蘑菇街电商交易平台服务架构及改造优化历程(含PPT)

原创：潘福江 高可用架构 2016-08-11

导读：高可用架构 7 月 30 日在上海举办了『互联网架构的基石』专题沙龙，进行了闭门私董会研讨及对外开放的四个专题的演讲，期望能促进业界对互联网基础架构的建设及发展，本文是潘福江分享蘑菇街电商交易系统架构。

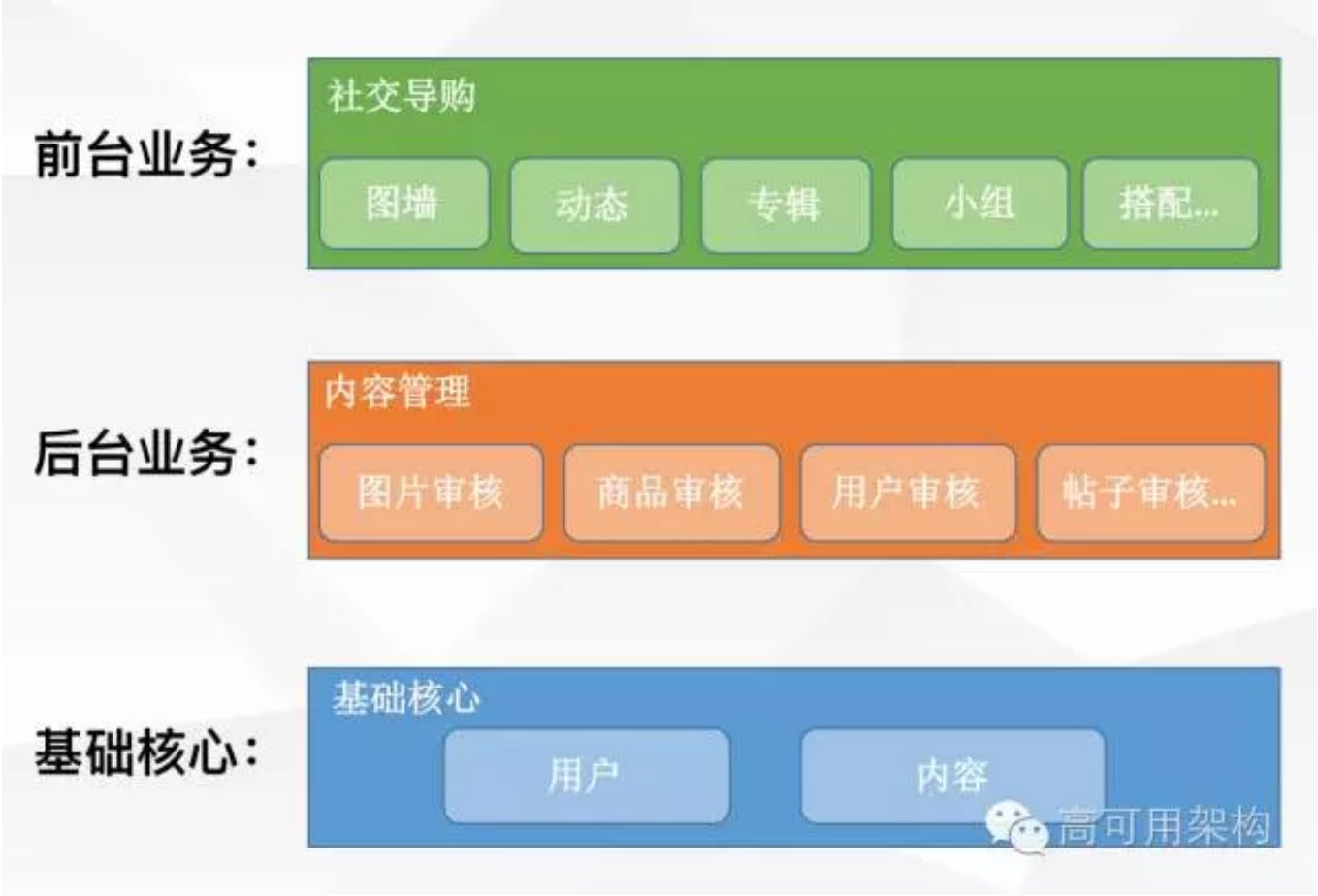
潘福江，蘑菇街高级研发工程师，2014 年之前在阿里，搞过电商垂直业务平台建设，也搞过中间件相关的研发工作，2015 年加入蘑菇街（现美丽联合集团），负责蘑菇街交易资金，购物车等电商基础平台的服务化建设工作。

