OZON tech

Облегчаем жизнь разработчикам при помощи плагинов Protoc

Святослав Петров
Старший разработчик

svpetrov@ozon.ru

Святослав Петров

- Старший разработчик @ Go Платформа
- Фреймворк, библиотеки, инструменты разработчика

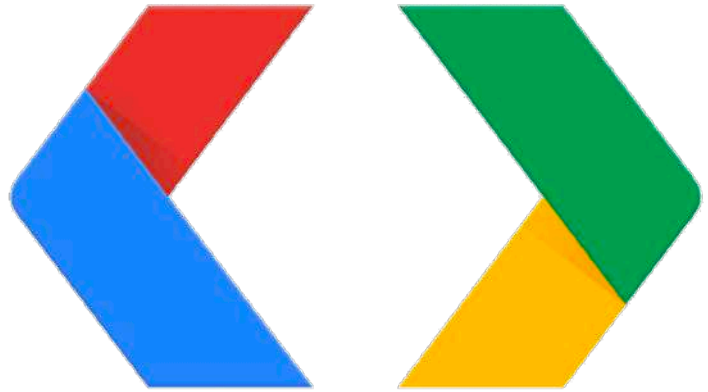
Для кого этот доклад

01 Вы хотите углубиться в то, как работает тулчейн Protobuf, и что с ним можно сделать крутого

02 Вы разрабатываете инструменты для других разработчиков, которые используют Protobuf

Protobuf

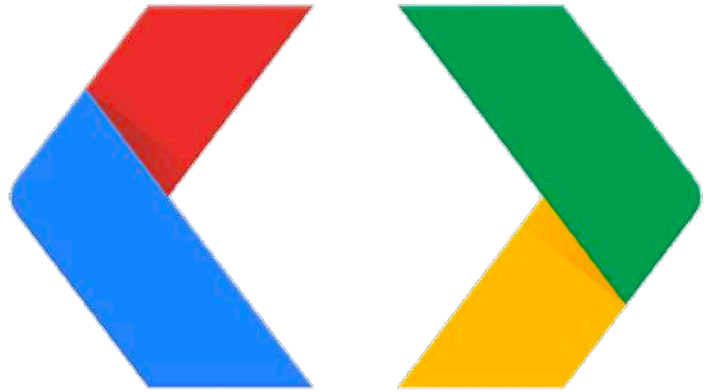
Это механизм сериализации данных



protobuf.dev

Protobuf

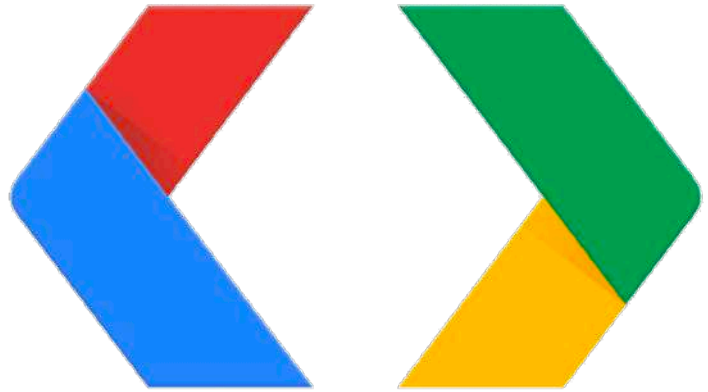
- **Это механизм сериализации данных**
- **Он не зависит от конкретного языка**



protobuf.dev

Protobuf

- Это механизм сериализации данных
- Он не зависит от конкретного языка
- Его приоритет — производительность (скорость парсинга, размер данных)



protobuf.dev

Как выглядит Protobuf?

```
message CodeGeneratorResponse {
    optional string error = 1;

    optional uint64 supported_features = 2;
    enum Feature {
      FEATURE_NONE = 0;
      FEATURE_PROTO3_OPTIONAL = 1;
      FEATURE_SUPPORTS_EDITIONS = 2;
    }
    optional int32 minimum_edition = 3;
    optional int32 maximum_edition = 4;

    message File {
      optional string name = 1;
      optional string insertion_point = 2;
      optional string content = 15;
      optional GeneratedCodeInfo generated_code_info = 16;
    }
    repeated File file = 15;
}
```

Умеет описывать сообщения и сервисы с методами

Как выглядит Protobuf?

```
message CodeGeneratorResponse {
    optional string error = 1;

    optional uint64 supported_features = 2;
    enum Feature {
      FEATURE_NONE = 0;
      FEATURE_PROTO3_OPTIONAL = 1;
      FEATURE_SUPPORTS_EDITIONS = 2;
    }
    optional int32 minimum_edition = 3;
    optional int32 maximum_edition = 4;

    message File {
      optional string name = 1;
      optional string insertion_point = 2;
      optional string content = 15;
      optional GeneratedCodeInfo generated_code_info = 16;
    }
    repeated File file = 15;
}
```

- **Умеет описывать сообщения и сервисы с методами**
- **Имеет большинство известных типов из коробки (int, float, string, bool)**

Как выглядит Protobuf?

```
message CodeGeneratorResponse {
    optional string error = 1;

    optional uint64 supported_features = 2;
    enum Feature {
      FEATURE_NONE = 0;
      FEATURE_PROTO3_OPTIONAL = 1;
      FEATURE_SUPPORTS_EDITIONS = 2;
    }
    optional int32 minimum_edition = 3;
    optional int32 maximum_edition = 4;

    message File {
      optional string name = 1;
      optional string insertion_point = 2;
      optional string content = 15;
      optional GeneratedCodeInfo generated_code_info = 16;
    }
    repeated File file = 15;
}
```

- **Умеет описывать сообщения и сервисы с методами**
- **Имеет большинство известных типов из коробки (int, float, string, bool)**
- **Имеет встроенные map, slice, enum**

Как выглядит Protobuf?

```
message CodeGeneratorResponse {
    optional string error = 1;

    optional uint64 supported_features = 2;
    enum Feature {
      FEATURE_NONE = 0;
      FEATURE_PROTO3_OPTIONAL = 1;
      FEATURE_SUPPORTS_EDITIONS = 2;
    }
    optional int32 minimum_edition = 3;
    optional int32 maximum_edition = 4;

    message File {
      optional string name = 1;
      optional string insertion_point = 2;
      optional string content = 15;
      optional GeneratedCodeInfo generated_code_info = 16;
    }
    repeated File file = 15;
}
```

- **Умеет описывать сообщения и сервисы с методами**
- **Имеет большинство известных типов из коробки (int, float, string, bool)**
- **Имеет встроенные map, slice, enum**
- **Может расширяться с помощью дополнительных сообщений (вы можете объявлять и переиспользовать собственные типы)**

Почему Protobuf используют?

Декларативный и полностью типизированный API

01

Language-agnostic

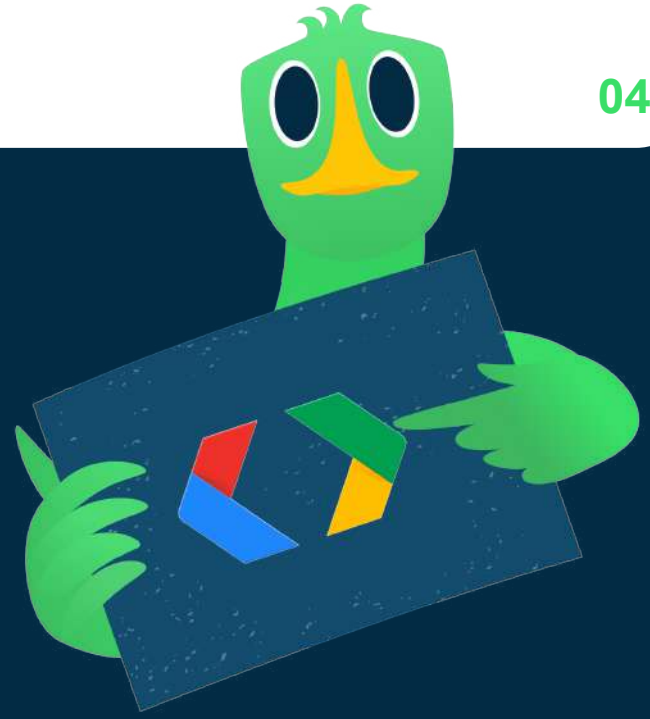
02

Легко расширяемые контракты и совместимость между версиями

03

Экосистема

04



А как Protobuf используют?

