

ЛЕКЦИЯ 6. PROTOBUF И GRPC

OZON

МОСКВА, 2021

ЛЕКЦИИ














1. Введение. Рабочее окружение. Структура программы. Инструментарий.
2. Базовые конструкции и операторы. Встроенные типы и структуры данных.
3. Структуры данных, отложенные вызовы, обработка ошибок и основы тестирования
4. Интерфейсы, моки и тестирование с ними
5. Асинхронные сущности и паттерны в Go
6. Protobuf и gRPC
7. Работа с БД в Go
8. Брокеры сообщений. Трассировка. Метрики.

ТЕМЫ

Сегодня мы поговорим про:

1. Мандат Безоса
2. Зачем нужны Protobuf и gRPC
3. Как построить создать микросервис на Go
4. Как подключить REST API
5. Как обратиться к сервису
6. Makefile и Docker для окружения (но можем не успеть)

ОБОЗНАЧЕНИЯ

-  - посмотри воркшоу
-  - проведи эксперимент
-  - изучи внимательно
-  - прочитай документация
-  - подумай о сложности
-  - запомни ошибку
-  - запомни решение
-  - обойди камень предковенья
-  - сделай перерыв
-  - попробуй дома
-  - обсуди светлые идеи
-  - задай вопрос
-  - запомни панику

МАНДАТ БЕЗОСА

Оригинал: <http://jesusgilhernandez.com/2012/10/18/jeff-bezos-mandate-amazon-and-web-services/>

1. Отныне все команды будут предоставлять свои данные и функциональные возможности через сервисные интерфейсы.
2. Команды должны взаимодействовать друг с другом через эти интерфейсы.
3. Не будет разрешена никакая другая форма межпроцессной связи: никаких прямых ссылок, никаких прямых считываний хранилища данных другой команды, никакой модели общей памяти, никаких “задних дверей” вообще. Единственная разрешенная связь-это вызовы интерфейса службы по сети.

МАНДАТ БЕЗОСА

4. Не имеет значения, какую технологию они используют. HTTP, Corba, Pubsub, пользовательские протоколы — не имеет значения.
5. Все интерфейсы обслуживания, без исключения, должны быть спроектированы с нуля, чтобы быть внешними. То есть команда должна планировать и проектировать, чтобы иметь возможность предоставлять интерфейс разработчикам во внешнем мире. Никаких исключений.
6. Любой, кто этого не сделает, будет уволен.
7. Спасибо, хорошего вам дня!

PROTOBUF

Protocol Buffers — протокол сериализации (передачи) структурированных данных, предложенный Google как эффективная бинарная альтернатива текстовому формату XML.

Разработчики сообщают, что Protocol Buffers проще, компактнее и быстрее, чем XML, поскольку осуществляется передача бинарных данных, оптимизированных под минимальный размер сообщения

🙋 Какие ещё методы сериализации вы знаете?

GRPC


gRPC (Remote Procedure Calls) — это система удалённого вызова процедур (RPC) с открытым исходным кодом, первоначально разработанная в Google в 2015 году. В качестве транспорта используется HTTP/2, в качестве языка описания интерфейса — буферы протоколов

gRPC предоставляет такие функции как аутентификация, двунаправленная потоковая передача и управление потоком, блокирующие или неблокирующие привязки, а также отмена и тайм-ауты.

GRPC

Генерирует кроссплатформенные привязки клиента и сервера для многих языков. Чаще всего используется для подключения служб в микросервисном стиле архитектуры и подключения мобильных устройств и браузерных клиентов к серверным службам.

Сложное использование HTTP/2 в gRPC делает невозможным реализацию клиента gRPC в браузере - вместо этого требуется прокси.

 Больше про разницу в версиях HTTP: <https://cheapsslsecurity.com/p/http2-vs-http1/>

PROTO ФАЙЛ

```
syntax = "proto3";  
option go_package = "github.com/ozoncp/osp-task-api/pkg/osp-task-api;osp_task_api";  
package osp.task.api;  
  
// Описание задачи  
message Task {  
    uint64 task_id = 1;  
    string description = 2;  
  
    string comment = 20;  
}
```

syntax определяет версию proto-спецификации:

- 3
- 2 (default)

package - наименование пакета, пространство имен

// comment - используются в документации

PROTO ФАЙЛ

; - часто встречаемая ошибка 🐛 - забыл поставить ;

`option` - для разных случаев, можно и свои (полный список [тут](#))

```
string description = 2 [deprecated = true];  
option optimize_for = SPEED, CODE_SIZE, LITE_RUNTIME
```

Расположение:

`./api` – директория для proto файлов, описывающих API сервисов

`./vendor.protogen` – директория для вендоринга proto файлов, включая proto файлы из директории `api`.

