# 1、概述

通过上一篇文章《[架构设计：系统间通信（10）——RPC的基本概念](http://blog.csdn.net/yinwenjie/article/details/49453303)》的介绍，相信读者已经理解了基本的RPC概念。为了加深这个理解，后面几篇文章我将详细讲解一款典型的RPC规范的实现Apache Thrift。Apache Thrift的介绍一共分为三篇文章，上篇讲解Apache Thrift的基本使用；中篇讲解Apache Thrift的工作原理（主要围绕Apache Thrift使用的消息格式封装、支持的网络IO模型和它的客户端请求处理方式）；下篇对Apache Thrift的不足进行分析，并基于Apache Thrift实现一个自己设计的RPC服务治理的管理方案。这样对我们后续理解Dubbo的服务治理方式会有很好的帮助作用。

# 2、基本知识

Thrift最初由facebook开发用做系统内各语言之间的RPC框架 。2007年由facebook贡献到apache基金 ，08年5月进入apache孵化器 ，称为Apache Thrift。和其他RPC实现相比，Apache Thrift主要的有点是：支持的语言多（C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk等多种语言）、并发性能高（还记得上篇文章中，我们提到的影响RPC性能的几个关键点吗？）。

为了支持多种语言，Apache Thrift有一套自己的接口定义语言，并且通过Apache Thrift的代码生成程序，能够生成各种编程语言的代码。这样是保证各种语言进行通讯的前提条件。为了能够实现简单的Apache Thrift实例，首先我们就需要讲解一下Apache Thrift的IDL。

## 2-1、Thrift代码生成程序安装

如果您是在windows环境下运行进行Apache Thrift的试验，那么您无需安装任何工具，直接下载Apache Thrift在windows下的代码生成程序<http://www.apache.org/dyn/closer.cgi?path=/thrift/0.9.3/thrift-0.9.3.exe>（在这篇文章写作时，使用的是Apache Thrift的0.9.3版本）；如果您运行在Linux系统下，那么下载<http://www.apache.org/dyn/closer.cgi?path=/thrift/0.9.3/thrift-0.9.3.tar.gz>，并进行编译、安装（过程很简单，这里就不再赘述了）。安装后记得添加运行位置到环境变量中。

## 2-2、IDL格式概要

以下是一个简单的IDL文件定义：

# 命名空间的定义 注意‘java’的关键字

namespace java testThrift.iface

# 结构体定义

struct Request {

1:required string paramJSON;

2:required string serviceName;

}

# 另一个结构体定义

struct Reponse {

1:required RESCODE responeCode;

2:required string responseJSON;

}

# 异常描述定义

exception ServiceException {

1:required EXCCODE exceptionCode;

2:required string exceptionMess;

}

# 枚举定义

enum RESCODE {

\_200=200;

\_500=500;

\_400=400;

}

# 另一个枚举

enum EXCCODE {

PARAMNOTFOUND = 2001;

SERVICENOTFOUND = 2002;

}

# 服务定义

service HelloWorldService {

Reponse send(1:Request request) throws (1:ServiceException e);

}

以上IDL文件是可以直接用来生成各种语言的代码的。下面给出常用的各种不同语言的代码生成命令：

# 生成java

thrift-0.9.3 -gen java ./demoHello.thrift

# 生成c++

thrift-0.9.3 -gen cpp ./demoHello.thrift

# 生成php

thrift-0.9.3 -gen php ./demoHello.thrift

# 生成node.js

thrift-0.9.3 -gen js:node ./demoHello.thrift

# 生成c#

thrift-0.9.3 -gen csharp ./demoHello.thrift

# 您可以通过以下命令查看生成命令的格式

thrift-0.9.3 -help

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

**2-2-1、基本类型**

基本类型就是：不管哪一种语言，都支持的数据形式表现。Apache Thrift中支持以下几种基本类型：

* bool: 布尔值 (true or false), one byte
* byte: 有符号字节
* i16: 16位有符号整型
* i32: 32位有符号整型
* i64: 64位有符号整型
* double: 64位浮点型
* string: 字符串/字符数组
* binary: 二进制数据（在java中表现为java.nio.ByteBuffer）

**2-2-2、struct结构**

在面向对象语言中，表现为“类定义”；在弱类型语言、动态语言中，表现为“结构/结构体”。定义格式如下：

struct <结构体名称> {

<序号>:[字段性质] <字段类型> <字段名称> [= <默认值>] [;|,]

}

* 1
* 2
* 3
* 4

实例：

struct Request {

1:required binary paramJSON;

2:required string serviceName

3:optional i32 field1 = 0;

4:optional i64 field2,

5: list<map<string , string>> fields3

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 结构体名称：可以按照您的业务需求，给定不同的名称（区分大小写）。但是要注意，一组IDL定义文件中结构体名称不能重复，且不能使用IDL已经占用的关键字（例如required 、struct 等单词）。
* 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。
* 字段类型：在struct中，字段类型可以是某一个基础类型，也可以是某一个之前定义好的struct，还可以是某种Apache Thrift支持的容器（set、map、list），还可以是定义好的枚举。字段的类型是必须指定的。
* 字段名称：字段名称区分大小写，不能重复，且不能使用IDL已经占用的关键字（例如required 、struct 等单词）。
* 默认值：您可以为某一个字段指定默认值（也可以不指定）。
* 结束符：在struct中，支持两种结束符，您可以使用“；”或者“，”。当然您也可以不使用结束符（Apache Thrift代码生成程序，会自己识别到）

**2-2-3、containers集合/容器**

Apache Thrift支持三种类型的容器，容器在各种编程语言中普遍存在：

* list< T >：有序列表（JAVA中表现为ArrayList），T可以是某种基础类型，也可以是某一个之前定义好的struct，还可以是某种Apache Thrift支持的容器（set、map、list），还可以是定义好的枚举。有序列表中的元素允许重复。
* set< T >：无序元素集合（JAVA中表现为HashSet），T可以是某种基础类型，也可以是某一个之前定义好的struct，还可以是某种Apache Thrift支持的容器（set、map、list），还可以是定义好的枚举。无序元素集合中的元素不允许重复，一旦重复后一个元素将覆盖前一个元素。
* map

**2-2-4、enmu枚举**

enum <枚举名称> {

<枚举字段名> = <枚举值>[;|,]

}

* 1
* 2
* 3

示例如下：

enum RESCODE {

\_200=200;

\_500=500;

\_400=400;

}

* 1
* 2
* 3
* 4
* 5

**2-2-5、常量定义**

Apache Thrift允许定义常量。常量的关键字为“const”，常量的类型可以是Apache Thrift的基础类型，也可以是某一个之前定义好的struct，还可以是某种Apache Thrift支持的容器（set、map、list），还可以是定义好的枚举。示例如下：

const i32 MY\_INT\_CONST = 111111;

const i64 MY\_LONG\_CONST = 11111122222222333333334444444;

const RESCODE MY\_RESCODE = RESCODE.\_200;

* 1
* 2
* 3
* 4
* 5

**2-2-6、exception 异常**

Apache Thrift的exception，主要在定义服务接口时使用。其定义方式类似于struct（您可以理解成，把struct关键字换成exception关键字即可），示例如下：

exception ServiceException {

1:required EXCCODE exceptionCode;

2:required string exceptionMess;

}

**2-2-7、service 服务接口**

Apache Thrift中最重要的IDL定义之一。在后续的代码生成阶段，通过IDL定义的这些服务将构成Apache Thrift客户端调用Apache Thrift服务端的基本远端过程。service服务接口的定义形式如下所示：

service <服务名称> {

<void | 返回指类型> <服务方法名>([<入参序号>:[required | optional] <参数类型> <参数名> ...]) [throws ([<异常序号>:[required | optional] <异常类型> <异常参数名>...])]

}

* 1
* 2
* 3
* 服务名称：服务名可以按照您的业务需求自行制定，注意服务名是区分大小写的。IDL中服务名称只有两个限制，就是不能重复使用相同的名称，不能使用IDL已经占用的关键字（例如required 、struct 等单词）。
* 返回值类型：如果这个调用方法没有返回类型，那么可以关键字“void”; 可以是Apache Thrift的基础类型，也可以是某一个之前定义好的struct，还可以是某种Apache Thrift支持的容器（set、map、list），还可以是定义好的枚举。
* 服务方法名：服务方法名可以根据您的业务需求自定制定，注意区分大小写。在同一个服务中，**不能重复使用一个服务方法名命名多个方法（一定要注意）**，不能使用IDL已经占用的关键字。
* 服务方法参数：<入参序号>:[required | optional] <参数类型> <参数名>。注意和struct中的字段定义相似，可以指定required或者optional；如果不指定则系统默认为required 。如果一个服务方法中有多个参数名，那么这些参数名称不能重复。
* 服务方法异常：throws ([<异常序号>:[required | optional] <异常类型> <异常参数名>。throws关键字是服务方法异常定义的开始点。在throws关键字后面，可以定义1个或者多个不同的异常类型。

Apache Thrift服务定义的示例如下：

service HelloWorldService {

Reponse send(1:Request request) throws (1:ServiceException e);

}

* 1
* 2
* 3

**2-2-8、namespace命名空间**

Apache Thrift支持为不同语言制定不同的命名空间：

namespace java testThrift.iface

namespace php testThrift.iface

namespace cpp testThrift.iface

* 1
* 2
* 3
* 4
* 5

**2-2-9、注释**

Apache Thrift 支持多种风格的注释。这是为了适应不同语言背景的开发者：

/\*

\* 注释方式1：

\*\*/

// 注释方式2

# 注释方式3

* 1
* 2
* 3
* 4
* 5
* 6
* 7

**2-2-10、include关键字**

如果您的整个工程中有多个IDL定义文件（IDL定义文件的文件名可以随便取）。那么您可以使用include关键字，在IDL定义文件A中，引入一个其他的IDL文件：

include "other.thrift"

* 1

请注意，一定使用双引号（不要用成中文的双引号咯），并且不使用“;”或者“,”结束符。

以上就是IDL基本的语法了，由于篇幅原因不可能把每种语法、每一个细节都讲到，但是以上的语法要点已经足够您编辑一个适应业务的，灵活的IDL定义了。如果您需要了解更详细的Thrift IDL语法，可以参考官方文档的讲述：<http://thrift.apache.org/docs/idl>

**2-3、最简单的Thrift代码**

* 定义Thrift中业务接口HelloWorldService.Iface的实现：

package testThrift.impl;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.apache.thrift.TException;

import testThrift.iface.HelloWorldService.Iface;

import testThrift.iface.RESCODE;

import testThrift.iface.Reponse;

import testThrift.iface.Request;

/\*\*

\* 我们定义了一个HelloWorldService.Iface接口的具体实现。<br>

\* 注意，这个父级接口：HelloWorldService.Iface，是由thrift的代码生成工具生成的<br>

\* 要运行这段代码，请导入maven-log4j的支持。否则修改LOGGER.info方法

\* @author yinwenjie

\*/

public class HelloWorldServiceImpl implements Iface {

/\*\*

\* 日志

\*/

private static final Log LOGGER = LogFactory.getLog(HelloWorldServiceImpl.class);

