(http://www.wiquan.com/flyne)

Flyne (http://www.wiquan.com/flyne)

非知名技术控，自封『微圈』技术leader

动态 58 粉丝 22 ✚ 关注

常用RPC框架及一个简单的RPC框架实现

Flyne (/home/10001) 2016-05-08 18:07 3103 0

RPC（Remote Procedure Call）：远程过程调用，“过程”在Java中指的就是对象中的方法，“远程”是指不同机器上的进程（狭义），或者不同的进程（广义）（为了简单，下文不对这种情况进行说明）。因此，**RPC就是允许程序调用其他机器上的对象方法。**

RPC是属于典型的C/S结构，提供服务的一方称为server，请求服务的一方称为client。客户端和服务端可以运行在不同的机器上，Client只需要引入接口（一般指的业务接口），接口的实现以及运行时需要的数据都在Server端。

博主之前曾写过一篇文章《Hadoop RPC机制及HDFS源码分析》：http://www.flyne.org/article/1095 (http://www.flyne.org/article/1095)，详细介绍了RPC的概念和Hadoop RPC框架。因此本文不再介绍RPC的基础知识，只介绍RPC的几种实现技术，主要包括：

- **RMI**：JDK中自带的RPC实现
- **Hessian**（☆）：基于HTTP的RPC框架，非常轻量级
- **Dubbo**：基于TCP的RPC框架，阿里开源，被广泛应用于阿里集团的各成员站点
- **Thrift**：FaceBook开源，最初由FB开发用做系统内个语言之间的RPC通信（支持多种编程语言，

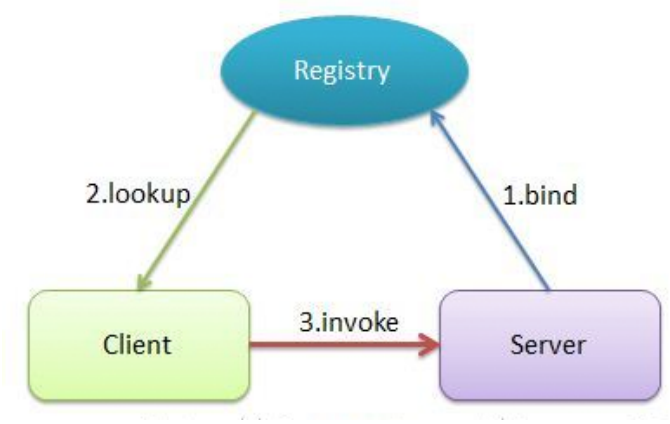
本文不讲解Thrift）

- **Webservice**：基于HTTP协议和XML数据的RPC技术，会在另外一篇文章单独讲
- **Hadoop RPC：**Hadoop框架中提供的RPC实现，在HDFS、Hadoop MR的主从通信中被广泛使用。具体参考《Hadoop RPC机制及HDFS源码分析》
- ……

RPC的主要依赖技术是序列化和传输协议。RMI的序列化和反序列化是JAVA内置的，传输则采用基于TCP的Socket编程；Hessian里的序列化机制是Hessian自带的，传输协议则是HTTP；Dubbo的序列化可以多种选择，一般使用Hessian的序列化机制，传输则是TCP协议，并使用了高性能的NIO框架Netty。

1、RMI

RMI的工作原理如下图：



1）创建RMI注册表服务（Registry），并向注册表绑定要提供的服务

2）客户端根据服务名获取服务

3）一旦客户端获取了服务，就可以调用服务方法，完成一次RMI（远程方法调用）

RMI的DEMO：

Demo中涉及的两个类说明如下：

- **Registry**：远程对象注册表，在提供远程服务之前，先要向RMI注册表注册服务。
- **LocateRegistry**：用于创建和获取"远程对象注册表"。其中，getRegistry方法用于获取特定主机（包括本地主机）上的远程对象注册表的引用，createRegistry方法用于创建一个"远程对象注册

表"。

服务接口和实现：

```
/**
 * 服务接口，封装了供远程调用的方法
 */
public interface HelloService extends Remote {
    // 服务方法
    String sayHello(String name) throws RemoteException;
}

public class HelloServiceImpl extends UnicastRemoteObject implements HelloService {

    // 底层采用JDK提供的序列化机制
    private static final long serialVersionUID = -6399513770403820034L;

    public HelloServiceImpl() throws RemoteException {
        super();
    }

    // 具体的服务方法实现
    public String sayHello(String name) throws RemoteException {
        return "Hello, " + name;
    }

}
```

