关于性能优化这是一个比较大的话题，在《[由12306.cn谈谈网站性能技术](http://coolshell.cn/articles/6470.html)》中我从业务和设计上说过一些可用的技术以及那些技术的优缺点，今天，想从一些技术细节上谈谈性能优化，主要是一些代码级别的技术和方法。**本文的东西是我的一些经验和知识，并不一定全对，希望大家指正和补充**。

在开始这篇文章之前，大家可以移步去看一下酷壳以前发表的《[代码优化概要](http://coolshell.cn/articles/2967.html)》，这篇文章基本上告诉你——**要进行优化，先得找到性能瓶颈**！ 但是在讲如何定位系统性能瓶劲之前，请让我讲一下系统性能的定义和测试，因为没有这两件事，后面的定位和优化无从谈起。

**一、系统性能定义**

让我们先来说说如何什么是系统性能。这个定义非常关键，如果我们不清楚什么是系统性能，那么我们将无法定位之。我见过很多朋友会觉得这很容易，但是仔细一问，其实他们并没有一个比较系统的方法，所以，在这里我想告诉大家如何系统地来定位性能。 总体来说，系统性能就是两个事：

1. **Throughput** ，吞吐量。也就是每秒钟可以处理的请求数，任务数。
2. **Latency**， 系统延迟。也就是系统在处理一个请求或一个任务时的延迟。

一般来说，一个系统的性能受到这两个条件的约束，缺一不可。比如，我的系统可以顶得住一百万的并发，但是系统的延迟是2分钟以上，那么，这个一百万的负载毫无意义。系统延迟很短，但是吞吐量很低，同样没有意义。所以，一个好的系统的性能测试必然受到这两个条件的同时作用。 有经验的朋友一定知道，这两个东西的一些关系：

* **Throughput越大，Latency会越差。**因为请求量过大，系统太繁忙，所以响应速度自然会低。
* **Latency越好，能支持的Throughput就会越高。**因为Latency短说明处理速度快，于是就可以处理更多的请求。

**二、系统性能测试**

经过上述的说明，我们知道要测试系统的性能，需要我们收集系统的Throughput和Latency这两个值。

* 首先，**需要定义Latency这个值**，比如说，对于网站系统响应时间必需是5秒以内（对于某些实时系统可能需要定义的更短，比如5ms以内，这个更根据不同的业务来定义）
* 其次，**开发性能测试工具**，一个工具用来制造高强度的Throughput，另一个工具用来测量Latency。对于第一个工具，你可以参考一下“[十个免费的Web压力测试工具](http://coolshell.cn/articles/2589.html)”，关于如何测量Latency，你可以在代码中测量，但是这样会影响程序的执行，而且只能测试到程序内部的Latency，真正的Latency是整个系统都算上，包括操作系统和网络的延时，你可以使用Wireshark来抓网络包来测量。这两个工具具体怎么做，这个还请大家自己思考去了。
* 最后，**开始性能测试**。你需要不断地提升测试的Throughput，然后观察系统的负载情况，如果系统顶得住，那就观察Latency的值。这样，你就可以找到系统的最大负载，并且你可以知道系统的响应延时是多少。

再多说一些，

* 关于Latency，如果吞吐量很少，这个值估计会非常稳定，当吞吐量越来越大时，系统的Latency会出现非常剧烈的抖动，所以，我们在测量Latency的时候，我们需要注意到Latency的分布，也就是说，有百分之几的在我们允许的范围，有百分之几的超出了，有百分之几的完全不可接受。也许，平均下来的Latency达标了，但是其中仅有50%的达到了我们可接受的范围。那也没有意义。
* 关于性能测试，我们还需要定义一个时间段。比如：在某个吞吐量上持续15分钟。因为当负载到达的时候，系统会变得不稳定，当过了一两分钟后，系统才会稳定。另外，也有可能是，你的系统在这个负载下前几分钟还表现正常，然后就不稳定了，甚至垮了。所以，需要这么一段时间。这个值，我们叫做峰值极限。
* 性能测试还需要做Soak Test，也就是在某个吞吐量下，系统可以持续跑一周甚至更长。这个值，我们叫做系统的正常运行的负载极限。

性能测试有很多很复要的东西，比如：burst test等。 这里不能一一详述，这里只说了一些和性能调优相关的东西。总之，性能测试是一细活和累活。

**三、定位性能瓶颈**

有了上面的铺垫，我们就可以测试到到系统的性能了，再调优之前，我们先来说说如何找到性能的瓶颈。我见过很多朋友会觉得这很容易，但是仔细一问，其实他们并没有一个比较系统的方法。

**3.1）查看操作系统负载**

首先，当我们系统有问题的时候，我们不要急于去调查我们代码，这个毫无意义。我们首要需要看的是操作系统的报告。看看操作系统的CPU利用率，看看内存使用率，看看操作系统的IO，还有网络的IO，网络链接数，等等。Windows下的perfmon是一个很不错的工具，Linux下也有很多相关的命令和工具，比如：[SystemTap](http://sourceware.org/systemtap/)，[LatencyTOP](https://latencytop.org/)，vmstat, sar, iostat, top, tcpdump等等 。通过观察这些数据，我们就可以知道我们的软件的性能基本上出在哪里。比如：

