课程内容

1. 如何将一个应用改造为一个Dubbo应用
2. Dubbo3.0新特性介绍
3. Dubbo3.0跨语言调用
4. Dubbo3.0与gRPC互通
5. Dubbo3.0与Spring Cloud互通

有道云链接：https://note.youdao.com/s/XtLfKOoj

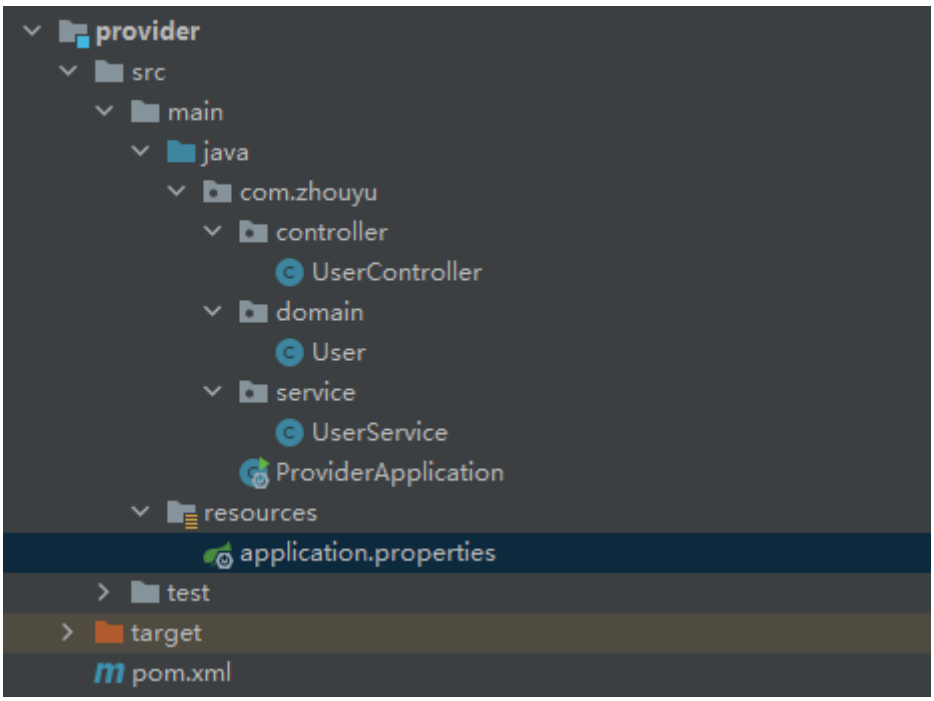
Dubbo3.0 Demo：https://gitee.com/archguide/dubbo3-demo
grpc Demo：https://gitee.com/archguide/grpc-demo
springcloud Demo：https://gitee.com/archguide/spring-cloud-demo
spring-cloud-alibaba-dubbo-examples：https://gitee.com/archguide/spring-cloud-alibaba-dubbo-examples.git

如何将一个应用改造为一个Dubbo应用

首先，新建两个SpringBoot项目，一个叫consumer，一个叫provider

provider项目

项目结构



pom文件

```
<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
                <version>2.6.6</version>
        </dependency>

        <dependency>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>1.18.22</version>
        </dependency>
</dependencies>
```

ProviderApplication

```
@SpringBootApplication
public class ProviderApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProviderApplication.class);
    }
}
```

User

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {
    private String uid;
    private String username;
}
```

UserService

```java
@Service
public class UserService {

    public User getUser(String uid) {
        User zhouyu = new User(uid, "zhouyu");
        return zhouyu;
    }
}
```

UserController

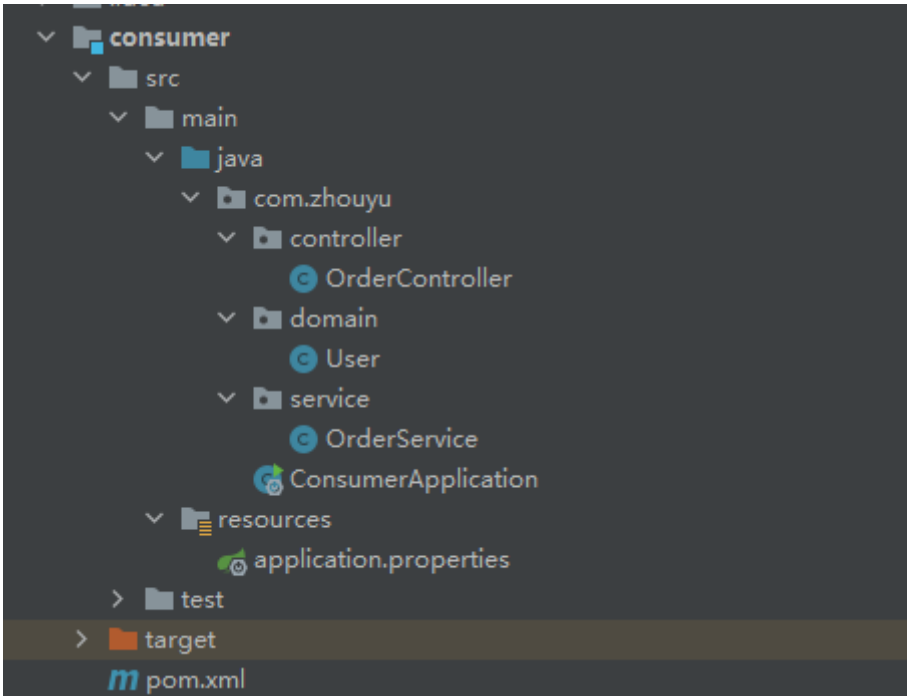
```java
@RestController
public class UserController {

    @Resource
    private UserService userService;

    @GetMapping("/user/{uid}")
    public User getUser(@PathVariable("uid") String uid) {
        return userService.getUser(uid);
    }
}
```

application.properties

```
server.port=8082
```

consumer项目

项目结构



pom文件

```xml
<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
                <version>2.6.6</version>
        </dependency>

        <dependency>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>1.18.22</version>
        </dependency>
</dependencies>
```

ConsumerApplication

```java
@SpringBootApplication
public class ConsumerApplication {

    @Bean
```

```
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }

    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class);
    }
}
```

User

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {
    private String uid;
    private String username;
}
```

OrderService

```
@Service
public class OrderService {

    @Resource
    private RestTemplate restTemplate;

    public String createOrder(){
        User user = restTemplate.getForObject("http://localhost:8082/user/1", User.class);
        System.out.println("创建订单");

        return user.toString()+" succeeded in creating the order";
    }
}
```

OrderController

```
@RestController
public class OrderController {

    @Resource
    private OrderService orderService;

    @GetMapping("/createOrder")
    public String createOrder() {
        return orderService.createOrder();
    }
}
```

application.properties

```
server.port=8081
```

consumer中的OrderService会通过RestTemplate调用provider中的UserService。

改造成Dubbo

改造成Dubbo项目，有几件事情要做：

1. 添加dubbo核心依赖
2. 添加要使用的注册中心依赖
3. 添加要使用的协议的依赖
4. 配置dubbo相关的基本信息
5. 配置注册中心地址
6. 配置所使用的协议

增加依赖

```
<dependency>
```

```
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-spring-boot-starter</artifactId>
        <version>3.0.7</version>
</dependency>

<dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-rpc-dubbo</artifactId>
        <version>3.0.7</version>
</dependency>

<dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-registry-zookeeper</artifactId>
        <version>3.0.7</version>
</dependency>
```

并且可以把不使用spring-boot-starter-web了，只需要使用spring-boot-starter即可。

配置properties

```
dubbo.application.name=provider-application

dubbo.protocol.name=dubbo
dubbo.protocol.port=20880

dubbo.registry.address=zookeeper://127.0.0.1:2181
```

改造服务

consumer和provider中都用到了User类，所以可以单独新建一个maven项目用来存consumer和provider公用的一些类，新增一个common模块，把User类转移到这个模块中

要改造成Dubbo，得先抽象出来服务，用接口表示。

像UserService就是一个服务，不过我们得额外定义一个接口，我们把之前的UserService改为UserServiceImpl，然后新定义一个接口UserService，该接口表示一个服务，UserServiceImpl为该服务的具体实现。

