

GOLANG在京东的应用

概况

1. Golang 开发的项目
2. 实现技术与经验
3. 正在进行的工作

Golang开发的项目

- 京东云消息推送系统(团队人数:4)
 - 单机并发tcp连接数峰值118w
 - 内存占用23G(Res)
 - Load 0.7左右
 - 心跳包 4k/s
 - gc时间2-3.x s

Golang开发的项目

- 云存储(团队人数12)

 - 小文件存储

 - 块存储

 - 通用存储(大部分兼容s3)

 - 除存储引擎用c/c++， sdk用java， 其它均用go实现

兄弟团队开发的项目：自动部署系统

快乐与痛苦同在

- happy

简洁，开发速度快

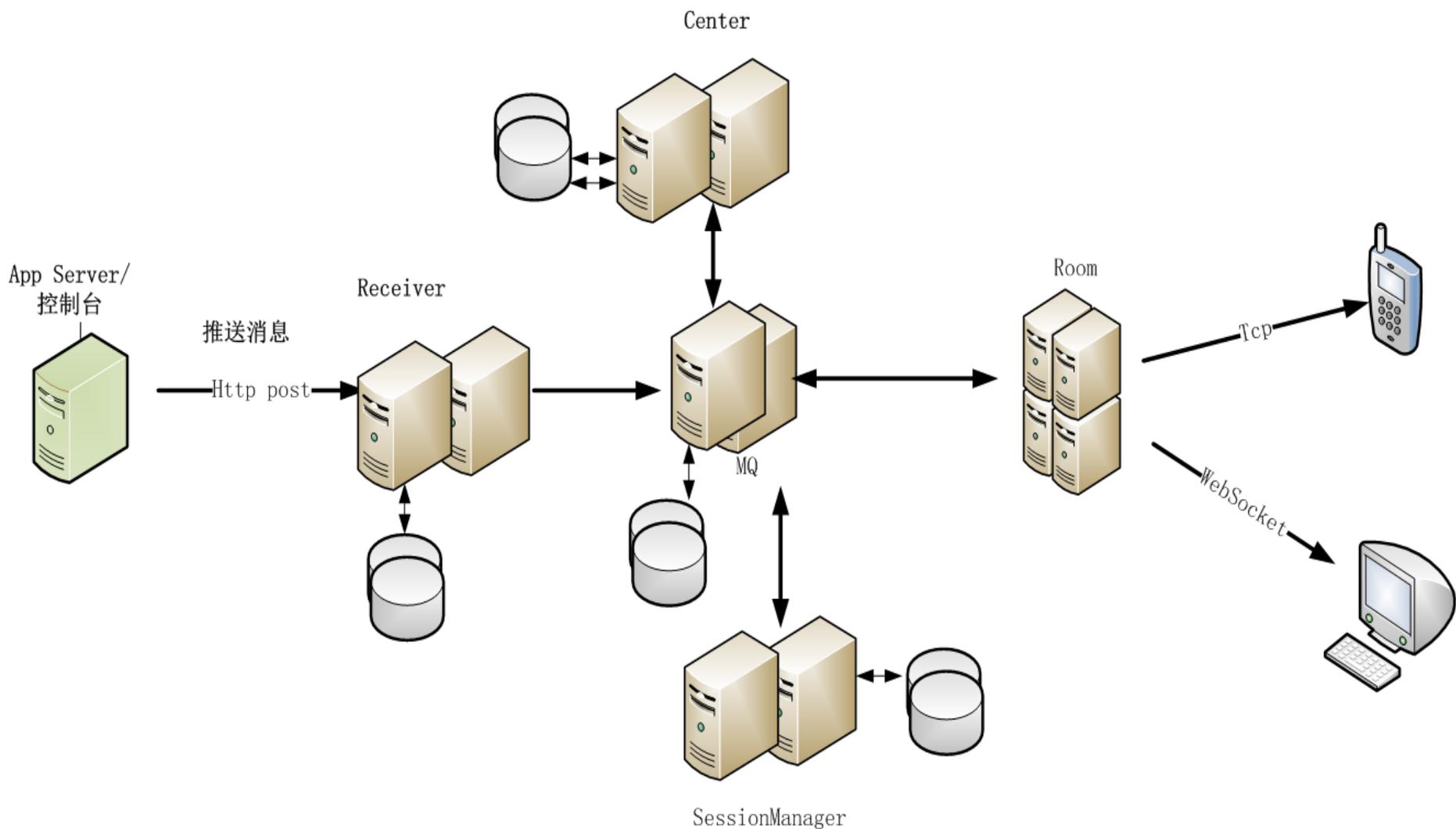
并发实现方便，基础库完善

- pains

GC

还是GC

消息推送整体架构



关于消息队列

- Redis + lua实现(高可用: vip + replication)
- 那么多现成的, 为什么重复造轮子?
- 简单, 轻量, 高性能, 持久化, 有问题也hold得住
- Rabbitmq: 不懂erlang, 整个东西搞得太复杂
- Kafka & activemq: 不会java, 另外过于复杂

