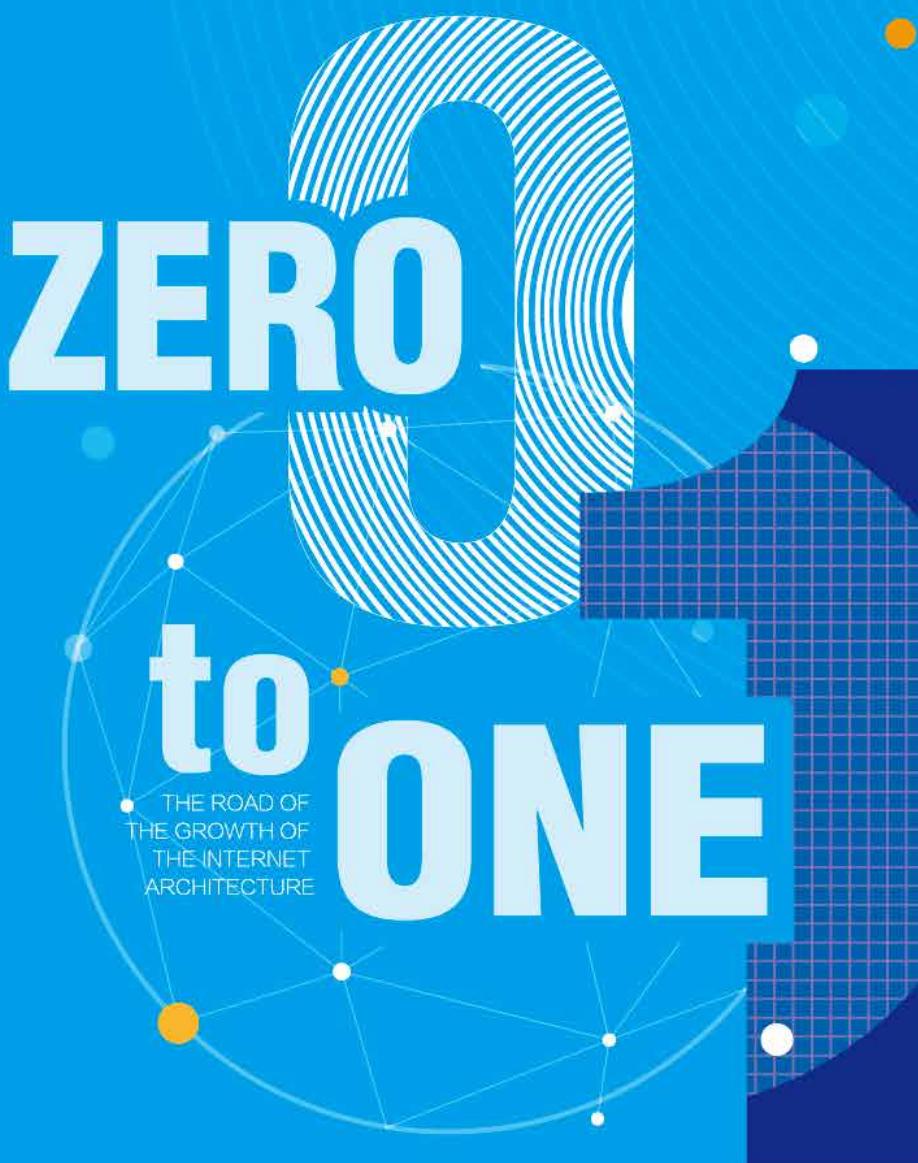


从0到1 互联网架构成长之路



第一期 2016年1月

InfoQ



ArchSummit深圳2016启动！

实践第一 案例为主

15大热门技术专题

- ▶ 研发体系构建管理
- ▶ 海量服务架构探索
- ▶ 技术创业
- ▶ 移动应用架构
- ▶ 高性能高效的运维体系构建
- ▶ 电商大促背后的技术较量
- ▶ 分享经济下的架构
- ▶ 云上的技术转变
- ▶ IoT，让世界更加智能
- ▶ 互联网金融
- ▶ 新产业，新技术
- ▶ 互联网安全
- ▶ 大数据与个性化
- ▶ 游戏
- ▶ 社交网络



扫描二维码 进入大会官网

2016年7月15-16日

中国·深圳 南山区
华侨城洲际酒店

旧金山 伦敦 北京 圣保罗 东京 纽约 上海
San Francisco London Beijing Sao Paulo Tokyo New York Shanghai

QCon

全球软件开发大会

2016年4月21-23日 | 北京·国际会议中心

主办方 **Geekbang > InfoQ**
极客邦科技



扫描获取更多大会信息

8折 优惠(截至2月21日)
现在报名, 团购可享更多优惠, 详情咨询: 010-64738142

崔 康 InfoQ 中国总编辑

卷 首 语

对于这本小册子，大家可能觉得有些意外。之前 InfoQ 发布的迷你书主要是《架构师》月刊，现在这本《ArchSummit 全球架构师峰会会刊》又是做什么的？

这是我们今年在运营大会品牌方面的思路转变的尝试。会刊将保持平均每月一期的发布频率，内容都是精选历届 ArchSummit 大会的讲师演讲，同时加上对于未来大会筹备的进展通报和讲师采访。希望以这种形式，让我们的读者对大会有一种持续和全面的了解，增强参与感。我们和参会者的关系，不再是一年两次的短暂交流，而是像朋友一样，定期的沟通



交流，会刊只是一种形式，目前启动的还包括架构周报、ArchSummit 互动交流群等，我们将第一时间把有关大会的精彩内容呈现给读者朋友。

会刊的基本原则是：“干货、案例、实用”，本期内容精选了历届大会讲师有关从 0 到 1 构建互联网技术架构的案例，这个话题在最近一次 ArchSummit 全球架构师峰会的讨论群中讨论地非常热烈，大家分别从商业模式、技术选型、流量变现等角度给出了精彩的观点，相信读者会从本期会刊中找到值得借鉴的经验。

目 录

从 0 到 100——知乎架构变迁史	6
从无到有：微信后台系统的演进之路	12
手机淘宝构架演化实践	25
春晚微信红包，是怎么扛住一百亿次请求的	32
百花齐放，锄其九九——Twitter 的技术坎坷之路	42

从 0 到 100——知乎架构变迁史



作者 藏秀涛

也许很多人还不知道，知乎在规模上是仅次于百度贴吧和豆瓣的中文互联网最大的UGC（用户生成内容）社区。知乎创业三年来，从0开始，到现在已经有了100多台服务器。目前知乎的注册用户超过了1100万，每个月有超过8000万人使用；网站每个月的PV超过2.2亿，差不多每秒钟的动态请求超过2500。

在ArchSummit全球架构师峰会上，知乎联合创始人兼CTO李申申带来了知乎创业三年多来的首次全面技术分享（[演讲视频](#)）。本文系根据演讲内容整理而成。

初期架构选型

在2010年10月真正开始动手做知乎这个产品时，包含李申申在内，最初只有两位工程师；到2010年12月份上线时，工程师是四个。

知乎的主力开发语言是Python。因为Python简单且强大，能够快速上手，开发效率

高，而且社区活跃，团队成员也比较喜欢。

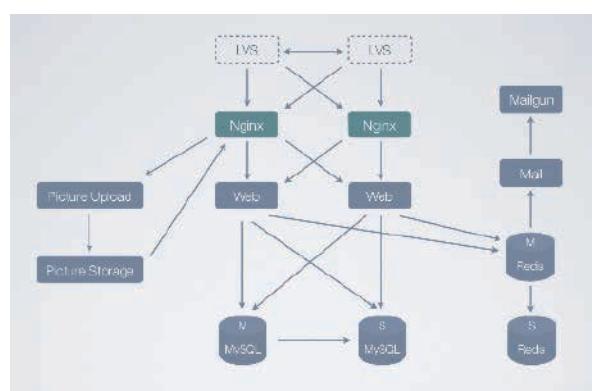
知乎使用的是Tornado框架。因为它支持异步，很适合做实时Comet应用，而且简单轻量，学习成本低，再就是有FriendFeed的成熟案例，Facebook的社区支持。知乎的产品有个特性，就是希望跟浏览器端建立一个长连接，

便于实时推送 Feed 和通知，所以 Tornado 比较合适。

最初整个团队的精力全部放在产品功能的开发上，而其他方面，基本上能节约时间、能省的都用最简单的方法来解决，当然这在后期也带来了一些问题。

最初的想法是用云主机，节省成本。知乎的第一台服务器是 512MB 内存的 Linode 主机。但是网站上线后，内测受欢迎程度超出预期，很多用户反馈网站很慢。跨国网络延迟比想象的要大，特别是国内的网络不均衡，全国各地用户访问的情况都不太一样。这个问题，再加上当时要做域名备案，知乎又回到了自己买机器找机房的老路上。

买了机器、找了机房之后又遇到了新的问题，服务经常宕掉。当时服务商的机器内存总是出问题，动不动就重启。终于有一次机器宕掉起不来了，这时知乎就做了 Web 和数据库的高可用。创业就是这样一件事情，永远不知道明早醒来的时候会面临什么样的问题。



这是当时那个阶段的架构图，Web 和数据库都做了主从。当时的图片服务托管在又拍云上。除了主从，为了性能更好还做了读写分离。

为解决同步问题，又添加了一个服务器来跑离线脚本，避免对线上服务造成响应延迟。另外，为改进内网的吞吐量延迟，还更换了设备，使整个内网的吞吐量翻了 20 倍。

在 2011 年上半年时，知乎对 Redis 已经很依赖。除了最开始的队列、搜索在用，后来像 Cache 也开始使用，单机存储成为瓶颈，所以引入了分片，同时做了一致性。

知乎团队是一个很相信工具的团队，相信工具可以提升效率。工具其实是一个过程，工具并没有所谓的最好的工具，只有最适合的工具。而且它是在整个过程中，随着整个状态的变化、环境的变化在不断发生变化的。知乎自己开发或使用过的工具包括 Profiling（函数级追踪请求，分析调优）、Werkzeug（方便调试的工具）、Puppet（配置管理）和 Shipit（一键上线或回滚）等。

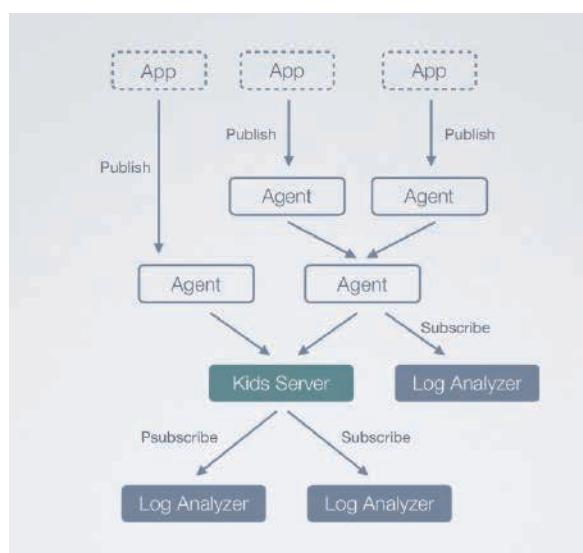
日志系统

知乎最初是邀请制的，2011 年下半年，知乎上线了申请注册，没有邀请码的用户也可以通过填写一些资料申请注册知乎。用户量又上了一个台阶，这时就有了一些发广告的账户，需要扫除广告。日志系统的需求提上日程。

这个日志系统必须支持分布式收集、集中存储、实时、可订阅和简单等特性。当时调研了一些开源系统，比如 Scribe 总体不错，但是不支持订阅。Kafka 是 Scala 开发的，但是团队在 Scala 方面积累较少，Flume 也是类似，而且比较重。所以开发团队选择了自己开发一个日

志系统——Kids (Kids Is Data Stream)。顾名思义，Kids 是用来汇集各种数据流的。

Kids 参考了 Scribe 的思路。Kdis 在每台服务器上可以配置成 Agent 或 Server。Agent 直接接受来自应用的消息，把消息汇集之后，可以打给下一个 Agent 或者直接打给中心 Server。订阅日志时，可以从 Server 上获取，也可以从中心节点的一些 Agent 上获取。



具体细节如下图所示：

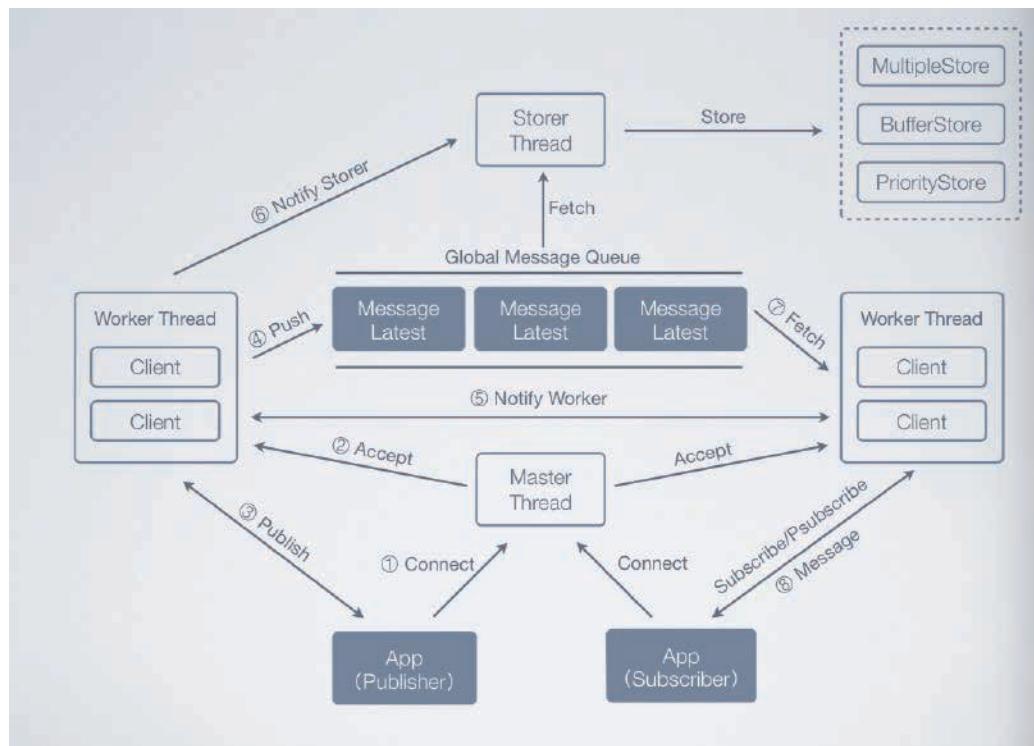
知乎还基于 Kids 做了一个 Web 小工具 (Kids Explorer)，支持实时看线上日志，现在已经成为调试线上问题最主要的工具。

Kids 已经开源，放到了 Github 上。

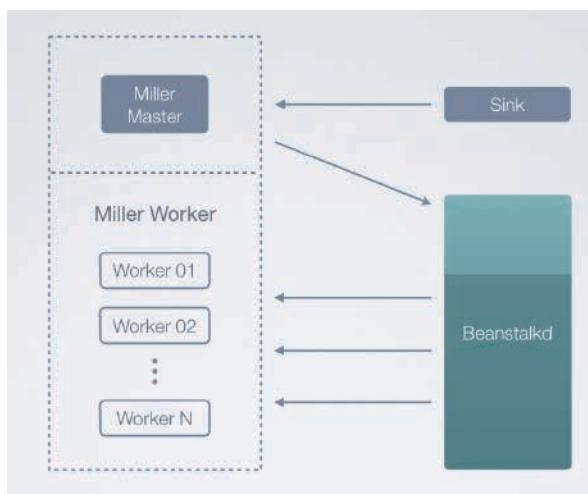
事件驱动的架构

知乎这个产品有一个特点，最早在添加一个答案后，后续的操作其实只有更新通知、更新动态。但是随着整个功能的增加，又多出了一些更新索引、更新计数、内容审查等操作，后续操作五花八门。如果按照传统方式，维护逻辑会越来越庞大，维护性也会非常差。这种场景很适合事件驱动方式，所以开发团队对整个架构做了调整，做了事件驱动的架构。

这时首先需要的是一个消息队列，它应该可以获取到各种各样的事件，而且对一致性有



很高的要求。针对这个需求，知乎开发了一个叫 Sink 的小工具。它拿到消息后，先做本地的保存、持久化，然后再把消息分发出去。如果那台机器挂掉了，重启时可以完整恢复，确保消息不会丢失。然后它通过 Miller 开发框架，把消息放到任务队列。Sink 更像是串行消息订阅服务，但任务需要并行化处理，Beanstalkd 就派上了用场，由其对任务进行全周期管理。架构如下图所示：



举例而言，如果现在有用户回答了问题，首先系统会把问题写到 MySQL 里面，把消息塞到 Sink，然后把问题返回给用户。Sink 通过 Miller 把任务发给 Beanstalkd，Worker 自己可以找到任务并处理。

最开始上线时，每秒钟有 10 个消息，然后有 70 个任务产生。现在每秒钟有 100 个事件，有 1500 个任务产生，就是通过现在的事件驱动架构支撑的。

页面渲染优化

知乎在 2013 年时每天有上百万的 PV，页面渲染其实是计算密集型的，另外因为要获取数据，所以也有 IO 密集型的特点。这时开发团队就对页面进行了组件化，还升级了数据获取机制。知乎按照整个页面组件树的结构，自上而下分层地获取数据，当上层的数据已经获取了，下层的数据就不需要再下去了，有几层基本上就有几次数据获取。

结合这个思路，知乎自己做了一套模板渲染开发框架——ZhihuNode。

经历了一系列改进之后，页面的性能大幅度提升。问题页面从 500ms 减少到 150ms，Feed 页面从 1s 减少到 600ms。

面向服务的架构（SOA）

随着知乎的功能越来越庞杂，整个系统也越来越大。知乎是怎么做的服务化呢？

首先需要一个最基本的 RPC 框架，RPC 框架也经历了好几版演进。

第一版是 Wish，它是一个严格定义序列化的模型。传输层用到了 STP，这是自己写的很简单的传输协议，跑在 TCP 上。一开始用的还不错，因为一开始只写了一两个服务。但是随着服务增多，一些问题开始出现，首先是 ProtocolBuffer 会生成一些描述代码，很冗长，

