

架构师

ARCHITECT



推荐文章 | Article

微服务迁移实践

一号专车系统重构细节回顾

京东前端：三级列表页持续架构优化

观点 | Opinion

怎样才能叫高级程序员

专题 | Topic

怎样打造分布式数据库

10亿级流数据交互查询

为什么选择VoltDB



卷首语：谈大数据存储

创业公司在业务上做一个支付模块的时候，首先要解决的是，如何建造一个高效、稳定、安全的数据库。而金融领域要求数据具有强一致性、系统具有高性能，在此基础上，如果再考虑上大数据，那问题就更复杂了。这里结合金融系统的数据库的研发经验，谈谈我眼中的金融大数据存储策略。

数据库的功能要考虑编程实现吗

如果不用数据库，可以通过自己编写程序实现吗？对应数据库都提供的五类功能，进行一一分析。

1. 数据结构：表定义。【类、结构体都可以】
2. 固化：数据落袋为安。【序列化成文件，读时反序列化】
3. RPC：提供远程访问接口。【本地或远程函数调用即可】
4. 锁：保证并发情况下数据的完整和安全。【程序里面锁机制很多，同步块、互斥变量等】

5. 计算：各种联合，聚合函数(sum、count、avg等)和存储过程

【自写逻辑，编写各种函数】

确实有一些数据比如树、图等非二维结构数据，用程序比用数据库，操作起来会更方便。但大部分情况下，完全自主实现上述五点难度很大，如果考虑上大数据下分布式部署、负载均衡、灰度发布、分布式锁等场景那就更复杂了。所以数据库的出现就是为了解决这些通用问题的。

随着数据量的增长，数据库的单机存储和计算能力也都会遇见瓶颈。如果应用重度使用联合，聚合函数和存储过程。分库分表的难度会非常大，需要使用非常复杂的数据库集群或中间件；性能还会大打折扣。所以我推荐轻度使用数据库，侧重数据结构和固化，不要让数据库做重度计算。锁机制最好也拿到外部去实现。

微服务架构下怎样管理好数据

数据库层层分解有这么几个维度：库、表、行。我们把数据的权限细化到表这个级别。如果仅有一个微服务模块有增加修改删除的权限，那这个模块就是这张表的主人。主人要负责维护好这张表，主人不要太多，多了要不然谁都不管，要不然就是互相打架。跨微服务模块读危害小于写，但也是不建议的，因为主人修改了表结构。读就会有问题，建议还是通过接口来。以上说的理想情况，实际场景请斟酌使用。

如果多个微服务模块同时操作一张表的一条记录，分布式锁的问题就不可避免了。除了使用悲观锁，还可以使用版本控制。为表增加一个状态字段，每个模块只操作符合自己状态要求的行。比如订单表的订单状态，入库模块是初始化状态，交易检查模块只操作初始化状态的行。交易拆分模块只操作检查完成状态的行，通过状态来实现数据的隔离。即是版本控

制和状态的前置判断，也实现了有限状态机。涉及业务需要同时占有两个以上的独占资源时，会出现死锁。避免死锁的关键点是程序要按照统一的规则去给资源加锁。

数据更新操作可以采用先删除再增加的方法。增加操作是顺序 IO，一直在表的末尾追加。比起直接更新，这样做的好处是：性能优；不需考虑并发问题；可保存所有的历史版本。不过，这样做的弊端是数据量会成倍增长，程序实现逻辑也会更复杂。

如何控制数据的规模

一个系统的数据，会不断累计，变得越来越多。数据增长有快有慢，但不会凭空减小。考虑到数据节点的单机性能恒定，系统的整体性能会慢慢降低。如何控制单机数据的规模？如何平滑的增加存储计算节点？目前用的比较多的分库分表规则有两种：一种基于规则，一种基于配置。

比如根据时间规则，分割数据为日表、月表、在线库、离线库、历史库。如果时间不敏感根据主键取模或 hash，也是一个办法。由于规则是一定的，每次读写调用算法就知道数据的片区了。基于配置就是要把规则存起来，每次读写先要去读规则，这样才知道数据的片区。配置更灵活可变化，规则更简单，每种都有自己的场景。

因为篇幅的问题，有些话题我就不展开讲了，下次有机会我们详聊。

付钱拉 CTO 史晓慕

CONTENTS / 目录

专题 | Topic

怎样打造一个分布式数据库

10 亿级流数据交互查询，为什么抛弃 MySQL 选择 VoltDB

推荐文章 | Article

从单体架构迁移到微服务，8 个关键的思考、实践和经验

50 天 10 万行代码，一号专车系统重构细节回顾

京东前端：三级列表页持续架构优化

观点 | Opinion

怎样才能叫高级程序员



架构师

2016 年 9 月刊

本期主编 穆 寰

流程编辑 丁晓昀

发行人 霍泰稳

提供反馈 feedback@cn.infoq.com

商务合作 sales@cn.infoq.com

内容合作 editors@cn.infoq.com

QCon

全球软件开发大会2016

[上海站]

主办方 **Geekbang** 极客邦科技 **InfoQ**



扫码关注QCon公众号



《手机淘宝的高效研发支撑体系建设之路》

周延婷 (古雪)

阿里巴巴高级项目管理专家



《蘑菇街前后端分离实践》

陈辉 (沉辉)

蘑菇街商品团队开发工程师



《3D技术在电商网站中的应用和发展》

黄如华 (仙羽)

阿里巴巴前端开发专家



《人人车线下核心业务系统发展之路：浅谈O2O互联网创业公司，如何高效打造线下团队业务系统》

徐章健

人人车业务平台总架构师



《比Buck更快——蚂蚁聚宝Android秒级编译方案Freeline》

何嘉文 (弦影)

蚂蚁金服高级技术专家，蚂蚁聚宝终端基础架构负责人



《携程多机房微服务灰度发布》

王潇俊

携程网系统研发部总监



《CQRS&Event Sourcing在点融网支付系统中的应用与实践》

姚晨

点融网首席软件工程师



《基于云平台的Docker多租户安全》

汤志敏

阿里云高级专家



《聚划算无线的插件化生存之道》

金津 (敛心)

阿里巴巴无线技术专家



《构建React同构应用及优化》

蒋吉麟

eBay分析平台基础架构部门高级软件工程师



《Mesos在教育课件云中的应用》

黄凯

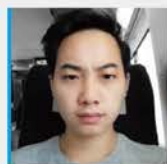
沪江Java架构师



《视频直播主观质量的几个细节》

姚冬

欢聚时代直播技术架构师



《大数据与电商四大核心要素》

杨光耀

1号店供应链优化部负责人



《中美技术团队管理的取长补短》

杨骏

新达达CTO & 联合创始人



《携程redis多数据中心实践》

孟文超

携程资深工程师

9折优惠中 截至9月25日
团购享更多优惠

2016年10月20~22日
上海·宝华万豪酒店

怎样打造一个分布式数据库

作者 刘奇

在技术方面，我自己热衷于 Open Source，写了很多 Open Source 的东西，擅长的是 Infrastructure 领域。Infrastructure 领域现在范围很广，比如说很典型的分布式 Scheduler、Mesos、Kubernetes，另外它和 Microservices 所结合的东西也特别多。Infrastructure 领域还有比如 Database 有分 AP（分析型）和 TP（事务型），比如说很典型的大家知道的 Spark、Greenplum、Apache Phoenix 等等，这些都属于在 AP 的，它们也会去尝试支持有限的 TP。另外，还有一个比较有意思的就是 Kudu——Cloudera Open Source 的那个项目，它的目标很有意思：我不做最强的 AP 系统，也不做最强的 TP 系统，我选择一个相对折中的方案。从文化哲学上看，它比较符合中国的中庸思想。

另外，我先后创建了 Codis、TiDB。去年 12 月份创建了 TiKV 这个 project，TiKV 在所有的 rust 项目里目前排名前三。

首先我们聊聊 Database 的历史，在已经有这么多种数据库的背景下我们为什么要创建另外一个数据库；以及说一下现在方案遇到的困境，说

一下 Google Spanner 和 F1、TiKV 和 TiDB，说一下架构的事情，在这里我们会重点聊一下 TiKV。因为我们产品的很多特性是 TiKV 提供的，比如说跨数据中心的复制、Transaction、auto-scale。

接下来聊一下为什么 TiKV 用 Raft 能够实现所有这些重要的特性，以及 scale、MVCC 和事务模型。东西非常多，我今天不太可能把里面的技术细节都描述得特别细，因为几乎每一个话题都可以找到一篇或者是多篇论文，所以详细的技术问题大家可以单独来找我聊。

后面再说一下我们现在遇到的窘境，就是大家常规遇到的分布式方案有哪些问题，比如 MySQL Sharding。我们创建了无数 MySQL Proxy，比如官方的 MySQL proxy、Youtube 的 Vitess、淘宝的 Cobar、TDDL 以及基于 Cobar 的 MyCAT、金山的 Kingshard、360 的 Atlas、京东的 JProxy，我在豌豆荚也写了一个。可以说，随便一个大公司都会造一个 MySQL Sharding 的方案。

为什么我们要创建另外一个数据库

昨天晚上我还跟一个同学聊到，基于 MySQL 的方案它的天花板在哪里，它的天花板特别明显。有一个思路是能不能通过 MySQL 的 server 把 InnoDB 变成一个分布式数据库，听起来这个方案很完美，但是很快就会遇到天花板。因为 MySQL 生成的执行计划是个单机的，它认为整个计划的 cost 也是单机的，我读取一行和读取下一行之间的开销是很小的，比如迭代 next row 可以立刻拿到下一行。实际上在一个分布式系统里面，这是不一定的。

另外，你把数据都拿回来计算这个太慢了，很多时候我们需要把我们的 expression 或者计算过程等等运算推下去，向上返回一个最终的计算

结果，这个一定要用分布式的 plan，前面控制执行计划的节点，它必须要理解下面是分布式的东西，才能生成最好的 plan，这样才能实现最高的执行效率。

比如说你做一个 sum，你是一条条拿回来加，还是让一堆机器一起算，最后给我一个结果。例如我有 100 亿条数据分布在 10 台机器上，并行在这 10 台机器我可能只拿到 10 个结果，如果把所有的数据每一条都拿回来，这就太慢了，完全丧失了分布式的价值。聊到 MySQL 想实现分布式，另外一个实现分布式的方案就是 Proxy。但是 Proxy 本身的天花板在那里，就是它不支持分布式的 transaction，它不支持跨节点的 join，它无法理解复杂的 plan，一个复杂的 plan 打到 Proxy 上面，Proxy 就傻了，我到底应该往哪一个节点上转发呢，如果我涉及到 subquery sql 怎么办？所以这个天花板是瞬间会到，在传统模型下面的修改，很快会达不到我们的要求。

另外一个很重要的是，MySQL 支持的复制方式是半同步或者是异步，但是半同步可以降级成异步，也就是说任何时候数据出了问题你不敢切换，因为有可能是异步复制，有一部分数据还没有同步过来，这时候切换数据就不一致了。前一阵子出现过某公司突然不能支付了这种事件，今年有很多这种类似的 case，所以微博上大家都在说“说好的异地多活呢？”……

为什么传统的方案在这上面解决起来特别的困难，天花板马上到了，基本上不可能解决这个问题。另外是多数据中心的复制和数据中心的容灾，MySQL 在这上面是做不好的（见图 1）。

在前面三十年基本上是关系数据库的时代，那个时代创建了很多伟大的公司，比如说 IBM、Oracle、微软也有自己的数据库，早期还有一个公司叫 Sybase，有一部分特别老的程序员同学在当年的教程里面还可以找

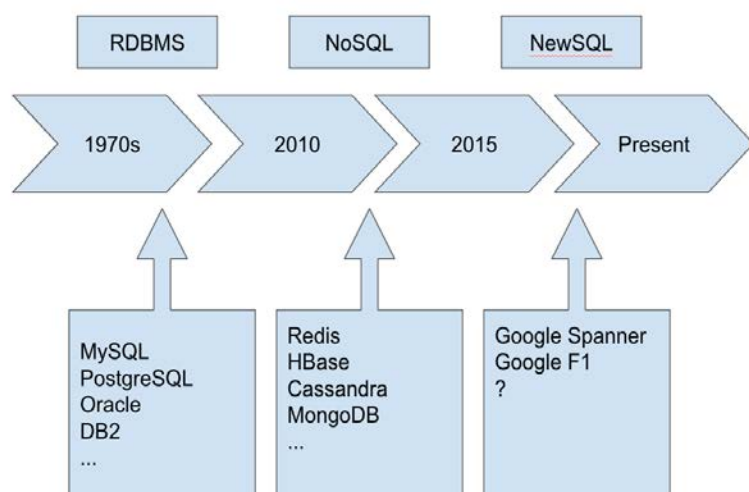


图 1

到这些东西，但是现在基本上看不到了。

另外是 NoSQL。NoSQL 也是一度非常火，像 Cassandra、MongoDB 等等，这些都属于在互联网快速发展的时候创建这些能够 scale 的方案，但 Redis scale 出来比较晚，所以很多时候大家把 Redis 当成一个 Cache，现在慢慢大家把它当成存储不那么重要的数据的数据库。因为它有了 scale 支持以后，大家会把更多的数据放在里面。

然后到了 2015，严格来讲是到 2014 年到 2015 年之间，Raft 论文发表以后，真正的 NewSQL 的理论基础终于完成了。我觉得 NewSQL 这个理论基础，最重要的划时代的几篇论文，一个是谷歌的 Spanner，是在 2013 年初发布的；再就是 Raft 是在 2014 年上半年发布的。这几篇相当于打下了分布式数据库 NewSQL 的理论基础，这个模型是非常重要的，如果没有模型在上面是堆不起来东西的。说到现在，大家可能对于模型还是可以理解的，但是对于它的实现难度很难想象。

前面我大概提到了我们为什么需要另外一个数据库，说到 Scalability 数据的伸缩，然后我们讲到需要 SQL，比如你给我一个纯粹

的 key-value 系统的 API，比如我要查找年龄在 10 岁到 20 岁之间的 email 要满足一个什么要求的。如果只有 KV 的 API 这是会写死人的，要写很多代码，但是实际上用 SQL 写一句话就可以了，而且 SQL 的优化器对整个数据的分布是知道的，它可以很快理解你这个 SQL，然后会得到一个最优的 plan，他得到这个最优的 plan 基本上等价于一个真正理解 KV 每一步操作的人写出来的程序。通常情况下，SQL 的优化器是为了更加了解或者做出更好的选择。

另外一个就是 ACID 的事务，这是传统数据库必须要提供的基础。以前你不提供 ACID 就不能叫数据库，但是近些年大家写一个内存的 map 也可以叫自己是数据库。大家写一个 append-only 文件，我们也可以叫只读数据库，数据库的概念比以前极大的泛化了。

另外就是高可用和自动恢复，他们的概念是什么呢？有些人会有一些误解，因为今天还有朋友在现场问到，出了故障，比如说一个机房挂掉以后我应该怎么做切换，怎么操作。这个实际上相当于还是上一代的概念，还需要人去干预，这种不算是高可用。

未来的高可用一定是系统出了问题马上可以自动恢复，马上可以变成可用。比如说一个机房挂掉了，十秒钟不能支付，十秒钟之后系统自动恢复了变得可以支付，即使这个数据中心再也不起来我整个系统仍然是可以支付的。Auto-Failover 的重要性就在这里。大家不希望在睡觉的时候被一个报警给拉起来，我相信大家以后具备这样一个能力，5 分钟以内的报警不用理会，挂掉一个机房，又挂掉一个机房，这种连续报警才会理。我们内部开玩笑说，希望大家都能睡个好觉，很重要的事情就是这个。

说完应用层的事情，现在很有很多业务，在应用层自己去分片，比如说我按照 user ID 在代码里面分片，还有一部分是更高级一点我会用到

一致性哈希。问题在于它的复杂度，到一定程度之后我自动的分库，自动的分表，我觉得下一代数据库是不需要理解这些东西的，不需要了解什么叫做分库，不需要了解什么叫做分表，因为系统是全部自动搞定的。同时复杂度，如果一个应用不支持事务，那么在应用层去做，通常的做法是引入一个外部队列，引入大量的程序机制和状态转换，A 状态的时候允许转换到 B 状态，B 状态允许转换到 C 状态。

举一个简单的例子，比如说在京东上买东西，先下订单，支付状态之后这个商品才能出库，如果不是支付状态一定不能出库，每一步都有严格的流程。

Google Spanner / F1

说一下 Google 的 Spanner 和 F1，这是我非常喜欢的论文，也是我最近几年看过很多遍的论文。Google Spanner 已经强大到什么程度呢？Google Spanner 是全球分布的数据库，在国内目前普遍做法叫做同城两地三中心，它们的差别是什么呢？以 Google 的数据来讲，谷歌比较高的级别是他们有 7 个副本，通常是美国保存 3 个副本，再在另外 2 个国家可以保存 2 个副本，这样的好处是万一美国两个数据中心出了问题，那整个系统还能继续可用，这个概念就是比如美国 3 个副本全挂了，整个数据都还在，这个数据安全级别比很多国家的安全级别还要高，这是 Google 目前做到的，这是全球分布的好处。

现在国内主流的做法是两地三中心，但现在基本上都不能自动切换。大家可以看到很多号称实现了两地三中心或者异地多活，但是一出现问题都说不好意思这段时间我不能提供服务了。大家无数次的见到这种 case， 我就不列举了。

