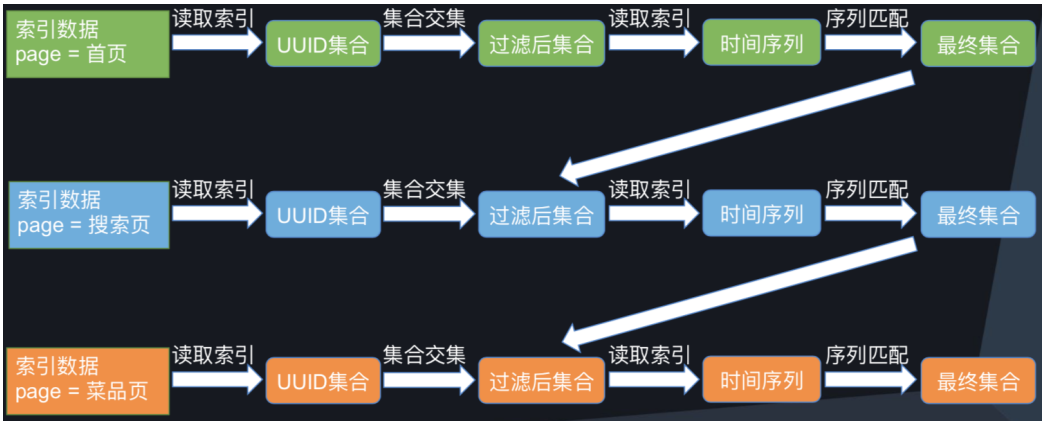
在实际应用中，通常会同时指定多个属性或维度条件，通过AND或OR的条件组织起来。这在处理时也很简单，通过语法分析可以把查询条件转为一颗表达树，树上的叶子节点对应的是单个索引数据，非叶子节点就是AND或OR类型的索引，通过并集或交集的思路做集合筛选和序列匹配即可。

上面解决的是维度筛选的问题，另一个序列匹配的问题相对简单很多。基于上述的数据格式，读取UUID对应的每个事件的时间戳序列，检查是否能按照顺序匹配即可。需要注意的是，由于存在最大时间窗口的限制，匹配算法中需要考虑回溯的情况，下图展示了一个具体的例子。在第一次匹配过程中，由于第一层节点的起始时间戳为100，并且时间窗口为10，所以第二层节点的时间戳101符合要求，但第三层节点的时间戳112超过了最大截止时间戳110，因此只能匹配两层节点，但通过回溯之后，第二次可以完整的匹配三层节点。



匹配算法

通过上述的讨论和设计，完整的算法如下图所示。其中的核心要点是先通过UUID集合做快速的过滤，再对过滤后的UUID分别做时间戳的匹配，同时上一层节点输出也作为下一层节点的输入，由此达到快速过滤的目的。



算法设计

工程实现和优化

有了明确的算法思路，接下来再看看工程如何落地。

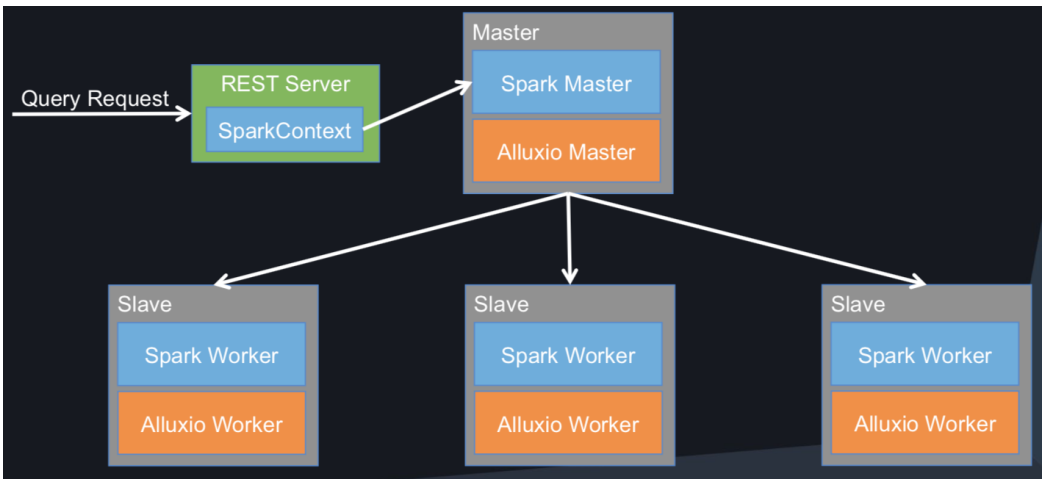
首先明确的是需要一个分布式的服务，主要包括接口服务、计算框架和文件系统三部分。其中接口服务用于接收查询请求，分析请求并生成实际的查询逻辑；计算框架用于分布式的执行查询逻辑；文件系统存储实际的索引数据，用于响应具体的查询。