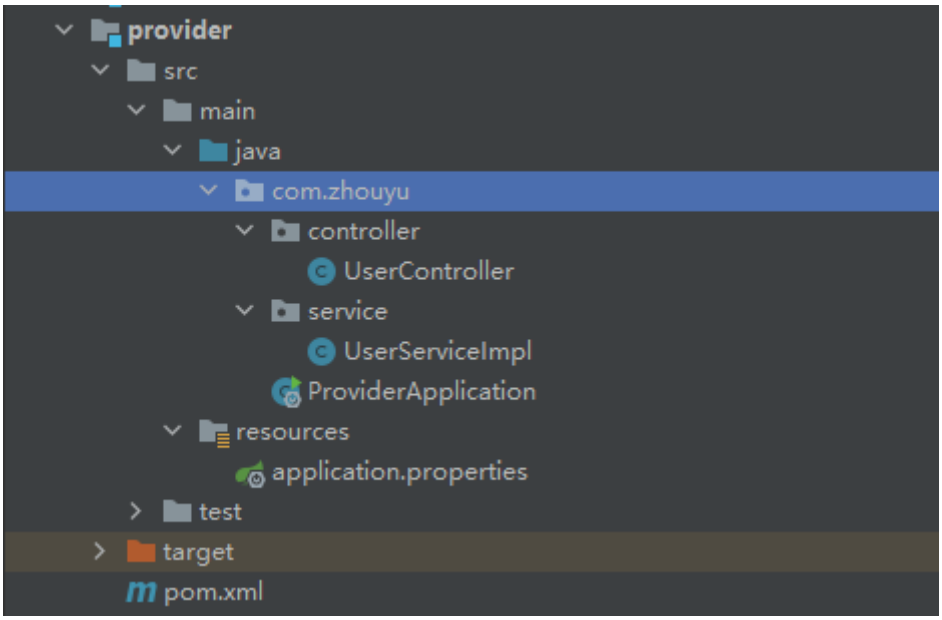
```java
public interface UserService {

    public User getUser(String uid);
}
```

```java
@DubboService
public class UserServiceImpl implements UserService {

    public User getUser(String uid) {
        User zhouyu = new User(uid, "zhouyu");
        return zhouyu;
    }
}
```

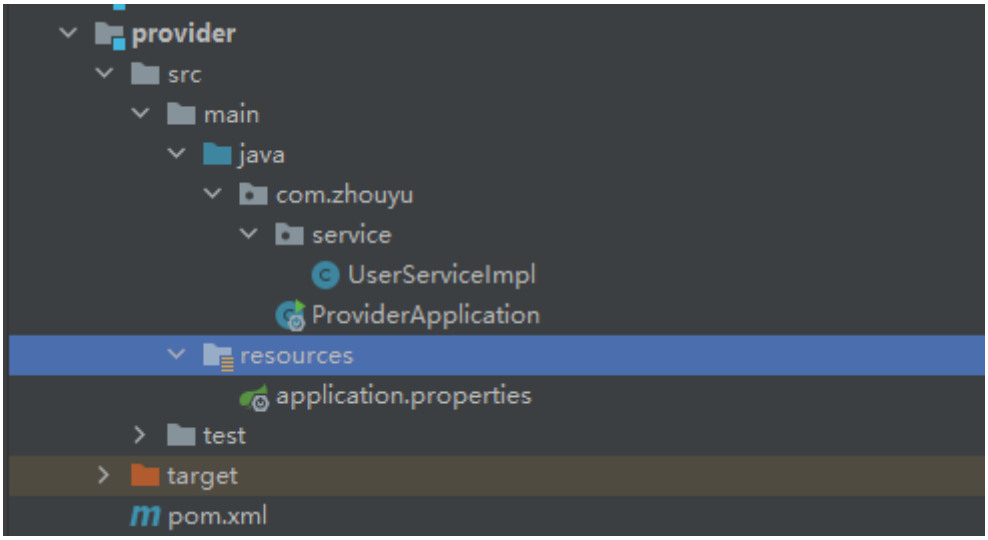
注意：要把Spring中的@Service注解替换成Dubbo中的@DubboService注解。

然后把UserService接口也转移到common模块中去，在provider中依赖common。

改造之后的provider为：



其实UserController也可以去掉，去掉之后provider就更加简单了



此时就可以启动该Provider了，注意先启动zookeeper（**高版本的Zookeeper启动过程中不仅会占用2181，也会占用8080，所以可以把provider的端口改为8082**）

开启Dubbo

在ProviderApplication上加上**@EnableDubbo(scanBasePackages = "com.zhouyu.service")**，表示Dubbo会去扫描某个路径下的@DubboService，从而对外提供该Dubbo服务。

```
@SpringBootApplication
@EnableDubbo(scanBasePackages = "com.zhouyu.service")
public class ProviderApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProviderApplication.class);
    }
}
```

兼容

前面我们仅仅是把provider应用改造成了Dubbo，但是在开发过程中，很有可能有其他多个应用在调用provider应用，对于provider应用要做到，既要能支持dubbo协议调用，也要能支持http调用（和controller效果一样），所以，要么仍然保留SpringMVC那一套，如果不想保留那一套，就可以开启dubbo中的rest协议，比如：

```
dubbo.application.name=provider-application

```

```
dubbo.protocols.p1.name=dubbo
dubbo.protocols.p1.port=20880

dubbo.protocols.p2.name=rest
dubbo.protocols.p2.port=8082

dubbo.registry.address=zookeeper://127.0.0.1:2181
```

并且添加rest的依赖：

```
<dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-rpc-rest</artifactId>
        <version>3.0.7</version>
</dependency>
```

然后改造UserServiceImpl：

```
@DubboService
@Path("/user")
@Produces
public class UserServiceImpl implements UserService{

    @Override
    @GET
    @Path("/{uid}")
        @Produces(MediaType.APPLICATION_JSON)
    public User getUser(@PathParam("uid")  String uid) {
        User zhouyu = new User(uid, "zhouyu");
        return zhouyu;
    }
}
```

这样只要启动provider应用，consumer虽然还没有改造为dubbo，但是照样可以用restTemplate调用provider中的getUser()方法。

调用Dubbo服务

引入依赖

在consumer中如果想要调用Dubbo服务，也要引入相关的依赖：

1. 引入common，主要是引入要用调用的接口
2. 引入dubbo依赖
3. 引入需要使用的协议的依赖
4. 引入需要使用的注册中心的依赖

```
            <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-spring-boot-starter</artifactId>
        <version>3.0.7</version>
    </dependency>

    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-rpc-dubbo</artifactId>
        <version>3.0.7</version>
    </dependency>

    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-registry-zookeeper</artifactId>
        <version>3.0.7</version>
    </dependency>

    <dependency>
        <groupId>com.zhouyu</groupId>
        <artifactId>common</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
```

引入服务

通过@DubboReference注解来引入一个Dubbo服务。

```java
@Service
public class OrderService {

    @DubboReference
    private UserService userService;

    public String createOrder(){
        User user = userService.getUser("1");
        System.out.println("创建订单");

        return user.toString()+" succeeded in creating the order";
    }
}
```

这样就不需要用RestTemplate了。

配置properties

```
dubbo.application.name=consumer-application
dubbo.registry.address=zookeeper://127.0.0.1:2181
```

调用

如果User没有实现Serializable接口，则会报错。

总结

自此，Dubbo的改造就完成了，总结一下：

1. 添加pom依赖
2. 配置dubbo应用名、协议、注册中心
3. 定义服务接口和实现类
4. 使用@DubboService来定义一个Dubbo服务
5. 使用@DubboReference来使用一个Dubbo服务
6. 使用@EnableDubbo开启Dubbo

log4j.properties

```
###set log levels###
log4j.rootLogger=info, stdout
###output to the console###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%d{dd/MM/yy hh:mm:ss:sss z}] %t %5p %c{2}: %m%n

```

Dubbo3.0新特性介绍

注册模型的改变

在服务注册领域，市面上有两种模型，一种是应用级注册，一种是接口级注册，在Spring Cloud中，一个应用是一个微服务，而在Dubbo2.7中，一个接口是一个微服务。

