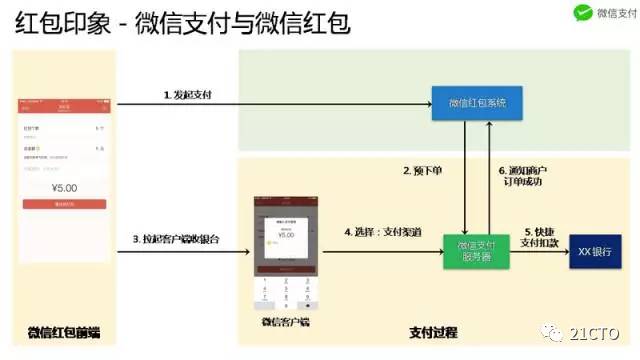
**微信红包与订单处理**

订单在完成支付前可以先落在缓存中，完成支付后再持久化。（订单生成时未存入数据库，待支付成功才落库）



微信红包的业务包含包、发、抢、拆、查询发送红包和收红包数量，其中最关键的步骤是发红包和抢红包。

微信红包是微信支付的商户，微信红包这个商户出售的是钱。发红包用户在微信红包平台使用微信支付购买一份钱，微信红包将钱发放到相对应的微信群。群里的用户抢红包得到微信零钱。这个过程中，微信红包和微信支付之间的关系是商家和第三方支付平台的关系。

微信红包和微信支付之间的交互，与普通商家与微信支付的交互一样，需要经过六个步骤。用户发红包时，进入微信红包下一笔订单，系统记录发红包用户、发红包金额、红包数量和要发送到的用微信群。然后微信红包系统请求微信支付服务器进行下单，用户使用微信支付进行支付。

支付成功后，微信支付后台系统通知微信红包后台系统支付成功结果，微信红包后台系统收到通知后推送微信红包消息到微信群。微信群里用户便可抢红包。这就是微信红包和微信支付的关系以及交互过程。

**微信红包的系统流程**



微信红包系统角度上的流程，业务主流程是包、发、抢、拆四个操作，每个操作包括几个关键步骤。

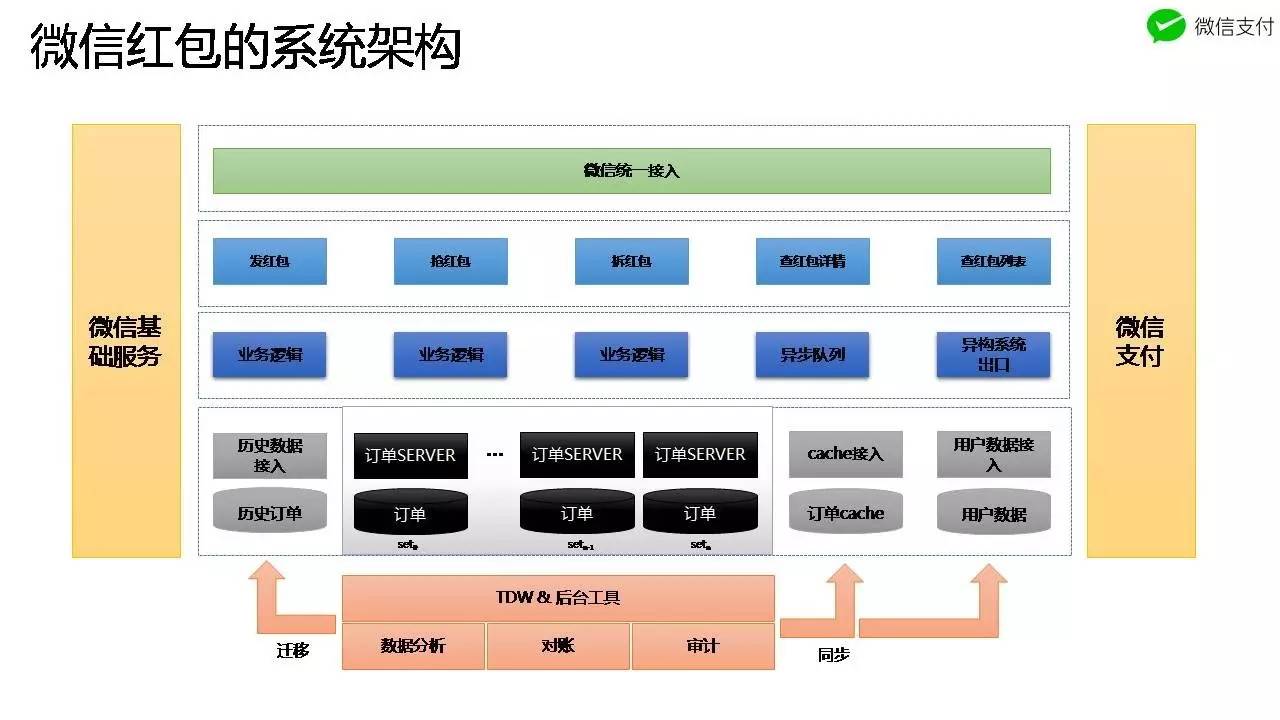
包红包，系统为每个红包分配一个唯一 ID，即红包发送订单号，然后将发红包用户、红包个数、红包数额写入存储，最后去微信支付下单。