放到整个库里显得很丑陋。另外严格的定义使其不便使用。这时有位工程师开发了新的 RPC 框架——Snow。它使用简单的 JSON 做数据序列化。但是松散的数据定义面对的问题是，比如说服务要去升级，要改写数据结构，很难知道有哪几个服务在使用，也很难通知它们，往往错误就发生了。于是又出了第三个 RPC 框架，写 RPC 框架的工程师，希望结合前面两个框架的特点，首先保持 Snow 简单，其次需要相对严格的序列化协议。这一版本引入了 Apache Avro。同时加入了特别的机制，在传输层和序列化协议这一层都做成了可插拔的方式，既可以用 JSON，也可以用 Avro，传输层可以用 STP，也可以用二进制协议。

再就是搭了一个服务注册发现，只需要简

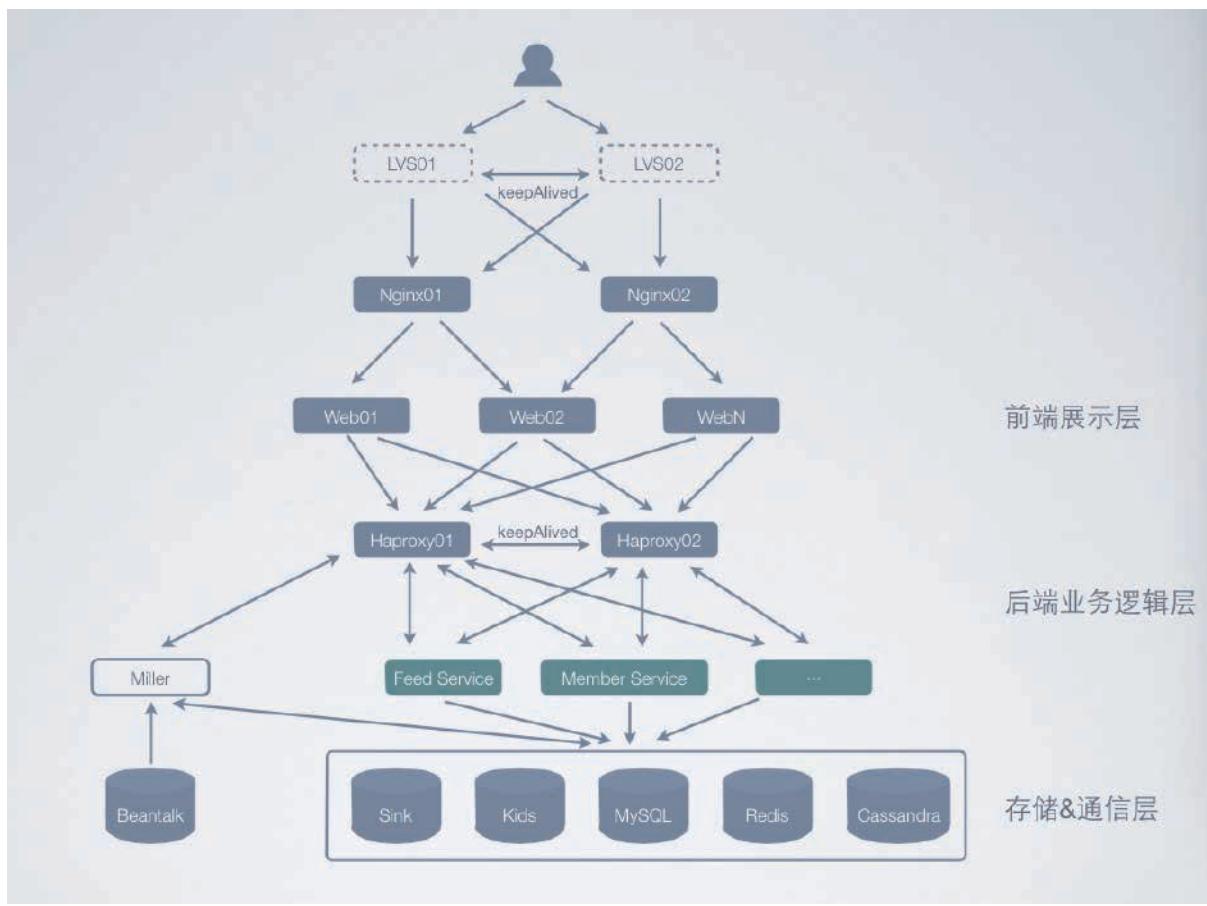
单的定义服务的名字就可以找到服务在哪台机器上。同时，知乎也有相应的调优的工具，基于 Zipkin 开发了自己的 Tracing 系统。

按照调用关系，知乎的服务分成了 3 层：聚合层、内容层和基础层。按属性又可以分成 3 类：数据服务、逻辑服务和通道服务。数据服务主要是一些要做特殊数据类型的存储，比如图片服务。逻辑服务更多的是 CPU 密集、计算密集的操作，比如答案格式的定义、解析等。通道服务的特点是没有存储，更多是做一个转发，比如说 Sink。

这是引入服务化之后整体的架构。

演讲中还介绍了基于 AngularJS 开发知乎专栏的新实践，感兴趣的读者可以观看视频。





“ArchSummit 会员微信群”是一个基于专家运营的高端技术社区。群内定期举行线上线下专家讲座、话题讨论，专注于“分享、交流、成长”。微信添加群主 cuikang10，申请入群。

讲师介绍：李申申，现任知乎联合创始人兼 CTO。职业经历中大部分时间在创业，在创办知乎和上一家公司 Meta 搜索之间，曾在爱奇艺短期担任高级工程师；在 Meta 搜索创业期间担任技术总监；这之前担任过 Icebreaker 高级工程师。李申申最早专业为汽车设计制造，硕士时期转入计算机科学与技术专业。从多年的创业经历中磨练出「全栈工程师」的综合实践经验，从产品规划设计到前后端开发，再到服务器部署运维。

从无到有：微信后台系统的演进之路



作者 张文瑞

作者介绍：张文瑞，微信高级工程师，微信接入系统负责人，一直从事后台系统设计开发，早期涉足传统行业软件，后投身互联网。作为微信最早的后台开发之一，见证了微信从零开始到逐渐发展壮大的过程

从无到有

2011.1.21 微信正式发布。这一天距离微信项目启动日约为 2 个月。就在这 2 个月里，微信从无到有，大家可能会好奇这期间微信后台做的最重要的事情是什么？

我想应该是以下三件事：

1. 确定了微信的消息模型

微信起初定位是一个通讯工具，作为通讯工具最核心的功能是收发消息。微信团队源于广广团队，消息模型跟邮箱的邮件模型也很有渊源，都是存储转发。



图 1 微信消息模型

图 1 展示了这一消息模型，消息被发出后，会先在后台临时存储；为使接收者能更快接收到消息，会推送消息通知给接收者；最后客户端主动到服务器收取消息。

2. 制定了数据同步协议

由于用户的帐户、联系人和消息等数据都在服务器存储，如何将数据同步到客户端就成了很关键的问题。为简化协议，我们决定通过一个统一的数据同步协议来同步用户所有的基

础数据。

最初方案是客户端记录一个本地数据的快照 (Snapshot)，需要同步数据时，将 Snapshot 带到服务器，服务器通过计算 Snapshot 与服务器数据的差异，将差异数据发给客户端，客户端再保存差异数据完成同步。不过这个方案有两个问题：一是 Snapshot 会随着客户端数据的增多变得越来越大，同步时流量开销大；二是客户端每次同步都要计算 Snapshot，会带来额外的性能开销和实现复杂度。

几经讨论后，方案改为由服务计算 Snapshot，在客户端同步数据时跟随数据一起下发给客户端，客户端无需理解 Snapshot，只需存储起来，在下次数据同步数据时带上即可。同时，Snapshot 被设计得非常精简，是若干个 Key-Value 的组合，Key 代表数据的类型，Value 代表给到客户端的数据的最新版本号。Key 有三个，分别代表：帐户数据、联系人和消息。这个同步协议的一个额外好处是客户端同步完数据后，不需要额外的 ACK 协议来确认数据收取成功，同样可以保证不会丢数据：只要客户端拿最新的 Snapshot 到服务器做数据同步，服务器即可确认上次数据已经成功同步完成，可以执行后续操作，例如清除暂存在服务的消息等等。

此后，精简方案、减少流量开销、尽量由服务器完成较复杂的业务逻辑、降低客户端实现的复杂度就作为重要的指导原则，持续影响着后续的微信设计开发。记得有个比较经典的案例是：我们在微信 1.2 版实现了群聊功能，但为了保证新旧版客户端间的群聊体验，我们通过服务器适配，让 1.0 版客户端也能参与群聊。

3. 定型了后台架构



图 2 微信后台系统架构

微信后台使用三层架构：接入层、逻辑层和存储层。

- 接入层提供接入服务，包括长连接接入服务和短连接接入服务。长连接接入服务同时支持客户端主动发起请求和服务器主动发起推送；短连接接入服务则只支持客户端主动发起请求。
- 逻辑层包括业务逻辑服务和基础逻辑服务。业务逻辑服务封装了业务逻辑，是后台提供给微信客户端调用的 API。基础逻辑服务则抽象了更底层和通用的业务逻辑，提供给业务逻辑服务访问。
- 存储层包括数据访问服务和数据存储服务。数据存储服务通过 MySQL 和 SDB（广研早期后台中广泛使用的 Key-Table 数据存储系统）等底层存储系统来持久化用户数据。数据访问服务适配并路由数据访问请求到不同的底层数据存储服务，面向逻辑层提供结构化的数据服务。比较特别的是，微信后台每一种不同类

型的数据都使用单独的数据访问服务和数据存储服务，例如帐户、消息和联系人等等都是独立的。

微信后台主要使用 C++。后台服务使用 Svrkit 框架搭建，服务之间通过同步 RPC 进行通讯。

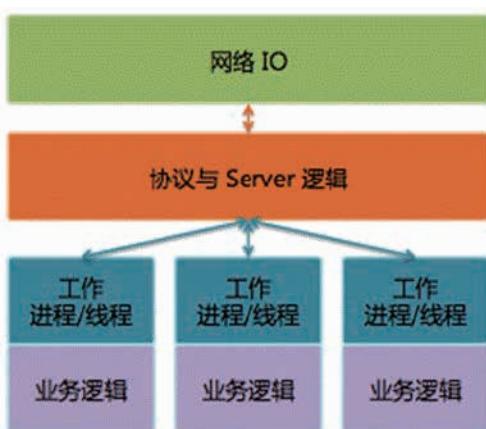


图 3 Svrkit 框架

Svrkit 是另一个广 去后就已经存在的高性能 RPC 框架，当时尚未广泛使用，但在微信后台却大放异彩。作为微信后台基础设施中最重要的部分，Svrkit 这几年一直不断在进化。我们使用 Svrkit 构建了数以千计的服务模块，提供数万个服务接口，每天 RPC 调用次数达几十万亿次。

这三件事影响深远，以至于 5 年后的今天，我们仍继续沿用最初的架构和协议，甚至还可以支持当初 1.0 版的微信客户端。

这里有一个经验教训——运营支撑系统真的很重要。第一个版本的微信后台是仓促完成的，当时只是完成了基础业务功能，并没有配套的业务数据统计等等。我们在开放注册后，一时间竟没有业务监控页面和数据曲线可以看，

注册用户数是临时从数据库统计的，在线数是从日志里提取出来的，这些数据通过每个小时运行一次的脚本（这个脚本也是当天临时加的）统计出来，然后自动发邮件到邮件组。还有其他各种业务数据也通过邮件进行发布，可以说邮件是微信初期最重要的数据门户。

2011.1.21 当天最高并发在线数是 491，而今天这个数字是 4 亿。

小步慢跑

在微信发布后的 4 个多月里，我们经历了发布后火爆注册的惊喜，也经历了随后一直不温不火的困惑。

这一时期，微信做了很多旨在增加用户好友量，让用户聊得起来的功能。打通腾讯微博私信、群聊、工作邮箱、QQ/ 邮箱好友推荐等等。对于后台而言，比较重要的变化就是这些功能催生了对异步队列的需求。例如，微博私信需要跟外部门对接，不同系统间的处理耗时和速度不一样，可以通过队列进行缓冲；群聊是耗时操作，消息发到群后，可以通过异步队列来异步完成消息的扩散写等等。

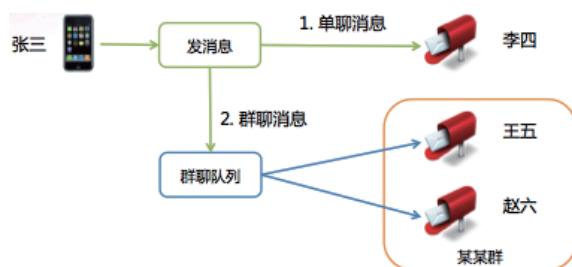


图 4 单聊和群聊消息发送过程

图 4 是异步队列在群聊中的应用。微信的

群聊是写扩散的，也就是说发到群里的一条消息会给群里的每个人都存一份（消息索引）。为什么不是读扩散呢？有两个原因：

- 群的人数不多，群人数上限是 10（后来逐步加到 20、40、100，目前是 500），扩散的成本不是太大，不像微博，有成千上万的粉丝，发一条微博后，每粉丝都存一份的话，一个是效率太低，另一个存储量也会大很多；
- 消息扩散写到每个人的消息存储（消息收件箱）后，接收者到后台同步数据时，只需要检查自己收件箱即可，同步逻辑跟单聊消息是一致的，这样可以统一数据同步流程，实现起来也会很轻量。

异步队列作为后台数据交互的一种重要模式，成为了同步 RPC 服务调用之外的有力补充，在微信后台被大量使用。

快速成长

微信的飞速发展是从 2.0 版开始的，这个版本发布了语音聊天功能。之后微信用户量急速增长，2011.5 用户量破 100 万、2011.7 用户量破 1000 万、2012.3 注册用户数突破 1 亿。

伴随着喜人成绩而来的，还有一堆幸福的烦恼。

- 业务快速迭代的压力

微信发布时功能很简单，主要功能就是发消息。不过在发语音之后的几个版本里迅速推出了手机通讯录、QQ 离线消息、查看附近的人、摇一摇、漂流瓶和朋友圈等等功能。

有个广为流传的关于朋友圈开发的传奇——朋友圈历经 4 个月，前后做了 30 多个版本迭代才最终成型。其实还有一个鲜为人知的故事——那时候因为人员比较短缺，朋友圈后台长时间只有 1 位开发人员。

- 后台稳定性的要求

用户多了，功能也多了，后台模块数和机器量在不断翻番，紧跟着的还有各种故障。帮助我们顺利度过这个阶段的，是以下几个举措：

1. 极简设计

虽然各种需求扑面而来，但我们每个实现方案都是一丝不苟完成的。实现需求最大的困难不是设计出一个方案并实现出来，而是需要在若干个可能的方案中，甄选出最简单实用的那个。

这中间往往需要经过几轮思考——讨论——推翻的迭代过程，谋定而后动有不少好处，一方面可以避免做出华而不实的过度设计，提升效率；另一方面，通过详尽的讨论出来的看似简单的方案，细节考究，往往是可靠性最好的方案。

2. 大系统小做

逻辑层的业务逻辑服务最早只有一个服务模块（我们称之为 mmweb），囊括了所有提供给客户端访问的 API，甚至还有一个完整的微信官网。这个模块架构类似 Apache，由一个 CGI 容器（CGIHost）和若干 CGI 组成（每个 CGI 即为一个 API），不同之处在于每个 CGI 都

是一个动态库 so，由 CGIHost 动态加载。

在 mmweb 的 CGI 数量相对较少的时候，这个模块的架构完全能满足要求，但当功能迭代加快，CGI 量不断增多之后，开始出现问题：

1) 每个 CGI 都是动态库，在某些 CGI 的共用逻辑的接口定义发生变化时，不同时期更新上线的 CGI 可能使用了不同版本的逻辑接口定义，会导致在运行时出现诡异结果或者进程 crash，而且非常难以定位；

2) 所有 CGI 放在一起，每次大版本发布上线，从测试到灰度再到全面部署完毕，都是一个很漫长的过程，几乎所有后台开发人员都会被同时卡在这个环节，非常影响效率；

3) 新增的不太重要的 CGI 有时稳定性不好，某些异常分支下会 crash，导致 CGIHost 进程无法服务，发消息这些重要 CGI 受影响没法运行。

于是我们开始尝试使用一种新的 CGI 架构——Logicsvr。

Logicsvr 基于 Svrkit 框架。将 Svrkit 框架和 CGI 逻辑通过静态编译生成可直接使用 HTTP 访问的 Logicsvr。我们将 mmweb 模块拆分为 8 个不同服务模块。拆分原则是：实现不同业务功能的 CGI 被拆到不同 Logicsvr，同一功能但是重要程度不一样的也进行拆分。例如，作为核心功能的消息收发逻辑，就被拆为 3 个服务模块：消息同步、发文本和语音消息、发图片和视频消息。

每个 Logicsvr 都是一个独立的二进制程序，可以分开部署、独立上线。时至今日，微信后台有数十个 Logicsvr，提供了数百个 CGI 服务，部署在数千台服务器上，每日客户端访问量几

千亿次。

除了 API 服务外，其他后台服务模块也遵循“大系统小做”这一实践准则，微信后台服务模块数从微信发布时的约 10 个模块，迅速上涨到数百个模块。

3. 业务监控

这一时期，后台故障很多。比故障更麻烦的是，因为监控的缺失，经常有些故障我们没法第一时间发现，造成故障影响面被放大。

监控的缺失一方面是因为在快速迭代过程中，重视功能开发，轻视了业务监控的重要性，有故障一直是兵来将挡水来土掩；另一方面是基础设施对业务逻辑监控的支持度较弱。基础设施提供了机器资源监控和 Svrkit 服务运行状态的监控。这个是每台机器、每个服务标配的，无需额外开发，但是业务逻辑的监控就要麻烦得多了。当时的业务逻辑监控是通过业务逻辑统计功能来做的，实现一个监控需要 4 步：

- 1) 申请日志上报资源；
- 2) 在业务逻辑中加入日志上报点，日志会被每台机器上的 agent 收集并上传到统计中心；
- 3) 开发统计代码；
- 4) 实现统计监控页面。

可以想象，这种费时费力的模式会反过来降低开发人员对加入业务监控的积极性。于是有一天，我们去公司内的标杆——即通后台（QQ 后台）取经了，发现解决方案出乎意料地简单且强大：

