# 字节跳动在 Go 网络库上的实践

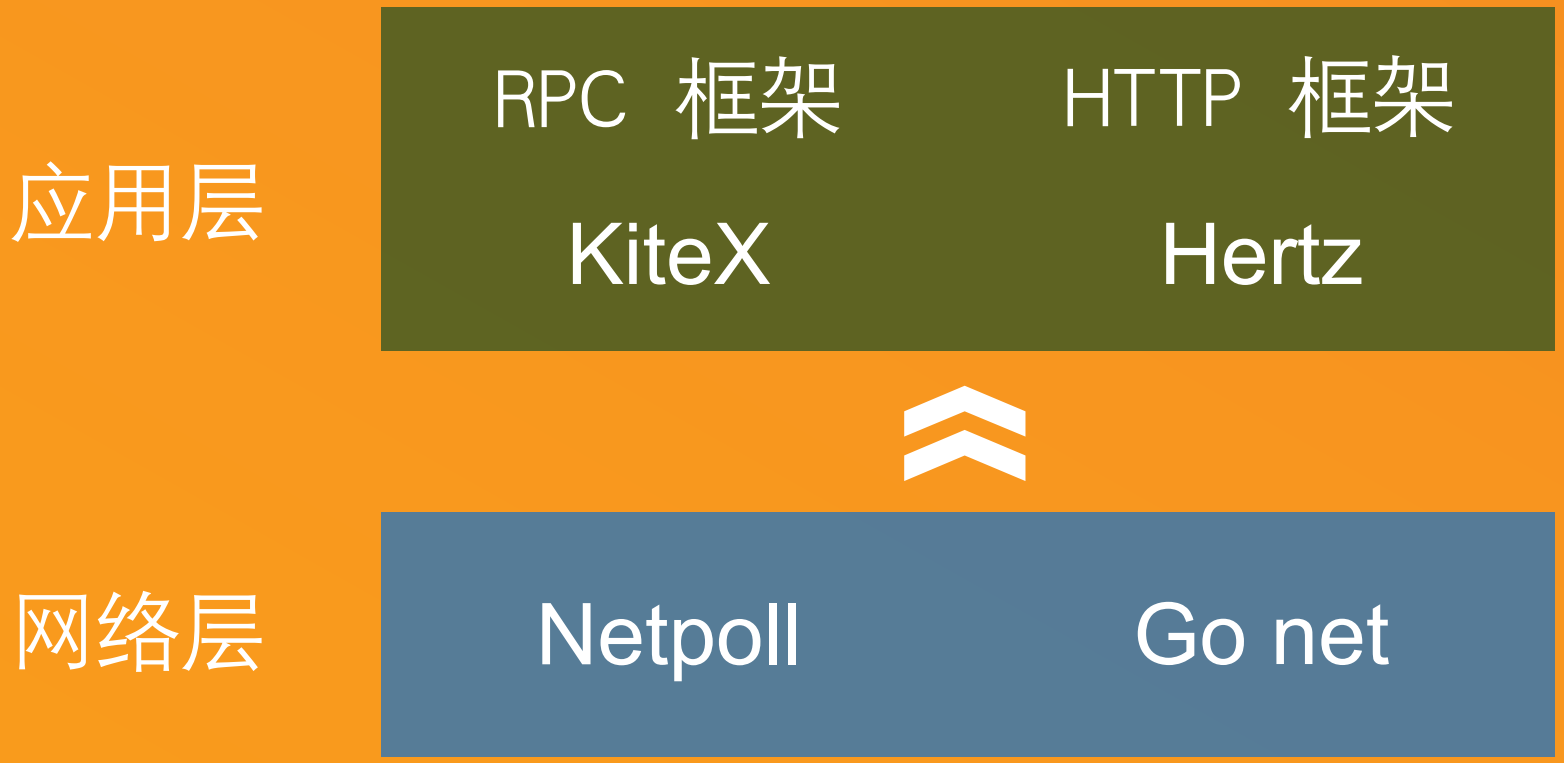



何晨

字节跳动

基础架构 – 研发

# Netpoll – 面向 RPC 场景的网络库



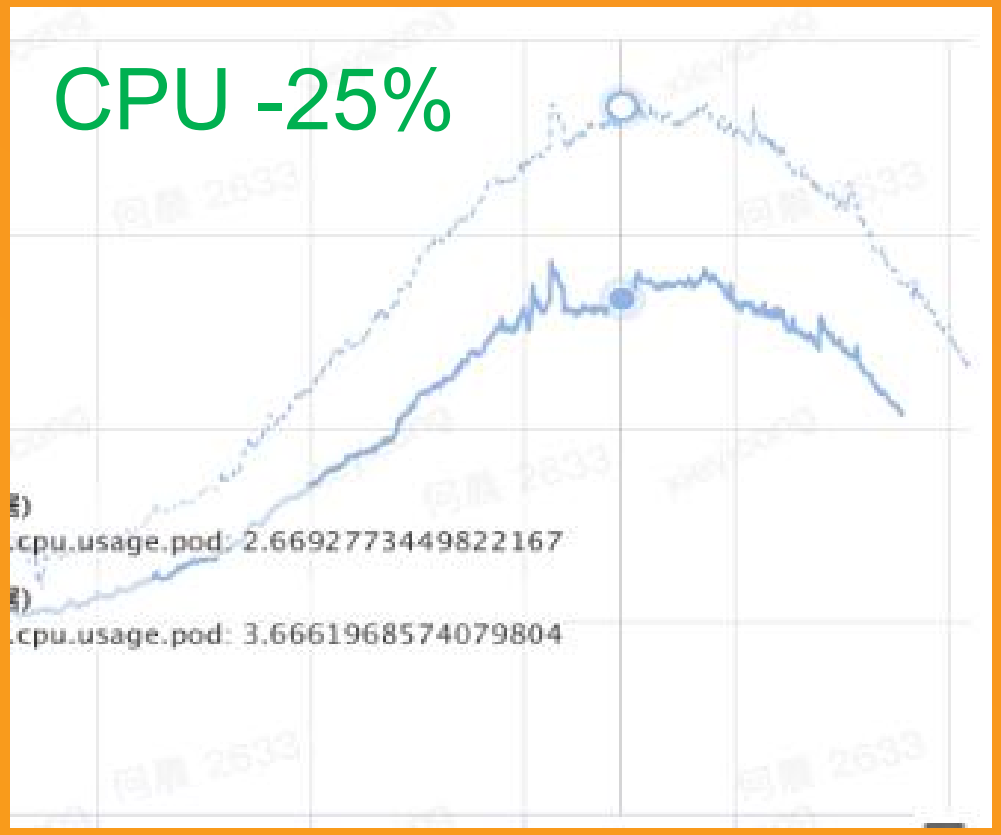
# Netpoll - 性能表现

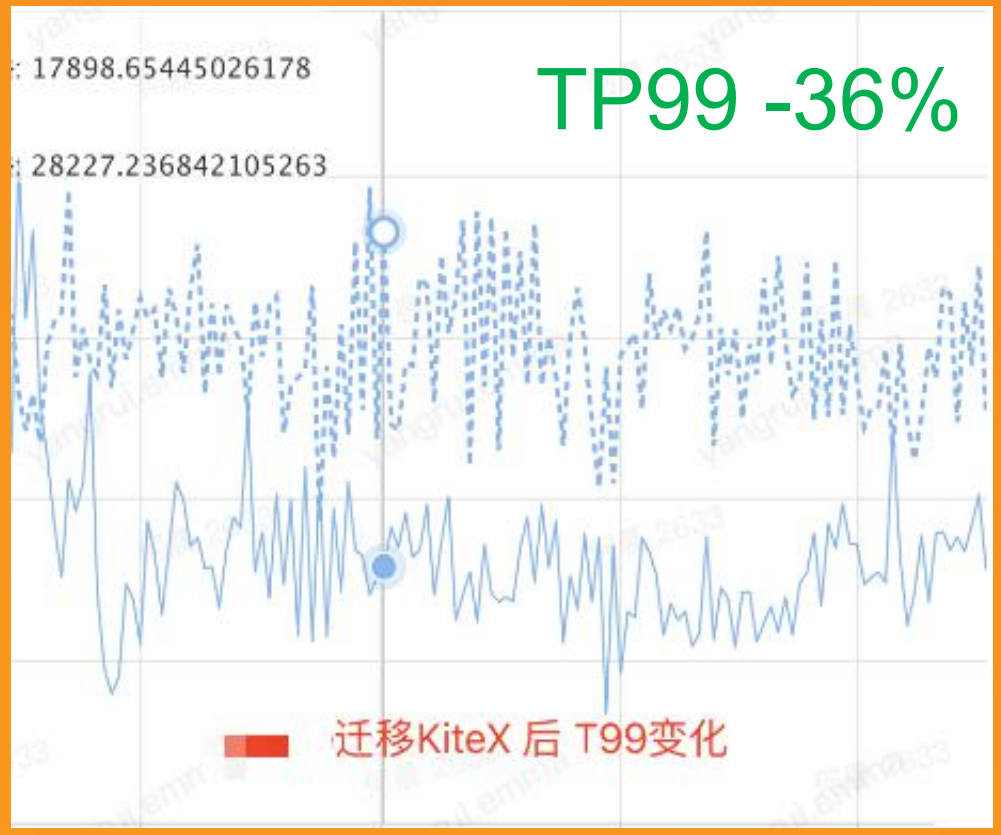
| Thrift RPC (echo 1KB) | Netpoll | Go net |
| --- | --- | --- |
| QPS | 2.5x | 1.0x |
| TP99 | 0.34x | 1.0x |

