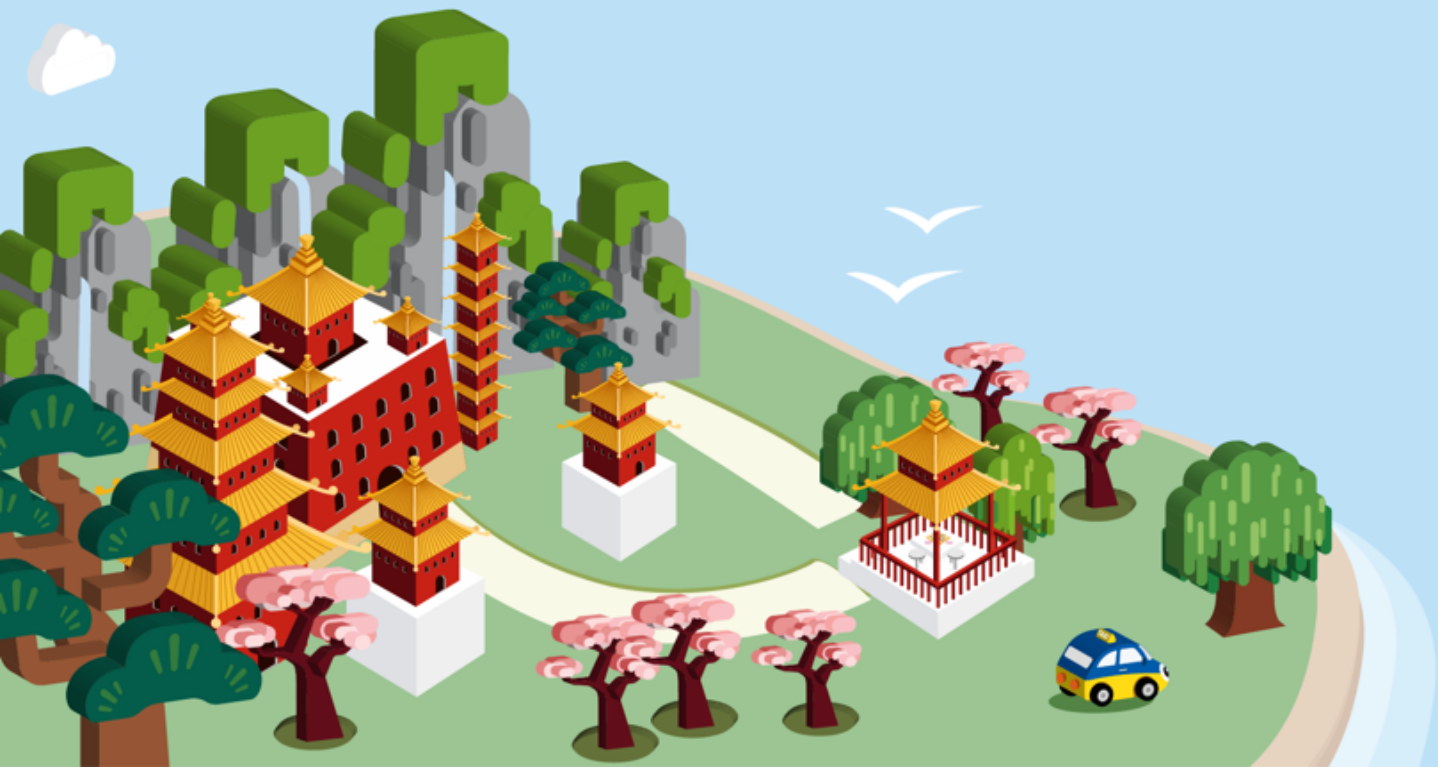




GopherChina2018

# 基于Go构建滴滴核心业务平台的实践





# 目录

- 1 Go In DiDi
- 2 治理经验
- 3 两个问题
- 4 两只轮子



# Golang使用现状



1500+ 个模块



1800+ 位Gopher



2000+ 台 (仅中台)



# 我们用Go做了什么

## DUSE

滴滴分单引擎

## DOS

滴滴订单系统

## DISE

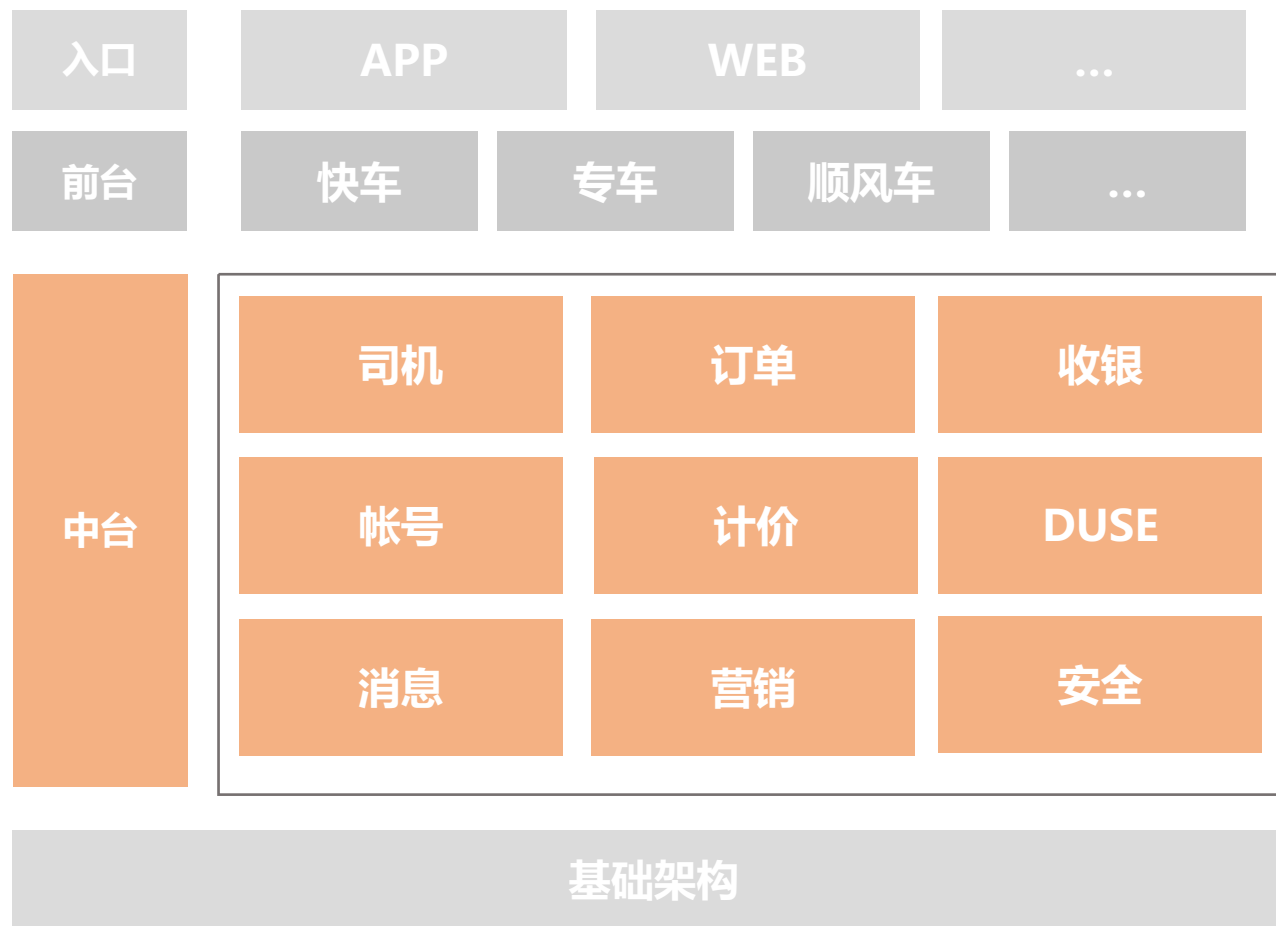
schemaless数据存储引擎

## DESE

serverless分布式事务框架



# 中台业务





# Challenge

## 高可用

- 高服务可用时间

## 高并发

- 服务承载能力
- 服务响应速度

## 复杂度

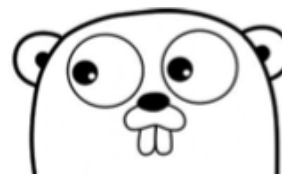
- 业务需求复杂
- 子系统较多
- 问题追查困难





# Why Golang

- 执行效率较高
- 开发效率
  - 便利的并发控制
  - 便利的网络服务开发
  - GC
- 丰富工具&库
  - go tool
  - go test
- 学习成本低



```
-----  
-----  
-----  
-----  
if err != nil {  
    return err  
}
```



# 目录

- 1 Go In DiDi
- 2 治理经验
- 3 两个问题
- 4 两只轮子





# 庞大的业务系统

微服务过多带来的问题



快车订单：**1单**



子模块：**50+**



Rpc请求：**300+**



日志行数：**1000+**



# 服务治理的难题

微服务过多带来的问题



异常定位



链路优化



服务迁移

.....



