分布式消息推送

rfyiamcool

xiaorui.cc

github.com/rfyiamcool

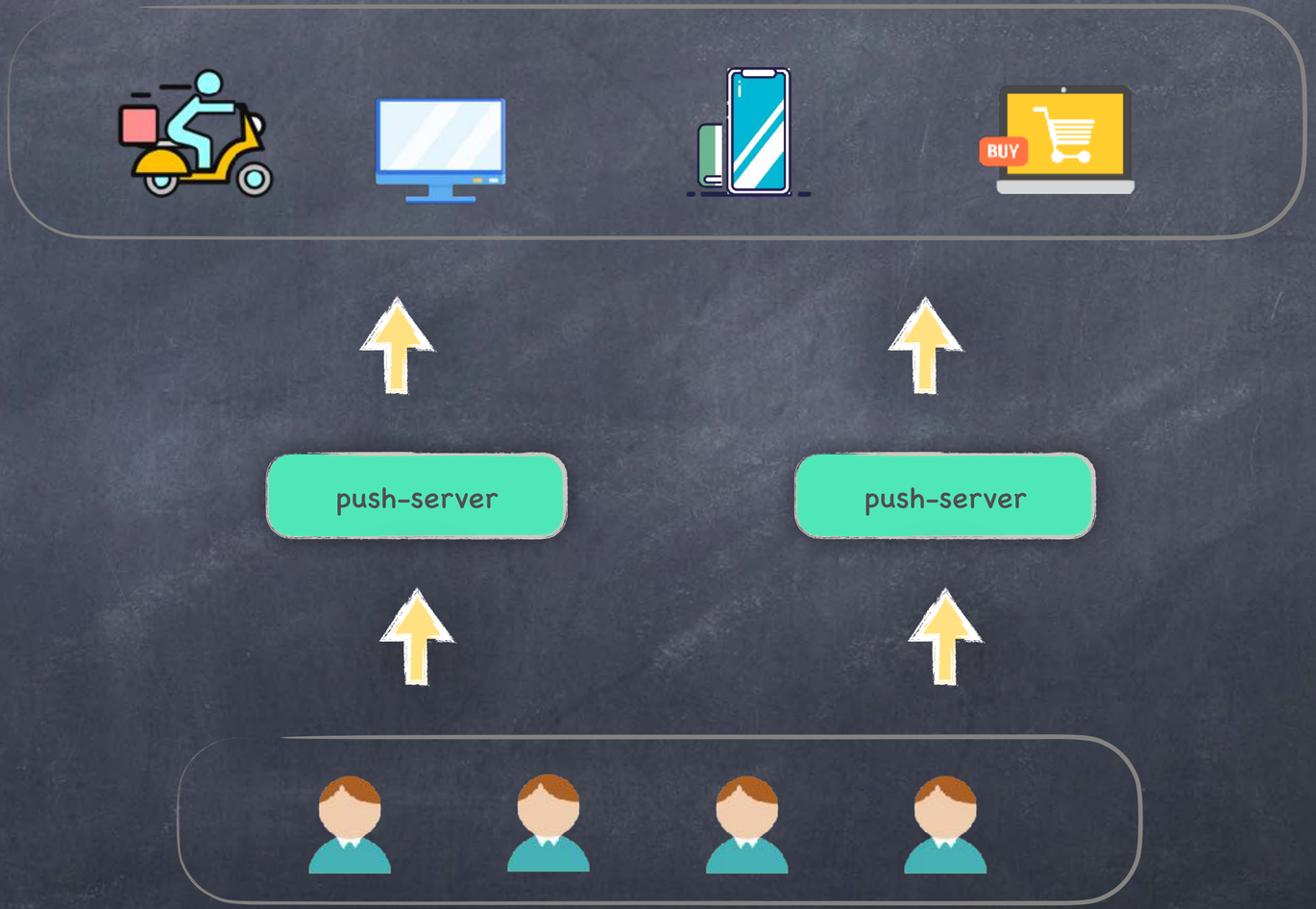
agenda

- 聊上下文
- 系统架构
- 性能调优
- 压力测试
- 运维部署

上下文

应用消息推送

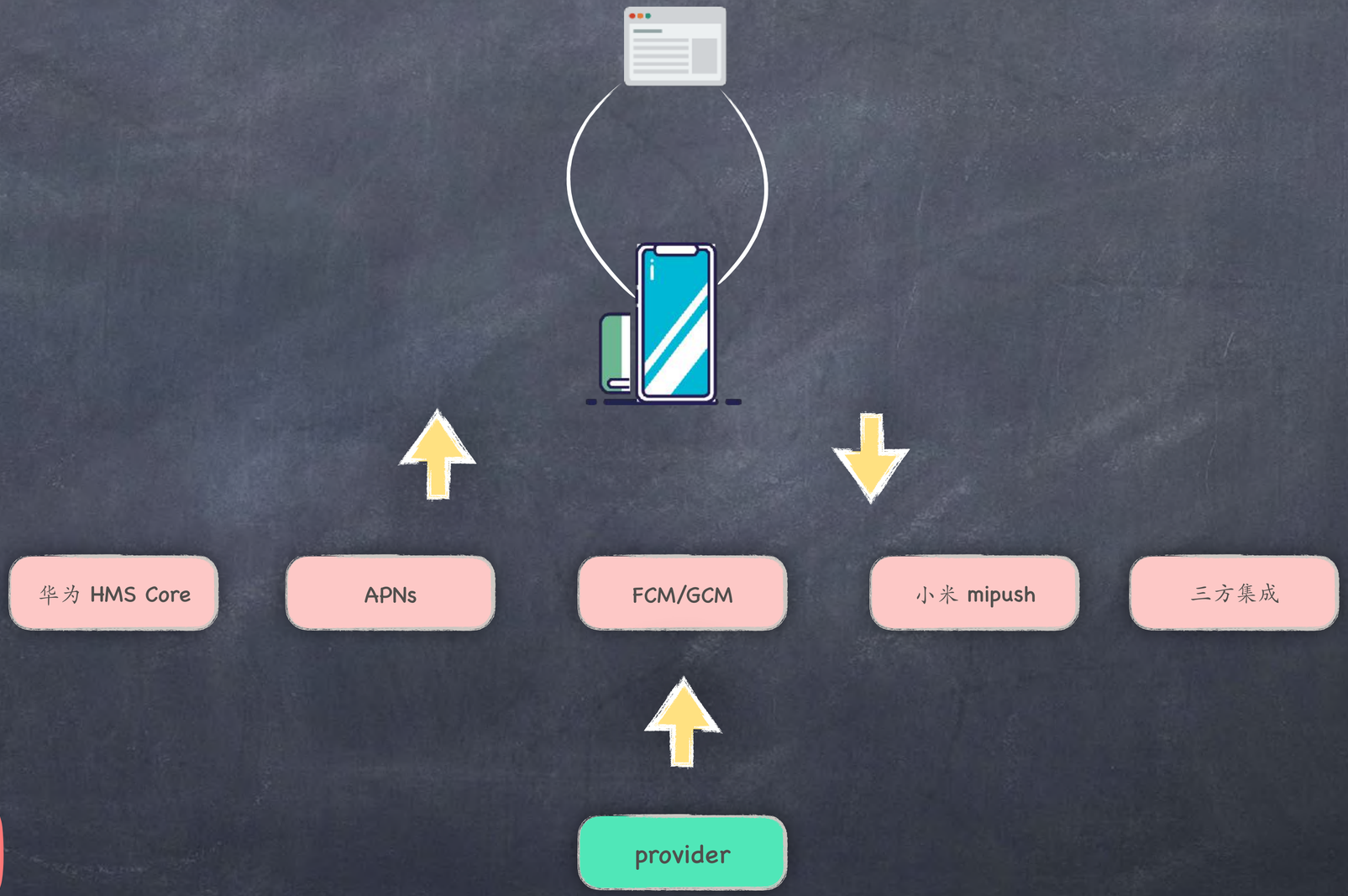
把业务的消息推送到端上



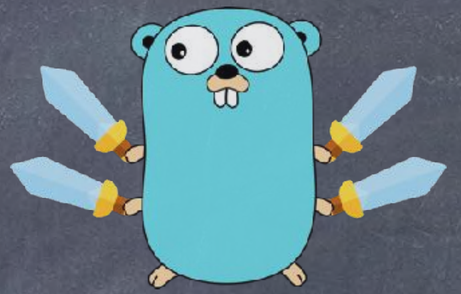
厂商系统推送

- 缺点
  - 限制繁多
  - 额外收费
  - 消息及时性不满足
  - 消息可靠性不满足
  - ... ...

更适合通知栏提醒



rust vs golang



当前痛点

- 代码逻辑很脏
- 代码结构很乱
- 出问题不知怎么排查
- 丢消息风险点太多
- 存在性能问题
- 资源开销比较大
- 无单元测试

一定要重构 !!!!

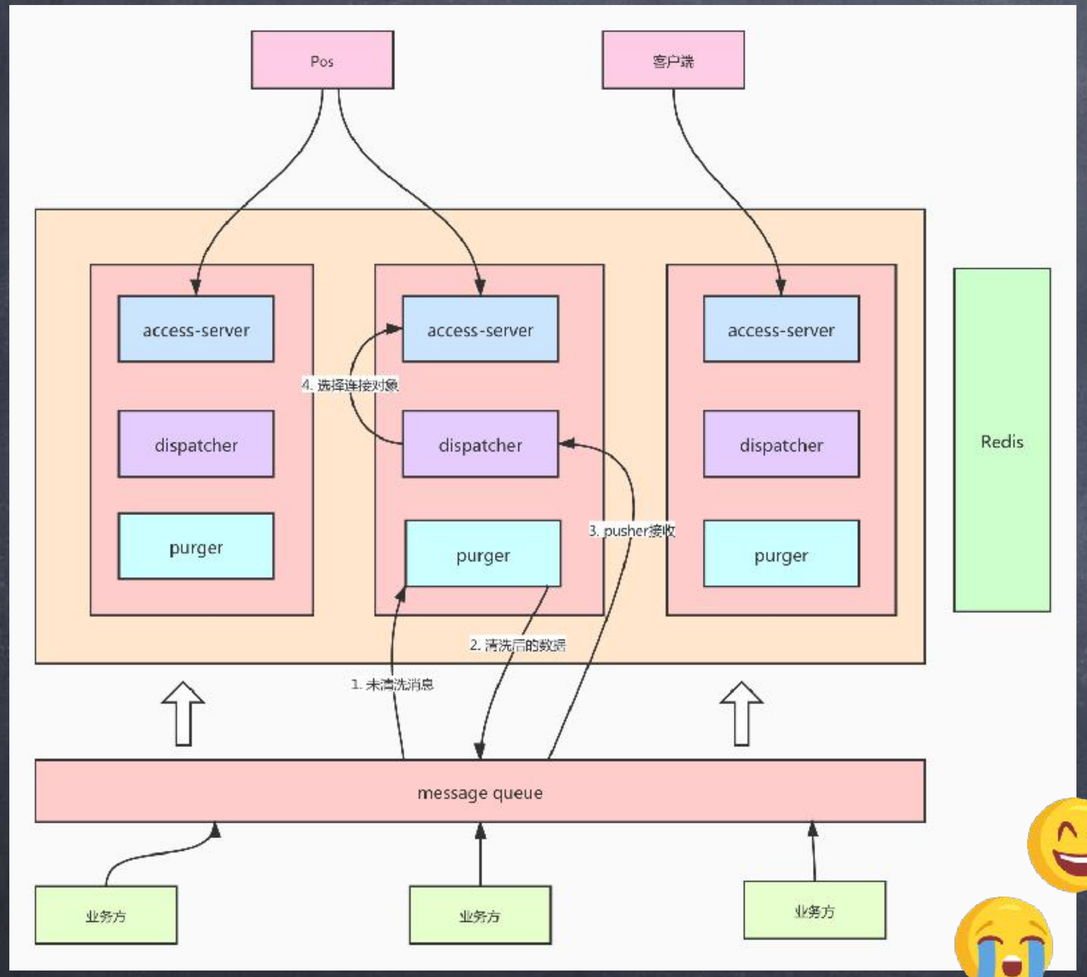
```
for {
    select {
    case client := <-h.register:
        //var deviceTypeCount int
        deviceTypeCount := make(map[string]int, 2)
        typeString := client.getTypeString()
        CentralHubV3Lock.RLock()
        temp, ok := h.clients[client.shopId]
        CentralHubV3Lock.RUnlock()
        if ok {
            CentralHubV3Lock.Lock()
            tempType, okType := temp[client.deviceType]
            if okType {
                if c, ok := tempType[client.clientId]; ok {
                    c.OnUnRegistered()
                    // old type --
                    // new type ++
                    oldType := c.getTypeString()
                    if oldType != typeString {
                        h.deviceTypeCount[oldType]--
                        h.deviceTypeCount[typeString]++
                        deviceTypeCount[oldType] = h.deviceTypeCount[oldType]
                        deviceTypeCount[typeString] = h.deviceTypeCount[typeString]
                    }
                } else {
                    h.clientCount++
                    h.deviceTypeCount[typeString]++
                    deviceTypeCount[typeString] = h.deviceTypeCount[typeString]
                }
                tempType[client.clientId] = client
                client.OnRegistered()
                CentralHubV3Lock.Unlock()
```

```
func concatRedisInfo(shopId, remoteAddr string, wsVersion int, deviceType int, clientId string, userId
int,
    id int64, msgType int, msg string) string {
    return fmt.Sprintf("$$%v$$$$$$%v$$%v$$%v$$%v$$%v$$%v$$$$%v",
        shopId, remoteAddr, wsVersion, deviceType, clientId, userId,
        id, msgType, msg)
}

func ClientVersion5Add(shopId string) error {
    c := redisPool.Get()
    defer c.Close()
    ...
}

func MessageShopAdd(id int64, shopId string, message string, expire int64, version int) error {
    c := redisPool.Get()
    defer c.Close()
    ...
}
```

