Netpoll 模型的抽象和问题

By Xargin

网络编程基础知识

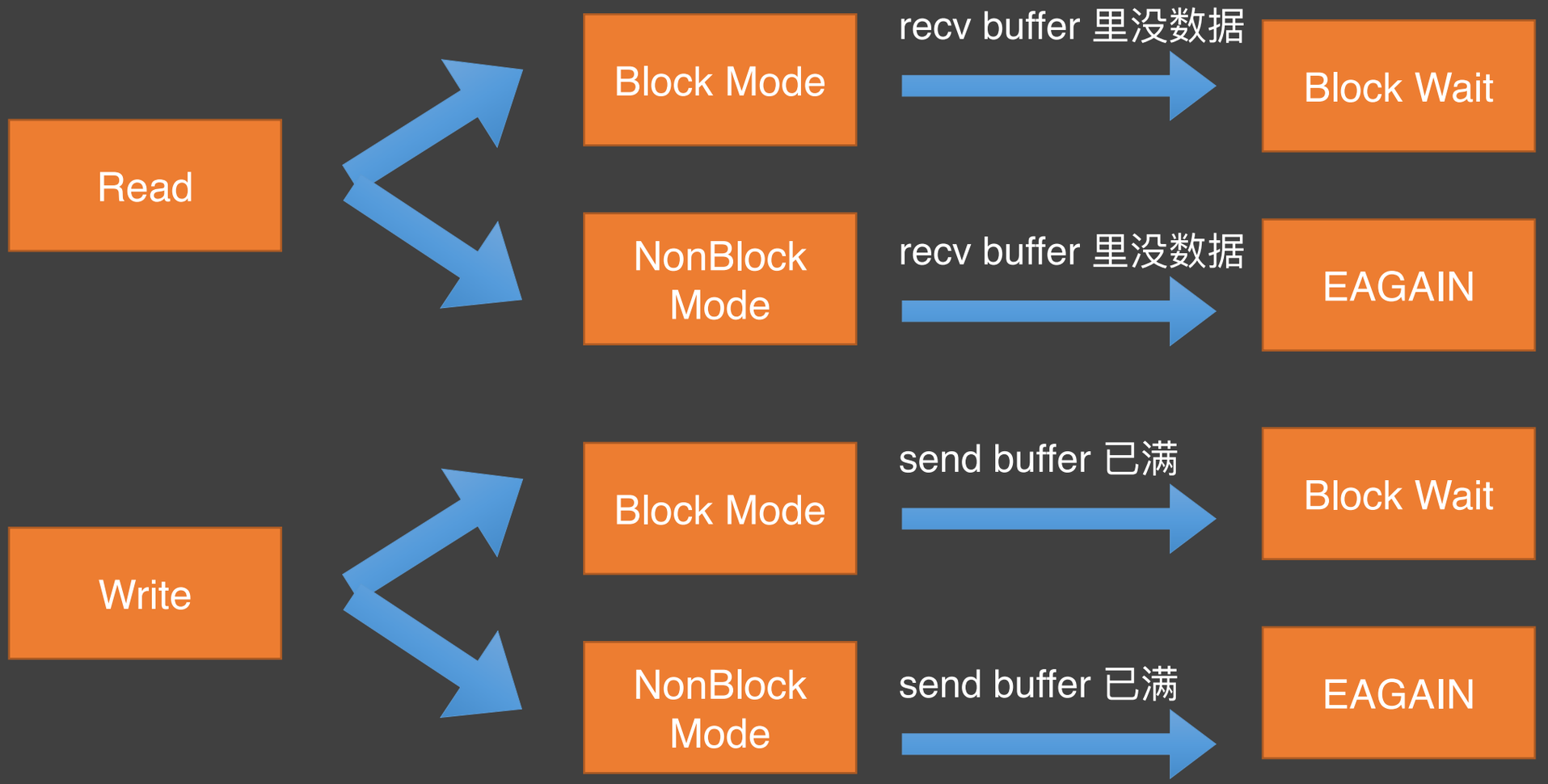
Send Buffer & & Recv Buffer

```
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp4       0      0  192.168.1.5.50422      192.168.1.2.52632      ESTABLISHED
tcp4       0      0  192.168.1.5.50419      47.93.95.217.443       ESTABLISHED
tcp4       0      0  192.168.1.5.50418      47.93.95.217.443       ESTABLISHED
tcp6       0      0  2408:8207:2627:f.50416 2001:470:1:791::.443   ESTABLISHED
tcp6       0      0  2408:8207:2627:f.50415 2001:470:1:791::.443   ESTABLISHED
tcp4       0      0  192.168.1.5.50410      45.77.6.125.12345      ESTABLISHED
tcp4       0      0  127.0.0.1.1080         127.0.0.1.50409        ESTABLISHED
tcp4       0      0  127.0.0.1.50409        127.0.0.1.1080         ESTABLISHED
tcp4       0      0  192.168.1.5.50401      184.86.93.121.443      ESTABLISHED
tcp4       0      0  192.168.1.5.50399      107.21.170.218.443     ESTABLISHED
tcp4       0      0  192.168.1.5.50393      111.230.120.127.443    ESTABLISHED
tcp4       0      0  192.168.1.5.50373      44.231.203.246.443     ESTABLISHED
tcp4       0      0  192.168.1.5.50368      45.77.6.125.12345      ESTABLISHED
tcp4       0      0  127.0.0.1.1080         127.0.0.1.50367        ESTABLISHED
tcp4       0      0  127.0.0.1.50367        127.0.0.1.1080         ESTABLISHED
tcp4       0      0  192.168.1.5.50364      45.77.6.125.12345      ESTABLISHED
tcp4       0      0  127.0.0.1.1080         127.0.0.1.50363        ESTABLISHED
tcp4       0      0  127.0.0.1.50363        127.0.0.1.1080         ESTABLISHED
tcp4       0      0  192.168.1.5.50362      45.77.6.125.12345      ESTABLISHED
```

