Golang
在接入层长连接服务中的实践

黄欣
基础平台－架构部

滴滴
滴滴一下 美好出行

目录

- 背景
- 架构
- 心得

目录

- 背景
- 架构
- 心得

背景—why 长连接？

- 业务场景
  - 大量实时计算
    - 司机乘客撮合
    - 实时计价
  - 高频度的数据交互
    - 坐标数据
    - 计价数据
  - App和服务端双向可达
    - 上行（抢单）
    - 下行（派单）

背景—why golang？

- 开发效率
- 异步模型，同步原语
  - C：代码上各种回调、思维中保持冷静
  - Go：代码上同步，思维自然
- 性能够用，工具齐全
  - 100w？10w？~~
  - Memprof、cpuprof~
- 社区活跃，发展迅猛

背景—使用现状

- 每天服务于千万级别的司机，数亿的用户
- 实时在线百万级别
- 每天平均70亿次的推送量

背景—总结

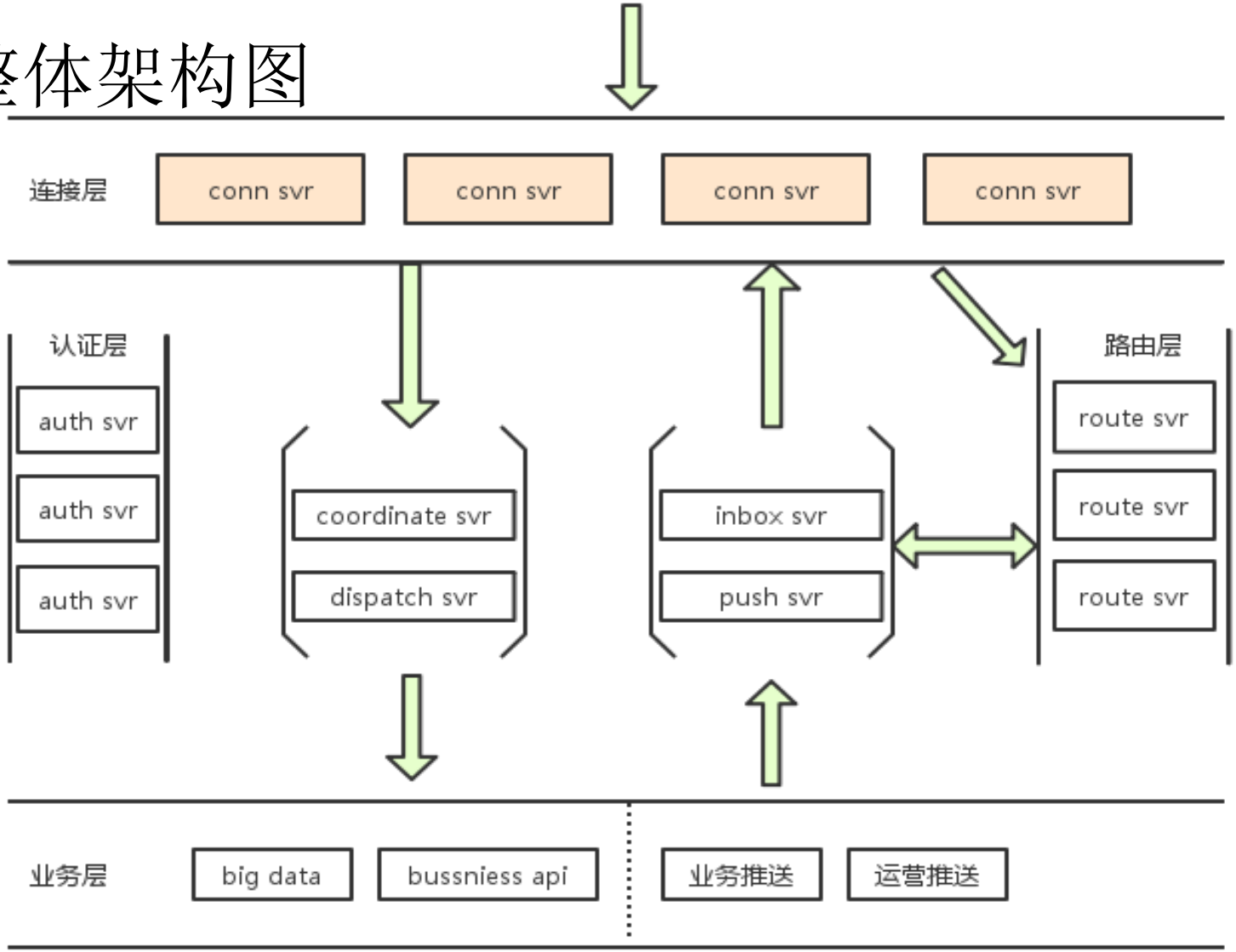
- 业务上核心依赖
- Golang成功的使用案例

目录

- ~~背景~~
- 架构
- 心得

架构

- 整体架构图


连接层
conn svr
conn svr
conn svr
conn svr
认证层
auth svr
auth svr
auth svr
coordinate svr
dispatch svr
inbox svr
push svr
路由层
