网络库的性能优化与取舍

—— 以 Netpoll 为例

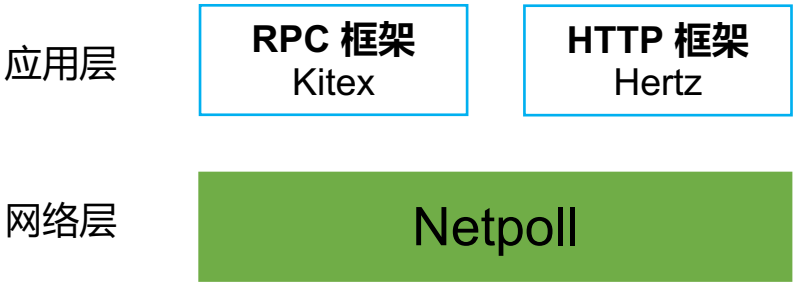
王卓炜 | 字节跳动研发工程师

ByteDance 字节跳动

介绍

CloudWeGo

https://github.com/cloudwego

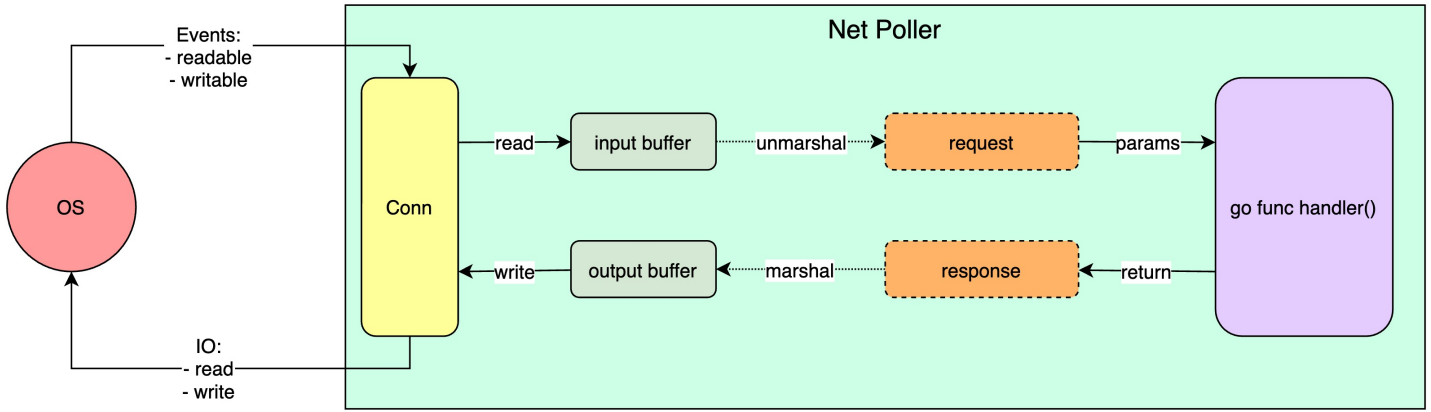


演讲大纲

1. 什么是网络库
2. 量化性能
3. 优化与取舍
4. 总结

什么是网络库 — 系统概览

一个网络库需要做哪些事情：



什么是网络库 — 伪代码实现

```
for {
    events := Poll(timeout)

    for _, event := range events {
        handler(event)
    }
}
```

本质上就是一个事件调度器

演讲大纲

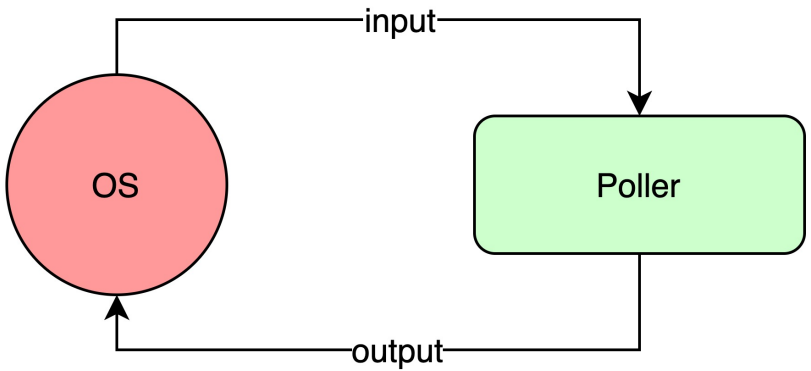
1. 什么是网络库
2. 量化性能
3. 优化与取舍
4. 总结

量化性能

If you can't measure it, you can't improve it.

