# I/O复用

下述情况需要使用I/O复用技术：

1. 客户端需要同时处理多个socket
2. 客户端需要同时处理用户输入和网络连接
3. 服务器需要同时处理监听socket和连接socket
4. 服务器需要同时处理TCP请求和UDP请求
5. 服务器需要同时监听多个端口

注：这里的I/O指的是网络套接字socket的I/O。I/O复用虽然能够监听多个端口，但是其本身是阻塞的，如果需要提高效率需要采用多进程多线程技术。

总结：



# select()

## 原型

#include <sys/select.h>

int select(int nfds,fd\_set\* readfds,fd\_set\* writefds,fd\_set\*exceptfds,struct timeval\* timeout);

参数：

nfds—被监听文件描述符的总数，通常设置为select监听所有文件描述符最大值+1（一般从3开始，0/1/2系统输入/输出/错误）

readfds, writefds, exceptfds—可读/可写/异常文件描述符集合，在调用select的时候，通过这3个参数传入自己感兴趣的文件描述符

timeout—设置select函数的超时时间（select调用失败时返回timeout不确定）

可以访问fd\_set结构体的函数：

FD\_ZERO(fd\_set\* fdset); // 清除fdset的所有位

FD\_SET(int fd,fd\_set\* fdset); // 设置fdset的位fd

FD\_CLR(int fd,fd\_set\* fdset); // 清除fdset的位fd

int FD\_ISSET(int fd,fd\_set\* fdset); // 测试fdset的位fd是否被设置

struct timeval结构体：

struct timeval{

long tv\_sec;//秒数

long tv\_usec;//微秒数

};

如果将两个参数都设置为0，则select立即返回；如果设置为NULL，则select会阻塞。

socket可读的条件：

1. socket接收缓冲区中字节大于等于低水位标记
2. socket对方关闭连接（应该返回0）
3. 监听socket上有新的连接
4. socket上有待处理的错误

socket可写的条件：

1. socket中发送缓冲区字节大于低水位
2. socket对方写操作关闭
3. socket上使用非阻塞connect连接成功或失败之后
4. socket上有未处理的错误

## 特点

select()调用的缺点：

1. 单个进程可监视的fd数量被限制（fd\_set仅包含一个整型数组，这个整型数组的每一位标记一个文件描述符fd\_set能容纳的文件描述符数量是由一个宏决定的）；
2. 对socket采用轮询的方法；
3. 需要维护一个用来存放大量fd的数据结构。

## 实例

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <assert.h>

#include <stdio.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <fcntl.h>

#include <stdlib.h>

int main(int argc,char\* argv[])

{

if(argc <= 2)

{

printf("please input ip address and port num\r\n");

exit(1);

}

const char\* ip = argv[1];

int port = atoi(argv[2]);

int ret = 0;

struct sockaddr\_in address;

bzero(&address,sizeof(address));

address.sin\_family = AF\_INET;

inet\_pton(AF\_INET,ip,&address.sin\_addr);

address.sin\_port = htons(port);

int listenfd = socket(AF\_INET,SOCK\_STREAM,0);

assert(listenfd >= 0);

ret = bind(listenfd,(struct sockaddr\*)&address,sizeof(address));

assert(ret != -1);

ret = listen(listenfd,5);

assert(ret != -1);

struct sockaddr\_in client\_address;

socklen\_t client\_addresslen = sizeof(client\_address);

int connfd = accept(listenfd,(struct sockaddr\*)&client\_address,

&client\_addresslen);

if(connfd < 0)

{

printf("accept error\r\n");

exit(1);

}

char buf[1024];

fd\_set read\_fds;

fd\_set exceptions\_fds;

FD\_ZERO(&read\_fds);

FD\_ZERO(&exceptions\_fds);

while(1)

{

memset(buf,0x00,sizeof(buf));

FD\_SET(connfd,&read\_fds);

FD\_SET(connfd,&exceptions\_fds);

ret = select(connfd+1,&read\_fds,NULL,&exceptions\_fds,NULL);

if(ret < 0)

{

printf("select error\r\n");

exit(1);

}

if(FD\_ISSET(connfd,&read\_fds))

{

ret = recv(connfd,buf,sizeof(buf)-1,0);

if(ret <= 0)

{

break;

}

printf("recv data form %d and buf is %s\r\n",connfd,buf);

}

else if(FD\_ISSET(connfd,&exceptions\_fds))

