零声学院 Mark 老师 QQ : 2548898954

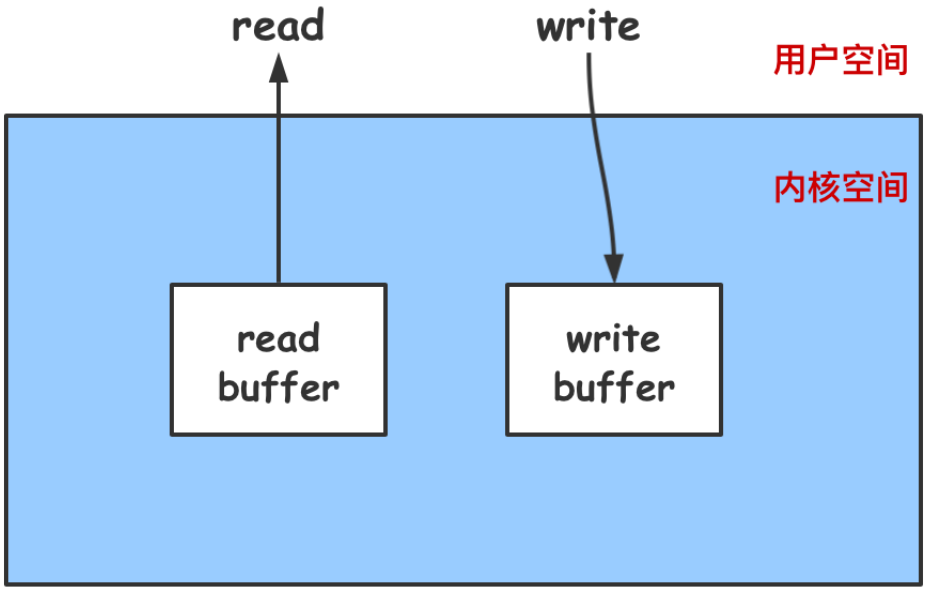
网络编程关注的问题

连接建立

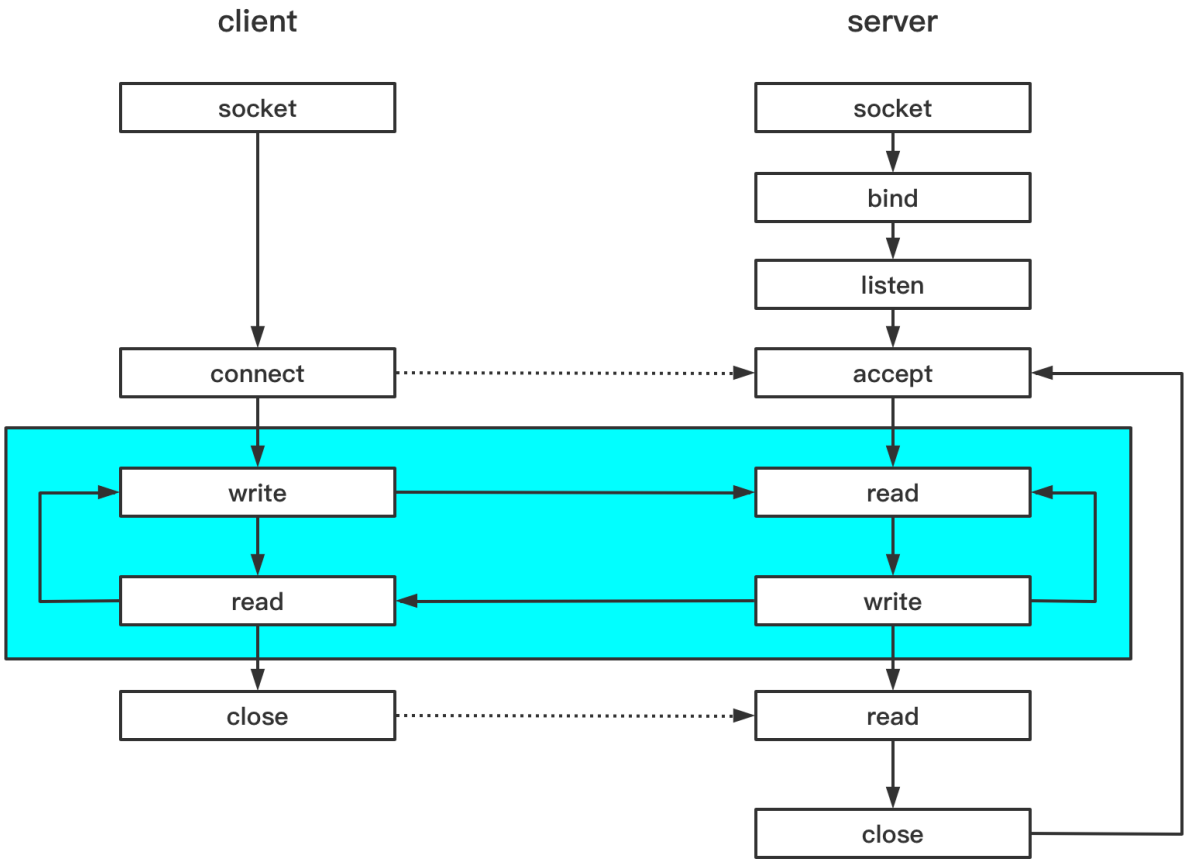
连接断开

消息到达

消息发送完毕



网络编程流程



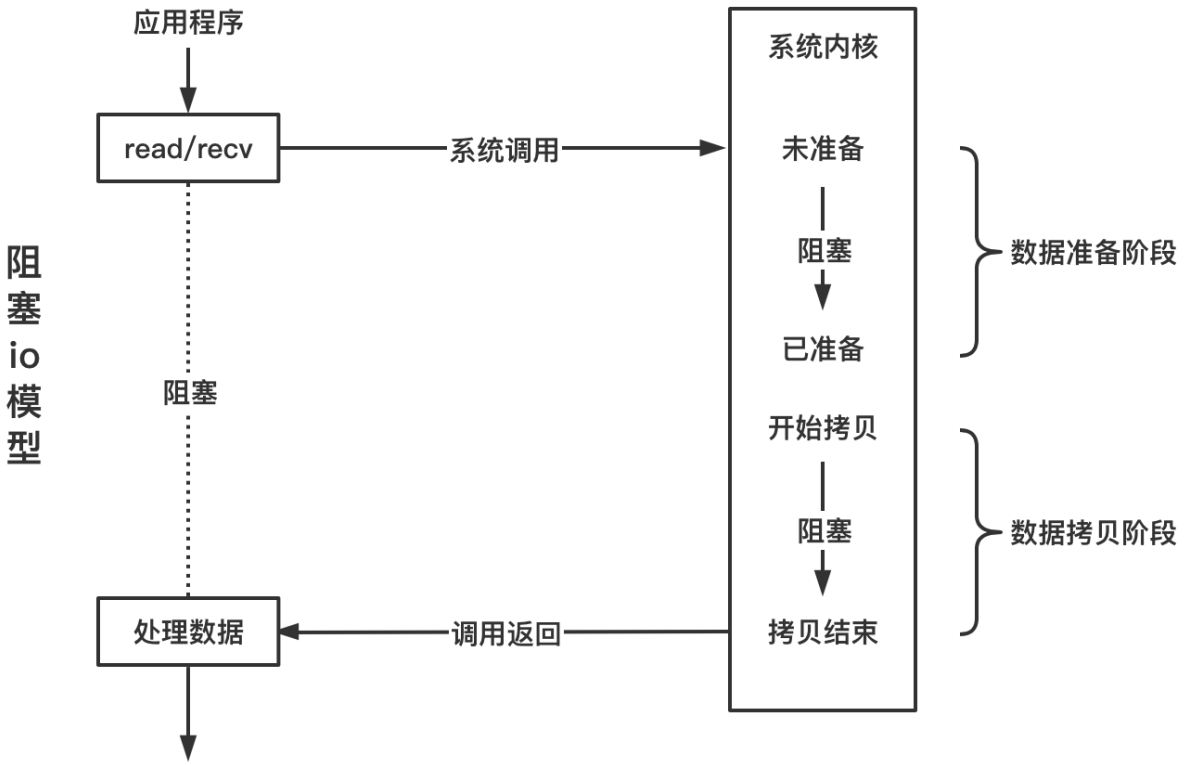
阻塞io模型和非阻塞io模型

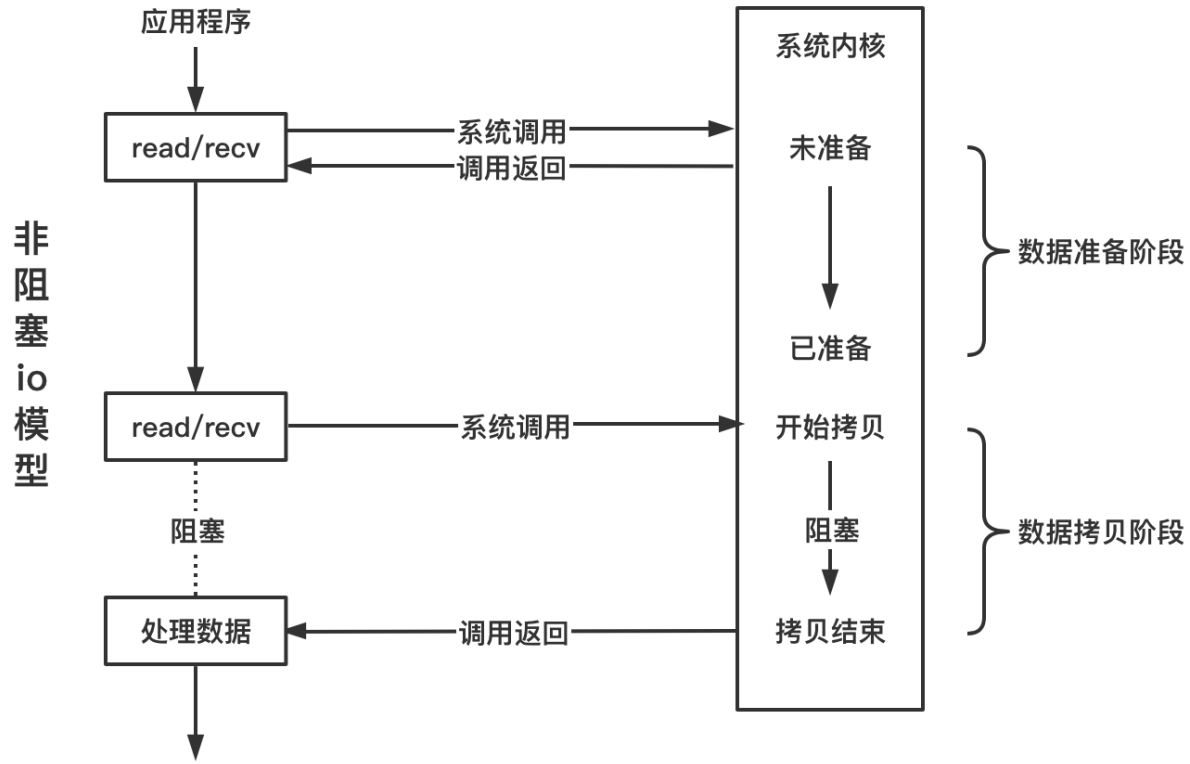
- 阻塞在哪里？ 阻塞在网络线程
- 什么来决定阻塞还是非阻塞?

```
连接的fd
fcntl(c->fd, F_SETFL, O_NONBLOCK);
```

- 具体的差异?

