



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

应用层

RPC 框架

KiteX

HTTP 框架

Hertz



网络层

Netpoll

Go net

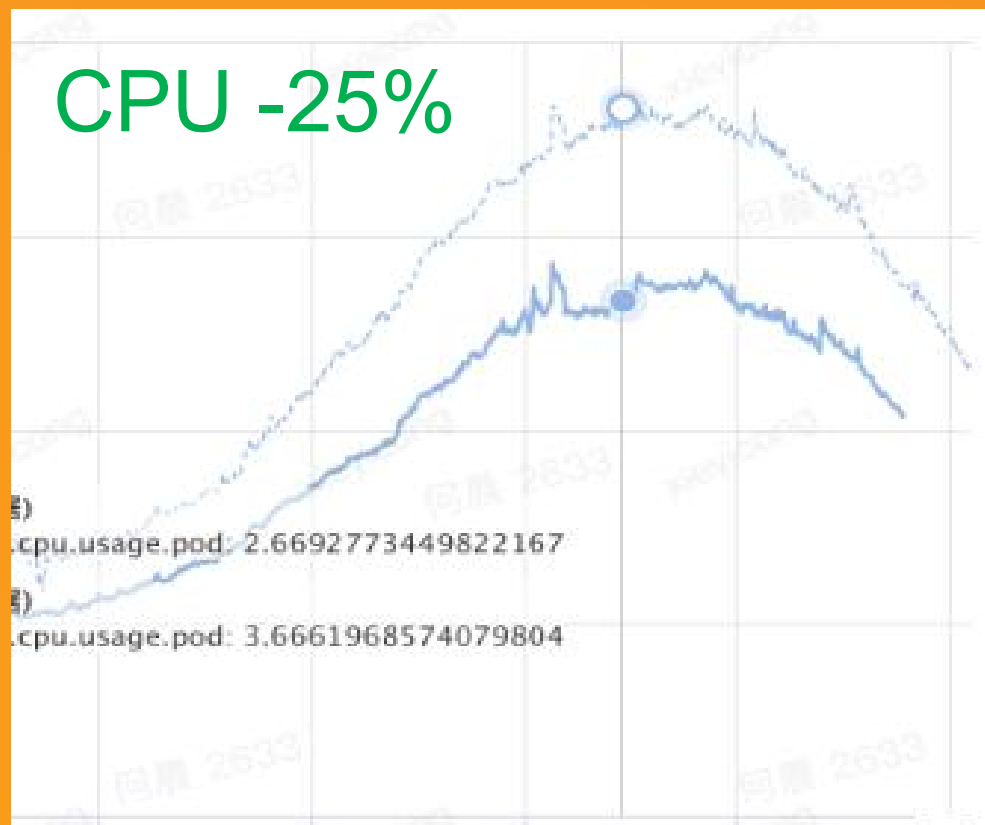
Netpoll – 性能表现

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

CPU: 4 cores
Memory: 8GB
Go: 1.15.4



Netpoll - 业务实测表现





GopherChina 2021

设计实现

01

性能亮点

02

高级特性

03

展望未来

04



GopherChina 2021

设计实现

01

性能亮点

02

高级特性

03

展望未来

04

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)  
        }  
    }  
}
```

