

毛剑

# GO在猎豹移动的应用

# Why Go?

- ◎ 优雅简洁，少就是多；
- ◎ 性能好、系统级语言；
- ◎ 静态语言、强类型约束；
- ◎ 交叉编译&部署；
- ◎ 网络模型&并发同步模型；
- ◎ 标准库、内置工具强大支持；
- ◎ 开源&社区活跃；

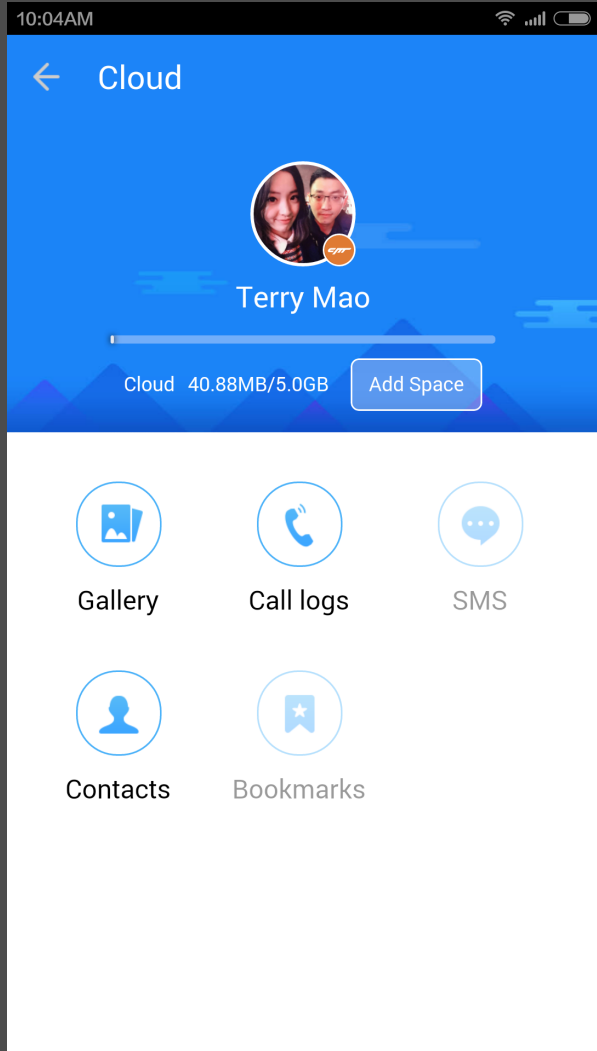
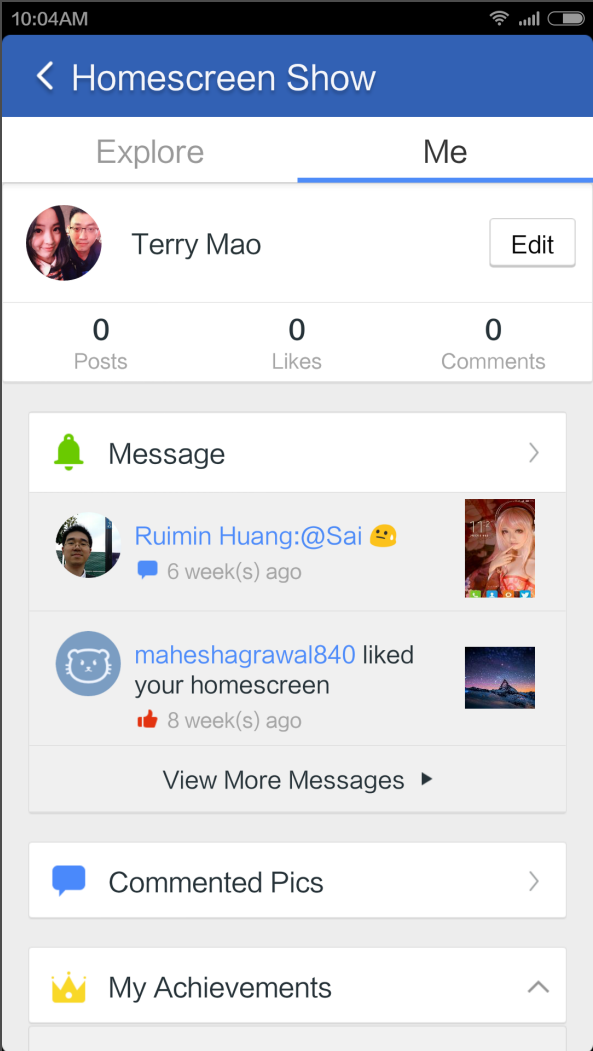
# 我们做了啥？

