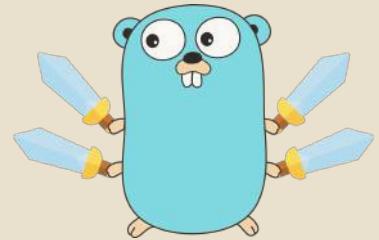


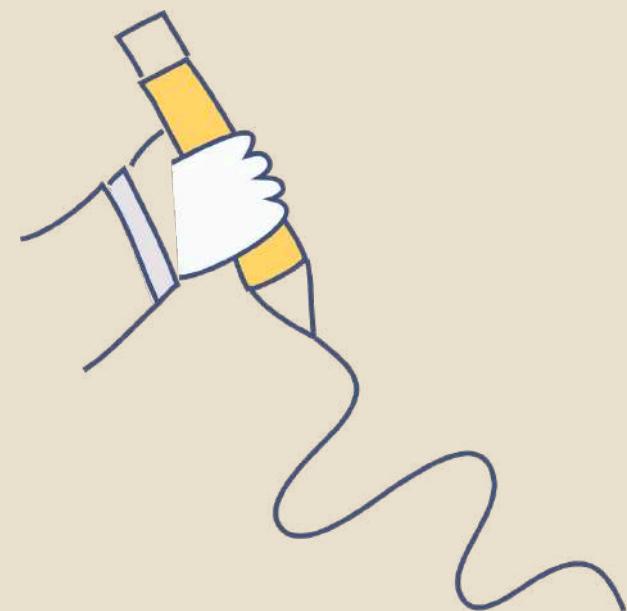
# 网络编程的那些事儿<sup>v1</sup>



@ 芮峰云

[github.com/rfyiamcool](https://github.com/rfyiamcool)





- 1
- 2
- 3
- 4

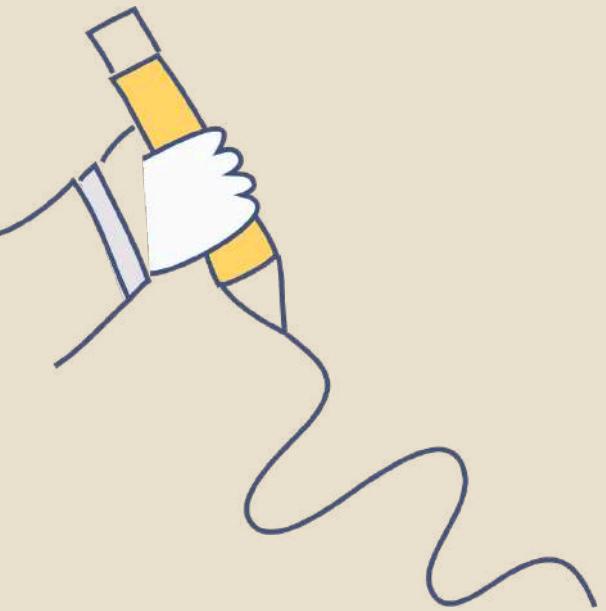
socket基本原理

io 多路复用

网络编程模型

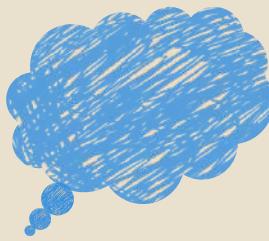
常见问题



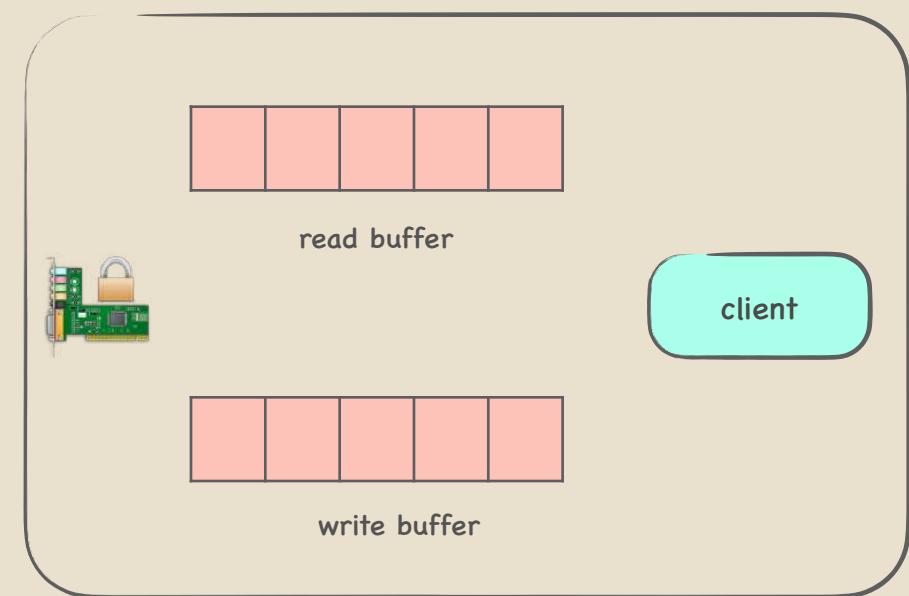
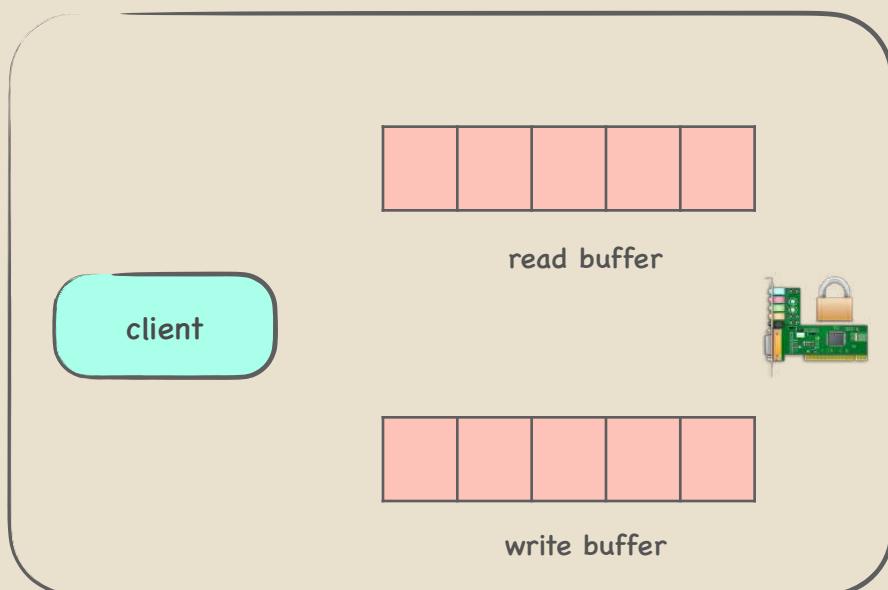


1

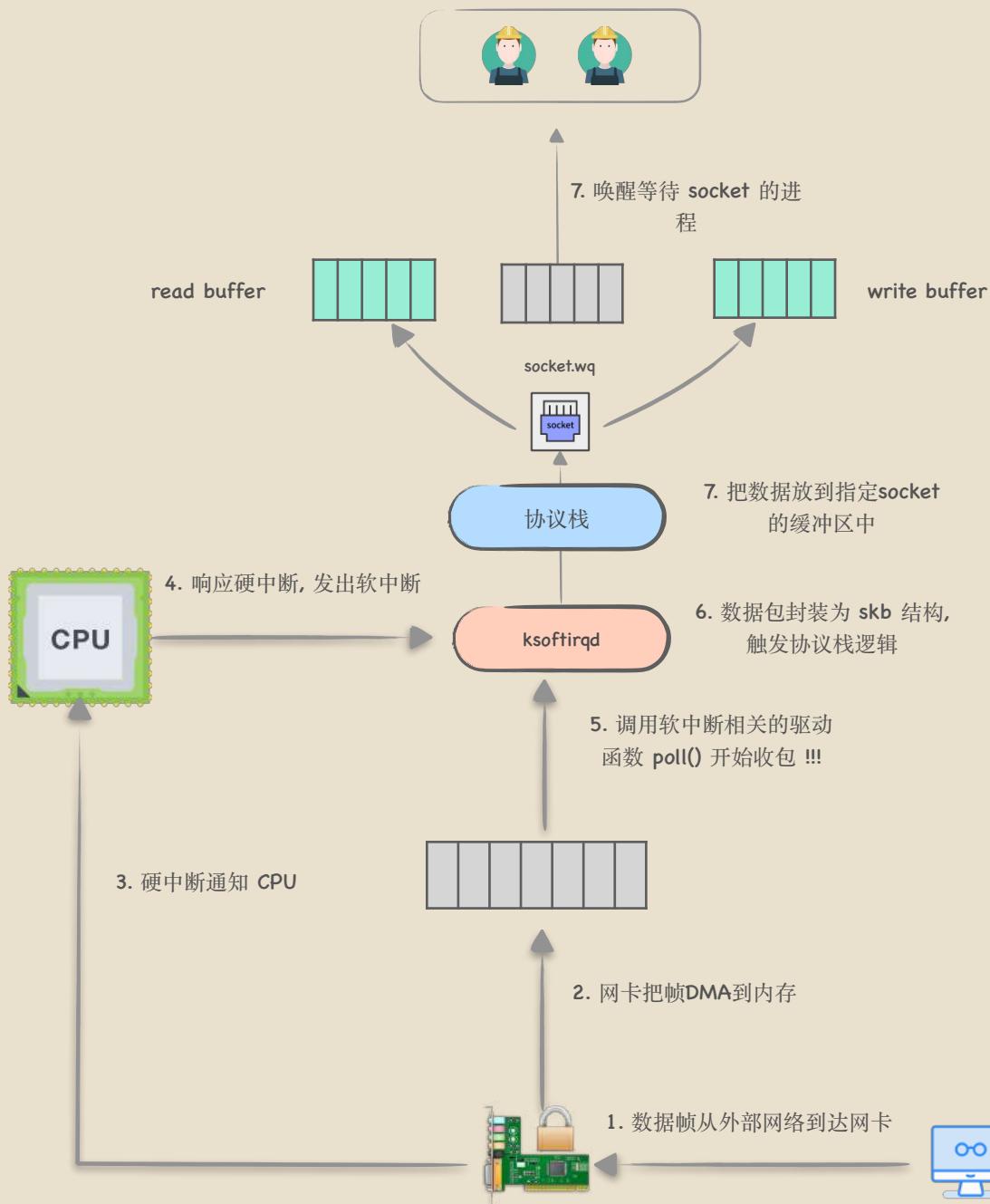
socket



# socket

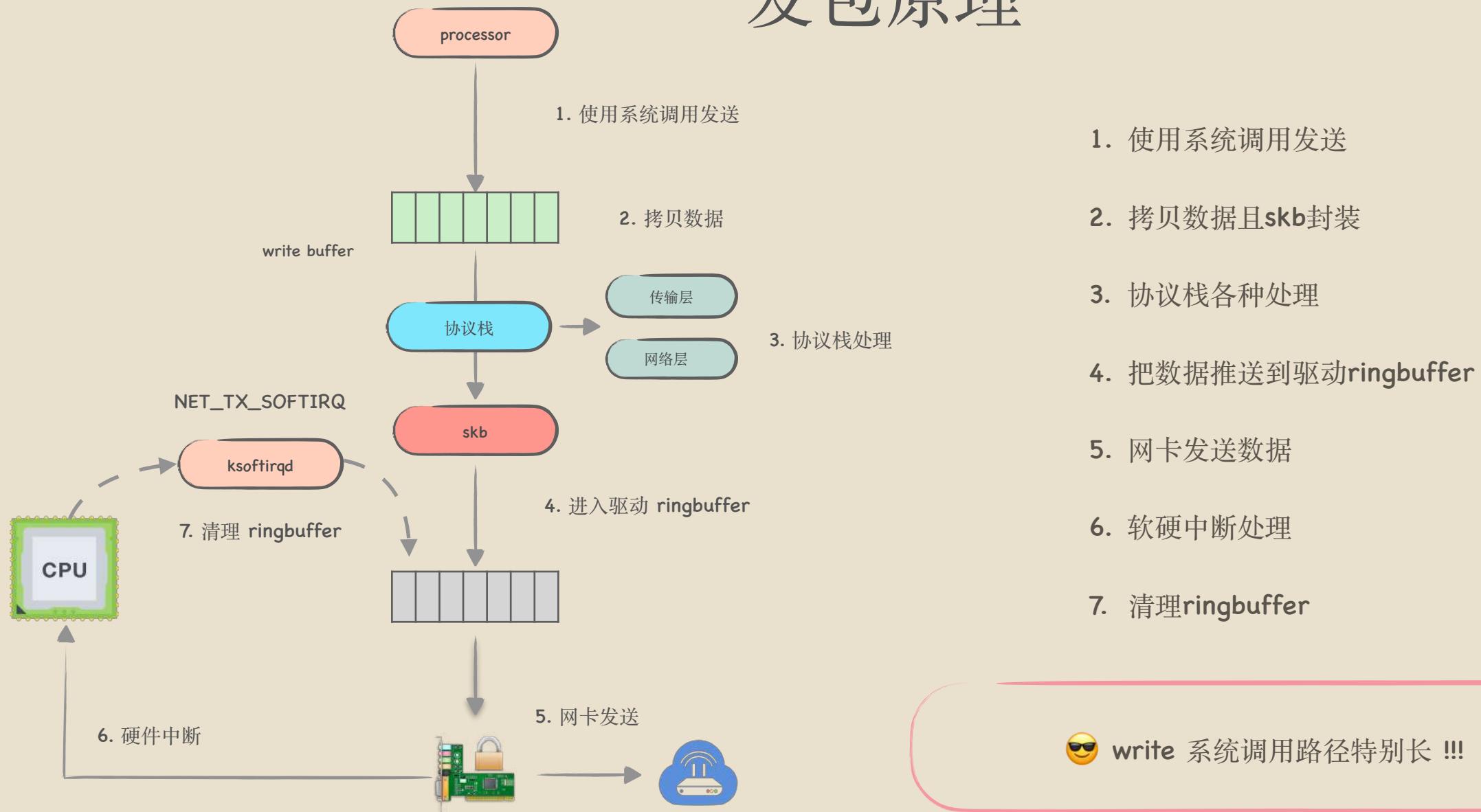


# 收包原理



1. 网卡将数据帧DMA到内存 RingBuffer中, 然后向CPU硬中断;
2. CPU响应中断请求, 调用网卡启动时注册的中断处理函数;
3. 中断处理函数几乎没干啥, 就发起了软中断请求;
4. 内核线程ksoftirqd线程发现有软中断请求到来, 先关闭硬中断;
5. ksoftirqd 线程开始调用驱动的 poll 函数收包;
6. 协议栈把数据包放到对应的socket的缓存队列里;
7. 调用 socket waitqueue 的回调函数链, 激活等待的进程.

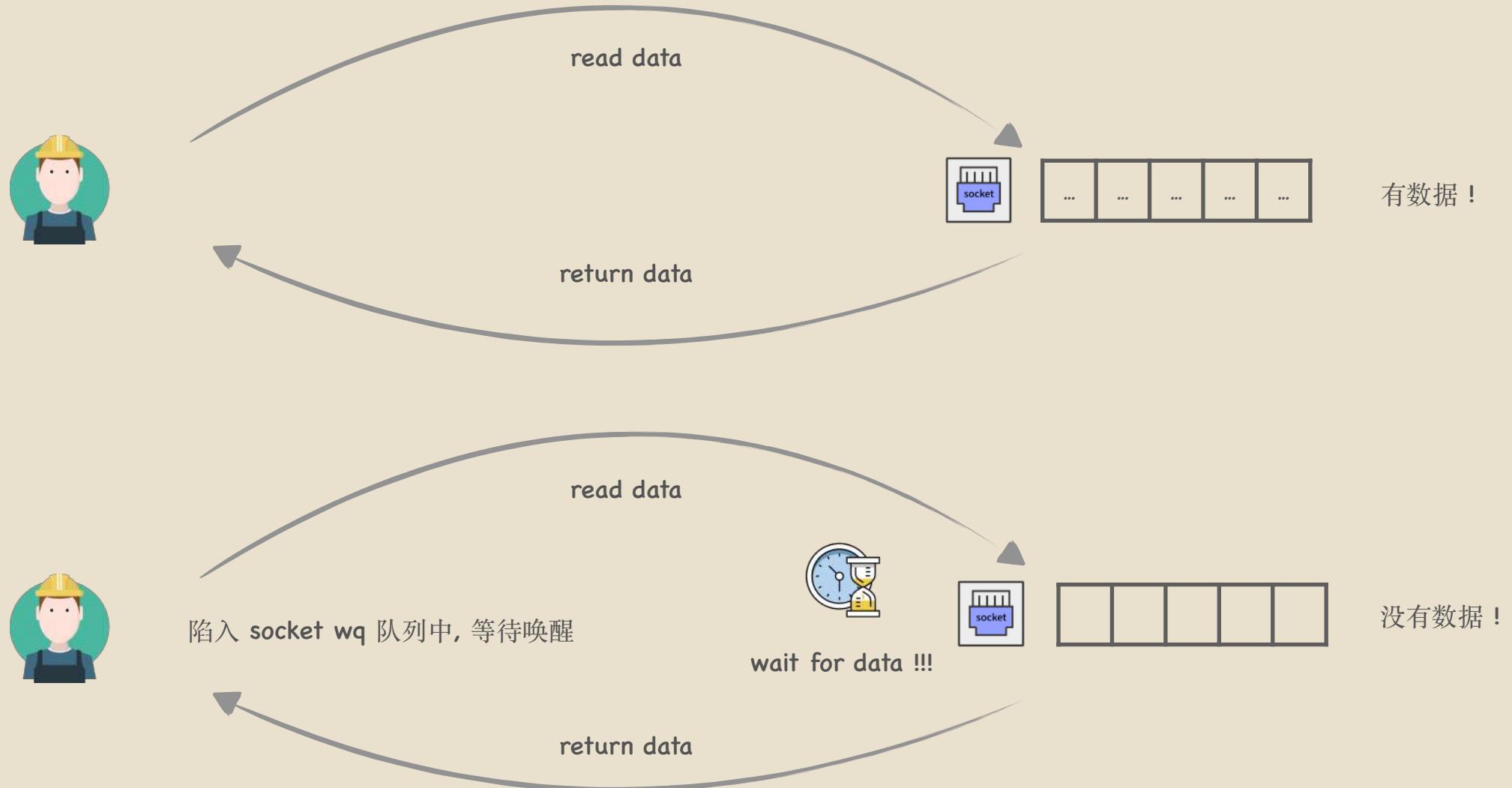
# 发包原理



# 阻塞 vs 非阻塞



# 阻塞模式



# 非阻塞模式



read data



没有数据！

errno again



read data



没有数据！

errno again  
errno again  
errno again  
errno again

read data



有数据！

return data

# 两阶段

阻塞/非阻塞 IO

硬件网卡

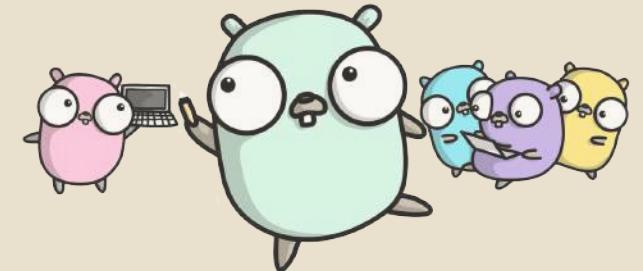
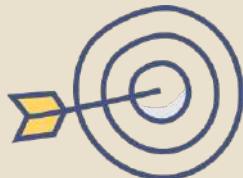
第一阶段

同步/异步 IO

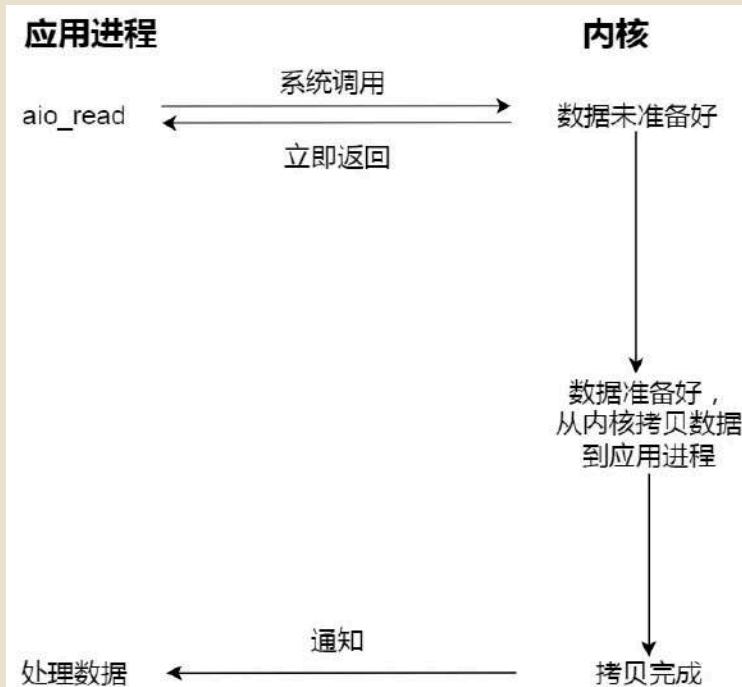
内核空间

第二阶段

用户空间



# 同步 vs 异步

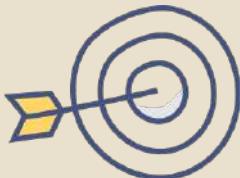
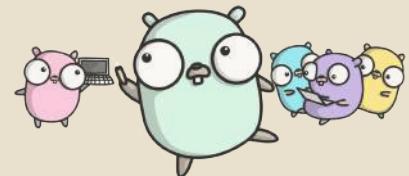


