The C10K problem

http://www.kegel.com/c10k.html

翻译：oschina

2013/10/07

The C10K problem

是时候让 Web 服务器同时处理一万客户端了，你不觉得吗？毕竟，现在的 Web 是一个大地盘了。

并且，计算机也是一样的大。 你可以花 $1200 左右购买一台 1000MHz，2Gb RAM 和一块 1000Mbit/s 以太网卡的机器。我们来看看——在 20000 客户端（是 50KHz，100Kb 和 50Kb/s/客户端）时，它不采取任何更多的马力而是采用 4Kb 的硬盘和为 2 万客户端中的每个一秒一次的发送它们到网络。（顺便说一句，这是$0.08 每客户端。 那些按 $100/客户端 许可费来收取费用的一些操作系统开始看起来有点沉重！）。所以硬件不再是瓶颈了。

1999 年最繁忙的 FTP 网站： cdrom.com，实际上通过一个千兆的以太网管道同时地处理了 10000 客户端。截至 2001 年，同样的速度现在由多个 ISP 提供，期望它变得越来越受大型商业客户的欢迎。

瘦客户端计算模式显现出回来的风格——这一时刻，在互联网上的服务器正在为数千计的客户端服务。

考虑到这一点，这里有几个关于如何配置操作系统和编写代码以支持数千客户端的注意事项。讨论的中心是围绕类 Unix 操作系统的。因为这是我个人感兴趣的领域。但 Windows 也包括了一点。

目录

- 例程
  - 有趣的服务器 select()
  - 有趣的服务器 /dev/poll
  - 有趣的服务器 kqueue()
  - 有趣的服务器的实时信号
  - 有趣的服务器线程
  - 有趣的服务器内核
- 其他有趣的链接

相关站点

阅读 Nick Black 写的超级棒的 Fast UNIX Servers 文章.

2003 年十月， Felix von Leitner 做了一个超级好的网页和展示，了网络的可扩展性，他完成了在各种不同的网络系统请求和操作系统下的 benchmark 比较。他的一个实验观察结果就是 linux2.6 的内核确实比 2.4 的要好，但还有很多很多好的图表数据可以引起 OS 开发者的深思（如有兴趣可以看看 Slashdot 的评论;是否真的有遵循 Felix 的实验结果对 benchmark 的提高进行跟踪的）。

提前阅读

如果你还没有读过上述，那么先出去买一本 W. Richard Stevens 写的 Unix Network Programming : Networking Apis: Sockets and Xti (Volume 1) ．这本书描述了很多编写高性能服务器的 I/O 策略和误区，甚至还讲解了关于 'thundering herd' 问题，惊群问题。

如果你读过了，那么请读这本 Jeff Darcy's notes on high-performance server design。(Cal Henderson 写的，对更加倾向于使用一款 web 服务器而非开发一款服务器 来构建可扩展 web 站点的同志，这本书更加有用)。

I/O 框架

以下所列的为几个包装好的库，它们抽象出了一些下面所表达的技术，并且可以使你的代码与具体操作系统隔离，从而具有更好的移植性。

- ACE, 一个重量级的 C++ I/O 框架，用面向对象实现了一些 I/O 策略和其它有用的东西，特别的，它的 Reactor 框架是用 OO 方式处理非阻塞 I/O，而 Proactor 框架是用 OO 方式处理异步 I/O 的。
- ASIO 一个 C++的 I/O 框架，正在成为 Boost 库的一部分。它像是 ACE 过渡到 STL 时代。（译注：ACE 内部有自己的容器实现，它和 C++ 标准库中的容器是不兼容的。）
- libevent 由 Niels Provos 用 C 语言编写的一个轻量级的 I/O 框架。它支持 kqueue 和 select，并且很快就可以支持 poll 和 epoll(翻译此文时已经支持)。我想它应该是只采用了水平触发机制，该机制功过参半。Niels 给出了 一张图 来说明时间和连接数目在处理一个事件上的功能，从图上可以看出 kqueue 和 sys_epoll 明显胜出。
- 我本人也尝试过写一个轻量级的框架(很可惜没有维持至今):
  - Poller 是一个轻量级的 C++ I/O 框架，它使用任何一种准备就绪 API(poll, select, /dev/poll, kqueue, sigio)实现水平触发准备就绪 API。以其它 不同的 API 为基准，Poller 的性能好得多。该链接文档的下面一部分说明了如何使用这些准备就绪 API。
  - rn 是一个轻量级的 C I/O 框架，也是我继 Poller 后的第二个框架。该框架可以很容易的被用于商业应用中，也容易的适用于非 C++应用中。它如今已经在几个商业产品中使用。

- 2000 年 4 月，Matt Welsh 就构建服务器如何平衡工作线程和事件驱动技术的使用方面写了一篇 论文，在该论文中描述了他自己的 Sandstorm I/O 框架。
- Cory Nelson's Scale! library - 一个 Windows 下的异步套接字，文件和管道的 I/O 库。

I/O 策略

网络软件设计者往往有很多种选择，以下列出一些:

- 是否处理多个 I/O？如何处理在单一线程中的多个 I/O 调用？
- 不处理，从头到尾使用阻塞和同步 I/O 调用，可以使用多线程或多进程来达到并发效果。
- 使用非阻塞调用(如在一个设置 O_NONBLOCK 选项的 socket 上使用 write）读取 I/O，当 I/O 完成时发出通知（如 poll，/dev/poll）从而开始下一个 I/O。这种主要使用在网络 I/O 上，而不是磁盘的 I/O 上.
- 使用异步调用（如 aio_write()）读取 I/O，当 I/O 完成时会发出通知（如信号或者完成端口），可以同时使用在网络 I/O 和磁盘 I/O 上.
- 如何控制对每个客户的服务?
- 对每个客户使用一个进程（经典的 Unix 方法，自从 1980 年一直使用）
- 一个系统级的线程处理多个客户，每个客户是如下一种:
- 一种用户级的线程(如 GNU 状态线程，经典的 java 绿色线程)
- 一个状态机 (有点深奥，但一些场景下很流行; 我喜欢)
- 一个延续性线程 (有点深奥，但一些场景下很流行)
- 一个系统级的线程对应一个来自客户端的连接 (如经典的 java 中的带 native 线程)
- 一个系统级的线程对应每一个活动的客户端连接(如 Tomcat 坐镇而 apache 做前端的；NT 完成端口; 线程池)
- 是否使用标准的操作系统服务，还是把一些代码放入内核中（如自定义驱动，内核模块，VxD）

下面的五种组合应该是最常用的了:

- 一个线程服务多个客户端，使用非阻塞 I/O 和**水平触发**的就绪通知
- 一个线程服务多个客户端，使用非阻塞 I/O 和就绪**改变**时通知
- 一个服务线程服务多个客户端，使用异步 I/O
- 一个服务线程服务一个客户端，使用阻塞 I/O
- 把服务器代码编译进内核