我来自蘑菇街，蘑菇街是一个主要面向女性用户的电商平台，男同胞们可能用的比较少。不过蘑菇街里有大量的模特妹子，而且颜值都比较高，建议大家可以下来用用，写代码累的时候，可以偷偷打开蘑菇街看看妹子，感觉还是很不错的。

今天我的主题是蘑菇街交易平台的服务架构，以及在服务化建设过程中，我们做的一些改造历程分享。

蘑菇街导购时期 业务结构

蘑菇街是做导购起家的，当时所有的业务都是**基于用户和内容这两大核心**展开。那个时候前台业务主要做的是社交导购，后台业务主要做的是内容管理。一句话总结就是小而美的状态，业务相对来也不是很复杂。



当时的技术架构是典型的创业型公司技术架构。网站整体是用 PHP 搭建的，系统做了简单的分层，基础设施以现成的开源产品为主。2013 年时蘑菇街做了转型，主要原因那段时间很大一批导购网站遭到了封杀，于是就转型做社会化电商平台。

社会化电商平台分两部分，一部分社会化，我们之前做导购时积累了一些经验。电商是我们之前没有接触过的，这块基本上是从零开始一手建立起来。要做电商平台，首先就要搭建一个交易平台。起初比较简单，我们重写了一套系统，系统结构和之前相比并没有本质上的变化，所有业务都写在一个巨大的工程里面, 中间通过一套代理层和我们的基础设施进行交互。

电商转型面临的问题

- 业务高速发展，每年保持 3 倍以上的增长（2015 年用户过亿，PV 超过 10亿）
- 用户购买链路大促峰值是日常的百倍（2015 年初最高只支持 400 单/秒）
- 业务异常复杂，业务形态快速膨胀
- 历史包袱沉重，系统耦合非常严

蘑菇街转型到电商平台以后，业务基本上每年以三倍以上的速度增长，这个时候问题开始暴露了。电商平台在发展过程中尤其在发展中期遇到的一些的问题，不仅仅是蘑菇街的，其它平台可能也会遇到。如系统代码臃肿、模块耦合程度高，依赖复杂，业务扩展能力差等。

蘑菇街那个时候主要面临了几个问题：

一个是我们业务在高速增长，**系统容量跟不上**，当时交易系统只能支持到每秒四百单的容量(大促的时候流量是平时的百倍以上)。

另一个是电商业务形态变的特别快，**业务支撑不够灵活，不够快**。

此外还有历史包袱，**系统耦合非常严重**。

解决这一系列问题的关键就一个字：“拆”。

系统拆分历程

- DB 垂直拆分

- 业务系统垂直拆分（购物车，下单，资金...）
- 数据 & 业务模型统一，服务接口设计逻辑清晰，粒度合适
- 基础业务逻辑下沉到服务，Web 层专注表示逻辑和编排
- 服务治理

系统拆分——交易购物车为例

以交易购物车的例子来说明下我们的改造过程，以前我们就一个工程，所有的代码都写在这，不同的终端或业务都有一个不同的模块代码在维护。访问数据也比较随意，各自维护一套数据访问的代码。因此就有两个非常头疼的问题：

一方面由于交易就一个库，所有内容都是放在里面，因此这些随意散乱的 SQL 可能会冷不丁的给你来个慢查询，别的业务代码带来的不稳定会相互影响，还很难定位这种“野 SQL”是哪里查过来的，导致我们的 DB 很不稳定，对后续改造非常不利。

另外一个是业务支撑方面，产品提一个需求过来，在各种端都要实现一遍，复用性很差，业务支撑非常不灵活，系统毫无扩展性，开发同学也是苦不堪言，经常加班加点搞，还经常搞出一堆 bug。

