thrift使用指导

阿烈

历史：

2013-03 建立文档

2015-09 补充文档

# 目录

[目录 1](#_Toc431111467)

[1. thrift介绍 2](#_Toc431111468)

[2. thrift安装 2](#_Toc431111469)

[1.1. OS X Setup 2](#_Toc431111470)

[1.2. windows 3](#_Toc431111471)

[3. 简单实例 3](#_Toc431111472)

[3.1. 建立login.thrift文件 3](#_Toc431111473)

[3.2. 生成代码 3](#_Toc431111474)

[3.3. 建立项目环境 4](#_Toc431111475)

[3.4. 实现UserService接口 5](#_Toc431111476)

[3.5. 实现服务器端SimpleServerDemo 6](#_Toc431111477)

[3.6. 实现客户端SimpleClientDemo 7](#_Toc431111478)

[3.7. 测试运行 8](#_Toc431111479)

[4. 基础知识 8](#_Toc431111480)

[4.1. 数据类型 8](#_Toc431111481)

[4.1.1. 基本类型 8](#_Toc431111487)

[4.1.2. 结构体类型 9](#_Toc431111488)

[4.1.3. 容器类型 9](#_Toc431111489)

[4.1.4. 异常类型 9](#_Toc431111490)

[4.1.5. 服务类型 9](#_Toc431111491)

[4.2. 注释 10](#_Toc431111492)

[4.3. 枚举类型 10](#_Toc431111493)

[4.4. 命名空间 11](#_Toc431111494)

[4.5. 文件包含 11](#_Toc431111495)

[4.6. 常量 12](#_Toc431111496)

[4.7. 定义结构体 12](#_Toc431111497)

[4.8. 定义服务 12](#_Toc431111498)

[5. 生成代码 13](#_Toc431111499)

[5.1. thrift架构 13](#_Toc431111500)

[5.2. 基本概念 14](#_Toc431111501)

[5.2.1. Protocol 15](#_Toc431111502)

[5.2.2. Transport 17](#_Toc431111503)

[5.2.3. Server 17](#_Toc431111504)

[5.3. 服务端编码基本步骤 18](#_Toc431111505)

[5.3.1. 实现服务处理接口impl 18](#_Toc431111506)

[5.3.2. 创建TProcessor 18](#_Toc431111507)

[5.3.3. 创建TServerTransport 19](#_Toc431111508)

[5.3.4. 创建TProtocol 19](#_Toc431111509)

[5.3.5. 创建TServer 19](#_Toc431111510)

[5.3.6. 启动Server 23](#_Toc431111511)

[5.4. 客户端编码基本步骤 23](#_Toc431111512)

[5.4.1. 创建Transport 23](#_Toc431111513)

[5.4.2. 创建TProtocol 23](#_Toc431111514)

[5.4.3. 基于TTransport和TProtocol创建 Client 23](#_Toc431111515)

[5.4.4. 调用Client的相应方法 23](#_Toc431111516)

[6. 最佳实践 24](#_Toc431111517)

[6.1. 如何选择服务TServer类型? 24](#_Toc431111518)

[TSimpleServer 24](#_Toc431111519)

[TNonblockingServer vs. THsHaServer 25](#_Toc431111520)

[THsHaServer vs. TThreadedSelectorServer 26](#_Toc431111521)

[TThreadedSelectorServer vs. TThreadPoolServer 27](#_Toc431111522)

[6.2. 如何实现服务接口的版本变更? 29](#_Toc431111523)

[7. 参考资料 30](#_Toc431111524)

# thrift介绍

The Apache Thrift software framework, for scalable cross-language services development, combines a software stack with a code generation engine to build services that work efficiently and seamlessly between C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi and other languages.

语言支持程度可参考这篇文章：

<http://wiki.apache.org/thrift/LibraryFeatures?action=show&redirect=LanguageSupport>

# thrift安装

## OS X Setup

直接执行：

brew install thrift

## windows

windows有编译好了的工具，下载即可用。

下载地址：

https://dist.apache.org/repos/dist/release/thrift/0.9.0/thrift-0.9.0.exe

# 简单实例

## 建立login.thrift文件

login.thrift

namespace java com.dianziq.test

struct LoginResponse

{

1:string code

2:string msg

3:string token

}

service UserService

{

LoginResponse login(1:string username,2:string password)

}

## 生成代码

$ thrift -gen java login.thrift

$ tree

.

├── gen-java

│   └── com

│   └── dianziq

│   └── test

│   ├── LoginResponse.java

│   └── UserService.java

└── login.thrift

4 directories, 3 files

## 建立项目环境

本案例使用gradle构建,项目名thriftDemo

build.gradle

apply plugin: 'java'

apply plugin: 'eclipse'

repositories {

mavenCentral()

}

dependencies {

compile 'org.apache.thrift:libthrift:0.9.0'

compile 'org.slf4j:slf4j-log4j12:1.5.8'

