

# 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：包亦晟	学号：10215501451
上机实践名称：Java RPC原理及实现	上机实践日期：2023.6.2	
上机实践编号：13	组号：1-451	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**

Java中的代理模式是指给目标对象提供一个代理对象，代理对象包含该目标对象，并控制对该目标对象的访问。代理模式有以下的作用：

- 通过代理对象的隔离，可以在对目标对象访问前后增加额外的业务逻辑，实现功能增强
- 通过代理对象访问目标对象，可以防止系统大量地直接对目标对象进行不正确地访问，出现不可预测的后果

我们特别要注意的一点是，代理对象并没有真正提供服务，它只是替访问者访问目标对象的一个**中转**，实际上真正提供服务的还是目标对象，代理对象仅仅是在目标对象提供服务之前和之后执行一些额外的逻辑。

代理模式分为静态代理和动态代理两类。

静态代理的代理对象，需要我们**事先写好代理类**，实际运行的时候只需要new一个代理对象即可。实验讲义中给出的代码就是事先写好了代理类，之后在测试文件中直接new一个代理类的实例。

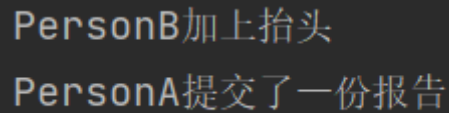
**既然有了静态代理，为什么我们又需要动态代理呢？**这里我们就要说说静态代理所存在的问题。

如果我们现在需要在接口中添加新方法呢？我们就必须手动修改目标对象和代理对象，这很不灵活。而且，在静态代理中，我们要对每个目标类都单独写一个代理类，这实在是有点麻烦。

面对静态代理的这些问题，动态代理便应运而生了。在动态代理中，代理对象是**在运行的时候根据被代理对象的接口信息动态生成的**，它并不具备java源文件。动态代理是通过**反射**机制来实现。动态代理可以代理任意类型的对象。

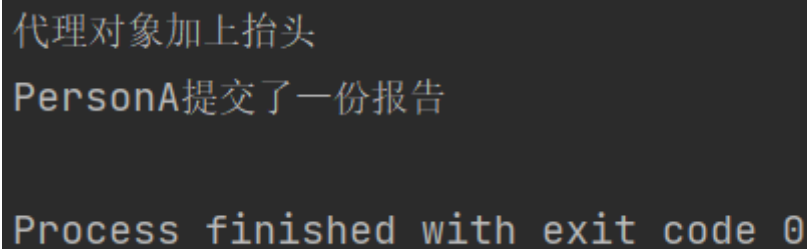
动态代理解决的问题就是：**面对新的需求时，不需要修改代理对象的代码，只需要新增接口和真实对象，在客户端调用即可完成新的代理。**它提高了类的可扩展性和可维护性。

我们来运行下实验中给出的代码，静态代理的结果如下：



```
PersonB加上抬头
PersonA提交了一份报告
```

动态代理的结果如下：



```
代理对象加上抬头
PersonA提交了一份报告

Process finished with exit code 0
```

我们可以发现区别就是在第一行，静态代理中是PersonB，而动态代理中是代理对象。这正好说明了我之前所提过的**在动态代理中，我们不需要事先写好代理类，而是在运行过程中动态生成。**

这里可以做一个简单的类比来促进一下理解。如果静态代理中的代理对象是一个**具体的代理人**的话，那么在动态代理中，实现InvocationHandler接口并实现invoke方法的类就好像是一个**代理中介**。每当你有新的需求的时候，代理中介就会自动帮你找到一个合适的代理人。

当然，动态代理也是有缺点的。它需要利用到反射机制，使得其性能会比静态代理稍差一些，但是比起它的优点，这些劣势几乎可以**忽略不计**。

代理模式一个很常见的应用场景就是**访问控制**。它可以用来控制对实际对象的访问权限。比如，只有特定用户才能访问某些敏感数据。

```
import java.util.Arrays;
import java.util.List;
interface Info {
    void accessInfo();
    void deleteInfo();
}
class PersonC implements Info {
    @Override
    public void accessInfo() {
        System.out.println("成功访问信息");
    }

    @Override
    public void deleteInfo() {
        System.out.println("成功修改信息");
    }
}

class PersonD implements Info {
    private Info productService;
    private List<String> allowedUsers;
```

```

public PersonD(Info productService, List<String> allowedUsers) {
    this.productService = productService;
    this.allowedUsers = allowedUsers;
}

@Override
public void accessInfo() {
    if (isAllowed()) {
        productService.accessInfo();
    } else {
        System.out.println("无权限执行该操作");
    }
}

@Override
public void deleteInfo() {
    if (isAllowed()) {
        productService.deleteInfo();
    } else {
        System.out.println("无权限执行该操作");
    }
}

private boolean isAllowed() {
    String currentUser = getCurrentUser();
    return allowedUsers.contains(currentUser);
}

private String getCurrentUser() {
    return "alice";
}
}

public class Main {
    public static void main(String[] args) {
        Info productService = new PersonD(new PersonC(), Arrays.asList("jack"));
        productService.accessInfo();
        productService.deleteInfo();
    }
}

