打造支撑海量用户的

高性能 server 系列之二

高性能 server 的通信模型

By　@codebox-腾讯

http://weibo.com/codebox

引

以此系列文章纪念因 QQ 空间、QQ 农场等互联网业务爆炸式增长，而日夜不眠地为公共组件做架构调整、性能优化的激情岁月。

序

以 QQ 农场、好友买卖为代表的 SNS 网页游戏的突然兴起，全国男女老少一起白黑不分的在电脑面前奋战，鼠标、键盘还有各路外挂汇成的请求像洪水一般涌来。当流量把机房核心交换机冲得七零八落，当我们把市场上的服务器全部买光都不够用时，我终于懂了，这就是 TMD 的所谓的海量。

在这个过程中，从 web 层到逻辑层，再到数据层，每一个组件都经过了一次洗礼，从小男孩成长为真正的男人。

现在我已经投身于云平台的建设了，这个支撑海量用户的高性能 server 系列文章就想为这段激情燃烧的岁月做个迟到的总结。目前打算从以下几个方面展开来：

1) 高性能 server 的内存管理。
2) 大并发下的 server 模型选择。
3) 高效定时器的实践。
4) 异步框架的设计与威力。
5) socket 及 OS 优化。
6) 业务应用的架构调整与思考。

这其中大部分是泥腿子办法，土法炼钢思路，不过却的的确确解决了问题，经受住了考验。

在这儿把这些东东放出来，我按我的想法写，您按您的想法看，然后，然后就没有了。。。

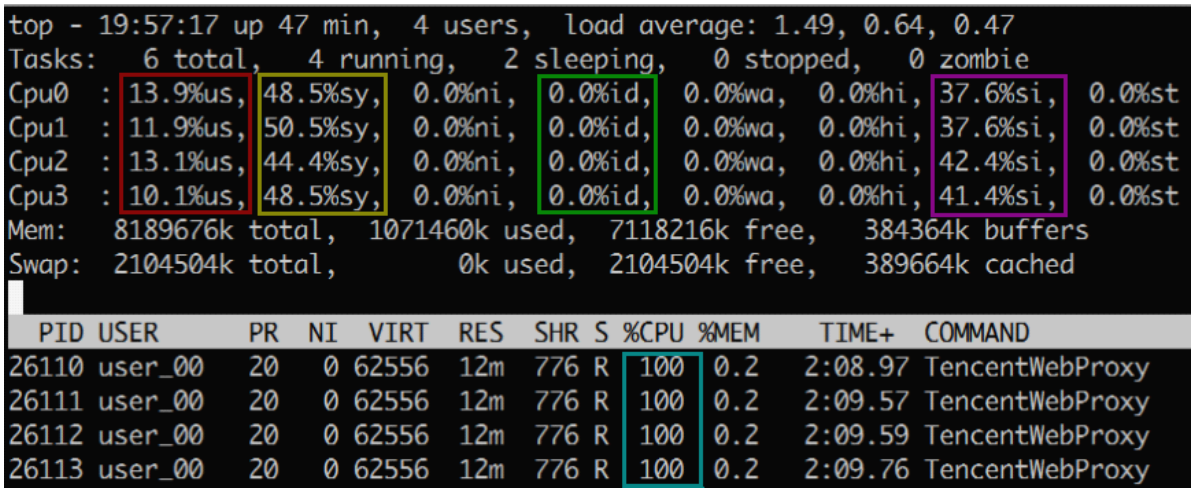
如何衡量好与坏

首先，要做好一件事情，就要知道什么是好与坏。那如何评价一个 server 写得怎么样？我的看法是，**能最有效地榨干系统资源**。

- 两个关键词，**有效**和**榨干**。

有效表明全在干正事儿，不做无用功；榨干表明充分利用，不浪费。

拿我自己实现的一个 web server 来举例(嘿嘿，我比较得意的一个作品 TencentWebProxy)，在压力测试过程中，表现如下：



i. 充分地利用了 CPU，机器共四个核，TencentWebProxy 相应地启动了四个工作线程，每个线程都完全把系统 CPU 吃尽。每个 CPU 使用都非常均匀，不管是系统态，用户态，还是软中断。当有更多的 CPU 核心时，也有通过启动相匹配的线程数，很好的利用多核。**此为榨干**。

ii. 按 CPU 各个维度的比例来说，系统态和软中断占了近 90%，说明系统大部分资源在处理网络服务 ,只有 10%多一点的用在用户态 ,说明对 HTTP 协议解析等操作的资源消耗控制地比较理想，全在干正事儿。**此为有效**。

