

# 零声教育 Mark 老师 QQ: 2548898954

---

## 网络编程关注的问题

---

### 连接的建立

分为两种：服务端处理接收客户端的连接；服务端作为客户端连接第三方服务；

```
1
2 int clientfd = accept(listenfd, addr, sz);
3
4 // 举例为非阻塞io，阻塞io成功直接返回0；
5 int connectfd = socket(AF_INET, SOCK_STREAM, 0);
6 int ret = connect(connectfd, (struct sockaddr
    *)&addr, sizeof(addr));
7 // ret == -1 && errno == EINPROGRESS 正在建立连接
8 // ret == -1 && errno = EISCONN 连接建立成功
```

### 连接的断开

分为两种：主动断开和被动断开；

```
1 // 主动关闭
2 close(fd);
3 shutdown(fd, SHUT_RDWR);
4 // 主动关闭本地读端，对端写段关闭
5 shutdown(fd, SHUT_RD);
6 // 主动关闭本地写端，对端读段关闭
7 shutdown(fd, SHUT_WR);
8
9 // 被动：读端关闭
```

```

10 // 有的网络编程需要支持半关闭状态
11 int n = read(fd, buf, sz);
12 if (n == 0) {
13     close_read(fd);
14     // write()
15     // close(fd);
16 }
17 // 被动：写端关闭
18 int n = write(fd, buf, sz);
19 if (n == -1 && errno == EPIPE) {
20     close_write(fd);
21     // close(fd);
22 }

```

## 消息的到达

从读缓冲区中读取数据；

```

1 int n = read(fd, buf, sz);
2 if (n < 0) { // n == -1
3     if (errno == EINTR || errno == EWOULDBLOCK)
4         break;
5     close(fd);
6 } else if (n == 0) {
7     close(fd);
8 } else {
9     // 处理 buf
10 }

```

## 消息发送完毕

往写缓冲区中写数据；

```
1 int n = write(fd, buf, dz);
2 if (n == -1) {
3     if (errno == EINTR || errno == EWOULDBLOCK) {
4         return;
5     }
6     close(fd);
7 }
```

## 网络 IO 职责

### 检测 IO

io 函数本身可以检测 io 的状态；但是只能检测一个 fd 对应的状态；io 多路复用可以同时检测多个 io 的状态；区别是：io 函数可以检测具体状态；io 多路复用只能检测出可读、可写、错误、断开等笼统的事件；

