IO多路复用

功能：IO多路复用使得程序能够同时监听多个文件描述符，能够提高程序的性能，Linux下实现IO多路复用的系统调用主要有select、poll、epoll。

两种模型

模型

1、（BIO模型）阻塞等待
服务端在等待客户端连接以及读取客户端信息两处位置都会进行阻塞。
好处：不占用CPU的时间片。
缺点：同一时刻只能处理一个操作，效率低。
解决缺点：可以使用多进程/多线程的方式，使得服务端能够和多个客户端连接通信。
缺点：线程或者进程会消耗资源，且线程或者进程调度比较消耗CPU资源。

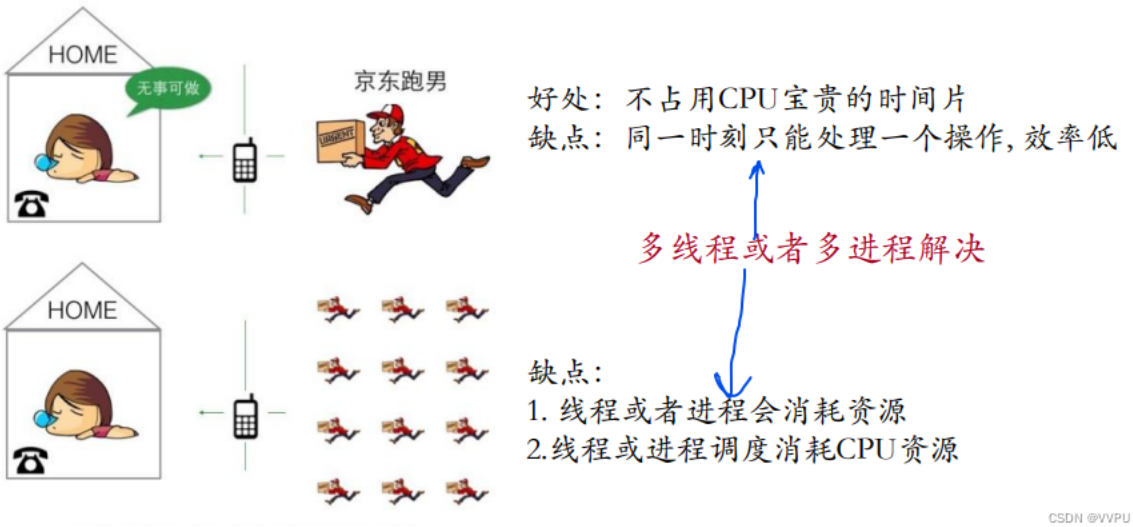
BIO模型想要能够并发就必须要使用多进程或者多线程来解决问题，因此就不可避免的遇到上面的缺点。
该模型的缺点的根本问题在于阻塞等待。

2、（NIO模型）非阻塞，忙轮询
服务端在等待客户端连接和读取客户端信息两处都不会进行阻塞，而是不断的轮询。
这种模式下，当有多个客户端连接时，需要有一个表记录客户端通信的socket fd；假设有1万客户端连接，那么会遍历所有的客户端，进行read的系统调用。
此时，即使客户端没有向服务端发送数据，服务端也会进行系统调用read。
优点：提高了程序的执行效率
缺点：需要占用更多的CPU和系统资源。

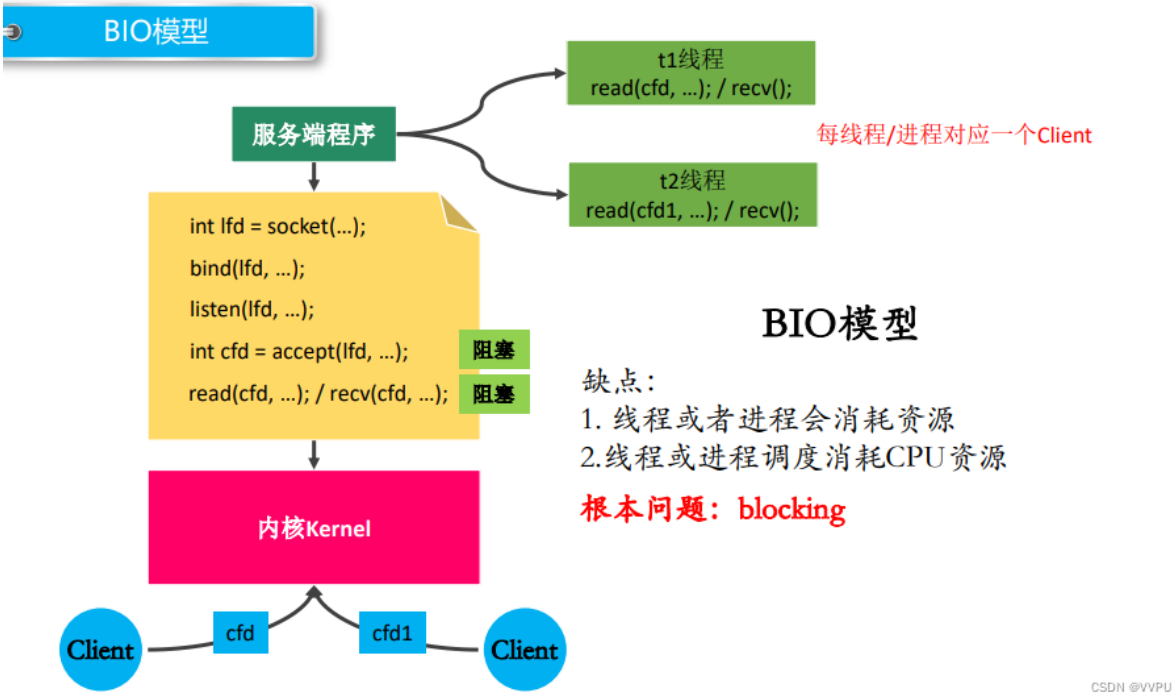
为了解决NIO模型的缺点，需要使用IO多路复用技术，（select / poll / epoll）三种常用的API，每种都可以解决NIO模型的缺点。
selet / poll ： 这两种API只会告诉服务端有几个客户端发送了数据，而不会告诉具体是哪几个。所以还需要遍历，找到那几个，但是不是像上面的直接read系统调用。
epoll ：而这个API功能更加强大，既能告诉客户端有几个客户端还能告诉是哪几个客户端。
三种API的功能都是交由内核实现的。

