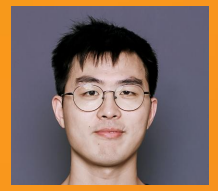
使用gRPC go实现
基于Topic的高效消息订阅发布模型



姓名 张凯

中国电子云

目 录

gRPC go介绍

gRPC是什么？

1. 什么是gRPC？gRPC go与gRPC的关系是什么？
2. gRPC与HTTP相比优势是什么？能否使用HTTP平替gRPC的实现？
3. 相比HTTP的swagger，gRPC为什么没有Swagger？
4. 如何判断一个项目是否适合用gRPC来做实现？
5. 如果要使用gRPC实现，有没有什么快速学习的方法？

1. 什么是gRPC？gRPC go与gRPC的关系是什么？
   -- gRPC是2015年Google开源，RPC的一种，底层基于HTTP2传输
   -- gRPC是一种框架，gRPC go是gRPC的一种实现；同理有其他各语言的实现。

2. gRPC与HTTP相比优势是什么？能否使用HTTP平替gRPC的实现？
   -- HTTP在Web服务、浏览器兼容性、简单易用性等方面仍有不可替代的优势
   -- http1.X + websocket可以做到长连接流式传输
   -- gRPC允许在一次连接中连续发送多个请求或响应，适用于实时数据流、大数据传输、视频传输

3. 相比HTTP的swagger，gRPC为什么没有Swagger？
   -- gRPC生成各语言的SDK，通过ProtoBuffer定义了输入输出的参数类型，与Swagger一样的效果
   -- swagger主是Web开发时方便与前端交互，可以使用gRPC Gateway起到一样的效果，
      通过调用对应插件可以生成Swagger

4. 如果要使用gRPC实现，有没有什么快速学习的方法？
   -- 官方github的Example是很好的学习场景：
      https://github.com/grpc/grpc-go/tree/master/examples/route_guide

6. gRPC跟protobuf 有啥区别？
   -- gRPC好比是http，protobuf好比是http传输时的restful json传输协议格式

7. go实现的rpc 和 java里面的rpc 有啥差异 ？
   -- rpc是远程过程调用，都需要通过每个语言的sdk调用，本质没差异

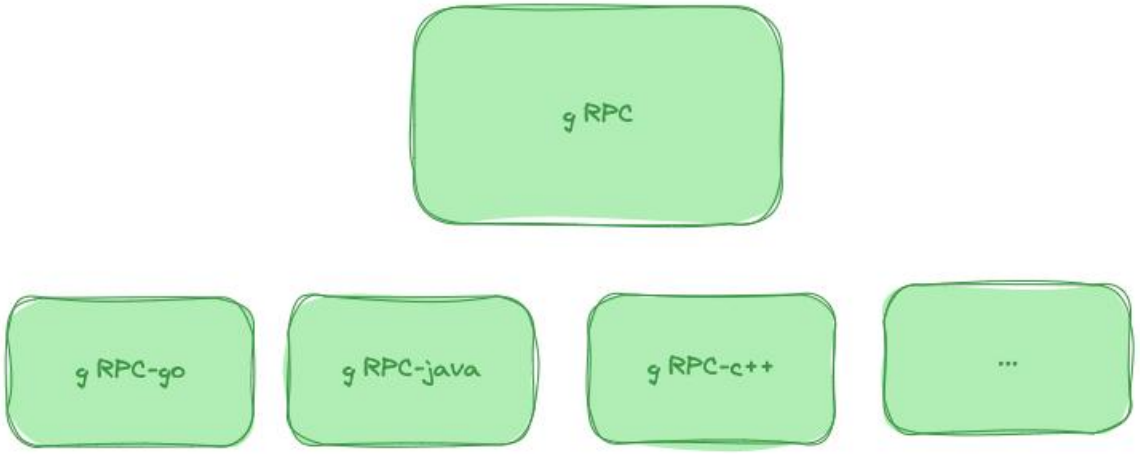
8. k8s内部实现用了大量gRPC，对外通信为什么又要用http？
   -- 对内使用gRPC加快传输过程，对外使用http，方便客户端调用

