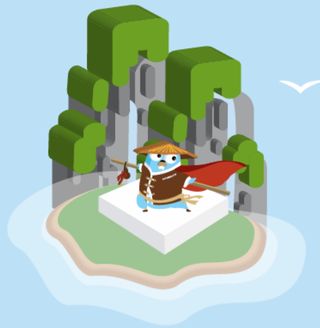
GopherChina2018



Golang在阿里巴巴调度系统Sigma中实践交流

阿里巴巴-系统软件事业部
调度系统-鹰缘 20180415

大纲

取材
-资源调度领域Sigma
-选“共性的、具备借鉴性”(阿里特有的不讲)
-**更多的时间留在QA环节**

工程问题
-架构设计与语言选择
-业务并发模式下的任务粒度和锁处理
-综合解决方案，以Slave架构引出

工程背后的那些事
-围绕规模化，上云，运维友好
-踩过的坑，大规模场景下解决方案的一种实现
-**最难调的bug 往往是 低级错误引起的**

- Sigma总览
- 4个案例+4段代码
- 小结

Sigma总览-> 业务+规模

(外)ECP企业级容器平台(EDAS+公有云)
(内)全集团BU 100%(天猫、淘宝、菜鸟、高德、优酷等)

全集团Pouch化100%(包括DB、中间件、搜索等)
百万级容器实例(1%发布也就是万级实例)