/\*\*

\* 在接口定义中，只有一个方法需要实现。<br>

\* HelloWorldServiceImpl.send(Request request) throws TException <br>

\* 您可以理解成这个接口的方法接受客户端的一个Request对象，并且在处理完成后向客户端返回一个Reponse对象<br>

\* Request对象和Reponse对象都是由IDL定义的结构，并通过“代码生成工具”生成相应的JAVA代码。

\*/

@Override

public Reponse send(Request request) throws TException {

/\*

\* 这里就是进行具体的业务处理了。

\* \*/

String json = request.getParamJSON();

String serviceName = request.getServiceName();

HelloWorldServiceImpl.LOGGER.info("得到的json：" + json + " ；得到的serviceName: " + serviceName);

// 构造返回信息

Reponse response = new Reponse();

response.setResponeCode(RESCODE.\_200);

response.setResponseJSON("{\"user\":\"yinwenjie\"}");

return response;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45

各位可以看到，上面一段代码中具体业务和过程和普通的业务代码没有任何区别。**甚至这段代码的实现都不知道自己将被Apache Thrift框架中的客户端调用**。

* 然后我们开始书写Apache Thrift的服务器端代码：

package testThrift.man;

import java.util.concurrent.Executors;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.apache.log4j.BasicConfigurator;

import org.apache.thrift.TProcessor;

import org.apache.thrift.protocol.TBinaryProtocol;

import org.apache.thrift.server.TThreadPoolServer;

import org.apache.thrift.server.TThreadPoolServer.Args;

import org.apache.thrift.transport.TServerSocket;

import testThrift.iface.HelloWorldService;

import testThrift.iface.HelloWorldService.Iface;

import testThrift.impl.HelloWorldServiceImpl;

public class HelloBoServerDemo {

static {

BasicConfigurator.configure();

}

/\*\*

\* 日志

\*/

private static final Log LOGGER =LogFactory.getLog(HelloBoServerDemo.class);

public static final int SERVER\_PORT = 9111;

public void startServer() {

try {

HelloBoServerDemo.LOGGER.info("看到这句就说明thrift服务端准备工作 ....");

// 服务执行控制器（只要是调度服务的具体实现该如何运行）

TProcessor tprocessor = new HelloWorldService.Processor<Iface>(new HelloWorldServiceImpl());

// 基于阻塞式同步IO模型的Thrift服务，正式生产环境不建议用这个

TServerSocket serverTransport = new TServerSocket(HelloBoServerDemo.SERVER\_PORT);

// 为这个服务器设置对应的IO网络模型、设置使用的消息格式封装、设置线程池参数

Args tArgs = new Args(serverTransport);

tArgs.processor(tprocessor);

tArgs.protocolFactory(new TBinaryProtocol.Factory());

tArgs.executorService(Executors.newFixedThreadPool(100));

// 启动这个thrift服务

TThreadPoolServer server = new TThreadPoolServer(tArgs);

server.serve();

} catch (Exception e) {

HelloBoServerDemo.LOGGER.error(e);

}

}

/\*\*

\* @param args

\*/

public static void main(String[] args) {

HelloBoServerDemo server = new HelloBoServerDemo();

server.startServer();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62

以上的代码有几点需要说明：

1. TBinaryProtocol：这个类代码Apache Thrift特有的一种二进制描述格式。它的特点是传输单位数据量所使用的传输量更少。Apache Thrift还支持多种数据格式，例如我们熟悉的JSON格式。后文我们将详细介绍Apache Thrift中的数据格式。
2. tArgs.executorService()：是不是觉得这个executorService很熟悉，是的这个就是JAVA JDK 1.5+ 后java.util.concurrent包提供的异步任务调度服务接口，Java标准线程池ThreadPoolExecutor就是它的一个实现。
3. server.serve()，由于是使用的同步阻塞式网络IO模型，所以这个应用程序的主线程执行到这句话以后就会保持阻塞状态了。不过下层网络状态不出现错误，这个线程就会一直停在这里。

另外，同HelloWorldServiceImpl 类中的代码，请使用Log4j。如果您的测试工程里面没有Log4j，请改用System.out。

* 接下来我们进行最简单的Apache Thrift Client的代码编写：

package testThrift.client;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.apache.log4j.BasicConfigurator;

import org.apache.thrift.protocol.TBinaryProtocol;

import org.apache.thrift.protocol.TProtocol;

import org.apache.thrift.transport.TSocket;

import testThrift.iface.HelloWorldService;

import testThrift.iface.Reponse;

import testThrift.iface.Request;

/\*\*

\* 同样是基于同步阻塞模型的thrift client。

\* @author yinwenjie

\*/

public class HelloClient {

static {

BasicConfigurator.configure();

}

/\*\*

\* 日志

\*/

private static final Log LOGGER = LogFactory.getLog(HelloClient.class);

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

// 服务器所在的IP和端口

TSocket transport = new TSocket("127.0.0.1", 9111);

TProtocol protocol = new TBinaryProtocol(transport);

// 准备调用参数

Request request = new Request("{\"param\":\"field1\"}", "\\mySerivce\\queryService");

HelloWorldService.Client client = new HelloWorldService.Client(protocol);

// 准备传输

transport.open();

// 正式调用接口

Reponse reponse = client.send(request);

// 一定要记住关闭

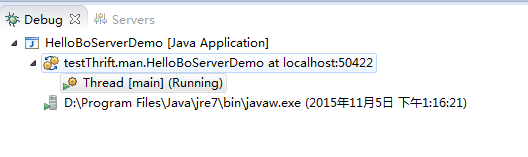
transport.close();

HelloClient.LOGGER.info("response = " + reponse);

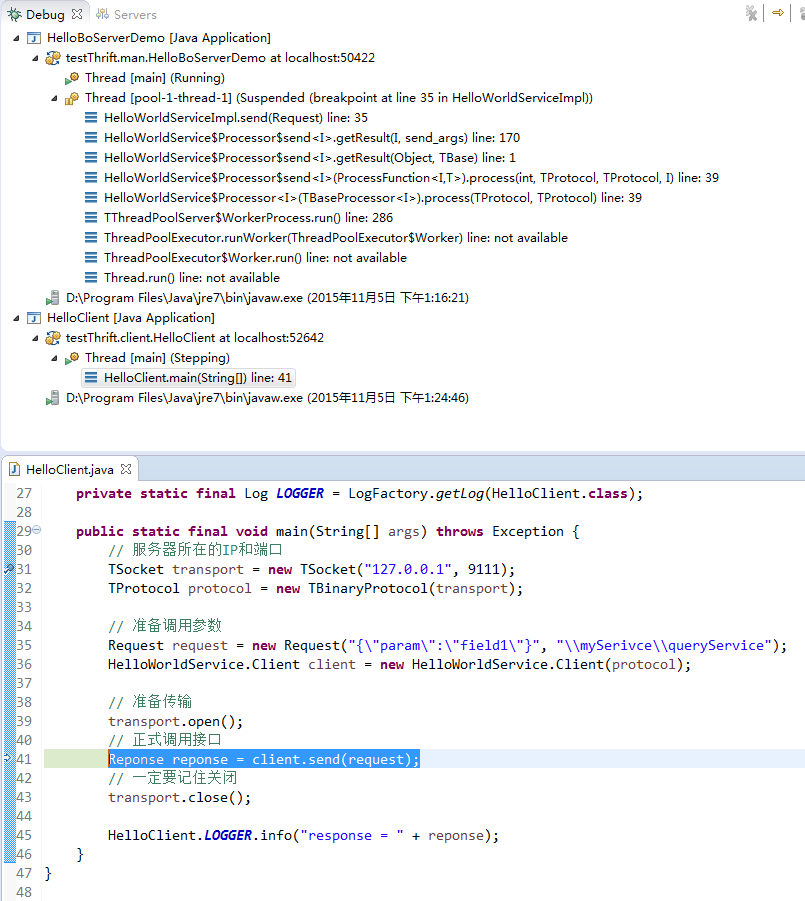
}

}

* Thrift客户端所使用的网络IO模型，必须要与Thrift服务器端所使用的网络IO模型一致。也就是说服务器端如果使用的是阻塞式同步IO模型，那么客户端就必须使用阻塞式同步IO模型。
* Thrift客户端所使用的消息封装格式，必须要与Thrift服务器端所使用的消息封装格式一直。也就是说服务器端如果使用的是二进制流的消息格式TBinaryProtocol，那么客户端就必须同样使用二进制刘的消息格式TBinaryProtocol。
* 其它的代码要么就是由IDL定义并由Thrift的代码生成工具生成；要么就不是重要的代码，所以为了节约篇幅就没有必要再贴出来了。以下是运行效果。
* 服务器端运行效果



* 服务器端收到客户端请求后，取出线程池中的线程进行运行



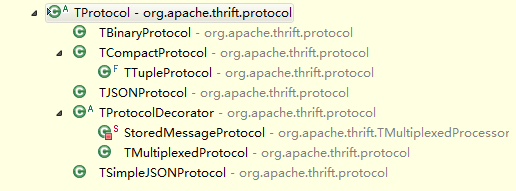
请注意服务器端在收到客户端请求后的运行方式：取出一条线程池中的线程，并且运行这个服务接口的具体实现。接下来我们马上介绍Apache Thrift的工作细节。

# 3、Aapche Thrift详解

在《[架构设计：系统间通信（10）——RPC的基本概念](http://blog.csdn.net/yinwenjie/article/details/49453303)》一文中，我专门介绍了一款RPC规范的具体实现中哪些要素和性能息息相关。包括了RPC通讯采用的数据封装格式、RPC通讯采用的网络IO模型和RPC所采用的请求处理方式。这个小节我们对Apache Thrift中的这三个要素，这样读者就可以知晓为什么Apache Thrift的性能如此高效了。

## 3-1、Aapche Thrift与消息格式

Apache Thrift支持多种消息格式封装。这些消息格式是如果进行编码和解码的是不需要使用者关心的，只需要根据自己的需要制定不同的消息封装格式即可。Apache Thrift所有消息格式封装的实现，都继承与TProtocol这个抽象类，如下图所示：



### 3-1-1、TBinaryProtocol

二进制流的编码格式。由于需要支持跨语言，所以Apache Thrift支持有限的几种通用类型，包括基本类型（Float、Double、Integer、Long、String、Short）、集合类型（Map、Set、List）还有Pojo类型（实际上就是前两者若干类型的组合形式）。

那么这个类所生成的二进制流和传统的java序列化后生成的二进制流有什么样的区别（或者是优势）呢？我们可以通过阅读TBinaryProtocol的源代码进行研究。

我们以TBinaryProtocol中，对Integer的序列化过程进行详细的解释，来对比java提供的其他几种序列化的方式找到不同。首先java中，如果要将一个Integer对象通过网络发送出去，要做的第一件事情就是序列化，那么我们常用的序列化方式有两种，如下所示：

* java中序列化Integer对象的第一种方法：

Integer integerObject = 10066329;

integerObject.toString().getBytes();

* java中序列化Integer对象的第二种方法：

ByteArrayOutputStream aStream = new ByteArrayOutputStream();

ObjectOutputStream oStream = new ObjectOutputStream(aStream);

oStream.writeObject(integerObject);

aStream.toByteArray();

第一种方式是将Integer对象中的值序列化；第二种方式，是将Integer整个对象序列化。这两种方式虽然都产生byte[]，实际上性质是完全不一样的。我们来看一下这两种方式产生的byte[]的内容：

* 序列化Integer的值：

[49, 48, 48, 54, 54, 51, 50, 57]

