大型微服务框架设计实践

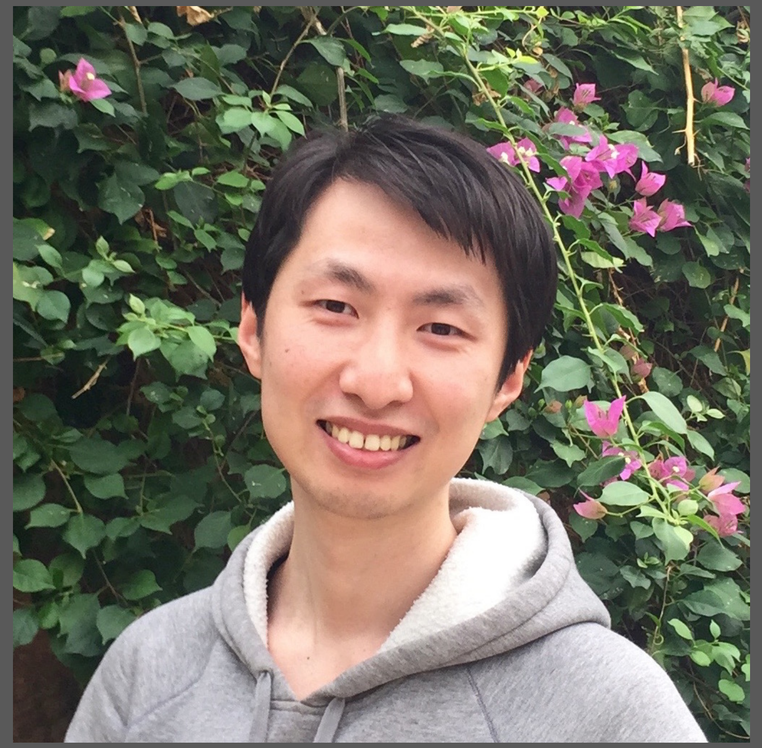
杜欢

滴滴出行 R lab

duhuan@didiglobal.com

自我介绍

- 工作经历
  - 2015 年至今：历任滴滴出行平台产品中心技术负责人、出行创新业务技术负责人、R lab 配送业务技术负责人
  - 2010 年至 2015 年：自主创业，作为创始人和 CTO，专注于游戏领域创新项目研发落地
  - 2007 年至 2010 年：先后在微软和百度任职
- https://github.com/huandu



大纲

- 发现问题：服务开发过程中的痛点
- 以史鉴今：从服务框架的演进历程中找到规律
- 大道至简：大型微服务框架的设计要点
- 精雕细琢：框架关键实现细节

复杂业务开发过程中的痛点

- 痛点
  - 时间紧、任务多、团队大、业务增长快，如何还能保证架构稳定可靠?
  - 研发水平参差不齐、项目压力自顾不暇，如何保证质量基线不被突破?
  - 公司有各种工具平台、SDK、最佳实践，如何尽可能的在业务中使用?
- 用什么“框架”可以解决问题?

大纲

- 发现问题：服务开发过程中的痛点
- 以史鉴今：从服务框架的演进历程中找到规律
  - 服务框架的进化史
  - 服务框架的演进趋势
- 大道至简：大型微服务框架的设计要点
- 精雕细琢：框架关键实现细节

服务框架的进化史


Pyramid
ZEND FRAMEWORK
Tornado
STRUTS
CppCMS -Faster -Lighter -Greener
BEEGO
Symfony
yiiframework
CakePHP Build fast, grow solid.
CODEIGNITER
sails
Istio
Connect, secure, control, and observe services.
django
Express
koa
Sinatra
ASP Active Server Pages
spring
WEB2PY
laravel
GRPC
Apache Thrift ™
Microsoft
ASP.NET
protobuf Protocol Buffers
Revel
php
RAILS
Flask web development, one drop at a time
METEOR
1995
2000
2005
2010
2015
2019

标志性的服务框架

- Web 服务框架：MVC 架构
  - ASP.Net（since 2002）：传统 C/S 开发模式在 Web 上的应用
  - Ruby on Rails（since 2005）：MVC 框架的巅峰，“约定大于配置”
- Web 服务框架：SaaS 与 RESTful
  - Sinatra（since 2007）：纯路由框架，诸多框架的灵感源泉
- 微服务框架：RPC 服务
  - Thrift（since 2007）：开源 IDL-based 框架的鼻祖
- 微服务架构：容器化与 FaaS
  - Serverless（since 2015）：基于云计算平台，回归框架本质
  - Istio（since 2018）：专注于解决网络问题

