

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：2021 级

上机实践成绩：

指导教师：张召

姓名：彭一坤

学号：10215501412

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：

上机实践时间：9:50

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

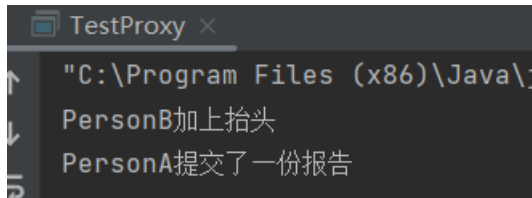
IntelliJ IDEA

JDK 版本：Java 19

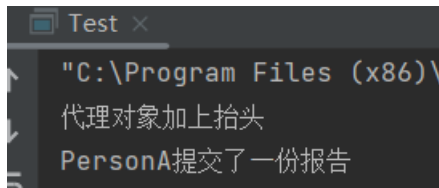
四、实验过程

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。

运行 TestProxy，得到如下结果：



运行 Test，得到如下结果：



通过运行结果可以看出，动态代理绕过了代理类 PersonB，运用反射机制，允许程序在运行时动态获取 PersonA 的类中的对象，无需提前硬编码目标类。

静态代理和动态代理都是代理模式的实现方式，不同之处在于静态代理需要在编译时确定代理类，而动态代理则是在运行时通过 getClass 方法动态生成代理类。

代理技术用于在不修改原始类的情况下，为原始类的方法提供额外的功能。在 RPC 技术中，服务器存在大量的服务类，比如处理用户上传操作和处理用户的下载操作，如果客户端想要调用上传操作，就可以生成一个代理类，代理了服务器中实际的实现类，给客户端提供服务。但是如果服务器的这些方法都需要打印日志，如果用静态代理，那么每

个方法的代理类需要分别实现重复的打印日志方法。如果使用动态代理，就可以统一处理这些类，只加入一次打印日志的方法就可以了。

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。

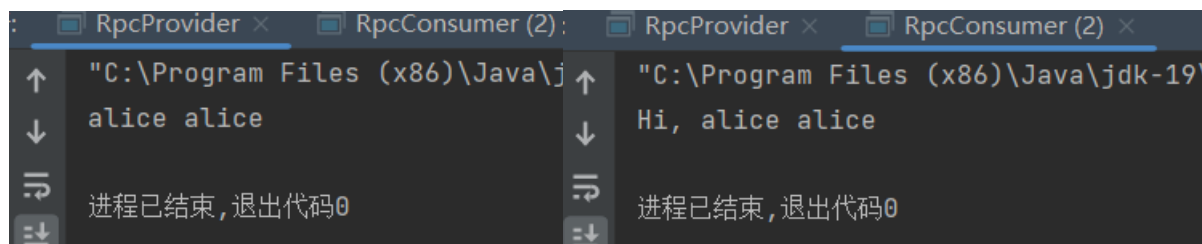
在客户端发送报文的过程中，往序列中输入了参数的个数。

```
os.writeInt(args.length);
for (Object obj : args) {
    os.writeObject(obj);
}
```

在服务器解析参数的过程中，先读取参数个数 lengthArgs，然后创建对象 Object，按照长度读取流中的参数。最后打印一下参数进行验证。

```
int lengthArgs = is.readInt();
Object[] arguments = new Object[lengthArgs];
for (int i = 0; i < lengthArgs; i++) {
    arguments[i] = is.readObject();
}
System.out.println(arguments[0]);
```

程序运行结果如下：



The screenshot shows two side-by-side command prompt windows. The left window, titled 'RpcProvider', shows the command 'C:\Program Files (x86)\Java\jdk-19\bin\java.exe' and the output 'alice alice'. The right window, titled 'RpcConsumer', shows the command 'C:\Program Files (x86)\Java\jdk-19\bin\java.exe' and the output 'Hi, alice alice'. Both windows show '进程已结束,退出代码0' (Process ended, exit code 0) at the bottom.

完整代码如下：

客户端 RpcConsumer 首先创建了一个动态代理对象 iProxy2，实现了类 IProxy2 的接口。

然后客户端调用 iProxy2 的方法 sayHi，参数为 alice alice。

在客户端调用这个方法时，实际上通过动态代理机制，去执行接口的 invoke 方法。

Invoke 方法运用反射机制，调用对象的任意方法。此处的 invoke 方法连接了服务器，并向服务器写入一个报文，报文格式是：方法名、参数类型、参数长度、参数。

```
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(s: "alice 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(port: 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.writeInt(args.length);
        for (Object obj : args) {
            os.writeObject(obj);
        }
        return new ObjectInputStream(socket.getInputStream()).readObject();
    }
}
```

对于服务器，首先阻塞等待客户端连接。

然后按照格式读取客户端发来的报文。

最后调用 `getMethod` 方法，获取 `ProxyImpl` 类中的特定方法，并用 `invoke` 调用这个方法。

```
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();
                int lengthArgs = is.readInt();
                Object[] arguments = new Object[lengthArgs];
                for (int i = 0; i < lengthArgs; i++) {
                    arguments[i] = is.readObject();
                }
                System.out.println(arguments[0]);
                Object result;
                result = Proxy2Impl.class.getMethod(methodName,parameterTypes).invoke(proxy2Impl, arguments);
                // rpc提供方调用本地的对象的方法
                // 将结果序列化并返回
                new ObjectOutputStream(socket.getOutputStream()).writeObject(result);
            }
        }
    }
}
```

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 是一种基于二进制协议的远程过程调用（RPC）框架，它使用特殊的消息格式来编码请求和响应参数，以提高性能。相比之下，HTTP1.0 协议是基于文本的协议，其消息头和主体都是文本形式，因此存在一定的传输开销。

自定义报文的 RPC 通常比 HTTP1.0 更快，因为它不需要像 HTTP1.0 那样解析和生成文本消息。另外，自定义报文的 RPC 还支持多路复用，可以在一个连接上同时处理多个请求，从而减少了建立和关闭连接的开销。因此，自定义报文的 RPC 更适合后端进程间的通信，因为这些进程可能不需要规范化的协议，例如 HTTP1.0，而是由开发者自定义一套规则用于 RPC 即可。

五、总结

Java 中的代理技术是一种常用的设计模式，可以通过代理对象来控制对另一个对象的访问。在 Java 中，代理技术主要包括静态代理和动态代理两种。

静态代理是在编译时就已经确定代理对象的实现，程序运行时代理类并不会发生变化。静态代理相对简单，易于理解和实现，且可以提前检查代理对象的合法性，确保代理对象符合要求。但是，静态代理需要为每个被代理的类编写一个代理类，当被代理类很多时，这会导致代码冗余，并且维护成本较高。

动态代理是在运行时根据接口创建代理对象，可以动态地代理多个类或者接口，避免了静态代理中的代码冗余问题。动态代理还可以通过 `InvocationHandler` 接口进行拦截和处理原始方法调用，从而实现更灵活的代理逻辑。但是，动态代理的实现过程较复杂，需要使用 Java 反射技术，可能导致一定的性能损失。

RPC 运用了动态代理技术，实现了客户端和服务端的通信，根据需要动态生成代理对

象，并且在代理对象上执行方法调用时，将实际的远程调用发送到服务端。