

# 服务治理中间件 Dubbo 原理解析

斩秋

博客: [http://blog.csdn.net/quhongwei\\_zhanqiu](http://blog.csdn.net/quhongwei_zhanqiu)

## 前言:

此文档是从学习 dubbo 源码过程中的笔记中整理出来的，由于时间及能力原因，理解有误之处还请谅解，希望对大家学习使用 dubbo 有所帮助。

dubbo 是阿里基于开源思想 java 实现的服务治理中间件，目前除了阿里之外已有很多公司直接使用或者基于阿里开源版本扩展使用。代码托管于 github 上 <https://github.com/alibaba/dubbo>，要想学好用好 dubbo 请从 <http://alibaba.github.io/dubbo-doc-static/Home-zh.htm> 获取最权威的文档、问题解答、原理介绍等。

## 第一章：Dubbo 内核实现

### 一：SPI 简单介绍

Dubbo 采用微内核+插件体系，使得设计优雅，扩展性强。那所谓的微内核+插件体系是如何实现的呢！大家是否熟悉 spi(service provider interface) 机制，即我们定义了服务接口标准，让厂商去实现（如果不了解 spi 的请谷歌百度下），jdk 通过 ServiceLoader 类实现 spi 机制的服务查找功能。

JDK 实现 spi 服务查找：ServiceLoader

首先定义下示例接口

```
package com.example;

public interface Spi {
    boolean isSupport(String name);
    String sayHello();
}
```

ServiceLoader 会遍历所有 jar 查找 META-INF/services/com.example.Spi 文件

A 厂商提供实现

```
package com.a.example;

public class SpiAImpl implements Spi {
    public boolean isSupport(String name) {
        return "SPIA".equalsIgnoreCase(name.trim());
    }
    public String sayHello() {
        return "hello 我是厂商 A";
    }
}
```

在 A 厂商提供的 jar 包中的 META-INF/services/com.example.Spi 文件内容为：  
com.a.example.SpiAImpl #厂商 A 的 spi 实现全路径类名

B 厂商提供实现

```
package com.b.example;

public class SpiBImpl implements Spi {
    public boolean isSupport(String name) {
```

```

        return "SPIB".equalsIgnoreCase(name.trim());
    }
    public String syaHello() {
        return "hello 我是厂商 B";
    }
}

```

在 B 厂商提供的 jar 包中的 META-INF/services/com.example.Spi 文件内容为：  
com.b.example.SpiImpl #厂商 B 的 spi 实现全路径类名

ServiceLoader.load(Spi.class) 读取厂商 A、B 提供 jar 包中的文件，ServiceLoader 实现了 Iterable 接口可通过 while for 循环语句遍历出所有实现。

一个接口多种实现，就如策略模式一样提供了策略的实现，但是没有提供策略的选择，使用方可以根据 isSupport 方法根据业务传入厂商名来选择具体的厂商。

```

public class SpiFactory {
    //读取配置获取所有实现
    private static ServiceLoader spiLoader =
ServiceLoader.load(Spi.class);
    //根据名字选取对应实现
    public static Spi getSpi(String name) {
        for (Spi spi : spiLoader) {
            if (spi.isSupport(name) ) {
                return spi;
            }
        }
        return null;
    }
}

```

## 二：基于 SPI 思想 Dubbo 内核实现

Dubbo 微内核基于 spi 思想的实现的扩展机制

## SPI 接口定义

定义了@SPI 注解

```
public @interface SPI {  
    String value() default ""; //指定默认的扩展点  
}
```

只有在接口打了@SPI 注解的接口类才会去查找扩展点实现  
会依次从这几个文件中读取扩展点

META-INF/dubbo/internal/ //dubbo 内部实现的各种扩展都放在了  
这个目录

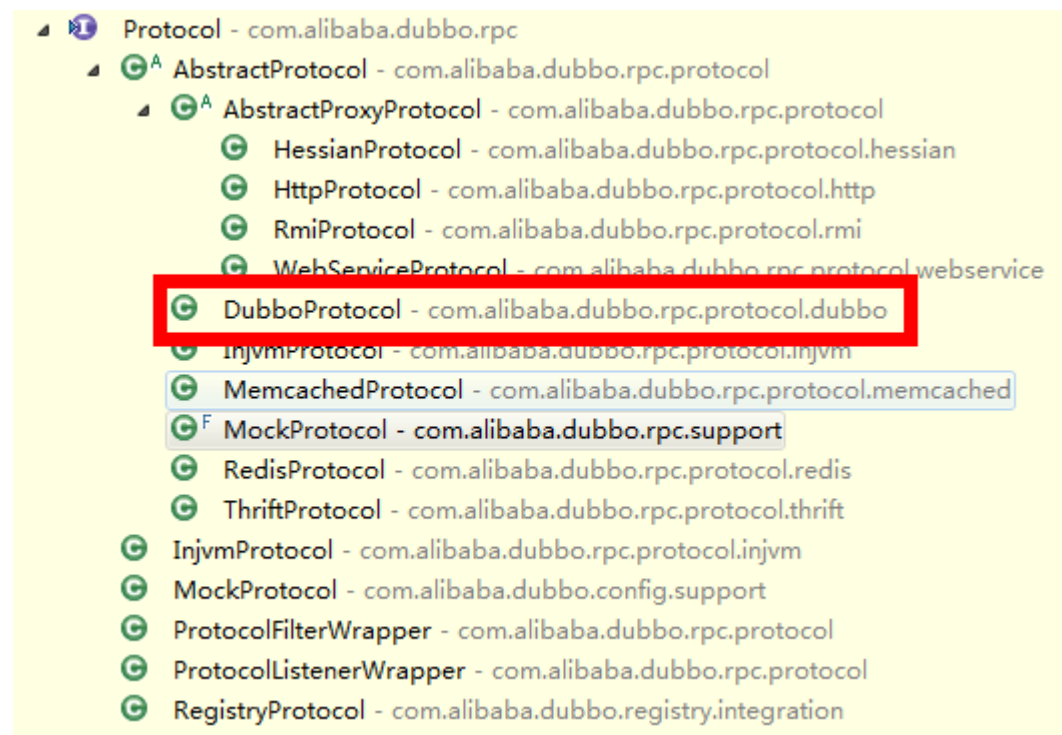
META-INF/dubbo/

META-INF/services/

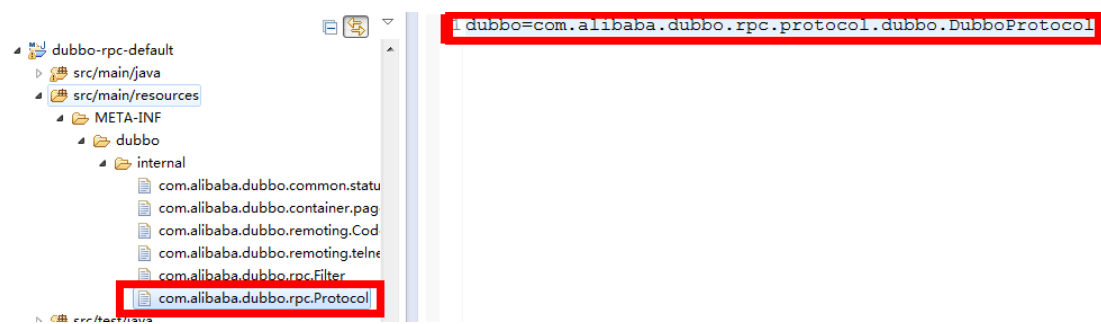
我们以 Protocol 接口为例， 接口上打上 SPI 注解，默认扩展点名字为 dubbo  
@SPI("dubbo")

```
public interface Protocol {  
}
```

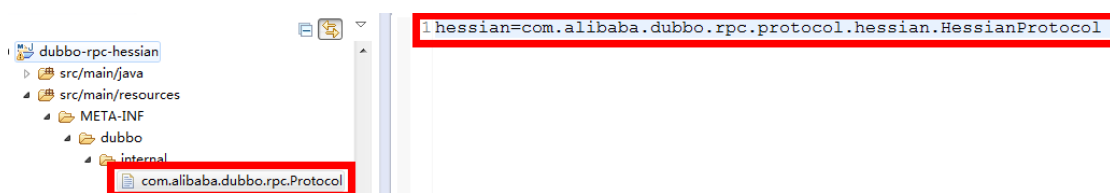
dubbo 中内置实现了各种协议如：DubboProtocol InjvmProtocol  
HessianProtocol WebServiceProtocol 等等



Dubbo 默认 rpc 模块默认 protocol 实现 DubboProtocol, key 为 dubbo



Dubbo rpc hessian 协议实现 HessianProtocol, key 为 hessian



其他协议提供方式雷同, 不在一一列举了。

## 下面我们来细讲 ExtensionLoader 类

1. `ExtensionLoader.get ExtensionLoader(Protocol.class)`  
每个定义的 spi 的接口都会构建一个 `ExtensionLoader` 实例, 存储在 `ConcurrentMap<Class<?>, ExtensionLoader<?>> EXTENSION_LOADERS` 这个 map 对象中
2. `loadExtensionClasses` 读取扩展点中的实现类
  - a) 先读取 SPI 注解的 value 值, 有值作为默认扩展实现的 key
  - b) 依次读取路径的文件  
`META-INF/dubbo/internal/ com.alibaba.dubbo.rpc.Protocol`  
`META-INF/dubbo/ com.alibaba.dubbo.rpc.Protocol`  
`META-INF/services/ com.alibaba.dubbo.rpc.Protocol`
3. `loadFile` 逐行读取 `com.alibaba.dubbo.rpc.Protocol` 文件中的内容, 每行内容以 key/value 形式存储的。
  - a) 判断类实现 (如: `DubboProtocol`) 上有没有打上 `@Adaptive` 注解, 如果打上了注解, 将此类作为 Protocol 协议的适配类缓存起来, 读取下一行; 否则适配类通过 `javassist` 修改字节码生成, 关于适配类功能作用后续

介绍

- b) 如果类实现没有打上@Adaptive， 判断实现类是否存在入参为接口的构造器（就是 DubboProtocol 类是否还有入参为 Protocol 的构造器），有的话作为包装类缓存到此 ExtensionLoader 的 Set<Class<?>>集合中，这个其实是个装饰模式

```
public class ProtocolFilterWrapper implements Protocol {  
  
    private final Protocol protocol;  
  
    public ProtocolFilterWrapper(Protocol protocol){  
        if (protocol == null) {  
            throw new IllegalArgumentException("protocol  
        }  
        this.protocol = protocol;  
    }  
}  
public class ProtocolListenerWrapper implements Protocol {  
  
    private final Protocol protocol;  
  
    public ProtocolListenerWrapper(Protocol protocol){  
        if (protocol == null) {  
            throw new IllegalArgumentException("protocol ==  
        }  
        this.protocol = protocol;  
    }  
}
```

- c) 如果即不是设配对象也不是 wrapped 的对象，那就是扩展点的具体实现对象

查找实现类上有没有打上 @Activate 注解，有缓存到变量 cachedActivates 的 map 中

将实现类缓存到 cachedClasses 中，以便于使用时获取

#### 4. 获取或者创建设配对象 getAdaptiveExtension

- a) 如果 cachedAdaptiveClass 有值，说明有且仅有一个实现类打了 @Adaptive，实例化这个对象返回
- b) 如果 cachedAdaptiveClass 为空， 创建设配类字节码。

为什么要创建设配类，一个接口多种实现，SPI 机制也是如此，这是策略模式，但是我们在代码执行过程中选择哪种具体的策略呢。Dubbo 采用统一数据模式 com.alibaba.dubbo.common.URL(它是 dubbo 定义的数据模型不是 jdk 的类)，它会穿插于系统的整个执行过程，URL 中定义的协议类型字段 protocol，会根据具体业务设置不同的协议。url.getProtocol() 值可以是 dubbo 也是可以 webservice， 可以是 zookeeper 也可以是 redis。

适配类的作用是根据 `url.getProtocol()` 的值 `extName`，去 `ExtensionLoader.getExtension(extName)` 选取具体的扩展点实现。

所以能够利用 `javasist` 生成适配类的条件

- 1) 接口方法中必须至少有一个方法打上了 `@Adaptive` 注解
- 2) 打上了 `@Adaptive` 注解的方法参数必须有 `URL` 类型参数或者有参数中存在 `getURL()` 方法

下面给出 `createAdaptiveExtensionClassCode()` 方法生成 `javasist` 用来生成 `Protocol` 适配类后的代码

```
import com.alibaba.dubbo.common.extension;

public class Protocol$Adaptive implements
com.alibaba.dubbo.rpc.Protocol {
    //没有打上@Adaptive 的方法如果被调到抛异常
    public void destroy() {
        throw new UnsupportedOperationException(
            "method public abstract void
com.alibaba.dubbo.rpc.Protocol.destroy() of interface
com.alibaba.dubbo.rpc.Protocol is not adaptive method!")
    }
    //没有打上@Adaptive 的方法如果被调到抛异常
    public int getDefaultPort() {
        throw new UnsupportedOperationException(
            "method public abstract int
com.alibaba.dubbo.rpc.Protocol.getDefaultPort() of interface
com.alibaba.dubbo.rpc.Protocol is not adaptive method!");
    }

    //接口中 export 方法打上@Adaptive 注册
    public com.alibaba.dubbo.rpc.Exporter export(
        com.alibaba.dubbo.rpc.Invoker arg0)
        throws com.alibaba.dubbo.rpc.Invoker {
        if (arg0 == null)
            throw new IllegalArgumentException(
                "com.alibaba.dubbo.rpc.Invoker argument == null");
        //参数类中要有 URL 属性
        if (arg0.getUrl() == null)
```



```

        throw new IllegalArgumentException(
            "com.alibaba.dubbo.rpc.Invoker      argument
getUrl() == null");
        //从入参获取统一数据模型 URL
        com.alibaba.dubbo.common.URL url = arg0.getUrl();
        String extName = (url.getProtocol() == null ? "dubbo" :
url.getProtocol());
        //从统一数据模型 URL 获取协议, 协议名就是 spi 扩展点实现类
        的 key
        if (extName == null) throw new IllegalStateException(
            "Fail to get extension(com.alibaba.dubbo.rpc.Protocol)
name from url("
        + url.toString() + ") use keys([protocol])");

        //利用 dubbo 服务查找机制根据名称找到具体的扩展点实现
        com.alibaba.dubbo.rpc.Protocol      extension      =
(com.alibaba.dubbo.rpc.Protocol)
ExtensionLoader.getExtensionLoader(com.alibaba.dubbo.rpc.Prot
ocol.class).getExtension(extName);
        //调具体扩展点的方法
        return extension.export(arg0);
    }

//接口中 refer 方法打上@Adaptive 注册
public  com.alibaba.dubbo.rpc.Invoker  refer(java.lang.Class
arg0,
        com.alibaba.dubbo.common.URL      arg1)      throws
java.lang.Class {
    //统一数据模型 URL 不能为空
    if (arg1 == null)
        throw new IllegalArgumentException("url == null");
    com.alibaba.dubbo.common.URL url = arg1;
    //从统一数据模型 URL 获取协议, 协议名就是 spi 扩展点实
    现类的 key
    String extName = (url.getProtocol() == null ? "dubbo" :

```

```

url.getProtocol());
    if (extName == null)
        throw new IllegalStateException("Fail to get extension (com. alibaba. dubbo. rpc. Protocol) name from url (" + url.toString() + ")
use keys ([protocol])");
    //利用 dubbo 服务查找机制根据名称找到具体的扩展点实现
    com. alibaba. dubbo. rpc. Protocol extension =
    (com. alibaba. dubbo. rpc. Protocol)
    ExtensionLoader.getExtensionLoader(com. alibaba. dubbo. rpc. Protocol.class)
    .getExtension(extName);
    //调具体扩展点的方法
    return extension.refer(arg0, arg1);
}
}

```

5. 通过 createAdaptiveExtensionClassCode() 生成如上的 java 源代码，要被 java 虚拟机加载执行必须得编译成字节码，dubbo 提供两种方式去执行代码的编译 1) 利用 JDK 工具类编译 2) 利用 javassist 根据源代码生成字节码。

```

private Class<?> createAdaptiveExtensionClass() {
    String code = createAdaptiveExtensionClassCode();
    ClassLoader classLoader = findClassLoader();
    com. alibaba. dubbo. common. compiler. Compiler compiler =
        ExtensionLoader.getExtensionLoader(com. alibaba. dubbo. common. compiler. Compiler.class)
        .getAdaptiveExtension();
    return compiler.compile(code, classLoader);
}

```

如上图：

- 1) 生成 Adaptive 代码 code
- 2) 利用 dubbo 的 spi 扩展机制获取 compiler 的设配类
- 3) 编译生成的 adaptive 代码

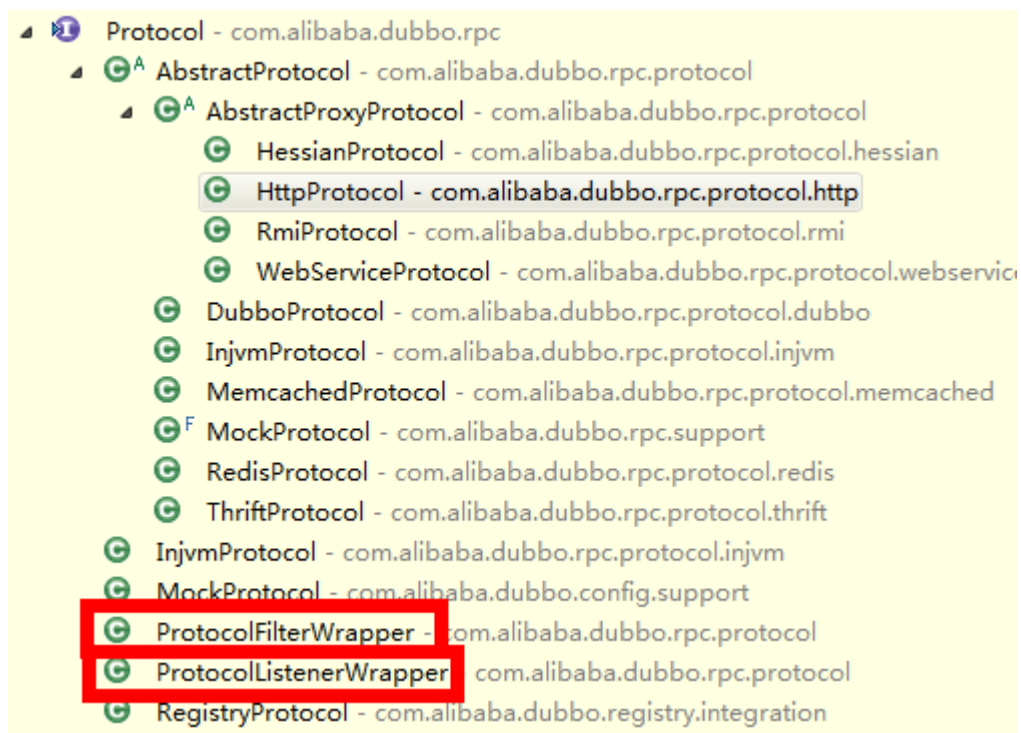
在此顺便介绍下 @Adaptive 注解打在实现类上跟打在接口方法上的区别

- 1) 如果有打在接口方法上，调 ExtensionLoader.getAdaptiveExtension() 获取设配类，会先通过前面的过程生成 java 的源代码，在通过编译器编译成 class 加载。但是 Compiler 的实现策略选择也是通过 ExtensionLoader.getAdaptiveExtension()，如果也通过编译器编译成 class 文件那岂不是要死循环下去了吗？
- 2) ExtensionLoader.getAdaptiveExtension()，对于有实现类上去打了注解 @Adaptive 的 dubbo spi 扩展机制，它获取设配类不在通过前面过程生成

设配类 java 源代码，而是在读取扩展文件的时候遇到实现类打了注解 @Adaptive 就把这个类作为设配类缓存在 ExtensionLoader 中，调用是直接返回

## 6. 自动 Wrap 上扩展点的 Wrap 类

这是一种装饰模式的实现，在 jdk 的输入输出流实现中有很多这种设计，在于增强扩展点功能。这里我们拿对于 Protocol 接口的扩展点实现作为实例讲解。



如图 Protocol 继承关系 ProtocolFilterWrapper, ProtocolListenerWrapper 这两个类是装饰对象用来增强其他扩展点实现的功能。ProtocolFilterWrapper 功能主要是在 refer 引用远程服务的中透明的设置一系列的过滤器链用来记录日志，处理超时，权限控制等功能；ProtocolListenerWrapper 在 provider 的 exporter, unporter 服务和 consumer 的 refer 服务，destory 调用时添加监听器，dubbo 提供了扩展但是没有默认实现哪些监听器。

Dubbo 是如何自动的给扩展点 wrap 上装饰对象的呢？

1) 在 ExtensionLoader.loadFile 加载扩展点配置文件的时候

```

try {
    clazz.getConstructor(type);
    Set<Class<?>> wrappers = cachedWrapperClasses;
    if (wrappers == null) {
        cachedWrapperClasses = new ConcurrentHashMap<Class<?>>();
        wrappers = cachedWrapperClasses;
    }
    wrappers.add(clazz);
}

