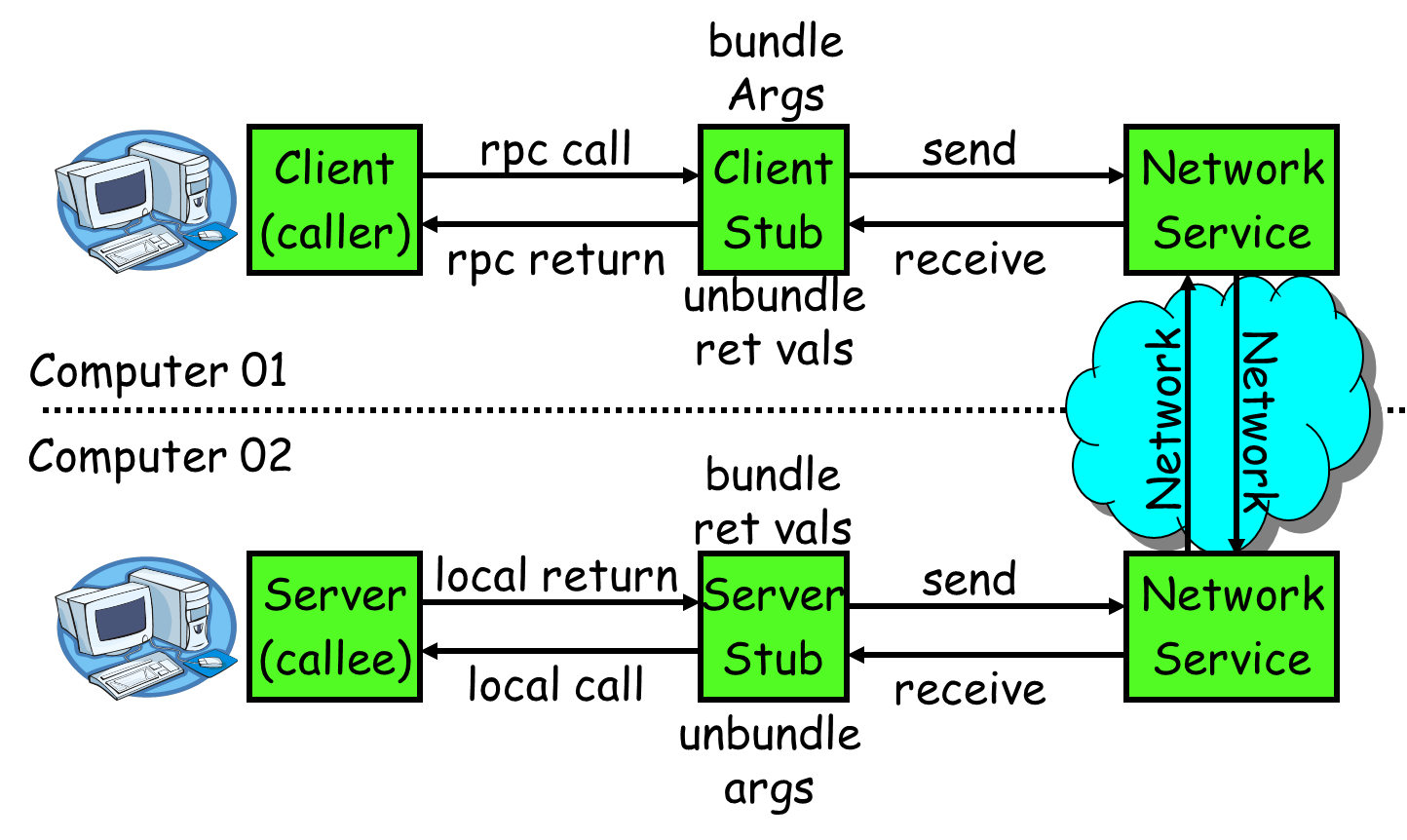
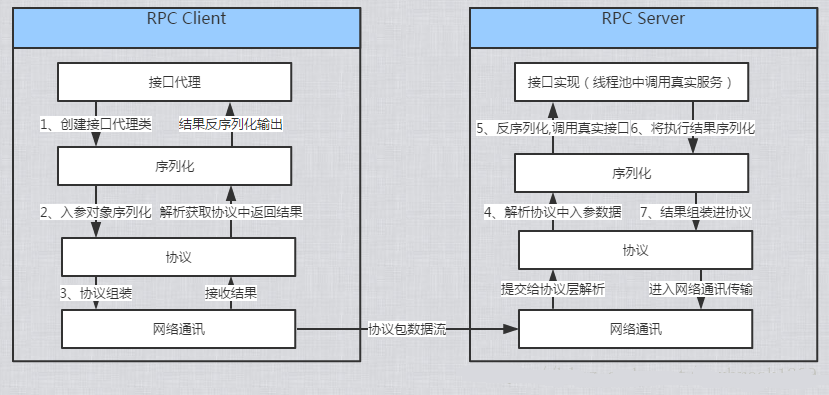
RPC(远程过程调用)框架