BIO模型图解

图一

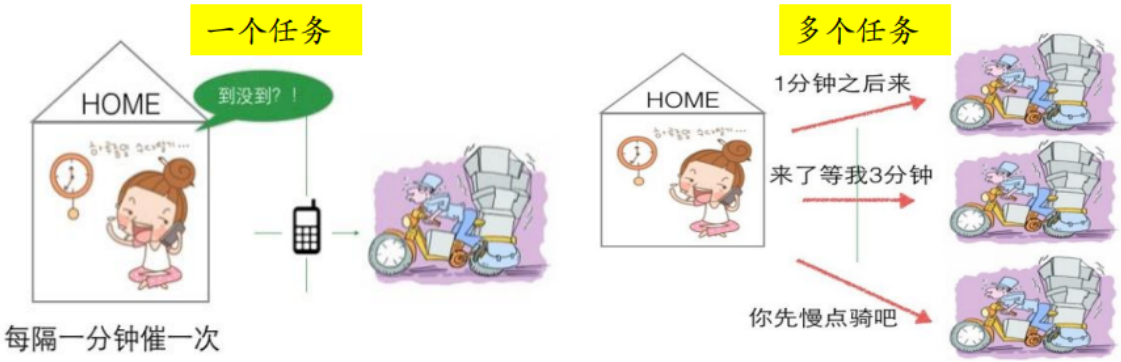


图二



NIO模型图解

非阻塞, 忙轮询



优点: 提高了程序的执行效率
缺点: 需要占用更多的CPU和系统资源

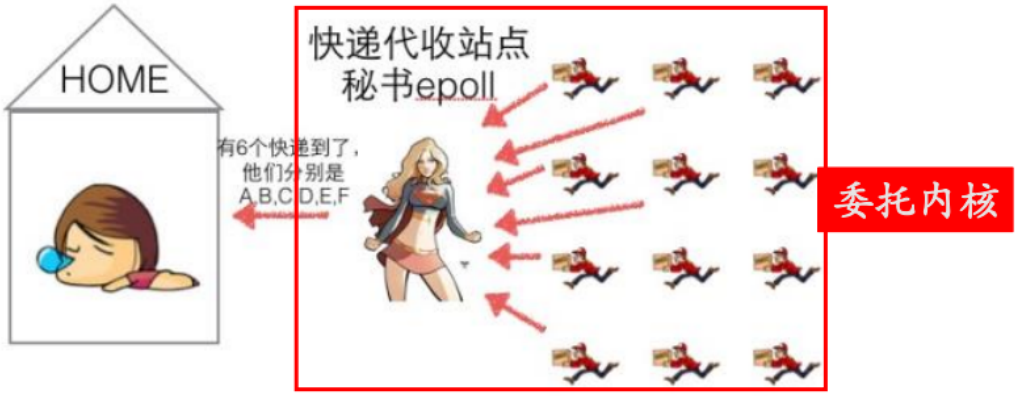
Solve

使用IO多路转接技术select/poll/epoll

第一种： select/poll

select代收员比较懒，她只会告诉你有几个快递到了，但是哪个快递，你需要挨个遍历一遍。

第二种： epoll



epoll代收快递员很勤快，她不仅会告诉你有几个快递到了，还会告诉你是哪个快递公司的快递

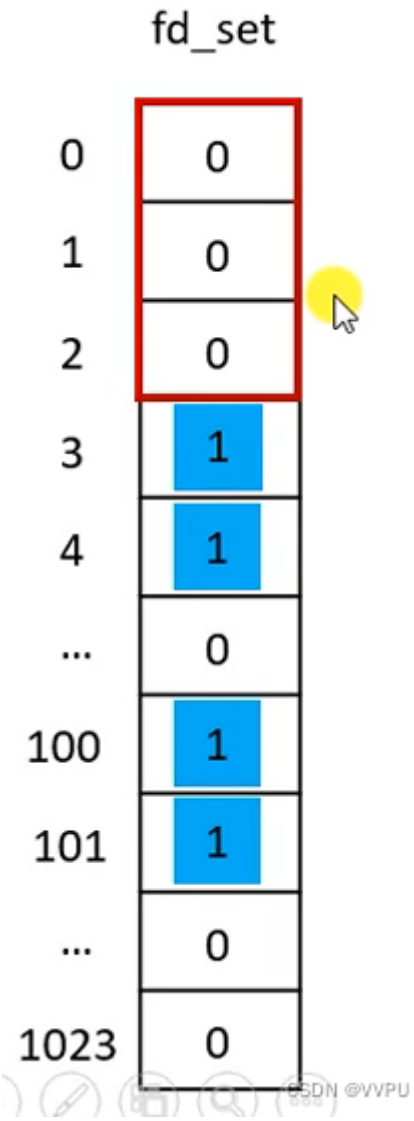
select函数

函数介绍

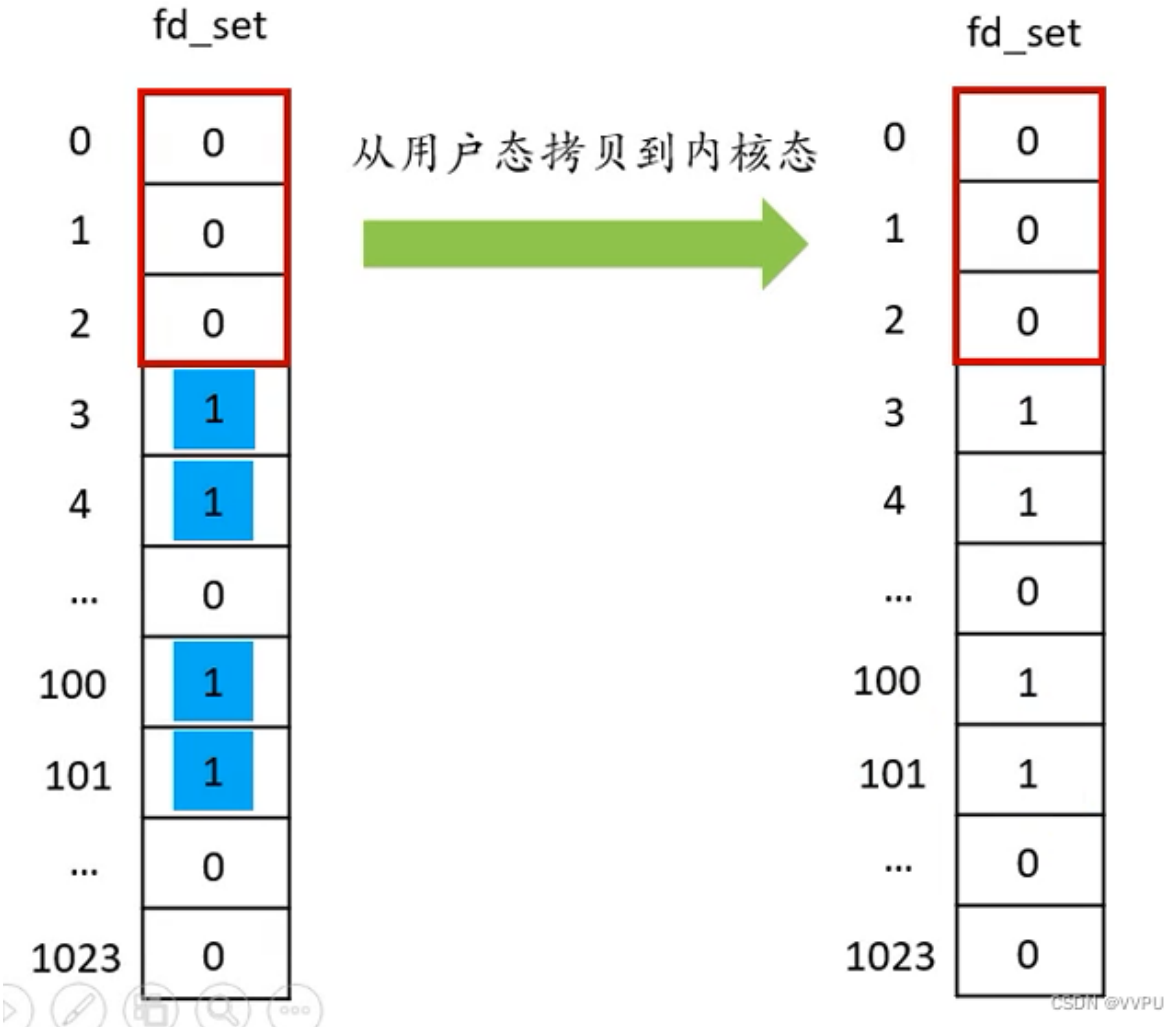
```
select：
    主旨思想：
        1. 首先要构造一个关于文件描述符的列表，将要监听的文件描述符添加到该列表中。
        2. 调用这个系统函数，监听该列表中的文件描述符，直到这些描述符中的一个或者多个进行I/O操作时，该函数才返回。
            a.这个函数是阻塞的
            b.函数对文件描述符的检测的操作是由内核完成的
        3. 在返回时，它会告诉进程有多少（哪些）描述符要进行I/O操作。
    函数详解：
        #include <sys/time.h>
        #include <sys/types.h>
        #include <unistd.h>
        #include <sys/select.h>
        int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
            - 参数：
            - nfds : 委托内核检测的最大文件描述符的值 + 1(遍历从0开始，为了能检测到最后一个)
            - readfds : 要检测的文件描述符的读的集合，委托内核检测哪些文件描述符的读的属性
                - 一般检测读操作
                - 对应的是对方发送过来的数据，因为读是被动的接收数据，检测的就是读缓冲区
                - 是一个传入传出参数
            - writefds : 要检测的文件描述符的写的集合，委托内核检测哪些文件描述符的写的属性