Spanner 现在也提供一部分 SQL 特性。在以前，大部分 SQL 特性是在 F1 里面提供的，现在 Spanner 也在逐步丰富它的功能，Google 是全球第一个做到这个规模或者是做到这个级别的数据库。事务支持里面 Google 有点黑科技（其实也没有那么黑），就是它有 GPS 时钟和原子钟。大家知道在分布式系统里面，比如说数千台机器，两个事务启动先后顺序，这个顺序怎么界定（事务外部一致性）。这个时候 Google 内部使用了 GPS 时钟和原子钟，正常情况下它会使用一个 GPS 时钟的一个集群，就是说我拿的一个时间戳，并不是从一个 GPS 上来拿的时间戳，因为大家知道所有的硬件都会有误差。如果这时候我从一个上拿到的 GPS 本身有点问题，那么你拿到的这个时钟是不精确的。而 Google 它实际上是在一批 GPS 时钟上去拿了能够满足 majority 的精度，再用时间的算法，得到一个比较精确的时间。大家知道 GPS 也不太安全，因为它是美国军方的，对于 Google 来讲要实现比国家安全级别更高的数据库，而 GPS 是可能受到干扰的，因为 GPS 信号是可以调整的，这在军事用途上面很典型的，大家知道导弹的制导需要依赖 GPS，如果调整了 GPS 精度，那么导弹精度就废了。所以他们还用原子钟去校正 GPS，如果 GPS 突然跳跃了，原子钟上是可以检测到 GPS 跳跃的，这部分相对有一点黑科技，但是从原理上来讲还是比较简单，比较好理解的。

最开始它 Spanner 最大的用户就是 Google 的 Adwords，这是 Google 最赚钱的业务，Google 就是靠广告生存的，我们一直觉得 Google 是科技公司，但是他的钱是从广告那来的，所以一定程度来讲 Google 是一个广告公司。Google 内部的方向先有了 Big table，然后有了 MegaStore，MegaStore 的下一代是 Spanner，F1 是在 Spanner 上面构建的。

TiDB and TiKV

TiKV 和 TiDB 基本上对应 Google Spanner 和 Google F1，用 Open Source 方式重建。目前这两个项目都开放在 GitHub 上面，两个项目都比较火爆，TiDB 是更早一点开源的，目前 TiDB 在 GitHub 上有 4300 多个 Star，每天都在增长。

另外，对于现在的社会来讲，我们觉得 Infrastructure 领域闭源的东西是没有任何生存机会的。没有任何一家公司，愿意把自己的身家性命压在一个闭源的项目上。举一个很典型的例子，在美国有一个数据库叫 FoundationDB，去年被苹果收购了。FoundationDB 之前和用户签的合约都是一年的合约。比如说，我给你服务周期是一年，现在我被另外一个公司收购了，我今年服务到期之后，我是满足合约的。但是其他公司再也不能找它服务了，因为它现在不叫 FoundationDB 了，它叫 Apple 了，你不能找 Apple 给你提供一个 Enterprise service（见图 2）。

TiDB 和 TiKV 为什么是两个项目，因为它和 Google 的内部架构对比差不多是这样的：TiKV 对应的是 Spanner，TiDB 对应的是 F1。F1

Architecture overview (Software)

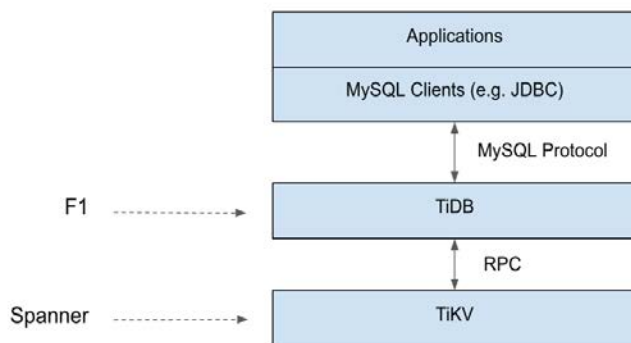


图 2

里面更强调上层的分布式的 SQL 层到底怎么做，分布式的 Plan 应该怎么做，分布式的 Plan 应该怎么去做优化。同时 TiDB 有一点做的比较好的是，它兼容了 MySQL 协议，当你出现了一个新型的数据库的时候，用户使用它是有成本的。大家都知道作为开发很讨厌的一个事情就是，我要每个语言都写一个 Driver，比如说你要支持 C++、你要支持 Java、你要支持 Go 等等，这个太累了，而且用户还得改他的程序，所以我们选择了一个更加好的东西兼容 MySQL 协议，让用户可以不用改。一会我会用一个视频来演示一下，为什么一行代码不改就可以用，用户就能体会到 TiDB 带来的所有的好处。

图 3 实际上是整个协议栈或者是整个软件栈的实现。大家可以看到整个系统是高度分层的，从最底下开始是 RocksDB，然后再上面用 Raft 构建一层可以被复制的 RocksDB，在这一层的时候它还没有 Transaction，但是整个系统现在的状态是所有写入的数据一定要保证它复制到了足够多的副本。也就是说只要我写进来的数据一定有足够多的副本去 cover 它，这样才比较安全，在一个比较安全的 Key-value store

Architecture overview (Logical)

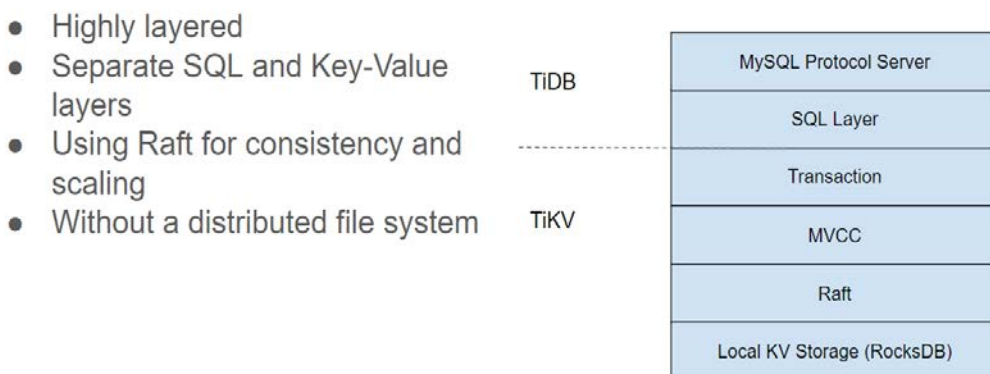


图 3

上面，再去构建它的多版本，再去构建它的分布式事务，然后在分布式事务构建完成之后，就可以轻松的加上 SQL 层，再轻松的加上 MySQL 协议的支持。然后，这两天我比较好奇，自己写了 MongoDB 协议的支持，然后我们可以用 MongoDB 的客户端来玩，就是说协议这一层是高度可插拔的。TiDB 上可以在上面构建一个 MongoDB 的协议，相当于这个是构建一个 SQL 的协议，可以构建一个 NoSQL 的协议。这一点主要是用来验证 TiKV 在模型上面的支持能力。

图 4 是整个 TiKV 的架构图，从这个看来，整个集群里面有很多 Node，比如这里画了四个 Node，分别对应了四个机器。每一个 Node 上可以有多个 Store，每个 Store 里面又会有很多小的 Region，就是说一小片数据，就是一个 Region。从全局来看所有的数据被划分成很多小片，每个小片默认配置是 64M，它已经足够小，可以很轻松的从一个节点移到另外一个节点，Region 1 有三个副本，它分别在 Node1、Node 2 和 Node4 上面，类似的 Region 2，Region 3 也是有三个副本。每个 Region 的所有副本组成一个 Raft Group，整个系统可以看到很多这样的

TiKV Overview

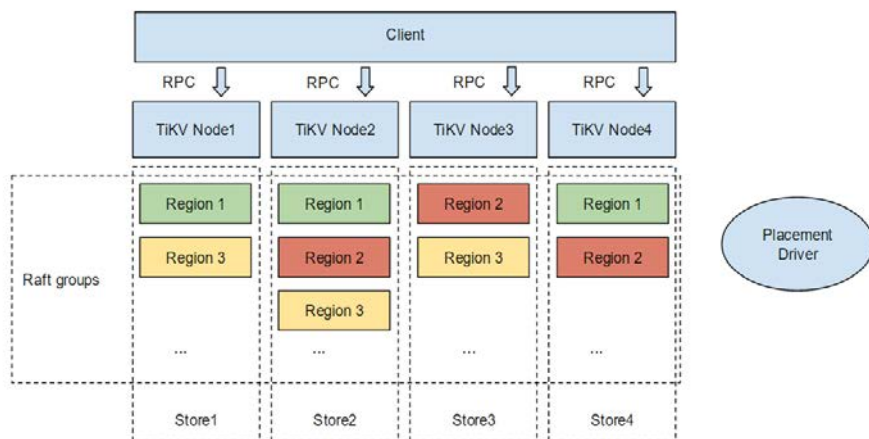


图 4

Raft groups。

Raft 细节我不展开了，大家有兴趣可以找我私聊或者看一下相应的资料。

因为整个系统里面我们可以看到上一张图里面有很多 Raft group 给我们，不同 Raft group 之间的通讯都是有开销的。所以我们有一个类似于 MySQL 的 group commit 机制，你发消息的时候实际上可以 share 同一个 connection，然后 pipeline + batch 发送，很大程度上可以省掉大量 syscall 的开销。

另外，其实在一定程度上后面我们在支持压缩的时候，也有非常大的帮助，就是可以减少数据的传输。对于整个系统而言，可能有数百万的 Region，它的大小可以调整，比如说 64M、128M、256M，这个实际上依赖于整个系统里面当前的状况。

比如说我们曾经在有一个用户的机房里面做过测试，这个测试有一个香港机房和新加坡的机房。结果我们在做复制的时候，新加坡的机房大于 256M 就复制不过去，因为机房很不稳定，必须要保证数据切的足够小，这样才能复制过去。

如果一个 Region 太大以后我们会自动做 SPLIT，这是非常好玩的过程，有点像细胞的分裂。

然后 TiKV 的 Raft 实现，是从 etcd 里面 port 过来的，为什么要从 etcd 里面 port 过来呢？首先 TiKV 的 Raft 实现是用 Rust 写的。作为第一个做到生产级别的 Raft 实现，所以我们从 etcd 里面把它用 Go 语言写的 port 到这边。

图 5 是 Raft 官网上面列出来的 TiKV 在里面的状态，大家可以看到 TiKV 把所有 Raft 的 feature 都实现了。比如说 Leader Election、

Membership Changes, 这个是非常重要的, 整个系统的 scale 过程高度依赖 Membership Changes, 后面我用一个图来讲这个过程。后面这个是 Log Compaction, 这个用户不太关心。

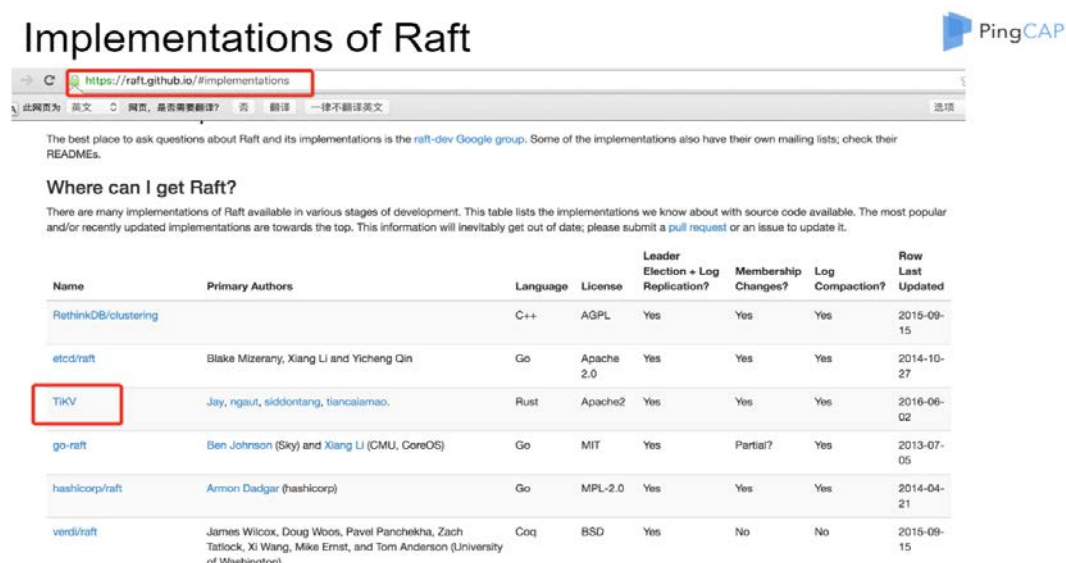
图 6 是很典型的细胞分裂的图, 实际上 Region 的分裂过程和这个是类似的。

我们看一下扩容是怎么做的 (见图 7)。

比如说以现在的系统假设, 我们刚开始说只有三个节点, 有 Region1 分别是在 1、2、4, 我用虚线连接起来代表它是一个 Raft group, 大家可以看到整个系统里面有三个 Raft group, 在每一个 Node 上面数据的分布是比较均匀的, 在这个假设每一个 Region 是 64M, 相当于只有一个 Node 上面负载比其他的稍微大一点点。

一个在线视频默认我们都是推荐 3 个副本或者 5 个副本的配置。Raft 本身有一个特点, 如果一个 leader down 掉之后, 其它的节点会选一个新的 leader, 那么这个新的 leader 会把它还没有 commit 但已经

Implementations of Raft



The best place to ask questions about Raft and its implementations is the [raft-dev Google group](#). Some of the implementations also have their own mailing lists; check their READMEs.

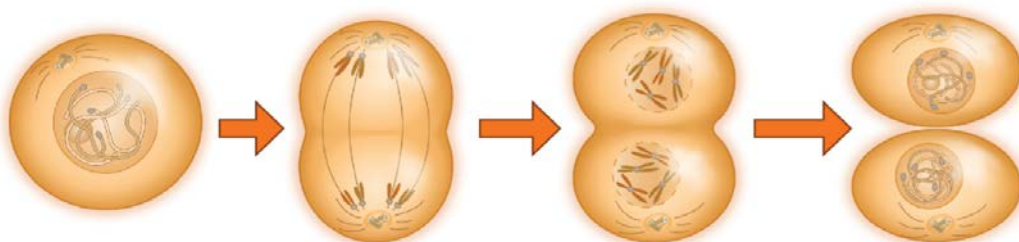
Where can I get Raft?

There are many implementations of Raft available in various stages of development. This table lists the implementations we know about with source code available. The most popular and/or recently updated implementations are towards the top. This information will inevitably get out of date; please submit a [pull request](#) or an issue to update it.

Name	Primary Authors	Language	License	Leader Election + Log Replication?	Membership Changes?	Log Compaction?	Row Last Updated
RethinkDB/clustering		C++	AGPL	Yes	Yes	Yes	2015-09-15
etcd/raft	Blake Mizerany, Xiang Li and Yicheng Qin	Go	Apache 2.0	Yes	Yes	Yes	2014-10-27
TiKV	Jay, ngaut, siddontang, tiancaiamao.	Rust	Apache2	Yes	Yes	Yes	2016-06-02
go-raft	Ben Johnson (Sky) and Xiang Li (CMU, CoreOS)	Go	MIT	Yes	Partial?	Yes	2013-07-05
hashicorp/raft	Armon Dadgar (hashicorp)	Go	MPL-2.0	Yes	Yes	Yes	2014-04-21
verdi/raft	James Wilcox, Doug Woos, Pivrel Panchekha, Zach Tatlock, Xi Wang, Mike Ernst, and Tom Anderson (University of Washington)	Coq	BSD	Yes	No	No	2015-09-15

图 5

Cell division



© 2013 Encyclopædia Britannica, Inc.

Scale-out

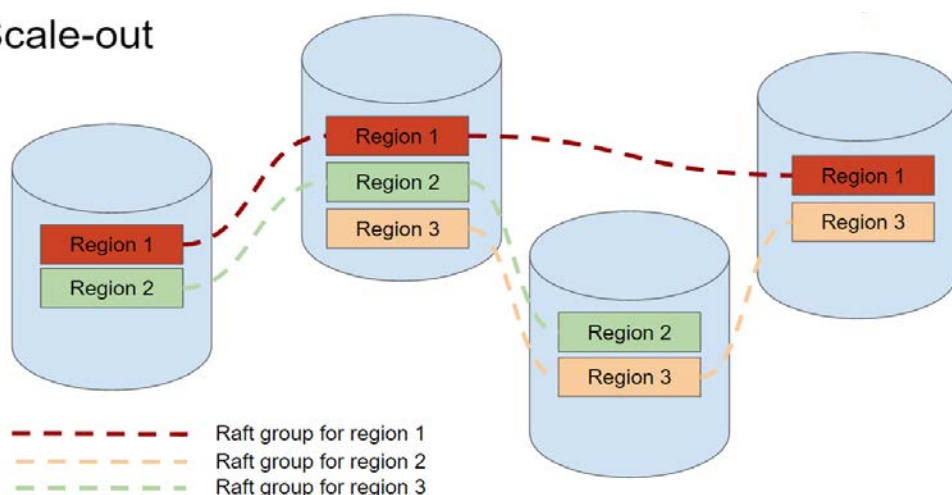


图6、图7

reply 过去的 log 做一个 commit ，然后会再做 apply ，这个有点偏 Raft 协议，细节我不讲了。

复制数据的小的 Region，它实际上是跨多个数据中心做的复制。这里面最重要的一点是永远不丢失数据，无论如何我保证我的复制一定是复制到 majority ，任何时候我只要对外提供服务，允许外面写入数据一定要复制到 majority 。很重要的一点就是恢复的过程一定要是自动化的，我前面已经强调过，如果不能自动化恢复，那么中间的宕机时间或者对外不可服务的时间，便不是由整个系统决定的，这是相对回到了几十年前的状态。

MVCC

MVCC 我稍微仔细讲一下这一块。MVCC 的好处，它很好支持 Lock-free 的 snapshot read，一会儿我有一个图会展示 MVCC 是怎么做的。isolation level 就不讲了，MySQL 里面的级别是可以调的，我们的 TiKV 有 SI，还有 SI+lock，默认是支持 SI 的这种隔离级别，然后你写一个 select for update 语句，这个会自动的调整到 SI 加上 lock 这个隔离级别。这个隔离级别基本上和 SSI 是一致的。还有一个就是 GC 的问题，如果你的系统里面的数据产生了很多版本，你需要把这个比较老的数据给 GC 掉，比如说正常情况下我们是不删除数据的，你写入一行，然后再写入一行，不断去 update 同一行的时候，每一次 update 会产生新的版本，新的版本就会在系统里存在，所以我们需要一个 GC 的模块把比较老的数据给 GC 掉，实际上这个 GC 不是 Go 里面的 GC，不是 Java 的 GC，而是数据的 GC。

