

华东师范大学数据科学与工程学院实验报告

课程名称：计算机网络与编程

年级：21 级

上机实践成绩：

指导教师：张召

姓名：杨茜雅

学号：10215501435

上机实践名称：Lab13

上机实践日期：2023.6.2

上机实践编号：13

组号：

上机实践时间：2023.6.2

一、实验目的

- 掌握RPC的工作原理
- 掌握反射和代理

二、实验任务

- 编写静态/动态代理代码
- 编写 RPC 相关代码并测试

三、使用环境

- IntelliJ IDEA
- JDK 版本: Java 19

四、实验过程

Task1: 测试并对比静态代理和动态代理，尝试给出一种应用场景，能使用到该代理设计模式。

测试静态代理：

The screenshot displays the IntelliJ IDEA IDE interface. On the left, the 'Project' view shows the file structure of 'Lab13', including 'src' and 'Task1' folders. The 'Task1' folder contains 'Main.java', 'IProxy', 'PersonA', 'PersonB', and 'TestProxy'. The 'Main' class is also visible. The 'Run' view at the bottom shows the execution of 'TestProxy' with the following output:

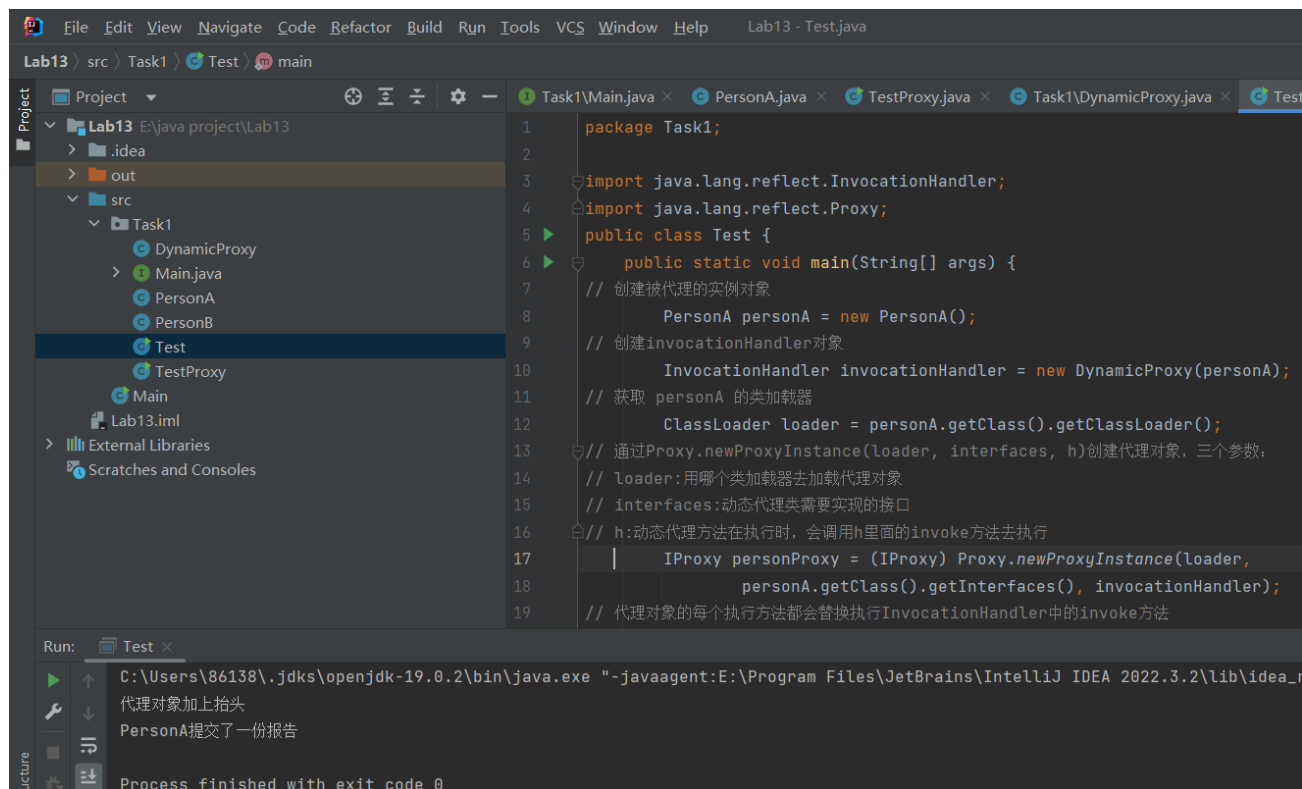
```
Run: TestProxy
C:\Users\86138\.jdk\openjdk-19.0.2\bin\java.exe "-javaagent:E:\Program Files\JetBrains\IntelliJ IDEA\lib\idea_rt.jar=19.0.2:C:\Users\86138\.jdk\openjdk-19.0.2\bin" -jar Lab13.jar
PersonB加上抬头
PersonA提交了一份报告
Process finished with exit code 0
```

