# Thrift

## 1：介绍和安装

### 1.1：简单介绍：

Thrift是Apache下的一个子项目，最早是Facebook的项目，后来Facebook提供给Apache作为开源项目，在官网上，Thrift被描述为“Scalable Cross-Language Services Implementation”，说的通俗一些，Thrift具有以下特征：

它有自己的跨机器的通信框架，并提供一套库。

它是一个代码生成器，按照它的规则，可以生成多种编程语言的通讯过程代码。

        一般情况下的跨机器的通信框架都是跨软件平台的（Linux,windows）, 而Thrift最特别之处在于它是跨语言的：例如，你可以用几乎所有流行语言（C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript等等）来实现通讯过程，这样做的好处就是你不用为编程语言发愁，如果服务器端与客户端都需要编写，选择你最拿手或项目规定的语言，就可以生成一个通讯框架；如果编写一个服务器端程序，定义好通讯规则（在Thrift中是.thrift文件）后，你所采用的服务器端实现语言不会影响到客户端，以后使用的人可以采用其他编程语言来实现客户端。

### 1.2：官方文档地址：

https://thrift-tutorial.readthedocs.io/en/latest/intro.html

### 1.3：下载地址

http://archive.apache.org/dist/thrift/

## 2：数据类型

### 2.1：基本类型

bool：布尔值，true 或 false，对应 Java 的 boolean

byte：8 位有符号整数，对应 Java 的 byte

i16：16 位有符号整数，对应 Java 的 short

i32：32 位有符号整数，对应 Java 的 int

i64：64 位有符号整数，对应 Java 的 long

double：64 位浮点数，对应 Java 的 double

string：未知编码文本或二进制字符串，对应 Java 的 String

不支持无符号整数类型（有符号整数可以安全的转换为无符号整数）

### 2.2：结构体类型

struct Example {

1:i32 number**=**10,

2:i64 bigNumber,

3:double decimals,

4:string name**=**"thrifty"

}

struct：定义公共的对象，在 Java 中是一个 JavaBean

序号：序号非常重要。正整数，按照顺序排列使用。这个属性在Apache Thrift进行序列化的时候被使用。

字段性质：包括两种关键字：required 和 optional，如果您不指定，那么系统会默认为required。required表示这个字段必须有值，并且Apache Thrift在进行序列化时，这个字段都会被序列化；optional表示这个字段不一定有值，且Apache Thrift在进行序列化时，这个字段只有有值的情况下才会被序列化。

### 2.3：容器类型

list：对应 Java 的 ArrayList

set：对应 Java 的 HashSet

map：对应 Java 的 HashMap

### 2.4：异常类型

exception InvalidOperation {

1: i32 what,

2: string why

}

exception：对应 Java 的 Exception（继承该语言的异常）

### 2.5：服务类型

service **<**name**>** {

**<**returntype**>** **<**name**>**(**<**arguments**>**)

[throws (**<**exceptions**>**)]

**...**

}

An example:

service StringCache {

void set(1:i32 key, 2:string value),

string get(1:i32 key) throws (1:KeyNotFound knf),

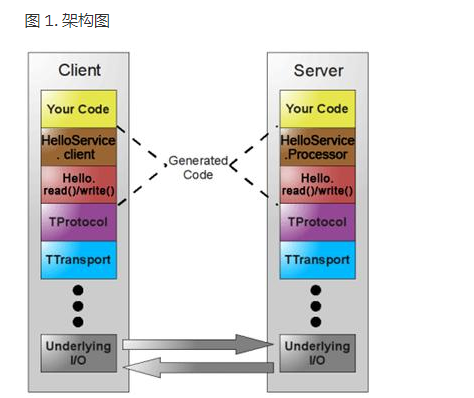
void delete(1:i32 key)