自顶向下

Zeus Lark Normandy


Sigma-ApiServer

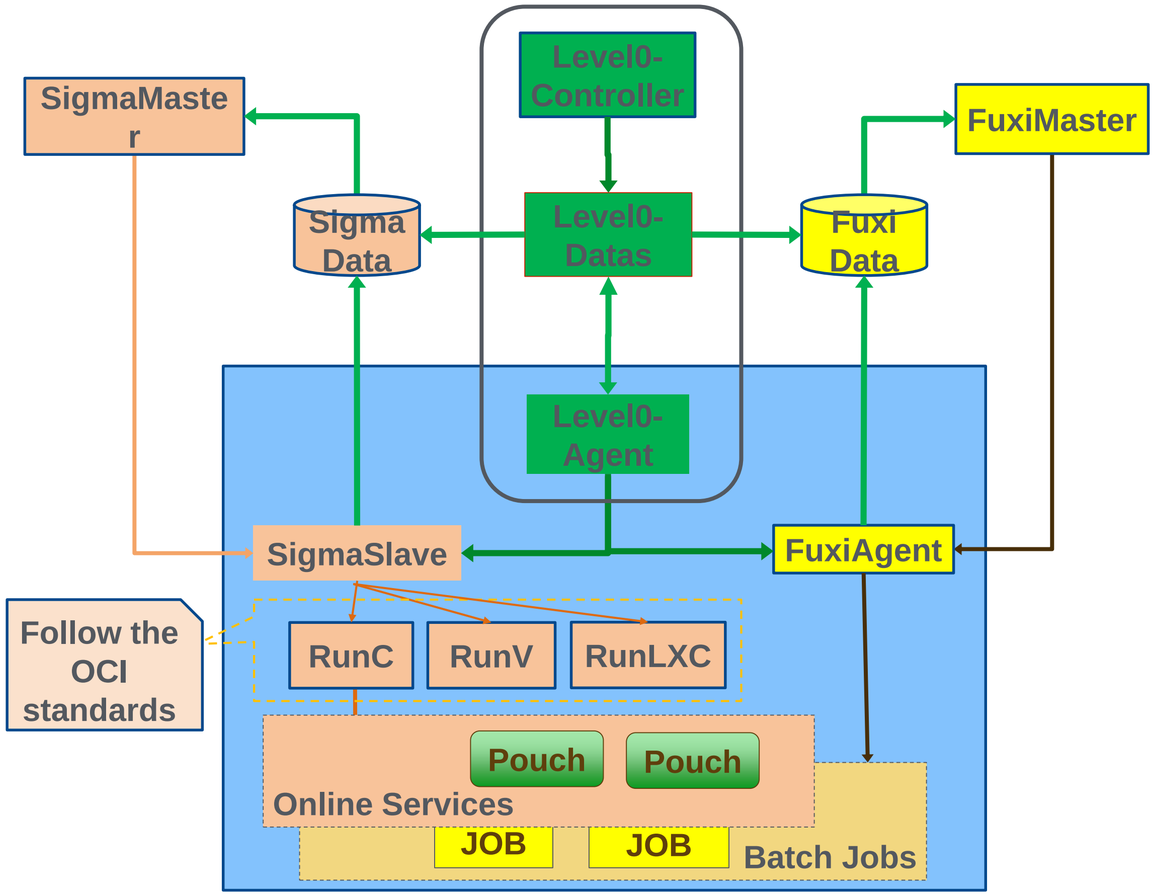
Sigma-Etcd


Sigma-Master


Sigma-Slave


Pouch

Sigma总览-> Sigma架构



存在即合理，但不是最佳

抽象出

4个案例

Sigma APIServer

Sigma Scheduler

Sigma Slave

PouchContainer

案例1：APIServer -> 思路 -> 架构设计与语言选择

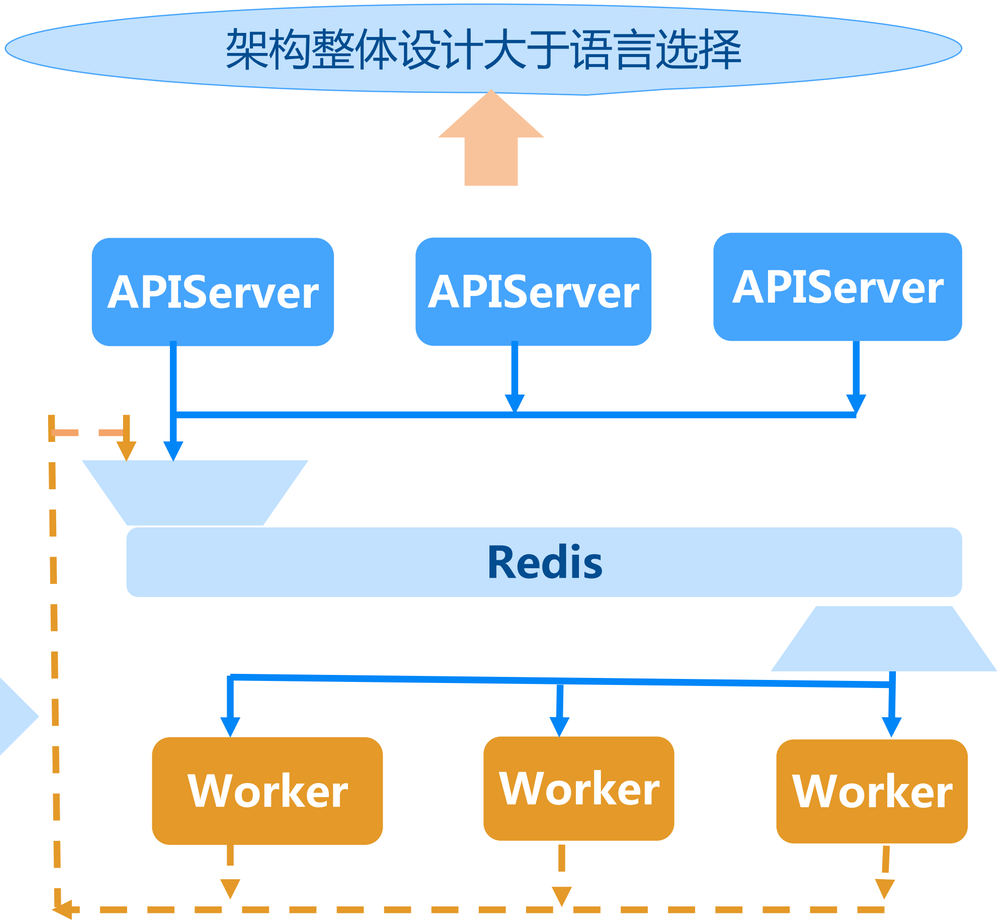
