序列化/反序列化

2018年8月20日 星期一　　上午 9:02

1. 序列化/反序列化概念

java是面向对象的语言,万物皆对象,对象平常是存活在内存中的,动态变化的数据.但是有的时候,我们需要将内存中存活的动态变化的对象数据转化成可以固定下来的字节信息,以便于保存或传输.而序列化技术就可以实现这个效果,而将已经固定下来的对象的字节信息再转换回内存对象的过程就称之为反序列化的过程.

2. 序列化/反序列化应用场景

a. 将对象序列化后[持久化]保存

将内存中的对象信息序列化后保存到外部存储设备中.这个过程也称之为将对象持久化保存了.之后再需要的时候可以将持久化保存的对象再读取回程序内进行反序列化再恢复为对象,这个过程也称之为反持久化.

b. 将对象序列化化后通过网络传输实现[RPC]

将内存中的对象信息序列化后,将字节信息通过网络发送给其他程序,其他程序收到这些数据后,进行反序列化就可以恢复出对象,得到对象中的信息.

3. 常见的序列化反序列技术

序列化 反序列化 是在分布式开发中 包括 [持久化] [RPC] 操作的基础,是在分布式环境下非常常见的操作 所以序列化反序列时性能的好坏 直接影响分布式程序的性能 所以非常的重要 在分布式程序中 选择一个良好的 序列化反序列化技术 往往对性能有非常直接影响

a. java自带的序列化反序列化技术

java.io

类 ObjectInputStream

java.io

类 ObjectOutputStream

java.io

接口 Serializable

缺点:

效率低下 - 浪费时间

产生的结果数据大 - 浪费空间

只能再java中使用 - 无法跨语言

b. 其他开源序列化反序列化技术 - AVRO

由APACHE开源组织提供的开源序列化反序列化技术

性能优良 体积较小 可以跨语言

在大数据技术中常见

c. 其他开源序列化反序列化技术 - GoogleProtobuffer

由Google提供的开源序列化反序列化技术

性能优良 体积较小 可以跨语言

在业界广泛使用

d. 其他开源序列化反序列化技术 - Thrift

由facebook提供的开源序列化反序列化技术

性能优良 体积较小 可以跨语言

在特定领域用的较多

JAVA自带的序列化反序列化机制

2018年8月20日 星期一　　上午 11:12

1. 关键的类

java.io

类 ObjectInputStream

java.io

类 ObjectOutputStream

java.io

接口 Serializable

2. 优缺点

优点:

java原生的序列化反序列化机制,不需要导入任何第三方包就可以使用

缺点:

速度慢

体积大

不能跨平台

3. 案例

```
1  package cn.tedu.javaseria;
2
3
4  import java.io.Serializable;
5
6
7  public class Person implements Serializable {
8          private String name;
9          private int age;
10         private transient String psw;
11
12
13         public Person() {
14         }
15
16         public Person(String name, int age,String psw) {
17                 this.name = name;
18                 this.age = age;
19                 this.psw = psw;
20         }
21         public String getName() {
22                 return name;
23         }
24         public void setName(String name) {
25                 this.name = name;
26         }
27         public int getAge() {
28                 return age;
29         }
30         public void setAge(int age) {
31                 this.age = age;
32         }
33         public String getPsw() {
34                 return psw;
35         }
36         public void setPsw(String psw) {
37                 this.psw = psw;
         }


    }
```

```java
package cn.tedu.javaseria;


import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;


public class OOSDemo01 {
        public static void main(String[] args) throws Exception {
                //1.创建对象
                Person p = new Person("zs", 19,"123321");
                //2.创建序列化的流
                OutputStream out = new FileOutputStream("p.data");
                ObjectOutputStream oos = new ObjectOutputStream(out);
                //3.序列化对象 输出到文件 实现持久化
                oos.writeObject(p);
                oos.flush();
                oos.close();
        }
}
```

