

阿里中间件技术揭秘

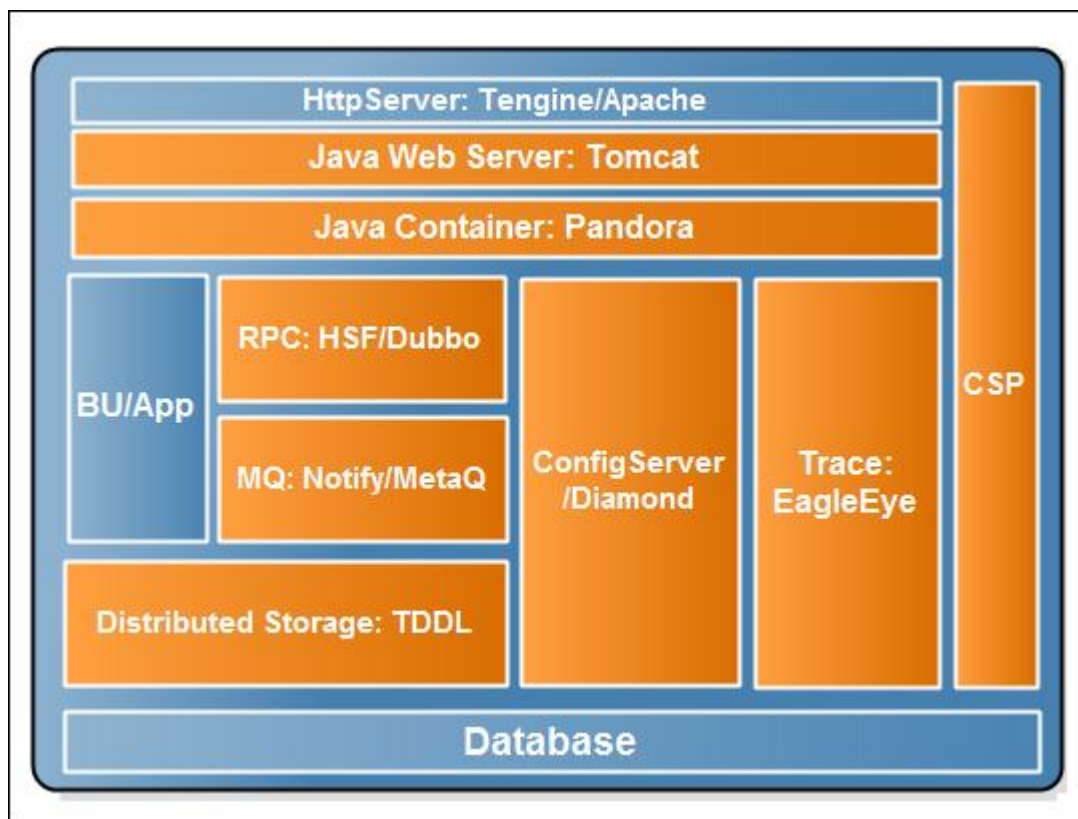
阿里中间件
2014 年 4 月

<http://jm.taobao.org>

中间件总体介绍¹

阿里巴巴中间件与稳定性平台团队，是一个给业务应用团队以提供低成本，高可用，可扩展的弹性互联网系统解决方案为己任的技术团队，前身是成立于 7 年之前的淘宝平台架构部，而后随着业务领域，尤其是针对性能和稳定性技术领域的成功探索与突破，目前已经发展为一个涵盖消息通信，数据处理，性能优化和稳定性等各类技术的互联网架构服务平台。他成功地支持了包括淘宝，天猫，阿里云，小微金融等众多兄弟 bu 的业务与技术需求。在 2013 年的双 11 狂欢节中，中间件与稳定性平台团队也再一次站在了前排，与其他团队一起，为了“让天下没有难做的生意”贡献了自己的一份力量。

中间件与稳定性平台团队，是国内为数不多的极具技术挑战性的团队之一，依托于全球规模最大的阿里巴巴电子商务平台所带来的巨大流量和海量数据，以及对于电子商务平台固有的稳定性要求，使得团队有机会去面对一个又一个技术难题，创造一个又一个技术奇迹。从整体来看，中间件与稳定性平台的技术体系可以分为软负载配置、分布式服务框架、消息中间件、数据访问层、应用服务器和稳定性平台等，如下图所示。



中间件整体技术体系

如果我们将整个网站看成是一个大工厂，每个业务逻辑单元(Business Unit)看成是工厂内完成单个工序的车间，那么中间件就是能够让所有车间发挥出最大生产效率的各类支持性部门。

¹本文发表在《程序员》2014 年 1 月刊:11.11 背后的技术 <http://www.csdn.net/article/2013-12-23/2817882>

软负载系统（**Software Load Balancing**），通过软件系统解决请求的均衡负载。相对于 F5 或者 LVS 这些负载设备，软负载系统有以下特点：无中心化，成本更低，效率更高，功能更强。在解决长连接的负载均衡场景上，软负载系统可以做到长连接中的每个请求包级别的负载均衡，最大程度的优化资源的使用。

Java 的运行时容器(**Java Container** 和 **Web Server**)，主要为用户提供了软件库版本隔离和依赖升级推送的功能，就像是这家工厂的厂房，每个车间都要配备自己最趁手的工具，不同车间的工具如果相互共享使用，很容易出现螺丝找不到螺母的时候，从而造成生产停滞，因此，每个车间都应该有他们自己所需要的专用工具，并且要保证工具永远维持在最趁手的状态，才能让我们的工厂发挥出最理想的效能。

远程方法调用(**RPC**)，传统意义上也被称为 **SOA**，主要为用户提供了远程调用和服务治理的功能，他们能够让应用方将原来的整套业务逻辑拆分到不同的机器中运行。就像是车间之间的传送带，能够将各个车间的生产结合到一起，促进了分工合作，从而提升了生产效率。

消息系统(**MQ**)，主要为用户提供了发送通知的功能，让一些非核心流程可以并行执行。他们就像一家工厂的电子工单，最终产品一般是由多条生产线一起协作生产出来的：产品的各个部件的生产是完全可以并行的。但最终用户则需要的是包装好的全部产品。这时候就需要工单系统，让整个企业内可以并行生产的部分能够协调一致的进行产品的生产，并最终能够以合适的数量进行成品组装。

分布式存储(**Distributed Storage**)，则主要为用户提供了可无限扩展的数据存储服务。这就像这家工厂的仓库，能否按照实际的需要，做到仓库的自动化运维和管理，按需扩展和收缩，是仓库运维管理中最为重要的挑战。

分布式调用跟踪系统（**Distributed Tracing**）通过收集和分析在不同中间件上网络调用的日志埋点，可以得到同一次请求上的各个系统的调用链关系，有助于梳理应用的请求入口与服务的调用来源、依赖关系，同时，也对分析系统调用瓶颈、估算链路容量、快速定位异常有很大帮助。

持续稳定性平台（**Continue Stable Platform**）是一个平台化的产品，涉及依赖治理、容量规划、实时监控和降级管理等多个领域，致力为阿里巴巴的各个系统提供稳定的数据和工具支持。

在本文后面的章节中，我们将依次概要介绍这些中间件和稳定产品，并以双 11 大促作为契机，为大家介绍那些我们在双 11 中使用的技术手段以及优化案例，与大家一起重新经历那些激动又紧张的日日夜夜。

1、软负载——分布式系统的引路人

综述

软负载是分布式系统中极为普遍的技术之一。在分布式环境中，为了保证高可用性，通常同一个应用或同一个服务的提供方都会部署多份，以达到对等服务。而软负载就像一个引路人，帮助服务的消费者在这些对等的服务中合理地选择一个来执行相关的业务逻辑。

1.1、ConfigServer

ConfigServer 主要提供非持久配置的发布和订阅。07/08 年间在淘宝内部开发使用的时候，由于 ZooKeeper 还没有开源，不然可能会基于 ZooKeeper 来进行改造。主要使用场景是为分布式服务框架提供软负载功能所必须的服务地址列表。

结合淘宝业务场景的发展，与 ZooKeeper 相比主要有以下一些特点：

- ConfigServer 基于无 Master 架构

ConfigServer 是无中心化的架构，不存在单点问题，通过特定的算法进行数据增量同步。

- ConfigServer 支持数据的自动聚合

配置数据的聚合功能是 ConfigServer 结合淘宝的使用场景开发的特殊功能，主要使用场景：集群服务地址发现。每台机器向 ConfigServer 注册自己的服务元信息，ConfigServer 会根据服务名和分组自动聚合所有元数据，提供给服务订阅方使用。

- ConfigServer 是推数据的模型

ConfigServer 的服务端与客户端是基于长连接的方式进行通信，在客户端订阅关系确定后，配置信息一旦发生变化，会主动推送配置数据到客户端，并回调客户端的业务监听器。

- ConfigServer 客户端本地容灾

ConfigServer 客户端在收到配置数据时，会记录到本地容灾文件中。在 ConfigServer 服务端不可用的时候，应用继续使用内存中数据；即使应用这时候重启，ConfigServer 的客户端使用本地容灾文件进行离线启动。

1.2、ConfigServer 双 11 准备与优化

面对双 11 巨大的网站请求,对 ConfigServer 的数据推送环节考验尤其巨大,如何稳定、高效且正确的完成数据推送任务,成为双 11 前夕困扰 ConfigServer 团队最大的问题。在平时的正常运行中,ConfigServer 管理的配置数据和订阅者数量已经达到一定量级,服务端的一次重启可能会引起大量的数据推送。举例来说:交易的某个服务有 2000 台,每个服务发布者信息 200B,订阅者 5000 个,那么这个交易服务可用地址列表有变动,会产生 2GB(2000*200B*5000)的数据推送。交易的应用有很多个服务,如果应用发布或者重启,会瞬间产生巨大的流量将 ConfigServer 服务端的网卡压满。当网卡压满后:保持长连接的心跳检测包会超时,导致与客户端的连接会持续在一个不稳定的状态,集群间数据同步也会受影响。

针对这个问题, ConfigServer 在推送数据上做了两个方面的优化:

- 流量控制

加入出口流量统计和监控,通过严格限制出口流量预留出足够的带宽给维持心跳和集群数据同步。这样即使有数据堆积,只会暂时的影响到数据推送,客户端的连接不会频繁的断开,集群之间的同步也不会受到影响。

- 减少推送量

出于两方面考虑:压缩数据对代码的侵入不大;ConfigServer 服务端 CPU 相对空闲,最终我们采用压缩数据的方式来减小数据包。我们经过对比主流的压缩算法(huffman, deflate, BZip2, LZMA)对真实推送数据的压缩测试,最终选择 deflate 算法。该功能上线后的检测结果显示 1500M 的推送数据最终推送大小为 170M。效果很明显。

1.3、Diamond

Diamond 主要提供持久配置的发布和订阅服务,最大特点是结构简单,稳定可靠。

Diamond 的主要使用场景是用来进行动态数据库切换与扩容,进行一些业务系统运行时开关配置的推送。Diamond 产品专注于高可用性,基于此在架构、容灾机制、数据获取模型上有一些与同类产品的不同之处。

Diamond 结构非常简单，也属于是无单点的架构模型，如图 1-1 所示。

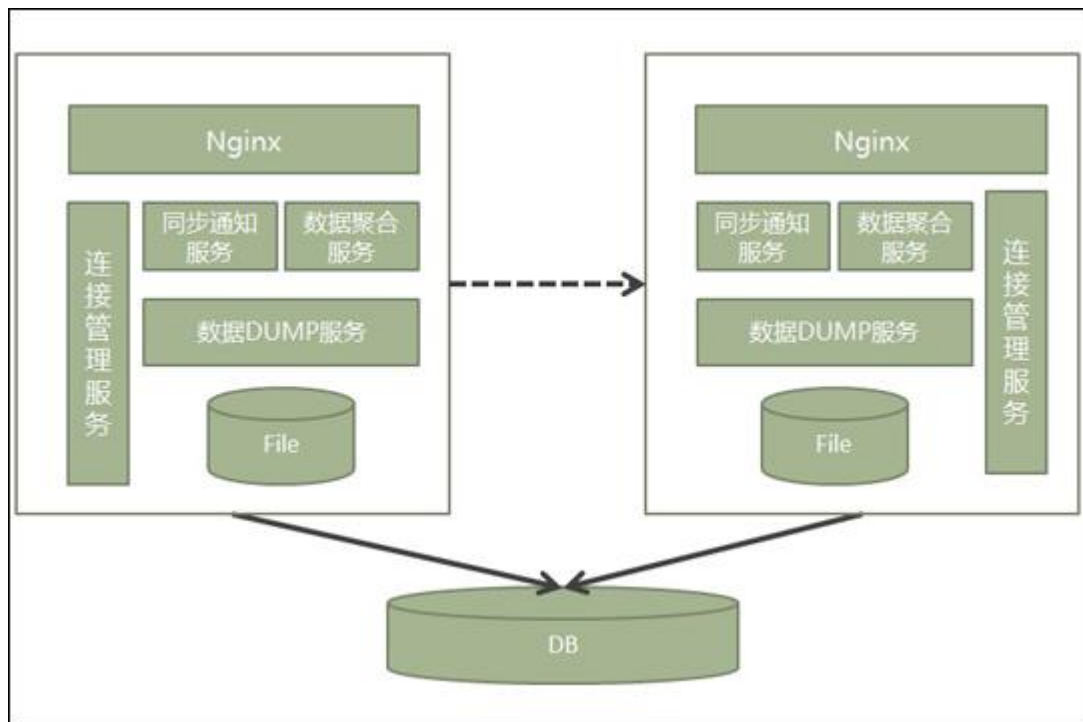


图 1-1-Diamond 架构模型

发布或者更新配置数据时，步骤如下：

- 写入 MySQL 数据库
- 写本地磁盘
- 通知集群其他机器去数据库 dump 更新的数据

订阅方获取配置数据时，直接读取服务端本地磁盘文件，尽量减少对数据库压力。这种架构用短暂的延时换取最大的性能和一致性，一些配置不能接受延时的情况下，通过 API 可以获取数据库中的最新配置。