- 1) 故障报告

之前每次故障后，是由 QA 牵头出一份故

障报告，着重点是对故障影响的评估和故障定级。新的做法是每个故障不分大小，开发人员需要彻底复盘故障过程，然后商定解决方案，补充出一份详细的技术报告。这份报告侧重于：如何避免同类型故障再次发生、提高故障主动发现能力、缩短故障响应和处理过程。

2) 基于 ID-Value 的业务无关的监控告警体系

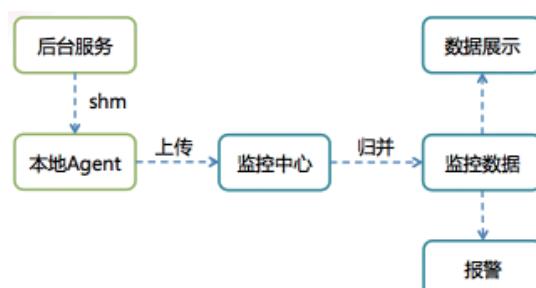


图 5 基于 ID-Value 的监控告警体系

监控体系实现思路非常简单，提供了 2 个 API，允许业务代码在共享内存中对某个监控 ID 进行设置 Value 或累加 Value 的功能。每台机器上的 Agent 会定时将所有 ID-Value 上报到监控中心，监控中心对数据汇总入库后就可以通过统一的监控页面输出监控曲线，并通过预先配置的监控规则产生报警。

对于业务代码来说，只需在要被监控的业务流程中调用一下监控 API，并配置好告警条件即可。这就极大地降低了开发监控报警的成本，我们补全了各种监控项，让我们能主动及时地发现问题。新开发的功能也会预先加入相关监控项，以便在少量灰度阶段就能直接通过监控曲线了解业务是否符合预期。

4. KV Svr

微信后台每个存储服务都有自己独立的存储模块，是相互独立的。每个存储服务都有一个业务访问模块和一个底层存储模块组成。业务访问层隔离业务逻辑层和底层存储，提供基于 RPC 的数据访问接口；底层存储有两类：SDB 和 MySQL。

SDB 适用于以用户 UIN(uint32_t) 为 Key 的数据存储，比方说消息索引和联系人。优点是性能高，在可靠性上，提供基于异步流水同步的 Master-Slave 模式，Master 故障时，Slave 可以提供读数据服务，无法写入新数据。

由于微信账号为字母 + 数字组合，无法直接作为 SDB 的 Key，所以微信帐号数据并非使用 SDB，而是用 MySQL 存储的。MySQL 也使用基于异步流水复制的 Master-Slave 模式。

第 1 版的帐号存储服务使用 Master-Slave 各 1 台。Master 提供读写功能，Slave 不提供服务，仅用于备份。当 Master 有故障时，人工切换服务到 Slave，无法提供写服务。为提升访问效率，我们还在业务访问模块中加入了 memcached 提供 Cache 服务，减少对底层存储访问。

第 2 版的帐号存储服务还是 Master-Slave 各 1 台，区别是 Slave 可以提供读服务，但有可能读到脏数据，因此对一致性要求高的业务逻辑，例如注册和登录逻辑只允许访问 Master。当 Master 有故障时，同样只能提供读服务，无法提供写服务。

第 3 版的帐号存储服务采用 1 个 Master 和

多个 Slave，解决了读服务的水平扩展能力。

第 4 版的帐号服务底层存储采用多个 Master-Slave 组，每组由 1 个 Master 和多个 Slave 组成，解决了写服务能力不足时的水平扩展能力。

最后还有个未解决的问题：单个 Master-Slave 分组中，Master 还是单点，无法提供实时的写容灾，也就意味着无法消除单点故障。另外 Master-Slave 的流水同步延时对读服务有很大影响，流水出现较大延时会导致业务故障。于是我们寻求一个可以提供高性能、具备读写水平扩展、没有单点故障、可同时具备读写容灾能力、能提供强一致性保证的底层存储解决方案，最终 KVSvr 应运而生。

KVSvr 使用基于 Quorum 的分布式数据强一致性算法，提供 Key-Value/Key-Table 模型的存储服务。传统 Quorum 算法的性能不高，KVSvr 创造性地将数据的版本和数据本身做了区分，将 Quorum 算法应用到数据的版本的协商，再通过基于流水同步的异步数据复制提供了数据强一致性保证和极高的数据写入性能，另外 KVSvr 天然具备数据的 Cache 能力，可以提供高效的读取性能。

KVSvr 一举解决了我们当时迫切需要的无单点故障的容灾能力。除了第 5 版的帐号服务外，很快所有 SDB 底层存储模块和大部分 MySQL 底层存储模块都切换到 KVSvr。随着业务的发展，KVSvr 也在不断进化着，还配合业务需要衍生出了各种定制版本。现在的 KVSvr 仍然作为核心存储，发挥着举足轻重的作用。

平台化

2011.8 深圳举行大运会。微信推出“微信深圳大运志愿者服务中心”服务号，微信用户可以搜索“szdy”将这个服务号加为好友，获取大会相关的资讯。当时后台对“szdy”做了特殊处理，用户搜索时，会随机返回“szdy01”，“szdy02”，…，“szdy10”这 10 个微信号中的 1 个，每个微信号背后都有一个志愿者在服务。

2011.9 “微成都”落户微信平台，微信用户可以搜索“wechengdu”加好友，成都市民还可以在“附近的人”看到这个号，我们在后台给这个帐号做了一些特殊逻辑，可以支持后台自动回复用户发的消息。

这种需求越来越多，我们就开始做一个媒体平台，这个平台后来从微信后台分出，演变成了微信公众平台，独立发展壮大，开始了微信的平台化之路。除微信公众平台外，微信后台的外围还陆续出现了微信支付平台、硬件平台等一系列平台。

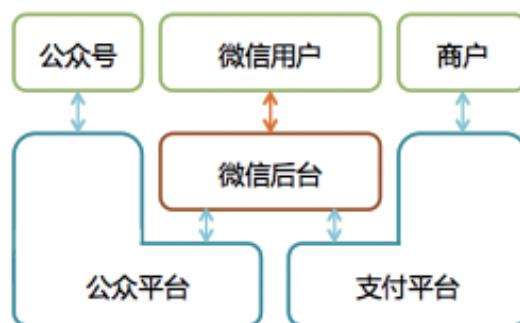


图 6 微信平台

走出国门

微信走出国门的尝试开始于 3.0 版本。从这个版本开始，微信逐步支持繁体、英文等多种语言文字。不过，真正标志性的事情是第一个海外数据中心的投入使用。

1. 海外数据中心

海外数据中心的定位是一个自治的系统，也就是说具备完整功能，能够不依赖于国内数据中心独立运作。

1) 多数据中心架构

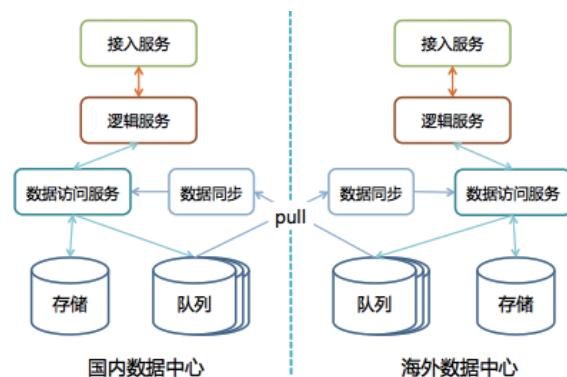


图 7 多数据中心架构

系统自治对于无状态的接入层和逻辑层来说很简单，所有服务模块在海外数据中心部署一套就行了。

但是存储层就有很大麻烦了——我们需要确保国内数据中心和海外数据中心能独立运作，但不是两套隔离的系统各自部署，各玩各的，而是一套业务功能可以完全互通的系统。因此我们的任务是需要保证两个数据中心的数据一致性，另外 Master-Master 架构是个必选项，也即两个数据中心都需要可写。

2) Master-Master 存储架构

Master-Master 架构下数据的一致性是个很大的问题。两个数据中心之间是个高延时的网络，意味着在数据中心之间直接使用 Paxos 算法、或直接部署基于 Quorum 的 KVServer 等看似一劳永逸的方案不适用。

最终我们选择了跟 Yahoo! 的 PNUTS 系统类似的解决方案，需要对用户集合进行切分，国内用户以国内上海数据中心为 Master，所有数据写操作必须回到国内数据中心完成；海外用户以海外数据中心为 Master，写操作只能在海外数据中心进行。从整体存储上看，这是一个 Master-Master 的架构，但细到一个具体用户的数据，则是 Master-Slave 模式，每条数据只能在用户归属的数据中心可写，再异步复制到其他数据中心。

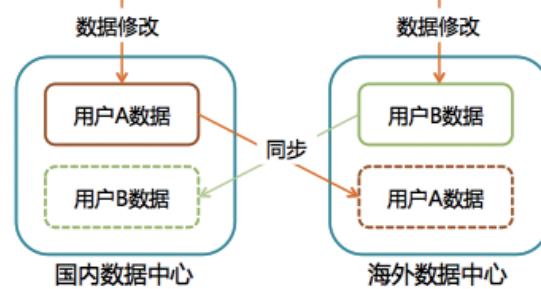


图 8 多数据中心的数据 Master-Master 架构

3) 数据中心间的数据一致性

这个 Master-Master 架构可以在不同数据中心间实现数据最终一致性。如何保证业务逻辑在这种数据弱一致性保证下不会出现问题？

这个问题可以被分解为 2 个子问题：

- * 用户访问自己的数据

用户可以满世界跑，那是否允许用户就近接入数据中心就对业务处理流程有很大影响。

如果允许就近接入，同时还要保证数据一致性不影响业务，就意味着要么用户数据的 Master 需要可以动态的改变；要么需要对所有业务逻辑进行仔细梳理，严格区分本数据中心和跨数据中心用户的请求，将请求路由到正确的数据中心处理。

考虑到上述问题会带来很高昂的实现和维护的复杂度，我们限制了每个用户只能接入其归属数据中心进行操作。如果用户发生漫游，其漫游到的数据中心会自动引导用户重新连回归属数据中心。

这样用户访问自己数据的一致性问题就迎刃而解了，因为所有操作被限制在归属数据中心内，其数据是有强一致性保证的。此外，还有额外的好处：用户自己的数据（如：消息和联系人等）不需要在数据中心间同步，这就大大降低了对数据同步的带宽需求。

- 用户访问其他用户的数据

由于不同数据中心之间业务需要互通，用户会使用到其他数据中心用户创建的数据。例如，参与其他数据中心用户创建的群聊，查看其他数据中心用户的朋友圈等。

仔细分析后可以发现，大部分场景下对数据一致性要求其实并不高。用户稍迟些才见到自己被加入某个其他数据中心用户建的群、稍迟些才见到某个好友的朋友圈动态更新其实并不会带来什么问题。在这些场景下，业务逻辑直接访问本数据中心的数据。

当然，还是有些场景对数据一致性要求很高。比方说给自己设置微信号，而微信号是需要在整个微信帐号体系里保证唯一的。我们提

供了全局唯一的微信号申请服务来解决这一问题，所有数据中心通过这个服务申请微信号。这种需要特殊处置的场景极少，不会带来太大问题。

4) 可靠的数据同步

数据中心之间有大量的数据同步，数据是否能够达到最终一致，取决于数据同步是否可靠。为保证数据同步的可靠性，提升同步的可用性，我们又开发一个基于 Quorum 算法的队列组件，这个组件的每一组由 3 机存储服务组成。与一般队列的不同之处在于，这个组件对队列写入操作进行了大幅简化，3 机存储服务不需要相互通讯，每个机器上的数据都是顺序写，执行写操作时在 3 机能写入成功 2 份即为写入成功；若失败，则换另外一组再试。因此这个队列可以达到极高的可用性和写入性能。每个数据中心将需要同步的数据写入本数据中心的同步队列后，由其他数据中心的数据重放服务将数据拉走并进行重放，达到数据同步的目的。

2. 网络加速

海外数据中心建设周期长，投入大，微信只在香港和加拿大有两个海外数据中心。但世界那么大，即便是这两个数据中心，也还是没法辐射全球，让各个角落的用户都能享受到畅快的服务体验。

通过在海外实际对比测试发现，微信客户端在发消息等一些主要使用场景与主要竞品有不小的差距。为此，我们跟公司的架构平台部、网络平台部和国际业务部等兄弟部门一起合作，围绕海外数据中心，在世界各地精心选址建设

了数十个 POP 点（包括信令加速点和图片 CDN 网络）。另外，通过对移动网络的深入分析和研究，我们还对微信的通讯协议做了大幅优化。微信最终在对比测试中赶上并超过了主要的竞品。

精耕细作

1. 三园区容灾

2013.7.22 微信发生了有史以来最大规模的故障，消息收发和朋友圈等服务出现长达 5 个小时的故障，故障期间消息量跌了一半。故障的起因是上海数据中心一个园区的主光纤被挖断，近 2 千台服务器不可用，引发整个上海数据中心（当时国内只有这一个数据中心）的服务瘫痪。

故障时，我们曾尝试把接入到故障园区的用户切走，但收效甚微。虽然数百个在线模块都做了容灾和冗余设计，单个服务模块看起来没有单点故障问题；但整体上看，无数个服务实例散布在数据中心各个机房的 8 千多台服务器内，各服务 RPC 调用复杂，呈网状结构，再加上缺乏系统级的规划和容灾验证，最终导致故障无法主动恢复。在此之前，我们知道单个服务出现单机故障不影响系统，但没人知道 2 千台服务器同时不可用时，整个系统会出现什么不可控的状况。

其实在这个故障发生之前 3 个月，我们已经在着手解决这个问题。当时上海数据中心内网交换机异常，导致微信出现一个出乎意料的

故障，在 13 分钟的时间里，微信消息收发几乎完全不可用。在对故障进行分析时，我们发现一个消息系统里一个核心模块三个互备的服务实例都部署在同一机房。该机房的交换机故障导致这个服务整体不可用，进而消息跌零。这个服务模块是最早期（那个时候微信后台规模小，大部分后台服务都部署在一个数据园区里）的核心模块，服务基于 3 机冗余设计，年复一年可靠地运行着，以至于大家都完全忽视了这个问题。

为解决类似问题，三园区容灾应运而生，目标是将上海数据中心的服务均匀部署到 3 个物理上隔离的数据园区，在任意单一园区整体故障时，微信仍能提供无损服务。

1) 同时服务

传统的数据中心级灾备方案是“两地三中心”，即同城有两个互备的数据中心，异地再建设一个灾备中心，这三个数据中心平时很可能只有一个在提供在线服务，故障时再将业务流量切换到其他数据中心。这里的主要问题是灾备数据中心无实际业务流量，在主数据中心故障时未必能正常切换到灾备中心，并且在平时大量的备份资源不提供服务，也会造成大量的资源浪费。

三园区容灾的核心是三个数据园区同时提供服务，因此即便某个园区整体故障，那另外两个园区的业务流量也只会各增加 50%。反过来说，只需让每个园区的服务器资源跑在容量上限的 2/3，保留 1/3 的容量即可提供无损的容灾能力，而传统“两地三中心”则有多得多的服务器资源被闲置。此外，在平时三个园

区同时对外服务，因此我们在故障时，需要解决的问题是“怎样把业务流量切到其他数据园区？”，而不是“能不能把业务流量切到其他数据园区？”，前者显然是更容易解决的一个问题。

2) 数据强一致

三园区容灾的关键是存储模块需要把数据均匀分布在3个数据园区，同一份数据要在不同园区有2个以上的一致的副本，这样才能保证任意单一园区出灾后，可以不中断地提供无损服务。由于后台大部分存储模块都使用KVSVR，这样解决方案也相对简单高效——将KVSVR的每1组机器都均匀部署在3个园区里。

3) 故障时自动切换

三园区容灾的另一个难点是对故障服务的自动屏蔽和自动切换。即要让业务逻辑服务模块能准确识别出某些下游服务实例已经无法访问，然后迅速自动切到其他服务实例，避免被拖死。我们希望每个业务逻辑服务可以在不借助外部辅助信息（如建设中心节点，由中心节点下发各个业务逻辑服务的健康状态）的情况下，能自行决策迅速屏蔽掉有问题的服务实例，自动把业务流量分散切到其他服务实例上。另外，我们还建设了一套手工操作的全局屏蔽系统，可以在大型网络故障时，由人工介入屏蔽掉某个园区所有的机器，迅速将业务流量分散到其他两个数据园区。

4) 容灾效果检验

三园区容灾是否能正常发挥作用还需要进行实际的检验，我们在上海数据中心和海外的香港数据中心完成三园区建设后，进行了数次

实战演习，屏蔽单一园区上千台服务，检验容灾效果是否符合预期。特别地，为了避免随着时间的推移某个核心服务模块因为某次更新就不再支持三园区容灾了，我们还搭建了一套容灾拨测系统，每天对所有服务模块选取某个园区的服务主动屏蔽掉，自动检查服务整体失败量是否发生变化，实现对三园区容灾效果的持续检验。

2. 性能优化

之前我们在业务迅速发展之时，优先支撑业务功能快速迭代，性能问题无暇兼顾，比较粗放的贯彻了“先扛住再优化”的海量之道。2014年开始大幅缩减运营成本，性能优化就被提上了日程。

我们基本上对大部分服务模块的设计和实现都进行了重新review，并进行了有针对性的优化，这还是可以节约出不少机器资源的。但更有效的优化措施是对基础设施的优化，具体的说是对Svrkit框架的优化。Svrkit框架被广泛应用到几乎所有服务模块，如果框架层面能把机器资源使用到极致，那肯定是事半功倍的。

结果还真的可以，我们在基础设施里加入了对协程的支持，重点是这个协程组件可以不破坏原来的业务逻辑代码结构，让我们原有代码中使用同步RPC调用的代码不做任何修改，就可以直接通过协程异步化。Svrkit框架直接集成了这个协程组件，然后美好的事情发生了，原来单实例最多提供上百并发请求处理能力的服务，在重编上线后，转眼间就能提供上千并发请求处理能力。Svrkit框架的底层实现在这一