```

对扩展点类有接口类型为参数的构造器就是包转对象，缓存到集合中去

2) 在调 ExtensionLoader 的 createExtension(name) 根据扩展点 key 创建扩展的时候，先实例化扩展点的实现，在判断时候有此扩展时候有包装类缓存，有的话利用包转器增强这个扩展点实现的功能。如下图是实现流程

```

private T createExtension(String name) {
    Class<?> clazz = getExtensionClasses().get(name);
    if (clazz == null) {
        throw findException(name);
    }
    try {
        T instance = (T) EXTENSION_INSTANCES.get(clazz);
        if (instance == null) {
            EXTENSION_INSTANCES.putIfAbsent(clazz, (T) clazz.newInstance());
            instance = (T) EXTENSION_INSTANCES.get(clazz);
        }
        injectExtension(instance);
        Set<Class<?>> wrapperClasses = cachedWrapperClasses;
        if (wrapperClasses != null && wrapperClasses.size() > 0) {
            for (Class<?> wrapperClass : wrapperClasses) {
                instance = injectExtension((T) wrapperClass.getConstructor(type).newInstance(instance));
            }
        }
        return instance;
    }
}

```

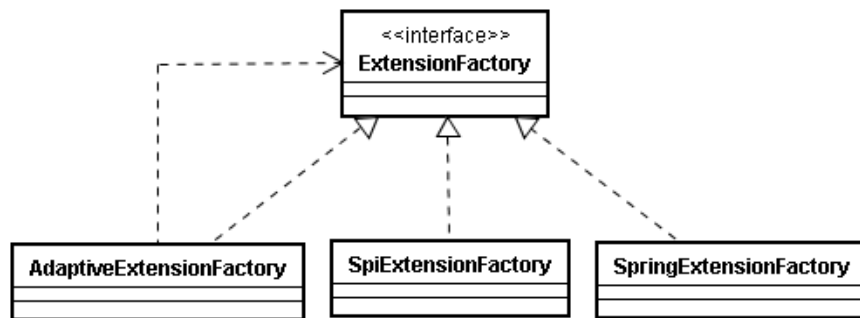
7. IOC 大家所熟知的 ioc 是 spring 的三大基础功能之一，dubbo 的 ExtensionLoader 在加载扩展实现的时候内部实现了个简单的 ioc 机制来实现对扩展实现所依赖的参数的注入，dubbo 对扩展实现中公有的 set 方法且入参参数为一个的方法，尝试从对象工厂 ObjectFactory 获取值注入到扩展点实现中去。

```

private T injectExtension(T instance) {
    try {
        if (objectFactory != null) {
            for (Method method : instance.getClass().getMethods()) {
                if (method.getName().startsWith("set")
                    && method.getParameterTypes().length == 1
                    && Modifier.isPublic(method.getModifiers())) {
                    Class<?> pt = method.getParameterTypes()[0];
                    try {
                        String property = method.getName().length() > 3 ? method.getName().substring(3, 4).toLowerCase() : "";
                        Object object = objectFactory.getExtension(pt, property);
                        if (object != null) {
                            method.invoke(instance, object);
                        }
                    } catch (Exception e) {
                        // ignore
                    }
                }
            }
        }
    }
}

```

上图代码应该不能理解，下面我们来看看 ObjectFactory 是如何根据类型和名字来获取对象的，ObjectFactory 也是基于 dubbo 的 spi 扩展机制的



它跟 Compiler 接口一样设配类注解 @Adaptive 是打在类 AdaptiveExtensionFactory 上的不是通过 javassist 编译生成的。

AdaptiveExtensionFactory 持有所有 ExtensionFactory 对象的集合, dubbo 内部默认实现的对象工厂是 SpiExtensionFactory 和 SpringExtensionFactory, 他们经过 TreeMap 排好序的查找顺序是优先先从 SpiExtensionFactory 获取, 如果返回空在从 SpringExtensionFactory 获取。

- 1) SpiExtensionFactory 工厂获取要被注入的对象, 就是要获取 dubbo spi 扩展的实现, 所以传入的参数类型必须是接口类型并且接口上打上了 @SPI 注解, 返回的是一个设配类对象。

```

public class SpiExtensionFactory implements ExtensionFactory {
    public <T> T getExtension(Class<T> type, String name) {
        if (type.isInterface() && type.isAnnotationPresent(SPI.class)) {
            ExtensionLoader<T> loader = ExtensionLoader.getExtensionLoader(type);
            if (loader.getSupportedExtensions().size() > 0) {
                return loader.getAdaptiveExtension();
            }
        }
        return null;
    }
}
  
```

- 2) SpringExtensionFactory, Dubbo 利用 spring 的扩展机制跟 spring 做了很好的融合。在发布或者去引用一个服务的时候, 会把 spring 的容器添加到 SpringExtensionFactory 工厂集合中去, 当 SpiExtensionFactory 没有获取到对象的时候会遍历 SpringExtensionFactory 中的 spring 容器来获取要注入的对象

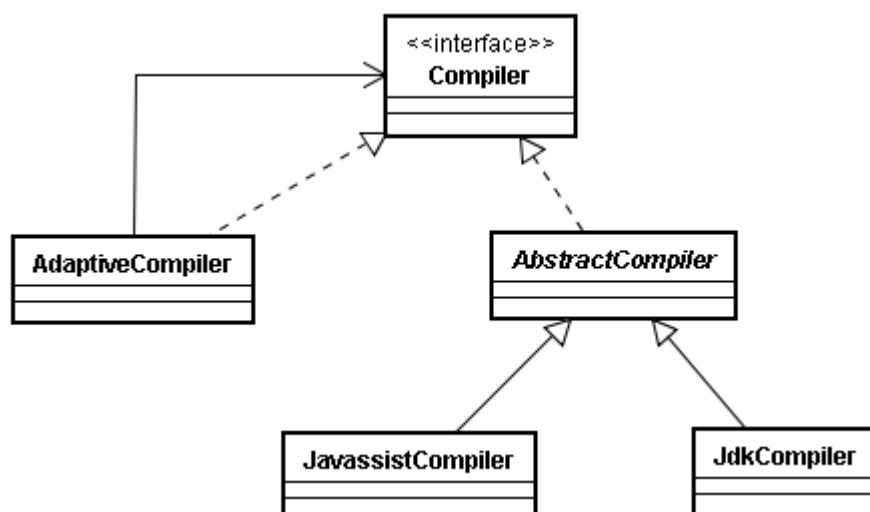
```

public <T> T getExtension(Class<T> type, String name) {
    for (ApplicationContext context : contexts) {
        if (context.containsBean(name)) {
            Object bean = context.getBean(name);
            if (type.isInstance(bean)) {
                return (T) bean;
            }
        }
    }
    return null;
}
  
```

### 三： 动态编译

我们运行的 java 代码，一般都是编译之后的字节码。Dubbo 为了实现基于 spi 思想的扩展特性，特别是能够灵活添加额外功能，对于扩展或者说是策略的选择这个叫做控制类也好设配类也好的类要能够动态生成。当然对应已知需求如 Protocol, ProxyFactory 他们的策略选择的设配类代码 dubbo 直接提供也无妨，但是 dubbo 作为一个高扩展性的框架，使得用户能够添加自己的需求，根据配置动态生成自己的设配类代码，这样就需要在运行的时候去编译加载这个设配类的代码。下面我们就是来了解下 Dubbo 的动态编译。

动态编译的实现的类图



#### 编译接口定义

```
@SPI("javassist")
public interface Compiler {
    Class<?> compile(String code, ClassLoader classLoader);
}
```

SPI 注解表示如果没有配置，dubbo 默认选用 javassist 编译源代码

接口方法 compile 第一个入参 code，就是 java 的源代码

接口方法 compile 第二个入参 classLoader，按理是类加载器用来加载编译后的字节码，其实没用到，都是根据当前线程或者调用方的 classLoader 加载的

```
@Adaptive
public class AdaptiveCompiler implements Compiler {
    private static volatile String DEFAULT_COMPILER;
```

```

    public Class<?> compile(String code, ClassLoader classLoader) {
        Compiler compiler;
        ExtensionLoader<Compiler> loader =
ExtensionLoader.getExtensionLoader(Compiler.class);
        String name = DEFAULT_COMPILER; // copy reference
        if (name != null && name.length() > 0) {
            compiler = loader.getExtension(name);
        } else {
            compiler = loader.getDefaultExtension();
        }
        return compiler.compile(code, classLoader);
    }
}

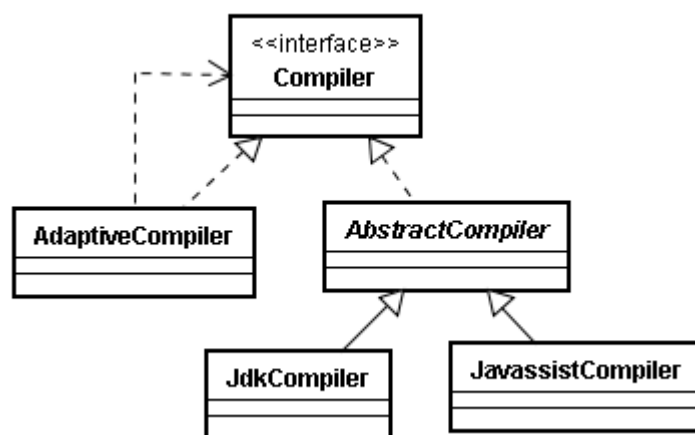
```

AdaptiveCompiler 是 Compiler 的设配类， 它有类注解@Adaptive 表示这个 Compiler 的设配类不是动态编译生成的。AdaptiveCompiler 作用就是策略的选择， 根据条件选择何种编译策略来编译动态生成的源代码。

AbstractCompiler 为编译的抽象类， 抽象出公用逻辑， 这里它主要是利用正则匹配出源代码中的包名和类名后先在 jvm 中 Class.forName 看下是否存在， 如果存在反回， 不存在在执行编译与加载。

关于 JavassistCompiler 和 JdkCompiler 执行 doCompile 的过程都是利用 Javassist 和 Jdk 提供的相关 api 或者扩展接口实现的。

1)



```

@SPI("javassist")
public interface Compiler {

    /**
     * Compile java source code.
     *
     * @param code Java source code
     * @param classLoader TODO
     * @return Compiled class
     */
    Class<?> compile(String code, ClassLoader classLoader);
}

@Adaptive
public class AdaptiveCompiler implements Compiler {

    private static volatile String DEFAULT_COMPILER;

    public static void setDefaultCompiler(String compiler) {
        DEFAULT_COMPILER = compiler;
    }

    public Class<?> compile(String code, ClassLoader classLoader) {
        Compiler compiler;
        ExtensionLoader<Compiler> loader = ExtensionLoader.getExtensionLoader(Compiler.class);
        String name = DEFAULT_COMPILER; // copy reference
        if (name != null && name.length() > 0) {
            compiler = loader.getExtension(name);
        } else {
            compiler = loader.getDefaultExtension();
        }
        return compiler.compile(code, classLoader);
    }
}

```

如果系统配置中没有给 AdaptiveCompiler 设置哪种 compiler，那么获取默认 compiler，默认策略根据打在接口 Compiler 上的 @SPI 值为 javassist，所以 dubbo 默认利用 javassist 生成 SPI 机制的设配用来根据统一数据模型 URL 中获取协议来选择使用何种策略

## 第二章：代理

代理模式这里不再逻辑介绍，dubbo 中有使用这种模式，如：dubbo 服务的消费端获取的就是对远程服务的一个代理。Dubbo 由代理工厂 ProxyFactory 对象创建代理对象

### 一：ProxyFactory 的接口定义

```
@SPI("javassist")
```

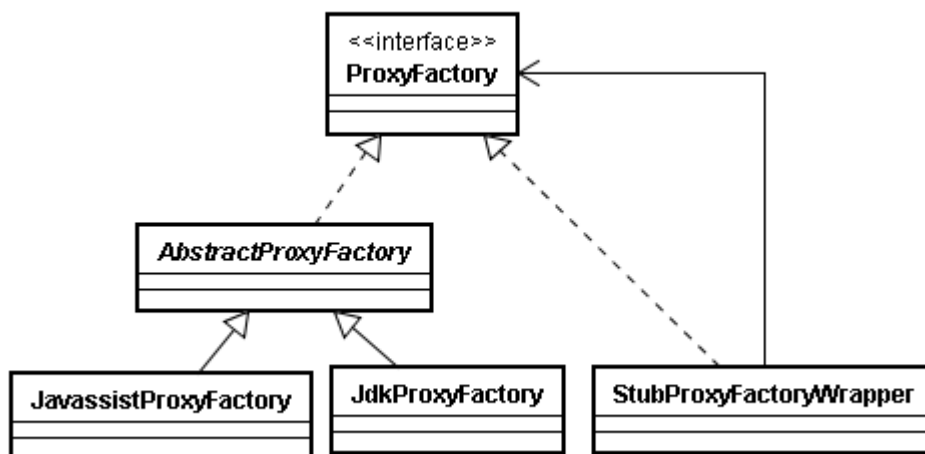


```

public interface ProxyFactory {
    @Adaptive({Constants.PROXY_KEY})
    <T> T getProxy(Invoker<T> invoker) throws RpcException;
    @Adaptive({Constants.PROXY_KEY})
    <T> Invoker<T> getInvoker(T proxy, Class<T> type, URL url) throws
    RpcException;
}

```

1. @SPI 指定默认使用 javassist 字节码技术来生成代理对象
2. 接口定义了生成代理对象的方法 getProxy, 入参是 invoker 对象
3. 接口定义了获取 invoker 对象, invoker 对象是个可执行对象, 这里 invoker 对象的 invoke 方法其实执行的是根据 url 获取的方法对第一个入参的实体对象的调用, 即: 如果 url 的得知调用方法 sayHello, 入参 proxy 为空 Test 对象实现 test, 那 invoker.invoke() 就是 test.sayHello()



**AbstractProxyFactory**: 代理工厂的公共抽象, 这里抽象实现主要是获取需要代理的接口, 代理接口可以在设置在 url 中 key 为 interfaces, 如果是多个接口用逗号分隔, 如果没有在 url 中指定, 代理 invoker 获取的和 EchoService 接口

```

public <T> T getProxy(Invoker<T> invoker) throws RpcException {
    Class<?>[] interfaces = null;
    String config = invoker.getUrl().getParameter("interfaces");
    if (config != null && config.length() > 0) {
        String[] types = Constants.COMMA_SPLIT_PATTERN.split(config);
        if (types != null && types.length > 0) {
            interfaces = new Class<?>[types.length + 2];
            interfaces[0] = invoker.getInterface();
            interfaces[1] = EchoService.class;
            for (int i = 0; i < types.length; i++) {
                interfaces[i + 1] = ReflectUtils.forName(types[i]);
            }
        }
    }
    if (interfaces == null) {
        interfaces = new Class<?>[] {invoker.getInterface(), EchoService.class};
    }
    return getProxy(invoker, interfaces);
}

```

JdkProxyFactory: 利用 jdk 动态代理来创建代理，实现来说比较简单

JDK 动态代理获取代理对象

```

public <T> T getProxy(Invoker<T> invoker, Class<?>[] interfaces) {
    return (T)
        Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(), interfaces, new InvokerInvocationHandler(invoker));
}

```

InvokerInvocationHandler 是 jdk 动态代理创建一定要构建的参数，这里它的 invoke 方法只是简单的调用了 invoker.invoke 方法，Invoker 在 dubbo 中代表一个可执行体，一切都向它靠拢。

获取 invoker 对象

```

public <T> Invoker<T> getInvoker(T proxy, Class<T> type, URL url) {
    return new AbstractProxyInvoker<T>(proxy, type, url) {
        @Override
        protected Object doInvoke(T proxy, String methodName,
            Class<?>[] parameterTypes, Object[] arguments) throws
            Throwable {
            Method method = proxy.getClass().getMethod(methodName,
                parameterTypes);
            return method.invoke(proxy, arguments);
        }
    };
}

```

这里创建的 Invoker 对象，执行 invoke 方法，其实就是利用反射利用入参执行对应对象的对应方法。

## 二: Javassist 字节码技术生成代理

JavassistProxyFactory: 利用字节码技术来创建对象

```

public <T> T getProxy(Invoker<T> invoker, Class<?>[] interfaces) {
    return (T) Proxy.getProxy(interfaces).newInstance(new
        InvokerInvocationHandler(invoker));
}

```

看似跟 jdk 生成代理一样，其实这里的 Proxy 类不是 jdk 中自带那个生成代理对象的类是 com.alibaba.dubbo.common.bytecode.Proxy。

这个 dubbo 自己写的 Proxy 类，利用要代理的接口利用 javassist 工具生成代理代码。

获取 Invoker 对象

```

public <T> Invoker<T> getInvoker(T proxy, Class<T> type, URL url) {
    final Wrapper wrapper =
        Wrapper.getWrapper(proxy.getClass().getName().indexOf('$') < 0 ? proxy.getClass() : type);
    return new AbstractProxyInvoker<T>(proxy, type, url) {
        protected Object doInvoke(T proxy, String methodName,
            Class<?>[] parameterTypes, Object[] arguments) throws
Throwable {
            return wrapper.invokeMethod(proxy, methodName,
                parameterTypes, arguments);
        }
    };
}

```

根据传入的 proxy 对象的类信息创建对它的包装对象 Wrapper

返回 Invoker 对象实例，这个 invoker 对象 invoke 方法可以根据传入的 invocation 对象中包含的方法名，方法参数来调用 proxy 对象返回调用结果

com.alibaba.dubbo.common.bytecode.Proxy 生成代理对象的工具类

1. 遍历所有入参接口，以；分割连接起来，以它为 key 以 map 为缓存查找如果有，说明代理对象已创建返回
2. 利用 AtomicLong 对象自增获取一个 long 数组来作为生产类的后缀，防止冲突
3. 遍历接口获取所有定义的方法，加入到一个集合 Set<String> worked 中，用来判重，

获取方法 y 应该在 methods 数组中的索引下标 ix

获取方法的参数类型以及返回类型

构建方法体 return ret= handler.invoke(this, methods[ix], args);这里的方法调用其实是委托给 InvokerInvocationHandler 实例对象的，去调用

真正的实例

方法加入到 methods 数组中

#### 4. 创建代理实例对象 ProxyInstance

类名为 pkg + “.poxy” + id = 包名 + “.poxy” + 自增数值

添加静态字段 Method[] methods;

添加实例对象 InvokerInvocationHandler hanler

添加构造器参数是 InvokerInvocationHandler

添加无参构造器

利用工具类 ClassGenerator 生成对应的字节码

#### 5. 创建代理对象，它的 newInstance(handler) 方法用来创建基于我们接口的代理

```
String fcn = Proxy.class.getName() + id;
ccm = ClassGenerator.newInstance(cl);
ccm.setClassName(fcn);
ccm.addDefaultConstructor();
ccm.setSuperClass(Proxy.class);
ccm.addMethod("public Object newInstance(" + InvokerHandler.class.getName() + " h){ return new " + pcn + "($1);
Class<?> pc = ccm.toClass();
proxy = (Proxy)pc.newInstance();
```

代理对象名 Proxy + id

继承于 Proxy，所以要实现 newInstance 方法

添加默认构造器

实现方法 newInstance 代码， new pcn(hadler) 这里 pcn 就是前面生成的代理对象类名

利用工具类 ClassGenerator 生成字节码并实例化对象返回

### 三： Javassist 生成伪代码

下面我们以伪代码来展示下生成的代理类

比如我们要对如下接口生成代理

```
public interface DemoService {
    String sayHello(String name);
    String sayHelloAgain(String name);
}
```

生成的代理对象

```
public class DemoService.proxy10001 implements DemoService {
    public static Method[] methods = new Method[] { “sayHello”,
    “sayHelloAgain” };
    private InvocationHandler handler;
    public void DemoService.proxy10001() {}
```

```

    public void DemoService.proxy10001(InvocationHandler handler) {
        this.handler = handler;
    }
    public String sayHello(String name) {
        Object ret = handler.invoke(this, methods[0], new Object[] {name})
    }
    public String sayHello(String name) {
        Object ret = handler.invoke(this, methods[1], new Object[] {name})
    }
}

```

生成创建代理对象的代理

```

public class Proxy10001 extends Proxy {
    public void Proxy10001() {}
    public Object newInstance(InvocationHandler h) {
        return new DemoService.proxy10001(h);
    }
}

```

Proxy.*getProxy*(DemoService).newInstance(**new** InvokerInvocationHandler(invoker))代码最终创建了基于 DemoService 接口的代理对象

ClassGenerator 是 dubbo 提供的基于 javassist 之上的封装，方便 dubbo 用于生成字节码操作，ClassGenerator 主要用来收集 java 的类信息如接口，字段，方法，构造器等等信息，具体如何利用 javassist 生成字节码不在本文档介绍范围，如有兴趣请谷歌百度

Wrapper：抽象类定义了 Class 类中的常用的获取类信息的一些方法， Wrapper 包装了一个接口或者一个类可以，可以通过 Wrapper 对实例对象进行赋值取值以及指定方法调用， 如果对 spring 原理有了解的话 spring 中对 bean 的操作都是通过 BeanWrapper 这个包装器进行了的， Dubbo 的 Wrapper 的功能与它类似。 Wrapper 生成包装类的过程其实同上面生成代理类过程类似，有兴趣的同学阅读下源代码即可， 下面我们通过伪代码来展示下生成的包装类的来了解它要达到的功能。

如我们要 Wrapper 的类

```

public class Impl1 implements I1{
    private String name = "you name";
    public String getName() {return name;}
    public void setName(String name) {this.name = name; }
    public void hello(String name) {System.out.println("hello " +
name);}
}