The main code window shows the following Java code for 'TestProxy.java':

```
package Task1;

public class TestProxy {
    public static void main(String[] args) {
        // 构造一个PersonA对象
        PersonA personA = new PersonA();
        // 构造一个代理，将personA作为参数传递进去
        IProxy proxy = new PersonB(personA);
        // 由代理者来提交
        proxy.submit();
    }
}
```

测试动态代理：



静态代理：

静态代理是通过手动编写代理类来实现的。在编译时就已经确定了代理类和被代理类的关系。代理类持有对被代理对象的引用，并在调用时调用被代理对象的方法，同时可以在方法前后进行一些额外的操作。

动态代理：

动态代理是在运行时生成代理对象，无需手动编写代理类。它使用Java的反射机制来动态创建代理类和代理对象。动态代理可以根据被代理对象的接口在运行时动态生成代理对象。

动态代理与静态代理的区别：

- (1) 静态代理类：由程序员创建或由特定工具自动生成源代码，再对其编译。程序运行前代理类的.class文件就已经存在了，在代码运行之前，JVM 会读取.class 文件，解析.class 文件内的信息，取出二进制数据，加载进内存中，从而生成对应的Class 对象。
- (2) 动态代理类：程序运行时运用反射机制动态创建而成，在代码运行之前不存在代理类的.class 文件，在代码运行时才动态的生成代理类。

静态代理的应用场景：

- 访问控制：代理可以在调用前进行权限验证，确保只有具有特定权限的用户能够调用被代理对象的方法。
- 日志记录：代理可以在调用前后记录日志，用于调试和跟踪。
- 性能监控：代理可以在调用前后计时，用于监控方法的执行时间。
- 事务管理：代理可以在调用前后开启和提交事务，用于实现事务的一致性。

动态代理的应用场景：

·**AOP（面向切面编程）**：动态代理常用于实现AOP。通过在被代理对象的方法执行前后插入切面逻辑，可以实现横切关注点的统一管理，如日志记录、性能监控、事务管理等。

·**延迟加载**：动态代理可以延迟加载对象，减少初始化时间和资源消耗。只有在真正需要使用对象时才会进行加载和初始化。

·**分布式远程调用**：动态代理可以在分布式系统中远程调用对象的方法，隐藏网络通信细节，简化分布式系统的开发。

例如，一个应用场景是在一个Web应用中使用动态代理实现权限控制。假设有一个用户服务类（UserService），包含各种操作用户的方法（例如创建用户、删除用户等）。我们可以使用动态代理来实现权限控制，只有具有管理员权限的用户才能调用UserService中的敏感方法。代理类可以在调用敏感方法前检查用户的权限，并决定是否允许调用。这样可以在不修改UserService的情况下，实现对敏感方法的访问控制，提高系统的安全性。

Task2: 运行RpcProvider和RpcConsumer，给出一种新的自定义的报文格式，将修改的代码和运行结果截图，并结合代码阐述从客户端调用到获取结果的整个流程。

