网站加速--服务器编写篇

--提升性能的同时为你节约 10 倍以上成本
From: http://blog.sina.com.cn/iyangjian

一，如何节约 CPU
二，怎样使用内存
三，减少磁盘 I/O
四，优化你的网卡
五，调整内核参数
六，衡量 Web Server 的性能指标
七，NBA js 直播的发展历程
八，新浪财经实时行情系统的历史遗留问题 (7 byte = 10.68w RMB/year)

一，如何节约 **CPU**

1,选择一个好的 I/O 模型(epoll, kqueue)
3 年前，我们还关心 c10k 问题，随着硬件性能的提升，那已经不成问题，但如果想让 PIII 900 服务器支撑 5w+ connections,还是需要些能耐的。

epoll 最擅长的事情是监视大量闲散连接，批量返回可用描述符,这让单机支撑百万 connections 成为可能。linux 2.6 以上开始支持 epoll，freebsd 上相应的有 kqueue，不过我个人偏爱 linux，不太关心 kqueue。

边缘触发 ET 和 水平触发 LT 的选择：
早期的文档说 ET 很高效，但是有些冒进。但事实上 LT 使用过程中，我苦恼了将近一个月有余，一不留神 CPU 利用率 99%了，可能是我没处理好。后来 zhongying 同学帮忙把驱动模式改成了 ET 模式，ET 既高效又稳定。

简单地说，如果你有数据过来了，不去取 LT 会一直骚扰你，提醒你去取，而 ET 就告诉你一次，爱取不取，除非有新数据到来，否则不再提醒。

重点说下 ET,非阻塞模式，
man 手册说，如果 ET 提示你有数据可读的时候，你应该连续的读一直读到返回 EAGAIN or EWOULDBLOCK 为止，但是在具体实现中，我并没有这样做，而是根据我的应用做了优化。因为现在操作系统绝大多数实现都是最大传输单元值为 1500。 MTU:1500 - ipheader:20 - tcpheader:20 = 1460 byte .
HTTP header,不带 cookie 的话一般只有 500+ byte。留 512 给 uri，也基本够用，还有节余。
更正:如果只读前 1460 个字节的 header 的话，99.999%的用户都能正常访问，但是有一两个用户是通过某 http proxy 来上网，这种用户请求的特征是会带 2k 以上的垃圾信息过来,类似于 cookie,但又不是，重要的 http header 被放在了最后,导致我认为是非法请求,所以现在来看,为了避免被投诉,还是建议读取到"\r\n\r\n"作为结束,性能的影响也是比较有限的. (最后修改时间 2010.05.11)

如果请求的 header 恰巧比这大是 2050 字节呢？
会有两种情况发生：1，数据紧挨着同时到达，一次 read 就搞定。 2，分两个 ethernet frame 先后到达有一定时间间隔。
我的方法是，用一个比较大的 buffer 比如 1M 去读 header，如果你很确信你的服务对象请求比

1460 小，读一次就行。如果请求会很大分几个 ethernet frame 先后到达，也就是恰巧你刚刚 read 过，它又来一个新数据包，ET 会再次返回，再处理下就是了。

更正: 在追踪通过 http proxy 上网的用户问题中发现,这个方法并不严谨:1, 2050 字节有可能会分两个,以及两个以上的 ethernet frame 到达,虽然我测试还没遇到 3 个的情况，但是现在我断定它是会存在的. 2,即使比 1460 字节小的数据,也不敢保证装载在一个 ethernet frame 里,仍然有可能分多次到达. 所以用 1M 的 buffer 去读，返回数据比 1460 小，并不代表数据已经读完，只能说明，本次读完了，下次再有数据到达时会 ET 提示你.测试中发现，对于大于 1460 字节的头的读区,如果分多次到达，第一次返回为 1460 字节的概率大一些而已. 测试中甚至还遇到了另外一个临界情况,就是用 1M 的 buffer 刚读完数据,立刻再接着读,有可能会读到数据,这个数据应该是紧接着到达,很小的时间间隔,如果你不去读,我认为 ET 应该会再次返回(仅仅出现过一次,未能验证,有时间再求证). (最后修改时间 2010.05.11)