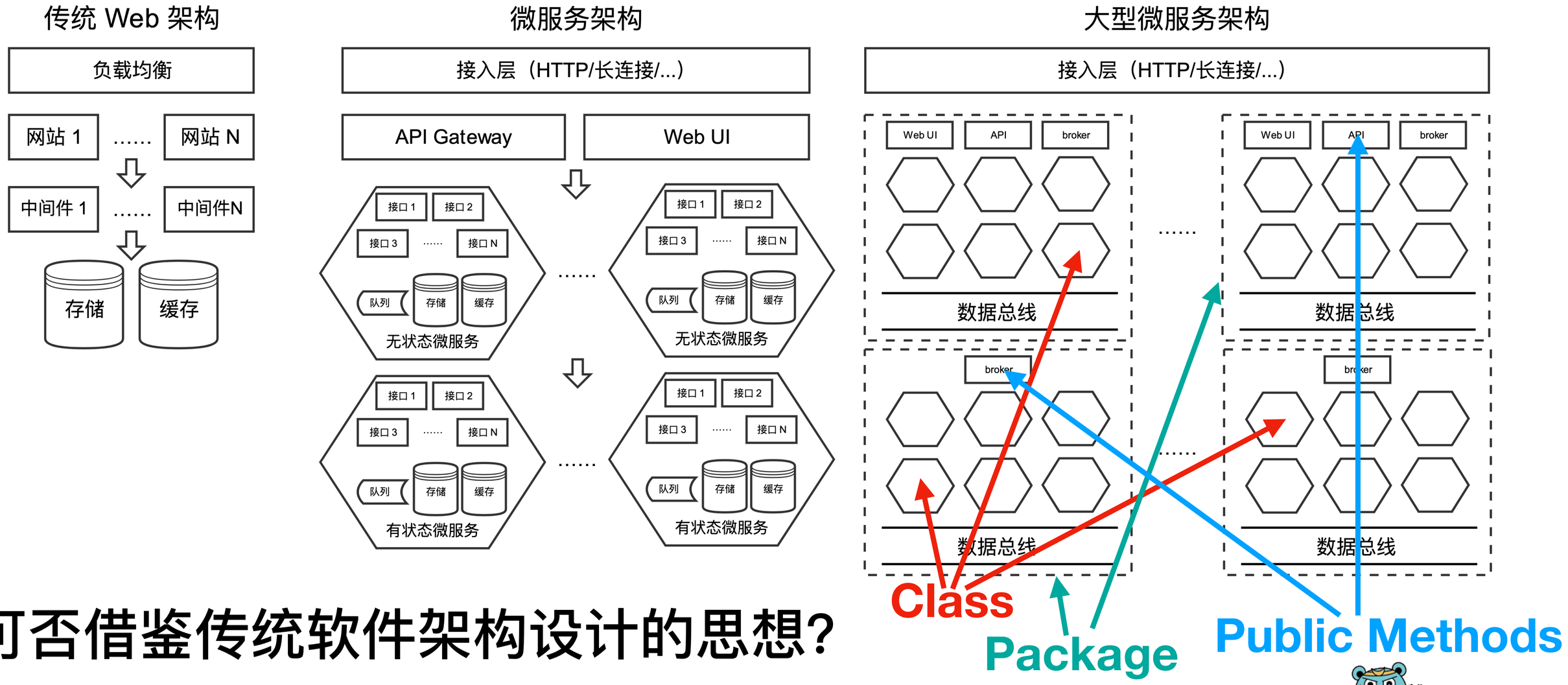
服务框架的演进趋势

- 服务框架正在演变成新的“操作系统”
  - 学习曲线：Exponential Rise（渐进式） → Sigmoid（阶跃式）
  - 风格：配置 → 约定 → DSL → 容器化
  - 业务代码与框架代码的关系：Is-a → Has-a → Duck-typing
  - 工具链：IDE → 代码生成器 → 编译器

大纲

- 发现问题：服务开发过程中的痛点
- 以史鉴今：从服务框架的演进历程中找到规律
- 大道至简：大型微服务框架的设计要点
  - 站在全局视角观察微服务架构
  - 大型微服务框架的设计目标
  - 大型微服务框架的设计要点
- 精雕细琢：框架关键实现细节

站在全局视角观察微服务架构



可否借鉴传统软件架构设计的思想?