图 8 是一个数据版本，大家可以看到我们的数据分成两块，一个是 meta，一个是 data。meta 相对于描述我的数据当前有多少个版本。大家可以看到绿色的部分，比如说我们的 meta key 是 A，keyA 有三个版本，是 A1、A2、A3，我们把 key 自己和 version 拼到一起。那我们用 A1、A2、A3 分别描述 A 的三个版本，那么就是 version 1/2/3。meta 里面描述，就是我的整个 key 相对应哪个版本，我想找到那个版本。比如说我现在要读取 key A 的版本 10，但显然现在版本 10 是没有的，那么小于版本 10 最大的版本是 3，所以这时我就能读取到 3，这是它的隔离级别决定的。关于 data，我刚才已经讲过了。

分布式事务模型

接下来是分布式事务模型，其实是基于 Google Percolator，这是

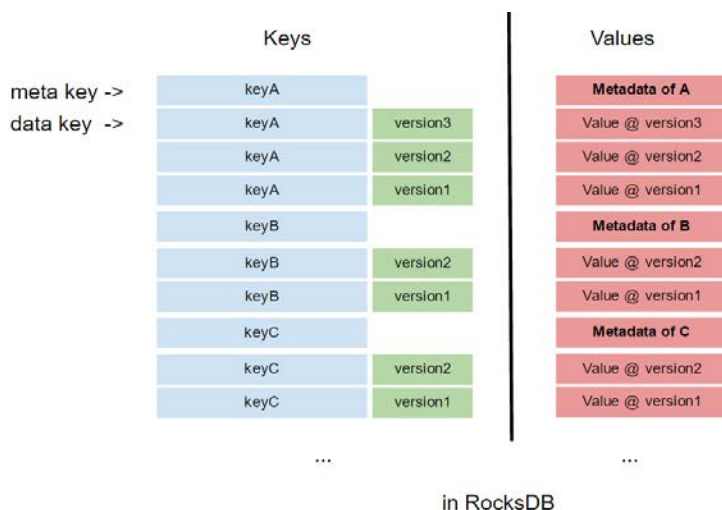


图 8

Google 在 2006 发表的一篇论文，是 Google 在做内部增量处理的时候发现了这个方法，本质上还是二阶段提交的。这使用的是一个乐观锁，比如说我提供一个 transaction，我去改一个东西，改的时候是发布在本地的，并没有马上 commit 到数据存储那一端，这个模型就是说，我修改的东西我马上去 Lock 住，这个基本就是一个悲观锁。但如果到最后一刻我才提交出去，那么锁住的这一小段的时间，这个时候实现的是乐观锁。乐观锁的好处就是当你冲突很小的时候可以得到非常好的性能，因为冲突特别小，所以我本地修改通常都是有效的，所以我不需要去 Lock，不需要去 roll back。本质上分布式事务就是 2PC（两阶段提交）或者是 2+x PC，基本上没有 1PC，除非你在别人的级别上做弱化。比如说我允许你读到当前最新的版本，也允许你读到前面的版本，书里面把这个叫做幻读。如果你调到这个程度是比较容易做 1PC 的，这个实际上还是依赖用户设定的隔离级别的，如果用户需要更高的隔离级别，这个 1PC 就不太好做了。

这是一个路由，正常来讲，大家可能会好奇一个 SQL 语句怎么最后

会落到存储层，然后能很好的运行，最后怎么能映射到 KV 上面，又怎么能路由到正确的节点，因为整个系统可能有上千个节点，你怎么能正确路由到那一个的节点。我们在 TiDB 有一个 TiKV driver，另外 TiKV 对外使用的是 Google Protocol Buffer 来作为通讯的编码格式。

Placement Driver

来说一下 Placement Driver。Placement Driver 是什么呢？整个系统里面有一个节点，它会时刻知道现在整个系统的状态。比如说每个机器的负载，每个机器的容量，是否有新加的机器，新加机器的容量到底是怎样的，是不是可以把一部分数据挪过去，是不是也是一样下线，如果一个节点在十分钟之内无法被其他节点探测到，我认为它已经挂了，不管它实际上是不是真的挂了，但是我也认为它挂了。因为这个时候是有风险的，如果这个机器万一真的挂了，意味着你现在机器的副本数只有两个，有一部分数据的副本数只有两个。那么现在你必须马上要在系统里面重新选一台机器出来，它上面有足够的空间，让我现在只有两个副本的数据重新再做一份新的复制，系统始终维持在三个副本。整个系统里面如果机器挂掉了，副本数少了，这个时候应该会被自动发现，马上补充新的副本，这样会维持整个系统的副本数。这是很重要的，为了避免数据丢失，必须维持足够的副本数，因为副本数每少一个，你的风险就会再增加。这就是 Placement Driver 做的事情。

同时，Placement Driver 还会根据性能负载，不断去 move 这个 data。比如说你这边负载已经很高了，一个磁盘假设有 100G，现在已经用了 80G，另外一个机器上也是 100G，但是他只用了 20G，所以这上面还可以有几十 G 的数据，比如 40G 的数据，你可以 move 过去，这样可

以保证系统有很好的负载，不会出现一个磁盘巨忙无比，数据已经多的装不下了，另外一个上面还没有东西，这是 Placement Driver 要做的东西。

Raft 协议还提供一个很高级的特性叫 leader transfer。leader transfer 就是说不移动数据的时候，我把我的 leadership 给你，相当于从这个角度来讲，我把流量分给你，因为我是 leader，所以数据会到我这来，但我现在把 leader 给你，我让你来当 leader，原来打给我的请求会被打给你，这样我的负载就降下来。这就可以很好的动态调整整个系统的负载，同时又不搬移数据。不搬移数据的好处就是，不会形成一个抖动。

MySQL Sharding

MySQL Sharding 我前面已经提到了它的各种天花板，MySQL Sharding 的方案很典型的就解决基本问题以后，业务稍微复杂一点，你在 sharding 这一层根本搞不定。它永远需要一个 sharding key，你必须要告诉我的 proxy，我的数据要到哪里找，对用户来说是极不友好的，比如我现在是一个单机的，现在我要切入到一个分布式的环境，这时我必须要改我的代码，我必须要知道我这个 key，我的 row 应该往哪里 Sharding。如果是用 ORM，这个基本上就没法做这个事情了。有很多 ORM 它本身假设我后面只有一个 MySQL。但 TiDB 就可以很好的支持，因为我所有的角色都是对的，我不需要关注 Sharding、分库、分表这类的事情。

这里面有一个很重要的问题没有提，我怎么做 DDL。如果这个表非常大的话，比如说我们有一百亿吧，横跨了四台机器，这个时候你要给它做一个新的 Index，就是我要添加一个新的索引，这个时候你必须要不影响

任何现有的业务，实际上这是多阶段提交的算法，这个是 Google 和 F1 一起发出来那篇论文。

简单来讲是这样的，先把状态标记成 `delete only`，`delete only` 是什么意思呢？因为在分布式系统里面，所有的系统对于 `schema` 的视野不是一致的，比如说我现在改了一个值，有一部分人发现这个值被改了，但是还有一部分人还没有开始访问这个，所以根本不知道它被改了。然后在一个分布系统里，你也不可能实时通知到所有人在同一时刻发现它改变了。比如说从有索引到没有索引，你不能一步切过去，因为有的人认为它有索引，所以他给它建了一个索引，但是另外一个机器他认为它没有索引，所以他就把数据给删了，索引就留在里面了。这样遇到一个问题，我通过索引找的时候告诉我有，实际数据却没有有了，这个时候一致性出了问题。比如说我 `count` 一个 `email` 等于多少的，我通过 `email` 建了一个索引，我认为它是在，但是 `UID` 再转过去的时候可能已经不存在了。

比如说我先标记成 `delete only`，我删除它的时候不管它现在有没有索引，我都会尝试删除索引，所以我的数据是干净的。如果我删除掉的话，我不管结果是什么样的，我尝试去删一下，可能这个索引还没 `build` 出来，但是我仍然删除，如果数据没有了，索引一定没有了，所以这可以很好的保持它的一致性。后面再类似于前面，先标记成 `write only` 这种方式，连续再迭代这个状态，就可以迭代到一个最终可以对外公开的状态。比如说当我迭代到一定程度的时候，我可以从后台 `build index`，比如说我一百亿，正在操作的 `index` 会马上 `build`，但是还有很多没有 `build index`，这个时候后台不断的跑 `map-reduce` 去 `build index`，直到整个都 `build` 完成之后，再对外 `public`，就是说我这个索引已经可用了，你可以直接拿索引来找，这个是非常经典的。在这个 `Online`，

Asynchronous Schema Change in F1 paper 之前，大家都不知道这事该怎么做。

Proxy Sharding 的方案不支持分布式事务，更不用说跨数据中心的一致性事务了。TiKV 很好的支持 transaction，刚才提到的 Raft 除了增加副本之外，还有 leader transfer，这是一个传统的方案都无法提供的特性。以及它带来的好处，当我瞬间平衡整个系统负载的时候，对外是透明的，做 leader transfer 的时候并不需要移动数据，只是个简单的 leader transfer 消息。

然后说一下如果大家想参与我们项目的话是怎样的过程，因为整个系统是完全开源的，如果大家想参与其中任何一部分都可以，比如说我想参与到分布式 KV，可以直接贡献到 TiKV。TiKV 需要写 Rust，如果大家对这块特别有激情可以体验写 Rust 的感觉。

TiDB 是用 Go 写的，Go 在中国的群众基础是非常多的，目前也有很多人在贡献。整个 TiDB 和 TiKV 是高度协作的项目，因为 TiDB 目前还用到了 etcd，我们在和 CoreOS 在密切的合作，也特别感谢 CoreOS 帮我们做了很多的支持，我们也为 CoreOS 的 etcd 提了一些 patch。同时，TiKV 使用 RocksDB，所以我们也为 RocksDB 提了一些 patch 和 test，我们也非常感谢 Facebook RocksDB team 对我们项目的支持。

另外一个 PD，就是我们前面提的 Placement Driver，它负责监控整个系统。这部分的算法比较好玩，大家如果有兴趣的话，可以去自己控制整个集群的调度，它和 Kubernetes 或者是 Mesos 的调度算法是不一样的，因为它调度的维度实际上比那个要更多。比如说磁盘的容量，你的 leader 的数量，你的网络当前的使用情况，你的 IO 的负载和 CPU 的负载都可以放进去。同时你还可以让它调度不要跨一个机房里面建多个副本。

10 亿级流数据交互查询

为什么抛弃 MySQL 选择 VoltDB

作者 谭正海 武毅

大数据时代，随着数据量的爆炸式增长，对于数据的处理速度要求也越来越高，以往基于 MySQL 的数据处理方案已无法满足大吞吐、低延迟的写入和高速查询的场景；百分点总结出了一套完整的解决方案，本文就带你一同了解 VoltDB 在流数据交互查询的应用实践。

流式数据交互查询场景

在百分点，每天有 10 亿条记录产生，针对这些大量实时产生的数据，不仅要做到实时写入，类似推荐调优、数据验证等查询要在秒级响应。有简单的单条验证，也有几个小时或一天的聚合计算，也有基于几千万 / 几亿数据表间的联合聚合查询。例如图 1 的 SQL 查询。

对于前期的 MySQL 方案，虽然已经根据一定规则做了人工的分库，但是对于上面 SQL 中的表 Event 落在单机上的数据量达到几千万，Result 表也近千万，在这样的大表之间进行复杂的联合聚合查询，MySQL 查下来要花费 30 分钟左右，甚至更长，或是没响应了。

因此在针对同时要求大吞吐、低延迟的写入和高速查询的场景下，基于 MySQL 的现存方案完全无法实现。在不放弃 SQL 语句的便利基础上，经


```

SELECT b.ab_tactics, SUM(a.t) AS show_times
FROM
  (SELECT session_id,COUNT(1) AS t FROM Event
   WHERE event_name = 'Dfeedback'
     AND banner_id = 'F1667615_AD97_722F_19A4_8B33934D9382'
     AND creation_time >= '2015-12-24 15:37:00'
     AND creation_time <='2015-12-24 18:00:00'
   GROUP BY session_id) AS a
LEFT JOIN
  (SELECT session_id, ab_tactics
   FROM RESULT
   WHERE p_bid LIKE '%F1667615_AD97_722F_19A4_8B33934D9382%'
     AND creation_time >= '2015-12-24 15:37:00'
     AND creation_time <='2015-12-24 18:00:00'
   GROUP BY session_id)AS b ON a.session_id = b.session_id
GROUP BY b.ab_tactics;

```

图 1

历过多种选型和方案调研，最终选择了 VoltDB 来解决此类问题。

如图 2，线上的全量流量，通过 Streaming 总线同时到达 VoltDB 和离线 Hive 表。不同的是，数据写入 VoltDB 使用实时方式，写入 Hive 使用批量方式。新的数据要求在极短的延迟内马上写入 VoltDB 待查询；批量写入 Hive 的数据也可以做到小时级以内刷写到对应分区。

VoltDB 简介

VoltDB 是一种开源的极速的内存关系型数据库，由 Ingres 和 Postgres 联合创始人 Mike Stonebraker 带领开发的 NewSQL，提供社区版本和商业版本。VoltDB 采用 shard-nothing 架构，既获得了 NoSQL 的良好可扩展性以及高吞吐量数据处理，又没有放弃传统关系型数据库的事务支持——ACID。

一般 VoltDB 数据库集群由大量的站点（分区）组成，分散在多台机器上，数据的存储与处理都是分布在各个站点的，架构图 3 所示。

如图 3，集群有 3 个节点、每个节点 1 个站点构成。因此图中的表都

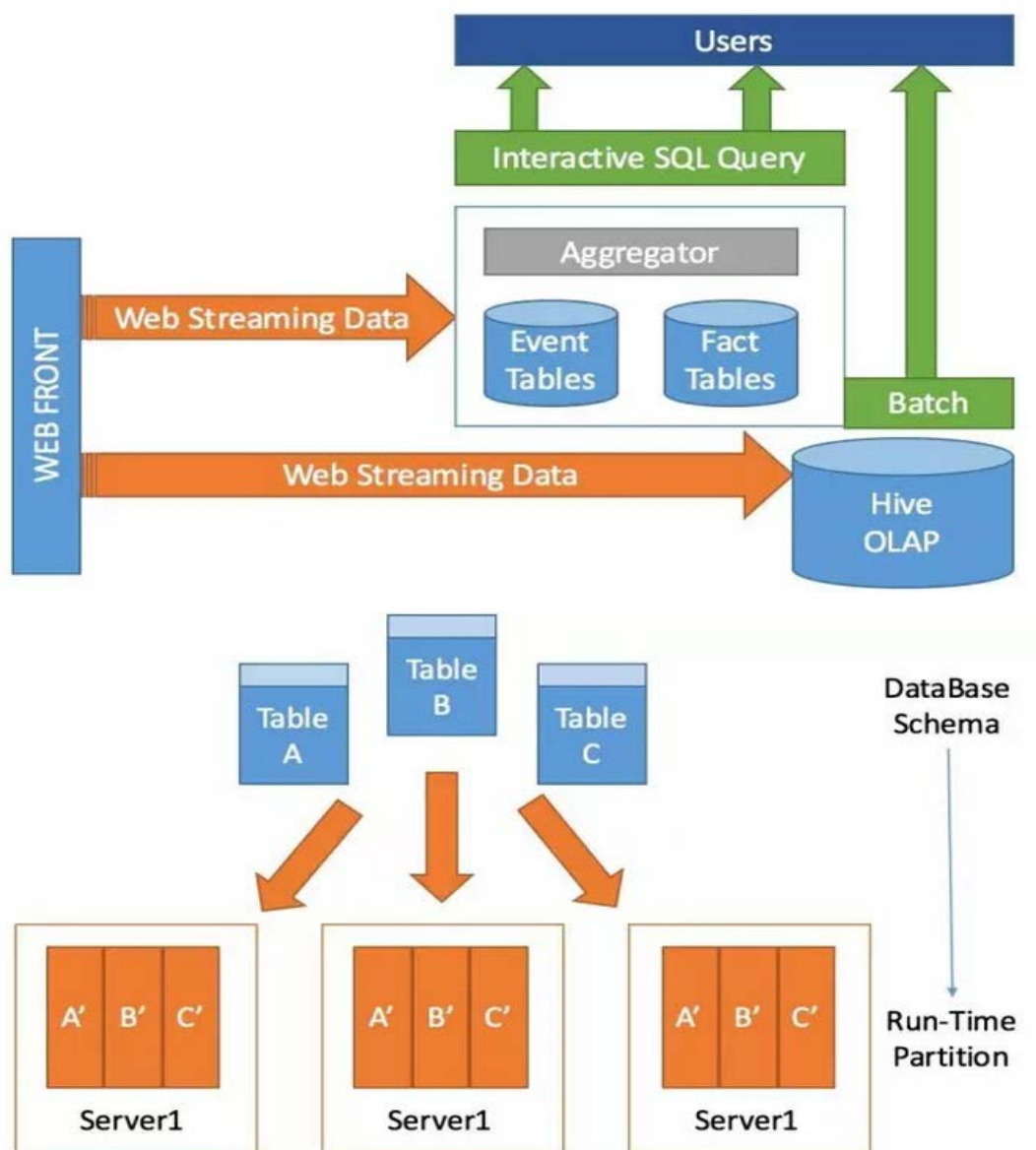


图 2 图 3

只分成 3 个区，当然也可以分成更多的区，那么一张表在单个节点上则存在多个分区。

具体在使用上涉及以下几个概念：

- 客户端可以连接集群中任意一个节点，集群中所有节点是对等的，采用的也是水平分区的方式；

- 每张表指定一个字段作为分区键，VoltDB使用该键采用哈希算法方式分布表数据到各个分区。事实上VoltDB中存在两种类型的表，一种是分区表，还有一种叫做” Replicated table”。” Replicated表” 在每个节点存储的不是某张表的部分数据，而是全部数据，适用于小数据量的表。