顺便再说下写数据，一般一次可以 write 十几 K 数据到内核缓冲区。
所以对于很多小的数据文件服务来说，是没有必要另外为每个 connections 分配发送缓冲区。
只有当一次发送不完时候才分配一块内存，将数据暂存，待下次返回可写时发送。
这样避免了一次内存 copy，而且节约了内存。

选择了 epoll 并不代表就就拥有了一个好的 I/O 模型，用的不好，你还赶不上 select,这是实话。
epoll 的问题我就说这么多，关于描述符管理方面的细节请参见我早期的一个帖子，epoll 模型的使用及其描述符耗尽问题的探讨 大概讨论了 18 页，我刚才把解决方法放在第一个帖子里了。如果你对 epoll 有感兴趣，我这有 一个简单的基于 epoll 的 web server 例子 。

另外你要使用多线程，还是多进程，这要看你更熟悉哪个，各有好处。
多进程模式，单个进程 crash 了，不影响其他进程，而且可以为每个 worker 分别帮定不同的 cpu,让某些 cpu 单独空出来处理中断和系统事物。多线程，共享数据方便，占用资源更少。进程或线程的个数，应该固定在（cpu 核数-1）～ 2 倍 cpu 核数间为宜，太多了时间片轮转时会频繁切换，少了，达不到多核并发处理的效果。

还有如何 accept 也是一门学问，没有最好，只有更适用，你需要做很多实验，确定对自己最高效的方式。有了一个好的 I/O 框架，你的效率想低也不容易,这是程序实现的大局。

关于更多网络 I/O 模型的讨论请见 <Scalable Network Programming > 中文版。
另外，必须强调的是,代码和结构应该简洁高效,一定要具体问题具体分析，没什么法则是万能的，要根据你的服务量身定做。

2,关闭不必要的标准输入和标准输出
```
close(0);  //stdin
close(1);  //stdout
```
如果你不小心，有了 printf 输出调试信息，这绝对是一个性能杀手。
一个高性能的服务器不出错是不应该有任何输出的，免得耽误干活。
这样做，至少能为你节约两个描述符资源。

3，避免用锁 (i++ or ++i)
多线程编程用锁是普遍现象，貌似已经成为习惯。
但各线程最好是独立的，不需要同步机制的。
锁会消耗资源，而且造成排队，甚至死锁，尽量想办法避免。
非用不可时候，比如，实时统计各线程的负载情况，多个线程要对全局变量进行写操作。
请用 ++i ，因为它是一个原子操作。

更正: pinggao， 同学对++i 的原子性提出了质疑，并做了个测试程序，经过讨论得出结论：++i 在多线程的程序中，在单核环境下，不一定是原子的，在多核环境下肯定不是原 子的，我把这个当作原子

操作用在了统计中，所以我得出的统计值是要比实际处理能力小很多的。。。亏大了。。等有新的数据再更正过来。。想想犯这个基本错误 的原因，当初我们大学教授在讲＋＋i 是原子操作的时候，距离现在计算机环境发生了巨大变化。 ＋＋i 原子性讨论 (最后修改时间 2010.10.02)

4,减少系统调用

系统调用是很耗的，因为它通常需要钻进内核再钻出来。
我们应该避免用户空间和内核空间的切换。
比如我要为每个请求打个时间戳，以计算超时，我完全可以在返回一批可用描述符前只调用一次 time(),而不用每个请求都调用一次。 time()只精确到秒，一批请求处理都是毫秒级，所以也没必要那么做，再说了，计算超时误差那么一秒有什么影响吗？

5, Connection: close vs Keep-Alive ?