                - 委托内核检测写缓冲区是不是还可以写数据（不满的就可以写）
            - exceptfds : 检测发生异常的文件描述符的集合
            - timeout : 设置的超时时间
                struct timeval {
                    long tv_sec;    /* seconds */
                    long tv_usec;   /* microseconds */
                };
                - NULL : 永久阻塞，直到检测到了文件描述符有变化
                - tv_sec = 0 tv_usec = 0,  不阻塞
                - tv_sec > 0 tv_usec > 0,  阻塞对应的时间
        - 返回值 :
            - -1 : 失败
            - >0(n) : 检测的集合中有n个文件描述符发生了变化

        /*
            fd_set类型：sizeof(fd_set) = 128byte = 1024bit
            在内核内部，对该类型的操作是按照位进行操作的，0 / 1
            第二个参数readfds，如果有n个文件文件描述符都进行了写操作，那么服务端就可以读取了，此时readfds中对应这几个二进制位都会设置位1，如果原本为1，但是现在没有检测到置为0
            第三个参数writefds，如果写缓冲区没有满可写则置为1，否则为0
        */

        // 将参数文件描述符fd对应的标志位设置为0
        void FD_CLR(int fd, fd_set *set);
        // 判断fd对应的标志位是0还是1，  返回值 :  fd对应的标志位的值，0，返回0，  1，返回1
        int FD_ISSET(int fd, fd_set *set);
        // 将参数文件描述符fd 对应的标志位，设置为1
        void FD_SET(int fd, fd_set *set);
        // fd_set一共有1024 bit, 全部初始化为0
        void FD_ZERO(fd_set *set);
```

