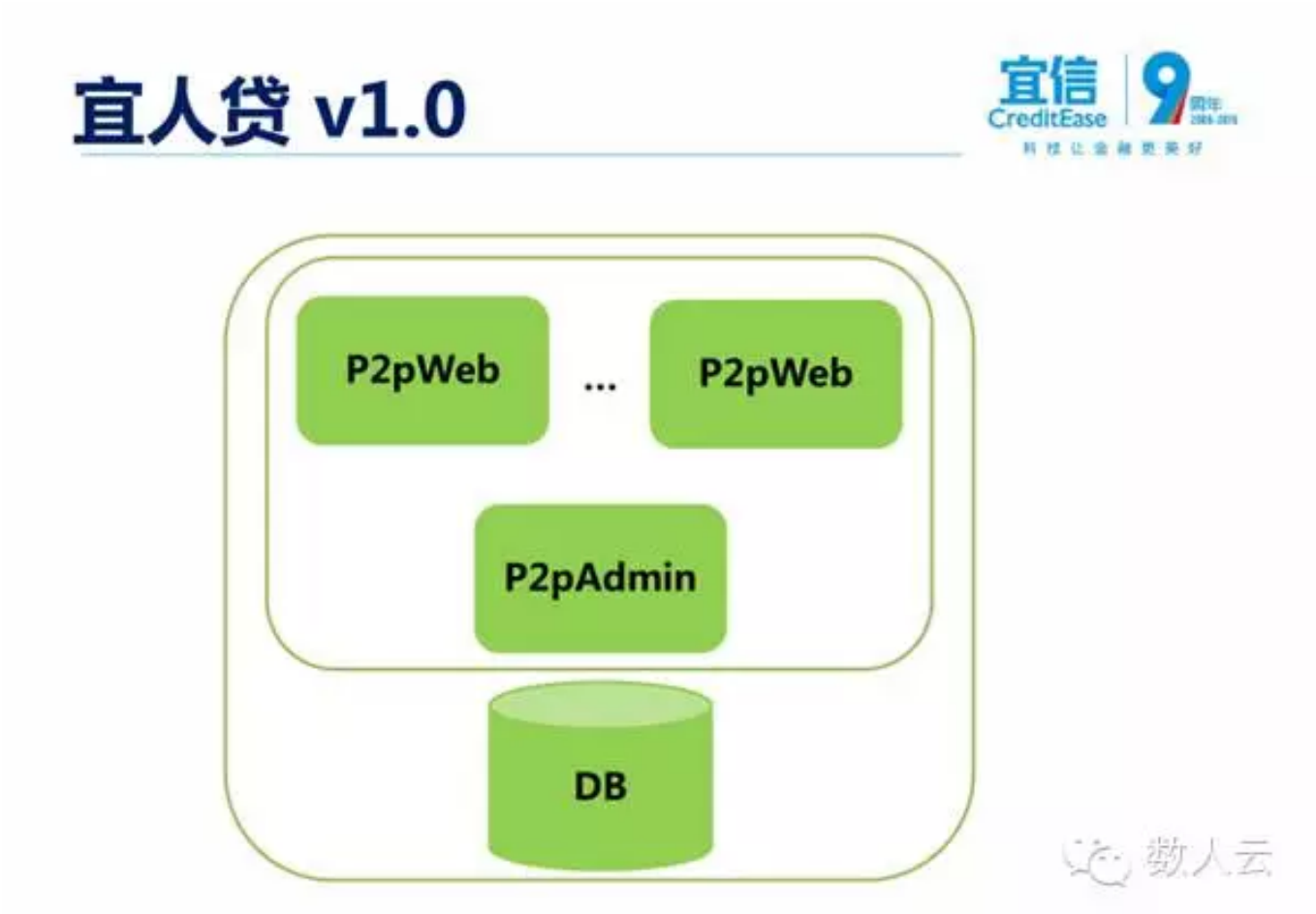
宜人贷系统架构——高并发下的进化之路

优云数智 2016-01-15

演讲嘉宾：宜人贷架构师孙军，拥有10 年的 Java 开发经验，先后在人民银行、1 号店、人人网、当当网从事软件开发与技术架构工作。本次分享以宜人贷的系统迭代发展过程为主，着重介绍系统发展过程中遇到的实际问题和解决的办法，并重点介绍宜人贷理财系统的高并发解决方案。

