Kitex/Hertz 助力大模型：三周年重要特性回顾

分享人：杨芮

字节跳动服务框架研发工程师

2024.09.21

大模型

用户体验

性能

关键字 | Keywords

目录 | Contents

Part 01 加强流式能力助力大模型

Part 02 新功能、用户体验/性能提升回顾

Part 03 总结和展望

01 加强流式能力助力大模型

SSE、Thrift Streaming、Streaming 泛化调用、用户体验

流式能力 | Kitex - gRPC , Hertz - HTTP Chunked / WebSocket

支持情况 **Kitex/Hertz 均支持流式场景**

- Kitex 支持 gRPC；性能优于官方 gRPC；功能基本对齐
- Hertz 支持 HTTP Chunked Transfer Encoding, WebSocket

以上能力 不足以支持 LLM 快速发展

端上 SSE 应用更多

大模型应用在文本对话场景多使用 SSE 协议实时返回服务端结果给客户端

- 文本推送场景更简单
- 浏览器支持友好

Thrift -> PB 切换负担

字节服务端服务主要使用 Thrift 定义，研发对 Thrift 使用更加熟悉

- 减少研发心智负担
- 广泛增加 Protobuf 定义服务，对统一IDL/接口管理不友好

缺乏工程实践

流式通信相比原来 PingPong 模型在服务治理上增加了复杂度

- 没有工程实践的沉淀
- 流式接口很容易用错
- 流式监控如何定义

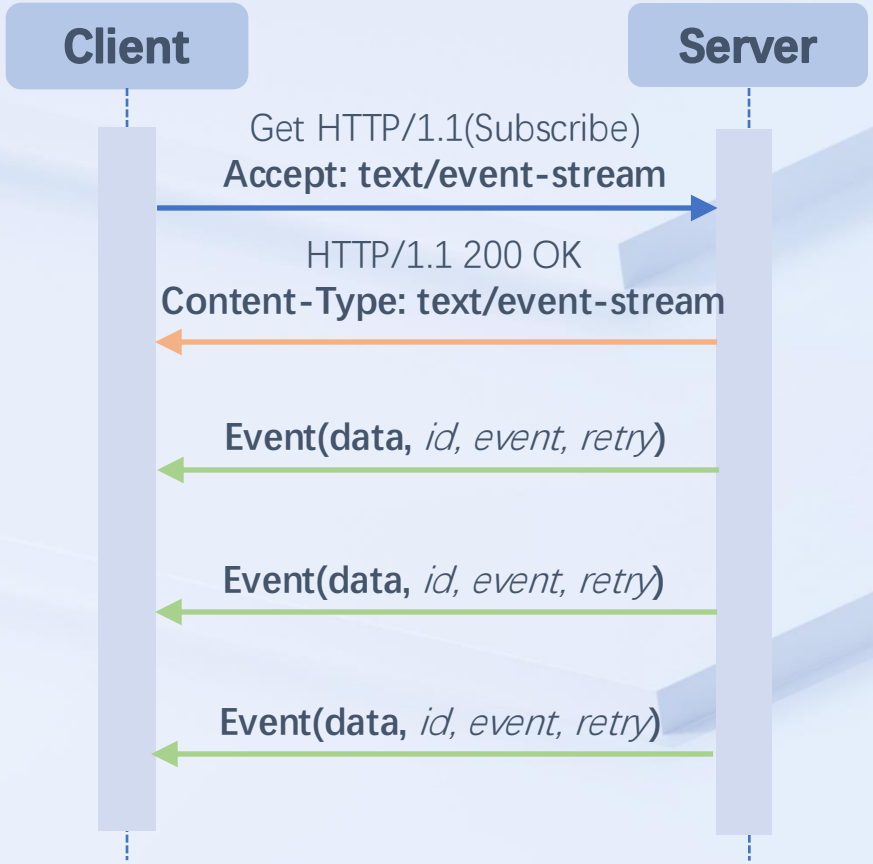
流式能力 | Hertz - SSE

SSE Server-Send Events

- 基于 HTTP 协议，支持服务端向客户端**单向**推送数据
- 简单，适合文本传输

该功能由社区同学贡献支持，这里表示感谢~ (仓库)

适合端上	不适合服务端
适合文本对话模型 - 简单易用，开发者友好 - 可明确定义事件类型 - 对比 WebSocket 更轻量	服务端计算和传输性能要求较高 - 不适合采用低效的文本协议 - JSON 简单但并不适合服务端复杂交互场景，强类型的 RPC 更友好 - 部分场景需要双向流式通信



流式能力 | Kitex - Thrift Streaming [New]

支持背景 助力大模型快速发展 & 满足其他业务场景发展

- 字节微服务主要使用 Thrift 定义，研发对 Thrift 使用更加熟悉
- Protobuf/Thrift 遍地开花会影响研发效率

Thrift Streaming

业务场景

大模型（LLM）

大模型应用的业务快速发展，交互多采用 Server Streaming 通信。发给用户端需要进行协议转换（如 -> SSE）。
除字节内部需求，还有企业用户。

抖音搜索

为提升性能，希望 RPC 流式返回结果。例如，在视频打包阶段（根据召回的视频 ID 获取物料等相关信息），希望一次请求打包服务（10个doc），先打包完的先返回。

飞书 People

数据导出场景会并发获取数据，如果等待所有数据都获取到后再填充 excel 返回，当数据量过大时会导致 OOM，进程异常退出。

Thrift Streaming | 支持方案

快速落地方案　Thrift Streaming over HTTP2/gRPC

- **协议方案**：基于 gRPC 的 PRC 通信规范，将 Protobuf 编码改为 Thrift 编码

优点

- ServiceMesh 兼容：基于 HTTP2 传输，ServiceMesh 无需单独支持
- Kitex 支持成本低：根据 SubContentType 明确解码类型（gRPC 规范）

缺点

- 资源开销大：流控、动态窗口引入额外的开销
- 延迟影响大：流控机制，流量大一些或发送大包会导致延迟显著劣化，需要用户自行调整 WindowSize
- 问题排查难：复杂度也增加了问题排查难度

Stream 1 - Header Frame

:method: POST
:scheme: http
:path: /example.XXXSerivice/XXX
content-type: application/grpc**+thrift**
...