之所以没有列出 QPS，TPS 之类的测试数据，我是这么看的，一台机器有它的物理性能极限，程序只要能充分把物理资源利用起来，自然会有比较好的结果。绝对性能数值会因机器硬件配置不同而带来完全不同结果 ,写出来意义感觉不是很大。

下面就进入这篇的主题，如果构建高性能 server 的第二篇，Server 模型。

高并发下的 server 模型

研究 server 模型的目的

一是为了适应特定的硬件体系与 OS 特点

比如说相同的 server 模型在 SMP 体系下与 NUMA 下的表现就可能不尽相同，又如在 linux 下表现尚可的进程模型在 windows 下面就非常吃力。

好的 server 模型应该从硬件和 OS 的进步发展中，得到最大化的好处。

二是为了实现维护成本与性能的平衡

好的模型会在编程难度与性能之间做比较好的平衡，并且会很容易地在特定场景下做重心偏移。

本篇文章的讨论的背景限定在基于 SMP 体系下的 Linux 系统上的 Server。

说在前面的话

关于线程和进程

在以下所述的所有模型中，进程或线程的选用，没有太本质的区别。只是一个重一点，另一个轻量一点。本文所论述的场景中，基本上都可以互换。

在具体的使用过程中唯一要注意的是：函数重入性（线程安全），进程间通信与线程间共享的区别。

关于这个话题

这个话题的水非常深，我在这儿也是只是结合特定场景和亲身经历分享一下自己的心得，不可能面面俱到，更不敢保证所讲的正确性。仅仅是老老实实把自己实践的经验说出来，不敢有半点装逼之倾向。。

基本 server 模型演化

我们先从最简单的模型介绍起，然后我们一点点地将其改进。

单进程 helloword 模型

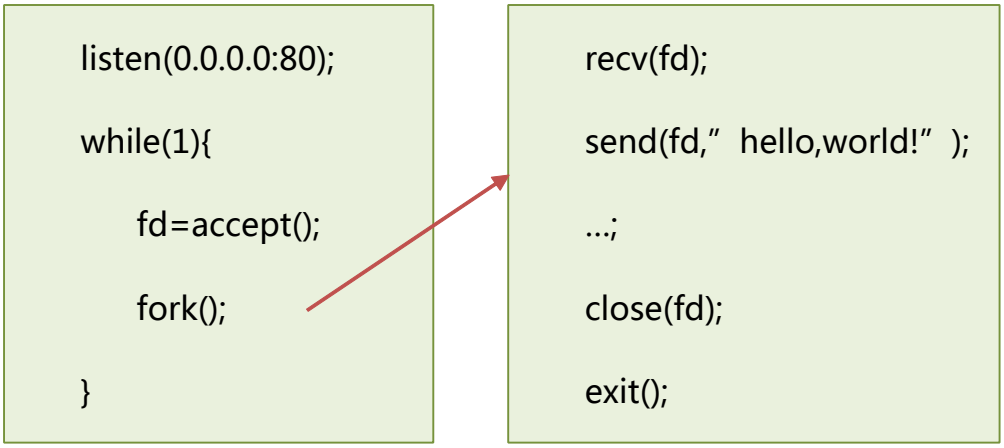
当我们学着写第一个 server 的时候 ,基本上它就算 server 界的 hello world 了。

```
listen(0.0.0.0:80);
while(1){
    fd = accept();
    recv(fd);
    send(fd,” hello world!” );
    close(fd);
}
```

很明显这个最简单的伪代码片段并不支持并发，同时只能处理一个客户端的请求。如何以最简单的方式支持并发(在这篇文章中我们不严格区分并发和并行☺)呢？最简单的方式是为每个任务 fork 出一个子进程来处理事务就可以，即 fork 模型。

Fork 模型

支持并发最简单的方式就是为每个一客户端连接分配一个子进程来处理。由于从某种程度上来讲，进程是可以并发执行的，也就是利用了进程的并发性来实现了 server 的并发模型。



这个模型表面上看十分简单，但功能却十分强大。在性能要求不是很高的场景中运用非常广泛。例如 Linux 中万能的 INETD 基本上就是脱胎于此。

这个模型致命的优势就是简单与安全。简单，因为简单，所以就犯不着说它为什么简单了。安全，是因为进程本身具有非常好的隔离性，其中某个任务出现问题基本上不会影响到另外一个任务的运行。甚至有一定的资源泄漏关系也不太大，因为运行一次就退出，泄漏就泄漏，管它呢！

由于它这两个致命的优点，在性能要求不是非常高的场合，fork 模型都是能胜任的。最最典型的应用当数传统 CGI 了。想想写 CGI 的自由程度，c/c++、php、python、perl、甚至 shell，只要写出来的东东能运行，就能当 CGI。这个就全仰仗 fork 模型的优点了。