}

Void也是一种有效的类型

类似java中的接口（这里面定义的方法就是处理方法所要完成的处理逻辑）

## 3：thrift架构



黄色部分是用户实现的业务逻辑，褐色部分是根据 Thrift 定义的服务接口描述文件生成的客户端和服务器端代码框架，红色部分是根据 Thrift 文件生成代码实现数据的读写操作。红色部分以下是 Thrift 的传输体系、协议以及底层 I/O 通信，使用 Thrift 可以很方便的定义一个服务并且选择不同的传输协议和传输层而不用重新生成代码。

### 3.1：协议层

提供序列化和反序列化，将TTransport中的字节流转化为数据流（具有类型的数据）

传输协议分为文本（text）和二进制（binary）协议

#### TBinaryProtocol —— 二进制编码格式进行数据传输

##### 进行Integer序列化

public void writeI32(int i32) throws TException {  
 inoutTemp[0] = (byte)(0xff & (i32 >> 24));  
 inoutTemp[1] = (byte)(0xff & (i32 >> 16));  
 inoutTemp[2] = (byte)(0xff & (i32 >> 8));  
 inoutTemp[3] = (byte)(0xff & (i32));  
 trans\_.write(inoutTemp, 0, 4);  
}

public synchronized void write(byte b[], int off, int len) {  
 if ((off < 0) || (off > b.length) || (len < 0) ||  
 ((off + len) - b.length > 0)) {  
 throw new IndexOutOfBoundsException();  
 }  
 ensureCapacity(count + len);  
 System.*arraycopy*(b, off, buf, count, len);  
 count += len;  
}

通过四次位运算，得到长度为4的byte数组，这个数组不会随着整数的增加而变化，而且位运算是比较快的，。

##### 反序列化过程重新进行位运算就可以

public int readI32() throws TException {  
 byte[] buf = inoutTemp;  
 int off = 0;  
  
 if (trans\_.getBytesRemainingInBuffer() >= 4) {  
 buf = trans\_.getBuffer();  
 off = trans\_.getBufferPosition();  
 trans\_.consumeBuffer(4);  
 } else {  
 readAll(inoutTemp, 0, 4);  
 }  
 return  
 ((buf[off] & 0xff) << 24) |  
 ((buf[off+1] & 0xff) << 16) |  
 ((buf[off+2] & 0xff) << 8) |  
 ((buf[off+3] & 0xff));  
}

#### TCompactProtocol —— 高效率的、密集的二进制编码格式进行数据传输

使用zigzag编码格式紧凑传输协议，zigzag的优势在于记录数字类型，特别是对负数的记录，在计算机中，都会使用很大的数字表示负数，为了节约传输量，zigzag采用正数和负数交错的方式，把负数转化成一个正数进行记录

##### 对Integer进行序列化

private void writeVarint32(int n) throws TException {  
 int idx = 0;  
 while (true) {  
 if ((n & ~0x7F) == 0) {  
 temp[idx++] = (byte)n;  
 // writeByteDirect((byte)n);  
 break;  
 // return;  
 } else {  
 temp[idx++] = (byte)((n & 0x7F) | 0x80);  
 // writeByteDirect((byte)((n & 0x7F) | 0x80));  
 n >>>= 7;  
 }  
 }  
 trans\_.write(temp, 0, idx);  
}

public void write(byte[] buf, int off, int len) throws TTransportException {  
 try {  
 byteBuffer.put(buf, off, len);  
 } catch (BufferOverflowException e) {  
 throw new TTransportException("Not enough room in output buffer", e);  
 }  
}

首先使用zigzag编码把某个负数的符号位从高位移动到低位，在大多数情况下，完成一个32位整数的序列化，TCompactProtocol使用的空间比TbinaryProtocol小

##### 反序列化

private int readVarint32() throws TException {  
 int result = 0;  
 int shift = 0;  
 if (trans\_.getBytesRemainingInBuffer() >= 5) {  
 byte[] buf = trans\_.getBuffer();  
 int pos = trans\_.getBufferPosition();  
 int off = 0;  
 while (true) {  
 byte b = buf[pos+off];  
 result |= (int) (b & 0x7f) << shift;  
 if ((b & 0x80) != 0x80) break;  
 shift += 7;  
 off++;  
 }  
 trans\_.consumeBuffer(off+1);  
 } else {  
 while (true) {  
 byte b = readByte();  
 result |= (int) (b & 0x7f) << shift;  
 if ((b & 0x80) != 0x80) break;  
 shift += 7;  
 }  
 }  
 return result;  
}