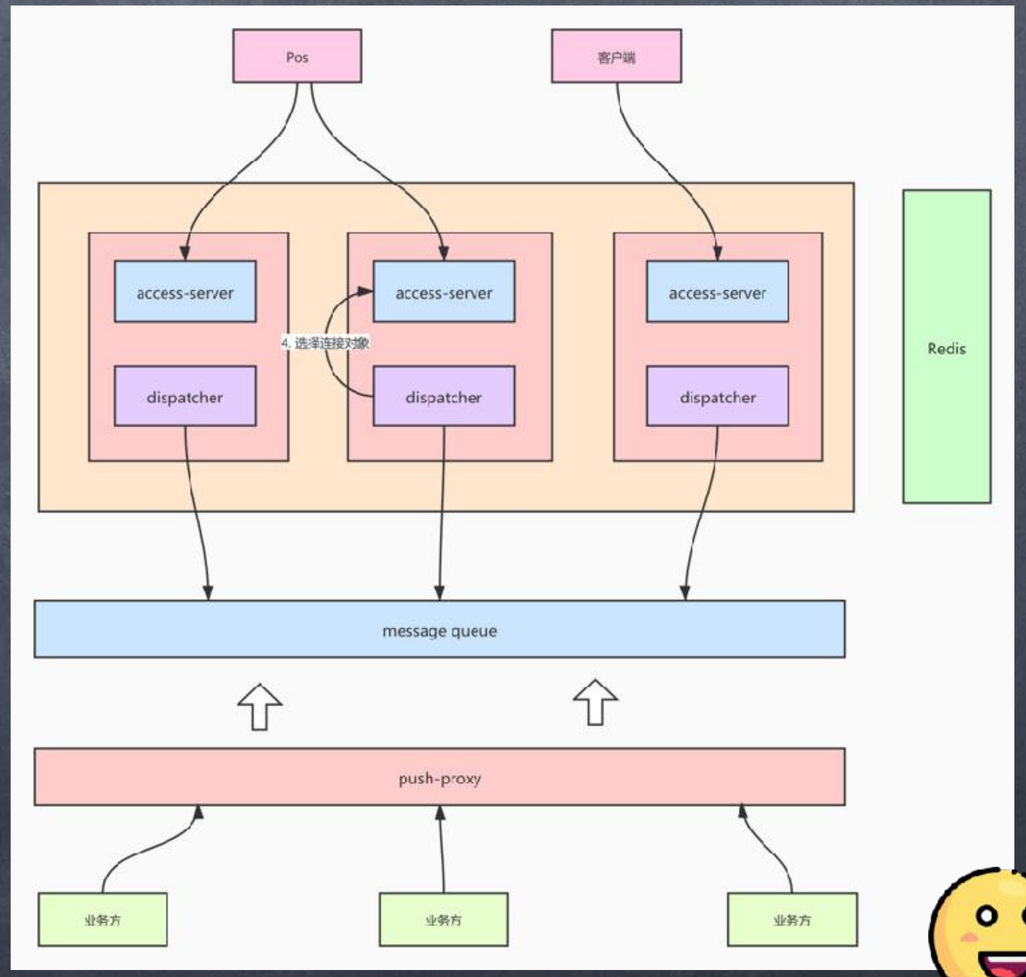
2

架构介绍

架构图



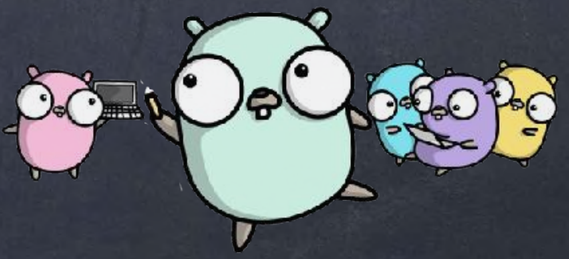
涉及到业务方不好改造 .



理想的架构 .

组件介绍

* access-server
* delayerController
* purger
* dispatcher
* discache
  * L1 - localcache
  * L2 - redis cache
* message queue
* redis



多端设计

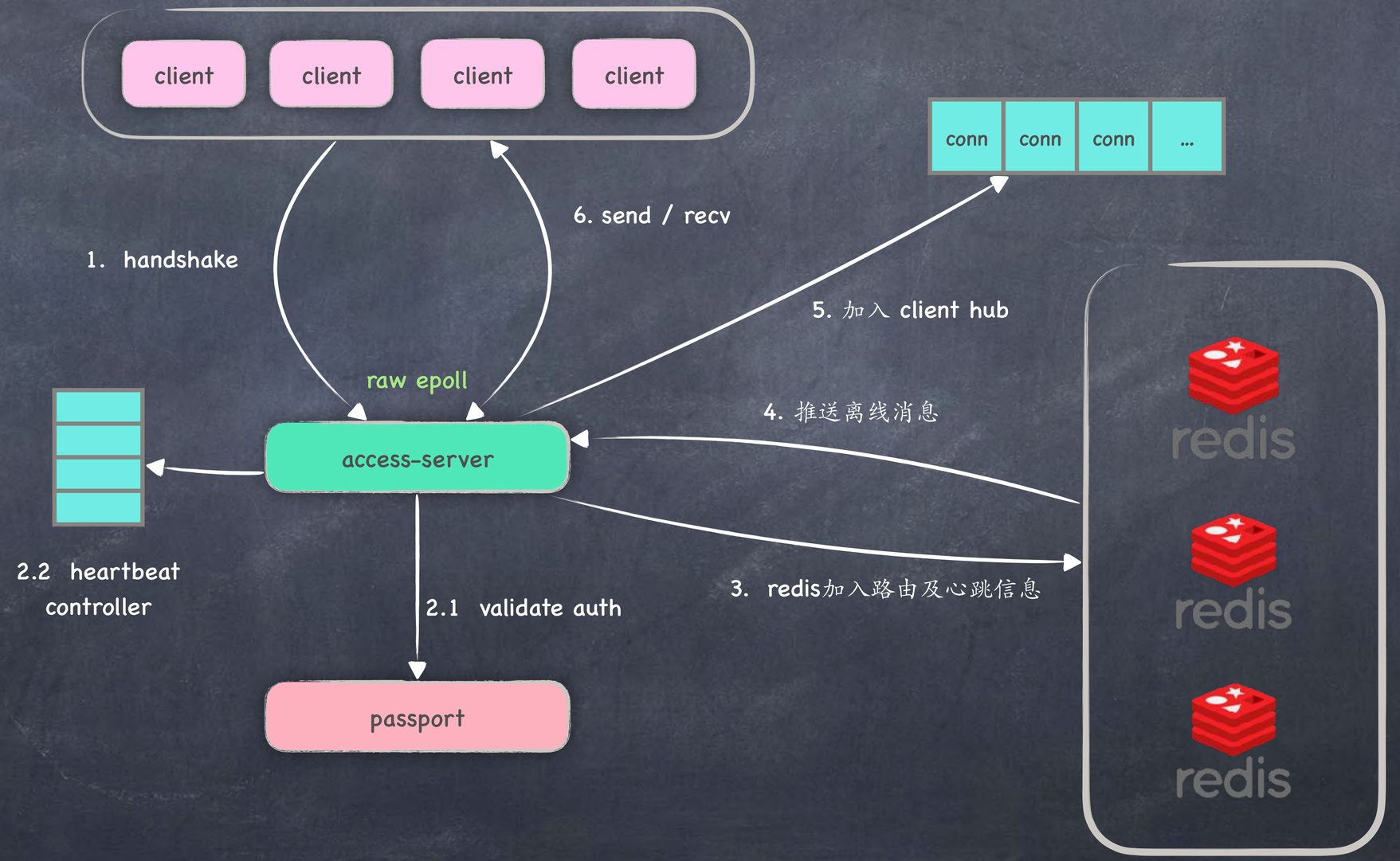
- protocol
  - websocket
  - websocket
  - websocket
  - 😅

