day03 Dubbo

Apache Dubbo 是一款高性能、轻量级的开源Java RPC框架，提供面向接口代理的高性能RPC调用、智能负载均衡、服务自动注册和发现、运行期流量调度、可视化服务治理和运维等功能。

官网：

http://dubbo.apache.org/

1、RPC核心

RPC（Remote Procedure Call）—远程过程调用，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议，在面向对象的编程语言中，远程过程调用即是远程方法调用

RPC调用过程

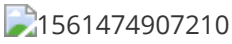
java中RPC框架比较多，常见的有RMI、Hessian、Thrift、gRPC、bRPC、motan、Dubbo等，其实对于RPC框架而言，核心模块就是通讯和序列化

接下来我们就分别看一下常见的RPC框架

- RMI

1）RMI(remote method invocation)是java原生支持的远程调用，RMI采用JRMP（Java RemoteMessageing Protocol）作为通信协议，可以认为是纯java版本的分布式远程调用解决方案。

2）RMI的核心概念



3）RMI步骤

1. 创建远程接口， 并且继承**java.rmi.Remote**接口
2. 实现远程接口，并且继承：**UnicastRemoteObject**
3. 创建服务器程序： **createRegistry()**方法注册远程对象
4. 创建客户端程序 (获取注册信息，调用接口方法)

常见实体类（注意实体类Serializable）

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User implements Serializable {

    private String name;
    private int age;
    private String sex;

}
```

```
public interface UserService extends Remote {

    String sayHello(String name) throws RemoteException;

}
```

提供接口的实现类

```
public class UserServiceImpl extends UnicastRemoteObject implements UserService
{

    public UserServiceImpl() throws RemoteException{}

    @Override
    public String sayHello(String name) throws RemoteException{
        return "hello"+name;
    }
}
```

将本地服务暴露出去，供外部调用

```
        try {
            UserService userService = new UserServiceImpl();
            LocateRegistry.createRegistry(8888);
            //暴露服务
            Naming.bind("rmi://localhost:8888/UserService", userService);
            System.out.println("提供userService服务。。。。");
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (AlreadyBoundException e) {
            e.printStackTrace();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
```

客户端远程调用服务，客户端需要依赖服务接口

```
userService = (UserService) Naming.lookup("rmi://localhost:8888/UserService");
//userService代理对象
System.out.println(userService);

System.out.println(userService.sayHello("传智小月"));
```

- Hessian

Hessian使用C/S方式，基于HTTP协议传输，使用Hessian二进制序列化。

server端：

添加hessian的maven依赖

```
            <groupId>com.caucho</groupId>
            <artifactId>hessian</artifactId>
            <version>4.0.7</version>
        </dependency>
```

创建接口UserService

```
public interface UserService  {

    String sayHello(String name);

}
```

实现类

```
public class UserServiceImpl implements UserService {

    @Override
    public String sayHello(String name){
        return "hello"+name;
    }
}
```

web.xml中配置HessianServlet

```
  <servlet>
    <servlet-name>HessianServlet</servlet-name>
    <servlet-class>com.caucho.hessian.server.HessianServlet</servlet-class>
    <init-param>
      <param-name>service-class</param-name>
      <param-value>com.itheima.demo.impl.UserServiceImpl</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>HessianServlet</servlet-name>
    <url-pattern>/api/service</url-pattern>
  </servlet-mapping>
```

添加tomcat7插件启动服务

```
<build>
        <plugins>
            <plugin>
                    <groupId>org.apache.tomcat.maven</groupId>
                    <artifactId>tomcat7-maven-plugin</artifactId>
                    <version>2.2</version>
                    <configuration>
                      <port>8888</port>
                      <path>/</path>
                      <uriEncoding>UTF-8</uriEncoding>
                    </configuration>
                  </plugin>
        </plugins>
```

客户端：

添加hessian的maven依赖

```
      <dependency>
            <groupId>com.caucho</groupId>
            <artifactId>hessian</artifactId>
            <version>4.0.7</version>
        </dependency>
```

创建跟server端相同的接口UserService(同上)

创建测试类测试

```
public class ClientTest {

    public static void main(String[] args) throws MalformedURLException {
        String url = "http://localhost:8888/api/service";