## ◎ 业务

- 猎豹移动全球passport体系；
- 游戏开放平台；
- 游戏支付体系；

## ◎ 平台

- 基于gopush的推送平台&goim；
- 基于redis sentinel的smart client；
- rpc框架；
- gosnowflake发号器集群；
- goconf统一配置管理；



接入层

Lvs VIP 四层协议转发

Nginx Upstream

ELB 七层协议转发

应用层

业务 API

前端 PHP

服务层

Go Service

缓存层

Redis

存储层

MySQL

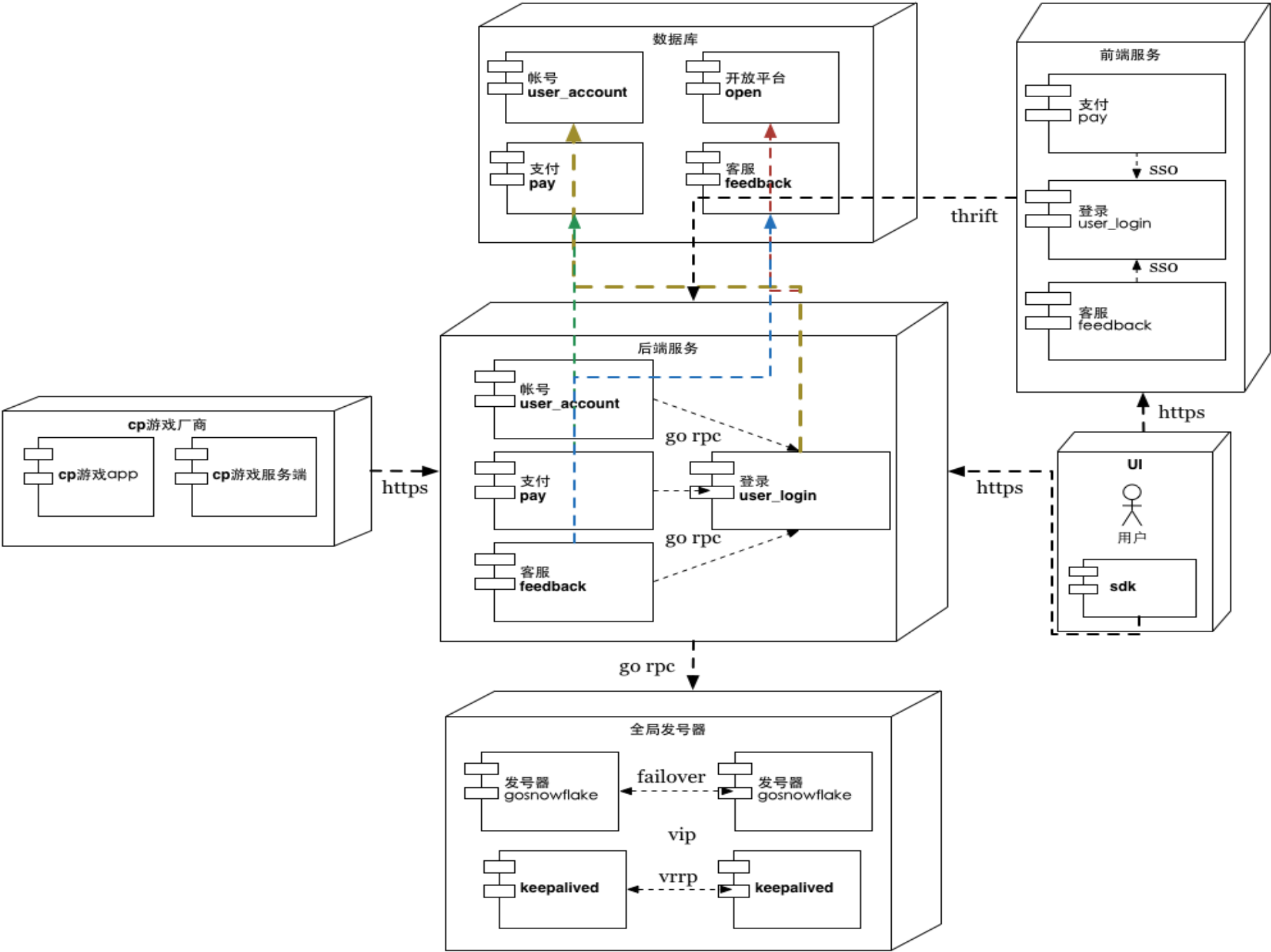
S3 / Gluster FS

# 接入层优化

- ◎ DNS在移动网络下不适用；
  - 避免劫持、失效，dns提供商故障；
- ◎ 协议压缩：pb+gzip；
  - 节约流量；
- ◎ 协议设计：职责单一不适用；
  - 合并请求；
- ◎ TCP Handshake影响RTT；
  - keepalived&长连接；
- ◎ API动态加速；
  - proxy模式&动态CDN；

# SOA

- ⊙ Web站点是PHP开发的，通过RPC交互；
- ⊙ rpc和api都是基于Go开发的服务；
- ⊙ 国内DNS->VIP->lvs->tengine->Go；
- ⊙ 海外DNS->ELB DNS->ELB->Go；
- ⊙ Service之间通过Thrift或者net/rpc通讯；
- ⊙ 依赖redis sdk访问redis；
- ⊙ 直连mysql；
- ⊙ 用户的所有业务逻辑基于Go处理；





# Service

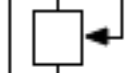
- ⦿ 服务应该是无状态的；
- ⦿ api服务出现瓶颈的时候，直接scale out；
- ⦿ graceful restart依赖健康检测；
- ⦿ api质量监控，使用日志来追踪，通过本地日志+flume+hdfs+hive；
- ⦿ 实时监控可以考虑flume sink到kafka，再依赖Spark计算；

