# 计算机网络与编程Exp6

## 一、实验目的

- 熟悉远程过程调用
- 熟悉反射和代理

## 二、实验任务

- 编写rpc相关代码并测试
- 尝试使用grpc进行跨语言调用

## 三、使用环境

- **IntelliJ IDEA** Version 2019.3.2
- **JDK** 版本 11.0.6

## 四、实验过程

### 1.1 什么是RPC

- 简单来说是一种通过网络从远程计算机程序上请求服务，像调用本地的函数一样去调远程函数
  - 本地调用:

```
int Add(int l, int r) {
    int res = l + r;
    return res;
}
int lvalue = 1;
int rvalue = 2;
int l_adds_r = Add(lvalue, rvalue);
```

  - 远程调用带来的问题
    1. 函数指针
    2. 参数序列化
    3. 网络传输

### 1.2 反射

- 在Java**运行时环境**中，对于任意一个类，可以知道这个类有哪些属性和方法。
  对于任意一个对象，可以调用它的任意一个方法。
  这种动态获取类的信息以及动态调用对象的方法的功能来自于Java 语言的反射（Reflection）机制。

```
Public Object invoke(Object implicitPara,Object[] explicitPara)
// 调用某类的任意方法 参数1：为实现类 参数2：为方法参数
```

## 1.3 代理

给某一个对象提供一个代理，并由代理对象来控制对真实对象的访问

### 1.3.1 静态代理

- 编写一个接口类

```
package exp6;

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

- 编写小A

```
package exp6;

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

- 编写小B

```
package exp6;

public class PersonB implements IProxy {
    //被代理者的引用
    private IProxy mPerson;

    public PersonB(IProxy person) {
        mPerson = person;
    }

    @Override
    public void submit() {
        before();
        mPerson.submit();
        after();
    }
```

```
    private void before() {
        System.out.println("小B加上报告头");
    }
    private void after() {
        System.out.println("小B加上报告尾");
    }
}
```

- 编写测试类

```
package exp6;

public class TestProxy {
    public static void main(String[] args) {
        // 构造一个小A
        IProxy personA = new PersonA();

        // 构造一个代理小B将小A作为构造参数传递进去
        IProxy personB = new PersonB(personA);

        // 小B提交报告
        personB.submit();
    }
}
```

- 静态代理的缺点
  虽然静态代理实现简单、不侵入原代码,但容易产生过多的代理类且不易维护

#### 1.3.2 动态代理

- 用到了反射机制
- 编写DynamicProxy类

```
package exp6;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class DynamicProxy implements InvocationHandler {
    private Object obj; // 被代理类的引用

    public DynamicProxy(Object obj) {
        this.obj = obj;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
```

```
        before();
        Object result = method.invoke(obj, args); //反射
        after();
        return result;
    }

    private void before() {
        System.out.println("小B加上报告头");
    }
    private void after() {
        System.out.println("小B加上报告尾");
    }
}
```

- 修改测试类测试

```
package exp6;

import java.lang.reflect.Proxy;

public class TestProxy {
    public static void main(String[] args) {
        // 构造一个小A
        IProxy personA = new PersonA();

//        // 构造一个代理小B将小A作为构造参数传递进去
//        IProxy personB = new PersonB(personA);
//
//        // 小B提交报告
//        personB.submit();
        DynamicProxy proxy = new DynamicProxy(personA);

        // 获取被代理类小A的ClassLoader
        ClassLoader loader = personA.getClass().getClassLoader();

        // 动态构造一个代理者小B
        IProxy personB = (IProxy) Proxy.newProxyInstance(loader, new
Class[] { IProxy.class }, proxy);

        personB.submit();
    }
}
```

- $task1$：描述动态代理的基本原理并举例在实际使用中动态代理可以在哪些方面被使用到(如注解、日志等)

## 1.4 RPC

- 编写一个接口类

```
package exp6;

public interface IProxy2 {
    String sayHello(String s);
    String upperString(String s);
}
```

- 编写该接口的实现类

```
package exp6;

public class Proxy2Impl implements IProxy2 {

    @Override
    public String sayHello(String s) {
        return "Hello" + s;
    }

    @Override
    public String upperString(String s) {
        return s.toUpperCase();
    }
}
```

- 编写RPCServer

```
package exp6;

import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;

public class RPCServer {
    public static void main(String[] args) {
        Proxy2Impl service = new Proxy2Impl();

        try (ServerSocket serverSocket = new ServerSocket()){
            serverSocket.bind(new InetSocketAddress(9091));

            try(Socket accept = serverSocket.accept()){
                ObjectInputStream is = new
ObjectInputStream(accept.getInputStream());
                String methodName = is.readUTF();
```

```
                Class<?>[] parameterTypes = (Class<?>[]) is.readObject();
                Object[] arguments = (Object[]) is.readObject();
                Object result =
Proxy2Impl.class.getMethod(methodName,parameterTypes).invoke(service,argume
nts);
                new
ObjectOutputStream(accept.getOutputStream()).writeObject(result);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

- 编写RPCClient