背后的事

需求：发布、扩容、销毁、启停、升级、云化(混合云)
规模：总应用 十万级，活跃万级，交易核心千级，总实例百万级

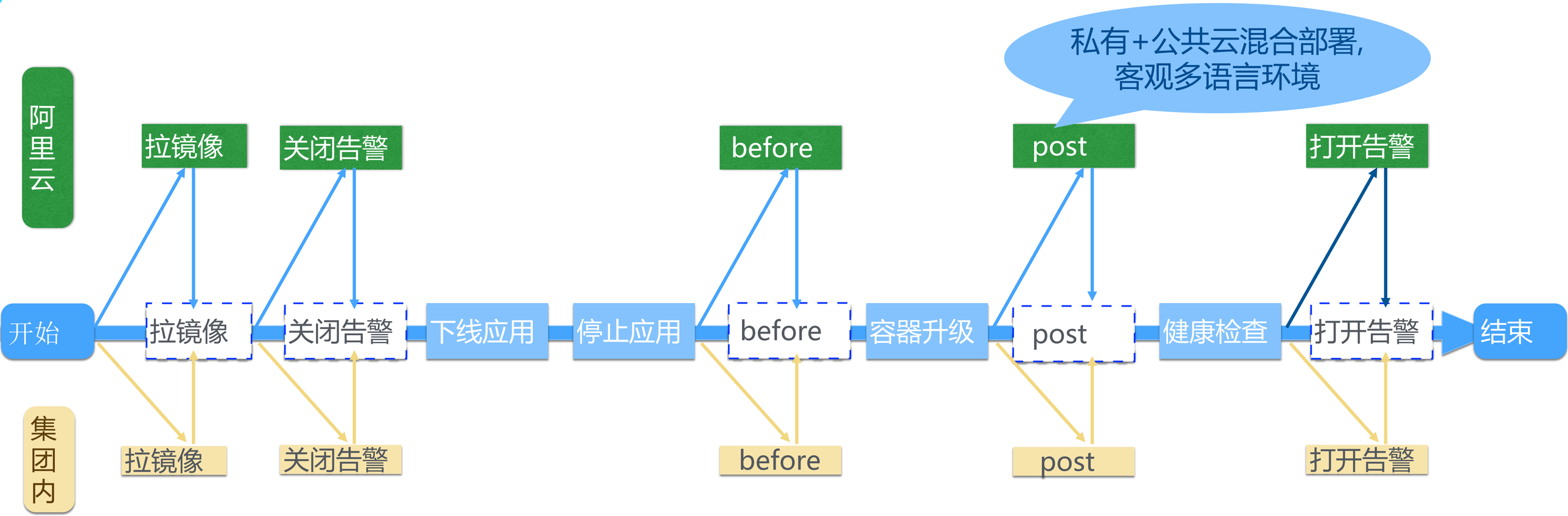
规模引发的问题
运维友好 高可用 一致性

解决思路

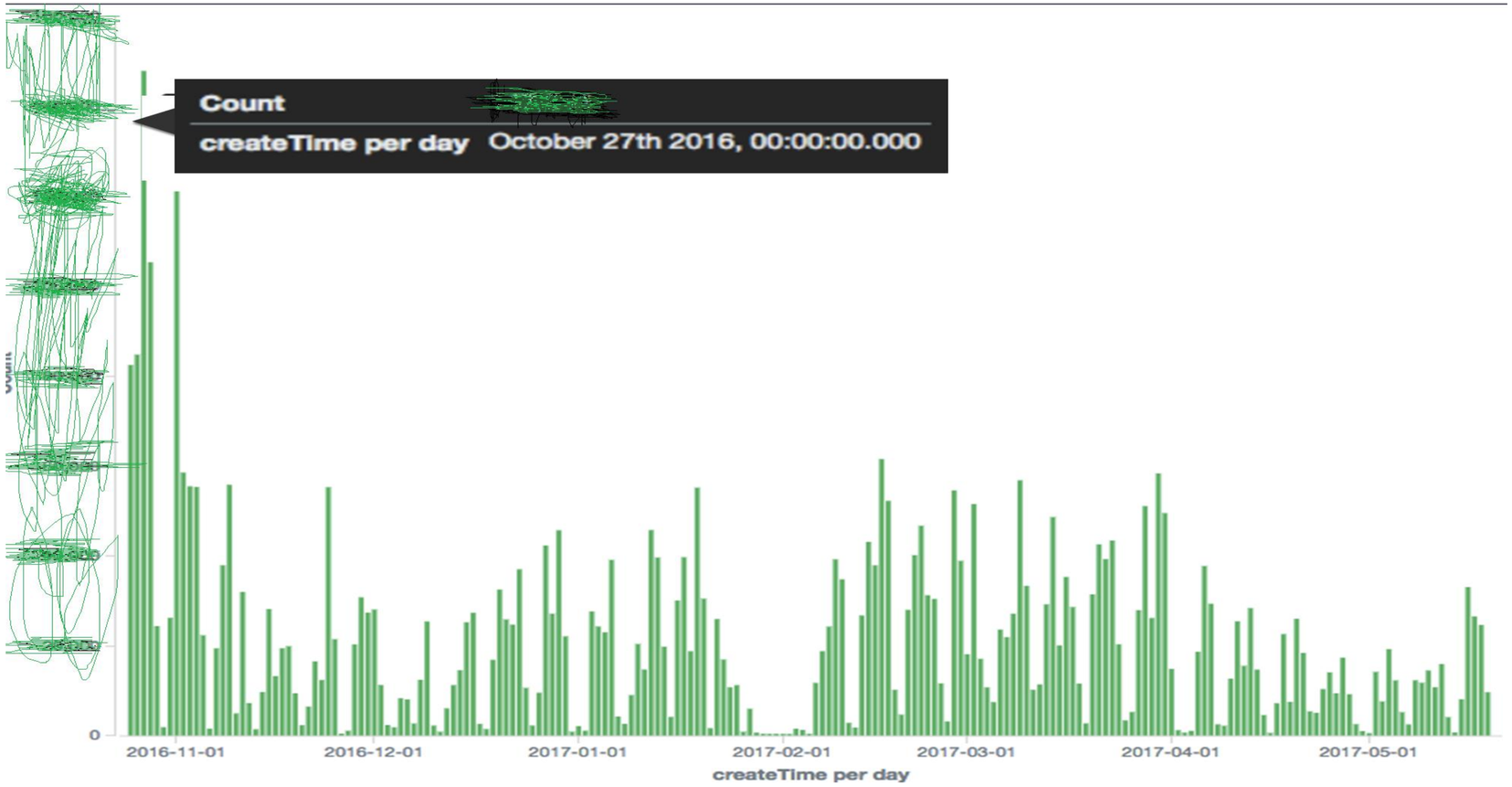
(1)数据一致性：etcd/redis 实时+全量
(2)状态一致性问题 转为 存储一致性问题
(3)简单的-够用就好
(4)高可用-无状态(master|slave、stateless、fast failover)
(5)降级-抢占
(6)内外兼容：一个团队两块牌子