所以，Spring Cloud在进行服务注册时，是把应用名以及应用所在服务器的IP地址和应用所绑定的端口注册到注册中心，相当于key是应用名，value是ip+port，而在Dubbo2.7中，是把接口名以及对应应用的IP地址和所绑定的端口注册到注册中心，相当于key是接口名，value是ip+port。

所以在Dubbo2.7中，一个应用如果提供了10个Dubbo服务，那么注册中心中就会存储10对keyvalue，而Spring Cloud就只会存一对keyvalue，所以以Spring Cloud为首的应用级注册是更加适合的。

所以Dubbo3.0中将注册模型也改为了应用级注册，提升效率节省资源的同时，通过统一注册模型，也为各个微服务框架的互通打下了基础。

新一代RPC协议-Triple协议

在上节课中，服务消费者是通过发送一个Invocation对象来表示要调用的是哪个接口中的哪个方法，我们是直接把Invocation对象进行JDK序列化得到字节流后然后发送出去的，那如果现在不用JDK序列化呢，比如还有很多序列化的方式，比如JSON、Hessian等等。

此时服务消费者最好能在发送的请求中，能标记所使用的序列化方式，这个标记是不能放在Invocation对象中的，因为这个标记指的就是Invocation对象的序列化方法，服务端接收到字节流后，首先得能拿到序列化标记，看采用的是什么序列化方法，再解析反序列化。

如果我们通过HTTP协议（特指HTTP1.x，HTTP2后面分析），那实现起来就比较方便，把序列化标记放在请求头，Invocation对象序列化之后的结果放在请求体，服务端收到HTTP请求后，就先解析请求头得到序列化标记，再取请求体进行反序列化。

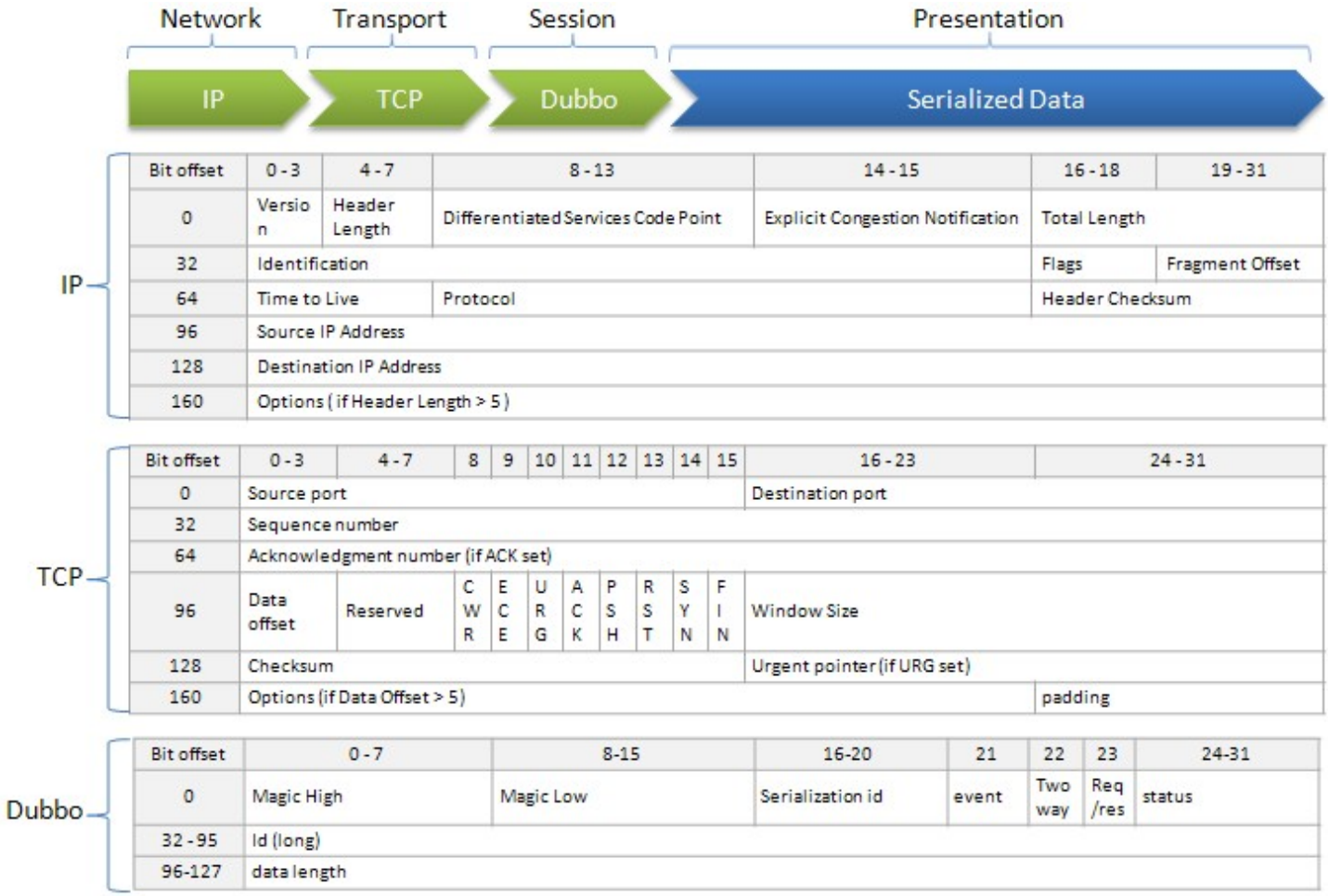
但是Dubbo觉得用HTTP1.x协议性能太低了，原因在于：

1. HTTP1.x协议中，多余无用的字符太多了，比如回车符、换行符，这每一个字符都会占用一个字节，这些字节占用了网络带宽，降低了网络IO的效率
2. HTTP1.x协议中，一条Socket连接，一次只能发送一个HTTP请求，因为如果连续发送两个HTTP请求，然后收到了一个响应，那怎么知道这个响应对应的是哪个请求呢，这样导致Socket连接的利用低，并发、吞吐量低。

所以有了dubbo协议，它就是为了解决上面两个问题，协议，描述的就是一份数据长什么样子，HTTP协议也是一样，描述的就是一个HTTP请求长什么样子，以什么开始，到哪里结束。

比如HTTP协议就描述了，从第一个字节开始，遇到第一个空格符时，那就是表示前面每个字节对应的字符组合成的字符串就表示请求方法（字符编码为ascii，一个字符对应一个字节），紧接着继续解析字节，将会按HTTP协议格式解析出请求行、请求头，解析完请求头之后，取出content-length对应的value，该value表示请求体的字节数，所以直接再获取content-length个字节，就表示获取到了请求体（Invocation对象序列化之后的字节），从而一个HTTP请求就获取出来，下一个字节就是另外一个HTTP请求了。

而dubbo协议也有自己的格式，比如：



源码中ExchangeCodec类中。

dubbo协议在Dubbo框架内使用还是比较舒服的，并且dubbo协议相比于http1.x协议，性能会更好，因为请求中没有多余的无用的字节，都是必要的字节，并且每个Dubbo请求和响应中都有一个请求ID，这样可以基于一个Socket连接同时发送多个Dubbo请求，不用担心请求和响应对不上，所以dubbo协议成为了Dubbo框架中的默认协议。

但是dubbo协议一旦涉及到跨RPC框架，比如一个Dubbo服务要调用gPRC服务，就比较麻烦了，因为发一个dubbo协议的请求给一个gPRC服务，gPRC服务只会按照gRPC的格式来解析字节流，最终肯定会解析不成功的。

dubbo协议虽好，但是不够通用，所以这就出现了Triple协议，Triple协议是基于HTTP2，没有性能问题，另外HTTP协议非常通用，全世界都认它，兼容起来也比较简单，而且还有很多额外的功能，比如流式调用。

关于Triple协议，后面会深入的讲，今天先分析到这，大家只需要知道它是基于HTTP2的，性能更高，并且兼容了gRPC，还支持流式调用，就可以了。

大概对比一下triple、dubbo、rest这三个协议

- triple协议基于的是HTTP2，rest协议目前基于的是HTTP1，都可以做到跨语言。
- triple协议兼容了gPRC（Triple服务可以直接调用gRPC服务，反过来也可以），rest协议不行
- triple协议支持流式调用，rest协议不行
- rest协议更方便浏览器、客户端直接调用，triple协议不行（原理上支持，当得对triple协议的底层实现比较熟悉才行，得知道具体的请求头、请求体是怎么生成的）
- dubbo协议是Dubbo3.0之前的默认协议，triple协议是Dubbo3.0之后的默认协议，优先用Triple协议
- dubbo协议不是基于的HTTP，不够通用，triple协议底层基于HTTP所以更通用（比如跨语言、跨异构系统实现起来比较方便）
- dubbo协议不支持流式调用

