并发socket服务器设计

Andrew Huang <bluedrum@163.com>

课程目标

- 基于多路复用的并发服务器
  - select应用
  - 基于select的服务器
- 基于epoll的并发服务器

select 多路复用

1. 文件的阻塞函数

- 一般文件读取函数read,socket的接收函数recv,recvfrom都是阻塞型函数,即没有数据收到时,整个程序被阻塞这个函数
- 在服务器软件上,经常要接收多个客户端的数据.如果单纯采用recv造成整个程序的阻塞.
- 一种方法采用多线程.
- 更为常用是采用多路复用函数**select**来同时控制多个**socket/file** 描述符**.**

2. select多路复用

- select系统调用是用来让我们的程序监视多个文件句柄(file descrīptor)的状态变化的。程序会停在select这里等待，直到被监视的文件句柄有某一个或多个发生了状态改变。
  - 普通文件读写
  - socket的收发
  - 设备文件的收发

fd_set数据结构

- select主要操作fd_set的数据结构.fd_set是一个文件描述符的矢量数组
  - 大体上可以把fd_set看成一个只有1024项的整数数组.
  - 每一个socket或fd都是fd_set中的一项,
- 一般采用一组宏来操作fd_set
  - void FD_SET(int fd,fd_set *fdset)
  - void FD_CLR(int fd,fd_set *fdset)
    - FD_CLR将fd从fdset里面清除
  - void FD_ZERO(fd_set *fdset)
    - FD_ZERO从fdset中清除所有的文件描述符
  - int FD_ISSET(int fd,fd_set *fdset)
    - FD_ISSET判断fd是否在fdset集合中

```
typeset struct fd_set
  {
        u_int fd_count;
       int fd_array[fd_setsize];
  }
```

select定义

- int select(int max_fd, fd_set* readfds, fd_set* writefds, fd_set* exceptfds, struct timeval* timeout);
  - 其中max_fd为我们要监听的套接字中值最大的一个，同时在调用select是要将其加1，
  - readfd即为我们监听的要进行读操作的套接字连接集合，
  - 第三个参数是我们监听的要进行写操作的套接字连接集合，
  - 第四个参数用于异常，而最后一个参数可以用来设定超时，这里同样使用了struct timeval结构，
  - 当有文件被写时,返回一个大于0值,出错返回一个负数,等于表示在timeout的时间,没有任何读写,select是超时返回的

select()的使用

- select同时监控多个激活的socket(最大值一般为1024)
- 当相应的socket上有数据接收时,select将其值写入readfd值中.并返回一个大于0值.
- 这样通过FD_ISSET可以查出是哪一个socket被读写.因为只有一个阻塞点.大大提高程序的性能
- 因为带有超时机制,也能防止长时间阻塞导致程序无法响应的后果
- Select也能处理一般的文件或设备文件,如把标准输入或普通文件加入到监控的集合中
- **WinSock**的**select**版本只能监控**socket,**不能监控设备文件**,**如**0,1,2.**强行监控会造成阻塞失败

select实例

```
int main()     {
        int ret;
        fd_set fds;
        struct timeval tv;

        FD_ZERO(&fds);
        FD_SET(0,&fds);//把标准输入加入监控
         tv.tv_sec = 5;
        tv.tv_usec = 0;

         ret = select(1, &fds, NULL, NULL, &tv);
         if(ret < 0)
         { perror("select"); exit(-1)
         }else if(ret == 0)
```

```
/*接上一页*/
      {//5 秒钟内用户没有按下键
       printf(“timeout”);
      }
    else
       {  //读入用户输入
        scanf("%s", buf);
       }
   }
```