谈 httpd 实现，就不能不提长连接 Keep-Alive 。
Keep-Alive 是 http 1.1 中加入的，现在的浏览器 99。99%应该都是支持 Keep-Alive 的。

先说下什么是 Keep-Alive:
这是基于 tcp 的 connections 说的，也就是一个描述符(fd)，它并不代表独立占用一个进程或线程。一个线程用非阻塞模式可以保持成千上万个长连接。

先说一个完整的 HTTP 1.0 的请求和响应:
建立 tcp 连接 (syn; ack, syn2; ack2; 三个分组握手完成)
请求
响应
关闭连接 (fin; ack; fin2; ack2 四个分组关闭连接)

再说 HTTP 1.1 的请求和响应:
建立 tcp 连接 (syn; ack, syn2; ack2; 三个分组握手完成)
请求
响应
…
…

请求
响应
关闭连接 (fin; ack; fin2; ack2 四个分组关闭连接)

如果请求和响应都只有一个分组，那么 HTTP 1.0 至少要传输 11 个分组(补充：请求和响应数据还各需要一个 ack 确认)，才拿到一个分组的数据。而长连接可以更充分的利用这个已经建立的连接，避免的频繁的建立和关闭连接，减少网络拥塞。

我做过一个测试，在 2cpu*4core 服务器上，不停的 accept，然后不做处理，直接 close 掉。一秒最多可以 accept 7w/s，这是极限。那么我要是想每秒处理 10w 以上的 http 请求该怎么办呢？
目前唯一的也是最好的选择，就是保持长连接。
比如我们 NBA JS 直播页面，刚打开就会向我的 js 服务器发出 6 个 http 请求，而且随后平均每 10 秒会产生两个请求。再比如，我们很多页面都会嵌几个静态池的图片，如果每个请求都是独立的（建立连接然后关闭），那对资源绝对是个浪费。

长连接是个好东西，但是选择 Keep-Alive 必须根据你的应用决定。比如 NBA JS 直播,我肯定 10 秒内会产生一个请求，所以超时设置为 15 秒，15 秒还没活动，估计是去打酱油了，资源就得被我回收。超时设置过长，光连接都能把你的服务器堆死。

为什么有些 apache 服务器，负载很高，把 Keep-Alive 关掉负载就减轻了呢？
apache 有两种工作模式，prefork 和 worker。apache 1.x 只有，prefork。
prefork 比较典型，就是个进程池，每次创建一批进程,还有 apache 是基于 select 实现的。在用户

不是太多的时候，长连接还是很有用的，可以 节约分组，提升响应速度，但是一旦超出某个平衡点，由于为了保持很多长连接，创建了太多的进程，导致系统不堪重负，内存不够了，开始换入换出，cpu 也被 很多进程吃光了,load 上去了。这种情况下，对 apache 来说，每次请求重新建立连接要比保持这么多长连接和进程更划算。

6,预处理 (预压缩，预取 lastmodify,mimetype)

预处理,原则就是，能预先知道的结果，我们绝不计算第二次。

预压缩：我们在两三年前就开始使用预压缩技术，以节约 CPU，伟大的微软公司在现在的 IIS 7 中也开始使用了。所谓的预压缩就是，从数据源头提供的就是预先压缩好的数据，IDC 同步传输中是压缩状态，直到最后 web server 输出都是压缩状态，最终被用户浏览器端自动解压。

预取 lastmodify： 文件的 lastmodify 时间，如果不更新，我们不应该取第二次，别忘记了 fsat 这个系统调用是很耗的。

预取 mimetype： mimetype,如果你的文件类型不超过 256 种，一个字节就可以标识它，然后用数组下标直接输出，而且不是看到一个 js 文件，然后 strcmp()了近百种后缀名后，才知道应该输出 Content-Type: application/x-javascript，而且这种方法会随文件类型增加而耗费更多 cpu 资源。当然也可以写个 hash 函数来做这事，那也至少需要一次函数调用，做些求值运算，和分配比实际数据大几倍的 hash 表。

如何更好的使用 cpu 一级缓存
数据分解
CPU 硬亲和力的设置

待补充。。。。

二，怎样使用内存

1，避免内存 copy (strcpy,memcpy)

虽然内存速度很快，但是执行频率比较高的核心部分能避免 copy 的就尽量别使用。如果必须要 copy，尽量使用 memcpy 替代 sprintf,strcpy，因为它不关心你是否遇到'\0'; 内存拷贝和 http 响应又涉及到字符串长度计算。如果能预先知道这个长度最好用中间变量保留，增加多少直接加上去，不要用 strlen()去计算，因为它 会数数直到遇见'\0'。能用 sizeof()的地方就不要用 strlen,因为它是个运算符，在预编的时被替换为具体数字，而非执行时计算。