Environment

CPU: 4 cores
Memory: 8GB
Go: 1.15.4

# Netpoll – 业务实测表现



GopherChina 2021

GopherChina 2021

# Go net 在 RPC 场景下的问题

1. Conn 难以探活，
   维护连接池成本高

```
conn := connpool.Get(address)
// Is conn active ?
conn.Write(request)

// Is conn active ?
connpool.Put(conn)
```

# Go net 在 RPC 场景下的问题

1. Conn 难以探活，
   维护连接池成本高

2. BIO 式编程，
   连接量大时，调度开销大

```
go func() {
  for {
    conn, _ := listener.Accept()
    go func() {
      conn.Read(request)

      handle ...

      conn.Write(response)
    }
  }
}
```

# Go net 在 RPC 场景下的问题

1. Conn 难以探活，
   维护连接池成本高

2. BIO 式编程，
   连接量大时，调度开销大

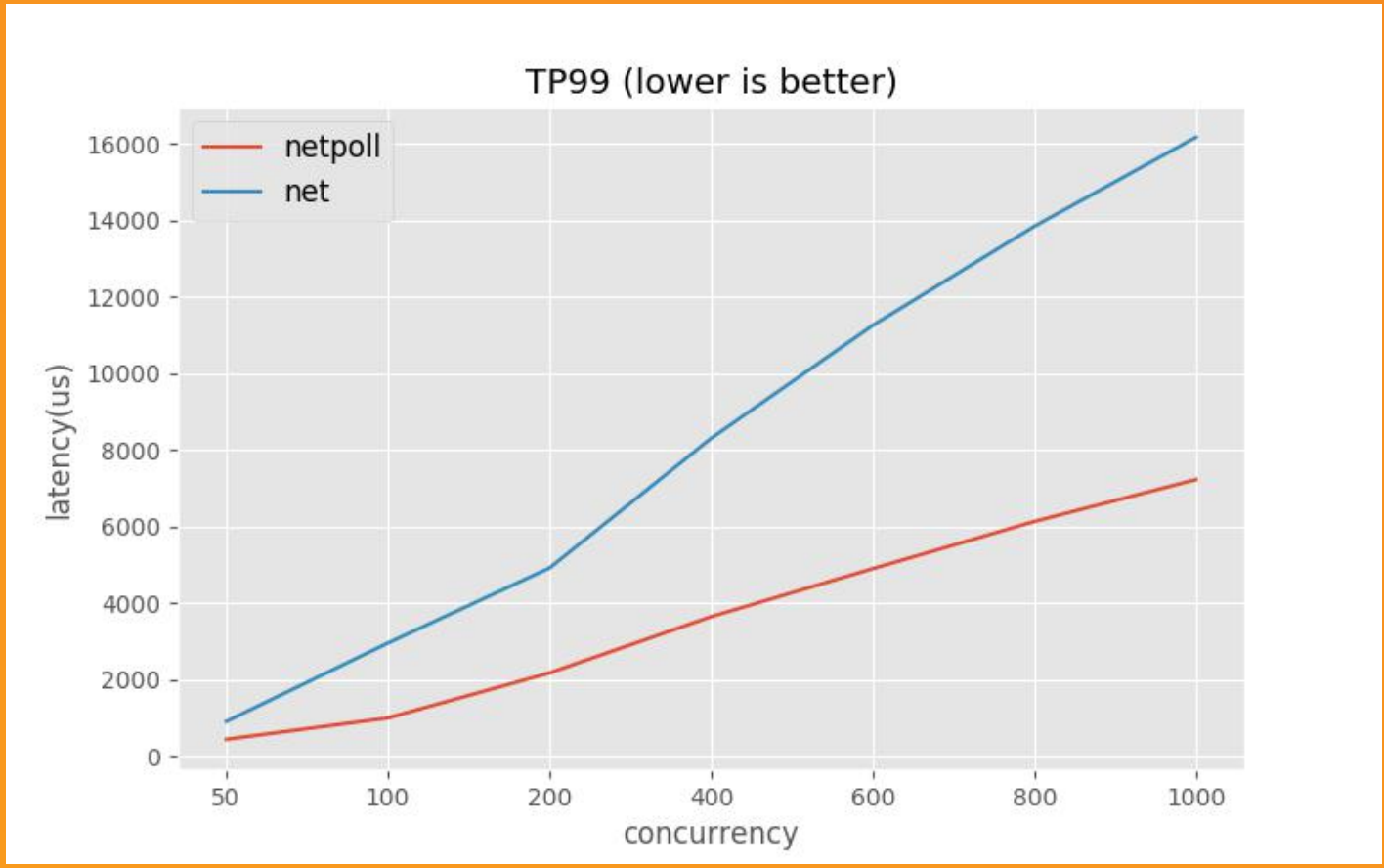
```
go func() {
  conn, _ := listener.Accept()
  epoll_ctl(conn.fd, readable...)
}

go func() {
  events := make([]event, 128)
  for {
    n, _ := epoll_wait(epoll_fd, events, wait_msec)
    for i:=0; i<n; i++{
      go func(){
        handle events[i] ...
      }
    }
  }
}
```

# Go net 在 RPC 场景下的问题

1. Conn 难以探活，
   维护连接池成本高

2. BIO 式编程，
   连接量大时，调度开销大



# 业界调研

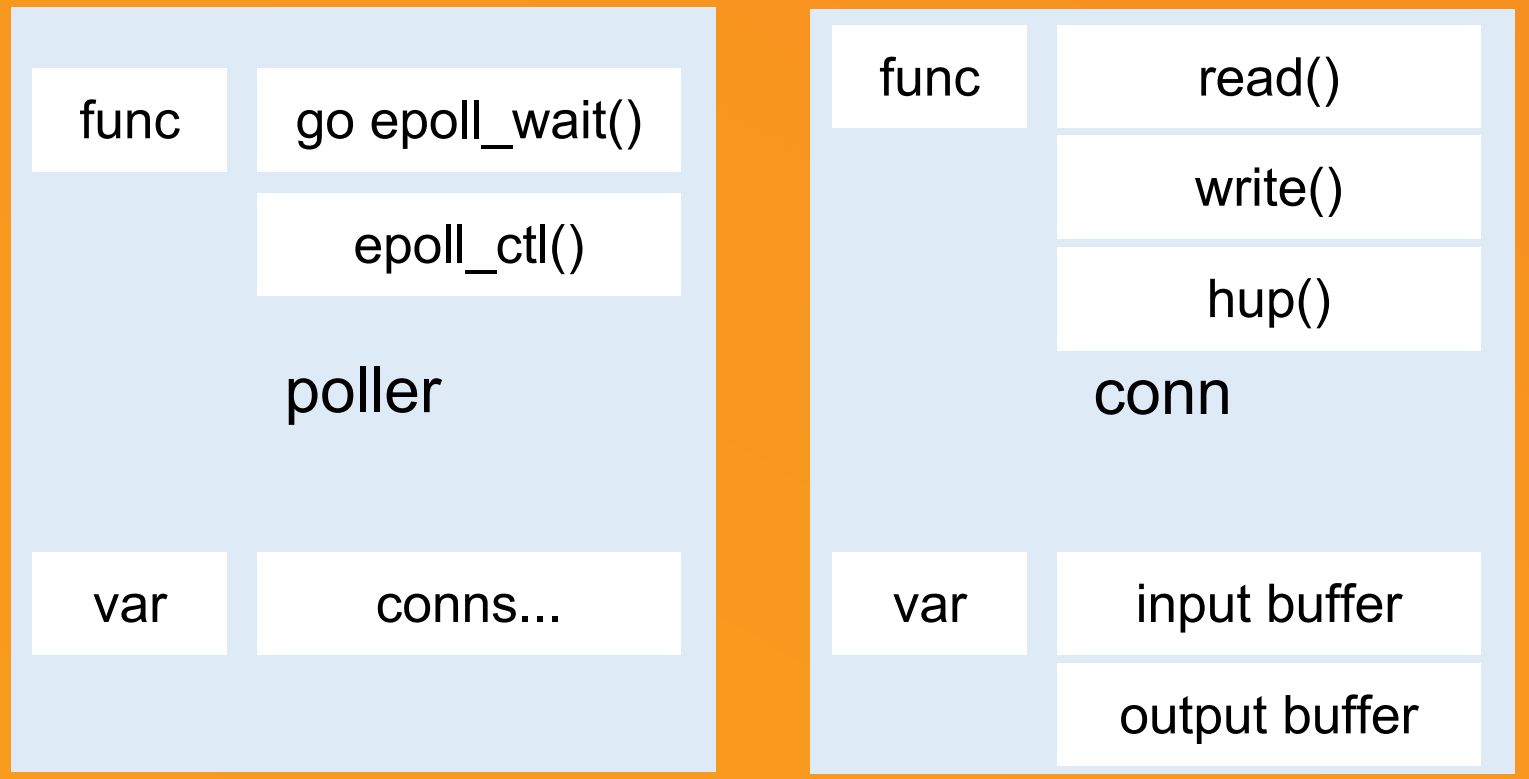
| | **netpoll** | **gnet** | **easygo (sofa-mosn)** | **evio** | **go net** |
|---|---|---|---|---|---|
| Epoll(ET/LT) | LT | LT | ET/LT | LT | ET |
| NIO | √ | √ | √ | √ | |
| ZeroCopy Buffer | √ | | | | |
| Multisyscall | √ | | | | |