## \* 同步

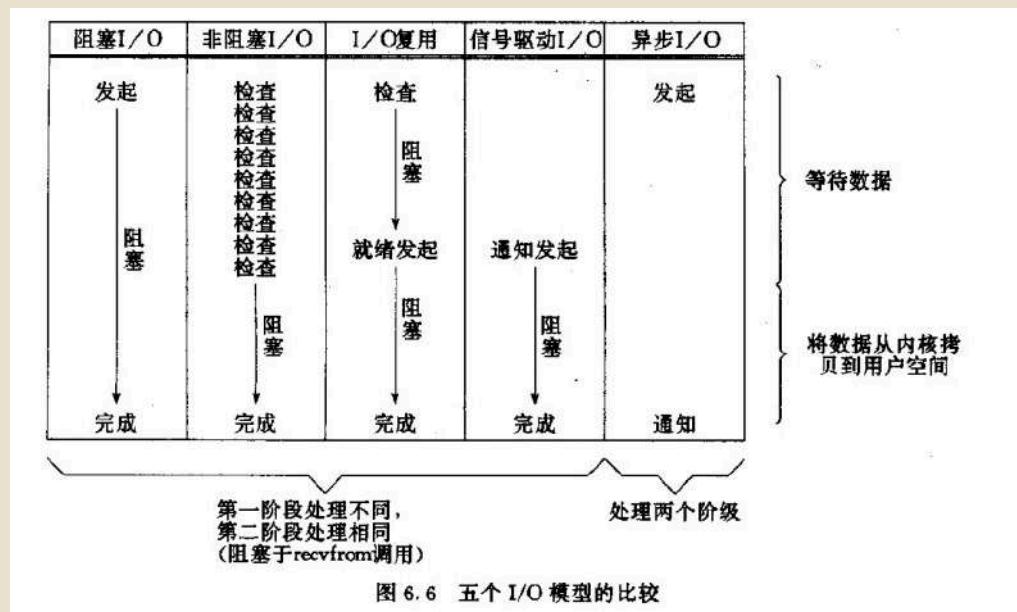
- \* 应用进程主动将 `socket` 读缓存中的数据读到应用进程内存中，这个过程是同步的

## \* 异步

- \* 无需主动读写数据，由操作系统内核完成数据的读写，等待完成通知



# 五种 IO 模型



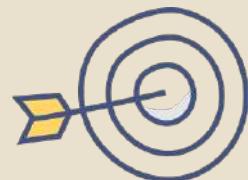
\* 阻塞io模型

\* 非阻塞io模型

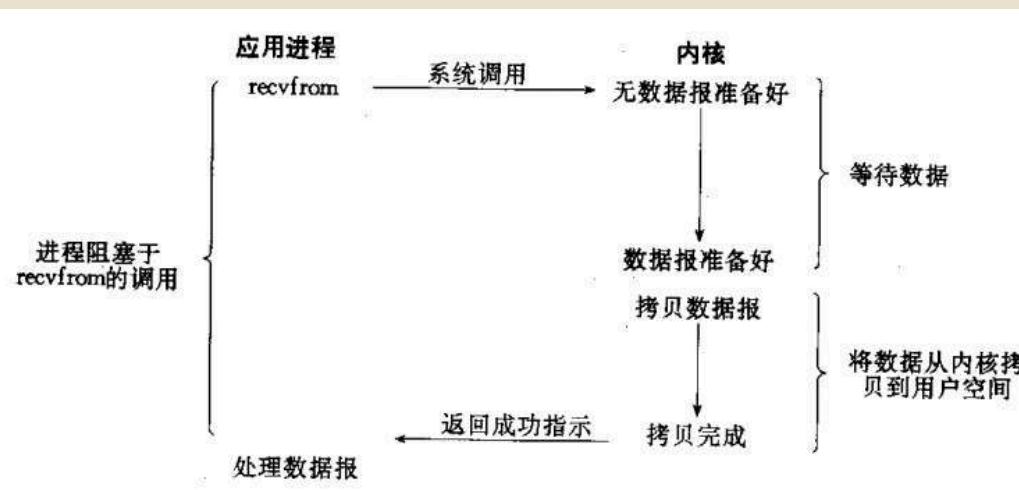
\* io多路复用模型

\* 信号驱动IO模型

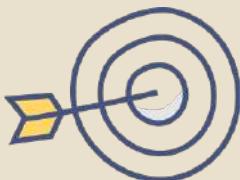
\* 异步io模型



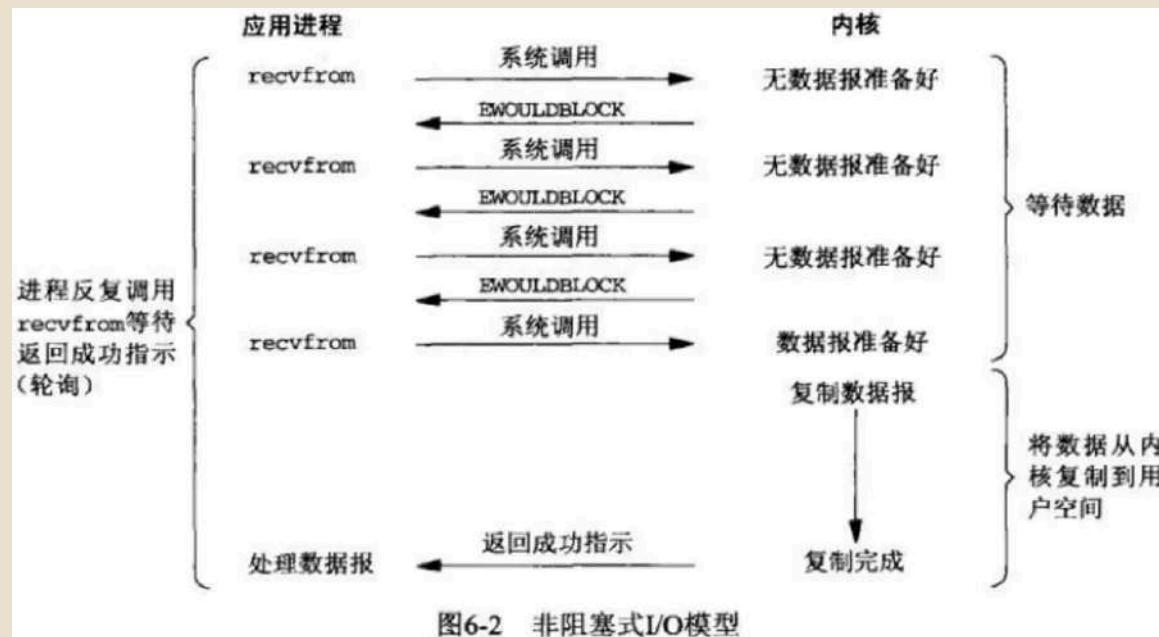
# 阻塞 io 模型



- \* 当数据不可读被挂起, 阻塞等待有数据可读.
- \* 当写缓冲区写满不可再写时, 阻塞挂起等待可写.
- \* 优点
  - \* 无资源时直接休眠等待
- \* 缺点
  - \* 一个线程只能监听一个fd



# 非阻塞 io 模型

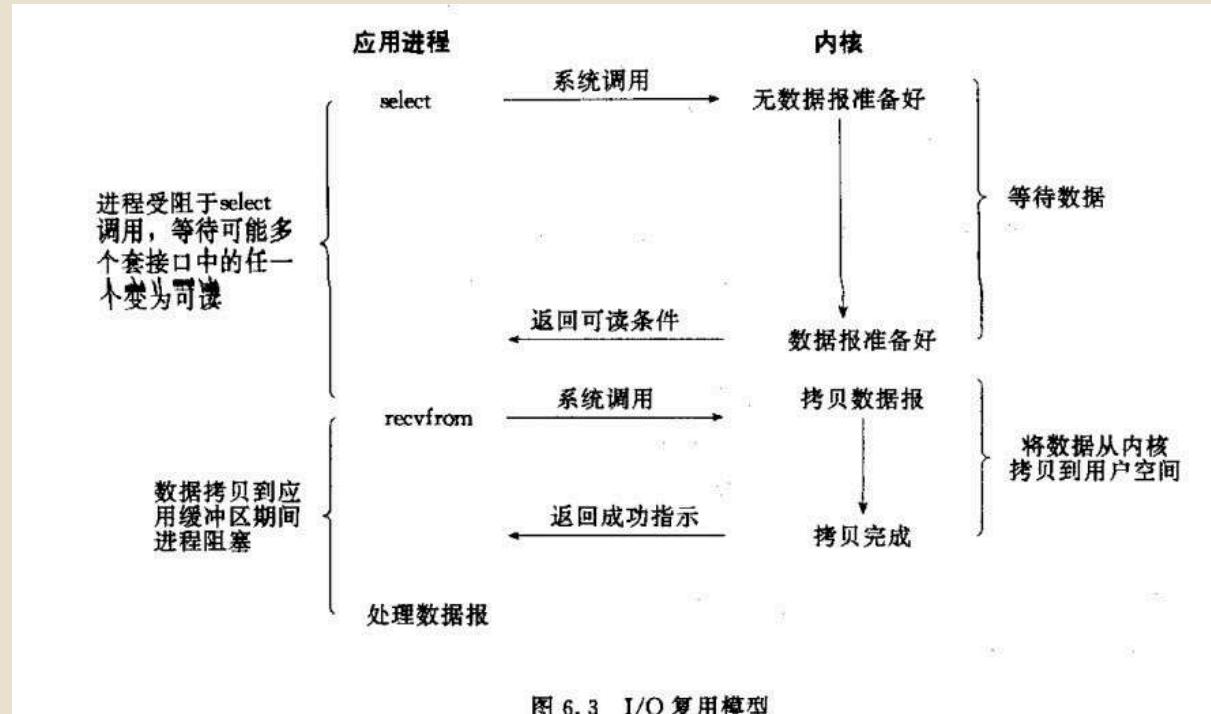


```
int s = socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK, IPPROTO_TCP);
fcntl(sockfd, F_SETFL, fcntl(sockfd, F_GETFL, 0) | O_NONBLOCK);

ioctl(sockfd, FIONBIO, 1); //1:非阻塞 0:阻塞
```

- \* socket 设为非阻塞, 然后不断的短轮询, 空耗 cpu 资源
- \* 不管是否可读, 一个劲的读, 无数据时返回 EAGAIN 或 EWOULDBLOCK
- \* 不管是否可写, 一个劲的尝试写, 不能写返回 EAGAIN 或 EWOULDBLOCK
- \* 优点?
  - \* 可以遍历轮询多个资源 😊
- \* 缺点
  - \* 太粗暴, 空耗资源

# io 多路复用模型



\* lib

\* select

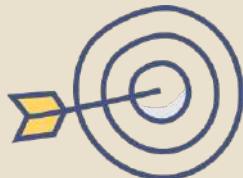
\* poll

\* epoll

\* 可以同时监听多个资源

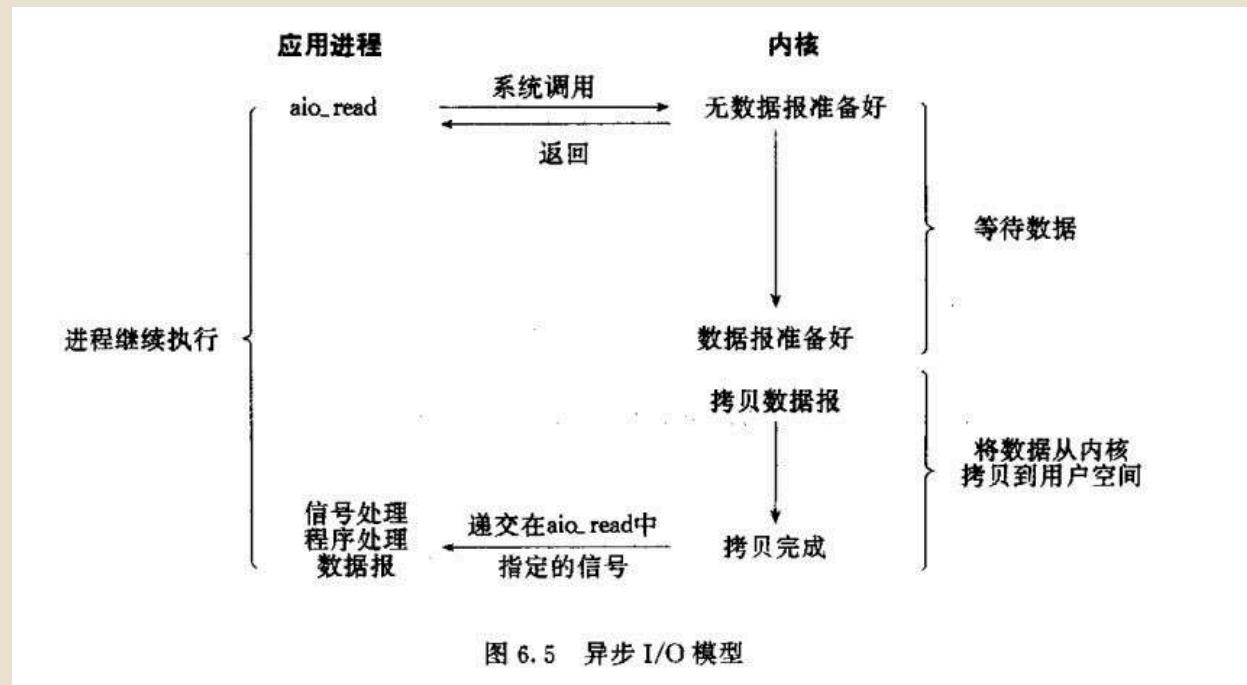
\* 无需轮询, 有数据被唤醒, 没数据则 block 等待

\* 当 kernel 中数据准备好的时候, `recvfrom` 会将数据从 kernel 拷贝到用户内存中, 这个时候进程是被block了



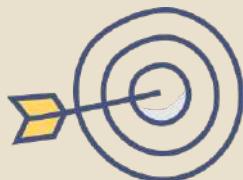
epoll 为 同步非阻塞 模式

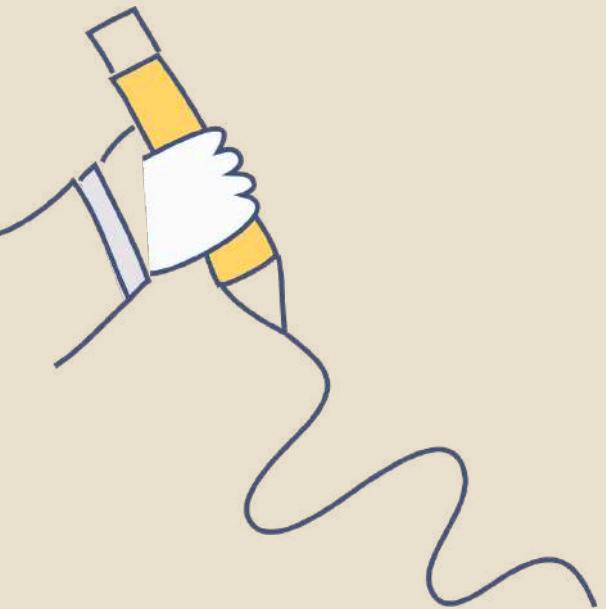
# aio 模型



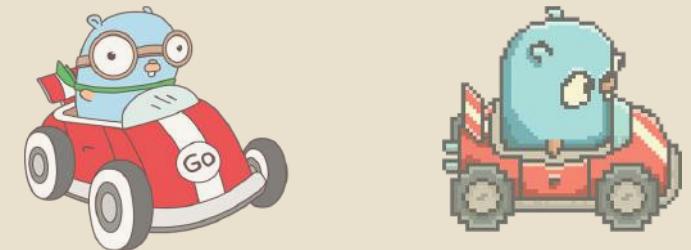
- \* **aio\_read** 产生系统调用, **kernel** 在数据准备好后将数据从内核空间拷贝到用户空间, 返回事件告知数据成功;
- \* 业务进程无需 **block** 进程去同步读取, 内核帮你把数据 **read** 读取;

aio 为异步非阻塞 !!!





## IO 多路复用



# select

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(2000);
addr.sin_addr.s_addr = INADDR_ANY;
bind(sockfd, (struct sockaddr*)&addr, sizeof(addr));
listen (sockfd, 5);

for (i=0;i<1024;i++)
{
    memset(&client, 0, sizeof(client));
    addrlen = sizeof(client);
    fds[i] = accept(sockfd,(struct sockaddr*)&client, &addrlen);
    if(fds[i] > max)
        max = fds[i];
}

while(1){
    FD_ZERO(&rset);
    for (i = 0; i < 1024; i++ ) {
        FD_SET(fds[i],&rset);
    }
    select(max+1, &rset, NULL, NULL, NULL);

    for(i=0;i<1024;i++) {
        if (FD_ISSET(fds[i], &rset)){
            memset(buffer,0,MAXBUF);
            read(fds[i], buffer, MAXBUF);
        }
    }
}
return 0;
}
```

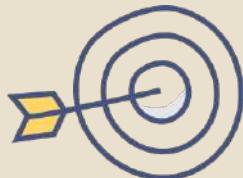
- \* 最大文件描述符限制, 32bit 为 1024 , 64bit 为 2048
- \* fd\_set 不可重用, 每次调用都需要设置
- \* 用户态和内核态拷贝传递 fd\_set
- \* O(n) 时间复杂度来轮询判断是否有事件

简单粗暴

# poll

```
struct pollfd {  
    int fd;  
    short events;  
    short revents; // 可重用  
};  
  
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

- \* 解决了 `select fd_size` 限制
- \* 无需来回重置 `fd_set` 事件标志
- \* 但 `fd` 集合仍需要传递
- \* 依旧需要 遍历轮询 判断事件



# epoll



```
struct eventpoll {  
    spin_lock_t lock;  
    struct mutex mtx;  
  
    // 等待队列  
    wait_queue_head_t wq;  
  
    // 事件满足条件的链表  
    struct list_head rdllist;  
  
    // 用于管理所有fd的红黑树  
    struct rb_root rbr;  
}
```

- \* 没有文件数量限制
- \* 基于回调通知机制，无需轮询事件，只需遍历就绪 ready list 表
- \* 存放 **epitem** 的红黑树存在于内核空间中
- \* 可动态地添加/删除/修改事件



# epoll

```

● ● ●

int epfd = epoll_create(POLL_SIZE);
struct epoll_event ev;
struct epoll_event *events = NULL;
nfds = epoll_wait(epfd, events, 20, 500);
{
    for (n = 0; n < nfds; ++n) {
        if (events[n].data.fd == listener) {
            //如果是主socket的事件的话，则表示有新连接进入了，进行新连接的处理。
            client = accept(listener, (struct sockaddr *)&local, &addrlen);
            if (client < 0) {
                perror("accept");
                continue;
            }
            setnonblocking(client);           //将新连接置于非阻塞模式
            ev.events = EPOLLIN | EPOLLET; //并且将新连接也加入EPOLL的监听队列。
            ev.data.fd = client;
            if (epoll_ctl(epfd, EPOLL_CTL_ADD, client, &ev) < 0) {
                fprintf(stderr, "epollsetinsertionerror:fd=%d", client);
                return -1;
            }
        }
        else if(event[n].events & EPOLLIN)
        {
            // 对已连接的用户进行读取
            int sockfd_r;
            if ((sockfd_r = event[n].data.fd) < 0)
                continue;
            read(sockfd_r, buffer, MAXSIZE);
            ev.events = EPOLLOUT | EPOLLET;
            epoll_ctl(epfd, EPOLL_CTL_MOD, sockfd_r, &ev)
        }
        else if(event[n].events & EPOLLOUT)
        {
            // 可数据发送
            int sockfd_w = events[n].data.fd;
            write(sockfd_w, buffer, sizeof(buffer));
            ev.events = EPOLLIN | EPOLLET;
            epoll_ctl(epfd, EPOLL_CTL_MOD, sockfd_w, &ev)
        }
    ...
}
}

```

- \* `epfd = epoll_create(intsize);`
- \* `epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)`
  - \* `EPOLL_CTL_ADD`
  - \* `EPOLL_CTL_MOD`
  - \* `EPOLL_CTL_DEL`
- \* `epoll_wait(int epfd, struct epoll_event * events, intmaxevents, int timeout) ;`



# epoll event

## \* 事件 Event

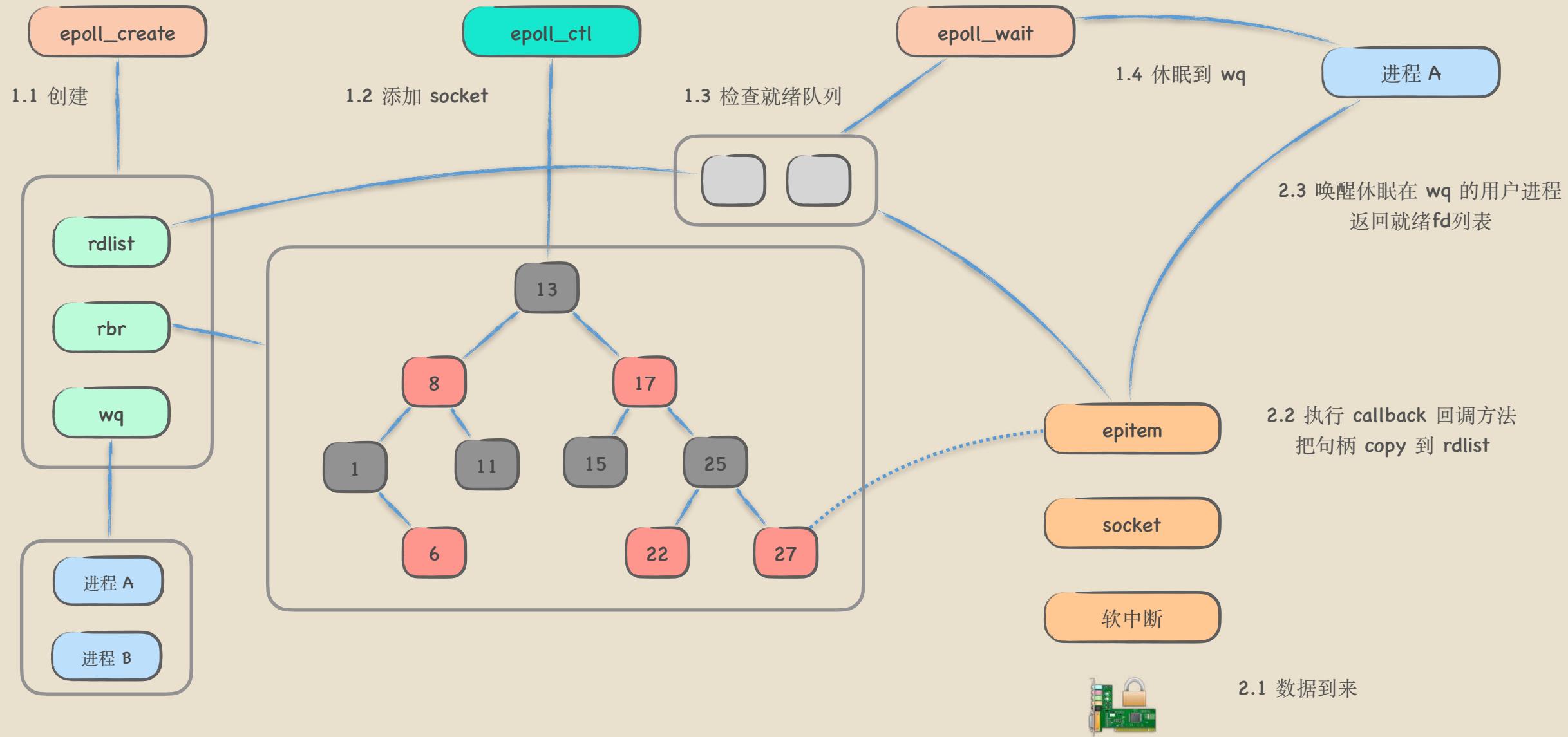
- \* EPOLLIN 读事件
- \* EPOLLOUT 写事件
- \* EPOLLET 边缘触发模式
- \* EPOLLLT 水平触发模式
- \* EPOLLRDHUP, EPOLLHUP 连接异常
- \* EPOLLERR 连接异常



```
struct epoll_event ev;
setnonblocking(cfd);
ev.data.fd = cfd;
ev.events = EPOLLIN | EPOLLET;
epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &ev);
```



# epoll design

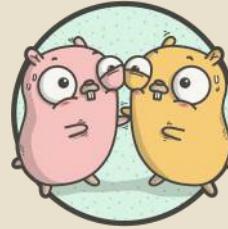


# epoll

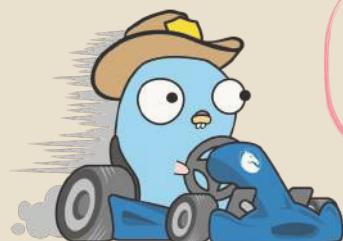
- \* 水平触发 ( level trigger )
- \* 开发相对简单
- \* 只要可读 (读缓冲区有数据), 可写 (写缓冲区未满) 就一直可以拿到事件
- \* 当无数据可写时, 需要去除 `epollout` 事件, 不然一直触发 !

```
// 水平触发
evt.events = EPOLLIN;           // LT 水平触发 (默认) EPOLLLT
evt.data.fd = pfd[0];

// 边沿触发
evt.events = EPOLLIN | EPOLLET; // ET 边沿触发
evt.data.fd = pfd[0];
```



- \* 边缘触发 ( edge trigger )
  - \* 当有读事件到来变化才触发, 当缓冲区可写才触发 .
  - \* 只会通知一次, 未处理就麻烦了
- \* 理论上性能比 lt 更好



当用户发送一个http请求, 服务端未读完或丢弃, 这时用户不再发送数据, 那么该fd就不会再通知 !