}

建立项目基本目录，并将gen-java目录中文件拷到src/main/java中

mkdir -p src/main/{java,resources}

mkdir -p src/test/{java,resources}

生成eclipse工程文件

$ gradle eclipse

:eclipseClasspath

:eclipseJdt

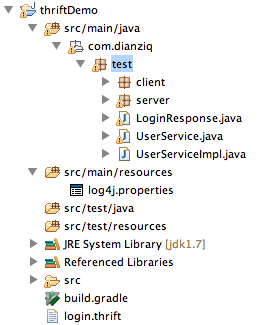
:eclipseProject

:eclipse

BUILD SUCCESSFUL

Total time: 11.441 secs

将项目导入eclipse，最终项目目录显示为



## 实现UserService接口

UserServiceImpl.java

**package** com.dianziq.test;

**import** org.apache.thrift.TException;

**public** **class** UserServiceImpl **implements** UserService.Iface {

@Override

**public** LoginResponse login(String username, String password)

**throws** TException {

LoginResponse response = **new** LoginResponse();

**if** ("tom".equals(username) && "123456".equals(password)) {

response.setCode("0000");

response.setMsg("登录成功");

response.setToken("" + System.*currentTimeMillis*());

} **else** {

response.setCode("1000");

response.setMsg("用户名或密码不正确");

}

**return** response;

}

}

## 实现服务器端SimpleServerDemo

SimpleServerDemo.java

**package** com.dianziq.test.server;

**import** org.apache.thrift.TProcessor;

**import** org.apache.thrift.protocol.TBinaryProtocol;

**import** org.apache.thrift.server.TServer;

**import** org.apache.thrift.server.TSimpleServer;

**import** org.apache.thrift.transport.TServerSocket;

**import** com.dianziq.test.UserService;

**import** com.dianziq.test.UserServiceImpl;

**public** **class** SimpleServerDemo {

**public** **static** **final** **int** *SERVER\_PORT* = 7090;

**public** **void** start() {

**try** {

System.*out*.println("UserServer start ....");

TProcessor tprocessor = **new** UserService.Processor<UserService.Iface>(

**new** UserServiceImpl());

TServerSocket serverTransport = **new** TServerSocket(*SERVER\_PORT*);

TServer.Args tArgs = **new** TServer.Args(serverTransport);

tArgs.processor(tprocessor);

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

TServer server = **new** TSimpleServer(tArgs);

server.serve();

} **catch** (Exception e) {

System.*out*.println("Server start error!!!");

e.printStackTrace();

}

}

/\*\*

\* **@param** args

\*/

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

SimpleServerDemo server = **new** SimpleServerDemo();

server.start();

}

}

## 实现客户端SimpleClientDemo

SimpleClientDemo.java

**package** com.dianziq.test.client;

**import** org.apache.thrift.TException;

**import** org.apache.thrift.protocol.TBinaryProtocol;

**import** org.apache.thrift.protocol.TProtocol;

**import** org.apache.thrift.transport.TSocket;

**import** org.apache.thrift.transport.TTransport;

**import** org.apache.thrift.transport.TTransportException;

**import** com.dianziq.test.LoginResponse;

**import** com.dianziq.test.UserService;

**public** **class** SimpleClientDemo {

**public** **static** **final** String *SERVER\_IP* = "localhost";

**public** **static** **final** **int** *SERVER\_PORT* = 7090;

**public** **static** **final** **int** *TIMEOUT* = 30000;

**public** **void** login(String userName, String passwd) {

TTransport transport = **null**;

**try** {

transport = **new** TSocket(*SERVER\_IP*, *SERVER\_PORT*, *TIMEOUT*);

// 协议要和服务端一致

TProtocol protocol = **new** TBinaryProtocol(transport);

UserService.Client client = **new** UserService.Client(protocol);

transport.open();

LoginResponse loginResponse = client.login(userName, passwd);

System.*out*.println("Thrify client result =: " + loginResponse);

} **catch** (TTransportException e) {

e.printStackTrace();

} **catch** (TException e) {

e.printStackTrace();

} **finally** {

**if** (**null** != transport) {

transport.close();

}

}

}

/\*\*

\* **@param** args

\*/

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

SimpleClientDemo client = **new** SimpleClientDemo();

client.login("tom", "123456");

client.login("mark", "123456");

}

}

## 测试运行

先运行服务器端SimpleServiceDemo

UserServer start ....

运行客户端SimpleClientDemo

Thrify client result =: LoginResponse(code:0000, msg:登录成功, token:1362103521010)

Thrify client result =: LoginResponse(code:1000, msg:用户名或密码不正确, token:null)

# 基础知识

## 数据类型

Thrift类型系统包括预定义的基本类型，用户自定义结构体，容器类型，异常和服务定义



### 基本类型

1. bool：布尔值，true 或 false，对应 Java 的 boolean
2. byte：8 位有符号整数，对应 Java 的 byte
3. i16：16 位有符号整数，对应 Java 的 short
4. i32：32 位有符号整数，对应 Java 的 int
5. i64：64 位有符号整数，对应 Java 的 long
6. double：64 位浮点数，对应 Java 的 double
7. string：未知编码文本或二进制字符串，对应 Java 的 String