Protobuf + gRPC

- **Очень часто они идут бок о бок друг с другом**

Protobuf + gRPC

- **Очень часто они идут бок о бок друг с другом**
- **gRPC требует Protobuf, однако Protobuf можно использовать и без gRPC**

Protobuf + gRPC

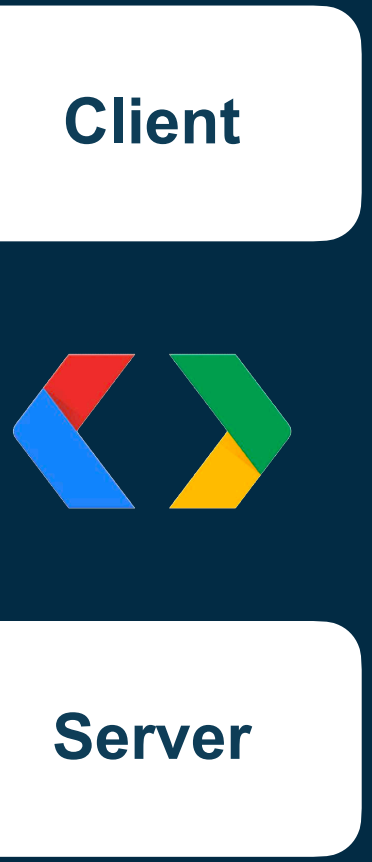
- Очень часто они идут бок о бок друг с другом
- gRPC требует Protobuf, однако Protobuf можно использовать и без gRPC
- Если вы уже используете gRPC, то расширять тулчейн вокруг Protobuf получится с минимальными усилиями

Pure Protobuf

Прямое формирование Protobuf-сообщений

Pure Protobuf

- **Прямое формирование Protobuf-сообщений**
- **Может использовать любой транспортный протокол**

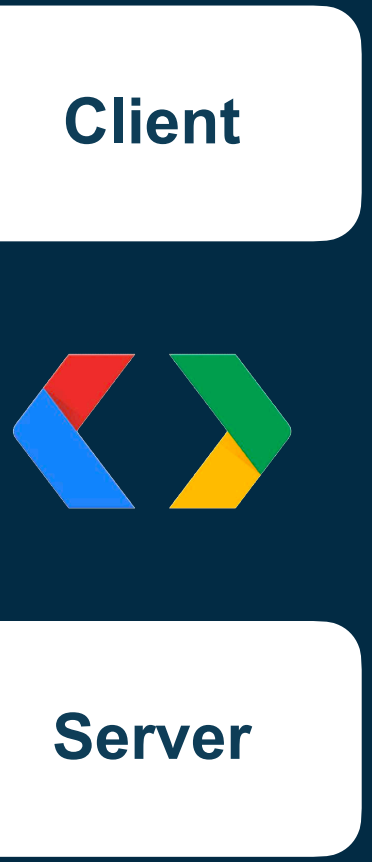


Pure Protobuf

- **Прямое формирование Protobuf-сообщений**
- **Может использовать любой транспортный протокол**
- **Неявный API, принимающий body в формате Protobuf**

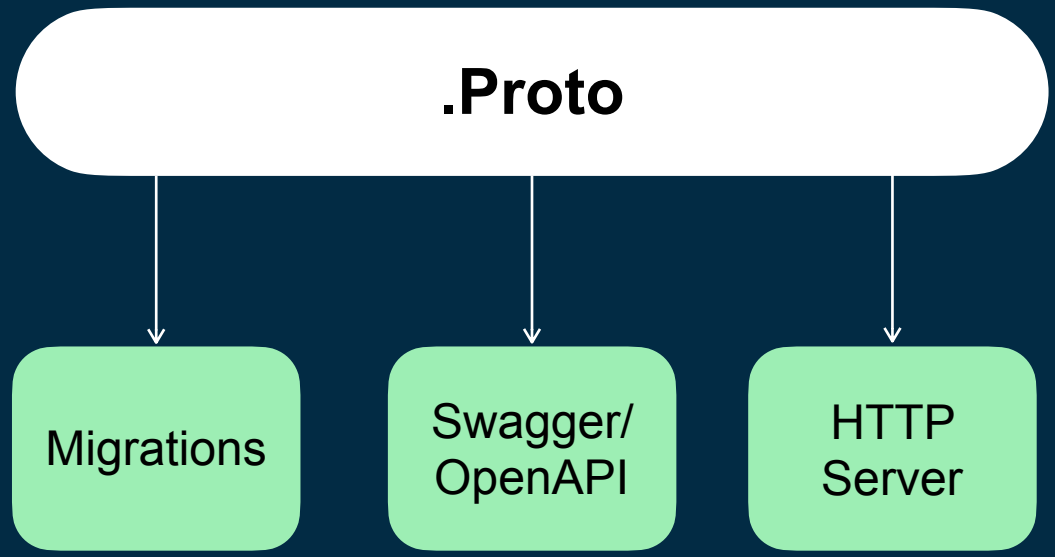
Pure Protobuf

- Прямое формирование Protobuf-сообщений
- Может использовать любой транспортный протокол
- Неявный API, принимающий body в формате Protobuf
- Пример — Thanos



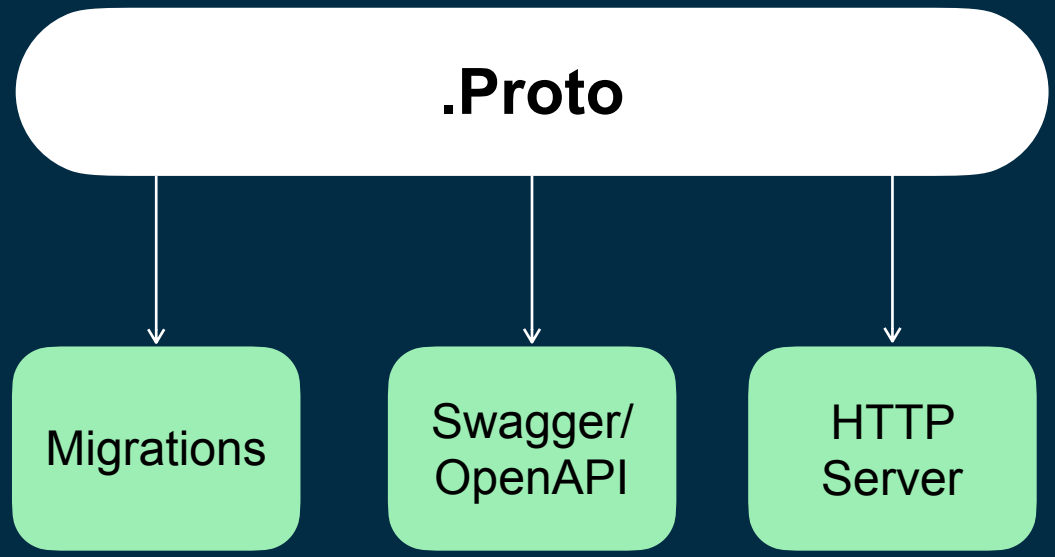
No-protobuf Protobuf

Декларативное использование Protobuf



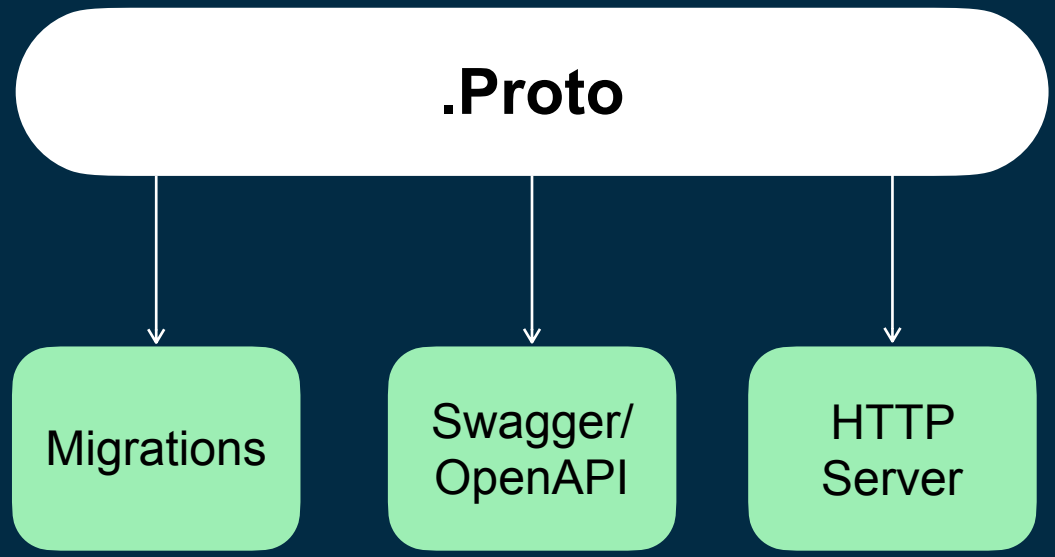
No-protobuf Protobuf

- **Декларативное использование Protobuf**
- **Генерируем вспомогательные артефакты**



No-protobuf Protobuf

- **Декларативное использование Protobuf**
- **Генерируем вспомогательные артефакты**
- **Не используем сам формат данных**



Зачем делать что-то своё?