案例1：APIServer -> 场景化解释 -> 架构设计优先语言选择



案例1：APIServer -> 架构设计与语言选择 –> 数据表现



案例2：Scheduler -> 任务并发粒度与锁 ->并发模式1

背后的事

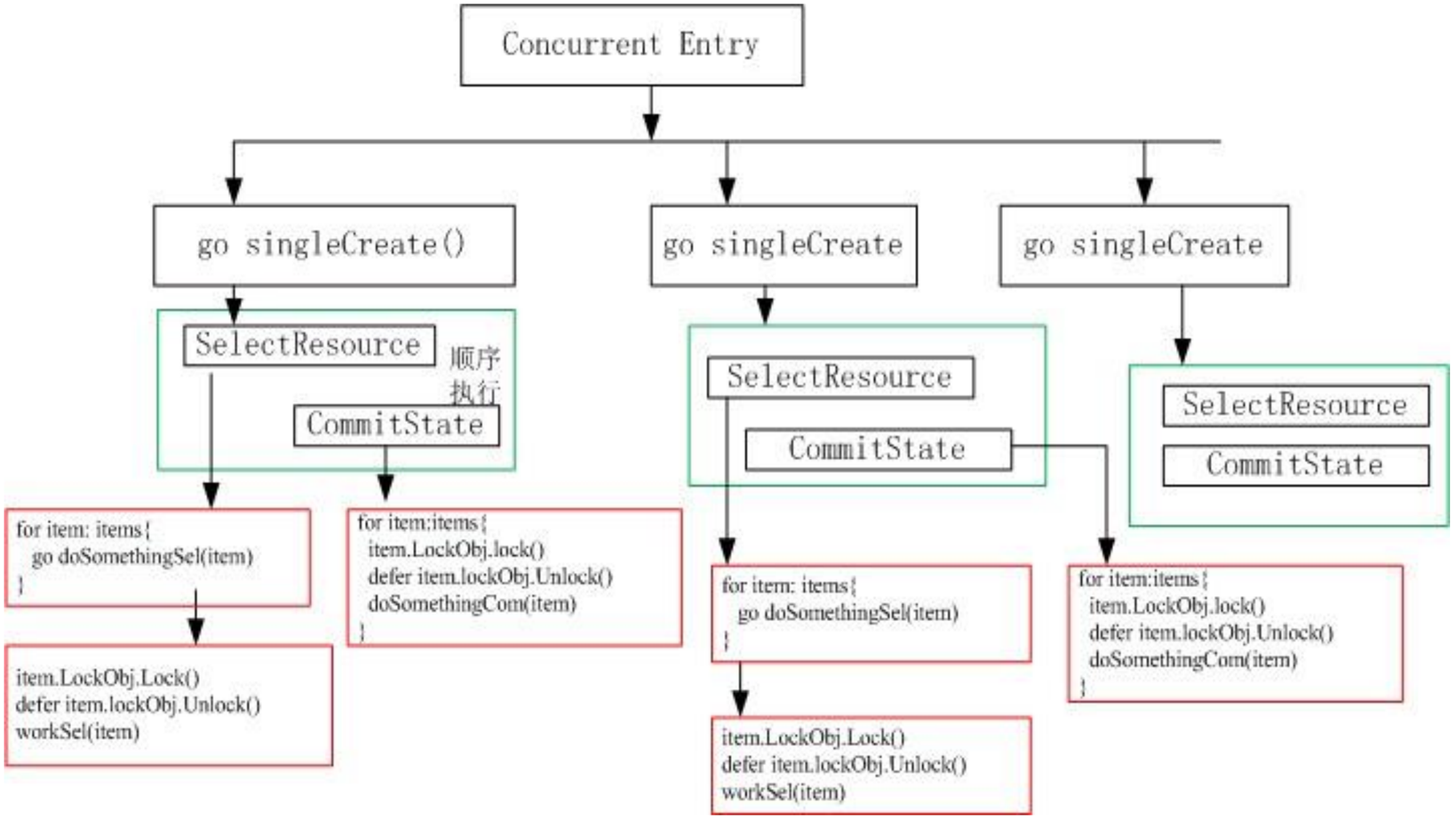
-上万结点中筛选

-链式流

FilterChain

WeightChain

CreatePouch Container

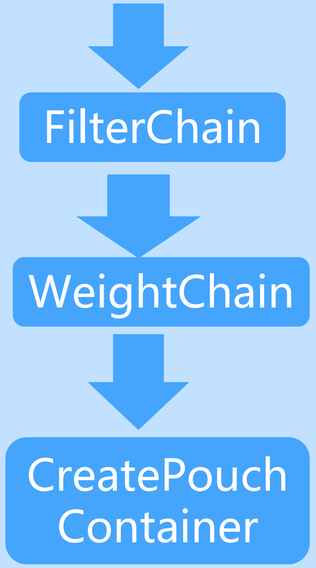