注：

这里使用了服务接口的概念，而没有称之业务接口。区别在于业务接口是系统内部使用，如果该业务用来对外提供服务，就是服务接口。

RMI服务端

```
public class Server {
    public static void main(String[] args) {
        // RMI注册表
        Registry registry = null;

        try {
            registry = LocateRegistry.createRegistry(8888);

            // 创建一个服务
            HelloService helloService = new HelloServiceImpl();

            // 向RMI注册表注册服务，并将该服务命名为hello-service
            registry.bind("hello-service", helloService);
            System.out.println("> Ok, I could provice RPC service now~");
        } catch (Exception e) {

        }
    }
}
```

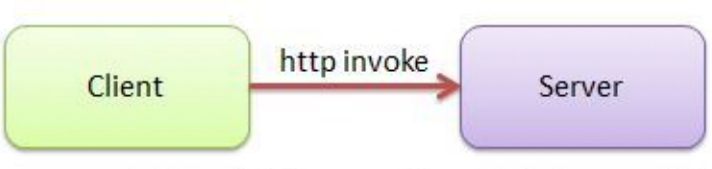
RMI客户端

```
public class Client {
    public static void main(String[] args) {
        // RMI注册表
        Registry registry = null;
        try {
            registry = LocateRegistry.getRegistry("localhost", 8888);

            // 根据服务名称获取服务
            HelloService helloService = (HelloService) registry.lookup("hello-service"
);

            // 调用远程方法
            System.out.println("RMI 服务器返回的结果是：" + helloService.sayHello("flyn
e"));
        } catch (Exception e) {

        }
    }
}
```

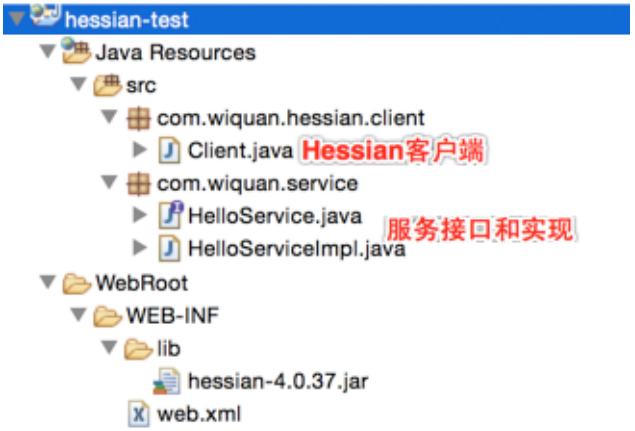
说明：本Demo中，RMI客户端和服务端在一个Java项目中，因此可以直接使用HelloService接口。但是大部分情况下，客户端和服务端都是不同的项目，此时可将服务接口和接口依赖的类打成jar包给客户端引用。（在Eclipse中，选中需要的类，右键 — export... ）

2、Hessian

基于HTTP协议传输，在性能方面还不够完美，负载均衡和失效转移依赖于应用的负载均衡器，Hessian的使用则与RMI类似，区别在于淡化了Registry的角色，通过显示的地址调用，利用HessianProxyFactory根据配置的地址create一个代理对象。如下图：



Hessian的DEMO：



说明：

1）使用Hessian需要引入Hessian的Jar包，可去官网下载最新的jar包：http://hessian.caucho.com/ (http://hessian.caucho.com/)

2）Hessian一般通过Web应用来提供服务，因此非常类似于 WebService

服务接口和实现：

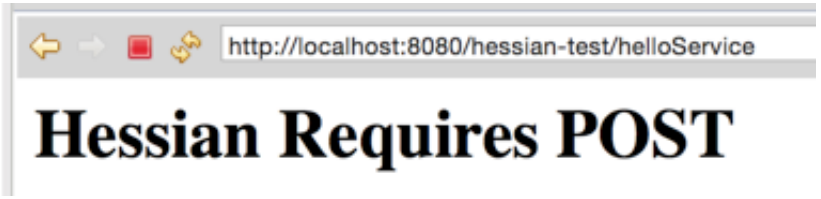
```
public interface HelloService {
    // 服务方法
    String sayHello(String name);
}

public class HelloServiceImpl implements HelloService {

    public String sayHello(String name) {
        return "Hello, " + name;
    }

}
```