### 结构体类型

struct：定义公共的对象，类似于 C 语言中的结构体定义，在 Java 中是一个 JavaBean

### 容器类型

Thrift容器与类型密切相关，它与当前流行编程语言提供的容器类型相对应，采用java泛型风格表示的。Thrift提供了3种容器类型：

List<t1>：一系列t1类型的元素组成的有序表，元素可以重复。对应 Java 的 List。

Set<t1>：一系列t1类型的元素组成的无序表，元素唯一。对应 Java 的 Set。

Map<t1,t2>：key/value对（key的类型是t1且key唯一，value类型是t2）。对应 Java 的 Map。

### 异常类型

exception：对应 Java 的 Exception。

Thrift结构体在概念上同C语言结构体类型—-一种将相关属性聚集（封装）在一起的方式。在面向对象语言中，thrift结构体被转换成类。

异常在语法和功能上类似于结构体，只不过异常使用关键字exception而不是struct关键字声明。但它在语义上不同于结构体，当定义一个RPC服务时，开发者可能需要声明一个远程方法抛出一个异常。

### 服务类型

服务的定义方法在语法上等同于面向对象语言中定义接口。Thrift编译器会产生实现这些接口的client和server桩。

service：对应服务的类

说明：

假设我们定义两个结构体：

struct s1

{

1:bool filed1

2:byte filed2

3:i16 filed3

4:i32 filed4

5:i64 filed5

6:double filed6

7:string filed7

}

struct s2

{

1:list<s1> s1List

2:map<string,string> filedMap

3:set<string> filedSet

}

则生成代码后相应的类型为：

s1

public boolean filed1; // required

public byte filed2; // required

public short filed3; // required

public int filed4; // required

public long filed5; // required

public double filed6; // required

public String filed7; // required

s2

public List<s1> s1List; // required

public Map<String,String> filedMap; // required

public Set<String> filedSet; // required

## 注释

Thrfit支持shell注释风格，C/C++语言中单行或者多行注释风格

# This is a valid comment.

/\*

\* This is a multi-line comment.

\* Just like in C.

\*/

// C++/Java style single-line comments work just as well.

## 枚举类型

可以像C/C++那样定义枚举类型，如：

enum TweetType {

TWEET, //a

RETWEET = 2, //b

DM = 0xa, //c

REPLY

} //d

struct Tweet {

1: required i32 userId;

2: required string userName;

3: required string text;

4: optional Location loc;

5: optional TweetType tweetType = TweetType.TWEET // e

16: optional string language = "english"

}

说明：

a. 编译器默认从0开始赋值

b. 可以赋予某个常量某个整数

c. 允许常量是十六进制整数

d. 末尾没有逗号

e. 给常量赋缺省值时，使用常量的全称

注意，不同于protocol buffer，thrift不支持枚举类嵌套，枚举常量必须是32位的正整数

## 命名空间

Thrift中的命名空间同C++中的namespace和java中的package类似，它们均提供了一种组织（隔离）代码的方式。因为每种语言均有自己的命名空间定义方式（如python中有module），thrift允许开发者针对特定语言定义namespace：

namespace cpp com.example.project // a

namespace java com.example.project // b

说明：

a． 转化成namespace com { namespace example { namespace project {

b． 转换成package com.example.project

## 文件包含

Thrift允许thrift文件包含，用户需要使用thrift文件名作为前缀访问被包含的对象，如：

include "tweet.thrift" // a

...

struct TweetSearchResult {

1: list<tweet.Tweet> tweets; // b

}

说明：

a． thrift文件名要用双引号包含，末尾没有逗号或者分号

b． 注意tweet前缀

## 常量

Thrift允许用户定义常量，复杂的类型和结构体可使用JSON形式表示。

const i32 INT\_CONST = 1234; // a

const map<string,string> MAP\_CONST = {"hello": "world", "goodnight": "moon"}

说明：

a． 分号是可选的，可有可无；支持十六进制赋值。

## 定义结构体

结构体由一系列域组成，每个域有唯一整数标识符，类型，名字和可选的缺省参数组成。如：

struct Tweet {

1: required i32 userId; // a

2: required string userName; // b

3: required string text;

4: optional Location loc; // c

16: optional string language = "english" // d

}

struct Location { // e

1: required double latitude;

2: required double longitude;

}

说明：

a. 每个域有一个唯一的，正整数标识符

b. 每个域可以标识为required或者optional（也可以不注明）

c. 结构体可以包含其他结构体

d. 域可以有缺省值

e. 一个thrift中可定义多个结构体，并存在引用关系

规范的struct定义中的每个域均会使用required或者optional关键字进行标识。如果required标识的域没有赋值，thrift将给予提示。如果optional标识的域没有赋值，该域将不会被序列化传输。如果某个optional标识域有缺省值而用户没有重新赋值，则该域的值一直为缺省值。

与service不同，结构体不支持继承，即，一个结构体不能继承另一个结构体。

## 定义服务

在流行的序列化/反序列化框架（如protocol buffer）中，thrift是少有的提供多语言间RPC服务的框架。

Thrift编译器会根据选择的目标语言为server产生服务接口代码，为client产生桩代码。

//“Twitter”与“{”之间需要有空格！！！

service Twitter {

// 方法定义方式类似于C语言中的方式，它有一个返回值，一系列参数和可选的异常

// 列表. 注意，参数列表和异常列表定义方式与结构体中域定义方式一致.