1）先看CPU利用率，如果CPU利用率不高，但是系统的Throughput和Latency上不去了，这说明我们的程序并没有忙于计算，而是忙于别的一些事，比如IO。（另外，CPU的利用率还要看内核态的和用户态的，内核态的一上去了，整个系统的性能就下来了。而对于多核CPU来说，CPU 0 是相当关键的，如果CPU 0的负载高，那么会影响其它核的性能，因为CPU各核间是需要有调度的，这靠CPU0完成）

2）然后，我们可以看一下IO大不大，IO和CPU一般是反着来的，CPU利用率高则IO不大，IO大则CPU就小。关于IO，我们要看三个事，一个是磁盘文件IO，一个是驱动程序的IO（如：网卡），一个是内存换页率。这三个事都会影响系统性能。

3）然后，查看一下网络带宽使用情况，在Linux下，你可以使用iftop, iptraf, ntop, tcpdump这些命令来查看。或是用Wireshark来查看。

4）如果CPU不高，IO不高，内存使用不高，网络带宽使用不高。但是系统的性能上不去。这说明你的程序有问题，比如，你的程序被阻塞了。可能是因为等那个锁，可能是因为等某个资源，或者是在切换上下文。

**通过了解操作系统的性能，我们才知道性能的问题，比如：带宽不够，内存不够，TCP缓冲区不够，等等，很多时候，不需要调整程序的，只需要调整一下硬件或操作系统的配置就可以了**。

**3.2）使用Profiler测试**

接下来，我们需要使用性能检测工具，也就是使用某个Profiler来差看一下我们程序的运行性能。如：Java的JProfiler/TPTP/CodePro Profiler，GNU的gprof，IBM的PurifyPlus，Intel的VTune，AMD的CodeAnalyst，还有Linux下的OProfile/perf，后面两个可以让你对你的代码优化到CPU的微指令级别，如果你关心CPU的L1/L2的缓存调优，那么你需要考虑一下使用VTune。 使用这些Profiler工具，可以让你程序中各个模块函数甚至指令的很多东西，如：**运行的时间** ，**调用的次数**，**CPU的利用率**，等等。这些东西对我们来说非常有用。

我们重点观察运行时间最多，调用次数最多的那些函数和指令。这里注意一下，对于调用次数多但是时间很短的函数，你可能只需要轻微优化一下，你的性能就上去了（比如：某函数一秒种被调用100万次，你想想如果你让这个函数提高0.01毫秒的时间 ，这会给你带来多大的性能）

使用Profiler有个问题我们需要注意一下，因为Profiler会让你的程序运行的性能变低，像PurifyPlus这样的工具会在你的代码中插入很多代码，会导致你的程序运行效率变低，从而没发测试出在高吞吐量下的系统的性能，对此，一般有两个方法来定位系统瓶颈：

1）在你的代码中自己做统计，使用微秒级的计时器和函数调用计算器，每隔10秒把统计log到文件中。

2）分段注释你的代码块，让一些函数空转，做Hard Code的Mock，然后再测试一下系统的Throughput和Latency是否有质的变化，如果有，那么被注释的函数就是性能瓶颈，再在这个函数体内注释代码，直到找到最耗性能的语句。

最后再说一点，**对于性能测试，不同的Throughput会出现不同的测试结果，不同的测试数据也会有不同的测试结果。所以，用于性能测试的数据非常重要，性能测试中，我们需要观测试不同Throughput的结果**。

**四、常见的系统瓶颈**

下面这些东西是我所经历过的一些问题，也许并不全，也许并不对，大家可以补充指正，我**纯属抛砖引玉**。关于系统架构方面的性能调优，大家可移步看一下《[由12306.cn谈谈网站性能技术](http://coolshell.cn/articles/6470.html)》，关于Web方面的一些性能调优的东西，大家可以看看《[Web开发中需要了解的东西](http://coolshell.cn/articles/6043.html)》一文中的性能一章。我在这里就不再说设计和架构上的东西了。

一般来说，性能优化也就是下面的几个策略：

* **用空间换时间**。各种cache如CPU L1/L2/RAM到硬盘，都是用空间来换时间的策略。这样策略基本上是把计算的过程一步一步的保存或缓存下来，这样就不用每次用的时候都要再计算一遍，比如数据缓冲，CDN，等。这样的策略还表现为冗余数据，比如数据镜象，负载均衡什么的。
* **用时间换空间**。有时候，少量的空间可能性能会更好，比如网络传输，如果有一些压缩数据的算法（如前些天说的“[Huffman 编码压缩算法](http://coolshell.cn/articles/7459.html)” 和 “[rsync 的核心算法](http://coolshell.cn/articles/7425.html)”），这样的算法其实很耗时，但是因为瓶颈在网络传输，所以用时间来换空间反而能省时间。
* **简化代码**。最高效的程序就是不执行任何代码的程序，所以，代码越少性能就越高。关于代码级优化的技术大学里的教科书有很多示例了。如：减少循环的层数，减少递归，在循环中少声明变量，少做分配和释放内存的操作，尽量把循环体内的表达式抽到循环外，条件表达的中的多个条件判断的次序，尽量在程序启动时把一些东西准备好，注意函数调用的开销（栈上开销），注意面向对象语言中临时对象的开销，小心使用异常（不要用异常来检查一些可接受可忽略并经常发生的错误），…… 等等，等等，这连东西需要我们非常了解编程语言和常用的库。
* **并行处理**。如果CPU只有一个核，你要玩多进程，多线程，对于计算密集型的软件会反而更慢（因为操作系统调度和切换开销很大），CPU的核多了才能真正体现出多进程多线程的优势。并行处理需要我们的程序有Scalability，不能水平或垂直扩展的程序无法进行并行处理。从架构上来说，这表再为——是否可以做到不改代码只是加加机器就可以完成性能提升？