大型微服务框架的设计目标

- 框架即一款面向开发人员的**效率产品**，基于公司的基础设施**量身定制**
  - 目标用户：来自不同背景、具有基本业务研发能力的开发者
  - 设计要点：让开发人员专注于业务开发本身，无需关注滴滴各种基础设施底层细节
  - 设计原则：直观、简洁、智能、个性化
  - 预期收益：提升人效，降低维护成本；提升整体架构稳定性和可伸缩性；简化技术升级难度

Rule of least power

by TimBL, the inventor of WWW

https://en.wikipedia.org/wiki/Rule_of_least_power

探探 Gopher China 2019

大型微服务框架的设计要点

- 完全屏蔽业务无关的通用技术细节
  - 功能：服务治理、虚拟化、水平扩容、问题定位、性能压测、系统监控、兼容遗留系统……
  - 工具链：项目模板、代码生成器、文档生成器、发布打包脚本……
  - 设计风格：Interceptors、组合模式、依赖注入……
- 让不可靠的调用变得可靠
  - RPC 调用 ≈ 函数调用
  - 访问基础服务 ≈ 访问本地存储
  - 服务拆分/合并 ≈ 类拆分/合并

大纲

- 发现问题：服务开发过程中的痛点和痒点
- 以史鉴今：从服务框架的演进历程中找到规律
- 大道至简：大型微服务框架的设计要点
- 精雕细琢：框架关键实现细节
  - 业务实践
  - 整体架构
  - 实现要点
    - 框架与业务正交
    - 隔离层屏蔽业务与底层的联系
    - 协议劫持
    - 跨服务边界的 context
    - 防雪崩
  - 业务收益

业务实践

- 业务背景：复杂的业务流程，快速增长与迭代，异构服务架构，跨国多机房部署
- 核心能力
  - 隔离层封装：各种存储、队列、平台服务封装
  - 透明支持各种运维基础设施：构建、发布、多机房配置、metrics
  - 提供效率和测试工具：日志采集、问题自动跟踪、全链路压测、mock、接口测试
  - 应用层协议隔离：支持 thrift/http 协议 interceptor
  - 工具链：模板、代码生成器、依赖管理、版本管理、发布脚本