web.xml中配置Hessian服务：

```
<servlet>
    <!-- 配置 HessianServlet -->
    <servlet-name>helloService</servlet-name>
    <servlet-class>com.caucho.hessian.server.HessianServlet</servlet-class>

    <!-- 配置接口的具体实现类 -->
    <init-param>
        <param-name>service-class</param-name>
        <param-value>com.wiquan.service.HelloServiceImpl</param-value>
    </init-param>
</servlet>
<!-- 映射 HessianServlet的访问URL地址 -->
<servlet-mapping>
    <servlet-name>helloService</servlet-name>
    <url-pattern>/helloService</url-pattern>
</servlet-mapping>
```

OK，在web.xml中配置好Hessian服务后，启动应用，并访问 http://localhost:8080/hessian-test/helloService (http://localhost:8080/hessian-test/helloService)

，如果显示结果如下，表明可以对外提供Hessian服务。



客户端

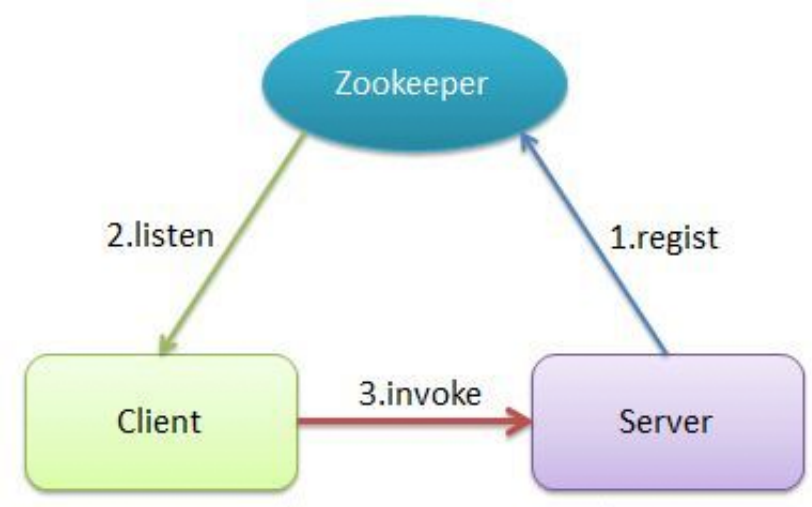
```
public class Client {
    public static void main(String[] args) {
        // 远程Hessian服务
        String url = "http://localhost:8080/hessian-test/helloService";
        try {
            // 利用hessianProxyFactory.create()方法创建一个代理对象
            HelloService helloService = (HelloService) (new HessianProxyFactory().crea
te(HelloService.class, url));

            // 调用服务方法
            System.out.println(helloService.sayHello("flyne"));
        } catch (MalformedURLException e) {

        }

    }
}
```

上面是一个Hessian的入门案例，在实际应用中，通常将Hessian和Spring整合使用。一个常见的情形就是：将DAO层的操作放在Hessian服务端，将业务层（Service层）放在Hessian客户端，DAO层通过Hessian通信向Service层提供服务，此时业务逻辑和数据操作就真正实现了完全的分离。

关于Hessian和Spring的整合，此处不再扩展。

3、Dubbo

Dubbo的工作原理如下图：



在了解Dubbo之前，要先对Zookeeper有深入的理解，当理解了zookeeper后，Dubbo也就了无秘密了。

在学习Dubbo之前，强烈推荐大家看这篇文章《轻量级分布式RPC框架》：http://my.oschina.net/huangyong/blog/361751?fromerr=aHKWUyYX (http://my.oschina.net/huangyong/blog/361751?fromerr=aHKWUyYX) 良心推荐。

4、一个简单的RPC框架的实现

本节使用Socket通信实现了一个简单的RPC框架（RPCFramework），并使用该框架模拟了RPC服务端和RPC客户端。主要类如下图：

通过该实例，可以深刻理解RPC通信原理（☆）。实例摘自Dubbo作者梁飞的博客：http://javatar.iteye.com/blog/1123915 (http://javatar.iteye.com/blog/1123915)