```
package exp6;

import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.net.InetSocketAddress;
import java.net.Socket;

public class RPCClient {
    public static void main(String[] temp) {
        IProxy2 service = (IProxy2)
Proxy.newProxyInstance(IProxy2.class.getClassLoader(), new Class<?>[]
{IProxy2.class}, (Object proxy, Method method, Object[] args) -> {
            try(Socket socket = new Socket()){
                socket.connect(new InetSocketAddress(9091));
                ObjectOutputStream os = new
ObjectOutputStream(socket.getOutputStream());
                os.writeUTF(method.getName());
                os.writeObject(method.getParameterTypes());
                os.writeObject(args);
                return new
ObjectInputStream(socket.getInputStream()).readObject();
            }catch (Exception e){
                return null;
            }
        });
        //System.out.println(service.sayHello("1"));
        System.out.println(service.upperString("a"));
    }
}
```

- $task2$：运行一个server和client将运行的结果截图，并结合代码详细阐述从客户端调用到拿到结果的整个过程是怎么样的(包括客户端和服务端各进行了哪些操作)
- $task3$：修改server，使之使用多线程监管连接，使用多客户端、每个客户端调用多次不同的方法进行测试，将结果截图
- $task4$：对比restful和rpc,简析各自的优缺点

### 1.5 gRPC

- 新建一个maven项目
- 修改pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.example</groupId>
    <artifactId>grpc</artifactId>
    <version>1.0-SNAPSHOT</version>
    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>
    <build>
        <extensions>
            <extension>
                <groupId>kr.motd.maven</groupId>
                <artifactId>os-maven-plugin</artifactId>
                <version>1.6.2</version>
            </extension>
        </extensions>
        <plugins>
            <plugin>
                <groupId>org.xolstice.maven.plugins</groupId>
                <artifactId>protobuf-maven-plugin</artifactId>
                <version>0.6.1</version>
                <configuration>

 <protocArtifact>com.google.protobuf:protoc:3.11.0:exe:${os.detected.classi
fier}</protocArtifact>
                    <pluginId>grpc-java</pluginId>
                    <pluginArtifact>io.grpc:protoc-gen-grpc-
java:1.29.0:exe:${os.detected.classifier}</pluginArtifact>
                    <source>11</source>
                    <target>11</target>
                </configuration>
                <executions>
                    <execution>
```

```
                        <goals>
                            <goal>compile</goal>
                            <goal>compile-custom</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>io.grpc</groupId>
            <artifactId>grpc-netty-shaded</artifactId>
            <version>1.29.0</version>
        </dependency>
        <dependency>
            <groupId>io.grpc</groupId>
            <artifactId>grpc-protobuf</artifactId>
            <version>1.29.0</version>
        </dependency>
        <dependency>
            <groupId>io.grpc</groupId>
            <artifactId>grpc-stub</artifactId>
            <version>1.29.0</version>
        </dependency>
        <dependency> <!-- necessary for Java 9+ -->
            <groupId>org.apache.tomcat</groupId>
            <artifactId>annotations-api</artifactId>
            <version>6.0.53</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

- 编写GrpcServer

```
import io.grpc.Server;
import io.grpc.ServerBuilder;
import io.grpc.stub.StreamObserver;
import java.io.IOException;
import java.util.concurrent.TimeUnit;

public class GrpcServer {

    private Server server;

    private void start() throws IOException {
        int port = 9091;
```

```
        server = ServerBuilder.forPort(port)
                .addService(new NetworkImpl())
                .build()
                .start();

        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                System.err.println("*** shutting down gRPC server since JVM
is shutting down");
                try {
                    GrpcServer.this.stop();
                } catch (InterruptedException e) {
                    e.printStackTrace(System.err);
                }
                System.err.println("*** server shut down");
            }
        });
    }

    private void stop() throws InterruptedException {
        if (server != null) {
            server.shutdown().awaitTermination(30, TimeUnit.SECONDS);
        }
    }

    private void blockUntilShutdown() throws InterruptedException {
        if (server != null) {
            server.awaitTermination();
        }
    }

    public static void main(String[] args) throws IOException,
InterruptedException {
        final GrpcServer server = new GrpcServer();
        server.start();
        server.blockUntilShutdown();
    }

    static class NetworkImpl extends NetworkGrpc.NetworkImplBase {

        @Override
        public void callBack(Request req, StreamObserver<Reply>
responseObserver) {
            Reply reply =
Reply.newBuilder().setMessage(req.getClientName()+" get server
response").build();
            responseObserver.onNext(reply);
            responseObserver.onCompleted();
```

```
        }
    }
}
```

- 编写proto文件(推荐放在src/main/proto文件夹下，方便插件生成对应文件)

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "";
option java_outer_classname = "MyProto";
option objc_class_prefix = "MY";

package testgrpc;

service Network {
  rpc CallBack (Request) returns (Reply) {}
}

message Request {
  string clientName = 1;
}

message Reply {
  string message = 1;
}
```

- $task5$：使用gRPC,利用两种不同语言编写client,client的Request中clientName为"java/python/c++/go client",启动server和client进行测试，将结果截图