优化　自研 Streaming 协议，简化流式通信

Thrift Streaming | 支持方案

IDL 定义方案

通过注解定义流式接口

- 原生 Apache Thrift 不支持流式接口的定义
- 不能新增关键字，会导致其它 Thrift 通用解析工具无法支持

使用文档

优点

- 注解方式不会影响到其它 Thrift 解析工具，包括 IDE 插件解析

```
1  namespace go echo
2
3  struct Request {
4      1: required string message,
5  }
6
7  struct Response {
8      1: required string message,
9  }
10
11 service EchoService {
12     Response EchoBidirectional (1: Request req) (streaming.mode="bidirectional"),
13     Response EchoClient (1: Request req) (streaming.mode="client"),
14     Response EchoServer (1: Request req) (streaming.mode="server"),
15 }
```

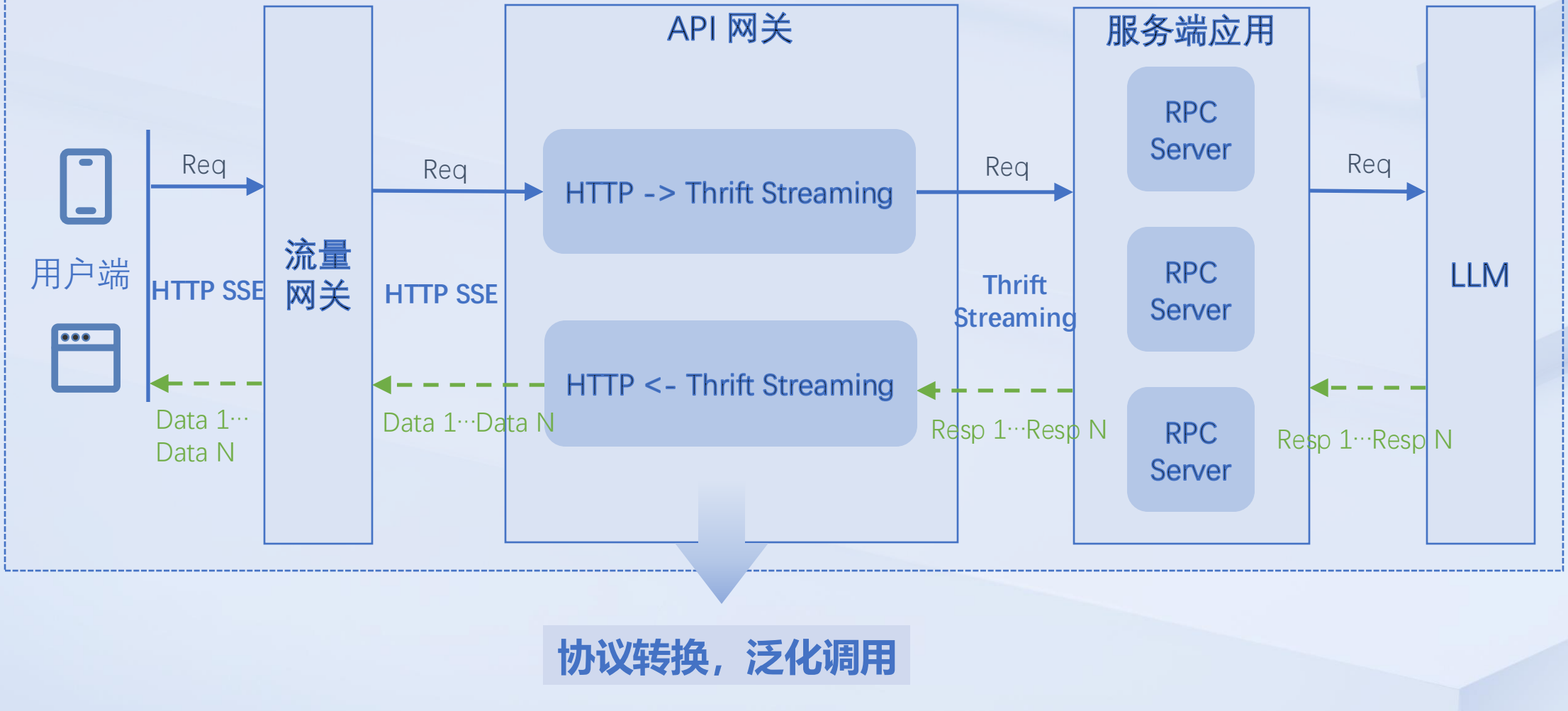
Now:
- HTTP2 + Thrift Serialization (Thrift over GRPC)