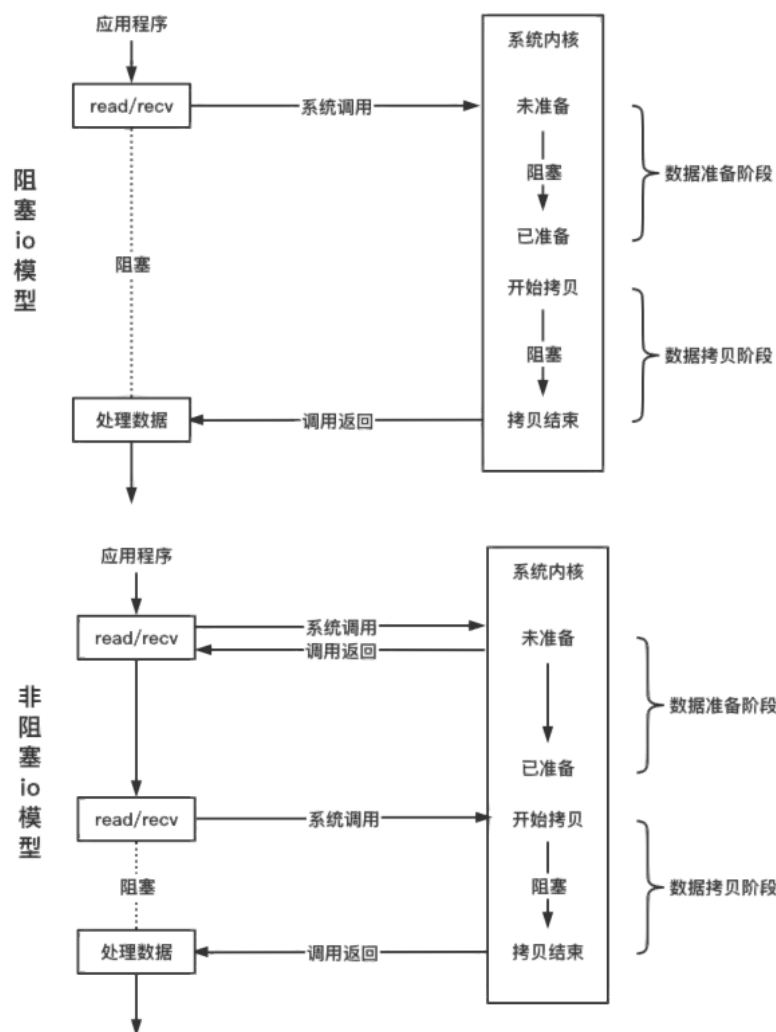
### 操作 IO

只能使用 io 函数来进行操作；分为两种操作方式：阻塞 io 和非阻塞 io；

## 阻塞 IO 和非阻塞 IO

- 阻塞在网络线程；
- 连接的 fd 阻塞属性决定了 io 函数是否阻塞；
- 具体差异在：io 函数在数据未到达时是否立刻返回；

```
1 // 默认情况下，fd 是阻塞的，设置非阻塞的方法如下；
2 int flag = fcntl(fd, F_GETFL, 0);
3 fcntl(fd, F_SETFL, flag | O_NONBLOCK);
```



## IO 多路复用

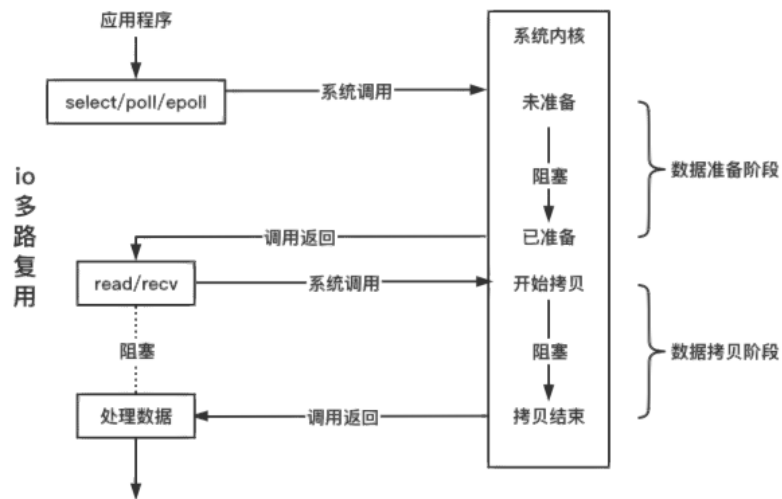
io 多路复用只负责检测 io，不负责操作 io；

```
int n = epoll_wait(epfd, evs, sz, timeout);
```

`timeout = -1` 一直阻塞直到网络事件到达；

`imeout = 0` 不管是否有事件就绪立刻返回；

`timeout = 1000` 最多等待 1 s，如果 1 s 内没有事件触发则返回；



# epoll

## 结构以及接口

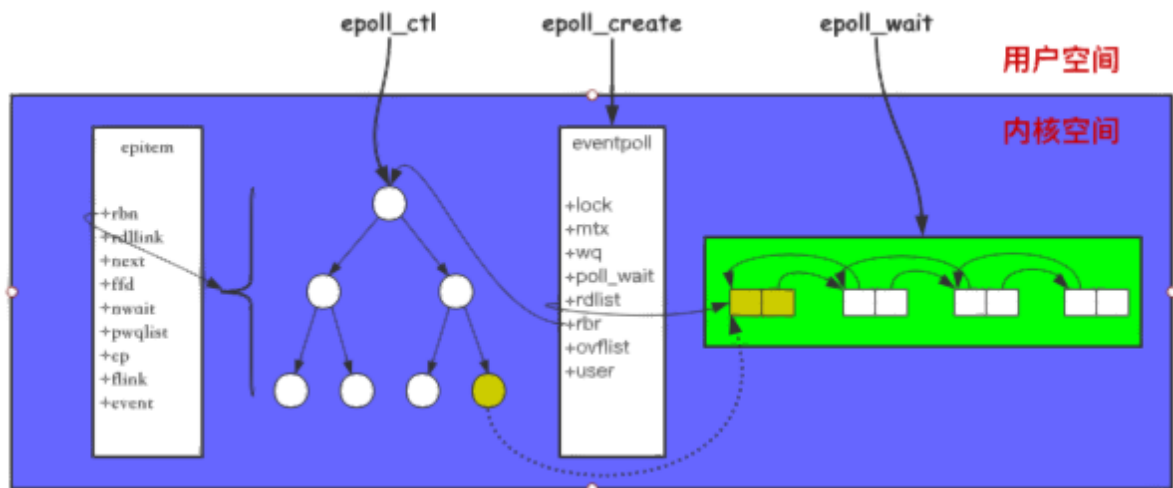
```

1 struct eventpoll {
2     // ...
3     struct rb_root rbr; // 管理 epoll 监听的事件
4     struct list_head rdllist; // 保存着 epoll_wait
    返回满足条件的事件
5     // ...
6 };
7
8 struct epitem {
9     // ...
10    struct rb_node rbn; // 红黑树节点
11    struct list_head rdllist; // 双向链表节点
12    struct epoll_filefd ffd; // 事件句柄信息
13    struct eventpoll *ep; // 指向所属的eventpoll对
    象
14    struct epoll_event event; // 注册的事件类型
15    // ...
16 };
17 struct epoll_event {
18     __uint32_t events; // epollin epollout
    epollet(边缘触发)
19     epoll_data_t data; // 保存 关联数据
20 };
  
