Apache Thrift - 可伸缩的跨语言服务开发框架

**简介：** Apache Thrift 是 Facebook 实现的一种高效的、支持多种编程语言的远程服务调用的框架。本文将从 Java 开发人员角度详细介绍 Apache Thrift 的架构、开发和部署，并且针对不同的传输协议和服务类型给出相应的 Java 实例，同时详细介绍 Thrift 异步客户端的实现，最后提出使用 Thrift 需要注意的事项。

**前言：**

目前流行的服务调用方式有很多种，例如基于 SOAP 消息格式的 Web Service，基于 JSON 消息格式的 RESTful 服务等。其中所用到的数据传输方式包括 XML，JSON 等，然而 XML 相对体积太大，传输效率低，JSON 体积较小，新颖，但还不够完善。本文将介绍由 Facebook 开发的远程服务调用框架 Apache Thrift，它采用接口描述语言定义并创建服务，支持可扩展的跨语言服务开发，所包含的代码生成引擎可以在多种语言中，如 C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk 等创建高效的、无缝的服务，其传输数据采用二进制格式，相对 XML 和 JSON 体积更小，对于高并发、大数据量和多语言的环境更有优势。本文将详细介绍 Thrift 的使用，并且提供丰富的实例代码加以解释说明，帮助使用者快速构建服务。

**一个简单的 Thrift 实例**

本文首先介绍一个简单的 Thrift 实现实例，使读者能够快速直观地了解什么是 Thrift 以及如何使用 Thrift 构建服务。

创建一个简单的服务 Hello。首先根据 Thrift 的语法规范编写脚本文件 Hello.thrift，代码如下：

**清单 1. Hello.thrift**

|  |
| --- |
| namespace java service.demo  service Hello{  string helloString(1:string para)  i32 helloInt(1:i32 para)  bool helloBoolean(1:bool para)  void helloVoid()  string helloNull()  } |

其中定义了服务 Hello 的五个方法，每个方法包含一个方法名，参数列表和返回类型。每个参数包括参数序号，参数类型以及参数名。 Thrift 是对 IDL(Interface Definition Language) 描述性语言的一种具体实现。因此，以上的服务描述文件使用 IDL 语法编写。使用 Thrift 工具编译 Hello.thrift，就会生成相应的 Hello.java 文件。该文件包含了在 Hello.thrift 文件中描述的服务 Hello 的接口定义，即 Hello.Iface 接口，以及服务调用的底层通信细节，包括客户端的调用逻辑 Hello.Client 以及服务器端的处理逻辑 Hello.Processor，用于构建客户端和服务器端的功能。

创建 HelloServiceImpl.java 文件并实现 Hello.java 文件中的 Hello.Iface 接口，代码如下：

**清单 2. HelloServiceImpl.java**

|  |
| --- |
| package service.demo;  import org.apache.thrift.TException;  public class HelloServiceImpl implements Hello.Iface {  @Override  public boolean helloBoolean(boolean para) throws TException {  return para;  }  @Override  public int helloInt(int para) throws TException {  try {  Thread.sleep(20000);  } catch (InterruptedException e) {  e.printStackTrace();  }  return para;  }  @Override  public String helloNull() throws TException {  return null;  }  @Override  public String helloString(String para) throws TException {  return para;  }  @Override  public void helloVoid() throws TException {  System.out.println("Hello World");  }  } |

创建服务器端实现代码，将 HelloServiceImpl 作为具体的处理器传递给 Thrift 服务器，代码如下：

**清单 3. HelloServiceServer.java**

|  |
| --- |
| package service.server;  import org.apache.thrift.TProcessor;  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;  import org.apache.thrift.transport.TServerSocket;  import org.apache.thrift.transport.TTransportException;  import service.demo.Hello;  import service.demo.HelloServiceImpl;  public class HelloServiceServer {  /\*\*  \* 启动 Thrift 服务器  \* @param args  \*/  public static void main(String[] args) {  try {  // 设置服务端口为 7911  TServerSocket serverTransport = new TServerSocket(7911);  // 设置协议工厂为 TBinaryProtocol.Factory  Factory proFactory = new TBinaryProtocol.Factory();  // 关联处理器与 Hello 服务的实现  TProcessor processor = new Hello.Processor(new HelloServiceImpl());  TServer server = new TThreadPoolServer(processor, serverTransport,  proFactory);  System.out.println("Start server on port 7911...");  server.serve();  } catch (TTransportException e) {  e.printStackTrace();  }  }  } |