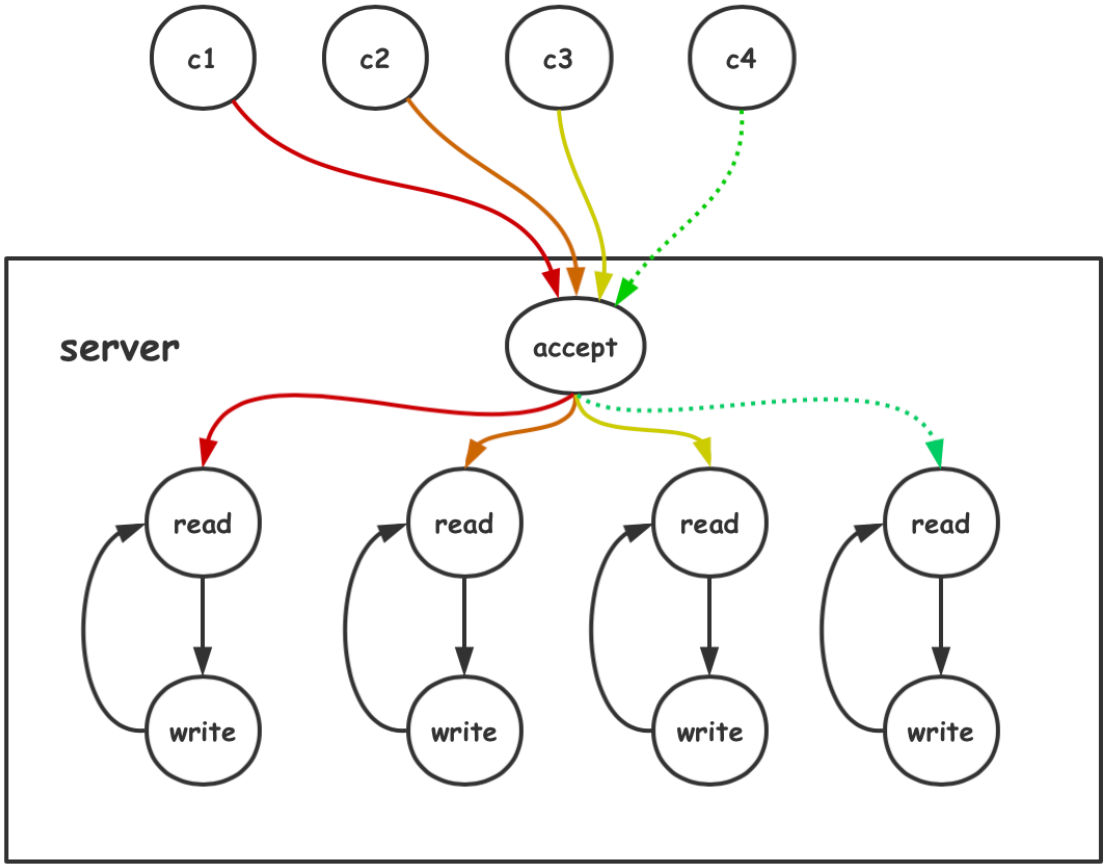
> io函数在数据未到达时是否立刻返回





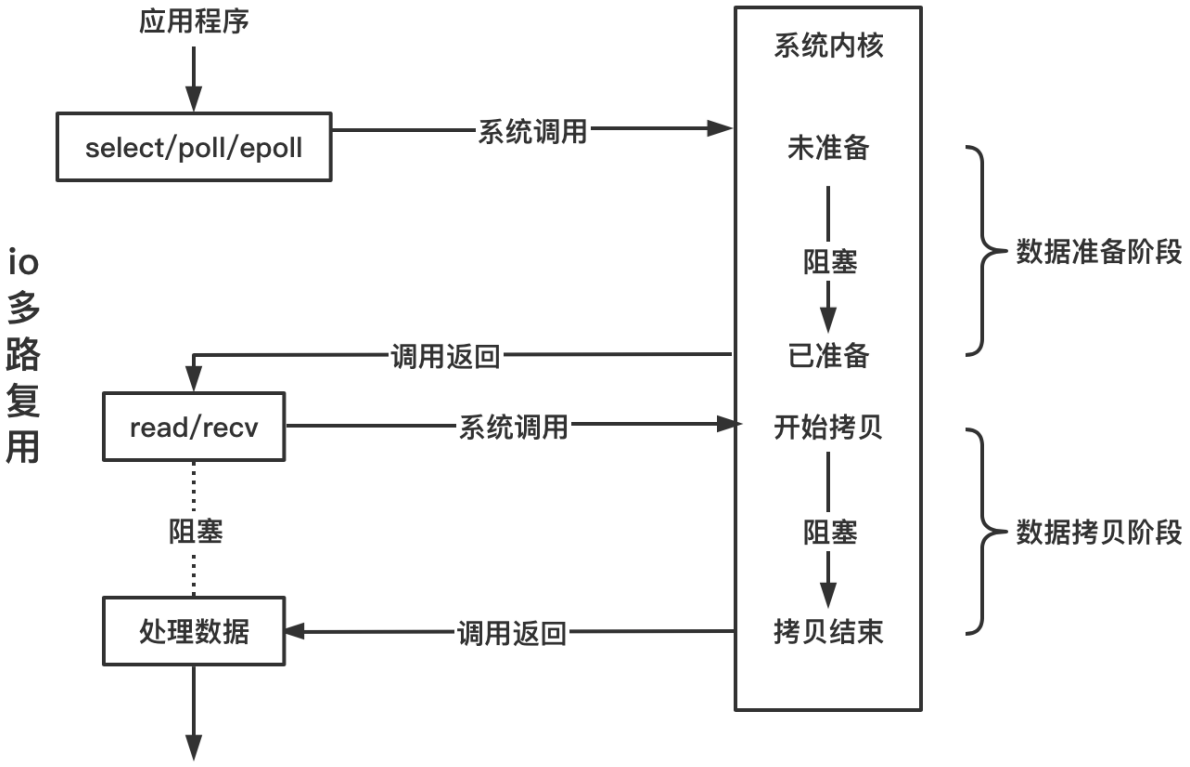
阻塞io模型 + 多线程

> 每一个线程处理一个 fd 连接 bio
>
> 优点：处理及时
>
> 缺点：线程利用率很低，线程的数量是有限的



io多路复用（网络线程）

> 用一个线程来检测多个io事件



> 水平触发的时候，io函数既可以是阻塞的也可以是非阻塞的。

> 边缘触发的时候，io函数只能是非阻塞的。

epoll基础

重要数据结构

```
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;
    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;
```

主要函数

- epoll_create系统调用

```
int epoll_create(int size);
```

> size参数告诉内核这个epoll对象会处理的事件大致数量，而不是能够处理的事件的最大数。
>
> 在现在linux版本中，这个size参数已经没有意义了；
>
> 返回：epoll对象句柄；之后针对该epoll的操作需要通过该句柄来标识该epoll对象；

