



INTRODUCTION
TO OPEN SOURCE
PROJECTS

CloudWeGo 开源项目介绍

CLOUDWEGO →

目录

第一章 项目介绍

1.1 CloudWeGo 的项目和用户	01
1.2 CloudWeGo 如何帮助企业用户 解决微服务构建中遇到的问题	02
1.2.1 提供免费企业用户支持	

第二章 不同场景下的解决方案

2.1 Golang RPC 框架 Kitex	03
2.1.1 架构设计（框架特点/ 框架性能 / 扩展能力）	
2.1.2 如何使用 Kitex 与内部基础设施集成	
2.1.3 Kitex 示范demo/ 使用案例	
- 扩展 demo 示例	
- Kitex + k8s 架构帮助森马解决线上双11电商性能瓶颈	
- Kitex 带来ROI更高的的混合云部署下的链路追踪方案	
2.2 Golang HTTP 框架 Hertz	13
2.2.1 架构设计（框架特点/ 框架性能 / 扩展能力）	
2.2.2 为什么 Hertz 性能更优？字节跳动内部 Go HTTP 框架的变迁	
2.2.3 Hertz 示例 demo / 使用案例	
- 扩展 demo 示例	
- 字节服务网格使用 Hertz 的落地带来的收益	
2.3 Kitex 与 Hertz 的工程实践案例	23
2.3.1 Kitex Proxyless 之流量路由：配合 Istio 与 OpenTelemetry 实现全链路泳道	
2.3.2 使用 Hertz、Kitex 重写经典的 Istio Bookinfo 项目	
2.4 Rust 首选 RPC 框架 Volo	33
2.4.1 架构设计（框架特点/ 框架性能 / 扩展能力）	

| 项目介绍

CloudWeGo 是一套由字节跳动开源的、可快速构建企业级云原生微服务架构的中间件集合。
在 **github** 开源后获得超过 1w 的 star 和超 142 位贡献者。

项目官网: <https://www.cloudwego.io/>

github: <https://github.com/cloudwego>



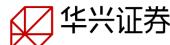
● CloudWeGo 开源的主打项目

- 1.Golang RPC 框架: Kitex (目前已是字节内部使用最广泛的 RPC 框架, 高峰 QPS 达到数十亿)
- 2.Golang HTTP 框架 : Hertz (超大级企业 HTTP 框架, 适用于网关、服务网格等多场景)
- 3.Rust RPC 框架: Volo (Rust 首选 RPC 框架)

| CloudWeGo 的用户

CloudWeGo 致力于帮助更多的企业用户，解决微服务构建中遇到的问题，落地自己的一套云原生微服务架构。

CloudWeGo 项目除在字节内部被大量使用外，外部企业用户：森马、华兴证券、贪玩游戏、Moonton、禾多科技 均在使用。



| CloudWeGo 如何帮助企业用户 解决微服务构建中遇到的问题

● 高易用

提供高性能的框架项目，让用户不用关注其中过多技术细节，同时还提供开箱即用的脚手架工具，外加配套齐全的扩展能力。（后文详细介绍）

● 高可靠

所有开源在 CloudWeGo 项目中的特性，都经过在字节内部的可靠性使用验证。不会随意开源特性进入开源库。

内外一致。开源项目版本和字节内部使用版本一致，统一维护，不维护两套代码。可靠性同时获得外部验证



CloudWeGo 企业支持

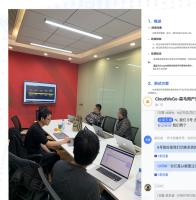
● 免费提供企业技术支持

CloudWeGo 免费为企业用户提供专属飞书群问题跟进解决机制，帮助企业用户使用和落地

在使用过程中或希望使用不知道如何下手的用户，可扫码填写表单，CloudWeGo 团队将提供免费 1v1 飞书企业群技术支持



扫一扫



CloudWeGo 团队为森马电商提供技术支持的场景

| 不同场景下的解决方案

● Golang RPC 框架 Kitex

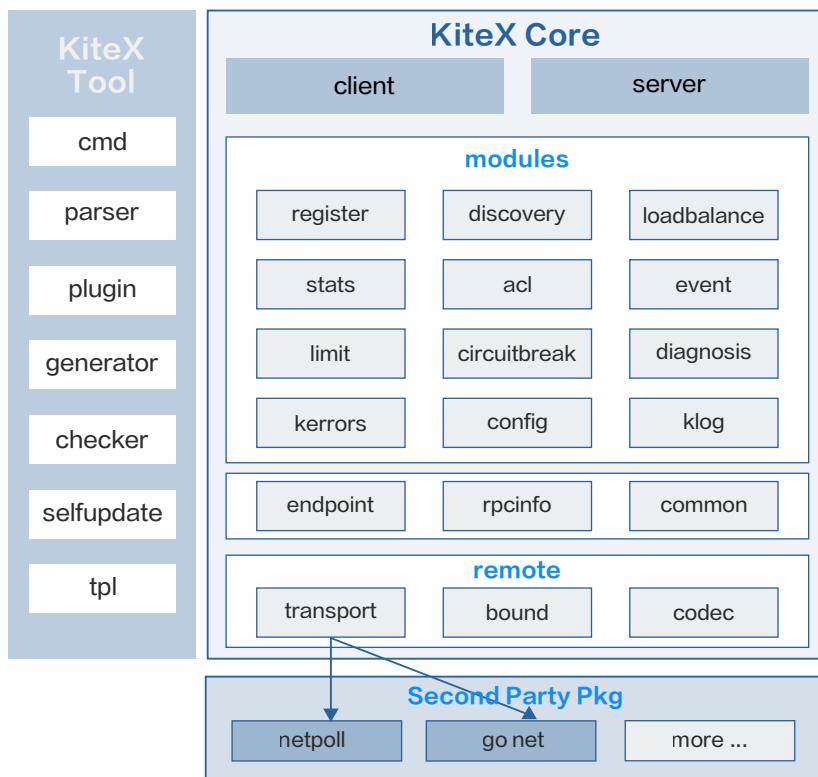
Github 地址: <https://github.com/cloudwego/kitex>

文档: <https://www.cloudwego.io/zh/docs/kitex/overview/>

● 架构设计

Kitex 的架构主要包括四个部分: Kitex Tool、Kitex Core、Kitex Byted、Second Party Pkg

- Kitex Core 是一个携带了一套微服务治理功能的 RPC 框架, 它是 Kitex 的核心部分。
- Kitex Tool 是一个命令行工具, 能够在命令行生成我们的代码以及服务的脚手架, 可以提供非常便捷的开发体验。
- Second Party Pkg, 例如 netpoll, netpoll-http2, 是 Kitex 底层的网络库, 这两个库也开源在 CloudWeGo 组织中。



● 框架特点

► 高性能

使用自研的高性能网络库 Netpoll，性能相较 go net 具有显著优势。

► 扩展性

提供了较多的扩展接口以及默认扩展实现，使用者也可以根据需要自行定制扩展，具体见下面的框架扩展。

► 多消息协议

RPC 消息协议默认支持 Thrift、Kitex Protobuf、gRPC。Thrift 支持 Buffered 和 Framed 二进制协议；Kitex Protobuf 是 Kitex 自定义的 Protobuf 消息协议，协议格式类似 Thrift；gRPC 是对 gRPC 消息协议的支持，可以与 gRPC 互通。除此之外，使用者也可以扩展自己的消息协议。

► 多传输协议

传输协议封装消息协议进行 RPC 互通，传输协议可以额外透传元信息，用于服务治理，Kitex 支持的传输协议有 TTHeader、HTTP2。TTHeader 可以和 Thrift、Kitex Protobuf 结合使用；HTTP2 目前主要是结合 gRPC 协议使用，后续也会支持 Thrift。

► 多种消息类型

支持 PingPong、Oneway、双向 Streaming。其中 Oneway 目前只对 Thrift 协议支持，双向 Streaming 只对 gRPC 支持，后续会考虑支持 Thrift 的双向 Streaming。

► 服务治理

支持服务注册/发现、负载均衡、熔断、限流、重试、监控、链路跟踪、日志、诊断等服务治理模块，大部分均已提供默认扩展，使用者可选择集成。

► 代码生成

Kitex 内置代码生成工具，可支持生成 Thrift、Protobuf 以及脚手架代码。

生成代码结构说明 https://www.cloudwego.io/zh/docs/kitex/tutorials/code-gen/code_generation/#%E7%94%9F%E6%88%90%E4%BB%A3%E7%A0%81%E7%9A%84%E7%BB%93%E6%9E%84

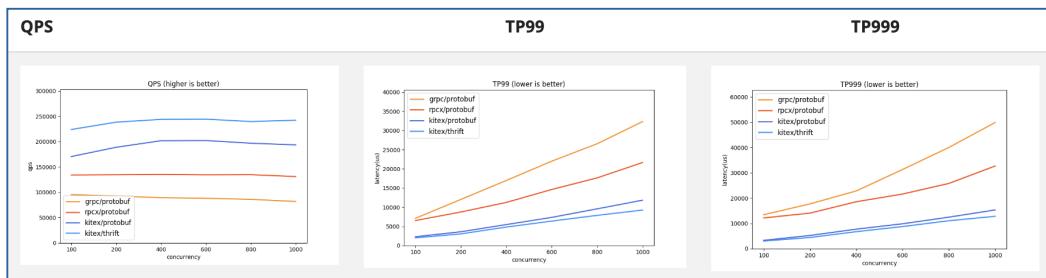
● 框架性能

性能测试只能提供相对参考，工业场景下，有诸多因素可以影响实际的性能表现。

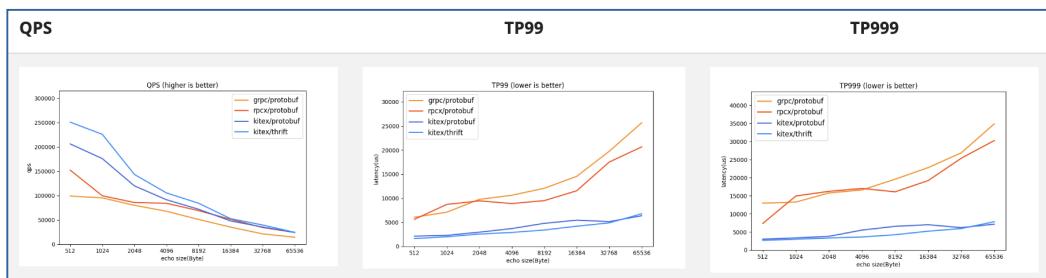
由于开源社区缺少支持 thrift 的优秀 RPC 框架，当前对比项目为 grpc, rpcx, 均使用 protobuf 协议。

我们通过 测试代码 比较了它们的性能，测试表明 **Kitex** 具有明显优势。

▶ 并发表现 (Echo 1KB, 改变并发量)