websocket over kcp

- pc 端
- 移动端
- pos 端

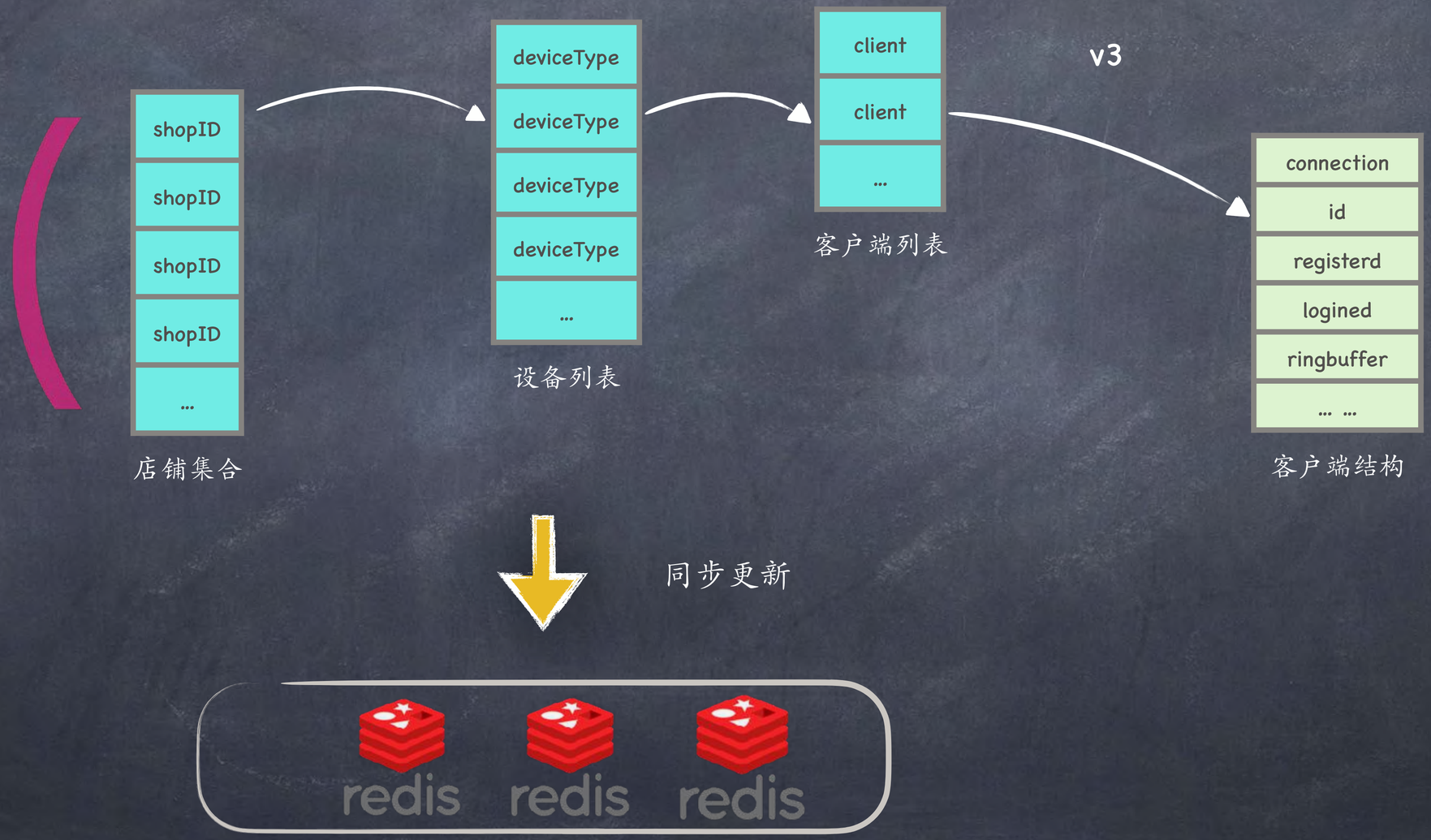
access server

- 握手包进行鉴权
- 自定义心跳管理
- rawepoll 网络编程
- 存放注册信息到redis
- 从redis拉取离线消息
- 用户长连接剔除
- 连接数均衡