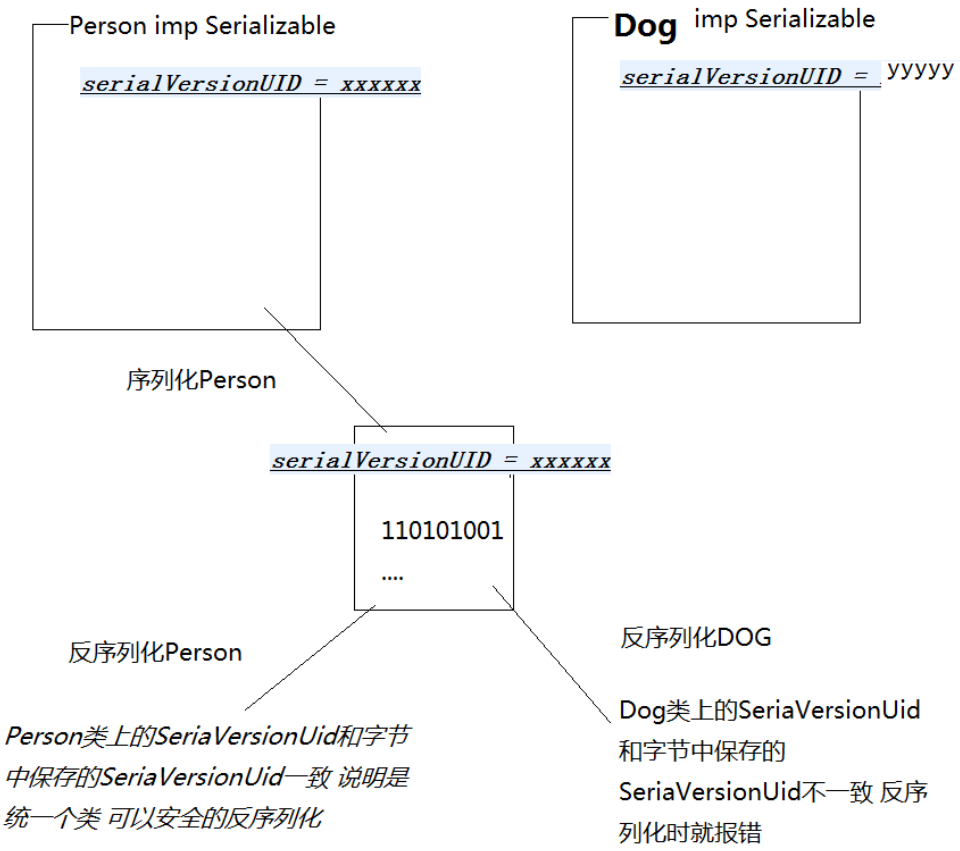
```java
package cn.tedu.javaseria;


import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;


public class OISDemo01 {
      public static void main(String[] args) throws Exception {
            //1.创建反序列化流
            InputStream in = new FileInputStream("p.data");
            ObjectInputStream ois = new ObjectInputStream(in);
            //2.进行反序列化操作
            Person p = (Person) ois.readObject();
            ois.close();

            //3.获取属性
            System.out.println(p.getName());
            System.out.println(p.getAge());
            System.out.println(p.getPsw());
      }
}
```

4. 详解

a. 想要被java序列化机制操作的对象的类必须实现Serializable接口

b. 序列化和反序列化 操作的对象并不是同一个 而是相同类型的不同对象具有相同的信息

c. 在java序列化反序列化过程中,实现了Serializable接口的类,需要指定 静态常量的serialVersionUID属性,如果不指定虚拟机在编译过程中会自动生成一个,来指定当前类的序列化版本编号,在序列化时,在产生的字节信息中会保存这个编号,而在反序列化时也会去检查这个编号,如果不一致则认为序列化和反序列化时不是同一个类 抛出异常 保证了 序列化反序列化时类型的一致安全



d. 在通过网络将序列化的对象字节信息发送给另一台机器进行反序列化时,虽然两端是同样的类但是并不是同一个类,默认生成的SeriaVersionUid很可能不同,会造成反序列化失败,此时需要在两头都手动指定SeriaVersionUid为相同的值,才可以保证正常反序列化

e. 可以在实现了Serializable接口的类的属性中,用transient关键字声明指定属性不需要被序列化,这样在序列化该类的对象时,该属性会自动被忽略,这在保护对象中一些隐秘信息时非常的有用