容灾机制

Diamond 作为一个分布式环境下的持久配置系统，有一套完备的容灾机制，数据存储在于：数据库、服务端磁盘、客户端缓存目录以及可以手工干预的容灾目录。客户端通过 API 获取配置数据按照固定的顺序去不同的数据源获取数据：容灾目录->服务端磁盘->客户端缓存。因此，面对如下情况，Diamond 均能很好的应对：

- 数据库主库不可用，可以切换到备库，Diamond 继续提供服务
- 数据库主备库全部不可用，Diamond 通过本地缓存可以继续提供读服务
- 数据库主备库全部不可用，Diamond 服务端全部不可用，Diamond 客户端使用缓存目录继续运行，支持离线启动
- 数据库主备库全部不可用，Diamond 服务端全部不可用，Diamond 客户端缓存数据被删，可以通过拷贝备份的缓存目录到容灾目录下继续使用

综上所述，只有在同时碰到如下四个条件的情况下，客户端应用才无法启动：数据库主备库全部不可用、Diamond 服务端全部不可用、Diamond 客户端缓存被清空、客户端没有备份的缓存文件。

1.4、Diamond 双 11 准备与优化

长轮询改造

客户端采用推拉结合的策略在长连接和短连接之间取得一个平衡，让服务端不用太关注连接的管理，又可以获得长连接的及时性。

- 客户端发起一个对比请求到服务端，请求中包含客户端订阅的数据的指纹
- 服务端检查客户端的指纹是否与最新数据匹配
 - 如果匹配，服务端持有连接
 - 如果 30 秒内没有相关数据变化，服务端持有连接 30 秒后释放
 - 如果 30 秒内有相关数据变化，服务端立即返回变化数据的 ID
- 如果不匹配，立即返回变化数据的 ID
- 客户端根据变化数据的 ID 去服务端获取最新的内容

Diamond 通过这种多重容灾机制以及推拉结合的方式，让客户端逻辑尽量简单，而且高效稳定，使其成为名副其实的“钻石”。

小结

软负载系统引导着分布式请求的来龙去脉，管理着分布式离散数据的聚合与分发，成为了合理且最大化使用分布式系统资源的保证。在 2013 年的双 11 购物狂欢节中，面对巨大的消费者请求，ConfigServer 集群共 10 台机器，支撑近 60,000 长连接，发布配置总量 700,000 条，订阅配置总量 3,000,000 条，数据推送峰值 600MB/S。Diamond 集群共 48 台，90,000 条非聚合配置，380,000 条聚合配置，TPS 高峰期间达到 18000，数据变更 15000 次，客户端感知最大延时小于 2 秒。双十一当天部分数据库做了切换，用户完全没有感知。

2、分布式服务框架——分布式服务的组织者

综述

06/07 年以后，随着淘宝用户数量和网站流量的增长，应用系统的数量和复杂程度也急剧增加。诸多前台系统都需要使用一些公共的业务逻辑，这些业务逻辑通常具有共性的

东西，比如，获取用户信息或查询宝贝详情等。如果将这些业务逻辑在各个系统内部都实现一遍，则大大增加了开发成本和后期维护成本。于是，像服务框架这类的中间件产品就应运而生。服务框架帮助各个系统将那些相似的业务逻辑抽离出来，单独部署，而前台系统在需要调用这些业务逻辑时，只需要通过服务框架远程调用即可，大大节约了前端系统的开发成本，也提高了系统的可维护性和可扩展性。

2.1、HSF 简介

HSF 是淘宝的分布式服务框架。服务框架从分布式应用层面以及统一的发布/调用方式层面为业务系统提供支持，从而可以让他们很容易地开发分布式应用并提供和使用公用功能模块，而不用考虑分布式领域中的各种细节技术，例如远程通讯、性能损耗、调用的透明化、同步/异步调用方式的实现等等问题。

服务框架的实现有三种角色：服务提供者、服务消费者和注册中心。服务提供者在服务可用的前提下，将地址注册到注册中心。服务消费者启动时，会订阅注册中心的相关服务，获取服务地址，通过一定的负载均衡策略调用服务。由于注册中心这个软负载集群的存在，服务提供者和服务消费者可以任意扩容和下线，注册中心可以实时将提供者地址的变更推送给消费者。

服务治理

服务治理是服务框架的核心功能。所谓服务治理，是指服务的提供方和消费方达成一致的约定，保证服务的高质量。服务治理功能，可以解决将某些特定流量引入某一批机器，以及限制某些非法消费者的恶意访问，和在提供者处理量达到一定程度时，拒绝接受新的请求等功能。

2.2、HSF 双 11 准备与优化

在双 11 中，HSF 主要通过精简日志输出、流量限制、解决应用依赖冲突等措施，保证了服务的稳定可靠。

- 精简日志输出

消费者在调用过程中，容易因为网络问题或服务提供方等原因引起调用失败。如果没有足够的日志，有时候排查问题会很困难。因此，服务框架在生产环境使用时，往往将日志级别设置比较低或打印较多日志，记录下足够多的信息。这在平时没有问题，而且在遇到问题时也有足够的信息来排查问题。但是日志打印本身耗费性能，在双 11 这种高峰调用期间，尽量要减少日志的输出。为了达到灵活控制日志输出的目标，服务框架优化了日志打印，精简了日志输出。

- 流量限制

虽然很多应用设置了流量限制等规则，但平时的流量远远低于阈值，只有在双 11 这种流量高峰，才会起到效果。在双 11 之前，我们检查了线上所有的限流规则，发现有不少配置错误或者配置不合理的情况，其中有些是由于 HSF 对于一些默认参数设置不合理造成的。通过性能测试，将不合理的规则和参数进行改正。

- 解决应用依赖冲突

由于淘宝业务发展迅速，前端应用需要依赖越来越多的其他系统，这很容易造成应用依赖的冲突。服务框架引入了 Pandora 容器，对应用进行了依赖的隔离，防止应用和服务框架的依赖相互冲突。

小结

HSF 已经经受了淘宝各种复杂、高并发的调用场景。今年来，HSF 在易用性、服务治理和性能上有了很大的改进，是很稳定的分布式服务框架。作为淘宝中间件团队最早诞生的中间件框架之一，HSF 将在未来继续发挥其巨大的作用。

3、消息中间件——分布式消息的广播员

综述

消息中间件是一种由消息传送机制或消息队列模式组成的最典型的中间件技术。通过消息中间件，应用程序或组件之间可以进行可靠的异步通讯来降低系统之间的耦合度，从而提高整个系统的可扩展性和可用性。

3.1、Notify

Notify 是淘宝自主研发的一套消息服务引擎，是支撑双 11 最为核心的系统之一，在淘宝和支付宝的核心交易场景中都有大量使用。消息系统的核心作用就是三点：解耦，异步和并行。下面让我以一个实际的例子来说明一下解耦异步和并行分别所代表的具体意义吧：

假设我们有这么一个应用场景，为了完成一个用户注册淘宝的操作，可能需要将用户信息写入到用户库中，然后通知给红包中心给用户发新手红包，然后还需要通知支付宝给用户准备对应的支付宝账号，进行合法性验证，告知 sns 系统给用户导入新的用户等 10 步操作。

那么针对这个场景，一个最简单的设计方法就是串行的执行整个流程，如图 3-1 所示：



图 3-1-用户注册流程

这种方式的最大问题是，随着后端流程越来越多，每步流程都需要额外的耗费很多时间，从而会导致用户更长的等待延迟。自然的，我们可以采用并行的方式来完成业务，能够极大的减少延迟，如图 3-2 所示。

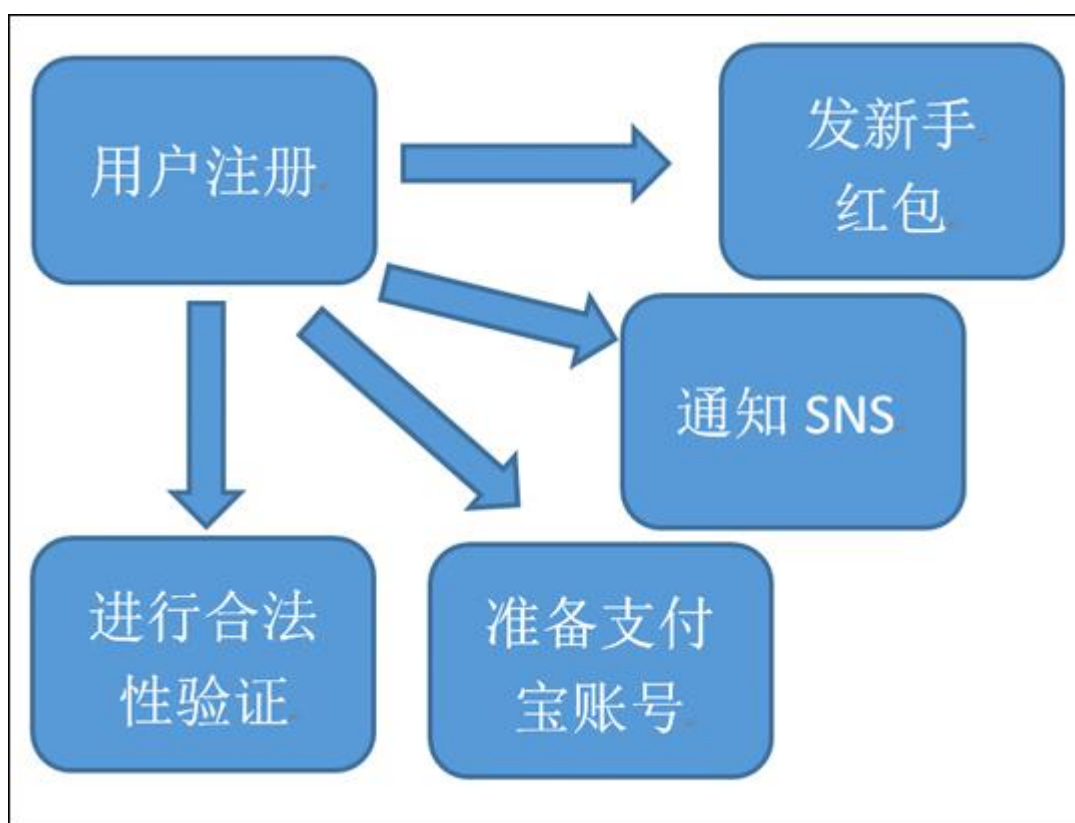


图 3-2-用户注册流程-并行方式

但并行以后又会有一个新的问题出现了，在用户注册这一步，系统并行的发起了 4 个请求，那么这四个请求中，如果通知 SNS 这一步需要的时间很长，比如需要 10 秒钟的话，那么就算是发新手包，准备支付宝账号，进行合法性验证这几个步骤的速度再快，用户也仍然需要等待 10 秒以后才能完成用户注册过程。因为只有当所有的后续操作全部完成的时候，用户的注册过程才算真正的“完成”了。用户的信息状态才是完整的。而如果这时候发生了更严重的事故，比如发新手红包的所有服务器因为业务逻辑 bug 导致 down 机，那么因为用户的注册过程还没有完全完成，业务流程也就是失败的了。这样明显是不符合实际的需要的，随着下游步骤的逐渐增多，那么用户等待的时间就会越来越长，并且更加严重的是，随着下游系统越来越多，整个系统出错的概率也就越来越大。

通过业务分析我们能够得知，用户的实际的核心流程其实只有一个，就是用户注册。而后续的准备支付宝，通知 sns 等操作虽然必须要完成，但却是不需要让用户等待的。这种模式有个专业的名词，就叫最终一致。为了达到最终一致，我们引入了 MQ 系统。业务流程如下：

