高并发环境下 Apache 与 Nginx 的 I/O 性能比较①

凌质亿, 刘哲星, 曹 蕾

(南方医科大学 生物医学工程学院, 广州 510515)

摘 要: 通过分析 linux 环境下的 I/O 模型以及 Apache 和 Nginx 的 I/O 事件驱动机制, 比较二者在高并发网络环境下的 I/O 性能. 文中结合应用实例, 综合对比了高并发环境下 Apache 与 Nginx 的功能与用途, 总结和讨论了各自的优缺点, 并就二者的选用提出建议.

关键词: Apache; Nginx; I/O 模型; 高并发

I/O Performance Comparison of Apache and Nginx in High Concurrence Environment

LING Zhi-Yi, LIU Zhe-Xing, CAO Lei

(School of Biomedical Engineering, Southern Medical University, Guangzhou 510000, China)

Abstract: A comparative study is conducted on the I/O performance between the popular web service applications of Apache and Nginx in high concurrence environment, based on the analysis of linux I/O model and the related I/O event-driven mechanism of Apache and Nginx. The functionality and usage are also compared with an application instance. Both the advantages and disadvantages of Apache and Nginx are discussed. Some suggestions for properly choosing of the 2 applications are given in the end.

Key words: Apache; Nginx; I/O model; high concurrence

随着计算机科学的进步, 网络应用得到了快速的发展. 呈爆发式增长的网络用户和频繁的网络资源访问, 迫使网络服务器常常工作在高并发访问的环境下. 因此高并发环境下的 I/O 性能成为网络服务器的瓶颈之一. 除了利用最新的高速、高带宽的硬件资源以外, 具有先进的网络 I/O 模型的高性能网络服务程序的也是最大化利用现有硬件资源, 提升网络服务器性能的重要保证. 本文研究了高并发网络环境下 Apache 与 Nginx 这两种 web 服务器的 I/O 性能.

1 Apache与Nginx

1.1 简介

Apache 是一个开放源代码的网页服务器. 由于其可以在大多数电脑操作系统下运行和具有较高的安全性而被广泛使用, 是目前使用排名第一的 web 服务器.

Nginx 是一个高性能的 web 服务器, 也可作为 IMAP/POP3/SMTP 代理服务器, 以其丰富的功能集、稳定性和低系统资源消耗而闻名.

1.2 综合性能对比

Apache 作为目前应用最广泛的服务器, 具有以下优点[1]:

1) 可移植性. 几乎所有的平台都能支持运行 Apache, 普及性广.

2) 开源性. 完全免费, 且拥有一支热爱开源的开发团队, 漏洞填补及时, 安全性能高.

3) 稳定性. Apache 所有的配置都保存在配置文件中, 一般不会出现假死.

4) 扩展性. Apache 模块数多, 提供大量功能, 且支持 php、jsp、asp 等多种编程语言.

但是, 在面对当今的 C10K[2]问题时, Apache 显得力不从心. 为此, Igor Sysoev 编写出了 Nginx. 目前, Nginx 正得到越来越广泛的应用. 选择 Nginx 的理由在于[3]:

1) 支持高并发连接, 合理优化配置 nginx+php(fastcgi)可以承受30000以上并发连接数, 如

① 基金项目:国家自然科学基金青年基金(61102114);广东省教育部产学研结合引导项目(2011B090400037)
收稿时间:2012-12-06;收到修改稿时间:2013-01-10

图 1 所示(图片摘自参考文献[3]).

