http://www.infoq.com/cn/articles/ctrip-big-data-high-concurrency-applications-architecture

面对百亿用户数据，日均亿次请求，携程应用架构如何涅槃？



[**喜欢**](http://www.infoq.com/cn/articles/ctrip-big-data-high-concurrency-applications-architecture) | 作者 [**董锐**](http://www.infoq.com/cn/profile/%E8%91%A3%E9%94%90) 发布于 2016年10月8日. 估计阅读时间: 不到一分钟 | 道AI风控、Serverless架构、EB级存储引擎，尽在[**ArchSummit**](http://sz2017.archsummit.com/?utm_source=infoq&utm_medium=notice)![讨论](http://www.infoq.com/cn/articles/ctrip-big-data-high-concurrency-applications-architecture#theCommentsSection)

* [分享到：微博微信FacebookTwitter有道云笔记邮件分享](http://www.infoq.com/cn/articles/ctrip-big-data-high-concurrency-applications-architecture)
* [**稍后阅读**](http://www.infoq.com/cn/articles/ctrip-big-data-high-concurrency-applications-architecture)
* [**我的阅读清单**](http://www.infoq.com/cn/showbookmarks.action)

互联网二次革命的移动互联网时代，如何吸引用户、留住用户并深入挖掘用户价值，在激烈的竞争中脱颖而出，是各大电商的重要课题。通过各类大数据对用户进行研究，以数据驱动产品是解决这个课题的主要手段，携程的大数据团队也由此应运而生；经过几年的努力，大数据的相关技术为业务带来了惊人的提升与帮助。

以基础大数据的用户意图服务为例，通过将广告和栏位的“千人一面”变为“千人千面”，在提升用户便捷性，可用性，降低费力度的同时，其转化率也得到了数倍的提升，体现了大数据服务的真正价值。

在新形势下，传统应用架构不得不变为大数据及新的高并发架构，来应对业务需求激增及高速迭代的需要。

一、业务高速发展带来的应用架构挑战

**相关厂商内容**

[**微信Android模块化架构重构实践**](http://www.infoq.com/infoq/url.action?i=15945&t=f)

[**蘑菇街分布式消息中间件Corgi的架构演进**](http://www.infoq.com/infoq/url.action?i=15946&t=f)

[**Serverless架构：一条SQL到一个服务有多远？**](http://www.infoq.com/infoq/url.action?i=15947&t=f)

[**对抗复杂性，架构设计中可借鉴复用这些手段**](http://www.infoq.com/infoq/url.action?i=15948&t=f)

[**阿里：风控场景的模型平台架构设计**](http://www.infoq.com/infoq/url.action?i=15949&t=f)

**相关赞助商**

[http://cdn.infoqstatic.com/statics_s2_20170516-0317u5/resource/sponsorship/featuredcategory/6821/ASBOXtupian.jpg](http://www.infoq.com/infoq/url.action?i=15951&t=f)

**ArchSummit深圳2017，7月7-8日，深圳·华侨城洲际酒店，**[精彩内容抢先看](http://www.infoq.com/infoq/url.action?i=15950&t=f)

公司业务高速发展带来哪些主要的变化，以及给我们的系统带来了哪些挑战？

1. **业务需求的急速增长**，访问请求的并发量激增，2016年1月份以来，业务部门的服务日均请求量**激增了5.5倍**。
2. **业务逻辑日益复杂化**，基础业务研发部需要支撑起OTA数十个业务线，业务逻辑日趋复杂和繁多。
3. **业务数据源多样化，异构化**，接入的业务线、合作公司的数据源越来越多；接入的数据结构由以前的数据库结构化数据整合转为Hive表、评论文本数据、日志数据、天气数据、网页数据等多元化异构数据整合。
4. **业务的高速发展和迭代**，部门一直以追求以最少的开发人力，以架构和系统的技术优化，支撑起携程各业务线高速发展和迭代的需要。

在这种新形势下，传统应用架构不得不变，做为工程师也必然要自我涅槃，改为大数据及新的高并发架构，来应对业务需求激增及高速迭代的需要。计算分层分解、去SQL、去数据库化、模块化拆解的相关技改工作已经刻不容缓。

以用户意图（AI 点金杖）的个性化服务为例， 面对BU业务线的全面支持的迫切需要，其应用架构必须解决如下**技术难点**：

1. **高访问并发**：每天近亿次的访问请求；
2. **数据量大**：每天TB级的增量数据，近百亿条的用户数据，上百万的产品数据；
3. **业务逻辑复杂**：复杂个性化算法和LBS算法；例如：满足一个复杂用户请求需要大量计算和30次左右的SQL数据查询，服务延时越来越长；
4. **高速迭代上线**：面对OTA多业务线的个性化、Cross-saling、Up-saling、需满足提升转化率的迫切需求，迭代栏位或场景要快速，同时减少研发成本。

二、应对挑战的架构涅磐

面对这些挑战，我们的应用系统架构应该如何涅磐？主要分如下**三大方面**系统详解:

**存储的涅磐**，这一点对于整个系统的吞吐量和并发量的提升起到最关键的作用，需要结合数据存储模型和具体应用的场景。

**计算的涅磐**，可以从横向和纵向考虑：横向主要是增加并发度，首先想到的是分布式。纵向拆分就是要求我们找到计算的结合点从而进行分层，针对不同的层次选择不同的计算地点。然后再将各层次计算完后的结果相结合，尽可能最大化系统整体的处理能力。

**业务层架构的涅磐**，要求系统的良好的模块化设计，清楚的定义模块的边界，模块自升级和可配置化。

三、应用系统的整体架构

认识到需要应对的挑战，我们应该如何设计我们的系统呢，下面将全面的介绍下我们的应用系统整体架构。

下图就是我们应用系统整体架构以及系统层次的模块构成。



**数据源部分**， Hermes是携程框架部门提供的消息队列，基于Kafka和Mysql做为底层实现的封装，应用于系统间实时数据传输交互通道。Hive和 HDFS是携程海量数据的主要存储，两者来自Hadoop 生态体系。Hadoop 这块大家已经很熟悉， 如果不熟悉的同学只要知道Hadoop 主要用于大数据量存储和并行计算批处理工作。

Hive 是基于Hadoop平台的数据仓库，沿用了关系型数据库的很多概念。比如说数据库和表，还有一套近似于SQL的查询接口的支持，在Hive里 叫做HQL，但是其底层的实现细节和关系型数据库完全不一样，Hive底层所有的计算都是基于MR来完成，我们的数据工程师90%都数据处理工作都基于它来完成。

**离线部分**，包含的模块有MR, Hive , Mahout, SparkQL/MLLib。Hive 上面已经介绍过，Mahout 简单理解提供基于Hadoop平台进行数据挖掘的一些机器学习的算法包。Spark类似hadoop也是提供大数据并行批量处理平台，但是它是基于内存的。SparkQL 和Spark MLLib是基于Spark平台的SQL查询引擎和数据挖掘相关算法框架。我们主要用Mahout和Spark MLLib 进行数据挖掘工作。

**调度系统zeus**，是淘宝开源大数据平台调度系统，于2015年引进到携程，之后我们进行了重构和功能升级，做为携程大数据平台的作业调度平台。

**近线部分**，是基于Muise来实现我们的近实时的计算场景，Muise是也是携程OPS提供的实时计算流处理平台，内部是基于Storm实现与HERMES消息队列搭配起来使用。例如，我们使用MUSIE通过消费来自消息队列里的用户实时行为，订单记录，结合画像等一起基础数据，经一系列复杂的规则和算法，实时的识别出用户的行程意图。

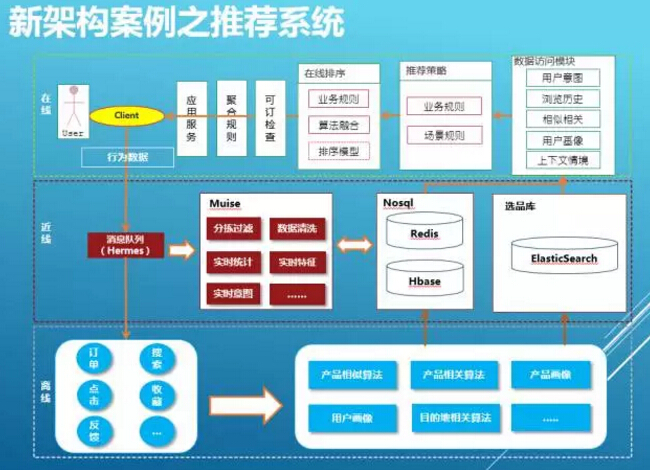
**后台/线上应用部分**，Mysql用于支撑后台系统的数据库。ElasticSearch 是基于Lucene实现的分布式搜索引擎，用于索引用户画像的数据，支持离线精准营销的用户筛选，同时支持线上应用推荐系统的选品功能 。Hbase 基于Hadoop的Hdfs 上的列存储Nosql数据库，用于后台报表可视化系统和线上服务的数据存储。

这里说明一下, 在线和后台应用使用的ElasticSearch和Hbase集群是分开的，互不影响。 Redis 支持在线服务的高速缓存，用于缓存统计分析出来的热点数据。

四、推荐系统案例

介绍完我们应用系统的整体构成， 接下来分享基于这套系统架构实现的一个实例——携程个性化推荐系统。

推荐系统的架构图：



1存储的涅磐

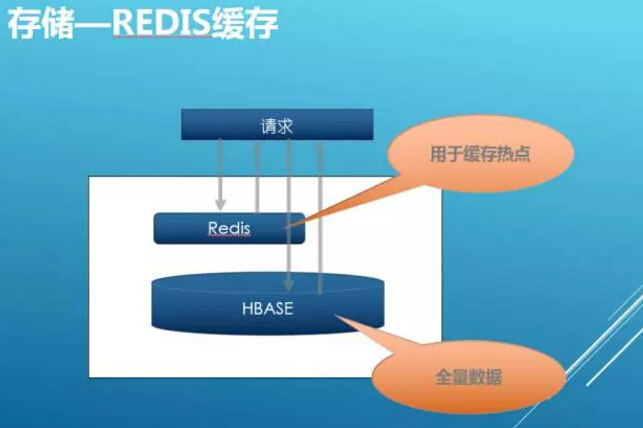
**1）Nosql (Hbase+Redis)**



我们之前存储使用的是Mysql,  一般关系型数据库会做为应用系统存储的首选。大家知道Mysql非商业版对分布式支持不够，在存储数据量不高，查询量和计算复杂度不是很大的情况下，可以满足应用系统绝大部分的功能需求。

我们现状是需要安全存储海量的数据，高吞吐，并发能力强，同时随着数据量和请求量的快速增加，能够通过加节点来扩容。另外还需要支持故障转移，自动恢复，无需额外的运维成本。综上几个主要因素，我们进行了大量的调研和测试，最终我们选用Hbase和Redis 两个Nosql数据库来取代以往使用的Mysql,。我们把用户意图以及推荐产品数据以KV的形式存储在Hbase中，我对操作Hbase进行一些优化，其中包括rowkey的设计，预分配，数据压缩等， 同时针对我们的使用场景对Hbase本身配置方面的也进行了调优。目前存储的数据量已经达到TB级别，支持每天千万次请求，同时保证99%在50毫秒内返回 。

Redis这块和多数应用系统使用方式一样，主要用于缓存热点数据，这里就不多说了。



**2）搜索引擎 (ElasticSearch)**



ES索引各业务线产品特征数据，提供基于用户的意图特征和产品特征复杂的多维检索和排序功能，当前集群由4台大内存物理机器构成，采用全内存索引。对比某一个复杂的查询场景， 之前用Mysql将近需要30次查询，使用ES只需要一次组合查询且在100毫秒内返回 。目前每天千万次搜索，99%以上在300毫秒以内返回。



