Go语言开发框架性能王者
——TARS-GO

陈明杰

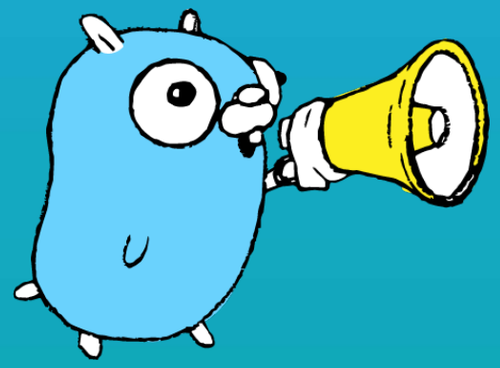
腾讯

TARS开源团队

“

陈明杰

腾讯TARS团队核心成员。负责腾讯容器云平台及机器学习平台的建设和运营，目前专注TARS开发框架的Golang版本开发。对容器技术，内核技术，高可用架构，微服务等有较深理解。



Today’s (glorious) blather.

SECTION ONE

TARS的架构体系

开源微服务框架现状

无服务治理类

专注于通信框架，RPC或消息队列模式，部分框架支持多语言开发

Apache Thrift ™

GRPC

ServiceMesh

支持服务治理，通过SideCar模式解决多语言问题，目前处于发展成熟期

LINKERD

envoy

ISTIO

单语言带服务治理类

在通信框架的基础上支持服务治理能力，单一编程语言实现，JAVA语言为主流

Spring Cloud

DUBBO

多语言带服务治理类

在通信框架的基础上支持服务治理能力，多种编程语言实现

TARS

Github.com/TarsCloud

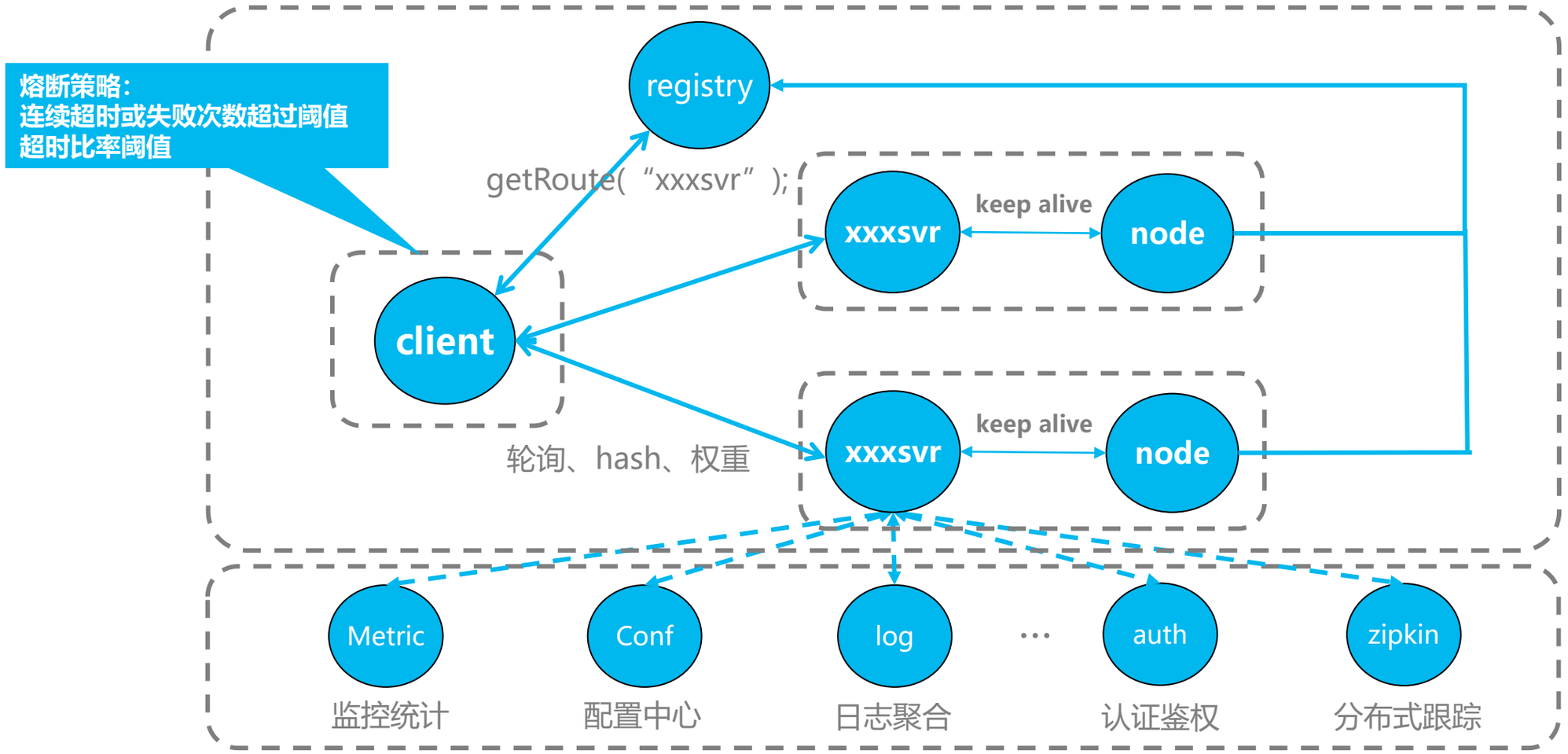
TARS与流行微服务框架对比

对比项	Tars	GRPC	备注
网络协议	udp/tcp/http2/自定义	http2	Tars默认提供udp/tcp支持，grpc只支持http2
IDL	tars/pb/自定义	pb	IDL本身思路差不多，tars可以设置字段默认值
多语言	支持	支持	Grpc/tars都支持多语言开发
RPC性能	非常高	一般	经过测试，tars性能高grpc 5倍
服务治理	提供整套体系	无	Tars提供整个服务治理生态体系，grpc更多的是提供思路，由其它社区或者自己构建生态。
应用规模	已大规模应用10年	社区广泛应用	Tars已经大规模应用10年，grpc社区用得更多，时间不长。

