实践：服务器编写/系统架构/cache

从基本HTTP协议，服务器编写(只讲思路)，到完整系统搭建(包括负载均衡LVS，IDC分布，DNS解析)，到浏览器缓存的使用(重点讲述)，结合线上实例图文讲解如何以最低廉的成本构建快速，高并发，高可用，可扩展的Web服务。最后将拿本公司一些线上产品做实例分析。如果能灵活应用这些方法，你也许会发现对于某些网站来说节约10倍成本，其实是个很保守的说法。

- **一、服务器编写篇**
- **二、系统架构篇**
- **三、Cache为王篇**
- **四、实例分析篇**
- **五、动态应用篇**

服务器编写

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

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)

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

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１３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

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
```

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）

```
[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。

七，NBA js直播的发展历程

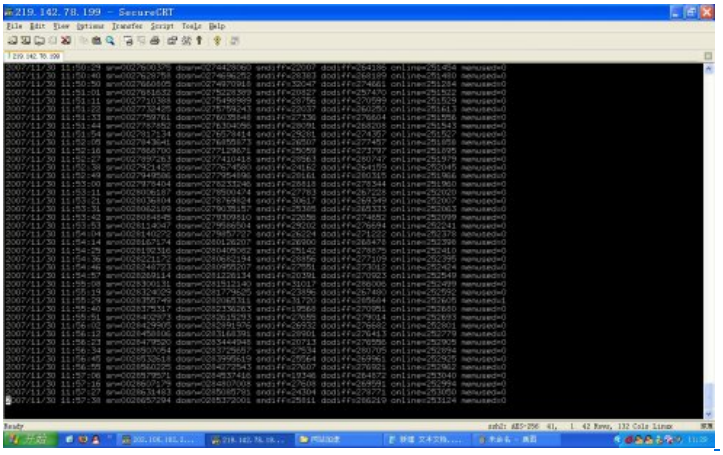
这一节就谈下这个项目发展过程中所遇到的瓶颈，以及如何解决的。
应该是06年吧，当时NBA 比赛比较火,woocall负责高速模式图文直播放，普通模式和动态比分数据等都放在一群破服务器上，大概有十几20台，这些破服务器有些扛不住了。

因为第二天有一场比较大的比赛，我想埋连接在线上测一下效果，于是连夜把财经实时行情server改写成了NBA JS直播server. 当时有两台 Intel(R) Xeon(TM) CPU 3.00GHz 双cpu的服务器，在F5后面。先启用一台服务器，比赛开始前静悄悄的，不一会，迅速串到了20w connections，再往上增长，就慢的几乎不可访问， ethtool eth0 , Speed: 100Mb/s, 网卡出口带宽跑满了（那时候支持千兆的交换机还不多）。迅速把另一台服务器启用，后来又卡了，好象是F5处理能力不足。 后来升级服务器出口带宽为1G，当然这需要交换机支持千兆口，更换网线，服务器也从F5后面转移出来，通过DNS直接轮询。

后来又测试了几次，等到新申请的Intel(R) Xeon(R) CPU 5120 @ 1.86GHz， 双核双cpu服务器一到，就开始大规模部署，比赛更火了，不巧的是行情也火了起来，我的财经实时图片系统和行情系统也是带宽杀手，同时我也成了服务器杀 手，到处蹭服务器。IDC带宽出口开始告急,我每天开始关注哪个机房还有富余带宽，有的机房被我们跑的太满，开始有人劝我们迁移到别的机房。后来 yangguang4劝我支持gzip输出，我觉得动态压缩太耗费cpu，不知道预先压缩是否可行，写个程序试了一把，没问题，于是NBA JS直播的的带宽一下子被砍掉了70%,而且没浪费一点我们的cpu，赚大了。

事，woocall瞬间仍给我近200w connections，网通的服务器被秒杀了1/3。 这其实就是善意的DDOS攻击，这些用户如果正常进入是没有问题了，瞬间扔过来，超出了操作系统极限，系统挂掉了，我的服务也over了。在调整内核参数里有讲，怎么修改内核参数提高服务器抗秒杀能力，但是不能杜绝。

下图为2007年一场比赛时,单机25w+ connections，2.86w req/s,的状态(2CPU*2Core 1.86GHZ):



奥运结束后，我对服务器程序和架构做了调整，砍掉了2/3的服务器。 但我没留意的是，同样connections，实际http请求增加了一倍，因为新上了一个flash方位图，里面增加了3个js，增加就增加吧，既然砍了就不准备再恢复了。但是服务在2CPU*4Core centos5 32bit上的表现却让我很失望，跑不过2CPU*2Core centos4 32bit 。 我开始怀疑程序升级的时候是不是有什么地方没考虑到，开始调程序，折腾几天没有结果，症状是单机支撑12.5万时候没有任何异常，内存使用1%左右，总cpu使用了5%左右，load 0.5，但是再增加0.1w用户server肯定崩溃，每次都是相同的表现,我知道在什么地方卡住了。 后来看了下dmesg，才恍然大悟，我是被32bit centos 5的内核暗杀的(Out of memory: Killed process xxx)。

32位机上LowFree一般是会变化的(cat /proc/meminfo | grep LowFree)，最大不能超过880M（这个值不能改变除非用hugemem内核)，在centos4 上有内核参数vm.lower_zone_protection(单位是M)来调节LowFree，默认vm.lower_zone_protection=0 ，LowFree=16M，但是在centos5上这个参数貌似被取消了，改变不了。 从使用经验来看，也就是你能申请16M~880M这么大的连续内存，但是16M以上不保证你能申请的到。centos4用到64M以上都没什么问题，centos5 用40M+ 就被OOM Killer给毙了。

本周开始使用64bit centos5.2进行测试，迁移很顺利，没有修改一行代码，他们把整个16G物理内存都作为LowFree，这下可以随便挥霍了(尽管如此我还是会节约的)，前几天的比赛，这个64bit机跑了18w connections，很安静，未见异常，等有大比赛再检验下，没问题的话就开始大规模使用64bit系统。

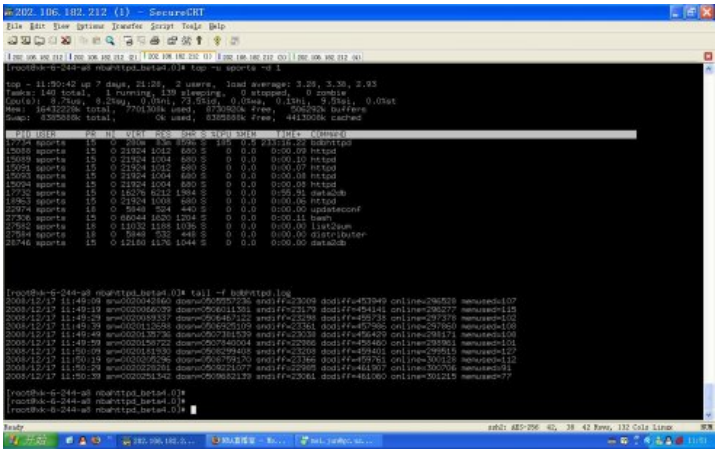
目前看来，如果成功迁移64bit系统似乎可以高枕无忧了，但是我还是有两个忧虑:

1,再突然用200w connections给我，我不敢保证能扛的住，因为现在服务器数量消减太多，需要yangguang4那边做策略调整，在比赛结束后，平滑的把用户丢给我,这应该有个持续过程，而不是一秒内的瞬间。

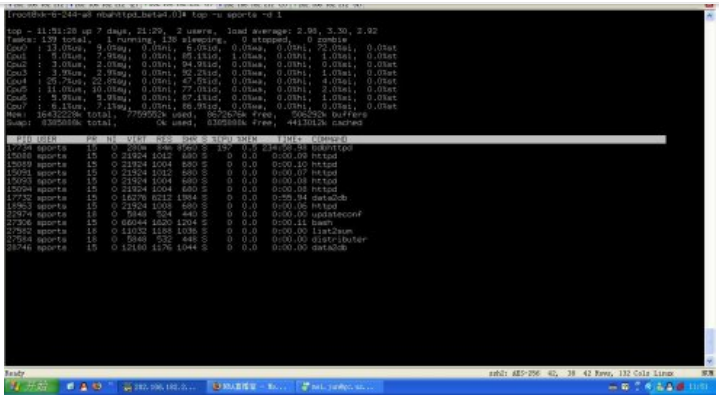
2,我猜这个系统，单机支撑到30w conections的时候会遇到瓶颈，因为网卡的中断集中在cpu0上，没有均衡开。我们有硬件解决方案已经实现（每个服务器会多2000RMB开销）我只部署了一台，但是软的还没实现,寄希望于xiaodong2 。

补充：
昨天的比赛中，一台64bit机，单机支撑30w+ connections，cpu0空闲率只剩6%，和我的预料是一致的。当时的CPU总量被我用掉近 1/4，内存被我用掉 0.5% 。

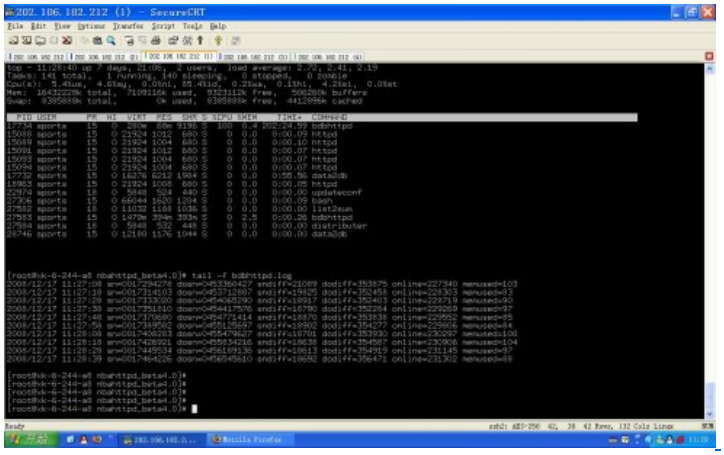
下图为30w+ connections, 4.6w req/s 的时候我的程序使用的资源情况(2cpu*4Core):



下图为cpu使用分布情况,cpu0空闲率只剩 6% (2cpu*4Core):



