

1_简易版_RPC_框架开发

仅供 编程导航 <<https://www.code-nav.cn/post/1816420035119853569>> 内部成员观看，请勿对外分享！

大家好，我是程序员鱼皮。这次带大家做一个轮子类项目 —— 从 0 到 1 开发 RPC 框架。

很多同学听到“开发框架”可能会有点胆怯，但其实开发 RPC 框架并不难，最多几个小时就能学会核心流程！能够快速给简历增加一个区别于增删改查的项目。而且，开发 RPC 框架涉及很多常用的技术知识点、还能学习到很多架构设计方面的思路 and 技巧。因此，强烈建议所有后端方向的同学，动手做个自己的 RPC 框架。

本项目代码开源：<https://github.com/liyupi/yu-rpc> <<https://github.com/liyupi/yu-rpc>>

一、基本概念

什么是 RPC？

专业定义：RPC (Remote Procedure Call) 即远程过程调用，是一种计算机通信协议，它允许程序在不同的计算机之间进行通信和交互，就像本地调用一样。

简单理解，新开了一家卖鱼皮的熟食店，现在你作为消费者想要把鱼皮购买到家。如果是以前，你只能自己跑腿到线下店铺购买，耗时耗力。但现在有了手机、网络、外卖平台，你只需要在家动动手指，就能点个外卖让骑手把鱼皮配送到家，你不需要关注网络是怎么传输的、外卖平台是怎么操作的、骑手小哥是怎么配送的，只负责享受鱼皮就行了。

为什么需要 RPC？

回到 RPC 的概念，RPC 允许一个程序（称为服务消费者）像调用自己程序的方法一样，调用另一个程序（称为服务提供者）的接口，而不需要了解数据的传输处理过程、底层网络通信的细节等。这些都会由 RPC 框架帮你完成，使得开发者可以轻松调用远程服务，快速开发分布式系统。

举个例子，现在有个项目 A 提供了点餐服务，项目 B 需要调用点餐服务完成下单。

点餐服务和接口的示例伪代码如下：

```
1 interface OrderService {  
2     // 点餐, 返回 orderId  
3     long order(参数1, 参数2, 参数3);  
4 }
```

如果没有 RPC 框架, 项目 B 怎么调用项目 A 的服务呢?

首先, 由于项目 A 和项目 B 都是独立的系统, 不能像 SDK 一样作为依赖包引入。那么就需要项目 A 提供 web 服务, 并且编写一个点餐接口暴露服务, 比如访问 `http://yupi.icu` 就能调用点餐服务; 然后项目 B 作为服务消费者, 需要自己构造请求, 并通过 HttpClient 请求上述地址。如果项目 B 需要调用更多第三方服务, 每个服务和方法的调用都编写一个 HTTP 请求, 那么会非常麻烦!

示例伪代码如下:

```
1 url = "http://yupi.icu"  
2 req = new Req(参数1, 参数2, 参数3)  
3 res = httpClient.post(url).body(req).execute()  
4 orderId = res.data.orderId
```

而有了 RPC 框架, 项目 B 可以通过一行代码完成调用!

示例伪代码如下:

```
1 orderId = orderService.order(参数1, 参数2, 参数3)
```

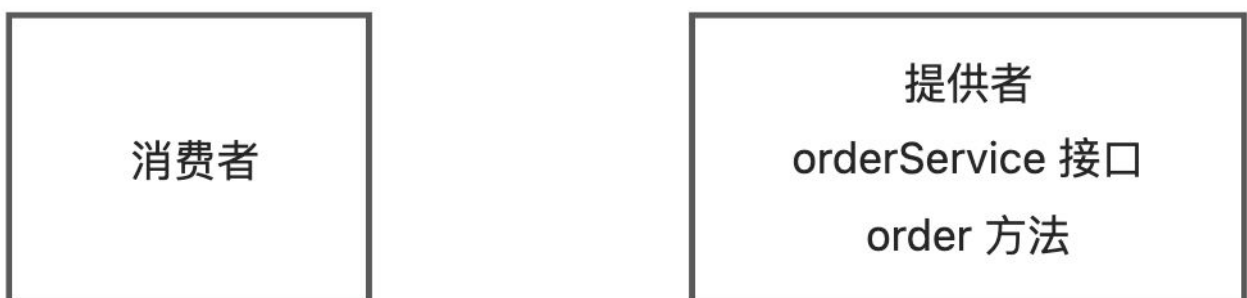
看起来就跟调用自己项目的方法没有任何区别! 是不是很丝滑?

二、RPC 框架实现思路

基本设计

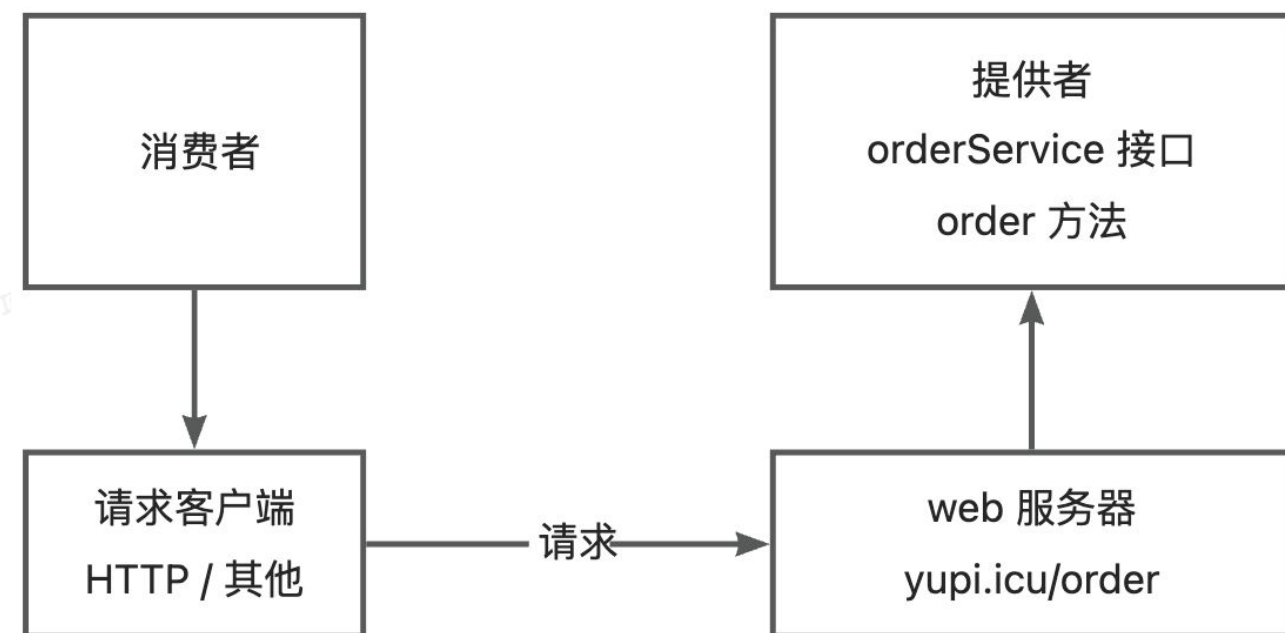
RPC 框架为什么能帮我们简化调用? 如何实现一个 RPC 框架呢?

其实很简单, 开局一张图, 有服务消费者和服务提供者两个角色:



消费者想要调用提供者，就需要提供者启动一个 `web 服务`，然后通过 `请求客户端` 发送 HTTP 或者其他协议的请求来调用。

比如请求 `yupi.icu/order` 地址后，提供者会调用 `orderService` 的 `order` 方法：

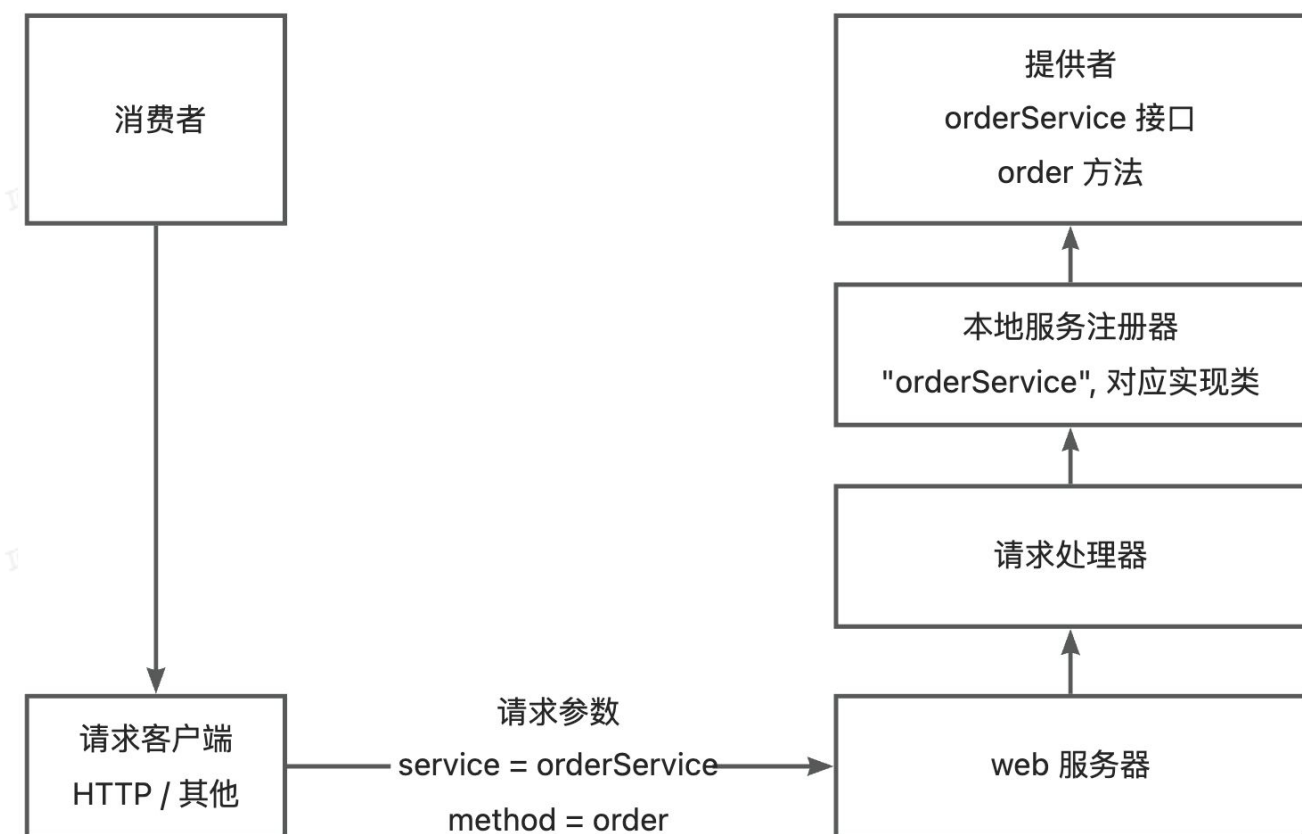


但如果提供者提供了多个服务和方法，每个接口和方法都要单独写一个接口？消费者要针对每个接口写一段 HTTP 调用的逻辑么？

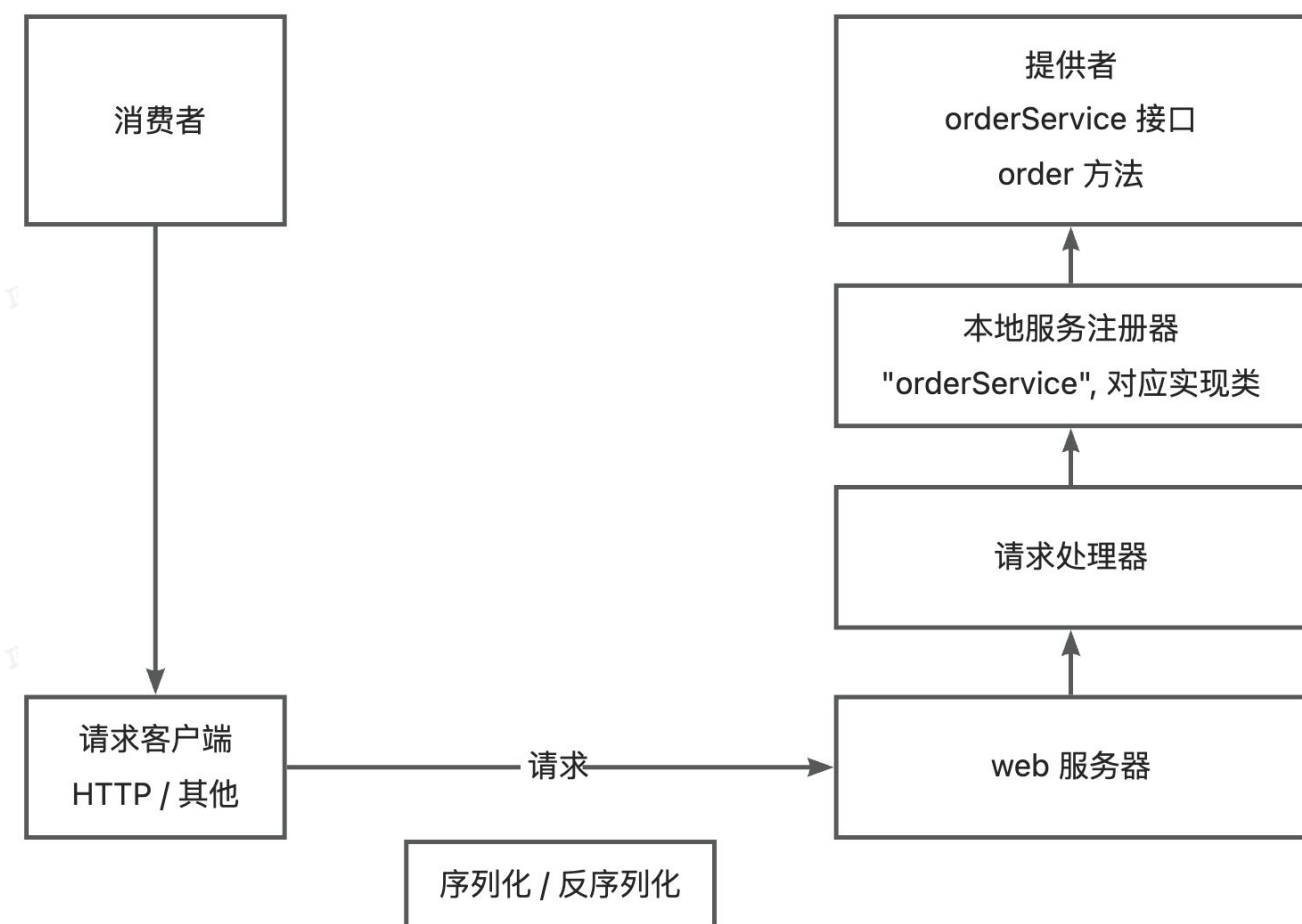
其实可以提供一个统一的服务调用接口，通过 `请求处理器` 根据客户端的请求参数来进行不同的处理、调用不同的服务和方法。