滴滴在 Go 方向拥有丰富的人才和技术储备

1500+
Go 模块

2000+
Go 服务实例

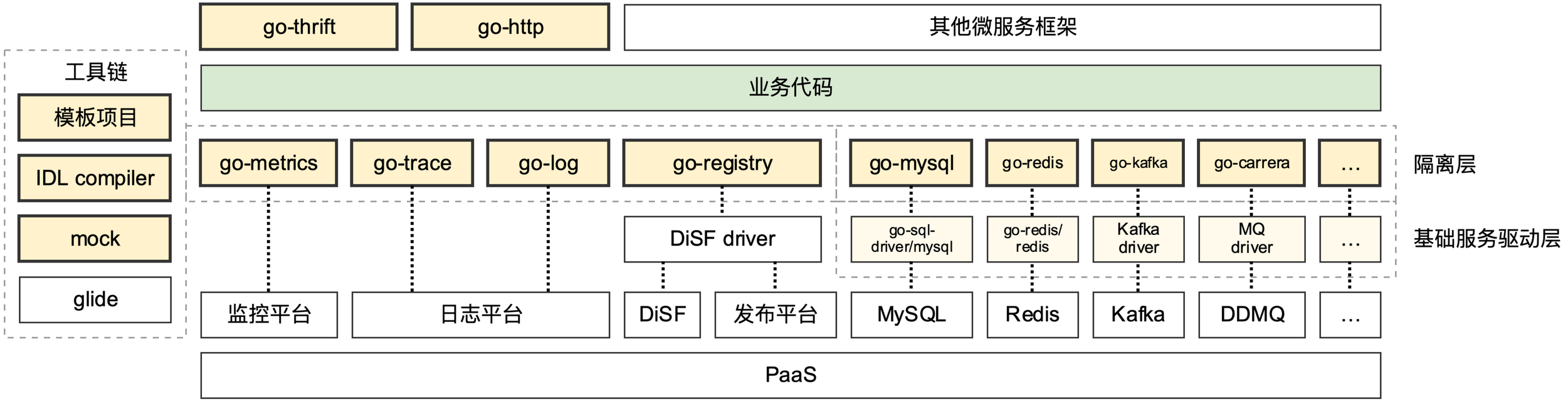
1000+
Go 开发者

这使得我们选用 Go 作为最核心的开发语言

站在巨人肩膀上：滴滴基础平台建设现状

- Odin：运维平台，提供 metrics 上报、多维度监控、报警、服务树等功能
- 把脉：日志平台，提供日志采集通道、基于 traceid 的全链路日志查询能力
- DiSF：服务注册平台，提供高可用的服务名字服务、管理服务分组
- RDS：提供高可用、透明水平扩展的 MySQL 集群，支持数据总线、分身等能力
- DDMQ：低延迟高可用的消息队列服务，单机 TPS 吞吐超过百万，支持延时消息
- Fusion：基于 rocksdb 的高性能高可用分布式持久化存储方案，完全兼容 Redis 协议
- 弹性云：基于 k8s，高效、可伸缩的集群管理平台，服务自动容错，基础设施免运维

整体架构



核心框架/工具链　　公司基础平台能力　　业务代码

实现要点：框架与业务正交

- 实现思路
  - 传统框架的 MVC、middleware、AOP、执行流程……都不存在也不需要
  - 框架是一个执行环境，由一堆不关联的基础库组成高度可扩展，业务可独立于框架运行
- 如何实现
  - 提供工具链，用于生成最初的项目模板并通过代码生成器实现类似 AOP 的效果
  - 基于 Go interface 的 duck-typing 特性和运行时反射，动态生成业务路由
- 收益
  - 业务开发无需关注框架本身，
  - 框架本身的升级可以做到完全透明，方便所有服务统一框架版本

框架的启动逻辑

```
// 使用上面读出来的配置来初始化 server。
s := server.New(config.ThriftServer)

// service 实现了某个具体服务 IDL service 接口，每个具体服务的类型都不一样。
handler := service.New()

// 使用 handler 和对应的应用层协议的 processor factory 方法来启动服务。
// 注意 s.ServeHandler 调用之后代码就不会继续执行，直到收到系统信号后会停止 thrift server。
s.ServeHandler(handler, server1.NewServer1Processor)
```


实现要点：隔离层屏蔽业务与底层的联系

- 如何实现
  - 为所有基础服务（mysql/redis/mq/es/...）定义 interface，业务只允许调用 interface 的方法
  - 基于 SPI 设计思路，提供基础服务的工厂，动态实例化对应 interface
- 收益
  - 可透明的升级服务驱动，快速在大量服务中实现共性逻辑或者修复共性问题
  - 透明的管理基础服务的资源（长连接、mysql cursor 等），避免出现资源泄露
  - 统一控制重试、超时、服务发现、故障摘除逻辑，业务无感知且不易出错，提升整体稳定性
  - 对所有基础服务提供了 mock 能力，可以实现 AOP 能力

案例：Redis 接口设计

githbu.com/go-redis/redis

```
type Cmdable interface {
        Pipeline() Pipeliner
        Pipelined(fn func(Pipeliner) error) ([]Cmder, error)

        TxPipelined(fn func(Pipeliner) error) ([]Cmder, error)
        TxPipeline() Pipeliner

        ClientGetName() *StringCmd
        Echo(message interface{}) *StringCmd
        Ping() *StatusCmd
        Quit() *StatusCmd
        Del(keys ...string) *IntCmd
        Unlink(keys ...string) *IntCmd
        Dump(key string) *StringCmd
        Exists(keys ...string) *IntCmd
        Expire(key string, expiration time.Duration) *BoolCmd
        ExpireAt(key string, tm time.Time) *BoolCmd
        Keys(pattern string) *StringSliceCmd
```

我们的 Redis 隔离层

```
type Redis interface {
    Close() error

    Append(key string, value string) (reply int, err error)
    BitCount(key string) (bits int, err error)
    BitCountWithRange(key string, start int, end int) (bits int, err error)
    BitOp(op Op, destKey string, key ...string) (size int, err error)
    BitPos(key string, bit int) (pos int, err error)
    BitPosWithStart(key string, bit int, start int) (pos int, err error)
    BitPosWithRange(key string, bit int, start int, end int) (pos int, err error)
    BLPop(timeout time.Duration, key ...string) (values []Value, err error)
    BRPop(timeout time.Duration, key ...string) (values []Value, err error)
    DBSize() (size int64, err error)
    Decr(key string) (result int64, err error)
    DecrBy(key string, decrement int64) (result int64, err error)
    Del(key ...string) (removed int, err error)
    Dump(key string) (value Value, err error)
    Echo(msg string) (sent Value, err error)
```