这里简单谈一下架构选型的方法论，主要有四点：简单、成熟、可控、可调。

1.**简单**。不管是架构设计，还是逻辑复杂度和运维成本，都希望尽可能简单。这样的系统可以快速落地，也比较容易掌控。 2.**成熟**。评估一个系统是否成熟有很多方面，比如社区是否活跃，项目是否有明确的发展规划并能持续落地推进？再比如业界有没有足够多的成功案例，实际应用效果如何？一个成熟的系统在落地时的问题相对较少，出现问题也能参考其它案例比较容易的解决，从而很大程度上降低了整体系统的风险。 3.**可控**。如果一个系统持续保持黑盒的状态，那只能是被动的使用，出了问题也很难解决。反之现在有很多的开源项目，可以拿到完整的代码，这样就可以有更强的掌控力，不管是问题的定位解决，还是修改、定制、优化等，都更容易实现。 4.**可调**。一个设计良好的系统，在架构上一定是分层和模块化的，且有合理的抽象。在这样的架构下，针对其中一些逻辑做进一步定制或替换时就比较方便，不需要对代码做大范围的改动，降低了改造成本和出错概率。

基于上述的选型思路，服务的三个核心架构分别选择了Spring，Spark和Alluxio。其中Spring的应用非常广泛，在实际案例和文档上都非常丰富，很容易落地实现；Spark本身是一个非常优秀的分布式计算框架，目前团队对Spark有很强的掌控力，调优经验也很丰富，这样只需要专注在计算逻辑的开发即可；Alluxio相对HDFS或HBase来说更加轻量，同时支持包括内存在内的多层异构存储，这些特性可能会在后续优化中得到利用。

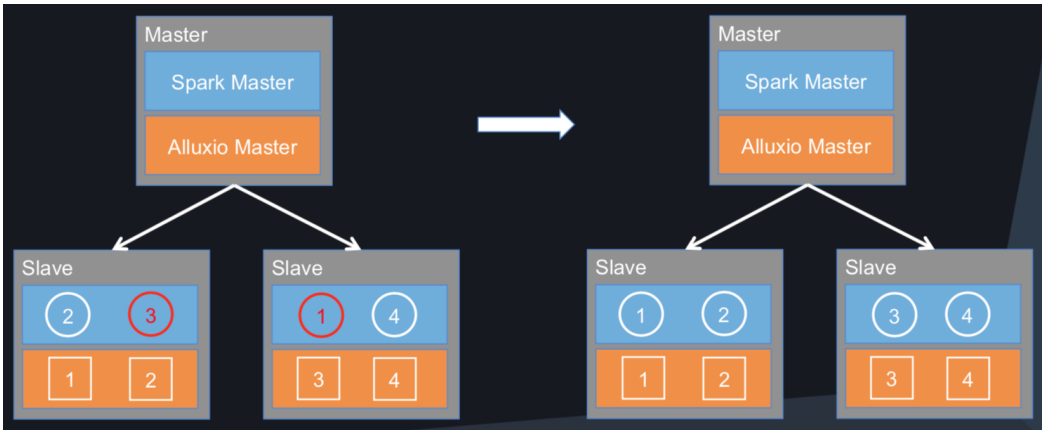
在具体的部署方式上，Spring Server单独启动，Spark和Alluxio都采用Standalone模式，且两个服务的slave节点在物理机上共同部署。Spring进程中通过SparkContext维持一个Spark长作业，这样接到查询请求后可以快速提交逻辑，避免了申请节点资源和启动Executor的时间开销。



架构概览

上述架构通过对数据的合理分区和资源的并发利用，可以实现一个查询请求在几分钟内完成。相对原来的几个小时有了很大改观，但还是不能满足交互式分析的需求，因此还需要做进一步的优化。

1. **本地化调度**。存储和计算分离的架构中这是常见的一种优化手段。以下图为例，某个节点上task读取的数据在另外节点上，这样就产生了跨机器的访问，在并发度很大时对网络IO带来了很大压力。如果通过本地化调度，把计算调度到数据的同一节点上执行，就可以避免这个问题。实现本地化调度的前提是有包含数据位置信息的元数据，以及计算框架的支持，这两点在Alluxio和Spark中都很容易做到。