▶ 吞吐表现 (并发 100, 改变包大小)



▶ 测试环境

- CPU: Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz, 4 cores
- Memory: 8GB
- OS: Debian 5.4.56.bsk.1-amd64 x86_64 GNU/Linux
- Go: 1.15.4。

▶ 扩展能力

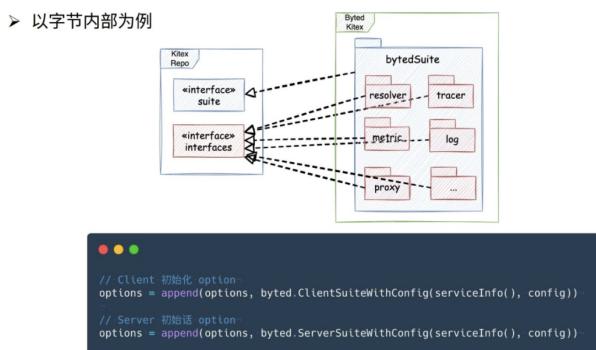
Kitex-contrib: <https://github.com/kitex-contrib>

Kitex-example : <https://github.com/cloudwego/kitex-examples>

● 如何使用 Kitex 与内部基础设施集成

以字节内部为例，内部仓库里有开源库中的扩展实现，集成内部的能力，在 byted-Suite 中，我们针对不同场景对 Kitex 进行初始化。如下面的代码示例，用户只需要在构造 Client 和 Server 时增加一个 option 配置就可以完成集成，不过为了让用户完全不需关注内部能力的集成，我们将该配置放在了生成的脚手架代码中，关于配置如何内嵌在生成代码中，后续我们也会开放出来，方便外部的框架二次开发者能以同样的方式为业务开发同学提供集成能力。

▶ 并发表现 (Echo 1KB, 改变并发量)



● Kitex 示范 demo / 使用案例

▶ 1. 扩展 demo 示例

<https://github.com/cloudwego/kitex-examples>

▶ 2. Kitex + k8s 架构帮助森马解决线上双11电商性能瓶颈

案例地址：<https://www.cloudwego.io/zh/cooperation/semir/>

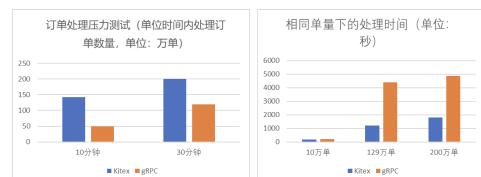
● 面临的主要问题：

- **高并发**。在电商业务场景下，不管是面向用户，比如秒杀，还是面向业务，比如订单处理，如果实现不了高并发，系统就很难做大，很难适应业务的增长。
- **高性能**。除了用高并发来实现业务的快速处理外，性能也是一个挑战。例如在当前疫情状态下，各行各业都在降本增效，解决不了性能问题，就会不断地增加服务器资源，大大增加企业成本。
- **技术保障**。电商行业的公司，大多资源和精力都在销售端，运营端，技术方面投入相对薄弱。因此在技术选型上需要从可靠、安全、支持等维度去考量。

● Kitex 接入 Istio 性能压测对比

我们将 Kitex 和 gRPC 在以下相同服务器硬件资源和网络环境下进行了压测对比：

- 压测工具：JMeter；
- 阿里云 ECS (8 vCPU, 16 GiB, 5 台)；
- 集群：Kubernetes 1.20.11；
- 服务网格：Istio v1.10.5.39。



通过对比发现，在指定时间相同的情况下，Kitex 在单位时间内处理订单数量更多。在指定订单数量的情况下，Kitex 对于处理相同数量的订单所需时间更短，且订单量越大，这种性能差别越明显。总体来看，Kitex 在处理大批订单时优势还是非常突出的。

● Kitex 产生性能优势的原因

CloudWeGo 团队来森马做技术支持时讲到对自研网络库 Netpoll 做了一些性能优化：

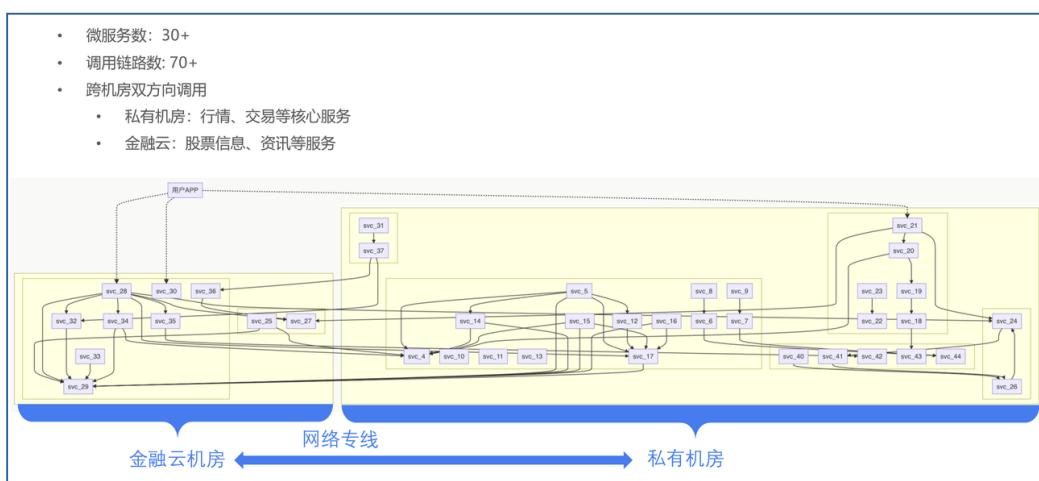
- 连接利用率；
- 调度延迟优化；
- 优化 I/O 调用；
- 序列化/反序列化优化；

► 3. Kitex 带来 ROI 更高的混合云部署下的链路追踪方案

案例地址：<https://www.cloudwego.io/zh/cooperation/huaxingsec/>

● 面临的主要问题：

下图是业务的微服务调用关系图，一共有三十多个微服务，调用链路数超过 70。服务分别部署在两个机房。核心业务比如交易、行情等部署在私有机房。非核心的业务，比如资讯、股票信息等部署在阿里的金融云，这样能够更好地利用金融云已有的基础设施比如 MySQL、Kafka 等，作为初创团队，能够降低整体的运维压力。考虑到性能以及安全方面的因素，两个机房之间专门拉了专线。服务之间存在一些跨机房的依赖。



● 基于 Kitex 可观测性体系的搭建

-Tracing 选型

服务数多了之后，我们需要一套链路追踪系统来描绘调用链路每个环节的耗时情况。考虑到 Kitex 原生支持 Opentracing，为减少集成成本，我们调研了符合 Opentracing 规范的产品。

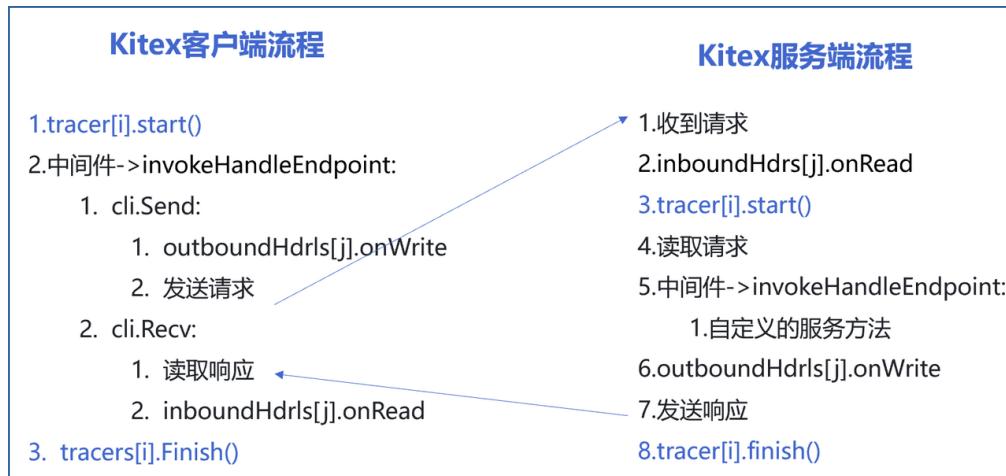
排除掉收费的、客户端不支持 Go 之后，就剩阿里云的链路追踪产品和 Uber 公司出品的 Jaeger，考虑到私有机房也要部署，最终选择了 Jaeger。

-Kitex 接入 Tracing

选定方案之后，开始对 Kitex 的这个功能进行测试，结果发现当时去年 9 月初的 Kitex 版本并不支持跨服务的 Tracing，原因是调用的时候，没有把 Trace 信息发送给下游，如图所示，这样上下游是两个孤立的 Trace (OpenTracing 规范里称为 Span)，于是就无法通过一个 TraceID 去串起整条链路。当时任务比较急，于是我们没有等 Kitex 官方的实现，决定自研。



为了自研，我们结合 Kitex 的源码，梳理出客户端和服务端的流程。可以看出 Kitex 的上下游都内置了 Tracer 的 Hook。这里我们要解决的问题是，如何把 Span 信息进行跨服务传输？



经调研，实现透传有三种方案。

第一种是在消息层搞一个 Thrift 协议的拓展，把 Trace 信息塞进去。原因是 Thrift 本身没有 Header 结构，只能进行协议的拓展。好在 Kitex 支持自定义的协议拓展，因此具备可行性，然而开发成本较高，所以没选择这种方案。

第二种是在 IDL 里增加通用参数，在字段里存 Trace 信息。缺点是业务无关的字段要在 IDL 里，对性能有一定的影响。毕竟需要通过 Kitex 的中间件，通过反射来提取。

第三种是利用了 Kitex 提供的传输层透传能力，对业务没有侵入性。最后选择了这一种方案。

Trace信息跨服务传输方案	说明	Pros	Cons
消息层: thrift协议扩展, 以容纳trace信息	Thrift没有header结构, 只能进行协议扩展 (Kitex支持自定义协议扩展)	对业务本身无侵入性	实现成本较高
消息层: idl里增加通用参数, 例如BaseReq和BaseResp, 在其字段里存trace信息	可通过Kitex的中间件里提取request, 拿到字段	实现成本低	需要修改idl, 新增业务无关的字段, 侵入性强
传输层: 透传trace信息	Kitex提供了 传输层透传能力 (https://github.com/cloudwego/kitex/blob/develop/docs/guide/extension/transmeta_cn.md) , 可利用该特性在传输层设置\读取trace信息	实现成本低, 对业务本身无侵入性	无

透传方案定了之后, 整体的流程就清晰了。首先客户端会在 metaHandler.write 里通过 CTX 获取当前 Span, 提取并写入 spanContext 到 TransInfo 中。

然后服务端, 在 metaHandler.Read 里读取 spanContext 并创建 ChildOf 关系的 Span, 中间件结束时 span.finish(), 最后为了防止产生孤立 Trace, New 服务端时不使用 Kitex 提供的 Tracing 的 Option。

