GO-12 03: gRPC Streams, interceptors

Описание:

gRPC имеет ещё ряд других типов взаимодействия и возможностей. В прошлом задании нами был изучен унарный тип взаимодействия, когда клиент делает один блокирующий запрос и получает один ответ. Всего в grpc четыре таких типа запросов:

- унарный (Unary RCP);
- серверный стрим (Server streaming), когда при подключении клиента сервер открывает стрим и начинает передачу сообщений клиенту;
- клиентский стрим (Client streaming), когда клиент, подключаясь к серверу, передает на сервер сообщения;
- двунаправленный стрим (Bidirectional streaming), когда при соединении открывается 2 потока в разных направлениях.

В этот момент чаще всего можно встретить использование унарного типа взаимодействия, но стримы тоже крайне полезны и могут применяться для разных кейсов, где нужна быстрая передача нескольких сообщений получателю.
Например,К примеру, можно фиксировать поведения клиента на сайте в рамках одной сессии. Создать gprc stream и отправлять все его действия в базу, elastic или еще куда-нибудь. Также через стримы может производиться загрузка или потоковая передача файлов. В общем, применить эту технологию можно во многих кейсах. Теперь давайте рассмотрим работу стрима в Go-приложениях.
Объявим proto файл:

```
syntax = "proto3";
option go_package="productservice;productservice";
package product;

service Products {
 rpc GetProducts (Filter) returns (stream Product) {}
}

message Product {
 int32 Price = 1;
 string Name = 2;
 string Category = 3;
}

message Filter {
 string Category = 1;
}
```

У нас будет один gRPC вызов GetProduct, который принимает структуру Filter и возвращает стрим продуктов stream Product.
Если нам необходимо объявить серверный стрим, то в возвращаемом значении нужно указать директиву stream, если клиентский стрим - то в аргументе. А если необходим двунаправленный стрим, то и в аргументе, и в возвращаемом значении.
В нашем случае директива указана в возвращаемом значении, а значит, мы объявили серверный стрим. Сгенерируем пакет productservice и запустим с помощью него сервер.
Напишем структуру Server, в которой опишем метод получения продуктов. Мы просто получаем объект из заранее проинициализированного массива и отправляем его в стрим stream.Send(p). Стрим будет считаться активным до тех пор, пока мы не вернем в функции GetProducts nil или ошибку.

```
//файл internal/handlers/server.go
package handlers

import "12_03/pkg/productservice"

type Server struct {
  Products []*productservice.Product
  productservice.UnimplementedProductsServer
}

func NewProductServer() Server {
  s := Server{}
  s.Products = []*productservice.Product{
     &productservice.Product{
        Price:    100000,
        Name:     "Iphone 12 pro",
        Category: "Smartphone",
     },
     &productservice.Product{
        Price:    214000,
        Name:     "MacBook pro 16",
        Category: "Laptop",
     },
     &productservice.Product{
        Price:    99000,
        Name:     "MacBook Air",
        Category: "Laptop",
     },
     &productservice.Product{
```

```
            Price:    69000,
            Name:     "Iphone 12 mini",
            Category: "Smartphone",
        },
        &productservice.Product{
            Price:    79000,
            Name:     "Iphone 12",
            Category: "Smartphone",
        },
    }

    return s
}

func (s Server) GetProducts(filter *productservice.Filter, stream
productservice.Products_GetProductsServer) error {
    for _, p := range s.Products {
        if p.Category == filter.Category {
            err := stream.Send(p)
            if err != nil {
                return err
            }
        }
    }

    return nil
}
```

Далее в main пакете начинаем слушать порт и регистрируем структуру Server в качестве grpc сервера.

```
//файл cmd/server/server.go
package main

import (
    "12_03/internal/handlers"
    "12_03/pkg/productservice"
    "fmt"
    "google.golang.org/grpc"
    "net"
```

```
)

func main()  {
  fmt.Println("Starting server ...")
  lis, err := net.Listen("tcp", ":50051")
  if err != nil {
     panic(err)
  }

  s := grpc.NewServer()
  server := handlers.NewProductServer()
  productservice.RegisterProductsServer(s, &server)
  if err := s.Serve(lis); err != nil {
     panic(err)
  }
}
```