GoolgeProtoBuffer序列化反序列化

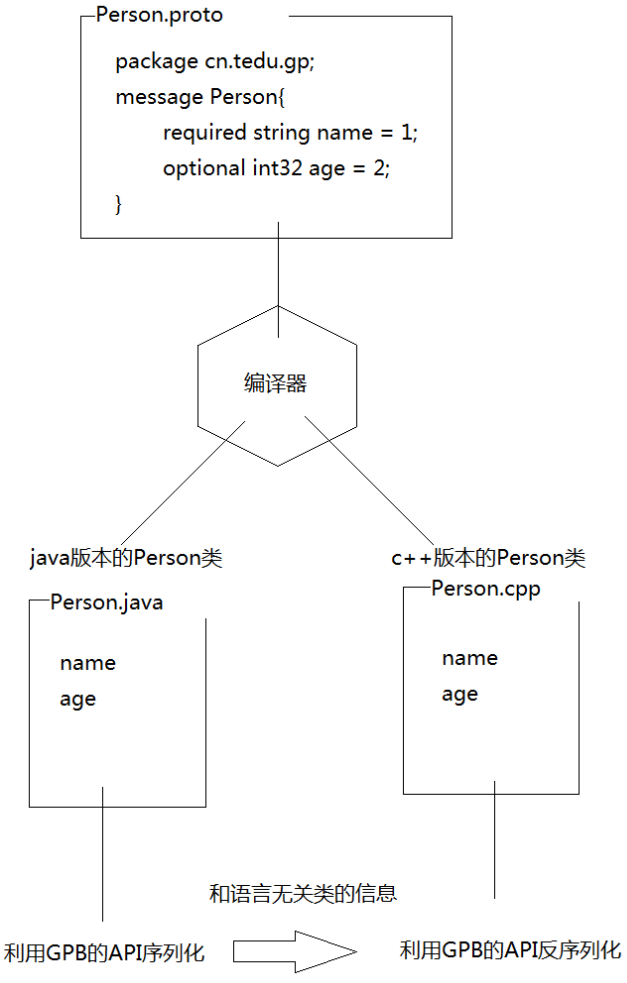
2018年8月20日 星期一　　下午 2:04

1. GoogleProtoBuffer概述

谷歌提供的第三方序列化框架,以效率高,体积小,支持多种语言而著称,在许多技术汇总都可以见到它的身影.

使用GPB的步骤:

a. 编写proto文件,利用GPB自定义的语法来声明要进行序列化的类的结构
b. 通过GPB提供的编译器将proto文件编译为需要的语言的类文件
c. 将生成的类导入项目中,利用GPB提供的API实现序列化反序列化



2. GoogleProtoBuffer详解 - 编写.proto文件

GPB中定义了和具体语言无关的独立的语法来编写.proto文件

```
message SearchRequest {
  required string query = 1;
```

```
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}
```

数据类型:

声明数据对应的类型要求

.proto类型	Java 类型	C++类型	备注
double	double	double	
float	float	float	
int32	int	int32	使用可变长编码方式。编码负数时不够高效——如果你的字段可能含有负数，那么请使用sint32。
int64	long	int64	使用可变长编码方式。编码负数时不够高效——如果你的字段可能含有负数，那么请使用sint64。
uint32	int[1]	uint32	Uses variable-length encoding.
uint64	long[1]	uint64	Uses variable-length encoding.
sint32	int	int32	使用可变长编码方式。有符号的整型值。编码时比通常的int32高效。
sint64	long	int64	使用可变长编码方式。有符号的整型值。编码时比通常的int64高效。
fixed32	int[1]	uint32	总是4个字节。如果数值总是比总是比2^{28}大的话，这个类型会比uint32高效。
fixed64	long[1]	uint64	总是8个字节。如果数值总是比总是比2^{56}大的话，这个类型会比uint64高效。
sfixed32	int	int32	总是4个字节。
sfixed64	long	int64	总是8个字节。
bool	boolean	bool	
string	String	string	一个字符串必须是UTF-8编码或者7-bit ASCII编码的文本。
bytes	ByteString	string	可能包含任意顺序的字节数据。

标识号:

消息中的每个属性都应该有一个唯一的标识号,用来在二进制消息中唯一的代表该属性,且一经使用不可再改.[1-15]占1个字节,[16,2047]占两个字节,所以经常用的字段优先使用[1-15]

字段规则:

声明字段在类中存在的规则

required：一个格式良好的消息一定要含有1个这种字段。表示该值是必须要设置的

optional：消息格式中该字段可以有0个或1个值（不超过1个）。

repeated：在一个格式良好的消息中，这种字段可以重复任意多次（包括0次）。重复的值的顺序会被保留。表示该值可以重复，相当于java中的List。

案例:

```
package cn.tedu.gp;


option java_package  = "cn.tedu.gp";
option java_outer_classname = "GP_Person";


message Person{
        required string name = 1;
        optional int32 age = 2;
}
```