使用 netstat 可以查看两个 buffer 的情况

网络编程基础知识

阻塞与非阻塞



EAGAIN 的处理方法 https://blog.csdn.net/timid_brother/article/details/52200737

```
sock_fd = socket(PF_INET, SOCK_STREAM, 0);
.....

if(bind(sock_fd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) == -1) {
    error_handling("bind error");
}

if(listen(sock_fd, 5) == -1) {
    error_handling("listen error");
}

epfd = epoll_create(EPOLL_SIZE);
.....

while(1) {
    event_cnt = epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
    .....

    for(i=0; i<event_cnt; i++) {
        if(ep_events[i].data.fd == sock_fd) {
            .....
            conn_fd = accept(sock_fd, (struct sockaddr*)&client_addr, &addr_size);
            event.events = EPOLLIN;
            event.data.fd = conn_fd;
            epoll_ctl(epfd, EPOLL_CTL_ADD, conn_fd, &event);
            printf("connected client : %d\n", conn_fd);
        } else {
            str_len = read(ep_events[i].data.fd, buf, BUF_SIZE);
            if(str_len == 0) {
                epoll_ctl(epfd, EPOLL_CTL_DEL, ep_events[i].data.fd, NULL);
                close(ep_events[i].data.fd);
                printf("closed client: %d\n", ep_events[i].data.fd);
            } else {
                write(ep_events[i].data.fd, buf, str_len);
            }
        }
    }
}
```

- socket
- bind
- listen
- accept
- epoll_create
- epoll_wait
- epoll_ctl
- read
- write

本示例只是 demo，很多错误未处理

```c
sock_fd = socket(PF_INET, SOCK_STREAM, 0);
.....

if(bind(sock_fd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) == -1) {
    error_handling("bind error");
}

if(listen(sock_fd, 5) == -1) {
    error_handling("listen error");
}

epfd = epoll_create(EPOLL_SIZE);
.....

while(1) {
    event_cnt = epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
    .....

    for(i=0; i<event_cnt; i++) {
        if(ep_events[i].data.fd == sock_fd) {
            .....
            conn_fd = accept(sock_fd, (struct sockaddr*)&client_addr, &addr_size);
            event.events = EPOLLIN;
            event.data.fd = conn_fd;
            epoll_ctl(epfd, EPOLL_CTL_ADD, conn_fd, &event);
            printf("connected client : %d\n", conn_fd);
        } else {
            str_len = read(ep_events[i].data.fd, buf, BUF_SIZE);
            if(str_len == 0) {
                epoll_ctl(epfd, EPOLL_CTL_DEL, ep_events[i].data.fd, NULL);
                close(ep_events[i].data.fd);
                printf("closed client: %d\n", ep_events[i].data.fd);
            } else {
                write(ep_events[i].data.fd, buf, str_len);
            }
        }
    }
}
```

