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：计算机网络原理与编程　　年级：2018　　上机实践成绩:

指导教师：张召　　姓名：孙秋实　　学号：10185501402

上机实践名称：JavaRPC　　上机实践日期：2020/5/25

上机实践编号：Exp6　　组号：　　上机实践时间：

Part 1
实验目的

- 熟悉远程过程调用
- 熟悉反射和代理

Part 2
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

Part 3
使用环境

- IntelliJ IDEA
- JDK 版本 11.0.6
- PyCharm

Part 4
实验过程

Task 1
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方面被使用到（如注解，日志等）

动态代理的基本原理

动态代理可以通过一个代理类来代理 N 个被代理类，它在更换接口时，不需要重新定义代理类，因为动态代理不需要根据接口提前定义代理类，而是将这种实现推迟到程序运行时由 JVM 来实现。

使用动态代理的几本流程如下所示:

(1) 声明调用处理器类

(2) 声明目标对象类的抽象接口

(3) 声明目标对象类

(4) 通过动态代理对象，调用目标对象的方法

动态代理和静态代理的区别

(1) 静态代理类：由程序员创建或由特定工具自动生成源代码，再对其编译。程序运行前代理类的.class 文件就已经存在了，在代码运行之前，JVM 会读取.class 文件，解析.class 文件内的信息，取出二进制数据，加载进内存中，从而生成对应的 Class 对象。

(2) 动态代理类：程序运行时运用反射机制动态创建而成，在代码运行之前不存在代理类的.class 文件，在代码运行时才动态的生成代理类。

动态代理运用场景

(1) Web 开发中事务提交或回退

(2) 权限管理

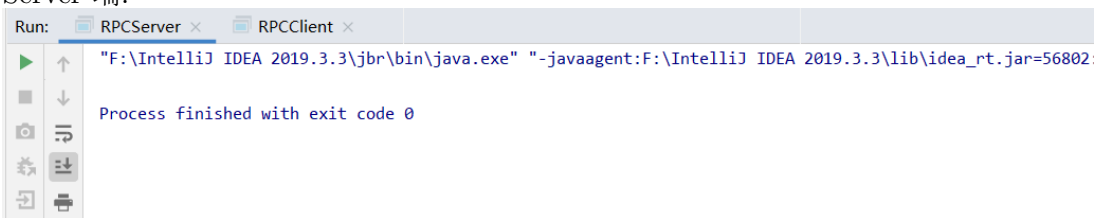
(3) 自定义缓存逻辑处理

(4) 日志写入

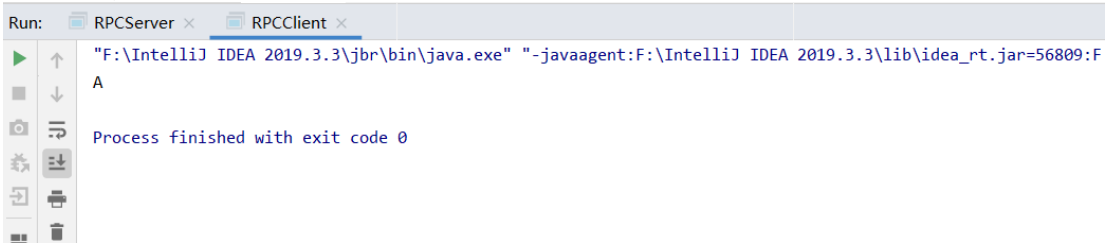
Task 2

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）

Server 端:



Client 端:



以下是客户端调用到拿到结果的整个过程:

首先是 Server 端:

首先要实例化一个接口

```
Proxy2Impl service = new Proxy2Impl(); //实例化
```

然后新建一个 server socket 并且绑定一个地址

```
try (ServerSocket serverSocket = new ServerSocket()){.... }
```

等待客户端连接

```
try(Socket accept = serverSocket.accept()){ //WAITING
```

获得请求的方法名和参数

```
String methodName = is.readUTF(); //GET CLASS NAME AND PARA
```

利用反射机制，服务端通过反射调用客户端请求的方法

```
Object result = Proxy2Impl.class.getMethod(methodName,parameterTypes).invoke(service,arguments); //
    REFLECTION
```

其次再是 Client 端 (详见注释)：

```
public static void main(String[] temp) {
    try {
        Thread.sleep(10000); //这里设置睡眠是为了后面同时有多个CLIENT做准备
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    IProxy2 service = (IProxy2) Proxy.newProxyInstance(IProxy2.class.getClassLoader(), new Class<?>[]
    //定义接口但不用实现这个接口
    //传入CLASSLOADER()
                {IProxy2.class}, (Object proxy, Method method, Object[] args) -> {
            try(Socket socket = new Socket()){
                //动态代理调用方法时，会新建一个SOCKET把调用的方法传过去
                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();
                //等待服务端的返回
            }catch (Exception e){
                return null;
            }
        }
```

```
24      );
25      //以下是需要被打印的信息
26      //System.out.println(service.sayHello("1"));
27      System.out.println(service.upperString("a"));
28  }
```