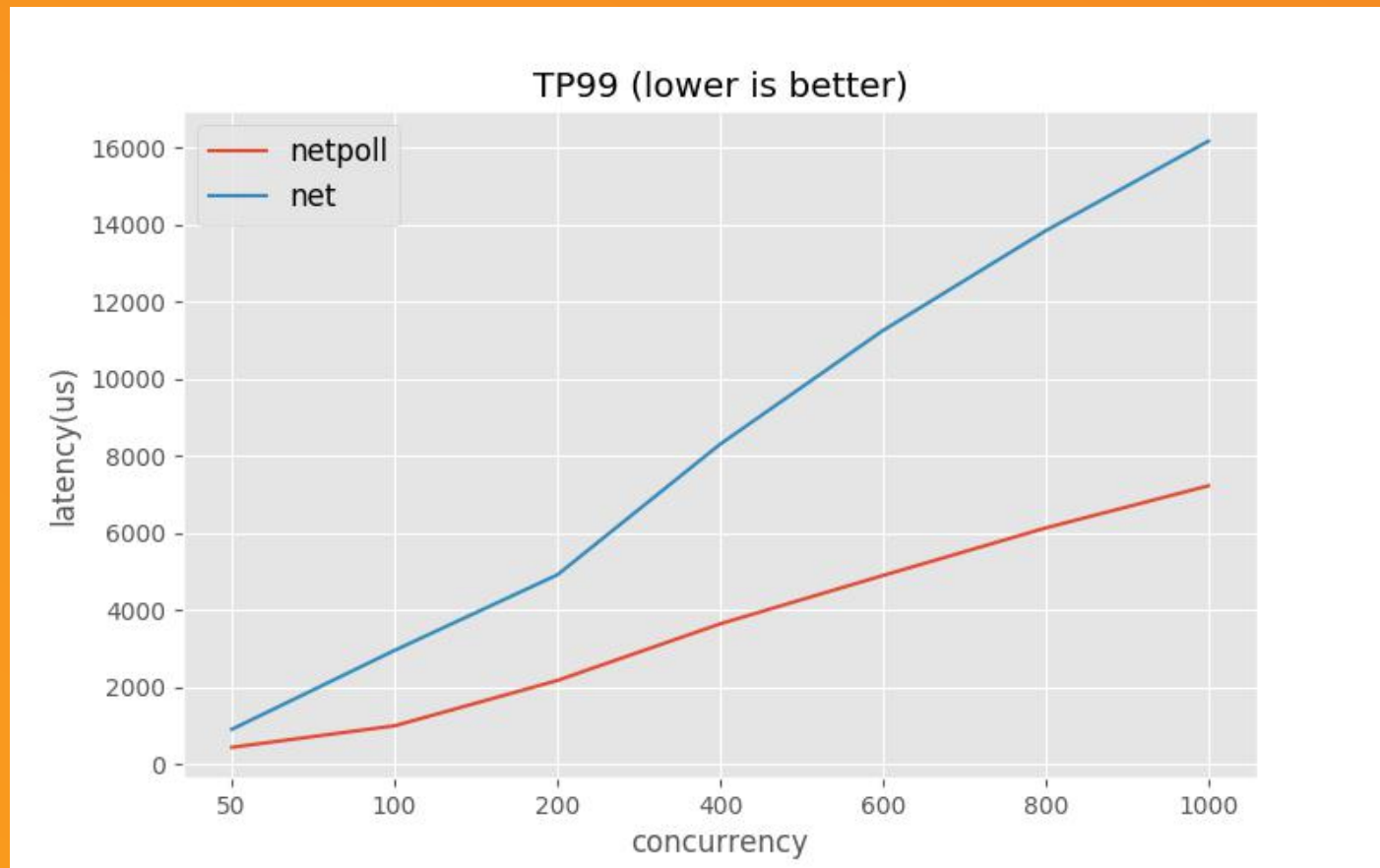

Go net 在 RPC 场景下的问题

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

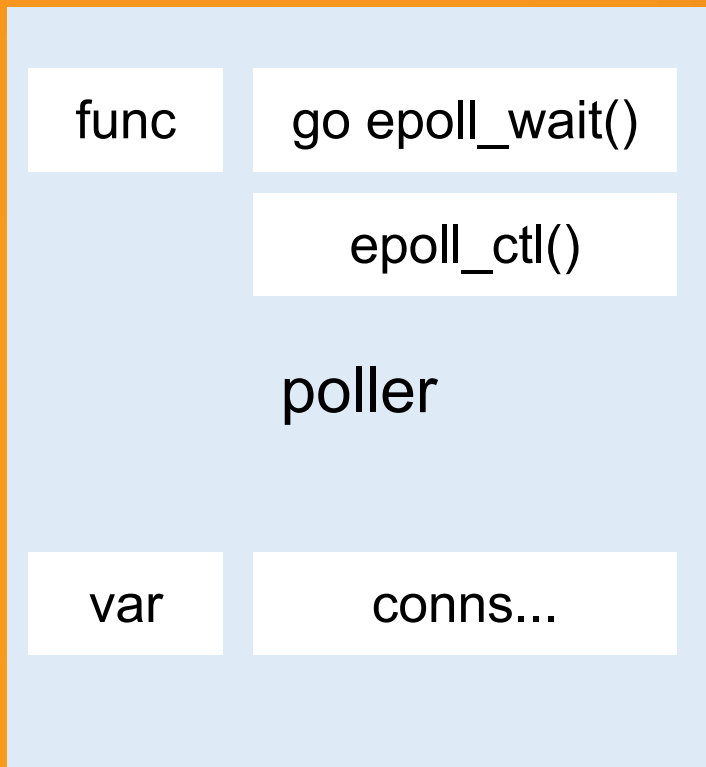
1. Conn 难以探活,
维护连接池成本高
2. BIO 式编程,
连接量大时, 调度开销大



业界调研

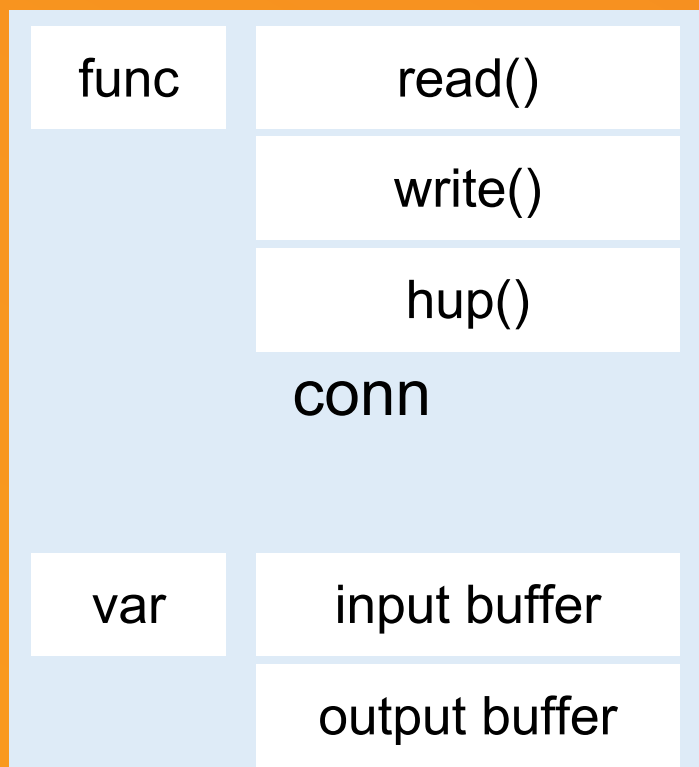
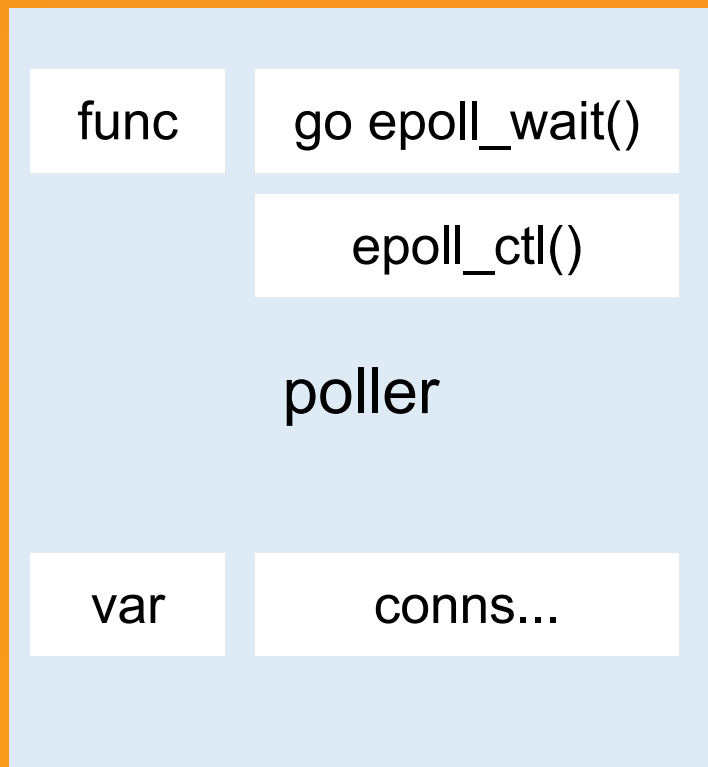
	netpoll	gnet	easygo (sofa-mosn)	evio	go net
Epoll(ET/LT)	LT	LT	ET/LT	LT	ET
NIO	√	√	√	√	
ZeroCopy Buffer	√				
Multisyscall	√				

搭建 Netpoll

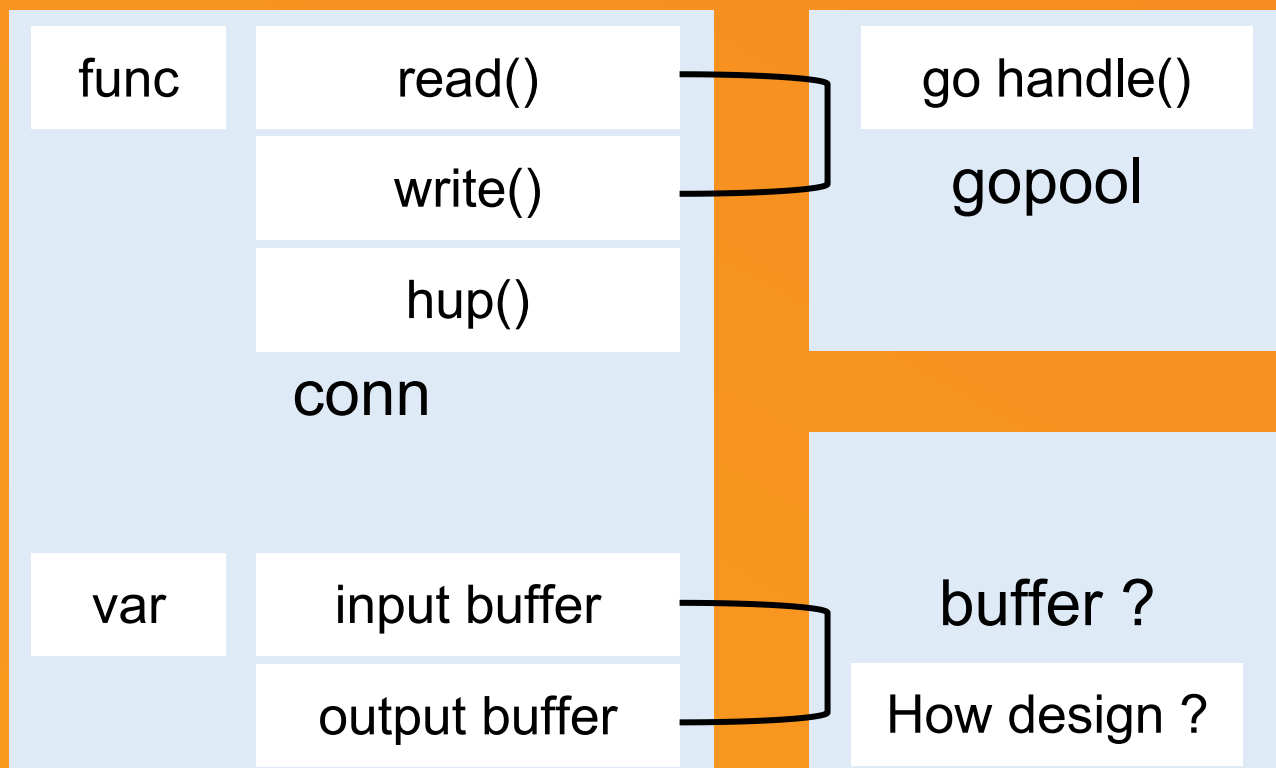
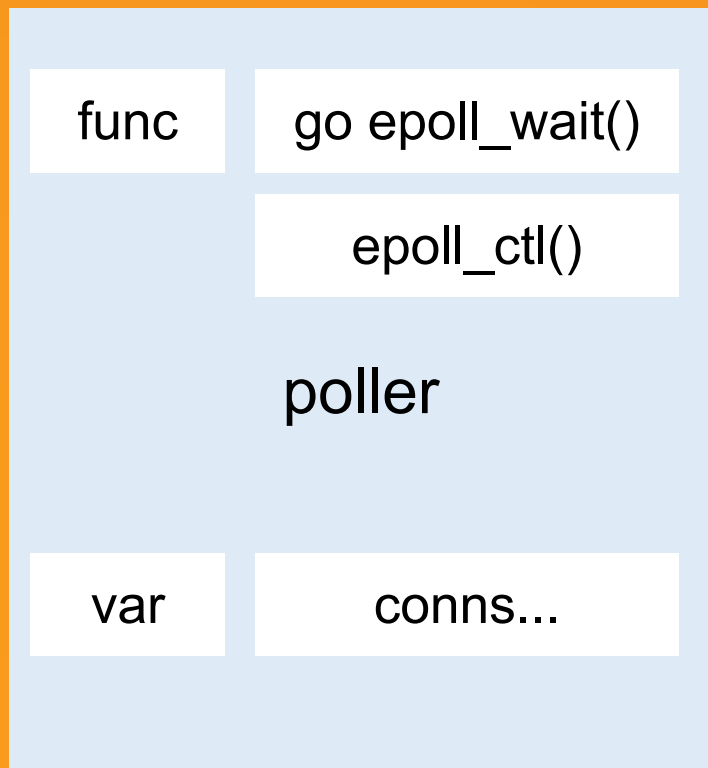