Как применяется Protobuf в Ozon?

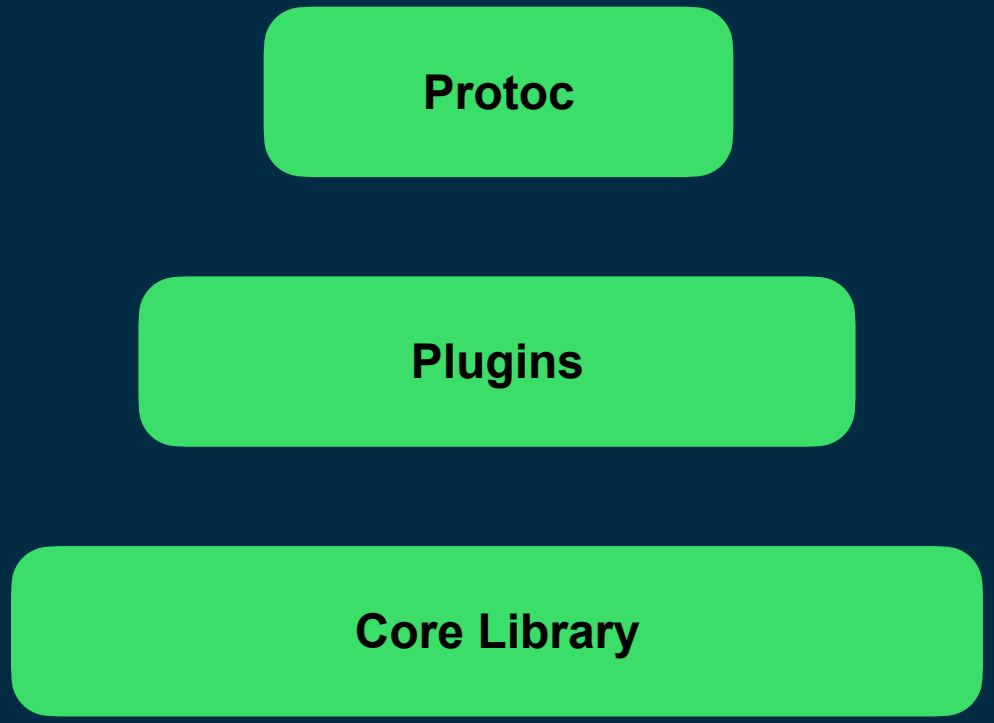
Декораторы для фреймворка вокруг gRPC- и HTTP-серверов 01

Типизированные клиенты для ресурсов. Например, для Kafka 02

Телеметрия — генерируем schema-aware-телеметрию для сервисов 03

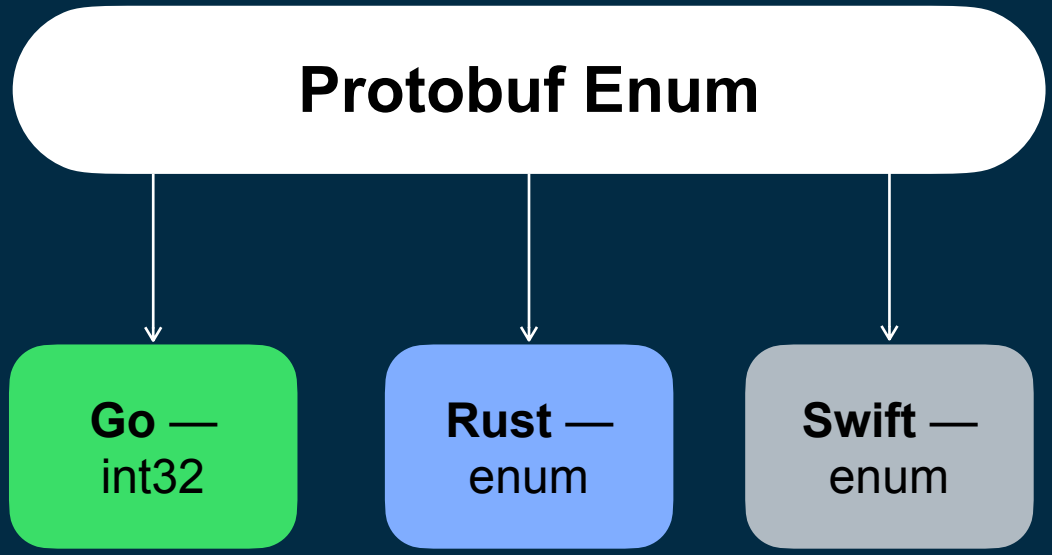
Как сделать что-то своё?

Protobuf-пирамида



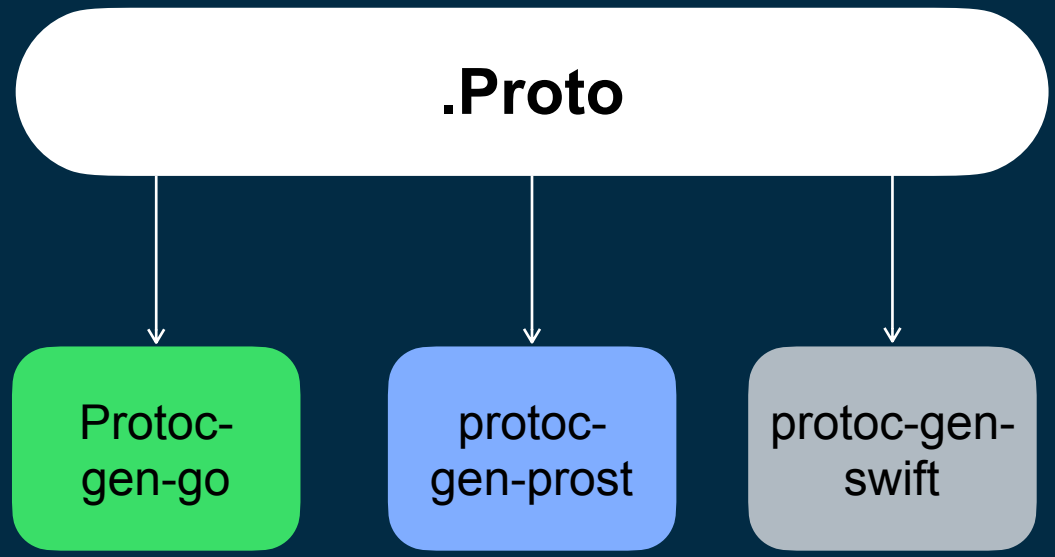
Core Library

- **Базовые блоки для эн(де)кодинга Protobuf**
- **Обертки и хелперы вокруг сырых методов**
- **Маппинг между Protobuf- и Native-типами**



Plugins

- **Обеспечивают генерацию таргет-кода на языке**
- **Могут быть написаны на любом языке, необязательно на том же самом**
- **Генерируют код на основе Core Library**