- epoll_ctl系统调用

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event* event);
```

> epoll_ctl向epoll对象添加、修改或删除事件；
>
> 返回：0表示成功，-1表示错误，根据errno错误码判断错误类型。

op类型：

> EPOLL_CTL_ADD　　添加新的事件到epoll中
>
> EPOLL_CTL_MOD　　修改epoll中的事件
>
> EPOLL_CTL_DEL　　删除epoll中的事件

event.events 取值：

> EPOLLIN　　表示该连接上有数据可读（tcp连接远端主动关闭连接，也是可读事件，因为需要处理发送来的FIN包；FIN包就是read 返回 0）
>
> EPOLLOUT　　表示该连接上可写发送（主动向上游服务器发起非阻塞tcp连接，连接建立成功事件相当于可写事件）
>
> EPOLLRDHUP　　表示tcp连接的远端关闭或半关闭连接
>
> EPOLLPRI　　表示连接上有紧急数据需要读
>
> EPOLLERR　　表示连接发生错误
>
> EPOLLHUP　　表示连接被挂起
>
> EPOLLET　　将触发方式设置为边缘触发，系统默认为水平触发
>
> EPOLLONESHOT　　表示该事件只处理一次，下次需要处理时需重新加入epoll

- epoll_wait系统调用

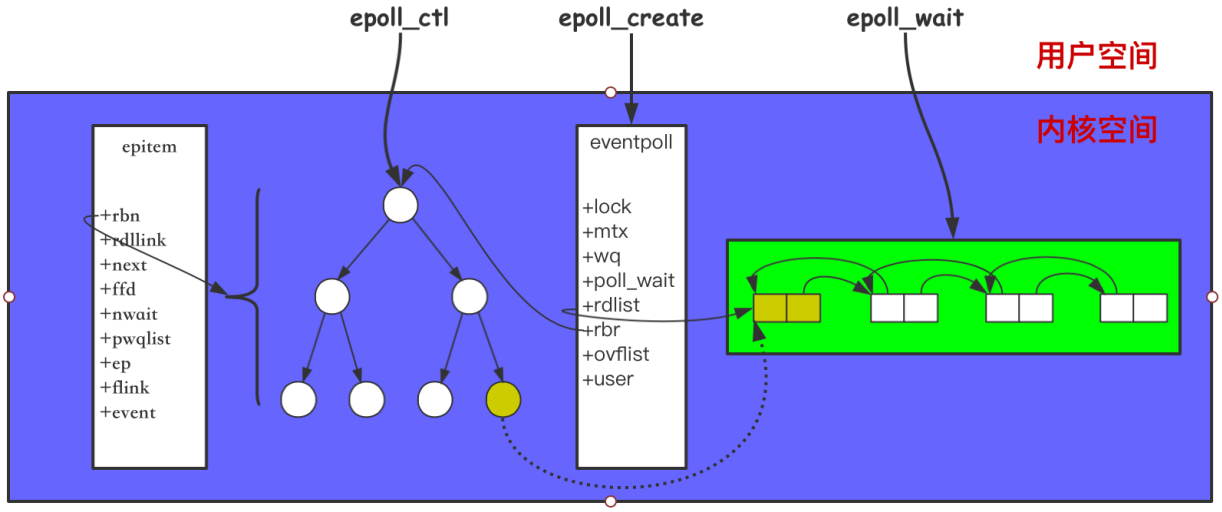
```
int epoll_wait(int epfd, struct epoll_event* events, int maxevents, int timeout);
```

> 收集 epoll 监控的事件中已经发生的事件，如果 epoll 中没有任何一个事件发生，则最多等待 timeout 毫秒后返回。
>
> 返回：表示当前发生的事件个数
>
> 返回0表示本次没有事件发生；
>
> 返回-1表示出现错误，需要检查errno错误码判断错误类型。

注意：

> events 这个数组必须在用户态分配内存，内核负责把就绪事件复制到该数组中；
>