Tars整体架构

- **Tars是一个支持多语言、内嵌服务治理功能，与Devops能很好协同的微服务框架**

服务治理



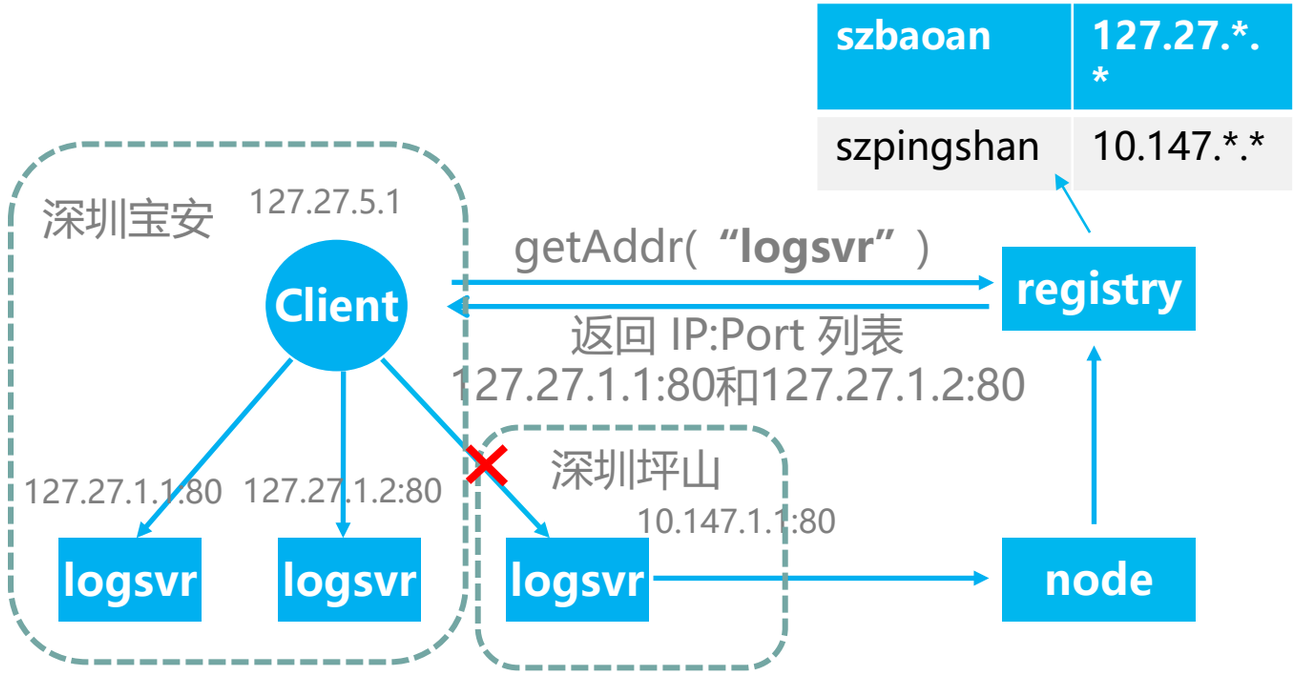
开发框架、Registry、node和基础服务集群协同工作，透明完成服务发现/注册、负载均衡、鉴权、调用链等等服务治理相关工作

自动区域感知

- 常规的负载均衡方式面对跨地区或者跨机房部署的服务会因为网络原因造成延时增大
- 使用不同服务名来解决该问题时会带来繁重的运维工作
- 通过Registry和开发框架配合实现自动区域感知