3. GoogleProtoBuffer详解 - 将.proto文件编译为对应语言的类

可以使用Google提供的编译器[protoc.exe],通过不同的参数将.proto文件编译成不同语言的类

protoc --proto_path=IMPORT_PATH --cpp_out=DST_DIR --java_out=DST_DIR --python_out=DST_DIR path/to/file.proto

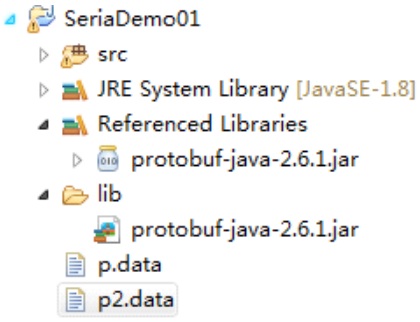
案例:

```
protoc.exe --java_out=./ ./Person.proto
```

4. GoogleProtoBuffer详解 - 将编译好的.java导入项目

导入后会报错,还需要额外导入GPB相关的jar包[protobuf-java-2.6.1.jar]

案例:



5. GoogleProtoBuffer详解 - 利用GPB提供的API来实线序列化反序列化

使用Builder创建对象:

案例:

```
GP_Person.Person p = GP_Person.Person.newBuilder()
        .setName("zs")
        .setAge(19)
        .build();
```

将对象序列化:

- byte[] toBytesArray(): 生成字节数组
- void writeTo(OutputStresam output) : 序列化并写入到指定的输出流中

案例:

```java
package cn.tedu.gp.test;


import java.io.FileOutputStream;
import java.io.OutputStream;


import cn.tedu.gp.GP_Person;

public class GPTestDemo01 {
        public static void main(String[] args) throws Exception {
                //1.创建对象
                GP_Person.Person p = GP_Person.Person.newBuilder()
                        .setName("zs")
                        .setAge(19)
                        .build();

                //2.序列化对象
                OutputStream out = new FileOutputStream("p2.data");
                p.writeTo(out);
                out.flush();
                out.close();
        }
}
```

将字节反序列化:

- static Person parseFrom(byte[] data)：解析二进制数组，反序列化指定对象
- static Person parseFrom(InputStream input)：解析输入流，反序列化指出指定对象

案例:

```java
package cn.tedu.gp.test;


import java.io.FileInputStream;
import java.io.InputStream;


import cn.tedu.gp.GP_Person;

public class GPTestDemo02 {
      public static void main(String[] args) throws Exception {
            //1.通过流读取序列化的数据
            InputStream in = new FileInputStream("p2.data");
            //2.进行反序列化
            GP_Person.Person p = GP_Person.Person.parseFrom(in);
            //3.获取对象信息
            System.out.println(p.getName());
            System.out.println(p.getAge());
      }
}
```

利用GoogleProtoBuffer实现RPC

2018年8月20日 星期一　　下午 3:22

1. RPC概述

RPC即java的远程方法调用,是java在分布式环境下,分布式程序的各个部分之间进行互相调用的机制.

传统的方法调用都是在一台JVM虚拟机中运行,直接在内存中完成方法的调用,但是在分布式的环境下,方法的调用可能需要跨JVM虚拟机 跨进程 跨网络来进行,此时无法直接在内存中完成,需要通过序列化反序列化机制 网络通信机制 动态代理机制 等等的机制来实这种远程的方法的调用,这个过程称之为java的远程方法调用,即RPC.

RPC并不是一项独立的技术,而是基于若干种技术,针对分布式环境下远程的方法调用提出的一种解决方案.

2. GoogleProtoBuffer中的RPC

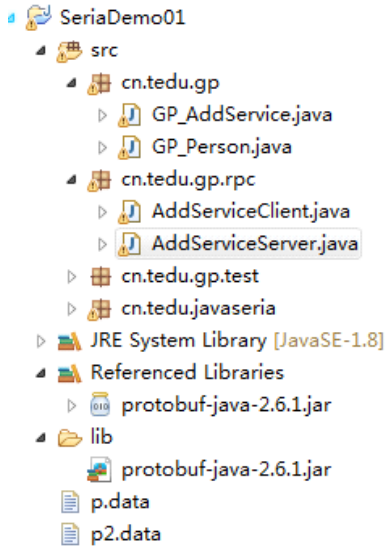
GoogleProtoBuffer中封装了RPC的编程接口,使我们可以非常方便的实现RPC的功能.