```

```

21 typedef union epoll_data {
22     void *ptr;
23     int fd;
24     uint32_t u32;
25     uint64_t u64;
26 }epoll_data_t;
27
28 int epoll_create(int size);
29
30 /**
31     op:
32     EPOLL_CTL_ADD
33     EPOLL_CTL_MOD
34     EPOLL_CTL_DEL
35
36     event.events:
37     EPOLLIN          注册读事件
38     EPOLLOUT         注册写事件
39     EPOLLET          注册边缘触发模式，默认是水平触发
40 */
41 int epoll_ctl(int epfd, int op, int fd, struct
42 epoll_event* event);
43
44 /**
45     events[i].events:
46     EPOLLIN          触发读事件
47     EPOLLOUT         触发写事件
48     EPOLLERR         连接发生错误
49     EPOLLRDHUP       连接读端关闭
50     EPOLLHUP         连接双端关闭
51 */
52 int epoll_wait(int epfd, struct epoll_event*
53 events, int maxevents, int timeout);

```



调用 `epoll_create` 会创建一个 `epoll` 对象；调用 `epoll_ctl` 添加到 `epoll` 中的事件都会与网卡驱动程序建立回调关系，相应事件触发时会调用回调函数（`ep_poll_callback`），将触发的事件拷贝到 `rdlist` 双向链表中；调用 `epoll_wait` 将会把 `rdlist` 中就绪事件拷贝到用户态中；

## reactor 编程

### 连接建立

```

1 // 一、处理客户端的连接
2 // 1. 注册监听 listenfd 的读事件
3 struct epoll_event ev;
4 ev.events |= EPOLLIN;
5 epoll_ctl(efd, EPOLL_CTL_ADD, listenfd, &ev);
6 // 2. 当触发 listenfd 的读事件，调用 accept 接收新的连接
7 int clientfd = accept(listenfd, addr, sz);
8 struct epoll_event ev;
9 ev.events |= EPOLLIN;
10 epoll_ctl(efd, EPOLL_CTL_ADD, clientfd, &ev);
11 // 二、处理连接第三方服务
12 // 1. 创建 socket 建立连接
13 int connectfd = socket(AF_INET, SOCK_STREAM, 0);

```

```

14 connect(connectfd, (struct sockaddr *)&addr,
    sizeof(addr));
15 // 2. 注册监听 connectfd 的写事件
16 struct epoll_event ev;
17 ev.events |= EPOLLOUT;
18 epoll_ctl(efd, EPOLL_CTL_ADD, connectfd, &ev);
19 // 3. 当 connectfd 写事件被触发, 连接建立成功
20 if (status == e_connecting && e->events &
    EPOLLOUT) {
21     status == e_connected;
22     // 这里需要把写事件关闭
23     epoll_ctl(epfd, EPOLL_CTL_DEL, connectfd,
        NULL);
24 }

```

## 连接断开

```

1 if (e->events & EPOLLRDHUP) {
2     // 读端关闭
3     close_read(fd);
4     close(fd);
5 }
6 if (e->events & EPOLLRHUP) {
7     // 读写端都关闭
8     close(fd);
9 }

```

## 数据到达

```

1 // reactor 要用非阻塞io
2 // select
3 if (e->events & EPOLLIN) {
4     while (1) {
5         int n = read(fd, buf, sz);
6         if (n < 0) {
7             if (errno == EINTR)

```



```

8         continue;
9         if (errno == EWOULDBLOCK)
10            break;
11        close(fd);
12    } else if (n == 0) {
13        close_read(fd);
14        // close(fd);
15    }
16    // 业务逻辑了
17 }
18 }