自动区域感知优势

- 运维简单
- 降低延时减少带宽消耗
- 更强的容灾能力

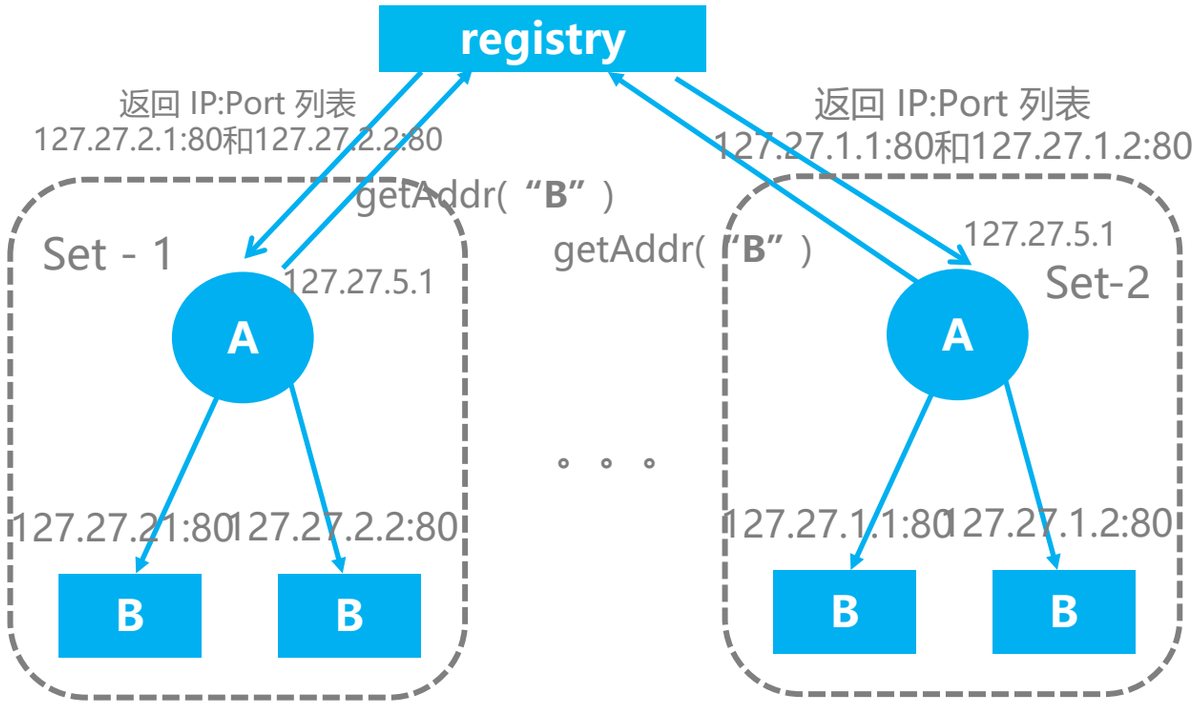


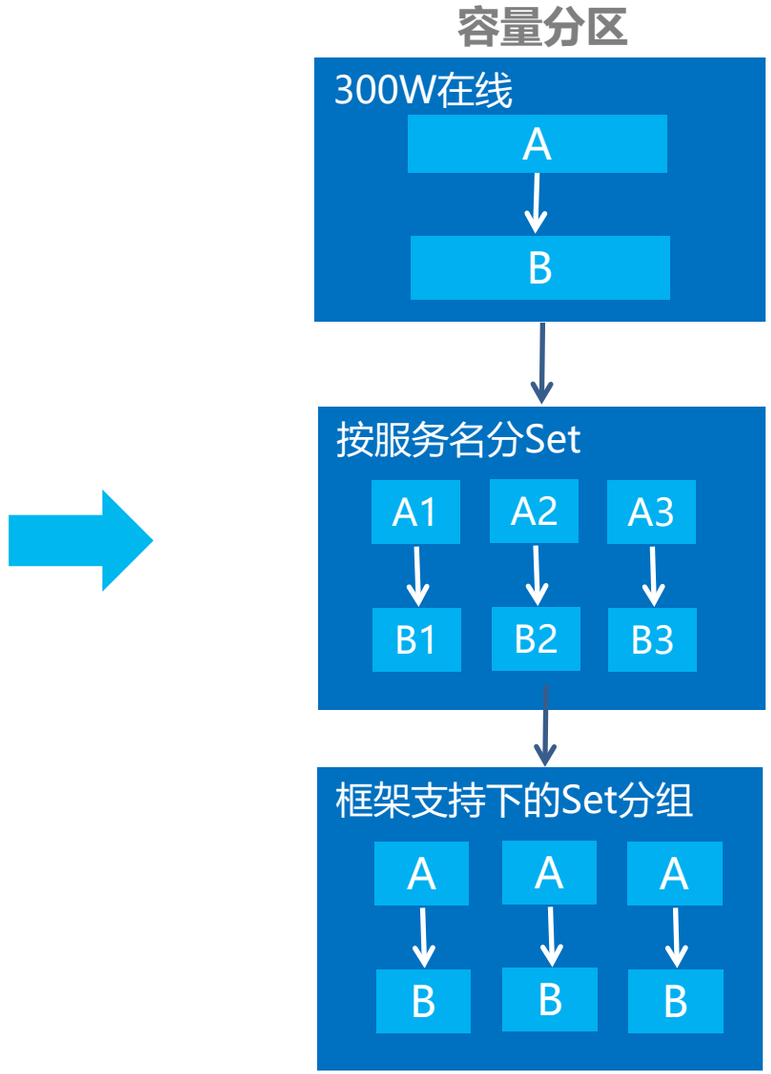
SET模型

根据业务功能特征对部署进行规范化和标准化，以Set为单元进行部署

优点：

- **有效防止故障扩散**
- **实现海量服务的高效运营**
- **通过Registry和开发框架配合实现快速动态构建Set模型**

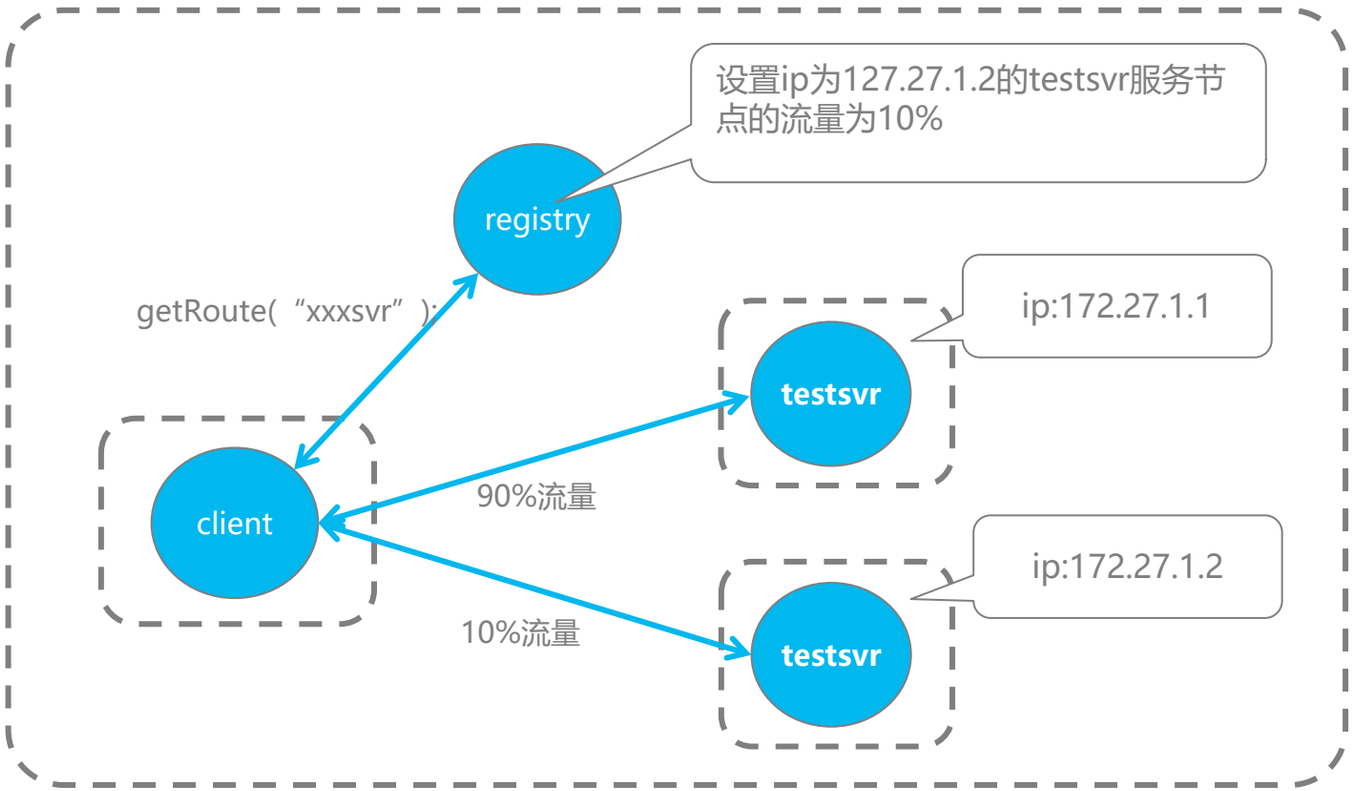




流量控制

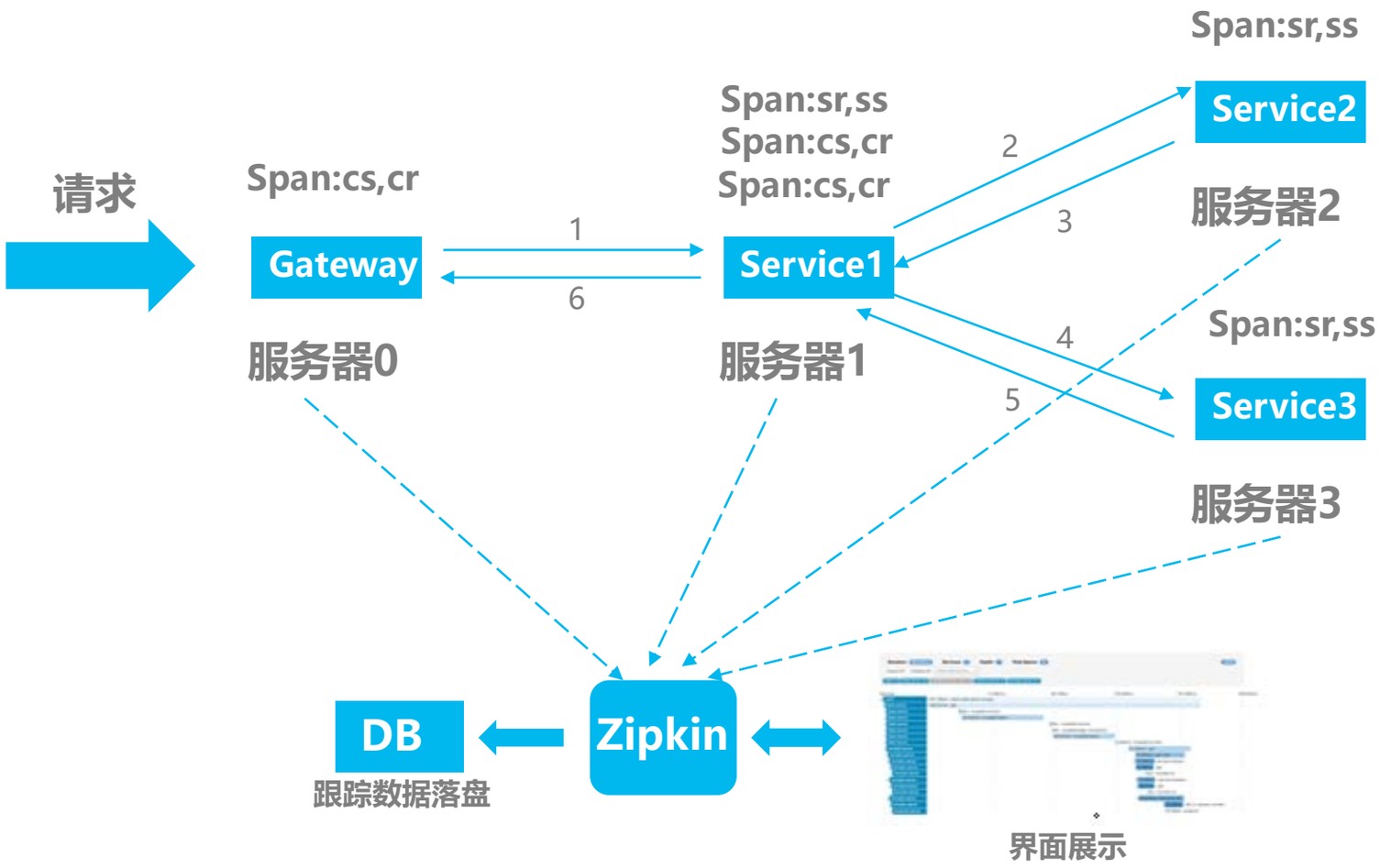
服务发布上线面对的问题：

- 如何做对业务无损的服务变更
- 如何做灰度验证



通过Registry和开发框架配合实现按需进行流量控制，达到无损发布和灰度流量的目的

分布式跟踪

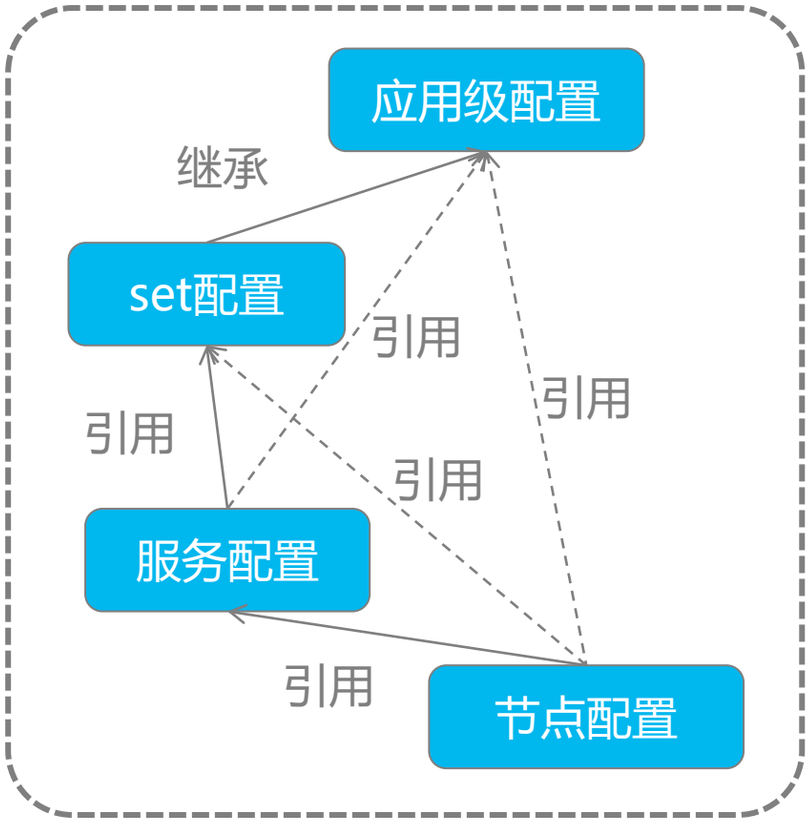


- 利用开源的Zipkin实现分布式跟踪，避免重复造轮子
- 框架内部嵌入跟踪锚点使用对业务透明

服务配置

配置中心的配置有两类：

1. 模版配置（框架自身配置）