route svr
route svr
route svr
业务层
big data
bussniess api
业务推送
运营推送

架构—接口设计

- 原则
  - 扩展性
  - 稳定性（最好不用升级）
- 解决方法
  - Protobuf（golang）
  - 接口设计分层
    - 框架层：模块间通信协议（类似tcp/udp）
    - 业务层：bytes（类似应用层）留给业务自己定义就好了

架构—性能

- conn svr

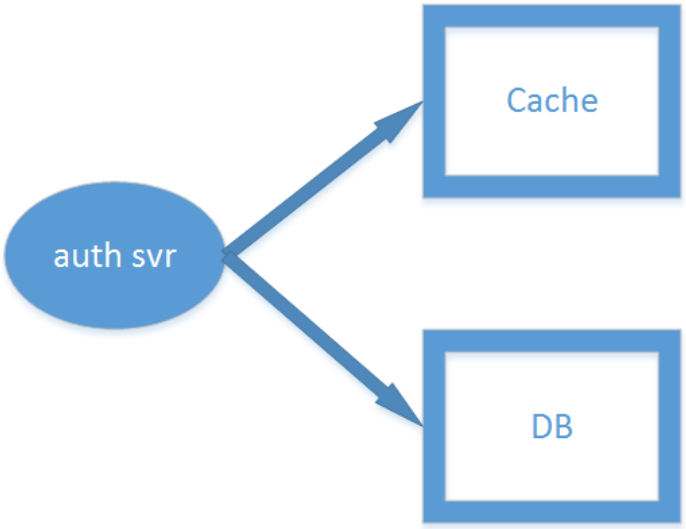
连接数	qps	内存	cpu（平均）	gc（STW）
30000	3w上行 3w下行	3～4G	300%左右	8～40ms
60000	6w上行 4w下行	5～6G	500%左右	8～40ms
90000	9w上行 5w下行	9～10G	850%左右	8～40ms

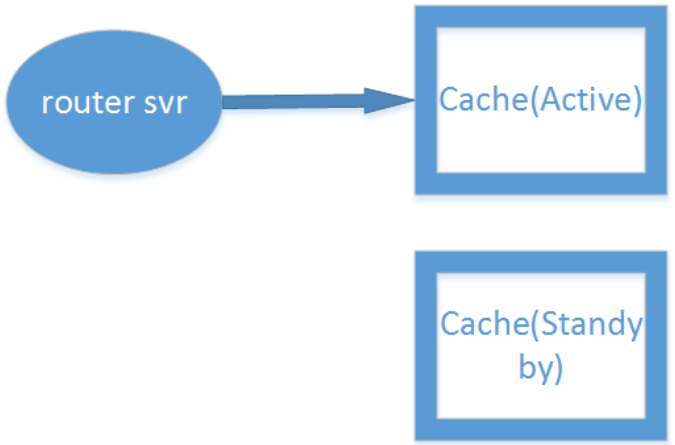
架构—集群扩展

- Proxy本身无限扩容（无状态）
- 依赖的存储可无限扩容（状态交给存储）
  - Redis集群：codis集群方案
  - Mysql集群：中间件方案

架构—灾备

- 这里的灾备主要指的是依赖的存储降级方案，涉及到存储的主要两个模块
  - Auth svr：cache（redis） + db（mysql）
  - Route svr：cache + cache（standy）

