gRPC 入门

www.javaboy.org

gRPC 是一个由 Google 发起的开源框架，它依赖于 HTTP/2、协议缓冲区等技术栈。和传统的 REST 相比，gRPC 具备更高的性能和更强大的可扩展性。

江南一点雨

1 gRPC 入门

这篇文章本来要在年前和小伙伴们见面，但是因为我之前的 Mac 系统版本是 10.13.6，这个版本比较老，时至今天在运行一些新鲜玩意的时候有时候会有一些 BUG（例如运行最新版的 Nacos 等），运行 gRPC 的插件也有 BUG，代码总是生成有问题，但是因为系统升级是一个大事，所以一直等到过年放假，在家才慢慢折腾将 Mac 升级到目前的 13.1 版本，之前这些问题现在都没有了，gRPC 的案例现在也可以顺利跑起来了。

所以今天就来和小伙伴们简单聊一聊 gRPC。

1.1 缘起

我为什么想写一篇 gRPC 的文章呢？其实本来我是想和小伙伴们梳理一下在微服务中都有哪些跨进城调用的方式，在梳理的过程中想到了 gRPC，发现还没写文章和小伙伴们聊过 gRPC，因此打算先来几篇文章和小伙伴们详细介绍一下 gRPC，然后再梳理微服务中的跨进程方案。

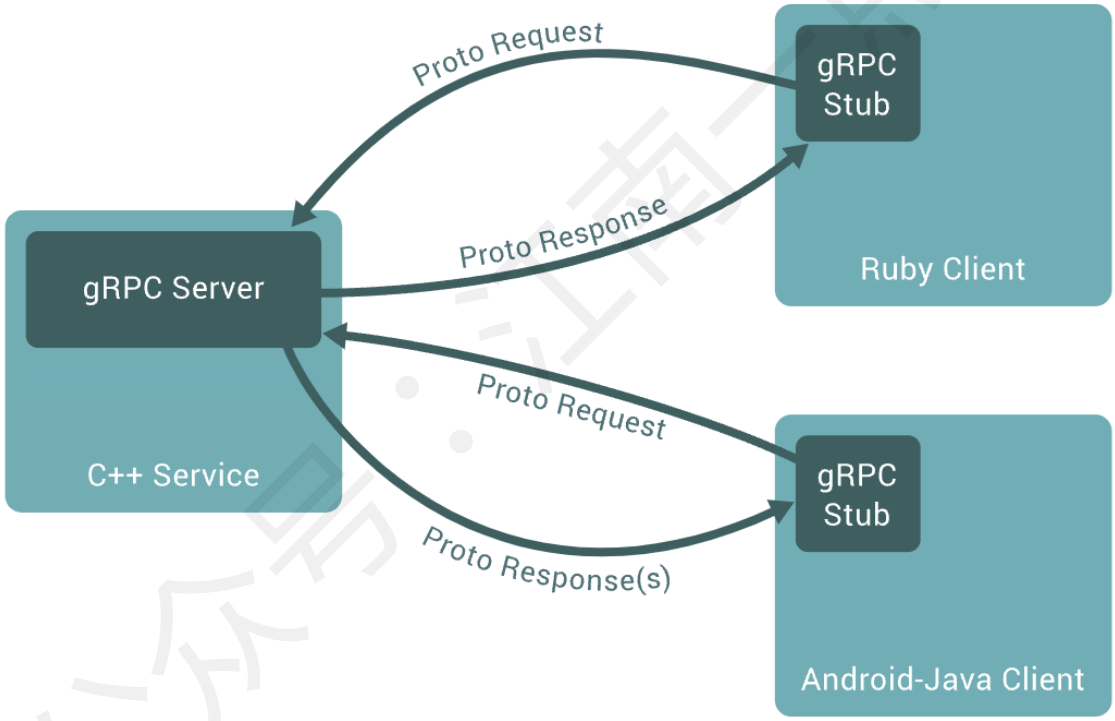
1.2 什么是 gRPC

了解 gRPC 之前先来看看什么是 RPC。

RPC 全称是 Remote Procedure Call，中文一般译作远程过程调用。RPC 是一种进程间的通信模式，程序分布在不同的地址空间里。简单来说，就是两个进程之间互相调用的一种方式。

gRPC 则是一个由 Google 发起的开源的 RPC 框架，它是一个高性能远程过程调用 (RPC) 框架，可以在任何环境中运行。gRPC 通过对负载均衡、跟踪、健康检查和身份验证的可插拔支持，有效地连接数据中心内和数据中心之间的服务。

在 gRPC 中，客户端应用程序可以直接调用部署在不同机器上的服务端应用程序中的方法，就好像它是本地对象一样，使用 gRPC 可以更容易地创建分布式应用程序和服务。与许多 RPC 系统一样，gRPC 基于定义服务的思想，指定基于参数和返回类型远程调用的方法。在服务端侧，服务端实现接口，运行 gRPC 服务，处理客户端调用。在客户端侧，客户端拥有存根（Stub，在某些语言中称为客户端），它提供与服务端相同的方法。



gRPC 客户端和服务端可以在各种环境中运行和相互通信 – 从 Google 内部的服务器到你自己的桌面 – 并且可以使用 gRPC 支持的任何语言编写。因此，你可以轻松地用 Java 创建 gRPC 服务端，使用 Go、Python 或 Ruby 创建客户端。此外，最新的 Google API 将包含 gRPC 版本的接口，使你轻松地将 Google 功能构建到你的应用程序中。

gRPC 支持的语言版本：

Language	OS	Compilers / SDK
C/C++	Linux, Mac	GCC 6.3+, Clang 6+
C/C++	Windows 10+	Visual Studio 2017+
C#	Linux, Mac	.NET Core, Mono 4+
C#	Windows 10+	.NET Core, NET 4.5+
Dart	Windows, Linux, Mac	Dart 2.12+
Go	Windows, Linux, Mac	Go 1.13+
Java	Windows, Linux, Mac	Java 8+ (KitKat+ for Android)
Kotlin	Windows, Linux, Mac	Kotlin 1.3+
Node.js	Windows, Linux, Mac	Node v8+
Objective-C	macOS 10.10+, iOS 9.0+	Xcode 12+
PHP	Linux, Mac	PHP 7.0+
Python	Windows, Linux, Mac	Python 3.7+
Ruby	Windows, Linux, Mac	Ruby 2.3+

说了这么多，还是得整两个小案例小伙伴们可能才会清晰，所以我们也不废话了，上案例。

1.3 实践

先来看下我们的项目结构：

```
├── grpc-api
│   ├── pom.xml
│   ├── src
├── grpc-client
│   ├── pom.xml
│   ├── src
├── grpc-server
│   ├── pom.xml
│   ├── src
└── pom.xml
```

大家看下，这里首先有一个 grpc-api，这个模块用来放我们的公共代码；grpc-server 是我们的服务端，grpc-client 则是我们的客户端，这些都是普通的 maven 项目。

1.3.1 grpc-api

在 grpc-api 中，我们首先引入项目依赖，如下：

```
<dependencies>
    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-netty-shaded</artifactId>
        <version>1.52.1</version>
    </dependency>
    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-protobuf</artifactId>
        <version>1.52.1</version>
    </dependency>
    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-stub</artifactId>
        <version>1.52.1</version>
    </dependency>
    <dependency> <!-- necessary for Java 9+ -->
        <groupId>org.apache.tomcat</groupId>
        <artifactId>annotations-api</artifactId>
        <version>6.0.53</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

除了这些常规的依赖之外，还需要一个插件：

```
<build>
    <extensions>
        <extension>
            <groupId>kr.motd.maven</groupId>
```

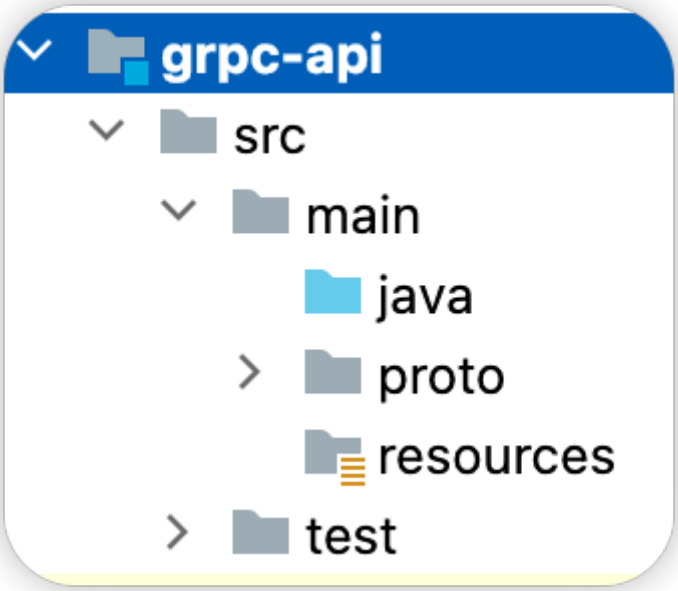
```
            <artifactId>os-maven-plugin</artifactId>
            <version>1.6.2</version>
        </extension>
    </extensions>
    <plugins>
        <plugin>
            <groupId>org.xolstice.maven.plugins</groupId>
            <artifactId>protobuf-maven-plugin</artifactId>
            <version>0.6.1</version>
            <configuration>

 <protocArtifact>com.google.protobuf:protoc:3.21.7:exe:${os.detected.cla
ssifier}</protocArtifact>
                <pluginId>grpc-java</pluginId>
                <pluginArtifact>io.grpc:protoc-gen-grpc-
java:1.51.0:exe:${os.detected.classifier}</pluginArtifact>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                        <goal>compile-custom</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

我来说一下这个插件的作用。

默认情况下，gRPC 使用 Protocol Buffers，这是 Google 提供的一个成熟的开源的跨平台的序列化数据结构的协议，我们编写对应的 proto 文件，通过上面这个插件可以将我们编写的 proto 文件自动转为对应的 Java 类。

> *多说一句，使用 Protocol Buffers 并不是必须的，也可以使用 JSON 等，但是目前来说这个场景更常用的还是 Portal Buffers。*

接下来我们在 main 目录下新建 proto 文件夹，如下：



注意，这个文件夹位置是默认的。如果我们的 proto 文件不是放在 src/main/proto 位置，那么在配置插件的时候需要指定 proto 文件的位置，咱们本篇文章主要是入门，我这里就使用默认的位置。

在 proto 文件夹中，我们新建一个 product.proto 文件，内容如下：

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.javaboy.grpc.demo";
option java_outer_classname = "ProductProto";

package product;

service ProductInfo {
  rpc addProduct (Product) returns (ProductId);
  rpc getProduct(ProductId) returns(Product);
}

message Product {
  string id = 1;
  string name=2;
  string description=3;
  float price=4;
}