创建客户端实现代码，调用 Hello.client 访问服务端的逻辑实现，代码如下：

**清单 4. HelloServiceClient.java**

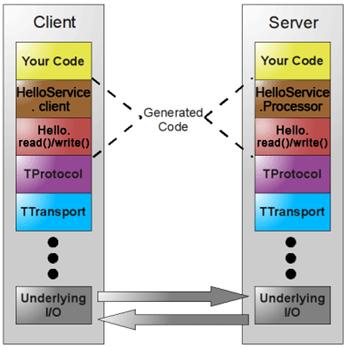
|  |
| --- |
| package service.client;  import org.apache.thrift.TException;  import org.apache.thrift.protocol.TBinaryProtocol;  import org.apache.thrift.protocol.TProtocol;  import org.apache.thrift.transport.TSocket;  import org.apache.thrift.transport.TTransport;  import org.apache.thrift.transport.TTransportException;  import service.demo.Hello;  public class HelloServiceClient {  /\*\*  \* 调用 Hello 服务  \* @param args  \*/  public static void main(String[] args) {  try {  // 设置调用的服务地址为本地，端口为 7911  TTransport transport = new TSocket("localhost", 7911);  transport.open();  // 设置传输协议为 TBinaryProtocol  TProtocol protocol = new TBinaryProtocol(transport);  Hello.Client client = new Hello.Client(protocol);  // 调用服务的 helloVoid 方法  client.helloVoid();  transport.close();  } catch (TTransportException e) {  e.printStackTrace();  } catch (TException e) {  e.printStackTrace();  }  }  } |

代码编写完后运行服务器，再启动客户端调用服务 Hello 的方法 helloVoid，在服务器端的控制台窗口输出“Hello World”(helloVoid 方法实现在控制台打印字符串，没有返回值，所以客户端调用方法后没有返回值输出，读者可以自己尝试其他有返回值方法的调用，其结果可以打印在客户端的控制台窗口 )。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-apachethrift/#ibm-pcon)

**Thrift 架构**

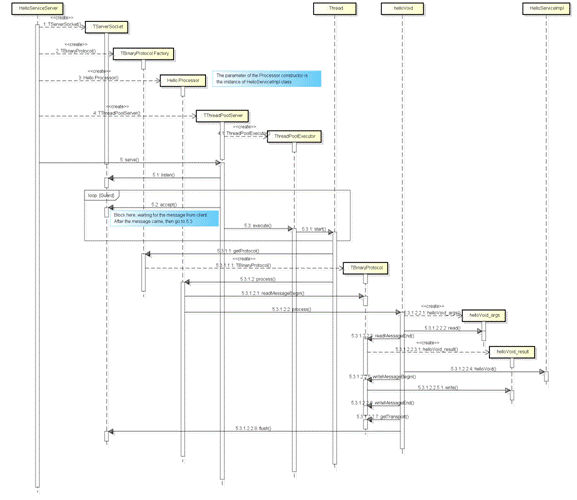
Thrift 包含一个完整的堆栈结构用于构建客户端和服务器端。下图描绘了 Thrift 的整体架构。