Высокоуровневая схема

Схема чуть посложнее

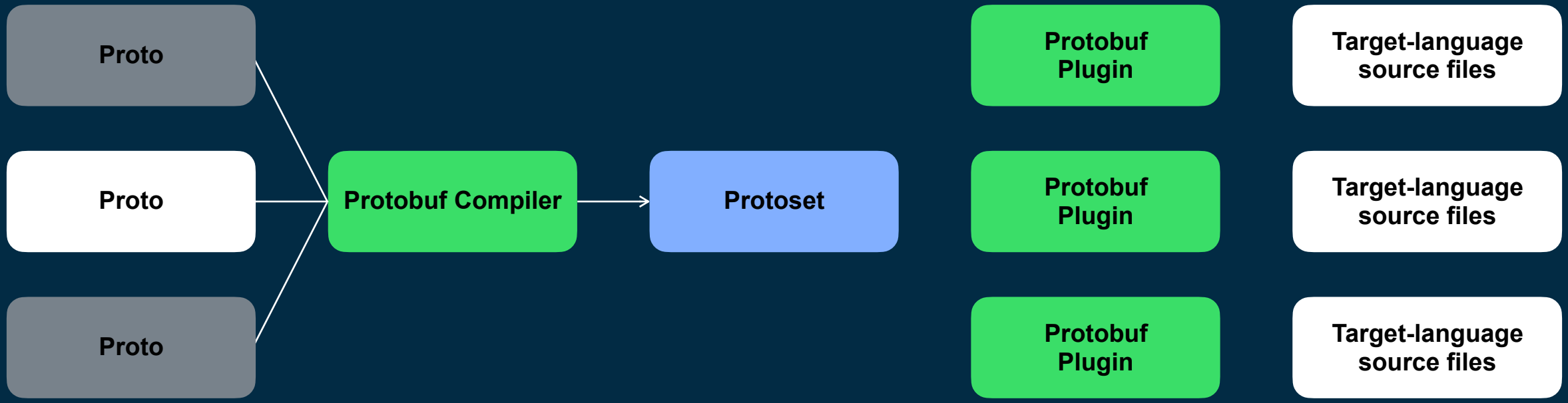
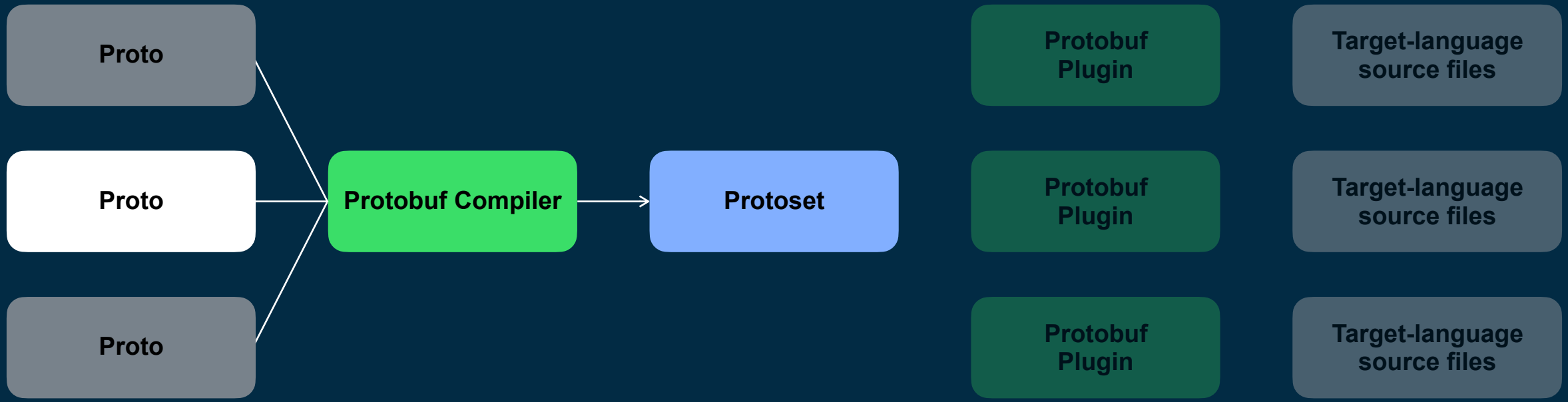
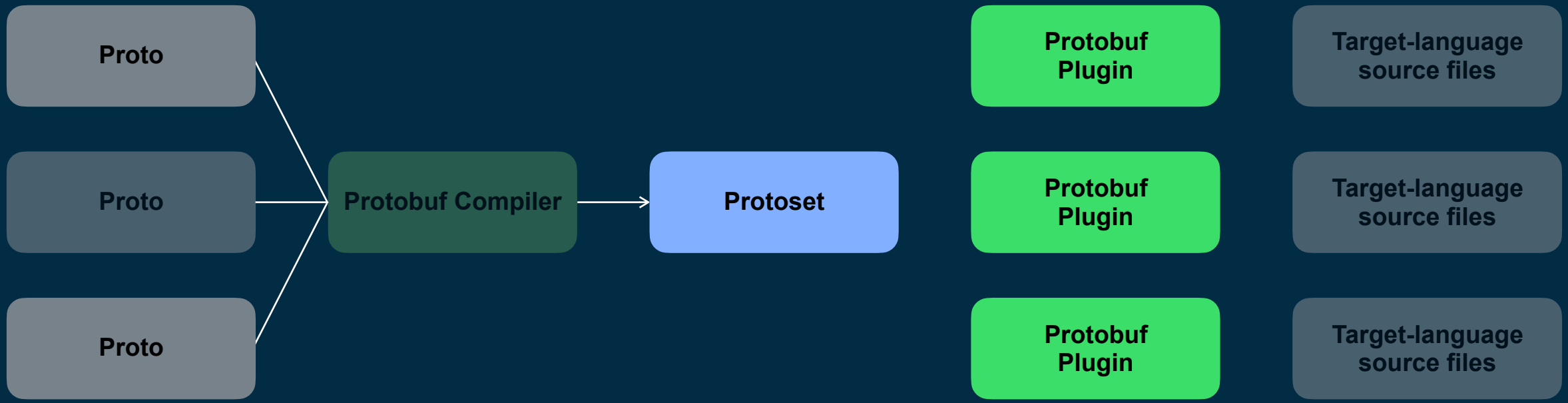


Схема чуть посложнее



- Compiler собирает все зависимости и собирает их в единый ProtoSet
- На этом этапе происходит поиск всех зависимостей, в т.ч. транзитивных
- Такая схема позволяет иметь единую точку логики сбора зависимостей

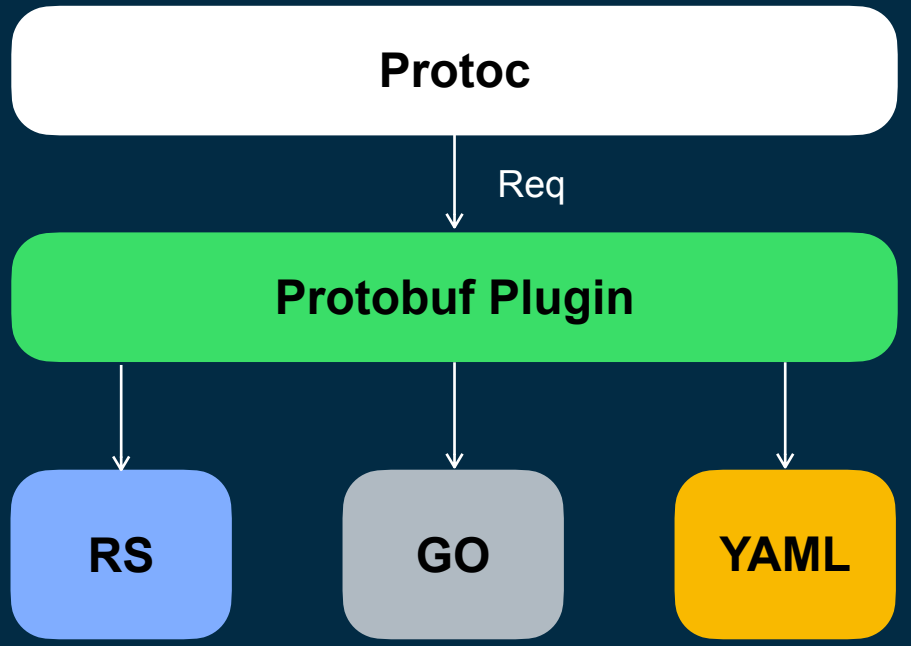
Схема чуть посложнее



- Compiler вызывает каждый из указанных плагинов с полученным ProtoSet
- Плагин обрабатывает поступивший вызов и опционально может производить произвольные действия, включая генерацию файлов
- Каждый плагин исполняется независимо от других!

Что такое Plugin?

- **Отдельная программа/ бинарь**
- **Принимает на вход запрос от Protoc, в котором содержится ProtoSet**
- **Опционально генерирует N файлов и возвращает их в ответе**



Как Plugin вызывается?