实现要点：协议劫持

- 如何实现
  - HTTP 协议：包装 http.Handler，用责任链模式处理 http.Request 和 http.ResponseWriter
  - RPC 协议：劫持协议序列化流程，用 FSM（有限状态机）来跟踪序列化过程并适时修改数据
- 收益
  - 将业务数据和服务框架数据充分隔离，避免互相干扰
  - 实现接口热补丁和 in/out 数据录制与重放，方便测试
  - 可透明的增强服务能力，为实现跨服务边界的 context 打好基础

使用 FSM 劫持 thrift protocol

- FSM 实现思路
  - 利用 Go interface 特性，实现一个 interface proxy，代理并劫持部分感兴趣的接口
  - 维护一个 FSM 状态机，当 protocol read/write 走到感兴趣的地方时候篡改 read/write 数据

```go
// Protocol 是一个根据 soda 通信协议劫持了客户端请求的 TProtocol 包装。
// 由于客户端无法区分 input/output protocol，这里同时劫持了读和写。
type Protocol struct {
        thrift.TProtocol
        Context           context.Context
        OnWriteMessageEnd func(protocol *Protocol, err error) // 在 WriteMessageEnd 调用时被调用。
        OnReadMessageEnd  func(protocol *Protocol, err error) // 在 ReadMessageEnd 调用时被调用。

        readState  int
        writeState int

        trace     trace.Trace
        errno     int32
        msg       string
        hasErrno  bool
        cachedKey string
}
```


实现要点：跨服务边界的 context

- 如何实现
  - 实现符合 context.Context 接口的自定义 context，支持序列化与反序列化，支持超时控制
  - 结合协议劫持，透明的从服务框架数据中提取必要信息进行反序列化，并在所有 RPC 调用前透明的将最新 context 序列化并放在服务框架数据中传输给下游
  - 需要重新实现一个基于时间片轮转的低精度 time.Timer，提升并发效率并避免 timer 泄露
- 收益
  - 可透明的在服务间传递上下文信息，从而实现流量染色、调用跟踪、防雪崩等功能

低精度 timer 实现原理

```
// Pool 是一个用来存储并跟踪所有 timer 的池子，通过降低 timer 的精度来尽量的复用 chan，
// 降低内存碎片和提升性能。
//
// 这里实现的是一个 Hashing Wheel with Ordered Timer Lists，算法详见：
// https://blog.acolyer.org/2015/11/23/hashed-and-hierarchical-timing-wheels/
type Pool struct {
        mu            sync.Mutex
        timerChanPool sync.Pool

        precision  int64
        timerChans [ringBufferSize]*timerChan
        timerCnt   int64
        loopStart  chan struct{}
}
```

```
type timerChan struct {
        tick int64
        next *timerChan
        done chan struct{}
}
```


实现要点：防雪崩

- 如何实现
  - 通过跨服务边界的 context 来传递上游超时预期，并不断记录各个环节耗时
  - 一旦框架发现自己的可用时间已经耗尽，主动终止后续 rpc 调用并快速返回，防止请求积压
- 收益
  - 从根本上避免请求堆积造成的雪崩

跨服务边界的超时时间控制

- 超时信息如何跨服务边界传递
  - 超时时间由最上游设置，框架捕捉到超时信息并将时间记录在 trace 里透明的传播到下游每一个服务节点
  - 每个节点从接收到请求开始后计时，自动计算自己消耗的时间并计算当前调用链路总耗时
  - 当链路总耗时超过超时时间，自动 fail-fast，快速返回失败信息
  - 利用 Go context deadline 只会缩短不会提前的特性，方便的用 context 管理超时
  - 避免服务器之间的时钟差异影响计时，始终使用时间差来记录号是，而不使用绝对 deadline

业务收益

- 支撑规模
  - 涉及开发人员近 100 人，上线 70+ 服务，国内外双机房部署
  - 可支撑百万级日订单规模、万级并发长连接
- 业务收益
  - 零成本：透明接入公司运维、发布、日志、压测等平台，支持服务注册发现、弹性伸缩等能力
  - 零事故：从未出现因为单点故障造成的全局稳定性事故
  - 高质量：快速实现全链路压测常态化，透明实现多环境部署、单测、集成测试等
  - 易维护：透明升级各种 driver，简化代码依赖管理过程

未来计划

- 提升开发者体验
  - 命令行工具，用于整合各种工具
  - 进一步与滴滴线上线下环境整合
  - 整合更多的公司服务和框架
  - 配置管理中心化
- 开源?

Thanks

We make it real with ❤️

探探 Gopher China 2019