**图 1. 架构图**  
 

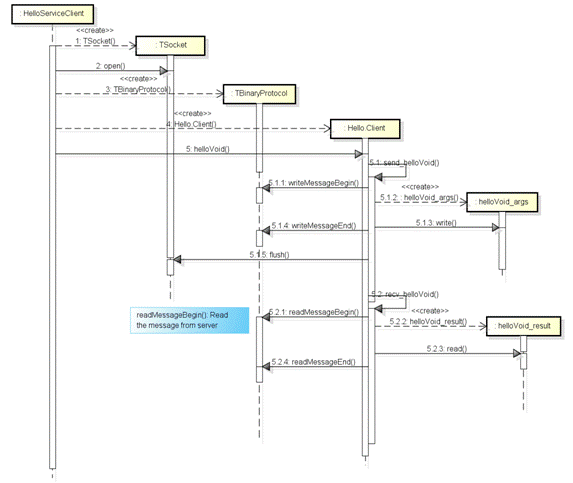
如图所示，图中黄色部分是用户实现的业务逻辑，褐色部分是根据 Thrift 定义的服务接口描述文件生成的客户端和服务器端代码框架，红色部分是根据 Thrift 文件生成代码实现数据的读写操作。红色部分以下是 Thrift 的传输体系、协议以及底层 I/O 通信，使用 Thrift 可以很方便的定义一个服务并且选择不同的传输协议和传输层而不用重新生成代码。

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。

服务端和客户端具体的调用流程如下：

**图 2. Server 端启动、服务时序图（[查看大图](http://www.ibm.com/developerworks/cn/java/j-lo-apachethrift/image004.png)）**  
 

该图所示是 HelloServiceServer 启动的过程以及服务被客户端调用时，服务器的响应过程。从图中我们可以看到，程序调用了 TThreadPoolServer 的 serve 方法后，server 进入阻塞监听状态，其阻塞在 TServerSocket 的 accept 方法上。当接收到来自客户端的消息后，服务器发起一个新线程处理这个消息请求，原线程再次进入阻塞状态。在新线程中，服务器通过 TBinaryProtocol 协议读取消息内容，调用 HelloServiceImpl 的 helloVoid 方法，并将结果写入 helloVoid\_result 中传回客户端。

**图 3. Client 端调用服务时序图（[查看大图](http://www.ibm.com/developerworks/cn/java/j-lo-apachethrift/image006.png)）**  
 

该图所示是 HelloServiceClient 调用服务的过程以及接收到服务器端的返回值后处理结果的过程。从图中我们可以看到，程序调用了 Hello.Client 的 helloVoid 方法，在 helloVoid 方法中，通过 send\_helloVoid 方法发送对服务的调用请求，通过 recv\_helloVoid 方法接收服务处理请求后返回的结果。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-apachethrift/#ibm-pcon)

**数据类型**

Thrift 脚本可定义的数据类型包括以下几种类型：

* 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
* 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：
  + list：对应 Java 的 ArrayList
  + set：对应 Java 的 HashSet
  + map：对应 Java 的 HashMap
* 异常类型：
  + exception：对应 Java 的 Exception
* 服务类型：
  + service：对应服务的类

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-apachethrift/#ibm-pcon)

**协议**

Thrift 可以让用户选择客户端与服务端之间传输通信协议的类别，在传输协议上总体划分为文本 (text) 和二进制 (binary) 传输协议，为节约带宽，提高传输效率，一般情况下使用二进制类型的传输协议为多数，有时还会使用基于文本类型的协议，这需要根据项目 / 产品中的实际需求。常用协议有以下几种：

* TBinaryProtocol —— 二进制编码格式进行数据传输

使用方法如清单 3 和清单 4 所示。

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

构建 TCompactProtocol 协议的服务器和客户端只需替换清单 3 和清单 4 中 TBinaryProtocol 协议部分即可，替换成如下代码：

**清单 5. 使用 TCompactProtocol 协议构建的 HelloServiceServer.java**

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

**清单 6. 使用 TCompactProtocol 协议的 HelloServiceClient.java**

|  |
| --- |
| TCompactProtocol protocol = new TCompactProtocol(transport); |

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

构建 TJSONProtocol 协议的服务器和客户端只需替换清单 3 和清单 4 中 TBinaryProtocol 协议部分即可，替换成如下代码：

**清单 7. 使用 TJSONProtocol 协议构建的 HelloServiceServer.java**

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

**清单 8. 使用 TJSONProtocol 协议的 HelloServiceClient.java**

|  |
| --- |
| TJSONProtocol protocol = new TJSONProtocol(transport); |

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

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-apachethrift/#ibm-pcon)