主流程如图 3-3 所示：

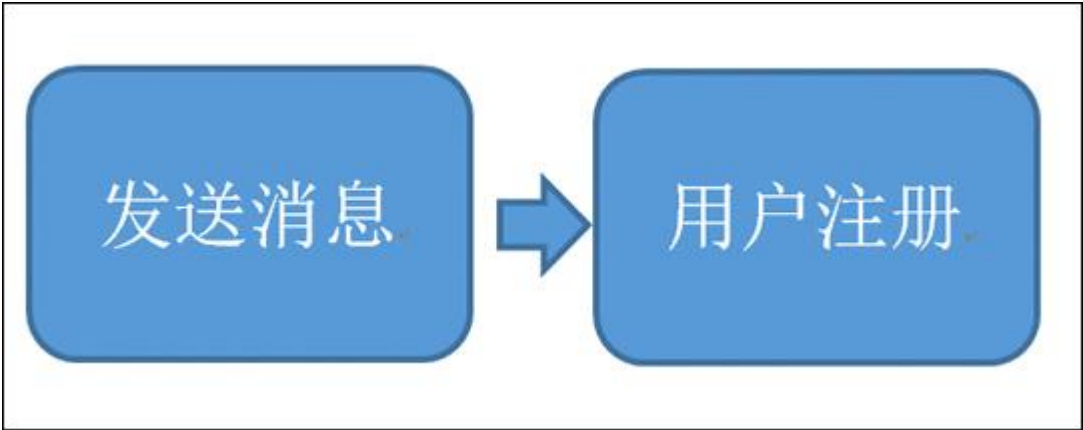


图 3-3-用户注册流程-引入 MQ 系统-主流程

异步流程如图 3-4 所示：

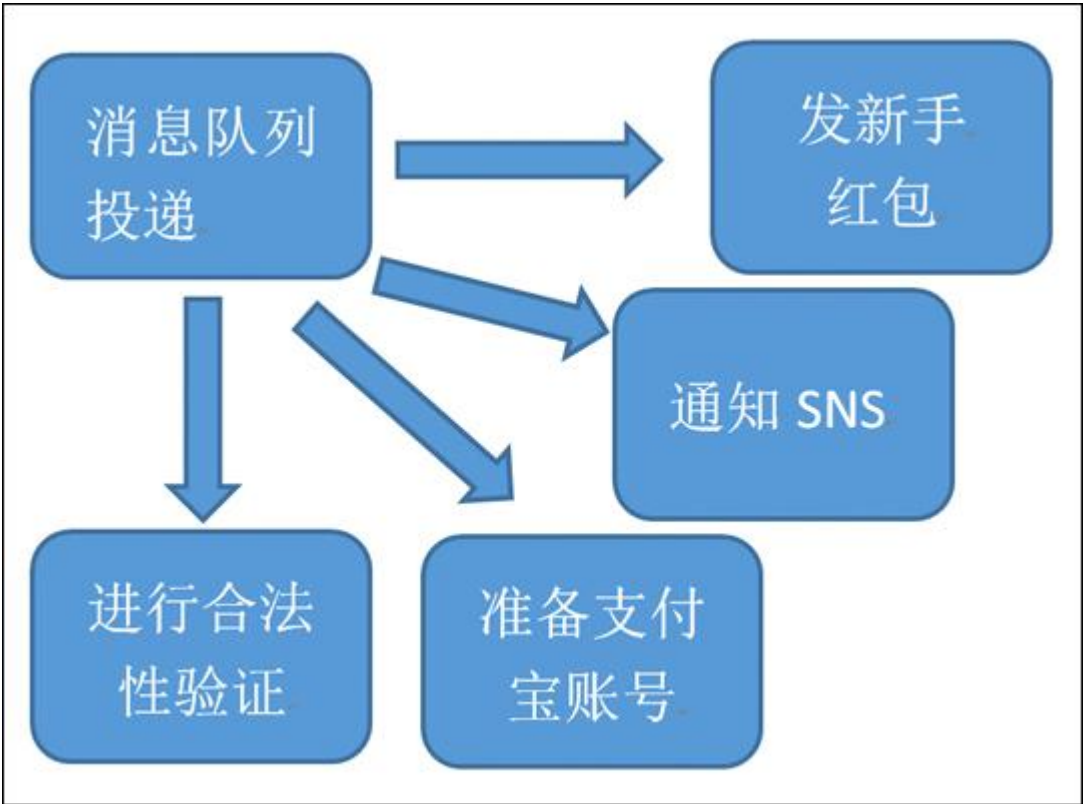


图 3-4-用户注册流程-引入 MQ 系统-异步流程

核心原理

Notify 在设计思路上的传统的 MQ 有一定的不同，他的核心设计理念是

1. 为了消息堆积而设计系统
2. 无单点，可自由扩展的设计

下面就请随我一起，进入到我们的消息系统内部来看看他设计的核心原理

- 为了消息堆积而设计系统

在市面上的大部分 MQ 产品，大部分的核心场景就是点对点的消息传输通道，然后非常激进的使用内存来提升整体的系统性能，这样做虽然标称的 tps 都能达到很高，但这种设计的思路是很难符合大规模分布式场景的实际需要的。

在实际的分布式场景中，这样的系统会存在着较大的应用场景瓶颈，在后端有大量消费者的前提下，消费者出现问题是个非常常见的情况，而消息系统则必须能够在后端消费不稳定的情况下，仍然能够保证用户写入的正常并且 TPS 不降，是个非常考验消息系统能力的实际场景。

也因为如此，在 Notify 的整体设计中，我们最优先考虑的就是消息堆积问题，在目前的设计中我们使用了持久化磁盘的方式，在每次用户发消息到 Notify 的时候都将消息先落盘，然后再异步的进行消息投递，而没有采用激进的使用内存的方案来加快投递速度。

这种方式，虽然系统性能在峰值时比目前市面的 MQ 效率要差一些，但是作为整个业务逻辑的核心单元，稳定，安全可靠是系统的核心诉求。

- 无单点，可自由扩展的设计

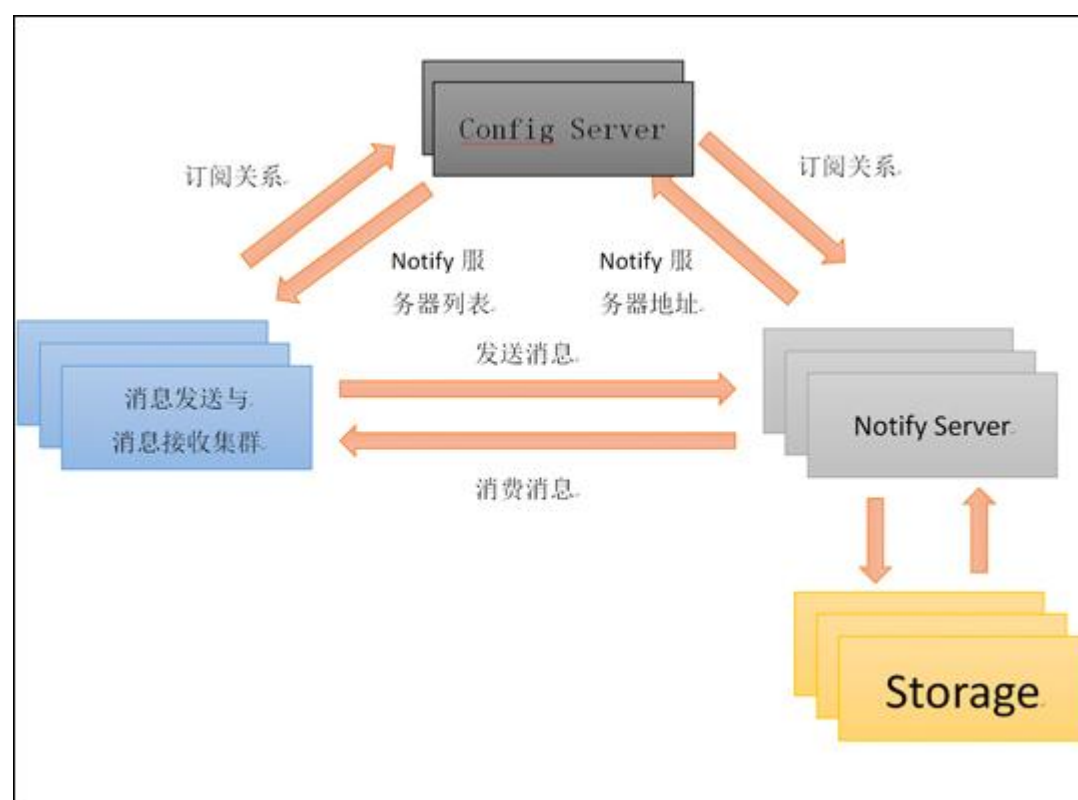


图 3-5-Notify 系统组成结构

图 3-5 展示了组成 Notify 整个生态体系的有五个核心的部分。

- 发送消息的集群

这主要是业务方的机器，这些 APP 的机器上是没有任何状态信息的，可以随着用户请求量的增加而随时增加或减少业务发送方的机器数量，从而扩大或缩小集群能力。

- 配置服务器集群(Config server)

这个集群的主要目的是动态的感知应用集群，消息集群机器上线与下线的过程，并及时广播给其他集群。如当业务接受消息的机器下线时，config server 会感知到机器下线，从而将该机器从目标用户组内踢出，并通知给 notify server，notify server 在获取通知后，就可以将已经下线的机器从自己的投递目标列表中删除，这样就可以实现机器的自动上下线扩容了。

- 消息服务器(Notify Server)

消息服务器，也就是真正承载消息发送与消息接收的服务器，也是一个集群，应用发送消息时可以随机选择一台机器进行消息发送，任意一台 server 挂掉，系统都可以正常运行。当需要增加处理能力时，只需要简单地增加 notify Server 就可以了

- 存储(Storage)

Notify 的存储集群有多种不同的实现方式，以满足不同应用的实际存储需求。针对消息安全性要求高的应用，我们会选择使用多份落盘的方式存储消息数据，而对于要求吞吐量而不要求消息安全的场景，我们则可以使用内存存储模型的存储。自然的，所有存储也被设计成了随机无状态写入存储模型以保障可以自由扩展。

- 消息接收集群

业务方用于处理消息的服务器组，上下线机器时候也能够动态的由 config server 感知机器上下线的时机，从而可以实现机器自动扩展。

3.2、Notify 双 11 准备与优化

在双 11 的整个准备过程中，Notify 都承载了非常巨大的压力，因为我们的核心假定就是后端系统一定会挂，而我们需要能够承载整个交易高峰内的所有消息都会堆积在数据库内的实际场景。

在多次压测中，我们的系统表现还是非常稳定的，以 60w/s 的写入量堆积 4.5 亿消息的时候，整个系统表现非常淡定可靠。在真正的大促到来时，我们的后端系统响应效率好于预期，所以我们很轻松的就满足了用户所有消息投递请求，比较好的满足了用户的实际需要。

3.3、MetaQ

METAQ 是一款完全的队列模型消息中间件，服务器使用 Java 语言编写，可在多种软硬件平台上部署。客户端支持 Java、C++编程语言，已于 2012 年 3 月对外开源，开源地址是：<http://metaq.taobao.org/>。MetaQ 大约经历了下面 3 个阶段

- 在 2011 年 1 月份发布了 MetaQ 1.0 版本，从 Apache Kafka 衍生而来，在内部主要用于日志传输。
- 在 2012 年 9 月份发布了 MetaQ 2.0 版本，解决了分区数受限问题，在数据库 binlog 同步方面得到了广泛的应用。
- 在 2013 年 7 月份发布了 MetaQ 3.0 版本，MetaQ 开始广泛应用于订单处理，cache 同步、流计算、IM 实时消息、binlog 同步等领域。MetaQ3.0 版本已经开源，[参见这里](#)

综上，MetaQ 借鉴了 Kafka 的思想，并结合互联网应用场景对性能的要求，对数据的存储结构进行了全新设计。在功能层面，增加了更适合大型互联网特色的功能点。

MetaQ 简介

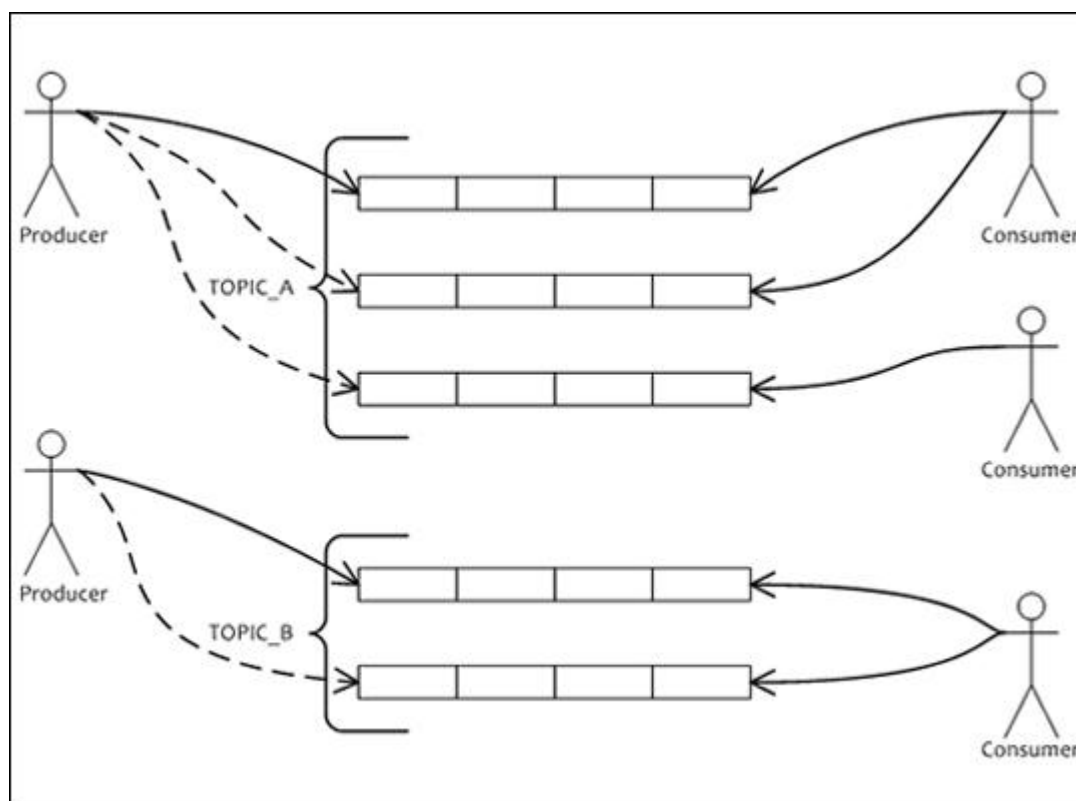


图 3-6-MetaQ 整体结构

如图 3-6 所示，MetaQ 对外提供的是一个队列服务，内部实现也是完全的队列模型，这里的队列是持久化的磁盘队列，具有非常高的可靠性，并且充分利用了操作系统 cache 来提高性能。

- 是一个队列模型的消息中间件，具有高性能、高可靠、高实时、分布式特点。
- Producer、Consumer、队列都可以分布式。

- **Producer** 向一些队列轮流发送消息，队列集合称为 **Topic**，**Consumer** 如果做广播消费，则一个 **consumer** 实例消费这个 **Topic** 对应的所有队列，如果做集群消费，则多个 **Consumer** 实例平均消费这个 **topic** 对应的队列集合。
- 能够保证严格的消息顺序
- 提供丰富的消息拉取模式
- 高效的订阅者水平扩展能力
- 实时的消息订阅机制
- 亿级消息堆积能力