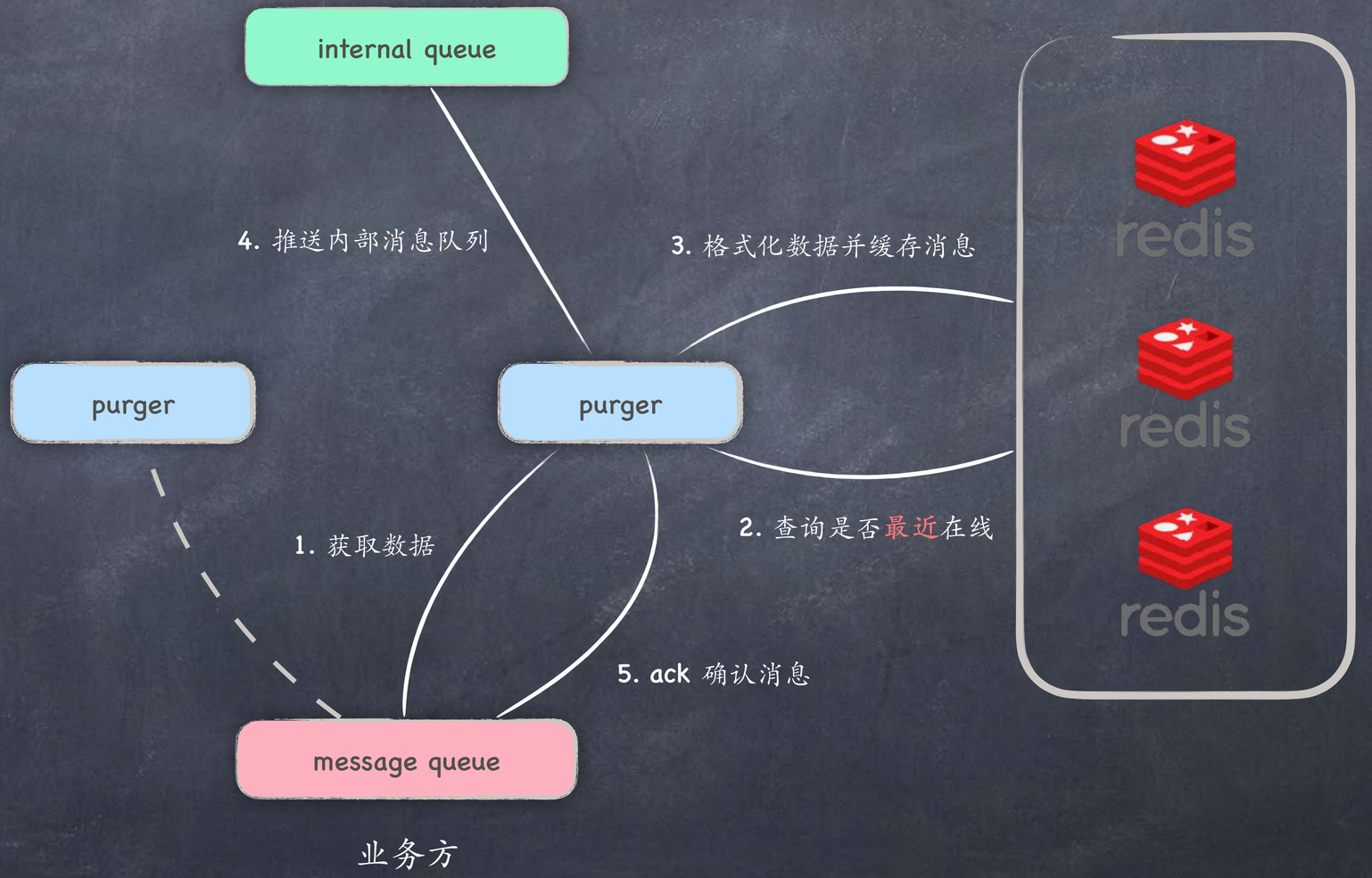
clienthub

- 统计客户端数量
- 增删改查客户端



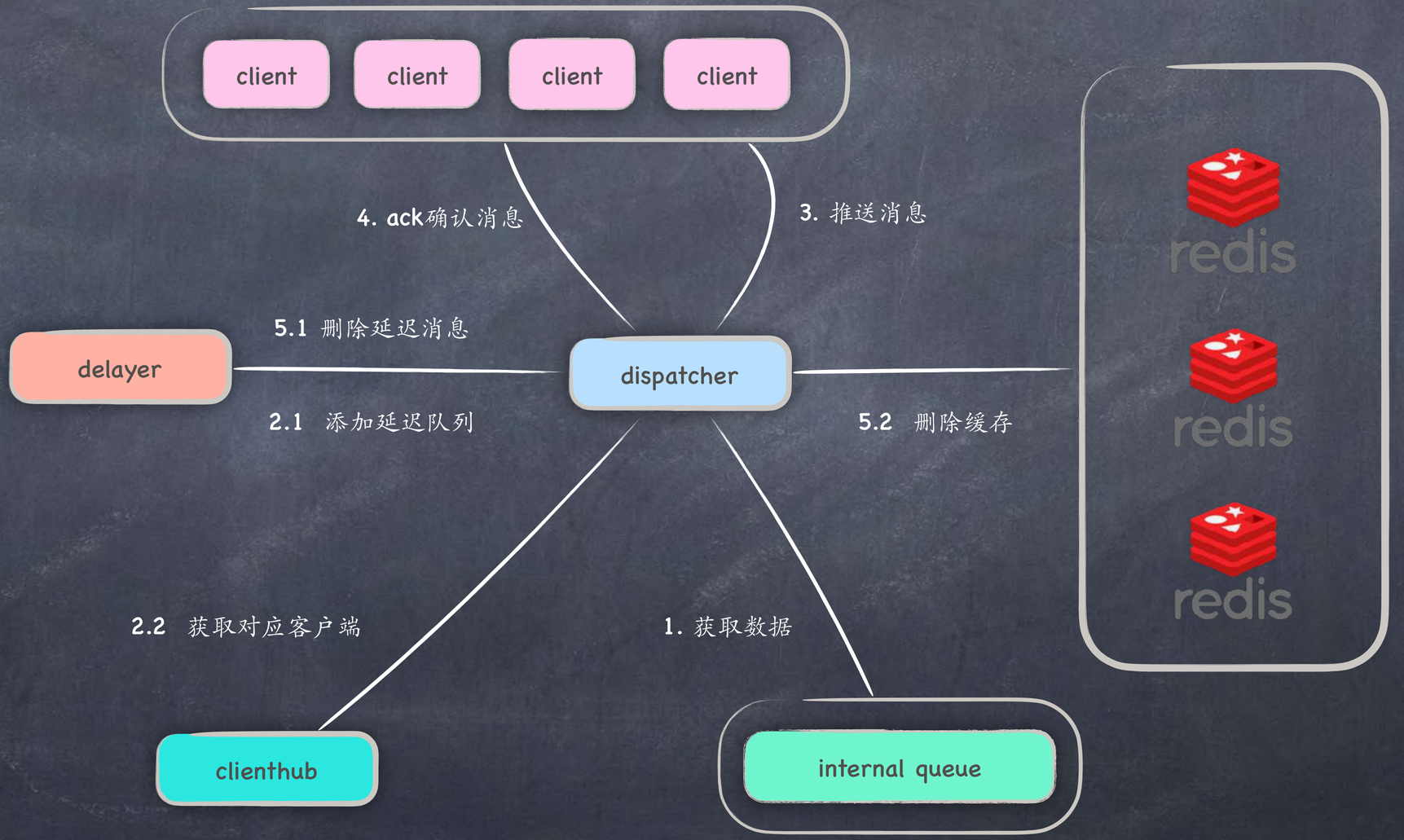
purger

- purger 可横向扩展 !!!
- 一条消息只被一个purger消费
- 手动 ack 确认mq消息可靠

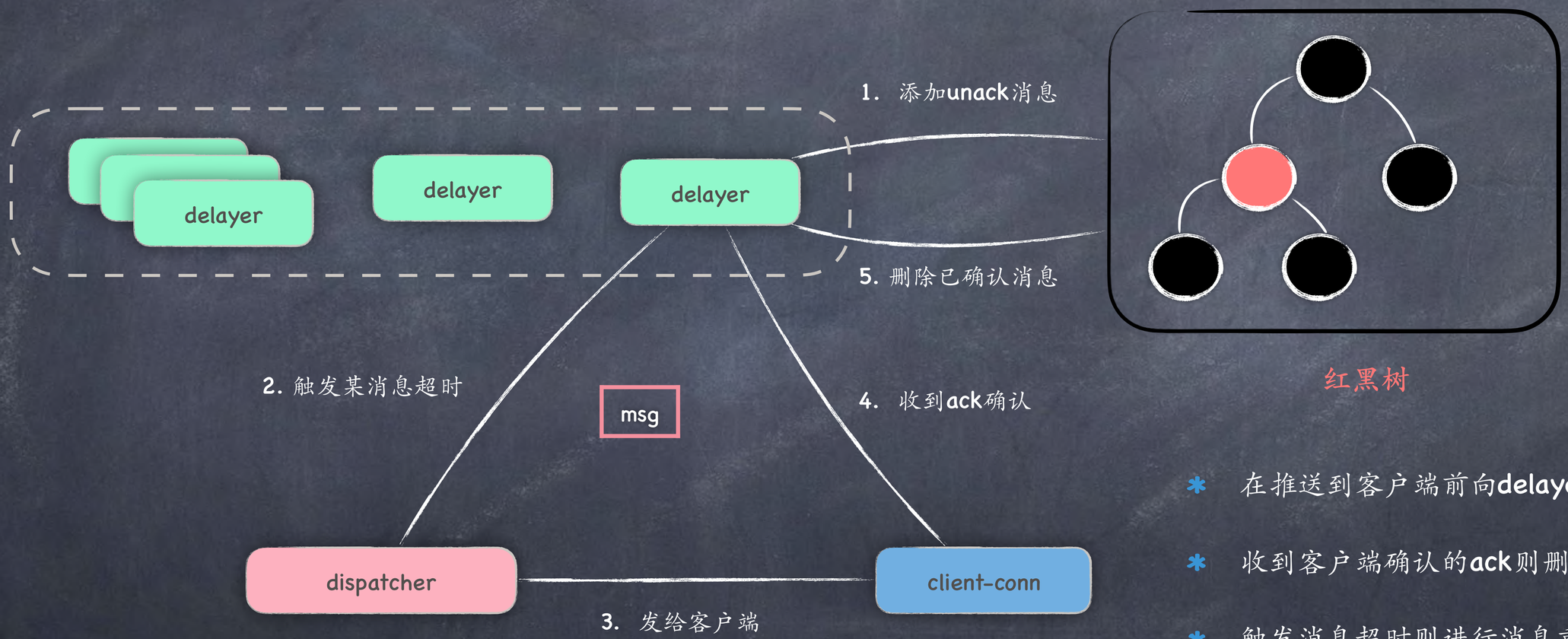


dispatcher

- dispatcher 也可横向扩展 !!!
- mq 采用 pubsub 模式
- 通过延迟队列做重试
- 通过redis做离线消息
- 通过客户端的ack来确认消息



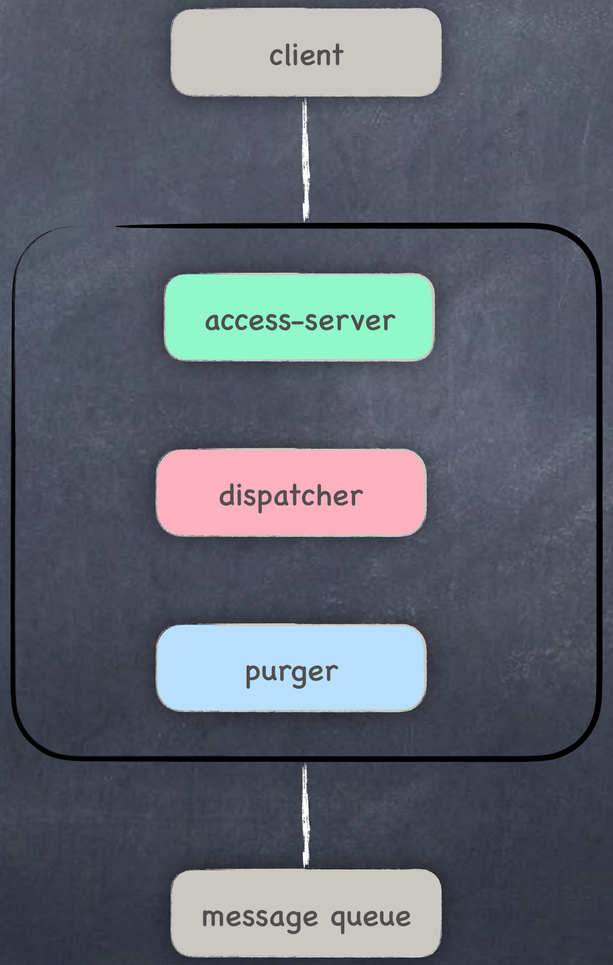
delayer



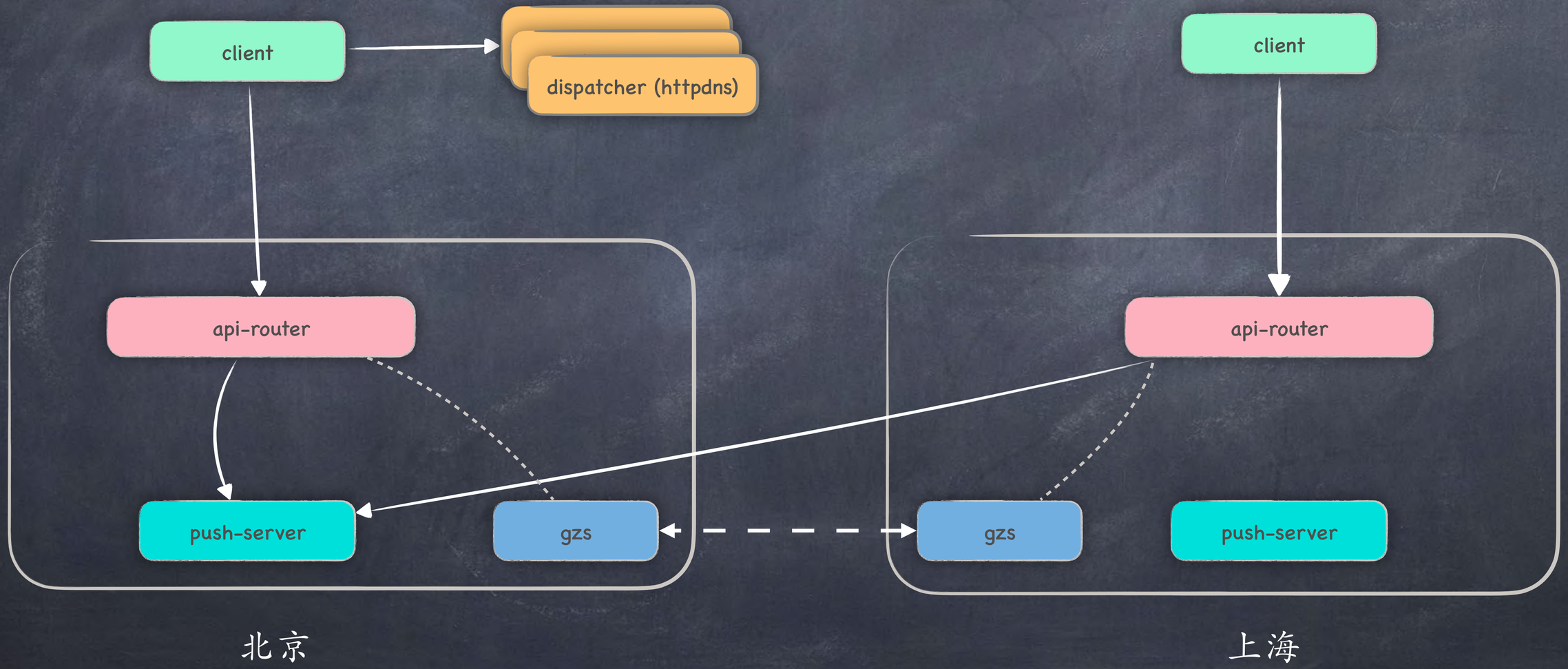
- 在推送到客户端前向delayer添加消息
- 收到客户端确认的ack则删除消息
- 触发消息超时则进行消息重试
- 共重试5次，backoff退避时间间隔

优雅下线

- 绑定 signal 信号
- 先关 purger
- 再关 dispatcher
- access server shutdown
- 等待退出或触发超时退出



异地多活


client
dispatcher (httpdns)
client
api-router
api-router
push-server
gzs
gzs
push-server
北京
上海

可靠性

- purger 使用 mq ack 模式处理消息
  - 避免处理 mq 消息中 crash 丢消息场景?
  - 上下线需求
- 消息会缓存到 redis 一份，作为离线消息存在
  - 避免未转发、转发中就 crash 场景?
- 消息的 ack 确认机制
  - 只要服务端没收到 ack，那么就发起重传
- 离线消息机制
  - 客户端优先拉取离线消息

可用性

- 无状态
  - push-server
- 有状态
  - redis-cluster
  - message queue

- push-server
  - 可扩缩容
  - failover
  - 连接均衡

3

性能优化

协程池

- many go func()
  - more stack ?
  - runtime ?
  - allgs gc ?

- 协程池
  - 流量整形
  - 约束并发
  - 减少runtime开销

gc 优化

```go
var (
    inputMessageV3Pool = sync.Pool{
        New: func() interface{} {
            return new(InputMessageV3)
        },
    }
)

func newInputMessageV3() *InputMessageV3 {
    inmsg := inputMessageV3Pool.Get().(*InputMessageV3)
    return inmsg
}

// pos-service 投递的数据结构
type InputMessageV3 struct {
    ...
}

func (in *InputMessageV3) reset() {
    in.MsgID = ""
    in.StartSendTime = ""
    in.EndSendTime = ""
    in.GroupID = ""
    in.ShopID = ""
    in.OrderKey = ""
    in.SaasOrderKey = ""
    in.ChannelOrderKey = ""
}

func (in *InputMessageV3) putPool() {
    in.reset()
    inputMessageV3Pool.Put(in)
}

...
```

- 使用 sync.pool 实现对象缓冲
- 日志使用 bytes.buffer 拼凑

```go
var (
    defaultEntryPool = sync.Pool{
        New: func() interface{} {
            return new(InfoEntry)
        },
    }
)

func GetEntry() *InfoEntry {
    obj := defaultEntryPool.Get().(*InfoEntry)
    obj.Reset()
    return obj
}

func PutEntry(ie *InfoEntry) {
    defaultEntryPool.Put(ie)
}

func (l *InfoEntry) String() string {
    l.buffer.WriteString(l.ShopID)
    l.buffer.WriteString("$$")

    l.buffer.WriteString(l.SaasOrderKey)
    l.buffer.WriteString("$$")

    l.buffer.WriteString(l.ChannelOrderKey)
    l.buffer.WriteString("$$")

    l.buffer.WriteString(l.ClientAddr)
    l.buffer.WriteString("$$")

    l.buffer.WriteString(l.WsVersion)
    l.buffer.WriteString("$$")

    l.buffer.WriteString(l.UserID)
    l.buffer.WriteString("$$")

    if l.ID != 0 {
        l.buffer.WriteString(common.Int64ToStr(l.ID))
    }
    l.buffer.WriteString("$$")

    if l.MsgType >= 0 {
        l.buffer.WriteString(common.IntToStr(l.MsgType))
    }
    l.buffer.WriteString("$$")

    l.buffer.WriteString(l.MsgID)
    l.buffer.WriteString("$$")

    l.buffer.WriteString(l.BusinessType)
    l.buffer.WriteString("$$")

    l.buffer.WriteString(l.Text)

    return l.buffer.String()
}
```

mq 瓶颈

- 不管怎么？
  - 开多个实例
  - 开多个并发
- rabbitmq 就这么慢 !!!
- 请使用 kafka、rocketmq 这类高性能 mq .

rabbitmq ack 模式过慢 !!!

