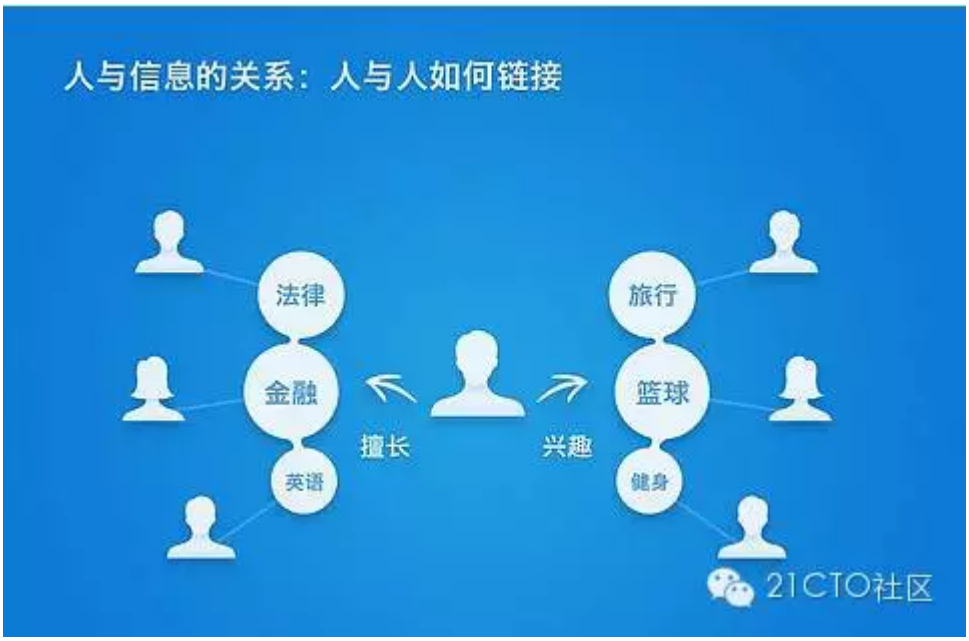
从0到100：知乎网站架构变迁史

21CTO 2015-12-02

导语

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。

知乎联合创始人兼 CTO 李申申的知乎创业三年多来的首次全面技术分享（点击阅读全文可获取幻灯片文件）。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。创业就是这样一个情况，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，Github上可见。）

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。

> 原文：http://www.codeceo.com/article/zhihu-artch.html

关于21CTO

21CTO.com是中国互联网第一技术人脉与社交平台。我们为国内最优秀的开发者提供社交、学习等产品，帮助企业快速对接开发者，包括人才招聘，项目研发，顾问咨询服务。
看微信文章不过瘾，请移步到网站，诚挚欢迎您加入社区作者团队。
网站地址：www.21cto.com
投稿邮箱：info@21cto.com
QQ群： 79309783 （欢迎扫描下列二维码关注本微信号）

阅读原文