# 异常定位

滴滴如何定位业务问题

## 日志格式混乱

- 大量adaptor
- 人工配置与分析
- 处理性能低，资源消耗巨大

## 服务串联困难

- 上下游定位困难
- 跨业务线定位效率低
- 缺乏服务调用拓扑关系

## 链路难以分析

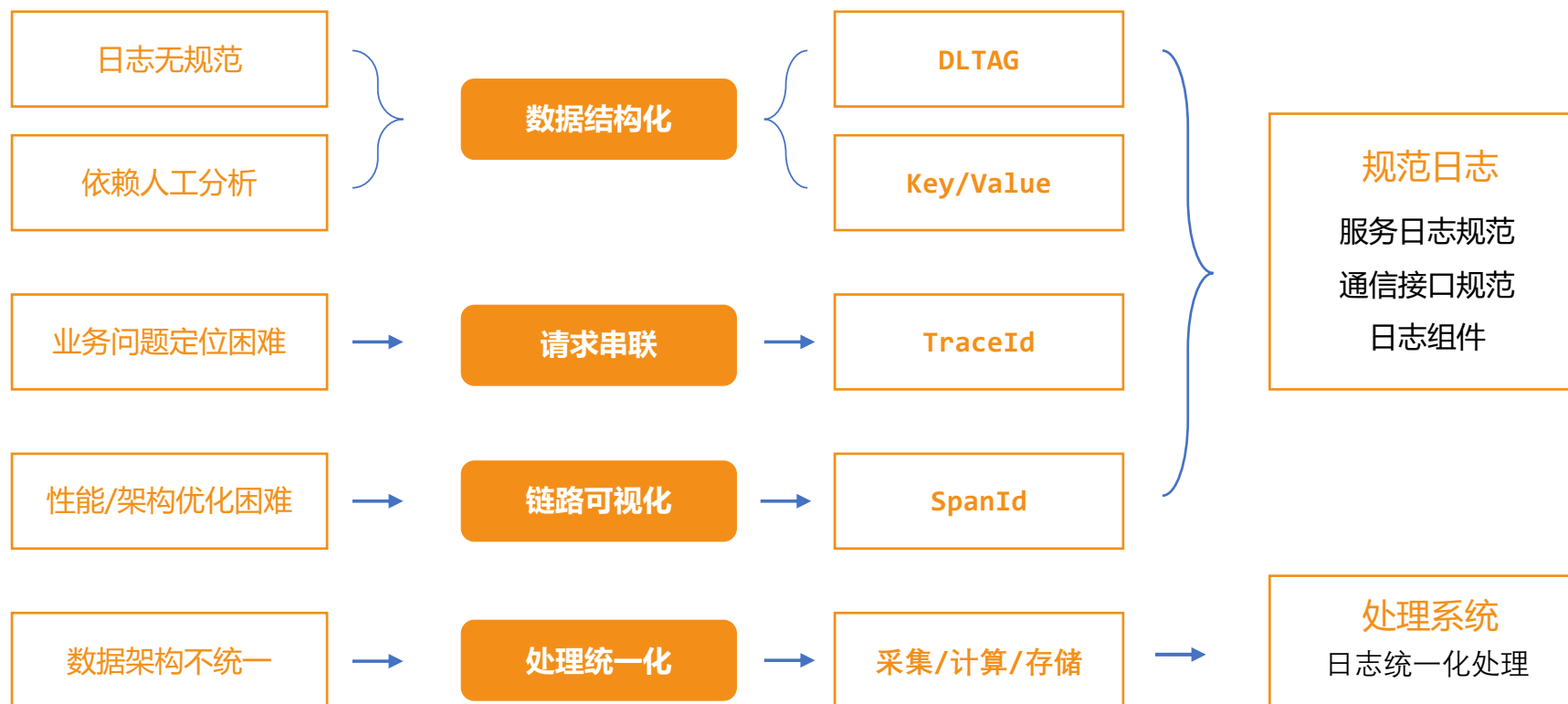
- 日志孤立
- 性能要素缺失





# 日志规范化

滴滴如何定位业务问题





# 日志串流

滴滴如何定位业务问题

日志源

服务端

APP

日志采集

SWAN

日志清洗

SRIUS

日志索引

ARIUS

把脉

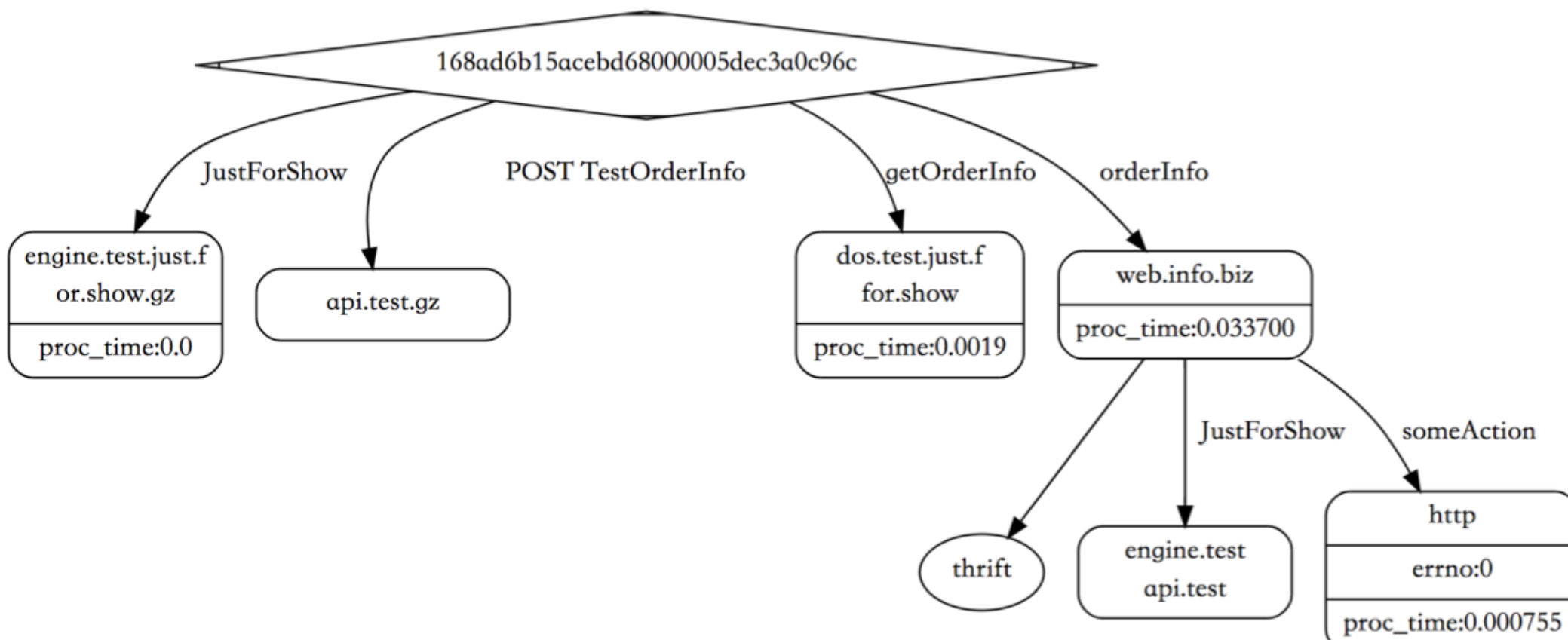
场景还原

服务分析

性能追查



# 调用拓扑





# 链路优化-压测

如何为诊断系统瓶颈

## 需要回答的问题

- 系统能够承载多少流量？
- 吞吐瓶颈在什么地方？
- 新建机房是否可用？
- 灾备预案是否可行？

---

## 传统压测的问题

- 非“函数式”业务
- 难以通过流量回放压测
- 难以通过线下等比放大估计



# 全链路压测

滴滴如何在线上环境压测

## 方案

### 流量标识方案

## 实施基础

### 全局流量标识

thrift

```
struct ReqHead {  
    1: optional i64 hintCode;  
    2: optional string hintContent;  
}
```

