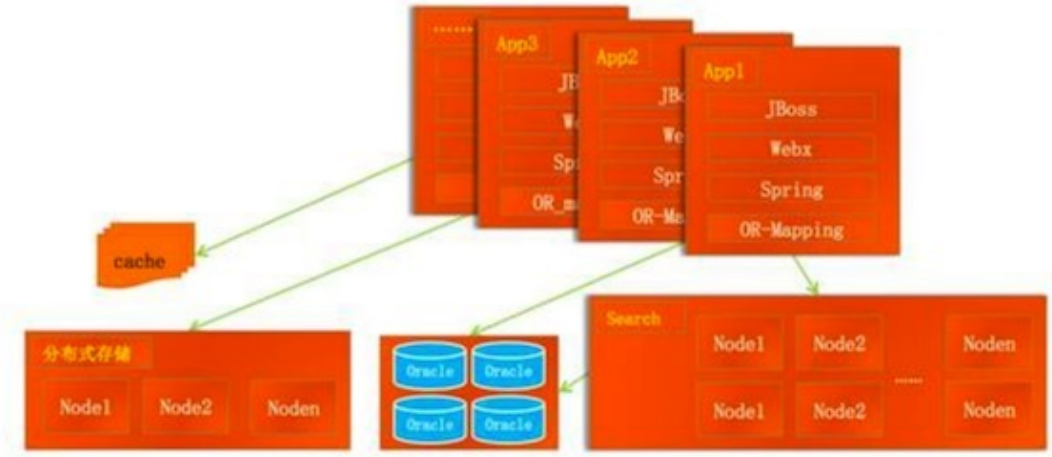
阿里技术架构演进及过程中遇到的问题

2003 年，淘宝最初的架构建设是采用 PHP，实践发现系统抗压能力相对薄弱。因为淘宝是一个企业级系统，所以选择了当时非常重要的企业级技术 JavaBean。之后把整个 Web 的容器、EJB 等整套体系引入淘宝，雇用有经验的工程师一起做架构。淘宝在技术方面也走过很长的路，在架构建设过程中，大家讨论最多的事情是如何划分模块。

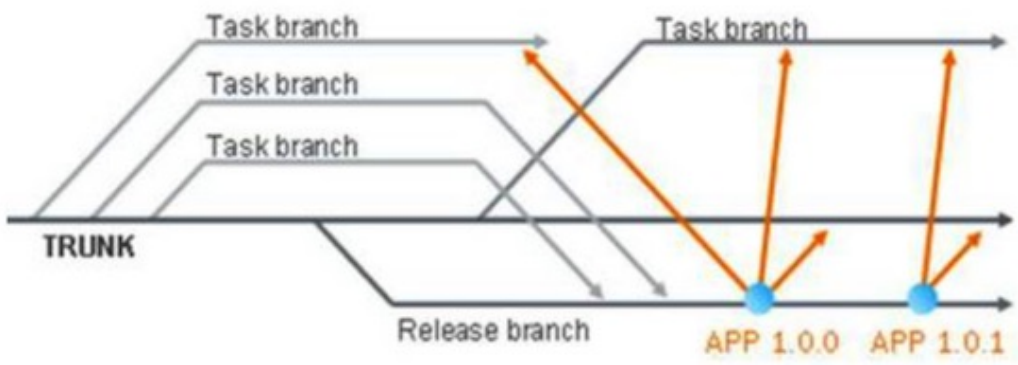
随着技术的不断发展，到 2006 年，淘宝技术又从 EJB 过渡到 Spring，如下图：



目前，这个产品已经开源，最上层采用的是 JBoss、中间采用 Webx，之后是 Spring、OR-Mapping，底层用到是 Oracle 数据库。用 Search 做搜索引擎，是因为当时收购雅虎，把雅虎的搜索引擎挂接到淘宝。像这样采用分布式的存储方式在现在看来很常见。