2计算的涅磐

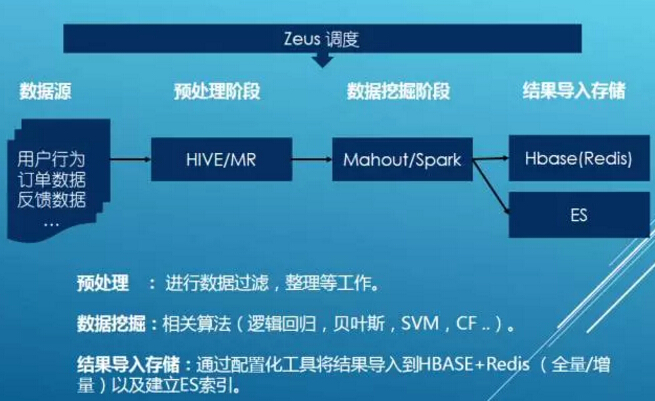
**1）数据源**，我们的数据源分结构化和半结构化数据以及非结构化数据。

结构化数据主要是指携程各产线的产品维表和订单数据，有酒店，景酒，团队游，门票，景点等；还有一些基础数据，比如城市表，车站等，这类数据基本上都是T+1。每天会有流程去各BU的生产表拉取数据。

半结构化数据是指，携程用户的访问行为数据，例如浏览，搜索，预订，反馈等，这边顺便提一下，这些数据这些是由前端采集框架实时采集，然后下发到后端的收集服务，由收集服务在写入到Hermes消息队列，一路会落地到Hadoop上面做长期存储，另一路近线层可以通过订阅Hermes此类数据Topic 进行近实时的计算工作。

我们还用到外部合作渠道的数据，还有一些评论数据，评论属于非结构化的，也是T+1更新。

**2）离线计算**，主要分三个处理阶段 。



**预处理阶段**，这块主要为后续数据挖掘做一些数据的准备工作，数据去重，过滤，对缺失信息的补足。举例来说采集下来的用户行为数据，所含有的产品信息很少，我们会使用产品表的数据进行一些补足，确保给后续的数据挖掘使用时候尽量完整的。

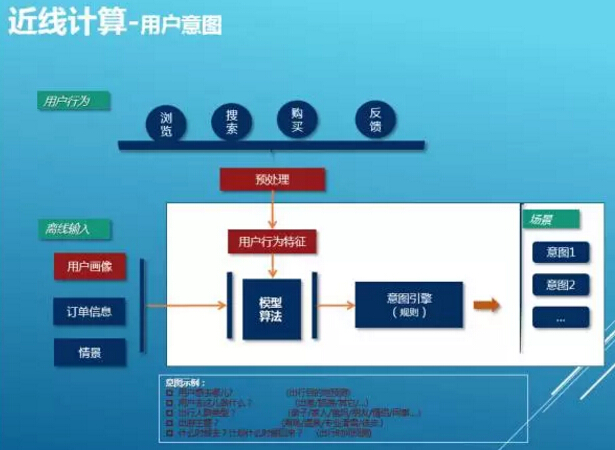
**数据挖掘阶段**，主要运用一些常用的数据挖掘算法进行模型训练和推荐数据的输出(分类，聚类，回归 ，CF等)。

**结果导入阶段**，我们通过可配置的数据导入工具将推荐数据，进行一系列转换后，导入到HBASE，Redis以及建立ES索引， Redis存储的是经统计计算出的热点数据 。

**3）近线计算（用户意图, 产品缓存）**

当用户没有明确的目的性情况下，很难找到满足兴趣的产品，我们不仅需要了解用户的历史兴趣，用户实时行为特征的抽取和理解更加重要，以便快速的推荐出符合用户当前兴趣的产品，这就是用户意图服务需要实现的功能。

一般来说用户特征分成两大类：一种是稳定的特征（用户画像），如用户性别，常住地，主题偏好等特征；另一类是根据用户行为计算获取的特征，如用户对酒店星级的偏好，目的地偏好，跟团游/自由行偏好等。基于前面所述的计算的特点，我们使用近在线计算来获取第二类用户特征，整体框图如下。从图中可以看出它的输入数据源包括两大类：第一类是实时的用户行为，第二类是用户画像，历史交易以及情景等离线模块提供的数据。结合这两类数据，经一些列复杂的近线学习算法和规则引擎，计算得出用户当前实时意图列表存储到Hbase和Redis中。

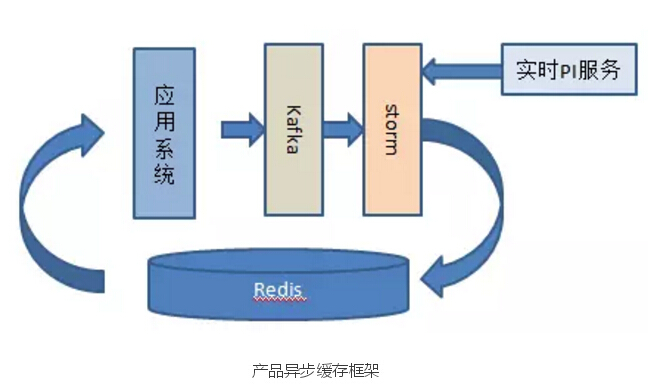


携程用户意图框架

近线另一个工作是产品数据缓存，携程的业务线很多，而我们的推荐系统会推各个业务线的产品，因此我们需要调用所有业务线的产品服务接口，但随着我们上线的场景的增加，这样无形的增加了对业务方接口的调用压力。而且业务线产品接口服务主要应用于业务的主流程或关键型应用，比较重，且SLA服务等级层次不齐，可能会影响到整个推荐系统的响应时间。

为了解决这两个问题，我们设计了近在线计算来进行业务的产品信息异步缓存策略，具体的流程如下。

我们会将待推荐的产品Id全部通过Kafka异步下发，在Storm中我们会对各业务方的产品首先进行聚合，达到批处理个数或者时间gap时，再调用各业务方的接口，这样减少对业务方接口的压力。通过调用业务方接口更新的产品状态临时缓存起来（根据各业务产品信息更新周期分别设置缓存失效时间），在线计算的时候直接先读取临时缓存数据，缓存不存在的情况下，再击穿到业务的接口服务。



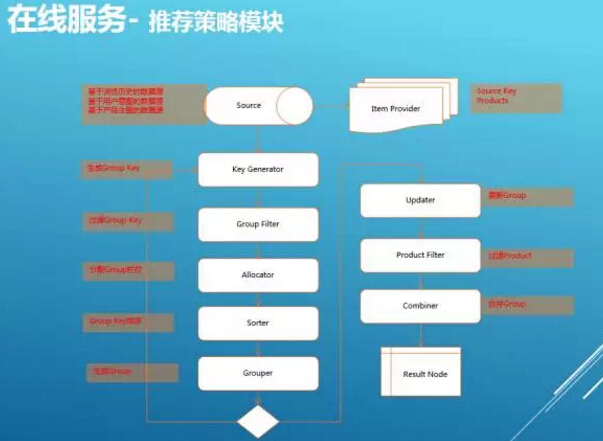
**4）在线计算（2个关键业务层架构模块介绍）**

**1，业务层架构-数据治理和访问模块**，支持的存储介质，目前支持的存储介质有Localcache，Redis，Hbase，Mysql 可以支持横向扩展 。统一配置，对同一份数据，采用统一配置，可以随意存储在任意介质，根据id查询返回统一格式的数据，对查询接口完全透明。

穿透策略和容灾策略，Redis只存储了热数据，当需要查询冷数据则可以自动到下一级存储如Hbase查询，避免缓存资源浪费。当Redis出现故障时或请求数异常上涨，超过整体承受能力，此时服务降级自动生效，并可配置化。



**2，业务层架构-推荐策略模块**，整个流程是先将用户意图、用户浏览，相关推荐策略生成的产品集合等做为数据输入，接着按照场景规则，业务逻辑重新过滤，聚合、排序。最后验证和拼装业务线产品信息后输出推荐结果；



我们对此流程每一步进行了一些模块化的抽象，将重排序逻辑按步骤抽象解耦，抽象如右图所示的多个组件，开发新接口时仅需要将内部DSL拼装便可以得到满足业务需求的推荐服务；提高了代码的复用率和可读性，减少了超过50%的开发时间；对于充分验证的模块的复用，有效保证了服务的质量。

作者介绍

**董锐**，近9年的互联网从业经验。2013年1月加入携程，曾在商业智能部设计和开发基于HADOOP生态体系的大数据数据仓库。现任基础业务研发部-数据智能应用组研发Leader，专注于携程个性化推荐系统，ABTEST等系统研发工作。

感谢[杜小芳](http://www.infoq.com/cn/author/%E6%9D%9C%E5%B0%8F%E8%8A%B3)对本文的审校。

http://www.infoq.com/cn/articles/flash-deal-architecture-optimization

秒杀业务架构优化之路



[**喜欢**](http://www.infoq.com/cn/articles/flash-deal-architecture-optimization) | 作者 [**沈剑**](http://www.infoq.com/cn/profile/%E6%B2%88%E5%89%91) 发布于 2016年3月28日. 估计阅读时间: 不到一分钟 | 道AI风控、Serverless架构、EB级存储引擎，尽在[**ArchSummit**](http://sz2017.archsummit.com/?utm_source=infoq&utm_medium=notice)![6**讨论**](http://www.infoq.com/cn/articles/flash-deal-architecture-optimization#theCommentsSection)

* [分享到：微博微信FacebookTwitter有道云笔记邮件分享](http://www.infoq.com/cn/articles/flash-deal-architecture-optimization)
* [**稍后阅读**](http://www.infoq.com/cn/articles/flash-deal-architecture-optimization)
* [**我的阅读清单**](http://www.infoq.com/cn/showbookmarks.action)

本文整理自“ArchSummit微课堂”线上分享——沈剑解析秒杀业务架构优化的思路。

一、秒杀业务为什么难做

IM系统，例如QQ或者微博，每个人都读自己的数据（好友列表、群列表、个人信息）。

微博系统，每个人读你关注的人的数据，一个人读多个人的数据。

秒杀系统，库存只有一份，所有人会在集中的时间读和写这些数据，多个人读一个数据。

例如小米手机每周二的秒杀，可能手机只有1万部，但瞬时进入的流量可能是几百几千万。又例如12306抢票，票是有限的，库存一份，瞬时流量非常多，都读相同的库存。读写冲突，锁非常严重，这是秒杀业务难的地方。那我们怎么优化秒杀业务的架构呢？

二、优化方向

优化方向有两个：

1. 将请求尽量拦截在系统上游（不要让锁冲突落到数据库上去）。传统秒杀系统之所以挂，请求都压倒了后端数据层，数据读写锁冲突严重，并发高响应慢，几乎所有请求都超时，流量虽大，下单成功的有效流量甚小。以12306为例，一趟火车其实只有2000张票，200w个人来买，基本没有人能买成功，请求有效率为0。
2. 充分利用缓存，秒杀买票，这是一个典型的读多些少的应用场景，大部分请求是车次查询，票查询，下单和支付才是写请求。一趟火车其实只有2000张票，200w个人来买，最多2000个人下单成功，其他人都是查询库存，写比例只有0.1%，读比例占99.9%，非常适合使用缓存来优化。好，后续讲讲怎么个“将请求尽量拦截在系统上游”法，以及怎么个“缓存”法，讲讲细节。

三、常见秒杀架构

常见的站点架构基本是这样的（特别是流量上亿的站点架构）:



1. 浏览器端，最上层，会执行到一些JS代码
2. 站点层，这一层会访问后端数据，拼HTML页面返回给浏览器
3. 服务层，向上游屏蔽底层数据细节，提供数据访问
4. 数据层，最终的库存是存在这里的，MySQL是一个典型（当然还有会缓存）

这个图虽然简单，但能形象的说明大流量高并发的秒杀业务架构，大家要记得这一张图。

后面细细解析各个层级怎么优化。

四、各层次优化细节

第一层，客户端怎么优化（浏览器层，APP层）

问大家一个问题，大家都玩过微信的摇一摇抢红包对吧，每次摇一摇，就会往后端发送请求么？回顾我们下单抢票的场景，点击了“查询”按钮之后，系统那个卡呀，进度条涨的慢呀，作为用户，我会不自觉的再去点击“查询”，对么？继续点，继续点，点点点……有用么？平白无故的增加了系统负载，一个用户点5次，80%的请求是这么多出来的，怎么整？