```

生成的包装类

```

public class Wrapper1234 extends Wrapper {
    public static String[] pns = new String[] { "name" };
    public static Map pts = { "name" : "java.lang.String" };
    public static String[] mns= new String[] { "getName", "setName",
"hello" };
    public static String[] dmns= new String[] { "getName", "setName",
"hello" };
    public String[] getPropertyNames() { return pns; }
    public boolean hasProperty(String n) { return pts.containsKey(n); }
    public Class getPropertyType(String n) { return (Class)pts.get(n); }
    public String[] getMethodNames() { return mns; }
    public String[] getDeclaredMethodNames() { return dmns; }

    public void setPropertyValue(Object o, String n, Object v) {
        Wrapper1234 w;
        try { w = ((Wrapper1234) o);}
        catch (Throwable e) {throw new IllegalArgumentException(e); }
        if (n.equals("name")) {
            w.setName((java.lang.String) v);
            return;
        }
        throw 方法不存在异常
    }
}

```

```

public Object getPropertyValue(Object o, String n) {
    Wrapper1234 w;
    try { w = ((Wrapper1234) o);
    } catch (Throwable e) { throw new IllegalArgumentException(e); }
}

```

```

        if (n.equals("name")) {
            return w.getName();
        }
        抛方法不存在异常、
    }

    public Object invokeMethod(Object o, String n, Class[] p, Object[]
        v)
        throws java.lang.reflect.InvocationTargetException {
        Wrapper1234 w;
        try { w = ((Wrapper1234) o);
        } catch (Throwable e) { throw new IllegalArgumentException(e); }
        try {
            if ("getName".equals(n) && p.length == 0) {
                return w.getName();
            }
            if ("setName".equals(n) && p.length == 1) {
                w.setName((java.lang.String) v[0]);
                return null;
            }
            if ("hello".equals(n) && p.length == 1) {
                w.hello((java.lang.String) v[0]);
                return null;
            }
        } catch (Throwable e) {
            throw new java.lang.reflect.InvocationTargetException(e);
        }
        throw new
            com.alibaba.dubbo.common.bytecode.NoSuchMethodExceptio
            n("Not found method " + n + " in class Wrapper1234.");
    }
}

```

Wrapper1234 是 Wrapper 生成 Impl1 的包装对象，对 dubbo 来说把各种无法预知的接口或者类转换成对 dubbo 可知的 Wrapper 对象子类，dubbo 内部在向 Invoker 模型靠拢，Invoker 是 dubbo 内部通用可执行对象，这样一个远程调用只要根据请求 invocation 对象获取调用方法参数即可完成调用返回调用结果

### 第三章：与 spring 融合

Spring 中 bean 的定义可以通过编程，可以定义在 properties 文件，也可以定义在通过 xml 文件中，用的最多的是通过 xml 形式，由于 xml 格式具有很好的自说明便于编写及维护。对于 xml 的文档结构、数据定义及格式验证可以通过 DTD 和 Schema，在 spring2.0 之前采用的是 DTD，在 spring2.0 之后采用 Schema。使用 Schema 方式使得 spring 更加便于与第三方进行集成以及第三方可以提供更简单更便于使用的个性化配置方式。对于 Xml Schema 具体知识这里不做介绍，但是 Schema 中有一个重要的概念命名空间(namespace)必须要提一下，spring 就是利用它来做第三方自定义配置格式的解析的，在 spring 中 aop, transaction 的就是给第三个实现自己自定义配置很好实例。

```
<?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:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
                           http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

    <bean id="txManager">
        .....
    </bean>
    .....
    <tx:advice id="txAdvice" transaction-manager="txManager">
        <tx:attributes>
            <tx:method name="insert*" propagation="REQUIRED" />
            <tx:method name="update*" propagation="REQUIRED" />
            <tx:method name="delete*" propagation="REQUIRED" />
            <tx:method name="list" propagation="REQUIRED" read-only="true" />
        </tx:attributes>
    </tx:advice>
    <aop:config>
        <aop:pointcut id="interceptorPointCuts"
            expression="execution(* services.*Service(..))" />
        <aop:advisor advice-ref="txAdvice"
            pointcut-ref="interceptorPointCuts" />
    </aop:config>
</beans>
```



如上图: xmlns=http://www.springframework.org/schema/beans 是默认命名空间

xmlns:aop=http://www.springframework.org/schema/aop 定义的 aop 的命名空间

xmlns:tx="http://www.springframework.org/schema/tx" 定义了事物的命名空间

各命名空间下的格式定义文件通过 xsi:schemaLocation 来指定。

本文档不是讲解 spring 的,所以下面只是简略的来讲 spring 如果通过 schema 方式来解析配置文件的。

类 DefaultBeanDefinitionDocumentReader 会把 spring 的 xml 配置文件当做一个文档格式来读取

```
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
    if (delegate.isDefaultNamespace(root.getNamespaceURI())) {
        NodeList nl = root.getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            if (node instanceof Element) {
                Element ele = (Element) node;
                String namespaceUri = ele.getNamespaceURI();
                if (delegate.isDefaultNamespace(namespaceUri)) {
                    parseDefaultElement(ele, delegate);
                }
                else {
                    delegate.parseCustomElement(ele);
                }
            }
        }
    }
    else {
        delegate.parseCustomElement(root);
    }
}
```

每读取一个元素节点都会判断下这个元素的命名空间, 如果是默认命名空间 (http://www.springframework.org/schema/beans) 则按默认方式读取 bean 的定义, 如果不是如 namespaceUri 如下

http://www.springframework.org/schema/aop,

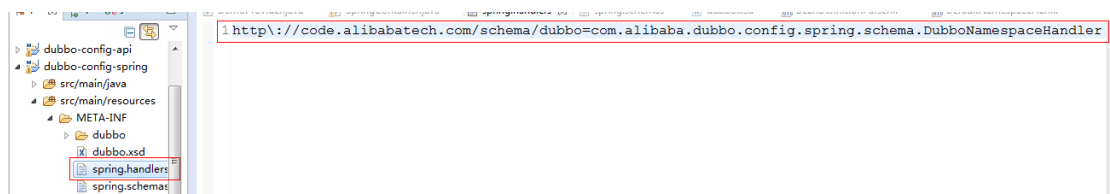
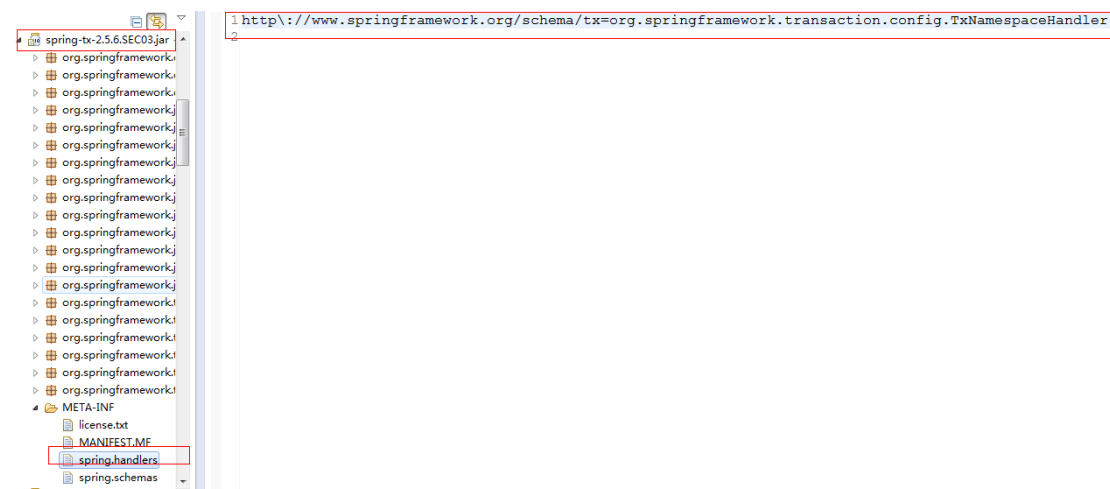
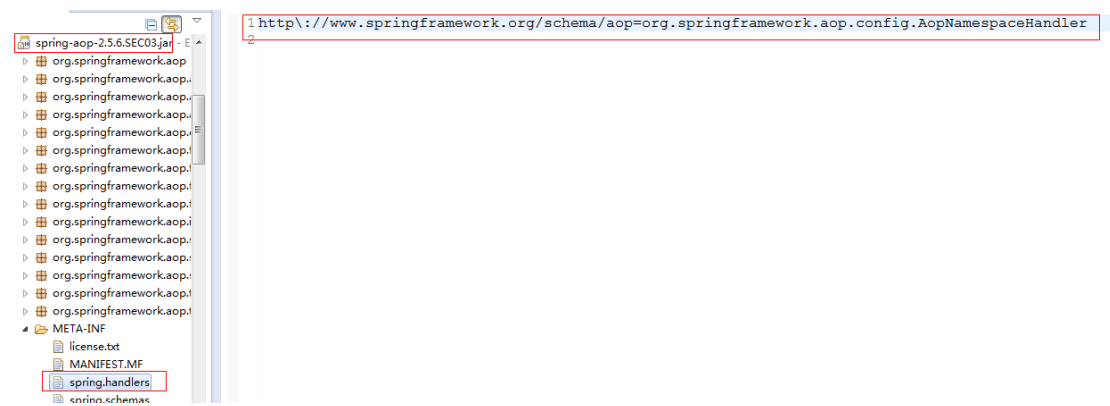
http://www.springframework.org/schema/tx,

<http://code.alibabatech.com/schema/dubbo>

则走解析自定义元素流程。根据命名空间去获取具体的处理器 NamespaceHandler

```
public BeanDefinition parseCustomElement(Element ele, BeanDefinition containingBd) {
    String namespaceUri = ele.getNamespaceURI();
    NamespaceHandler handler = this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri);
    if (handler == null) {
        error("Unable to locate Spring NamespaceHandler for XML schema namespace [" + namespaceUri + "]");
        return null;
    }
    return handler.parse(ele, new ParserContext(this.readerContext, this, containingBd));
}
```

DefaultNamespaceHandlerResolver 类传了 key 为 namespaceUri, 在类中有个 Map 存储类所有的自定义 NamespaceHandler, 这个 Map 中的值是通过工具类 PropertiesLoaderUtils 加载所有在 "META-INF/spring.handlers" 中的值



此文档主要讲解 dbbo 的自定义处理器 DubboNamespaceHandler 来怎样把 dubbo 自定义的元素转换成的 bean 定义并注册到 spring 的容器中去。

```
public class DubboNamespaceHandler extends NamespaceHandlerSupport {
    static {
        Version.checkDuplicate(DubboNamespaceHandler.class);
    }
    public void init() {
        registerBeanDefinitionParser("application", new DubboBeanDefinitionParser(ApplicationConfig.class));
        registerBeanDefinitionParser("module", new DubboBeanDefinitionParser(ModuleConfig.class, true));
        registerBeanDefinitionParser("registry", new DubboBeanDefinitionParser(RegistryConfig.class, true));
        registerBeanDefinitionParser("monitor", new DubboBeanDefinitionParser(MonitorConfig.class, true));
        registerBeanDefinitionParser("provider", new DubboBeanDefinitionParser(ProviderConfig.class, true));
        registerBeanDefinitionParser("consumer", new DubboBeanDefinitionParser(ConsumerConfig.class, true));
        registerBeanDefinitionParser("protocol", new DubboBeanDefinitionParser(ProtocolConfig.class, true));
        registerBeanDefinitionParser("service", new DubboBeanDefinitionParser(ServiceBean.class, true));
        registerBeanDefinitionParser("reference", new DubboBeanDefinitionParser(ReferenceBean.class, false));
        registerBeanDefinitionParser("annotation", new DubboBeanDefinitionParser(AnnotationBean.class, true));
    }
}
```

DubboNamespaceHandler 中注册了这么多的 BeanDefinitionParser 用来解析 dubbo 定义的那些 xml 元素节点如：

```
<dubbo:application name="dubbo-admin" />
```

```
<dubbo:registry address="${dubbo.registry.address}" check="false"
```

```

file="false" />
<dubbo:reference                                id="registryService"
interface="com.alibaba.dubbo.registry.RegistryService"  check="false"
/>
<dubbo:reference                                id="demoService"
interface="com.alibaba.dubbo.demo.DemoService" />
<dubbo:service          interface="com.alibaba.dubbo.demo.DemoService"
ref="demoService" />

```

各个 BeanDefinitionParser 会把上面的 xml 元素转换成 spring 内部的数据结构 BeanDefinition, 最终当被引用时实例化成对应的 bean 如<dubbo:application/> 节点得到 ApplicationConfig.

当然通过默认配置方式也是可以的如:

```

<dubbo:registry address="${dubbo.registry.address}" check="false"
file="false" />

```

也可以配置成

```

<bean          id="          registry"          class="
com.alibaba.dubbo.config.RegistryConfig" />
    <property name=" address" value=" ${dubbo.registry.address}" />
    <property name=" check" value=" false" />
    <property name=" file" value=" false" />
</bean/>

```

利用自定义元素解析更加简洁, 同时也可以屏蔽一些具体的实现类型, 如你不需要知道 com.alibaba.dubbo.config.RegistryConfig 这个类, 只需要知道注册 registry 这个元素就可以了, 用户可以通过文档以及 schema 的在 ide 中的自动提示可以很方便的去配置。

## 第四章：服务发布

服务发布是服务提供方注册中注册服务过程, 以便服务消费者从注册中心查阅并调用服务。

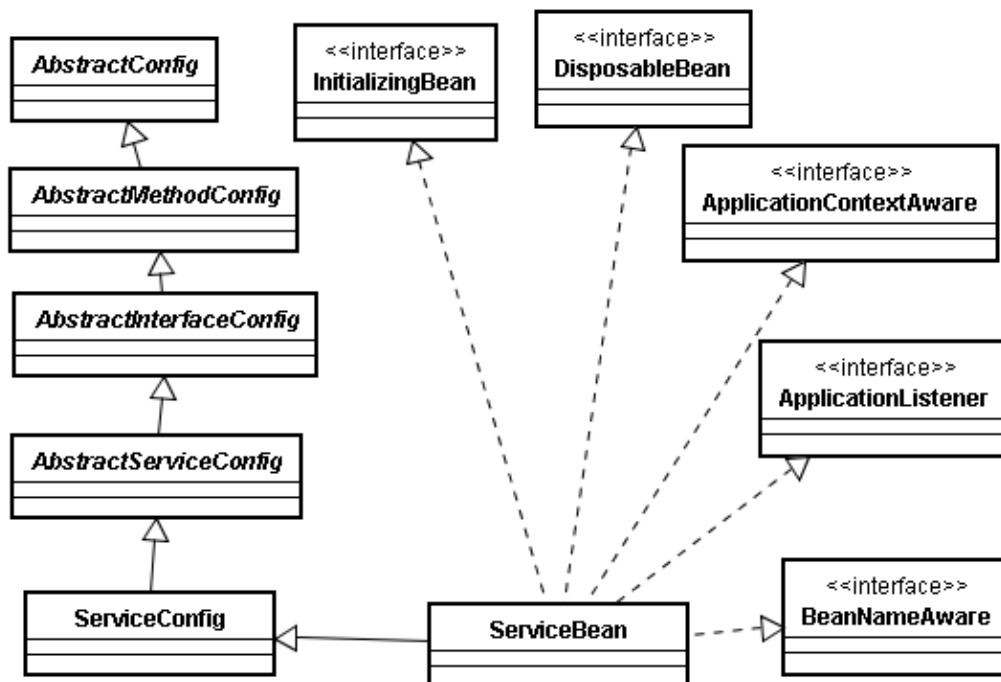
服务发布方在 spring 的配置文件中配置如下:

```

<bean id="demoService"
class="com.alibaba.dubbo.demo.provider.DemoServiceImpl" />
上面是在spring中配置的服务的具体实现, 是spring中的一个普通的bean
<dubbo:service          interface="com.alibaba.dubbo.demo.DemoService"
ref="demoService" />

```

上面的配置 spring 容器在启动的过程中会解析自定义的 schema 元素 dubbo 转换成实际的配置实现 ServiceBean , 并把服务暴露出去。



`ServiceBean` 处理继承 dubbo 自己的配置抽象类以外，还实现了一系列的 spring 接口用来参与到 spring 容器的启动以及 bean 的创建过程中去。由于 spring 的实例化 `ServiceBean` 是单例模式的，在 Spring 的容器 `ApplicationContext` 的启动过程 refresh 过程中最后第二步会预先初始化单例的 bean，在 bean 的初始化过程会设置 `beanName`，设置容器 `applicationContext`，回调 `InitializingBean` 的 `afterPropertiesSet`

最后一步 `finishRefresh` 会触发 `ContextRefreshedEvent` 事件，而 `ServiceBean` 实现了 `ApplicationListener` 接口监听了此事件，而在之前一步实例化的 `ServiceBean` 注册了这个事件，所以 `ServiceBean` 的 `onApplicationEvent(ApplicationEvent event)` 方法被触发，在这个方法中触发了 `export` 方法来暴露服务。

### ServiceConfig.doExportUrls()执行具体的 export 过程

1. `loadRegistries(true)`  
`checkRegistry` 如果 xml 中没有配置注册中，从 `dubbo.properties` 中读取配置，构建 `RegistryConfig` 对象并赋值  
 构建注册中心 URL 统一数据模型集合 `List<registryUrl>`
2. 因为 dubbo 支持多协议配置，遍历所有协议分别根据不同的协议把服务 export 到不同的注册中心上去
  - a) 判断是否是泛型暴露
  - b) 根据协议构建暴露服务的统一数据模型 URL
  - c) 配置的了 `monitor` 加载 `monitor`，并给 URL 设置 `MONITOR_KEY`
  - d) 给注册中 `registryUrl` 设置 `EXPORT_KEY` 值为前面构建的暴露服务 url

- e) 根据服务具体实现，实现接口以及 registryUrl 从代理工厂 ProxyFactory 获取代理 Invoker（继承于 AbstractProxyInvoker），它是对具体实现的一种代理
- f) Protocol.export(invoker) 暴露服务 invoker  
 Invoker 包含上一步传入的 RegistryUrl，registryUrl 的 protocol 值为 registry  
 ProtocolListenerWrapper 和 ProtocolFilterWrapper 对于协议为 REGISTRY\_PROTOCOL 直接跳过，最终由 RegistryProtocol 处理 export 的过程

**RegistryProtocol 暴露服务过程**，这里传入的 Invoker 是由 RegistryUrl 从 ProxyFactory 得到的 Invoker

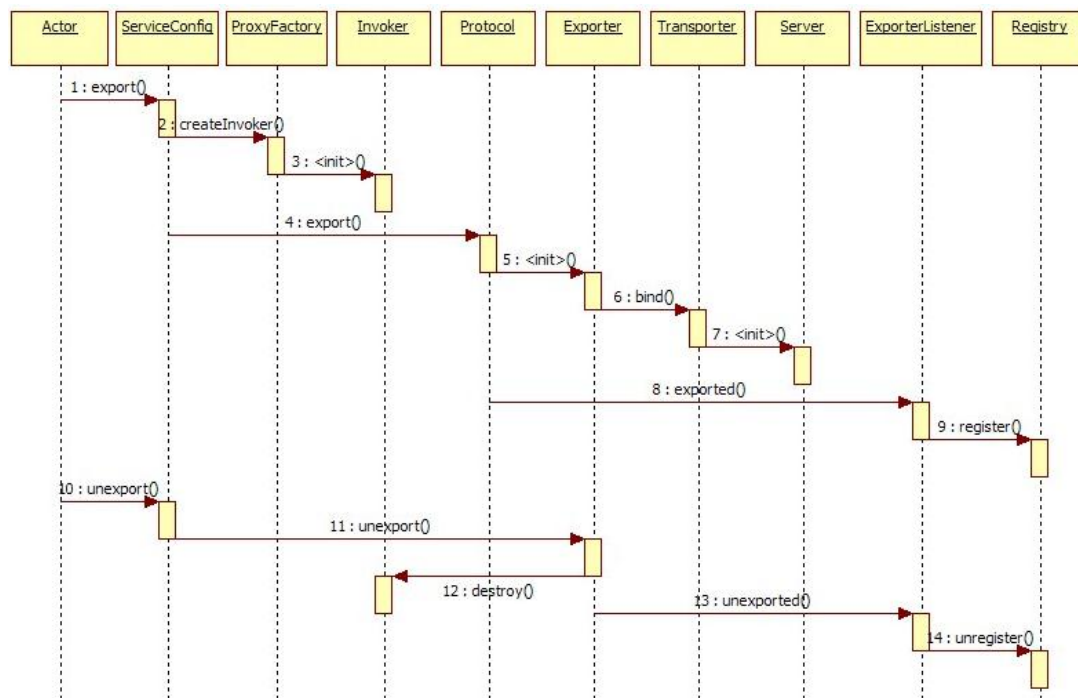
1. 从 Invoker 获取 providerUrl，在获取 cacheKey，根据 cacheKey 获取本地缓存的 ExporterChangeableWrapper（exporter 代理, 建立返回的 exporter 与 protocol export 出的 exporter 的对应关系），如果存在返回。
2. 如果不存在，根据传入的 Invoker 获取 providerUrl，在构建 InvokerDelegete(originInvoker, providerUrl)
3. Protocol.export(invokerDelegete) 根据 providerUrl 的协议（一般是 dubbo 协议）通过 Protocol 的设配类暴露服务，得到 exporter
4. 利用 providerUr 导出的 exporter 和 invoker 构建对象 ExporterChangeableWrapper 缓存到本地
5. 由 Invoker 得到 registryUrl。  
 在根据 registryUrl 从 RegistryFactory 获取 Registry，获取 RegistryUrl 的注册中心协议，这里我们拿 zooKeeper 协议为例。由 dubbo 的扩展机制得到的是 ZookeeperRegistryFactory，得到注册器为 ZookeeperRegistry
6. 由 Invoker 获取 ProviderUrl 在去除不需要在注册中心看到的字段得到 registryProviderUrl
7. 注册中心(ZookeeperRegistry)注册 registryProviderUrl  
 Registry.register(registryProviderUrl)
8. 由 registryProviderUrl 获取 overrideSubscribeUrl，在构建 OverrideListener
9. registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener) 注册中心订阅这个 url，用来当数据变化通知重新暴露，哪 zookeeper 为例，暴露服务会在 zookeeper 生成一个节点，当节点发生变化的时候会触发 overrideSubscribeListener 的 notify 方法重新暴露服务
10. 构建并返回一个新的 exporter 实例