* 序列化整个Integer对象：

[-84, -19, 0, 5, 115, 114, 0, 17, 106, 97, 118, 97, 46, 108, 97, 110, 103, 46, 73, 110, 116, 101, 103, 101, 114, 18, -30, -96, -92, -9, -127, -121, 56, 2, 0, 1, 73, 0, 5, 118, 97, 108, 117, 101, 120, 114, 0, 16, 106, 97, 118, 97, 46, 108, 97, 110, 103, 46, 78, 117, 109, 98, 101, 114, -122, -84, -107, 29, 11, -108, -32, -117, 2, 0, 0, 120, 112, 0, -103, -103, -103]

第一种方式序列化后，byte数组有8个byte元素（因为是首先转换成字符串的，所以实际上这个大小会随着Integer值的大小增加而增加）；第二中方式序列化后，byte数组一共有 > 20 个byte元素，其中除了记录Integer的值以外，还包括描述这个类型的其他属性。

那么我们再来看看TBinaryProtocol中，是如何序列化Integer类型的。首先我们来看一下TBinaryProtocol进行Integer序列化的这部分源代码，如下图所示：

private byte[] i32out = new byte[4];

public void writeI32(int i32) throws TException {

i32out[0] = (byte)(0xff & (i32 >> 24));

i32out[1] = (byte)(0xff & (i32 >> 16));

i32out[2] = (byte)(0xff & (i32 >> 8));

i32out[3] = (byte)(0xff & (i32));

trans\_.write(i32out, 0, 4);

}

计算过程可以通过下图来表示：



通过4次位计算，得到了一个长度为4个byte数组，并且这个数组的大小并不会随着整数大小的增加而变化。并且位运算的速度是所有计算中速度最快的一种计算。反序列化的过程相似，对这个大小为4的byte[]数组重新进行位计算即可：

((buf[off] & 0xff) << 24) |

((buf[off+1] & 0xff) << 16) |

((buf[off+2] & 0xff) << 8) |

((buf[off+3] & 0xff));

由于本文的篇幅和写作目的所限，不能一一介绍TBinaryProtocol的各种序列化方式，但是通过对TBinaryProtocol中Integer的序列化过程，我们可以找到TBinaryProtocol处理过程的优势，包括速度和大小的优势。所以，**如果您的使用环境对序列化过程没有特别的要求（例如后面要提到的大量的负数情况），那么直接使用TBinaryProtocol进行数据格式的封装的就可以了。**

byte是一个8位二进制描述（一个字节），在java中，一个int需要4个byte进行表示，而。“0x”的前缀表示16进制数字，那么0xff的二进制表示就是 1111 1111；“&”是“与”运算符，这个运算符用于二进制计算，1 & 1 = 1，其余情况都 = 0；“<<” 表示左移运算，0011 << 2 = 1100；”>>”表示右移运算，1100 >> 2 = 0011；

### 3-1-2、TCompactProtocol

使用zigzag编码方式紧凑传输协议。zigzag编码的优势在于记录数字类型（整数、单精度浮点和双精度浮点），最特别的是zigzag编码对负数的记录。在计算机中，都会使用很大的数字表示负数，为了保证节约传输量，zigzag编码采用正数与负数交错的方式，把负数转换为一个正数进行记录。下面我们具体来分析一下TCompactProtocol中对32位整数的序列化方式，以下是TCompactProtocol中对32为整数的处理代码：

/\*\*

\* Write an i32 as a zigzag varint.

\*/

public void writeI32(int i32) throws TException {

writeVarint32(intToZigZag(i32));

}

/\*\*

\* Convert n into a zigzag int. This allows negative numbers to be

\* represented compactly as a varint.

\*/

private int intToZigZag(int n) {

return (n << 1) ^ (n >> 31);

}

/\*\*

\* Write an i32 as a varint. Results in 1-5 bytes on the wire.

\* TODO: make a permanent buffer like writeVarint64?

\*/

byte[] i32buf = new byte[5];

private void writeVarint32(int n) throws TException {

int idx = 0;

while (true) {

if ((n & ~0x7F) == 0) {

i32buf[idx++] = (byte)n;

// writeByteDirect((byte)n);

break;

} else {

i32buf[idx++] = (byte)((n & 0x7F) | 0x80);

// writeByteDirect((byte)((n & 0x7F) | 0x80));

n >>>= 7;

}

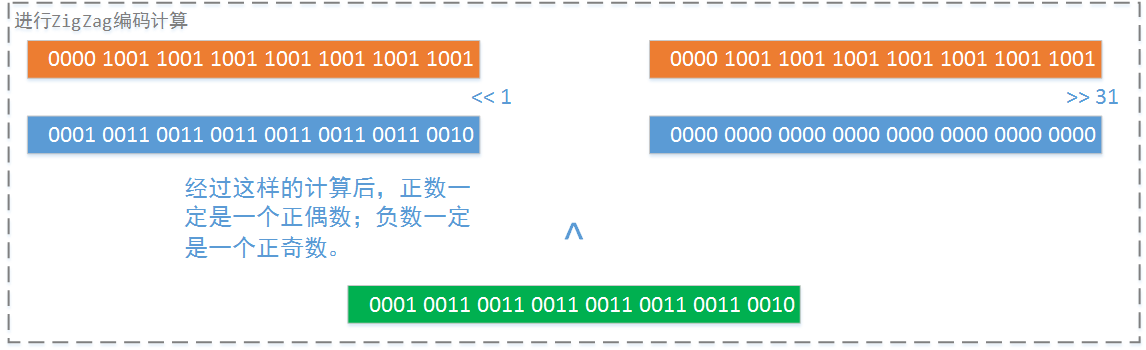
}

trans\_.write(i32buf, 0, idx);

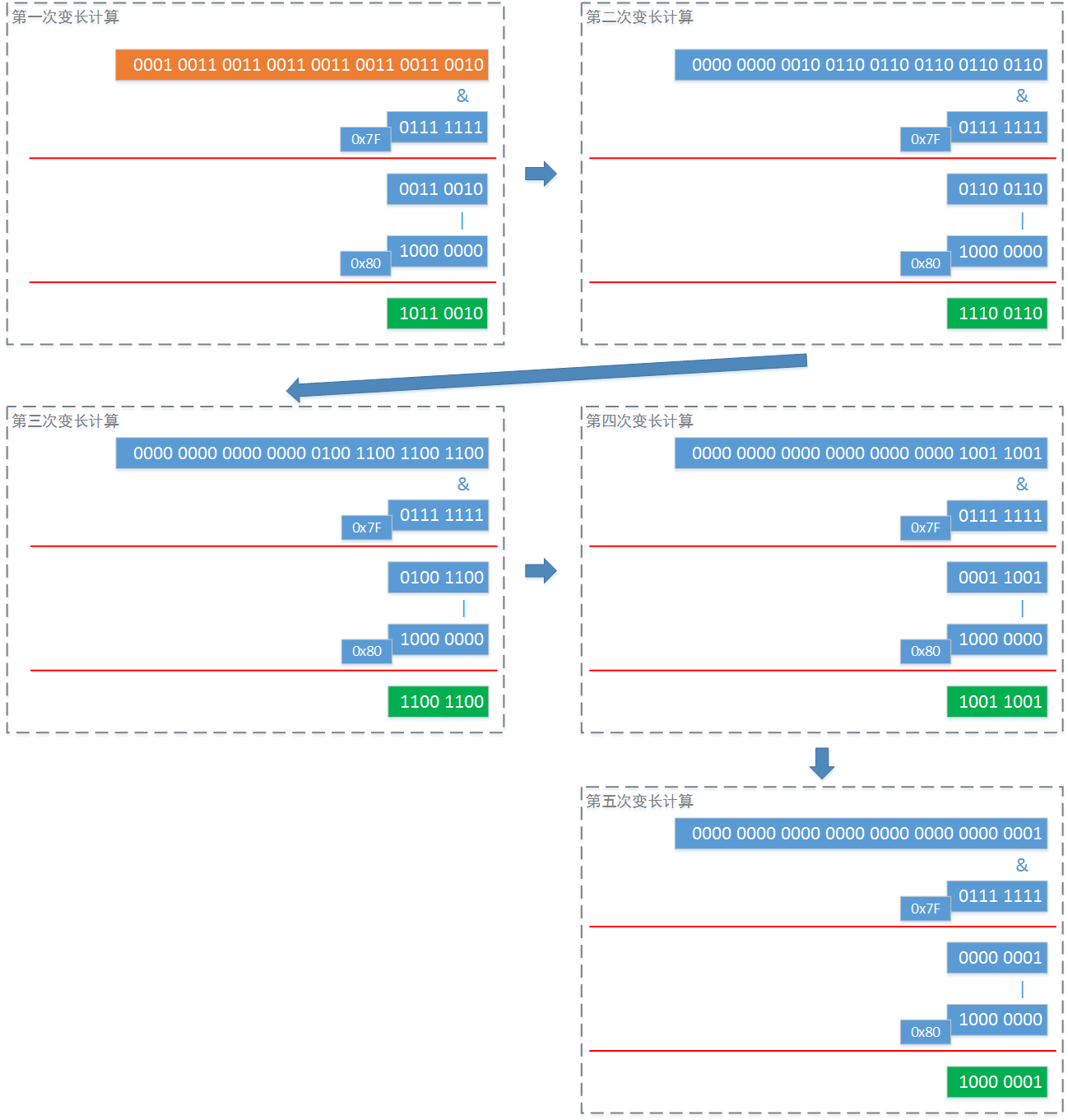
}

以上代码片段一共有一个对外的调用方法，和两个分别名为intToZigZag和writeVarint32的私有方法。从字面上的意义我们可以知道：当对一个32位整数进行编码时，首先将这个32位整数转成ZigZag编码格式，然后在序列化为“变长的32位整数”。那么这个处理的具体过程是什么样的呢？我们以一个较大的32位整数（161061273，二进制计数为：1001100110011001100110011001）为例，进行讲解：

* 首先将整个这个整数做成ZigZag编码格式：



然后进行“变长”处理：



* 可以看到，上面的“变长”计算一共进行了5次，比TBinaryProtocol中的32位整数序列化还要多出一个byte。这是为什么呢？因为这个数字比较长。
* 但实际处理中，我们一般使用的数据都是比较小的。这也是为什么首先要使用ZigZag编码把某个负数的符号位从高位移动到低位的原因。实际上，在实际过程中，变长计算一般只会进行二至三次就完成。这样，在大多数情况下，完成一个32位整数的序列化，TCompactProtocol做使用的空间就比TBinaryProtocol要小。
* 那么经过分析，对于TCompactProtocol和TBinaryProtocol的选择的经验是：如果传输的信息中，基本都是字符串，那么使用TCompactProtocol还是使用TBinaryProtocol基本上都是差不多的；如果需要传输的信息中，会有较多的“低位数字”，那么建议使用TCompactProtocol。

### 3-1-3、其他传输格式封装：

当然Apache Thrift还提供其他的传输格式封装。不同的需求场景下，您可以使用根据需要选用这些信息传输格式：

## 3-2、Aapche Thrift与通信模型

Apache Thrift支持阻塞式同步IO通讯模型和非阻塞式异步IO通信模型。这里说明一下，我在这个系列的文章中，已经详细讲述了各种IO模型的特点和工作原理（请参见我另外几篇文章《[架构设计：系统间通信（3）——IO通信模型和JAVA实践 上篇](http://blog.csdn.net/yinwenjie/article/details/48472237)》、《[架构设计：系统间通信（4）——IO通信模型和JAVA实践 中篇](http://blog.csdn.net/yinwenjie/article/details/48522403)》、《[架构设计：系统间通信（5）——IO通信模型和JAVA实践 下篇](http://blog.csdn.net/yinwenjie/article/details/48784375)》）。所以读者您如果度过本人的拙作，那么您一定清楚，要发挥Apache Thrift性能上的优势，那么一定要在正式生产环境中采用Apache Thrift对非阻塞式异步IO通信模型的支持。下面的代码我们将向您展示Apache Thrift的这种特性：