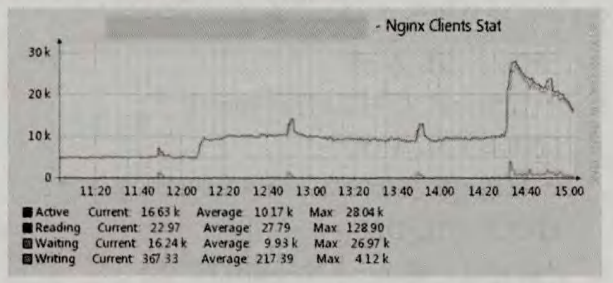

图 1. 金山游戏官方网站 2009 年 nginx 集群连接数

2) 内存消耗少, 在实际应用环境下, 单台服务器 nginx+php5(fastcgi)处理程序能力已超过 700 次/秒, 但 CPU 负载并不高, 如图 2 所示.

```
top - 15:04:42 up 55 days, 18:30,  1 user,  load average: 3.93, 4.40, 4.
Tasks: 380 total,   3 running, 377 sleeping,   0 stopped,   0 zombie
Cpu(s): 34.0% us, 20.5% sy,  0.0% ni, 45.5% id,  0.0% wa,  0.0% hi,  0.0
Mem:   4149156k total,  3630112k used,   519044k free,    42620k buffers
Swap:  2096440k total,     3364k used,  2093076k free,  2937700k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2672 www       16   0 85340 8252 6376 S  7.6  0.2   0:01.44 php-cgi
29247 www       16   0 13316  10m  844 S  2.6  0.3  28:18.26 nginx
29245 www       15   0 13084  10m  844 S  2.0  0.2  26:21.69 nginx
 2616 www       15   0 85340 8280 6400 S  1.7  0.2   0:01.68 php-cgi
 2664 www       16   0 85336 8236 6372 S  1.7  0.2   0:01.09 php-cgi
 2681 www       15   0 85340 8268 6388 S  1.7  0.2   0:00.94 php-cgi
 2782 www       15   0 85344 8252 6372 S  1.7  0.2   0:00.76 php-cgi
 2827 www       16   0 85352 8256 6380 S  1.7  0.2   0:00.56 php-cgi
29246 www       16   0 13084  10m  844 S  1.3  0.2  23:37.50 nginx
```

图 2 nginx+php 系统负载与 CPU 使用率

3) 成本低廉, 通过开源的 nginx 做反向代理实现负载均衡, 节省购买硬件负载均衡交换机的运维成本.

4) 支持热部署, 并且运行数月也无需重启, 在不间断服务的前提下对软件进行升级.

2 Linux的网络I/O模型

2.1 I/O 模型

Linux 环境下 I/O 模型包含以下几种[4]:

1) 阻塞式 I/O(blocking I/O 模型): 阻塞式 I/O 模型非常经典, 由于简单, 被广泛使用. 当 I/O 操作无法立即完成时, 进程从系统调用开始到返回的整段时间内都是阻塞的.

2) 非阻塞式 I/O(nonblocking I/O 模型): 与阻塞式 I/O 相对, 当文件描述符(file descriptor, fd)未就绪时, 进程不会被阻塞, 而是返回一个错误. 此时进程将持续调用内核进行查询.

3) I/O 复用模型 (I/O multiplexing): linux 提供 select/epoll 函数, 进程将若干描述符传递给 select 或 poll 系统调用, 这样 select/poll 可以帮我们侦测许多 fd 是否就绪. 当 I/O 操作就绪时, 应用能够被通知到. 借助 select 函数, 当操作未就绪时, 其阻塞在函数上, 而不是在真正的 I/O 系统调用上.

4) 信号驱动式 I/O(signal driven I/O 模型): 信号驱动模型是在文件描述符准备好时采取信号通知, 随后应用程序读取内核.

5) 异步 I/O(asynchronous I/O 模型): 异步 I/O 与信号驱动 I/O 比较相似, 但区别在于: 信号驱动式 I/O 由内核通知我们何时可以进行 I/O 操作, 而异步 I/O 模型是内核告知 I/O 操作何时完成.

针对高并发的网络环境: 阻塞式 I/O 将建立大量线程, 而非阻塞式 I/O 将不断遍历内核, 系统性能都将大受影响; 对于信号驱动 I/O, 信号无法传达事件发生在何种描述符, 通常很少用, 而异步 I/O, 至今很少有支持这一模型的系统; I/O 复用能在多个描述符上等待事件的发生, 当没有发生时, 进程进入阻塞, 正适合多并发的环境[5]. Apache 与 Nginx 的 I/O 事件驱动机制正是基于 I/O 复用得以实现.

2.2 Apache 的 I/O 事件驱动机制—select[4]

在 UNIX/Linux 系统编程中, 一个进程往往需要等待诸如可读, 可写或异常等多个描述符发生的某一事件, 但是进程不可能为了其中单独某个描述符而一直阻塞下去. select 函数就是实现这一情况的方法之一. 它允许进程指示内核等待多个描述符, 并只在有一个或多个发生变化或者经历指定的一段时间时被唤醒.