2. 业务配置（业务配置分4级）



业务可以灵活使用

配置管理简单

添加配置

	服务名称	文件名称	最后修改时间	操作
◉	TestApp.TracingJavaFourServer	application.properties	2018-05-14 21:34:24	修改配置 | 删除配置 | 查看内容 | 查看历史

引用文件列表

添加引用文件

服务名称	节点	文件名称	最后修改时间	操作
TestApp		TestApp.conf	2018-06-18 20:50:29	删除引用 | 查看引用内容 | 查看历史

节点配置列表

PUSH配置文件

	服务名称	节点	文件名称	最后修改时间	操作
☐	TestApp.TracingJavaFourServer	10.120.129.226	application.properties	2018-05-14 21:34:24	修改配置 | 查看合并后配置 | 查看节点内容 | 查看历史 | 管理引用文件

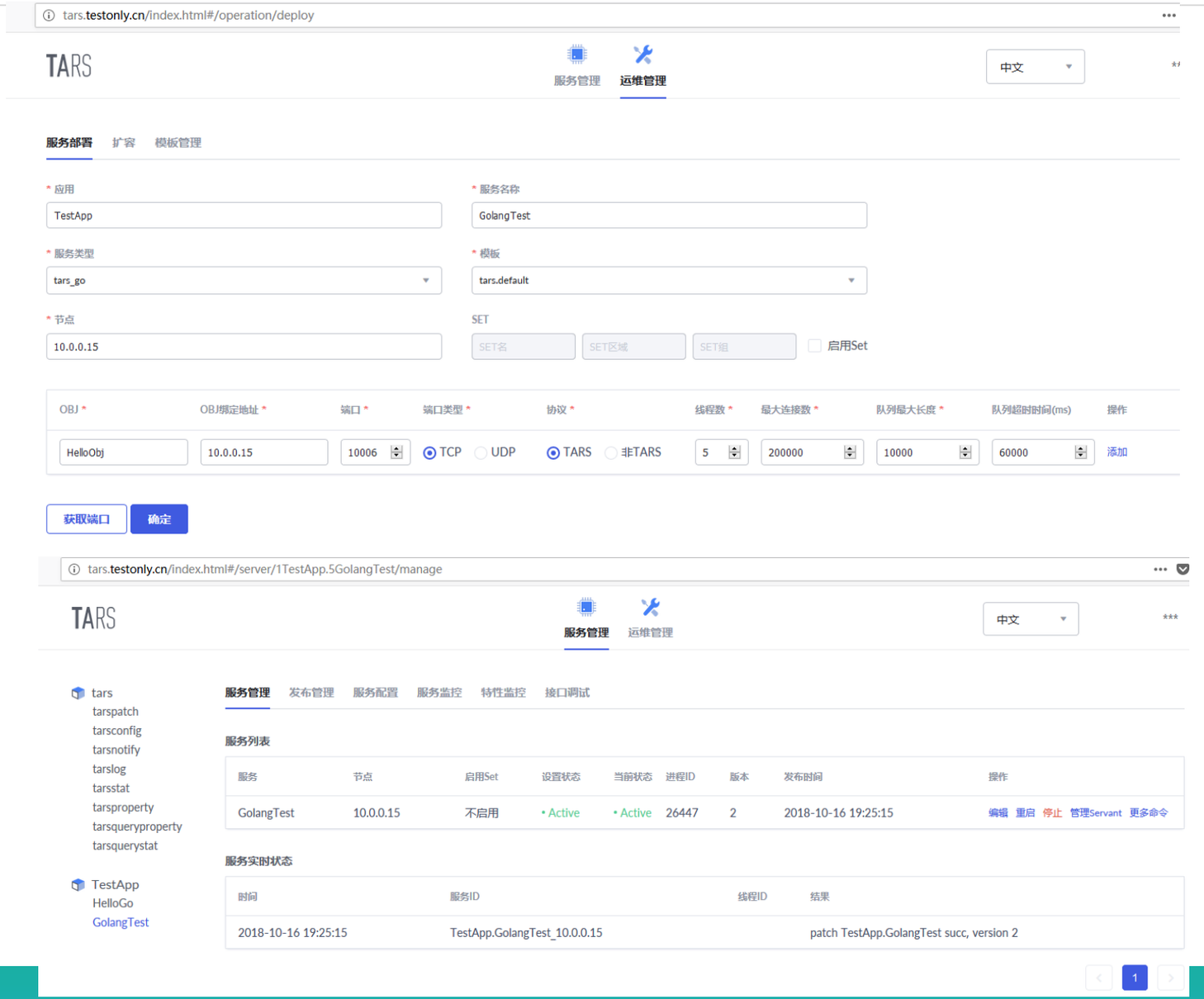
开发使用方便：

```
//获取配置文件
bool addConfig(const string &filename);
```