时期也做了全新的实现，服务的处理能力大幅提高。

3. 防雪崩

我们一直以来都不太担心某个服务实例出现故障，导致这个实例完全无法提供服务的问题，这个在后台服务的容灾体系里可以被处理得很好。最担心的是雪崩：某个服务因为某些原因出现过载，导致请求处理时间被大大拉长。于是服务吞吐量下降，大量请求积压在服务的请求队列太长时间了，导致访问这个服务的上游服务出现超时。更倒霉的是上游服务还经常会重试，然后这个过载的服务仅有的一点处理能力都在做无用功（即处理完毕返回结果时，调用端都已超时放弃），终于这个过载的服务彻底雪崩了。最糟糕的情况是上游服务每个请求都耗时那么久，雪崩顺着 RPC 调用链一级级往上传播，最终单个服务模块的过载会引发大批服务模块的雪崩。

我们在一番勒紧裤腰带节省机器资源、消灭低负载机器后，所有机器的负载都上来了，服务过载变得经常发生了。解决这一问题的有力武器是 Svrkit 框架里的具有 QoS 保障的 FastReject 机制，可以快速拒绝掉超过服务自身处理能力的请求，即使在过载时，也能稳定地提供有效输出。

4. 安全加固

近年，互联网安全事件时有发生，各种拖库层出不穷。为保护用户的隐私数据，我们建设了一套数据保护系统——全程票据系统。其

核心方案是，用户登录后，后台会下发一个票据给客户端，客户端每次请求带上票据，请求在后台服务的整个处理链条中，所有对核心数据服务的访问，都会被校验票据是否合法，非法请求会被拒绝，从而保障用户隐私数据只能由用户通过自己的客户端发起操作来访问。

新的挑战

1. 资源调度系统

微信后台有成千的服务模块，部署在全球数以万计的服务器上，一直依靠人工管理。此外，微信后台主要是提供实时在线服务，每天的服务器资源占用在业务高峰和低谷时相差很大，在业务低谷时计算资源被白白浪费；另一方面，很多离线的大数据计算却受制于计算资源不足，难以高效完成。

我们正在实验和部署的资源调度系统（Yard）可以把机器资源的分配和服务的部署自动化、把离线任务的调度自动化，实现了资源的优化配置，在业务对服务资源的需求有变化时，能更及时、更弹性地自动实现服务的重新配置与部署。

2. 高可用存储

基于 Quorum 算法的 KVSvr 已经实现了强一致性、高可用且高性能的 Key-Value/Key-Table 存储。最近，微信后台又诞生了基于 Paxos 算法的另一套存储系统，首先落地的是 PhxSQL，一个支持完整 MySQL 功能，又同时

具备强一致性、高可用和高性能的 SQL 存储。

“ArchSummit 会员微信群”是一个基于专家运营的高端技术社区。群内定期举行线上线下专家讲座、话题讨论，专注于“分享、交流、成长”。微信添加群主 cuikang10，申请入群。

手机淘宝构架演化实践



作者 藏秀涛

“移动互联网，随时随地”是非常火爆的一个专题。阿里无线事业部技术负责人庄卓然（花名南天）任出品人。来自阿里无线事业部的高级专家李敏在 ArchSummit 全球架构师峰会上分享了《手机淘宝架构演化实践》([视频连接](#))。

李敏主要负责淘宝无线客户端和无线网站基础服务、购物主链路的架构、研发方面的工作。从 09 年开始参与手机淘宝研发团队的组建和线上产品研发，先后负责过无线部门的社区、会员、营销、交易等多条产品线的技术工作，构建和发展了阿里无线技术体系中包括交易链路、百亿级别高性能 API 网关、WebApp 平台等多个重要技术产品，经历和见证了阿里巴巴无线从开始之初到成为日活上亿级别电商应用技术变迁和积累。

本文即根据李敏的演讲整理而成。

发展阶段

从 2009 年开始，DAU 从 100 万增长到超过 1 亿，面临的问题、包括研发支撑所需要解决的事情各不相同。在用户量和业务复杂度的线性递增下，架构也进行了相应的演进。如下图所示，具体可以分为四个阶段：

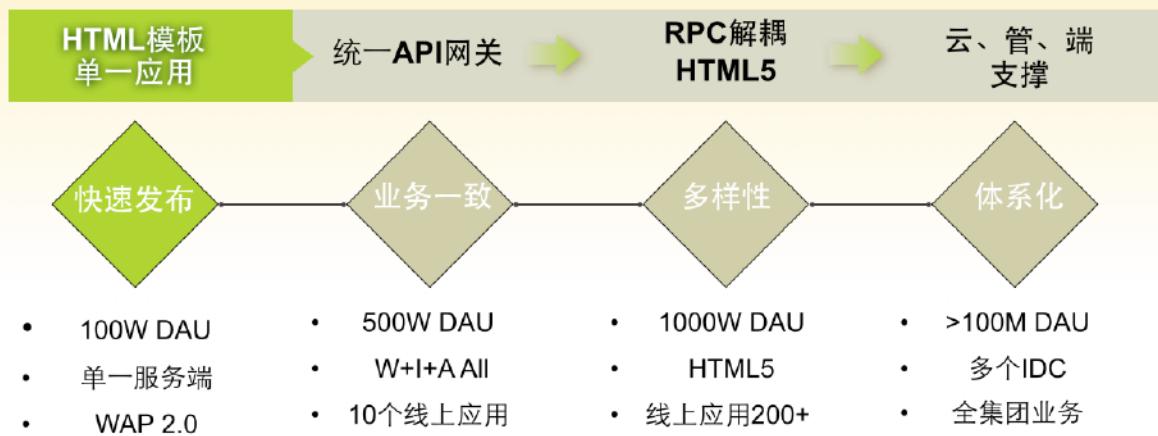
- 第一阶段，手淘的前身 WAP 网站，业务初立、变化快，需要快速发布，采取

HTML 模板和单一应用，最大程度满足快速发布和修改的需要；甚至不需要改动后端的业务代码，在前面的模板上做一些修改就可以了。

- 第二阶段，DAU 的快速增长，WAP/Android/iOS 多个平台的业务起来了，需要在多个平台上进行快速的业务复制和业务管控，统一 API 网关出现。

走过的路

手淘在用户量和业务复杂度的线性递增下
架构也进行了相应的演进



- 第三阶段，DAU 进一步增长，线上系统越来越多，业务的多样性需求更多的体现出来，基于 HTML5 的一整套解决方案上线，更多的 HTML5 和 Native 混合的业务形态，API 网关进行进一步优化和扩展，更方便的接入方式。
- 第四阶段，当 DAU 达到 100M 的时候，全集团的业务都需要在手淘透出，API 网关被部署到更多的 IDC 机房，如何更有体系化的进行有效研发、接入更多业务、并进行更有效的业务监控，需要更加体系化的架构治理。

API 网关

做 WAP 的时候没有所谓的 API 网关，为什么要用 API 网关呢？

随着应用数量的增多，每个应用分别暴露

的 API 出口很多，修改的话逻辑很复杂，这时候应该引入一个统一的网关。

但随着 DAU 的增长，API 网关会成为一个单点。开发团队在中间做了很多技术和架构上的努力，主要有几个关键点。一是后端接入很多应用，其实 API 网关只是通路，理论上不存在调用的上限，只要内存够大，包括网卡的流量够的话都可以上来。二是有必要的机制做到宽阔的调用网关。还有一点，当后端业务要经过 API 网关时，其实现在业界很多都是典型的 RPC 的模式，RPC 的模式有一个绕不开的问题，就是可能要设定一些东西，这时后端服务跟 API 会有一定程度上的耦合。现阶段要接入服务，后端服务器随时都会变化，不可能后端服务变化的时候都对 API 做相应的发布，这是不现实的。所以有一套自己的 RPC 机制，解除了这种强类型的约束。

此外，可以在网关上附加很多功能，比如

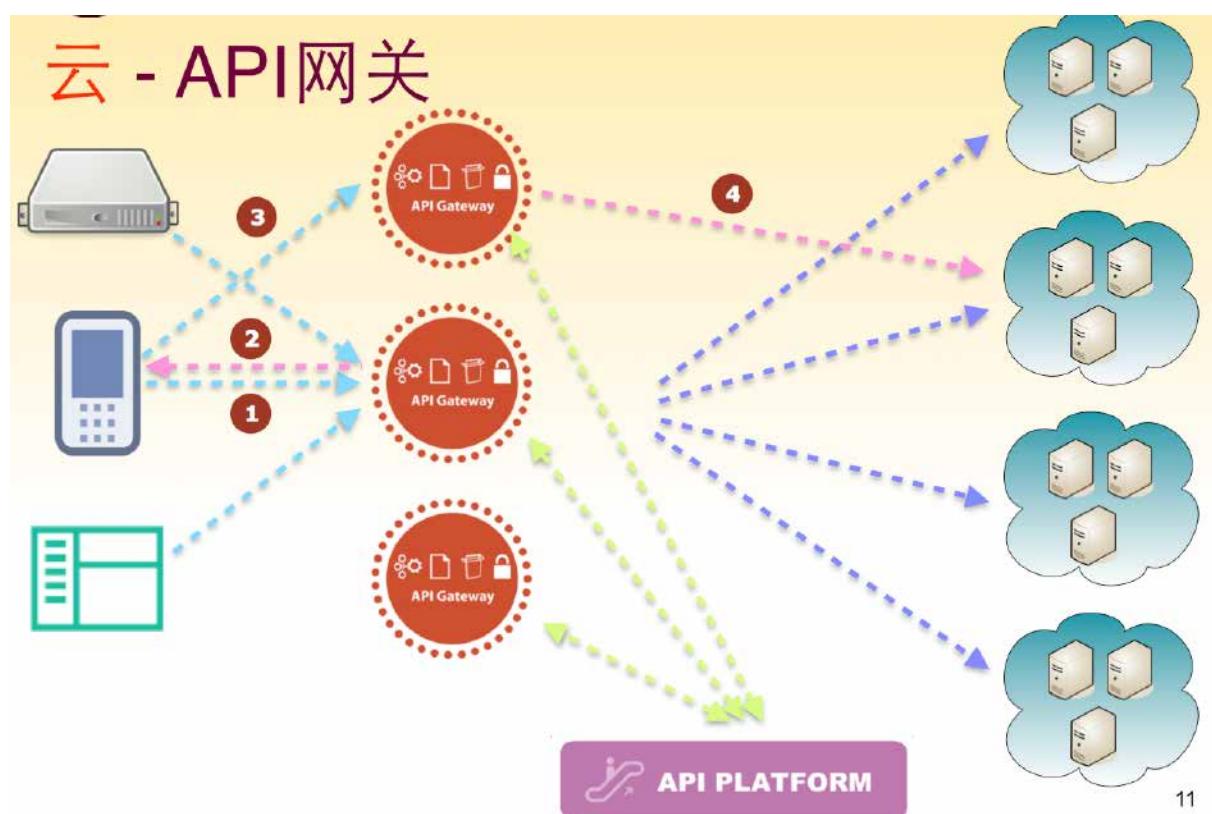
安全、审计，还有一些日志、审查等。

到了现在这个阶段，要进行异地部署，很多 IDC，这样的话引入 API 网关很可能会带来问题。包括今年的双 11 或者是双 12，要在多个异地机房支撑手机淘宝的业务，会有很多 API 网关。

手机端

1.Bundle

去年下半年，开发团队对整个手机淘宝的架构做了比较大的调整，如下图所示，左侧是



比如说像 APP 可以在中心网关上面询问，应该去哪个真正的 API 网关。然后中心 API 网关会告诉它结果，它再连接到所在地的 API 网关上，然后再向后端 API 发起调用，所有 API 的服务网关都受管控中心统一管控。比如说增加一些新的功能，上线一些新的 API，包括一些引流、切换，这些指令都会在管理平台上向各个 API 网关发送。

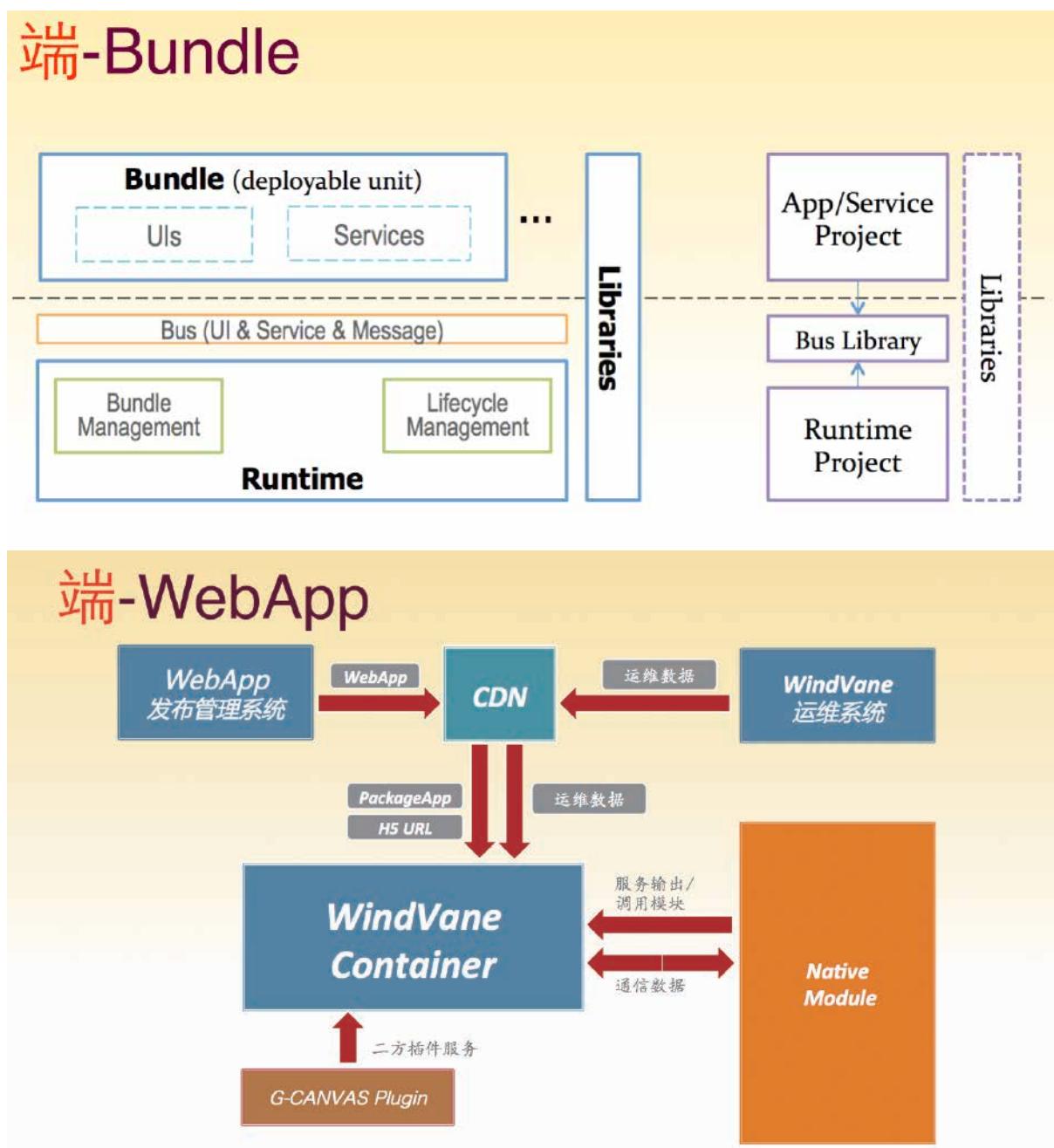
运行时的架构分布，右侧是工程代码级别的分布。在运行的时候，其他的业务团队提供的都是一个个的业务 Bundle，这是可部署的单元，包括 UI、服务和标准中间件的调用代码，下面有一个总线程，负责管理和开发好统一的 UI 服务，包括消息服务的总线。再下面是运行容器，上面跑的是所有的 Bundle 的东西，对应运行时的东西，右侧是真正在开发时候的结构。比如说聚划算，它要开发它的业务，就做一个单独的工程，然后去开发；它只用开发自己的，

开发到差不多的时候，就将其代码打成一个 Bundle 提供过来，然后一起打包发出去。

2.WebApp

下图是现在手机淘宝上关于 HTML5 的整体框架图。手机淘宝上的方案大致分为两部分，中间那一部分是手机淘宝自己开发的 HTML5 的运行容器，它负责在上面跑各种各样

的 WebApp，在线上有一个统一发布管理系统，它可能对性能进行检测，包括 CDN 是否符合规格，HTML 本身有没有异常等情况，经过这些必要的检测，包括审查之后，它统一发到 CDN 上。容器本身其实也会接受运行时的信息，容器接收到这两边的指令之后，它自己会做一些更新配置，也可能会装载运行，从线上系统下发新的 WebApp。此外，还可以运行 WebApp