修改后的RpcProvider:

```
package Task3;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;

public class RpcProvider {
    public static void main(String[] args) {
        Proxy2Impl proxy2Impl = new Proxy2Impl();
        try (ServerSocket serverSocket = new ServerSocket()) {
            serverSocket.bind(new InetSocketAddress( port: 9091));
            try (Socket socket = serverSocket.accept()) {
                DataInputStream dis = new DataInputStream(socket.getInputStream());
                DataOutputStream dos = new DataOutputStream(socket.getOutputStream());

                // 读取请求报文
                String request = dis.readUTF();

                // 解析请求报文
                String methodName = request.substring(0, request.indexOf(":"));
                String argument = request.substring( beginIndex: request.indexOf(":") + 1);

                // 调用代理对象的方法
                String result = proxy2Impl.sayHi(argument);

                // 构造响应报文
                String response = methodName + ":" + result;

                // 发送响应报文
                dos.writeUTF(response);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

修改后的RpcConsumer:

```
package Task3;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.net.InetSocketAddress;
import java.net.Socket;

public class RpcConsumer {
    public static void main(String[] args) {
        String methodName = "sayHi";
        String argument = "alice";

        try (Socket socket = new Socket()) {
            socket.connect(new InetSocketAddress("localhost", 9091));
            DataInputStream dis = new DataInputStream(socket.getInputStream());
            DataOutputStream dos = new DataOutputStream(socket.getOutputStream());

            // 构造请求报文
            String request = methodName + ":" + argument;

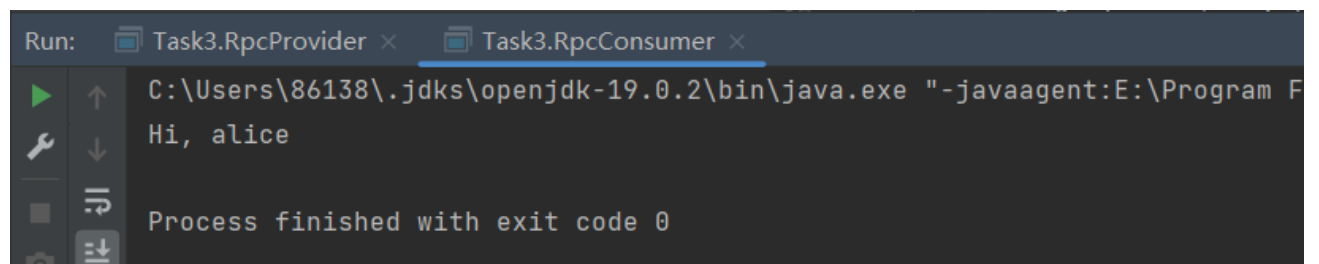
            // 发送请求报文
            dos.writeUTF(request);

            // 接收响应报文
            String response = dis.readUTF();

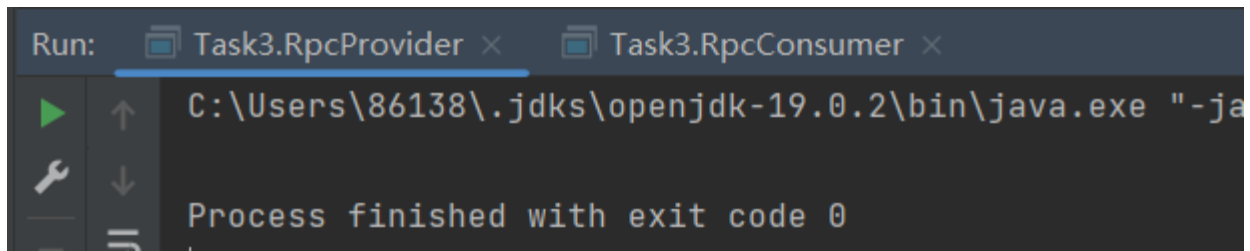
            // 解析响应报文
            String result = response.substring(response.indexOf(":") + 1);