Future:
- TTHeader Streaming

流式能力 | SSE <-> Thrift Streaming

以文本对话模型举例

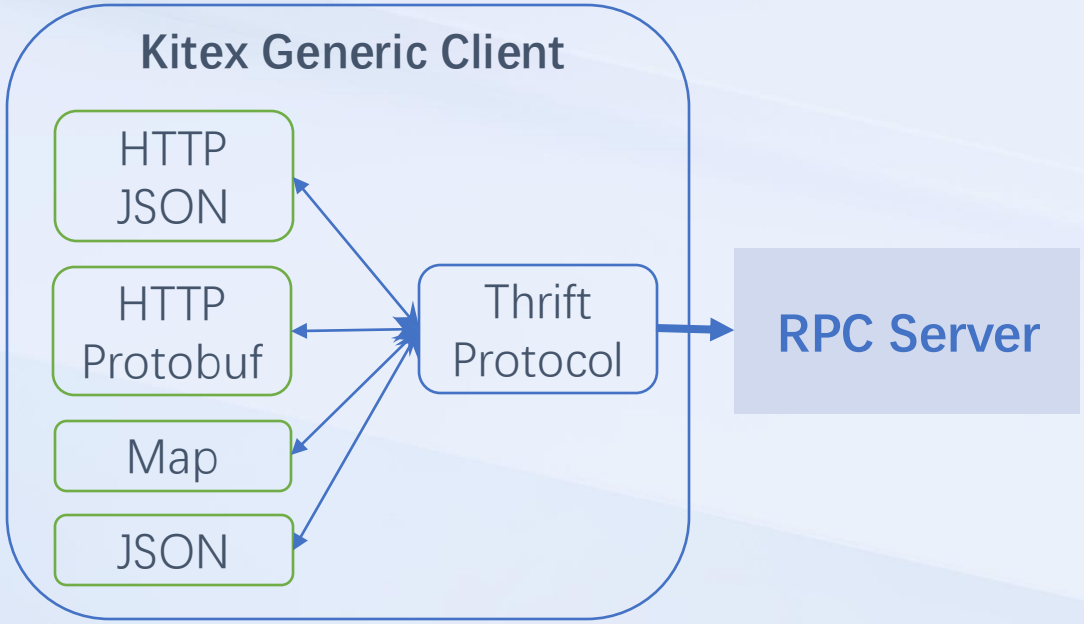
快速落地方案



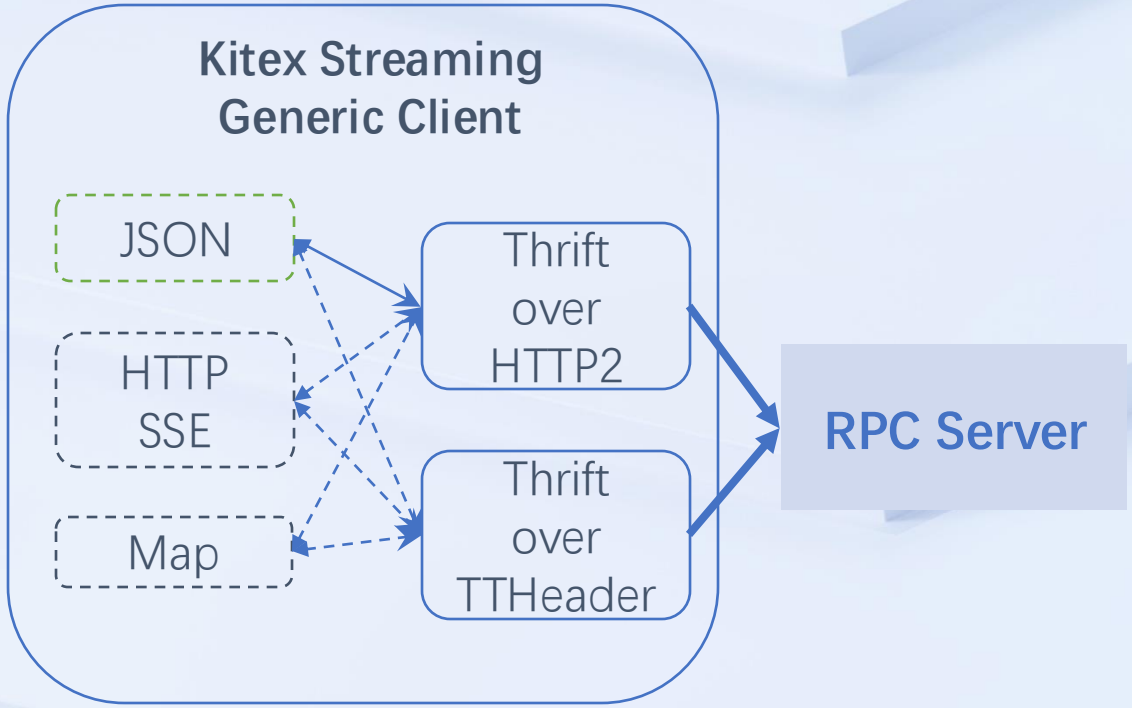
协议转换，泛化调用

流式泛化调用 | 网关、测试平台等通用服务

Thrift(PingPong) 泛化调用



Streaming 泛化调用



DONE　On Trial　DOING/TODO

流式泛化调用 | 网关、测试平台等通用服务

PingPong 泛化调用接口

```
type Client interface {
    GenericCall(ctx context.Context, method string, request interface{},
callOptions ...callopt.Option) (response interface{}, err error)
}
```

Streaming 泛化调用接口

```
type ClientStreaming interface {
    streaming.Stream
    Send(ctx context.Context, req interface{}) error
    CloseAndRecv(ctx context.Context) (resp interface{}, err error)
}

type ServerStreaming interface {
    streaming.Stream
    Recv(ctx context.Context) (resp interface{}, err error)
}

type BidirectionalStreaming interface {
    streaming.Stream
    Send(ctx context.Context, req interface{}) error
    Recv(ctx context.Context) (resp interface{}, err error)
}
```

- JSON 流式泛化调用已在字节内**压测**、**接口测试平台**上线，支持流式场景的测试
- 因为 Streaming 后续会发布 v2 接口，暂未正式发布，就绪后统一发布
- 但功能就绪，可以试用，*试用点这里*