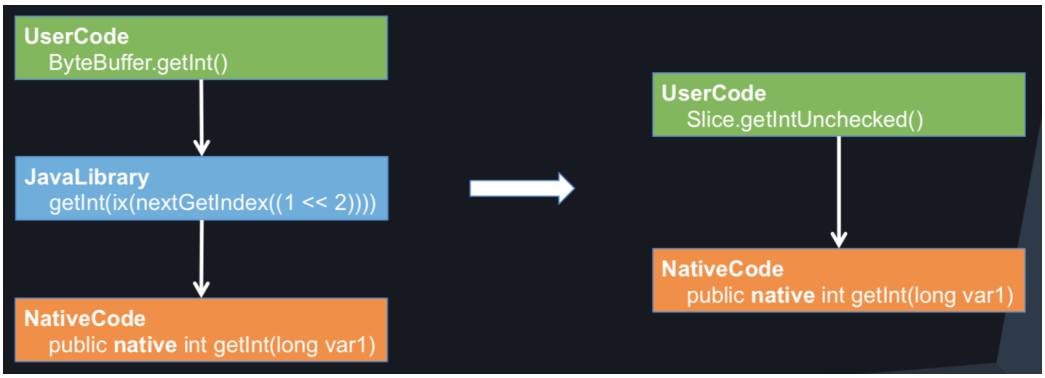


优化1

1. **内存映射**。常规实现中，数据需要从磁盘拷贝到JVM的内存中，这会带来两个问题。一是拷贝的时间很长，几百MB的数据对CPU时间的占用非常可观；二是JVM的内存压力很大，带来GC等一系列的问题。通过mmap等内存映射的方式，数据可以直接读取，不需要再进JVM，这样就很好的解决了上述的两个问题。

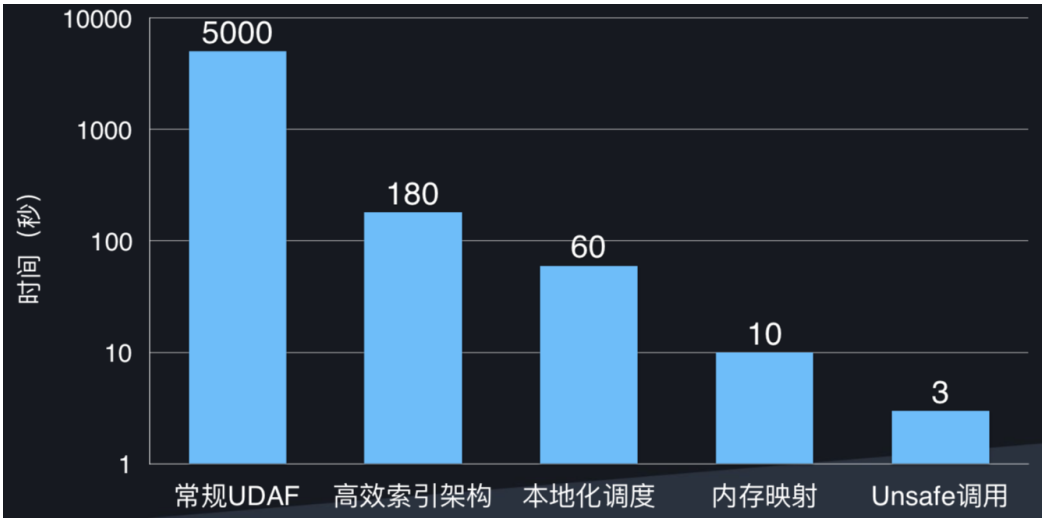
优化2

1. **Unsafe调用**。由于大部分的数据通过ByteBuffer访问，这里带来的额外开销对最终性能也有很大影响。Java lib中的ByteBuffer访问接口是非常安全的，但安全也意味着低效，一次访问会有很多次的边界检查，而且多层函数的调用也有很多额外开销。如果访问逻辑相对简单，对数据边界控制很有信心的情况下，可以直接调用native方法，绕过上述的一系列额外检查和函数调用。这种用法在很多系统中也被广泛采用，比如Presto和Spark都有类似的优化方法。



