支付宝的高可用与容灾架构演进

2016 年 01 月 20 日 19:21:04 带鱼兄 阅读数：3809 标签： 架构 更多
个人分类： 高可用

持续可用和快速容灾切换的能力，是技术人员追求的极致目标。在架构设计中，容灾设计强调的是系统对外界环境影响具备快速响应能力，节点级别的快速恢复能力，保障系统的持续可用。

去年 12 月 18 日，全球架构师峰会上，阿里巴巴高级系统工程师曾欢（善衡）结合互联网金融业务及系统特性，分享了在支付宝系统架构演进中，每个阶段的高可用和容灾能力建设的解决思路。

高可用和容灾架构的意义

企业服务、云计算、移动互联网领域中，高可用的分布式技术为支撑平台正常运作提供着关键性的技术支撑。从用户角度，特别是作为主要收入来源的企业用户的角度出发，保证业务处理的正确性和服务不中断（高可用性）是支撑用户信心的重要来源。高性能，高可用的分布式架构就成了访问量高峰期时，网站得以成功运维的关键。

在当今信息时代，数据和信息逐渐成为各行各业的业务基础和命脉。当企业因为信息化带来快捷的服务决策和方便管理时，也必须面对着数据丢失的危险。

容灾系统，作为为计算机信息系统提供的一个能应付各种灾难的环境，尤其是计算机犯罪、计算机病毒、掉电、网络/通信失败、硬件/软件错误和人为操作错误等人为灾难时，容灾系统将保证用户数据的安全性（数据容灾），甚至，一个更加完善的容灾系统，还能提供不间断的应用服务（应用容灾）。可以说，容灾系统是数据存储备份的最高层次。

每年的“双 11”、“双 12”都是全球购物者的狂欢节，今年的双 11 有 232 个国家参与进来，成为名副其实的全球疯狂购物节。11 月 11 日，全天的交易额达到 912.17 亿元，其中在移动端交易额占比 68%今年每秒的交易峰值达到 14 万笔，蚂蚁金服旗下的支付宝交易峰值达到 8.59 万笔/秒，这一系列的数字，考验的是支付宝背后强大的 IT 支持能力。而持续可用和快速容灾切换的能力，是支付宝技术人员追求的极致目标。

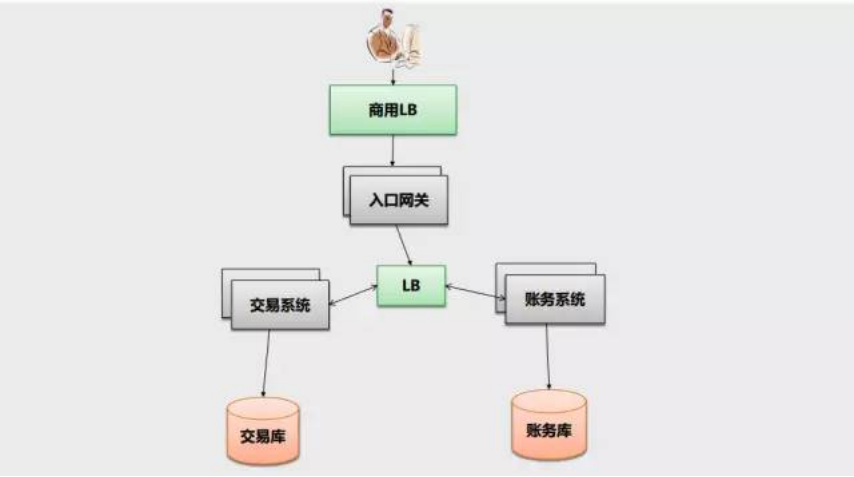
在架构设计中，作为系统高可用性技术的重要组成部分，容灾设计强调的是系统对外界环境影响具备快速响应能力，尤其是当发生灾难性事件并对 IDC 节点产

生影响时，能够具备节点级别的快速恢复能力，保障系统的持续可用。2015 年 12 月 18 日，年度高端技术盛会：“全球架构师峰会——ArchSummit”在北京国际会议中心隆重召开，会上，阿里巴巴高级系统工程师：善衡（曾欢）结合互联网金融业务及系统特性，分享了在支付宝系统架构演进中，每个阶段的高可用和容灾能力建设的解决思路。本文由其演讲内容整理而成。

支付宝的系统架构，其发展历程可以分为清晰的 3 个阶段，每一个阶段都有自己独特的特点和架构上相应的痛点。在每一个阶段的发展过程中，支付宝的技术人员针对不同的问题进行诸多的思考，在解决这些问题的过程中也做了诸多的尝试。