—— *William Thomson*

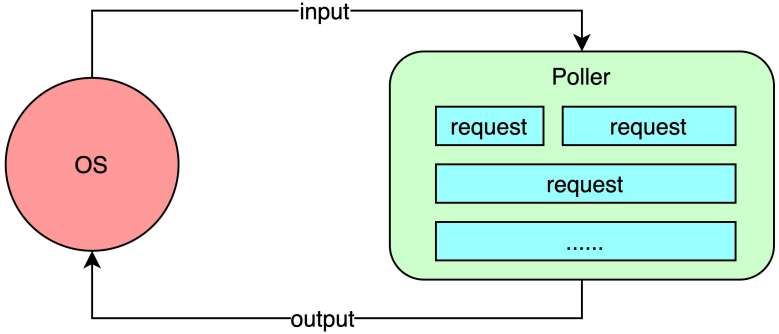
量化性能 — 确立性能指标



需求：单位时间内尽可能多地处理请求

指标：**吞吐**

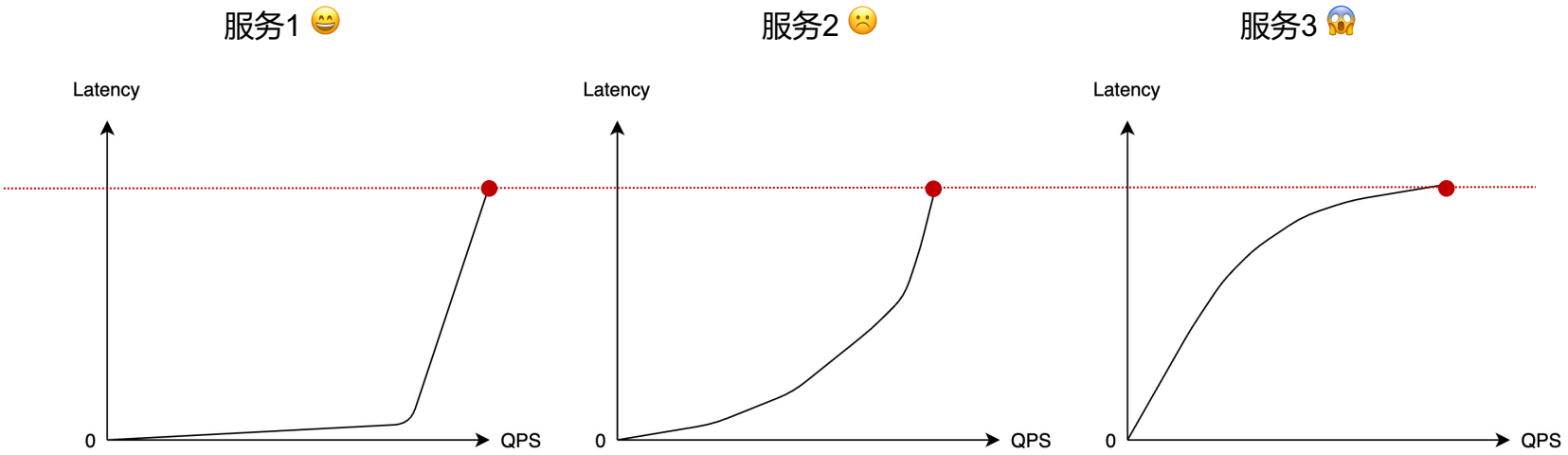
量化性能 — 确立性能指标



需求：**公平**地为每个请求分配执行时间

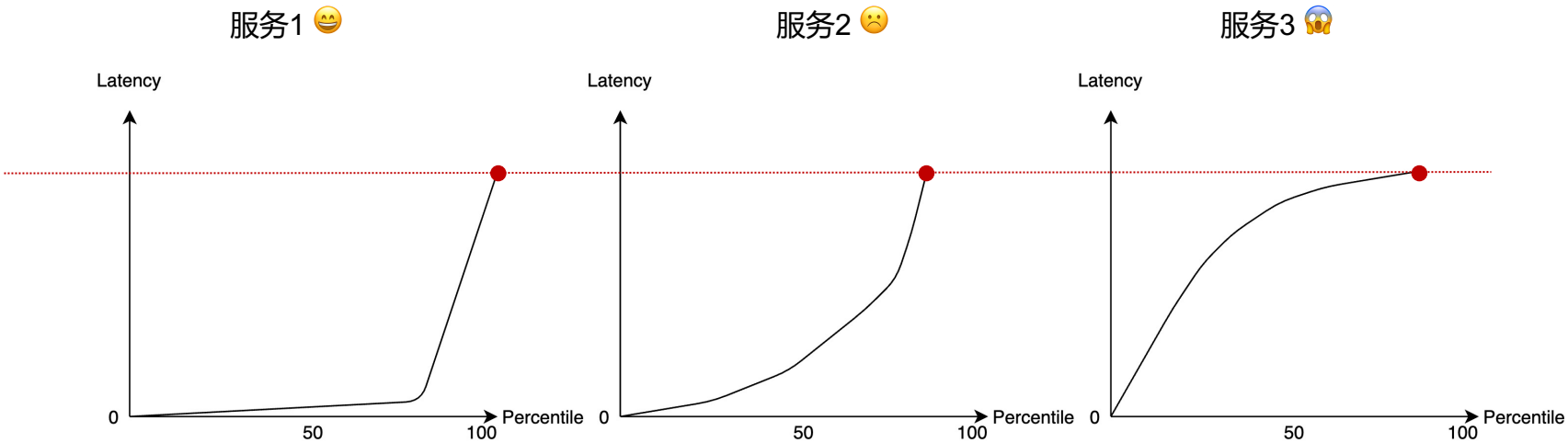
指标：**平均延迟** && **百分位延迟**

量化性能 — QPS 与延迟



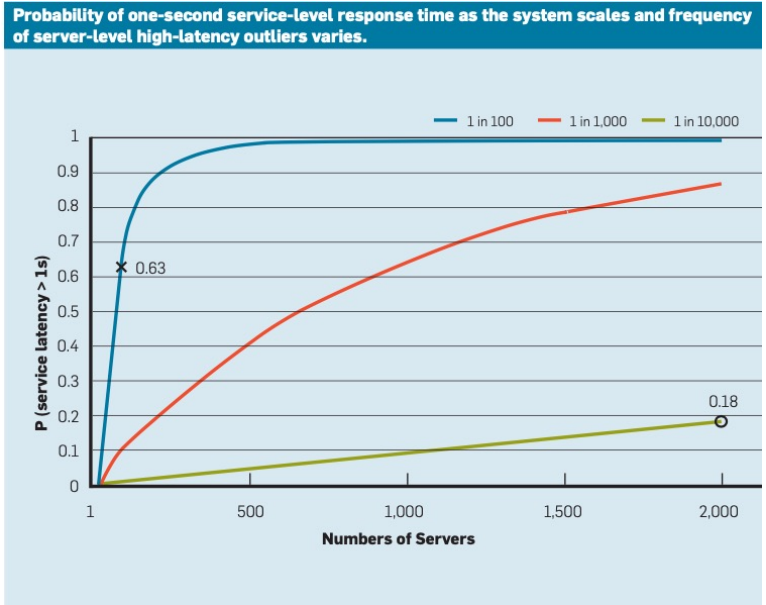
结论1：QPS 不能作为衡量性能的单一指标

量化性能 — 百分位延迟



结论2：百分位延迟面积越小性能越好

量化性能 — 微服务下的长尾延迟



https://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

演讲大纲

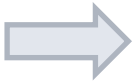
1. 什么是网络库
2. 量化性能
3. 优化与取舍
4. 总结

优化与取舍 - 择时调度

忙时积极抢占，闲时主动让出

```
for {
    events := Poll(timeout)

    for _, event := range events {
        handler(event)
    }
}
```