流式能力 | 用户体验

开发过流式接口的同学，你们觉得好用吗？

熟悉如何正确使用吗？

流式能力用户体验 | 用户体验

Client 接口

```
type Client interface {
    CallUnary(ctx context.Context, Req *grpc_demo.Request, callOptions ...callopt.Opti
    CallClientStream(ctx context.Context, callOptions ...callopt.Option) (stream Servi
    CallServerStream(ctx context.Context, Req *grpc_demo.Request, callOptions ...callo
    CallBidiStream(ctx context.Context, callOptions ...callopt.Option) (stream Service
}
```

Server 接口

```
type ServerIface interface {
    CallUnary(ctx context.Context, req *Request) (res *Reply, err error)
    CallClientStream(stream ServiceA_CallClientStreamServer) (err error)
    CallServerStream(req *Request, stream ServiceA_CallServerStreamServer) (err error)
    CallBidiStream(stream ServiceA_CallBidiStreamServer) (err error)
}
```

Streaming Handler 里获取 ctx 方式

```
func (s ServiceCImpl) CallClientStream(stream grpc_demo.ServiceA_CallClientStreamServer)
    ctx := stream.Context()
    // ...
}
```

想做调整担心考虑不周 ☹

向 gRPC 官方咨询为什么服务端 stream 接口没提供 ctx 参数

issue

大胆改! 😄

流式能力用户体验 | 优化用户体验、完善工程实践

易用性不够、缺乏工程实践

可观测性

- 流式接口如何定义请求量?
- 流式接口如何定义延迟?
- 流式接口做细粒度埋点

用户体验问题

- RPC 接口定义 Client/Server 差异
- Middleware 扩展
- Option 混杂，用户误用
- 错误信息不明确

工程实践问题

- 优雅退出
- 错误处理规范
- 重试能力
- 超时控制
- Ctx cancel 传播

进行中

专项优化流式问题

可观测性

- 针对 Send/Recv 单独 Metric 吞吐指标
- Send/Recv 不上报延迟指标
- 增加细粒度埋点记录 Send/Recv Event 信息

用户体验问题

- 新增 Streaming v2 接口
- 流式单独 Middleware 避免混用行为不同
- Option 对流式单独分包，避免无用
- 优化错误信息，便于快速确认问题

工程实践问题

- 制定优雅退出规范，需用户代码协同处理
- 制定错误处理规范，如明确优雅退出的错误
- 单独支持流式重试能力
- 制定超时控制的使用建议
- 制定流式接口使用规范，避免不合理使用

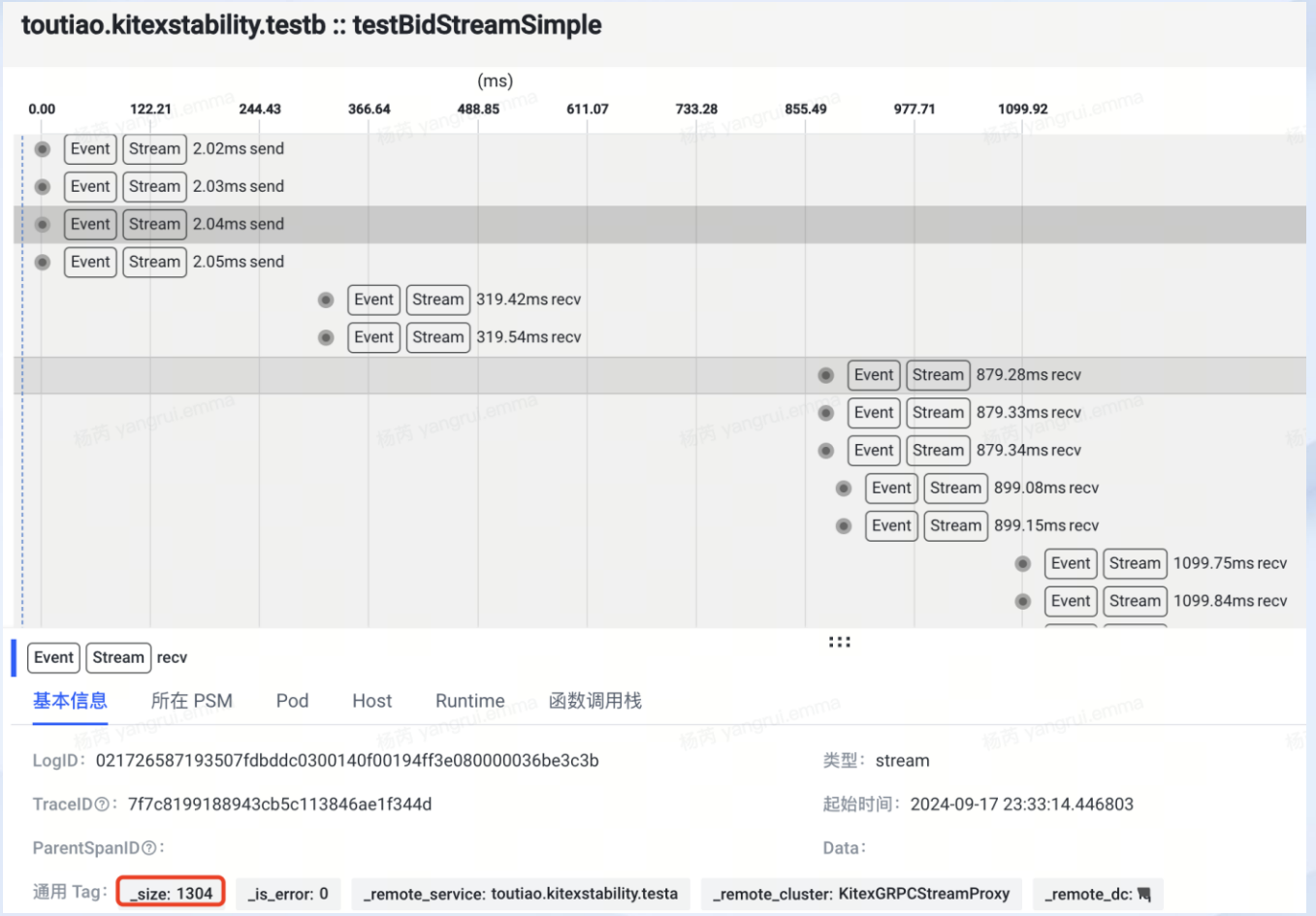
流式能力用户体验 | 可观测性

流式消息监控

- **新增 StreamSend & StreamRecv 事件**：
  调用 stream.Send 或 stream.Recv 时产生，记录时间和 Send/Recv Size

