select 函数

看书

select函数详解

select函数使用

1. fcntl()函数

`fcntl`（File Control）函数是一个 POSIX 标准定义的系统调用，用于对文件描述符进行各种控制操作。这个函数通常用于实现对文件描述符的非阻塞设置、文件锁、获取或修改文件描述符的属性等操作。在网络编程中，它常常用于设置套接字为非阻塞模式。

以下是 `fcntl` 函数的基本原型：

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */ );
```

- **fd：** 需要进行操作的文件描述符。
- **cmd：** 需要执行的操作命令，可以是 `F_GETFL`（获取文件状态标志）、`F_SETFL`（设置文件状态标志）等。
- **arg：** 根据不同的命令可能需要的参数。

常见的 fcntl 命令：

```
F_DUPFD:
    复制文件描述符，返回一个新的文件描述符。

F_GETFD:
    获取文件描述符标志。

F_SETFD:
    设置文件描述符标志。

F_GETFL:
    获取文件状态标志。

F_SETFL:
    设置文件状态标志。

F_GETLK:
    获取记录锁的信息。

F_SETLK:
    设置或释放记录锁。

F_SETLKW:
    阻塞地设置或释放记录锁。
```

示例用法：

获取和设置文件状态标志：

```
// 获取文件状态标志
int flags = fcntl(fd, F_GETFL);

// 设置文件状态标志为非阻塞模式
flags |= O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
```

当 `socket` 函数设置为非阻塞时，`recv()` 和 `send()` 函数的行为与阻塞模式下会有所不同。在阻塞模式下，当调用 `recv()` 或 `send()` 函数时，如果没有数据可用或者无法立即发送数据，函数会一直等待直到数据可用或者发送缓冲区有足够的空间。

而在非阻塞模式下，`recv()` 和 `send()` 函数会立即返回，无论是否有数据可用或者发送缓冲区是否有足够的空间。如果没有数据可用或者发送缓冲区已满，`recv()` 函数会返回一个错误码，通常是 `EWOULDBLOCK` 或 `EAGAIN`，而 `send()` 函数也会返回一个错误码，通常是 `EWOULDBLOCK` 或 `EAGAIN`。这样，程序可以立即处理其他任务而不必等待数据的到来或者发送的完成。

在非阻塞模式下，通常需要使用循环来不断地尝试接收或发送数据，直到操作成功或者出现错误。这种方式称为轮询，它允许程序在等待数据时继续执行其他任务，提高了程序的效率和响应性。

2.设置文件描述符

首先介绍一个重要的结构体：`fd_set`，它会作为下面某些函数的参数而多次用到，fd_set可以理解为一个集合，这个集合中存放的是文件描述符(file descriptor)，即文件句柄，它用一位来表示一个fd（下面会仔细介绍）。fd_set集合可以通过下面的宏来进行人为来操作

`0`代表标准输入文件描述符（File Descriptor），它指向程序的标准输入流。在Unix/Linux系统中，标准输入流通常被分配给文件描述符`0`。

1》FD_ZERO

用法：`FD_ZERO(fd_set*);`

用来清空fd_set集合，即让fd_set集合不再包含任何文件句柄。在对文件描述符集合进行设置前，必须对其进行初始化，如果不清空，由于在系统分配内存空间后，通常并不作清空处理，所以结果是不可知的。

2》FD_SET

用法：`FD_SET(int ,fd_set *);`

用来将一个给定的文件描述符加入集合之中。

3》FD_CLR

用法：`FD_CLR(int ,fd_set*);`

用来将一个给定的文件描述符从集合中删除。

4》FD_ISSET

用法：`FD_ISSET(int ,fd_set*);`

检测fd在fdset集合中的状态是否变化，当检测到fd状态发生变化时返回真，否则，返回假（也可以认为集合中指定的文件描述符是否可以读写）。

3.select()函数