void ping(), // a

bool postTweet(1:Tweet tweet); // b

TweetSearchResult searchTweets(1:string query); // c

// ”oneway”标识符表示client发出请求后不必等待回复（非阻塞）直接进行下面的操作，

// ”oneway”方法的返回值必须是void

oneway void zip() // d

}

说明：

a． 函数定义可以使用逗号或者分号标识结束

b． 参数可以是基本类型或者结构体，参数是只读的（const），不可以作为返回值！！！

c． 返回值可以是基本类型或者结构体

d． 返回值可以是void

注意，函数中参数列表的定义方式与struct完全一样

Service支持继承，一个service可使用extends关键字继承另一个service

# 生成代码

## thrift架构

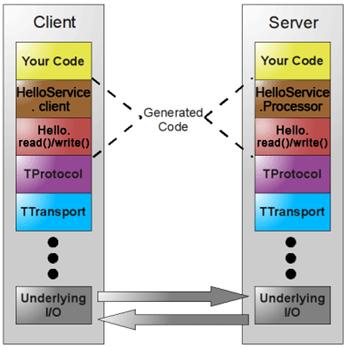
资料来源：

Apache Thrift - 可伸缩的跨语言服务开发框架

<http://www.ibm.com/developerworks/cn/java/j-lo-apachethrift/>

Thrift 包含一个完整的堆栈结构用于构建客户端和服务器端。下图描绘了 Thrift 的整体架构。

**图 1. 架构图**



如图所示，图中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。

Thrift 服务器包含用于绑定协议和传输层的基础架构，它提供阻塞、非阻塞、单线程和多线程的模式运行在服务器上，可以配合服务器 / 容器一起运行，可以和现有的 J2EE 服务器 /Web 容器无缝的结合。

## 基本概念

Thrift中的几个概念：

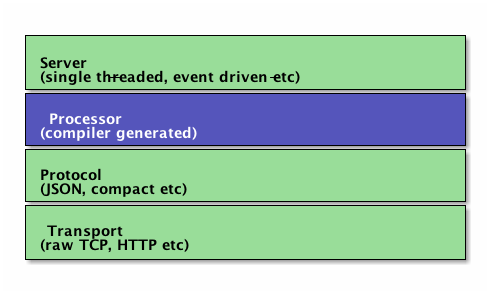
Server 服务模型

Handler 数据处理接口

Processor 数据处理对象

Protocol 数据传输协议

Transport 数据传输方式



Handler为抽象接口，需要在编译后的代码上自行实现。Processor调用Handler中的代码，编译自动生成，不用关心。

### Protocol

Protocol抽象层定义了一种将内存中数据结构映射成可传输格式的机制。换句话说，Protocol定义了datatype怎样使用底层的Transport对自己进行编解码。因此，Protocol的实现要给出编码机制并负责对数据进行序列化。

Protocol接口的定义如下：

writeMessageBegin(name, type, seq)

writeMessageEnd()

writeStructBegin(name)

writeStructEnd()

writeFieldBegin(name, type, id)

writeFieldEnd()

writeFieldStop()

writeMapBegin(ktype, vtype, size)

writeMapEnd()

writeListBegin(etype, size)

writeListEnd()

writeSetBegin(etype, size)

writeSetEnd()

writeBool(bool)

writeByte(byte)

writeI16(i16)

writeI32(i32)

writeI64(i64)

writeDouble(double)

writeString(string)

name, type, seq = readMessageBegin()

readMessageEnd()

name = readStructBegin()

readStructEnd()

name, type, id = readFieldBegin()

readFieldEnd()

k, v, size = readMapBegin()

readMapEnd()

etype, size = readListBegin()

readListEnd()

etype, size = readSetBegin()

readSetEnd()

bool = readBool()

byte = readByte()

i16 = readI16()

i32 = readI32()

i64 = readI64()

double = readDouble()

string = readString()

下面是一些对大部分thrift支持的语言均可用的protocol：

(1) binary：简单的二进制编码

(2) Compact：具体见THRIFT-11

(3) Json

支持的传输格式：

**TBinaryProtocol** – 二进制格式.

**TCompactProtocol** – 压缩格式

#使用 TCompactProtocol 协议构建的 Server

TCompactProtocol.Factory proFactory = new TCompactProtocol.Factory();

#使用 TCompactProtocol 协议的Client

TCompactProtocol protocol = new TCompactProtocol(transport);

**TJSONProtocol** – JSON格式

#使用 TJSONProtocol 协议构建的 Server

TJSONProtocol.Factory proFactory = new TJSONProtocol.Factory();

#使用 TJSONProtocol 协议的 client

TJSONProtocol protocol = new TJSONProtocol(transport);

**TSimpleJSONProtocol** –提供JSON只写协议, 生成的文件很容易通过脚本语言解析。

**TDebugProtocol** – 使用易懂的可读的文本格式，以便于debug

### Transport

Transport层提供了一个简单的网络读写抽象层。这使得thrift底层的transport从系统其它部分（如：序列化/反序列化）解耦。以下是一些Transport接口提供的方法：

open

close

read

write

flush

除了以上几个接口，Thrift使用ServerTransport接口接受或者创建原始transport对象。正如名字暗示的那样，ServerTransport用在server端，为到来的连接创建Transport对象。

open

listen

accept

close

支持的数据传输方式：

**TSocket**

使用阻塞式 I/O 进行传输，是最常见的模式

**TFramedTransport**

使用非阻塞方式，按块的大小进行传输，类似于 Java 中的 NIO

若使用 TFramedTransport 传输层，其服务器必须修改为非阻塞的服务类型，客户端只需替换清单 TTransport 部分， TNonblockingServerTransport 类是构建非阻塞 socket 的抽象类，TNonblockingServerSocket 类继承 TNonblockingServerTransport