可以在服务提供者程序维护一个 `本地服务注册器`，记录服务和对应实现类的映射。

举个例子，消费者要调用 `orderService` 服务的 `order` 方法，可以发送请求，参数为 `service=orderService,method=order`，然后请求处理器会根据 `service` 从服务注册器中找到对应的服务实现类，并且通过 Java 的反射机制调用 `method` 指定的方法。



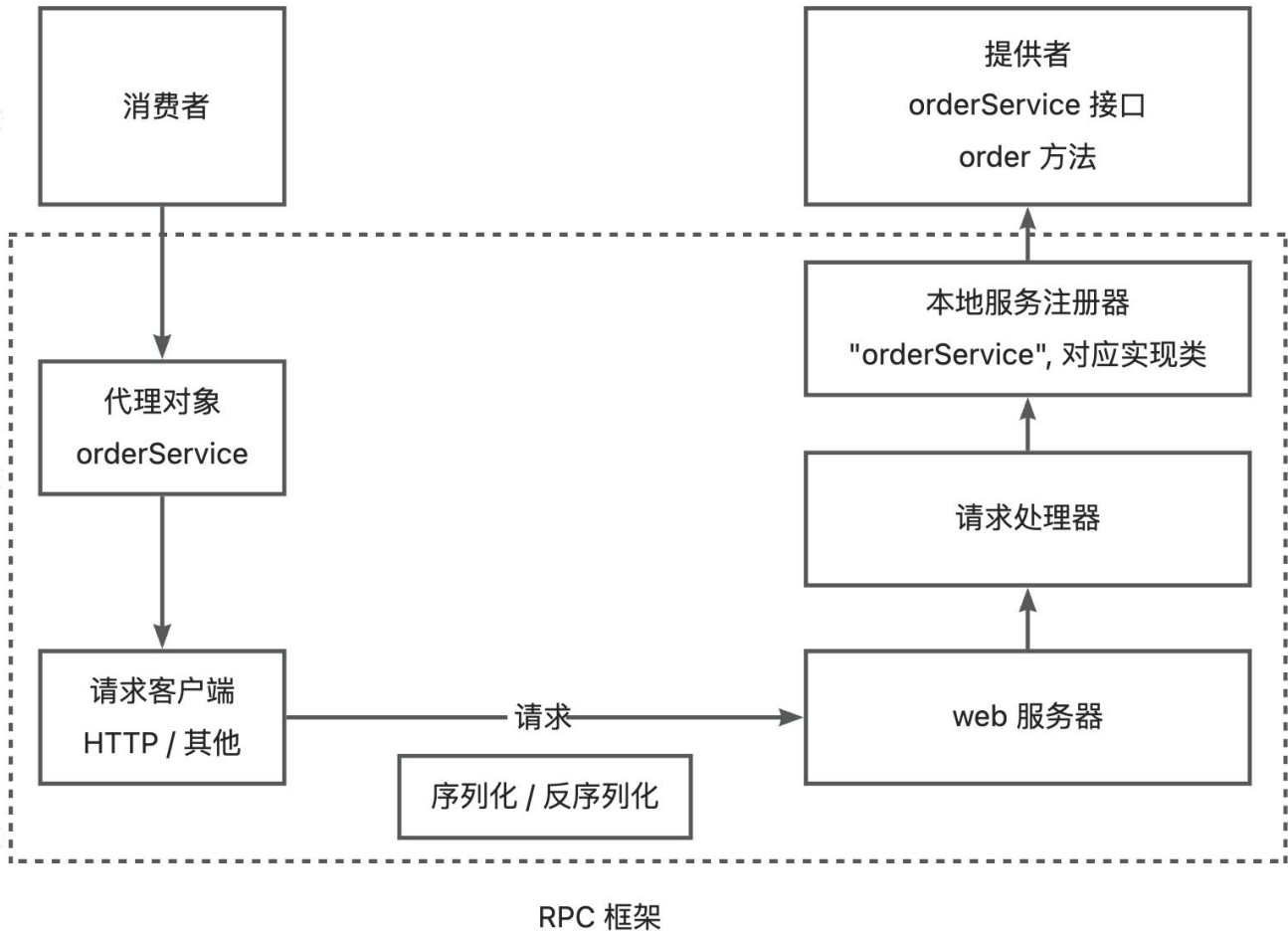
需要注意的是，由于 Java 对象无法直接在网络中传输，所以要对传输的参数进行 **序列化** 和 **反序列化**。



为了简化消费者发请求的代码，实现类似本地调用的体验。可以基于代理模式，为消费者要调用的接口生成一个代理对象，由代理对象完成请求和响应的过程。

所谓代理，就是有人帮你做一些事情，不用自己操心。

至此，一个最简易的 RPC 框架架构图诞生了：



上图中的虚线框部分，就是 RPC 框架需要提供的模块和能力。

扩展设计

虽然上述设计已经跑通了基本调用流程，但离一个完备的 RPC 框架还有很大的差距，让我们带着问题来进一步完善下架构设计。

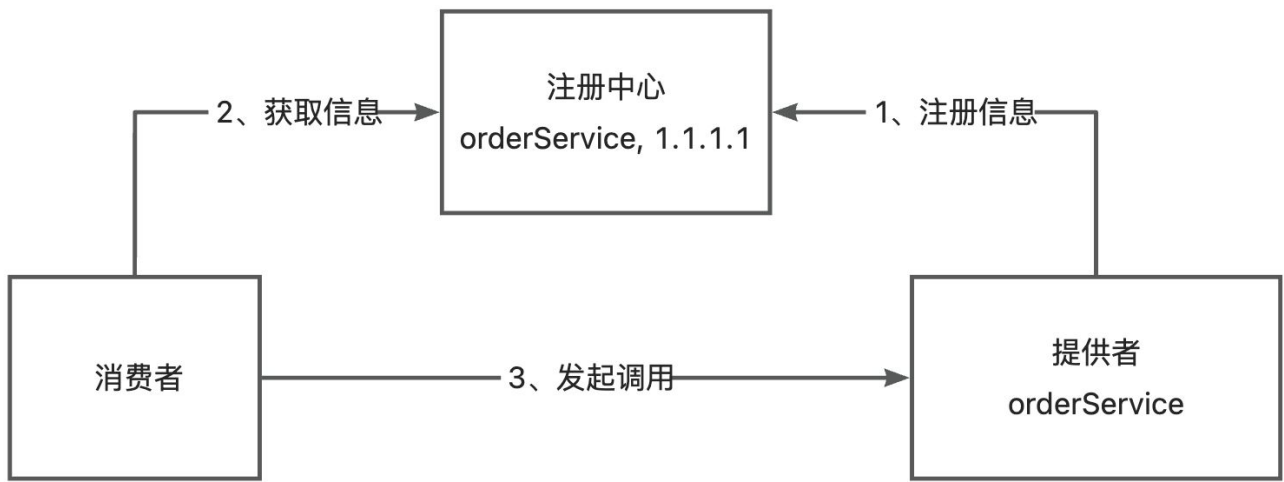
1、服务注册发现

问题 1：消费者如何知道提供者的调用地址呢？

类比生活场景，我们点外卖时，外卖小哥如何知道我们的地址和店铺的地址？肯定是买家和卖家分别填写地址，由平台来保存的。

因此，我们需要一个 **注册中心**，来保存服务提供者的地址。消费者要调用服务时，只需从注册中心获取对应服务的提供者地址即可。

架构图如下：



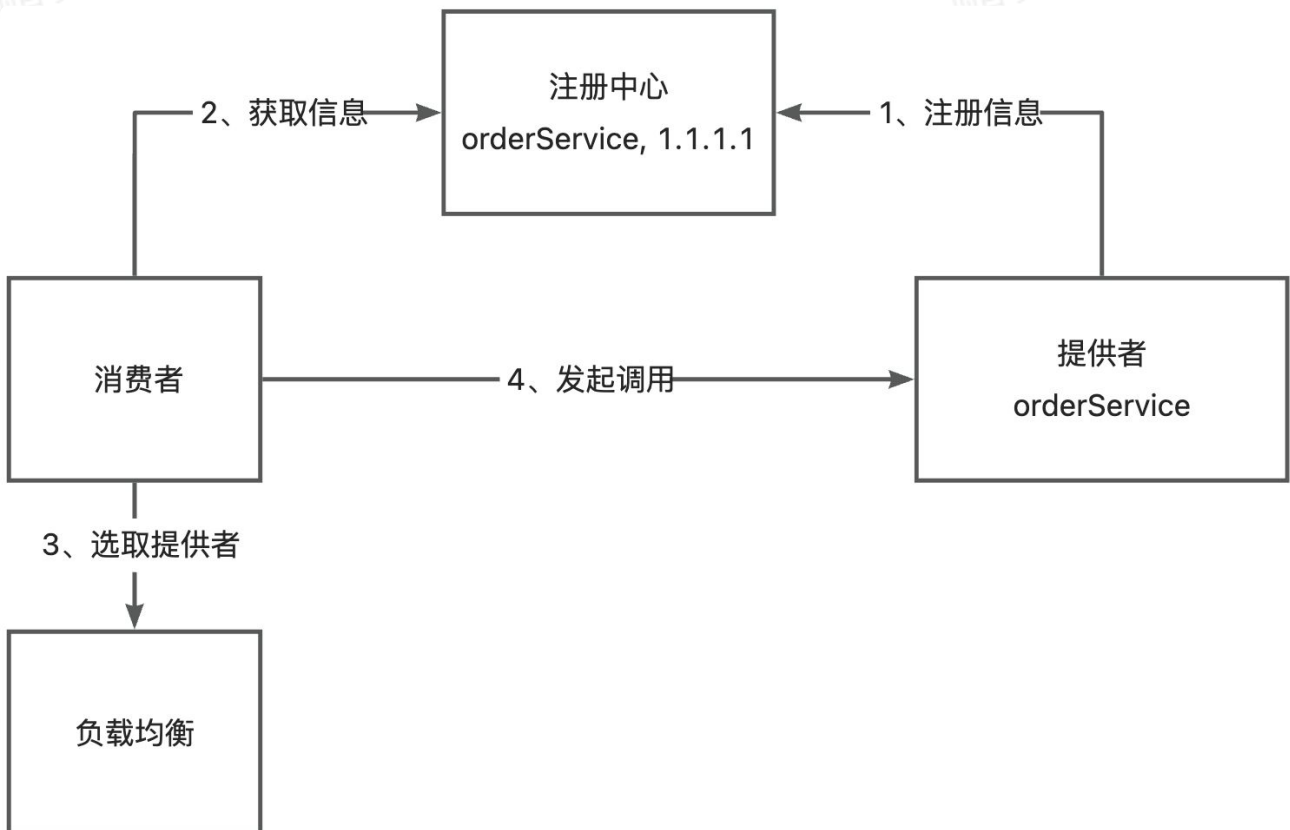
一般用现成的第三方注册中心，比如 Redis、Zookeeper 即可。

2、负载均衡

问题 2：如果有多个服务提供者，消费者应该调用哪个服务提供者呢？

我们可以给服务调用方增加负载均衡能力，通过指定不同的算法来决定调用哪一个服务提供者，比如轮询、随机、根据性能动态调用等。

架构图如下：

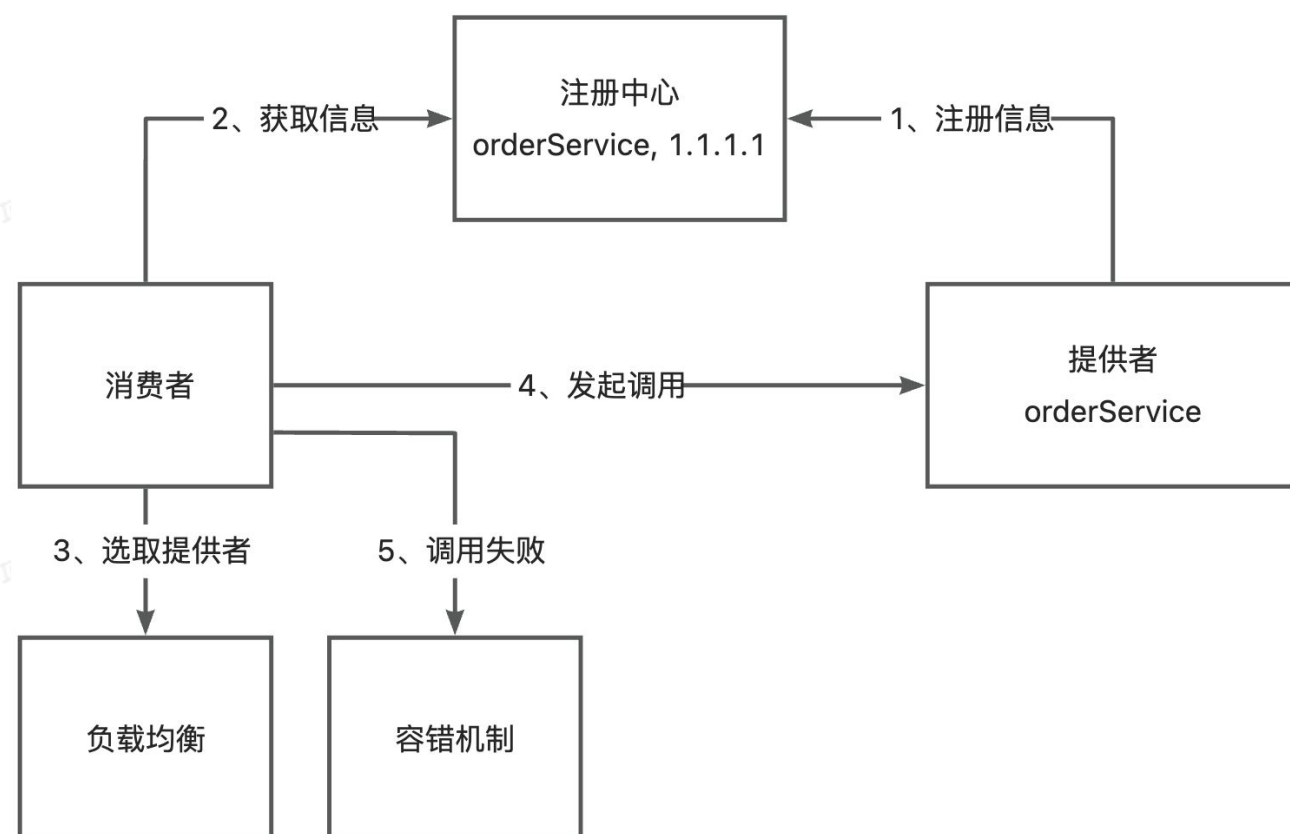


3、容错机制

问题 3：如果服务调用失败，应该如何处理呢？

为了保证分布式系统的高可用，我们通常会给服务的调用增加一定的容错机制，比如失败重试、降级调用其他接口等等。

架构图如下：



4、其他

除了上面几个经典设计外，如果想要做一个优秀的 RPC 框架，还要考虑很多问题。

比如：

- 服务提供者下线了怎么办？需要一个失效节点剔除机制。
- 服务消费者每次都从注册中心拉取信息，性能会不会很差？可以使用缓存来优化性能。
- 如何优化 RPC 框架的传输通讯性能？比如选择合适的网络框架、自定义协议头、节约传输体积等。
- 如何让整个框架更利于扩展？比如使用 Java 的 SPI 机制、配置化等等。