另外附上一个23w connections, 3.5w req/s 的时候我的程序使用的资源情况(2cpu*4Core),当时cpu只被用掉1/8，内存被用掉 0.4% ，cpu没有发挥线性增加的作用,我肯定不说能我可以支撑23w*8,但是保守地说，只要把网卡中断分散一下，单机50w+ connections很easy。



八，新浪财经实时行情系统的历史遗留问题 (7 byte = 10.68w RMB/year)

这点我还是提下吧，估计我不说，大家也想不到。
先感谢wangyun同学的大胆使用才有了今天的财经实时行情系统（当初是从一台PIII 900服务器上发展起来的，前几天刚被我下线）不过 "hq_str_" 这7个字节的前缀，也是他造成的,当初他说改抓取页面有些麻烦，就让我写死在server里，虽然意识到将来也许会有隐患，但我还是照做了。见下面返回数据：
http://hq.sinajs.cn/list=s_sz000609,s_sz000723,s_sh000001

var hq_str_s_sz000609="绵世股份,9.29,-0.05,-0.54,170945,16418";
var hq_str_s_sz000723="美锦能源,0.00,0.00,0.00,0,0";
var hq_str_s_sh000001="上证指数,2031.681,-47.436,-2.28,1216967,8777380";

我算了一笔帐，行情好的时候每秒会产生30~40w个请求，一般一个请求会请求3~50只股票，保守点就按每个请求5只股票来计算，每秒会产生200w只股票查询信息。 由于这7个字节产生的带宽为：
200w * 7byte * 8bit / 1024 /1024 = 106.8 M ,而往往我们的带宽要按峰值来准备，按1G带宽100w RMB/year 计算，每年耗费10.68w RMB。把这笔钱拿给我做奖金，我会很happy的 ^-^ . 现在因为很多页面都使用了行情数据，想修改，代价很高。

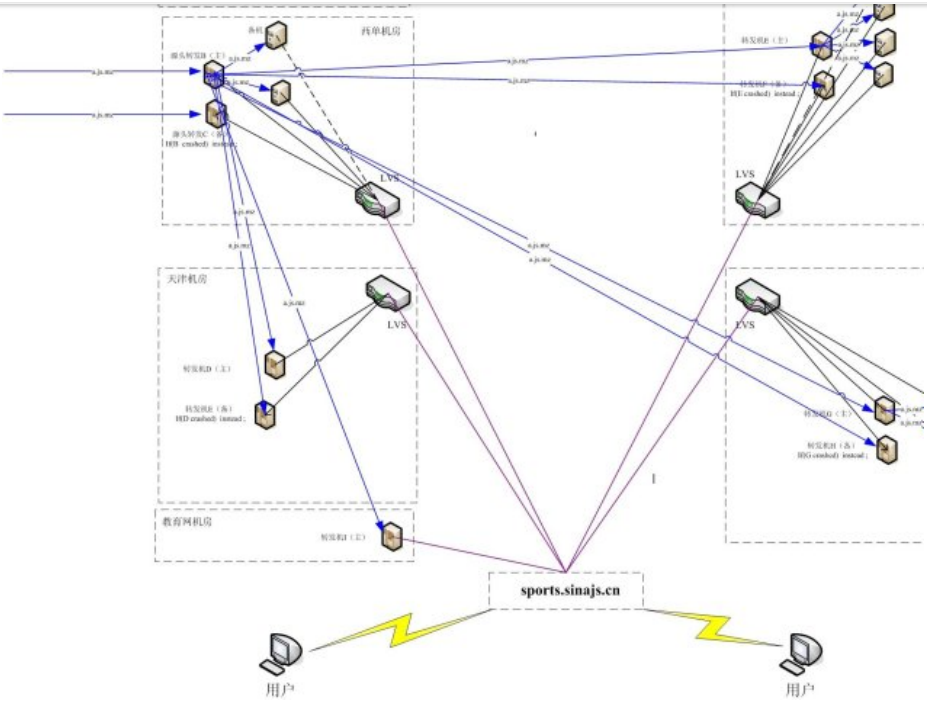
所以设计系统的时候一定要考虑的稍微远一些,哪怕当时只是一点点微不足道的地方，要考虑将来访问规模变大了会是什么后果。还有就是要敢于坚持自己的原则。

系统架构篇

一，系统部署（高并发,可扩展）
二，负载均衡LVS（高可用,低成本）
三，IDC分布，DNS解析（快速）

一，系统部署（高并发,可扩展）

本来想画在手稿上然后扫描上去的，貌似方法太土，在朋友的帮助下费了n个小时用Visio画了个，感觉很好看 ^-^ 。这一篇将主要围绕这个图来讲述。



首先从数据源说起，所谓狡兔三窟，我们数据源也是按三路设计，以保证IDC内部和不同IDC之间实现灾备。源头转发机A,B,C拥有往集群中任何一台服务器同步数据的权限，所以他们三个有一个活着，数据就可以同步更新,而且可以自动切换。从源头转发机到其他各IDC的数据都是双路的，然后每个IDC的前两台服务器具备转发功能，往IDC内部其他服务器分发数据,同一IDC内部的主备转发机可以自动切换。这样就实现了数据同步更新的高可用性。

介绍下这个集群里的角色，备机A来自行情系统，兼任源头转发的异地备份。系统内的另外两个备机属于轻负荷服务器，80端口空出来，必要时候只要一启动，就会立即自动加入到LVS后面服役。除了A以外所有具备转发功能的机器同时也是集群内的普通成员，需要提访问供服务的。各IDC的LVS本身也是有主备的，可以实现自动切换。

整个系统增减服务器非常方便，用户根本感觉不到，备机的启用更快，也就3~5秒，具备很好的扩展性。

我们的数据从源头上就是使用我编写的myzip压缩好了的，后缀名用"*.mz" ,比如 a.js.mz ,一直到用户的浏览器端才解压。数据传输量小，速度快。 源头转发机上同时运行一个checkchange的程序，确保内容实际更新过的文件才往其他IDC转发，这样能有效的减少传输文件数量,以达到更快的更新速度。

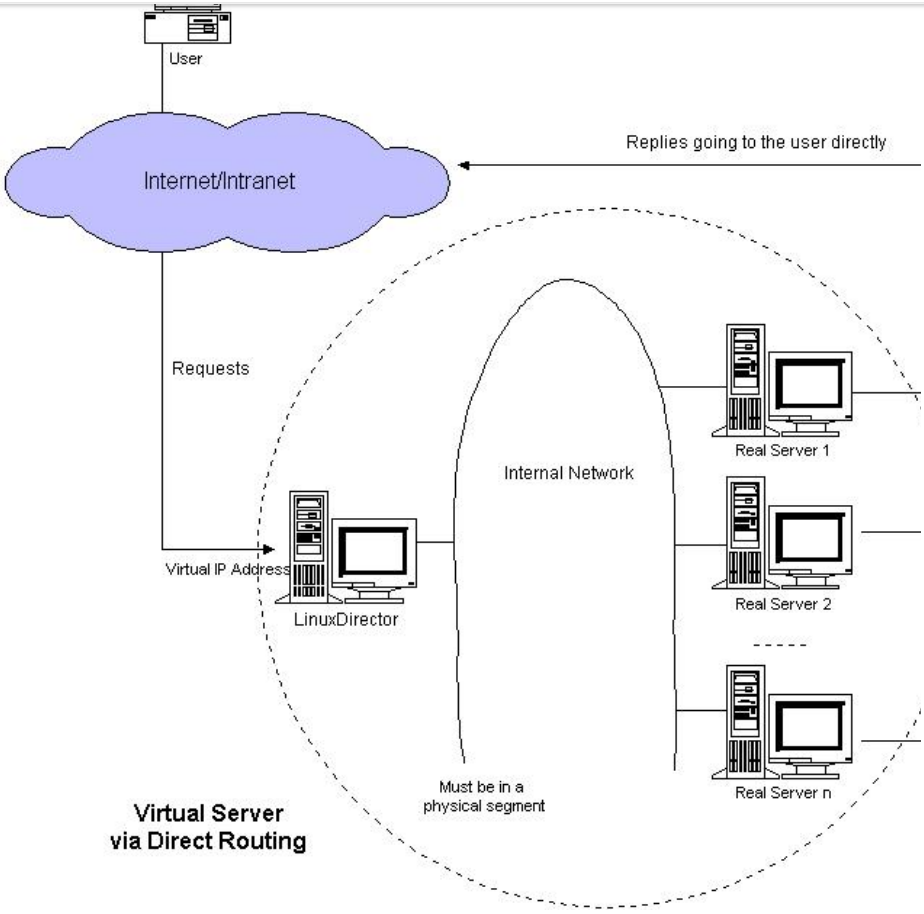
另外，跨IDC系统部署，很重要的一点是，内网连通，路由选择，这影响数据传输速度的关键。北京的各机房间一般都有比较好的专线连通，只需要把路由打通就ok了。跨IDC的，一般都使用vpn来做内网传输，有条件的使用专线，这个比较昂贵，省着点用。另外跨网通，电信，和移动机房的一般都从双线机房路由，或者说，从到不同信息服务商连通性都比较好的机房路由。总之跨IDC数据传输，要做到各IDC之间的传输速度心中有数。

最后，请稍微注意下系统的安全性，包括数据传输的安全性，和网络安全性，避免遭受攻击。

二，负载均衡LVS（高可用,低成本）

LVS 有三种模式，NAT，TUN，DR,其中DR是最高效的，下面我将主要介绍DR的应用。更多LVS资料参见 LVS项目中文文档。 目前我们公司的LVS应用规模在国内应该至少可以排前三，更多技术细节请咨询我们的LVS大牛xiaodong2.