**传输层**

常用的传输层有以下几种：

* TSocket —— 使用阻塞式 I/O 进行传输，是最常见的模式

使用方法如清单 4 所示。

* 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 4 中 TTransport 部分，代码如下，清单 9 中 TNonblockingServerTransport 类是构建非阻塞 socket 的抽象类，TNonblockingServerSocket 类继承 TNonblockingServerTransport

**清单 9. 使用 TFramedTransport 传输层构建的 HelloServiceServer.java**

|  |
| --- |
| TNonblockingServerTransport serverTransport;  serverTransport = new TNonblockingServerSocket(10005);  Hello.Processor processor = new Hello.Processor(new HelloServiceImpl());  TServer server = new TNonblockingServer(processor, serverTransport);  System.out.println("Start server on port 10005 ...");  server.serve(); |

**清单 10. 使用 TFramedTransport 传输层的 HelloServiceClient.java**

|  |
| --- |
| TTransport transport = new TFramedTransport(new TSocket("localhost", 10005)); |

* TNonblockingTransport —— 使用非阻塞方式，用于构建异步客户端

使用方法请参考 Thrift 异步客户端构建

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-apachethrift/#ibm-pcon)

**服务端类型**

常见的服务端类型有以下几种：

* TSimpleServer —— 单线程服务器端使用标准的阻塞式 I/O

代码如下：

**清单 11. 使用 TSimpleServer 服务端构建的 HelloServiceServer.java**

|  |
| --- |
| TServerSocket serverTransport = new TServerSocket(7911);  TProcessor processor = new Hello.Processor(new HelloServiceImpl());  TServer server = new TSimpleServer(processor, serverTransport);  System.out.println("Start server on port 7911...");  server.serve(); |

客户端的构建方式可参考清单 4。

* TThreadPoolServer —— 多线程服务器端使用标准的阻塞式 I/O

使用方法如清单 3 所示。

* TNonblockingServer —— 多线程服务器端使用非阻塞式 I/O

使用方法请参考 Thrift 异步客户端构建

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-apachethrift/#ibm-pcon)

**Thrift 异步客户端构建**

Thrift 提供非阻塞的调用方式，可构建异步客户端。在这种方式中，Thrift 提供了新的类 TAsyncClientManager 用于管理客户端的请求，在一个线程上追踪请求和响应，同时通过接口 AsyncClient 传递标准的参数和 callback 对象，服务调用完成后，callback 提供了处理调用结果和异常的方法。

首先我们看 callback 的实现：

**清单 12.CallBack 的实现：MethodCallback.java**

|  |
| --- |
| package service.callback;  import org.apache.thrift.async.AsyncMethodCallback;  public class MethodCallback implements AsyncMethodCallback {  Object response = null;  public Object getResult() {  // 返回结果值  return this.response;  }  // 处理服务返回的结果值  @Override  public void onComplete(Object response) {  this.response = response;  }  // 处理调用服务过程中出现的异常  @Override  public void onError(Throwable throwable) {  }  } |

如代码所示，onComplete 方法接收服务处理后的结果，此处我们将结果 response 直接赋值给 callback 的私有属性 response。onError 方法接收服务处理过程中抛出的异常，此处未对异常进行处理。