真实场景的 epoll 程序，read 和 write 流程大多是在回调函数里的

回调函数会将业务逻辑打散，难以阅读

可以挑一个流行的 C 项目看看，代码不是很容易理解

Go 语言中的网络程序

从 accept 开始，都是线性逻辑，流程是顺序的，符合人类思维习惯

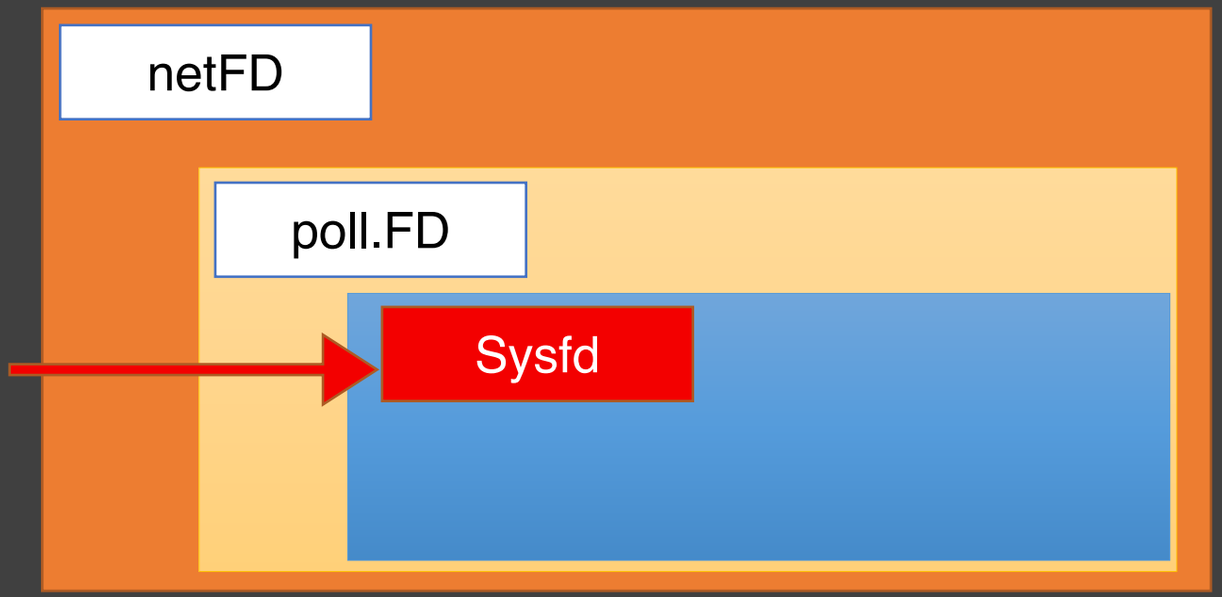
```
 1  package main
 2
 3  import "net"
 4
 5  func main() {
 6          var l net.Listener
 7          for {
 8                  c, _ := l.Accept()
 9                  go func() {
10                          // do your business logic
11                          var buf = make([]byte, 1024)
12                          c.Read(buf)
13                  }()
14          }
15  }
```

Go 语言是怎么做到的呢？主要就是把 Read 和 Write 的非阻塞流程封装成了阻塞流程。因为人们容量理解阻塞，但不容易理解回调

Go 底层一堆 FD 之间的关系

在 Go 语言中 FD 被包了好几层

这是 syscall 给我们返回的 fd



Go 底层一堆 FD 之间的关系

网络 FD 描述符

```
// 网络 FD 描述符，unix 和 windows 是两套实现
type netFD struct {
    pfd poll.FD

    // immutable until Close
    family      int
    sotype      int
    isConnected bool
    net         string
    laddr       Addr
    raddr       Addr
}
```

net/fd_windows.go
net/fd_unix.go

本地文件描述符

```
type File struct {
    *file // os specific
}

type file struct {
    pfd         poll.FD
    name        string
    dirinfo     *dirInfo // nil unless directory being read
    nonblock    bool     // whether we set nonblocking mode
    stdoutOrErr bool     // whether this is stdout or stderr
    appendMode  bool     // whether file is opened for appending
}
```