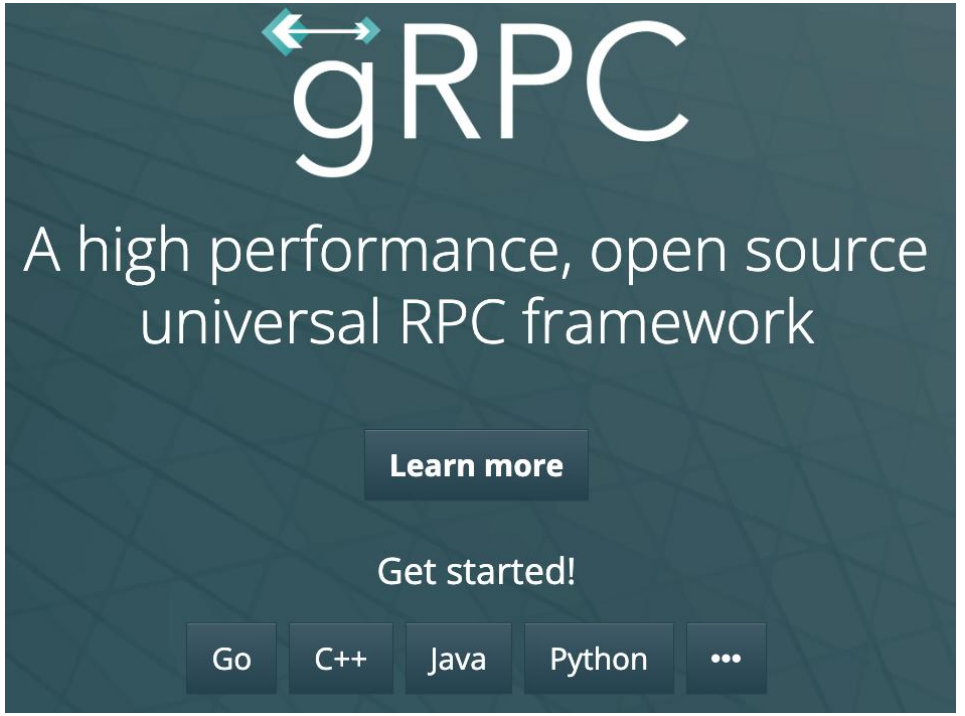
Why gRPC?

gRPC is a modern open source high performance Remote Procedure Call (RPC) framework that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication. It is also applicable in last mile of distributed computing to connect devices, mobile applications and browsers to backend services.

gRPC介绍

gRPC是框架，七层协议
gRPC go是实现，每个语言都有自己的实现





protobuf介绍

定义Service

Service是接口，需要被实现，实例化后注册到gRPC server，一个实例内存可被rpc访问与修改

调用的时候，获取Service Cient

定义RPC

一个rpc需要被实例化的Server去实现对应的方法，类似http里面的handler

调用的时候获取RPC Client

定义Message

定义rpc访问入参出参类型

protobuf示例

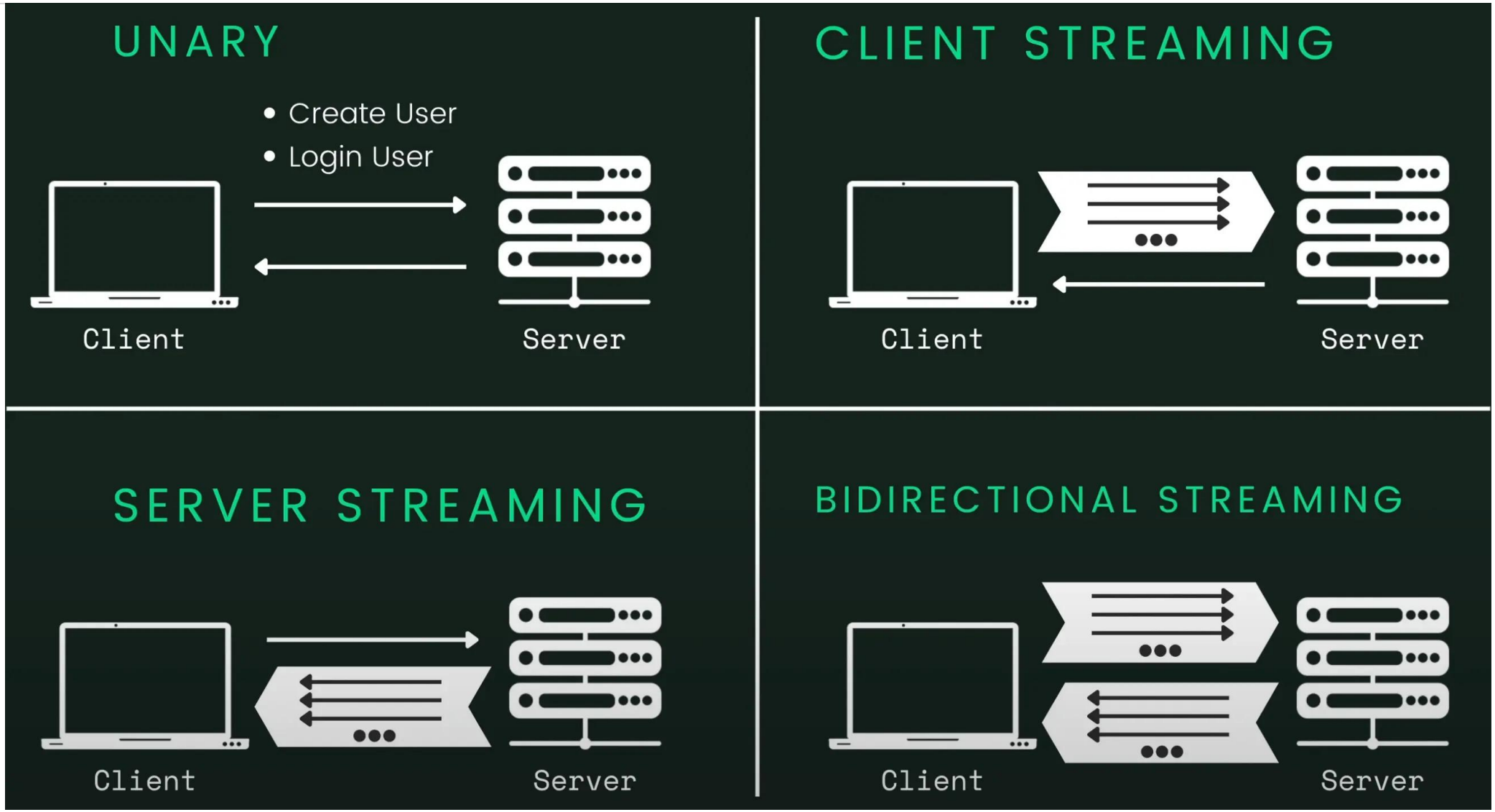
```
// 定义一个GetPersonService的服务，这个服务里有一个GetPerson的RPC方法
service GetPersonService {
  rpc GetPerson (GetPersonRequest) returns (Person) {}
}

// 定义一个Response Person的消息类型
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
  }

// 定义一个Request GetPersonRequest的消息类型
message GetPersonRequest {
  required string name = 1;
}
```