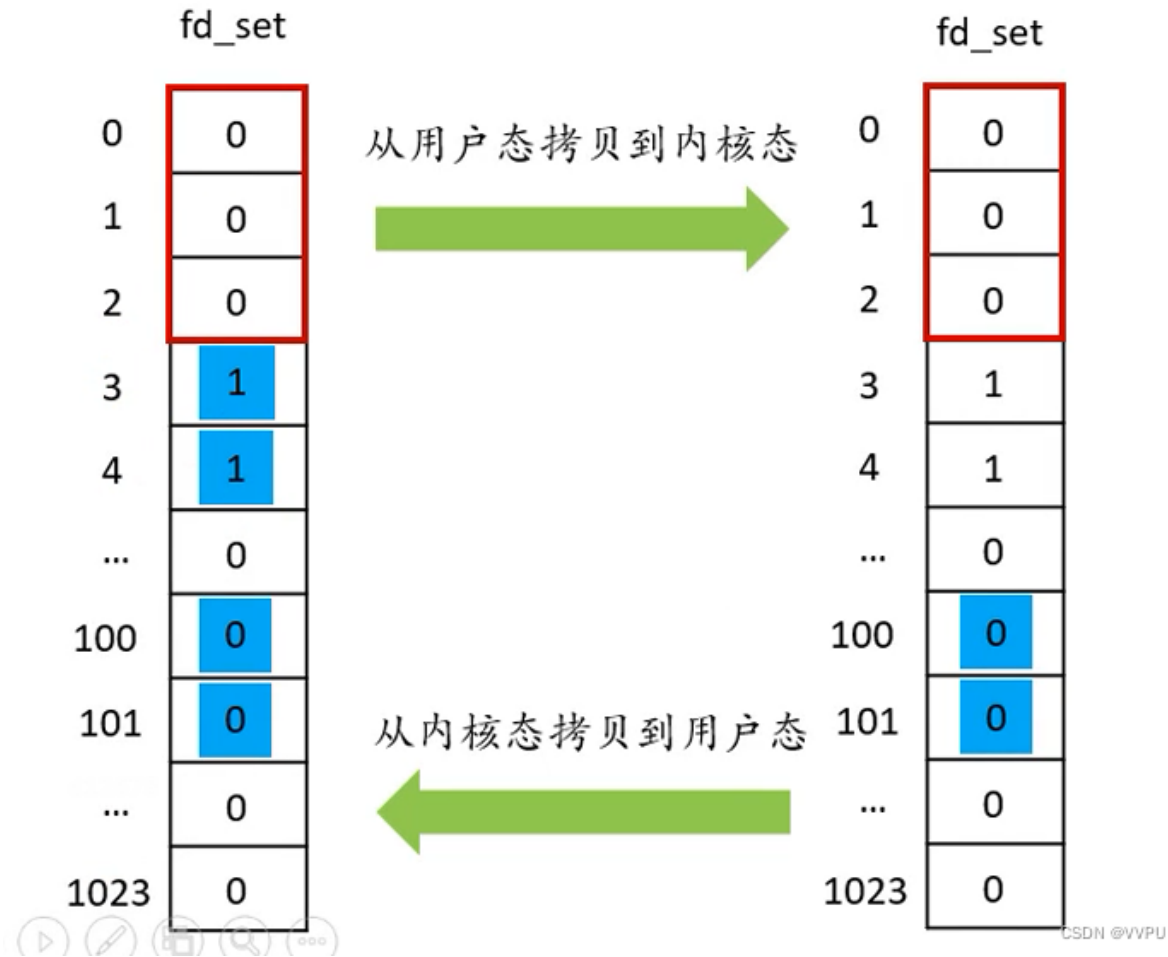
select工作过程

假设ABCD四个客户端连接到服务器，他们对应的文件描述符分别是3，4，100， 101。
首先要建立一个文件描述符表fd_set reads，用于检测是否有客户端发送过来了数据可以读。一开始，reads的1024位都是0；此时要检测的是上面的四个客户端，把对应的文件描述符位置的0设置为1-->通过`FD_SET(3, &reads)，FD_SET(4, &reads)，FD_SET(100, &reads)，FD_SET(101, &reads)`。



然后调用`select(101 + 1, &reads, NULL, NULL, NULL);`第二个参数就是要检测的集合。此时，select会把用户态的reads拷贝到内核态，委托内核去检测。内核就会去检测reads，判断哪些文件描述符需要去检测。如果AB客户端发送了数据。在遍历过程中，只会判断标志位为1的，如果该标志位对应的文件描述符发送了数据那么他的标志位还是为1，而被检测的其他的就要置为0。根据条件，只有AB发送了数据，CD未发送，那么CD的标志位需要置为0。