> maxevents 表示本次可以返回的最大事件数目，一般设置为 events 数组的长度；
>
> timeout表示在没有检测到事件发生时最多等待的时间；如果设置为0，检测到rdllist为空立刻返回；如果设置为-1，一直等待；

原理图



要点

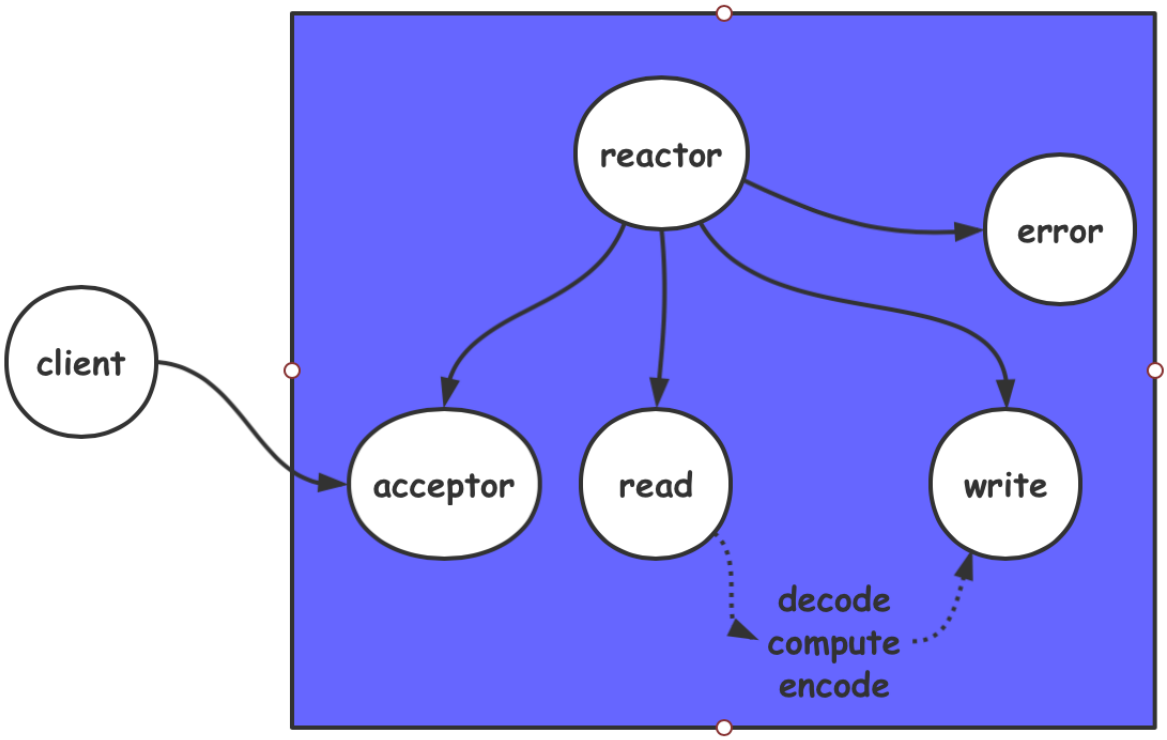
> 所有添加到epoll中的事件都会与网卡驱动程序建立回调关系，相应的事件发生时会调用这里的回调方法（ep_poll_callback）,它会把这样的事件放在rdllist双向链表中。

reactor模型

- 组成：非阻塞的io + io多路复用；
- 特征：基于事件循环，以事件驱动或者事件回调的方式来实现业务逻辑；
- 表述：将连接的io处理转化为事件处理；

单reactor模型

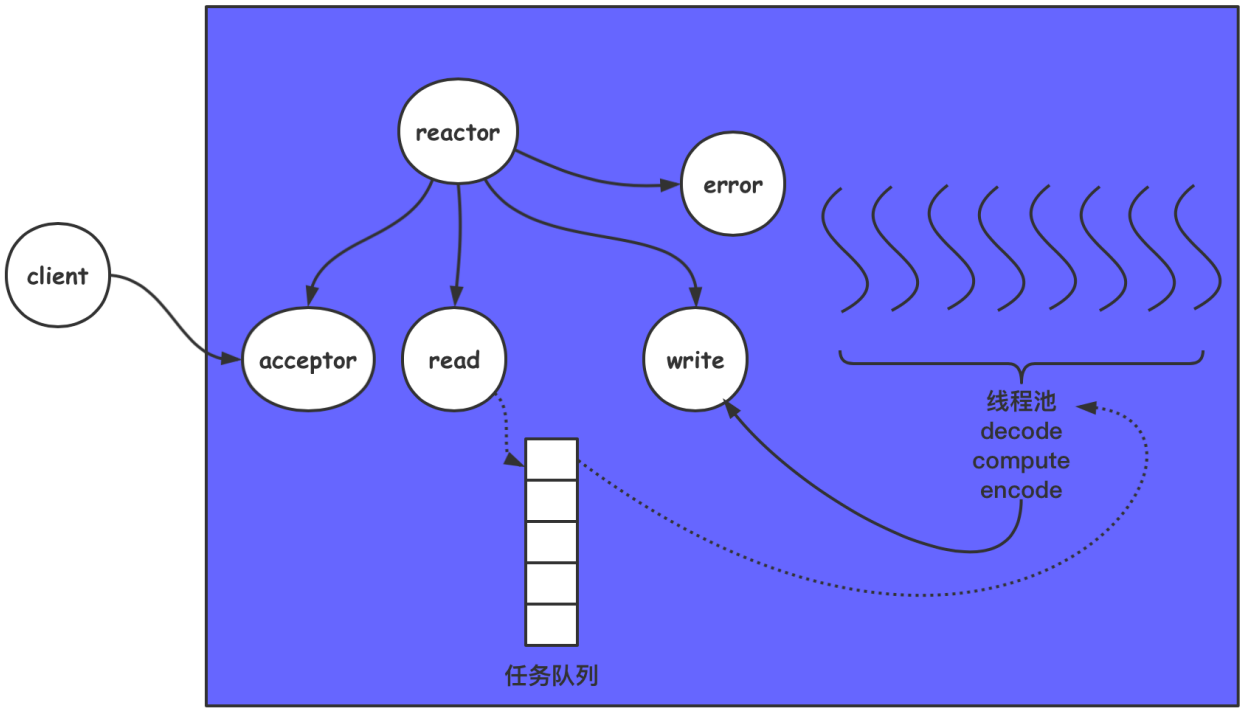
原理图



> 代表：redis 内存数据库 操作redis当中的数据结构
>
> redis 6.0 多线程

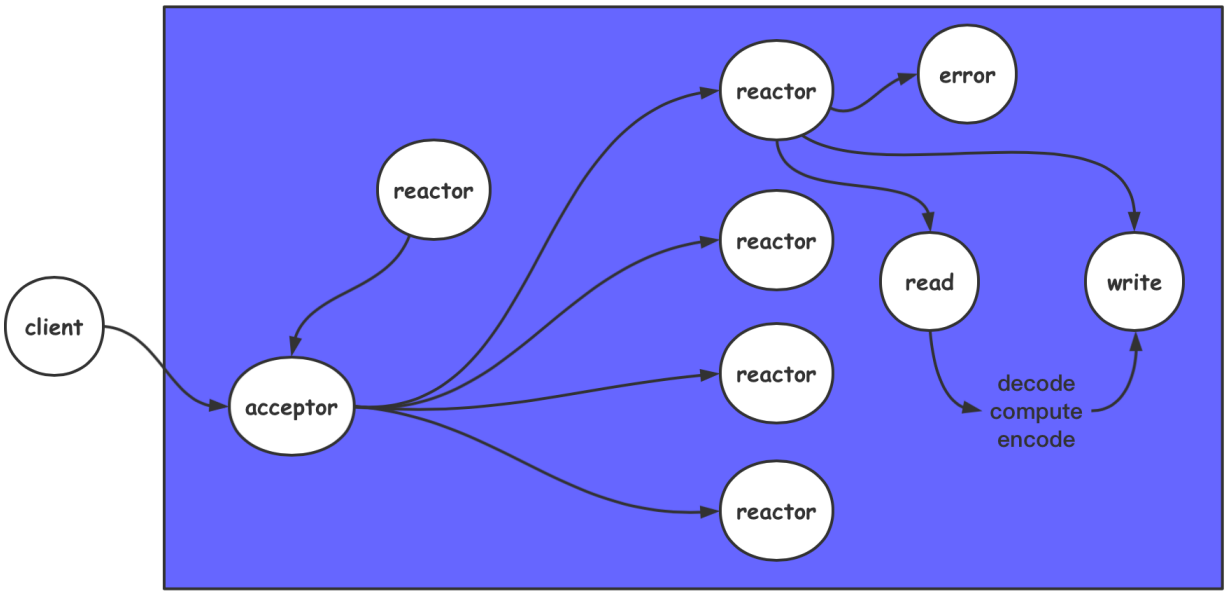
单reactor模型 + 任务队列 + 线程池

原理图



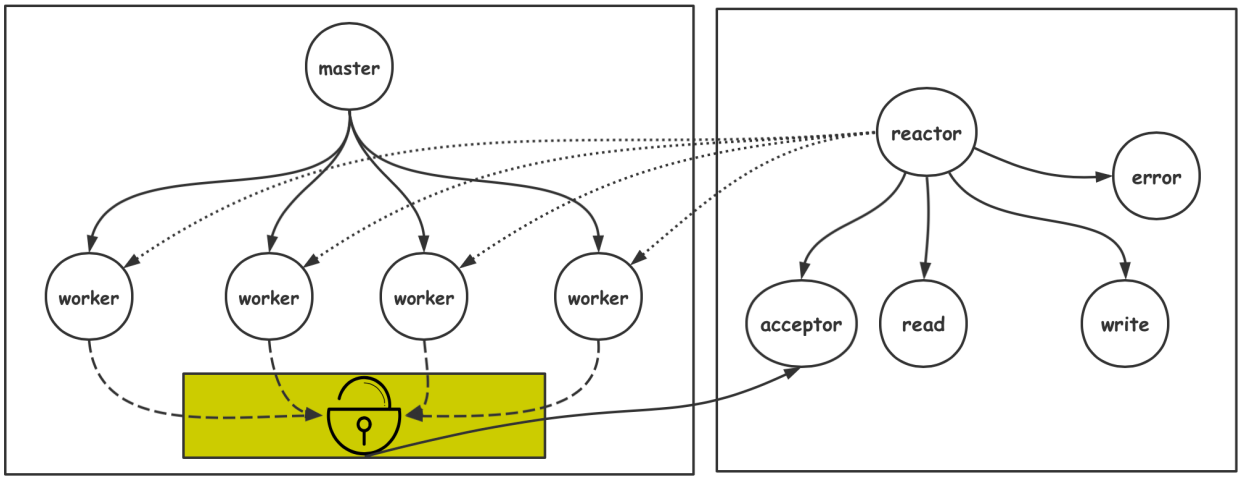
> 代表：skynet

多reactor



> 应用： memcached accept(fd, backlog) one eventloop per thread

- **多进程**

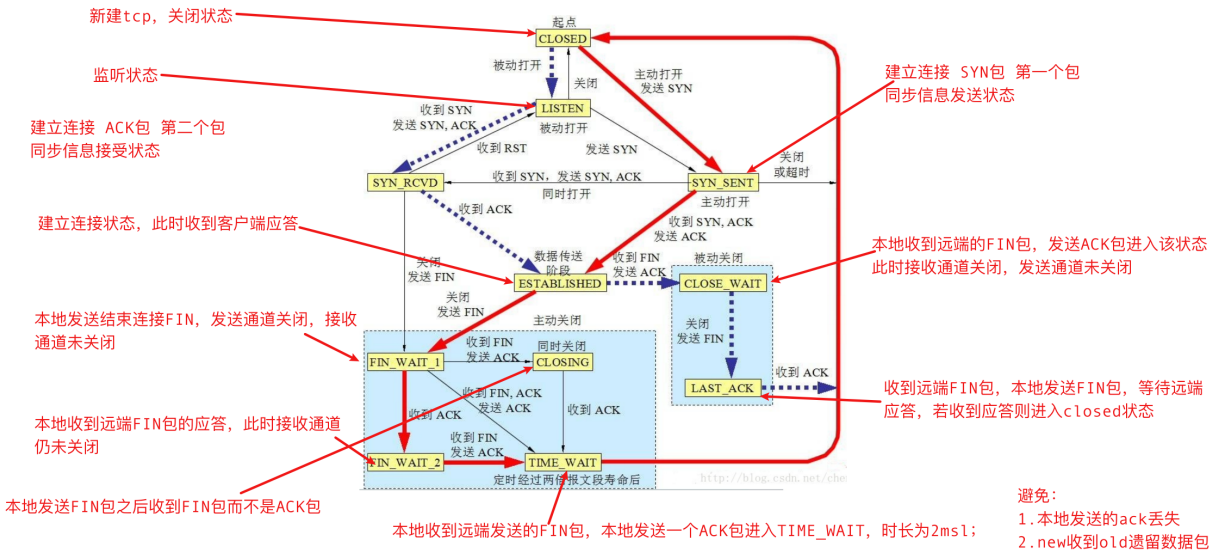


> 应用：nginx

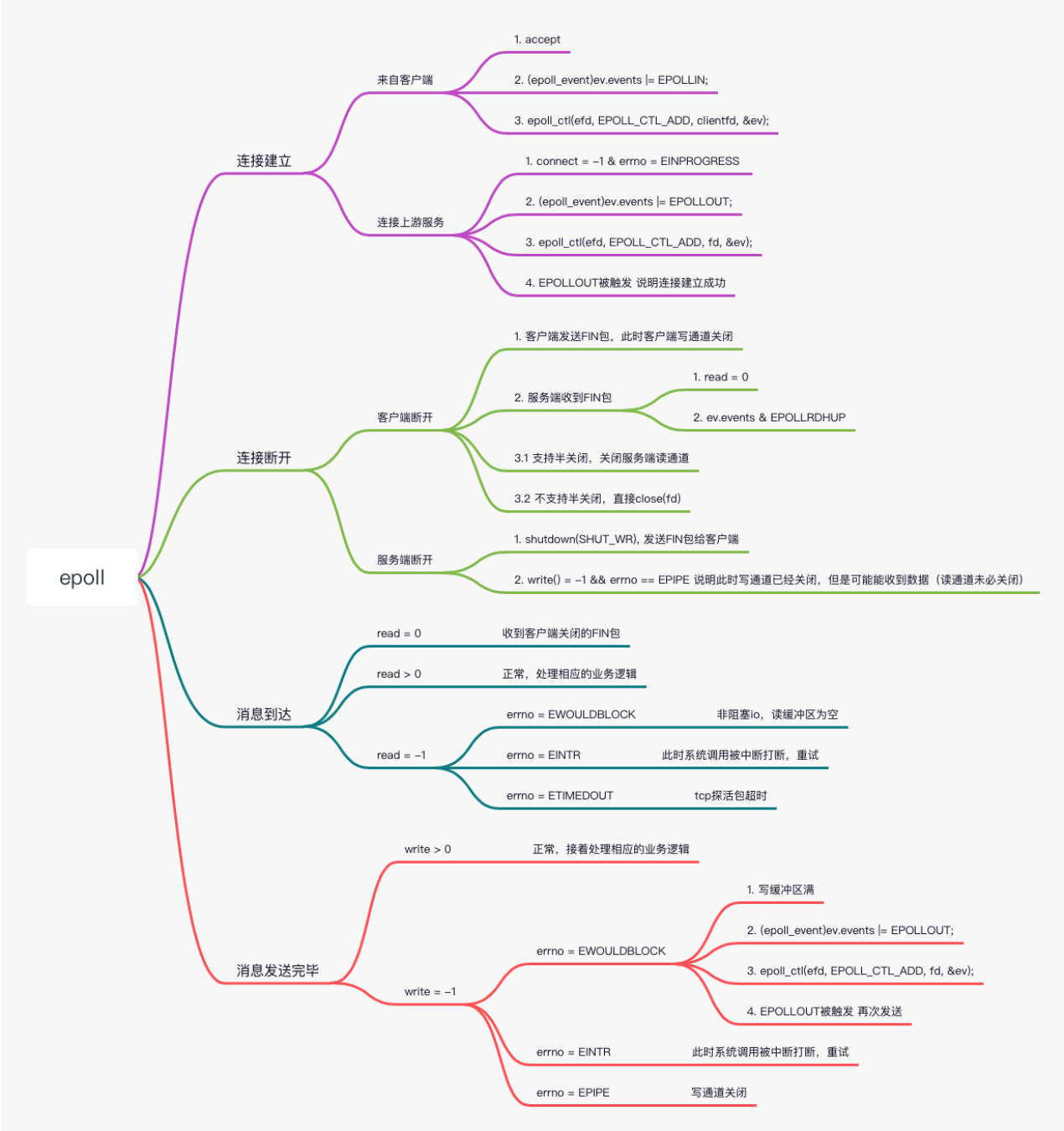
多reactor + 消息队列 + 线程池

> 业务场景中比较多 网络密集型 + 业务密集型

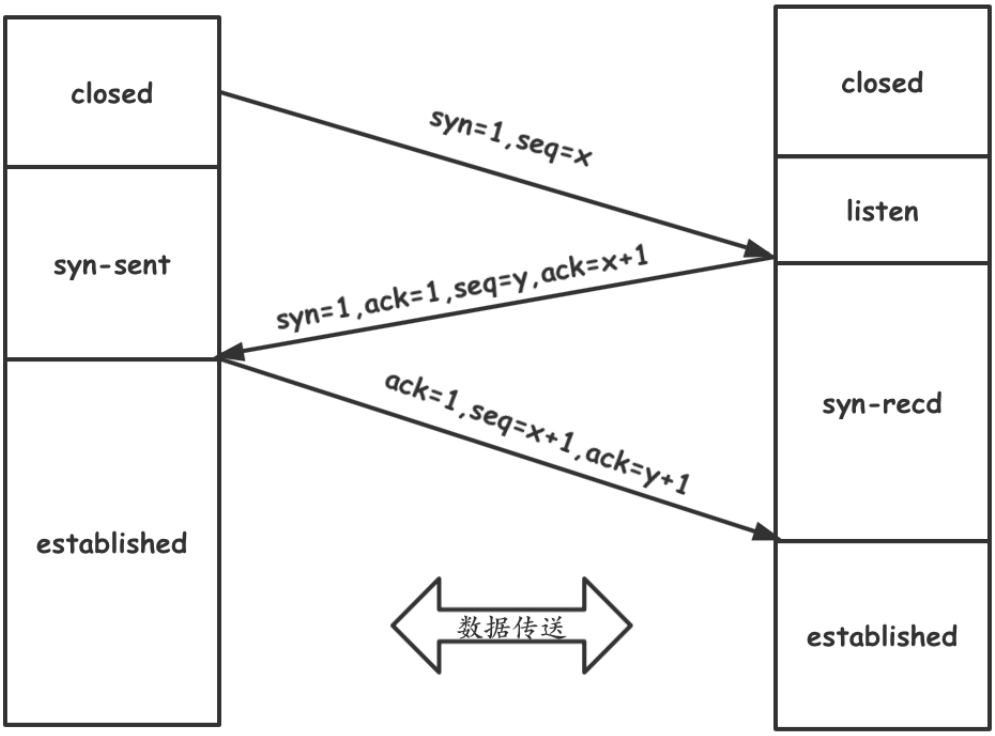
TCP状态转换图


新建tcp，关闭状态
监听状态
建立连接 ACK包 第二个包
同步信息接受状态
建立连接状态，此时收到客户端应答
本地发送结束连接FIN，发送通道关闭，接收
通道未关闭
本地收到远端FIN包的应答，此时接收通道
仍未关闭