## DubboProtocol 暴露服务的过程

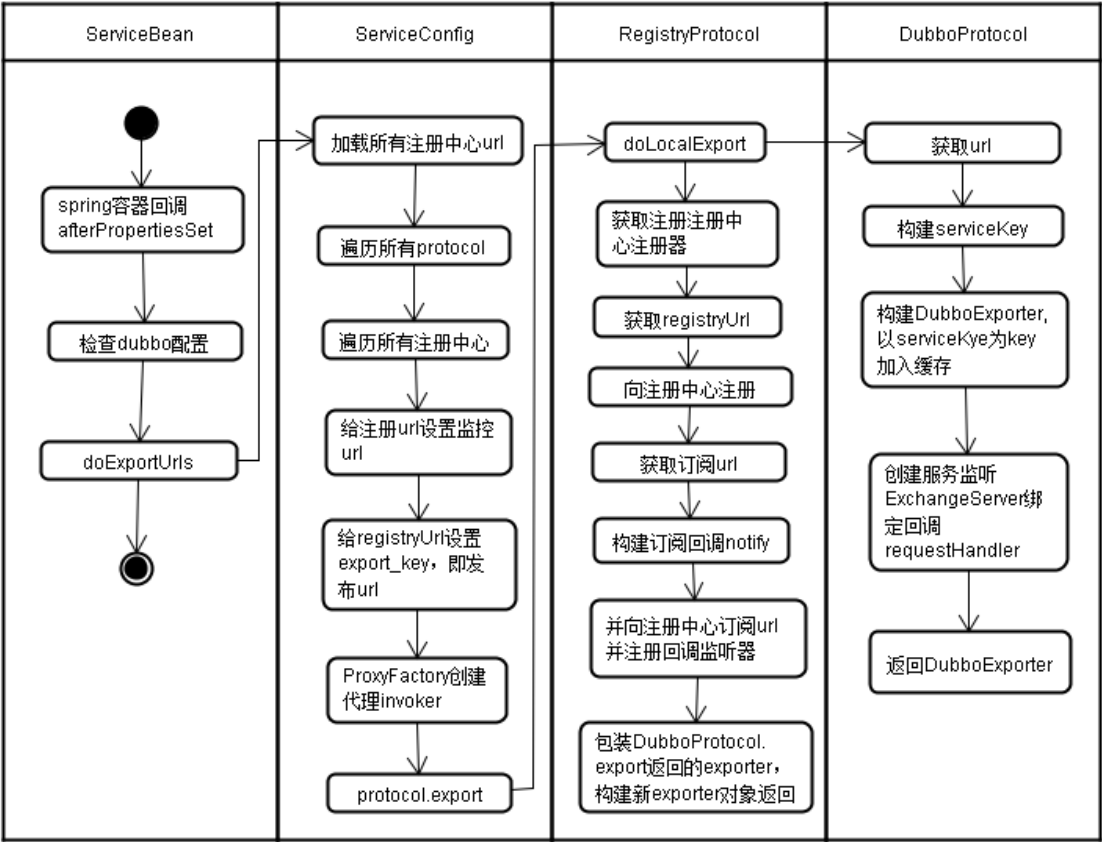
1. 从 invoker 获取统一数据模型 url

2. 由 url 构建 serviceKey (一般由端口, 接口名, 版本, group 分组)  
如: com.alibaba.dubbo.demo.DemoService:20880 这个是由接口和端口组成的
3. 构建 DubboExporter 放入本地 map 做缓存
4. 根据 url openserver。 查找本地缓存以 key 为 url.getAddress 如果没有 ExchangeServer 创建。设置 heartbeat 时间, 设置编码解码协议  
根据 url 和 ExchangeHandler 绑定 server 并返回(具体如何绑定专题介绍)
5. 返回 DubboExporter 对象

## 官方文档服务发布序列图



发布活动图



第五章：服务引用

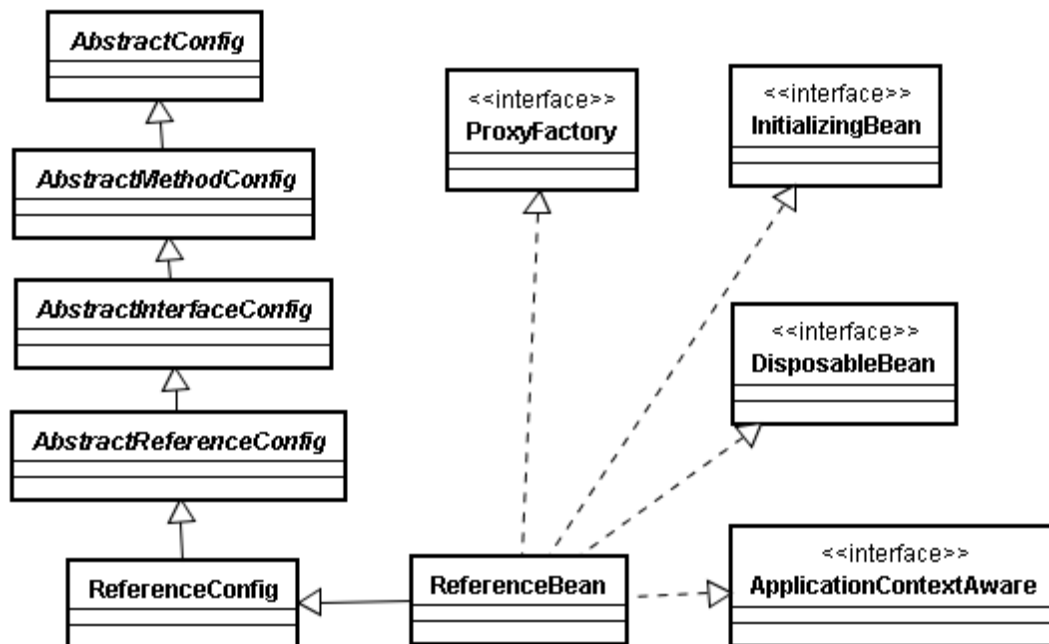
服务引用是服务的消费方向注册中心订阅服务提供方提供的服务地址后向服务

提供方引用服务的过程。

服务的应用方在 spring 的配置实例如下：

```
<dubbo:reference id="demoService" interface="com.alibaba.dubbo.demo.DemoService" />
```

如上配置 spring 在容器启动的时候会解析自定义的 schema 元素<dubbo:reference/>转换成 dubbo 内部数据结构 ReferenceBean



ReferenceBean 除了继承了配置的抽象类来处理配置信息外，它还实现 spring 容器的一些接口，这里我们分析下 FactoryBean，首先它对于 spring 来说是个 bean，参与 Bean 创建的所有生命周期，关键在于 spring 的 bean 工厂 beanFactory.getBean(“demoService”)获取的 bean 的时候会判断下是不是 FactoryBean 的实例，如果是调 factoryBean.getObject() 返回，否则返回 bean。我们对于远程调用获取的 demoService 其实并不是想要 ReferenceBean 这个对象实例本身，我们是想获取对远程调用的代理，能够通过这个代理服务调用远程服务。这里就是通过 factoryBean.getObject() 来创建引用返回基于 DemoService 接口的代理给引用，对用户透明 dubbo 封装了复杂实现。

### 创建代理的过程：

1. 获取消费者配置
2. 获取配置的注册中心，通过配置中心配置拼装 URL，线上应该是个配置中心集群
3. 遍历注册中心 List<URL>集合



加载监控中心 URL，如果配置了监控中心在注册中心 url 加上 MONITOR\_KEY  
根据配置的引用服务参数给注册中 URL 上加上 REFER\_KEY

4. 遍历注册中心 List<URL>集合，这里注册中心 url 包含了 monitorUrl 和 referUrl

protocol.refer(interface, url)调用 protocol 引用服务返回 invoker 可执行对象（这个 invoker 并不是简单的 DubboInvoker，而是由 RegistryProtocol 构建基于目录服务的集群策略 Invoker，这个 invoker 可以通过目录服务 list 出真正可调用的远程服务 invoker）

对于注册中心Url设置集群策略为AvailableCluster，由AvailableCluster将所有对象注册中调用的 invoker 伪装成一个 invoker

5. 通过代理工厂创建远程服务代理返回给使用着 proxyFactory.getProxy(invoker);

### **protocol.refer(interface, url) 引用服务的过程**

1. 经过 ProtocolListenerWrapper， ProtocolFilterWrapper 由于是注册中心 url 调用 RegistryProtocol.refer
2. 获取注册中心协议 zookeeper, redis, 还是 dubbo，并根据注册中心协议通过注册器工厂 RegistryFactory.getRegistry(url) 获取注册器 Registry 用来跟注册中心交互
3. 根据配置的 group 分组
4. 创建注册服务目录 RegistryDirectory 并设置注册器
5. 构建订阅服务的 subscribeUrl
6. 通过注册器 registry 向注册中心注册 subscribeUrl 消费端 url
7. 目录服务 registryDirectory.subscribe(subscribeUrl) 订阅服务（这里我们以开源版本 zookeeper 为注册中心为例来讲解， dubbo 协议的注册中心有点不一样）

其实内部也是通过注册器 registry.subscribe(url, this) 这里 this 就是 registryDirectory 它实现了 NotifyListener。

服务提供者在向 zookeeper 注册服务 /dubbo/com.alibaba.dubbo.demo.DemoService/providers/节点下写下自己的 URL 地址

服务消费者向 zookeeper 注册服务 /dubbo/com.alibaba.dubbo.demo.DemoService/consumers/节点下写下自己的 URL 地址

服务消费者向 zookeeper 订阅服务 /dubbo/com.alibaba.dubbo.demo.DemoService/ providers /节点下所有服务提供者 URL 地址

Zookeeper 通过 watcher 机制实现对节点的监听，节点数据变化通过节点上的 watcher 回调客户端，重新生成对服务的 refer

在订阅的过程中通过获取 /dubbo/com.alibaba.dubbo.demo.DemoService/providers / 下的所有服务提供者的 urls( 类似 dubbo://10.33.37.8:20880/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=demo-consumer&check=false&dubbo=2.0.0&generic=false&interface=com.alibaba.dubbo.demo.DemoService&methods=sayHello&owner=william&pid=7356&side=consumer&timestamp=1416971340626) ，主动回调 NotifyListener 来根据 urls 生成对服务提供者的引用生成可执行 invokers, 供目录服务持有着，

看下如下 RegistryDirectory.notify(urls) 方法中的代码实现

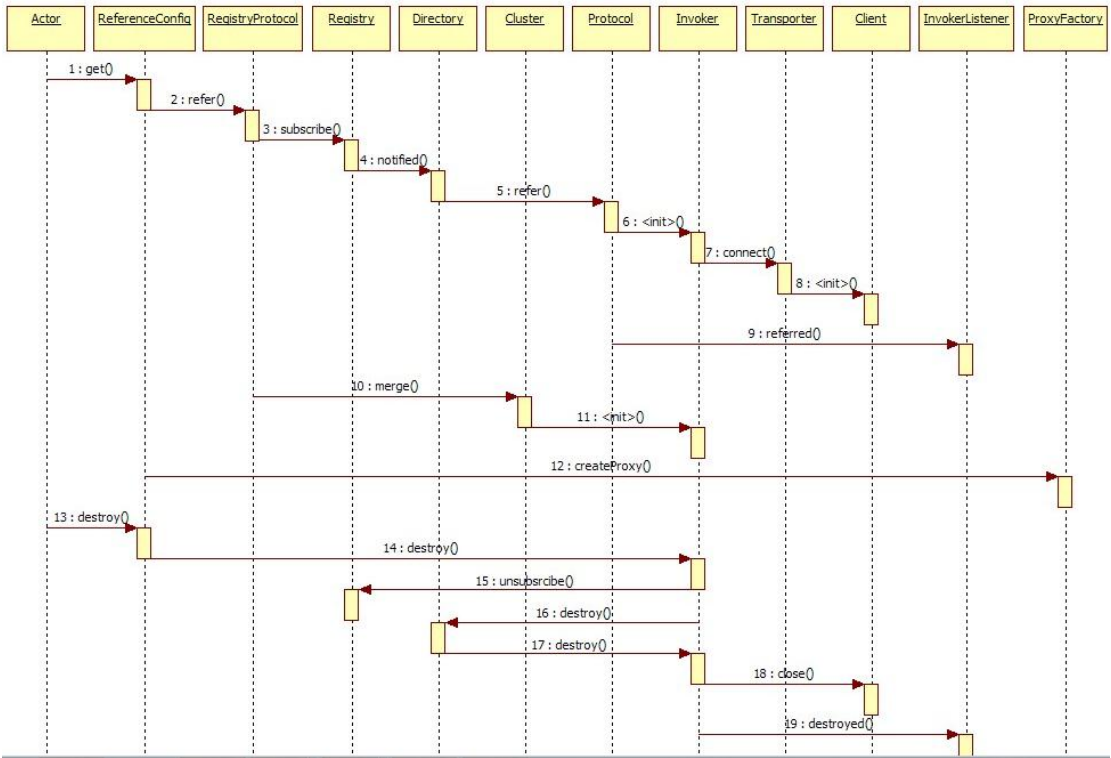
```
RegistryDirectory.java
391         } else {
392             enabled = url.getParameter(Constants.ENABLED_KEY, true);
393         }
394         if (enabled) {
395             invoker = new InvokerDelegator<T>(protocol.refer(serviceType, url), url, providerUrl);
396         }
397     } catch (Throwable t) {
```

## 8. 通过 cluster.join(directory) 合并 invoker 并提供集群调用策略

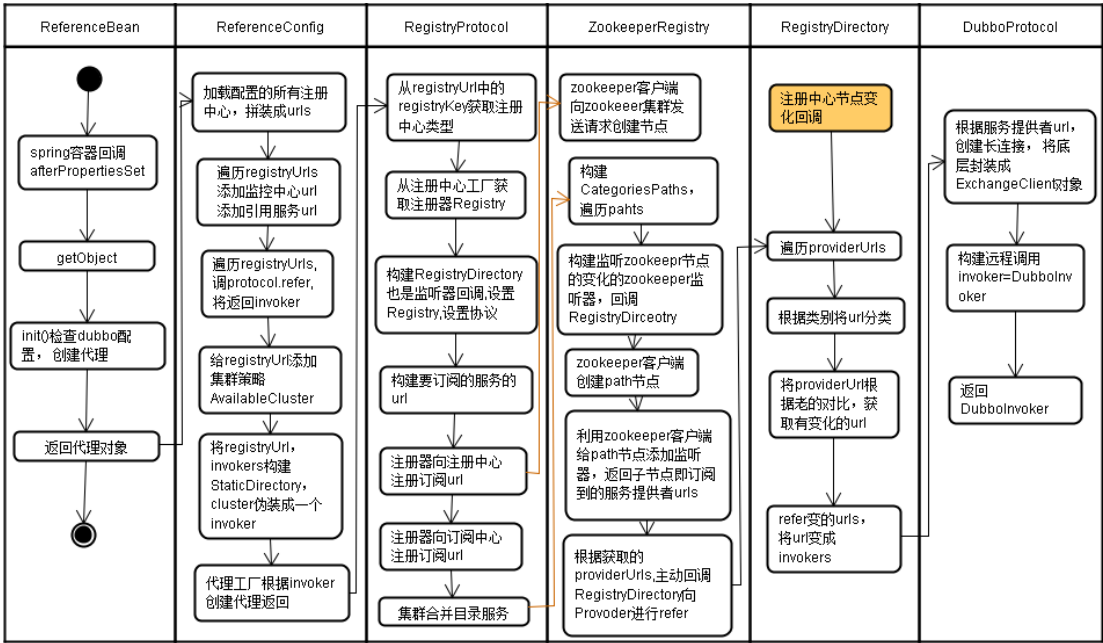
### DubboProtocol.refer 过程:

1. 经过 ProtocolListenerWrapper, ProtocolFilterWrapper 构建监听器链和过滤器链。
2. DubboProtocol 根据 url 获取 ExchangeClient 对象，如果是 share 存在就返回不存在创建新对象不是 share 直接创建。ExchangeClient 是底层通信的客户端，对于通信层的创建功能不在这里讲解。
3. 创建 DubboInvoker, 这个 invoker 对象包含对远程服务提供者的长链接，是真正执行远程服务调用的可执行对象
4. 将创建的 invoker 返回给目录服务

官方文档的应用服务的序列图



发布服务活动图:



## 第六章： Listener & filter

### 一： Listener

#### ExporterListener:

dubbo 在服务暴露(exporter)以及销毁暴露(unexporter)服务的过程中提供了回调窗口，供用户做业务处理。ProtocolListenerWrapper 在暴露过程中构建了监听器链

```
public class ProtocolListenerWrapper implements Protocol {
    public <T> Exporter<T> export(Invoker<T> invoker) throws
        RpcException {
        ..... //注册中心代码

        return new
            ListenerExporterWrapper<T>(protocol.export(invoker),
                Collections.unmodifiableList(ExtensionLoader.get
                    ExtensionLoader(ExporterListener.class).getActiv
                        ateExtension(invoker.getUrl(),
                            Constants.EXPORTER_LISTENER_KEY)));
    }
}
```

1. 根据 Dubbo 的 SPI 扩展机制获取所有实现了 ExporterListener 的监听器 listeners
2. Protocol.export(invoker)暴露服务返回结果 exporter 对象
3. ListenerExporterWrapper 装饰 exporter, 在构造器中遍历 listeners 构建 export 的监听链
4. ListenerExporterWrapper 实现 Exporter<T>接口，在 unexport 方法实现中构建 unexport 的监听链

#### InvokerListener:

dubbo 在服务引用(refer)以及销毁引用(destroy)服务的过程中提供了回调窗口，供用户做业务处理。ProtocolListenerWrapper 在暴露过程中构建了监听器链

```
public <T> Invoker<T> refer(Class<T> type, URL url) throws
    RpcException {
    return new ListenerInvokerWrapper<T>(protocol.refer(type,
        url),
        Collections.unmodifiableList(ExtensionLoader.get
            ExtensionLoader(InvokerListener.class).getActiv
                vateExtension(url,
```

```

        Constants.INVOKER_LISTENER_KEY))));
    }

```

1. 根据 Dubbo 的 SPI 扩展机制获取所有实现了 InvokerListener 的监听器 listeners
2. Protocol.refer(type, url)暴露服务返回结果 invoker 对象
3. ListenerInvokerWrapper 装饰 invoker, 在构造器中遍历 listeners 构建 referer 的监听链
4. ListenerInvokerWrapper 实现 Invoker<T>接口, 在 destroy 方法实现中构建 destroy 的监听链

Dubbo 的开源版本中没有监听的实现, 但是开放了口子, 业务方如有需要可以利用这个功能实现特定的业务

## 二: Filter

Filter:是一种递归的链式调用, 用来在远程调用真正执行的前后加入一些逻辑, 跟 aop 的拦截器 servlet 中 filter 概念一样的

Filter 接口定义

@SPI

```

public interface Filter {
    Result invoke(Invoker<?> invoker, Invocation invocation)
    throws RpcException;
}

```

ProtocolFilterWrapper: 在服务的暴露与引用的过程中根据 KEY 是 PROVIDER 还是 CONSUMER 来构建服务提供者与消费者的调用过滤器链

```

public <T> Exporter<T> export(Invoker<T> invoker) throws
RpcException {
    return protocol.export(buildInvokerChain(invoker,
        Constants.SERVICE_FILTER_KEY, Constants.PROVIDER));
}
public <T> Invoker<T> refer(Class<T> type, URL url) throws
RpcException {
    return buildInvokerChain(protocol.refer(type, url),
        Constants.REFERENCE_FILTER_KEY,
        Constants.CONSUMER);
}

```

Filter 的实现类需要打上 @Activate 注解, @Activate 的 group 属性是个 string 数组, 我们可以通过这个属性来指定这个 filter 是在 consumer, provider 还是两者情况下激活, 所谓激活就是能够被获取, 组成 filter 链

```

List<Filter> filters =

```

```
ExtensionLoader.getExtensionLoader(Filter.class).getActivateExtension(invoker.getUrl(), key, group);
```