Queues

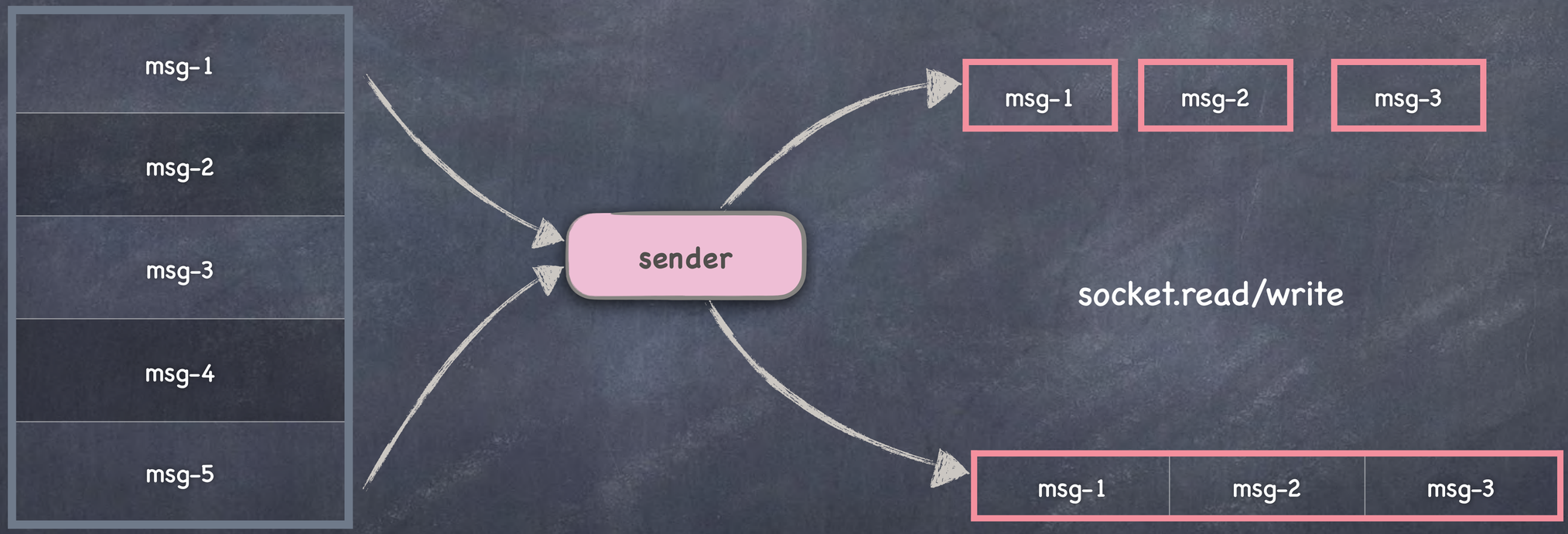
All queues (6)

Overview				Messages			Message rates		
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
hualala:message:push:pos:delivery	classic		running	415,377	100,000	515,377	4,000/s	2,973/s	2,898/s
hualala:message:push:pos:delivery:new	classic		idle	0	0	0			
hualala:message:push:v3:mdb	classic		idle	0	0	0			
hualala:message:push:v3:pos	classic		idle	0	0	0			
hualala:message:push:v3:pos:new	classic		idle	0	0	0			
pusher-server-7bc92d81-1cf2-440e-9520-94f2e7ad1fa5	classic	AD	running	0	18,260	18,260	2,868/s	2,898/s	2,928/s

All queues (5)

Overview				Messages			Message rates		
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
hualala:message:push:pos:delivery	classic		running	42,209	3,000	45,209	1,000/s	3,680/s	3,681/s
hualala:message:push:pos:delivery:new	classic		idle	0	0	0			
hualala:message:push:v3:mdb	classic		idle	0	0	0			

