第三部分

淘宝技术发展3

分布式时代

服务化

在系统发展的过程中，架构师的眼光至关重要，作为程序员，只要把功能实现即可，但作为架构师，要考虑系统的扩展性、重用性，对于这种敏锐的感觉，有人说是一种“代码洁癖”。淘宝早期有几个架构师具备了这种感觉，周悦虹开发的Webx是一个扩展性很强的框架，行癫在这个框架上插入了数据分库路由的模块、Session框架等。在做淘宝后台系统的时候，同样需要这几个模块，行癫指导我把这些模块单独打成JAR包。另外，在做淘宝机票、彩票系统的时候，页面端也有很多东西需要复用，最直观的是页眉和页脚，一开始，我们的每个系统中都复制了一份，但奇妙的是，那段时间页脚要经常修改，例如，把“雅虎中国”改成“中国雅虎”，过一段时间又加了一个“口碑网”，再过一段时间变成了“雅虎口碑”，最后又变成了“中国雅虎”。后来我就把这部分Velocity模板独立出来做成了公用的模块。

上面说的都是比较小的复用模块，到2006年，我们做了一个

商品类目属性的改造，在类目中引入了属性的概念。项目的代号叫做“泰山”，如同它的名字一样，这是一个举足轻重的项目，这个改变是一个划时代的创新。在这之前的三年时间内，商品的分类都是按照树状一级一级的节点来分的，随着商品数量的增长，类目也变得越来越深，且越来越复杂，这样，买家如果要查找一件商品，就要逐级打开类目，找商品之前要弄清商品的分类。而淘宝运营部门管理类目的小二也发现了一个很严重的问题，例如，男装里有T恤、T恤下面有耐克、耐克有纯棉的，女装里也有T恤、T恤下面还是有耐克、耐克下面依然有纯棉的，那是先分男女装，再分款式、品牌和材质呢，还是先分品牌，再分款式、材质和男女装呢？弄得很乱。这时候，一位大侠出来了——一灯，他说品牌、款式、材质等都可以叫做“属性”，属性是类似Tag（标签）的一个概念，与类目相比更加离散、灵活，这样也缩减了类目的深度。这个思想的提出一举解决了分类的难题！从系统的角度来看，我们建立了“属性”这样一个数据结构，由于除了类目的子节点有属性外，父节点也可能有属性，于是类目属性合起来也是一个结构化的数据对象。这个做出来之后，我们把它独立出来作为一个服务，叫做Catserver（Category Server）。跟类目属性密切关联的商品搜索功能独立出来，叫做Hesper（金星）。Catserver和Hesper供淘宝的前后台系统调用。

现在淘宝的商品类目属性已经是全球最大的，几乎没有什么类目的商品在淘宝上找不到（除了违禁的），但最初的类目属性

改造完之后，我们很缺乏属性数据，尤其是数码类。从哪里弄这些数据呢？我们跟“中关村在线”合作，拿到了很多数据，那个时候，很多商品属性信息的后边标注着：“来自中关村在线”。有了类目属性，给运营工作带来了很大的便利，我们知道淘宝的运营主要就是类目的运营，什么季节推什么商品，都要在类目属性上做调整，让买家更容易找到。例如，夏天让用户在女装一级类目下标出材质是不是蕾丝的、是不是纯棉的，冬天却要把羽绒衣调到女装一级类目下，什么流行，就要把什么商品往更高级的类目调整。这样类目和属性要经常调整，随之而来的问题就出现了——调整到哪个类目，所属商品的卖家就要编辑一次自己的商品，随着商品量的增长，卖家的工作量越来越大，他们肯定受不了。到了2008年，我们研究了超市里前后台商品的分类，发现超市前台商品可以随季节和关联来调整摆放场景（例如著名的啤酒和尿布的关联），后台仓库里要按照自然类目来存储，二者密切关联，却又相互分开这样卖家发布商品选择的是自然类目和属性，淘宝前台展示的是根据运营需要摆放商品的类目和属性。改造后的类目属性服务取名为Forest（森林，与类目属性有点神似。Catserver还用于提供卖家授权、品牌服务、关键词等相关的服务）。类目属性的服务化是淘宝在系统服务化方面做的第一个探索。

虽然个别架构师具备了“代码洁癖”，但淘宝前台系统的业

务量和代码量还是呈爆炸式的增长。业务方总在后面催，开发人员不够就继续招人，招来的人根本看不懂原来的业务，只好摸索着在“合适的地方”加一些“合适的代码”，看看运行起来像那么回事后，就发布上线。在这样的恶性循环中，系统越来越臃肿，业务的耦合性越来越高，开发的效率越来越低。借用当时比较流行的一句话“你写一段代码，编译一下能通过，半个小时就过去了；编译一下没通过，半天就过去了。”在这种情况下，系统出错的概率也逐步增长，常常是你改了商品相关的某些代码，发现交易出问题了，甚至你改了论坛上的某些代码，旺旺出问题了。这让开发人员苦不堪言，而业务方还认为开发人员办事不力。

大概是在2007年年底的时候，研发部空降了一位从硅谷来的高管——空闻大师。空闻是一位温厚的长者，他告诉我们一切要以稳定为中心，所有影响系统稳定的因素都要解决掉。例如，每做一个日常修改，都必须对整个系统回归测试一遍；多个日常修改如果放在一个版本中，要是一个功能没有测试通过，整个系统都不能发布。我们把这个叫做“火车模型”，即任何一个乘客没有上车，都不许发车。这样做最直接的后果就是火车一直晚点，新功能上线更慢了，我们能明显感觉到业务方的不满，空闻的压力肯定非常大。

现在回过头来看看，其实我们并没有理解背后的思路。正是

在这种要求下，我们不得不开始改变一些东西，例如，把回归测试日常化，每天晚上都跑一遍整个系统的回归。另外，在这种要求下，我们不得不对这个超级复杂的系统做肢解和重构，其中复用性最高的一个模块——用户信息模块开始拆分出来，我们叫它UIC（User Information Center）。在UIC中，它只处理最基础的用户信息操作，例如，getUserById、getUserByName等。

在另一方面，还有两个新兴的业务对系统基础功能的拆分也提出了要求。在那个时候，我们做了淘宝旅行（trip.taobao.com）和淘宝彩票（caipiao.taobao.com）两个新业务，这两个新业务在商品的展示和交易的流程上都跟主站的业务不一样，机票是按照航班的信息展示的，彩票是按照双色球、数字和足球的赛程来展示的。但用到的会员功能和交易功能是与主站差不多的，当时做起来就很纠结，因为如果在主站中做，会有一大半跟主站无关的东西，如果重新做一个，会有很多重复建设。最终我们决定不再给主站添乱了，就另起炉灶做了两个新的业务系统，从查询商品、购买商品、评价反馈、查看订单这一整个流程都重新写了一套。现在在“我的淘宝”中查看交易记录的时候，还能发现“已买到的宝贝”中把机票和彩票另外列出来了，他们没有加入到普通的订单中。在当时，如果已经把会员、交易、商品、评价这些模块拆分出来，就不用什么都重做一遍了。

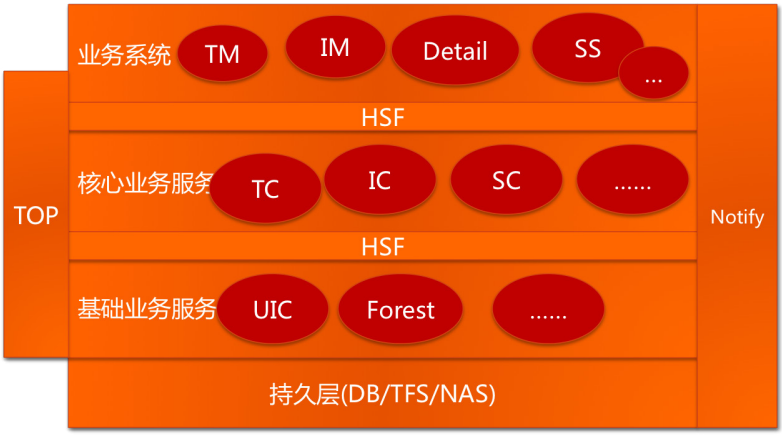
到2008年初，整个主站系统（有了机票、彩票系统之后，把原来的系统叫做主站）的容量已经到了瓶颈，商品数在1亿个以上，PV在2.5亿个以上，会员数超过了5000万个。这时Oracle的连接池数量都不够用了，数据库的容量到了极限，即使上层系统加机器也无法继续扩容，我们只有把底层的基础服务继续拆分，从底层开始扩容，上层才能扩展，这才能容纳以后三五年的增长。