Сервер готов. Осталось написать клиент, который сделает запрос и дальше будет вычитывать сообщения из стрима.

```
//файл cmd/client/client.go
package main

import (
  "12_03/pkg/productservice"
  "context"
  "fmt"
  "google.golang.org/grpc"
  "io"
  "time"
)

func main() {
  cwt, _ := context.WithTimeout(context.Background(), time.Second*5)
  conn, err := grpc.DialContext(cwt, "localhost:50051",
grpc.WithInsecure(), grpc.WithBlock())
  if err != nil {
     panic(err)
  }
  defer conn.Close()
```

```
  pc := productservice.NewProductsClient(conn)

  filter := &productservice.Filter{
     Category: "Smartphone",
  }

  productsStream, err := pc.GetProducts(cwt, filter)
  if err != nil {
     panic(err)
  }

  for {
     product, err := productsStream.Recv()
     if err == io.EOF {
        break
     }
     if err != nil {
        panic(err)
     }
     fmt.Println(product)
  }
}
```

Функция GetProducts возвращает нам данные, которые нужно вычитывать в бесконечном цикле, до тех пор, пока не вернется ошибка EOF, которая обозначает, что стрим закрыт.
Запустим сервер и клиент. Посмотрим на вывод
Сервер

```
Starting server ...
```

Клиент

```
Price:100000 Name:"Iphone 12 pro" Category:"Smartphone"
Price:69000 Name:"Iphone 12 mini" Category:"Smartphone"
Price:79000 Name:"Iphone 12" Category:"Smartphone"
```

Как и следовало предполагать, мы получили все сообщения, стрим закрылся и мы окончили выполнения client.go.
Давайте теперь рассмотрим Interceptors - это аналоги middleware для grpc.
Мы сделаем так, чтобы каждый запрос к серверу условно логировался и передавал условные метрики. Из коробки библиотека "google.golang.org/grpc" поддерживает только

один интерсептор, что нас не очень устраивает. Поэтому для решения этого момента мы воспользуемся пакетом "github.com/grpc-ecosystem/go-grpc-middleware", он позволяет создавать цепочки вызовов интерсепторов.

```
//файл cmd/server/server.go
package main

import (
  "12_03/internal/handlers"
  "12_03/pkg/productservice"
  "fmt"
  "github.com/grpc-ecosystem/go-grpc-middleware"
  "google.golang.org/grpc"
  "net"
)

func main()  {
  fmt.Println("Starting server ...")
  lis, err := net.Listen("tcp", ":50051")
  if err != nil {
     panic(err)
  }

  s :=
grpc.NewServer(grpc_middleware.WithStreamServerChain(metricInterceptor
, logInterceptor))
  server := handlers.NewProductServer()
  productservice.RegisterProductsServer(s, &server)
  if err := s.Serve(lis); err != nil {
     panic(err)
  }
}

func metricInterceptor(srv interface{}, ss grpc.ServerStream, info
*grpc.StreamServerInfo, handler grpc.StreamHandler)  error {
  fmt.Println("Metric Interceptor")
  err := handler(srv, ss)
  if err != nil {
     return err
  }
  return nil
```

```
}


func logInterceptor(srv interface{}, ss grpc.ServerStream, info
*grpc.StreamServerInfo, handler grpc.StreamHandler)  error {
  fmt.Println("Log Interceptor")
  err := handler(srv, ss)
  if err != nil {
     return err
  }
  return nil
}
```

Описали 2 функции metricInterceptor и logInterceptor, которые имплементируют сигнатуру метода grpc.StreamServerInterceptor. Затем передаем эти методы в функцию grpc.NewServer(grpc_middleware.WithStreamServerChain(metricInterceptor, logInterceptor)), которая реализует цепочку вызовов интерсепторов.
Запускаем, обращаемся к серверу через клиент и видим, что оба интерсептора отработали при запросе метода GetProducts.

```
Starting server ...
Metric Interceptor
Log Interceptor
```

Полезные ссылки:

- Writing gRPC Interceptors in Go
- A basic tutorial introduction to gRPC in Go.
- Sending files via gRPC

Задание:

Во время выполнения задания можете ориентироваться на этот репозиторий - module12_03.

1. Создайте у себя в проекте из предыдущего задания module12 ветку module12_03.
2. Реализуйте передачу файлов между микросервисами из задания 12_01 через grpc stream, вычитывая файл в байтах чанками и передавая его в клиентский стрим.
3. На стороне сервера реализуйте интерсептор, который залогирует размер передаваемого на сервер файла.

4. В ответе пришлите ссылку на merge request в ветку master своего проекта ветки module12_03.