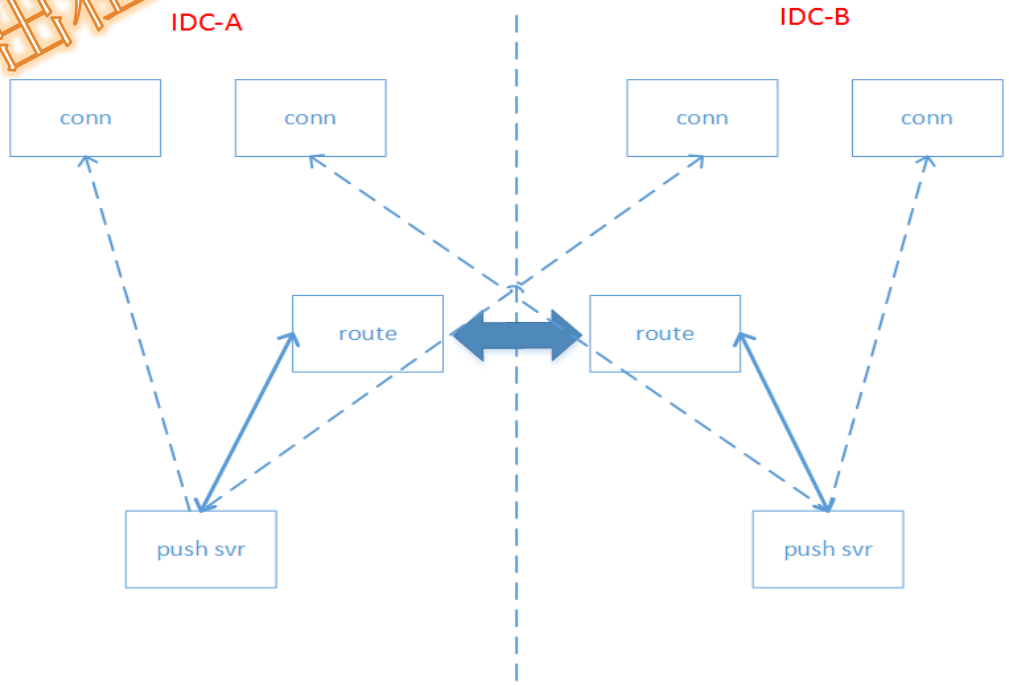




架构—异地双活

- 要求
  - 正常情况下：
    - 任何一个机房可推送到所有机房app
  - 异常情况下：
    - 本机房内推送可达
- 架构图如下

（核心解决路由共享问题）



架构—总结

- 异步通信接口
- 协议包业务态隔离
- 简单无状态
- 有状态的服务（涉及到存储）做到可降级
- 核心业务有自愈逻辑

简单实用，避免过度设计

目录

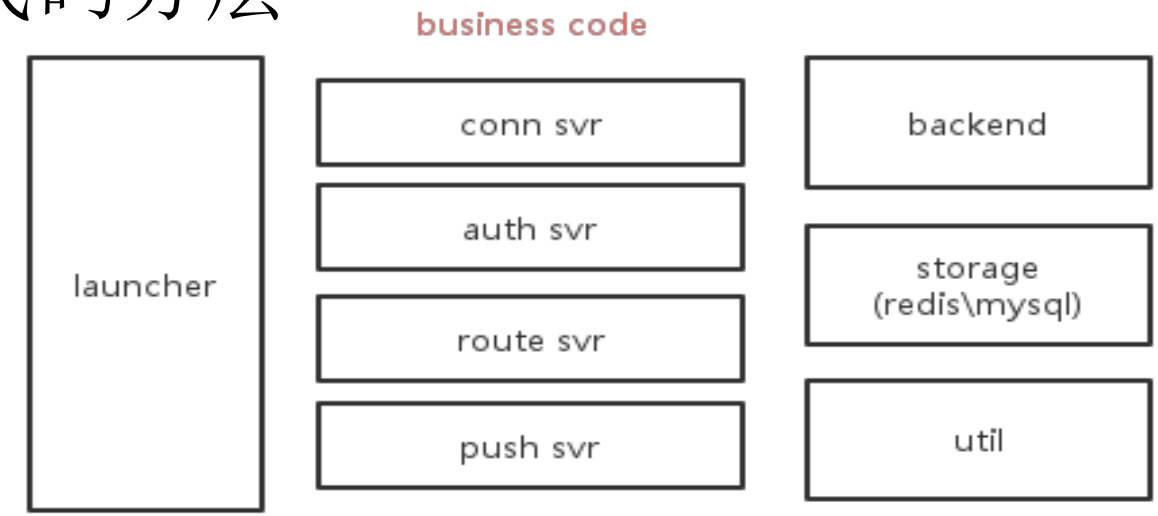
- ~~背景~~
- ~~架构~~
- 心得
  - Coding
  - profiling