当时，业务不断高速发展，一些问题随之逐渐暴露出来。这里主要分享工程维护、人员变动、数据孤岛、数据集能力不足等问题。

工程维护与人员变动



随着业务不断壮大，技术团队的工程也会越来越多，源代码加速膨胀。多个工程之间，源代码冲突严重同时因为工作没有边界导致相互协同的成本不可估量。

假设出现项目完成，核心人员离职的情况，项目维护也会成为问题，当新人入职之后，学习老代码的难度也可想而知。

数据孤岛

数据孤岛是各个公司很普遍的问题，在那个时期，天猫还叫淘宝商城，不是基于淘宝，而是完全独立的一个组。

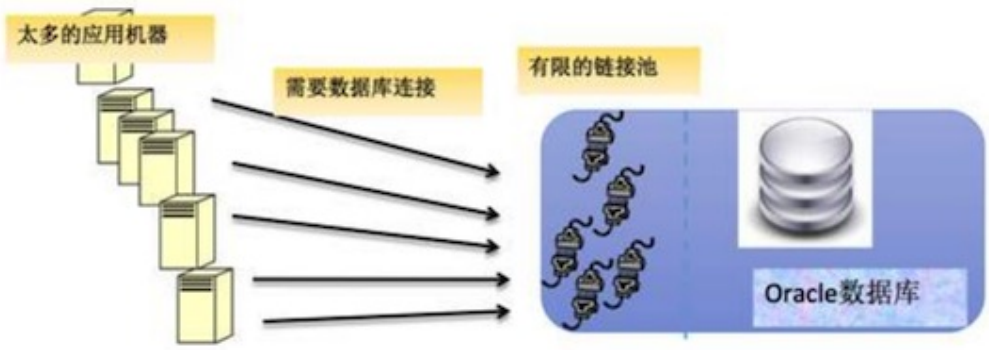
后期因为一些原因，想要把两个体系合并，却因为各自独立的业务体系、用户 ID、数据存储格式等等差异导致操作困难。且数据本身质量不高，做统一分析也有难度。

数据库能力达到上限

当时用的是小型机+Oracle，CPU90% 以上，每年宕机最少一次。这主要是因为有大量新业务写入，两周一次的频度，不断地有新 SQL 产出。

在新的 SQL 中，如出现一个慢 SQL，就会出现宕机。当时我们用的小型机重启一次需要 20 分钟，切换到异地也是 20 分钟。

关于连接数问题，如下图：



当时后端 Oracle 的连接池有限，约 8000 个左右，一旦超过就会出现问题。因为超过数量，链接占的内存会非常大，且连接数单点风险系统很高。

阿里面对 DBA 相关问题的应对方法

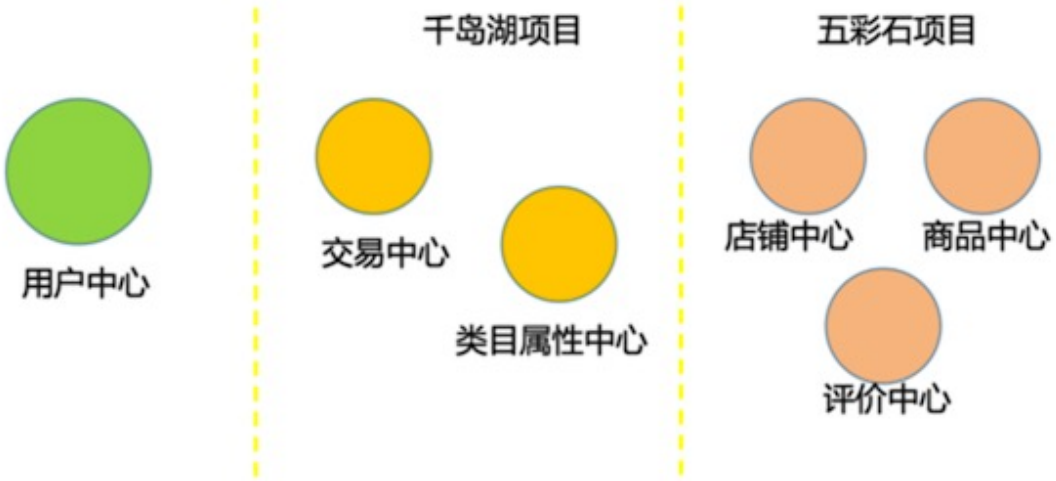
综上所述，当时阿里 DBA 面临维护人员很多，团队职责不清、数据无法共享，团队各自为战、小型机数据库压力过大，连接数单点风险系统很高等问题。

