# 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中字节大于低水位
2. socket对方关闭连接
3. 监听socket上有新的连接
4. socket上有待处理的错误

## 特点

select()调用的缺点：

1. 单个进程可监视的fd数量被限制；
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。

# 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：

/\*

\* =====================================================================================

\*

\* Filename: recv\_oob.c

\*

\* Description:

\*

\* Version: 1.0

\* Created: 06/18/15 06:45:10

\* Revision: none

\* Compiler: gcc

\*

\* Author: wang@xd (),

\* Organization:

\*

\* =====================================================================================

\*/

#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()