```
#include <sys/time.h>
#include <unistd.h>
int select(int maxfd,fd_set *rdset,fd_set *wrset,fd_set            *exset,struct
timeval *timeout);
```

- 参数maxfd是需要监视的最大的文件描述符值+1；
- rdset,wrset,exset分别对应于需要检测的可读文件描述符的集合，可写文件描述符的集 合及异常文件描述符的集合。
- readfds是 **等待读事件的文件描述符集合**，.如果不关心读事件（缓冲区有数据)，则可以传NULL值。
- 与readfds类似，writefds是等待写事件(缓冲区中是否有空间)的集合，如果不关心写事件，则可以传值NULL。
- 如果**内核等待相应的文件描述符发生异常**，则**将失败的文件描述符设置进exceptfds中**，如果不关心错误事件，可以传值NULL。
- timeout设置select在内核中阻塞的时间，如果想要设置为非阻塞，则设置为NULL
    - 如果没有文件描述符就绪就返回0；
    - 如果调用失败返回-1；
    - 如果timeout中中readfds中有事件发生，则返回timeout剩下的时间。

```
struct timeval {
      long    tv_sec;         /* seconds */
      long    tv_usec;        /* microseconds */
               };
```

理解select模型的关键在于理解fd_set,为说明方便，取fd_set长度为1字节，fd_set中的每一bit可以对应一个文件描述符fd。则1字节长的fd_set最大可以对应8个fd。

（1）执行fd_set set; FD_ZERO(&set);则set用位表示是0000,0000。
（2）若fd＝5,执行FD_SET(fd,&set);后set变为0001,0000(第5位置为1)
（3）若再加入fd＝2，fd=1,则set变为0001,0011
（4）执行select(6,&set,0,0,0)阻塞等待
（5）若fd=1,fd=2上都发生可读事件，则select返回，此时set变为0000,0011。注意：没有事件发生的fd=5被清空。

注意select函数执行后会清空无事件发生的fd_set中的fd

select模型的特点：

（1)可监控的文件描述符个数取决与sizeof(fd_set)的值。我这边服务　器上sizeof(fd_set)＝512，每bit表示一个文件描述符，则我服务器上支持的最大文件描述符是512*8=4096。据说可调，另有说虽　然可调，但调整上限受于编译内核时的变量值。

（2）将fd加入select监控集的同时，还要再使用一个数据结构array保存放到select监控集中的fd，一是用于再select返回后，array作为源数据和fd_set进行FD_ISSET判断。二是select返回后会把以前加入的但并无事件发生的fd清空，则每次开始select前都要重新从array取得fd逐一加入（FD_ZERO最先），扫描array的同时取得fd最大值maxfd，用于select的第一个　参数。

（3）可见select模型必须在select前循环array（加fd，取maxfd），select返回后循环array（FD_ISSET判断是否有时间发生）。

使用select函数的过程一般是：

先调用宏FD_ZERO将指定的fd_set清零，然后调用宏FD_SET将需要测试的fd加入fd_set，接着调用函数select测试fd_set中的所有fd，最后用宏FD_ISSET检查某个fd在函数select调用后，相应位是否仍然为1。

以下是一个测试单个文件描述字可读性的例子：

```
int isready(int fd)
{
    int rc;
    fd_set fds;
    struct tim tv;
    FD_ZERO(&fds);
    FD_SET(fd,&fds)
    tv.tv_sec = tv.tv_usec = 0;
    rc = select(fd+1, &fds, NULL, NULL, &tv);
    if (rc < 0)   //error
    return -1;
    return FD_ISSET(fd,&fds) ? 1 : 0;
}
```

select 函数调用示例

下面通过示例把select函数所有知识点进行整合，希望各位通过如下示例完全理解之前的内容。

❖ select.c

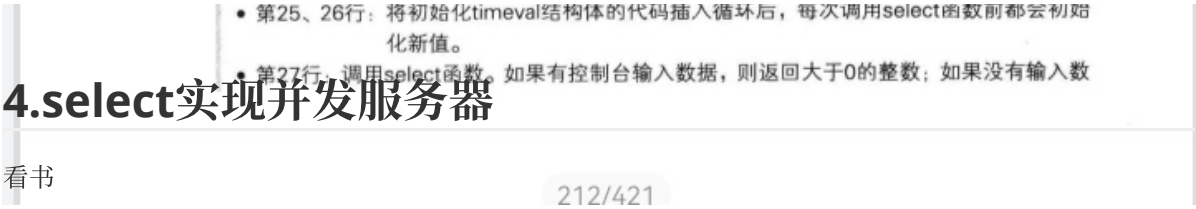
```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <sys/time.h>
4.  #include <sys/select.h>
5.  #define BUF_SIZE 30
6.
7.  int main(int argc, char *argv[])
8.  {
9.      fd_set reads, temps;
10.     int result, str_len;
11.     char buf[BUF_SIZE];
12.     struct timeval timeout;
13.
```