Tars可视化管理

可视化管理

- 注册业务服务
- 上传包并发布
- 可视化管理，可以扩容节点，操作节点，修改配置等等操作



SECTION TWO

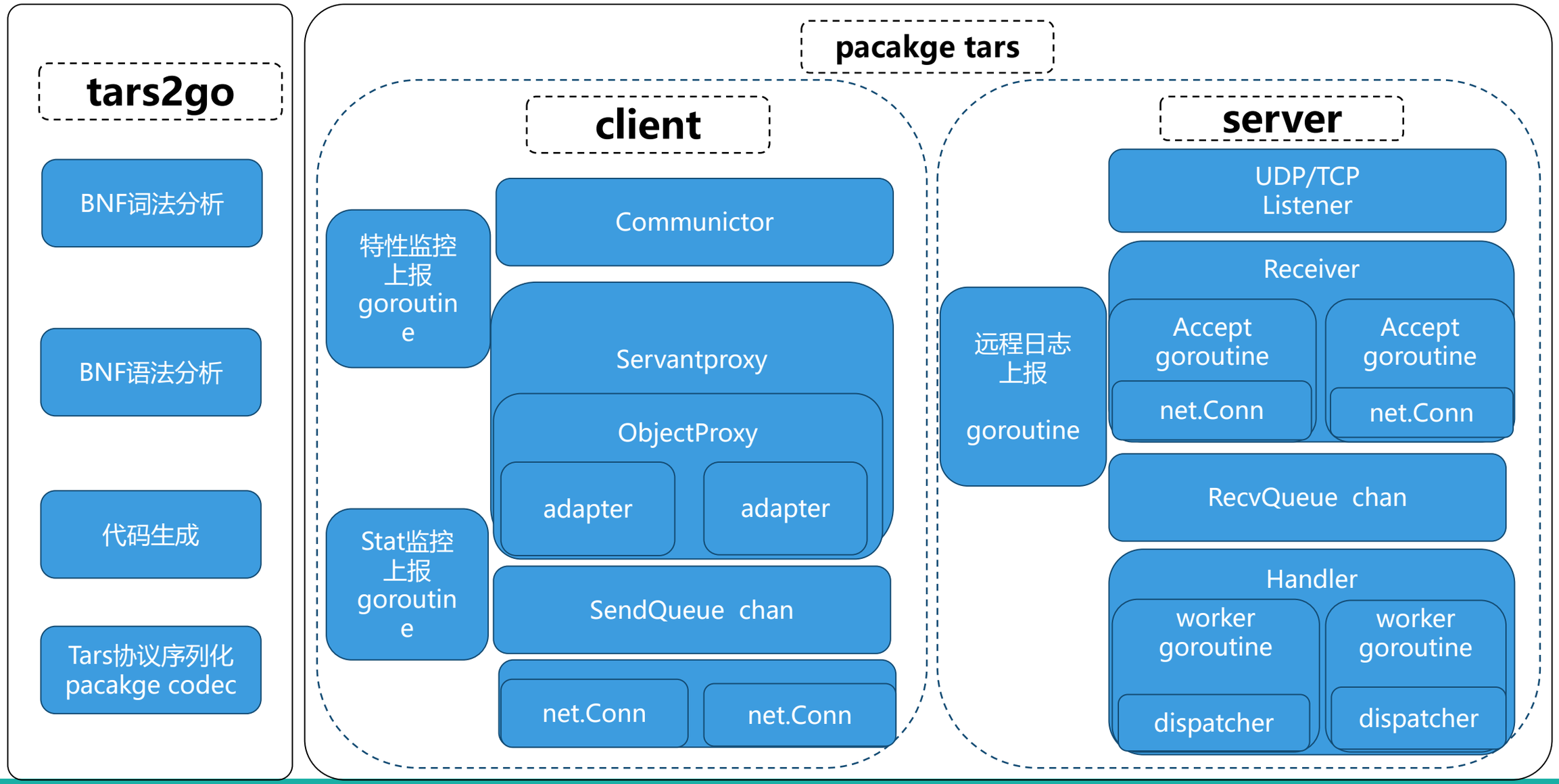
TARS为什么要加入Go语言版本

TarsGo研发背景

- 统一：
  - 微服务迭代交接节奏快，go的编程风格和设计统一，适合多人迭代开发
- 高效：
  - 实际应用过程中除了大规模后台服务场景，更多的是运营小工具场景，使用go开发运营工具也更快
- 时代：
  - docker/k8s/etcd等开源项目大规模应用go语言，需要跟上云时代

- 学习成本
  - 关键字少：新人一周上手开发，C++转GO几乎无门槛
  - 语言设计直观：新接手业务可以快速开发，没有奇怪的语法糖与复杂的定义
- 开发难度
  - 编译速度快：普通办工机即可编译
  - 库提供源码：快速排查引用库问题
  - 自带工具库：pprof、交叉编译、文档生成
- 版本管理
  - 远程git管理：go get安装与更新库，引用库集中管理
- 运营管理
  - 静态编译：运行程序不会对第三方库产生依赖
  - 只依赖系统调用：不依赖glibc，对操作系统版本无限制，线上多环境运营至关重要

TarsGo架构


tars2go
BNF词法分析
BNF语法分析
代码生成
Tars协议序列化
pacakge codec
pacakge tars
client
特性监控
上报
goroutin
e
Communictor
Servantproxy
ObjectProxy
adapter
adapter
Stat监控
上报
goroutin
e
SendQueue chan
net.Conn
net.Conn
server
远程日志
上报
goroutine
UDP/TCP
Listener
Receiver
Accept
goroutine
net.Conn
Accept
goroutine
net.Conn
RecvQueue chan
Handler
worker
goroutine
dispatcher
worker
goroutine
dispatcher

SECTION THREE

高性能秘诀
——TARS协议及编码

Tars协议

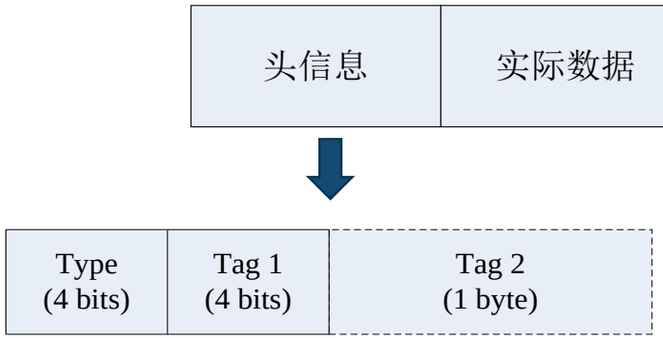
1. 二进制协议
2. 语言无关
3. 接口描述（IDL）
4. 工具自动生成服务端客户端代码

```
struct LoginInfo {
    0 require string sid;
    1 require string code;
};

struct ProfileInfo {
    0 require string nick;
    1 require int level;
};
interface TestServant {
    int test(int qq, LoginInfo li, out ProfileInfo pi);
}
```

Tars协议

Tars协议编码:



虚线部分的Tag 2是可选的，当Tag的值不超过14时，只需要用Tag 1就可以表示；当Tag的值超过14而小于256时，Tag 1固定为15，而用Tag 2表示Tag的值。Tag不允许大于255。
Type表示类型，用4个二进制位表示，取值范围是0~15，用来标识该数据的类型。不同类型的数据，其后紧跟着的实际数据的长度和格式都是不一样的，详见一下的类型表。
Tag由Tag 1和Tag 2一起表示。取值范围是0~255，即该数据在结构中的字段ID，用来区分不同的字段。

总结：用4个bit表示类型，其中元素一般情况下用4个bit表示，超过15后使用1byte表示及255个。

RPC协议:

RequestF.tars

```
module requestf
{
    struct RequestPacket
    {
        1  require short        iVersion;
        2  require byte         cPacketType  = 0;
        3  require int          iMessageType = 0;
        4  require int          iRequestId;
        5  require string       sServantName = "";
        6  require string       sFuncName    = "";
        7  require vector<unsigned byte> sBuffer;
        8  require int          iTimeout     = 0;
        9  require map<string, string> context;
        10 require map<string, string> status;
    };

    struct ResponsePacket
    {
        1 require short         iVersion;
        2 require byte          cPacketType  = 0;
        3 require int           iRequestId;
        4 require int           iMessageType = 0;
        5 require int           iRet         = 0;
        6 require vector<unsigned byte>  sBuffer;
        7 require map<string, string> status;
        8 optional string        sResultDesc;
        9 optional map<string, string> context;
    };
};

```

TarsGo开发示例

Tars文件接口定义

示例：

- 定义一个模块TestApp，
- 定义一个Hello接口类型
- 定义一个Sub的接口
- a 和 b为入参，c为出参，均为整型

Servant.tars

```
1  module TestApp{
2  interface Hello{
3      int Sub(int a,int b,out int c);
4  };
5  };
6
```

TarsGo开发示例

生成接口代码

- tars2go -outdir=vendor Servant.tars
- 生成Hello_IF.go文件，包含客户端与服务端的RPC代码

Servant.tars | Hello_IF.go

```go
func (_obj *Hello) Sub(A int32, B int32, C *int32, _opt ...map[string]string) (ret int32, err error) {

    var length int32
    var have bool
    var ty byte
    _os := codec.NewBuffer()
    err = _os.Write_int32(A, 1)
    if err != nil {
        return ret, err
    }

    err = _os.Write_int32(B, 2)
    if err != nil {
        return ret, err
    }

    var _status map[string]string
    var _context map[string]string
    _resp := new(requestf.ResponsePacket)
    err = _obj.s.Tars_invoke(0, "Sub", _os.ToBytes(), _status, _context, _resp)
    if err != nil {
        return ret, err
    }
    _is := codec.NewReader(_resp.SBuffer)
    err = _is.Read_int32(&ret, 0, true)
    if err != nil {
        return ret, err
    }

    err = _is.Read_int32(&(*C), 3, true)
    if err != nil {
        return ret, err
    }

    _ = length
    _ = have
    _ = ty
    return ret, nil
}
```

TarsGo开发示例

服务端实现

- 业务编写Sub函数的实现并设置指针返回值
- 将实现的imp对象实例化并AddServant(注册)进入框架
- 将tars服务启动，完成服务开发
- 通过web平台发布

Servant.tars | ServantImp.go ×

```go
package main

//HelloImp Imp类型
type HelloImp struct {
}

//Sub 实现减法函数
func (imp *HelloImp) Sub(a int32, b int32, c *int32) (int32, error) {
    tmp := a - b
    *c = tmp
    return 0, nil
}

```

Servant.tars | ServantImp.go | Server.go ×

```go
package main

import (
    "github.com/TarsCloud/TarsGo/tars"

    "TestApp"
)

func main() { //Init servant
    imp := new(HelloImp)                                    //New Imp
    app := new(TestApp.Hello)                               //New init the A Tars
    cfg := tars.GetServerConfig()                           //Get Config File Object
    app.AddServant(imp, cfg.App+"."+cfg.Server+".HelloObj") //Register Servant
    tars.Run()
}

```

TarsGo开发示例

客户端实现

- 引用tarsgo生成的接口模块TestApp
- 创建一个通信器(Conmunicator)
- 使用StringToProxy初始化通信器
- 直接调用本地函数实现远程的函数调用

Servant.tars | ServantImp.go | client.go

```go
package main

import (
    "TestApp"
    "fmt"

    "github.com/TarsCloud/TarsGo/tars"
)

func main() {
    comm := tars.NewCommunicator()
    obj := fmt.Sprintf("TestApp.GolangTest.HelloObj@tcp -h 127.0.0.1 -p 10015 -t 60000")
    app := new(TestApp.Hello)
    comm.StringToProxy(obj, app)
    var a, b, c int32
    a = 3
    b = 4
    ret, err := app.Sub(a, b, &c)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(ret, c)
}

```

SECTION FOUR

TARS-Go性能提升措施

Tars编解码优化

- **问题：**
  - Tars协议编解码情况性能低下，tars2go依赖cpp代码
- **解决方案：**
  - 尽可能减少数据copy，使用指针传递返回数据
  - 类型去掉反射，使用tars协议type类型
  - 纯go实现了AST，抛弃了yacc与bison，实现了框架从生成到实现的纯go化

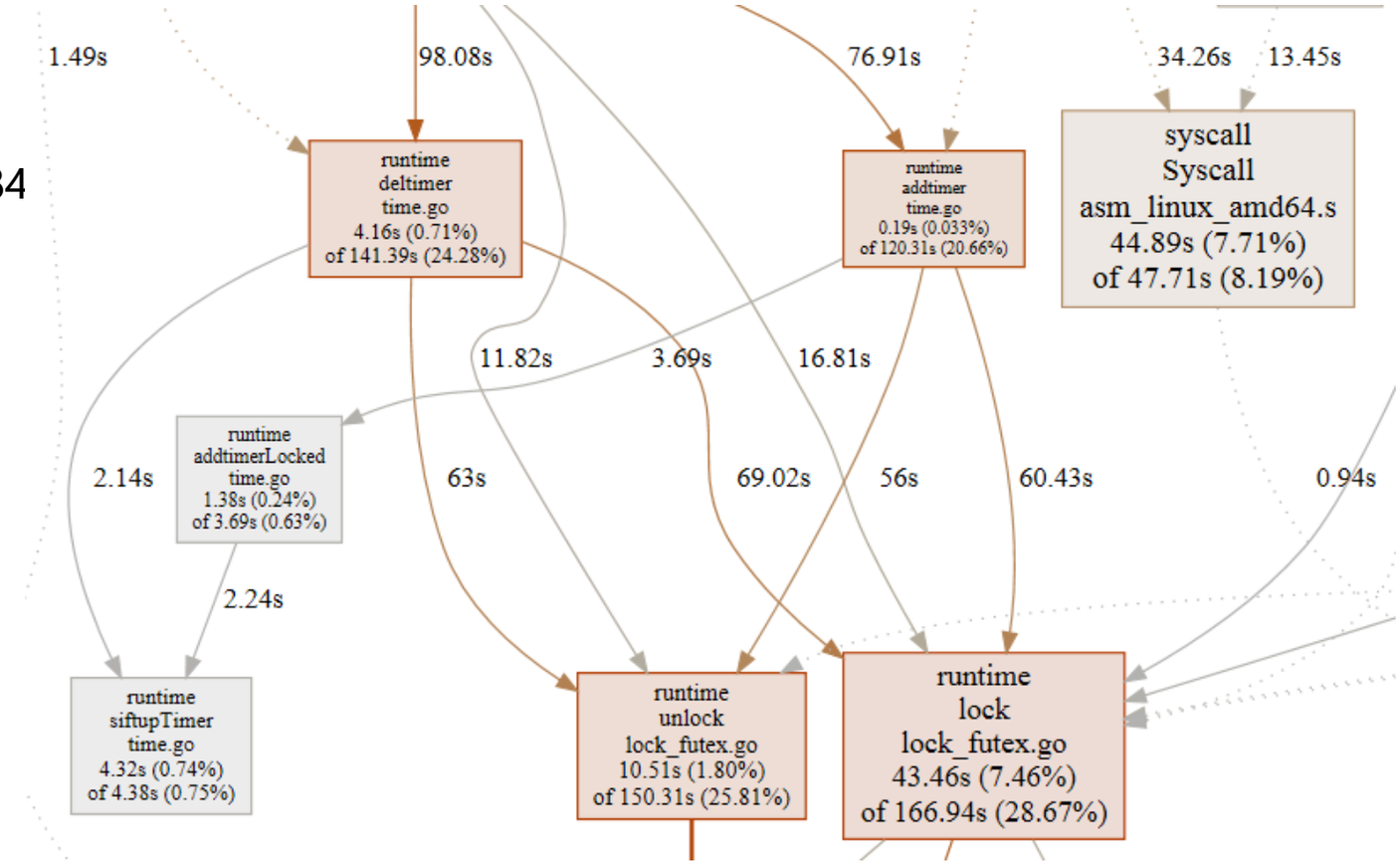
```
_i, _err = _is.Read(reflect.TypeOf(_obj.M_iVersion), 1, true)
if _err != nil {
    return _err
}
if _i != nil {
    _obj.M_iVersion = _i.(int16)
}
_i, _err = _is.Read(reflect.TypeOf(_obj.M_cPacketType), 2, true)
if _err != nil {
    return _err
}
```

```
err = _is.Read_int16(&st.IVersion, 1, true)
if err != nil {
    return err
}

err = _is.Read_int8(&st.CPacketType, 2, true)
if err != nil {
    return err
}
```