SERVICE

```
syntax = "proto3";

import "google/api/annotations.proto";
import "google/protobuf/empty.proto";

package siriusfreak.lecture_6_demo;

option go_package = "gitlab.com/siriusfreak/lecture-6-demo/pkg/lecture-6-demo;lecture_6_demo";

// Lecture6Demo сервис для обработки текстов
service Lecture6Demo {
    // AddV1 добавляет задание в очередь на обработку
    rpc AddV1(AddRequestV1) returns (google.protobuf.Empty) {
    }
}

message AddRequestV1 {
    int64 id = 1;
    string text = 2;
    bool result = 3;
    string callback_url = 4;
}
```

SERVICE

Что надо обсудить заранее:

- Наименования сервисов
- Наименование RPC ручек - verb + subject
- Множественное определение сервисов 🏔
- Потокосное передачу данных: когда и зачем ?
- Общие типы запросов и ответов 🏔
- Порядок расположения сообщений и сервисов 💡
- Свой пустой тип или `google.protobuf.Empty`?

SERVICE

И помните, что часы планирования лишают нас недель разработки не того, чего мы хотели / не того, чего надо.

ГЕНЕРАЦИЯ

Установка компилятора protoc:

```
→ brew install protobuf # или apt-get или просто отсюда: https://github.com/protocolbuffers/protobuf/r
→ which protoc
→ protoc --version
```

Установка зависимостей (часть Makefile):

```
LOCAL_BIN:=$(CURDIR)/bin

.PHONY: deps
deps: install-go-deps

.PHONY: .install-go-deps
.install-go-deps:
    ls go.mod || go mod init gitlab.com/siriusfreak/lecture-6-demo
    GOBIN=$(LOCAL_BIN) go get -u github.com/grpc-ecosystem/grpc-gateway/protoc-gen-grpc-gateway
    GOBIN=$(LOCAL_BIN) go get -u github.com/golang/protobuf/proto
    GOBIN=$(LOCAL_BIN) go get -u github.com/golang/protobuf/protoc-gen-go
    GOBIN=$(LOCAL_BIN) go get -u google.golang.org/grpc
    GOBIN=$(LOCAL_BIN) go get -u google.golang.org/grpc/cmd/protoc-gen-go-grpc
    GOBIN=$(LOCAL_BIN) go install google.golang.org/grpc/cmd/protoc-gen-go-grpc
    GOBIN=$(LOCAL_BIN) go install github.com/grpc-ecosystem/grpc-gateway/protoc-gen-swagger
```

```
make deps
```

MAKEFILE

В общем случае: файл, который описывает как собрать проект на сях, со своими плюшками.

В нашем случае: удобная форма записи консольных команд.

ГЕНЕРАЦИЯ

Генерация кода из proto-файлов

```
→ GOBIN=$(LOCAL_BIN) protoc -I vendor.protogen \
  --go_out=pkg/lecture-6-demo --go_opt=paths=import \
  --go-grpc_out=pkg/lecture-6-demo --go-grpc_opt=paths=import \
  api/lecture-6-demo/lecture-6-demo.proto
→ ls -l pkg/lecture-6-demo/gitlab.com/siriusfreak/lecture-6-demo/pkg/lecture-6-demo
```

```
lecture-6-demo.pb.go    # types
lecture-6-demo_grpc.pb.go # grpc
```

ИМПЛЕМЕНТАЦИЯ ОБРАБОТЧИКА

internal/app/lecture-6-demo/service.go

```
package lecture_6_demo

import desc "gitlab.com/siriusfreak/lecture-6-demo/pkg/lecture-6-demo"

type Lecture6DemoAPI struct {
    desc.UnimplementedLecture6DemoServer
}

func NewLecture6DemoAPI() desc.Lecture6DemoServer {
    return &Lecture6DemoAPI{}
}
```