正面收益：减少了无效系统调用次数

```
var timeout = 0
for {
    events := EpollWait(timeout)
    if len(events) == 0 {
        timeout = -1
        runtime.Gosched()
        continue
    }
    timeout = 0
    handler(events)
}
```

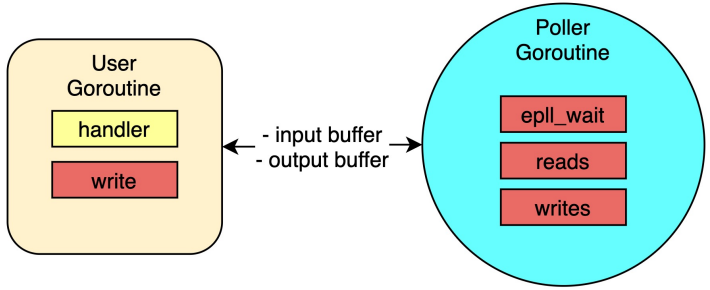
优化与取舍 - 减少协程切换

```
func handler(conn) {
    req := Decode(buf)
    resp := Process(req)
    write(resp)
}

for {
    events := EpollWait(timeout)

    for _, event := range events {
        read(event.fd)
        write(event.fd)
        go handler(event.conn)
    }
}
```

一次执行权尽可能多地处理任务



正面收益：

- 减少了整体调度次数

负面效应：

- Poller 压力增大导致请求间隔离性降低（可通过创建更多 Poller 解决）

优化与取舍 - 复用协程

```
func handler() {
    // if more than 2KB
    runtime.morestack()
}
```

问题：

在实际线上服务中，每个业务 goroutine 往往都存在固定的栈扩张行为

传统 Goroutine 池实现

```
type worker struct {
    task chan func()
}

func (w *worker) Submit(task func()) {
   w.task <- task
}

func (w *worker) run() {
  go func() {
      for f := range w.task {
         f()
      }
  }()
}
```