MetaQ 存储结构

MetaQ 的存储结构是根据阿里大规模互联网应用需求，完全重新设计的一套存储结构，使用这套存储结构可以支持上万的队列模型，并且可以支持消息查询、分布式事务、定时队列等功能，如图 3-7 所示。

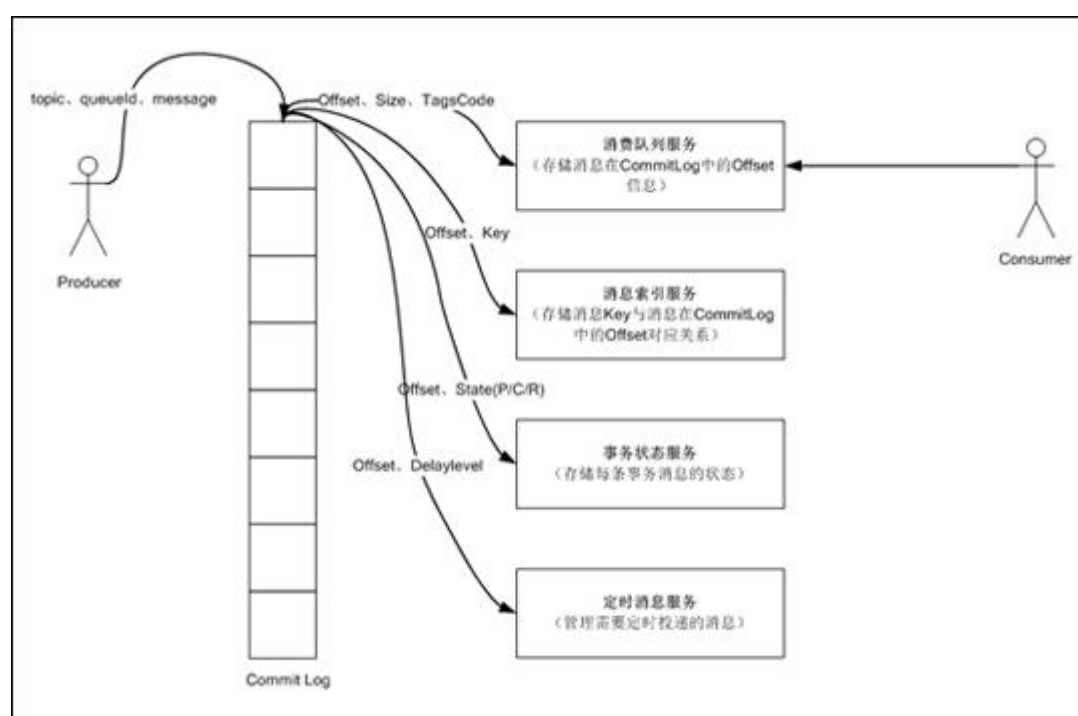


图 3-7-MetaQ 存储体系

MetaQ 单机上万队列

MetaQ 内部大部分功能都靠队列来驱动，那么必须支持足够多的队列，才能更好的满足业务需求，如图所示，MetaQ 可以在单机支持上万队列，这里的队列全部为持久化磁盘方式，从而对 IO 性能提出了挑战。MetaQ 是这样解决的

- **Message** 全部写入到一个独立的队列，完全的顺序写
- **Message** 在文件的位置信息写入到另外的文件，串行方式写。

通过以上方式，既做到数据可靠，又可以支持更多的队列，如图 3-8 所示。

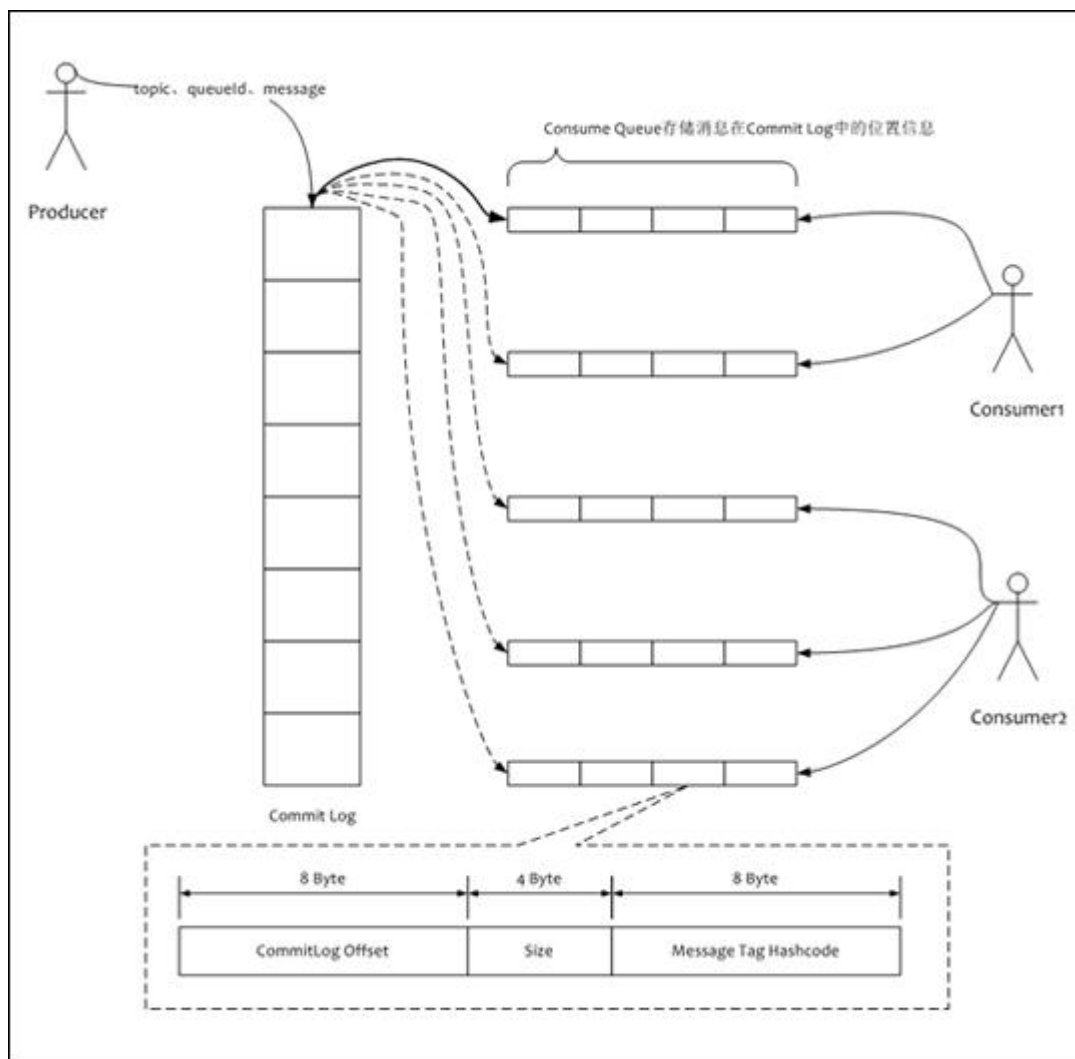


图 3-8-MetaQ 单机上万队列

MetaQ 与 Notify 区别

- Notify 侧重于交易消息，分布式事务消息方面。
- MetaQ 侧重于顺序消息场景，例如 binlog 同步。以及主动拉消息场景，例如流计算等。

3.4、MetaQ 双 11 准备与优化

- Notify 交易消息转 MetaQ 方案改进

MetaQ 交易集群主要是 Notify 交易消息的一个镜像，旧有的方案是通过 Notify-Client 订阅 Notify 交易消息，然后再转投到 MetaQ 集群，这个方案的缺点：1. 通过消息订阅的方式给 Notify 集群带来比较大的压力 2. 一旦 MetaQ 集群处理不及时会给 Notify 造成消息的堆积，从而带来连锁不良效应。

新的方案则是从 Notify DB 直接拉取 binlog 到 MetaQ，它带来的优势：1. 解放 NotifyServer 集群的压力；2. 通过 binlog 批量处理可以提升系统的吞吐量。

- 交易集群低延迟优化
天猫直播间，旨在通过实时获取活动当天的交易数据，通过实时流计算的方式，及时、准确的展示各业务数据。它的特点就是数据全而准确、实时性要求较高。而在全链路压测过程中发现从 **Notify Mysql binlog** 获取数据时，出现较大的延迟，最严重的延迟高达 **4h+**，这显然是不合系统需求的。基于这些问题，我们在 **Notify Mysql** 端做了很多的优化：
 1. **Mysql** 数据库实例扩容，从而提高集群的整体吞吐量；
 2. 对 **binlog** 的存放位置进行优化，将数据存储以及 **binlog** 存储进行分离，从而发挥 **DB** 的最大写性能；
 3. 由于 **MySQL** 的 **binlog** 操作存在锁操作，优化了 **MySQL** 生成 **binlog** 的配置，保证了拉 **binlog** 无延时。
- 针对不同集群运行参数调优
根据业务的特点，对不同集群的运行参数调优，如批量拉取大小，刷盘方式，数据有效期等等；同时对 **io** 调度、虚拟内存等参数进行调优，以提供更为高效的堆积。
- 监控与实时告警
任何一个值得信赖的系统，最低限度是需要做到及时发现并处理异常，在第一时间排除故障发生的可能，从而提高产品的可用性。在双十一活动之前，我们实现了由 **MetaQ** 系统内部，根据集群状态，各消息的业务数据指标进行监控统计并主动告警，同时还能通过 **Diamond** 做到动态调整，从而提高监控的及时性以及灵活性。

回顾双十一活动当日，淘宝消息写入总量 **112** 亿，消息投递总量 **220** 亿，支付宝消息写入总量 **24** 亿，消息投递总量 **24** 亿。其中实时直播间集群消息写入峰值为 **13.1w**，消息投递峰值为 **27.8w**。

从总体上看，我们的前期准备还是比较充分的，**MetaQ** 各集群在高峰期表现稳定，全天表现很平稳，个别订阅组对消息进行重溯，部分消息有少量的堆积，但都没有对系统造成影响，效果还是非常好的。**75%**的交易在聚石塔上完成，实时直播间交易统计延迟在 **1s** 左右，加减库存做到零超卖。

小结

目前分布式消息中间件产品已经服务于整个集团，支持了阿里集团各个公司的 **500** 多个业务应用系统。每日处理海量消息超 **350** 亿次，保证所有交易数据高可靠，高性能，分布式事务处理，是中间件团队最老牌的中间件产品之一。

4、EagleEye——分布式调用的跟踪者

综述

阿里巴巴电子商务平台现在是一个由很多个应用集群组成的非常复杂的分布式系统。这些应用里面主要有处理用户请求的前端系统和有提供服务的后端系统等，各个应用之间一般有 **RPC** 调用和异步消息通讯两种手段，**RPC** 调用会产生一层调一层的嵌套，一个消息发布出来更会被多个应用消费。另外，应用还会访问分库分表的数据库、缓存、存储等后端，以及调用其他外部系统如支付、物流、机彩票等。

请试想一下，现在淘宝一个买家点击下单按钮所产生的网络请求到达淘宝服务器之后，就会触发淘宝内网数百次的网络调用。这些调用中有哪些出问题会影响这次交易，有哪些步骤会拖慢整个处理流程，双十一的交易高峰需要给应用集群分配多少台机器，这些都是支撑双十一需要考虑的。但是调用环境的复杂度，已经很难用人力去做准确的分析和评估了，这时候 **EagleEye** 就派上了用场。

4.1、EagleEye

EagleEye（鹰眼）是 **Google** 的分布式调用跟踪系统 **Dapper** 在淘宝的实现。分布式调用跟踪的意思，就是对一次前端请求产生的分布式调用都汇总起来做分析。同一次请求的所有相关调用的情况，在 **EagleEye** 里称作调用链。同一个时刻某一台服务器并行发起的网络调用有很多，怎么识别这个调用是属于哪个调用链的呢？我们可以在各个发起网络调用的中间件上下手。

在前端请求到达服务器时，应用容器在执行实际业务处理之前，会先执行 **EagleEye** 的埋点逻辑（类似 **Filter** 的机制），埋点逻辑为这个前端请求分配一个全局唯一的调用链 **ID**。这个 **ID** 在 **EagleEye** 里面被称为 **TraceId**，埋点逻辑把 **TraceId** 放在一个调用上下文对象里面，而调用上下文对象会存储在 **ThreadLocal** 里面。调用上下文里还有一个 **ID** 非常重要，在 **EagleEye** 里面被称作 **RpcId**。**RpcId** 用于区分同一个调用链下的多个网络调用的发生顺序和嵌套层次关系。对于前端收到请求，生成的 **RpcId** 固定都是 0。

当这个前端执行业务处理需要发起 **RPC** 调用时，淘宝的 **RPC** 调用客户端 **HSF** 会首先从当前线程 **ThreadLocal** 上面获取之前 **EagleEye** 设置的调用上下文。然后，把 **RpcId** 递增一个序号。在 **EagleEye** 里使用多级序号来表示 **RpcId**，比如前端刚接到请求之后的 **RpcId** 是 0，那么它第一次调用 **RPC** 服务 A 时，会把 **RpcId** 改成 0.1。之后，调用上下文会作为附件随这次请求一起发送到远程的 **HSF** 服务器。

HSF 服务端收到这个请求之后，会从请求附件里取出调用上下文，并放到当前线程 **ThreadLocal** 上面。如果服务 A 在处理时，需要调用另一个服务，这个时候它会重复之前提到的操作，唯一的差别就是 **RpcId** 会先改成 0.1.1 再传过去。服务 A 的逻辑全部处理完毕之后，**HSF** 在返回响应对象之前，会把这次调用情况以及 **TraceId**、**RpcId** 都打