下面给出 select()的原型:

```
#include<sys/select.h>
#include<sys/time.h>
int select (int maxfdpl, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timeval *timeout);
```

返回: 整数为描述符就绪的个数, 超时为 0, 错误为-1. 关于文件描述符就是一个整数, 我们最熟悉的文件描述符是 0, 1, 2. 0 是标准输入, 1 是标准输出, 2 是标准错误.

Select()一共有五个参数, 中间的三个参数 readset, writeset, exceptset 本质相同. 用户用 readset 告诉内核需要监视的文件描述符集合, select()通过 readset 测试这些描述符是否可读. 同理, writeset 指定需要监视的可写文件描述符集合, exceptset 指定需要监视的异常文件描述符集合. 数据结构 fd_set 是一个文件描述符

集合，用一位来表示一个文件描述符，我们借助 POSIX 标准为我们提供的四个宏可以完成我们需要对 fd_set 做的所有操作. 这四个宏分别是:

void FD_ZERO(fd_set *fdset); /* 清空 fdset 中的所有文件描述符 */

void FD_SET(int fd, fd_set *fdset); /* 将一个文件描述符添加进 fdset */

void FD_CLR(int fd, fd_set *fdset); /* 将一个文件描述符清除出 fdset */

int FD_ISSET(int fd, fdset *fdset); /* 检查 fdset 中的文件描述符是否可读写 */

Select()的参数 maxfdpl 指定待测试的描述符个数，它的值等于待测最大描述符加一，因为描述符从 0 开始.

参数 timeout 指定超时的时长. timeval 结构体指定这段时长的秒数和微妙数:

```
struct timeval
{
    long tv_sec;//秒数
    long tv_usec;//微秒数
}
```

该参数设置有三种可能: (1)设置为空，函数一直等待仅有的一个文件描述符 I/O 就绪后才返回; (2)设置指定的秒数和微妙数，函数在一个文件描述符 I/O 就绪后返回，但等待的时间不超过指定的时间; (3)设置为 0，函数检查文件描述符后立即返回，这称为轮询(polling). 不同的参数设置，使 select()表现出无超时阻塞，超时结束和轮询三种特性.

2.3 Nginx 的 I/O 事件驱动机制—epoll

epoll 在 Linux2.6 内核中正式引入，它的设计以取代过时的 select. 下面详细说明 epoll 是如何实现的.

epoll 机制由 3 个系统调用函数完成:

int epoll_create(int size)创建一个专属于 epoll 的文件描述符，实质是在内核中建立一个文件系统 eventpoll，接下来的所有操作都将通过这个系统中进行. Size 大于 0 即可.

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)为 epoll 的事件注册函数，相对于 select 在监听事件时通知内核要监听的类型，这里先将要监听的事件类型进行注册. 它共有四个参数:

第一个参数 epfd 就是 epoll_create()的返回值.

第二个参数表示行为，借由如下三个宏完成对某个文件描述符的注册，修改，删除:

EPOLL_CTL_ADD

/* 注册新的文件描述符到 epfd 中 */;

EPOLL_CTL_MOD

/* 修改已经注册的文件描述符的监听事件 */;

EPOLL_CTL_DEL

/*从 epfd 中删除一个文件描述符 */;

第三个参数是指定需要监听的文件描述符.

第四个参数通知内核需要监听的事件，它的结构体[6]如下:

```
struct epoll_event {
    __uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
};
```

epoll_event 用来注册需要监听和回传所发生等待处理的事件，events 字段表示所监听和被触发的事件，它可以是以下宏的集合:

EPOLLIN ; /*表示对应的文件描述符可以读*/

EPOLLOUT; /*表示对应的文件描述符可以写*/

EPOLLPRI; /*表示对应的文件描述符有紧急的数据可读(这里应该表示有外带数据到来)*/

EPOLLERR; /*表示对应的文件描述符发生错误*/

EPOLLHUP; /*表示对应的文件描述符被挂断*/

EPOLLET; /*将 epoll 设为边缘触发[7](Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的*/

EPOLLONESHOT /*只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个 socket 的话，需要再次把这个 socket 加入到 EPOLL 队列里. 字段 data 是一个联合体:

```
typedef union epoll_data {
void *ptr;
int fd;
__uint32_t u32;
__uint64_t u64;
} epoll_data_t;
```

它用于保存事件的某个描述符的具体数据.

int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout)类似于 select 调用，它用来等待事件的产生.

