解剖Twitter：Twitter系统架构设计分析

2010/10/30

随着信息爆炸的加剧，微博客网站Twitter横空出世了。用横空出世这个词来形容Twitter的成长，并不夸张。从2006年5月Twitter上线，到2007年12月，一年半的时间里，Twitter用户数从0增长到6.6万。又过了一年，2008年12月，Twitter的用户数达到5百万。[1]

Twitter网站的成功，先决条件是能够同时给千万用户提供服务，而且提供服务的速度要快。[2,3,4]

有观点认为，Twitter的业务逻辑简单，所以竞争门槛低。前半句正确，但是后半句有商榷余地。Twitter的竞争力，离不开严谨的系统架构设计。

【1】万事开头易

Twitter的核心业务逻辑，在于Following和Be followed。[5]

进入Twitter个人主页，你会看到你following的那些作者，最近发表的微博客。所谓微博客，就是一则短信，Twitter规定，短信的长度不得超过140个字。短信不仅可以包含普通文字信息，也可以包含URL，指向某个网页，或者照片及视频等等。这就是following的过程。

当你写了一则短信并发表以后，你的followers会立刻在他们的个人主页中看到你写的最新短信。这就是be followed的过程。

实现这个业务流程似乎很容易。

1. 为每一个注册用户订制一个Be-followed的表，主要内容是每一个follower的ID。同时，也订制一个Following的表，主要内容是每一个following作者的ID。

2. 当用户打开自己的个人空间时，Twitter先查阅Following表，找到所有following的作者的ID。然后去数据库读取每一位作者最近写的短信。汇总后按时间顺序显示在用户的个人主页上。

3. 当用户写了一则短信时，Twitter先查阅Be-followed表，找到所有followers的IDs。然后逐个更新那些followers的主页。

如果有follower正在阅读他的Twitter个人主页，主页里暗含的JavaScript会自动每隔几十秒，访问一下Twitter服务器，检查正在看的这个个人主页是否有更新。如果有更新，立刻下载新的主页内容。这样follower就能读到最新发表的短信了。

从作者发表到读者获取，中间的延迟，取决于JavaScript更新的间隔，以及Twitter服务器更新每个follower的主页的时间。

从系统架构上来说，似乎传统的三段论(Three-tier architecture [6])，足够满足这个业务逻辑的需要。事实上，最初的Twitter系统架构，的确就是三段论。

Reference：

[1] Fixing Twitter. （http://www.bookfm.com/courseware/coursewaredetail.html?cid=100777）
[2] Twitter blows up at SXSW conference. （http://gawker.com/tech/next-big-thing/twitter-blows-up-at-sxsw-conference-243634.php）
[3] First Hand Accounts of Terrorist Attacks in India on Twitter and Flickr.
（http://www.techcrunch.com/2008/11/26/first-hand-accounts-of-terrorist-attacks-in-india-on-twitter/）

[4] Social Media Takes Center Stage in Iran. （http://www.findingdulcinea.com/news/technology/2009/June/Twitter-on-Iran-a-Go-to-Source-or-Almost-Useless.html）
[5] Twitter的这些那些. (http://www.ccthere.com/article/2363334) (http://www.ccthere.com/article/2369092)
[6] Three tier architecture. http://en.wikipedia.org/wiki/Multitier_architecture

【2】三段论

网站的架构设计，传统的做法是三段论。所谓“传统的”，并不等同于“过时的”。大型网站的架构设计，强调实用。新潮的设计，固然吸引人，但是技术可能不成熟，风险高。所以，很多大型网站，走的是稳妥的传统的路子。

2006年5月Twitter刚上线的时候，为了简化网站的开发，他们使用了Ruby-On-Rails工具，而Ruby-On-Rails的设计思想，就是三段论。

1. 前段，即表述层(Presentation Tier) 用的工具是Apache Web Server，主要任务是解析HTTP协议，把来自不同用户的，不同类型的请求，分发给逻辑层。

2. 中段，即逻辑层 (Logic Tier）用的工具是Mongrel Rails Server，利用Rails现成的模块，降低开发的工作量。

3. 后段，即数据层 (Data Tier) 用的工具是MySQL 数据库。

先说后段，数据层。

Twitter 的服务，可以概括为两个核心，1. 用户，2. 短信。用户与用户之间的关系，是追与被追的关系，也就是Following和Be followed。对于一个用户来说，他只读自己“追”的那些人写的短信。而他自己写的短信，只有那些“追”自己的人才会读。抓住这两个核心，就不难理解 Twitter的其它功能是如何实现的[7]。

围绕这两个核心，就可以着手设计Data Schema，也就是存放在数据层(Data Tier)中的数据的组织方式。不妨设置三个表[8]，

1. 用户表：用户ID，姓名，登录名和密码，状态（在线与否）。

2. 短信表：短信ID，作者ID，正文（定长，140字），时间戳。

3. 用户关系表，记录追与被追的关系：用户ID，他追的用户IDs (Following)，追他的用户IDs (Be followed)。

再说中段，逻辑层。

当用户发表一条短信的时候，执行以下五个步骤，

1. 把该短信记录到“短信表” 中去。

2. 从“用户关系表”中取出追他的用户的IDs。

3. 有些追他的用户目前在线，另一些可能离线。在线与否的状态，可以在“用户表”中查到。过滤掉那些离线的用户的IDs。

4. 把那些追他的并且目前在线的用户的IDs，逐个推进一个队列(Queue)中去。

5. 从这个队列中，逐个取出那些追他的并且目前在线的用户的IDs，并且更新这些人的主页，也就是添加最新发表的这条短信。

以上这五个步骤，都由逻辑层(Logic Tier)负责。前三步容易解决，都是简单的数据库操作。最后两步，需要用到一个辅助工具，队列。队列的意义在于，分离了任务的产生与任务的执行。

队列的实现方式有多种，例如Apache Mina[9]就可以用来做队列。但是Twitter团队自己动手实现了一个队列，Kestrel [10,11]。Mina与Kestrel，各自有什么优缺点，似乎还没人做过详细比较。

不管是Kestrel还是Mina，看起来都很复杂。或许有人问，为什么不用简单的数据结构来实现队列，例如动态链表，甚至静态数组？如果逻辑层只在一台服务器上运行，那么对动态链表和静态数组这样的简单的数据结构，稍加改造，的确可以当作队列使用。Kestrel和Mina这些“重量级”的队列，意义在于支持联络多台机器的，分布式的队列。在本系列以后的篇幅中，将会重点介绍。

最后说说前段，表述层。

表述层的主要职能有两 个，1. HTTP协议处理器(HTTP Processor)，包括拆解接收到的用户请求，以及封装需要发出的结果。2. 分发器(Dispatcher)，把接收到的用户请求，分发给逻辑层的机器处理。如果逻辑层只有一台机器，那么分发器无意义。但是如果逻辑层由多台机器组成，什么样的请求，发给逻辑层里面哪一台机器，就大有讲究了。逻辑层里众多机器，可能各自专门负责特定的功能，而在同功能的机器之间，要分摊工作，使负载均衡。

访问Twitter网站的，不仅仅是浏览器，而且还有手机，还有像QQ那样的电脑桌面工具，另外还有各式各样的网站插件，以便把其它网站联系到Twitter.com上来[12]。因此，Twitter的访问者与Twitter网站之间的通讯协议，不一定是HTTP，也存在其它协议。

三段论的Twitter架构，主要是针对HTTP协议的终端。但是对于其它协议的终端，Twitter的架构没有明显地划分成三段，而是把表述层和逻辑层合二为一，在Twitter的文献中，这二合一经常被称为“API”。

综上所述，一个能够完成Twitter基本功能的，简单的架构如Figure 1 所示。或许大家会觉得疑惑，这么出名的网站，架构就这么简单？Yes and No，2006年5月Twitter刚上线的时候，Twitter架构与Figure 1差距不大，不一样的地方在于加了一些简单的缓存(Cache)。即便到了现在，Twitter的架构依然可以清晰地看到Figure 1 的轮廓。