关于消息队列

- Crash safe
- Worker取到消息后crash，消息会超时重发，不会丢失
- Worker从队列中pull消息，若队列不空，则持续取消息，否则sleep 1/20秒(可以结合redis的pub/sub机制实现实时消息通知，暂未实现)
- 多个worker
- 消息队列的逻辑由lua维护

数据库选型

- 个人觉得基本什么数据库(拆分)都能搞定，无明显喜好
- Mongodb：正在使用，异步写入，sharding + replication 公司有高手，无需自己操心，省心
- Mysql：个人偏好
- Couchbase：兴趣较大，迁移做得不错，目前国内商用不多，暂不冒险☺
- 中途硬盘坏过两次，有raid，直接热插拔更换，对服务没有任何影响

Session信息存储

- 分布式cache, redis
- 自己在sdk中做sharding? 麻烦
- Twemproxy: 一致性hash, redis主从 + vip
- 部署方式: 和应用部署在一起, 应用连接localhost的twemproxy

消息路由算法

- 由于移动网络不稳定，客户端可能短时间出现在多个server上，仅仅通过客户端id无法区分
- 引入session id，自增，由各个server分配
- Session存储格式Clientid:<addr:server1:id:100, addr:server2:id:123>
- 同时将消息路由到server1和server2

优化的基本原则

- 优化方案不会显著提高维护成本
- 有良好的兼容性，严格遵守golang的规范，以便更好的利用新版本的改进
- 先优化明显的热点
- 优化方案在其它语言也有效(内存池，连接池)

一些经验

- 1.Object pool (gc & performance)
- 2.Group commit
- 3.JSON encoding/decoding (gc & performance)
- 4.标准库优化(network, gc & performance)
- 5.Defer是性能和gc杀手
- 6.Dump信息获取

Object pool (gc & performance)

一个数组实现的栈

代码示例：

```
func (c *Cache) Put(x interface{}) {
    c.mu.Lock()
    if len(c.saved) < cap(c.saved) {
        c.saved = append(c.saved, x)
    } else {
        logging.Warning(c.name, "is full, you may need to increase pool size")
    }
    c.mu.Unlock()
}

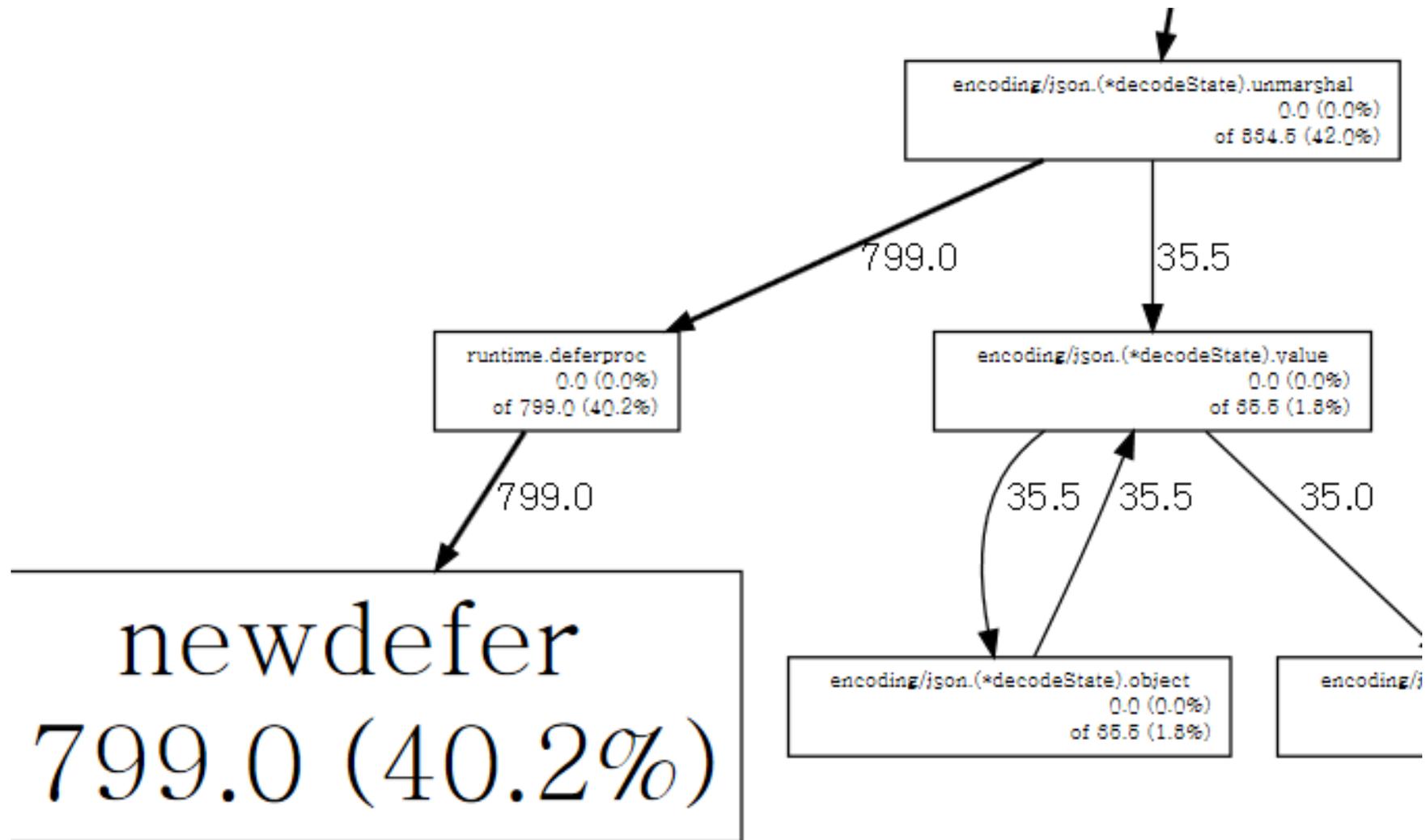
func (c *Cache) Get() interface{} {
    c.mu.Lock()
    n := len(c.saved)
    if n == 0 {
        c.mu.Unlock()
        return c.new()
    }
    x := c.saved[n-1]
    c.saved = c.saved[0 : n-1]
    c.mu.Unlock()
    return x
}
```