发红包，用户使用微信支付完成付款，微信红包后台系统收到微信支付系统的支付成功通知。红包系统将红包发送订单状态更新为用户已支付，并写入用户发红包记录（用户发红包记录，就是微信钱包中，查看到的用户每一年总共发出及收到的红包记录）。最后微信红包后台系统发送微信红包消息到微信群。

抢红包，指微信群里的用户收到微信红包消息后，点开红包消息。这个过程，微信红包后台系统会检查红包是否已被抢完，是否已过期，是否已经抢过。

拆红包是最复杂的业务是操作。包括查询这个红包发送订单，判断用户是否可拆，然后计算本次可拆到的红包金额。然后写入一条抢红包记录。如果把拆红包过程，类比为一个秒杀活动的过程，相当于扣库存与写入秒杀记录的过程。更新库存对应于更新红包发送订单，写入秒杀记录对应于写入这个红包的领取红包记录。另外，还要写入用户整体的红包领取记录。最后请求微信支付系统给拆到红包用户转入零钱，成功后更新抢红包的订单状态为已转账成功。

**微信红包的整体架构**



上图所示，是微信红包系统之总体架构。

包括微信统一接入层，下面是微信红包系统 API，包括发、抢、拆、查红包详情、查红包用户列表。再下面是封装微信红包关键业务的逻辑服务；最下面一层是数据存储层，微信红包最主要的数据是订单数据，包括发红包订单和拆红包订单两部分。业务逻辑和存储服务器之间是数据接入层，它最重要的作用是封装数据库操作的领域逻辑，使得业务逻辑服务不需要感知对 MySQL 的连接管理、性能、容灾等问题。

微信红包数据的访问热度，随着时间流逝会急剧降低，也就是数据的访问时间段非常集中，一般红包发出三天后，99% 的用户不会再去点开这个红包了。因此微信红包系统采取按时间做冷热数据分离，降低数据的存储成本，同时提升了热数据的访问性能。

数据平台用于对红包数据的分析计算，比如朋友圈里的文章，统计从 2016 年 1 月 1 日到 2017 年 1 月一个用户总共抢红包的金额，在全国的排名情况，发红包数最多的城市等。另外一个作用就是对账，红包的订单和微信支付的订单需要对账，以保证最终资金的一致性；订单的数据和订单的 cache 需要做对账，以保证数据的完整性；订单数据和用户的收发记录需要对账，以保证用户列表完整性。

**微信红包系统可用性实践**

**系统可用性影响因素**

系统的可用性影响因素可分成两类，一类计划外，一类计划内。计划外包含很多因素，系统用到的所有东西都可能产生故障，都可能成功影响可用性的因素。从这个角度上来讲，可以说故障是无法避免的，系统的运作一定会产生故障，尤其是服务器有成千上万个的时候。计划内的影响因素，主要有与升级相关、运维相关的操作，以及日常的备份等。这一类影响因素，通过精细地设计方案，是可以避免对可用性造成影响的。