1）框架的核心类：RPCFramework

```
/**
 * 实现一个RPC框架，主要有两个功能：
 *     ① 供服务端暴露服务：export方法
 *     ② 供客户端引用服务：refer方法
 */
public class RPCFramework {
    /**
     * 暴露服务
     * @param service 服务实现
     * @param port 服务端口
     */
    @SuppressWarnings("resource")
    public static void export(final Object service, int port) throws IOException {
        // ① 创建serverSocket，等待客户端连接
        ServerSocket serverSocket = new ServerSocket(port);
        System.out.println("Export service " + service.getClass().getName() + " on por
t " + port);

        while (true) {
            // ② 对每一个客户端连接，新建一个线程进行处理
            final Socket socket = serverSocket.accept();
            new Thread(new Runnable() {
                public void run() {
                    try {
                        // ③ 依次解析客户端的请求参数（通信协议），包括服务方法、参数类型
数组、参数值数组
                        ObjectInputStream in = new ObjectInputStream(socket.getInputSt
ream());
                        String methodName = in.readUTF();
                        Class<?>[] parameterTypes = (Class<?>[]) in.readObject();
                        Object[] arguments = (Object[]) in.readObject();

                        // ④ 服务提供方根据请求参数，执行相应的服务方法，并将结果返回
                        ObjectOutputStream out = new ObjectOutputStream(socket.getOutp
utStream());
                        Method method = service.getClass().getMethod(methodName, param
eterTypes);
                        Object result = method.invoke(service, arguments);
                        out.writeObject(result);

                        out.close();
                        in.close();
                        socket.close();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }).start();
```

```
        }
    }

    /**
     * 引用服务
     * <T> 接口泛型
     * @param interfaceClass 接口类型
     * @param host RPC服务器主机名
     * @param port RPC服务器端口
     * @return 远程服务的代理对象
     */
    @SuppressWarnings("unchecked")
    public static <T> T refer(final Class<T> interfaceClass, final String host, final
int port) {
        System.out.println("Get remote service " + interfaceClass.getName() + " from s
erver " + host + ":" + port);

        // （☆）RPC客户端获取的是远程服务的代理对象
        return (T) Proxy.newProxyInstance(interfaceClass.getClassLoader(), new Class<?
>[] { interfaceClass },
                new InvocationHandler() {
                    // ① 每当客户端调用服务方法时，会触发invoke方法
                    public Object invoke(Object proxy, Method method, Object[] args) t
hrows Throwable {
                        // ② 和服务端建立连接
                        Socket socket = new Socket(host, port);

                        // ③ 向服务端发送请求参数（通信协议），包括服务方法、参数类型数组
、参数值数组
                        ObjectOutputStream out = new ObjectOutputStream(socket.getOutp
utStream());
                        out.writeUTF(method.getName());
                        out.writeObject(method.getParameterTypes());
                        out.writeObject(args);

                        // ④ 接收并返回服务端的返回值
                        ObjectInputStream in = new ObjectInputStream(socket.getInputSt
ream());
                        Object result = in.readObject();

                        in.close();
                        out.close();
                        socket.close();

                        return result;
                    }
                });
    }
}
```

2）服务接口及实现

```
public interface HelloService {
    String sayHello(String name);
}

public class HelloServiceImpl implements HelloService {
    public String sayHello(String name) {
        return "Hello, " + name;
    }
}
```

3）服务提供方（RPC服务端）

```
public class RPCProvider {
    public static void main(String[] args) throws Exception {
        HelloService service = new HelloServiceImpl();
        RPCFramework.export(service, 8888);
    }
}
```

4）服务调用方（RPC客户端）

```
public class RPCConsumer {
    public static void main(String[] args) throws InterruptedException {
        HelloService service = RPCFramework.refer(HelloService.class, "127.0.0.1", 888
8);

        int count = 10;
        while (count-- != 0) {
            System.out.println(service.sayHello("World" + count));
            Thread.sleep(100);
        }
    }
}
```

RPC调用的执行过程如下：

① 客户端获取远程服务的代理对象：这一过程由RPC框架完成，客户端只需指定服务接口类、服务提供方的主机名和端口号

② 当客户端调用服务方法时，会触发相应的代理方法（invoke）

③ 在代理方法中，首先建立和服务端的Socket连接

④ 客户端根据约定的通信协议发送请求参数：本实例中包括服务方法、参数类型数组、参数值数组

⑤ 服务端根据请求参数，执行相应的服务方法，并将结果放回：一定要注意RPC服务方法是在服务端被执行的！！！

⑥ 代理方法返回执行结果

3

评论是美德……

评论

暂无评论