运维

接入层

接入层监控检测

切出流量



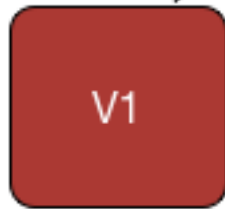
部署服务

冒烟测试



部署成功

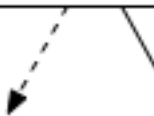
切入流量



V1

V2

接入层监控检测

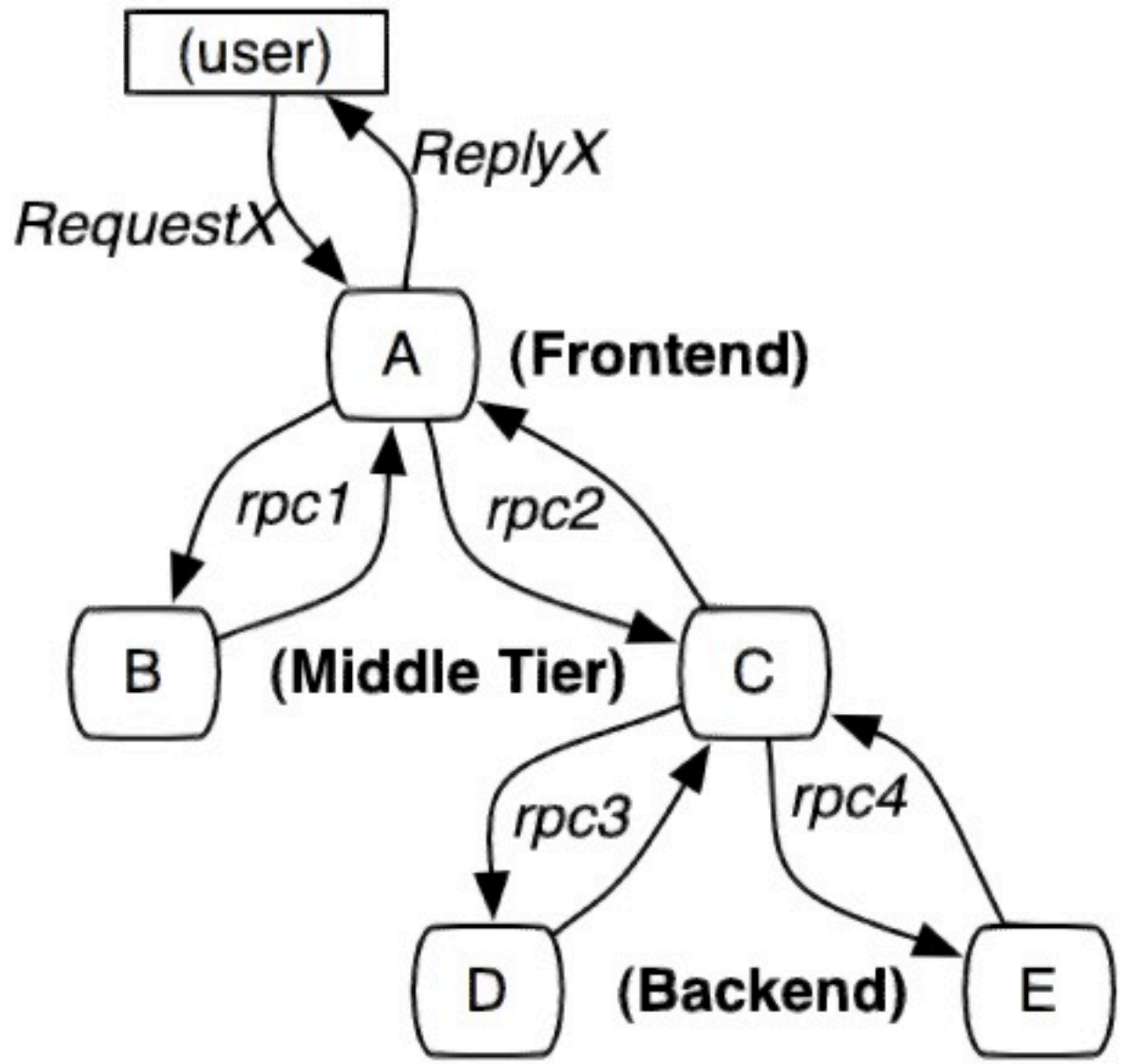


# RPC

- ◎ 协议&远程调用的选型；
  - net/rpc, thrift, grpc等；
- ◎ 链路追踪，参考Google Dapper论文，核心思路是关键库植入代码，因为缺乏AOP编程支持，我们使用golang blog推荐的依赖context对象；
- ◎ 服务发现、负载均衡依赖ZK；
- ◎ 弹性调度的支持，降级处理、动态扩容；

# RPC选型

- ◎ 是否多语言支持？ net/rpc不支持
- ◎ 性能如何？
- ◎ thrift num:111324, time:30s, num/s:3710;
- ◎ grpc num:159999, time:30s, num/s:5333;
- ◎ net/rpc不依赖context, 实现数据跟踪, 需要修改源码;
- ◎ grpc支持http2, 方便移动端app使用;



```
func (client *Client) go2(ctx context.Context, serviceMethod string, args
call := new(Call)
call.ServiceMethod = serviceMethod
call.Args = args
call.Reply = reply
if done == nil {
    done = make(chan *Call, 10) // buffered.
} else {
    // If caller passes done != nil, it must arrange that
    // done has enough buffer for the number of simultaneous
    // RPCs that will be using that channel. If the channel
    // is totally unbuffered, it's best not to run at all.
    if cap(done) == 0 {
        log.Panic("rpc: done channel is unbuffered")
    }
}
call.Done = done
tr := trace.FromContext(ctx)
if tr != nil {
    // If the request already has trace information attached to it,
    // the service should use that information as server receive and
    // send events are part of the same span as the client send and c
    tr.Link()
} else {
    // For the initial receipt of a request no trace information exist
    // So we create a trace id and span id. These should be 64 random
    // The span id can be the same as the trace id.
    tr = trace.NewTrace()
    ctx = trace.SaveContext(ctx, tr)
}
call.Trace = tr
// cs - Client Start. The client has made the request.
// This sets the beginning of the span.
if client.tracer != nil {
    client.tracer.Record(tr, serviceMethod, trace.EventCS)
}
client.send(call)
return ctx, tr, call
}
```

```
// Call2 invokes the named function (with ctx), wait
func (client *Client) Call2(ctx context.Context, s
var err error
ctx, tr, call := client.go2(ctx, serviceMethod)
select {
case <-call.Done:
    err = call.Error
case <-ctx.Done():
    err = ctx.Err()
}
// cr - Client Recieve.
// The client has received the response from the
// This sets the end of the span. The RPC is complete.
if client.tracer != nil {
    client.tracer.Record(tr, serviceMethod, trace.EventCR)
}
return ctx, err
}
```