总之，**根据2：8原则来说，20%的代码耗了你80%的性能，找到那20%的代码，你就可以优化那80%的性能**。 下面的一些东西都是我的一些经验，我只例举了一些最有价值的性能调优的的方法，供你参考，也欢迎补充。

**4.1）算法调优**。算法非常重要，好的算法会有更好的性能。举几个我经历过的项目的例子，大家可以感觉一下。

* 一个是**过滤算法**，系统需要对收到的请求做过滤，我们把可以被filter in/out的东西配置在了一个文件中，原有的过滤算法是遍历过滤配置，后来，我们找到了一种方法可以对这个过滤配置进行排序，这样就可以用二分折半的方法来过滤，系统性能增加了50%。
* 一个是**哈希算法**。计算哈希算法的函数并不高效，一方面是计算太费时，另一方面是碰撞太高，碰撞高了就跟单向链表一个性能（可参看[Hash Collision DoS 问题](http://coolshell.cn/articles/6424.html)）。我们知道，算法都是和需要处理的数据很有关系的，就算是被大家所嘲笑的“冒泡排序”在某些情况下（大多数数据是排好序的）其效率会高于所有的排序算法。哈希算法也一样，广为人知的哈希算法都是用英文字典做测试，但是我们的业务在数据有其特殊性，所以，对于还需要根据自己的数据来挑选适合的哈希算法。对于我以前的一个项目，公司内某牛人给我发来了一个哈希算法，结果让我们的系统性能上升了150%。（关于各种哈希算法，你一定要看看[StackExchange上的这篇关于各种hash算法的文章](http://programmers.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed/145633#145633) ）
* **分而治之和预处理**。以前有一个程序为了生成月报表，每次都需要计算很长的时间，有时候需要花将近一整天的时间。于是我们把我们找到了一种方法可以把这个算法发成增量式的，也就是说我每天都把当天的数据计算好了后和前一天的报表合并，这样可以大大的节省计算时间，每天的数据计算量只需要20分钟，但是如果我要算整个月的，系统则需要10个小时以上（SQL语句在大数据量面前性能成级数性下降）。这种分而治之的思路在大数据面前对性能有很帮助，就像merge排序一样。SQL语句和数据库的性能优化也是这一策略，如：使用嵌套式的Select而不是笛卡尔积的Select，使用视图，等等。