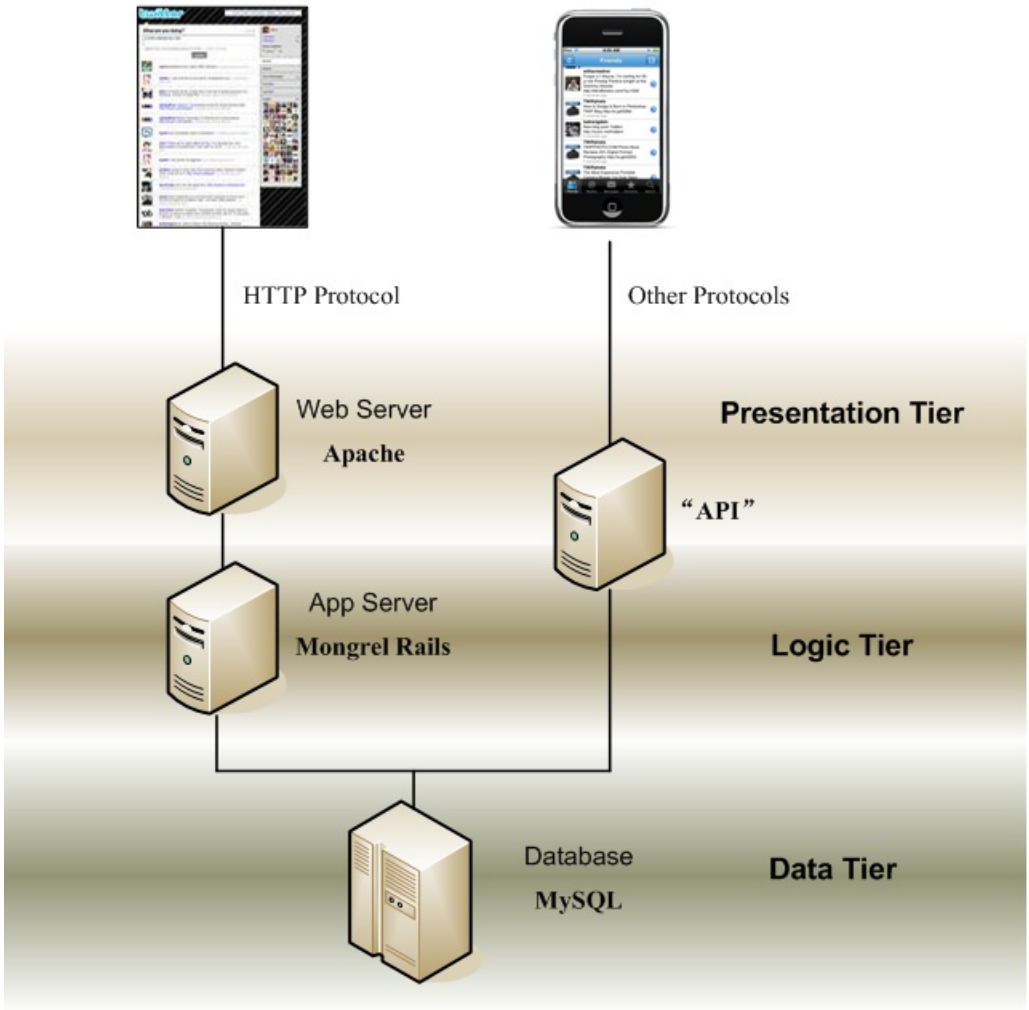


Figure 1. The essential 3-tier of Twitter architecture

Courtesy http://farm3.static.flickr.com/2683/4051785892_e677ae9d33_o.png

Reference,

[7] Tweets中常用的工具 （http://www.ccthere.com/article/2383833）
[8] 构建基于PHP的微博客服务 (http://webservices.ctocio.com.cn/188/9092188.shtml)
[9] Apache Mina Homepage (http://mina.apache.org/)
[10] Kestrel Readme (http://github.com/robey/kestrel)
[11] A Working Guide to Kestrel. (http://github.com/robey/kestrel/blob/master/docs/guide.md)
[12] Alphabetical List of Twitter Services and Applications
(http://en.wikipedia.org/wiki/List_of_Twitter_services_and_applications)

【3】Cache == Cash

Cache == Cash，缓存等于现金收入。虽然这话有点夸张，但是正确使用缓存，对于大型网站的建设，是至关重要的大事。网站在回应用户请求时的反应速度，是影响用户体验的一大因素。而影响速度的原因有很多，其中一个重要的原因在于硬盘的读写(Disk IO)。

Table 1 比较了内存(RAM)，硬盘(Disk)，以及新型的闪存(Flash)，在读写方面的速度比较。硬盘的读写，速度比内存的慢了百万倍。所以，要提高网站的速度，一个重要措施是尽可能把数据缓存在内存里。当然，在硬盘里也必须保留一个拷贝，以此防范万一由于断电，内存里的数据丢失的情况发生。

Media	Price	Access time			Transfer bandwidth	Power Cons. Act/Idle/Slp
		Read	Write	Erase		
Magnetic Disk	$0.3/GB	12.7 ms (2 KB)	13.7 ms (2 KB)	N/A	85 MB/s	13W/8W/1W
NAND Flash	$30/GB	80 µs (2 KB)	200 µs (2 KB)	1.5 ms (128 KB)	25 MB/s	1W/0.1W/ 0.1W
DDR2-533 RAM	$25/GB	22.5 ns	22.5 ns	N/A	4266 MB/s	12W/6W/NA per GB

Table 1. Storage media comparison of Disk, Flash and RAM [13]
Courtesy http://farm3.static.flickr.com/2736/4060534279_f575212c12_o.png

Twitter 工程师认为，一个用户体验良好的网站，当一个用户请求到达以后，应该在平均500ms以内完成回应。而Twitter的理想，是达到200ms- 300ms的反应速度[17]。因此在网站架构上，Twitter大规模地，多层次多方式地使用缓存。Twitter在缓存使用方面的实践，以及从这些实践中总结出来的经验教训，是Twitter网站架构的一大看点。

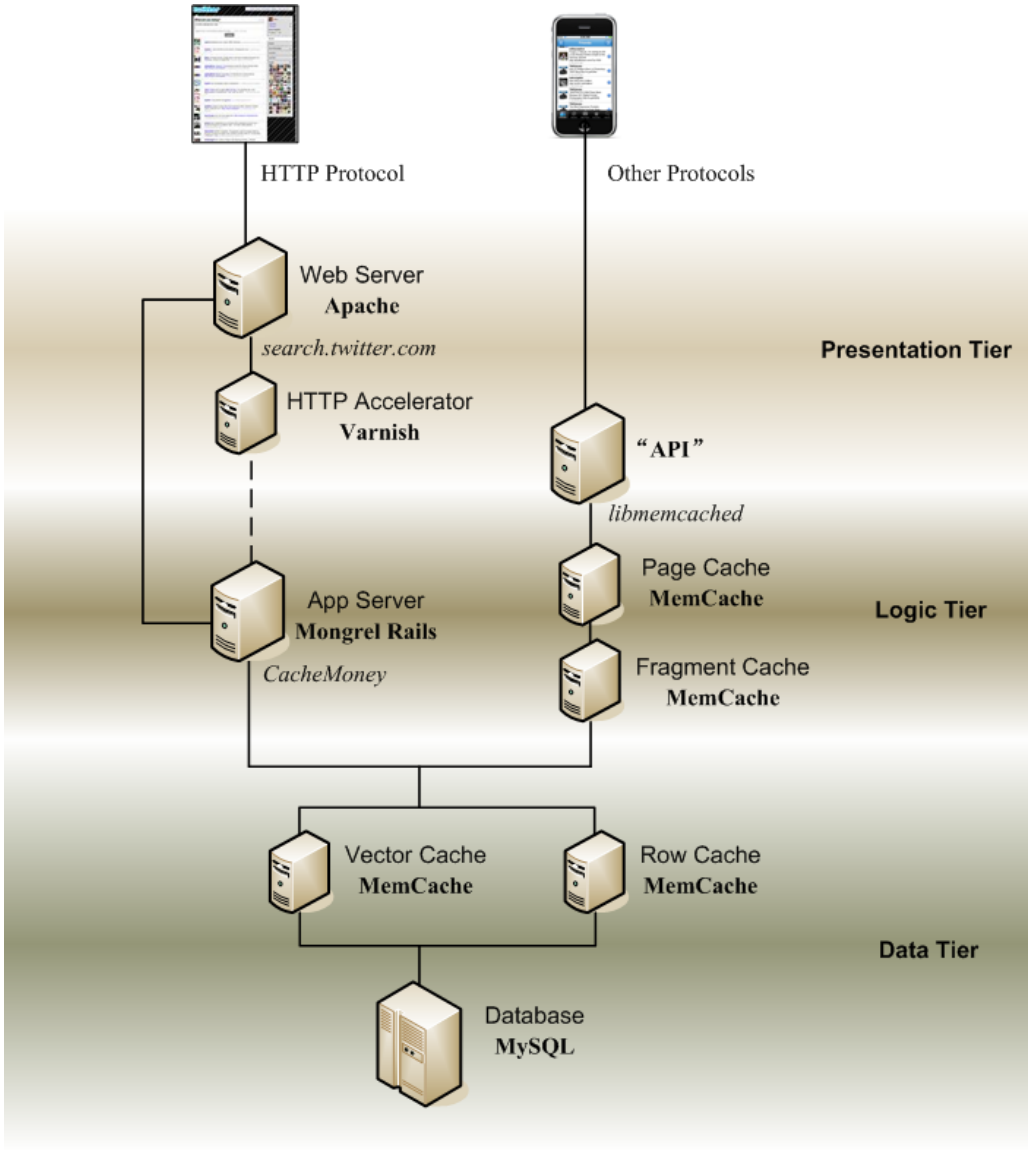


Figure 2. Twitter architecture with Cache
Courtesy http://farm3.static.flickr.com/2783/4065827637_bb2ccc8e3f_o.png

哪里需要缓存？越是Disk IO频繁的地方，越需要缓存。

前面说到，Twitter业务的核心有两个，用户和短信(Tweet)。围绕这两个核心，数据库中存放着若干表，其中最重要的有三个，如下所示。这三个表的设置，是旁观者的猜测，不一定与Twitter的设置完全一致。但是万变不离其宗，相信即便有所不同，也不会本质区别。

1. 用户表：用户ID，姓名，登录名和密码，状态（在线与否）。
2. 短信表：短信ID，作者ID，正文（定长，140字），时间戳。
3. 用户关系表，记录追与被追的关系：用户ID，他追的用户IDs (Following)，追他的用户IDs (Be followed)。

有没有必要把这几个核心的数据库表统统存放到缓存中去？Twitter的做法是把这些表拆解，把其中读写最频繁的列放进缓存。

1. Vector Cache and Row Cache

具体来说，Twitter工程师认为最重要的列是IDs。即新发表的短信的IDs，以及被频繁阅读的热门短信的IDs，相关作者的IDs，以及订阅这些作者的读者的IDs。把这些IDs存放进缓存 (Stores arrays of tweet pkeys [14])。在Twitter文献中，把存放这些IDs的缓存空间，称为Vector Cache [14]。

Twitter工程师认为，读取最频繁的内容是这些IDs，而短信的正文在其次。所以他们决定，在优先保证Vector Cache所需资源的前提下，其次重要的工作才是设立Row Cache，用于存放短信正文。

命中率(Hit Rate or Hit Ratio)是测量缓存效果的最重要指标。如果一个或者多个用户读取100条内容，其中99条内容存放在缓存中，那么缓存的命中率就是99%。命中率越高，说明缓存的贡献越大。

设立Vector Cache和Row Cache后，观测实际运行的结果，发现Vector Cache的命中率是99%，而Row Cache的命中率是95%，证实了Twitter工程师早先押注的，先IDs后正文的判断。

Vector Cache和Row Cache，使用的工具都是开源的MemCached [15]。

2. Fragment Cache and Page Cache

前文说到，访问Twitter网站的，不仅仅是浏览器，而且还有手机，还有像QQ那样的电脑桌面工具，另外还有各式各样的网站插件，以便把其它网站联系到Twitter.com上来[12]。接待这两类用户的，是以Apache Web Server为门户的Web通道，以及被称为“API”的通道。其中API通道受理的流量占总流量的80%-90% [16]。