- При наличии gofunc_out флага, protoc ищет исполняемый файл в PATH с именем protoc-gen-gofunc и пытается использовать его для генерации

- Альтернативно, можно напрямую указать путь до исполняемого файла с помощью флага —plugin

```
protoc \
    --plugin=protoc-gen-gofunc=/Users/.../protoc-gen-gofunc
    --gofunc_out=. \
    --gofunc_opt=k1=v1 \
    --gofunc_opt=k2=v2 \
    -I proto \
    $(find proto -type f)
```

Plugin Types — STDIN

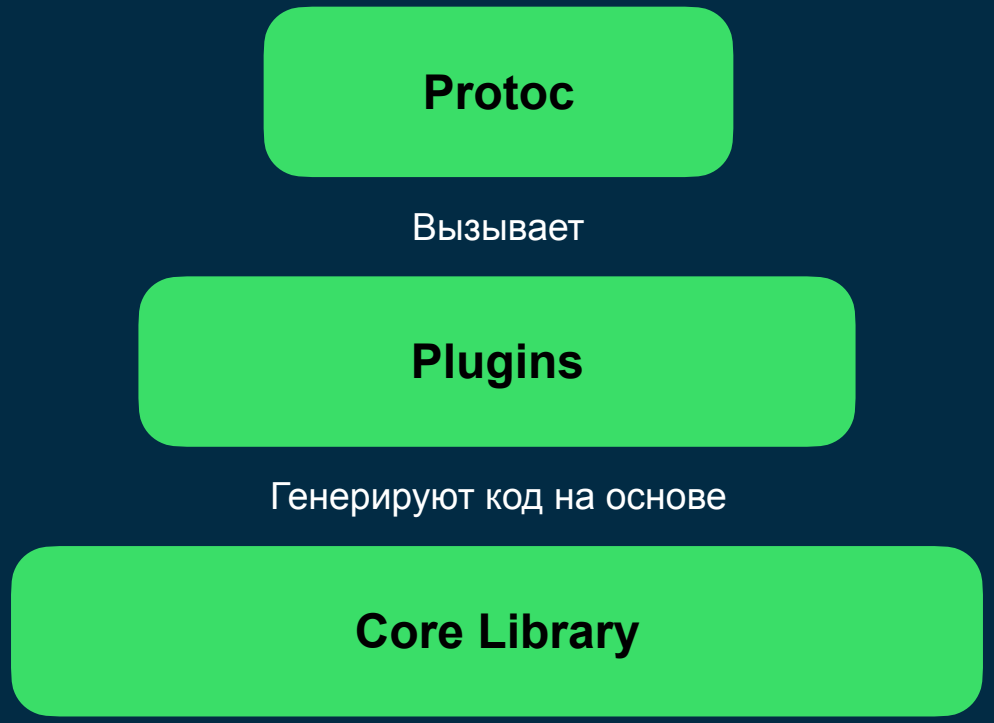
STDIN — CodeGeneratorRequest

https://github.com/protocolbuffers/protobuf/blob/main/src/google/protobuf/compiler/plugin.proto

На входе имеем список файлов, для которых вызвана генерация, их содержимое и метаданные

```
message CodeGeneratorRequest {
    repeated string file_to_generate = 1;

    optional string parameter = 2;

    repeated FileDescriptorProto proto_file = 15;

    repeated FileDescriptorProto source_file_descriptors = 17;

    optional Version compiler_version = 3;
}
```

Protobuf-пирамида

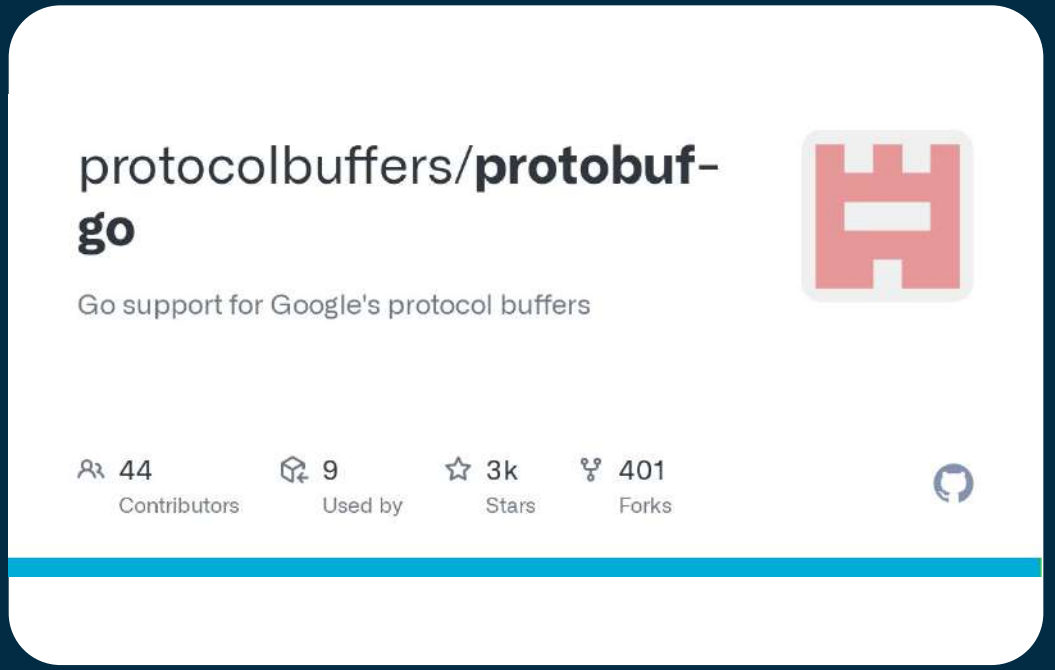


Как написать плагин на Go?

Первый блок — Core Library

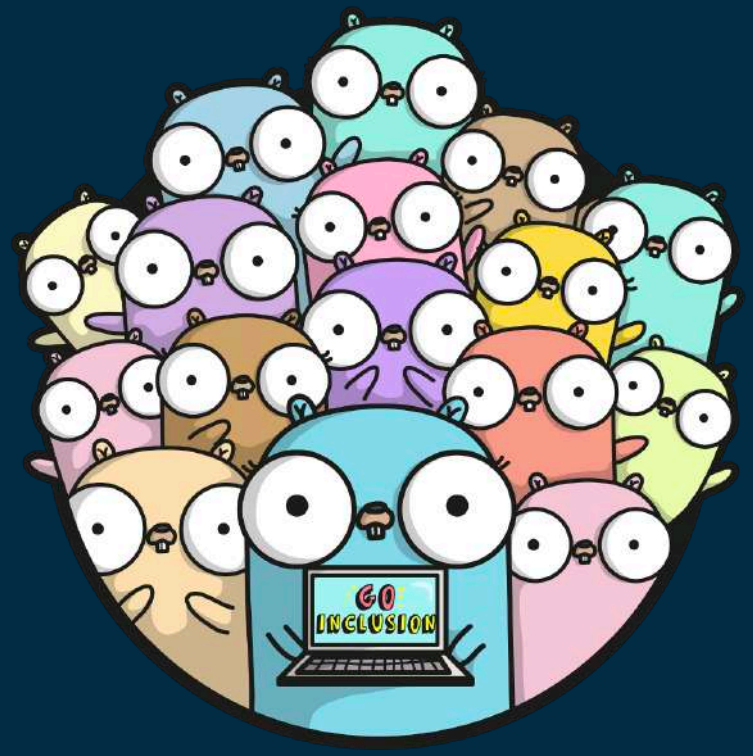
https://google.golang.org/protobuf

Официальный модуль, в котором содержится готовая машинерия – плагины, типы, маршаллинг-функции



