thrift (转)

一、**ubuntu**下**thrift**的安装

1.下载源代码
http://thrift.apache.org/download/
下载最新版本thrift-0.8.0.tar.gz

2.安装boost库
sudo apt-get install libboost-dev libboost-dbg libboost-doc bcp libboost-*

3.安装其他相关工具包
sudo apt-get install libboost-dev libboost-test-dev libboost-program-options-dev libevent-dev automake libtool flex bison pkg-config g++ libssl-dev ant
如果需要支持java，需要安装jdk，配置java环境变量。

4.解压文件，进入目录thrift-0.8.0安装
./configure --with-cpp --with-boost --without-python --without-csharp --with-java --without-erlang --without-perl --with-php --without-php_extension --without-ruby --without-haskell --without-go
make
sudo make install
要支持java，需要编译生成jar包，到lib/java目录下，执行ant命令。将在lib/java/build目录下生成libthrift-0.8.0.jar和libthrift-0.8.0-javadoc.jar。编译过程中，可能出错，需要检查lib/java/build/tools/maven-ant-tasks-2.1.3.jar是否正确下载。

5.测试
直接输入thrift命令，看是否有用法提示

二、**thrift**自带的测试样例
进入tutorial文件夹，shared.thrift和tutorial.thrift是接口定义文件。
thrift -r --gen java tutorial.thrift
thirft -r --gen cpp tutorial.thrift
执行这两条命令可以生成gen-java和gen-cpp两个文件夹，这些是thrift编译器自动生成的代码。

然后到java目录下，执行ant命令，编译成功后，在两个不同的窗口下执行以下命令:
./JavaServer
./JavaClient simple

进入cpp目录下，执行make命令，如果编译出错，第一个错误是
/usr/local/include/thrift/protocol/TBinaryProtocol.tcc:147:35: error: there are no arguments to ‘htons’ that depend on a template parameter, so a declaration of ‘htons’ must be available
则修改Makefile，加上编译选项-DHAVE_NETINET_IN_H
```
server: CppServer.cpp
	g++ -DHAVE_NETINET_IN_H -o CppServer -I${THRIFT_DIR} -I${BOOST_DIR} -I../gen-cpp -L${LIB_DIR} -lthrift CppServer.cpp ${GEN_SRC}
```

```
client: CppClient.cpp
	g++ -DHAVE_NETINET_IN_H -o CppClient -I${THRIFT_DIR} -I${BOOST_DIR}  -I../gen-cpp
-L${LIB_DIR} -lthrift CppClient.cpp ${GEN_SRC}
```

编译通过生，将生成CppClient和CppServer两个可执行程序。同样，在两个不同的窗口执行以下命令:

```
./CppServer
./CppClient
```

而且java和c++的Client和Server都可以交叉运行。比如运行JavaServer和CppClient也能得到同样的结果。以此达到了多语言的相互调用。

三、**Hello World**

仿照tutorial，写一个简单的hello world。在tutorial平级目录下，建立目录hello，这里只是为了测试需要。
服务端用java，客户端用java和c++。

1. 编写tt.thrift

```
namespace java demo
namespace cpp demo

service Hello{
  string helloString(1:string para)
}
```

2. 编译生成代码

```
thrift -r --gen java tt.thrift
thrift -r --gen cpp tt.thrift
```

生成gen-java和gen-cpp两个文件夹
gen-cpp下有如下文件

```
Hello.cpp  Hello.h  Hello_server.skeleton.cpp  tt_constants.cpp  tt_constants.h  tt_types.cpp
 tt_types.h
```

其中Hello.h,Hello.cpp中定义了远程调用的接口，实现了底层通信。可以在Hello.h中找到客户端远程调用需要用到的类HelloClient，调用方法:

```
void helloString(std::string& _return, const std::string& para);
```

这个跟thrift文件中申明的方法有点不一定，返回参数是通过引用传回来的。
Hello_server.skeleton.cpp将实现Hello.h的服务端接口，如果要用c++作为服务端，还需要将这个文件拷出去，重命名，实现类HelloHandler的方法helloString，远程调用方法的业务逻辑也就写在helloString中。可能还需要改main函数中的端口信息。
gen-java/demo下只有Hello.java一个文件，它定义了服务端和客户端的接口，实现了底层的通信。

3. 编写java服务端和客户端
仿照tutorial，在hello目录下建立java目录，将tutorial/java/下的一些文件和目录拷到hello/java下

```
build.xml JavaClient  JavaServer  src
```

删除src下所有文件，在src下编写代码。

1） HelloImpl.java 远程过程调用的业务逻辑

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

class HelloImpl implements Hello.Iface {
 public HelloImpl() {}
 public String helloString(String para) throws org.apache.thrift.TException {
       //各种业务逻辑
        return "hello " + para;
 }
}
```

2) Server.java 服务端程序

```
import demo.*;
import java.io.IOException;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.protocol.TBinaryProtocol.Factory;
```