**TNonblockingTransport**

使用非阻塞方式，用于构建异步客户端

### Server

Server将以上所有特性集成在一起：

（1） 创建一个transport对象

（2） 为transport对象创建输入输出protocol

（3） 基于输入输出protocol创建processor

（4） 等待连接请求并将之交给processor处理

Server支持的服务模型：

**TSimpleServer**

简单的单线程服务模型，常用于测试

**TThreadedServer**

多线程服务模型，使用阻塞式IO，每个请求创建一个线程。

**TThreadPoolServer**

线程池服务模型，使用标准的阻塞式IO，预先创建一组线程处理请求。

**TNonblockingServer**

多线程服务模型，使用非阻塞式IO（需使用TFramedTransport数据传输方式），(Java 实现使用 NIO channels).

## 服务端编码基本步骤

部分代码：

public static final int SERVER\_PORT = 7090;

public void startServer() {

try {

System.out.println("UserServer start ....");

TProcessor tprocessor = new UserService.Processor<UserService.Iface>(

new UserServiceImpl());

TServerSocket serverTransport = new TServerSocket(SERVER\_PORT);

TServer.Args tArgs = new TServer.Args(serverTransport);

tArgs.processor(tprocessor);

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

TServer server = new TSimpleServer(tArgs);

server.serve();

} catch (Exception e) {

System.out.println("Server start error!!!");

e.printStackTrace();

}

}

### 实现服务处理接口impl

实例：

**public** **class** UserServiceImpl **implements** UserService.Iface {…}

### 创建TProcessor

实例：

TProcessor tprocessor = **new** UserService.Processor<UserService.Iface>(

**new** UserServiceImpl());

### 创建TServerTransport

**TServerSocket**

public static final int SERVER\_PORT = 7090;

TServerSocket serverTransport = new TServerSocket(SERVER\_PORT);

TServerSocket扩展TServerTransport，有以下常用构造函数

TServerSocket(java.net.InetSocketAddress bindAddr)

TServerSocket(java.net.InetSocketAddress bindAddr, int clientTimeout)

TServerSocket(int port)

TServerSocket(int port, int clientTimeout)

TServerSocket(java.net.ServerSocket serverSocket)

TServerSocket(java.net.ServerSocket serverSocket, int clientTimeout)

**TNonblockingServerSocket**

TNonblockingServerSocket(java.net.InetSocketAddress bindAddr)

TNonblockingServerSocket(java.net.InetSocketAddress bindAddr, int clientTimeout)

TNonblockingServerSocket(int port)

TNonblockingServerSocket(int port, int clientTimeout)

### 创建TProtocol

### 创建TServer

关于如何选择服务器类型问题，请看后面的文章

#### TSimpleServer

TProcessor tprocessor = **new** UserService.Processor<UserService.Iface>(

**new** UserServiceImpl());

TServerSocket serverTransport = **new** TServerSocket(*SERVER\_PORT*);

TServer.Args tArgs = **new** TServer.Args(serverTransport);

tArgs.processor(tprocessor);

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

TServer server = **new** TSimpleServer(tArgs);

#### TThreadPoolServer

TThreadPoolServer.Args tArgs = **new** TThreadPoolServer.Args(serverTransport);

tArgs.minWorkerThreads(10);

tArgs.maxWorkerThreads(20);

tArgs.processor(tprocessor);

tArgs.protocolFactory(profactory);

TServer server = **new** TThreadPoolServer(tArgs);

#### TNonblockingServer

服务器端代码NonBlockingServerDemo

**public** **class** NonBlockingServerDemo {

**public** **static** **final** **int** *SERVER\_PORT* = 7090;

**public** **void** start() {

**try** {

System.*out*.println("UserServer start ....");

TNonblockingServerTransport serverTransport = **new** TNonblockingServerSocket(

*SERVER\_PORT*);

TBinaryProtocol.Factory profactory = **new** TBinaryProtocol.Factory();

TProcessor tprocessor = **new** UserService.Processor<UserService.Iface>(

**new** UserServiceImpl());

TNonblockingServer.Args tArgs = **new** TNonblockingServer.Args(

serverTransport);

tArgs.processor(tprocessor);

tArgs.protocolFactory(profactory);

TServer server = **new** TNonblockingServer(tArgs);

server.serve();

} **catch** (Exception e) {

System.*out*.println("Server start error!!!");

e.printStackTrace();

}

}

/\*\*

\* **@param** args

\*/

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

NonBlockingServerDemo server = **new** NonBlockingServerDemo();

server.start();

}

}

客户端代码NonBlockingClientDemo

public static final String SERVER\_IP = "localhost";

public static final int SERVER\_PORT = 7090;

public static final int TIMEOUT = 30000;

public void login(String userName, String passwd) {

TTransport transport = null;

try {

transport = new TFramedTransport(

new TSocket(SERVER\_IP, SERVER\_PORT));

// 协议要和服务端一致

TProtocol protocol = new TBinaryProtocol(transport);

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

transport.open();

LoginResponse loginResponse = client.login(userName, passwd);

System.out.println("Thrify client result =: " + loginResponse);

} catch (TTransportException e) {

e.printStackTrace();