下面是DR单臂模式的系统结构图:



下面引用一下官网的介绍: 在VS/DR 中，调度器根据各个服务器的负载情况，动态地选择一台服务器，不修改也不封装IP报文，而是将数据帧的MAC地址改为选出服务器的MAC地址，再将修改后 的数据帧在与服务器组的局域网上发送。因为数据帧的MAC地址是选出的服务器，所以服务器肯定可以收到这个数据帧，从中可以获得该IP报文。当服务器发现 报文的目标地址VIP是在本地的网络设备上，服务器处理这个报文，然后根据路由表将响应报文直接返回给客户。

我在财经时要使用LVS的初衷只是为了解决负载的均衡性，因为DNA轮询各前端服务器上连接数有不小的差距，那时候我们老大阿图对于这个项目给予了很大支持，还亲自组织过几次会议。话说恰巧yingyuan做了个新技术讲座，我从中发现LVS/DR后想让它帮忙修改负载均衡算法，后来部署上以后发现，不用修改，均衡的很,再后来xiaodong2接手后对性能和稳定性做了很大的提升，我们使用两年来没出过问题。另外lvs还额外带来了两个好处，高可用性，和可伸缩性。是可以随时把lvs后面的一台服务器下掉，扛走，用户是不知道的。服务器坏了也不用着急修，也不用修改DNS(另外DNS的层层cache影响不是一时半会就能消除的)。新增加一台服务器也是同理，最绝的就是备机的启用可以用秒来衡量（这些F5都能实现，代价不菲）。财经应用对公司内lvs的项目推动有不可磨灭的贡献 ,xiaodong2也这么说地 :) 。

三，IDC分布，DNS解析（快速）

这里思路跟CDN是一致的，尽量减少主干线路上的拥塞，让用户就近访问，以达到最快的数据传输速率。我们要做的就是了解自己应用的用户分布情况，然后再结合现有资源以及各地网络出口特征，信息服务商的特征来部署我们的服务。

1，各省市网络用户分布依次排名(数据来自cnnic2007年的统计)：
广东 13.4%
山东 8.2%
江苏 7.5
浙江
四川
河北
河南
福建
上海
辽宁
北京
湖南
山西
黑龙江

2，运营商的网络分布特点：

去就，让他们在自己省内访问，访问速度会立刻提升n倍。

电信:以几大省市为核心的环状结构，省市内部也是大环套小环。其中以广东用户最多，必须要部点的地方。记得很久以前我拿到一份数据说，上海人访问本IDC的数据，不如访问广东的速度快，不晓得现在是否还存在这种情况。电信有7个主要核心，分布在广州，上海，江苏，西安，成都，武汉，北京。

教育网：以国内主要的八个结点为核心。这八个结点分布在北京，西安，成都，广州，武汉，南京，上海，和沈阳。

3，DNS解析时候需要权衡的：
现在了解了这些信息，那我们开始讨论如何部署我们的服务。要考虑两个问题：一，要部署在哪几个IDC。二，每个IDC部署的服务器数量。三，DNS如何按区域划片。

要部署在哪几个IDC？
其实这里还涉及到规模化应用的好处，一个小应用就部署了N多个IDC显然不划算。如果我的应用上了规模，我可以在每个省都部署上，那样用户体验将非常好，而且规模化以后会有专业人员对应用进行优化。所以公司里有动态池，和静态池这样的公用平台是好事(也许将来还会有我的js池)。如果我们的服务还没有上升到公司级别的规模，那就得考虑下取舍。

网通：东北三省，可以在沈阳和哈尔滨选择一个部署，有条件可以都部署。沈阳到北京的速率比哈尔滨到北京的速率快一倍，而哈尔滨到沈阳的速率，还不如到北京的快。北京，如果只让我在网通部署一个点的话，毫无疑问我会选择北京，其他所有结点到它的速率都比较快，但是北京的带宽比较昂贵。天津，这个点重要性仅次于北京,可以辐射河北，河南,江苏，离北京也比较近,价钱便宜。太原，可以辐射到西北一带。山东，前面已经说过，最好要部署的。

电信：那七个核心结点上部署了，速度就有保证。具体覆盖范围。广州覆盖周边几省，上海覆盖本地，江浙一带。 武汉覆盖华中一带，西安可以覆盖西北5省，成都覆盖西南5省。

每个IDC部署的服务器数量?
这要根据具体应用来决定。比如财经用户网通,电信比例：3:4 而体育是 1:2 。教育网用户一般占1/30左右。这里还不能单纯考虑用户分布，还要考虑IDC内部灾备和IDC间灾备，是要有个取舍的。拿咱们的某个具体项目来说，教育网，够不上一台服务器，但是不得不部，因为它访问外界实在太慢了，我就住在学校里，也为了方便自己。我把北京作为主要结点部署了3台，天津，其实一台就够了,山东一台有点多。但是考虑到北京IDC一旦倒了，实力相当的IDC可以灾备，同时考虑到，天津，和山东只有一台，idc内部，都无法实现灾自动切换。所以，我选择天津两台，山东不部署，以性能换安全。

DNS如何按区域划片?
原则，就近分片，以达到最快传输速率。其次，考虑到各IDC间快速切换比较容易，DNS解析文件要写的简洁一些。另外，DNS解析有有个缺陷，每个单独域名里写在最前面的那个ip，它被轮询到的概率要比同组的服务器高10%，而且随着同组服务器的增多，这个差距会变大。所以最解析时候，每个IDC我都把硬件性能最好的服务器ip放在最前面。

另外：
做系统架构不提数据库，有点过不去。这块问题可以请教我们的DBA大牛zongwen同学。数据库是我将来一年的学习重点，争取一年后在DB方面能达到我们DBA六层功力。

Cache为王篇

一，Cache， 王道也
二，Cache 基本原理介绍
三，我划分的３个刷新级别
四，我对HTTP协议做的一点创新(?maxage=6000000)
五，Yslow优化网站性能的14条军规点评
六，上线了 != Finished
七，提速度同时节约成本方法汇总

一，Cache，王道也

我觉得系统架构不应该仅仅是搭建一个强硬的能承受巨大并发压力的后台，前端页面也是需要架构的而且同等重要，不理解前台的的后台工程师是不合格的。中国人讲究钢柔相济，后台强硬只能说你内功深厚，前端用的巧，那叫四两拨千斤。

一般后台工程师很少关心前端如何使用自己的资源，而前端工程师，不知道自己的一个简单的用法会对后端造成多大影响。我会给出一些数据，来震撼下你的眼球。

二，Cache 基本原理介绍 (参考Caching Tutorial)

为什么使用Cache？
1，减少延迟，让你的网站更快，提高用户体验。
2，避免网络拥塞，减少请求量，减少输出带宽。

Cache的种类?
浏览器Cache，代理Cache，网关Cache。
后端还有 disk cache ,server cache，php cache，不过不属于我们今天讨论范围。

Cache如何工作的?
1，如果响应头告诉cache别缓存它，cache不对它做缓存；
2，如果请求需要验证的或者是需要安全性的，它将不被缓存；
3，如果响应头里没有ETag或Last-Modifed header这类元素，而且也没有任何显式的信息告诉如何对数据保鲜，则它被认为不可缓存。
4，在下面情况下，一个缓存项被认为是新鲜的(即，不需到原server上检查就可直接发送给client):
它设置了一个过期时间或age-controlling响应头，而且现在仍未过期。
如果浏览器cache里有某个数据项，并且被被设置为每个会话(session)过程中只检查一次；
如果一个代理cache里能找个某个数据项，并且它是在相对较长时间之前更新过的。
以上情况会认为数据是新鲜的，就直接走cache，不再查询源server。
5，如果有一项过期了，它将会让原server去更新它，或者告诉cache这个拷贝是否还是可用的。

怎么控制你的Cache？
Meta tags： 在html页面中指定，这个方法只被少数浏览器支持，Proxy一般不会读你html的具体内容然后再做cache决策的。

Pragma: no-cache : 一般被大家误用在http响应头中，这不会产生任何效果。而实际它仅仅应该用在请求头中。 不过google的Server: GFE/1.3 响应中却这样用，难道人家也误用了呢。

Date: 当前主机GMT时间。

Last-Modified : 文件更新GMT时间，我在响应头中带上这个元素的时候，通常浏览器在cache时间内再发请求都会稍带上If-Modified-Since，让我们判断需要重新传输文件内容，还是仅仅返回个304告诉浏览器资源还没更新，需要缓存策略的服务器肯定都得支持的。有了这个请求，head请求在基本没太多用处了，除非在telnet上调试还能用上。

If-Modified-Since： 用在请求头里，见Last-Modified 。

Etag: 标识资源是否发生变化，etag的生成算法各是各样,通常是用文件的inode+size+LastModified进行Hash后得到的,可以根据应用选择适合自己的。Last-Modified 只能精确到秒的更新，如果一秒内做了多次更新，etag就能派上用场。貌似大家很少有这样精确的需求，浪费了http header的字节数，建议不要使用。
更正：Etag 其实在某种情况下可以很好的减少数据传输。在stonehuang的提醒下我才恍然大悟，转眼好几个月了也一直忘记更新。Etag应用场景。比如，数据为php的动态输出。每次请求把上一次Etag带来，跟本次计算的Etag进行比较，相等就可以避免一次数据传输。(最后修改时间 2009.12.07)

Expires： 指定缓存到期GMT的绝对时间，这个是http 1.0里就有的。 这个元素有些缺点，一，服务器和浏览器端时间不一致时会有问题。 二，一旦失效后如果忘记重新设置新的过期时间会导致cache失效。三，服务器端需要根据当前Date时间 + 应该cache的相对时间去计算这个值，需要cpu开销。 我不推荐使用。

Cache-Control:
这个是http 1.1中为了弥补 Expires 缺陷新加入的，现在不支持http 1.1的浏览器已经很少了。
max-age: 指定缓存过期的相对时间秒数，max-ag=0或者是负值，浏览器会在对应的缓存中把Expires设置为1970-01-01 08:00:00 ,虽然语义不够透明，但却是我最推荐使用的。
s-maxage: 类似于max-age，只用在共享缓存上，比如proxy.
public: 通常情况下需要http身份验证的情况，响应是不可cahce的，加上public可以使它被cache。
no-cache: 强制浏览器在使用cache拷贝之前先提交一个http请求到源服务器进行确认。这对身份验证来说是非常有用的,能比较好的遵守 (可以结合public进行考虑)。它对维持一个资源总是最新的也很有用，与此同时还不完全丧失cache带来的好处，因为它在本地是有拷贝的，但是在用之前都进行了确认，这样http请求并未减少，但可能会减少一个响应体。
no-store: 告诉浏览器在任何情况下都不要进行cache，不在本地保留拷贝。
must-revalidate: 强制浏览器严格遵守你设置的cache规则。
proxy-revalidate: 强制proxy严格遵守你设置的cache规则。
用法举例: Cache-Control: max-age=3600, must-revalidate