Key就是SERVICE\_FILTER\_KEY还是REFERENCE\_FILTER\_KEY

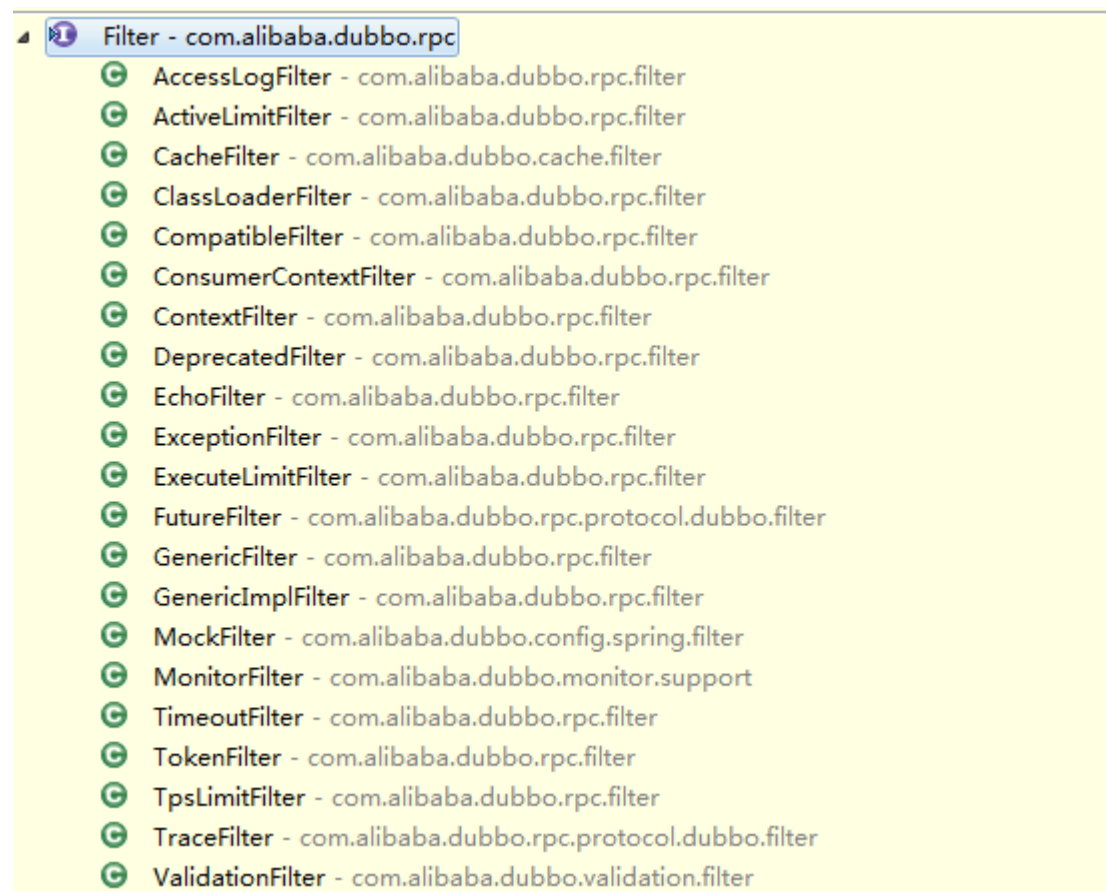
Group就是consumer或者provider

构建filter链，当我们获取激活的filter集合后就通过buildInvokerChain方法来构建

```
for (int i = filters.size() - 1; i >= 0; i --) {
    final Filter filter = filters.get(i);
    final Invoker<T> next = last;
    last = new Invoker<T>() {
        public Result invoke(Invocation invocation) throws
RpcException {
            return filter.invoke(next, invocation);
        }
        . . . . . //其他方法
    };
}
```

以上代码展示了构建 filter 链的过程

Dubbo 内容提供了大量内部实现，用来实现调用过程额外功能， 如向监控中心发送调用数据， Tps 限流等等， 每个 filter 专注一块功能。用户同样可以通过Dubbo 的 SPI 扩展机制现在自己的功能



## 第七章：注册中心

### 一： 接口介绍

服务注册与发现的中心，服务的提供者将服务发布到注册中心，服务的使用者到注册中心引用服务。

Dubbo 的注册中心提供了多种实现，其实现是基于 dubbo 的 spi 的扩展机制的，使用者可以直接实现自己的注册中心。

@SPI("dubbo")

**public interface** RegistryFactory {

/\*\*

\* 连接注册中心.

\* 连接注册中心需处理契约

\* 1. 当设置check=false时表示不检查连接，否则在连接不上时抛出异常。

\* 2. 支持URL上的username:password权限认证。

\* 3. 支持backup=10.20.153.10备选注册中心集群地址。

\* 4. 支持file=registry.cache本地磁盘文件缓存。

\* 5. 支持timeout=1000请求超时设置。

\* 6. 支持session=60000会话超时或过期设置。

\* @param url 注册中心地址，不允许为空

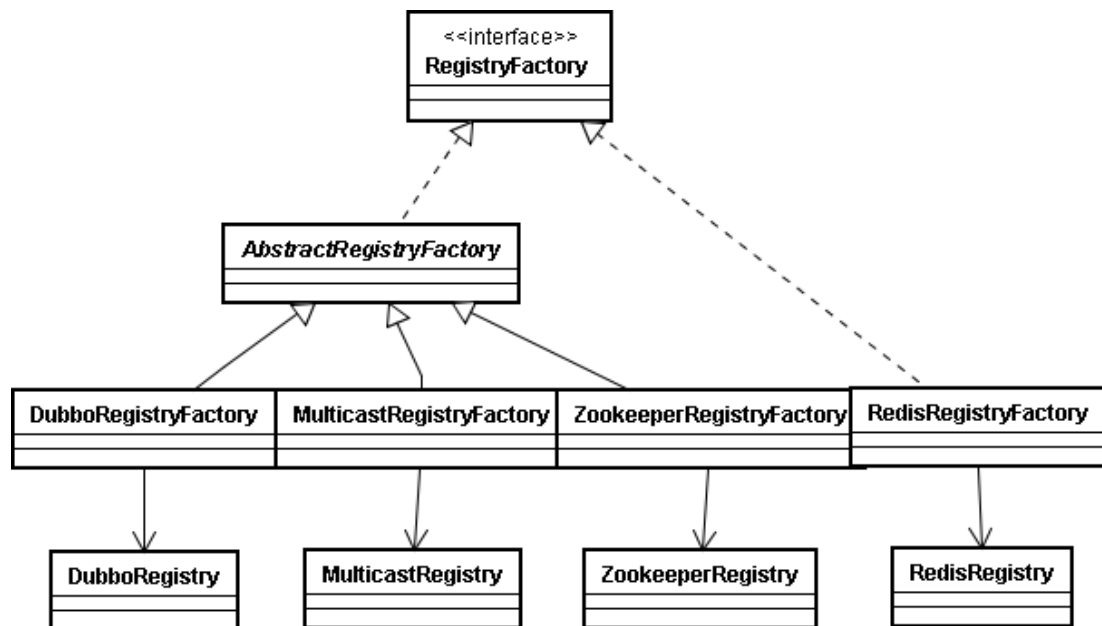
\* @return 注册中心引用，总不返回空

\*/

```

@Adaptive({"protocol"})
Registry getRegistry(URL url);
}

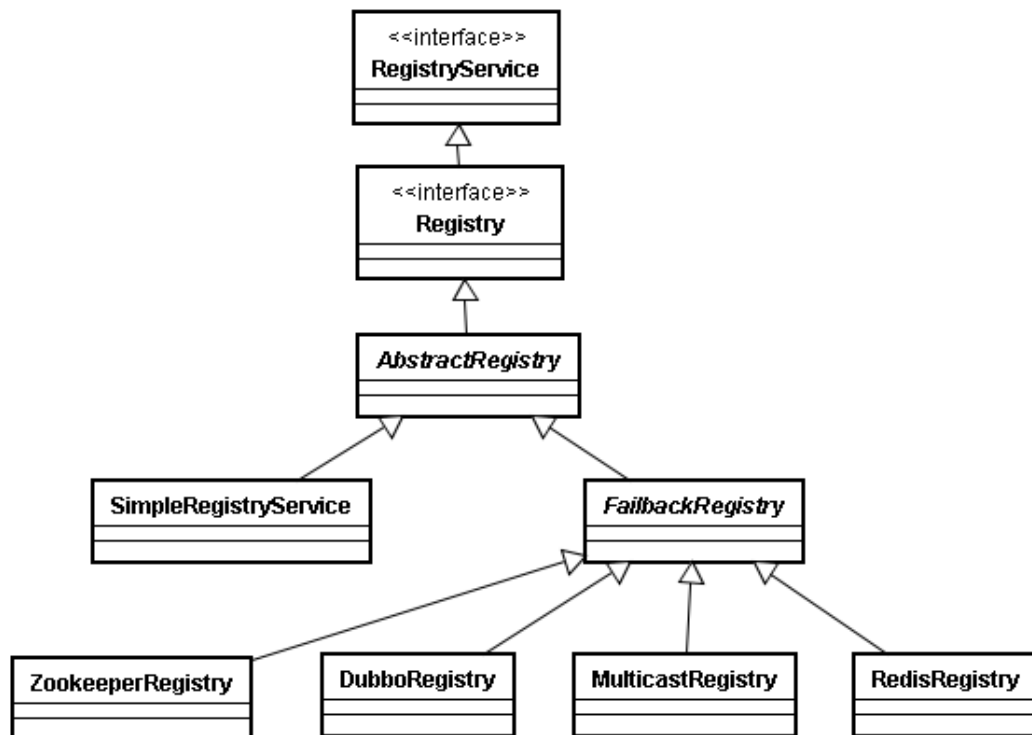
```



RegistryFactory 用来创建注册中心， 默认的注册中心是 dubbo 协议， 由于阿里的注册中心并没有开源， dubbo 协议注册中心只提供了一个简单实现。开源 dubbo 的注册中心推荐使用 zookeeper。这里我们主要去分析基于 dubbo 和 zookeeper 协议的注册中心实现及使用。

注册中心服务类图：





服务接口定义

```
public interface RegistryService {
    void register(URL url);
    void unregister(URL url);
    void subscribe(URL url, NotifyListener listener);
    void unsubscribe(URL url, NotifyListener listener);
    List<URL> lookup(URL url);
}
```

Register: 注册数据, 比如: 提供者地址, 消费者地址, 路由规则, 覆盖规则, 等数据。

注册需处理契约

1. 当URL设置了check=false时, 注册失败后不报错, 在后台定时重试, 否则抛出异常。
2. 当URL设置了dynamic=false参数, 则需持久存储, 否则, 当注册者出现断电等情况异常退出时, 需自动删除。
3. 当URL设置了category=routers时, 表示分类存储, 缺省类别为providers, 可按分类部分通知数据。

4. 当注册中心重启，网络抖动，不能丢失数据，包括断线自动删除数据。

5. 允许 URI 相同但参数不同的 URL 并存，不能覆盖。

Unregister: 取消注册

Subscribe: 订阅符合条件的已注册数据，当有注册数据变更时自动推送  
订阅需处理契

1. 当URL设置了check=false时，订阅失败后不报错，在后台定时重试。

2. 当URL设置了category=routers，只通知指定分类的数据，多个分类用逗号分隔，并允许星号通配，表示订阅所有分类数据。

3. 允许以interface,group,version,classifier作为条件查询，

如: interface=com.alibaba.foo.BarService&version=1.0.0

4. 并且查询条件允许星号通配，订阅所有接口的所有分组的所有版本，

或: interface=\*&group=\*&version=\*&classifier=\*

5. 当注册中心重启，网络抖动，需自动恢复订阅请求

6. 允许URI相同但参数不同的URL并存，不能覆盖

7. 必须阻塞订阅过程，等第一次通知完后再返回。

Unsubscribe:取消订阅

Lookup: 查询符合条件的已注册数据，与订阅的推模式相对应，这里为拉模式，只返回一次结果

## 二: Dubbo 协议注册中心

基于 dubbo 协议开源只是给出了默认一个注册中心实现 SimpleRegistryService, 它只是一个简单实现，不支持集群，就是利用 Map<String/\*ip:port\*/, Map<String/\*service\*/, URL>来存储服务地址，具体不在啰嗦了，请读者翻看源代码，可作为自定义注册中的参考。

## 注册中心启动

SimpleRegistryService 本身也是作为一个 dubbo 服务暴露。

```
<dubbo:protocol port="9090" />
<dubbo:service interface="com.alibaba.dubbo.registry.RegistryService"
    ref="registryService" registry="N/A" ondisconnect="disconnect"
    callbacks="1000">
    <dubbo:method name="subscribe"><dubbo:argument index="1"
        callback="true" /></dubbo:method>
    <dubbo:method name="unsubscribe"><dubbo:argument index="1"
        callback="false" /></dubbo:method>
</dubbo:service>
<bean id="registryService"
    class="com.alibaba.dubbo.registry.simple.SimpleRegistryService" />
```

上面是暴露注册中心的 dubbo 服务配置，

定义了注册中心服务的端口号

发布 RegistryService 服务，registry 属性是 "N/A" 代表不能获取注册中心，注册中心服务的发布也是一个普通 dubbo 服务的发布，如果没有配置这个属性它也会寻找注册中心，去通过注册中心发布，因为自己本身就是注册中心，直接对外发布服务，外部通过 ip: port 直接使用。

服务发布定义了回调接口，这里定义了 subscribe 的第二个入参类暴露的回调服务供注册中心回调，用来当注册的服务状态变更时反向推送到客户端。

Dubbo 协议的注册中心的暴露以及调用过程跟普通的 dubbo 服务的其实是一样的，可能跟绝大多数服务的不同的是在 SimpleRegistryService 在被接收订阅请求 subscribe 的时候，同时会 refer 引用调用方暴露的 NotifyListener 服务，当有注册数据变更时自动推送

## 生产者发布服务

Dubbo 协议向注册中心发布服务：当服务提供方，向 dubbo 协议的注册中心发布服务的时候，是如何获取，创建注册中心的，如何注册以及订阅服务的，下面我们来分析其流程。

看如下配置发布服务：

```

<dubbo:registry protocol="dubbo" address="127.0.0.1:9090"
/>
<bean id="demoService"
class="com.alibaba.dubbo.demo.provider.DemoServiceImpl" />
<dubbo:service
interface="com.alibaba.dubbo.demo.DemoService"
ref="demoService" />

```

1. 指定了哪种的注册中心，是基于 dubbo 协议的，指定了注册中心的地址以及端口号
2. 发布 DemoService 服务，服务的实现为 DemoServiceImpl

每个<dubbo:service/>在 spring 内部都会生成一个 ServiceBean 实例，ServiceBean 的实例化过程中调用 export 方法来暴露服务

1. 通过 loadRegistries 获取注册中心 registryUrls

```

registry://127.0.0.1:9090/com.alibaba.dubbo.registry.RegistryService?application=demo-provider&dubbo=2.5.4-SNAPSHOT&owner=william&pid=7084&registry=dubbo&timestamp=1415711791506

```

用统一数据模型 URL 表示：

protocol=registry 表示一个注册中心 url

注册中心地址 127.0.0.1:9090

调用注册中心的服务 RegistryService

注册中心协议是 registry=dubbo

。。。。。。

2. 构建发布服务的 URL

```

dubbo://192.168.0.102:20880/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=demo-provider&dubbo=2.5.4-SNAPSHOT&generic=false&interface=com.alibaba.dubbo.demo.DemoService&loadbalance=roundrobin&methods=sayHell

```

o&owner=william&pid=7084&side=provider&timestamp=1415712331601

发布协议 `protocol = dubbo`

服务提供者的地址为 `192.168.0.102:20880`

发布的服务为 `com.alibaba.dubbo.demo.DemoService`

。 。 。 。 。

### 3. 遍历 `registryUrls` 向注册中心注册服务

给每个 `registryUrl` 添加属性 `key` 为 `export`, `value` 为上面的发布服务

`url` 得到如下 `registryUrl`

```
registry://127.0.0.1:9098/com.alibaba.dubbo.registry.RegistryService?application=demo-provider&dubbo=2.5.4-SNAPSHOT&export=dubbo%3A%2F%2F192.168.0.102%3A20880%2Fcom.alibaba.dubbo.demo.DemoService%3Fanyhost%3Dtrue%26application%3Ddemo-provider%26dubbo%3D2.5.4-SNAPSHOT%26generic%3Dfalse%26interface%3Dcom.alibaba.dubbo.demo.DemoService%26loadbalance%3Droundrobin%26methods%3DsayHello%26owner%3Dwilliam%26pid%3D7084%26side%3Dprovider%26timestamp%3D1415712331601&owner=william&pid=7084&registry=dubbo&timestamp=1415711791506
```

### 4. 由发布的服务实例，服务接口以及 `registryUrl` 为参数，通过代理工厂

`proxyFactory` 获取 `Invoker` 对象, `Invoker` 对象是 `dubbo` 的核心模型，其他对象都向它靠拢或者转换成它。

### 5. 通过 `Protocol` 对象暴露服务 `protocol.export(invoker)`

通过 `DubboProtocol` 暴露服务的监听 (不是此节内容)

通过 `RegistryProtocol` 将服务地址发布到注册中心，并订阅此服务

`RegistryProtocol.export(Invoker)` 暴露服务

#### 1. 调 `DubboProtocol` 暴露服务的监听

## 2. 获取注册中心 getRegistry(Invoker)

URL 转换， 由 Invoker 获取的 url 是 registryURL 它的协议属性用来选择何种的 Protocol 实例如 RegistryProtocol, DubboProtocol 或者 RedisProtocol 等等。 这里要通过 URL 去选择何种注册中心， 所以根据 registry=dubbo 属性， 重新设置 url 的协议属性得 registryUrl `dubbo://127.0.0.1:9098/com.alibaba.dubbo.registry.RegistryService?application=demo-provider&dubbo=2.5.4-SNAPSHOT&export=dubbo%3A%2F%2F192.168.0.102%3A20880%2Fcom.alibaba.dubbo.demo.DemoService%3Fanyhost%3Dtrue%26application%3Ddemo-provider%26dubbo%3D2.5.4-SNAPSHOT%26generic%3Dfalse%26interface%3Dcom.alibaba.dubbo.demo.DemoService%26loadbalance%3Droundrobin%26methods%3DsayHello%26owner%3Dwilliam%26pid%3D5040%26side%3Dprovider%26timestamp%3D1415715706560&owner=william&pid=5040&timestamp=1415715706529`

RegistryFactory.getRegistry(url) 通过工厂类创建注册中心， RegistryFactory 通过 dubbo 的 spi 机制获取对应的工厂类， 这里的是基于 dubbo 协议的注册中心， 所以是 DubboRegistryFactory

## 3. 获取发布 url 就是 registryUrl 的 export 参数的值

registryProviderUrl=dubbo://10.33.37.7:20880/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=demo-provider&dubbo=2.5.4-SNAPSHOT&generic=false&interface=com.alibaba.dubbo.demo.DemoService&loadbalance=roundrobin&methods=sayHello&owner=william&pid=6976&side=provider&timestamp=1415846958825

## 4. DubboRegistry.register(registryProviderUrl)

通过注册器向注册中心注册服务

这里注意 registryProviderUrl 的并没有设置 category 属性，在注册中心 UrlUtils.ismatch(consumerUrl, providerUrl) 比较的时候，providerUrl 的 category 属性取默认值 providers，这点消费者订阅的时候会指定订阅的 url 的 category=providers，去判断有没有注册的提供者。

#### 5. 构建订阅服务 overrideProviderUrl, 我们是发布服务

```
provider://10.33.37.7:20880/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=demo-provider&category=configurators&check=false&dubbo=2.5.4-SNAPSHOT&generic=false&interface=com.alibaba.dubbo.demo.DemoService&loadbalance=roundrobin&methods=sayHello&owner=william&pid=6432&side=provider&timestamp=1415847417663
```

#### 6. 构建 OverrideListener 它实现与 NotifyListener, 当注册中心的订阅的 url 发生变化时回调重新 export

#### 7. registry.subscribe(overrideProviderUrl, OverrideListener), 注册器向注册中心订阅 overrideProviderUrl, 同时将 OverrideListener 暴露为回调服务, 当注册中心的 overrideProviderUrl 数据发生变化时回调,

注册器 DubboRegistry 的 registry, subscribe, unRegistry, unsubscribe 都类似, 是一个 dubbo 的远程服务调用

### DubboRegistryFactory 创建注册中心过程

#### 1. 根据传入 registryUrl 重新构建

移除 EXPORT\_KEY REFER\_KEY

添加订阅回调参数

```
dubbo://127.0.0.1:9098/com.alibaba.dubbo.registry.RegistryService?application=demo-provider&callbacks=10000&connect.timeout=10000&dubbo=2.5.4-SNAPSHOT&interface=com.alibaba.dubbo.registry.RegistryService&lazy=true&methods=register,subscribe,unregister,unsubscribe,lookup&owner=william&pid=8492&reconnect=false&sticky=true&subscribe.1.callback=true&timeout=10000&timestamp=1415783872554&unsubscribe.1.callback=false
```

2. 根据 url 注册服务接口构建注册目录对象 RegistryDircectory, 实现了 NotiyfLisener, 这里 NotiyfLisener 实现主要是根据 urls 去 引用远程服务 RegistryService 得到对应的 Invoker, 当 urls 变化时重新 refer; 目录服务可以列出所有可以执行的 Invoker
3. 利用 cluster 的 join 方法, 将 Dirctory 的多个 Invoker 对象伪装成一个 Invoker 对象, 这里默认集群策略得到 FailoverClusterInvoker
4. FailoverClusterInvoker 利用 ProxyFactory 获取到 RegistryService 服务的代理对象
5. 由 RegistryService 服务的代理对象和 FailoverClusterInvoker 构建 dubbo 协议的注册中心注册器 DubboRegistry
6. RegistryDircectory 设置注册器 DubboRegistry, 设置 dubbo 的协议
7. 调用 RegistryDircectory 的 notify(urls) 方法  
主要是根据 registryUrls, 引用各个注册中心的 RegistryService 服务实现, 将引用的服务按 key=menthodName/value=invoker 缓存起来, 目录服务 Directory.list(Invocation) 会列出所调用方法的所有 Invoker , 一个 Invoker 代表对一个注册中心的调用实体。