**微信红包系统可用性设计方向**

基于上面两个分析结论，可以总结出微信红包后台系统的可用性的设计方向。就是在不能避免意外故障的情况下，尽可能降低出现意外故障时对可用性的影响。另一方面，绝大多数计划内的日常维护可以通过方案的设计避免影响可用性，其中平行扩容特指关于存储层的平行扩容。

下面从降低故障影响和微信红包系统的平行扩容两方面进行分析。

首先是降低意外故障的影响，重点讲解订单存储层在订单 DB 故障的情况下如何降低对红包系统可用性的影响。

**业务逻辑层 - 部署方案设计**

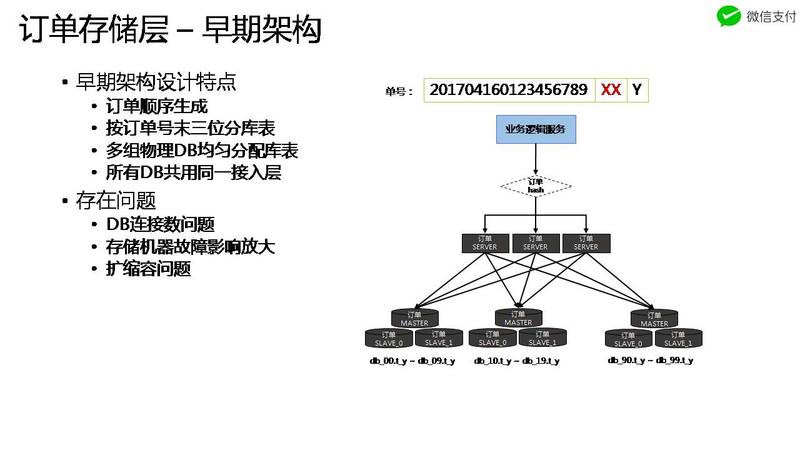
首先是业务逻辑层的部署方案。业务逻辑层是无状态的，微信红包系统的业务逻辑层，部署在两个城市，即两地部署，每一个城市部署至少三个园区，即三个 IDC。并且每个服务需要保证三个 IDC 的部署均衡。另外，三个 IDC 总服务能力需要冗余三分之一，当一个 IDC 出现故障时，服务能力仍然足够。从而达到 IDC 故障不会对可用性产生影响。

**业务逻辑层 - 异步化设计**



第二是异步化设计。如上图所示，微信红包的某些步骤不实时完成也不会影响用户对红包业务可用性的体验。比如拆红包，正常的业务流程很长，但关键步骤只有订单相关的几步。至于转零钱、写红包记录等操作不需要实时。用户抢到红包时，一般不会实时去钱包查看微信零钱，而是在微信群中点开消息查看本次抢到金额和他人抢红包金额。

所以拆红包时只需要从 cache 查询用户是否拆过红包，然后写入拆红包的订单记录，更新发红包订单，其他的操作都可以异步化。当然，不是每个业务都可以进行异步化设计，需要进行业务分析，判断是否存在非关键步骤之外的事情可以将其异步化，并通过异步对账保证最终一致。



接下来是微信红包订单存储设计。上图是 2014 年微信红包存储层的模型。业务逻辑层请求数据层操作时，使用订单号 hash 路由到订单服务器。订单服务器与每一组 MySQL 数据库连接。

微信红包的订单号是在发红包时系统生成唯一标识，使用序列号服务生成唯一 ID，后面拼接三位微信红包的订单分库表的标识。所以，总共可以分一百个逻辑库，每个逻辑库含有十张表。一百个逻辑库均匀地分布到十组物理 DB，每组 DB 存十个逻辑库。