Triple协议的流式调用

当使用Triple协议进行RPC调用时，支持多种方式来调用服务，只不过在服务接口中要定义不同的方法，比如：

```
public interface UserService {

    // UNARY
    String sayHello(String name);

    // SERVER_STREAM
    default void sayHelloServerStream(String name, StreamObserver<String> response) {
    }

    // CLIENT_STREAM / BI_STREAM
    default StreamObserver<String> sayHelloStream(StreamObserver<String> response) {
        return response;
    }

}
```

StreamObserver在dubbo-common模块下：

```
<dependencies>
        <dependency>
                <groupId>org.apache.dubbo</groupId>
                <artifactId>dubbo-common</artifactId>
```

```
                <version>3.0.7</version>
        </dependency>
</dependencies>
```

另外在provider和consumer下都要加上triple协议的依赖：

```
<dependency>
            <groupId>org.apache.dubbo</groupId>
            <artifactId>dubbo-rpc-triple</artifactId>
            <version>3.0.7</version>
        </dependency>
```

UNARY

unary，就是正常的调用方法

服务实现类对应的方法：

```
// UNARY
@Override
public String sayHello(String name) {
    return "Hello " + name;
}
```

服务消费者调用方式：

```
String result = userService.sayHello("zhouyu");
```

SERVER_STREAM

服务实现类对应的方法：

```
// SERVER_STREAM
@Override
public void sayHelloServerStream(String name, StreamObserver<String> response) {
    response.onNext(name + " hello");
    response.onNext(name + " world");
    response.onCompleted();
}
```

```
8
```

服务消费者调用方式：

```
userService.sayHelloServerStream("zhouyu", new StreamObserver<String>() {
    @Override
    public void onNext(String data) {
        // 服务端返回的数据
        System.out.println(data);
    }

    @Override
    public void onError(Throwable throwable) {}

    @Override
    public void onCompleted() {
        System.out.println("complete");
    }
});
```

CLIENT_STREAM

服务实现类对应的方法：

```
// CLIENT_STREAM
@Override
public StreamObserver<String> sayHelloStream(StreamObserver<String> response) {
    return new StreamObserver<String>() {
        @Override
        public void onNext(String data) {
            // 接收客户端发送过来的数据，然后返回数据给客户端
            response.onNext("result: " + data);
        }

        @Override
        public void onError(Throwable throwable) {}

        @Override
        public void onCompleted() {
```

```
            System.out.println("completed");
        }
    };
}
```

服务消费者调用方式：

```
StreamObserver<String> streamObserver = userService.sayHelloStream(new
StreamObserver<String>() {
    @Override
    public void onNext(String data) {
        System.out.println("接收到响应数据：" + data);
    }

    @Override
    public void onError(Throwable throwable) {}

    @Override
    public void onCompleted() {
        System.out.println("接收到响应数据完毕");
    }
});

// 发送数据
streamObserver.onNext("request zhouyu hello");
streamObserver.onNext("request zhouyu world");
streamObserver.onCompleted();
```

BI_STREAM

和CLIENT_STREAM一样

Dubbo3.0跨语言调用

在工作中，我们用Java语言通过Dubbo提供了一个服务，另外一个应用（也就是消费者）想要使用这个服务，如果消费者应用也是用Java语言开发的，那没什么好说的，直接在消费者应用引入Dubbo和服务接口相关的依赖即可。

但是，如果消费者应用不是用Java语言写的呢，比如是通过python或者go语言实现的，那就至少需要满足两个条件才能调用Java实现的Dubbo服务：

1. Dubbo一开始是用Java语言实现的，那现在就需要一个go语言实现的Dubbo框架，也就是现在的dubbo-go，然后在go项目中引入dubbo-go，从而可以在go项目中使用dubbo，比如使用go语言去暴露和使用Dubbo服务。
2. 我们在使用Java语言开发一个Dubbo服务时，会把服务接口和相关类，单独抽象成为一个Maven项目，实际上就相当于一个单独的jar包，这个jar能被Java项目所使用，但不能被go项目所使用，所以go项目中该如何使用Java语言所定义的接口呢？直接用是不太可能的，只能通过间接的方式来解决这个问题，除开Java语言之外，那有没有其他技术也能定义接口呢？并且该技术也是Java和go都支持，这就是protobuf。

protobuf

我们可以通过protobuf来定义接口，然后通过protobuf的编译器将接口编译为特定语言的实现。

在provider项目中定义一个userservice.proto文件，路径为src/main/proto/userservice.proto:

```
syntax = "proto3";

package api;

option go_package = "./;api";

option java_multiple_files = true;
option java_package = "com.zhouyu";
option java_outer_classname = "UserServiceProto";

service UserService {
  rpc GetUser (UserRequest) returns (User) {}
}

// The response message containing the greetings
message UserRequest {
  string uid = 1;
}

// The response message containing the greetings
message User {
  string uid = 1;
  string username = 2;
```

```
}
```