宜人贷系统版本的迭代

1.0 版本——简单的烦恼



迭代之前宜人贷的系统，其实就是一个前台，一个后台，一个 DB ，前台采用的是多机部署的方式。软件层也是跟最传统的软件一样分三层，第一层是 Controller ，第二层是 Service ，第三层是 DAO 。显然这个系统并不适合互联网，有一些难以避免的问题。首先当用户过万，在线用户上千的时候，这样的部署方式会产

生一些瓶颈，包括服务器和数据库两方面。第二个就是团队规模变大，所有开发人员集中开发同一个系统，冲突严重。

1.5 版本——“吃大补”试试！

针对上面的问题我们做了一些修改，我把它定义成“吃大补”。吃大补通常有一个很明显的特点，就是立马见效，但是副作用也很大。

首先，我们在宜人贷的页面层更加关注性能的提升，比如说使用浏览器缓存，压缩传输，页面都经过了 YSlow 的优化，链路层增加了 CDN ，做了静态化甚至反向代理，这样可以抵挡 80% 的流量。应用服务器与数据库层增加了一个缓存集群，这个缓存集群基本上又可以挡掉 80% 流量，最后系统层按照业务垂直拆分成多个系统。数据库也有一些变化，开始只是一台主机，一台数据库，现在变成了主从，甚至一主多从。用户可以撑到过百万，在线用户上万。即便如此，我们的制约因素依然在数据库，优化的两层虽然挡掉了大约 95% 的流量，但是业务发展依然超过了数据库所能承受的负载，所以数据库依然是一个很大的瓶颈。

第二个问题就是团队划分，其实每个团队都做自己的系统，但是大家仍然使用同一个数据库，这个时候对于设计和修改数据库，都非常麻烦。甚至每次都要问一下其他团队，我这么改行不行，对你有什么影响等等。

第三个问题也非常棘手，大量使用了缓存，数据的时效性和一致性的问题越来越严重。

2.0 版本——“开小灶”精细化

为了解决 1.5 版本存在的问题，我们需要做精细化的优化，我把它定义成开小灶。首先，合理规划数据归属、优化查询效率、缩短数据库事务时间；其次，分系统，每个系统用固定的表。我们每天都要做的事，就是让运维找出线上最慢的 SQL 有哪些，对它们做优化。第三，做去事务，或者尽可能地缩短事务的时间。

然后开始关注代码质量，提高执行效率，并且开始关注并发问题。用户达到这个量的时候，就会有用户帮我们测试并发问题。举一个例子，同一个用户用同一个帐户登录了两个客户端，他同时点取现，这个时候如果程序处理的不好，很有可能让他提现两次。

最后，要区分强一致性与最终一致性的请求，合理使用缓存与读写分离来解决这些问题。

2.0 解决了很多性能问题，但还是会有新的问题——系统越来越多，系统间依赖关系变得复杂，这个时候很容易出现 A 调 B， B 调 C， C 再调回 A 的循环调用。第二个是系统间互相调用增多，上游系统压垮下游系统；第三个也是非常头疼的问题，系统很多，查找线上问题变得越来越困难——试想一下多个系统部署到很多机器上，想找一个线上的问题，通过日志的形式会非常难查。

所以在这个基础上我们做了几件事。一是限流。限流通常基于两点：最大活动线程数和每秒运行次数；活动最大线程数适合于高消耗的任务，每秒运行次数适合于低消耗的任务。第二，我建议在这个时期尽可能统一内部系统间的返回值，返回值中一定要记录返回状态（业务正常、业务异常、程序异常）和错误说明；第三，可重用 RPC 框架或在原框架基础上继续开发完成限流工作。