接上，内核区处理完之后，会把reads再次拷贝到用户区，此时用户区就知道哪些文件描述发送了数据。

遍历reads，如果标志位为1，那么就可以进行read通信了。

select代码实现

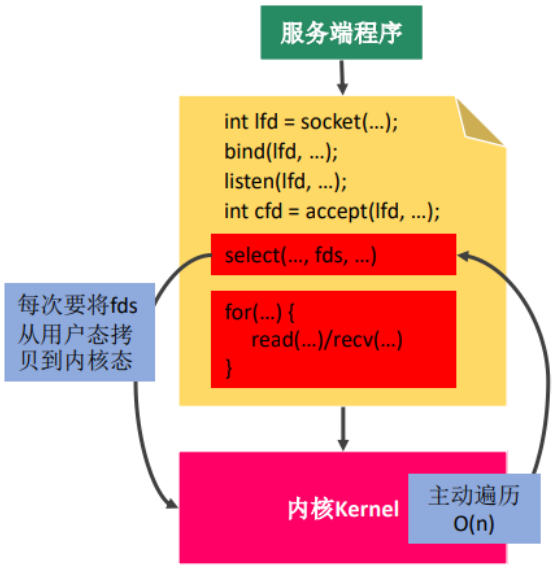
服务端：（客户端只要能连接通信就行，和上面的一样）
即使没有使用多线程多进程也能实现并发。

```
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>

int main()
{
    // 创建socket
    int lfd = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in saddr;
    saddr.sin_port = htons(9999);
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;

    // 绑定
    bind(lfd, (struct sockaddr *)&saddr, sizeof(saddr));

    // 监听
    listen(lfd, 8);

    // 创建一个fd_set的集合，存放的是需要检测的文件描述符
    /*
        根据select工作流程，每次去内核区之后，会把没有检测到数据的文件描述符从1置为0，
        所以这里用temp变量去进行select操作，rdset则负责记录需要检测的文件描述符，防止把需要检测的文件描述符的标志位置为了0

        监听文件描述符也是需要加入到集合中的，客户端的连接也是一次发送数据，用于监听客户端连接。
    */
    fd_set rdset, tmp;
    FD_ZERO(&rdset);
    FD_SET(lfd, &rdset);
    int maxfd = lfd;

    while (1)
    {
        tmp = rdset;    //每次循环都要更新tmp，把要检测的文件描述符的标志位重新赋值。因为经历过一次select，里面的标志位已经变了。

        // 调用select系统函数，让内核帮检测哪些文件描述符有数据。
        int ret = select(maxfd + 1, &tmp, NULL, NULL, NULL);
        if (ret == -1)
        {
            perror("select");
            exit(-1);
        }
        else if (ret == 0)
        {
```

```
            continue;   //select的最后一个超时参数置为了NULL，所以不会遇到ret == 0的情况
        }
        else if (ret > 0)   //说明有文件描述符发生了数据变动
        {
            // 说明检测到了有文件描述符的对应的缓冲区的数据发生了改变
            if (FD_ISSET(lfd, &tmp))
            {
                // 表示有新的客户端连接进来了。此时获取通信文件描述符，并且将其添加到rdset集合中，后期检测其是否发送了数据
                struct sockaddr_in cliaddr;
                int len = sizeof(cliaddr);
                int cfd = accept(lfd, (struct sockaddr *)&cliaddr, &len);

                // 将新的文件描述符加入到集合中
                FD_SET(cfd, &rdset);

                // 更新最大的文件描述符
                maxfd = maxfd > cfd ? maxfd : cfd;
            }
            //遍历到最大的文件描述符为止，从监听文件描述符的后一个文件描述符开始。
            //判断是否有文件描述符的标志位置为了1，如果为1，那么就说明有客户端发送过来的数据可读。
            for (int i = lfd + 1; i <= maxfd; i++)
            {
                if (FD_ISSET(i, &tmp))  //对标志位置为1的文件描述符处理，这里用tmp表，因为它是从内核区返回过来的数据。
                {
                    // 说明这个文件描述符对应的客户端发来了数据
                    char buf[1024] = {0};
                    int len = read(i, buf, sizeof(buf));
                    if (len == -1)
                    {
                        perror("read");
                        exit(-1);
                    }
                    else if (len == 0)
                    {
                        //如果出现客户端断开连接，那么既要关闭这个通信文件描述符，也要将rdset表中的对应的标志位置为0，下次将不再监测它。
                        printf("client closed...\n");
                        close(i);
                        FD_CLR(i, &rdset);
                    }
                    else if (len > 0)
                    {
                        printf("read buf = %s\n", buf);
                        write(i, buf, strlen(buf) + 1);
                    }
                }
            }
        }
    }
    close(lfd);
    return 0;
}
```

poll函数

select的缺点



缺点：

1.每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大

2.同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大

3.select支持的文件描述符数量太小了，默认是1024

4.fds集合不能重用，每次都需要重置

poll详解

对于select的缺点，而poll函数相当于改进版的select，可以有效解决这些问题。

1、函数详解

```
#include <poll.h>
struct pollfd {
    int fd; /* 委托内核检测的文件描述符 */
    short events; /* 委托内核检测文件描述符的什么事件 */
    short revents; /* 文件描述符实际发生的事件，由内核赋值返回 */
};

示例：
struct pollfd myfd;
myfd.fd = 5;    //监测的文件描述符为5
myfd.events = POLLIN | POLLOUT; //监测两种事件，要监测多个事件用按位或

int poll(struct pollfd *fds, nfds_t nfds, int timeout);
- 参数：
    - fds ： 是一个struct pollfd 结构体数组，这是一个需要检测的文件描述符的集合（传入传出参数）
    - nfds ： 这个是第一个参数数组中最后一个有效元素的下标 + 1
    - timeout ： 阻塞时长
        0 ： 不阻塞
        -1 ： 阻塞，当检测到需要检测的文件描述符有变化，解除阻塞
        >0 ： 阻塞的时长
- 返回值：
    -1 ： 失败
    >0（n） ： 成功,n表示检测到集合中有n个文件描述符发生变化
```

pollfd结构体中的events和revents可以使用的值。

事件	常值	作为events的值	作为revents的值	说明
读事件	POLLIN	✔	✔	普通或优先带数据可读
	POLLRDNORM	✔	✔	普通数据可读
	POLLRDBAND	✔	✔	优先级带数据可读
	POLLPRI	✔	✔	高优先级数据可读
写事件	POLLOUT	✔	✔	普通或优先带数据可写
	POLLWRNORM	✔	✔	普通数据可写
	POLLWRBAND	✔	✔	优先级带数据可写
错误事件	POLLERR		✔	发生错误
	POLLHUP		✔	发生挂起
	POLLNVAL		✔	描述不是打开的文件