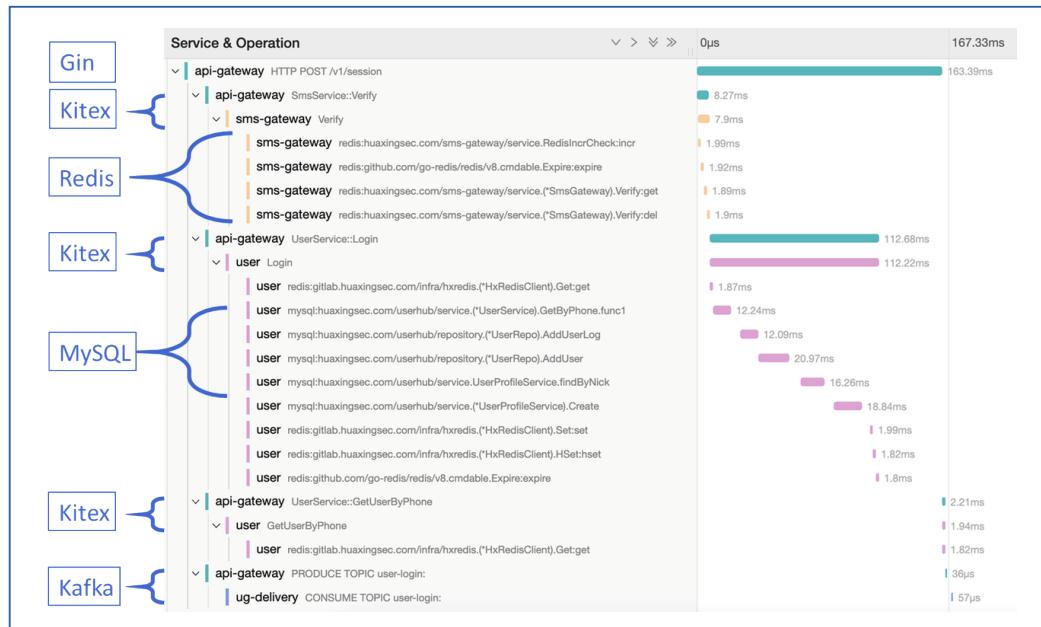
这里是因为同一个服务可能分别作为 Kitex 上下游, Tracer 如果共用, 需要分别加特殊逻辑, 实现上有点复杂。

Kitex客户端流程	Kitex服务端流程
<pre> 1.tracer[i].start() 2.中间件->invokeHandleEndpoint: 1. cli.Send: 1. outboundHdrls[j].onWrite (写入spanContext) 2. 发送请求 2. cli.Recv: 1. 读取响应 2. inboundHdrls[j].onRead 3. tracers[i].Finish() </pre>	<pre> 1.收到请求 2.inboundHdrls[j].onRead (读取透传的trace, 创建span) 3.tracer[i].start() (不注入tracer, 防止产生孤立trace) 4.读取请求 5.中间件->invokeHandleEndpoint: 1.自定义的服务方法 (中间件返回时结束span) 6.outboundHdrls[j].onWrite 7.发送响应 8.tracer[i].finish() (不注入tracer, 防止产生孤立trace) </pre>

-Tracing 基础库

为了充分利用 Tracing 的能力，除了 Kitex，我们在基础库中也增加了 **Gin**、**Gorm**、**Redis**、**Kafka** 等组件的 Tracing。

下面展示实际的一条链路。功能是通过短信验证码进行登录。先是作为 HTTP 服务的 API 入口，然后调用了一个短信的 RPC 服务，RPC 服务里面通过 Redis 来检查验证码。通过之后调用用户服务，里面可能进行一些增加用户的 MySQL 操作。最后把用户登录事件发给 Kafka，然后运营平台进行消费，驱动一些营销活动。可以看出最耗时的部分是关于新增用户的



-对错误的监控

Tracing 一般只关注调用耗时，然而一条链路中可能出现各种错误：

1. Kitex

- Kitex RPC 返回的 err (Conn Timeout、Read Timeout 等)；
- IDL 里自定义的业务 Code (111: 用户不存在)。

2. HTTP

- 返回的 HTTP 状态码 (404、503)；
- JSON 里的业务 Code (-1: 内部错误)。

```
resp, err := s.client.MIsInPortfolio(ctx, req)
if err != nil {
    log.Errorf(ctx, msg: "rpc call err: %v", err)
    return nil, err
}

if resp.BaseResp.Code != base.OK {
    log.Errorf(ctx, msg: "rpc call err: %v", resp.BaseResp)
    return nil, hxerr.New(int(resp.BaseResp.Code), resp.BaseResp.Msg)
}
```

如何对这类错误进行监控？主要有以下三种方案：

1. 打日志 + 日志监控，然后通过监控组件，这种方案需要解析日志，所以不方便；
2. 写个中间件上报到自定义指标收集服务，这种方案优点是足够通用，但是需要新增中间件。同时自定义指标更关注具体的业务指标；
3. 利用 Tracing 的 Tag，这种方案通用且集成成本低。

具体实现如下：

- Kitex 的 err、以及 HTTP 的状态码，定义为系统码；
- IDL 里的 Code 以及 HTTP 返回的 JSON 里的 Code，定义成业务码；
- Tracing 基础库里提取相应的值，设置到 span.tag 里；
- Jaeger 的 tag-as-field 配置里加上相应的字段（原始的 Tags，为 es 里的 Nested 对象，无法在 Grafana 里使用 Group By）。

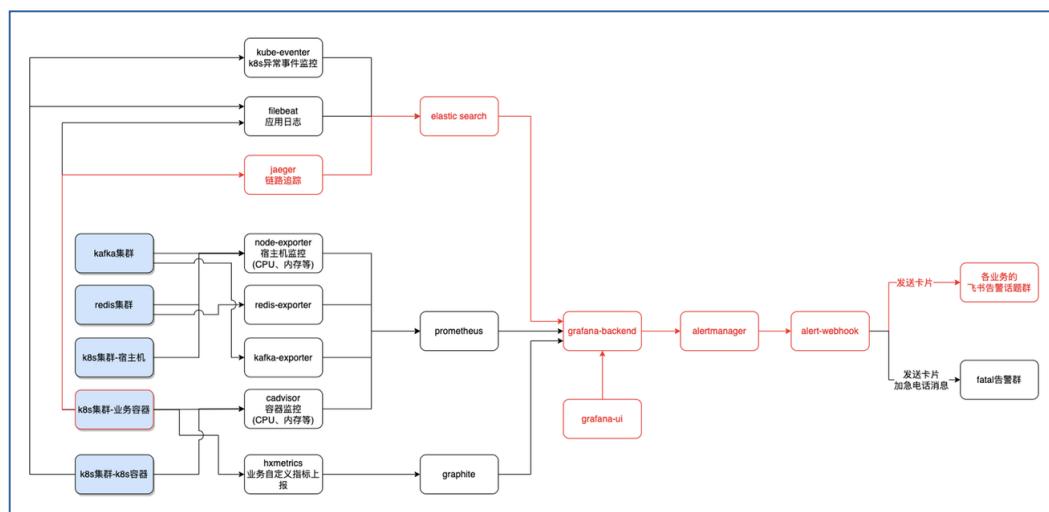
- 监控告警

在增加错误监控的基础上，我们构建了一套监控告警系统体系。

这里重点看一下刚才的链路追踪相关的内容。首先每个业务容器会把指标发送到 Jaeger 服务里。Jaeger 最终把数据落盘到 es 中。然后我们在 Grafana 上配置了一堆看板以及对应的告警规则。

触发报警时，最终会发送到我们自研的 alert-webhook 里。

自研的部分首先进行告警内容的解析，提取服务名等信息，然后根据服务的业务分类，分发到不同的飞书群里，级别高的报警会打加急电话。这里也是用到了飞书的功能。



Grafana 里我们配置了各类型服务调用耗时、错误码一体化看板，描述了一个服务的方方面面的指标。包括日志监控、错误码监控、QPS 和调用耗时、容器事件监控、容器资源监控等。

The screenshot shows the CloudWatch Metrics Dashboard interface. The top navigation bar includes 'namespace' (set to 'production'), 'appname' (set to 'api-gateway'), and a dropdown for '服务名称' (Service Name). The main pane lists various metric panels categorized by service role:

- 作为错误日志、panic、慢SQL日志 (2 panels)
- 作为RPC客户端-连接建立耗时和收发数据大小 (2 panels)
- 作为HTTP服务器-系统码、业务码 (2 panels)
- 作为HTTP客户端-系统码、业务码 (2 panels)
- 作为RPC客户端-系统码、业务码 (2 panels)
- 作为HTTP客户端-请求速率和耗时 (3 panels)
- 作为HTTP服务器-请求速率和耗时 (3 panels)
- 作为RPC客户端-请求速率和耗时 (3 panels)
- 作为RPC服务器-请求速率和耗时 (3 panels)
- 作为MySQL客户端-请求速率和耗时 (3 panels)
- 作为Redis客户端-请求速率和耗时 (3 panels)
- Go服务指标 (4 panels)
- k8s异常事件 (1 panel)

Below the main pane, there are two sections with blue arrows pointing to them:

- 日志监控 (Log Monitoring) pointing to the first two items.
- 错误码监控 (Error Code Monitoring) pointing to the next three items.
- QPS和耗时监控 (QPS and Latency Monitoring) pointing to the remaining items.

At the bottom left, there are buttons for '容器事件 (存活探针失败等) 监控' (Container Event Monitoring) and '容器资源 (CPU内存等) 监控' (Container Resource Monitoring).

