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；

```

int clientfd = accept(listenfd, addr, sz);

// 举例为非阻塞io，阻塞io成功直接返回0；
int connectfd = socket(AF_INET, SOCK_STREAM, 0);
int ret = connect(connectfd, (struct sockaddr *)&addr, sizeof(addr));
// ret == -1 && errno == EINPROGRESS 正在建立连接
// 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；

```
// 主动关闭
close(fd);
shutdown(fd, SHUT_RDWR);
// 主动关闭本地读端，对端写段关闭
shutdown(fd, SHUT_RD);
// 主动关闭本地写端，对端读段关闭
shutdown(fd, SHUT_WR);

// 被动：读端关闭
// 有的网络编程需要支持半关闭状态
int n = read(fd, buf, sz);
if (n == 0) {
    close_read(fd);
    // write()
    // close(fd);
}
// 被动：写端关闭
int n = write(fd, buf, sz);
if (n == -1 && errno == EPIPE) {
    close_write(fd);
    // close(fd);
}
```

消息的到达

> 从读缓冲区中读取数据；

```c
int n = read(fd, buf, sz);
if (n < 0) { // n == -1
    if (errno == EINTR || errno == EWOULDBLOCK)
        break;
    close(fd);
} else if (n == 0) {
    close(fd);
} else {
    // 处理 buf
}
```

消息发送完毕

> 往写缓冲区中写数据；

```c
int n = write(fd, buf, dz);
if (n == -1) {
    if (errno == EINTR || errno == EWOULDBLOCK) {
        return;
    }
    close(fd);
}
```

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；

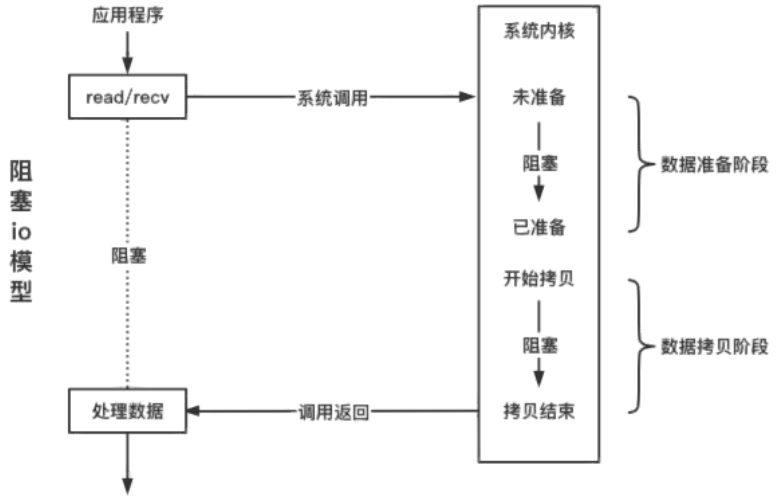
操作 IO

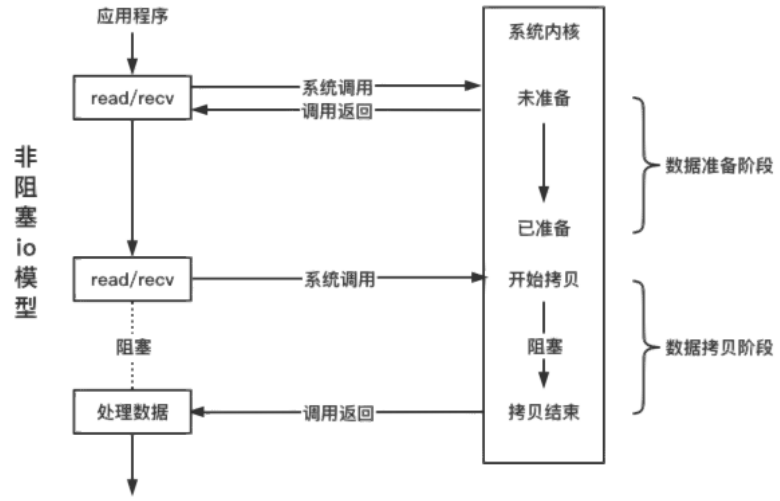
> 只能使用 io 函数来进行操作；分为两种操作方式：阻塞 io 和非阻塞 io；

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；

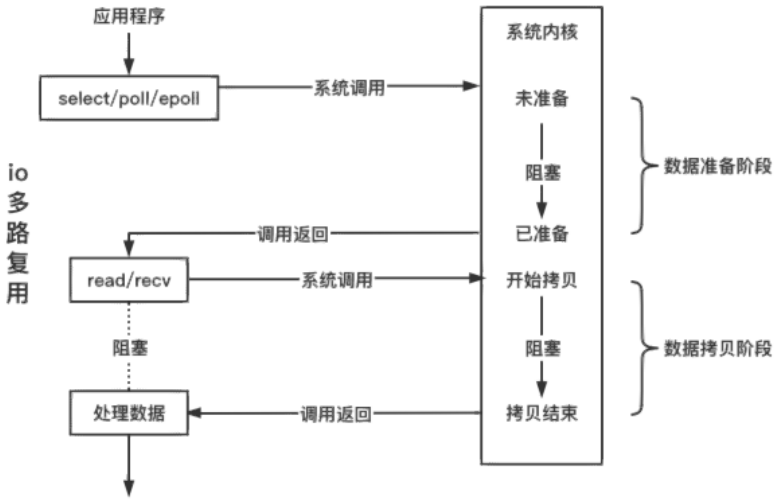
```c
// 默认情况下，fd 是阻塞的，设置非阻塞的方法如下：
int flag = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flag | O_NONBLOCK);
```





IO 多路复用

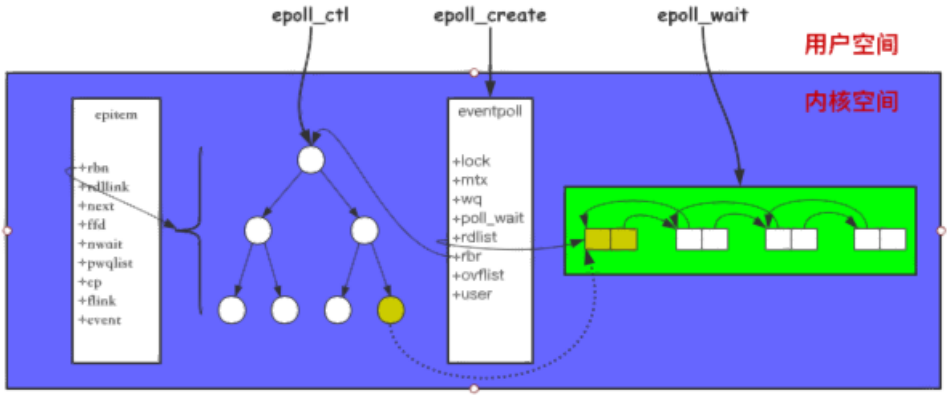
> io 多路复用只负责检测io，不负责操作io；
>
> `int n = epoll_wait(epfd, evs, sz, timeout);`
>
> `timeout = -1` 一直阻塞直到网络事件到达；
>
> `imeout = 0` 不管是否有事件就绪立刻返回；
>
> `timeout = 1000` 最多等待 1 s，如果1 s内没有事件触发则返回；



epoll

结构以及接口