所以，继Vector Cache和Row Cache以后，Twitter工程师们把进一步建筑缓存的工作，重点放在如何提高API通道的反应速度上。

读者页面的主体，显示的是一条又一条短信。不妨把整个页面分割成若干局部，每个局部对应一条短信。所谓Fragment，就是指页面的局部。除短信外，其它内容例如Twitter logo等等，也是Fragment。如果一个作者拥有众多读者，那么缓存这个作者写的短信的布局页面(Fragment)，就可以提高网站整体的读取效率。这就是Fragment Cache的使命。

对于一些人气很旺的作者，读者们不仅会读他写的短信，而且会访问他的主页，所以，也有必要缓存这些人气作者的个人主页。这就是Page Cache的使命。

Fragment Cache和Page Cache，使用的工具也是MemCached。

观测实际运行的结果，Fragment Cache的命中率高达95%，而Page Cache的命中率只有40%。虽然Page Cache的命中率低，但是它的内容是整个个人主页，所以占用的空间却不小。为了防止Page Cache争夺Fragment Cache的空间，在物理部署时，Twitter工程师们把Page Cache分离到不同的机器上去。

3. HTTP Accelerator

解决了API通道的缓存问题，接下去Twitter工程师们着手处理Web通道的缓存问题。经过分析，他们认为Web通道的压力，主要来自于搜索。尤其是面临突发事件时，读者们会搜索相关短信，而不理会这些短信的作者，是不是自己“追”的那些作者。

要降低搜索的压力，不妨把搜索关键词，及其对应的搜索结果，缓存起来。Twitter工程师们使用的缓存工具，是开源项目Varnish [18]。

比较有趣的事情是，通常把Varnish部署在Web Server之外，面向Internet的位置。这样，当用户访问网站时，实际上先访问Varnish，读取所需内容。只有在Varnish没有缓存相应内容时，用户请求才被转发到Web Server上去。而Twitter的部署，却是把Varnish放在Apache Web Server内侧[19]。原因是Twitter的工程师们觉得Varnish的操作比较复杂，为了降低Varnish崩溃造成整个网站瘫痪的可能性，他们便采取了这种古怪而且保守的部署方式。

Apache Web Server的主要任务，是解析HTTP，以及分发任务。不同的Mongrel Rails Server负责不同的任务，但是绝大多数Mongrel Rails Server，都要与Vector Cache和Row Cache联系，读取数据。Rails Server如何与MemCached联系呢？Twitter工程师们自行开发了一个Rails插件(Gem)，称为CacheMoney。

虽然Twitter没有公开Varnish的命中率是多少，但是[17]声称，使用了Varnish以后，导致整个Twitter.com网站的负载下降了50%，参见Figure 3.

Figure 3. Cache decreases Twitter.com load by 50% [17]
Courtesy http://farm3.static.flickr.com/2537/4061273900_2d91c94374_o.png

Reference,

[12] Alphabetical List of Twitter Services and Applications.
(http://en.wikipedia.org/wiki/List_of_Twitter_services_and_applications)
[13] How flash changes the DBMS world.
(http://hansolav.net/blog/content/binary/HowFlashMemory.pdf)
[14] Improving running component of Twitter.
(http://qconlondon.com/london-2009/file?path=/qcon-london-2009/slides/
EvanWeaver_ImprovingRunningComponentsAtTwitter.pdf)
[15] A high-performance, general-purposed, distributed memory object caching system.
(http://www.danga.com/memcached/)
[16] Updating Twitter without service disruptions.
(http://gojko.net/2009/03/16/qcon-london-2009-upgrading-twitter-without-service-disruptions/)
[17] Fixing Twitter. (http://assets.en.oreilly.com/1/event/29/
Fixing_Twitter_Improving_the_Performance_and_Scalability_of_the_World_s_
Most_Popular_Micro-blogging_Site_Presentation%20Presentation.pdf)
[18] Varnish, a high-performance HTTP accelerator.
(http://varnish.projects.linpro.no/)
[19] How to use Varnish in Twitter.com?
(http://projects.linpro.no/pipermail/varnish-dev/2009-February/000968.html)
[20] CacheMoney Gem, an open-source write-through caching library.
(http://github.com/nkallen/cache-money)

【4】抗洪需要隔离

如果说如何巧用Cache是Twitter的一大看点，那么另一大看点是它的消息队列(Message Queue)。为什么要使用消息队列？[14]的解释是“隔离用户请求与相关操作，以便烫平流量高峰 (Move operations out of the synchronous request

cycle, amortize load over time)”。

为了理解这段话的意思，不妨来看一个实例。2009年1月20日星期二，美国总统Barack Obama就职并发表演说。作为美国历史上第一位黑人总统，Obama的就职典礼引起强烈反响，导致Twitter流量猛增，如Figure 4 所示。

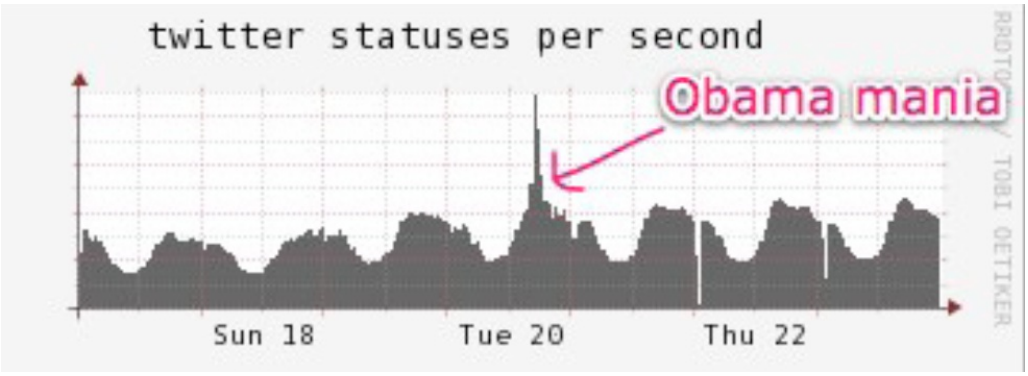


Figure 4. Twitter burst during the inauguration of Barack Obama, 1/20/2009, Tuesday
Courtesy http://farm3.static.flickr.com/2615/4071879010_19fb519124_o.png

其中洪峰时刻，Twitter网站每秒钟收到350条新短信，这个流量洪峰维持了大约5分钟。根据统计，平均每个Twitter用户被120人“追”，这就 是说，这350条短信，平均每条都要发送120次 [16]。这意味着，在这5分钟的洪峰时刻，Twitter网站每秒钟需要发送350 x 120 = 42,000条短信。

面对洪峰，如何才能保证网站不崩溃？办法是迅速接纳，但是推迟服务。打个比方，在晚餐高峰时段，餐馆常常客满。对于新来的顾客，餐馆服务员不是拒之门外，而是让这些顾客在休息厅等待。这就是[14] 所说的 “隔离用户请求与相关操作，以便烫平流量高峰”。

如何实施隔离呢？当一位用户访问Twitter网站时，接待他的是Apache Web Server。Apache做的事情非常简单，它把用户的请求解析以后，转发给Mongrel Rails Sever，由Mongrel负责实际的处理。而Apache腾出手来，迎接下一位用户。这样就避免了在洪峰期间，用户连接不上Twitter网站的尴尬局面。

虽然Apache的工作简单，但是并不意味着Apache可以接待无限多的用户。原因是Apache解析完用户请求，并且转发给 Mongrel Server以后，负责解析这个用户请求的进程(process)，并没有立刻释放，而是进入空循环，等待Mongrel Server返回结果。这样，Apache能够同时接待的用户数量，或者更准确地说，Apache能够容纳的并发的连接数量(concurrent connections)，实际上受制于Apache能够容纳的进程数量。Apache系统内部的进程机制参见Figure 5，其中每个Worker代表一个进程。

Apache能够容纳多少个并发连接呢？[22]的实验结果是4,000个，参见Figure 6。如何才能提高Apache的并发用户容量呢？一种思路是不让连接受制于进程。不妨把连接作为一个数据结构，存放到内存中去，释放进程，直到 Mongrel Server返回结果时，再把这个数据结构重新加载到进程上去。

事实上Yaws Web Server[24]，就是这么做的[23]。所以，Yaws能够容纳80,000以上的并发连接，这并不奇怪。但是为什么Twitter用 Apache，而不用Yaws呢？或许是因为Yaws是用Erlang语言写的，而Twitter工程师对这门新语言不熟悉(But you need in house Erlang experience [17])。

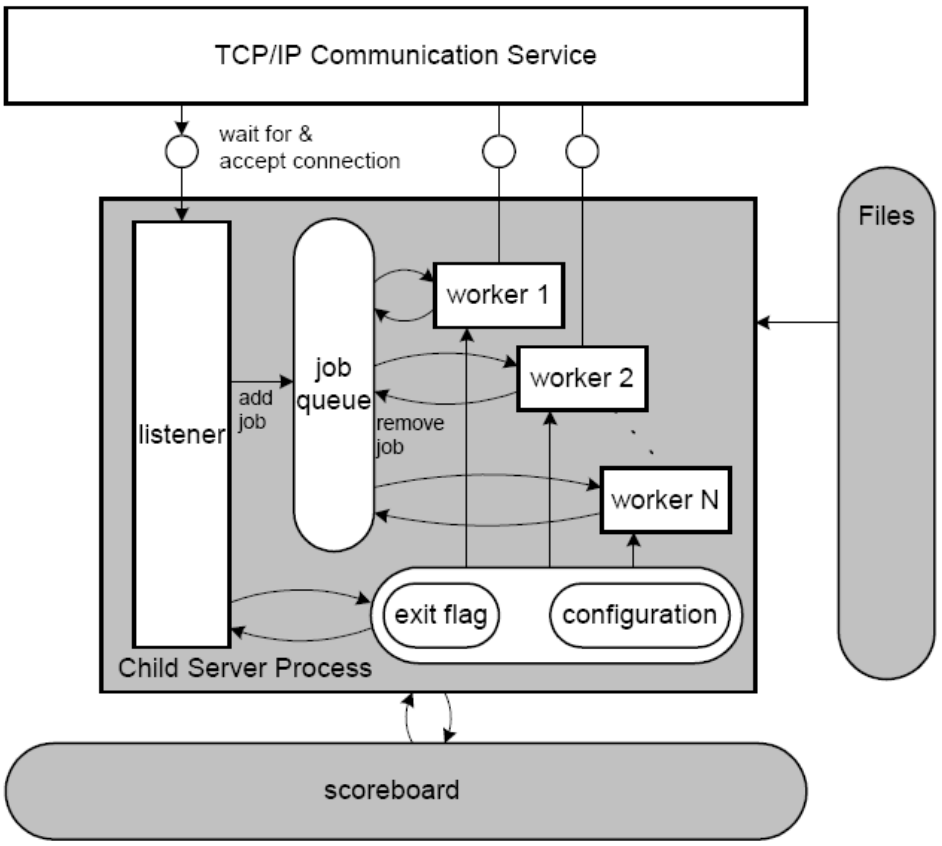


Figure 5. Apache web server system architecture [21]
Courtesy http://farm3.static.flickr.com/2699/4071355801_db6c8cd6c0_o.png