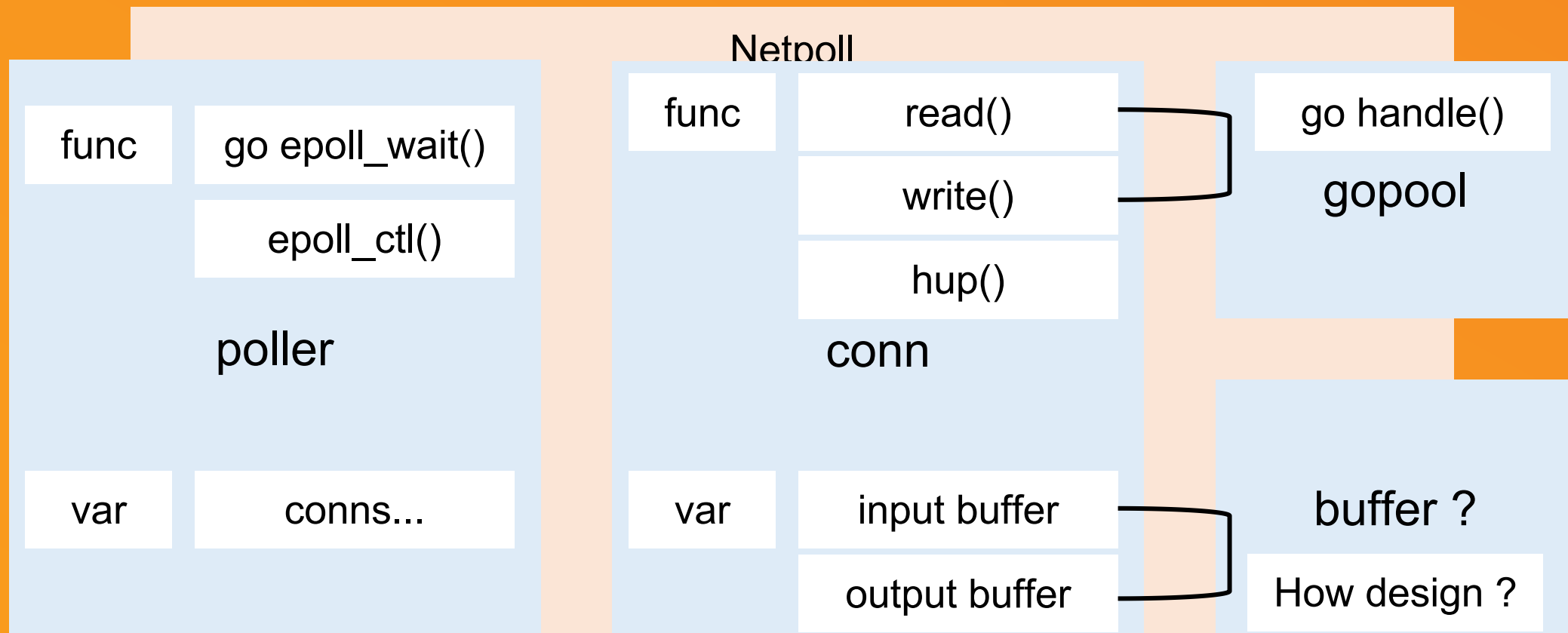
```
go func() {  
    events := make([]event, 128)  
    for {  
        n, _ := epoll_wait(epoll_fd, events, msec)  
        for i:=0; i<n; i++ {  
            Read()/Write()/Catch(error)  
        }  
    }  
}
```

搭建 Netpoll



搭建 Netpoll







GopherChina 2021

设计实现

01

性能亮点

02

高级特性

03

展望未来

04

优化方向

优化调度效率(poller)

优化 Buffer 设计(zerocopy)

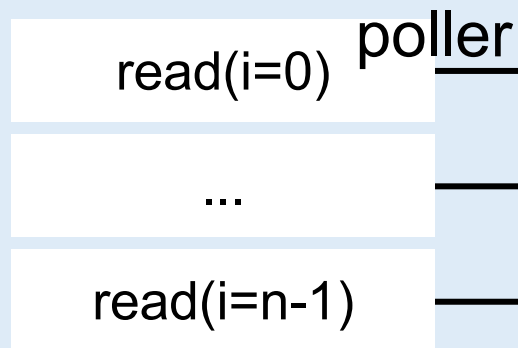
优化方向

优化调度效率(poller)

优化 Buffer 设计(zerocopy)

优化调度效率 - TP99 分析

