GO-12 02: Взаимодействие через gRPC

Описание:

В прошлом задании мы разобрали взаимодействие сервисов посредством HTTP REST. Это очень распространенный способ. Но если речь идет о go, то вероятно и о микросервисах (так как go очень часто используется для создания микросервисов). И обычно бэкенд системы состоят из сотен микросервисов, что создает очень большой трафик в сети в момент их взаимодействия. И если оно происходит через REST, то в современном времени это не очень актуально, так как:

- HTTP 1.1 и REST достаточно тяжеловесные.
- Создают новое tcp-соединение для каждого параллельного запроса.
- Тело и заголовки передаются не в сжатом виде, используется json, который сам по себе несет лишние данные.

И когда микросервисов (а как следствие, и запросов) становится все больше и больше, то напрашивается необходимость оптимизировать сам протокол. Так и было сделано разработчиками из Google. В 2015 году появился gRPC - фреймворк для вызова удаленных процедур, использующий HTTP/2 в качестве транспорта и protobuf для сериализации данных. Благодаря более легковесному протоколу HTTP/2 и бинарной сериализации данных в protobuf, gRPC в разы легче и быстрее, чем REST. Как удалось достигнуть этой оптимизации, можно узнать, ознакомившись со статьями в полезных ссылках.

После того, как мы выяснили, что для взаимодействия микросервисов gRPC подходит практически идеально, давайте рассмотрим, как с ним работать именно на Go.

В прошлом задании мы генерировали для сервисов REST клиент и сервер из спецификации swagger. По этому же принципу создается сервер и клиент для gRPC. Но в качестве спецификации и описания API в gRPC используется protobuf и файл .proto.

Посмотрим, как выглядит файл users.proto:

```
syntax = "proto3";
option go_package="userservice;userservice";
package users;

service Users {
 rpc CreateUser (User) returns (UserState) {}
 rpc GetUsers (Filter) returns (UsersList) {}
}

message User {
```

```
 string Email = 1;
 string FirstName = 2;
 string LastName = 3;
 string password = 4;
}

message UsersList {
 repeated User list = 1;
}

message Filter {
 repeated int32 Ids = 1;
}

message UserState {
 bool Success = 1;
}
```

Предположим, у нас есть CRUD сервис, который отвечает за пользователей (создание, получение списка). Мы генерируем для этого сервиса gRPC сервер, а для всех потребителей - gRPC клиенты. В proto файле описана строго типизированная спецификация взаимодействия между сервером и клиентом. Рассмотрим подробнее:

- syntax = "proto3"; - это определение версии спецификации proto.
- option go_package="userservice;userservice"; - proto поддерживает дополнительные опции, которые потом могут использоваться различными плагинами. В этом случае мы будем использовать плагин для генерации proto в go и передаем путь и название вновь сгенерированного пакета в контексте go.
- package users; - название пакета в контексте protobuf.

Далее определяем сервис, в котором 2 метода - создание пользователя и получение списка пользователей.

```
service Users {
 rpc CreateUser (User) returns (UserState) {}
 rpc GetUsers (Filter) returns (UsersList) {}
}
```

На вход методу CreateUser передается User - это message в контексте proto, а после генерации это будет структура в контексте Go.

```
message User {
 string Email = 1;
 string FirstName = 2;
 string LastName = 3;
```

```
 string password = 4;
}
```

Message в proto определяется таким образом: указывается тип, название поля и через знак равенства его номер. Номер поля необходим для того, чтобы не передавать название поля целиком, это экономит передаваемый по сети трафик. Возвращает же метод CreateUser также message/go структуру, в которой одно поле - Success, оно скажет нам, успешно ли создался пользователь.
Далее еще один аналогичный метод - GetUsers (Filter) returns (UsersList). Он получает структуру Filter и возвращает UsersList список пользователей. Обратите внимание, что массив в proto объявляется через директиву repeated repeated int32 Ids = 1;.
Итак, у нас есть proto, в котором объявлен сервис с двумя методами. И теперь на основании этого сервиса мы сгенерируем go сервер и клиент для обращения к нему.
Но вначале нам нужно установить protoc - утилиту, которая умеет генерировать код из .proto файла. А также установить пару плагинов для генерации именно go кода.
Установка protobuf и плагинов
Определите переменную PROTO_OS_VERSION для своей OS:

```
$ PROTO_OS_VERSION="osx-x86_64"
$ PROTO_OS_VERSION="linux-x86_64"
$ PROTO_OS_VERSION="linux-x86_32"
$ PROTO_OS_VERSION="win32"
$ PROTO_OS_VERSION="win64"
```

Далее выполните команды для непосредственной установки:

```
$ PB_REL="https://github.com/protocolbuffers/protobuf/releases"
$ curl -LO
$PB_REL/download/v3.13.0/protoc-3.13.0-$PROTO_OS_VERSION.zip
$ unzip protoc-3.13.0-$PROTO_OS_VERSION.zip -d $HOME/.local
$ export PATH="$PATH:$HOME/.local/bin

go get -u google.golang.org/protobuf/cmd/protoc-gen-go
go install google.golang.org/protobuf/cmd/protoc-gen-go

go get -u google.golang.org/grpc/cmd/protoc-gen-go-grpc
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc
```