```
import org.apache.thrift.server.TServer;
import org.apache.thrift.server.TThreadPoolServer.Args;
import org.apache.thrift.server.TThreadPoolServer;
import org.apache.thrift.transport.TServerSocket;
import org.apache.thrift.transport.TTransportException;

public class Server {
private void start() {
try {
   TServerSocket serverTransport = new TServerSocket(7911);
   Hello.Processor processor = new Hello.Processor(new HelloImpl());
   Factory protFactory = new TBinaryProtocol.Factory(true, true);
   Args args = new Args(serverTransport);
   args.processor(processor);
   args.protocolFactory(protFactory);
   TServer server = new TThreadPoolServer(args);
   //TServer server = new TThreadPoolServer(processor, serverTransport, protFactory);
   System.out.println("Starting server on port 7911 ...");
   server.serve();

  } catch (TTransportException e) {
   e.printStackTrace();
  } catch (Exception e) {
   e.printStackTrace();
}
}

public static void main(String args[]) {
   Server srv = new Server();
   srv.start();
}
}
```

3) Client.java 客户端程序

```
import demo.*;
import java.io.IOException;
import org.apache.thrift.*;
import org.apache.thrift.protocol.*;
import org.apache.thrift.transport.*;

public class Client {
    public static void main(String [] args) {
        try {
                TTransport transport = new TSocket("localhost", 7911);
                TProtocol protocol = new TBinaryProtocol(transport);
                Hello.Client client = new Hello.Client(protocol);
                transport.open();
                System.out.println("Client calls hello");
                System.out.println(client.helloString("world"));
                transport.close();
            } catch (TException x) {
                x.printStackTrace();
            }
      }
}
```

4) 修改 build.xml

```
<project name="hello" default="hello" basedir=".">

  <description>Thrift Hello</description>
```

```
  <property name="src" location="src" />
  <property name="gen" location="../gen-java" />
  <property name="build" location="build" />

  <path id="libs.classpath">
    <fileset dir="http://www.cnblogs.com/lib/java/build">
      <include name="*.jar" />
      <exclude name="-test.jar" />
    </fileset>
    <fileset dir="http://www.cnblogs.com/lib/java/build/lib">
      <include name="*.jar" />
    </fileset>
  </path>
  <path id="build.classpath">
    <path refid="libs.classpath" />
    <pathelement path="${gen}" />
  </path>

  <target name="init">
    <tstamp />
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init">
    <javac srcdir="${gen}" destdir="${build}" classpathref="libs.classpath" />
    <javac srcdir="${src}" destdir="${build}" classpathref="build.classpath" />
  </target>

  <target name="hello" depends="compile">
    <jar jarfile="hello.jar" basedir="${build}"/>
  </target>

  <target name="clean">
    <delete dir="${build}" />
    <delete file="hello.jar" />
  </target>

</project>
```

5) 编译

ant

将生成build文件夹

Client.class demo hello HelloImpl.class hello.jar Server.class

6) 修改执行脚本

JavaClient

java -cp http://www.cnblogs.com/lib/java/build/lib/*:http://www.cnblogs.com/lib/java/build/*:hello.jar Client

JavaServer

java -cp http://www.cnblogs.com/lib/java/build/lib/*:http://www.cnblogs.com/lib/java/build/*:hello.jar Server

4. 编写c++客户端

同样仿照tutorial，将tutorial/cpp中的Makefile和CppClient.cpp拷到hello/cpp下。

1) 将CppClient.cpp重命名为Client.cpp，并修改

```
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
```

```
#include <protocol/TBinaryProtocol.h>
#include <transport/TSocket.h>
#include <transport/TTransportUtils.h>

#include "../gen-cpp/Hello.h"
#include <string>

using namespace std;
using namespace apache::thrift;
using namespace apache::thrift::protocol;
using namespace apache::thrift::transport;

using namespace demo;

using namespace boost;

int main(int argc, char** argv) {
  shared_ptr<TTransport> socket(new TSocket("localhost", 7911));
  shared_ptr<TTransport> transport(new TBufferedTransport(socket));
  shared_ptr<TProtocol> protocol(new TBinaryProtocol(transport));
  HelloClient client(protocol);

  try {
    transport->open();

    string ret;
    client.helloString(ret, "world");
    printf("%s\n", ret.c_str());

    transport->close();
  } catch (TException &tx) {
    printf("ERROR: %s\n", tx.what());
  }

}
```

2). 修改Makefile

```
BOOST_DIR = /usr/local/boost/include/boost-1_33_1/
THRIFT_DIR = /usr/local/include/thrift
LIB_DIR = /usr/local/lib

GEN_SRC = ../gen-cpp/tt_types.cpp ../gen-cpp/Hello.cpp

default: client

client: Client.cpp
	g++ -DHAVE_NETINET_IN_H -o client -I${THRIFT_DIR} -I${BOOST_DIR}  -I../gen-cpp -
L${LIB_DIR} -lthrift Client.cpp ${GEN_SRC}

clean:
	$(RM) -r client
```

3). 编译

make

生成可执行文件client

5. 运行程序

运行服务端程序，java目录下：./JavaServer

运行客户端程序，cpp目录下：./client

这样c++程序通过 client.helloString(ret, "client") 可以调用服务端的java接口String helloString(String

para)。
从而实现了远程多语言调用。