# epoll

- \* 水平触发 lt

- \* java nio

- \* redis

- \* muduo

- \* skynet

- \* tornado

- \* golang gev

- \* libevent ( default )

- \* 边缘触发 et

- \* nginx

- \* envoy

- \* mysql

- \* netty ( default )

- \* golang net

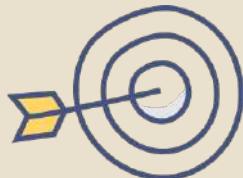
- \* golang gnet



# epoll



- \* epoll 线程安全的 ?
- \* epoll 通过 mmap 共享空间 ?
- \* epoll 是异步的 ?
- \* epoll 使用阻塞 fd 读写 ?

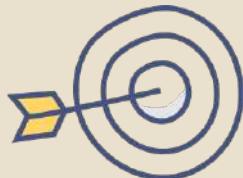


# epoll 可监听的资源

- \* 可监听哪些资源 ?

- \* sockfd
- \* eventfd
- \* timerfd
- \* signalfd
- \* inotifyfd
- \* pipefd

- \* 不能直接监听普通磁盘文件 ???
- \* 文件系统未实现 poll 接口
- \* 磁盘始终是“Ready”就绪状态



```
#include <stdio.h>
#include <sys/epoll.h>
#include <fcntl.h>

int main()
{
    int epfd, fd;
    struct epoll_event ev, events[2];
    int result;

    epfd = epoll_create(10);
    if (epfd < 0) {
        perror("epoll_create()");
        return -1;
    }

    fd = open("./test.txt", O_RDONLY | O_CREAT);
    if (fd < 0) {
        perror("open()");
        return -1;
    }

    ev.events = EPOLLIN;

    result = epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
    if (result < 0) {
        perror("epoll_ctl()");
        return -1;
    }

    epoll_wait(epfd, events, 2, -1);

    return 0;
}
```

epoll\_ctl(): Operation not permitted

# AIO (async io)

## \* kernel native aio

- \* 原生实现
- \* nginx, mysql 有在使用
- \* 只能使用 direct io 直接io
- \* 不支持 socket !!! ⚡

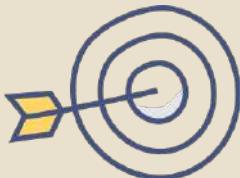
```
# nginx aio config

location /video/ {
    sendfile          on;
    aio               on;
    directio         8m;    !!!
    directio_alignment 4k;
}
```

## \* glibc aio

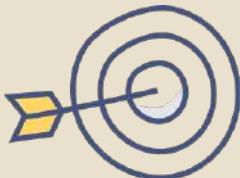
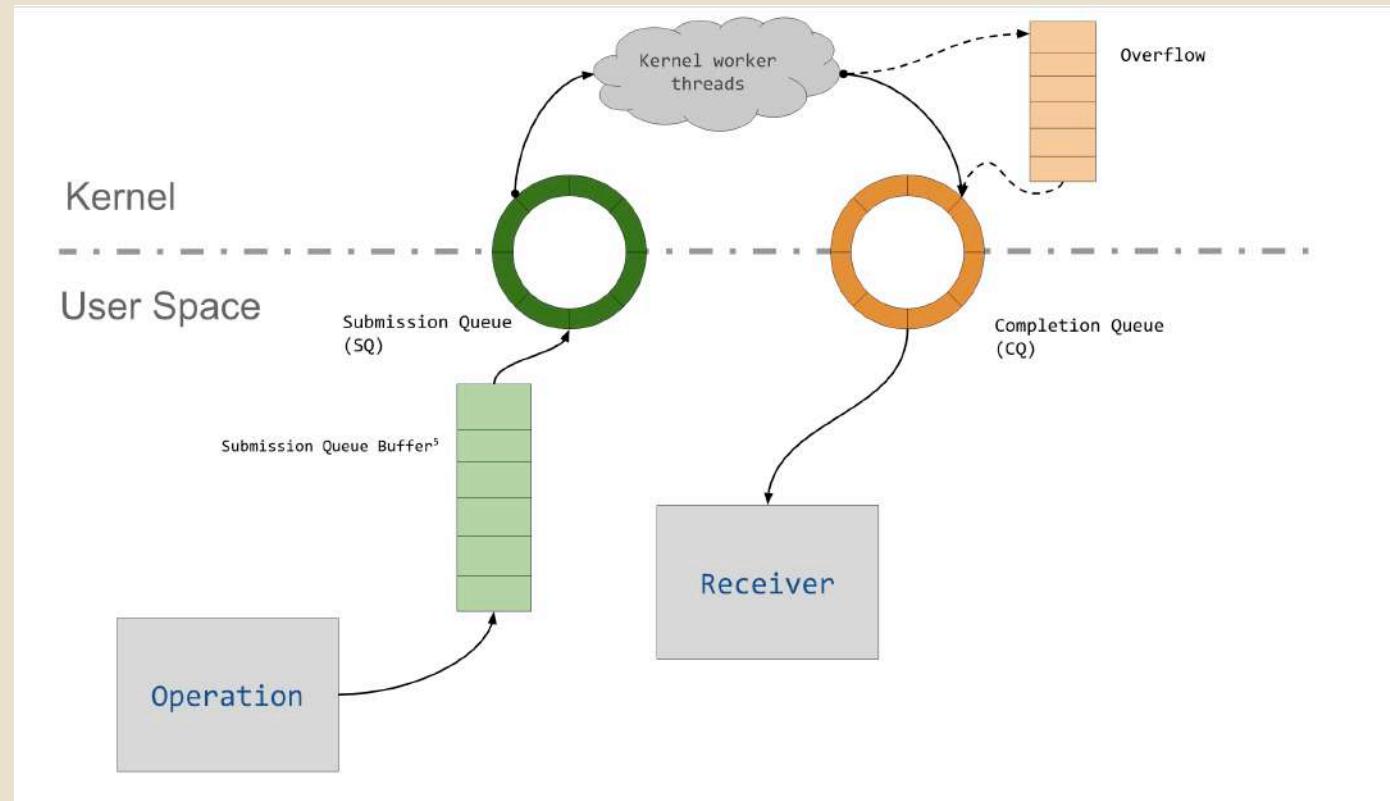
- \* 使用线程池实现 user 层异步io
- \* 可以使用 buffer io 缓存io

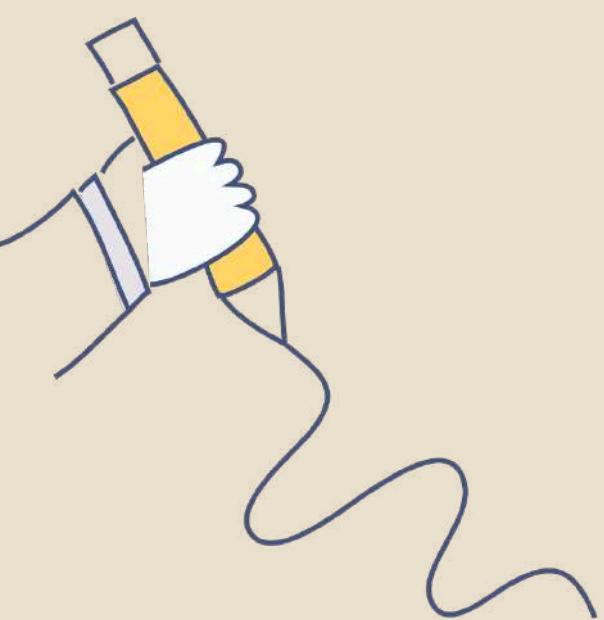
nginx aio 强制使用 direct io



# io\_uring

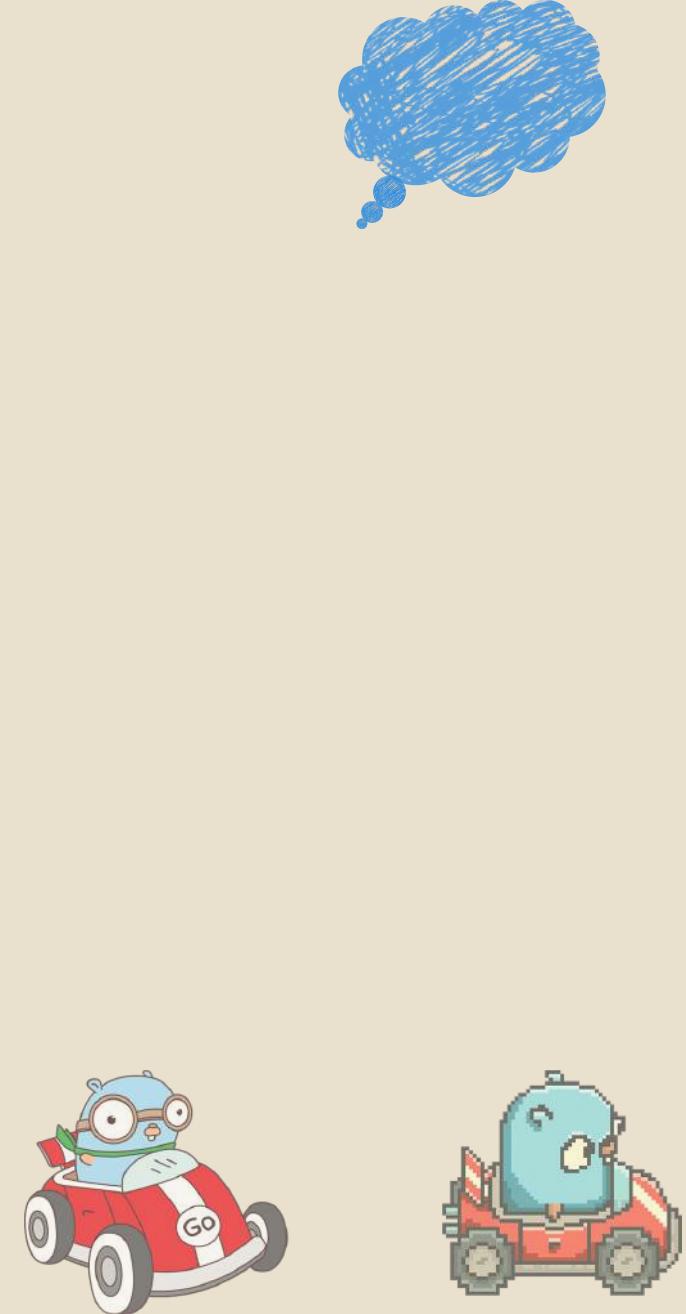
- \* io\_uring ( 全新的系统级异步接口 )
  - \* kernel 5.1 基础, kernel 5.6 完善
  - \* 可替换 epoll
  - \* 支持文件读写
  - \* 支持socket fd
  - \* mmap 映射
  - \* 支持 buffered io



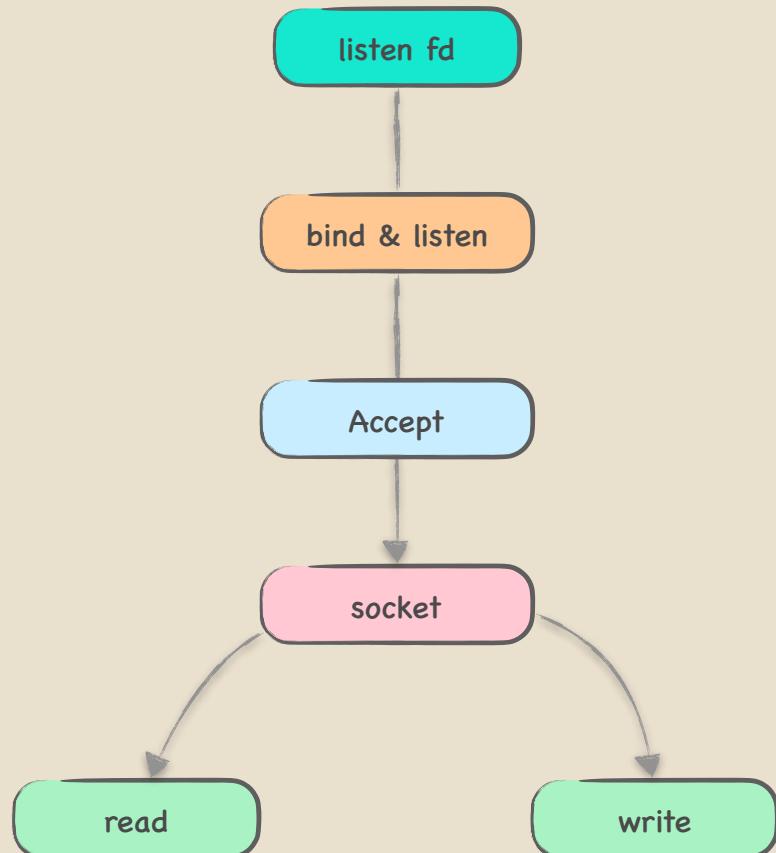


3

## 网络编程模型



# single thread



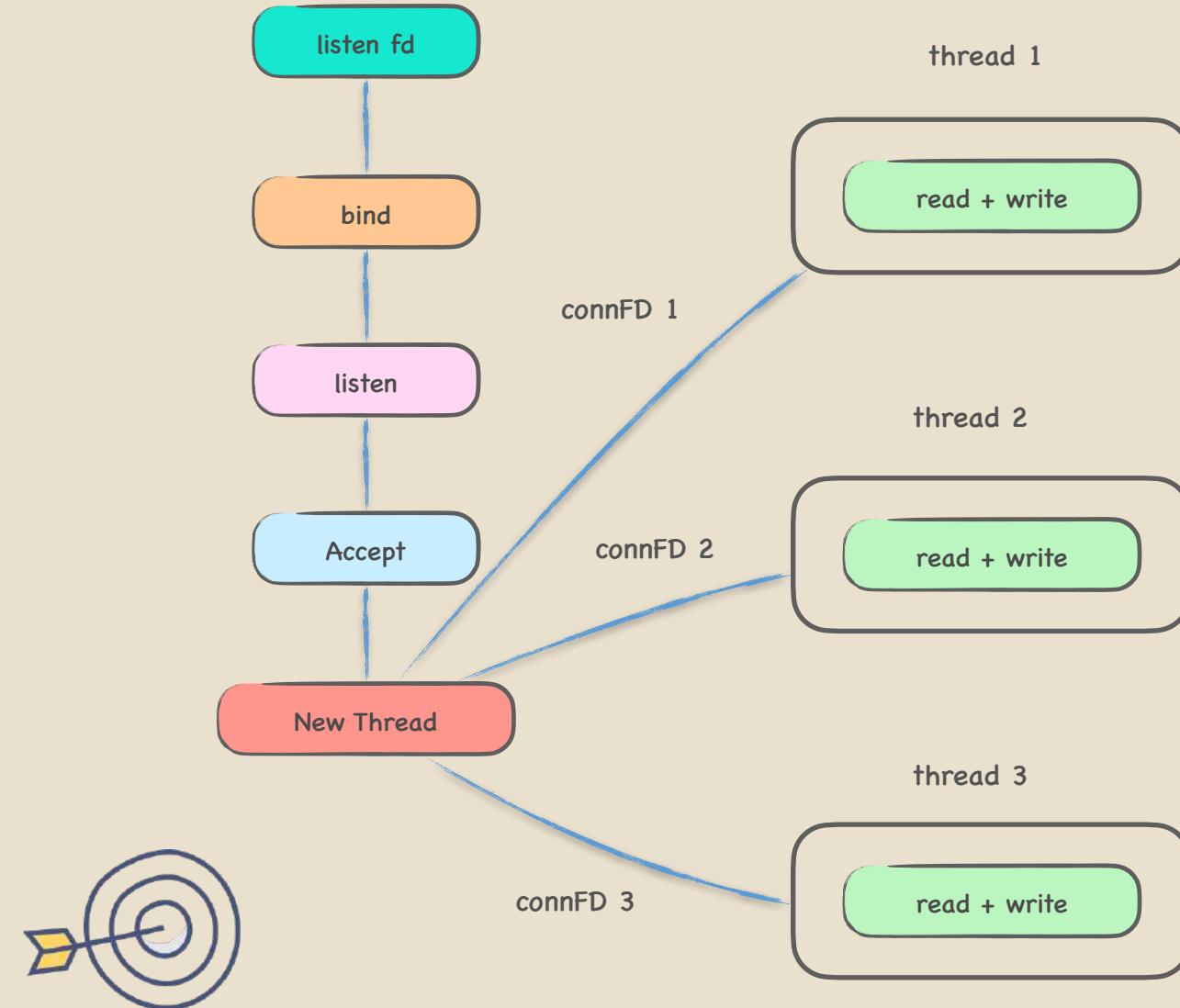
```
import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)

# Bind the socket to the address given on the command line
server_address = ('localhost', 10000)
sock.bind(server_address)
sock.listen(1)

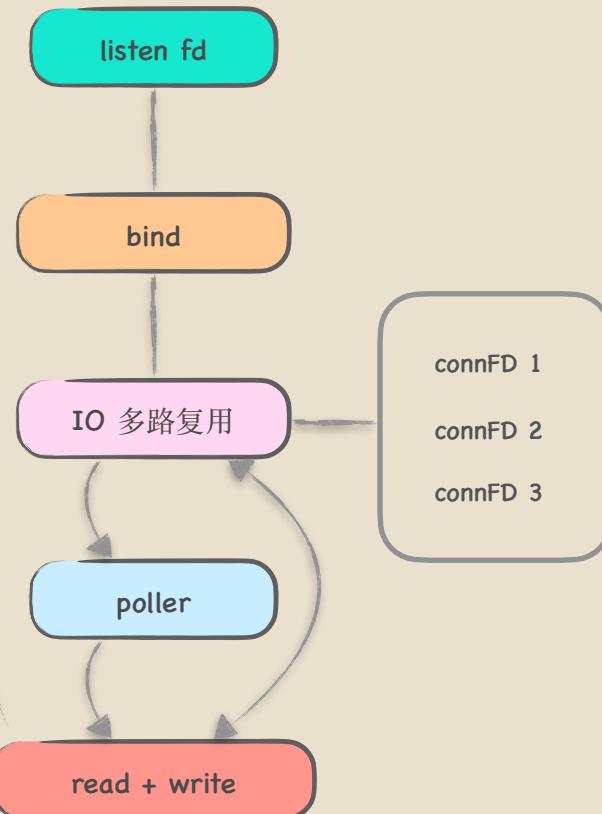
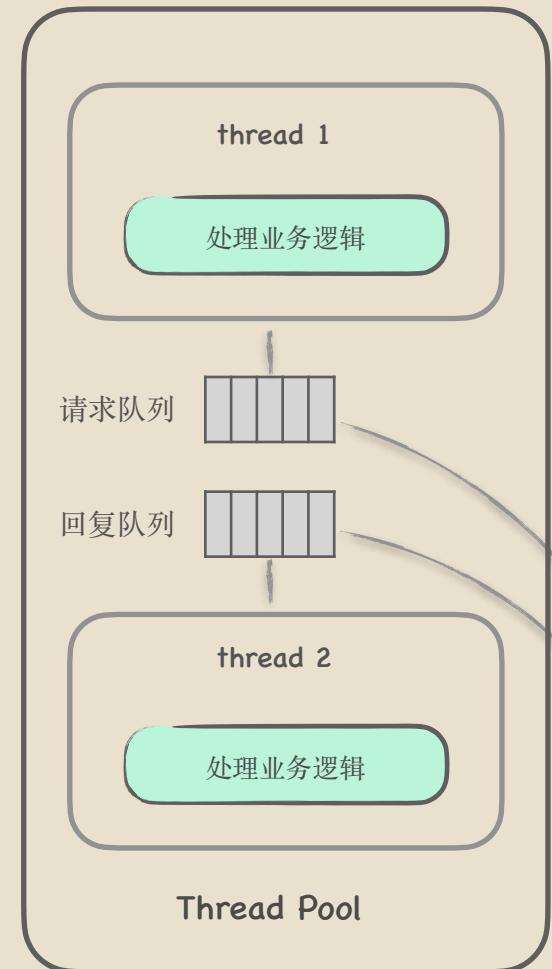
while True:
    connection, client_address = sock.accept()
    try:
        print >>sys.stderr, 'client connected:', client_address
        while True:
            data = connection.recv(16)
            print >>sys.stderr, 'received "%s"' % data
            if data:
                connection.sendall(data)
            else:
                break
    finally:
        connection.close()
```