其他一些使用cache需要注意的东西，不要使用post，不要使用ssl，因为他们不可被cache，另外保持url一致。只在必要的地方，通常是动态页面使用cookie，因为coolie很难cache。至于apache如何支持cache和php怎么用header函数设置cache，暂不做介绍，网上资料比较多。

如何设置合理的cache时间 ？
http://image2.sinajs.cn/newchart/min/n/sz000609.gif?1230015976759
拿我分时图举例，我们需要的更新频率是1分钟。但为了每次都拿到最新的资源，我们在后面加了个随机数，这个数在同一秒内的多次刷新都会变化。我们的js虽然能够很好的控制，一分钟只请求一次，但是如果用户点了刷新按纽呢？这样的调用是完全cache无关的，连返回304的机会都没有。

试想，如果很多人通过同一个代理出去的，那么所有的请求都会穿透代理，弄不好被网管封掉了。如果我们做只做一秒的cache，对直接访问源服务器的用户没太多影响，但对于代理服务器来说，他的请求可能会从10000 req/min 减少为 60 req/min ，这是160倍。

对于我们行情图片这样的情况，刷新频率为1分钟，比较好的做法是把后面的随机数(num)修改为 num=t-

拿不到最新的数据，其实对用户来说，用那个多变的随即数和我这个分钟级的随即数，看到的效果是相同的
下面我给你分析一下： 如果用户打开了我们的分时间页面，当前随即数对他来说是新的，所以他会拿到一个当前最新的图片，然后他点了刷新按纽，用户会产生http请求，即使url没变，服务器有最新图片也一定会返回，否则返回304，一分钟后js刷新图片，分钟数加了1，会得到全新资源。这和那个随时变化的随即数效果有区别吗？都拿到了最新的数据，但是却另外收益了cache带来的好处，对后端减少很多压力。

三，我划分的３个刷新级别

名词解释 全新请求： url产生了变化,浏览器会把他当一个新的资源(发起新的请求中不带If-Modified-Since)。

更正：在firefox后来的版本中对此做了改进，倾向于更多的使用cache，曾经访问过的都会尽量捎带If-Modified-Since头。这些表现和IE一致。修改部分用红色标出。(最后修改时间 2009.12.07)

注： sports.sinajs.cn 在IE下的表现存在一个小bug，由于不是使用的strncpy，导致IE下难以返回304,
需要修改一行代码，把比较字符串长度设置为29即可解决。不过目前本人已不在职，难以修改。
情况一 FF 捎带的头： If-Modified-Since Mon, 07 Dec 2009 10:54:43 GMT
情况二 IE 捎带的头： If-Modified-Since Mon, 07 Dec 2009 10:54:43 GMT; length=6

1,在地址栏中输入http://sports.sinajs.cn/today.js?maxage=11地址按回车。重复n次，直到cache时间11秒过去后，才发起请求，这个请求会带If-Modified-Since。

2,按F5刷新. 在你发起一个全新的请求以后，然后多次按F5都会产生一个带If-Modified-Since的请求。

3, ctrl+F5 ,总会发起一个全新的请求。

下面是按F5刷新的例子演示: http://sports.sinajs.cn/today.js?maxage=11
(如果这个值大于浏览器最大cache时间maxage，将以浏览器最大cache为准)

----------发起一个全新请求
GET /today.js?maxage=11 HTTP/1.1
Host: sports.sinajs.cn
Connection: keep-alive

HTTP/1.x 200 OK
Server: Cloudia
Last-Modified: Mon, 24 Nov 2008 11:03:02 GMT
Cache-Control: max-age=11 (浏览器会cache这个页面内容，然后将cache过期时间设置为当前时间+11秒)
Content-Length: 312
Connection: Keep-Alive
---------- 按F5刷新
GET /today.js?maxage=11 HTTP/1.1
Host: sports.sinajs.cn
Connection: keep-alive
If-Modified-Since: Mon, 24 Nov 2008 11:03:02 GMT (按F5刷新，If-Modified-Since将上次服务器传过来的Last-Modified时间带过来)
Cache-Control: max-age=0

HTTP/1.x 304 Not Modified
Server: Cloudia
Connection: Keep-Alive
Cache-Control: max-age=11 (这个max-age有些多余，浏览器发现Not Modified，将使用本地cache数据，但不会重新设置本地过期时间)

继续按F5刷新n次.....

这11秒内未产生http请求.直到11秒过去了.....
----------按F5刷新
GET /today.js?maxage=11 HTTP/1.1
Host: sports.sinajs.cn
Connection: keep-alive
If-Modified-Since: Mon, 24 Nov 2008 11:03:02 GMT (多次按F5都会产生一个带If-Modified-Since的请求)
Cache-Control: max-age=0

HTTP/1.x 304 Not Modified
Server: Cloudia
Connection: Keep-Alive
Cache-Control: max-age=11
----------按F5刷新
GET /today.js?maxage=11 HTTP/1.1
Host: sports.sinajs.cn
Connection: keep-alive
If-Modified-Since: Mon, 24 Nov 2008 11:03:02 GMT （同上 ...）
Cache-Control: max-age=0

Server: Cloudia
Connection: Keep-Alive
Cache-Control: max-age=11

四，我对HTTP协议做的一点创新(?maxage=6000000)

上面看到了url后面有 ?maxage=xx 这样的用法，这不是一个普通的参数，作用也不仅仅是看起来那么简单。他至少有以下几个好处：

1，可以控制HTTP header的的 max-age 值。
2, 让用户为每个资源灵活定制精确的cache时间长度。
3, 可以代表资源版本号。

首先谈论对后端的影响：
服务器实现那块，不用再load类似mod_expires，mod_headers 这样额外的module，也不用去加载那些规则去比较，它属于什么目录，或者什么文件类型，应该cache多少时间，这样的操作是需要开销的。

再说说对前端的影响：
比如同一个分时行情图片，我们的分时页中需要1分钟更新，而某些首页中3分钟更新好。不用js控制的话，那我cache应该设置多少呢？ 有了maxage就能满足这种个性化定制需求。

另一种情况是，我们为了cache，把某个图片设置了一个永久cache，但是由于需求，我必须更新这个图片，那怎么让用户访问到这个更新了的图片呢？ 从yahoo的资料和目前所有能找到的资料中都描述了同一种方法，更改文件名字，然后引用新的资源。 我觉得这方法太土, 改名后，老的还不能删除，可能还有地方在用，同一资源可能要存两份，再修改，又得改个名，存3份，不要不把inode当资源。 我就不那样做，只需要把maxage=6000000 修改成 maxage=6000001 ，问题就解决了。

maxage=6000000 所产生的威力 (内存块消耗减少了250倍 ,请求数减少了37倍) ：
体育那边要上一个新功能，一开始动态获取那些数据，我觉得那样太浪费动态池资源，就让他们把xml文件到转移到我的js池上来，为了方便，他们把那个84k的flash文件也放在了一起，而且是每个用户必须访问的。 说实在的，我不欢迎这种大块头，因为它不可压缩，按正常来说，它应该代表一个3M的文件。我的服务器只这样设计的，如果一次发送不完的就暂存在内存里，每个内存块10k，如果不带参数默认maxage=120 。 我发现，由于这个文件，10w connections的时候，我消耗了10000个内存块。我自己写的申请连续内存的算法也是消耗cpu地， 一个84k的文件，发送一次后，剩余的64k就应该能装的下，于是我把最小内存块大小改为64K。 这样消耗10w conn的时候消耗1500个左右内存快，虽然内存消耗总量没怎么变小，但是它能更快的拿到64K的连续内存资源，cpu也节约下来了。接下来我让meijun把所应用的flash资源后面加上maxage=6000000 (大概=79天,浏览器端最长cache能达到着个就不错了)， 10w connections的时候，只消耗了不到40个内存块,也就是说内存块消耗减少了250倍 ,请求数减少了37倍。 35w+ connections, 5.67w req/s的时候也就消耗100块左右，比线性增加要少很多。也就是这点发现让我有了做这个技术分享的冲动，其他都是顺便讲讲。

五，Yslow优化网站性能的14条军规点评

其中黑色部分，跟后端是紧密相连的，在我们的内容中都已经涉及到了，而且做了更深入的讨论。兰色部分，5，6，7是相关页面执行速度的，构建前端页面的人应该注意的。 11属于避免使用的方法。 红色部分我着重说一下：

gzip 我不推荐使用，因为有些早期IE支持的不好，它的表现为直接用IE访问没问题，用js嵌进去，就不能正常解压。这样的用户占比应该在2%左右。这个问题我跟踪了近一个月，差点放弃使用压缩。 后来发现我以前用deflate压缩的文件却能正常访问。 改用deflate问题解决。apache 1.x使用mod_gzip ,到了 2.x 改用cmod_deflate，不知道是否跟这个原因有关。 另外对于小文件压缩来说，deflate 可比 gzip 省不少字节。

减少 DNS 查询: 这里也是有个取舍的，一般浏览器最多只为一个域名创建两个连接通道。 如果我一个页面嵌了 image.xx.com 的很多图片，你就会发现，图片从上往下一张张显示出来这个过程。这造成了浏览器端的排队。 我们可以通过增加域名提高并发度，例如 image0.xx.com ,image1.xx.com ,image2.xx.com，image3.xx.com 这样并发度就提上去了，但是会造成很多cache失效，那很简单，假如我们对文件名相加，对4取mod，就能保证，某个图片只能通过某个域名进行访问。 不过，我也很反对一页面请求了数十个域名，很多域名下只有一到两个资源的做法，这样的时间开销是不划算的。

另外，我在这里再添一个第15条：错开资源请求时间，避免浏览器端排队。
随着ajax的广泛使用，动态刷新无处不在，体育直播里有个页面调用了我一个域名下的6个文件，3个js，3个xml。 刷新频率大致是两个10秒的，两个30秒的，两个一次性载入的。观察发现正常响应时间都在7ms, 但是每过一会就会出现一次在100ms以上的，我就很奇怪，服务器负载很轻呢。meijun帮我把刷新时间错开，11秒的，9秒的,31秒的，这样响应在100ms以上的概率减少了好几倍，这就是所谓的细节决定成败吧。