            System.out.println(result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

运行结果:



```
Run: Task3.RpcProvider x Task3.RpcConsumer x
C:\Users\86138\.jdk\openjdk-19.0.2\bin\java.exe "-javaagent:E:\Program F
Hi, alice
Process finished with exit code 0
```



```
Run: Task3.RpcProvider x Task3.RpcConsumer x
C:\Users\86138\.jdk\openjdk-19.0.2\bin\java.exe "-ja
Process finished with exit code 0
```

在这个简单的示例中，我使用了基于文本格式的自定义报文进行通信。请求报文和响应报文都是简单的字符串，以"methodName:argument"和"methodName:result"的形式进行传输。RpcProvider负责监听端口、接收请求报文、解析报文、调用方法并构造响应报文。RpcConsumer负责连接到RpcProvider、构造请求报文并发送，接收响应报文并解析出结果。流程如下：

- 1、RpcProvider启动并绑定到指定端口9091。
- 2、RpcConsumer创建Socket并连接到RpcProvider的地址和端口。
- 3、RpcConsumer构造请求报文，例如"sayHi:alice"。
- 4、RpcConsumer将请求报文发送给RpcProvider。
- 5、RpcProvider接收到请求报文，解析出方法名"sayHi"和参数"alice"。
- 6、RpcProvider调用Proxy2Impl的sayHi方法，并传入参数"alice"。
- 7、RpcProvider得到方法的返回结果，例如"Hi, alice"。
- 8、RpcProvider构造响应报文，例如"sayHi:Hi, alice"。
- 9、RpcProvider将响应报文发送给RpcConsumer。
- 10、RpcConsumer接收到响应报文，解析出方法名"sayHi"和结果"Hi, alice"。
- 11、RpcConsumer输出结果"Hi, alice"。

Task3: 查阅资料，比较自定义报文的RPC和http1.0协议，哪一个更适合用于后端进程通信，为什么？

自定义报文的RPC（Remote Procedure Call）是一种远程过程调用协议，它基于自定义的报文格式进行通信。在自定义报文的RPC中，通信双方使用预定义的报文结构进行数据交换，包括请求和响应的格式、字段、编码规则等。RPC通常使用TCP作为传输层协议。

自定义报文的RPC通常由以下组件组成：

- 通信协议：**自定义报文的RPC使用特定的通信协议进行数据传输，常见的包括基于TCP或UDP的传输协议。通信协议负责在客户端和服务端之间建立连接、传输报文等。
- 报文格式：**自定义报文的RPC使用自定义的报文格式来进行数据交换。报文格式包括请求和响应的结构，以及字段的定义和编码规则。通常会定义报文头部和报文体，报文头部用于标识请求类型、版本信息等，报文体用于携带具体的方法调用和参数。
- 序列化和反序列化：**自定义报文的RPC需要将方法调用和参数转换为字节流进行传输，以及将接收到的字节流转换为方法调用和参数。序列化和反序列化组件负责将方法调用和参数转换为字节流，并进行相应的解析和还原。
- 客户端代理和服务端实现：**自定义报文的RPC使用客户端代理和服务端实现来实现远程方法调用。客户端代理负责将方法调用转换为请求报文，并将请求报文发送给服务端。服务端实

现接收请求报文，解析报文内容，并执行相应的方法调用，最后将执行结果封装为响应报文发送给客户端。

自定义报文的RPC相对于其他RPC框架（如gRPC、Apache Thrift等）具有更高的灵活性，可以根据具体的需求和场景进行定制化设计。它适用于需要细粒度控制报文格式、字段和编码规则的情况。自定义报文的RPC也常常用于特定领域和系统之间的通信，特别是在一些内部使用的系统和组件间通信的场景中。然而，相对于使用标准化的RPC框架，自定义报文的RPC需要额外的开发工作，并且在生态系统和工具支持方面可能较为有限。因此，选择是否使用自定义报文的RPC需要根据具体的需求和权衡来决定。