或者是做 URL 的导航拦截，甚至做一些交互。

3.PackageApp

这是今年新的建设，整个系统是基于前面整个体系来做的，称之为 PackageApp。

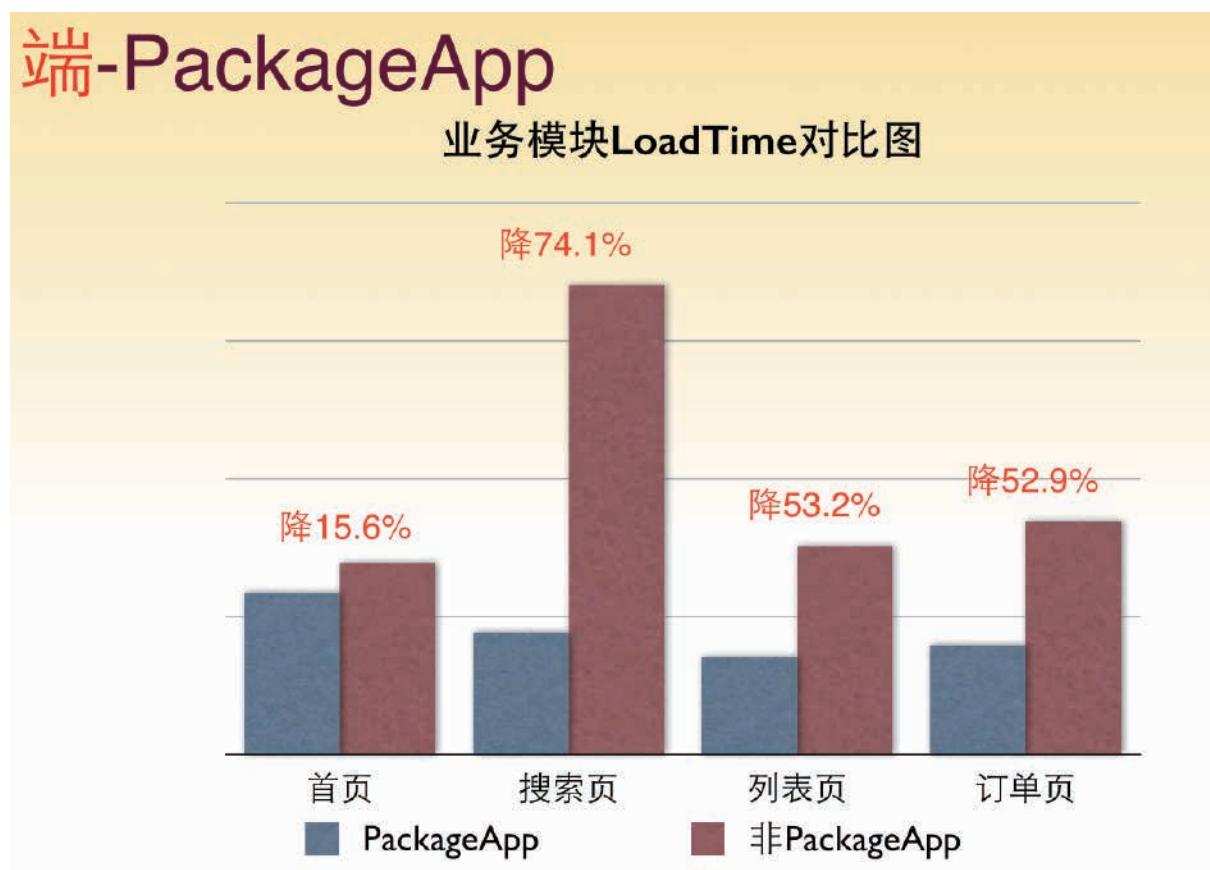
这个跟前面最大的区别就是让用户感知不到前面同步下载的过程，大概的做法是：把 HTML5 以及 WebApp 在发版之前先做一些预知放到客户端里面，前面会做两件事情，首先按照原来的逻辑运行，其次就是在右侧的蓝图里面，它会去做一些 UI 的拦截，发现用户点击的 icon 进去之后，整个 URL 是符合用户规范的，它会启动检测机制去检查线上是不是有新的版本需要下载，如果说有的话会启动异步更新模块，从 CDN 上拉取新的 WebApp 版本，否则

会走到原来的地方，最后既不影响用户去使用

WebApp，又能把自己最新的版本更新到所期望的版本，这由统一的管理平台去发布。

在方案设计之初，还思考了三个方面，首先它是标准的 URL，在点击进去之后是导航的 URL，对于前端工程师来说，他设计研发 WebApp 跟客户端或者是线上跟 HTML5 的网站是一致的。其次，手机淘宝自己的容器，制定了自己的规范，在底层的容器上面可以实现手淘定义的规范。第三，“不同网络、全量、差量更新”，这点很重要，在移动互联网场景下，到底要发起几条链接拉取资源，在 WIFI 下怎么拉取资源，其实都是不一样的。在不同网络下面，对策都不太一样。

下面是采用 PackageApp 后业务模块 LoadTime 对比图：



支撑体系

除了前面介绍的内容，比较大的电商 App，还需要一个很完备的支撑体系。如果没有的话，在线上运行的情况是不可感知的。手淘在不同的维度也做了很多支撑的工具。

1. 研发支撑

在研发支撑上面，像传统的 Reivew 代码，特别是做客户端的同学几乎都会做统一的 UI 库，大家会设计模板，比较典型的，会有所谓的日常预发、线上染色等等。它的集群数量跟能够进来的用户是很有限的，通过这个环境来确认所开发的代码发布到线上可能会有什么问题。一套代码经过预发之后再发布到线上去，最后有一个染色环境，比如说用户打电话反馈遇到的问题，比如说下单下不了或者是搜索无结果，这时会有一个染色集群，把用户定位到染色集群上面，对用户专门进行一些分析，现在还没有做到直接在用户机器上做调试，但是用户到了染色集群上面，整个调用的链路会剥离出来，比较好分析用户到底发生了什么事情。

2. 测试支撑

App 的测试很重要，除了比较常规的单元功能测试，还有很重要的像稳定性跟性能，以及自动化这些都是很重要的。像手机淘宝差不多是一个月左右的时间可能会迭代一个版本。比如说新的功能开发会不会影响到老的功能，智能化测试很重要，可能分成两部分，一部分是线上所有的 API，包括业务逻辑是不是正常。

另一部分是新写的代码会不会有问题，因为前面架设了统一的 API 网关，所以会在网关这个层面做很多自动化的调用回归，构造很多正常用户的数据去测试线上 API 系统的返回值，包括一些异常是不是正常，来保证线上业务逻辑的正常。在客户端这一侧，则会做很多自动脚本的回归，保障整个客户端新做的代码跟原来相比没有什么问题。另外还引入了比较多的静态代码扫描，保证不会出现低级问题。

3. 运维支撑

移动 App 的运维支撑跟线上不太一样。除了常见的性能跟稳定性分析，还有针对 App 的业务监控跟舆情监控。舆情监控这个应该是移动 App 所特有的环节，大家通过市场去分发，很多用户会发评论，iOS 特别明显，有人说好，有人说不好，安卓更复杂，特别是国内有大大小小非常多的应用市场，不一而足。所以怎么对舆情做一个有效的监控，既能通过舆情监控，快速收集问题，也能做一些梳理分析，找到产品或者是性能方面的提升点。

4. 发布支撑

发布支撑，其实也是在大的 App 上面才会出现的，针对一个人群的发布。传统的直接是发到市场上，大家都收到了。而手淘现在有很多内部灰度和外部灰度正式发布，可能有一些内测版本只发给阿里巴巴集团内部员工，这可以通过自己做的发布系统来支撑，有比较灵活的发布策略调整：可以圈定一批用户，也可以选定一个区域，甚至可以用后台数据做合理的

设置给特定的版本推送特定升级的版本。

如果 App 发到用户手上，结果发生了致命的问题，怎么办呢？其实有两种方法修复线上的问题，第一个是直接替换 Bundle，另外就是更小维度的补丁——热补丁，现在在安卓上做的比较多。

李敏还分享了一个案例，在上半年有一次大促的时候发生了一个问题，零点就要促销了，版本可能是前天刚发布给用户的，那怎么办呢？替换 Bundle 也可以做到，但是数据下载量非常大，而且刚发布不久，这样对用户影响比较大，所以选择了用热补丁修复，主要是类似于 ClassLoader 替换，用 JAVA 开发的应该知道，主要是用类替换的方式做的。在 iOS 上也有一些方案，不过还在尝试当中。

客户端监控

可以在分钟级别确定用户调用某个操作的成功次数、失败次数和失败率，实现对业务可用性的监控。

“ArchSummit 会员微信群”是一个基于专家运营的高端技术社区。群内定期举行线上线下专家讲座、话题讨论，专注于“分享、交流、成长”。微信添加群主 cuikang10，申请入群。

讲师介绍：李敏（花名心石，微博：@allblue_ 华丽地低调），阿里巴巴集团无线事业部高级无线技术专家，2009 年 4 月加入阿里巴巴。负责淘宝无线客户端和无线网站基础服务、购物主链路的架构、研发方面的工作。从 09 年开始参与手机淘宝研发团队的组建和线上产品研发，先后负责过无线部门的社区、会员、营销、交易等多条产品线的技术工作，构建和发展了阿里无线技术体系中包括交易链路、百亿级别高性能 API 网关、webapp 平台等多个重要技术产品，经历和见证了阿里巴巴无线从开始之初到成为日活上亿级别电商应用技术变迁和积累。从 1994 年接触计算机以来，对计算机各个领域的内容都始终保持了基于乐趣的关注，同时也是一个狂热的科幻电影爱好者。

舆情平台

舆情平台是移动 App 所独有的。要获取信息，会从用户手机淘宝自己填的反馈，利用市场和微博，实时抓取，然后把内容聚合起来，进行热门词归类，筛选出一些热门的标签话题做一些智能分类，分类之后大致知道到底是支付、详情、退款出现了什么问题，确定问题的重点之后，可以直接联系用户，甚至去跟踪用户，根据这些问题去修复线上的紧急问题或者是改善产品，这就是在线上实际使用的舆情平台。通过热门词的分类排名，就可以知道某一个版本在某一个阶段最重要的问题是什么，还扩展了用户集中反馈。

比如举办一个抢红包的活动，这个活动出现了什么问题，大量的用户重复反馈这个问题，就可以把热门的话题聚集起来。另外还可以通过舆情平台确定某个技术的改造是否成功。

舆情平台早期主要用于收集一些信息，后来发现把舆情收集起来做一些大数据分析，可以得出很多自动化的结论，甚至可以验证研发的结果是好是坏。

春晚微信红包，是怎么扛住一百亿次请求的



作者 崔 康

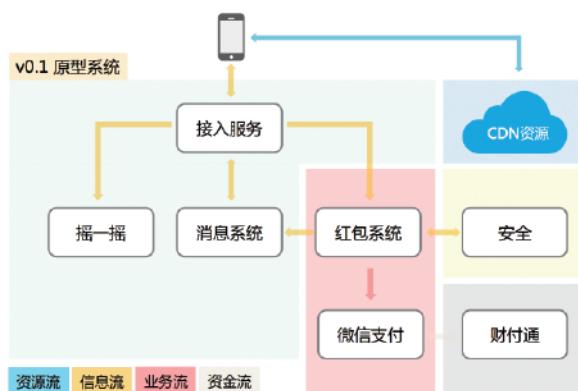
本文根据微信资深工程师张文瑞在 ArchSummit 深圳 2015 大会的演讲整理而成，略有修改（[视频连接](#)）

本文跟大家分享的主题是如何实现“有把握”的春晚摇一摇系统。回忆一下春晚的活动，有什么样的活动形式呢？

当时我们是直接复用客户端摇一摇入口，专门给春晚摇一摇定制了一个页面，可以摇出“现金拜年”、“红包”。底下的红包肯定是大家比较感兴趣的，也是今天下午重点介绍的内容。比较精彩的活动背后一定会有一个设计比较独到的系统。



V0.1 原型系统



我们看一下这个系统，我们当时做了一个原型系统，比较简单，它已经实现了所有的功能，摇那个手机的时候会通过客户端发出一个请求，接入服务器，然后摇一摇服务，进行等级判断，判断以后把结果给到后端，可能摇到拜年或红包，假设摇到红包，上面有LOGO和背景图，客户端把这个LOGO和背景图拉回去，用户及时拆开红包，拆的请求会来到红包系统，

红包系统进行处理之后会到支付系统，到财富通的转帐系统，最终用户拿到红包。拿到钱以后，只是其中一份，还有好几份是可以分享出去，我们称之为“分裂红包”，通过信息系统转发给好友或群里，好友跟群里的人可以再抢一轮。

整个过程归一下类，叫资源流、信息流、业务流、资金流，今天讲的主要是资源流跟信息流。

原始系统看起来比较简单，是不是修改一下直接拿到春晚上用就可以了？肯定不行的。到底它有什么样的问题呢，为什么我们不能用，在回答这个问题之前想请大家看一下我们面临的挑战。

1、我们面临怎样的挑战？

第一个挑战是比较容易想到的，用户请求量很大，当时预计7亿观众，微信用户也挺多

的，当时预估一下当时峰值达到一千万每秒，通过图对比一下，左边是春运抢火车票，一秒钟请求的峰值是 12 万，第二个是微信系统，微信系统发消息有个小高峰，那时候峰值每秒钟是 33 万，比较高一点的是预估值一千万每秒，右边是春晚时达到的请求峰值是 1400 万每秒。

这个活动跟春晚是紧密互动的，有很多不确定因素，体现在几个方面。一个是在开发过程中，我们的活动怎么配合春晚，一直没有定下来，很可能持续到春晚开始前，显然我们的客户端跟我们的系统到那时候才发布出去，这时候我们的开发就会碰到比较多的问题了，这是第一个。

第二个，在春晚过程中，因为春晚是直播型节目，节目有可能会变，时长会变，顺序会变，活动过程跟春晚节目紧密衔接在一起，自己也是会有挑战的，这也是不确定的因素。再就是我们系统是定制的，专门为春晚定制，只能运行这么一次，这是挺大的挑战，运行一次的系统不能通过很长的时间，检查它其中有什么问题很难，发出去了以后那一次要么就成功了，要么就失败了。

第三个，因为春晚观众很多，全国人民都在看，高度关注，我们必须保证成功，万一搞砸了就搞砸在全国人民面前了。这么大型的活动在业界少见，缺少经验，没有参考的东西。还有就是我们需要做怎样的准备才能保证万无一失或者万有一失，保证绝大部分用的体验是 OK 的，有很多问题需要我们不断地摸索思考。原型系统不能再用的，再用可能就挂了。

2、原型系统存在哪些问题？

原型系统有哪些问题呢？第一个是在流量带宽上，大量的用户请求会产生大量的带宽，预估带宽峰值是 3000pb 每秒，假设我们资源是无限的能够满足带宽需求，也会碰到一个问题，用户摇到以后有一个等待下载的过程。第二个问题，在接入质量这一块，我们预估同时在线 3.5 亿左右，特别是在外网一旦产生波动的时候怎么保证用户体验不受损而系统正常运作。第三个挑战，请求量很大，1000 万每秒，如何转到摇一摇服务，摇一摇服务也面临一千万请求量，我们系统要同时面对两个一千万请求量，这不是靠机器的，大家都有分布式的经验，这么大规模的时候任何一点波动都会带来问题，这是一个很大的挑战。

3、我们是如何解决这些问题的？

针对以上几点，我们详细看一下每一点我们是怎么做到的。我们首先看一下信心指数，把这个系统拿到春晚去跑有多少信心，这里的指数是 10，如果这个系统拿到春晚去用，而且还成功了，这个概率是 10%。当然我们的系统不能建立在运气的基础上，应该怎么做？第一个，在带宽这一块客户端可以摇到多种多样的结果，结果大部分都是静态资源，静态资源我们可以提前制作出来下发到客户端，在后台做了资源推送的服务，客户端拿到列表以后可以先行下载，客户端利用闲时把资源拉过去。碰到几个问题，资源交付情况的问题，需要增量的发下去；二是资源更新；三是资源下载失败，

失败的话怎么办呢；四是资源覆盖率，依靠这个系统下载资源的用户，比如覆盖率只有 20%、30%，两个东西就没有意义了，覆盖率要达到 90% 左右；五是离线资源下载，万一有些人把里面的东西修改了，可能会产生意想不到的结果，怎么保证离线资源的安全。

这里有个数据，2月9号到2月18号下发资源 65 个，累积流量 3.7PB，峰值流量 1Tb/s。通过这种方式解决了下载资源的问题。



再就是外网接入质量，在上海跟深圳两地建立了十八个接入集群，每个城市有三网的介入，总共部署了 638 台接入服务器，可以支持同时 14.6 亿的在线。

所有用户的请求都会进入到接入服务器，我们建立了 18 个接入集群，保证如果一个出现问题的时候用户可以通过其它的接入，但是在我们内部怎么把请求转给摇一摇服务，摇一摇处理完还要转到后端，怎么解决呢？解决这个问题代价非常大，需要很多资源，最终我们选择把摇一摇服务去掉，把一千万每秒的请求干掉了，把这个服务挪入到接入服务。除了处理摇一摇请求之外，所有微信收消息和发消息都需要中转，因为这个接入服务本身，摇一摇的

逻辑，因为时间比较短，如果发消息也受影响就得不偿失了。

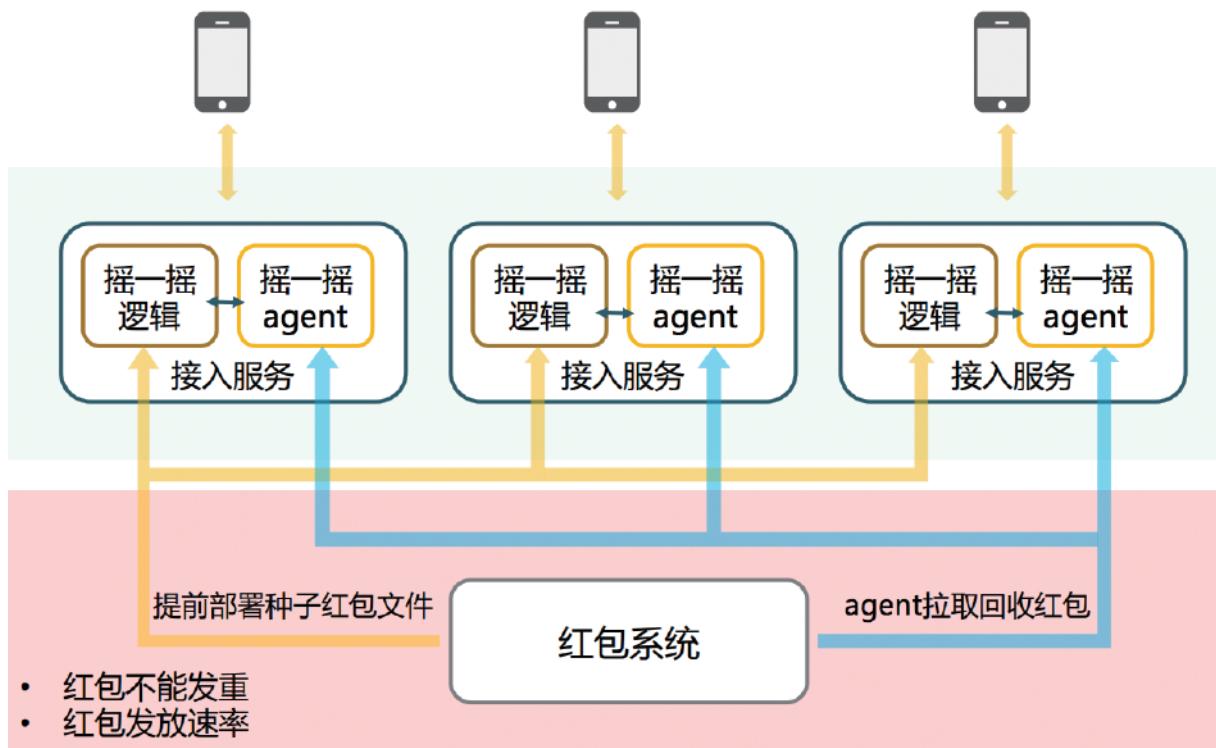
恰好有一个好处，我们的接入服务的架构是有利于我们解决这个问题的，在这个接入节点里分为几个部分，一个是负责网络 IO 的，提供长链接，用户可以通过长链接发消息，回头可以把请求中转到另外一个模块，就是接入到逻辑模块，平时提供转发这样的功能，现在可以把接入逻辑插入。这样做还不够，比方说现在做一点修改，还需要上线更新，摇一摇的活动形式没有怎么确定下来，中间还需要修改，但是上线这个模块也不大对，我们就把接入的逻辑这一块再做一次拆分，把逻辑比较固定、比较轻量可以在本地完成的东西，不需要做网络交互的东西放到了接入服务里。