os/file_windows.go
os/file_unix.go

unix 和 windows 有各自的定义和实现

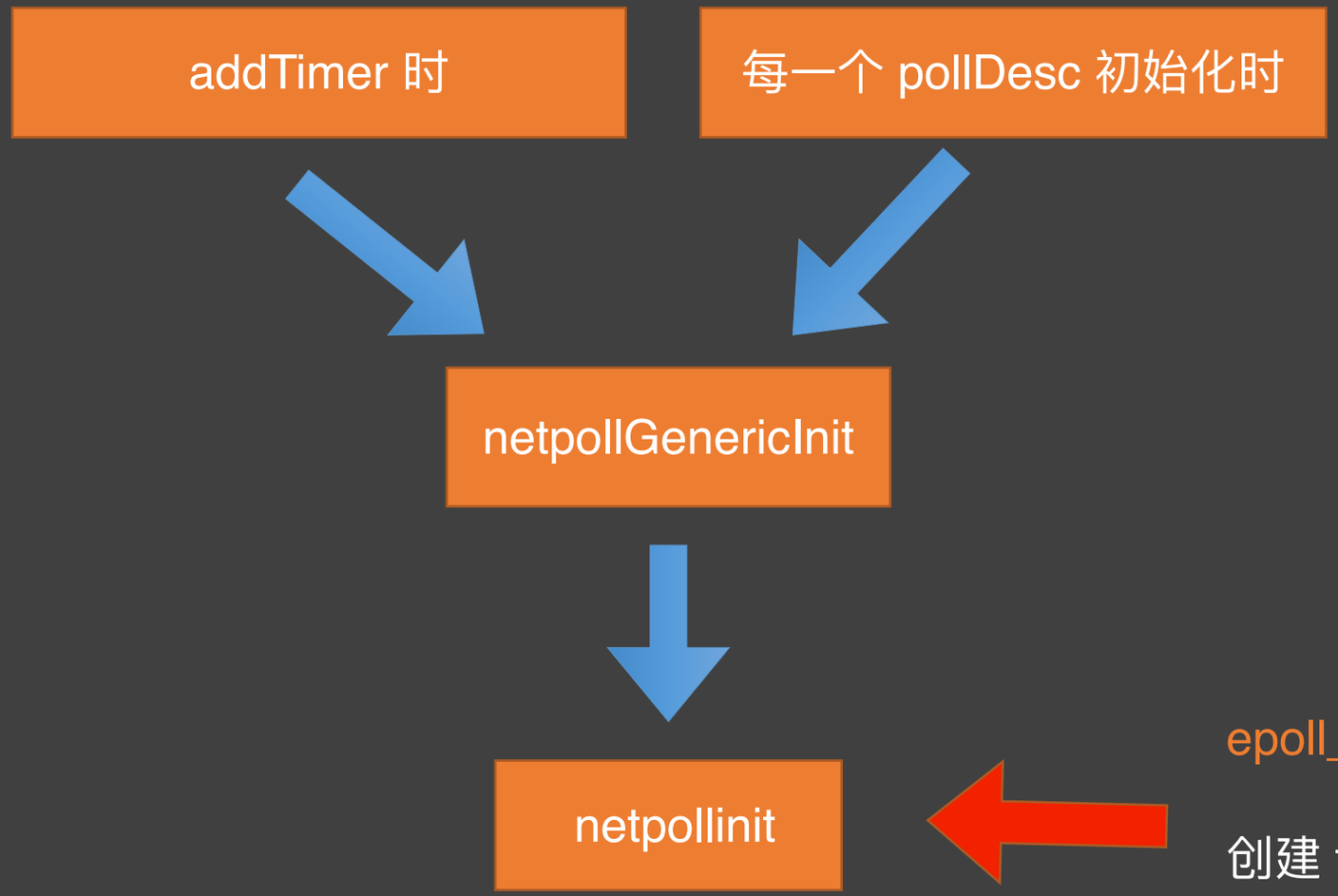
Go 底层一堆 FD 之间的关系

```go
// windows 和 linux 对这个 poll.FD 的定义是有些区别的
type FD struct {
    // 这个锁是为了保证对同一个文件的读、写操作能分别被序列化
    fdmu fdMutex
    // 这个是操作系统给我们返回的 fd 值
    Sysfd int
    // I/O poller. 这是 Go 对 poll 过程的一个抽象，所有平台的抽象都是一样的
    pd pollDesc
    // Writev cache.
    iovecs *[]syscall.Iovec
    // 文件被关闭时会触发该 sema
    csema uint32
    // 如果非 0，说明 FD 被设置为了 blocking 模式
    isBlocking uint32
    // 区分这个是不是一个流式的 FD，与流式相反的是基于 packet 的，即 UDP socket。该值不可变
    IsStream bool
    // 当连接读到 0 长度时，用来区分是否代表 EOF。如果是基于消息的 socket 连接始终是 false
    ZeroReadIsEOF bool
    // 区分这个 FD 代表的是文件，还是网络连接
    isFile bool
}
```