**4.2）代码调优**。从我的经验上来说，代码上的调优有下面这几点：

* **字符串操作**。这是最费系统性能的事了，无论是strcpy, strcat还是strlen，最需要注意的是字符串子串匹配。所以，能用整型最好用整型。举几个例子，第一个例子是N年前做银行的时候，我的同事喜欢把日期存成字符串（如：2012-05-29 08:30:02），我勒个去，一个select  where between语句相当耗时。另一个例子是，我以前有个同事把一些状态码用字符串来处理，他的理由是，这样可以在界面上直接显示，后来性能调优的时候，我把这些状态码全改成整型，然后用位操作查状态，因为有一个每秒钟被调用了150K次的函数里面有三处需要检查状态，经过改善以后，整个系统的性能上升了30%左右。还有一个例子是，我以前从事的某个产品编程规范中有一条是要在每个函数中把函数名定义出来，如：const char fname[]=”functionName()”, 这是为了好打日志，但是为什么不声明成 static类型的呢？
* **多线程调优**。有人说，thread is evil，这个对于系统性能在某些时候是个问题。因为多线程瓶颈就在于互斥和同步的锁上，以及线程上下文切换的成本，怎么样的少用锁或不用锁是根本（比如：[多版本并发控制(MVCC)在分布式系统中的应用](http://coolshell.cn/articles/6790.html) 中说的乐观锁可以解决性能问题），此外，还有读写锁也可以解决大多数是读操作的并发的性能问题。这里多说一点在C++中，我们可能会使用线程安全的智能指针AutoPtr或是别的一些容器，只要是线程安全的，其不管三七二十一都要上锁，上锁是个成本很高的操作，使用AutoPtr会让我们的系统性能下降得很快，如果你可以保证不会有线程并发问题，那么你应该不要用AutoPtr。我记得我上次我们同事去掉智能指针的引用计数，让系统性能提升了50%以上。对于Java对象的引用计数，如果我猜的没错的话，到处都是锁，所以，Java的性能问题一直是个问题。另外，线程不是越多越好，线程间的调度和上下文切换也是很夸张的事，尽可能的在一个线程里干，尽可能的不要同步线程。这会让你有很多的性能。
* **内存分配**。不要小看程序的内存分配。malloc/realloc/calloc这样的系统调非常耗时，尤其是当内存出现碎片的时候。我以前的公司出过这样一个问题——在用户的站点上，我们的程序有一天不响应了，用GDB跟进去一看，系统hang在了malloc操作上，20秒都没有返回，重启一些系统就好了。这就是内存碎片的问题。这就是为什么很多人抱怨STL有严重的内存碎片的问题，因为太多的小内存的分配释放了。有很多人会以为用内存池可以解决这个问题，但是实际上他们只是重新发明了Runtime-C或操作系统的内存管理机制，完全于事无补。当然解决内存碎片的问题还是通过内存池，具体来说是一系列不同尺寸的内存池（这个留给大家自己去思考）。当然，少进行动态内存分配是最好的。说到内存池就需要说一下池化技术。比如线程池，连接池等。池化技术对于一些短作业来说（如http服务） 相当相当的有效。这项技术可以减少链接建立，线程创建的开销，从而提高性能。
* **异步操作**。我们知道Unix下的文件操作是有block和non-block的方式的，像有些系统调用也是block式的，如：Socket下的select，Windows下的WaitforObject之类的，如果我们的程序是同步操作，那么会非常影响性能，我们可以改成异步的，但是改成异步的方式会让你的程序变复杂。异步方式一般要通过队列，要注间队列的性能问题，另外，异步下的状态通知通常是个问题，比如消息事件通知方式，有callback方式，等，这些方式同样可能会影响你的性能。但是通常来说，异步操作会让性能的吞吐率有很大提升（Throughput），但是会牺牲系统的响应时间（latency）。这需要业务上支持。
* **语言和代码库**。我们要熟悉语言以及所使用的函数库或类库的性能。比如：STL中的很多容器分配了内存后，那怕你删除元素，内存也不会回收，其会造成内存泄露的假像，并可能造成内存碎片问题。再如，STL某些容器的size()==0  和 empty()是不一样的，因为，size()是O(n)复杂度，empty()是O(1)的复杂度，这个要小心。Java中的JVM调优需要使用的这些参数：-Xms -Xmx -Xmn -XX:SurvivorRatio -XX:MaxTenuringThreshold，还需要注意JVM的GC，GC的霸气大家都知道，尤其是full GC（还整理内存碎片），他就像“恐龙特级克赛号”一样，他运行的时候，整个世界的时间都停止了。

**4.3）网络调优**

关于网络调优，尤其是TCP Tuning（你可以以这两个关键词在网上找到很多文章），这里面有很多很多东西可以说。看看Linux下TCP/IP的那么多参数就知道了（顺便说一下，你也许不喜欢Linux，但是你不能否认Linux给我们了很多可以进行内核调优的权力）。强烈建议大家看看《[TCP/IP 详解 卷1:协议](http://book.douban.com/subject/1088054/)》这本书。我在这里只讲一些概念上的东西。

**A） TCP调优**

我们知道TCP链接是有很多开销的，一个是会占用文件描述符，另一个是会开缓存，一般来说一个系统可以支持的TCP链接数是有限的，我们需要清楚地认识到TCP链接对系统的开销是很大的。正是因为TCP是耗资源的，所以，很多攻击都是让你系统上出现大量的TCP链接，把你的系统资源耗尽。比如著名的SYNC Flood攻击。

所以，我们要注意配置KeepAlive参数，这个参数的意思是定义一个时间，如果链接上没有数据传输，系统会在这个时间发一个包，如果没有收到回应，那么TCP就认为链接断了，然后就会把链接关闭，这样可以回收系统资源开销。（注：HTTP层上也有KeepAlive参数）对于像HTTP这样的短链接，设置一个1-2分钟的keepalive非常重要。这可以在一定程度上防止DoS攻击。有下面几个参数（下面这些参数的值仅供参考）：

|  |  |
| --- | --- |
| 1  2  3 | net.ipv4.tcp\_keepalive\_probes = 5  net.ipv4.tcp\_keepalive\_intvl = 20  net.ipv4.tcp\_fin\_timeout = 30 |

对于TCP的TIME\_WAIT这个状态，主动关闭的一方进入TIME\_WAIT状态，TIME\_WAIT状态将持续2个MSL(Max Segment Lifetime)，默认为4分钟，TIME\_WAIT状态下的资源不能回收。有大量的TIME\_WAIT链接的情况一般是在HTTP服务器上。对此，有两个参数需要注意，

|  |  |
| --- | --- |
| 1  2 | net.ipv4.tcp\_tw\_reuse=1  net.ipv4.tcp\_tw\_recycle=1 |