message ProductId {
  string value = 1;
}
```

这段配置算是一个比较核心的配置了，这里主要说明了负责进程传输的类、方法等到底是个啥样子：

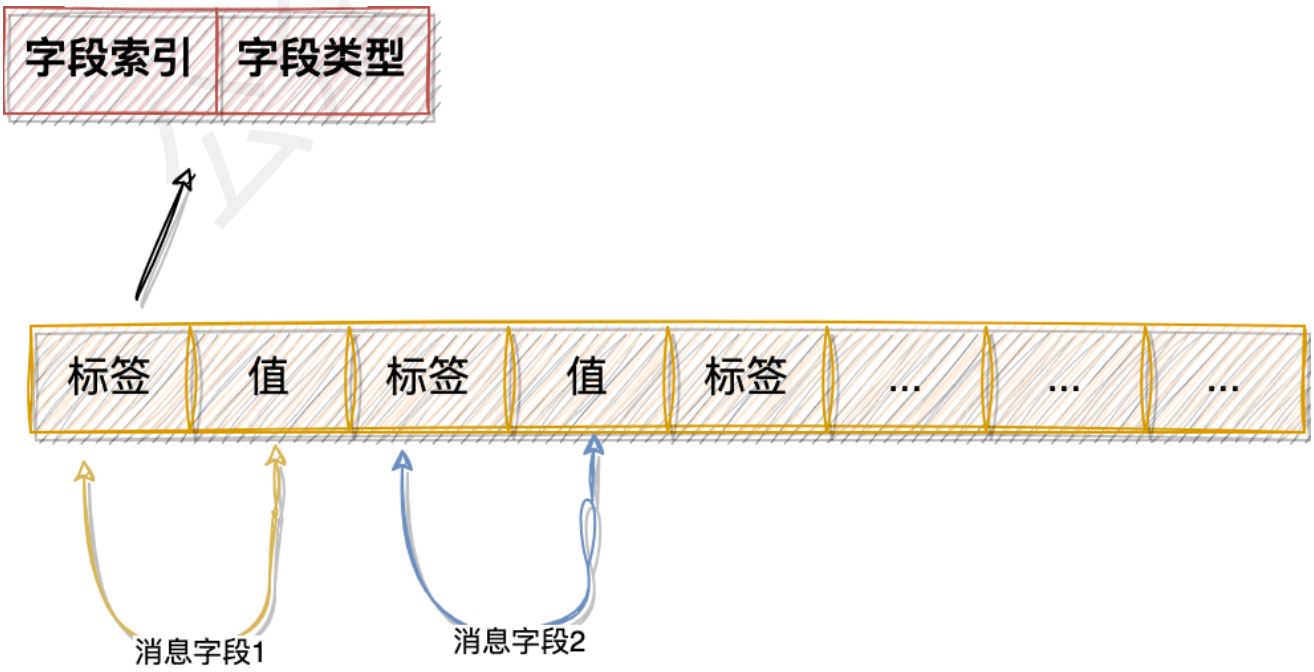
1. `syntax = "proto3";`：这个是 protocol buffers 的版本。
2. `option java_multiple_files = true;`：这个字段是可选的，如果设置为 true，表示每一个 message 文件都会有一个单独的 class 文件；否则，message 全部定义在 outerclass 文件里。
3. `option java_package = "org.javaboy.grpc.demo";`：这个字段是可选的，用于标识生成的 java 文件的 package。如果没有指定，则使用 proto 里定义的 package，如果package 也没有指定，那就会生成在根目录下。
4. `option java_outer_classname = "ProductProto";`：这个字段是可选的，用于指定 proto 文件生成的 java 类的 outerclass 类名。什么是 outerclass？简单来说就是用一个 class 文件来定义所有的 message 对应的 Java 类，这个 class 就是 outerclass；如果没有指定，默认是 proto 文件的驼峰式；
5. `package product;`：这个属性用来定义 message 的包名。包名的含义与平台语言无关，这个 package 仅仅被用在 proto 文件中用于区分同名的 message 类型。可以理解为 message 全名的前缀，和 message 名称合起来唯一标识一个 message 类型。当我们在 proto 文件中导入其他 proto 文件的 message，需要加上 package 前缀才行。所以包名是用来唯一标识 message 的。
6. `service`：我们定义的跨平台方法都写在 service 中，上面的案例中我们定义了两个方法：addProduct 表示添加一件商品，参数是一个 Product 对象，返回值则是刚刚添加成功的商品的 ID；getProduct 则表示根据 ID 查询一个商品，参数是一个商品 ID，返回值则是查询到的商品对象。这里的定义相当于一个接口，将来我们要在 Java 代码中实现这个接口。
7. `message`：这里有点像我们在 Java 中定义类，上文中我们定义了两个类，分别是 Product 和 ProductId 两个类。这两个类在 service 中被使用。

message 中定义的有点像我们 Java 中定义的类，但是不能直接使用 Java 中的数据类型，毕竟这是 Protocol buffers，这个是和语言无关的，将来可以据此生成不同语言的代码，这里我们可以使用的类型和我们 Java 类型之间的对应关系如下：

.proto Type	Notes	C++ Type	Java/Kotlin Type[1]	Python Type[3]	Go Type	Ruby Type	C# Type	PHP Type
double		double	double	float	float64	Float	double	float
float		float	float	float	float32	Float	float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int	int32	Fixnum or Bignum (as required)	int	integer
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long[4]	int64	Bignum	long	integer/str
uint32	Uses variable-length encoding.	uint32	int[2]	int/long[4]	uint32	Fixnum or Bignum (as required)	uint	integer
uint64	Uses variable-length encoding.	uint64	long[2]	int/long[4]	uint64	Bignum	ulong	integer/str
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int	int32	Fixnum or Bignum (as required)	int	integer
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long[4]	int64	Bignum	long	integer/str
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28}.	uint32	int[2]	int/long[4]	uint32	Fixnum or Bignum (as required)	uint	integer
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56}.	uint64	long[2]	int/long[4]	uint64	Bignum	ulong	integer/str
sfixed32	Always four bytes.	int32	int	int	int32	Fixnum or Bignum (as required)	int	integer
sfixed64	Always eight bytes.	int64	long	int/long[4]	int64	Bignum	long	integer/str
bool		bool	boolean	bool	bool	TrueClass/FalseClass	bool	boolean
string	A string must always contain UTF-8 encoded or 7-bit ASCII text, and cannot be longer than 2^{32}.	string	String	str/unicode[5]	string	String (UTF-8)	string	string
bytes	May contain any arbitrary sequence of bytes no longer than 2^{32}.	string	ByteString	str (Python 2) bytes (Python	[]byte	String (ASCII-8BIT)	ByteString	string

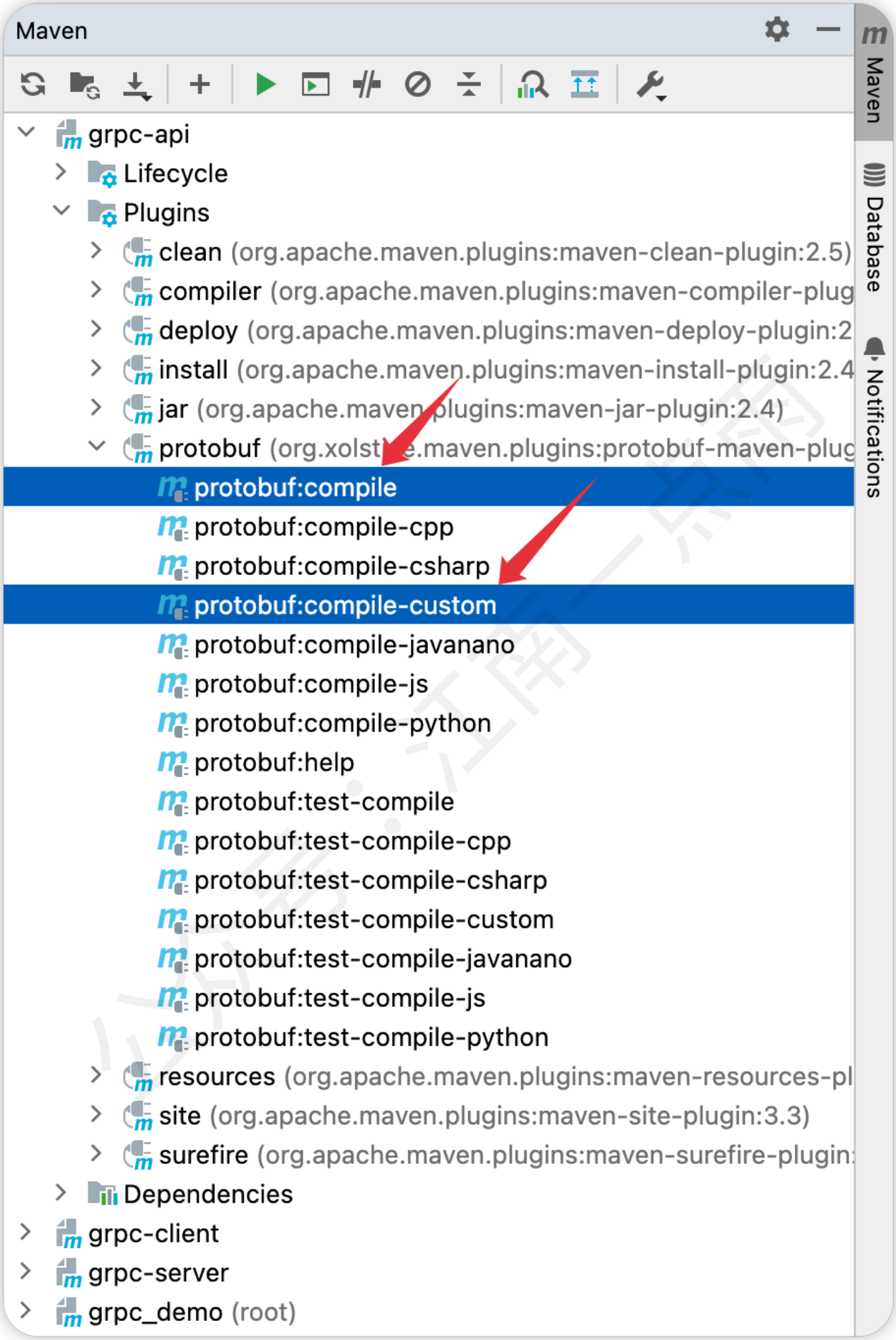
另外我们在 message 中定义的属性的时候，都会给一个数字，例如 id=1，name=2 等，这个数字将来会在二进制消息中标识我们的字段，并且一旦我们的消息类型被使用就不应更改，这个有点像序列化的感觉。

实际上，这个 message 编译后的字节内容大概像下面这样：



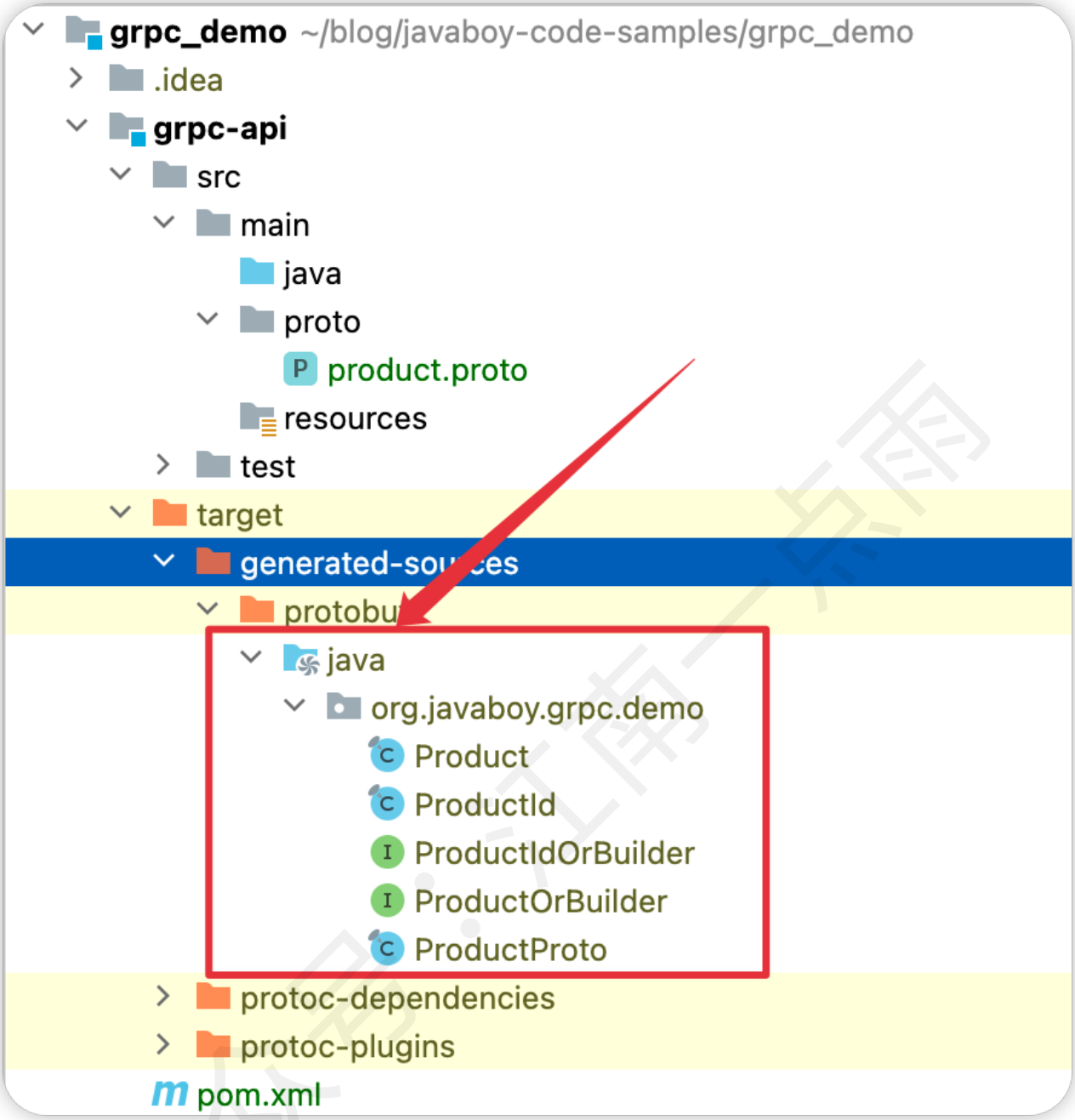
这里的标签中的内容包含两部分，字段索引和字段类型，字段索引其实就是我们上面定义的数字。

定义完成之后，接下来我们就需要使用插件来生成对应的 Java 代码了，插件我们在前面已经引入了，现在只需要执行了，如下图：

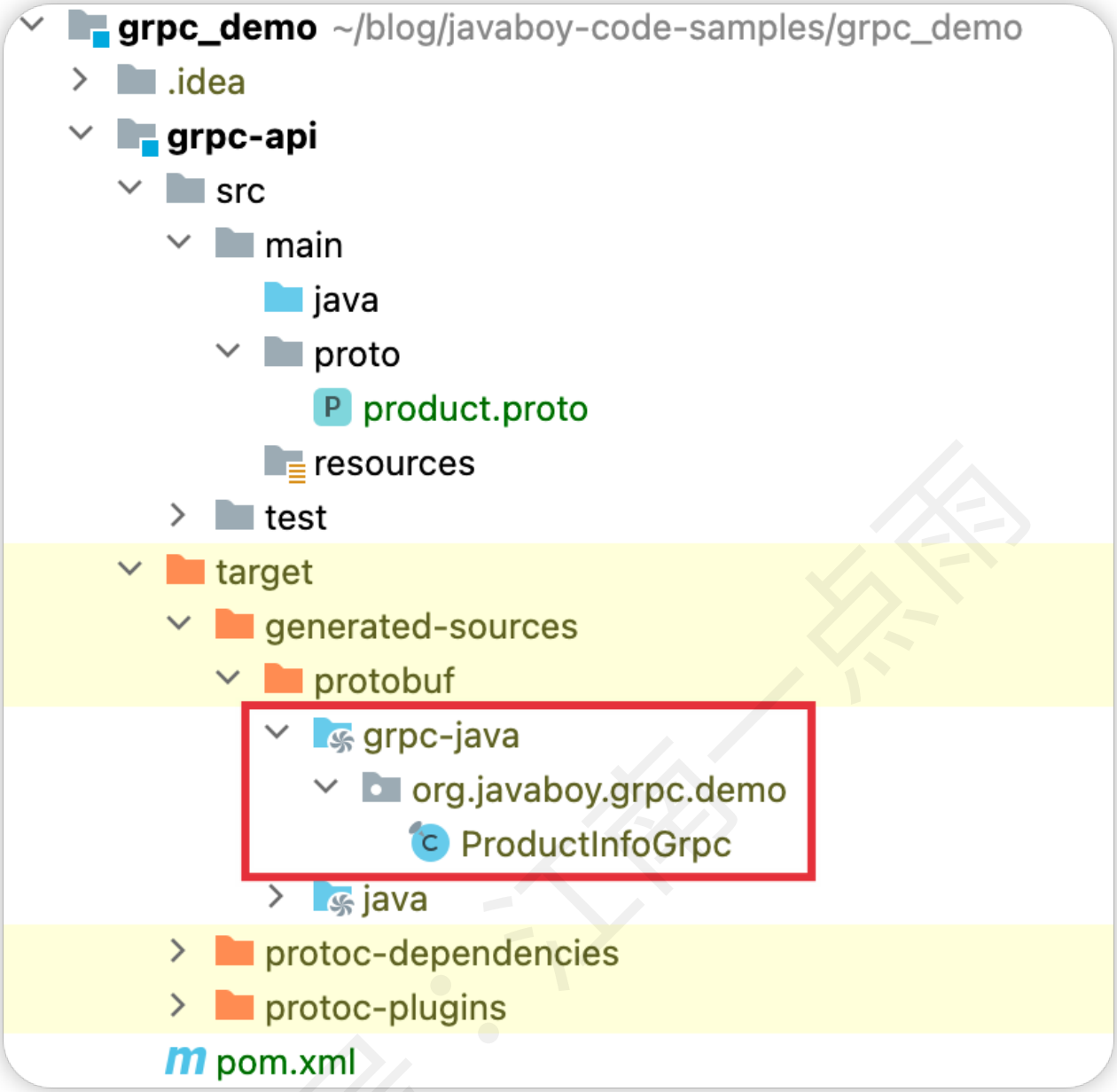


注意，compile 和 compile-custom 两个指令都需要执行。其中 compile 用来编译消息对象，compile-custom 则依赖消息对象,生成接口服务。

首先我们点击 compile 看看生成的代码，如下：



再看 compile-custom 生成的代码，如下：



好了，这样我们的准备工作就算完成了。

> *有的小伙伴生成的代码文件夹颜色不对劲，此时有两种解决办法：1.选中目标文件夹，右键单击，选择 Mark Directory as-> Generated Sources root；2.选中工程，右键单击，选择 Maven->Reload project。推荐使用第二种方案。*

1.3.2 grpc-server

接下来我们创建 grpc-server 项目，并使该项目依赖 grpc-api，然后在 grpc-server 中，提供 ProductInfo 的具体实现：

```
public class ProductInfoImpl extends ProductInfoGrpc.ProductInfoImplBase
{
    @Override
    public void addProduct(Product request, StreamObserver<ProductId>
responseObserver) {
```

```
        System.out.println("request.toString() = " +
request.toString());

 responseObserver.onNext(ProductId.newBuilder().setValue(request.getId()
).build());
        responseObserver.onCompleted();
    }

    @Override
    public void getProduct(ProductId request, StreamObserver<Product>
responseObserver) {

 responseObserver.onNext(Product.newBuilder().setId(request.getValue()).
setName("三国演义").build());
        responseObserver.onCompleted();
    }
}
```

`ProductInfoGrpc.ProductInfoImplBase` 是根据我们在 proto 文件中定义的 service 自动生成的，我们的 ProductInfoImpl 继承自该类，并且提供了我们给出的方法的具体实现。

以 addProduct 方法为例，参数 request 就是将来客户端调用的时候传来的 Product 对象，返回结果则通过 responseObserver 来完成。我们的方法逻辑很简单，我就把参数传来的 Product 对象打印出来，然后构建一个 ProductId 对象并返回，最后调用 `responseObserver.onCompleted();` 表示数据返回完毕。

剩下的 getProduct 方法逻辑就很好懂了，我这里就不再赘述了。

最后，我们再把这个 grpc-server 项目启动起来：

```
public class ProductInfoServer {
    Server server;

    public static void main(String[] args) throws IOException,
InterruptedException {
        ProductInfoServer server = new ProductInfoServer();
        server.start();
        server.blockUntilShutdown();
    }

    public void start() throws IOException {
        int port = 50051;
        server = ServerBuilder.forPort(port)
                .addService(new ProductInfoImpl())
                .build()
                .start();
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            ProductInfoServer.this.stop();
        }));
```

```
    }

    private void stop() {
        if (server != null) {
            server.shutdown();
        }
    }

    private void blockUntilShutdown() throws InterruptedException {
        if (server != null) {
            server.awaitTermination();
        }
    }
}
```

由于我们这里是一个 JavaSE 项目，为了避免项目启动之后就停止，我们这里调用了 `server.awaitTermination();` 方法，就是让服务启动成功之后不要停止。

1.3.3 grpc-client

最后再来看看客户端的调用。首先 grpc-client 项目也是需要依赖 grpc-api 的，然后直接进行方法调用，如下：

```
public class ProductClient {
    public static void main(String[] args) {
        ManagedChannel channel =
ManagedChannelBuilder.forAddress("localhost", 50051)
                .usePlaintext()
                .build();
        ProductInfoGrpc.ProductInfoBlockingStub stub =
ProductInfoGrpc.newBlockingStub(channel);
        Product p = Product.newBuilder().setId("1")
                .setPrice(399.0f)
                .setName("TienChin项目")
                .setDescription("SpringBoot+Vue3实战视频")
                .build();
        ProductId productId = stub.addProduct(p);
        System.out.println("productId.getValue() = " +
productId.getValue());
        Product product =
stub.getProduct(ProductId.newBuilder().setValue("99999").build());
        System.out.println("product.toString() = " +
product.toString());
    }
}
```

小伙伴们看到，这里首先需要和服务端建立连接，给出服务端的地址和端口号即可，usePlaintext() 方法表示不使用 TLS 对连接进行加密（默认情况下会使用 TLS 对连接进行加密），生产环境建议使用加密连接。

剩下的代码就比较好懂了，创建 Product 对象，调用 addProduct 方法进行添加；创建 ProductId 对象，调用 getProduct。Product 对象和 ProductId 对象都是根据我们在 proto 中定义的 message 自动生成的。

1.4 总结

好啦，一个简单的例子，小伙伴们先对 gRPC 入个门，后面松哥会再整几篇文章跟大家介绍这里边的一些细节。

2 gRPC 的四种通信模式

> *温馨提示：本文需要结合上一篇 gRPC 文章一起食用，否则可能看不懂。*

前面一篇文章松哥和大家聊了 gRPC 的基本用法，今天我们再来稍微深入一点点，来看下 gRPC 中四种不同的通信模式。

gRPC 中四种不同的通信模式分别是：

1. 一元 RPC
2. 服务端流 RPC
3. 客户端流 RPC
4. 双向流 RPC

接下来松哥就通过四个完整的案例，来分别和向伙伴们演示这四种不同的通信模式。

2.1 准备工作

关于 gRPC 的基础知识我们就不啰嗦了，咱们直接来看我今天的 proto 文件，如下：

这次我新建了一个名为 book.proto 的文件，这里主要定义了一些图书相关的方法，如下：

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.javaboy.grpc.demo";
option java_outer_classname = "BookServiceProto";
import "google/protobuf/wrappers.proto";

package book;

service BookService {
```

```
  rpc addBook(Book) returns (google.protobuf.StringValue);
  rpc getBook(google.protobuf.StringValue) returns (Book);
  rpc searchBooks(google.protobuf.StringValue) returns (stream Book);
  rpc updateBooks(stream Book) returns (google.protobuf.StringValue);
  rpc processBooks(stream google.protobuf.StringValue) returns (stream
BookSet);
}

message Book {
  string id = 1;
  repeated string tags = 2;
  string name = 3;
  float price = 4;
  string author = 5;
}

message BookSet {
  string id = 1;
  repeated Book bookList = 3;
}
```

这个文件中，有一些内容我们在上篇文章中都讲过了，讲过的我就不再重复了，我说一些上篇文章没有涉及到的东西：

1. 由于我们在这个文件中，引用了 Google 提供的 StringValue（`google.protobuf.StringValue`），所以这个文件上面我们首先用 import 导入相关的文件，导入之后，才可以使用。
2. 在方法参数和返回值中出现的 stream，就表示这个方法的参数或者返回值是流的形式（其实就是数据可以多次传输）。
3. message 中出现了一个上篇文章没有的关键字 repeated，这个表示这个字段可以重复，可以简单理解为这就是我们 Java 中的数组。

