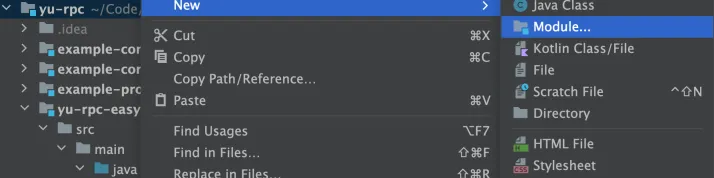
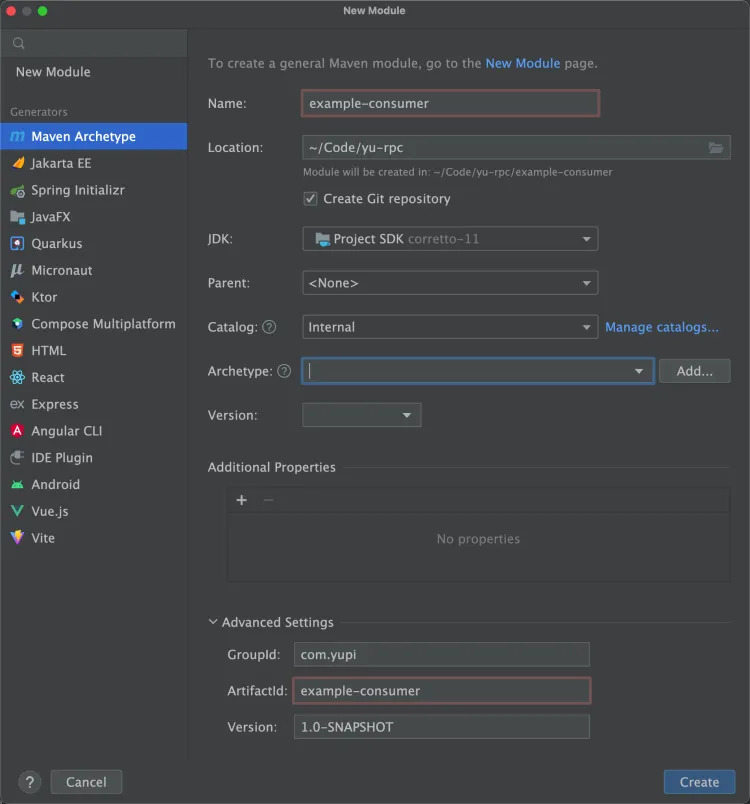
### 1.Project preparation

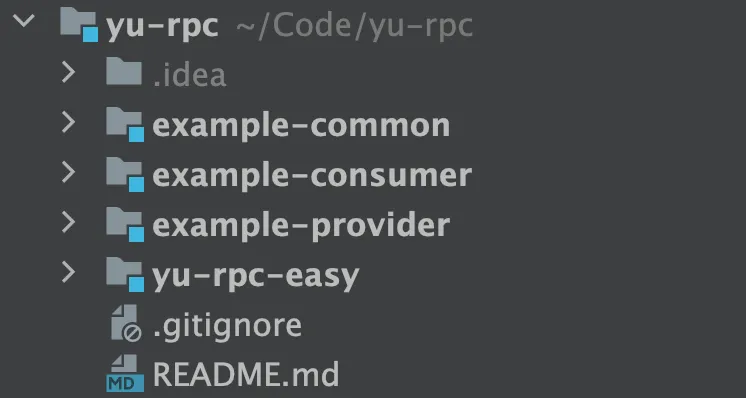
#### 1.1Project initialization

First create a project root directory yu-rpc, and then use the IDEA development tool to create several Maven modules in sequence.





The project directory of the entire basic RPC framework is as follows:



Function descriptions of several modules:

1.example-common: Common dependencies of the example code, including interfaces, Model, etc.