- **新增 StreamEventReport 接口**：
  用于上报流式消息监控

详见文档



02

新功能、用户体验/性能提升回顾

Thrift/gRPC 多 Services、产物优化、性能优化、内存分析工具...

新功能 | Thrift/gRPC 多 Services

支持在一个 Server 里注册多个 IDL Service，包括 Thrift、Protobuf

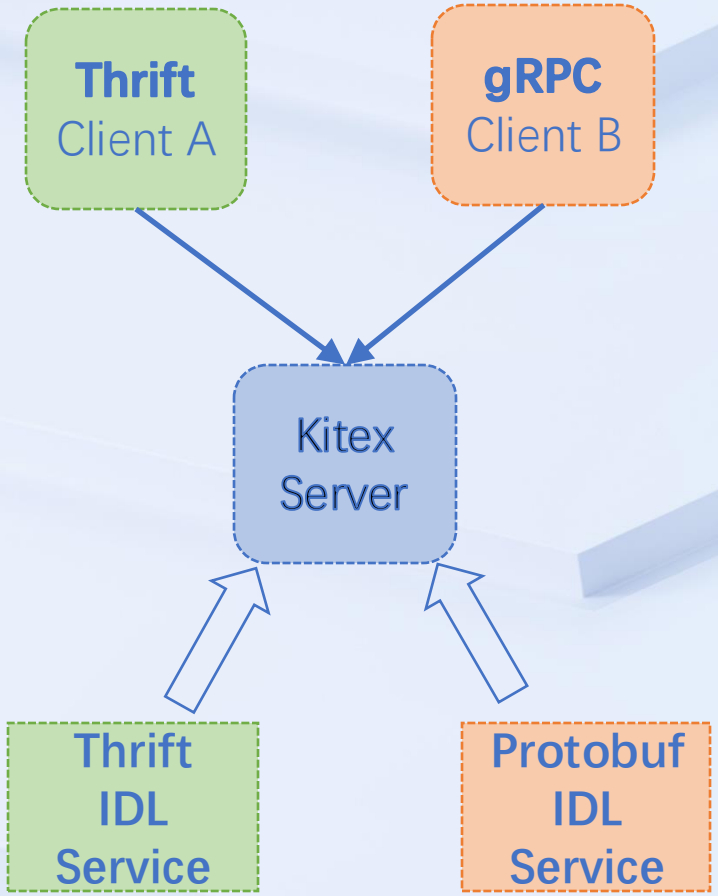
使用文档

gRPC 多 Service

- 对齐官方 gRPC 能力

Thrift 多 Service

- Thrift 基于 TTHeader 提供了协议层面真正的多 Service 功能，同时兼容旧的 CombineService
- 原 Combine Service 限制不同 Service 方法不能重名



注册方式

- Thrift 和 gRPC 注册方式一致

```go
func main() {
    // create a server by calling server.NewServer
    svr := server.NewServer(your_server_option)
    // register your multi-service on a server
    err := servicea.RegisterService(svr, new(ServiceAImpl))
    err := serviceb.RegisterService(svr, new(ServiceBImpl))

    if err := svr.Run(); err != nil {
        klog.Errorf("%s", err.Error())
    }
}
```