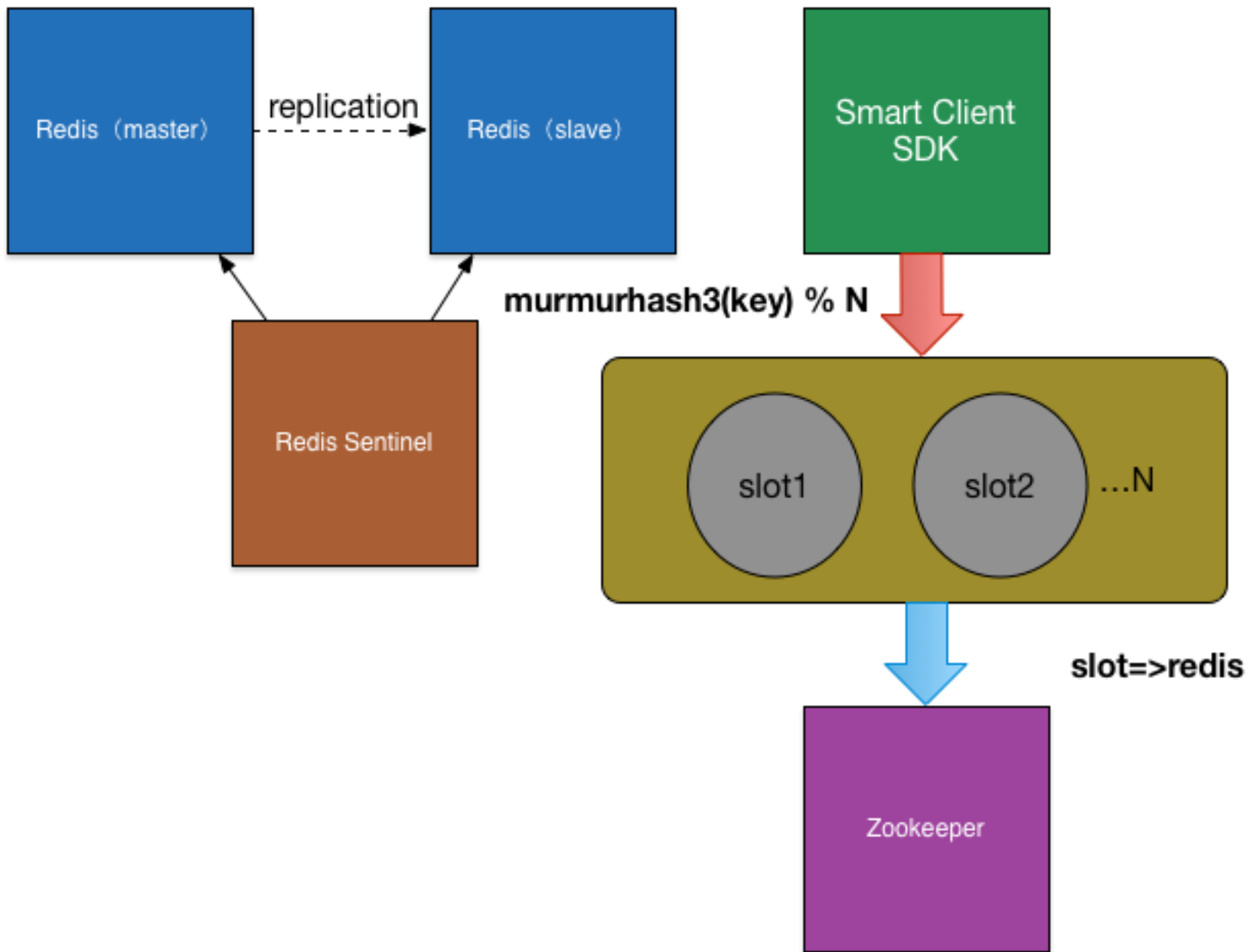
```
func (s *testServer) EmptyCall(ctx context.Context, in *testpb.Empty) {
    return new(testpb.Empty), nil
}

func (s *testServer) UnaryCall(ctx context.Context, in *testpb.SimpleRequest) {
    return &testpb.SimpleResponse{
        Payload: newPayload(in.GetResponse(), in),
    }, nil
}
```