Select的socket下使用流程

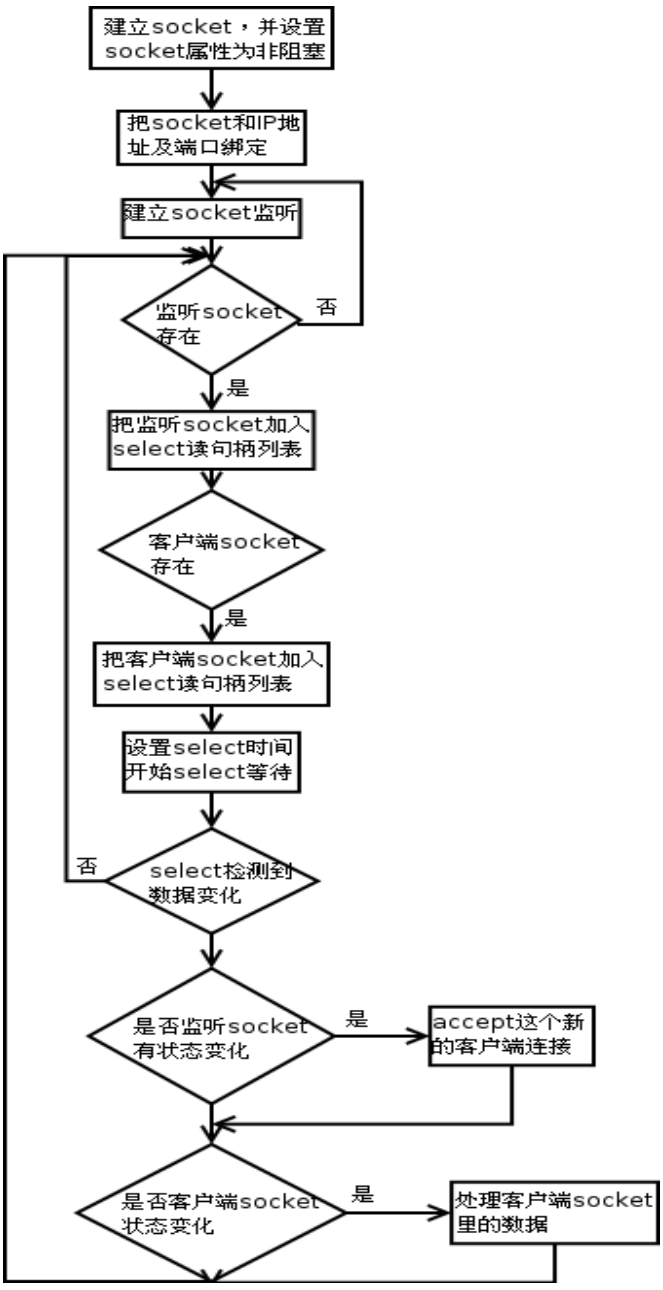
- Select 的使用是固定的流程

```
socket(...);
 bind(...);
 listen(...);

  while(1)
    {
       FD_ZERO(…)
       FD_SET(…)
       select(…);

        // 如果是服务器侦听套接字被触发,说明一个新的连接请求建立
       if(FD_ISSET(svr_fd,…))
       {
         //建立新的客户联接连接
         new_fd = accept(…) ;
        // 加入到监听文件描述符中去;
        FD_SET(new_fd,…)
       }
      else
       {
          //是一个客户端操作
          进行操作(read 或者 write);
        }

    }
```



epoll服务器

1. select的问题

- select　是常见的多路复用的机制,控制简单.可移植性好.但是有一些固有的缺点.
- Linux下最大并发数不超过1024.Windows为64
  - linux/posix_types.h定义
  - #define __FD_SETSIZE 1024

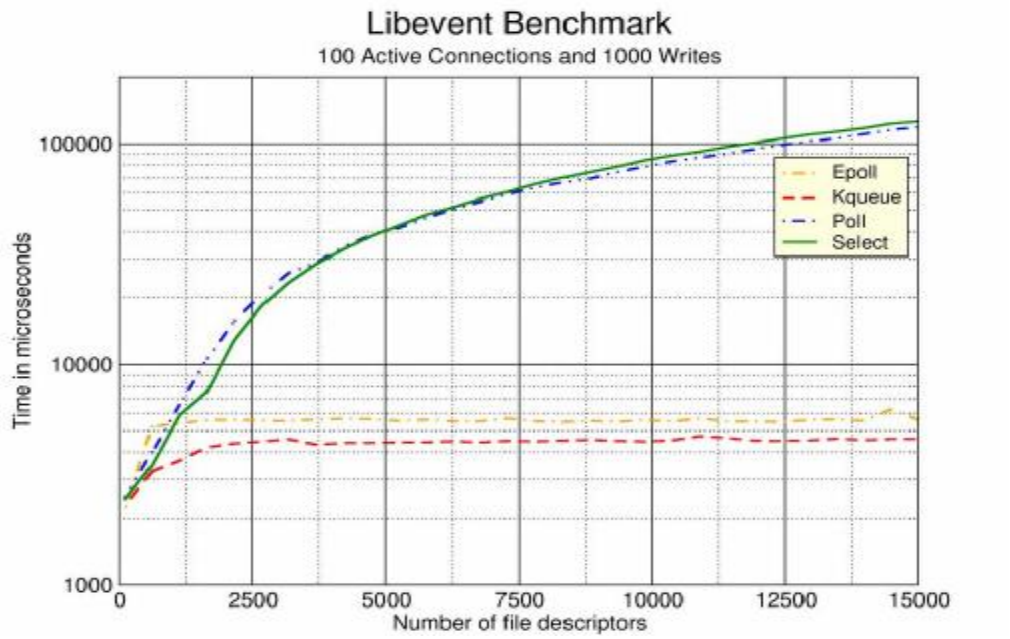
- 很多大型服务器并发数要求上万个.已经超过select极限

- select 采用轮询的方法来查询自己所监控的fd的状态,当需要监控数量过多时,轮询是一种比较低效的方法.
  - 特别是存在大量死链接的情况下,即连在服务器上但很少有数据收发.会极大影响select的效率