3 Apache与Nginx的I/O机制对比

Apache 的 select 与 nginx 的 epoll 机制根本上的差别在于在实现二者功能时所用的数据结构的差异.

select 中涉及到的重要数据结构就是 fd_set, 它一般为一个整型数组, 用每个 bit 位来对应文件描述符. 由于每次在使用 select 前, 我们不知道 fd_set 中的元素是何种状态, 所以必须在使用前调用 void FD_ZERO(fd_set *fdset)将所有位清零, 即初始化. 每添加一个待测文件描述符就在其所需要关注状态的对应 fd_set 中的某一位置 1. 在调用 select 时, 先遍历所有的文件描述符, 返回就绪事件的个数, 然后调用 int FD_ISSET 对 fd_set 中的文件描述符逐个判别是否发生事件. 结束后将 fd_set 中未就绪的文件描述符所对应的位置 0. 当再一次调用 select 时, 由于需要把 fd_set 初始化, 所以必须将需要把文件描述符对应的位再一次置 1.

在实现 epoll 时, 它所申请的专用文件系统 eventpoll 与其基本单元 epitem 的结构至关重要. 得益于这个文件系统, epoll 突破了单个进程能打开文件描述符数量的限制, 而不像 select 中的 maxfdpl 不能超过宏定义 FD_SETSIZE 大小, 一般为 1024. eventpoll 以红黑树保存与 epoll 关联的文件描述符, 便于增删查改, 辅以 ready_link_list 链表, 用以存放就绪的 I/O 事件. 而 epitem 顾名思义为树的各个节点, 保存相关事件信息(即 epoll_event), 辅以 ready_list_link, 表示此事件就绪. 当事件就绪即状态改变时, 事先注册的回调函数 callback 被执行, epitem 中的 ready_list_link 被链接到 eventpoll 中 read_link_list 表中. epoll_wait 无需遍历所有的文件描述符, 只需收集 ready_link_list 中是否数据到来即可. 当再一次调用 epoll 时, 由于 ready_link_list 链表一直存在, 无需像 select 一样初始化, 只需再一次调用 epoll_wait 即可.

在 Linux 中当若干个活跃的用户连接需要获取 I/O 事件时: 若调用 select, 它需要遍历所有的文件描述符, 在轮询过程中检查监视的描述符是否就绪, 调用结束清空 fd_set 后, 重复地将信息拷贝到内核中; 若调用 epoll, 通过事先注册的回调函数, 无需遍历, 直接获取 read_link_list 链表即可. 与 select 算法的时间复杂度 O(n) 不同, epoll 通过把操作拆分为 epoll_create, epoll_ctl, epoll_wait 省去轮询的时间, 复杂度为 O(1)[8]. 图 3 清晰地显示出 epoll 相对于 select 在处理大量文件描述符时的优越性能.

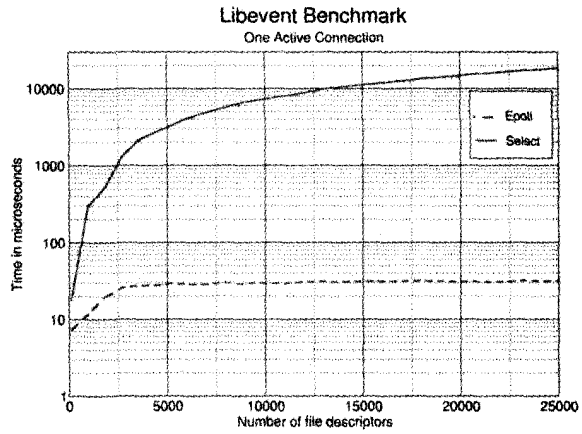

图 3 epoll 和 select 事件机制在 Libevent 的性能比较

与传统的 select 相比, epoll 机制的优势[9]在于: (1)支持进程打开大量文件描述符; (2)文件描述符数量不影响 I/O 效率; (3)使用 mmap 加速内核与用户空间的消息传递; (4)数据结构良好.