这个架构的最大问题是，一组 DB 故障时，会影响其他 DB。

在2014-2015 年期间，微信红包量涨得特别快，扩容速度跟不上业务增长速度。一组 DB 的性能出现瓶颈时，数据操作变慢, 拆红包的事务操作在 MYSQL 排队等待。由于所有十组 DB 机器与所有的订单服务器连接，导致所有的订单 SERVER 都被拖住，从而影响红包整体的可用性。这个架构的另一个问题是扩容不方便，后面会介绍。

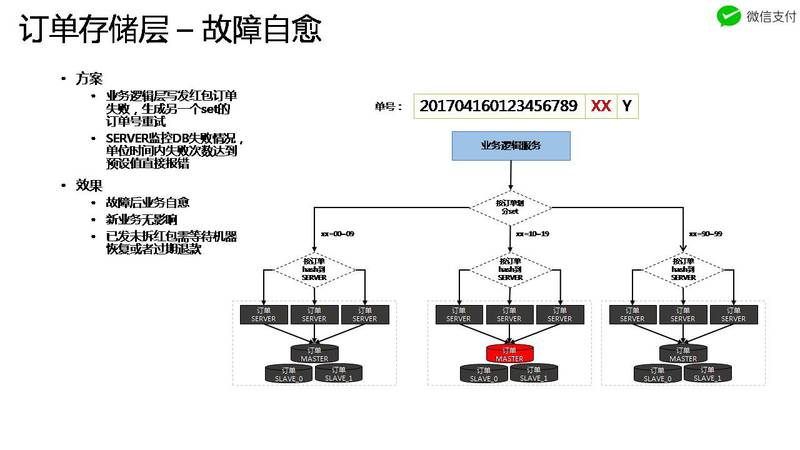
为解决 DB 间的相互影响，需要将 DB 间相互隔离，订单存储层 SET 化。SET 化指订单 DB 和订单接入 Server 垂直 Stick 一起。业务逻辑层访问订单时，根据订单倒数第二、三位数字找到所属订单 SET，一个 SET 的请求不能路由到其他 SET。

找到对应的订单接入服务器之后，在服务器内的多个进程中找到指定进程，让同个红包的所有拆请求串行化。当一组 DB 出现故障，只会影响该组 DB 对应的 Server。

这里有一个问题，DB 故障拖住某些订单 Server，会不会也拖住更上层业务逻辑服务？业务逻辑层为什么不一起 SET 化？业务逻辑层承载了用户维度相关的业务操作，不可以按照订单的维度分业务逻辑，例如务逻辑层会请求用户的头像、昵称等，如果继续按照订单分业务逻辑，会导致跨地域调用。

微信红包系统采取的方案是，在订单服务端增加快速拒绝服务的能力。服务器主动监控 DB 的性能情况，DB 性能下降、自身的 CPU 使用升高，或者发现其他的监控维度超标时，订单 SERVER 直接向上层报错，不再去访问 DB，以此保证业务逻辑层的可用性。

一组 DB 故障不会影响整个系统的可用性。有影响的，只有十分之一，若扩成 100 组，影响便只有一百分之一。所以通过 SET 化得到的好处是，控制 DB 连接数、隔离故障影响和分流并发。



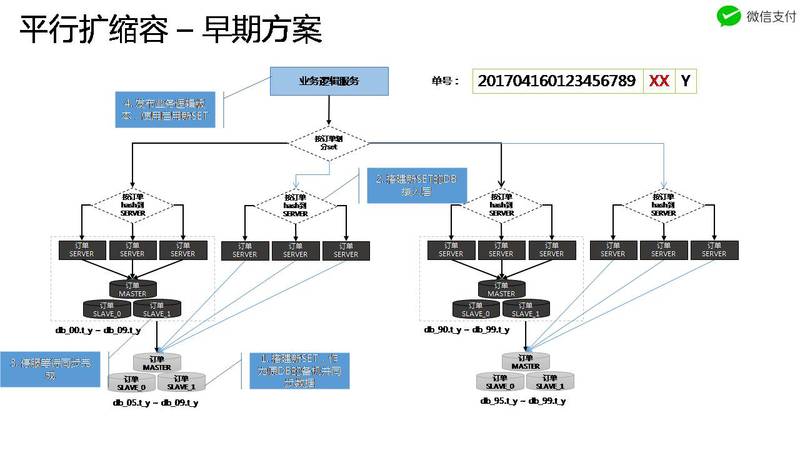
完成 SET 化之后，DB 故障仍对业务有十分之一影响，那么这十分之一该怎么解决？通过对系统进行研究分析之后，发现 DB 可以做到故障自愈。