{

ret = recv(connfd,buf,sizeof(buf)-1,MSG\_OOB);

if(ret <=0 )

{

break;

}

printf("recv oob data fomr %d and buf is %s\r\n",connfd,buf);

}

}

close(connfd);

close(listenfd);

return 0;

}

# poll()

## 原型

#include <poll.h>

int poll(struct pollfd\* fds,nfds\_t nfds,int timeout);

struct pollfd结构体：

struct pollfd{

int fd; //文件描述符

short events; //注册的事件

short revents;//实际发生的事件，由内核填充

};

poll事件类型：

示例：

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <assert.h>

#include <stdio.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <fcntl.h>

#include <stdlib.h>

int main(int argc,char\* argv[])

{

if(argc <= 2)

{

printf("please input ip address and port num\r\n");

exit(1);

}

const char\* ip = argv[1];

int port = atoi(argv[2]);

int ret = 0;

struct sockaddr\_in address;

bzero(&address,sizeof(address));

address.sin\_family = AF\_INET;

inet\_pton(AF\_INET,ip,&address.sin\_addr);

address.sin\_port = htons(port);

int listenfd = socket(AF\_INET,SOCK\_STREAM,0);

assert(listenfd >= 0);

ret = bind(listenfd,(struct sockaddr\*)&address,sizeof(address));

assert(ret != -1);

ret = listen(listenfd,5);

assert(ret != -1);

struct sockaddr\_in client\_address;

socklen\_t client\_addresslen = sizeof(client\_address);

int connfd = accept(listenfd,(struct sockaddr\*)&client\_address,

&client\_addresslen);

if(connfd < 0)

{

printf("accept error\r\n");

exit(1);

}

char buf[1024];

fd\_set read\_fds;

fd\_set exceptions\_fds;

FD\_ZERO(&read\_fds);

FD\_ZERO(&exceptions\_fds);

while(1)

{

memset(buf,0x00,sizeof(buf));

FD\_SET(connfd,&read\_fds);

FD\_SET(connfd,&exceptions\_fds);

ret = select(connfd+1,&read\_fds,NULL,&exceptions\_fds,NULL);

if(ret < 0)

{

printf("select error\r\n");

exit(1);

}

if(FD\_ISSET(connfd,&read\_fds))

{

ret = recv(connfd,buf,sizeof(buf)-1,0);

if(ret <= 0)

{

break;

}

printf("recv data form %d and buf is %s\r\n",connfd,buf);

}

else if(FD\_ISSET(connfd,&exceptions\_fds))

{

ret = recv(connfd,buf,sizeof(buf)-1,MSG\_OOB);

if(ret <=0 )

{

break;

}

printf("recv oob data fomr %d and buf is %s\r\n",connfd,buf);

}

}

close(connfd);

close(listenfd);

return 0;

}

## 特点

# epoll()

## 原型

### epoll\_create

#include<sys/epoll.h

int epoll\_create(int size);

### epoll\_ctl

#include<sys/epoll.h

int epoll\_ctl(int epfd,int op,int fd,struct epoll\_event\* event);

op参数的类型：



注：epoll是个事件表，这是放置在内核中的，我们只需要调用系统函数往这个事件表上注册、修改、删除关注的套接字描述符即可，不像select和poll系统调用需要往集合上注册，而且注册到集合上的文件描述符需要在内核态和用户态来回切换。

struct epoll\_event结构体：

struct epoll\_event

{

\_\_uint32\_t events; //epoll事件表（事件类型）

epoll\_data\_t data; //用户数据

};

### epoll\_wait

**原型：**

#include<sys/epoll.h

int epoll\_wait(int epfd,struct epoll\_event\* events,int maxevents,int

timeout);

**作用：**在一段超时时间内，等待一组文件描述符上的事件

**返回值：**成功返回事件数，失败-1

**说明：**epoll如果监测到事件，就把该事件从内核的事件表中复制到第二个参数event指向的数组中（这里只拷贝就绪事件，而select/poll拷贝的是所有的事件）。

## 工作模式



注：LT模式编程简单，效率低，ET模式工作效率高。

## 实现机制



注：epoll之所以那么高效，因为在内核中epoll是通过链表和红黑树来实现的，红黑树的作用就是事件表，链表就是返回的就绪事件的文件描述符。

## 实例

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <assert.h>

#include <stdio.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <fcntl.h>

#include <stdlib.h>

#include <sys/epoll.h>

#include <pthread.h>

#define MAX\_SOCKET\_NUMBERS 1024

#define BUF\_SIZE 1024

int setnonblocking(int fd);

void addfd(int epollfd,int fd, int flag);

void lt(struct epoll\_event\* events,int number,int epollfd,int listenfd);

void et(struct epoll\_event\* events,int number,int epollfd,int listenfd);

int main(int argc,char\* argv[])