```
struct eventpoll {
    // ...
    struct rb_root rbr; // 管理 epoll 监听的事件
    struct list_head rdllist; // 保存着 epoll_wait 返回满足条件的事件
    // ...
};

struct epitem {
    // ...
    struct rb_node rbn; // 红黑树节点
    struct list_head rdllist; // 双向链表节点
    struct epoll_filefd ffd; // 事件句柄信息
    struct eventpoll *ep; // 指向所属的eventpoll对象
    struct epoll_event event; // 注册的事件类型
    // ...
};
struct epoll_event {
    __uint32_t events; // epollin epollout epollel(边缘触发)
    epoll_data_t data; // 保存 关联数据
};
typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
}epoll_data_t;

int epoll_create(int size);

/**
    op:
    EPOLL_CTL_ADD
    EPOLL_CTL_MOD
    EPOLL_CTL_DEL

    event.events:
    EPOLLIN         注册读事件
    EPOLLOUT        注册写事件
    EPOLLET         注册边缘触发模式，默认是水平触发
*/
int epoll_ctl(int epfd, int op, int fd, struct epoll_event* event);

/**
    events[i].events:
    EPOLLIN         触发读事件
    EPOLLOUT        触发写事件
    EPOLLERR        连接发生错误
    EPOLLRDHUP      连接读端关闭
    EPOLLHUP        连接双端关闭
*/
int epoll_wait(int epfd, struct epoll_event* 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；

epoll 编程

连接建立

```c
// 一、处理客户端的连接
// 1. 注册监听 listenfd 的读事件
struct epoll_event ev;
ev.events |= EPOLLIN;
epoll_ctl(efd, EPOLL_CTL_ADD, listenfd, &ev);
// 2. 当触发 listenfd 的读事件，调用 accept 接收新的连接
int clientfd = accept(listenfd, addr, sz);
struct epoll_event ev;
ev.events |= EPOLLIN;
epoll_ctl(efd, EPOLL_CTL_ADD, clientfd, &ev);
// 二、处理连接第三方服务
// 1. 创建 socket 建立连接
int connectfd = socket(AF_INET, SOCK_STREAM, 0);
connect(connectfd, (struct sockaddr *)&addr, sizeof(addr));
// 2. 注册监听 connectfd 的写事件
struct epoll_event ev;
ev.events |= EPOLLOUT;
epoll_ctl(efd, EPOLL_CTL_ADD, connectfd, &ev);
// 3. 当 connectfd 写事件被触发，连接建立成功
if (status == e_connecting && e->events & EPOLLOUT) {
    status == e_connected;
    // 这里需要把写事件关闭
    epoll_ctl(epfd, EPOLL_CTL_DEL, connectfd, NULL);
}
```

连接断开

```c
if (e->events & EPOLLRDHUP) {
    // 读端关闭
    close_read(fd);
    close(fd);
}
if (e->events & EPOLLHUP) {
    // 读写端都关闭
    close(fd);
}
```

数据到达

```c
 // reactor 要用非阻塞io
// select
if (e->events & EPOLLIN) {
    while (1) {
        int n = read(fd, buf, sz);
        if (n < 0) {
            if (errno == EINTR)
                continue;
            if (errno == EWOULDBLOCK)
                break;
            close(fd);
        } else if (n == 0) {
            close_read(fd);
            // close(fd);
        }
        // 业务逻辑了
    }
}
```

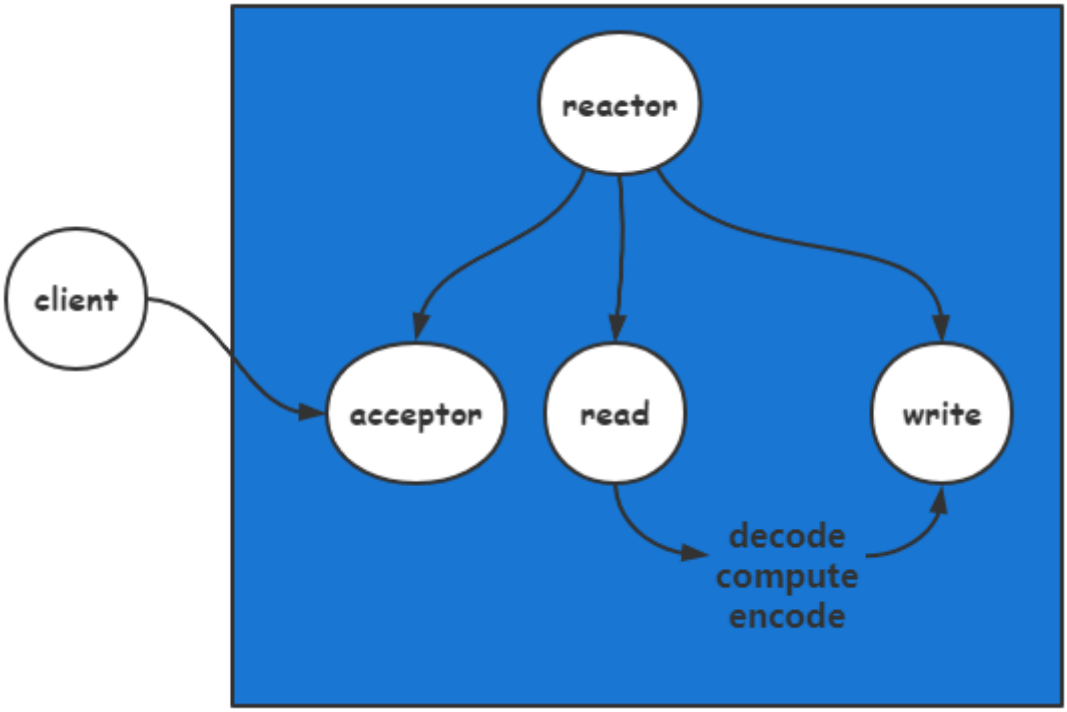
数据发送完毕

```c
int n = write(fd, buf, dz);
if (n == -1) {
    if (errno == EINTR)
        continue;
    if (errno == EWOULDBLOCK) {
        struct epoll_event ev;
        ev.events = EPOLLOUT;
        epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
    }
    close(fd);
}
// ...
if (e->events & EPOLLOUT) {
    int n = write(fd, buf, sz);
    //...
    if (n > 0) {
        epoll_ctl(epfd, EPOLL_CTL_DEL, fd, NULL);
    }
}
```