* 产品层面，用户点击“查询”或者“购票”后，按钮置灰，禁止用户重复提交请求；
* JS层面，限制用户在x秒之内只能提交一次请求；

APP层面，可以做类似的事情，虽然你疯狂的在摇微信，其实x秒才向后端发起一次请求。这就是所谓的“将请求尽量拦截在系统上游”，越上游越好，浏览器层，APP层就给拦住，这样就能挡住80%+的请求，这种办法只能拦住普通用户（但99%的用户是普通用户）对于群内的高端程序员是拦不住的。

FireBug一抓包，HTTP长啥样都知道，JS是万万拦不住程序员写for循环，调用HTTP接口的，这部分请求怎么处理？

第二层，站点层面的请求拦截

怎么拦截？怎么防止程序员写for循环调用，有去重依据么？IP？cookie-id？…想复杂了，这类业务都需要登录，用uid即可。在站点层面，对uid进行请求计数和去重，甚至不需要统一存储计数，直接站点层内存存储（这样计数会不准，但最简单）。一个uid，5秒只准透过1个请求，这样又能拦住99%的for循环请求。

5s只透过一个请求，其余的请求怎么办？缓存，页面缓存，同一个uid，限制访问频度，做页面缓存，x秒内到达站点层的请求，均返回同一页面。同一个item的查询，例如车次，做页面缓存，x秒内到达站点层的请求，均返回同一页面。如此限流，既能保证用户有良好的用户体验（没有返回404）又能保证系统的健壮性（利用页面缓存，把请求拦截在站点层了）。

页面缓存不一定要保证所有站点返回一致的页面，直接放在每个站点的内存也是可以的。优点是简单，坏处是HTTP请求落到不同的站点，返回的车票数据可能不一样，这是站点层的请求拦截与缓存优化。

好，这个方式拦住了写for循环发HTTP请求的程序员，有些高端程序员（黑客）控制了10w个肉鸡，手里有10w个uid，同时发请求（先不考虑实名制的问题，小米抢手机不需要实名制），这下怎么办，站点层按照uid限流拦不住了。

第三层 服务层来拦截（反正就是不要让请求落到数据库上去）

服务层怎么拦截？大哥，我是服务层，我清楚的知道小米只有1万部手机，我清楚的知道一列火车只有2000张车票，我透10w个请求去数据库有什么意义呢？没错，请求队列！

对于写请求，做请求队列，每次只透有限的写请求去数据层（下订单，支付这样的写业务）：

* 1w部手机，只透1w个下单请求去db：
* 3k张火车票，只透3k个下单请求去db。

如果均成功再放下一批，如果库存不够则队列里的写请求全部返回“已售完”。

对于读请求，怎么优化？Cache抗，不管是memcached还是redis，单机抗个每秒10w应该都是没什么问题的。如此限流，只有非常少的写请求，和非常少的读缓存mis的请求会透到数据层去，又有99.9%的请求被拦住了。

当然，还有业务规则上的一些优化。回想12306所做的，分时分段售票，原来统一10点卖票，现在8点，8点半，9点，...每隔半个小时放出一批：将流量摊匀。

其次，数据粒度的优化：你去购票，对于余票查询这个业务，票剩了58张，还是26张，你真的关注么，其实我们只关心有票和无票？流量大的时候，做一个粗粒度的 “有票”“无票”缓存即可。

第三，一些业务逻辑的异步：例如 下单业务与 支付业务的分离。这些优化都是结合 业务 来的，我之前分享过一个观点“一切脱离业务的架构设计都是耍流氓”架构的优化也要针对业务。

最后是数据库层

浏览器拦截了80%，站点层拦截了99.9%并做了页面缓存，服务层又做了写请求队列与数据缓存，每次透到数据库层的请求都是可控的。db基本就没什么压力了，闲庭信步，单机也能扛得住，还是那句话，库存是有限的，小米的产能有限，透这么多请求来数据库没有意义。

全部透到数据库，100w个下单，0个成功，请求有效率0%。透3k到数据，全部成功，请求有效率100%。

五、总结

上文应该描述的非常清楚了，没什么总结了，对于秒杀系统，再次重复下我个人经验的两个架构优化思路：

1. 尽量将请求拦截在系统上游（越上游越好）；
2. 读多写少的常用多使用缓存（缓存抗读压力）；

浏览器和APP：做限速。 站点层：按照uid做限速，做页面缓存。 服务层：按照业务做写请求队列控制流量，做数据缓存。 数据层：闲庭信步。 以及结合业务做优化

六、Q&A

**问题1、按你的架构，其实压力最大的反而是站点层，假设真实有效的请求数有1000万，不太可能限制请求连接数吧，那么这部分的压力怎么处理？**

**答：**每秒钟的并发可能没有1kw，假设有1kw，解决方案2个：

1. 站点层是可以通过加机器扩容的，最不济1k台机器来呗。
2. 如果机器不够，抛弃请求，抛弃50%（50%直接返回稍后再试），原则是要保护系统，不能让所有用户都失败。

**问题2、“控制了10w个肉鸡，手里有10w个uid，同时发请求” 这个问题怎么解决哈？**

**答：**上面说了，服务层写请求队列控制

**问题3： 限制访问频次的缓存，是否也可以用于搜索？例如A用户搜索了“手机”，B用户搜索“手机”，优先使用A搜索后生成的缓存页面？**

**答：**这个是可以的，这个方法也经常用在 “动态”运营活动页，例如短时间推送4kw用户app-push运营活动，做页面缓存。

**问题4：如果队列处理失败，如何处理？肉鸡把队列被撑爆了怎么办？**

**答：**处理失败返回下单失败，让用户再试。队列成本很低，爆了很难吧。最坏的情况下，缓存了若干请求之后，后续请求都直接返回“无票”（队列里已经有100w请求了，都等着，再接受请求也没有意义了）。

**问题5：站点层过滤的话，是把uid请求数单独保存到各个站点的内存中么？如果是这样的话，怎么处理多台服务器集群经过负载均衡器将相同用户的响应分布到不同服务器的情况呢？还是说将站点层的过滤放到负载均衡前？**

**答：**可以放在内存，这样的话看似一台服务器限制了5s一个请求，全局来说（假设有10台机器），其实是限制了5s 10个请求，解决办法：

1. 加大限制（这是建议的方案，最简单）
2. 在nginx层做7层均衡，让一个uid的请求尽量落到同一个机器上

**问题6：服务层过滤的话，队列是服务层统一的一个队列？还是每个提供服务的服务器各一个队列？如果是统一的一个队列的话，需不需要在各个服务器提交的请求入队列前进行锁控制？**

**答：**可以不用统一一个队列，这样的话每个服务透过更少量的请求（总票数/服务个数），这样简单。统一一个队列又复杂了。

**问题7：秒杀之后的支付完成，以及未支付取消占位，如何对剩余库存做及时的控制更新 ？**

**答：**数据库里一个状态，未支付。如果超过时间，例如45分钟，库存会重新会恢复（大家熟知的“回仓”），给我们抢票的启示是，开动秒杀后，45分钟之后再试试看，说不定又有票哟。

**问题8：不同的用户 浏览同一个商品 落在不同的缓存实例 显示的库存完全不一样 请问老师怎么做缓存数据一致 或者是允许脏读？**

**答：**目前的架构设计，请求落到不同的站点上，数据可能不一致（页面缓存不一样），这个业务场景能接受。但数据库层面真实数据是没问题的。

**问题9：就算处于业务把优化考虑 “3k张火车票，只透3k个下单请求去db”那这3k个订单就不会发生拥堵了吗？**

**答：**（1）数据库抗3k个写请求还是ok的；（2）可以数据拆分；（3）如果3k扛不住，服务层可以控制透过去的并发数量，根据压测情况来吧，3k只是举例；

**问题10：如果在站点层或者服务层处理后台失败的话，需不需要考虑对这批处理失败的请求做重放？还是就直接丢弃？**

**答：**别重放了，返回用户查询失败或者下单失败吧，架构设计原则之一是“fail fast”。

**问题11：对于大型系统的秒杀，比如12306，同时进行的秒杀活动很多，如何分流？**

**答：**垂直拆分

**问题12：额外又想到一个问题。这套流程做成同步还是异步的？如果是同步的话，应该还存在会有响应反馈慢的情况。但如果是异步的话，如何控制能够将响应结果返回正确的请求方？**

**答：**用户层面肯定是同步的（用户的HTTP请求是夯住的），服务层面可以同步可以异步。

**问题13：秒杀群提问：减库存是在那个阶段减呢？如果是下单锁库存的话，大量恶意用户下单锁库存而不支付如何处理呢？**

**答：**数据库层面写请求量很低，还好，下单不支付，等时间过完再“回仓”，之前提过了。

讲师介绍

沈剑，58到家技术总监，互联网架构技术专家，“架构师之路”公众号作者，曾多次代表58集团作为演讲嘉宾参与业内技术会议，分享58的架构技术。

http://www.voidcn.com/blog/zqf\_office/article/p-6338003.html

# 大数据高并发系统架构解决千万上亿请求

[zqf\_office](http://www.voidcn.com/blog/zqf_office/) 2016-12-09  [技术资料](http://www.voidcn.com/blog/zqf_office/cata/1330570/)   10[原文链接](http://blog.csdn.net/zqf_office/article/details/53542533)

微信——腾讯战略级产品，创造[移动互联网](http://www.west.cn/cms/info/wireless/" \t "_blank)增速记录，10个月5000万手机用户，433天之内完成用户数从零到一亿的增长过程，千万级用户同时在线，摇一摇每天次数过亿……在技术架构上，微信是如何做到的?日前，在腾讯大讲堂在中山大学校园宣讲活动上，腾讯广研助理总经理、微信技术总监周颢在两小时的演讲中揭开了微信背后的秘密。



周颢，2001年毕业于华南理工大学，计算机专业硕士。2005年加入腾讯广州研发部，历任QQ邮箱架构师，广研技术总监，T4技术专家，微信中心助理总经理。

周颢把微信的成功归结于腾讯式的“三位一体”策略：即产品精准、项目敏捷、技术支撑。微信的成功是在三个方面的结合比较好，能够超出绝大多数同行或对手，使得微信走到比较前的位置。所谓产品精准，通俗的讲就是在恰当的时机做了恰当的事，推出了重量级功能，在合适的时间以最符合大家需求的方式推出去。他认为在整个微信的成功中，产品精准占了很大一部分权重。

敏捷是一种态度，敏捷就是试错

微信研发团队里鼓励一种试错的信仰：他们坚信，在互联网开发里，如果能够有一个团队在更短的时间内尝试了更多机会（并能改进过来），就能有（更多的）机会胜出。敏捷是一种态度，在软件开发过程中，项目管理者都会非常忌讳“变更”这个词，但是在微信的项目运作中是不可以的。因为微信必须要容忍说哪怕在发布前的十分钟，也要允许他变更。这是非常大的挑战，因为打破了所有传统项目开发的常识。所有人都说不可能做到的，但微信做到了。研发团队所做的一切都是要给产品决策者有最大的自由度，而这个决策正是微信能够胜出的关键。

海量系统上的敏捷，无异于悬崖边的跳舞

敏捷有很多困境，如果做一个单机版程序，是可以做到很敏捷的，但是腾讯正在运作的是一个海量系统，有千万级用户同时在线，在一个单独的功能上每天有百亿级的访问，同时还要保证99.95%的可用性。在海量系统上应对项目开发会有很严谨的规范，都说要尽可能少的变化，因为90%-95%的错误都是在变更中产生的，如果系统一直不变更会获得非常高的稳定度，但是微信就是要在悬崖边跳舞。微信的研发团队要做一些事情，让敏捷开发变得更简单。

如何做到这一切？周颢认为，首先，必须建立起一种狂热的技术信念，就是一定是可以做到的。然后，需要用一些稳固的技术（理念）来支撑，例如大系统小做、让一切可扩展、必须有基础组件、轻松上线（灰度、灰度、再灰度，精细监控，迅速响应）……等等来支撑。