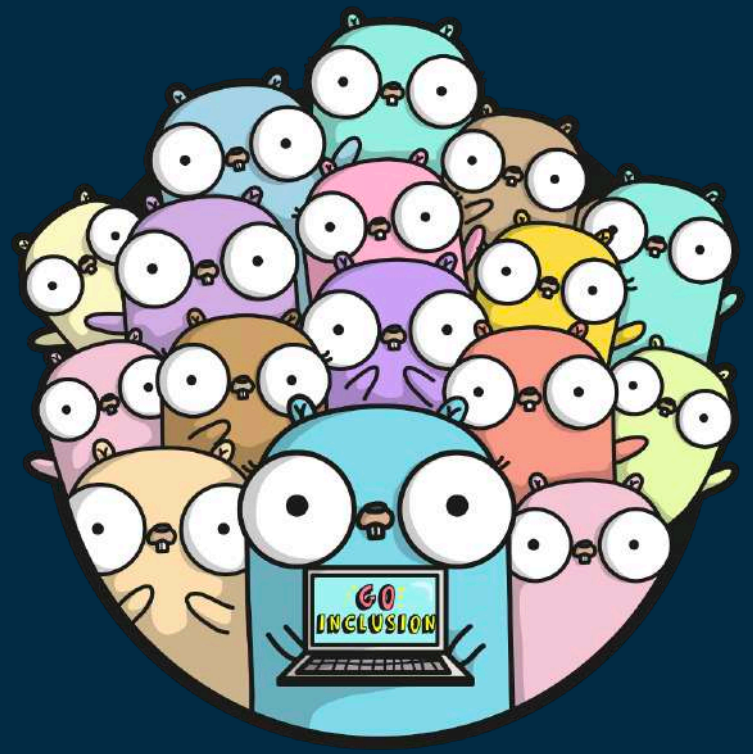
Второй блок — Plugin

Маршаллинг/Анмаршаллинг —
protoc-gen-go



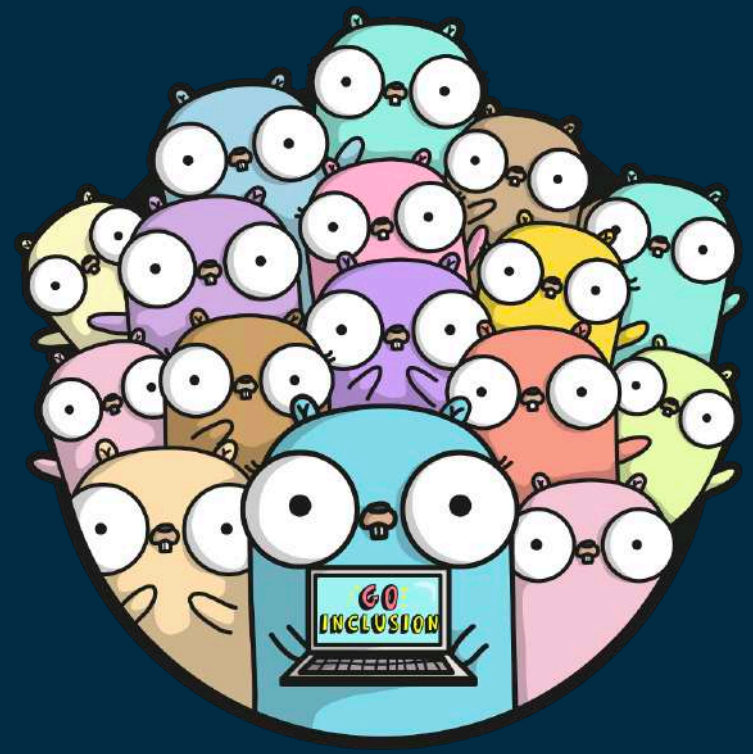
Второй блок — Plugin

- **Маршаллинг/Анмаршаллинг — protoc-gen-go**
- **gRPC — protoc-gen-go-grpc**



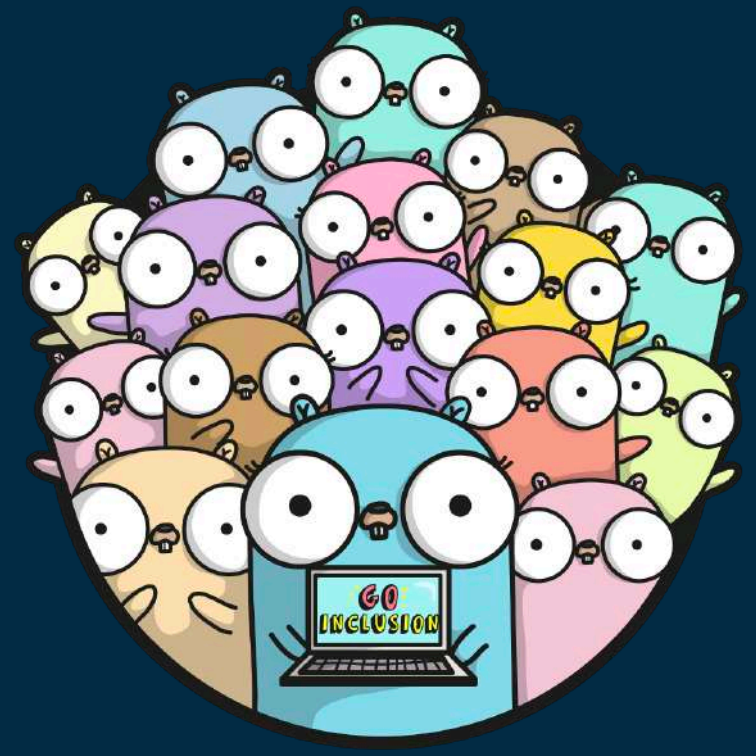
Второй блок — Plugin

- **Маршаллинг/Анмаршаллинг — protoc-gen-go**
- **gRPC — protoc-gen-go-grpc**
- **Для написания плагинов есть все готовое — как компоненты, так и работающие примеры**