本地发送FIN包之后收到FIN包而不是ACK包
建立连接 SYN包 第一个包
同步信息发送状态
本地收到远端的FIN包，发送ACK包进入该状态
此时接收通道关闭，发送通道未关闭
收到远端FIN包，本地发送FIN包，等待远端
应答，若收到应答则进入closed状态
本地收到远端发送的FIN包，本地发送一个ACK包进入TIME_WAIT，时长为2msl；
避免：
1.本地发送的ack丢失
2.new收到old遗留数据包
起点
CLOSED
被动打开
关闭
主动打开
发送 SYN
LISTEN
收到 SYN
发送 SYN, ACK
被动打开
收到 RST
发送 SYN
SYN_RCVD
收到 SYN，发送 SYN, ACK
同时打开
SYN_SENT
关闭
或超时
主动打开
收到 ACK
收到 SYN, ACK
发送 ACK
关闭
发送 FIN
数据传送
阶段
ESTABLISHED
收到 FIN
发送 ACK
被动关闭
CLOSE_WAIT
关闭
发送 FIN
关闭
发送 FIN
主动关闭
收到 FIN
发送 ACK
同时关闭
FIN_WAIT_1
CLOSING
收到 FIN, ACK
发送 ACK
收到 ACK
收到 ACK
LAST_ACK
收到 ACK
收到 FIN
发送 ACK
FIN_WAIT_2
TIME_WAIT
定时经过两倍报文段寿命后

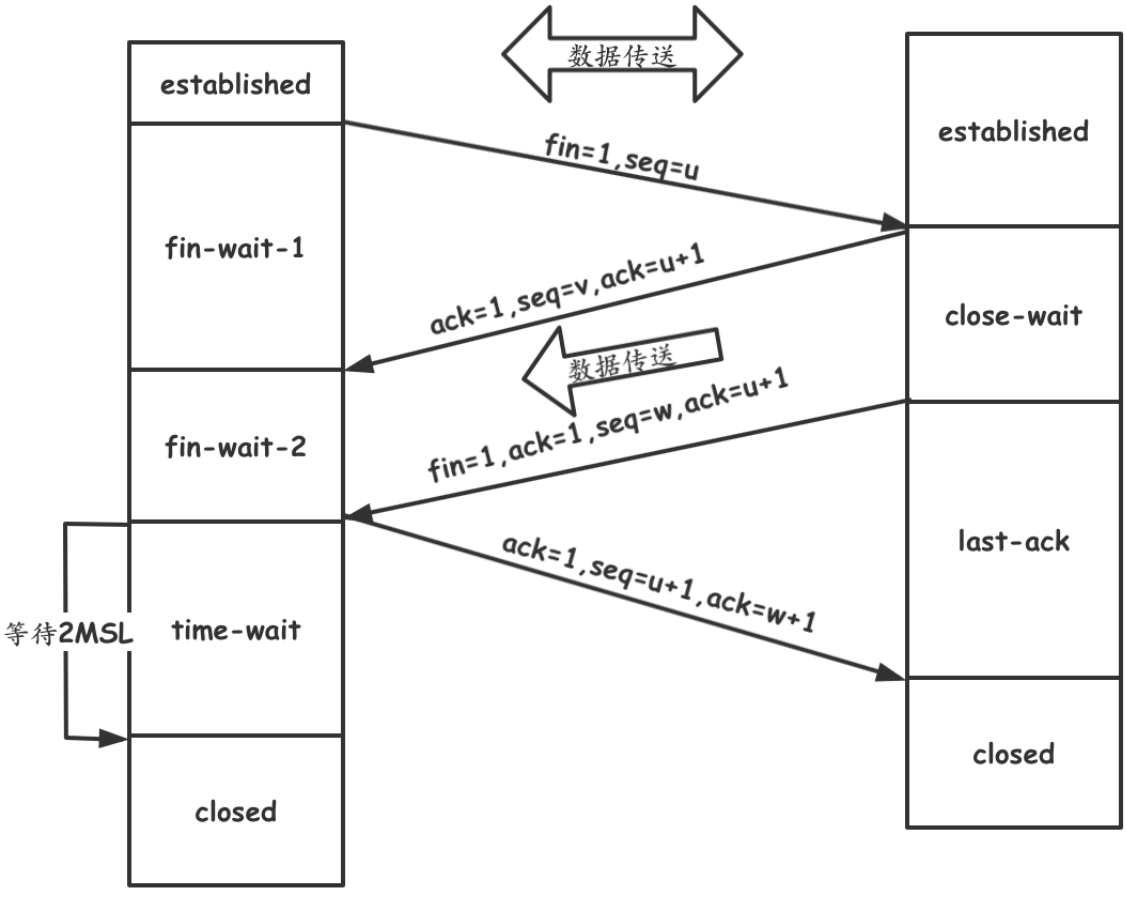
epoll处理细节图


epoll
连接建立
来自客户端
1. accept
2. (epoll_event)ev.events |= EPOLLIN;
3. epoll_ctl(efd, EPOLL_CTL_ADD, clientfd, &ev);
连接上游服务
1. connect = –1 & errno = EINPROGRESS
2. (epoll_event)ev.events |= EPOLLOUT;
3. epoll_ctl(efd, EPOLL_CTL_ADD, fd, &ev);
4. EPOLLOUT被触发 说明连接建立成功
连接断开
客户端断开
1. 客户端发送FIN包，此时客户端写通道关闭
2. 服务端收到FIN包
1. read = 0