在给出示例代码之前一定要再强调一次，Apache Thrift的服务器端和客户端一定要采用相同的通信模型。这就是说如果Apache Thrift的服务器端采用的是非阻塞异步通信模型，那么Apache Thrift客户端也一定要采用非阻塞异步通信模型，否则就无法通信。

* 服务器的非阻塞异步通信代码：
* package testThrift.man;
* import java.nio.channels.Selector;
* import java.util.concurrent.Executors;
* import org.apache.commons.logging.Log;
* import org.apache.commons.logging.LogFactory;
* import org.apache.log4j.BasicConfigurator;
* import org.apache.thrift.TProcessor;
* import org.apache.thrift.protocol.TBinaryProtocol;
* import org.apache.thrift.server.THsHaServer;
* import org.apache.thrift.transport.TNonblockingServerSocket;
* import testThrift.iface.HelloWorldService;
* import testThrift.iface.HelloWorldService.Iface;
* import testThrift.impl.HelloWorldServiceImpl;
* public class HelloNonServerDemo {
* static {
* BasicConfigurator.configure();
* }
* /\*\*
* \* 日志
* \*/
* private static Log LOGGER = LogFactory.getLog(HelloNonServerDemo.class);
* public static final int SERVER\_PORT = 8090;
* public void startServer() {
* try {
* // log4j日志，如果您工程里面没有加入log4j的支持，请待用system.out
* HelloNonServerDemo.LOGGER.info("HelloWorld TSimpleServer start ....");
* // 服务执行控制器（告诉apache thrift，实现了HelloWorldService.Iface接口的是具体的哪一个类）
* // HelloWorldServiceImpl类的代码，就不在赘述了，无论采用哪种通信模型，它的代码都不会变化
* TProcessor tprocessor = new HelloWorldService.Processor<Iface>(new HelloWorldServiceImpl());
* // 非阻塞异步通讯模型（服务器端）
* TNonblockingServerSocket serverTransport = new TNonblockingServerSocket(HelloNonServerDemo.SERVER\_PORT);
* // Selector这个类，是不是很熟悉。
* serverTransport.registerSelector(Selector.open());
* THsHaServer.Args tArgs = new THsHaServer.Args(serverTransport);
* tArgs.processor(tprocessor);
* // 指定消息的封装格式（采用二进制流封装）
* tArgs.protocolFactory(new TBinaryProtocol.Factory());
* // 指定处理器的所使用的线程池。
* tArgs.executorService(Executors.newFixedThreadPool(100));
* // 启动服务
* THsHaServer server = new THsHaServer(tArgs);
* server.serve();
* } catch (Exception e) {
* HelloNonServerDemo.LOGGER.error(e);
* }
* }
* /\*\*
* \* @param args
* \*/
* public static void main(String[] args) {
* HelloNonServerDemo server = new HelloNonServerDemo();
* server.startServer();
* }
* }
* 客户端的非阻塞异步通信代码：
* package testThrift.client;
* import org.apache.commons.logging.Log;
* import org.apache.commons.logging.LogFactory;
* import org.apache.log4j.BasicConfigurator;
* import org.apache.thrift.TException;
* import org.apache.thrift.async.AsyncMethodCallback;
* import org.apache.thrift.async.TAsyncClientManager;
* import org.apache.thrift.protocol.TBinaryProtocol;
* import org.apache.thrift.transport.TNonblockingSocket;
* import testThrift.iface.HelloWorldService;
* import testThrift.iface.Reponse;
* import testThrift.iface.HelloWorldService.AsyncClient;
* import testThrift.iface.HelloWorldService.AsyncClient.send\_call;
* import testThrift.iface.Request;
* public class HelloNonClient {
* static {
* BasicConfigurator.configure();
* }
* /\*\*
* \* 日志
* \*/
* private static final Log LOGGER = LogFactory.getLog(HelloNonClient.class);
* private static Object WAITOBJECT = new Object();
* public static final void main(String[] args) throws Exception {
* TNonblockingSocket transport = new TNonblockingSocket("127.0.0.1", 8090);
* TAsyncClientManager clientManager = new TAsyncClientManager();
* // 准备调用参数(这个testThrift.iface.Request，是我们通过IDL定义，并且生成的)
* Request request = new Request("{\"param\":\"field1\"}", "\\mySerivce\\queryService");
* // 这是客户端对非阻塞异步网络通信方式的支持。
* // 注意使用的消息封装格式，一定要和服务器端使用的一致
* HelloWorldService.AsyncClient asyncClient =
* new HelloWorldService.AsyncClient.Factory(clientManager, new TBinaryProtocol.Factory()).getAsyncClient(transport);
* // 既然是非阻塞异步模式，所以客户端一定是通过“事件回调”方式，接收到服务器的响应通知的
* asyncClient.send(request,new AsyncMethodCallback<AsyncClient.send\_call>() {
* /\*\*
* \* 当服务器正确响应了客户端的请求后，这个事件被触发
* \*/
* @Override
* public void onComplete(send\_call call) {
* Reponse response = null;
* try {
* response = call.getResult();
* } catch (TException e) {
* HelloNonClient.LOGGER.error(e);
* return;
* }
* HelloNonClient.LOGGER.info("response = " + response);
* }
* /\*\*
* \* 当服务器没有正确响应了客户端的请求，或者其中过程中出现了不可控制的情况。
* \* 那么这个事件会被触发
* \*/
* @Override
* public void onError(Exception exception) {
* HelloNonClient.LOGGER.info("exception = " + exception);
* }
* });
* //这段代码保证客户端在得到服务器回复前，应用程序本身不会终止
* synchronized (HelloNonClient.WAITOBJECT) {
* HelloNonClient.WAITOBJECT.wait();
* }
* }
* }

以上代码是可以直接工作的。读者可以直接在自己的工程中执行。运行的结果和Apache Thrift上一节中Apache Thrift阻塞模式下的运行结果是一致的，只是运行过程不一样。目前各种主流的RPC框架基本都支持非阻塞式异步IO网络通信，如果您有兴趣进行这些RPC框架的性能比较，一定要在相同的IO通信模型下进行。

## 3-3、Aapche Thrift与线程池

在之前的文章（《[架构设计：系统间通信（10）——RPC的基本概念](http://blog.csdn.net/yinwenjie/article/details/49453303)》），我们已经提到影响一款RPC框架性能的主要指标。除了RPC框架实现的数据封装格式、RPC框架支持的网络通信模型外，还有一个重要的指标就是它如何执行客户端的请求。

在Apache Thrift中，它使用线程池技术运行具体的接口实现，响应客户端请求（无论Apahce Thrift使用哪种数据封装格式、使用哪种网络通信模型）。

org.apache.thrift.server.THsHaServer.Args.executorService(ExecutorService executorService)

* 1

可以看到，实际上Apache Thrift中设置线程池的方法，所要求的参数类型是java.util.concurrent.ExecutorService接口，也就是说只要实现了ExecutorService接口的类都可以被传入。一般我们常使用的是java.util.concurrent.ThreadPoolExecutor这个类。

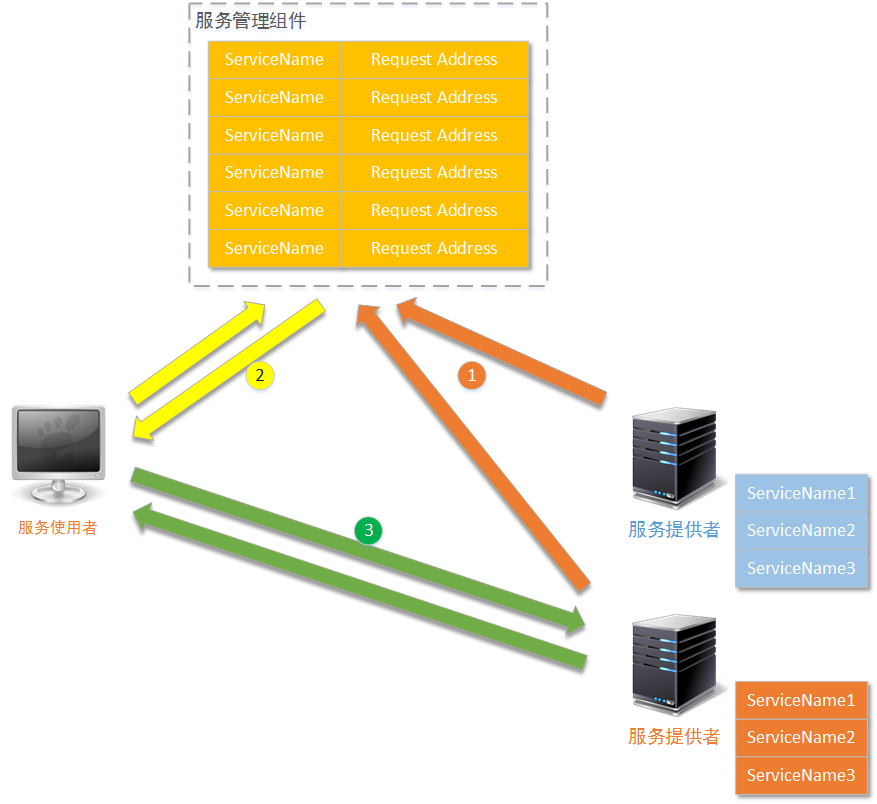
# 1、服务治理

通过前面两篇文章（《[架构设计：系统间通信（12）——RPC实例Apache Thrift 中篇](http://blog.csdn.net/yinwenjie/article/details/49685763)》、《[架构设计：系统间通信（11）——RPC实例Apache Thrift 上篇](http://blog.csdn.net/yinwenjie/article/details/49620623)》）的介绍，相信读者已经可以将Apache Thrift应用到实际工作中，并且理解了为什么Apache Thrift的性能要比大多数RPC框架优秀。但如果您使用过Apache thrift，那么相信您会发现它的一些不足（或者说是所有单纯的RPC框架的不足）：

* 由于Apache Thrift使用IDL定义RCP 调用接口，实现跨语言性。那么一旦当业务发生变化后，是否要重新编写IDL，重新生成接口代码呢？
* 如果以上的事实成立，那如果在生成环境使用了多种语言，且服务节点又很多的情况下。岂不是重新部署的工作量会很大？
* 另外，生产环境的服务是不能停机的？那么就会出现一部分接口是新部署的，另外一部分接口是还未更新的。服务者怎么保证接口的稳定呢？
* 再说，我的生产环境下一共有20个相对独立运行的系统：计费系统、客户系统、订单系统、库存系统、物流系统、税务联动系统，等等。负责他们的开发团队都是不一样的。如何在某个系统的接口发生变动后，通知到其它系统“我的接口变动了”？即便是不能通知到所有系统“我的接口变动了”，又如何做到之前的接口也一样可以使用呢？