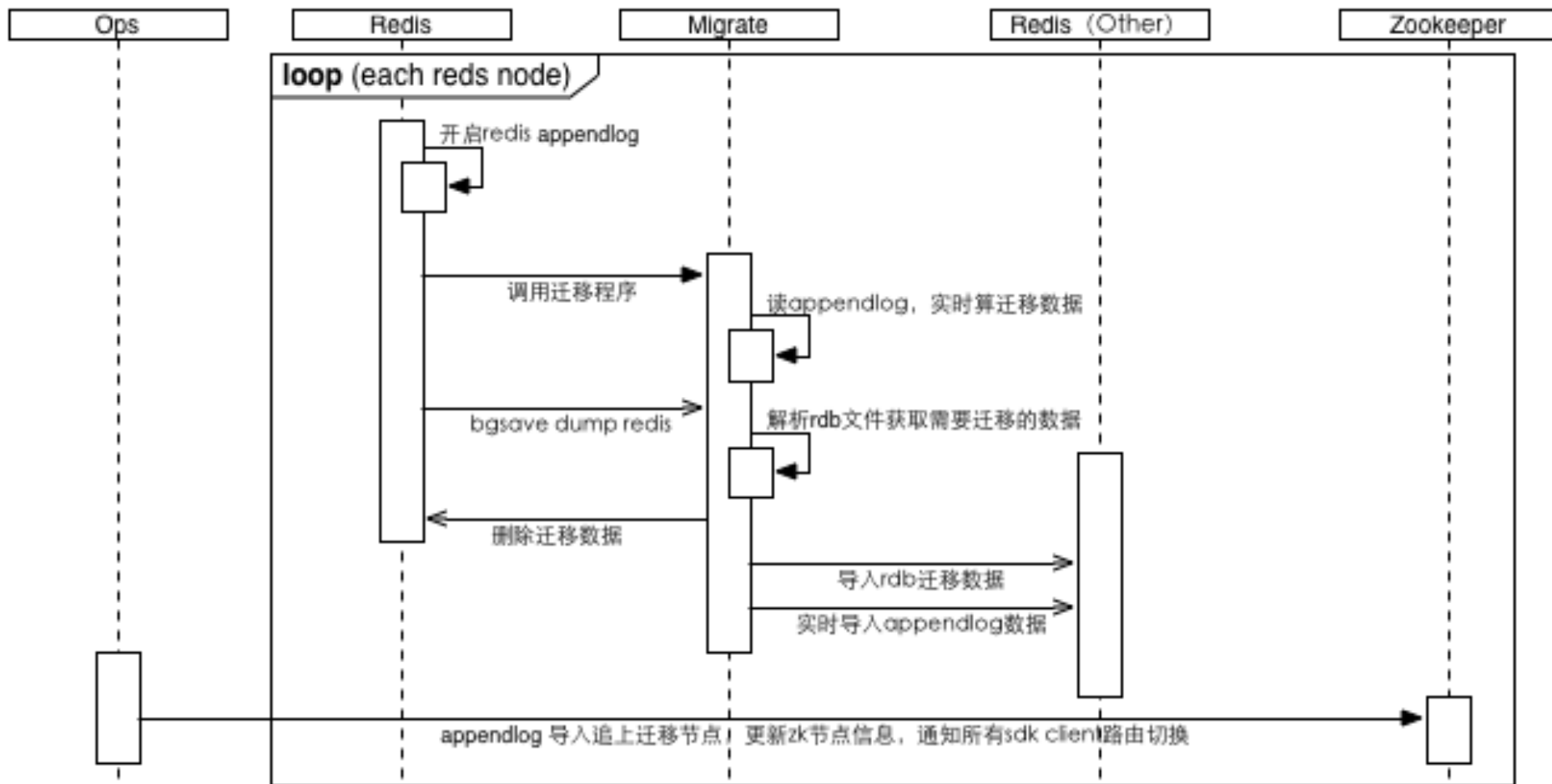


# Cache

- ◎ 模仿cpu使用多级cache；
- ◎ L1 cache：不经常修改，大量访问对性能要求极致的，我们使用go map缓存信息，使用COW保证无锁更新和访问；
- ◎ 使用redis作为核心的cache store；
- ◎ 使用hash mod region方式对cache进行扩展；