2.example-consumer: Example service consumer code

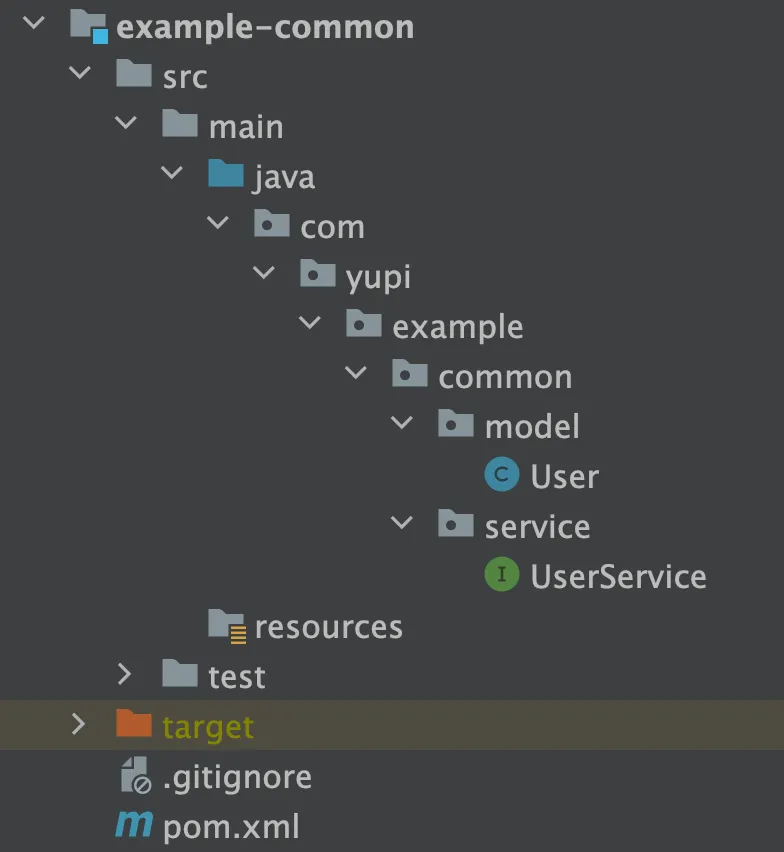
3.example-provider: Example service provider code

4.yu-rpc-easy: Simple RPC framework

#### 1.2example-common

Public modules need to be introduced by consumers and service providers at the same time, mainly to write interfaces and data models related to services.

The structure of the entire module is as follows:



1)Write the user entity class User:

Note that the object needs to implement the serialization interface to provide support for subsequent network transmission serialization.



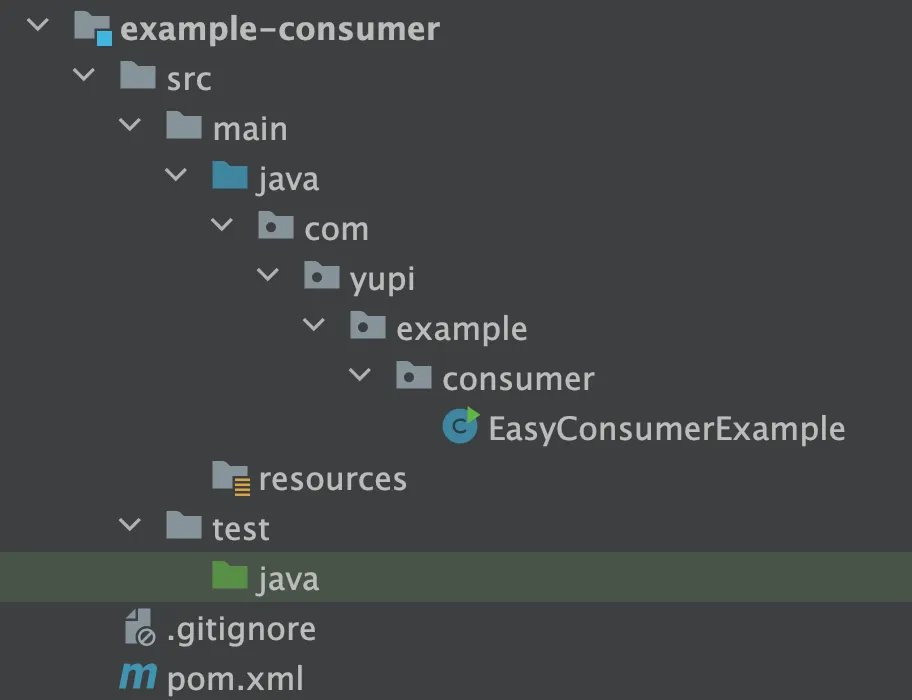
2)Write the user service interface UserService and provide a method to obtain users:



#### 1.3example-consumer

Service consumers are modules that need to call services.

The module directory is as follows:



1) Create the service consumer startup class EasyConsumerExample and write the code to call the interface.