internal/poll/fd_windows.go

internal/poll/fd_unix.go

Go 网络编程基本流程-Epoll 初始化

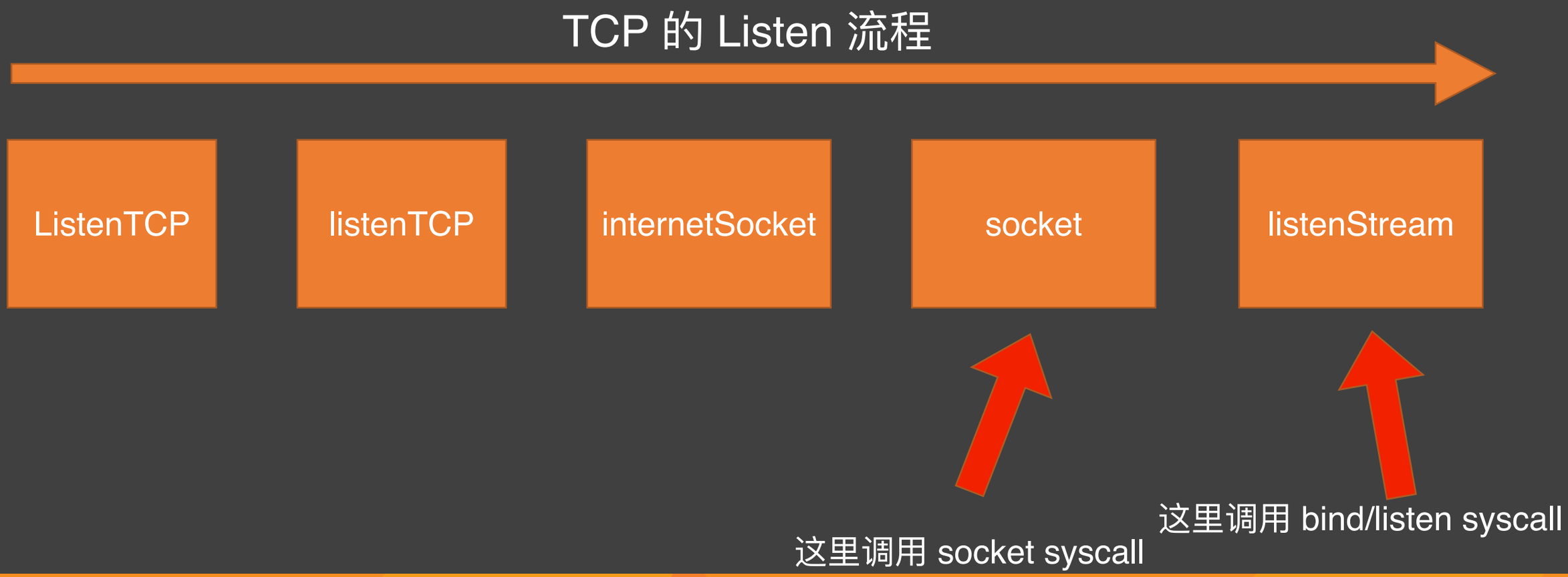


epoll_create

创建 timer 的 non blocking pipe

通过 epoll_ctl 订阅 pipe 的 read 端 fd 的可读事件

Go 网络编程基本流程-Listen



Go 网络编程基本流程-Accept

TCP 的 Accept 流程

TCPListener accept | netFD accept | poll.FD Accept | pollDesc waitRead | pollDesc wait

Accept 成功后，会：

- 将 fd 设置为 nonblocking
- 在 netpollopen 中通过 epoll_ctl 注册 fd 的读、写事件的订阅

```
1 @ 0x10383b0 0x1031a6a 0x1030fd5 0x10c5025 0x10c7884 0x10c7866 0x11f3192 0x120cc62 0x120baa4 0x131122d 0x1310f77 0x1350737 0x1350700 0x1037fea 0x1067a51
#       0x1030fd4       internal/poll.runtime_pollWait+0x54             /usr/local/go/src/runtime/netpoll.go:203
#       0x10c5024       internal/poll.(*pollDesc).wait+0x44             /usr/local/go/src/internal/poll/fd_poll_runtime.go:87
#       0x10c7883       internal/poll.(*pollDesc).waitRead+0x1d3        /usr/local/go/src/internal/poll/fd_poll_runtime.go:92
#       0x10c7865       internal/poll.(*FD).Accept+0x1b5                /usr/local/go/src/internal/poll/fd_unix.go:384
#       0x11f3191       net.(*netFD).accept+0x41                        /usr/local/go/src/net/fd_unix.go:238
#       0x120cc61       net.(*TCPListener).accept+0x31                  /usr/local/go/src/net/tcpsock_posix.go:139
#       0x120baa3       net.(*TCPListener).Accept+0x63                  /usr/local/go/src/net/tcpsock.go:261
#       0x131122c       net/http.(*Server).Serve+0x25c                  /usr/local/go/src/net/http/server.go:2930
#       0x1310f76       net/http.(*Server).ListenAndServe+0xb6          /usr/local/go/src/net/http/server.go:2859
#       0x1350736       net/http.ListenAndServe+0x96                    /usr/local/go/src/net/http/server.go:3115
#       0x13506ff       main.main+0x5f                                  /Users/xargin/test/go/http/httptest.go:28
#       0x1037fe9       runtime.main+0x1f9                              /usr/local/go/src/runtime/proc.go:203
```