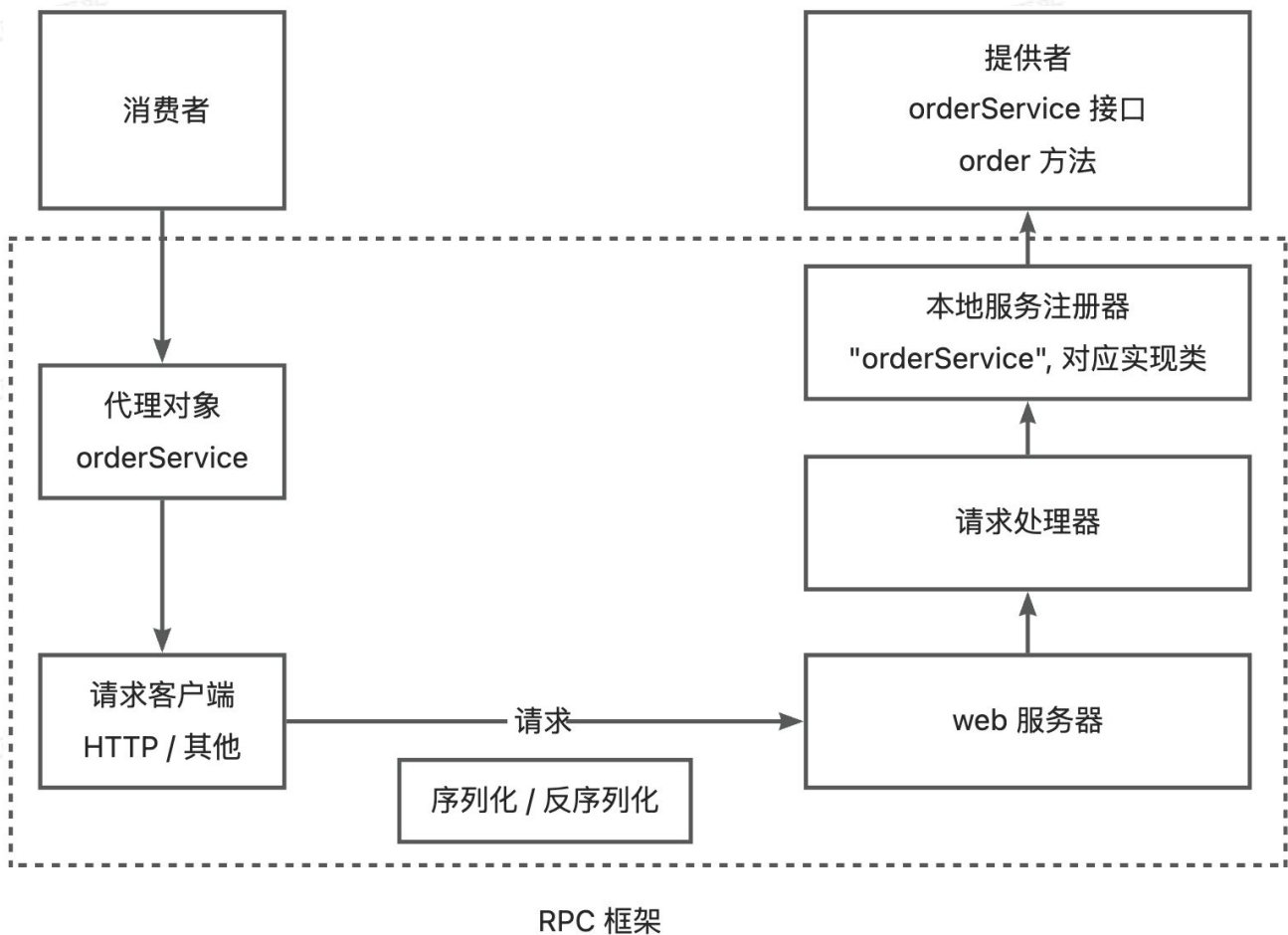
所以，完成 RPC 项目并不难，但做一个完美的 RPC 项目却是难于上青天啊！

总结一下，我们可以通过做一个 RPC 项目学习到网络、序列化、代理、服务注册发现、负载均衡、容错、可扩展设计等知识，相信完成项目后会收获满满。

三、开发简易版 RPC 框架

下面我们就从 0 开始，先完成一个简易版的 RPC 框架，后面再持续扩展优化。

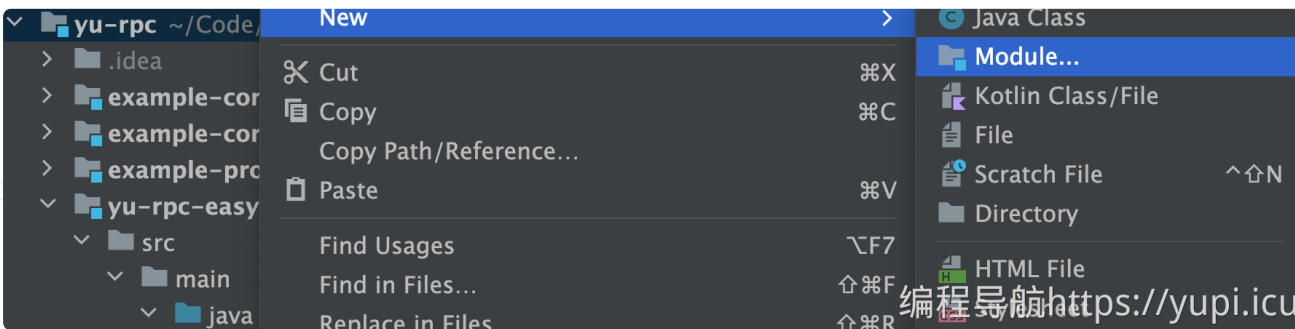
架构设计图如下：



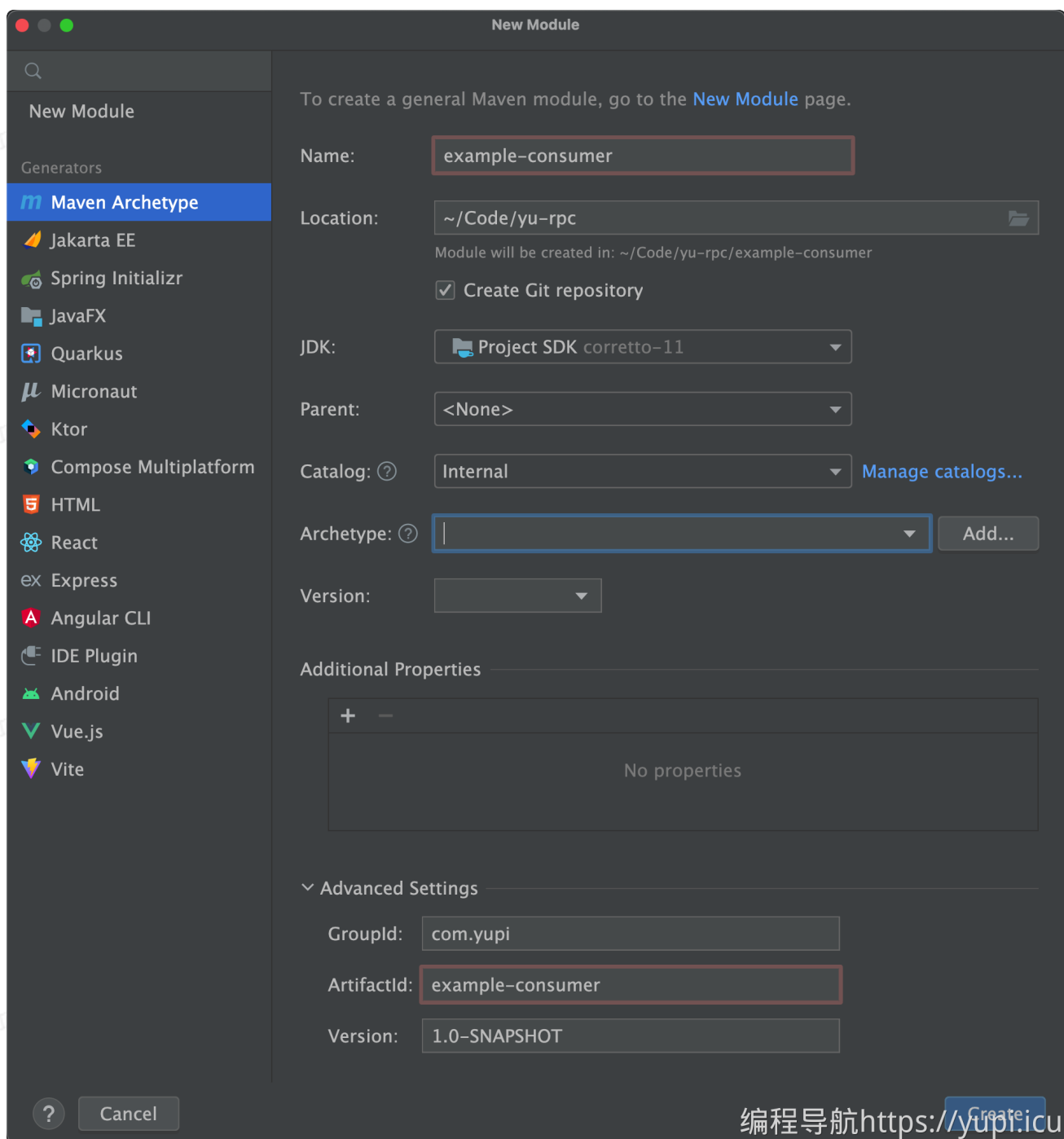
项目准备

1、项目初始化

首先创建一个项目根目录 `yu-rpc`，然后使用 IDEA 开发工具依次创建几个 Maven 模块。



整个基础 RPC 框架的项目目录如图：



分别介绍几个模块：

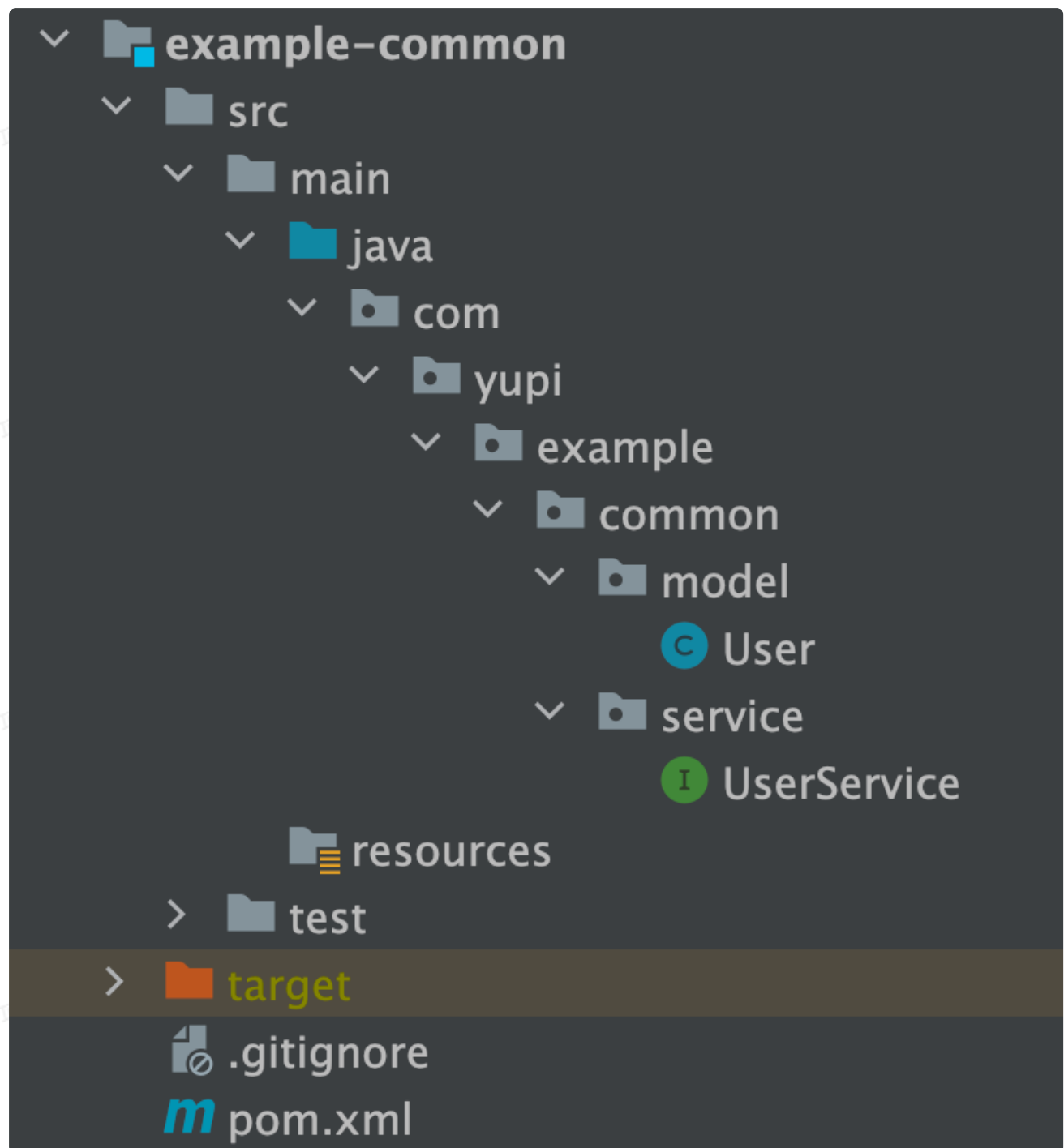
- example-common：示例代码的公共依赖，包括接口、Model 等
- example-consumer：示例服务消费者代码
- example-provider：示例服务提供者代码
- yu-rpc-easy：简易版 RPC 框架