于是我们专门启动了一个更大的项目，即把交易这个核心业务模块拆分出来。原来的淘宝交易除了跟商品管理耦合在一起，还在支付宝和淘宝之间转换，跟支付宝耦合在一起，这会导致系统很复杂，用户体验也很不好。我们把交易的底层业务拆分出

来，叫交易中心（Trade Center，TC），所谓底层业务，就如创建订单、减库存、修改订单状态等原子型的操作；交易的上层业务叫交易管理（Trade Manager，TM），例如，拍下一件普通商品要对订单、库存、物流进行操作，拍下虚拟商品不需要对物流进行操作，这些在TM中完成。这个项目取了一个很没有创意的名字——“千岛湖”，开发人员取这个名字的目的是想在开发完毕之后，去千岛湖玩一圈，后来他们如愿以偿了。这个时候还有一个淘宝商城的项目在做，之前拆分出来的那些基础服务给商城的快速构建提供了良好的基础。

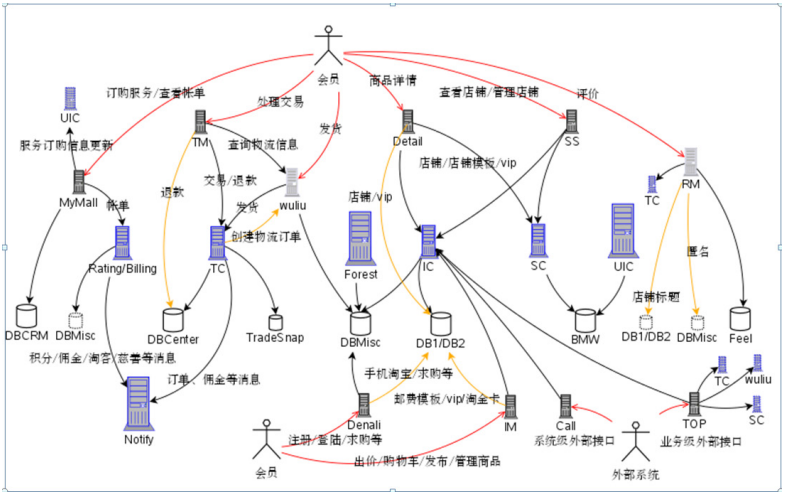
类目属性、用户中心、交易中心，随着这些模块的逐步拆分和服务化改造，我们在系统架构方面也积累了不少经验。到2008年年底就做了一个更大的项目，把淘宝所有的业务都模块化，这是继2004年从LAMP架构到Java架构之后的第二次脱胎换骨。我们对这个项目取了一个很霸气的名字——“五彩石”（女娲炼石补天用的石头）。这个系统重构的工作非常惊险，有人称为“给一架高速飞行的飞机换发动机”。“五彩石”项目发布之后，相关工程师去海南三亚玩了几天。

他们把淘宝的系统拆分成了如下架构。



其中，UIC和Forest在上文已说过，TC、IC、SC分别是交易中心（Trade Center）、商品中心（Item Center）、店铺中心（Shop Center），这些中心级别的服务只提供原子级的业务逻辑，如根据ID查找商品、创建交易、减少库存等操作。再往上一层是业务系统TM（Trade Manager，交易业务）、IM（Item Manager，商品业务）、SM（Shop Manager，后来改名叫SS，即Shop System，店铺业务）、Detail（商品详情）。

拆分之后，系统之间的交互关系变得非常复杂，示意图如下所示。



系统这么拆分的好处显而易见，拆分之后的每个系统可以单独部署，业务简单，方便扩容；有大量可重用的模块便于开发新的业务；能够做到专人专事，让技术人员更加专注于某一个领域。这样要解决的问题也很明显，分拆之后，系统之间还是必须要打交道的，越往底层的系统，调用它的客户越多，这就要求底层的系统必须具有超大规模的容量和非常高的可用性。另外，拆分之后的系统如何通信？这里需要两种中间件系统，一种是实时调用的中间件（淘宝的HSF，高性能服务框架），一种是异步消息通知的中间件（淘宝的Notify）。另外，一个需要解决的问题是用户在A系统登录后，到B系统的时候，用户的登录信息怎么保存？这又涉及一个Session框架。再者，还有一个软件工程方面的问题，这么多层的一套系统，怎么去测试它？

中间件

互联网系统的发展看似非常专业，其实在生活中也存在类似的“系统”，正如一位哲学家说“太阳底下无新事”。我们可以从生活中的一个小例子来看网站系统的发展，这个例子是HSF的作者毕玄写的。

一家小超市，一个收银员，同时还兼着干点其他的事情，例如，打扫卫生、摆货。

来买东西的人多起来了，排队很长，顾客受不了，于是增加了一个收银台，雇了一个收银员。

忙的时候收银员根本没时间去打扫卫生，超市内有点脏，于是雇了一个专门打扫卫生的。

随着顾客不断增加，超市也经过好几次装修，由以前的一层变成了两层，这个时候所做的事情就是不断增加收银台、收银员和打扫卫生的人。

在超市运转的过程中，老板发现一个现象，有些收银台排很长的队，有些收银台排的人不多，了解后知道是因为收银台太多了，顾客根本看不到现在各个收银台的状况。对于这个现象，一种简单的方法就是继续加收银台。但一方面，超市没地方可加收

银台了，另一方面，作为老板，当然不需要雇太多的人，于是开始研究怎样让顾客了解到收银台的状况，简单地加了一个摄像头和一个大屏幕，在大屏幕上显示目前收银台的状况，这样基本解决了这个问题。

排队长度差不多后，又出现了一个现象，就是有些收银台速度明显比其他的慢，原因是排在这些收银台的顾客买的东西特别多，于是又想了一招，就是设立专门的10件以下的通道，这样买东西比较少的顾客就不用排太长的队了，这一招施展后，顾客的满意度明显提升，销售额也好了不少，后来就继续用这招应对团购状况、VIP 状况。

在解决了上面的一些烦心事后，老板关注到了一个存在已久的现象，就是白天收银台很闲，晚上则很忙，于是从节省成本上考虑，决定实行部分员工只在晚上上班的机制，白天则关闭一些收银台，顾客仍然可以通过大屏幕看到哪些收银台是关闭的，避免走到没人的收银台去，实行这招后，成本大大降低了。

这个生活中的例子及其解决的方法，其实和互联网网站发展过程中的一些技术是非常类似的，只是在技术层面用其他名词来表达了而已，例如，有集群、分工、负载均衡、根据QoS分配资源等。

- 集群：所有的收银员提供的都是收银功能，无论顾客到哪一个收银员面前，都可完成付款，可以认为所有的收银员就构成了一个集群，都希望能做到顾客增加的时候只需增加收银员就行。在现实生活中有场地的限制，而在互联网应用中，能否集群化还受限于应用在水平伸缩上的支撑程度，而集群的规模通常会受限于调度、数据库、机房等。

- 分工：收银员和打扫卫生的人分开，这种分工容易解决，而这种分工在互联网中是一项重要而复杂的技术，没有现实生活中这么简单，涉及的主要有按功能和数据库的不同拆分系统等，如何拆分以及拆分后如何交互是需要面临的两个挑战。因此，会有高性能通信框架、SOA平台、消息中间件、分布式数据层等基础产品的诞生。

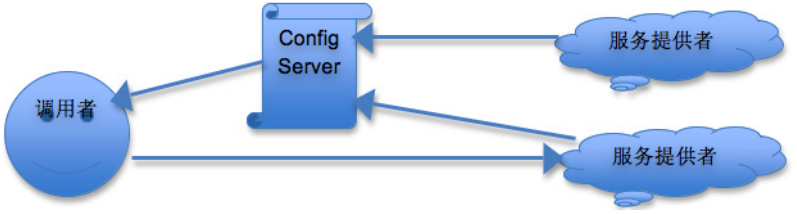
- 负载均衡：让每个收银台排队差不多长，设立小件通道、团购通道、VIP通道等，这些可以认为都是集群带来的负载均衡的问题，从技术层面上说，实现起来自然比生活中复杂很多。

- 根据QoS分配资源：部分员工仅在晚上上班的机制要在现实生活中做到不难，而对互联网应用而言，就是一件复杂而且极具挑战的事。

参照生活中的例子来说，在面对用户增长的情况下，想出这

些招应该不难，不过要掌握以上四点涉及的技术就相当复杂了，而且互联网中涉及的其他很多技术还没在这个例子中展现出来，例如缓存、CDN等优化手段；运转状况监测、功能降级、资源劣化、流控等可用性手段，自建机房、硬件组装等成本控制手段。因此，构建一个互联网网站确实是不容易的，技术含量十足，当然，经营一家超市也不简单。