1. 尽可能的减少 HTTP 的请求数 [content]
2. 使用 CDN（Content Delivery Network） [server]
3. 添加 Expires 头(或者 Cache-control) [server]
4. Gzip 组件 [server]
5. 将 CSS 样式放在页面的上方 [css]
6. 将脚本移动到底部（包括内联的） [javascript]

9. 减少 DNS 查询 [content]
10. 压缩 JavaScript 和 CSS (包括内联的) [javascript] [css]
11. 避免重定向 [server]
12. 移除重复的脚本 [javascript]
13. 配置实体标签（ETags） [css]
14. 使 AJAX 缓存

六，上线了 != Finished

奥运期间我按1500w~2000w connections在线，设计了一套备用系统，现在看来，如果用户真达到了这个数目我会很危险，因为有部分服务器引入了32bit的centos 5未经实际线上检验，而我当时简单的认为它应该和centos 4表现出一样的特性。所以现在未经过完全测试的lib库和新版本，我都很谨慎的使用。没在真实环境中检验过，不能轻易下结论。

很多项目组好象不停的忙，做新项目，上线后又继续下个新项目，然后时不时的转过头去修理以前的bug。如果一个项目上线后，用户量持续上升，就应该考虑优化了，一个人访问，和100w人访问，微小的修改对后端影响是不能比较的，不该请求的资源就让它cache在用户的硬盘上，用户访问块了，你也省资源。上线仅仅代表可以交差了而已，对于技术人员来说持续的对一个重要项目进行跟踪和优化是必要的。

七，提速度同时节约成本方法汇总

1，编写节约的HTTP服务器 (高负载下速度明显提升，节约5~10倍服务器)
对一些重要的服务器量身定做。或者选用比较高效的开源软件进行优化。

2，不同服务混合使用 （节约1~2倍服务器）
如果我们一台服务器只支持30w conn的话，那么剩余的75% cpu资源，95%的内存资源，和几乎所有的磁盘资源都可以部署动态池系统，我觉得DB对网卡中断的消耗还是有限的，我也不用新买网卡了。

3,对于纯数据部分启用新的域名(速度有提升，上行带宽节约1倍以上)
比如我们另外购买了sinajs.cn 来做数据服务，以避免cookie,节约带宽. Cookie不但会浪费服务器端处理能力，而且它要上行数据，而通常情况上行比下行慢。

4， 使用长连接 (速度明显提升，节约带宽2倍以上，减少网络拥塞3~无数倍)
对于一次性请求多个资源，或在比较短的间隔内会有后续请求的应用，使用长连接能明显提升用户体验，减少网络拥塞，减少后端服务器建立新连接的开销。

5，数据和呈现分离，静态数据和动态数据分离 (速度明显提升，同时节约3倍带宽)
div+css 数据和呈现分离以后，据说文件大小能降到以前的1/3。
把页面中引用的js文件分离出来，把动态部分和静态部分也分离开来。

6，使用deflate压缩算法 (速度明显提升，节约3.33倍带宽)
一般来说压缩过的文件大小不到以前的30% 。
将上面分离出来的数据进行压缩(累计节约带宽10倍)。

7, 让用户尽可能多的Cache你的资源 （速度明显提升，节约3~50倍服务器资源和带宽资源）
将上面分离出来的css和不经常变动的js数据部分cache住合适的时间。(理想情况,累计节约带宽30~500倍)
。

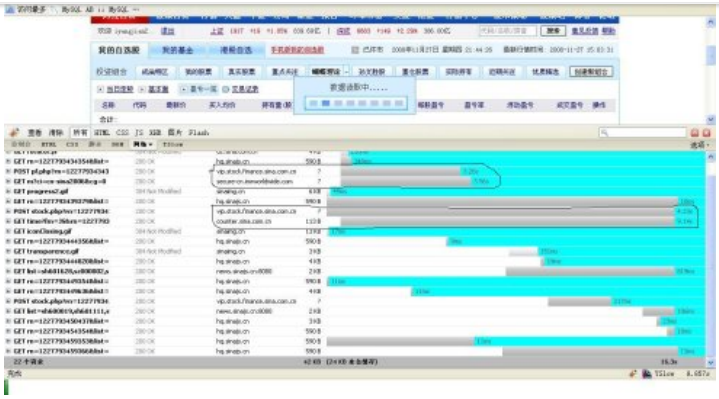
以上改进可以让速度大幅度提升的同时，服务器资源节约 5~20 倍 ，减少网络拥塞3~无数倍, 上行带宽节约1倍以上，下行带宽节约30~500倍，甚至更多。

实例分析篇

一，自选股分析
二，NBA比赛分析
三，播客分析
四，开心网分析

下面的图片都是在教育网访问的情况，我故意放大了某些缺陷，这样可以很好的模拟没有部署服务的地区对用户体验的影响。我只能针对我熟悉和了解的项目进行分析，另外还有我们经常访问的网站也会被拿来做素材分析。我们老大也说问题暴露出来，能推动解决的话也很好，大家别拍我。

一，自选股分析



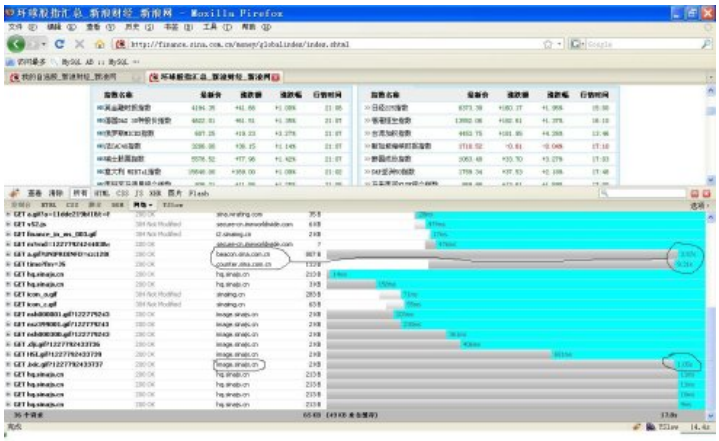
某天我终于在教育网部署了一台行情服务，兴致冲冲的回家一试，貌似没啥变化，该慢还慢。打开页面过程持续了几十秒，然后终于露出了行情，我再电击每个组合的时候就出现了上面的一幕。看了下firebug，最慢资源排名前三依次为：高效计数服务，secure-cn统计服务，动态池服务。

高效计数服务是早期我参与的项目，那时候资源有限，全部部署在了网通。
secure-cn统计服务: 这个服务不慢是不正常的，到处都嵌，还不能不嵌。
动态池数据库很牛，但在偏远地区也鞭长莫及。这个缺点比较典型:
一，没有在教育网部署。
二，没有保持长连接。
三，没有使用cahce
四，没有使用压缩
五，长达2.46K的http 请求header，捎带大量cookie，见下面。

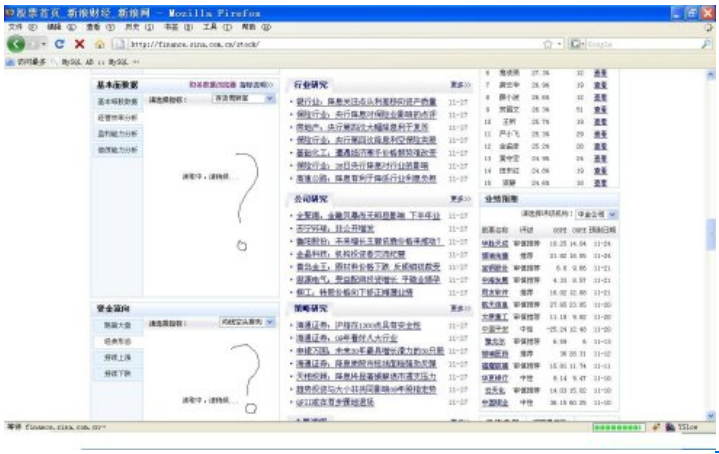
解决方法：我分析了下，下面这个数据变化很慢的，主要放一些市盈率和用户股票列表。市盈率可以通过去年的每股收益来计算，以年计，可以变通一下。用户股票列表我也好几个月没更新了,大家并不是总更新。所以这部分数据是可以被设置一个很长的cache的，如果用户更新了股票列表，我们也只需要在maxage版本号上加1就ok了。另外，用户点了一个组合，接看来也都要看几个别的组合，没有维持长连接显然不合理的。在没有部点的idc，压缩就能明显的提升响应速度，这里就没考虑。那个cookie太长点了吧，真的用的了那么长吗。