2. ev.events & EPOLLRDHUP
3.1 支持半关闭，关闭服务端读通道
3.2 不支持半关闭，直接close(fd)
服务端断开
1. shutdown(SHUT_WR), 发送FIN包给客户端
2. write() = –1 && errno == EPIPE 说明此时写通道已经关闭，但是可能能收到数据（读通道未必关闭）
消息到达
read = 0
收到客户端关闭的FIN包
read > 0
正常，处理相应的业务逻辑
read = –1
errno = EWOULDBLOCK
非阻塞io，读缓冲区为空
errno = EINTR
此时系统调用被中断打断，重试
errno = ETIMEDOUT
tcp探活包超时
消息发送完毕
write > 0
正常，接着处理相应的业务逻辑
write = –1
errno = EWOULDBLOCK
1. 写缓冲区满
2. (epoll_event)ev.events |= EPOLLOUT;
3. epoll_ctl(efd, EPOLL_CTL_ADD, fd, &ev);
4. EPOLLOUT被触发 再次发送
errno = EINTR
此时系统调用被中断打断，重试
errno = EPIPE
写通道关闭

三次握手


closed
syn=1,seq=x
closed
listen
syn-sent
syn=1,ack=1,seq=y,ack=x+1
syn-recd
ack=1,seq=x+1,ack=y+1
established
数据传送
established

四次挥手


数据传送
established
fin=1,seq=u
established
fin-wait-1
ack=1,seq=v,ack=u+1
close-wait
数据传送
fin-wait-2
fin=1,ack=1,seq=w,ack=u+1
last-ack
等待2MSL
time-wait
ack=1,seq=u+1,ack=w+1
closed
closed

半关闭状态

背景

> 客户端关闭写通道，此时服务端想要推送完所有数据后再关闭； 推送系统中 close_read() close_write() close()
>
> 服务端 close_read send send ... close