1. 一个线程服务多个客户端，并使用非阻塞 I/O 和电平触发的就绪通知

将所有网络处理单元设置为非阻塞状态，并使用 select() 或 poll()识别哪个网络处理单元有等待数据。这是传统所推崇的。在这种场景，内核会告诉你一个

文件描述符是否已经具备，自从上次内核告诉你这个文件描述符以后，你是否对它完成了某种事件。（名词“电平触发”（level triggered）来自于计算机硬件设计领域；它是'边缘触发'（edge triggered）的对立面。Jonathon Lemon 在他的 BSDCON 2000 关于 kqueue()的论文 中引入了这些术语。）

注意：特别重要的是，要记住来自内核的就绪通知只是一个提示；当你准备从文件描述符读的时候，它可能还未准备就绪。这就是为什么当使用就绪通知的时候要使用非阻塞状态如此重要了。

这个方法中的一个重要瓶颈，就是 read()或 sendfile()对磁盘块操作时，如果当前内存中并不存在该页；将磁 盘文件描述符设置为非阻塞将没有效果。同样的问题也发生在内存映射磁盘文件中。当一个服务端第一次需要磁盘 I/O 时，它的进程模块，所有的客户端都必须等待，因此最初的非线程的性能就被消耗了。

这也是异步 I/O 的目的所在，但是仅仅是在没有 AIO 的系统，处理磁盘 I/O 的工作线程或工作进程也可能遭遇此 瓶颈。一条途径就是使用内存映射文件，如果 mincore()指明 I/O 为必需的话，那么就会要求一个工作线 程来完成此 I/O，然后继续处理网络事件。Jef Poskanzer 提到 Pai，Druschel 和 Zwaenepoel 的 1999 Flash web 服务器就使用了这个方法；他们还就此在 Usenix '99 上做了一个演讲。看上去就好像 BSD 衍生出来的 Unixes 如 FreeBSD 和 Solaris 中提供了 mincore()一样，但是它并不是 单一 Unix 规范的一部分，在 Linux 的 2.3.51 的内核中提供了该方法， 感谢 Chuck Lever。

但在 2003 年 11 月的 freebsd-hackers 列表中，Vivek Pei 报告了一个不错的成果，他们对它们的 Flash Web 服务器在系统范围做性能分析，然后再攻击其瓶颈。他们找到的一个瓶颈就是 mincore（猜测根本就不是什么好方法），另外一个事实就是 sendfile 在磁盘块访问时阻塞了；他们引入了一个修改后的 sendfile()，当需要读 取的磁盘页不在内核中时，返回类似 EWOULDBLOCK 的值，这样便提高了性能。（不确定你怎样告诉用户页面现在位于何处…在我看来这里真正需要的是 aio_sendfile()。） 他们优化的最终结果是 SpecWeb99，在 1GHZ/1GB FreeBSD 沙箱上跑了约 800 分，这要比在 spec.org 存档的任何记录都要好。

对于单线程来说有很多方法来分辨一组非阻塞 socket 中哪一个已经准备好 I/O 了：

传统的 select()

很不幸 select()受 FD_SETSIZE 限制。这个限制已经被编译到了标准库和用户程序。（有些版本的 C 语言库允许在用户应用编译的时候提高这个值。）

参照 Poller_select (cc, h) 做为一个如何使用 select()替代其它就绪通知场景例子。

传统的 poll()

poll()可以处理的文件描述符数量没有硬编码的限制，但是这种方式会慢几千倍，因为大部分文件描述符总是空闲的，扫描上千个文件描述符是耗时的。
有些操作系统（比如 Solaris8）使用类似轮询暗示（poll hinting）的办法来加速

poll(),在 1999 年 Niels Provos 实现了这种办法并且做了基准测试。

参照 Poller_poll (cc, h, benchmarks) 做为一个如何使用 poll()替代其它就绪通知场景的例子。

/dev/poll

在 Solaris 系统中，这是推荐的 poll 的替代品。
隐藏在/dev/poll 后面的想法是利用 poll()经常以相同的参数调用很多次的事实。使用/dev/poll，你获取一个对/dev/poll 的打开的句柄，通过写这个句柄就可以一次性告诉操作系统你对哪些文件感兴趣；从这以后，你只需要读从那个句柄返回的当前准备好的文件描述符。

这一特性悄悄出现在了 Solaris7 (see patchid 106541) 但是首先公开出现是在 Solaris 8; 参照 Sun 的数据，在 750 个客户端的时候，这种实现仅占 poll()开销的 10%。

在 Linux 上/dev/poll 有很多种实现，但是没有一种性能与 epoll 一样好，这些实现从来没有真正完整过。在 linux 上/dev/poll 是不建议的。

参照 Poller_devpoll (cc, h benchmarks) 做为一个如何使用/dev/poll 替代其它就绪通知场景的例子。(注意-这个例子是针对 linux /dev/poll 的，在 Solaris 上可能是无法工作的。)

kqueue()

在 FreeBSD（以及 NetBSD）上，这是推荐的 poll 的替代品。
看下面，kqueue()可以被指定为边缘触发或者水平触发。

2. 一个线程服务多个客户端，并使用非阻塞 I/O 和变更就绪通知

变更就绪通知（或边沿触发就绪通知）意味着你给内核一个文件描述符，这之后，当描述符从 未就绪 变换到 就绪 时，内核就以某种方式通知你。然后假设你知道文件描述符是就绪的，直到你做一些事情使得文件描述符不再是就绪的时才不会继续发送更多该类型的文件描述符的就绪通知（例如：直到在一次发送，接收或接受调用，或一次发送或接收传输小于请求的字节数的时候你收到 EWOULDBLOCK 错误）。

当你使用变更就绪通知时，你必须准备好伪事件，因为一个共同的信号就绪实现时，任何接收数据包，无论是文件描述符都已经是就绪的。

这是“水平触发”就绪通知的对立面。这是有点不宽容的编程错误，因为如果你错过了唯一的事件，连接事件就将永远的卡住。不过，我发现边沿触发就绪通知使使用 OpenSSL 编程非阻塞客户端更容易，所以是值得尝试的。

[Banga, Mogul, Drusha '99] 在 1999 年描述了这种方案。

有几种 APIs 可以使得应用程序获得“文件描述符已就绪”的通知:

kqueue()

这是在 FreeBSD 系统上推荐使用边缘触发的方法 (and, soon, NetBSD)。

FreeBSD 4.3 及以后版本，NetBSD（2002.10）都支持 kqueue()/kevent()，支持边沿触发和水平触发（请查看 Jonathan Lemon 的网页和他的 BSDCon 2000 关于 kqueue 的论文）。

就像/dev/poll 一样，你分配一个监听对象，不过不是打开文件/dev/poll，而是调用 kqueue ()来获得。需要改变你所监听的事件或者获得当前事件的列表，可以在 kqueue()返回的描述符上 调用 kevent()来达到目的。它不仅可以监听套接字，还可以监听普通的文件的就绪，信号和 I/O 完 成的事件也可以.

Note: 在 2000.10，FreeBSD 的线程库和 kqueue()并不能一起工作得很好，当 kqueue()阻塞时， 那么整个进程都将会阻塞，而不仅仅是调用 kqueue()的线程。

See Poller_kqueue (cc, h, benchmarks) for an example of how to use kqueue() interchangeably with many other readiness notification schemes.