好了，和上篇文章相比，本文主要就是这几个地方不一样。

proto 文件写好之后，按照上篇文章介绍的方法进行编译，生成对应的代码，这里就不再重复了。

2.2 一元 RPC

一元 RPC 是一种比较简单的 RPC 模式，其实说白了我们上篇文章和大家介绍的就是一种一元 RPC，也就是客户端发起一个请求，服务端给出一个响应，然后请求结束。

上面我们定义的五个方法中，addBook 和 getBook 都算是一种一元 RPC。

2.2.1 addBook

先来看 addBook 方法，这个方法的逻辑很简单，我们提前在服务端准备一个 Map 用来保存 Book，addBook 调用的时候，就把 book 对象存入到 Map 中，并且将 book 的 ID 返回，大家就这样一件事，来看看服务端的代码：

```
public class BookServiceImpl extends BookServiceGrpc.BookServiceImplBase
{
    private Map<String, Book> bookMap = new HashMap<>();

    public BookServiceImpl() {
        Book b1 = Book.newBuilder().setId("1").setName("三国演
义").setAuthor("罗贯中").setPrice(30).addTags("明清小说").addTags("通俗小
说").build();
        Book b2 = Book.newBuilder().setId("2").setName("西游
记").setAuthor("吴承恩").setPrice(40).addTags("志怪小说").addTags("通俗小
说").build();
        Book b3 = Book.newBuilder().setId("3").setName("水浒
传").setAuthor("施耐庵").setPrice(50).addTags("明清小说").addTags("通俗小
说").build();
        bookMap.put("1", b1);
        bookMap.put("2", b2);
        bookMap.put("3", b3);
    }

    @Override
    public void addBook(Book request, StreamObserver<StringValue>
responseObserver) {
        bookMap.put(request.getId(), request);

 responseObserver.onNext(StringValue.newBuilder().setValue(request.getId
()).build());
        responseObserver.onCompleted();
    }
}
```

看过上篇文章的小伙伴，我觉得这段代码应该很好理解。

客户端调用方式如下：

```
public class BookServiceClient {
    public static void main(String[] args) throws InterruptedException {
        ManagedChannel channel =
ManagedChannelBuilder.forAddress("localhost", 50051)
                .usePlaintext()
                .build();
        BookServiceGrpc.BookServiceStub stub =
BookServiceGrpc.newStub(channel);
        addBook(stub);
```

```
    }

    private static void addBook(BookServiceGrpc.BookServiceStub stub)
throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(1);

 stub.addBook(Book.newBuilder().setPrice(99).setId("100").setName("java"
).setAuthor("javaboy").build(), new StreamObserver<StringValue>() {
            @Override
            public void onNext(StringValue stringValue) {
                System.out.println("stringValue.getValue() = " +
stringValue.getValue());
            }

            @Override
            public void onError(Throwable throwable) {

            }

            @Override
            public void onCompleted() {
                countDownLatch.countDown();
                System.out.println("添加完毕");
            }
        });
        countDownLatch.await();
    }
}
```

这里我使用了 CountDownLatch 来实现线程等待，等服务端给出响应之后，客户端再结束。这里在回调的 onNext 方法中，我们就可以拿到服务端的返回值。

2.2.2 getBook

getBook 跟上面的 addBook 类似，先来看服务端代码，如下：

```
public class BookServiceImpl extends BookServiceGrpc.BookServiceImplBase
{
    private Map<String, Book> bookMap = new HashMap<>();

    public BookServiceImpl() {
        Book b1 = Book.newBuilder().setId("1").setName("三国演
义").setAuthor("罗贯中").setPrice(30).addTags("明清小说").addTags("通俗小
说").build();
        Book b2 = Book.newBuilder().setId("2").setName("西游
记").setAuthor("吴承恩").setPrice(40).addTags("志怪小说").addTags("通俗小
说").build();
```

```
        Book b3 = Book.newBuilder().setId("3").setName("水浒
传").setAuthor("施耐庵").setPrice(50).addTags("明清小说").addTags("通俗小
说").build();
        bookMap.put("1", b1);
        bookMap.put("2", b2);
        bookMap.put("3", b3);
    }

    @Override
    public void getBook(StringValue request, StreamObserver<Book>
responseObserver) {
        String id = request.getValue();
        Book book = bookMap.get(id);
        if (book != null) {
            responseObserver.onNext(book);
            responseObserver.onCompleted();
        } else {
            responseObserver.onCompleted();
        }
    }
}
```

这个 getBook 就是根据客户端传来的 id，从 Map 中查询到一个 Book 并返回。

客户端调用代码如下：

```
public class BookServiceClient {
    public static void main(String[] args) throws InterruptedException {
        ManagedChannel channel =
ManagedChannelBuilder.forAddress("localhost", 50051)
                .usePlaintext()
                .build();
        BookServiceGrpc.BookServiceStub stub =
BookServiceGrpc.newStub(channel);
        getBook(stub);
    }

    private static void getBook(BookServiceGrpc.BookServiceStub stub)
throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(1);
        stub.getBook(StringValue.newBuilder().setValue("2").build(), new
StreamObserver<Book>() {
            @Override
            public void onNext(Book book) {
                System.out.println("book = " + book);
            }

            @Override
            public void onError(Throwable throwable) {
```

```
            }

            @Override
            public void onCompleted() {
                countDownLatch.countDown();
                System.out.println("查询完毕");
            }
        });
        countDownLatch.await();
    }
}
```

小伙伴们大概也能看出来，addBook 和 getBook 基本上操作套路是一模一样的。

2.3 服务端流 RPC

前面的一元 RPC，客户端发起一个请求，服务端给出一个响应，请求就结束了。服务端流则是客户端发起一个请求，服务端给一个响应序列，这个响应序列组成一个流。

上面我们给出的 searchBook 就是这样一个例子，searchBook 是传递图书的 tags 参数，然后在服务端查询哪些书的 tags 满足条件，将满足条件的书全部都返回去。

我们来看下服务端的代码：

```
public class BookServiceImpl extends BookServiceGrpc.BookServiceImplBase
{
    private Map<String, Book> bookMap = new HashMap<>();

    public BookServiceImpl() {
        Book b1 = Book.newBuilder().setId("1").setName("三国演
义").setAuthor("罗贯中").setPrice(30).addTags("明清小说").addTags("通俗小
说").build();
        Book b2 = Book.newBuilder().setId("2").setName("西游
记").setAuthor("吴承恩").setPrice(40).addTags("志怪小说").addTags("通俗小
说").build();
        Book b3 = Book.newBuilder().setId("3").setName("水浒
传").setAuthor("施耐庵").setPrice(50).addTags("明清小说").addTags("通俗小
说").build();
        bookMap.put("1", b1);
        bookMap.put("2", b2);
        bookMap.put("3", b3);
    }
    @Override
    public void searchBooks(StringValue request, StreamObserver<Book>
responseObserver) {
        Set<String> keySet = bookMap.keySet();
        String tags = request.getValue();
        for (String key : keySet) {
```

```
                Book book = bookMap.get(key);
                int tagsCount = book.getTagsCount();
                for (int i = 0; i < tagsCount; i++) {
                    String t = book.getTags(i);
                    if (t.equals(tags)) {
                        responseObserver.onNext(book);
                        break;
                    }
                }
            }
            responseObserver.onCompleted();
        }
    }
```

小伙伴们看下，这段 Java 代码应该很好理解：

1. 首先从 request 中提取客户端传来的 tags 参数。
2. 遍历 bookMap，查看每一本书的 tags 是否等于客户端传来的 tags，如果相等，说明添加匹配，则通过 `responseObserver.onNext(book);` 将这本书写回到客户端。
3. 等所有操作都完成后，执行 `responseObserver.onCompleted();`，表示服务端的响应序列结束了，这样客户端也就知道请求结束了。

我们来看看客户端的代码，如下：

```
public class BookServiceClient {
    public static void main(String[] args) throws InterruptedException {
        ManagedChannel channel =
ManagedChannelBuilder.forAddress("localhost", 50051)
                .usePlaintext()
                .build();
        BookServiceGrpc.BookServiceStub stub =
BookServiceGrpc.newStub(channel);
        searchBook(stub);
    }
    private static void searchBook(BookServiceGrpc.BookServiceStub stub)
throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(1);
        stub.searchBooks(StringValue.newBuilder().setValue("明清小
说").build(), new StreamObserver<Book>() {
            @Override
            public void onNext(Book book) {
                System.out.println(book);
            }

            @Override
            public void onError(Throwable throwable) {

            }
```

```
            @Override
            public void onCompleted() {
                countDownLatch.countDown();
                System.out.println("查询完毕！");
            }
        });
        countDownLatch.await();
    }
}
```

客户端的代码好理解，搜索的关键字是 明清小说，每当服务端返回一次数据的时候，客户端回调的 onNext 方法就会被触发一次，当服务端之行了 `responseObserver.onCompleted();` 之后，客户端的 onCompleted 方法也会被触发。

这个就是服务端流，客户端发起一个请求，服务端通过 onNext 可以多次写回数据。

2.4 客户端流 RPC

客户端流则是客户端发起多个请求，服务端只给出一个响应。

上面的 updateBooks 就是一个客户端流的案例，客户端想要修改图书，可以发起多个请求修改多本书，服务端则收集多次修改的结果，将之汇总然后一次性返回给客户端。

我们先来看看服务端的代码：

```
public class BookServiceImpl extends BookServiceGrpc.BookServiceImplBase
{
    private Map<String, Book> bookMap = new HashMap<>();

    public BookServiceImpl() {
        Book b1 = Book.newBuilder().setId("1").setName("三国演
义").setAuthor("罗贯中").setPrice(30).addTags("明清小说").addTags("通俗小
说").build();
        Book b2 = Book.newBuilder().setId("2").setName("西游
记").setAuthor("吴承恩").setPrice(40).addTags("志怪小说").addTags("通俗小
说").build();
        Book b3 = Book.newBuilder().setId("3").setName("水浒
传").setAuthor("施耐庵").setPrice(50).addTags("明清小说").addTags("通俗小
说").build();
        bookMap.put("1", b1);
        bookMap.put("2", b2);
        bookMap.put("3", b3);
    }

    @Override
    public StreamObserver<Book> updateBooks(StreamObserver<StringValue>
responseObserver) {
        StringBuilder sb = new StringBuilder("更新的图书 ID 为: ");
```

```
        return new StreamObserver<Book>() {
            @Override
            public void onNext(Book book) {
                bookMap.put(book.getId(), book);
                sb.append(book.getId())
                        .append(",");
            }

            @Override
            public void onError(Throwable throwable) {

            }

            @Override
            public void onCompleted() {

 responseObserver.onNext(StringValue.newBuilder().setValue(sb.toString()
).build());
                responseObserver.onCompleted();
            }
        };
    }
}
```

客户端每发送一本书来，就会触发服务端的 onNext 方法，然后我们在这方法中进行图书的更新操作，并记录更新结果。最后，我们在 onCompleted 方法中，将更新结果汇总返回给客户端，基本上就是这样一个流程。

我们再来看看客户端的代码：

```
public class BookServiceClient {
    public static void main(String[] args) throws InterruptedException {
        ManagedChannel channel =
ManagedChannelBuilder.forAddress("localhost", 50051)
                .usePlaintext()
                .build();
        BookServiceGrpc.BookServiceStub stub =
BookServiceGrpc.newStub(channel);
        updateBook(stub);
    }

    private static void updateBook(BookServiceGrpc.BookServiceStub stub)
throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(1);
        StreamObserver<Book> request = stub.updateBooks(new
StreamObserver<StringValue>() {
            @Override
            public void onNext(StringValue stringValue) {
```

```
                System.out.println("stringValue.getValue() = " +
stringValue.getValue());
            }

            @Override
            public void onError(Throwable throwable) {

            }

            @Override
            public void onCompleted() {
                System.out.println("更新完毕");
                countDownLatch.countDown();
            }
        });

 request.onNext(Book.newBuilder().setId("1").setName("a").setAuthor("b")
.build());

 request.onNext(Book.newBuilder().setId("2").setName("c").setAuthor("d")
.build());
        request.onCompleted();
        countDownLatch.await();
    }
}
```

在客户端这块，updateBooks 方法会返回一个 StreamObserver 对象，调用该对象的 onNext 方法就是给服务端传递数据了，可以传递多个数据，调用该对象的 onCompleted 方法就是告诉服务端数据传递结束了，此时也会触发服务端的 onCompleted 方法，服务端的 onCompleted 方法执行之后，进而触发了客户端的 onCompleted 方法。

2.5 双向流 RPC

双向流其实就是 3、4 小节的合体。即客户端多次发送数据，服务端也多次响应数据。

我们先来看下服务端的代码：

```
public class BookServiceImpl extends BookServiceGrpc.BookServiceImplBase
{
    private Map<String, Book> bookMap = new HashMap<>();
    private List<Book> books = new ArrayList<>();

    public BookServiceImpl() {
        Book b1 = Book.newBuilder().setId("1").setName("三国演
义").setAuthor("罗贯中").setPrice(30).addTags("明清小说").addTags("通俗小
说").build();
```

```
        Book b2 = Book.newBuilder().setId("2").setName("西游
记").setAuthor("吴承恩").setPrice(40).addTags("志怪小说").addTags("通俗小
说").build();
        Book b3 = Book.newBuilder().setId("3").setName("水浒
传").setAuthor("施耐庵").setPrice(50).addTags("明清小说").addTags("通俗小
说").build();
        bookMap.put("1", b1);
        bookMap.put("2", b2);
        bookMap.put("3", b3);
    }
    @Override
    public StreamObserver<StringValue>
processBooks(StreamObserver<BookSet> responseObserver) {
        return new StreamObserver<StringValue>() {
            @Override
            public void onNext(StringValue stringValue) {
                Book b =
Book.newBuilder().setId(stringValue.getValue()).build();
                books.add(b);
                if (books.size() == 3) {
                    BookSet bookSet =
BookSet.newBuilder().addAllBookList(books).build();
                    responseObserver.onNext(bookSet);
                    books.clear();
                }
            }

            @Override
            public void onError(Throwable throwable) {

            }

            @Override
            public void onCompleted() {
                BookSet bookSet =
BookSet.newBuilder().addAllBookList(books).build();
                responseObserver.onNext(bookSet);
                books.clear();
                responseObserver.onCompleted();
            }
        };
    }
}
```

这段代码没有实际意义，单纯为了给小伙伴们演示双向流，我的操作逻辑是客户端传递多个 ID 到服务端，然后服务端根据这些 ID 构建对应的 Book 对象，然后三个三个一组，再返回给客户端。客户端每次发送一个请求，都会触发服务端的 onNext 方法，我们在这个方法中对请求分组返回。最后如果还有剩余的请求，我们在 onCompleted() 方法中返回。

再来看看客户端的代码：

```
public class BookServiceClient {
    public static void main(String[] args) throws InterruptedException {
        ManagedChannel channel =
ManagedChannelBuilder.forAddress("localhost", 50051)
                .usePlaintext()
                .build();
        BookServiceGrpc.BookServiceStub stub =
BookServiceGrpc.newStub(channel);
        processBook(stub);
    }

    private static void processBook(BookServiceGrpc.BookServiceStub
stub) throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(1);
        StreamObserver<StringValue> request = stub.processBooks(new
StreamObserver<BookSet>() {
            @Override
            public void onNext(BookSet bookSet) {
                System.out.println("bookSet = " + bookSet);
                System.out.println("============");
            }

            @Override
            public void onError(Throwable throwable) {
            }

            @Override
            public void onCompleted() {
                System.out.println("处理完毕！");
                countDownLatch.countDown();
            }
        });
        request.onNext(StringValue.newBuilder().setValue("a").build());
        request.onNext(StringValue.newBuilder().setValue("b").build());
        request.onNext(StringValue.newBuilder().setValue("c").build());
        request.onNext(StringValue.newBuilder().setValue("d").build());
        request.onCompleted();
        countDownLatch.await();
    }
}
```