这里我们主要看重分区表，分区表的分区字段的选择很重要，应该尽量选择使数据分散均匀的字段。

VoltDB 支持的客户端语言或接口：

- C++
- C#
- Erlang
- Go
- Java
- Python
- Node.js

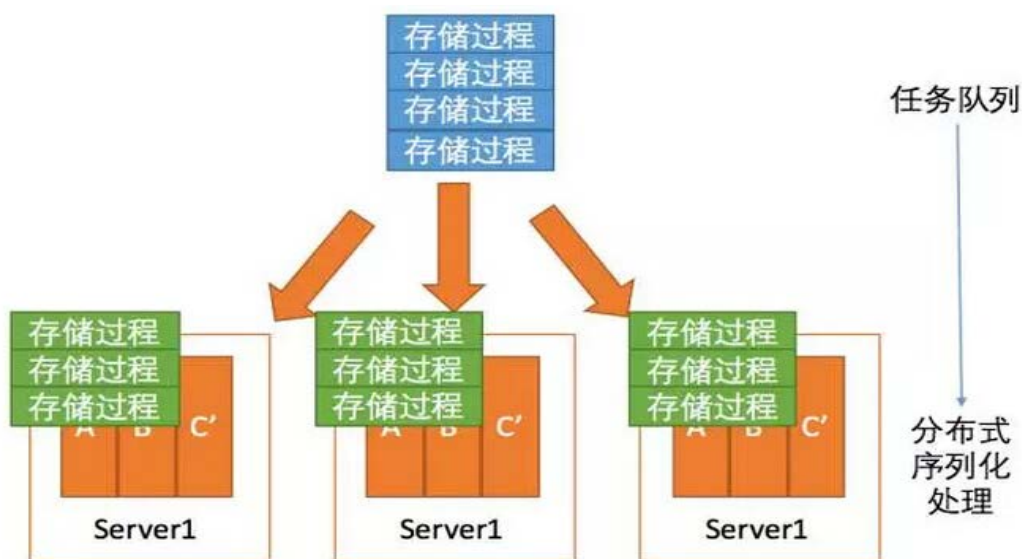


图 4

- JDBC 驱动接口

HTTPJSON 接口（如图 4，这意味着所有能实现 http 请求语言，都能编写 VoltDB 的客户端程序，且非常直观）

VoltDB 的思想是，所有的事务都是由 java 实现的预编译存储过程完成，且所有的存储过程在任意站点上都是序列化执行的，这样使 VoltDB 达到了最高的隔离级别，且消除了锁的使用，很好地提高了处理速度。在官方的测试结果中，VoltDB 可以轻松的扩展到 39 台服务器（300 核），120 个分区，每秒处理 160 万复杂事务。

百分点的实践

VoltDB 在百分点的使用如图 5 所示。

数据从 Kafka 接入，补充下我们使用的是社区版本，社区版是没有持久化到磁盘功能的。因此使用 Kafka 非常适合，一旦出现故障，可以迅速从 Kafka 中恢复数据。

使用 Storm 分布式实时计算框架，作为 VoltDB 的客户端，同时也

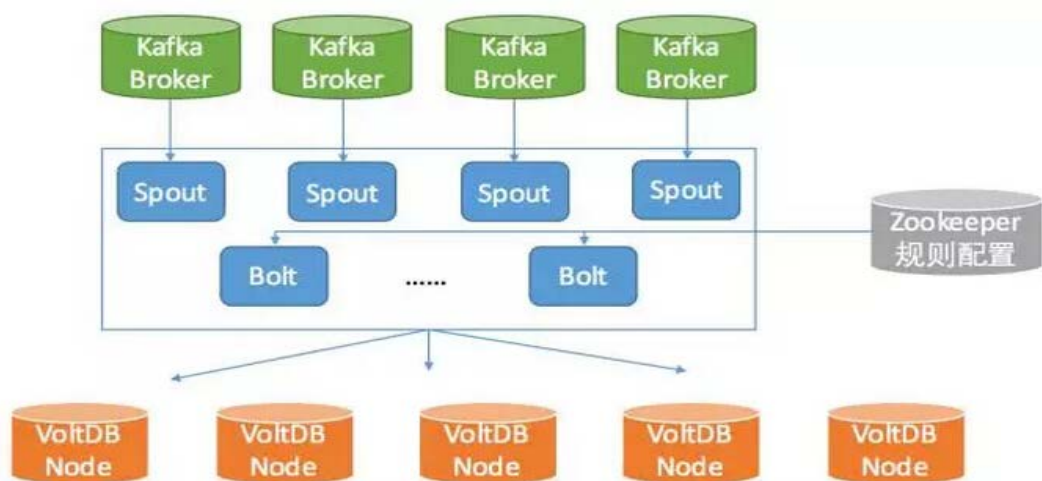


图 5

是 Kafka 的消费者，充分利用 Storm 框架的高可用性和低延时特性。从数据产生，到进入 VoltDB，逐条写入延时在 16ms 内，如果优化成汇聚小 batch 写入，可以提升吞吐但降低写入延迟。

图 6 中的 Zookeeper 保存数据写入 VoltDB 的规则配置，Storm 拓扑通过监听配置决定写入 VoltDB 的数据。由于需要根据业务场景，创建合理的 Schema、数据存储策略以及索引的应用。

数据库	机器数	数据量(条)	同样的查询耗时	数据持久化
MySQL	13	13 * 8000 万=10 亿	30 分钟	有
VoltDB	5	10 亿	10~30 秒	无(社区版)

图 6

例如，在不同表之间的关联查询，关联条件是变化的，可能用户在两表间需要根据商品进行关联查询，也可能会要求根据用户信息进行关联查询，但对于 VoltDB 来讲，在分区表间进行 Join 查询时还要求同时符合 VoltDB 分区规则；由于数据量大，数据结构也存在一定的差异，而 Storm 拓扑内存资源总是有限的。

于是我们采用了 Zookeeper 来做实时的动态配置，可以控制那些变化的数据进入 VoltDB，也可以控制数据进入不同的 Schema（本来是同一张表，但却使用了不同的分区策略而分成多个表）。

另外，VoltDB 目前还没有 TTL 的功能，我们构建自动剔除过期数据的程序，每隔 10 分钟删除一次最老的数据。这样一来，每天 10 亿条记录进出 VoltDB，时刻保持着 24 小时数据的 VoltDB 应用平台就构建起来了。如下是在测试环境上进行的写入测试：

- CPU cores: 24
- Machines: 4

- Client threads: 64

经过在测试环境中运行，在 64 线程时压测达到了 100,000/s 的 TPS，机器的平均 CPU 使用率在 17% 左右，而且 VoltDB 的处理能力基本上随着机器数增加呈线性增长。在高吞吐写入过程中，执行日常查询工作可以达到 10 秒内。

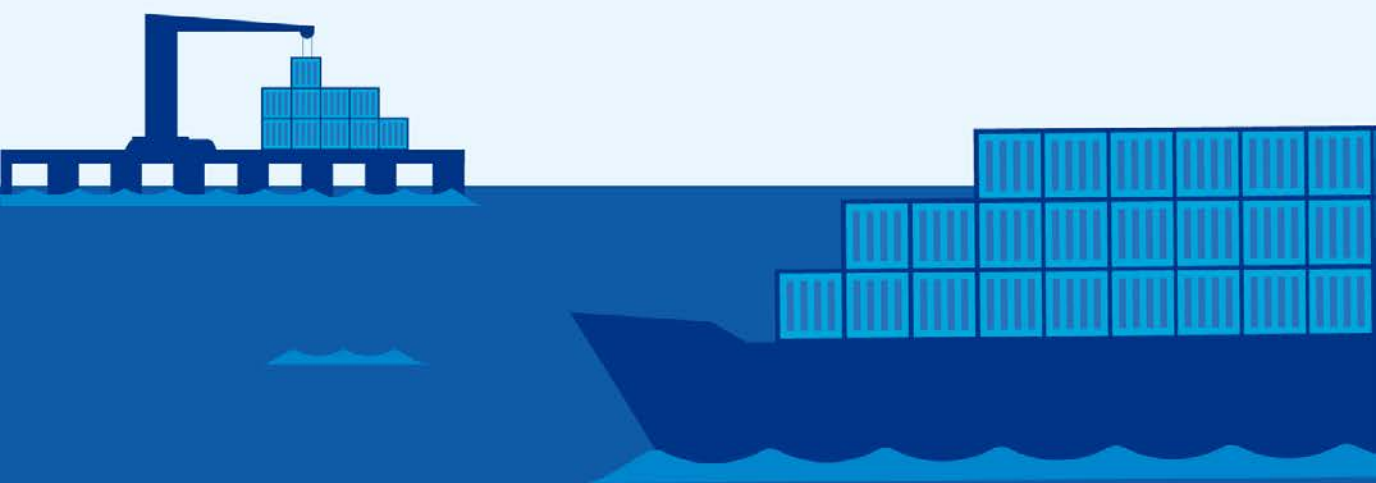
上线后，部署 5 台机器，每台 384G 内存，同样 24 cpu cores。写入的峰值接近 70,000/s，相比原来同样的数据分布在 13 台 MySQL 机器上，在一台机器上（也就是部分数据上）花费 30 分钟的查询甚至使 MySQL 失去响应的复杂查询，到了 VoltDB 集群里基本上减少到十几秒，或者二十几秒，实现流式数据的交互查询。

总结

VoltDB 是一种性能极好的 OLTP 分布式内存 SQL 数据库，也存在一些缺陷，需要看使用场景，如其采用哈希的数据分布策略，进行范围查询可能不能体现出很大的优势。还有动态 SQL，不支持动态表指定等等。

数据库在大数据领域是核心组件，目前数据库产品非常多，可谓百花齐放，各有各的优势 / 不足，作为数据库的设计，关键还是要了解自己的需求，需要数据库来完成什么工作，再去理解每个数据库的功能和使用场景，这样才能根据需求选择合适的产品，设计出合理的 Schema。

容器实践经验谈 **专场**



2016年9月9日 9:15~17:45

中国·北京

喜来登长城饭店 芙蓉厅

阿里百川 Hotfix全面公测

第一时间修复APP问题

立即使用



从单体架构迁移到微服务， 8 个关键的思考、实践和经验

作者 郭蕾

随着微服务架构的持续火热，网络上针对微服务和单体架构的讨论也是越来越多。去年的时候，社区更多的关注点是在二者的区别以及优缺点辨析上，而今年，越来越多的人开始关注如何从单体架构迁移到微服务上。毋庸置疑，微服务的理念正在席卷整个开发者社区，像 Netflix、Uber 这样的公司都是非常成功的应用案例。

但需要注意的是，实施微服务，也需要付出额外的代价，Martin 曾经就说过，除非面对的是一个过于复杂以至于难于管理的单体应用，否则绝对不要考虑使用微服务。大多数的软件系统应该构建为独立的单块程序。确保注重单体应用自身的模块化，而不要试图把它们分离成单独的服务。

普元软件产品部技术经理刘相在微服务架构上有很多的实践和思考，InfoQ 记者就单体应用迁移到微服务的 8 个关键问题对他进行了专访，内容涵盖传统单体式架构的挑战、实施微服务架构的铺垫、改造原则、数据库、中间件、分布式事务、风险规避等。

InfoQ：就你目前的观察来看，企业为什么要从单体式架构迁移到微服务架构？他们遇到的最大的问题是什么？

刘相：我所服务的企业多为传统企业，代码在 100 万行的项目比比皆是，庞大复杂的项目使得开发、测试、维护、运维极度复杂，在弹性、扩展性、可维护性方面也困难重重。除此之外，传统单体式架构遇到的问题还有：

1、团队接手困难

8 年前，我曾接手一个 100 万行级别的项目，那段经历算是一段噩梦：花了 3 个月的时间通读一遍代码；每次修改代码心惊胆战，修改一个 Bug 极可能带来各种隐含的缺陷。

2、臃肿的部署

单体应用每次功能或者缺陷的变更导致重新部署整个应用，这种部署方式影响大、风险高，决定了部署频率低，导致两次发布之间有大量的功能或者缺陷需要进行变更，出错概率增高。

3、局限的弹性与扩展能力

单体应用作为一个强耦合的整体，无法根据业务功能伸缩，只能作为一个整体进行扩展。这造成资源浪费，同时无法针对不同业务模块的特性进行有针对性的伸缩，比如计算密集型服务、IO 密集型服务。

4、阻碍技术革新

团队对新技术的渴望是不言而喻的，团队士气会因为持续的关注在巨石应用的技术栈而降低；单体式架构下的组织通常来说技术选型非常单一，团队技术能力相对单薄，团队的吸引力一般。

除此之外，对于服务等级、安全要求、业务监管等多个维度均需要针对不同的服务实现不同的治理，迁移到微服务架构成为必然。

InfoQ：微服务有它固有的优势，但对企业的基础设施以及团队要求也比较高。你认为在企业准备迁移到微服务架构前，需要做好哪些准备？

刘相：企业迁移到微服务架构前，零号原则就是对业务充分了解，大量企业因历史原因导致了解业务系统的人屈指可数时，就试图转向微服务架构，即使采用最好的技术、工具、架构、团队，最后都会摔得很痛（造成无休止的拆分与变更）。

在充分了解业务的前提下，我认为向微服务迁移，还需在如下三个维度准备：

1、偿还技术债务

自动化测试、持续集成与自动化部署是向微服务架构大规模迁移前必须补偿的技术欠债。微服务架构下，团队管理大量服务，其复杂度和测试难度是几何级增加，利用自动化测试能帮助团队快速有效的验证应用；持续集成与自动化部署保障团队更快速、更容易的修改代码，缺少持续集成和自动化部署，向微服务架构转型过程会异常痛苦。

2、新的架构设计原则

采用微服务架构，应用交付高度复杂化。架构设计原则需要从原来单体式架构下的关注功能、性能等维度向 MVP（最小可用产品）、面向失败的设计（拥抱失败，而不是阻止失败）、宽进严出（对请求宽进严出，对外的响应要严格规范化）、宁花机器一分，不花人工一秒（自动与自助、复杂重复的事情交给平台工具去做，让程序员去做更有价值的事情）、一切皆资源等设计原则转变，形成架构渐进优化的设计风格。

3、团队变革

《Exploring the Duality Between Product and Organizational Architectures》书中给了一个很有意思的观点，组织的耦合度与系统的

模块化成正比。即组织耦合度越高，所开发的产品耦合度越高；组织耦合度越低，所开发的产品耦合度也越低。微服务架构本质上在强调松耦合的架构，因此在微服务架构迁移前，我们有必要对组织进行微调（不要变革，对组织影响很大），确保独立的、小的团队交付一个微服务，同时小团队是微服务的 Owner（除了负责开发外，同时负责测试和运维）。这会极大提供团队的责任感，加速微服务的自治和交付能力。

InfoQ：在整个的架构改造过程中，你觉得有哪些可以遵循的规则？

刘相：微服务架构改造原则业界已经总结非常多了，包括：基于业务进行拆分、采用自动化文化、去中心化、服务独立部署、服务完全自治、隔离失败、渐进式拆分、避免大规模改造原有代码等原则，这些原则相信关注微服务架构的已经相对清楚。结合我们具体的实践，提供一些实际微服务化改造时经验总结：

1、先分离数据库、后分离服务

数据模型能否彻底分开，决定了微服务的边界功能是否彻底划清。我们已经见过太多直接从服务分离而造成多次重构和返工的案例；

2、采用“绞杀者模式”

对于无法修改的遗留系统，推荐采用绞杀者模式：在遗留系统外面增加新的功能做成微服务方式，而不是直接修改原有系统，逐步的实现新老系统替换；

3、建立统一的日志规范

规范整个系统而非微服务的日志体系，采用标准的日志格式非常便于后续的日志聚合检索，便于整体的视角分析、监控、查看系统；

4、选择成熟框架

同时做两件不可控的事情（微服务改造、新技术的冲击）注定项目成

功概率较低，千万避免自己重复发明轮子，尽量选择市面上成熟的开源技术框架进行支撑，比如 Spring Boot、Spring Cloud、Netflix、WildFly Swarm、Docker、Kubernetes 等框架；

当然还有很多的细节规范，比如前后端分离原则、采用全局唯一流水号原则实现全链路交易跟踪、如何进行服务的文档化管理及服务测试 Mock 等。

InfoQ：数据库方面是不是有应该做相应的调整？

刘相：这个话题非常有意义，微服务改造，第一件事情就需要针对数据库模型进行拆分，数据模型边界划清后，服务顺利成章容易划清界限。我们实践过程中强烈推荐的原则是一个微服务对应一个库。当然，随着微服务规模壮大，可以针对性的做读写分离；如果单表数据庞大，可以分表来解决。

InfoQ：如何解决分布式事务一致性呢？

刘相：微服务架构下，完整交易跨越多个系统运行，事务一致性是一个极具挑战的话题。依据 CAP 理论，必须在可用性（Availability）和一致性（Consistency）之间做出选择。我们认为在微服务架构下应选择可用性，然后保证数据的最终一致性。在我们的实践中总结出了三种模式：可靠事件模式、业务补偿模式、TCC 模式。

可靠事件模式：可靠事件模式属于事件驱动架构，当某件重要的事情发生时，例如更新一个业务实体，微服务会向消息代理发布一个事件，消息代理向订阅事件的微服务推送事件，当订阅这些事件的微服务接受此事件时，就可以完成自己的业务，可能会会引发更多的事件发布。

业务补偿模式：补偿模式使用一个额外的协调服务来协调各个需要保证一致性的工作服务，协调服务按顺序调用各个工作服务，如果某个工作

服务调用失败就撤销之前所有已经完成的工作服务。要求需要保证一致性的工作服务提供补偿操作。

TCC 模式：一个完整的 TCC 业务由一个主业务服务和若干个从业务服务组成，主业务服务发起并完成整个业务活动，TCC 模式要求从服务提供三个接口 Try（负责资源检查）、Confirm（真正执行业务）、Cancel（释放 Try 阶段预留的资源）。

