

华东师范大学数据科学与工程学院上机实践报告

课程名称：计算机网络与编程	年级：2022	上机实践成绩：
指导教师：张召	姓名：李芳	学号：10214602404
上机实践名称： Week13_Java RPC 原理及实现		上机实践日期：2024.05.31
上机实践编号：13	组号：	上机实践时间：

一、题目要求及实现情况

- **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。

代码测试结果：

➤ 静态代理测试结果：

```
34 public class TestProxy {
35     public static void main(String[] args) {
36         // 构造一个PersonA对象
37         PersonA personA = new PersonA();
38         // 构造一个代理，将personA作为参数传递进去
39         IProxy proxy = new PersonB(personA);
40         // 由代理者来提交
41         proxy.submit();
42     }
43 }
44
```

运行 TestProxy x

"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java_idea\IntelliJ\lib\idea_rt.jar=5043:E:\java_idea\IntelliJ\bin" -Dfile.encoding=UTF-8

PersonB加上抬头
PersonA提交了一份报告

进程已结束，退出代码为 0

➤ 动态代理测试结果：

```
6 public class Test {
7     public static void main(String[] args) {
8         // 创建被代理的实例对象
9         PersonA personA = new PersonA();
10        // 创建InvocationHandler对象
11        InvocationHandler invocationHandler = new DynamicProxy(personA);
12        // 获取 personA 的类加载器
13        ClassLoader loader = personA.getClass().getClassLoader();
14        // 通过Proxy.newProxyInstance(loader, interfaces, h)创建代理对象，三个参数：
15        // loader:用哪个类加载器去加载代理对象
16        // interfaces:动态代理类需要实现的接口
17        // h:动态代理方法在执行时，会调用h里面的invoke方法去执行
18        IProxy personProxy = (IProxy) Proxy.newProxyInstance(loader,
19            personA.getClass().getInterfaces(), invocationHandler);
20        // 代理对象的每个执行方法都会替换执行InvocationHandler中的invoke方法
21        personProxy.submit();
22    }
23 }

```

运行 Test x

"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java_idea\IntelliJ\lib\idea_rt.jar=5043:E:\java_idea\IntelliJ\bin" -Dfile.encoding=UTF-8

代理对象加上抬头
PersonA提交了一份报告

进程已结束，退出代码为 0

对比两种代理:

➤ 静态代理:

- 在编译期间就已经确定代理对象的方式，代理类是在编译期间就创建的。
- 为每个被代理的类创建一个代理类，导致类的数量增加。

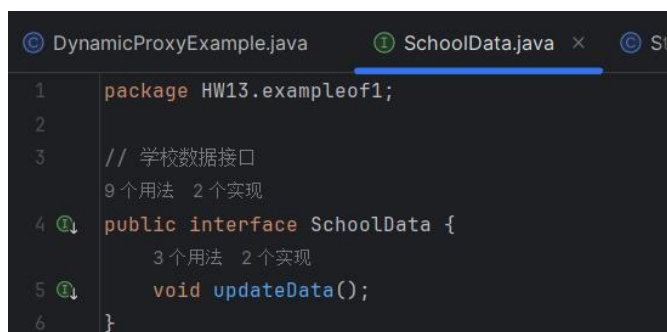
➤ 动态代理:

- 在运行时动态生成代理对象，不需要为每个被代理的类创建一个代理类。
- 更加灵活，可以减少重复代码的编写，并且可以在运行时动态改变代理行为。

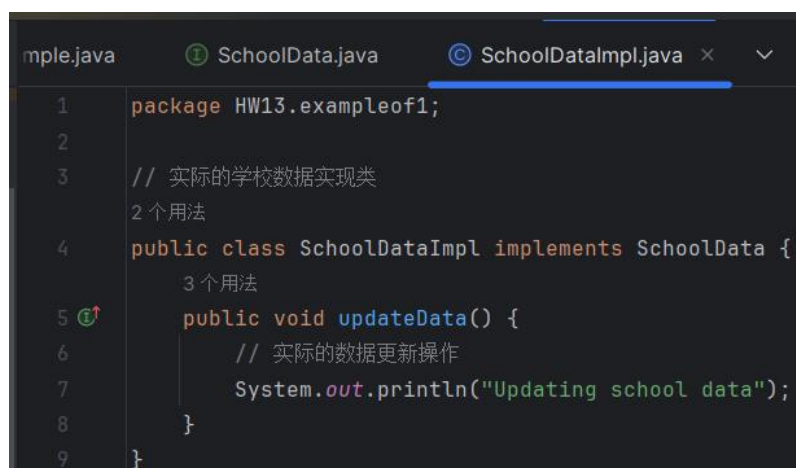
应用场景:

- 假设有一个学校数据平台系统，其中包括学生管理模块和教师管理模块，可以使用代理模式来控制对这些模块的访问:

- 假设一个接口 `SchoolData` 用于管理学校数据，同时有一个实际的实现类 `SchoolDataImpl`。



```
DynamicProxyExample.java SchoolData.java x St.
1 package HW13.exampleof1;
2
3 // 学校数据接口
  9 个用法 2 个实现
4 public interface SchoolData {
  3 个用法 2 个实现
5     void updateData();
6 }
```



```
mple.java SchoolData.java SchoolDataImpl.java x v
1 package HW13.exampleof1;
2
3 // 实际的学校数据实现类
  2 个用法
4 public class SchoolDataImpl implements SchoolData {
  3 个用法
5     public void updateData() {
6         // 实际的数据更新操作
7         System.out.println("Updating school data");
8     }
9 }
```

- 静态代理: 创建 `SchoolDataProxy` 来代理 `SchoolData` 接口的实现，可以添加其他代码在访问实际实现类之前或之后执行一些额外的操作（权限验证、日志记录）。主函数中，先创建了实际的学校数据实现类 `realSubject`，然后创建了静态代理对象 `proxy`，最后通过代理对象调用了 `updateData` 方法。

```
DynamicProxyExample.java StaticProxyExample.java x
4 class SchoolDataProxy implements SchoolData {
5     2个用法
6     private SchoolData schoolData;
7     1个用法
8     public SchoolDataProxy(SchoolData schoolData) { this.schoolData = schoolData; }
9
10    3个用法
11    public void updateData() {
12        // 执行一些额外操作, 比如权限验证、日志记录等
13        System.out.println("Permission check before updating data");
14        // 调用实际的数据更新操作
15        schoolData.updateData();
16        // 执行一些额外操作, 比如日志记录等
17        System.out.println("Log after updating data");
18    }
19 }
20 public class StaticProxyExample {
21     public static void main(String[] args) {
22         // 创建实际的学校数据实现类
23         SchoolData realSubject = new SchoolDataImpl();
24
25         // 创建静态代理对象
26         SchoolData proxy = new SchoolDataProxy(realSubject);
27
28         // 调用代理对象的方法
29         proxy.updateData();
30     }
31 }
```

```
运行 StaticProxyExample x
"C:\Program Files\Java\jdk-20\bin\java.
Permission check before updating data
Updating school data
Log after updating data
进程已结束, 退出代码为 0
```

- 动态代理: 创建一个 `SchoolDataInvocationHandler` 类作为动态代理的处理器, 它同样可以像静态代理一样, 在实际调用前后执行了一些额外的操作。然后使用 `Proxy.newProxyInstance` 方法创建了动态代理实例, 并调用了 `updateData` 方法来触发代理处理器的逻辑。

```
DynamicProxyExample.java StaticProxyExample.java
8 class SchoolDataInvocationHandler implements InvocationHandler {
9     2个用法
10    private Object target;
11
12    1个用法
13    public SchoolDataInvocationHandler(Object target) {
14        this.target = target;
15    }
16
17    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
18        System.out.println("Permission check before updating data");
19        Object result = method.invoke(target, args);
20        System.out.println("Log after updating data");
21        return result;
22    }
23 }
24 // 示例代码
25 public class DynamicProxyExample {
26     public static void main(String[] args) {
27         SchoolData realSubject = new SchoolDataImpl();
28         SchoolData proxyInstance = (SchoolData) Proxy.newProxyInstance(
29             realSubject.getClass().getClassLoader(),
30             realSubject.getClass().getInterfaces(),
31             new SchoolDataInvocationHandler(realSubject)
32         );
33         proxyInstance.updateData();
34     }
35 }
```

```
运行 DynamicProxyExample x
"C:\Program Files\Java\jdk-20\bin\java.exe
Permission check before updating data
Updating school data
Log after updating data
进程已结束, 退出代码为 0
```

- **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。

修改代码& 运行结果:

➤ 报文格式:

- 方法名的长度 (int)
- 方法名 (UTF-8)
- 参数的数量 (int)
- 对于每个参数:
 - ◆ 参数类型 (UTF-8)
 - ◆ 参数值 (Object)

➤ 修改代码:

■ RpcProvider:

首先，读取方法名的长度（以字节为单位），再根据读取的长度，从输入流中读取方法名；然后，读取参数的数量；最后，循环读取每个参数的类型（以字符串形式）和参数值（作为对象）

```

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()) {
                // 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
                String methodName = is.readUTF();
                Class<?>[] parameterTypes = (Class<?>[]) is.readObject();
                Object[] arguments = (Object[]) is.readObject();

                int methodNameLength = is.readInt();
                String methodName = readString(is, methodNameLength);
                int parameterCount = is.readInt();
                Class<?>[] parameterTypes = new Class<?>[parameterCount];
                Object[] arguments = new Object[parameterCount];

                for (int i = 0; i < parameterCount; i++) {
                    parameterTypes[i] = Class.forName(is.readUTF());
                    arguments[i] = is.readObject();
                }

                // rpc提供方调用本地的对象的方法
                Object result = Proxy2Impl.class.getMethod(methodName, parameterTypes).invoke(proxy2Impl, arguments);
                // 将结果序列化并返回
            }
        }
    }

    private static String readString(ObjectInputStream is, int length) throws Exception {
        byte[] bytes = new byte[length];
        is.readFully(bytes);
        return new String(bytes);
    }
}

```

添加一个 readString()函数：根据指定的长度，从输入流中读取字节，并将其转换为字符串。

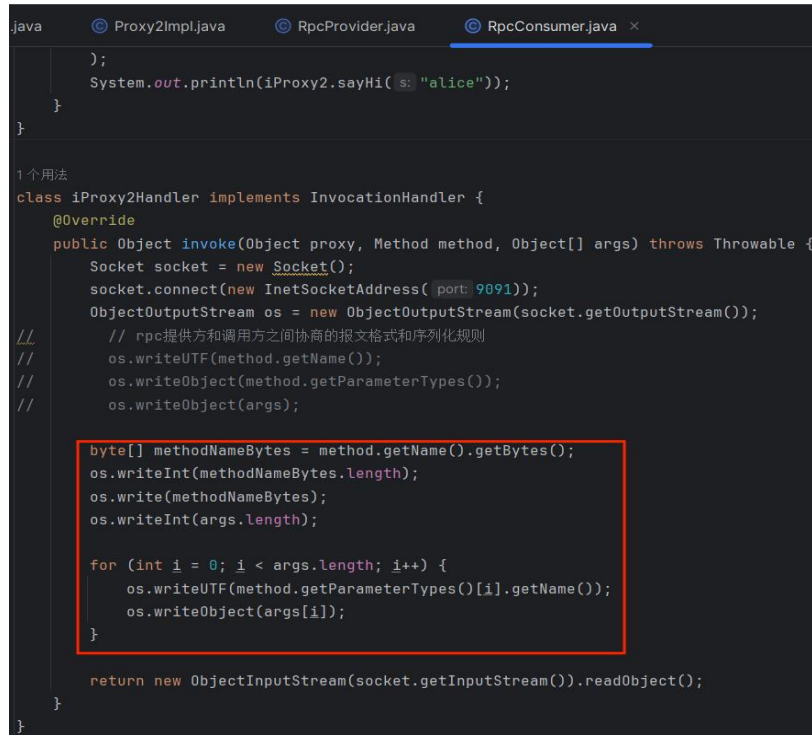
```

private static String readString(ObjectInputStream is, int length) throws Exception {
    byte[] bytes = new byte[length];
    is.readFully(bytes);
    return new String(bytes);
}

```

■ RpcConsumer:

首先，获取方法名的字节数组，写入长度后再写入方法名本身；然后，写入参数的数量；最后，循环写入每个参数的类型名称和参数值。



```
java Proxy2Impl.java RpcProvider.java RpcConsumer.java x
    };
    System.out.println(iProxy2.sayHi("alice"));
}

1 个用法
class iProxy2Handler implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        Socket socket = new Socket();
        socket.connect(new InetSocketAddress("127.0.0.1", 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
        // os.writeUTF(method.getName());
        // os.writeObject(method.getParameterTypes());
        // os.writeObject(args);

        byte[] methodNameBytes = method.getName().getBytes();
        os.writeInt(methodNameBytes.length);
        os.write(methodNameBytes);
        os.writeInt(args.length);

        for (int i = 0; i < args.length; i++) {
            os.writeUTF(method.getParameterTypes()[i].getName());
            os.writeObject(args[i]);
        }

        return new ObjectInputStream(socket.getInputStream()).readObject();
    }
}
```