前者表示重用TIME\_WAIT，后者表示回收TIME\_WAIT的资源。

TCP还有一个重要的概念叫RWIN（TCP Receive Window Size），这个东西的意思是，我一个TCP链接在没有向Sender发出ack时可以接收到的最大的数据包。为什么这个很重要？因为如果Sender没有收到Receiver发过来ack，Sender就会停止发送数据并会等一段时间，如果超时，那么就会重传。这就是为什么TCP链接是可靠链接的原因。重传还不是最严重的，如果有丢包发生的话，TCP的带宽使用率会马上受到影响（会盲目减半），再丢包，再减半，然后如果不丢包了，就逐步恢复。相关参数如下：

|  |  |
| --- | --- |
| 1  2  3  4 | net.core.wmem\_default = 8388608  net.core.rmem\_default = 8388608  net.core.rmem\_max = 16777216  net.core.wmem\_max = 16777216 |

一般来说，理论上的RWIN应该设置成：吞吐量  \* 回路时间。Sender端的buffer应该和RWIN有一样的大小，因为Sender端发送完数据后要等Receiver端确认，如果网络延时很大，buffer过小了，确认的次数就会多，于是性能就不高，对网络的利用率也就不高了。也就是说，对于延迟大的网络，我们需要大的buffer，这样可以少一点ack，多一些数据，对于响应快一点的网络，可以少一些buffer。因为，如果有丢包（没有收到ack），buffer过大可能会有问题，因为这会让TCP重传所有的数据，反而影响网络性能。（当然，网络差的情况下，就别玩什么高性能了） 所以，高性能的网络重要的是要让网络丢包率非常非常地小（基本上是用在LAN里），如果网络基本是可信的，这样用大一点的buffer会有更好的网络传输性能（来来回回太多太影响性能了）。

另外，我们想一想，如果网络质量非常好，基本不丢包，而业务上我们不怕偶尔丢几个包，如果是这样的话，那么，我们为什么不用速度更快的UDP呢？你想过这个问题了吗？

**B）UDP调优**

说到UDP的调优，有一些事我想重点说一样，那就是MTU——最大传输单元（其实这对TCP也一样，因为这是链路层上的东西）。所谓最大传输单元，你可以想像成是公路上的公交车，假设一个公交车可以最多坐70人，带宽就像是公路的车道数一样，如果一条路上最多可以容下100辆公交车，那意味着我最多可以运送7000人，但是如果公交车坐不满，比如平均每辆车只有20人，那么我只运送了2000人，于是我公路资源（带宽资源）就被浪费了。 所以，我们对于一个UDP的包，我们要尽量地让他大到MTU的最大尺寸再往网络上传，这样可以最大化带宽利用率。对于这个MTU，以太网是1500字节，光纤是4352字节，802.11无线网是7981。但是，当我们用TCP/UDP发包的时候，我们的有效负载Payload要低于这个值，因为IP协议会加上20个字节，UDP会加上8个字节（TCP加的更多），所以，一般来说，你的一个UDP包的最大应该是1500-8-20=1472，这是你的数据的大小。当然，如果你用光纤的话， 这个值就可以更大一些。（顺便说一下，对于某些NB的千光以态网网卡来说，在网卡上，网卡硬件如果发现你的包的大小超过了MTU，其会帮你做fragment，到了目标端又会帮你做重组，这就不需要你在程序中处理了）

再多说一下，使用Socket编程的时候，你可以使用setsockopt() 设置 SO\_SNDBUF/SO\_RCVBUF 的大小，TTL和KeepAlive这些关键的设置，当然，还有很多，具体你可以查看一下Socket的手册。

最后说一点，UDP还有一个最大的好处是multi-cast多播，这个技术对于你需要在内网里通知多台结点时非常方便和高效。而且，多播这种技术对于机会的水平扩展（需要增加机器来侦听多播信息）也很有利。

**C）网卡调优**

对于网卡，我们也是可以调优的，这对于千兆以及网网卡非常必要，在Linux下，我们可以用ifconfig查看网上的统计信息，如果我们看到overrun上有数据，我们就可能需要调整一下txqueuelen的尺寸（一般默认为1000），我们可以调大一些，如：ifconfig eth0 txqueuelen 5000。Linux下还有一个命令叫：ethtool可以用于设置网卡的缓冲区大小。在Windows下，我们可以在网卡适配器中的高级选项卡中调整相关的参数（如：Receive Buffers, Transmit Buffer等，不同的网卡有不同的参数）。把Buffer调大对于需要大数据量的网络传输非常有效。

**D）其它网络性能**

关于多路复用技术，也就是用一个线程来管理所有的TCP链接，有三个系统调用要重点注意：一个是select，这个系统调用只支持上限1024个链接，第二个是poll，其可以突破1024的限制，但是select和poll本质上是使用的轮询机制，轮询机制在链接多的时候性能很差，因主是O(n)的算法，所以，epoll出现了，epoll是操作系统内核支持的，仅当在链接活跃时，操作系统才会callback，这是由操作系统通知触发的，但其只有Linux Kernel 2.6以后才支持（准确说是2.5.44中引入的），当然，如果所有的链接都是活跃的，过多的使用epoll\_ctl可能会比轮询的方式还影响性能，不过影响的不大。