ИМПЛЕМЕНТАЦИЯ ОБРАБОТЧИКА

internal/app/lecture-6-demo/add_v1.go

```
package lecture_6_demo

import (
    "context"

    desc "gitlab.com/siriusfreak/lecture-6-demo/pkg/lecture-6-demo"
    "google.golang.org/protobuf/types/known/emptypb"
)

func (a *Lecture6DemoAPI) AddV1(ctx context.Context, req *desc.AddRequestV1) (*emptypb.Empty, error) {
    return &emptypb.Empty{}, nil
}
```

ЗАПУСК СЕРВИСА

```
const (  
    grpcPort = ":82"  
    grpcServerEndpoint = "localhost:82"  
)  
  
func run() error {  
    listen, err := net.Listen("tcp", grpcPort)  
    if err != nil {  
        log.Fatalf("failed to listen: %v", err)  
    }  
  
    s := grpc.NewServer()  
    desc.RegisterLecture6DemoServer(s, api.NewLecture6DemoAPI())  
  
    if err := s.Serve(listen); err != nil {  
        log.Fatalf("failed to serve: %v", err)  
    }  
  
    return nil  
}
```

ЗАПУСК СЕРВИСА

```
package main

import (
    "log"
    "net"

    api "gitlab.com/siriusfreak/lecture-6-demo/internal/app/lecture-6-demo"
    "google.golang.org/grpc"

    desc "gitlab.com/siriusfreak/lecture-6-demo/pkg/lecture-6-demo"
)

func main() {
    if err := run(); err != nil {
        log.Fatal(err)
    }
}
```

ПРОКСИРОВАНИЕ

Добавляем определения для шлюза.

```
syntax = "proto3";

import "google/api/annotations.proto";
import "google/protobuf/empty.proto";

package siriusfreak.lecture_6_demo;

option go_package = "gitlab.com/siriusfreak/lecture-6-demo/pkg/lecture-6-demo;lecture_6_demo";

// Lecture6Demo сервис для обработки текстов
service Lecture6Demo {
    // AddV1 добавляет задание в очередь на обработку
    rpc AddV1(AddRequestV1) returns (google.protobuf.Empty) {
        option (google.api.http) = {
            post: "/v1/add"
        };
    }
}

message AddRequestV1 {
    int64 id = 1;
    string text = 2;
    bool result = 3;
    string callback_url = 4;
}
```

ПРОКСИРОВАНИЕ

Добавляем генерацию шлюза и swagger.

```
protoc -I vendor.protogen \  
  --go_out=pkg/lecture-6-demo --go_opt=paths=import \  
  --go-grpc_out=pkg/lecture-6-demo --go-grpc_opt=paths=import \  
  --grpc-gateway_out=pkg/lecture-6-demo \  
  --grpc-gateway_opt=logtostderr=true \  
  --grpc-gateway_opt=paths=import \  
  --swagger_out=allow merge=true,merge_file_name=api:swagger \  
  api/lecture-6-demo/Lecture-6-demo.proto
```

ПРОКСИРОВАНИЕ

Запускаем обработчик REST API:

```
func runJSON() {
    ctx := context.Background()
    ctx, cancel := context.WithCancel(ctx)
    defer cancel()

    mux := runtime.NewServeMux()
    opts := []grpc.DialOption{grpc.WithInsecure()}

    err := desc.RegisterLecture6DemoHandlerFromEndpoint(ctx, mux, grpcServerEndpoint, opts)
    if err != nil {
        panic(err)
    }

    err = http.ListenAndServe(":8081", mux)
    if err != nil {
        panic(err)
    }
}

func main() {
    go runJSON()

    if err := run(); err != nil {
        log.Fatal(err)
    }
}
```


КАК ДЁРГАТЬ РУЧКИ?

gRPC: <https://github.com/uw-labs/bloomrpc>

Gateway: <https://www.postman.com/>

ВЕРСИОНИРОВАНИЕ

Before	After
DescribeTask	DescribeTaskV1
DescribeTaskRequest	DescribeTaskV1Request
DescribeTaskResponse	DescribeTaskV1Response
“/tasks/{task_id}”	“/v1/tasks/{task_id}”