另外一个涉及到网络交互的，需要经常变更的，处理起来也比较复杂的，做了个 Agent，通过这种方式基本上实现了让接入能够内置摇一摇的逻辑，而且接入服务本身的逻辑性不会受到太大的损伤。解决这个问题之后就解决了接入的稳定性问题，后面的问题是摇一摇怎么玩，摇一摇怎么玩是红包怎么玩，在红包过程中怎么保证红包是安全的呢，红包涉及到钱，钱是不能开玩笑的。第三个问题是怎样跟春晚保持互动，春晚现场直播，我们怎么跟现场直播挂钩衔接起来。

先看红包如何发放。

前面说道摇一摇请求，其实是在接入服务做的，红包也是在接入服务里发出去的，为了在发红包过程中不依赖这个系统，我们把红包的种子文件在红包系统里生成出来，切分，分

红包如何发放？



到每个接入服务器里，每个接入服务器里都部署了专门的红包文件。一个红包不能发两次，红包的发放速率需要考虑，发放红包一定有用户拆，拆了还要再抢，我们需要精确控制，确保所有请求量都是在红包系统能够接受的范围内。在这个过程中还会有另外一个风险，用户摇到红包之后还可以有一些分裂红包分出去，他也可以不分享，不分享的也不会浪费，可以收回过来，会通过本地拉回去。这部分因为是比较少量的，问题不大，因为所有红包已经发出去了，只是补充的。这里我们就搞定了红包发放。

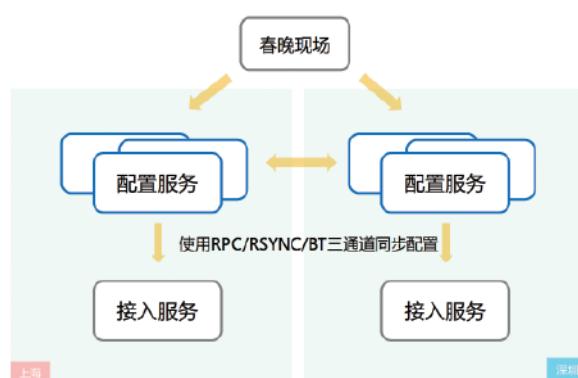
二是怎么样保证红包不被多领或恶意领取，每个客户领三个红包，这是要做限制的，但这是有代价的，就是存储的代价。

我们在我们的协议里后台服务接入的摇一摇文件里下发红包的时候写一个用户领取的情况，客户端发再次摇一摇请求的时候带上来，我们检查就行了，这是一个小技巧，这种方式解决用户最多只能领三个、企业只能领一个限制的问题。这个只能解决正版客户端的问题，恶意用户可能不用正版，绕过你的限制，这是有可能的。怎么办呢？一个办法是在 Agent 里面，通过检查本机的数据能够达到一个目的，摇一摇接入服务例有 638 台，如果连到不同的机器，我们是长连，还可以短连，还可以连到另一台服务器，可以连到不同的地方去。还有一个问题是人海战术，有些人拿着几万、几十万的号抢，抢到都是你的，那怎么办呢？这个没有太好的办法，用大数据分析看用户的行

为，你平时养号的吗，正常养号吗，都会登记出来。

怎样跟春晚现场保持互动？需要解决的问题有两个，一个是要迅速，不能拖太长时间，比如现在是刘德华唱歌，如果给出的明星摇一摇还是上一个节目不太合适，要求我们配置变更需要迅速，二是可靠。我们怎么做的呢？

如何与春晚同步进行互动？



春晚现场我们是专门有同学过去的，在他们电脑装了系统，可以跟我们后台进行交互的，节目变了节切一下，变动的请求会发到后台，我们部署两套，一套在深圳、一套在上海，在这个配置里还准备了三步服务，哪一步都可以，同时还可以同步这个数据，这个数据还可以下放到所有的接入机器，会把它同步过去，不是用一种方式，而是用三种方式，通过这种方式可以迅速的在一千台服务器成功，是不是能够达到配置一定能够用？不一定，春晚现场是不可控的，万一指令没有发出怎么办？如果六个配置服务都挂了怎么办，从上一个节目切到下一个节目的时候发生这种问题不是太大，但是主持人在十点三十的时候如果摇红包一摇啥都没有，这就怒了，口播不出来也就挂了。

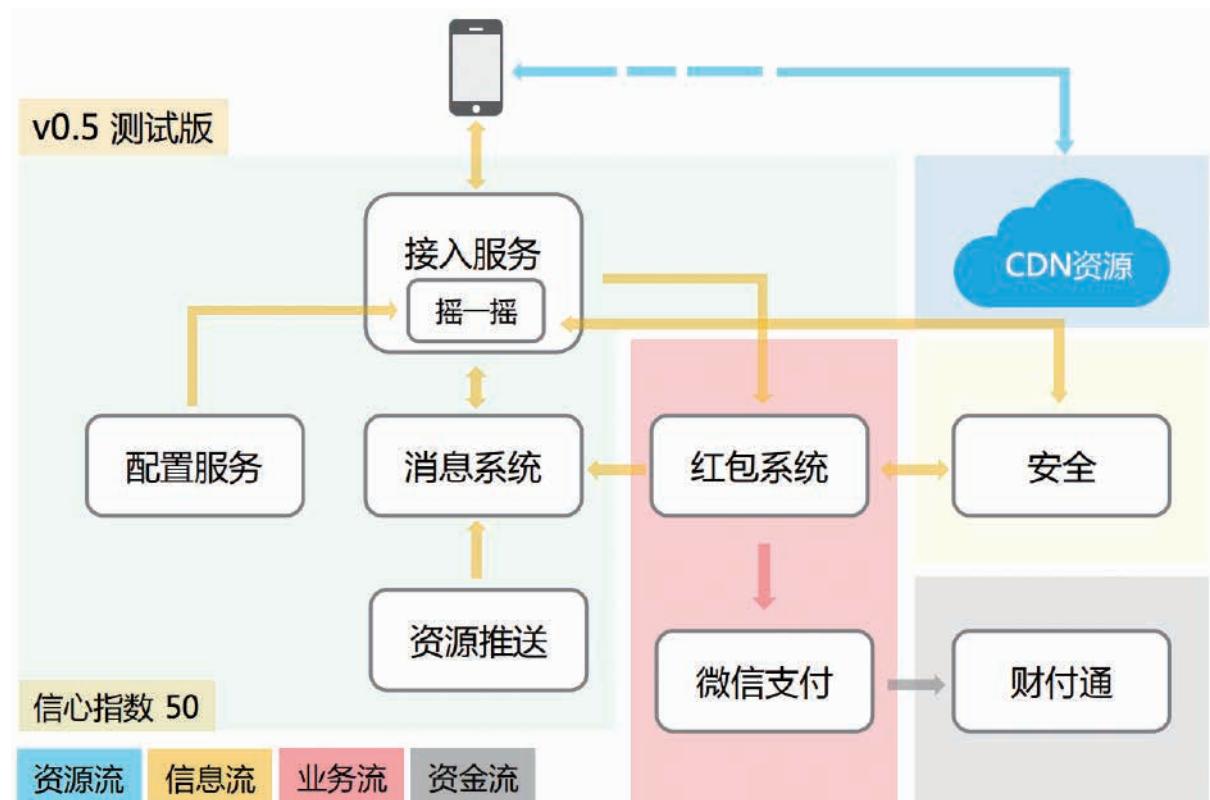
怎么做的呢？主持人肯定有口播，口播的

时间点我们大致知道，虽然不知道精确的时间点，比如彩排的时候告诉我们一个时间，后来变了，我们大致知道时间范围，可以做倒计时的配置，比如十点半不管你有没有口播我们都要发红包了。如果节目延时太长了，你的红包十分钟发完了，之后摇不到，那也不行，这种情况下我们做了校正，在节目过程中我们不断校正倒计时的时间，设了一个策略，定了一个流程，半小时的时候要通知一下我这个时间，因为它是预估的，节目越到后面能定下来的时间范围越精确，提前告诉我们，我们就可以调整。那时候现场是在春晚的小会议室，在小会议室看不到现场什么情况，也是通过电视看，结果电视没信号了，就蒙了，校准就不知道现在怎么回事，进行到哪一步了，当时很着急，还好后来没事，后续的几个节目还是校正回来了，最终我们是精确的在那个时间点出现了抢红包。前面讲了怎么在系统解决流量的问题、请求量的问题，最重要的一点是我们预估是一千万每秒，但如果春晚现场出来的是两千万、三千万或四千万怎么办，是不是整个系统就挂掉了。

我们就是采用过载保护，过载保护中心点是两点，前端保护后端，后端拒绝前端。一个是在客户端埋入一个逻辑，每次摇变成一个请求，摇每十秒钟或五秒钟发送一个请求，这样可以大幅度降低服务器的压力，这只会发生到几个点，一个是服务访问不了、服务访问超时和服务限速。实时计算接入负载，看CPU的负载，在衔接点给这台服务器的用户返回一个东西，就是你要限速了，你使用哪一档的限速，

通过这种方式，当时有四千万用户在摇我们也能扛得住。

V0.5 测试版



这是我们的 0.5 测试版，对这个我们的信心指数是 50，为什么只有 50% 的把握？

我们前面解决的问题都是解决用户能摇到红包，服务器还不会坏掉，但是对摇红包来说那是第一步，后面还有好几步，还要把红包拆出来，还要分享，分享完以后其它人可以抢，这个体验是要保证的，简单分析一下可以发现前面是本人操作，后面是好友操作，这里就存在一个契机，你可以做一些服务，一旦出现问题是可以利用的点，可以做延时。剩下的问题是保证本人操作比较好，后面出点问题可以延

迟，有延迟表示有时间差处理那个东西。

1、核心体验是什么？

这里面我们需要确保成功，确保体验是完全 OK 的，确保成功的时候前面提到原型的系

统里解决了摇到红包的问题，剩下的就是拆红包和分享红包。怎么样确保拆红包和分享红包的用户体验？

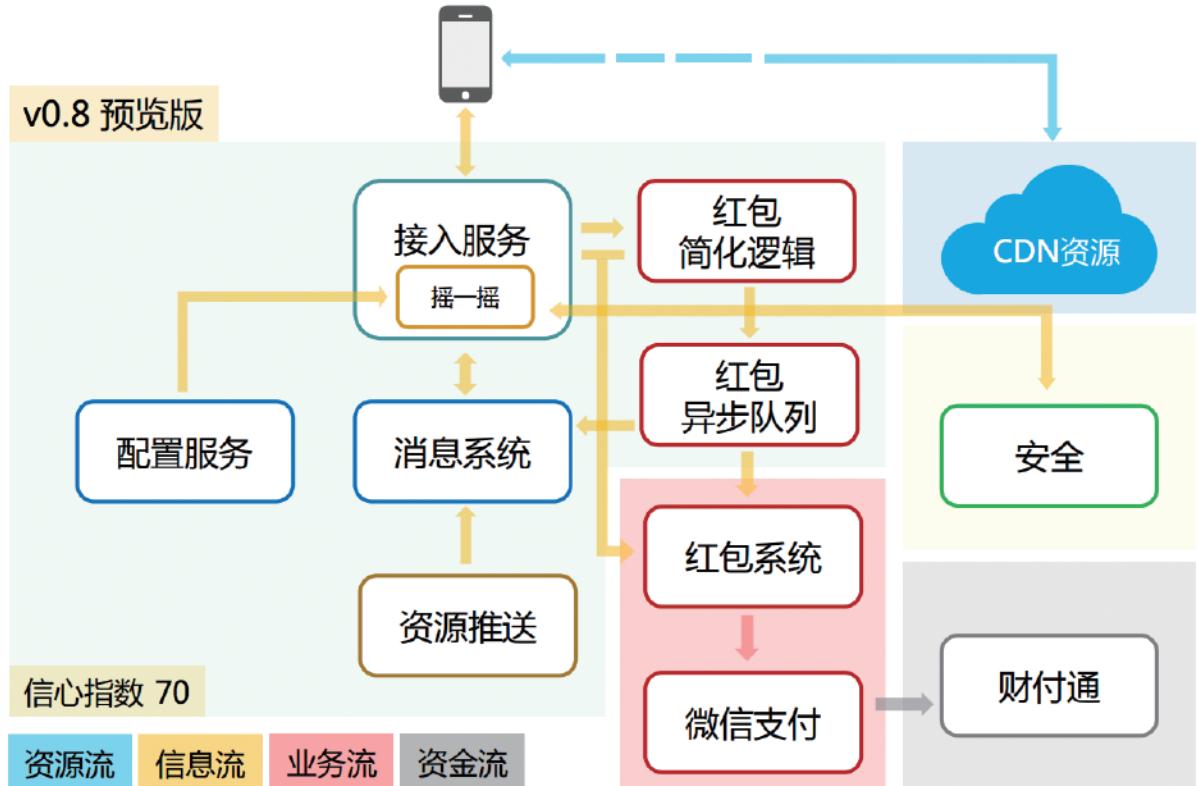
2、如何确保拆/分享红包的用户体验？

拆红包和分享红包可以做一下切割，可以切割成两个部分，一个是用户的操作，点了分享红包按钮，之后是转帐，对我们来说确保前面一点就可以了，核心体验设计的东西再次缩小范围，确保用户操作这一步能够成功。怎么确保呢？我们称之为“铁三角”的东西，拆/分

享红包 = 用户操作 + 后台彰武逻辑。这是我们能做到的最高程度了。

V0.8 预览版

我们又做了 0.8 的版本，预览版，信心指



3、还能做得更极致吗？

但我们还可以做的更好一点，前面这个用户看起来还是成功的，只是入帐入的稍微迟一点点，用户感觉不到。如果我们异步队列这里挂了，或者网络不可用了，概率比较低，我们有三个数据中心，挂掉一个问题不大，万一真的不能用呢，我们又做了一次异步，分两部分：一个是业务逻辑，校验这个红包是不是这个用户的，还有一个透传队列，把这个数据再丢到后边，其实可以相信本机的处理一般是可以成功的，只要做好性能测试基本上是靠谱的。在后面出现问题的时候我们用户的体验基本不受损，保证绝大多数用户的体验是 OK 的。

数 70，我们认为这个东西有七成把握是可以成功的。

大家知道设计并不等于实现，设计的再好实践有问题也很崩溃，要保证设计一是全程压测，二是专题 CODE REVIEW，三是内部演练，四是线上预热，五是复盘与调整。

复盘包括两部分，有问题的时候可以把异常问题看出来，二是很正常，跑的时候是不是跟想象的一样，需要对正常情况下的数据做预估的重新评估，看看是不是符合预期。两次预热，一次是摇了 3.1 亿次，峰值 5000 万一分钟，100 万每秒，跟我们估算的一千万每秒差很远，当时只是针对 iPhone 用户，放开一个小红点，

你看到的时候可以抢，发放红包 5 万每秒，春晚当晚也是五万每秒。后面又发了一次，针对前面几个问题再做一次。

V1.0 正式版

做完这两次连接后面就迎接 2 月 18 号春晚的真正考验。这是 1.0 正式版，信心指数达到 80，我们认为 80% 是可以搞定的。

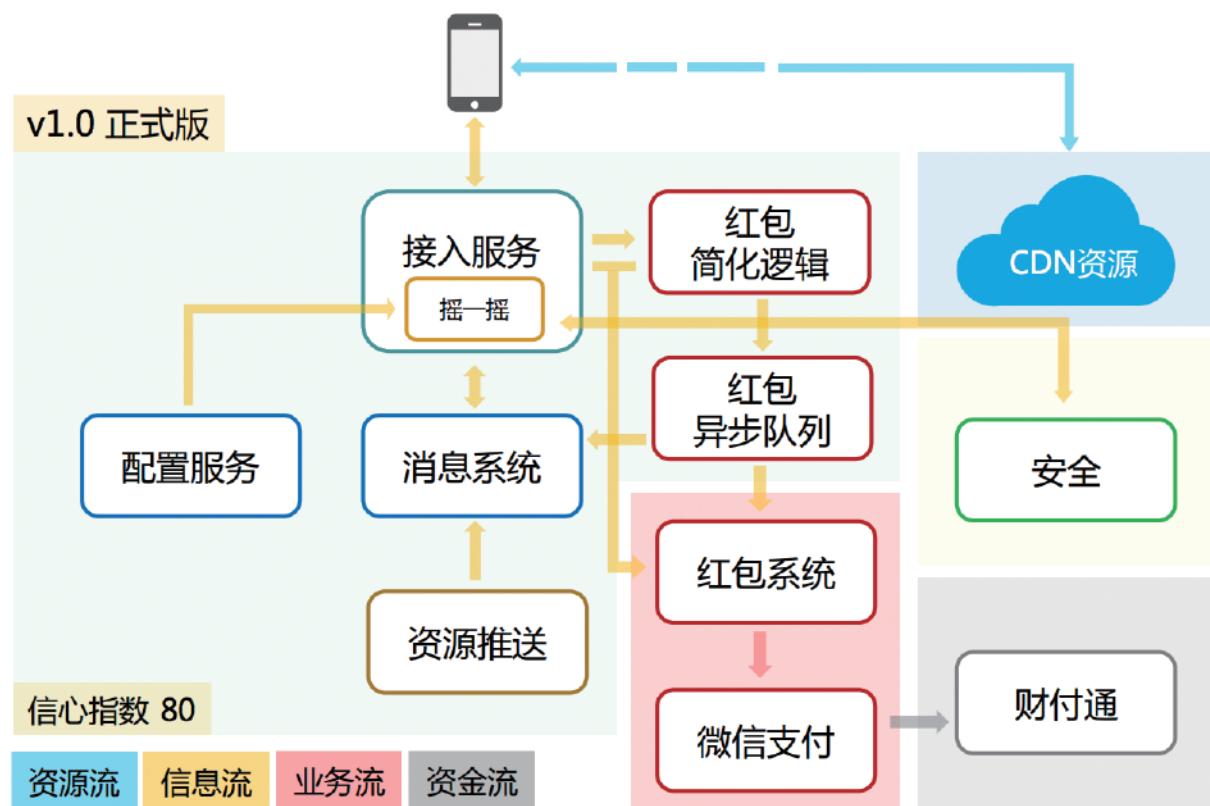
剩下 20% 在哪里？有 10% 是在现场，现场不可能一帆风顺，有可能现场摇的很 High，但后面到处灭火，10% 是在现场处置的时候有比较好的预案和方案能够解决，另外 10% 是人