印到它的访问日志之中，同时，会从 ThreadLocal 清理掉调用上下文。如图 4-1 展示了一个浏览器请求可能触发的系统间调用。

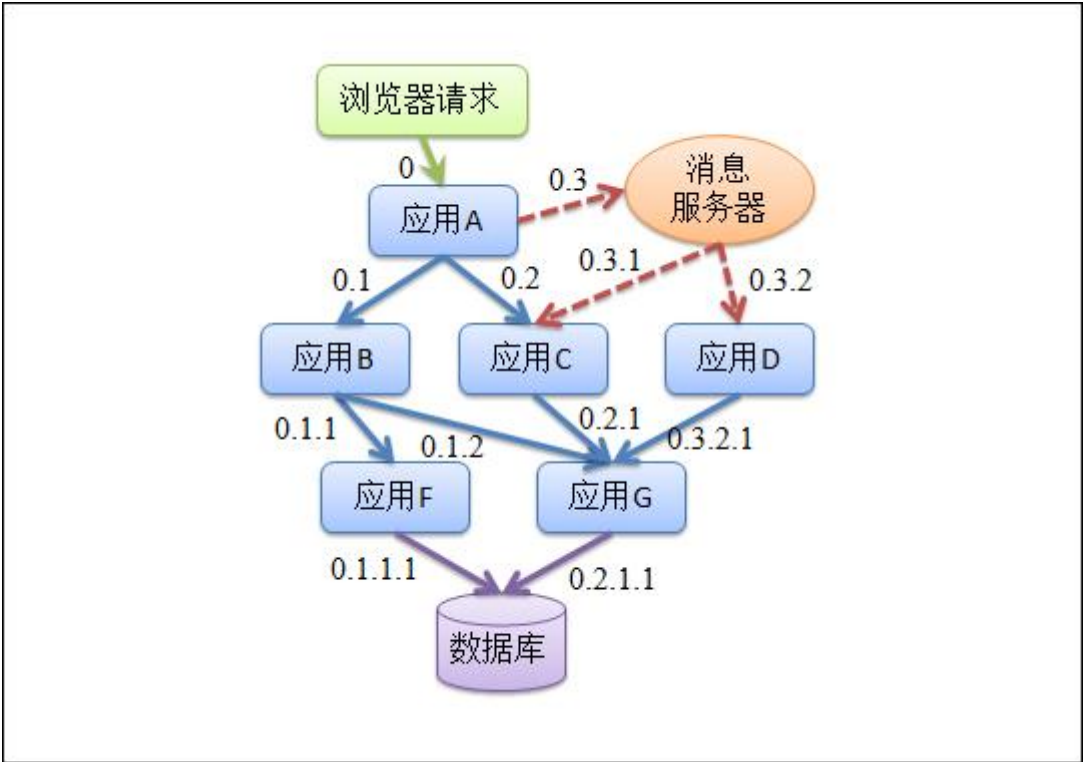


图 4-1-一个浏览器请求可能触发的系统间调用

图 4-1 描述了 EagleEye 在一个非常简单的分布式调用场景里做的事情，就是为每次调用分配 Traceld、Rpclid，放在 ThreadLocal 的调用上下文上面，调用结束的时候，把 Traceld、Rpclid 打印到访问日志。类似的其他网络调用中间件的调用过程也都比较类似，这里不再赘述了。访问日志里面，一般会记录调用时间、远端 IP 地址、结果状态码、调用耗时之类，也会记录与这次调用类型相关的一些信息，如 URL、服务名、消息 topic 等。很多调用场景会比上面说的完全同步的调用更为复杂，比如会遇到异步、单向、广播、并发、批处理等等，这时候需要妥善处理好 ThreadLocal 上的调用上下文，避免调用上下文混乱和无法正确释放。另外，采用多级序号的 Rpclid 设计方案会比单级序号递增更容易准确还原当时的调用情况。

最后，EagleEye 分析系统把调用链相关的所有访问日志都收集上来，按 Traceld 汇总在一起之后，就可以准确还原调用当时的情况了。

ac18287913742691251746923					
调用链入口 IP: 开始时间: 2013-07-20 05:25:25.174, 调用链总时长: 16s262ms 日志原文					
应用名	IP	类型	状态	大小	服务/方法
mtop		TRACE	OK	-	http://api.m.taobao.com/rest/api3.do
wdc		HSF	OK	8.5KB	
sinus		HSF	TIMEOUT	6.9KB	wireless.TmallBagInterface@buildConfirmOrder~P
(tair@wireless)		TAIR	NOTEXSE	65B	
(tair@uic)		TAIR	OK	57B	
buyapi		HSF	OK	11.6KB	
cartapi		HSF	OK	2.4KB	
delivery		HSF	OK	844B	
tradeplatform		HSF	OK	1.3KB	
inventoryplatfo		HSF	OK	6.1KB	
inventoryplatfo		HSF	OK	8.9KB	
inventoryplatfo		HSF	OK	5.0KB	
delivery		HSF	OK	6.5KB	
delivery		HSF	OK	6.3KB	
delivery		HSF	TIMEOUT	6.2KB	delivery.DeliveryTradeService@getItemsSupportPost~LL
(tair@1)		TAIR	CONNERR	-	GET:group_1:214
tradeplatform		HSF	OK	727B	
logisticscenter		HSF	OK	805B	
ump		HSF	OK	13.6KB	
delivery		HSF	OK	11.4KB	
tradeplatform		HSF	OK	9.4KB	
tradeplatform		HSF	OK	705B	
tradeplatform		HSF	OK	815B	

图 4-2-一个典型的调用链

如图 4-2 所示，就是采集自淘宝线上环境的某一条实际调用链。调用链通过树形展现了调用情况。调用链可以清晰的看到当前请求的调用情况，帮助问题定位。如上图，mtop 应用发生错误时，在调用链上可以直接看出这是因为第四层的一个(tair@1)请求导致网络超时，使最上层页面出现超时问题。这种调用链，可以在 EagleEye 系统监测到包含异常的访问日志后，把当前的错误与整个调用链关联起来。问题排查人员在发现入口错误量上涨或耗时上升时，通过 EagleEye 查找出这种包含错误的调用链采样，提高故障定位速度。

调用链数据在容量规划和稳定性方面的分析

如果对同一个前端入口的多条调用链做汇总统计，也就是说，把这个入口 URL 下面的所有调用按照调用链的树形结构全部叠加在一起，就可以得到一个新的树结构（如图 4-3 所示）。这就是入口下面的所有依赖的调用路径情况。

层次	名称	应用	GPS	峰值 GPS	调用 次数	平均 耗时	本地 耗时	依赖度	耗时 比例	标记
根	com.taobao.android.hsf	hsf	118.13	269.64	1.0	435ms	51ms	100.0%	11.86%	◇
1	com.taobao.android.hsf.adapter	hsf.adapter	226.02	483.12	1.04	8ms	8ms	98.51%	3.64%	◇
2	hsf.adapter.impl	hsf.adapter.impl	282.74	533.96	4.06	0ms	0ms	58.94%	0.02%	◇
2	hsf.adapter.impl	hsf.adapter.impl	74.41	170.57	1.77	0ms	0ms	35.56%	0.08%	◇
1	com.taobao.android.hsf.adapter.impl	hsf.adapter.impl	183.31	357.96	1.71	3ms	2ms	90.58%	1.0%	◇
2	hsf.adapter.impl	hsf.adapter.impl	182.59	356.96	1.71	0ms	0ms	90.28%	0.32%	◇
1	com.taobao.android.hsf.adapter.impl	hsf.adapter.impl	130.91	244.85	1.71	19ms	7ms	64.79%	1.98%	◇
2	hsf.adapter.impl	hsf.adapter.impl	219.58	412.81	2.88	3ms	3ms	64.52%	1.37%	◇
2	hsf.adapter.impl	hsf.adapter.impl	219.55	412.73	2.88	1ms	1ms	64.51%	0.45%	◇
2	hsf.adapter.impl	hsf.adapter.impl	131.28	243.79	1.72	0ms	0ms	64.51%	0.12%	◇
2	hsf.adapter.impl	hsf.adapter.impl	125.25	231.11	1.7	0ms	0ms	62.2%	0.12%	◇
1	com.taobao.android.hsf.adapter.impl	hsf.adapter.impl	124.27	230.45	1.1	25ms	19ms	96.07%	4.66%	强依赖
2	hsf.adapter.impl	hsf.adapter.impl	120.47	223.52	1.07	5ms	5ms	95.48%	1.3%	◇
1	com.taobao.android.hsf.adapter.impl	hsf	116.05	247.93	1.0	4ms	4ms	98.25%	1.05%	◇
1	com.taobao.android.hsf.adapter.impl	hsf.adapter.impl	114.76	216.99	1.08	43ms	37ms	90.14%	8.28%	强依赖
1	com.taobao.android.hsf.adapter.impl	hsf.adapter.impl	113.49	214.98	1.0	5ms	5ms	96.08%	1.16%	强依赖
1	com.taobao.android.hsf.adapter.impl	hsf.adapter.impl	111.23	236.02	1.03	3ms	3ms	91.82%	0.68%	强依赖
1	com.taobao.android.hsf.adapter.impl	hsf.adapter.impl	111.2	208.29	1.05	13ms	13ms	89.46%	2.88%	◇
1	com.taobao.android.hsf.adapter.impl	hsf.adapter.impl	110.24	213.12	1.0	214ms	207ms	93.32%	44.5%	强依赖
2	hsf.adapter.impl	hsf.adapter.impl	116.92	220.86	1.07	5ms	5ms	92.59%	1.28%	◇
1	com.taobao.android.hsf.adapter.impl	hsf	106.5	205.27	1.0	3ms	3ms	90.16%	0.81%	◇
1	com.taobao.android.hsf.adapter.impl	hsf.adapter.impl	103.07	194.45	1.05	4ms	4ms	83.05%	0.94%	◇
1	com.taobao.android.hsf.adapter.impl	hsf	58.47	141.23	1.0	12ms	12ms	49.5%	1.37%	◇
1	com.taobao.android.hsf.adapter.impl	hsf.adapter.impl	48.9	104.22	1.45	22ms	2ms	28.54%	0.2%	◇
2	com.taobao.android.hsf.adapter.impl	hsf.adapter.impl	48.89	104.22	1.45	19ms	7ms	28.53%	0.75%	◇



图 4-4-弱依赖发生错误时的调用链

如图 4-4 所示，应用 **tee** 由于存在调用错误，在 **EagleEye** 的调用链里被标记为红色，但是它的异常并没有影响调用链继续往下执行。**EagleEye** 里面把这种依赖识别为弱依赖。那么强依赖又是怎样的呢？如图 4-5 所示，**tradeplatform** 的这个调用出现了错误，导致当前的入口请求被直接中断（读者可以对比上图的弱依赖里面，**tradeplatform** 调用正常返回之后还有很多其他的调用会发生），**EagleEye** 把这种类型的依赖标识作强依赖。在双十一大并发的高压之下，任何错误出现的可能都会被成倍放大。对于系统内的依赖情况我们需要提前做好甄别，尽量降低在关键路径上的强依赖发生错误的机会，并把非关键依赖降级为弱依赖。



图 4-5-强依赖发生错误时的调用链

对于弱依赖，我们也不能掉以轻心。**EagleEye** 识别了另外一种潜在的问题模式。请看下图的调用链，**delivery** 应用发生错误时，从调用链来看，尽管它是弱依赖，但是因为它的错误占用了 3 秒时间，导致整个页面的响应也长达 3 秒。从用户角度看，这是非常不好的体验；从系统层面看，一个弱依赖错误导致系统一个处理线程无法释放，意味着

如果这个弱依赖真的挂了，系统此刻大部分处理线程会有可能都堵塞在这个服务调用上，从而整个系统的吞吐量会降低很多。



图 4-6-弱依赖的超时异常导致线程堵塞

EagleEye 可以得到相关的依赖列表，并且对依赖的错误影响作了一定判别，通过这份数据，可以再人为模拟出问题场景来验证这个异常是否确实存在，处理后再验证是否彻底修复。相关的内容可以利用后面提到的稳定性平台做检测。

4.2、EagleEye 双 11 准备与优化

- 应对双十一的日志输出问题

EagleEye 的调用日志是每次网络调用都会打印一条，在双十一高峰期日志的打印量会非常大，如何降低 EagleEye 的日志输出对应用的影响呢？

采用通用日志框架会引入很多不需要的特性，带了更多性能损耗，为了提高性能和对写日志底层做更细致的控制，EagleEye 自己实现了日志输出：利用无锁环形队列做日志异步写入来避免 hang 住业务线程，调节日志输出缓冲大小，控制每秒写日志的 IO 次数等。

另外还很重要的一点就是设置全局采样开关，用来在运行期控制调用链的采样率。所谓调用链采样，就是根据 Traceld 来决定当前的这一次访问日志是否输出。比如采样率被设置为 10 时，只有 $\text{hash}(\text{traceld}) \bmod 10$ 的值等于 0 的日志才会输出，这样可以保证有一部分调用链日志完全不输出，有一部分调用链会完整输出。由于核心入口的调用量都在每日百万次以上级别，样本空间足够大，实测开启 1/10 的采样下，调用量统计误差在 0.1% 左右，大于 10 万以上的调用统计点，误差超过 5% 的概率为 2% 左右，是可以接受的，因此采样会作为常态化的配置存在。

小结

EagleEye 是基于网络调用日志的分布式跟踪系统，它可以分析网络请求在各个分布式系统之间的调用情况，从而得到处理请求的调用链上的入口 URL、应用、服务的调用关