# new thread



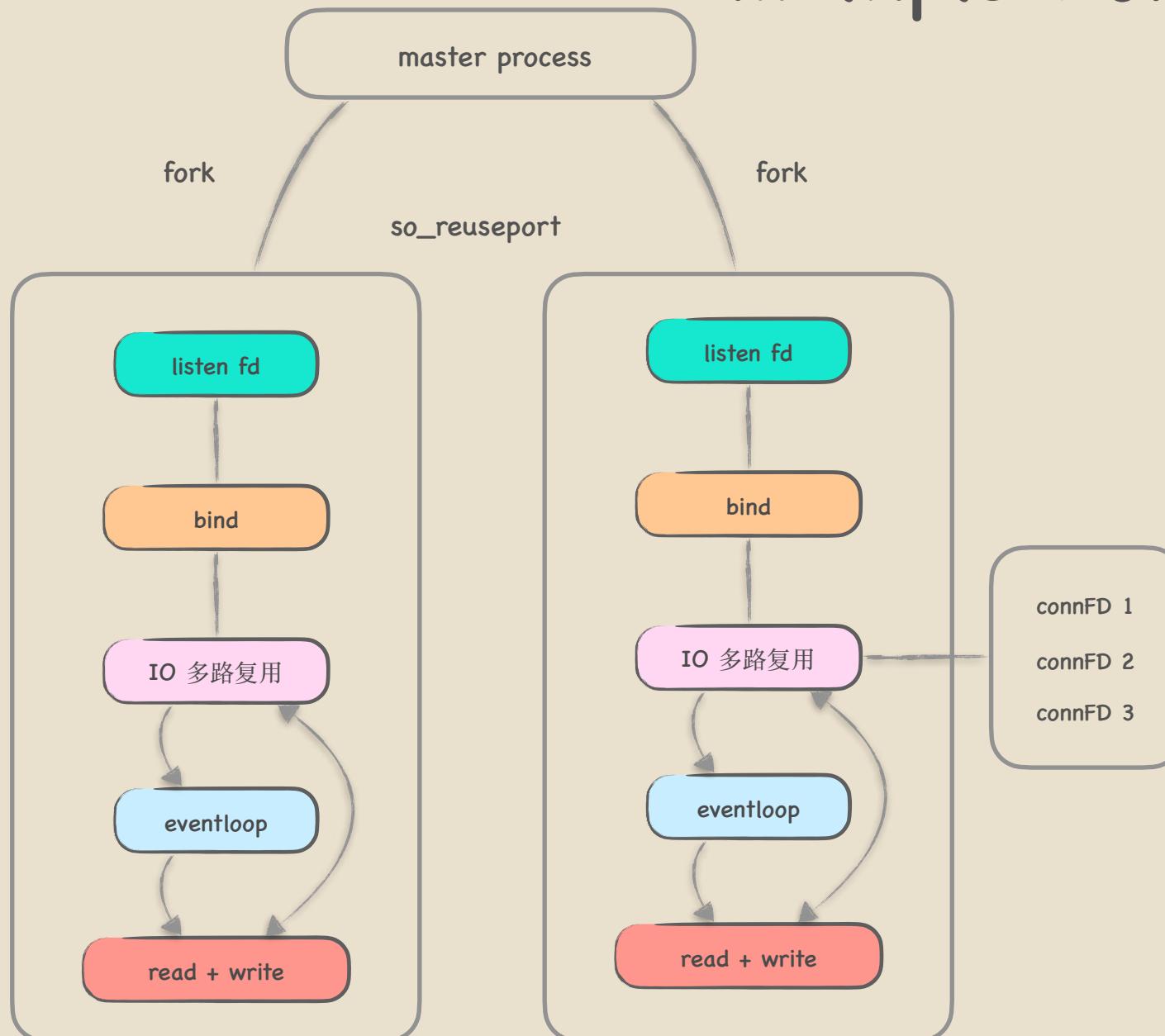
- \* 单线程**Accept**+多线程读写业务（无IO复用）
- \* 优点
  - \* 简单
  - \* 粗暴
- \* 缺点
  - \* 连接跟线程数成正比关系
  - \* 没法高并发

# thread pool



- \* 单线程多路复用 + 多线程业务工作池
- \* 操作过程
  1. **eventloop** 按照协议完整读取请求体
  2. 把请求体和**conn**传递给线程池队列
  3. 业务线程处理完后, 写回复队列
  4. 由 **poller** 回复报文
- \* 优点
  - \* 以前的**web**框架多数是这样的
  - \* 将耗时的业务分离到工作线程池
- \* 缺点
  - \* 同时超过线程池数量的任务会排队等待！
  - \* **socket** 读写只有主线程来操作, 网络单核心！

# multiple worker

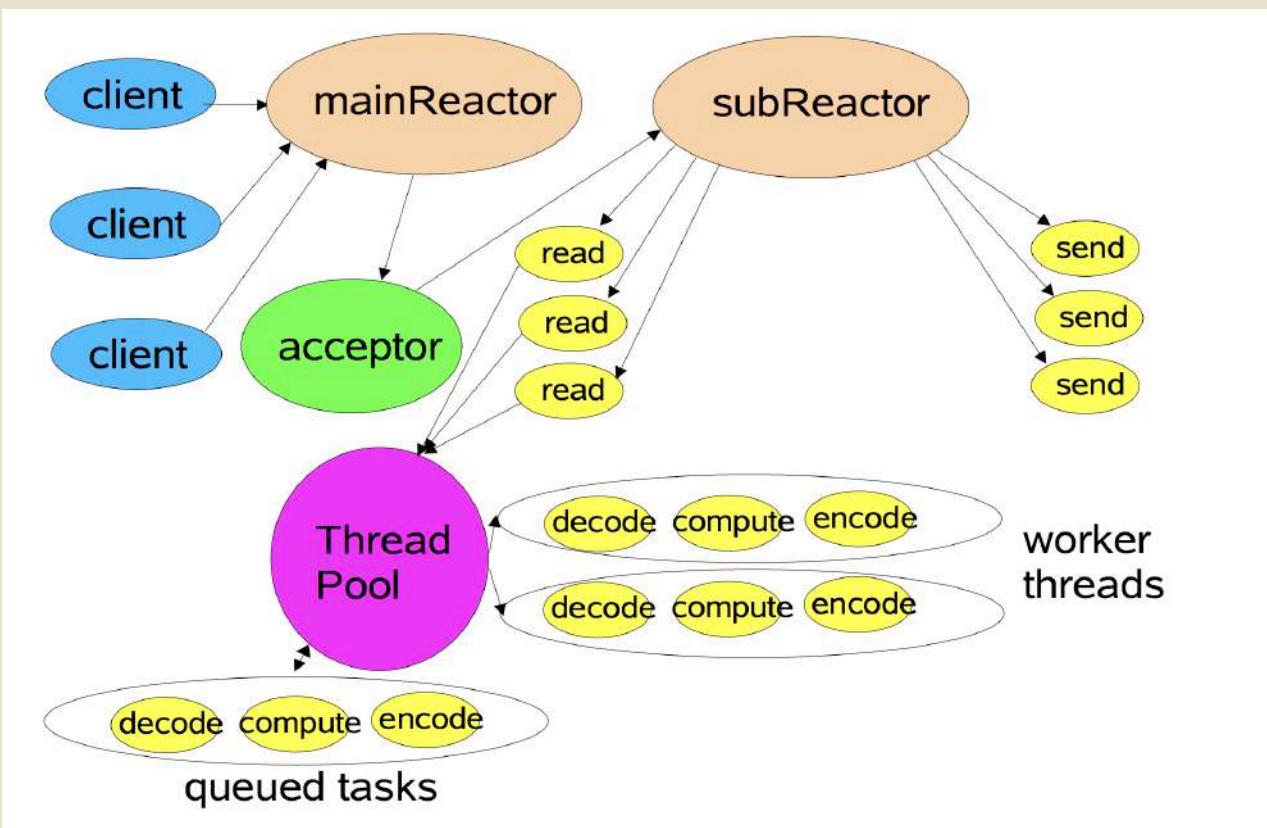


- \* 预先分配 **worker** 线程/进程的数量 .
- \* 通过 **master** 主进程来管理 **worker** 进程 .
- \* 每个线程为一个独立的 **event-loop** !!!
- \* nginx, haproxy, apache 类似该模式 .



# reactor

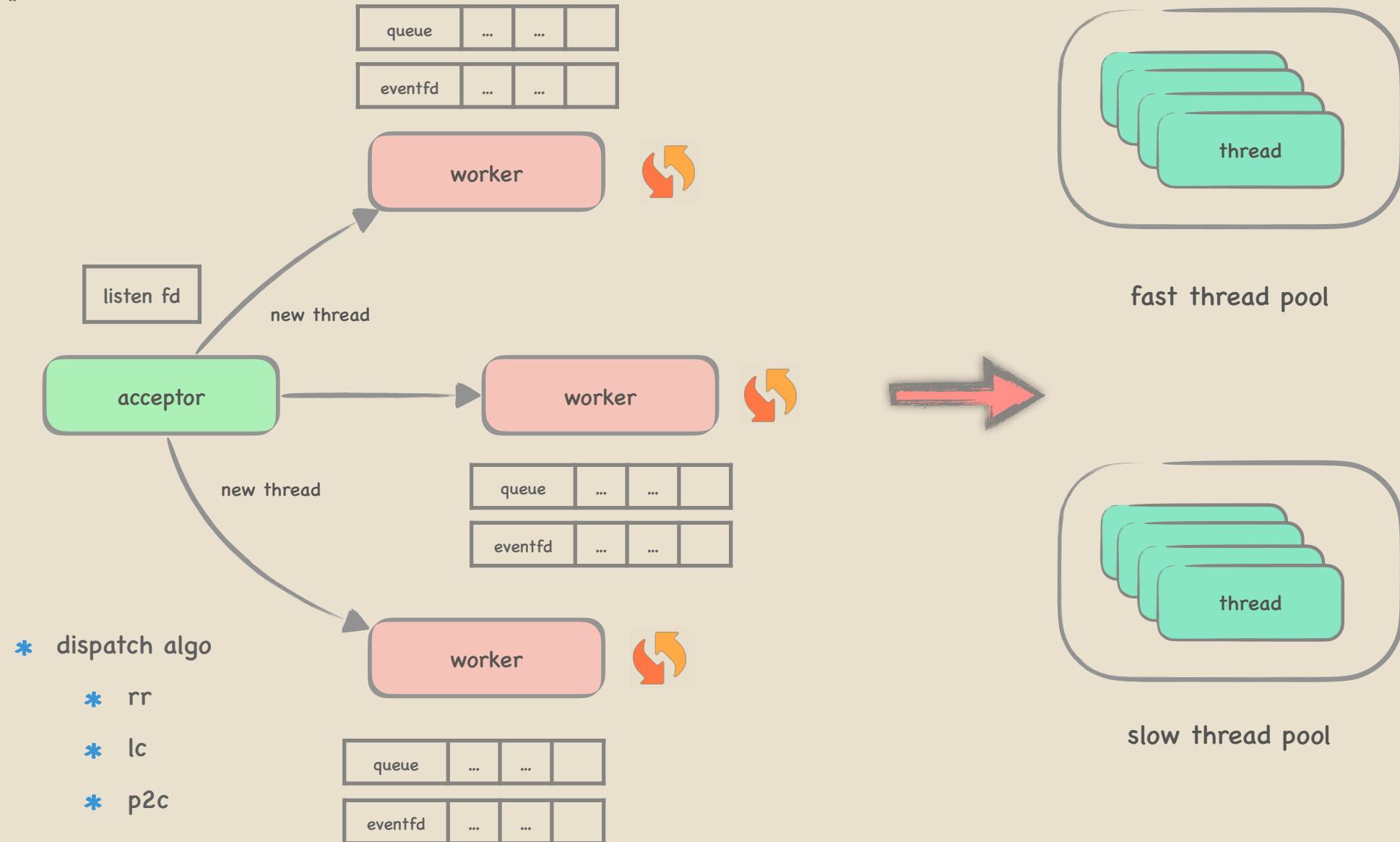
- \* java netty
- \* c++ muduo
- \* golang gnet
- \* golang gev
- \* ...



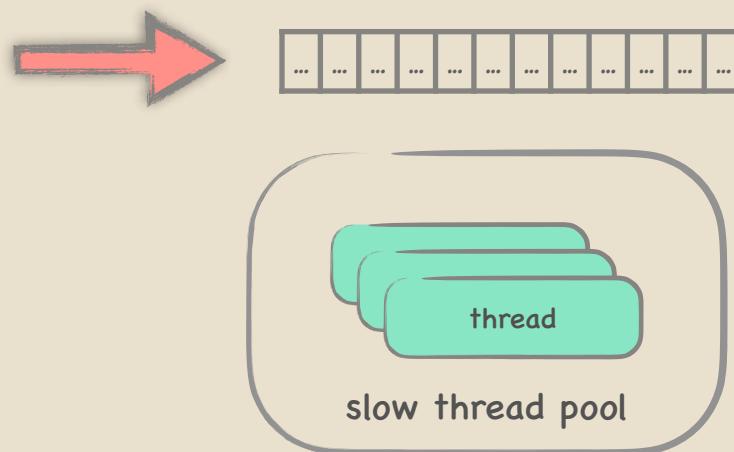
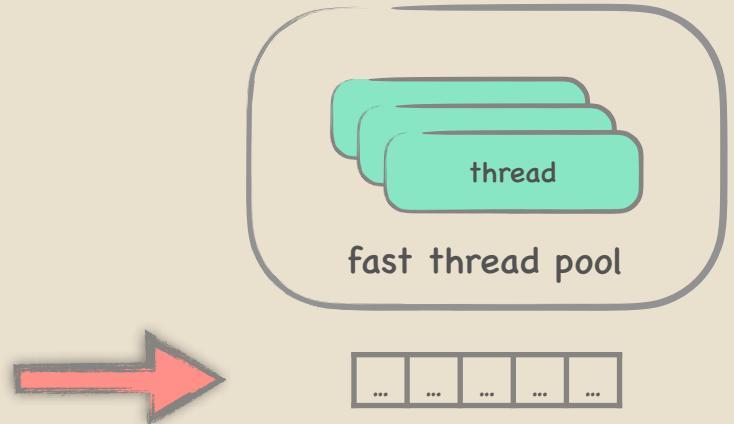
# reactor

- \* OnConnection()
- \* OnMessage()
- \* OnClose()

connection  
connection  
connection  
connection  
connection



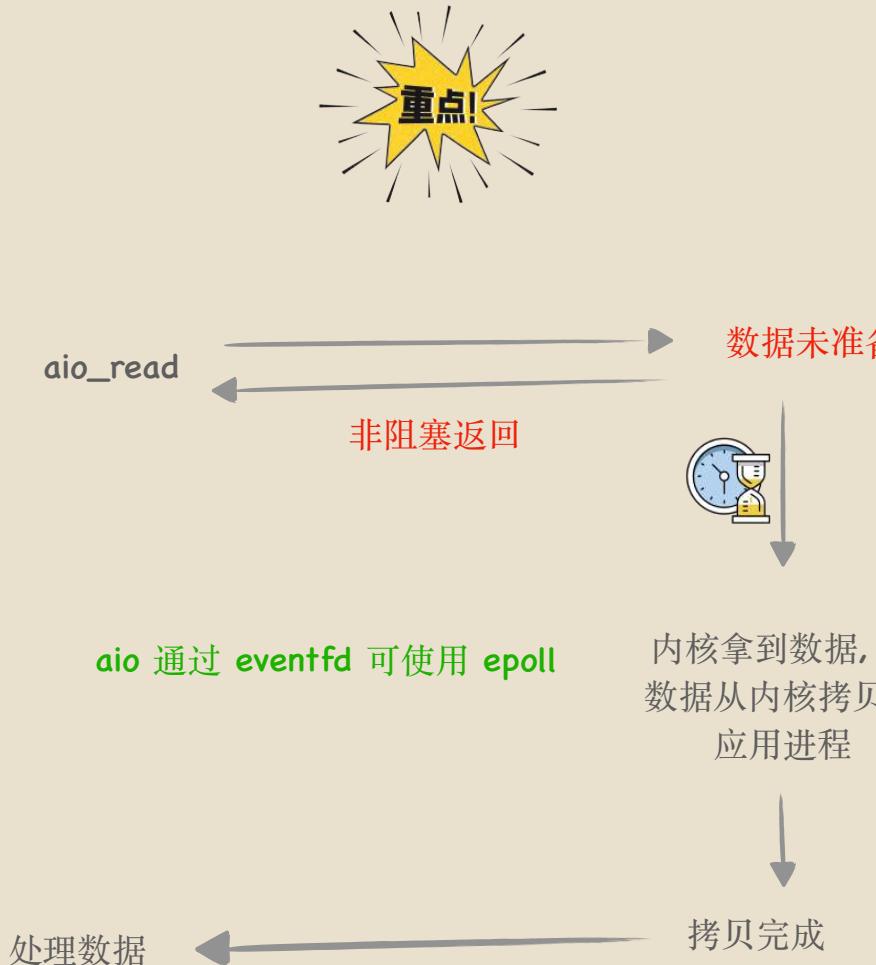
# 思考ing



- \* 😅 如果业务逻辑是阻塞的, 怎么处理 ?
- \* http request upstream backend
- \* query mysql, redis, http api
- \* block queue
- \* mutex / semaphore
- \* disk io
- \*

根据业务扔到 快/慢 线程池

# proactor



当发起 `aio_read` (异步 I/O) 之后, 就立即返回, 事件到来后, 内核自动将数据从内核空间拷贝到用户空间, 应用程序并不需要主动发起拷贝动作 !!!

- \* **Reactor**, 同步非阻塞同步网络模式, 感知的是就绪可读写事件
- \* **Proactor**, 异步非阻塞模式, 感知的是已完成的读写事件

基于 AIO 模型 !!!

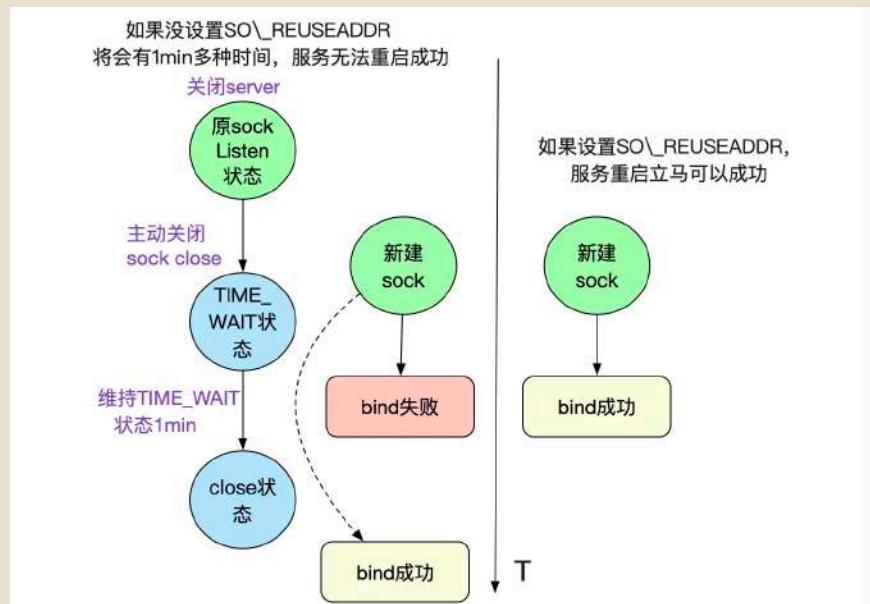
# proactor



linux aio 很不靠谱 !!!

- \* 为什么在linux都在使用 reactor 架构 😅
- \* 不能实现 socket aio, 只能对文件进行 aio
- \* 文件也只能是 direct io
- \* windows/darwin 实现系统级别的 aio, 可实现高性能 proactor 架构

# so\_reuseaddr



用于对处在 time-wait 状态下的 socket 进行重新bind

- \* 比如服务端进程主动`close`退出, 那么相关的连接被置为 `time-wait` 状态
- \* `socket` 对处于 `time-wait` 的套接字进行`bind`时, 异常报错.
- \* 错误: “`Address already in use`”

```

def main(self):
    server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)

    server_sock.bind(('localhost', 8080))
    server_sock.listen(self.backlog_size)
    ...

```

# so\_reuseport

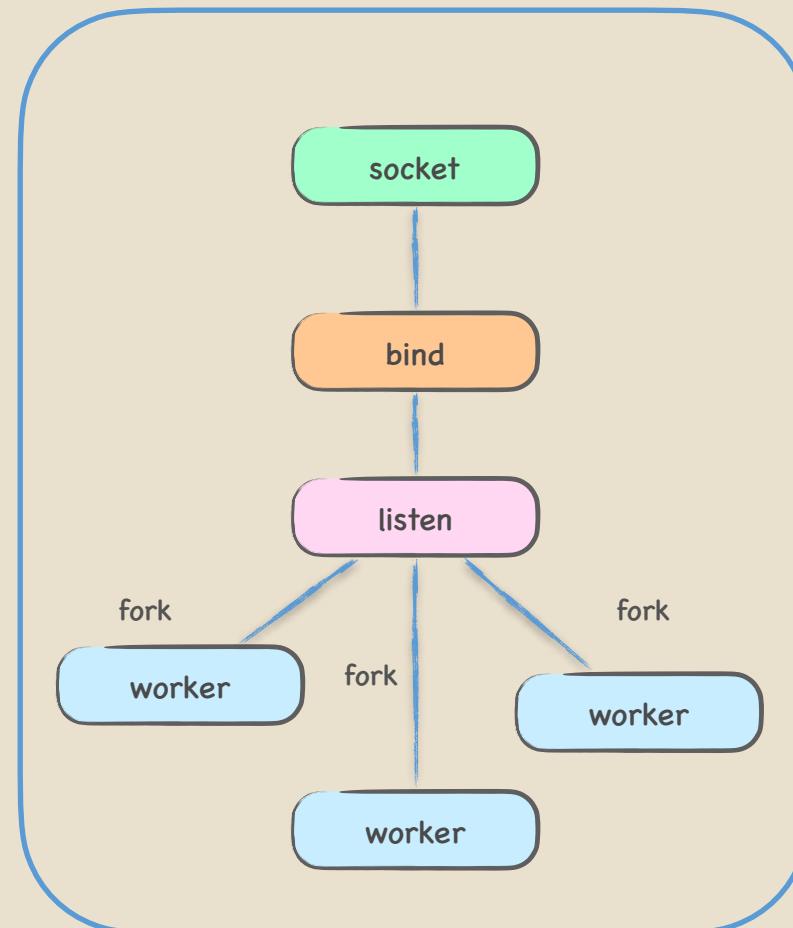
## \* SO\_REUSEPORT

- \* 支持多个进程或者线程绑定到同一端口
- \* 避免多个子进程竞争 listen fd (惊群) ?
- \* 内核层面实现 listen worker 的负载均衡
- \* 更容易实现平滑升级

```
● ● ●
def main(self):
    server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)

    server_sock.bind(('localhost', 8080))
    server_sock.listen(self.backlog_size)
    ...