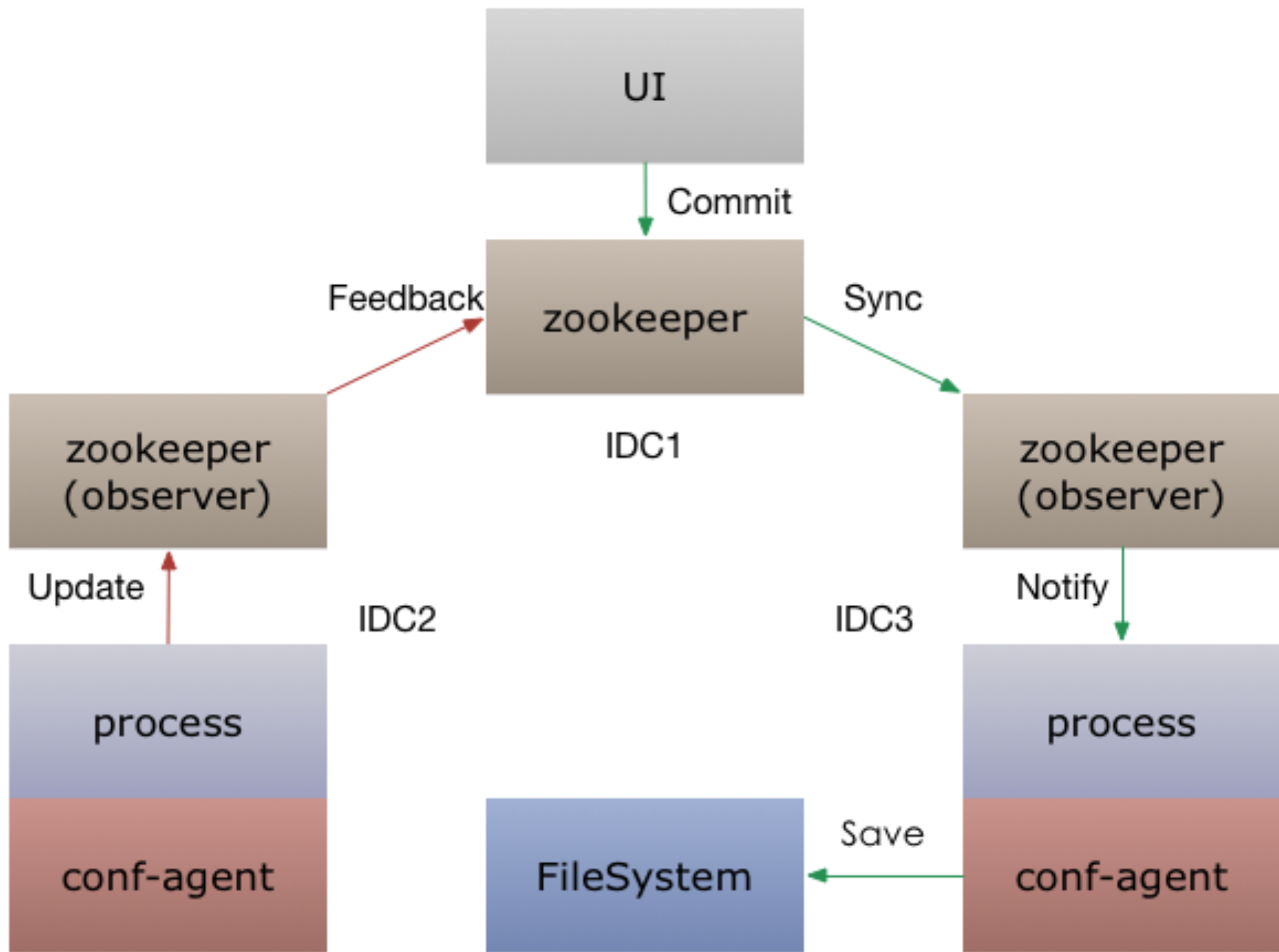


# goconf

- ◎ xml, yaml, json, ini?
- ◎ 阶段1: 逐idc, 逐机器配置修改;
- ◎ 阶段2: svn统一提交修改, 每个idc一份;
- ◎ 阶段3: 配置统一管理化 (agent模型);
- ◎ 一处修改, 统一管理;
- ◎ 节点状态查看、回滚配置;
- ◎ 数据安全、强一致性;

# goconf




```
/config/  
  | service/  
    | idc1/  
      | current/ ( {"last_ver":"v1.2", "cur_ver": "v1.1"})  
        | section/  
          | key-value ( {"value":"xxxx", "comment":"x"})  
        | snapshot/  
          | v1.1/ (value是整个配置文件)  
        | agent/  
          | node/ ( {"ver":"v1.2"})
```



# 配置管理

## 服务管理

[+添加](#)

名称	介绍	操作
a1	server a1	  




## IDC管理

[+添加](#)

名称	介绍	操作