于是我们就去拆系统，到底怎么拆？其实也是有些讲究的。

系统拆分的优先顺序

如果把 DB 比如成一个木桶，各类业务就可以比喻为往里面倒的水，一开始往木桶里面倒的水可能并不多，木桶装的下没问题。但是随着业务增长，木桶总有一天是会装不下的。

首先木桶需要足够大，并且能很方便扩容，这样才不会有后顾之忧。业务量有时候并不好预测，指不定什么时候量就起来了，如果不把这个底层的木桶做得足够强大，而优先去搞业务上的拆分优化，那量一旦起来整个系统就歇菜了。

因此 **DB 是系统拆分的基础，需要优先拆分**。

DB 拆出来的同时还要关注稳定性，前面也提到，当时 SQL 是比较散乱，极易造成 DB 不稳定，所以数据访问/模型统一也很关键，**我们建了统一的数据访问层**。有了这一层之后，后面对 DB 的改造扩容都能够比较有效的掌控。

基础的东西都建好了，再来解决业务支撑困难这个问题。**业务模型上需要统一抽象，能够支持定制扩展**。流程改造过程同时也孵化出了 **SPI 业务框架、流程引擎、规则引擎**等这些基础业务框架。在业务支撑上做到了灵活可扩展。系统也做了比较合理的分层，每层只需要关心本层所需关注的能力即可。

系统拆分成果

交易系统在整体拆分完成以后，公司 SOA 化雏形也基本已经形成了，包括**基础服务化框架、消息中间件、数据中间件、配置中心**也都落地实施了，此外还孵化了一系列基础设施工具，包括**监控系统，调度系统，日志搜集，链路跟踪系统**等。

还有一个背景拆分过程中，公司战略整体往 Java 语言转，这个是公司综合层面考虑，Java 的人才尤其是杭州相对比较多，技术体系也比较成熟，有大牛在能 hold 住问题，当时确实 PHP 资源比较少。

容量提升

做了系统拆分改造以后，接下来更多的会去关注应用本身的容量、性能以及稳定性方面的事情。我们也在这些方面分别作了一些改造和尝试。

- 按业务对 DB 进行垂直拆分
- 读写分离，保证读可以任意扩展
- 分库分表，提升中心服务写入容量

系统拆分时，已经按业务把 DB 垂直拆分出来了，并且 DB 也做了读写分离（基于 MySQL）。

下面重点介绍一下分库分表上的改造，当时目的主要是为了提升中心服务的写入容量，因为当时 DB 读写分离是单 Master 结构，会有一个写入瓶颈。

以交易创建为例来说明我们分库分表的历程，交易创建应该算是交易里面最为复杂的业务场景之一。创建一笔订单的时，会同同时写入其他很多的数据，当时系统容量大约是每秒能够处理一千单，DB 单点存在写入瓶颈，并且写入过多会造成主从延迟严重。另外 DB 磁盘空间也已经突破了 80%，不稳定性非常高，有可能随时会崩掉。

所以我们就决定去做拆分，当时的背景是中间件还没建立起来，没有分库分表相关的组件，于是就决定内部先搞起来。

当时对比了业内比较流行的一些方案，比如阿里的 TDDL，Cobar，谷歌的 Vitess 等，比较下来发现这些组件都比较重，接入和使用成本都相对比较高。我们的原则是符合我们业务场景下，选一种接入和使用成本都相对简单的组件。于是我们采取的是最后一种方式，通过 MyBatis Plugin 字节码增强的方式实现分库分表功能，该组件目前已开源：https://github.com/baihui212/tsharding

分库分表业界方案对比如下图：

自研分库分表组件 TSharding，完成分库分表

- 足够简单，投入较少资源
- 支持分库分表
- 支持数据源路由
- 支持事务
- 支持结果集合并
- 支持读写分离

这个组件叫 TSharding，它的特点是足够简单，是符合我们预期的，支持分库且分表，支持数据源路由，支持事务，支持结果集合并，支持读写分离，满足我们所有的要求。

