系统负载能力浅析

互联网时代，高并发是一个老生常谈的话题。无论对于一个web站点还是app应用，高峰时能承载的并发请求都是衡量一个系统性能的关键标志。像阿里双十一顶住了上亿的峰值请求、订单也确实体现了阿里的技术水平（当然有钱也是一个原因）。

那么，何为系统负载能力？怎么衡量？相关因素有哪些？又如何优化呢？

一. 衡量指标

用什么来衡量一个系统的负载能力呢？有一个概念叫做每秒请求数（Requests per second），指的是每秒能够成功处理请求的数目。比如说，你可以配置tomcat服务器的maxConnection为无限大，但是受限于服务器系统或者硬件限制，很多请求是不会在一定的时间内得到响应的，这并不作为一个成功的请求，其中成功得到响应的请求数即为每秒请求数，反应出系统的负载能力。

通常的，对于一个系统，增加并发用户数量时每秒请求数量也会增加。然而，我们最终会达到这样一个点，此时并发用户数量开始“压倒”服务器。如果继续增加并发用户数量，每秒请求数量开始下降，而反应时间则会增加。这个并发用户数量开始“压倒”服务器的临界点非常重要，此时的并发用户数量可以认为是当前系统的最大负载能力。

二. 相关因素

一般的，和系统并发访问量相关的几个因素如下：

- 带宽
- 硬件配置
- 系统配置
- 应用服务器配置
- 程序逻辑

其中，带宽和硬件配置是决定系统负载能力的决定性因素。这些只能依靠扩展和升级提高。我们需要重点关注的是在一定带宽和硬件配置的基础上，怎么使系统的负载能力达到最大。

2.1 带宽

毋庸置疑，带宽是决定系统负载能力的一个至关重要的因素，就好比水管一样，细的水管同一时间通过的水量自然就少（这个比喻解释带宽可能不是特别合适）。一个系统的带宽首先就决定了这个系统的负载能力，其单位为Mbps,表示数据的发送速度。

2.2 硬件配置

系统部署所在的服务器的硬件决定了一个系统的最大负载能力，也是上限。一般说来，以下几个配置起着关键作用：

- cpu频率/核数：cpu频率关系着cpu的运算速度，核数则影响线程调度、资源分配的效率。
- 内存大小以及速度：内存越大，那么可以在内存中运行的数据也就越大，速度自然而然就快；内存的速度从原来的几百hz到现在几千hz，决定了数据读取存储的速度。
- 硬盘速度：传统的硬盘是使用磁头进行寻址的，io速度比较慢，使用了SSD的硬盘，其寻址速度大大较快。

很多系统的架构设计、系统优化，最终都会加上这么一句：使用ssd存储解决了这些问题。

可见，硬件配置是决定一个系统的负载能力的最关键因素。

2.3 系统配置

一般来说，目前后端系统都是部署在Linux主机上的。所以抛开win系列不谈，对于Linux系统来说一般有以下配置关系着系统的负载能力。

- 文件描述符数限制：Linux中所有东西都是文件，一个socket就对应着一个文件描述符，因此系统配置的最大打开文件数以及单个进程能够打开的最大文件数就决定了socket的数目上限。
- 进程/线程数限制: 对于apache使用的prefork等多进程模式，其负载能力由进程数目所限制。对tomcat多线程模式则由线程数所限制。
- tcp内核参数：网络应用的底层自然离不开tcp/ip，Linux内核有一些与此相关的配置也决定了系统的负载能力。

2.3.1 文件描述符数限制

- 系统最大打开文件描述符数：/proc/sys/fs/file-max中保存了这个数目,修改此值

  临时性
  ```
  echo 1000000 > /proc/sys/fs/file-max
  ```
  永久性：在/etc/sysctl.conf中设置
  ```
  fs.file-max = 1000000
  ```

- 进程最大打开文件描述符数：这个是配单个进程能够打开的最大文件数目。可以通过ulimit -n查看/修改。如果想要永久修改，则需要修改/etc/security/limits.conf中的nofile。

通过读取/proc/sys/fs/file-nr可以看到当前使用的文件描述符总数。另外，对于文件描述符的配置，需要注意以下几点：

- 所有进程打开的文件描述符数不能超过/proc/sys/fs/file-max
- 单个进程打开的文件描述符数不能超过user limit中nofile的soft limit
- nofile的soft limit不能超过其hard limit
- nofile的hard limit不能超过/proc/sys/fs/nr_open