2，避免内核空间和用户进程空间内存 copy (sendfile, splice and tee)

sendfile: 它的威力在于，它为大家提供了一种访问当前不断膨胀的 Linux 网络堆栈的机制。这种机制叫做“零拷贝(zero-copy)”,这种机制可以把“传输控制协议（TCP）”框架直接的从主机存储器中传送到网卡的缓存块（network card buffers）中去，避免了两次上下文切换。详细参见 <使用 sendfile()让数据传输得到最优化> 。据同事测试说固态硬盘 SSD 对于小文件的随机读效率很高，对于更新不是很频繁的图片服务，读却很多，每个文件都不是很大的话，sendfile+SSD 应该是绝配。

splice and tee: splice 背后的真正概念是暴露给用户空间的“随机内核缓冲区”的概念。“也就是说，splice 和 tee 运行在用户控制的内核缓冲区上，在这个缓冲区 中，splice 将来自任意文件描述符的数据传送到缓冲区中(或从缓冲区传送到文件描述符)，而 tee 将一个缓冲区中的数据复制到另一个缓冲区中。因此， 从一个很真实(而抽象)的意义上讲，splice 相当于内核缓冲区的 read/write，而 tee 相当于从内核缓冲区到另一个内核缓冲区的 memcpy。”。本人觉得这个技术用来做代理，很合适。因为数据可以直接从一个 soket 到另一个 soket，不需要经用户和内核空间的切换。这是 sendfile 不支持的。详细参见 <linux2.6.17 以上内核中的 splice and tee> ,具体实例请参见 man 2 tee，里面有个完整的程序。

3，如何清空一块内存(memset ?)

比如有一个 buffer[1024*1024],我们需要把它清空然后 strcat(很多情况下可以通过记录写的起始位置+memcpy 来代替)追加填充字符串。
其实我们没有必要用 memset(buffer,0x00,sizeof(buffer))来清空整个 buffer,
memset(buffer,0x00,1)就能达到目的。 我平时更喜欢用 buffer[0]='\0'; 来替代，省了一次函数调用的开销。

4，内存复用（有必要为每个响应分配内存 ?)

对于 NBA JS 服务来说，我们返回的都是压缩数据，99%都不超过 15k，基本一次 write 就全部出去了，是没有必要为每个响应分配内存的，公用一个 buffer 就够了。如果真的遇到大数据，我先 write 一次，剩下的再暂存在内存里，等待下次发送。

5，避免频繁动态申请/释放内存（malloc）

这个似乎不用多说，要想一个 Server 启动后成年累月的跑，就不应该频繁地去动态申请和释放内存。原因很简单一，避免内存泄露。二，避免碎片过多。三， 影响效率。一般来说，都是一次申请一大块内存，然后自己写内存分配算法。为 http 用户分配的缓冲区生命期的特点是，可以随着 fd 的关闭，而回收，避免漏 网。还有 Server 的编写者应该对自己设计的程序达到最高支撑量的时候所消耗的内存心中有数。

6，字节对齐

先看下面的两个结构体有什么不同:

```
struct A {
        short size;
        char *ptr;
        int left;
} a ;

struct B {
        char *ptr;
        short size;
        int left;
} b ;
```

仅仅是一个顺序的变化,结构体 B 顺序是合理的:
在 32bit linux 系统上，是按照 32/8bit=4byte 来对齐的， sizeof(a)=12 ,sizeof(b)=12 。
在 64bit linux 系统上，是按照 64/8bit=8byte 来对齐的， sizeof(a)=24 ,sizeof(b)=16 。
32bit 机上看到的 A 和 B 结果大小是一样的，但是如果把 int 改成 short 效果就不一样了。

如果我想强制以 2byte 对齐，可以这样:

```
#pragma pack(2)
struct A {
        short size;
        char *ptr;
        int left;
} a ;
#pragma pack()
```

注意 pack()里的参数，只能指定比本机支持的字节对齐标准小，而不能更大。

7，内存安全问题

先举个好玩的例子，不使用 a，而给 a 赋上值:

```
int main()
{
        char a[8];
```

```
        char b[8];
        memcpy(b,"1234567890\0",10);
        printf("a=%s\n",a);
        return 0;
}
```