正如上图所示，所设尾号 90-99 的 SET 故障时，如果业务逻辑服务后续不再生成属于这个 SET 的订单，那后续的业务就可以逐渐恢复。

也就是在发生故障时，业务逻辑层发布一个版本，屏蔽故障号段的单号生成，就可以恢复业务。进一步想，除了人为发版本，有没有方法可以让 DB 故障时自动恢复？在 DB 故障导致业务失败时，业务逻辑层可获取到故障 DB 的号段，在发红包时，将这些故障的号段，换一个可用的号段就可恢复业务。订单号除了最后三位，前面的部分已能保证该红包唯一性，后面的数字只代表着分库表信息，故障时只需要将最后三位换另外一个 SET 便可自动恢复。

完成这个设计后，即使 DB 出现故障，业务的可用性也不会有影响。这里还有一点，新的发红包请求可避免 DB 故障的影响，但那些故障之前已发出未被领取的红包，红包消息已发送到微信群，单号已确定，拆红包时还是失败。对这种情况，由于不会有增量，采用正常的主备切换解决即可。

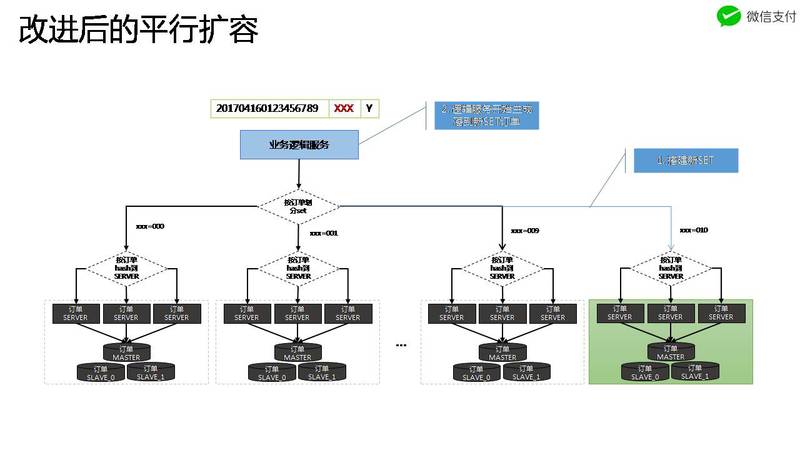
**平行扩缩容设计**



上图是微信红包早期的扩缩容方式。这个扩容方式，对扩容的机器数有限制。前面讲到，红包系统按红包单号后面两个数字分多 SET，为了使扩容后数据保持均衡，扩容只能由 10 组 DB 扩容到 20 组、50 组或者 100 组。另外，这个扩容方式，过程也比较复杂。首先，数据要先从旧数据库同步复制到新扩容的 DB，然后部署 DB 的接入 Server，最后在凌晨业务低峰时停服扩容。

这个扩容方式的复杂性，根本原因是数据需要从旧 SET 迁到新 SET。如果新产生数据与旧数据没关系，那么就可以省掉这部分的迁移动作，不需停服输。分析发现，需要把旧数据迁出来的原因是订单号段 00-99 已全部被用，每个物理数据库包含了 10 个逻辑库。如果将订单号重新设计，预留三位空间，三位数字每一个代表独立的物理 DB，原来 10 组 DB 分别为 000-009 号段。