心得—coding

- 代码分层
  - 提高开发效率
  - 代码合理复用，各司其职
- 实现
  - 过程编程
  - 对象编程

心得—coding-分层

- 代码分层



launcher: 接收连接，接收请求，go出去，等待业务层反回结果，并write back

business：业务代码，拿到请求自行处理，完事之后return到laucher

backend：和长连接系统中的其他模块异步通信模块

storage：和存储交互模块，提供统一的封装

util：其他通用模块（时间轮、对象池、wait封装等）

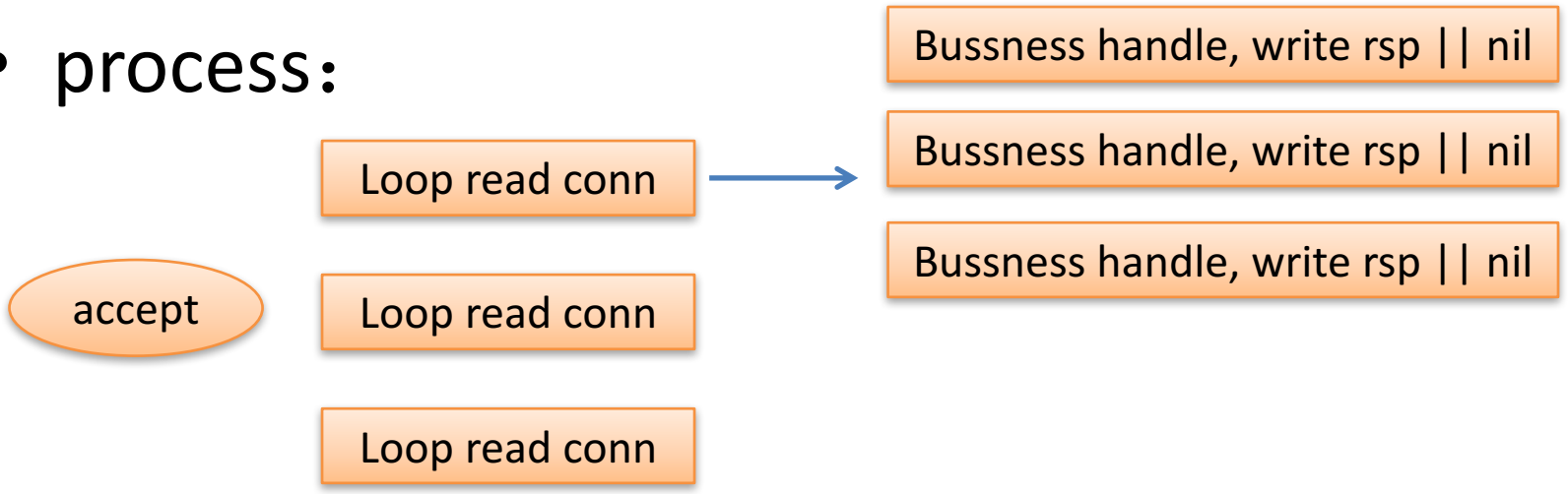
心得—coding-实现

- 过程编程
  - route svr
  - auth svr
  - push svr