```



创建新进程且 listen



创建新进程且 listen

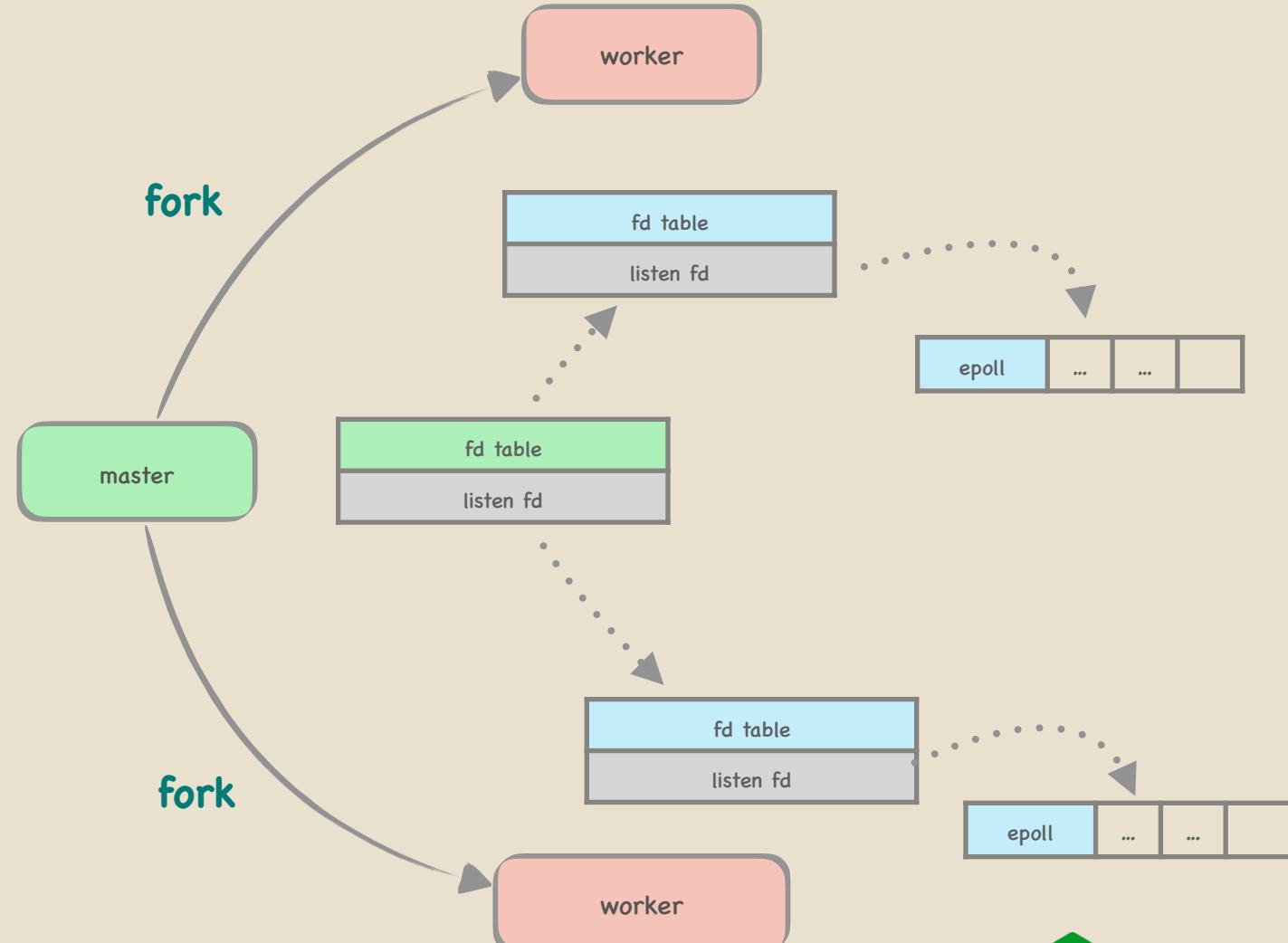


# 惊群问题

- \* 什么是惊群?

- \* 同一个事件通知给多人，类似观察者模式
- \* 造成无效唤醒线程，无故上下文切换
- \* 多线程 accept 的惊群已在 kernel 2.6 解决！
- \* 多个 eventloop 同时监听一个 fd 造成 epoll 惊群

`accept` 是线程安全， 不会重复和丢事件！



# 惊群问题

- \* epoll listen fd ?

- \* “accept\_mutex”，参考 nginx 的做法

- \* reuseport 端口复用模式，每个worker都实例化监听 listen fd

- \* epoll exclusive on Linux 4.5 +

- \* 不够均衡

- \* 内核总是唤醒等待队列的第一个 worker 进程

- \* reactor 网络模型

```
尝试获取accept锁

if 获取成功:
    在epoll中注册 listen fd
else:
    在epoll中取消监听 listen fd

处理所有事件 || 触发超时
释放accept锁
```

```
http {
    server {
        listen 80 reuseport;
        server_name localhost;
        # ...
    }
}

stream {
    server {
        listen 81 reuseport;
        # ...
    }
}
```

```
m_epollfd = epoll_create(0);

ev.events = EPOLLIN|EPOLLEXCLUSIVE;
ev.data.fd = m_listenfd;

if(epoll_ctl(m_epollfd, EPOLL_CTL_ADD, m_listenfd, &ev) < 0)
{
    cout << "add listen socket to epoll err" << endl;
    exit(EXIT_FAILURE);
}
```

```

for(;;)
{
    nfds = epoll_wait(epfd, events, 20, 500);
    for(i = 0; i < nfds; ++i)
    {
        if(events[i].data.fd == listenfd) {
            printf("accept connection, fd is %d\n", listenfd);
            ...
        }
        else if(events[i].events & EPOLLOUT) {
            ...
        }
        else if(events[i].events & EPOLLIN)
        {
            sockfd = events[i].data.fd;
            n = read(sockfd, line, MAXLINE);
            if(n == 0)
            {
                printf("%s\n", "客户端断开连接。");
                epoll_ctl(epfd, EPOLL_CTL_DEL, sockfd, &ev);
                continue;
            }

            if(n > 0)
            {
                printf("读到 %d 字节数据", n);
                continue;
            }

            if (n < 0 && (errno == EAGAIN))
            {
                printf("%s\n", "没有更多的数据可以读了");
                continue;
            }

            printf("读异常, 误代码为%d\n", errno);
            epoll_ctl(epfd, EPOLL_CTL_DEL, sockfd, &ev);
            ...
        }
    }
}

```

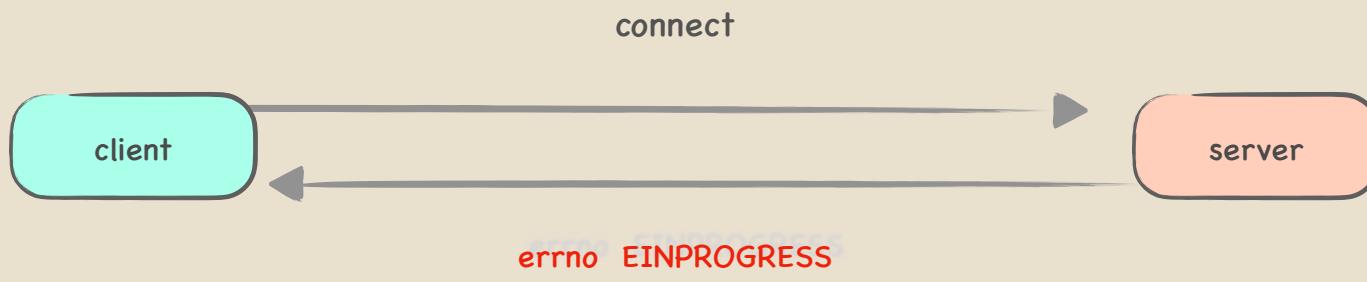
# 读写返回值



## \* Errno

- \* **EINTR**, 信号中断了系统调用
  - \* **EAGAIN**, 继续监听事件
  - \* **EWOULDBLOCK**, 同上（老系统接口有差异）
  - \* **ECONNRESET**, `tcp rst`连接重置
  - \* **EPIPE**, 管道破裂
  - \* **ETIMEDOUT**, 连接超时
  - \* **EINPROGRESS**, 正在连接
  - \* ...
- \*** N
- \* **> 0**, 说明读到了N个字节的数据
  - \* **= 0**, 说明对端发起关闭
  - \* **-1**, 返回 `errno` 错误

# 异步建立连接



- \* socket 配置非阻塞, 发起 connect 连接
- \* 如 ret 返回0表示连接成功, 如 -1 且 errno = EINPROGRESS 异步连接中
- \* 把该 fd 加入select或者epoll关心的 EPOLLOUT 事件
- \* 写事件触发后, 需调用getsockopt去获取status, 如果等于0说明非阻塞connect成功了, 不为0那么连接异常, 比如端口没开, ip找不到。

```
# 异步建立连接
int res = connect(fd, ...);
if (res < 0 && errno != EINPROGRESS) {
    // error, fail somehow, close socket
    return;
}

if (res == 0) {
    // connection has succeeded immediately
} else {
    // connection attempt is in progress
}

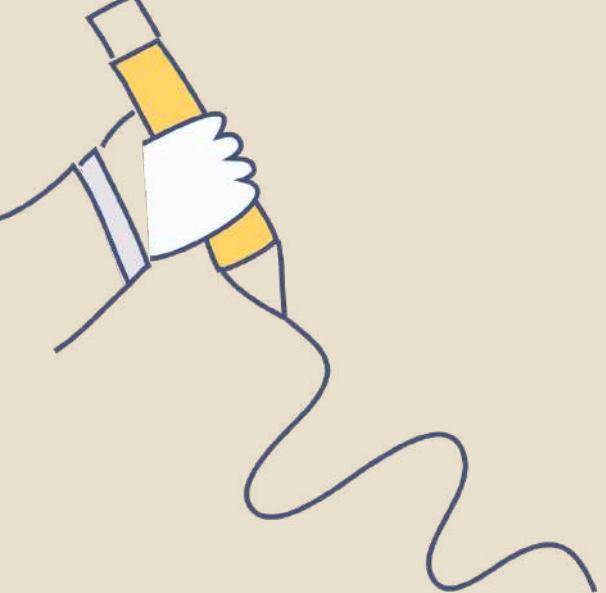
# 加入 EPOLLOUT 事件
ev.events = EPOLLOUT|EPOLLET;
ev.data.fd = fd;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev) == -1 ) {
    perror("epoll_ctl error");
}

# 监听事件并且getsockopt判断成功.
for (;;) {
    nevents = epoll_wait(epfd, events, EPOLL_MAXEVENTS, -1);
    if (nevents < 0) {
        perror("epoll_wait failed");
    }

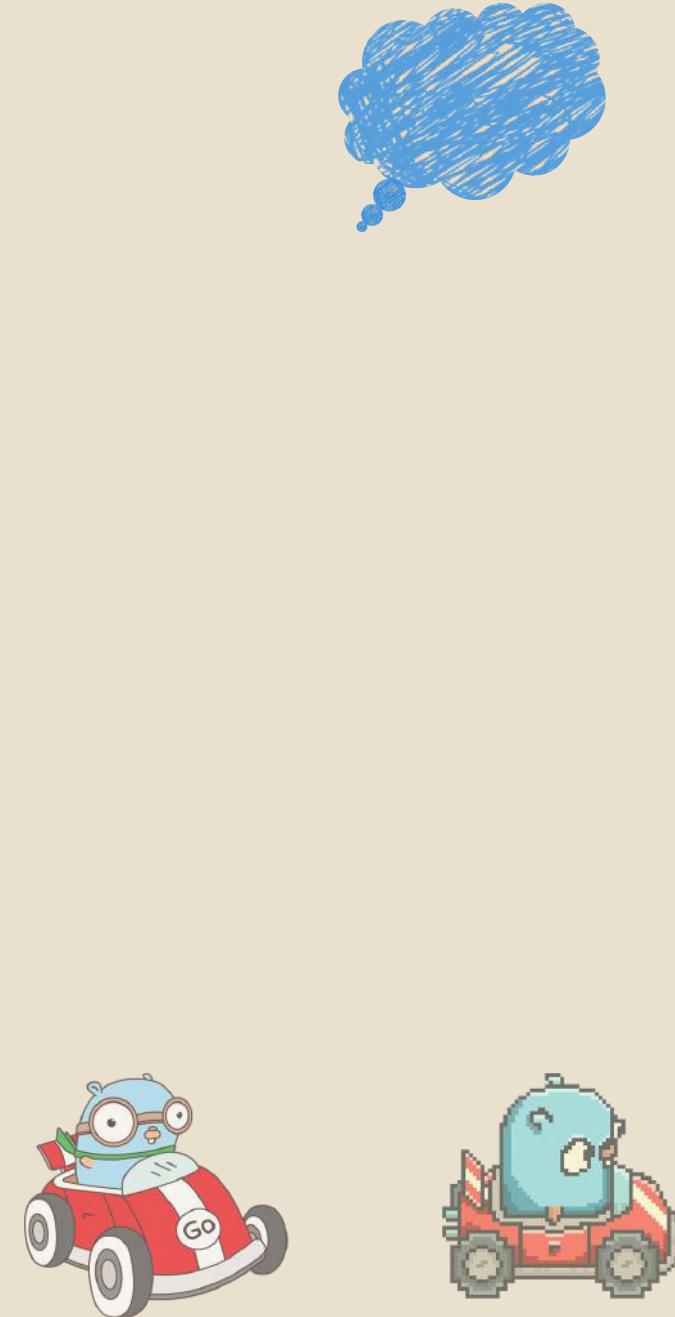
    for (i = 0; i < nevents; i++) {
        if (events[i].data.fd == fd) {
            int result;
            socklen_t result_len = sizeof(result);
            if (getsockopt(fd, SOL_SOCKET, SO_ERROR, &result, &result_len) < 0) {
                perror("getsockopt");
                return;
            }

            if (result != 0) {
                perror("connection failed; error code is in 'result'");
                return;
            }

            // result == 0
            // 连接成功了, socket 可读可写.
        }
    }
}
```



## 常见问题



# tcp/udp 下的粘包问题



- \* 怎么产生的？
- \* 如何解决粘包半包问题？
- \* udp 是否有粘包问题？



# 粘包半包

😡 这是第一个请求.

😎 这是第二个请求.

😡 这 是第一 个请求. 😎 这 是 第二 个请求.

\* 半包, 一条数据, 被拆分发送, 对端分了多次读完

\* 粘包, 多条数据, 被对方一下都读出来了

\* 原因多种多样

- \* mtu/mss (1500/1460)
- \* nagle

\* rto / ofo

\* network delay

\* 分多次来 `socket.send()`

\* 接收端读的太快或太慢

\* ...

tlv

# 粘包半包

Type	Length	Value	Crc

```

● ● ●

# redis
# set mykey myvalue
"*3\r\n$3\r\nSET\r\n$5\r\nmykey\r\n$7\r\nmyvalue\r\n"

# http length
HTTP/1.1 200 OK
Content-Length: 1111
Connection: keep-alive
Server: Apache

# http chunked
HTTP/1.1 200 OK
Transfer-Encoding: chunked
5\r\n
Media\r\n
8\r\n
Services\r\n
4\r\n
Live\r\n
0\r\n
\r\n

```

```

type Package struct {
    Version      [2]byte // 协议版本
    Length       int16   // 数据部分长度
    Timestamp    int64   // 时间戳
    HostnameLength int16  // 主机名长度
    Hostname     []byte  // 主机名
    TagLength    int16   // Tag长度
    Tag          []byte  // Tag
    Msg          []byte  // 数据部分长度
}

func (p *Package) Pack(writer io.Writer) error {
    var err error
    err = binary.Write(writer, binary.BigEndian, &p.Version)
    err = binary.Write(writer, binary.BigEndian, &p.Length)
    err = binary.Write(writer, binary.BigEndian, &p.Timestamp)
    err = binary.Write(writer, binary.BigEndian, &p.HostnameLength)
    err = binary.Write(writer, binary.BigEndian, &p.Hostname)
    err = binary.Write(writer, binary.BigEndian, &p.TagLength)
    err = binary.Write(writer, binary.BigEndian, &p.Tag)
    err = binary.Write(writer, binary.BigEndian, &p.Msg)
    return err
}

func (p *Package) Unpack(reader io.Reader) error {
    var err error
    err = binary.Read(reader, binary.BigEndian, &p.Version)
    err = binary.Read(reader, binary.BigEndian, &p.Length)
    err = binary.Read(reader, binary.BigEndian, &p.Timestamp)
    err = binary.Read(reader, binary.BigEndian, &p.HostnameLength)
    p.Hostname = make([]byte, p.HostnameLength)
    err = binary.Read(reader, binary.BigEndian, &p.Hostname)
    err = binary.Read(reader, binary.BigEndian, &p.TagLength)
    p.Tag = make([]byte, p.TagLength)
    err = binary.Read(reader, binary.BigEndian, &p.Tag)
    p.Msg = make([]byte, p.Length-8-2-p.HostnameLength-2-p.TagLength)
    err = binary.Read(reader, binary.BigEndian, &p.Msg)
    return err
}

```

golang pack/unpack

udp 没有粘包半包的风险 !!!

# 粘包半包



- \* udp 面向消息通信, 有保护消息边界
- \* 内核协议栈不会对 udp 做消息合并处理
- \* 发送端发了 10 次消息, 接收端就需要读10次
- \* udp 最大包不能超过 65507 byte
- \* 建议把 udp 拆分成小包, 规避 IP 层的分包重组
  - \* 包太大被 mtu 分包, 多个包中有一个丢包, 其他包被接收层丢弃 !!!
  - \* 更好的适配三方的可靠性传递 (kcp, quic)
  - \* 大包在网络中太危险, 容易被过滤
- \* 建议对 udp 做 tlv 处理, 业务上更可控 !!!

# 粘包半包

http 是如何识别一个完整的请求 ?

```
$ curl -i https://www.baidu.com
HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: private, no-cache, no-store, proxy-revalidate, no-transform
Connection: keep-alive
Content-Length: 2443
Content-Type: text/html
Date: Wed, 29 Nov 2023 09:19:24 GMT
Etag: "588603eb-98b"
Last-Modified: Mon, 23 Jan 2017 13:23:55 GMT
Pragma: no-cache
Server: bfe/1.0.8.18
Set-Cookie: BDORZ=27315; max-age=86400; domain=.baidu.com; path=/
```

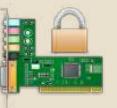
```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked
7\r\n
Mozilla\r\n
11\r\n
Developer Network\r\n
0\r\n
\r\n
```



# 协议栈和应用层心跳

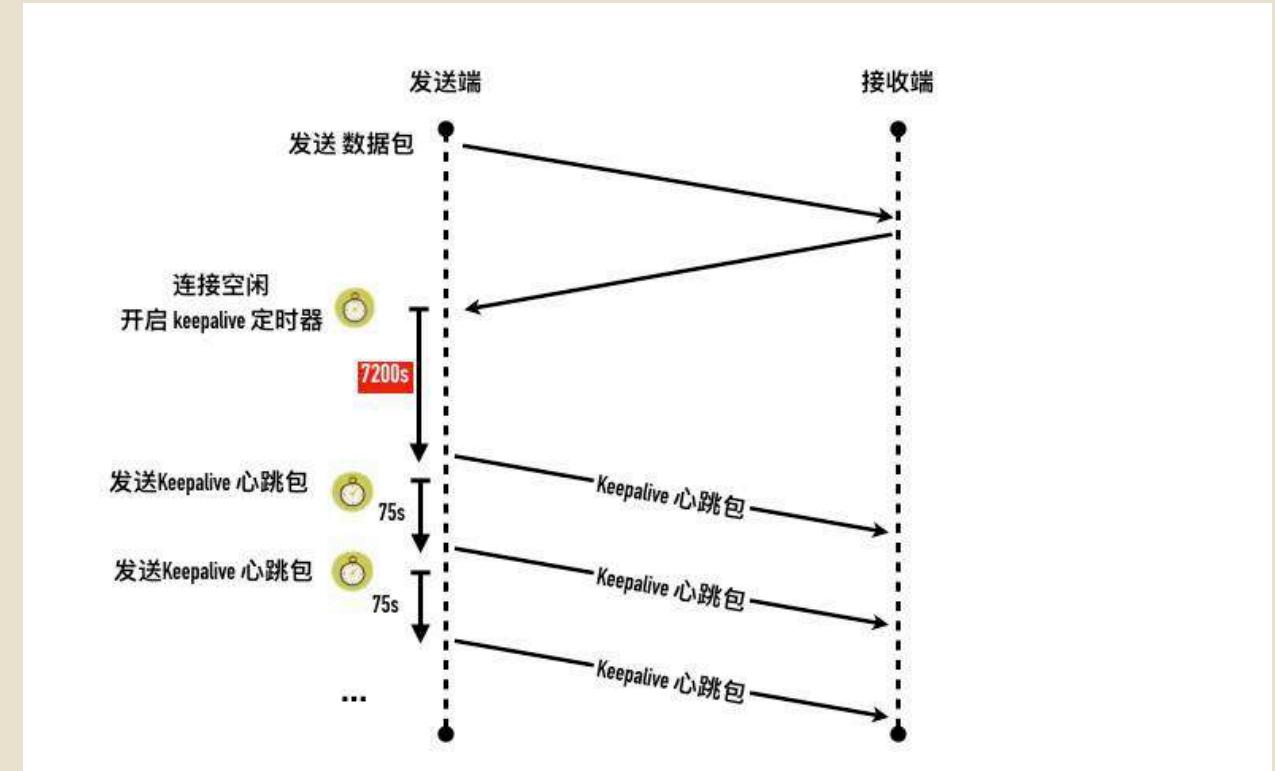


- \* 为什么要自定义 `socket` 心跳
- \* `golang` 的心跳的默认值及实现
- \* `tcp keepalive` 心跳包是个什么样子？
- \* 应用层心跳的优缺点



# 默认心跳配置

- \* sysctl -a
- \* net.ipv4.tcp\_keepalive\_time = 7200
  - \* 空闲7200秒后才开启保活检测
- \* net.ipv4.tcp\_keepalive\_intvl = 75
  - \* 每次间隔 75s
- \* net.ipv4.tcp\_keepalive\_probes = 9
  - \* 共尝试9次
- \* 最少需要 **2 小时 11 分 15 秒** 才可判定连接死亡！



nima, 时间太长了 !!!