gRPC四种通信模式

gRPC四种通信模式


UNARY
• Create User
• Login User
Client
Server
CLIENT STREAMING
Client
Server
SERVER STREAMING
Client
Server
BIDIRECTIONAL STREAMING
Client
Server

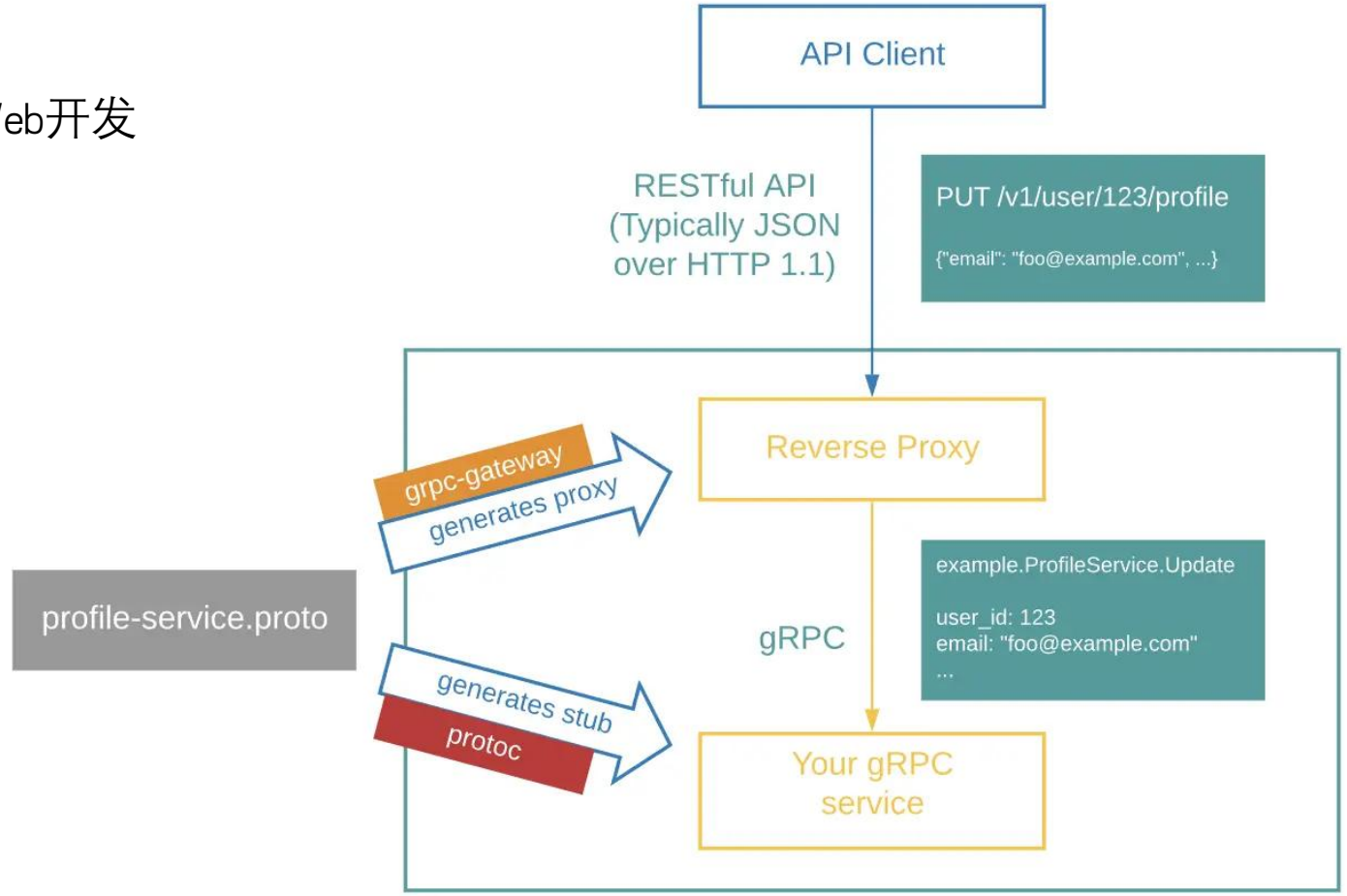
gRPC四种通信模式

1. **普通模式（Unary）**：
   - 这是最基本的消息交换方式，类似于HTTP中的简单请求-响应模式。客户端发送一个请求给服务器，服务器处理后返回一个响应。这是一种单向通信。
2. **服务器端流模式（Server Streaming）**：
   - 在这种模式下，客户端发送一个请求到服务器，然后服务器返回一系列响应消息。客户端只能接收数据，不能发送更多的数据到服务器。这是一种半双工通信，因为通信的方向从服务器到客户端是连续的，但客户端不能连续发送消息到服务器。
3. **客户端流模式（Client Streaming）**：
   - 客户端向服务器发送一系列请求消息，而服务器则在收到所有请求后返回一个响应消息。这种情况下，客户端可以连续发送多个请求，服务器端则是单向接收，因此也是半双工通信。
4. **双向流模式（Bidirectional Streaming 或 Full Duplex）**：
   - 双向流模式允许客户端和服务端同时发送和接收消息，形成一个真正的全双工通信通道。在这个模式下，双方都可以自由地发送多个请求/响应消息，就像一个实时聊天或者数据交换的场景。