Prefork

Fork 模型的优点非常明显，缺点也直接了当：每个新连接都会新建一个进程，任务完成后又接着退出。一个进程创建并不容易，打开 exe 文件加载到内存中，

还要再加载一系列 so，准备各种 task struct 和页表等内核数据结构，辛辛苦苦创建起来的进程往往只说个 hello world 就拜拜了，似乎很不经济。

提到 fork 模型与 prefork 模型的区别，我想到一个段子：

> 用 QQ 6 年多了，今天发现一个惊天大秘密心情异常激动，我估计没有几个人会知道的，所以拿出来和大家分享，也让大家长长见识！！就是以前每次我要用 QQ 聊天时每次都要申请一个号码，今天我发现原来只要申请一个号码下次还可以用，重要的是，你只要记住当时的号码和密码就可以了，这样每次要用 QQ 聊天的时候就不用那么麻烦去申请一个新号码了。真的太神奇了！

这位哥们就是发现了原来 QQ 是可以使用 prefork 模型的，不用每次都 fork 一个新 QQ 号。

Prefork 的伪代码如下：

```
listen(0.0.0.0:80);
for(n){
    fork();
}
```

```
while(1){
    fd=accept()
    recv(fd);
    send(fd,“ hello,world!“ );
    ...;
    close(fd);
}
```

```
while(1){
    ...
    ...
}
```

主进程　　预先 fork 出来的子进程

我们可以从上面的段子和伪代码中得到 prefork 的优势：一提前 fork 好进程可以提高请求处理速度，二进程可以重复利用，大大提高了系统使用效率。（即之前提到的有效原则）

Prefork 带来一个额外的工作是要对空闲进程数进行管理，既要求有一定的空闲进程来服务新进的请求，又不使空闲进程数过多造成浪费。进程数的经验值可以这样估计 `n=current * delay` 。Current 是并发量，delay 是平均任务时延。如当 delay 为 2，并发量 current 为 100 时，进程数至少要为 200。

如果性能要求不是太变态，prefork 模型基本上可以应付一般的高并发应用场景。如果要求真的很变态，那这种模式的致命伤在哪儿呢？ 是“**用进程数来抗并发用户数**”，因一台机器可启动的进程数是有限的，运营中还没见到过进程能开到 K 级别，并且当进程太多时 OS 调度和切换开销往往也大于业务本身了。

Perfork+异步模型

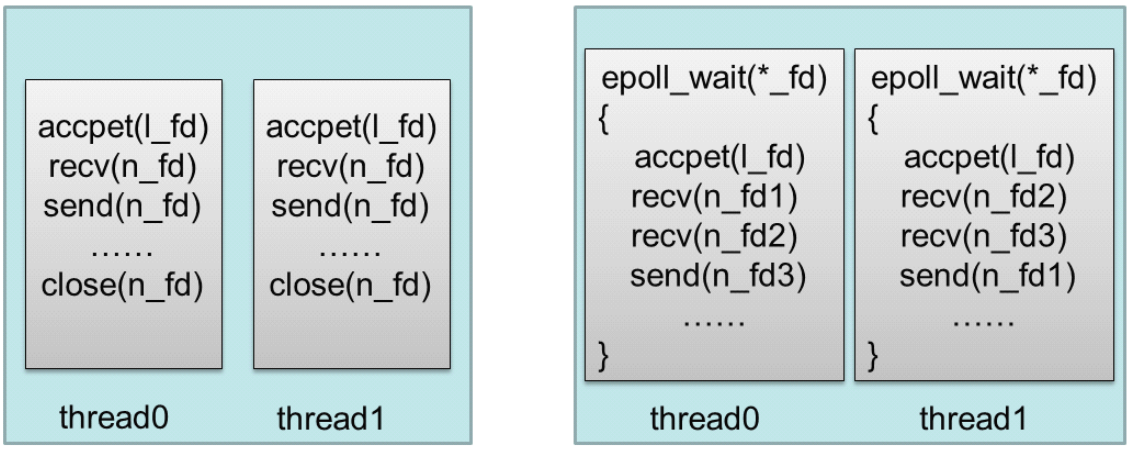
上面提到传统的 perfork 使用进程数来抗并发用户数，这种方式天然受到进程数限制和进程切换带来的巨大开销两方面的限制。

解决这个问题的思路就是使用 IO 多路复用(epoll)来同时 hold 住海量连接，也叫做异步模式。在这种模型下，进程数跟 CPU 核数基本一样多就可以了(，也不用像同步模型一样管理进程数。如何使用一个进程同时 hold 海量请求，我喜欢用一个视频来说明问题（点击链接，可直接叉掉贴片广告)。 用比较学究的话来说，一个进程同时处理多个任务叫做**并发(concurrency)**，就像一个 CPU 上同时承载多个进程一样(两种情况下都是宏观的)。