算不如天算，一个很小的点出现问题导致被放大所有用户受影响也是有可能的，我们很难控制了。要做出完美无缺的基本不太可能，剩下 10% 就是留运气。

2 月 18 号跑出来的结果是这样的，当时摇了 110 亿次，峰值是 8.1 亿每分钟，1400 万每秒。

2015.2.18 春晚摇一摇

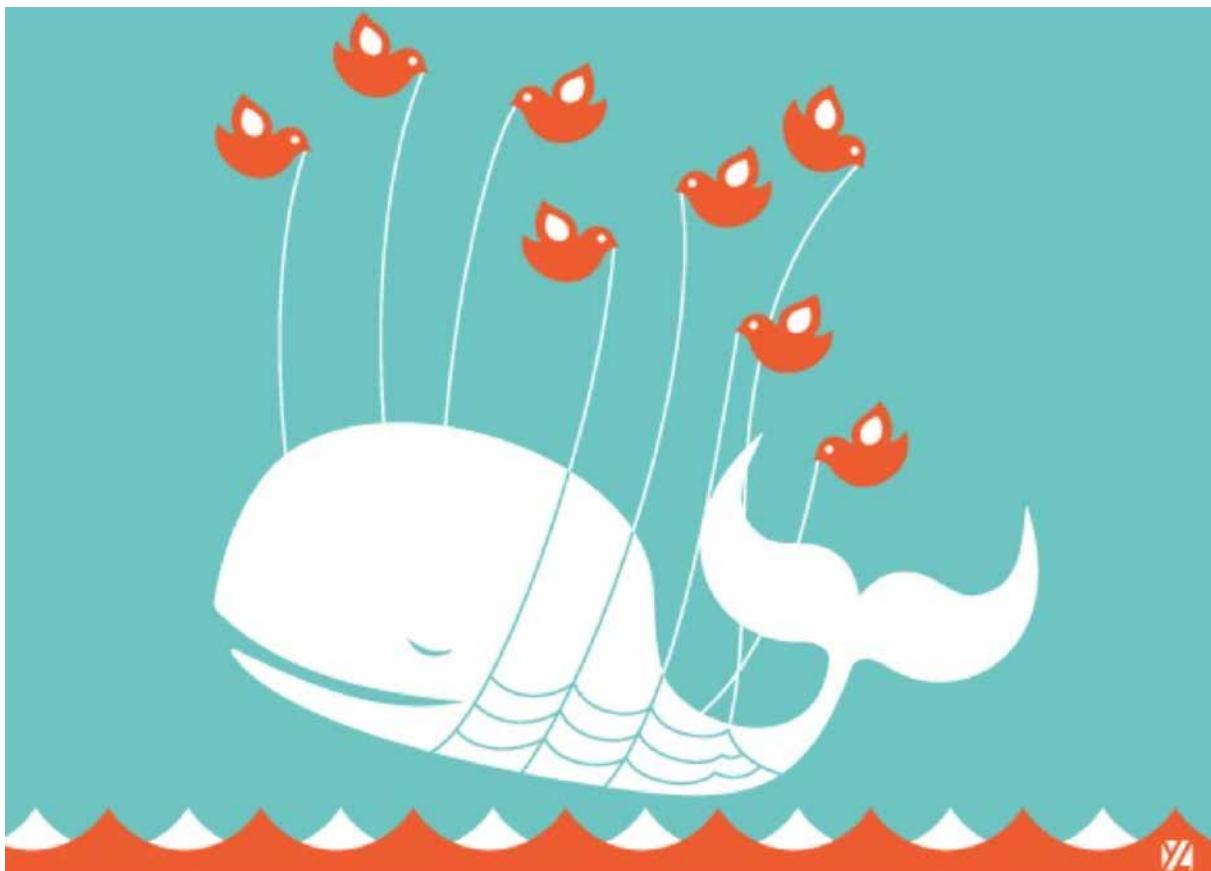
- 全程摇动 110 亿次
- 峰值 8.1 亿/分钟，1400 万/秒



“ArchSummit 会员微信群”是一个基于专家运营的高端技术社区。群内定期举行线上线下专家讲座、话题讨论，专注于“分享、交流、成长”。微信添加群主 cuikang10，申请入群。

讲师介绍：张文瑞，微信高级工程师，微信接入系统负责人，一直从事后台系统设计开发，早期涉足传统行业软件，后投身互联网。作为微信最早的后台开发之一，见证了微信从零开始到逐渐发展壮大的过程

百花齐放，锄其九九——Twitter 的技术坎坷之路



作者 藏秀涛

12月18日，年末技术盛会 ArchSummit 北京 2015 正式召开。Twitter Senior Staff Engineer 王天做主题演讲《百花齐放，锄其九九——Twitter 的技术坎坷之路》，分享了 Twitter 在技术演进过程中遇到的挑战和解决之道。本文即根据演讲内容整理而成。

首先，王天通过一组数字分享了 Twitter 的一些信息：

- 微博客始祖，成立于 2006 年
- 3.2 亿月活跃登录用户
- 10 亿月活跃独立访问用户（包括网站嵌入推文）
- 80% 流量来自移动设备
- 79% 流量来自美国以外
- 每日数亿条，每年逾 2000 亿条推文

- 4300 名员工，其中 44% 为工程师

Twitter 是一个和世界息息相关的实时信息平台，经常会遇到一些可预测或不可预测的事件，从而面临很大压力。

- 可预测的，比如：
 - 奥运会开闭幕式
 - NBA 决赛
 - NFL 决赛
 - 奥斯卡颁奖

- 日本《天空之城》重播 (2013.8)

不可预测的，比如：

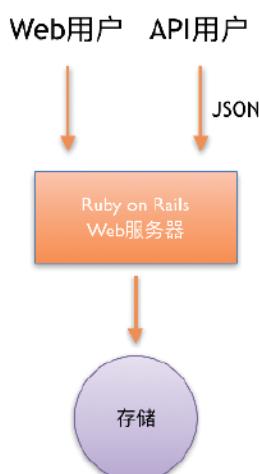
- 日本海啸 (2011.3)
- 世界杯德国巴西半决赛 (2014.7)
- 奥斯卡 Ellen 自拍事件 (2014.3)
- 巴黎恐怖袭击
- 巴西音乐节的奇怪网站

在去年的奥斯卡颁奖典礼现场，主持人 Ellen Lee DeGeneres 在 Twitter 上发了一张全明星自拍照，很多人去搜索、转发，给 Twitter 造成很大压力，致使系统宕机一段时间。之后，很多人又会去搜索 Twitter 宕机情况，情形进一步恶化。当遇到峰值无法应对的情况，则会出现 Twitter 特有的报错页面——Fail Whale。

Twitter 的技术历史：从远古到现代

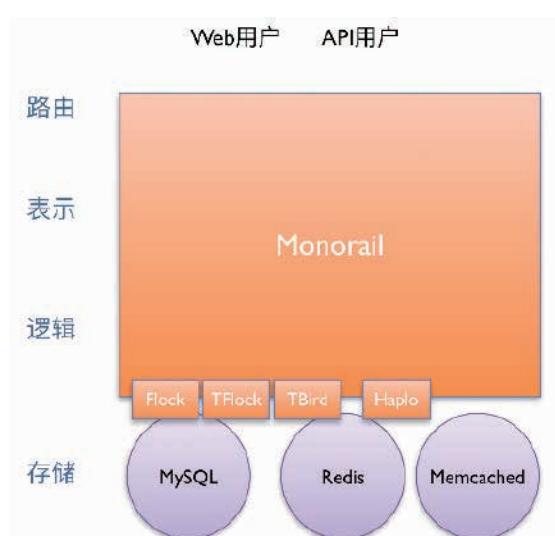
从 2006 年到 2015 年，回顾 Twitter 架构的十年演进之路，可以大概分为远古、古代、近代和现代 4 个时代来看。

1. 远古时代



最初创办时，创始人 Jack Dorsey 考虑过用 Python、C 和 OCaml 编写。不过机缘巧合，他找到了 Ruby on Rails 的核心贡献用 RoR 实现。

2. 古代



随着 Twitter 用户规模不断增长，其 Ruby on Rails 部署规模已经是世界第一，最多时机器达到 3000 台。如图所示，所有逻辑都在 Monorail 中。当时有超过 200 名工程师往里面 check in 代码。难以加入新功能，发布周期很长。

这个架构存在的问题是：

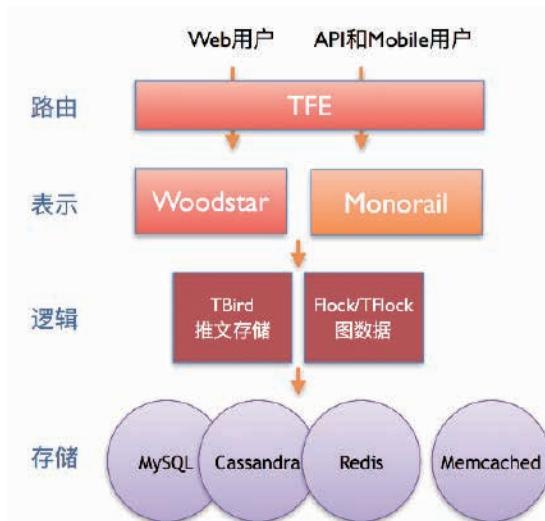
- 效率低下，延迟长，同步处理请求
- 单一数据库，热点明显
- 性能改善缓慢，增添机器的无底洞

当时没有很好地挺过 2010 年世界杯的考验。

技术债累积迅速，这也是很大的一个问题。

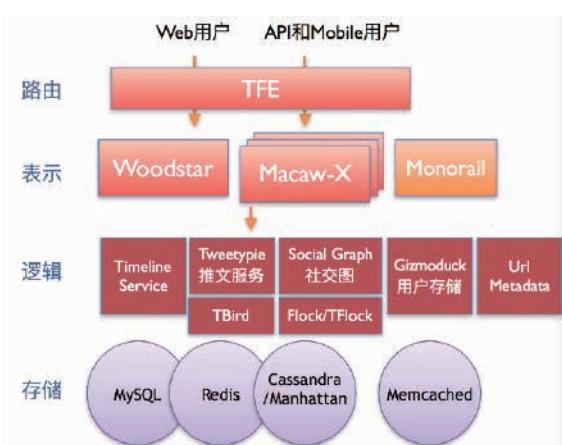
3. 近代

这一阶段可以用两张图表示。



为减少耦合，对系统进行拆分。为提高效率，用 Scala 重写了服务器。在网络异步编程方面，开发了 Finagle，这是基于 Netty 的一个异步编程库，也是用 Scala 编写的。存储方面，尝试创建较为高级的数据服务。

经过进一步分解，Monorail 逐渐被分离出来。更多业务被分解出来；团队围绕模块组织。



这个时期最重要的事情就是 Monorail 退休了。整个系统从 Ruby 平台迁移到 JVM 上。单机 QPS 处理能力从 200~300 提高到 10000~20000，延迟减小到 1/3；减少了 90% 资源使用。

4. 现代：产品系统和周边支持

走到这一步，实际经过了非常多的系统拆分。时至今日，Twitter 已经搭建起完整的工程生态。

回顾发展历史，服务化是很重要的变化。最初，所有的东西都在一个大系统中，知识无法压缩，开发人员要关注很多东西；而在服务化之后，开发人员可以将精力放到具体的业务逻辑上，同时享受质量可以预测的服务。

下面再用一张图回顾一下 Twitter 这 10 年的技术演进史。

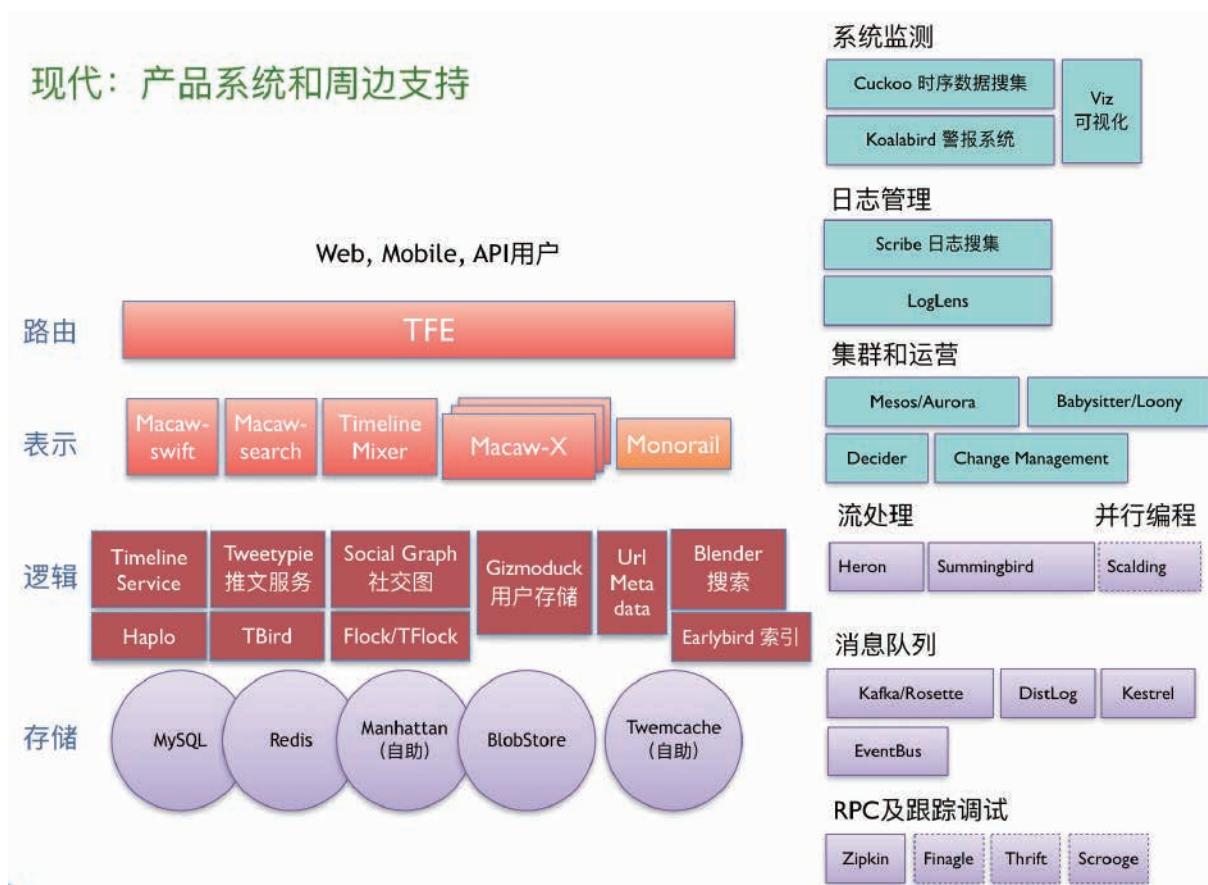
纵观拆分过程，可以总结出逻辑存在的不同形式。逻辑放到一起，就是单体结构。通过关注点分离，慢慢拆开，将逻辑拆到不同的模块中。通过服务化或平台化，可以将逻辑放到服务或平台中。具体如下图所示。

服务、平台和技术三者是可以相互转化的。红色的路径比较理想，而绿色则比较糟糕。

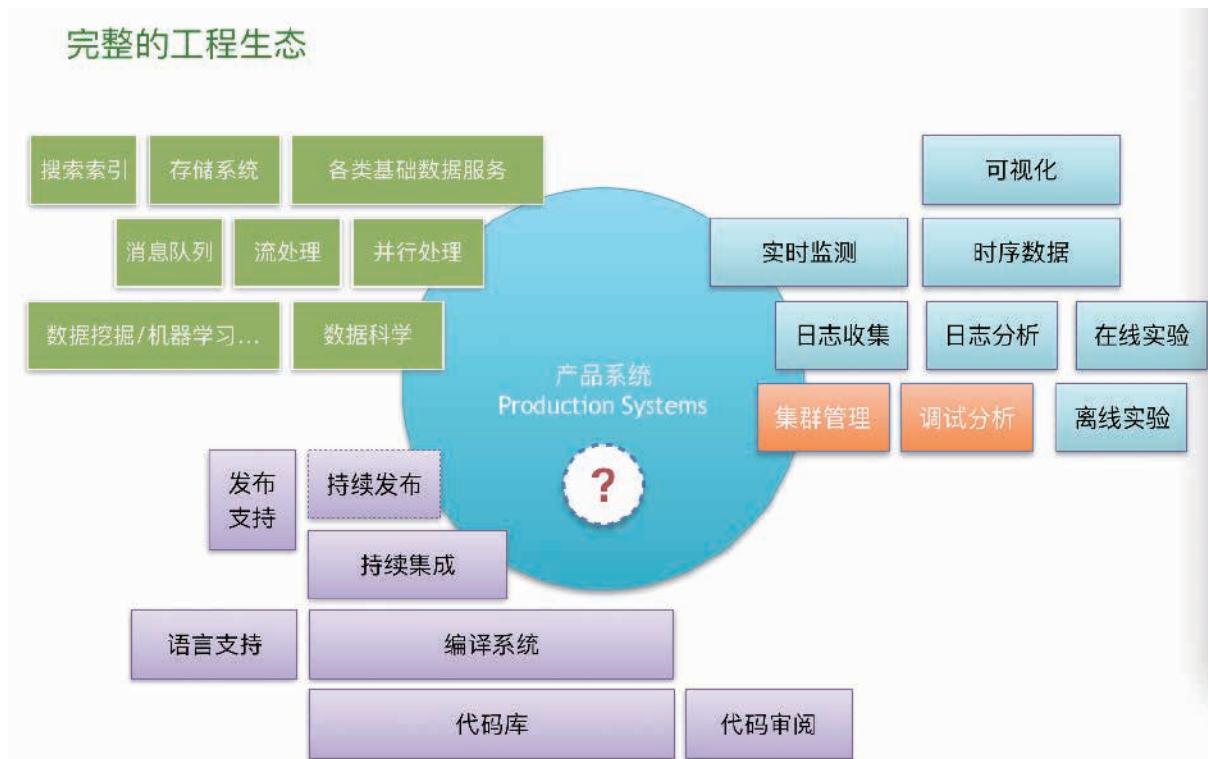
当把逻辑变成一个服务、技术或者平台时，这时候要慎重考虑，以对待顾客的方式对待使用该服务的同事。那么，如何当好一个服务生呢？