RPC(Remote Procedure Call)——远程过程调用，其通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。在分布式系统中，组件部署在不同的主机上，通过Rpc可以使得程序能够像访问本地系统资源一样，通过网络传输去访问远端系统资源。目前在各大公司被广泛使用，如阿里的dubbo，Facebooke的Thrift、Google的gRpc及Twitter的finagle等。

由于服务部署在不同的主机，服务间的调用需要涉及到网络通信，服务消费方调用服务需要处理网络通信相关的代码，复杂而且易出错。远程过程调用可以帮助调用者简化网络通信，可以像调用本地服务一样调用远程服务，要让网路通信对使用者透明，需要对通信细节进行封装，Rpc的调用流程如下图所示：



通信细节的封装在Client/Server Stub中完成，其核心是如何将调用相关的方法、参数等封装成能够进行网络传输的消息体并在Server Stub中进行解码。Rpc通信的核心流程如下：



Rpc的流程重客户端开始，通过动态代理创建Server目标接口代理，将调用函数及参数对象序列化后进行协议封装，协议包包括以下数据

* 序列化方式
* 协议版本
* 协议类型
* 方法签名等

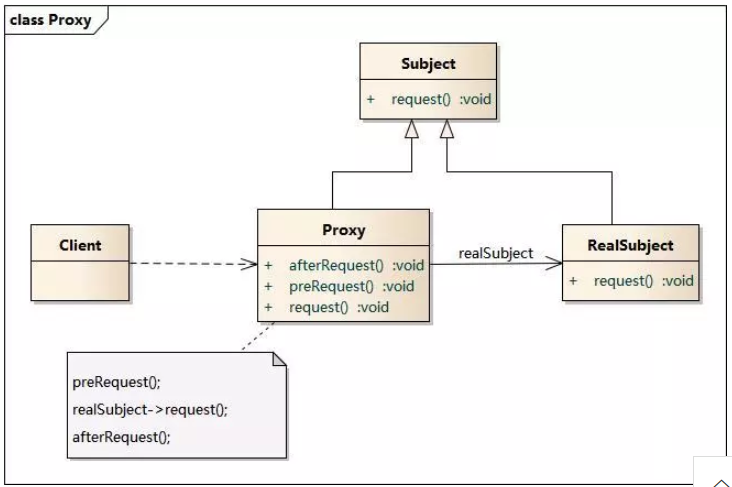
这些信息组成一个协议包，并交付给传输层进行发送。

Server通过网络通信层接收到协议包，将协议包解析，从中取出接口签名、入参对象字节流信息。对字节流信息进行反序列化操作，找出接口实现类，invoke执行的目标方法后进行执行。最后将结构以同样的序列化方法进行序列化操作，其同样包括序列化方式、版本、协议类型、方法签名等信息，发送给Client。

从Rpc架构和流程可以总结出，Rpc核心技术包括：动态代理、协议、序列化、通信技术、注册中心等技术，下面进行详细介绍

# 1. 动态代理

代理是一种设计模式，提供了对目标对象的间接访问方式，在某些情况下对象不适合或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介作用，其类图如下所示：

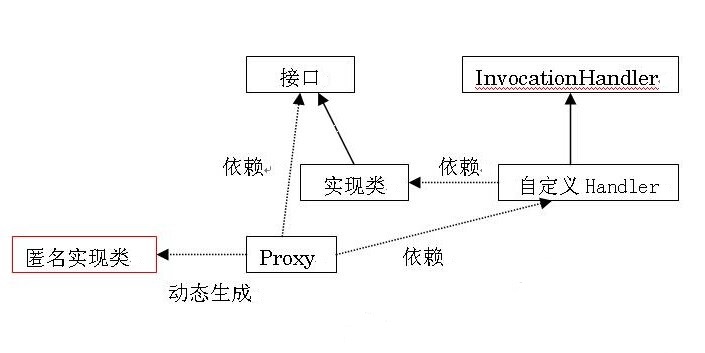


通过代理模式，可以在目标实现的基础上增加额外的功能操作，如前拦截/后拦截等，以满足自身的业务需求，同时使用代理模式可以扩展目标对象的功能。在代理模式中，有以下角色：

* 抽象角色(Subject)，通过接口或者抽象类声明真实角色实现的业务方法
* 真实角色(RealSubject)，实现抽象角色，真实角色所要实现的业务逻辑，供代理角色调用
* 代理角色(Proxy)，实现抽象角色，是真实角色的代理，通过真实角色的业务逻辑方法来实现抽象方法，并可以附件自己的操作