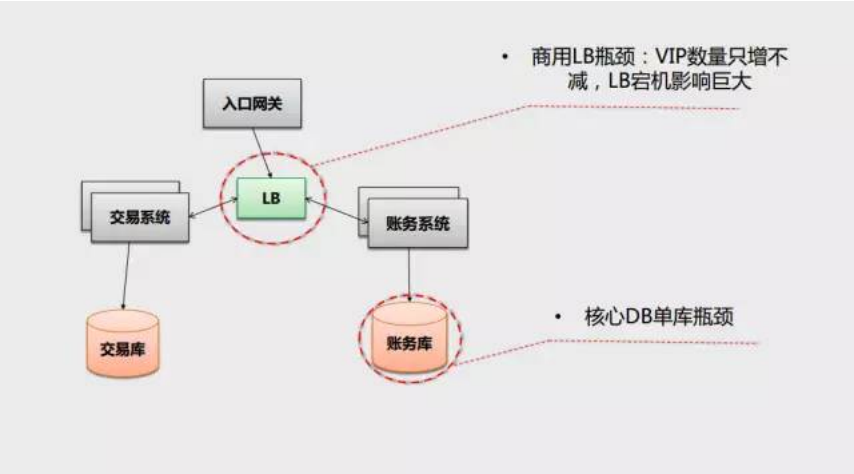
1 纯真：童年时期 2004 年~2011 年

在此阶段，支付宝的系统架构相对比较简化，如图 1 所示，通过商用 LB 让用户的流量进到入口网关系统，支付宝的系统服务暴露也通过商用设备挂在 VIP 下，每个提供服务的应用机器通过 VIP 来进行负载均衡。早期支付宝的核心系统库都在一个数据库上（后期拆为多个数据库），即每个核心系统都只用单独的数据库。在这样一个“物理上多机房，逻辑上单机房”的架构背后，每天的业务量仅仅为数十万级，应用系统也只有数十个，容灾能力相对较低：例如单应用出现问题时无法及时有效地切换、主机和备用机进行切换时，一定程度上会导致业务中断，甚至有时会有不得不进行停机维护的情况，使得整个系统面对数据故障时显得十分被动。



随着业务量的不断增长，该架构发展到 2011 年，出现了一些比较典型的问题。如下图所示：由于系统内部使用的都是 LB 设备，商用 LB 的瓶颈就尤其明显，由于业务的发展累计，VIP 及其上面发布的服务越堆越多，设备如果出现抖动或者宕机会对业务造成严重影响，这即是架构上的单点。第二个问题就是数据库的

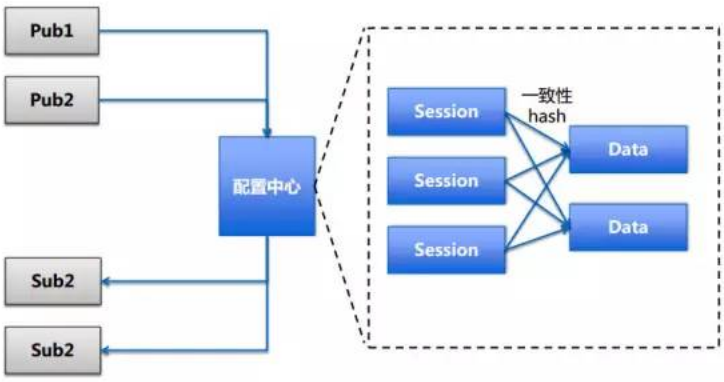
单点瓶颈。随着业务量的不断增加，单一的核心数据库一旦出现异常，比如硬件故障、负载异常等等，进而会导致整个核心链路上的业务都不可用。



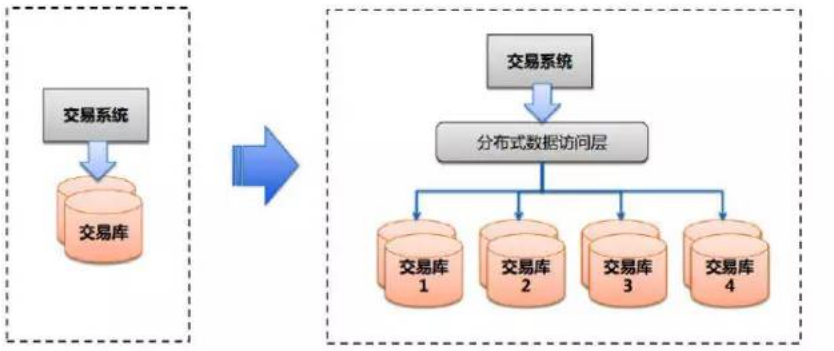
如何消除系统架构当中存在的单点问题，优雅解决未来 1-3 年之间业务量增长（数百万级/天）和机器数量增长（数百个系统），是首先要解决的问题，于是带来了下一代架构革新。