{

if(argc <= 2)

{

printf("please input ip address and port num\r\n");

exit(1);

}

const char\* ip = argv[1];

int port = atoi(argv[2]);

int ret = 0;

struct sockaddr\_in address;

bzero(&address,sizeof(address));

address.sin\_family = AF\_INET;

inet\_pton(AF\_INET,ip,&address.sin\_addr);

address.sin\_port = htons(port);

int listenfd = socket(AF\_INET,SOCK\_STREAM,0);

assert(listenfd >= 0);

ret = bind(listenfd,(struct sockaddr\*)&address,sizeof(address));

assert(ret != -1);

ret = listen(listenfd,5);

assert(ret != -1);

struct epoll\_event events[MAX\_SOCKET\_NUMBERS];

int epollfd = epoll\_create(5);

assert( epollfd != -1);

addfd(epollfd,listenfd,1);

while(1)

{

int ret = epoll\_wait(epollfd,events,MAX\_SOCKET\_NUMBERS,-1);

if(ret < 0)

{

printf("epoll\_wait error\r\n");

exit(1);

}

lt(events,ret,epollfd,listenfd);

// et(events,ret,epollfd,listenfd);

}

close(listenfd);

return 0;

}

int setnonblocking(int fd)

{

int old\_option = fcntl(fd,F\_GETFL);

int new\_option = old\_option | O\_NONBLOCK;

fcntl(fd,F\_SETFL,new\_option);

return new\_option;

}

void addfd(int epollfd,int fd,int flag)

{

struct epoll\_event event;

memset(&event,0x00,sizeof(event));

event.data.fd = fd;

event.events = EPOLLIN;

if(flag )

{

event.events |= EPOLLET;

}

epoll\_ctl(epollfd,EPOLL\_CTL\_ADD,fd,&event);

setnonblocking(fd);

}

void lt(struct epoll\_event\* events,int number ,int epollfd,int listenfd)

{

char buf[BUF\_SIZE];

int i;

for(i=0;i<number;i++)

{

int sockfd = events[i].data.fd;

if(sockfd == listenfd)

{

struct sockaddr\_in client\_address;

socklen\_t client\_addresslen = sizeof(client\_address);

int connfd = accept(listenfd,(struct sockaddr\*)&client\_address,

&client\_addresslen);

if(connfd < 0)

{

printf("accept error\r\n");

exit(1);

}

addfd(epollfd,connfd,0);

}

else if(events[i].events & EPOLLIN)

{

printf("LT once\r\n");

memset(buf,0x00,sizeof(buf));

int ret = recv(sockfd,buf,sizeof(buf)-1,0);

if(ret <=0)

{

printf("recv 0\r\n");

close(sockfd);

continue;

}

printf("recv data form %d buf is %s\r\n",sockfd,buf);

}

else

{

printf("somthing else happen\r\n");

}

}

}

void et(struct epoll\_event\* events,int number,int epollfd,int listenfd)

{

char buf[BUF\_SIZE];

int i;

for(i=0;i<number;i++)

{

int sockfd = events[i].data.fd;

if(sockfd == listenfd )

{

struct sockaddr\_in client\_address;

socklen\_t client\_addresslen = sizeof(client\_address);

int connfd = accept(listenfd,(struct sockaddr\*)&client\_address,

&client\_addresslen);

addfd(epollfd,connfd,0);

}

else if(events[i].events & EPOLLIN)

{

printf("ET once\r\n");

while(1)

{

memset(buf,0x00,sizeof(buf));

int ret = recv(sockfd,buf,sizeof(buf)-1,0);

if(ret < 0)

{

if((errno == EAGAIN ) || (errno == EWOULDBLOCK))

{

printf("read later\r\n");

break;

}

close(sockfd);

break;

}

else if(ret == 0)

{

close(sockfd);

}

else

{

printf("recv data from %d buf is %s\r\n",sockfd,buf);

}

}

}

else

{

printf("somthin else happen\r\n");

}

}

}

# 对比