三种模式的详细介绍可以参见同事田向阳的微课堂文章：

- 微服务架构下数据一致性保证（一）：<http://dwz.cn/3TVFHs>
- 微服务架构下数据一致性保证（二）：<http://dwz.cn/3TVHBW>
- 微服务架构下数据一致性保证（三）：<http://dwz.cn/3TVJaB>

InfoQ：中间件是否需要做调整，或者重新规划很多新的中间件？

刘相：对于现有中间件，套用 Gartner 流行的一个词“双模”，比如 MySQL、Redis 等中间件适合作为微服务独立出现；对于大块头 Oracle、DB2 数据库或者诸如 Queue 的产品，不适合作为独立微服务出现，可以采用集成的方式工作。

微服务架构需要和新的中间件平台提供融合，比如 IaaS 平台、PaaS 平台等。当然在微服务框架内部，有大量新的中间件的产品，比如 etcd、motan、resteasy、ELK 等。

对于中间件的使用，我们一直保持一个原则：业务逻辑放在服务中，尽量保持中间件的简单。

InfoQ：整个改造过程中，你认为应该如何规避风险以保证平滑过度？

刘相：微服务架构迁移在业务上、技术上都充满了挑战。从规避风险的层面来讲，给大家两点建议：

- 1、重视运营平台建设

在实施微服务改造前，建议先行搭建好运营支撑平台，平台至少提供微服务的编译、集成、打包、部署、配置等工作；如果有能力建议采用PaaS平台，解决微服务从开发到运行的全生命周期管理，同时提供异构环境管理、容器资源隔离与互通、服务伸缩漂移、服务升级与回退、服务熔断与降级、服务注册与发现。平台帮助开发人员解决更多的技术问题，开发人员专注在业务功能的拆分上。

2、从试点入手，逐步推进

为企业的业务应用分级，先从外围应用试点开始；待经验丰富后，进行核心应用当前迁移和大规模的改造。

InfoQ：结合你的实战经验，分享下你认为的从单体式架构迁移到微服务架构过程中的几个关键点？

刘相：结合我们自身的微服务实战经验，迁移过程可以总结出三个关键点：

1. 针对业务系统，重新梳理概念模型+数据模型，切分出松耦合、高内聚的微服务，保障项目组在做正确的事情；
2. 制定微服务开发规范（包括技术架构，Spring Boot+Motan+etcd+RESTEasy+Elasticsearch+Docker+Kubernetes是我们的技术架构选型），保障项目组按照统一的开发规范（技术架构）正确做事；
3. 微服务拆分之后，最大的挑战来自于运维、监控、故障处理，如果团队没有微服务运行的经验，故障之后无法快速定位、快速回复，会受到更大的业务压力，因此后期的微服务运营平台或者管理平台（具体功能参见问题7中的第一部分）对团队异常重要，需要配套设计及时跟上，支撑微服务的运行管理。

50 天 10 万行代码， 一号专车系统重构细节回顾

作者 陈美珍

2013 年底，我关闭当时的创业项目，无所事事之时，打电话向快的 CEO Dexter 请教，当时快的和大黄蜂刚刚合并，他建议我可以先和大黄蜂 CEO 李祖闽（Joe）聊聊。

和 Joe 第一次见面是在虹桥火车站的一家肯德基里碰头，当时我看不太懂打车这个项目。那次碰面，我们聊的却不是出租车，聊的全是专车。两年多前的那时候，做这一块的不多，真正意识到这是个大机遇的且投入资源的创业公司比较少。Joe 跟我讲了许多他对未来专车的构想，一下子就挑起我非常大的兴趣，简单说就是 High 了：）。

Joe 也跟我讲了现在公司在技术上碰到的一些瓶颈，限制了业务的开展，比较痛苦，我们原定半个小时的碰面，结果一聊聊了 4 个多小时后，两人意犹未尽，又一起打车回市区聊了一路，下车的时候我和他约好第二天就去大黄蜂报到。

故事就这样展开了……

定时炸弹

2013 年 12 月 26 日，刚入职的第一天，我看到了打车大战满地销烟的惨状，快的、滴滴、大黄蜂等多家打车公司刚刚在上海打过几波大战，当时大黄蜂打车在上海市场出租车市场的占有率可以排进前三。快的大黄蜂合并后，虽然明确大黄蜂未来的战略方向转做专车业务，同时也上线了第一版本的专车系统，并将部分出租车请求转发给快的打车。但出于种种因素考虑，还是把一大部分的工作重心依然留在出租车上，并没有完全放弃出租车业务。

我初进大黄蜂，又恰逢临近春节，专车系统刚刚上线不久，我们想要在上海做一次“春节 50/100 元专车接送机场（虹桥、浦东两个机场）”的活动，但原来的产品及部分开发人员出现了一些波动，正在陆续办理离职手续，留下的开发人员状态也不太稳定，好在专车系统的即将离职的同学还比较靠谱，被我拉着讲了一个周末的数据库表结构及代码，于是我就是边看代码边熟悉系统的基础上，进行业务开发，技术总监写代码不算什么，其实后面重构上线前期我还做过一整周的测试。

2014 年春节很快过去，快的和滴滴在全国掀起一波补贴大战，大黄蜂的出租车业务在上海市场协同快的也侧面参与了一些，但大部分精力已经开始转向专车业务，我对原来的系统也已经比较了解，当时的系统情况大致如图 1 所示。

Kubernetes 在企业中的场景运用及管理实践 基于 Docker+CI/CD 的 DevOps 实践经验分享 基于 kubernetes 的容器云平台落地与实践 性能可视化实践之路，邀您一起探索！ 又拍云 - 领先的直播云、点播云等多场景 CDN 服务！

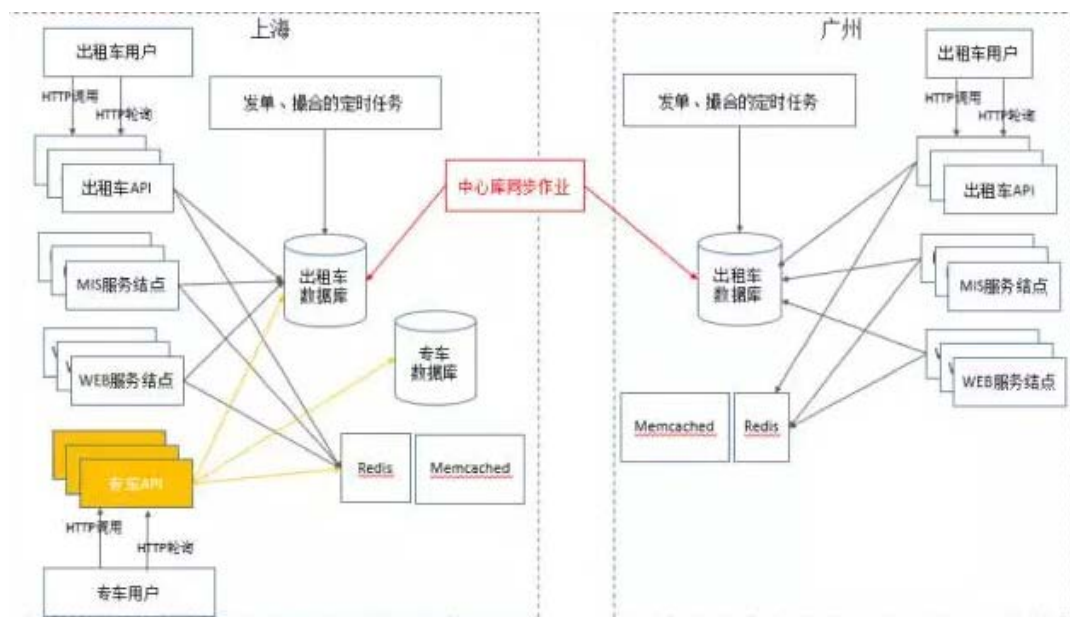


图 1

当时出租车主要开了两个城市，上海和广州，各部署一套系统，数据库、缓存、Web 服务、API、定时任务都在两地部署，由中心库负责定时同步主要的数据库，比如用户信息等，但因为两地市场开拓的时候，运营策略不同，比如各项优惠、加急令、司机补贴等政策在两地各有不同，于是就维护了两套 PHP 代码，大致逻辑相似，细节又各有不同。

而专车系统是后来开发，以 Java 为主要开发语言，基础数据（用户、优惠券、订单等）依然依赖于出租车数据库，但主要数据库和代码部署在上海，因为专车只开了上海市场，所以暂时没有什么影响。这样的系统架构限制了当时的业务发展，也给未来留下了许多定时炸弹，下面我们会展开讨论。

但在这里需要申明一点：看到这样的系统结构，确实存在许多不合理性，单纯一张九十几个字段订单表就能干掉一大票观众，但我要为其辩护，在当时打车大战初期，业务的发展是野蛮式的，每天各种补贴政策、活动

的调整，开发人手严重不足，技术完全是被业务拖着跑，相信经历过创业这个阶段的人都会有深刻感触。

很显然，这样的系统架构已经不能满足当前的业务需求，越往后拖，隐患越大，定时炸弹越多，我评估下来，主要存在这么几类大的问题：

1、多地保存数据，多地多份代码不可维护。

- 多地逻辑代码冗余：代码无法维护和管理，添加一个业务逻辑要同时改两份码，或者只改一份，但下一次增加新业务逻辑时，因为两个地区的代码逻辑差异越来越大，需要非常小心，坑越挖越深。
- 数据同步问题：上海的乘客到广州不能下单、优惠券不见了，反之亦然，上海的乘客出差去外地后，也不能下上海的机场、车站的预约订单，甚至不能登陆。
- 并发冲突：两套数据的维护，一不小心就出现数据不一致的问题，无法合并，也留下被攻击的隐患。

2、无法快速开城，每开一个城市都要部署一套存储和代码，准备硬件、进机房、部署、调试，效率太低，还要做好数据的同步，及当地的新政策的业务开发，开一个城市需要一个月左右的开发周期。

3、数据库表结构和代码许多地方都是堆出来的，不可维护，比如一张订单表有九十几个字段；PS：你没有看错，就是九十几个字段。

4、手机端通过轮询接口调用 API，为了快速获取成单、通知、补贴政策调整等信息。但毫无疑问，这个对于手机的资源消耗（流量、电量）很大，也增加服务器的开销。

5、公共服务和业务代码耦合太紧，比如优惠券、支付等模块；

6、专车司机公里数计算不准确，出租车司机带有打表器，专车司机

主要靠手机 GPS，手机会有信号不准，接受不到 GPS（跳点、断点、隧道、高架）等情况；

7、系统压力大，活动一上来，慢查层出不穷，各个状况出现；

8、API 接口没有加密机制，容易被刷，用户身份容易被窃取；

9、无法适应专车灵活多变的业务场景，每做一个活动，都需要 2,3 周的开发；

还有许多细节，比如轮询接口直接访问数据库、GPS 坐标系混乱、司乘价格绑定、重复上报 GPS、电子围栏不可维护、支付、定位、保险、第三方接口接入等，已不能满足业务发展，这里就不再一一列举。在这样的前提下，内部反复多轮沟通后，我们启动了重构计划。

和时间赛跑的重构马拉松

2014 年 4 月 9 日，我们正式启动了重构，重构目标就是在解决这些问题的基础上，完成整个系统的第一阶段的服务化工作，大致目标包括有：

- 设计一套灵活的专车系统，以应对专车复杂的业务场景；
- 整个系统，包括存储及服务集中部署，便于管理维护；
- 需要支持出租车、专车两项业务的迅速开城；
- 手机端和服务器的实时消息（包括接单、抢单、成单、通知、计费等）推送采用长连接；
- 接口定义一套的新的数据协议，请求加密，防 token 窃取，请求防篡改；
- 采用多级缓存方案，数据库、Redis、MongoDB、分布式本地缓存；
- 引入分布式日志系统（重构后期来不及加入，只做了一部分，但后期为此吃足苦头，快的的兄弟们给了我们很大的帮助）；

- 公共服务及业务服务的抽象及服务化；
- 南、北向接口分离，南向接口主要对接外部供应商，北向接口仅指出租车部分，主要面向第三方渠道、比如携程、去哪儿等；
- 数据库分库、分表、读写分离；
- 优化寻找周边司机，道路距离计算等算法；
- 上阿里云。

图 2 是第一阶段的服务化设计后的架构图。

图 2 中大部分的设计都实现了，并取得了不错的效果，但也埋了坑。其中订单服务在重构中期因为出租车业务的停止，为了贪图调用方便，又把它合到专车核心服务，后续在订单量爆发的时候，为此吃足苦头。出租车司机 API 也因为方向问题停止维护。

在当时的环境下，重构这样的一个系统主要有这么几大挑战：

- 1、 时间不够：项目计划梳理需求和旧有系统逻辑 2 周，开发 & 测试 10 周，上线 2 周，预计在 7 月底左右上线。



图 2

2、开发资源严重不足：前线的还在继续作战，原来系统还需要有人留下做维护性开发，真正能抽出来做后端系统重构的开发人员只有5个（包括我），4个Java开发，1个PHP开发，其中有3个都是新招的开发人员。APP开发也只有4个人参加重构，同时负责iOS、Android。

3、对公共模块业务不熟悉：因为专车系统是搭建在原出租车系统之上，所以大部分公共模块都依赖出租车业务，而原出租车业务的开发人员流动比较大，代码也过了好几手之后，留下来的人员对其业务细节，遗留代码并不了解，这将对后续的数据迁移留下很大隐患；

4、业务复杂：如何设计一个灵活多变的专车系统，以适应快节奏的运营策略。专车因为其特性决定了它的业务复杂多变，随便举几个例子。

- 策略1：希望做接送机场订单一口价的活动，春节前，终点为虹桥/浦东机场，起点为全上海任意地区的，可享受50/100元一口价，春节后，起/终点倒置，为回程的乘客服务；
- 策略2：当某种车型不够用时，只有下机场单或者特定的好用户时可以看到；
- 策略3：高峰期时，高级车型可以向下接单，但司机收入不变，给抢单积极的司机更好的订单；
- 希望在上海内环以内做一口价活动，20元一口价，随便打车，但司机价格会根据活动情况分等级调整；
- 策略4：好的预约订单给抢单积极的司机，好的即时订单给近的司机；
- 策略5：多加一个专车产品，带有婴儿椅的车子，女性乘客可以打到；
- 策略6：打一个专车，如果时间超过2个小时，希望变成包车服务，但价格要有给乘客优惠；

- 策略7：我想让某个起点的乘客享受费用折扣，甚至免单，但平台按正常价格给司机计费；
- 策略8：每开一个城市，我想对机场、火车站的订单实行一口价策略；
- 策略9：在某些城市，有一部分司机是合作司机，一部分加盟司机，两者计费规则完全不同，需要满足。

基本上，这样的策略每个月都有，全国几十个城市，每个城市可能还不同，如果需要通过开发来满足这样的需求，那么几百人的开发团队都不够用，运营时间上也等不了。所以，如何设计一个高可用，灵活多变的系统来适应业务的需求，这是一个非常大的挑战。

5、技术挑战：剩下还有就是一些技术上的挑战，比如快速搭建一套分布式服务框架，如何快速找到周边的司机、如何通过道路拟合比较准确的计算道路距离、长连接的 QoS 如何保证、司乘两端心跳上报异常如何处理、如何防止心跳风暴、规则引擎的性能优化、发单 / 抢单处理队列的守护机制等。

如何解决这几个难题

先说开发资源吧，开发资源属于硬伤，创业公司前期招人，招到好的人比较难。业务等不了，有句话说的好“有条件上，没有条件创造条件也要上”，这就是当时的情况，人就这么多了，边做边招吧，后面两个多月我们陆陆续续招了六七个开发人员，但新招聘的大部分经验还比较浅，但也基本上顶过去了。

再说到公共模块，当我入职一周左右的时候我就知道系统必须大规模重构，但当时因为种种因素提重构必然会给业务、团队造成比较大的影响，也会影响刚刚合并的团队信心，所以前期更多是花精力在整合团队和拖着

专车系统往前走。

公共模块的代码都是在出租车业务系统，向原来的开发人员了解细节时，被告知许多历史遗留问题已不可追溯，只知道不要轻易去动，一动就崩。于是，重构计划开始后，我挑了一个对原出租车业务比较了解的哥们，拉着他两个人慢慢翻 PHP 代码，翻数据库表，通读一遍，才开始抽象公共模块及进行数据库设计。

其次，说到专车系统灵活多变的设计，我首先把专车业务的预估、选择车型、发单、抢单、撮合、做单、计费、通知等流程环节做了抽象，并固化，把可变的部分剥离。

- 比如预估算价、显示车型、计费时将司机和乘客完全分开，实际上在业务抽象的角度来看，专车的乘客和司机发生的交易对象都不是对方，而是平台，这样就平台在运营过程中就具备非常灵活主导权；
- 比如消息通知模块做了抽象，把不同阶段消息体抽象成模板，模板中带有变量，举个最简单的例子：如参加某个活动下单的用户和普通下单用户，在下单或计费时收到的短信内容不同，但实际只是短信模板不同，中间的价格用变量替换，有的甚至不发短信；
- 比如某些特定场景，第一轮发单不希望发给某类司机，要第二、三轮才发给他，撮合成单时也根据不同维度的指标进行排序定义；
- 比如将优惠券可用在不同业务场景定义成模板，再模板上增加 Scope，Scope 又反作用到产品上面；
- 比如，对专车运营中最经常发生变化的预估、可见车型、发单、抢单、撮合逻辑，通过将乘客和司机的特征指标抽象，这个乘

客、司机、订单三个纬度的特征指标抽象很关键，在后期和大数据结合，也可以将乘客司机的画像指标引入，再利用规则引擎来配置每个环节的变化，见图3。

在将以上所有这些灵活配置的内容之上，又定义了大小产品的概念，把这些灵活多变的配置参数抽象成一个个产品，每个产品维护这些配置参数或规则，从而达到运营策略和运行系统的隔离，实现了一套灵活的专车业务系统。