需要在.proto文件中编写代码声明RPC远程方法

代码:

```
1  package cn.tedu.gp;
2
3
4  option java_package  = "cn.tedu.gp.rpc";
5  option java_outer_classname = "GPAddService";
6  option java_generic_services = true;
7
8
9  message Req{
10         required int32 num1 = 1;
11         required int32 num2 = 2;
12 }
13
14
15 message Resp{
16         required int32 rnum = 1;
17 }
18
   service AddService{
           rpc addNum (Req) returns (Resp);
   }
```

经过编译将生成的类 导入到程序中



编写客户端代码,创建本地存根,通过存根调用远程方法,在存根内部传入BlockingRpcChannel,在其中编写通过网络调用远程方法的代码

```
1  package cn.tedu.gp.rpc;
2
3
4  import java.io.IOException;
5  import java.io.InputStream;
6  import java.io.OutputStream;
7  import java.net.InetSocketAddress;
8  import java.net.Socket;
9  import java.net.UnknownHostException;
10
11
12 import com.google.protobuf.BlockingRpcChannel;
13 import com.google.protobuf.Descriptors.MethodDescriptor;
14 import com.google.protobuf.Message;
15 import com.google.protobuf.RpcController;
16 import com.google.protobuf.ServiceException;
17
18
19 import cn.tedu.gp.GP_AddService.AddService;
20 import cn.tedu.gp.GP_AddService.AddService.BlockingInterface;
21 import cn.tedu.gp.GP_AddService.Request;
22 import cn.tedu.gp.GP_AddService.Response;
23
24 class MyBlockingRpcChannel implements BlockingRpcChannel{
25
26
27         //真正的实现通过网络远程调用的过程
28         @Override
29         public Message callBlockingMethod(MethodDescriptor md, RpcController controlle,
Message req, Message resp)throws ServiceException {
30                 try {
31                         //--连接远程服务器
32                         Socket socket = new Socket();
33                         socket.connect(new InetSocketAddress("127.0.0.1", 44444));
34                         //--将req对象序列化传给服务器
35                         OutputStream out = socket.getOutputStream();
36                         req.writeTo(out);
37                         out.flush();
38                         socket.shutdownOutput();
39                         //--获取服务器返回的二进制
40                         InputStream in = socket.getInputStream();
41                         //--将二进制反序列化成resp对象
42                         Response respx = Response.parseFrom(in);
43                         //--将结果返回
44                         return respx;
45                 } catch (Exception e) {
46                         e.printStackTrace();
47                         throw new RuntimeException(e);
48                 }
49         }
50
51 }
```

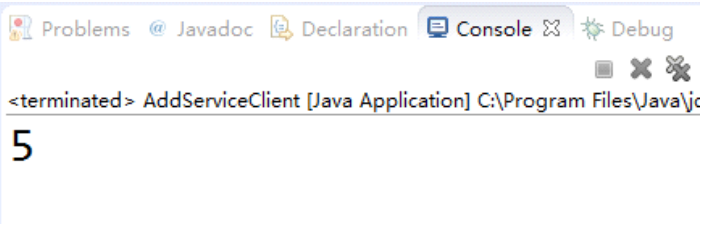
```
52
53  public class AddServiceClient {
54          public static void main(String[] args) throws Exception {
55                  //1.创建本地存根 存根具有所有可供调用的远程方法
56                  //--在创建存根时 传入了一个BlockingRpcChannel的实现类的对象
57                  //--从此 调用此存根上的远程方法时
58                  //--这些方法都会间接的取调用BlockingRpcChannel中
59                  //--的 callBlockingMethod方法来真正实现远程方法调用
60                  BlockingInterface stub = AddService.newBlockingStub(new
61  MyBlockingRpcChannel());
62                  //2.调用存根对象上的远程方法 就像在调用本地方法一样
63                  Request req = Request.newBuilder().setNum1(5).setNum2(8).build();
64                  Response resp = stub.add(null, req);
                    //3.得到结果打印
                    int result = resp.getResult();
                    System.out.println(result);
            }
    }
```