好在阿里那时正处于增长期，所以这时通过招聘一些技术大牛来解决问题。

基于 EDAS 进行服务化改造

针对阿里 DBA 遇到的问题，从硅谷请来的技术人用服务化的方式试着解决。当时在中国只有用友做过服务化，且效果不是很好，没有借鉴，只能谨慎小心的自己往前走。

如下图，是阿里以服务化方式将系统专业分工的三个关键战役。



用户中心服务化

选择用户中心的第一个是做服务化，因为用户中心是最小集合，最简单清楚，还因为确实有业务需求，也是想要验证这条服务化的理念是不是正确。

服务化之前的用户中心，有六个不一样的查询方法，看起来遍历的方式差不多，但可能某个参数不同，因为数据来自不同的团队。

服务化的原则是能不改不改，能简化简化，采用的传输方式是 HTTP。然而，这样做行不通，是因为除了服务化 HTTP，其他内容没有改变，就需要布设 Load Balance。

为了保证 Load Balance 尽可能稳定，所以选择硬件 F5 来配置。把前端进入的用户流量打到 F5，额外在增加新 VIP 接口，请求通过 F5 转出去。

这里发现一个很严重的问题，就是每当用户登陆一次，出现一个节点，跳转一次流量就要增加一倍。但F5 是很贵的设备，未来如果所有都变成服务化，用 F5 就不可行。

千岛湖项目

配置 F5 负载均衡行不通，换了另一种思路就是由集中的单点模式变成真正意义的分散模式。当时阿里把这样的方式叫软负载，做的是分散负载均衡的事情。

当做交易中心服务化时，必然要用事物相关的方式，来保证整个流程的稳定性、一致性，当时采用的是最终一致性的设计方法。之后，通过实践反复修改，优化，得到稳定的消息系统。

有了消息系统的研发经验，随后类目属性等中心也随之服务化，之所以叫千岛湖项目，是因为大家很辛苦，完成项目之后去千岛湖旅游。

五彩石项目

随着千岛湖项目完成，底层架构、中间件的稳定，之后要做的事情就是把庞大的系统全部一次性服务化。

恰逢此时，淘宝商城和淘宝需要合并，所以整个系统在那个时期进行了彻底的拆分，也就是淘宝 3.0。

之后再也没有出现 4.0，一直采用服务化的架构方式。

架构解耦之：共享服务化体验

为了解决业务扩展性问题，首先需要建立**共享服务层**，把公共的业务元素抽离出来形成共享的服务。

比如 tao bao.com、tmall.com、ju.taobao.com 等应用，这些都需要用到会员服务，那么就把会员服务作为共享服务抽取出来，任何系统需要获取会员信息时只需通过调用会员服务的API就可以，而不需要每个业务方自己再开发一套会员系统。

同样思路，把电商业务公共的服务，如 商品服务、交易服务、营销服务、店铺服务、推荐服务、库存、物流等从各个业务抽离出来建设成共享服务，后续新建的业务市场均基于这些公共的电商元素来构建。