再说一下关于查找日志的问题。图中为宜人贷日志系统部署框架，最左侧的是我们的业务系统，在业务系统上把日志收集到 Kafka 队列，然后把 Kafka 队列日志放到 ES 集群做索引，最终采用 Kibana 和我们自己研发的日志查询系统去查看日志，这样日志被集中到一个点后会更便于查找。

关于软件方面，宜人贷统一使用 SLF4J+Logback 来输出日志，然后业务系统间实现日志串联，所有服务端和客户端之间都隐含地传递一些参数，这些参数会随着调用链一步一步往下传，通过 AOP 来实现。日志串联需要传递哪些参数，或者日志中到底要打哪些参数呢？第一个是时间，这个时间应该到毫秒级，第二个是流水号，流水号就是每次请求生成的一个唯一的值。然后是用户 Session 、设备号、调用者时间（ APP 使用手机本地时间）、本机 IP 、客户端 IP 、用户真实 IP 、跨越系统次数——如果我们发现了一个错误，根据错误日志可以找到流水号，再通过流水号可以到日志查询平台查询出这次请求途径的所有系统和每个系统对这次请求的日志。有了这些找问题就非常容易。

做到 2.0 之后，宜人贷的网站基本能支撑中大型网站的规模，短时间内不会有太多的性能问题了，但是我们依然会继续往下走，进一步提升系统版本。

3.0 版本——拆分做服务化

3.0 总结下来就是要做服务化，通俗一点说就是拆分，包括业务上的垂直拆分，以及垂直拆分基础上的系统之上的水平拆分，那么服务化要怎么做呢？

首先，做业务拆分的时候，可以按照基础服务和业务服务先做一个大的服务拆分，然后基础服务又包括无业务型的基础服务和有业务型的基础服务，无业务型的系统非常明显跟其他的系统没有太大的关系。而业务型基础服务跟业务之间的关系很小，基本上跟业务系统之间的关联关系仅限于主键和外键的关联关系。

宜人贷可以天然地拆卸分成两大系统，一个是借款业务，一个是理财业务，借款业务可以拆分成后台、Web 、合作渠道等，这个系统之下会有一个基础服务，就是提供一些基础服务和接口的一层系统。而基础服务又拆成了两部分，一个是基础服务的进件，一个是基础服务的贷后。在拆分过程中我们又发现一个问题，理财和借款有两个业务怎么拆都拆不开，就是撮合业务和债券关系，这种拆不开的可以单独再提成一个系统来提供服务。

拆分系统看起来好像很容易，但是实际操作问题会很多。拆分的办法我总结了如下几个：

第一，适当冗余，冗余可以确保数据库依然可以进行关联查询。大部分重构过程并不是做一个全新的系统，而是在原来系统之上进行修改，这个时候可以做一些冗余，避免修改代码。

第二，数据复制，但必须保证数据归属系统有修改和发起复制的权限。这个比较适合于上文说的全局配置，比如说基本上所有公司都会有几张表，记录了全国的省市区县，这些在每个系统中都会用，不一定每个系统都以接口的形式调用它，可以在每个系统里面都冗余一份数据。

第三，小技巧——如何验证数据库，并不一定非把它拆分成两个物理的数据库来验证，可以一个数据库上建两个帐号，这两个帐号分别的权限指向拆分之后的表，这样就可以通过帐号来直接验证拆分效果。

第四，提前规划服务，拆分之前确定一下服务类型是读多还是写多的服务，是快请求还是慢请求服务，不同服务需要分开部署。

最后，同一数据不能由超过一个以上的系统控制，同一系统不能由超过一个以上的团队负责。

4.0 版本——云的展望

做到以上几点， 3.0 版本已经做的差不多了，但是后面宜人贷依然还有很多要做的， 4.0 版本是不是要做云平台，异地部署的方案，表很大的时候是不是要做垂直拆分，去 IOE 或者使用 Docker 快速部署等等这些，这些其实都是我们做 4.0 或者 5.0 将来要考虑的事情。