这个客户端的代码跟第四小节一模一样，不再赘述了。

好啦，这就是松哥和小伙伴们介绍的 gRPC 的四种不同的通信模式，文章中只给出了一些关键代码，如果小伙伴们没看明白，建议结合上篇文章一起阅读就懂啦～

3 gRPC 中的拦截器

今天我们继续 gRPC 系列。

前面松哥跟大家聊了 gRPC 的简单案例，也说了四种不同的通信模式，感兴趣的小伙伴可以戳这里：

1. 一个简单的案例入门 gRPC
2. 聊一聊 gRPC 的四种通信模式

今天我们来继续聊一聊 gRPC 中的拦截器。

有请求的发送、处理，当然就会有拦截器的需求，例如在服务端通过拦截器统一进行请求认证等操作，这些就需要拦截器来完成，今天松哥先和小伙伴们来聊一聊 gRPC 中拦截器的基本用法，后面我再整一篇文章和小伙伴们做一个基于拦截器实现的 JWT 认证的 gRPC。

gRPC 中的拦截器整体上来说可以分为两大类：

1. 服务端拦截器
2. 客户端拦截器

我们分别来看。

3.1 服务端拦截器

服务端拦截器的作用有点像我们 Java 中的 Filter，服务端拦截器又可以继续细分为**一元拦截器**和**流拦截器**。

一元拦截器对应我们上篇文章中所讲的一元 RPC，也就是一次请求，一次响应这种情况。

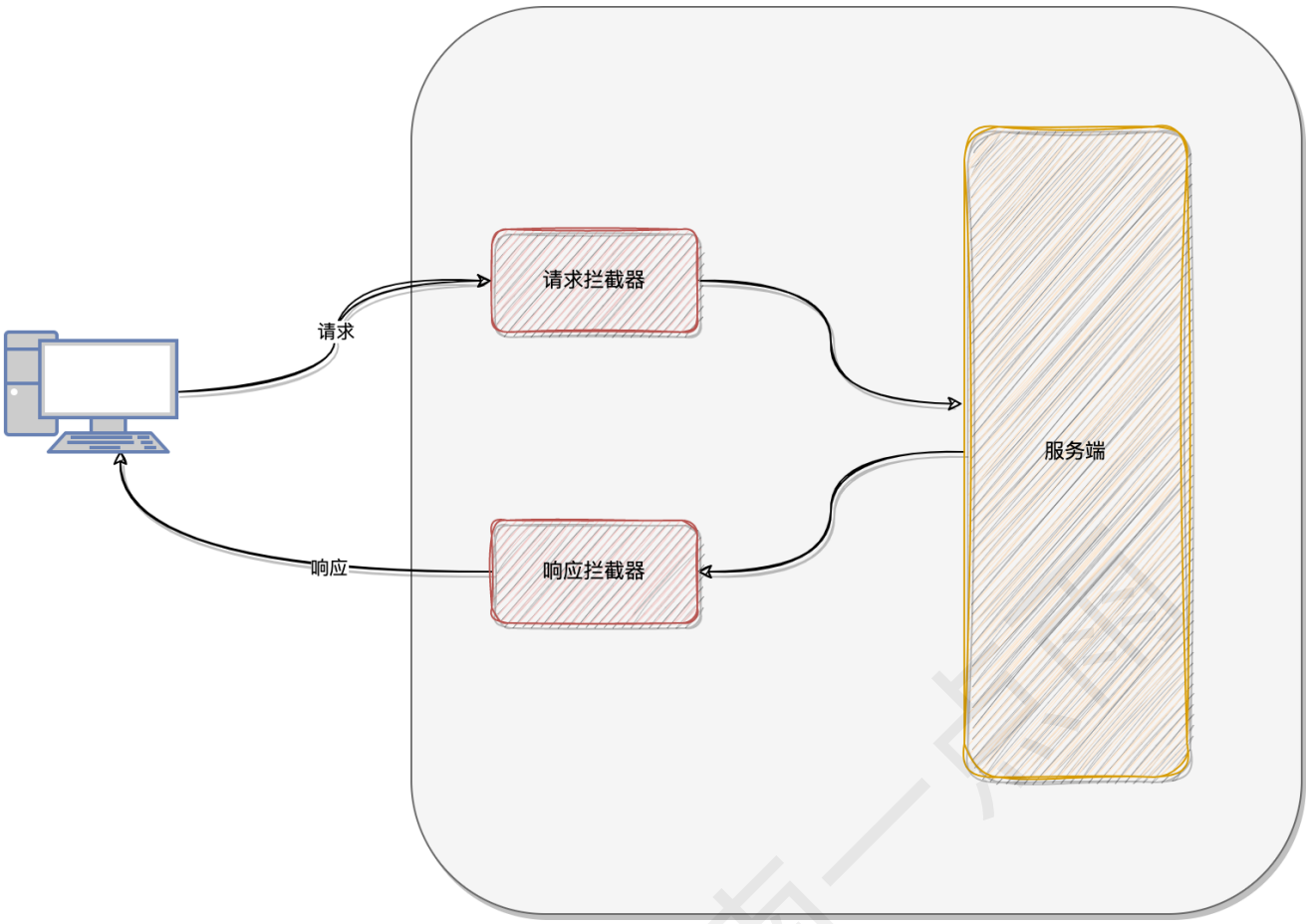
流拦截器则对应我们上篇文章中所讲的服务端流 RPC、客户端流 RPC 以及双向流 RPC。

不过，在 Java 代码中，无论是一元拦截器还是流拦截器，代码其实都是一样的。不过如果你是用 Go 实现的 gRPC，那么这块是不一样的。

所以接下来的内容我就不去区分一元拦截器和流拦截器了，我们直接来看一个服务端拦截器的例子。

这里我就不从头开始写了，我们直接在上篇文章的基础之上继续添加拦截器即可。

服务端拦截器工作位置大致如下：



从这张图中小伙伴们可以看到，我们可以在服务端处理请求之前将请求拦截下来，统一进行权限校验等操作，也可以在服务端将请求处理完毕之后，准备响应的时候将响应拦截下来，可以对响应进行二次处理。

首先我们来看请求拦截器，实际上是一个监听器：

```
public class BookServiceCallListener<R> extends
ForwardingServerCallListener<R> {
    private final ServerCall.Listener<R> delegate;

    public BookServiceCallListener(ServerCall.Listener<R> delegate) {
        this.delegate = delegate;
    }

    @Override
    protected ServerCall.Listener<R> delegate() {
        return delegate;
    }

    @Override
    public void onMessage(R message) {
        System.out.println("这是客户端发来的消息，可以在这里进行预处
理："+message);
        super.onMessage(message);
    }
}
```

这里我们自定义一个类，继承自 ForwardingServerCallListener 类，在这里重写 onMessage 方法，当有请求到达的时候，就会经过这里的 onMessage 方法。如果我们需要对传入的参数进行验证等操作，就可以在这里完成。

再来看看响应拦截器：

```
public class BookServiceCall<ReqT,RespT> extends
ForwardingServerCall.SimpleForwardingServerCall<ReqT,RespT> {
    protected BookServiceCall(ServerCall<ReqT, RespT> delegate) {
        super(delegate);
    }

    @Override
    protected ServerCall<ReqT, RespT> delegate() {
        return super.delegate();
    }

    @Override
    public MethodDescriptor<ReqT, RespT> getMethodDescriptor() {
        return super.getMethodDescriptor();
    }

    @Override
    public void sendMessage(RespT message) {
        System.out.println("这是服务端返回给客户端的消息："+message);
        super.sendMessage(message);
    }
}
```

小伙伴们可能发现了，我这里用到了很多泛型，请求类型和响应类型都不建议指定具体类型，因为拦截器可能会拦截多种类型的请求，请求参数和响应的数据类型都不一定一样。

这里是重写 sendMessage 方法，在这个方法中我们可以对服务端准备返回给客户端的消息进行预处理。

所以这个位置就相当于**响应拦截器**。

最后，我们需要在启动服务的时候，将这两个拦截器配置进去，代码如下：

```
public void start() throws IOException {
    int port = 50051;
    server = ServerBuilder.forPort(port)
            .addService(ServerInterceptors.intercept(new
BookServiceImpl(), new ServerInterceptor() {
                @Override
                public <ReqT, RespT> ServerCall.Listener<ReqT>
interceptCall(ServerCall<ReqT, RespT> call, Metadata headers,
ServerCallHandler<ReqT, RespT> next) {
```

```
                    String fullMethodName =
call.getMethodDescriptor().getFullMethodName();
                    System.out.println(fullMethodName + ":pre");
                    Set<String> keys = headers.keys();
                    for (String key : keys) {
                        System.out.println(key + ">>>" +
headers.get(Metadata.Key.of(key, ASCII_STRING_MARSHALLER)));
                    }
                    return new BookServiceCallListener<>
(next.startCall(new BookServiceCall(call), headers));
                }
            }))
            .build()
            .start();
    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        BookServiceServer.this.stop();
    }));
}
```

这是我之前服务启动的方法，以前我们调用 addService 方法的时候，直接添加对应的服务就可以了，现在，我们除了添加之前的 BookServiceImpl 服务之外，还额外给了一个拦截器。

每当请求到达的时候，就会经过拦截器的 interceptCall 方法，这个方法有三个参数：

- 第一个参数 call 是消费传入的 RPC 消息的一个回调。
- 第二个参数 headers 则是请求的消息头，如果我们通过 JWT 进行请求校验，那么就从这个 headers 中提取出请求的 JWT 令牌然后进行校验。
- 第三个参数 next 就类似于我们在 Java 过滤器 filter 中的 filterChain 一样，让这个请求继续向下走。

在这个方法中，我们请求头的信息都打印出来给小伙伴们参考了。然后在返回值中，将我们刚刚写的请求拦截器和响应拦截器构建并返回。

好啦，这样我们的服务端拦截器就搞好啦～无论是一元的 RPC 消息还是流式的 RPC 消息，都会经过这个拦截器，响应也是一样。

3.2 客户端拦截器

客户端拦截器就比较简单了，客户端拦截器可以将我们的请求拦截下来，例如我们如果想为所有请求添加统一的令牌 Token，那么就可以在这里来做，方式如下：

```
ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost",
50051)
        .usePlaintext()
        .intercept(new ClientInterceptor() {
            @Override
            public <ReqT, RespT> ClientCall<ReqT, RespT>
interceptCall(MethodDescriptor<ReqT, RespT> method, CallOptions
callOptions, Channel next) {
                System.out.println("!!!!!!!!!!!!!!!!");
                callOptions = callOptions.withAuthority("javaboy");
                return next.newCall(method,callOptions);
            }
        })
        .build();
BookServiceGrpc.BookServiceStub stub = BookServiceGrpc.newStub(channel);
```

当我们的请求执行的时候，这个客户端拦截器就会被触发。

3.3 小结

好啦，今天就和小伙伴们简单介绍一下服务端拦截器和客户端拦截器。下篇文章，松哥会通过一个 JWT 认证来和小伙伴们演示这个拦截器的具体用法。

4 gRPC + JWT

上篇文章松哥和小伙伴们聊了在 gRPC 中如何使用拦截器，这些拦截器有服务端拦截器也有客户端拦截器，这些拦截器的一个重要使用场景，就是可以进行身份的校验。当客户端发起请求的时候，服务端通过拦截器进行身份校验，就知道这个请求是谁发起的了。今天松哥就来通过一个具体的案例，来和小伙伴们演示一下 gRPC 如何结合 JWT 进行身份校验。

4.1 JWT 介绍

4.1.1 无状态登录

4.1.1.1 什么是有状态

有状态服务，即服务端需要记录每次会话的客户端信息，从而识别客户端身份，根据用户身份进行请求的处理，典型的设计如 Tomcat 中的 Session。例如登录：用户登录后，我们把用户的信息保存在服务端 session 中，并且给用户一个 cookie 值，记录对应的 session，然后下次请求，用户携带 cookie 值来（这一步有浏览器自动完成），我们就能识别到对应 session，从而找到用户的信息。这种方式目前来看最方便，但是也有一些缺陷，如下：

- 服务端保存大量数据，增加服务端压力
- 服务端保存用户状态，不支持集群化部署

4.1.1.2 什么是无状态

微服务集群中的每个服务，对外提供的都使用 RESTful 风格的接口。而 RESTful 风格的一个最重要的规范就是：服务的无状态性，即：

- 服务端不保存任何客户端请求者信息
- 客户端的每次请求必须具备自描述信息，通过这些信息识别客户端身份

那么这种无状态性有哪些好处呢？

- 客户端请求不依赖服务端的信息，多次请求不需要必须访问到同一台服务器
- 服务端的集群和状态对客户端透明
- 服务端可以任意的迁移和伸缩（可以方便的进行集群化部署）
- 减小服务端存储压力

4.1.2 如何实现无状态

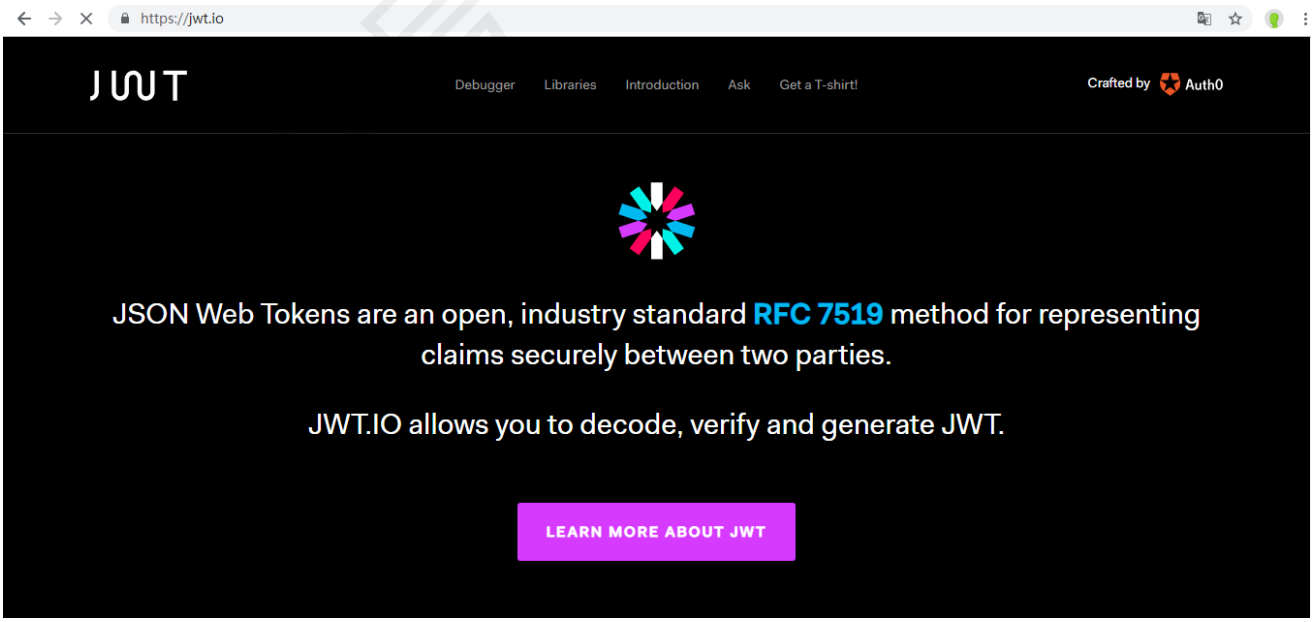
无状态登录的流程：

- 首先客户端发送账户名/密码到服务端进行认证
- 认证通过后，服务端将用户信息加密并且编码成一个 token，返回给客户端
- 以后客户端每次发送请求，都需要携带认证的 token
- 服务端对客户端发送来的 token 进行解密，判断是否有效，并且获取用户登录信息

4.1.3 JWT

4.1.3.1 简介

JWT，全称是 Json Web Token， 是一种 JSON 风格的轻量级的授权和身份认证规范，可实现无状态、分布式的 Web 应用授权：



JWT 作为一种规范，并没有和某一种语言绑定在一起，常用的 Java 实现是 GitHub 上的开源项目 jjwt，地址如下：`https://github.com/jwtk/jjwt`

4.1.3.2 JWT数据格式

JWT 包含三部分数据：

- Header：头部，通常头部有两部分信息：
  - 声明类型，这里是JWT
  - 加密算法，自定义

我们会对头部进行 Base64Url 编码（可解码），得到第一部分数据。

- Payload：载荷，就是有效数据，在官方文档中(RFC7519)，这里给了7个示例信息：
  - iss (issuer)：表示签发人
  - exp (expiration time)：表示token过期时间
  - sub (subject)：主题
  - aud (audience)：受众
  - nbf (Not Before)：生效时间
  - iat (Issued At)：签发时间
  - jti (JWT ID)：编号

这部分也会采用 Base64Url 编码，得到第二部分数据。

- Signature：签名，是整个数据的认证信息。一般根据前两步的数据，再加上服务的的密钥secret（密钥保存在服务端，不能泄露给客户端），通过 Header 中配置的加密算法生成。用于验证整个数据完整和可靠性。

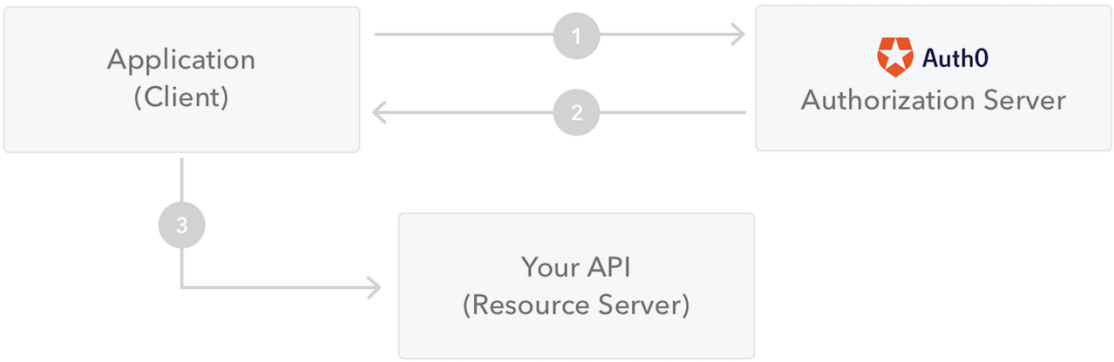
生成的数据格式如下图：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

注意，这里的数据通过 . 隔开成了三部分，分别对应前面提到的三部分，另外，这里数据是不换行的，图片换行只是为了展示方便而已。

4.1.3.3 JWT 交互流程

流程图：



步骤翻译：

1. 应用程序或客户端向授权服务器请求授权
2. 获取到授权后，授权服务器会向应用程序返回访问令牌
3. 应用程序使用访问令牌来访问受保护资源（如 API）

因为 JWT 签发的 token 中已经包含了用户的身份信息，并且每次请求都会携带，这样服务的就无需保存用户信息，甚至无需去数据库查询，这样就完全符合了 RESTful 的无状态规范。