关于json

- 标准库的实现性能比较挫，由于使用了反射，还有defer，profiling时轻松上榜
- 代码片段和热点图

Json unmarshal热点图



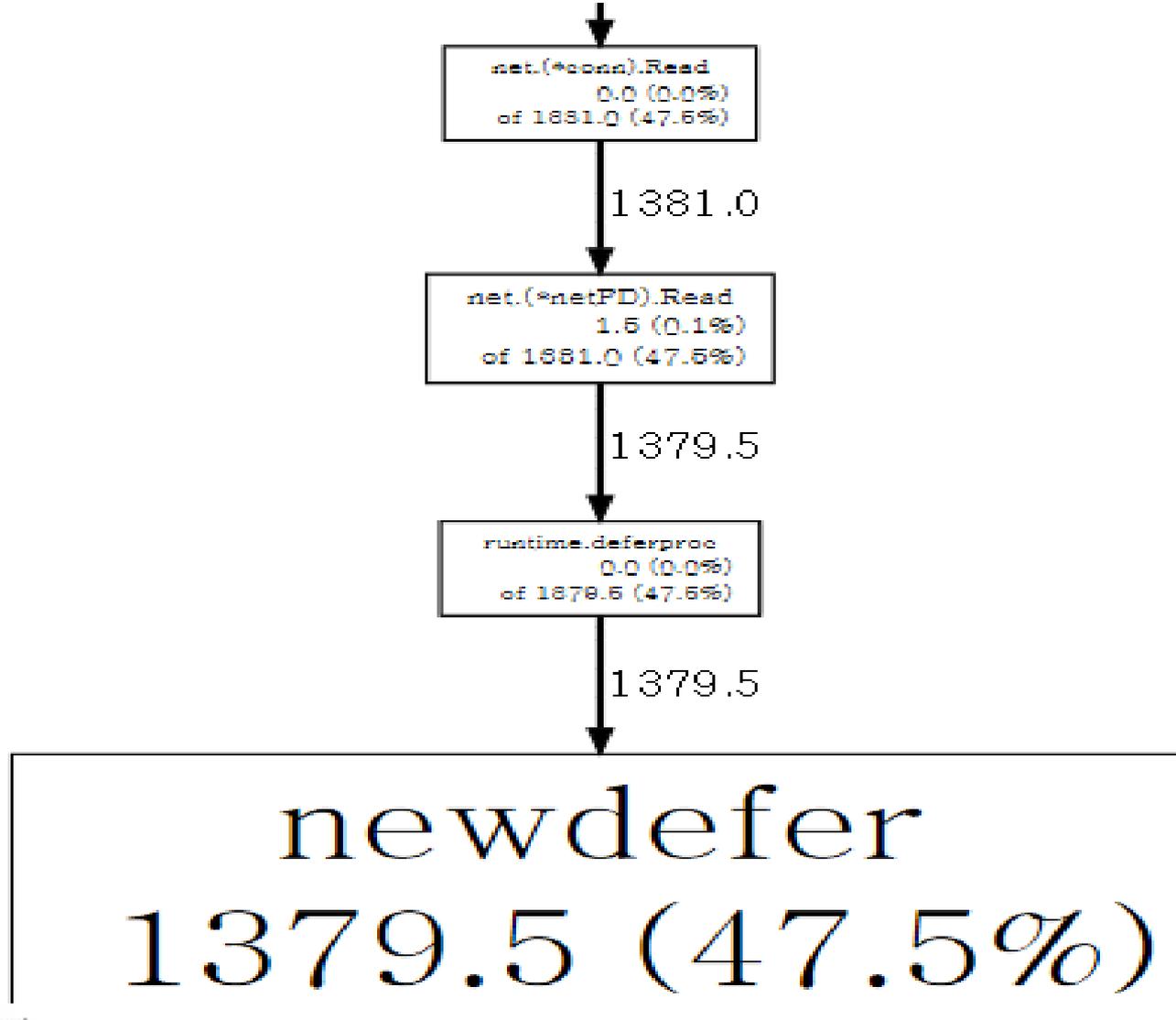
关于json

- 解决方案: magajson
- github.com/benbjohnson/megajson
- Code generation, No reflection, no defer

标准库优化

- 网络库中读写数据都需要lock, unlock是在defer中完成的, 在长连接的程序中, 大部分时间是idle的, 造成defer分配的内存长时间驻留内存, 使得内存压力较大, 也增加了gc的压力
- 看个热点图(有图有真相😊)

网络库热点图



标准库优化(network)

- 解决方案:
- 去defer unlock, 改为手动在合适的地方unlock

Group commit

- IO通常较慢，压力测试下消息队列的速度是瓶颈，通过golang的buffered channel，合并作为一个大消息提交
- 通过buffered channel合并消息给客户端，减少网卡中断次数

其它

- errors.New 优化
- time 存储
- socket close, runtime.SetFinalizer

Group commit

•代码示例:

```
case msg, ok := <-self.inbox:
    if !ok {
        self.cleanUp()
        return
    }

    if self.PostWrite(msg) != nil {
        self.cleanUp()
        return
    }

    min := util.MinInt(len(self.inbox), 10)
    for i := 0; i < min; i++ {
        if self.PostWrite(<-self.inbox) != nil {
            self.cleanUp()
            return
        }
    }

    if self.FlushWriter() != nil {