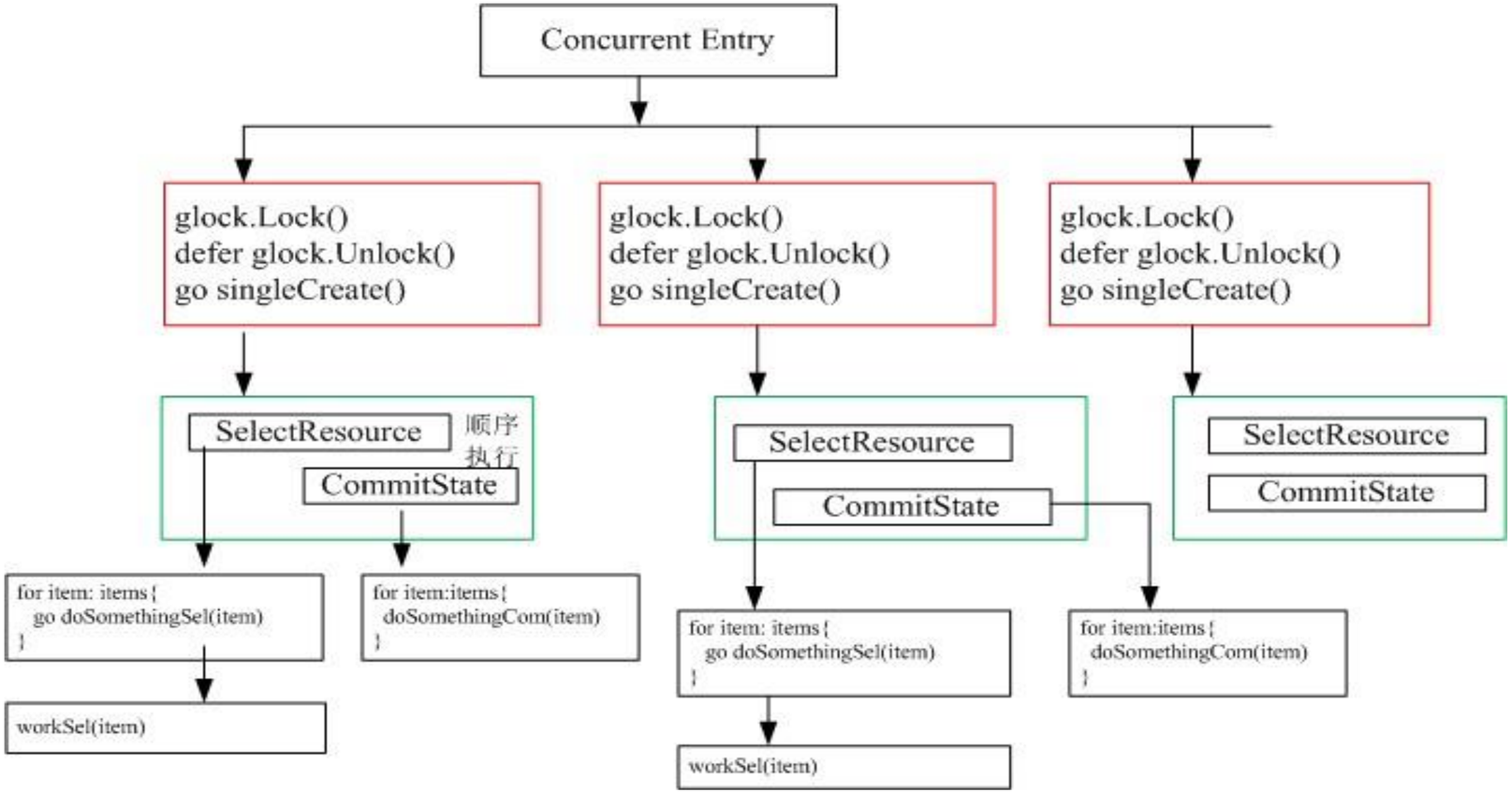
案例2：Scheduler -> 任务并发粒度与锁 ->并发模式2

背后的事

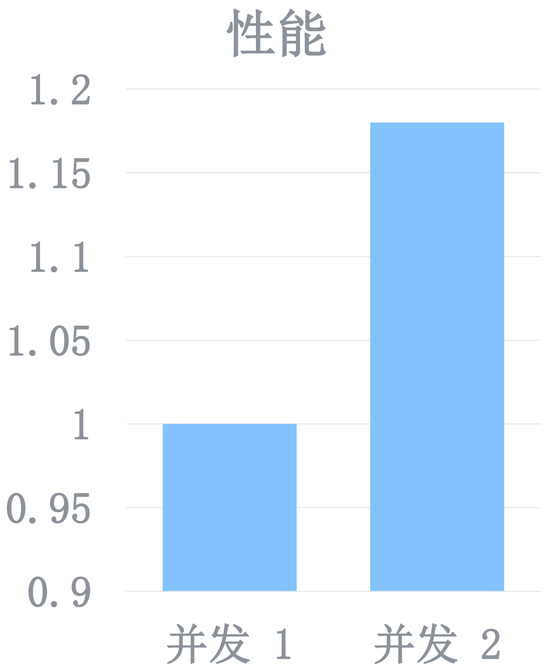
-上万结点中筛选

-链式流





案例2：Scheduler -> 任务并发粒度与锁 – 总结



目标：简单-高性能
第二种：受顶层并发数NC数量、每次并发计算内部的并发过滤维度影响。
顶层并发多、内部并发量大，性能不会太差
性能瓶颈：内部并发的开销，内部开销大，上层lock影响弱化。