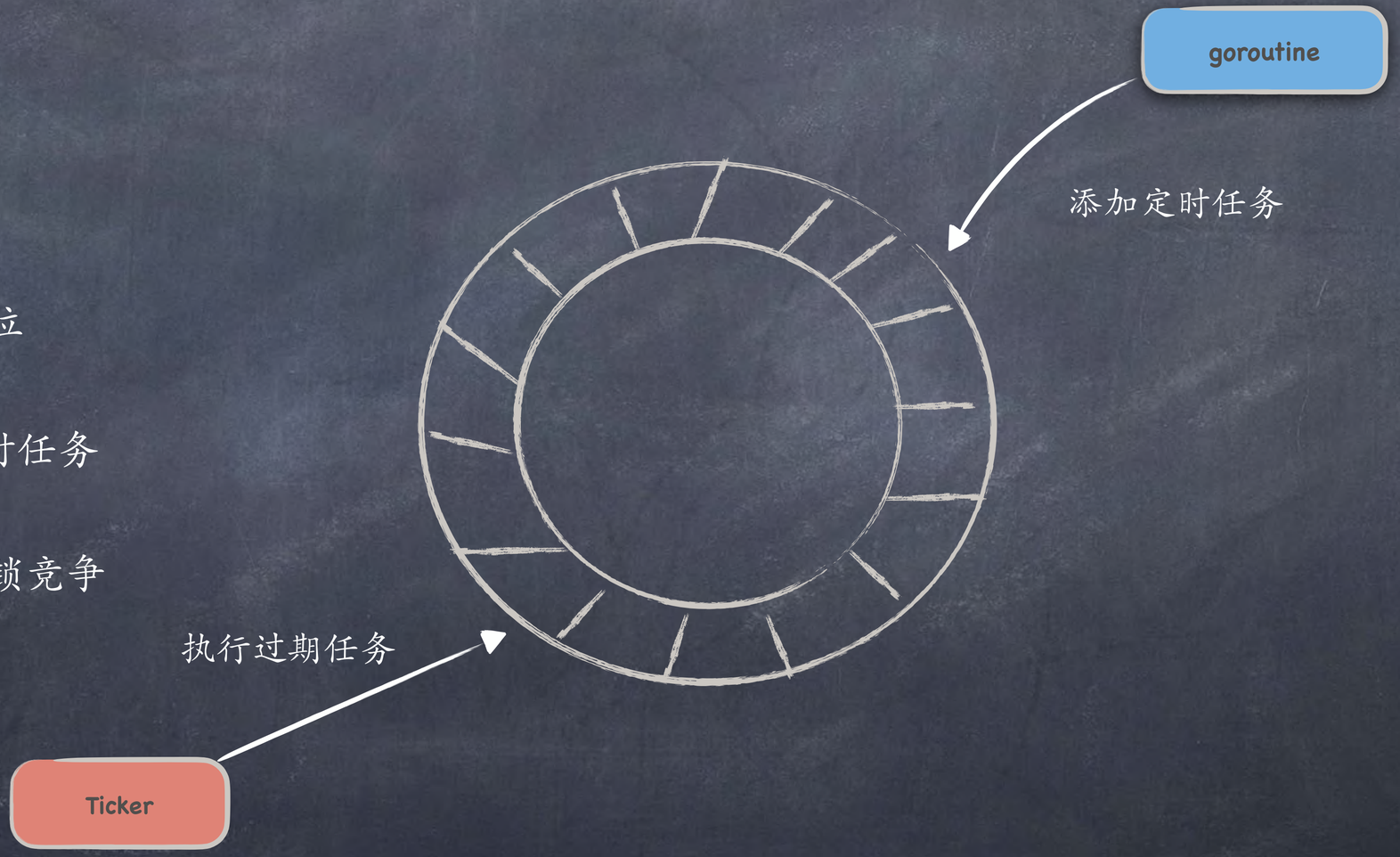
减少系统调用



* 减少上下文切换
* 减少cpu sys的使用率

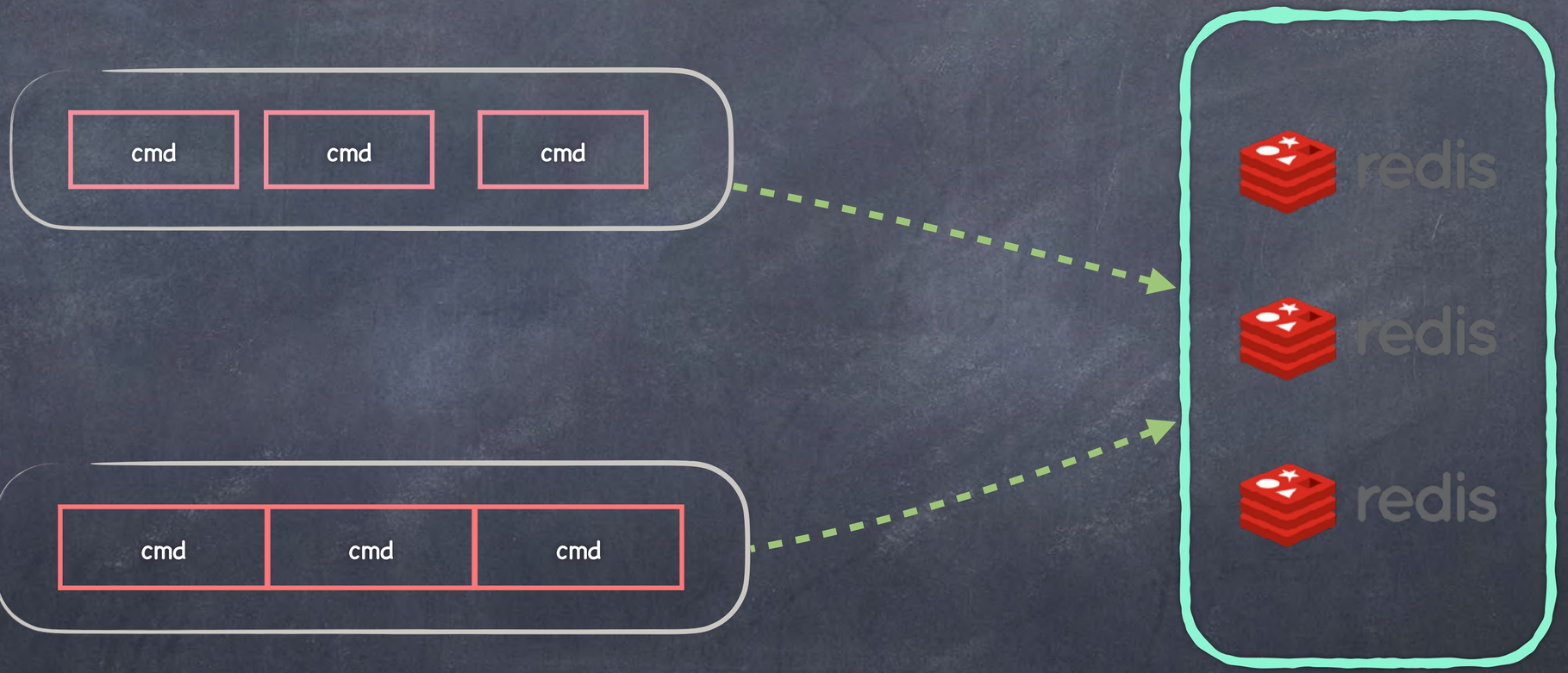
心跳定时器优化

- 实现自定义时间轮
  - 锁分散到每个槽位
  - 使用map存储定时任务
  - 损失精度来减少锁竞争

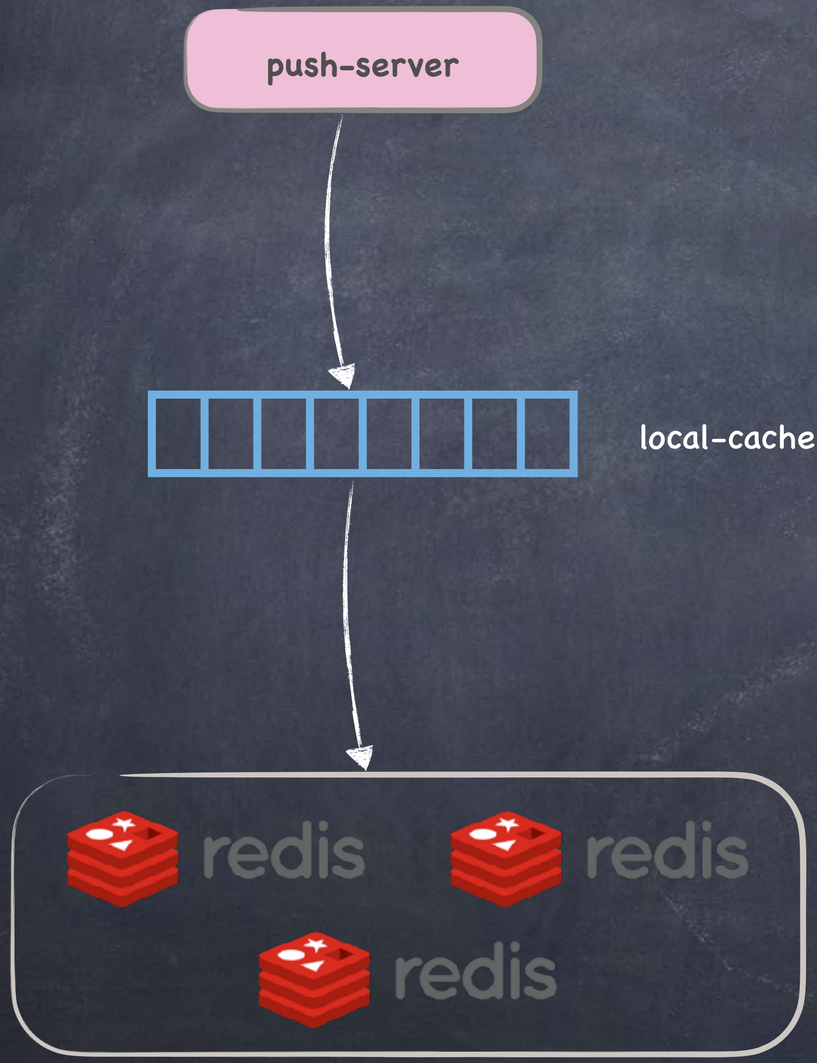


redis pipeline

- 目的
  - 减少时延
  - 减少服务开销
  - 减少 redis 开销
- 不使用管道
  - qps 14w
- 使用管道
  - qps 50w +



localcache



- 目的
  - 减少时延
  - 减少服务开销
  - 减少redis开销
- 做法
  - 优先从缓存中获取状态
  - 再从 redis 获取状态
  - 反向本地更新缓存

开启压缩

- 目的
  - 减少redis存储开销
  - 减少mq的存储开销
- 小文本压缩
  - gzip
  - zlib
  - facebook zstd
  - google snappy

```
// size

source 3880
gzip 1384
zlib 1372
zstd 1474
snappy 1903

// golang benchmark

BenchmarkGzipCompress
BenchmarkGzipCompress-12          4274      252478 ns/op      877825 B/op        48 allocs/op
BenchmarkZlibCompress
BenchmarkZlibCompress-12          4743      255540 ns/op      877196 B/op        52 allocs/op
BenchmarkSnappyCompress
BenchmarkSnappyCompress-12      160815        7202 ns/op       13056 B/op         3 allocs/op
BenchmarkZstdCompress
BenchmarkZstdCompress-12         45781       26186 ns/op       12297 B/op         3 allocs/o
```

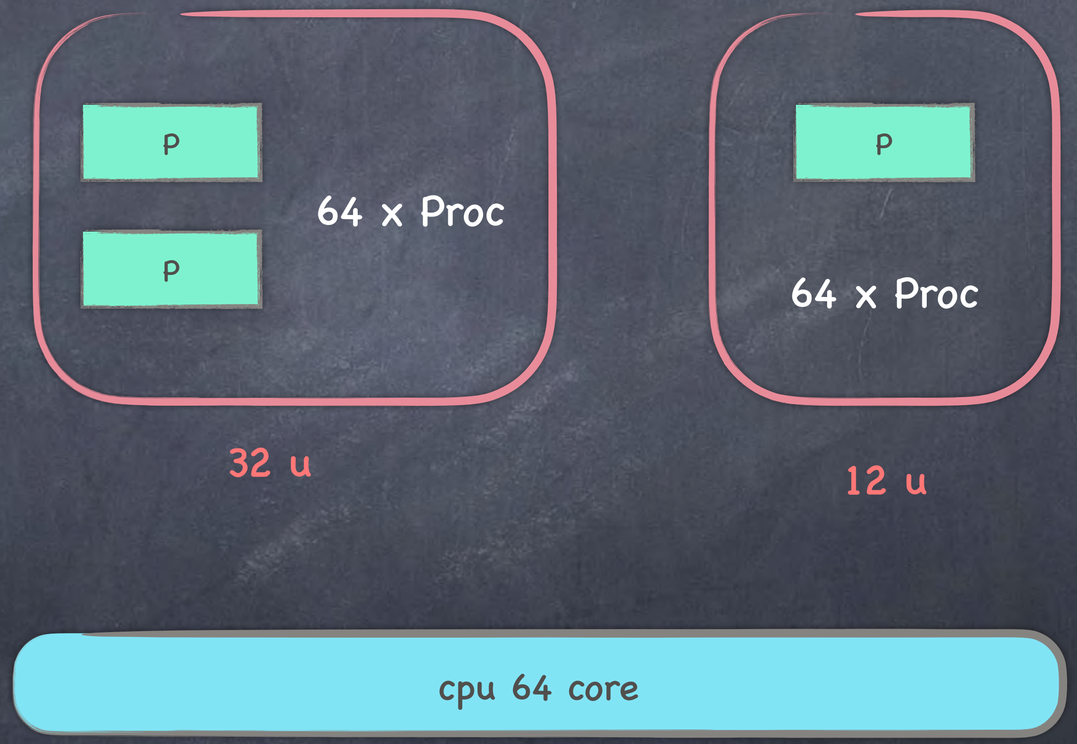
弱网络优化

- 超过心跳时间直接断连接
- 超过 write buffer 直接断链
- 基于 backoff 退避重试发送

golang in docker

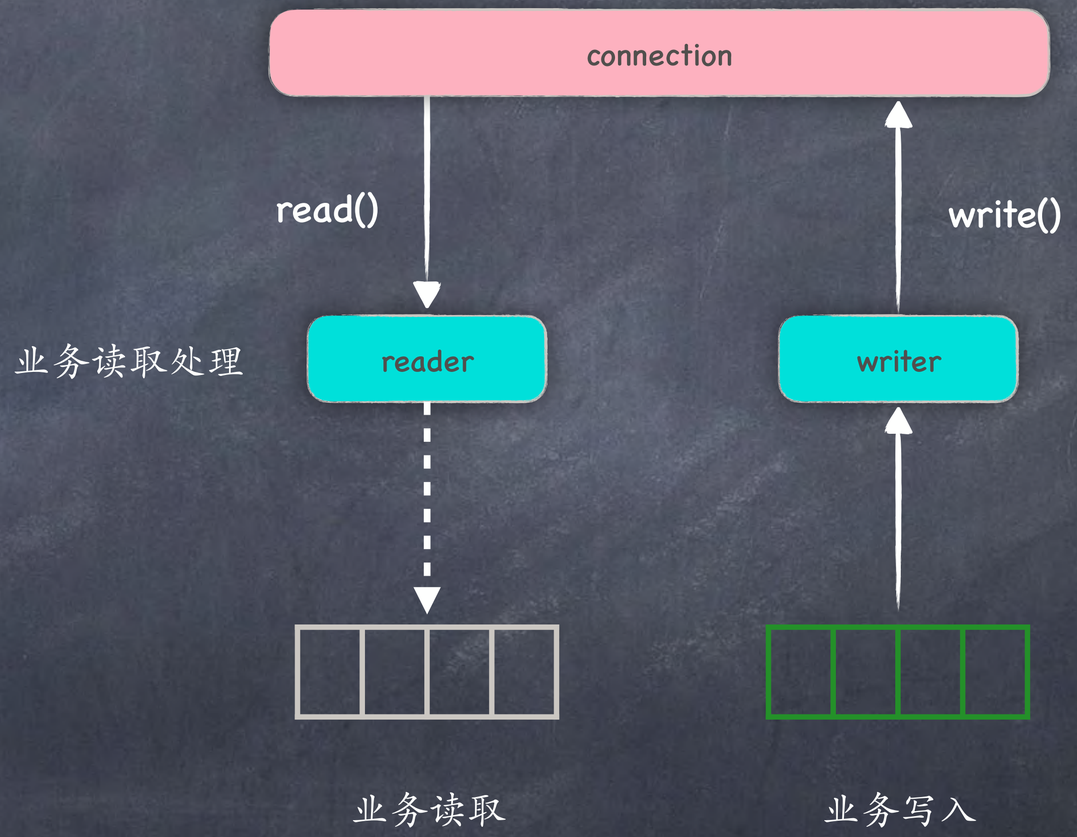
* golang默认P的数量为取cpu core
* docker内cpuinfo为宿主机配置
* P数的增多会增加runtime消耗
* 根据docker的cpu-quota来动态配置P

github.com/uber-go/automaxprocs

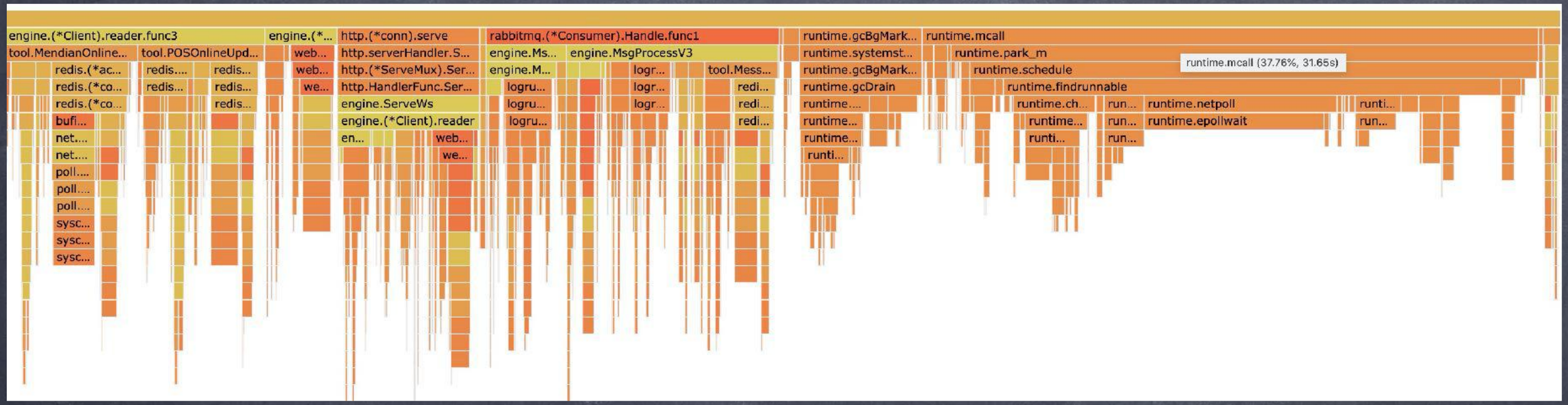


长连接 netpoll 开销

- go netpoll 太爽了 !!!
  - 使用标准net库实现的网络库，可同步编码方式写代码！
  - 以同步方式写异步代码 !!! 底层自动实现非阻塞io !!!
- goroutine-per-connection 模式
  - 长连接场景下通常开读写两个协程
  - 业务层通过channel交互数据！