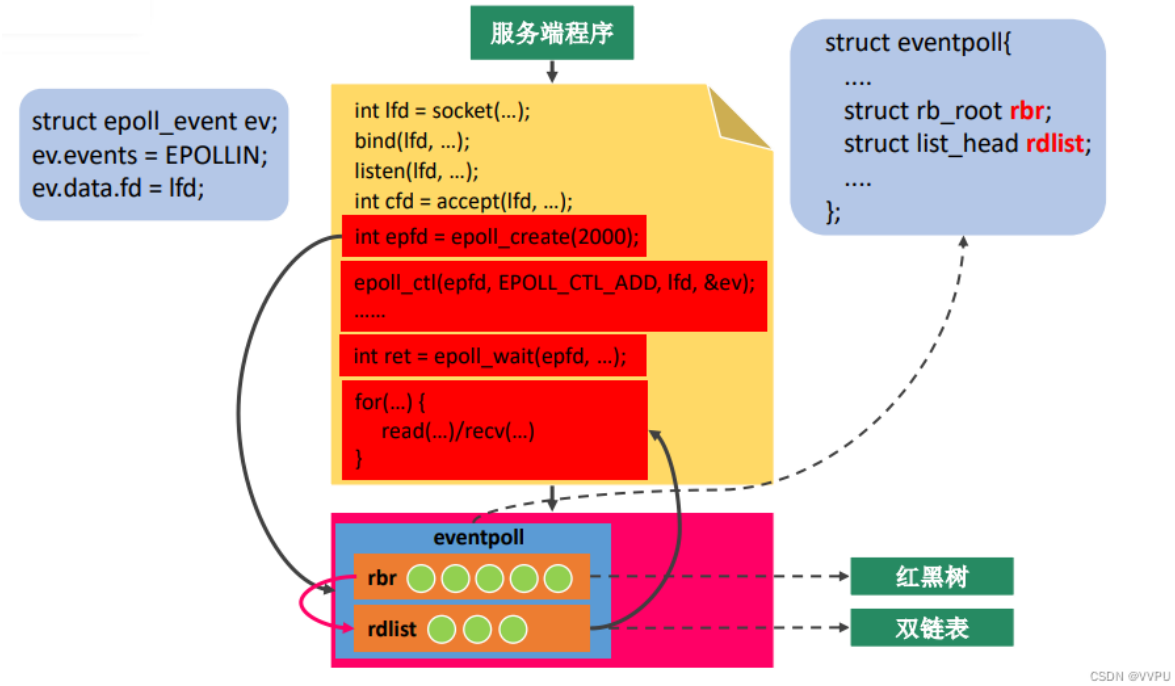
代码实现

```c
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <poll.h>

int main() {

    // 创建socket
    int lfd = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in saddr;
    saddr.sin_port = htons(9999);
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;

    // 绑定
    bind(lfd, (struct sockaddr *)&saddr, sizeof(saddr));

    // 监听
    listen(lfd, 8);

    // 初始化检测的文件描述符数组，可以判断哪个元素的文件描述符没有被使用
    struct pollfd fds[1024];
    for(int i = 0; i < 1024; i++) {
        fds[i].fd = -1;
        fds[i].events = POLLIN;
    }
    fds[0].fd = lfd;    //结构体的第一个元素设置为服务器的监听文件描述符
    int nfds = 0;

    while(1) {

        // 调用poll系统函数，让内核帮检测哪些文件描述符有数据
        int ret = poll(fds, nfds + 1, -1);
        if(ret == -1) {
            perror("poll");
            exit(-1);
        } else if(ret == 0) {   //阻塞情况下，不会有这个返回情况
            continue;
        } else if(ret > 0) {
```

```
            // 说明检测到了有文件描述符的对应的缓冲区的数据发生了改变（一种情况是有客户端连接，另一种情况是有客户端发送数据；或者两种情况都有）
            if(fds[0].revents & POLLIN) {    //根据pollfd结构体的第三个参数判断是否有可读的数据，需要按位与判断，不能直接用等于。
                // 表示有新的客户端连接进来了
                struct sockaddr_in cliaddr;
                int len = sizeof(cliaddr);
                int cfd = accept(lfd, (struct sockaddr *)&cliaddr, &len);

                // 将新的文件描述符加入到集合中，找一个可以使用的（重用原来的）
                int i = 1;
                for(; i < 1024; i++) {  //从1开始，0已经被监听描述符占用了，除非服务器终止，否则会被一直占用
                    if(fds[i].fd == -1) {
                        fds[i].fd = cfd;
                        fds[i].events = POLLIN;
                        break;
                    }
                }

                // 更新最大的文件描述符的索引
                //nfds = nfds > cfd ? nfds : cfd;
                nfds = nfds > i ? nfds : i;     //索引最大的一个有效元素所在的索引，poll函数第二个参数要使用
            }

            for(int i = 1; i <= nfds; i++) {    //这里是判断是否有文件描述符发送过来了数据，从数组的第二个元素开始遍历到最大的位置
                if(fds[i].revents & POLLIN) {
                    // 说明这个文件描述符对应的客户端发来了数据
                    char buf[1024] = {0};
                    int len = read(fds[i].fd, buf, sizeof(buf));
                    if(len == -1) {
                        perror("read");
                        exit(-1);
                    } else if(len == 0) {
                        printf("client closed...\n");
                        close(fds[i].fd);   //客户端连接断开，然后将文件描述符设置为-1，方便给新的客户端重用
                        fds[i].fd = -1;
                    } else if(len > 0) {
                        printf("read buf = %s\n", buf);
                        write(fds[i].fd, buf, strlen(buf) + 1);
                    }
                }
            }

        }

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

epoll函数

epoll工作模式



1、首先，`epoll_create`函数会创建一个epoll实例，返回值是一个文件描述符指向内核的一块空间，这块空间是epoll的工作空间，主要有两块重要的内存，一块是红黑树类型的`rbr`，里面存放的是所有要监测的文件描述符；另一块是双链表类型的`rdlist`，里面存放的是被监测到有数据变动的文件描述符（来自于`rbr`）。

2、然后，遇到一个新的文件描述符就将这个文件描述符通过`epoll_ctl`函数添加到上面的epoll实例中，也就是将其放到rbr空间中，作为待监测的文件描述符。同时，这个函数可以设置监测文件描述符发生的行为，比如客户端发送到服务端数据。

3、最后，如果内核监测到rbr中的文件描述符出现了`epoll_ctl`设置的要监听的的行为，那么就会将其拷贝的`rdlist`。`epoll_wait`函数则是可以获取到rdlist中的数据，通过传入传出参数返回，它的返回值就是数据变动的文件描述符的数量。

4、根据epoll_wait的传出参数，遍历之，这是一个结构体数组。获取每个元素中的文件描述符，判断它是监听文件描述符还是其他的，如果是监听文件描述符，那么就有新的客户端连接，此时就要将其添加到`rbr`空间中（使用`epoll_ctl`），如果是其他文件描述符就说明有客户端发送了数据，此时可以根据文件描述符读取数据。

函数详解