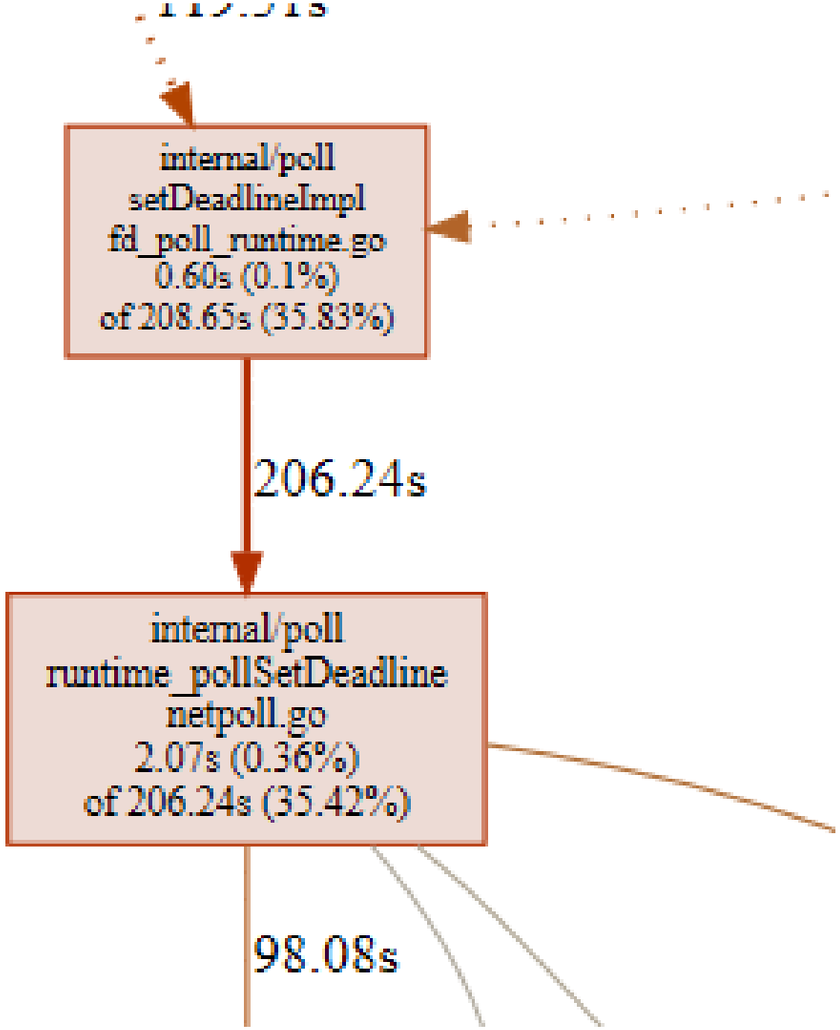
Timer优化

- **问题：**
  - 见 https://go-review.googlesource.com/c/go/+/34784
  - Muti-cpu场景下，timer性能低下
- **解决方案：**
  - 升级Go到最新的1.10.3及以上版本
  - Use per-P timers, so each P may work with its own timers
  - 基于时间轮算法实现自己的timer



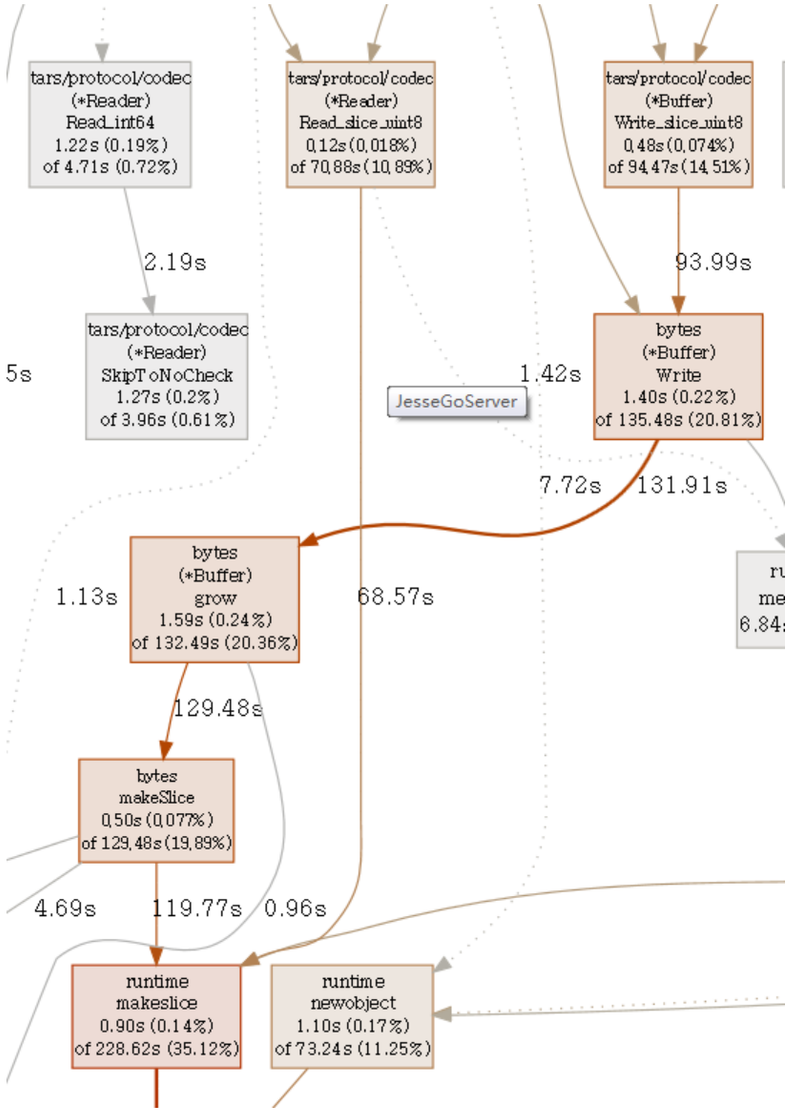
Net包的SetDeadline调用性能问题

- **问题：**
  - 使用pacakge net 的SetDeadline /SetWriteDeadline/SetReadDeadline 等相关调用，当并发很大时，性能低下
- **解决方案：**
  - 通过 Sysfd，设置socket的读写超时