在示例项目中，我们将以一个最简单的用户服务为例，演示整个服务调用过程。下面我们依次实现上述的几个模块。

2、公共模块

公共模块需要同时被消费者和服务提供者引入，主要是编写和服务相关的接口和数据模型。

整个模块的结构如下：



1) 编写用户实体类 User:

```

1  package com.yupi.example.common.model;
2
3  import java.io.Serializable;
4
5  /**
6   * 用户
7   */
8  public class User implements Serializable {
9
10     private String name;
11
12     public String getName() {
13         return name;
14     }
15
16     public void setName(String name) {
17         this.name = name;
18     }
19 }

```

注意，对象需要实现序列化接口，为后续网络传输序列化提供支持。

项目-更新

项目-更新

2) 编写用户服务接口 UserService，提供一个获取用户的方法：

```

1  package com.yupi.example.common.service;
2
3  import com.yupi.example.common.model.User;
4
5  /**
6   * 用户服务
7   */
8  public interface UserService {
9
10     /**
11      * 获取用户
12      *
13      * @param user
14      * @return
15      */
16     User getUser(User user);
17 }

```

3、服务提供者

服务提供者是真正实现了接口的模块。

1) 在 `pom.xml` 文件中引入依赖：

```
1 <dependencies>
2   <dependency>
3     <groupId>com.yupi</groupId>
4
5     <artifactId>yu-rpc-easy</artifactId>
6
7     <version>1.0-SNAPSHOT</version>
8
9   </dependency>
10
11  <dependency>
12    <groupId>com.yupi</groupId>
13
14    <artifactId>example-common</artifactId>
15
16    <version>1.0-SNAPSHOT</version>
17
18  </dependency>
19
20  <!-- https://doc.hutool.cn/ -->
21  <dependency>
22    <groupId>cn.hutool</groupId>
23
24    <artifactId>hutool-all</artifactId>
25
26    <version>5.8.16</version>
27
28  </dependency>
29
30  <!-- https://projectlombok.org/ -->
31  <dependency>
32    <groupId>org.projectlombok</groupId>
33
34    <artifactId>lombok</artifactId>
35
36    <version>1.18.30</version>
37
38    <scope>provided</scope>
39
40  </dependency>
41
42 </dependencies>
43
```

2) 编写服务实现类，实现公共模块中定义的用户服务接口。

功能是打印用户的名称，并且返回参数中的用户对象。

代码如下：

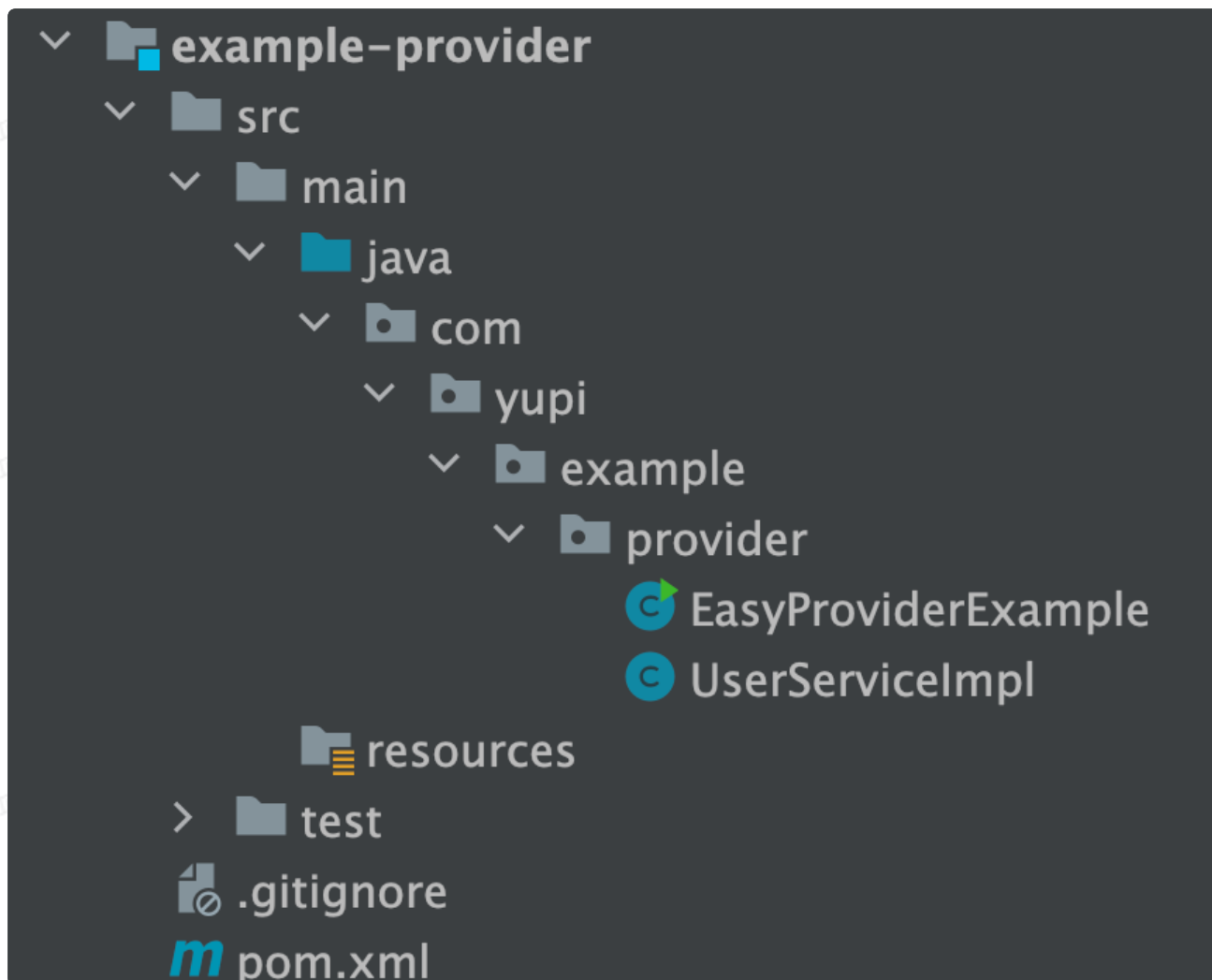
```
1  package com.yupi.example.provider;
2
3  import com.yupi.example.common.model.User;
4  import com.yupi.example.common.service.UserService;
5
6  /**
7   * 用户服务实现类
8   */
9  public class UserServiceImpl implements UserService {
10
11     public User getUser(User user) {
12         System.out.println("用户名: " + user.getName());
13         return user;
14     }
15 }
```

3) 编写服务提供者启动类 EasyProviderExample，之后会在该类的 main 方法中编写提供服务的代码。

代码如下：

```
1  package com.yupi.example.provider;
2
3  import com.yupi.example.common.service.UserService;
4  import com.yupi.yurpc.registry.LocalRegistry;
5  import com.yupi.yurpc.server.HttpServer;
6  import com.yupi.yurpc.server.VertxHttpServer;
7
8  /**
9   * 简易服务提供者示例
10  */
11 public class EasyProviderExample {
12
13     public static void main(String[] args) {
14         // 提供服务
15     }
16 }
```

最终得到的该模块目录如下：



4、服务消费者

服务消费者是需要调用服务的模块。

1) 在 `pom.xml` 文件中引入依赖，和提供者模块的依赖一致：

```

1  <dependencies>
2      <dependency>
3          <groupId>com.yupi</groupId>
4
5          <artifactId>yu-rpc-easy</artifactId>
6
7          <version>1.0-SNAPSHOT</version>
8
9      </dependency>
10
11     <dependency>
12         <groupId>com.yupi</groupId>
13
14         <artifactId>example-common</artifactId>
15
16         <version>1.0-SNAPSHOT</version>
17
18     </dependency>
19
20     <!-- https://doc.hutool.cn/ -->
21     <dependency>
22         <groupId>cn.hutool</groupId>
23
24         <artifactId>hutool-all</artifactId>
25
26         <version>5.8.16</version>
27
28     </dependency>
29
30     <!-- https://projectlombok.org/ -->
31     <dependency>
32         <groupId>org.projectlombok</groupId>
33
34         <artifactId>lombok</artifactId>
35
36         <version>1.18.30</version>
37
38         <scope>provided</scope>
39
40     </dependency>
41
42 </dependencies>
43

```

2) 创建服务消费者启动类 EasyConsumerExample，编写调用接口的代码。

代码如下：

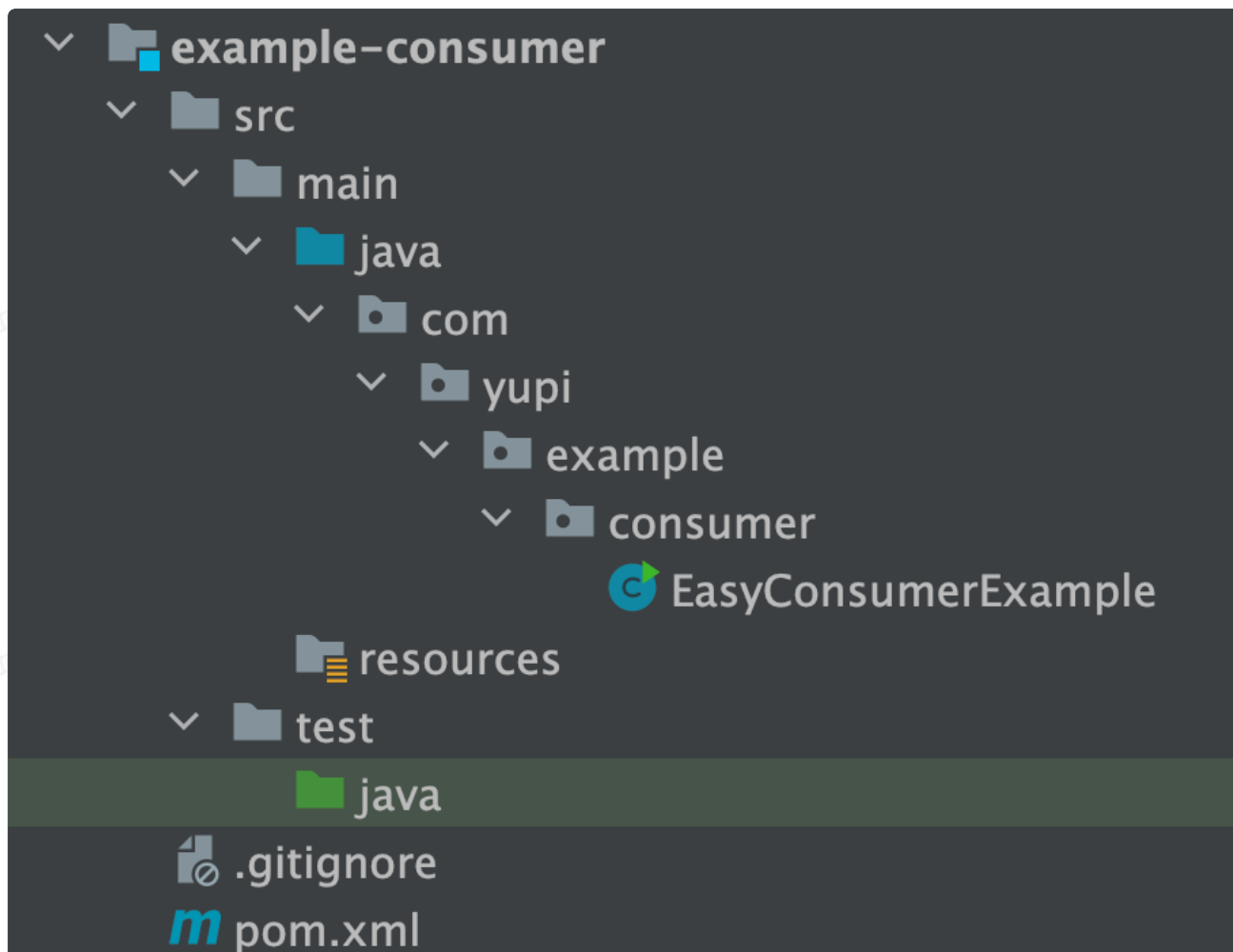
```

1  package com.yupi.example.consumer;
2
3  import com.yupi.example.common.model.User;
4  import com.yupi.example.common.service.UserService;
5  import com.yupi.yurpc.proxy.ServiceProxyFactory;
6
7  /**
8   * 简易服务消费者示例
9   */
10 public class EasyConsumerExample {
11
12     public static void main(String[] args) {
13         // todo 需要获取 UserService 的实现类对象
14         UserService userService = null;
15         User user = new User();
16         user.setName("yupi");
17         // 调用
18         User newUser = userService.getUser(user);
19         if (newUser != null) {
20             System.out.println(newUser.getName());
21         } else {
22             System.out.println("user == null");
23         }
24     }
25 }

```

需要注意的是，现在是无法获取到 userService 实例的，所以预留给 null。我们之后的目标是，能够通过 RPC 框架，快速得到一个支持远程调用服务提供者的代理对象，像调用本地方法一样调用 UserService 的方法。