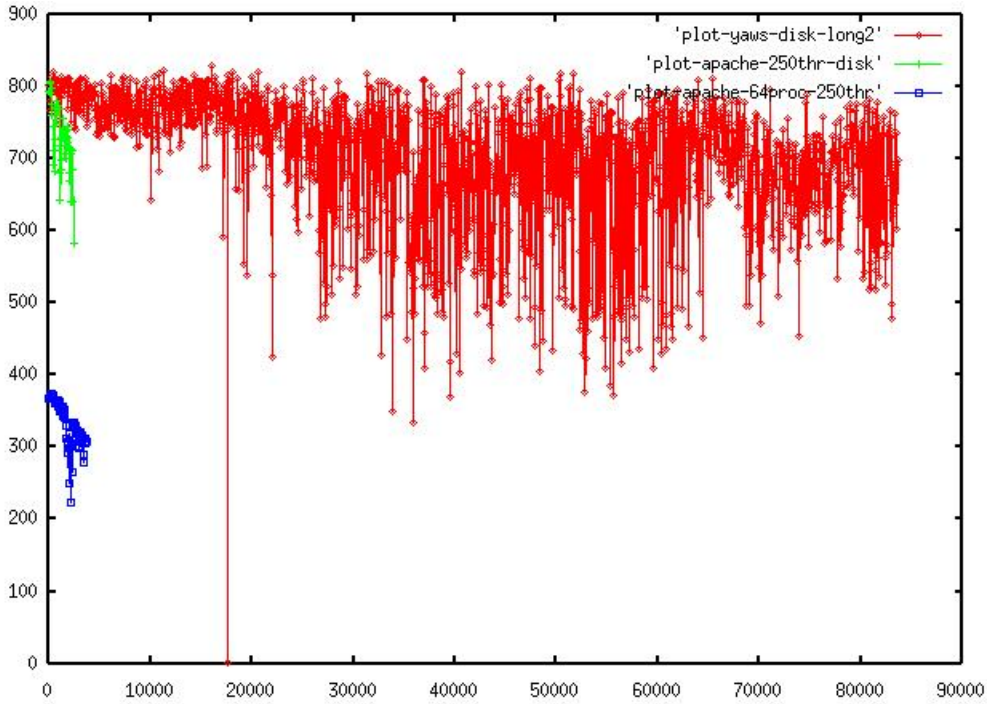


Figure 6. Apache vs. Yaws.
The horizonal axis shows the parallel requests,
the vertical one shows the throughput (KBytes/second).

The red curve is Yaws, running on NFS.
The blue one is Apache, running on NFS,
while the green one is also Apache but on a local file system.
Apache dies at about 4,000 parallel sessions,
while Yaws is still functioning at over 80,000 parallel connections. [22]
Courtesy http://farm3.static.flickr.com/2709/4072077210_3c3a507a8a_o.jpg

Reference,

[14] Improving running component of Twitter.
(http://qconlondon.com/london-2009/file?path=/qcon-london-
2009/slides/EvanWeaver_ImprovingRunningComponentsAtTwitter.pdf)
[16] Updating Twitter without service disruptions.
(http://gojko.net/2009/03/16/qcon-london-2009-upgrading-
twitter-without-service-disruptions/)
[17] Fixing Twitter. (http://assets.en.oreilly.com/1/event/29/Fixing_Twitter_
Improving_the_Performance_and_Scalability_of_the_World_s_Most_Popular_
Micro-blogging_Site_Presentation%20Presentation.pdf)
[21] Apache system architecture.
(http://www.fmc-modeling.org/download/publications/
groene_et_al_2002-architecture_recovery_of_apache.pdf)
[22] Apache vs Yaws. (http://www.sics.se/~joe/apachevsyaws.html)
[23] 质疑Apache和Yaws的性能比较. (http://www.javaeye.com/topic/107476)
[24] Yaws Web Server. (http://yaws.hyber.org/)
[25] Erlang Programming Language. (http://www.erlang.org/)

【5】数据流与控制流

通过让Apache进程空循环的办法，迅速接纳用户的访问，推迟服务，说白了是个缓兵之计，目的是让用户不至于收到“HTTP 503” 错误提示，“503错误” 是指 “服务不可用(Service Unavailable)”，也就是网站拒绝访问。

大禹治水，重在疏导。真正的抗洪能力，体现在蓄洪和泄洪两个方面。蓄洪容易理解，就是建水库，要么建一个超大的水库，要么造众多小水库。泄洪包括两个方面，1. 引流，2. 渠道。

对于Twitter系统来说，庞大的服务器集群，尤其是以MemCached为主的众多的缓存，体现了蓄洪的容量。引流的手段是Kestrel消息队列，用于传递控制指令。渠道是机器与机器之间的数据传输通道，尤其是通往MemCached的数据通道。渠道的优劣，在于是否通畅。

Twitter的设计，与大禹的做法，形相远，实相近。Twitter系统的抗洪措施，体现在有效地控制数据流，保证在洪峰到达时，能够及时把数据疏散到多个机器上去，从而避免压力过度集中，造成整个系统的瘫痪。

2009 年6月，Purewire公司通过爬Twitter网站，跟踪Twitter用户之间“追”与“被追”的关系，估算出Twitter用户总量在7,000,000左右 [26]。在这7百万用户中，不包括那些既不追别人，也不被别人追的孤立用户。也不包括孤岛人群，孤岛内的用户只相互追与被追，不与外界联系。如果加上这 些孤立用户和孤岛用户群，目前Twitter的用户总数，或许不会超过1千万。

截止2009年3月，中国移动用户数已达 4.7亿户[27]。如果中国移动的飞信[28] 和139说客[29] 也想往Twitter方向发展，那么飞信和139的抗洪能力应该设计到多少呢？简单讲，需要把Twitter系统的现有规模，至少放大47倍。所以，有人这样评论移动互联网产业，“在中国能做到的事情，在美国一定能做到。反之，不成立”。

但是无论如何，他山之石可以攻玉。这就是我们研究Twitter的系统架构，尤其是它的抗洪机制的目的。

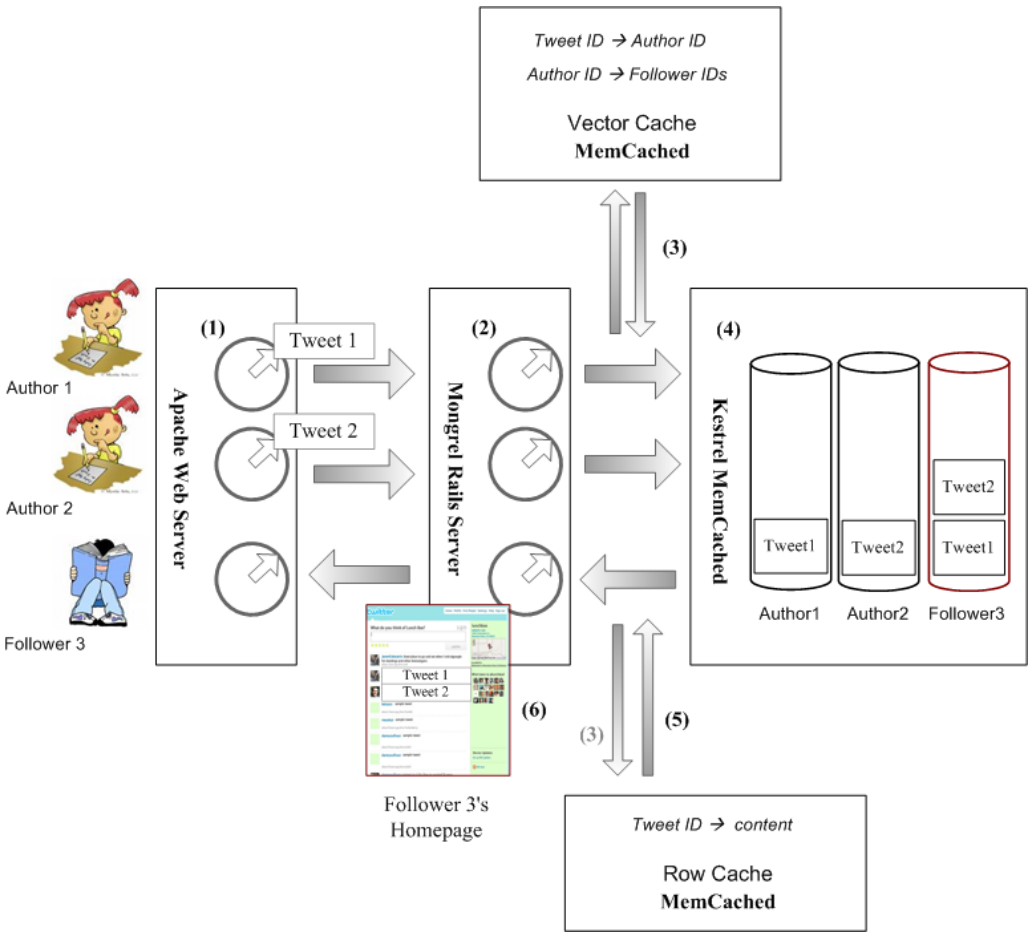


Figure 7. Twitter internal flows
Courtesy http://farm3.static.flickr.com/2766/4095392354_66bd4bcc30_o.png
下面举个简单的例子，剖析一下Twitter网站内部的流程，借此考察Twitter系统有哪些机制，去实现抗洪的三要素，“水库”，“引流”和“渠道”。
假设有两个作者，通过浏览器，在Twitter网站上发表短信。有一个读者，也通过浏览器，访问网站并阅读他们写的短信。