```
go func() {  
    events := make([]event, 128)  
    for {  
        n, _ := epoll_wait(epoll_fd, events, msec)  
        for i:=0; i<n; i++{  
            read(i=0)  
            ...  
            read(i=n-1)  
        }  
    }  
}()
```



gopool

go handle()

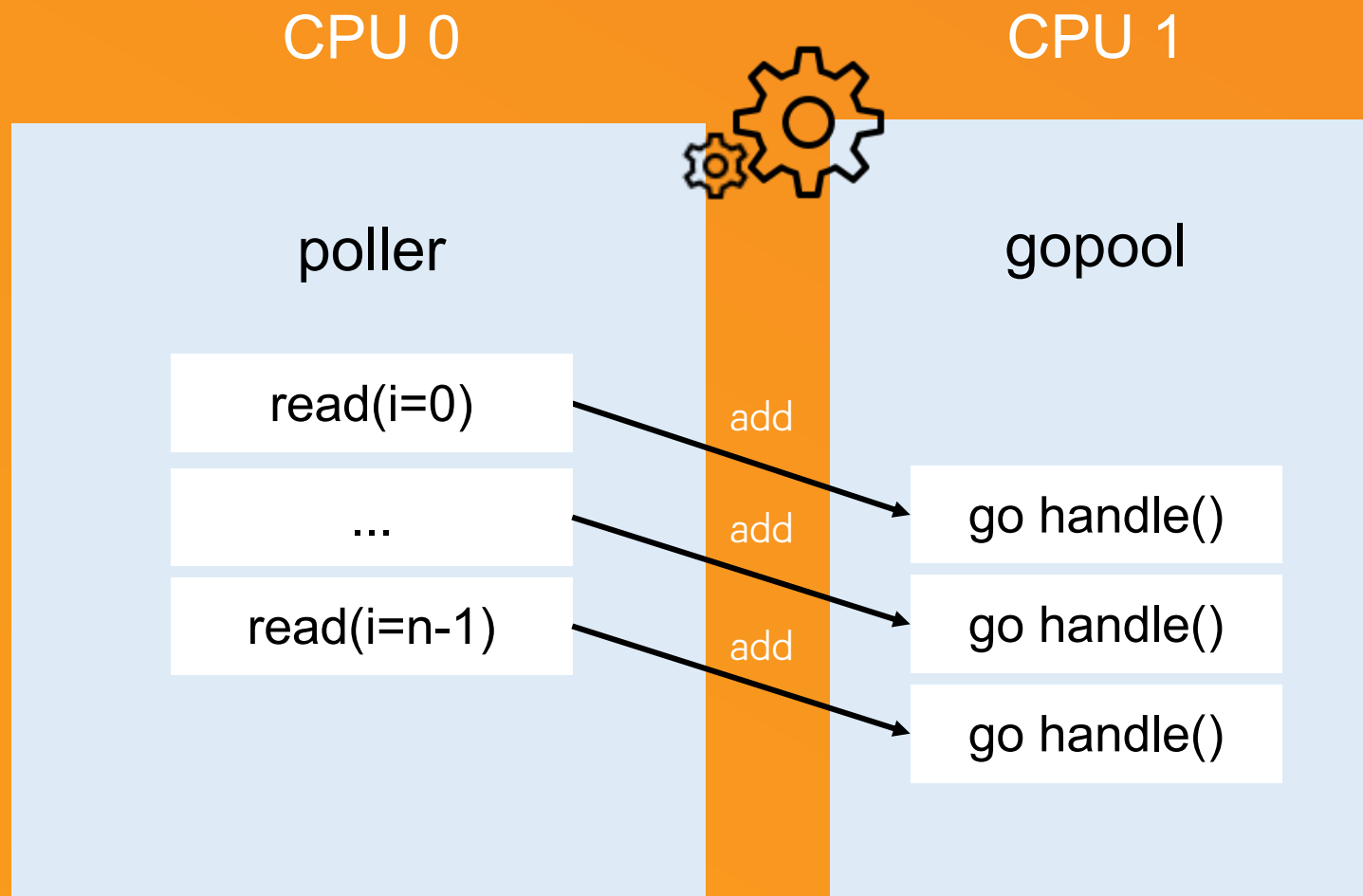
cost = read(0) + handle

go handle()

go handle()

cost = $\sum(\text{read}) + \text{handle}$

优化调度效率 - 吞吐分析



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  
}
```