  - dispatch svr
  - Inbox svr

特点：以请求为单位，每个请求一个goroutine处理，不存在任何状态

心得—coding-实现

- process：



So easy, So efficient

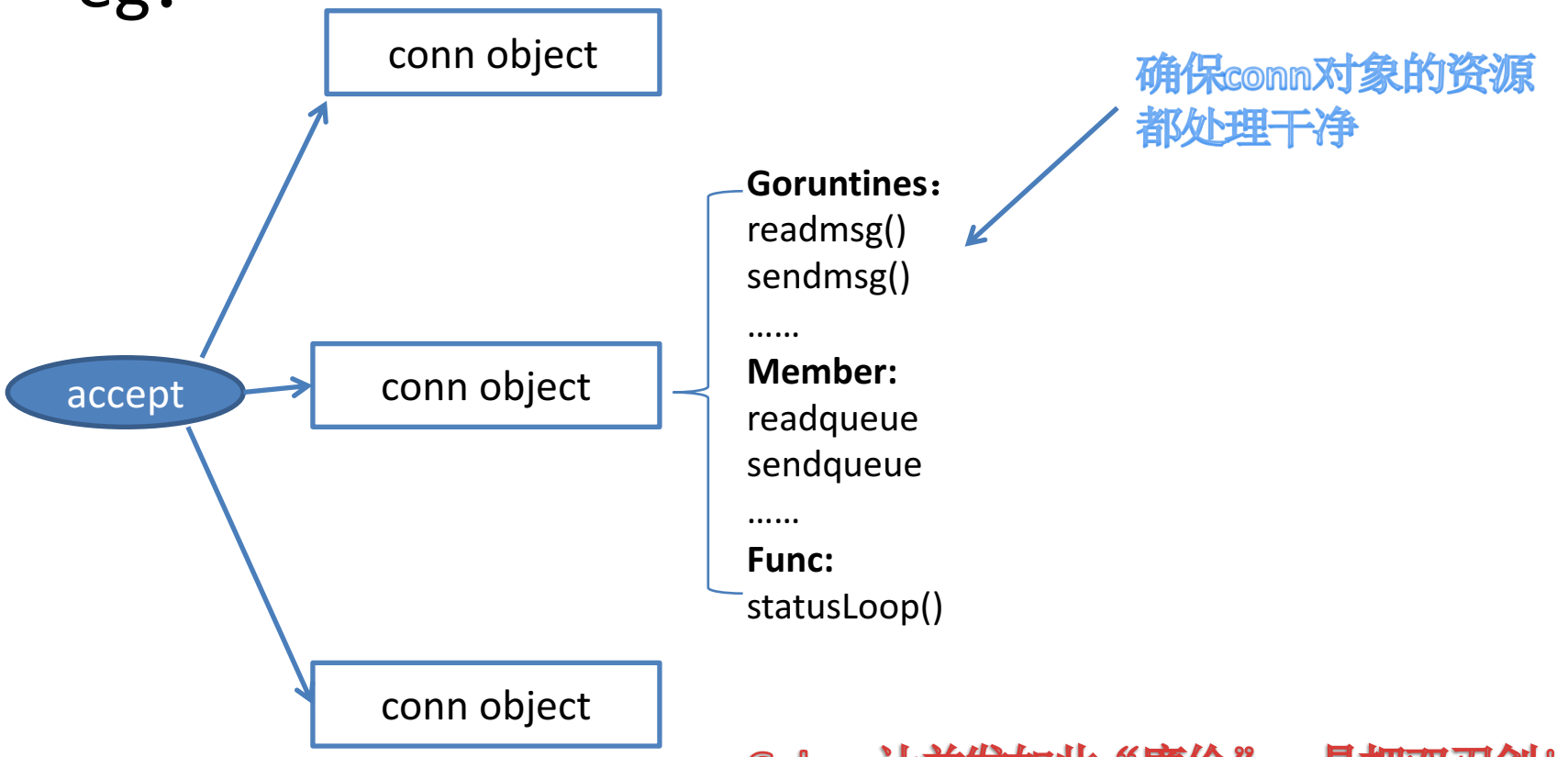
心得—coding—实现

- what’s diff?
  - conn svr
    - 常驻内存，内存中有个大连接对象map（资源问题）
    - 请求都是基于连接的(如果模块间存在资源的互相引用，当资源变更的情况下，容易发生panic)（竟态问题）
- 对象编程
  - 封装：conn资源（包括goruntine）作为结构体封装起来，保证所有资源销毁干净
  - 解耦：保证其他模块不直接使用对象中资源