1. 作者的浏览器与网站建立连接，Apache Web Server分配一个进程(Worker Process)。作者登录，Twitter查找作者的ID，并作为Cookie，记忆在HTTP邮包的头属性里。

2. 浏览器上传作者新写的短信(Tweet)，Apache收到短信后，把短信连同作者ID，转发给Mongrel Rails Server。然后Apache进程进入空循环，等待Mongrel的回复，以便更新作者主页，把新写的短信添加上去。

3. Mongrel收到短信后，给短信分配一个ID，然后把短信ID与作者ID，缓存到Vector MemCached服务器上去。

同时，Mongrel让Vector MemCached查找，有哪些读者“追”这位作者。如果Vector MemCached没有缓存这些信息，Vector MemCached自动去MySQL数据库查找，得到结果后，缓存起来，以备日后所需。然后，把读者IDs回复给Mongrel。

接着，Mongrel把短信ID与短信正文，缓存到Row MemCached服务器上去。

4. Mongrel通知Kestrel消息队列服务器，为每个作者及读者开设一个队列，队列的名称中隐含用户ID。如果Kestrel服务器中已经存在这些队列，那就延用以往的队列。

对应于每个短信，Mongrel已经从Vector MemCached那里知道，有哪些读者追这条短信的作者。Mongrel把这条短信的ID，逐个放进每位读者的队列，以及作者本人的队列。

5. 同一台Mongrel Server，或者另一台Mongrel Server，在处理某个Kestrel队列中的消息前，从这个队列的名称中解析出相应的用户ID，这个用户，既可能是读者，也可能是作者。

然后Mongrel从Kestrel队列中，逐个提取消息，解析消息中包含的短信ID。并从Row MemCached缓存器中，查找对应于这个短信ID的短信正文。

这时，Mongrel既得到了用户的ID，也得到了短信正文。接下去Mongrel就着手更新用户的主页，添加上这条短信的正文。

6. Mongrel把更新后的作者的主页，传递给正在空循环的Apache的进程。该进程把作者主页主动传送(push)给作者的浏览器。

如果读者的浏览器事先已经登录Twitter网站，建立连接，那么Apache给该读者也分配了一个进程，该进程也处于空循环状态。Mongrel把更新后的读者的主页，传递给相应进程，该进程把读者主页主动传递给读者的浏览器。

咋一看，流程似乎不复杂。“水库”，“引流”和“渠道”，这抗洪三要素体现在哪里呢？盛名之下的Twitter，妙处何在？值得细究的看点很多。

Reference,

[26] Twitter user statistics by Purewire, June 2009.
(http://www.nickburcher.com/2009/06/twitter-user-statistics-purewire-report.html)
[27] 截止2009年3月，中国移动用户数已达4.7亿户.
(http://it.sohu.com/20090326/n263018002.shtml)
[28] 中国移动飞信网. (http://www.fetion.com.cn/)
[29] 中国移动139说客网. (http://www.139.com/)

【6】流量洪峰与云计算

上一篇历数了一则短信从发表到被阅读，Twitter业务逻辑所经历的6个步骤。表面上看似乎很乏味，但是细细咀嚼，把每个步骤展开来说，都有一段故事。

美国年度橄榄球决赛，绰号超级碗(Super Bowl)。Super Bowl在美国的收视率，相当于中国的央视春节晚会。2008年2月3日，星期天，该年度Super Bowl如期举行。纽约巨人队(Giants)，对阵波士顿爱国者队(Patriots)。这是两支实力相当的球队，决赛结果难以预料。比赛吸引了近一亿美国人观看电视实况转播。

对于Twitter来说，可以预料的是，比赛进行过程中，Twitter流量必然大涨。比赛越激烈，流量越高涨。Twitter无法预料的是，流量究竟会涨到多少，尤其是洪峰时段，流量会达到多少。

根据[31]的统计，在Super Bowl比赛进行中，每分钟的流量与当日平均流量相比，平均高出40%。在比赛最激烈时，更高达150%以上。与一周前，2008年1月27日，一个平静的星期天的同一时段相比，流量的波动从平均10%，上涨到40%，最高波动从35%，上涨到150%以上。

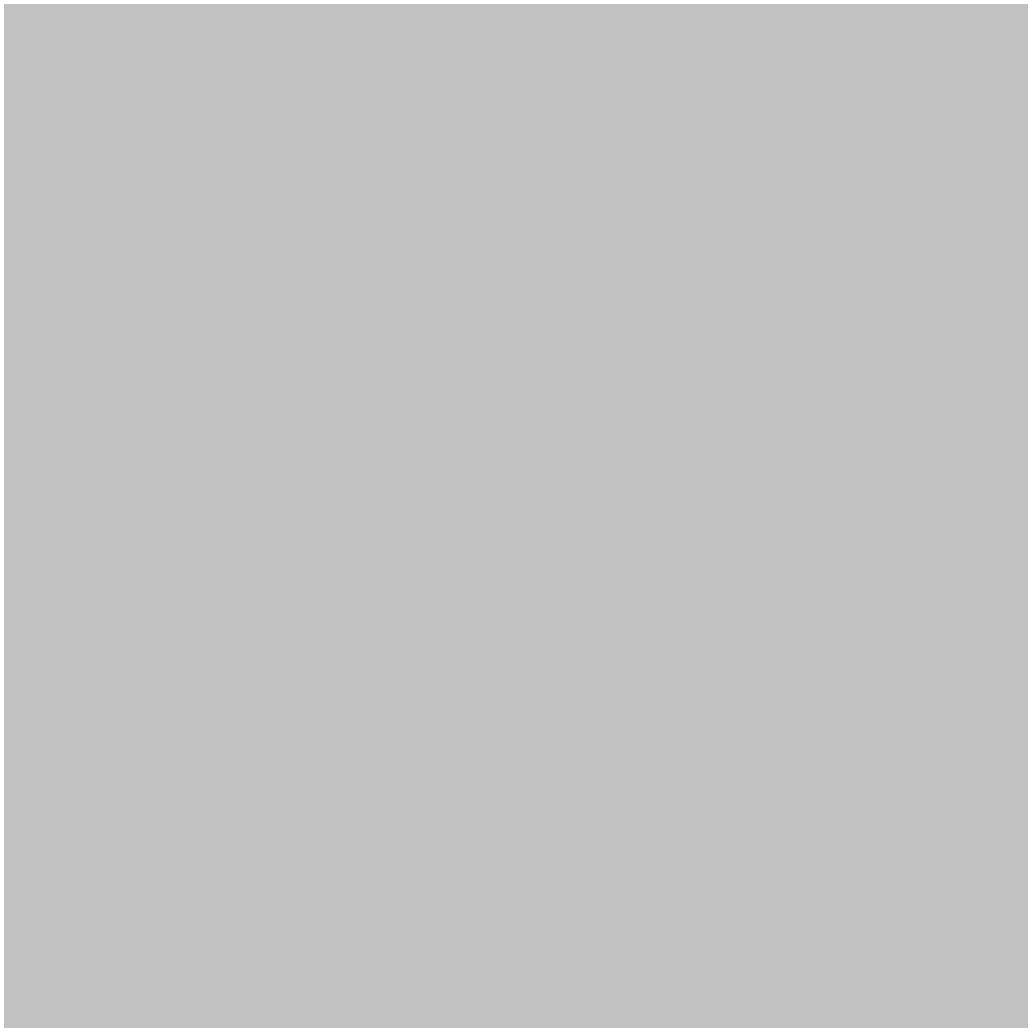


Figure 8. Twitter traffic during Super Bowl, Sunday, Feb 3, 2008 [31]. The blue line represents the percentage of updates per minute during the Super Bowl normalized to the average number of updates per minute during the rest of the day, with spikes annotated to show what people were twittering about. The green line represents the traffic of a "regular" Sunday, Jan 27, 2008.
Courtesy http://farm3.static.flickr.com/2770/4085122087_970072e518_o.png

由此可见，Twitter流量的波动十分可观。对于Twitter公司来说，如果预先购置足够的设备，以承受流量的变化，尤其是重大事件导致的洪峰流量，那么这些设备在大部分时间处于闲置状态，非常不经济。但是如果缺乏足够的设备，那么面对重大事件，Twitter系统有可能崩溃，造成的后果是用户流失。

怎么办？办法是变买为租。Twitter公司自己购置的设备，其规模以应付无重大事件时的流量压力为限。同时租赁云计算平台公司的设备，以应付重大事件来临时的洪峰流量。租赁云计算的好处是，计算资源实时分配，需求高的时候，自动分配更多计算资源。

Twitter公司在2008年以前，一直租赁Joyent公司的云计算平台。在2008年2月3日的Super Bowl即将来临之际，Joyent答应Twitter，在比赛期间免费提供额外的计算资源，以应付洪峰流量[32]。但是诡异的是，离大赛只剩下不到4天，Twitter公司突然于1月30日晚10时，停止使用Joyent的云计算平台，转而投奔Netcraft [33,34]。

Twitter弃Joyent，投Netcraft，其背后的原因是商务纠葛，还是担心Joyent的服务不可靠，至今仍然是个谜。

变买为租，应对洪峰，这是一个不错的思路。但是租来的计算资源怎么用，又是一个大问题。查看一下[35]，不难发现Twitter把租赁来的计算资源，大部分用于增加Apache Web Server，而Apache是Twitter整个系统的最前沿的环节。

为什么Twitter很少把租赁来的计算资源，分配给Mongrel Rails Server，MemCached Servers，Varnish HTTP

Accelerators等等其它环节？在回答这个问题以前，我们先复习一下前一章“数据流与控制流”的末尾，Twitter从写到读的6个步骤。

这6个步骤的前2步说到，每个访问Twitter网站的浏览器，都与网站保持长连接。目的是一旦有人发表新的短信，Twitter网站在500ms以内，把新短信push给他的读者。问题是在没有更新的时候，每个长连接占用一个Apache的进程，而这个进程处于空循环。所以，绝大多数Apache进程，在绝大多数时间里，处于空循环，因此占用了大量资源。