# 搭建 Netpoll

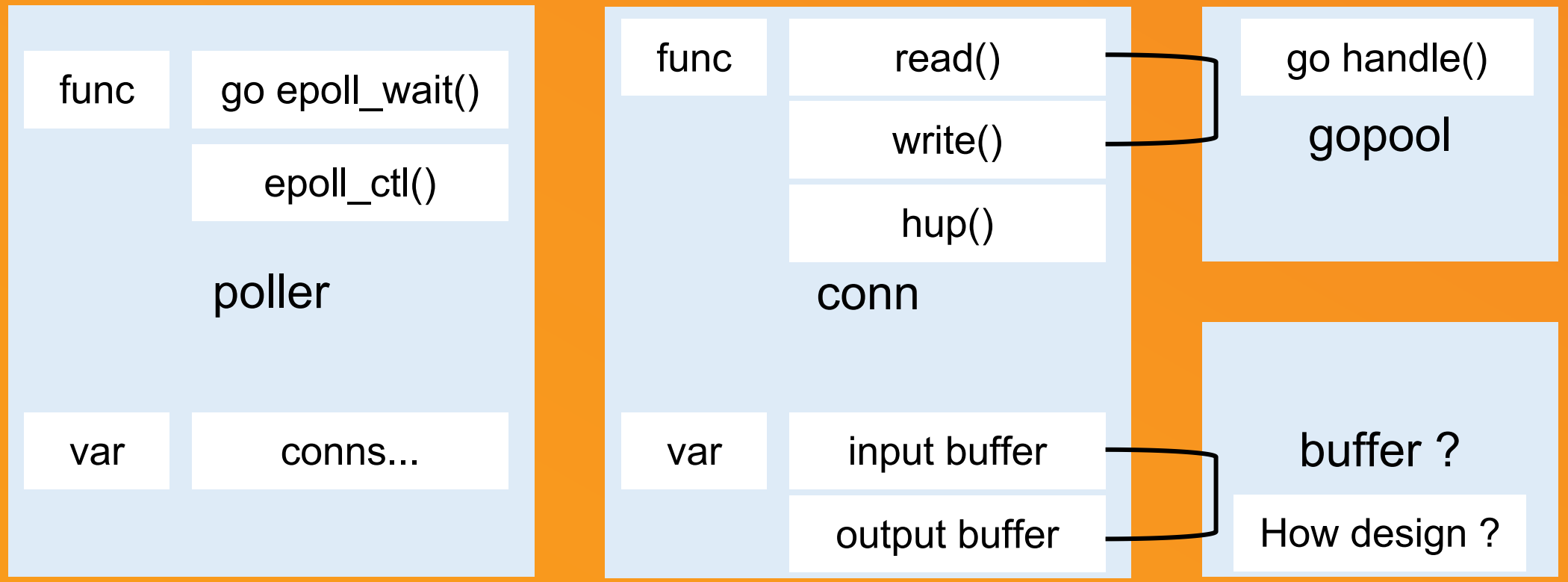
| | |
|---|---|
| func | go epoll_wait() |
| | epoll_ctl() |
| | poller |
| var | conns... |

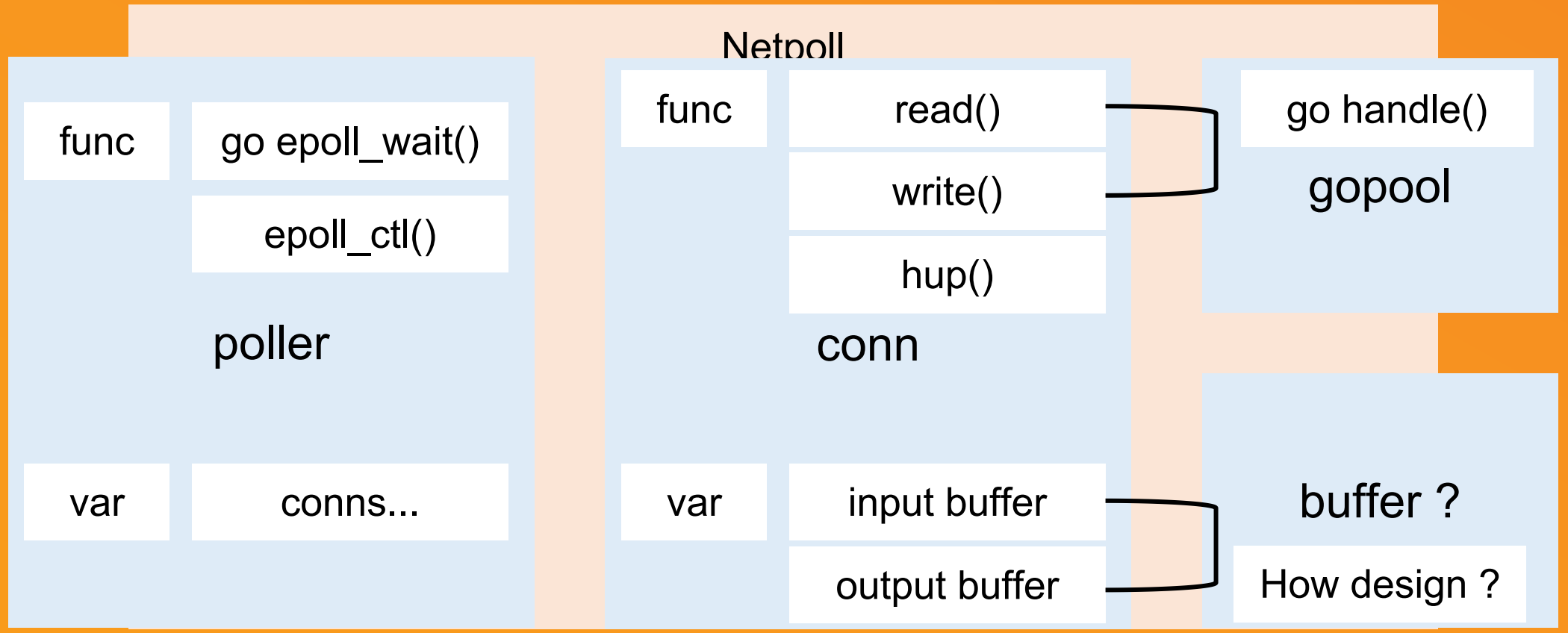
```
go func() {
  events := make([]event, 128)
  for {
    n, _ := epoll_wait(epoll_fd, events, msec)
    for i:=0; i<n; i++ {
        Read()/Write()/Catch(error)
    }
  }
}
```

# 搭建 Netpoll


func
go epoll_wait()
epoll_ctl()
poller
var
conns...
func
read()
write()
hup()
conn
var
input buffer
output buffer

# 搭建 Netpoll


func
go epoll_wait()
epoll_ctl()
poller
var
conns...
func
read()
write()
hup()
conn
var
input buffer
output buffer
go handle()
gopool
buffer ?
How design ?


Netpoll
func
go epoll_wait()
epoll_ctl()
poller
var
conns...
func
read()
write()
hup()
conn
go handle()
gopool
var
input buffer
output buffer
buffer ?
How design ?