2 懵懂：少年时期 2011 年~2012 年

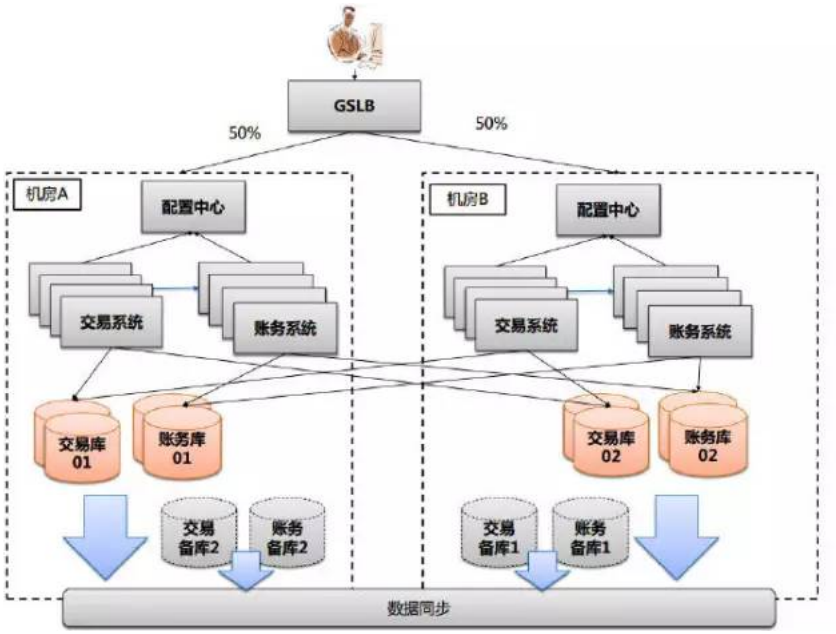
鉴于第一阶段支付宝碰到的这些痛点，在第二个阶段，它将逻辑上的单个机房拆分成为多个机房，通过把硬负载转换成为软负载，以实现分布式的服务调用，如下图所示。下图为基于常见的消费者和生产者模型来构建的业务模型，其中配置中心负责服务注册以及对应服务提供方可用状态变化的通知，从而将信息实时推送到消费方的订阅关系上。值得注意的是，支付宝对原有架构做了一个较大的改进：它将普通的一体化配置中心分拆成两个模块，一个 Session 模块，用于管理消费者和生产者的长连接保持；一个 Data 模块，用于注册服务时存储相关。通过这两个模块的深度解耦，进一步提高整个配置中心的性能。



除此之外，支付宝还做了数据的水平扩展。其实早在 2011 年之前，支付宝就已经开始从事此项工作，根据用户的 UID，将一个交易库水平拆分为多个库，如下图所示。在此期间，需要解决的问题就是“对应用透明”，如何通过“应用无感知”的方式进行用户数据库的拆分，是水平扩展实施时首先要考虑的；其次，如何通过一个数据中间件来管理数据分片的规则，也是数据水平扩展过程中的关键问题。



通过上述两个比较典型的优化，支付宝的整个系统架构如图 5 所示。从应用层面上来说每个机房都是一个节点；从数据层面上，每个机房有特定的几个数据分片在里面。在部署模式上，当支付宝扩张到 3 个或 4 个机房的时候，需要全局考虑数据分片部署，每个机房都有可能不可用，这些备库们应当分散到不同的多个机房里面，从而达到多机房备灾的目的。



