|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 手写soa框架流程： | | | | | | | | |
| 1、自定义spring标签：src/main/resources下建立 META-INF包，META-INF包中创建 soa.xsd约束文件，spring.handlers文件，spring.schemas文件； | | | | | | | | |
| <?xml version="1.0" encoding="UTF-8"?>  <xsd:schema xmlns="http://www.dongnaoedu.com/schema/soa"  xmlns:xsd="http://www.w3.org/2001/XMLSchema"  targetNamespace="http://www.dongnaoedu.com/schema/soa"  elementFormDefault="qualified" attributeFormDefault="unqualified">  <xsd:element name="registry">  <xsd:complexType>  <xsd:attribute name="protocol" type="xsd:string"></xsd:attribute>  <xsd:attribute name="address" type="xsd:string"></xsd:attribute>  </xsd:complexType>  </xsd:element>  <xsd:element name="reference">  <xsd:complexType>  <xsd:attribute name="id" type="xsd:string"></xsd:attribute>  <xsd:attribute name="interface" type="xsd:string"></xsd:attribute>  <xsd:attribute name="loadbalance" type="xsd:string"></xsd:attribute>  <xsd:attribute name="protocol" type="xsd:string"></xsd:attribute>  </xsd:element>  <xsd:element name="protocol">  <xsd:complexType>  <xsd:attribute name="name" type="xsd:string"></xsd:attribute>  <xsd:attribute name="port" type="xsd:string"></xsd:attribute>  <xsd:attribute name="host" type="xsd:string"></xsd:attribute>  <xsd:attribute name="contextpath" type="xsd:string"></xsd:attribute>  </xsd:complexType>  </xsd:element>  <xsd:element name="service">  <xsd:complexType>  <xsd:attribute name="interface" type="xsd:string"></xsd:attribute>  <xsd:attribute name="ref" type="xsd:string"></xsd:attribute>  <xsd:attribute name="protocol" type="xsd:string"></xsd:attribute>  </xsd:complexType>  </xsd:element>  </xsd:schema> | | | | spring.handlers:定义命名空间对应的解析类  http\://www.dongnaoedu.com/schema/soa  =  com.dongnao.jack.spring.parse.SOANamespaceHandler | | | spring.schemas:定义soa.xsd文件位置  http\://www.dongnaoedu.com/schema/soa.xsd=META-INF/soa.xsd | |
| 2、定义spring解析类，定义 com.dongnao.jack.spring.parse.SOANamespaceHandler extends NamespaceHandlerSupport  package com.dongnao.jack.spring.parse;  import org.springframework.beans.factory.xml.NamespaceHandlerSupport;  import com.dongnao.jack.configBean.Protocol;  import com.dongnao.jack.configBean.Reference;  import com.dongnao.jack.configBean.Registry;  import com.dongnao.jack.configBean.Service;public class SOANamespaceHandler extends NamespaceHandlerSupport {  public void init() {  registerBeanDefinitionParser("registry",//3、registry为xsd元素名称，通过解析，将soa.xsd中registry元素内容封装到Registry  new RegistryBeanDefinitionParse(Registry.class));//4、定义parser类 \*BeanDefinitionParse(\*.class)  registerBeanDefinitionParser("protocol",  new ProtocolBeanDefinitionParse(Protocol.class));  registerBeanDefinitionParser("reference",  new ReferenceBeanDifinitionParse(Reference.class));  registerBeanDefinitionParser("service", new ServiceBeanDefinitionParse(  Service.class));  }  } | | | | | | | | |
| 3、定义 soa.xsd中元素名称对应的类 | | | | | | | | |
| package com.dongnao.jack.configBean;  public class Registry{  private String protocol;  private String address;  setter,getter省略  } | package com.dongnao.jack.configBean;  public class Protocol{  private String name;  private String port;  private String host;  private String contextpath;  setter,getter省略  } | | | | Service同左 | | Reference同左 | 同左 |
| 4、定义标签解析类 \*BeanDefinitionParse implements BeanDefinitionParse | | | | | | | | |