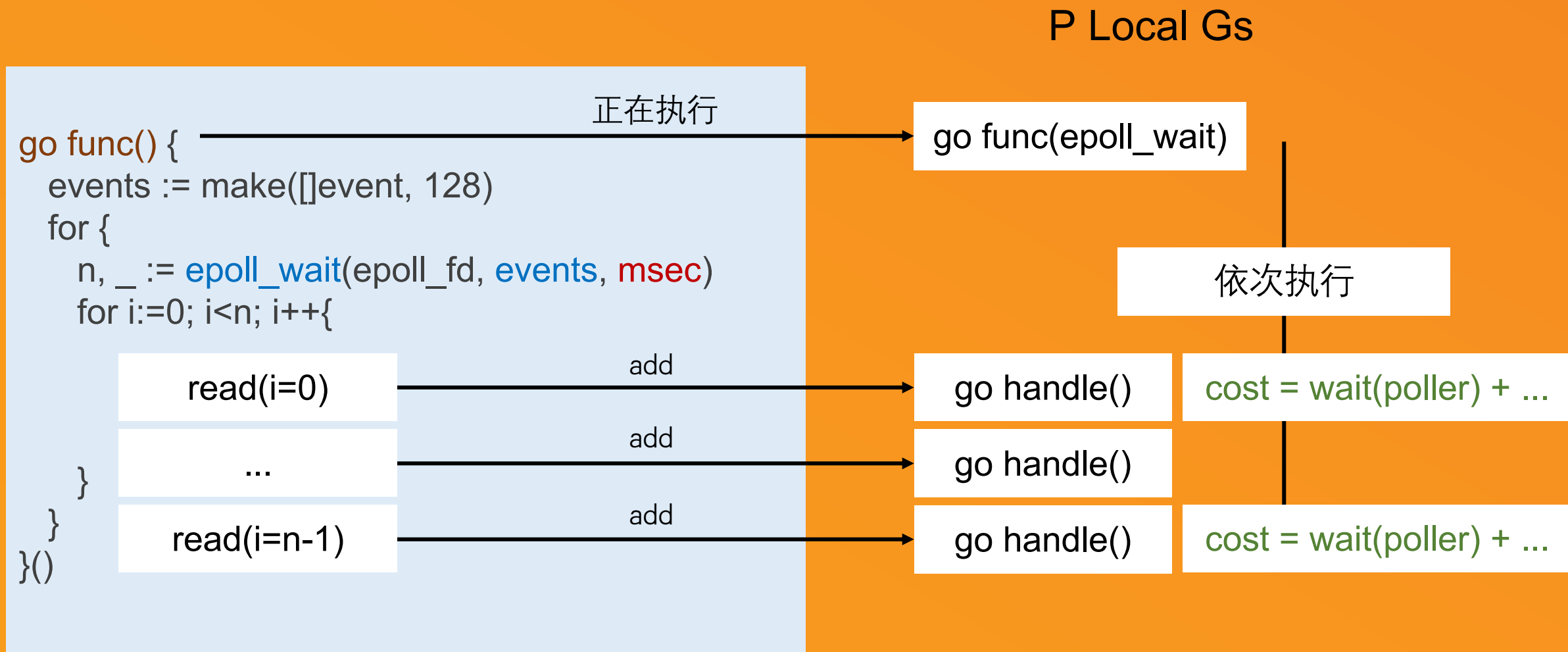
改后 ↓

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



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

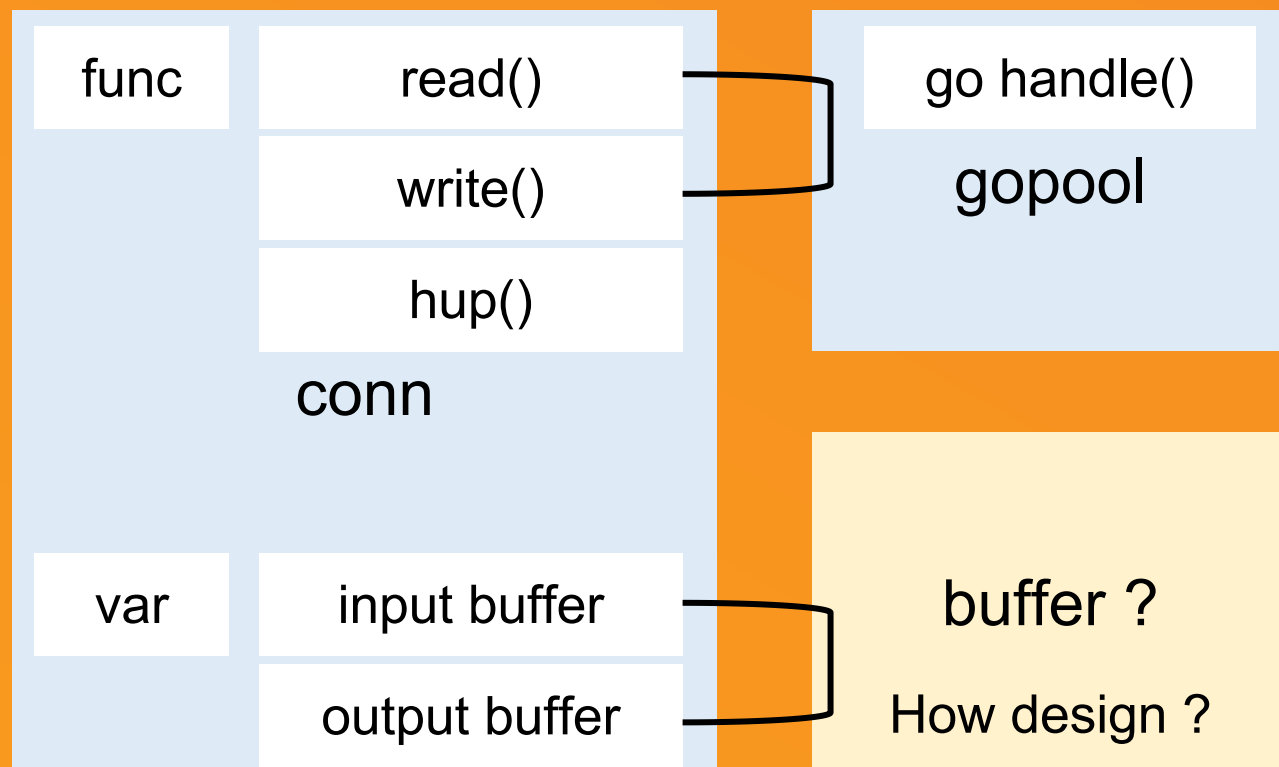
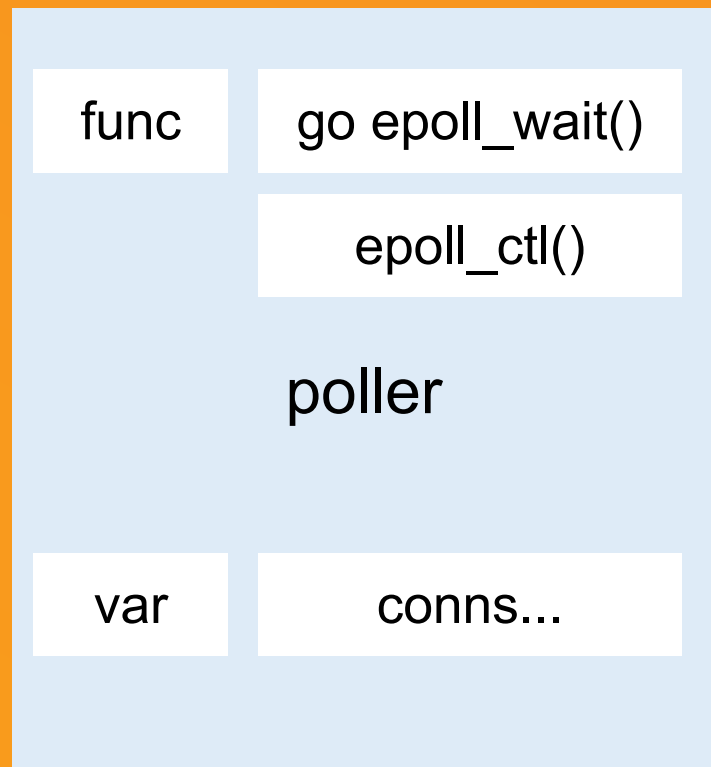
Benchmark	time/op
EpollWait, msec=0	270 ns/op
EpollWait, msec=-1	328 ns/op
Delta	-17.68%

优化方向

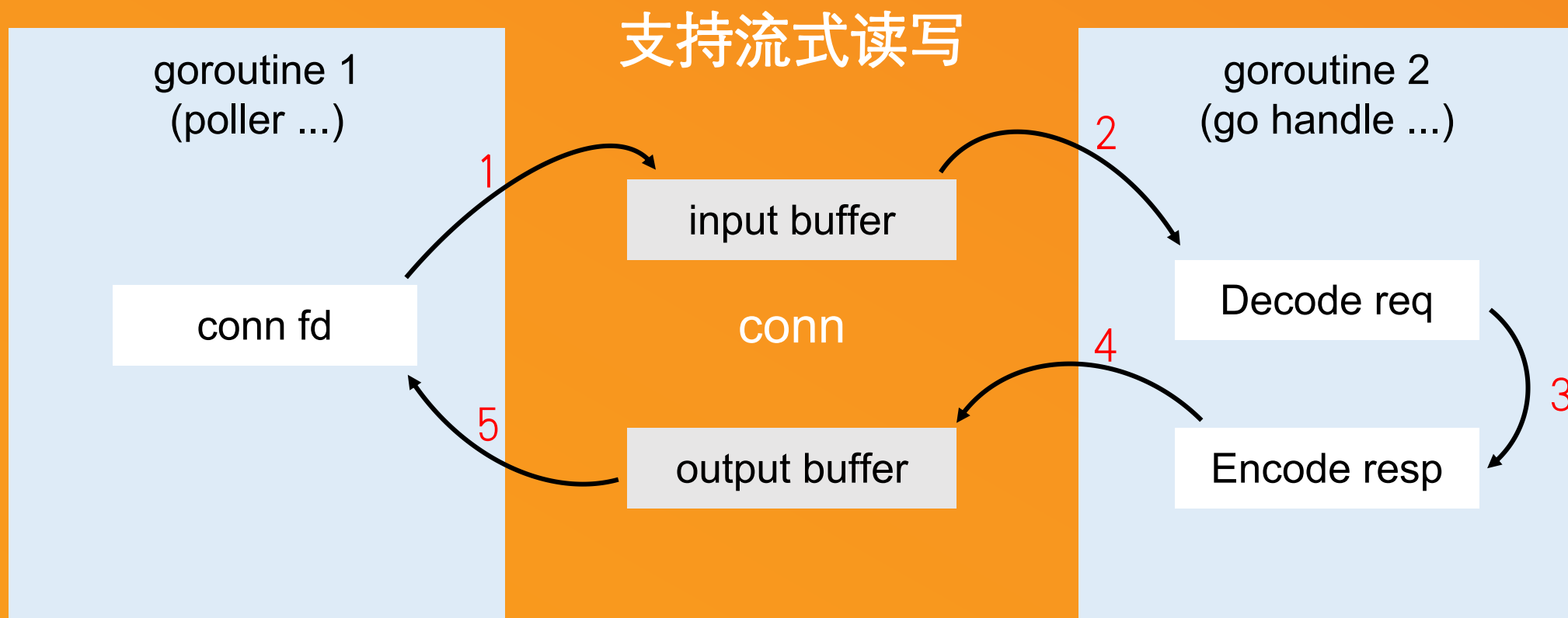
优化调度效率(poller)