        self.cleanUp()
        return
    }
}
```

Dump信息获取

- Bug不可避免, Crash不可避免
- Golang程序crash时会打出调用栈, 硬编码了写入的fd(2), 需要重定向到自定义的文件存储
- `syscall.Dup2(int(crashFile.Fd()), 2)`

小结

- 每个连接2个goroutine，一读写
- 关键代码没有defer，整个server不使用defer
- 16个gc线程和8个gc线程差别不大，肉眼观察提升不到1/5
- 整体优化效果：内存占用减少一半，gc时间减少一半左右

小结

- 还有优化的余地，细节改进
- Golang的gc能力不能根据cpu核数线性扩充，多进程拆分效果显著，gc时间至少可以再缩短1/3，但管理更多的进程也麻烦
- 今年的目标：单机连接数达到180w左右，gc时间缩短到1.3秒左右

云存储

- 商品订单
每年500TB
- 库房流水记录
每年超过1PB
- 商品图片
近百TB，持续增长

元数据存储引擎

- key-value, memcache类似的cas指令, 支持ListBucket操作(参考amazon s3)
- 持久化字典树(Adaptive Radix Tree)实现, key和value分离, 尽量让字典树常驻内存, 快速索引, value的缓存交给应用决定
- 原型由golang实现
- 小文件(1k以内)直接存储到元数据引擎中

元数据存储引擎

- 再说缓存
- 私有云：用户存储数据量非常大，每天几百万到千万，数据特点明显，新的数据较热(一周内)，因此缓存策略在业务层可以较好处理(LRU, FIFO ...)
- 内存是存储引擎宝贵的资源，能省则省，好钢花在刀刃上

元数据存储引擎

- 如果字典树在内存放不下呢？好像只能剪枝了
- 剪枝策略：这是个麻烦的事情...

云存储分布式模块简介

- Replication, split, migration均由golang实现
- Split按key做切分，迁移算法类似couchbase，但更复杂些，因为迁移后，ListBucket要merge
- ListBucket时从多个机器取得数据后做合并操作
- 高可用：强一致，主从同步复制，standby提供暂时性容错

- Thank You!