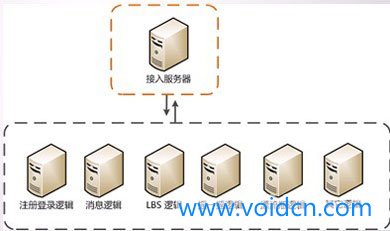
四大法器：大系统小做、让一切可扩展、要有基础组件、轻松上线

#### 大系统小做

当设计庞大系统的时候，应该尽量分割成更小的颗粒，使得项目之间的影响是最小的。一切可扩展：在高稳定度、高性能的系统中间，为了稳定性能把它设计成不变化的系统，但为了支持敏捷需要让一切的东西都要变得可以扩展。必须建立基础组件：要解决复杂问题的时候，需要将已有的经验固化下来，固化下来的东西会成为系统中的一部分。轻松上线：当做了变化并把它从开发环境中部署到现有的运营环境中去，在这个过程中，“灰度”这个词非常关键，就是在黑和白之间的选择，必须要变成一种小规模尝试，再逐步扩展到海量过程中的一个问题。

大系统小做——

仅仅把模块变得更为清晰，这在海量系统设计开发中是不够的，还需要在物理环境上进行分离部署，出现问题的时候可以快速发现，并且在最快的情况下解决掉。



大系统小做，混搭模式

将不同的应用逻辑物理分割独立出来，用户注册登录、LBS逻辑、摇一摇逻辑、漂流瓶逻辑、消息逻辑独立开来。把关键的逻辑混搭在一起，当所有的逻辑部署在同一个服务器上，确实也会带来很大敏捷上的好处，因为不需要额外的考虑部署和监控的问题。在整个微信的逻辑中，可能现在已经有上百种不同的逻辑，因为会在逻辑的分割上拆分成8-10种做分离部署。

#### 一切可扩展——网络协议可扩展、数据存储可扩展

扩展的关键点有两块。一个是网络协议需要扩展，当要升级一个新功能的时候，会有一些比较大的困难，所以所有协议设计都比较向前兼容，但是向前兼容还是不够的，因为网络协议设计本身有非常多的功能也会有比较大的字段，相关的代码可能会有数千行，这一块不能通过手写方式完成。可以通过XML描述，再通过工具自动生成所有的代码，这是微信获得快速开发的一个重要的点。

另外一块就是在数据存储方面是必须可扩展的。在2005年绝大多数海量系统的设计都是采用固定字段的存储，但是在现代系统中会意识到这个问题，会采用KV或者TLV的方式，微信也做了不同的设计。

#### 把复杂逻辑都固化下来，成为基础软件

在微信后台会有几种不同的基础组件，大致包括：

* Svrkit——Client/Server自动代码生成框架:10分钟搭建内部服务器
* LogicServer——逻辑容器：随时添加新逻辑
* OssAgent——监控/统计框架：所见即所得的监控
* 报表存储组件——屏蔽容灾/扩容等复杂问题

灰度、灰度、再灰度

在变更后的部署方式上，微信在一些规则会限定不能一次把所有的逻辑变更上去，每一次变更一小点观察到每一个环节没有问题的时候，才能布局到全网上去。微信后台每一天可以支撑超过20个后台变更，在业界来说，通常做到5个已经是比较快了，但是微信可以做到快4倍。

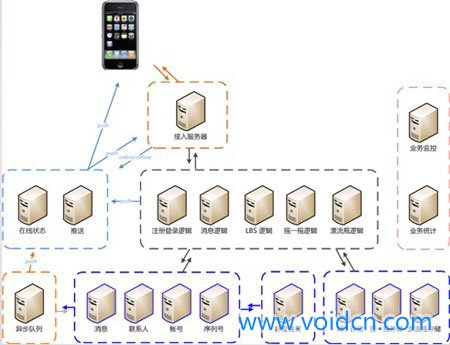


腾讯内部的上线系统

而所谓灰度发布，是指在黑与白之间，能够平滑过渡的一种发布方式。AB test就是一种灰度发布方式，让一部用户继续用A，一部分用户开始用B，如果用户对B没有什么反对意见，那么逐步扩大范围，把所有用户都迁移到B上面来。灰度发布可以保证整体系统的稳定，在初始灰度的时候就可以发现、调整问题，以保证其影响度。（在腾讯，灰度发布是最常采用的发布方式之一）

孙子兵法：古之所谓善战者，胜于易胜者也

常识上，解决一个复杂问题的时候，会用高明的技巧解决复杂的问题，这个不是微信团队的目标，他们追求的要做到让所有问题很自然和简单的方式解决掉。在周颢看来，微信架构的技术复杂点在四个要点：协议、容灾、轻重、监控。



微信架构

* 协议：手机终端跟后台服务器之间的交互协议，这个协议的设计是整个系统的骨架，在这一点做好设计可以使得系统的复杂度大大降低。
* 容灾：当系统出现了若干服务器或若干支架（宕机的时候），仍然需要让系统尽可能的提供正常的服务。
* 轻重：如何在系统架构中分布功能，在哪一个点实现哪一个功能，代表系统中间的功能配置。
* 监控：为系统提供一个智能仪表盘。

在协议设计上，移动互联网和常规互联网有很大的区别。首先有CMWAP和CMNET的不同，在中国现在有相当多的手机用户使用WMWAP连接，还有就是在线和离线的概念，当QQ下线的时候叫离线，当你登录的时候叫在线。但是在移动互联网这两个概念比较模糊。从微信的设计中，不管在线还是离线系统表现都应该是一致的。

还有一个是连接不稳定的问题，由于手机信号强弱的变化，当时信号很好，5秒钟走到信号不好的地区，连接就必须断掉。这个中间带来不稳定的因素为协议设计带来较大困难。此外就是资费敏感的问题，因为移动互联网是按照流量计费的，这个计费会使得在协议设计中如何最小化传输的问题。最后就是高延迟的问题。

对此，业界标准的解决方案：Messaging And Presence Protocol：1)XMPP，2)SIP/SIMPLE。它的优点是简单，大量开源实现。而缺点同样明显：1)流量大：状态初始化，2)消息不可靠。

微信在系统中做了特殊设计，叫SYNC协议，是参考Activesyec来实现的。特点首先是基于状态同步的协议，假定说收发消息本身是状态同步的过程，假定终端和服务器状态已经被迟了，在服务器端收到最新的消息，当客户端、终端向服务器对接的时候，收取消息的过程实际上可以简单的归纳为状态同步的过程，收消息以及收取你好友状态更新都是相同的。在这样的模式之下，我们会也许会把交互的模式统一化，只需要推送一个消息到达的通知就可以了，终端收到这个通知就来做消息的同步。在这样的简化模式之下，安卓和塞班都可以得到统一。这样的系统本身的实现是更为复杂的，但是获得很多额外的好处。

让剩下系统实现的部分更加简单，简化了交互模式，状态同步可以通过状态同步的差值获得最小的数据变更，通过增量的传输得到最小的数据传输量。通过这样的协议设计，微信可以确保消息是稳定到达的，而且是按序到达。引用一句俗话：比它炫的没它简单，比它简单的没它快，没谁比他更快，哪怕在GPRS下，微信也能把进度条轻易推到底。

追求完美设计的团队不能胜任海量服务

在容灾之前面向最坏的思考，如果系统真的挂了，需要做一些事情，首先是防止雪崩，避免蝴蝶效应。如果关注春节订火车票就知道了，用户的请求量会因为系统服务不了而不断的重试，意味着发生雪崩的时候，系统可能会承载原先3-10倍的流量，使得所有的事情更加恶化。所以微信有很多“放雪”功能的设计。

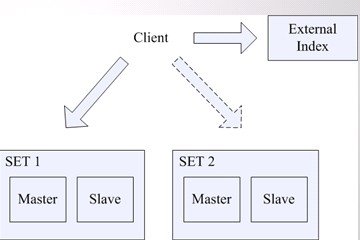
第二个词是柔性可用，在任何的系统中不要追求完美设计，追求完美设计的是团队是不能胜任海量服务的。如果在一个系统出现问题的时候，这个系统就挂了，那么这是一个不好的设计，最好的做法是提供0-1中间的选择。举一个例子，当一个用户向另外一个用户发消息的时候，可能会通过一个垃圾信息过滤的检测，如果垃圾信息过滤这个模块突然挂掉了，这个消息难道就不能达到了吗？在这样的情况下，要忽略掉这个错误，使得消息正常达到对方。要精确定位出哪一个环节是最为重要的，把不是重要的错误尽可能的忽略掉。当不能做到完美的时候，尽可能为用户提供服务。

另外一个重要方面叫做“保护点前置”，最前的一个点就是终端，在手机终端上蕴埋更多的保护点，这样会为用户系统赢得更大的处理空间。如果终端具备这样的能力，会获得更大的反应空间。

周颢介绍了在微信上具体容灾设计的做法。在所有的容灾中存储层的容灾是最难的，一个系统的设计分为三层：接入层、逻辑层、存储层。接入层和逻辑层的容灾都有比较成熟的方案。

逻辑层的容灾相对来说比较简单，尽量不要有状态的设计，比如说当你做上一个请求的时候，会保持一些状态，要使得下一个请求发到下一个服务器。如果任何一个请求之间互相不关联的话，这个就是无状态的设计，只要做到这一点逻辑层的容灾可以随意的切换。在回到存储层本身的容灾设计上，相对来说困难一些，但是微信研发团队采用了一些技巧，叫分而治之，分离业务场景，寻求简单的设计，并不会寻求大而同一的解决方案，因为这样会使得系统的复杂度大幅度上升，而微信会尽可能把产品拆细，寻求简化的设计。

首先是主备容灾，这是最常见的方案。在有一些业务场景中是可以容忍最终一致性的，比如账号系统的设计，每天写入账号系统的请求是非常少的，但是访问的请求非常多，这个差异可能会达到数万倍的规模，在这样的场景下，微信会在账号系统中采用简化的方案，也可以获得比较大的稳定度。



SET模型+双写

第二种容灾的模式叫双写，两台Master的机器，当一台机故障的时候，另外一台机还是可以接收到写请求，当两台机交错启动的时候，会得到数据的丢失。但是有一些场景是可以容忍轻度数据丢失的，比如说会有一个存储专门记录用户终端的类型，比如说安卓还是塞班以及他们使用终端的微信版本是什么，这样的数据是可以容忍轻度数据丢失的，因为偶尔有一些丢失的话，下一次访问会把这些数据带上来，会尽快的修复所有的数据。双写也是非常简单的模式。

微信的研发团队做了一个叫Simple Quorum的机制，在微信的后台中，同步协议有一个很重要的基石叫序列发生器，这样的一个序列发生器需要有极高的稳定度。首先可以看到序列号有一个特点永远是递增的，用递增方式往前推进的时候，最大的序列号就是最新的系列号。有一个毕业才加入广研的毕业生想到一个绝佳的方案，按SET分布，从2G减到 200K。

前轻后重，功能点后移

周颢还谈到了轻重的概念。这个概念的提出主要是从终端本身的一些困境所带来的。首先在终端上需要表现最多的一个产品的逻辑，逻辑非常复杂，变更的成本也非常高，当需要修复的时候必须发布一个新版本，这个新版必须由自己下载才能完成，下载的成本非常高。在这样的前提下，如果手机终端产生了任何变化的时候，如果这个变化有非常大的问题就会有极大的困境，所以需要在每一个发布之前做一些充分的数据，确保不会发生致命问题。如果一旦出现致命问题难以修复，需要把关键的点从终端移到后台实现，把功能点后移，来充分发挥后台快速变更的能力。

接入优化：从GSLB到IP重定向

在接入层的优化，速度很重要的因素，是不是能够就近接入一个最优的节点，比如说移动用户最好接入移动的节点，海外的用户可能需要寻找更佳的路由，有的时候可能无法自动做到这一点，一点是在终端上做测速，微信会通过在后台IP逆向的能力，通过后台指挥微信终端联网的能力，寻找最优的接入点。上图就是每分钟收到同一项指令曲线的报表。

如何解决“偷流量”的问题——当国内类微信类产品发布的时候出现一个大的问题就是“偷流量”，当用户在某一些逻辑下进行一个死循环，不断访问某一些数据，这样的死循环是非常可怕的，如果在用户不知觉的情况之下，可能会在一个小时之内偷到数10兆甚至数百兆的流量。有非常多业内的同行都需要花大量的精力解决这个问题，微信研发团队用了非常强大的方式解决它。通过在后台建立起严厉的监控系统，对每一个用户的行为做一个监控，当发现异常的时候，后台会给终端发出指令，使得微信终端在一段时间无法联网，但是可以保证用户流量不会白白的使用掉。