程序输出 a=90 。
这就是典型的溢出，如果是空闲的内存，用点也就罢了，可是把别人地盘上的数据覆盖了，就不好了。
接收的用户数据一定要严格判断，确定不会越界，不是每个人都按规矩办事的，搞不好就挂了。

8，云风的内存管理理论（sd2c 大会所获 blog & ppt）
没有永远不变的原则
大原则变化的慢
没有一劳永逸的解决方案
内存访问很廉价但有代价
减少内存访问的次数是很有意义的
随机访问内存慢于顺序访问内存
请让数据物理上连续
集中内存访问优于分散访问
尽可能的将数据紧密的存放在一起
无关性内存访问优于相关性内存访问
请考虑并行的可能性、即使你的程序本身没有使用并行机制
控制周期性密集访问的数据大小
必要时采用时间换空间的方法
读内存快于写内存
代码也会占用内存，所以、保持代码的简洁

物理法则
晶体管的排列
批量回收内存
不释放内存，留给系统去做
list map vector (100 次调用产生 1 3 次内存分配和释放)
长用字符串做成 hash，使用指针访问
直接内存页处理控制

三，减少磁盘 **I/O**
这个其实就是通过尽可能的使用内存达到性能提高和 i/o 减少。从系统的读写 buffer 到用户空间自己的 cache，都是可以有效减少磁盘 i/o 的方法。 用户可以把数据暂存在自己的缓冲区里，批量读写大块数据。cache 的使用是很必要的，可以自己用共享内存的方法实现，也可以用现成的 BDB 来实现。欢迎访问我的公益站点 berkeleydb.net ,不过我不太欢迎那种问了问题就跑的人。BDB 默认的 cache 只有 256K，可以调大这个数字，也可以纯粹使用 Mem Only 方法。对于预先知道的结果，争取不从磁盘取第二次，这样磁盘基本就被解放出来了。BDB 取数据的速度每秒大概是 100w 条（2CPU*2Core Xeon(R) E5410 @ 2.33GHz 环境测试,单条数据几十字节），如果你想取得更高的性能建议自己写。

四，优化你的网卡
首先 ethtool ethx 看看你的外网出口是不是 Speed: 1000Mb/s 。
对于多核服务器，运行 top 命令，然后按一下 1，就能看到每个核的使用情况。如果发现 cpuid=0 的那颗使用率明显高于其他核，那就说明 id=0 的 cpu 将来也许会成为你的瓶颈。然后可以用 mpstat（非默认安装）命令查看系统中断分布，用 cat /proc/interrupts 网卡中断分布。

下面这个数据是我们已经做过优化了的服务器中断分布情况:

```
[yangjian2@D08043466 ~]$ mpstat -P ALL 1
Linux 2.6.18-53.el5PAE (D08043466)      12/15/2008
01:51:27 PM  CPU   %user   %nice    %sys %iowait    %irq   %soft  %steal
```

```
%idle    intr/s
01:51:28 PM  all    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
1836.00
01:51:28 PM    0    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
179.00
01:51:28 PM    1    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
198.00
01:51:28 PM    2    1.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
198.00
01:51:28 PM    3    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
346.00
01:51:28 PM    4    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
207.00
01:51:28 PM    5    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
167.00
01:51:28 PM    6    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
201.00
01:51:28 PM    7    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
339.00
```

没优化过的应该是这个样子:

```
yangjian2@xk-6-244-a8 ~]$ mpstat -P ALL 1
Linux 2.6.18-92.1.6.el5 (xk-6-244-a8.bta.net.cn)        12/15/2008
02:05:26 PM  CPU   %user   %nice    %sys %iowait    %irq   %soft  %steal
%idle    intr/s
02:05:27 PM  all    0.00    0.00    0.00    0.12    0.00    0.00    0.00   99.88
1593.00
02:05:27 PM    0    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
1590.00
02:05:27 PM    1    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
0.00
02:05:27 PM    2    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
2.00
02:05:27 PM    3    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
0.00
02:05:27 PM    4    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
0.00
02:05:27 PM    5    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
0.00
02:05:27 PM    6    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
0.00
02:05:27 PM    7    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
0.00
```