性能优化

我们在性能优化上也做了一些尝试，主要举下面三个场景：

- 分布式事务处理
- 单机异步并行
- 预处理 & 缓存

分布式事务处理----交易创建为例

优化思路：异步消息解耦

1. 交易创建流程中，订单、券和库存的状态必须要保证一致性
2. 营销优惠券服务和库存中心库存服务，与下单服务是分开部署的
3. 调用券/库存服务超时/失败，异步发消息通知回滚；复杂性可控
4. MQ 产生端发送失败重试 + 消费接受 ACK 机制保证做种一致

5. 消除了二阶段提交等分布式事务框架的侵入性影响

首先讲分布式事务处理，这里还是以交易创建为例，交易创建过程会跟多个服务进行交互，并且有些服务是强依赖，比如扣减库存，锁券服务，必须保持一致性。**二阶段/多阶段协议这种方式很重，当时并没有采纳**。

我们当时想了一个方法是，**通过异步消息解耦的方式来解决**，具体流程：

下单时先不要急着让这个订单暴露出来，我们先创建一笔不可见的订单（或者可以认为是先预创建了一笔订单），然后再去做减库存，锁券操作，当这些操作异常或者失败时，下单系统就会发出一条废单消息，它的下游系统（如促销，库存系统）收到这个废单消息以后，就会帮我们做回滚的操作，用这样的方式来解决我们分布式事务的问题。

分布式事务处理——支付回调为例

1. 支付回调流程中，资金系统回调交易后会促发订单状态更新，减库存，发券等操作
2. 资金作为发起方保证重试，消息可达，交易以及下游做好等
3. 失败业务进任务重试表，做异步补偿重试
4. 消除了二阶段提交等分布式事务框架的侵入性影响

还有一个场景就是支付回调，支付系统在订单完成付款之后会通知交易系统，交易系统会进行一些列的操作如定单状态更新，减库存，发券等，也是一个分布式事务问题。

我们的策略是，当业务失败的时候这个请求会进入我们的一个失败补偿表，通过不断的做异步补偿重试(阶梯式)，保证最终一致性。

单机异步并行——购物车为例

分析思路

1. 购物车是典型 IO 密集型应用
2. 代码串行执行，同步等待时间较长
3. CPU 利用率低

分析一下，购物车其实本身是一个典型的 IO 密集型的应用，有很多类似的应用都是这样的，会存在大量的网络 IO 请求。还有一点是我们习惯上代码是串行写的，所以存在很多同步等待时间。

既然每次购物车的查询都会经过这么多的节点，那如果两个节点之间没有依赖关系，我是不是就可以并行的去搞，分析一下其实每一次查询都会对应一颗查询依赖树，同一层节点之间是没有依赖关系的，在这一层我们其实是可以去做并行操作的，所以基于这个思路我们当时做了优化，并且效果还挺不错。

具体的优化就是加入这个概念，去查的时候我们先等一下别的查询，我们一块儿去查，最终做一个汇总，查询结果就出来了，就是这么一个流程。然后做的效果也不错，整个 RT 基本上能够降到一半以上。

预处理 & 缓存——营销计价服务为例

- 使用缓存降低 DB 读压力
- 尽可能地缓存需要的数据
- DB 数据有变化主动失效缓存（异步，低延迟），减少不一致问题

- 大促高峰前开启本地缓存；做缓存预热，有助于提高缓存命中率
- 预处理达到计价接口部分耦合：报名大促活动商品的折扣同步到商品表

预处理跟缓存化，其实也是一种比较通用的优化手段。我们采用了多级缓存的策略，本地缓存+分布式缓存。优先读取本地缓存，本地缓存没有就去分布式缓存取。分布式缓存取不到才去 DB 里面取。当数据发生变化的时候我们会有一套异步刷新缓存的系统来及时更新缓存里面的数据。

服务 SLA 保障

SLA: Service Level Agreement，是对服务提供者的要求。SLA 体现在对容器（QPS）、性能（RT）、程度（分布情况；可用性；出错率）的约束。提高 SLA 的一些手段如下