2.3.2 进程/线程数限制

- 进程数限制：ulimit -u可以查看/修改单个用户能够打开的最大进程数。/etc/security/limits.conf中的noproc则是系统的最大进程数。
- 线程数限制
  - 可以通过/proc/sys/kernel/threads-max查看系统总共可以打开的最大线程数。
  - 单个进程的最大线程数和PTHREAD_THREADS_MAX有关，此限制可以在/usr/include/bits/local_lim.h中查看,但是如果想要修改的话，需要重新编译。
  - 这里需要提到一点的是，Linux内核2.4的线程实现方式为linux threads，是轻量级进程，都会首先创建一个管理线程，线程数目的大小是受PTHREAD_THREADS_MAX影响的。但Linux2.6内核的线程实现方式为NPTL,是一个改进的LWP实现，最大一个区别就是，线程公用进程的pid（tgid），线程数目大小只受制于资源。
  - 线程数的大小还受线程栈大小的制约：使用ulimit -s可以查看/修改线程栈的大小，即每开启一个新的线程需要分配给此线程的一部分内存。减小此值可以增加可以打开的线程数目。

2.3.3 tcp内核参数

在一台服务器CPU和内存资源额定有限的情况下，最大的压榨服务器的性能，是最终的目的。在节省成本的情况下，可以考虑修改Linux的内核TCP/IP参数，来最大的压榨服务器的性能。如果通过修改内核参数也无法解决的负载问题，也只能考虑升级服务器了，这是硬件所限，没有办法的事。

```
netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a]}'
```

使用上面的命令，可以得到当前系统的各个状态的网络连接的数目。如下：

```
LAST_ACK 14
SYN_RECV 348
ESTABLISHED 70
FIN_WAIT1 229
FIN_WAIT2 30
CLOSING 33
```

TIME_WAIT 18122

这里，TIME_WAIT的连接数是需要注意的一点。此值过高会占用大量连接，影响系统的负载能力。需要调整参数，以尽快的释放time_wait连接。

一般tcp相关的内核参数在/etc/sysctl.conf文件中。为了能够尽快释放time_wait状态的连接，可以做以下配置：

- net.ipv4.tcp_syncookies = 1 //表示开启SYN Cookies。当出现SYN等待队列溢出时，启用cookies来处理，可防范少量SYN攻击，默认为0，表示关闭；
- net.ipv4.tcp_tw_reuse = 1 //表示开启重用。允许将TIME-WAIT sockets重新用于新的TCP连接，默认为0，表示关闭；
- net.ipv4.tcp_tw_recycle = 1 //表示开启TCP连接中TIME-WAIT sockets的快速回收，默认为0，表示关闭；
- net.ipv4.tcp_fin_timeout = 30 //修改系統默认的 TIMEOUT 时间。

这里需要注意的一点就是当打开了tcp_tw_reccycle，就会检查时间戳，移动环境下的发来的包的时间戳有些时候是乱跳的，会把带了“倒退”的时间戳的包当作是“recycle的tw连接的重传数据，不是新的请求”，于是丢掉不回包，造成大量丢包。

此外，还可以通过优化tcp/ip的可使用端口的范围，进一步提升负载能力。，如下：

- net.ipv4.tcp_keepalive_time = 1200 //表示当keepalive起用的时候，TCP发送keepalive消息的频度。缺省是2小时，改为20分钟。
- net.ipv4.ip_local_port_range = 10000 65000 //表示用于向外连接的端口范围。缺省情况下很小：32768到61000，改为10000到65000。（注意：这里不要将最低值设的太低，否则可能会占用掉正常的端口！）
- net.ipv4.tcp_max_syn_backlog = 8192 //表示SYN队列的长度，默认为1024，加大队列长度为8192，可以容纳更多等待连接的网络连接数。
- net.ipv4.tcp_max_tw_buckets = 5000 //表示系统同时保持TIME_WAIT的最大数量，如果超过这个数字，TIME_WAIT将立刻被清除并打印警告信息。默认为180000，改为5000。对于Apache、Nginx等服务器，上几行的参数可以很好地减少TIME_WAIT套接字数量，但是对于Squid，效果却不大。此项参数可以控制TIME_WAIT的最大数量，避免Squid服务器被大量的TIME_WAIT拖死。

2.4 应用服务器配置

说到应用服务器配置，这里需要提到应用服务器的几种工作模式,也叫并发策略。

- multi process:多进程方式，一个进程处理一个请求。
- prefork：类似于多进程的方式，但是会预先fork出一些进程供后续使用，是一种进程池的理念。
- worker：一个线程对应一个请求，相比多进程的方式，消耗资源变少，但同时一个线程的崩溃会引起整个进程的崩溃，稳定性不如多进程。
- master/worker：采用的是非阻塞IO的方式，只有两种进程：worker和master,master负责worker进程的创建、管理等，worker进程采用基于事件驱动的多路复用IO处理请求。mater进程只需要一个，woker进程根据cpu核数设置数目。