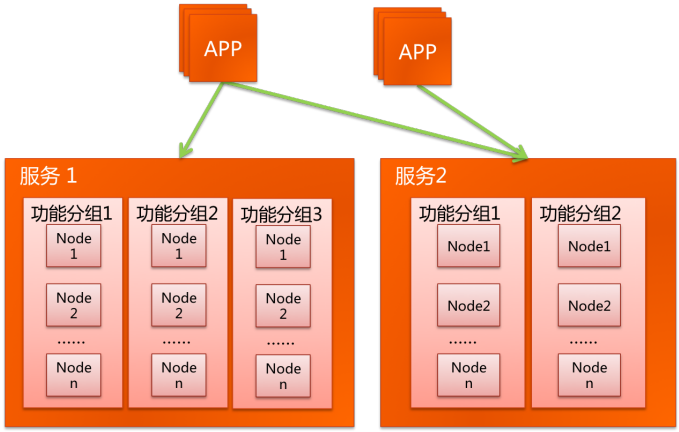
从超市的运维可以抽象出系统设计的一些思路，服务拆分之后，如何取得我需要的服务？在“电视机”上，把每个集群能提供的服务显示出来。你不需要关心哪个人为你服务，当你有需要的时候，请先看头顶的电视机，它告诉你哪个服务在哪个区域。当你直接去这个区域的时候，系统会给你找到一个最快速的服务通道。



这就是HSF的设计思想，服务的提供者启动时通过HSF框架向ConfigServer（类似超市的电视机）注册服务信息（接口、版本、超时时间、序列化方式等），这样ConfigServer上面就定义了所有可供调用的服务（同一个服务也可能有不同的版本）；服

务调用者启动的时候向ConfigServer注册对哪些服务感兴趣（接口、版本），当服务提供者的信息变化时，ConfigServer向相应的感兴趣的服务调用者推送新的服务信息列表；调用者在调用时则根据服务信息的列表直接访问相应的服务提供者，而无须经过ConfigServer。我们注意到ConfigServer并不会把服务提供者的IP地址推送给服务的调用者，HSF框架会根据负载状况来选择具体的服务器，返回结果给调用者，这不仅统一了服务调用的方式，也实现了“软负载均衡”。平时ConfigServer通过和服务提供者的心跳来感应服务提供者的存活状态。

在HSF的支持下，服务集群对调用者来说是“统一”的，服务之间是“隔离”的，这保证了服务的扩展性和应用的统一性。再加上HSF本身能提供的“软负载均衡”，服务层对应用层来说就是一片“私有云”了。

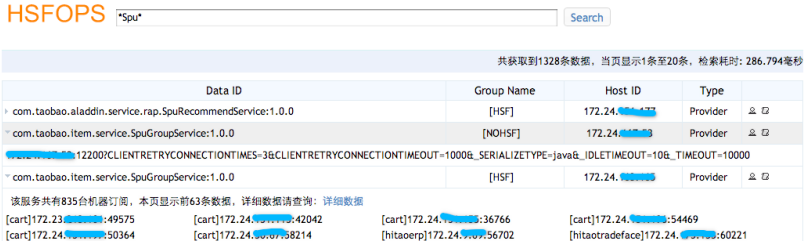


HSF框架以SAR包的方式部署到Jboss、Jetty或Tomcat下，在应用启动的时候，HSF（High-Speed Service Framework，在开发团队内部有一些人称HSF为“好舒服”）服务随之启动。HSF旨在为淘宝的应用提供一个分布式的服务框架，HSF从分布式应用层面以及统一的发布/调用方式层面为大家提供支持，从而可以很容易地开发分布式的应用以及提供或使用公用功能模块，而不用考虑分布式领域中的各种细节技术，例如，远程通讯、性能损耗、调用的透明化、同步/异步调用方式的实现等问题。

从上图HSF的标志来看，它的速度是很快的。HSF是一个分布式的标准Service方式的RPC（Remote Procedure Call Protocol，远程过程调用协议）框架，Service的定义基于OSGI的方式，通讯层采用TCP/IP协议。关于分布式的服务框架的理论基础，HSF的作者毕玄写了一篇博文（http://www.blogjava.net/BlueDavy/archive/2008/01/24/177533.html），有关基于OSGI的分布式服务框架，也有一系列的博文（http://www.blogjava.net/BlueDavy/archive/2008/01/14/175054.html）。

从下面这个HSF监控系统的截图中可以更直观地看到一些信

息，在两个集群中有两个服务器（其实有更多的，没有全部截图下来）都提供com.taobao.item.service.SpuGroupService 这一服务，版本号都是1.0.0，这个服务在ConfigServer上的注册信息中包含超时时间、序列化方式。在后面那条信息中可看到，在展开的这个集群中服务有835台机器已订阅，这些订阅者有淘宝的服务器（cart是购物车功能的服务器），也有hitao（淘花网）的服务器。



HSF系统目前每天承担了300亿次以上的服务调用。

一些读者可能会有一个疑问：既然淘宝的服务化是渐进式的，那么在HSF出现之前，系统之间的调用采用什么方式呢?

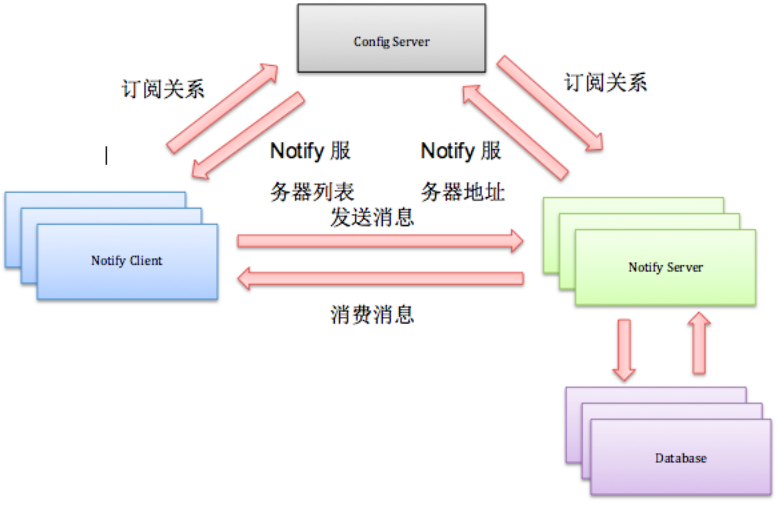
这个有点“五花八门”，例如，对于类目的调用方式是：Forest打包成一个JAR包，在应用启动的时候装载到内存中，仅这一个JAR包所占用的内存就有800MB之多（因为淘宝的类目数据太庞大了），对于当时一般只有2GB内存的开发机来说，加载完类目信息后，机器运行速度就非常慢。对于用户信息（UIC）

来说，一开始的调用方式是用Hessian接口。还有一些系统是通过WebService、Socket甚至是HTTP请求来相互调用的。每种调用方式都涉及各种超时、信息的加解/密、参数的定义等问题，由此可见，在没有HSF之前，系统之间的调用是错综复杂的。而随着系统拆分得越来越多，必须由一个统一的中间层来处理这种问题，HSF正是在这种背景下诞生的。

Notify

HSF解决了服务调用的问题，我们再提出一个很早就说过的问题：用户在银行的网关付钱后，银行需要通知到支付宝，但银行的系统不一定能发出通知；如果通知发出了，不一定能通知到；如果通知到了，不一定不重复通知一遍。这个状况在支付宝持续了很长时间，非常痛苦。支付宝从淘宝剥离出来的时候，淘宝和支付宝之间的通信也面临同样的问题，那是2005年的事情，支付宝的架构师鲁肃提出用MQ（Message Queue）的方式来解决这个问题，我负责淘宝这边读取消息的模块。但我们发现消息数量上来之后，常常造成拥堵，消息的顺序也会出错，在系统挂掉的时候，消息也会丢掉，这样非常不保险。然后鲁肃提出做一个系统框架上的解决方案，把要发出的通知存放到数据库中，如果实时发送失败，再用一个时间程序来周期性地发送这些通知，系统记录下消息的中间状态和时间戳，这样保证消息一定能发出，也一定能通知到，且通知带有时间顺序，这些通知甚至可以实现事务性的操作。

在“千岛湖”项目和“五彩石”项目之后，淘宝自家的系统也拆成了很多个，他们之间也需要类似的通知。例如，拍下一件商品，在交易管理系统中完成时，它需要通知商品管理系统减少库存，通知旺旺服务系统发送旺旺提醒，通知物流系统上门取货，通知SNS系统分享订单，通知公安局的系统这是骗子……用户的一次请求，在底层系统可能产生10次的消息通知。这一大堆的通知信息是异步调用的（如果同步，系统耦合在一起就达不到拆分的目的），这些消息通知需要一个强大的系统提供支持，从消息的数量级上看，比支付宝和淘宝之间的消息量又上了一个层次，于是按照类似的思路，一个更加强大的消息中间件系统就诞生了，它的名字叫做Notify。Notify是一个分布式的消息中间件系统，支持消息的订阅、发送和消费，其架构图如下所示。