reactor应用

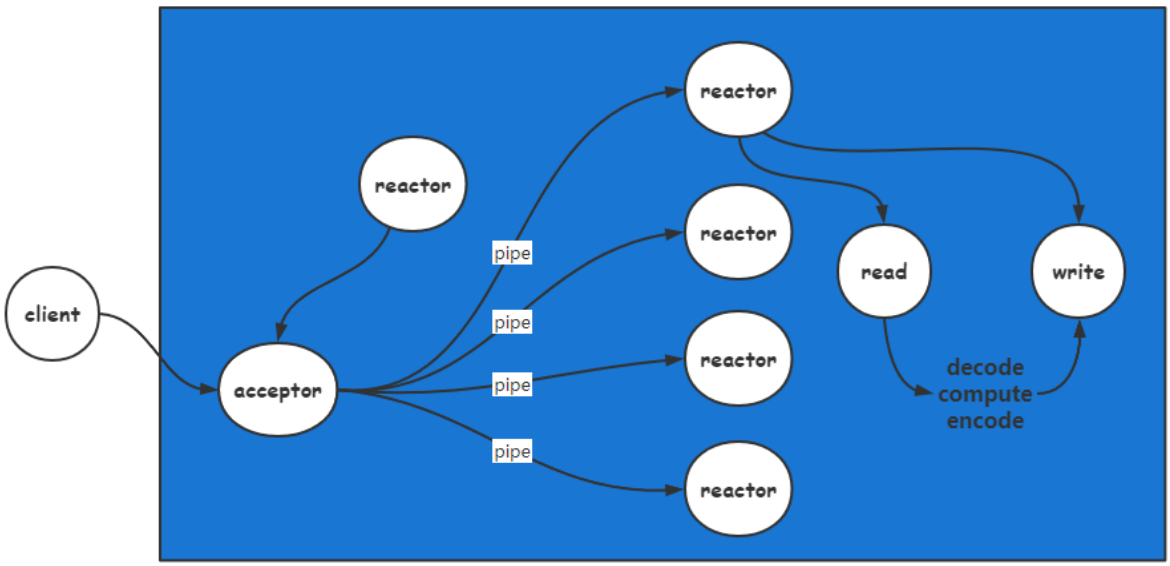
> The reactor design pattern is an **event handling pattern** （事件处理模式）for handling service requests delivered concurrently to a service **handler by one or more inputs**（处理一个或多个并发传递到服务端的服务请求）. The service handler then **demultiplexes** the incoming requests and **dispatches** them **synchronously** （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

单 reactor

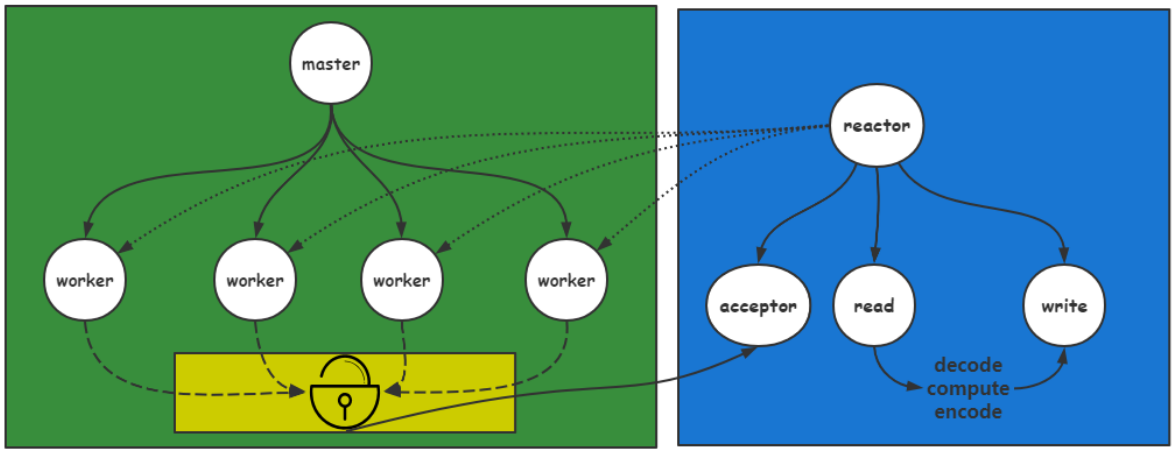


多 reactor （one eventloop per thread）

多线程



多进程


master
worker
worker
worker
worker
reactor
acceptor
read
write
decode
compute
encode