#ОБЫЧНЫЕ-СООБЩЕНИЯ #СОВМЕСТИМОЕ-ОБНОВЛЕНИЕ

ВЕРСИОНИРОВАНИЕ

Добавление новой версии ручки

- Задеплоить релиз с новой версией ручки
- Перевести всех нужных клиентов на новую версию

Переход на новую версию ручки

- Пометить, как deprecated
- Задеплоить релиз с новой версией ручки
- Перевести всех клиентов на новую версию
- Задеплоить релиз без старой версии ручки, если не планируется поддерживать

ДЕПЛОЙ ПАКЕТА

```
→ cd pkg/lecture-6-demo/  
→ go mod init && go mod tidy  
→ git tag pkg/lecture-6-demo/v0.1.0 # v1.0.0  
→ git push origin --tags  
  
→ z lecture-6-demo  
→ go get gitlab.com/siriusfreak/lecture-6-demo/pkg/lecture-6-demo# v1.0.0
```

ДЕПЛОЙ ПАКЕТА

```
import (
    taskApi "github.com/ozoncp/ocp-task-api/pkg/ocp-task-api"
    "google.golang.org/grpc"
)
conn, err := grpc.Dial(taskApiAddress, grpc.WithInsecure())
client := taskApi.NewOcpTaskApiClient(conn)
req := &taskApi.DescribeTaskV1Request{
    TaskId: taskId,
}
resp, err := client.DescribeTaskV1(ctx, req)
```

ДЕПЛОЙ ПАКЕТА

1. Использование `replace` для отладки
2. Использование `go get` для обновления
3. Версионирование `semver`
4. Генерация пакетов на других языках

ВАЛИДАЦИЯ

```
// ...  
import "github.com/envoyproxy/protoc-gen-validate/validate/validate.proto";  
// ...  
  
message AddRequestV1 {  
  int64 id = 1 [(validate.rules).uint64.gt = 0];  
  string text = 2;  
  bool result = 3;  
  string callback_url = 4;  
}
```

ВАЛИДАЦИЯ

```
protoc -I vendor.protogen \
  --go_out=pkg/lecture-6-demo --go_opt=paths=import \
  --go-grpc_out=pkg/lecture-6-demo --go-grpc_opt=paths=import \
  --grpc-gateway_out=pkg/lecture-6-demo \
  --grpc-gateway_opt=logtostderr=true \
  --grpc-gateway_opt=paths=import \
  --swagger_out=allow_merge=true,merge_file_name=api:swagger \
  --validate_out lang=go:pkg/lecture-6-demo\ # +
  api/lecture-6-demo/lecture-6-demo.proto
```


ВАЛИДАЦИЯ

```
→ ls -l pkg/lecture-6-demo  
lecture-6-demo.pb.go  
lecture-6-demo.pb.gw.go  
lecture-6-demo.pb.validate.go # new file  
lecture-6-demo_grpc.pb.go
```

ВАЛИДАЦИЯ

- Инсталлирование плагина
- Генерация кода
- Добавление проверки в обработчики
- Добавление проверки в вызывающий код

```
import (  
    "google.golang.org/grpc/codes"  
    "google.golang.org/grpc/status"  
)  
  
if err := req.Validate(); err != nil {  
    return nil, status.Error(codes.InvalidArgument, err.Error())  
}
```

валидация в вызывающий код

ПОЛЬЗОВАТЕЛЬСКИЕ ТИПЫ

```
message Author {  
    uint64 user_id = 1;  
    string name = 2;  
    string display_name = 3;  
}
```









```
message Task {  
    uint64 id = 1;  
    string description = 2;  
    Author author = 3;  
}
```

ПОЛЬЗОВАТЕЛЬСКИЕ ТИПЫ

Вопросы:

1. Вложенные сообщения 🏔️
2. Все поля по умолчанию optional ?
3. Наименование полей - snake_case
4. Идентификаторы только на расширение
5. Поля можно переименовывать ?
6. Важен ли порядок размещения полей ?