系，从而找到请求处理瓶颈，定位错误异常的根源位置。同时，业务方也可以在调用链上添加自己的业务埋点日志，使各个系统的网络调用与实际业务内容得到关联。

5、数据层——分布式数据存储的桥梁

综述

大型互联网架构中,数据存储会面临读写容量瓶颈问题，像淘宝双十一活动，核心数据存储集群读写日访问量可以达到 100 亿以上,在这种场景下,单机数据库方式必定面临极大挑战,类似的场景也在一些传统使用 IOE 的企业中成为一种制约业务发展的致命要素。而在阿里集团内,TDDL 体系就是解决此种场景的利器，这个体系是基于廉价 pc 和开源 mysql、以客户端依赖方式、分库分表为主要手段、集中化数据库配置等几个关键要素构建起来，成为阿里集团接入 mysql 的标准，提供整个集团上千个应用的数据库访问。

5.1 TDDL

TDDL 体系核心作用在于两个方面,1.直接提供分库分表，读写分离等解决数据库 Scale Out 问题的功能. 2.基于配置模型构建的包括数据库在线扩容、准实时数据同步服务、运维平台等支撑系统，接下来简单介绍下几个重要 feature。

分库分表

TDDL 主要部署在 ibatis 或者其他 ORM 框架之下，JDBC Driver 之上，整个中间件实现了 JDBC 规范， 所以可以将其当作与普通数据源实例化并且注入到各种 ORM 框架中使用。

TDDL 现时架构分为了 3 层，最上层的 TDataSource 负责分库分表路由，中间层 TGroupDataSource 负责主备切换和读写分离，最下层 TAtomDataSource 一对一对应数据库连接池，进行动态的数据库配置信息变更等。3 层数据源都可以单独使用，应对不同的应用场景，如图 5-1 所示。

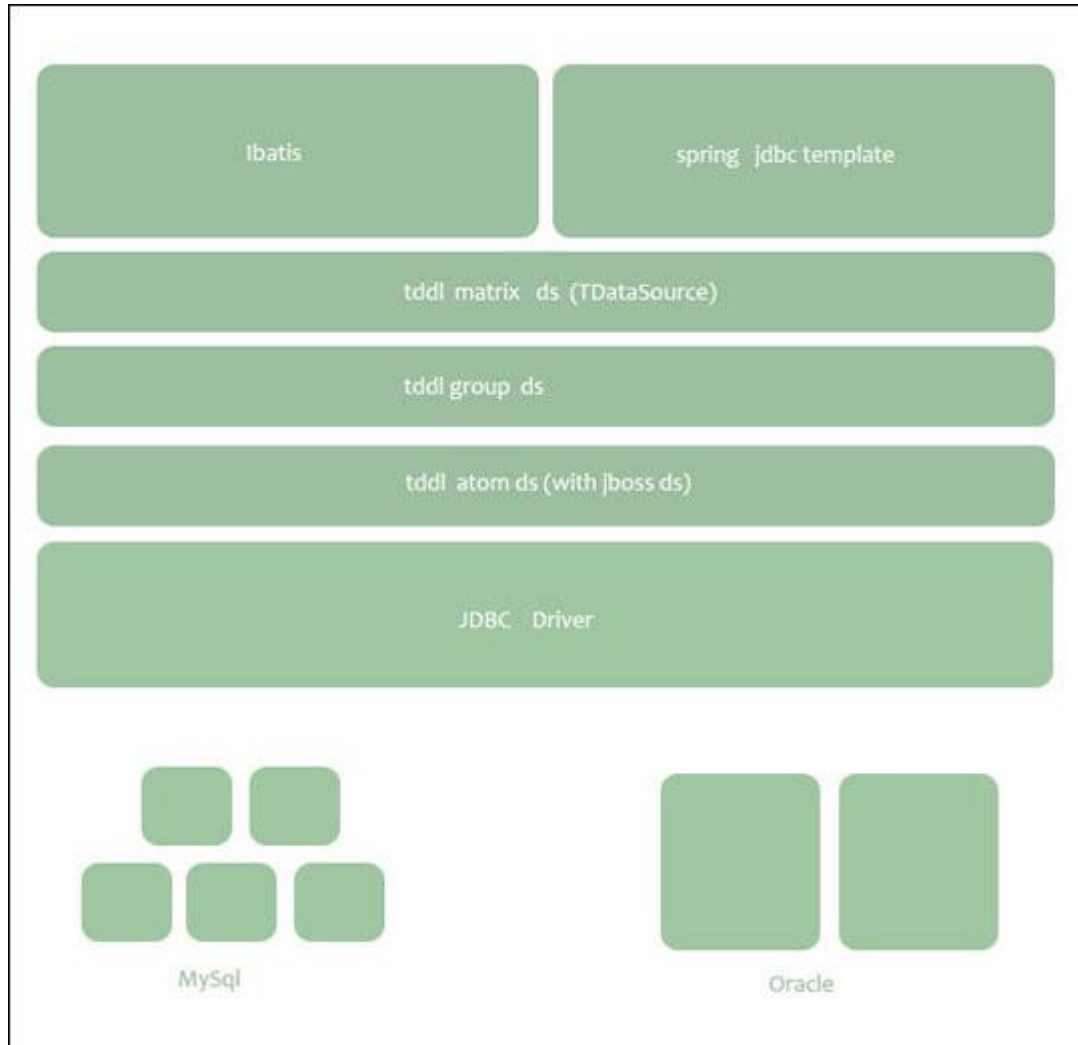


图 5-1-TDDL 体系结构

作为分布式数据库，首要工作是解决单机的访问瓶颈同时尽可能完整保留单机的特性，这也就意味着对于用户的一个操作，可能分拆到多个数据节点进行数据处理，然后等待节点数据返回再按照 **sql** 语义进行合并。

一般分库分表的具体执行方式可以分为几个步骤，包括 **sql** 解析，规则计算，表名替换，执行，结果合并。**sql** 解析主要将 **sql** 进行词法解析和语法解析，将其变成程序可识别的语法树，然后遍历整棵 **sql** 语法树取出我们所关心的内容，这些内容包括分库的字段是什么，对应的值是什么，是否有 **limit, order by, max, min** 等关键字，其中前者主要为了提供相关数据提供给规则进行计算，后者主要是在该 **sql** 跨多个表或者库的时候，在 **sql** 返回结果后按照这些关键字的语义进行合并。规则计算中，我们将解析得到的字段和值作为参数传递给规则引擎，这些规则引擎会进行类似简单数字取模，日期计算,组合计算等操作。规则计算完毕后，我们能够得到真实的表和数据库，那么接下来要做的事情其实是，如果有分表，那么得把传入的 **sql** 中实际不存在的表替换成真实的表，并且必要情况下对 **sql** 做一些改变，这个阶段我们称之为表名替换。执行阶段就是将 **sql** 发

送到数据库服务端，这里有串行和并行两种模式。如果单个 sql 最终需要通过多个子 sql 的形式在节点上执行，返回的结果可能不能直接返回给用户的，因为还需要根据原始 sql 的含义进行合并，这些合并一般遵循逐条流式进行，不能进行这种方式合并的 sql 一般放弃支持，因为强行合并，会有极大的内存溢出风险。

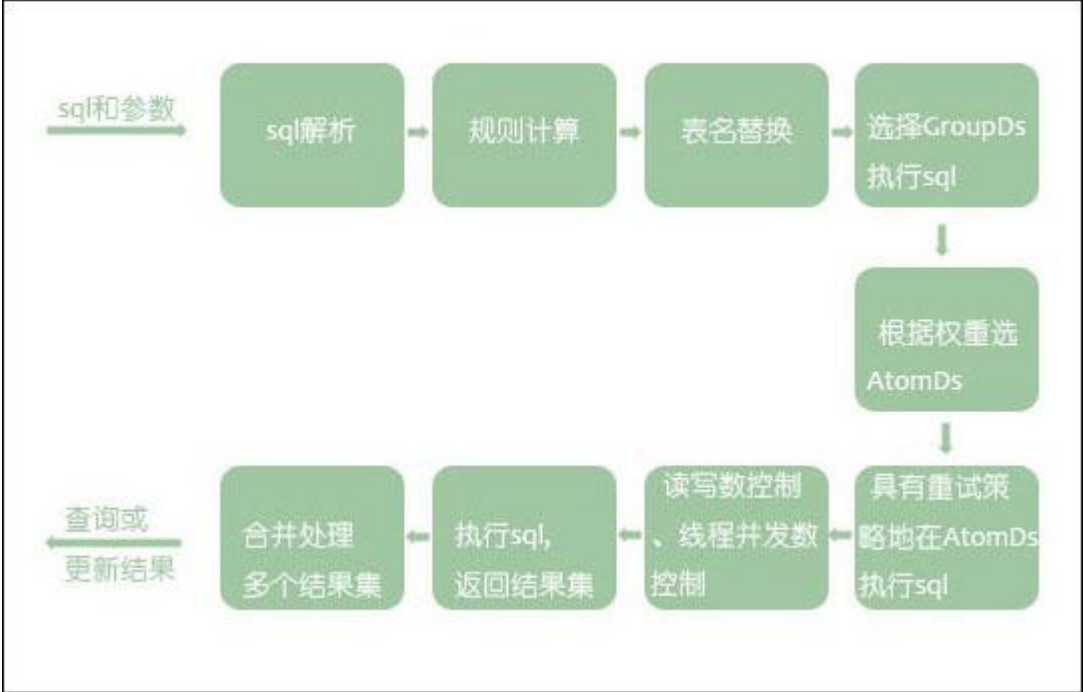


图 5-2-TDDL 总体流程图

简单描述了下整个分库分表的执行逻辑，这套逻辑经过多年的沉淀，相对比较成熟。但是在这些成熟逻辑的背后，一系列数据库运维支撑系统在保障数据库的正常有序运转，可以想象，如果数据库机器有成千上万台，每天坏个几台，上千个应用，隔三差五说数据库容量不够，种种挑战扑向你的时候，没有相对自动化的运维平台，那肯定是相当苦逼的。

在线数据库扩容

TDDL 体系中数据库在线扩容和数据准实时同步是两个重要的组成部分，这边简单介绍下前者。在线数据库扩容，重点在于在线，也就是不影响业务正常的写入，因为数据库扩容涉及到的数据迁移需要时间，未采取其他措施的情况下，显然无法实现不影响业务。那么我们的方案将这个扩容分为几个阶段，包括全量迁移，增量同步，切换数据库。全量迁移阶段，我们将数据库数据扫描出来复制到目标，完成后，从事先记录的增量位点开始追赶，直到赶上当前的进度。这种动态平衡的过程可以维持一段时间，应用安排某一天业务低谷期，直接切换到新的数据库即可。

数据准实时同步

如同流动的资金才能发挥作用一样，流动的数据也能够创造出大价值。每天，淘宝的交易订单，商品，用户等核心数据通过团队的数据同步管道进行了流转，流向搜索，流向

广告，流向大数据等等目的地。这套系统的实现原理基于增量日志的传输与重现，如mysql的binlog,oracle的redo log,hbase的HLog，系统将这些源系统生成的对应日志拉到独立机器进行解析和必要变换，通过有序消息中间件传递给多方消费者使用。原理十分简单,但是在这套系统上面也沉淀了很多的经验。

5.2、TDDL 双 11 准备与优化

- 数据库扩容

双十一的数据库访问量大过平时访问量的好几倍，所以我们将数据库进行大幅度的扩容，阿里的数据库部署结构主要是单机器多实例方式，在双十一之前，一些容量不足的数据库集群必须将原本在一台机器上的实例拆分到不同机器上，通过动态推送配置不影响正常的业务访问，而原本拆分数量不够的数据库通过在线数据库扩容系统在短时间之内 resharding,达到双十一的容量需求。

- 数据准实时同步的扩容

双十一之前，整个集团有大量的核心应用使用数据准实时同步系统进行多维度的数据同步，而双十一的写入量导致该系统的容量需要进一步扩展，而最终这个系统达到了近 500 台机器的规模，保证双十一写入高峰同步延迟达到期望。

- 前端的链接数调整和 mysql 并发控制 patch 的使用

双十一之前，应用机器进行了大幅度扩容，而其连接池模型导致单机 mysql 总链接数达到了一个峰值,有可能在零点高峰导致 mysql 活动链接过多而 hang 住。那么 dba 同学通过 tddl 的动态配置将压力最大的几个核心应用单台机器最大连接池保持在 3 个以下。另外将这几个可能出现压力过大的 mysql 集群打上并发控制 patch,在流量超过阈值的时候，对访问数据库的链接进行排队处理。这个解决方案保证了数据库在双十一不至于 hang 住从而产生雪崩效应。

小结

TDDL 其实并不是一个产品，它是一个完整的生态系统，这个生态系统以 tddl 产品为核心，构建出应用接入层，运维平台，数据支撑等组件。双十一中，TDDL 整个产品体系支撑着整个业务系统的巨大流量，稳定高效，最高一个数据库集群峰值 40w/s 的写入，系统也是十分淡定。而未来，TDDL 团队将继续本着创新务实的心态挖掘分布式数据库领域更多有价值的内容。

6、应用服务器——系统运行的 托管员

综述

阿里巴巴集团有国内最大规模的 Java 系统，几万台的应用服务器规模也空前庞大，目前主要使用的应用服务器有 Tomcat, JBoss 和 Jetty 三种。阿里巴巴自从 2004 年开始转向 Java 技术平台后，先后经历了从 WebLogic 到 Jboss 和 Tomcat 迁移。到了 2008 年，随着更为轻量级的 Tomcat 和 Jetty 容器的迅速发展，越来越多的应用系统开始尝试使用 Tomcat 或 Jetty 作为底层应用服务器。2013 年上半年，阿里巴巴集团中间件成立了独立的应用服务器团队，主要面向整个集团进行应用服务器相关的工作，目前在公司内部主推 Tomcat 服务器。