相当于定义了一个HelloService服务，并且定义了一个getUser方法，接收UserRequest类型的参数，返回User类型的对象。

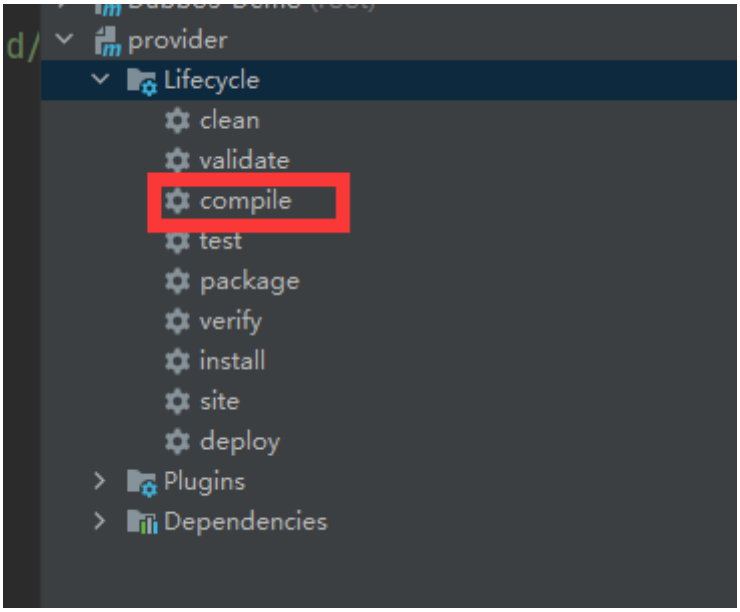
编译成Java

在provider项目中的pom文件中添加相关maven插件：

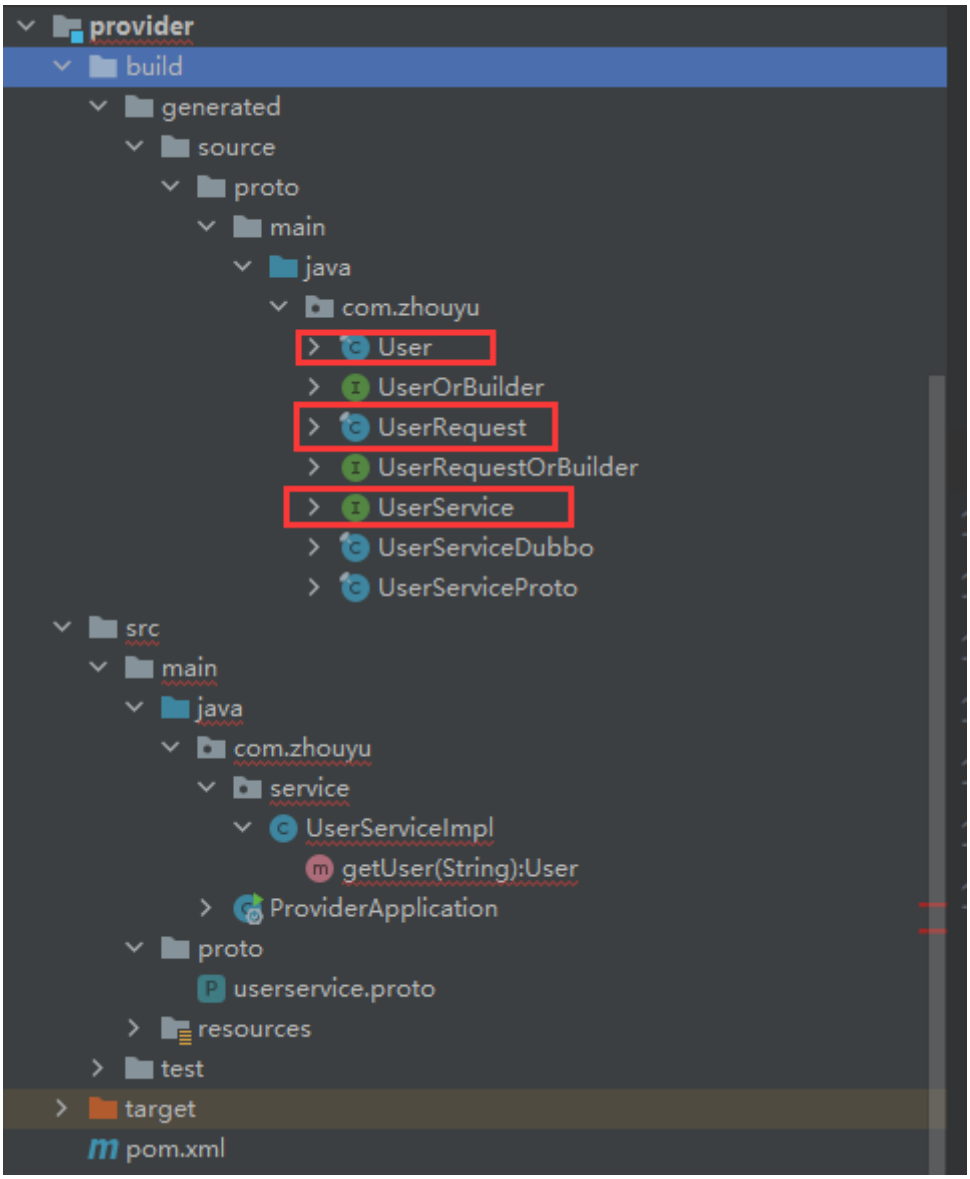
```
<build>
        <extensions>
            <extension>
                <groupId>kr.motd.maven</groupId>
                <artifactId>os-maven-plugin</artifactId>
                <version>1.6.1</version>
            </extension>
        </extensions>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.7.0</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.xolstice.maven.plugins</groupId>
                <artifactId>protobuf-maven-plugin</artifactId>
                <version>0.6.1</version>
                <configuration>

<protocArtifact>com.google.protobuf:protoc:3.7.1:exe:${os.detected.classifier}
</protocArtifact>

<outputDirectory>build/generated/source/proto/main/java</outputDirectory>
                    <clearOutputDirectory>false</clearOutputDirectory>
                    <protocPlugins>
                        <protocPlugin>
```

```
                            <id>dubbo</id>
                            <groupId>org.apache.dubbo</groupId>
                            <artifactId>dubbo-compiler</artifactId>
                            <version>0.0.4.1-SNAPSHOT</version>
<mainClass>org.apache.dubbo.gen.tri.Dubbo3TripleGenerator</mainClass>
                        </protocPlugin>
                    </protocPlugins>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>compile</goal>
                            <goal>test-compile</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>build-helper-maven-plugin</artifactId>
                <version>3.0.0</version>
                <executions>
                    <execution>
                        <phase>generate-sources</phase>
                        <goals>
                            <goal>add-source</goal>
                        </goals>
                        <configuration>
                            <sources>
                                <source>build/generated/source/proto/main/java</source>
                            </sources>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
```

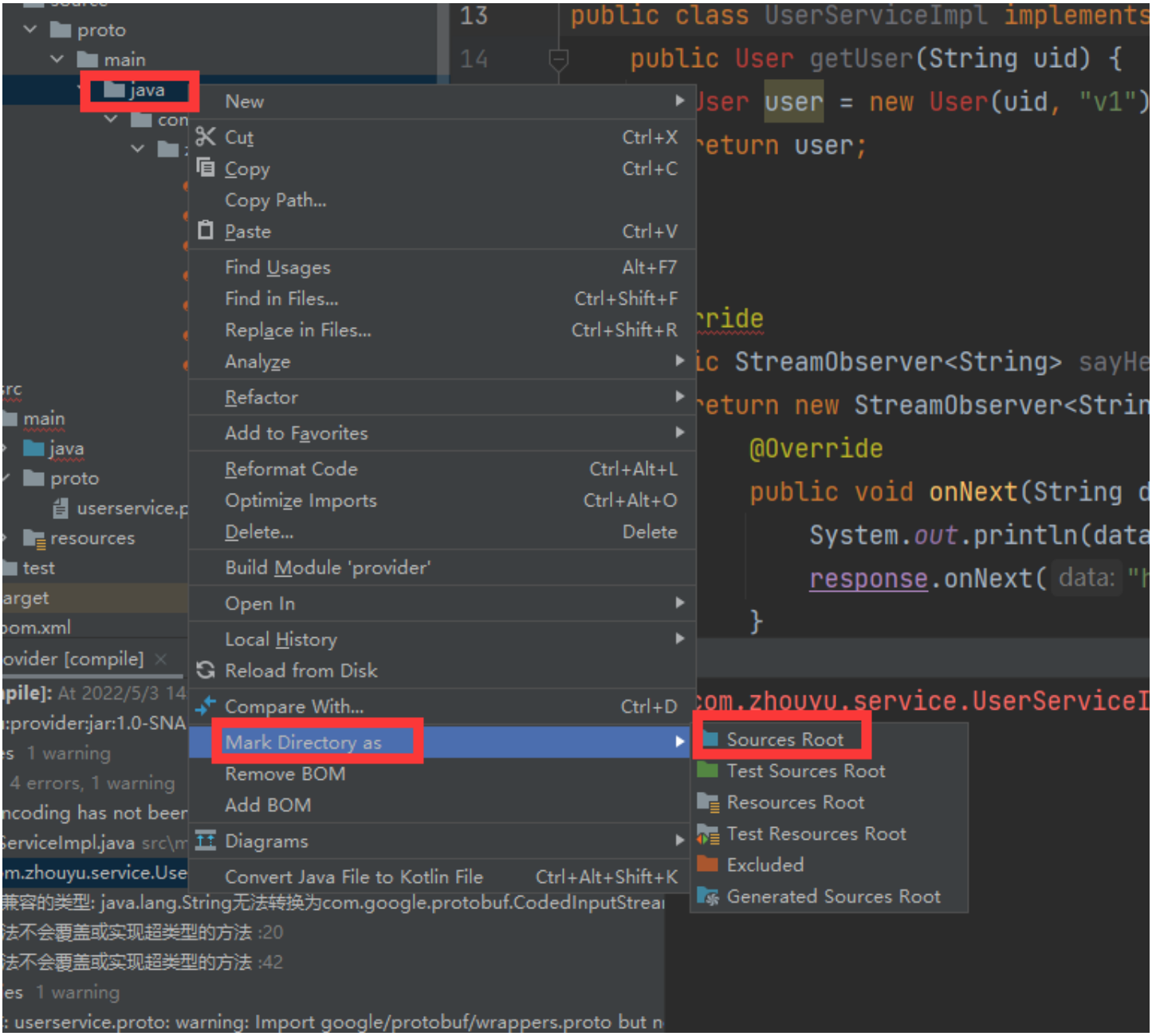
并且可以把common依赖去掉，然后运行provider中lifecycle的compile，就会进行编译了，