这些产品对应给乘客看到的其实是一个个车型，但看起来同样是经济型的车，在后台根据不同的条件已经被抽象配置成不同的定义规则，如机场订单的经济型和非机场的经济型就完全是不同的业务规则，无论从预估的价格、发单、抢单、撮合、通知等都完全不同。

基于指标的逻辑表达式设计

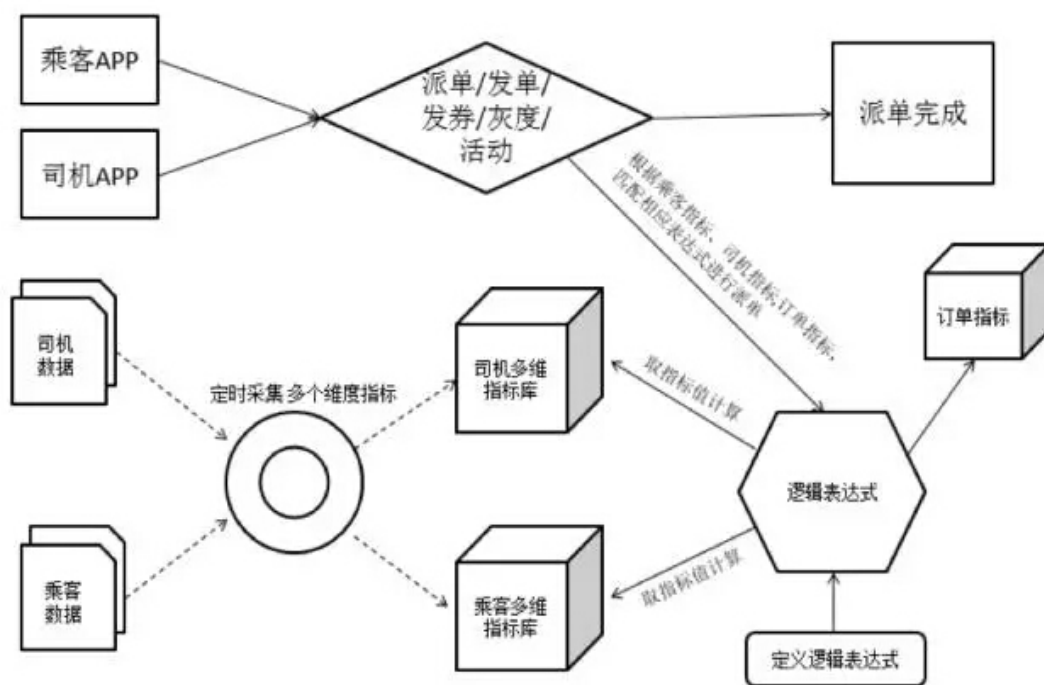


图 3

这样的设计在重构上线后,业务系统大框架在两年时间基本没有动过,最多是增加为一些新的指标或者函数进行开发,但不会影响主流程,业务系统的架构非常稳定。

市场部门或是业务部门要上线一个新的运营策略,比如一号专车在 2015 年和 Uber 打战的时候,上线一号快车,在研发部门实际只用了两天就已经配置测试完毕,为了配合运营的工作,才拖了一周才上线。

限于篇幅因素,技术细节这里就不再多说,给大家看一个最小片段的机场单的产品命中规则:

```
( 单v城市 ==1 ) && ( 函v电子围栏(单v目的地,上海浦东机场) ) && ( 单v渠道 ==0 || 单v渠道 ==1 || 单v渠道 ==201 ) && ( 单v类型 ==2 )&&( 单v入口 == 0 )&& !((单v用车时长>=1)&&(客v版本号>=4))&&(单v回调码==0)
```

题外话:在一次偶然的机会,和行业内的各家专车系统对标时,发现一号专车这种系统设计和 Uber 的设计思路非常类似,所以 Uber 才可以让各城市经理在不同城市开通不同车型做各种各样的活动,Uber 系统应该是经历了数年演化,而一号系统重构头尾只花了 2 个多月,实际上一号的指标维度非常多,有些甚至用自定义函数来组合使用一个指标,因为国内的运营策略变化实在太快太多了。殊途同归吧。

最后再说到时间、时间、时间啊!上面提的许多技术问题,如果从时间的维度上考虑,根本就不是问题,什么 GPS 距离计算算法不准、去噪及道路拟合算法不够优、流控还没有做、长连接服务质量不好、SQL 慢查优化,给我们时间,统统都不是问题。重构计划是 4 月 9 号开始,我们在阅读旧有代码、梳理业务、设计新的数据库结构、搭建团队、搭 API 接口层框架、搭分布式服务架构(Thrift + ZK + DHF SRV Framework + Chukwa[实际没用起来])的过程中,很快就到了 5 月 1 号。

劳动节后，第一波炸弹来袭，听说其他几家打车应用也在做专车，天下武功，唯快不破，我们必须最先上线，于是计划调整改到 7.15 上线，下了死命令，好吧，干吧，需求不能变，时间变了！临近 5 月中旬，新的需求来了，要求我们直接和快的打车端对接，至少要在重构上线后两周内也上，没有办法，在打车领域竞争的激烈程度超出所有的想象，业务发展的速度都是按天算的。

于是我们把上线时间又提前到 6.15，并计划 6.30 上线快的 APP 端的一号专车，时间，还是时间，需要提前梳理对接流程、接口定义及开发，人呢？时间呢？好吧，继续干吧！没有抱怨，也不谈梦想、更没有什么改造世界的想法，仅仅是一份信任、一个承诺，整个重构团队日夜加班，从 4 月 9 号计划启动，到 6 月 18 号上线，连续 70 天没有休息一天，前后端的重构团队都是全公司来得最早走得最晚的人，终于还是让系统上了线。

PS：苦过笑过，留下的都是许多欢乐回忆，当值得书写留念。

上线前一天

上线前一天，虽然已经预演过两次，但大家心里都没有底，因为专车业务量刚刚起来，系统如果真的出问题，将会给对手们一个绝佳的机会。内部在讨论的时候，问的最多的一个问题是如果系统挂了，我们有什么预案，做了两套，一套是切回老系统，但会出现新旧数据不一致的问题，后续补数据非常麻烦；再一套是需要运营的兄弟们支持，系统只记录下单信息，然后出后台报表，由运营同学手工派单，再打电话通知乘客和司机接送，回到原始社会，这就是现实。

通宵不眠，迁移当天增量数据、服务上线、回归验证、强制升级，深更半夜，运营同事们也开着车在路上晃荡着帮忙路测，悦耳的新订单铃声，

成单播报不停响起，终于熬过去了。

系统上线：快乐并痛着

系统刚上线，基本的业务流程虽然没有出什么大问题，但随着订单量的迅猛增长，迅速开城，业务变化，各种状况层出不穷：

- 长连接QoS没有做好，订单撮合成功，但司机或乘客却没有收到推送通知；
- 司机行驶距离根据GPS+WIFI+基站三种制式的GPS坐标值，去噪算法做得不够好，计算出来的距离差异巨大；
- 定位不准，司机有意或无意没有开GPS、网络，导致发单距离太远；
- 订单量上来，索引不够优化，慢查SQL层出不穷，数据库CPU百分百；
- 当时上的是阿里云，阿里云的数据库专家在监控应用的时候，曾经给过一份数据库的诊断报告，非常客观的将我们数据库的问题类比成某宝08、09年的状态，惭愧；
- 代码没有Review，小分支的逻辑测试不到位，出现死循环，拖垮系统；
- 当日订单超过10W的时候，司机做单时上报的心跳对后台存储（MongoDB、数据库）压力过大；
- 因时间问题，日志系统没有上线，在跨服务的应用中，无法实时监控各服务运行异常并及时告警，为此付出很大代价；
- Redis在日订单过百万后，也扛不住压力了；

一方面前方捷报不断，另一方面后方销烟弥漫，快乐并痛着，不停的发布新版本，修复BUG，优化系统，出现过好几次系统大规模宕机，大部分都是因为各种因素（不仅限于慢查）导致数据库不堪重负引起连锁反应，

在订单过百万之后更是连续一周的时间，在每天的晚下班高峰期系统负载扛不住，当然后来在前后端团队通力合作下，都顺利优化顶过去了，大致回想一下，挑一些简单的列一下：

- 优化道路距离优化算法；PS：高德地图的兄弟给了许多帮助；
- 所有道路距离计算只取GPS，基站及WIFI获取的点不做为道路计算使用，但听单可以使用WIFI的GPS点；
- 优化寻找周边司机算法；
- 提升长连接的通讯服务质量，在移动环境下网络不稳，需要双向确认及心跳包客户端打包机制；
- 服务继续拆分；
- 数据库分库、分表、一主多从、读写分离；
- 多级缓存，优化缓存的使用，数据库只做存储，如Redis Sharding分Memory和Persistence两类等；
- 设计了两层的限流熔断方案、服务降级策略等；
- 旧的SQL全量Review，新SQL必须DBA确认后方可上线；
- APP端也优化了访问策略；
- 加上日志监控系统，监控系统整体运行状况；Ps：感谢快的兄弟们的支持；
- 增加服务器，这一点我们做得不错，使用了不到一百台的4核*2.8G CPU，16G内存的服务节点，就挺过了数百万单的业务，且每个服务结点的资源消耗都在个位数；

还有很多细节已经回忆不起来，欢迎大家后续多交流。

一号专车的重构其实是重写，在业务不停往前奔跑的过程中，重写是下策，风险很高，要慎而又慎，好在当时的订单量还不小。大部分时候，系统重构应该尽量要在充分了解业务的基础上，采用分而治之，分阶段进

步的方式来，开着飞机换引擎还好，但我见过开着飞机换飞机的重构计划，着实为对方捏一把冷汗，不知道最后结果如何了？当然，如果决定要动手，那还是越早做越好了。

后序

以上行文仅做为系统重构的技术回顾，重构过程艰难而痛苦，它不仅限于技术层面的难度。此后的过程中系统挺过了一轮又一轮浪峰的冲击，也逐渐趋于稳定，基本上算是完成了组织交给我的任务。

2015 年底，Joe 开始了他的新项目——四叶草车险，开始了另外一次从 0 到 1 的挑战之旅，他再次邀请了我，我于 2015 年 12 月 31 日离开滴滴快的，加入四叶草车险平台参与互联网保险的创业项目。

这一次挑战更大，保险业在中国是一个万亿级的市场。据了解，国内的比较大的传统保险公司，一家公司往往都有三、四百个系统在支撑业务运转。保险未来的变化极其复杂多样，我们才刚刚开始，未来还要迎接诸多挑战，在这个过程中，我们需要更多的人才加入，来和我们一起体验新的重构之痛（le）。说句老套的话，我们可以提供业界有竞争力的薪资以及和团队一起战斗成长的经验。

作者介绍

陈美珍（Frank），微信号 zhaocaimaolin，12 年的软件研发以及技术管理经验。擅长互联网的高并发、高可用的分布式系统架构设计，组建并带领团队完成项目的订单从零到数百万量级的突破。对大中型复杂系统的需求分析、抽象、架构设计、拆分、服务化设计及整合也比较擅长，有多年证券、电信等传统业务系统实战经验。

京东前端： 三级列表页持续架构优化

作者 王向维

1. 京东三级列表页

三级列表页是什么

列表页是京东商城的三大核心系统之一。京东三级列表页是用户选取商品类型后，展示同类商品的页面，具体如图 1 所示。

如何进入三级列表页

用户在首页左侧的导航树中（见图 2）、全部商品分类列表页或者顶部面包屑导航中，选择到商品的最小分类级别后，就可以到达三级列表页。

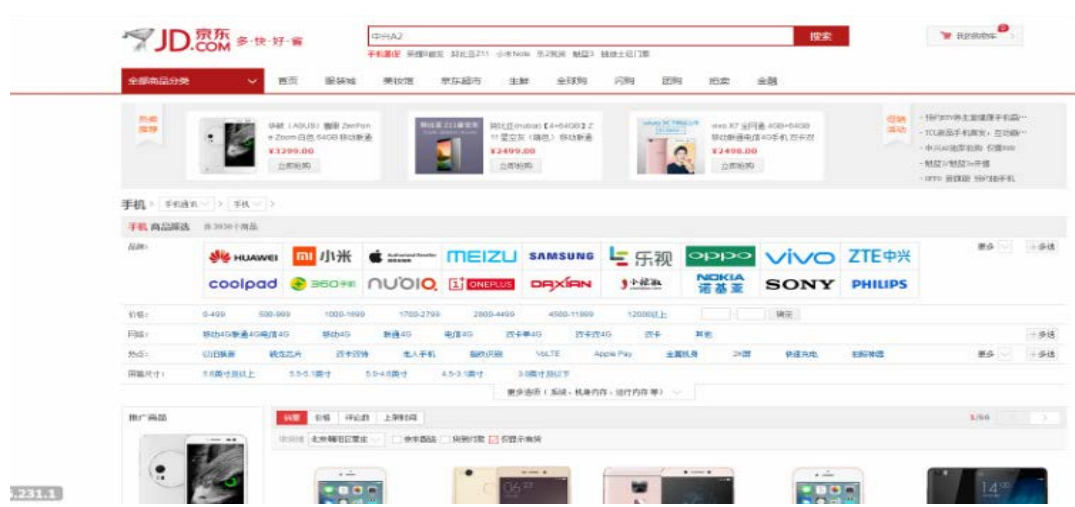


图 1



图2

三级列表页的作用

该页面根据用户选择的商品类目进行检索，将结果以列表的形式展现在页面上。使用户快速找到自己需要的产品，提高用户购买转化率。

三级列表页的业务特点

涉及到多维度多因子质量分综合排序，排序质量的效果直接关系到转化率，客单价，进而影响到 GMV，实质上是数据挖掘和机器学习技术在海量数据上的应用。

在不同三级类目下，通过复杂不确定的查询条件、属性区域和列表查询结果的实时联动，以及跟区域相关的库存、京东配送、货到付款等复杂业务逻辑下，做到高并发实时计算。

2. 优化原因

三级列表页的页面周围依赖的内部系统太多，要做到异步化展示，阻塞可降级。

在持续开发一个核心系统过程中，除了满足业务需求外，还应该考虑系统未来的架构，追求极致的系统的可用性、高性能和稳定性。这个过程

是一个长期积累和重构的过程，京东三级列表页的优化工作，就是这个过程的一部分。

优化前的状况

优化前的三级列表页有以下特点：

- 基于搜索实现；
- 全量数据，搜索结果不理想；
- 接口响应时间长，影响了用户体验；
- 没法针对数据做二次优化；
- 转化率相对较低；
- 基于以上原因，需要对三级列表页做出改变，也就是对老版本进行重构。

重构版本目的

通过优化，希望达到以下目的：

- 非全量数据，线下异步根据数据模型进行进行筛选部分最优数据；
- 要求实时过滤计算，接口响应时间要快，保证用户体验；
- 数据进行优化，提高转换率，提搞GMV；
- 实现前端降级和异步模块出错上报。

3. 优化原则

每个应用都要满足自己特定的需求，因为其商业条件、应用场景、用户期望，以及功能复杂性各不相同。尽管如此，如果应用必须对用户作出响应，那我们就必须从用户角度来考虑可感知的处理时间这个常量。事实上，虽然生活节奏越来越快（至少我们感觉如此），但人类的感知和反应时间则一直都没有变过。

图 3 表格展示了 Web 性能社区总结的经验法则：必须在 250 ms 内渲染页面，或者至少提供视觉反馈，才能保证用户不走开。如果想让人感觉很快，就必须在几百 ms 内响应用户操作。超过 1s，用户的预期流程就会中断，心思就会向其他任务转移，而超过 10s，除非你有反馈，否则用户基本上就会终止任务！

时间	感觉
0~100ms	很快
100~300ms	有一点点慢
300~1000ms	机械在工作呢
>1000ms	先干点别的吧
>10000ms	不能用了

图 3

此次的优化工作遵循以下四个原则：

- 首屏优先：精简和瘦身页面，首屏优先展示出来；
- 惰性交互：需用户交互的部分惰性加载；
- 惰性执行：能不执行的先别执行，惰性执行；
- 惰性滚屏：滚屏惰性加载。
- 遵循这四个原则，进行了优化工作。

4. 主要优化工作

（1）首屏优先

为了保证首屏优先展示，HTML 文档进行了适当精简。

目的：尽快渲染出页面并达到可交互的状态。

方法：

如果非必须，尽量只生成首屏需要的 HTML 数据。

优先获取资源、提前解析。如首屏需要的 CSS 和 JS；如果不考虑维护成本，可以把首屏需要的 CSS 和 JS 放到文档中。

发现和优先安排关键网络资源，尽早分派请求并取得页面。

文档精简后，服务端生成程序耗时短，性能才会好。

如图 4 所示，列表页的头、面包屑、品牌区、属性筛选区、60 个商品主图数据，这些是服务端模板渲染输出；而剩余部分是在前端 JS 惰性加载或生成。

(2) 惰性交互

惰性交互，即对需用户交互的部分进行惰性加载。

对于三级列表页品牌区，服务端只渲染 18 个品牌，用户在点更多时，AJAX 异步加载其他的。对于整个属性是筛选区服务端只渲染 5 行，其他行用户在点更多时，JS 从文档嵌入资源中取到数据，并渲染成 HTML。这样做可以保证服务端计算量少，提升服务端性能，减少数据传输。

如图 5，点“更多”时才加载更多的品牌，因为有些三级类目有非常多品牌，如果不采用这种方式，整个页面渲染非常慢。因为需要 SEO 的原因，京东三级列表页不能使用 BigPipe 等技术来进行更优的处理。

(3) 惰性执行



图 4 图 5

能不执行的先别执行，惰性执行。

图 6 是三级列表页最重要的商品区（商品主图 +N 个关联商品小图），每个商品的区域都是完全一样的；如果在服务端拼装整个商品区域的话，尤其涉及到小图部分，会有非常多的重复 HTML 元素。