## 8. 订阅注册中心服务，服务的提供者调注册中心的服务 RegistryService

属于消费方， 所以订阅服务的 url 的协议是 consumer

```
consumer://192.168.0.102/com.alibaba.dubbo.registry.RegistryService?application=demo-provider&callbacks=10000&connect.timeout=10000&dubbo=2.5.4-SNAPSHOT&interface=com.alibaba.dubbo.registry.RegistryService&lazy=true&methods=register,subscribe,unregister,unsubscribe,lookup&owner=william&pid=6960&reconnect=false&sticky=true&subscribe.1.callback=true&timeout=10000&timestamp=1415800789364&unsubscribe.1.callback=false
```

订阅的目的在于在注册中心的数据发送变化的时候反向推送给订阅方

directory.subscribe(url) 最终调用注册中心的 RegistryService 远

程服务， 它是一个普通的 dubbo 远程调用。要说跟绝大多数 dubbo 远程调

用的区别： url 的参数 `subscribe.1.callback=true` 它的意思是

RegistryService 的 subscribe 方法的第二个参数 NotifyListener

暴露为回调服务； url 的参数 `unsubscribe.1.callback=false` 的意

思是 RegistryService 的 `unsubscribe` 方法的第二个参数

NotifyListener 暴露的回调服务销毁。

这里 dubbo 协议的注册中心调注册中心的服务采用的默认集群调用策略是

FailOver, 选择一台注册中心， 只有当失败的时候才重试其他服务器， 注册中

心实现也比较简单不具备集群功能， 如果想要初步的集群功能可以选用

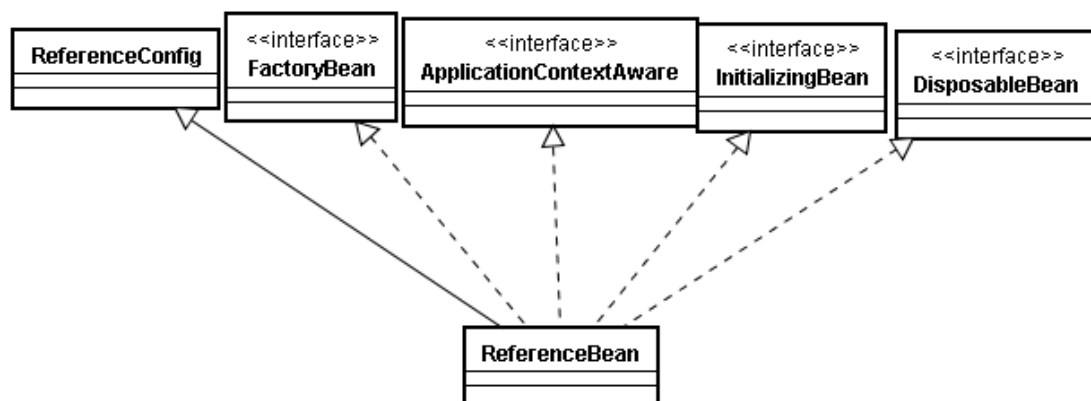
BroadcastCluster 它至少向每个注册中心遍历调用注册一遍

## 消费者引用服务

```
<dubbo:registry protocol="dubbo" address="127.0.0.1:9098"
/>
<dubbo:reference id="demoService" interface="com.alibaba.d
ubbo.demo.DemoService" />
```

1. 指定了哪种的注册中心，是基于 dubbo 协议的，指定了注册中心的地址以及端口号
2. 引用远程 DemoService 服务

每个<dubbo:reference/>标签 spring 加载的时候都会生成一个 ReferenceBean。



如上图 ReferenceBean 实现了 spring 的 FactoryBean 接口，实现了此接口的 Bean 通过 spring 的 BeanFactory.getBean("beanName") 获取的对象不是配置的 bean 本身而是通过 FactoryBean.getObject() 方法返回的对象，此接口在 spring 内部被广泛使用，用来获取代理对象等等。这里 getObject 方法用来生成对远程服务调用的代理

1. loadRegistries() 获取配置的注册中心的 registryUrls

2. 遍历 registryUrls 集合，给 registryUrl 加上 refer key 就是要引用的远程服务

```
[registry://127.0.0.1:9098/com.alibaba.dubbo.registry.RegistryService?application=demo-consumer&dubbo=2.0.0&pid=2484&refer=application%3Ddemo-consumer%26dubbo%3D2.0.0%26interface%3Dcom.alibaba.dubbo.demo.DemoService%26methods%3DsayHello%26pid%3D2484%26side%3Dconsumer%26timestamp%3D1415879965901&registry=dubbo&timestamp=1415879990670]
```

3. 遍历 registryUrls 集合，使用 Protocol.refer(interface, registryUrl) 的到可执行对象 invoker
4. 如果注册中心有多个的话，通过集群策略 Cluser.join() 将多个 invoker 伪装成一个可执行 invoker，这里默认使用 available 策略
5. 利用代理工厂生成代理对象 proxyFactory.getProxy(invoker)

#### Protocol.refer 过程流程

1. 根据传入的 registryUrl 是用来选择 RegistryProcol 它的协议属性是 registry，下面要选择使用哪种注册中心所以要根据 REGISTRY\_KEY 属性重新设置 registrUrl

```
dubbo://127.0.0.1:9098/com.alibaba.dubbo.registry.RegistryService?application=demo-consumer&dubbo=2.0.0&pid=4524&refer=application%3Ddemo-consumer%26dubbo%3D2.0.0%26interface%3Dcom.alibaba.dubbo.demo.DemoService%26methods%3DsayHello%26pid%3D4524%26side%3Dconsumer%26timestamp%3D1415881461048&timestamp=1415881461113
```

2. 根据 registrUrl 利用 RegistryFactory 获取注册器（过程跟暴露服务那边一样），这里是 dubbo 协议得到的是注册器是 DubboRegistry 引用并订阅注册中心服务，

### 3. 构建引用服务的 subscribeUrl

```
consumer://10.5.24.221/com.alibaba.dubbo.demo.DemoService?application=demo-consumer&category=consumers&check=false&dubbo=2.0.0&interface=com.alibaba.dubbo.demo.DemoService&methods=sayHello&pid=8536&side=consumer&timestamp=1415945205031
```

并通过注册器向注册中心注册消费方， 主要这里的 category 是 consumers

### 4. 构建目录服务 RegistryDirectory,

构建订阅消费者订阅 url, 这里主要 category=providers 去注册中心寻找注册的服务提供者

```
consumer://10.33.37.4/com.alibaba.dubbo.demo.DemoService?application=demo-consumer&category=providers,configurators,routers&dubbo=2.0.0&interface=com.alibaba.dubbo.demo.DemoService&methods=sayHello&pid=9692&side=consumer&timestamp=1415967547508
```

向注册中心订阅消费方, 注册中心根据消费者传入的 url 找到匹配的服务提供者 url (注意: 这里服务提供者没有设置 category, 注册中心对于没有设置的默认取 providers 值)

```
dubbo://10.33.37.4:20880/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=demo-provider&dubbo=2.5.4-SNAPSHOT&generic=false&interface=com.alibaba.dubbo.demo.DemoService&loadbalance=roundrobin&methods=sayHello&owner=william&pid=9828&side=provider&timestamp=1415968955329
```

然后注册中心回调服务消费者暴露的回调接口来对服务提供者的服务进行引用 refer 生成对应的可执行对象 invoker。服务提供者与服务的消费建立连接,

5. 通过 Cluster 合并 directory 中的 invokers, 返回可执行对象  
invoker

6. ProxyFactory.getProxy(invoker) 创建代理对象返回给业务方使用

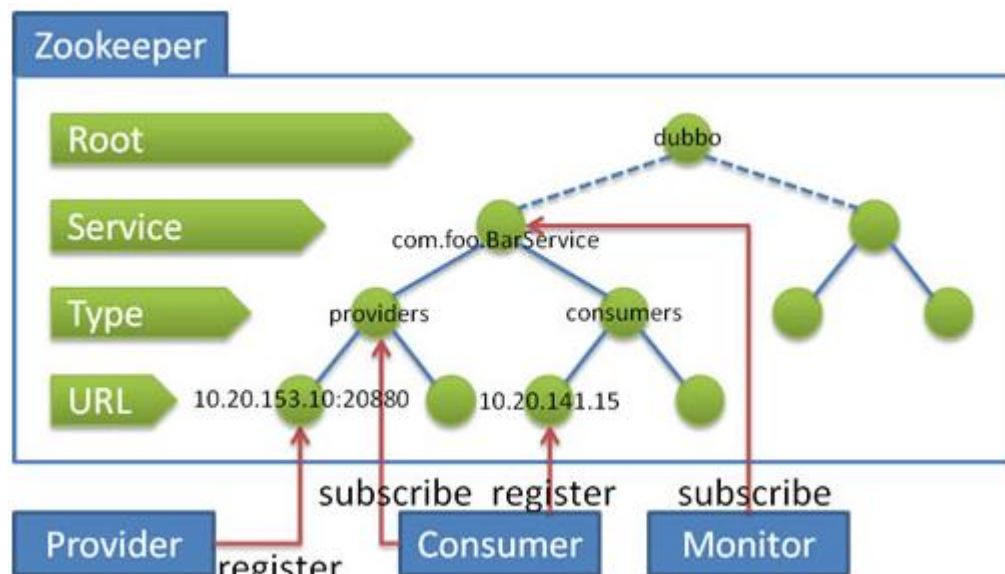
这里 dubbo 协议的注册中心调注册中心的服务采用的默认集群调用策略是 FailOver, 选择一台注册中心, 只有当失败的时候才重试其他服务器, 注册中心实现也比较简单不具备集群功能, 如果想要初步的集群功能可以选用 BroadcastCluster 它至少向每个注册中心遍历调用注册一遍

### 三: Zookeeper 协议注册中心

下面我们来看下开源 dubbo 推荐的业界成熟的 zookeeper 做为注册中心, zookeeper 是 hadoop 的一个子项目是分布式系统的可靠协调者, 他提供了配置维护, 名字服务, 分布式同步等服务。对于 zookeeper 的原理本文档不分析, 后面有时间在做专题。

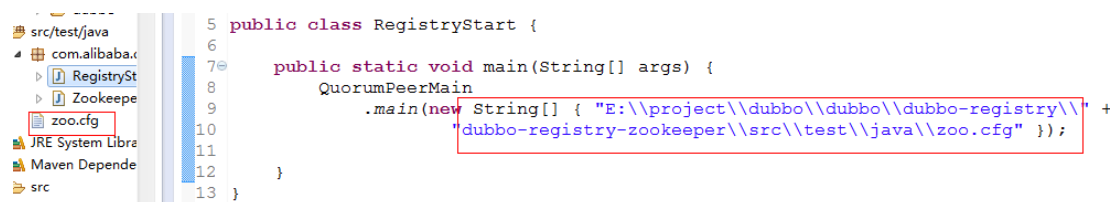
#### zookeeper 注册中心

Zookeeper 对数据存储类似 linux 的目录结构, 下面给出官方文档对 dubbo 注册数据的存储示例

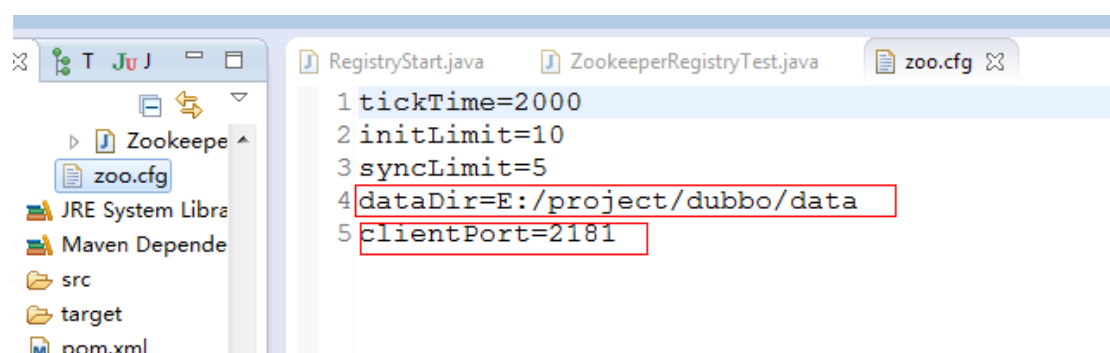


假设读者对 zookeeper 有所了解，能够搭建 zookeeper 服务，其实不了解也没关系，谷歌百度下分分钟搞起。

作为测试调试 dubbo，我是在本地起的 zookeeper



指定 zookeeper 配置文件地址



配置文件中两个关键参数：

dataDir zookeeper 存储文件的地址

clientPort 客户端链接的端口号

## Dubbo 服务提供者配置

```
<dubbo:registry protocol=" zookeeper" address="127.0.0.1:2181" />
```

```
<bean id="demoService" class="com.alibaba.dubbo.demo.provider.DemoServiceImpl" />
```

```
<dubbo:service interface="com.alibaba.dubbo.demo.DemoService"
ref="demoService" />
```

除了配置注册中心的，其他都一样

## Dubbo 服务消费者配置

```
<dubbo:registry protocol=" zookeeper" address="127.0.0.1:2181" />
```

```
<dubbo:reference id="demoService"
interface="com.alibaba.dubbo.demo.DemoService" />
```

除了配置注册中心的，其他都一样

## 客户端获取注册器

服务的提供者和消费者在 RegistryProtocol 利用注册中心暴露 (export) 和引用 (refer) 服务的时候会根据配置利用 Dubbo 的 SPI 机制获取具体注册中心注册器

Registry registry = registryFactory.getRegistry(url);

这里的RegistryFactory是ZookeeperRegistryFactory看如下工厂代码

```
public class ZookeeperRegistryFactory extends AbstractRegistryFactory {  
    public Registry createRegistry(URL url) {  
        return new ZookeeperRegistry(url, zookeeperTransporter);  
    }  
}
```

这里创建zookeeper注册器ZookeeperRegistry

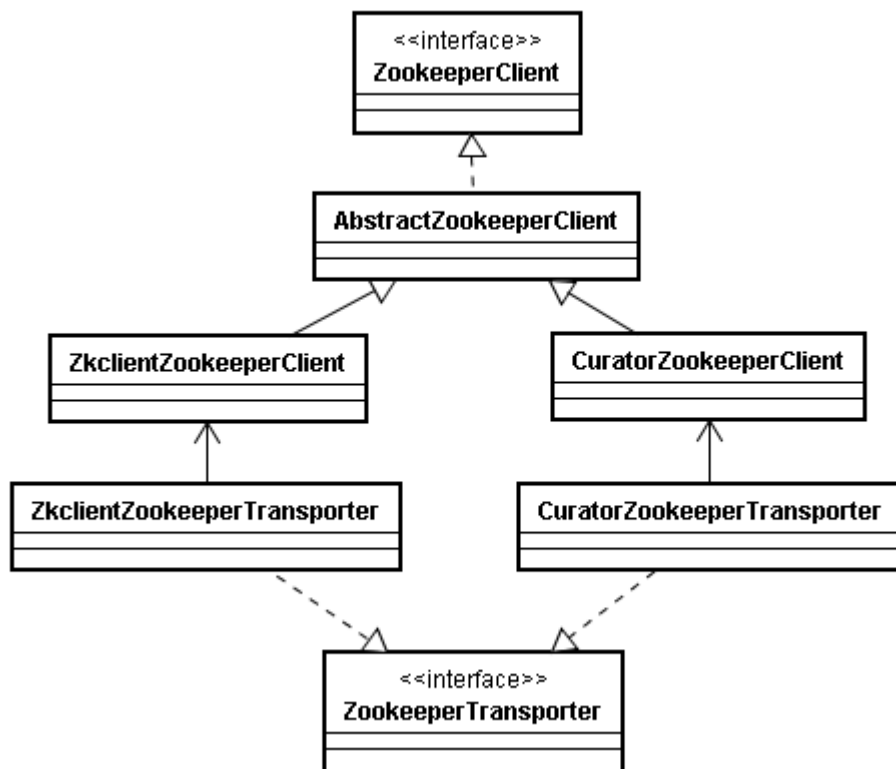
ZookeeperTransporter是操作zookeeper的客户端的工厂类，用来创建zookeeper客户端，这里客户端并不是zookeeper源代码的自带的，而是采用第三方工具包，主要来简化对zookeeper的操作，例如用zookeeper做注册中心需要对zookeeper节点添加watcher做反向推送，但是每次回调后节点的watcher都会被删除，这些客户会自动维护了这些watcher，在自动添加到节点上去。

接口定义：

@SPI("zkclient")

```
public interface ZookeeperTransporter {  
    @Adaptive({Constants.CLIENT_KEY, Constants.TRANSPORTER_KEY})  
    ZookeeperClient connect(URL url);  
}
```

默认采用zkClient， dubbo源码集成两种zookeeper客户端，除了zkClient还有一个是curator



## ZookeeperRegistry 注册器的实现

1. 构造器利用客户端创建了对 zookeeper 的连接，并且添加了自动回复连接的监听器。

```

zkClient = zookeeperTransporter.connect(url);
zkClient.addStateListener(new StateListener() {
    public void stateChanged(int state) {
        if (state == RECONNECTED)
            recover();
    }
});

```

2. 注册 url 就是利用客户端在服务器端创建 url 的节点，默认为临时节点，客户端与服务端断开，几点自动删除

```

zkClient.create(toUrlPath(url),
url.getParameter(Constants.DYNAMIC_KEY, true));

```

3. 取消注册的 url，就是利用 zookeeper 客户端删除 url 节点

```

zkClient.delete(toUrlPath(url));

```

4. 订阅 url， 功能是服务消费端订阅服务提供方在 zookeeper 上注册地址，这个功能流程跟 DubboRegister 不一样， DubboRegister 是通过 Dubbo 注册中心实现 SimpleResgiter 在注册中心端，对 url 变换、过滤筛选然后将获取的 provierUrl(提供者 ulr)利用服务消费者暴露的服务回调在 refer。



由于这里注册中心采用的是 zookeeper, zookeeper 不可能具有 dubbo 的业务逻辑, 这里对订阅的逻辑处理都在消费服务端订阅的时候处理。

1) 对传入 url 的 serviceInterface 是 \* 代表订阅 url 目录下所有节点即所有服务, 这个注册中心需要订阅所有

2) 如果指定了订阅接口通过 toCategoriesPath(url) 转换需要订阅的 url 如传入 url

```
consumer://10.33.37.8/com.alibaba.dubbo.demo.DemoService?application=demo-consumer&category=providers,configurators,routers&dubbo=2.5.4-SNAPSHOT&interface=com.alibaba.dubbo.demo.DemoService&methods=sayHello&pid=4088&side=consumer&timestamp=1417405597808
```

转换成 urls

```
/dubbo/com.alibaba.dubbo.demo.DemoService/providers,  
/dubbo/com.alibaba.dubbo.demo.DemoService/configurators,  
/dubbo/com.alibaba.dubbo.demo.DemoService/routers
```

3) 设配传入的回调接口 NotifyListener, 转换成 dubbo 对 zookeeper 操作的 ChildListener

4) 以 /dubbo/com.alibaba.dubbo.demo.DemoService/providers 为例创建节点 zkClient.create(path, false);

但是一般情况下如果服务提供者已经提供服务, 那么这个目录节点应该已经存在, Dubbo 在 Client 层屏蔽掉了创建异常。

5) 以 /dubbo/com.alibaba.dubbo.demo.DemoService/providers 为例给节点添加监听器, 返回所有子目录

```
List<String> children = zkClient.addChildListener(path,  
zkListener);  
if (children != null) {urls.addAll(toUrlsWithEmpty(url, path,  
children));}
```

toUrlsWithEmpty 用来配置是不是需要订阅的 url, 是加入集合

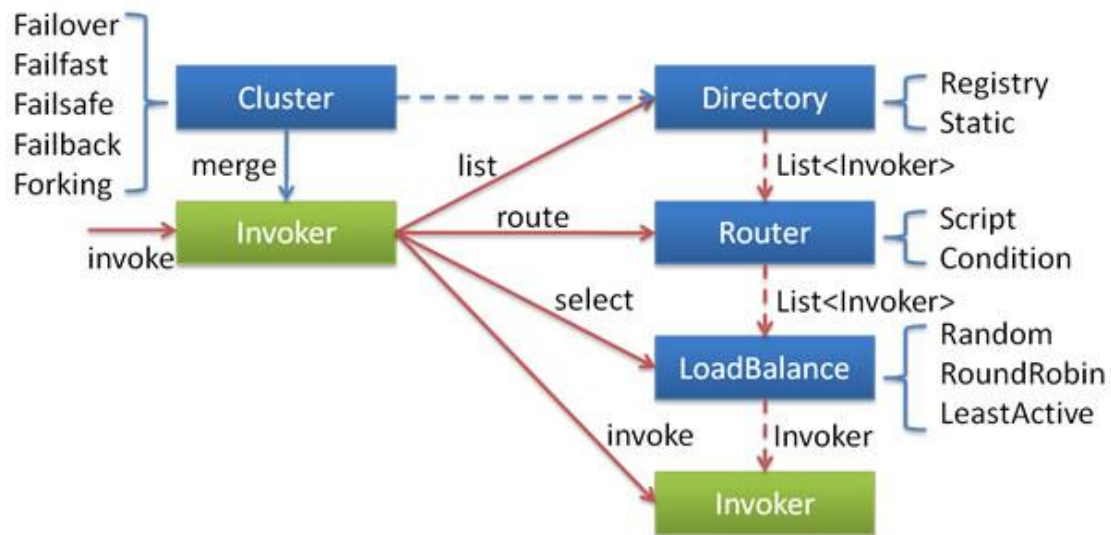
6) 主动根据得到服务提供者 urls 回调 NotifyListener, 引用服务提供者生成 invoker 可执行对象