GopherChina 2021

# 优化方向

优化调度效率(poller)

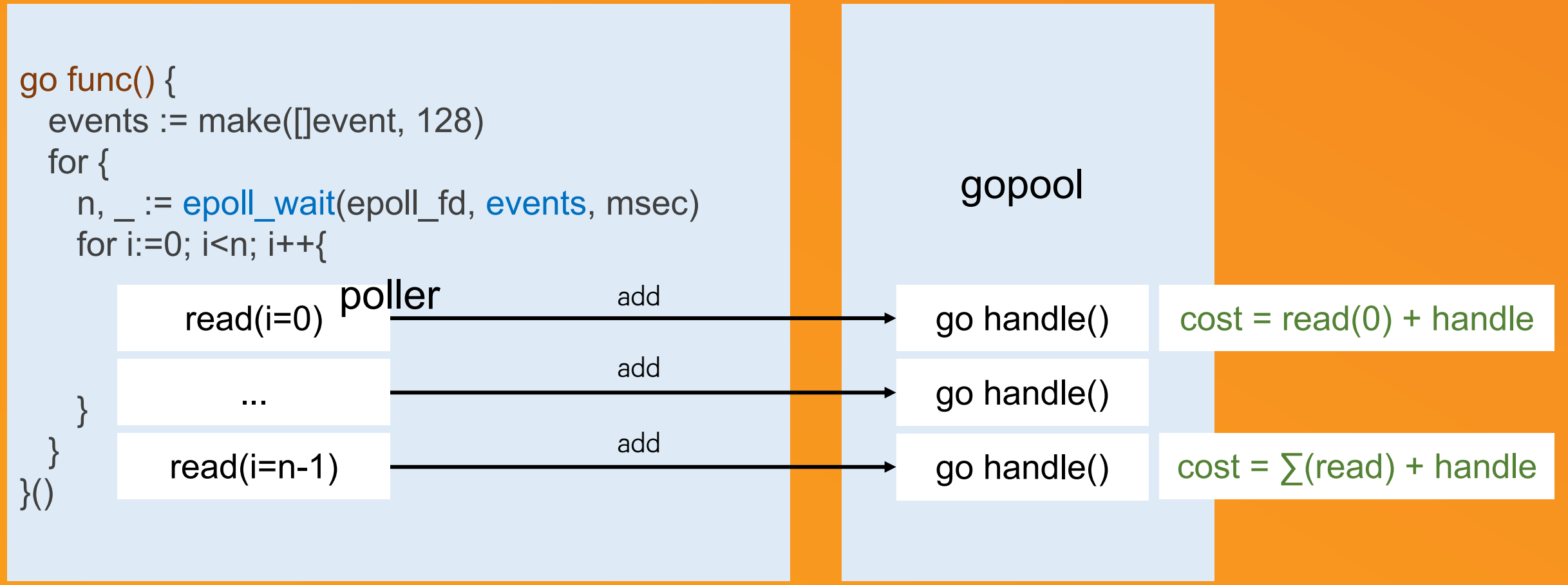
优化 Buffer 设计(zerocopy)

# 优化方向

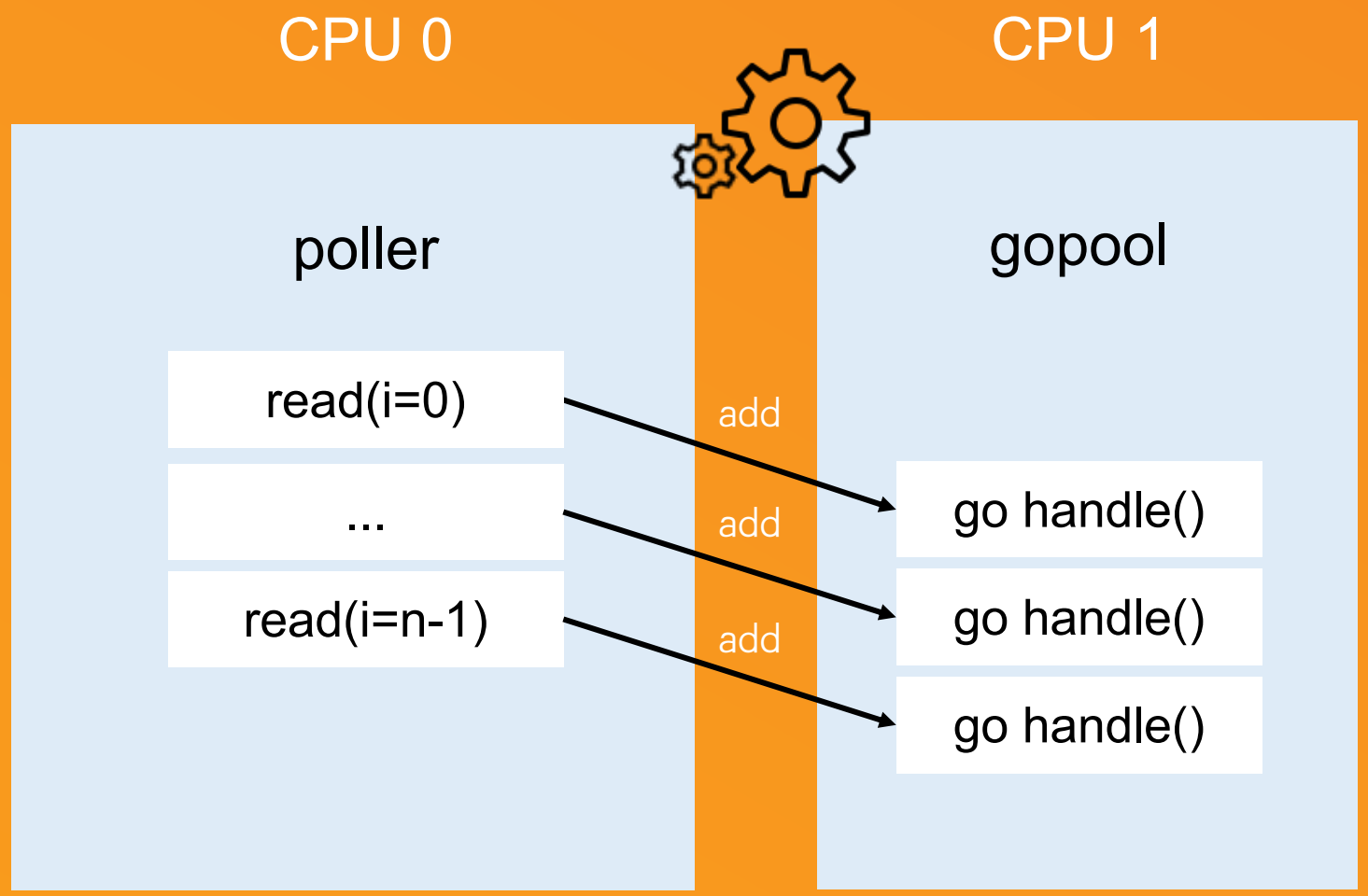
优化调度效率(poller)

优化 Buffer 设计(zerocopy)

# 优化调度效率 – TP99 分析



# 优化调度效率 – 吞吐分析


CPU 0
CPU 1
poller
gopool
read(i=0)
...
read(i=n-1)
add
add
add
go handle()
go handle()
go handle()

# 优化调度效率 – 优化系统调用

```
func Read(fd int, p []byte) (n int, err error) {
  ...
  r, _, e := syscall.Syscall(SYS_READ, uintptr(fd), ...)
  ...
  return int(r), e
}
```

改前 ⬆

```
func Read(fd int, p []byte) (n int, err error) {
  ...
  r, _, e := syscall.RawSyscall(SYS_READ, uintptr(fd), ...)
  ...
  return int(r), e
}
```