http

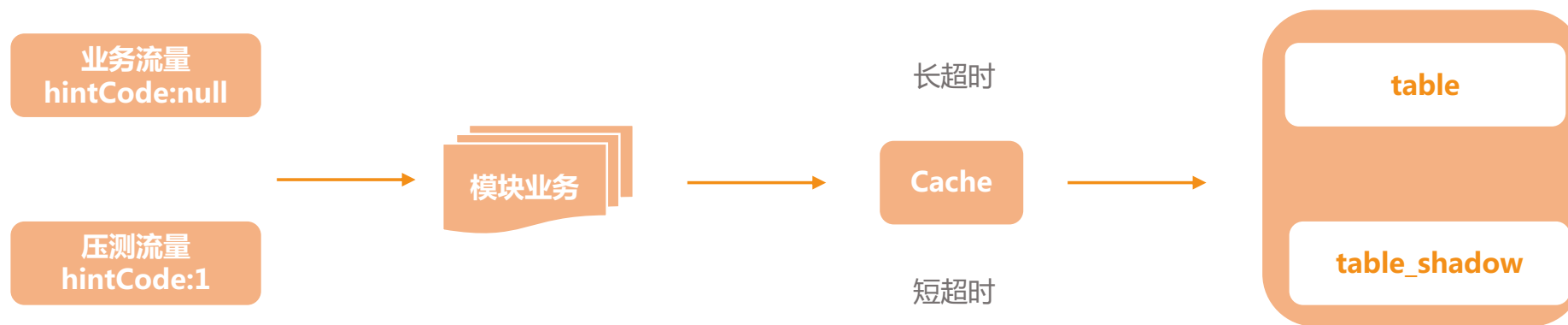
```
POST /didi/v1/just/for/show HTTP/1.1  
Host: 100.69.110.98:8000  
HintCode: 0000000  
HintContent: {}
```