事实上，通过Apache Web Servers的流量，虽然只占Twitter总流量的10%-20%，但是Apache却占用了Twitter整个服务器集群的50%的资源[16]。所以，从旁观者角度来看，Twitter将来势必罢黜Apache。但是目前，当Twitter分配计算资源时，迫不得已，只能优先保证Apache的需求。

迫不得已只是一方面的原因，另一方面，也表明Twitter的工程师们，对其系统中的其它环节，太有信心了。

在第四章“抗洪需要隔离”中，我们曾经打过一个比方，“在晚餐高峰时段，餐馆常常客满。对于新来的顾客，餐馆服务员不是拒之门外，而是让这些顾客在休息厅等待”。对于Twitter系统来说，Apache充当的角色就是休息厅。只要休息厅足够大，就能暂时稳住用户，换句行话讲，就是不让用户收到HTTP-503的错误提示。

稳住用户以后，接下去的工作是高效率地提供服务。高效率的服务，体现在Twitter业务流程6个步骤中的后4步。为什么Twitter对这4步这么有信心？

Reference,

[16] Updating Twitter without service disruptions.
(http://gojko.net/2009/03/16/qcon-london-2009-upgrading-twitter-without-service-disruptions/)
[30] Giants and Patriots draws 97.5 million US audience to the Super Bowl.
(http://www.reuters.com/article/topNews/idUSN0420266320080204)
[31] Twitter traffic during Super Bowl 2008.
(http://blog.twitter.com/2008/02/highlights-from-superbowl-sunday.html)
[32] Joyent provides Twitter free extra capacity during the Super Bowl 2008.
(http://blog.twitter.com/2008/01/happy-happy-joyent.html)
[33] Twitter stopped using Joyent’s cloud at 10PM, Jan 30, 2008.
(http://www.joyent.com/joyeurblog/2008/01/31/twitter-and-joyent-update/)
[34] The hasty divorce for Twitter and Joyent.
(http://www.datacenterknowledge.com/archives/2008/01/31/hasty-divorce-for-twitter-joyent/)
[35] The usage of Netcraft by Twitter.
(http://toolbar.netcraft.com/site_report?url=http://twitter.com)

【7】作为一种进步的不彻底

不彻底的工作方式，对于架构设计是一种进步。

当一个来自浏览器的用户请求到达Twitter后台系统的时候，第一个迎接它的，是Apache Web Server。第二个出场的，是Mongrel Rails Server。Mongrel既负责处理上传的请求，也负责处理下载的请求。Mongrel处理上传和下载的业务逻辑非常简洁，但是简洁的表象之下，却蕴含着反常规的设计。这种反常规的设计，当然不是疏忽的结果，事实上，这正是Twitter架构中，最值得注意的亮点。

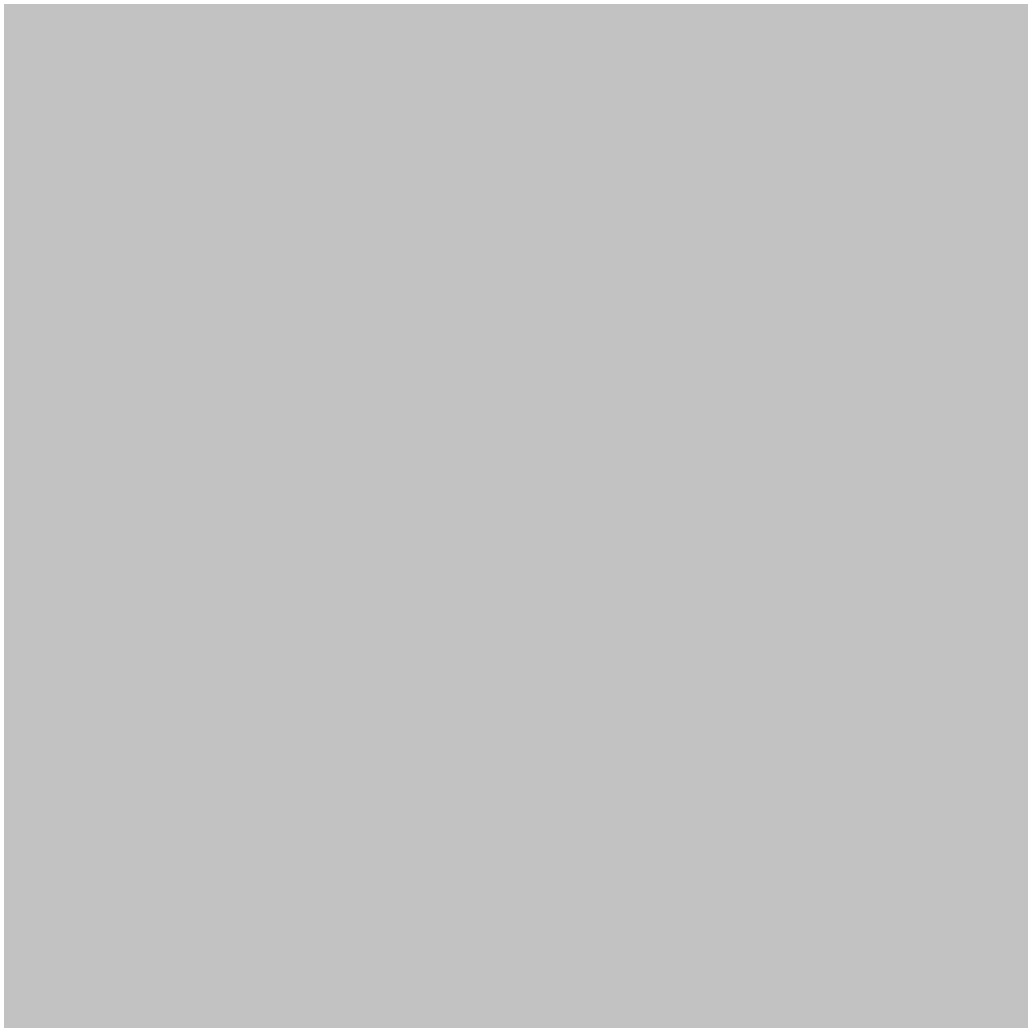


Figure 9. Twitter internal flows
Courtesy http://farm3.static.flickr.com/2766/4095392354_66bd4bcc30_o.png
所谓上传，是指用户写了一个新短信，上传给Twitter以便发表。而下载，是指Twitter更新读者的主页，添加最新发表的短信。Twitter下载的方式，不是读者主动发出请求的pull的方式，而是Twitter服务器主动把新内容push给读者的方式。先看上传，Mongrel处理上传的逻辑很简洁，分两步。

1. 当Mongrel收到新短信后，分配一个新的短信ID。然后把新短信的ID，连同作者ID，缓存进Vector MemCached服务器。接着，把短信ID以及正文，缓存进Row MemCached服务器。这两个缓存的内容，由Vector MemCached与Row MemCached在适当的时候，自动存放进MySQL数据库中去。

2. Mongrel在Kestrel消息队列服务器中，寻找每一个读者及作者的消息队列，如果没有，就创建新的队列。接着，Mongrel把新短信的ID，逐个放进“追”这位作者的所有在线读者的队列，以及作者本人的队列。

品味一下这两个步骤，感觉是Mongrel的工作不彻底。一，把短信及其相关IDs，缓存进Vector MemCached和Row Cached就万事大吉，而不直接负责把这些内容存入MySQL数据库。二，把短信ID扔进Kestrel消息队列，就宣告上传任务结束。Mongrel 没有用任何方式去通知作者，他的短信已经被上传。也不管读者是否能读到新发表的短信。

为什么Twitter采取了这种反常规的不彻底的工作方式？回答这个问题以前，不妨先看一看Mongrel处理下载的逻辑。把上传与下载两段逻辑联系起来，对比一下，有助于理解。Mongrel下载的逻辑也很简单，也分两步。

1. 分别从作者和读者的Kestrel消息队列中，获得新短信的ID。

2. 从Row MemCached缓存器那里获得短信正文。以及从Page MemCached那里获得读者以及作者的主页，更新这些主页，也就是添加上新的短信的正文。然后通过Apache，push给读者和作者。

对照Mongrel处理上传和下载的两段逻辑，不难发现每段逻辑都“不彻底”，合在一起才形成一个完整的流程。所谓不彻底的工作方式，反映了 Twitter架构设计的两个“分”的理念。一，把一个完整的业务流程，分割成几段相对独立的工作，每一个工作由同一台机器中不同的进程负责，甚至由不同的机器负责。二，把多个机器之间的协作，细化为数据与控制指令的传递，强调数据流与控制流的分离。

分割业务流程的做法，并不是Twitter的首创。事实上，三段论的架构，宗旨也是分割流程。Web Server负责HTTP的解析，Application Server负责业务逻辑，Database负责数据存储。遵从这一宗旨，Application Server的业务逻辑也可以进一步分割。

1996年，发明TCL语言的前伯克利大学教授John Ousterhout，在Usenix大会上做了一个主题演讲，题目是“为什么在多数情况下，多线程是一个糟糕的设计[36]”。2003年，同为伯克利大学教授的Eric Brewer及其学生们，发表了一篇题为“为什么对于高并发服务器来说，事件驱动是一个糟糕的设计[37]”。这两个伯克利大学的同事，同室操戈，他们在争论什么？

所谓多线程，简单讲就是由一根线程，从头到尾地负责一个完整的业务流程。打个比方，就像修车行的师傅每个人负责修理一辆车。而所谓事件驱动，指的是把一个完整的业务流程，分割成几个独立工作，每个工作由一个或者几个线程负责。打个比方，就像汽车制造厂里的流水线，有多个工位组成，每个工位由一位或者几位工人负责。

很显然，Twitter的做法，属于事件驱动一派。事件驱动的好处在于动态调用资源。当某一个工作的负担繁重，成为整个流程中的瓶颈的时候，事件驱动的架构可以很方便地调集更多资源，来化解压力。对于单个机器而言，多线程和事件驱动的两类设计，在性能方面的差异，并不是非常明显。但是对于分布式系统而言，事件驱动的优势发挥得更为淋漓尽致。