显然以上这些问题，单纯使用Apache Thrift（或者单纯的某一款RPC框架）是无法解决的；使用人工的方式就更不要想解决了。如果您的相关系统只有2-3个，又或者每个系统的服务节点数量也不多（例如5、6个），那么以上这些问题还不太明显。但是随着您的系统越来越大，系统间协作越来越复杂，那么这些问题就会凸现出来，甚至成为影响您架构扩容的显著问题。

解决这个问题的方式，**阿里的做法是在众多系统的RPC通信的上层再架一层专门进行RPC通信的协调管理，称之为服务治理框架**（DUBBO框架，目前这个框架已经开源，在后面的文章中，我会花比较大的篇幅进行介绍。和DUBBO框架类似的还有Taobao的HSF）。事实上现在的软件架构中，都是使用相似的“服务治理”思想，来解决这个问题的。如下图所示：



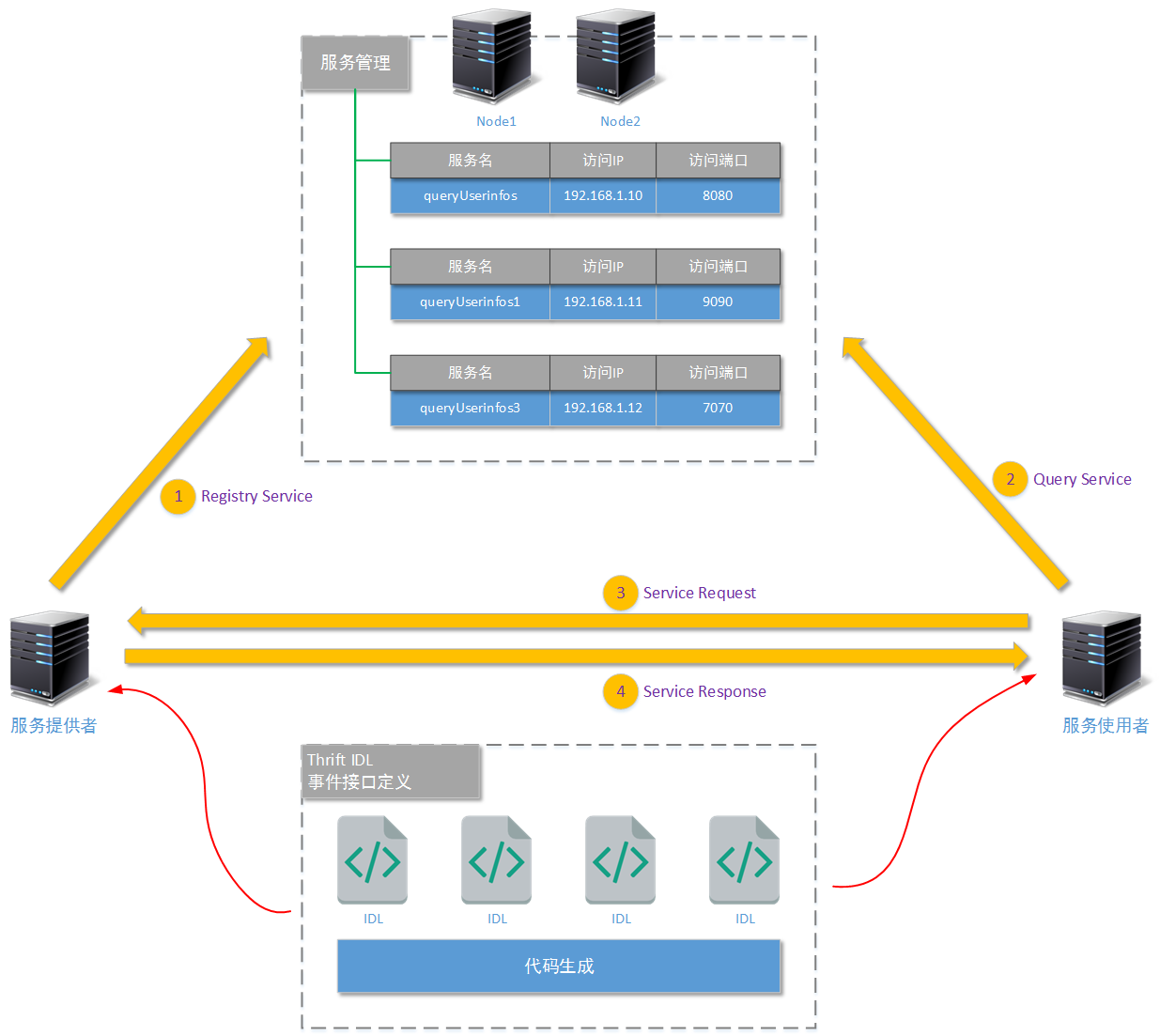
1. 当服务提供者能够向外部系统提供调用服务时（无论这个调用服务是基于RPC的还是基于Http的，一般来说前者居多），它会首先向“服务管理组件”注册这个服务，包括服务名、访问权限、优先级、版本、参数、真实访路径、有效时间等等基本信息。
2. 当某一个服务使用者需要调用服务时，首先会向“服务管理组件”询问服务的基本信息。当然“服务管理组件”还会验证服务使用者是否有权限进行调用、是否符合调用的前置条件等等过滤。最终“服务管理组件”将真实的服务提供者所在位置返回给服务使用者。
3. 服务使用者拿到真实服务提供者的基本信息、调用权限后，再向真实的服务提供者发出调用请求，进行正式的业务调用过程。

在服务治理的思想中，包含几个重要元素：

* 服务管理组件：这个组件是“服务治理”的核心组件，您的服务治理框架有多强大，主要取决于您的服务管理组件功能有多强大。它至少具有的功能包括：服务注册管理、访问路由；另外，它还可以具有：服务版本管理、服务优先级管理、访问权限管理、请求数量限制、连通性管理、注册服务集群、节点容错、事件订阅-发布、状态监控，等等功能。
* 服务提供者（服务生产者）：即服务的具体实现，然后按照服务治理框架特定的规范发布到服务管理组件中。这意味着什么呢？这意味着，服务提供者不一定按照RPC调用的方式发布服务，而是按照整个服务治理框架所规定的方式进行发布（如果服务治理框架要求服务提供者以RPC调用的形式进行发布，那么服务提供者就必须以RPC调用的形式进行发布；如果服务治理框架要求服务提供者以Http接口的形式进行发布，那么服务提供者就必须以Http接口的形式进行发布，但后者这种情况一般不会出现）。
* 服务使用者（服务消费者）：即调用这个服务的用户，调用者首先到服务管理组件中查询具体的服务所在的位置；服务管理组件收到查询请求后，将向它返回具体的服务所在位置（视服务管理组件功能的不同，还有可能进行这些计算：判断服务调用者是否有权限进行调用、是否需要生成认证标记、是否需要重新检查服务提供者的状态、让调用者使用哪一个服务版本等等）。服务调用者在收到具体的服务位置后，向服务提供者发起正式请求，并且返回相应的结果。第二次调用时，服务请求者就可以像服务提供者直接发起调用请求了（当然，您可以有一个服务提供期限的设置，使用租约协议就可以很好的实现）。

# 2、设计一个服务治理框架

为了更深入理解服务治理框架的作用、工作原理，下面我们就以Apache Thrift为服务治理框架基础技术，来实现一个简单的服务治理框架。为了保证快速实现，我们使用zookeeper作为服务管理组件的基础技术（如果您不清楚zookeeper的相关技术点，可以参考我另外的几篇文章《[hadoop系列：zookeeper（1）——zookeeper单点和集群安装](http://blog.csdn.net/yinwenjie/article/details/47361419)》、《[hadoop系列：zookeeper（2）——zookeeper核心原理（选举）](http://blog.csdn.net/yinwenjie/article/details/47613309)》、《[hadoop系列：zookeeper（3）——zookeeper核心原理（事件）](http://blog.csdn.net/yinwenjie/article/details/47685077)》）。下图为简单的工作原理：



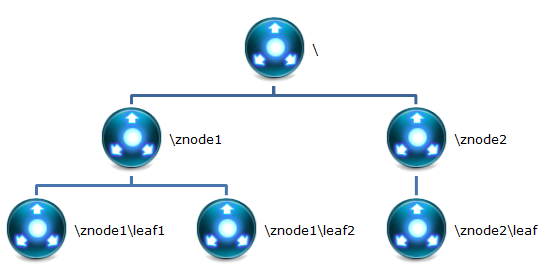
# 2-1、涉及技术

## 2-1-1、使用Zookeeper

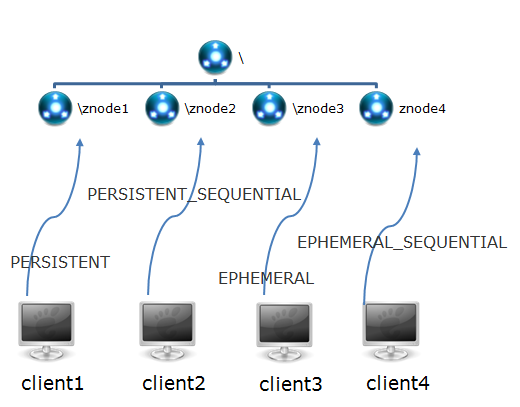
Zookeeper是一个分布式的，开放源码的分布式应用程序协调服务，是Hadoop和Hbase的重要组件。这里我们使用Zookeeper共享“已注册的服务”。为了保证所有服务提供者都能够向Zookeeper注册提供的服务，我们需要在Zookeeper上确定一个 服务提供者和服务使用者 协商一致的“服务描述格式”。

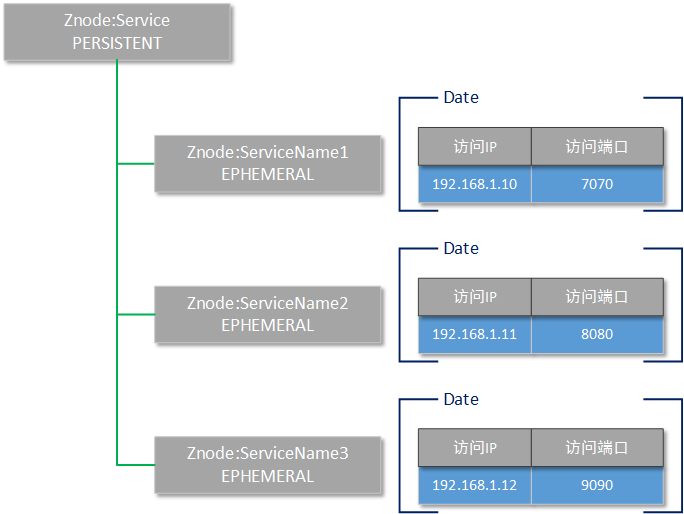
要设计这个“服务描述格式”，首先就要清楚Zookeeper是如何记录信息的。由于我在其他文章中，已经详细讲解过Zookeeper的信息记录方式了，所以这里就只进行一些关键要素的讲解：

* Zookeeper采用树型结构目录结构记录信息。树的深度没有限制（但实际中，不可能建立很深的树结构），每一个节点成为znode。
* 每一个znode都有一个名称，为了避免出现字符集编码问题，请不要使用中文作为znode的名称。另外，同一个znode下的子级znode名称，不允许重复。
* 一个znode允许存储最多1MB大小的数据信息。