```
14.     FD_ZERO(&reads);
15.     FD_SET(0, &reads); // 0 is standard input(console)
16.
17.     /*
18.     timeout.tv_sec=5;
19.     timeout.tv_usec=5000;
20.     */
21.
22.     while(1)
23.     {
24.         temps=reads;
25.         timeout.tv_sec=5;
26.         timeout.tv_usec=0;
27.         result=select(1, &temps, 0, 0, &timeout);
28.         if(result==-1)
29.         {
30.             puts("select() error!");
31.             break;
32.         }
33.         else if(result==0)
34.         {
35.             puts("Time-out!");
36.         }
37.         else
38.         {
39.             if(FD_ISSET(0, &temps))
40.             {
41.                 str_len=read(0, buf, BUF_SIZE);
42.                 buf[str_len]=0;
43.                 printf("message from console: %s", buf);
44.             }
45.         }
46.     }
47.     return 0;
48. }
```

代码说明

- 第14、15行：看似复杂，实则简单。首先在第14行初始化fd_set变量，第15行将文件描述符0对应的位设置为1。换言之，需要监视标准输入的变化。
- 第24行：将准备好的fd_set变量reads的内容复制到temps变量，因为之前讲过，调用select函数后，除发生变化的文件描述符对应位外，剩下的所有位将初始化为0。因此，为了记住初始值，必须经过这种复制过程。这是使用select函数的通用方法，希望各位牢记。
- 第18、19行：请观察被注释的代码，这是为了设置select函数的超时而添加的。但不能在此时设置超时。因为调用select函数后，结构体timeval的成员tv_sec和tv_usec的值将被替换为超时前剩余时间。因此，调用select函数前，每次都需要初始化timeval结构体变量。



4.select实现并发服务器

看书

```
//server.cpp
#include <iostream>
#include <netinet/in.h>
#include <cstring>
#include <string>
#include <cstdlib>
#include <unistd.h>

#define PORT 8111
#define BUFFER_SIZE 128

void error_handling(const std::string& buf) {
    std::cerr << buf << std::endl;
    exit(1);
}

int main() {
    struct sockaddr_in serv_addr{}, accept_addr{};
    int serv_socket, accept_socket;
    int def, fd_max;
    int ret, recv_a;
    int backlog = 128;
    char buf[BUFFER_SIZE];
    fd_set reads, reads_copy;
    ssize_t N = BUFFER_SIZE;

    serv_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (serv_socket == -1) {
        error_handling("Socket creation error");
    }

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    FD_ZERO(&reads);
    FD_SET(serv_socket, &reads);
    fd_max = serv_socket;
    struct timeval timeout{};

    if (bind(serv_socket, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) == -1)
{
        error_handling("Bind error");
    }

    if (listen(serv_socket, backlog) == -1) {
        error_handling("Listen error");
    }
```

```
    while (true) {
        timeout.tv_sec = 5;
        timeout.tv_usec = 5000;
        reads_copy = reads;
        ret = select(fd_max + 1, &reads_copy, nullptr, nullptr, &timeout);

        if (ret == -1) {
            error_handling("Select error");
        }
        if (ret == 0) {
            continue;
        }
        for (int i = 0; i <= fd_max; ++i) {
            if (FD_ISSET(i, &reads_copy)) {
                if (i == serv_socket) {
                    socklen_t serv_len = sizeof(accept_addr);
                    accept_socket = accept(serv_socket, (struct
sockaddr*)&accept_addr, &serv_len);
                    if (accept_socket < 0) {
                        error_handling("Accept error");
                    }
                    FD_SET(accept_socket, &reads);
                    if (fd_max < accept_socket) {
                        fd_max = accept_socket;
                    }
                    std::cout << "连接客户端: " << accept_socket << std::endl;
                } else {
                    recv_a = read(i, buf, N);
                    if (recv_a < 0) {
                        error_handling("Recv error");
                    } else if (recv_a == 0) {
                        FD_CLR(i, &reads);
                        close(i);
                        std::cout << "关闭客户端: " << i << std::endl;
                    } else {
                        if (send(i, buf, recv_a, 0) < 0) {
                            error_handling("Send error");
                        }
                    }
                }
            }
        }
    }
    close(serv_socket);
    return 0;
}
```