# 如何设置心跳

- \* 直接调整 `sysctl` 的 `keepalive` 时间间隔；
- \* 在服务端对每个 `socket opt` 配置心跳；
- \* 在应用层自定义心跳；

三种设置的方法 !!!

```
import socket

server = socket.socket()
server.bind(('localhost', 9999))
server.listen(5)

while True:
    client, addr = server.accept()
    client.setsockopt(socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1)
    client.setsockopt(socket.SOL_TCP, socket.TCP_KEEPIDLE, 120)
    client.setsockopt(socket.SOL_TCP, socket.TCP_KEEPCNT, 9)
    client.setsockopt(socket.SOL_TCP, socket.TCP_KEEPINTVL, 20)
```

```
09:45:53.433577 IP 127.0.0.1.9999 > 127.0.0.1.41834: Flags [.], ack 1, win 512, options [nop,nop,TS val 3173718853 ecr 3173598853], length 0
09:45:53.433600 IP 127.0.0.1.41834 > 127.0.0.1.9999: Flags [.], ack 1, win 512, options [nop,nop,TS val 3173718853 ecr 3173598853], length 0
09:47:53.433582 IP 127.0.0.1.9999 > 127.0.0.1.41834: Flags [.], ack 1, win 512, options [nop,nop,TS val 3173838853 ecr 3173718853], length 0
09:47:53.433614 IP 127.0.0.1.41834 > 127.0.0.1.9999: Flags [.], ack 1, win 512, options [nop,nop,TS val 3173838853 ecr 3173598853], length 0
09:49:53.433576 IP 127.0.0.1.9999 > 127.0.0.1.41834: Flags ...
```

# golang 心跳的配置

```

12 12     "time"
13 13 )
14 14
15 + // defaultTCPKeepAlive is a default constant value for TCPKeepAlive times
16 + // See golang.org/issue/31510
17 + const (
18 +     defaultTCPKeepAlive = 15 * time.Second
19 + )
20 +
21 // A Dialer contains options for connecting to an address.
22 //
23 // The zero value for each field is equivalent to dialing
@@ -425,7 +431,7 @@ func (d *Dialer) DialContext(ctx context.Context, network, address string) (Conn
425 431         setKeepAlive(tc.fd, true)
426 432         ka := d.KeepAlive
427 433         if d.KeepAlive == 0 {
428 -             ka = 15 * time.Second
434 +             ka = defaultTCPKeepAlive
429 435     }
430 436     setKeepAlivePeriod(tc.fd, ka)

```

- \* golang 默认对所有 socket 配置 15s 的心跳
- \* 使用 socket setsockopt 配置

```

[pid 29723] <... getsockname resumed> [socket@fd=14<-INET, SockType=SOCK_STREAM(5), ...
[pid 29723] <... setsockopt resumed>) = 0
[pid 29722] setsockopt(1917, SOL_TCP, TCP_KEEPIPDL, [15], 4 <unfinished ...>
[pid 29721] setsockopt(1677, SOL_TCP, TCP_KEEPIPDL, [15], 4 <unfinished ...>
[pid 29720] nanosleep({tv_sec=0, tv_nsec=20000}, <unfinished ...>
[pid 29719] setsockopt(190, SOL_TCP, TCP_KEEPINTVL, [15], 4 <unfinished ...>
[pid 29740] write(827, "Hello World.", 12 <unfinished ...>
[pid 29738] <... setsockopt resumed>) = 0
[pid 29736] setsockopt(1340, SOL_TCP, TCP_KEEPIPDL, [15], 4 <unfinished ...>
[pid 29735] getsockname(2262, <unfinished ...>

```

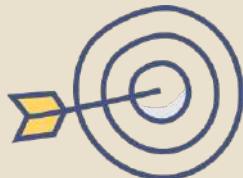


# 应用层心跳

## \* 为什么要应用层心跳？

- \* tcp keepalive 报文可能被设备特意过滤或屏蔽，如运营商设备；
- \* tcp keepalive 由内核帮忙回复，无法检测应用层状态，如进程阻塞、死锁、TCP缓冲区满等情况；
- \* 应用层的心跳包可以顺便携带业务数据；
- \* tcp keepalive 不方便判断连接被关闭的原因；

协议栈的心跳，应用层无法感知



# tcp time-wait 的那些事儿



\* 怎么产生的 timewait ?

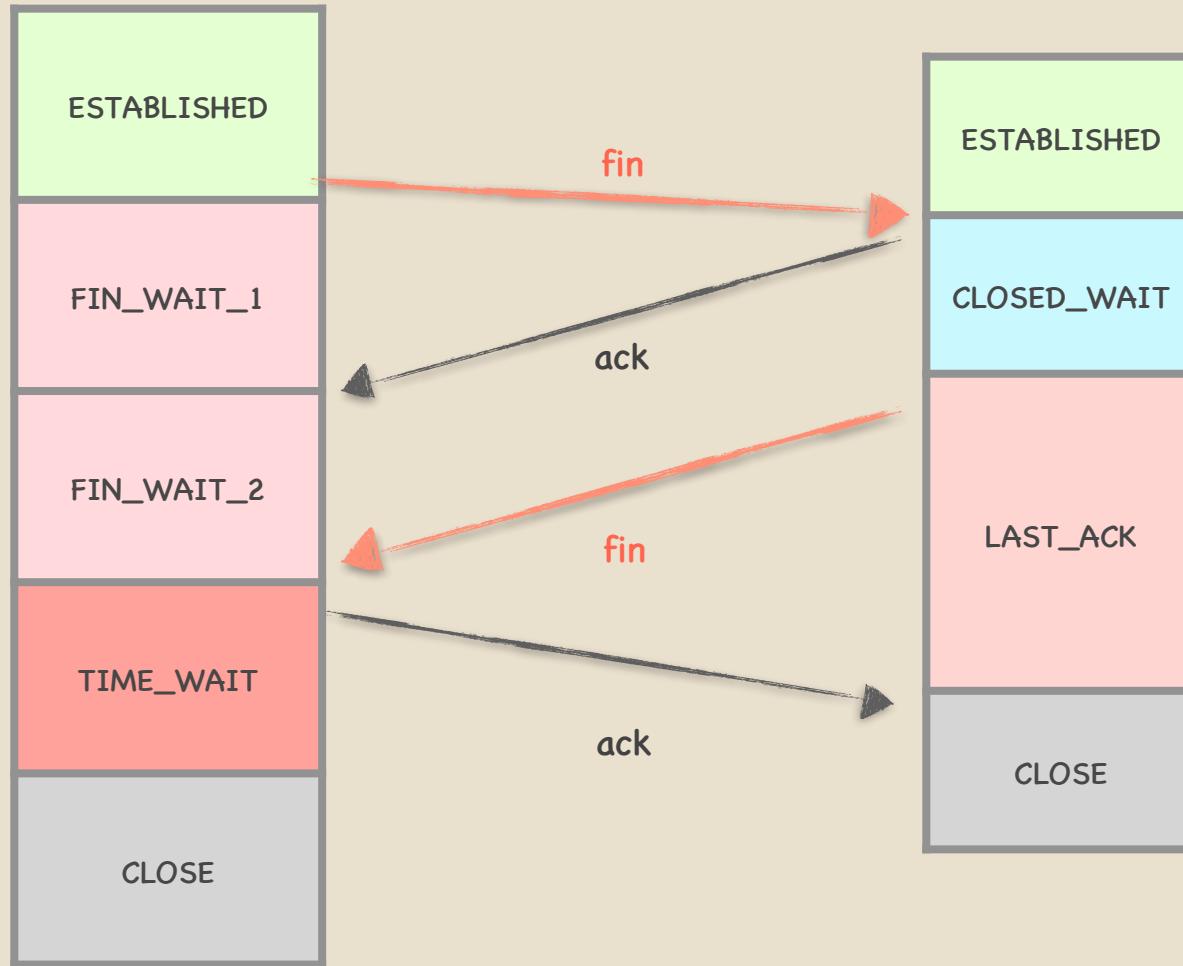
\* timewait 有什么危害 ?

\* 如何避免 timewait ?

\* 如何优化 timewait ?



# timewait



timewait 是怎么产生的 ?  
谁发起关闭，谁就会产生 timewait !!!

- \* RST
- \* enable socket so\_linger
- \* 关闭读缓冲不为空的 socket
- \* tcp\_max\_tw\_buckets
- \* . . .



# time-wait

- \* 短连接 ?
- \* 连接池爆满 ?
- \* 应用协议异常 ?
- \* 超过idle timeout ?
- \* ...

根本问题在于  
为什么有大量的time-wait  
!!!

- \* 为何跟 time-wait 过不去 ?
- \* socket占用缓冲区内存 ?
- \* 占用进程 RSS 内存 ?
- \* 占用 local\_range\_port ?
- \* 无法申请地址 ?



# time-wait 优化

- \* net.ipv4.tcp\_tw\_recycle = 1

  - \* 不要开, 不要开 , 不要开 !!!





  - \* nat环境有大问题 !!!

  - \* linux 4.12 会移除该配置 !!!

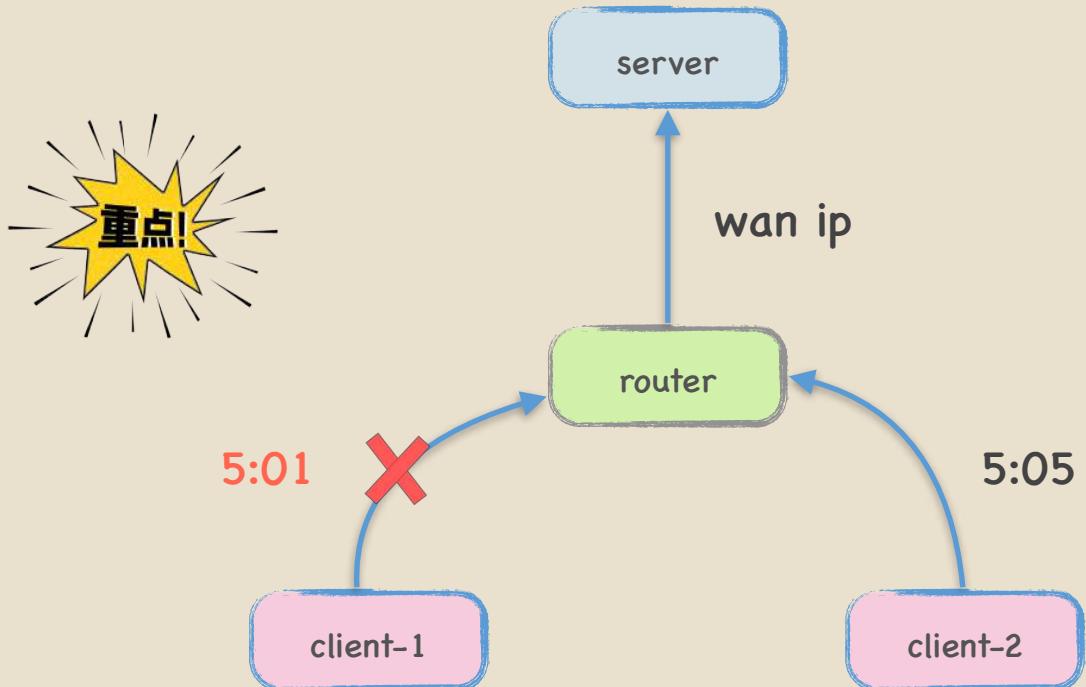
- \* tcp\_tw\_reuse=1

  - \* 安全选项

  - \* 一秒后复用tw状态的端口

- \* 适当优化 tcp\_max\_tw\_buckets

- \* setsocket(SO\_LINGER...)



client1时间晚于client2, client2 先于server 建联, client1会失败 !!!

server 开启 tw\_recycle 和 timestamp配置, 启用per-host的paws机制.

# 连接队列

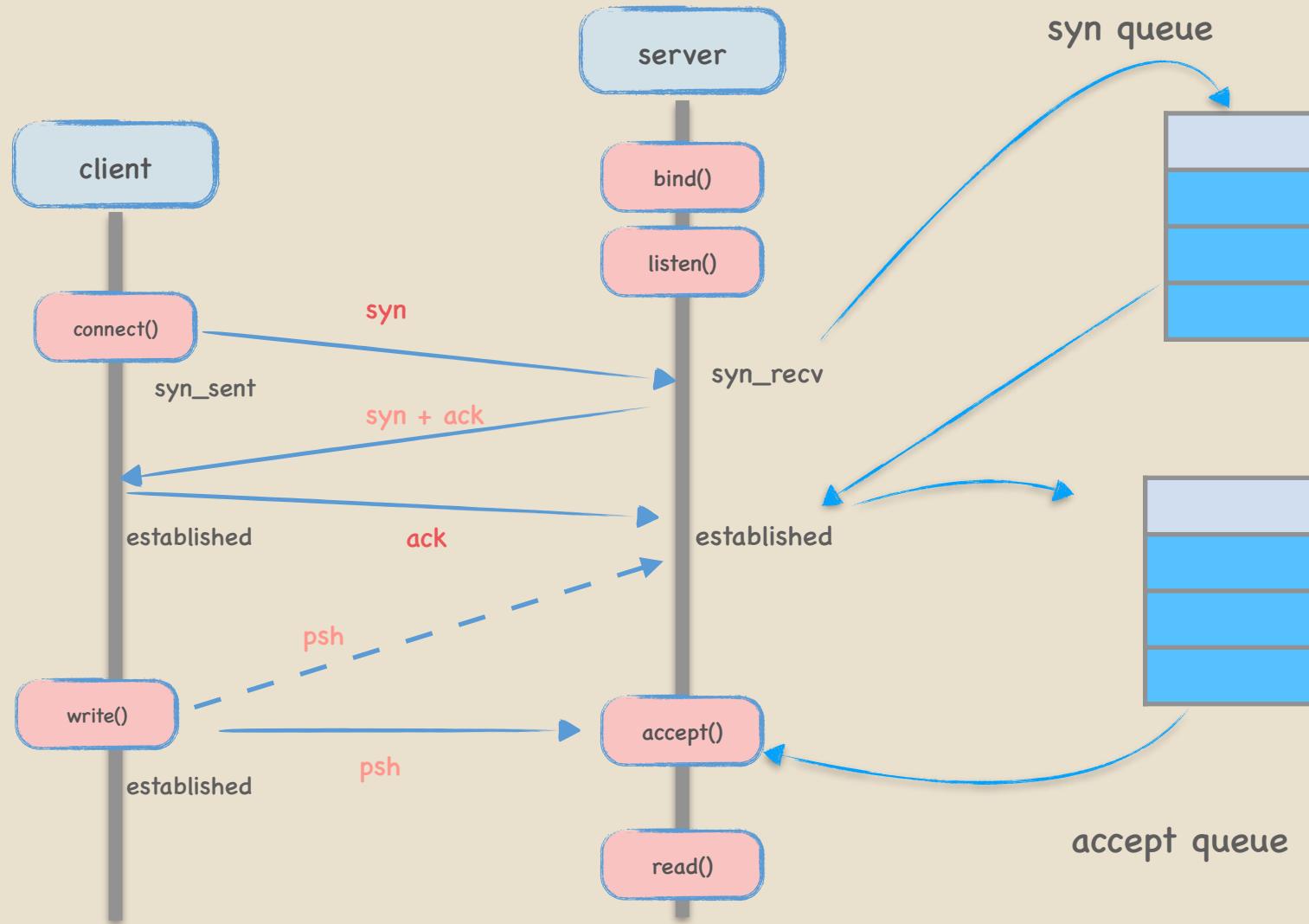


\* 半连接队列

\* 全连接队列



# 连接队列



\* 半连接队列

\* 收到第一个syn

\* 回复syn + ack

\* 全连接队列

\* 完成三次握手

\* 并未accept取走

# 连接队列

\* 半连接队列

\* `tcp_max_syn_backlog (1024)`

\* 其实跟三个值都有关系

\* `backlog`、`net.ipv4.tcp_max_syn_backlog`、  
`net.core.somaxconn`（内核代码有描述）

\* 全连接队列

\* `get_min_value`

\* `listen( backlog )`

\* `Nginx = 511`

\* `Redis = 511`

\* `net.core.somaxconn (128)`

\* `accept`原子消费且不惊群

\* 还在全队列的连接可接收数据

\* 超出则拒绝建连

```
$ ss -lnt  
# Redis  
State  Recv-Q Send-Q Local Address:Port Peer Address:Port  
LISTEN      0      511                      *:6379          *:*
```

# 连接队列

## \* `tcp_abort_on_overflow`

### \* `equal 0 (default)`

\* 半连接队列满了？

\* 忽略客户端syn, 服务端不做任何反应

\* 客户端不断重试发syn, 一直到 `tcp_syn_retries = 5`

\* 全连接队列满了？

\* 忽略客户端ack, 服务端设立定时器发送syn/ack

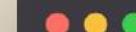
\* 一直到 `tcp_synack_retries = 5`

### \* `equal 1`

\* `retrun rst`



connection reset by peer  
&  
connection refused



```
$ netstat -s | egrep "overflowed|ignored"  
# 全连接  
820481 times the listen queue of a socket overflowed  
  
# 半连接  
820481 SYNs to LISTEN sockets ignored
```

# 为什么要使用可靠性 udp !!!

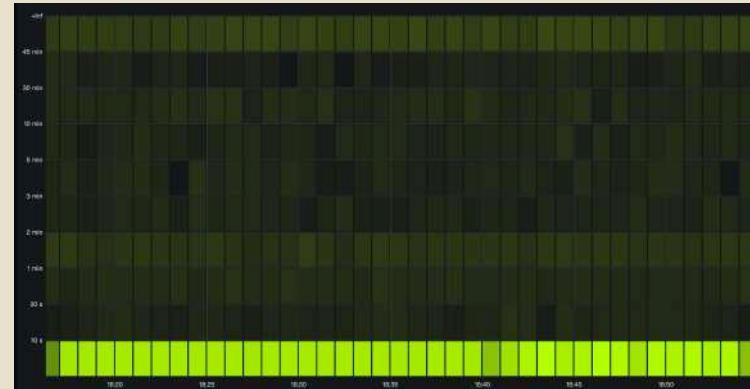


- \* 低延迟

- \* 低延迟

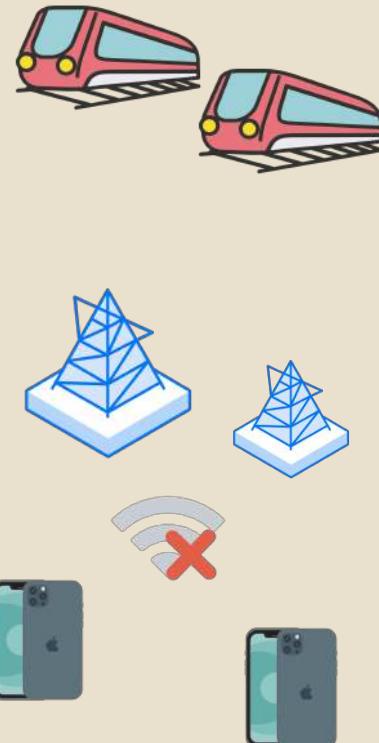
- \* 低延迟





# 移动网络

- \* 什么是弱网络 (通常说的是不稳定的移动网络)
  - \* 大量移动端接入引发拥塞
    - \* 基站拥塞
    - \* SGW拥塞
  - \* 拥塞就丢包, 丢包则引发 RTT 时延飙高
  - \* 距离基站中心较远, 时延也会高
  - \* 连接时不时被中断, 应用挂起
    - \* 切后台
    - \* 关屏幕



- \* 手机移动网络什么时候更换IP ?
- \* 手动关闭移动网络
- \* 开关飞行模式
- \* 去了其他大城区和跨市
- \* 触发了 IP TTL 的时间, 可到一天半左右

IP 不会因为高速行驶中频繁切换 IP !!!  
( 手机到基站之间是链路层)

```

09:31 18:47
Ping 启动
服务器
oss-cn-shanghai.aliyuncs...
输出信息:
64 bytes from 106.14.228.198: icmp_seq=0 ttl=32 time=914.194 ms
64 bytes from 106.14.228.198: icmp_seq=1 ttl=32 time=890.287 ms
64 bytes from 106.14.228.198: icmp_seq=2 ttl=32 time=897.894 ms
64 bytes from 106.14.228.198: icmp_seq=3 ttl=32 time=840.376 ms
64 bytes from 106.14.228.198: icmp_seq=4 ttl=32 time=149.259 ms
64 bytes from 106.14.228.198: icmp_seq=5 ttl=32 time=96.094 ms
64 bytes from 106.14.228.198: icmp_seq=6 ttl=32 time=346.185 ms
64 bytes from 106.14.228.198: icmp_seq=7 ttl=32 time=571.206 ms
--- 106.14.228.198 ping statistics ---
9 packets transmitted, 8 received, 11.11% packet loss
round-trip min / avg / max = 96.094 / 522.833 / 914.194 ms

```

```

18:47
返回 Ping 启动
服务器
oss-cn-shanghai.aliyuncs...
输出信息:
Request timeout for icmp_seq 0
64 bytes from 106.14.228.198: icmp_seq=1 ttl=32 time=2021.669 ms
Request timeout for icmp_seq 2
Request timeout for icmp_seq 3
Request timeout for icmp_seq 4
64 bytes from 106.14.228.198: icmp_seq=5 ttl=32 time=2295.966 ms
64 bytes from 106.14.228.198: icmp_seq=6 ttl=32 time=1056.658 ms
64 bytes from 106.14.228.198: icmp_seq=7 ttl=32 time=57.947 ms
64 bytes from 106.14.228.198: icmp_seq=8 ttl=32 time=1145.334 ms
64 bytes from 106.14.228.198: icmp_seq=9 ttl=32 time=94.308 ms
64 bytes from 106.14.228.198: icmp_seq=10 ttl=32 time=1225.027 ms
64 bytes from 106.14.228.198: icmp_seq=11 ttl=32 time=604.421 ms
64 bytes from 106.14.228.198: icmp_seq=12 ttl=32 time=1343.545 ms

```

# 为什么要重视丢包？



\* 一旦发生丢包 ...