新功能 | Mixed Retry

同时具备 Failure Retry 和 Backup Request 功能的混合重试功能

相比前两种重试的优势

- 可以优化 Failure Retry 的整体重试延迟
- 可以提高 Backup Request 的请求成功率

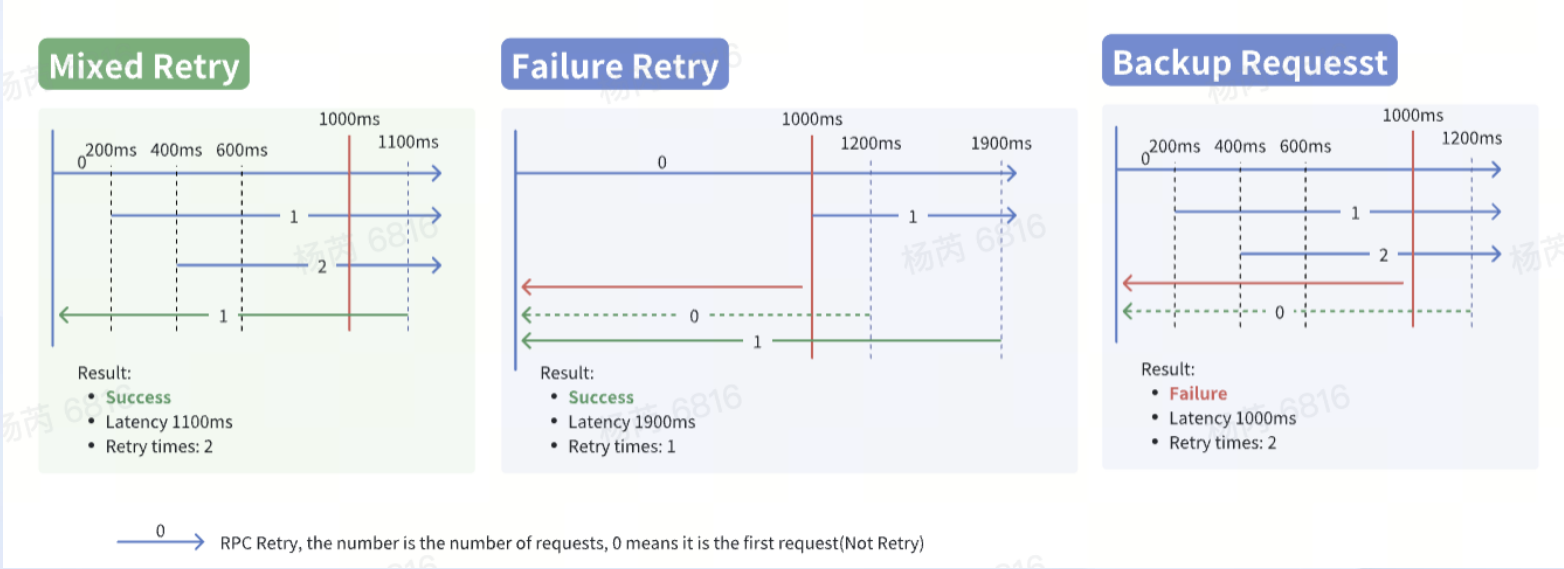
使用文档

重试策略对比

场景: 假设第一个请求耗时 1200ms，第二次请求耗时 900ms

- **配置:** RPCTimeout=**1000ms**、MaxRetryTimes=2 BackupDelay=200ms

- **重试结果:**
  - Mixed Retry: **Success**, cost **1100ms**
  - Failure Retry: **Success**, cost **1900ms**
  - Backup Retry: **Failure**, cost 1000ms



用户体验提升 | Frugal & FastCodec (Thrift)

问题：FastCodec 和 Frugal 的解码都必须依赖带头的包，如果是 Buffered 包 Fallback 到 Apache Codec

SkipDecode

- 通过 SkipDecode 解决协议绑定问题，SkipDecode + FastCodc 性能依然优于 Apache Thrift Codec

Frugal ARM 支持

- 针对 ARM 架构，提供了反射的支持，x86 还是 JIT
- Go 1.24 发布后，计划全部采用反射实现

Frugal 仓库

反射实现性能没有劣化反而略优于 JIT

```
goos: linux
goarch: amd64
pkg: github.com/cloudwego/frugal/tests
cpu: Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz

Marshal_ApacheThrift/small-4          2070745          584.0 ns/op          998.32 MB/s
Marshal_ApacheThrift/medium-4           78729          13680 ns/op         1280.57 MB/s
Marshal_ApacheThrift/large-4             3097         376184 ns/op         1179.75 MB/s
Marshal_Frugal_JIT/small-4            4939591          242.1 ns/op         2407.83 MB/s
Marshal_Frugal_JIT/medium-4            160820           7485 ns/op         2340.29 MB/s
Marshal_Frugal_JIT/large-4               5370         214258 ns/op         2071.35 MB/s
Marshal_Frugal_Reflect/small-4        10171197          117.3 ns/op         4970.90 MB/s
Marshal_Frugal_Reflect/medium-4         180207           6644 ns/op         2636.73 MB/s
Marshal_Frugal_Reflect/large-4            6312         185534 ns/op         2392.04 MB/s
```

用户体验提升 | 产物精简和生成提速优化

针对 IDL 复杂的服务，提供多种优化手段减少产物大小、提升产物生成速度　　详见文档

IDL 裁切

*$ kitex -module xx -thrift **trim_idl** xxxx.thrift*

```
[WARN] You Are Using IDL Trimmer
[WARN] removed 66567 unused structures with 537125 fields
```

no_fmt 提速

*$ kitex -module xx -thrift **no_fmt** xxxx.thrift*

- 字节内某平台生成耗时 P90 从 **80s** 下降到了 **20s**

删除 Kitex 无需依赖的代码

- Thrift Processor、Deep Equal、Apache Codec 都提供了对应参数可不生成
- Thrift Processor 在 v0.10.0 默认不生成、预计 v0.12.0 将默认删除 Apache Thrift 代码产物

Frugal Slim 极致精简

*$ kitex -thrift frugal_tag,**template=slim** -service p.s.m idl/api.thrift*

- 产物体积减小约 **90%**

用户体验提升 | kitexcall

问题：RPC 测试要生成代码构造测试代码，不及 HTTP 测试便利

kitexcall

- 基于 Kitex JSON 泛化调用提供了单独的命令工具方便用户使用 JSON 数据发起 Thrift 测试

该功能由社区同学贡献支持，这里表示感谢~ 仓库，工具介绍

使用实例

```
$ kitexcall -idl-path echo.thrift -m echo -d '{"message": "hello"}' -e 127.0.0.1:8888
```

未来支持

- 界面化，测试更便利
- 支持 gRPC 测试
- 无需指定 IDL，结合 server reflection 能力获取 idl 信息

性能优化 | Thrift 按需序列化

参考 Protobuf 提供了 Thrift FieldMask 功能，让用户选择编码字段，优化序列化和传输开销

举例

```
struct Resp {
    1: string Foo, // only serialize Foo
    2: i64 Bar  // won't serialize Bar
}
```

使用文档

用户构造了 Bar 的数据，但框架只会对 Foo 编码

```
respMask, err := fieldmask.NewFieldMask(
        (*Resp)(nil).GetTypeDescriptor(),
        "$.Foo")

func (s *BizServiceImpl) BizMethod1(ctx context.Context, req *biz.BizRequest) (resp *biz
    resp := NewResp() //
    resp.Foo = "Foo"
    resp.Bar = "Bar"

    resp.Set_FieldMask(respMask)
    return resp, nil
}
```