多路复用 IO 事件库这方面开源的库比较有名的是 libevent 和 libev，还有国人的骄傲@bnu_chenshuo 写的 muduo。我们几乎都是清一色用的廖生苗实现的一套 epoll 封装，非常轻量，使用起来也很方便。编写异步 server 的难度比较

大，所以我们内部对异步和业务流程做了一定的抽象，做了到通用逻辑 server 框架(serverbench++)里面，这个系列文章会专门有一篇来探讨这个问题。

> 注：“异步”在不同的场景下，意义不尽相同。这儿所谓的异步，就是使用 IO 多路复技术，使用有限状态机轮转的事件驱动方式处理业务逻辑。状态机里面的任何操作都必须是非阻塞的，由于 N 多请求在一个线程里面跑，所以只要有一个点有阻塞，整个状态机制轮转会停下来，所有下在处理的任务都会受到影响(结合上面的那个视频，就是其它的锅都会糊掉)。另一个问题是，本文中没有很严格地来区别并发(concurrency)、并行(parallelism)，大家在具体的上下文中会意就可以了，没必要很学究☺



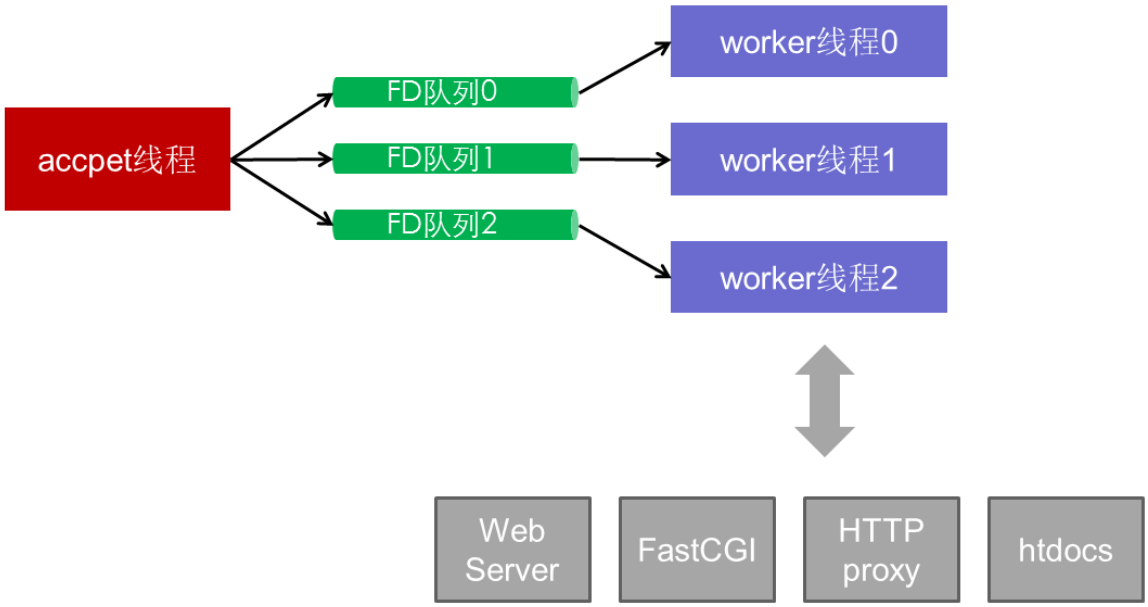
同步线程模型　　异步线程模型

上图是同步线程与异步线程伪代码对比。

左边同步模型里面多个进程同时 accept 一个 listen fd，当有一个链接过来的时候，Linux 只会唤成一个进程来处理。早期的内核这儿会有惊群效应的，当链接过来时，所有的进程都会被唤醒，但只有一个会抢到。现在的 linux 中已经没有这个问题了。

但在右边的异步模型中，当 listen fd 有事件时，所有线程的 epoll_wait 都会检测到事件通知，还是会有惊群的。这是这种方式的一个缺点，但由于进程不是很多(一般为 CPU 数)，所以问题也不是很大。如果要彻底解决这个问题，就是把 accept 操作单独拎出去，成为一个单独的线程，专门管 accept 操作。Accept 线程建立好连接后，将 fd 传递给 worker 线程处理。这样 worker 线程中 epoll 处理的就只有 net fd，而没有 listen fd 了。

之前提到的 TencentWebProxy 就是采用的这种架构。上面的模型具体到 TWP 身上的架构图如下：



注：所有的线程都是由事件驱动的，包括所有的网络事件，超时事件，FD 队列的事件等等，任何操作必须都是非阻塞的。FD 队列的实现是通知+轮询的方式，跟网卡驱动的 NAPI 的思想是一致的。