下图展示了飞书告警卡片。包括 RPC 调用超时、系统码错误、业务码错误。

这里我们做了两个简单的工作，一个是带上了 TraceID，方便查询链路情况。另一个是把业务码对应的含义也展示出来，研发收到报警之后就不用再去查表了。

● Golang HTTP 框架 Hertz

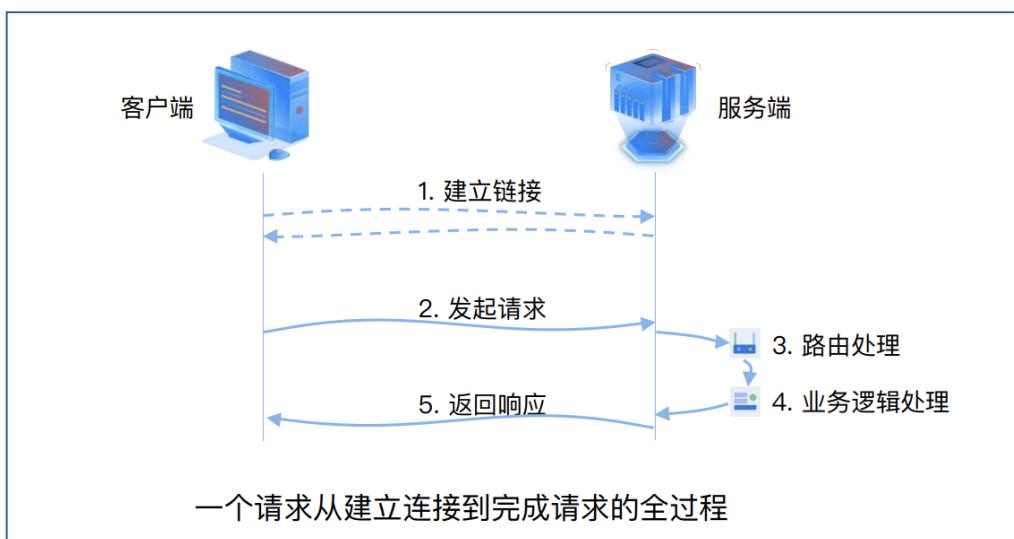
Github 地址: <https://github.com/cloudwego/hertz>

文档: <https://www.cloudwego.io/zh/docs/hertz/overview/>

● 架构设计

下图是一个请求从建立、连接到完成的全过程。左侧是客户端，右侧是服务端，在我们发起链接建立请求之后，链接建立完成；之后客户端发起请求到服务端，服务端进行路由处理，然后将路由导向业务逻辑处理；业务逻辑处理完毕后，服务端返回这个请求，完成一次 HTTP 请求的调用。

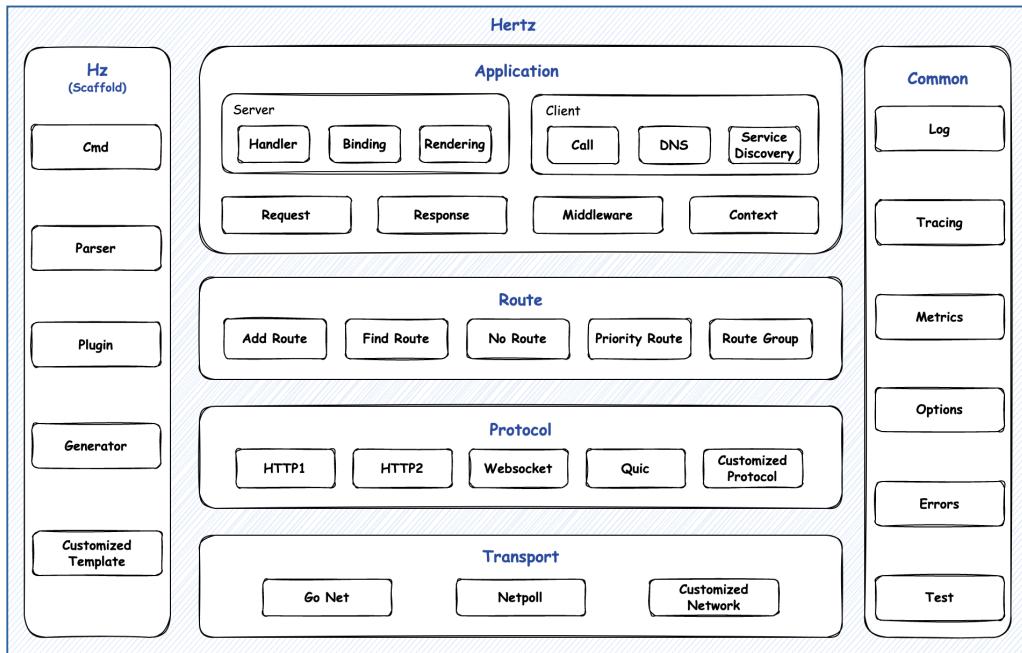
那么在这个过程中我们的框架到底做了哪些事情呢？从图中不难发现，首先框架进行了链接处理，其次是协议处理，之后基于路由做了逻辑分发，即路由处理，最后做了业务逻辑处理。我们把框架做成一个结构之后会发现，这个结构包含的就是这四部分。



基于这个逻辑，我们可以看一下 Hertz 的整体架构图。如下图所示，从下往上看红线框圈住的部分，可以发现这就是上文提到的请求建立的全过程。各层的能力及作用如下：

- 传输层 Transport: 抽象网络接口；
- 协议层 Protocol: 解析请求，渲染响应编码；
- 路由层 Route: 基于URL进行逻辑分发；
- 应用层 Application: 和业务直接交互，提供易用的 API 接口。

我们可以看到图中除了中间部分包含的四层，左右两侧各有两列。右侧是通用层 Common，主要负责提供通用能力、常用的日志接口、链路追踪以及一些配置处理相关的能力等。左侧是 Hertz 的代码生成工具 Hz，又称脚手架工具，它可以帮助我们在内部基于 IDL 快速地生成项目骨架，以加速业务迭代。



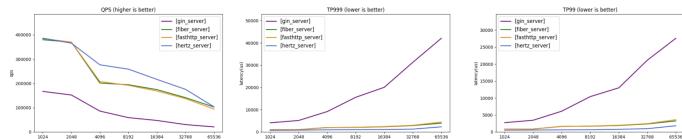
● 框架特点

▶ 高易用性

在开发过程中，快速写出来正确的代码往往是更重要的。因此，在 Hertz 在迭代过程中，积极听取用户意见，持续打磨框架，希望为用户提供一个更好的使用体验，帮助用户更快的写出正确的代码。

▶ 高性能

Hertz 默认使用自研的高性能网络库 Netpoll，在一些特殊场景相较于 go net，Hertz 在 QPS、时延上均具有一定优势。关于性能数据，可参考下图 Echo 数据。



关于详细的性能数据，可参考 <https://github.com/cloudwego/hertz-benchmark>

▶ 高扩展性

Hertz 采用了分层设计，提供了较多的接口以及默认的扩展实现，用户也可以自行扩展。同时得益于框架的分层设计，框架的扩展性也会大很多。目前仅将稳定的能力开源给社区，更多的规划参考 RoadMap。

► 多协议支持

Hertz 框架原生提供 HTTP1.1、ALPN 协议支持。HTTP/2、HTTP/3 即将发布除此之外，由于分层设计，Hertz 甚至支持自定义构建协议解析逻辑，以满足协议层扩展的任意需求。

► 网络层切换能力

Hertz 实现了 Netpoll 和 Golang 原生网络库间按需切换能力，用户可以针对不同的场景选择合适的网络库，同时也支持以插件的方式为 Hertz 扩展网络库实现。

► 代码生成

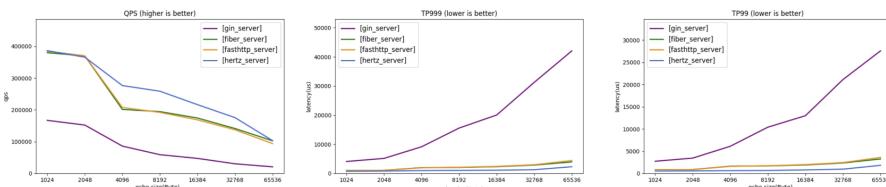
内置 Hz 工具。详见：<https://www.cloudwego.io/zh/docs/hertz/toolkits/toolkit/>

● 框架性能

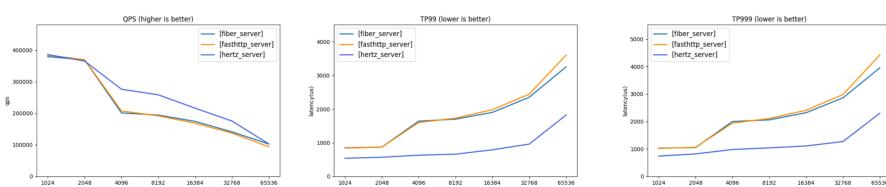
性能测试只能提供相对参考，工业场景下，有诸多因素可以影响实际的性能表现。

我们提供了 **hertz-benchmark** 项目用来长期追踪和比较 Hertz 与其他框架在不同情况下的性能数据以供参考。

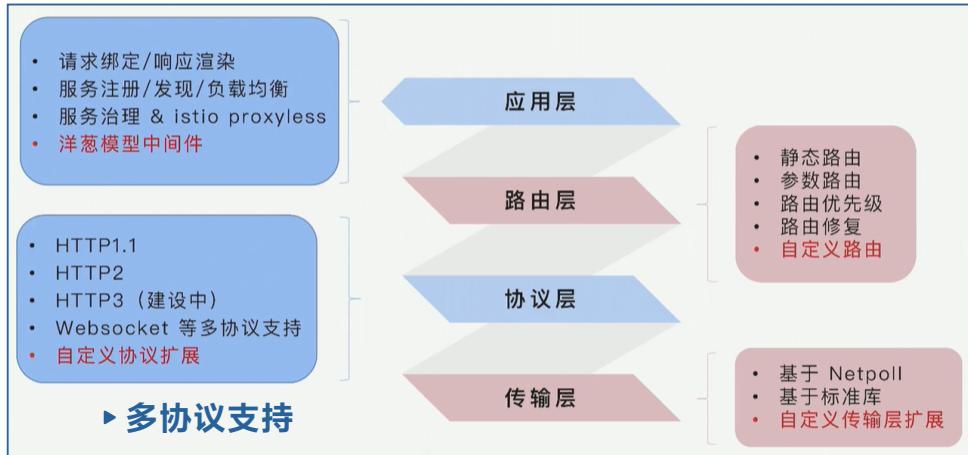
Benchmark 的开源数据。左侧第一张图是在同等的机器环境上，Hertz 和横向的框架 Gin、Fasthttp 极限 QPS 比较情况，蓝线是 Hertz 处于较高极限 QPS 的状态。第二张图是 TP99 时延状态，第三张图是 TP999 时延状态，可以看到 Hertz 的整体时延是处于一个更低的水平上。



和三大框架的性能对比



● 扩展能力



▶ 应用层

应用层提供了一些通用能力，包括绑定请求、响应渲染、服务发现/注册/负载均衡以及服务治理等等。其中，洋葱模型中间件的核心目的是让业务开发同学基于这个中间件快速地给业务逻辑进行扩展，扩展方式是可以在业务逻辑处理前和处理后分别插桩埋点做相应处理。一些比较有代表性的应用，包括日志打点、前置的安全检测，都是通过洋葱模型中间件进行处理的。

▶ 路由层

路由层也是非常通用的，主要提供静态路由、参数路由、为路由配置优先级以及路由修复的能力，如果我们的路由层没办法满足用户需求，它还能支撑用户做自定义路由的扩展。但实际应用中这些路由能力完全能够满足绝大多数用户的需求。

▶ 协议层

Hertz 同时提供 HTTP/1.1 和 HTTP/2，HTTP/3 也是我们在建设中的能力，我们还会提供 Websocket 等 HTTP 相关的多协议支持，以及支持完全由业务决定的自定义协议层扩展。