对于 32bit 的 centos5，mpstat -P ALL 1 表现跟第一种情况一样,分布比较平均，但是一但有了访问量，就可以看到差距。cat /proc/interrupts 看起来更直观些，很清楚的知道哪个网卡的中断在哪个 cpu 上处理。

其实，当你遇到网卡中断瓶颈的时候证明你的网站并发度已经相当高了，每秒三五万个请求还至于成为瓶颈。除非你的应用程序同时也在消耗 cpu0 的资源。对于 这种情况，建议使用多进程模式，每个进程用 sched_setaffinity 绑定特定的 cpu，把 cpu0 从用户事物中解放出来，专心处理系统事物，当然包括中断。这样你的极限应该能处理 20w+ http req/s (2CPU*4Core 服务器)。但是对于多线程

模式来说，我们就显得无能为力了，因为我们如果想使用多核，就没法不用 cpu0。目前的方法只有两个： 一，转化为多进程，然后进程内再使用多线程。二，让你的网卡中断分散在多个 cpu 上(目前只有硬件解决方案，感谢 xiaodong2 提供的技术支持)。 (修正：后来仔细读了几遍 man 手册，发现 sched_setaffinity 绑定特定的 cpu 对于多线程也是适用的，并且实验通过,只需要将第一个参数置为 0。这对 cpu0 的解放是个很好的发现。)

将网卡中断分散在多个 cpu 硬件解决方案: 我们新加了一块网卡（前提是这个网卡支持中断分布），然后通过通过 linux bonding 将两个网卡比如 eth0,eth1 联合成一个通道 bond0（当然这里还涉及到交换机的调整），然后 bond0 就有了 2G 的带宽吞吐量。把 eth0 的中断处理帮定在 cpu 0-3，把 eth1 中断处理帮定在 cpu 4-7，这样中断就被分布开了。这样会带来一些额外的 cpu 开销，但是跟好处相比可以忽略不计。我在网卡优化过的 32bit 服务器上测试 http 请求处理极限为 40w+ req/s，将近提升了一倍。