当时并没有商业软件可以使用，也没有合适的开源产品可以选。五彩石项目第一次大规模使用了中间件。系统分布式后，需要有一套统一的组件来解决分布式引发的共性技术问题。

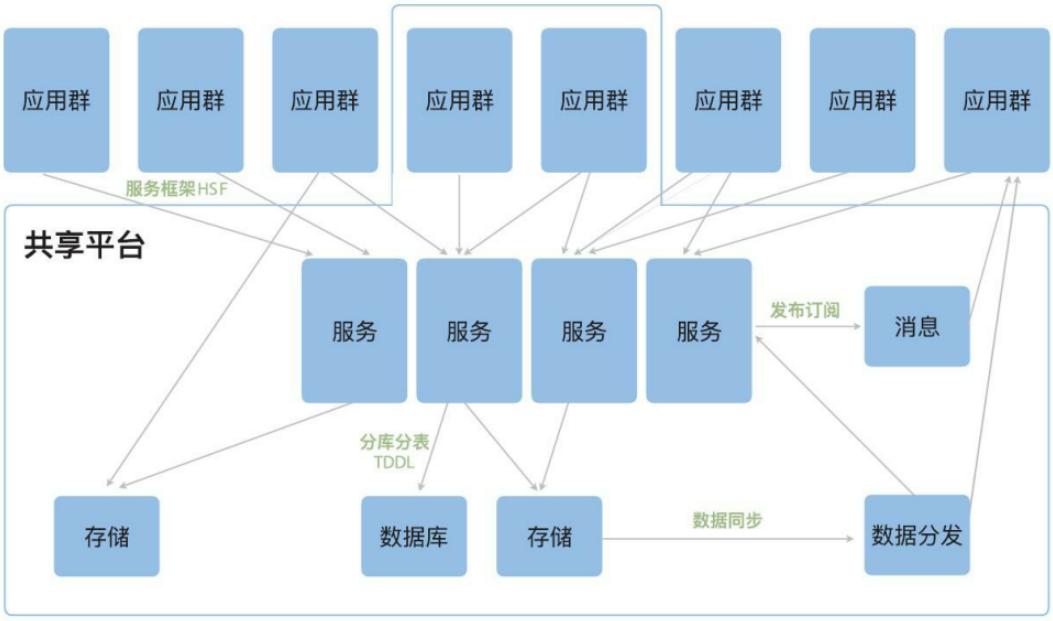
比如提供服务的发现机制、提供服务的分组路由机制、同机房优先机制等。我们将其沉淀在一个框架里，这个框架被称为HSF。

为了解决单库性能瓶颈问题，使用分库分表的技术，这个技术被沉淀在TDDL框架上面。

为了解决分布式事务的性能问题，把原本一个事务的工作拆成了异步执行，同时必须要保证最终数据的一致性，我们采用了消息发布订阅的方式来解决，这个消息框架就是 Notify。

有了HSF、TDDL、Notify这『三大件』，有效地解决了应用分布式后引发的技术扩展性问题，同时让整个系统的技术架构变得依旧如当初一样的简。

如果系统计算能力不够，基本上能做到只需要增加服务器即可。共享服务层和分布式中间件使频繁的业务变化封闭在了一个适合的系统层，同时技术的变化也隔离在了一个合适的范围。如下图所示：



五彩石项目的问题

我们一度认为阿里电商交易业务不会再有水平伸缩能力的问题。
直到2013年双11准备阶段，按照五彩石项目后每年双11的通常动作加机器，结果在加机器的过程中我们发现，整个集群中有个别系统出现了达到瓶颈的状况，紧急改造后才勉强支撑住。在接下来的系统架构复盘工作中我们发现，尽管随着五彩石项目的改造，整个系统的架构形成了一个巨大的可伸缩的分布式系统，但仍然会有几个组件是集中式的。事实上，随着不断增加机器，这些集中的点出现问题是迟早的事情，这也意味着架构要进行新一轮的升级改造。
在2013年年初的时候，我们也看到了另外一个问题——系统发展严重受限于数据中心只能部署在一个城市，并且随着规模的增大，单个机房的不稳定性也明显增加，这就产生了把系统部署在异地机房的需求。