4.1.3.4 JWT 存在的问题

说了这么多，JWT 也不是天衣无缝，由客户端维护登录状态带来的一些问题在这里依然存在，举例如下：

1. 续签问题，这是被很多人诟病的问题之一，传统的 cookie+session 的方案天然的支持续签，但是 jwt 由于服务端不保存用户状态，因此很难完美解决续签问题，如果引入 redis，虽然可以解决问题，但是 jwt 也变得不伦不类了。
2. 注销问题，由于服务端不再保存用户信息，所以一般可以通过修改 secret 来实现注销，服务端 secret 修改后，已经颁发的未过期的 token 就会认证失败，进而实现注销，不过毕竟没有传统的注销方便。
3. 密码重置，密码重置后，原本的 token 依然可以访问系统，这时候也需要强制修改 secret。
4. 基于第 2 点和第 3 点，一般建议不同用户取不同 secret。

> *当然，为了解决 JWT 存在的问题，也可以将 JWT 结合 Redis 来用，服务端生成的 JWT 字符串存入到 Redis 中并设置过期时间，每次校验的时候，先看 Redis 中是否存在该 JWT 字符串，如果存在就进行后续的校验。但是这种方式有点不伦不类（又成了有状态了）。*

4.2 实践

我们来看下 gRPC 如何结合 JWT。

4.2.1 项目创建

首先我先给大家看下我的项目结构：

```
├── grpc_api
│   ├── pom.xml
│   └── src
├── grpc_client
│   ├── pom.xml
│   └── src
├── grpc_server
│   ├── pom.xml
│   └── src
└── pom.xml
```

还是跟之前文章中的一样，三个模块，grpc_api 用来存放一些公共的代码。

grpc_server 用来放服务端的代码，我这里服务端主要提供了两个接口：

1. 登录接口，登录成功之后返回 JWT 字符串。
2. hello 接口，客户端拿着 JWT 字符串来访问 hello 接口。

grpc_client 则是我的客户端代码。

4.2.2 grpc_api

我将 protocol buffers 和一些依赖都放在 grpc_api 模块中，因为将来我的 grpc_server 和 grpc_client 都将依赖 grpc_api。

我们来看下这里需要的依赖和插件：

```
<dependencies>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-api</artifactId>
        <version>0.11.5</version>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-impl</artifactId>
        <version>0.11.5</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-jackson</artifactId>
```

```
        <version>0.11.5</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-netty-shaded</artifactId>
        <version>1.52.1</version>
    </dependency>
    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-protobuf</artifactId>
        <version>1.52.1</version>
    </dependency>
    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-stub</artifactId>
        <version>1.52.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat</groupId>
        <artifactId>annotations-api</artifactId>
        <version>6.0.53</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
<build>
    <extensions>
        <extension>
            <groupId>kr.motd.maven</groupId>
            <artifactId>os-maven-plugin</artifactId>
            <version>1.6.2</version>
        </extension>
    </extensions>
    <plugins>
        <plugin>
            <groupId>org.xolstice.maven.plugins</groupId>
            <artifactId>protobuf-maven-plugin</artifactId>
            <version>0.6.1</version>
            <configuration>

 <protocArtifact>com.google.protobuf:protoc:3.21.7:exe:${os.detected.cla
ssifier}</protocArtifact>
                <pluginId>grpc-java</pluginId>
                <pluginArtifact>io.grpc:protoc-gen-grpc-
java:1.51.0:exe:${os.detected.classifier}</pluginArtifact>
            </configuration>
            <executions>
                <execution>
                    <goals>
```

```
                        <goal>compile</goal>
                        <goal>compile-custom</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

这里的依赖和插件松哥在本系列的第一篇文章中都已经介绍过了，唯一不同的是，这里引入了 JWT 插件，JWT 我使用了比较流行的 JJWT 这个工具。JJWT 松哥在之前的文章和视频中也都有介绍过，这里就不再啰嗦了。

先来看看我的 Protocol Buffers 文件：

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.javaboy.grpc.api";
option java_outer_classname = "LoginProto";
import "google/protobuf/wrappers.proto";

package login;

service LoginService {
  rpc login (LoginBody) returns (LoginResponse);
}

service HelloService{
  rpc sayHello(google.protobuf.StringValue) returns
(google.protobuf.StringValue);
}

message LoginBody {
  string username = 1;
  string password = 2;
}

message LoginResponse {
  string token = 1;
}
```

经过前面几篇文章的介绍，这里我就不多说啦，就是定义了两个服务：

- LoginService：这个登录服务，传入用户名密码，返回登录成功之后的令牌。
- HelloService：这个就是一个打招呼的服务，传入字符串，返回也是字符串。

定义完成之后，生成对应的代码即可。

接下来再定义一个常量类供 grpc_server 和 grcp_client 使用，如下：

```
public interface AuthConstant {
    SecretKey JWT_KEY =
Keys.hmacShaKeyFor("hello_javaboy_hello_javaboy_hello_javaboy_hello_java
boy_".getBytes());
    Context.Key<String> AUTH_CLIENT_ID = Context.key("clientId");
    String AUTH_HEADER = "Authorization";
    String AUTH_TOKEN_TYPE = "Bearer";
}
```

这里的每个常量我都给大家解释下：

1. JWT_KEY：这个是生成 JWT 字符串以及进行 JWT 字符串校验的密钥。
2. AUTH_CLIENT_ID：这个是客户端的 ID，即客户端发送来的请求携带了 JWT 字符串，通过 JWT 字符串确认了用户身份，就存在这个变量中。
3. AUTH_HEADER：这个是携带 JWT 字符串的请求头的 KEY。
4. AUTH_TOKEN_TYPE：这个是携带 JWT 字符串的请求头的参数前缀，通过这个可以确认参数的类型，常见取值有 Bearer 和 Basic。

如此，我们的 gRPC_api 就定义好了。

4.2.3 grpc_server

接下来我们来定义 gRPC_server。

首先来定义登录服务：

```
public class LoginServiceImpl extends
LoginServiceGrpc.LoginServiceImplBase {
    @Override
    public void login(LoginBody request, StreamObserver<LoginResponse>
responseObserver) {
        String username = request.getUsername();
        String password = request.getPassword();
        if ("javaboy".equals(username) && "123".equals(password)) {
            System.out.println("login success");
            //登录成功
            String jwtToken =
Jwts.builder().setSubject(username).signWith(AuthConstant.JWT_KEY).compa
ct();

 responseObserver.onNext(LoginResponse.newBuilder().setToken(jwtToken).b
uild());
            responseObserver.onCompleted();
        }else{
            System.out.println("login error");
            //登录失败
```

```
 responseObserver.onNext(LoginResponse.newBuilder().setToken("login
error").build());
            responseObserver.onCompleted();
        }
    }
}
```

省事起见，我这里没有连接数据库，用户名和密码固定为 javaboy 和 123。

登录成功之后，就生成一个 JWT 字符串返回。

登录失败，就返回一个 login error 字符串。

再来看我们的 HelloService 服务，如下：

```
public class HelloServiceImpl extends
HelloServiceGrpc.HelloServiceImplBase {
    @Override
    public void sayHello(StringValue request,
StreamObserver<StringValue> responseObserver) {
        String clientId = AuthConstant.AUTH_CLIENT_ID.get();

 responseObserver.onNext(StringValue.newBuilder().setValue(clientId + "
say hello:" + request.getValue()).build());
        responseObserver.onCompleted();
    }
}
```

这个服务就更简单了，不啰嗦。唯一值得说的是 `AuthConstant.AUTH_CLIENT_ID.get();` 表示获取当前访问用户的 ID，这个用户 ID 是在拦截器中存入进来的。

最后，我们来看服务端比较重要的拦截器，我们要在拦截器中从请求头中获取到 JWT 令牌并解析，如下：

```
public class AuthInterceptor implements ServerInterceptor {
    private JwtParser parser =
Jwts.parser().setSigningKey(AuthConstant.JWT_KEY);

    @Override
    public <ReqT, RespT> ServerCall.Listener<ReqT>
interceptCall(ServerCall<ReqT, RespT> serverCall, Metadata metadata,
ServerCallHandler<ReqT, RespT> serverCallHandler) {
        String authorization =
metadata.get(Metadata.Key.of(AuthConstant.AUTH_HEADER,
Metadata.ASCII_STRING_MARSHALLER));
        Status status = Status.OK;
        if (authorization == null) {
```

```
            status = Status.UNAUTHENTICATED.withDescription("miss
authentication token");
        } else if
(!authorization.startsWith(AuthConstant.AUTH_TOKEN_TYPE)) {
            status = Status.UNAUTHENTICATED.withDescription("unknown
token type");
        } else {
            Jws<Claims> claims = null;
            String token =
authorization.substring(AuthConstant.AUTH_TOKEN_TYPE.length()).trim();
            try {
                claims = parser.parseClaimsJws(token);
            } catch (JwtException e) {
                status =
Status.UNAUTHENTICATED.withDescription(e.getMessage()).withCause(e);
            }
            if (claims != null) {
                Context ctx = Context.current()
                        .withValue(AuthConstant.AUTH_CLIENT_ID,
claims.getBody().getSubject());
                return Contexts.interceptCall(ctx, serverCall, metadata,
serverCallHandler);
            }
        }
        serverCall.close(status, new Metadata());
        return new ServerCall.Listener<ReqT>() {
        };
    }
}
```

这段代码逻辑应该好理解：

1. 首先从 Metadata 中提取出当前请求所携带的 JWT 字符串（相当于从请求头中提取出来）。
2. 如果第一步提取到的值为 null 或者这个值不是以指定字符 Bearer 开始的，说明这个令牌是一个非法令牌，设置对应的响应 status 即可。
3. 如果令牌都没有问题的话，接下来就进行令牌的校验，校验失败，则设置相应的 status 即可。
4. 校验成功的话，我们就会获取到一个 Jws 对象，从这个对象中我们可以提取出来用户名，并存入到 Context 中，将来我们在 HelloServiceImpl 中就可以获取到这里的用户名了。
5. 最后，登录成功的话，`Contexts.interceptCall` 方法构建监听器并返回；登录失败，则构建一个空的监听器返回。

最后，我们再来看看启动服务端：

```
public class LoginServer {
    Server server;
```

```
    public static void main(String[] args) throws IOException,
InterruptedException {
        LoginServer server = new LoginServer();
        server.start();
        server.blockUntilShutdown();
    }

    public void start() throws IOException {
        int port = 50051;
        server = ServerBuilder.forPort(port)
                .addService(new LoginServiceImpl())
                .addService(ServerInterceptors.intercept(new
HelloServiceImpl(), new AuthInterceptor()))
                .build()
                .start();
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            LoginServer.this.stop();
        }));
    }

    private void stop() {
        if (server != null) {
            server.shutdown();
        }
    }

    private void blockUntilShutdown() throws InterruptedException {
        if (server != null) {
            server.awaitTermination();
        }
    }
}
```

这个跟之前的相比就多加了一个 Service，添加 HelloServiceImpl 服务的时候，多加了一个拦截器，换言之，登录的时候，请求是不会被这个认证拦截器拦截的。

好啦，这样我们的 grpc_server 就开发完成了。

4.2.4 grpc_client

接下来我们来看 grpc_client。

先来看登录：

```
public class LoginClient {
    public static void main(String[] args) throws InterruptedException {
        ManagedChannel channel =
ManagedChannelBuilder.forAddress("localhost", 50051)
                .usePlaintext()
                .build();
```

```
        LoginServiceGrpc.LoginServiceStub stub =
LoginServiceGrpc.newStub(channel);
        login(stub);
    }

    private static void login(LoginServiceGrpc.LoginServiceStub stub)
throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(1);

 stub.login(LoginBody.newBuilder().setUsername("javaboy").setPassword("1
23").build(), new StreamObserver<LoginResponse>() {
            @Override
            public void onNext(LoginResponse loginResponse) {
                System.out.println("loginResponse.getToken() = " +
loginResponse.getToken());
            }

            @Override
            public void onError(Throwable throwable) {

            }

            @Override
            public void onCompleted() {
                countDownLatch.countDown();
            }
        });
        countDownLatch.await();
    }
}
```

这个方法直接调用就行了，看过前面几篇 gRPC 文章的话，这里都很好理解。

再来看 hello 接口的调用，这个接口调用需要携带 JWT 字符串，而携带 JWT 字符串，则需要我们构建一个 CallCredentials 对象，如下：

```
public class JwtCredential extends CallCredentials {
    private String subject;

    public JwtCredential(String subject) {
        this.subject = subject;
    }

    @Override
    public void applyRequestMetadata(RequestInfo requestInfo, Executor
executor, MetadataApplier metadataApplier) {
        executor.execute(() -> {
            try {
                Metadata headers = new Metadata();
```

```
                headers.put(Metadata.Key.of(AuthConstant.AUTH_HEADER,
Metadata.ASCII_STRING_MARSHALLER),
                        String.format("%s %s",
AuthConstant.AUTH_TOKEN_TYPE, subject));
                metadataApplier.apply(headers);
            } catch (Throwable e) {

 metadataApplier.fail(Status.UNAUTHENTICATED.withCause(e));
            }
        });
    }

    @Override
    public void thisUsesUnstableApi() {

    }
}
```

这里就是将请求的 JWT 令牌放入到请求头中即可。

最后来看看调用：

```
public class LoginClient {
    public static void main(String[] args) throws InterruptedException {
        ManagedChannel channel =
ManagedChannelBuilder.forAddress("localhost", 50051)
                .usePlaintext()
                .build();
        LoginServiceGrpc.LoginServiceStub stub =
LoginServiceGrpc.newStub(channel);
        sayHello(channel);
    }

    private static void sayHello(ManagedChannel channel) throws
InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(1);
        HelloServiceGrpc.HelloServiceStub helloServiceStub =
HelloServiceGrpc.newStub(channel);
        helloServiceStub
                .withCallCredentials(new
JwtCredential("eyJhbGciOiJIUzM4NCJ9.eyJzdWIiOiJqYXZhYm95In0.IMMp7oh1dl_t
rUn7sn8qiv9GtO-COQyCGDz_Yy8VI4fIqUcRfwQddP45IoxNovxL"))

.sayHello(StringValue.newBuilder().setValue("wangwu").build(), new
StreamObserver<StringValue>() {
            @Override
            public void onNext(StringValue stringValue) {
                System.out.println("stringValue.getValue() = " +
stringValue.getValue());
```

```
            }

            @Override
            public void onError(Throwable throwable) {
                System.out.println("throwable.getMessage() = " +
throwable.getMessage());
            }

            @Override
            public void onCompleted() {
                countDownLatch.countDown();
            }
        });
        countDownLatch.await();
    }
}
```

这里的登录令牌就是前面调用 login 方法时获取到的令牌。

好啦，大功告成。

4.3 小结

上面的登录与校验只是松哥给小伙伴们展示的一个具体案例而已，在此案例基础之上，我们还可以扩展出来更多写法，但是万变不离其宗，其他玩法就需要小伙伴们自行探索啦～

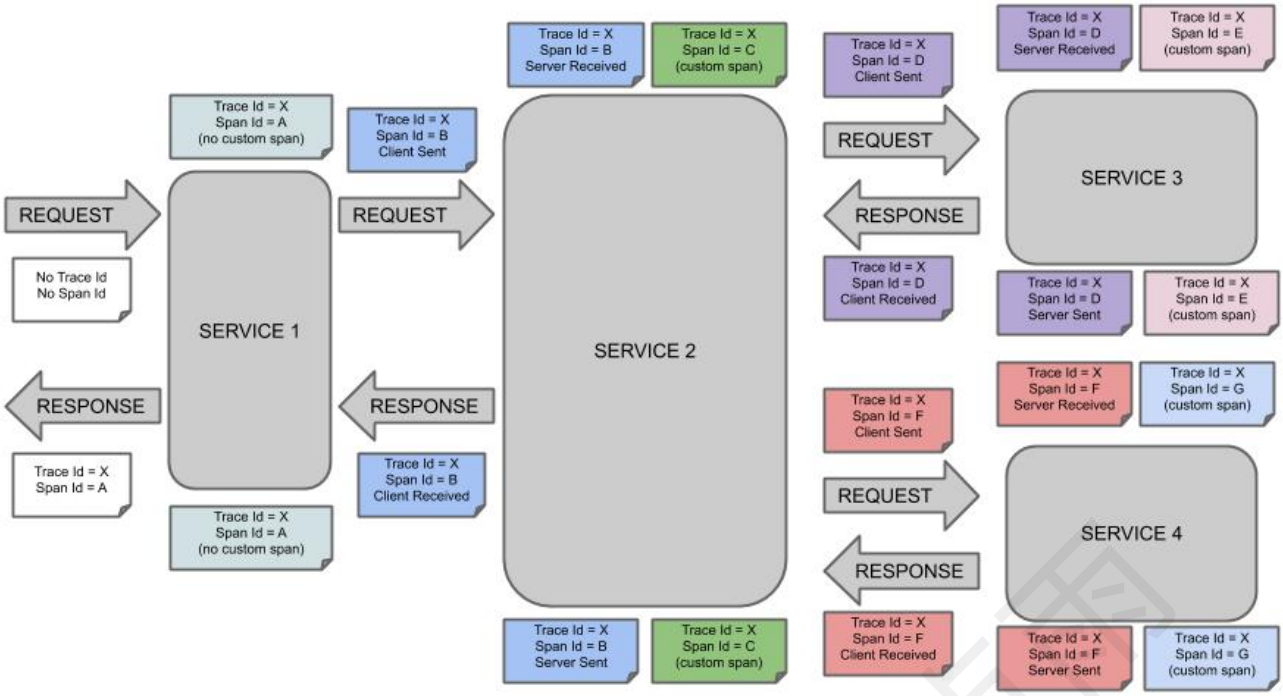
5 gRPC 请求截止时间

在 HTTP 请求中，我们发送请求的时候，可以设置一个请求超时时间-connectTimeout，即在指定的时间内，如果请求没有到达服务端，为了避免客户端一直进行不必要的等待，就会抛出一个请求超时异常。