新变化

正如我们在开头研究 server 模型的目的中说的，其一为“**适应特定的硬件体系与 OS 特点**”，那么 OS 的新特性肯定影响到 server 模型。比如 Google 的 Tom

Herbert 这哥们在一个月(2013-01-14)前提了一个新补丁(http://www.spinics.net/lists/netdev/msg222329.html)，多个 socket 可以绑定到同一个端口。这个特性极大的改变我们的网络接入模型。用他自己的话来说“The motivating case for so_resuseport in TCP would be something like a web server binding to port 80 running with multiple threads, where each thread might have it's own listener socket.”。简直就是为我们这个场景准备的。有了这个特性之后，我们的 server 就不用带 accept 线程的改进模型了，使用最开始的模型就 OK 了。

这个例子也算是对研究 server 模型的理由中”**适应特定硬件体系和 OS 特点**”的印证。

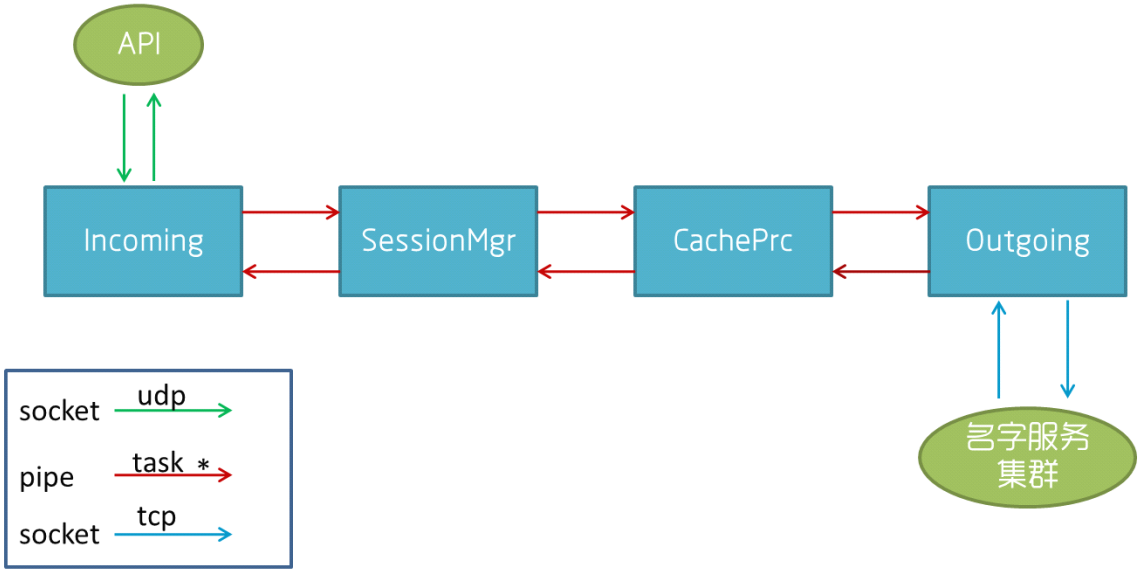
异步流水线

上面提到的异步模型处理的业务相对简单，包括 web proxy，所有的业务逻辑抽象成有限状态机也不是很复杂。但如果 server 的功能特性比较多的话，全部抽象到一个状态机，不管是实现还是维护、debug 都比较困难。针对这种情况，于是就引出了这儿要介绍的异步流水线架构。

如何将一个 server 中的流程和功能解耦，unix 哲学给我们提供了一条异常好用的解决方案：每个模块只干一件事情，把每个模块用管道连接起来！

于是对于复杂一些的 server 就把每个模块抽象成线程对象的子类，不同的模块也就是不同角色的线程。Perfork+异步模型中提到，简单地把 server 分解成 accept 线程和 worker 线程。

举一个现实中的例子，这是我们一个 cache server 的设计。它的持久化数据是存在右下方的名字服务中的，API 与 server 之间的通信协议是自已实现的一套可靠 udp 协议。另外由于产品特性上要支持反向通知，把以要维护会话与心跳。由此看来，这个 server 的复杂性还是比较高的。基于异步流水线的思路，设计了以下架构。



每个蓝色的模块都是一个角色线程，具体职责如下：

Incoming

负责网络层，实现可靠 udp 协议，并将用户请求转化为一个 task 结构体。在各模块之间传递的都是这个 task 指针。由于这个 server 是 udp 协议的，所以不存在 accept 线程。