五彩石项目完成后的3.0版本架构面临两个问题：

（1）集中式组件的存在影响到更大规模的水平伸缩能力；
（2）同城部署使得数据中心发展受限，同时还带来了稳定性隐患。

3.0版本之所以在更大规模时出现了水平伸缩能力问题，主要在于一个庞大的分布式系统中尚有若干集中式的节点存在，而要去掉这些集中式节点基本是没办法做到的。经过分析和讨论，我们认为一个比较好的解决办法是限制分布式系统的规模，当这个分布式系统达到一定的规模后，例如5000台机器，就再搭建一个新的。换言之，如果架构最终的目的是将糖果塞进盒子里，那么2.0版本选择了将尽量多的糖果塞进一个盒子里，哪怕最后的结果是部

分糖果被挤变形；而3.0版本的理念其实更像是不断增加盒子的数量来实现盛装更多的糖果客观上来说，出现如此大规模水平伸缩能力问题的业务并不很多，目前只有在交易业务上出现了，所以我们把这轮改造又称为“交易单元化改造”。

单元化要做到可以按照单元粒度伸缩，必须做到以下两点：
（1）用户流量可以随单元增加而均衡分配。假设当前是一个单元，如果增加了一个同等机器数的单元，那么应该可以给每个单元平均分配50%的流量。
（2）每个单元具备独立性。这意味着单元之间不应该有强交互关系，这样才能确保增加单元时不会因为有集中组件造成伸缩瓶颈。

在2013年启动这个项目的时候，我们认为单元化方案有非常多不清晰的细节需要摸索，存在较高的风险，为了尽可能地控制单元化给业务带来的风险，我们给单元化项目制订了一个三年计划，如图1-6所示。

