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：22级	上机实践成绩：
指导教师：张召	姓名：朱天祥	
上机实践名称：Java RPC原理及实现	学号：10225501461	上机实践日期：23/6/2
上机实践编号：No.13	组号：	

一、目的

掌握RPC的工作原理

掌握反射和代理

二、实验内容

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

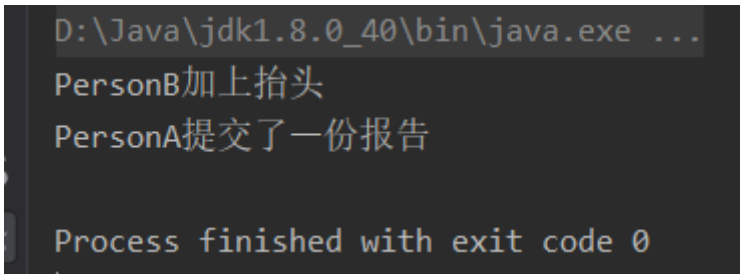
IntelliJ IDEA

JDK 版本: Java 19

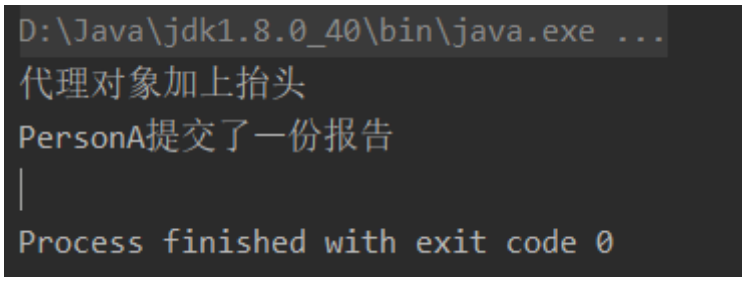
四、实验过程

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。

静态代理测试结果



给PersonA添加动态代理测试结果

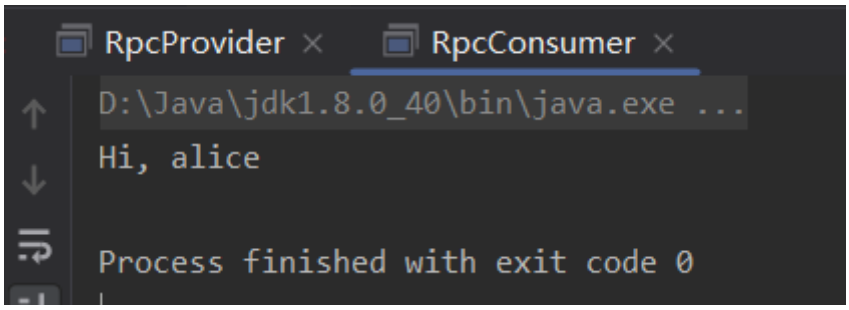


静态代理和动态代理都是代理设计模式的具体应用，用于实现不同的代理方式。静态代理需要程序员手动编写代理类，需要代理的类数量较大则会导致代码冗余并且维护成本增加，而动态代理则是在程序运行时动态创建代理，无需手动编写代理类。

当我们想要为一个接口添加新的功能时可以使用到这种代理设计模式，比如想要给一个数据库操作添加事务，那么就可以创建代理对象在数据库操作的前后开启事务并提交事务，而不用修改原来接口的代码就可以完成这一功能的增强。

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。

RpcProvider和PrcConsumer类的运行结果



自定义的rpc报文格式：

1.rpc报文头部

字段名	字段长度	描述
magic（魔数）	2字节	标识RPC报文的开头
headerLength（头部长度）	2字节	rpc报文头部的长度（在本次自定义报文格式中该字段恒为6）
messageType（消息类型）	1字节	消息类型，例如请求，响应，通知等，映射到具体的方法
serializationType(序列化类型)	1字节	数据序列化类型

2.rpc报文数据

字段名	字段长度	描述
方法名	可变	调用的方法名，指示要调用的是哪个方法
参数类型列表	可变字节	参数类型列表，指示方法的参数类型
参数列表	可变	方法的参数列表，可以是任何序列化格式的二进制数据，如JSON