# 全链路压测

滴滴如何在线上环境压测



## 压测频率

- 新机房容量测试
- 周期业务流程压测

## 压测范围

- 涉及所有业务模块
- 峰值压力的150%+

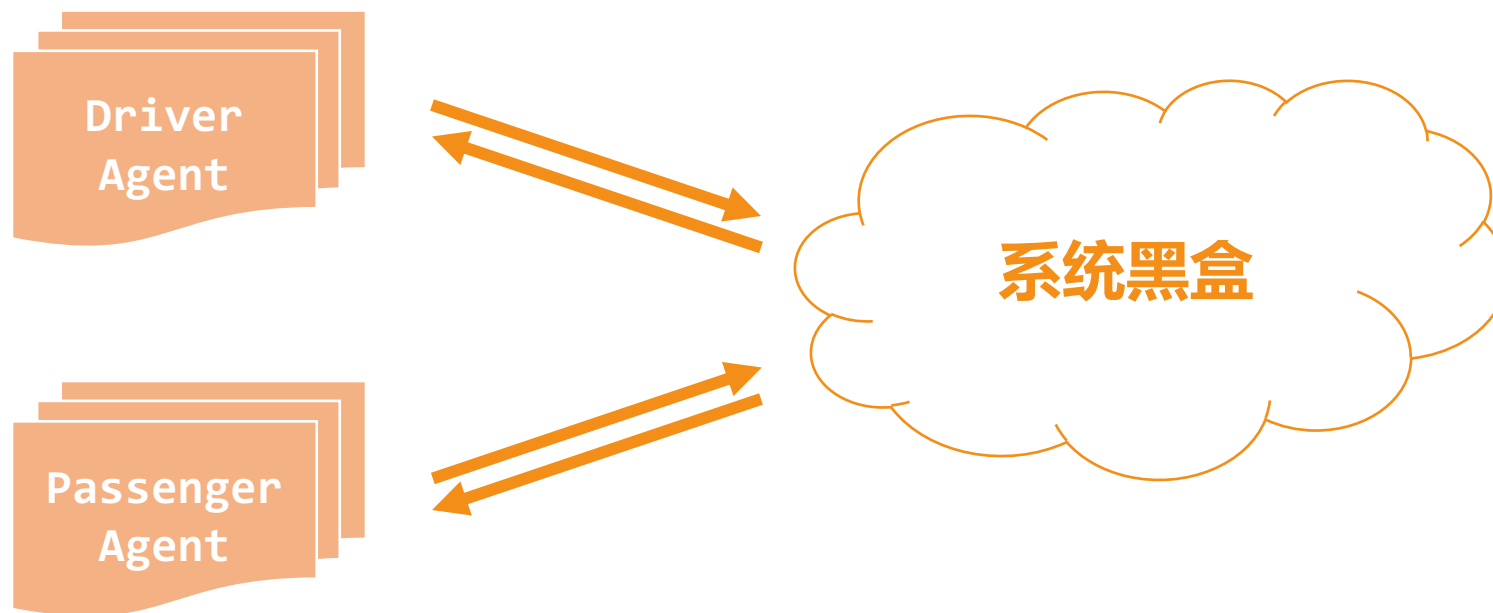
## 压测数据

- 抓取线上日志
- Agent模拟



# 全链路压测

滴滴如何在线上环境压测





# 全链路压测

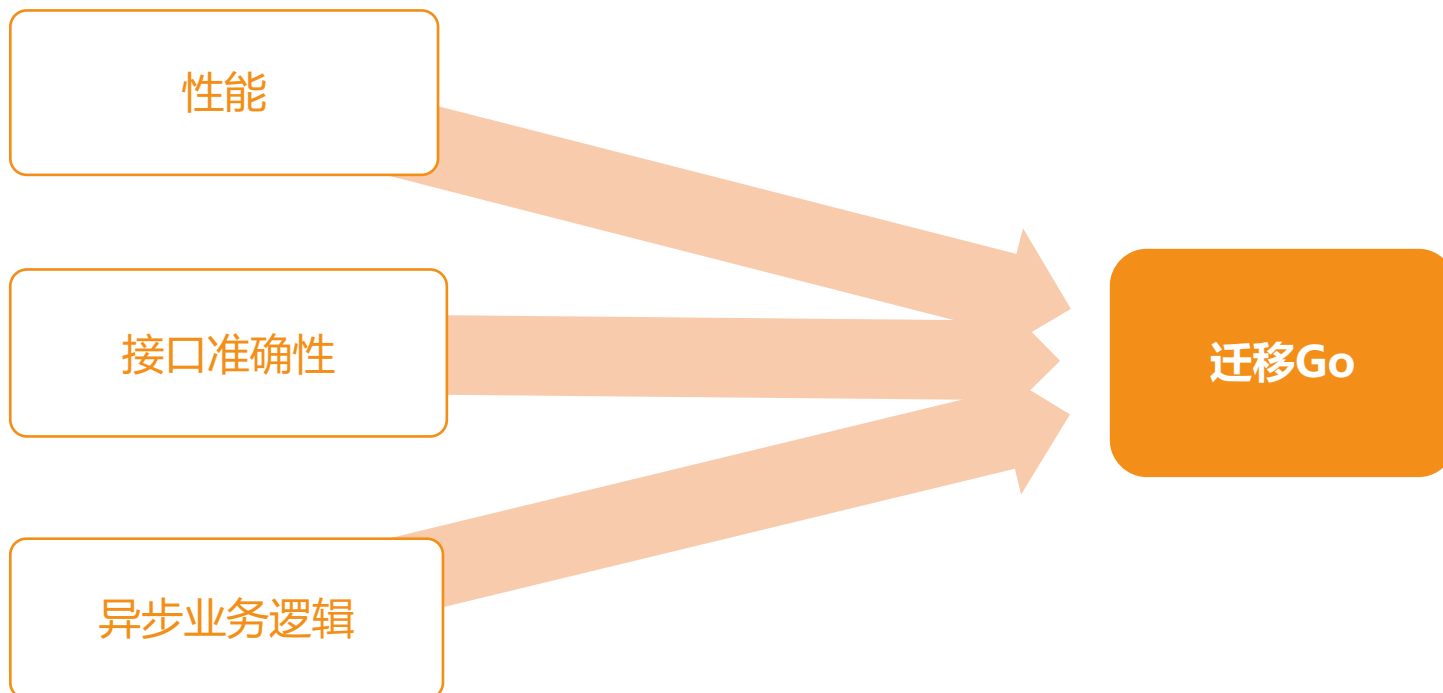
滴滴如何在线上环境压测





# 服务迁移

部分模块成为了系统瓶颈





# 希望什么

滴滴如何迁移业务

业务无感知/微感知

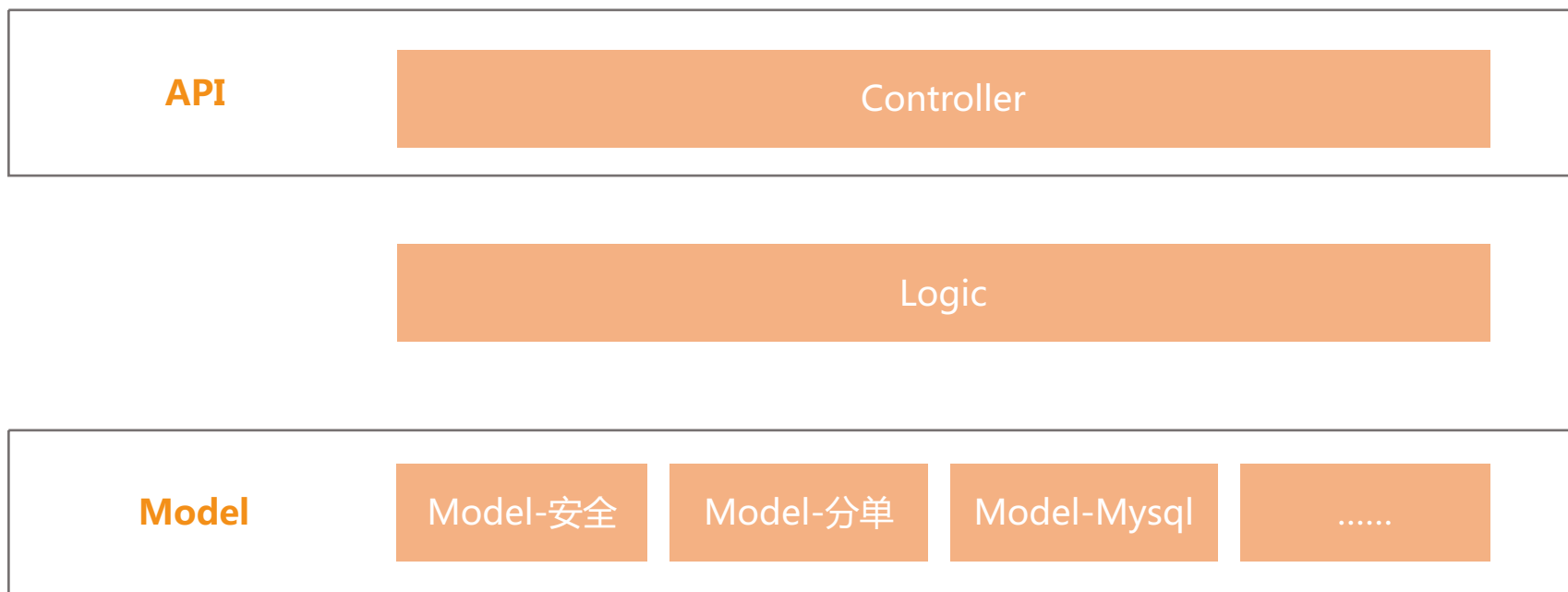
服务迁移稳定

逻辑功能无差异



# 迁移经验-How

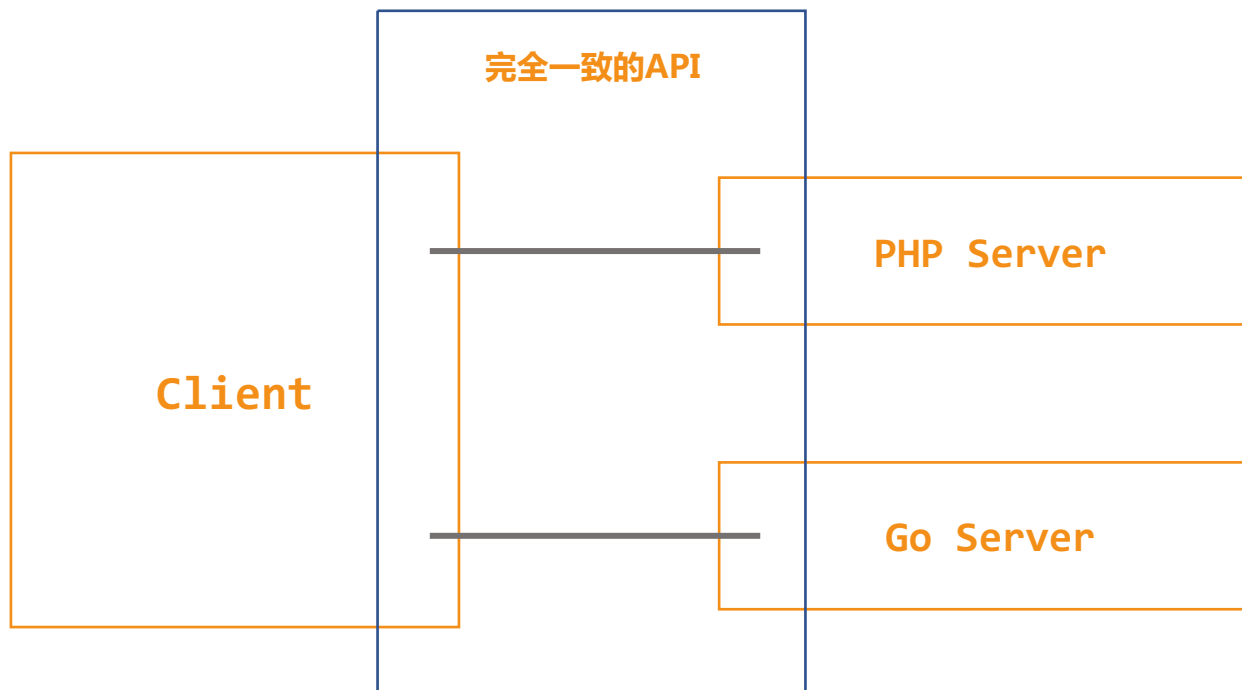
滴滴如何迁移业务





# 迁移经验-接口一致性

理想的方案

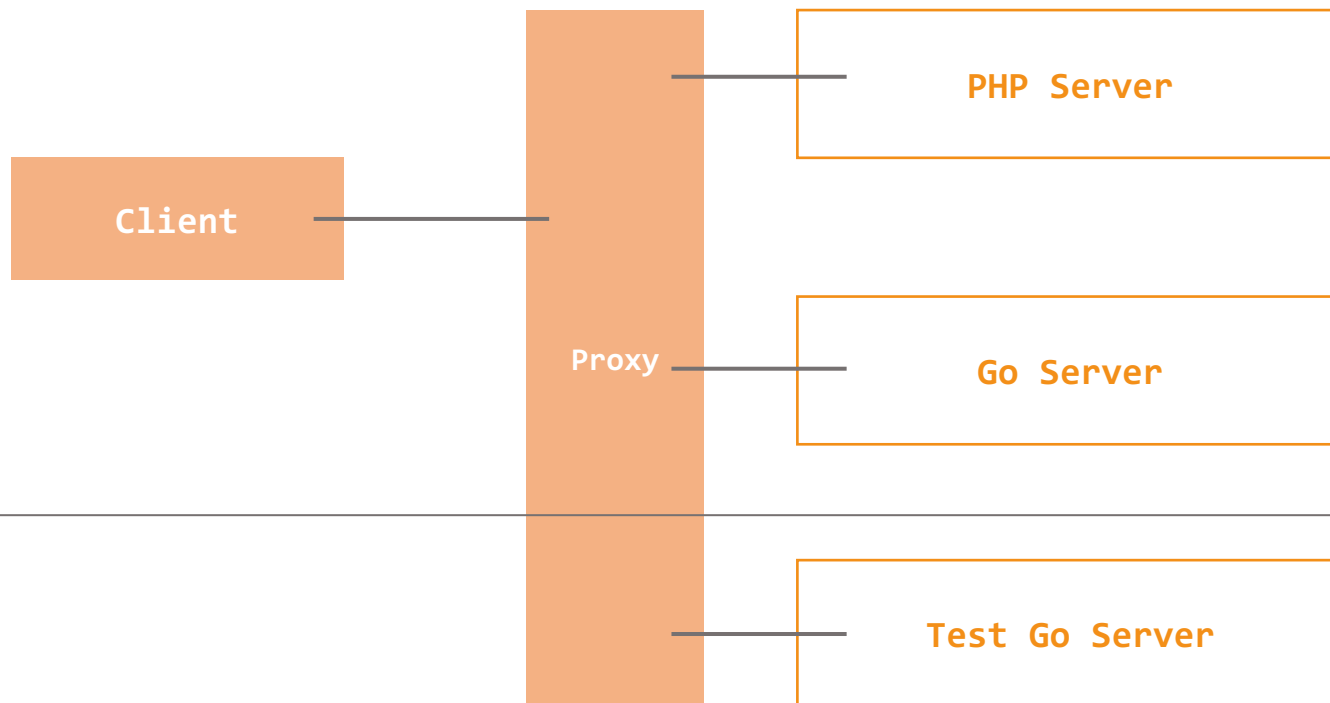




# 迁移经验-接口一致性-Proxy方案

SDK/代理方案

线上服务



线下测试



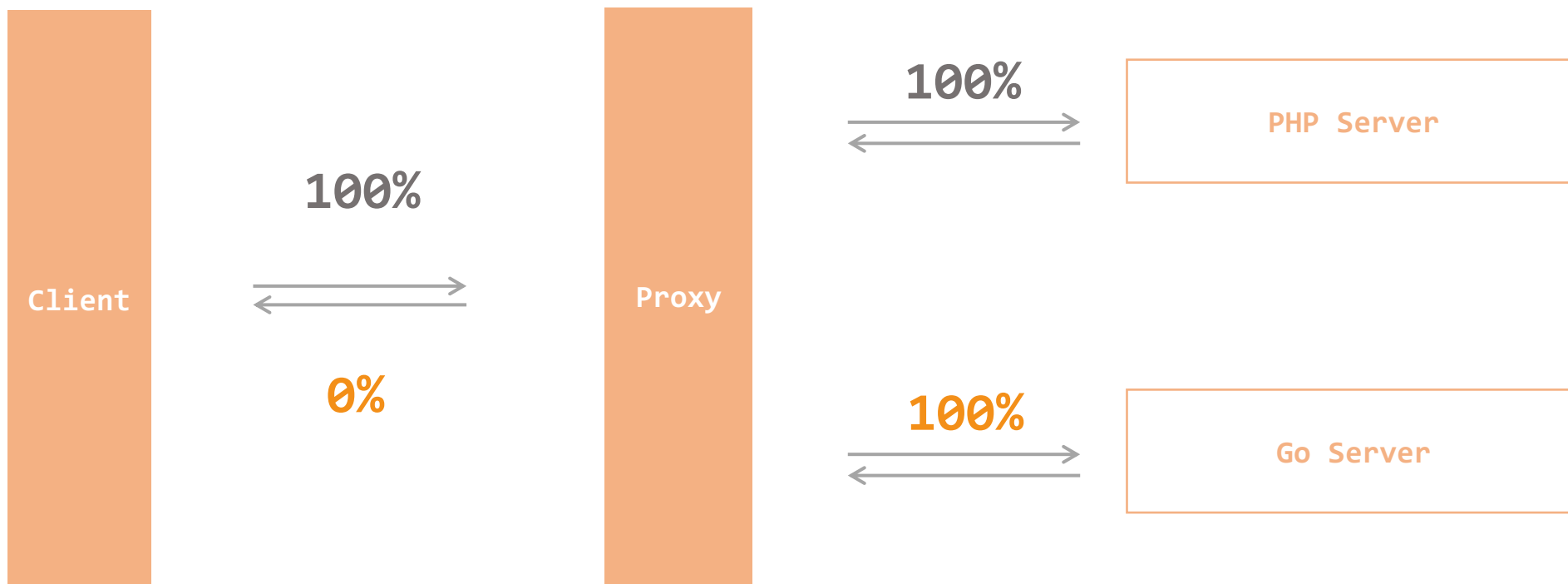


# 迁移经验



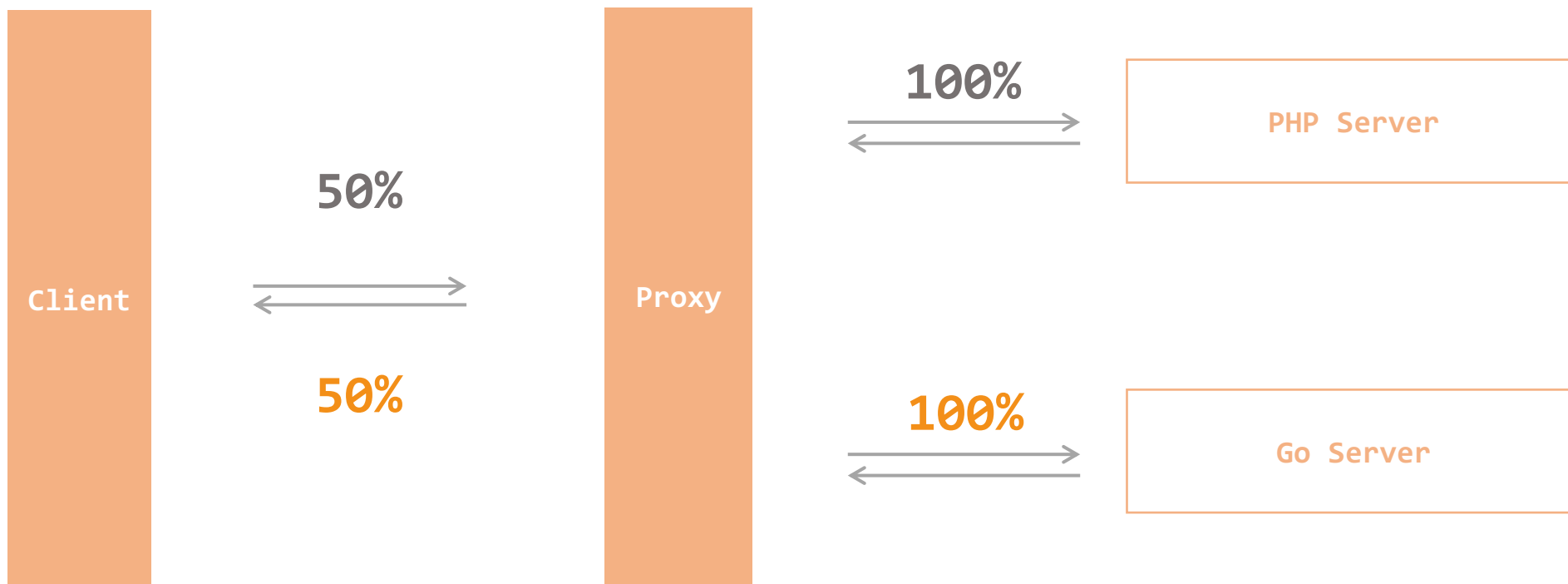


# 迁移经验-切流-旁路引流



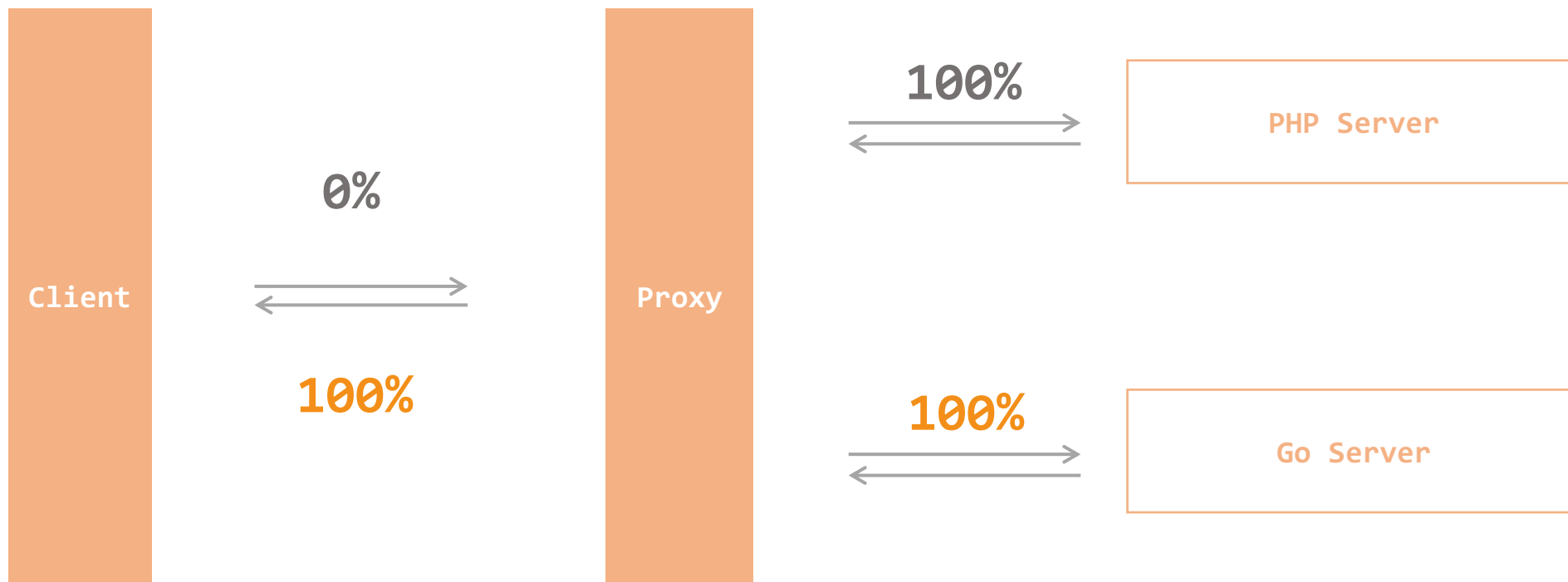


# 迁移经验-切流-流量切换





# 迁移经验-切流-线上观察





# 问题

治理组件繁复产生的新问题





# 在讨论什么？

当讨论RPC-SDK我们到底在讨论什么

## C端容错

- 错误的容忍