五，调整内核参数

我的内核心参数调整原则是，哪个遇到瓶颈调哪个，谨慎使用，不能凭想象乱调一气。看下面例子，其中 default 是我们公司定做的系统默认的一些参数值。add by yangjian2 并非全部都要调整，我只挑几个比较重要的参数说明一下，更多 TCP 方面的调优请参见 man 7 tcp 。

```
#++++++++++++++++++++default++++++++++++++++++++++++++++++++++
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_max_tw_buckets = 180000
net.ipv4.tcp_sack = 1
net.ipv4.tcp_window_scaling = 1
net.ipv4.tcp_rmem = 4096        87380   4194304
net.ipv4.tcp_wmem = 4096        16384   4194304
#++++++++++++++++++++add by yangjian2++++++++++++++++++++++++
net.ipv4.tcp_max_syn_backlog = 65536
net.core.netdev_max_backlog =  32768
net.core.somaxconn = 32768

net.core.wmem_default = 8388608
net.core.rmem_default = 8388608
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216

net.ipv4.tcp_timestamps = 0
net.ipv4.tcp_synack_retries = 2
net.ipv4.tcp_syn_retries = 2

net.ipv4.tcp_tw_recycle = 1
#net.ipv4.tcp_tw_len = 1
net.ipv4.tcp_tw_reuse = 1

net.ipv4.tcp_mem = 94500000 915000000 927000000
net.ipv4.tcp_max_orphans = 3276800
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

maxfd: 对于系统所能打开的最大文件描述符 fd，可以通过以 root 启动程序，setrlimit()设置

maxfd 后，再通过 setuid()转为普通用户提供服务,我用的 int set_max_fds(int maxfds); 函数是 zhongying 提供的。这比用 ulimit 来的方便的多，不晓得为什么那么多开源软件都没这样用。

net.ipv4.tcp_max_syn_backlog = 65536：这个参数可以肯定是必须要修改的，默认值 1024，我 google 了一下，几乎是人云亦云，没有说的明白的。 要讲明白得从 man listen 说起，int listen(int sockfd, int backlog); 早期的网络编程都中描述，int backlog 代表未完成队列 SYN_RECV 状态+已完成队列 ESTABLISHED 的和。但是这个意义在 Linux 2.2 以后的实现中已经被改变了，int backlog 只代表已完成队列 ESTABLISHED 的长度，在 AF_INET 协议族中（我们广泛使用的就是这个），当 int backlog 大于 SOMAXCONN (128 in Linux 2.0 & 2.2)的时候，会被调整为常量 SOMAXCONN 大小。这个常量可以通过 net.core.somaxconn 来修改。而未完成队列大小可以通过 net.ipv4.tcp_max_syn_backlog 来调整，一般遭受 syn flood 攻击的网站，都存在大量 SYN_RECV 状态，所以调大 tcp_max_syn_backlog 值能增加抵抗 syn 攻击的能力。

net.ipv4.tcp_syncookies = 1：当出现 syn 等候队列出现溢出时象对方发送 syncookies。目的是为了防止 syn flood 攻击，默认值是 0。 不过 man listen 说当启用 syncookies 时候，tcp_max_syn_backlog 的 sysctl 调整将失效，和这个描述不是很符合。参见下面两个描述分别是 man listen 和 man 7 tcp:
When syncookies are enabled there is no logical maximum length and this tcp_max_syn_backlog sysctl setting is ignored.
Send out syncookies when the syn backlog queue of a socket overflows.
但我可以肯定的说这个选项对你的性能不会有提高，而且它严重的违背 TCP 协议，不允许使用 TCP 扩展,除非遭受攻击，否则不推荐使用。

net.ipv4.tcp_synack_retries = 2：对于远端的连接请求 SYN，内核会发送 SYN + ACK 数据报，以确认收到上一个 SYN 连接请求包。这是所谓的三次握手(threeway handshake)机制的第二个步骤。这里决定内核在放弃连接之前所送出的 SYN+ACK 数目。如果你的网站 SYN_RECV 状态确实挺多，为了避免 syn 攻击，那么可以调节重发的次数。

net.ipv4.tcp_syn_retries = 2：对于一个新建连接，内核要发送多少个 SYN 连接请求才决定放弃。不应该大于 255，默认值是 5，对应于 180 秒左右。这个对防止 syn 攻击其实是没有用处的，也没必要调节。

net.ipv4.tcp_max_orphans = 3276800：这个最好不要修改，因为每增加 1，将消耗~64k 内存。即使报错 TCP: too many of orphaned sockets 也有可能是由于你的 net.ipv4.tcp_mem 过小，导致的 Out of socket memory，继而引发的。

