# 概述

Thrift是Facebook用于后台各个服务间的通讯,Thrift的设计强调统一的编程接口的多语言通讯框架。具体对Thrift的介绍，可以参考网上的各种资料，这样主要介绍JD\_Thrift(Java Dynamic Thrift)这个开发库的使用，它目前仅支持**v0.8.0**版本的Thrift。

在今天说到Thrift就不可避免的要谈到Avro。Avro出自Hadoop之父Doug Cutting,其目标是要建立一个新的，标准性的云计算的数据交换和存储的Protocol。它偏向实用，主张建立一个统一的标准，并不介意采用特定的优化。Avro的创新之处在于融合了显式,declarative的Schema和高效二进制的数据表达，强调数据的自我描述，克服了以往单纯XML或二进制系统的缺陷。

Avro-generic方式支持Schema的动态加载，用通用的结构(map)代表数据对象，不需要编译加载直接就可以处理新的数据源，这是相对Thrift的一个非常重要的区别，而这个功能确实在某些情况下非常有用，这正是JD\_Thrift开发库的原始启动原因。

由于Thrift的实现采用了较好的抽象和分层，这使得给Thrift动态加载Schema成为可能。

下面我们分别从服务器端和client端来分别介绍如何使用JD\_Thrift。

# 服务器端动态加载Schema

下面是示例代码：

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

{

**try** {

// 第一步: 用户自己编写的业务代码

AutoTestHandler handler = **new** AutoTestHandler();

// 第二步: 生成对应thrift配置文件解析引擎，并生成自动TProcessor实例

AutoThriftGenerator tmpAutoThriftGenerator = **new** AutoThriftGenerator();

ThriftServicesParser tmpThriftServicesParser =

tmpAutoThriftGenerator.generateAutoThrift("D:/资料/thrift", "firstTest.thrift");

AutoProcessorGenerator tmpAutoProcessorGenerator = **new** AutoProcessorGenerator();

TBaseProcessor tmpProcessor = tmpAutoProcessorGenerator.generate("UserStorage", handler, tmpThriftServicesParser);

// 第三步: 启动服务

TServerTransport serverTransport = **new** TServerSocket(9090);

TServer server = **new** TSimpleServer(**new** Args(serverTransport).processor(tmpProcessor));

System.*out*.println("Starting the simple server...");

server.serve();

}

**catch**(Exception ex)

{

ex.printStackTrace();

}

}

服务器端代码包括3个步骤：

* 用户编写的业务处理类

这个类负责具体请求的处理，它是个POJO，该类的public方法都会被自动注册为服务，每个方法分别对应thrift配置文件中的服务定义，包括方法名、输入参数、返回数据、异常抛出等。下面结合代码进行说明：

某thrift配置文件服务片段如下：

service UserStorage {

void **store**(1: UserProfile user),

string **getUserName**() throws(1: Xception err1, 2: Xception2 err2),

list<list<UserProfile>> **retrieve**(1: i32 uid, 2: required string test) throws(1: Xception err1)

}

某业务处理类XXXTestHandler

public class XXXTestHandler

{

public void **store**(CommonStruct commonStructParm) throws TException {

}

public String **getUserName**() throws TException, CommonException

{

return "abc";

}

public ArrayList<ArrayList<CommonStruct>> **retrieve**(int uid, String testParm) throws TException, CommonException

{

ArrayList retList = new ArrayList();

return retList;

}

}

* 生成自动TProcessor处理实例

通过AutoThriftGenerator类生成相应thrift配置文件的解析器，通过AutoProcessorGenerator类生成相应的自动TProcessor实例，通过这些操作就完成了服务调用的序列化和反序列化操作。

* 启动服务

这个和原生的thrift处理方式一样。

# Client端动态加载Schema

public static void main(String[] args) {

try {

// 第一步: 创建thrift的通讯链路

TTransport transport = null;

transport = new TSocket("localhost", 9090);

transport.open();

TProtocol protocol = new TBinaryProtocol(transport);

// 第二步: 生成对应thrift配置文件解析引擎，并生成自动Client实例

AutoThriftGenerator tmpAutoThriftGenerator = new AutoThriftGenerator();

ThriftServicesParser tmpThriftServicesParser =

tmpAutoThriftGenerator.generateAutoThrift("D:/资料/thrift", "firstTest.thrift");

AutoClient client = new AutoClient(protocol, "UserStorage", tmpThriftServicesParser);

// 第三步: 使用自动Client实例,调用具体的远端服务(用户自己编写的业务代码)

CommonArgs tmpCommonArgs = new CommonArgs();

tmpCommonArgs.addOneValue("uid", 10);

tmpCommonArgs.addOneValue("test", "test123");

ArrayList tmpList = (ArrayList) client.sendRequest("retrieve", tmpCommonArgs);

System.out.println("服务器端返回:"+tmpList.size());

// 第四步: 关闭通讯链路

transport.close();

}

catch (TException x) {

x.printStackTrace();

}

catch (CommonException ae) {

ae.printStackTrace();

}

}

client端代码包括4个步骤：

* 创建thrift的通讯链路

这个和原生的thrift处理方式一样。

* 生成自动Client实例

通过AutoThriftGenerator类生成thrift配置文件的解析器，并创建AutoClient的一个实例，后续真正的服务调用都是通过这个AutoClient实例发出的。

* 使用自动Client实例,调用具体的远端服务(用户自己编写的业务代码)

这里进行具体服务的业务调用

* 关闭通讯链路

这个和原生的thrift处理方式一样。