优化3

下图是对上述优化过程的对比展示。请注意纵轴是对数轴，也就是说图中每格代表了一个数据级的优化。从图中可以看到，常规的UDAF方案一次查询需要花几千秒的时间，经过索引结构的设计、本地化调度、内存映射和Unsafe调用的优化过程之后，一次查询只需要几秒的时间，优化了3~4个数据级，完全达到了交互式分析的需求。



优化对比

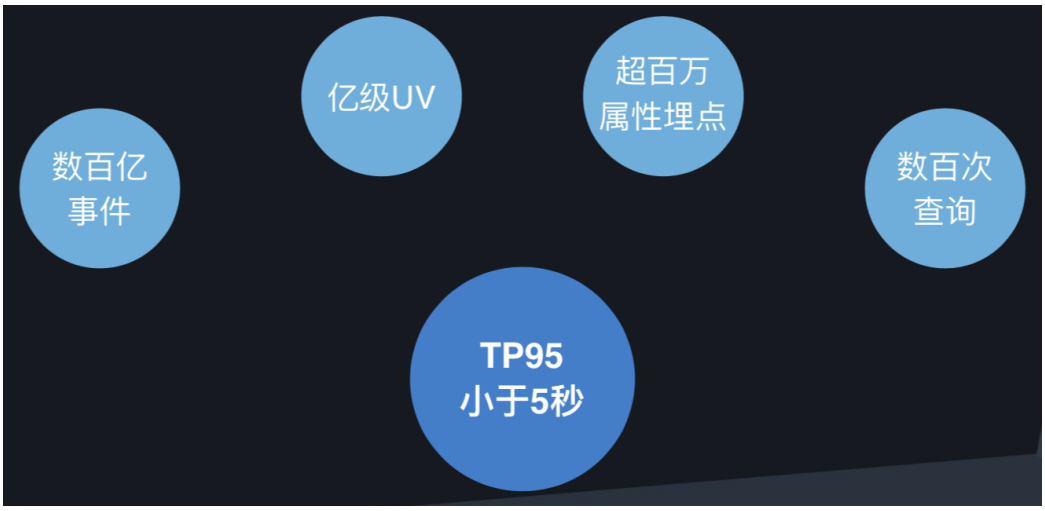
这里想多谈几句对这个优化结果的看法。主流的大数据生态系统都是基于JVM系语言开发的，包括Hadoop生态的Java，Spark的Scala等等。由于JVM执行机制带来的不可避免的性能损失，现在也有一些基于C++或其它语言开发的系统，有人宣称在性能上有几倍甚至几十倍的提升。这种尝试当然很好，但从上面的优化过程来看，整个系统主要是通过更高效的数据结构和更合理的系统架构达到了3个数量级的性能提升，语言特性只是在最后一步优化中有一定效果，在整体占比中并不多。

有一句鸡汤说“以大多数人的努力程度而言，根本没有到拼天赋的地步”，套用在这里就是“以大多数系统的架构设计而言，根本没有到拼语言性能的地步”。语言本身不是门槛，代码大家都会写，但整个系统的

架构是否合理，数据结构是否足够高效，这些设计依赖的是对问题本质的理解和工程上的权衡，这才是更考量设计能力和经验的地方。

总结

上述方案目前在美团点评内部已经实际落地，稳定运行超过半年以上。每天的数据有几百亿条，活跃用户达到了上亿的量级，埋点属性超过了百万，日均查询量几百次，单次查询的TP95时间小于5秒，完全能够满足交互式分析的预期。

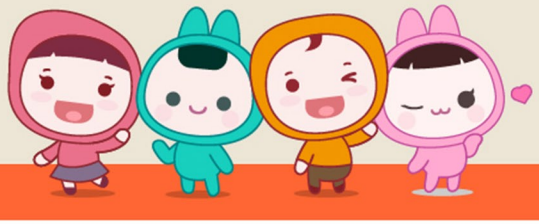


效果总结

整个方案从业务需求的实际理解和深入分析出发，抽象出了维度筛选、序列匹配和去重计数三个核心问题，针对每个问题都给出了合理高效的解决方案，其中结合实际数据特点对数据结构的优化是方案的最大亮点。在方案的实际工程落地和优化过程中，秉持“简单、成熟、可控、可调”的选型原则，快速落地实现了高效架构，通过一系列的优化手段和技巧，最终达成了3~4个数量级的性能提升。

作者简介

- 业锐，2015年加入美团，现任美团数据平台查询引擎团队负责人。主要负责数据生产和查询引擎的改进优化和落地应用，专注于分布式计算，OLAP分析，Adhoc查询等领域，对分布式存储系统亦有丰富经验。



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