```
// client.cpp
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```c
#include <arpa/inet.h>
#include <netinet/in.h>
#include <string.h>

#define BUFFER_SIZE 128

int main(int argc, char const *argv[]) {
    if (argc < 3) {
        fprintf(stderr, "Usage: %s [ip] [port]\n", argv[0]);
        exit(1);
    }

    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        perror("Socket creation failed");
        exit(1);
    }

    struct sockaddr_in serveraddr;
    socklen_t addrlen = sizeof(serveraddr);

    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = inet_addr(argv[1]);
    serveraddr.sin_port = htons(atoi(argv[2]));

    if (connect(sockfd, (struct sockaddr*)&serveraddr, addrlen) == -1) {
        perror("Connection failed");
        close(sockfd);
        exit(1);
    }

    char buf[BUFFER_SIZE];
    fgets(buf, BUFFER_SIZE, stdin);
    buf[strlen(buf) - 1] = '\0';  // 移除换行符

    if (send(sockfd, buf, strlen(buf), 0) == -1) {
        perror("Send failed");
        close(sockfd);
        exit(1);
    }

    char text[BUFFER_SIZE];
    if (recv(sockfd, text, BUFFER_SIZE, 0) == -1) {
        perror("Receive failed");
        close(sockfd);
        exit(1);
    }

    printf("来自服务器: %s\n", text);

    close(sockfd);
    return 0;
}
```

错误代码

```
#include <iostream>
#include <netinet/in.h>
#include<cstring>
#include<string>
#include <armadillo>

#define PORT 8111
void error_handling(std::string buf){
    std::cout<<buf<<std::endl;
    exit(1);
}

int main() {
    struct sockaddr_in serv_addr{},accept_addr{};
    int serv_socket, accept_socket = 0;
    int def,fd_max;
    int ret,i,recv_a;
    int backlog=128;
    socklen_t  serv_len;
    char  buf[]={};
    fd_set reads,reads_copy;
    ssize_t N=128;

    serv_socket=socket(AF_INET,SOCK_STREAM,0);
    memset(&serv_addr,0,sizeof(serv_addr));
    serv_addr.sin_family=AF_INET;
    serv_addr.sin_port=htons(PORT);
    serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
    FD_ZERO(&reads);
    FD_SET(serv_socket,&reads);
    fd_max=serv_socket;
    struct timeval timeout{};

    if(bind(serv_socket,(struct sockaddr*)&serv_addr,sizeof(serv_addr))==-1){
        error_handling("bind error");
    }

    if(listen(serv_socket,backlog)==-1){
        error_handling("listen error");
    }

    while(1) {
        timeout.tv_sec = 5;
        timeout.tv_usec = 5000;
        reads_copy = reads;
        ret = select(fd_max + 1, &reads_copy, nullptr, nullptr, &timeout);

        if (ret == -1) {
            std::cout << "select error" << std::endl;
            break;
```

```
        };
        if (ret == 0) {
            continue;
        }
        for(i=0;i<fd_max+1;i++){
            if(FD_ISSET(i,&reads_copy)){
                if(i==serv_socket){
                    serv_len = sizeof(accept_addr);
                    def = accept(serv_socket, (struct sockaddr *) &accept_addr,
&serv_len);
                    if (def < 0) {
                        error_handling("accept error");
                    }
                    FD_SET(def,&reads);
                    if(fd_max<def){
                        fd_max=def;
                    }
                    std::cout<<"连接客户端"<<def<<std::endl;
                }
                else{
                    if ((recv_a=read(accept_socket, buf, N) )< 0) {
                        error_handling("recv error");
                    }
                    if(recv_a==0){
                        FD_CLR(i,&reads);
                        close(i);
                        std::cout<<"关闭客户端: "<<i<<std::endl;
                    }
                    if (send(accept_socket, buf, 1024, 0) < 0) {
                        error_handling("send error");
                    }
                }
            }
        }



    }
    close(serv_socket);
    return 0;

}
```