编写服务端代码,接收客户端连接,反序列化得到请求参数,调用真正的方法得到结果对象,序列化变为字节,通过网络发回给客户端

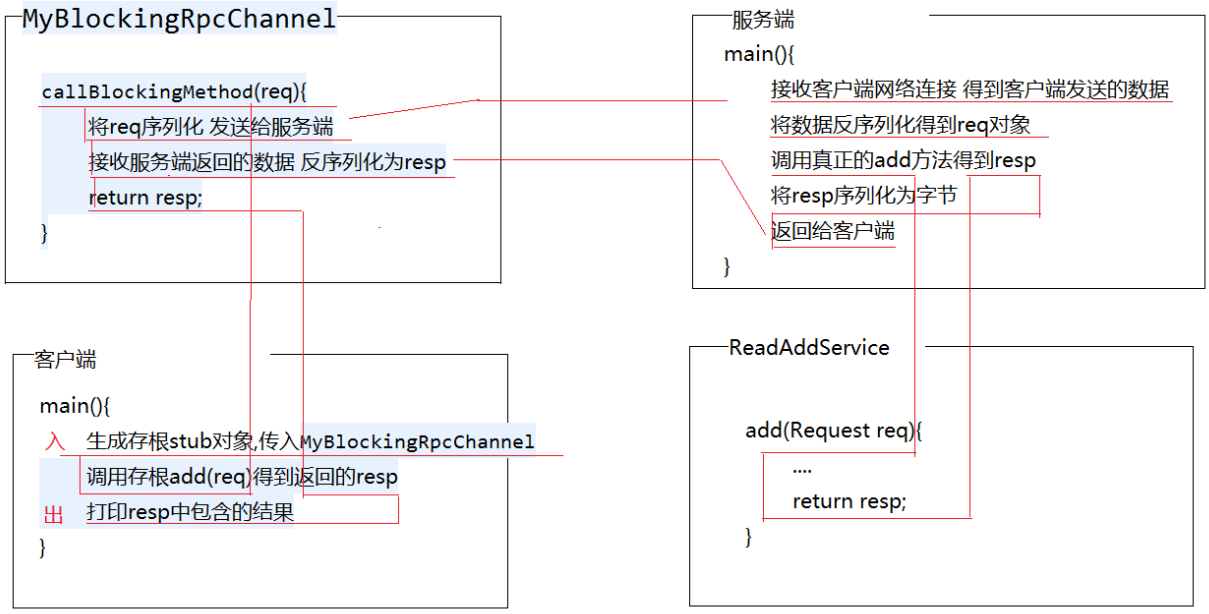
```
1   package cn.tedu.gp.rpc;
2
3
4   import java.io.IOException;
5   import java.io.InputStream;
6   import java.io.OutputStream;
7   import java.net.ServerSocket;
8   import java.net.Socket;
9
10  import com.google.protobuf.RpcController;
11  import com.google.protobuf.ServiceException;
12
13
14  import cn.tedu.gp.GP_AddService.AddService.BlockingInterface;
15  import cn.tedu.gp.GP_AddService.Request;
16  import cn.tedu.gp.GP_AddService.Response;
17
18
19  public class AddServiceServer {
20        private static RealAddService realAddService = new RealAddService();
21        public static void main(String[] args) throws Exception {
22              //1.等待远程连接
23              ServerSocket ss = new ServerSocket(44444);
24              Socket socket = ss.accept();
25              InputStream in = socket.getInputStream();
26              //2.从输入中反序列化为req对象
27              Request req = Request.parseFrom(in);
28              //3.真正调用add方法 得到resp对象
29              Response resp = realAddService.add(null, req);
30              //4.将resp序列化 通过网络发回给客户端
31              OutputStream out = socket.getOutputStream();
32              resp.writeTo(out);
33              socket.shutdownOutput();
34              //5.关闭网络
35              socket.close();
36              ss.close();
37        }
38
39  }
40
41  class RealAddService implements BlockingInterface{
42
43
44        @Override
45        public Response add(RpcController controller, Request req) throws ServiceException
46  {
47              int n1 = req.getNum1();
48              int n2 = req.getNum2();
49              int result = n1 + n2;
                Response resp = Response.newBuilder().setResult(result).build();
                return resp;
        }
```

```
        }
```

经过测试可以实现远程方法调用



整个执行过程:



3. RPC技术的重要性

rpc是通过序列化技术 网络通信技术 来实线的远程方法调用的技术方案, 可以非常方便的实现分布式环境下方法调用, 是分布式程序开发的基础, 是后续学习的大数据框架底层实现技术, 虽然以后很少回去写RPC的代码, 需要大家有所了解.

总结

2018年8月20日 星期一　　下午 4:58

掌握序列化反序列化的概念及应用场景

常见的序列化反序列化技术 及 为什么不用java的序列化反序列化

掌握java的序列化反序列化

了解GoogleProtoBuffer工作原理 不需要掌握具体使用

掌握RPC的概念和原理

了解基于GoogleProtoBuffer实现RPC的过程 不需要掌握