但是在微服务系统中，我们却很少设置请求超时时间，一般都是用另外一个概念代替，那就是请求截止时间。

这是什么原因呢？今天我们就来简单聊一聊这个话题。

在微服务中我们客户端的请求在服务端往往会有比较复杂的链条，我想起来 Spring Cloud Sleuth 官方给的一个请求链路追踪的图，我们直接拿来看下：



这张图中，请求从客户端发起之后，在服务端一共经历了四个 SERVICE，对于这样的请求，如果我们还是按照之前发送普通 HTTP 请求的方式，设置一个 connectTimeout 显然是不够的。

我举个例子：

假设我们发送一个请求，为该请求设置 connectTimeout 为 5s，那么这个时间只对第一个服务 SERVICE1 有效，也就是请求在 5s 之内没有到达 SERVICE1，那么就会抛出连接超时异常；请求如果在 5s 之内到达 SERVICE1，那么就不会抛出异常，但是！！！，请求到达 SERVICE1 并不意味着请求结束，后面从 SERVICE1 到 SERVICE2，从 SERVICE2 到 SERVICE3，从 SERVICE3 到 SERVICE4，还有四个 HTTP 请求待处理，这些请求超时了怎么办？很明显，connectTimeout 属性对于后面几个请求就鞭长莫及了。

所以，对于这种场景，我们一般使用截止时间来处理。

截止时间相当于设置**整个请求生命周期**的时间，也就是这个请求，我要多久拿到结果。很明显，这个时间应该在客户端发起请求的时候设置。

gRPC 中提供了对应的方法，我们可以非常方便的设置请求的截止时间 DeadLineTime，如下：

```
public class LoginClient {
    public static void main(String[] args) throws InterruptedException {
        ManagedChannel channel =
ManagedChannelBuilder.forAddress("localhost", 50051)
                .usePlaintext()
                .build();
        LoginServiceGrpc.LoginServiceStub stub =
LoginServiceGrpc.newStub(channel).withDeadline(Deadline.after(3,
TimeUnit.SECONDS));
        login(stub);
    }
```

```
    private static void login(LoginServiceGrpc.LoginServiceStub stub)
throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(1);

 stub.login(LoginBody.newBuilder().setUsername("javaboy").setPassword("1
23").build(), new StreamObserver<LoginResponse>() {
            @Override
            public void onNext(LoginResponse loginResponse) {
                System.out.println("loginResponse.getToken() = " +
loginResponse.getToken());
            }

            @Override
            public void onError(Throwable throwable) {
                System.out.println("throwable = " + throwable);
            }

            @Override
            public void onCompleted() {
                countDownLatch.countDown();
            }
        });
        countDownLatch.await();
    }
}
```

服务端通过 Thread.sleep 做个简单的休眠就行了，超时之后，客户端的 onError 方法会被触发，抛出如下异常：

```
throwable = io.grpc.StatusRuntimeException: DEADLINE_EXCEEDED: deadline
exceeded after 2.939621462s. [closed=[], open=
[[buffered_nanos=285550823, remote_addr=localhost/127.0.0.1:50051]]]
```

好啦，一个简单的小细节，感兴趣的小伙伴不妨去试试啦～

6 gRPC 异常处理

今天来和小伙伴们聊一聊该如何处理 gRPC 中遇到的异常。

在之前的几篇文章中，其实我们也遇到过异常问题，只是当时没有和小伙伴们细说，只是囫囵吞枣写了一个案例而已，今天我们就来把这个话题跟小伙伴们仔细捋一捋。

我们之前写过一个登录的案例，在之前的案例中，如果用户在登录时输入了错误的用户名密码的话，那么我们是通过一个普通的数据流返回异常信息，其实，对于异常信息，我们可以通过专门的异常通道来写回到客户端。

6.1 服务端处理异常

先来看看服务端如何处理异常。

还是以我们之前的 gRPC 登录案例为例，我们修改服务端的登录逻辑如下（完整代码小伙伴们可以参考之前的 手把手教大家在 gRPC 中使用 JWT 完成身份校验 一文）：

```
public class LoginServiceImpl extends
LoginServiceGrpc.LoginServiceImplBase {
    @Override
    public void login(LoginBody request, StreamObserver<LoginResponse>
responseObserver) {
        String username = request.getUsername();
        String password = request.getPassword();
        if ("javaboy".equals(username) && "123".equals(password)) {
            System.out.println("login success");
            //登录成功
            String jwtToken =
Jwts.builder().setSubject(username).signWith(AuthConstant.JWT_KEY).compa
ct();

 responseObserver.onNext(LoginResponse.newBuilder().setToken(jwtToken).b
uild());
            responseObserver.onCompleted();
        }else{
            System.out.println("login error");
            //登录失败

 responseObserver.onError(Status.UNAUTHENTICATED.withDescription("login
error").asException());
        }
    }
}
```

小伙伴们看到，在登录失败时我们通过 `responseObserver.onError` 方法将异常信息写回到客户端。这个方法的参数是一个 Throwable 对象，对于这个对象，在 Status 这个枚举类中定义了一些常见的值，分别如下：

- OK(0)：请求成功。
- CANCELLED(1)：操作被取消。
- UNKNOWN(2)：未知错误。
- INVALID_ARGUMENT(3)：客户端给了无效的请求参数。
- DEADLINE_EXCEEDED(4)：请求超过了截止时间。
- NOT_FOUND(5)：请求资源未找到。
- ALREADY_EXISTS(6)：添加的内容已经存在。
- PERMISSION_DENIED(7)：请求权限不足。
- RESOURCE_EXHAUSTED(8)：资源耗尽。
- FAILED_PRECONDITION(9)：服务端上为准备好。
- ABORTED(10)：请求被中止。

- OUT_OF_RANGE(11)：请求超出范围。
- UNIMPLEMENTED(12)：未实现的操作。
- INTERNAL(13)：服务内部错误。
- UNAVAILABLE(14)：服务不可用。
- DATA_LOSS(15)：数据丢失或者损毁。
- UNAUTHENTICATED(16)：请求未认证。

系统默认给出的请求类型大致上就这些。当然，如果这些并不能满足你的需求，我们也可以扩展这个枚举类。

6.2 客户端处理异常

当服务端给出异常信息之后，客户端的处理分为两种情况。

6.2.1 异步请求

如果客户端是异步请求，则直接在异常回调中处理即可，如下：

```
public class LoginClient {
    public static void main(String[] args) throws InterruptedException {
        ManagedChannel channel =
ManagedChannelBuilder.forAddress("localhost", 50051)
                .usePlaintext()
                .build();
        LoginServiceGrpc.LoginServiceStub stub =
LoginServiceGrpc.newStub(channel).withDeadline(Deadline.after(3,
TimeUnit.SECONDS));
        login(stub);
    }
    private static void login(LoginServiceGrpc.LoginServiceStub stub)
throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(1);

 stub.login(LoginBody.newBuilder().setUsername("javaboy").setPassword("1
234").build(), new StreamObserver<LoginResponse>() {
            @Override
            public void onNext(LoginResponse loginResponse) {
                System.out.println("loginResponse.getToken() = " +
loginResponse.getToken());
            }

            @Override
            public void onError(Throwable throwable) {
                System.out.println("throwable = " + throwable);
            }

            @Override
            public void onCompleted() {
                countDownLatch.countDown();
```

```
            }
        });
        countDownLatch.await();
    }
}
```

小伙伴们看到，直接在 onError 回到中处理异常即可。

6.2.2 同步请求

如果客户端请求是同步阻塞请求，那么就要通过异常捕获的方式获取服务端返回的异常信息了，如下：

```
public class LoginClient2 {
    public static void main(String[] args) throws InterruptedException {
        ManagedChannel channel =
ManagedChannelBuilder.forAddress("localhost", 50051)
                .usePlaintext()
                .build();
        LoginServiceGrpc.LoginServiceBlockingStub stub =
LoginServiceGrpc.newBlockingStub(channel).withDeadline(Deadline.after(3,
TimeUnit.SECONDS));
        login(stub);
    }

    private static void login(LoginServiceGrpc.LoginServiceBlockingStub
stub) throws InterruptedException {
        try {
            LoginResponse resp =
stub.login(LoginBody.newBuilder().setUsername("javaboy").setPassword("12
34").build());
            System.out.println("resp.getToken() = " + resp.getToken());
        } catch (Exception e) {
            System.out.println("e.getMessage() = " + e.getMessage());
        }
    }
}
```

同步阻塞请求就通过异常捕获去获取服务端返回的异常信息即可。

6.3 题外话

最后，再来和小伙伴们说一个提高 gRPC 数据传输效率的小技巧，那就是传输的数据可以使用 gzip 进行压缩。

具体处理方式就是在客户端调用 `withCompression` 方法指定数据压缩，如下：

```
public class LoginClient2 {
```

```
    public static void main(String[] args) throws InterruptedException {
        ManagedChannel channel =
ManagedChannelBuilder.forAddress("localhost", 50051)
                .usePlaintext()
                .build();
        LoginServiceGrpc.LoginServiceBlockingStub stub =
LoginServiceGrpc.newBlockingStub(channel).withDeadline(Deadline.after(3,
TimeUnit.SECONDS));
        login(stub);
    }

    private static void login(LoginServiceGrpc.LoginServiceBlockingStub
stub) throws InterruptedException {
        try {
            LoginResponse resp =
stub.withCompression("gzip").login(LoginBody.newBuilder().setUsername("j
avaboy").setPassword("123").build());
            System.out.println("resp.getToken() = " + resp.getToken());
        } catch (Exception e) {
            System.out.println("e.getMessage() = " + e.getMessage());
        }
    }
}
```

好啦，一个关于 gRPC 的小小知识点～

7 TLS、CA 证书

松哥最近在和小伙伴们连载 gRPC，如何确保 gRPC 通信的安全性？这就涉及到 TSL 了，但是考虑到可能有小伙伴对加密连接这一整套方案比较陌生，因此我们今天先用一篇文章跟大家捋清楚这些概念，概念搞明白了，再来看 TSL+gRPC 就很容易了。

7.1 HTTP 的问题

HTTP 协议是超文本传输协议（Hyper Text Transfer Protocol）的缩写，它是从 WEB 服务器传输超文本标记语言 HTML 到本地浏览器的传送协议。HTTP 设计之初是为了提供一种发布和接收 HTML 页面的方法，时至今日，它的作用已经不仅仅于此了。

对于我们 Java 工程师而言，HTTP 应该算是再熟悉不过的东西了，目前 HTTP 有多个版本，使用较多的是 HTTP/1.1 版本。

然而 HTTP 协议有一个缺陷那就是它是通过明文传输数据的，用户通过 HTTP 协议传输的内容很容易被恶意拦截，并且黑客可以伪装成服务端，向用户传送错误的信息，并且能轻易获取用户的隐私信息，而这些操作用户是完全无感知的。

由于存在这样的安全隐患，现在小伙伴们见到的大部分网站都在逐步转为 HTTPS，HTTP 网站会越来越少了。

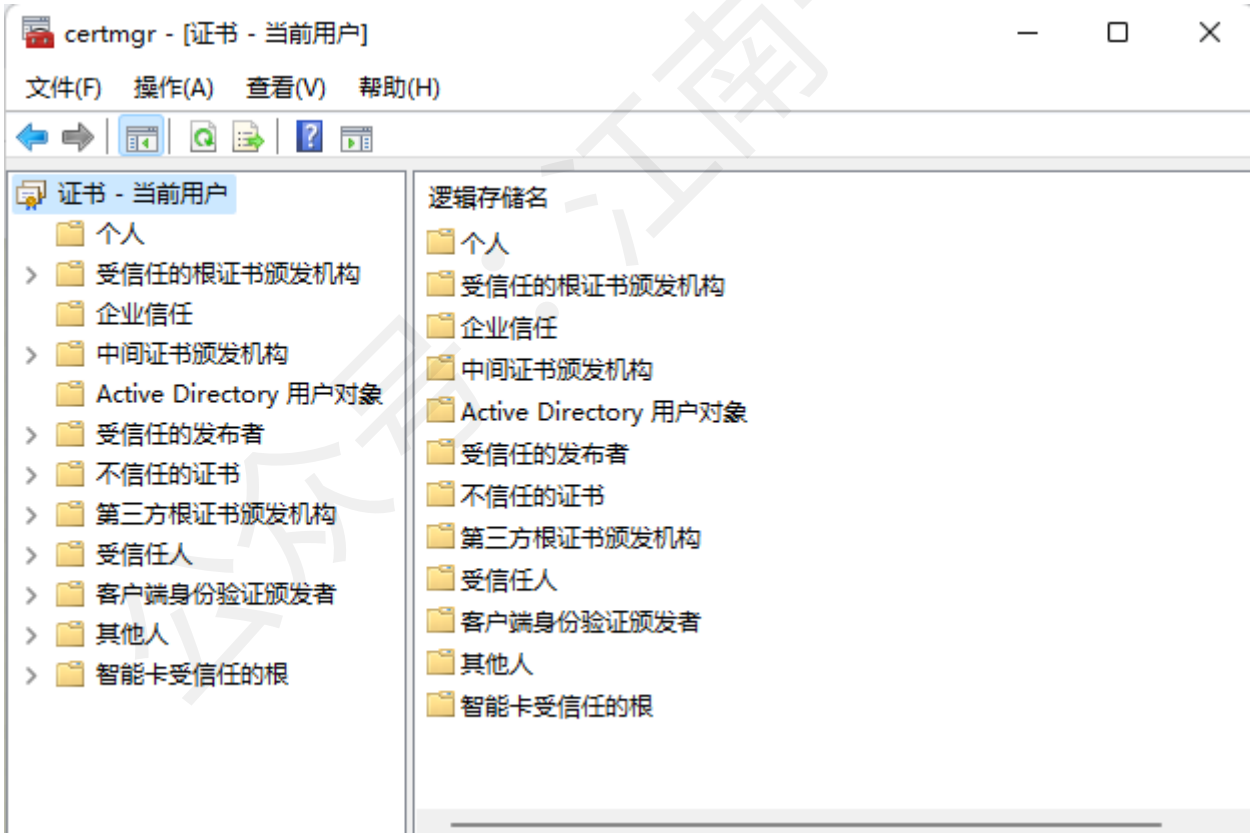
7.2 HTTPS

HTTPS（HyperText Transfer Protocol Secure）中文译作超文本传输安全协议，这是一种通过计算机网络进行安全通讯的传输协议。

HTTPS 本质上还是由 HTTP 进行通信，只是在 HTTP 协议和 TCP 层之间增加了一个 SSL 的安全传输协议。整个传输的加密过程都在新的安全层 SSL/TLS 中实现，而原来的 HTTP 层的传输流程保持不变，这样就很好地兼容了旧的 HTTP 协议，也沿袭了 TCP/IP 协议族的分层思想。

通过 HTTPS，客户端可以确认服务端的身份，保证数据在传输过程中不被篡改，当我们在自己的浏览器上与某一个网站建立 HTTPS 连接的时候，满足如下情况可以表示这个服务端可以被信任：

1. 首先我们的操作系统中安装了正确且受信任的证书。我们在 cmd 命令行中执行 `certmgr.msc` 命令，可以查看操作系统已经安装的证书列表。



2. 浏览器本身正确实现了 HTTPS。
3. 被访问的网站提供了一个证书，这个证书是由一个操作系统所信任的证书颁发机构签发的，操作系统所信任的证书颁发机构一般都预装在操作系统中，通过第一步的方式可以查看。
4. 被访问的网站所提供的证书被成功认证。

这里边涉及到一些证书和协议的概念，接下来松哥和大家把整个过程捋一捋。

7.3 TLS/SSL

前面我们提到，HTTPS 就是在 HTTP 的基础之上增加了 TLS/SSL，那么这两个东西该如何理解呢？

SSL/TLS 是一种密码通信方案，算是目前使用最广泛的密码通信方案了。SSL 全称是 Secure Socket Layer，中文译作安全套接层，是 1994 年由 Netscape 公司设计的一套协议，并与 1995 年发布了 3.0 版本；TLS 全称是 Transport Layer Security，中文译作传输层安全，则是 IETF 在 SSL3.0 基础上设计的协议，实际上相当于 SSL 的后续版本，目前 TLS 先后迭代了 `TLS 1.0`、`TLS 1.1`、`TLS 1.2` 和 `TLS 1.3`，目前被广泛使用的是 `TLS 1.2` 版本。

SSL/TLS 涉及到了密码学中的对称加密、非对称加密、数字签名等等，算是密码学领域里的集大成者了。

7.3.1 TLS

接下来我们就来看看 TLS 如何确保 HTTP 安全。

为了确保客户端和服务端之间的数据安全，我们很容易想到一种方案就是对传输的数据进行加密，没错，这是一个办法，事实上也是这么做的。

加密又分为两种：

1. 对称加密
2. 非对称加密

那么该使用哪一种呢？

对称加密，也就是加密密钥和解密密钥是同一个，当浏览器和服务端需要进行通信的时候，约定好一个密钥，然后使用这个密钥对发送的消息进行加密，对方收到消息之后再使用相同的密钥对消息进行解密。但是，在 B/S 架构的项目中，这种方案显然不合适，一个网站把自己的密钥告诉全世界所有的浏览器，那加密和不加密还有区别吗？

有小伙伴可能又想到了不对称加密，不对称加密倒是个办法，因为不对称加密是有一个密钥对公钥和私钥，公钥可以公布出来告诉所有人，私钥只有自己知道。通信的时候，客户端首先使用公钥对消息进行加密，服务端收到之后，再通过私钥对消息进行解密，这看起来似乎挺完美的。但是！！！非对称加密存在一个问题，就是非对称加密和解密相当耗时，通过这种方式处理加解密效率太低。