创建非阻塞服务器端实现代码，将 HelloServiceImpl 作为具体的处理器传递给异步 Thrift 服务器，代码如下：

**清单 13.HelloServiceAsyncServer.java**

|  |
| --- |
| package service.server;  import org.apache.thrift.server.TNonblockingServer;  import org.apache.thrift.server.TServer;  import org.apache.thrift.transport.TNonblockingServerSocket;  import org.apache.thrift.transport.TNonblockingServerTransport;  import org.apache.thrift.transport.TTransportException;  import service.demo.Hello;  import service.demo.HelloServiceImpl;  public class HelloServiceAsyncServer {  /\*\*  \* 启动 Thrift 异步服务器  \* @param args  \*/  public static void main(String[] args) {  TNonblockingServerTransport serverTransport;  try {  serverTransport = new TNonblockingServerSocket(10005);  Hello.Processor processor = new Hello.Processor(  new HelloServiceImpl());  TServer server = new TNonblockingServer(processor, serverTransport);  System.out.println("Start server on port 10005 ...");  server.serve();  } catch (TTransportException e) {  e.printStackTrace();  }  }  } |

HelloServiceAsyncServer 通过 java.nio.channels.ServerSocketChannel 创建非阻塞的服务器端等待客户端的连接。

创建异步客户端实现代码，调用 Hello.AsyncClient 访问服务端的逻辑实现，将 MethodCallback 对象作为参数传入调用方法中，代码如下：

**清单 14.HelloServiceAsyncClient.java**

|  |
| --- |
| package service.client;  import java.io.IOException;  import org.apache.thrift.async.AsyncMethodCallback;  import org.apache.thrift.async.TAsyncClientManager;  import org.apache.thrift.protocol.TBinaryProtocol;  import org.apache.thrift.protocol.TProtocolFactory;  import org.apache.thrift.transport.TNonblockingSocket;  import org.apache.thrift.transport.TNonblockingTransport;  import service.callback.MethodCallback;  import service.demo.Hello;  public class HelloServiceAsyncClient {  /\*\*  \* 调用 Hello 服务  \* @param args  \*/  public static void main(String[] args) throws Exception {  try {  TAsyncClientManager clientManager = new TAsyncClientManager();  TNonblockingTransport transport = new TNonblockingSocket(  "localhost", 10005);  TProtocolFactory protocol = new TBinaryProtocol.Factory();  Hello.AsyncClient asyncClient = new Hello.AsyncClient(protocol,  clientManager, transport);  System.out.println("Client calls .....");  MethodCallback callBack = new MethodCallback();  asyncClient.helloString("Hello World", callBack);  Object res = callBack.getResult();  while (res == null) {  res = callBack.getResult();  }  System.out.println(((Hello.AsyncClient.helloString\_call) res)  .getResult());  } catch (IOException e) {  e.printStackTrace();  }  }  } |

HelloServiceAsyncClient 通过 java.nio.channels.Socketchannel 创建异步客户端与服务器建立连接。在本文中异步客户端通过以下的循环代码实现了同步效果，读者可去除这部分代码后再运行对比。

**清单 15. 异步客户端实现同步效果代码段**

|  |
| --- |
| Object res = callBack.getResult();  // 等待服务调用后的返回结果  while (res == null) {  res = callBack.getResult();  } |

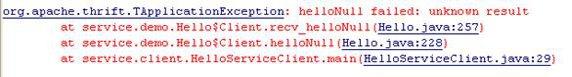
通过与清单 9 和清单 10 的代码比较，我们可以构建一个 TNonblockingServer 服务类型的服务端，在客户端构建一个 TFramedTransport 传输层的同步客户端和一个 TNonblockingTransport 传输层的异步客户端，那么一个服务就可以通过一个 socket 端口提供两种不同的调用方式。有兴趣的读者可以尝试一下。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-apachethrift/#ibm-pcon)

**常见问题**

**NULL 问题**

我们在对服务的某个方法调用时，有时会出现该方法返回 null 值的情况，在 Thrift 中，直接调用一个返回 null 值的方法会抛出 TApplicationException 异常。在清单 2 中，HelloServiceImpl 里实现了 helloNull 方法，返回 null 值，我们在 HelloServiceClient.java 中加入调用该方法的代码，出现如下图所示的异常：