net.ipv4.tcp_wmem = 4096 16384 4194304： 为自动调优定义每个 socket 使用的内存。第一个值是为 socket 的发送缓冲区分配的最少字节数。第二个值是默认值（该值会被 wmem_default 覆盖），缓冲区在系统负载不重的情况下可以增长到这个值。第三个值是发送缓冲区空间的最大字节数（该值会被 wmem_max 覆盖）。

net.ipv4.tcp_rmem = 4096 87380 4194304：接收缓冲区，原理同上。

net.ipv4.tcp_mem = 94500000 915000000 927000000：
low：当 TCP 使用了低于该值的内存页面数时，TCP 不会考虑释放内存。
pressure：当 TCP 使用了超过该值的内存页面数量时，TCP 试图稳定其内存使用，进入 pressure 模式，当内存消耗低于 low 值时则退出 pressure 状态。
high：允许所有 tcp sockets 用于排队缓冲数据报的内存页数。

一般情况下这个值是在系统启动时根据系统内存数量计算得到的，如果你的 dmesg 报 Out of socket memory，你可以试着修改这个参数，顺便介绍 3 个修改方法:

1, echo "94500000 915000000 927000000" > /proc/sys/net/ipv4/tcp_wmem
2, sysctl -w "net.ipv4.tcp_mem = 94500000 915000000 927000000"
3, net.ipv4.tcp_mem = 94500000 915000000 927000000　(vi /etc/sysctl.conf 然后 sysctl -p 生效)

下面命令也许能提供些信息，在你修改 tcp 参数时做个参考:
[sports@xk-6-244-a8 nbahttpd_beta4.0]$ cat /proc/net/sockstat
sockets: used 1195
TCP: inuse 1177 orphan 30 tw 199 alloc 1181 mem 216
UDP: inuse 0 mem 0
RAW: inuse 0
FRAG: inuse 0 memory 0

其他我就不多说了，知道这些基本就能解决绝大部分问题了。

六，衡量 **Web Server** 的性能指标

我认为一个好的 Server 应该能在有限的硬件资源上将性能发挥到极限。
Web Server 的衡量指标并非单一，要根据具体应用类型而定。比如财经实时图片系统，我们关注它每秒输出图片数量。NBA js 直播放系统，我们关心他的同时在线 connections 和当时的每秒请求处理量。行情系统，我们关心它 connections 和请求处理量的同时还 要关心每个请求平均查询多少支股票。但总体来说同时在线 connections 和当时的每秒请求处理量是两个最重要的指标。

对于图片系统再说一句,我觉得大图片和小图片是应该区别对待的，小图片不应该产生磁盘 I/O 。

Nginx 是我见过的 Web Server 中性能比较高的一个,他几乎是和我的 server 同时诞生，可能还更早些，框架很不错，我觉得目前版本稍微优化下，支持 10w connections 不成问题。 lighttpd 也不错，我对他的认识还是停留在几年前的性能测试上，它的性能会比 nginx 逊色一些。他们都支持 epoll,sendfile,可以起 多个进程 worker，worker 内部使用非阻塞，这是比较优良的 I/O 的模型。Squid,Apache，都是骨灰级软件了，好处就是支持的功能多， 另许多轻量级 Server 望尘莫及，可是性能太一般了，祝愿他们早日重写。

插点小插曲，我在财经项目组的时候，有的同事来我们组一年多了，问我是不是管机器的，我点点头，后来又有比较了解我的同事说我是系统管理员，我说“恩”。其实我的主业是写程序的。也许是我太低调了，觉得那些陈年往事不值再提，以至于别人对我做的东西了解甚少，今天我就高调一把，公布一些我写的程序的性能指标。我们的系统近几年来说在性能上是领先业内的(不争世界第一，那样压力太大，第二就很好,也许正在看我 blog 的你一不留神就把我超了呢 ^-^)，高效的原因很重要的一点是由于它是根据服务特点量身订做的。

实验环境数据： 我写了个 HTTP 服务框架，不使用磁盘 I/O，简化了逻辑处理部分，只会输出 "hello world!" 程序部署在 192.168.0.1 上(2cup*4Core,硬件和系统都做过优化)，我在另外 8 台同等配置服务器上同时执行命令 ./apache/bin/ab -c 1000 -n 3000000 -k "http://192.168.0.1/index.html" 几乎同时处理完毕，总合相加 40w req/s，我相信这是目前硬件水平上的极限值 。

真实环境数据：2cup*4Core Mem 16G, 64bit centos5，单机 23w+ connections, 3.5w req/s 时，CPU 总量消耗 1/8，内存消耗 0.4%（相当于正好消耗了一个 Core+64M Mem）。在 30w+ connections, 4.6w req/s 时,CPU 总量消耗 1/4，内存消耗 0.5% 。保守地说，只要把网卡中断分散一下，单机 50w+ connections 很 easy。 更多数据图文参见“NBA js 直播的发展历程”一节。

有些人了解我是由于财经的实时行情系统，虽然每天处理近百亿的 http 请求处理量还不错，但那并非我的得意之作，相反我觉得那个写的有些粗糙，至少有一倍 以上的性能提升空间。对于行情系统，我还是很想把它做成 push 的，目标仍然是单机 50w+在线，无延迟推送,可惜本人 js 功底太烂，所以要作为一个长期 的地下项目去做,如果可能，我想一开始就把它作为一个开源项目来做。

我个人比较喜欢追求性能极限，公司对此暂时还不是很认可,或者说重视程度还不够，可能是由于我们的硬件资源比较充裕吧。尽管如此，只要我认为对企业有价值 的，就依然会坚持做下去，我的目标是获得业界的认可。同时我相信中国的未来不缺乏互联网用户，当有人烧不起钱的时候想起了我，那我就是有价值的。

这里说的有点多了，不过放心，ppt 我会做的相当简单。