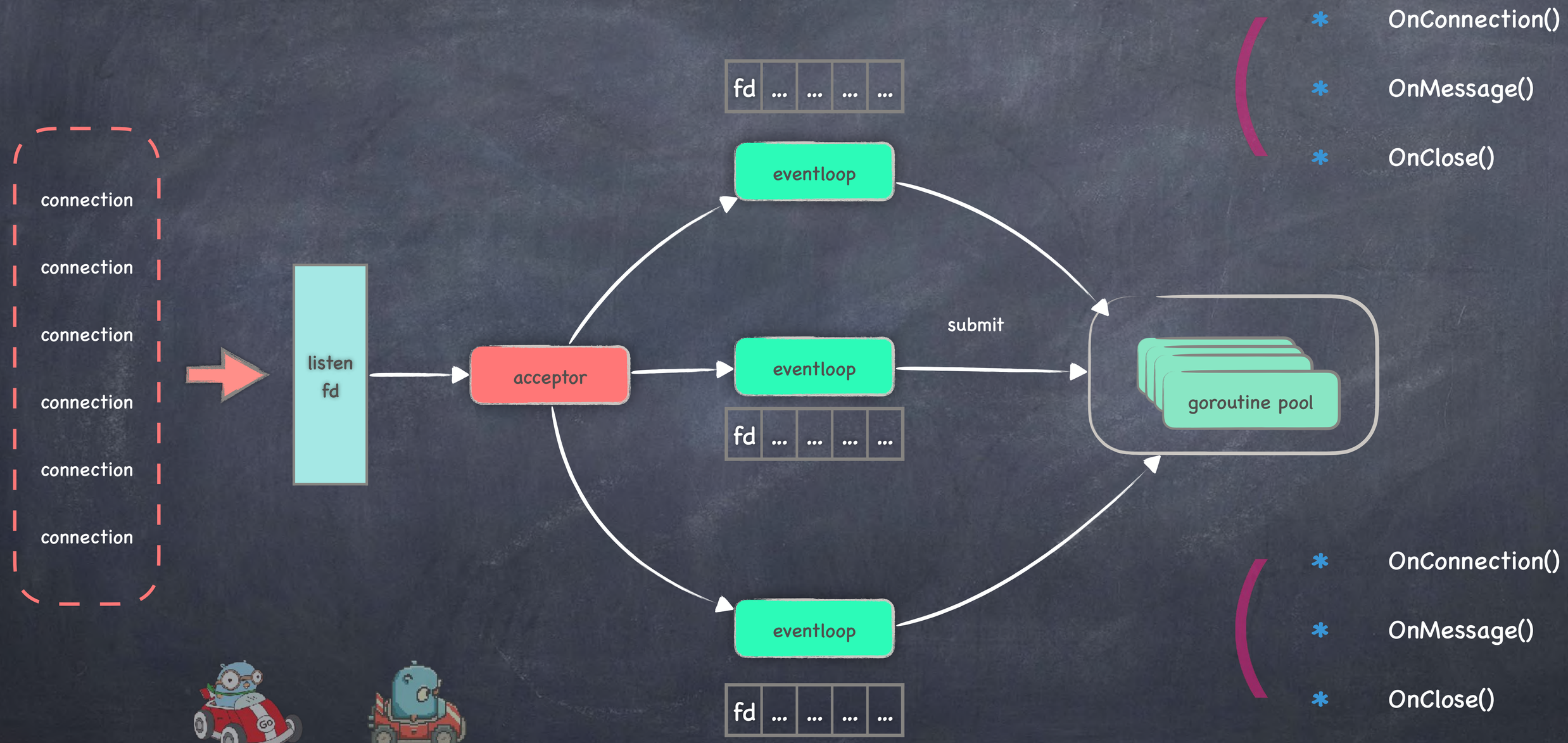
netpoll 开销



* 内存占用开销
* 协程太多造成runtime调度开销

netpoll vs rawepoll

rawepoll mode



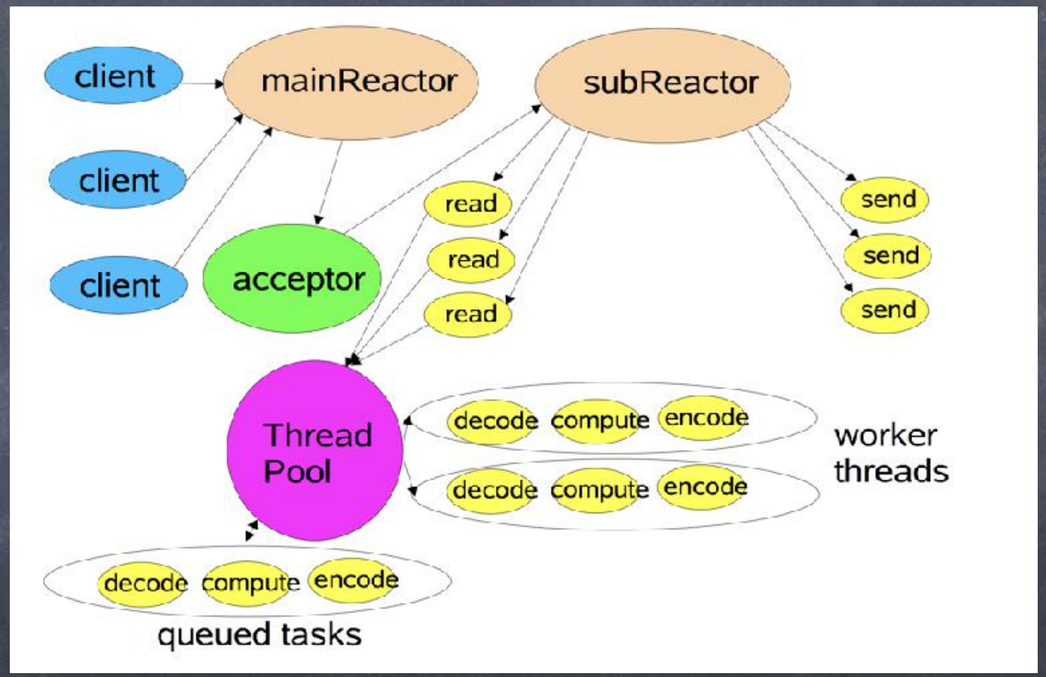
rawepoll optimize

- eventloop 为主循环, 尽量避免阻塞
- 水平触发模式更好写
- 开启多个 eventloop 加大吞吐
- 使用协程池规避主循环中 io 逻辑带来的阻塞
- 建议 eventloop 为cpu核数

```go
// HandleEvent 内部使用, event loop 回调
func (c *Connection) HandleEvent(fd int, events poller.Event) {
    now := time.Now()
    if c.idleTime > 0 {
        _ = c.activeTime.Swap(now.Unix())
    }

    if events&poller.EventErr != 0 {
        c.handleClose(fd)
        return
    }

    if c.outBuffer.Length() != 0 {
        if events&poller.EventWrite != 0 {
            c.handleWrite(fd)
        }
    } else if events&poller.EventRead != 0 {
        c.handleRead(fd)
    }

    c.inBufferLen.Swap(int64(c.inBuffer.Length()))
    c.outBufferLen.Swap(int64(c.outBuffer.Length()))
    ...
}

// Poll epoll wait
func (ep *Poller) Poll(handler func(fd int, event Event)) {
    ...
    events := make([]unix.EpollEvent, waitEventsBegin)
    var wake bool
    for {
        n, err := unix.EpollWait(ep.fd, events, -1)
        ...
        for i := 0; i < n; i++ {
            fd := int(events[i].Fd)
            if fd != ep.eventFd {
                var rEvents Event
                if ((events[i].Events & unix.POLLHUP) != 0) && ((events[i].Events & unix.POLLIN) == 0)
{
                    rEvents |= EventErr
                }
                if (events[i].Events&unix.EPOLLERR != 0) || (events[i].Events&unix.EPOLLOUT != 0) {
                    rEvents |= EventWrite
                }
                if events[i].Events&(unix.EPOLLIN|unix.EPOLLPRI|unix.EPOLLRDHUP) != 0 {
                    rEvents |= EventRead
                }

                handler(fd, rEvents)
            } else {
                ep.wakeHandlerRead()
                wake = true
            }
        }
        ...
    }
}
```

reactor

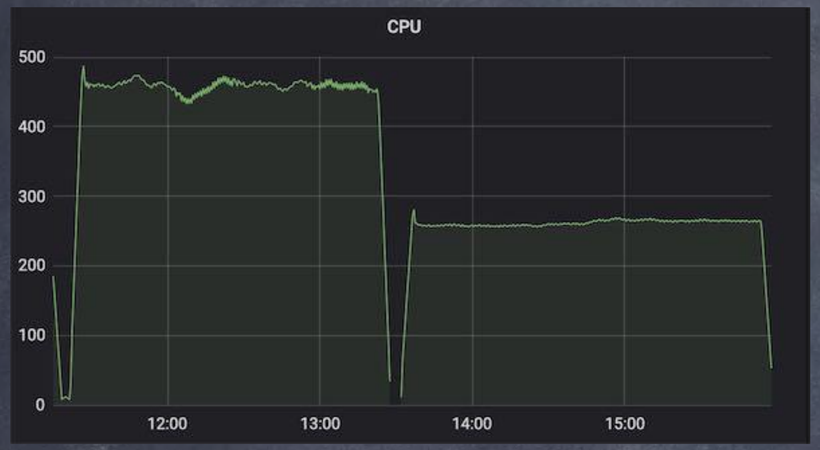
* java netty
* c++ muduo
* golang gnet
* golang gev
* ....