Go 网络编程基本流程-Read

TCP 的 Read 流程

net.TCPConn Read

net.conn Read

netFD Read

poll.FD Read

syscall.Read

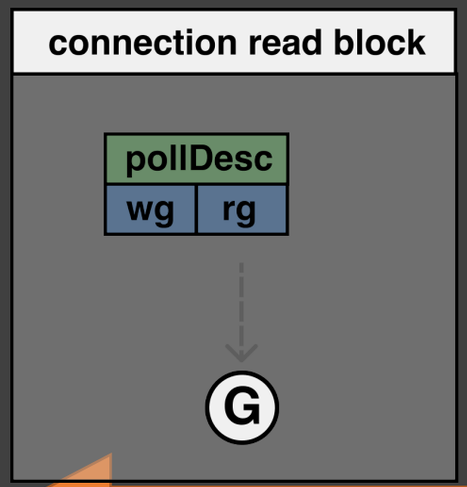
EAGAIN

这个在 pprof 里应该经常见到

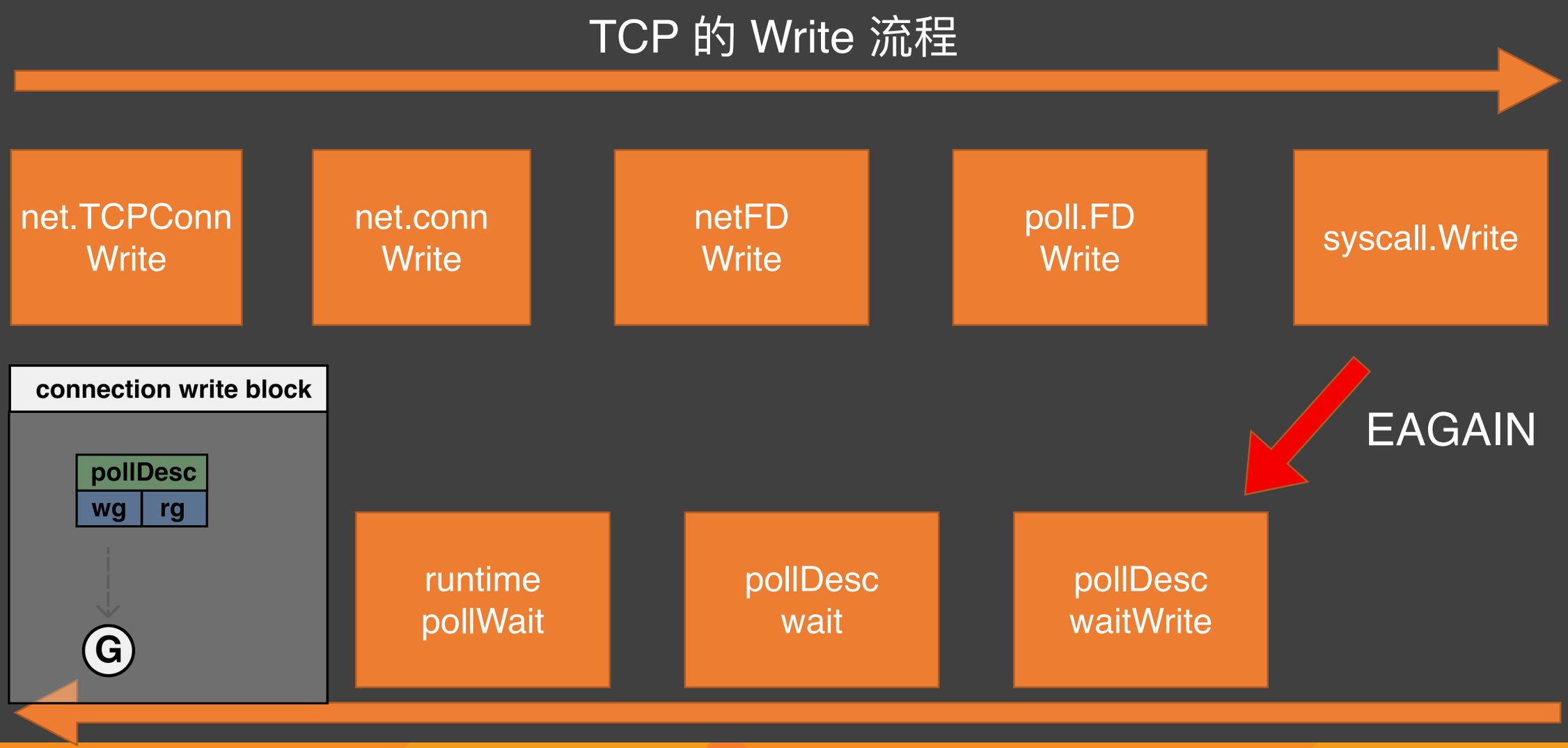
runtime pollWait

pollDesc wait

pollDesc waitRead



Go 网络编程基本流程-Write



runtime.pollWait 实现

```
func poll_runtime_pollWait(pd *pollDesc, mode int)
    errcode := netpollcheckerr(pd, int32(mode))
    if errcode ≠ pollNoError {
        return errcode
    }

    for !netpollblock(pd, int32(mode), false) {
        errcode = netpollcheckerr(pd, int32(mode))
        if errcode ≠ pollNoError {
            return errcode
        }
    }
    return pollNoError
}
```

```
func netpollblock(pd *pollDesc, mode int32, waitio bool) bool {
    gpp := &pd.rg
    if mode == 'w' {
        gpp = &pd.wg
    }

    // set the gpp semaphore to pdWait
    for {
        old := *gpp
        if old == pdReady {
            *gpp = 0
            return true
        }
        if old ≠ 0 {
            throw("runtime: double wait")
        }
        if atomic.Casuintptr(gpp, 0, pdWait) {
            break
        }
    }

    if waitio || netpollcheckerr(pd, mode) == 0 {
        gopark(netpollblockcommit, unsafe.Pointer(gpp),
ReasonIOWait, traceEvGoBlockNet, 5)
    }

    old := atomic.Xchguintptr(gpp, 0)
    if old > pdWait {
        throw("runtime: corrupted polldesc")
    }
    return old == pdReady
}
```

不管用户是暂时 Read 不了还是 Write 不了

你的 goroutine 都会被 gopark 起来

netpoll 执行流程

在调度和 GC 的关键点上都会检查一次 netpoll，看是否有已经 ready 的 fd

proc.go	startTheWorldWithSema	1246	`list := netpoll(0) // non-blocking`
proc.go	findrunnable	2756	`if list := netpoll(0); !list.empty() { // non-blocking`
proc.go	findrunnable	2947	`list := netpoll(delay) // block until new work is available`
proc.go	pollWork	3001	`if list := netpoll(0); !list.empty() {`
proc.go	sysmon	5398	`list := netpoll(0) // non-blocking - returns list of goroutines`
time.go		420	`// This is called by the netpoll code or time.Ticker.Reset or time.Timer.Reset.`