Проверяем установку

```
$ protoc  --version
libprotoc 3.13.0
```

Теперь добавим наш proto файл в проект:

```
~/12_02
    └── api
        └── users.proto
    └── cmd
            └── client
                └── client.go
            └── server
                └── server.go
    └── pkg
    └── internal
    └── go.mod
```

И сгенерируем go пакет с помощью команды:

```
protoc  api/*.proto --go-grpc_out=pkg --go_out=pkg
```

Первый аргумент - это путь к proto'файлам. А --go-grpc_out --go_out - это путь, куда будет помещен сгенерированный пакет. После успешного выполнения команды в pkg появится пакет userservice:

```
~/12_02
    └── api
        └── users.proto
    └── cmd
            └── client
                └── client.go
            └── server
                └── server.go
    └── internal
    └── pkg
            └── userservice
                └── users.pb.go
                └── users_grpc.pb.go
    └── go.mod
```

Сгенерированные файлы нельзя редактировать. В них определены структуры и методы клиента и сервера. Также есть интерфейс, который нужно будет имплементировать для нашего сервера.

```
// UsersServer is the server API for Users service.
// All implementations must embed UnimplementedUsersServer
// for forward compatibility
type UsersServer interface {
```

```
  CreateUser(context.Context, *User) (*UserState, error)
  GetUsers(context.Context, *Filter) (*UsersList, error)
  mustEmbedUnimplementedUsersServer()
}
```

Дальше в internal/handlers/server.go создаем структуру Server и прописываем в ней реализацию методов интерфейса UsersServer:

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

import (
  "12_02/pkg/userservice"
  "context"
  "errors"
)

type Server struct {
  Users []*userservice.User
  userservice.UnimplementedUsersServer
}

func (s *Server) CreateUser(ctx context.Context, user
*userservice.User) (*userservice.UserState, error) {
  if user.Email == "" || user.FirstName == "" || user.Password == "" {
     return &userservice.UserState{Success: false}, errors.New("Wrong
user ")
  }
  s.Users = append(s.Users, user)

  return &userservice.UserState{Success: true}, nil
}

func (s *Server) GetUsers(ctx context.Context, filter
*userservice.Filter) (*userservice.UsersList, error)  {
  return &userservice.UsersList{List: s.Users}, nil
}
```

Теперь в файле cmd/server/server.go определяем, на каком порту будем слушать grpc вызовы и передаем на вход серверу нашу структуру handlers.Server:

```go
package main

import (
  "12_02/internal/handlers"
  "12_02/pkg/userservice"
  "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.Server{}
  userservice.RegisterUsersServer(s, server)
  if err := s.Serve(lis); err != nil {
     panic(err)
  }
}
```

Запускаем наш grpc сервер:

```
$ go run cmd/server/server.go
Starting server ...
```

Далее создаем клиента и делаем через него 2 запроса CreateUser и GetUsers:

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

import (
  "12_02/pkg/userservice"
  "context"
  "fmt"
  "google.golang.org/grpc"
  "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()

  uc := userservice.NewUsersClient(conn)


  user := &userservice.User{
     Email:     "example@main.ru",
     FirstName: "Nikolai",
     LastName:  "Orlov",
     Password:  "12345",
  }


  us, err :=  uc.CreateUser(cwt, user)
  if err != nil {
     panic(err)
  }
  if us.Success {
     fmt.Println("User successfully created")
  }

  users, err := uc.GetUsers(cwt, &userservice.Filter{})
  if err != nil {
     panic(err)
  }

  fmt.Println(users)
}
```

Тут, используя пакет "google.golang.org/grpc", мы создали подключение к запущенному сервису. Затем проимпортировали пакет userservice. Получили из пакета клиент для общения с сервером. Вызвали соответствующие методы.

Запускаем клиент

```
$ go run cmd/client/client.go
User successfully created
list:{Email:"example@main.ru"  FirstName:"Nikolai"  LastName:"Orlov"
password:"12345"}
```

Отлично! Мы создали нового пользователя и получили список пользователей через grpc. Как видно из этого примера, использовать grpc в go достаточно просто и удобно. Необходимо сгенерировать с помощью protoc пакет, который содержит в себе код и клиента, и сервера. А дальше в одном микросервисе имплиментировать интерфейс сервера, а в других - использовать структуру клиента.
Также если у одного сервиса много клиентов, то удобным способом будет вынести сгенерированный пакет (userservice в нашем случае) в отдельный Go модуль. И просто импортировать его в новый микросервис, которому необходимо взаимодействовать с сервером.

Полезные ссылки:

- Введение в HTTP/2
- Protocol Buffers
- Сравнение служб gRPC с API-интерфейсами HTTP

Задание:

Во время выполнения задания можете ориентироваться на этот репозиторий - module12_02.

1. Создайте у себя в проекте из предыдущего задания module12 ветку module12_02
2. Опишите в proto'файле PdfComposeService, аналогичный по функционалу swagger файлу api.yaml из прошлого задания.
3. Сгенерируйте на основе вновь созданного proto файла go пакет pdfcompose.
4. Выделите pdfcompose пакет в отдельный go модуль (по желанию можно выделить его в отдельный git репозиторий или оставить в репозитории module12).
5. Переведите взаимодействие сервисов из прошлого задания (12_01) с REST на gRPC. Передавать файлы можно как в байтах, так и в base64.
6. В ответе пришлите ссылку на merge request в ветку master своего проекта ветки module12_02. А также ссылку на репозиторий с пакетом pdfcompose, если вы его выделили в отдельный репозиторий.