SessionMrg

负责维护会话信息，如会话的建立、终结、心跳、反向通知等等。

CachePrc

负责管理缓存的数据

Outgoing

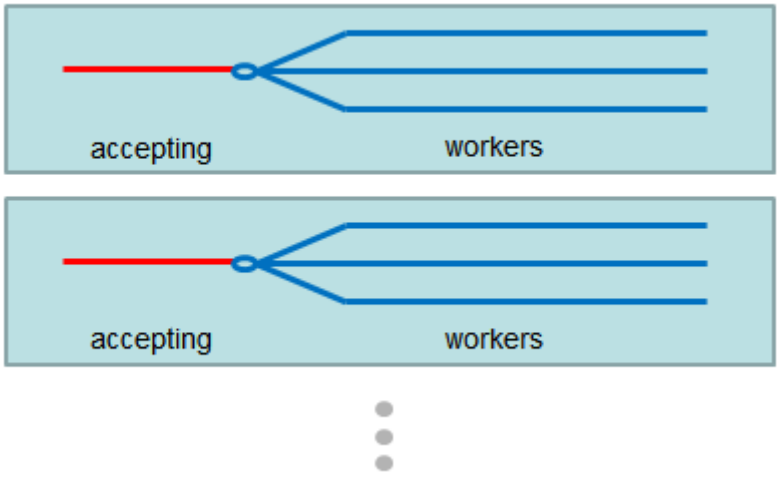
负责跟数据源打交道，当 cache 不命中时，outgoing 会向持久数据发请求。每个模块负责非常具体的任务，模块之间用管道连接起来，管道中传递的是 task 指针，将管道接收端的 fd 放到 epoll 池子里，这样能传递数据也能通知事件了。估计这个地方用 event fd 也比较合适，不过一来将就老内核，二来好处不是特别明显，所以一直以来没有改这块。

分析几个案例

APACHE

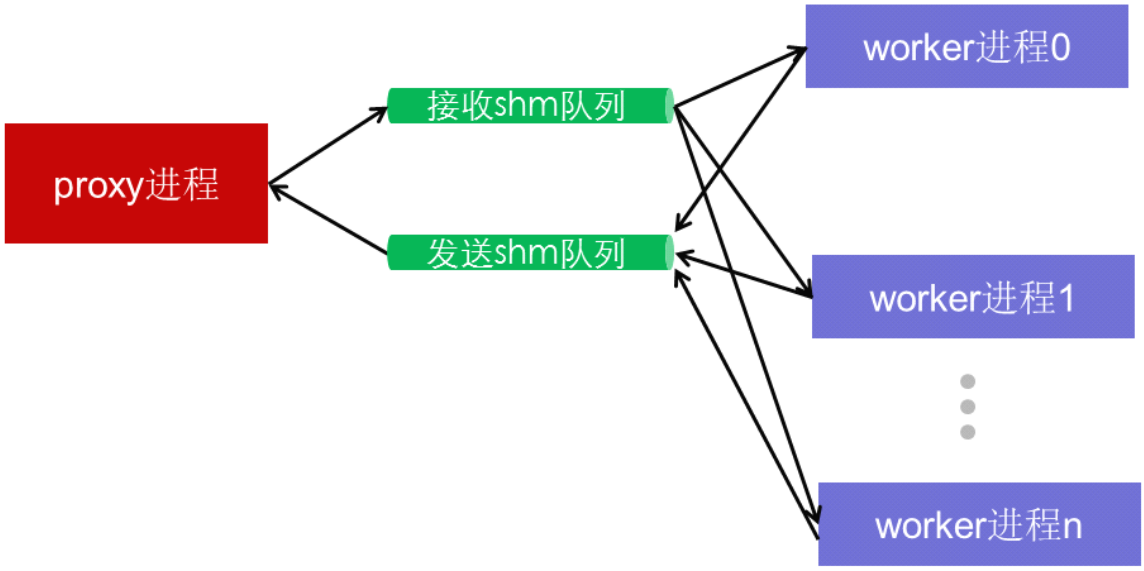
Apache MPM 在 linux 架构下默认的是 prfork。但做了一些比较有意思的变化，它混合了进程模型和线程模型，以期在安全性和性能之间做一个平衡。

Apaceh 每个线程里面都有两种线程，有一个异步的 accpet 线程和一组 worker 线程。accept 线程的作用跟我们前面所述的一致，但 worker 是同步的，每个 worker 同时只处理一个任务。整体结构如下：



ServerBench++(同步)