使用 kqueue()的例程和库：

- PyKQueue -- 一个 Python 的 kqueue()库.
- Ronald F.Guilmette 的 echo 的服务器例程; 另外可以查看他在 2000.9.28 在 freebsd 上发表的帖子。

Epoll

这是 Linux 2.6 的内核中推荐使用的边沿触发 poll。

2001.7.11， Davide Libenzi 提议了一个实时信号的可选方法，他称之为/dev/epoll， 该方法类似与实时信号就绪通知机制，但是结合了其它更多的事件，从而在大多数的事件获取上拥有更高的效率。

epoll 在将它的接口从一个/dev 下的指定文件改变为系统调用 sys_epoll 后就合并到 2.5 版本的 Linux 内核开发树中，另外也提供了一个为 2.4 老版本的内核可以使用 epoll 的补丁。

unifying epoll, aio, 2002 年万圣节前夕的 Linux 内核邮件列表就统一 epoll，aio 和其它的 event sources 展开了很久的争论，it may yet happen，but Davide is concentrating on firming up epoll in general first.

Polyakov's kevent (Linux 2.6+) 的最后新闻：2006.2.9 和 2006.7.9，Evgeniy Polyakov 发表了融合 epoll 和 aio 的补丁，他的目标是支持网络 AIO。See：

- the LWN article about kevent
- his July announcement

- his kevent page
- his naio page
- some recent discussion

Drepper 的最新网络接口 (proposal for Linux 2.6+)

在 2006 OLS 上，Ulrich Drepper 提议了一种最新的高速异步网络 API. See:

- his paper, "The Need for Asynchronous, Zero-Copy Network I/O"
- his slides
- LWN article from July 22

Realtime Signals 实时信号

Linux2.4 内核中推荐使用的边沿触发 poll。

2.4 的 linux 内核可以通过实时信号来分派套接字事件，示例如下：

```
/* Mask off SIGIO and the signal you want to use. */
sigemptyset(&sigset);
sigaddset(&sigset, signum);
sigaddset(&sigset, SIGIO);
sigprocmask(SIG_BLOCK, &m_sigset, NULL);
/* For each file descriptor, invoke F_SETOWN, F_SETSIG, and set
O_ASYNC. */
fcntl(fd, F_SETOWN, (int) getpid());
fcntl(fd, F_SETSIG, signum);
flags = fcntl(fd, F_GETFL);
flags |= O_NONBLOCK|O_ASYNC;
fcntl(fd, F_SETFL, flags);
```

当正常的 I/O 函数如 read()或 write()完成时，发送信号。要使用该段的话，在外层循环中编写 一个普通的 poll()，在循环里面，当 poll()处理完所有的描述符后，进入 sigwaitinfo()循环。 如果 sigwaitinfo()或 sigtimedwait()返回了实时信号，那么 siginfo.si_fd 和 siginfo_si_band 给出的信息和调用 poll()后 pollfd.fd 和 pollfd.revents 的几乎一样。如果你处 理该 I/O，那么就继续调用 sigwaitinfo()。

如果 sigwaitinfo()返回了传统的 SIGIO，那么信号队列溢出了，你必须通过临时 改变信号处理 程序为 SIG_DFL 来刷新信号队列，然后返回到外层的 poll()循环。

See Poller_sigio (cc, h) for an example of how to use rtsignals interchangeably with many other readiness notification schemes.

See Zach Brown's phhttpd 示例代码来如何直接使用这些特点. (Or don't; phhttpd is a bit hard to figure out...)

[Provos, Lever, and Tweedie 2000] 描述了最新的 phhttp 的基准测试，使用了不同的 sigtimewait()和 sigtimedwait4()，这些调用可以使你只用一次调用便获得多个信号。 有趣的是，sigtimedwait4()的主要好处是它允许应用程序测量系统负载(so it could behave appropriately)（poll()也提供了同样的系统负载 测量）。

Signal-per-fd

Signal-per-fd 是由 Chandra 和 Mosberger 提出的对实时信号的一种改进，它可以减少甚至削除实 时信号的溢出通过 oalescing redundant events。然而是它的性能并没有 epoll 好. 论文(www.hpl.hp.com/techreports/2000/HPL-2000-174.html) 比较了它和 select()，/dev/poll 的性能。

Vitaly Luban 在 2001.5.18 公布了一个实现 Signal-per-fd 的补丁; 授权见 www.luban.org/GPL/gpl.html.（到 2001.9，在很重的负载情况下仍然存在稳定性问题，利用 dkftpbench 测试在 4500 个用户时将引发问题。）

如何使用 signal-per-fd 互换及通知计划请参考参考 Poller_sigfd (cc, h)（转载于：http://www.cnblogs.com/fll/archive/2008/05/17/1201540.html）

3. 每个服务器线程处理多个客户端请求，并使用异步 I/O

可能因为少有操作系统支持异步 I/O，在 Unix 中这种方式还不是很流行，也可能因为这(非阻塞 I/O)使得你不得不重新考虑应用。在标准 unix 下，异步 I/O 功能是由 aio 接口(下滚查看 "异步输入输出"小节)提供, 它将一个信号和一个值与每个 I/O 操作关联。Signals 信号和他们的值通过队列形式 快速传递到用户进程中。这是 POSIX 1003.1b 实时扩展中的，同时也是 单 Unix 规范 的第 2 个版本。

AIO 通常和边界触发完成消息同时使用，例如当操作完成时，一个信号进入队列（也可以在层级触发完成通知时，通过调用 aio_suspend(), 但我觉得应该很少人这样做。）

glibc 2.1 和后续版本提供了一个普通的实现，仅仅是为了兼容标准，而不是为了获得性能上的提高。

Ben LaHaise 编写的 Linux AIO 实现合并到了 2.5.32 的内核中，它并没有采用内核线程，而是使 用了一个高效的 underlying api，但是目前它还不支持套接字（2.4 内核也有了 AIO 的补丁，不过 2.5/2.6 的实现有一定程序上的不同）。更多信息如下：

- "Kernel Asynchronous I/O (AIO) Support for Linux" 2.6 内核的 AIO 实现试图绑在一起的所有资料（发布于 2003 年 9 月 16 日）
- Round 3: aio vs /dev/epoll by Benjamin C.R. LaHaise (发布于 2002 的 OLS)
- Asynchronous I/O Suport in Linux 2.5, by Bhattacharya, Pratt, Pulaverty, and Morgan, IBM;发布于 2003 的 OLS
- Design Notes on Asynchronous I/O (aio) for Linux by Suparna

Bhattacharya -- SGI 的 KAIO 和其他几个 AIO 项目与 Ben 的 AIO 进行比较

- Linux AIO home page - Ben 的初步补丁，邮件列表等
- linux-aio mailing list archives
- libaio-oracle - 库实现标准 POSIX AIO 最上层的 libaio. First mentioned by Joel Becker on 18 Apr 2003.