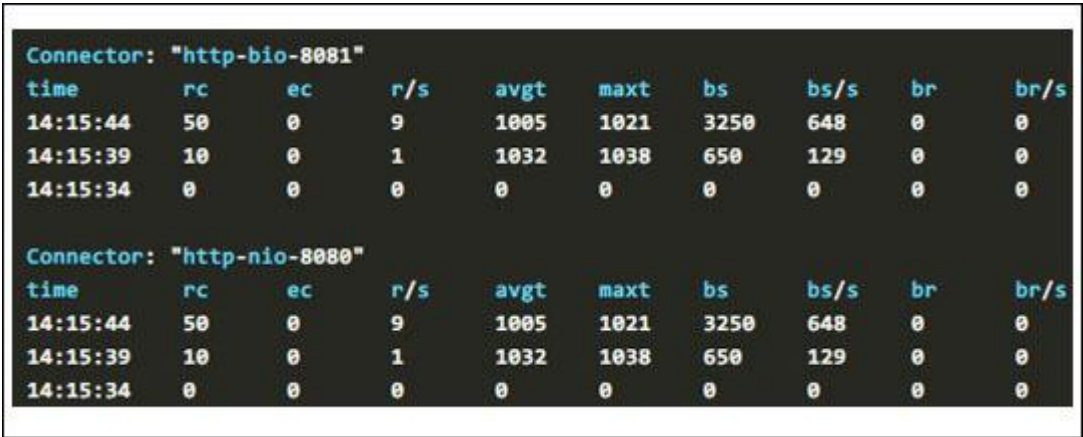
本文将从中间件团队在 2013 年双十一大促前针对应用服务器上进行的工作展开，重点讲解 Tomcat 监控诊断工具，以及 Pandora 隔离技术两方面内容。更多关于应用服务器的内容，可以到中间件团队博客（<http://jm.taobao.org>）上查看。

6.1、Tomcat 监控管理工具

Tomcat Monitor 模块是一个 Tomcat 的监控和诊断模块，提供了一些基本的工具，可以对 Tomcat 的连接池、线程、内存、类加载以及 JVM 相关等进行监控和诊断。Tomcat Monitor 的出现，解决了广大开发人员无法快速定位线上问题的尴尬问题，同时也帮助开发人员能够通过简单且统一的命令行工具来排查问题、查看程序运行时状态，而不需要使用各种包括 jmap、jstat 和 BTrace 等工具。

Tomcat Monitor 模块集成于 Tomcat 服务器内部，能够对线程、连接池、内存和类加载等方面进行详细且实时的监控与诊断。

- 进行连接池的监控和管理



Connector: "http-bio-8081"									
time	rc	ec	r/s	avgt	maxt	bs	bs/s	br	br/s
14:15:44	50	0	9	1005	1021	3250	648	0	0
14:15:39	10	0	1	1032	1038	650	129	0	0
14:15:34	0	0	0	0	0	0	0	0	0

Connector: "http-nio-8080"									
time	rc	ec	r/s	avgt	maxt	bs	bs/s	br	br/s
14:15:44	50	0	9	1005	1021	3250	648	0	0
14:15:39	10	0	1	1032	1038	650	129	0	0
14:15:34	0	0	0	0	0	0	0	0	0

图 6-1-查看 Tomcat 连接基本状态

- 检测出当前 Tomcat 服务器中那些慢连接

Connector: "http-nio-8080"						
start	addr	url	method	pt	bs	br
15:56:31	127.0.0.1	/ServletTest/sleep	GET	9139	71	0
15:56:36	127.0.0.1	/ServletTest/sleep	GET	4405	71	0
15:56:36	127.0.0.1	/ServletTest/sleep	GET	4008	71	0
Connector: "http-bio-8081"						
start	addr	url	method	pt	bs	br
15:56:31	127.0.0.1	/ServletTest/sleep	GET	9139	71	0
15:56:36	127.0.0.1	/ServletTest/sleep	GET	4405	71	0
15:56:36	127.0.0.1	/ServletTest/sleep	GET	4008	71	0

图 6-2-检测慢连接

- 线程死锁检测

```
observer.hany@ali-59375nm:~/hello$ /opt/taobao/tomcat/bin/tm.sh thread deadlock
"pool-1-thread-2" Id=32 BLOCKED on java.lang.String@114a3c6 owned by "pool-1-thread-1" Id=31
  at test.deadlock.Task.run(Task.java:24)
    - blocked on java.lang.String@114a3c6
    - locked java.lang.String@c4cee
  at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:895)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:918)
  at java.lang.Thread.run(Thread.java:662)

Number of locked synchronizers = 1
  - java.util.concurrent.locks.ReentrantLock$NonfairSync@1a9fcea

"pool-1-thread-1" Id=31 BLOCKED on java.lang.String@c4cee owned by "pool-1-thread-2" Id=32
  at test.deadlock.Task.run(Task.java:24)
    - blocked on java.lang.String@c4cee
    - locked java.lang.String@114a3c6
  at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:895)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:918)
  at java.lang.Thread.run(Thread.java:662)

Number of locked synchronizers = 1
  - java.util.concurrent.locks.ReentrantLock$NonfairSync@11b99c4
```

图 6-3-线程死锁检测

可以看出，线程 **pool-1-thread-1** 和 **pool-1-thread-2** 发生死锁.下面两行清晰描述了死锁原因：

```
"pool-1-thread-2" Id=32 BLOCKED on java.lang.String@114a3c6 owned by "pool-1-thread-1" Id=31
"pool-1-thread-1" Id=31 BLOCKED on java.lang.String@c4cee owned by "pool-1-thread-2" Id=32
```


thread-2 阻塞在被 thread-1 锁住的对象 java.lang.String@114a3c6 上,
thread-1 阻塞在被 thread-2 锁住的对象 java.lang.String@c4cee 上,

两个线程互相等待, 导致死锁.

输出结果还显示了发生死锁的线程堆栈, 以便开发人员进一步排查发生死锁的原因.

- 诊断出 CPU 占用高的线程

显示最近一段时间 cpu% 持续过高的线程列表, 及其最近一次统计的 cpu%.

```
observer.hany@ali-59375nm:~/hello$ /opt/taobao/tomcat/bin/tm.sh thread busy
Id State      ThreadName      cpu%
25 RUNNABLE    http-nio-7001-exec-3 99
```

- 在碰到烦人的 `ClassNotFoundException` 或是 `NoClassDefFoundError` 这些异常的时候, 可以定位类的加载情况

```
$ bin/tm.sh jar locate org.apache.zookeeper.WatchedEvent
Class:   org.apache.zookeeper.WatchedEvent
Location: WEB-INF\lib\zookeeper-3.4.5.jar
```

图 6-6-检测类加载情况

针对应用服务器的监控和诊断, 后续的发展规划是在目前 Tomcat Monitor 的基础上, 集成其他诸如 [HouseMD](#) 这样优秀的 Java 监控与诊断工具, 使得不同的工具能够以一种统一方式给开发人员使用。同时, 还会和公司内部已经成熟的监控报警系统打通, 作为数据提供方来帮助更深入的监控应用的运行情况。

6.2、隔离容器 Pandora

Pandora, 中文名潘多拉, 是阿里中间件团队打造的, 基于 HSF 隔离技术构建的全新一代隔离容器。从解决二方包依赖冲突出发, 致力于统一管理通用的二方包, 包括方便的二方包升级管理, 监控和管理, 建立统一的二方包扩展编程方式等。基于 Pandora 容器基础上进行改造而来的 Pandora-Framework, 是一个类 OSGi 的模块化运行框架。它的产出, 使得 OSGi 这个一直以来隐藏在应用服务器和 IDE 工具中的神秘技术, 第一次在生产环境中走入了我们的前台应用系统。2013 年 9 月在共享业务交易流程系统上线以来, 目前将逐步应用于整个阿里交易流程系统, 构建了交易系统的模块化运行环境。

功能介绍

- 隔离

解决三方包依赖冲突问题。针对三方包的依赖冲突问题，比如：`log4j`，`httpClient`，通常我们在开发过程中，常常碰到不同的二方包依赖了不同版本的三方包。面对这种情况，我们都是使用 **Maven** 工具强行将这些三方包指定到一个版本。但是，针对那些兼容性不好的三方包，这存在很大的风险。

- 提供了一套完整的二方包大规模快速升级机制

提供方便的二方包大规模升级方式，用户只需要将自己的包及依赖的包按照隔离容器的规范放到隔离容器里面，就可以达到升级的效果。不需要业务方做任何事情。**Pandora** 容器已经和 **Freedom**（新版发布系统）打通，在原有应用发布流程上，添加了 **Pandora** 发布流程，发布的时候，可以很方便的选择需要使用那个版本的 **Pandora** 容器，哪个版本

- 运行期开关

和 **Stableswitch**（**Stableswitch** 是中间件团队开发的，嵌入在应用内部，当服务器压力比较大时，会通过开关功能来关闭一些不太重要的功能点）开关有区别，**Stableswitch** 开关是业务逻辑开关，面向的对象是应用，也就是应用里面的开关。而 **Pandora** 容器面向的是二方包的开关。运行期可以对所有应用里面使用的二方包做调配，是一个轻量级的方案。另外，相对于订阅 **diamond** 数据方式实现的开关，这个粒度更细，可以针对每一个单机进行调控。

- 监控管理

Pandora 容器提供方便的命令行模式，二方包提供者只需要简单的实现 **Pandora** 的接口，就可以实现自己的命令行命令了。比如：可以实现一个功能，在运行期查看所有使用该二方包的应用的运行期数据，方便跟踪及排查问题。

6.3、应用服务器双 11 准备与优化

这里重点讲解下 **Pandora** 容器针对交易系统在双十一之前进行的模块化改造。谈到模块化，相信很多读者都会第一时间联想到 **OSGi**。没错，**OSGi**（**JSR 291**）是 **Open Services Gateway initiative** 的缩写，为系统的模块化开发定义了一个基础规范和架构模型。迄今为止，在一些著名的 IDE 产品（**Eclipse** 是第一个也是目前最成熟的 **OSGi** 实践者）和应用服务器厂商（**IBM**、**BEA**、**Oracle**）中都已经采用了 **OSGi** 来创建“微内核与插件”的软件架构，这样一来，这些 IDE 和容器就可以被更好的模块化，并且可以在运行时被动态装配。

显然，模块化和动态化，是 **OSGi** 最显著的两大特性。模块化，尤其是他的隔离机制，基本得到了大家的认可，但是针对动态化这个特性，是公认的 **OSGi** 中最具争议的地方。

- 从实用性角度来讲，目前我们其实对于热部署，动态替换等并没有太强烈的需求，开发人员通常都能够接受应用重启。
- 从复杂性角度来讲，想要做到平滑热替换，尤其是对于那些运行期有状态的 **bundle** 而言，实现动态化相当复杂。
- 从可行性角度来讲，实现动态化，需要改变开发人员和运维人员的开发与运维习惯，在推广上面临极大的挑战。

- 废弃 OSGi，实现应用系统模块化

因此，Panodrar 容器废弃了 OSGi 框架，只是引入了 OSGi 隔离机制的思想，自己重新实现了 ClassLoader 的隔离，形成了一个全新的轻量级的隔离容器。如图 6-7 所示。

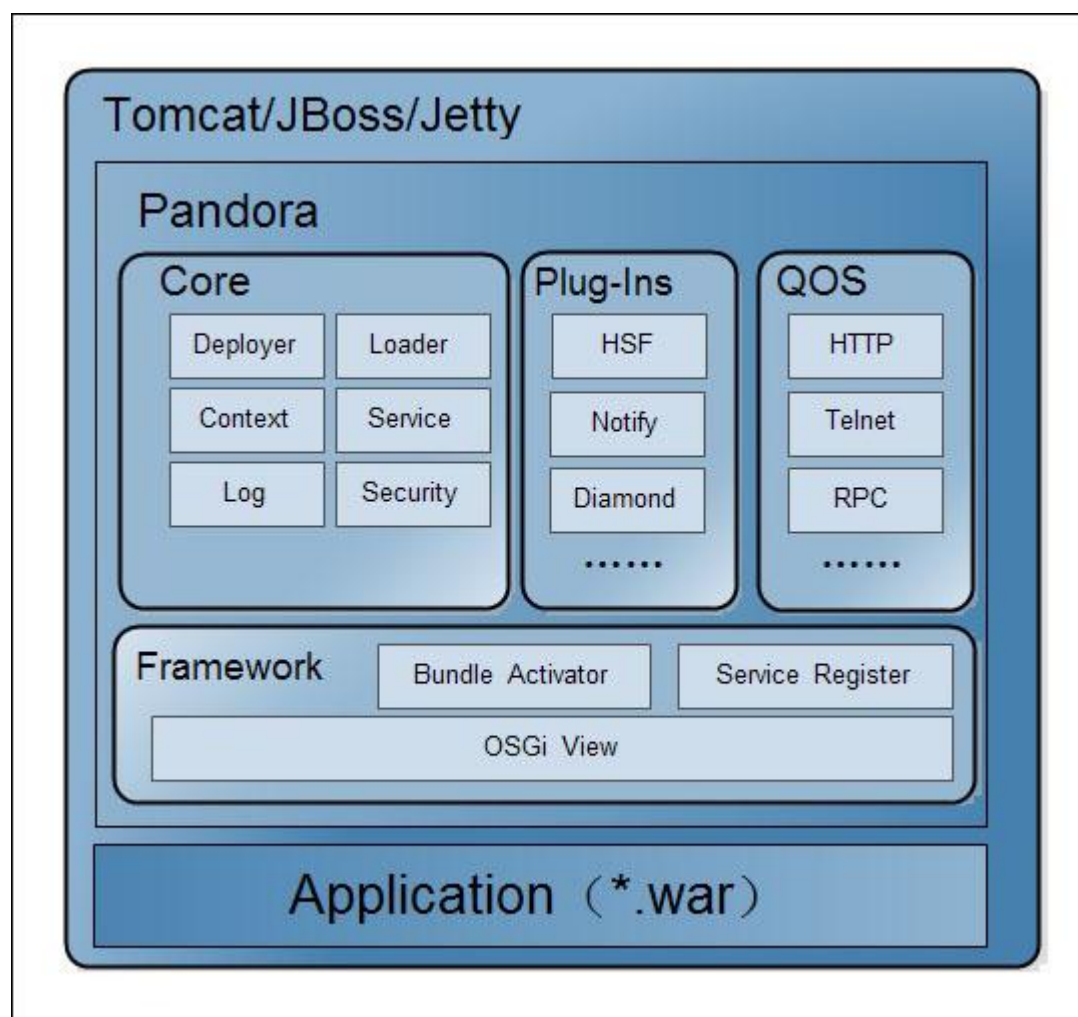


图 6-7-Pandora 体系结构

下面重点从 Bundle 和类加载两方面来讲解下 Pandora 针对业务模块化的改造。

- Bundle -- 最小的业务单元

首先引入了 Bundle 的概念，使得业务系统内部逻辑能够按照 bundle 为单元进行组织。同时提供了 Maven 插件用于 bundle 的生成，使得一个标准的 Maven Web 工程能够按照子工程为单位进行无缝迁移。