http://vip.stock.finance.sina.com.cn/portfolio/stock.php?rn=1228707043897&pid=1245111&type=complete
---
GET /portfolio/stock.php?rn=1228707043897&pid=1245111&type=complete HTTP/1.1
Host: vip.stock.finance.sina.com.cn
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN; rv:1.9.0.4) Gecko/2008102920 Firefox/3.0.4
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-cn,zh;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: gb2312,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: CurrentBar=attend; CurrentTab=state; CombinationSelected=154148; CommisionCookie=0; StampCookie=0; FeeCookie=0; BX=7t1oh653u6qvb&b=3&s=4k; SINA_NEWS_CUSTOMIZE_city=%u5317%u4EAC; userId=C7DHwoAi-ryCr69CGgyc3czekbyphdy5hcxQNhFcN6zCNe; FINA_VISITED_S=sh601988|-y?L,sh580989|W*JTP1,sh601988|-y?L,sh601988|-y?L,sh580989|W*JTP1,sh601988|-y?L,sh601988|-y?L,sh601988|-y?L,sh580989|W*JTP1; Iask2_visitID=10.217.21.44.177601199668733612; UNIPROCT=342-0-0:2; hold_sinabar_name=iyangjian2005997; UNIPROPATH=2:iyangjian2005997:0::1:|*|202.112.174.100.97191204115419966|pid:342-0-0-0-0|classad.sr/|st:25.906|et:1204118703312||hp:unkown|lb:1|*|; SINAPUID=10.217.21.64.250871201592749264; vjuids=-5600fbe60.117402dbc5e.0.42a2debdf9f46; VISITED_FANCHAN_SINA_ZHANGYQ=SINA_BEIJING; S_WC_USRTOK=SFyLe9; stat=080620160858972043653 3; MY_STOCK_LIST_2=sh600602; visited_futures=SI%7CCL%7CGC%7CCAD%7CTRB%7Cau0812%7CCC%7CPBD%7CCF907%7CNID; SINA_FINANCE=iyangjian2005997%3A1181509184%3A2; visited_funds=000011%7Csh000011%7C159902%7C160314%7C377016%7C270005%7C202009; SINA_FINANCE_SELECT_TYPE=stock; vjuid=-12b4fad5c.1174d78e8a5.0.6099c257a27eb; vjlast=1199616063; vjlast=1199616063,1228706963,10; sina_sort_default=117; SHOW_TIP_BOX=1; FINA_V_S_2=sz000609,sz000723,sh000001,sz002242,sz002274,sz000049,sz002272,sh600432,sh601186,sh601390,sh600036,sz000625; hk_visited_stocks=HSI%7C04338%7CHSCEI%7CHSCCI; visited_cfunds=050007%7Csz161010; __utma=269849203.390390911.1226996335.1226996335.1226996335.1; __utmz=269849203.1226996335.1.1.utmccn=(direct)|utmcsr=(direct)|utmcmd=(none); SINAGLOBAL=202.112.174.100.224381203683121713; Apache=202.112.174.100.771641228691763829; SessionID=e9bc0f217040ae10439d85f422f3187a; SINA_PORTFOLIO=sz000514%2Csh600729%2Csh600438%2Csh600528%2Csh600678%2Csh600877%2Csh600039%2Csh601005%2Csh600875%2Csz001696%2Csz000628%2Csh600116

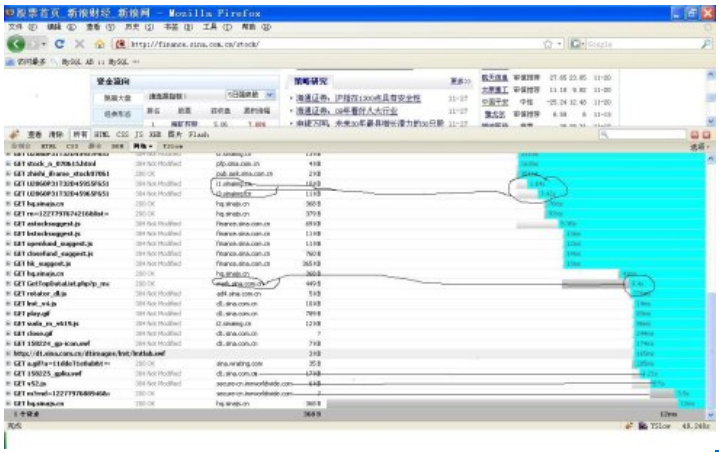
HTTP/1.x 200 OK
Date: Mon, 08 Dec 2008 03:32:52 GMT
Server: Apache
Cache-Control: no-cache
Expires: Mon, 08 Dec 2008 03:34:52 GMT
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=GBK



如果不是这几个资源的引用，这个页面的速度将非常快。



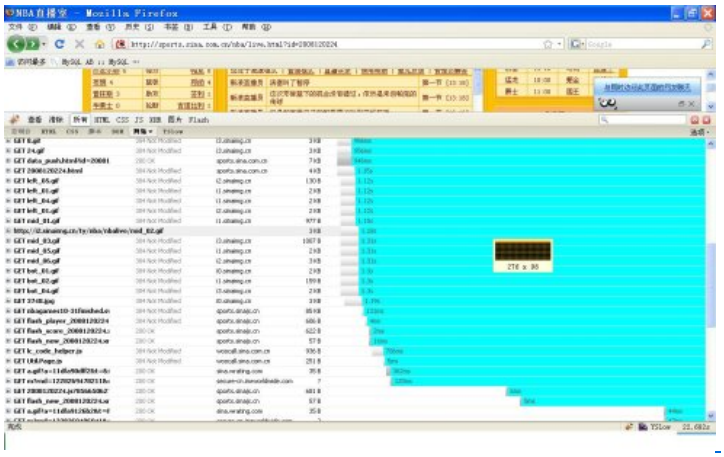
这里引用了某些未在教育网部署的服务，导致半天出不了数据。



由于引入了mark.sina.com.cn的数据导致整个页面卡在那里。引用别人数据的时候你了解过他们是怎么分布自己服务的吗？可能稍有不慎拖垮整个页面。

二，NBA比赛分析

这里的js真的有必要每次都发起请求吗？连续请求３同域个资源，为什么不维持下长连接？



这些图片的３０４响应为什么都在秒级以上？

三，播客分析

这些图片和视频由于解析错误，教育网用户被解析到广州服务器组，导致不可访问。

四，开心网分析

打开开心网，看到最多的就是人物图片，我就仅仅针对图片进行下分析：

1,浏览一个新人的页面,大概要下载30~40张小图片。使用单一的pic.kaixin001.com域名，不能提高并发，可以考虑多域名取模。
2,图片请求带了cookie，上行带宽浪费点无所谓，但是会影响响应速度和用户体验。
3, /logo/10/51/50_105146_1.jpg ,他们设置了一个比较大的maxage，通过改名来实现更新大可不必，我用我的方法更好。
4,每次点刷新页面，都会重新加载很多图片，虽然很多是304，我觉得绝大部分就不应该发这个请求。
5,他用的是ChinaCache的CDN，Server: nginx，我不知道ChinaCache对这个server修改到什么程度。统计发现这个人物小图片大都在2k左右。很多才1k多。没有必要把他们当作图片处理。尽量不产生磁盘i/o,包括fstat这样的系统调用，甚至sendfile这样的zero copy系统调用，我觉得都浪费. 同时还要保证图片更新立刻被感应到。

其他方面还有很多可以改进的，想让他们的页面响应速度上一个等级，节约更多带宽和服务器资源并非难事。

动态应用篇

一, 引子
二，总体结构图
三，系统结构综述
四，环境配置以及底层基础类库
五, Memcache & Mysql 常用场景案例
六，更多待续 .....

一, 引子

张三丰当初传授传授张无忌太极剑法的时候，刚传授完，就问是否已经忘记。直到张无忌说招式已经忘光了，才算学会。当时很不理解，现在终于明白了，招随心出，不应受到固定招式的限制。总有人问我什么样的架构是不是就好，或者某个很有名的网站用什么架构，我们就要用吗？甚至觉得直接拿个别人的配置文件

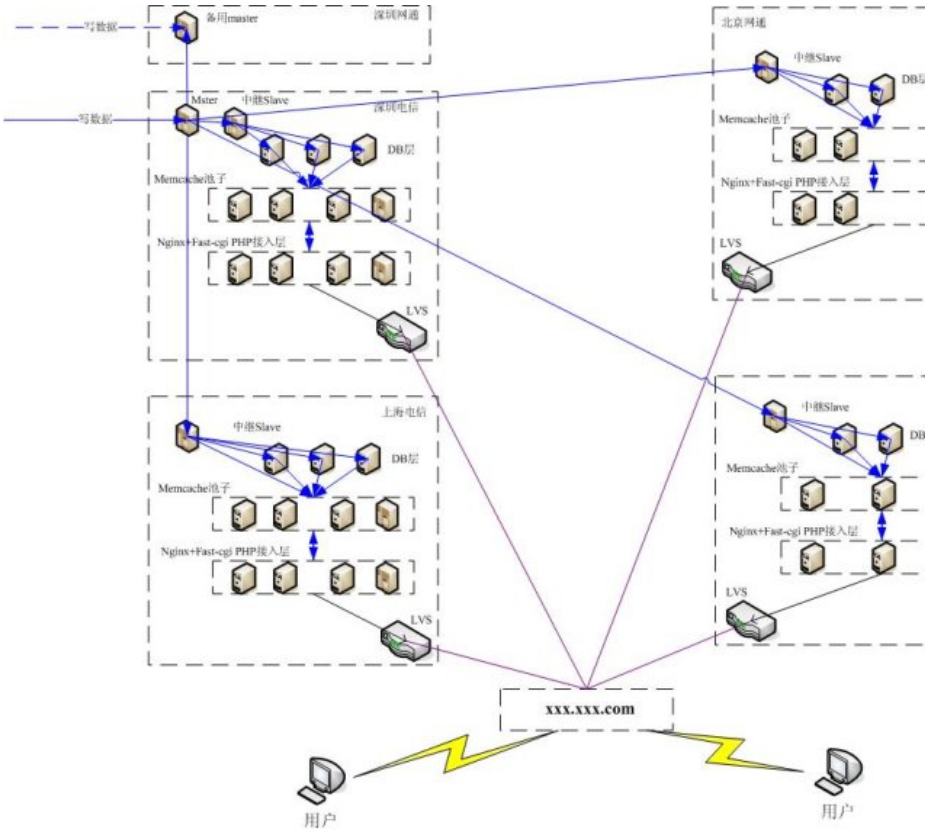
解你所处的环境，从实际需求出发，你才能做出最适合你应用的架构。

很多人对写server很感兴趣，甚至一开始就直接考虑从底层协议进行优化，我说远还没到那个程度，请找出你真正的瓶颈。优化一定是从宏观到微观的，有时候一个系统结构，业务逻辑，或策略的优化，带来的效果甚至更为可观。对于每天请求处理量小于100亿的系统，我都不建议自己去写server。如果说写高性能server的技能是我手中的那把太极剑的话，那么发挥出太极剑法的威力并不依赖于那把剑，而是取决于对心法的理解。那把剑既然可以是专用server，当然也可以是开源的那几款经典server，取决于你对设计容量，以及硬件成本，维护成本，时间等多方面的预期。架构本身就是一个取舍的过程。