```
#include <sys/epoll.h>
// 创建一个新的epoll实例。在内核中创建了一个数据，这个数据中有两个比较重要的数据，一个是需要检测的文件描述符的信息（红黑树），还有一个是就绪列表，存放检测到数据发送改变的文件描述符信息（双向链表）。
int epoll_create(int size);
- 参数：
    size ： 目前没有意义了。随便写一个数，必须大于0
- 返回值：
    -1 ： 失败
    > 0 ： 文件描述符，操作epoll实例的

// 对epoll实例进行管理：添加文件描述符信息，删除信息，修改信息
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
- 参数：
    - epfd ： epoll实例对应的文件描述符
```

```
    - op : 要进行什么操作（将fd文件描述符设置到rbr中，还是从rbr中删除，或者是修改要监测fd的event事情）
        EPOLL_CTL_ADD: 添加
        EPOLL_CTL_MOD: 修改
        EPOLL_CTL_DEL: 删除
    - fd : 要检测的文件描述符
    - event : 检测文件描述符什么事情

// 检测函数
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
- 参数:
    - epfd : epoll实例对应的文件描述符
    - events : 传出参数，保存了发送了变化的文件描述符的信息
    - maxevents : 第二个参数结构体数组的大小
    - timeout : 阻塞时间
        - 0 : 不阻塞
        - -1 : 阻塞，直到检测到fd数据发生变化，解除阻塞
        - > 0 : 阻塞的时长（毫秒）
- 返回值:
    - 成功，返回发送变化的文件描述符的个数 > 0
    - 失败 -1

## 两类结构体
typedef union epoll_data {
    void *ptr;
    int fd; //一般情况使用这个就可以了
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;

struct epoll_event {
    uint32_t events;    /* Epoll events */
    epoll_data_t data;  /* User data variable */
};
常见的Epoll检测事件:
    - EPOLLIN
    - EPOLLOUT
    - EPOLLERR
    - EPOLLET
```

代码实现

```
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    // 1、创建套接字
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd == -1)
    {
        perror("socket");
        exit(-1);
```

```
    }

    //连接到服务器
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    inet_pton(AF_INET, "192.168.47.131", &(addr.sin_addr.s_addr));
    addr.sin_port = htons(9999);
    int ret = connect(fd, (struct sockaddr *)&addr, sizeof(addr));
    if (ret == -1)
    {
        perror("connect");
        exit(-1);
    }

    // 3. 通信
    char recvBuf[1024] = {0};
    while (1)
    {

        char *data = "hello,i am client";
        // 给客户端发送数据
        write(fd, data, strlen(data));

        // sleep(1);
        usleep(1);

        int len = read(fd, recvBuf, sizeof(recvBuf));
        if (len == -1)
        {
            perror("read");
            exit(-1);
        }
        else if (len > 0)
        {
            printf("recv server data : %s\n", recvBuf);
        }
        else if (len == 0)
        {
            // 表示服务器端断开连接
            printf("server closed...");
            break;
        }
    }

    // 关闭连接
    close(fd);

    return 0;
}
```

epoll的两种工作模式

```
LT 模式 （水平触发）
    假设委托内核检测读事件 -> 检测fd的读缓冲区
    读缓冲区有数据 - > epoll检测到了会给用户（服务端）通知
        a.用户不读数据，数据一直在缓冲区，epoll 会一直通知
        b.用户只读了一部分数据，epoll会通知
        c.缓冲区的数据读完了，不通知

    LT（level - triggered）是缺省的工作方式，并且同时支持 block 和 no-block socket。在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的 fd 进行 IO 操作。如果你不作任何操作，内核还是会继续通知你的。
```

```
ET 模式（边沿触发）
    假设委托内核检测读事件 -> 检测fd的读缓冲区
    读缓冲区有数据 - > epoll检测到了会给用户通知
        a.用户不读数据，数据一致在缓冲区中，epoll下次检测的时候就不通知了
        b.用户只读了一部分数据，epoll不通知
        c.缓冲区的数据读完了，不通知

    ET（edge - triggered）是高速工作方式，只支持 no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了。
    但是请注意，如果一直不对这个 fd 作 IO 操作（从而导致它再次变成未就绪），内核不会发送更多的通知（only once）。
    ET 模式在很大程度上减少了 epoll 事件被重复触发的次数，因此效率要比 LT 模式高。epoll工作在 ET 模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。
```

水平模式代码实现

```c
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/epoll.h>

int main()
{

    // 创建socket
    int lfd = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in saddr;
    saddr.sin_port = htons(9999);
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;

    // 绑定
    bind(lfd, (struct sockaddr *)&saddr, sizeof(saddr));

    // 监听
    listen(lfd, 8);

    // 调用epoll_create()创建一个epoll实例
    int epfd = epoll_create(100);