4 实测对比

大学校园内部网站, 带宽充裕, 网络稳定, 提升服务器性能的关键因素在于如何适当的结合服务器的软硬件条件, 实现最高的性价比. Apache 与 Nginx 的本地并发压力对比测试的系统配置为: CPU:Intel core i5 @2.40GHz, RAM:3G DDR3, 系统 :Ubuntu 11.04(linux 2.6.38), Php5.3.5.

使用网站压力测试工具 webbench 对存放在服务器根目录中显示 php 信息的 info.php 文件进行测试. 安装好 webbench 后在终端输入以下代码开始测试:

Webbench -c 50 -t 30 http://localhost/info.php

上述命令中, -c 表示客户端数, -t 表示时间.

测试时间固定为 30 秒, 客户端数量由 50 递增至 15000. 结果如表 1 所示.

表 1 压力测试对比

客户端	Nginx 处理请求数		Apache 处理请求数	
	成功	失败	成功	失败
50	40073	0	23243	0
100	39452	0	22002	0
250	39992	0	20648	0
500	41719	0	19639	5
1000	42088	0	20733	54
1500	39577	0	18823	179
2000	40756	0	21526	142

3000	43104	0	17902	437
4000	41714	0	13591	483
6000	41386	2	/	/
8000	44553	37	/	/
10000	41689	357	/	/
15000	40439	234	/	/

我们将表 1 中的数据以折线图表示, 如图 4 所示.

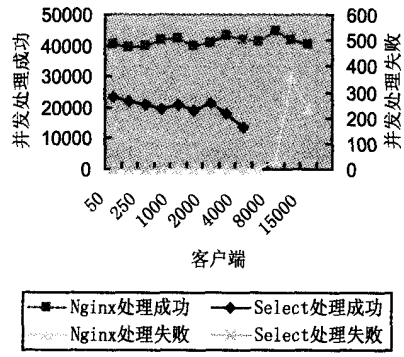

图 4 Apache 和 Nginx 的压对比力测试(纵轴为 30s 内成功处理的请求数)

由图 4 我们可以发现, Nginx 处理并发请求速度快且稳定, 测试时间内成功处理的请求数约为 Apache 的 2 倍. 当并发数到达 15000 时, 基于 Nginx 的网页仍能可以打开; 而 Apache 在并发数超过 6000 后导致内存不足系统卡死.

5 结论

综合以上对 Apache 与 Nginx 的对比分析以及实际测试, Apache 由于其出色的扩展性以及多功能性, 依然是许多大型网站的选择, 但在面对数千数量级的并发量时, 单台服务器显得力不从心, 只有升级硬件配置或者建立服务器集群来负载均衡应对这一问题. 而 Nginx 因其出色的处理高并发的性能以及轻便简洁经济的特性更容易被中小型网站接受. 在搭建网站时, 需要针对不同的网络使用环境以及硬件条件, 选择合适的服务程序, 以提升网站的性能.

参考文献

1 Nedelcu C. Nginx HTTP Server.Birmingham.Packt Publishing, 2010: 242–245.

2 Kegel D. The C10K problem.2011.http://www.kegel.com/c10k.html

3 张宴.实战 Nginx:取代 Apache 的高性能 Web 服务器.北京:电子工业出版社,2011.6–10.

4 Stevens WR, Fenner B, Rudoff AM. UNIX Network Programming Volume 1: The Socket Networking API. 3rd Ed. 北京:人民邮电出版社,2010.122–132.

5 李涛,房鼎益,陈晓江,冯健. Linux 系统中网络 I/O 性能改进方法的研究.计算机工程,2008,34(23):142–146.

6 吴敏,熊文龙.基于 Linux 的高性能服务器端的设计与研究.交通与计算机,2007,25(1):129–131.

7 Sivaraman M. How to use epoll: A complete example in C.2011.https://banu.com/blog/2/how-to-use-epoll-a-complete-example-in-c/.

8 Kovyrin O. Using epoll For Asynchronous Network Programming.2006.http://kovyrin.net/2006/04/13/epoll-asynchronous-network-programming/.

9 崔滨,万旺根,余小清,楼顺天.基于EPOLL机制的LINUX网络游戏服务器实现方法.微计算机信息,2006,22(21):64–66.