**图 4. TApplicationException 异常**  
 

为了处理返回 null 值情况，我们要捕获该异常，并进行相应的处理，具体客户端代码实现如下：

**清单 16. 处理服务返回值为 null 的代码**

|  |
| --- |
| package service.client;  import org.apache.thrift.TApplicationException;  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 service.demo.Hello;  public class HelloServiceClient {  /\*\*  \* 调用 Hello 服务，并处理 null 值问题  \* @param args  \*/  public static void main(String[] args) {  try {  TTransport transport = new TSocket("localhost", 7911);  transport.open();  TProtocol protocol = new TBinaryProtocol(transport);  Hello.Client client = new Hello.Client(protocol);  System.out.println(client.helloNull());  transport.close();  } catch (TTransportException e) {  e.printStackTrace();  } catch (TException e) {  if (e instanceof TApplicationException  && ((TApplicationException) e).getType() ==  TApplicationException.MISSING\_RESULT) {  System.out.println("The result of helloNull function is NULL");  }  }  }  } |

调用 helloNull 方法后，会抛出 TApplicationException 异常，并且异常种类为 MISSING\_RESULT，本段代码显示，捕获该异常后，直接在控制台打印“The result of helloNull function is NULL”信息。

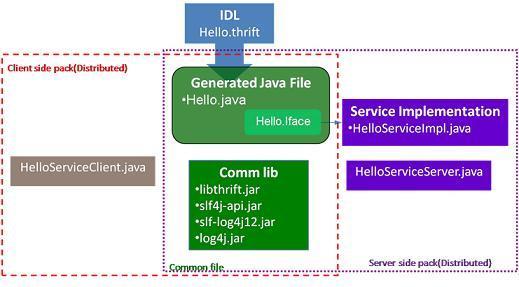
[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-apachethrift/#ibm-pcon)

**安装部署**

Apache Thrift 的官方网站为：[http://thrift.apache.org/tutorial/](http://thrift.apache.org/tutorial/%20)，具体安装步骤如下：

1. 下载 thrift 源文件（<http://svn.apache.org/repos/asf/thrift/tags/thrift-0.6.1/>）
2. 将 thrift 源文件导入 eclipse，进入 /lib/java 目录，使用 ant 编译 build.xml 获得 libthrift-0.6.1-snapshot.jar
3. 将 libthrift-0.6.1-snapshot.jar、slf4j-api-1.5.8.jar、slf4j-log4j12-1.5.8.jar 和 log4j-1.2.14.jar 导入 eclipse 开发环境
4. 下载 thrift 编译工具，该工具可将 thrift 脚本文件编译成 java 文件，下载地址：[http://apache.etoak.com//thrift/0.6.0/thrift-0.6.1.exe](http://apache.etoak.com/thrift/0.6.0/thrift-0.6.1.exe)
5. 创建 Hello.thrift 脚本文件，具体代码如上一章节所述，进入 thrift-0.6.1.exe 所在目录，执行命令"thrift-0.6.1.exe -gen java x:\Hello.thrift"，在当前运行盘符下，可看见 gen-java 目录，进入目录可看到生成的 Java 代码。更多 thrift 的命令内容，请参考 thrift 自带的 help 命令
6. 编写服务端和客户端代码，完成 thrift 的安装和部署

基于 Apache Thrift 框架生成的服务包括客户端和服务器端，具体的部署模式如下所示：

**图 5. 部署图**  
 

从图中我们可以看到，客户端和服务器端部署时，需要用到公共的 jar 包和 java 文件，如图“Common file”区域，其中 Hello.java 由 Hello.thrift 编译而来。在服务器端，服务必须实现 Hello.Iface 接口，同时要包括服务器的启动代码 HelloServiceServer.java。在客户端，包括客户端调用服务的代码 HelloServiceClient.java。客户端和服务器通过 Hello.java 提供的 API 实现远程服务调用。