最终得到的该模块目录如下：



web 服务器

接下来，我们要先让服务提供者提供 **可远程访问** 的服务。那么，就需要一个 web 服务器，能够接受处理请求、并返回响应。

web 服务器的选择有很多，比如 Spring Boot 内嵌的 Tomcat、NIO 框架 Netty 和 Vert.x 等等。

此处鱼皮带大家使用高性能的 NIO 框架 Vert.x 来作为 RPC 框架的 web 服务器。

Vert.x 官方文档: <https://vertx.io/> <<https://vertx.io/>>

1) 打开 `yu-rpc-easy` 项目，引入 Vert.x 和工具类的依赖：

```

1  <dependencies>
2      <!-- https://mvnrepository.com/artifact/io.vertx/vertx-core -->
3      <dependency>
4          <groupId>io.vertx</groupId>
5
6          <artifactId>vertx-core</artifactId>
7
8          <version>4.5.1</version>
9
10     </dependency>
11
12     <!-- https://doc.hutool.cn/ -->
13     <dependency>
14         <groupId>cn.hutool</groupId>
15
16         <artifactId>hutool-all</artifactId>
17
18         <version>5.8.16</version>
19
20     </dependency>
21
22     <!-- https://projectlombok.org/ -->
23     <dependency>
24         <groupId>org.projectlombok</groupId>
25
26         <artifactId>lombok</artifactId>
27
28         <version>1.18.30</version>
29
30         <scope>provided</scope>
31
32     </dependency>
33
34 </dependencies>
35

```

2) 编写一个 web 服务器的接口 HttpServer，定义统一的启动服务器方法，便于后续的扩展，比如实现多种不同的 web 服务器。

代码如下：

```
1 package com.yupi.yurpc.server;
2
3 /**
4  * HTTP 服务器接口
5  */
6 public interface HttpServer {
7
8     /**
9      * 启动服务器
10     *
11     * @param port
12     */
13     void doStart(int port);
14 }
```

3) 编写基于 Vert.x 实现的 web 服务器 VertxHttpServer，能够监听指定端口并处理请求。

代码如下：

```

1  package com.yupi.yurpc.server;
2
3  import io.vertx.core.Vertx;
4
5  public class VertxHttpServer implements HttpServer {
6
7      public void doStart(int port) {
8          // 创建 Vert.x 实例
9          Vertx vertx = Vertx.vertx();
10
11          // 创建 HTTP 服务器
12          io.vertx.core.http.HttpServer server = vertx.createHttpServer();
13
14          // 监听端口并处理请求
15          server.requestHandler(request -> {
16              // 处理 HTTP 请求
17              System.out.println("Received request: " + request.method() + " "
18
19              // 发送 HTTP 响应
20              request.response()
21                  .putHeader("content-type", "text/plain")
22                  .end("Hello from Vert.x HTTP server!");
23          });
24
25          // 启动 HTTP 服务器并监听指定端口
26          server.listen(port, result -> {
27              if (result.succeeded()) {
28                  System.out.println("Server is now listening on port " + port)
29              } else {
30                  System.err.println("Failed to start server: " + result.cause()
31              }
32          });
33      }
34  }

```

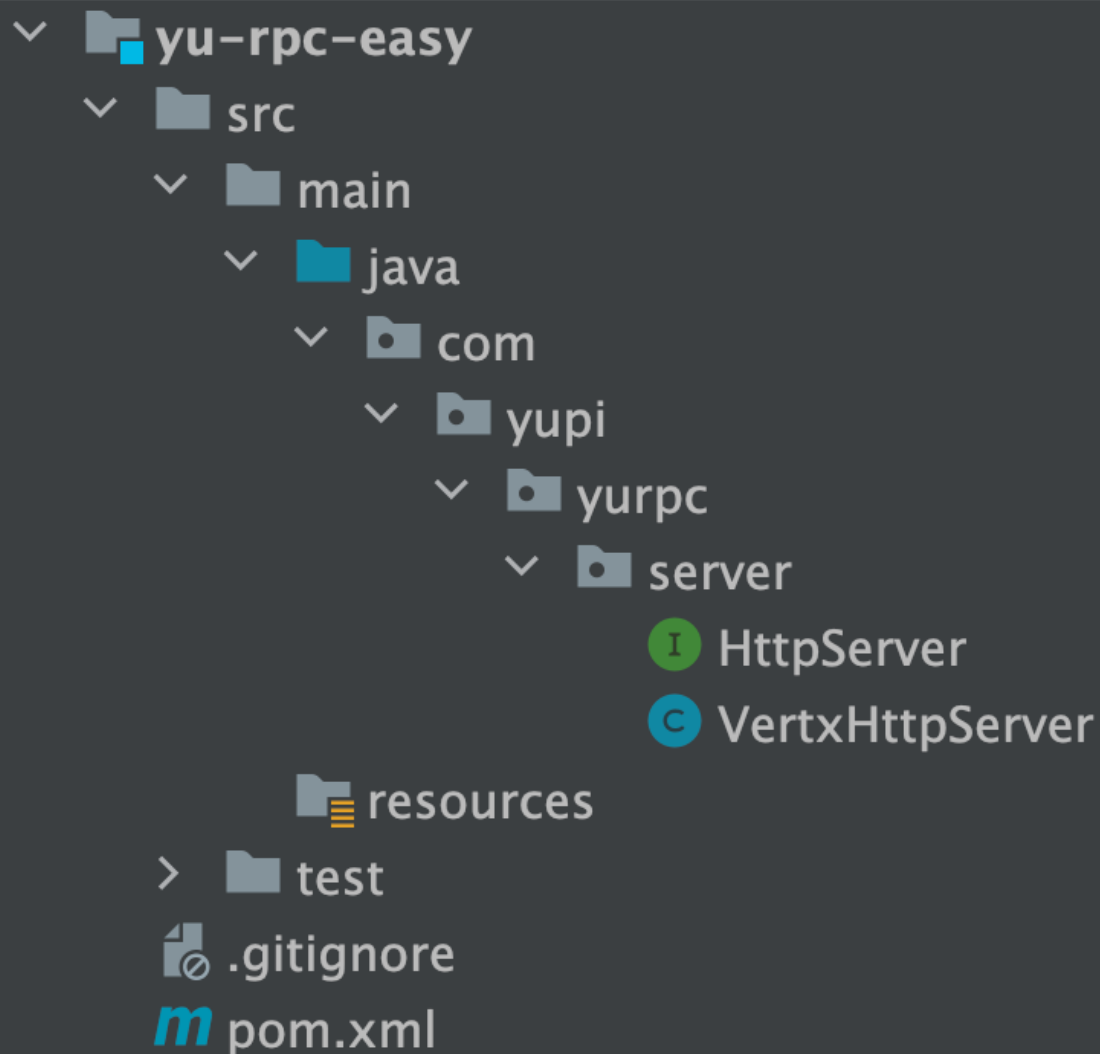
4) 验证 web 服务器能否启动成功并接受请求。

修改示例服务提供者模块的 `EasyProviderExample` 类，编写启动 web 服务的代码，如下：

```
1 package com.yupi.example.provider;
2
3 import com.yupi.example.common.service.UserService;
4 import com.yupi.yurpc.registry.LocalRegistry;
5 import com.yupi.yurpc.server.HttpServer;
6 import com.yupi.yurpc.server.VertxHttpServer;
7
8 /**
9  * 简易服务提供者示例
10  */
11 public class EasyProviderExample {
12
13     public static void main(String[] args) {
14         // 启动 web 服务
15         HttpServer httpServer = new VertxHttpServer();
16         httpServer.doStart(8080);
17     }
18 }
```

通过浏览器访问 `localhost:8080` , 查看能否正常访问并看到输出的文字。

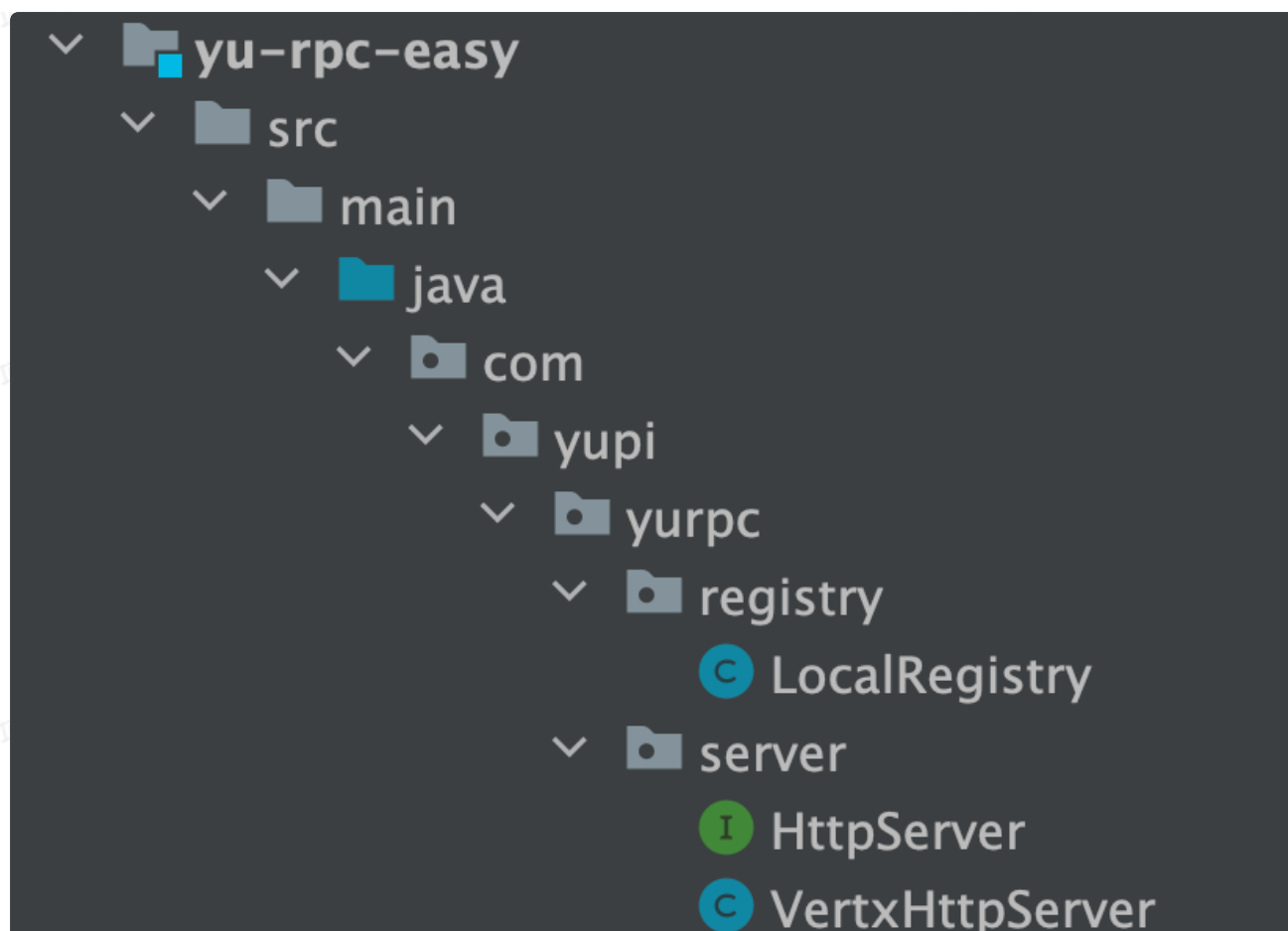
此时的 RPC 模块目录结构如下:



本地服务注册器

我们现在做的简易 RPC 框架主要是跑通流程，所以暂时先不用第三方注册中心，直接把服务注册到服务提供者本地即可。

在 RPC 模块中创建本地服务注册器 LocalRegistry，当前目录结构如下：



使用线程安全的 `ConcurrentHashMap` 存储服务注册信息，key 为服务名称、value 为服务的实现类。之后就可以根据要调用的服务名称获取到对应的实现类，然后通过反射进行方法调用了。

代码如下：

```

1  package com.yupi.yurpc.registry;
2
3  import java.util.Map;
4  import java.util.concurrent.ConcurrentHashMap;
5
6  /**
7   * 本地注册中心
8   */
9  public class LocalRegistry {
10
11      /**
12       * 注册信息存储
13       */
14      private static final Map<String, Class<?>> map = new ConcurrentHashMap<>();
15
16      /**
17       * 注册服务
18       *
19       * @param serviceName
20       * @param implClass
21       */
22      public static void register(String serviceName, Class<?> implClass) {
23          map.put(serviceName, implClass);
24      }
25
26      /**
27       * 获取服务
28       *
29       * @param serviceName
30       * @return
31       */
32      public static Class<?> get(String serviceName) {
33          return map.get(serviceName);
34      }
35
36      /**
37       * 删除服务
38       *
39       * @param serviceName
40       */
41      public static void remove(String serviceName) {
42          map.remove(serviceName);
43      }
44  }

```

项目-更新

项目-更新

注意，本地服务注册器和注册中心的作用是有区别的。注册中心的作用侧重于管理注册的服务、提供服务信息给消费者；而本地服务注册器的作用是根据服务名获取到对应的实现类，是完成调用必不可少的模块。

服务提供者启动时，需要注册服务到注册器中，修改 `EasyProviderExample` 代码如下：

```
1  package com.yupi.example.provider;
2
3  import com.yupi.example.common.service.UserService;
4  import com.yupi.yurpc.registry.LocalRegistry;
5  import com.yupi.yurpc.server.HttpServer;
6  import com.yupi.yurpc.server.VertxHttpServer;
7
8  /**
9   * 简易服务提供者示例
10  */
11  public class EasyProviderExample {
12
13      public static void main(String[] args) {
14          // 注册服务
15          LocalRegistry.register(UserService.class.getName(), UserServiceImpl.c
16
17          // 启动 web 服务
18          HttpServer httpServer = new VertxHttpServer();
19          httpServer.doStart(8080);
20      }
21  }
```

序列化器

服务在本地注册后，我们就可以根据请求信息取出实现类并调用方法了。

但是在编写处理请求的逻辑前，我们要先实现序列化器模块。因为无论是请求或响应，都会涉及参数的传输。而 Java 对象是存活在 JVM 虚拟机中的，如果想在其他位置存储并访问、或者在网络中进行传输，就需要进行序列化和反序列化。

什么是序列化和反序列化呢？

- 序列化：将 Java 对象转为可传输的字节数组。
- 反序列化：将字节数组转换为 Java 对象。

有很多种不同的序列化方式，比如 Java 原生序列化、JSON、Hessian、Kryo、protobuf 等。

为了实现方便，此处选择 Java 原生的序列化器。

1) 在 RPC 模块中编写序列化接口 `Serializer`，提供序列化和反序列化两个方法，便于后续扩展更多的序列化器。

代码如下:

```
1  package com.yupi.yurpc.serializer;
2
3  import java.io.IOException;
4
5  /**
6   * 序列化器接口
7   */
8  public interface Serializer {
9
10     /**
11      * 序列化
12      *
13      * @param object
14      * @param <T>
15      * @return
16      * @throws IOException
17      */
18     <T> byte[] serialize(T object) throws IOException;
19
20     /**
21      * 反序列化
22      *
23      * @param bytes
24      * @param type
25      * @param <T>
26      * @return
27      * @throws IOException
28      */
29     <T> T deserialize(byte[] bytes, Class<T> type) throws IOException;
30 }
```