Suparma 建议先看看 AIO 的 API。

RedHat AS 和 Suse SLES 都在 2.4 的内核中提供了高性能的实现，与 2.6 的内核实现相似，但并不完全一样。

2006.2，在网络 AIO 有了一个新的尝试，具体请看 Evgeniy Polyakov 的基于 kevent 的 AIO.

1999，**SGI 为 Linux 实现了一个高速的 AIO**，在到 1.1 版本时，据说可以很好的工作于磁盘 I/O 和网 络套接字，且使用了内核线程。目前该实现依然对那些不能等待 Ben 的 AIO 套接字支持的人来说是很有用的。

O'Reilly 的"POSIX.4: Programming for the Real World"一书对 aio 做了很好的介绍.

这里 有一个指南介绍了早期的非标准的 aio 实现，可以看看，但是请记住你得把"aioread"转换为"aio_read"。

注意 AIO 并没有提供无阻塞的为磁盘 I/O 打开文件的方法，如果你在意因打开磁盘文件而引起 sleep 的话，Linus 建议 你在另外一个线程中调用 open()而不是把希望寄托在对 aio_open()系统调用上。

在 Windows 下，异步 I/O 与术语"重叠 I/O"和"IOCP"(I/O Completion Port,I/O 完成端口)有一定联系。Microsoft 的 IOCP 结合了 先前的如异步 I/O(如 aio_write)的技术，把事件完成的通知进行排队(就像使用了 aio_sigevent 字段的 aio_write)，并且它 为了保持单一 IOCP 线程的数量从而阻止了一部分请求。更多信息请看 Mark russinovich 在 sysinternals.com 上的文章 Inside I/O Completion Ports，Jeffrey Richter 的书 "Programming Server-Side Applications for Microsoft Windows 2000" (Amazon, MSPress), U.S. patent #06223207, or MSDN.

4. 一个服务线程服务一个客户端，使用阻塞 I/O

让 read()和 write() 阻塞。这样不好的地方在于需要为每个客户端使用一个完整的栈，从而比较浪费内存。许多操作系统仍在处理数百个线程时存在一定的问题。如果每个线程使用 2MB 的栈，那么当你在 32 位的机器上运行 512（2^30 / 2^21=512）个线程时，你就会用光所有的 1GB 的用户可访问虚拟内存（Linux 也是一样运行在 x86 上的）。你可以减小每个线程所拥有的栈内存大小，但是由于大部分线程库在一旦线程创建后就不能增大线程栈大小，所以这样做就意味着你必须使你的程序最小程度地使用内存。当然你也可以把你的程序运行在 64 位的处理器上去。

Linux，FreeBSD 和 Solaris 系统的线程库一直在更新，64 位的处理器也已经开始在大部分的用户中所使用。也许在不远的将来，这些喜欢使用一个线程来服务一个客户端的人也有能力服务于 10000 个客户了。但是在目前，如果你想支持更多的客户，你最好还是使用其它的方法。

那些厚着脸皮赞成使用线程的观点，如加州大学伯克利分校的 von Behren,Condit 和 Brewer 在 HostOS IX 项目中提出的论文 Why Events Are A Bad Idea (for High-concurrency Servers)。有没有反对使用线程的人要反驳此论文呢?:-)

LinuxTheads

LinuxTheads 是标准 Linux 线程库.自 glib2.0 后已经将 LinuxTheads 集成，同时 LinuxTheads 是兼容 Posix 的，但是不支持优秀的信号量机制。

NGPT：下一代 Linux Posix 线程工程

NGPT 是 IBM 发起的兼容 Posix 线程的 Linux 工程。现在的稳定版本是 2.2，而且工作的很稳定...但是 NGPT 团队宣称他们已经对 NGPT 是仅支持模式，因为他们觉得“从长远的角度看，社区贡献是最好的方式”。NGPT 团队将继续致力于 Linux 线程的研究，但是目前他们关注 NPTL（Native POSIX Thread Library 比较译者加注）。(由于 NGPT 团队辛勤的工作，他们已经被 NPTL 所认可)

NPTL：Linux 的本地 Posix 线程库

NPTL 是由 Ulrich Drepper (glibc 的主要维护人员)和 Ingo Molnar 发起的项目，目的是提供 world-class 的 Posix Linux 线程支持。

2003.10.5，NPTL 作为一个 add-on 目录（就像 linuxthreads 一样）被合并到 glibc 的 cvs 树中，所以很有可能随 glibc 的下一次 release 而 一起发布。Red Hat 9 是最早的包含 NPTL 的发行版本（对一些用户来说有点不太方便，但是必须有人来打破这沉默[break the ice]...)

NPTL links：

- NPTL 讨论的邮件列表
- NPTL 源码
- NPTL 的最初发表
- 最初的描述 NPTL 目标的白皮书
- 修改的 NPTL 的最后设计的白皮书
- Ingo Molnar 最初的基准测试表明可以处理 10^6 个线程
- Ulrich 的基准测试 比较了 LinuxThreads，NPTL 和 IBM 的 NGPT 的各自性能，结果看来 NPTL 比 NGPT 快的多。

这是我尝试写的描述 NPTL 历史的文章(也可以参考 Jerry Cooperstein 的文章):

2002.3，NGPT 小组的 Bill Abt，glibc 的维护者 Ulrich Drepper 和其它人召开了个会议来探讨 LinuxThreads 的发展，会议的一个 idea 就是要改进 mutex 的性能。 Rusty Russell 等人 随后实现了 fast userspace mutexes (futexes), （如今已在 NGPT 和 NPTL 中应用了）。 与会的大部分人都认为 NGPT 应该合并到 glibc 中。

然而 Ulrich Drepper 并不怎么喜欢 NGPT，他认为他可以做得更好。 (对那些曾经想提供补丁给 glibc 的人来说，这应该不会令他们感到惊讶:-) 于是在接下来的几个月里，Ulrich Drepper, Ingo Molnar 和其它人致力于 glibc 和内核的改变，然后就弄出了 Native Posix Threads Library (NPTL). NPTL 使用了 NGPT 设计的所有内核改进（kernel enhancement），并且采用了几个最新的改进。Ingo Molnar 描述了 一下的几个内核改进：

> *NPTL 使用了三个由 NGPT 引入的内核特征：getpid() 返回 PID，CLONE_THREAD 和futexes; NPTL 还使用了(并依赖)也是该项目的一部分的一个更为wider 的内核特征集。*
>
> *一些由 NGPT 引入内核的 items 也被修改，清除和扩展，例如线程组的处理 (CLONE_THREAD). [the CLONE_THREAD changes which impacted NGPT's compatibility got synced with the NGPT folks, to make sure NGPT does not break in any unacceptable way.]*
>
> *这些为 NPTL 开发的并且后来在 NPTL 中使用的内核特征都描述在设计白皮书中， http://people.redhat.com/drepper/nptl-design.pdf ...*
>
> *A short list: TLS support, various clone extensions (CLONE_SETTLS, CLONE_SETTID, CLONE_CLEARTID), POSIX thread-signal handling, sys_exit() extension (release TID futex upon VM-release), the sys_exit_group() system-call, sys_execve() enhancements and support for detached threads.*
>
> *There was also work put into extending the PID space - eg. procfs crashed due to 64K PID assumptions, max_pid, and pid allocation scalability work. Plus a number of performance-only improvements were done as well.*
>
> *In essence the new features are a no-compromises approach to 1:1 threading - the kernel now helps in everything where it can improve threading, and we precisely do the minimally necessary set of context switches and kernel calls for every basic threading primitive.*