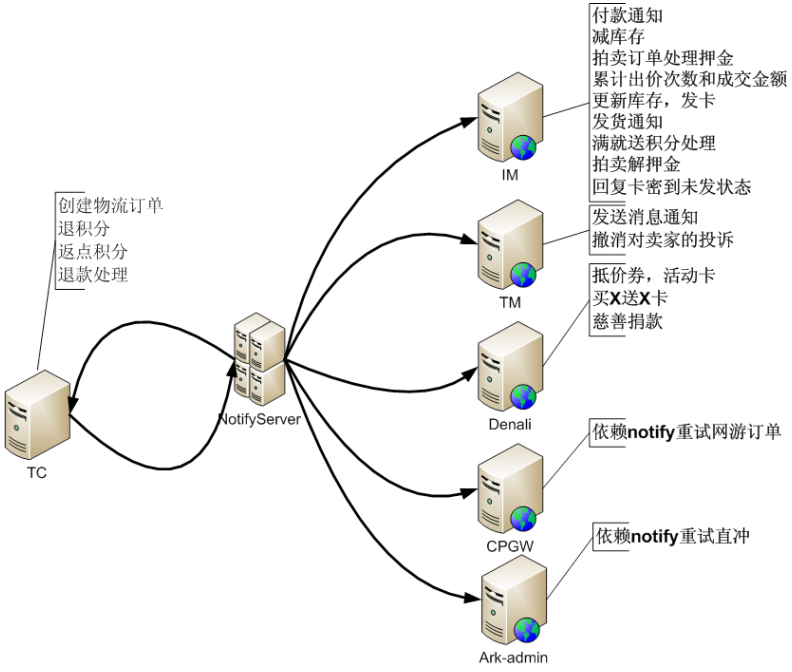


NotifyServer在ConfigServer上面注册消息服务，消息的客户端通过ConfigServer订阅消息服务。某个客户端调用NotifyServer发送一条消息，NotifyServer负责把消息发送到所有订阅这个消息的客户端（这个过程参照HSF一节，原理是一样的）。为了保证消息一定能发出，且对方也一定能收到，消息数据本身就需要记录下来，这些信息存放在数据库中（可以是各种数据库）。由于消息具有中间状态（已发送、未发送等），应用系统通过Notify可以实现分布式事物——BASE（基本可用（Basically Available）、软状态 （Soft State）、最终一致（Eventually Consistent））。NotifyServer可以水平扩展，NotifyClient也可以水平扩展，数据库也可以水平扩展，从理论上讲，这个消息系统的吞吐量是没有上限的，现在Notify系统每天承载了淘宝10亿次以上的消息通知。

下图展示了创建一笔交易之后，TC（交易中心）向Notify发送一条消息，后续Notify所完成的一系列消息通知。

TDDL

有了HSF和Notify的支持，在应用级别中，整个淘宝网的系统可以拆分了，还有一个制约系统规模的更重要的因素，就是数据库，也必须拆分。



在第二部分中讲过，淘宝很早就对数据进行过分库的处理，上层系统连接多个数据库，中间有一个叫做DBRoute的路由来对数据进行统一访问。DBRoute对数据进行多库的操作、数据的整合，让上层系统像操作一个数据库一样操作多个库。但是随着数据量的增长，对于库表的分法有了更高的要求，例如，你的商品数据到了百亿级别的时候，任何一个库都无法存放了，于是分成2个、4个、8个、16个、32个……直到1024个、2048个。好，分成这么多，数据能够存放了，那怎么查询它？这时候，数据查询的中间件就要能够承担这个重任了，它对上层来说，必须像查询一

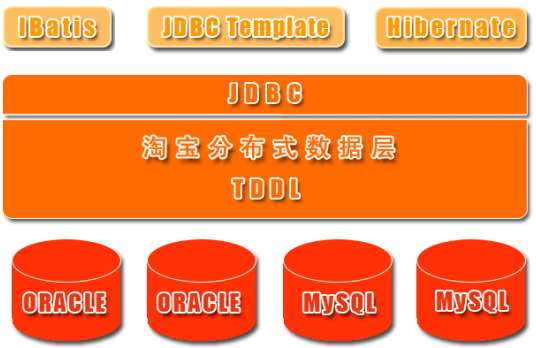
个数据库一样来查询数据，还要像查询一个数据库一样快（每条查询在几毫秒内完成），TDDL就承担了这样一个工作。

另外，加上数据的备份、复制、主备切换等功能，这一套系统都在TDDL中完成。在外面有些系统也用DAL（数据访问层）这个概念来命名这个中间件。

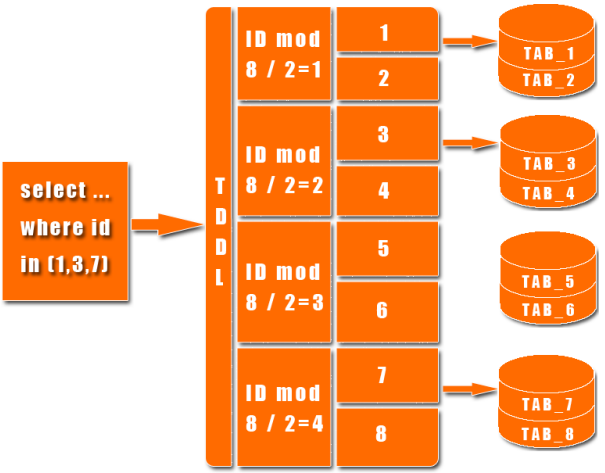
TDDL实现了下面三个主要的特性：

- 数据访问路由——将针对数据的读写请求发送到最合适的地方；
- 数据的多向非对称复制——一次写入，多点读取；
- 数据存储的自由扩展——不再受限于单台机器的容量瓶颈与速度瓶颈，平滑迁移。

下图展示了TDDL所处的位置。



下图展示了一个简单的分库分表数据查询策略。



下面是TDDL的主要开发者之一沈询描述的“TDDL的前世今生”——数据层的发展历程。

CommonDAO的时代

数据切分并不算是一个很新的概念，当商品库切分为两个时，就已经出现了名字叫做xingdian（笑，那时候行癫已经不写代码了，但从代码的版本信息可以看到作者）的人写的CommonDAO。

CommonDAO的思路非常简单实用，因为淘宝主要在使用ibatis作为访问数据库的DAO层，所以，CommonDAO的作用就是对ibatis层做了一个很浅的封装，允许你通过商品字串ID的第一个

字符来访问两台数据库中的一台。

比如，如果字符串ID的第一个字符是0~7，那么走到数据库1去，如果是8~f，则走到数据库2去。同时，也允许用户直接给定数据库的名字来访问数据库。

这应该是最早的数据层原型。

TDDL 1.0时代

后来，大家逐渐发现，如果按照业务的发展规模和速度，那么使用高端存储和小型机的Oracle存储的成本将难以控制，于是降低成本就成了必然。

如何能够在不影响业务正常发展的前提下，从一定程度上解决成本的问题呢？

“对一部分数据库使用MySQL”，DBA们的决策是这样，于是，分布式数据层的重担就落到了华黎的头上。

别看现在数据水平切分似乎已经成了基础知识。在2007年、2008年，如何设计它还真是让我们伤透了脑筋。

当时的我们，只知道eBay有一个数据层，却不知道如何设计和实现？

于是邀请了当时所有的业务负责人来畅想数据层的样子……

得到了以下需求：

- 对外统一一切数据访问；
- 支持缓存、文件存储系统；
- 能够在Oracle和MySQL之间自由切换；
- 支持搜索引擎。

然后，我们自己的问题与现在大家所问的问题也是完全一样的。

如何实现分布式Join（连接）？——在跨节点以后，简单的Join会变成M×N台机器的合并，这个代价比原来的基于数据库的单机Join大太多了。

如何实现高速多维度查询？——就像SNS中的消息系统，A发给B一个消息，那么A要看到的是我发给所有人的消息，而B要看到的是所有人发给我的消息。这种多维度查询，如何能够做到高效快捷呢？