Java代理有两种实现方式，静态代理和动态代理。静态代理比较简单，不再详述，动态代理相对于静态代理不需要实现与目标类相同的接口，而且不需要依赖接口的具体实现，运行时通过创建虚拟的代理类，在代理的目标方法实际执行时，通过Java反射技术获取该方法对象，并且在执行前后添加需要的操作。

在Java中动态代理依赖反射机制，两种重要的类为InvocationHandler和Proxy。每个动态代理类必须实现InvocationHandler接口，而且代理类的实例都关联到一个Handler，通过代理对象调用一个方法的时候，这个方法的调用会被转发为由InvocationHandler这个接口的invoke方法调用，调用关系如下图所示：



下面是一个示例，如下：

1. **定义接口和实现类**

接口Subject，方法rent和hello

*public interface Subject {*

*public void hello(String str);*

*}*

定义实现类：

*public class RealSubject implements Subject {*

*@Override*

*public void hello(String str) {*

*System.out.println("Hello: " + str);*

*}*

*}*

1. **实现动态代理类，DynamicProxy**

*public class DynamicProxy implements InvocationHandler {*

*private Object subject;*

*public DynamicProxy(Object subject ) {*

*this.subject = subject;*

*}*

*@Override*

*public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {*

*System.out.println("Before Invoke ,Method:" + method);*

*method.invoke(subject, args);*

*System.out.println("After Invoke");*

*return null;*

*}*

*}*

**3) 测试程序**

*public class Client {*

*public static void main(String[] args) {*

*Subject realSubject = new RealSubject();*

*InvocationHandler handler = new DynamicProxy(realSubject);*

*Subject subject = (Subject) Proxy.newProxyInstance(*

*handler.getClass().getClassLoader(),*

*realSubject.getClass().getInterfaces(),*

*handler);*

*System.out.println(subject.getClass().getName());*

*subject.hello("world");*

*}*

*}*

输出如下：

*com.sun.proxy.$Proxy0*

*Method:public abstract void com.fys.proxy.Subject.hello(java.lang.String)*

*Hello: world*

1. **Proxy源码解析**

动态代理的核心Proxy类，创建动态代理类及其实例的静态方法

*(Subject) Proxy.newProxyInstance(*

*handler.getClass().getClassLoader(), //动态代理类加载器*

*realSubject.getClass().getInterfaces(),*

*//interface,指定动态代理类所要实现的所有接口*

*handler); //指定与代理类相关联的InvocationHandler*

newProxyInstance是生成新的代理类，该代理类是继承getInterfaces中的接口，如下所示：

*public static Object newProxyInstance(ClassLoader loader,*

*Class<?>[] interfaces,*

*InvocationHandler h) {*

*final Class<?>[] intfs = interfaces.clone();*

*Class<?> cl = getProxyClass0(loader, intfs);*

*try { //这里使用反射技术来生成代理类*

*final Constructor<?> cons = cl.getConstructor(constructorParams);*

*final InvocationHandler ih = h;*

*.....*

*return cons.newInstance(new Object[]{h});*

*}*

*}*

ProxyClass0类的最终调用由ProxyGenerator.generateProxyClass来完成，源码如下所示：

*public static byte[] generateProxyClass(final String var0, Class<?>[] interfaces, int var2) {*

*ProxyGenerator var3 = new ProxyGenerator(var0, var1, var2);*

*final byte[] var4 = var3.generateClassFile();*

*if (saveGeneratedFiles) {*

*AccessController.doPrivileged(new PrivilegedAction<Void>() {*

*public Void run() {*

*try {*

*int var1 = var0.lastIndexOf(46);*

*Path var2;*

*if (var1 > 0) {*

*Path var3 = Paths.get(var0.substring(0, var1).replace('.', File.separatorChar));*

*Files.createDirectories(var3);*

*var2 = var3.resolve(var0.substring(var1 + 1, var0.length()) + ".class");*

*} else {*

*var2 = Paths.get(var0 + ".class");*

*}*

*Files.write(var2, var4, new OpenOption[0]);*

*......*

*}*

*return var4;*

*}*

在Rpc通信模型中，代理模式的使用方式是提供远程(Remote)代理，为一个位于不同的地址空间对象提供局域代表对象，封装网络通信细节，从而实现远程方法的透明调用。以Hadoop Rpc为例，反向代理封装了网络通信模块，如下所示:

*public static <T> Object create(Class<T> iface,*

*FailoverProxyProvider<T> proxyProvider,*

*Map<String,RetryPolicy> methodNameToPolicyMap,*

*RetryPolicy defaultPolicy) {*

*return Proxy.newProxyInstance(*

*proxyProvider.getInterface().getClassLoader(),*

*new Class<?>[] { iface },*

*new RetryInvocationHandler<T>(proxyProvider, defaultPolicy,*

*methodNameToPolicyMap)*

*);*

*}*

核心是RetryInvocationHandler，其中成员变量proxyProvider，负责具体的网络通信，最终的网络封装逻辑在Invoker实现类中，网络通信的核心实现如下：