另外，关于一些和DNS Lookup的系统调用要小心，比如：gethostbyaddr/gethostbyname，这个函数可能会相当的费时，因为其要到网络上去找域名，因为DNS的递归查询，会导致严重超时，而又不能通过设置什么参数来设置time out，对此你可以通过配置hosts文件来加快速度，或是自己在内存中管理对应表，在程序启动时查好，而不要在运行时每次都查。另外，在多线程下面，gethostbyname会一个更严重的问题，就是如果有一个线程的gethostbyname发生阻塞，其它线程都会在gethostbyname处发生阻塞，这个比较变态，要小心。（你可以试试GNU的gethostbyname\_r()，这个的性能要好一些） 这种到网上找信息的东西很多，比如，如果你的Linux使用了NIS，或是NFS，某些用户或文件相关的系统调用就很慢，所以要小心。

**4.4）系统调优**

**A）I/O模型**

前面说到过select/poll/epoll这三个系统调用，我们都知道，Unix/Linux下把所有的设备都当成文件来进行I/O，所以，那三个操作更应该算是I/O相关的系统调用。说到  I/O模型，这对于我们的I/O性能相当重要，我们知道，Unix/Linux经典的I/O方式是（关于Linux下的I/O模型，大家可以读一下这篇文章《[使用异步I/O大大提高性能](http://www.ibm.com/developerworks/cn/linux/l-async/)》）：

第一种，同步阻塞式I/O，这个不说了。

第二种，同步无阻塞方式。其通过fctnl设置 O\_NONBLOCK 来完成。

第三种，对于select/poll/epoll这三个是I/O不阻塞，但是在事件上阻塞，算是：I/O异步，事件同步的调用。

第四种，AIO方式。这种I/O 模型是一种处理与 I/O 并行的模型。I/O请求会立即返回，说明请求已经成功发起了。在后台完成I/O操作时，向应用程序发起通知，通知有两种方式：一种是产生一个信号，另一种是执行一个基于线程的回调函数来完成这次 I/O 处理过程。

第四种因为没有任何的阻塞，无论是I/O上，还是事件通知上，所以，其可以让你充分地利用CPU，比起第二种同步无阻塞好处就是，第二种要你一遍一遍地去轮询。Nginx之所所以高效，是其使用了epoll和AIO的方式来进行I/O的。

再说一下Windows下的I/O模型，

a）一个是WriteFile系统调用，这个系统调用可以是同步阻塞的，也可以是同步无阻塞的，关于看文件是不是以Overlapped打开的。关于同步无阻塞，需要设置其最后一个参数Overlapped，微软叫Overlapped I/O，你需要WaitForSingleObject才能知道有没有写完成。这个系统调用的性能可想而知。

b）另一个叫WriteFileEx的系统调用，其可以实现异步I/O，并可以让你传入一个callback函数，等I/O结束后回调之， 但是这个回调的过程Windows是把callback函数放到了APC（[Asynchronous Procedure Calls](http://msdn.microsoft.com/en-us/library/windows/desktop/ms681951(v=vs.85).aspx)）的队列中，然后，只用当应用程序当前线程成为可被通知状态（Alterable）时，才会被回调。只有当你的线程使用了这几个函数时[WaitForSingleObjectEx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms687036(v=vs.85).aspx), [WaitForMultipleObjectsEx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms687028(v=vs.85).aspx), [MsgWaitForMultipleObjectsEx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms684245(v=vs.85).aspx), [SignalObjectAndWait](http://msdn.microsoft.com/en-us/library/windows/desktop/ms686293(v=vs.85).aspx) 和 [SleepEx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms686307(v=vs.85).aspx)，线程才会成为Alterable状态。可见，这个模型，还是有wait，所以性能也不高。

c）然后是IOCP – IO Completion Port，IOCP会把I/O的结果放在一个队列中，但是，侦听这个队列的不是主线程，而是专门来干这个事的一个或多个线程去干（老的平台要你自己创建线程，新的平台是你可以创建一个线程池）。IOCP是一个线程池模型。这个和Linux下的AIO模型比较相似，但是实现方式和使用方式完全不一样。

当然，真正提高I/O性能方式是把和外设的I/O的次数降到最低，最好没有，所以，对于读来说，内存cache通常可以从质上提升性能，因为内存比外设快太多了。对于写来说，cache住要写的数据，少写几次，但是cache带来的问题就是实时性的问题，也就是latency会变大，我们需要在写的次数上和相应上做权衡。

**B）多核CPU调优**

关于CPU的多核技术，我们知道，CPU0是很关键的，如果0号CPU被用得过狠的话，别的CPU性能也会下降，因为CPU0是有调整功能的，所以，我们不能任由操作系统负载均衡，因为我们自己更了解自己的程序，所以，我们可以手动地为其分配CPU核，而不会过多地占用CPU0，或是让我们关键进程和一堆别的进程挤在一起。

* 对于Windows来说，我们可以通过“任务管理器”中的“进程”而中右键菜单中的“设置相关性……”（Set Affinity…）来设置并限制这个进程能被运行在哪些核上。
* 对于Linux来说，可以使用taskset命令来设置（你可以通过安装schedutils来安装这个命令：apt-get install schedutils）