如何实现分布式事务？——原始单机数据库中存在着大量的事务操作，在分布式以后，分布式事务的代价远远大于单机事务，那么这个矛盾也变得非常明显。

华黎带着我和念冰，坐在那里讨论了一个半月，还是没想出

来……于是决定先动起手来。名字是我起的——Taobao Distributed Data layer（TDDL，后来有人对它取了个外号：“头都大了”⊙﹏⊙b）

学习开源的Amoeba Proxy。

找到的目标应用是“收藏夹”，首先要做的两个关键的特性是：分库分表和异构数据库的数据复制。

开始本来希望和B2B的团队合作，因为我们觉得独立的Proxy没有太大必要。而SQL解析器因为有淘宝特殊的需求，所以也需要重写。

可惜，最后因为B2B的人搬到滨江去了，交流十分不畅，所以最后只是做了拿来主义，没有对开源的Amoeba和当时的Cobar有所贡献。

回到淘宝，因为有东西可以借鉴，我们在一个多月的时间内就完成了TDDL 1.0版本的工作。上线过程中虽然出了点小问题，不过总体来说是比较成功的。

TDDL 2.0时代

随着使用TDDL的业务越来越多，对业务方来说，DBA 对于使用MySQL以及数据切分也积累了比较多的经验，于是决定开始动核心应用了。

“评价”是第一个重要的应用，评价最重要的问题还是在于双向查询、评价、被评价。于是我们的异构数据源增量复制就派上了用场。

然后是“商品”，我们在商品上投入了近半年的时间，失败很多，也成长得最快。

- 容量规划做得不到位，机器到位后因压力过大，直接死掉，于是产生了数据库容量线上压力模拟测试。
- 历史遗留问题，商品几乎是所有的业务都会使用的资源，所以接口设计比较复杂。很多接口的调用在新架构上很难以低成本的方式实现。而推动业务改动，则需要大量的时间和成本。
- 数据层代码被业务代码侵染，看起来似乎应该是数据层的代码，但实际上又只有商品在使用。这种问题让数据层的依赖变得更加庞大，边缘代码变得更多，冲突更明显。

TDDL 3.0～TDDL 4.0时代

在商品之后，似乎所有的应用都可以使用类似的方式来解决业务增长上量的问题。但正当我们志得意满的时候，却被“交易”撞了一个满怀。

我一直很感谢交易线的所有同仁，他们是淘宝草根精神的典

型代表——功能可以做得不那么“漂亮”，但必须减少中间环节，真正做到了实用、干净、简洁。我们在向他们介绍产品的时候，他们对我们的实现细节提出了非常多的质疑，他们认为整个流程中只有规则、主备切换对他们是有意义的，而解析、合并则是他们所不需要的功能。

“不需要的功能为什么要放到流程里？增加的复杂度会导致更多的问题”。在当时，我感到很痛苦，因为我无法回答他们这些质疑之声。

不过，也正是因为这些质疑，让我有了一个契机，重新审视自己所创造出来的产品。

我问自己：它能够给业务带来什么益处？

对此，我的回答是：

- 规则引擎/切分规则可以用配置的方式帮助业务隔离具体的数据库地址与用户的业务逻辑；
- 单机主备切换；
- 数据源简化和管理。

于是，我们就产生了TDDL 3.0版本。其主要的作用就是将代码做了逻辑切分，将单机主备切换和数据源管理独立了出来。这

样，可以针对不同的业务需求，给予不同的逻辑分层。让每一个业务都有适合自己的使用数据库的方式。

同时，我们开始做工具，Rtools/JADE 作为数据库运维平台的组件被提了出来。在它的帮助下，我们发现能够极大地提升用户在使用单机数据源和多机数据源时的效率。用户只需要在客户端给定两个属性，就可以立刻开始使用。结果是用户反馈比以前好了很多。

这也坚定了我们开发工具的决心。

工具平台时代

在尝到工具平台的甜头以后，我们在工具组件上走得更远了。

首先被提出的是“愚公”数据迁移平台。该平台能够在多种异构的数据库中进行数据的平滑移动，对业务影响很小，并且也允许业务插入自己的业务逻辑。

这个东西主要能够帮助业务进行数据库自动扩容，自动缩容，单机、多机数据迁移，在Oracle到MySQL数据迁移等场景中都发挥了重要的作用。

然后，又以内部开源的方式提出了“精卫”数据增量复制平台。这个平台基于数据库的通用数据分发组件，基于开源的

Tungsten进行了大量Bug Fix和结构调优。在数据的一对多分发以及异步通知给DW和搜索等场景中都发挥了重要的作用。

TDDL的现在

粗略统计下来，TDDL已经走过了4年的时间，满足了近700个业务应用的使用需求。其中有交易商品评价用户等核心数据，也有不那么有名的中小型应用。量变产生质变，如何能够更好地帮助这些业务以更低的成本更快地完成业务需求，将成为数据层未来最重要的挑战。

Session框架

介绍Session框架之前，有必要先了解一下Session。Session在网络应用中称为“会话”，借助它可提供客户端与服务系统之间必要的交互。因为HTTP协议本身是无状态的，所以经常需要通过Session来解决服务端和浏览器的保持状态的解决方案。用户向服务器发送第一个请求时，服务器为其建立一个Session，并为此Session创建一个标识，用户随后的所有请求都应包括这个标识号。服务器会校对这个标识号以判断请求属于哪个Session。会话保持有效，默认状况下，直到浏览器关闭，会话才结束。

Session中存储的内容包括用户信息：昵称、用户ID、登录状态等。

当网站服务器只有一台的时候，用Session来解决用户识别是很简单的，但是当网站是一个集群的时候，同一用户的两次请求可能被分配到两台不同的服务器上处理。怎样保证两次请求中存取的Session值一致呢？还有一个问题：网站规模扩大时，对于一个具有上亿个访问用户的系统来说，当大部分用户的Session信息都存储在服务端时，要在服务端检索出用户的信息效率就非常低了，Session管理器不管用什么数据结构和算法都要耗费大量内存和CPU时间。如何解决服务端Session信息的管理？

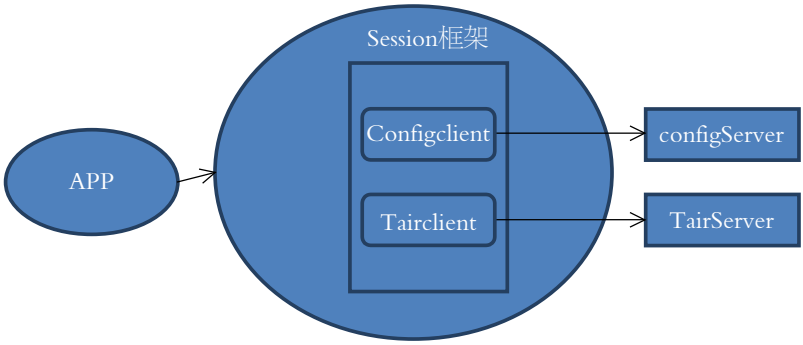
解决集群Session共享的问题，通常有以下两种办法。

- 硬件负载，将用户请求分发到特定的服务器。
- Session复制，就是将用户的Session复制到集群内所有的服务器。

这两种方法的弊端也很明显：

- 成本较高。
- 性能差。当访问量增大的时候，带宽增大，而且随着机器数量的增加，网络负担成指数级上升，不具备高度可扩展性，性能随着服务器数量的增加急剧下降，而且容易引起广播风暴。

这种情况下，Tbsession框架闪亮登场了。Tbsession框架致力于解决以下几个问题。



- Session的客户端存储，将Session信息存储到客户端浏览器Cookie中。
- 实现服务端存储，减少Cookie使用，增强用户信息的安全性，避免浏览器对Cookie数量和大小的限制。
- Session配置统一管理起来，集中管理服务端Session和客户端Cookie的使用情况，对Cookie的使用做有效的监管。
- 支持动态更新，Session的配置动态更新。

简单地说，就是要么用客户端Cookie来解决问题，要不用服务端的集中缓存区（Tair）的Session来解决登录问题。Tair已在前文介绍过，Session对它的使用就不再描述了。

为什么这里还要提到用Cookie这种比较“落伍”的方式呢？其实是因为在淘宝3.0版本以前，我们一直都用Cookie来识别用户，Cookie是放在客户端的，每一次HTTP请求都要提交到服务端，在访问量比较小的时候，采用Cookie避免了Session复制、硬件负载等高成本的情况。但随着用户访问规模的提高，我们可以折算一下，一个Cookie大概是2KB的数据，也就是说，一次请求要提交到服务器的数据是网页请求数据，再加上2KB的Cookie，当有上亿个请求的时候，Cookie所带来的流量已经非常可观了，而且网络流量的成本也越来越高。于是在3.0版本中，我们采用了集中式的缓存区的Session方式。

到此为止，应用服务切分了（TM、IM）、核心服务切分了（TC、IC）、基础服务切分了（UIC、Forest）、数据存储切分了（DB、TFS、Tair），通过高性能服务框架（HSF）、分布式数据层（TDDL）、消息中间件（Notify）和Session框架支持了这些切分。一个美好的时代到来了，高度稳定、可扩展、低成本、快速迭代、产品化管理，淘宝的3.0系统走上了历史的舞台。