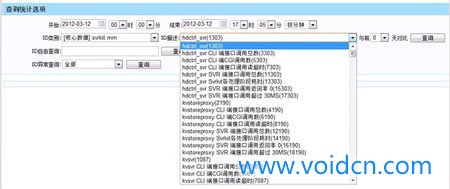
功能适配的例子——第一期微信版本发布的时候，当时没有群聊的功能，第二版发布的时候做了这个功能。当时有两个选择，对于早期版本的用户，因为不支持群聊，就无法享用到这个功能，但是微信希望提供更好的选择，想让早期不支持群聊的版本，也可以被拉到一个群里面收消息、发消息，通过后台功能的适配也能做到这个事情。

分而治之，把监控嵌入基础框架

对于一个海量系统来说，一个精密的仪表盘非常重要。监控是非常痛苦的，对于这样一个系统来说，每小时会产生数百G的监控日志。微信希望在1分钟之内监控的数据就能够显示在报表上，因为只有这样的精准和实时度才能够赢得处理故障的时间。微信会做关联统计，通过摇一摇加了好友，他们活跃度如何，过了一段时间他们的活跃度变化情况又是如何。这种需求是需要通过大量日志的关联统计来获得的。研发团队也花了一段时间来理解这个问题，发现了中间一个重要的经验叫做“鱼和熊掌不能兼得”。

为了让监控数值更敏感，需要把监控细化再细化，上面数据表示每一栏子系统的数据，下面这个是按微信版本号来划分的，这里的数据项是非常多。

微信还需要采集一些异常的点，如果有异常的话会发布紧急的版本，尽可能快的替换它。对收发消息延时做的监控，比如说0—1秒端到端的速度，会对不同的区段做一些统计，当某一个环节出现异常的时候，通常会在中间的延时上体现出来。有一个很重要的点叫自动报警，现在有数千项的数据，不可能每一项都靠人工去看的，必须要跟自动报警相关联，微信有一些智能的算法，是不是在正常的范围内，跟历史的数值进行对比，如果有异常的话，会通过短信、邮件还有微信本身来发出报警信息。



把监控嵌入基础框架

微信会把监控嵌入到基础框架里面去，因为并不是每一个人都会意识到在需要的地方嵌入一个监控点，所以在基础框架本身内置很重要的监控点，比如说这个表上的栏目，非常多的栏目大概会有数百项的栏目，都不需要程序员自己去写，当用基础组件搭建一个系统的时候，就可以直接观测系统数据。

在谈到微信未来的技术挑战时，周颢首先希望能够让微信成为可用性99.99%的系统;设计出面向现在10倍容量的系统以及完全的[idc](http://www.west.cn/" \t "_blank)容灾。

http://lib.csdn.net/article/php/10768

# 使用HAProxy、PHP、Redis和MySQL支撑10亿请求每周架构细节

作者：[aw344](http://my.csdn.net/aw344)

#### 发表于1小时前|536次阅读| 来源High Scalability|2 条评论| 作者Todd Hoff

[大数据](http://www.csdn.net/tag/%E5%A4%A7%E6%95%B0%E6%8D%AE/news)[架构](http://www.csdn.net/tag/%E6%9E%B6%E6%9E%84/news)[HAProxy](http://www.csdn.net/tag/haproxy/news)[PHP](http://www.csdn.net/tag/php/news)[Redis](http://www.csdn.net/tag/redis/news)[MySQL](http://www.csdn.net/tag/mysql/news)

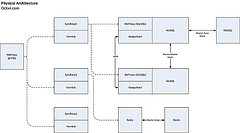
**摘要：**如果你还在为公司服务器架构的问题而纠结，不妨来看看Antoni Orfin是如何实现使用HAProxy、PHP、Redis和MySQL支撑每周10亿请求的架构的，更聪明的使用简单的工具提高服务器的性能。

【编者按】在公司的发展中，保证服务器的可扩展性对于扩大企业的市场需要具有重要作用，因此，这对架构师提出了一定的要求。本文将向你介绍一个非常简单的架构，使用HAProxy、PHP、Redis和MySQL就能支撑每周10亿请求。同时，你还能了解项目未来的横向扩展途径及常见的模式。

**免费订阅“CSDN云计算（左）和CSDN大数据（右）”微信公众号，实时掌握第一手云中消息，了解最新的大数据进展！**

[](http://cms.csdnimg.cn/article/201408/11/53e87db637d1b.jpg)[](http://cms.csdnimg.cn/article/201408/11/53e87dc4b2f5c.jpg)CSDN发布虚拟化、Docker、OpenStack、CloudStack、数据中心等相关云计算资讯，     分享Hadoop、Spark、NoSQL/NewSQL、HBase、Impala、内存计算、流计算、机器学习和智能算法等相关大数据观点，提供云计算和大数据技术、平台、实践和产业信息等服务。

**以下为译文：**

[](http://cms.csdnimg.cn/article/201408/14/53ec44891815e.jpg)

本文的作者是AntoniOrfin，Octivi联合创始人兼软件架构师。

在这篇文章中，我将展示一个非常简单的架构，使用HAProxy、PHP、Redis和MySQL支撑每周10亿请求。除此之外，我还将展示项目未来的横向扩展途径及常见的模式，下面我们一起看细节。

**状态：**

* 服务器

1. 3个应用程序节点
2. 2个MySQL+1个备份
3. 2个Redis

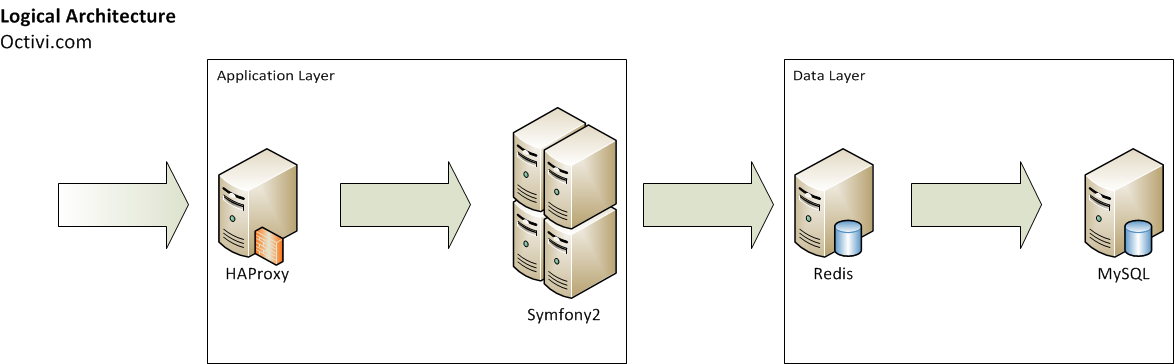
* 应用程序

1. 应用程序每周处理10亿请求
2. 峰值700请求每秒的单Symfony2实例（平均工作日约550请求每秒）
3. 平均响应时间30毫秒
4. Varnish，每秒请求超过1.2万次（压力测试过程中获得）

* 数据存储

1. Redis储存了1.6亿记录，数据体积大约100GB，同时它是我们的主要数据存储
2. MySQL储存了3亿记录，数据体积大约300GB，通常情况下它作为三级缓存层

**平台：**

[](http://cms.csdnimg.cn/article/201408/14/53ec4a4ce78c0.jpg)

* 监视：

1. Icinga
2. Collectd

* 应用程序

1. HAProxy + Keepalived
2. Varnish
3. PHP（PHP-FPM）+ Symfony2 Framework

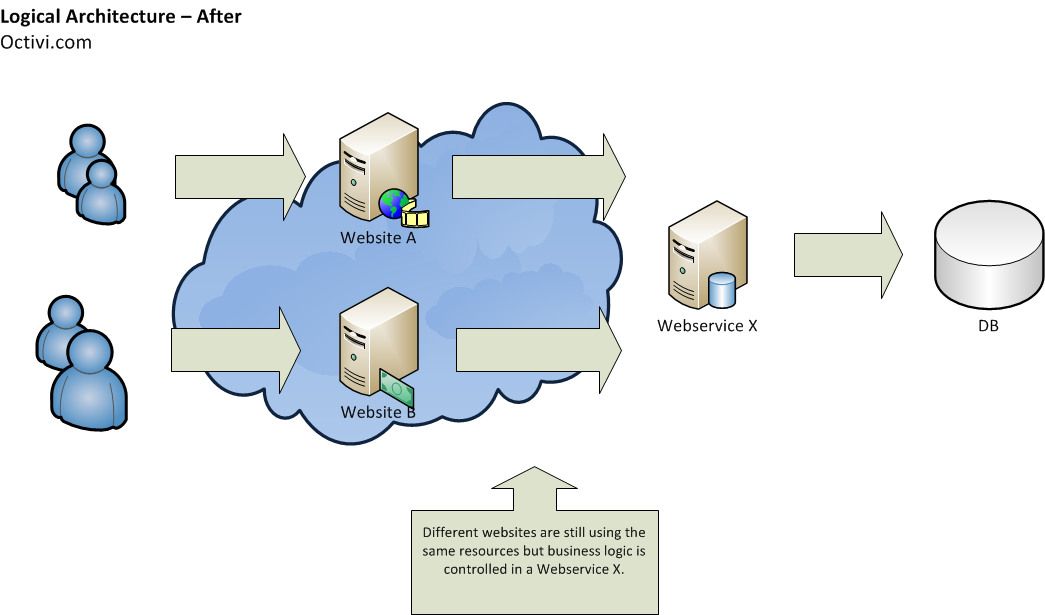
* 数据存储

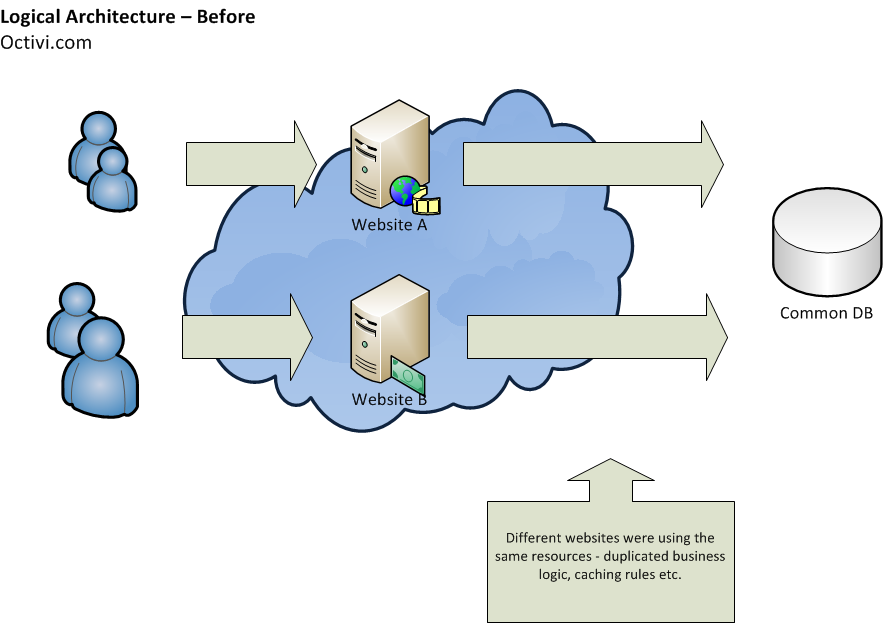
1. MySQL（主从配置），使用HAProxy做负载均衡
2. Redis （主从配置）

## **背景**

大约1年前，一个朋友找到我并提出了一个苛刻的要求：它们是一个飞速发展的电子商务初创公司，而当时已经准备向国际发展。介于那个时候他们仍然是一个创业公司，初始解决方案必须符合所谓的成本效益，因此也就无法在服务器上投入更多的资金。遗留系统使用了标准的LAMP堆栈，因此他们拥有一个强力的PHP开发团队。如果必须引入新技术的话，那么这些技术必须足够简单，不会存在太多架构上的复杂性；那么，他们当下的技术团队就可以对应用进行长期的维护。