我们把体验和减少页面内容进行了折中处理：服务端渲染输出商品主图部分；小图部分通过 Json 数据嵌入到页面，然后通过 JS 惰性执行渲染。这样可以很好地对页面进行瘦身。而且小图资源是页面嵌入的，非异步加载；没有网络请求。因此，用户基本感知不到异步带来的渲染闪动问题。

图 6 就是页面嵌入的小图 Json 数据。

（4）惰性滚屏

三级列表页的 60 个商品区域的图片和页尾都是当用户向下滚动页面时，才去加载当前屏幕中的图片和模块。这样可以节省服务器带宽和压力，提升页面整体渲染时间。

5. 细节优化工作

在实际优化过程中，还涉及到非常多的优化细节。

将一些 JS/CSS 资源直接嵌入页面。



图 6

把资源嵌入文档可以减少请求的次数。比如页面需要的 JS、CSS 数据。如图 7 所示。

图 7 中的这些 JS 对象，是后端渲染输出的，因此不适合放入单独的 JS 文件，直接在页面中嵌入输出会更好些。slaveWareList 是小图的列表对象。如果放在服务端模板渲染输出的话，首先需要进行一些循环拼装页面；另外会使页面体积变得非常大。



图 7

权衡之后决定放到前端 JS 渲染输出。这样也带来了一些好处：

- 减轻服务端压力，提升渲染模板性能和减少服务端执行时间；
- 服务端不用生成 HTML，文档减少上百个 div，减少页面大小和网络开销；
- 提前放到文档中，不用异步调用；
- 用户基本感知不到渲染过程。

对引入的资源排定优先次序

根据自己系统的业务，对每种资源定优先级：对必需的资源优先加载，而低优先级的请求保存在队列中延时加载或等待必需资源加载完再加载；如：搜索推荐热词、顶部三个热卖商品接口、60 个主商品的图片、价格优先加载。而对于库存、促销信息、广告词、预售商品、店铺信息等，延后加载。对于点击流，广告统计数据则延时两秒再加载。

应用 JS 缓存来存储公有属性和商品信息属性

三级列表页中的每个商品都是一个对象，存放在一个 Map 中，通过 AJAX 接口异步填充和维护商品的属性。用于后续用户交互用。同时维护

成本也会降低；即页面中用到的每个商品数据放入一个 map 中，如果没有则异步加载；如果有直接使用；即这些数据是公共数据（见图 8）。

AJAX接口最优调用

页面往往依赖很多的异步接口，因此要对异步接口进行压测，找出接口的最优调用方式。如京东三级列表页依赖价格、库存、广告词、店铺信息等异步调用接口。而页面有时候会出现多达 300 多个商品，如果用一个 get 请求把这些 sku 做参数，性能非常慢，那么就要采用分组分批调用。如页面商品在 300 个时，价格接口分六组，第一组 30 个，第二组 30 个，第三组 60 个，第四组 60 个，第五组 100 个，第六组 100 个。

DNS预解析

对可能的域名进行提前解析，避免将来 HTTP 请求时的 DNS 延迟。如对价格、库存、图片、单品页等服务预解析。

减少HTTP重定向

HTTP 重定向极费时间，特别是不同域名之间的重定向，更加费时；

```
> jdlist.slaveWareMap.values()
< ▼ Array[176] ⓘ
  ▼ [0 ... 99]
    ▼ 0: Object
      ad: "Apple产品618提前抢，好价不用等，6月1日起iPhone白条免息详情请点击"
      ▼ aos: Array[2]
        ▼ 0: Object
          n: "4.5-3.1英寸"
          v: "244_30815"
          ▶ __proto__: Object
        ▶ 1: Object
          length: 2
          ▶ __proto__: Array[0]
          comments: "404242"
          name: "Apple iPhone 5s (A1530) 16GB 深空灰色 移动联通4G手机"
          p: "2498.00"
          stock: "36"
          ▶ __proto__: Object
        ▶ 1: Object
```

图 8

这里面既有额外的 DNS 查询、TCP 握手，还有其他延迟。最佳的重定向次数为零。比如三级列表页以前是 `http://list.jd.com/9987-653-655.html`，而现在是 `http://list.jd.com/list.html?cat=9987,653,655`；在过渡期间可以重定向，但是过渡完成后就没必要重定向了。

使用CDN（内容分发网络）

把数据放到离用户地理位置更近的地方，可以显著减少每次 TCP 连接的网络延迟，增大吞吐量。比如京东三级列表页、商品详情页、公共 JS、CSS。

传输压缩过的内容（Gzip压缩）

传输前应该压缩应用资源，把要传输的字节减至最少：确保对每种要传输的资源采用最好的压缩手段。所有文本资源都应该使用 Gzip 压缩，然后再在客户端与服务端间传输。一般来说，Gzip 可以减少 60%~80% 的文件大小，也是一个相对简单（只要在服务器上配置一个选项），但优化效果较好的举措。（对于压缩级别，经过不同服务器多次压测，建议 Nginx 设置为 1-4）

去掉不必要的资源

任何请求都不如没有请求快，把一些非必须的或者可异步的，或者可延迟的尽量延迟请求。

在客户端缓存资源

应该缓存应用资源，从而避免每次请求都发送相同的内容。对静态资源 CSS/JS 或变化不频繁的 HTML 块，可以放到前端 `localStorage`。因为每次都传输一些不变的静态文件或者 HTML，实在是太浪费了。

无状态域名

Cookie 在很多应用中都是常见的性能瓶颈，很多开发者都会忽略

它给每次请求增加的额外负担；减少请求的 HTTP 首部数据（比如 HTTP cookie），节省的时间相当于几次往返的延迟时间。如列表页依赖的价格、库存接口，采用 3.cn 无状态域名，从而减少主域下 cookie 传输。

并行处理请求和响应

请求和响应的排队都会导致延迟，无论是客户端还是服务器端。这一点经常被忽视，但却会无谓地导致很长延迟。

域名分区

当页面中非常多请求都是一个域名下资源时，由于浏览器同时只能打开 6 个连接池，而且每个链接池是对不同域名起作用，所以很多请求一个域名会出现排队现象。如果把这些请求域名分区，让请求并行，从而加快资源下载。如：页面需要下载上百张图片，对图片进行域名分区调用。京东大部分页面都对图片进行了域名分区调用：

- <http://img10.360buyimg.com/>
- <http://img11.360buyimg.com/>
- <http://img12.360buyimg.com/>
- <http://img13.360buyimg.com/>
- <http://img14.360buyimg.com/>

拼合和连接

合并链接：把多个 JavaScript 或 CSS 文件组合为一个文件。

拼合：把多张图片组合为一个更大的复合的图片（CSS Sprites）。

服务端写相关信息到header

把服务器 IP 后两位写到 header，如果有问题，方便定位哪台服务器。

ups：后端路由的所有服务器都取到。把缓存命中信息或异常走兜底了，把后端运行状态写到header。Head-status：命中、未命中、异常等状态（见图9）。



图 9

6. 降级方案和异步模块出错上报功能的实现

降级方案

主动降级

页面依赖很多 AJAX 异步接口服务，难免保证这些服务从不出错。所以在调用这些接口服务时都提前判断该接口开关是否开启，如果开关关闭则不调用该接口服务。页面不展示相关模块。保证在一个接口服务出问题

被动降级

当某个异步接口服务返回非 200 状态码、请求超时、数据格式不正确等异常，就会被动隐藏或不展示相应模块。最上面三个热卖商品依赖的广告服务出问题时，会把每个三级分类对应的三个兜底商品展示出来，防止开天窗。对于其他模块因为是商品的属性，暂时做隐藏处理。

上报模块错误

当页面被动降级了，js 就会上报该模块，后台程序记录并报警。同

时也会上报 js 运行中出错的信息。记录什么浏览器，哪个版本，什么错误。我们会对这些问题验证和修改。保证每个用户都能访问。

Web性能监控

为什么要做 Web 性能监控，因为页面可能放在 CDN，前端 JS 执行很多业务逻辑不知道运行情况，整个链路网络偶尔不稳定、页面依赖的模块和第三方异步服务多人工难以实时监控等，这些情况请求还没有到后端就可能出问题，所以后端监控无能为力。

前端监控分两个方向：用 WebKit 内核模拟浏览器，定时抓取设定的页面；前端 JS 植入监控。

用WebKit内核模拟浏览器，定时抓取设定的页面

该 Web 监控项目采用一个中心服务，多个终端服务来完成大量页面抓取和校验。

部署到全国各个机房，实时监控页面是否打开正常（请求超时、返回非 200）、页面 HTML 关键元素是否丢失，页面是否出现乱码等。

每个终端定时向中心服务请求需要处理的页面 URL 和该页面需要验证的规则。如果验证不通过，则记录下来并报警。同时会保存现场（HTML 文档、页面截图）。

该项目在这次 618 起到很重要的作用，页面出现任何问题，都会提前检测出来。

前端JS植入监控

该 JS 统计页面白屏时间、首屏加载时间、每个 AJAX 异步方法调用耗时和请求状态码。

同时也会上报异步模块降级了，JS 运行中错误信息等。

埋点统计

京东列表页的埋点主要是来统计用户点击当前页面位置记数，帮助广告系统、业务、产品经理后续的工作。

埋点数据上报，就是通过 onclick 发送 AJAX 请求到后端服务。

其中对于点击后刷新当前页面的情况，需要在新页面记录上次点击的位置。因为在当前页面点击后上报 AJAX 方法还没执行就关闭当前窗口加载点击后的 URL 了。

下图是点击流插件的统计，数据敏感不做展示，大家只看功能（见图 10）。

7. 总结

用时

此次重构的时间段为：2014 年 12 月到 2015 年 4 月。

效果

京东三级列表页从优化到上线，已经经历了两个 618 和一个双 11 的考验，每天有上亿的访问量，页面打开时间在 20~80 毫秒（在某些地区或

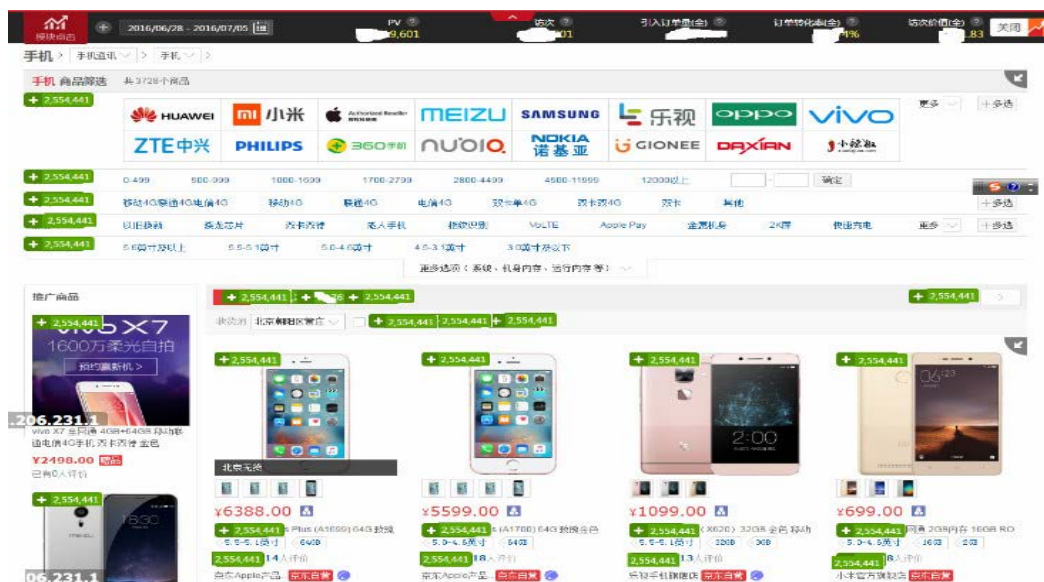


图 10

低带宽下会大于 100ms)。

后端方法调用 tp99 的性能数据如图 11 所示。

心得

列表页从开始 200+ms 到现在 100ms 内, QPS 单台机器几百到现在的近万, 页面从 1MB 到现在 200KB 内, 包扩后台系统的拆分, 逻辑算法后移、后台实时计算等优化。是需要有匠人的精神精雕细琢。

列表页每周都会根据业务方和产品经理的需求在开发功能。对于每个功能点都要深入思考, 列出多种方案, 最终选择一个简单、易维护、不影响系统性能、不降低用户体验的方案。这个过程要不断思考、或请教有这方面经验的人、包括参考外部公司的方案。有趣的是可能晚上突发奇想就有更好的方案。

中间也遇到无数的坑。对于每次遇到各种问题, 必须想方案避免再次

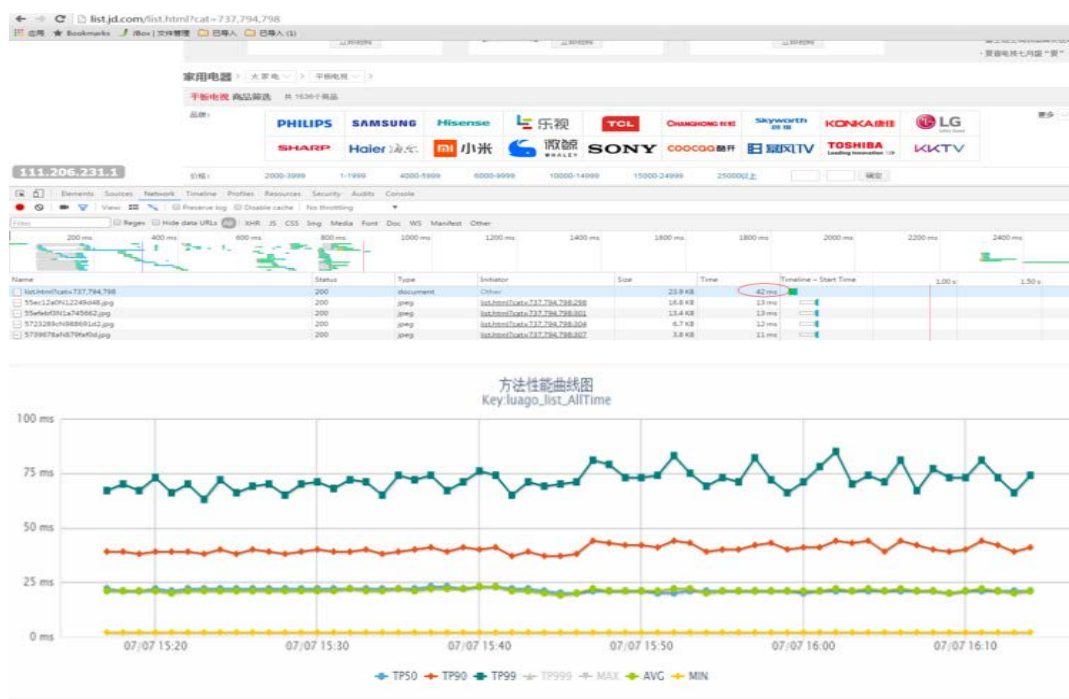


图 11

出现。同时要分析 Nginx 日志，分析每个请求，进而对爬虫、恶意参数访问、恶意请求做相应处理。这些都是前端服务必做的。当然后端服务也是非常重要的，后续会有列表页量身打造的缓存（加速、抗大流量、多样化兜底基础数据）、服务端架构、自动降级、架构高可用等方案。

作者简介

王向维，京东商城三级列表页架构师。工作期间，完成了京东三级列表页由 Node.js 版本到 Nginx+Lua 版本的变迁，并针对三级列表页前端即服务器端做了大量的优化工作。

在职技术人的一个学习方式

StuQ^{new}

About StuQ /关于StuQ

StuQ是极客邦科技旗下IT职业教育和服务平台，致力于帮助IT从业人员以及企业技术团队提升研发能力。

Personal Services /个人服务

为初入职场的技术人提供学习服务，通过企业一线的技术大咖实战经验传授，全面提升你的研发能力。

Corporate Services /企业服务

为技术团队提供学习服务，通过专业的课程和学习管理工具，帮助你的整个技术团队提升研发能力。



关注StuQ订阅号，获得优质学习资源，以及IT职业技能图谱

怎样才能叫高级程序员

作者 Brandon Hays 译者 足下

Stephen Tobolowsky 在定义联体三角形

“我真的开始对我在这里做的事情感觉不自信了。如果我们都不知道高级程序员到底是个什么样子，那我又该怎么朝这个目标努力？”

我们 Frontside 公司是习惯于每周二下午开个全公司例会的，会上大家谈谈上周取得的成绩，并为下一周订订计划。

在最近一次会议上，我们谈到了最近要招一位高级程序员，大家一谈到这个话题就都立刻激情爆发了。因为要提到对公司影响重大的事，非招聘新人加入团队莫属了，所以很自然的大家就开始各抒己见，热烈地讨论起我们要找的人到底应该具有什么样的资质来。

可是除了依靠直觉，一屋子的人里却没有一个能够把大家的想法归纳起来，到底要怎样才能叫做“高级”。

当一位同事说出了文章开始我引用的那段话之后，我意识到我们已经迷迷糊糊地碰上了一个对于整个公司来说都非常重要的问题：我们无法为我们想招聘的角色下一个定义，也不知道我们该怎样培养我们的程序员。

定义“高级程序员”的难题

就我个人来说，我是对“高级程序员”这个称号非常怀疑的，尤其因为当初在我有了9个月的正规编程经验，他们就为了给我涨工资而给了我这个称号之后。

事实上，如果你找来两个有经验的程序员，让他们分别描述一下他们心中的“高级”是个什么样子，我敢保证他们的答案会大相径庭。

“怎样才能叫高级程序员”这个问题其实非常依赖于语境，而且弹性空间非常大，以致于在我们这个行业里各个公司都可以给出任何自己需要的答案。

下面是一些身边人给出的我亲眼见到的关于“高级程序员”的定义：

- 有15年以上编程经验；
- 有2年编程经验并且有非常好的学习能力；
- 有1年使用一个非常热门的框架的经验，并且框架发布时间要超过一年；
- 一本技术书的作者；
- 可以在白板上默写出来某个计算机科学的算法；
- 写过一個开源库并且在公司里用起来了。