* znode根据创建性质的不一样，可分为四种行为类型不一样的znode。它们是：PERSISTENT、PERSISTENT\_SEQUENTIAL、EPHEMERAL、EPHEMERAL\_SEQUENTIAL。
* PERSISTENT-持久化节点：创建这个节点的客户端在与zookeeper服务的连接断开后，这个节点也不会被删除（除非您使用API强制删除）。
* PERSISTENT\_SEQUENTIAL-持久化顺序编号节点：当客户端请求创建这个节点A后，zookeeper会根据parent-znode的zxid状态，为这个A节点编写一个全目录唯一的编号（这个编号只会一直增长）。当客户端与zookeeper服务的连接断开后，这个节点也不会被删除。
* EPHEMERAL-临时目录节点：创建这个节点的客户端在与zookeeper服务的连接断开后，这个节点（还有涉及到的子节点）就会被删除。
* EPHEMERAL\_SEQUENTIAL-临时顺序编号目录节点：当客户端请求创建这个节点A后，zookeeper会根据parent-znode的zxid状态，为这个A节点编写一个全目录唯一的编号（这个编号只会一直增长）。当创建这个节点的客户端与zookeeper服务的连接断开后，这个节点被删除



那么按照Zookeeper的这些工作特点，我们对“服务描述格式”的结构进行了如下图所示的设计：

* Zookeeper的根目录名字叫做Service，这是一个持久化的znode节点，并且不需要存储任何数据。
* 当某一个服务提供者启动后，它将连接到Zookeeper集群，并且在Service目录下，创建一个以提供的服务名为znode名称的临时节点（例如上图所示的znode，分别叫做ServiceName1、ServiceName2、ServiceName3）。
* 每一个Service的子级znode都使用JSON格式存储两个信息，分别是这个服务的真实访问路径和访问端口。
* 这样一来，当某一个服务提供者由于某些原因不能再提供服务，并且断掉和zookeeper的连接后，它所注册的服务就会消失。通过zookeeper的通知机制（或者等待客户端的下一次询问），客户端就会知道已经没有某一个服务了。
* 对于服务调用者（服务使用者）而言，实际上并不是每一次调用服务前，都需要请求zookeeper询问访问地址。而是只需要询问一次，如果找到相关的服务，则记录到本地；待到下一次请求时，直接寻找本地的历史记录即可。

## 2-1-2、使用Apache Thrift

Apache Thrift的基本使用这里就不再赘述了，如果您对Apache Thrift的基本使用还不清楚，请查看前文。对于Apache Thrift的使用，在我们这个自行设计的服务治理框架中，要解决的重要问题，就是保证做到新增一个服务时，不需要重新改变IDL定义，不需要重新生成代码。

这个问题主要的解决思路就是将Apache Thrift的接口定义进行泛化，即这个接口不调用具体的业务，而只给出调用者需要调用的接口名称（包括参数），然后在服务器端，以反射的进行具体服务的调用。IDL文件进行如下的定义：

# 这个结构体定义了服务调用者的请求信息

struct Request {

# 传递的参数信息，使用格式进行表示

1:required binary paramJSON;

# 服务调用者请求的服务名，使用serviceName属性进行传递

2:required string serviceName

}

# 这个结构体，定义了服务提供者的返回信息

struct Reponse {

# RESCODE 是处理状态代码，是一个枚举类型。例如RESCODE.\_200表示处理成功

1:required RESCODE responeCode;

# 返回的处理结果，同样使用JSON格式进行描述

2:required binary responseJSON;

}

# 异常描述定义，当服务提供者处理过程出现异常时，向服务调用者返回

exception ServiceException {

# EXCCODE 是异常代码，也是一个枚举类型。

# 例如EXCCODE.PARAMNOTFOUND表示需要的请求参数没有找到

1:required EXCCODE exceptionCode;

# 异常的描述信息，使用字符串进行描述

2:required string exceptionMess;

}

# 这个枚举结构，描述各种服务提供者的响应代码

enum RESCODE {

\_200=200;

\_500=500;

\_400=400;

}

# 这个枚举结构，描述各种服务提供者的异常种类

enum EXCCODE {

PARAMNOTFOUND = 2001;

SERVICENOTFOUND = 2002;

}

# 这是经过泛化后的Apache Thrift接口

service DIYFrameworkService {

Reponse send(1:required Request request) throws (1:required ServiceException e);

}

# 2-2、服务提供者设计思路

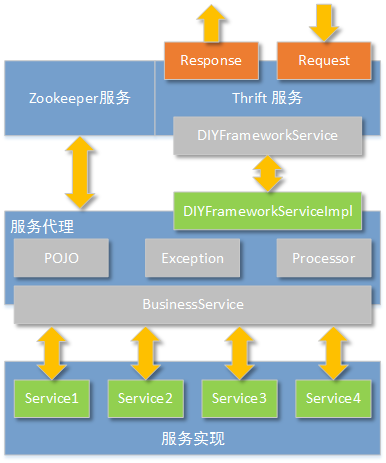
在给出全部示例代码前，首先就要把我们自定制的这个“服务治理”框架的设计思路讲清楚。这样各位读者在看示例代码的时候才不至于看昏过去。上文已经讲过，整个“服务治理”框架主要由四部分构成：基于zookeeper的服务管理器、服务提供者、服务调用者、为跨语言准备的IDL描述。

基于zookeeper的服务管理器，最重要的就是zookeeper中的目录结构如何设计的问题，这个问题在前文中已经讲得比较清楚，无须赘述了；为跨语言准备的IDL描述文件，以及为什么这样设计IDL描述也已经在上文中讲清楚了；那么对于服务调用者来说，最主要的就是两步调用过程：先查询zookeeper服务管理器，找到要调用的服务地址，然后请求具体服务，基本上是比较简单的，无需花很长的篇幅说明设计思路；

那么要说清楚整个“服务治理”框架的设计思路，最主要的还是说清楚服务提供者的设计思路。因为基本上所有业务过程、事件监听调用，都发生在服务提供者这一端。

## 2-2-1、服务提供者设计

下图表达了服务提供者的软件结构设计思路：



从上图可以看到，整个服务端的设计分为三层：

* 最外层由Zookeeper客户端和Apache Thrift服务构成。Zookeeper客户端用于向Zookeeper服务集群注册“提供的服务”；Apache Thrift用于接受服务调用者的请求，并按照格式响应处理结果。
* 由于我们定义的Apache Thrift接口（DIYFrameworkService）已经被泛化，所以具体的业务处理不能由Apache Thrift的实现（DIYFrameworkServiceImpl）来处理。由于这个原因，那么在服务端的设计中，就必须有一个服务代理层，这个服务代理层最重要的功能，就是根据Thrift收到的请求参数，决定调用哪个真实服务（在下文专门介绍具体代码的章节中，还将介绍如何集成spring，对代理层进行优化）。
* 根据软件功能需求的要求，具体的服务实现可以有多个。在设计中我们规定，所有的具体业务实现者，必须实现BusinessService接口中的handle方法。并且返回的类型都必须继承AbstractPojo。

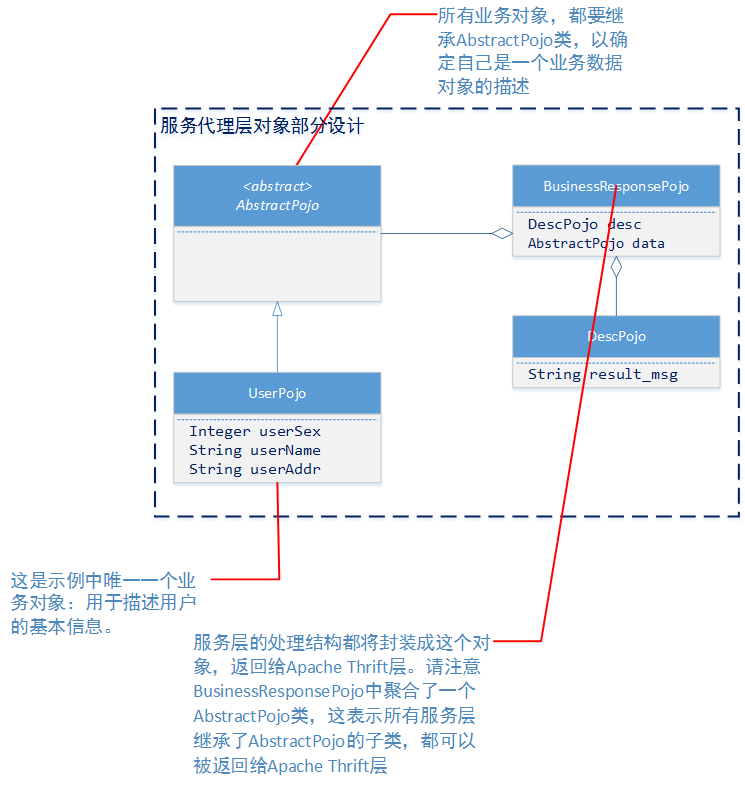
## 2-2-2、功能边界确认

这里我们提供的示例设计，是为了让各位读者了解“服务治理”的基本设计原理。我们目前介绍的示例如果要应用到实际工作中，那么还需要按照读者自己的业务特点进行调整、修改甚至是重新设计。对于这个示例提供的功能来说，我们提供一些简单的，具有代表意义的就可以了：

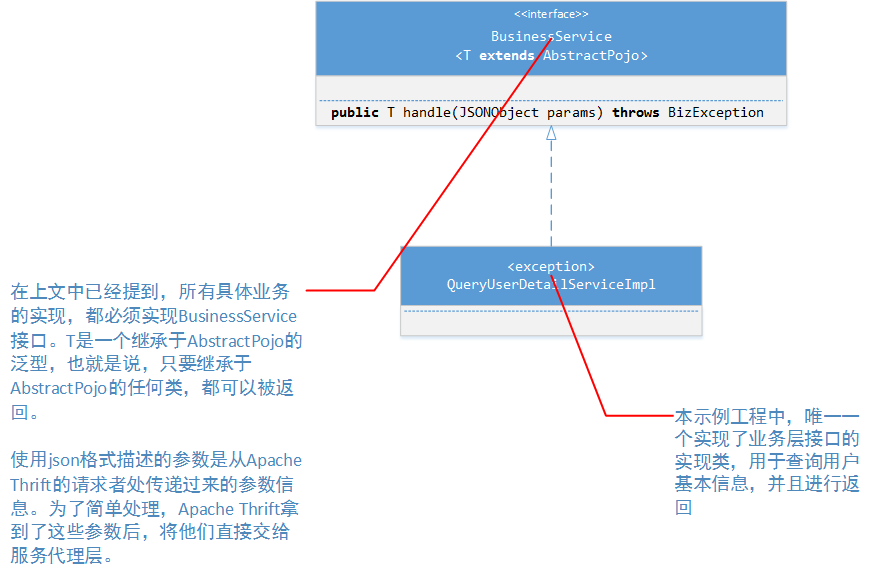
* zookeeper服务：服务提供者的zookeeper客户端只负责连接到zookeeper服务集群，并且向zookeeper服务集群注册“服务提供者所提供的服务”。注册zookeeper时所依据的目录结构见上文中zookeeper目录结构设计的介绍。为了处理简单，zookeeper服务并不考虑性能问题，无需监听zookeeper集群上任何目录结构的变化事件，也无需将远程zookeeper集群上的目录结构缓存到本地。设计的目录结构也无需考虑一个服务由多个服务节点同时提供服务的情况。也无需考虑访问权限、访问优先级的问题。
* Apache Thrift服务：服务提供者的Apache Thrift只负责提供远程RPC调用的监听服务。而且IDL的设计也很简单（参见上文中对IDL定义格式的介绍），只要的开发语言采用JAVA，无需生成多语言的代码。采用阻塞同步的网络通讯模式，无需考虑Apache Thrift的性能问题。
* 服务代理：在正式的生产环境中，实际上服务代理层需要负责的工作是最多的。例如它要对服务请求者的令牌环进行判断，以便确定服务是否过期；要对请求者的权限进行验证；要管理具体的服务实现的注册，以便向zookeeper客户端告知注册情况；要决定具体执行哪一个服务实现，等等工作。但是为了让示例简洁，服务代理层只提供一个简单的注册管理和具体服务实现的调用。
* 服务实现在整个实例代码中，我们只提供一个服务：实现BusinessService服务层接口（business.impl.QueryUserDetailServiceImpl），查询用户详细信息的服务。并且向服务代理层注册这个服务为：”queryUserDetailService” -> “business.impl.QueryUserDetailServiceImpl”