为了满足他们扩展到下一个市场的需求，架构师必须使用可扩展理念进行设计。首先，我们审视了他们的基础设施：

[](http://cms.csdnimg.cn/article/201408/14/53ec4ac7a4b67.jpg)

[](http://cms.csdnimg.cn/article/201408/14/53ec4afa6fbed.jpg)

老系统使用了单模块化设计思路，底层是一些基于PHP的Web应用程序。这个初创公司有许多所谓的前端网站，它们大多都使用了独立的数据库，并共享了一些支撑业务逻辑的通用代码。毫不客气的说，长期维护这种应用程序绝对是一个噩梦：因为随着业务的发展，有些代码必须被重写，这样的话，修改某个网站将不可避免导致业务逻辑上的不一致，这样一来，他们不得不在所有Web应用程序上做相同的修改。

通常情况下，这该归结于项目管理问题，管理员必须对横跨多个代码库的那些代码负责。基于这个观点，整改第一步就是提取核心的业务关键功能，并将之拆分为独立的服务（这也是本文的一个重点部分），也就是所谓的面向服务架构，在整个系统内遵循“separation of concern”原则。每个服务只负责一个业务逻辑，同时也要明确更高等级的业务功能。举个形象的例子也就是，这个系统可能是个搜索引擎、一个销售系统等。

前端网站通过REST API与服务交互，响应则基于JSON格式。为了简单起见，我们选择了SOAP，一个开发者比较无爱的协议，因为谁都不愿意解析一堆的XML。

提取一些不会经常处理的服务，比如身份验证和会话管理。这是非常必要的一个环节，因为它们的处理等级比较高。前端网站负责这个部分，只有它们可以识别用户。这样一来我们可以保持服务的足够简单，在处理扩展和代码相关问题时都具有巨大的优势，可谓各司其职，完美无缺。

**带来的好处：**

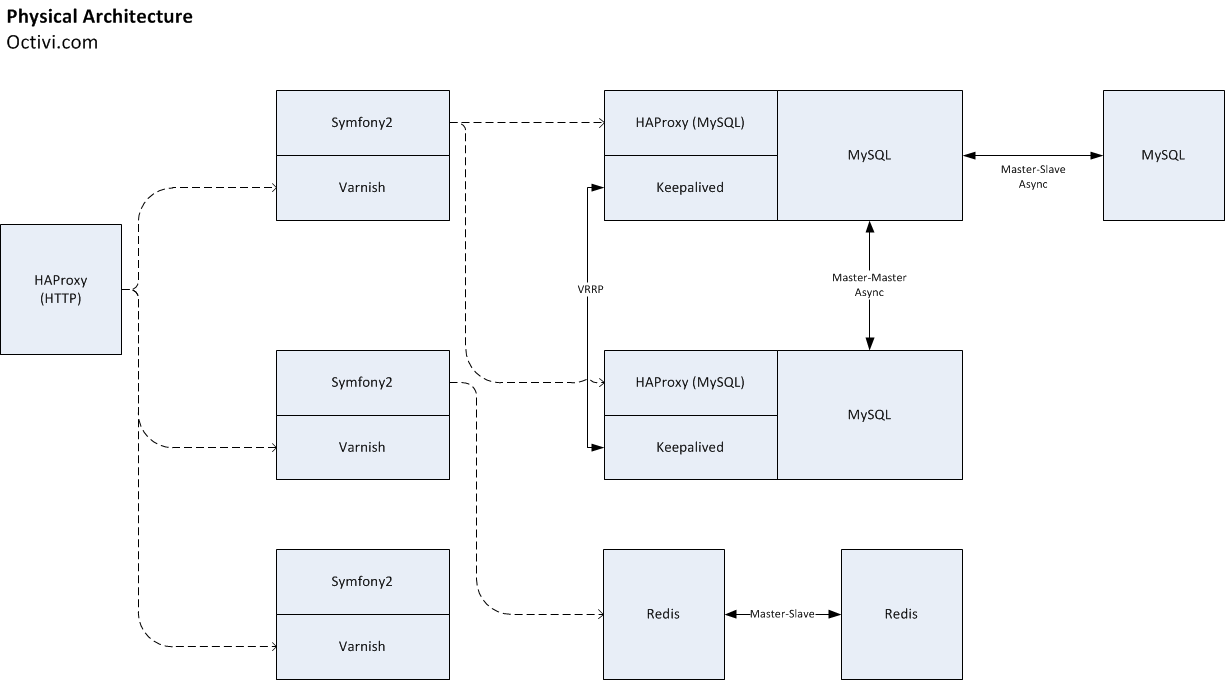
* 独立子系统（服务）可以便捷的在不同团队中开发，开发者互不干涉，效率理所当然提升。
* 身份验证和会话不会通过它们来管理，因此它们造成的扩展问题不翼而飞。
* 业务逻辑被区分，不同的前端网站不会再存在功能冗余。
* 显著地提高了服务的可用性。

**共生的缺点：**

为系统管理员带来更大的工作量。鉴于服务都使用了独立的基础设施，这将给管理员带来更多需要关注的地方。

很难保持向后兼容。在一年的维护之后，API方法中发生了数不尽的变化。因此问题发生了，它们必将破坏向后兼容，因为每个网站的代码都可能发生变化，还可能存在许多技术人员同时修改一个网站的情况……然而，一年后，所有方法匹配的仍然是项目开始时建立的文档。

## **应用程序层**

[](http://cms.csdnimg.cn/article/201408/14/53ec4b6110d50.jpg)

着眼请求工作流，第一层是应用程序。HAProxy负载均衡器、Varnish和Symfony2应用程序都在这一层。来自前端网站的请求首先会传递给HAProxy，随后负载均衡器将把他分给不同的节点。

**应用程序节点配置**

* Xeon E5-1620@3.60GHz，64GB RAM，SATA
* Varnish
* Apache2
* PHP 5.4.X（PHP-FPM），使用APC字节码缓存

我们购买了3个这样的服务器，N+1冗余配置的active-active模式，备份服务器同样处理请求。因为性能不是首要因素，我们为每个节点配置独立的Varnish以降低缓存hit，同时也避免了单点故障（SPOF）。在这个项目中，我们更重视可用性。因为一个前端网站服务器中使用了Apache2，我们保留了这个堆栈。这样一来，管理员不会困扰于太多新加入的技术。

**Symfony2应用程序**

应用程序本身基于Symfony2建立，这是一个PHP全堆栈框架，提供了大量加速开发的组件。作为基于复杂框架的典型REST服务可能受到很多人质疑，这里为你细说：

* **对 PHP/Symfony 开发者友好。**客户端IT团队由PHP开发者组成，添加新技术将意味必须招聘新的开发者，因为业务系统必须做长时间的维护。
* **清晰的项目结构。**PHP/Symfony虽然从来都不是必需品，但却是许多项目的默认选择。引入新的开发者将非常方便，因为对他们来说代码非常友好。
* **许多现成的组件。**遵循DRY思想……没有人愿意花力气去做重复的工作，我们也不例外。我们使用了大量的Symfony2Console Component，这个框架非常有利于做CLI命令，以及应用程序性能分析（debug工具栏）、记录器等。

在选用Symfony2之前，我们做了大量的性能测试以保证应用程序可以支撑计划流量。我们制定了概念验证，并使用JMeter执行，我们得到了让人满意的结果——每秒700请求时响应时间可以控制在50毫秒。这些测试给了我们足够的信心，让我们坚信，即使Symfony2这样复杂的框架也可以得到理想的性能。

**应用程序分析与监控**

我们使用Symfony2工具来监视应用程序，在收集指定方法执行时间上表现的非常不错，特别是那些与第三方网络服务交互的操作。这样一来，我们可以发现架构中潜在的弱点，找出应用程序中最耗时的部分。

冗长的日志同样是不可缺少的一部分，我们使用PHP Monolog库把这些日志处理成优雅的log-lines，便于开发者和管理员理解。这里需要注意的是尽可能多地添加细节，越详细越好，我们使用了不同的日志等级：

* **Debug，**可能会发生的事情。比如，请求信息在调用前会传送给一个外部Web服务；事情发生后从API调用响应。
* **Error，**当错误发生时请求流并未被终止，比如第三方API的错误响应。
* **Critical，**应用程序崩溃的瞬间。

因此，你可以清晰地了解Error和Critical信息。而在开发/测试环境中，Debug信息同样被记录。同时，日志被存储在不同的文件中，也就是Monolog库下的“channels”。系统中有一个主日志文件，记录了所有应用程序级错误，以及各个channel的短日志，从单独的文件中记录了来自各个channel的详细日志。

**扩展性**

扩展平台的应用程序层并不困难，HAProxy性能并不会在短时间耗尽，唯一需要考虑的就是如何冗余以避免单点故障。因此，当下需要做的只是添加下一个应用程序节点。

## **数据层**

我们使用Redis和MySQL存储所有的数据，MySQL更多作为三级缓存层，而Redis则是系统的主要数据存储。

### **Redis**

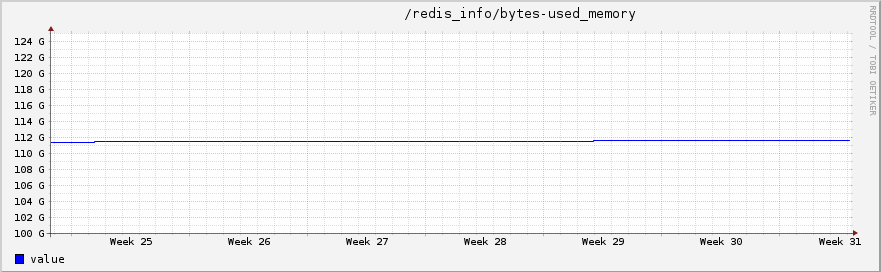
在系统设计时，我们基于以下几点来选择满足计划需求的数据库：

* 在存储大量数据时不会影响性能，大约2.5亿记录
* 通常情况下多是基于特定资源的简单GET请求，没有查找及复杂的SELECT操作
* 在单请求时尽可能多的获得资源以降低延时

在经过一些调查后，我们决定使用Redis

* 大部分我们执行的操作都具有O（1）或O（N）复杂性， N是需要检索键的数量，这意味着keyspace大小并不会影响性能。
* 通常情况下会使用MGET命令行同时检索100个以上的键，这样可以尽可能的避免网络延时，而不是在循环中做多重GET操作。

我们当下拥有两个Redis服务器，使用主从复制模式。这两个节点的配置相同，都是Xeon E5-2650v2@2.60GHz，128GB，SSD。内存限制被设置为100GB，通常情况下使用率都是100%。

[](http://cms.csdnimg.cn/article/201408/14/53ec4bf008fff.jpg)

在应用程序并没有耗尽单个Redis服务器的所有资源时，从节点主要作作备份使用，用以保证高有效性。如果主节点宕机，我们可以快速的将应用程序切换到从节点。在维护和服务器迁移时，复制同样被执行——转换一个服务器非常简单。

你可能会猜想当Redis资源被一直耗尽时的情景，所有的键都是持久化类型，大约占90%keyspace，剩余资源被全部被用于TTL过期缓存。当下，keyspace已经被分为两个部分：一个是TTL集（缓存），另一个则是用于持久化数据。感谢“volatile-lru”最大化内存设置的可行性，最不经常使用缓存键会被移除。如此一来，系统就可以一直保持单Redis实例同时执行两个操作——主存储和通用缓存。