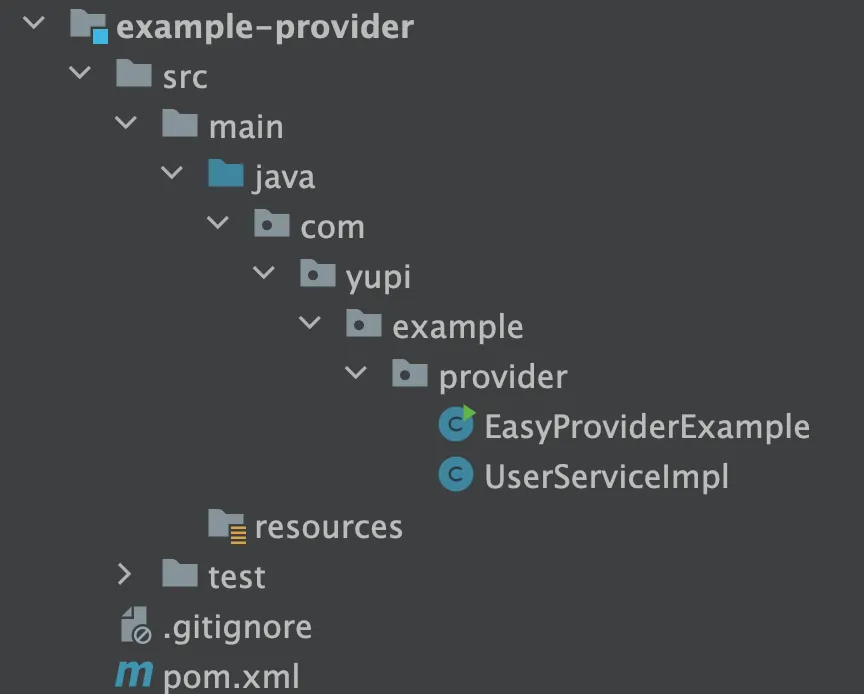


**It should be noted** that the userService instance cannot be obtained now, so it is reserved as null. Our subsequent goal is to quickly obtain a proxy object that supports remote invocation of the service provider through the RPC framework, and call UserService methods just like calling local methods.

#### 1.4example-provider

Service providers are modules that actually implement the interface.

The module directory is as follows

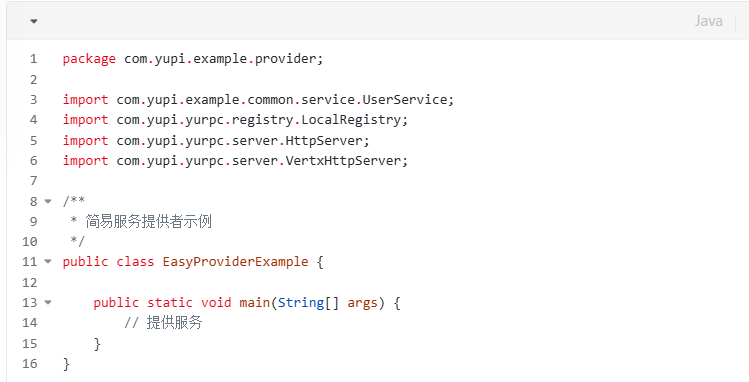


1) Write a service implementation class to implement the user service interface defined in the public module.

The function is to print the user's name and return the user object in the parameter.



1. Write the service provider startup class EasyProviderExample, and then write the code to provide services in the main method of this class.