NGPT 和 NPTL 的一个最大的不同就是 NPTL 是 1:1 的线程模型，而 NGPT 是 M:N 的编程模型(具体请看下面). 尽管这样， Ulrich 的最初的基准测试 还是表明 NPTL 比 NGPT 快很多。(NGPT 小组期待查看 Ulrich 的测试程序来核实他

的结果.)（转载于：http://www.cnblogs.com/fll/archive/2008/05/17/1201540.html）

FreeBSD 线程支持

FreeBSD 支持 LinuxThreads 和用户空间的线程库。同样，M:N 的模型实现 KSE 在 FreeBSD 5.0 中引入。具体请查看 www.unobvious.com/bsd/freebsd-threads.html。

2003.3.25, Jeff Roberson 发表于 freebsd-arch：

> *...感谢 Julian, David Xu, Mini, Dan Eischen,和其它的每一位参加了 KSE 和 libpthread 开发的成员所提供的基础， Mini 和我已经开发出了一个 1：1 模型的线程实现，它可以和 KSE 并行工作而不会带来任何影响。实际上，它有助于把 M：N 线程共享位测试. ...*

2006.7, Robert Watson 提议 1:1 的线程模型应该为 FreeBSD 7.x 的默认实现：

> *我知道曾经讨论过这个问题，但是我认为随着 7.x 的向前推进，这个问题应该重新考虑。 在很多普通的应用程序和特定的基准测试中，libthr 明显的比 libpthread 在性能上要好得多。 libthr 是在我们大量的平台上实现的，而 libpthread 却只有在几个平台上。 最主要的是因为我们使得 Mysql 和其它的大量线程的使用者转换到"libthr"，which is suggestive, also! ... 所以 strawman 提议:让 libthr 成为 7.x 上的默认线程库。*

NetBSD 线程支持

根据 Noriyuki Soda 的描述：

> *内核支持 M:N 线程库是基于调度程序激活模型，合并于 2003.1.18 当时的 NetBSD 版本中。*

详情请看 Nathan J. Williams, Wasabi Systems, Inc.在 2002 年的 FREENIX 上的演示 An Implementation of Scheduler Activations on the NetBSD Operating System。

Solaris 线程支持

Solaris 的线程支持还在进一步提高 evolving... 从 Solaris 2 到 Solaris 8，默认的线程库使用的都是 M:N 模型，但是 Solaris 9 却默认使用了 1:1 线程模型.查看 Sun 多线程编程指南 和 Sun 的关于 Java 和 Solaris 线程的 note。

Java 在 JDK 1.3.x 及更早的线程支持

大家都知道，Java 一直到 JDK1.3.x 都没有支持任何处理网络连接的方法，除了一个线程服务一个客户端的模型之外。Volanomark 是一个不错的微型测试

程序，可以用来测量在 某个时候不同数目的网络连接时每秒钟的信息吞吐量。在 2003.5, JDK 1.3 的实现实际上可以同时处理 10000 个连接，但是性能却严重下降了。从 Table 4 可以看出 JVMs 可以处理 10000 个连接，但是随着连接数目的增长性能也逐步下降。

注意: 1:1 线程 vs M:N 线程

在实现线程库的时候有一个选择就是你可以把所有的线程支持都放到内核中（也就是所谓的 1：1 的模型），也可以把一些线程移到用户空间上去（也就是所谓的 M：N 模型）。从某个角度来说，M:N 被认为拥有更好的性能，但是由于很难被正确的编写，所以大部分人都远离了该方法。

- 为什么 Ingo Molnar 相对于 M：N 更喜欢 1：1
- Sun 改为 1：1 的模型
- NGPT 是 Linux 下的 M：N 线程库
- Although Ulrich Drepper 计划在新的 glibc 线程库中使用 M：N 的模型, 但是还是选用了 1：1 的模型
- MacOSX 也将使用 1：1 的线程
- FreeBSD 和 NetBSD 仍然将使用 M：N 线程，FreeBSD 7.0 也倾向于使用 1：1 的线程（见上面描述），可能 M：N 线程的拥护者最后证明它是错误的

5. 把服务代码编译进内核

Novell 和 Microsoft 都宣称已经在不同时期完成了该工作，至少 NFS 的实现完成了该工作。khttpd 在 Linux 下为静态 web 页面完成了该工作， Ingo Molnar 完成了"TUX" (Threaded linUX webserver) ，这是一个 Linux 下的快速的可扩展的内核空间的 HTTP 服务器。 Ingo 在 2000.9.1 宣布 alpha 版本的 TUX 可以在 ftp://ftp.redhat.com/pub/redhat/tux 下载, 并且介绍了如何加入其邮件列表来获取更多信息。

在 Linux 内核的邮件列表上讨论了该方法的好处和缺点，多数人认为不应该把 web 服务器放进内核中，相反内核加入最小的钩子 hooks 来提高 web 服务器的性能，这样对其它形式的服务器就有益。具体请看 Zach Brown 的讨论 对比用户级别和内核的 http 服务器。在 2.4 的 linux 内核中为用户程序提供了足够的权力（power），就像 X15 服务器运行的速度和 TUX 几乎一样，但是它没有对内核做任何改变。

6. 把 TCP 栈放入用户空间

See for instance the netmap packet I/O framework, and the Sandstorm proof-of-concept web server based on it.

注释

Richard Gooch 曾经写了一篇讨论 I/O 选项的论文。

在 2001, Tim Brecht 和 MMichal Ostrowski 为使用简单的基于 select 的服务器 做了各种策略的测试 测试的数据值得看一看。

在 2003, Tim Brecht 发表了 userver 的源码, 该 web 服务器是整合了 Abhishek Chandra, David Mosberger, David Pariag 和 Michal Ostrowski 所写的几个服务器，可以使用 select(), poll(), epoll()和 sigio.

早在 1999 年, Dean Gaudet 就表示:

我一直在问"为什么你们不使用基于 select/event 像 Zeus 的模型，它明显是最快的。"...