## 2-2-3、建模设计

* 业务层模型设计



* 服务层设计



以上两种类简图和附带的说明，已经把示例工程中重要的设计详情进行了描述。当然工程中还有其他类，但是它们主要还是起辅助作用。例如工具类：JSONUtils、DateUtils；自定义异常：BizException；响应代码：ResponseCode；应用程序启动类：MainProcessor；这些我们将在下文具体代码中进行讲解。

# 3、正式开始编码

我已经在CSDN的资源区上传了这个示例工程的所有代码（<http://download.csdn.net/detail/yinwenjie/9289999>）。读者可以直接到资源下载站进行下载（不收积分哦~~^\_^）。这篇文章将紧接上文，主要介绍这个工程几个主要的类代码。

# 3-1、编写服务端主程序

服务端主程序的类名：processor.MainProcessor，它负责在服务端启动Apache Thrift并且在服务监听启动成功后，连接到zookeeper，注册这个服务的基本信息。

这里要注意一下，Apache Thrift的服务监听是阻塞式的，**所以processor.MainProcessor的Apache Thrift操作应该另起线程进行（processor.MainProcessor.StartServerThread），并且通过线程间的锁定操作，保证zookeeper的连接一定是在Apache Thrift成功启动后才进行**。

package processor;

import java.io.IOException;

import java.util.Set;

import java.util.concurrent.Executors;

import net.sf.json.JSONObject;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.apache.log4j.BasicConfigurator;

import org.apache.thrift.TProcessor;

import org.apache.thrift.protocol.TBinaryProtocol;

import org.apache.thrift.protocol.TProtocol;

import org.apache.thrift.server.ServerContext;

import org.apache.thrift.server.TServerEventHandler;

import org.apache.thrift.server.TThreadPoolServer;

import org.apache.thrift.server.TThreadPoolServer.Args;

import org.apache.thrift.transport.TServerSocket;

import org.apache.thrift.transport.TTransport;

import org.apache.thrift.transport.TTransportException;

import org.apache.zookeeper.CreateMode;

import org.apache.zookeeper.KeeperException;

import org.apache.zookeeper.WatchedEvent;

import org.apache.zookeeper.Watcher;

import org.apache.zookeeper.ZooKeeper;

import org.apache.zookeeper.ZooDefs.Ids;

import org.apache.zookeeper.data.Stat;

import business.BusinessServicesMapping;

import thrift.iface.DIYFrameworkService;

import thrift.iface.DIYFrameworkService.Iface;

public class MainProcessor {

static {

BasicConfigurator.configure();

}

/\*\*

\* 日志

\*/

private static final Log LOGGER = LogFactory.getLog(MainProcessor.class);

private static final Integer SERVER\_PORT = 8090;

/\*\*

\* 专门用于锁定以保证这个主线程不退出的一个object对象

\*/

private static final Object WAIT\_OBJECT = new Object();

/\*\*

\* 标记apache thrift是否启动成功了

\* 只有apache thrift启动成功了，才需要连接到zk

\*/

private boolean isthriftStart = false;

public static void main(String[] args) {

/\*

\* 主程序要做的事情：

\*

\* 1、启动thrift服务。并且服务调用者的请求

\* 2、连接到zk，并向zk注册自己提供的服务名称，告知zk真实的访问地址、访问端口

\* （向zk注册的服务，存储在BusinessServicesMapping这个类的K-V常量中）

\* \*/

//1、========启动thrift服务

MainProcessor mainProcessor = new MainProcessor();

mainProcessor.startServer();

// 一直等待，apache thrift启动完成

synchronized (mainProcessor) {

try {

while(!mainProcessor.isthriftStart) {

mainProcessor.wait();

}

} catch (InterruptedException e) {

MainProcessor.LOGGER.error(e);

System.exit(-1);

}

}

//2、========连接到zk

try {

mainProcessor.connectZk();

} catch (IOException | KeeperException | InterruptedException e) {

MainProcessor.LOGGER.error(e);

System.exit(-1);

}

// 这个wait在业务层面，没有任何意义。只是为了保证这个守护线程不会退出

synchronized (MainProcessor.WAIT\_OBJECT) {

try {

MainProcessor.WAIT\_OBJECT.wait();

} catch (InterruptedException e) {

MainProcessor.LOGGER.error(e);

System.exit(-1);

}

}

}

/\*\*

\* 这个私有方法用于连接到zk上，并且注册相关服务

\* @throws IOException

\* @throws InterruptedException

\* @throws KeeperException

\*/

private void connectZk() throws IOException, KeeperException, InterruptedException {

// 读取这个服务提供者，需要在zk上注册的服务

Set<String> serviceNames = BusinessServicesMapping.SERVICES\_MAPPING.keySet();

// 如果没有任何服务需要注册到zk，那么这个服务提供者就没有继续注册的必要了

if(serviceNames == null || serviceNames.isEmpty()) {

return;

}

// 默认的监听器

MyDefaultWatcher defaultWatcher = new MyDefaultWatcher();

// 连接到zk服务器集群，添加默认的watcher监听

ZooKeeper zk = new ZooKeeper("192.168.61.128:2181", 120000, defaultWatcher);

//创建一个父级节点Service

Stat pathStat = null;

try {

pathStat = zk.exists("/Service", defaultWatcher);

//如果条件成立，说明节点不存在（只需要判断一个节点的存在性即可）

//创建的这个节点是一个“永久状态”的节点

if(pathStat == null) {

zk.create("/Service", "".getBytes(), Ids.OPEN\_ACL\_UNSAFE, CreateMode.PERSISTENT);

}

} catch(Exception e) {

System.exit(-1);

}

// 开始添加子级节点，每一个子级节点都表示一个这个服务提供者提供的业务服务

for (String serviceName : serviceNames) {

JSONObject nodeData = new JSONObject();

nodeData.put("ip", "127.0.0.1");

nodeData.put("port", MainProcessor.SERVER\_PORT);

zk.create("/Service/" + serviceName, nodeData.toString().getBytes(), Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL);

}

//执行到这里，说明所有的service都启动完成了

MainProcessor.LOGGER.info("===================所有service都启动完成了，主线程开始启动===================");

}

/\*\*

\* 这个私有方法用于开启Apache thrift服务端，并进行持续监听

\* @throws TTransportException

\*/

private void startServer() {

Thread startServerThread = new Thread(new StartServerThread());

startServerThread.start();

}

private class StartServerThread implements Runnable {

@Override

public void run() {

MainProcessor.LOGGER.info("看到这句就说明thrift服务端准备工作 ....");

// 服务执行控制器（只要是调度服务的具体实现该如何运行）

TProcessor tprocessor = new DIYFrameworkService.Processor<Iface>(new DIYFrameworkServiceImpl());

// 基于阻塞式同步IO模型的Thrift服务，正式生产环境不建议用这个

TServerSocket serverTransport = null;

try {

serverTransport = new TServerSocket(MainProcessor.SERVER\_PORT);

} catch (TTransportException e) {

MainProcessor.LOGGER.error(e);

System.exit(-1);

}

// 为这个服务器设置对应的IO网络模型、设置使用的消息格式封装、设置线程池参数

Args tArgs = new Args(serverTransport);

tArgs.processor(tprocessor);

tArgs.protocolFactory(new TBinaryProtocol.Factory());

tArgs.executorService(Executors.newFixedThreadPool(100));

// 启动这个thrift服务

TThreadPoolServer server = new TThreadPoolServer(tArgs);

server.setServerEventHandler(new StartServerEventHandler());

server.serve();

}

}

/\*\*

\* 为这个TThreadPoolServer对象，设置是一个事件处理器。

\* 以便在TThreadPoolServer正式开始监听服务请求前，通知mainProcessor：

\* “Apache Thrift已经成功启动了”

\* @author yinwenjie

\*

\*/

private class StartServerEventHandler implements TServerEventHandler {

@Override

public void preServe() {

/\*

\* 需要实现这个方法，以便在服务启动成功后，

\* 通知mainProcessor： “Apache Thrift已经成功启动了”

\* \*/

MainProcessor.this.isthriftStart = true;

synchronized (MainProcessor.this) {

MainProcessor.this.notify();

}

}

/\* (non-Javadoc)

\* @see org.apache.thrift.server.TServerEventHandler#createContext(org.apache.thrift.protocol.TProtocol, org.apache.thrift.protocol.TProtocol)

\*/

@Override

public ServerContext createContext(TProtocol input, TProtocol output) {

/\*

\* 无需实现

\* \*/

return null;

}

@Override

public void deleteContext(ServerContext serverContext, TProtocol input, TProtocol output) {

/\*

\* 无需实现

\* \*/

}

@Override

public void processContext(ServerContext serverContext, TTransport inputTransport, TTransport outputTransport) {

/\*

\* 无需实现

\* \*/

}

}

/\*\*

\* 这是默认的watcher，什么也没有，也不需要有什么<br>

\* 因为按照功能需求，服务器端并不需要监控zk上的任何目录变化事件

\* @author yinwenjie

\*/

private class MyDefaultWatcher implements Watcher {

public void process(WatchedEvent event) {

}

}

}

# 3-2、编写服务具体实现

服务端具体实现的代码很简单，就是在IDL脚本生成了java代码后，对DIYFrameworkService接口进行的实现。

package processor;

import java.nio.ByteBuffer;

import net.sf.json.JSONObject;

import org.apache.commons.lang.StringUtils;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.apache.thrift.TException;

import business.BusinessService;

import business.BusinessServicesMapping;

import business.exception.BizException;

import business.exception.ResponseCode;

import business.pojo.AbstractPojo;

import business.pojo.BusinessResponsePojo;

import business.pojo.DescPojo;

import thrift.iface.DIYFrameworkService.Iface;

import thrift.iface.EXCCODE;

import thrift.iface.RESCODE;

import thrift.iface.Reponse;

import thrift.iface.Request;

import thrift.iface.ServiceException;

import utils.JSONUtils;

/\*\*

\* IDL文件中，我们定义的唯一服务接口DIYFrameworkService.Iface的唯一实现

\* @author yinwenjie

\*

\*/

public class DIYFrameworkServiceImpl implements Iface {

/\*\*

\* 日志

\*/

public static final Log LOGGER = LogFactory.getLog(DIYFrameworkServiceImpl.class);

/\* (non-Javadoc)

\* @see thrift.iface.DIYFrameworkService.Iface#send(thrift.iface.Request)

\*/

@SuppressWarnings("unchecked")