➤ 为了确保客户端和服务端之间的通信遵循正确一致的报文格式，主要的修改操作包括：

- 添加方法名长度和参数数量的读写操作，符合新的报文格式。
- 修改方法名和参数的序列化和反序列化逻辑，对应新的自定义格式。
- RpcProvider 中添加方法 readString()来根据长度读取字符串。

➤ 运行结果：和修改前的示例代码运行结果一致。



```
运行 RpcProvider x RpcConsumer x
"C:\Program Files\Java\jdk-20\bin\java.exe"
进程已结束，退出代码为 0

运行 RpcProvider x RpcConsumer x
"C:\Program Files\Java\jdk-20\bin\java.exe"
Hi, alice
进程已结束，退出代码为 0
```

流程解析:

➤ RpcConsumer:

- 创建动态代理实例。

- 调用代理方法 sayHi，触发 iProxy2Handler 的 invoke 方法。
- 连接服务器，发送方法名长度、方法名、参数数量和参数数据。
- RpcConsumer:
 - 接受客户端连接，读取输入流中的方法名长度、方法名、参数数量和参数数据。
 - 通过反射调用本地方法 sayHi。
 - 将调用结果序列化后发送回客户端。
- RpcConsumer:
 - 接收服务器返回的结果，并输出。

- **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:

- 一种基于消息传递的通信模式，允许一个进程调用另一个进程的函数或方法，就像本地调用一样。
- 报文：通常会经过序列化和反序列化。
- 优点：
 - 效率高：可以根据具体需求进行定制，更好地满足通信的需求，提高效率。
 - 灵活性：可以灵活定义报文格式、编码方式等，适应不同的应用场景和需求。
 - 性能优化：可以针对性地进行性能优化，提高通信效率。
- 缺点：
 - 复杂性：实现相对复杂，需要额外的开发和维护工作。
 - 兼容性：需要额外处理跨语言和跨平台的兼容操作。

http1.0 协议:

- 一种应用层协议，通过请求-响应的方式进行通信，
- 报文：有固定报文格式（如请求行、请求头部、空行和消息体）。
- 优点：
 - 简单易用：广泛应用于互联网中，实现相对简单。
 - 标准化：属于标准化协议，具有良好的兼容性和稳定性。
- 缺点：

- 性能限制：每次请求都需要建立和断开连接，存在较大的开销。
- 功能受限：功能相对受限，不够灵活，无法满足一些特定的通信需求。

更适宜后端进程通信？

- 如果需要更直接的进程间通信，对通信效率和灵活性有较高要求，可以接受额外的开发和维护工作，并且对跨语言和跨平台的兼容性要求不高，RPC 更适合。
- 如果需要利用现有的基础设施和工具，简单快速地实现通信，对性能要求不是特别高，并且对跨语言和跨平台的兼容性有较高要求，http1.0 更适合。

二、总结

这次的三个实验任务都各有收获：

Task1 通过实现静态代理和动态代理，我深入理解了代理设计模式的应用场景；通过运行两种情况下的 test 测试代理的实现原理，让我清楚了每一步是怎样实现的。同时，我还自己实现了一个两种代理会用的简单的适用场景，也就是学校数据管理平台的一些操作，也算是巩固复习了这部分的代码操作。

Task2 通过修改 RpcProvider 和 RpcConsumer，我定义了自定义报文格式，还能够描述出客户端到服务器的调用流程，掌握住了 RPC 的工作原理；其中，有遇到没有把两端操作对等的小错误，后面通过报错调试解决了。

最后，Task3 通过比较自定义报文的 RPC 和 HTTP1.0 协议，我明确了自定义报文的 RPC 在后端进程通信中的优势和适用场景，同时更加了解了两者的区别与联系。

本次实验让我在 RPC 这方面有了深入的了解，并且还实践和修改了代码，提高了我的动手和转化能力，收获很大。