▶ 传输层

目前我们已经内置了两个高性能的传输层实现。一个是基于 CloudWeGo 开源的高性能网络库 Netpoll 的传输层扩展，另一个是支持基于标准库的传输层扩展。此外，我们也同样能支持在传输层上进行自定义传输层协议扩展。下图每一层中标红的能力都能够体现出，我们能够在框架的任何一个分层上支撑用户做最大程度的自由定制，这样可以最大程度地满足企业级内部用户和潜在用户的业务需求。

● 为什么 Hertz 性能更优？字节跳动内部 Go HTTP 框架的变迁

► 字节通过高性能 Hertz 框架，实现降本增效

Hertz 不仅支持业务服务，同时还会横向支持字节内部的各种基础组件，包括但不限于字节跳动服务网格控制面、公司级别压测平台以及 FaaS，还包括各种业务网关等等。Hertz 的高性能和极强的稳定性可以支撑业务复杂多变的场景。在公司内部 Hertz 接替了大量基于 Gin 框架开发的存量服务，大幅度降低了业务资源使用成本以及服务延时，助力公司层面的降本增效。



► 字节 Go Http 框架的背景历史

● 基于 Gin 封装

众所周知，字节内部使用 Golang 比较早，在大约 2014 年左右，公司就已经开始尝试做一些 Golang 业务的转型。2016 年，我们基于已开源的 Golang HTTP 框架 Gin 框架，封装了 Ginex，这是 Ginex 刚开始出现的时期。

同时，2016 年还是一个开荒的时代，这个时期框架伴随着业务快速野蛮地生长，我们的口号是“大力出奇迹”，把优先解决业务需求作为第一要务。Ginex 的迭代方式是业务侧和框架侧在同一个仓库里面共同维护和迭代。



● 问题显现

2017 – 2019 年期间，也就是 Ginex 发布之后，问题逐渐显现。主要有以下几点：

- 迭代受开源项目限制

Ginex 是一个基于 Gin 的开源封装，所以它本身在迭代方面是受到一些限制的。一旦有针对公司级的需求开发，以及 Bugfix 等等，我们都需要和开源框架 Gin 做联合开发和维护，这个周期不能完全由我们自己控制。

- 代码混乱膨胀、维护困难

由于我们和业务同学共同开发和维护 Ginex 框架，因此我们对于控制整个框架的走向没有完全的自主权，从而导致了整体代码混乱膨胀，到后期我们发现越来越难维护。

- 无法满足性能敏感业务需求

另外，我们能用 Gin 做的性能优化非常少，因为 Gin 的底层是基于 Golang 的一个原生库，所以如果我们要做优化，需要在原生库的基础上做很多改造，这个其实是非常困难的。

- 无法满足不同场景的功能需求

我们内部逐渐出现了一些新的场景，因此会有对 HTTP Client 的需求，支持 Websocket、支持 HTTP/2 以及支持 HTTP/3 等等需求，而在原生的 Ginex 上还是很难扩展的这些功能需求。



► Hertz 的性能优化，优化了什么

● 场景描述

HTTP/1.1 协议中的 Header 为不定长数据段，往往需要解析到最后一行，才能够确定是否完成解析。同时，为了减少系统调用次数，提升整体解析效率，涉及 IO 操作时，通常引入带 buffer 的 IO 数据结构。如下图所示，它的核心点是最下层的 buffer，buffer 是一个类似于一块完整的内存空间，我们可以将 IO 读到的数据放进这个空间做暂存。

● bufio.Reader 的问题

这样做出现的问题是，原生的 bufio.Reader 长度是固定的，请求的 Header 大小超出 buffer 长度后，.Peek() 方法直接报错 (ErrBufferFull)，无法完成既定语义功能。

● 一些可能的解

对于上述问题，其实有一些可能的解决方法：

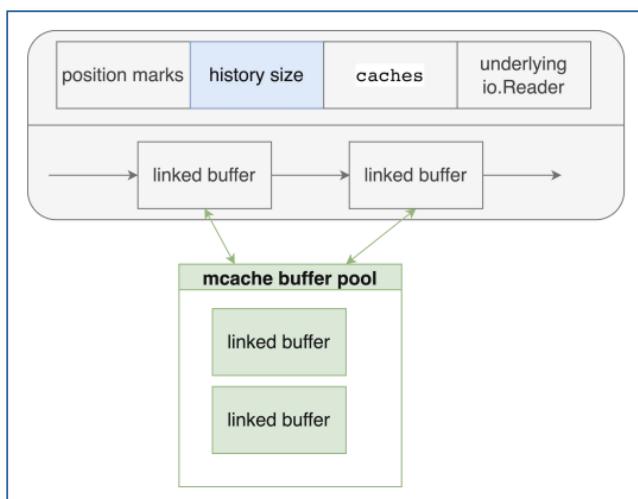
- 直接利用 `bufio.Reader` 的局限当做 Feature，通过 buffer 大小作为 Header 大小的限制。如果超出这个大小，Header 直接解析报错，这也是 Fasthttp 的做法。但实际上超出 buffer 长度后报错会导致我们没办法处理这部分请求，从而导致框架功能受限

- header 解析带状态，暂存中间数据，通过在上层堆叠额外复杂度的方式突破 `bufio` 本身的限制。但是暂存中间态会涉及到一些内存的拷贝，必然会导致性能受限。

● 真实使用环境复杂多变

字节内部的使用场景非常多，我们不仅要支持各种业务线的开发，还要支持一些横向的基础组件。不同的业务，不同的场景，数据规模各异。如何成为通用且高效的解决 `bufio.Reader` 的问题成为 Hertz 面临的内部重要挑战。我们既然已经站在 FastHTTP 这个“巨人”的肩膀上了，能否往前再走一步呢？

答案是肯定的。基于内部的使用场景，同时结合 Netpoll 的优势，我们设计出了自适应 linked buffer，并且用它替代掉了原生的 `bufio.Reader`。从下图可以看到，我们的 buffer 不再是一个固定长度的 buffer，而是一条链，这条链上的每一个 buffer 大小能够根据线上真实请求进行动态扩缩容调整，同时搭配 Netpoll 中基于 LT 触发的模型做数据预拷贝。从实施效果上来看，这个自适应调整能够让我们的业务方完全无感地支撑任何他们的业务特性。也是因为我们能够将 buffer 进行动态扩缩容调整，从而能够保证在协议层最大程度做到零拷贝协议解析，这能够带来整体解析上的性能提升，时延也会更低。



● 针对 HTTP/1.1 进行中的优化

因为目前在字节内部 HTTP/1.1 还是一个比较主流的协议，所以我们基于 HTTP/1.1 做了很多尝试。

首先是协议层探索。我们正在尝试基于 `Header Passer` 的重构，把解析 Header 的流程做得更高效。我们还尝试了做一些传输层预解析，将一些比较固化的逻辑下沉到传输层做加速。

其次是传输层探索。这包括使用 `writev` 整合发送 Header & Body 达到减少系统调用次数的目的，以及通过新增接口整合 `.Peek() + .Skip()` 语义，在内部提供一个更高效的实现。

● Hertz 示例 demo / 使用案例

► 相关扩展能力 demo 示例

https://github.com/cloudwego/hertz/blob/develop/README_cn.md

<https://github.com/hertz-contrib>

拓展	描述
Websocket	使Hertz支持WebSocket协议
Pprof	Hertz集成Pprof的扩展
Sessions	具有多状态储存支持的Session中间件
Obs-opentelemetry	Hertz的Opentelemetry扩展，支持Metric、Logger、Tracing并且达到开箱即用
Registry	提供服务注册与发现功能。到现在为止，支持的服务发现拓展有nacos、consul etcd、eureka、polaris、servicecomb、zookeeper
Keyauth	提供基于token的身份验证
Secure	具有多配置项的Secure中间件
Sentry	Sentry拓展提供了一些统一的接口来帮助用户进行实时的错误监控
Requestid	在response中添加request id
Limiter	提供了基础bbr算法的限流器
Jwt	Jwt拓展
Autotls	为Hertz支持Let's Encrypt
Monitor-prometheus	提供基于Prometheus服务监控功能
I18n	可帮助将Hertz程序翻译成多种语言
Reverseproxy	实现反向代理
Opensergo	Opensergo扩展
Gzip	含多个可选项的Gzip拓展
Cors	提供跨域资源共享支持
Swagger	使用Swagger2.0自动生成RESTful API文档
Tracer	基于Opentracing的链路追踪
Recovery	Hertz的异常恢复中间件
Basicauth	Bsaicauht中间件能够提供HTTP基本身份验证

▶ 字节服务网格使用 Hertz 的落地带来的收益

案例链接: https://mp.weixin.qq.com/s/koi9q_57Vk59YYtO9cyAFA

这可以从三个方面进行分析。

第一是从性能方面, 最初选择 Hertz 也是与很多开源框架从性能方面进行了对比。我们具体从性能方面考虑了两个主要的点, 一个是这个框架可以稳定地承载线上超过 13M QPS 的流量, 另一个是在上线前后和我们测试过程中发现 CPU 火焰图的占比是符合预期的。

第二是从应用性方面, 我们如何基于这样的一款框架快速地实现所需要的功能, 从而尽可能关注我们的业务逻辑, 而不是框架本身。如何基于这个框架实现一些高效代码, 避免还要花费更多精力学习相关语言。

第三是从长效支持方面, 目前 Hertz 已贡献给 CloudWeGo 开源社区, 而且开源和内部为统一版本, 这也为给我们提供长效支持提供了保证。

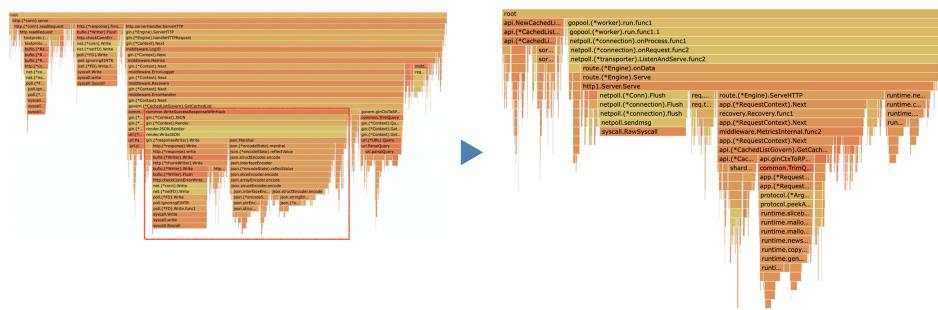
- 收益分析之性能

因为字节内部业务容器数量过多, 因此我们服务的 Goroutine 数量比较多, 从而导致稳定性较差。

通过 CPU 火焰图, 可以看到从 Gin 替换成 Hertz 之后, 我们获取的收益主要有四点。

第一, 同等吞吐量下具备更高稳定性。Hertz 的设计和实践都是基于 Netpoll 网络库实现的, 在同样的吞吐量下, 它就会比其他框架具备更高的稳定性。

第二, Goroutine 数量从 6w 降到 80。之前 Goroutine 的数量是 6 万, 使用 Hertz 从 6 万直接降到不足 100 个, Goroutine 稳定性得到极大地提升。