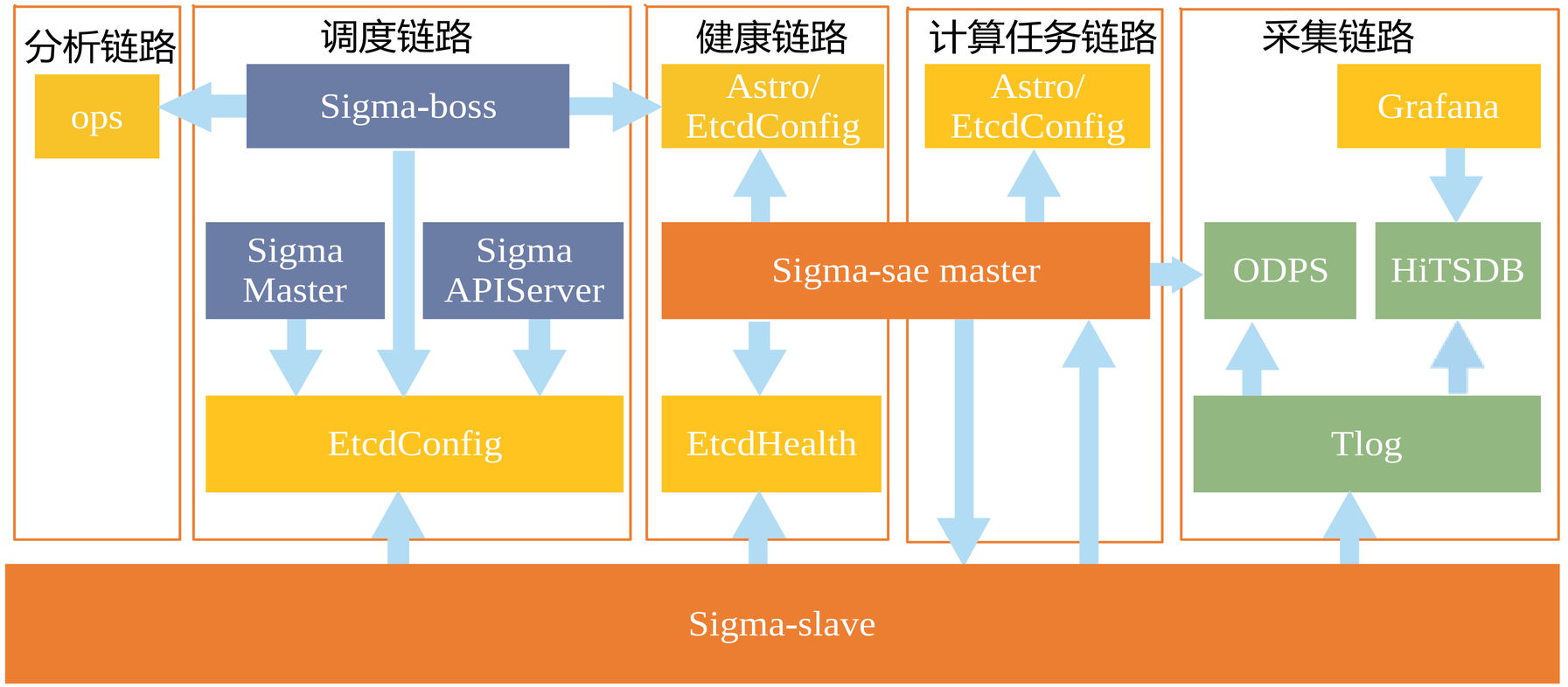
执行：利用语言特征
并发对共享资源的读写，在golang场景下，只能是第二种结构