这种设计，缩容时，比如要缩掉 000 这组，只需在业务逻辑服务上不生成订单号为 000 的红包订单。扩容时，比如扩为 11 组，只需多生成 010 的订单号，这个数据便自动写入新 DB。当然，缩容需要一个前提条件，也就是冷热分离，缩容后数据变为冷数据，可下线热数据机器。以上就是红包的平行扩缩容方案。



**存储层的高可用设计**

读写分离

写请求需要在DB主机上，实时读也需要走主机。有大量对延时不那么敏感，又影响性能的查询，完全可以放到DB从机。读写分离策略是MySQL分布式的入门，简洁地提高了系统容量。

水平切分

数据的水平切分，实质就是分库分表；选取一张数据表按照主要纬度把数据拆分开。实现存储层的平行扩展。有效降低了单台数据库机器的负载，也减小了服务不可用的可能性。单台数据库宕机只会导致部分数据不能访问。主要需要考虑路由规则的选定，方便扩缩容以及数据的均衡分布。

垂直切分

数据表除了水平切分，行内数据可以按属性进一步分开。核心表只保留最关键的字段，保证数据文件短小紧凑。以红包为例，昵称和祝福语这类较长的信息，不属于核心数据，完全可以切分到别的机器上，进一步提升核心数据库的容量。不同数据适合的存储类型也不一样，这类重复率高的长字符串更适合NoSQL存储，对存储空间和性能都是节约极大。

空间换时间

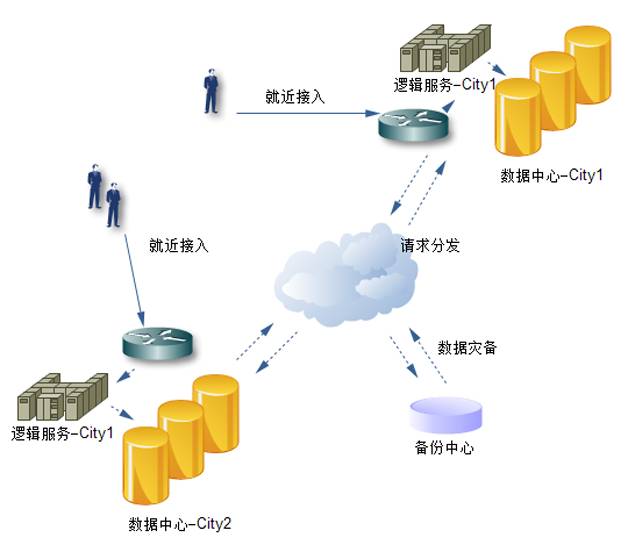
按不同纬度组织表，比如按订单属性和用户属性进行组织；适应不同的请求场景，避免复杂的查询。不同纬度的表可以通过对账对齐，非核心表可以适当冗余，减少多次请求。

锁的优化

多人争抢红包通过数据库事物来保证，必然存在竞争MySQL行锁。核心事物必须尽量精简，避免死锁。同一个订单的所有请求，尽量在逻辑层进程预排队后通过一个连接发送请求到数据库。

冷热分离

核心数据库存放高频数据，其他数据可以定时移到成本低的冷数据库中。



这样可以为核心数据库使用最好的SSD设备，快速设备容量较小较贵，不可能在全量数据上使用。同时可以保证数据表的容量不会一直积累，大表也会导致性能下降。

# 红包算法

随机，额度在0.01和剩余平均值\*2之间。

例如：发100块钱，总共10个红包，那么平均值是10块钱一个，那么发出来的红包的额度在0.01元～20元之间波动。

当前面3个红包总共被领了40块钱时，剩下60块钱，总共7个红包，那么这7个红包的额度在：0.01～（60/7\*2）=17.14之间。

注意：这里的算法是每被抢一个后，剩下的会再次执行上面的这样的算法（Tim老师也觉得上述算法太复杂，不知基于什么样的考虑）。

这样算下去，会超过最开始的全部金额，因此到了最后面如果不够这么算，那么会采取如下算法：保证剩余用户能拿到最低1分钱即可。

如果前面的人手气不好，那么后面的余额越多，红包额度也就越多，因此实际概率一样的。