gRPC的落地场景

基于通信模式如何落地？

Web开发

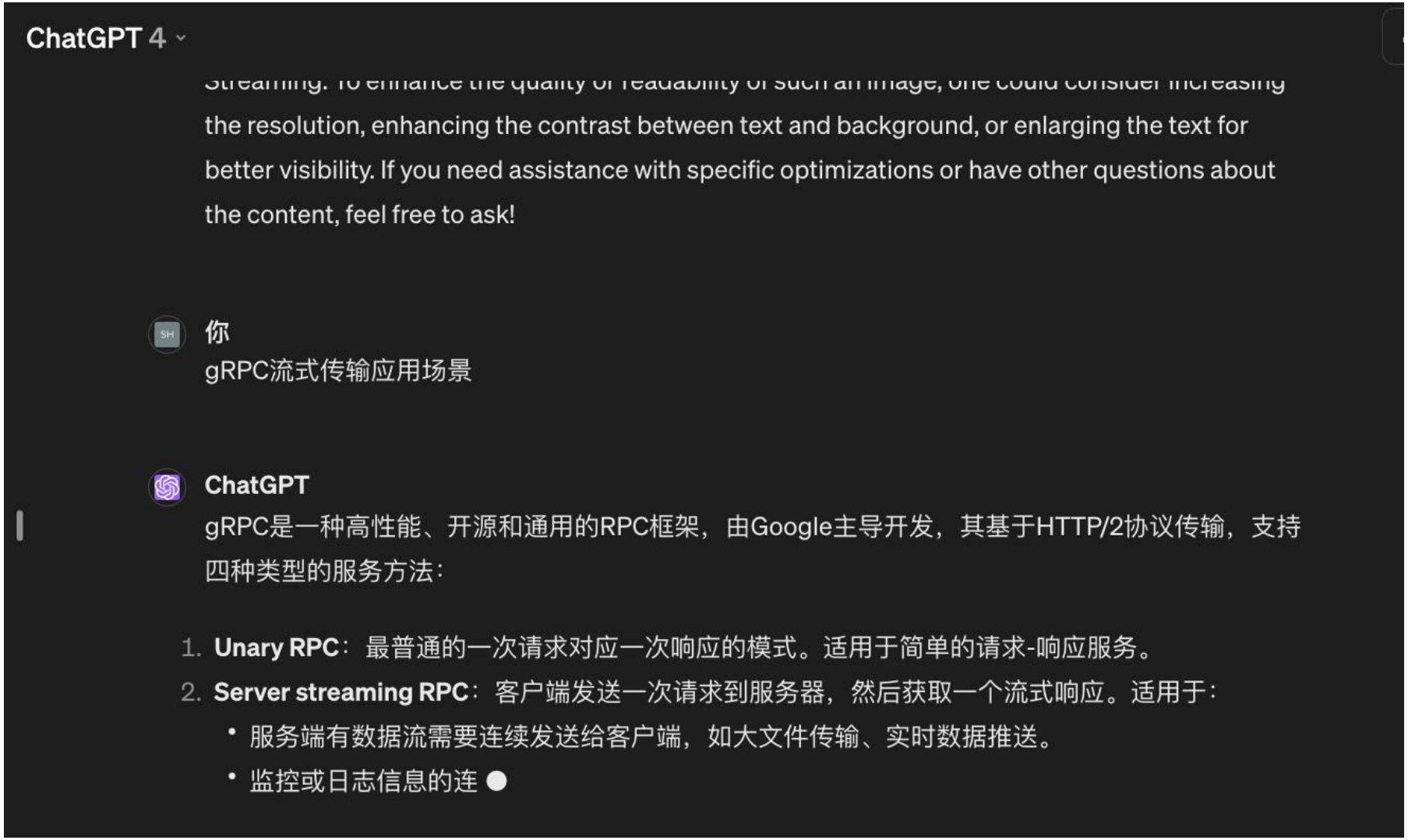


如果后端是协议gRPC，对于Web调用
可以通过gRPC Gateway转成http调用模式

unary模式，与http传输一样的效果

基于通信模式如何落地？

流式传输



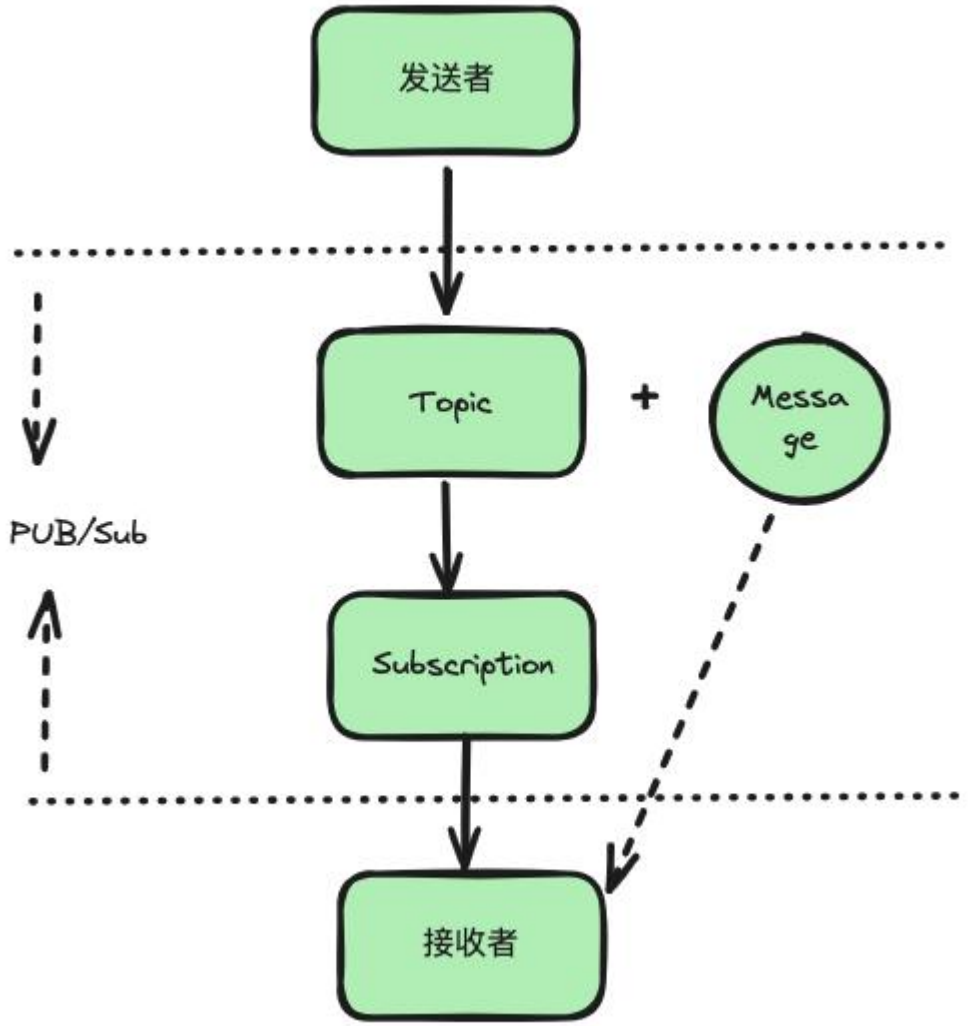
基于通信模式如何落地？

以及，本次分享的

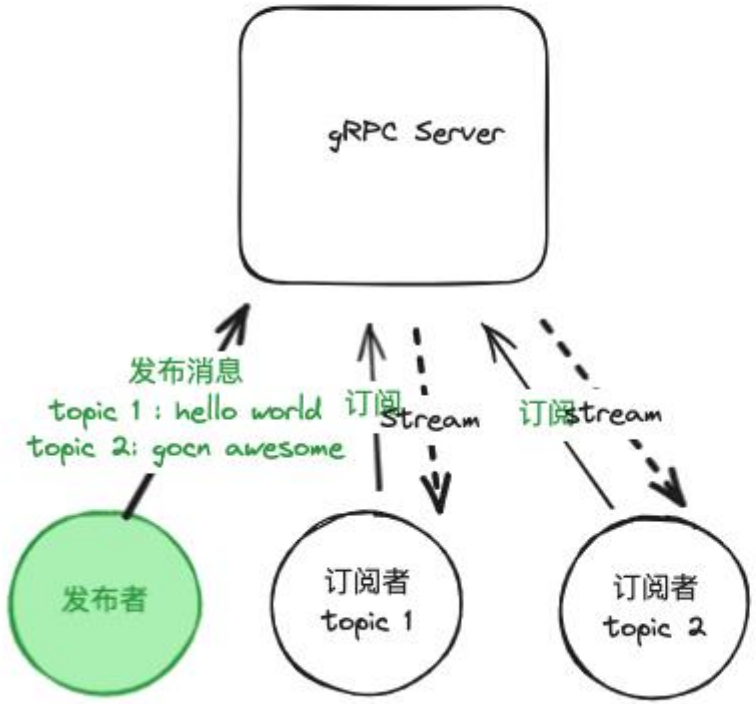
“基于TOPIC的消息订阅与发布模型”

基于Topic消息发布订阅

基于Topic的消息发布模型简介



基于Topic的消息发布模型简介




```
service PubSub {
  rpc Publish (PublishRequest) returns (google.protobuf.Empty);
  rpc Subscribe (SubscribeRequest) returns (stream SubscribeResponse);
}

// PublishRequest is passed when publishing
message PublishRequest {
  string topic = 1;
  bytes payload = 2;
}

// SubscribeRequest is passed when subscribing
message SubscribeRequest {
  string topic = 1;
}