这种多机房多活的第二代架构体系，其优势在于：

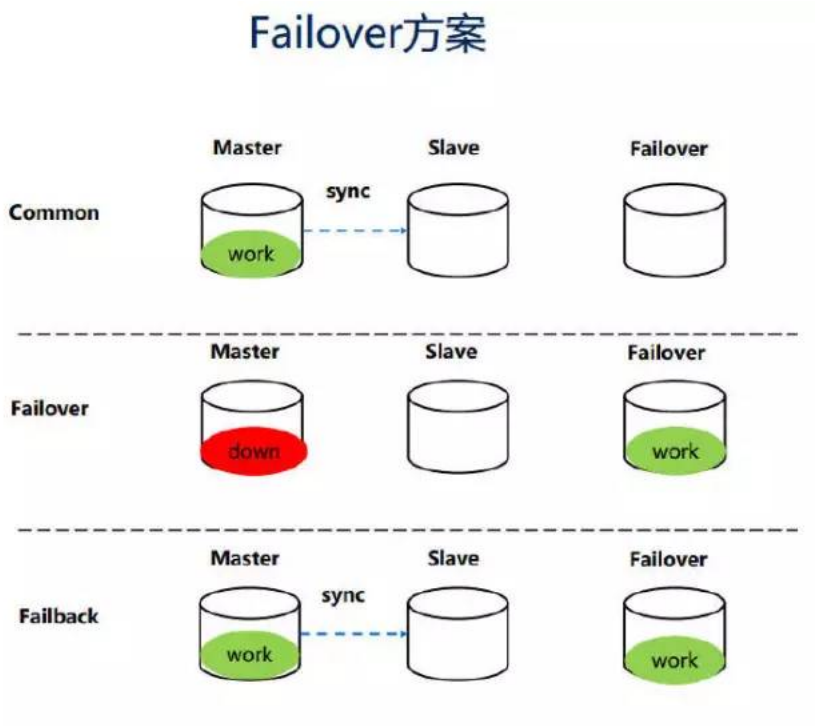
- 进行了数据的水平拆分，从而理论上可以无限扩展数据/资源；
- 应用多机房独立部署，不再存在单个机房影响整体的情况；
- 服务调用机房内隔离，通过软负载的方式去做机房内的服务本地化，不再去依赖另一个机房内相同的服务；
- 相较上一个阶段具有更高、更可靠的可用性。

虽然在此过程中自然解决了上述的单点问题，但仍存在一个问题没有解决。即数据库日常维护时，或因为硬件的故障导致数据库宕机时，支付宝需要进行主备切换，在此过程中业务是有损的状态。一般情况下当 IDC 出现问题的时候，工程师们会通过前端的流量管控系统先把用户的流量从出现异常的机房切换到正常的机房中，然后进行数据库的主备切换，即将备用负载替换主用负载。

在这个过程中，有两个比较大的痛点：

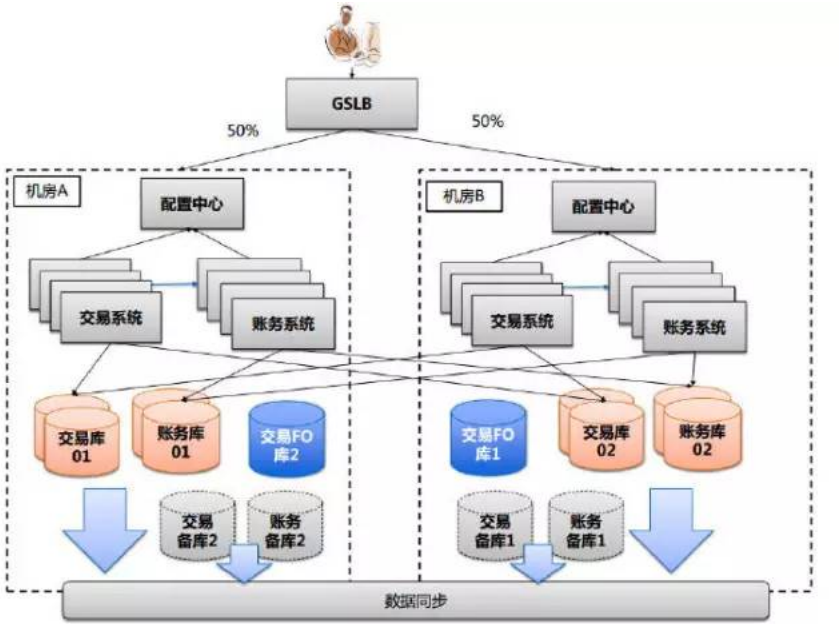
- 主备切换时数据“一致性”的问题，即主备切换时，如何在把影响时间降至最低的情况下，保证数据不被丢失，完完整整地拷贝至备用数据库。
- 主备切换时数据存取不可用导致的业务暂停问题。