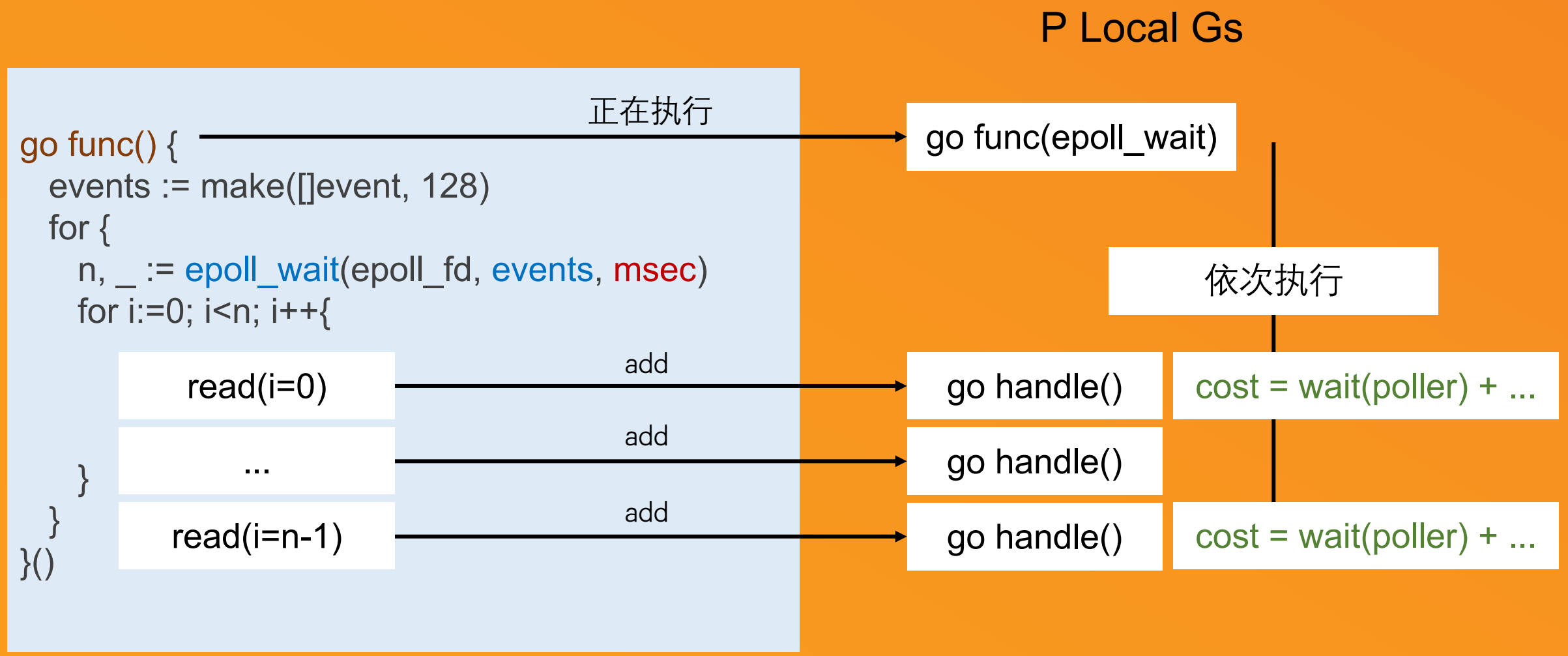
改后 ⬇

将 Syscall 改为 RawSyscall

Syscall 执行逻辑相当于

1. enter_runtime
2. raw_syscall
3. exit_runtime

# 优化调度效率 – 调度分析


P Local Gs
go func() {
events := make([]event, 128)
for {
n, _ := epoll_wait(epoll_fd, events, msec)
for i:=0; i<n; i++{
}
}
}()
正在执行
go func(epoll_wait)
依次执行
read(i=0)
add
go handle()
cost = wait(poller) + ...
...
add
go handle()
read(i=n-1)
add
go handle()
cost = wait(poller) + ...

# 优化调度效率 – 优化调度

```
func (p *poller) Wait() error {
  ...
  for {
    n, _ = EpollWait(p.fd, p.events, msec)
    ...
    if n <= 0 {
        msec = -1

        runtime.Gosched()
        continue
    }
    msec = 0

    handle p.events[:n] ...
  }
}
```

1.动态 msec, 加快调用速度

2.判断 n, 主动让出

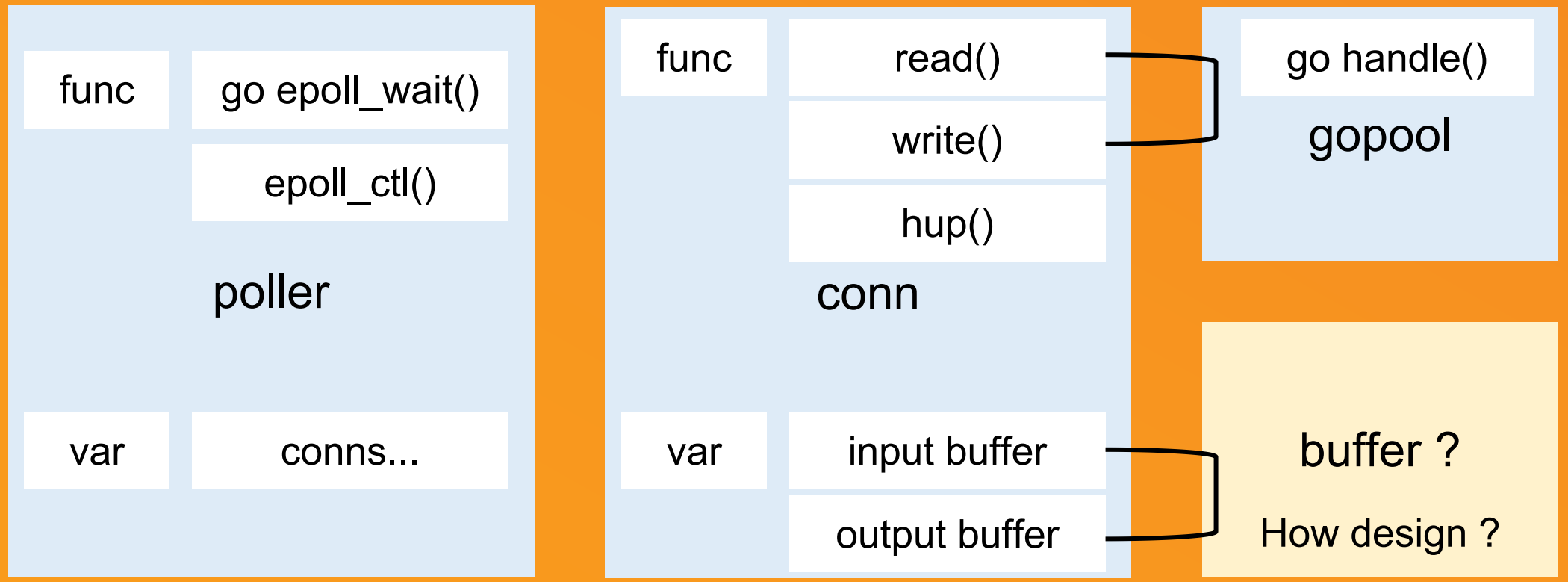
| Benchmark | time/op |
| --- | --- |
| EpollWait, msec=0 | 270 ns/op |
| EpollWait, msec=-1 | 328 ns/op |
| **Delta** | **-17.68%** |

# 优化方向

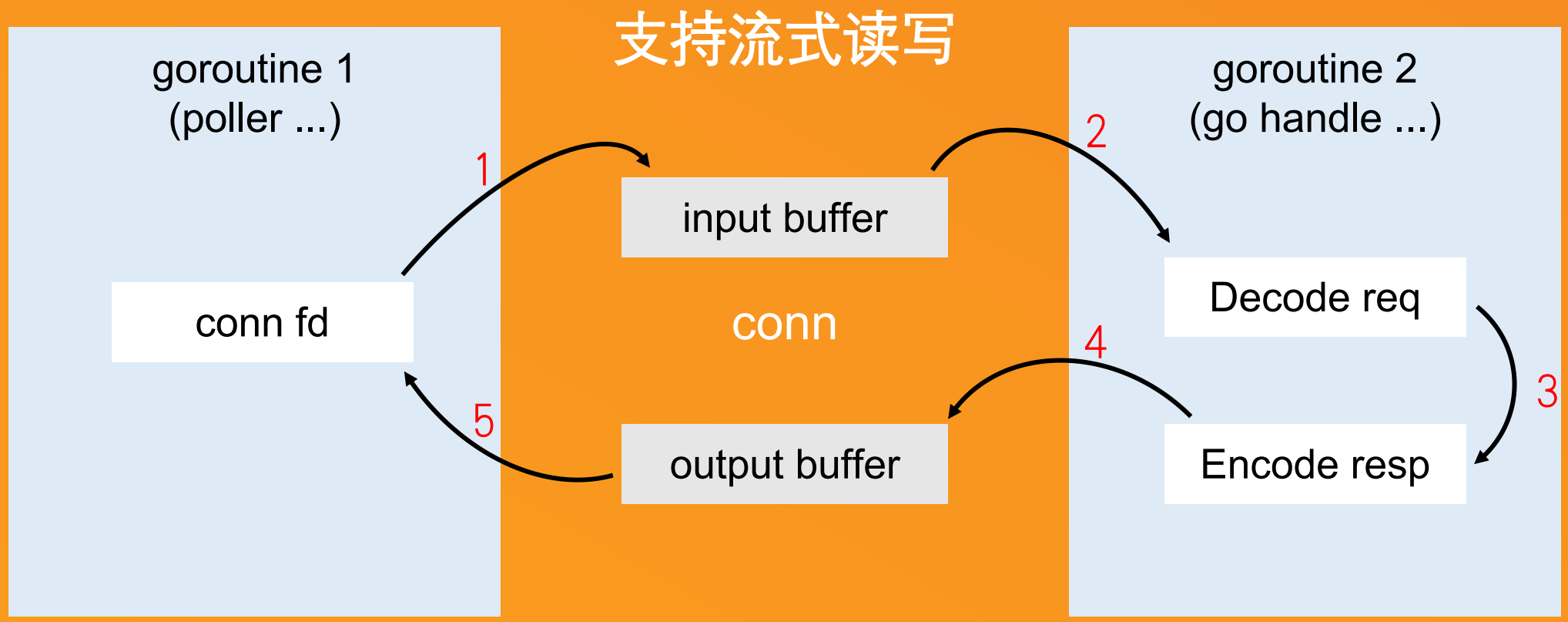
优化调度效率(poller)

优化 Buffer 设计(zerocopy)

# 优化 Buffer 设计


func
go epoll_wait()
epoll_ctl()
poller
var
conns...
func
read()
write()
hup()
conn
var
input buffer
output buffer
go handle()
gopool
buffer ?
How design ?