5. 取消订阅 url, 只是去掉 url 上的注册的监听器

## 第八章：集群&容错

Dubbo 作为一个分布式的服务治理框架, 提供了集群部署, 路由, 软负载均衡及容错机制

下图描述了 dubbo 调用过程中的对于集群, 负载等的调用关系。



## 一：cluster

将 Directory 中的多个 Invoker 伪装成一个 Invoker，对上层透明，包含集群的容错机制

Cluster 接口定义

@SPI(FailoverCluster.*NAME*)

**public interface** Cluster {

@Adaptive

<T> Invoker<T> join(Directory<T> directory) **throws** RpcException;

}

Cluster 可以看做是工厂类，将目录 directory 下的 invoker 合并成一个统一的 Invoker，根据不同集群策略的 Cluster 创建不同的 Invoker

我们来看下默认的失败转移，当出现失败重试其他服务的策略，这个 Cluster 实现很简单就是创建 FailoverClusterInvoker 对象

**public class** FailoverCluster **implements** Cluster {

**public final static** String *NAME* = "failover";

**public** <T> Invoker<T> join(Directory<T> directory) **throws**

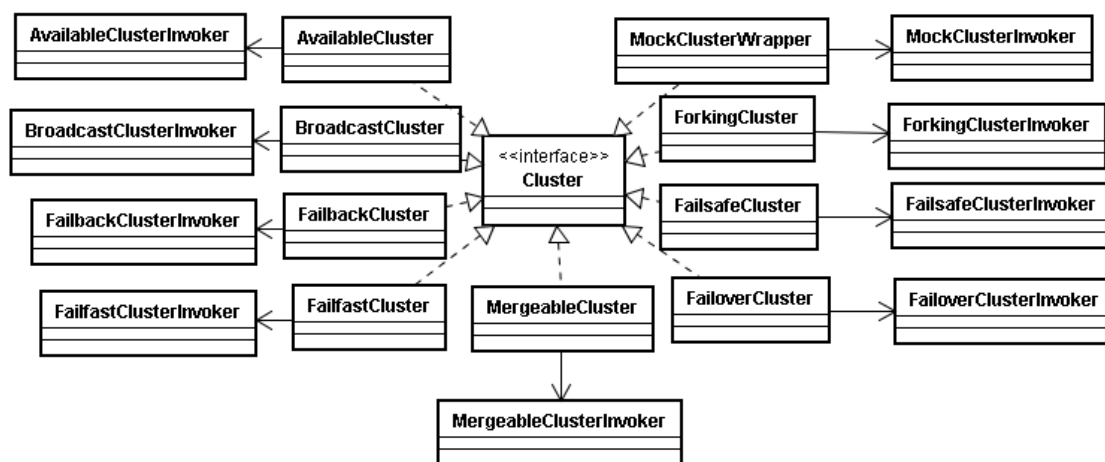
RpcException{

**return new** FailoverClusterInvoker<T>(directory);

}

}

下图展示了 dubbo 提供的所有集群方案



- 1) AvailableCluster: 获取可用的调用。遍历所有 Invokers 判断 `Invoker.isAvalible`, 只要一个有为 true 直接调用返回, 不管成不成功
- 2) BroadcastCluster: 广播调用。遍历所有 Invokers, 逐个调用每个调用 catch 住异常不影响其他 invoker 调用
- 3) FailbackCluster: 失败自动恢复, 对于 invoker 调用失败, 后台记录失败请求, 任务定时重发, 通常用于通知
- 4) FailfastCluster: 快速失败, 只发起一次调用, 失败立即报错, 通常用于非幂等性操作
- 5) FailoverCluster: 失败转移, 当出现失败, 重试其它服务器, 通常用于读操作, 但重试会带来更长延迟
  - (1) 目录服务 `directory.list(invocation)` 列出方法的所有可调用服务获取重试次数, 默认重试两次
  - (2) 根据 LoadBalance 负载策略选择一个 Invoker
  - (3) 执行 `invoker.invoke(invocation)` 调用
  - (4) 调用成功返回

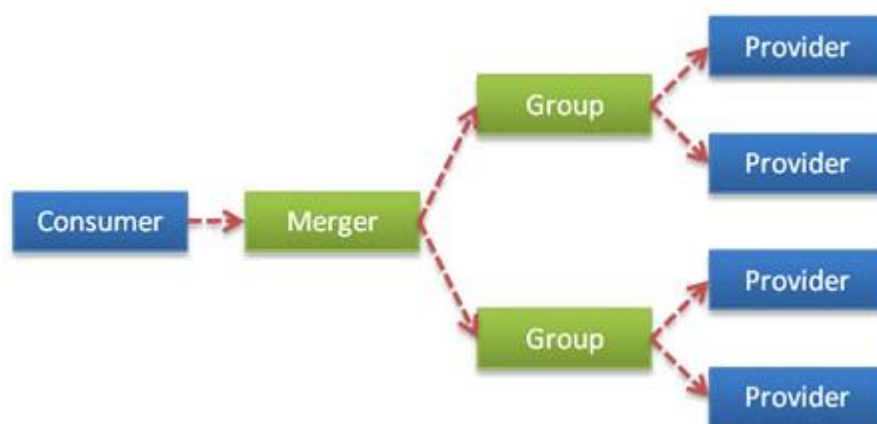
调用失败小于重试次数，重新执行从 3) 步骤开始执行

调用次数大于等于重试次数抛出调用失败异常

- 6) FailsafeCluster: 失败安全，出现异常时，直接忽略，通常用于写入审计日志等操作。
- 7) ForkingCluster: 并行调用，只要一个成功即返回，通常用于实时性要求较高的操作，但需要浪费更多服务资源。
- 8) MergeableCluster: 分组聚合，按组合并返回结果，比如菜单服务，接口一样，但有多种实现，用 group 区分，现在消费方需从每种 group 中调用一次返回结果，合并结果返回，这样就可以实现聚合菜单项。

这个还蛮有意思，我们分析下是如何实现的

- (1) 根据 MERGE\_KEY 从 url 获取参数值
- (2) 为空不需要 merge，正常调用
- (3) 按 group 分组调用，将返回接口保存到集合中
- (4) 获取 MERGE\_KEY 如果是默认的话，获取默认 merge 策略，主要根据返回类型判断
- (5) 如果不是，获取自定义的 merge 策略
- (6) Merge 策略合并调用结果返回



- 9) MockClusterWrapper: 具备调用 mock 功能其他 Cluster 包装获取 url 的 MOCK\_KEY 属性
  - (1) 不存在直接调用其他 cluster
  - (2) 存在值 startsWith("force") 强制 mock 调用
  - (3) 存在值不是 startsWith("force") 先正常调用，出现异常在 mock 调用

集群模式的配置

<dubbo:service cluster="failsafe" />      服务提供方  
<dubbo:reference cluster="failsafe" />    服务消费方

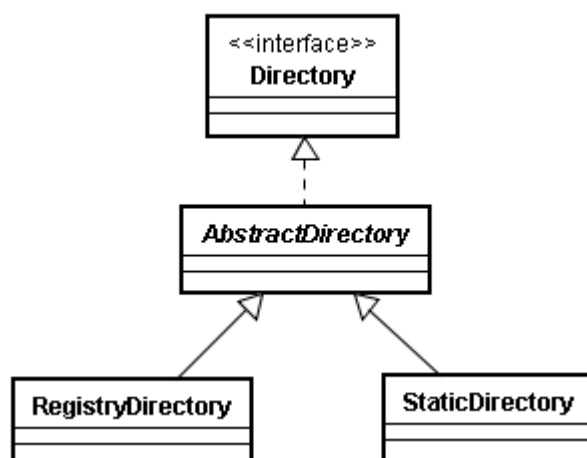
## 二：目录服务 Directory

集群目录服务 Directory，代表多个 Invoker，看成 List<Invoker>，它的值可能是动态变化的比如注册中心推送变更。集群选择调用服务时通过目录服务找到所有服务

Directory 的接口定义

```
public interface Directory<T> extends Node {  
    //服务类型  
  
    Class<T> getInterface();  
  
    //列出所有服务的可执行对象  
  
    List<Invoker<T>> list(Invocation invocation) throws  
RpcException;  
}
```

Directory有两个具体实现



**StaticDirectory**: 静态目录服务，它的所有Invoker通过构造函数传入，服务消费方引用服务的时候，服务对多注册中心的引用，将Invokers集合直接传入 StaticDirectory构造器，再由Cluster伪装成一个Invoker

```

396     for (URL url : urls) {
397         invokers.add(refprotocol.refer(interfaceClass, url));
398         if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {
399             registryURL = url; // 用了最后一个registry url
400         }
401     }
402     if (registryURL != null) { // 有注册中心协议的URL
403         // 对有注册中心的Cluster 只用 AvailableCluster
404         URL u = registryURL.addParameter(Constants.CLUSTER_KEY, AvailableCluster.NAME);
405         invoker = cluster.join(new StaticDirectory(u, invokers));
406     } else { // 不是注册中心的URL
407         invoker = cluster.join(new StaticDirectory(invokers));
408     }

```

StaticDirectory的list方法直接返回所有invoker集合

RegistryDirectory: 注册目录服务， 它的Invoker集合是从注册中心获取的，它实现了NotifyListener接口实现了回调接口notify(List<Url>)。比如消费方要调用某远程服务，会向注册中心订阅这个服务的所有服务提供方，订阅时和服务提供方数据有变动时回调消费方的NotifyListener服务的notify方法NotifyListener.notify(List<Url>) 回调接口传入所有服务的提供方的url地址然后将urls转化为invokers，也就是refer应用远程服务

```

381     }
382     keys.add(key);
383     // 缓存key为没有合并消费端参数的URL，不管消费端如何合并参数，如果服务端URL发生变化，则重新refer
384     Map<String, Invoker<T>> localUrlInvokerMap = this.urlInvokerMap; // local reference
385     Invoker<T> invoker = localUrlInvokerMap == null ? null : localUrlInvokerMap.get(key);
386     if (invoker == null) { // 缓存中没有，重新refer
387         try {
388             boolean enabled = true;
389             if (url.getParameter(Constants.DISABLED_KEY)) {
390                 enabled = ! url.getParameter(Constants.DISABLED_KEY, false);
391             } else {
392                 enabled = url.getParameter(Constants.ENABLED_KEY, true);
393             }
394             if (enabled) {
395                 invoker = new InvokerDelegete<T>(protocol.refer(serviceType, url), url, providerUrl);
396             }
397         } catch (Throwable t) {
398             logger.error("Failed to refer invoker for interface:" + serviceType + " url: (" + url + ") " + t.getMessage());

```

到此时引用某个远程服务的RegistryDirectory中有对这个远程服务调用的所有invokers。

RegistryDirectory.list(invocation)就是根据服务调用方法获取所有的远程服务引用的 invoker 执行对象

### 三：路由

Router服务路由， 根据路由规则从多个Invoker中选出一个子集AbstractDirectory是所有目录服务实现的上层抽象， 它在list列举出所有invokers后，会在通过Router服务进行路由过滤。

Router接口定义

```

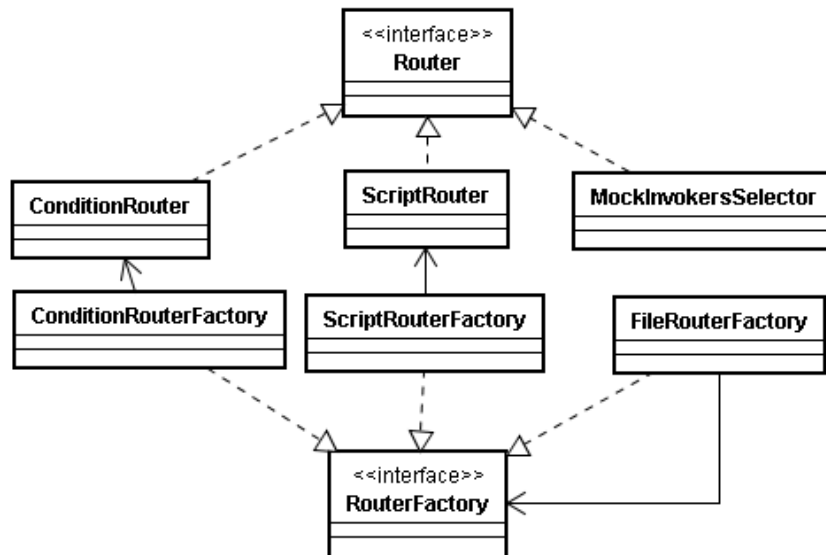
public interface Router extends Comparable<Router> {
    URL getUrl();

```

```

<T> List<Invoker<T>> route(List<Invoker<T>>
    invokers, URL url, Invocation invocation)
    throws RpcException;
}

```



ConditionRouter: 条件路由

我们这里简单分析下代码实现具体功能参考官方文档

条件表达式以 => 分割为whenRule和thenRule

ConditionRouter创建，构造器初始

- 1) 从url根据RULE\_KEY获取路由条件路由内容
- 2) rule.indexOf("=>") 分割路由内容
- 3) 分别调用parseRule(rule) 解析路由为whenRule和thenRules

ConditionRouter执行route方法

- 1) 如果url不满足when条件即过来条件， 不过滤返回所有invokers
- 2) 遍历所有invokers判断是否满足then条件， 将满足条件的加入集合result
- 3) Result不为空， 有满足条件的invokers返回
- 4) Result为空， 没有满足条件的invokers， 判断参数FORCE\_KEY是否强制过来， 如果强制过滤返回空， 不是返回所有即不过滤

ScriptRouter: 脚本路由，

通过url的RULE\_KEY参数获取脚本内容， 然后通过java的脚本引擎执行脚本代码， dubbo的测试用例都是通过javascript作为脚本但是理论上也支持groovy, jruby脚本， 大家可以参考下测试用例ScriptRouterTest。

ScriptRouter创建，构造器初始化

- 1) 从url获取脚本类型javascript, groovy等等
- 2) 从url根据RULE\_KEY获取路由规则内容
- 3) 根据脚本类型获取java支持的脚本执行引擎

ScriptRouter执行route方法

- 1) 执行引擎创建参数绑定
- 2) 绑定执行的参数
- 3) 执行引擎编译路由规则得到执行函数CompiledScript
- 4) CompiledScript.eval(binds) 根据参数执行路由规则

Dubbo提供了ConditionRouterFactory, ScriptRouterFactory来创建对应的路由，路由的规则从url的RULE\_KEY参数来获取，路由规则可以通过监控中心或者治理中心写入注册中心

```
RegistryFactory registryFactory =  
ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtension();  
Registry registry =  
registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));  
registry.register(URL.valueOf("condition://0.0.0.0/com.foo.BarService?category=routers&dynamic=false&rule=" +  
URL.encode("http://10.20.160.198/wiki/display/dubbo/host = 10.20.153.10 => host = 10.20.153.11") + "));
```

Dubbo 也支持通过 FileRouterFactory 从文件读取路由规则，将读取的规则设置到 url 的 RULE\_KEY 参数上，文件的后缀代表了路由的类型，选择具体的路由工厂 ConditionRouterFactory, ScriptRouterFactory 来创建路由规则

## 四：负载均衡

LoadBalance负载均衡，负责从多个 Invokers中选出具体的一个Invoker用于本次调用，调用过程中包含了负载均衡的算法，调用失败后需要重新选择

LoadBalance接口定义

```
@SPI(RandomLoadBalance.NAME)  
public interface LoadBalance {  
    @Adaptive("loadbalance")  
    <T> Invoker<T> select(List<Invoker<T>>
```



```

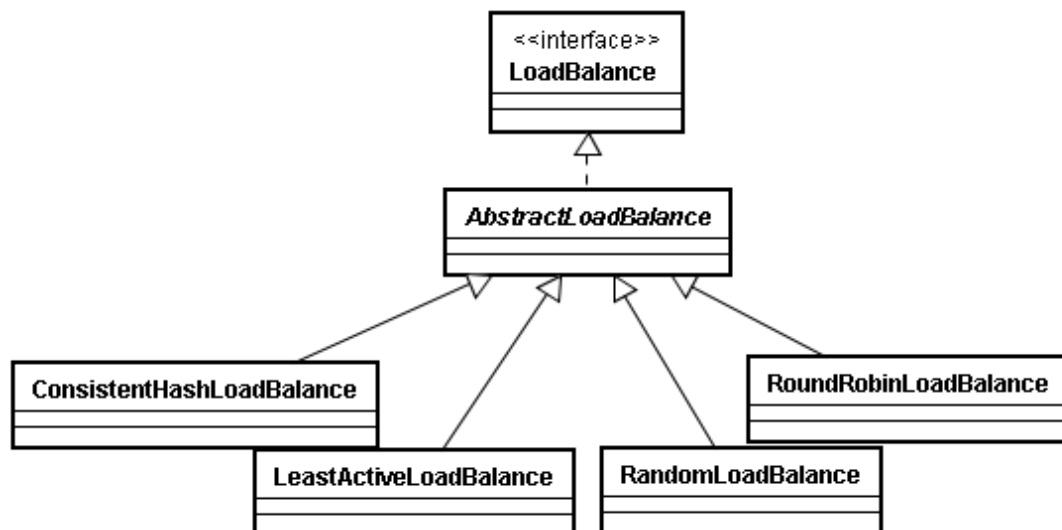
    invokers, URL url, Invocation invocation)
    throws RpcException;
}

```

类注解@SPI说明可以基于Dubbo的扩展机制进行自定义的负责均衡算法实现，默认是随机算法

方法注解@Adaptive说明能够生成设配方法

Select方法设配类通过url的参数选择具体的算法， 在从invokers集合中根据具体的算法选择一个invoker



RandomLoadBalance: 随机访问策略，按权重设置随机概率，是默认策略

- 1) 获取所有invokers的个数
- 2) 遍历所有Invokers, 获取计算每个invokers的权重，并把权重累计加起来  
每相邻的两个invoker比较他们的权重是否一样，有一个不一样说明权重不均等
- 3) 总权重大于零且权重不均等的情况下  
按总权重获取随机数offset = random.netx(totalWeight);  
遍历invokers确定随机数offset落在哪个片段(invoker上)

```

int offset = random.nextInt(totalWeight);
// 并确定随机值落在哪个片断上
for (int i = 0; i < length; i++) {
    offset -= getWeight(invokers.get(i), invocation);
    if (offset < 0) {
        return invokers.get(i);
    }
}

```

- 4) 权重相同或者总权重为0， 根据invokers个数均等选择  
invokers.get(random.nextInt(length))

RoundRobinLoadBalance: 轮询，按公约后的权重设置轮询比率

- 1) 获取轮询key 服务名+方法名  
获取可供调用的invokers个数length  
设置最大权重的默认值maxWeight=0  
设置最小权重的默认值minWeight=Integer.MAX\_VALUE
- 2) 遍历所有Inokers，比较出得出maxWeight和minWeight
- 3) 如果权重是不一样的  
根据key获取自增序列  
自增序列加一与最大权重取模默认得到currentWeight  
遍历所有invokers筛选出大于currentWeight的invokers  
设置可供调用的invokers的个数length
- 4) 自增序列加一并与length取模，从invokers获取invoker

LeastActiveLoadBalance: 最少活跃调用数， 相同的活跃的随机选择，  
活跃数是指调用前后的计数差， 使慢的提供者收到更少的请求，因为越慢的提供者前后的计数差越大。

活跃计数的功能消费者是在ActiveLimitFilter中设置的

```

long begin = System.currentTimeMillis();
RpcStatus beginCount(url, methodName);
try {
    Result result = invoker.invoke(invocation);
    RpcStatus endCount(url, methodName, System.currentTimeMillis() - begin, true);
    return result;
} catch (RuntimeException t) {
    RpcStatus endCount(url, methodName, System.currentTimeMillis() - begin, false);
    throw t;
}

```

最少活跃的选择过程如下：

- 1) 获取可调用invoker的总个数

初始化最小活跃数，相同最小活跃的个数

相同最小活跃数的下标数组

等等

- 2) 遍历所有invokers， 获取每个invoker的获取数active和权重

找出最小权重的invoker

如果有相同最小权重的inovkers， 将下标记录到数组leastIndexs[]数组中

累计所有的权重到totalWeight变量

- 3) 如果invokers的权重不相等且totalWeight大于0

按总权重随机offsetWeight = random.nextInt(totalWeight)

计算随机值在哪个片段上并返回invoker

```
int offsetWeight = random.nextInt(totalWeight);  
// 并确定随机值落在哪个片断上  
for (int i = 0; i < leastCount; i++) {  
    int leastIndex = leastIndexs[i];  
    offsetWeight -= getWeight(invokers.get(leastIndex), invocation);  
    if (offsetWeight <= 0)  
        return invokers.get(leastIndex);  
}
```