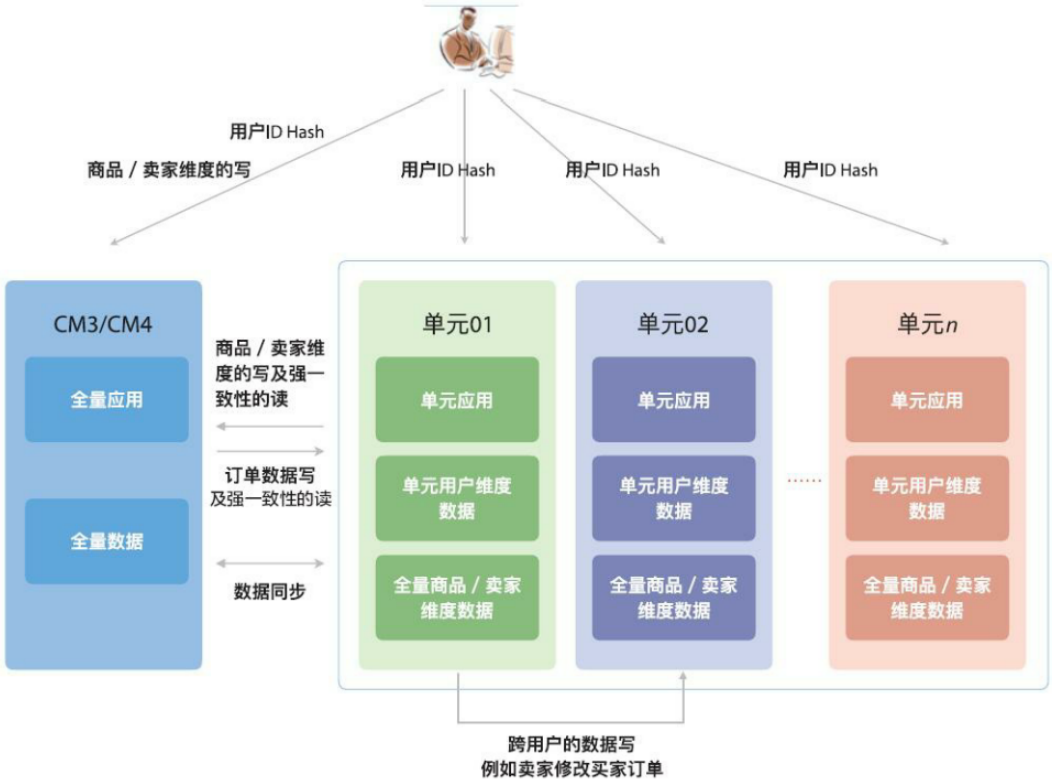
2013年：同城双单元

2013年2月28日单元化项目正式启动 。在这个会上，一方面给所有团队讲解单元化的必要性，另一方面详细阐述了三年计划及初步的方案框架。在陈述方案框架的时候，连宣讲的人都觉得不够清晰。启动会结束后，多个团队的同学进入了单元化一期的项目组。我们要做的第一件事就是确定单元化方案的方向选择。在多轮探讨后，大家就单元化如何做到独立性达成了初步一致。

形成交易单元和中心两个概念。**单元化主要解决的两个核心问题是伸缩能力和容灾能力问题**，同时单元化方案势必要对业务进行改造，但是并不是所有业务都需要去做相应的改造。在详细分析论证后，我们认为交易是必须做到单元化的，其他的非交易业（例如卖家业务等）在伸缩和容灾上所面临的挑战尚不需要采用单元化如此复杂的方案来支撑。根据这样的分析，我们把做了单元化的交易称为交易单元，把其他没做单元化的业务称为中心——中心只能在同城部署，交易单元则可以在异地部署。

基于买家数据划分单元，将卖家/商品数据从中心同步到所有单元。要做到单元请求处理的独立性，最重要的是数据。为了确保数据的一致性，要求同一条数据只能在一个单元进行处理，如果多个单元都修改同一条数据，那么数据的冲突将难以避免。

基于这样的原则，单元的数据必须做一个切分。交易数据的维度有买家、卖家和商品，最终选择的是以买家为基准。卖家/商品数据的修改集中在中心节点完成，中心成为一个拥有全量业务和全量数据的节点。在买家的请求处理和数据读写基本都封闭在单元内后，单元的独立性自然可以实现。



2014年：异地双活

按照计划，2014年将在一个距离比较近的城市部署交易单元，并在双11中启用，每个单元分别承担50%的用户流量。
2013年由于是在同城，网络延时几乎可以忽略，2014年尽管选择的是一个距离比较近的城市，但单次的网络延时仍然有5ms左右。这就意味着对于像阿里这样体量的大型分布式系统而言，如果单元化改造时不解决网络延迟问题，一个页面的访问延迟可能从5ms放大到500ms，那样的话，业务基本就不可用了。

因此，2014年单元化改造的重点是要在2013年方案的基础上，对交易链路上涉及的所有业务进行改造，同时改造一期中未涉及的中间件，以解决异地网络延迟给系统带来的问题。

单元化二期项目从2013年12月就开始准备，在2014年2月进入改造阶段。有了一期的铺垫后，二期的整个改造进展较为顺利。

二期项目在单元化细节上也做了更多的完善，例如去除用户进入不同单元的多域名的方案、**单元化流量切换的系统**、单元化梳理的系统 等。

由于二期涉及的业务改造数量远比一期多，在实施时间上还是面临了不小挑战。**在2014年双11准备阶段，单元化改造项目一直被列为最高风险**，从8月份开始就不断地折腾，最终在10月份的**双11全链路压测**中才完全通过考验。