Why
非channel、共享变量在一个协程内修改了，
另外一个协程不会立即感知修改后的值

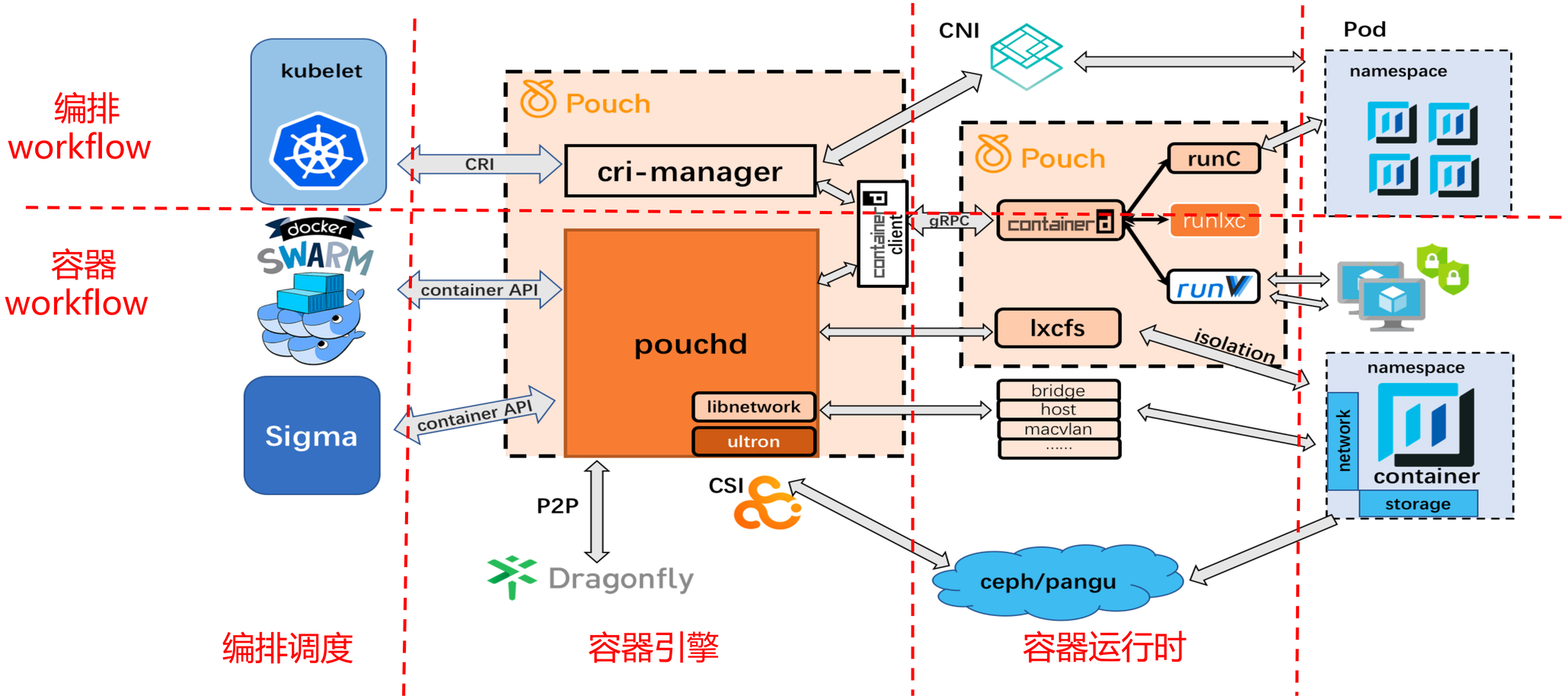
案例3：SigmaSlave-综合解决方案

背后的事
2015年初就引入Golang，当时还不是特别火
阿里主流语言Java
快速迭代打磨

Sigma-sae 架构



案例4：Sigma-PouchContainer-生态架构



https://github.com/alibaba/pouch/blob/master/docs/architecture.md

案例4：Sigma PouchContainer -> Features

Features

- Rich container
- Strong isolation
- P2P distribution
- Kernel compatibility
- Standard compatibility

https://github.com/alibaba/pouch

背后的故事

- 容器代表Docker之前是什么？ 虚拟机-XEN
- 容器编排k8s之前是什么？devOps？人肉普遍？
- 隔离？安全，排查问题，黑盒子，规模化，混部
- *互联网思维？发布快+规模化+高频发布迭代->image加速*
- 初创公司业务往往比技术跑的更快 ->2.6 2011,4.10 2018
- 社区往往比企业内部跑的更快->历史包袱
- *运维友好，pouch from 2011-> k8s*

代码1：Golang map循环陷阱:指针变量误用

低级错误

例如 for key,value:=range map_obj, 不可以将&value 作为对象返回函数体外。

value地址一直不变，**&value取**的value地址不变，而这个地址指向的内容是循环最后一个元素内容。

https://github.com/sebarzi/gopherchain2018/blob/master/map_loop_bug.go

https://github.com/golang/go/wiki/CommonMistakes#using-goroutines-on-loop-iterator-variables

代码2：Golang 异步对象序列化Map并发冲突

问题背景

seelog 异步化日志，用于追踪、回放、分析等 --> 规模引发

冲突现象

fatal error: concurrent map read and map write

触发条件

异步对象序列化，对象里面如果有map，且使用json.marshal to string,其他方式对map 执行了read、write操作。
本质原因在于：两个goroute对同一个map执行了读写并发，而golang map默认是不支持并发操作、没有加锁

冲突解决

(1)提前转String：在主业务逻辑里面，提前把对象转为string，丢给异步任务去执行
（不局限写log，可以是其他对map的操作）
(2)Map读写加锁：读写加锁对性能的影响，以及死锁等又带来新的麻烦。
关于优化并发map的可以参考这个分析，比较全面：https://misfra.me/optimizing-concurrent-map-access-in-go/
(3)交互参数改为string，或者对象的深clone，也就是完整的copy对象值，重造对象，而不是直接引用指针
https://github.com/sebarzi/gopherchain2018/blob/master/asy_map_conflict_bug.go

代码3：Goroute泄露&超时控制

问题描述：

主任务起 异步子任务，子任务执行循环任务，且有超时控制 -> **规模引发**

当子任务提前完成任务，子任务提前结束，主任务继续

当子任务超时,主任务通知子任务退出，不再继续执行

应用案例：

大促建站调度阿里云ECS SDK 进行**批量资源创建**。ECS异步的原子接口:创建、启动、销毁ECS等

创建ECS：

（1）**发起创建请求**，执行实例创建

（2）执行**starting操作**

（3）**轮询启动状态**

（4）启动成功的，资源**申请成功**。启动不成功或者超时时间窗口内，**还在启动中**，都做超时处理

（5）超时资源**清理**，避免资源泄露

一种解法：

主任务启动异步循环子任务；子任务内执行LoopCheck；check完毕，正常退出子任务

主任务设定超时时间，子任务还没返回，执行超时退出，并**通知子任务**退出循环

陷阱

1. 如主任务超时直接退出，不管子任务。从计算结果看，没有问题。但资源泄露
2. 如子任务自己控制，那么子任务需同时保留：超时前任务处理结果、超时退出未被处理的结果，用于主任务决策
3. 循环执行的子任务，写入管道和具体任务的位置，

select部分：default 具体任务执行部分，case 通知管道，如果反了，就不行

https://github.com/sebarzi/gopherchain2018/blob/master/timeout_goroute_demo.go

代码4：Pod打散算法实现

背后的事

高可用性角度，避免扎堆

(高密度部署、故障常态化事情、用户体验损失最小、快速恢复)