*@Override*

*public Message invoke(Object proxy, final Method method, Object[] args)*

*throws ServiceException {*

*RequestHeaderProto rpcRequestHeader = constructRpcRequestHeader(method);*

*final Message theRequest = (Message) args[1];*

*final RpcWritable.Buffer val;*

*try {*

*val = (RpcWritable.Buffer) client.call(RPC.RpcKind.RPC\_PROTOCOL\_BUFFER,*

*new RpcProtobufRequest(rpcRequestHeader, theRequest), remoteId,*

*fallbackToSimpleAuth);*

*} ......*

*}*

其调用如下：

*Writable call(RPC.RpcKind rpcKind, Writable rpcRequest,*

*ConnectionId remoteId, int serviceClass,*

*AtomicBoolean fallbackToSimpleAuth) throws IOException {*

*final Call call = createCall(rpcKind, rpcRequest);*

*final Connection connection = getConnection(remoteId, call, serviceClass,*

*fallbackToSimpleAuth);*

*try {*

*checkAsyncCall();*

*try {*

*connection.sendRpcRequest(call); // send the rpc request*

*}..... } }*

将rpcRequest发送给Remote Server。

# 消息的编解码

在invoke中要封装通信细节，即确定客户端和服务端互信通信的结构，包括：

* 接口名称，在示例中接口名为Subject
* 方法名，接口中多种方法
* 参数类型及参数值
* 超时时间
* requestID，标识唯一请求ID

消息的编解码可以看做是客户端和Server端商定Rpc的通信协议，其中包括序列化方式、协议版本、类型、方法签名等，这些信息组装成一个协议包后交付给传输层进行发送。

## 2.1 调用消息封装

在Hadoop RPC中Client端通过ProtobufRpcEngine::getProxy构建动态代理的接口对应，然后通过Invoker类将消息封装后发送给Server：

*public Message invoke(Object proxy, final Method method, Object[] args)*

*throws ServiceException {*

*RequestHeaderProto rpcRequestHeader = constructRpcRequestHeader(method);*

*final Message theRequest = (Message) args[1];*

*final RpcWritable.Buffer val;*

*try {*

*val = (RpcWritable.Buffer) client.call(RPC.RpcKind.RPC\_PROTOCOL\_BUFFER,*

*new RpcProtobufRequest(rpcRequestHeader, theRequest), remoteId,*

*fallbackToSimpleAuth);*

*}*

*}*

1. RequestHeaderProtoInvoker，根据访问接口签名信息构造RequestHeaderProto对象，Server接收到Socket消息后读取这个RequestHeaderProto，其结构如下：

* methodName，调用的方法名
* protocolName，调用的通信名
* clientProtocolVersion，客户端协议版本

1. Call对象，RequestHeaderProto对象随着Message对象封装成Call对象传递给Client发送，在Call对象中，其结构如下：

* rpcKind，目前支持RPC\_WRITABLE及RPC\_PROTOCOL\_BUFFER
* rpcRequest，参数值，args[1]
* callID，调用标识

最终将rpcRequestHandler和args封装成RpcProtobufRequest，即构成调用实体，其他信息为辅助信息。

1. Client在调用之前进行一步的封装，见sendRpcRequest其中添加的消息包括：

* RPC\_FINAL\_PACKET
* RETRY
* ClientID

1. 将消息转换成字节组，如下：

*final ResponseBuffer buf = new ResponseBuffer();  
header.writeDelimitedTo(buf);  
RpcWritable.wrap(call.rpcRequest).writeTo(buf);*

*// RpcRequestHeader + RpcRequest*

*ipcStreams.sendRequest(buf.toByteArray());*

## 2.2 调用消息解码

Server接收到消息后，在Hadoop RPC#Server中Reader模块负责内部通信数据的解析工作，其不断从Connection中读取对象，最终调用processOneRpc处理，其核心源码如下：

*private void processOneRpc(ByteBuffer bb)*

*throws IOException, InterruptedException {*

*int callId = -1;*

*int retry = RpcConstants.INVALID\_RETRY\_COUNT;*

*final RpcWritable.Buffer buffer = RpcWritable.Buffer.wrap(bb);*

*final RpcRequestHeaderProto header =*

*getMessage(RpcRequestHeaderProto.getDefaultInstance(), buffer);*

*callId = header.getCallId();*

*retry = header.getRetryCount();*

*processRpcRequest(header, buffer);*

*}*

1. 从ByteBuffer中获取RpcRequestHeaderProto

*final RpcRequestHeaderProto header =  
 getMessage(RpcRequestHeaderProto.getDefaultInstance(), buffer)*