他们不使用它的原因可以简单归结为"太难理解了，并且其中的性能指标还不清楚"，但是几个月后，当该模型变得易懂时人们就开始愿意使用它了。

Mark Russinovich 写了一篇评论和文章讨论了在 2.2 的 linux 内核只能够 I/O 策略问题。 尽管某些地方似乎有点错误，不过还是值得去看。特别是他认为 Linux2.2 的异步 I/O (请看上面的 F_SETSIG) 并没有在数据准备好时通知用户进程，而只有在新的连接到达时才有。这看起来是一个奇怪的误解。还可以看看 早期的一些 comments, Ingo Molnar 在 1999.4.30 所举的反例, Russinovich 在 1999.5.2 的 comments, Alan Cox 的 反例，和各种 linux 内核邮件. 我怀疑他想说的是 Linux 不支持异步磁盘 I/O，这在过去是正确的，但是现在 SGI 已经实现了 KAIO，它已不再正确了。

查看页面 sysinternals.com 和 MSDN 了解一下"完成端口"，据说它是 NT 中独特的技术，简单说，win32 的"重叠 I/O"被认为是太低水平而不方面使用，"完成端口"是提供了完成事件队列的封装，再加上魔法般的调度，通过允许更多的线程来获得完成事件如果该端口上的其它已获得完成事件的线程处于睡眠中时（可能正在处理阻塞 I/O），从而可以保持运行线程数目恒定。

查看 OS/400 的 I/O 完成端口支持。（转载于：http://www.cnblogs.com/fll/archive/2008/05/17/1201540.html）

在 1999 年 9 月，有一场很有趣的关于 linux 内核的讨论，它叫做 "> 15,000 个并发连接"（主题的第二周）。 重点如下：

- Ed Hall 根据他的经验发表一些他的观点；他在跑着 Solaris 系统的 UP P2/333 上完成了超过 1000 连接/秒的任务。他代码里使用很少的线程(一两个 CPU)，每个线程管理大量的使用事件模型模拟的用户
- Mike Jagdis posted 发表的分析 poll/select 系统开销的文章并说 "当前的

select/poll 实现部分可以改良，特别在阻塞状态下， 但系统开销仍然会随着描述符的增加而增加，因为 select/poll 不会也不能记住描述符参与了什么。不过很容易通过使用新的 API 来修复这个问题。欢迎提出建议....."

- Mike 发表了 关于 改善 select()和 poll()的文章。
- Mike 发表了一个可能替换 poll()/select()的 API: "为何不试试'类设备'的 API？你可以用它写出类 pollfd 的结构，然后“设备”就会侦探事件并在你读取它的时候传送类 pollfd 结构表示的结果... "
- Rogier Wolff 推荐使用 "digital guys 推荐的 API", http://www.cs.rice.edu/~gaurav/papers/usenix99.ps
- Joerg Pommnitz 指出 that 任何新 API 都应不仅能等待文件描述符事件，还要能受信和处理 SYSV-IPC。我们的同步主类型至少应该能够做到 Win32 下的 WaitForMultipleObjects 能够做的。
- Stephen Tweedie 指出 F_SETSIG, queued realtime signals 和 sigwaitinfo() 的结合是 API 的超集 详见 http://www.cs.rice.edu/~gaurav/papers/usenix99.ps. 他同时提到如果你对性能感兴趣，你应该把信号分段，而不是把信号异步传送，处理器在队列中抢占下一个 sigwaitinfo()调用。
- Jayson Nordwick 用 F_SETSIG 同步事件模型比较完成端口，并得出了相似的结论。
- Alan Cox 指出老版本的 SCT's SIGIO 补丁包括在了 2.3.18ac.
- Jordan Mendelson 贴出了一些示例代码并演示如何使用 F_SETSIG.
- Stephen C. Tweedie 继续比较完成端口和 F_SETSIG, 并称: "在信号出队列的机制下，如果类库使用同样的机制，那么你的应用程序就会为不同库组件预定受信" ，类库也可以建立自己的受信句柄，这样就不会过多地影响程序本身了。
- Doug Royer 指出当他在 Sun 的日程服务器上工作时，他已经在 Solaris 2.6 上建立了 100,000 个连接。其他人也附和地谈到在 Linux 上需要多少 RAM，可能会遇到什么瓶颈。

打开文件句柄的限制

- 任何 Unix 系统：限制数由 ulimit 或 setrlimit 来设置
- Solaris 系统：参考 the Solaris FAQ, question 3.46 (或前后不久，他们定期地重新编号了问题)
- FreeBSD 系统：

  编辑 /boot/loader.conf 文件，增加这一行：
  set kern.maxfiles=XXXX

这里 XXXX 是所期望的文件描述符的系统限制，然后重启。感谢一位匿名读者，回复说他在 FreeBSD 4.3 上已经做到设置超过 10000 的连接数：

> *FWIW：实际上你不能通过 sysctl 等随意地在 FreeBSD 里调节最大连接数，你必须在 /boot/loader.conf 文件里做。*

理由是 在系统启动早期，初始化 socket 和 tcpcn 结构体区域时调用 zalloci() 调用，以便该区是键入稳定的和热拔插的。
你仍然需要设置较高的 mbun 数，因为你将(在一个未修改过的内核上)让每一个 tcptempl 结构体连接都消耗一个 mbuf，该链接用于实现保持活跃。

另一位读者说：

如在 FreeBSD 4.4 里，tcptempl 结构体不再被分配内存，你也无需非要担心每个连接都吃掉一个 mbuf 。

另见：

- the FreeBSD handbook
- SYSCTL TUNING, LOADER TUNABLES, 和 KERNEL CONFIG TUNING 在 'man tuning'
- The Effects of Tuning a FreeBSD 4.3 Box for High Performance, Daemon News, Aug 2001
- postfix.org tuning notes, covering FreeBSD 4.2 和 4.4
- the Measurement Factory's notes, circa FreeBSD 4.3

- OpenBSD 系统： 一位读者说

在 OpenBSD 里，需要额外的调整以增加每个进程可打开的文件句柄数：/etc/login.conf 中的 openfiles-cur 参数需要增大。 你可以通过 sysctl -w 或在 sysctl.conf 更新 kern.macfiles ，但这并没有影响。这个情况是因为在发布时，login.conf 对于非授权进程被限制为较低的 64，授权进程限制为 128。

- Linux 系统：见 Bodo Bauer's /proc documentation 。

在 2.4 内核上：

```
echo 32768 > /proc/sys/fs/file-max
```

增加打开文件数的系统限制，并

```
ulimit -n 32768
```

以增加并发进程数限制。

在 2.2.x 内核上：

```
echo 32768 > /proc/sys/fs/file-max
echo 65536 > /proc/sys/fs/inode-max
```

增加打开文件数的系统限制，并

```
ulimit -n 32768
```

以增加并发进程数限制。

我验证发现 Red Hat 6.0（2.2.5 或补丁版）上用这种方式能打开至少 31000 个文件描述符。另一个哥们也验证发现在 2.2.12 上一个进程用这种方式可以打开至少 90000 个文件描述符（适当的限制）。上限似乎取决于内存。

Stephen C. Tweedie 发帖讲述过如何在系统引导期间使用 initscript 和 pam_limit 来设置 ulimit 以限制全局或单个用户。
在较旧的 2.2 内核上，每个进程可打开的文件数仍然被限制为 1024，即使有上述的变动。
参见 Oskar's 1998 post ，探讨了在 2.0.36 内核里每个进程以及系统范围内的文件描述符限制。

线程使用的限制

不管是何种体系架构，你都应该在每一个线程里尽可能的减少栈空间的使用，这样以防止耗光虚拟内存。如果你用 pthreads，你可以在运行时调用 pthread_attr_init 函数进行设置。