| package com.dongnao.jack.spring.parse;  import org.springframework.beans.factory.config.BeanDefinition;  import org.springframework.beans.factory.support.RootBeanDefinition;  import org.springframework.beans.factory.xml.BeanDefinitionParser;  import org.springframework.beans.factory.xml.ParserContext;  import org.w3c.dom.Element;  /\*\*  \* @Description Registry标签的解析类  \*/  public class RegistryBeanDefinitionParse implements BeanDefinitionParser {  private Class<?> beanClass;  public RegistryBeanDefinitionParse() {  }  public RegistryBeanDefinitionParse(Class<?> beanClass) {  this.beanClass = beanClass;/\*注册beanClass\*/  }  public BeanDefinition parse(Element element, ParserContext parserContext) {  RootBeanDefinition beanDefinition = new RootBeanDefinition();  beanDefinition.setBeanClass(beanClass);//将反射类对象交给beanDefiniton，beanClass会被spring实例化  beanDefinition.setLazyInit(false);  //从soa.xsd文件中读取到内存中  String protocol = element.getAttribute("protocol");  String address = element.getAttribute("address");  if (protocol == null || "".equals(protocol)) {  throw new RuntimeException("Regitry protocol 不能为空！");  }  if (address == null || "".equals(address)) {  throw new RuntimeException("Regitry address 不能为空！");  }  //从内存中保存到类中 beanDefinition.getPropertyValues().addPropertyValue("protocol",  protocol); //调用beanDefinition里面的反射类进行属性赋值 beanDefinition.getPropertyValues().addPropertyValue("address", address);  return beanDefinition;  }  } | | | package com.dongnao.jack.configBean;  public class Protocol{  private String name;  private String port;  private String host;  private String contextpath;} | | package com.dongnao.jack.configBean;  public class Service {  private String intf;  private String ref;  private String protocol;} | package com.dongnao.jack.configBean;  public class Reference {  private String id;  private String intf;/\*接口名称的字符串\*/  private String loadbalance;  private String protocol;  } | | 同左 |
| 5、定义xml文件引入自定义标签进行测试，src/main/resources中创建 mytest.xml  <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:jack="http://www.dongnaoedu.com/schema/soa" //定义以jack为前缀 \*.soa为uri的命名空间xmlns  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-3.2.xsd  http://www.dongnaoedu.com/schema/soa //以jack为前缀的命名空间xmlns  <http://www.dongnaoedu.com/schema/soa.xsd>" //引入soa.xsd约束文件  default-lazy-init="true">  <!--beans：跟节点  xmlns：命名空间  xmlns:xxx=uri：以xxx为前缀的xmlnamespace,uri表示命名空间，就是一个唯一标识符，不重复即可  例，xmlns:jack=uri表示以jack为前缀的命名空间，<jack:xxx/>  <jack:xxx/>中的xxx表示 jack/uri 这个命名空间里的属性  xmlns=uri：表示默认命名空间时是beans，当使用默认命名空间时 <xxx/>xxx是beans命名空间里面的元素  xsi:schemaLocation:各命名空间的uri与xsd文件的对应关系，"x x.xsd y y.xsd z z.xsd" 以空格隔开  -->  <jack:registry protocol="redis" address="127.0.0.1:6379"></jack:registry>  <jack:protocol name="http" port="27017" host="127.0.0.1"></jack:protocol>  <jack:service interface="com.dongnao.jack.test.UserService" ref="userServiceImpl"></jack:service>  <jack:reference id="testServiceImpl" interface="com.dongnao.jack.test.service.TestService" loadbalance="random" protocol="http"></jack:reference></beans> | | | | | | | | |
| 6、定义service接口及实现类 | | | | | | | | |
| package com.dongnao.jack.test;  public interface UserService {  public String eat(String param);  } | | package com.dongnao.jack.test;  public class UserServiceImpl implements UserService {  public String eat(String param) {  return null;  }  } | | | | | | |