## 服务发现

- 部分SDK依旧在使用VIP/机器列表

## 请求埋点

- 上下游状态收集
- 请求日志存留
- 压测通道

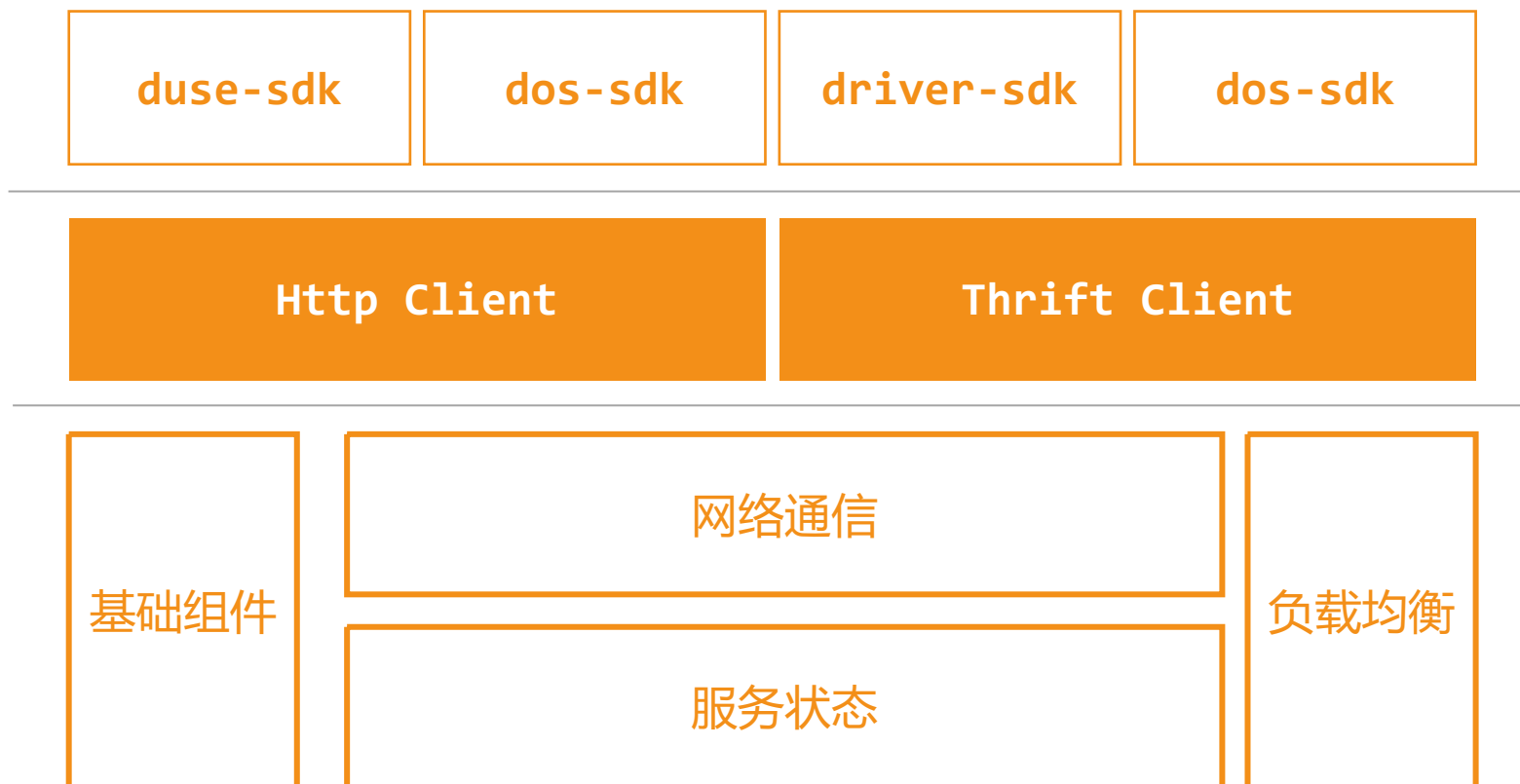
## 规范缺失

- IDL封装差异

一站式服务治理接入方案



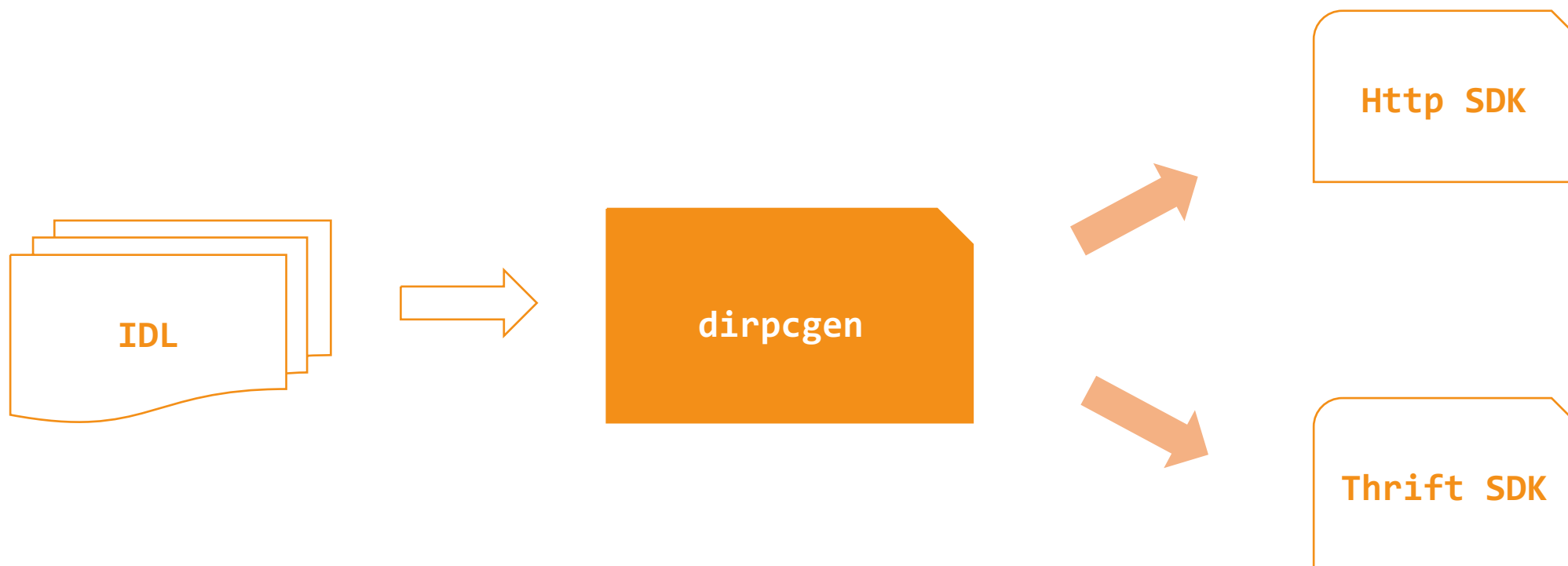
# 基础组件-DiRPC





# DiRPC-CodeGen

兼容thrift IDL语法的IDL







## 两个小问题



# 遇到问题-double close

WaitGroup导致的服务异常

链接开启

```
func (gl *graceListener) Accept() (c net.Conn, err error) {  
    tc, err := gl.Listener.(*net.TCPLListener).AcceptTCP()  
    .....  
    gl.server.wg.Add(1)  
    return  
}
```

链接关闭

```
type graceConn struct {  
    net.Conn  
    server *Server  
}  
  
func (c graceConn) Close() (err error) {  
    c.server.wg.Done()  
    return c.Conn.Close()  
}
```



# 遇到问题-double close

WaitGroup导致的服务异常

## 现象

- 服务Panic崩溃，回报"sync: negative WaitGroup counter"

## 排查

- 根据Panic堆栈，迅速定位到 " waitGroup.Add "
- waitGroup.Add检查操作后的结果，如果为负，触发Panic