在这个分布式系统的支持下，更多的业务迅速开发出来了，因为任何一个业务都基于淘宝的商品、交易、会员、评价等基础体系，而这些基础体系就像“云”一样存在，现在可以随处调用了。Hitao、淘花网、良无限、天猫、一淘、聚划算、各种SNS、各种移动客户端等如雨后春笋般地成长起来了。目前，淘宝已经

变成了一个生态体系，包含C2C、B2C、导购、团购、社区等各种电子商务相关的业务。

既然说是一种“生态体系”，那就不能把所有的业务把控在自己的手中，在开发3.0版本的过程中，我们就有一个团队把淘宝“开放”出去了，我们把自己的数据、自己的应用通过接口的方式让更多的开发者调用，他们可以开发出形形色色的产品，例如，你可以开发出心形的淘宝店铺、菱形的店铺，再放到淘宝上供商家购买。淘宝再多的员工，其创造力也是有限的，而开放出去之后，让无限的人都可以参与到这个生态体系的建设中来，这个生态体系才是完整的。下面就是开放平台的架构师放翁所记述的“开放平台这几年”[注1]。

开放平台

2006年年底：阿里巴巴提出了Workatalibaba的战略，二十多个人就被拉到湖畔花园马云的公寓里开始一个叫阿里软件的公司创业。当时对于Work at alibaba有一个朦朦胧胧的感觉，就是要为中小企业提供一个工作平台，但是工作平台又需要是一个开放的平台，因为卖家的需求是长尾的，当时火热的Salesforce给了阿里人

注1　作者放翁在文中涉及的所有技术点都可以在http://blog.csdn.net/cenwenchu79中找到详细的内容，同时本文主要介绍开放平台技术发展历程，产品和业务内容不涵盖在此，因此，受众群体主要是技术人员。——作者注

一些启示，那就是做一个支持二次开发的工作平台，半开放式地满足各种卖家的长尾管理需求。此时，软件市场上就开始培养起最早的一批TP（淘宝开放合作伙伴）。迄今为止，很多非常成功的TP就是从那个时候开始进入淘宝卖家市场的。

但经过一年的平台建设，发现开发者非常难利用平台做二次开发，只有阿里软件公司内部的团队构建了三个不同的CRM软件。这时候淘宝来了一个业界的技术牛人王文彬（花名：菲青），这位淘宝新晋的首席架构师找到阿里软件的平台架构团队，谈到了当时业界还非常新颖的一种技术平台——开放平台。由于阿里软件已经在做类似的开放工作，希望能够以合作的方式来试水开放平台。当时双方都是一种尝试的态度，因此，最后敲定投入一个人花两周时间，看是否能够做出原型，如果可以，就继续做，如果出不了原型，就此结束。两周时间里，负责阿里软件的架构师放翁参看着美国雅虎的开放模式就搞出了开放平台的第一个雏形，没想到就这样开启了5年的开放之路。后面会根据时间轴来说一下开放平台的产品和技术的变革，每一年会发生很多事情，但是调出的一点一滴是当年最有感触的。

2007年：萌芽。SOA盛行的年代，内部架构服务化成为开放的第一步，内部服务不做好隔离，开放就意味着风险不可控。支付宝今天的服务框架SOFA（类ESB）、淘宝的HSF（OSGI）、阿里软件的ASF（SCA）都是那个年代的产物，但服务化带来的痛却是一样的，不论是OSGI还是SCA之类的服务框架，本身的服务

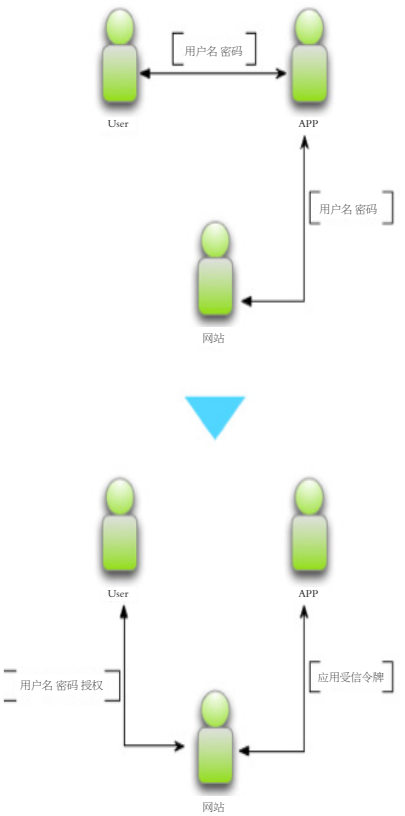
化规约设计都类似，但难题也都摆在每个架构师和开发者面前：服务单元Bundle的粒度控制，服务之间依赖管理，性能与规范的冲突，调试与隔离的平衡。这些都使得一线开发者和平台框架实现者出现非常多的矛盾，而这个过程能活下来的框架最后都是摒弃了很多企业级的设计思路，因为SOA架构从企业级产品演变而来，而服务化后的内部平台要面对的开放平台天生就是互联网的产物。

2008年：雏形。2008年年底，平台开放淘宝服务30个，每天调用量2000次，这一年开放平台的开发者面向的客户主要是阿里巴巴上的中小企业和淘宝C店卖家。开放平台建设初期要解决的就是三个问题：

- 服务路由。（外部可以获取内部信息）
- 服务接口标准化。（统一方式的获得各种标准化信息）
- 授权。（外部合法的获取内部信息）

服务路由其实就是写一个高效的HttpAgent，服务接口标准化就是对象文本化（JSON，XML）。今天在各大开放平台广为使用的OAuth协议，当前处于0.6版本，没有任何实际的互联网开放平台使用，直到Google于2008年年底慢慢地对外推广开放的时候，OAuth被封装到Google的Open SDK中，才使得很多中小型互联网公司使用这种看似复杂的两阶段授权交互模式。淘宝初期采

用的是自有协议，因为OAuth2以前的逻辑较复杂且使用不方便，直到2011年才开始支持OAuth2，同时做了部分的安全增强。授权解决了开放最大的一个问题：用户安全的对应用访问其数据受信。用户从此不用赤裸裸地将用户名和密码交给一个应用软件，应用也可以在允许的范围内（操作、数据、授权时长）充分利用用户授权来玩转创意。



有了上面的三板斧（路由、数据规范和授权），开放平台正式开门迎客了，没有对外做任何的推广，日均调用数据就猛增到了1000次，此时两个互联网的新兴技术Memcached和Hadoop开始在开放平台中尝试。今天看来，这两个技术已经被大规模地使用，Memcached无疑是最好的选择，但当时号称分布式缓存的Memcached其实是集中式缓存的一种，真正的分布式缓存还都在纠结于一致性和效率的问题（第2、3阶段提交）。此时需要有一种方式能够保证效率（可扩展）和稳定性，于是我们封装了Memcached客户端，提升当时BIO的Java客户端的性能，同时引入了客户端负载均衡和容灾的设计，这种设计已经被应用在现在很多大型分布式系统中。另一方面，每天上千万的访问也让技术和产品对访问的行为有很强的分析需求，此时，Hadoop在雅虎的充分利用引起了我们的重视（当时的雅虎技术创新一直都是业界的领头人），通过仅有的两台机器和一堆技术文档，我们摸索着搭建了公司内部的第一个Hadoop集群，而所写的Hadoop入门实践也成为当时Hadoop入门的基础文档，对于每天2000次调用量的日志分析需求来说，Hadoop用得游刃有余，但随着业务的不断发展，Hadoop离线分析所带来的问题也凸显出来，MR程序面对灵活多变的分析需求，显得不易维护且效率低下（数据反复读取分析），于是我们也开始思考怎样改进这个新玩意儿。

2009年：产品化。到2009年年底，平台开放淘宝服务100多个，每天调用量为4000次，这一年开放平台的开发者面对的主要

是淘宝C店卖家，卖家工具成为服务市场的主流。这一年也是变化的一年，阿里软件年中的分拆使得开放平台的归属有些微妙，一种情况是留在阿里云，作为集团的基础设施，另一种情况就是跟着主要的业务需求方淘宝走，最后我们还是说服了博士，结束了阿里软件的老平台，淘宝正式开始自己的开放之路。来到淘宝后，业务开放迅猛增长，从30个API猛增到了100个API，没有对外做任何业务推广，平台调用量到了年底翻番。此时技术上的挑战又聚焦到了性能上，一次API Call的业务消耗平均在30～40ms，开放平台当时的平台处理消耗平均在10ms左右。我们做了数据打点和分析，发现最大的消耗在于互联网数据的接收，同时大量的图片数据上行，更是加大了平台处理时间。另外，从访问日志分析中可以看到很多无效的请求也占用了非常多的处理时间，这也意味着无效请求和有效请求一样在消耗着有限的容器线程资源。于是我们开始尝试自己封装字节流解析模块，按需解析上行数据，一来可以提升数据分析的性能（并行业务和数据增量分析操作），二来可以用最小代价处理异常请求（当发现不满足业务规范时，则立刻丢弃后续所有的数据），这块实现被叫做LazyParser，主要的实现重点就是最小化数据缓存来进行并行业务和数据解析操作，上线后效果不错，整个系统平均处理时间从10ms降低到了4ms。（包含了异常处理的优化和解析性能的提升）