2) 基于 Java 自带的序列化器实现 JdkSerializer, 代码如下:

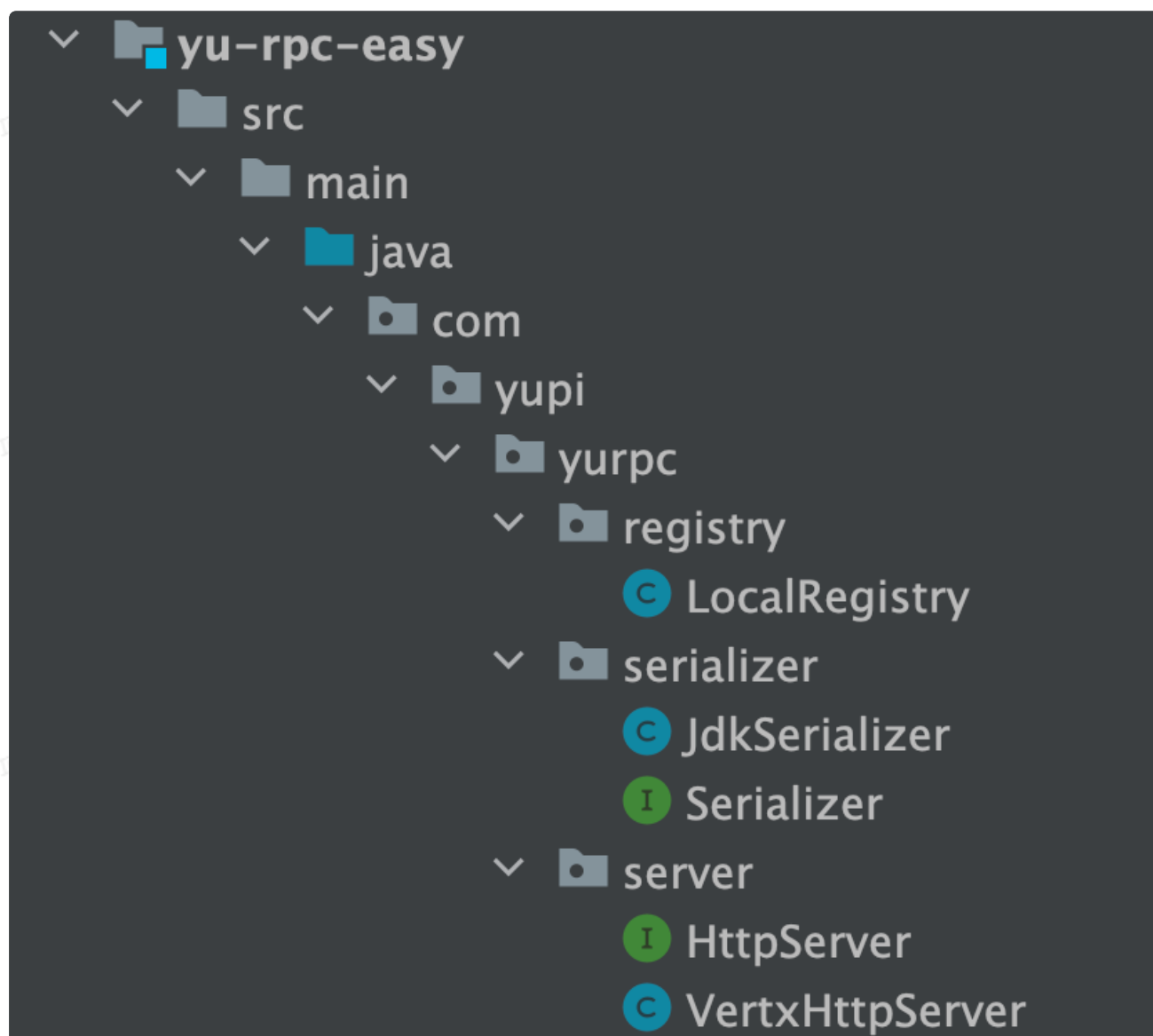
```

1  package com.yupi.yurpc.serializer;
2
3  import java.io.*;
4
5  /**
6   * JDK 序列化器
7   */
8  public class JdkSerializer implements Serializer {
9
10     /**
11      * 序列化
12      *
13      * @param object
14      * @param <T>
15      * @return
16      * @throws IOException
17      */
18     @Override
19     public <T> byte[] serialize(T object) throws IOException {
20         ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
21         ObjectOutputStream objectOutputStream = new ObjectOutputStream(outputStream);
22         objectOutputStream.writeObject(object);
23         objectOutputStream.close();
24         return outputStream.toByteArray();
25     }
26
27     /**
28      * 反序列化
29      *
30      * @param bytes
31      * @param type
32      * @param <T>
33      * @return
34      * @throws IOException
35      */
36     @Override
37     public <T> T deserialize(byte[] bytes, Class<T> type) throws IOException {
38         ByteArrayInputStream inputStream = new ByteArrayInputStream(bytes);
39         ObjectInputStream objectInputStream = new ObjectInputStream(inputStream);
40         try {
41             return (T) objectInputStream.readObject();
42         } catch (ClassNotFoundException e) {
43             throw new RuntimeException(e);
44         } finally {
45             objectInputStream.close();
46         }
47     }
48 }

```

上面这段代码无需记忆，需要用到的时候照抄即可，关键是要理解序列化和反序列化的区别。

当前 RPC 模块的目录结构如下：

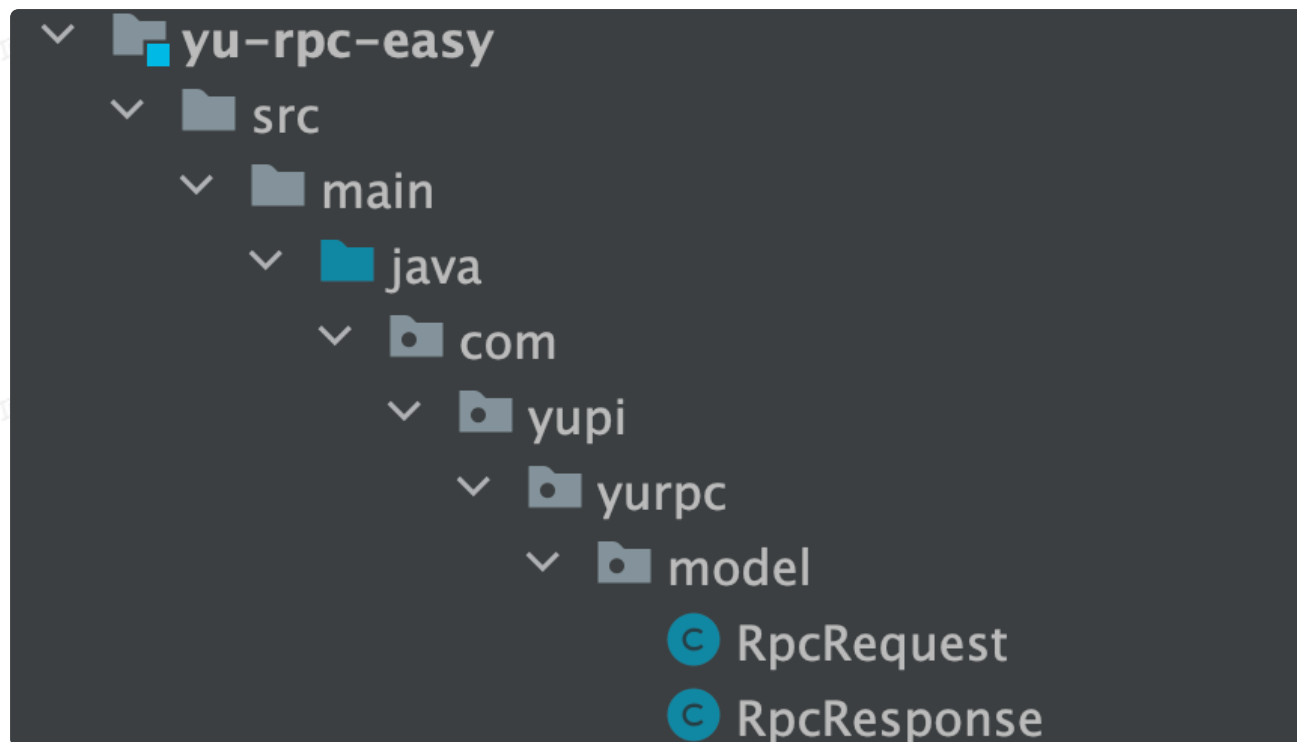


提供者处理调用 - 请求处理器

请求处理器是 RPC 框架的实现关键，它的作用是：处理接收到的请求，并根据请求参数找到对应的服务和方法，通过反射实现调用，最后封装返回结果并响应请求。

1) 在 RPC 模块中编写请求和响应封装类。

目录结构如下：



请求类 `RpcRequest` 的作用是封装调用所需的信息，比如服务名称、方法名称、调用参数的类型列表、参数列表。这些都是 Java 反射机制所需的参数。

代码如下：

```

1  package com.yupi.yurpc.model;
2
3  import lombok.AllArgsConstructor;
4  import lombok.Builder;
5  import lombok.Data;
6  import lombok.NoArgsConstructor;
7
8  import java.io.Serializable;
9
10 /**
11  * RPC 请求
12  */
13 @Data
14 @Builder
15 @AllArgsConstructor
16 @NoArgsConstructor
17 public class RpcRequest implements Serializable {
18
19     /**
20      * 服务名称
21      */
22     private String serviceName;
23
24     /**
25      * 方法名称
26      */
27     private String methodName;
28
29     /**
30      * 参数类型列表
31      */
32     private Class<?>[] parameterTypes;
33
34     /**
35      * 参数列表
36      */
37     private Object[] args;
38
39 }

```

响应类 `RpcResponse` 的作用是封装调用方法得到的返回值、以及调用的信息（比如异常情况）等。

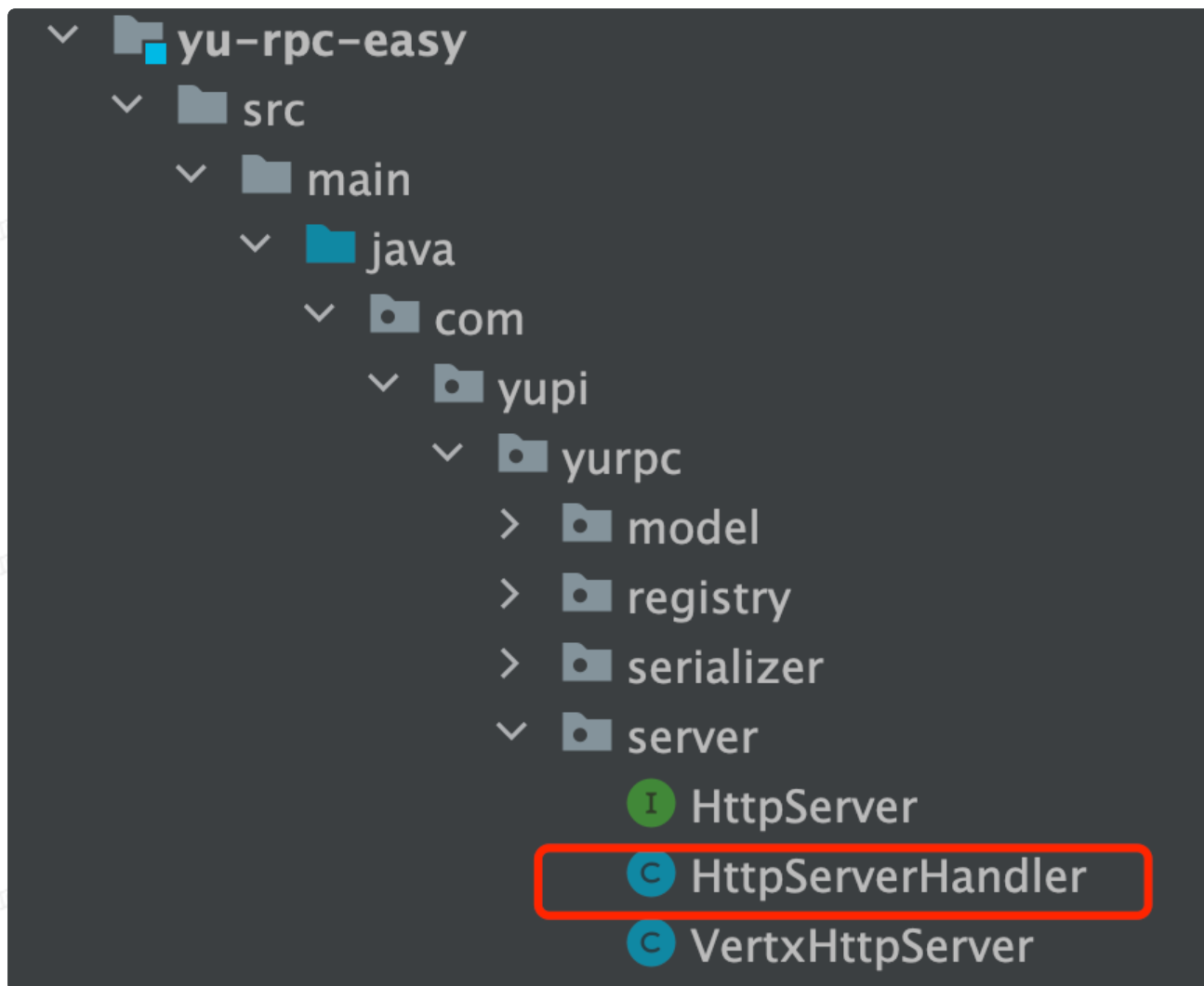
代码如下：

```

1  package com.yupi.yurpc.model;
2
3  import lombok.AllArgsConstructor;
4  import lombok.Builder;
5  import lombok.Data;
6  import lombok.NoArgsConstructor;
7
8  import java.io.Serializable;
9
10 /**
11  * RPC 响应
12  */
13 @Data
14 @Builder
15 @AllArgsConstructor
16 @NoArgsConstructor
17 public class RpcResponse implements Serializable {
18
19     /**
20      * 响应数据
21      */
22     private Object data;
23
24     /**
25      * 响应数据类型（预留）
26      */
27     private Class<?> dataType;
28
29     /**
30      * 响应信息
31      */
32     private String message;
33
34     /**
35      * 异常信息
36      */
37     private Exception exception;
38
39 }