为了让rpc协议的报文格式更加清晰，专门用RpcHeader类和RpcContent类分别存储报文头部和报文数据

```
import java.io.Serializable;

public class RpcHeader implements Serializable {
```

```
    private short magic;

    private short headerLength;

    private byte messageType;

    private byte serializerType;

    public RpcHeader(){};

    public RpcHeader(short magic, short headerLength, byte messageType,byte
serializerType) {
        this.magic = magic;
        this.headerLength = headerLength;
        this.messageType = messageType;
        this.serializerType = serializerType;
    }

    public short getMagic() {
        return magic;
    }

    public void setMagic(short magic) {
        this.magic = magic;
    }

    public byte getSerializerType() {
        return serializerType;
    }

    public void setSerializerType(byte serializerType) {
        this.serializerType = serializerType;
    }

    public short getHeaderLength() {
        return headerLength;
    }

    public void setHeaderLength(short headerLength) {
        this.headerLength = headerLength;
    }

    public byte getMessageType() {
        return messageType;
    }

    public void setMessageType(byte messageType) {
        this.messageType = messageType;
    }
}
```

```
import java.io.Serializable;

public class RpcContent implements Serializable {
```

```
    private String methodName;

    private Class<?>[] parameterTypes;

    private Object[] args;

    public RpcContent(String methodName, Class<?>[] parameterTypes, Object[]
args) {
        this.methodName = methodName;
        this.parameterTypes = parameterTypes;
        this.args = args;
    }

    public String getMethodName() {
        return methodName;
    }

    public void setMethodName(String methodName) {
        this.methodName = methodName;
    }

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

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

    public Object[] getArgs() {
        return args;
    }

    public void setArgs(Object[] args) {
        this.args = args;
    }
}
```

以及rpc报文类

```
import java.io.Serializable;

public class Rpc implements Serializable {

    private RpcHeader rpcHeader;

    private Object rpcContent;

    public Rpc(RpcHeader rpcHeader, Object rpcContent) {
        this.rpcHeader = rpcHeader;
        this.rpcContent = rpcContent;
    }

    public RpcHeader getRpcHeader() {
```

```
        return rpcHeader;
    }

    public void setRpcHeader(RpcHeader rpcHeader) {
        this.rpcHeader = rpcHeader;
    }

    public Object getRpcContent() {
        return rpcContent;
    }

    public void setRpcContent(Object rpcContent) {
        this.rpcContent = rpcContent;
    }


}
```

rpc类中的rpcContent是object类型而不是RpcContent是因为当消息类型是rpc请求报文时，其content才是rpcContent，但是如果消息类型时rpc响应报文，此时报文的数据主题只是一个object对象。为了让rpc类同时兼容请求报文和响应报文，rpcContent需要设置为obejct类型。

Constants类，用于定义一些常量，如rpc协议的magic，serializeType，rpcRequest对应的一个字节的二进制码以及rpcResponse对应的二进制码。

```
public interface Constants {

    short RPC_MAGIC = 0;

    byte rpcSerializerCode = (byte) 0;

    int headerLength = 6;

    int rpcRequest = 0;

    int rpcResponse = 1;
}
```

RpcProvider类

```
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
                Object rpcObj = is.readObject();
                Rpc rpc = (Rpc) rpcObj;
                RpcHeader rpcHeader = rpc.getRpcHeader();
                if(rpcHeader.getMagic() != Constants.RPC_MAGIC)
```

```
                    return;
                if(rpcHeader.getMessageType() == Constants.rpcRequest &&
rpcHeader.getSerializerType() == Constants.rpcSerializerCode) {
                    RpcContent content = (RpcContent) rpc.getRpcContent();
                    String methodName = content.getMethodName();
                    Class<?>[] parameterTypes = content.getParameterTypes();
                    Object[] arguments = content.getArgs();
// 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(new
Rpc(getHeader(),result));
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static RpcHeader getHeader(){
        short magic = Constants.RPC_MAGIC;
        short headerLength = Constants.headerLength;
        byte messageType = Constants.rpcResponse;
        byte serializeType = Constants.rpcSerializerCode;
        return new RpcHeader(magic,headerLength,messageType,serializeType);
    }
}
```

将接收到的对象转化为rpc后，判断其魔数是否标识本次请求为rpc请求，以及其请求类型和序列化类型，提取出rpc中的方法名，参数类型等请求信息后调用相应方法，最后将返回值封装到rpc中，发送响应报文。