- 唯一减值的地方为waitGroup.Done，唯一调用Done()的地方为Conn.Close()
- Conn为net.http包托管，难道Go有double close？

## 结论

- net/http/server.go:345，func (cw \*chunkWriter) Write()，中调用了conn.Close()
- net/http/server.go:1725，func (c \*conn) serve()中，再次调用conn.Close()



# 遇到问题-double close

WaitGroup导致的服务异常

## Bug ?

- 可能是Bug ?
- 提个Issue ?

## 不是Bug

- net/net.go:117

// Multiple goroutines may invoke methods on a Conn simultaneously.

```
// Conn is a generic stream-oriented network connection.
//
// Multiple goroutines may invoke methods on a Conn simultaneously.
↑ ↓ type Conn interface {
    // Read reads data from the connection.
    // Read can be made to time out and return an Error with Timeout() == true
    // after a fixed time limit; see SetDeadline and SetReadDeadline.
    ↓ Read(b []byte) (n int, err error)
```

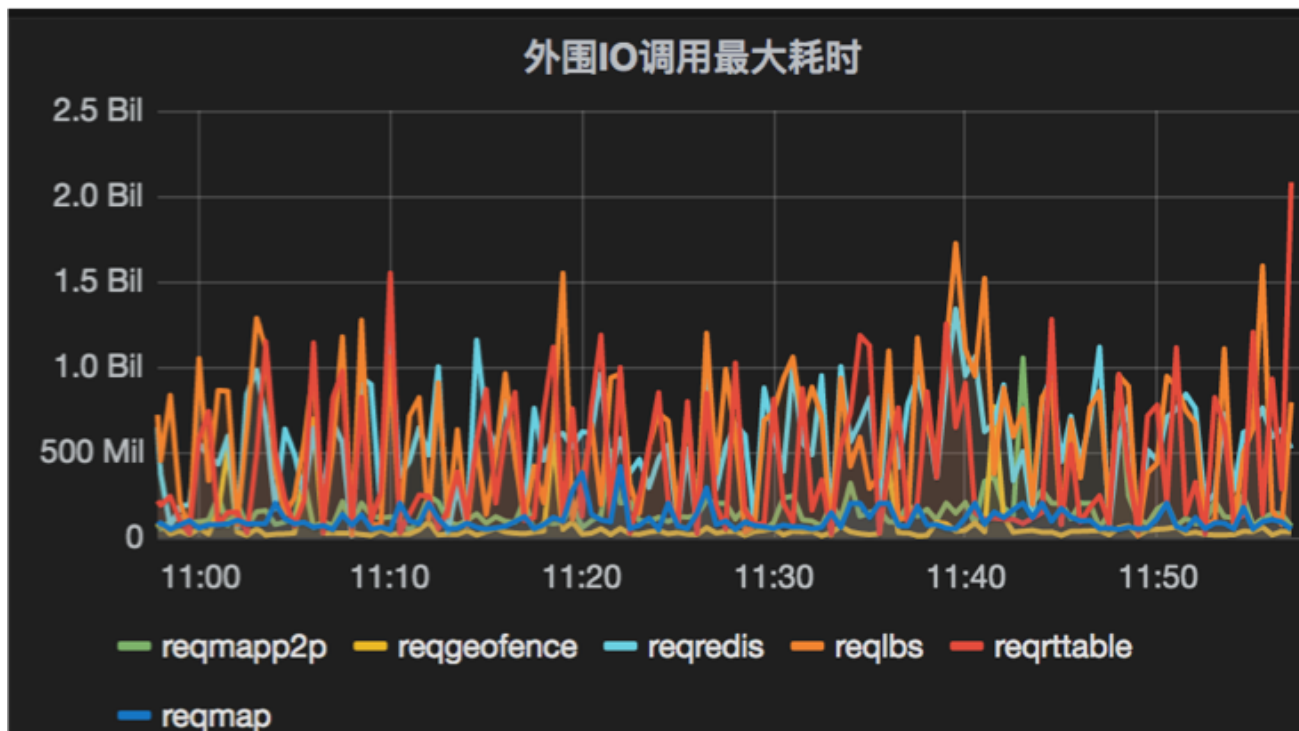


# 遇到问题-GC

小对象过多引起的服务吞吐问题

## 现象

- 随着流量增大，请求超时增多
- 耗时毛刺严重，99分位耗时较长
- 总体内存变化不大





# 遇到问题-GC

小对象过多引起的服务吞吐问题

## 排查

- go tool pprof -alloc\_objects
- go tool pprof -inuse\_objects

某函数生成20%的对象，约800W对象持续被引用

```
flat flat% sum%      cum  cum%
8847630 74.88% 74.88%   8847630 74.88% model.loadPassengerFeatures2.func2
2195488 18.58% 93.46%   2202042 18.64% fmt.Sprintf
201658 1.71% 95.17%   627648 5.31% model.loadGSModel
```

- go tool pprof bin/dupsd

GC扫描函数占据大量CPU，runtime.scanobject等

```
flat flat% sum%      cum  cum%
1330ms 7.82% 7.82%   1530ms 9.00% runtime.greyobject
1170ms 6.88% 14.71%   1220ms 7.18% syscall.Syscall
1040ms 6.12% 20.82%   2400ms 14.12% runtime.mallocgc
620ms 3.65% 24.47%   620ms 3.65% runtime.heapBitsForObject
590ms 3.47% 27.94%   2730ms 16.06% runtime.scanobject
```