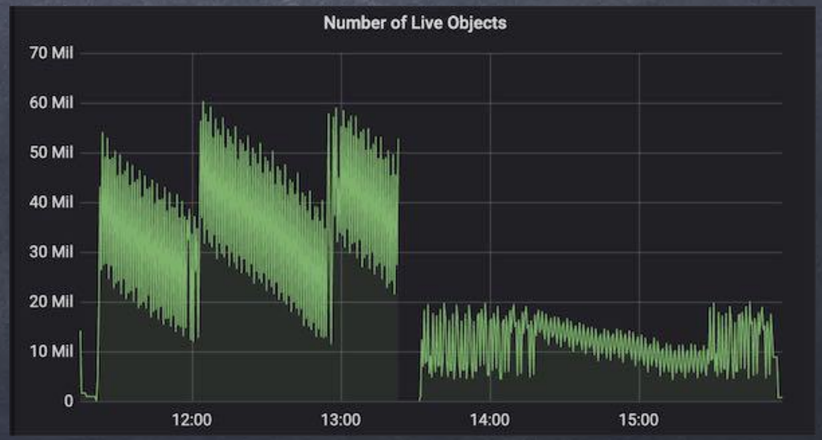
rawepoll mode

- github.com/Allenxuxu/gev
  - reuseport
  - timingwheel
  - ringbuffer
  - thread safe write
  - simple code
  - support websocket protocol
  - 大量的 sync.pool
  - 最关键跟作者是前同事 !!! 😅😄😑

netpoll vs raw epoll



* 20w connections
* biz send 1w msg per 1s
* reduce network latency

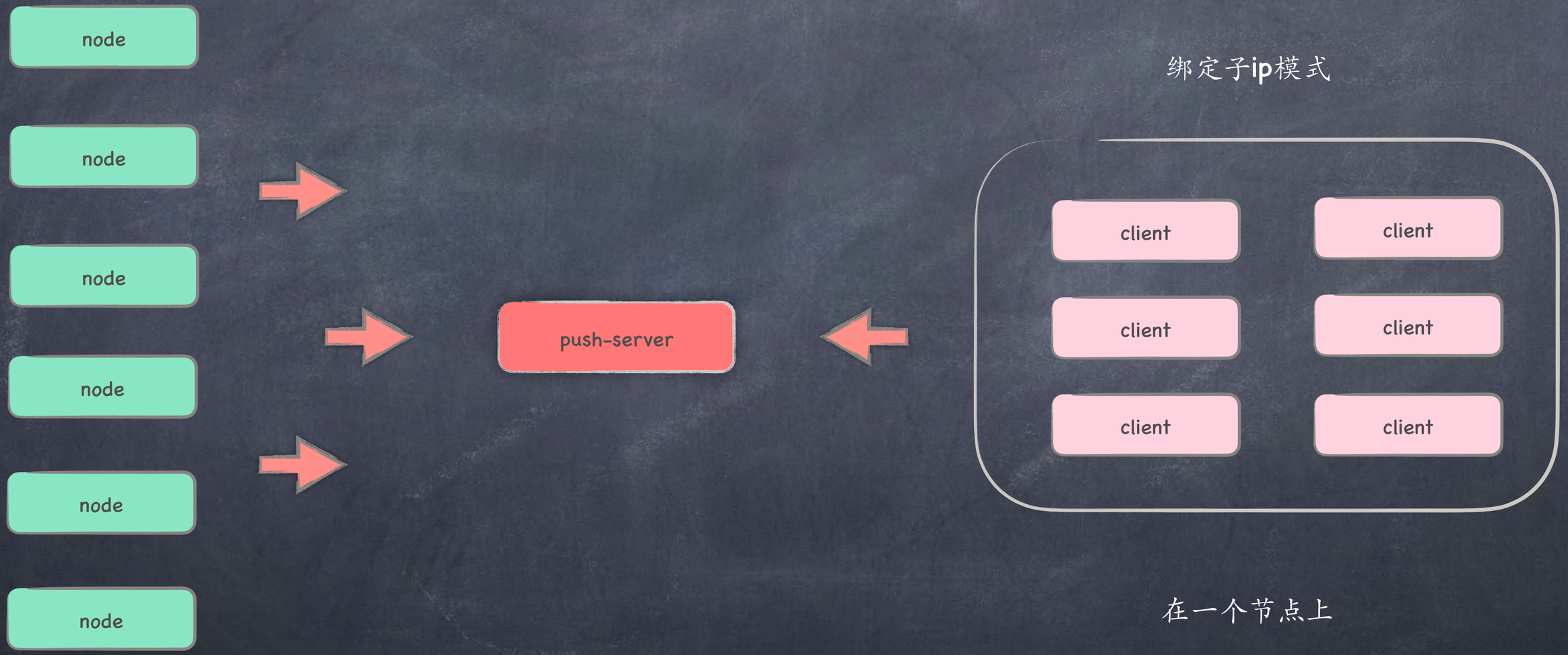


4

压力测试

分布式压测

每个节点发动5w 连接 !!!



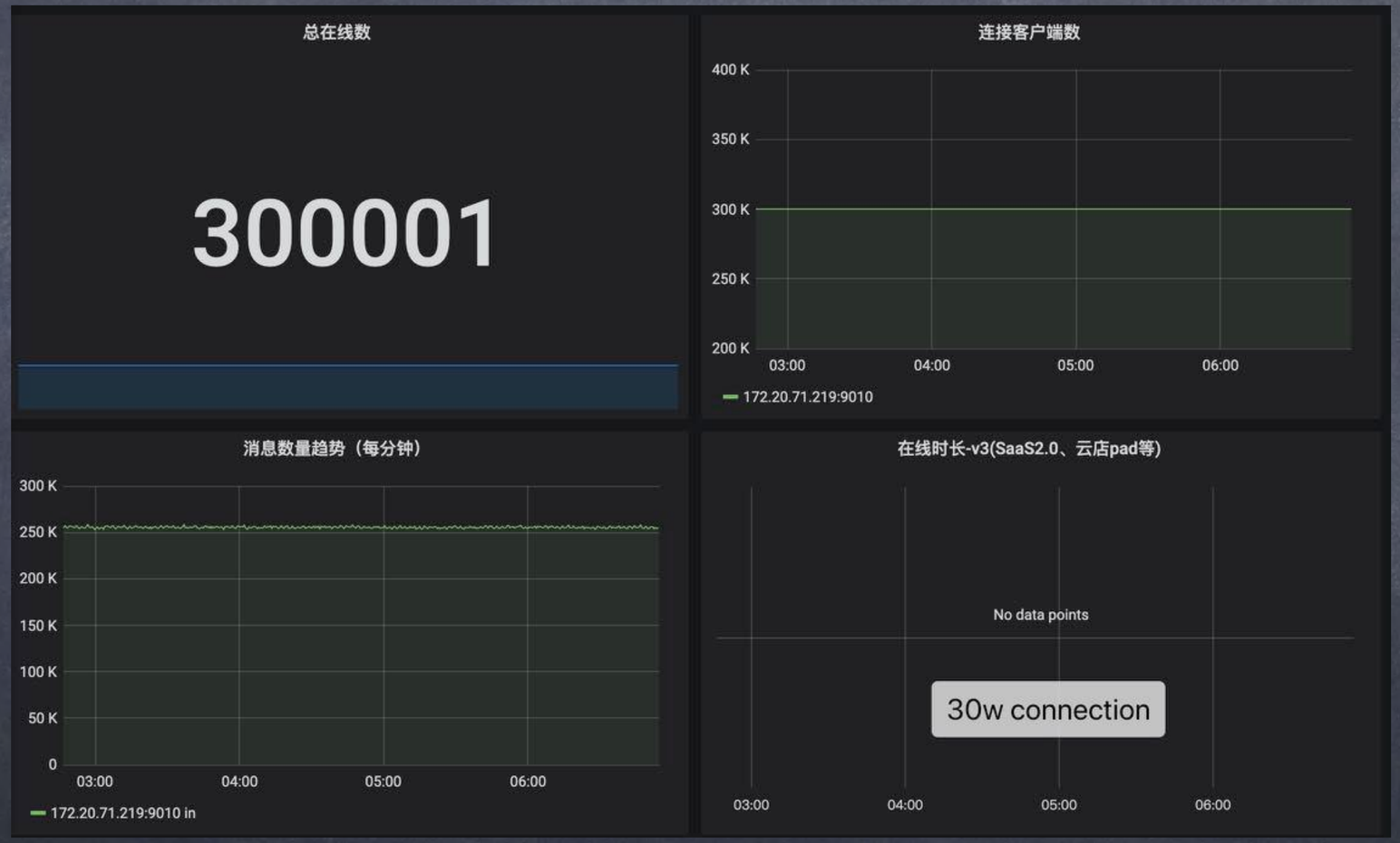
指标

- 加入消息时延指标
- 加入客户端在线时长指标
- 加入错误指标
- 加入警示指标
- 更多指标

尽量多的覆盖内部的耗时指标 !!!

```
clientCount = prometheus.NewGaugeVec(
    prometheus.GaugeOpts{
        Namespace: namespace,
        Name:      "client_count",
        Help:      "Count of connecting client.",
    },
    []string{"wsVersion"},
)

consumeInternalMQDuration = prometheus.NewHistogramVec(
    prometheus.HistogramOpts{
        Namespace: namespace,
        Name:      "consume_internal_mq_duration",
        Help:      "consumeInternalMQTime - consumeBusinessMQTime in milliseconds.",
        Buckets:   []float64{100, 200, 500, 1000, 3000, 5000, 15000, 30000, 60000},
    },
    []string{"wsVersion"},
)

errorCount = prometheus.NewCounterVec(
    prometheus.CounterOpts{
        Namespace: namespace,
        Name:      "error_count",
        Help:      "Count of error.",
    },
    []string{"error"},
)

warningCount = prometheus.NewCounterVec(
    prometheus.CounterOpts{
        Namespace: namespace,
        Name:      "warning_count",
        Help:      "Count of warn.",
    },
    []string{"error"},
)
```

单节点压测


总在线数
300001
连接客户端数
400 K
350 K
300 K
250 K
200 K
03:00
04:00
05:00
06:00
172.20.71.219:9010
消息数量趋势（每分钟）
300 K
250 K
200 K
150 K
100 K
50 K
0
03:00
04:00
05:00
06:00
172.20.71.219:9010 in
在线时长-v3(SaaS2.0、云店pad等)
No data points
30w connection
03:00
04:00
05:00
06:00


consume_internal_mq_duration (purger推入到消费内部队列的耗时)
+Inf
1 min
30 s
15 s
5 s
3 s
1 s
500 ms
200 ms
100 ms
03:00
04:00
05:00
06:00
get_sub_clients_duration (获取client列表时间)
+Inf
5 s
1 s
500 ms
200 ms
100 ms
50 ms
20 ms
10 ms
5 ms
03:00
04:00
05:00
06:00
消息到达率
100.0600%
100.0400%
100.0200%
100.0000%
99.9800%
99.9600%
03:00
04:00
05:00
06:00
instance
confirm_ack_duration (发送到接收ack报文的时间)
+Inf
1 min
30 s
15 s
5 s
3 s
1 s
500 ms
200 ms
100 ms
03:00
04:00
05:00
06:00
send_client_buffer_duration (把消息写到队列的时间)
+Inf
30 s
15 s
5 s
3 s
1 s
500 ms
200 ms
100 ms
03:00
04:00
05:00
06:00
消息的重试次数
No data points
30w connection
03:00
04:00
05:00
06:00


go协程数
10 K
8 K
6 K
4 K
2 K
03:00
04:00
05:00
06:00
172.20.71.219:9010
CPU
400
399
398
397
396
395
394
393
03:00
04:00
05:00
06:00
172.20.71.219:9010
内存
3.4055 GiB
3.4053 GiB
3.4051 GiB
3.4049 GiB
3.4047 GiB
3.4045 GiB
03:00
04:00
05:00
06:00
172.20.71.219:9010
go_gc_duration_seconds
50 ms
40 ms
30 ms
20 ms
10 ms
0 ns
03:00
03:30
04:00
04:30
05:00
05:30
06:00
06:30
min max avg current
172.20.71.219:9010-quantile 0 252 µs 316 µs 270 µs 262 µs
172.20.71.219:9010-quantile 0.25 424 µs 486 µs 458 µs 465 µs
172.20.71.219:9010-quantile 0.5 520 µs 603 µs 553 µs 562 µs
172.20.71.219:9010-quantile 0.75 650 µs 758 µs 696 µs 677 µs
172.20.71.219:9010-quantile 1 1 ms 43 ms 9 ms 1 ms
错误趋势
1.0
0.8
0.6
0.4
0.2
0
30w connection
03:00
04:00
05:00
06:00
172.20.71.219:9010 / createGoroutine

“ Q&A ”

- xiaorui.cc