使用这个模式必须一直监视“期满”键的数量：

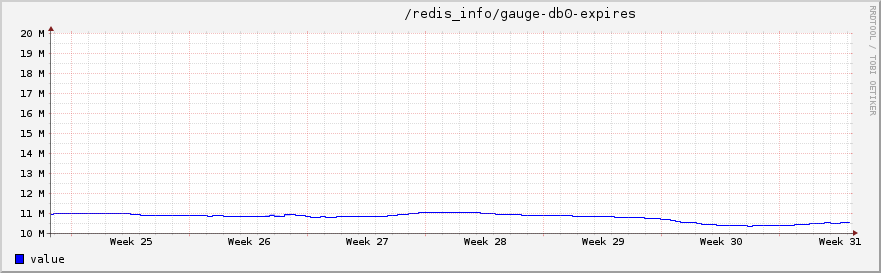
db.redis1:6379> info keyspace

# Keyspace

db0:keys=16XXXXXXX,expires=11XXXXXX,avg\_ttl=0

“期满”键数量越接近0情况越危险，这个时候管理员就需要考虑适当的分片或者是增加内存。

我们如何进行监控？这里使用Icingacheck，仪表盘会显示数字是否会达到临界点，我们还使用了Redis来可视化“丢失键”的比率。

[](http://cms.csdnimg.cn/article/201408/14/53ec4c2199d63.jpg)

在一年后，我们已经爱上了Redis，它从未让我们失望，这一年系统从未发生任何宕机情况。

### **MySQL**

在Redis之外，我们还使用了传统RDBMS——MySQL。但是区别于他人，我们通常使用它作为三级缓存层。我们使用MySQL存储一些不会经常使用对象以降低Redis的资源使用率，因此它们被放到了硬盘上。这里没有什么可说道的地方，我们只是尽可能地让其保持简单。我们使用了两个MySQL服务器，配置是XeonE5-1620@3.60GHz，64GB RAM，SSD。两个服务器使用本地、异步的主-主复制。此外，我们使用一个单独的从节点作为备份。

**MySQL的高可用性**

在应用程序中，数据库永远是最难的瓶颈。当前，这里还不需要考虑横向扩展操作，我们多是纵向扩展Redis和MySQL服务器。当下这个策略还存在一定的发展空间，Redis运行在一个126GB内存的服务器上，扩展到256GB也并不困难。当然，这样的服务器也存在劣势，比如快照，又或是是简单的启动——Redis服务器启动需要很长的时间。

在纵向扩展失效后进行的必然是横向扩展，值得高兴的是，项目开始时我们就为数据准备了一个易于分片的结构：

在Redis中，我们为记录使用了4个“heavy”类型。基于数据类型，它们可以分片到4个服务器上。我们避免使用哈希分片，而是选择基于记录类型分片。这种情况下，我们仍然可以运行MGET，它始终在一种类型键上执行。

在MySQL上，结构化的表格非常易于向另一台服务器上迁移——同样基于记录类型（表格）。当然，一旦基于记录类型的分片不再奏效，我们将转移至哈希。

## **学到的知识**

* **不要共享你的数据库。**一旦一个前端网站期望切换会话处理到Redis，Redis缓存空间将被耗尽，同时它会拒绝应用程序保存下一个缓存键。这样一来所有的缓存将转至MySQL服务器，这将导致大量开销。
* **日志越详细越好。**如果log-lines中没有足够的信息，快速Debug问题定位将成为难点。如此一来，你不得不等待一个又一个问题发生，直到找到根结所在。
* **架构中使用复杂的框架并不意味着低性能。**许多人惊讶我们使用全堆栈框架来支撑如此流量应用程序，其秘诀在于更聪明的使用工具，否则即使是Node.js也可能变得很慢。选择一个提供良好开发环境的技术，没有人期望使用一堆不友好的工具，这将降低开发团队士气。

原文链接：The Easy Way Of Building A Growing Startup Architecture Using HAProxy, PHP, Redis And MySQL To Handle 1 Billion Requests A Week

http://www.tokiwinter.com/highly-available-load-balancing-of-apache-tomcat-using-haproxy-stunnel-and-keepalived/

# Highly-Available Load Balancing of Apache Tomcat using HAProxy, stunnel and keepalived

This article will describe the process required to set up a highly-available SSL-enabled load balancer using HAProxy, stunnel and keepalived to front a pair of Apache Tomcat instances.

The configuration will start off simple, and extend and grow as more functionality is added. First, a session-aware HAProxy load balancer is configured. Next, Tomcat clustering is configured with session replication and the responsibility for maintaining session state is moved to Tomcat. Then, keepalived is added – providing a failover IP between HAProxy instances running on two nodes. The solution is then fully HA. To complete the article, SSL support will be enabled by way of adding stunnel in front of the HAProxy instances, and a few cleanup tasks performed.

Faithful **dolan** and **gooby** will be used for this in my lab environment. Each of the servers is running CentOS 6.3 x86\_64 and already has a Tomcat instance installed to/usr/local/tomcat7 (running as the tomcat user).



|  |  |
| --- | --- |
| 1  2  3 | 172.16.18.169   dolan www1.tokiwinter.com  172.16.18.172   gooby www2.tokiwinter.com  172.16.18.199   vip-address |

As you can see, I’ve also reserved an IP address for our floating IP (or Virtual IP – VIP).

The end goal is to have **dolan** and **gooby** both running Tomcat as before, also running their own HAProxy and stunnel instances but also running a failover VIP provided by keepalived.

## Session-aware Load Balancing with HAProxy

First, download [HAProxy](http://haproxy.1wt.eu/) and install to the default location (which will place the binary at/usr/local/sbin/haproxy, and expect its configuration to be at/etc/haproxy/haproxy.conf:



|  |  |
| --- | --- |
| 1  2  3  4 | # cd /usr/local/src  # tar xzf haproxy-1.4.22.tar.gz  # make TARGET=linux2628  # make install |

Install the software on both nodes. Only configure on **dolan** for now as HA will be added later. Create haproxy.conf as follows:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | # mkdir /etc/haproxy  # vi /etc/haproxy/haproxy.conf  global          log 127.0.0.1 local0          maxconn 4096          daemon    defaults          log global          mode http          option httplog          option dontlognull          retries 3          option redispatch          maxconn 2000          contimeout 5000          clitimeout 50000          srvtimeout 50000    frontend http-in          bind \*:80          default\_backend servers    backend servers          option httpchk OPTIONS /          option forwardfor          option http-server-close          appsession JSESSIONID len 52 timeout 3h          server node1 172.16.18.169:8080 check inter 5000          server node2 172.16.18.172:8080 check inter 5000 |

Most of the configuration above are sensible defaults taken from the HAProxy documentation. A few points to note: HAProxy is configured to bind on all interfaces on port 80 – this will enable us to test by hitting individual nodes, or to hit the floating IP address that we will configure later. Within the backend configuration, option http-server-close is very important – without this, only the first request will have its headers inspected for cookies and hence our JSESSIONID – so sticky sessions will not work. Newer versions of HAProxy (I’m using 1.4.22) support the appsession directive. This will, in our configuration, store the JSESSIONIDin a hashtable (well – 52 bytes of it) for 3 hours, along with the backend node that serviced the request – thus implementing sticky sessions.

This can also be done with the following backend declaration:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | backend servers          option httpchk OPTIONS /          option forwardfor          option http-server-close          cookie JSESSIONID prefix          server node1 172.16.18.169:8080 cookie node1 check inter 5000          server node2 172.16.18.172:8080 cookie node2 check inter 5000 |

This will cause HAProxy to check for a specific nodename value within the session ID.

I opted for appsession. Configure the Tomcat <Engine> directive to correctly supportjvmRoute based load balancing (only important if you opt for the second configuration). This will set the JSESSIONID as SESSIONID.jvmRoute which will enable HAProxy to route the request appropriately (as we define cookie checking in the server directives in the second example). Even when using appsession (which maintains its own session state table) it will help for testing to see which backend node set the SESSIONID in the first place, so I’ll set it:



|  |  |
| --- | --- |
| 1  2  3  4  5 | # vi server.xml  change  <Engine name="Catalina" defaultHost="localhost">  to  <Engine name="Catalina" defaultHost="localhost" jvmRoute="node1"> |

These values should be set to whichever nodename you’re checking for in your HAProxy configuration, if not using appsession (or even if you are – see above). Restart Tomcat (depending upon your setup this will be probably be different):



|  |  |
| --- | --- |
| 1  2  3  4 | # su - tomcat7  $ cd /usr/local/tomcat7/bin  $ ./catalina.sh stop  $ ./catalina.sh start |

Now, start HAProxy referencing the configuration file created previously. Note: HAProxy does support running inside a chroot as well as dropping privileges to a different user – I’ve not bothered for this simple article – it is easy to do and is well documented):



|  |  |
| --- | --- |
| 1 | # haproxy -f /etc/haproxy/haproxy.conf |

Next, browse to (presuming you’ve left the Tomcat examples intact)



|  |  |
| --- | --- |
| 1 | http://<haproxy-node1-name>/examples/servlets/servlet/SessionExample |

You will see your session ID returned. Refresh the page a few times, add a few session variables using the example, make sure the same session ID is returned each time. If you see new session IDs being generated, check your Tomcat access logs – you’ve misconfigured something and are probably round-robin load balancing across your Tomcat nodes, and your whole setup is session-unaware. Also check all your HAProxy directives are correct. Logging will be configured for HAProxy towards the end of the article once everything else is setup.

So – we now have a session-aware load balancer configured using HAProxy and appsession-based session tracking. The trouble with this configuration is that if we chose to make it highly-available and attempt a failover of HAProxy to a second node, the session state information would not be failed over with it – and all sessions would be broken.

## Tomcat Clustering

A rethink is required. Presuming your well-behaved application deployed to Tomcat is distributable and serialised, session replication can be configured within Tomcat. Presuming you have no other clusters running on your network, you can just use the default cluster configuration supplied with Tomcat and implemented via theorg.apache.catalina.ha.tcp.SimpleTcpCluster class. Remove the jvmRoute statements added to your <Engine> directives if you’ve opted for using them (depending on how you implemented the first stage of load balancing). Uncomment, or add, the following toserver.xml:



|  |  |
| --- | --- |
| 1 | <Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"/> |

You can modify the default configuration by following the documentation available on the[Tomcat site](http://tomcat.apache.org/tomcat-7.0-doc/cluster-howto.html). Restart your Tomcat instances.

If you check catalina.out, you’ll see messages such as the following when cluster membership changes. This example is taken from **dolan** as **gooby** joined the cluster:



|  |  |
| --- | --- |
| 1 | Feb 11, 2013 4:10:55 AM org.apache.catalina.ha.tcp.SimpleTcpCluster memberAdded INFO: Replication member added: org.apache.catalina.tribes.membership.MemberImpl[tcp://{172, 16, 18, 172}:4000,{172, 16, 18, 172},4000, alive=1048, securePort=-1, UDP Port=-1, id={-91 -111 103 -21 -38 -118 71 44 -99 113 -87 -112 -27 18 55 -30 }, payload={}, command={}, domain={}, ] |

Next, again presuming your Tomcat examples are still deployed to your instances, modify/path/to/tomcat7/webapps/examples/WEB-INF/web.xml, and add the following just above the <filter> directives:



|  |  |
| --- | --- |
| 1 | <distributable/> |

You will need to restart Tomcat or redeploy the application. We will be able to test using one of the examples provided. It is worth noting from the Tomcat 7.0 documentation:

- All your session attributes must implement java.io.Serializable  
- Uncomment the Cluster element in server.xml  
- Make sure your web.xml has the distributable element

So, if you do not implement java.io.Serializable then it’s still not going to work. I’ll just opt for one of the examples that does – but (as an example) the shopping cart example provided with Tomcat 7.0 does not implement java.io.Serializable. You can modify the code and recompile the class if you’d like to test further, but the Hello World JSP test will work fine, along with a header inspector to check our session IDs.

Next, modify the HAProxy configuration as follows:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | # vi /etc/haproxy/haproxy.conf  ...  backend servers  option httpchk OPTIONS /  option forwardfor  option http-server-close  server node1 172.16.18.169:8080 check inter 5000  server node2 172.16.18.172:8080 check inter 5000 |

As you can see, there is no special handling of cookies or sessions with our HAProxy configuration – it will all be handled by session replication within Tomcat.

Restart HAProxy:



|  |  |
| --- | --- |
| 1  2 | # pkill -f haproxy  # haproxy -f /etc/haproxy/haproxy.conf |

Browse to the following whilst using something like Live HTTP Headers with Firefox:



|  |  |
| --- | --- |
| 1 | http://<haproxy-node1-name>/examples/jsp/jsp2/simpletag/hello.jsp |

Note the session ID. Refresh your browser a few times; you should note the session ID doesn’t change, but the load is still being spread across your backend Tomcat servers (checking your Tomcat access logs for now – HAProxy will be configured for logging later).