一旦数据库发生故障，我们需要进行主备切换时，因为切换过程中的数据不可写，部分用户操作后的状态不对，对用户来说是会担心的。为了解决这个问题，我们制定了一个 Failover 方案。该方案主要通过业务层进行改造，例如针对流水型的业务数据，我们是这么来做的：正常进行数据流量存取的时候，只有主库提供路线服务，主库和备库之间进行正常的数据同步，但是 Failover 库不进行数据同步，正常状态下对业务系统不可见。即正常情况下，没有数据流经过 Failover 库，而且 Failover 库是空的，没有任何历史数据，如下图所示：



一旦故障发生，主库发生宕机，支付宝人员将通过容灾切换将所有数据的读写放置在 FailOver 数据层中进行。因为是实时流水型的数据，所以不会对历史数据产生任何依赖和影响。切换完成后，整个核心链路上的业务就已经全面恢复起来了。通过这种方式，使得支付宝可以将数据库在短短 5 分钟内进行切换，一旦故障解除，随时可以将数据读写重新切换到主存储库上来。

FailOver 方案上线后，支付宝基本上所有的核心业务都基于此进行了方案改造，并针对不同的业务（不仅限于应用层）都会有不同的 FailOver 方案。现在，支付宝在原有的方案基础上再多准备一个 Failover 数据库（如图 8 中蓝色图形所示），与之前提到的容灾设计一样，如果需要进一步保证 Failover 库的可用性，还可以增加 Failover 库的备库。此外 Failover 库需要与主库分开，可以与主库、备库都不放在同一个机房。



3 成熟：青年时期 2012 年~2015 年

通过“多机房多活”的架构改造以及 FailOver 方案的实施，满以为未来三年都无需为 IDC 资源去发愁的支付宝研发团队，在顺利支撑完 2012 年的“双 11”，为下一年做规划时却发现梦想破灭了。因为他们遇到了严峻的新问题：

- DB 连接数不够。由于一个机房中的应用系统就是一个节点，没有分片的概念，仅仅只有数据的分片。用户进入任意应用节点时，系统需要根据用户的 UID 分片查用户应该去往哪个数据库，这时候每个应用节点都会与所有数据库节点保持连接。而传统关系型数据库的连接数是十分宝贵的，当支付宝面对不断增长的用户扩容需求时，因为 DB 连接数资源无法继续扩充，导致应用也不能继续扩容了，不仅无法满足日常增长需求，更别提“双 11”这样短时间爆发式增长的用户需求。

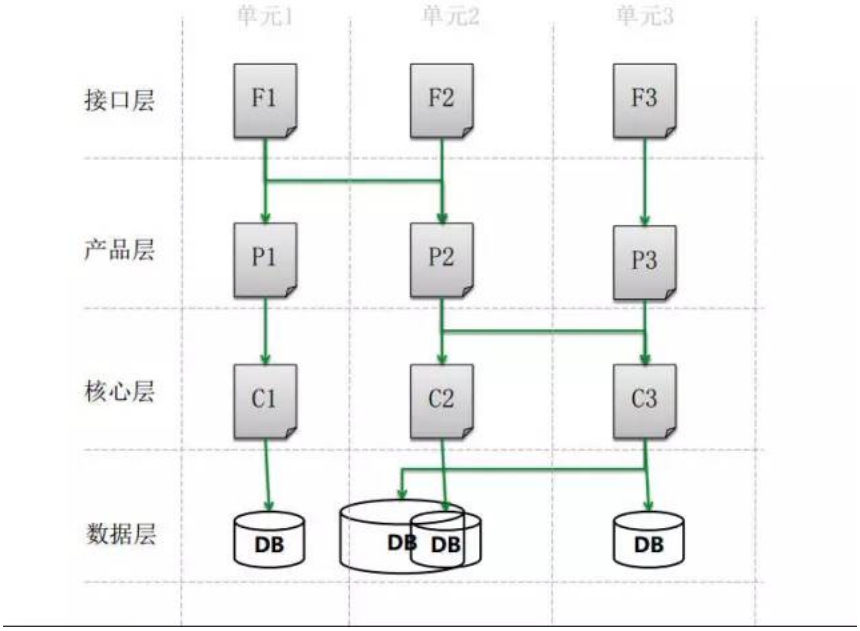
- 城市 IDC 资源限制问题。2012 年夏天，杭州经历了长时间高温天气，由于机房运行的耗电较高，市政为了缓解电力供应压力，下达了一纸通文，随时可能关闭掉支付宝的某些机房供电。