另一方面，Hadoop的MR问题也日益突出，一大堆MR的

Class（类）维护成本高、性能问题也随之出现。此时我们开始尝试抽象分析业务场景，想到的是是否能够通过配置就可以完成各种统计分析需求。要用配置替代代码，其实就看是否可以穷举代码所实现的各种统计需求。当回顾SQL的理念时，发现其实所有的统计在切割成为KV作为输入/输出时，所涵盖的需求无非是Max、Min、Average、Sum、Count、Distinct（这个是2012年实现的，用了bloomfilter和AtomicLong)，再复杂一些无非就是上述几个操作结果的数学表达式运算。因此，KV输入和KV输出的离散统计配置需求已经抽象出来了，接着就是把统计完的一组组KV根据K来做Groupby（分组），就生成了传统意义上的报表（K，v1,v2…）。从此以后，每天的统计需求都通过配置来改变，再也没有一大堆MR代码，同时一次数据输入就可以完成所有分析的处理，性能上得到了极大的提高。

虽然Hadoop每日分析抽象出模型配置解决了性能和易用性的问题，但是对于即时分析却不太适合，当时出于监控的需求，希望能够一个小时就可以对数据做一次增量的分析，用于监控服务整体的调用情况，保证对异常问题的即时排查。由于一天4000次的调用量还不算很大，因此，当时就直接考虑采用MySQL分库分表的方式，然后定时做SQL的查询，结果发现效果不错。当然，这个过程又产生了一个小组件，要直到4000次的日志数据写磁盘和DB双份必然会带来不少的I/O消耗，同时这个系统并不是账务系统，丢掉一点日志也没关系。因此，就采取了异步批量数据外

写的设计（多线程守护各自的一块Buffer页，定时外刷或者满页外刷），这样在双写的情况下，单机的Load也没有超过0.7。

但快到年底的时候，发生了一件事情让我们头痛不已，同时也成了开放平台的一个“隐形炸弹”。一天晚上，突然发生平台大规模拒绝服务的告警，观察整个集群发现，业务处理时间从平均的30～40ms，上升到了1s，仔细观察发现，某一个业务的响应时间大幅攀升，从原来20ms的响应时间飙升到了1s以上，此时由于HTTP请求的同步性，导致前端服务路由网关的集群线程都释放得非常慢，阻塞处理这个业务的请求，而其他正常的业务（淘宝开放平台背后的服务由不同的团队维护，处理时间从1ms到200ms都有）也无法被访问，因此，才有了开始的全线告警的产生。后来发现是这个业务团队的一次发布中忽略了数据库索引建立导致服务耗时增加，但这个问题开始时不时地拜访开放平台，开放平台稳定性受制于任何一个业务方，这是不可接受的。对于这个问题，起先考虑集群拆分，即将重要业务和不重要业务拆分，但考虑到实施成本（不同服务的利用率差异很大）和业务隔离是否彻底（重点业务也会相互影响），最终放弃了这个想法。当时又想到了软负载切割Haproxy和LVS，一个是七层的网络软负载切割，一个是四层的负载切割，由于涉及业务，于是考虑用七层的软负载切割，尝试一台Haproxy挂7台虚拟机，然后运行期可动态调整，配置在出现问题的时候可人工干预切割流量。就这样，我们有了告警以后可以手动切割的半人工方式干预措施。

但我们依然晚上睡不踏实……（期间考虑过Web请求异步化和Servlet3的模式来规避同步HTTP带来的平台阻塞，但当时唯一支持Servlet3的Jetty和Tomcat做压力测试，效果都很不稳定）

2010年：平台化。到2010年年底，平台开放淘宝服务300多个，每天调用量为8亿次，这一年淘宝正式开始对外宣传开放，淘宝开放年赢在淘宝，目前很多年收入上千万的TP在这个时候成了先锋（2010年以前的可以叫做先烈），产品层面上，这一年除了卖家工具的继续发展，SNS热潮的兴起带动了淘江湖的买家应用，游戏应用的淘金者蜂蛹而入，开放的服务也继续保持300%的增速，覆盖面从卖家类延伸到了买家类，从简单的API提供，到了淘宝网站支持深度集成应用到店铺和社区。

在8亿次访问量的情况下，再用MySQL做流式分析已经不可能了，分析时间要求也从一个小时提升到了20分钟，此时经过快一年半的Hadoop使用和学习，再加上对分布式系统的了解，正式开始写第一版的流式分析系统，MR的抽象依旧保留，而底层的数据计算分析改用其他方式，这个“其他方式”和Hadoop的差异在于：

- 分析任务数据来源于远端服务器日志（主要通过Pull，而非Push）。
- 任务分配和调度采用被动分配（有点类似于Volunteer Computing的模式），Mater轻量的管理任务，Slave加入即

可要求执行任务，对任务执行的情况不监控，只简单通过超时来重置任务状态。

- 任务统一由Master来做最后的Reduce，Slave可以支持做Shuffle来减少数据传输量和Master的合并压力，Master负责统一输出结果到本地。

总的来说，就是数据来源变了，数据不通过磁盘文件来做节点计算交互（只在内存使用一次就丢掉了），简化任务调度，简化数据归并。这样第一版本的流式分析出来了，当然，后面这些设计遇到的挑战让这个项目不断在演进，演进的各种优化几年后发现都在Hadoop或者Hive之类的设计中有类似的做法。（参看Blog，地址为http://blog.csdn.net/cenwenchu79，根据时间轴可以看到各种结构优化和性能优化的过程）这个系统三台虚拟机撑住了8亿次的日志即时分析，MySQL日志分析就此结束。

这一年另一个重大改变就是更多的人对开放的价值有所认同，淘宝从一个部门的开放走到了淘宝公司的开放，什么叫做部门开放？就是在10年以前大部分的API开放都是开放平台这个团队来做封装维护，30个API还可以支撑，100个API已经让一个专业的小团队应接不暇（当然不得不承认，迄今为止，淘宝最有全局业务知识的还属这个团队的成员），300多个API这种势头基本上就无法由一个团队来做了，业务变更带来的接口不稳定经常被投诉。因此，我们启动了服务轻量化的“长征项目”，逐渐通过工

具和平台将服务接入变成自动化的方式，将原来开放一个服务需要点对点，手把手花一周时间实施完成的过程，通过自动化服务发布平台，一个人一天时间就可以发布一个服务，并且服务的文档中，多语言版本SDK都自动生成。这样就具备了服务轻量化的基础，然后将各个新开放的业务采用这种模式接入，而老业务逐渐归还给各个业务方去维护。这样，服务的“稳定性”（业务方面）得到了非常大的提升，用户对于服务的满意度也得到了极大的提高。

但这个担子放下了，那个担子又挑上了，在上面谈到后台应用不稳定导致平台整体不稳定的问题在轻量化以后出现的频率和次数更多了，因为发布和维护都落到了后台部门，此时对于各个系统的把控就更弱了，KPI中的稳定性指标基本就没法定了。唯一能够彻底解决问题的办法就是HTTP服务异步化+事件驱动+虚拟隔离线程池。2010年年中对Jetty7做了一次压测，发现Continuations的效果已经可以上正式的环境了，于是开始在Jetty7的基础上做HTTP服务异步化+事件驱动的封装，同时也实现了一个虚拟隔离线程池做配合。具体设计细节这里就不再多说，参看Blog，简单描述原理如下：

- 将前端容器线程和业务处理隔离。（类似NIO和BIO的设计差异）
- 业务处理如果依赖于外部系统，则采用事件驱动的方式来

减少线程等待，同时提高线程占用资源的利用率。（从这点上说，理想和现实还是有很多细节差异的，在实现的时候必须根据依赖系统消耗时间占总时间的比例看是否需要事件驱动，事件驱动带来的切换消耗是比较大的）