// SubscribeResponse object
message SubscribeResponse {
  bytes payload = 1;
}
```

“

BRIAN KERNIGHAN

```
service PubSubService {
    rpc Publish(PublishRequest) returns (google.protobuf.Empty);
    rpc Subscribe(SubscribeRequest) returns (stream SubscribeResponse);
}
message PublishRequest {
    string topic = 1;
    bytes payload = 2;
}

message SubscribeRequest {
    string topic = 1;
}

message SubscribeResponse {
    bytes payload = 1;
}
```

发布接口Pub：Unary模式，发送主题及其内容

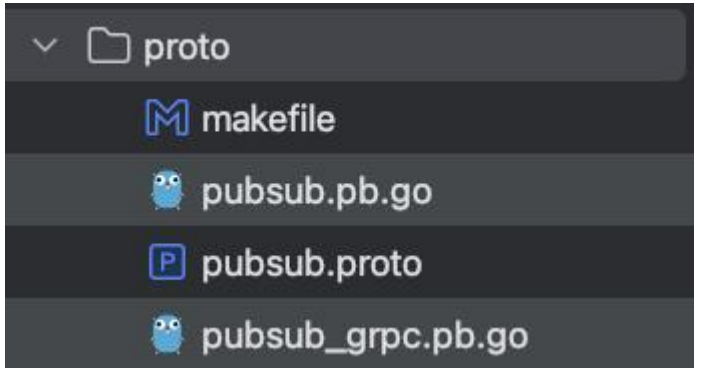
订阅接口Sub：服务端流模式，订阅主题后，
建立长连接，服务端推送消息

根据Proto生成桩代码

根据Proto生成桩代码

protoc --go-grpc_out=. --go_out=. pubsub.proto

- protoc: 这是 Protocol Buffers 编译器的执行命令。
- --go-grpc_out=.: 这个选项指定了输出 gRPC 相关的 Go 语言代码的目标目录，. 表示当前目录。protoc-gen-go-grpc 插件将会处理 .proto 文件中的服务定义并生成 gRPC 服务器和客户端的接口代码。
- --go_out=.: 类似地，这个选项指定了非 gRPC 相关的 Protocol Buffers 消息结构体等 Go 语言代码的输出目录，同样为当前目录。protoc-gen-go 插件会处理 .proto 文件中的消息定义并生成对应的 Go 结构体。



订阅者动态注册

```
type subscriber chan interface{}
type topicFunc func(v interface{}) bool

type Publisher struct {
        m           sync.RWMutex
        buffer      int
        timeout     time.Duration
        subscribers map[subscriber]topicFunc // chan中是数据，函数判断数据是否包含
topic
}

func (p *Publisher) SubscribeTopic(topic topicFunc) chan interface{} {
        ch := make(chan interface{}, p.buffer)
        p.m.Lock()
        p.subscribers[ch] = topic  // 动态注册，保存到map中
        p.m.Unlock()
        return ch
}
```