- 机房间高流量问题。由于一个业务请求与后端数据读取之间的比例为 1:N（其中 N 可能是几十甚至上百），在“双 11”高流量的冲击下，跨机房的流量会变得非常的大，因此对于网络带宽和网络质量也有着非常高的要求。

- 跨 IDC 网络延时问题。由于业务请求和后端数据读取 1:N 配比问题，导致同城机房之间距离稍微远一些，就会产生 1、2 毫秒的同城机房延时，被扩大后就会成为困扰用户的网络延时问题。

新的问题进而带来对新一代架构的要求：

- 彻底解决 DB 连接数的问题。
- 彻底解决 IDC 资源限制的问题。需要支付宝走出杭州，不能单单在杭州进行机房的扩张和建设。
- 保证业务的连续性。去减少故障发生的时候对于用户的打扰、对于业务的中断。
- 蓝绿发布。在日常发布时，会通过线下的发布测试，预发布，最终到线上的发布过程。线上发布通常采用的是金丝雀模式，应用分成多组进行发布，每一组的用户不固定，可能会影响到大部分乃至全站的用户。因此支付宝团队希望日常发布时，能够最小限度影响用户（可能是 1%甚至 1‰），新代码上线之后最小力度来验证新代码符合预期。因此需要一种新的发布方式来支持。
- 高可用-异地多活。对于支付宝来说，传统的“两地三中心”要求：机房分属两个不同地区，同城当中两个机房是“双活”，即活跃的主机房，异地机房通过复制数据来做“冷备”，即备用机房。若同城两个机房都出现问题，一旦需要切换异地机房，由于不知道异地冷备机房运行起来是什么情况，因此很难决策。支付宝希望异地的机房也能实时进行流量的读写，以便数据流量可以来回切换，而不用担心在切换后无法知道数据将是什么状态。
- 单元化。基于上述几个问题的思考，支付宝走到了单元化的一步。如图 9 所示，传统的服务化架构下每一次调用服务时，系统都会随机选择一台机器或一个节点完成整次的调用。而单元化之后支付宝可以把任何一个节点固定到一个单独的单元内，即固定的节点对应一条固定的链路，再基于数据分片的方法完成将节点“单元化”的设置。

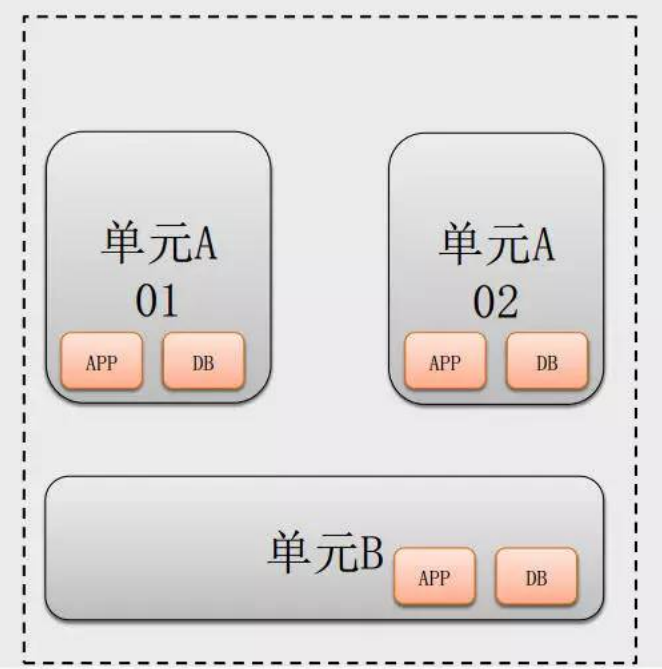


单元化的核心思想包括核心剥离以及长尾独立。核心剥离指的是将核心业务进行剥离；将业务数据按照 UserID 进行拆分，从而实现多机房部署；在此基础上将每一个机房的调用进行封闭式设置；每一个单元中的业务数据无需和其它单元进行同步。长尾独立则针对非核心的应用，这些业务数据并不按照 UID 进行拆分，核心业务并不依赖于长尾应用。