- 通过一个大的线程池虚拟设置不同的业务可消耗的最大资源数，来充分共享资源在异常情况下限制业务占用过多的资源（任务处理开始排队，而非无度地占用资源）。

这个组件上线以后，没过几天就发生了一个典型的案例，一个业务在下午2点开始响应时间从10ms上升到了40ms，然后继续上升到200ms，当时给这个业务模拟设置最大的线程资源数是20个，就发现那时候由于RT时间提升，线程资源释放得慢，20个慢慢地被消耗完了，此时这个业务的队列开始从0到100，再到200……（当然，防止内存过多地被占用，会丢弃超过队列长度的业务处理），而其他业务还是正常地使用着资源，平台平稳，到了下午4点多，业务方收到告警修复以后，RT时间下降到了10ms，队列中的请求数量开始减少，最后队列清空，线程资源占用下降到正常水平。

从此以后，震子开心地和我说：开放平台稳定性的KPI可以随便大胆地写几个9了。

2011年：市场化。到2011年年底，平台开放淘宝服务758个，每天调用量19亿次，这一年SNS热潮消退，游戏逐渐淡出，卖家

市场依旧生意火爆，营销工具崭露头角，成为开发者新宠，淘宝客成为开放新宠（这一年返利网和团购一样火，只是前者收钱，后者烧钱）。

就在开放平台前景一片大好的时候，出现了一个让开放转变和收缩的导火索，一家做营销工具的公司“团购宝”每天凌晨都会通过接口同步客户设置的一些优惠商品信息到淘宝网，结果那天凌晨，微博上突然有很多人说某些店都是一块钱的便宜货，要知道这种事情在微博盛行的时代，传播速度之快，影响之大，当即很多卖家商品都被1块钱拍下。最后发现是线下的商品价格不知道怎么全被修改成1块钱，然后凌晨一同步，就导致出现了上面的一幕。从那时候开始，开放平台的KPI中增加了一个重中之重的功能：安全，包括后面的很多技术产品都围绕安全展开。此时第一个被波及提升能力的系统就是流式分析集群，20分钟一轮的数据分析要求压缩到3分钟，同时数据量已经从每天8亿条增长到了每天19亿条，花了两个月时间断断续续地优化集群结构设计和单机处理能力，对于其中经历的内容，有一天我翻Hadoop的优化过程时看到了相似的场景，具体就不在这里赘述，详细内容可参考Blog。简单地说，包括四个方面：充分利用多核能力用计算换内存；磁盘换内存，用并行设计来保证整体业务时间消耗不变甚至减少；Slave Shuffle来减少Mater的合并压力；数据压缩减少数据传输消耗和内存占用。

另一方面，由于2010年对于Jetty7的充分理解和封装，此时看到了又一个新技术的契机，2010年去美国参加Javaone，当时看到有老外用Jetty7的特性来实现Comet功能，Comet原先主要用于CS结构的应用搬到互联网上，因为不能用TCP的长连接，所以不得不用HTTP的长连接来替代原来的模式，同时国外开放平台也关注很多新型的API设计，其中就有Twitter的Streaming API，这种通过HTTP长连接方式推送消息到外部ISV（独立软件开发商）的模式引起了我们的注意。因此，我们决定将Jetty7上的封装近一步升级，支持Comet长连接方式，后端通过事件驱动的模式主动推送内部消息给外部，避免外部轮询业务接口。这个设计最重要的一点就是如何用最有效且最少的线程来守护多个长连接，支持到后端事件驱动的数据下行，如果给每一个长连接支持一个数据推送守护线程，即时性自然最高，但代价就是消耗众多空置连接的守护线程（详细内容见Blog）。这种模式刚出来的时候，从上到下都是质疑声，觉得太不符合常规做法，常规做法就是pull，认为开发人员无法接受，稳定性一定不靠谱。经过2011年的“双十一”，当天几个“尝鲜”的开发者用一台PC就支持几百万笔订单的高速处理，就让很多人明白了，技术要敢想，代码要敢写，细节要敢专，没什么不可能。也就从这以后，多样化服务TQL、Schedule API、ATS从开放平台的土壤上都长了出来，为更多的场景和更多的终端提供了各种解决方案和创新实现。

2012年：垂直化。这一年到现在，平台开放淘宝服务900多

个，每天调用量为25亿次，这一年淘宝客由于公司方向转变热潮消退，无线乘势而起，新业务（机彩票、酒店、理财等）、P4P、数据类服务都开始运营API，开放平台开发者的客户群体也从C店卖家增加到了B的品牌商和渠道商等。

这是一个业务多变的一年，这也是淘宝内部对开放平台认可的新阶段。第一个阶段是放任不管，任由开放平台部门开放和封装。第二阶段是由业务方负责支持开放业务，但开放后的结果概不了解，也无所谓了解。第三阶段就是业务主动要开放，开放后开始运营服务，培养ISV市场，带动业务的正向发展。

这一年由于业务量的增长以及分析需求到用户纬度，因此，在2011年年底启动了流式分析集群重构升级的项目，将新的分析集群项目命名为Beatles，希望它能够象甲壳虫一样，小虫吃树叶，再多都能吃下。2011年底到2012年初，用了近两个半月的时间做了一次完整的重构，将那么多年的补丁经验和老代码重新设计和实现，并且将Mater根据业务可垂直切分，最终解决Master归并压力的问题，当然期间的技术优化点也不少，因为我们的目标从3分钟压缩到了1分钟，而我们的数据量翻番，统计纬度细化到了用户纬度。（意味着结果也会很大，如果不靠文件做中转，如何来实现需要更多的分拆和协同设计）

这一年起了两个比较创新的项目：JS SDK和无线SDK（IOS，安卓），这两个SDK的出现在一定程度上由业务和安全两方面决

定。首先，2011年年底启动了社区电子商务化的项目，也就是现在所说的轻电商（XTao）项目，将更多的网站和淘宝衔接起来，此时网站间的融合就要求更轻便和简易，最成功的案例就是Facebook，于是2012年年初的时候，拿这FackBook的JS SDK一阵看，就开始动手写了，期间很高兴拉了UED入伙，这才使得这个JS SDK变得更加靠谱，更加专业。同时有了JS SDK，买家的服务安全性有所保证，因为原先的REST调用在授权以后是无法知道是用户发起的还是服务器发起的，而JS SDK从一定程度上还要校验Cookie的有效性，可以部分保证用户的在场和知情。而下半年的无线SDK，就是苦读一个月的各种文档，然后就开始动手玩儿了，由于对Java语言、动态语言、脚本语言都有比较多的使用，因此，Objective-C语言上手并不是那么困难，同时没有涉及过多的MVC的内容，做SDK基础层的东西还是比较得心应手的，就这样IOS的无线SDK版本就生成了，此时在开放平台的技术团队内部正在执行一个叫做Hack project的活动，其中一个自主项目就是安卓的SDK，因此，一个月后，安卓的SDK顺利诞生了。这两个无线SDK所担负的职责就是把控无线安全问题，不仅是淘宝，业界其实很多公司都还没理解无线开放的风险到底有多大，OAuth2基本就无法保证无线的用户安全，因此，如何在SDK和服务端融入更高级别的安全设计，成了无线SDK诞生的第一个重要需求。

另一方面，开放平台安全体系的构建成为2012年的重点，从两个角度对安全做了全方位的控制。

第一，用户。用户授权更细化了授权操作范围（细粒度到了数据范畴），授权时长。所有的信息可监控、归档、快速定位，我们内部叫做Top Ocean，简单说来就是对所有的访问日志做归档，归档的载体是块状文件，归档时对块状文件的所有记录按照需求建立索引，然后保留索引，上传本地文件到远端分布式文件系统备份。实时的监控服务调用和应用访问，授权异动。

第二，第三方应用。采用监控集群对所有ISV的服务器做安全扫描，对普通的Web安全漏洞做扫描，对应用的可用性和响应时间做监控。同时，正式启动“聚石塔”项目，提供弹性计算和存储能力及可靠的安全网络环境给ISV，帮助ISV提供自身应用的安全性。

至此为止，5年左右的技术历程已部分展示在了大家的面前，这些只是5年中比较有代表性的一部分，同时技术的发展也只是开放平台的一部分，前5年是技术变革带动开放平台发展，而接下去的5年将会是业务变革和理解带动开放平台的阶段，对业务的理解直接决定了开放平台的价值所在。前面轻描淡写地介绍了5年来不同开放业务的兴衰，其实这背后却有更多耐人寻味的故事，而5年后的今天，淘宝的格局为：集市（C2C）、天猫（B2C）、一淘（电商搜索返利入口）、无线、新业务、O2O（本地生活）、团购平台（聚划算），这些平台的价值是什么？如何找到自身定位？如何借助外力发展？如何面对流量入口的兴起、传统互联网企业的电商化、电商平台的竞争？这些才是开放平台2012年及下一个5年的精彩所在。