Twitter把业务流程做了两次分割。一，分离了Mongrel与MySQL数据库，Mongrel不直接插手MySQL数据库的操作，而是委托MemCached全权负责。二，分离了上传和下载两段逻辑，两段逻辑之间通过Kestrel队列来传递控制指令。

在John Ousterhout和Eric Brewer两位教授的争论中，并没有明确提出数据流与控制流分离的问题。所谓事件，既包括控制信号，也包括数据本身。考虑到通常数据的尺寸大，传输成本高，而控制信号的尺寸小，传输简便。把数据流与控制流分离，可以进一步提高系统效率。

在Twitter系统中，Kestrel消息队列专门用来传输控制信号，所谓控制信号，实际上就是IDs。而数据是短信正文，存放在Row MemCached中。谁去处理这则短信正文，由Kestrel去通知。

Twitter完成整个业务流程的平均时间是500ms，甚至能够提高到200-300ms，说明在Twitter分布式系统中，事件驱动的设计是成功。

Kestrel消息队列，是Twitter自行开发的。消息队列的开源实现很多，Twitter为什么不用现成的免费工具，而去费神自己研发呢？

Reference,

[36] Why threads are a bad idea (for most purposes), 1996.
(http://www.stanford.edu/class/cs240/readings/threads-bad-usenix96.pdf)
[37] Why events are a bad idea (for high-concurrency servers), 2003.
(http://www.cs.berkeley.edu/~brewer/papers/threads-hotos-2003.pdf)

【8】 得过不且过

北京西直门立交桥的设计，经常遭人诟病。客观上讲，对于一座立交桥而言，能够四通八达，就算得上基本完成任务了。大家诟病的原因，主要是因为行进路线太复杂。

当然，站在设计者角度讲，他们需要综合考虑来自各方面的制约。但是考虑到世界上立交桥比比皆是，各有各的难处，然而像西直门立交桥这样让人迷惑的，还真是少见。所以，对于西直门立交桥的设计者而言，困难是客观存在的，但是改进的空间总还是有的。

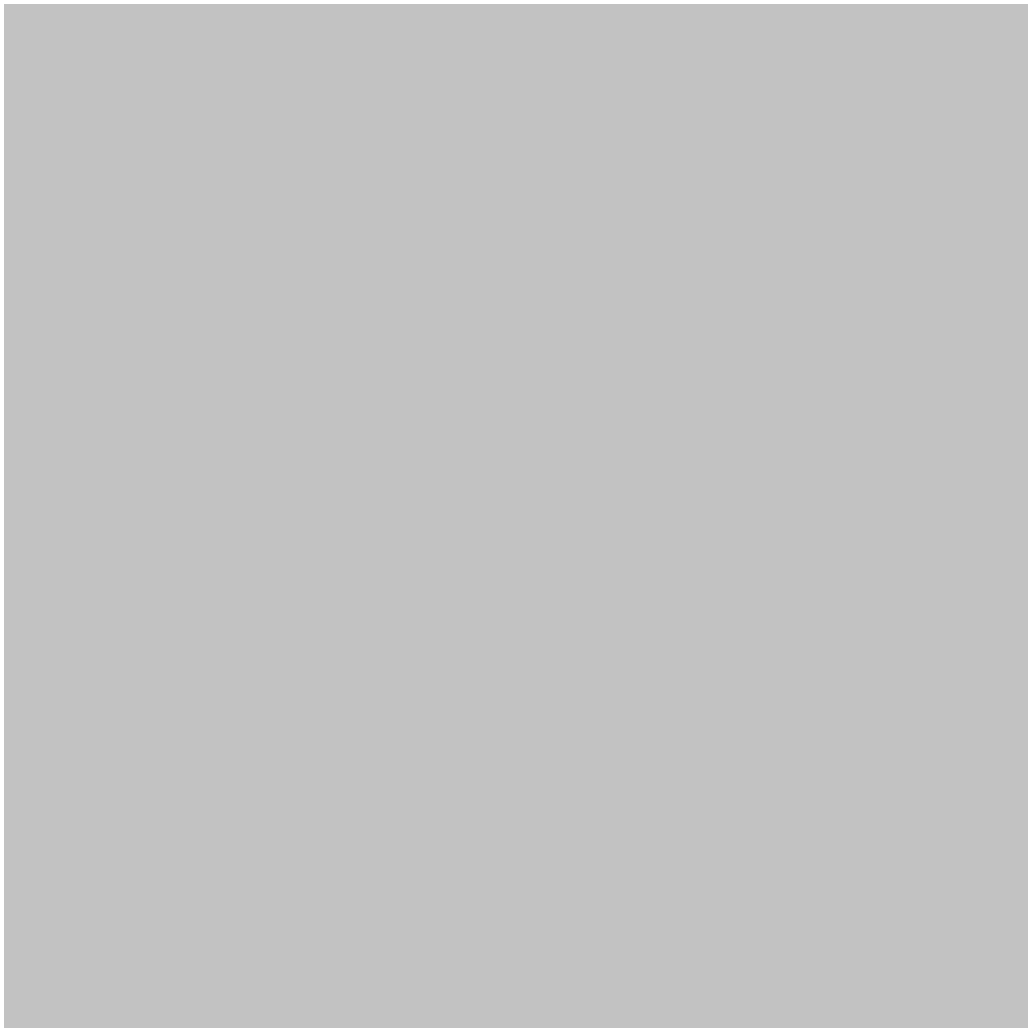


Figure 10. 北京西直门立交桥行进路线
Courtesy http://farm3.static.flickr.com/2671/4113112287_86cfb1cffd_o.png

大型网站的架构设计也一样，沿用传统的设计，省心又省力，但是代价是网站的性能。网站的性能不好，用户的体验也不好。Twitter这样的大型网站之所以能够一飞冲天，不仅功能的设计迎合了时代的需要，同时，技术上精益求精也是成功的必要保障。

例如，从Mongrel到MemCached之间，需要一个数据传输通道。或者严格地说，需要一个client library communicating to the memcached server。Twitter的工程师们，先用Ruby实现了一个通道。后来又用C实现了一个更快的通道。随后，不断地改进细节，不断地提升数据传输的效率。这一系列的改进，使Twitter的运行速度，从原先不设缓存时，每秒钟处理3.23个请求，到现在每秒处理139.03个请求，参见Figure 11。这个数据通道，现在定名为libmemcached，是开源项目 [38]。

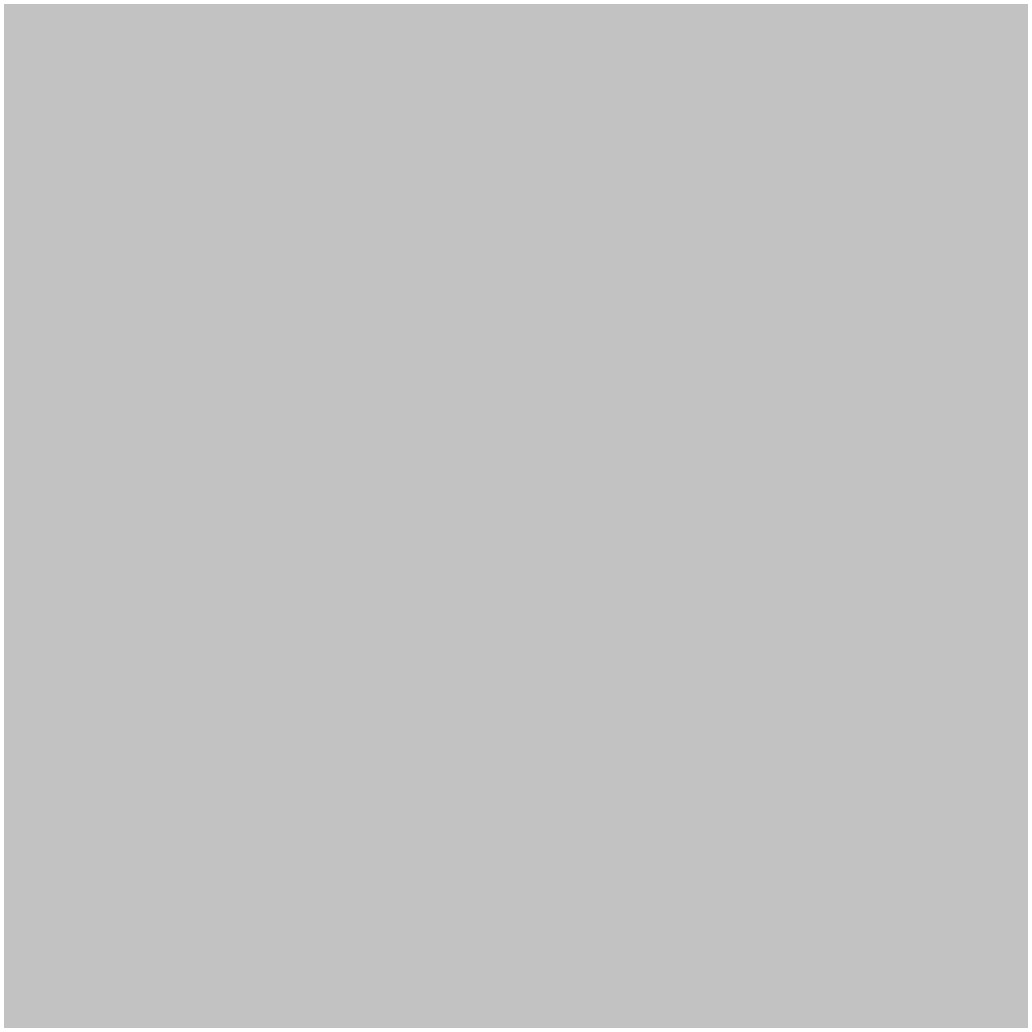


Figure 11. Evolving from a Ruby memcached client to a C client with optimised hashing. These changes increases Twitter performance from 3.23 requests per second without caching, to 139.03 requests per second nowadays [14]. Courtesy http://farm3.static.flickr.com/2767/4115077218_55c7250d43_o.png