>
> 客户端 close();

实现

> 需要实现半关闭状态；close-wait阶段；必须要收到ack包，才能知道客户端收到了我们推送的所有数据；

细节

> 发送端：
>
> shutdown(SHUT_WR) 发送一个 FIN 包，并且标记该 socket 为 SEND_SHUTDOWN；
>
> shutdown(SHUT_RD) 不发送任何包，但是标记该 socket 为 RCV_SHUTDOWN；
>
> 接收端：
>
> 收到 FIN 包标记该 socket 为RCV_SHUTDOWN；
>
> 对于epoll而言，如果一个socket同时标记为 SEND_SHUTDOWN 和 RCV_SHUTDOWN；那么poll会返回 EPOLLHUP；
>
> 如果一个socket被标记为 RCV_SHUTDOWN；poll会返回 EPOLLRDHUP；
>
> 应用：skynet 支持半关闭状态

tcp-keepalive

背景

> tcp是面向连接的，一般情况下，两端应用可以通过发送接收数据得知对端的存活；当两端都没有数据的时候，如何判断连接是否正常？系统默认keepalive是关闭的，当keepalive开启时，可以保持连接检测对方主机是否崩溃；

属性

> 1. tcp_keepalive_time 两端多久没有数据交换，开始发送 keepalive syn 探活包；
> 2. tcp_keepalive_probes 发送多少次探活包
> 3. tcp_keepalive_intvl 探活包发送间隔
>
> 传输层

如何开启

- linux全局

> /etc/sysctl.conf
>
> net.ipv4.tcp_keepalive_time=7200
>
> net.ipv4.tcp_keepalive_intvl=75
>
> net.ipv4.tcp_keepalive_probes=9
>
> sysctl -p 使其生效
>
> 没有配置开启

- 单个连接

> 可以使用三个属性设置：TCP_KEEPCNT、TCP_KEEPIDLE、TCP_KEEPINTVL；
>
> 开启：SO_KEEPALIVE

redis源码