算法描述

从一堆候选服务器列表中，按照pod打散、服务器得分，选择一批相对靠谱的服务器

算法实现

按pod将已得分的候选服务器分类，在同一pod下面的服务器，按得分从高到底排序

循环对每个pod的得分队列按队列头元素得分排序，从高到低，取头部元素，加入新的候选列表

直到所有pod的元素全部加入候选列表。

算法延伸

算法演变下，运用另外一种场景：多队列求并或者求交集

https://github.com/sebarzi/gopherchain2018/blob/master/pod_spread.go

举例：10个结点，结点如下，

{nc:a,score:1.0,pod:1}
{nc:b,score:1.0,pod:2}
{nc:c,score:4.0,pod:3}
{nc:d,score:3.0,pod:2}
{nc:e,score:4.0,pod:5}
{nc:f,score:2.0,pod:4}
{nc:g,score:2.0,pod:3}
{nc:h,score:2.0,pod:1}
{nc:i,score:4.0,pod:3}
{nc:j,score:3.0,pod:2}

top5：输出如下：

{nc:e,score:4.0,pod:5}
{nc:c,score:4.0,pod:3}
{nc:b,score:3.0,pod:2}
{nc:a,score:2.0,pod:1}
{nc:e,score:2.0,pod:4}

top4：输出如下：

{nc:e,score:4.0,pod:5}
{nc:c,score:4.0,pod:3}
{nc:b,score:3.0,pod:2}
{nc:a,score:2.0,pod:1}

或者 任意一个输出都算有效

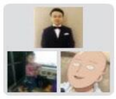
{nc:e,score:4.0,pod:5}
{nc:c,score:4.0,pod:3}
{nc:b,score:3.0,pod:2}
{nc:e,score:2.0,pod:4}

小结

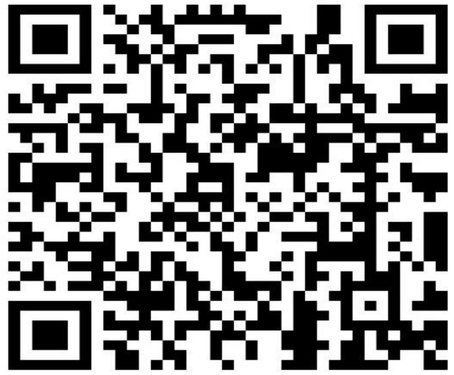
案例和代码分享背后共同的线索
--> **规模化场景**

设计层面：整体架构设计优先语言选择
性能层面：任务粒度选择
数据驱动：状态的一致性 转移 为存储一致性
语言理解: Map异步序列化，Map循环指针
多层并发：可控超时
调度算法：规模下Pod打散
PouchContainer：拥抱开源、回馈社区

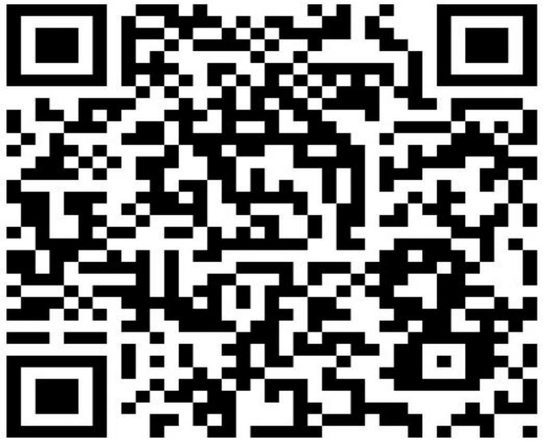
The End -Thanks



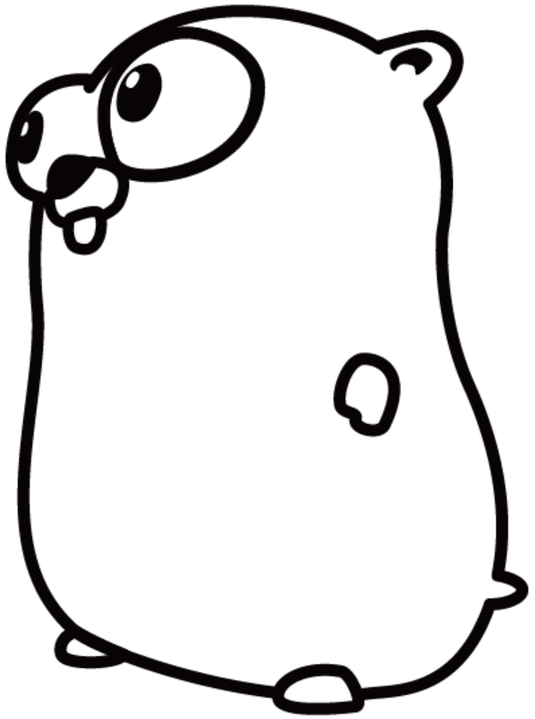
sigma交流
@gopherchina2018



阿里巴巴PouchContainer
讨论✌



Q/A



Github http://github.com/sebarzi (内含应聘笔试参考题)
Pouch https://github.com/alibaba/pouch
CluserData https://github.com/alibaba/clusterdata
Email: yingyuan.lyq@alibaba-inc.com