# 优化 Buffer 设计 – 需求分析

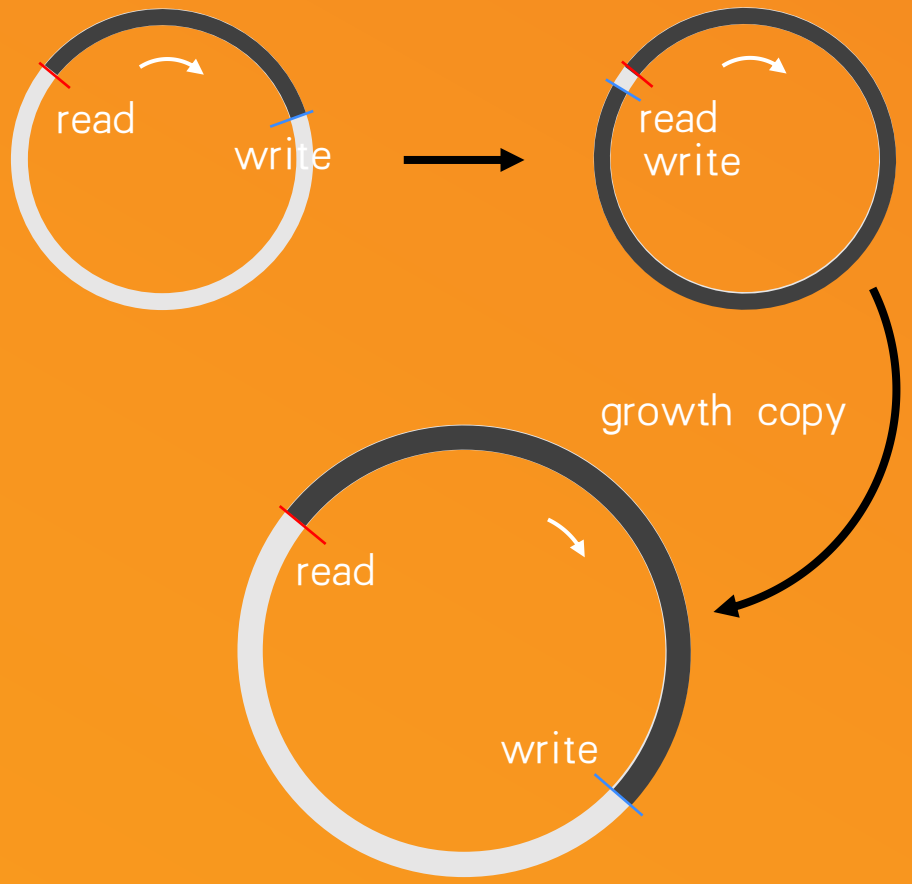


# 优化 Buffer 设计 - RingBuffer 分析

buffer(full) need growth

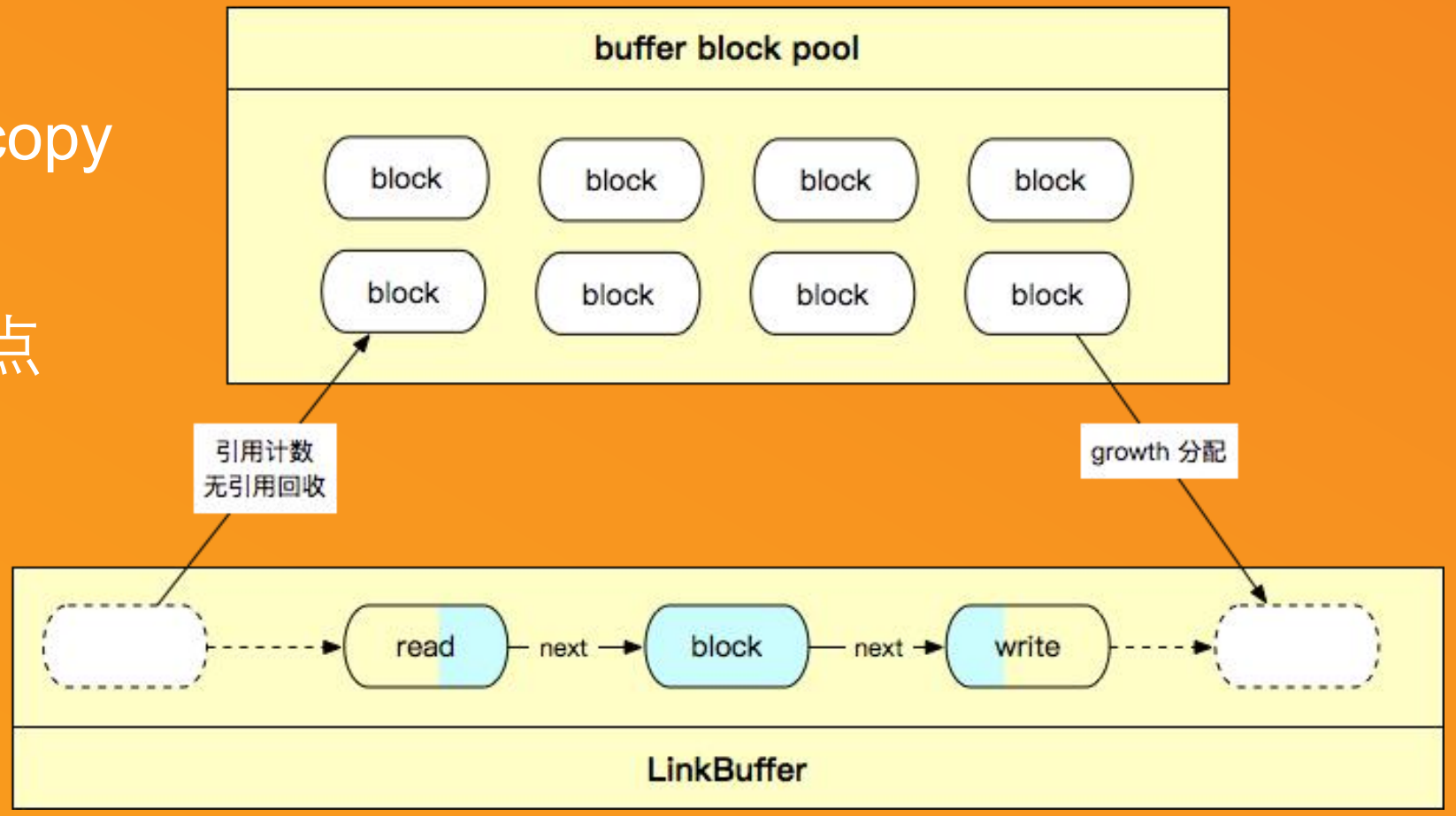
growth need copy

copy will data race



# 优化 Buffer 设计 – LinkBuffer 设计

1. 链表解决 growth copy
2. sync.Pool 复用节点
3. atomic 访问 size
   解决 data race



# Q: 为什么业界没使用 LinkBuffer ?

A: 无法使用 Read/Write API

Read([]byte), Write([]byte)

readv([][]byte), writev([][]byte)

# writev/readv 实现

```go
func writev(fd int, bs [][]byte, ...) (n int, err error) {
    ...
    r, _, e := syscall.RawSyscall(syscall.SYS_WRITEV, uintptr(fd), ...)
    ...
    return int(r), nil
}
```

```go
func readv(fd int, bs [][]byte, ...) (n int, err error) {
    ...
    r, _, e := syscall.RawSyscall(syscall.SYS_READV, uintptr(fd), ...)
    ...
    return int(r), nil
}
```