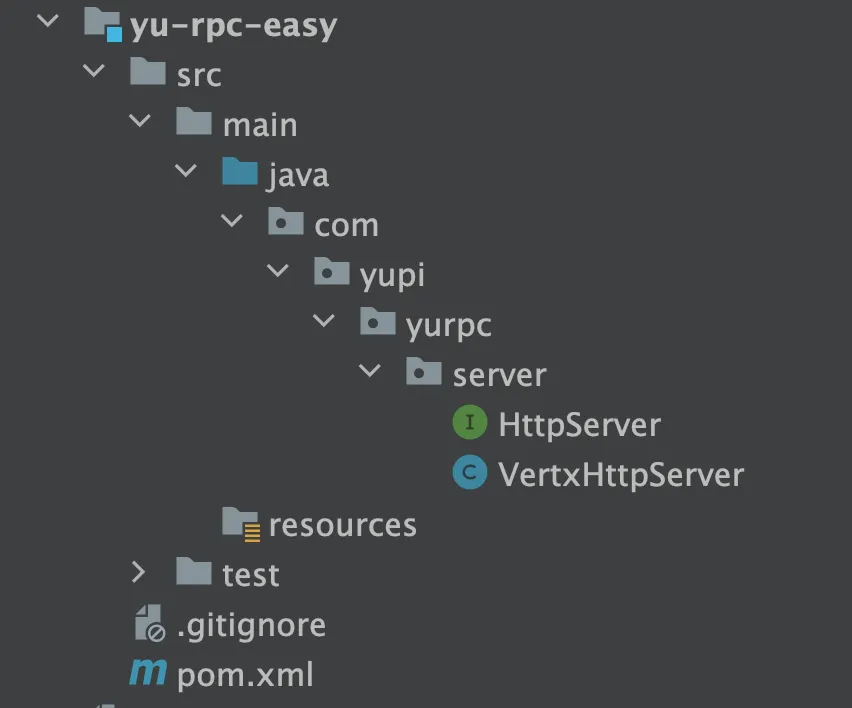
### 2.web server

Next, we need to let the service provider provide remote access services. Then, you need a web server that can accept and process requests and return responses.

There are many choices for web servers, such as Spring Boot's built-in Tomcat, NIO framework Netty and Vert.x, etc.

Here I try to use the high-performance NIO framework Vert.x as the web server of the RPC framework.

The module directory is as follows:



1)Write a web server interface HttpServer and define a unified method to start the server to facilitate subsequent expansion, such as implementing multiple different web servers.



2) Write the web server VertxHttpServer based on Vert.x, which can listen to the specified port and process requests.



3) Verify that the web server can start successfully and accept requests.

Modify the EasyProviderExample class of the sample service provider module and write the code to start the web service, as follows:

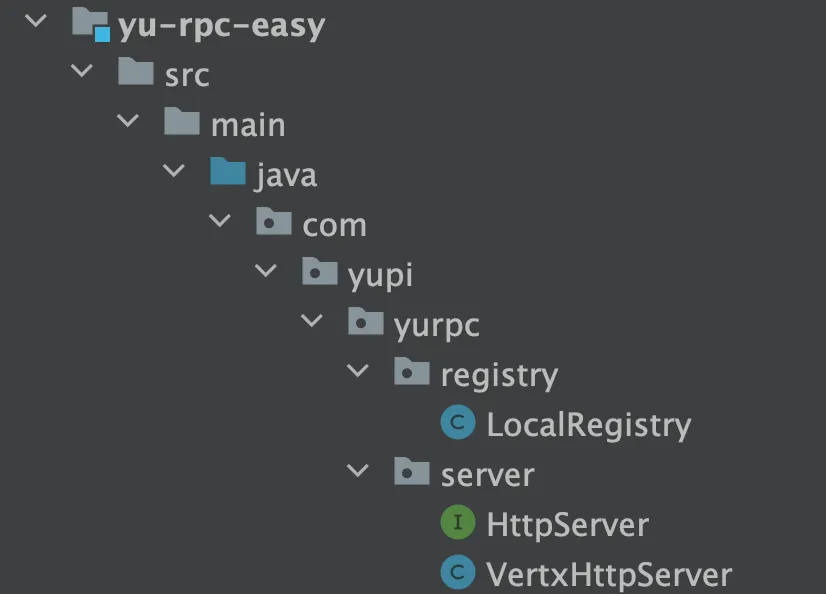


Access localhost:8080 through a browser to see the output text.

### 3.Local Service Registry

The simple RPC framework we are working on now is mainly for running through the process, so there is no need for a third-party registration center for the time being. Just register the service locally with the service provider.

Create the local service register LocalRegistry in the RPC module. The current directory structure is as follows:



Use thread-safe ConcurrentHashMap to store service registration information. The key is the service name and the value is the implementation class of the service. After that, you can get the corresponding implementation class based on the name of the service to be called, and then call the method through reflection.