如果传输的信息中，基本都是字符串，那么使用TbinaryProtocol和TCompactProtocol在效率上基本没有差别，如果需要传输的信息中有很多低位数字，那么使用TCompactProtocol

#### TJSONProtocol —— 使用 JSON 的数据编码协议进行数据传输

#### TSimpleJSONProtocol —— 只提供 JSON 只写的协议，适用于通过脚本语言解析

### 3.2：传输层

#### TSocket  使用阻塞式socket I/O

#### TFramedTransport  使用块进行传输，当使用非阻塞式服务器时，需要使用TFramedTransport

#### TMemoryTransport  使用缓存，java的实现是ByteArrayOutputStream

### 3.3：Processor

对TServer中的一次请求的InputProtocol和OutputTProtocol进行操作，从InputProtocol读取client的请求数据，向outputProtocol中写入用户逻辑的返回值。

TServer收到RPC请求时，调用TProcessor.process处理

public boolean process(TProtocol in, TProtocol out) throws TException {  
 TMessage msg = in.readMessageBegin();  
 ProcessFunction fn = processMap.get(msg.name);  
 if (fn == null) {  
 TProtocolUtil.*skip*(in, TType.*STRUCT*);  
 in.readMessageEnd();  
 TApplicationException x = new TApplicationException(TApplicationException.*UNKNOWN\_METHOD*, "Invalid method name: '"+msg.name+"'");  
 out.writeMessageBegin(new TMessage(msg.name, TMessageType.*EXCEPTION*, msg.seqid));  
 x.write(out);  
 out.writeMessageEnd();  
 out.getTransport().flush();  
 return true;  
 }  
 fn.process(msg.seqid, in, out, iface);  
 return true;  
}

首先调用得到rpc调用名称

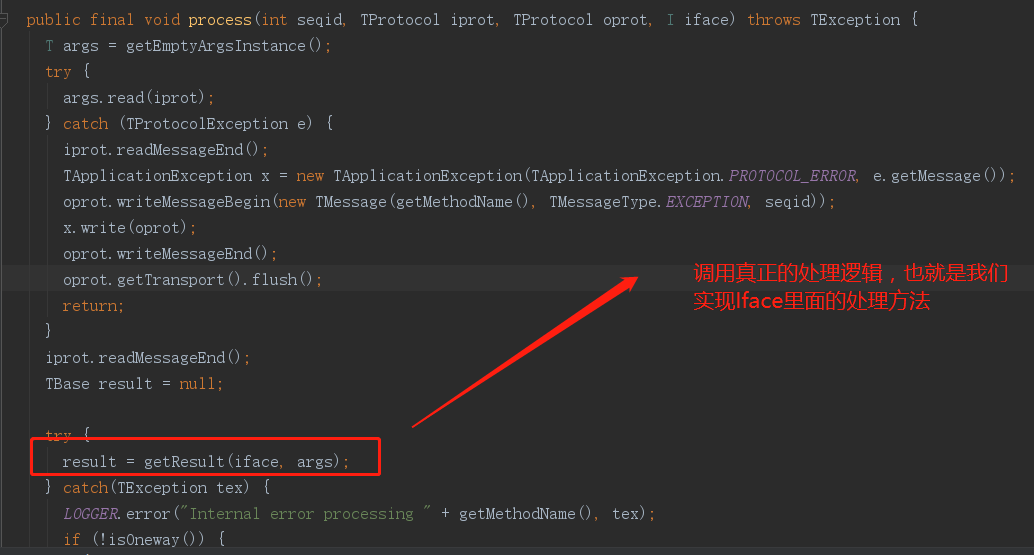
TMessage msg = in.readMessageBegin();