那怎么办？我们可以将两者结合起来。

具体来说，就是这样：**首先服务端会生成一个非对称加密的密钥对，私钥自己保存，公钥发送给客户端，客户端拿到这个公钥之后，再生成一个对称加密的密钥，然后把对称加密的密钥通过公钥进行加密，加密之后发送给服务端，服务端通过私钥进行解密，这样客户端和服务端就可以通过对称加密进行通信了。**

事实上，TLS 大致上的思路就是这样的。

不过上面这个方案还是有一个漏洞，那就是服务端要通过明文传输的方式把公钥发送给客户端，这个过程还是不安全的，可能被人恶意截胡，那么这个问题该如何解决呢？

这就涉及到另外一个概念叫做数字证书了。

7.3.2 CA

数字证书是一个**包含了目标网站各种信息如网站域名、证书有效期、签发机构、用于生成对称密钥的公钥、上级证书签发的签名等**的文件，通过数字证书我们可以确认一个用户或者服务站点的身份。

实际场景中的数字证书是一系列的，形成了一个信任链，信任链的最顶端是 CA。

CA 是 Certificate Authority 的简写，它是一个负责发放和管理数字的证书的第三方权威机构。CA 的工作流程是这样的：

1. CA 自己给自己颁发的用自己的私钥签名的证书称为根证书，根证书的私钥安全性至关重要，根证书的私钥都是被保存在离线计算机中，有严格的操作规章，每次需要使用时，会有专人将数据通过 USB 拷贝过去，操作完了以后，再将数据带出来（这个专指 CA 根证书的私钥）。
2. 一个用户想要获取一个证书，首先自己得有一个密钥对，私钥自己留着，公钥以及其他信息发送给 CA，向 CA 提出申请，CA 判明用户的身份之后，会将这个公钥和用户的身份信息绑定，并且为绑定后的信息进行签名（签名是通过 CA 根证书的私钥进行的），最后将签名后的证书发给申请者。
3. 一个用户想要鉴定一个证书的真伪，就通过 CA 的公钥对证书上的数字签名进行验证，验证通过，就认为这个这个证书是有效的。

上面这个流程中有一个重要前提，那就是 CA 受到大家所有人的信任。

然而在实际操作中，我们并不能直接去跟 CA 申请一个数字证书，因为全世界要认证的内容太多了，CA 搞不过来，而且频繁的找 CA 申请，还有可能导致私钥泄漏，这可就是一个大的灾难了。

那怎么办呢？实际操作中，我们可以基于 CA 来构建一个信任链。具体来说，步骤是这样：

1. 首先我们的手机、笔记本等操作系统中都预装了 CA 颁发的根证书，他们是所有信任构建的基石，前面松哥已经截图给大家看了 Windows 中预装的根证书了。
2. 假设 CA 签发了一个证书 A，在这个过程中 CA 称为 Issuer，A 称为 Subject，假设 A 是一个受信任的中间证书，已经预装在我们的操作系统中了。现在由 A 利用它自己的私钥给某一个网站签发了一个证书 B。
3. 现在当我们的电脑需要访问该网站的时候，该网站就会给我们发来一个证书 B，由于我们的浏览器并不知道 B 证书是否合法，但是我们的电脑上已经预装了 A 证书，我们可以从 A 证书中提取出 A 的公钥，然后利用 A 的公钥对 B 证书的签名进行验证（因为 B 证书的签名是 A 的私钥签的），如果验证通过了，就说明 B 是合法的。
4. 相同的道理，B 也可以继续签发 C 证书，C 继续签发 D 证书，这样就形成了一个信任链。
5. 如果服务端的签名是 D 证书，那么一般来说，服务器返回给浏览器的就会包含 B、C、D 三个证书（因为 A 证书已经在我们的电脑上了），即使只返回 D 证书，浏览器也可以根据 D 书中的信息，自动下载到 B、C 两个证书然后进行验证。

> *松哥记得以前上大学的时候，在 12306 网站上买火车票，第一次访问的时候必须要自己先手动安装一个根证书到系统中，然后才能访问。这就是因为当时 12306 所使用的证书的签发机构不被浏览器认可，类似于上面的第 3 步，12306 给我发了一个数字证书 B 回来，但是浏览器上没有合适的公钥对这个 B 证书进行验证，当我往自己的系统上安装了 12306 给的证书之后，相当于我的电脑上有了一个证书 A，现在就可以对 B 证书进行验证了。*

总结一下：

1. CA 是一个权威的机构，是一个发证机关，CA 发出来的证书可以证明一个人或者组织的身份。
2. 任何人都可以得到 CA 的证书（含公钥），用以验证 CA 所签发的证书。
3. 每一个数字证书都是由上级证书的私钥来签发的，处于最顶层的就是 CA 签发的根证书了，这个根证书没有上级证书了，所以这个根证书实际上是由 CA 自己的私钥来签发的，这也叫做自签名，即 Self-Signed。

当我们有了数字签名之后，就可以解决 3.1 小节最后提出的问题了。服务端将数字签名发给浏览器，浏览器利用系统已经内置的公钥验签，确认签名没问题，然后就提取出来数字签名中的公钥，开始协商对称加密的私钥了～

好啦，有了这些知识储备之后，下篇文章松哥来和大家聊一聊 TLS+gRPC 怎么玩！

8 gRPC+TLS

前面松哥发了一篇文章和小伙伴们仔细聊了聊 TLS、CA 证书这些问题，还没看过的小伙伴可以先戳下面了解下：

- TLS、SSL、CA 证书、公钥、私钥。。。今天捋一捋！

今天我们要在前文的基础之上，来和小伙伴们聊一聊如何确保 gRPC 的通信安全。

确保 gRPC 的通信安全我们有很多种不同的方式，其中一种，就是对通信过程进行加密，使用上 TLS。对于 TLS 如何加密，如何协商密钥，这些我这里就不再啰嗦了，我在之前的文章中都已经介绍过了。咱们就直接来看具体的玩法。

这块整体上可以分为两大类：

- 启用单向安全连接
- 启用 mTLS 安全连接

我们分别来看。

8.1 启用单向安全连接

单向安全连接其实就是说只需要客户端校验服务端，确保客户端收到的消息来自预期的服务端，整个的校验就涉及到我们前文所说的 TLS、CA 等内容了，具体流程是这样：

1. 首先我们先在自己电脑本地生成一个自签名的 CA 证书。
2. 利用这个 CA 证书，生成一个服务证书。

大致上就这两个步骤就行了，然后在客户端和服务端中分别加载相应的证书即可。

> *上面我们提到了需要先有一个自签名的 CA 证书，这一步其实也可以省略，省略之后就直接生成一个自签名的服务证书即可，然后在客户端和服务端都使用这个服务证书。*

来实际操作一下。

先自己安装一下 openssl 工具，配置一下环境变量，软件安装比较简单，我这里就不啰嗦了。

8.1.1 生成 CA 证书

首先我们来看下如何生成 CA 证书。

一共是三个步骤：

1. 生成 `.key` 私钥文件：

```
openssl genrsa -out ca.key 2048
```

- out 表示输出的文件名。
- 2048 表示私钥的位数。

2. 生成 `.csr` 证书签名请求文件：

CSR 即证书签名申请（Certificate Signing Request），获取 SSL 证书，需要先生成 CSR 文件并提交给证书颁发机构（CA）。CSR 包含了用于签发证书的公钥、用于辨识的名称信息（Distinguished Name）（例如域名）、真实性和完整性保护（例如数字签名），通常从 Web 服务器生成 CSR，同时创建加解密的公钥私钥对。

```
openssl req -new -key ca.key -out ca.csr  -subj
"/C=CN/L=GuangZhou/O=javaboy/CN=local.javaboy.org"
```

- subj 中描述的是一些国家、城市、组织以及通用名称（域名）等信息。

3. 自签名生成 `.crt` 证书文件

```
openssl req -new -x509 -days 3650 -key ca.key -out ca.crt  -subj
"/C=CN/L=GuangZhou/O=javaboy/CN=local.javaboy.org"
```

- -x509 表示是要生成自签名证书。
- -days 3650 表示证书有效期是 3650 天。

- -key 表示生成证书所需要的密钥。

有人说公钥呢？公钥其实就在 .crt 证书文件中。

8.1.2 生成服务证书

再来看生成服务证书，生成服务证书和生成 CA 证书其实整个过程差不多，唯一的区别在于，CA 证书是自签名的，而服务证书是 CA 的私钥给签名的，就这个差别。

1. 生成 .key 私钥文件：

```
openssl genrsa -out server.key 2048
```

2. 生成 .csr 证书签名请求文件：

```
openssl req -new -key server.key -out server.csr -subj
"/C=CN/L=GuangZhou/O=javaboy/CN=local.javaboy.org"
```

3. 签名生成 .crt 证书文件

```
openssl x509 -req -days 3650 -in server.csr -out server.crt -CA ca.crt -
CAkey ca.key
```

- -req 和 -in 指定了 server.csr，这个是证书请求文件，这里实际上是表示签署证书请求文件。

证书现在就生成完毕。

这里我们生成的私钥都是 .key 文件，这个用我们 Java 代码加载的时候会有问题，我们要将之转为 .pem 格式然后再用 Java 代码进行加载，转换的命令如下：

```
openssl pkcs8 -topk8 -inform pem -in server.key -outform pem -nocrypt -
out server.pem
```

8.1.3 单向加密

现在证书都有了，在当前项目目录下新建一个文件夹，专门用来放证书，项目目录结构如下：

```
├── certs
│   ├── ca.crt
│   ├── ca.csr
│   ├── ca.key
│   ├── server.crt
│   ├── server.csr
│   ├── server.key
│   └── server.pem
├── grpc_api
│   ├── pom.xml
```

```
│   ├── src
│   └── target
├── grpc_client
│   ├── pom.xml
│   ├── src
│   └── target
├── grpc_server
│   ├── pom.xml
│   ├── src
│   └── target
└── pom.xml
```

我们看下代码该如何改造实现单向加密通信。

先来看服务端代码：

```
public void start() throws IOException {
    int port = 50051;
    File certFile = Paths.get( "certs", "server.crt").toFile();
    File keyFile = Paths.get("certs", "server.pem").toFile();
    server = ServerBuilder.forPort(port)
            .addService(new LoginServiceImpl())
            .addService(ServerInterceptors.intercept(new
HelloServiceImpl(), new AuthInterceptor()))
            .useTransportSecurity(certFile,keyFile)
            .build()
            .start();
    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        LoginServer.this.stop();
    }));
}
```

> *大家注意，由于我生成签名的时候，使用的域名是 `local.javaboy.org` 这是我在本地 hosts 文件中配置的，指向本地地址，所以在后续的通信中，我使用的域名都将是 `local.javaboy.org`。*

1. Paths.get 方法表示从项目的根目录下开始查找文件，参数是可变长度参数，参数共同组成文件完整路径。
2. 服务端需要加载服务签名和服务私钥，签名证书是客户端验证服务端身份用的，私钥则是服务端解密客户端消息使用的。

服务端的改造就这些。

再来看客户端的改造：

```
File certFile = Paths.get( "certs", "ca.crt").toFile();
SslContext sslContext =
GrpcSslContexts.forClient().trustManager(certFile).build();
ManagedChannel channel =
NettyChannelBuilder.forAddress("local.javaboy.org", 50051)
        .useTransportSecurity()
        .sslContext(sslContext)
        .build();
```

客户端主要是加载 CA 证书文件，服务端的证书就是 CA 私钥签发的，但是需要 CA 公钥也就是 ca.crt 进行验签，所以这里客户端加载了 ca.crt 即可。

好啦，整体上的流程差不多就是这个样子。

8.2 启用 mTLS 安全连接

上面的例子只是客户端校验了服务端的身份，服务端并没有校验客户端的身份，如果想要双向校验，那么就把上面的流程对称操作一遍就可以了。

首先我们需要为客户端生成相应的证书，步骤跟前面也基本上一直，使用 CA 进行签名，如下：

1. 生成 `.key` 私钥文件：

```
openssl genrsa -out client.key 2048
```

2. 生成 `.csr` 证书签名请求文件：

```
openssl req -new -key client.key -out client.csr -subj
"/C=CN/L=GuangZhou/O=javaboy/CN=local.javaboy.org"
```

3. 签名生成 `.crt` 证书文件

```
openssl x509 -req -days 3650 -in client.csr -out client.crt -CA ca.crt -
CAkey ca.key
```

然后来看看代码。

先来看服务端：

```
public void start() throws IOException {
    int port = 50051;
    File certFile = Paths.get( "certs", "server.crt").toFile();
    File keyFile = Paths.get("certs", "server.pem").toFile();
    File caFile = Paths.get("certs", "ca.crt").toFile();
    server = NettyServerBuilder.forPort(port)
            .addService(new LoginServiceImpl())
```

```
                .addService(ServerInterceptors.intercept(new
HelloServiceImpl(), new AuthInterceptor()))

.sslContext(GrpcSslContexts.forServer(certFile,keyFile).trustManager(caF
ile).clientAuth(ClientAuth.REQUIRE).build())
                .build()
                .start();
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            LoginServer3.this.stop();
        }));
    }
```

服务端要加载的文件多了 ca.crt，这是给客户端验签的时候需要用到。

再来看看客户端代码：

```
File caFile = Paths.get( "certs", "ca.crt").toFile();
File certFile = Paths.get( "certs", "client.crt").toFile();
File keyFile = Paths.get( "certs", "client.pem").toFile();
SslContext sslContext = GrpcSslContexts.forClient().trustManager(caFile)
        .keyManager(certFile, keyFile).build();
ManagedChannel channel =
NettyChannelBuilder.forAddress("local.javaboy.org", 50051)
        .useTransportSecurity()
        .sslContext(sslContext)
        .build();
```

客户端多了 client.crt 和 client.pem，两者的作用根服务端中这两者的作用基本一致，前文已有说明，这里就不再赘述了。

好啦，如此之后，我们的 gRPC 通信就加上了 TLS 的外壳，更加安全了。

9 gRPC 两种认证方式

在之前的文章中，松哥和小伙伴们聊了 gRPC+JWT 进行认证，这也是我们常用的认证方式之一，考虑到文章内容的完整性，今天松哥再来和小伙伴们聊一聊在 gRPC 中通过 HttpBasic 进行认证，HttpBasic 认证有一些天然的缺陷，这个在接下来的文章中松哥也会和大家进行分析。

好啦，如果还没看过之前的 gRPC+JWT 的文章，戳这里：

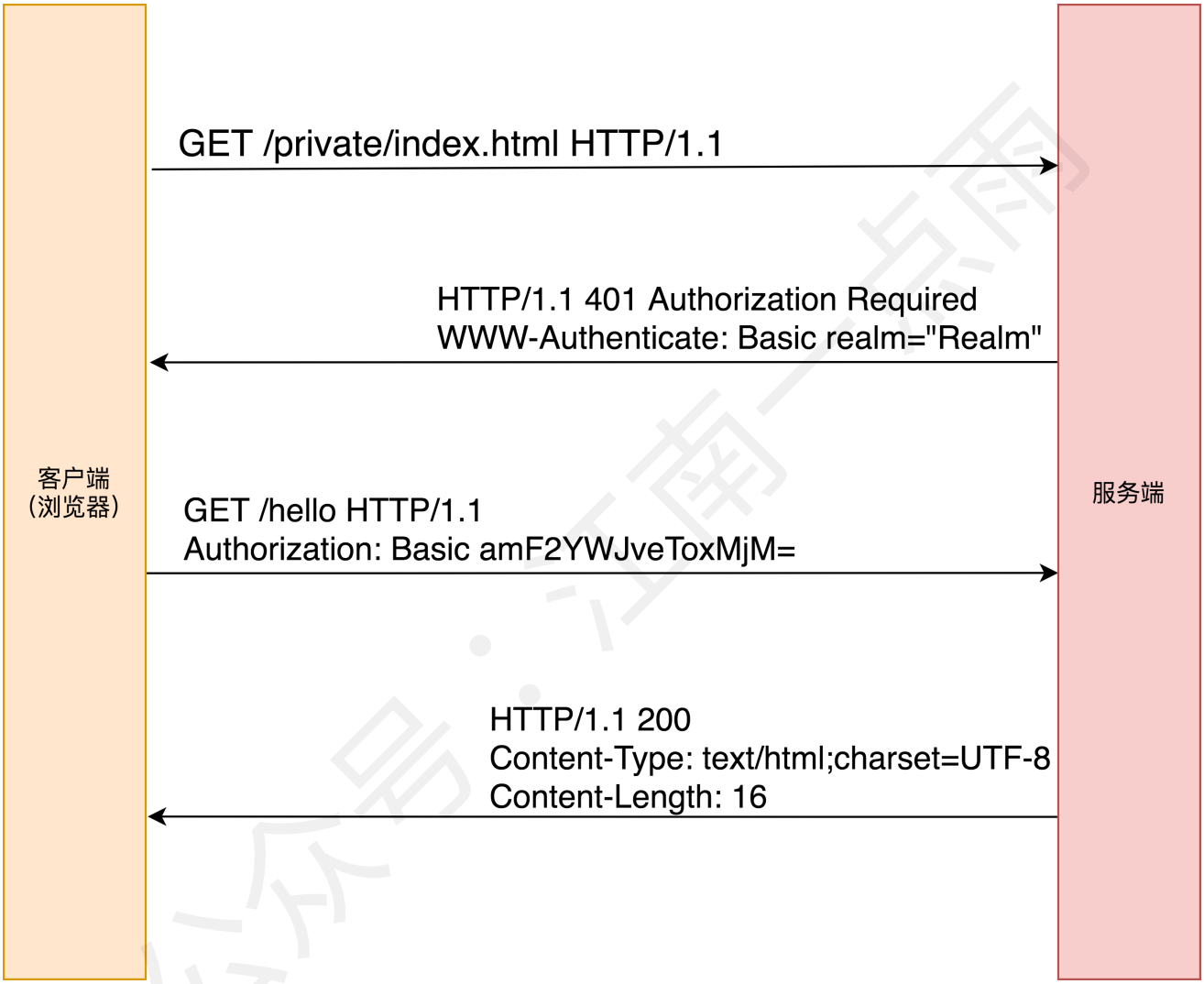
- 手把手教大家在 gRPC 中使用 JWT 完成身份校验

今天我们就来看看如何在 gRPC 中进行 Http Basic 认证。

9.1 什么是 Basic 认证

HTTP Basic authentication 中文译作 HTTP 基本认证，在这种认证方式中，将用户的登录用户名/密码经过 Base64 编码之后，放在请求头的 Authorization 字段中，从而完成用户身份的 认证。