并且会编译出对应的接口等信息，编译完成后，会生成一些类：



如果Java没有蓝色的，就



其中就包括了一个UserService接口，所以我们的UserServiceImpl就可以实现这个接口了：

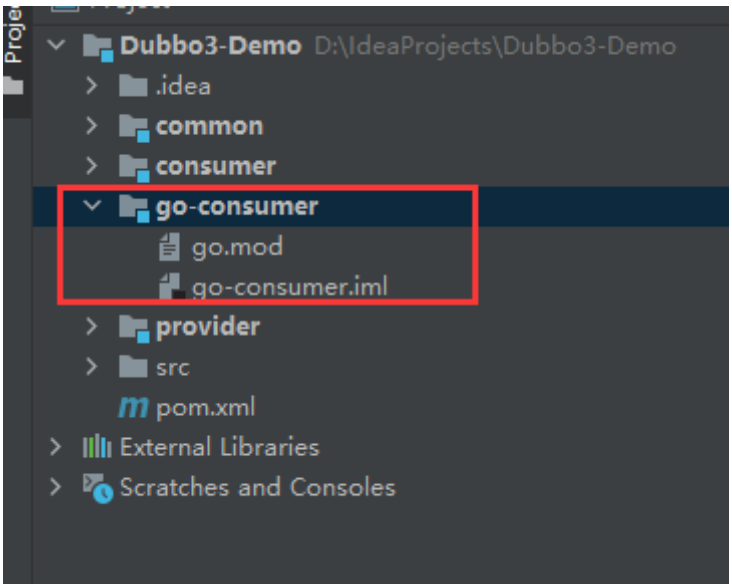
```
@DubboService
public class UserServiceImpl implements UserService {


    public User getUser(UserRequest userRequest) {
        User user =
User.newBuilder().setUid(userRequest.getUid()).setUsername("zhouyu").build();
        return user;
    }

```

```
9 }
```

而对于想要调用UserService服务的消费者而言，其实也是一样的改造，只需要使用同一份userservice.proto进行编译就可以了，比如现在有一个go语言的消费者。

go消费者调用java服务

首先，在IDEA中新建一个go模块：



然后把userservice.proto复制到go-consumer/proto下，然后进行编译，编译成为go语言对应的服务代码，只不过go语言中没有maven这种东西可以帮助我们编译，我们只能用原生的protobuf的编译器进行编译。

这就需要大家在机器上下载、安装protobuf的编译器：protoc

1. 下载地址：https://github.com/protocolbuffers/protobuf/releases/download/v3.20.1/protoc-3.20.1-win64.zip
2. 解压之后，把protoc-3.20.1-win64\bin添加到环境变量中去
3. 在cmd中执行protoc --version，能正常看到版本号即表示安装成功

另外还需要安装go：

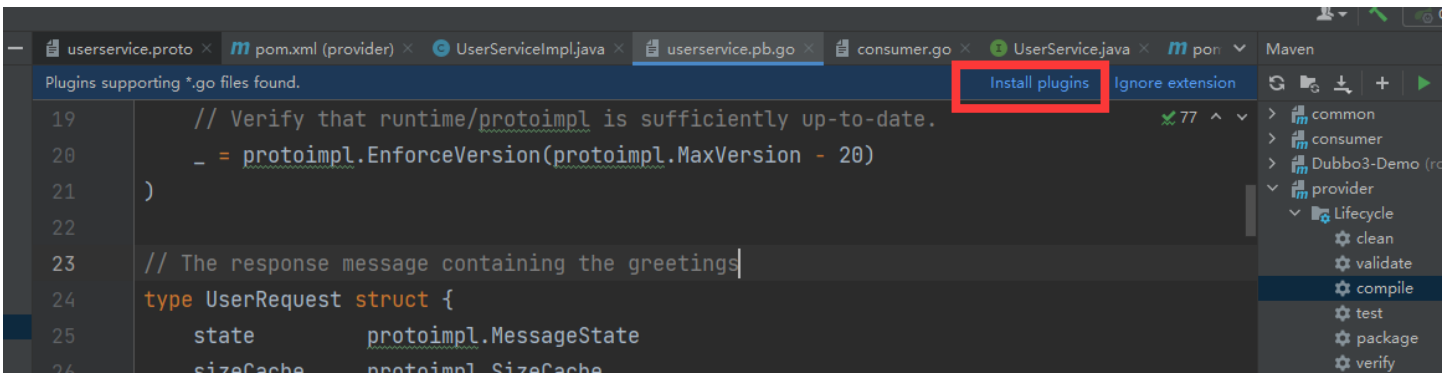
1. 下载地址：https://studygolang.com/dl/golang/go1.18.1.windows-amd64.msi
2. 然后直接下一步安装
3. 在cmd中（新开一个cmd窗口）执行go version，能正常看到版本号即表示安装成功

然后在go-consumer下新建文件夹api，进入到go-consumer/proto下，运行：

```
go env -w GO111MODULE=on
go env -w GOPROXY=
https://goproxy.cn,direct


go get -u github.com/dubbogo/tools/cmd/protoc-gen-go-triple
go install github.com/golang/protobuf/protoc-gen-go
go install github.com/dubbogo/tools/cmd/protoc-gen-go-triple

protoc -I. userservice.proto --go_out=../api --go-triple_out=../api
```

这样就会在go-consumer/api下生成一个userservice.pb.go文件和userservice_triple.pb.go文件

如果IDEA中提示要安装插件，就安装一下：



安装完之后，代码可能会报错，可以在**go-consumer目录**下执行命令下载依赖：

```
go mod tidy
```

然后就可以写go语言的服务消费者了，新建一个consumer.go，内容为：

```
package main

import (
        "context"
        "dubbo.apache.org/dubbo-go/v3/common/logger"
```

```
        "dubbo.apache.org/dubbo-go/v3/config"
        _ "dubbo.apache.org/dubbo-go/v3/imports"
        "go-consumer/api"
)

var userServiceImpl = new(api.UserServiceClientImpl)

// export DUBBO_GO_CONFIG_PATH=conf/dubbogo.yml
func main() {
        config.SetConsumerService(userServiceImpl)
        config.Load()

        logger.Info("start to test dubbo")
        req := &api.UserRequest{
                Uid: "1",
        }

        user, err := userServiceImpl.GetUser(context.Background(), req)

        if err != nil {
                logger.Error(err)
        }

        logger.Infof("client response result: %v\n", user)
}