ServerBench++是腾讯互联网部门应用最广泛的通用逻辑 server，采用的是比较经典的半同步/半异步架构。



所谓半异步是指 proxy 这一部分是异步的，用一个进程挂住所有的连接，并做包的收发与解析工作，当包完整后投入到共享内存中去，由 worker 取回处理。

所谓的并同步是指 worker 这部分是同步的。Worker 处理所有的业务逻辑，不关心网络交互，同步模式是开发最简单的。之所以采用进程模式也是因为 S++ 框架运行是加载业务的插件运行的，代码质量不一而足，当有问题时不致使整个框架崩溃掉。

异步的 ServerBench++中，不仅 proxy 是全异步的，worker 也是全异步的，这个话题比较大，这个系列文章中有会有一篇专门来介绍。

几个性能注意点

TCP_DEFER_ACCEPT

在 listen fd 上设置 TCP_DEFER_ACCEPT 后，accpet(2)和 listen fd 的行为会发行变化。三次握手之后，应用层是感知不到，只有当有真正的用户数据到达的时候，应用层才会感知到(阻塞的 accept 返回或 listen fd 有事件通知)。这时 accept 出来的 fd 直接进行读操作就可以了，不用再等 INPUT 事件通知了。这样可以节省一点点资源。

accept4、socket 与 SOCK_NONBLOCK

在将 fd 放入 epoll 池子进行管理的时候，一定要先将 fd 用 fcntl(2)设置成非阻塞。从 Linux 2.6.28 开始的 accept4 支持 SOCK_NOBLOCK 参数，accept 出来的 fd 就已经是非阻塞的了。这样就可以减少一次 fcntl(2)的调用开销了。

同样 socket(2)调用从 2.6.27 开始支持 SOCK_NONBLOCK 参数了。

特别是从老的系统中迁移过来的 server，可以注意一下这两个地方，看看能不能优化。

非事件触发 send

在 epoll 状态机中，send 操作最好是需要的时候直接调用，而不是先打开 OUTPUT 事件等回调。因为一般情况下，fd 都是可写的，只有当发送缓存区满的情况下才打开 OUTPUT 事件等合适的机会再发送出去。这一点一般大伙儿也都是这么实现的。

上面实现虽然效率会高些，但是代码结构会不清晰，在不同的地方都会出现 send 操作。以下是我们的解决思路：

在 Server 的定时器单元中，实现一个特殊的定时器—ReadyTimer。挂到这个 timer 上面的事件，会在检查超进操作时立刻被回调。也就是说超时时间是 0。实际上就是一个推后的回调机制，借用这个机制可以在 send 的地方把事件挂到 ReadyTimer 上，由 ReadyTimer 负责回调 send 操作(跟 OUTPUT 事件的回调函数是一个)。这样既没有使用多余的 OUTPUT 事件搞高了效率，代码又没有冗余。

这个跟内核中硬中断通过 raise_sofirq()置位软件中断，do_softirq()来轮询并回调 handler 的思路是类似的。

gettimeofday

在 server 有很多地方需要时间，而在在比较老的系统中(不支持 vdso)，gettimeofday 是非常消耗资源的，所以在使用 gettimeofday 的时候必须克勤克俭。虽然现在新的内核中，问题已经不是那么大了，但我们都从苦日子中过来的，节约一点是一点。

节省的方式实际上在各个 server 中的方式大体上都是一致的，即牺牲一定的精度换取性能。一般都是在 epoll_wait(2)后取一次，然后在整个大循环中都用这一值作为当前时间。由于整过程都是非阻塞的，基本上可以保证精度在毫秒级别。对于一般的应用场景足够了。我隐约记得 nginx 中好像也是这么处理的。

针对 gettimeofday 的问题，之前还在微博上提到另外一种不修改代码的方案，

也附在这儿：

timerkeeper系统简介

- 我们在业务中有时候会频的调用gettimeofday,time时间有关的函数（比如模块点上报等统计类操作），而这两个系统调用的消耗十分可观，特别是在不支持vdso操作系统上。
- timerkeeper就是为解决此种问题而产生的。

timerkeeper实现原理

timerkeeper分为两个模块，Timerkeeper_daemon负责取基准时间，Timerkeeper_lib跟据基本时间和tsc得出校准时间供用户使用。

- Timerkeeper_daemon
  - 负责每隔固定的周期取出时间，机器频率，tsc的值放在map文件中，供lib库使用。
  - Watcher dog进程负责监控 timerkeeper_daemon，如果出现异常，则重启。
  - 只有root才能启动进程，保证进程和map文件的安全性。
  - 提高自己的nice为-5，使得取时间相对更加实时。