这是一种在 RFC7235(https://tools.ietf.org/html/rfc7235) 规范中定义的认证方式，当客户端发起一个请求之后，服务端可以针对该请求返回一个质询信息，然后客户端再 供用户的凭 证信息。具体的质询与应答流程如图所示：



由上图可以看出，客户端的用户名和密码只是简单做了一个 Base64 转码，然后放到请求头中就传输到服务端了。

我们在日常的开发中，其实也很少见到这种认证方式，有的读者可能在一些老旧路由器中见过这种认证方式;另外，在一些非公开访问的 Web 应用中，可能也会见到这种认证方式。为什么很少见到这种认证方式的应用场景呢?主要还是安全问题。

HTTP 基本认证没有对传输的凭证信息进行加密，仅仅只是进行了 Base64 编码，这就造成了很大的安全隐患，所以如果用到了 HTTP 基本认证，一般都是结合 HTTPS 一起使用；同 时，一旦使用 HTTP 基本认证成功后，由于令牌缺乏有效期，除非用户重启浏览器或者修改密码，否则没有办法退出登录。

9.2 gRPC 中的基本认证

gRPC 并没有为 Http Basic 认证提供专门的 API，如果我们需要在 gRPC 中进行 Http Basic 认证，需要自己手工处理。

不过相信小伙伴们看了上面的流程图之后，对于手工处理 gRPC+Http Basic 也没啥压力。

首先我们先来看客户端的代码：

```
public class HttpBasicCredential extends CallCredentials {
    private String username;
    private String password;

    public HttpBasicCredential(String username, String password) {
        this.username = username;
        this.password = password;
    }

    @Override
    public void applyRequestMetadata(RequestInfo requestInfo, Executor
executor, MetadataApplier metadataApplier) {
        executor.execute(() -> {
            try {
                String token = new
String(Base64.getEncoder().encode((username + ":" +
password).getBytes()));
                Metadata headers = new Metadata();
                headers.put(Metadata.Key.of(AuthConstant.AUTH_HEADER,
Metadata.ASCII_STRING_MARSHALLER),
                        String.format("%s %s",
AuthConstant.AUTH_TOKEN_TYPE, token));
                metadataApplier.apply(headers);
            } catch (Throwable e) {

 metadataApplier.fail(Status.UNAUTHENTICATED.withCause(e));
            }
        });
    }

    @Override
    public void thisUsesUnstableApi() {

    }
}
```

- 当客户端发起一个请求的时候，我们构建一个 HttpBasicCredential 对象，并传入用户名和密码。
- 该对象核心的处理逻辑在 applyRequestMetadata 方法中，我们先按照 `username + ":" + password` 的形式将用户名和密码拼接成一个字符串，并对这个字符串进行

Base64 编码。
- 最后将编码结果放在请求头中，请求头的 KEY 就是 `AuthConstant.AUTH_HEADER` 变量，对应的具体值是 Authorization，请求头的 value 是通过 `String.format` 函数拼接出来的，实际上就是在 Base64 的编码的字符串上加上了 `Basic` 前缀。

这块就是纯手工操作，技术原理跟我们之前讲的 JWT+gRPC 没有任何差别，基本上是一模一样的，所以我就不啰嗦了。

来看下前端请求该如何发起：

```
public class LoginClient {
    public static void main(String[] args) throws InterruptedException,
SSLException {

        File certFile = Paths.get( "certs", "ca.crt").toFile();
        SslContext sslContext =
GrpcSslContexts.forClient().trustManager(certFile).build();

        ManagedChannel channel =
NettyChannelBuilder.forAddress("local.javaboy.org", 50051)
                .useTransportSecurity()
                .sslContext(sslContext)
                .build();

        LoginServiceGrpc.LoginServiceStub stub =
LoginServiceGrpc.newStub(channel).withDeadline(Deadline.after(3,
TimeUnit.SECONDS));
        sayHello(channel);
    }

    private static void sayHello(ManagedChannel channel) throws
InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(1);
        HelloServiceGrpc.HelloServiceStub helloServiceStub =
HelloServiceGrpc.newStub(channel);
        helloServiceStub
                .withCallCredentials(new HttpBasicCredential("javaboy",
"123"))

.sayHello(StringValue.newBuilder().setValue("wangwu").build(), new
StreamObserver<StringValue>() {
            @Override
            public void onNext(StringValue stringValue) {
                System.out.println("stringValue.getValue() = " +
stringValue.getValue());
            }

            @Override
            public void onError(Throwable throwable) {
```

```
                System.out.println("throwable.getMessage() = " +
throwable.getMessage());
            }

            @Override
            public void onCompleted() {
                countDownLatch.countDown();
            }
        });
        countDownLatch.await();
    }
}
```

通过 withCallCredentials 方法，在客户端发起请求的时候，把这段认证信息携带上。

再来看看服务端的处理。

服务端通过一个拦截器来统一处理，从请求头中提取出来认证信息并解析判断，逻辑如下：

```
public class AuthInterceptor implements ServerInterceptor {
    private JwtParser parser =
Jwts.parser().setSigningKey(AuthConstant.JWT_KEY);

    @Override
    public <ReqT, RespT> ServerCall.Listener<ReqT>
interceptCall(ServerCall<ReqT, RespT> serverCall, Metadata metadata,
ServerCallHandler<ReqT, RespT> serverCallHandler) {
        String authorization =
metadata.get(Metadata.Key.of(AuthConstant.AUTH_HEADER,
Metadata.ASCII_STRING_MARSHALLER));
        Status status = Status.OK;
        if (authorization == null) {
            status = Status.UNAUTHENTICATED.withDescription("miss
authentication token");
        } else if
(!authorization.startsWith(AuthConstant.AUTH_TOKEN_TYPE)) {
            status = Status.UNAUTHENTICATED.withDescription("unknown
token type");
        } else {
            try {
                String token =
authorization.substring(AuthConstant.AUTH_TOKEN_TYPE.length()).trim();
                String[] split = new
String(Base64.getDecoder().decode(token)).split(":");
                String username = split[0];
                String password = split[1];
                if ("javaboy".equals(username) &&
"123".equals(password)) {
                    Context ctx = Context.current()
```

```
                                .withValue(AuthConstant.AUTH_CLIENT_ID,
username);
                        return Contexts.interceptCall(ctx, serverCall,
metadata, serverCallHandler);
                    }
                } catch (JwtException e) {
                    status =
Status.UNAUTHENTICATED.withDescription(e.getMessage()).withCause(e);
                }
            }
            serverCall.close(status, new Metadata());
            return new ServerCall.Listener<ReqT>() {
            };
        }
    }
```

1. 首先从请求头中取出 Base64 编码之后的令牌。
2. 如果取出的值为 null，则返回 `miss authentication token`。
3. 如果取出的令牌的起始字符不对，则返回 `unknown token type`。
4. 如果前面都没问题，则开始对拿到的字符串进行 Base64 解码，解码之后做字符串拆分，然后分别判断用户名和密码是否正确，如果正确，则将用户名存入到 Context 中，在后续的业务逻辑中就可以使用了。

服务端的启动代码如下：

```
public class LoginServer {
    Server server;

    public static void main(String[] args) throws IOException,
InterruptedException {
        LoginServer server = new LoginServer();
        server.start();
        server.blockUntilShutdown();
    }

    public void start() throws IOException {
        int port = 50051;
        File certFile = Paths.get( "certs", "server.crt").toFile();
        File keyFile = Paths.get("certs", "server.pem").toFile();
        server = ServerBuilder.forPort(port)
                .addService(ServerInterceptors.intercept(new
HelloServiceImpl(), new AuthInterceptor()))
                .useTransportSecurity(certFile,keyFile)
                .build()
                .start();
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            LoginServer.this.stop();
        }));
    }
```

```
    private void stop() {
        if (server != null) {
            server.shutdown();
        }
    }

    private void blockUntilShutdown() throws InterruptedException {
        if (server != null) {
            server.awaitTermination();
        }
    }
}
```

小伙伴们看下，就是用了下这个拦截器而已。

最后，在业务代码中，也可以直接访问到刚刚认证成功的用户名：

```
public class HelloServiceImpl extends
HelloServiceGrpc.HelloServiceImplBase {
    @Override
    public void sayHello(StringValue request,
StreamObserver<StringValue> responseObserver) {
        String clientId = AuthConstant.AUTH_CLIENT_ID.get();

 responseObserver.onNext(StringValue.newBuilder().setValue(clientId + "
say hello:" + request.getValue()).build());
        responseObserver.onCompleted();
    }
}
```

好啦，大功告成。

9.3 小结

和之前的 JWT 相比，Http Basic 认证的缺点还是非常明显的，但是从认证流程来说，感觉两者差别不大，只是创建令牌和解析令牌的方式不同而已。

感兴趣的小伙伴可以尝试一下哦。

本文松哥只贴出来了一些关键代码，完整的代码小伙伴们可以从 GitHub 上下载：https://github.com/lenve/javaboy-code-samples

10 Spring Boot+Nacos+gRPC

gRPC 的基础知识前面跟小伙伴们分享了很多了，今天再写一篇给这个小小的系列收个尾。

我们前面介绍的都是 gRPC 的基本用法，最终目的当然是希望能够在 Spring Boot 中用上这个东西，相信大部分小伙伴对于微服务的通信方案如 OpenFeign、Dubbo、消息驱动都有所了解，但是对于这三种方案之外的其他方案，可能听的多用的少，今天我们就来实践一下 gRPC 这种方案。

> *顺便说一下我为什么会想到写 gRPC 教程呢，是因为之前我想给小伙伴们总结一下常见的各种微服务通信方案。整理到 gRPC 的时候发现我还没写过 gRPC 相关的教程，因此就有了一个小系列。*

10.1 依赖选择

Spring Boot 整合 gRPC，官方其实并没有提供相应的依赖，不过目前有一个比较流行的第三方库可以使用：

- https://github.com/yidongnan/grpc-spring-boot-starter

接下来松哥就结合这个库，来和小伙伴们演示一下 Spring Boot+Nacos+gRPC 的用法。

可能有小伙伴也会见到一些其他的第三方库，这个其实都可以，只要稳定可靠就行，本文就以上面这个库为例来和小伙伴们介绍。

10.2 准备工作

这里我采用了 Nacos 来做服务注册中心，使用的 Nacos 版本是 2.0.2 这个版本。Nacos 简单安装一下就行了，为了省事，数据持久化啥的可以先不配置。也就是 Nacos 下载解压之后，直接执行如下命令单体运行就行了：

```
sh startup.sh -m standalone
```

这块没啥好说的，松哥在 vhr 系列里也有相关的视频教程，这里就不啰嗦了。

10.3 代码实践

首先我们来看看我们的项目结构：

```
├─grpc-api
│  └─src
│      └─main
│          └─proto
├─grpc-client
```

```
|   └─src
|       ├─main
|       |   ├─java
|       |   └─resources
|       └─test
└─grpc-server
    └─src
        ├─main
        |   ├─java
        |   └─resources
        └─test
```

首先有一个公共的模块 grpc-api，这个模块用来放我们的公共代码和依赖，包括 Protocol Buffers 文件也放在这里。

grpc-client 和 grpc-server 就不用多说了，分别是我们的客户端和服务端。

10.3.1 grpc-api

grpc-api 中主要是处理 grpc 相关的事情，包括添加需要的依赖、插件等，编写 Protocol Buffers 文件等。

我们先来看看该项目的 pom.xml 文件中的依赖：

```
<properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <protobuf.version>3.21.7</protobuf.version>
    <protobuf-plugin.version>0.6.1</protobuf-plugin.version>
    <grpc.version>1.52.1</grpc.version>
</properties>
<dependencies>
    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-stub</artifactId>
        <version>${grpc.version}</version>
    </dependency>
    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-protobuf</artifactId>
        <version>${grpc.version}</version>
    </dependency>
    <dependency>
        <!-- Java 9+ compatibility - Do NOT update to 2.0.0 -->
        <groupId>jakarta.annotation</groupId>
        <artifactId>jakarta.annotation-api</artifactId>
        <version>1.3.5</version>
        <optional>true</optional>
```

```
    </dependency>
</dependencies>
<build>
    <extensions>
        <extension>
            <groupId>kr.motd.maven</groupId>
            <artifactId>os-maven-plugin</artifactId>
            <version>1.7.0</version>
        </extension>
    </extensions>
    <plugins>
        <plugin>
            <groupId>org.xolstice.maven.plugins</groupId>
            <artifactId>protobuf-maven-plugin</artifactId>
            <version>${protobuf-plugin.version}</version>
            <configuration>

 <protocArtifact>com.google.protobuf:protoc:${protobuf.version}:exe:${os
.detected.classifier}</protocArtifact>
                <pluginId>grpc-java</pluginId>
                <pluginArtifact>io.grpc:protoc-gen-grpc-
java:${grpc.version}:exe:${os.detected.classifier}</pluginArtifact>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                        <goal>compile-custom</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

这块的依赖跟我们之前的 gRPC 文章中案例的依赖基本上都是一致的，没有区别，再来看看我们的 Protocol Buffers 文件：

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.javaboy.grpc.api";
option java_outer_classname = "LoginProto";
import "google/protobuf/wrappers.proto";

package login;

service HelloService{
  rpc sayHello(google.protobuf.StringValue) returns
(google.protobuf.StringValue);
}
```

很简单的一个 HelloService 服务。

10.3.2 grpc-server

grpc-server 则是我们的服务端，这是一个 Spring Boot 工程，项目依赖如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.7</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>org.javaboy</groupId>
<artifactId>grpc-server</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>grpc-server</name>
<description>grpc-server</description>
<properties>
    <java.version>17</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
```

```
    <dependency>
        <groupId>org.javaboy</groupId>
        <artifactId>grpc-api</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
        <groupId>net.devh</groupId>
        <artifactId>grpc-server-spring-boot-starter</artifactId>
        <version>2.14.0.RELEASE</version>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>2021.0.5.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

由于第三方库 `grpc-server-spring-boot-starter` 在支持 Spring Boot3 上还有一些瑕疵，因此我这里使用了 Spring Boot2.7.7 这个版本。

这里需要注意的是就是添加了 gRPC 的依赖 `grpc-server-spring-boot-starter` 和 nacos 的依赖。其他都是常规配置。

接下来我们来在服务端提供 gRPC 方法的实现：

```
@GrpcService
public class HelloServiceImpl extends
HelloServiceGrpc.HelloServiceImplBase {
    @Override
    public void sayHello(StringValue request,
StreamObserver<StringValue> responseObserver) {
        String value = request.getValue();
        responseObserver.onNext(StringValue.newBuilder().setValue("hello
" + value).build());
        responseObserver.onCompleted();
    }
}
```

小伙伴们看到，通过 @GrpcService 注解去标记我们的一个服务即可。

最后，在 application.yaml 中进行配置，将当前服务注册到 nacos 容器中：

```
grpc:
  server:
    port: 9099
spring:
  cloud:
    nacos:
      discovery:
        server-addr: hc.javaboy.org:8848
        username: nacos
        password: nacos
        enabled: true
  application:
    name: grpc_server
```

OK，服务端搞定。

10.3.3 grpc-client

最后再来看看客户端。

先来看依赖：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.7</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>org.javaboy</groupId>
<artifactId>grpc-client</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>grpc-client</name>
```

```
<description>grpc-client</description>
<properties>
    <java.version>17</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.javaboy</groupId>
        <artifactId>grpc-api</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
        <groupId>net.devh</groupId>
        <artifactId>grpc-client-spring-boot-starter</artifactId>
        <version>2.14.0.RELEASE</version>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>2021.0.5.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

注意，客户端 grpc 的依赖是 grpc-client-spring-boot-starter，其他的基本上和服务端一致。

然后配置客户端，将之注册到 nacos 上，如下：

```
server:
  port: 8088
spring:
  cloud:
    nacos:
      discovery:
        enabled: true
        server-addr: hc.javaboy.org:8848
        username: nacos
        password: nacos
  application:
    name: grpc_client
grpc:
  client:
    grpc_server:
      negotiation-type: plaintext
```

最后面有一个 grpc_server，这个是固定的（依据就是 grpc_server 注册到 nacos 上的名称），表示这个服务的通信不使用 TLS 加密。

最后再来看看调用代码：

```
@RestController
public class HelloController {

    @Autowired
    GrpcClientService grpcClientService;

    @GetMapping("/hello")
    public void hello() {
        grpcClientService.hello();
    }

}
@Component
public class GrpcClientService {

    @GrpcClient("grpc_server")
    HelloServiceGrpc.HelloServiceBlockingStub helloServiceBlockingStub;

    public void hello() {
        StringValue s =
helloServiceBlockingStub.sayHello(StringValue.newBuilder().setValue("jav
aboy").build());
```

```
        System.out.println("s = " + s.getValue());
    }
}
```

这里的核心其实就是通过 `@GrpcClient` 注解注入一个 HelloServiceBlockingStub 实例，其中 `@GrpcClient` 注解中的参数就是注册到 nacos 上服务的名字，将来会自动根据服务的名字查找到服务的具体地址进行调用。

好啦，大功告成。

接下来我们启动 grpc_server 和 grpc_client 就可以进行测试了。