When the service provider starts, it needs to register the service in the registrar. Modify the EasyProviderExample code as follows:

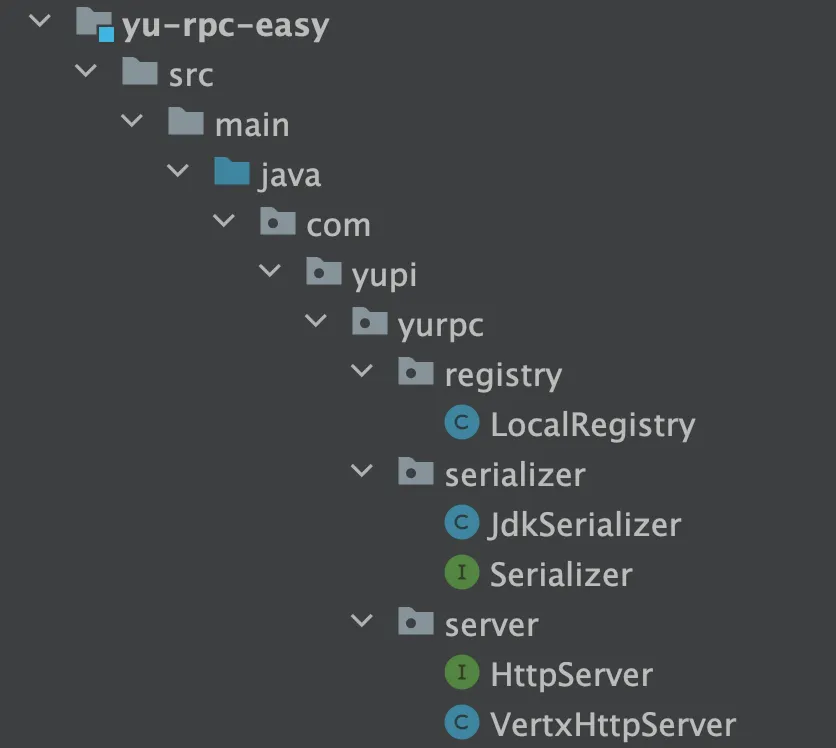


### 4.Serializer

After the service is registered locally, we can take out the implementation class and call the method based on the request information.

But before writing the logic to handle requests, we must first implement the serializer module. Because whether it is a request or a response, it will involve the transmission of parameters. Java objects live in the JVM virtual machine. If you want to store and access them in other locations, or transmit them over the network, you need to serialize and deserialize them.

The current directory structure is as follows:



What are serialization and deserialization?

Serialization: Convert Java objects into transferable byte arrays.

Deserialization: Convert byte array to Java object.

There are many different serialization methods, such as Java native serialization, JSON, Hessian, Kryo, protobuf, etc.

For the convenience of implementation, the Java native serializer is selected here.

1) Write the serialization interface Serializer in the RPC module, which provides two methods of serialization and deserialization to facilitate subsequent expansion of more serializers.

code show as below:



2) Implement JdkSerializer based on Java’s own serializer. The code is as follows:



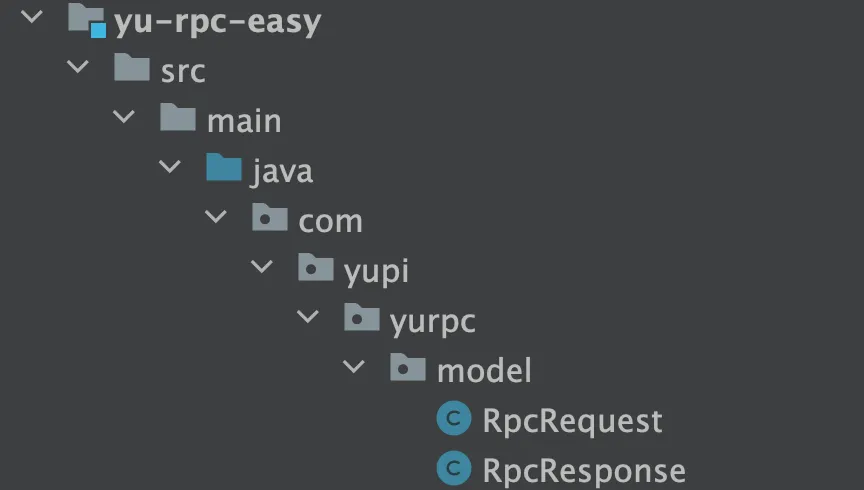
I copied it from the Internet totally. I don’t understand it yet and I can’t write yet.