性能优化 | Thrift 内存分配优化

Span Cache

- 优化 String/Binary 解码开销:
  - 预分配内存，减少 mallocgc 调用
  - 减少实际生成的对象数量 -> 减少GC 开销

代码配置开启

```go
import "github.com/cloudwego/gopkg/protocol/thrift"
// v0.11.0
func main() {
    // add to the first line in main function
    thrift.SetSpanCache(true)
    // ...
}
```

容器字段集中分配内存

- 同理，原本对每个元素单独分配内存改为集中分配

优化效果

Kind	Concurrency	Data Size	QPS	P999
[KITEX-MUX]	100	1024	126325.23(+14.4%)	3.16(-9.7%)
[KITEX]	100	1024	106132.62(+9.9%)	1.83(-32.7%)

内存分析工具 | Go 对象引用分析 - goref

RPC/HTTP 的接收对象由框架构造、分配内存、赋值返回给用户，但用户代码里如果一直持有对象就会内存泄漏，但 pprof heap 只能告诉你哪里分配了，无法告诉你哪里引用了🤷‍♀️，那怎么知道 Go 对象到底被谁引用了呢?

示例 1

[Goref 使用说明]

$ grf attach ${PID}
successfully output to `grf.out`
$ go tool pprof -http=:5079 ./grf.out