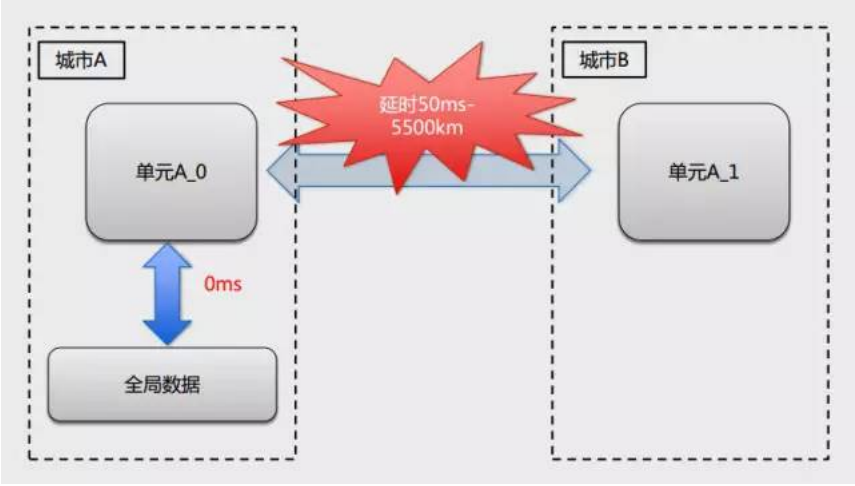
支付宝单元化架构实现的主要思想有两点，如下图所示：

1. 数据水平拆分，即将所有线上核心业务分离开来。由于核心业务集合都能按照用户 ID 去做水平切片，支付宝团队将数据切片后按照机房进行分布，然后通过循序渐进的方式逐步将每个单元之间的调用完整地封闭掉。每一个单元的数据都是经过分片的数据，不需和其它单元进行同步。

2. 上层单元化改造，即将历史的、不能拆分的业务独立出来。2013 年，支付宝实现了两个单元：单元 A 放置核心业务，例如核心链路的支付、交易等；单元 B 放置无法拆分的历史业务，例如某些不重要的业务等。



支付宝单元化架构于 2013 年完成，帮助支付宝顺利支撑过了 2013 年的“双 11”。而 2014 年~2015 年，支付宝一直在尝试解决异地延时的问题：即如果不去对全局数据进行分割或是本地化的话，当把业务单元搬到异地时，由于每一次业务的发生基本上都会依赖全局数据，且会对应多次数据访问，而每一次数据访问都会产生异地所带来的延时，积少成多，时间延迟就会达到秒级，用户体验大幅度下降。基于这个问题的考虑，支付宝研发团队做了一个很大的决定：他们引入了单元 C，通过将无法拆分的全量数据进行全局复制（异地机房之间的复制），以便于支撑核心业务本地调用，进而实现读写分离，将跨城市的调用减少到最小力度。如下图所示。

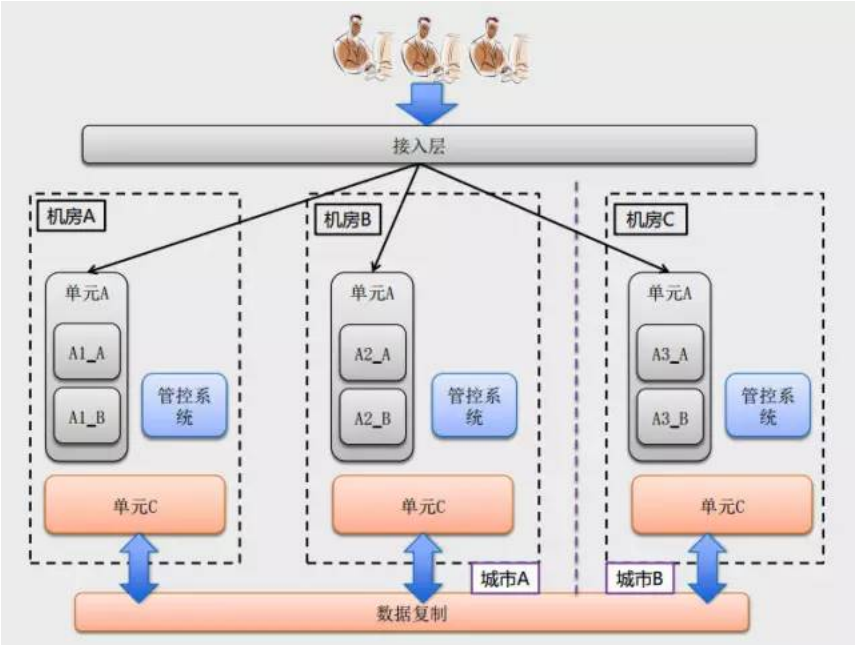


由于数据的写入操作会在每一个数据单元上进行，而单元 C 的数据读取是要求全量的，所以进行这一项改造的时候需要底层的支持，从而解决架构改造时数据一致性和时效性的问题。为了解决这个问题，支付宝团队提出了两种解决方案：