对select的改进

- 网络服务变得越来越复杂,很多操作系统都增加一些机制用于大并发数的IO复用.以取代select.
- Linux改进方案是epoll
- Windows的改进方案是IOCP(完全端口)
- FreeBSD的改进方案是kqueue

不同机制测试结果



2. Epoll的改进

- Epoll扩展了struct file结构,增加了IO事件回调函数指针
  - **rwlock_t f_cblock;struct list_head f_cblist;**
  - 当IO事情触发时.直接调用回调来处理,这是一个异步非阻塞操作
  - 而select必须阻塞才能工作.
- Epoll是处理内核机制,只是传输文件描述符到用户空间,而不是把文件数据拷贝到用户空间.
  - 这一些数据将放在一些共享内存当中,这大大减少很多I/O操作

EPOLL 要在 Linux 2.6 下编译执行

3. epoll使用

- 在操作系统下必须有/dev/epoll设备结点,如果没有,可以手工用mknod创建

- **mknod /dev/epoll c 10 124**

- 编程使用头文件
  - #include <sys/epoll.h>
- 操作epoll基本数据也是int句柄,共有三个操作函数
  - 1. **int epoll_create(int size);** #创建一个epoll的句柄
  - 2. **int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)**;#epoll的事件注册函数
  - 3. **int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);** #等待事件的产生，类似于select()调用,
- 关闭epoll句柄使用close()

1)epoll_create:创建

- **int epoll_create(int size);**
- 创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大。这个参数不同于select()中的第一个参数，给出最大监听的fd+1的值。需要注意的是，当创建好epoll句柄后，它就是会占用一个fd值，在linux下如果查看/proc/进程id/fd/，是能够看到这个fd的，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽

2)**epoll_ctl:事件处理**

- **int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);**
- epoll的事件注册函数，它不同与select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。
  - 第一个参数是epoll_create()的返回值，
  - 第二个参数表示动作，用三个宏来表示：
    - EPOLL_CTL_ADD：注册新的fd到epfd中；
    - EPOLL_CTL_MOD：修改已经注册的fd的监听事件；
    - EPOLL_CTL_DEL：从epfd中删除一个fd；
  - 第三个参数是需要监听的fd，
  - 第四个参数是告诉内核需要监听什么事，使用struct epoll_event结构
    events可以是以下几个宏的集合：
    - EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；
    - 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（这里应该表示有带外数据到来）；
    - 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(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

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

3)epoll_wait:等待事件发生

- 3. int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
- 等待事件的产生，类似于select()调用。参数events用来从内核得到事件的集合，maxevents告之内核这个events有多大，这个maxevents的值不能大于创建epoll_create()时的size，参数timeout是超时时间（毫秒，0会立即返回，-1将不确定，也有说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回0表示已超时。

epoll实例

```
kdpfd = epoll_create(MAXEPOLLSIZE);
len = sizeof(struct sockaddr_in);
ev.events = EPOLLIN | EPOLLET;
ev.data.fd = listener; //这里的 listener 是指 TCP 服务器侦听 socket
//增加 epoll 监听事件
if (epoll_ctl(kdpfd, EPOLL_CTL_ADD, listener, &ev) < 0) {
    fprintf(stderr, "epoll set insertion error: fd=%d\n", listener);
    return -1;
}
curfds = 1;
while (1) {
    /* 等待有事件发生 */
    nfds = epoll_wait(kdpfd, events, curfds, -1);
    if (nfds == -1) {
        perror("epoll_wait");
        break;
    }
    /* 处理所有事件 */
    for (n = 0; n < nfds; ++n) {
        if (events[n].data.fd == listener) {
            new_fd = accept(listener, (struct sockaddr *) &their_addr,
                            &len);
            if (new_fd < 0) {
                perror("accept");
                continue;
            } else
                //有客户端加入,将其 socket 新增到监听的 fd 当中
            setnonblocking(new_fd);
            ev.events = EPOLLIN | EPOLLET;
            ev.data.fd = new_fd;
            if (epoll_ctl(kdpfd, EPOLL_CTL_ADD, new_fd, &ev) < 0) {
                fprintf(stderr, "把 socket '%d' 加入 epoll 失败！ %s\n",
                        new_fd, strerror(errno));
                return -1;
              }
```

```
/*接上一页*/

            curfds++;
        }
    else {
            ret = handle_message(events[n].data.fd);
            if (ret < 1 && errno != 11) {
                epoll_ctl(kdpfd, EPOLL_CTL_DEL, events[n].data.fd, &ev);
                curfds--;
            }
        }
    }
}
close(listener);
```

课堂练习

- 用getpeername()把聊天程序的广播消息加入IP和端口信息.
- 请将原有简单文件下载服务器,由一次下载一个文件,升级为一次可下多个文件的.
  - 多个文件名可由配置文件或直接写在源码里
- 请将广播和聊天服务器代码移植到Linux上