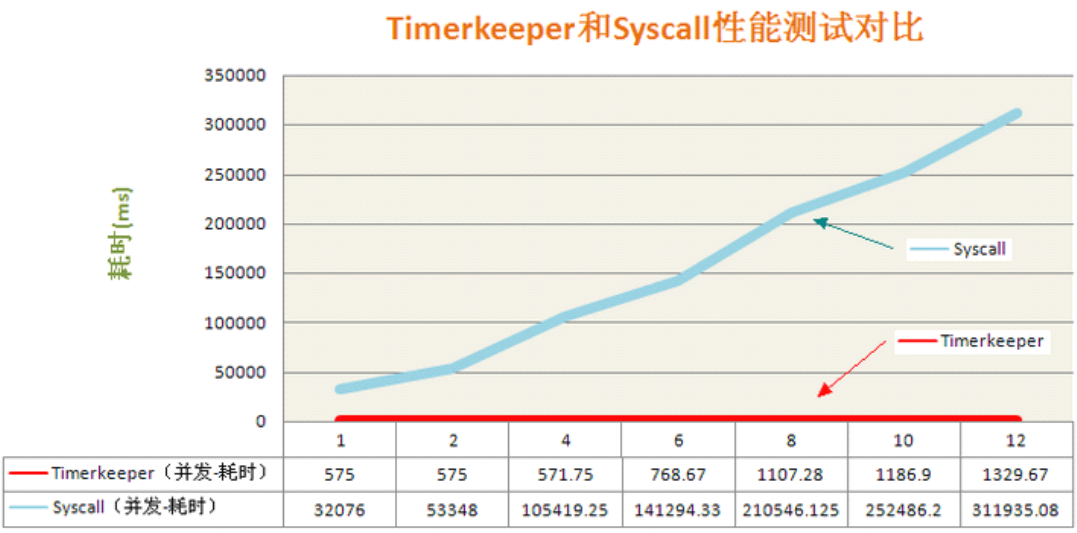
- Timerkeeper_lib
  - 负责从map文件中取出时间，机器频率，tsc，通过tsc校正时间使之精确到1ms之内。
  - 当出现tsc偏差过大，或产生任何错误时，则调用真正的系统调用。

timerkeeper使用方法

- 启用timerkeeper_daemon
  使用-h参数可能查看版本号及使用说明。

- 用户可以采用两种方式使用timerkeeper_lib
  - 使用preload方式(不用改代码，不用重新编译)
    在启动脚本加入 export LD_PRELOAD=/path/to/libtimerkeeper.So
    比如
    export LD_PRELOAD=/path/to/libtimerkeeper.So
    ./myprogram
  - 使用链接的方式（不用改代码）
    gcc [...] -o myprogram -ltimerkeeper
    timerkeeper库要放在最左边（即其它库的前面）

timerkeeper性能报告



	1	2	4	6	8	10	12
Timerkeeper（并发-耗时）	575	575	571.75	768.67	1107.28	1186.9	1329.67
Syscall（并发-耗时）	32076	53348	105419.25	141294.33	210546.125	252486.2	311935.08

测试平台　Intel(R) Xeon(R) CPU X3210 @ 2.13GHz 172.25.0.29
测试方法　每个测试程序做100000000(1亿)次，然后分并发1,2,4,6,8,10,12个进程，测试其耗时(ms)。
测试程序　都用同一个测试程序，timerkeeper使用perload方式。

有用的资料

Unix 网络编程 第 1 卷 第 30 章 客户端/服务器程序设计范式

这一章对 server 模型做了非常经典的论述，但因针对的是传统 Unix，所以直接套用到 Linux 中可能多少会有点问题。但作者经典分析问题的思路是绝不过时的。

面向模式的软件体系结构 卷 2 用于并发和网络对象的模式

这本书从更高的层次是分析了并发模型，多了些思想，少了些细节，是一本相当不错的参考。

The C10K problem

这篇文章在高性能 server 领域是家喻户晓的，对不同操作系统的各种通信机制有非常详尽的介绍。跟“面向模式的软件体系结构 卷 2”互为补充，是很好的参考。

小结

性能优化与业务抽象很大程度上是矛盾的两件事。性能优化代码要往底层靠，而业务抽象要往上层走。所以性能优化一定要适度，做到合适我们的应用场景就好了。为了性能，在代码中各种 hack，导致代码可读性、可维护性降低，是得不偿失的。

在性能优化和代码清晰之间做选择的时候，绝大多数时候要选择后者。我曾经无数次想在代码里装装逼，几乎无一例外地被雷劈了！

跋

Server 模型这个话题的水非常深，写这篇文章的时候也是诚惶诚恐。仅仅是把自己实践和运营过程中的体会真实的记录下来，供网友参考。既不可能面面俱到，更遑论保证所讲的正确性，写起来也是如履薄冰。大伙以批判求真的心态来阅读最合适了。

我在 **http://weibo.com/codebox** 期待大家的交流。