```
// 开启tcp-keepalive
int val = 1;
if (setsockopt(fd, SOL_SOCKET, SO_KEEPALIVE, &val, sizeof(val)) == -1)
{
    anetSetError(err, "setsockopt SO_KEEPALIVE: %s", strerror(errno));
    return ANET_ERR;
}

// 设置
val = interval;
if (setsockopt(fd, IPPROTO_TCP, TCP_KEEPIDLE, &val, sizeof(val)) < 0) {
    anetSetError(err, "setsockopt TCP_KEEPIDLE: %s\n", strerror(errno));
    return ANET_ERR;
}

/* Send next probes after the specified interval. Note that we set the
     * delay as interval / 3, as we send three probes before detecting
     * an error (see the next setsockopt call). */
val = interval/3;
if (val == 0) val = 1;
if (setsockopt(fd, IPPROTO_TCP, TCP_KEEPINTVL, &val, sizeof(val)) < 0) {
    anetSetError(err, "setsockopt TCP_KEEPINTVL: %s\n", strerror(errno));
    return ANET_ERR;
}

/* Consider the socket in error state after three we send three ACK
     * probes without getting a reply. */
val = 3;
if (setsockopt(fd, IPPROTO_TCP, TCP_KEEPCNT, &val, sizeof(val)) < 0) {
```

```
30      anetSetError(err, "setsockopt TCP_KEEPCNT: %s\n", strerror(errno));
31      return ANET_ERR;
32  }
```

为什么应用层需要开启心跳检测?

> 因为传输层探活检测，无法判断进程阻塞或者死锁的情况；
>
> 心跳检测： 每隔10秒发送一次心跳包 3次没有收到 close

应用

> 1. 数据库间，主从复制，使用心跳检测；
> 2. 客户端与服务器，使用心跳检测；
> 3. 客户端->反向代理->上游服务器；反向代理与上游服务器使用探活检测；
> 4. 服务端->数据库，使用探活检测；对于数据库而言，服务端是否阻塞跟它无关；