# 性能亮点 - 小结

## 优化调度效率(poller)

1. RawSyscall
2. runtime.Gosched
3. msec 动态调参

## 优化 Buffer 设计(nocopy)

1. LinkBuffer
2. readv/writev

GopherChina 2021

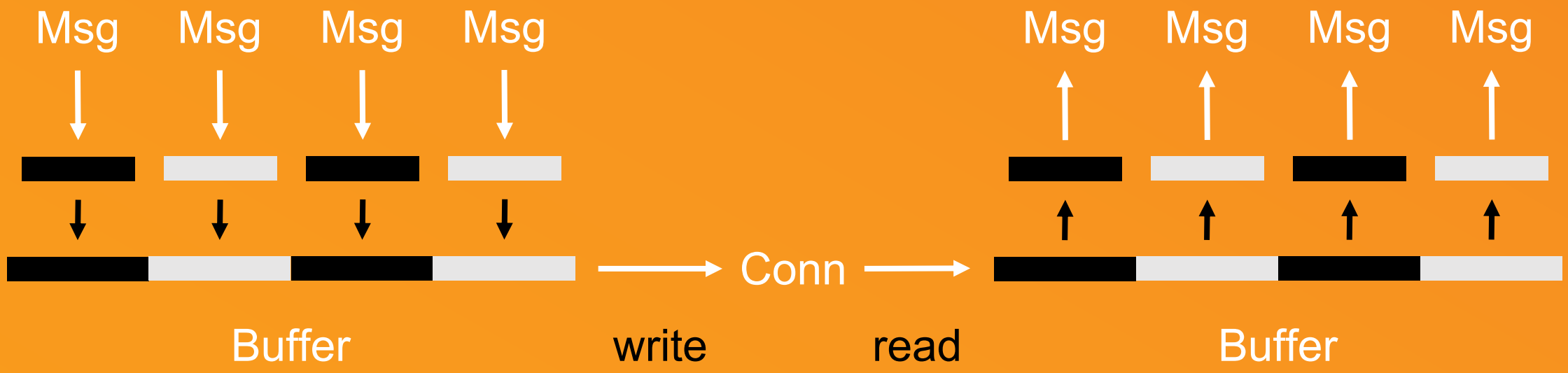
1. 单连接多路复用(ZeroCopy)