\* 发送的窗口数降低

\* 时延增长



# 拥塞控制

- \* 慢启动

- \* 每收到一个ack,  $cwnd = cwnd + 1$

- \* 每一轮RTT,  $cwnd = cwnd \times 2$

- \* 拥塞避免

- \*  $cwnd > ssthresh$  使用拥塞避免算法

- \* 每一轮RTT,  $cwnd = cwnd + 1$

- \* 拥塞发生

- \* 超时重传

- \*  $threshold = cwnd/2$ ,  $cwnd = 1$

- \* 进入慢启动阶段

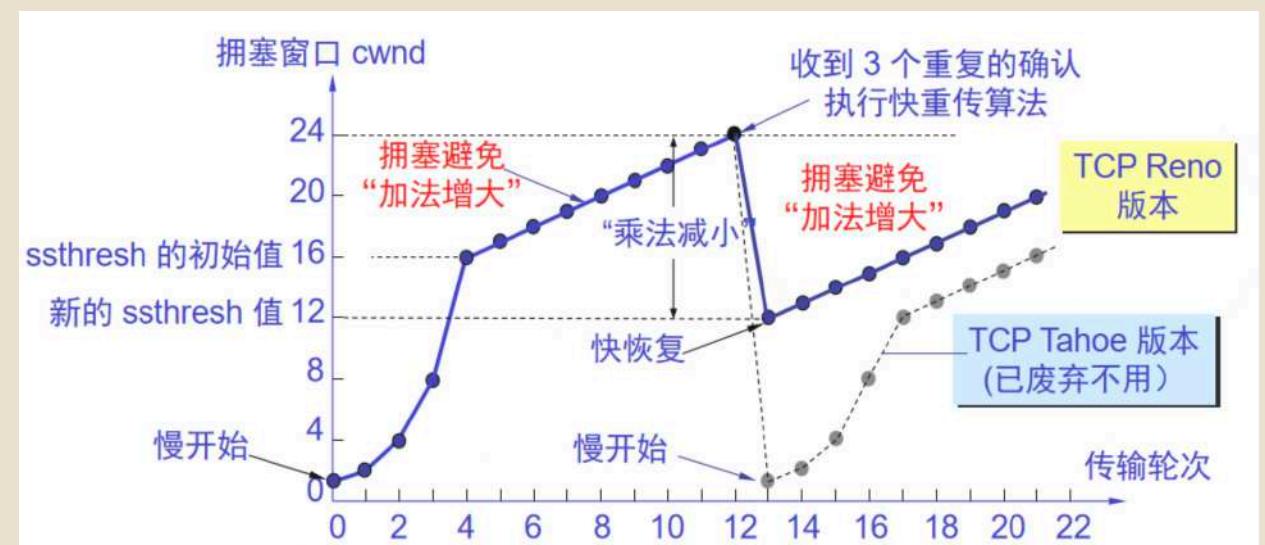
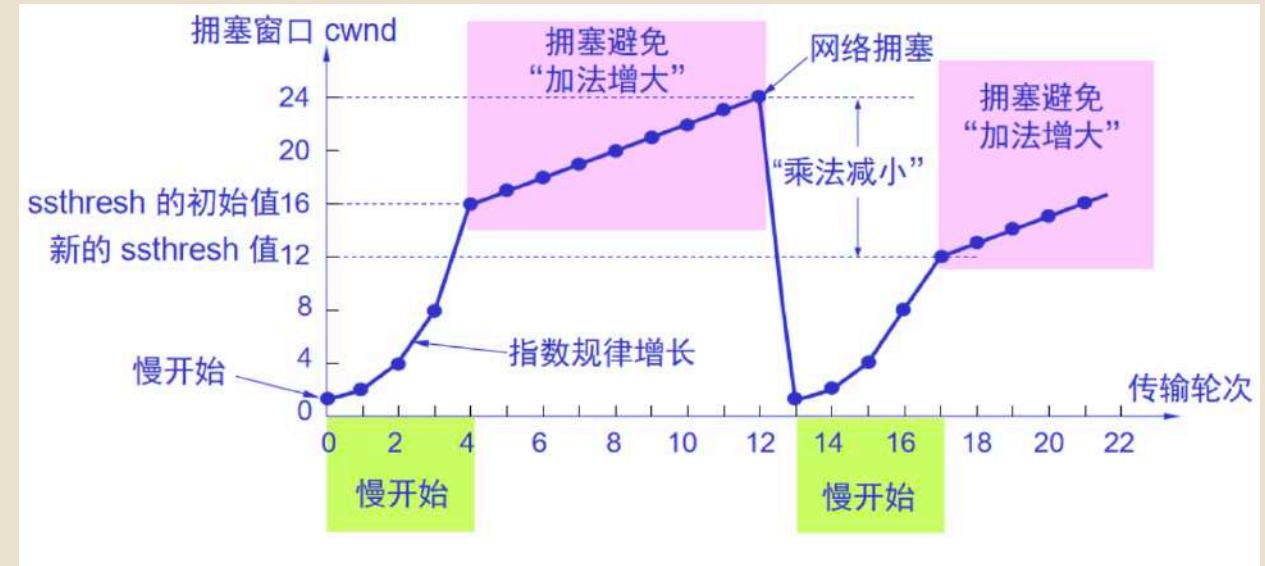
- \* 快速重传

- \*  $cwnd = cwnd / 2$ ,  $threshold = cwnd$

- \* 进入快速恢复

- \* 快速恢复

- \*  $cwnd = threshold + 3 MSS$



# kcp

- \* 基于udp实现的快速可靠传输协议 !!!
- \* 更加激进贪婪, 牺牲网络的公平性
- \* 特点
  - \* 比 TCP浪费10%-20%的带宽的代价
  - \* 换取平均延迟降低 30%-40%
  - \* 且最大延迟降低三倍的传输效果 !!!

## \* 相比tcp



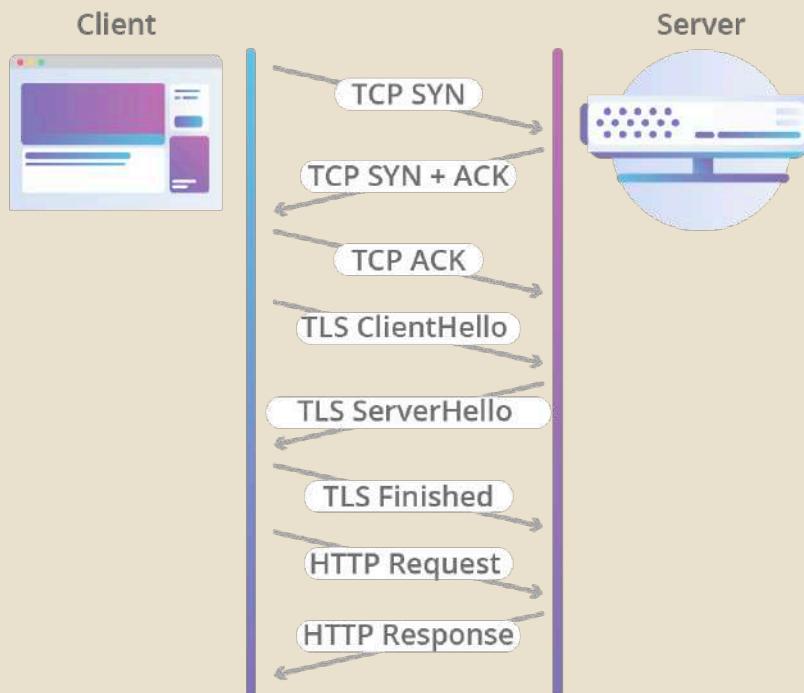
- \* 无需三次握手和四次挥手
- \* RTO不翻倍, 只做 1.5 倍
- \* 快速重传 ( 收到2个重复ack )
- \* FEC (Reed-Solomon纠删码)
- \* 多路复用
- \* ...

使用udp实现可靠性协议

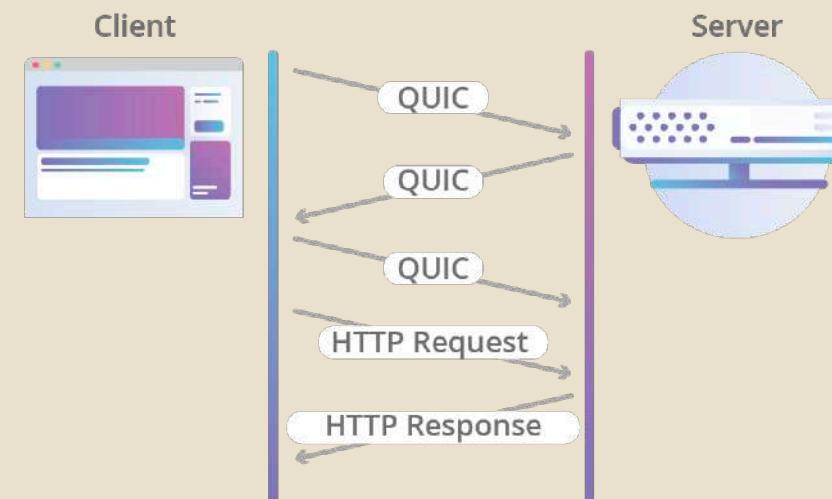


# protocol over quic

HTTP Request Over TCP + TLS



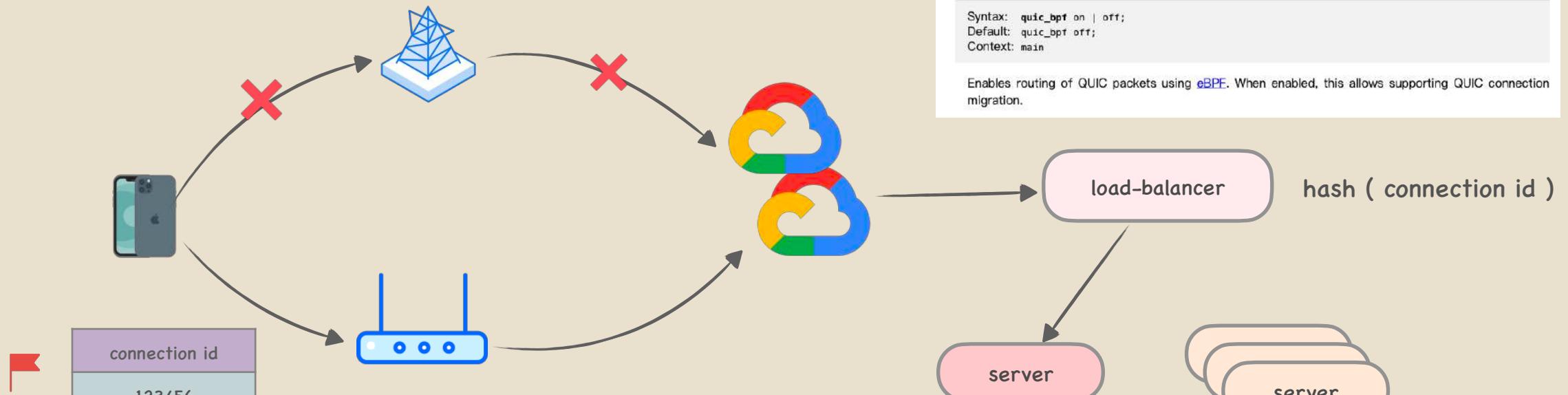
HTTP Request Over QUIC



- \* 基于 UDP 通信
- \* 用户态协议（拥塞, 流量控制）
- \* 支持 0-RTT
- \* FEC 前向纠删码
- \* 连接迁移
- \* 解决线头阻塞问题 HOL



# connection migration



Join the [NGINXCommunity Slack](#) to ask and answer questions, discuss NGINX, and share useful adv...

**Support for QUIC and HTTP/3**

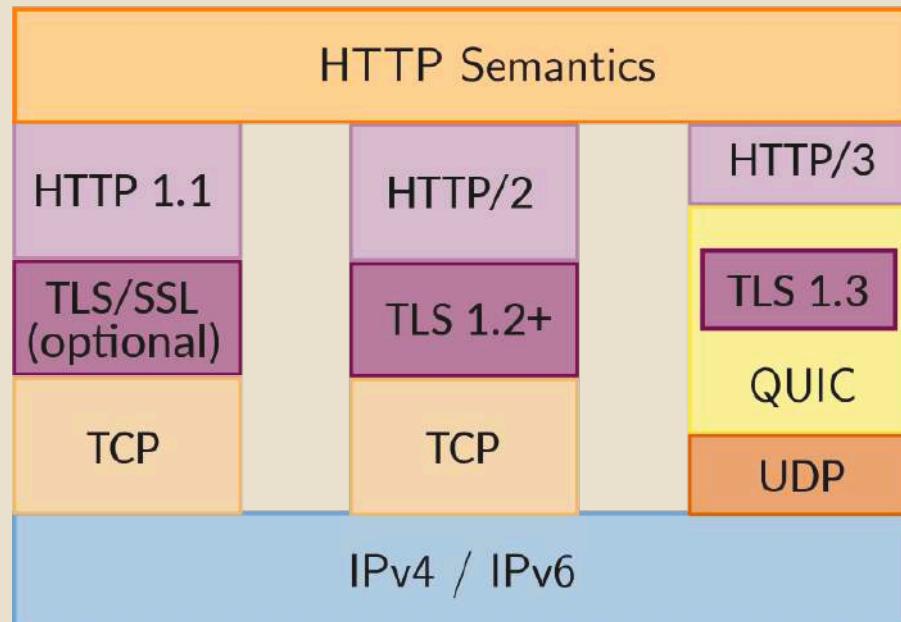
[Building from sources](#)  
[Configuration](#)  
[Example Configuration](#)  
[Troubleshooting](#)

Support for [QUIC](#) and [HTTP/3](#) protocols is available since 1.25.0. Also, since 1.25.0, the QUIC and HTTP/3 support is available in Linux [binary packages](#).

The QUIC and HTTP/3 support is [experimental](#) caveat emptor applies.



# protocol over quic



\* mqtt over quic

\* http over quic

\* grpc over quic

\* rpcx over quic

\* ...



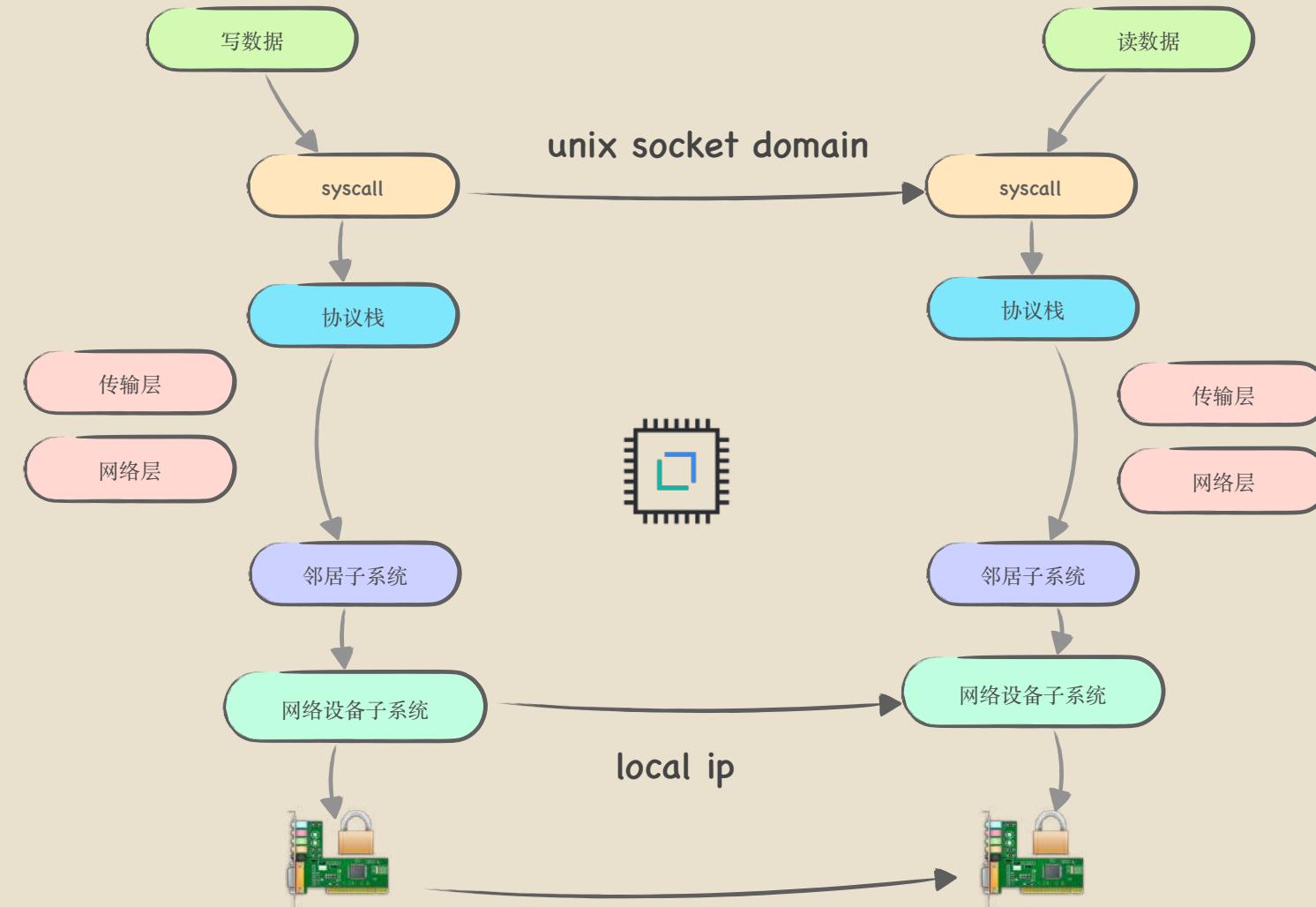
# 其他问题



- \* 本地 IO vs 本地 IP vs UDS (Unix Domain Socket)
- \* client/server 的连接数的限制 ?
- \* client/server 在 crash 后出现问题 ?
- \* socket buffer 大小问题



# local ip vs uds kernel path



如何抓包？

- \* uds 不走协议栈
- \* local ip 不走网卡
- \* uds 性能要比 localhost 高的多



# 连接端口限制？

\* 常见的限制

\* 文件描述符限制 (`ulimit`)

\* 内存限制 (`buffer/cache`)

\* `other sysctl.conf`

\* 服务端

\* 排除限制条件下，连接数要看tcp 4 元祖

\* client的ip数量最大为2的32次方, port为2的16次方

\* 所以单机理论是大约可到2的48次方

\* 客户端

\* 排除限制条件下，连接数要看tcp 4 元祖

\* `ip_local_port_range (default 32768 - 60999)`

```
$ ss -ant|grep -v :22|grep 34006
ESTAB      0      0      192.168.124.10:34006      111.206.176.91:80
ESTAB      0      0      192.168.124.10:34006      220.194.111.148:80
ESTAB      0      0      192.168.124.10:34006      123.126.104.68:80
ESTAB      0      0      192.168.124.10:34006      220.194.111.149:80
ESTAB      0      0      192.168.124.10:34006      115.159.231.124:80
ESTAB      0      0      192.168.124.10:34006      61.182.138.3:80
ESTAB    263120  0      192.168.124.10:34006      123.125.114.5:80
```



客户端在四元组不冲突下，没有端口限制！！！

# peer to peer



ip\_local\_port\_range = 3w - 6w

客户端在四元组不冲突下，没有端口限制！！！

压测场景

3w



src-ip	src-port	dst-ip	dst-port
192.168.0.99	30001	192.168.0.100	80
192.168.0.99	30002	192.168.0.100	80
192.168.0.99	...	192.168.0.100	80
192.168.0.99	60000	192.168.0.100	80

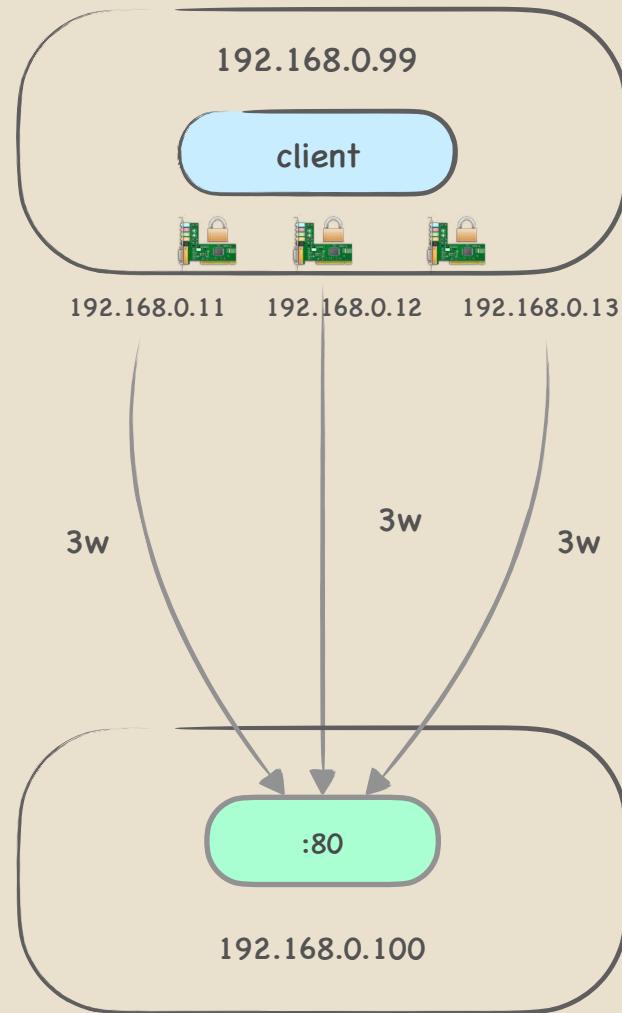
四元组中的三个字段是固定的, 所以只能开 3 w

# bind local interface

`ip_local_port_range = 3w - 6w`

客户端在四元组不冲突下，没有端口限制！！！

压测场景



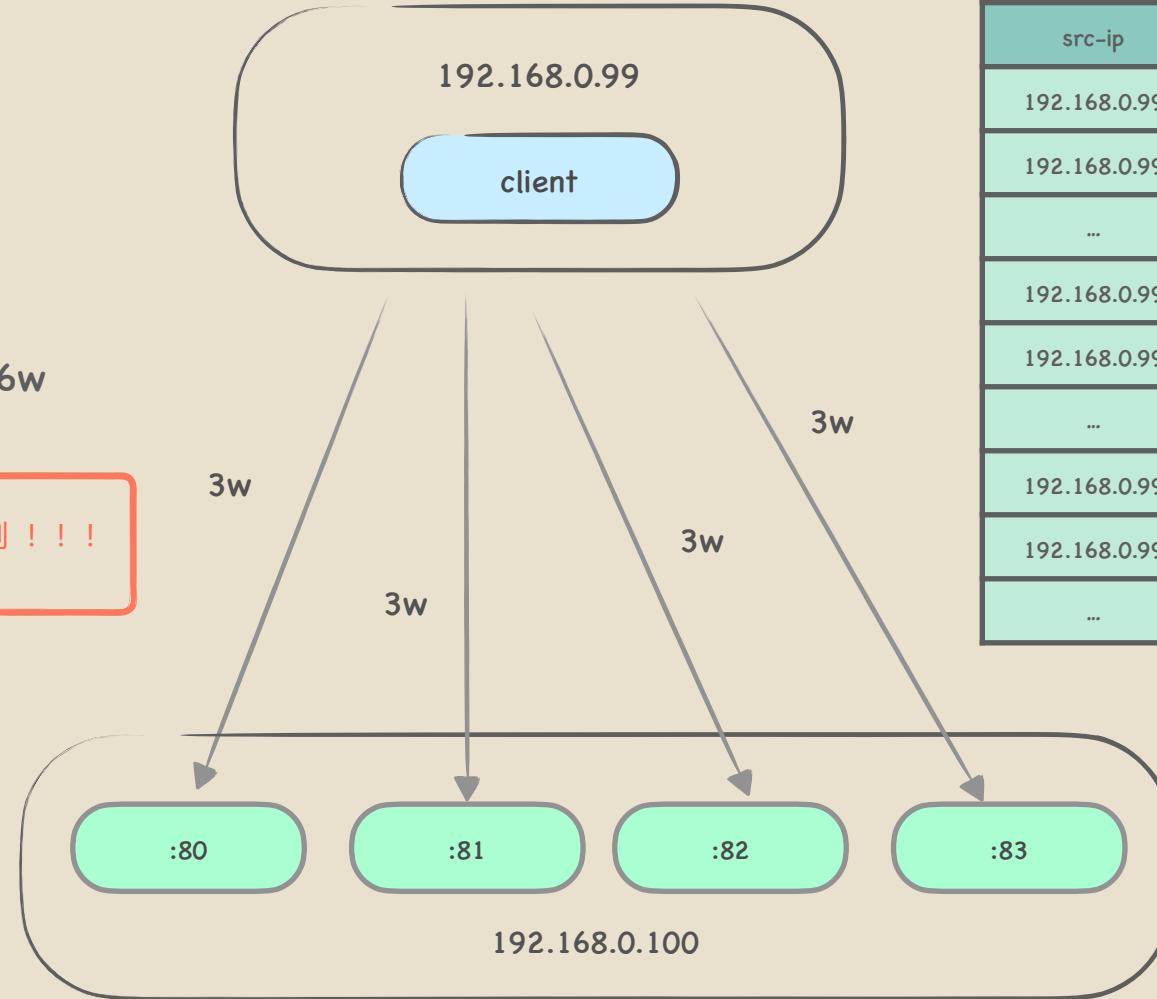
src-ip	src-port	dst-ip	dst-port
192.168.0.11	30001	192.168.0.100	80
192.168.0.11	30002	192.168.0.100	80
...	...	192.168.0.100	80
192.168.0.12	30001	192.168.0.100	80
192.168.0.12	60000	192.168.0.100	80
...	...	192.168.0.100	80
192.168.0.13	30001	192.168.0.100	80
192.168.0.13	60000	192.168.0.100	80