前三者是传统应用服务器apache和tomcat采用的方式，最后一种是nginx采用的方式。当然这里需要注意的是应用服务器和nginx这种做反向代理服务器（暂且忽略nginx+cgi做应用服务器的功能）的区别。应用服务器是需要处理应用逻辑的，有时候是耗cup资源的；而反向代理主要用作IO，是IO密集型的应用。使用事件驱动的这种网络模型，比较适合IO密集型应用，而并不适合CPU密集型应用。对于后者，多进程/线程则是一个更好地选择。

当然，由于nginx采用的基于事件驱动的多路IO复用的模型，其作为反向代理服务器时，可支持的并发是非常大的。淘宝tengine团队曾有一个测试结果是“24G内存机器上，处理并发请求可达200万”。

2.4.1 nginx/tengine

ngixn是目前使用最广泛的反向代理软件，而tengine是阿里开源的一个加强版nginx,其基本实现了nginx收费版本的一些功能，如：主动健康检查、session sticky等。对于nginx的配置，需要注意的有这么几点：

- worker数目要和cpu（核）的数目相适应
- keepalive timout要设置适当
- worker_rlimit_nofile最大文件描述符要增大
- upstream可以使用http 1.1的keepalive

典型配置可见：https://github.com/superhj1987/awesome-config/blob/master/nginx/nginx.conf

2.4.2 tomcat

tomcat的关键配置总体上有两大块：jvm参数配置和connector参数配置。

- jvm参数配置：
    - 堆的最小值：Xms
    - 堆的最大值：Xmx
    - 新生代大小: Xmn
    - 永久代大小: XX:PermSize：
    - 永久代最大大小: XX:MaxPermSize：
    - 栈大小：-Xss或-XX:ThreadStackSize

  这里对于栈大小有一点需要注意的是：在Linux x64上ThreadStackSize的默认值就是1024KB，给Java线程创建栈会用这个参数指定的大小。如果把-Xss或者-XX:ThreadStackSize设为0，就是使用“系统默认值”。而在Linux x64上HotSpot VM给Java栈定义的“系统默认”大小也是1MB。所以普通Java线程的默认栈大小怎样都是1MB。这里有一个需要注意的地方就是java的栈大小和之前提到过的操作系统的操作系统栈大小（ulimit -s）：这个配置只影响进程的初始线程；后续用pthread_create创建的线程都可以指定栈大小。HotSpot VM为了能精确控制Java线程的栈大小，特意不使用进程的初始线程（primordial thread）作为Java线程。

  其他还要根据业务场景，选择使用那种垃圾回收器，回收的策略。另外，当需要保留GC信息时，也需要做一些设置。

  典型配置可见：https://github.com/superhj1987/awesome-config/blob/master/tomcat/java_opts.conf

- connector参数配置
    - protocol: 有三个选项：bio；nio；apr。建议使用apr选项，性能为最高。
    - connectionTimeout：连接的超时时间
    - maxThreads：最大线程数，此值限制了bio的最大连接数
    - minSpareThreads: 最大空闲线程数
    - acceptCount：可以接受的最大请求数目（未能得到处理的请求排队）
    - maxConnection: 使用nio或者apr时，最大连接数受此值影响。

  典型配置可见：https://github.com/superhj1987/awesome-config/blob/master/tomcat/connector.conf

  一般的当一个进程有500个线程在跑的话，那性能已经是很低很低了。Tomcat默认配置的最大请求数是150。当某个应用拥有250个以上并发的时候，应考虑应用服务器的集群。

  另外，并非是无限调大maxTreads和maxConnection就能无限调高并发能力的。线程越多，那么cpu花费在线程调度上的时间越多，同时，内存消耗也就越大，那么就极大影响处理用户的请求。受限于硬件资源，并发值是需要设置合适的值的。

对于tomcat这里有一个争论就是：***使用大内存tomcat好还是多个小的tomcat集群好？***（针对64位服务器以及tomcat来说）

其实，这个要根据业务场景区别对待的。通常，大内存tomcat有以下问题：

- 一旦发生full gc，那么会非常耗时
- 一旦gc，dump出的堆快照太大，无法分析

因此，如果可以保证一定程度上程序的对象大部分都是朝生夕死的，老年代不会发生gc,那么使用大内存tomcat也是可以的。但是在伸缩性和高可用却比不上使用小内存（相对来说）tomcat集群。

使用小内存tomcat集群则有以下优势：

- 可以根据系统的负载调整tc的数量，以达到资源的最大利用率，
- 可以防止单点故障。

2.4.3 数据库

mysql是目前最常用的关系型数据库，支持复杂的查询。但是其负载能力一般。很多时候一个系统的瓶颈就发生在mysql这一点，当然有时候也和sql语句的效率有关。比如，牵扯到联表的查询一般说来效率是不会太高的。