发布者消息推送

```
func (p *Publisher) Publish(v interface{}) {
        ...
        // 发布时，轮询所有注册上来的对象
        for sub, topic := range p.subscribers {
                ...
                go p.sendTopic(sub, topic, v, wg)   // 消息推送到订阅者的chan
        }

func (p *Publisher) sendTopic(sub subscriber, topic topicFunc, v interface{}...) {
    select {
        case sub <- v:
        default:  // default分支为空，也就是说如果没有通道准备好接收数据，则什么也不做，程序会阻塞在此处，直到至少有一个通道准备好为止。
        }
}
```

gRPC Server启动

```go
func main() {
    lis, _ := net.Listen("tcp", "127.0.0.1:1234")
    opts := []grpc_recovery.Option{
        grpc_recovery.WithRecoveryHandlerContext(func(ctx context.Context, rec interface{}) (e
            klog.Warningf("Recovered in f %v", rec)
            return grpc.Errorf(codes.Internal, "Recovered from panic")
        }),
    }
    grpcServer := grpc.NewServer(
        grpc.KeepaliveEnforcementPolicy(keepalive.EnforcementPolicy{
            MinTime:             5 * time.Second,
            PermitWithoutStream: true, // Allow pings even when there are no active streams
        }),
        grpc.KeepaliveParams(keepalive.ServerParameters{
            Time:    2 * time.Hour,
            Timeout: 20 * time.Second,
        }),
        grpc_middleware.WithUnaryServerChain(
            grpc_recovery.UnaryServerInterceptor(opts...),
        ),
        grpc_middleware.WithStreamServerChain(
            grpc_recovery.StreamServerInterceptor(opts...),
        ),
    )
    klog.Info("start server, listen: 1234")
    proto.RegisterPubSubServiceServer(grpcServer, service.NewService())
    if err := grpcServer.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
    klog.Warningf("stop server")
    grpcServer.GracefulStop()
}
```

注册服务到grpcServer

```go
type Service struct {
    bus *pubsub.Publisher
    proto.UnimplementedPubSubServiceServer
}
```

实现protoBuf 定义rpc的接口

```go
func (s *Service) Publish(ctx context.Context, in *proto.PublishRequest) (*emptypb.Empty, error) {
    data := in.GetTopic() + ":// " + string(in.GetPayload())
    s.bus.Publish(data)
    return &emptypb.Empty{}, nil
}

func (service *Service) Subscribe(in *proto.SubscribeRequest, stream proto.PubSubService_SubscribeServer) error {

    fn := func(v interface{}) bool {
        if s, ok := v.(string); ok {
            return strings.HasPrefix(s, in.Topic+":// ")
        }
        return false
    }

    for v := range service.bus.SubscribeTopic(fn) {
        err := stream.Send(&proto.SubscribeResponse{Payload: []byte(v.(string)[len(in.Topic+":// "):])})
        if err != nil {
            return err
        }
    }

    klog.Info("stream closed")
    return nil
}
```

```go
func NewService() *Service {
    publisher := pubsub.NewPublisher(1*time.Second, 1)
    return &Service{bus: publisher}
}
```

实例化

发送端发布消息

```go
func main() {
    conn, err := grpc.Dial("localhost:1234", grpc.WithInsecure())
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    client := proto.NewPubSubServiceClient(conn)

    _, err = client.Publish(context.Background(), &proto.PublishRequest{Topic: "gocn", Payload: []byte("hello, gophers!")})
    if err != nil {
        log.Fatal(err)
    }

    _, err = client.Publish(context.Background(), &proto.PublishRequest{Topic: "greeting", Payload: []byte("hello, world!")})
    if err != nil {
        log.Fatal(err)
    }
}
```

gRPC 连接时允许非 TLS

获取 service 的 client

获取 pub 端 client，发送消息。
连接发送两条消息，同一个连接。
发送完后，程序退出，连接断开

订阅端监听消息

```go
func main() {
    conn, err := grpc.Dial("localhost:1234", grpc.WithInsecure())
    if err != nil {
        log.Fatal(err)
    }
    client := proto.NewPubSubServiceClient(conn)
    subscribeClient, err := client.Subscribe(context.Background(), &proto.SubscribeRequest{Topic: "gocn"})
    if err != nil {
        return
    }

    for {
        resp, err := subscribeClient.Recv()
        if err != nil {
            if err == io.EOF {
                break
            }
            log.Fatal(err)
        }
        log.Println(string(resp.Payload))
    }
    defer conn.Close()
}
```