```

## 数据发送完毕

```

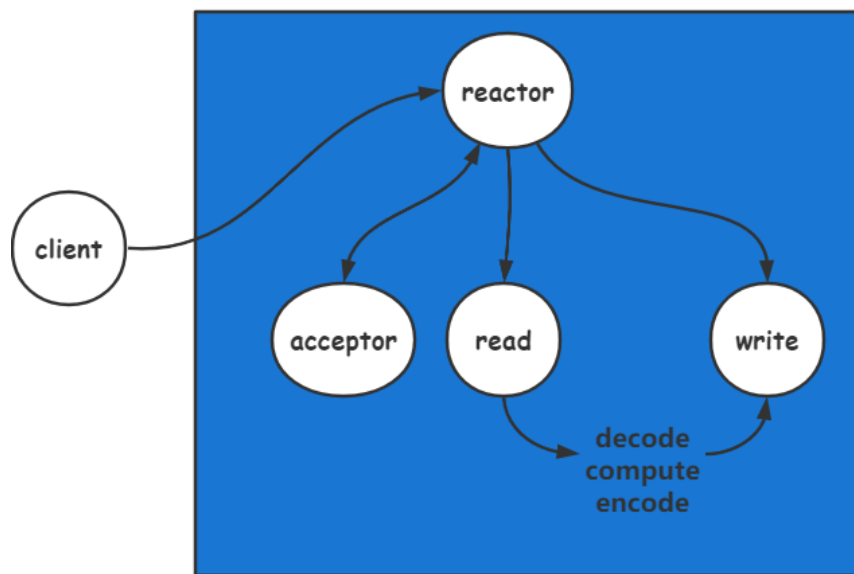
1  int n = write(fd, buf, dz);
2  if (n == -1) {
3      if (errno == EINTR)
4          continue;
5      if (errno == EWOULDBLOCK) {
6          struct epoll_event ev;
7          ev.events = EPOLLOUT;
8          epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
9          return;
10     }
11     close(fd);
12 }
13 // ...
14 if (e->events & EPOLLOUT) {
15     int n = write(fd, buf, sz);
16     //...
17     if (n == sz) {
18         epoll_ctl(epfd, EPOLL_CTL_DEL, fd, NULL);
19     }
20 }

```

## reactor应用

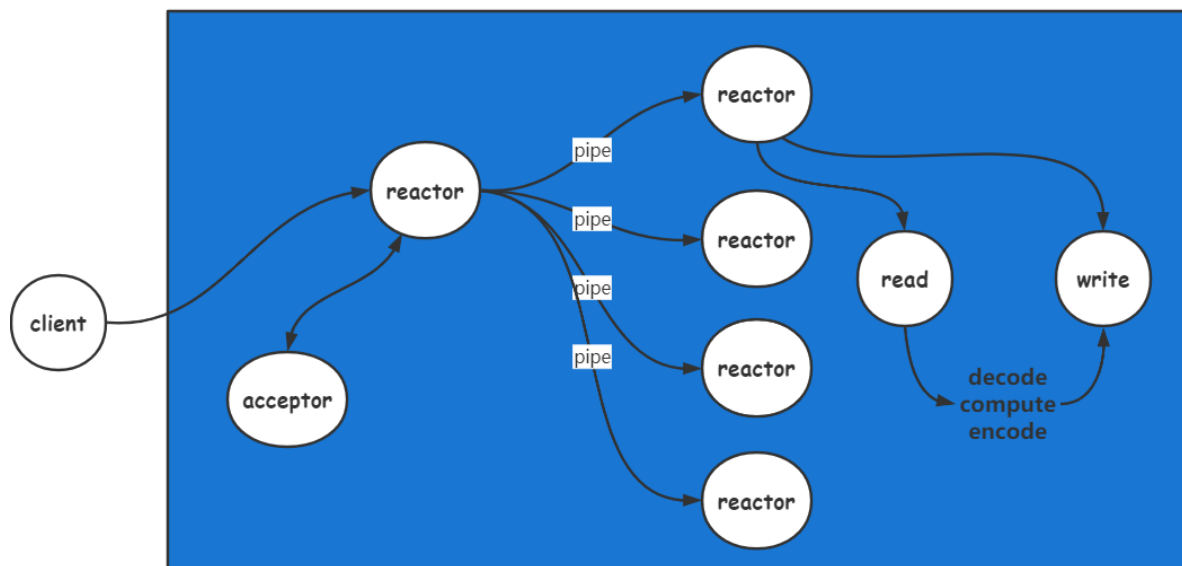
The reactor design pattern is an **event handling pattern** (事件处理模式) for handling service requests delivered concurrently to a service **handler by one or more inputs** (处理一个或多个并发传递到服务端的服务请求). The service handler then **demultiplexes** the incoming requests and **dispatches** them **synchronously** (同步) to the associated request handlers.

## 单 reactor



## 多 reactor (one eventloop per thread)

### 多线程



## 多进程