```

2) 编写请求处理器 HttpServerHandler。



业务流程如下：

1. 反序列化请求为对象，并从请求对象中获取参数。
2. 根据服务名称从本地注册器中获取到对应的服务实现类。
3. 通过反射机制调用方法，得到返回结果。
4. 对返回结果进行封装和序列化，并写入到响应中。

完整代码如下，配合上述流程和注释应该不难理解：

```

1  package com.yupi.yurpc.server;
2
3  import com.yupi.yurpc.model.RpcRequest;
4  import com.yupi.yurpc.model.RpcResponse;
5  import com.yupi.yurpc.registry.LocalRegistry;
6  import com.yupi.yurpc.serializer.JdkSerializer;
7  import com.yupi.yurpc.serializer.Serializer;
8  import io.vertx.core.Handler;
9  import io.vertx.core.buffer.Buffer;
10 import io.vertx.core.http.HttpServerRequest;
11 import io.vertx.core.http.HttpServerResponse;
12
13 import java.io.IOException;
14 import java.lang.reflect.Method;
15
16 /**
17  * HTTP 请求处理
18  */
19 public class HttpServerHandler implements Handler<HttpServerRequest> {
20
21     @Override
22     public void handle(HttpServerRequest request) {
23         // 指定序列化器
24         final Serializer serializer = new JdkSerializer();
25
26         // 记录日志
27         System.out.println("Received request: " + request.method() + " " + re
28
29         // 异步处理 HTTP 请求
30         request.bodyHandler(body -> {
31             byte[] bytes = body.getBytes();
32             RpcRequest rpcRequest = null;
33             try {
34                 rpcRequest = serializer.deserialize(bytes, RpcRequest.class);
35             } catch (Exception e) {
36                 e.printStackTrace();
37             }
38
39             // 构造响应结果对象
40             RpcResponse rpcResponse = new RpcResponse();
41             // 如果请求为 null, 直接返回
42             if (rpcRequest == null) {
43                 rpcResponse.setMessage("rpcRequest is null");
44                 doResponse(request, rpcResponse, serializer);
45                 return;
46             }
47
48             try {
49                 // 获取要调用的服务实现类, 通过反射调用
50                 Class<?> implClass = LocalRegistry.get(rpcRequest.getServiceName

```

```

51         Method method = implClass.getMethod(rpcRequest.getMethodName(
52         Object result = method.invoke(implClass.newInstance(), rpcReq
53         // 封装返回结果
54         rpcResponse.setData(result);
55         rpcResponse.setDataType(method.getReturnType());
56         rpcResponse.setMessage("ok");
57     } catch (Exception e) {
58         e.printStackTrace();
59         rpcResponse.setMessage(e.getMessage());
60         rpcResponse.setException(e);
61     }
62     // 响应
63     doResponse(request, rpcResponse, serializer);
64 });
65 }
66
67 /**
68  * 响应
69  *
70  * @param request
71  * @param rpcResponse
72  * @param serializer
73  */
74 void doResponse(HttpServerRequest request, RpcResponse rpcResponse, Seriali
75     HttpServerResponse httpResponse = request.response()
76     .putHeader("content-type", "application/json");
77     try {
78         // 序列化
79         byte[] serialized = serializer.serialize(rpcResponse);
80         httpResponse.end(Buffer.buffer(serialized));
81     } catch (IOException e) {
82         e.printStackTrace();
83         httpResponse.end(Buffer.buffer());
84     }
85 }
86

```

需要注意，不同的 web 服务器对应的请求处理器实现方式也不同，比如 Vert.x 中是通过实现 `Handler<HttpServerRequest>` 接口来自定义请求处理器的。并且可以通过 `request.bodyHandler` 异步处理请求。

3) 给 HttpServer 绑定请求处理器。

修改 `VertxHttpServer` 的代码，通过 `server.requestHandler` 绑定请求处理器。

修改后的代码如下：

```

1  package com.yupi.yurpc.server;
2
3  import io.vertx.core.Vertx;
4
5  /**
6   * Vertx HTTP 服务器
7   */
8  public class VertxHttpServer implements HttpServer {
9
10     /**
11      * 启动服务器
12      *
13      * @param port
14      */
15     public void doStart(int port) {
16         // 创建 Vert.x 实例
17         Vertx vertx = Vertx.vertx();
18
19         // 创建 HTTP 服务器
20         io.vertx.core.http.HttpServer server = vertx.createHttpServer();
21
22         // 监听端口并处理请求
23         server.requestHandler(new HttpServerHandler());
24
25         // 启动 HTTP 服务器并监听指定端口
26         server.listen(port, result -> {
27             if (result.succeeded()) {
28                 System.out.println("Server is now listening on port " + port)
29             } else {
30                 System.err.println("Failed to start server: " + result.cause()
31             }
32         });
33     }
34 }

```

至此，引入了 RPC 框架的服务提供者模块，已经能够接受请求并完成服务调用了。

消费方发起调用 - 代理

在项目准备阶段，我们已经预留了一段调用服务的代码，只要能够获取到 UserService 对象（实现类），就能跑通整个流程。

但 UserService 的实现类从哪来呢？

总不能把服务提供者的 UserServiceImpl 复制粘贴到消费者模块吧？要能那样做还需要 RPC 框架干什么？分布式系统中，我们调用其他项目或团队提供的接口时，一般只关注请求参数和响应结

果，而不关注具体实现。

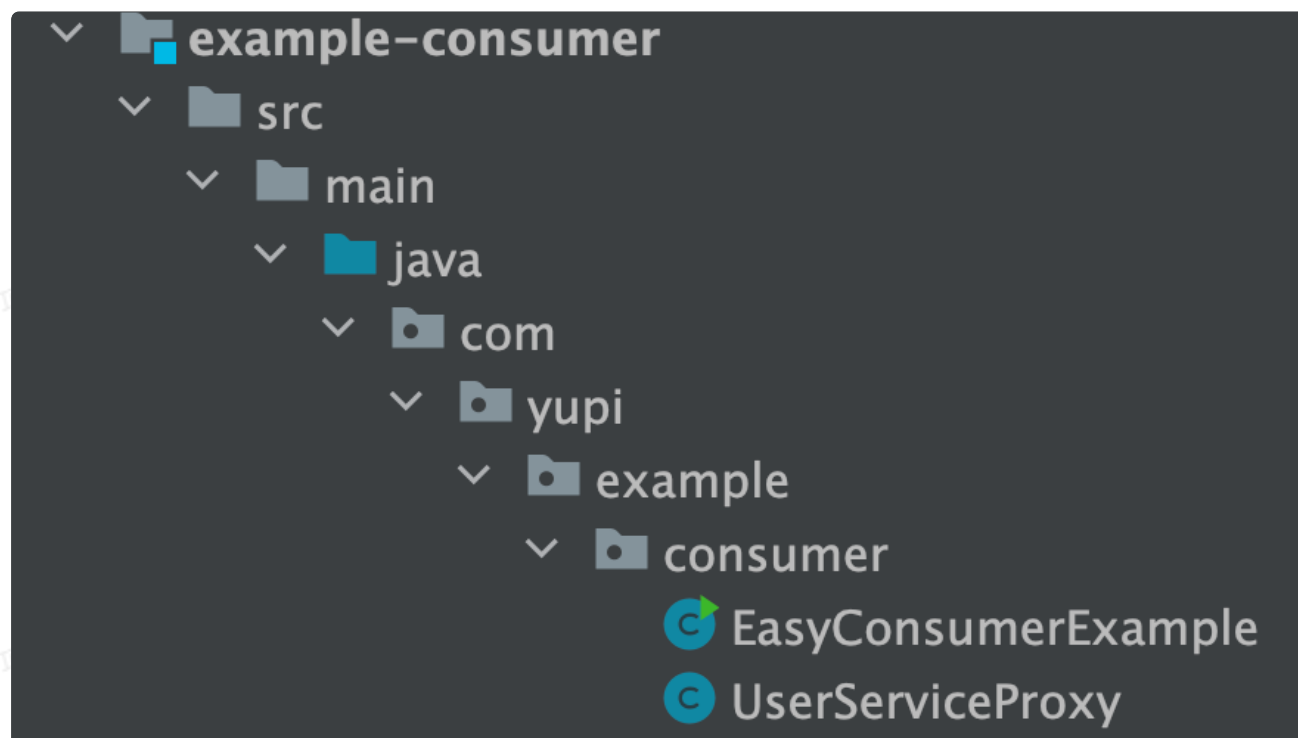
在之前的架构中讲过，我们可以通过生成代理对象来简化消费方的调用。

代理的实现方式大致分为 2 类：静态代理和动态代理，下面依次实现。

静态代理

静态代理是指为每一个特定类型的接口或对象，编写一个代理类。

比如在 `example-consumer` 模块中，创建一个静态代理 `UserServiceProxy`，实现 `UserService` 接口和 `getUser` 方法。



只不过实现 `getUser` 方法时，不是复制粘贴服务提供者 `UserServiceImpl` 中的代码，而是要构造 HTTP 请求去调用服务提供者。

需要注意发送请求前要将参数序列化，代码如下：

```

1  package com.yupi.example.consumer;
2
3  import cn.hutool.http.HttpRequest;
4  import cn.hutool.http.HttpResponse;
5  import com.yupi.example.common.model.User;
6  import com.yupi.example.common.service.UserService;
7  import com.yupi.yurpc.model.RpcRequest;
8  import com.yupi.yurpc.model.RpcResponse;
9  import com.yupi.yurpc.serializer.JdkSerializer;
10 import com.yupi.yurpc.serializer.Serializer;
11
12 import java.io.IOException;
13
14 /**
15  * 静态代理
16  */
17 public class UserServiceProxy implements UserService {
18
19     public User getUser(User user) {
20         // 指定序列化器
21         Serializer serializer = new JdkSerializer();
22
23         // 发请求
24         RpcRequest rpcRequest = RpcRequest.builder()
25             .serviceName(UserService.class.getName())
26             .methodName("getUser")
27             .parameterTypes(new Class[]{User.class})
28             .args(new Object[]{user})
29             .build();
30         try {
31             byte[] bodyBytes = serializer.serialize(rpcRequest);
32             byte[] result;
33             try (HttpResponse httpResponse = HttpRequest.post("http://localho
34                 .body(bodyBytes)
35                 .execute()) {
36                 result = httpResponse.bodyBytes();
37             }
38             RpcResponse rpcResponse = serializer.deserialize(result, RpcRespo
39             return (User) rpcResponse.getData();
40         } catch (IOException e) {
41             e.printStackTrace();
42         }
43
44         return null;
45     }
46 }

```

然后修改 EasyConsumerExample，new 一个代理对象并赋值给 userService，就能完成调用：

```
1  /**
2   * 简易服务消费者示例
3   */
4  public class EasyConsumerExample {
5
6      public static void main(String[] args) {
7          // 静态代理
8          UserService userService = new UserServiceProxy();
9
10         ...
11     }
12 }
```

静态代理虽然很好理解（就是写个实现类嘛），但缺点也很明显，我们如果要给每个服务接口都写一个实现类，是非常麻烦的，这种代理方式的灵活性很差！

所以 RPC 框架中，我们会使用动态代理。

动态代理

动态代理的作用是，根据要生成的对象的类型，自动生成一个代理对象。

常用的动态代理实现方式有 JDK 动态代理和基于字节码生成的动态代理（比如 CGLIB）。前者简单易用、无需引入额外的库，但缺点是只能对接口进行代理；后者更灵活、可以对任何类进行代理，但性能略低于 JDK 动态代理。

此处我们使用 JDK 动态代理。

1) 在 RPC 模块中编写动态代理类 ServiceProxy，需要实现 InvocationHandler 接口的 invoke 方法。

代码如下（几乎就是把静态代理的代码搬运过来）：

```

1  package com.yupi.yurpc.proxy;
2
3  import cn.hutool.http.HttpRequest;
4  import cn.hutool.http.HttpResponse;
5  import com.yupi.yurpc.model.RpcRequest;
6  import com.yupi.yurpc.model.RpcResponse;
7  import com.yupi.yurpc.serializer.JdkSerializer;
8  import com.yupi.yurpc.serializer.Serializer;
9
10 import java.io.IOException;
11 import java.lang.reflect.InvocationHandler;
12 import java.lang.reflect.Method;
13
14 /**
15  * 服务代理（JDK 动态代理）
16  */
17 public class ServiceProxy implements InvocationHandler {
18
19     /**
20      * 调用代理
21      *
22      * @return
23      * @throws Throwable
24      */
25     @Override
26     public Object invoke(Object proxy, Method method, Object[] args) throws T
27         // 指定序列化器
28         Serializer serializer = new JdkSerializer();
29
30         // 构造请求
31         RpcRequest rpcRequest = RpcRequest.builder()
32             .serviceName(method.getDeclaringClass().getName())
33             .methodName(method.getName())
34             .parameterTypes(method.getParameterTypes())
35             .args(args)
36             .build();
37         try {
38             // 序列化
39             byte[] bodyBytes = serializer.serialize(rpcRequest);
40             // 发送请求
41             // todo 注意，这里地址被硬编码了（需要使用注册中心和服务发现机制解决）
42             try (HttpResponse httpResponse = HttpRequest.post("http://localho
43                 .body(bodyBytes)
44                 .execute()) {
45                 byte[] result = httpResponse.bodyBytes();
46                 // 反序列化
47                 RpcResponse rpcResponse = serializer.deserialize(result, RpcR
48                 return rpcResponse.getData();
49             }
50         } catch (IOException e) {

```

```

51         e.printStackTrace();
52     }
53
54     return null;
55 }
56