```

上述代码就实现了一个简单的用静态代理模式实现访问控制的示例。上面的代码中假设了唯一允许的访问者是jack，而当前的访问者是alice，所以当alice想要访问并删除信息的时候，会出现如下的结果：

无权限执行该操作  
无权限执行该操作

## 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。**

这里我第一个不是很理解的地方是序列化和反序列化是什么意思。后来经过查询，序列化指的是把Java对象转换为字节序列的过程，而反序列化是指把字节序列恢复为Java对象的过程。

以下是我自定义的RPC报文格式。

RPC请求报文：

字段名	描述
headerLength	请求报文头部长度（恒定为3字节 2 + 1）
serializeType	数据序列化类型
methodName	需要调用的方法名
parameterTypes	调用的方法所需参数的类型
arguments	调用的方法所需的参数

其中，长度和序列化类型构成了请求报文头部，剩下的则是数据。我这里的请求报文有点简略了，实际上还可以加上整体长度、序号等等字段。

RPC响应报文：

字段名	说明
error	如果方法调用失败，错误类型
isSuccess	是否成功
result	方法调用后的结果

以下是代码：

```
import java.io.Serializable;

public class RpcRequest implements Serializable {
    public static final byte serializeCode = (byte) 0;
    public static final byte length = 3;
    private short headerLength;
    private byte serializeType;
    private String methodName;
    private Class<?>[] parameterTypes;
    private Object[] arguments;

    public RpcRequest(short headerLength, byte serializeType, String methodName,
Class<?>[] parameterTypes, Object[] arguments) {
        this.headerLength = headerLength;
        this.serializeType = serializeType;
        this.methodName = methodName;
        this.parameterTypes = parameterTypes;
        this.arguments = arguments;
    }

    public Object[] getArguments() {
        return arguments;
    }

    public void setArguments(Object[] arguments) {
        this.arguments = arguments;
    }
}
```

```

    public Class<?>[] getParameterTypes() {
        return parameterTypes;
    }

    public void setParameterTypes(Class<?>[] parameterTypes) {
        this.parameterTypes = parameterTypes;
    }

    public short getHeaderLength() {
        return this.headerLength;
    }
    public void setHeaderLength(short headerLength) {
        this.headerLength = headerLength;
    }
    public byte getSerializeType() {
        return this.serializeType;
    }
    public void setSerializeType(byte serializeType) {
        this.serializeType = serializeType;
    }
    public String getMethodName() {
        return this.methodName;
    }
    public void setMethodName(String methodName) {
        this.methodName = methodName;
    }
}

```

```

import java.io.Serializable;

public class RpcResponse implements Serializable {
    public static final boolean SUCCESS = true;
    public static final boolean FAILURE = false;
    private Object result;
    private Throwable error;
    private boolean isSuccess;

    public RpcResponse(Object result, Throwable error, boolean isSuccess) {
        this.result = result;
        this.error = error;
        this.isSuccess = isSuccess;
    }

    public Object getResult() {
        return result;
    }

    public void setResult(Object result) {
        this.result = result;
    }

    public Throwable getError() {
        return error;
    }
}

```

```

    public void setError(Throwable error) {
        this.error = error;
    }

    public boolean isSuccess() {
        return isSuccess;
    }

    public void setSuccess(boolean success) {
        isSuccess = success;
    }
}

```

可以看到我在每个类中都定义了一些常量。

RpcProvider:

```

import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
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(9091));
            try(Socket socket = serverSocket.accept()) {
                // ObjectInputStream/ObjectOutputStream 提供了将对象序列化和反序列化的功能
                ObjectInputStream is = new
                    ObjectInputStream(socket.getInputStream());
                // rpc提供方和调用方之间协商的报文格式和序列化规则
                Object rpcObj = is.readObject();
                RpcRequest rpc = (RpcRequest) rpcObj;
                if (rpc.getSerializeType() == RpcRequest.serializeCode) {
                    String methodName = rpc.getMethodName();
                    Class<?>[] parameterTypes = rpc.getParameterTypes();
                    Object[] arguments = rpc.getArguments();
                    Object result =

                    Proxy2Impl.class.getMethod(methodName,parameterTypes).invoke(proxy2Impl,
                    arguments);

                    new
                    ObjectOutputStream(socket.getOutputStream()).writeObject(new RpcResponse(result,
                    null, RpcResponse.SUCCESS));

                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

RpcConsumer:

```

import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.net.InetSocketAddress;
import java.net.Socket;
public class RpcConsumer {
    public static void main(String[] args) {
        IProxy2 iProxy2 = (IProxy2) Proxy.newProxyInstance(
            IProxy2.class.getClassLoader(), new Class<?>[]{IProxy2.class},
            new iProxy2Handler()
        );
        System.out.println(iProxy2.sayHi("alice"));
    }
}
class iProxy2Handler implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
    Throwable {
        Socket socket = new Socket();
        socket.connect(new InetSocketAddress(9091));
        ObjectOutputStream os = new
        ObjectOutputStream(socket.getOutputStream());
        // rpc提供方和调用方之间协商的报文格式和序列化规则
        RpcRequest rpc = new RpcRequest(RpcRequest.length,
        RpcRequest.serializeCode,method.getName(), method.getParameterTypes(), args);
        os.writeObject(rpc);
        Object result = new
        ObjectInputStream(socket.getInputStream()).readObject();
        RpcResponse rpcRes = (RpcResponse) result;
        return rpcRes.getResult();
    }
}

```

我接下来说一下RpcProvider和RpcConsumer分别在做什么事情。

RpcProvider:

- 创建一个ServerSocket并绑定到9091端口，等待连接
- 一旦有客户端连接成功，创建Socket对象
- 通过ObjectInputStream从Socket的输入流中读取数据
- 识别RPC请求报文的头部信息，并读取方法名，参数类型和参数值
- 使用反射调用Proxy2Impl类的相应方法，并将参数传递给该方法
- 将方法的返回结果封装到RpcResponse当中，并通过ObjectOutputStream序列化并写入Socket的输出流，返回给客户端

RpcConsumer:

- 创建一个iProxy2Handler对象，实现了InvocationHandler接口，用于处理代理对象的方法调用
- 使用Proxy.newProxyInstance方法创建一个代理对象iProxy2，将代理对象的方法调用转发到iProxy2Handler处理
- 调用代理对象iProxy2的方法，实际上会调用iProxy2Handler的invoke方法
- 在invoke方法中，创建一个Socket对象并连接到RPC提供方的9091端口
- 将请求头部的信息和方法名、参数类型和参数值封装到RpcRequest中
- 通过ObjectOutputStream将RpcRequest写入Socket的输出流

- 通过ObjectInputStream从Socket的输入流中读取返回的结果，并返回给调用方

以下是从客户端调用到获取结果的整个流程：

客户端将请求报文头部信息和想要远程调用的方法名、参数类型以及参数封装到RpcRequest类当中，发送给服务器。服务器接收到之后，解析出头部信息并加以检查。通过后就根据传过来的方法名，执行相应的方法，将返回值和响应头部信息封装到RpcResponse中返回给客户端。客户端对响应报文进行解析，得到方法的返回值。

运行结果：

```
Hi, alice

Process finished with exit code 0
```

## 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是一种基于TCP协议的一种远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。HTTP1.0是一种基于TCP协议的无状态，无连接的应用层协议。

它们两者之间在以下几个方面有着不同：

- 报文格式：
  - 自定义报文的RPC协议可以根据具体需求进行设计，灵活性高。可以自定义报文头和体的结构，选择适合的序列化方式
  - HTTP 1.0协议是有着固定的报文格式的。虽然可以通过自定义HTTP头来传递额外的信息，但相对于自定义报文格式的RPC协议来说，扩展性较差
- 序列化方式：
  - 自定义报文的RPC协议可以灵活选择序列化和反序列化的方式，例如使用Java的序列化、JSON、XML等
  - HTTP 1.0协议通常使用文本格式传输数据，如JSON或XML，需要在请求和响应中进行序列化和反序列化。这可能会带来一些额外的开销，并且对于大规模的数据传输效率可能较低
- 功能特性：
  - 自定义报文的RPC协议可以根据具体需求进行定制，可以支持丰富的功能特性，如异步调用和负载均衡等，以满足后端进程通信的复杂需求
  - HTTP 1.0协议相对较简单，功能特性相对有限，主要用于传输请求和响应数据，不太适合复杂的后端进程通信场景

从上述几点来看，对于后端进程通信来说，自定义报文的RPC协议可能更适合。它可以根据具体需求进行定制，提供更高的灵活性和扩展性，支持更多的功能特性，并能选择更适合的序列化方式。而HTTP 1.0协议相对简单，适用于简单的请求和响应数据传输场景。

## 五、总结

本次的实验内容对我来说很具有难度，因为有很多概念我原先根本就不知晓，比如说代理模式以及序列化。我是在网上查询阅读了大量的资料才搞清楚了代理模式的意义何在以及静态代理和动态代理的不同之处。



Task2是对我来说是一个非常新颖的任务。因为过去的实验一直在让我们使用wireshark抓包，去分析各种各样的协议，但是今天还是第一次让我们自己来设计协议，让我们自己决定该放些什么字段，挺有趣味性的。

张老师说本次实验学习的RPC是对我们日后做系统中非常非常重要的知识点，我相信通过本次实验我已经有了初步的了解了，希望以后自己能够更加深入地进行学习。