- 基础监控先行，把关键指标监控起来
- 依赖治理、逻辑优化：减少不必要的依赖
- 负载均衡；服务分组；限流
- 降级预案、容灾、压测、在线演练

这个是我们内部的一个监控系统，我们会监控每个应用的一些关键指标，观察整个链路上的情况。

总结及下一步规划

总结

- 服务化构架不是一成不变的，是随着业务不断发展而演化
- 没有最佳的方案，在合适场景下采用合适的方案

目前在做

- 服务治理、SLA 保障系统化

下一步

- 同城/异地双活

Q&A

提问：如果消费者收到一条消息，这时候我就告诉系统可以把这个消息删除了，那我后端在执行这条消息的过程当中，比如我做一些入库操作或者其他的操作，但是这个服务死掉了，那么你们有没有遇到类似这样的情况，你们是怎么来处理的?

潘福江：目前没有，因为其实这是一个需要合作的过程，我们需要下游系统去配合保证，保证业务 OK 之后才去 ACK 消息，主要是几个系统之间的协作问题。

提问：电商大部分分布式事务解决方案用消息队列的机制，有没有一种更通用的解决方式？我们开发一套分布式组件，比如两阶段协议，更高效解决这种分布式事务。

潘福江：分布式事务问题其实还是看场景的，支付宝（以前在支付宝）有类似的框架，但是比较重，也需要一定的接入成本，需要一定的配合。Case by Case 的分析问题会更好一些，比如有些场景他的一致性要求并没有那么高，没有必要采用两阶段协议这种来处理，主要还是看业务场景。

提问：数据库迁移的时候，能做到平滑迁移吗？因为我看到你之前用的中间件的时候切换过两次，这个时候肯定遇到一些数据库在线平滑迁移，你是怎么弄的？

潘福江：这个我们内部会有一套数据同步工具，另外有套开关系统来完成灰度切换，你可以动态去推送一些值，到你的应用里面，然后动态的可以把这个值改掉。此外我们这个数据同步工具是支持回溯的，遇到紧急情况可以迅速切回，并且把数据回溯回来。

提问：你做了一个分库之后你要把老的库保留到新的库里面，因为上去之后你要分步发布，但你的老库还在跑，这个时候有人在用你的老库，但是因为你又在发布，一半流量已经切到新的库上去了，如果这个时候有人正在修改数据怎么办？

潘福江：上文也提到，老库和新库是有建立一个通道的（数据同步通道），并且数据同步工具一直在工作。老库的数据会实时的同步到新库，并且我们的灰度是通过开关系统动态推送的，实时生效。

提问：在分库分表的拆分之后，假如有表的关联查询你们怎么处理的？

潘福江：我们貌似没有关联查询，也不推荐，关联查询对后续水平拆分十分不利，不利于 DB 的扩展。可以分开查，然后在应用层把关联的事情做掉。

提问：如果分开去查的话，性能上会不会受影响呢？

潘福江：性能上可能是会受到一定的影响，会多访问几次数据库，但是 DB 扩展性能大大的提升，互联网玩的是大数据，比起这点，DB 扩展性更重要，应用层可以有很多的方式去优化(比如缓存)，如果你用了 JOIN 再去做水平拆分是非常困难的。

提问：你们在拆库之前，有没有考虑什么好的方案回退？

潘福江：我们的数据同步工具是支持回溯的，出了问题通过开关系统可以立马切回，并且能回溯数据，把影响面降到可控范围。

相关阅读

（点击标题可直接阅读）

- Mercury:唯品会全链路应用监控系统解决方案详解
- 同程旅游缓存系统设计:如何打造Redis时代的完美体系
- 保证分布式系统数据一致性的6种方案（含蘑菇街方案）
- 对话：一个工程师在蘑菇街4年的架构感悟
- 10个互联网团队应对高压的容量评估与高可用体系:私董会1期

本文及本次沙龙相关 PPT 链接如下，也可点击阅读原文直接下载

http://pan.baidu.com/s/1nvnOEBf

想更多了解高可用架构沙龙内容，请关注「ArchNotes」微信公众号以阅读后续文章。关注公众号并回复 ***城市圈*** *可以更及时了解后续活动信息。转载请注明来自高可用架构及包含以下二维码。*

高可用架构
改变互联网的构建方式

长按二维码 关注「高可用架构」公众号