有几条要领：

- 用顾客需求驱动你的设计
- 最简可行产品 (MVP)
 - 不要实现既没有人需要也不能给你提供规划反馈的功能
- 尽早实现效益
 - 部署之际已经能服务第一个客户
- 考虑多顾客支持
 - 保证足够的灵活性
- 尽早实现效益

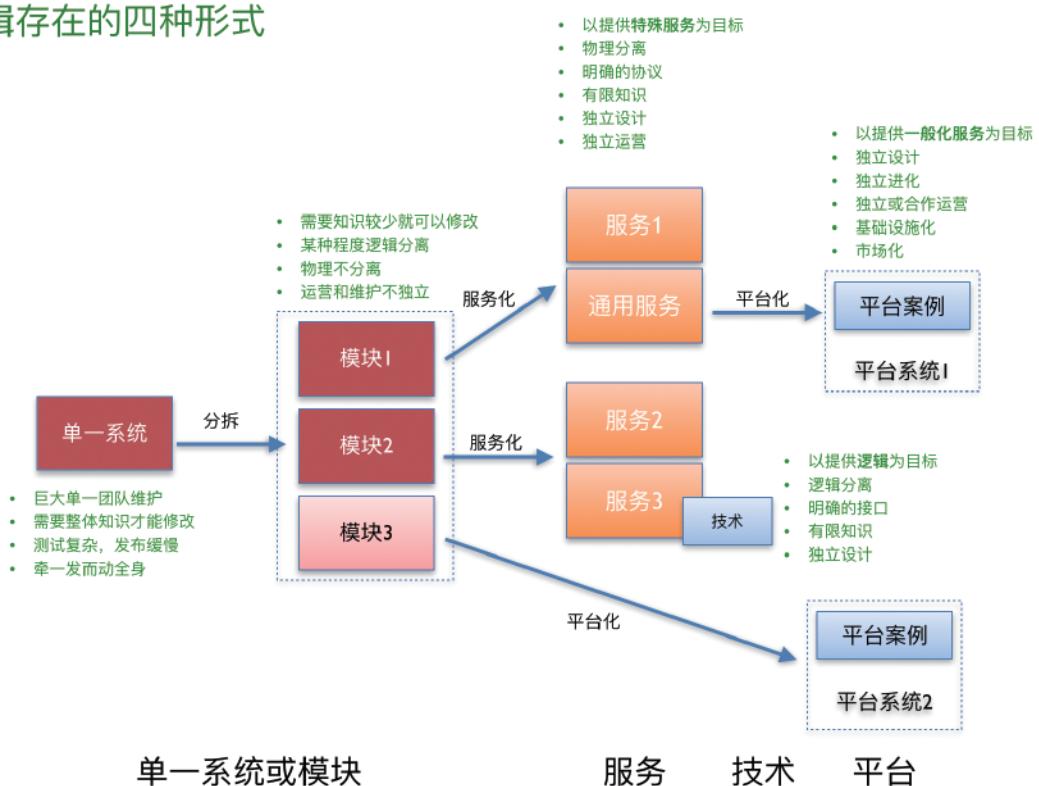


完整的工程生态





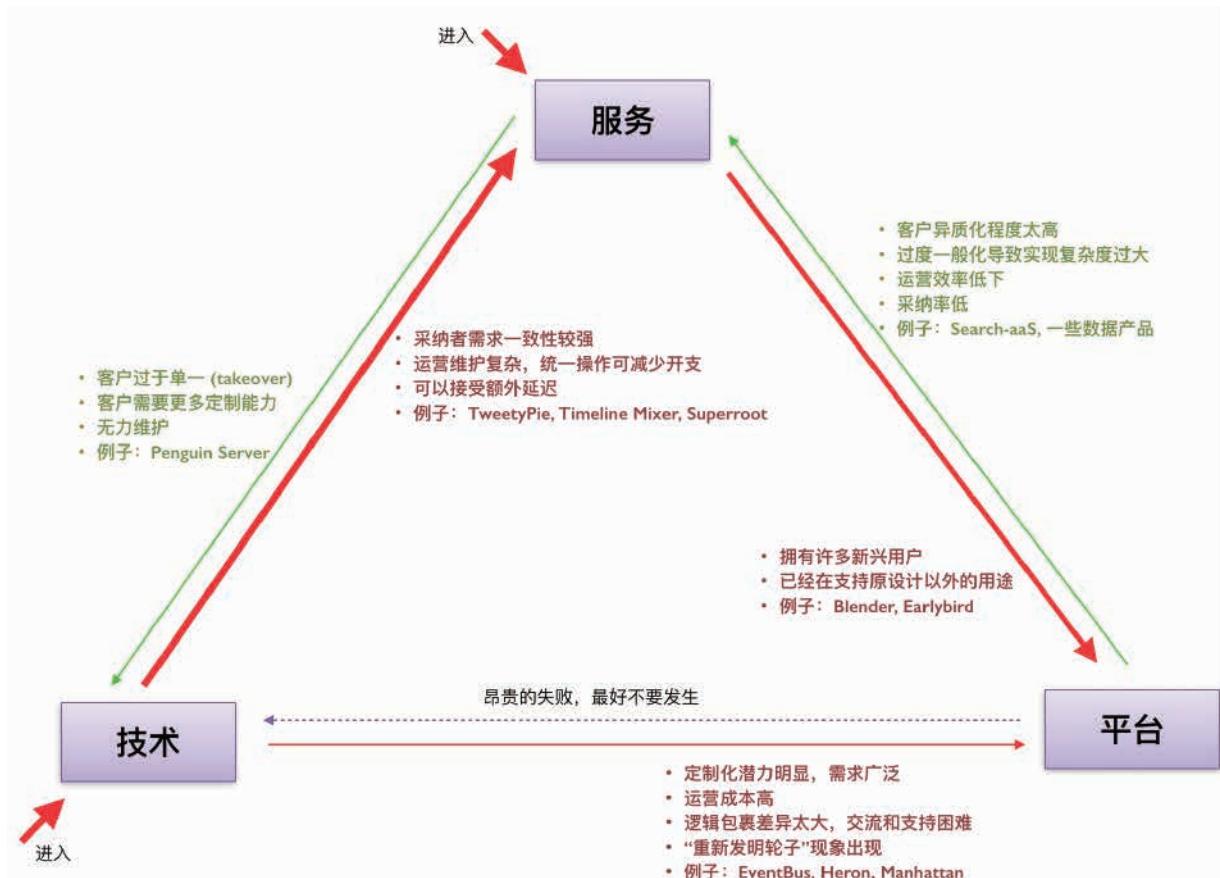
逻辑存在的四种形式



- 上马之际就能服务第一个客户
- 注重客户体验
 - 好用的才会被采纳，被采纳的才能存活
- 用服务的语言来交流
- 明确服务期望（服务级别协议：SLA）
- 思考“收费”模式
- 创造市场和社区

为方便他人使用，你可能需要提供设计文档、上手文档、示例代码、监测工具，并提供客户支持，还要协调未来功能规划，等等。

对于自己的服务、平台或技术，还要持之以恒地推广。包括推广你的服务和实践；扩大其用户群，增加采纳率；思考和类似服务共生



和竞争的关系。

一些设计底线:

- 规范设计过程
- 设计文档
- 设计审核会议
- 确保符合当前最佳实践
- 有意识地提升工程质量底线
- 设立设计排查清单
- 充分讨论新技术引入的集成代价和支持

代价

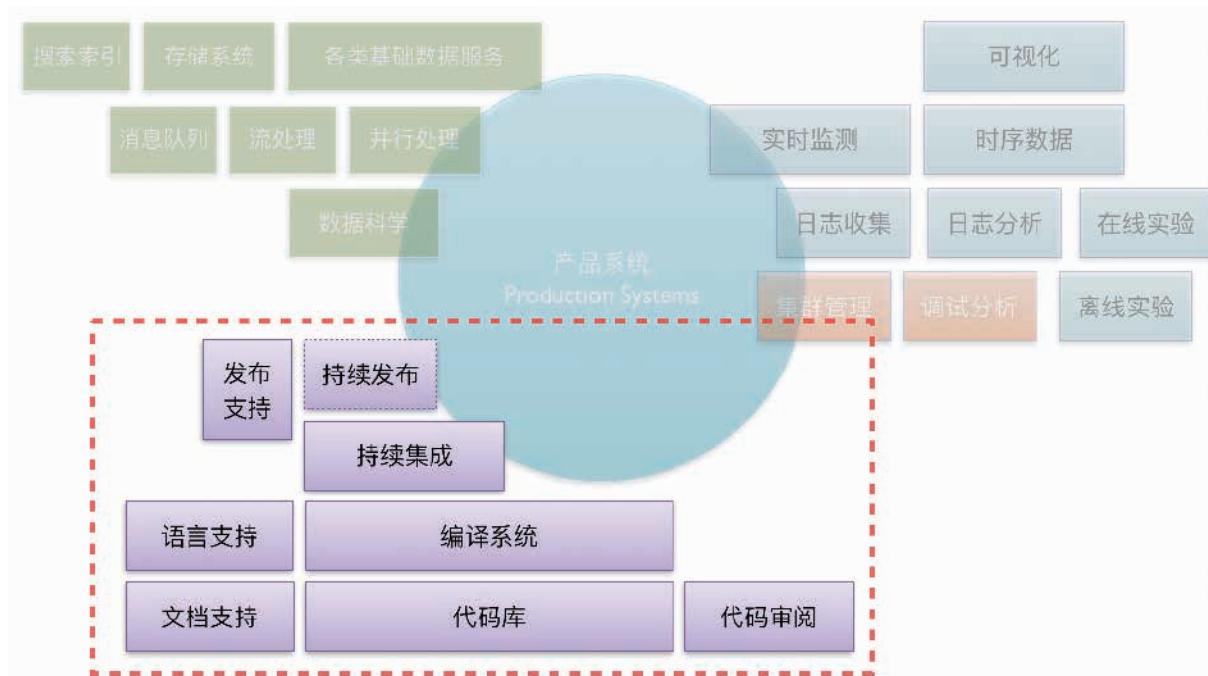
- 公开和协作
- 尽早引入利益方参与讨论
- 尽早思考产品化过程
- 设计导师: Design Shepherd

工程支持: 磨刀不误砍柴工

在完整的工程生态中, 有一些是幕后英雄: 它们和产品并没有太大的关系, 主要是语言支持、编译构建等, 但是这些东西是工程师使用最多的。提高这些东西的效率实际上对整个工程效率影响非常大。

假设一个工程师一年工作 2000 小时 (250 天):

效率提高比例	节省时间
1%	4.8分钟/天
2%	9.6分钟/天
5%	2.8小时/周
10%	2天/月



工程支持消耗的资源有限，但是可以让大部分工程师把精力用在刀刃上。

工程师的效率模型：

再来看一下 Twitter 的语言支持演变：

可以看到后面在收缩。语言过多还是会影响高效沟通。应该尽量减少。

Twitter 的代码库和编译系统演进：

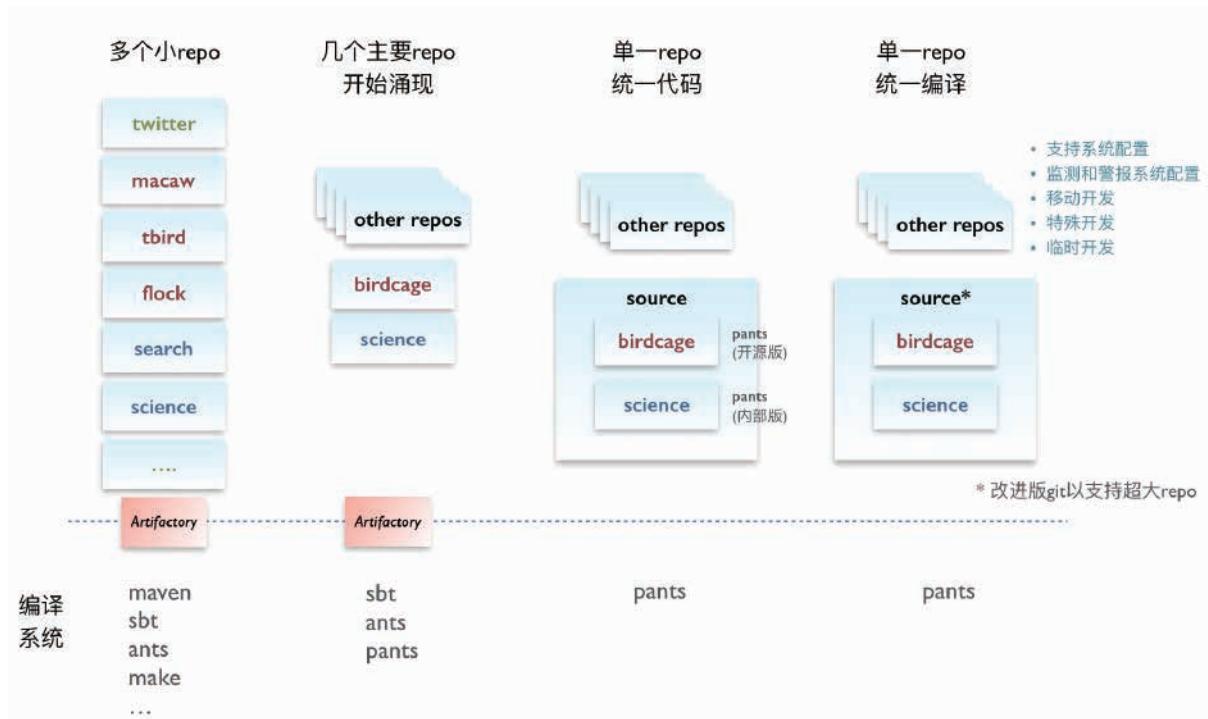
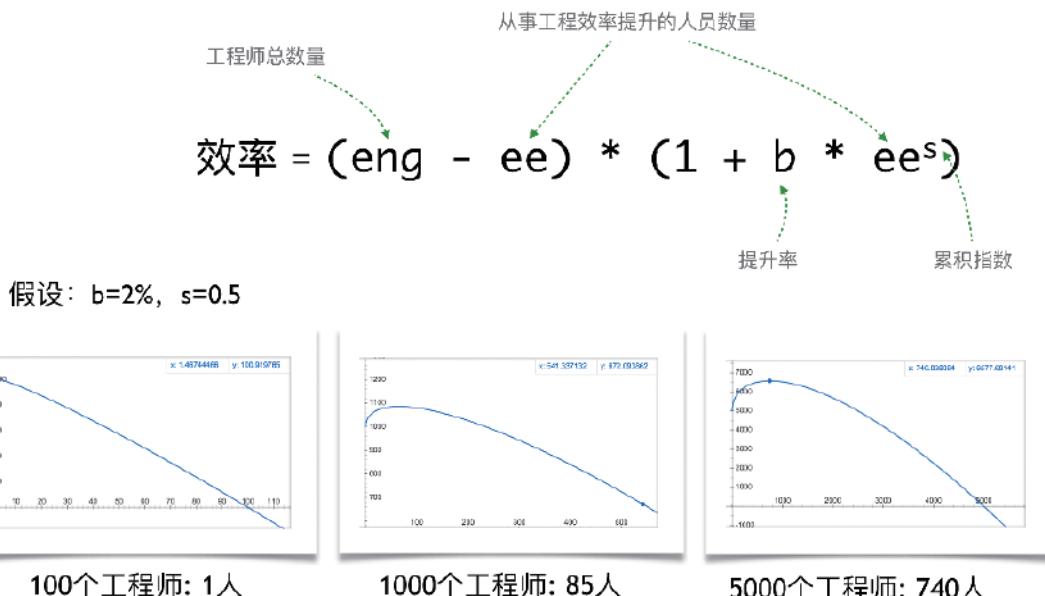
Twitter 最开始有多个代码库。现在是单一 repo，统一编译。其优势是，开发者能看到最新的、所有的东西。代码对所有人可见，可以直接协作。不过代码量非常庞大的情况下，这么做成本也非常高，像 Twitter 就自己修改了 git 之类的工具。可以说这是一个哲学选择。

其他工具：

- ReviewBoard：代码审核工具

- 经过扩展以匹配公司代码审阅流程
- JIRA：任务规划和追踪
- 各团队自行选择任务产生、分配和规划方式
- Confluence：公司内部 Wiki
- 维护团队文档、内部资料，指南等
- HipChat：聊天室
- DocBird：自开发和代码库集成的技术文档系统
- Google Docs
- 协同编辑和审阅文档，共享文档、表格、幻灯片
- Google Calendar
- 日历安排和协调

工程师的效率模型



开源：

Twitter有很多项目都在 github.com/twitter 上开源了。

“ArchSummit 会员微信群”是一个基于专家运营的高端技术社区。群内定期举行线上线下专家讲座、话题讨论，专注于“分享、交流、成长”。微信添加群主 cuikang10，申请入群。

讲师介绍：王天，2005年7月加入Google，从事移动搜索、新闻搜索、搜索质量等工作；2011年3月加入Twitter搜索部门，工作至今。他主要带领Twitter的搜索质量团队，改进实时搜索产品。