2. TCP ZeroCopy

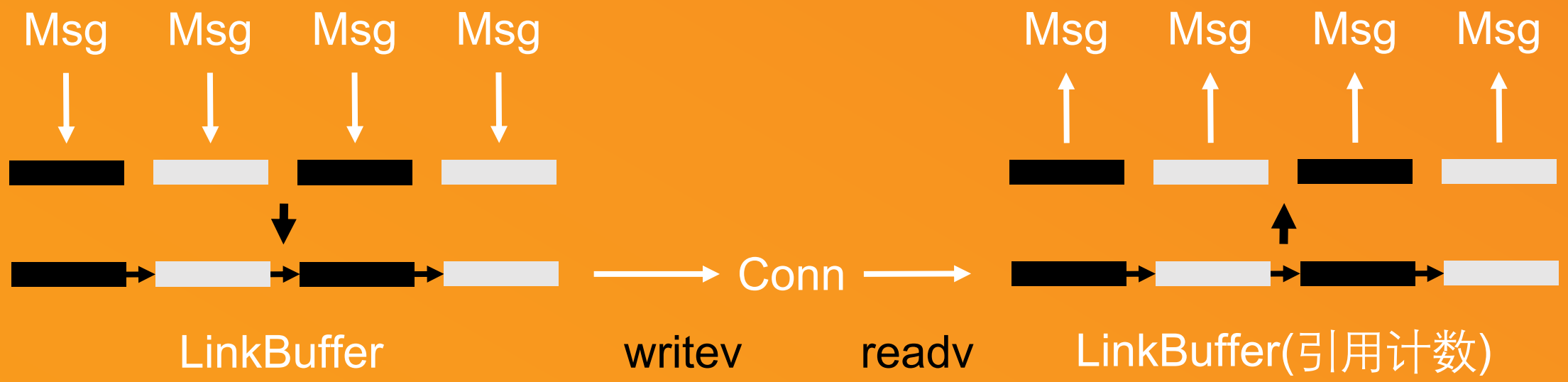
3. Multisyscall

4. io_uring

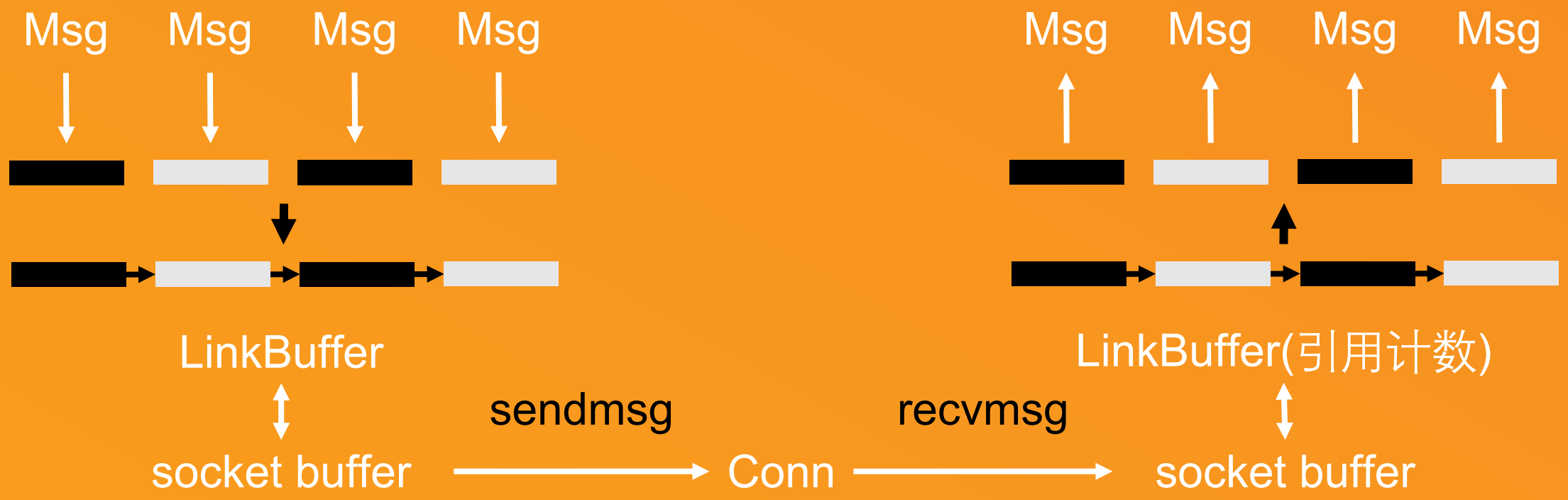
# 单连接多路复用 – 组包/拆包



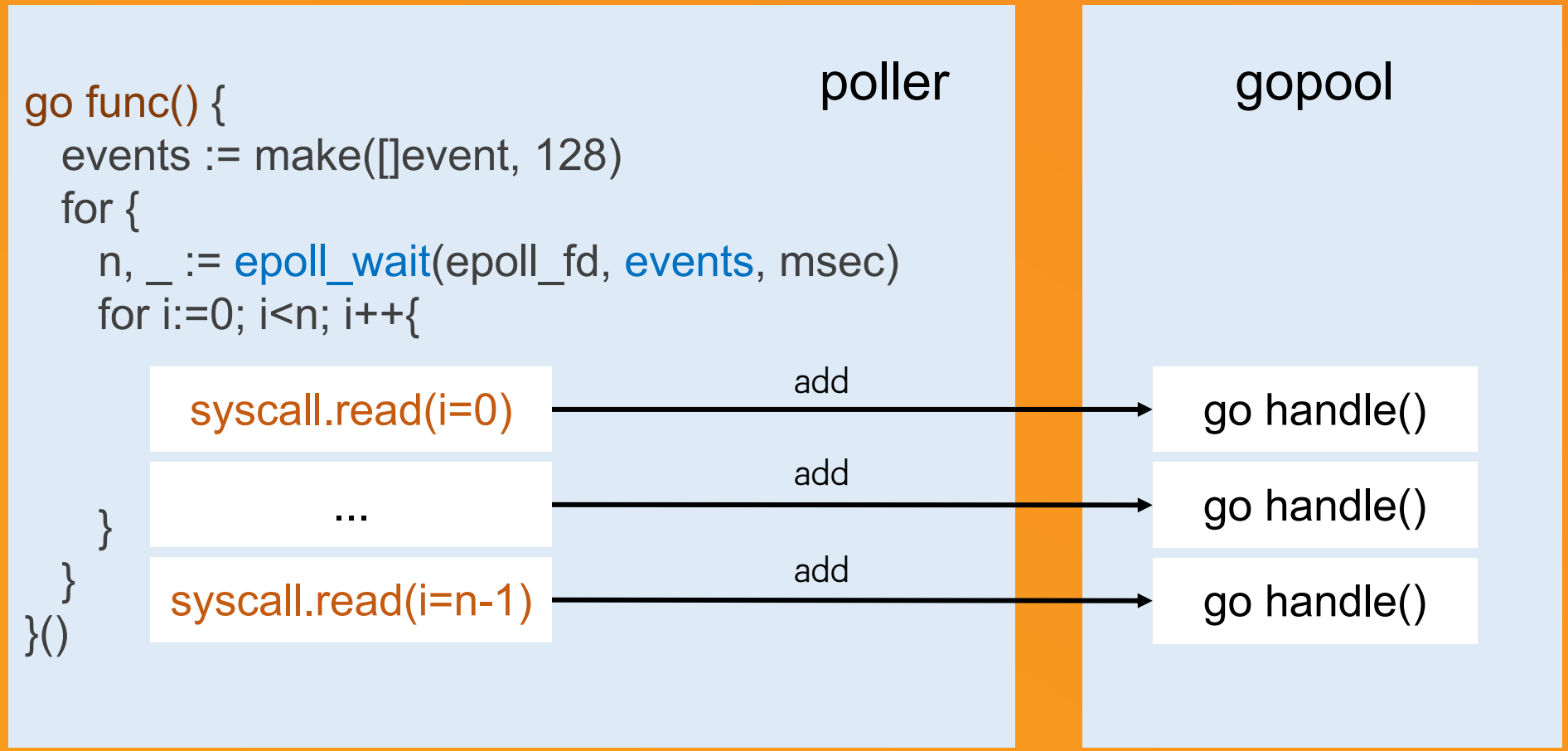
# 单连接多路复用 – ZeroCopy LinkBuffer



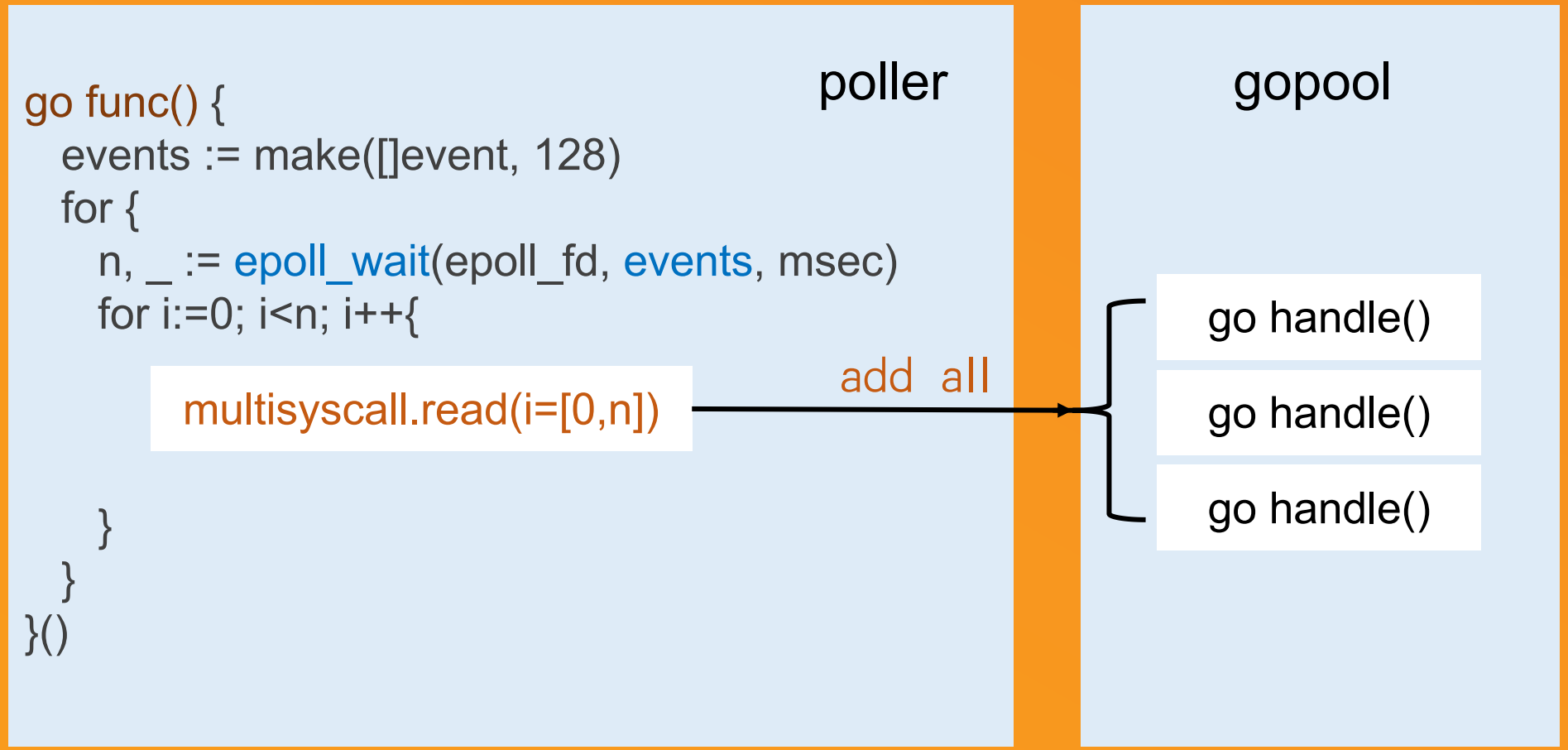
# TCP ZeroCopy



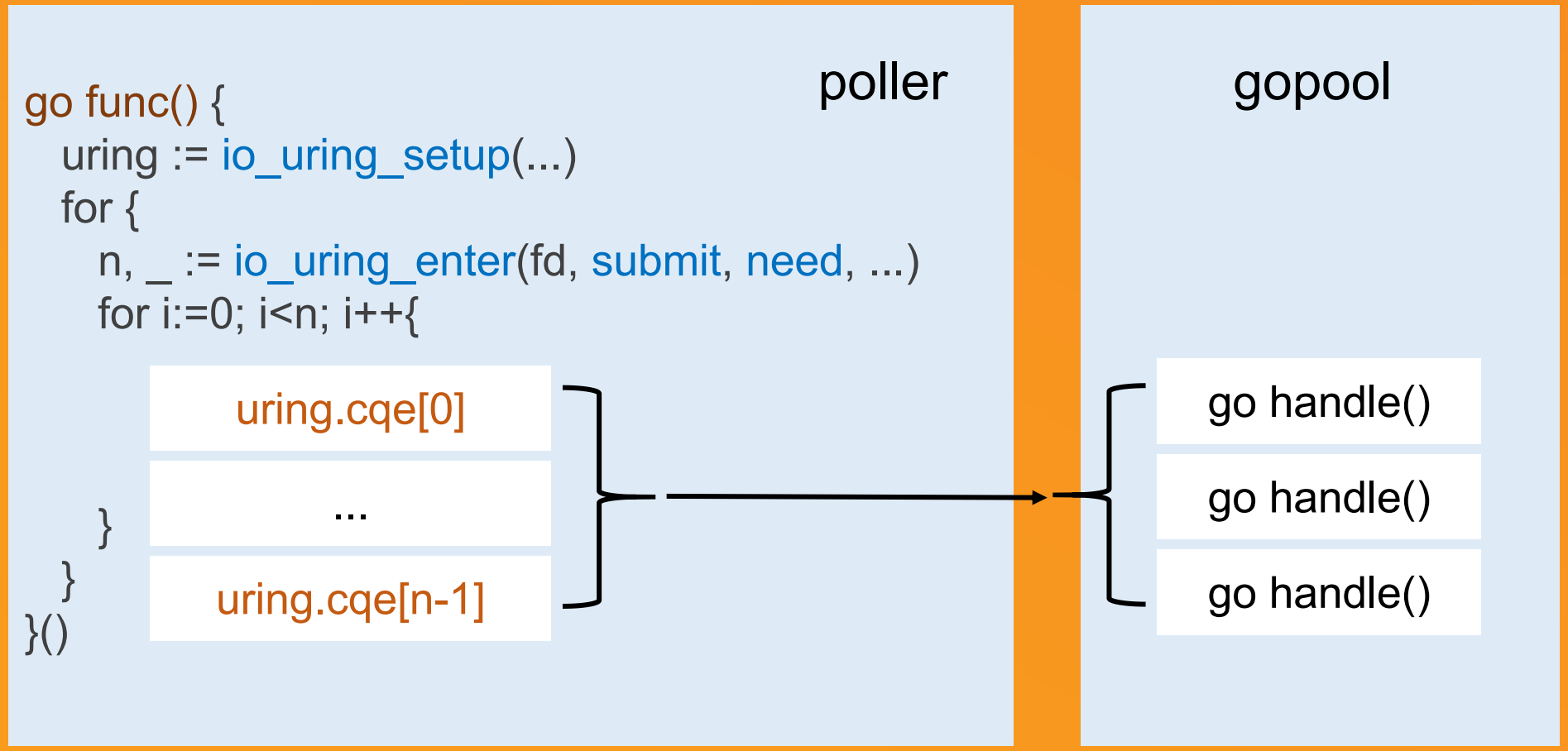
# Multisyscall – 常规 poller



# Multisyscall – 批量调用



# io_uring – 异步调用



GopherChina 2021

新思路：unsafe，mcache(no gc)，...

新技术(火山引擎)：share memory IPC，...

场景特化(火山引擎)：同机部署，纯计算/cache ...

# Thanks

contact us