| 7、修改mytest.xml，转5  ①<bean id="userServiceImpl" class="com.dongnao.jack.test.UserServiceImpl"/>//新增，注册userServic..  ②<jack:service interface="com.dongnao.jack.test.UserService" ref="userServiceImpl" protocol="http"></jack:service>//修改，引用userServiceImpl，并指定interface为 \*. UserService | | | | | | | | |
| 8、创建一个 main(String[] args) 方法，启动spring，即ClassPathXmlApplicationContext构造方法读取 /src/main/resources/mytest.xml文件  package com.dongnao.jack.test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class MyTest {  public static void main(String[] args) {  ApplicationContext app = new ClassPathXmlApplicationContext(  "mytest.xml");  UserService userservice = app.getBean(UserService.class);  userservice.eat("xxxxx");  }  } | | | | | | | | |
| 9、打包，要想被别人引用，就要打包，maven打包命令：mvn clean install –Dmaven test.skip=true，注意，package只打包，不会把包保存到本地仓库中，install除了打包，还会将包保存到本地仓库； | | | | | | | | |
| 10、创建测试工程，在general/springmvc中 | | | | | | | | |
| springmvc的pom.xml中引入dn-jack-soa 包  <dependency>  <groupId>com.gouyan</groupId>  <artifactId>dn-jack-soa</artifactId>  <version>1.0-SNAPSHOT</version>  </dependency> | | springmvc resources中 定义jacksoaconsumer.xml与dn-jack-soa的配置文件相同和spring-dispatcher.xml文件 | | | |  | | |
| 11、dubbo框架是如何跟spring进行整合的，消费者创建的代理实例是如何创建的？？？  例如 消费者中 @Autowired  private UserService user，通过类型注入获得接口UserService的代理实例user | | | | | | | | |
| 12、消费者创建的代理实例是如何创建的 | | | | | | | | |
| 1. Reference，Service，Protocol，Registry类都 实现 序列化接口 Serializable，并添加随机序列编号属   private static final long serialVersionUID = 12345L; | | | | | | | | |
| 1. Reference中的接口的远程对象是如何创建的？   Reference类实现 FactoryBean接口，覆写getObject(),getObjetType(),isSingleton()方法 | | | | | | | | |
| 1. 定义jdk动态代理 com.dongnao.jack.proxy.advice.InvokeInvocationHandler   public class InvokeInvocationHandler implements InvocationHandler {…}，InvokeInvocationHandler会调用里面的 invoke()方法，这个invoke方法里面进行了 rpc 远程调用，rpc分类：http，rmi，netty； | | | | | | | | |
| 1. 为了解决 rpc 这三种远程调用，采用策略模式，策略模式即 接口对象调用接口方法(例如IService service;   service.do();)，只调用，不实现，具体实现交给接口实现子类，程序会进入到实现子类的实现方法中；因此，此处定义接口 com.dongnao.jack.invoke.Invoke; | | | | | | | | |
| 1. com.dongnao.jack.invoke;public interface invoke{String invoke(Invocation invocation);} | | | | | | | | |
| 1. 定义 com.dongnao.jack.invoke.Invocation; public class Invocation{…} | | | | | | | | |
| 1. 回看③，定义 private Invoke invoke;用于策略模式调用 invoke()代理方法；private Reference reference;   里面有调用者的所有信息，它的 getObject()方法返回成员变量 intf 的代理类；  public InvokeInvocationHandler(Invoke invoke,Reference reference) {  this.reference = reference;  this.invoke = invoke;  } | | | | | | | | |
| 1. 回看② getObject()方法   public Object getObject() throws Exception {  Object proxy = Proxy.newProxyInstance(this.getClass().getClassLoader(),  new Class<?>[]{Class.forName(intf)},  new InvokeInvocationHandler(this , **?** )//?应该为Invoke接口对象  );  return proxy;  }  <jack:reference id="testServiceImpl" interface="com.dongnao.jack.test.service.TestService" loadbalance="random" protocol="http">  消费者要调用 \*.TestService这个接口，这里采用http协议的模式调用，但是还可以设置为 rmi netty；  因此先在Reference中注册不同协议，注册到静态map中：  private static Map<String,Invoke> invokes = new HashMap<String,Invoke>();  static {  invokes.put("http",new HttpInvoke());  invokes.put("http",new RmiInvoke());  invokes.put("http",new NettyInvoke());  }  回看 Reference getObeject()方法：  首先定义成员变量：private Invoke invoke;//持有接口对象，接口对象.