  - 同步：竞态需要锁

特点：有状态，存在大量的公共资源并发访问

心得—coding—实现

- eg：



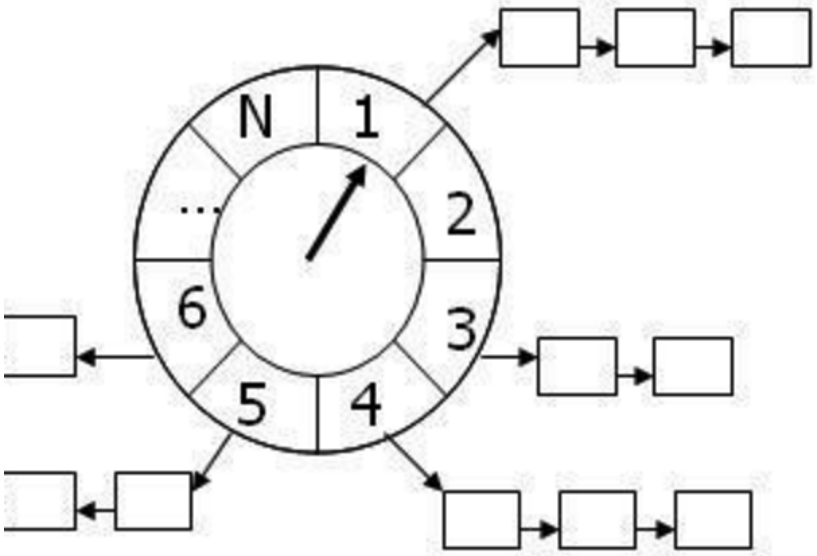
心得—profiling

- Timer优化
- Channel使用优化

心得—timer优化

- 为什么需要优化？
  - 万级别的连接
  - 每个连接上大量的定时任务（心跳检测，注册检测，认证检测）

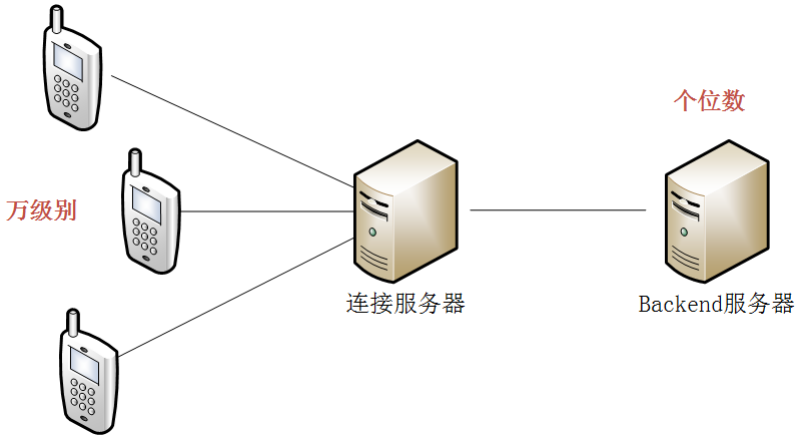
  实际情况：当10w左右连接，什么数据不收发，只有定时器检测心跳超时，cpu能耗掉一个core
- 怎么优化？
  - 特点：
    - 秒级别定时任务
    - 范围最多60s
  - 方案：
    - 时间轮