1. 将HeaderProto及rpcRequest，封装成RpcCall

*private void processRpcRequest(RpcRequestHeaderProto header,*

*RpcWritable.Buffer buffer) throws RpcServerException,*

*InterruptedException {*

*Class<? extends Writable> rpcRequestClass = //Header中有RpcKind*

*getRpcRequestWrapper(header.getRpcKind());*

*Writable rpcRequest;*

*try { //读取RpcRequest序列化对象，即 RpcProtobufRequest*

*rpcRequest = buffer.newInstance(rpcRequestClass, conf);*

*}*

*RpcCall call = new RpcCall(this, header.getCallId(),*

*header.getRetryCount(), rpcRequest,*

*ProtoUtil.convert(header.getRpcKind()),*

*header.getClientId().toByteArray(), traceScope, callerContext)*

*internalQueueCall(call);*

*}*

1. 将信息封装成RpcCall，其包括：

* rpcRequest , RpcProtobufRequest
* CallId
* Retry
* Rpc Kinkd
* Client Id等

将RpcCall存放到CallQueue中，等待调度

1. ProtobufRpcEngine处理RpcCall，最终调用RpcCall#run，其执行源码如下：

*public Void run() throws Exception {*

*Writable value = null;*

*ResponseParams responseParams = new ResponseParams();*

*try {*

*value = call(*

*rpcKind, connection.protocolName, rpcRequest, timestamp);*

*} .....*

*}*

在call中会处理RpcCall信息，如下：

*public Writable call(RPC.Server server,*

*String connectionProtocolName,*

*Writable writableRequest,*

*long receiveTime) {*

*//将writableRequest转换成RpcProtobufRequest*

*RpcProtobufRequest request = (RpcProtobufRequest) writableRequest;*

*//获取RequestHeader信息*

*RequestHeaderProto rpcRequest = request.getRequestHeader();*

*//获取方法名*

*String methodName = rpcRequest.getMethodName();*

*//获取协议类名*

*String declaringClassProtoName =*

*rpcRequest.getDeclaringClassProtocolName();*

*//获取客户端版本号*

*long clientVersion = rpcRequest.getClientProtocolVersion();*

*ProtoClassProtoImpl protocolImpl = getProtocolImpl(server,*

*declaringClassProtoName, clientVersion);*

*BlockingService service = (BlockingService) protocolImpl.protocolImpl;*

*MethodDescriptor methodDescriptor = service.getDescriptorForType()*

*.findMethodByName(methodName);*

*Message prototype = service.getRequestPrototype(methodDescriptor);*

*Message param = request.getValue(prototype);*

*try {*

*result = service.callBlockingMethod(methodDescriptor, null, param);*

*}*

*return RpcWritable.wrap(result);*

*}*

*}*

以上标红为具体从Server中获取方法并执行。

# 序列化

根据上节介绍，Rpc客户端和Server端通过协议好的数据结构进行调用消息的封装，最后通过网络通信进行调用信息的传输。为了优化网络通信的传输效率，需要对信息进行序列化，以实现网络传输及调高传输效率。序列化一般是将应用对象转换成字节流的过程，反之由字节流转换为应用对象的过程叫做解码或者反序列化。序列化是将对象的状态信息转换为可以存储或传输形式的过程，在序列化期间，对象将其当前状态写入到临时存储区或持久性存储区，之后可以通过从存储区中读取或反序列化对象的状态信息来重新创建该对象。