优化 Buffer 设计(zerocopy)

优化 Buffer 设计



优化 Buffer 设计 - 需求分析

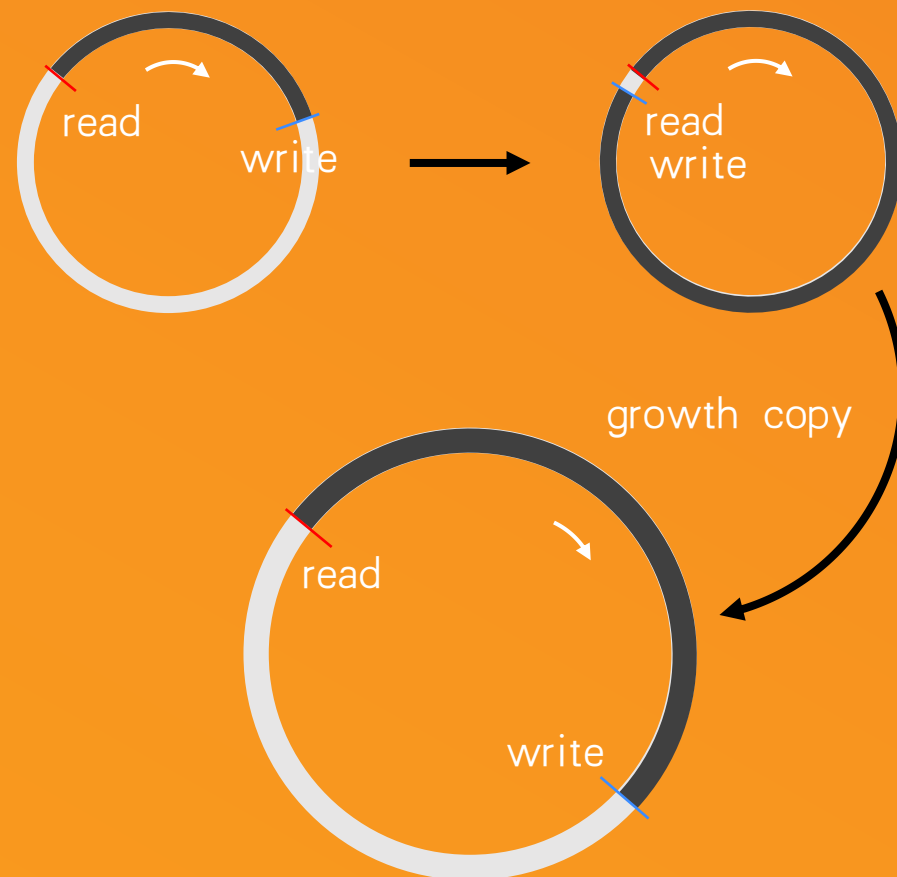


优化 Buffer 设计 – RingBuffer 分析

buffer(full) need growth

growth need copy

copy will data race

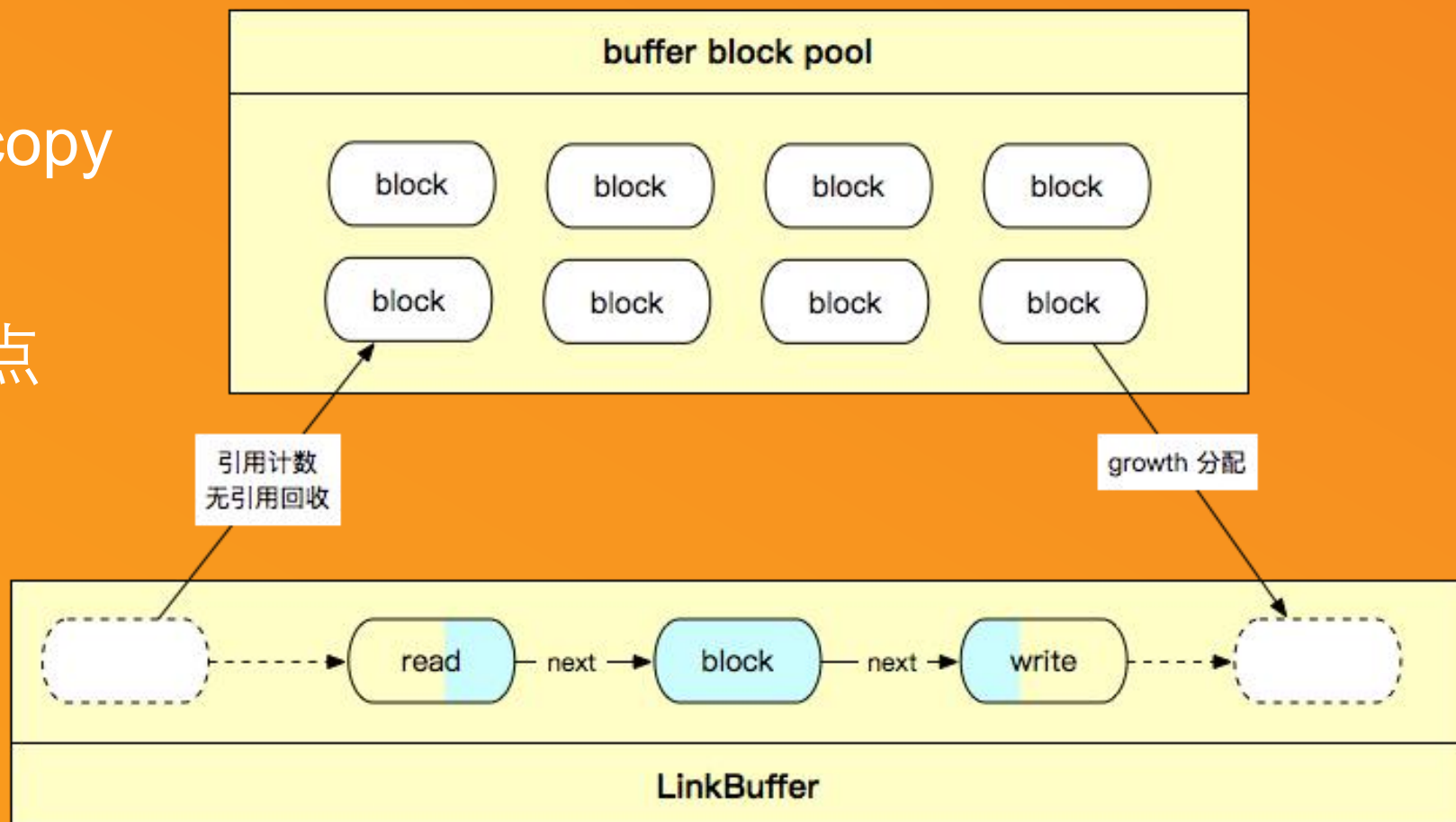


优化 Buffer 设计 - LinkBuffer 设计

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)