宜人贷理财系统的优化

合理预估流量——强一致与最终一致

图中这三个界面分别为首页、列页，详情页。

在做优化之前，首先要合理预估流量，常用方法有下面两个。

评估方法一：平日 PV / 热度时间；

评估方法二：热度时间内在线用户数 * 平均每人操作次数 / 热度时间。以宜人贷理财端为例，假设在高峰时期有 N 万人，然后平均做 M 次操作，在 R 分钟左右基本上就把所有的债券抢光，计算出来大概是 N * M / R 万次 / 秒。

预估完以后要做更细的预估，区分什么是强一致，什么是最终一致，这两个流量分别是多少。强一致性要求请求的数据必须是当时最准确的数据，这个数据不能用读写分离或缓存。最终一致性的数据时效性没有那么

高，只要最后的结果是正确的就可以。

假设这 M 次操作包含：注册、注册验证码、登录、解锁手势密码、首页、浏览产品列表等等这些操作，这里面其中有一些操作，比如说产品余额、生成订单、支付短信、付款，这些都是强一致的要求。

针对最终一致的方案非常简单，增加机器就可以解决，实时性较高的可以直接使用数据库的读写分离，如果使用 cache 的话，可以缩短 cache 时间；实时性较低的应当使用较长时间的 cache 。

强一致性的流量处理方案，总的来说就是加锁，可以使用数据库的锁，也可以使用 ZK （ Zookeeper ）这样的分布式锁，或者直接使用队列，因为队列总得来说也是一种锁。如果使用数据库的锁，基本上可以支持到并发在 2000 次每秒上下。使用数据库的锁来处理并发，第一个方法就是有事务的处理并发。先开启事务，加锁共享资源，然后再更新共享资源，最后再查询一次共享资源，然后判断一下结果。假如说这个结果是成立的，就直接继续执行，假如说这个结果是不成立的，直接回滚事务。第二个方法就是无事务的处理并发，在数据库 SQL 的 where 条件加上判断条件，如果 update 条数为 1 则更新成功，如果为 0 则更新失败，这时需要用写代码的形式回滚数据。

如果流量依然承受不住该怎么办?

做到这些其实已经能够承受非常大的流量，但是业务可能继续发展，还承受不住怎么办呢?

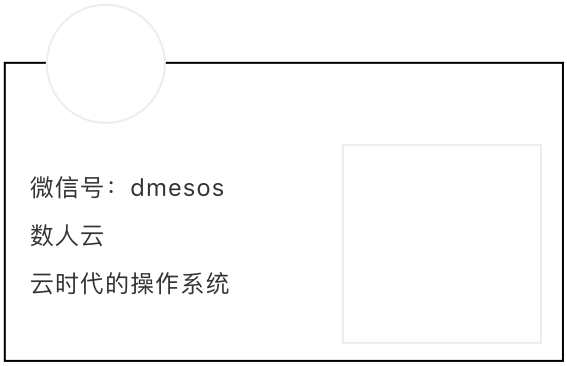
首先的一个原则就是，没有任何一个分布式算法适合并发操作，最好的方法就是单点并排队进行处理。

第二，单点并发过大，使用合适的方式拆分锁的粒度。

第三，增加降级需求，不影响用户正常使用情况下可以适当降低服务质量。适当修改需求、适当增加用户等待结果的时间；如果让用户多等一倍的时间，可能就能承受之前两倍的并发，这个可以在交互上优化，让用户有更好的体验。

最后，适当调整运营策略，分散用户的集中活跃时间。

以上为数人云“高并发”活动嘉宾演讲实录



阅读原文