# dial more server ports

ip\_local\_port\_range = 3w - 6w

客户端在四元组不冲突下，没有端口限制！！！

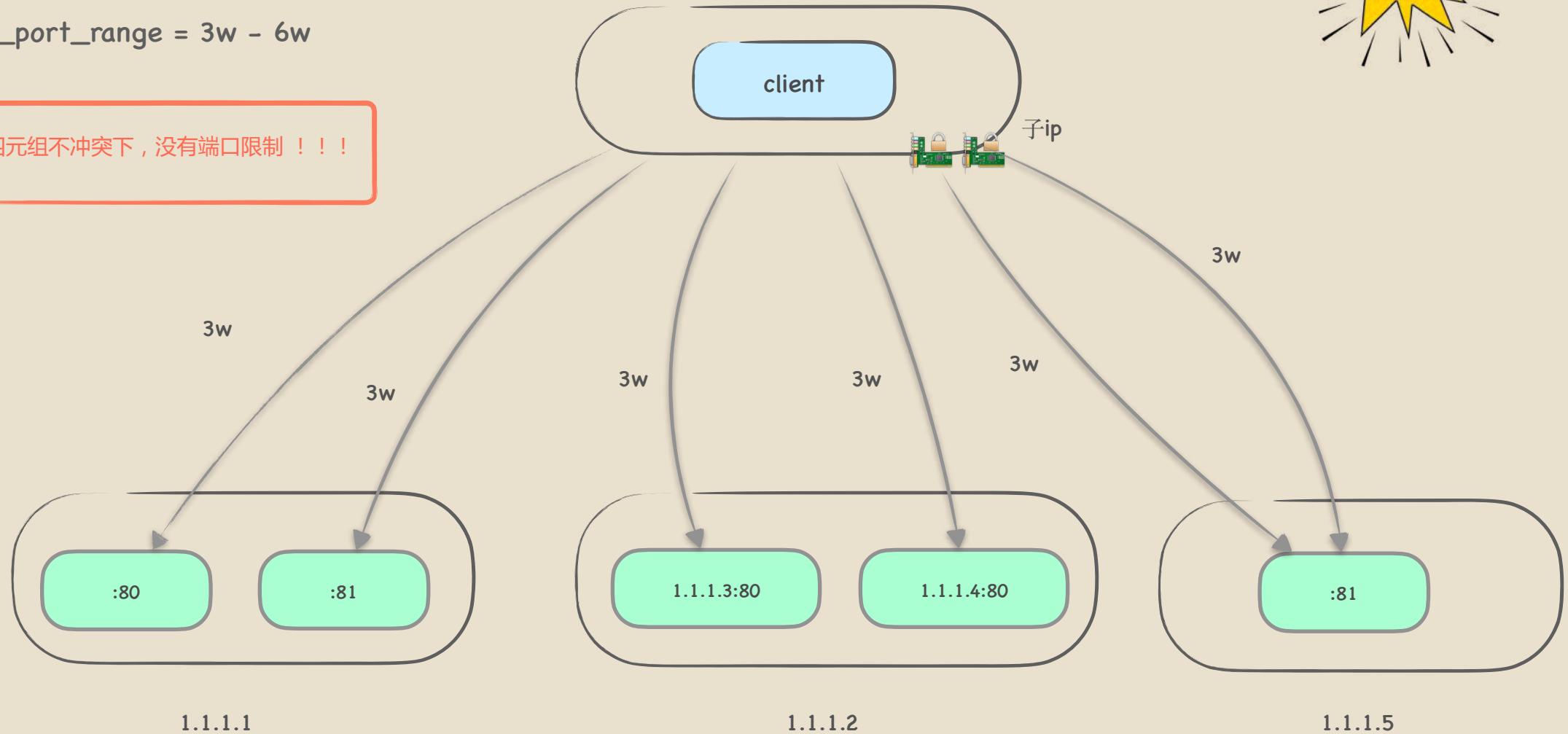


src-ip	src-port	dst-ip	dst-port
192.168.0.99	30001	192.168.0.100	80
192.168.0.99	30002	192.168.0.100	80
...	...	192.168.0.100	...
192.168.0.99	30001	192.168.0.100	81
192.168.0.99	60000	192.168.0.100	81
...	...	192.168.0.100	81
192.168.0.99	30001	192.168.0.100	82
192.168.0.99	60000	192.168.0.100	82
...	...	192.168.0.100	...

# dial more servers

ip\_local\_port\_range = 3w - 6w

客户端在四元组不冲突下，没有端口限制！！！



# when process crash ?

\* 当进程退出后，由操作系统内核来收尾 ...

\* 无未读缓冲

\* 内核接管并主动发起 fin

\* 有未读缓冲

\* 触发 rst

\* linger

\* 触发 rst

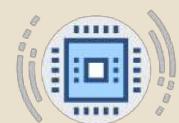
\* FIN

\* init 0 (关机)

\* kill -15 pid

\* kill -9 pid

\* OOM without recv-q



内核帮你搞定 !!!



# when server crash ?

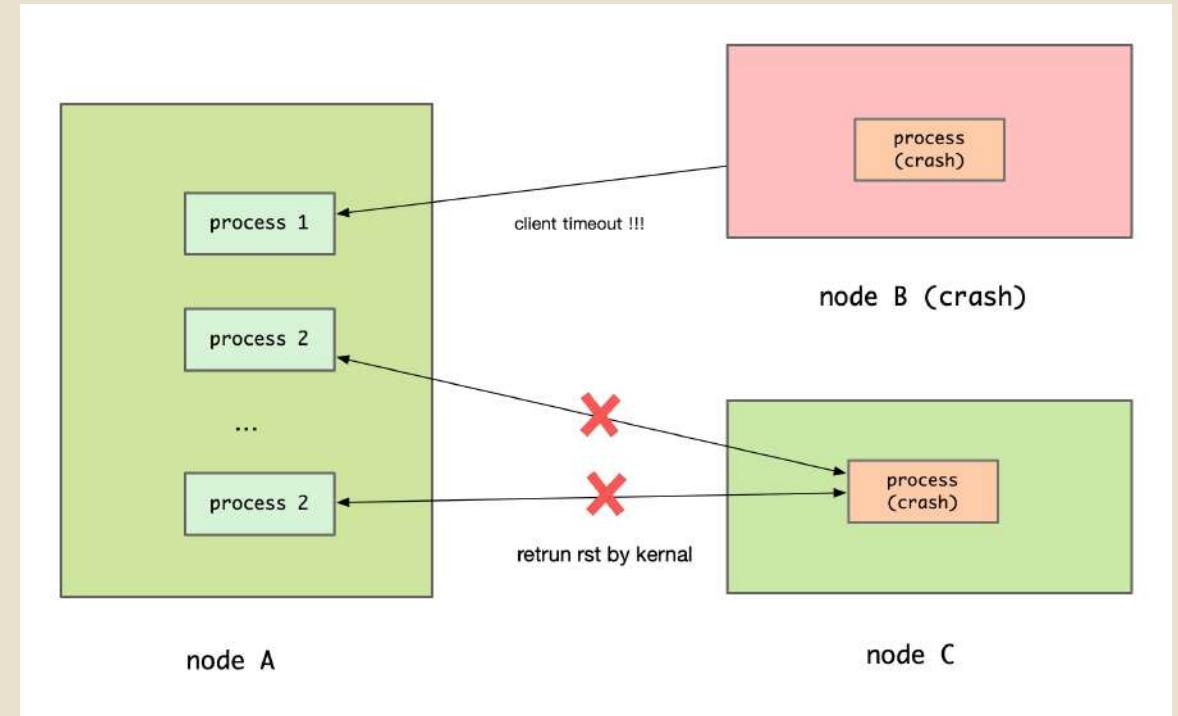
```
$ ping -c 1 8.8.1.1
PING 8.8.1.1 (8.8.1.1): 56 data bytes

--- 8.8.1.1 ping statistics ---
1 packets transmitted, 0 packets received, 100.0% packet loss

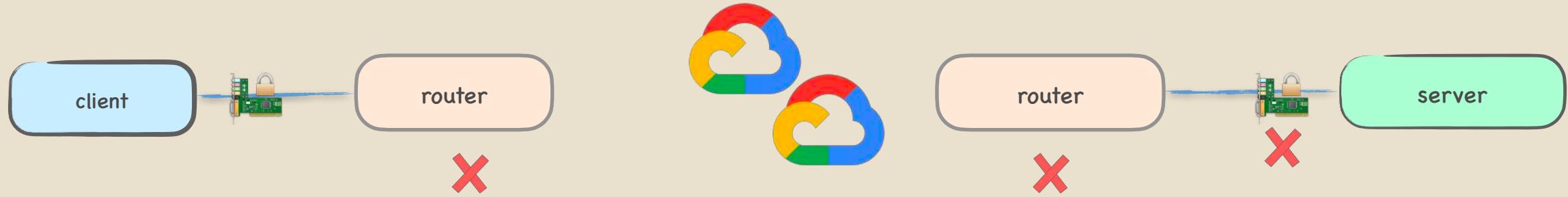
$ time curl 8.8.1.1
curl: (28) Failed to connect to 8.8.1.1 port 80: Operation timed out
curl 8.8.1.1  0.00s user 0.01s system 0% cpu 1:15.09 total
```

- \* 为什么 curl/telnet 建连时 ...
- \* 有时立马就返回失败？
- \* 服务端系统内核反馈 rst

- \* 有时会阻塞很久才失败？
- \* 服务端关机, 没人回应



# when router crash ?



两端如何感知连接已关闭 ?

tcp的各状态超时或逻辑心跳 !!!

# buffer

- \* 尽量不要对 socket 指定 SO\_SNDBUF, SO\_RCVBUF 缓冲区
- \* linux 下默认对读写缓冲区大小自动调节
- \* 自动根据 BDP 带宽时延乘积  $BDP=rtt*(带宽/8)$  优化缓冲区

```
$ > sudo sysctl -a | egrep "rmem|wmem|moderate"

net.ipv4.tcp_moderate_rcvbuf = 1          // 开启接收缓冲区自动调节
net.ipv4.tcp_rmem = 4096    87380    6291456 // 最小值 默认值 最大值
net.ipv4.tcp_wmem = 4096    16384    4194304 // 最小值 默认值 最大值
```



# grace upgrade

- \* 创建进程

- \* fork

- \* exec

- \* system = fork + exec + waitpid



- \* 只要不指定 close-on-exec flag, 那么子进程就继承文件描述符表 !!!



```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(){
    int serv_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    serv_addr.sin_port = htons(1234);
    bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    listen(serv_sock, 20);

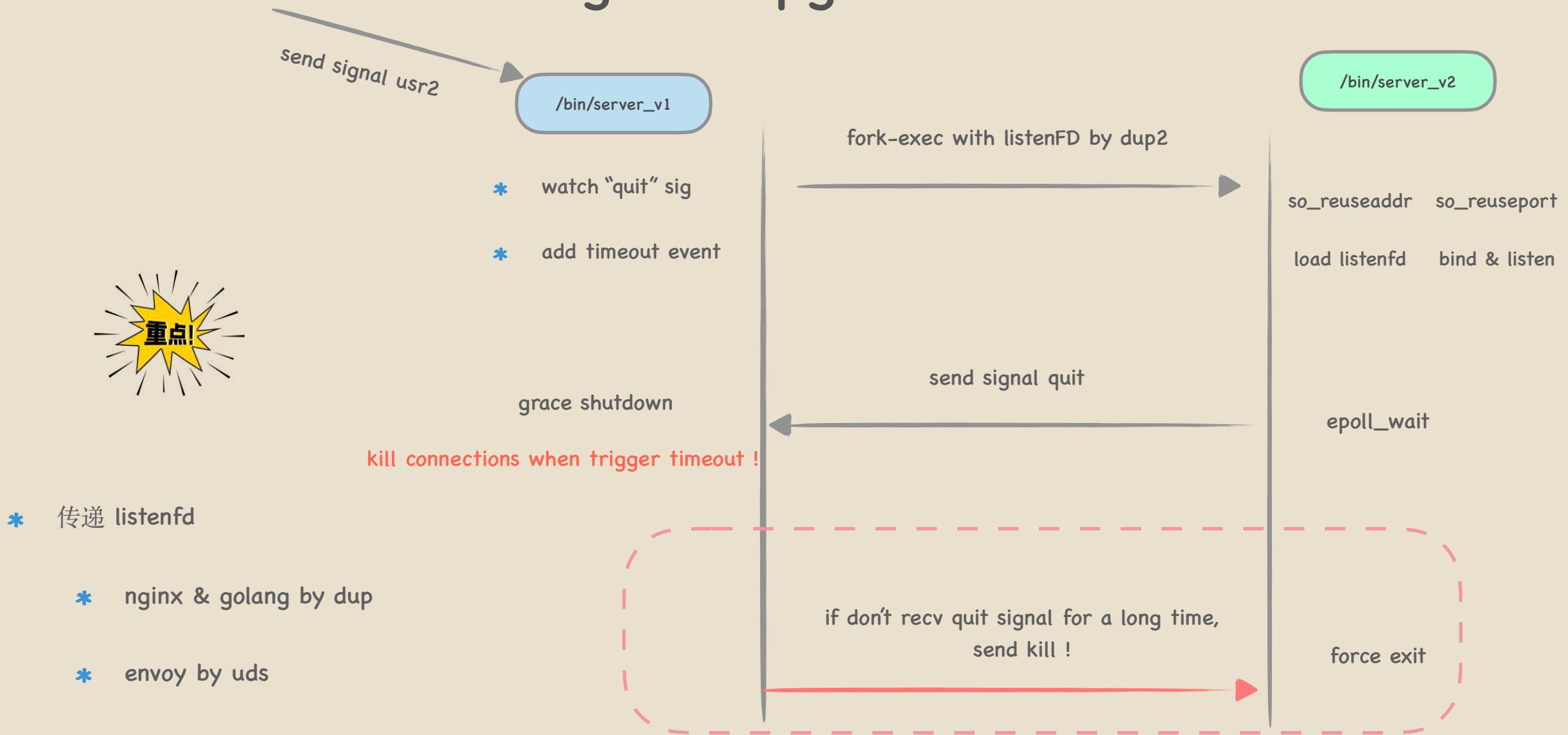
    // method 1
    // system("sleep 1111");

    // method 2
    char* arr[] = {"sleep", "1111", NULL};
    execv("/usr/bin/sleep", arr);
    return 0;
}
```

```
[root@host-46 ~ ]$ ps aux|grep -v grep| grep "sleep 1111"
root      393065  0.0  0.0 108052   360 pts/6    S    11:02   0:00 sleep 1111

[root@host-46 ~ ]$ lsof -p 393065 -Pn
COMMAND   PID USER   FD   TYPE   DEVICE SIZE/OFF NODE NAME
sleep  393065 root cwd DIR    8,2     4096 2564288 /root/test
sleep  393065 root rtd DIR    8,2     4096       64 /
sleep  393065 root txt REG    8,2     33128 3221227973 /usr/bin/sleep
...
sleep  393065 root  3u  IPv4 409765637          0t0      TCP *:1234 (LISTEN)
```

# grace upgrade



```
func (g *grace) run() (err error) {
    if _, ok := syscall.Getenv(strings.Split(graceEnv, "=")[0]); ok {
        f := os.NewFile(3, "")
        if g.listener, err = net.FileListener(f); err != nil {
            return
        }
    } else {
        if g.listener, err = net.Listen("tcp", g.srv.Addr); err != nil {
            return
        }
    }

    terminate := make(chan error)
    go func() {
        if err := g.srv.Serve(g.listener); err != nil {
            terminate <- err
        }
    }()
}

quit := make(chan os.Signal)
signal.Notify(quit)

for {
    select {
    case s := <-quit:
        switch s {
        case syscall.SIGINT, syscall.SIGTERM:
            signal.Stop(quit)
            return g.stop().err

        case syscall.SIGUSR2:
            return g.reload().stop().err
        }

    case err = <-terminate:
        return
    }
}
}

func ListenAndServe(addr string, handler http.Handler) error {
    return WithTimeout(defaultTimeout).ListenAndServe(addr, handler)
}
```

```
const (
    graceEnv      = "go_http_reload=true" // env flag for reload
)

func (g *grace) reload() *grace {
    f, err := g.listener.(*net.TCPListener).File()
    if err != nil {
        g.err = err
        return g
    }
    defer f.Close()

    var args []string
    if len(os.Args) > 1 {
        args = append(args, os.Args[1:]...)
    }
    cmd := exec.Command(os.Args[0], args...)
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
    cmd.Env = append(os.Environ(), graceEnv)
    cmd.ExtraFiles = []*os.File{f}

    g.err = cmd.Start()
    return g
}

func (g *grace) stop() *grace {
    if g.err != nil {
        return g
    }

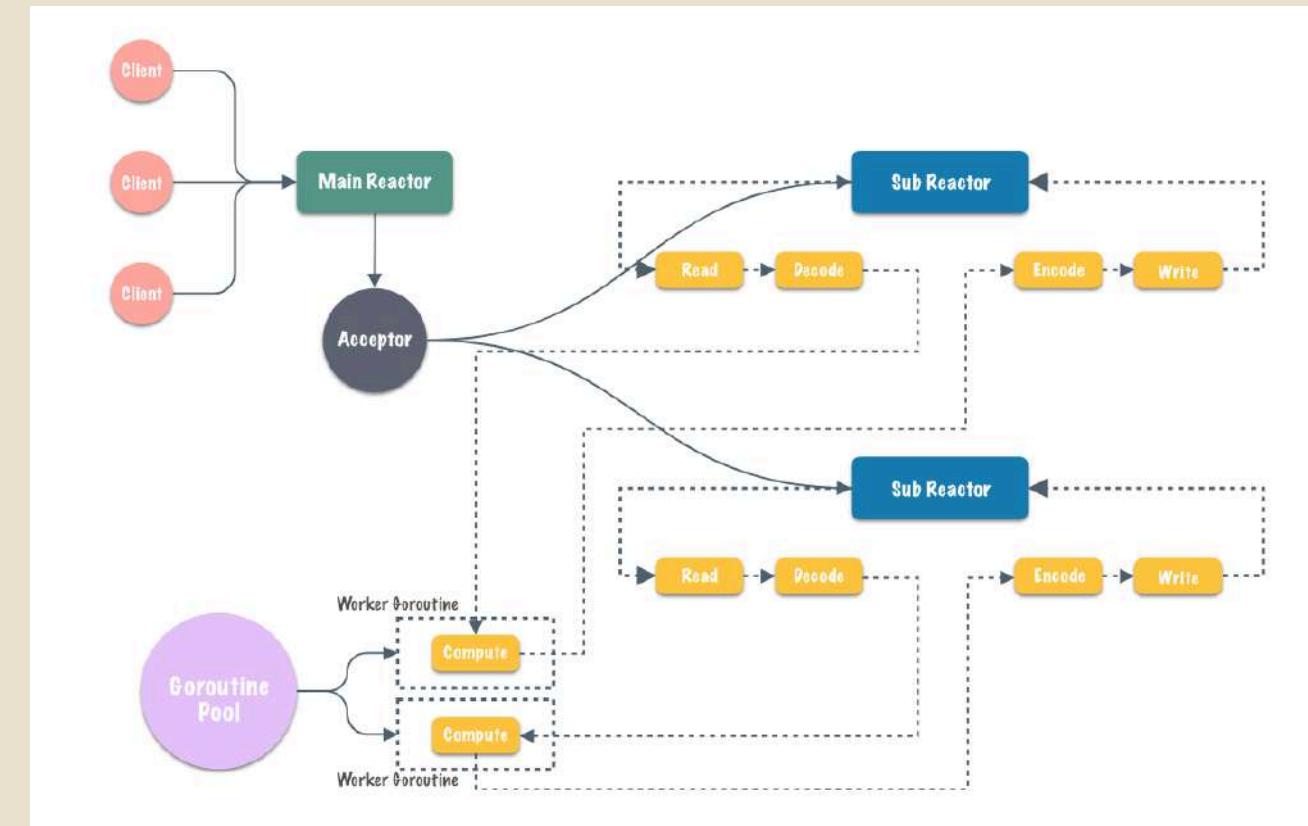
    ctx, cancel := context.WithTimeout(context.Background(), g.timeout)
    defer cancel()

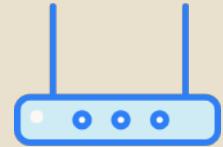
    if err := g.srv.Shutdown(ctx); err != nil {
        g.err = err
    }

    return g
}
```

# golang 网络编程库

- \* <https://github.com/openimsdk/open-im-server>
- \* <https://github.com/tidwall/evio>
- \* <https://github.com/panjf2000/gnet>
- \* <https://github.com/lesismal/nbio>
- \* <https://github.com/Allenxuxu/gev>
- \* <https://github.com/shaovie/goev>





# Q & A



- [xiaorui.cc](http://xiaorui.cc)
- [github.com/rfyiamcool](https://github.com/rfyiamcool)