这些定义之间相差实在太远了。但想想在我们的生活中，很多东西都是没法下定义的，那又有什么问题呢？

为什么要费力下这个定义？凭直觉做判断不好吗

当大家在会议上说出这个困惑时，大家实际上说的是我们并没有非常清晰和可定义的标准来雇佣人、开除人和提拔人这个问题。大家说的对，事实上也就是这么混乱。

更糟的是，我们的核心使命——培养程序员——完不成了，因为我们

没办法帮他们设定出一条发展路线来。

“我一见到这个人我就知道他是个高级程序员”——这种说法揭示了另一个重大问题：“高级程序员”已经根深蒂固地成了一个偏见的有效载体。

把“高级程序员”作为供奉偏见的一种方法

当我们描述一个高级程序员应有的样子时，我们都是根据自己的经验和喜好来的，这就意味着这个词已经有了非常强的主观色彩。

当我们没有明确具体的标准，只能凭着直觉来判断一个人的资历的时候，我们就没有办法不带有偏见，但我们还是要做出判断。当一个人同时申请几份开发工作的时候，非常有可能有的公司认为他只是初级，有的会认为他是中级，还有的却认为他是高级，当然大家都不会直说自己是怎么判断的。

作为招聘经理，当我们做出判断的时候我们都会自认为非常正确，即使大家得到的结论相距甚远。

这样的结果就是不断被加强的偏见会阻止一些人进步，最终导致“头衔通货膨胀”。在当今技术界，各种偏见都不可避免的偏向白种男人，那么这种凭直觉做判断的体系就更多的会伤害女士和有色人种。

为什么大家还没有解决这个问题

首先给这个问题下定义就很难，因为它和工作环境的具体情况关系太大了。大多数公司领导人处理这个问题的办法都是走着瞧，而最终解决方案也都是“差不多”就行了。

解决这个问题也没有什么动力，因为当定下明确标准之后，公司领导人靠直觉做决定的权力很大程度上就会被剥夺了，而且还要为做出的决定负责。有谁会主动做一件让自己又要让出权力又要背上责任的事呢？

加上问责

我喜欢被问责，我也非常习惯。我懂得在为某件事负责任的同时，实际上我的自由度也是非常高的。就是否雇佣某个人这个问题来说，凭直觉下的决定往往比依据清晰的标准做出的决定更容易让人后悔。因为我们的直觉太容易受影响了，从我早上是不是忘了吃早餐，到那个人是不是能即兴谈起某个动画片，都有可能。

问责也为我们打开了改进之门。作为招聘经理，我的责任是打造一支有战斗力、快乐和能力互补的团队。要不断改进并且朝着这样的目标努力，可以靠直觉，可以全凭运气，但我们也可以创建一种先定义、再衡量、又问责反思然后再从头开始这样的循环，来保证我们通向最终目标。

问责可以帮助我们在通向未来的道路上完成从乘客到驾驶员的角色转换。

始于责任

现在问题变成了：我们怎样才能创造一种用于衡量资历的可度量的标准，而不是用那些有明显缺陷、象玩游戏一样的方法？

我能想到唯一相对公平的评判一个候选人的方式就是问几个关键问题：这个人的责任是什么？他是怎么完成任务的？他工作上需要什么样的帮助？

我们先从定义我们的情况开始了。当我们总结了 Frontside 的工作环境特征后，事情就开始变得清晰：

公司很小，所以每个人都要肩负多种责任，承担多种角色，并且要从头到尾的跟进解决问题。在我们这台机器上没有居中的传动齿轮。

我们依赖并打造内部社区的力量，以及我们参与的外部社区，尤其是

开源界。

我们在技术目标和代码可维护性可用性上追求极致。

于是团队成员的责任就非容易确定了：

- 可以为队友及客户提供清晰专业的技术和项目指导。
- 在内部和更大的编程社区里可以辅导他人、教授并且做出贡献。
- 可以很愉快的将软件交接给用户或者接手维护它的其他程序员。

所有这些责任构成了我们评估资历的标准基础。

联体三角形：简单的解释

我恰好最近有机会与好几家不同规模公司的负责人讨论了“高级”的定义，大家意见的共同点只有一个。

大家最简单的关于资历的共同解释就是：这个人需要多少指导？这个人能给别人提供多少指导（见图 1）？

我赞成这个“高级程序员的组合三角形”是一个不错的主意，可以简明的表达出内在含义，就象 Action Jack 的“成功的组合三角形”一样。

但即使这样的标准也是非常容易引入偏见的。它缺少一些重要的标准，并且过度强调了口碑这类容易见到的东西，以及解释深奥的计算机术语的能力。

在会上我们讨论出了一个新的框架

会上的热烈讨论有了一个非常酷的结果。在我试图灌输上面的三角形理论时，另一位员工提出了一个崭新的心理模形，吸引了大家的注意力。

她把我们在 Frontside 确定资历的方法描绘成了一个文氏图（用闭合的区域表示集合的图示法）。三个集合分别是：这个人有多强的独立工作能力及领导力？这个人技术实力如何？这个人和外部环境关系如何、有多

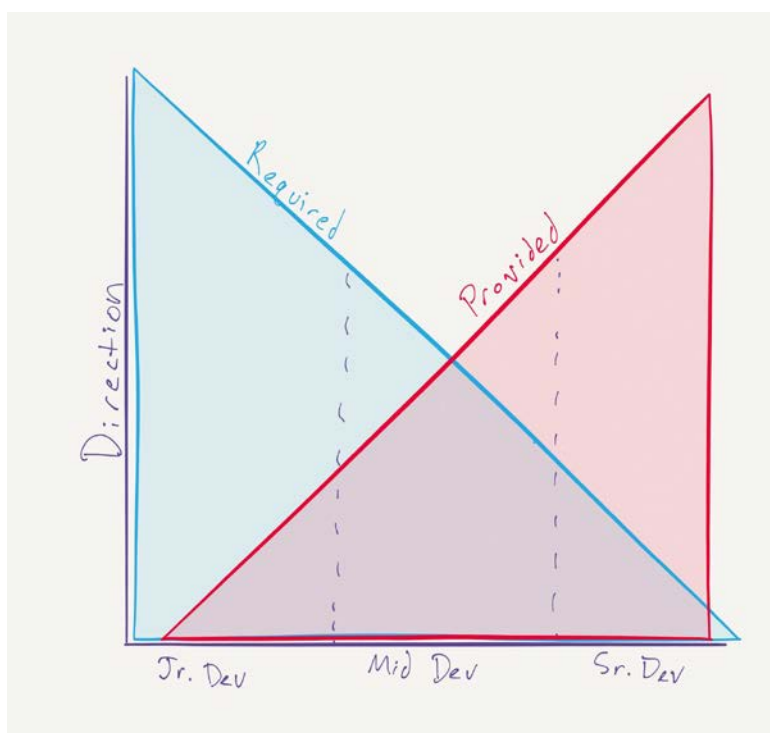


图 1

大贡献？

文氏图：更复杂的解释

在上文中我们已经把对资历的评估方法提升到了更高境界：“这个人需要多少指导？这个人可以给其他人提供多少指导？”但正像我们的员工指出来的，如果到此为止的话，还会有非常多令人困惑的地方。

那我们该怎样定义一位候选人到底能把他的本职工作做得多好？我们怎样能把评判标准引向一些具体的方面，而千万不要变成数学公式？

我们最终按照候选人要做的事总结出了三方面：技术能力、领导力和交际能力，并细化提炼出了 12 个特质（见图 2）。我将在下一篇文章中详细阐述这 12 个特质，但现在我可以简单说说。

三大方面

技术能力 (Technical capability)：技术能力强的人通常都对技术

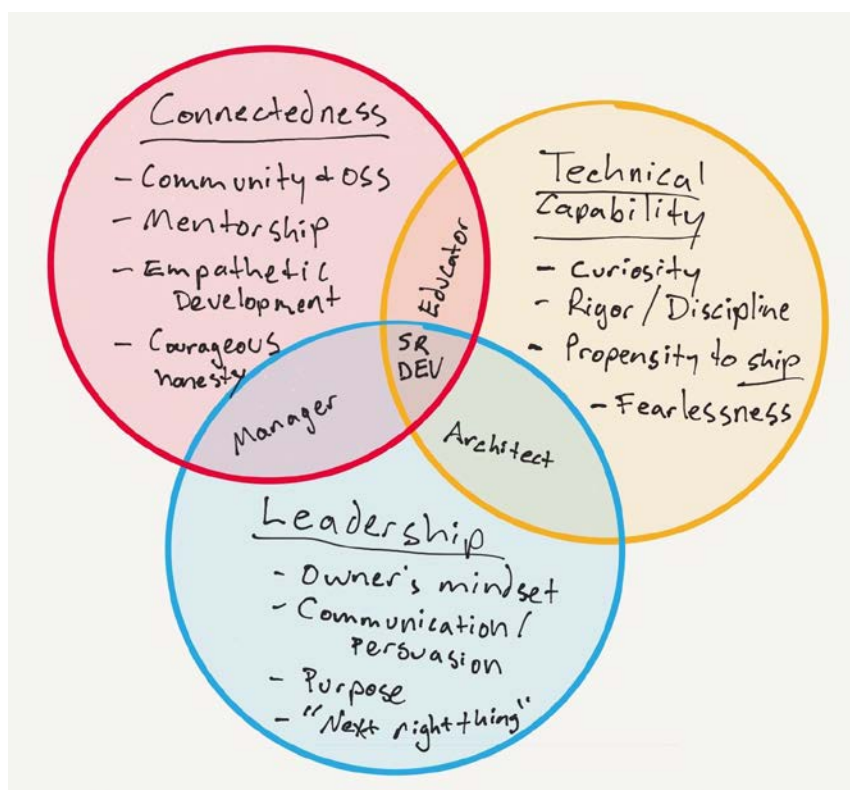


图 2

有浓厚兴趣，他们会不断钻研决不放弃，最终会做出可供经验不足的工程师使用、维护和学习的解决方案。

领导力（Leadership）：有领导力的人知道怎样为自己及别人发展并保持一种目的感。他们会指出公司里及自己职业生涯中出现的问题，并且揽到自己身上最终解决掉。

交际能力（Community/Connectedness）：交际能力强的人非常希望自己成为一个大集体中的一员，有非常强的奉献意识，身上有别人（同事、客户等）无法轻易描述的个人魅力，并且存在感非常强，生活充实快乐。

“对文化的适应能力” 怎么样

我们最初差点把交际能力叫做“对文化的适应能力”了，但我非常怀疑这个定义实际上是个扼杀思想的陈词滥调。“对文化的适应能力”就是

一个万金油，所有你想在程序员身上见到的可你又说不出的东西都可以用它往上套，而且这里面也非常容易藏入偏见。

当我们定义好了可以让 Frontside 的文化一致的标准之后，上面的观点就定义成了交际能力。

在三个不同方面衡量资质

还记得那三个方面吗？技术能力、领导力和交际能力，每个方面都有自己的从初级到高级的发展路线。

现在人们换职业都不是什么新鲜事了。很容易见到那些有很强领导力和交际能力但刚参加完代码训练营的人，他们的技术水平就只能被认为是一般。相反，一个经验丰富又受过正规培训的技术人员却有可能缺乏领导力和交际能力。

很少有人真的能在三方面都能达到高级水平，事实上也很少有人真的想在三方面都成为高级。我们 Frontside 把资质定义为这些方面的混合体，并努力帮助人们在他们想提高的方面获得进步。

证据：唯一能得到的衡量依据

衡量每个方面的资质都需要证据。如果你已经做了一些工作，那你手上应该已经有了一些证据。

我们将在下一篇文章中讨论这 12 个特质，每一个都有详细的标准，可以让候选人提供证据来说明他们经过时间的积累的确具有这些特质并且经验丰富。

但总的来说，如果在某个方面有一两项特别擅长和精通的特质的话，就可以认为他在那个方面是高级了。

比方说，假如某个人告诉你他的代码用好几种语言实现过，那在“技

术好奇心”这个特质上就可以得高分了。如果他还会非常严谨的为项目的核心代码写出全面、高质量的测试用例并用于持续集成，你就差不多可以认为他在技术能力上达到了高级水平。

或者如果某个人经常辅导别人、组织聚会，或者会做一些让大家过得更轻松的事，那我们就差不多可以在交际能力这方面给他打高分。

如果某个人曾经带过几个团队，那他就应该已经掌握了带团队的技巧。再加上挖掘问题根本原因的能力，那你就可以认为他在领导力的方向上达到高级了。

我们怎么定义“高级”

我们的衡量标准是如果某个人在技术能力上达到高级水平，他在领导力或交际能力中有一方面也能达到高级水平，我们就认为他是高级程序员了。如果他还想继续提高剩下的一方面，我们愿意提供帮助。

如果他是在领导力和交际能力都能达到高级水平，在技术方面能属于中高级的话，我们也认为是高级程序员。

举个一年前发生过的真实例子，我们雇佣了一个初级程序员，因为据我们评估，起码在最初的六个月中他需要非常多的指导。

到了第六个月，他的技术水平就已经达到中级了。到第一年结束时他就已经达到了高级水平。我敢这么说的原因是我们知道如果他离职，我们需要雇佣一个高级程序员来顶替他。

这样的事情为什么能发生？因为他是在我们公司起步的，而当时他已经在交际能力和领导力方面都可以达到高级水平了。所以他要在我们团队中做高级程序员的工作只是需要提高技术能力而已。

只看技术水平并不够

对于技术水平高但在领导力和交际能力方面都缺乏经验的人，不能直说“在我们这里你达不到高级程序员的标准”，这话太刺耳了。但对于他在团队中能承担的责任来说，我们可以暂时评订为中级，等他把另一方面或者两方面都提高了之后，我们再把他的提升为高级。

很多公司只根据技术水平来做判断，但这样对于我们这种小型的而且非常依赖合作模式工作的公司来说行不通。其实我非常担心那些只衡量技术能力的公司是认可“孤独的天才开发者”这样的危险想法的，觉得一个人技术水平高，就想当然的认为领导力和交际能力也很好。

在大公司中每个人都只负责一小部分工作，我非常乐于见到他们分享对于“高级程序员”的定义，那应该会在技术和非技术的方面都更加全面，让我们工作得效率更高，尤其是在需要与客户打交道的团队里。

成为高级需要多久

“高级程序员”是不是就意味着“若干年的经验”？事实上我并没有看到过哪个人不用五年就可以成为高级程序员的。要在很短的时间内就把一些特质发展得非常好来在某一方面达到高级水平其实是非常困难、甚至不可能的，更别说在多个方面全部成为高级了。

而且“五年经验”并不一定要意味着“五年的软件开发经验”。如果一个人已经在领导力和（或）交际能力上满足了条件，那他只需要提升技术能力，就已经可以发挥高级程序员的作用了。

我们招聘的“秘密武器”很大程度上源于我们观察到的事实：对于具有领导力和交际能力的人来说，要再提升技术能力并不需要很多时间，反之则不然。我见过很多这样的人，从代码集训营中出来两三年后就已经成了非常好的高级程序员。

更多要讨论的

这篇文章留下了非常多未能回答的问题。我们在这三个方面是用什么具体方法来评估候选人的能力和特质的？在面试前和面试中该怎么衡量呢？该如何把这些评估结果与一些具体的东西联系起来，比如工资？

这个框架又如何应用于非高级程序员？程序员们该什么时候升级？怎么升级？初级、中级和高级之间的区别是什么？它们之间差了些什么？这些词会不会实际上毫无意义而该被替换掉？

最后，如果真的可以的话，这个框架该如何应用于其他与 Frontside 有着非常大的文化和需求差异的公司？

在下一篇文章中我们会详细回答这些问题。

告别直觉

定义“高级”是一个仍在进行中的而且出乎人意料困难的过程，但我们还是要做这件事，因为它对我们非常重要。如果不能给“高级程序员”下一个清晰的定义，我们就迷失了培养员工的方向，就没有具体的办法来衡量要加入我们团队的人，也没有办法让员工相信我们可以信赖，更没办法来改进流程。

这个行业已经应该告别“我一见到这个人我就知道他是个高级程序员”这样下结论的年代了，我们该向着一些我们可以定义和分享的东西努力。让我们一起把开源的思路带到我们雇佣和发展员工上吧。

希望我能在下一篇文章中把留下的主要问题都解释清楚。如果你对文中内容有问题或者不同想法，你可以在 Twitter (@tehviking) 上找我，或者更好的方式是，你也把你的想法写出来，我会在我的文章里链接过去。

十八大专题 场场精彩

日新月异的移动架构

机器学习实战

社交网络与视频直播

微服务

容器

运维创造价值的时代

大数据处理及系统架构

...

北京 · 国际会议中心

2016年12月2日-12月3日

扫码查看更多精彩内容



想要了解更多？
进入官网查看精彩内容

bj2016.archsummit.com



架构师 月刊 2016年8月

本期主要内容：专访吕毅：链家网技术架构的演进之路，专访蘑菇街七公：25倍增长远非极限，优化需要偏执狂，为什么Android开发者应该使用FlatBuffers替代JSON？我为什么选择Angular 2？IBM、Google、Oracle三巨头的公有云之殇



云生态专刊 08

《云生态专刊》是InfoQ为大家推出的一个新产品，目标是“打造中国最优质的云生态媒体”。



顶尖技术团队访谈录 第六季

本次的《中国顶尖技术团队访谈录》·第六季挑选的六个团队虽然都来自互联网企业，却是风格各异。希望通过这样的记录，能够让一家家品牌背后的技术人员形象更加鲜活，让更多人感受到他们的可爱与坚持。



架构师特刊 大数据平台架构

本期技术特刊中我们总结了酷狗、美团、Airbnb的大数据平台架构实践范例，以及携程、IFTTT、卷皮等公司业务结合大数据平台的架构分析，希望读者能通过不同的角度从中收获到搭建大数据平台知识。