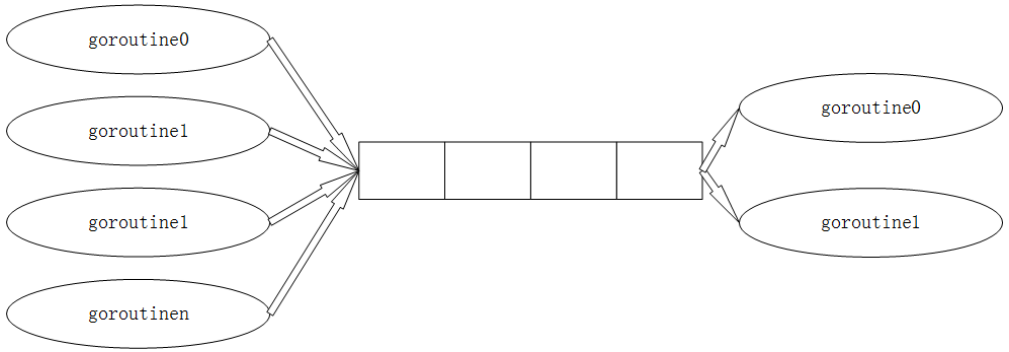
- 实现
  - Channel

心得—channel使用优化

- 业务场景：



- 程序层面：



- 问题？
  - Channel在这种高并发场景下，锁消耗巨大，性能有瓶颈

心得—channel优化

- 测试
  - Golang1.5
  - GOMAXPROCS = A
  - Send goruntine = B
  - Read goruntine = C
  - Count（message） = 100000000

单线程模式效率性能最好，消耗最少

多线程并发下，大量上下文切换，sy占用高，且性能低下

A	B	C	cs	Cpu (us/sy)	Cost（s）
1	10000	10	1658	5/0	33
1	50000	10	1658	5/0	35
2	50000	10	54718	9/1	83
10	50000	10	672448	10/20	86
15	50000	10	672448	8/24	81

心得—channel优化

- 方案一：无锁队列（cas）

```go
//node
type Node struct {
    val  interface{}
    next unsafe.Pointer
}

// new fqueue
func NewFQueue() *FQueue {
    queue := new(FQueue)
    queue.head = unsafe.Pointer(new(Node))
    queue.tail = queue.head
    queue.size = 0
    return queue
}
```

```go
// en fqueue
func (self *FQueue) EnFQueue(val interface{}) {
    newValue := unsafe.Pointer(&Node{val: val, next: nil})
    var tail, next unsafe.Pointer
    for {
        tail = self.tail
        next = ((*Node)(tail)).next
        if next != nil {
            if !atomic.CompareAndSwapPointer(&(self.tail), tail, next){
                runtime.Gosched()
            }
        } else if atomic.CompareAndSwapPointer(&((*Node)(tail).next), nil, newValue) {
            atomic.AddInt64(&self.size, 1)
            break
        }else{
            runtime.Gosched()
        }
    }
}
```

```go
//de fqueue
func (self *FQueue) DeFQueue() (val interface{}, success bool) {
    var head, tail, next unsafe.Pointer
    for {
        head = self.head
        tail = self.tail
        next = ((*Node)(head)).next
        if head == tail {
            if next == nil {
                return nil, false
            } else {
                if !atomic.CompareAndSwapPointer(&(self.tail), tail, next){
                    runtime.Gosched()
                }
            }
        } else {
            val = ((*Node)(next)).val
            if atomic.CompareAndSwapPointer(&(self.head), head, next) {
                atomic.AddInt64(&self.size, -1)
                return val, true
            }else{
                runtime.Gosched()
            }
        }
    }
    return
}
```