# 遇到问题-GC

小对象过多引起的服务吞吐问题

## 结论

- **对象数量过多**，导致GC三色算法耗费较多CPU

**优化** 思路，减少对象分配

对象 `string, map[key]value, slice, *Type`

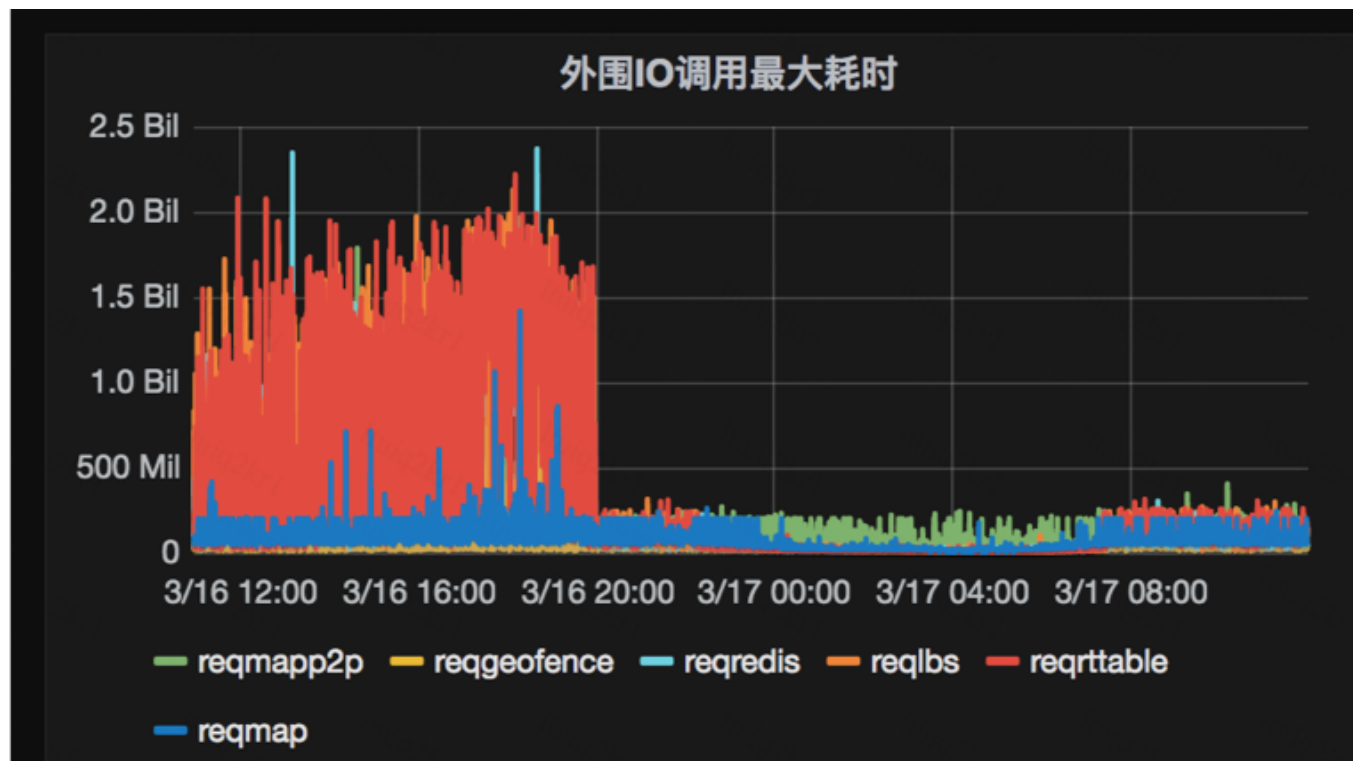
原数据结构	优化数据结构	优化点
<code>map[string]SampleStruct</code>	<code>map[[32]byte]SampleStruct</code>	Key使用值类型避免对map遍历
<code>map[int]*SampleStruct</code>	<code>map[int]SampleStruct</code>	Value使用值类型避免对map遍历
<code>sampleSlice []float64</code>	<code>sampleSlice [32]float64</code>	利用值类型代替对象类型



# 遇到问题-GC

小对象过多引起的服务吞吐问题

效果







# 两只轮子



# 开源项目-Gendry

github.com/didi/gendry

数据库操作辅助工具

## Gendry-Manager

连接池管理类

## Gendry-builder

SQL构建工具

## Gendry-scanner

结构映射工具

```
where := map[string]interface{}{
    "city in": []interface{}{"beijing", "shanghai"}
}

table := "some_table"
selectFields := []string{"name", "age", "sex"}
cond, values, err := builder.BuildSelect(table, where,
selectFields)

rows, err := db.Query(cond, values...)
defer rows.Close()

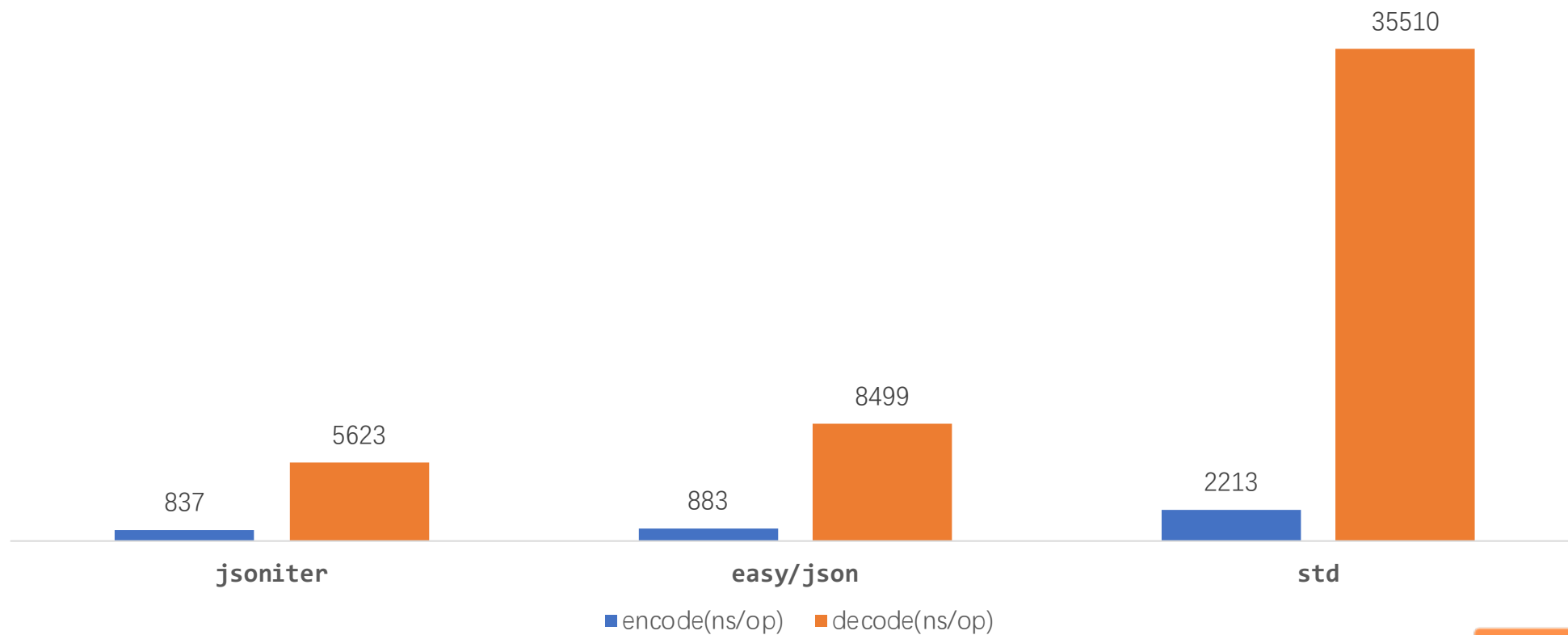
var students []Person
scanner.Scan(rows, &students)
```



# 开源项目-Jsoniter

[github.com/json-iterator/go](https://github.com/json-iterator/go)

兼容原生API的Json编解码工具





# GopherChina2018



# Q & A