Rpc系统中，为了支持扩展性，通常会支持多种序列化方式以供使用者插拔使用，目前常用的序列化方案包括Probobuf,Thrift,Avro及Kyro等。

## 3.1 Java原生序列化

将Java对象序列化为二进制文件的是Java技术重要的技术，JDK中提供被序列化接口Serializable接口，定义如下所示：

*public interface Serializable {  
}*

Java中需要被序列化的类只需要实现该接口即可，示例如下：

1. 定义可序列化类

*public class Employee implements Serializable {*

*public String name;*

*public String address;*

*public transient int SSN;*

*public void mailCheck() {*

*System.out.println("Mailing a check to " + name + " " + address);*

*}*

*}*

该类必须实现Serializable接口而且所有属性必须是可序列化的，如果有属性不可序列化则注明为transient类型

1. 序列化对象

使用ObjectOutputStream类来序列化对象，定义SerializeDemo的例子来实例化Employee对象，并将该对象序列化到文件中

*public class SerializeDemo {*

*public static void main(String[] args) throws Exception {*

*Employee emp = new Employee();*

*emp.name = "Reyan Ali";*

*emp.address = "Suzhou,CN";*

*emp.SSN = 112233;*

*FileOutputStream fileOut = new FileOutputStream("e://employee.ser");*

*ObjectOutputStream out = new ObjectOutputStream(fileOut);*

*out.writeObject(emp);*

*out.close();*

*fileOut.close();*

*System.out.println("Serialized Data is saved in c://employee.ser");*

*}*

*}*

1. 反序列化对象，示例如下：

*public class DeserializeDemo {*

*public static void main(String[] args) throws Exception {*

*Employee emp = null;*

*FileInputStream fileIn = new FileInputStream("e://employee.ser");*

*ObjectInputStream in = new ObjectInputStream(fileIn);*

*emp = (Employee) in.readObject();*

*in.close();*

*fileIn.close();*

*System.out.println("Deserialized Employee..");*

*System.out.println("Employee Info: " + emp.name + "");*

*System.out.println("Employee SSN: " + emp.SSN);*

*}*

*}*

Serializable主要用来支持两种主要的特性：

* Java RMI，Java类库中的Rpc原生实现，通过该序列化方法来发送参数和接收返回值
* Java Bean，Bean的状态信息在设计是配置，以便程序运行时恢复这些状态信息

http://www.runoob.com/java/java-serialization.html

## 3.2 Protobuf

Protocal Buffers是Google开源的用于结构化的数据序列化和反序列化，可以用于Rpc和持续数据存储系统，类似于XML生成和解析，但是效率高

1. 定义proto文件

*syntax = "proto2";*

*option java\_package = "com.fys.io.ser.protobuf";*

*option java\_outer\_classname = "PersonModel";*

*message Person {*

*required int32 id = 1;*

*required string name = 2;*

*required string email = 3;*

*}*

1. 使用protoc命令编译，有两种方式：命令行及maven 插件，这里使用命令行，如下：

*protoc --java\_out=./ person.proto*

1. 测试程序，执行如下：

*public class ProtobufTest {*

*public static void main(String[] args) throws Exception {*

*PersonModel.Person.Builder builder = PersonModel.Person.newBuilder();*

*builder.setId(1);*

*builder.setName("Test");*

*builder.setEmail("Test@Test.com");*

*PersonModel.Person person = builder.build();*

*System.out.println("befrore:" + person);*

*System.out.println("Person Bytes:");*

*for(byte b : person.toByteArray()) {*

*System.out.println(b);*

*}*

*System.out.println("====================");*

*byte[] byteArray = person.toByteArray();*

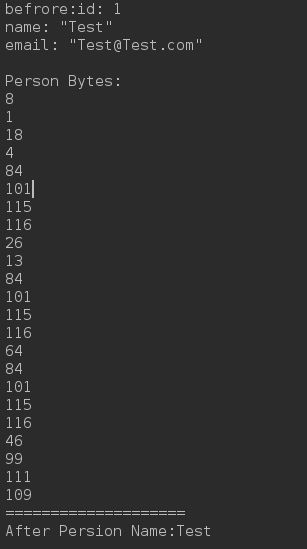
*PersonModel.Person p2 = PersonModel.Person.parseFrom(byteArray);*

*System.out.println("After Persion Name:" + p2.getName());*

*}*

*}*

输出如下：



https://www.cnblogs.com/kaituorensheng/p/9022591.html

Thrift的序列化使用方法与Protobuf类似，但是Thrift提供全套的RPC解决方案，包括传输层、并发处理框架等。Protobuf仅Service实现接口，其底层的通讯，需要自己实现。

## 3.3 Avro

Apache Avro是一个数据序列化系统，完成对象与二进制之间的转换，在传输数据前将对象转换成二进制流，到达目的地址后，Avro再将二进制刘转换成对象。其提供：

1. 丰富的数据结构
2. 紧凑、快速的二进制数据格式
3. 远程过程调用
4. 简单的动态语言集成
5. 代码的生成不需要读写数据文件，也不需要使用或者实现RPC协议，代码的生成仅是一个可选的优化，但是只对静态语言有意义

在对象序列化技术中，使用JSON序列化是一种轻量级的数据传输格式，但是JSON的格式是<key,value>类型，对存储空间浪费，而且会增加数据传输的压力。使用Avro可以解决这个问题，其序列化的文件由schema和真实数据组成：schema只是数据的元数据，相当于JSON数据的key信息，单独保存在一个JSON文件中，只需要保存一次。

Schama是描述对象的文件，通过模式定义各种数据结构，在数据进行序列化和反序列化之前都要先确定模式的结构，Avro的schema是由JSON定义的。当Avro用于RPC时，客户端和服务端在连接握手阶段进行Schema的交换，示例如下：