```

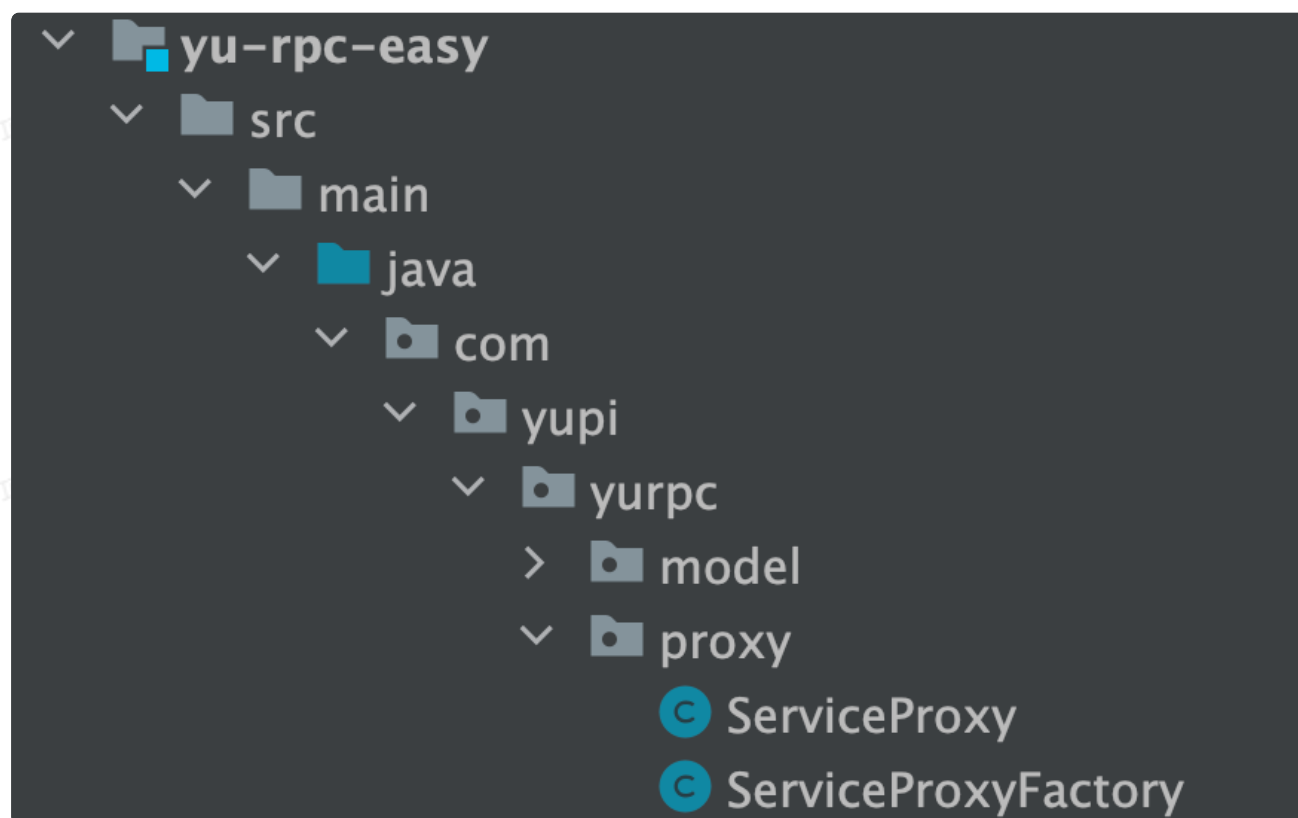
解释下上述代码，当用户调用某个接口的方法时，会改为调用 invoke 方法。在 invoke 方法中，我们可以获取到要调用的方法信息、传入的参数列表等，这不就是我们服务提供者需要的参数么？用这些参数来构造请求对象就可以完成调用了。

需要注意的是，上述代码中，请求的服务提供者地址被硬编码了，需要使用注册中心和服务发现机制来解决。

没办法直接看懂上述代码也没关系，先跟着敲完，之后可以通过 debug 来帮助理解。

2) 创建动态代理工厂 ServiceProxyFactory，作用是根据指定类创建动态代理对象。

目录结构如图：



这里是使用了 **工厂设计模式**，来简化对象的创建过程，代码如下：

```

1  package com.yupi.yurpc.proxy;
2
3  import java.lang.reflect.Proxy;
4
5  /**
6   * 服务代理工厂（用于创建代理对象）
7   */
8  public class ServiceProxyFactory {
9
10     /**
11      * 根据服务类获取代理对象
12      *
13      * @param serviceClass
14      * @param <T>
15      * @return
16      */
17     public static <T> T getProxy(Class<T> serviceClass) {
18         return (T) Proxy.newProxyInstance(
19             serviceClass.getClassLoader(),
20             new Class[]{serviceClass},
21             new ServiceProxy());
22     }
23 }

```

上述代码中，主要是通过 `Proxy.newProxyInstance` 方法为指定类型创建代理对象。

3) 最后，在 `EasyConsumerExample` 中，就可以通过调用工厂来为 `UserService` 获取动态代理对象了。

代码如下：

```

1  // 动态代理
2  UserService userService = ServiceProxyFactory.getProxy(UserService.class);

```

至此，简易版的 RPC 框架已经开发完成，下面我们进行测试。

四、测试验证

1) 以 debug 模式启动服务提供者，执行 main 方法：

```
Run: EasyProviderExample x
/Users/yupi/Library/Java/JavaVirtualMach
2月 19, 2024 7:45:05 下午 io.netty.resolv
警告: Can not find io.netty.resolver.dns.
Server is now listening on port 8080
|
```

2) 以 debug 模式启动服务消费者，执行 main 方法。

在 ServiceProxy 代理类中添加断点，可以看到调用 userService 时，实际是调用了代理对象的 invoke 方法，并且获取到了 serviceName、methodName、参数类型和列表等信息。

如下图：

```
ServiceProxy.java x EasyConsumerExample.java x
liyupi
29 @Override
30 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable { proxy: "nu
31 // 指定序列化器
32 Serializer serializer = new JdkSerializer(); serializer: JdkSerializer@1423
33
34 // 构造请求
35 RpcRequest rpcRequest = RpcRequest.builder() rpcRequest: "RpcRequest(serviceName=com.yupi.e
36 .service
37 .method
38 .param
39 .args(a
40 .build(
41
42 try {
43 // 序列化
44 // 发送请求
45 // todo 注意 Set value F2 Create renderer
```

```
> f serviceName = "com.yupi.example.common.service.UserService"
> f methodName = "getUser"
> f parameterTypes = {Class[1]@1518}
> 0 = {Class@824} "class com.yupi.example.common.model.User" ... Navigate
> f args = {Object[1]@1422}
> 0 = {User@1520}
> f name = "yupi"
```

3) 继续 debug，可以看到序列化后的请求对象，结构是字节数组：

```
ServiceProxy.java x EasyConsumerExample.java x
39         .args(args) args: Object[1]@1422
40         .build();
41     try {
42         // 序列化
43         byte[] bodyBytes = serializer.serialize(rpcRequest); serializer
44         // 发送请求
45         // todo 注意,
46         try (HttpRes
47             .bod
48             .exe
49             byte[] r
50             // 反序列
51             RpcRespo
52             return r
53         }
54     } catch (IOException
```

bodyBytes = {byte[383]@1530} [-84, -19, 0, 5, 115, 114, 0,

01 0 = -84

01 1 = -19

01 2 = 0

01 3 = 5

01 4 = 115

01 5 = 114

01 6 = 0

01 7 = 31

01 8 = 99

01 9 = 111

01 10 = 109

4) 在服务提供者模块的请求处理器中打断点，可以看到接受并反序列化后的请求，跟发送时的内容一致：

```
HttpServerHandler.java x ServiceProxy.java x EasyConsumerExample.java x
32
33 // 异步处理 HTTP 请求
34 request.bodyHandler(body -> { request: Http1xServerRequest@2515 body: "??
35     byte[] bytes = body.getBytes(); bytes: [-84, -19, 0, 5, 115, 114, 0, 31,
36     RpcRequest rpcRequest = null; rpcRequest: "RpcRequest(serviceName=com.yu
37     try {
38         rpcReque
39     } catch (Exc
40         e.printS
41     }
42
43 // 构造响应结果
44 RpcResponse
45 // 如果请求为
46 if (rpcReque
```

rpcRequest = {RpcRequest@2518} "RpcRequest(serviceName=com.yupi.e

> f serviceName = "com.yupi.example.common.service.UserService"

> f methodName = "getUser"

> f parameterTypes = {Class[1]@2887}

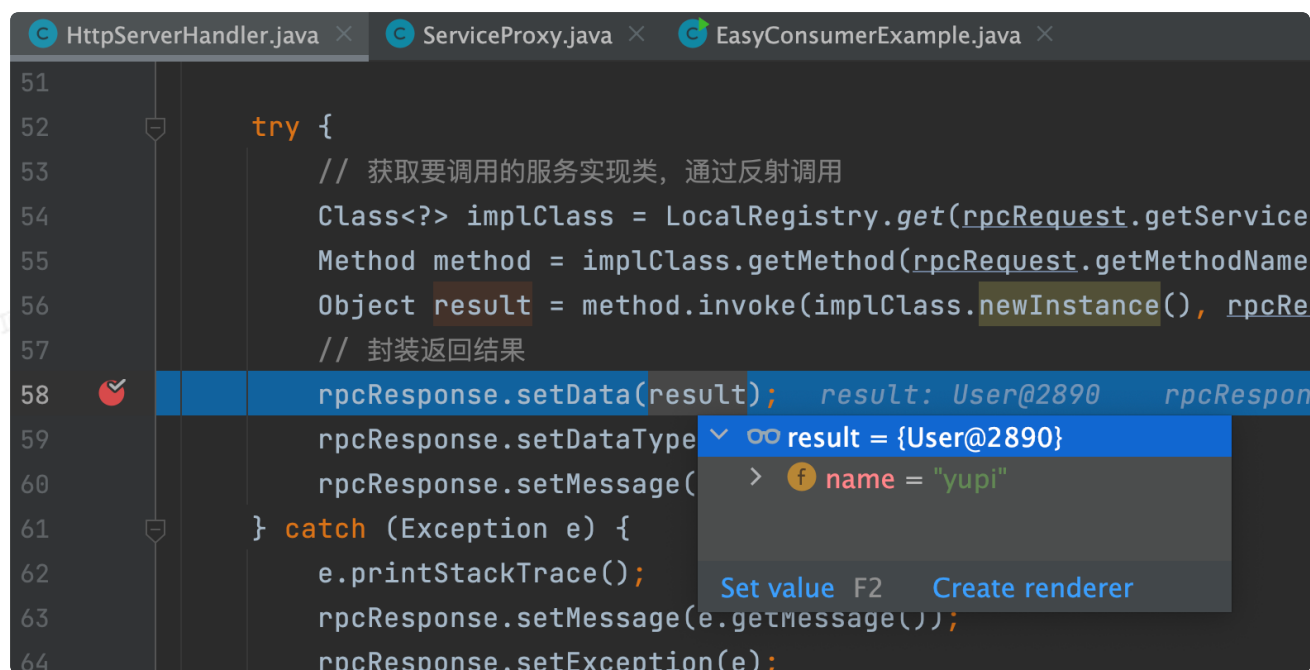
> 0 = {Class@2510} "class com.yupi.example.common.model.User"

> f args = {Object[1]@2888}

> 0 = {User@2890}

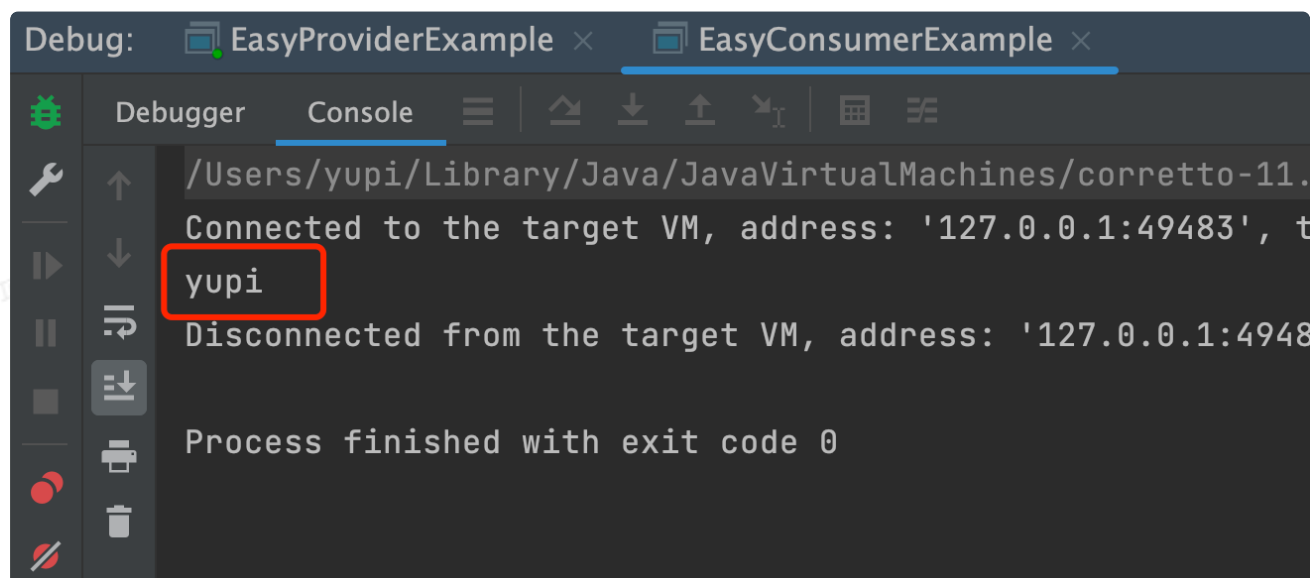
> f name = "yupi"

5) 继续 debug，可以看到在请求处理器中，通过反射成功调用了方法，并得到了返回的 User 对象。



```
51
52     try {
53         // 获取要调用的服务实现类，通过反射调用
54         Class<?> implClass = LocalRegistry.getService
55         Method method = implClass.getMethod(rpcRequest.getMethodName
56         Object result = method.invoke(implClass.newInstance(), rpcRe
57         // 封装返回结果
58         rpcResponse.setData(result); result: User@2890 rpcRespon
59         rpcResponse.setDataType
60         rpcResponse.setMessage(
61     } catch (Exception e) {
62         e.printStackTrace();
63         rpcResponse.setMessage(e.getMessage());
64         rpcResponse.setException(e);
```

6) 最后，在服务提供者和消费者模块中都输出了用户名称，说明整个调用过程成功。



```
Debug: EasyProviderExample x EasyConsumerExample x
Debugger Console
/Users/yupi/Library/Java/JavaVirtualMachines/corretto-11.
Connected to the target VM, address: '127.0.0.1:49483', t
yupi
Disconnected from the target VM, address: '127.0.0.1:4948
Process finished with exit code 0
```

以上，就是本期教程。麻雀虽小，五脏俱全。大家一定要自己动手实现，印象才会更深刻。

url=https%3A%2F%2Fwww.yuque.com%2Fu37765561%2Fak85bt%2F2f3245bbb600c03a30851c65