Shutdown a tomcat instance, the session ID will be the same. Bring a Tomcat instance back up, and shut the other one down – you should note the ID is not changing. Session replication is working at the Tomcat layer. If the session ID is changing, check your Tomcat cluster configuration, and that you’ve made the application both distributable and implemented serialisation if required.

## Making our Load Balancer Highly-Available: keepalived

We can now make our load balancer highly-available by installing keepalived. First, copy the HAProxy configuration across from the first node to the second node, as it will not change any further:



|  |  |
| --- | --- |
| 1  2 | # cd /etc/haproxy  # scp -p user@node1:/etc/haproxy/haproxy.conf . |

Test HAProxy on the second node:



|  |  |
| --- | --- |
| 1 | # haproxy -f /etc/haproxy/haproxy.conf |

Hit the Tomcat examples and verify that all is working as expected. All remaining steps should be performed on both nodes unless noted.

Install the required prerequisite packages:



|  |  |
| --- | --- |
| 1 | # yum -y install kernel-headers kernel-devel popt-devel |

Now, download the latest [keepalived](httpd://www.keepalived.org) and install it:



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | # cd /usr/local/src  # tar xzf keepalived-1.2.7.tar.gz  # cd keepalived-1.2.7  # ./configure --with-kernel-dir=/lib/modules/$(uname -r)/build  # make  # make install |

Without specifying a --prefix during the compile, make install happily scatters files all over/usr/local. Ho hum. For now, copy the appropriate files into place:



|  |  |
| --- | --- |
| 1  2  3  4 | # cp /usr/local/etc/rc.d/init.d/keepalived /etc/init.d  # chmod +x /etc/init.d/keepalived  # chkconfig keepalived on  # cp /usr/local/etc/sysconfig/keepalived /etc/sysconfig |

Next, configure keepalived:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | # mkdir /etc/keepalived  # vi /etc/keepalived/keepalive.conf  vrrp\_script check\_haproxy {     script "killall -0 haproxy"     interval 2     weight 2  }    vrrp\_instance VI\_1 {     interface eth0     state MASTER     virtual\_router\_id 1     priority 101          # 101 on master, 100 on backup     virtual\_ipaddress {        172.16.18.199     }     track\_script {        check\_haproxy     }  } |

As per the comment – configure the master server (**dolan**) to have a higher priority than the slave (**gooby**).

Modify the daemon line in /etc/init.d/keepalived so it actually works:



|  |  |
| --- | --- |
| 1 | daemon /usr/local/sbin/keepalived ${KEEPALIVED\_OPTIONS} |

Ensure that net.ipv4.ip\_nonlocal\_bind is set:



|  |  |
| --- | --- |
| 1  2  3  4 | # vi /etc/sysctl.conf  ...  net.ipv4.ip\_nonlocal\_bind = 1  # sysctl -p |

Before starting keepalived, ensure that HAProxy is running on both nodes. Start keepalived:



|  |  |
| --- | --- |
| 1 | # service keepalived start |

You should see the VIP as a subinterface of eth0 (or whatever you’ve defined inkeepalived.conf) on the master (**dolan**):



|  |  |
| --- | --- |
| 1  2 | # ip addr | grep '172\.16\.18\.199'  inet 172.16.18.199/32 scope global eth0 |

Shutdown HAProxy on the master:



|  |  |
| --- | --- |
| 1 | # pkill -f haproxy |

You should see the IP address failover to the slave:



|  |  |
| --- | --- |
| 1  2 | # ip addr | grep '172\.16\.18\.199'  inet 172.16.18.199/32 scope global eth0 |

Start HAProxy again on the master, and hit the Tomcat examples via the VIP address. Your cluster should be working as expected. Verify the cluster by failing the VIP between nodes by starting/stopping HAProxy, and ensure that your session-replicated Tomcat application is still working, and that your session ID doesn’t change.

## Adding SSL Support

Our home-brewed load balancer is working, but it lacks one important thing for any decent web-facing application: SSL Support. Only the latest development releases of HAProxy support SSL natively, so I’ll use stunnel to provide SSL termination, listening on our VIP on port 443, before hand off to HAProxy listening on our VIP on port 80. Again, as with HAProxy, stunnelwill need to be configured and running on both nodes, although only one node will be receiving requests at any one time.

Download stunnel from [http://www.stunnel.org](http://www.stunnel.org/). Compile and install:



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | # cd /usr/local/src  # tar xzf stunnel-4.54.tar.gz  # cd stunnel-4.54  # ./configure  # make  # make install |

Create the configuration directory:



|  |  |
| --- | --- |
| 1 | # mkdir -p /etc/stunnel/ssl |

Create a self-signed certificate for testing:



|  |  |
| --- | --- |
| 1  2  3  4 | # openssl genrsa -des3 -out /etc/stunnel/ssl/stunnel.key 2048  # openssl rsa -in /etc/stunnel/ssl/stunnel.key -out /etc/stunnel/ssl/stunnel.key-empty  # openssl req -new -key /etc/stunnel/ssl/stunnel.key-empty -out /etc/stunnel/ssl/stunnel.csr  # openssl x509 -req -days 3650 -in /etc/stunnel/ssl/stunnel.csr -signkey /etc/stunnel/ssl/stunnel.key-empty -out /etc/stunnel/ssl/stunnel.pem |

Now for the important bit – stunnel expects the key and the certificate to be in the same file. Let’s do that:



|  |  |
| --- | --- |
| 1 | # cat /etc/stunnel/ssl/stunnel.key-empty /etc/stunnel/ssl/stunnel.pem > /etc/stunnel/ssl/stunnel.crt |

As stunnel will have access to our SSL private key, I decided to chroot it and drop privileges down to user/group stunnel/stunnel. So, create the chroot and the appropriate user/group:



|  |  |
| --- | --- |
| 1  2  3  4 | # mkdir -p /var/chroot/stunnel  # groupadd -g 443 stunnel  # useradd -M -d /dev/null -s /sbin/nologin -g stunnel -u 443 stunnel  # chown stunnel:stunnel /var/chroot/stunnel |

Configure stunnel appropriately:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | # vi /etc/stunnel/stunnel.conf  cert = /etc/stunnel/ssl/stunnel.crt  chroot = /var/chroot/stunnel  setuid = stunnel  setgid = stunnel  pid = /stunnel.pid  [https]  accept = 172.16.18.199:443  connect = 172.16.18.199:80 |

As you can see, we accept traffic on VIP:443, terminate SSL, and handoff to HAProxy on VIP:80.pid must be set relative to the chroot.

Start stunnel manually on both nodes:



|  |  |
| --- | --- |
| 1 | # stunnel /etc/stunnel/stunnel.conf |

Hit the tomcat examples (https://VIP/examples) and verify that all is well. You’ll obviously need to accept any browser-based warnings about the self-signed certificate.

Modify /etc/keepalived/keepalived.conf to check that both HAProxy AND stunnel are running:



|  |  |
| --- | --- |
| 1  2  3  4 | # vi /etc/keepalived/keepalived.conf  ...  script "killall -0 haproxy && killall -0 stunnel"  ... |

Now, failover will occur should either component fail.

Restart keepalived:



|  |  |
| --- | --- |
| 1 | # service keepalived restart |

Test killing off either haproxy or stunnel on a node, and ensure failover occurs. Double check that SSL and the examples are still working.

We are almost there. Our load-balancer is now SSL-enabled, and the Tomcat backend is session-aware and happily clustered and replicating session data.

Time for a little cleanup.

## Cleanup and Final Tasks

First, I’ll configure rsyslog (we’re using a RHEL 6.x derivative so rsyslog is being used).

On both servers, edit /etc/rsyslog.conf and add the following to the syslog rules:



|  |  |
| --- | --- |
| 1 | local0.\*                                                /var/log/haproxy.log |

Note – we’re using local0 as our facility – this must match whatever you’ve defined inhaproxy.conf. Also, enable UDP reception by uncommenting the following lines:



|  |  |
| --- | --- |
| 1  2  3 | # Provides UDP syslog reception  $ModLoad imudp  $UDPServerRun 514 |

Restart rsyslog:



|  |  |
| --- | --- |
| 1 | # service rsyslog restart |

Next, restart your HAProxy instances (watching keepalived failing the VIP around), and watch/var/log/haproxy.log. Make some requests to the load balancer and you should see entries such as:



|  |  |
| --- | --- |
| 1  2 | Feb 11 05:20:12 localhost haproxy[16874]: 172.16.18.199:39421 [11/Feb/2013:05:20:12.190] http-in servers/node1 1/0/0/4/5 200 540 - - ---- 1/1/0/1/0 0/0 "GET /examples/jsp/jsp2/simpletag/hello.jsp HTTP/1.1"  Feb 11 05:20:12 localhost haproxy[16874]: 172.16.18.199:39421 [11/Feb/2013:05:20:12.196] http-in servers/node2 480/0/0/4/484 200 540 - - ---- 1/1/0/1/0 0/0 "GET /examples/jsp/jsp2/simpletag/hello.jsp HTTP/1.1 |

As you can see, our requests are hitting the two backend nodes (defined in our haproxy.confas **node1** and **node2**) round-robin, which is what we expect.

Next, install the /etc/init.d script for HAProxy. The sed commands are specific to my installation and you may not need them:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | # cp /usr/local/src/haproxy-1.4.22/examples/haproxy.init /etc/init.d/haproxy  # chmod +x /etc/init.d/haproxy  # chkconfig haproxy on  # sed -i 's!usr/sbin!usr/local/sbin!g' /etc/init.d/haproxy  # sed -i 's/\.cfg/\.conf/g' /etc/init.d/haproxy  # pkill -f haproxy  # service haproxy start |

From this point, haproxy should be managed via the service command. Install the/etc/init.d script for stunnel:



|  |  |
| --- | --- |
| 1  2  3  4 | # cp /usr/local/src/stunnel-4.54/tools/stunnel.init /etc/init.d/stunnel  # chmod +x /etc/init.d/stunnel  # chkconfig stunnel on  # echo "ENABLED=1" > /etc/default/stunnel |

And (re)start it:



|  |  |
| --- | --- |
| 1  2 | # pkill -f stunnel  # service stunnel start |

Again, stunnel should now also be managed via the service command.

The HA load balancer solution is complete!

## Conclusion

This article has walked through building a load balancer that slowly grows in complexity, into a highly functional SSL-enabled HA load-balancer pair, sitting in front of a clustered Tomcat installation.

HAProxy, stunnel, keepalived and Tomcat all support much more functionality than I can cram into a single article, but I’ll be sure to write about any interesting nuggets over at[tokiwinter.com](http://www.tokiwinter.com/).

### Related posts:

1. [How to Load Balance Tomcat with Apache HTTPD and mod\_jk](http://www.tokiwinter.com/load-balance-tomcat-apache-httpd-mod_jk/)
2. [Apache httpd: How to Use htpasswd to Password Protect Areas of your Site](http://www.tokiwinter.com/apache-httpd-htpasswd-password-protect-site/)
3. [Apache HTTPD mod\_rewrite: one RewriteCond to many RewriteRules](http://www.tokiwinter.com/apache-httpd-mod_rewrite-one-rewritecond-many-rewriterules/)

This entry was posted in [UNIX & Linux How Tos and Tutorials](http://www.tokiwinter.com/category/unix-linux-howtos-tutorials/) and tagged [clustering](http://www.tokiwinter.com/tag/clustering/), [HAProxy](http://www.tokiwinter.com/tag/haproxy/), [High-Availability](http://www.tokiwinter.com/tag/high-availability/), [keepalived](http://www.tokiwinter.com/tag/keepalived/), [Load balancing](http://www.tokiwinter.com/tag/load-balancing/), [stunnel](http://www.tokiwinter.com/tag/stunnel/), [Tomcat](http://www.tokiwinter.com/tag/tomcat/) on [February 15, 2013](http://www.tokiwinter.com/highly-available-load-balancing-of-apache-tomcat-using-haproxy-stunnel-and-keepalived/).