ВСТРОЕННЫЕ ТИПЫ

Protobuf	C++	Go
double	double	float64
float	float	float32
int32 	int32	int32
int64 	int64	int64
uint32 	uint32	uint32
uint64 	uint64	uint64
sint32 	int32	int32
sint64 	int64	int64
bool  ?	bool	bool
string 	string	string
bytes	string	[]byte

ВСТРОЕННЫЕ ТИПЫ

отображения map и повторяемые repeated

The value_type can be any type except another map

```
message MutliDescribeTaskRequest {  
  repeated uint64 ids = 1;  
}  
  
message MutliDescribeTaskResponse {  
  map<uint64, Task> tasks = 1;  
}
```

Для совместимости массив пар: ключ - значение

ИЗВЕСТНЫЕ ТИПЫ

wrappers.proto	Field	Type	Description
BoolValue	value	bool	The bool value
BytesValue	value	bytes	The bytes value
DoubleValue	value	double	The double value
FloatValue	value	float	The float value
Int32Value	value	int32	The int32
Int64Value	values	int64	The int64 value
StringValue	values	string	The string value

ИЗВЕСТНЫЕ ТИПЫ

timestamp.proto	Field	Type	Description
Timestamp	seconds	int64	Represents seconds of UTC time since Unix epoch 1970-01-01T00:00:00Z. Must be from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59Z inclusive
	nanos	int32	Non-negative fractions of a second at nanosecond resolution. Negative second values with fractions must still have non-negative nanos values that count forward in time. Must be from 0 to 999,999,999 inclusive

ИЗВЕСТНЫЕ ТИПЫ

duration.proto	Field	Type	Description
Duration	seconds	int64	Signed seconds of the span of time. Must be from -315,576,000,000 to +315,576,000,000 inclusive
	nanos	int32	Signed fractions of a second at nanosecond resolution of the span of time. Durations less than one second are represented with a 0 seconds field and a positive or negative nanos field. For durations of one second or more, a non-zero value for the nanos field must be of the same sign as the seconds field. Must be from -999,999,999 to +999,999,999 inclusive

ИЗВЕСТНЫЕ ТИПЫ

В `empty.proto` определен `Empty`.

```
import "google/protobuf/empty.proto";

service OcpTaskApi {
  // Удаляет задачу по ее идентификатору
  rpc RemoveTaskV1(RemoveTaskV1Request) returns (google.protobuf.Empty) {
    option (google.api.http) = {
      delete: "/tasks/{task_id}"
    };
  }
}
```

REPEATED

```
message Task {  
    uint64 task_id = 1;  
    string description = 2;  
    repeated string tags = 3;  
    repeated Author authors = 4;  
}
```

ПЕРЕЧИСЛЕНИЯ

```
enum TaskDifficulty {  
    Begin = 0;  
    Easy = 1;  
    Normal = 2;  
    Hard = 3;  
}  
  
// Описание задачи  
message Task {  
    uint64 task_id = 1;  
    string description = 2;  
    TaskDifficulty difficulty = 3; // +  
}
```

ВОПРОСЫ

- Минимальное значение для первого элемента ?
- Максимальное значение для первого элемента ?
- Unknown со значение 0 💡
- Разрывы в нумерации 💡
- Наименование Type + Value
- Что может быть ключом для отображения ? 🧐
- Когда что лучше использовать ? 🧐

GRPC КОДЫ

```
resp, err := a.client.DescribeTaskV1(ctx, req)
if err != nil {
    status, ok := status.FromError(err)
    if !ok {
        return nil, fmt.Errorf("get status from to error: %w", err)
    }
    if status.Code() == codes.NotFound {
        return nil, ErrTaskNotFound
    }
    return nil, fmt.Errorf("describe task: %w", err)
}
```

GRPC КОДЫ

- OK
- Canceled
- Unknown
- InvalidArgument
- DeadlineExceeded
- NotFound
- AlreadyExists
- PermissionDenied
- ResourceExhausted
- FailedPrecondition
- Aborted
- OutOfRange
- Unimplemented
- Internal
- Unavailable
- DataLoss
- Unauthenticated

MAKEFILE

Фальшивая (PHONY) цель-это цель, которая на самом деле не является именем файла; скорее, это просто имя рецепта, который будет выполняться при выполнении явного запроса.

Есть две причины использовать фальшивую цель: избежать конфликта с файлом с тем же именем и повысить производительность.