- Solaris:据我所知，它根据内存大小可以创建非常多的线程。
- Linux 2.6 内核上的 NPTL:在/proc/sys/vm/max_map_count 需要设置最多增加到 32000 个线程。（除非你在 64 位处理器上工作，在大量线程工作时，你必须用栈空间小的线程。）看看 NPTL 的邮件列表，如，有个关于线程主题的“不能创建超过 32K 的线程？”，以此你可以获得更多的信息。
- Linux 2.4：/proc/sys/kernel/threads-max 是创建线程数的最大值，在我的 Red Hat8 系统中默认是 2047。你可以用 echo 命令来增加数目，如：“echo 4000 > /proc/sys/kernel/threads-max”
- Linux 2.2：2.2.13 内核至少在 Intel 平台上是限制线程数目的，在其他平台上的情况我就不知道了。Mingo 在 Intel 上的补丁版本 2.1.131 中取消了线程数的限制。正式版会出现在 2.3.20 版本上.你可以参考 Volano 中有关文件，线程和 FD_SET 限制的详细介绍。哦，这个文档讲了很多对你来说很难的东西，但有些是过时了。
- Java:看看 Volano 详细的介绍，里面讲到 Java 是如何适应很多不同的系统，但却能处理很多的线程。

Java 相关问题

在 JDK1.3 及以前，标准的 Java 网络库主要提供每个连接单独线程处理的模型。这是一种非阻塞读的工作方式，但并不支持非阻塞写。

2001 年 5 月，JDK 1.4 引入了 java.nio 包以支持完整的非阻塞 I/O 操作（以及其他优点）。详情请见发行说明。请尝试使用并向 Sun 提交反馈。
惠普（HP）的 Java 库也包含了线程轮询 API。

2000 年，Matt Welsh 在 Java 上实现了非阻塞网络接口（socket）； 对这个实现的负载测试表明在服务端处理大量（达到 10000）连接时，非阻塞的方式比阻塞更具优势。这个库叫做 java-nbio： Sandstorm 项目的一部分。负载测试报告可参见 10000 连接的性能 。

另请参考 Dean Gaudet 论文中关于 Java、网络 I/O 以及线程部分；Matt Welsh 的论文：事件驱动 vs 工作线程。

在 NIO 之前，改进 Java 网络 API 主要有以下几种建议：

- Matt Welsh 的捷豹系统建议预先序列化对象、使用新的 Java 字节码以及改造内存管理机制从而使 Java 支持异步 I/O。
- C-C. Chang 和 T. von Eicken 提出将 Java 抽象成虚拟接口架构（Virtual Interface Architecture）。他们建议改造内存管理机制从而使 Java 支持异步 I/O。
- JSR-51 是 Sun 的一个项目，它引入了 java.nio 包。Matt Welsh 也参与其中。（谁说 Sun 不听他人意见？）

其他提示

零拷贝

通常，数据在传输的过程中会被拷贝好多次。一些消除这些拷贝直到达到绝对最小值的方案被称为“零拷贝”。

1. Thomas Ogrisegg's zero-copy send patch，用于 Linux 2.4.17-2.4.20 下的 mmaped 文件。它被号称快于 sendfile()。
2. IO-Lite 是一组去掉了多次不必要拷贝的 I/O 基本元素提案。

- 早在 1999 年，Alan Cox 提出零拷贝有时不值得去费心。(尽管他也喜欢 sendfile()。)
- 2000 年七月，在 TUX 1.0 的 2.4 内核上，Ingo 实现了 TCP 零拷贝的一种形式，并称他会很快在用户空间上实现这一功能。
- Drew Gallatin 和 Robert Picco 为 FreeBSD 增添了一些零拷贝特性；这主意看来就像：如果你在一个 socket 中调用 write() 或 read()，指针是按页对齐，同时要传输的数据量至少是一页，*同时*你不立即重用缓冲，使用内存管理诀窍来避免拷贝。但是看一看 linux-kernel 上这个消息的跟贴，人们对这些内存管理诀窍的速度的担忧。

  根据来自 Noriyuki Soda 的说明：
  自 NetBSD-1.6 发布，通过指定 "SOSEND_LOAN" 内核选项，来支持发送端零拷贝。现在这个选项在 NetBSD-current 中是默认的（你可以通过在 NetBSD_current 的内核选项中指定 "SOSEND_NO_LOAN" 来禁用这项特性）。使用这个特性，如果要发送的数据超过 4096 字节，零拷贝自动启用。
- sendfile() 系统调用能够实现零拷贝的网络工作。
  Linux 和 FreeBSD 中的 sendfile() 函数让你告诉内核去发送一个文件的部分或全部。这让操作系统尽可能高效地去做。它也可以同样用在使用线程或非阻塞 I/O 的服务器上。（在 Linux 上，目前缺乏文档；使用 syscall4 调用它。Andi Kleen 正在写一个关于这个的新手册页。参阅在 in Linux Gazette issue 91 上，Jeff Tranter 的 探索 sendfile 系统调用。）有传言称，ftp.cdrom.com 从 sendfile() 中显著获益。
  一个 2.4 内核上的 sendfile() 零拷贝实现正处于进程之中。参见 LWN Jan 25 2001。

一个在 Freebsd 上使用 sendfile() 的开发者报告说用 POLLWRBAND 取代 POLLOUT 会有大改观。
Solaris 8 （截至 2001 年 7 月的更新）有一个新的系统调用 'sendfilev'。 这里有一个手册页的副本。Solaris 8 7/01 发布说明 也提到了它。我怀疑这将会在阻塞模式下发送到一个 socket 时非常有用；而在非阻塞 socket 下使用时会有点痛苦。

通过使用 writev （或 TCP_CORK）避免小帧

Linux 下一个新的 socket 选项，TCP_CORK，告诉内核避免发送并不完全的帧，这有些帮助，比如出于某些原因，你不能将许多小的 write() 调用聚集在一起时。去掉这个选项刷新缓冲。尽管用 writev() 更好……

参阅 LWN Jan 25 2001，在 linux-kernel 上关于 TCP_CORK 和一个可能的替代者 MSG_MORE 的一些非常有趣的讨论的总结。

对过载切合实际地做出反应

[Provos, Lever, and Tweedie 2000] 记录了当服务器过载时，丢弃传入的连接，改善了性能曲线的形状，同时减少了整体错误率。 他们使用了一个“I/O 就绪的客户端的数目”的一个平滑描述作为过载的一个尺度。这种技术应该可以很容易地应用于使用 select，poll，或每次调用返回一个就绪事件计数的其他系统调用（如 /dev/poll 或 sigtimedwait4()）编写的服务器上。

一些程序可以从使用非 Posix 线程中获益

并非所有的线程生来相同。Linux 中的 clone()（和其他操作系统中它的朋友们）让你创建一个拥有它自己当前工作目录的线程，例如，当实现一个 ftp 服务器时，这将非常有用。参看 Hoser FTPd，它是一个使用本地线程而非 pthreads 的例子。

