Week 13：Java RPC原理及实现

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

1. 什么是RPC (Remote Procedure Call)

- RPC协议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，像调用本地函数一样去调远程函数。
  - 本地过程调用

```
int Multiply(int l, int r) {
    int y = l * r;
    return y;
}
int lvalue = 10;
int rvalue = 20;
int l_times_r = Multiply(lvalue, rvalue);
```

  - 从本地调用变成远程调用需要考虑的问题
    - 函数指针
    - 参数序列化与反序列化
    - 网络传输

2. Java反射机制 (Reflection)

- 允许运行中的Java程序**动态获取类/对象信息、动态调用对象的方法**，包括：
  - 在运行时获知任意一个对象所属的类；
  - 在运行时构造任意一个类的对象；

- 在运行时获知任意一个类所具有的成员变量和方法；
- 在运行时调用任意一个对象的方法和属性；

```
Public Object invoke(Object implicitPara, Object[] explicitPara)
// 通过Method类的invoke方法来调用某对象的任意方法 参数1：为实现类 参数2：为方法参数
```

> 反射提高了程序的灵活性，允许程序在运行时动态地创建和控制任何类的对象，无需提前硬编码目标类。

3. 代理

给某一个对象提供代理，并由代理对象来控制对真实对象的访问

3.1 静态代理

- 编写一个接口

```
interface IProxy {
    void submit();
}
```

- 编写一个PersonA类

```
class PersonA implements IProxy {
    @Override
    public void submit() {
        System.out.println("PersonA提交了一份报告");
    }
}
```

- 编写一个PersonB类

```
// 编写类PersonB
class PersonB implements IProxy {
    //被代理者的引用
    private IProxy m_Person;

    PersonB(IProxy person) {
        m_Person = person;
    }

    @Override
    public void submit() {
        before();
        m_Person.submit();
    }

    public void before() {
        System.out.println("PersonB加上抬头");
```

```
    }
}
```

- 编写测试类

```
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();
    }
}
```

> 虽然静态代理实现简单，不侵入式修改源代码，但容易产生过多的代理类且不易维护。

3.2 动态代理

- 编写DynamicProxy类（用到反射机制）

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
/**
* DynamicProxy类持有一个被代理类的对象
* InvocationHandler接口用于实现动态代理
* 实现了InvocationHandler接口的invoke方法，
* 执行所有代理对象的方法都会被替换成执行接口的invoke方法
*/
public class DynamicProxy implements InvocationHandler {
    private Object obj; //被代理类的对象（Object类型），接受任意类型对象

    DynamicProxy(Object _ojb) {
        obj = _ojb;
    }

    @Override
    /**
     * 这个方法不是我们显式地去调用
     * proxy:代表代理对象，
     * method:代表正在执行的方法，
     * args:代表调用目标方法时传入的实参
    */
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        before();
        Object result = method.invoke(obj, args);
        return result;
    }
```

```
    private void before() {
        System.out.println("代理对象加上抬头");
    }
}
```

- **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。**
- 编写测试类

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;

public class Test {
    public static void main(String[] args) {
        // 创建被代理的实例对象
        PersonA personA = new PersonA();
        // 创建invocationHandler对象
        InvocationHandler invocationHandler = new DynamicProxy(personA);
        // 获取 personA 的类加载器
        ClassLoader loader = personA.getClass().getClassLoader();

        // 通过Proxy.newProxyInstance(loader, interfaces, h)创建代理对象，三个参数:
        // loader:用哪个类加载器去加载代理对象
        // interfaces:动态代理类需要实现的接口
        // h:动态代理方法在执行时，会调用h里面的invoke方法去执行
        IProxy personProxy = (IProxy) Proxy.newProxyInstance(loader,
personA.getClass().getInterfaces(), invocationHandler);

        // 代理对象的每个执行方法都会替换执行InvocationHandler中的invoke方法
        personProxy.submit();
    }
}
```

4. RPC实现

- 编写一个接口类

```
public interface IProxy2 {
    String sayHi(String s);
}
```

- 编写实现该接口的类

```
public class Proxy2Impl implements IProxy2 {
    @Override
    public String sayHi(String s) {
        return "Hi, " + s;
    }
}
```

- 编写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/ObjectOutStream 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();

                // 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
                new ObjectOutputStream(socket.getOutputStream()).writeObject(result);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- 编写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
        os.writeUTF(method.getName());
        os.writeObject(method.getParameterTypes());
        os.writeObject(args);

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

- **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。
- **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?