心得—channel优化

- 方案一：fqueue没有触发机制
  - 轮训（no）
  - 做一个通知机制（yes）

```
func (b *Backend) Send(req *Request) (err error) {
    if b.GetQueueSize() > b.maxbuffpack {
        return errors.New("over max buffer size")
    }
    b.sendQueue.EnQueue(req)
    atomic.AddUint64(&b.reqnum, 1)

    select {
    case b.sendQueueNoticeChan <- 1:
    default:
    }
    return
}
```

```
func (c *Conn) write() {
    sendQueue := c.pool.backend.sendQueue
    sendQueueNoticeChan := c.pool.backend.sendQueueNoticeChan
    var req *Request

    for {
        val, ok := sendQueue.DeQueue()
        if ok {
            req = val.(*Request)
            goto sendmsg
        }

        select {
        case <-c.closeCh:
            log.Errorf("wexit||conn=%s:%s||recv exit msg", c.remtoe, c.local)
            goto exit
        case <-sendQueueNoticeChan:
            continue
        }
```

心得—channel优化

- 效果
  - GOMAXPROCS = A
  - Send goruntine = B
  - Read goruntine = C
  - Count（message） = 100000000

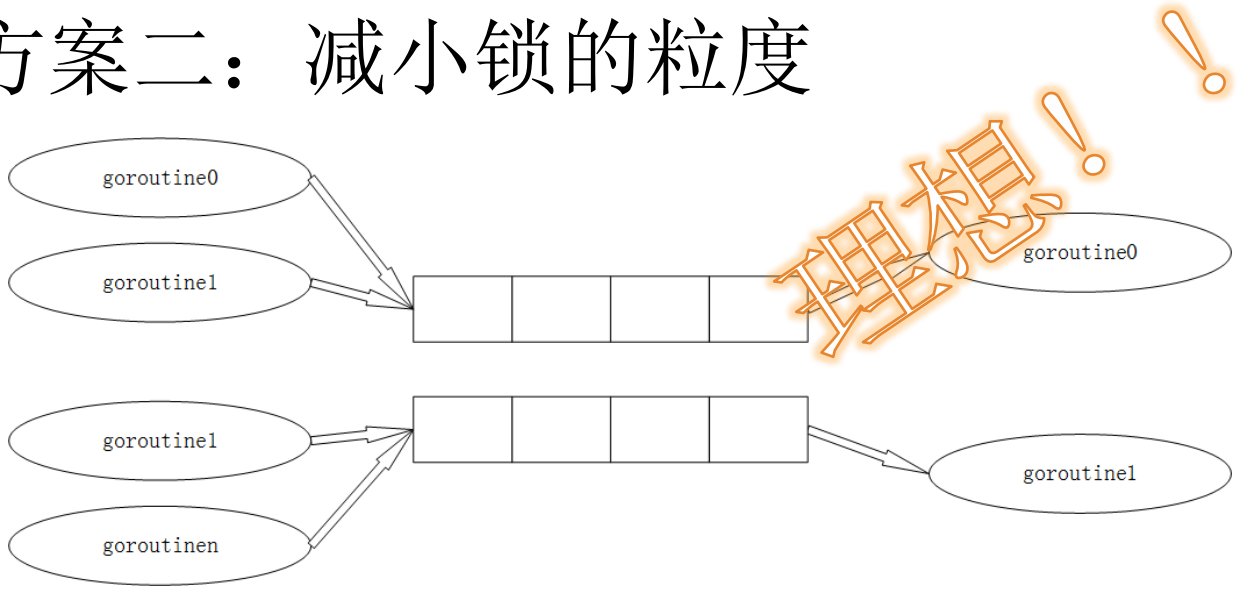
A	B	C	cs	Cpu(channel) (us/sy)	Cos(channel)（s）	Cpu(fqueue) (us/sy)	Cos(fqueue)（s）
10	50000	10	672448	10/20	86	22/8	85
15	50000	10	672448	8/24	81	23/17	91

在这种极端压测情况下（冲突及其严重的场景下）

- Cpu消耗从sy转移到us，因为有大量的cas fail，导致大量重试
- 整体耗时也没有明显减少

心得—channel优化

- 方案二：减小锁的粒度



队列从一条变成N条，缩小竞争范围

A	B	C	cs	Cpu (us/sy)	Cost（s）	Channel个数
1	10000	10	1658	5/0	33	1
10	50000	10	672448	10/20	86	1
10	**50000**	**10**	**71698**	**6/0**	**28**	**3**

THANK YOU

滴滴

滴滴一下 美好出行