HTTP 1.0协议是一种应用层协议，用于在客户端和服务端之间进行通信。它使用文本格式的请求和响应报文进行通信，包含请求方法、URL、头部字段、正文等信息。HTTP 1.0使用TCP作为传输层协议。它是HTTP协议的第一个正式版本，于1996年发布。

HTTP 1.0的主要特点和特性如下：

·**请求-响应模型**：HTTP 1.0采用了经典的请求-响应模型，客户端发送请求到服务器，服务器接收请求并返回响应。请求和响应都是由文本格式的报文组成。

·**简单的报文格式**：HTTP 1.0使用文本格式的请求和响应报文进行通信。请求报文包括请求方法（GET、POST等）、URL、头部字段、正文等信息。响应报文包括状态码、状态消息、头部字段、正文等信息。

·**持久连接**：HTTP 1.0引入了持久连接（persistent connection）的概念，允许在单个TCP连接上发送多个HTTP请求和响应，减少了连接的建立和关闭开销，提高了通信的效率。

·**缓存**：HTTP 1.0支持简单的缓存机制，通过在响应头部添加缓存相关的字段，例如Expires和Last-Modified，客户端可以缓存服务器返回的响应，并在下次请求相同资源时使用缓存。

·**无状态协议**：HTTP 1.0是无状态协议，每个请求和响应都是独立的，服务器不会保留客户端的状态信息。为了实现状态管理，引入了Cookie机制。

·**支持多种媒体类型**：HTTP 1.0支持多种媒体类型的传输，通过Content-Type头部字段来指定传输的媒体类型。

·**安全性较低**：HTTP 1.0本身没有内置的加密和安全机制，通信数据以明文方式传输，容易受到窃听和篡改。

需要注意的是，虽然HTTP 1.0在当时是一项重要的协议，但由于其一些限制和性能问题，后来被HTTP 1.1和HTTP 2.0所取代。HTTP 1.1引入了持久连接的默认支持、管道化请求、分块传输编码等特性，提高了性能和效率。而HTTP 2.0更进一步，引入了二进制协议、头部压缩、多路复用等特性，进一步提升了性能和效率，使得HTTP协议更加适应现代互联网应用的需求。

在后端进程通信的场景下，以下是对两者进行比较的一些因素：

·**报文格式和灵活性**：自定义报文的RPC可以根据具体需求设计和定义报文格式，可以更加精确地控制通信的内容和结构。这使得自定义报文的RPC在灵活性方面更胜一筹，可以满足各种定制化的需求。相比之下，HTTP 1.0的报文格式相对固定，虽然可以使用头部字段和正文进行扩展，但相对于自定义报文的RPC而言，其灵活性较低。

·**性能和效率**：由于自定义报文的RPC是为特定场景和需求设计的，其通信的数据量和协议开销可以更加精细地控制，从而提供更高的性能和效率。相比之下，HTTP 1.0的报文格式相对

冗长，会带来一定的协议开销和传输负担，可能会对性能产生一定的影响。

·生态系统和支持： HTTP 1.0是广泛使用的Web通信协议，拥有成熟的生态系统和丰富的工具支持。相比之下，自定义报文的RPC可能需要自行设计和实现，其生态系统相对较小，可能需要更多的开发和维护工作。

综合考虑以上因素，对于后端进程通信，如果有特定的定制化需求，并且对性能和灵活性要求较高，那么自定义报文的RPC可能更适合。但如果更关注生态系统支持和广泛应用，以及对性能和灵活性的要求没有特别高的话，HTTP 1.0协议可能是一个更简单和便捷的选择。

五、总结

通过本次实验，我了解了RPC的工作原理以及java的反射机制和代理机制，自己动手测试了静态代理和动态代理的区别。使用一种新的自定义的报文格式修改了RpcProvider和RpcConsumer的代码。查阅资料学习了自定义报文的Rpc和http1.0协议相关的知识。