Task 3

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方法进行测试，将结果截图

在一切开始前为了能清楚的看清多个 client 先后到达，所以在 Client 端加入一个睡眠时间

```
1  public class RPCClient {
2      public static void main(String[] temp) {
3          try {
4              Thread.sleep(10000);
5          } catch (InterruptedException e) {
6              e.printStackTrace();
7          }
8          IProxy2 service = (IProxy2) Proxy.newProxyInstance(IProxy2.class.getClassLoader(), new Class
               <?>[]
9                      {IProxy2.class}, (Object proxy, Method method, Object[] args) -> {
10                 try(Socket socket = new Socket()){
11                     socket.connect(new InetSocketAddress(9091));
12                     ObjectOutputStream os = new ObjectOutputStream(socket.getOutputStream());
13                     os.writeUTF(method.getName());
14                     os.writeObject(method.getParameterTypes());
15                     os.writeObject(args);
16                     return new ObjectInputStream(socket.getInputStream()).readObject();
17                 }catch (Exception e){
18                     return null;
19                 }
20             }
21             );
22         final double d = Math.random();
23         final int i = (int)(d*100);
24         if(i%2==0){
25             System.out.println(service.sayHello("1"));
26         }else{
27             System.out.println(service.upperString("a"));}
28     }
29 }
```

我在选择调用的方法时增加了一个随机数，随机选择调用 sayHello 还是 upperString

接下来是改写 Server 类可以使用多客户端进行测试

首先开启多线程

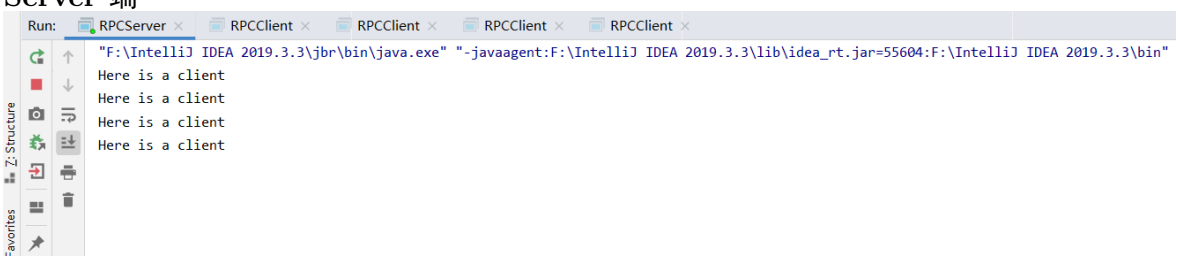
```
public static class ClientHandler extends Thread{
    private Socket accept;
    Proxy2Impl service;
    ClientHandler(Socket accept, Proxy2Impl service){
        this.accept=accept;
        this.service=service;
    }
    public void run(){
        super.run();
        try{
            ObjectInputStream 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,
                arguments);
            new ObjectOutputStream(accept.getOutputStream()).writeObject(result);
            System.out.println("Here is a client");
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

然后对新加入的 Client 进行监听

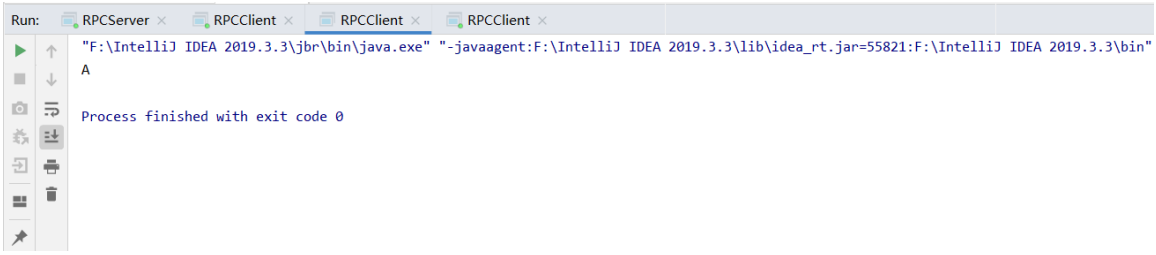
```
public static void main(String[] args) {
    Proxy2Impl my_service = new Proxy2Impl();
    try (ServerSocket serverSocket = new ServerSocket()){
        serverSocket.bind(new InetSocketAddress(9091));
        while (true){
            Socket my_accept=serverSocket.accept();
            ClientHandler clientHandler =new ClientHandler(my_accept,my_service);
            clientHandler.start();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