连接 Server

初始化 service 客户端

获取订阅 rpc 的客户端

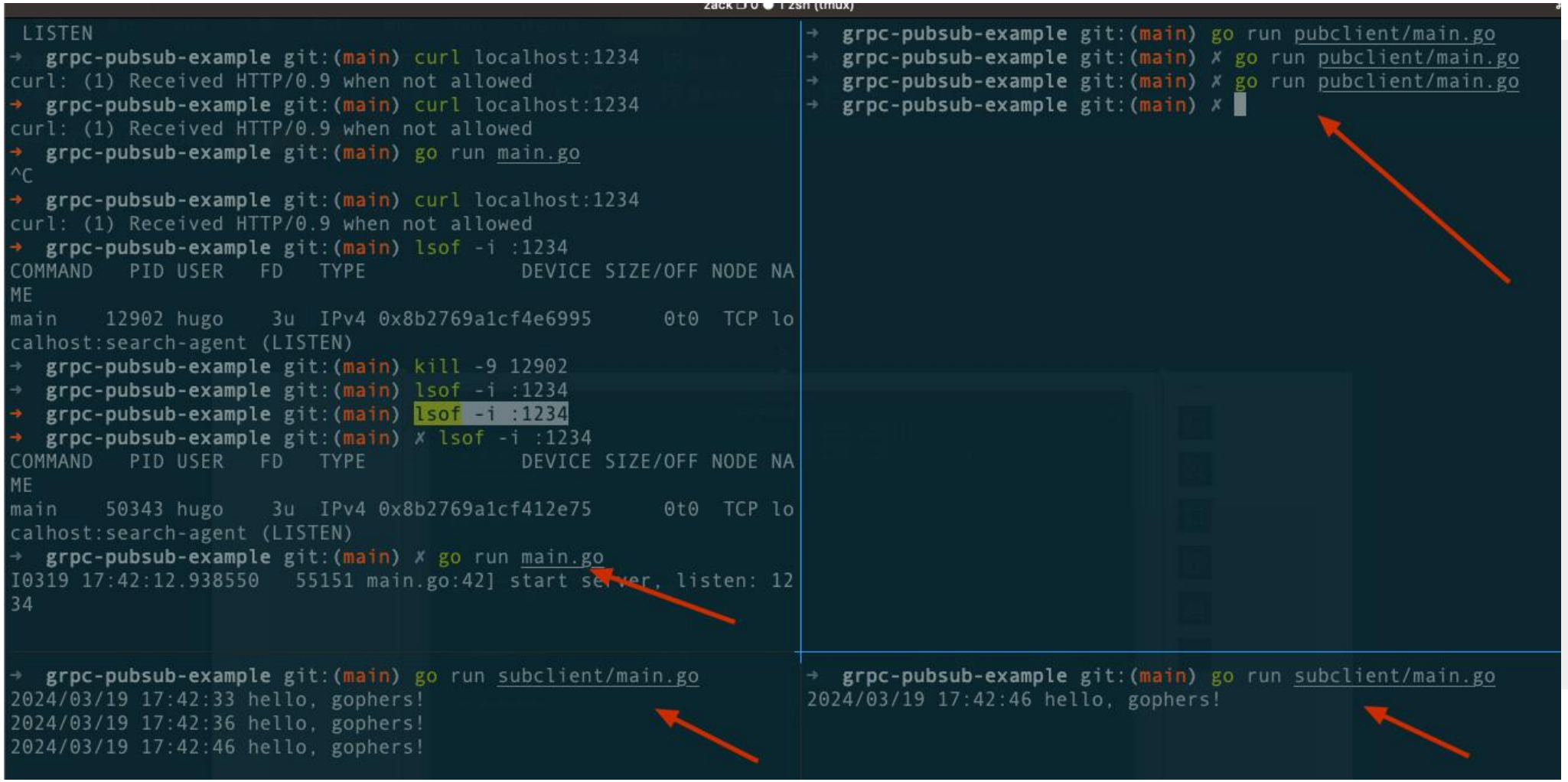
获取字节流，Recv 阻塞，当有字节流时，循环读取

通道关闭标识符 EOF，退出程序

演示

代码示例：

https://github.com/zackzhangkai/grpc-pubsub-example


LISTEN
→ grpc-pubsub-example git:(main) curl localhost:1234
curl: (1) Received HTTP/0.9 when not allowed
→ grpc-pubsub-example git:(main) curl localhost:1234
curl: (1) Received HTTP/0.9 when not allowed
→ grpc-pubsub-example git:(main) go run main.go
^C
→ grpc-pubsub-example git:(main) curl localhost:1234
curl: (1) Received HTTP/0.9 when not allowed
→ grpc-pubsub-example git:(main) lsof -i :1234
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NA
ME
main 12902 hugo 3u IPv4 0x8b2769a1cf4e6995 0t0 TCP lo
calhost:search-agent (LISTEN)
→ grpc-pubsub-example git:(main) kill -9 12902
→ grpc-pubsub-example git:(main) lsof -i :1234
→ grpc-pubsub-example git:(main) lsof -i :1234
→ grpc-pubsub-example git:(main) ✗ lsof -i :1234
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NA
ME
main 50343 hugo 3u IPv4 0x8b2769a1cf412e75 0t0 TCP lo
calhost:search-agent (LISTEN)
→ grpc-pubsub-example git:(main) ✗ go run main.go
I0319 17:42:12.938550 55151 main.go:42] start server, listen: 12
34
→ grpc-pubsub-example git:(main) go run pubclient/main.go
→ grpc-pubsub-example git:(main) ✗ go run pubclient/main.go
→ grpc-pubsub-example git:(main) ✗ go run pubclient/main.go
→ grpc-pubsub-example git:(main) ✗
→ grpc-pubsub-example git:(main) go run subclient/main.go
2024/03/19 17:42:33 hello, gophers!
2024/03/19 17:42:36 hello, gophers!
2024/03/19 17:42:46 hello, gophers!
→ grpc-pubsub-example git:(main) go run subclient/main.go
2024/03/19 17:42:46 hello, gophers!

谢谢大家