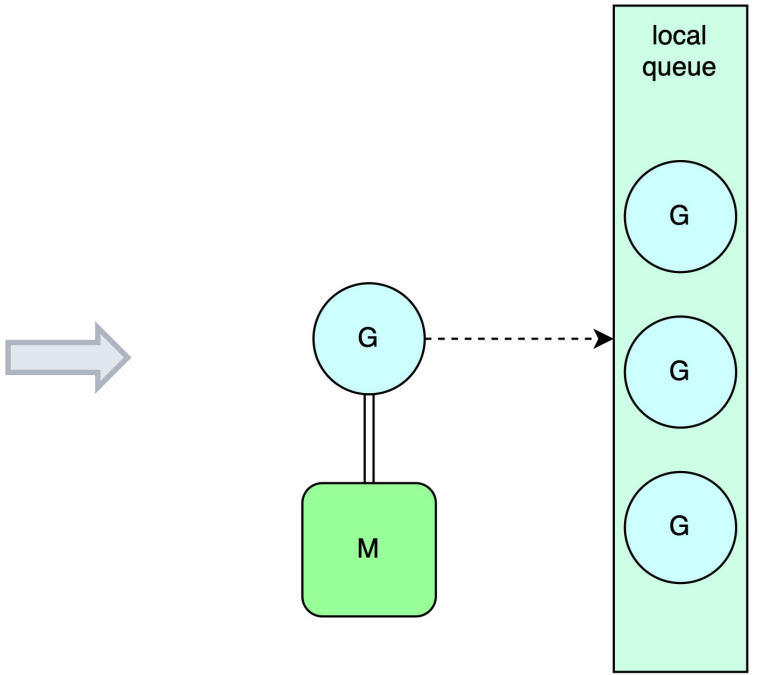
负面效应：

- 从 Submit 任务到真正被调度执行中间有不确定性的调度延迟
- 后台有持久 Goroutine 存活等待

优化与取舍 - 复用协程

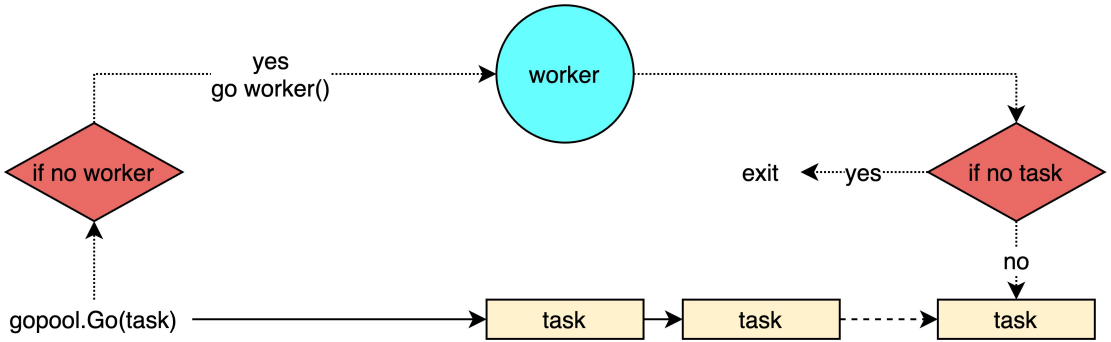
Poller 创建 Goroutine 的真实情况：

```
for {
    events := EpollWait(timeout)
    for _, event := range events {
        go handler(events)
    }
}
```



优化与取舍 - 复用协程

[Gopool 实现]



正面收益：

- 最小化减少了额外的调度延迟

负面效应：

- 当 CPU 核心过多时，全局锁开销增大

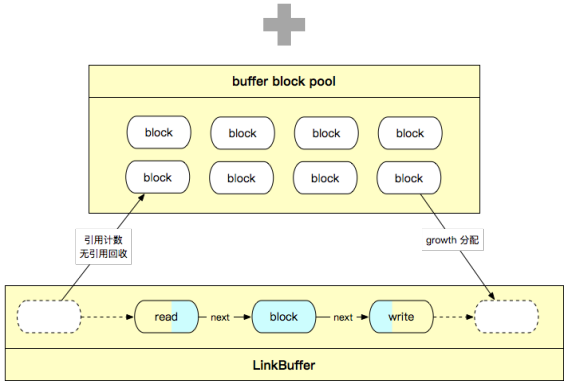
- 最好情况下，另一个 goroutine 可以无调度地执行新的任务
- 最差情况下，创建一个新 goroutine 执行任务

优化与取舍 - 复用内存

NoCopy API + LinkBuffer

```
// net.Conn
Read(p []byte) (n int, err error)

// netpoll.Connection
Next(n int) (p []byte, err error)
```



跨连接复用内存

正面收益：

- 减少内存申请
- 减少内存拷贝

负面效应：

- 应用层需要从 net.Conn 改造到专有的 NoCopy API

优化与取舍 - 降低系统调用开销

使用 RawSyscall

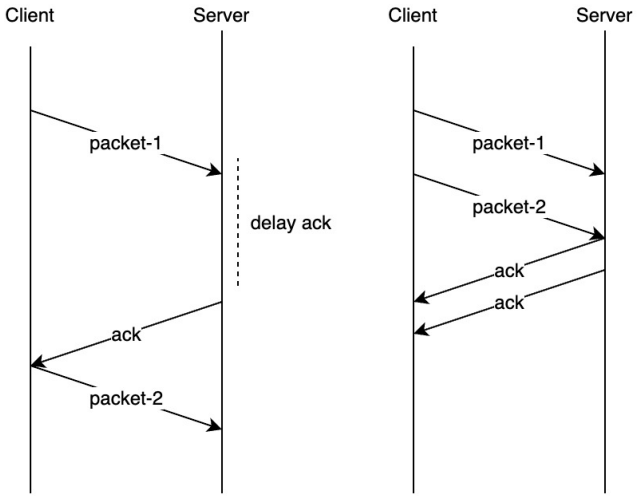
- Syscall = enter_runtime + RawSyscall + exit_runtime
- 非阻塞调用使用 RawSyscall