- 类加载 -- 隔离的核心

类加载机制是模块化隔离的核心。根据业务系统模块化的需要，我们需要设计一种既要使 bundle 具有严格的私密性，又要使 bundle 和主应用以及 bundle 之间具有灵活的互通性，因此重新设计了类加载机制。大体的类加载可以分为以下三步：

第一步：尝试从 **import** 中加载。

Pandora 在加载 **bundle** 的类的时候，首先会判断当前类是否需要从其他 **bundle** 中获取一些共享类。

第二步：尝试从 **bundle** 自己类路径下进行类加载。

Bundle 的私有性需求已经规定了，每个 **bundle** 都应该有能力和外部业务系统环境隔离开来，因此一些三方包的加载，**bundle** 自身目录下的都会优先于业务系统环境。

第三步：尝试从外部三方容器中加载。

如果 **bundle** 声明了需要从外部三方容器的 **biz classloader** 中来加载这个类，那么会尝试从这个 **biz classloader** 中去加载。

小结

总的来说，**Pandora** 的这次改造，伴随着阿里交易系统第三次大规模的改造升级过程，不仅满足了业务模块化改造的需求，同时也使得 **Pandora** 容器在原有解决二方包问题的基础上，新增解决业务系统模块化改造需求的能力。传统 IT 公司的出现与发展远早于互联网，因此，很多早期的应用服务器，包括 **WebLogic** 和 **WebSphere** 在内，更多都是为大型的单机的系统设计，尤其是从运维角度来说，都已经无法满足互联网时代大规模分布式系统。越来越多的互联网应用转移到了以 **Tomcat**、**Jboss** 和 **Jetty** 等为代表的轻量级的应用服务器上。然而，随着互联网应用多样性的不断发展，分布式系统规模的不断增大，尤其是移动互联网时代的到来，目前的主流服务器可能都无法满足未来日益变化需求，因此我们还正在探索下一代应用服务器的路上。

7、稳定性平台——系统稳定运行的保障者

综述

大多数互联网公司都会根据业务对自身系统做一些拆分，大变小，1 变 n，系统的复杂度也 n 倍上升。当面对几十甚至几百个应用的时候，再熟悉系统的架构师也显得无能为力。稳定性平台从 2011 年就开始了依赖治理方面的探索，目前实现了应用级别和接口级别的依赖自动化治理。在 2013 的双 11 稳定性准备中，为共享交易链路的依赖验证和天猫破坏性测试都提供了支持，大幅度减小了依赖治理的成本和时间。另一方面，线上容量规划的一面是稳定性，另一面是成本。在稳定性和成本上找到一个最佳的交汇点是线上容量规划的目的所在。通过容量规划来进行各个系统的机器资源分配，在保证系统正常运行的前提下，避免对机器资源的过度浪费。

7.1、依赖治理实践

依赖治理的一些基础概念

依赖模型分为关系、流量、强弱，实际的使用场景有：

- 依赖关系：线上故障根源查找、系统降级、依赖容量、依赖告警、代码影响范围、系统发布顺序、循环依赖等。
- 依赖流量：分配流量比、优化调用量、保护大流量。
- 依赖强弱：线上开关演练，系统改造评估。

关系数据可以通过人工梳理、代码扫描、线上端口扫描的方式获取。流量数据可以通过分析调用日志的方式获取。强弱数据则必须通过故障模拟才能拿到。故障模拟分为调用屏蔽和调用延迟两种情况，分别代表服务不可用和服务响应慢的情况。依赖的级别分为应用级依赖和接口方法级依赖，两个级别的故障模拟手段完全不同，下面分开来描述。

应用级别强弱依赖检测

应用级别故障模拟比较做法有几种，即：修改代码，写开关，远程调试，填错服务的配置项。这几种方式对配置人要求相对较高，并且对应用代码有一定的侵入性，所以没有被我们采用。Linux 有一些原生的命令（如 iptables、tc）默认就有流量流控功能，我们就是通过控制 linux 命令来达到模拟故障的效果。命令举例：

```
iptables -A INPUT -s xxx.xxx.xxx.111 -j DROP
```

上面的命令表示：当前主机屏蔽掉来自 xxx.xxx.xxx.11 的网络包。

```
tc qdisc del dev eth0 root
```

```
tc qdisc add dev eth0 root handle 1: prio
```

```
tc qdisc add dev eth0 parent 1:1 handle 10: netem delay 6000ms
```

```
tc filter add dev eth0 protocol ip parent 1: prio 1 u32 match ip
dst xxx.xxx.xxx.111/32 flowid 1:1
```

命令表示：在网卡 `eth0` 上面设置规则，对 `xxx.xxx.xxx.111` 的网络包进行延迟，延迟的时间是 `6000ms`。

接口级别强弱依赖检测

理想情况下，我们希望确定任意一次远程方法调用的强弱，确定到接口方法级别的强弱数据。要想达到这个目的，就只能在通信框架和一些基础设施上面做文章。基于这个思路，我们设计了接口级别强弱依赖检测的方案。方案如下：

过滤规则配置组件(服务器端)

过滤规则配置组件提供一个 `web` 界面给用户，接受用户配置的屏蔽指令和测试机器 IP 信息，并把配置信息更新到配置中心组件中去。

配置的规则举例：

```
client|throw|xxx.ItemReadService:1.0.0.daily@queryItemById~lQA
|java.lang.Exception

client|wait|xxx.ItemReadService:1.0.0.daily@queryItemById~lQA|
4000
```

上面的规则分别表示在客户端发起对远程接口 `xxx.ItemReadService:1.0.0.daily` 的 `queryItemById~lQA` 调用时，在客户端模拟一次异常或延迟 `4000` 毫秒后调用。

配置中心组件

配置中心组件的主要作用是接受客户端（过滤规则配置组件）发来的配置信息，持久化配置信息到存储介质中，并实时把配置信息实时推送到配置中心的所有客户端（即每一个故障模拟组件）。此部分功能通过中间件开源产品 `Diamond` 实现。

分布式服务调用组件

发生 `RPC` 调用时，会传递一些调用信息，如：`RPC` 发起者的 IP、当前的方法名称、下一级调用的方法名称。

故障模拟组件

故障模拟组件是一个插件，可以被服务调用组件（`HSF`）加载。插件可以接受配置中心推送的配置信息，在服务调用组件发生调用前都比对一下据配置信息的内容，当 `RPC` 发起者的 IP、调用方法都合条件的时候，发生故障模拟行为，从而达到故障模拟的效果。

7.2、容量规划实践

线上容量规划最重要的一个步骤为线上压力测试，通过线上压力测试来得知系统的服务能力，同时暴露一些在高压场景下才能出现的隐藏系统问题。我们搭建了自己的线上自动压测平台来完成这一工作，线上自动压测归纳起来主要包含 4 种模式：模拟请求、复制请求、请求引流转发以及修改负载均衡权重。

模拟请求

完全的假请求，可以通过代码或者采用工具进行模拟，常用到的工具有 `http_load`、`webbench`、`apache ab`、`jmeter`、`siege` 等。模拟请求有一个很明显的问题，即如何处理“写请求”？一方面由于“写请求”的场景不大好模拟（一般需要登录），另一方面“写请求”将要面临如何处理一致性场景和脏数据等。模拟请求方式的压测结果准确性我们认为是最底的。

复制请求

可以看成是半真实的假请求。说它半真实，因为它是由复制真实请求而产生。说它是假请求，因为即使复制的真实请求，它的响应是需要被特殊处理的，不能再返回给调用方（自从它被复制的那一刻，它就已经走上了不真实的轨道）。复制请求同样可以通过代码实现（比如我们有通过 `btrace` 去复制对服务的调用），此外也有一些比较好用的工具：比如 `tcpcopy` 等。如果想在 `nginx` 上做请求复制，可以通过 `nginx` 的 `nginx post_action` 来实现。“复制请求”模式被压测的那台机器是不能提供服务的，这将是一个额外的成本，此外复制请求模式的压测结果准确性也会由于它的半真实而打上折扣。

请求引流转发

完全真实的压测模型，常用于集群部署的 `web` 环境当中。我们对于 `apache` 和 `nginx` 的系统基本上都采取这种方式来做线上压力测试。用到的方式主要通过：`apache` 的 `mod_jk` 和 `mod_proxy` 模块；`nginx` 的 `proxy` 以及 `upstream` 等。这种方式压测的结果准确性高，唯一的不足是这种方式依赖系统流量，如果系统流量很低，就算是将所有的流量引到一台机器上面，仍不足以达到压测目的。请求引流转发模式的压测结果准确性高。

修改负载均衡权重

同样为完全真实的压测模型，可以用于集群部署的 `web` 环境中，也可用于集群部署的服务类系统。在 `web` 环境中我们通过修改 `F5` 或者 `LVS` 的机器负载均衡权重来使得流量更多的倾斜到其中的某一台或者某几台机器上；对于服务类系统，我们通过修改服务注册中心的机器负载均衡权重来使得服务的调用更多分配到某一台或者某几台机器上。修改负载均衡权重式的压测结果准确性高。

系统的服务能力我们定义为“系统能力”。在系统机器配置都差不多的情况下，系统能力等于线上压力测试获取的单台服务能力乘以机器数。在得知了系统能力之后，接下来我们需要知道我们的系统跑在怎么样的一个容量水位下，从而指导我们做一些决策，是该

加机器了？还是该下掉一些多余的机器？通常系统的调用都有相关日志记录，通过分析系统的日志等方式获取系统一天当中最大的调用频率（以分钟为单位），我们定义为系统负荷；当前一分钟的调用频率我们定义为当前负荷。计算系统负荷可以先把相关日志传到 **hdfs**，通过离线 **hadoop** 任务分析；计算当前负荷我们采用 **storm** 的流式计算框架来进行实时的统计。

水位公式

系统水位 = 系统负荷 / 系统能力；当前水位 = 当前负荷 / 系统能力。

水位标准

单机房（70%）；双机房（40%）；三机房（60%）。

单机房一般都是不太重要的系统，我们可以压榨下性能；

双机房需要保障在一个机房完全挂掉的情况下另一个机房能够撑得住挂掉机房的流量；

三机房同样需要考虑挂掉一个机房的场景后剩下两个机房能够撑得住挂掉机房的流量。

机器公式

理论机器数 = （实际机器数 * 系统负荷 * 系统水位） / （系统能力 * 水位标准）

机器增减 = 理论机器数 - 实际机器数

7.3、稳定性平台双 11 准备与优化

强弱依赖检测面临的最大挑战就是如何使用户使用方便，接入成本最小，主要需要解决下面两件事情：

- 如何复用现有的测试用例？
我们开发一个注解包，里面封装与 **CSP** 的交互协议。服务器端完成测试环境的管理，测试用例端专注应用系统的验证。这是一种测试平台无关的方式，不需要修改现有的测试代码，只需要配置注解的方式就使测试用例支持了强弱依赖验证的功能。
- 如何解决故障模拟组件覆盖不全导致的验证局限？
依赖调用一定存在 **client** 和 **server** 端，很有可能出现一端没有安装故障模拟组件的情况。为此，我们改造了故障描述协议，增加了 **client** 和 **server** 两种模式，只要 **client** 或 **server** 有一方安装了故障模拟组件就可以完成强弱依赖校验。

小结

稳定性平台通过依赖治理、容量规划、降级管理、实时监控等手段，对阿里各系统稳定性的治理给予了支持。未来我们将继续深挖稳定性这个领域，汇总各种数据，真正做到稳定性的智能化、自动化。

阿里中间件与稳定性平台团队的同学是一群不安于现状且喜欢折腾的人，未必很资深但
是很执着，充满热情。大家来自五湖四海，到这里一起解决技术难题，提升系统性能，
完成业务突破，构建新的应用，玩儿转技术、业务、数据、无线。如果你酷爱技术、喜
欢钻研、愿意去帮助多个业务系统的发展，并且认为编程是别人不能剥夺的权利的话，
欢迎加入我们，一起提升我们的技术产品，一起去支持业务更好的发展。你可以到我们的
官方博客 <http://jm.taobao.org> 上了解更多关于我们的信息。