*{*

*"namespace": "example.avro",*

*"type": "record",*

*"name": "User",*

*"fields": [*

*{ "name": "name","type": "string" },*

*{ "name": "favorite\_number", "type": ["int", "null"]},*

*{ "name": "favorite\_color", "type": ["string", "null"] }*

*]*

*}*

Schema由下列JSON对象定义：

* JSON字符串，命名
* JSON对象：{“type”:”typeName”,...attributes}
* JSON数组，Avro中Union的定义

其实现示例如下：

1. 添加Avro依赖

*<dependency>*

*<groupId>org.apache.avro</groupId>*

*<artifactId>avro</artifactId>*

*<version>1.8.1</version>*

*</dependency>*

*<plugin>*

*<groupId>org.apache.avro</groupId>*

*<artifactId>avro-maven-plugin</artifactId>*

*<version>1.8.1</version>*

*<executions>*

*<execution>*

*<phase>generate-sources</phase>*

*<goals>*

*<goal>schema</goal>*

*<goal>protocol</goal>*

*<goal>idl-protocol</goal>*

*</goals>*

*<configuration>*

*<sourceDirectory>${project.basedir}/src/main/avro/</sourceDirectory>*

*<outputDirectory>${project.basedir}/src/main/java/</outputDirectory>*

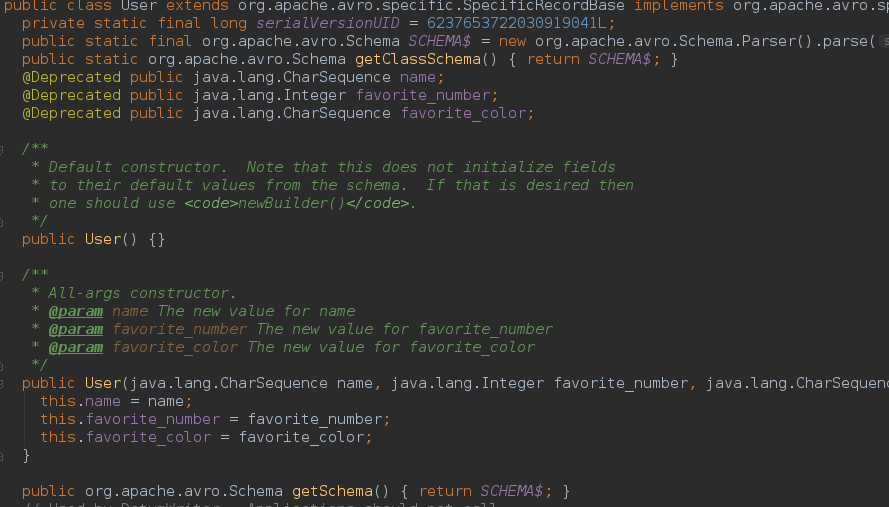
*</configuration>*

*</execution>*

*</executions>*

*</plugin>*

1. 定义Avro Schema文件，上面示例中的user.schema，放置目录main/avro，编译后在java目录中生成类User.java，截图如下：



1. 使用如下：

*public class TestUser {*

*public static void main(String[] args) throws IOException {*

*User user1 = new User();*

*user1.setName("Alysa");*

*user1.setFavoriteNumber(256);*

*User user2 = new User("ben",7,"red");*

*User user3 = User.newBuilder()*

*.setFavoriteNumber(null)*

*.setFavoriteColor("blue")*

*.setName("charlie")*

*.build();*

*DatumWriter<User> userDatumWriter = new SpecificDatumWriter<User>(User.class);*

*DataFileWriter<User> dataFileWriter = new DataFileWriter<User>(userDatumWriter);*

*dataFileWriter.create(user1.getSchema(), new File("users.avro"));*

*dataFileWriter.append(user1);*

*dataFileWriter.append(user2);*

*dataFileWriter.append(user3);*

*dataFileWriter.close();*

*DatumReader<User> userDatumReader = new SpecificDatumReader<User>(User.class);*

*DataFileReader<User> dataFileReader = new DataFileReader<User>(new File("users.avro"), userDatumReader);*

*User user = null;*

*while(dataFileReader.hasNext()) {*

*user = dataFileReader.next(user);*

*System.out.println(user);*

*}*

*}*

*}*

Avro中实现了Rpc通信库，其定义在avro-ipc中，其网络通信协议基于Netty来实现，在这里不再介绍。

## 3.4 示例-Hadoop序列化

以上介绍的几种序列化方式是Hadoop中常用的序列化方式，其DFSClient访问NameNode为例，简要介绍Protobuf的使用，其ClientNameNodeProtocol.proto定义如下：