2014年双11，我们按计划启用了部署在两个不同城市的交易单元，每个交易单元承担50%的用户流量，完美地通过了双11巨大流量的考验。

2015年：异地多活

2014年双11成功启用异地双活后，2015年被我们定义为单元化项目的收尾阶段，在这个阶段中，最主要的目标是形成多地多单元的架构。
2015年年初，在做当年双11的机器预算时，我们依照交易单元的机器数量进行了规划，这意味着**以单元为粒度的伸缩能力**已具备。

> 最终我们决定在2015年双11形成三地四单元的架构。

从异地双活向异地多活演进的过程中，除了继续完善之前未改造完成的一些跨单元交互节点外，同时还需要改造一些在异地双活中被遗漏的改造点。

例如，在异地双活时为了保持单元的容灾能力，**两个单元里的数据量都是全量**，这个时候单元到中心、中心到单元的买家数据同步只需确保不循环同步就可以。

但在异地多活中，因为不是每个单元的买家都需要保持全量，**所以在单元到中心、中心到单元的数据同步时就要支持按规则过滤**的功能。

在异地双活时代，流量切换时会先禁止流量变更中涉及的用户的所有数据库写动作，直到流量切换完成才恢复。

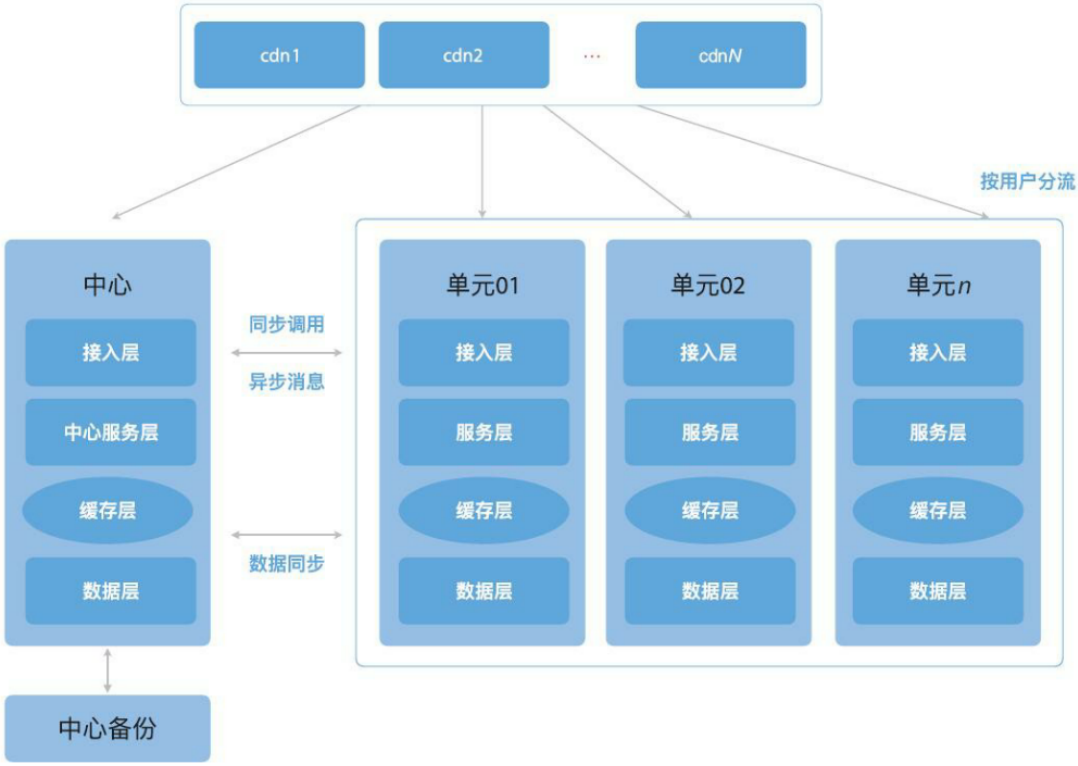
这个方案的问题在于，如果在切换时，用户之前所在的节点出现了因网络中断等导致数据未同步的情况，就会造成流量切换一直完成不了，故障持续时间也会较长。