接口方法表示策略模式，此处只调用不实现，具体实现交给实现子类，程序会自动进入实现子类的实现方法中；  System.out.println("返回 intf 代理对象");  if (protocol != null && !"".equals(protocol)){  this.invoke = invokes.get(protocol);  }else{  /\*如果<jack:reference 中没有 protocol 属性，将去 <jack:protocol 中找\*/  /\*protocol对象在spring容器中，要从容器中获取实例，要拿到spring的上下文applicationContext，因此Reference要实现ApplicationContextAware接口，定义成员变量privateApplicationContextApplicationContext;覆写setApplicationContext(ApplicationContext applicationContext){this. applicationContext = applicationContext }方法 \*/  Protocol protocol =this.applicationContext.getBean(Protocol.class);/\*没有id，则根据类型获得bean\*/  if (protocol != null){  this.invoke = invokes.get(protocol.getName());  }else {  this.invoke = invokes.get("http");/\*否则默认http协议\*/  }  } | | | | | | | | |
| 1. 修改 Reference 中 isSingleton(){return true;} //true表示getObject()返回的是单实例 | | | | | | | | |
| 1. com.dongnao.jack.test.MyTest类中进行测试，看是否会进入到InvokeInvocationHandler类的invoke()方法   public class MyTest {  public static void main(String[] args) {  ApplicationContext app = new ClassPathXmlApplicationContext(  "mytest.xml");  UserService userservice = app.getBean(UserService.class);  String result = userservice.eat("xxxxx");  System.out.println(result);  }  } | | | | | | | | |
| 1. app.getBean(UserService.class)，调用 getBean(…)方法时会调用 Reference类里面的 getObject()方法，该方法里面调用jdk动态代理，其中InvocationHandler h会在代理字符串中去掉 invoke()方法，至此，成功进入invoke()方法，位置：com.dongnao.jack.proxy.advice. InvokeInvocationHandler类里面   在这个invoke()里面最终要调用多个远程的provider，在生产者启动时，就要在redis里面注册provider信息，怎么做，才能在启动的时候就注册呢？  Service类实现InitializingBean接口，实现afterPropertiesSet()方法，该方法在Service类加载到spring容器之后执行；  创建注册的包package com.dongnao.jack.registry;创建委托者类com.dongnao.jack.registry. BaseRegistryDelegate，委托给其他类去处理；  public class BaseRegistryDelegate {  public static void registry(String ref){  }  }  Service类中，afterPropertiesSet()方法调用委托类的注册方法，委托避免了Service与Registry的耦合  @Override  public void afterPropertiesSet() throws Exception {  BaseRegistryDelegate.registry(ref);  }  BaseRegistryDelegate要在spring容器中取得<jack:registry protocol="redis" address="127.0.0.1:6379"></jack:registry>中registry对象，先要获得应用上下文ApplicationContext，通过参数穿过来，修改registry(String ref)方法为registry(String ref, ApplicationContext ac) | | | | | | | | |
| 1. 上面registry()方法中，注册中心有多个该如何处理？   定义接口BaseRegistry  package com.dongnao.jack.registry;  import java.util.List;  import org.springframework.context.ApplicationContext;  public interface BaseRegistry {  public boolean registry(String param, ApplicationContext application);  }  其中，redis是其中一个注册中心，定义 RedisRegistry implements BaseRegistry，回到Registry类，增加成员变量：private Map<String,BaseRegistry> registryMap = new HashMap<String,BaseRegistry>();  回到BaseRegistryDelegate：/\*根据protocol在registryMap中获取处理类\*/  BaseRegistry registryBean = Registry.getRegistryMap().get(protocol);  在RedisRegistry类中如何处理呢？  <bean id="userServiceImpl" class="com.dongnao.jack.test.UserServiceImpl"/>  <jack:service interface="com.dongnao.jack.test.UserService" ref="userServiceImpl"/>  Redis注册中心中数据格式：  {"userServiceImpl":{  "host:port":{  protocol:{},  service:{}  },  "host:port":{  protocol:{/\*将protocol对象转换成JSON字符串，最后上传到redis中\*/},  service:{ /\*将service中ref对应的bean对象转换成JSON字符串，最后上传到redis中\*/}  }  }  }**用json来装，一个大括号{}代表一个JSONObject对象，一个一个先创建，最后根据关系往里面装**  一个接口对应多个生产者，其中host，port都封装在Protocol类中，可以通过应用上下文获取；  一个Service类对应一个<dubbo:service…/>标签  回到 RedisRegistry类:  /\*getBeansOfType(Service.class)获得Service类的所有实例\*/  Map<String, Service> services = applicationContext.getBeansOfType(Service.class);  Registry registry = applicationContext.getBean(Registry.class);  /\*获取注册标签registry的属性address\*/  String address = registry.getAddress();  /\*需要redis客户端连接redis服务端，并操作服务端\*/  创建 RedisApi类  继续上面代码：  RedisApi.createJedisPool(address);/\*表示已连接redis服务端\*/  for (Map.Entry<String,Service> entry : services.entrySet()){  if (entry.getValue().getRef().equals(ref)){  JSONObject jsonObject = new JSONObject();  jsonObject.put("protocol",jsonObject.toJSONString(protocol));  jsonObject.put("service",jsonObject.toJSONString(entry.getValue()));  JSONObject ipport = new JSONObject();  ipport.put(protocol.getHost() + ":" + protocol.getPort(),jsonObject);  /\*定义lpush()方法，目的：去重\*/  lpush(ipport,ref);//service注册到reigistry的redis上  }  } | | | | | | | | |
| 1. BasedRegistryDelegate类中定义getRegistry()方法获取注册信息   public static List<String> getRegistry(String id,  ApplicationContext application) {  Registry registry = application.getBean(Registry.class);  String protocol = registry.getProtocol();  BaseRegistry registryBean = registry.getRegistryMap().get(protocol);    return registryBean.getRegistry(id, application);  }  Reference实现InitializingBean接口，实现afterPropertiesSet()方法，该方法表示当Reference在spring实例化之后调用该方法：  @Override  public void afterPropertiesSet() throws Exception {  registryInfo = BaseRegistryDelegate.getRegistry(id, applicationContext);/\*id继承自父类\*/  System.out.println(registryInfo);  }  getRegistry()方法如下：  public static List<String> getRegistry(String id,  ApplicationContext applicationContext) {  Registry registry = applicationContext.getBean(Registry.class);  String protocol = registry.getProtocol();  BaseRegistry registryBean = registry.getRegistryMap().get(protocol);  return registryBean.getRegistry(id, applicationContext);  } | | | | | | | | |
| 1. 测试   package com.dongnao.jack.test;  public class MyTest {  public static void main(String[] args) {  ApplicationContext app = new ClassPathXmlApplicationContext(  "mytest.xml");  }  ClssPathXmlApplicationContext("mytest.xml")加载完这个 配置文件，service就已经注册到registry中了，reference也已经从registry中获取到了  **注意：redis.conf中 bind 127.0.0.1 表示只有本机可以访问，应该注释掉：# bind 127.0.0.1**  **注册结果：**  **127.0.0.1:6379> keys \* 查看redis中所有key**  **1) "userServiceImpl2"**  **2) "userServiceImpl"**  **3) "gy"**  **127.0.0.1:6379> get gy**  **"123"**  **127.0.0.1:6379> lrange userServiceImpl 0 -1 取出user\* 这个key中的所有内容**  **1) "{\"127.0.0.1:27017\":{\"protocol\":\"{\\\"contextpath\\\":\\\"\\\",\\\"host\\\":\\\"127.0.0.1\\\",\\\"name\\\":\\\"http\\\",\\\"port\\\":\\\"27017\\\"}\",\"service\":\"{\\\"intf\\\":\\\"com.dongnao.jack.test.UserService\\\",\\\"protocol\\\":\\\"\\\",\\\"ref\\\":\\\"userServiceImpl\\\"}\"}}"** | | | | | | | | |
| 13、回到 InvokeInvocationHandler 类，invoke()方法根据从registry获取的JSON字符串信息，生成接口代理对象  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  System.out.println("已经进入了代理");  /\*在这个invoke()里面最终要调用多个远程的provider\*/  /\*reference已经从registry中获取到了相应的JSON字符串，那么怎么调用呢\*/  /\*invocation对象主要用来封装 invoke()方法的method和args\*/  Invocation invocation = new Invocation();  invocation.setMethod(method);  invocation.setArgs(args);  invocation.setReference(this.reference);  String result = invoke.invoke(invocation);  return result; }  其中，invoke是接口对象，只调用，不实现，idea中 ctr + alt + B 进入实现子类HttpInvoke中  package com.dongnao.jack.invoke;  import java.util.List;  public class HttpInvoke implements Invoke {  public String invoke(Invocation invocation) {  /\*获取registry中的JSON字符串信息\*/  List<String> registryInfo = invocation.getReference().getRegistryInfo();  /\*负载均衡算法\*/  String loadbalance = invocation.getReference().getLoadbalance();  return null;  }  }  回到Reference类，定义负载均衡这个成员变量  private static Map<String, LoadBalance> loadBalances = new HashMap<String, LoadBalance>();  这里LoadBalance是一个接口，因此这是一个策略模式(接口对象调用接口方法)  static{  loadBalances.put("romdom", new RandomLoadBalance());  loadBalances.put("roundrob", new RoundRobinLoadBalance());  }  回到HttpInvoke类 | | | | | | | | |