正面收益： 减少系统调用的额外 runtime 开销

```
func EpollWait(epfd int, events []epollevent, msec int) (n int, err error) {
   if msec == 0 {
      r0, _, err = syscall.RawSyscall6(syscall.SYS_EPOLL_WAIT, ...)
   } else {
      r0, _, err = syscall.Syscall6(syscall.SYS_EPOLL_WAIT, ...)
   }
}
```

优化与取舍 - 开启 TCP_NODELAY

Nagle's Alg vs TCP_NODELAY



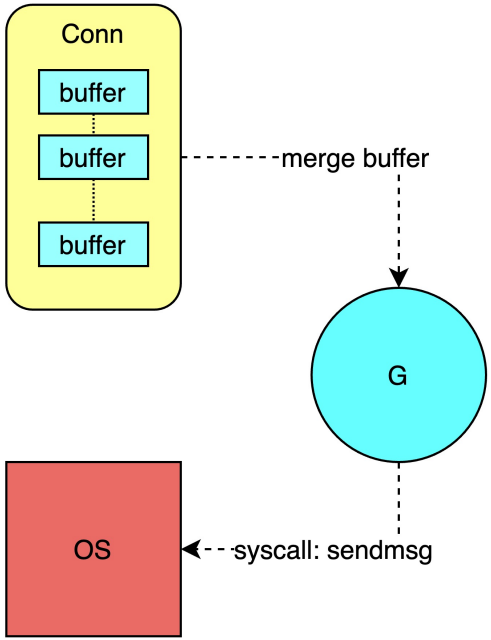
正面收益：

- 小包无需等待 ACK 直接发送，降低了延迟

负面效应：

- 小包流量被放大，降低了网络利用率

优化与取舍 - 合并写入



正面收益：

- 减少了系统调用次数，提升了吞吐

负面效应：

- 从写入 Buffer 到被另一个 G 发送数据产生了额外的调度延迟

问题：为什么去掉了 TCP 的合并包却自己实现合并？

演讲大纲

1. 什么是网络库
2. 量化性能
3. 优化与取舍
4. 总结

总结 – 延迟与吞吐的关系

同时优化吞吐与延迟

- 减少计算开销
- 减少无用的调度

吞吐换延迟

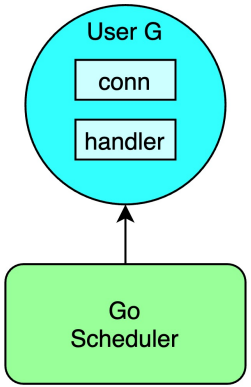
- 增加调度以实现换取公平的执行权分配

延迟换吞吐

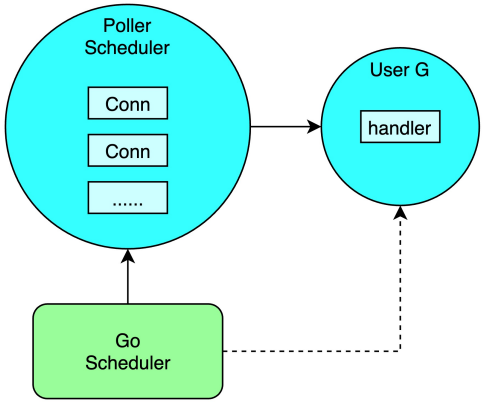
- 延后等待可合并事件再批量执行

总结 – 为什么要做 **Netpoll**

Go Net



Netpoll



优势：

1. 能够减少大量 Goroutine
2. 能够主动调度减少 P99 延迟
3. 能够主动管理内存

总结 – 欢迎关注 CloudWeGo

扫码查看 CloudWeGo

扫码访问项目官网

Q&A

THANKS

.

ByteDance字节跳动