### 5.Provider handling call - request handler

The request processor is the key to the implementation of the RPC framework. Its role is to process the received request, find the corresponding service and method according to the request parameters, implement the call through reflection, and finally encapsulate the return result and respond to the request.

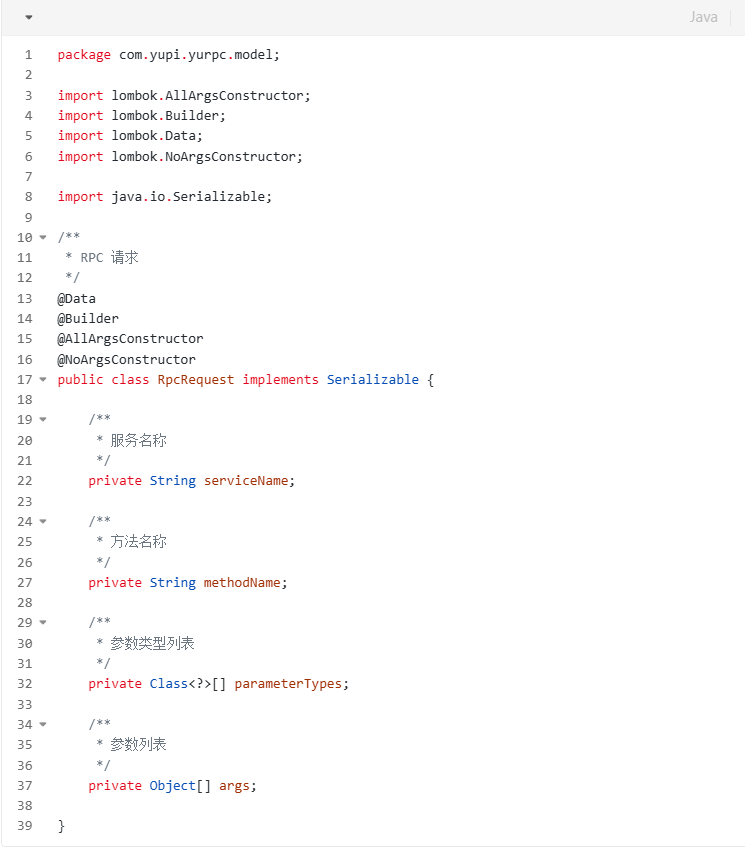
1) Write request and response encapsulation classes in the RPC module.

The directory structure is as follows:



The function of the request class RpcRequest is to encapsulate the information required for the call, such as service name, method name, call parameter type list, and parameter list. These are parameters required by the Java reflection mechanism.

code show as below:

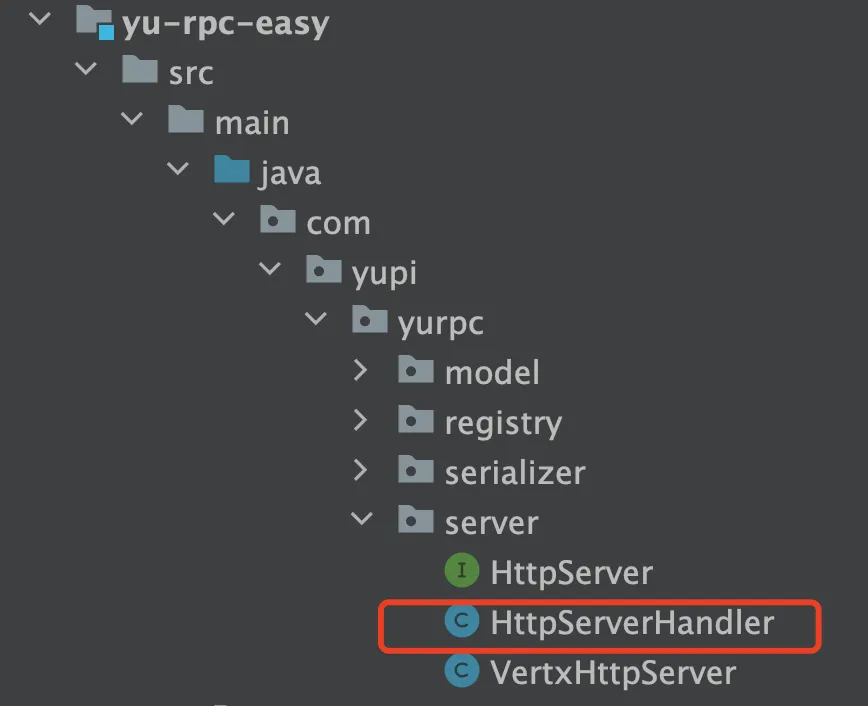


The function of the response class RpcResponse is to encapsulate the return value obtained by calling the method, as well as the calling information (such as exceptions), etc.

code show as below:



2) Write the request handler HttpServerHandler.