writew/readv 实现



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



```
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

设计实现

01

性能亮点

02

高级特性

03

展望未来

04

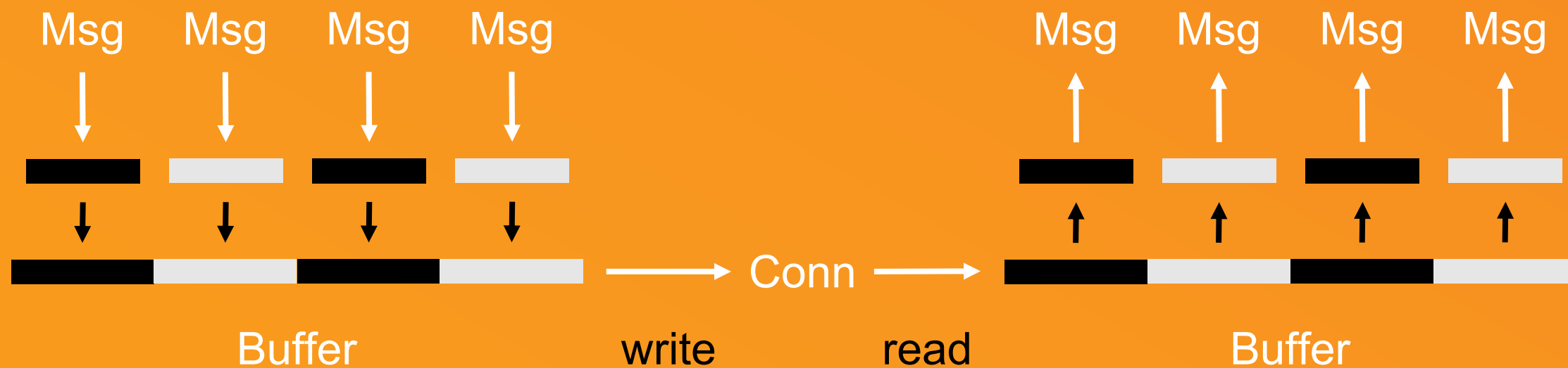
1. 单连接多路复用(ZeroCopy)

2. TCP ZeroCopy

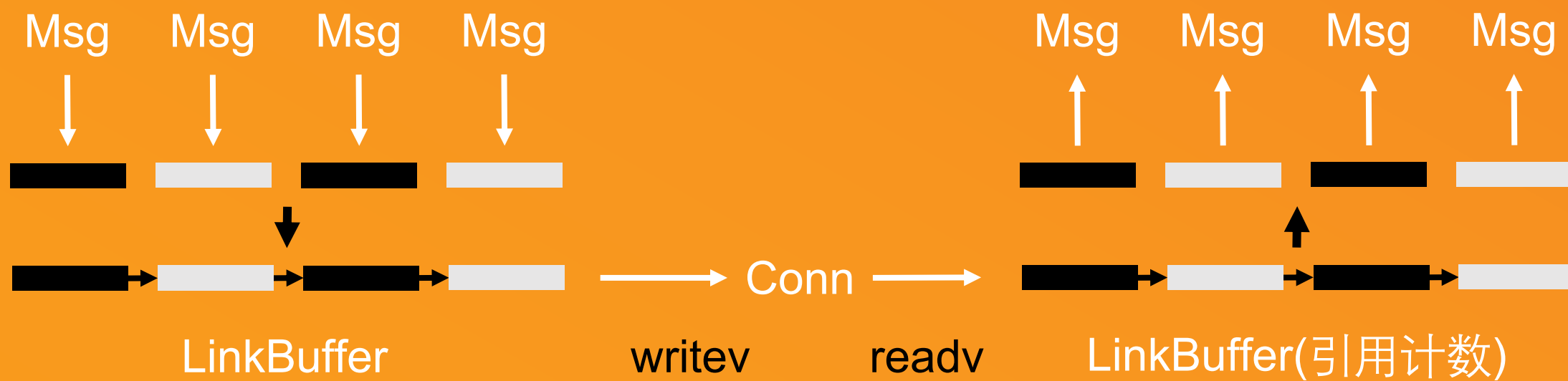
3. Multisyscall

4. io_uring

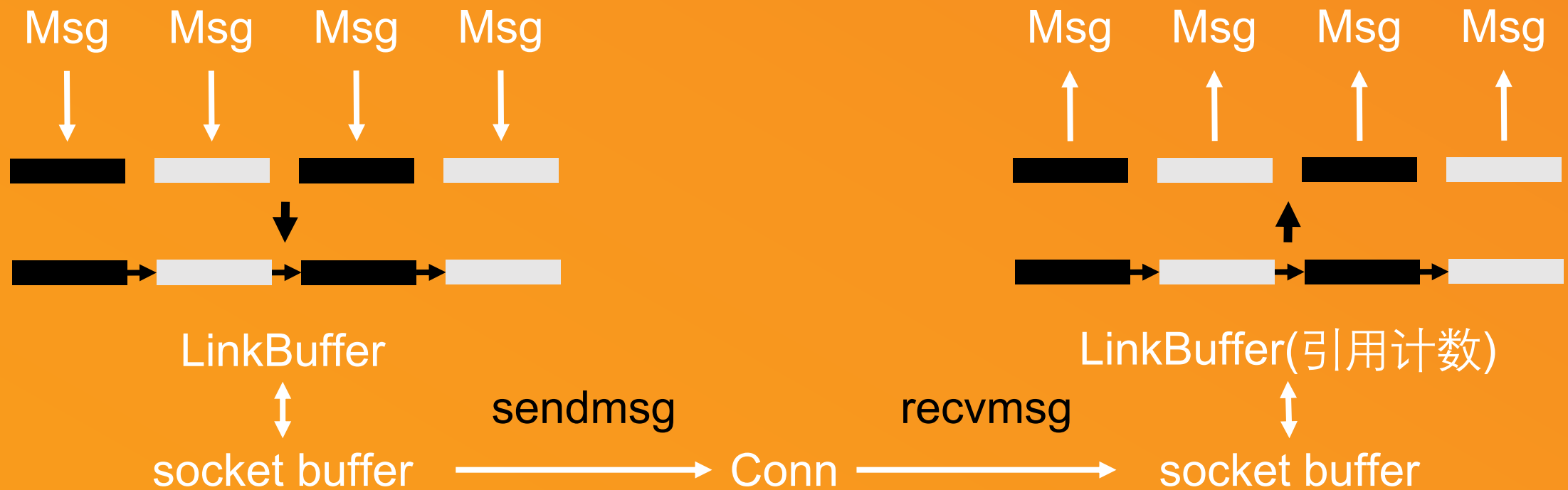
单连接多路复用 - 组包/拆包



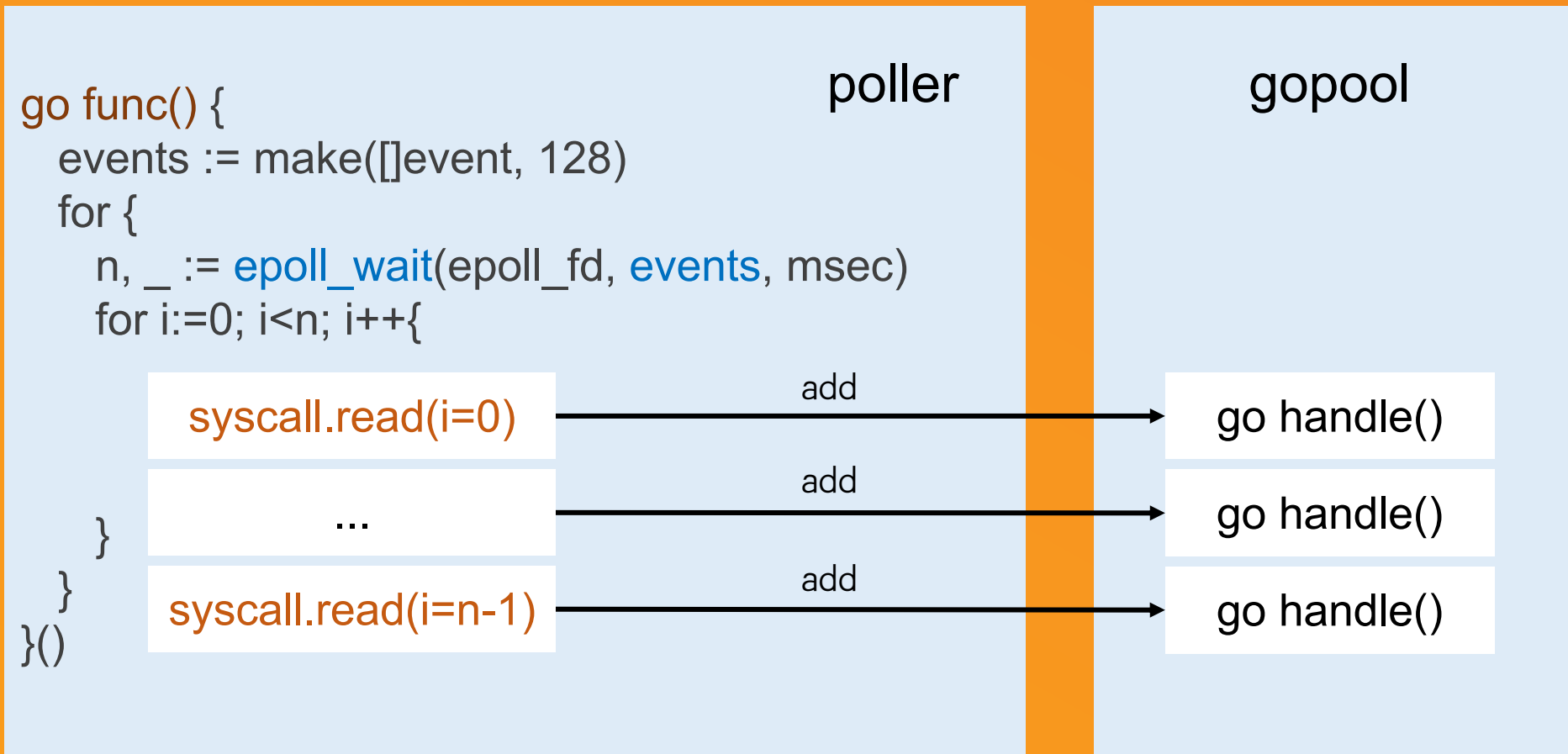