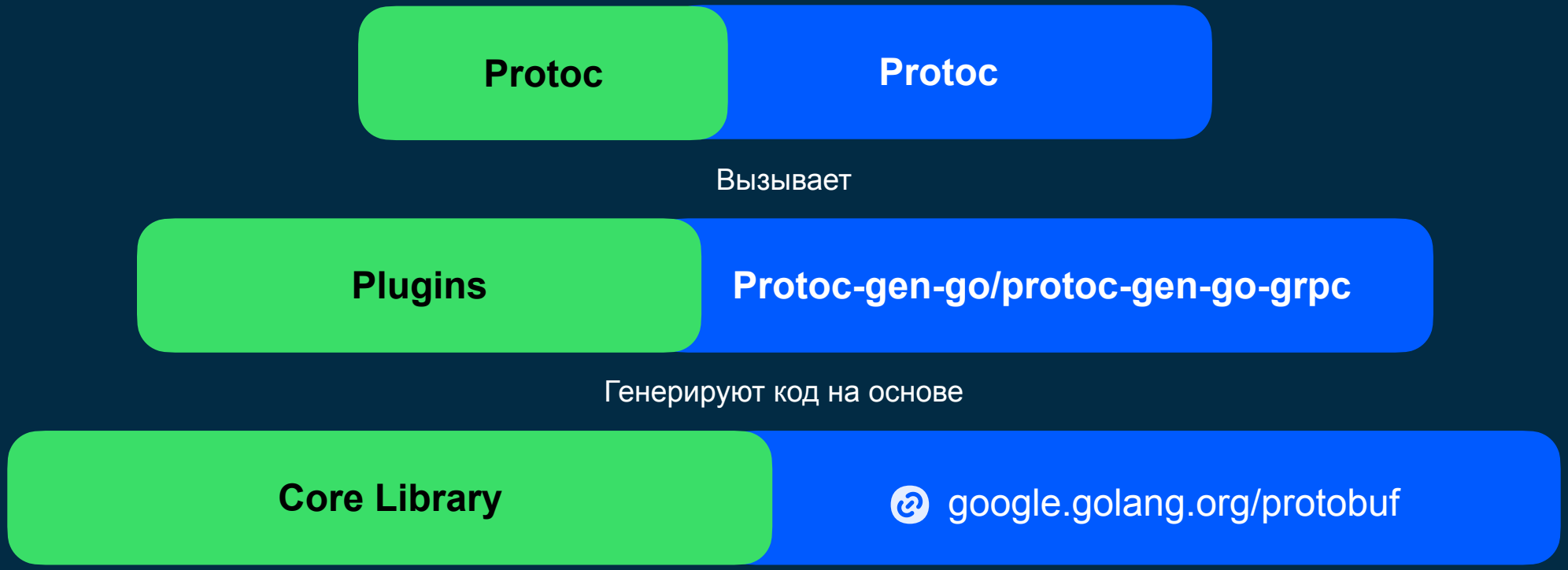


Второй блок — Plugin

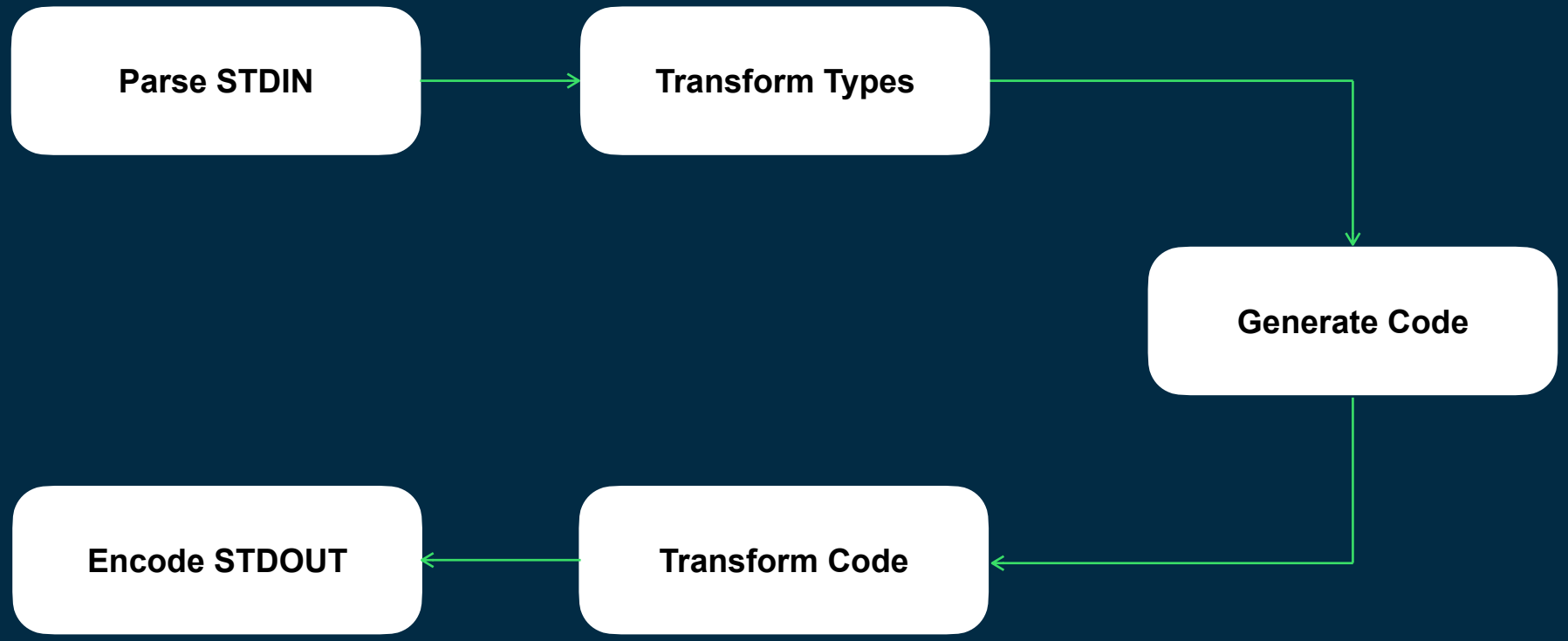
- **Маршаллинг/Анмаршаллинг — protoc-gen-go**
- **gRPC — protoc-gen-go-grpc**
- **Для написания плагинов есть все готовое — как компоненты, так и работающие примеры**
- **Как правило, каждый плагин генерирует какой-то свой кусочек кода и складывает в общий пакет**