相当于每次调度循环都要走到 netpoll，检查的频率还是比较高的

netpoll 执行流程

epoll 版的 netpoll

event 中的 data 就是 pollDesc

根据 ready 的事件是 Read 或 Write 分别从 pollDesc 的 rg、wg 上拿到该唤醒的 goroutine

```go
func netpoll(delay int64) gList {
    var events [128]epollevent
    n := epollwait(epfd, &events[0], int32(len(events)), waitms)

    var toRun gList
    for i := int32(0); i < n; i++ {
        ev := &events[i]
        if ev.events == 0 {
            continue
        }

        var mode int32
        if ev.events&(_EPOLLIN|_EPOLLRDHUP|_EPOLLHUP|_EPOLLERR) != 0 {
            mode += 'r'
        }
        if ev.events&(_EPOLLOUT|_EPOLLHUP|_EPOLLERR) != 0 {
            mode += 'w'
        }
        if mode != 0 {
            pd := *(**pollDesc)(unsafe.Pointer(&ev.data))
            pd.everr = false
            if ev.events == _EPOLLERR {
                pd.everr = true
            }
            netpollready(&toRun, pd, mode)
        }
    }
    return toRun
}
```

netpoll 执行流程

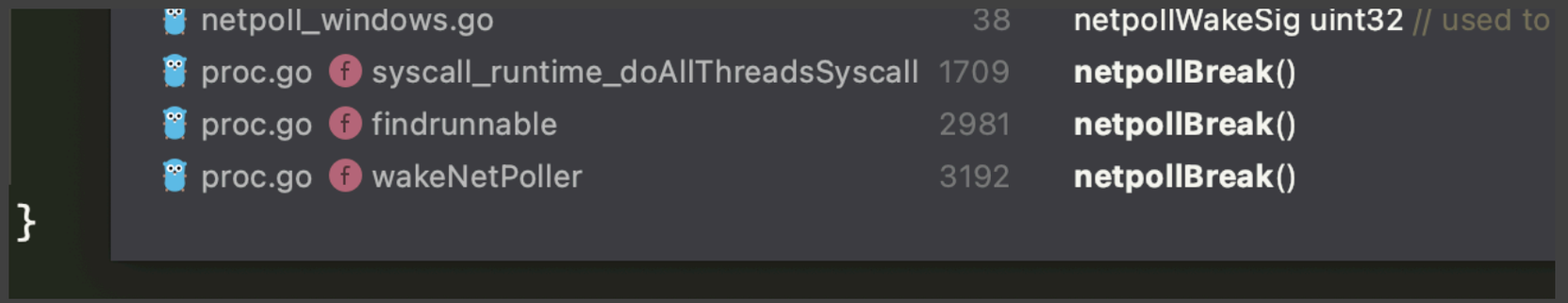
已经 ready 的 goroutine push 到 toRun 链表

```
func netpollready(toRun *gList, pd *pollDesc, mode int32) {
    var rg, wg *g
    if mode == 'r' || mode == 'r'+'w' {
        rg = netpollunblock(pd, 'r', true)
    }
    if mode == 'w' || mode == 'r'+'w' {
        wg = netpollunblock(pd, 'w', true)
    }
    if rg ≠ nil {
        toRun.push(rg)
    }
    if wg ≠ nil {
        toRun.push(wg)
    }
}
```

toRun 列表最终从 netpoll() 中返回，通过 injectglist 进入全局队列

netpoll 与 timer 结合

有 timer 过期，且没有被 checkTimers 检查到，需要处理时：



唤醒可能正在阻塞的 netpoll 线程进行处理

这时候如果有阻塞的 epoll_wait，会立刻返回

网络连接上的读写锁

- 同一条连接上的读需要被序列化
- 同一条连接上的写需要被序列化
- 读和写可以并发
- 同一个 FD 上的读/写操作，底层是一定有锁的

```
144  // Read implements io.Reader.
145  func (fd *FD) Read(p []byte) (int, error) {
146      if err := fd.readLock(); err != nil {
147          return 0, err
148      }
149      defer fd.readUnlock()
150      if len(p) == 0 { 0, nil }
158      if err := fd.pd.prepareRead(fd.isFile); err != nil { 0, err }
161      if fd.IsStream && len(p) > maxRW {
162          p = p[:maxRW]
163      }
164      for {
```