Bytes的Buffer带来的性能问题

- **问题：**
  - Tarsgo使用package bytes的Buffer来做编解码的流的缓冲，当Buffer不断增大时，底层的byte slice需要进行扩容，频繁的的内存分配回收对性能带来较大冲击
- **解决方案：**
  - 使用sync.pool来缓存临时对象，避免频繁内存分配导致的gc
  - 后续使用sync.pool进行封装，对不同size的buffer进行分类缓存。 类似于linux 的Slab分配机制



其它优化

避免使用反射

反射有时候很好用，但往往会成为性能杀手

尽量提前确定好类型

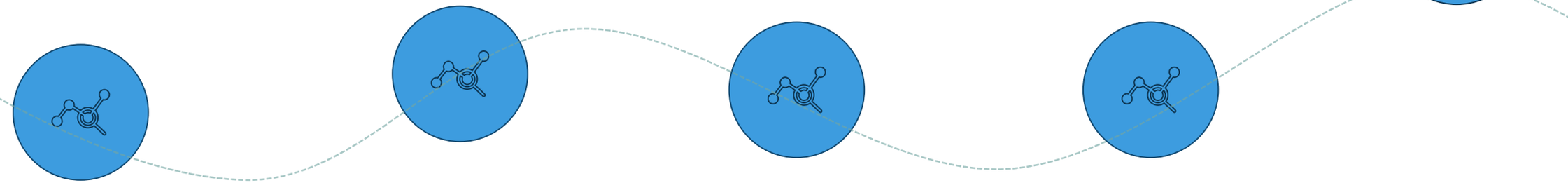
TCP连接优化

采用长连接，设置较大的读写buffer，有选择地设置tcpNodelay

尽量避免多协程竞争chan

chan在读取和写入的时候，存在锁的开销，当并发大时，会影响整体性能

个别场景用原子加减替代



采用协程池

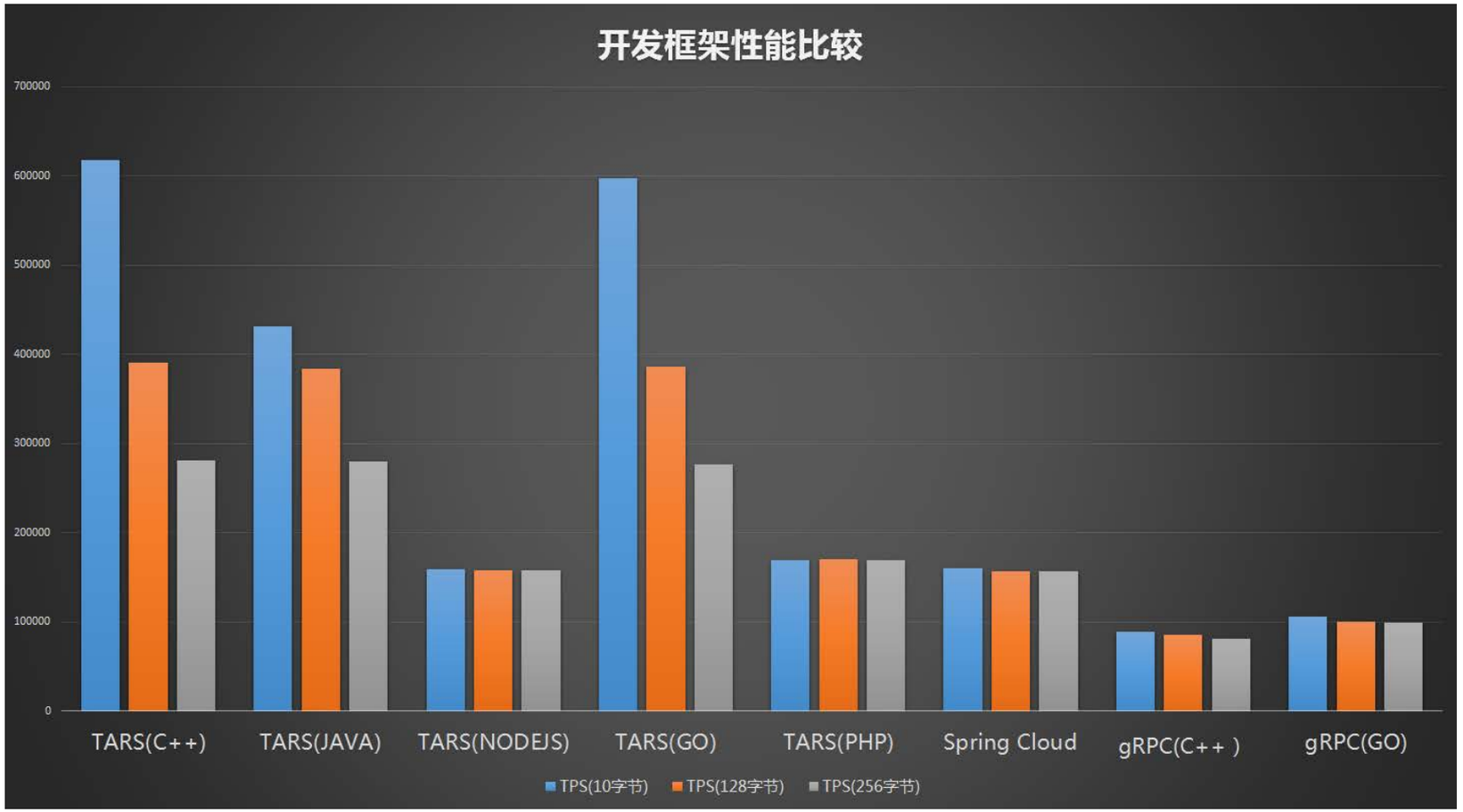
Go的协程虽然足够轻量和高效，但高并发下频繁的创建和销毁协程，还是会带来一定的性能损耗

使用指针而不是值传递

特别是大结构体，在函数设计的时候，函数参数使用指针而不是值传递，会有不错的性能提升

Tars性能数据

- 压测机型： 4 核 /8线程CPU 3.3Ghz 主频 16G内存
- 压测逻辑：客户端带着一定大小的数据给服务端，服务端原样返回给客户端 。
- 服务端单进程，多个客户端发起测试



SECTION FIVE

TARS应用案例及未来规划

TARS应用案例及未来规划

- Tars在腾讯内部使用十年，并于2017年4月10日开源，开源后与业界众多企业交流，同时也得到了广泛应用。
- 开源地址： https://github.com/TarsCloud
- 2018年6月加入Linux 基金会
- Go语言开源： https://github.com/TarsCloud/TarsGo

扫码添加**TARS小助手**为好友，加入TARS GO官方交流群

（PS：申请时请备注**公司企业名称+开发语言**）