| package com.dongnao.jack.invoke;  public class HttpInvoke implements Invoke {  public String invoke(Invocation invocation) {  /\*获取registry中的JSON字符串信息\*/  List<String> registryInfo = invocation.getReference().getRegistryInfo();  /\*负载均衡算法\*/  String loadbalance = invocation.getReference().getLoadbalance();  Reference reference = invocation.getReference();  LoadBalance loadbalanceBean = Reference.getLoadBalances().get(loadbalance);  NodeInfo nodeInfo = loadbalanceBean.doSelect(registryInfo);  /\*既然要远程调用，就要传一下参数给远程的生产者，这里 通过 json字符串 传递\*/  JSONObject sendparam = new JSONObject();  sendparam.put("methodName",invocation.getMethod().getName());  sendparam.put("methodParams",invocation.getArgs());  **/\*从远程生产者的工程的spring容器中，获取serviceId对应的服务层的实例\*/**  sendparam.put("serviceId",reference.getId());  sendparam.put("paramTypes",invocation.getMethod().getParameterTypes());  String url = "http://" + nodeInfo.getHost() + ":" + nodeInfo.getPort() + "/" + nodeInfo.getContextpath();  //创建 com.dongnao.jack.rpc.http. HttpRequest类，处理url  /\*调用对端的生产者的服务\*/  String redult = HttpRequest.sendPost(url,sendparam.toJSONString());  /\*生产者怎么接收请求呢？答：servlet，因此建立 \*.remote.servlet包，创建DispatcherServlet类 \*/ return null;  }  }  DispatcherServlet类，注意，其他项目使用该 DispatcherServlet类的时候，要在项目web.xml文件中定义<servlet>作映射；  完成DispatcherServlet类后，进行测试：   1. dn-jack-soa项目打包:clean mvn install，然后将该包写到其他项目的pom中 | | | | | | | | |
| 14、dubbo的http协议的缺点是，在生产者的web.xml中必须配置<servlet>映射到自定义Dispatcher  http协议实质：消费者将调用方法的信息(方法名，接口，参数等)通过json字符串通过http请求发送给生产者，生产者解析json字符串后，利用反射调用方法，方法的执行还是在生产者的虚拟机，而不是在消费者的虚拟机； | | | | | | | | |
| 15、第三次课，第二节  rmi通信协议：创建com.dongnao.jack.invoke. RmiInvoke类，rmi实质：底层是socket，只能用于java，不能跨语言；  定义 RmiUtil 类，用于启动 rmi；定义SoaRmi接口必须继承 Remote接口 | | | | | | | | |
| 16、netty协议，dubbo的默认协议  定义NettyInvoke类 016-第一课 | | | | | | | | |
| 17、集群容错 016-第二课(00:22:00)  集群容错：当生产者集群中某个生产者宕机时，规范消费者调用形式；建立com.dongnao.jack.cluster包，定义Cluster接口；  集群容错分类：①failover 自动切换到好的机器；②failfast 如果调用节点失败，就直接失败；③failsafe 如果失败了就忽略；  分别定义 FailoverClusterInvoke类，FailfastClusterInvoke类，FailsafeClusterInvoke类；  dubbo的xml配置文件中定义 <dubbo:reference retries=”3” cluster=”failover”/> cluster代表容错策略，retries代表出错后重试次数；  回到 InvokeInvocationHandler类invoke()方法，里面注释掉 String result = invoke.invoke(invocation);  增加：  invocation.setInvoke(invoke);  /\*获取集群配置\*/  Cluster cluster = Reference.getClusters().get(reference.getCluster());  String result = cluster.invoke(invocation);  return result; | | | | | | | | |
| 18、当访问压力增大时，需要增加生产者，但是此时，原来的生产者集群，消费者集群都已经启动了，消费者Reference类的private List<String> registryInfo = new ArrayList<String>();已经注册过了，怎么把新增加的生产者也注册进来呢？？？  答：需要redis发布与订阅，创建 RedisServerRegistry extends JedisPubSub 类，广播形式发布与订阅，跟activemq中的topic机制类似；生产者端发布消息，消费者端接收消息；  RedisApi类中定义publish()方法发布消息 ，subsribe()方法订阅消息；  Reference类跟Service类中afterPropertiesSet()方法，当spring创建完类对象，就执行该方法，用于数据更新；  Service类中：RedisApi.publish("channel" + ref,"里面内容跟redis节点内容一致");  Reference类中：RedisApi.subsribe("channel" + id,new RedisServerRegistry());如果往这个频道发布了这个消息，那么RedisServerRegistry这个类的 onMessage()方法自然就会收到消息； | | | | | | | | |
| 19、服务发现与剔除功能还要完善； | | | | | | | | |