} catch (TException e) {

e.printStackTrace();

} finally {

if (null != transport) {

transport.close();

}

}

}

#### THshaServer

**public** **void** start() {

**try** {

System.*out*.println("UserServer start ....");

TNonblockingServerTransport serverTransport = **new** TNonblockingServerSocket(

*SERVER\_PORT*);

TBinaryProtocol.Factory profactory = **new** TBinaryProtocol.Factory();

TProcessor tprocessor = **new** UserService.Processor<UserService.Iface>(

**new** UserServiceImpl());

THsHaServer.Args tArgs = **new** THsHaServer.Args(serverTransport);

tArgs.processor(tprocessor);

tArgs.protocolFactory(profactory);

TServer server = **new** THsHaServer(tArgs);

server.serve();

} **catch** (Exception e) {

System.*out*.println("Server start error!!!");

e.printStackTrace();

}

}

#### TThreadedSelectorServer

**public** **void** start() {

**try** {

System.*out*.println("UserServer start ....");

TNonblockingServerTransport serverTransport = **new** TNonblockingServerSocket(

*SERVER\_PORT*);

TBinaryProtocol.Factory profactory = **new** TBinaryProtocol.Factory();

TProcessor tprocessor = **new** UserService.Processor<UserService.Iface>(

**new** UserServiceImpl());

TThreadedSelectorServer.Args tArgs = **new** TThreadedSelectorServer.Args(

serverTransport);

tArgs.processor(tprocessor);

tArgs.protocolFactory(profactory);

TServer server = **new** TThreadedSelectorServer(tArgs);

server.serve();

} **catch** (Exception e) {

System.*out*.println("Server start error!!!");

e.printStackTrace();

}

}

### 启动Server

server.serve();

## 客户端编码基本步骤

### 创建Transport

### 创建TProtocol

### 基于TTransport和TProtocol创建 Client

### 调用Client的相应方法

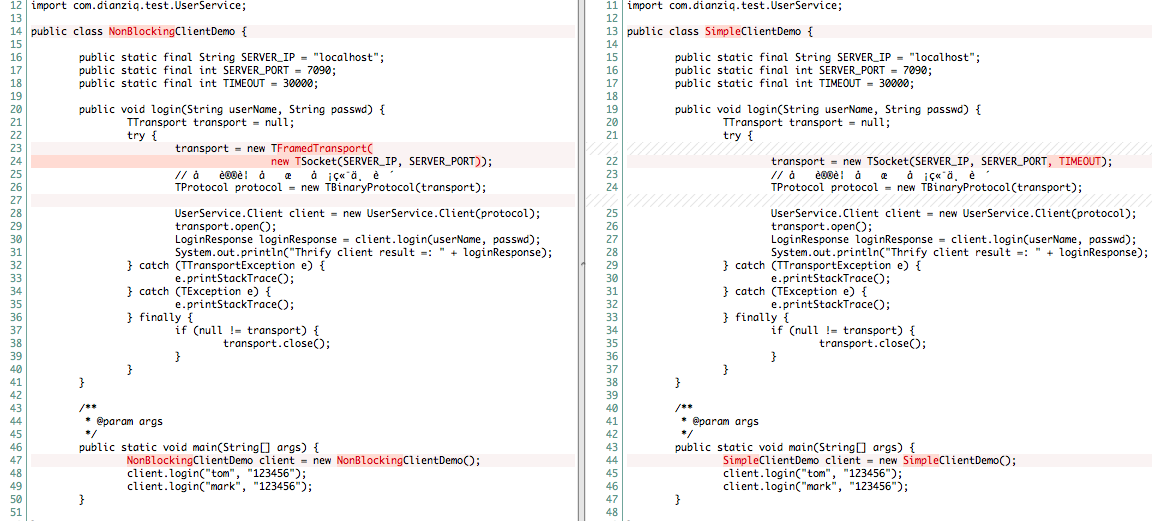
说明：客户端调用其实就两种方式：阻塞和非阻塞

**阻塞**

transport = **new** TSocket(*SERVER\_IP*, *SERVER\_PORT*, *TIMEOUT*);

**非阻塞**

transport = **new** TFramedTransport(**new** TSocket(*SERVER\_IP*, *SERVER\_PORT*));



# 最佳实践

## 如何选择服务TServer类型?

Thrift Java Servers Compared

原文URL：

<https://github.com/m1ch1/mapkeeper/wiki/Thrift-Java-Servers-Compared>

翻译版URL：

<http://www.codelast.com/?p=4824#more-4824>

Java中可用的server实现

· TSimpleServer

· TNonblockingServer

· THsHaServer

· TThreadedSelectorServer

· TThreadPoolServer

### TSimpleServer

TSimplerServer接受一个连接，处理连接请求，直到客户端关闭了连接，它才回去接受一个新的连接。正因为它只在一个单独的线程中以阻塞I/O的方式完成这些工作，所以它只能服务一个客户端连接，其他所有客户端在被服务器端接受之前都只能等待。**TSimpleServer主要用于测试目的，不要在生产环境中使用它！**