MAKEFILE

Сборка, тесты, линтер:

```
LOCAL_BIN:=$(CURDIR)/bin
.PHONY: run
run:
    GOBIN=$(LOCAL_BIN) go run cmd/lecture-6-demo/main.go
.PHONY: lint
lint:
    GOBIN=$(LOCAL_BIN) golint ./...
.PHONY: test
test:
    GOBIN=$(LOCAL_BIN) go test -v ./...
.PHONY: .build
.build:
    GOBIN=$(LOCAL_BIN) go build -o $(LOCAL_BIN)/lecture-6-demo cmd/lecture-6-demo/main.go
```

MAKEFILE

Генерация кода:

```
.PHONY: build
build: vendor-proto .generate .build

.PHONY: .generate
.generate:
    mkdir -p swagger
    mkdir -p pkg/lecture-6-demo
    GOBIN=$(LOCAL_BIN) protoc -I vendor.protogen \
        --go_out=pkg/lecture-6-demo --go_opt=paths=import \
        --go-grpc_out=pkg/lecture-6-demo --go-grpc_opt=paths=import \
        --grpc-gateway_out=pkg/lecture-6-demo \
        --grpc-gateway_opt=logtostderr=true \
        --grpc-gateway_opt=paths=import \
        --swagger_out=allow_merge=true,merge_file_name=api:swagger \
        api/lecture-6-demo/lecture-6-demo.proto
    mv pkg/lecture-6-demo/gitlab.com/siriusfreak/lecture-6-demo/pkg/lecture-6-demo/* pkg/lecture-6-demo/
    rm -rf pkg/lecture-6-demo/gitlab.com
    mkdir -p cmd/lecture-6-demo
    cd pkg/lecture-6-demo && ls go.mod || go mod init gitlab.com/siriusfreak/lecture-6-demo/pkg/lecture-6-demo

.PHONY: generate
generate: .vendor-proto .generate
```

MAKEFILE

Скачивание протофайлов:

```
.PHONY: vendor-proto
vendor-proto: .vendor-proto

.PHONY: .vendor-proto
.vendor-proto:
    mkdir -p vendor.protogen
    mkdir -p vendor.protogen/api/lecture-6-demo
    cp api/lecture-6-demo/lecture-6-demo.proto vendor.protogen/api/lecture-6-demo/lecture-6-demo.p
    @if [ ! -d vendor.protogen/google ]; then \
        git clone https://github.com/googleapis/googleapis vendor.protogen/googleapis &&\
        mkdir -p vendor.protogen/google/ &&\
        mv vendor.protogen/googleapis/google/api vendor.protogen/google &&\
        rm -rf vendor.protogen/googleapis ;\
    fi
```

 Добавить скачивание для: github.com/envoyproxy/protoc-gen-validate/validate/validate.proto

MAKEFILE

Установка зависимостей:

```
.PHONY: deps
deps: install-go-deps

.PHONY: install-go-deps
install-go-deps: .install-go-deps

.PHONY: .install-go-deps
.install-go-deps:
    ls go.mod || go mod init gitlab.com/siriusfreak/lecture-6-demo
    GOBIN=$(LOCAL_BIN) go get -u github.com/grpc-ecosystem/grpc-gateway/protoc-gen-grpc-gateway
    GOBIN=$(LOCAL_BIN) go get -u github.com/golang/protobuf/proto
    GOBIN=$(LOCAL_BIN) go get -u github.com/golang/protobuf/protoc-gen-go
    GOBIN=$(LOCAL_BIN) go get -u google.golang.org/grpc
    GOBIN=$(LOCAL_BIN) go get -u google.golang.org/grpc/cmd/protoc-gen-go-grpc
    GOBIN=$(LOCAL_BIN) go install google.golang.org/grpc/cmd/protoc-gen-go-grpc
    GOBIN=$(LOCAL_BIN) go install github.com/grpc-ecosystem/grpc-gateway/protoc-gen-swagger
```

DOCKER

Чтобы ваш код работал не только дома.

Установка:

- Mac и Windows: <https://www.docker.com/products/docker-desktop>
- Linux: <https://docs.docker.com/engine/install/ubuntu/>

Docker-compose для БД `docker-compose.yml`:

```
# Use postgres/example user/password credentials
version: '3.1'

services:
  db:
    image: postgres
    restart: always
    environment:
      POSTGRES_PASSWORD: example
  adminer:
    image: adminer
    restart: always
    ports:
      - 8080:8080
```

`docker compose up`