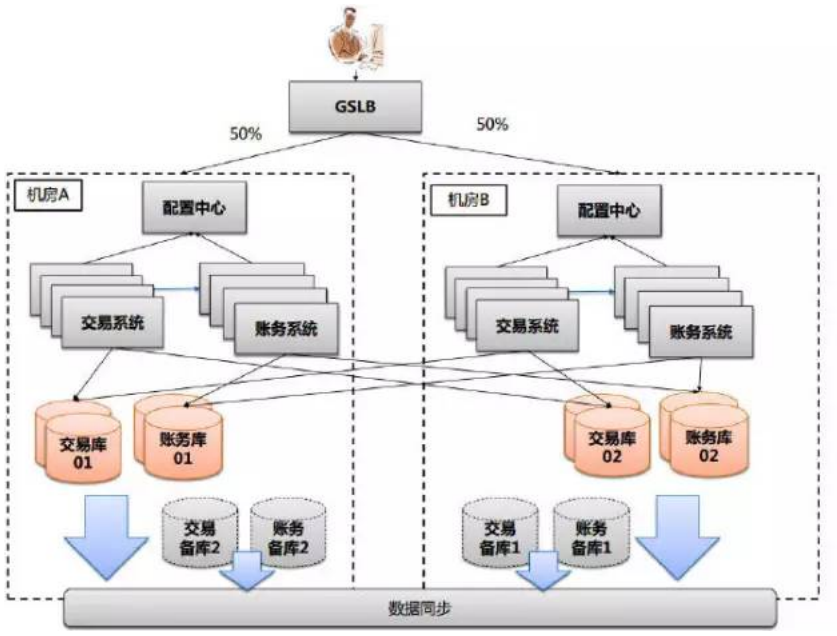
- 基于 DB 同步的数据复制。针对某些对于延时并非十分敏感的业务，单就基于主备同步来做数据的同步。
- 基于消息系统的数据复制。由于异地数据同步比较耗时，对于延时非常敏感的业务来说，支付宝基于可靠的消息系统做一个数据复制（内部称为数据总线），通过它将上层基于应用去做数据的复制，大概时间位于毫秒级。底层 DB 主备同步同时进行。

通过本地化调用，支付宝有效地解决了 DB 连接数问题、机房间流量限制问题、跨 IDC 的延时问题；通过异地的单元部署以及不断的容灾演练和数据切换，支付宝有效地解决了城市 IDC 资源限制和异地 IDC 容灾问题。另外还有一个诉求——“蓝绿发布”，支付宝是这么实现的。

如下图所示，蓝绿发布即每个单元里面分为一个 Blue 组和一个 Green 组，日常模式下两个组各承担 50%的用户流量；发布前将 Green 组中的 50%流量移到 Blue 组中，然后对 Green 进行两批无序发布；新代码发布上去后，将 Blue 组中的流量先切换 1%~2%到 Green 组中进行验证，以最大程度减少对用户的打扰，验证完毕后，再逐步将 100%的流量全部切换至新的 Green 组，重复前面的步骤发布 Blue 组，验证完毕后切回每个组各 50%流量的日常模式。



至此，支付宝单元化全局架构如下图所示。每一个机房单元中有单元 A 和单元 C，单元 A 中按逻辑分为了两个组，单元 C 之间进行全局的数据复制；每个单元中部署了一套分布式的容灾管控系统，使得支付宝能在单个机房故障的时候去做更加快速的应对。



总结

通过单元化的系统配置，使得支付宝整个系统架构具有很强的高可用性和容灾能力。具体来说有三点：

- 更灵活的流量管控，可以实现以更小的力度和更快的速度来进行数据切换。
- 自定义化的数据流量分配。由于每一个数据单元需要多少资源、需要支撑多少交易量可以前期确定，因此支付宝可以按照实际的需求量进行单元维度的扩展。
- 快速恢复的容灾能力。通过单元化之后，不仅使得数据单元内 Blue 组和 Green 组之间可以切换流量，再向上一个级别，单元和单元之间、机房和机房之间、同城内数据中心之间甚至城市和城市之间也可以自如地进行故障发生时的应急切换。不仅使得支付宝实现了“异地多活”的架构能力，更使其顺利经过了 2015 年“双 11”的洗礼。