然后得到对应的RPC处理函数

ProcessFunction fn = processMap.get(msg.name);

如果没有对应的处理函数 抛出异常

ProcessFunction是用户真正处理逻辑的封装



### 3.4：服务端

#### TSimpleServer  单线程的服务器，使用标准的阻塞式IO，用来测试

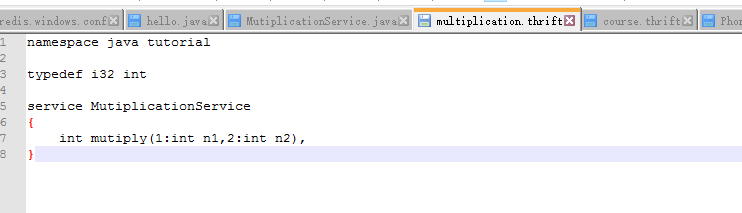
#### TThreadPoolServer  多线程的服务器，使用标准的阻塞式IO

#### TNonblockingServer 多线程的服务器，使用非阻塞式的IO（java中使用NIO）使用TFrameTransport必须使用此服务器

## 4：简单示例

配置环境变量D:\wbqutil\wbqSoftware\Thrift

新建文件multiplication.thrift



namespace java tutorial

typedef i32 int

service MutiplicationService

{

int mutiply(1:int n1,2:int n2),

}

使用管理员在命令行中输入thrift –gen java multiplication.thrift（避免新建目录失败）



生成service.java文件



### 4.1：客户端：

public class MultiplicationClient {  
 public static void main(String[] args) {  
 try {  
 TTransport tTransport;  
 tTransport = new TSocket("localhost", 9090);  
 tTransport.open();  
  
 TProtocol tProtocol = new TBinaryProtocol(tTransport);  
 MutiplicationService.Client client = new MutiplicationService.Client(tProtocol);  
  
 *perform*(client);  
 tTransport.close();  
 } catch (TException e) {  
 e.printStackTrace();  
 }  
 }  
  
 private static void perform(MutiplicationService.Client client) throws TException {  
 int product = client.mutiply(3, 5);  
 System.*out*.println("3\*5=" + product);  
 }  
}

ThriftClient跟TProcessor一样都主要操作InputProtocol和OutputProtocol，不同的是ThritClient将RPC调用分为Send和receive两个步骤。

1. Send步骤，将用户的调用参数作为一个整体的Struct写入TProcotol，并发送到TServer。
2. Send结束之后，ThriftClient便立刻进入Receive状态等待TServer的相应。对于TServer返回的响应，使用返回值解析类进行返回值解析，完成RPC调用。

### 4.2：处理逻辑（实现service中的Iface接口 编写具体的业务逻辑）

public class MultiplicationHandler implements MutiplicationService.Iface {  
 @Override  
 public int mutiply(int n1, int n2) throws TException {  
 System.*out*.println("Mutiply(" + n1 + "," + n2 + ")");  
 return n1 \* n2;  
 }  
}

### 4.3：服务端

public class MultiplicationServer {  
 public static MultiplicationHandler *multiplicationHandler*;  
 public static MutiplicationService.Processor *processor*;  
  
 public static void main(String[] args) {  
 try {  
 *multiplicationHandler* = new MultiplicationHandler();  
 *processor* = new MutiplicationService.Processor(*multiplicationHandler*);  
  
 Runnable simple = () -> *simple*(*processor*);  
 new Thread(simple).start();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 public static void simple(MutiplicationService.Processor processor) {  
 try {  
 TServerTransport serverTransport = new TServerSocket(9090);  
 TServer server = new TSimpleServer(new TServer.Args(serverTransport).processor(processor));  
  
 System.*out*.println("Starting the simple server");  
 server.serve();  
 } catch (TTransportException e) {  
 e.printStackTrace();  
 }  
  
 }  
}

### 4.4：结果

#### 4.4.1：开启服务器

#### 4.4.2：开启客户端