第三, 框架开销大幅降低。从火焰图可以看到, 替换成 Hertz 后, 之前 Gin 框架相关的开销已经基本消失不见。更多的还是像网络序列化、反序列化和业务逻辑相关的开销。

第四, 稳定承载线上超 13M QPS 流量。替换成 Hertz 后, 服务网格在线上稳定承载了超过 13M QPS 的流量。

最后再介绍一下替换前后 CPU 优化情况, 替换成为 Hertz 框架后, CPU 流量从大概快到 4k 降到大约只有 2.5k。



-收益分析之易用性

之前服务网格面临的痛点问题是很难基于已有的框架写出非常高性能的代码，可能需要对 Go 语言有更深入的了解。同时我们很难设计一个好的容错机制，比如要基于已有的框架去实现一个高性能的代码，要关注连接池、对象池，然后去控制连接超时、请求超时和读写超时等等。这中间如果有一个环节实现错误，就会导致非常灾难性的后果，之后要不停地回滚上线修复。

使用 Hertz 之后，代码实现变得非常简单。Hertz 自带比较易用的对象回调机制，我们不用再特别关注请求和响应的对象复用，以及怎样配置连接超时和请求重试，这些都是 Hertz 框架直接提供的，用户只需按照它的说明进行配置，它就可以达到用户预期。

```
on C
    // Create a new http.Client
    cli := &http.Client{
        Transport: &Transport{
            Context: context.Background(),
            Timeout: time.Second,
            // If you want to do this, how to control head timeout
            // and body timeout, it's better to do this.
            // How to control head timeout
            // How to control body timeout
        },
        // ...
    }
    // Create a new http.Request
    v := requestPool.Get()
    v.ctx = ctx
    v.cancel = context.WithTimeout(context.Background(), time.Second)
    v.req_err := http.NewRequest("GET", "http://localhost:8080/test", nil)
    v.res_err := &http.Response{
        Body: nil,
        Header: http.Header{},
        StatusCode: 0,
    }
    return v
}

func (v *C) Do() (*http.Response, error) {
    v.ctx, v.cancel = context.WithCancel(v.ctx)
    requestPool.Put(v)
    return v.res_err, v.req_err.Error()
}

func (v *C) SetMethod(method string) {
    v.req_err.Method = method
}

func (v *C) SetRequestURI(uri string) {
    v.req_err.URL.Path = uri
}

func (v *C) SetContent-TypeBytes(contentType []byte) {
    v.req_err.Header.Set("Content-Type", string(contentType))
}

func (v *C) SetBody(body []byte) {
    v.req_err.Body = bytes.NewReader(body)
}
```

```
req := hertz_protocol.AcquireRequest()
defer hertz_protocol.ReleaseRequest(req)
req.SetMethod(consts.MethodPost)
req.SetRequestURI("xxx")
req.Header.SetContentTypeBytes([]byte("application/json"))
req.SetBody(mysqlReqBytes)

res := hertz_protocol.AcquireResponse()
defer hertz_protocol.ReleaseResponse(res)
err = cli.Do(ctx, req, res)
```

-收益分析之长效支持

一些框架的开源项目，迭代速度往往达不到预期。比如 Fasthttp，它是一个非常优秀的开源高性能 HTTP 框架，但如果用户有 HTTP/2.0 或者 WebSockets 相关的需求，会发现最近几年它都没有做相应的支持，在官方文档上也声明这方面的研发还在进展之中。

如果过一段时间开源项目迭代了，还要花时间做 Rebase 或与社区的同步等等，这个维护过程成本比较高。

CloudWeGo/Hertz 是由字节内部非常强大的社区组织运营的，同时 Hertz 也是字节跳动内部广泛使用的框架。业务同学或其他公司用户使用 Hertz 时遇到的问题在字节内部使用过程中都已经出现过，而且已经针对相关问题做了修复或者功能的添加，因此用户的要求大部分需求会直接得到满足。

同时 CloudWeGo/Hertz 字节内部的使用版本是基于 Hertz 的开源版本构建的。很多特性在内部上线之后，我们也会同步地发布到外部的开源版本上，使得大家能够快速地使用到这样的特性，享受到 Hertz 更高的性能。

● Kitex 与 Hertz 的工程实践案例

▶ Kitex Proxyless 之流量路由：配合 Istio 与 OpenTelemetry 实现全链路泳道

Kitex 是字节跳动开源的 Golang RPC 框架，目前已经原生支持了 xDS 标准协议，支持以 **Proxyless** 的方式被 ServiceMesh 统一纳管。

-详细设计见：

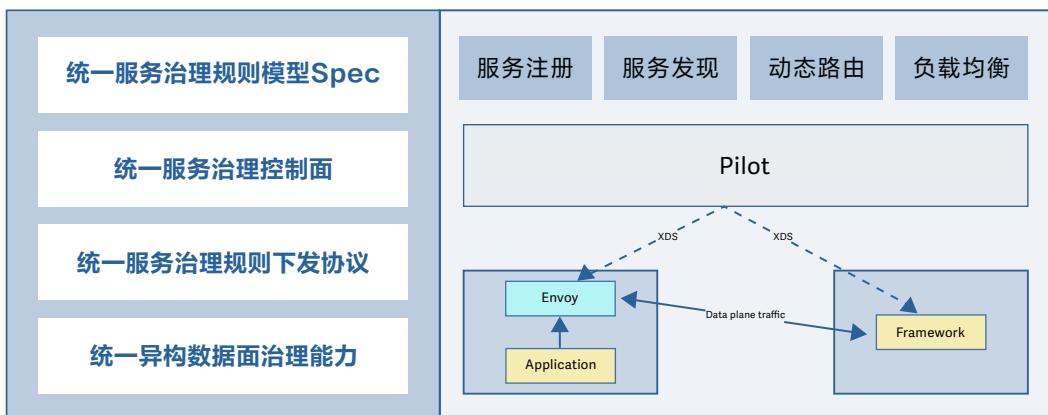
[Proposal: Kitex support xDS Protocol · Issue #461 · cloudwego/kitex](#)

-具体使用方式见：

<https://www.cloudwego.io/zh/docs/kitex/tutorials/advanced-feature/xds/>

Kitex Proxyless 简单来说就是 Kitex 服务能够不借助 envoy sidecar 直接与 istiod 交互，基于 xDS 协议动态获取控制面下发的服务治理规则，并转换为 Kitex 对应规则来实现一些服务治理功能（例如本文的重点：**流量路由**）。

💡 基于 Kitex Proxyless，让我们实现 Kitex 能够无需代理就可以被 ServiceMesh 统一管理，进而实现多种部署模式下的治理规则 Spec、治理控制面、治理下发协议、异构数据治理能力的统一。

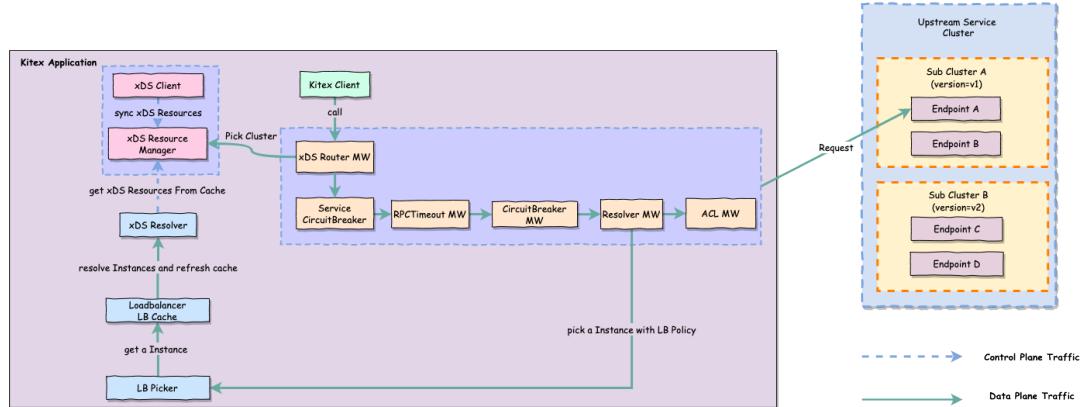


▶ 流量路由

流量路由是指，能够将流量根据其自身特定的元数据标识路由到指定目的地。

流量路由属于服务治理中比较核心的能力之一，也是 Kitex Proxyless 优先支持的场景之一。

Kitex 基于 xDS 实现 流量路由 的方案大致如下：



• 具体流程：

1. 增加一个 **xDS Router MW** 来负责 Pick Cluster (路由)，并 watch 目标服务的 LDS 及 RDS。
2. 感知 LDS 变化，并提取目标服务的 LDS 中的 Filter Chain 及其 inline RDS。
3. 感知 RDS 变化，根据 VirtualHost 和 ServiceName 来匹配（支持前缀、后缀、精确、通配），获取目标服务的路由配置。
4. 遍历处理匹配到的 RDS 中的路由规则，路由规则主要分为两部分（参考：[路由规范定义](#)）：
 - a. **Match**（支持前缀、后缀、精确、通配等），目前版本我们支持以下两种即可：
 - Path（必须项）：从 rpcinfo 提取 Method 进行匹配。
 - HeaderMatcher（可选项）：从 metainfo 中提取对应元数据 KeyValue，并进行匹配。
 - b. **Route**：
 - Cluster：标准 Cluster。
 - WeightedClusters（权重路由）：MW 内根据权重来选择 cluster。
 - 将选择到的 Cluster 写入 EndpointInfo.Tag，用于之后的服务发现。

💡 可以看到，流量路由其实是一个根据一定规则选择对应 SubCluster 的流程。

▶ 全链路泳道

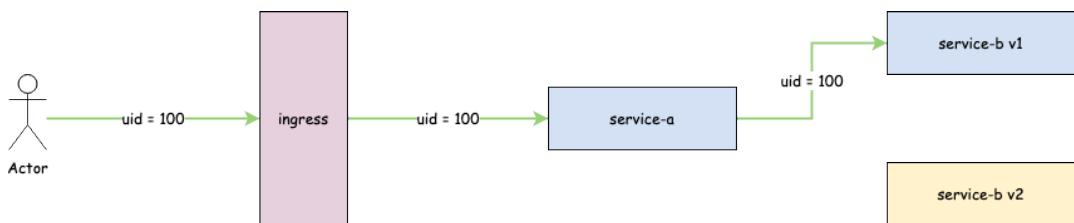
基于流量路由能力，我们可以延伸出很多使用场景，如：A/B 测试、金丝雀发布、蓝绿发布等等，以及本文重点：全链路泳道。