以下是测试结果

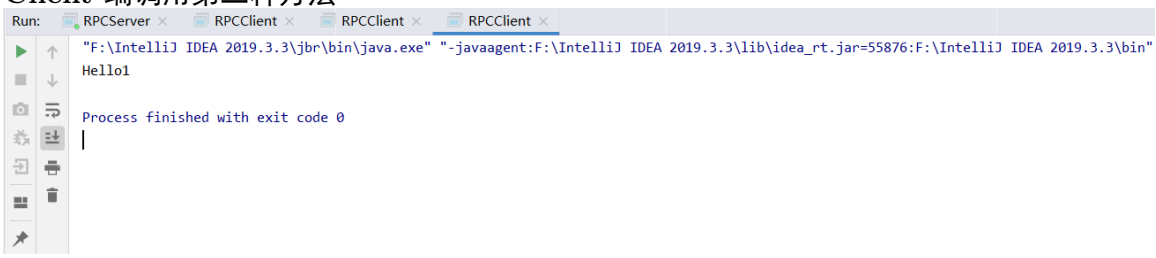
Server 端



Client 端调用第一种方法



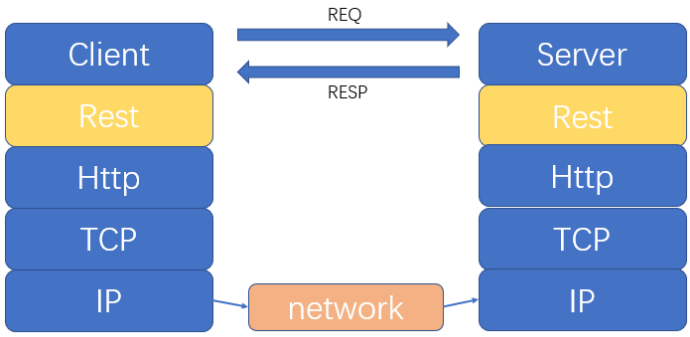
Client 端调用第二种方法



Task 4

对比 Restful 和 RPC，简析各自的优缺点

REST(Representational State Transfer)，是基于 HTTP 协议之上的一组约束和属性，即“表现层状态转移”。



REST 是一种设计风格，用于描述的是在网络中 Client 和 Server 的一种交互形式，目的是便于不同的软件或程序在网络中互相传递消息，RESTful 即实现 REST 设计风格的一种架构。

RESTful API 的优缺点：

(1) **优点**

(a) 可以直接通过 http 使用，不需要额外的协议

(b) 行为和资源分离

(2) **缺点**

(a) 公开的内部数据结构

(b) 优化难度大

RPC API 的优缺点：

(1) **优点**

(a) 简单且易于理解

(b) 轻量级数据载体

(c) 扩展性强

(2) **缺点**

(a) 框架开发难度大

(b) 异常处理困难

Task 5

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

首先是在 IntelliJ IDEA 和 PyCharm 中安装插件 (需要设置 JDK 版本为 11)，插件安装路径上的所有文件路径都**不能是中文**，否则会报错

把以下文件放入 PyCharm 的同路径下并安装

```
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;
}
```

然后安装需要的库

```
1    pip3 install grpcio-tools
2    pip3 install protobuf
3    pip3 install grpcio
```

接着生成适用于 python 的代码

```
python -m grpc_tools.protoc -I ./proto --python_out=. --grpc_python_out=. ./proto/
network.proto
```

以下是 Client(python) 端实现具体代码

```python
import grpc
import network_pb2_grpc
import network_pb2
channel = grpc.insecure_channel("localhost:9091")
my_stub = network_pb2_grpc.NetworkStub(channel)
reply_message= network_pb2.Reply = my_stub.CallBack(network_pb2.Request(clientName ="I'M HERE!!
    IMPLEMENTED BY PYTHON"))
print("REP MESSAGE(PYTHON): {0}".format(reply_message.message))
```

运行结果如下图所示

其次是 Java 实现，原理类似

```java
public class GrpcClient {

    private final ManagedChannel my_CHANNEL;
    private final NetworkGrpc.NetworkBlockingStub my_STUB;

    public static void main(String[] args) {
        try {
            GrpcClient client = new GrpcClient("localhost",9091);
            client.Greeting("I'M HERE! IMPLEMENTED BY JAVA");
            client.shutdown();
        } catch (Exception e) {
            System.out.println("此处出错！！"+e);
        }
    }

    public void shutdown() throws InterruptedException {
        my_CHANNEL.shutdown().awaitTermination(10, TimeUnit.SECONDS);//这里.SECONDS是时间粒度
    }

    public GrpcClient(String host,int port){
        my_CHANNEL = ManagedChannelBuilder.forAddress(host,port)
                    .usePlaintext()
                    .build();
        my_STUB = NetworkGrpc.newBlockingStub(my_CHANNEL);
```

```
25      }
26
27      public void Greeting(String name){
28          Request request= Request.newBuilder().setClientName(name).build();
29          Reply response = my_STUB.callBack(request);
30          System.out.print(response.getMessage());
31      }
32  }
```

运行结果如下图所示

Part 5
实验总结

本次实验稍微有些难度，尤其是 task3 和 task5。第一次尝试跨语言调用，难处在开启多线程的部分逻辑有些不清，花费了一些时间，其次是 task5 的环境配置上遇到了一些小问题（好在在助教和搜索引擎的帮助下解决了）。通过本次实验巩固了一些有关 Socket 编程的技能，并且了解了跨语言调用的具体过程和两种不同 API 的优缺点，还对 RPC 的运作方式有了深入的了解。