缓存自己的数据有时可以取得双赢

5 月 9 号，new-httpd 上，Vivek Sadananda Pai (vivek@cs.rice.edu) 在 "Re: fix for hybrid server problems"中陈述：

> *“在 FreeBSD 和 Solaris/x86 上，我都比较了一个基于 select 的服务器和一个多处理器服务器的原始（raw）性能。在微型基准（microbenchmarks）方面，仅有一些来源于软件架构的细微性能差别。基于 select 的服务器的性能大获全胜源于做了一些应用程序级别的缓存。同时多处理器服务器可以用更高的代价做这件事，在实际的工作负担上很难获得同等收益（较之微型基准）。我要将这些测量作为下一期的 Usenix 会议上出现的论文的一部分。如果你有 postscript，可在 http://www.cs.rice.edu/~vivek/flash99/ 上*

找到这篇论文。”

其它限制

- 老的系统库使用 16 位的变量来保存文件句柄，当有超过 32767 个句柄时，这就会引起麻烦。glibc2.1 应该没有这个问题.
- 许多系统使用 16 位变量来保存进程或线程 ID。It would be interesting to port the Volano scalability benchmark to C, and see what the upper limit on number of threads is for the various operating systems.
- 太多的线程局部内存被某些操作系统事件分配好，如果每个线程分配 1MB，而总共的虚拟内存空间才 2GB，这会造成只能生成 2000 个线程的上限。
- 参看这个页面最后的性能比较图。http://www.acme.com/software/thttpd/benchmarks.html. Notice how various servers have trouble above 128 connections, even on Solaris 2.6? 如果有人想出为什么，请告诉我。

  注意 :如果 TCP 栈存在一个 bug，而引起在 SYN 或 FIN 时间上的小小的延时（200ms）,这在 Linux 2.2.0-2.2.6 中存在的，操作系统或 http 后台程序在打开的连接数上有一个硬性限制, you would expect exactly this behavior. There may be other causes.

内核问题

对 Linux 来说，核心的瓶颈正不断的被突破。可以查看 Linux Weekly News, Kernel Traffic, the Linux-Kernel mailing list, 和 my Mindcraft Redux page.

1999 年 3 月，微软主办了一次基准测试来比较 NT 和 Linux，比较他们在可服务的 http 和 smb 客户的最大数量上面的性能。结果显示 Linux 性能不佳。更多的信息可以参考 my article on Mindcraft's April 1999 Benchmarks。

The Linux Scalability Project. They're doing interesting work, including Niels Provos' hinting poll patch, and some work on the thundering herd problem.

Mike Jagdis' work on improving select() and poll(); here's Mike's post about it.

Mohit Aron (aron@cs.rice.edu) writes that rate-based clocking in TCP can improve HTTP response time over 'slow' connections by 80%.

测试服务器性能

两种测试很简单、也有趣但同时也很难。

1. 每秒钟原始连接数。(你可以在一秒钟内处理多少个512字节的文件呢？)
2. 有很多慢速客户端时在大文件上的传输速率(可以支持多少个 28.8k 的调制解调器类型的客户端同时从你的服务器下载，在你的性能急剧下降前?)

Jef Poskanzer 发表了许多关于比较 web 服务器性能的基准测试。参看 http://www.acme.com/software/thttpd/benchmarks.html 他的结果。

我的笔记 关于 thttpd 和 Apache 的对比 可能适合初学者查看。Chuck Lever 一直提醒我们 Banga and Druschel 的 "关于 web 服务器基准测试"论文 值得一读。

IBM 的精彩论文 Java 服务器基准测试 [Baylor et al, 2000]. 也值得一看。

示例

Nginx 是一个基于目标系统的高效率网络事件机制的 Web 服务器。它愈发受欢迎起来；甚至有相关书籍。

有趣的基于 select() 服务器

- thttpd 很简单的。使用单一进程。它具有良好的性能，但不会随 CPU 的数目扩展。也可使用 kqueue。
- mathopd。同 thttpd 类似。
- fhttpd
- boa
- Roxen
- Zeus，一个试图成为速度最快的商业化服务器软件。见其 调整指南。
- 其它非 Java 服务器列表于 http://www.acme.com/software/thttpd/benchmarks.html
- BetaFTPd
- Flash-Lite - 使用 IO-Lite 的 Web 服务器
- Flash：一个高效可移植 Web 服务器 -- 使用 select(), mmap(), mincore()
- 2003 年度 Flash Web 服务器 -- 使用 select(), modified sendfile(), async open()
- xitami - 使用 select() 实现自己的线程抽象，以移植到无线程系统。
- Medusa - 一个 Python 服务器编写工具包，试图提供高性能。
- userver - 可使用 select, poll, epoll, 或 sigio 功能的小型 http 服务器。

有趣的基于 /dev/poll 的服务器

- 2000 年五月，*N. Provos, C. Lever*, "Linux 平台可扩展网络 I/O," 「2000 年 6 月，加利福尼亚州圣地亚哥，FREENIX track, Proc。USENIX 2000」描述了一个支持 /dev/poll 的 thttpd 修改版。同 phhttpd 进行了性能比较。

有趣的基于 kqueue() 的服务器

- thttpd (从 2.21 开始?)
- Adrian Chadd 说 "为了使 Squid 像一个真正的基于内核队列的 IO 系统，我做了很多工作"; 它是一个官方的 Squid 子项目；参见 http://squid.sourceforge.net/projects.html#commloops. (这显然比 Benno 的 patch 更加新)

有趣的基于实时信号的服务器

- Chromium's X15. 使用了 2.4 内核的 SIGIO 特性加上 sendfile()和 TCP_CORK,其宣称拥有比 TUX 更快的速度. 基于社区源码许可下(非开源)的 源码 已经发布。查看 Fabio Riccardi 发布的原始公告.
- Zach Brown's phhttpd - "为了展示 sigio/siginfo 事件模型而编写的快速 web 服务器。考虑它的嗲吗是一个高实验性的，如果你要将它使用到生产环境，那一将要花费很多的脑力"。要使用 siginfo 特性你需要 2.3.21 或以后的版本，它们包含了针对早期内核的补丁。其宣传比 khttpd 更快。查看 1999.5.31 号发布的邮件 获取更多信息。

值得关注的基于线程的服务器

- Hoser FTPD. See their benchmark page.
- Peter Eriksson's phttpd and
- pftpd
- The Java-based servers listed at http://www.acme.com/software/thttpd/benchmarks.html
- Sun's Java Web Server (which has been reported to handle 500 simultaneous clients)

值得关注的基于内核中的服务器

- khttpd
- "TUX" (Threaded linUX webserver) by Ingo Molnar et al. For 2.4 kernel.

其它值得关注的链接

- Jeff Darcy's notes on high-performance server design
- Ericsson's ARIES project -- benchmark results for Apache 1 vs. Apache 2 vs. Tomcat on 1 to 12 processors
- Prof. Peter Ladkin's Web Server Performance page.
- Novell's FastCache -- claims 10000 hits per second. Quite the pretty performance graph.
- Rik van Riel's Linux Performance Tuning site