### TNonblockingServer vs. THsHaServer

TNonblockingServer使用非阻塞的I/O解决了TSimpleServer一个客户端阻塞其他所有客户端的问题。它使用了java.nio.channels.Selector，通过调用select()，它使得你阻塞在多个连接上，而不是阻塞在单一的连接上。当一或多个连接准备好被接受/读/写时，select()调用便会返回。TNonblockingServer处理这些连接的时候，要么接受它，要么从它那读数据，要么把数据写到它那里，然后再次调用select()来等待下一个可用的连接。通用这种方式，server可同时服务多个客户端，而不会出现一个客户端把其他客户端全部“饿死”的情况。

然而，还有个棘手的问题：所有消息是被调用select()方法的同一个线程处理的。假设有10个客户端，处理每条消息所需时间为100毫秒，那么，latency和吞吐量分别是多少？当一条消息被处理的时候，其他9个客户端就等着被select，所以客户端需要等待1秒钟才能从服务器端得到回应，吞吐量就是10个请求/秒。如果可以同时处理多条消息的话，会很不错吧？

因此，THsHaServer（半同步/半异步的server）就应运而生了。它使用一个单独的线程来处理网络I/O，一个独立的worker线程池来处理消息。这样，只要有空闲的worker线程，消息就会被立即处理，因此多条消息能被并行处理。用上面的例子来说，现在的latency就是100毫秒，而吞吐量就是100个请求/秒。

为了演示，我做了一个测试，有10客户端和一个修改过的消息处理器——它的功能仅仅是在返回之前简单地sleep 100毫秒。我使用的是有10个worker线程的THsHaServer。消息处理器的代码看上去就像下面这样：

public ResponseCode sleep() throws TException

{

    try {

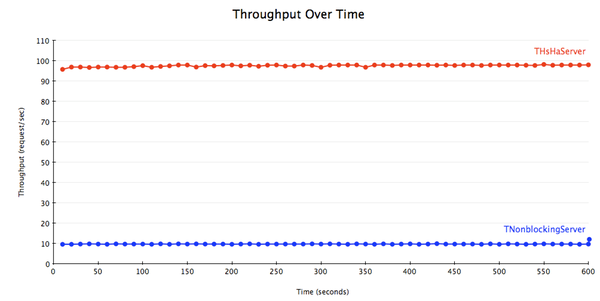
        Thread.sleep(100);

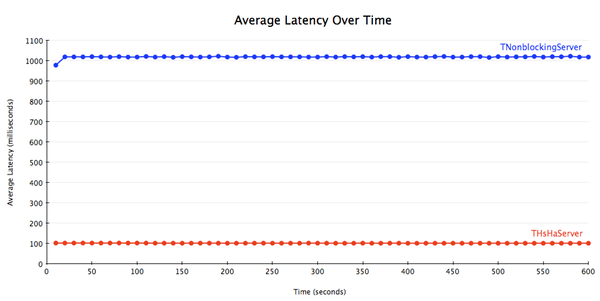
    } catch (Exception ex) {

    }

    return ResponseCode.Success;

}

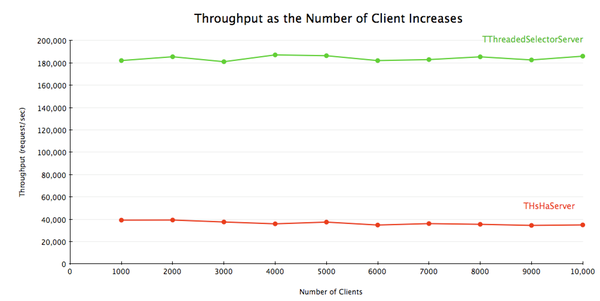


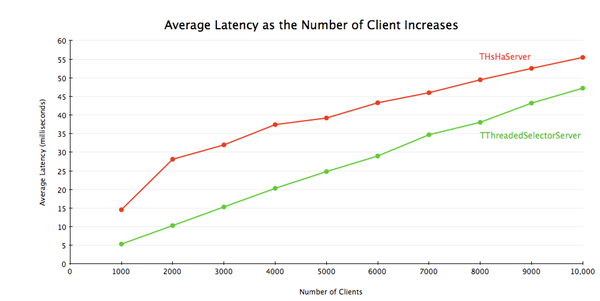


结果正如我们想像的那样，THsHaServer能够并行处理所有请求，而TNonblockingServer只能一次处理一个请求。

### THsHaServer vs. TThreadedSelectorServer

Thrift0.8引入了另一种server实现，即TThreadedSelectorServer。它与THsHaServer的主要区别在于。TThreadedSelectorServer允许你用多个线程来处理网络I/O。它维护了两个线程池，一个用来处理网络I/O，另一个用来进行请求的处理。当网络I/O是瓶颈的时候，TThreadedSelectorServer比THsHaServer的表现要好。为了展现它们的区别，我进行了一个测试，令其消息处理器在不做任何工作的情况下立即返回，以衡量在不同客户端数量的情况下的平均latency和吞吐量。对THsHaServer，我使用32个worker线程；对TThreadedSelectorServer，我使用16个worker线程和16个selector线程。