多核CPU还有一个技术叫[NUMA](http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access)技术（Non-Uniform Memory Access）。传统的多核运算是使用SMP(Symmetric Multi-Processor )模式，多个处理器共享一个集中的存储器和I/O总线。于是就会出现一致存储器访问的问题，一致性通常意味着性能问题。NUMA模式下，处理器被划分成多个node， 每个node有自己的本地存储器空间。关于NUMA的一些技术细节，你可以查看一下这篇文章《[Linux 的 NUMA 技术](http://www.ibm.com/developerworks/cn/linux/l-numa/index.html)》，在Linux下，对NUMA调优的命令是：**numactl**。如下面的命令：（指定命令“myprogram arg1 arg2”运行在node 0 上，其内存分配在node 0 和 1上）

|  |  |
| --- | --- |
| 1 | numactl --cpubind=0 --membind=0,1 myprogram arg1 arg2 |

当然，上面这个命令并不好，因为内存跨越了两个node，这非常不好。最好的方式是只让程序访问和自己运行一样的node，如：

|  |  |
| --- | --- |
| 1 | $ numactl --membind 1 --cpunodebind 1 --localalloc myapplication |

**C）文件系统调优**

关于文件系统，因为文件系统也是有cache的，所以，为了让文件系统有最大的性能。首要的事情就是分配足够大的内存，这个非常关键，在Linux下可以使用free命令来查看 free/used/buffers/cached，理想来说，buffers和cached应该有40%左右。然后是一个快速的硬盘控制器，SCSI会好很多。最快的是Intel SSD 固态硬盘，速度超快，但是写次数有限。

接下来，我们就可以调优文件系统配置了，对于Linux的Ext3/4来说，几乎在所有情况下都有所帮助的一个参数是关闭文件系统访问时间，在/etc/fstab下看看你的文件系统 有没有noatime参数（一般来说应该有），还有一个是dealloc，它可以让系统在最后时刻决定写入文件发生时使用哪个块，可优化这个写入程序。还要注间一下三种日志模式：data=journal、data=ordered和data=writeback。默认设置data=ordered提供性能和防护之间的最佳平衡。

当然，对于这些来说，ext4的默认设置基本上是最佳优化了。

这里介绍一个Linux下的查看I/O的命令—— iotop，可以让你看到各进程的磁盘读写的负载情况。

其它还有一些关于NFS、XFS的调优，大家可以上google搜索一些相关优化的文章看看。关于各文件系统，大家可以看一下这篇文章——《[Linux日志文件系统及性能分析](http://www.ibm.com/developerworks/cn/linux/l-jfs/)》

**4.5）数据库调优**

数据库调优并不是我的强项，我就仅用我非常有限的知识说上一些吧。注意，下面的这些东西并不一定正确，因为在不同的业务场景，不同的数据库设计下可能会得到完全相反的结论，所以，我仅在这里做一些一般性的说明，具体问题还要具体分析。

**A）数据库引擎调优**

我对数据库引擎不是熟，但是有几个事情我觉得是一定要去了解的。

* **数据库的锁的方式**。这个非常非常地重要。并发情况下，锁是非常非常影响性能的。各种隔离级别，行锁，表锁，页锁，读写锁，事务锁，以及各种写优先还是读优先机制。性能最高的是不要锁，所以，分库分表，冗余数据，减少一致性事务处理，可以有效地提高性能。NoSQL就是牺牲了一致性和事务处理，并冗余数据，从而达到了分布式和高性能。
* **数据库的存储机制**。不但要搞清楚各种类型字段是怎么存储的，更重要的是数据库的数据存储方式，是怎么分区的，是怎么管理的，比如Oracle的数据文件，表空间，段，等等。了解清楚这个机制可以减轻很多的I/O负载。比如：MySQL下使用show engines;可以看到各种存储引擎的支持。不同的存储引擎有不同的侧重点，针对不同的业务或数据库设计会让你有不同的性能。
* **数据库的分布式策略**。最简单的就是复制或镜像，需要了解分布式的一致性算法，或是主主同步，主从同步。通过了解这种技术的机理可以做到数据库级别的水平扩展。

**B）SQL语句优化**

关于SQL语句的优化，首先也是要使用工具，比如：[MySQL SQL Query Analyzer](http://www.mysql.com/products/enterprise/query.html)，[Oracle SQL Performance Analyzer](http://www.oracle-base.com/articles/11g/sql-performance-analyzer-11gr1.php)，或是微软[SQL Query Analyzer](http://msdn.microsoft.com/en-us/library/aa216945(v=sql.80).aspx)，基本上来说，所有的RMDB都会有这样的工具，来让你查看你的应用中的SQL的性能问题。 还可以使用explain来看看SQL语句最终Execution Plan会是什么样的。