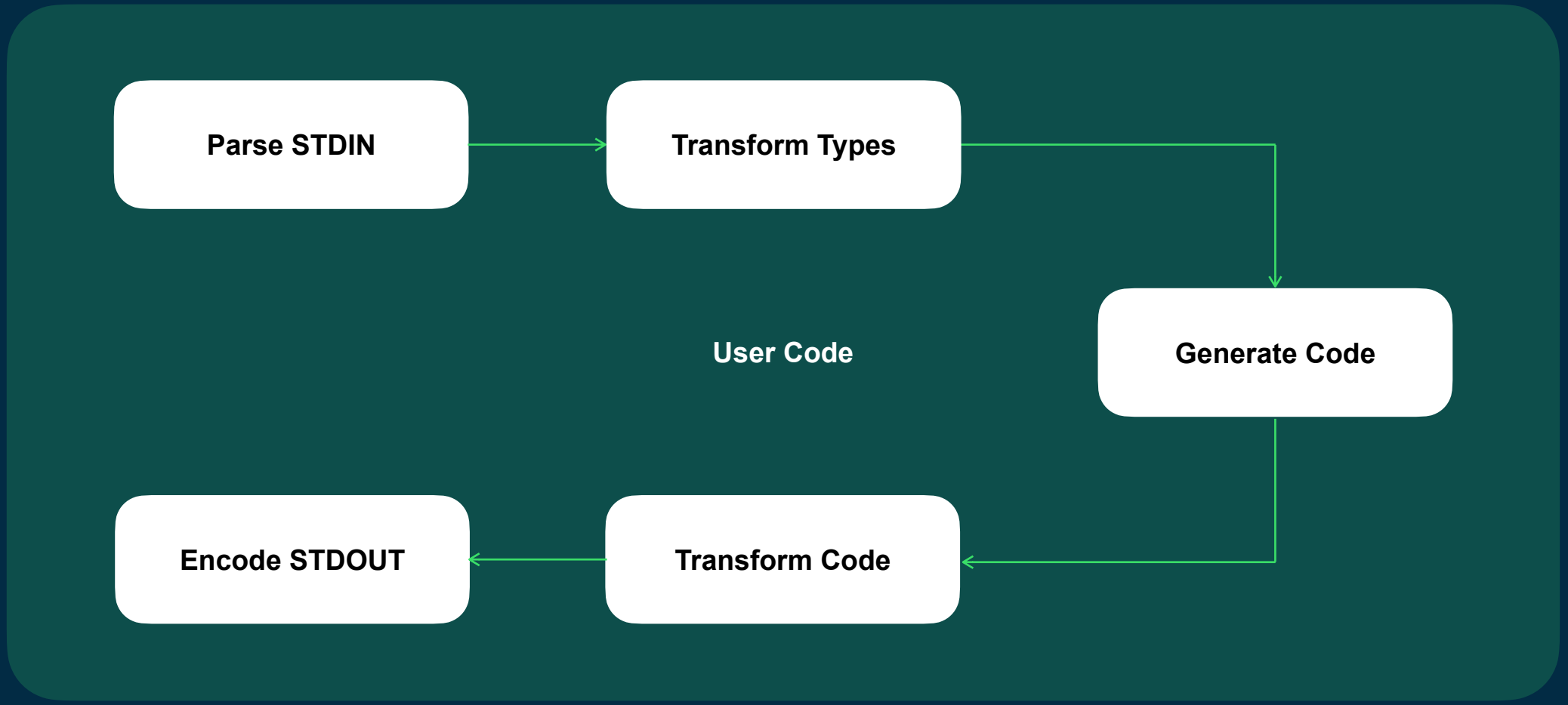
Protobuf-пирамида



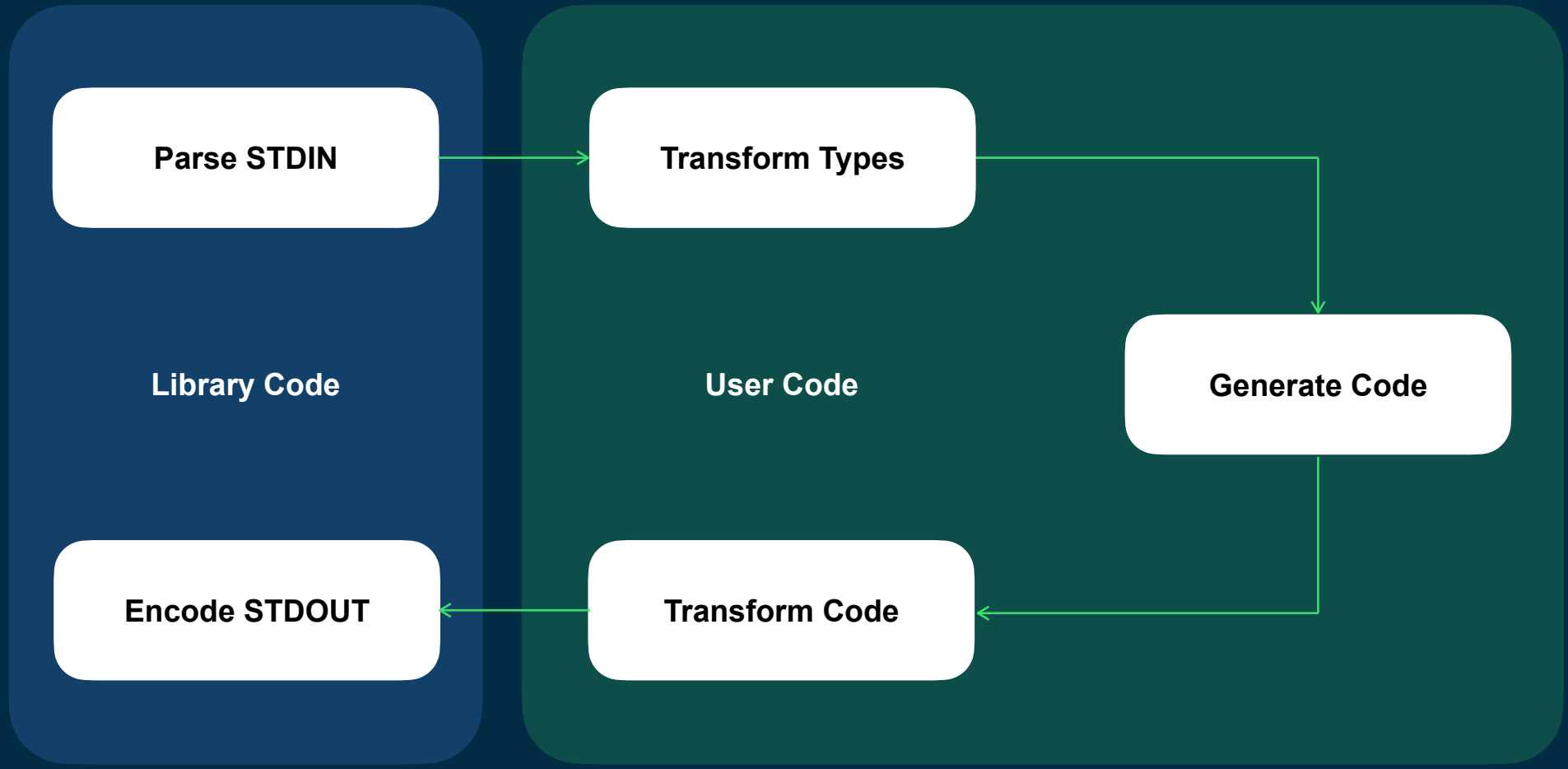
Plugin Flow


Parse STDIN
Transform Types
Generate Code
Transform Code
Encode STDOUT

Plugin Flow — General Way



Plugin Flow — Go Way

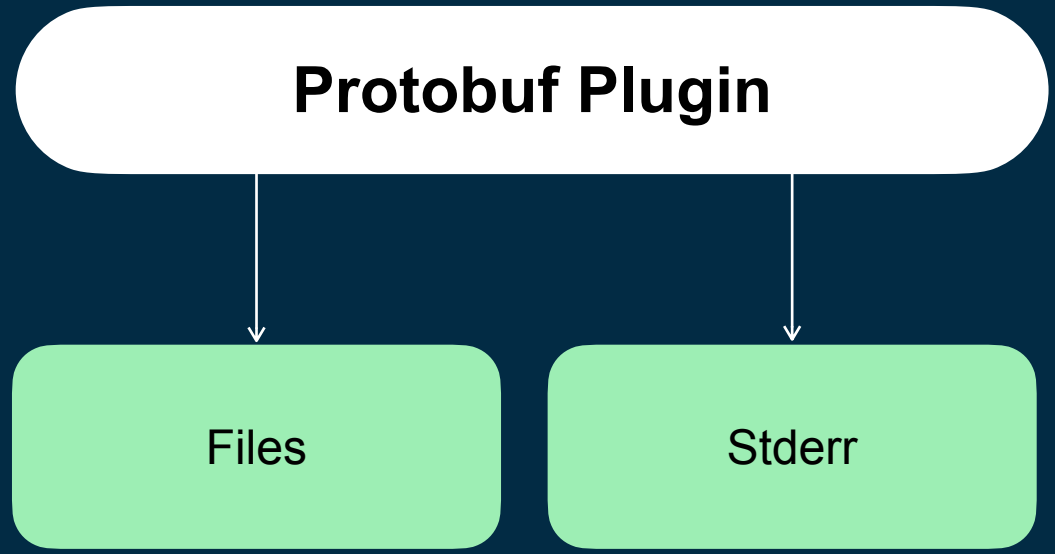


А может ли быть еще проще?

```go
package main

import "google.golang.org/protobuf/compiler/protogen"

func main() {
    protogen.Options{}.Run(func(p *protogen.Plugin) error {
        file := p.NewGeneratedFile("example.pb.go", "example")

        file.P("// Example ...")
        file.P("package example")

        return nil
    })
}

```

Как дебажить плагины?

- **Stderr — ваш друг. Все логи, записанные в него, будут переданы protoc пользователю**
- **Альтернатива — писать временные файлы с артефактами генерации**



Как расширять Protobuf – option

- **Protobuf предлагает механизм опций — настроек, которые можно применять как к файлу целиком, так и к отдельным сообщениям/полям**
- **В Protobuf можно создавать собственные опции**

```proto
syntax = "proto3";
package your.service.v1;

option go_package = "github.com/yourorg/yourprotos/gen/go/your/service/v1";

import "google/api/annotations.proto";

message StringMessage {
    string value = 1;
}

service YourService {
    rpc Echo(StringMessage) returns (StringMessage) {
        option (google.api.http) = {
            post: "/v1/example/echo"
            body: "*"
        }
    }
}
```

Как расширять Protobuf – option

- **Protobuf предлагает механизм опций — настроек, которые можно применять как к файлу целиком, так и к отдельным сообщениям/полям**
- **В Protobuf можно создавать собственные опции**

```proto
syntax = "proto3";
package your.service.v1;

option go_package = "github.com/yourorg/yourprotos/gen/go/your/service/v1";

import "google/api/annotations.proto";

message StringMessage {
    string value = 1;
}

service YourService {
    rpc Echo(StringMessage) returns (StringMessage) {
        option (google.api.http) = {
            post: "/v1/example/echo"
            body: "*"
        }
    }
}
```

Как можно применять option?

- **Переопределять пакет и имя файла**
- **Помечать сообщения для генерации**
- **Добавлять опциональные настройки**

```proto
syntax = "proto3";
package your.service.v1;

option go_package = "github.com/yourorg/yourprotos/gen/go/your/service/v1";

import "google/api/annotations.proto";

message StringMessage {
    string value = 1;
}

service YourService {
    rpc Echo(StringMessage) returns (StringMessage) {
        option (google.api.http) = {
            post: "/v1/example/echo"
            body: "*"
        }
    }
}
```

Пара тонкостей в Go-генерации

Пакет Protobuf не маппится
1 в 1 к пакетам Go

01

Для каждого файла proto задается
go_package – это нужно для связи
символов между Go-пакетами

02

go_package можно переопределять
с помощью M-флага без
модификации исходного файла

03

Это стандартный механизм, его
должны поддерживать все плагины

04

option go_package

go_package и M-флаги

3 output-режима

paths=source_relative — положит файл рядом с исходниками proto **01**

```
→  go-paths-example cat proto/subdir/example.proto | rg go_package
option go_package = "github.com/some/com/example";
→  go-paths-example exa --tree
.
└── proto
    └── subdir
        ├── example.pb.go
        └── example.proto
```

3 output-режима

paths=import — положит файл по Go-импорту **02**

```
● → go-paths-example cat proto/subdir/example.proto | rg go_package
option go_package = "github.com/some/com/example";
● → go-paths-example exa --tree
.
└── proto
    ├── github.com
    │   └── some
    │       └── com
    │           └── example
    │               └── example.pb.go
    └── subdir
        └── example.proto
```

3 output-режима

paths=$PREFIX — положит файл по Go-импорту без указанного префикса **03**

```
● → go-paths-example protoc -I proto --go_out=proto --go_opt=module=github.com/some/com  proto/subdir/example.proto
● → go-paths-example cat proto/subdir/example.proto | rg go_package
option go_package = "github.com/some/com/example";
● → go-paths-example exa --tree
.
└── proto
    ├── example
    │   └── example.pb.go
    └── subdir
        └── example.proto
```

Пример

- Как вызывать плагин/дебажить через логи
- Как обрабатывать Protobuf модели
- Как обрабатывать Protobuf аннотации

Source Code: github.com/defaulterrr/protoc-gen-pgx

Вместо заключения

Protobuf — удобный инструмент для описания доменных сущностей

01

Тулинг вокруг Protobuf построен специально так, чтобы его можно было легко расширять

02

Если вы уже работаете с Protobuf – возможно, дополнительная генерация на основе него может упростить ваши процессы

03

OZON tech

Спасибо за внимание!

Святослав Петров, старший разработчик

svpetrov@ozon.ru