全链路泳道可以理解成是对一组服务实例按照一定方式进行拆分（例如部署环境），并基于全链路灰度路由能力，让流量能够精准按照规则在指定服务实例泳道中流动（逻辑上如同游泳场中的泳道）。

在 Istio 中我们一般会通过 DestinationRule 的 subset 对实例进行分组，将一个服务拆分成不同子集（例如：按照版本、区域等属性拆分），然后配合 VirtualService 来定义对应的路由规则，将流量路由到对应子集中，从而完成泳道中的单跳路由能力。

不过单单只有流量路由能力，还不足以实现全链路泳道，因为当一个请求跨越多个服务的时候，我们需要有一个比较好的机制能够准确识别出该流量，并基于这个特征来为每一跳流量配置路由规则。

如下图所示：假设我们要实现一个用户的请求能够精确灰度到 service-b 的 v1 版本。最先想到的做法可能是所有请求都带上 uid = 100 的请求头，然后配置对应 VirtualService 来根据 header 里的 uid = 100 匹配。



- 但这样的做法有几个明显的缺点：

1. **不够通用：**以具体某个业务属性标识（如：uid）作为流量路由匹配规则，我们需要将这个业务属性手动在全链路里透传，这本身对业务侵入性较大，需要业务配合改造。并且当我们使用其他业务属性的时候，又需要全链路业务都改造一遍，可想而知，是非常不通用的做法
2. **路由规则容易频繁变动，容易造成规则臃肿：**以具体某个业务属性标识（如：uid）作为流量路由匹配规则，假设我们要换一个业务属性，或者给其他用户设置路由规则的时候，得去改造原有的路由规则，或者针对不同业务属性重复定义多套路由规则，很容易就会造成路由臃肿，以至于难以维护

💡 因此，要实现全链路的流量路由统一，我们还需要额外借助一个更通用的 流量染色与 染色标识全链路透传 能力。

▶ 流量染色

流量染色是指对请求流量打上特殊标识，并在整个请求链路中携带这个标识，而所谓的全链路泳道，就是整个链路基于统一的灰度流量染色标识来设置流量路由规则，使得流量能够精准控制在不同泳道中。

通常我们会在网关层进行流量染色，通常会根据原始请求中的元数据，来进行一定规则（条件、比例）转换成对应的染色标识。

-**按条件染色**：当请求元数据满足一定条件之后，就给当前请求打上染色标识，如：请求头中 uid=100、cookie 匹配等等。

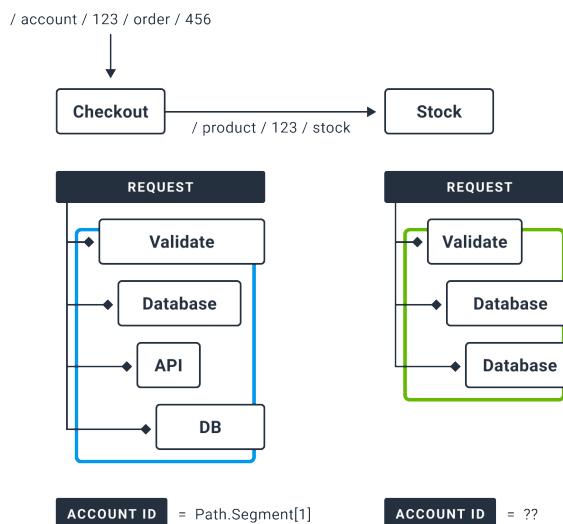
-**按比例染色**：按照一定比例，给请求打上染色标识。

有了一套统一的流量染色机制之后，我们配置路由规则的时候，就不需要关心具体的业务属性标识了，只需要根据 染色标识 来配置即可。

将具体的业务属性抽象成条件染色规则，使其更通用，即使业务属性发生了变化，路由规则也无需再频繁变动了。

▶ 染色标识全链路透传

染色标识通常会依靠 Tracing Baggage 来透传，Baggage 是用于在整个链路中传递业务自定义 KV 属性，例如传递流量染色标识、传递 AccountID 等业务标识等等。



要实现流量染色标识在全链路透传，我们通常会借助 Tracing Baggage 机制，在全链路中传递对应染色标识，大部分 Tracing 框架都支持 Baggage 概念机能力，如：OpenTelemetry、Skywalking、Jaeger 等等。

有了一套通用的全链路透传机制，业务方就只需要接入一遍 tracing 即可，无需每次业务属性标识发生变化就配合改造一次。

下面会借助一个具体的工程案例介绍，来介绍并演示如何基于 Kitex Proxyless 和 OpenTelemetry Baggage 实现全链路泳道功能。

▶ 使用 Hertz、Kitex 重写经典的 Istio Bookinfo 项目

- 💡 - 使用 istiod 来作为 xDS server，作为 CRD 配置和下发的入口；
- 使用 wire 来实现依赖注入；
- 使用 opentelemetry 来实现全链路追踪；
- 使用 Kitex-xds 和 opentelemetry baggage 来实现 proxyless 模式下的全链路泳道；
- 使用 arco-design 和 react 实现一个 Bookinfo UI。

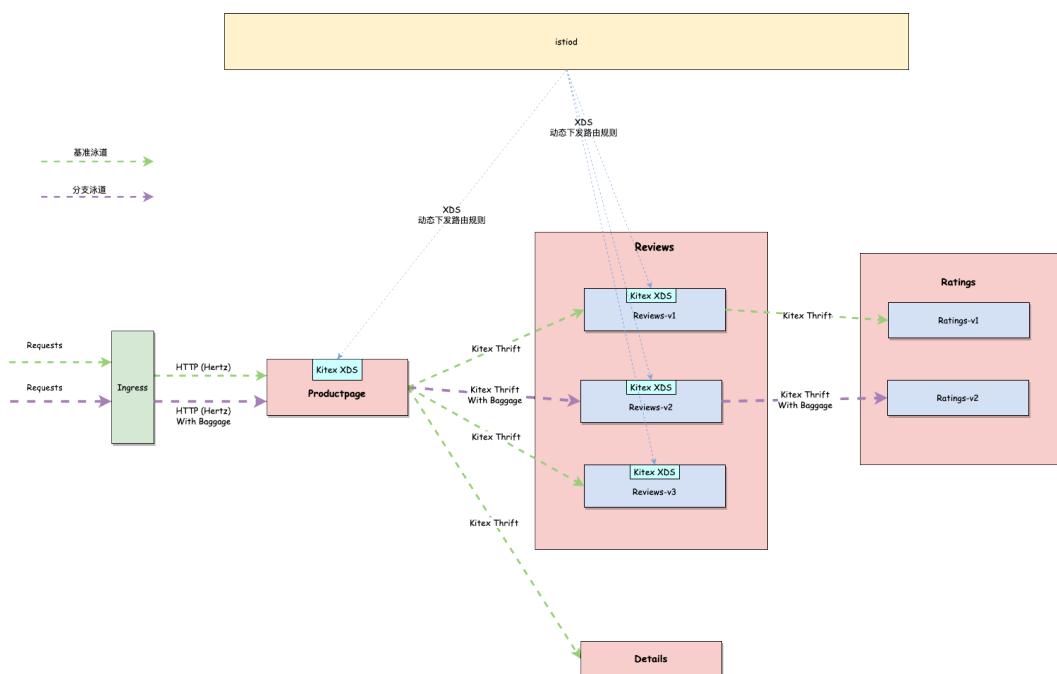
● 架构

整体架构与 Bookinfo 保持一致，分为四个单独的微服务：

- productpage. 这个微服务会调用 details 和 reviews 两个微服务；
- details. 这个微服务中包含了书籍的信息；
- reviews. 这个微服务中包含了书籍相关的评论。它还会调用 ratings 微服务；
- ratings. 这个微服务中包含了由书籍评价组成的评级信息。

reviews 微服务有 3 个版本：

- v1 版本会调用 ratings 服务，并使用 1 颗 ★ 显示评分；
- v2 版本会调用 ratings 服务，并使用 5 颗 ★★★★★ 显示评分；
- v3 版本不会调用 ratings 服务。。