    // 将监听的文件描述符相关的检测信息添加到epoll实例中
```

```
    struct epoll_event epev;
    epev.events = EPOLLIN;
    epev.data.fd = lfd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, lfd, &epev);

    struct epoll_event epevs[1024];

    while (1)
    {

        int ret = epoll_wait(epfd, epevs, 1024, -1);
        if (ret == -1)
        {
            perror("epoll_wait");
            exit(-1);
        }

        printf("ret = %d\n", ret);

        for (int i = 0; i < ret; i++)
        {

            int curfd = epevs[i].data.fd;

            if (curfd == lfd)
            {
                // 监听的文件描述符有数据达到，有客户端连接
                struct sockaddr_in cliaddr;
                int len = sizeof(cliaddr);
                int cfd = accept(lfd, (struct sockaddr *)&cliaddr, &len);

                epev.events = EPOLLIN;
                epev.data.fd = cfd;
                epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &epev);
            }
            else
            {
                if (epevs[i].events & EPOLLOUT)
                {
                    continue;
                }
                // 有数据到达，需要通信
                /*
                    char buf[5]数组用于接收来自客户端数据，当客户端发送过来的数据大于5个字节时，
                    水平模式下，会不断通知有数据到达，epoll_wait函数就会接收到通知，直到读完位置，每次都会接着后面的数据读取。
                */
                char buf[5] = {0}; //该数组用于接收数据，设置尽量小些，一次装不满获取的数据
                int len = read(curfd, buf, sizeof(buf));
                if (len == -1)
                {
                    perror("read");
                    exit(-1);
                }
                else if (len == 0)
                {
```

```
                    printf("client closed...\n");
                    epoll_ctl(epfd, EPOLL_CTL_DEL, curfd, NULL);
                    close(curfd);
                }
                else if (len > 0)
                {
                    printf("read buf = %s\n", buf);
                    write(curfd, buf, strlen(buf) + 1);
                }
            }
        }
    }

    close(lfd);
    close(epfd);
    return 0;
}
```

边沿模式代码实现

```
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/epoll.h>
#include <fcntl.h>
#include <errno.h>

int main()
{
    // 创建socket
    int lfd = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in saddr;
    saddr.sin_port = htons(9999);
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;

    // 绑定
    bind(lfd, (struct sockaddr *)&saddr, sizeof(saddr));

    // 监听
    listen(lfd, 8);

    // 调用epoll_create()创建一个epoll实例
    int epfd = epoll_create(100);

    // 将监听的文件描述符相关的检测信息添加到epoll实例中
    struct epoll_event epev;
    epev.events = EPOLLIN;
    epev.data.fd = lfd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, lfd, &epev);

    struct epoll_event epevs[1024];

    while (1)
    {
```

```
        int ret = epoll_wait(epfd, epevs, 1024, -1);
        if (ret == -1)
        {
            perror("epoll_wait");
            exit(-1);
        }

        printf("ret = %d\n", ret);

        for (int i = 0; i < ret; i++)
        {
            int curfd = epevs[i].data.fd;

            if (curfd == lfd)
            {
                // 监听的文件描述符有数据达到，有客户端连接
                struct sockaddr_in cliaddr;
                int len = sizeof(cliaddr);
                int cfd = accept(lfd, (struct sockaddr *)&cliaddr, &len);

                // 设置cfd属性非阻塞。和客户端通信的文件描述要设置为非阻塞，因为使用了边沿触发的模式。
                //后面要通过循环才能完全读取客户端一次发过来的数据，为了防止读完数据后，read一直处于阻塞状态，导致无法继续监测文件描述符的问题。
                int flag = fcntl(cfd, F_GETFL);
                flag | O_NONBLOCK;
                fcntl(cfd, F_SETFL, flag);

                epev.events = EPOLLIN | EPOLLET; // 设置边沿触发（需要主动开启该模式）
                epev.data.fd = cfd;
                epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &epev);
            }
            else
            {
                if (epevs[i].events & EPOLLOUT)
                {
                    continue;
                }

                /*
                    边沿触发模式下，只有在客户端发送了数据epoll_wait才会被触发，
                    如果下面的buf容量太小，一次性无法读取客户端发送过来的数据，那么只能截取到其中一部分，
                    即使缓冲区中还有数据没有读完，下次循环也不会触发epoll_wait，
                    只有当用户端再次发送数据时，服务端才能被触发读取到缓冲区的数据，不是新发送的数据，而是接着上次读到位置继续读取缓冲区中的数据。

                    所以，就不得不使用循环来解决一次性无法读取所有数据的问题，不然数据会不完整。
                */

                // 循环读取出所有数据
                char buf[5];
                int len = 0;
                while ((len = read(curfd, buf, sizeof(buf))) > 0)
                {
                    // 打印数据
                    // printf("recv data : %s\n", buf);
```

```
                    write(STDOUT_FILENO, buf, len);
                    write(curfd, buf, len);
                }
                if (len == 0)
                {
                    printf("client closed....");
                }
                else if (len == -1)
                {
                    if (errno == EAGAIN)
                    {
                        // EAGAIN --> the file descriptor fd refers to a file
other than a socket and has been marked non-blocking, and the read would block.
                        printf("data over.....");
                    }
                    else
                    {
                        perror("read");
                        exit(-1);
                    }
                }
            }
        }
    }

    close(lfd);
    close(epfd);
    return 0;
}
```

```
#include <stdio.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main() {

    // 创建socket
    int fd = socket(PF_INET, SOCK_STREAM, 0);
    if(fd == -1) {
        perror("socket");
        return -1;
    }

    struct sockaddr_in seraddr;
    inet_pton(AF_INET, "127.0.0.1", &seraddr.sin_addr.s_addr);
    seraddr.sin_family = AF_INET;
    seraddr.sin_port = htons(9999);

    // 连接服务器
    int ret = connect(fd, (struct sockaddr *)&seraddr, sizeof(seraddr));

    if(ret == -1){
        perror("connect");
        return -1;
    }
```

```
    int num = 0;
    while(1) {
        char sendBuf[1024] = {0};
        // sprintf(sendBuf, "send data %d", num++);
        fgets(sendBuf, sizeof(sendBuf), stdin);

        write(fd, sendBuf, strlen(sendBuf) + 1);

        // 接收
        int len = read(fd, sendBuf, sizeof(sendBuf));
        if(len == -1) {
            perror("read");
            return -1;
        }else if(len > 0) {
            printf("read buf = %s\n", sendBuf);
        } else {
            printf("服务器已经断开连接...\n");
            break;
        }
    }

    close(fd);

    return 0;
}
```