如果这时忽视数据同步未完成，强行切换流量，就会导致尽管用户可以进行新的购买等动作，但可能会有一段时间的数据是不一致的，那样问题会更加严重。

我们之后增加了一个区分数据修改和数据新增的逻辑，确保在数据未完全完成同步但路由规则逻辑已推送完成
时，即可恢复数据新增的动作，在保障数据一致性的同时，又最大可能地实现可用性。

2015年双11，我们按计划启用了部署在三个城市的四个交易单元，其中一个城市的距离更是在1000km以上，结果大家都看到了，系统完美通过了双11巨大流量的考验，至此也宣告单元化项目在经过三年的改造后，到达了一个成熟阶段。

摘抄一段2015年双11结束后邮件内容：“三年了，我们终于从当初的连单元化是什么意思都搞不清楚，到2013年杭州双单元双活，到2014年杭州、上海双活，到今天的三地多活，成功地将单元化打造为了架构的能力，淘系电商交易业务终于能够在全国范围内任意地点部署，并且能够部署多个。”



与用户地理位置无关的异地多活

淘宝将买家的操作按照买家id，分到不同的单元去。

这里能看出淘宝在国内业务的一个特点，用户所在的地理位置和他的下单操作在哪个机房处理是无关的，并不存在一个就近的关系。例如，如果你的userid取模后落到了深圳机房的区间内，那么即使你在上海，你离上海机房更近，你的下单操作都由深圳机房完成。

为什么这样，这也是和淘宝的业务特点相关，下单是一个比较重的业务逻辑，上面提到过内部有上百次的服务调用，这些服务调用加起来的耗时，要远高于客户端到机房的网络延迟。所以只要保证这些服务调用内部都在一个机房内完成即可，至于客户端在哪里，其实并不重要（我们这里不涉及静态的在CDN中的内容）。

所以在这种异地多活方案里，**切流操作**是与用户的地理位置无关的。它只需要按百分比将一大批userid进行切换即可，并不需要关注他们实际的地理位置在哪。

淘宝异地多活的特点

我们简单总结下淘宝风格的异地多活有哪些特点，以及响应的，对数据库有哪些要求。

需要能随时按比例进行切流。

如果我们采用类似这样的架构，每个单元各有一套数据库，但他们之间的数据毫无重叠，那当需要切流的时候，再去迁移数据吗？

这个显然是不对的，所以这一点实际上要求**每个单元的数据库必须有全量的数据**，这样才有切流的基础。

业务响应时间要求高。

既然每个单元都要有全量的数据，那就涉及到一个问题，如何去做单元之间数据的复制？抛开具体的实现方式不谈（通过binlog也好，通过paxos也好），简单分为同步和异步两种类型。

同步复制的方式（例如PAXOS的LEADER-FOLLOWER这种复制），我们有机会做到RPO=0，但这样数据库的写入响应时间会非常的高（想想我们前面说到的，间距1000公里的机房代表30-60ms的的延迟），除非业务做全面的异步化改造，不然是很难接受如此高的响应时间的。

异步复制的方式（例如使用binlog，或者PAXOS的LEADER-LEARNER间的复制），可以完全不受机房之间距离的影响，响应时间可以做的很低。但相应的代价是， 机房之间数据是一种不一致的状态。这种不一致在计划内的切流数据库层是可以直接解决的，但如果是灾难下（机房挂掉）的切流，单纯数据库层是无法处理这种问题的，这就需要业务层做一些手段（例如对账），来保证数据的一致性。

从淘宝的业务特点来说，响应时间是第一优先级，选择的是异步复制的方式。

机房与用户地理位置无关。这个特点要求数据库只需要提供分片级的切换能力即可，并不需要更细粒度的切换。

参考文献

https://blog.51cto.com/wangxy/1953308