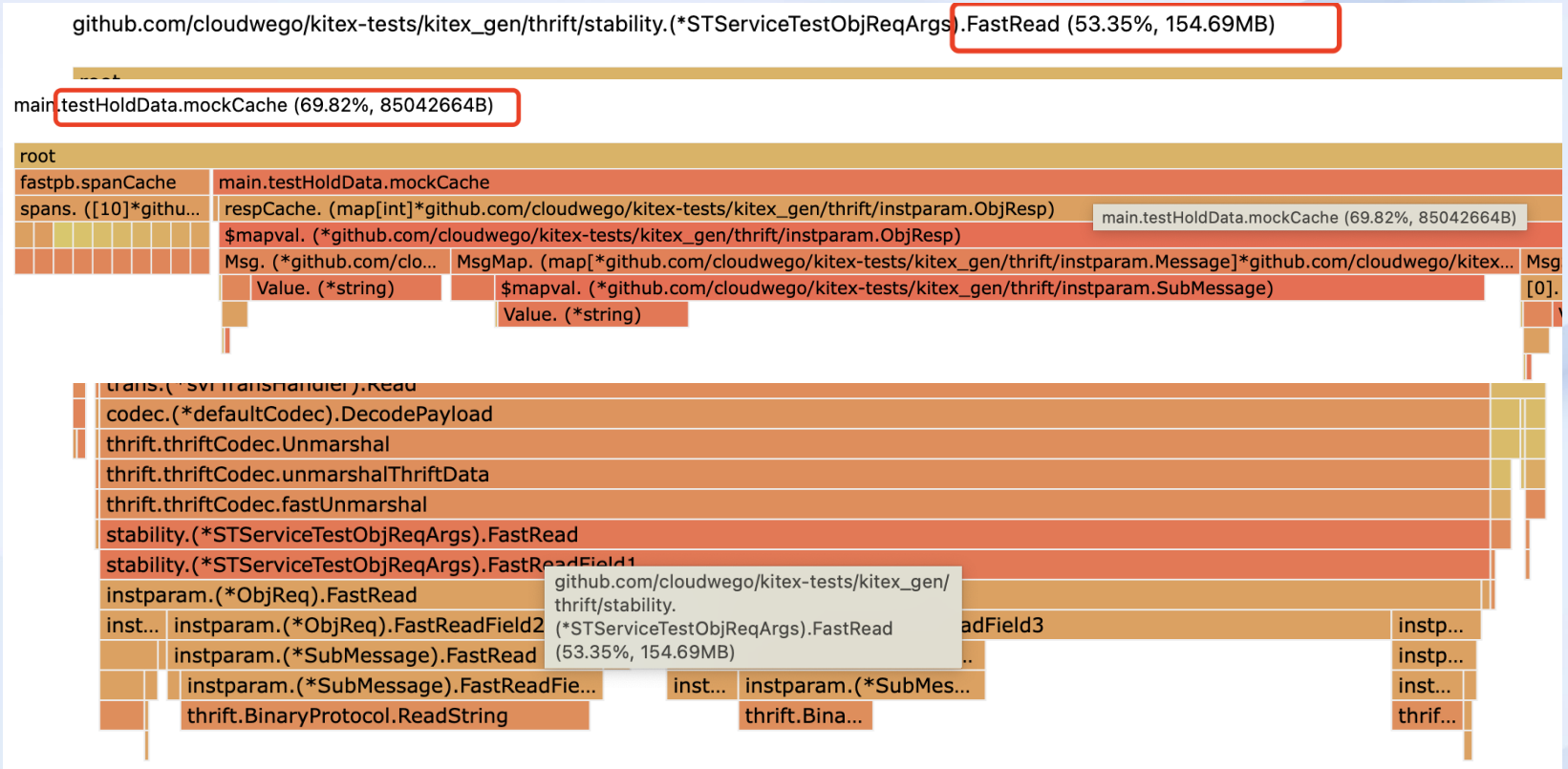


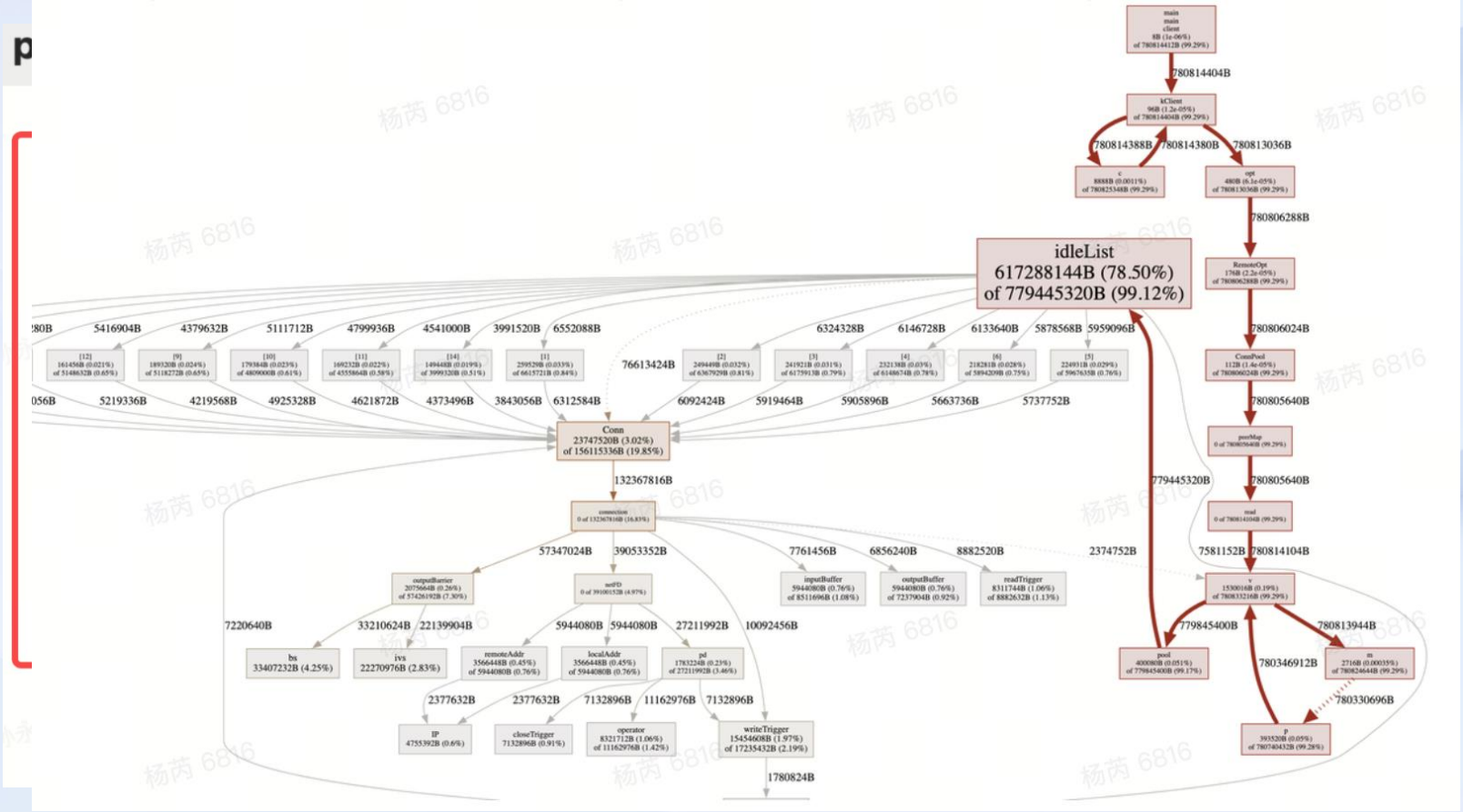
图1 是 pprof heap 火焰图
图2 是 goref 内存引用火焰图

内存分析工具 | Go 对象引用分析 - goref

示例 2

[Goref 使用说明]

```
$ grf attach ${PID}
successfully output to `grf.out`
$ go tool pprof -http=:5079 ./grf.out
```



03

总结 & 展望

助力大模型、新功能、用户体验/性能提升

总结

加强流式能力助力大模型

- Kitex/Hertz 提供的流式能力：gRPC、HTTP 1.1 Chunked、 WebSocket,、SSE、Thrift Streaming
- SSE <-> Thrift Streaming
- Streaming 泛化调用
- Streaming 能力的优化，提升用户体验、完善工程实践

新功能、用户体验/性能提升回顾

- 新功能：Thrift/gRPC 多 Service、Mixed Retry
- 用户体验：Frugal/FastCodec、产物精简和生成提速优化、 kitexcall
- 性能优化：Thrift 按需序列化、内存分配优化
- 内存分析工具：goref

展望 | 流式能力、用户体验相关

Streaming v2

- 发布 streaming v2 接口
- 发布 TTHeader Streaming 提升性能
- 工程实践：优雅退出、重试、超时控制
- 发布流式相关规范：错误规范、接口使用规范

Streaming 生态能力

- SSE <-> Thrift Streaming（HTTP2 and TTHeader Streaming）
- WebSocket <-> Thrift Streaming （HTTP2 and TTHeader Streaming）
- 二进制、Map 泛化调用

工具

- 增量代码更新
- 冗余字段消除
- FastCodec 代码重构：预计可减少代码产物同时进一步提升序列化性能
- Fastpb 进一步提升性能、完善功能如 unknown field

特别预告

Kitex 计划在 v0.12.0 默认去除 Apache Thrift 生成代码
目的： 去除 Apache Thrift 依赖
原因： Apache Thrift 0.14 接口的不兼容变更迫使 Kitex 绑定 0.13，给用户造成了不便

特别说明

虽然 Kitex 会发布 Streaming v2 接口，但会保留 v1 接口，不会造成编译问题让用户锁定版本。
用户可以选择使用 v2 接口。

THANKS