结果显示，TThreadedSelectorServer比THsHaServer的吞吐量高得多，并且维持在一个更低的latency上。

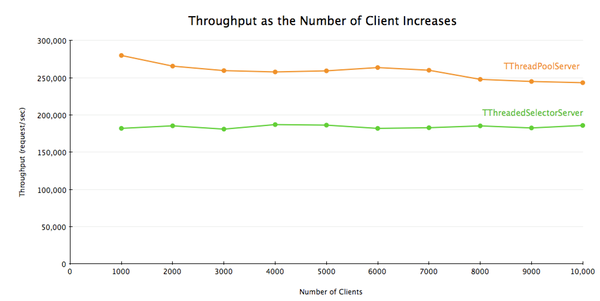
### TThreadedSelectorServer vs. TThreadPoolServer

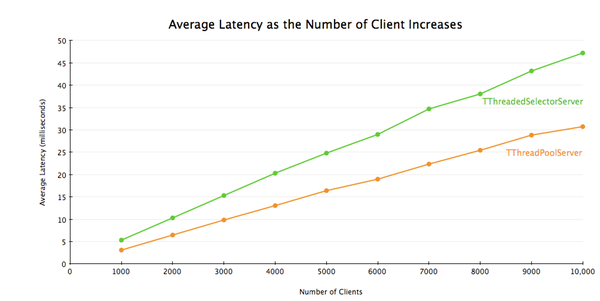
最后，还剩下 TThreadPoolServer。TThreadPoolServer与其他三种server不同的是：

1. 有一个专用的线程用来接受连接。
2. 一旦接受了一个连接，它就会被放入ThreadPoolExecutor中的一个worker线程里处理。
3. worker线程被绑定到特定的客户端连接上，直到它关闭。一旦连接关闭，该worker线程就又回到了线程池中。
4. 你可以配置线程池的最小、最大线程数，默认值分别是5（最小）和Integer.MAX\_VALUE（最大）。

这意味着，如果有1万个并发的客户端连接，你就需要运行1万个线程。所以它对系统资源的消耗不像其他类型的server一样那么“友好”。此外，如果客户端数量超过了线程池中的最大线程数，在有一个worker线程可用之前，请求将被一直阻塞在那里。

我们已经说过，TThreadPoolServer的表现非常优异。在我正在使用的计算机上，它可以支持1万个并发连接而没有任何问题。如果你提前知道了将要连接到你服务器上的客户端数量，并且你不介意运行大量线程的话，TThreadPoolServer对你可能是个很好的选择。





结论

希望本文能帮你做出决定：哪一种Thrift server适合你。我认为TThreadedSelectorServer对大多数案例来说都是个安全之选。如果你的系统资源允许运行大量并发线程的话，你可能会想考虑使用TThreadPoolServer。

## 如何实现服务接口的版本变更?

http://diwakergupta.github.com/thrift-missing-guide/#\_defining\_services

Versioning/Compatibility

Protocols evolve over time. If an existing message type no longer meets all your needs — for example, you’d like the message format to have an extra field — but you’d still like to use code created with the old format, don’t worry! It’s very simple to update message types without breaking any of your existing code. Just remember the following rules:

1. Don’t change the numeric tags for any existing fields.
2. Any new fields that you add should be optional. This means that any messages serialized by code using your "old" message format can be parsed by your new generated code, as they won’t be missing any required elements. You should set up sensible default values for these elements so that new code can properly interact with messages generated by old code. Similarly, messages created by your new code can be parsed by your old code: old binaries simply ignore the new field when parsing. However, the unknown fields are not discarded, and if the message is later serialized, the unknown fields are serialized along with it — so if the message is passed on to new code, the new fields are still available.
3. Non-required fields can be removed, as long as the tag number is not used again in your updated message type (it may be better to rename the field instead, perhaps adding the prefix "OBSOLETE\_", so that future users of your .thrift can’t accidentally reuse the number).
4. Changing a default value is generally OK, as long as you remember that default values are never sent over the wire. Thus, if a program receives a message in which a particular field isn’t set, the program will see the default value as it was defined in that program’s version of the protocol. It will NOT see the default value that was defined in the sender’s code.

协议随着岁月而演变，如果一个已经存在的消息类型不再满足需求，但是你仍然想使用原来使用thrift文件生成的代码，比如你想在现在的消息类型上增加一个额外的字段。

不搞坏原来的任何代码而更新消息类型，是非常简单的，只需要时刻记录以下的规则：

1. 不要修改现存的任何已经存在字段的数字标签
2. 任何新增的字段应该是optional的，这就意味着，所有新定义产生的代码都能解析新老的格式，因为一定会有任何一个标上required的元素。你应该设置可能的默认值给这个元素，让新代码与老数据也能正常协作。与此类似，新数据也能被老代码解析，老的实现解析时会忽略新的字段。尽管如此，未知的字段不能被丢弃，并且如果消息被延迟序列化，未知的字段也会被一起序列化，因此当消息到达新代码的时候，新字段依旧可用。
3. 当你更新的消息类型里不再包括相关的标签数字时，非required字段可以删除。（把这个废字段改名，比如说加个前缀OBSOLETE\_ ，可以防止未来维护.thrift文件人误用原来的数字，这做法可能更好。)
4. 修改默认值基本上是OK的，但是你记得这个默认值永远不会“串线”。比如，如果程序收到一个在某个字段没有值的消息，程序会看那个版本里定义的默认值。这可能会导致得不到正确的默认值。

# 参考资料

1. Thrift使用指南 dong

http://dongxicheng.org/search-engine/thrift-guide/

1. Thrift: The Missing Guide

http://diwakergupta.github.io/thrift-missing-guide/