```

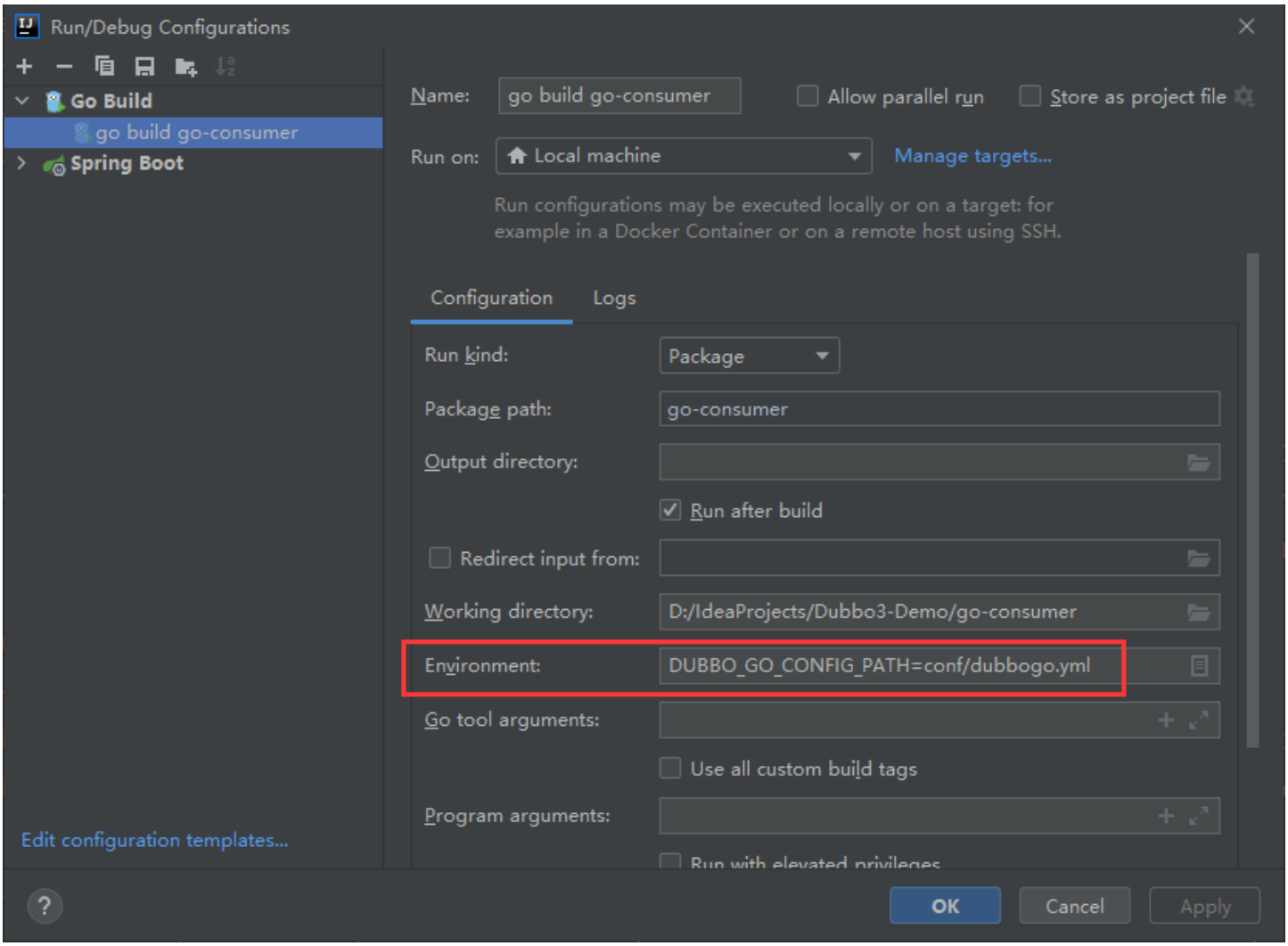
然后在go-consumer下新建conf/dubbogo.yml，用来配置注册中心：

```
dubbo:
  registries:
    demoZK:
      protocol: zookeeper
      address: 127.0.0.1:2181
  consumer:
    references:
      UserServiceClientImpl:
        protocol: tri
        interface: com.zhouyu.UserService
```

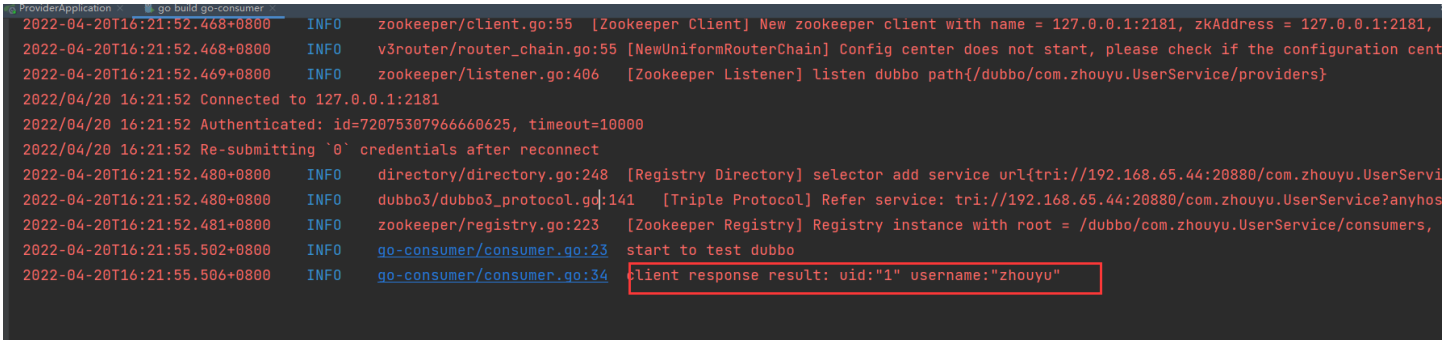
注意这里配置的协议为tri，而不是dubbo，在provider端也得把协议改为tri：

```
server.port=8082

dubbo.application.name=provider-application

dubbo.protocol.name=tri
dubbo.protocol.port=20880

dubbo.registry.address=zookeeper://127.0.0.1:2181
```