RpcConsumer类

```
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));
```

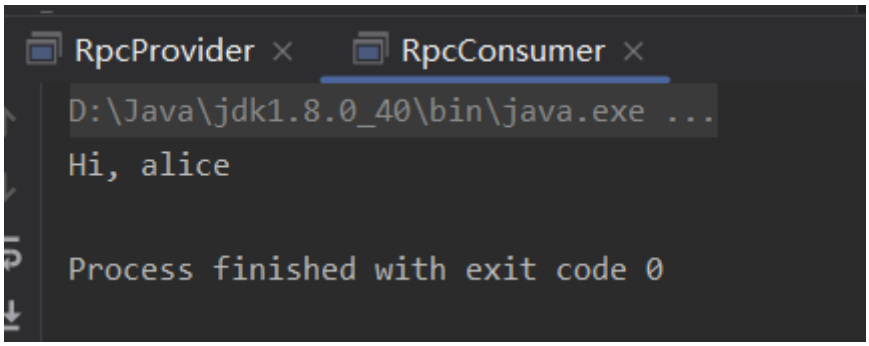
```
        ObjectOutputStream os = new
ObjectOutputStream(socket.getOutputStream());
// rpc提供方和调用方之间协商的报文格式和序列化规则
        RpcHeader header = getHeader();
        RpcContent content = new
RpcContent(method.getName(),method.getParameterTypes(),args);
        os.writeObject(new Rpc(header,content));

        Object result = new
ObjectInputStream(socket.getInputStream()).readObject();
        Rpc rpc = (Rpc) result;
        return rpc.getRpcContent();
    }

    private static RpcHeader getHeader(){
        short magic = Constants.RPC_MAGIC;
        short headerLength = Constants.headerLength;
        byte messageType = Constants.rpcRequest;
        byte serializeType = Constants.rpcSerializerCode;
        return new RpcHeader(magic,headerLength,messageType,serializeType);
    }
}
```

因为rpc报文的请求和响应都是用rpc类来进行的，所以无论发送还是接受都是用rpc类，请求方只要将接收到的rpc报文中数据主题提取出来进行返回即可。

运行结果：



客户端首先指明一些报文头信息，如rpc报文的魔数、本次消息类型为rpc请求、头部长度为6字节等，并将想要远程调用的方法名、参数类型以及对应的参数封装到rpc类中，并发送给服务端。服务端接受到rpc报文，得到rpc报文的头部信息并进行检查，并调用rpc报文的数据主体对应的方法，将返回值再封装到rpc类中发送给客户端，客户端接收到响应报文后进行解析，得到最后的返回值。

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?

1. 自定义报文的RPC协议

自定义报文的RPC协议是一种基于TCP协议的面向过程的远程过程调用协议，使用自定义报文进行通信。支持多种操作系统和编程语言，提供了良好的可扩展性和灵活性。它的优点有：

- 效率高：RPC协议使用二进制数据传输，能够快速且高效地传输数据。
- 进程间通信效果良好：RPC协议是面向过程的通信协议，适用于后端进程之间直接的通信。
- 提供多种序列化方式：RPC协议支持多种数据序列化方式，包括 JSON、XML、二进制等。

2. HTTP 1.0协议

HTTP 1.0协议是一种基于TCP协议的应用层协议，采用请求-响应模式进行通信，使用固定的 HTTP 报文格式进行交互。它的优点有：

- 具有广泛的支持：HTTP 1.0协议被广泛使用，几乎所有的编程语言和操作系统都提供了对 HTTP 协议的支持。
- 简单易用：HTTP 1.0协议使用简单的文本格式的报文进行通信，易于理解和使用。
- 支持缓存：HTTP 1.0协议支持缓存机制，可以将一些静态文件（如图片、CSS、JS等）缓存在浏览器端，提高了性能和性能的可扩展性。

综合来看，自定义报文的RPC协议更适合用于后端进程通信。RPC协议是一种面向过程的通信协议，可以直接操作底层二进制数据，效率高，因此适用于对效率要求较高的后端进程之间的直接通信，而HTTP 1.0协议则更适合用于客户端和服务端之间的通信，因为HTTP 1.0协议可以使用缓存机制，对于静态资源的访问效率比较高。

五、实验总结

RPC是一种常见的网络通信协议，用于在分布式系统之间进行通信。它允许一个计算机程序通过网络请求另一个计算机程序的服务，就像本地调用一样。

使用RPC协议时，客户端不需要知道服务端的具体实现细节和内部结构，只需要知道服务端可以支持哪些程序接口（RPC服务）并提供给客户端一个统一的调用方式，在客户端对程序接口（服务）发起调用时，请求会经过网络发送到远程的服务端，服务端收到请求后，解析客户端的请求，并调用本地的目标函数处理请求，最后将处理结果返回给客户端。

RPC协议的实现通常有以下几个步骤：

定义远程调用接口：在分布式系统中，远程调用的接口是非常关键的，需要事先进行协商和定义。接口的定义需要包括：接口名称、服务端和客户端传递的参数、返回值以及异常处理等。

实现接口的Stub：在客户端和服务端都需要实现接口的Stub。在客户端，Stub 将通过网络传输请求参数，然后将请求发送到远程的服务端。在服务端，Stub 将通过网络处理请求参数，并将请求发送到本地的服务实现函数。

序列化：在客户端和服务器之间的数据传输需要进行序列化操作，将数据编码成可传输的格式，常用的序列化协议有XML、JSON和Google的 Protocol Buffers等。

传输协议：客户端和服务端之间需要使用一种传输协议来发送和接收请求。常用的传输协议包括TCP、UDP和HTTP 等。