现在有这样的一个项目，就拿我上一篇文章里提到的自选股来举例吧，用户可以在各个市场里创建自己的多个投资组合，然后在组合里定制自己关注的股票。描述一下环境以及需求。
环境：注册用户小几千万，同时在线预计峰值不到10万。
需求：
1，3G，IM，Mail，Web 任何一个地方以及不同的主流浏览器更新数据，其他地方立即可见。
2，北京，天津，上海，深圳，任何一个IDC数据更新，其他三个IDC立即可见。
3，各IDC附近用户，获取股票列表的平均响应速度要控制在20ms以内。
4，IDC间专线中断服务不能受影响。
5，IDC内部的相同功能的服务器，允许宕掉一半，服务完全不受影响。
6，IDC允许宕掉一到两个，受灾IDC 95%的用户服务影响不超过5分钟。
7，另外，需要开发人员能够在这个系统上快速开发，要具备易用性，良好扩展性以及移植性。

基于以上的需求构建的实现，动态应用篇侧重点主要是如何快速响应，同步更新，如何容灾，消除安全隐患，让系统更稳定，如何容易迁移和扩展应用，如何让程序员容易使用这个平台。这个系统性能不是主要关注的问题(10万同时在线，确实比较微量，何况仅有的不多的压力，也通过后面介绍的各种缓存机制转移的差不多了)，架构上采用当前主流经典架构(Nginx+PHP Fast-CGI+APC+Mysql+Memcache+LVS+Linux)，各层次之间低偶合，每一层都具备单独优化的空间, 随着用户量的增加，我再逐步推出动态应用处理并发和压力方面的文章。

二，总体结构图



(实际结构中去掉了中继slave，改成所有slave直连master，这样结构更简单，易于管理和故障恢复。)

三，系统结构综述
此系统主要分为几个低偶合的层次组装而成，具备多IDC分布的特性。从底往上依次为DB层，MC池子层，Nginx+PHP Fast-CGI层，LVS层，然后通过DNS接入用户。每一层都具备良好的扩展性以及灾备能力。

DNS:

从大的结构上来说，此系统分布在4个主要IDC，网通电信各2。 机器数量按照网通：电信 1：2的比例配置，同一运营商下的两IDC机器数量等同。这样在宕掉一个IDC的情况下，可以通过切换DNS，临时访问相

LVS层:

每个IDC的接入层通过LVS作四层负载均衡。IDC内部任何接入机器宕掉，可以通过failover机制在一分钟内自动摘除。

Nginx+PHP Fast-CGI层:

通过fpm管理PHP Fast-CGI进程，Nginx通unix域协议与fpm通信。转发 *.php的请求以及响应。使用APC作为OP代码加速器，加速php响应。PHP直接与MC池子或Mysql层进行交互。

MC池子层：

每个IDC内部的多个MC机是作为一个整体来管理的，也就是说，假如你有100个数据，4个MC机，那么每个MC上只会存25个数据，而不会每个都存100个。好处是，获得了相当于以前4倍的内存池, 增大了缓存命中率。更额外的好处是，减少了，本IDC内部各MC间数据同步的时间开销,使得本IDC任何一服务器更新的数据，同IDC内其他服务器立即可见。也使得IDC之间MC的同步更为节约，每个IDC一个池子，每个池子只需要写一份数据。

MC池子增减机器的问题：如果是传统的hash算法根据key值，将数据平均的hash到4台机器上，那么按照新的hash规则，增加一台机器后，将会有近80%的缓存失效，造成大量数据迁移。所以我们改用Consistent Hashing，首先求出memcached服务器（节点）的哈希值，并将其配置到0～2的32次方的圆上。然后用同样的方法求出存储数据的键的哈希值，并映射到圆上。然后从数据映射到的位置开始顺时针查找，将数据保存到找到的第一个服务器上。如果超过2的32次方仍然找不到服务器，就会保存到第一台memcached服务器上。它能最大限度地抑制键的重新分布。而且，有的实现还采用了虚拟节点的思想，使得分布更加均匀。由服务器台数（m）和增加的服务器台数（n）计算增加服务器后的命中率计算公式如下：(1 - n/(n+m)) * 100 ，按这个计算，增加一台机器后还将得到80%的命中率。更较幸运的是这个算法已经被php所支持，可以通过php.ini设置。

宕机重启后初始化的问题：如果初始化的数据只从一台DB上获得，那么在高峰期间必将压垮DB，所以，我们需要将这个压力，分散到本IDC的多台DB上。细节会在基础类库中进行讲述。

异地MC池子的数据同步：已经封装在基础类库中，更新MC时指定特定的参数，就会在其他异地MC池子上也进行更新。所以我们如果在北京添了一个数据，那么在深圳也会立刻看到。

DB层：

主从结构集群，主库可切换，从库互为备份。从DB宕掉一台，可以自动跳过，不影响服务，目前通过基础类库实现，以后会采用内部DNS。所有从库直接和主库相连，结构简单，方便管理。不过写的数据量相对比较小的，而且我们允许DB延迟（由MC来保证实时性，或者异地本来就可以接受短暂的延迟），这里将不是问题。

MC和DB写队列：

在专线中断时不影响写操作(读都是本地的，当然也不受影响)，而且在更新数据时能够快速返回(因为不需要远程通信)。

具体来说就是将MC的异地写，以及DB的写操作封装成直接追加写本地队列文件。每个机器上有个守护进程，每0.1秒检查一次，有则rename,然后开始将队列数据逐条往深圳数据中心post，同时每成功一条记录当前offset，以便意外宕掉后接着上次的后面处理。（优化点的可以改为批量确认，冒进点的，可以使用内存盘优化写速度）。如果线路不通，或者数据中心写失败，post程序将sleep and retry，线路恢复则继续，不会丢数据。这里有个细节需要注意，就是往数据中心同步的时候，尽量发往同一台机器，也即绑定，以保证序列的顺序，这个保证可以通过对post守护进程的pid取mod来实现，如果绑定的机器出了故障，则应该选定另外一台机器绑定，一旦检测到以前的机器好了，则应该切回来，以保证应有的负载均衡。另外，最外层的接入端也同样需要注意这个问题，我们对lvs做了会话保持，以确保同一用户在相当长的一段时间内，会访问到同一台机器。对于仅仅使用DNS轮询，又没有使用长连接的服务，如果在短时间内做数据更新，肯定会出现乱序的问题。除非你各服务器时间都很精准，而且，每个记录写的时候打上精确到ms的时间戳。

然后，深圳数据中心接收到post的数据后，根据消息类型按一定规则的命名分别存放，这样即使增加一种新的消息，也可以方便的使用队列。比如收到MC的消息后，要写三份，分别发往三个IDC，各自不受影响。DB，可以根据不同的port实例子来创建队列，免得有一个port宕了，影响其他。具体实现细节可以自己调整，同样得有个程序定期（比如每0.1秒,0.01秒也无所谓，对cpu的消耗非常小）检查队列文件存在就rename，然后往master里写。

以上则能保证，DB的master宕掉，或者专线的中断情况下，用户服务基本不受影响，只是异地的同步会延迟一些。额外收获是将来做DB升级调整的时候，可以用队列分流，分别做不同的处理。

祛除安全隐患：

不管你的系统设计的多么完美，疏忽这一条足以致命。
以上设计看起来似乎已经没什么问题了，各种容灾，异常也基本考虑到了，不幸的是，这个平台并不是仅仅给设计者自己使用，如果有个新手使用了 file_get_contents($url); 如果url所依赖的服务器负载过重，那么整个系统都有被拖垮的危险。动态应用的一个铁律就是，凡是依赖本系统外部资源的地方必须加超时限制，尽可能的减少依赖。file_get_contents的超时不是很精确，推荐使用curl的库进行封装，可以设置connect超时，也可以设置整个函数的执行时间超时。我一般会设置my_curl();函数的最大默认执行时间为一秒，因为是从内网拉取数据，一秒还拉不到，肯定有问题。另外，别忘记了mysql读服务，当某一idc的一台slave连接数满了以后，如果没设置过mysql.connect_timeout=1;那这个IDC的服务会整个被拖垮，因为默认的是60秒。（mysql的写由队列保证，不存在此问题）。

另外，我们项目还有个特殊性，即，每次拉取用户股票信息时，需要从后台专线到深圳进行身份合法性验

况下依然正常提供服务，就需要在得不到验证的情况下，要通过代理的手段到邻近的IDC去验证。同时，我们也支持另外一种验证方式，对于已经通过验证的用户，专线中断一小段时间，比如一小时，服务是可以不受影响。

总之就是尽量低偶合，少依赖，加超时。另外一个维护的安全，则由安全中心把关，我就不多说了。

四，环境配置以及底层基础类库

PHP网络版环境：每IDC之间差异化的Nginx环境变量配置，使得相同的应用程序在不同的IDC运行时，使用各自IDC内部的MC以及DB等资源。

PHP Shell版环境：

PHP Shell是指通过命令行方式执行的php脚本程序。

比如/path/php/bin/php test.php

或在php程序第一行加上

#!/path/php/bin/php

然后赋予php脚本可执行权限，使作为shell程序运行。

由于php作为shell运行时，无法继承nginx配置的环境变量。所以它必须依赖一个独立的配置文件。

由于图片里含有帐户等敏感信息，就不在此贴了。

底层基础类库：

底层基础类库，起到粘合剂的作用，将环境配置，服务器资源等全部结合起来，使得这些资源以及配置信息对上层开发人员透明，无须考虑。总的来说有以下一些功能。

1， 两环境融合，天衣无缝。 php 网络环境和shell使用同一基础类库，代码无任何一行差异。使得平时编写的php网络程序，以及类库积累，可以方便的直接用来做shell编程，进行复用。具体原理是，类库需要用到配置信息时，先通过if(isset ($_ENV["SERVER_SOFTWARE"]))变量判断自己是否网络环境，如果是就直接使用配置项比如：_SERVER["DB_stock_host"] ，若不是，则先将配置文件数据项，section名和下面的字段相加转化成 _SERVER["DB_stock_host"] = “m3306_sz_gtimg_cn” ; 跟网络环境一致后，再继续后面操作。

[DB_stock]

host = m3306_sz_gtimg_cn

类库目前除支持模拟DNS外，还直接兼容真实域名以及IP地址，方便将来进行数据迁移。之所以没有直接使用 DNS是也有历史原因的。

2， 对DB资源的封装。

在没有内部DNS的情况下，将DB读写分离，帐户选择，连接的建立（包括 何时候真正建立连接，建立长连接，还是短连接，连接的绑定，以及生命周期），负载均衡以及failvoer等封装成对用户透明的如下简单用法：