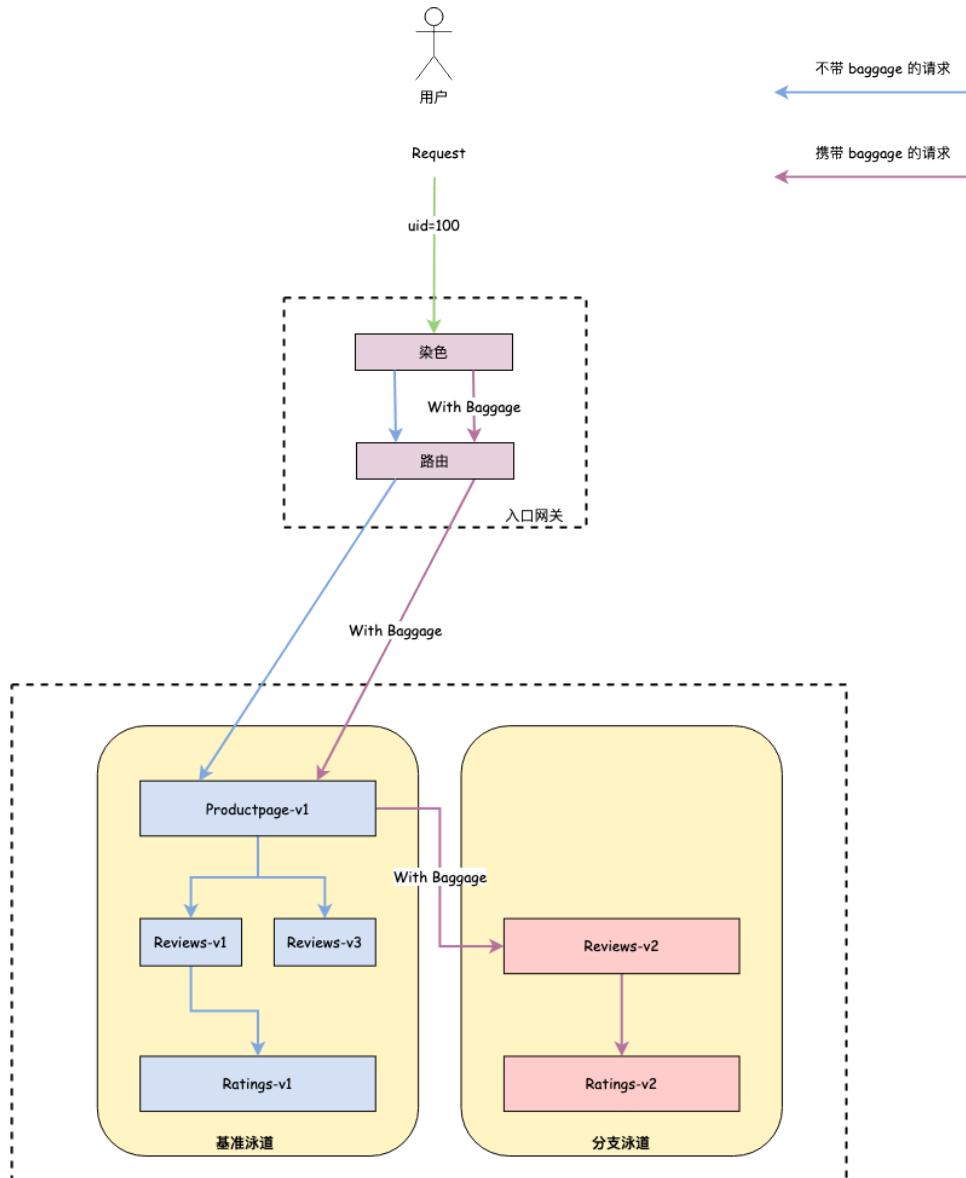


● 池道示意图

整体区分成 2 个池道：

-**基准池道**：未被染色的流量会被路由到基准池道中。

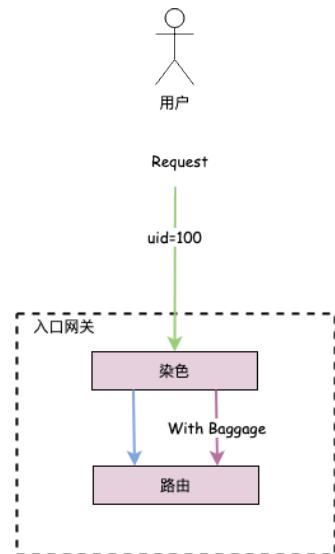
-**分支池道**：被染色的流量会被路由到 reviews-v2 -> ratings-v2 的分支池道中。



● 池道示意图

网关统一负责对流量进行染色，例如请求 header 中 uid=100 的流量都统一进行染色，为请求携带上 env=dev 的 baggage。

染色方式可以根据不同的网关实现具体选择，例如当我们选择 istio ingress 作为网关的时候，我们可以借助 EnvoyFilter + Lua 的方式来编写网关染色规则。



● 为服务实例打标

1. 为对应 workload 打上对应 version 标识：

以 reviews 为例，只需要给对应 pod 打上 version: v1 的 label 即可。

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   labels:
5     app.kubernetes.io/instance: reviews
6     app.kubernetes.io/name: reviews
7     name: reviews-v1
8   spec:
9     replicas: 1
10    selector:
11      matchLabels:
12        app.kubernetes.io/instance: reviews
13        app.kubernetes.io/name: reviews
14        version: v1
15    template:
16      metadata:
17        annotations: CoderPoet, 2022/9/19, 3:05 上午 • feat(bookinfo)
18        sidecar.istio.io/inject: "false"
19        labels:
20          app.kubernetes.io/instance: reviews
21          app.kubernetes.io/name: reviews
22          version: v1
23      spec:
24        containers:
25          - args:
26            - reviews
27            - --config=config/reviews.yaml
28            env:
29              - OTEL_RESOURCE_ATTRIBUTES = service.version=v1
              - OTEL_EXPORTER_OTLP_ENDPOINT = http://localhost:4317
```

2. 基于 DestinationRule 为服务设置一系列的 subsets:

```
| Productpage: v1  
| Reviews: v1、v2、v3  
| Ratings: v1、v2
```

```
apiVersion: networking.istio.io/v1alpha3  
kind: DestinationRule  
metadata:  
  name: productpage  
spec:  
  host: productpage  
  subsets:  
    - name: v1  
      labels:  
        version: v1  
---  
apiVersion: networking.istio.io/v1alpha3  
kind: DestinationRule  
metadata:  
  name: reviews  
spec:  
  host: reviews  
  subsets:  
    - name: v1  
      labels:  
        version: v1  
    - name: v2  
      labels:  
        version: v2  
    - name: v3  
      labels:  
        version: v3  
---  
apiVersion: networking.istio.io/v1alpha3  
kind: DestinationRule  
metadata:  
  name: ratings  
spec:  
  host: ratings  
  subsets:  
    - name: v1  
      labels:  
        version: v1  
    - name: v2  
      labels:  
        version: v2
```

● 流量路由规则

网关已经将请求头中携带了 uid=100 的流量进行了染色，自动带上了 env=dev 的 baggage，因此我们只需要根据 header 进行路由匹配即可，下面是具体的路由规则配置示例：

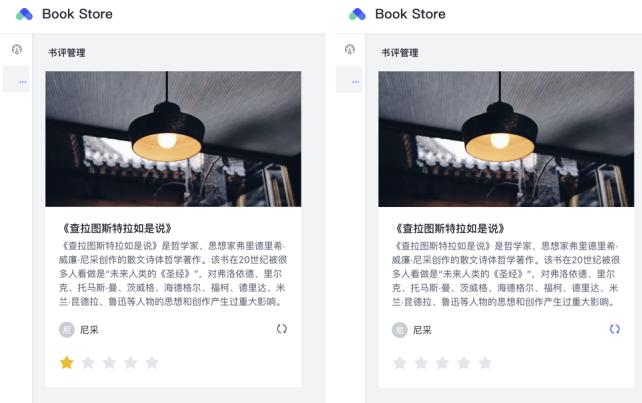
```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
    - match:
        - headers:
            baggage:
              exact: "env=dev"
      route:
        - destination:
            host: reviews
            subset: v2
            weight: 100
        - route:
            - destination:
                host: reviews
                subset: v1
                weight: 80
            - destination:
                host: reviews
                subset: v3
                weight: 20
    ---
```

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
    - ratings
  http:
    - match:
        - headers:
            baggage:
              exact: "env=dev"
      route:
        - destination:
            host: ratings
            subset: v2
            weight: 100
        - route:
            - destination:
                host: ratings
                subset: v1
                weight: 100
```

● 查看效果

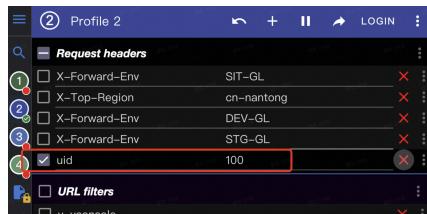
基准泳道

网关已经将请求头中携带了 uid=100 的流量进行了染色，自动带上了 env=dev 的 baggage，因此我们只需要根据 header 进行路由匹配即可，下面是具体的路由规则配置示例：



分支泳道

1. 我们这边通过浏览器 mod-header 插件，来模拟入口流量请求头中携带了 uid=100 的场景。



2. 再点击刷新按钮，可以发现请求打到了分支泳道，流量泳道功能成功生效。



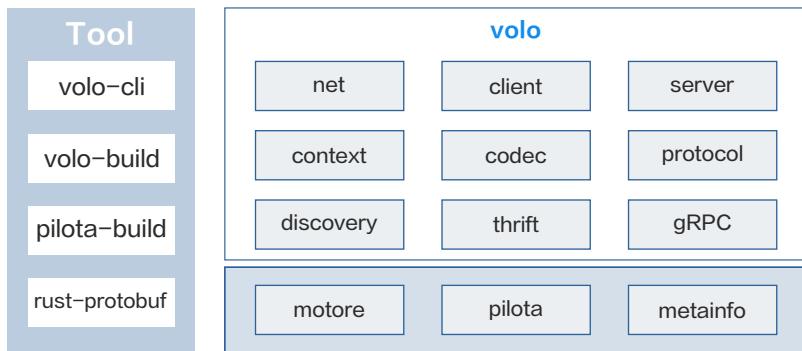
当然，除了满足 **流量路由** 能力之外，**Kitex Proxyless** 也在持续迭代优化，满足更多数据面治理能力需求。Proxyless 作为一种 ServiceMesh 数据面探索和实践，除了能够丰富网格数据面部署形态之外，也希望可以不断打磨 **Kitex**，增强其在开源生态兼容方面的能力，打造一个开放包容的微服务生态体系。

● Rust 首选 RPC 框架 Volo

Github 地址: <https://github.com/cloudwego/volo>

文档: <https://www.cloudwego.io/zh/docs/volo/overview/>

架构图



▶ 特性

● 基于 GAT 设计

我们热爱并追随最新的技术，Volo 的核心抽象使用了 Rust 最新的 GAT 特性，在这个过程中我们也借鉴了 Tower 的设计。Tower 是一个非常优秀的抽象层设计，适用于非 GAT 的情况下，非常感谢 Tower 团队。

通过 GAT，我们可以避免很多不必要的 Box 内存分配，以及提升易用性，给用户提供更友好的编程接口和更符合人体工程学的编程范式。

● 高性能

Rust 以高性能和安全著称，我们在设计和实现过程中也时刻以高性能作为我们的目标，尽可能降低每一处的开销，提升每一处实现的性能。

首先要说明，和 Go 的框架对比性能是极不公平的，因此我们不会着重比较 Volo 和 Kitex 的性能，并且我们给出的数据仅能作为参考，希望大家能够客观看待；同时，由于在开源社区并没有找到另一款成熟的 Rust 语言的 Async 版本 Thrift RPC 框架，而且性能对比总是容易引战，因此我们希望尽可能弱化性能数据的对比，仅会公布我们自己极限 QPS 的数据。

在和 Kitex 相同的测试条件（限制 4C）下，Volo 极限 QPS 为 35W；同时，我们内部正在验证基于 Monoio (CloudWeGo 开源的 Rust Async Runtime) 的版本，极限 QPS 可以达到 44W。

从我们线上业务的火焰图来看，得益于 Rust 的静态分发和优秀的编译优化，框架部分的开销基本可以忽略不计（不包含 syscall 开销）。

● 易用性好

Rust 以难学难用而闻名，我们希望尽可能降低用户使用 Volo 框架以及使用 Rust 语言编写微服务的难度，提供最符合人体工程学和直觉的编码体验。因此，我们把易用性作为我们重要的目标之一。

比如，我们提供了 `volo` 命令行工具，用于初始化项目以及管理 `IDL`；同时，我们将 `thrift` 及 `gRPC` 拆分为两个独立（但共用一些组件）的框架，以提供最符合不同协议语义的编程范式及接口。

我们还提供了`#[service]`宏（可以理解为不需要 `Box` 的 `async_trait`）来使得用户可以无心理负担地使用异步来编写 `Service` 中间件。

● 扩展性强

收益于 Rust 强大的表达和抽象能力，通过灵活的中间件 `Service` 抽象，开发者可以以非常统一的形式，对 RPC 元信息、请求和响应做处理。

比如，服务发现、负载均衡等服务治理功能，都可以以 `Service` 形式进行实现，而不需要独立实现 `Trait`。



公众号二维码



CloudWeGo
项目介绍电子版