        HessianProxyFactory factory = new HessianProxyFactory();
        UserService api = (UserService) factory.create(UserService.class, url);
        System.out.println(api.sayHello("黑马程序员"));
    }
}
```

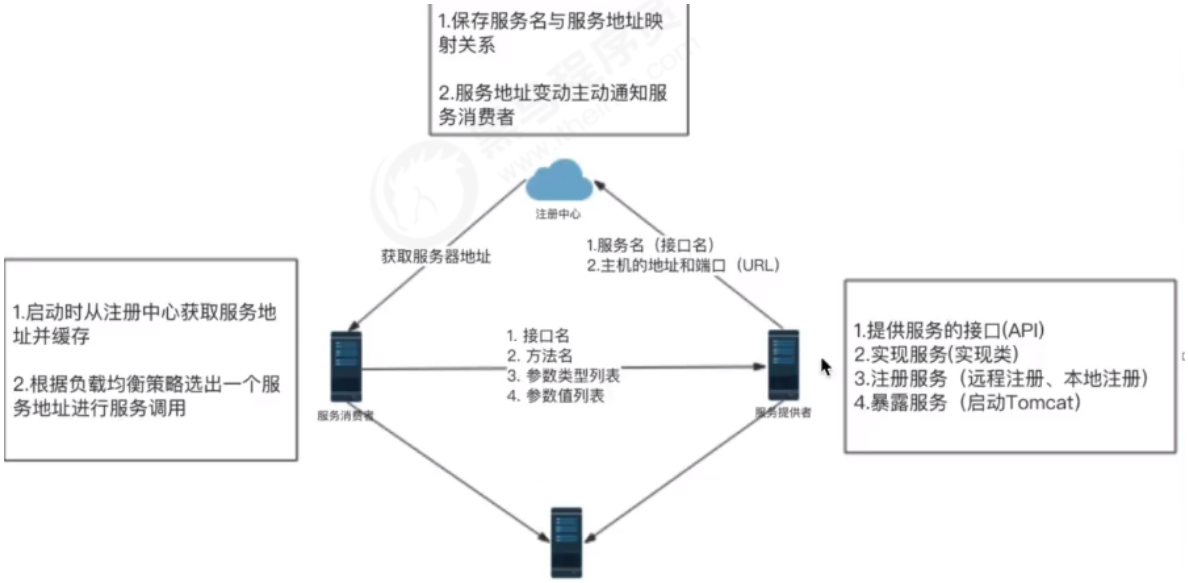
运行结果如下：

```
"C:\Program Files\Java\jdk1.8.0_161\bin\java.exe" ...
hello黑马程序员
```

- Thrift：FaceBook开源RPC框架，典型的CS架构，支持跨语言，客户端和服务端可以使用不同的语言开发，thrift通过IDL(Interface Description Language)来关联客户端和服务端。
- gRPC google
- dubbo

2、手写RPC框架

基本实现思路：



- registry注册
- protocol协议

服务提供者：

1、定义服务接口和实现类

接口HelloService

```
public interface HelloService {

   void sayHello(String message);

}
```

实现类HelloServiceImpl

```
public class HelloServiceImpl implements HelloService {

    @Override
    public void sayHello(String message) {
        System.out.println(message);
    }
}
```

2、服务注册

此处注册中心我们将服务注册在map集合中，结构:Map<String,Map<URL,Class>> 外边map的key存储服务接口的全类名,URL封装了调用服务的ip和port,里边value指定指定具体实现类的全类名

注册中心类提供注册服务并暴露服务和发现服务功能：

```
  private static Map<String,Map<URL,Class>> REGISTER = new HashMap<String,
Map<URL, Class>>();

    /**
     * 注册服务（暴露接口）
     * @param url
     * @param interfaceName
     * @param implClass
     */
    public static void regist(URL url,String interfaceName,Class implClass){
        Map<URL,Class> map = new HashMap<URL, Class>();
        map.put(url,implClass);
        REGISTER.put(interfaceName,map);
    }

    /**
     * 从注册中心获取实现类（发现服务）
     * @param url
     * @param interfaceName
     * @return
     */
    public static Class get(URL url,String interfaceName){
        return REGISTER.get(interfaceName).get(url);
```

URL类定义：

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class URL {
    private String hostname;
    private Integer port;
}
```

启动服务提供类注册服务并暴露服务

```
public class ServiceProvider {
    public static void main(String[] args) {
        // 注册服务
        URL url = new URL("localhost",8080);
        Register.regist(url, HelloService.class.getName(),
HelloServiceImpl.class);

        // 启动tomcat暴露服务
        HttpServer httpServer = new HttpServer();
        httpServer.start(url.getHostname(),url.getPort());
    }
}
```