xxdy1	ok	  




## 版本管理

[+添加](#)

名称	介绍	操作
edit	可编辑	  

## 模块管理

[+添加](#)

名称	介绍	操作
mod1	description	  

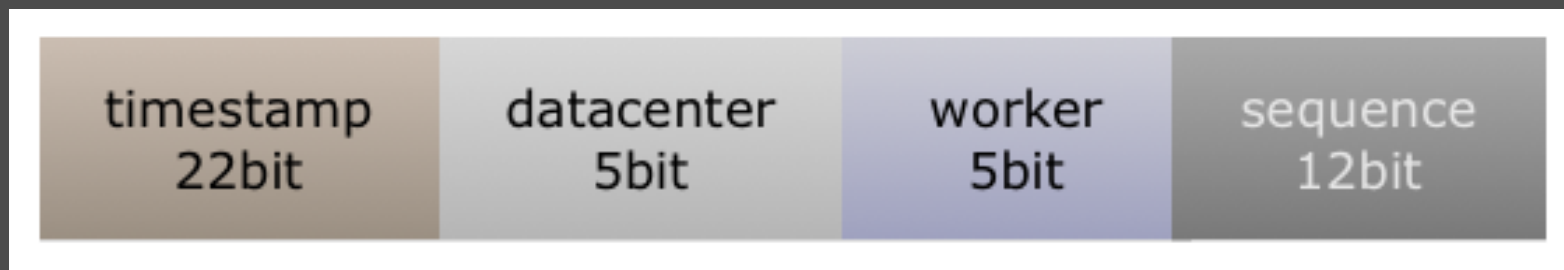
## 键值管理

[+添加](#)

Key	Value	操作
ip	127.1.2.2	 
data_source	mysql://122.22.22.1	 
mod2	description	  
running	运行状态	  

# gosnowflake

- ◎ 参考twitter snowflake id算法实现；
- ◎ 支持datacenter id & worker id（32个）；
- ◎ 同一个毫秒支持4096的sequence滚动；
- ◎ golang重新实现；
- ◎ 支持net/rpc级别的failover；



# 采坑&建议

- ◎ defer内存占用；
- ◎ database/sql 连接一定要设置idle conn；
- ◎ database/sql prepare bug；
- ◎ gc带来的stop the world；
- ◎ := 操作符，以及变量作用域；
- ◎ setuid setgid无效；
- ◎ map, slice预分配；
- ◎ ....

谢谢

QA