又例如，Twitter系统中用消息队列来传递控制信号。这些控制信号，从插入队列，到被删除，生命周期很短。短暂的生命周期，意味着消息队列的垃圾回收(Garbage Collection)的效率，会严重影响整个系统的效率。因此，改进垃圾回收的机制，不断提高效率，成为不可避免的问题。Twitter使用的消息队列，原先不是Kestrel，而是用Ruby编写的一个简单的队列工具。但是如果继续沿用Ruby这种语言，性能优化的空间不大。Ruby的优点是集成了很多功能，从而大大减少了开发过程中编写程序的工作量。但是优点也同时是缺点，集成的功能太多，拖累也就多，牵一发而动全身，造成优化困难。

Twitter工程师戏言，”Ruby抗拒优化”，(“Ruby is optimization resistant”, by Evan Weaver [14])。几经尝试以后，Twitter的工程师们最终放弃了Ruby语言，改用Scala语言，自行实现了一个队列，命名为Kestrel [39]。

改换语言的主要动机是，Scala运行在JVM之上，因此优化Garbage Collection性能的手段丰富。Figure 12. 显示了使用Kestrel以后，垃圾回收的滞后，在平时只有2ms，最高不超过4ms。高峰时段，平均滞后5ms，最高不超过35ms。

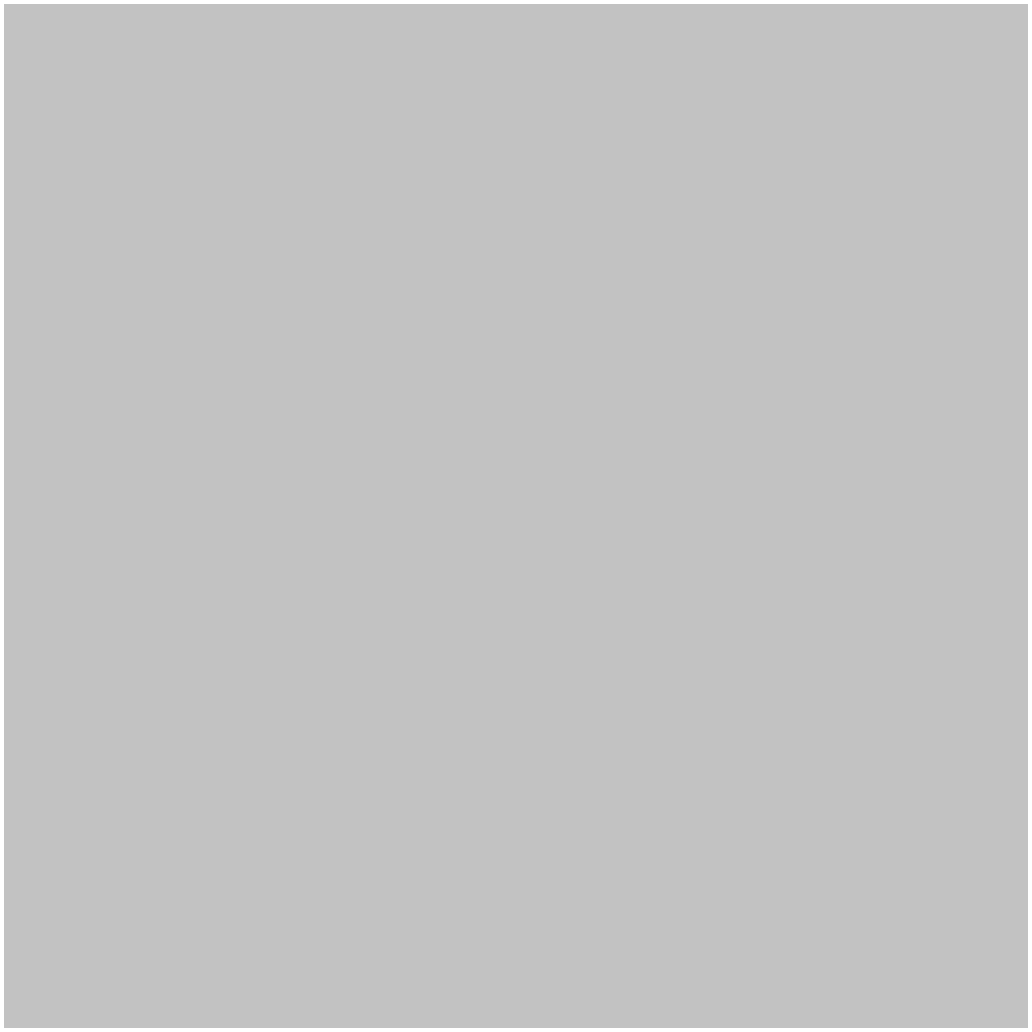


Figure 12. The latency of Twitter Kestrel garbage collection [14].
Courtesy http://farm3.static.flickr.com/2617/4115072726_c611955bb2_o.png

RubyOnRails逐渐淡出Twitter，看来这是大势所趋。最后一步，也是最高潮的一步，可能是替换Mongrel。事实上，Twitter所谓“API Server”，很可能是他们替换Mongrel的前奏。

Twitter的Evan Weaver说，“API Server”的运行效率，比Apache+Mongrel组合的速度快4倍。所谓Apache+Mongrel组合，是RubyOnRails的一种实现方式。Apache+Mongrel组合，每秒能够处理139个请求，参见Figure 11，而“API Server” 每秒钟能够处理大约550个请求 [16]。换句话说，使用Apache+Mongrel组合，优点是降低了工程师们写程序的负担，但是代价是系统性能降低了4倍，换句话说，用户平均等待的时间延长了4倍。

活着通常不难，活得精彩永远很难。得过不且过，这是一种精神。

Reference,
[14] Improving running component of Twitter.
(http://qconlondon.com/london-2009/file?path=/qcon-london-2009/slides/EvanWeaver_ImprovingRunningComponentsAtTwitter.pdf)
[16] Updating Twitter without service disruptions.
(http://gojko.net/2009/03/16/qcon-london-2009-upgrading-twitter-without-service-disruptions/)
[38] Open source project, libmemcached, by Twitter.
(http://tangent.org/552/libmemcached.html)
[39] Open source project, Kestrel Messaging Queue, by Twitter.
(http://github.com/robey/kestrel)

【9】结语

这个系列讨论了Twitter架构设计，尤其是cache的应用，数据流与控制流的组织等等独特之处。把它们与抗洪抢险中，蓄洪，引流，渠道三种手段相对比，便于加深理解。同时参考实际运行的结果，验证这样的设计是否能够应付实际运行中遇到的压力。

解剖一个现实网站的架构，有一些难度。主要体现在相关资料散落各处，而且各个资料的视点不同，覆盖面也不全。更严重的问题是，这些资料不是学术论文，质量良莠不齐，而且一些文章或多或少地存在缺失，甚至错误。

单纯把这些资料罗列在一起，并不能满足全景式的解剖的需要。整理这些资料的过程，很像是侦探办案。福尔摩斯探案的方法，是证据加推理。

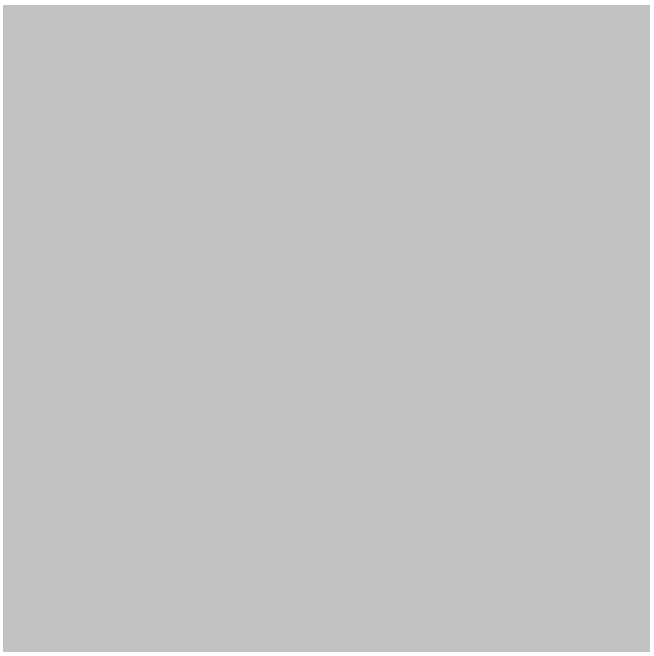
1. 如果观察到证据O1，而造成O1出现的原因，有可能是R1，也有可能是R2或者R3。究竟哪一个原因，才是真正的原因，需要进一步收集更多的证据，例如O2，O3。如果造成O2 出现的可能的原因是R2和R4，造成O3 出现的可能原因是R3和R5。把所有证据O1 O2 O3，综合起来考虑，可能性最大的原因必然是(R1,R2,R3), (R2,R4), (R3,R5) 的交集，也就是R2。这是反绎推理的过程。

2. 如果反绎推理仍然不能确定什么是最可能的原因，那么假定R2是真实的原因，采用演绎推理，R2必然导致O4证据的出现。接下去要做的事情是，确认O4是否真的出现，或者寻找O4肯定不会出现的证据。以此循环。

解剖网络架构的方法，与探案很相似。只读一篇资料是不够的，需要多多收集资料，交叉印证。不仅交叉印证，而且引申印证，如果某一环节A是这样设计的，那么关联环节B必然相应地那样设计。如果一时难以确定A到底是如何设计的，不妨先确定B是如何设计的。反推回来，就知道A应该如何设计了。

解剖网站架构，不仅有益，而且有趣。

Figure 13. Sherlock Holmes，福尔摩斯探案
Courtesy http://www.fantasticfiction.co.uk/images/c0/c2053.jpg



【全文完】

本文来自：http://www.tektalk.org

原文链接：

1. 解剖Twitter 【1】万事开头易
2. 解剖Twitter 【2】三段论
3. 解剖Twitter 【3】Cache == Cash
4. 解剖Twitter 【4】抗洪需要隔离
5. 解剖Twitter 【5】数据流与控制流
6. 解剖Twitter【6】流量洪峰与云计算
7. 解剖Twitter 【7】 作为一种进步的不彻底
8. 解剖Twitter 【8】 得过不且过
9. 解剖Twitter 【9】结语

相关文章

- Twitter系统运维经验
- Web应用中的轻量级消息队列
- 说说大型高并发高负载网站的系统架构

- 软件架构设计中的同步与异步问题(修改版)
- 杨卫华：新浪微博的架构发展历程