然后就可以运行consumer.go了，只不过需要在environment中添加一个参数：



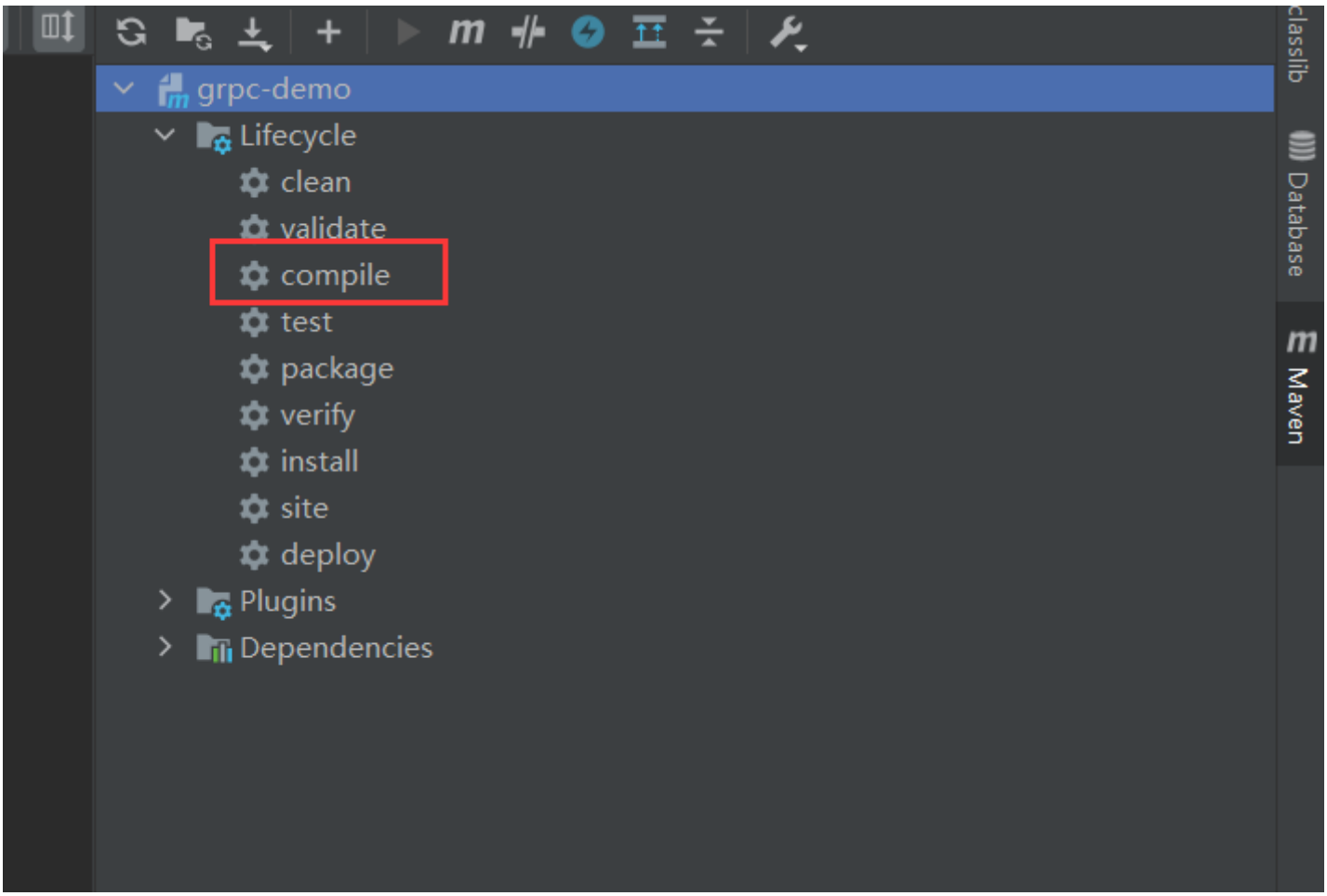
运行成功：



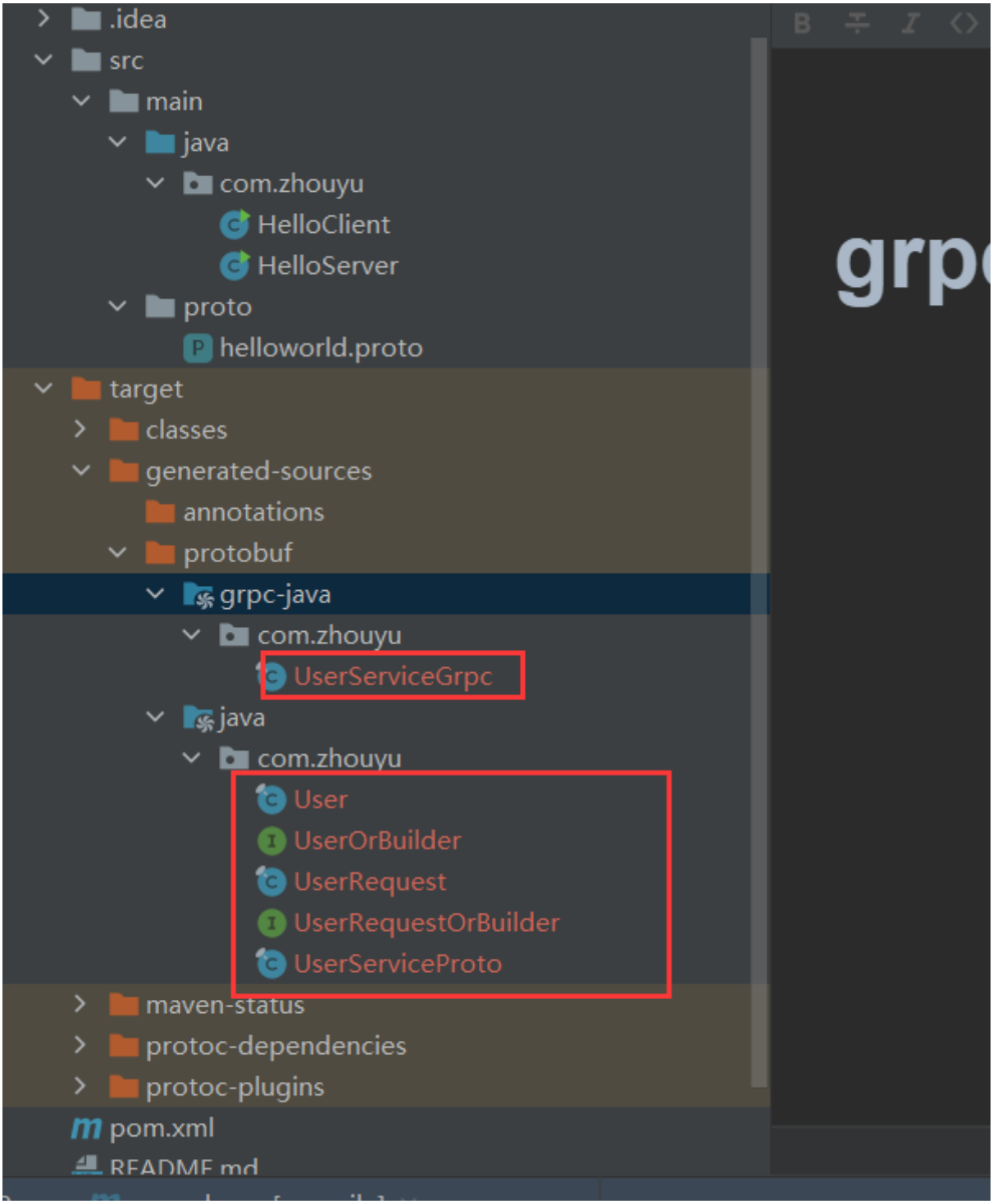
Triple与gRPC互通

git clone：https://gitee.com/archguide/grpc-demo

运行compile：



compile完之后，如果成功，那么检查，看是否生成了如下java文件：



接下来就可以运行HelloServer了，并且可以使用HelloClient来调用gRPC的服务。

HelloServer中相当于在8008端口暴露了一个UserService服务，所以如果希望通过Triple来调用一个gRPC服务，可以这么做：

```
@Service
public class OrderService {

    @DubboReference(protocol = "tri", url = "tri://localhost:8008", proxy =
CommonConstants.NATIVE_STUB)
```

```
    private UserService userService;

    public String createOrder() {
        User user = userService.getUser(UserRequest.newBuilder().build());
        return user.getUsername();
    }
}
```

确定使用tri协议，直连gRPC服务器，设置proxy为CommonConstants.NATIVE_STUB，然后就可以调用gRPC服务了。

反过来，如果现在通过tri协议提供了一个服务，想要通过一个grpc来调用也是可以的，先正常的把dubbo的provider启动起来，此时就可以通过tri协议以及20881端口库来调用UserService中的方法了，此时修改grpc-demo中的HelloClient，改成连接20881，就可以调用tri协议的UserServcie了。

之所以这么方便，是因为tri兼容了grpc，兼容的意思是，tri协议在发送请求和发送响应时，都是按照grpc的格式来发送的，比如在请求头和响应头中设置grpc能识别的信息。

Dubbo3.0与Spring Cloud互通

目前Dubbo3.0和Spring Cloud之间的互通，还没做到特别方便，比如我们知道SpringCloud在使用的过程中，需要程序员知道某个服务的controller访问路径，就算用openFeign也避免不了，比如：

```
@FeignClient("spring-cloud-provider-application")
public interface HelloService {

    @RequestMapping("/hello")
    public String hello();
}

```

需要调用一个SpringCloud的微服务时，得在消费端应用中自己去确定**要调用的应用名，以及具体的controller路径。**

所以，现在如果想要在一个Dubbo应用去调用另一个Spring Cloud应用中的某个Service中的方法，那也得程序员在Dubbo应用去指定调用controller地址（程序员不指定，就不知道该访问哪个地址了，除非Spring Cloud能够在某个地方，把某个Service的某个方法和它对应的controller地址进行映射，这就需要在Spring Cloud侧去做改动了）。

还有一点，要调用Spring Cloud的服务，得用http协议，那tri协议行不行呢？原理上行，但是我们在用tri协议去调用另外一个服务时，并不能去指定controller地址，得用rest协议，底层也是http协议，比如：

就像openFeign一样，定义要调用的信息：

```
@Path("/")
@Consumes
public interface HelloService {

    @GET
    @Path("hello")
    String hello();
}
```

相当于controller的路径"/hello"

然后：

```
@Service
public class OrderService {


//    调用spring-cloud-provider-application上的/hello
//    @DubboReference(providedBy = "spring-cloud-provider-application", protocol = "rest")

//    调用localhost:7070上的/hello
    @DubboReference(url = "rest://localhost:7070", protocol = "rest")
    private HelloService helloService;

    public String createOrder() {
        return helloService.hello();
    }
}
```

本质上，以上两种方式都可以使用，但是第一种目前有问题，虽然Dubbo3.0也支持了应用注册，正常来说可以根据应用名找到应有的ip+port，但是源码实现上有bug。

通过这种方式，可以实现Dubbo服务调用Spring Cloud服务。

那反过来，如果想实现Spring Cloud服务调用Dubbo服务，这就需要考虑更多了，首先得看要调用的Dubbo服务支持什么协议，比如如果Dubbo服务只支持dubbo协议，那么Spring Cloud应用只能引入Dubbo的依赖，然后配置Dubbo相关的配置，然后引入common模块，然后通过@DubboReference来使用。如果支持的是tri协议，那也是一样的（底层原理是这样，但是真正的Spring Cloud+Dubbo的整合停复杂，Spring Cloud系列课中会讲）

只有如果Dubbo服务支持rest协议调用，那么SpringCloud应用就不需要引入Dubbo的依赖，直接使用Feign就可以完成调用，比如：

Dubbo服务支持rest协议：

```
@DubboService(protocol = "rest")
@Produces
@Path("/")
public class HelloServiceImpl implements HelloService{

    @Override
        @GET
    @Path("hello")
    public String hello() {
        return "dubbo hello";
    }
}
```

Spring Cloud定义要调用的服务（也得指定url，不能经注册中心服务发现，有bug）：

```
@FeignClient(name = "dubbo-provider-application", url = "localhost:20881")
public interface HelloService {

    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    public String hello();
}
```