The business process is as follows:

1.Deserialize the request into an object and get the parameters from the request object.

2.Obtain the corresponding service implementation class from the local registrar according to the service name.

3.Call the method through the reflection mechanism and get the return result.

4.Encapsulate and serialize the returned results and write them into the response.

This part is still under debugging

3)Bind the request processor to HttpServer.

Modify the code of VertxHttpServer and bind the request handler through server.requestHandler.

The modified code is as follows:



At this point, the service provider module of the RPC framework has been introduced and can accept requests and complete service calls.

### 6.Consumer initiates call - proxy

During the project preparation stage, we have reserved a piece of code for calling the service. As long as we can obtain the UserService object (implementation class), we can run the entire process.

But where does the implementation class of UserService come from?

You can't copy and paste the UserServiceImpl of the service provider into the consumer module, right? What else does an RPC framework need to be able to do that? In a distributed system, when we call interfaces provided by other projects or teams, we generally only focus on the request parameters and response results, not the specific implementation.

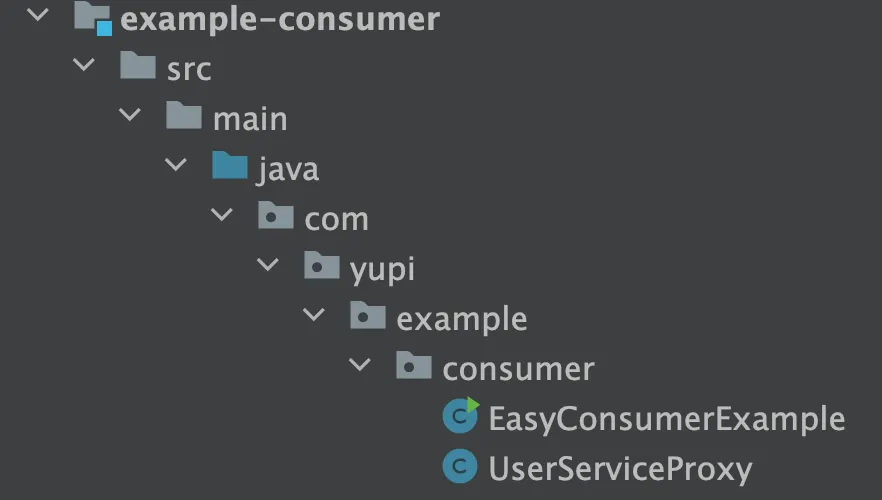
As mentioned in the previous architecture, we can simplify consumer calls by generating proxy objects.

The implementation methods of agents are roughly divided into two categories: static agents and dynamic agents.

#### 6.1static proxy

Static proxy refers to writing a proxy class for each specific type of interface or object.

For example, in the example-consumer module, create a static proxy UserServiceProxy to implement the UserService interface and getUser method.



However, when implementing the getUser method, instead of copying and pasting the code in the service provider UserServiceImpl, you need to construct an HTTP request to call the service provider.

It should be noted that the parameters must be serialized before sending the request. The code is as follows:



Then modify EasyConsumerExample, create a new proxy object and assign it to userService to complete the call:



Although static proxy is easy to understand (just write an implementation class), its shortcomings are also obvious. If we want to write an implementation class for each service interface, it will be very troublesome. The flexibility of this proxy method is very poor!

So in the RPC framework, we will use dynamic proxy.

#### 6.2dynamic proxy

The function of a dynamic proxy is to automatically generate a proxy object based on the type of object to be generated.

Commonly used dynamic proxy implementation methods include JDK dynamic proxy and dynamic proxy generated based on bytecode (such as CGLIB). The former is simple and easy to use and does not require the introduction of additional libraries, but the disadvantage is that it can only proxy interfaces; the latter is more flexible and can proxy any class, but its performance is slightly lower than JDK dynamic proxy.

Here we use JDK dynamic proxy.

This part is under construction