@Override

public Reponse send(Request request) throws ServiceException, TException {

/\*

\* 由于MainProcessor中，在Apache Thrift 服务端启动时已经加入了线程池，所以这里就不需要再使用线程池了

\* 这个服务方法的实现，需要做以下事情：

\*

\* 1、根据request中，描述的具体服务名称，在配置信息中查找具体的服务类

\* 2、使用java的反射机制，调用具体的服务类(BusinessService接口的实现类)。

\* 3、根据具体的业务处理结构，构造Reponse对象，并进行返回

\* \*/

//1、===================

String serviceName = request.getServiceName();

String className = BusinessServicesMapping.SERVICES\_MAPPING.get(serviceName);

//未发现服务

if(StringUtils.isEmpty(className)) {

return this.buildErrorReponse("无效的服务" , null);

}

//2、===================

// 首先得到以json为描述格式的请求参数信息

JSONObject paramJSON = null;

try {

byte [] paramJSON\_bytes = request.getParamJSON();

if(paramJSON\_bytes != null && paramJSON\_bytes.length > 0) {

String paramJSON\_string = new String(paramJSON\_bytes);

paramJSON = JSONObject.fromObject(paramJSON\_string);

}

} catch(Exception e) {

DIYFrameworkServiceImpl.LOGGER.error(e);

// 向调用者抛出异常

throw new ServiceException(EXCCODE.PARAMNOTFOUND, e.getMessage());

}

// 试图进行反射

BusinessService<AbstractPojo> businessServiceInstance = null;

try {

businessServiceInstance = (BusinessService<AbstractPojo>)Class.forName(className).newInstance();

} catch (Exception e) {

DIYFrameworkServiceImpl.LOGGER.error(e);

// 向调用者抛出异常

throw new ServiceException(EXCCODE.SERVICENOTFOUND, e.getMessage());

}

// 进行调用

AbstractPojo returnPojo = null;

try {

returnPojo = businessServiceInstance.handle(paramJSON);

} catch (BizException e) {

DIYFrameworkServiceImpl.LOGGER.error(e);

return this.buildErrorReponse(e.getMessage() , e.getResponseCode());

}

// 构造处理成功情况下的返回信息

BusinessResponsePojo responsePojo = new BusinessResponsePojo();

responsePojo.setData(returnPojo);

DescPojo descPojo = new DescPojo("", ResponseCode.\_200);

responsePojo.setDesc(descPojo);

// 生成json

String returnString = JSONUtils.toString(responsePojo);

byte[] returnBytes = returnString.getBytes();

ByteBuffer returnByteBuffer = ByteBuffer.allocate(returnBytes.length);

returnByteBuffer.put(returnBytes);

returnByteBuffer.flip();

// 构造response

Reponse reponse = new Reponse(RESCODE.\_200, returnByteBuffer);

return reponse;

}

/\*\*

\* 这个私有方法，用于构造“Thrift中错误的返回信息”

\* @param erroe\_mess

\* @return

\*/

private Reponse buildErrorReponse(String erroe\_mess , ResponseCode responseCode) {

// 构造返回信息

BusinessResponsePojo responsePojo = new BusinessResponsePojo();

responsePojo.setData(null);

DescPojo descPojo = new DescPojo(erroe\_mess, responseCode == null?ResponseCode.\_504:responseCode);

responsePojo.setDesc(descPojo);

// 存储byteBuffer；

String responseJSON = JSONUtils.toString(responsePojo);

byte[] responseJSON\_bytes = responseJSON.getBytes();

ByteBuffer byteBuffer = ByteBuffer.allocate(responseJSON\_bytes.length);

byteBuffer.put(byteBuffer);

byteBuffer.flip();

Reponse reponse = new Reponse(RESCODE.\_500, byteBuffer);

return reponse;

}

}

# 3-3、编写客户端实现

在上文中已经介绍过了，客户端有两件事情需要做：连接到zookeeper查询注册的服务该如何访问；然后向真实的服务提供者发起请求。代码如下：

package client;

import java.nio.ByteBuffer;

import java.util.List;

import net.sf.json.JSONObject;

import org.apache.commons.lang.StringUtils;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.apache.log4j.BasicConfigurator;

import org.apache.thrift.protocol.TBinaryProtocol;

import org.apache.thrift.protocol.TProtocol;

import org.apache.thrift.transport.TSocket;

import org.apache.zookeeper.CreateMode;

import org.apache.zookeeper.WatchedEvent;

import org.apache.zookeeper.Watcher;

import org.apache.zookeeper.ZooKeeper;

import org.apache.zookeeper.ZooDefs.Ids;

import org.apache.zookeeper.data.Stat;

import thrift.iface.DIYFrameworkService.Client;

import thrift.iface.Reponse;

import thrift.iface.Request;

import utils.JSONUtils;

public class ThriftClient {

/\*\*

\* 日志

\*/

private static final Log LOGGER = LogFactory.getLog(ThriftClient.class);

private static final String SERVCENAME = "queryUserDetailService";

static {

BasicConfigurator.configure();

}

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

/\*

\* 服务治理框架的客户端示例，要做以下事情：

\*

\* 1、连接到zk，查询当前zk下提供的服务列表中是否有自己需要的服务名称（queryUserDetailService）

\* 2、如果没有找到需要的服务名称，则客户端终止工作

\* 3、如果找到了服务，则通过服务给出的ip，port，基于Thrift进行正式请求

\* （这时，和zookeeper是否断开，关系就不大了）

\* \*/

// 1、===========================

// 默认的监听器

ClientDefaultWatcher defaultWatcher = new ClientDefaultWatcher();

// 连接到zk服务器集群，添加默认的watcher监听

ZooKeeper zk = new ZooKeeper("192.168.61.128:2181", 120000, defaultWatcher);

/\*

\* 为什么客户端连接上来以后，也可能创建一个Service根目录呢？

\* 因为正式的环境下，不能保证客户端一点就在服务器端全部准备好的情况下，再来做调用请求

\* \*/

Stat pathStat = null;

try {

pathStat = zk.exists("/Service", defaultWatcher);

//如果条件成立，说明节点不存在（只需要判断一个节点的存在性即可）

//创建的这个节点是一个“永久状态”的节点

if(pathStat == null) {

zk.create("/Service", "".getBytes(), Ids.OPEN\_ACL\_UNSAFE, CreateMode.PERSISTENT);

}

} catch(Exception e) {

System.exit(-1);

}

// 2、===========================

//获取服务列表(不需要做任何的事件监听，所以第二个参数可以为false)

List<String> serviceList = zk.getChildren("/Service", false);

if(serviceList == null || serviceList.isEmpty()) {

ThriftClient.LOGGER.info("未发现相关服务，客户端退出");

return;

}

//然后查看要找寻的服务是否在存在

boolean isFound = false;

byte[] data;

for (String serviceName : serviceList) {

if(StringUtils.equals(serviceName, ThriftClient.SERVCENAME)) {

isFound = true;

break;

}

}

if(!isFound) {

ThriftClient.LOGGER.info("未发现相关服务，客户端退出");

return;

} else {

data = zk.getData("/Service/" + ThriftClient.SERVCENAME, false, null);

}

/\*

\* 执行到这里，zk的工作就完成了，接下来zk是否断开，就不重要了

\* \*/

zk.close();

if(data == null || data.length == 0) {

ThriftClient.LOGGER.info("未发现有效的服务端地址，客户端退出");

return;

}

// 得到服务器地值说明

JSONObject serverTargetJSON = null;

String serverIp;

String serverPort;

try {

serverTargetJSON = JSONObject.fromObject(new String(data));

serverIp = serverTargetJSON.getString("ip");

serverPort = serverTargetJSON.getString("port");

} catch(Exception e) {

ThriftClient.LOGGER.error(e);

return;

}

//3、===========================

TSocket transport = new TSocket(serverIp, Integer.parseInt(serverPort));

TProtocol protocol = new TBinaryProtocol(transport);

// 准备调用参数

JSONObject jsonParam = new JSONObject();

jsonParam.put("username", "yinwenjie");

byte[] params = jsonParam.toString().getBytes();

ByteBuffer buffer = ByteBuffer.allocate(params.length);

buffer.put(params);

buffer.flip();

Request request = new Request(buffer, ThriftClient.SERVCENAME);

// 开始调用

Client client = new Client(protocol);

// 准备传输

transport.open();

// 正式调用接口

Reponse reponse = client.send(request);

byte[] responseBytes = reponse.getResponseJSON();

// 一定要记住关闭

transport.close();

// 将返回信息显示出来

ThriftClient.LOGGER.info("respinse value = " + new String(responseBytes));

}

}

/\*\*

\* 这是默认的watcher，什么也没有，也不需要有什么<br>

\* 因为按照功能需求，客户端并不需要监控zk上的任何目录变化事件

\* @author yinwenjie

\*/

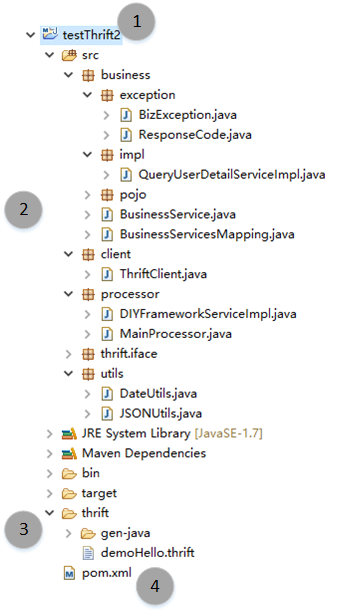
class ClientDefaultWatcher implements Watcher {

public void process(WatchedEvent event) {

}

# 3-4、工程结构说明

以上代码是服务器端、客户端的主要代码。整个工程还有其他的辅助代码，为了让各位读者能够看得清楚直接，我们将整个工程结构进行一下说明，下载后导入的工程结构如下图所示：

}

1. 这是一个典型的JAVA工程。请使用 JDK 1.6+ 版本。我们将讲解整个工程结构。首先来看看这个工程中主要的package和它们的作用。
2. business：具体的业务层逻辑都在这个包里面，其中exception包含了一个业务层异常的定义BizException，还有错误代码ResponseCode；impl包中放置具体的业务层实现，它们都必须实现BusinessService接口；Pojo是业务层对象模型。client：为了简单起见，我将服务端的实现和客户端的实现放置在一个工程中，client这个包就是客户端的实现代码了；utils包放置了两个工具类，用来进行日期格式化的DataUtils和用来进行json转换的JSONUtils。
3. 定义的apache thrift IDL文件放置在thrift文件夹下面，名字叫做：demoHello.thrift；您可以使用它生成各种语言的代码；
4. 工程需要maven的支持。
5. 2016年08月08日，由网友OneZhous发现了一个程序的bug，这是由于Apache Thrift内部并不会在进行org.apache.thrift.TBaseHelper.copyBinary执行时，将java.nio.ByteBuffer自动进行flip()。所以在完成request和response对象设置后，需要开发人员自行进行flip()。感谢OneZhous对文章中的问题进行纠正，但是CSDN由于无法修改已上传的资源，所以还请各位读者在下载运行时注意这个问题：

......

ByteBuffer buffer = ByteBuffer.allocate(params.length);

buffer.put(params);

buffer.flip();

// 以及位置

ByteBuffer byteBuffer = ByteBuffer.allocate(responseJSON\_bytes.length);

byteBuffer.put(byteBuffer);

byteBuffer.flip();

......