还有一点很重要，数据库的各种操作需要大量的内存，所以服务器的内存要够，优其应对那些多表查询的SQL语句，那是相当的耗内存。

下面我根据我有限的数据库SQL的知识说几个会有性能问题的SQL：

* **全表检索**。比如：select \* from user where lastname = “xxxx”，这样的SQL语句基本上是全表查找，线性复杂度O(n)，记录数越多，性能也越差（如：100条记录的查找要50ms，一百万条记录需要5分钟）。对于这种情况，我们可以有两种方法提高性能：一种方法是分表，把记录数降下来，另一种方法是建索引（为lastname建索引）。索引就像是key-value的数据结构一样，key就是where后面的字段，value就是物理行号，对索引的搜索复杂度是基本上是O(log(n)) ——用B-Tree实现索引（如：100条记录的查找要50ms，一百万条记录需要100ms）。
* **索引**。对于索引字段，最好不要在字段上做计算、类型转换、函数、空值判断、字段连接操作，这些操作都会破坏索引原本的性能。当然，索引一般都出现在Where或是Order by字句中，所以对Where和Order by子句中的子段最好不要进行计算操作，或是加上什么NOT之类的，或是使用什么函数。
* **多表查询**。关系型数据库最多的操作就是多表查询，多表查询主要有三个关键字，EXISTS，IN和JOIN（关于各种join，可以参看[图解SQL的Join](http://coolshell.cn/articles/3463.html)一文）。基本来说，现代的数据引擎对SQL语句优化得都挺好的，JOIN和IN/EXISTS在结果上有些不同，但性能基本上都差不多。有人说，EXISTS的性能要好于IN，IN的性能要好于JOIN，我各人觉得，这个还要看你的数据、schema和SQL语句的复杂度，对于一般的简单的情况来说，都差不多，所以千万不要使用过多的嵌套，千万不要让你的SQL太复杂，宁可使用几个简单的SQL也不要使用一个巨大无比的嵌套N级的SQL。还有人说，如果两个表的数据量差不多，Exists的性能可能会高于In，In可能会高于Join，如果这两个表一大一小，那么子查询中，Exists用大表，In则用小表。这个，我没有验证过，放在这里让大家讨论吧。另，有一篇关于SQL Server的文章大家可以看看《[IN vs JOIN vs EXISTS](http://explainextended.com/2009/06/16/in-vs-join-vs-exists/)》
* **JOIN操作**。有人说，Join表的顺序会影响性能，只要Join的结果集是一样，性能和join的次序无关。因为后台的数据库引擎会帮我们优化的。Join有三种实现算法，嵌套循环，排序归并，和Hash式的Join。（MySQL只支持第一种）
  + 嵌套循环，就好像是我们常见的多重嵌套循环。注意，前面的索引说过，数据库的索引查找算法用的是B-Tree，这是O(log(n))的算法，所以，整个算法复法度应该是O(log(n)) \* O(log(m)) 这样的。
  + Hash式的Join，主要解决嵌套循环的O(log(n))的复杂，使用一个临时的hash表来标记。
  + 排序归并，意思是两个表按照查询字段排好序，然后再合并。当然，索引字段一般是排好序的。

还是那句话，具体要看什么样的数据，什么样的SQL语句，你才知道用哪种方法是最好的。

* **部分结果集。**我们知道MySQL里的Limit关键字，Oracle里的rownum，SQL Server里的Top都是在限制前几条的返回结果。这给了我们数据库引擎很多可以调优的空间。一般来说，返回top n的记录数据需要我们使用order by，注意在这里我们需要为order by的字段建立索引。有了被建索引的order by后，会让我们的select语句的性能不会被记录数的所影响。使用这个技术，一般来说我们前台会以分页方式来显现数据，Mysql用的是OFFSET，SQL Server用的是FETCH NEXT，这种Fetch的方式其实并不好是线性复杂度，所以，如果我们能够知道order by字段的第二页的起始值，我们就可以在where语句里直接使用<=的表达式来select，这种技术叫seek，而不是fetch，seek的性能比fetch要高很多。
* **字符串**。正如我前面所说的，字符串操作对性能上有非常大的恶梦，所以，能用数据的情况就用数字，比如：时间，工号，等。
* **全文检索**。千万不要用Like之类的东西来做全文检索，如果要玩全文检索，可以尝试使用[Sphinx](http://sphinxsearch.com/)。
* **其它**。
* 不要select \*，而是明确指出各个字段，如果有多个表，一定要在字段名前加上表名，不要让引擎去算。
* 不要用Having，因为其要遍历所有的记录。性能差得不能再差。
* 尽可能地使用UNION ALL  取代  UNION。
* 索引过多，insert和delete就会越慢。而update如果update多数索引，也会慢，但是如果只update一个，则只会影响一个索引表。
* 等等。

关于SQL语句的优化，网上有很多文章， 不同的数据库引擎有不同的优化技巧，正如本站以前转发的《[MySQL性能优化的最佳20+条经验](http://coolshell.cn/articles/1846.html)》

先写这么多吧，欢迎大家指正补充。

**注：**这篇文章的确是个大杂烩。其实其中的说到的很多技术在网上都有很多很多的技术文章，google一下就能找到一堆有很多细节的文章，所以我也就不写了。这篇性能调优的文章写作的动机是之前看到 [@淘宝褚霸](http://weibo.com/n/%E6%B7%98%E5%AE%9D%E8%A4%9A%E9%9C%B8) 强推的[highscalability.com](http://highscalability.com/)上的这篇文章：[Big List Of 20 Common Bottlenecks](http://highscalability.com/blog/2012/5/16/big-list-of-20-common-bottlenecks.html)，觉得这篇文章泛泛而谈，觉得自己能写得比它好，所以就产生了动机。