3、protocol协议

服务之间调用的通信协议采用http协议，所以在服务provider中启动tomcat暴露服务

添加内嵌tomcat的依赖

```
        <!--内嵌tomcat-->
        <dependency>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-core</artifactId>
            <version>9.0.12</version>
        </dependency>
```

创建HttpServer

```
public class HttpServer {

    /**
     * tomcat服务启动
     * 参考tomcat配置
     * <Server port="8005" shutdown="SHUTDOWN">
     *  <Service name="Catalina">
     *      <Connector port="8080" protocol="HTTP/1.1"
     *                 connectionTimeout="20000"
     *                 redirectPort="8443"
     *          URIEncoding="UTF-8"/>
     *      <Engine name="Catalina" defaultHost="localhost">
     *          <Host name="localhost"  appBase="webapps"
```

```
     *              </Host>
     *      </Engine>
     *   </Service>
     * </Server>
     */

    public void start(String hostname,Integer port){

        // 实例一个tomcat
        Tomcat tomcat = new Tomcat();

        // 构建server
        Server server = tomcat.getServer();

        // 获取service
        Service service = server.findService("Tomcat");

        // 构建Connector
        Connector connector = new Connector();
        connector.setPort(port);
        connector.setURIEncoding("UTF-8");

        // 构建Engine
        Engine engine = new StandardEngine();
        engine.setDefaultHost(hostname);

        // 构建Host
        Host host = new StandardHost();
        host.setName(hostname);

        // 构建Context
        String contextPath = "";
        Context context = new StandardContext();
        context.setPath(contextPath);
        context.addLifecycleListener(new Tomcat.FixContextListener());// 生命周期
监听器

        // 然后按照server.xml，一层层把子节点添加到父节点
        host.addChild(context);
        engine.addChild(host);
        service.setContainer(engine);
        service.addConnector(connector);
        // service在getServer时就被添加到server节点了

        // tomcat是一个servlet,设置路径与映射
        tomcat.addServlet(contextPath,"dispatcher",new DispatcherServlet());
        context.addServletMappingDecoded("/*","dispatcher");

        try {
            tomcat.start();// 启动tomcat
            tomcat.getServer().await();// 接受请求
        }catch (LifecycleException e){
            e.printStackTrace();
        }
    }
}
```

```
public class HttpServerHandler {

    public void handler(HttpServletRequest req, HttpServletResponse resp){
        try{
            // Http请求流转为对象
            InputStream is = req.getInputStream();
            ObjectInputStream ois = new ObjectInputStream(is);
            Invocation invocation = (Invocation)ois.readObject();

            // 寻找注册中心的实现类，通过反射执行方法
            Class implClass = NativeRegister.get(new
URL("localhost",8080),invocation.getInterfaceName());
            Method method =
implClass.getMethod(invocation.getMethodName(),invocation.getParamTypes());
            String result = (String)
method.invoke(implClass.newInstance(),invocation.getParams());

            // 将结果返回
            IOUtils.write(result,resp.getOutputStream());
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

封装调用参数Invocation

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Invocation implements Serializable {

    private String interfaceName;
    private String methodName;
    private Object[] params;
    private Class[] paramTypes;

}
```

4、consumer服务消费端

封装HttpClient对象，发起远程调用

```
public class HttpClient{

        /**
         * 远程方法调用
         * @param hostname 远程主机名
         * @param port 远程主机端口
         * @param invocation
         * @return
         */
```

```
            try {
                // 进行http连接
                URL url = new URL("http",hostname,port,"/client/");
                HttpURLConnection connection =
(HttpURLConnection)url.openConnection();
                connection.setRequestMethod("POST");
                connection.setDoOutput(true);// 必填项

                // 将对象写入输出流
                OutputStream os = connection.getOutputStream();
                ObjectOutputStream oos = new ObjectOutputStream(os);
                oos.writeObject(invocation);
                oos.flush();
                oos.close();

                // 将输入流转为字符串（此处可是java对象）
                InputStream is = connection.getInputStream();
                return IOUtils.toString(is);
            }catch (MalformedURLException e){
                e.printStackTrace();
            }catch (IOException e){
                e.printStackTrace();
            }
            return null;
        }
}
```

发送rpc调用测试

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

        // 调用哪个方法
        Invocation invocation = new Invocation(
                HelloService.class.getName(),
                "sayHello2",
                new Object[]{"学IT,来黑马"},
                new Class[]{String.class});

        // 发现服务器
        String result = new HttpClient().post("localhost", 8080, invocation);
        System.out.println(result);
    }
```

###