当数据量单表突破千万甚至百万时（和具体的数据有关），查询和插入效率都会受到影响。此时，需要对mysql数据库进行优化，一种常见的方案就是分表：

- 垂直分表：在列维度的拆分
- 水平分表：行维度的拆分

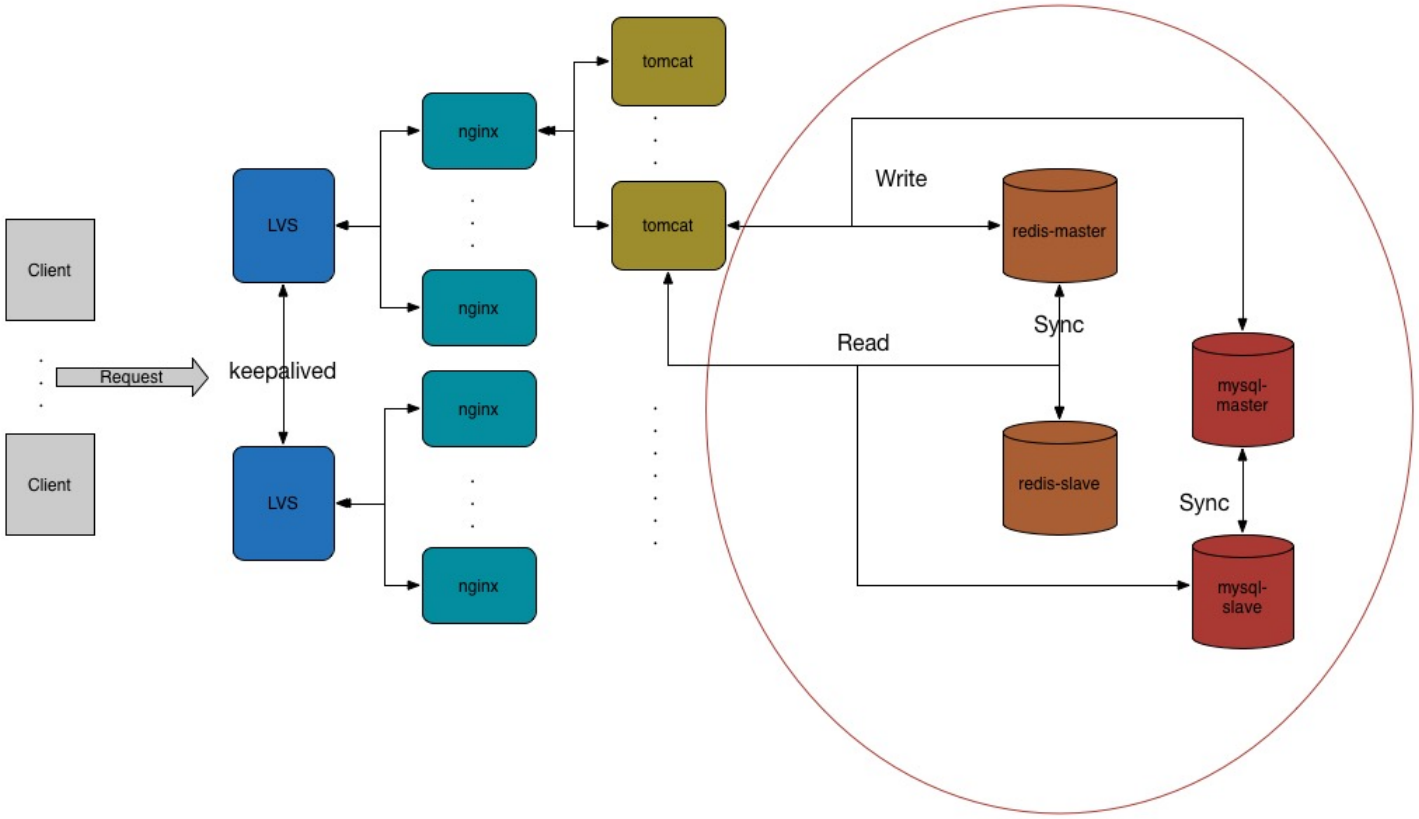
对于系统中并发很高并且访问很频繁的数据，会放到缓存数据库中，以隔离对mysql的访问,防止mysql崩溃。

redis是目前用的比较多的缓存数据库（当然，也有直接把redis当做数据库使用的）。

此外，对于数据库，可以使用读写分离的方式提高性能，尤其是对那种读频率远大于写频率的业务场景。这里采用master/slave的方式实现读写分离，前面用程序控制或者加一个proxy层。

三. 一般架构

一般的web应用架构如下图所示：lvs+nginx+tomcat+mysql+redis



本文对LVS没有做相关说明，后续会补充进来。