*option java\_package = "org.apache.hadoop.hdfs.protocol.proto";*

*option java\_outer\_classname = "ClientNamenodeProtocolProtos";*

*option java\_generic\_services = true;*

*option java\_generate\_equals\_and\_hash = true;*

*package hadoop.hdfs;*

*import "Security.proto";*

*import "hdfs.proto";*

*import "acl.proto";*

*import "xattr.proto";*

*import "encryption.proto";*

*import "inotify.proto";*

*import "erasurecoding.proto";*

*message GetBlockLocationsRequestProto { //请求Proto，客户端将请求序列化后传给NN*

*required string src = 1; // file name*

*required uint64 offset = 2; // range start offset*

*required uint64 length = 3; // range length*

*}*

*message GetBlockLocationsResponseProto { //NameNode 响应序列化结果*

*optional LocatedBlocksProto locations = 1;*

*}*

1. 客户端调用，通过接口ClientNameNode#getBlockLocations调用，其实现类为ClientNameNodeProtocolTranslatorPB，将其转换为Protobuf请求

*public LocatedBlocks getBlockLocations(String src, long offset, long length)*

*throws IOException {*

*GetBlockLocationsRequestProto req = GetBlockLocationsRequestProto*

*.newBuilder()*

*.setSrc(src)*

*.setOffset(offset)*

*.setLength(length)*

*.build();*

*try {*

*GetBlockLocationsResponseProto resp = rpcProxy.getBlockLocations(null,*

*req);*

*return resp.hasLocations() ?*

*PBHelperClient.convert(resp.getLocations()) : null;*

*} catch (ServiceException e) {*

*throw ProtobufHelper.getRemoteException(e);*

*}*

*}*

反向代理类rpcProxy将request进行封装成RpcProtobufRequest，其执行如下：

*final Message theRequest = (Message) args[1] <= GetBlockLocationRequestProto*

*val = (RpcWritable.Buffer) client.call(RPC.RpcKind.RPC\_PROTOCOL\_BUFFER,  
 new RpcProtobufRequest(rpcRequestHeader, theRequest), remoteId,  
 fallbackToSimpleAuth);*

后传给RPC Server，具体过程不再介绍。下面是将其写入到Buffer中，通过网络发送到Server端

*void writeTo(ResponseBuffer out) throws IOException {*

*int length = message.getSerializedSize();*

*length += CodedOutputStream.computeRawVarint32Size(length);*

*out.ensureCapacity(length);*

*message.writeDelimitedTo(out);*

*}*

Server端接收到请求后，将buffer转换成RpcRequest，执行如下：

*Writable rpcRequest rpcRequest = buffer.newInstance(rpcRequestClass, conf);*

https://github.com/eishay/jvm-serializers/wiki

https://blog.csdn.net/fenglongmiao/article/details/79425218

# 网络通信层

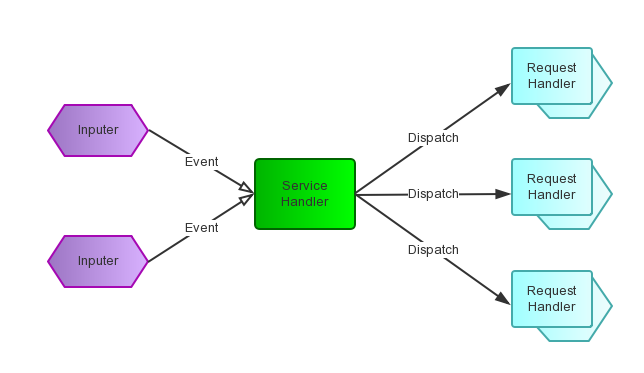
消息结构被序列化化二进制传以后，下一步要进行通络通信，目前有两种IO通信模型：

1. BIO，Linux网络编程中最常用的是阻塞IO模型，目前不常用
2. NIO，JDK 1.4之后引入NIO，弥补BIO的不足，它在标准Java代码中提供高速、面向块的IO，其核心思路是处理多个客户端请求使用IO多路复用技术

目前常见的Rpc通信方式，基于Java NIO实现的Reacotr设计模式，其用于处理多客户端并发输入的事件处理模式，其使用多路复用器将请求分发到相关的请求处理handler上。核心概念如下：

1. 事件驱动
2. 一个或者多个并发输入源
3. Service Handler，同步将输入请求（event）多路复用的分发给Request Handler
4. Request Handler

如下图所示:



在结构上类似生产者消费者模式，即有一个或者多个生产者将事件放入一个Queue中，消费者主动从Queue中Poll事件来处理。在Reactor模式则没有Queue来做缓冲，每当Event输入到Service Handler之后，Service Handler主动的根据不同的Event类型将其分发给对应的Request Handler来处理。

目前常见的Rpc通信方式的实现由以下几种：

* 基于Java NIO方式，这种方式较为复杂，要实现全套的Reactor模型，如Hadoop Rpc，Vert.x Rpc
* 基于Mina
* 基于Netty，目前使用最多的Rpc框架，例如Spark, Dubbo，Finagle，HSF等

https://blog.csdn.net/u011170180/article/details/59495836

https://blog.csdn.net/daybreak1209/article/details/80509964

http://www.importnew.com/22003.html