单连接多路复用 - ZeroCopy LinkBuffer



TCP ZeroCopy



Multisyscall – 常规 poller



Multisyscall – 批量调用

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

poller

add all

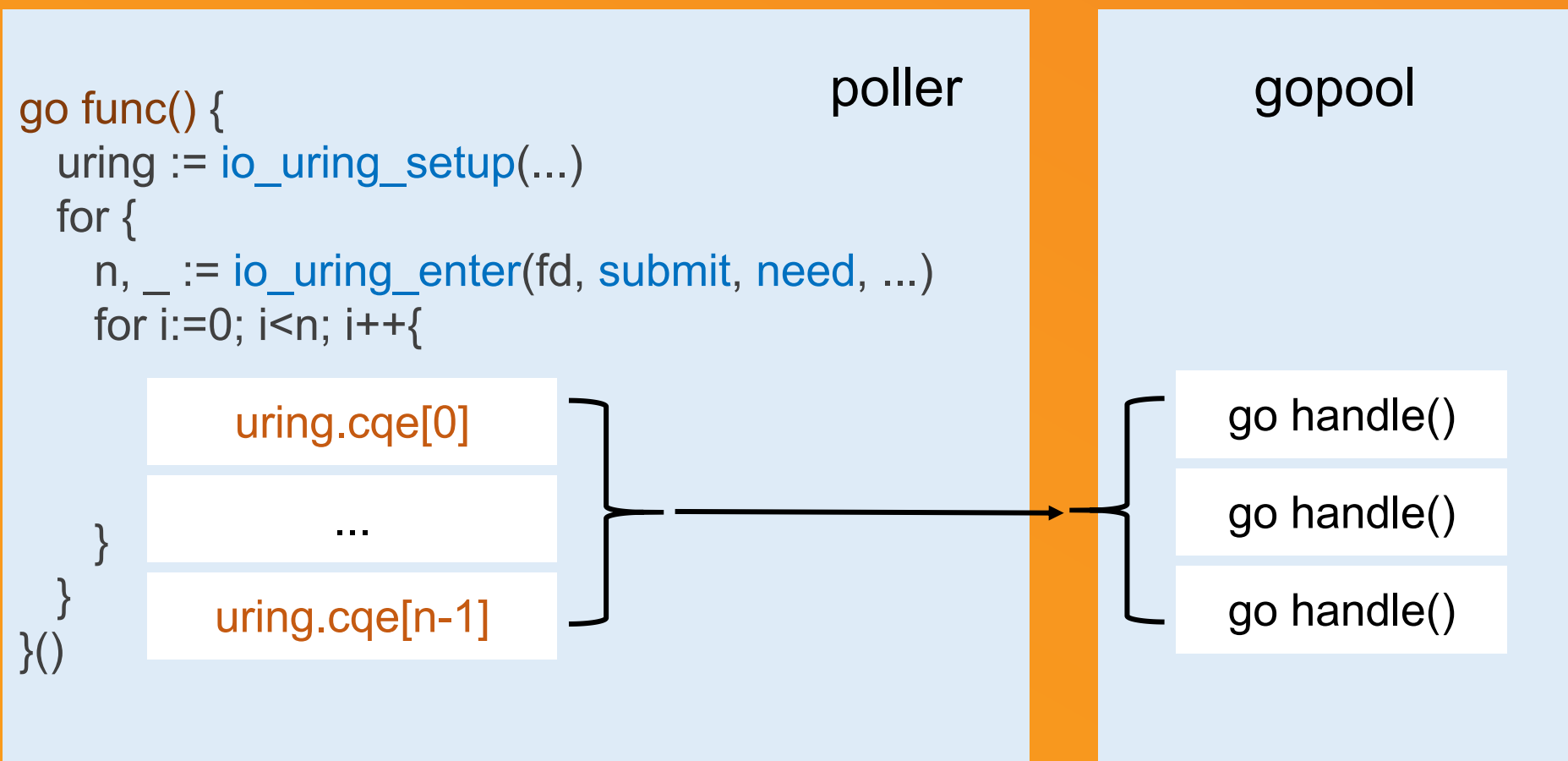
gopool

go handle()

go handle()

go handle()

io_uring - 异步调用





GopherChina 2021

设计实现

01

性能亮点

02

高级特性

03

展望未来

04

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