//指定以读(r) 或 写(w) 的方式打开一个库。不指定的情况下，默认是"r"方式打开。

```
$db_r =new MYSQL(“testdb”,“r”);
 <?
require_once (dirname(__FILE__).'/../mysql.php');
 $db_w =new MYSQL("test","w");
 $arr = array(  "id" => "8",  "name" => "yangjian8"  );
 if( $db_w->insert("test",$arr) )
{
      echo "query ok ...<br>";
}else
{
      echo "query failed ...<br>";
      echo "errno=$db_w->errno<br> errmsg=$db_w->errmsg<br>";
}
 echo "read .....<br>";
 $db_r =new MYSQL("test","r");
$sql = "select * from test";
```

```
{
    while ($row = mysql_fetch_array($result, MYSQL_BOTH))
    {
        printf ("id: %s name: %s<br>", $row[id], $row[name]);
    }
    $db_r->free();  //free result. if you not free it,it will auto free at the end of the php script.
}else   //if not success,you can print the error info.
{
    echo "errno=$db_r->errno<br> errmsg=$db_r->errmsg<br>";
}
```

3， 对MC池子资源的封装。

简化MC池子使用方法，支持异地MC数据同步。

```
//if rset=1,the mc data will sync to other idcs, default 0.
function set($key, $value, $flag, $expire, $r_set=0)
<?php
require_once (dirname(__FILE__).'/../memcache.php');
$mc = new MC("test");
for($i=0;$i<2;$i++)
{
    $mc->set("key".$i,"value".$i,0,100);
}
for($i=0;$i<2;$i++)
{
    echo $mc->get("key".$i);
    echo "<br>";
}
```

DB以及MC的异地写已经封装进去，对上层开发人员来说都是透明的。

注意：我在引用头文件时候，都会使用类似require_once (dirname(__FILE__).'/../mysql.php'); 的方法。我暂且管它叫动态绝对路径。它的好处是，

跟相对路径比：当一个头文件被多层引用时，目录结构又不一致，不会找不到。

也不会去搜索所有可能的目录，执行多余的fstat以及open操作。

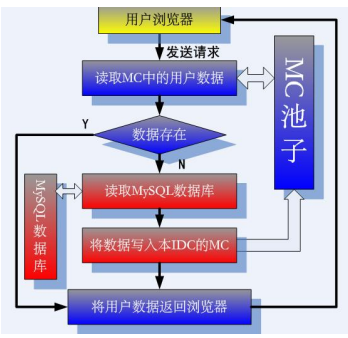
跟绝对路径相比：如果我将整个项目，包括头文件全部平移到别的目录，不需要挨个修改文件。

当然，你也可以采用另外一个方法，将本项目相关的配置信息绝对路径放在一个统一的文件里，然后通过任何一种方法引用那个文件。

五， Memcache & Mysql 常用场景案例

经典篇:

更新数据：写全局MC，然后再写DB。
读数据：先读MC，命中返回数据。不命中则读DB，更新到本地MC，然后返回数据。



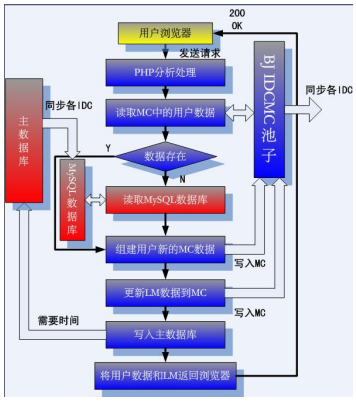
(这几个逻辑图由kinggen同学提供，看起来比文字直观多了,感谢一下)
为什么更新数据写全局MC，而读数据不命中只写本地MC？

如果更新数据和不命中的情况下都只写本地MC会有什么后果?
因为MC不会主动获得数据更新，如果更新数据不写全局，会造成其他IDC的cache在失效以前仍然是旧的。出现数据不同步的现象。

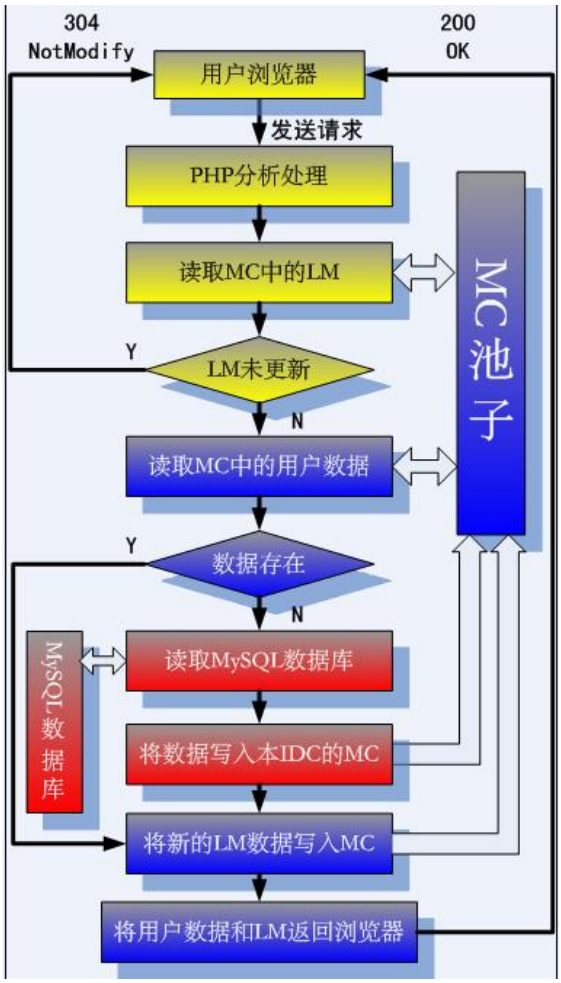
进阶篇：

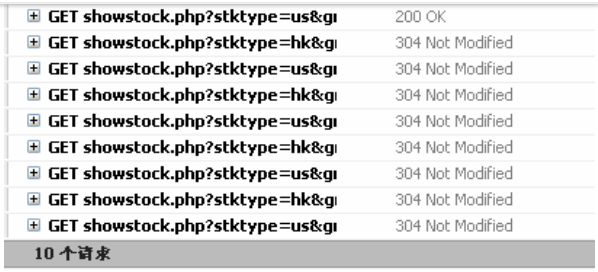
这里是在动态数据中引入静态数据的last modify特性，以使得在动态应用中可以返回HTTP 304状态。只比较最后更新时间便可以做出判断，减少后续逻辑处理以及数据内容传输，快速做出响应。对于读多，写少的项目，意义巨大。
对于js调用的部分，并不等同与刷新，要想让每次都产生请求，而且还带If-Modified-Since过去，必须加个max-age=1。只能精确到1秒。

更新数据：将数据和LM写到全局MC，然后将数据写到DB，不用把LM也写入，LM只存在于MC中。



读数据：如果MC中存在LM： 比较浏览器请求带过来的LM， 大于等于MC中LM则直接返回304。否则返回数据和最新的LM。如果MC中不存在LM：把当前的响应时间作为LM存在本地MC中，然后返回数据和此LM。





返回数据方法同经典篇里的读数据。取时间请用 $_SERVER['REQUEST_TIME']。
LM的cache时间可以设置的尽量长些，比如一个月。

PHP中动态数据使用Last-Modified加速原理详细说明：

动态应用项目中充分利用LM来加速响应，减少逻辑处理以及数据传输。
最初考虑是用etag实现，引入这一机制并不仅仅是为了节约带宽。它还用来减化应用程序逻辑。比如正常取一个数据，需要取好几个表的东西，大概消耗200ms。如果我把etag作为数据版本来用，只需要取memcache里的版本号判断一下，对于大多数用户来说，都没更新数据，就不用走后面的判断了，直接返回304状态。但是IE6里，如果同时使用gzip，又使用etag，etag就会失效。这是ie6的bug，没有遵守http 1.1。

现在使用方法，把数据最后更改时间戳作为版本。模拟静态数据使用Last-Modified 。这样做的缺陷是，单位只能精确到秒，如果一秒内做多次修改，将不能区分。不过对我们目前应用来说，精确到一秒已经足够用了，用户的动作没那么快。另外，还有一个细节，将决定这个机制能否应用在我们的项目中。我们既要使用缓存，又要其他任终端，或者浏览器通过js拿数据的时候立刻拿到最新的。大家知道，如果你使用了Last-Modified，通过js在当前浏览器下再次取数据的时候，浏览器不会发起任何请求，新数据当然无从拿到。如果能让浏览器发送请求的时候带上If-Modified-Since，又能每次都让浏览器产生请求，便能解决问题。
于是，我们通过php输出数据的时候同时使用这样的两个头信息，便达到了目的。

Cache-Control: max-age=1

Last-Modified: Tue, 11 May 2010 10:58:11 GMT

这样做的假设是：用户点一个组合查看数据，然后用户在手机上添加一个股票信息，然后用户切到了别的组合，然后又切回这个组合查看数据，这4个动作不可能在同一秒内完成。我反正是做不到那么快，超人例外。

转自：http://blog.sina.com.cn/s/articlelist_1181509184_0_1.html

扩展知识：一致哈希算法、delate压缩算法、NAT/TUN/DR

标签: Optimization

aitao
关注 - 1
粉丝 - 100
+加关注

1 0

« 上一篇：海量数据处理分析(开发角度)
» 下一篇：负载均衡与HTTP加速

posted @ 2012-09-21 21:35 aitao 阅读(1516) 评论(0) 编辑 收藏

相关博文：
· 服务器系统架构
· 【架构★我的系统架构】我的系统架构<服务器架构>
· IM服务器架构实现/QQ服务器架构剖析
· 服务器架构（二）
· 服务器架构优化

最新新闻：
· IBM成Z代人最青睐科技公司 谷歌和亚马逊分列第二三名
· 为什么我们更像是在为抖音筛选内容，而非消费内容？
· 人人车变脸：曾经想干掉黄牛，如今成了“黄牛公司”
· 张一鸣豪赌千亿营收，但字节跳动仍将面临三重难关
· 马斯克私有化推文影响犹在 特斯拉还在应付股东集体诉讼案
» 更多新闻...