注意，这里的 readLock 不是我们常见的 read lock，同一条连接的所有 read 都是彼此互斥的

网络连接上的读写锁

- 同一条连接上的读需要被序列化
- 同一条连接上的写需要被序列化
- 读和写可以并发
- 同一个 FD 上的读/写操作，底层是一定有锁的

```
254 func (fd *FD) Write(p []byte) (int, error) {
255     if err := fd.writeLock(); err != nil { return 0, err }
258     defer fd.writeUnlock()
259     if err := fd.pd.prepareWrite(fd.isFile); err != nil { return 0, err }
262     var nn int
263     for {
264         max := len(p)
265         if fd.IsStream && max-nn > maxRW {
266             max = nn + maxRW
267         }
268         n, err := syscall.Write(fd.Sysfd, p[nn:max])
269         if n > 0 {
```

同一条连接的所有写也是彼此互斥的

标准库的 netpoll 缺陷分析

当前我们线上的系统，一般活跃连接较少，大多是类似于：

总连接数：5w；活跃连接数：1k 这样的情况，活跃连接在总连接中的比例较低。

不活跃的连接，一般情况下都是阻塞在 conn.Read上，

所有阻塞的 goroutine 都会占用 goroutine 的栈空间，以及 Read buffer 的空间

```
goroutine profile: total 257
174 @ 0x10383b0 0x1031a6a 0x1030fd5 0x10c5025 0x10c5f21 0x10c5f03 0x11f28df 0x12044de 0x1306dc4 0x10e7a13 0x10e872d 0x10e8964 0x12
#       0x1030fd4       internal/poll.runtime_pollWait+0x54             /usr/local/go/src/runtime/netpoll.go:203
#       0x10c5024       internal/poll.(*pollDesc).wait+0x44             /usr/local/go/src/internal/poll/fd_poll_runtime.go:87
#       0x10c5f20       internal/poll.(*pollDesc).waitRead+0x200        /usr/local/go/src/internal/poll/fd_poll_runtime.go:92
#       0x10c5f02       internal/poll.(*FD).Read+0x1e2                  /usr/local/go/src/internal/poll/fd_unix.go:169
#       0x11f28de       net.(*netFD).Read+0x4e                          /usr/local/go/src/net/fd_unix.go:202
#       0x12044dd       net.(*conn).Read+0x8d                           /usr/local/go/src/net/net.go:184
#       0x1306dc3       net/http.(*connReader).Read+0xf3                /usr/local/go/src/net/http/server.go:797
#       0x10e7a12       bufio.(*Reader).fill+0x102                      /usr/local/go/src/bufio/bufio.go:100
#       0x10e872c       bufio.(*Reader).ReadSlice+0x3c                  /usr/local/go/src/bufio/bufio.go:359
#       0x10e8963       bufio.(*Reader).ReadLine+0x33                   /usr/local/go/src/bufio/bufio.go:388
#       0x129320b       net/textproto.(*Reader).readLineSlice+0x6b      /usr/local/go/src/net/textproto/reader.go:58
#       0x13014e3       net/textproto.(*Reader).ReadLine+0xa3           /usr/local/go/src/net/textproto/reader.go:39
#       0x1301512       net/http.readRequest+0xd2                       /usr/local/go/src/net/http/request.go:1015
#       0x13081a0       net/http.(*conn).readRequest+0x190              /usr/local/go/src/net/http/server.go:983
#       0x130c753       net/http.(*conn).serve+0x6d3                    /usr/local/go/src/net/http/server.go:1850
```

可以认为，这种模式最大的缺陷：那些不活跃的连接要占用太多的资源

社区里的 netpoll 到底在优化什么?

将 Go 的 goroutine per connection 抽象抹掉，修改回 syscall.EpollWait 的回调模式，编程方式与 C/C++ 类似

这样非活跃连接在 Go 的应用层是不占用任何资源的，活跃连接由 EpollWait 返回的事件决定，当 EpollWait 返回 events 时，再启动/复用 goroutine 处理

```go
func (ep *Epoll) wait(onError func(error)) {
    defer func() {
        if err := unix.Close(ep.fd); err ≠ nil {
            onError(err)
        }
        close(ep.waitDone)
    }()

    events := make([]unix.EpollEvent, maxWaitEventsBegin)
    callbacks := make([]func(EpollEvent), 0, maxWaitEventsBegin)

    for {
        n, err := unix.EpollWait(ep.fd, events, -1)
        if err ≠ nil {
            if temporaryErr(err) {
                continue
            }
            onError(err)
            return
        }

        callbacks = callbacks[:n]

        ep.mu.RLock()
        for i := 0; i < n; i++ {
            fd := int(events[i].Fd)
            if fd == ep.eventFd { // signal to close
                ep.mu.RUnlock()
                return
            }
            callbacks[i] = ep.callbacks[fd]
        }
        ep.mu.RUnlock()

        for i := 0; i < n; i++ {
            if cb := callbacks[i]; cb ≠ nil {
                cb(EpollEvent(events[i].Events))
                callbacks[i] = nil
            }
        }

        if n == len(events) && n*2 ⩽ maxWaitEventsStop {
            events = make([]unix.EpollEvent, n*2)
            callbacks = make([]func(EpollEvent), 0, n*2)
        }
    }
}
```

Q && A