- 4) 如果invokers的权重相等或者totalWeight等于0，均等随机

ConsistentHashLoadBalance:一致性 hash，相同参数的请求总是发到同一个提供者，当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。对于一致性哈希算法介绍网上很多，这个给出一篇<http://blog.csdn.net/sparkliang/article/details/5279393> 供参考，读者请自行阅读 ConsistentashLoadBalance 中对一致性哈希算法的实现，还是比较通俗易懂的这里不再啰嗦。

## 第九章：服务调用

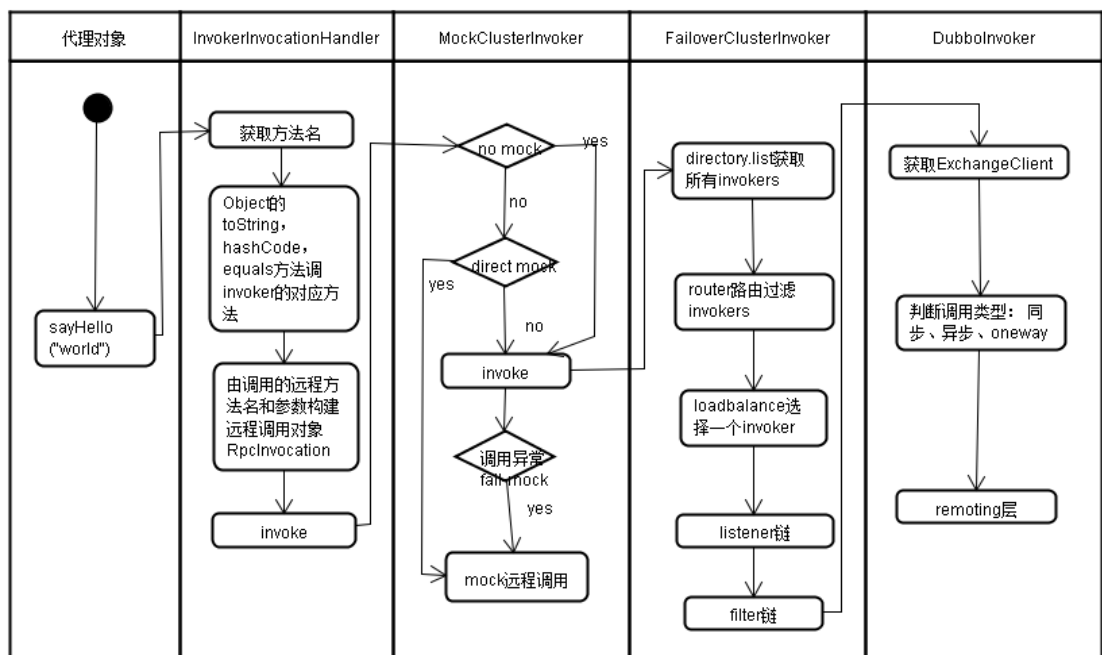
### 服务消费方发起请求

当服务的消费方引用了某远程服务，服务的应用方在 spring 的配置实例如下：

```
<dubbo:reference id="demoService"  
interface="com.alibaba.dubbo.demo.DemoService" />
```

demoService 实例其实是代理工厂生产的代理对象（大家可以参考代理那部分生成的伪代码），在代码中调用 demoService.sayHello("world!")时，

1. 将方法名方法参数传入 `InvokerInvocationHandler` 的 `invoke` 方法  
对于 `Object` 中的方法 `toString`, `hashCode`, `equals` 直接调用 `invoker` 的对应方法,  
这里对于 `Object` 的方法需要被远程调用吗? 调用了是不是报错比默认处理更好呢??  
远程调用层是以 `Invocation`, `Result` 为中心, 这里根据要调用的方法以及传入的参数构建 `RpcInvocation` 对象, 作为 `Invoker` 的入参
2. `MockClusterInvoker` 根据参数提供了三种调用策略  
不需要 `mock`, 直接调用 `FailoverClusterInvoker`  
强制 `mock`, 调用 `mock`  
先调 `FailoverClusterInvoker`, 调用失败在 `mock`、
3. `FailoverClusterInvoker` 默认调用策略  
通过目录服务查找到所有订阅的服务提供者的 `Invoker` 对象  
路由服务根据策略来过滤选择调用的 `Invokers`  
通过负载均衡策略 `LoadBalance` 来选择一个 `Invoker`
4. 执行选择的 `Invoker.inoker(invocation)`  
经过监听器链, 默认没有  
经过过滤器链, 内置实现了很多  
执行到远程调用的 `DubboInvoker`
5. `DubboInvoker`  
根据 `url` 也就是根据服务提供者的长连接, 这里封装成交互层对象 `ExchangeClient` 供这里调用  
判断远程调用类型同步, 异步还是 `oneway` 模式  
`ExchangeClient` 发起远程调用, 底层 `remoting` 不在这里描述了  
获取调用结果:
  - `Oneway` 返回空 `RpcResult`
  - 异步, 直接返回空 `RpcResult`, `ResponseFuture` 回调
  - 同步, `ResponseFuture` 模式同步转异步, 等待响应返回



## 服务提供方接收调用请求

同样我们也是 rpc 调用层 DubboProtocol 层开始分析，对于通信层 remoting 的数据接收反序列等等过程不做分析。

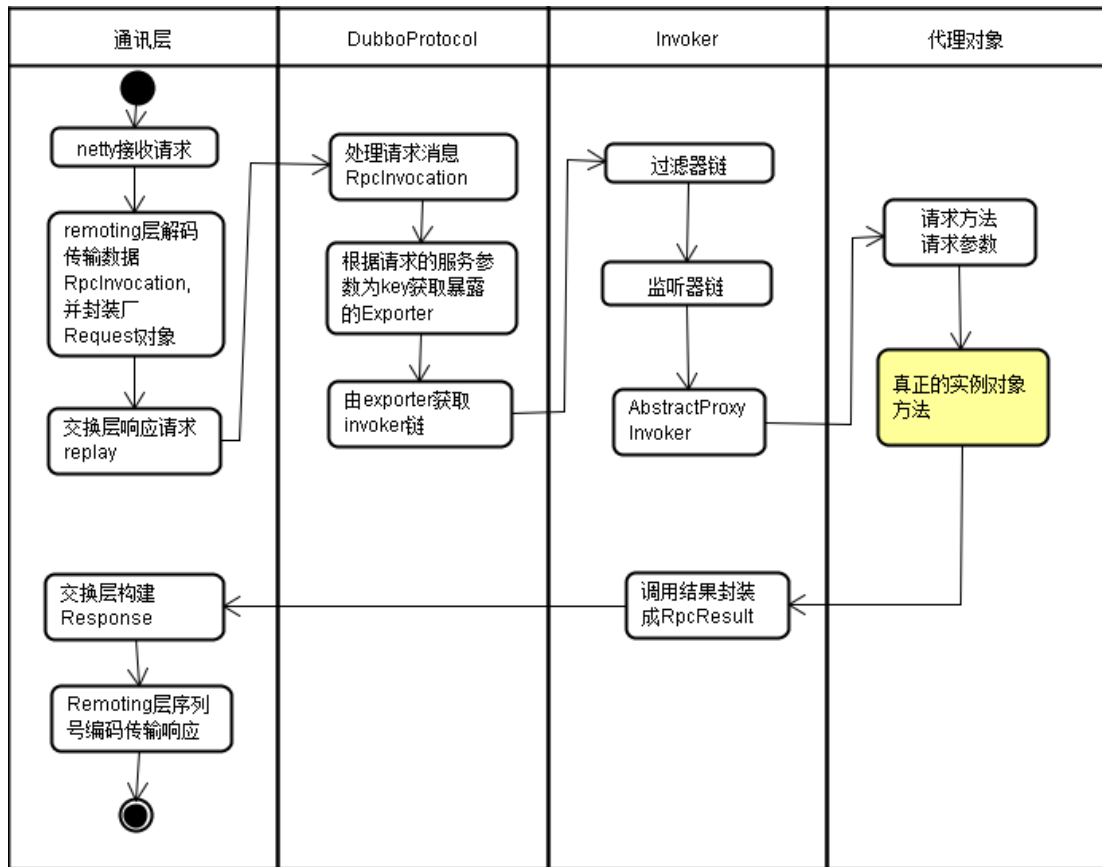
DubboProtocol 的 requestHandler 是 ExchangeHandler 的实现，是 remoting 层接收数据后的回调。

requestHandler.replay 方法接收请求消息，这里只处理远程调用消息 Invocation。

1. 通过 Invocation 获取服务名和端口组成 serviceKey=com.alibaba.dubbo.demo.DemoService:20880, 从 DubboProtocol 的 exproterMap 中获取暴露服务的 DubboExporter, 在从 dubboExporter 获取 invoker 返回
2. 经过过滤器链
3. 经过监听器链
4. 到达执行真正调用的 invoker，这个 invoker 由代理工厂 ProxyFactory.getInvoker(demoService, DemoService.class, registryUrl)创建，具体请看代理那部分介绍。

调用 demoService 实例方法，将结果封装成 RpcResult 返回

5. 交换层构建 Response，通过 Remoting 层编码传输将结果响应给调用方



## 第十章：通信层

Dubbo 的整个远程通信层由 exchange, transport, serialize  
exchange, 信息交换层, 封装请求响应模式, 同步转异步, 以 Request, Response  
为中心, 扩展接口为 Exchanger, ExchangeChannel,  
HeaderExchangeHandler ,ExchangeClient, ExchangeServer  
transport, 网络传输层, 抽象 mina 和 netty 为统一接口, 以 Message 为中心,  
扩展接口为 Channel, Transporter, Client, Server, Codec  
serialize, 数据序列化层, 可复用的一些工具, 扩展接口为 Serialization,  
ObjectInput, ObjectOutput, ThreadPool

## 一：提供方暴露服务

### 1. 服务提供方在 DubboProtocol 暴露服务 export 的过程

DubboProtocol 构建 requestHandler 实现了 reply 方法，用来响应服务调用方的请求，调用具体的服务，并将请求结果封装为 RpcResult 返回给调用方  
requestHandler 在创建服务 createServer 的 Exchanger bind 的时候传入，  
Exchangers.bind(url, requestHandler) //传入 url， 处理器，这个处理器会被层层装饰

根据 spi 策略 HeaderExchanger.bind(url, requestHandler)

构建 HeaderExchangeServer 对象，new

HeaderExchangeServer(Transporters.bind(url, new DecodeHandler(new HeaderExchangeHandler(handler))))

装饰 requestHandler 对象

根据 spi 策略 NettyTransporter.bind(url, channelHandler)，构建 NettyServer 对象，绑定 netty 服务启动服务监听

### 2. NettyServer 初始化过程，它是 dubbo 默认的通讯框架

构造器入参 URL 统一数据模型包含服务器端地址，超时时间等参数

构造器中装饰传入的 ChannelHandler，ChannelHandler 其实是一个处理器

链,transporter 层做了装饰，这里 NettyServer 构造其中通过

ChannelHandlers.wrap 方法再次进行装饰

RequestHandler→DecodeHandler→ HeaderExchangeHandler→

MultiMessageHandler→

HeartbeatHandler→ AllChannelHandler

NettyServer 的构造其中会调 doOpen 方法启动 netty。 Netty 是基于 pipeline 管道机制的，我们可以添加阀(org.jboss.netty.channel.ChannelHandler)来实现自己的需求

```
bootstrap.setPipelineFactory(new ChannelPipelineFactory() {  
    public ChannelPipeline getPipeline() {  
        NettyCodecAdapter adapter = new NettyCodecAdapter(getCodec(), getUrl(), NettyServer.this);  
        ChannelPipeline pipeline = Channels.pipeline();  
        /*int idleTimeout = getIdleTimeout();  
        if (idleTimeout > 10000) {  
            pipeline.addLast("timer", new IdleStateHandler(timer, idleTimeout / 1000, 0, 0));  
        }*/  
        pipeline.addLast("decoder", adapter.getDecoder());  
        pipeline.addLast("encoder", adapter.getEncoder());  
        pipeline.addLast("handler", nettyHandler);  
        return pipeline;  
    }  
});
```

Decoder: InternalDecoder 继承 netty 对象 SimpleChannelUpstreamHandler

Dubbo 自定义了一套 buffer 用来适配底层的 nio， 自定的 channelBuffer 读取 netty 传递过的字节然后调用 dubbo 的编码解码器 codec.decode 解码成 Request/Response 对象返回

触发 netty 的 messageReceived 事件

Encoder: InternalEncoder 继承于 netty 的 OneToOneEncoder， 将传入的 Request/Response/String 等对象类型转换成字节流

注：为什么 Decoder 对象没有直接继承于 netty 的 OneToOneDecoder 对象能，因为 decoder

时候要把字节流读入到 dubbo 自定义的 buffer 中，然后再调 codec，这样 codec 的接口就不直接依赖于 nio 的 api，便于 codec 的扩展及替换

3. NettyHandler: 继承于 netty 对象 SimpleChannelHandler 触发 netty 事件来处理 dubbo 需求，它会触发一系列的 dubbo 的 ChannelHandler。它是一个处理链 AllChannelHandler→ MultiMessageHandler→ HeartbeatHandler→ DecodeHandler HeaderExchangeHandler → RequestHandler

AllChannelHandler: 默认分发请求实现对于 netty 事件中除了 writeRequested 的其他请求 channelConnected, channelDisconnected, messageReceived, exceptionCaught 这四种事件通过构建 ChannelEventRunnable 任务放入线程池分发

MultiMessageHandler: 对于消息的 received 事件，如果消息是批量传输的 decode 后会由 MultiMessage 承载，如果是 MultiMessage 遍历其中的消息分别交由下一个 handler 处理

```
public void received(Channel channel, Object message) throws RemotingException {
    if (message instanceof MultiMessage) {
        MultiMessage list = (MultiMessage)message;
        for(Object obj : list) {
            handler.received(channel, obj);
        }
    } else {
        handler.received(channel, message);
    }
}
```

4. HeartbeatHandler: 处理心跳

Connected 事件设置读写时间戳为当前时间

Disconnected 事件清空读写时间戳

Sent 事件发送请求设置写时间戳为当前时间

Received 事件接收请求设置读时间戳为当前时间

- 1) 判断消息是心跳请求，当 Request 对象的 mEvent 属性为 true 且 mData 为 null 时为心跳事件。

如果此心跳事件需要响应即 Request.isTwoway(), 构建心跳响应对象

Response, 设置响应对象的 mEvent=true && mData=null

下面 channel 处理，最终会通过 netty 响应客户端。

- 2) 判断消息是心跳响应，Response 的 mEvent=true && mData=null，记录日志直接返回，不在做后续的 channel 处理了。

- 3) 如果不是心跳处理，交由下一个处理器处理。

5. DecodeHandler: 对于读取的消息或者消息的数据实现了 Decodeable 接口的 decode



```

public void received(Channel channel, Object message) throws RemotingException {
    if (message instanceof Decodeable) {
        decode(message);
    }

    if (message instanceof Request) {
        decode(((Request)message).getData());
    }

    if (message instanceof Response) {
        decode(((Response)message).getResult());
    }

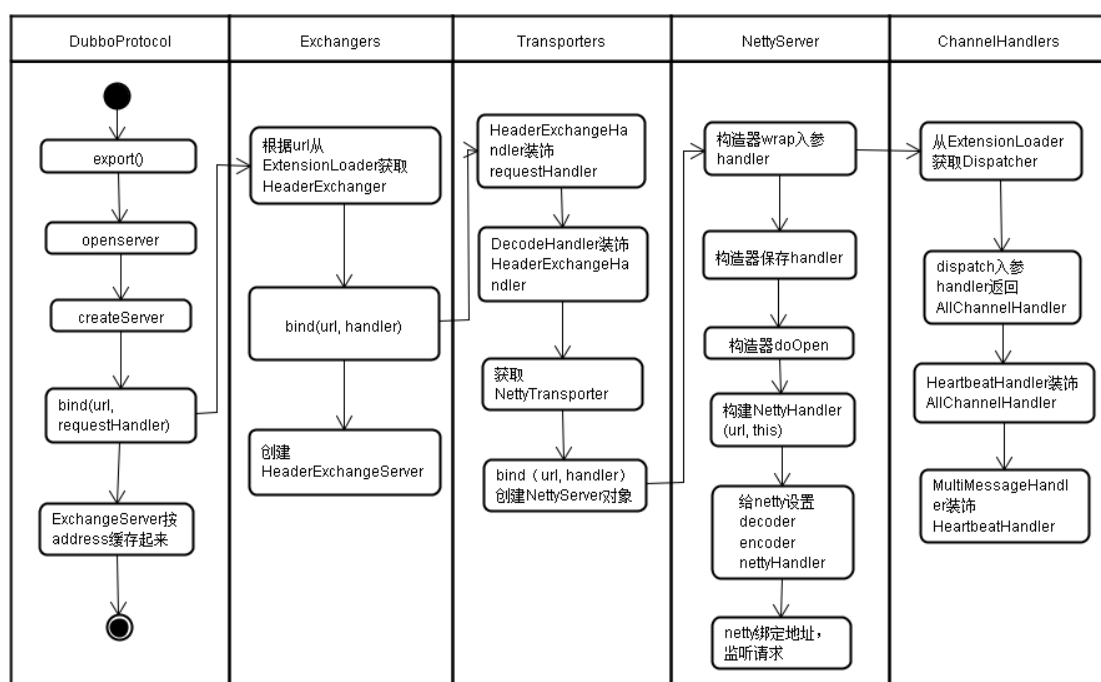
    handler.received(channel, message);
}

```

6. HeaderExchangeHandler: 这里已经到了交换层的 handler 了, 对于这个 handler 我们主要关心 sent 事件: 设置 channel 的写时间戳, 交由后面 hanler 来发送消息, 如果消息是 request, 根据 request 获取 DefaultFuture 设置它的发送时间为当前时间, 用来判断是否超时。received 事件: 接收读取数据

- 1) 给 channel 设置读取时间戳
- 2) 如果是 msg 是请求类型的 Request, 根据 request 的 id 构建 response 对象  
调 DubboProtocol 中构建的 requestHandler 的 replay 方法, 返回调用结果  
跟 response 对象设置返回结果  
Channel.send(response) 向客户端发送响应
- 3) 如果读取的数据是 Response 响应并且不是心跳  
根据 response 透传的 id 获取 defaultFuture, 设置 defaultFuture 的 response 的值, 唤醒线程, defaultFuture 的 get 方法返回调用值
- 4) 如果是 string 类型, 处理 telnet 命令
- 5) 其他的交由下一个 handler 处理 //什么场景??

下面给出服务提供方服务绑定的活动图



## 二：费方引用服务

服务调用方在引用服务 refer 时候创建对服务提供者的链接: 构建 DubboInvoker 时候需要获取 ExchangeClient 作为构造器参数传入

Exchangers.connect(url, requestHanler)→ HeaderExchanger.connect(url, exchangeHandler)

构建 HeaderExchangeClient, 获取传输层作为构造器参数

Transporters.connect(url, channelHannler)→ NettyTransporter.connect(url, channelHandler)

构建 NettyClient(url, channelHandler)返回

HeaderExchange 中装饰 ChannelHandler new DecodeHandler(new HeaderExchange Handler(requestHandler)))

在 NettyClient 构造器中装饰了 ChannelHandler

new MultiMessageHandler(new

HeartbeatHandler(ExtensionLoader.getExtension

Loader(Dispatcher.class).getAdaptiveExtension().dispatch(handler, url)))

ExtensionLoader.getExtensionLoader(Dispatcher.class).getAdaptiveExtension().dispatch(handler, url))= AllChannelHandler

requestHandler 是在 DubboProtocol 中实现的 ExchangeHandlerAdapter

DubboInvoker→ReferenceCountExchangeClient →HeaderExchangeClient→

HeaderExchangeChannel→NettyClient

DubboInvoker 根据 Invocation 对象判断是同步异步还是单向调用。选取 ExchangeClient 如果是共享的返回唯一的那个, 如果不是, 轮询获取一个。

ReferenceCountExchangeClient.request 没做什么事情, 直接调下一个

HeaderExchangeClient.request 调 ExchangeChannel 的 request 方法

HeaderExchangeChannel 的 request 方法

将请求转换成 Request 对象, 构建 Request 对象, 将入参(如: Invocation)

设置到 data 属性上, 构建 DefaultFuture, 这里同步转异步返回

由 HeaderExchanger 构建的 client 根据 spi 策略默认为 NettyClient, 所以下一步回调 nettyClient.send(request)

返回 DefaultFuture 对象

NettyClient 的 send 流程

获取 com.alibaba.dubbo.remoting.transport.netty.NettyChannel,

NettyChannel 覆写 client 方法, 根据 netty 跟 server 建立的链接获取的

org.jboss.netty.channel.Channel, 在利用 NettyChannel 来构建获取从缓存

总获取。com.alibaba.dubbo.remoting.transport.netty.NettyChannel 封装了

netty 通道 channel, 统一数据模型 url 以及 channelHandler

下一步调 NettyChannel 的 send 方法

NettyChannel 的 send 流程， 直接向 org.jboss.netty.channel.Channel 中写入 Request 数据  
DefaultFuture.get() 获取响应接口

NettyHandler 继承了 netty 的 SimpleChannelHandler 类  
messageReceived 接收对方的数据

获取 nettychannel, nettyClient.receive(nettychannel, response)

NettyClient.receive

MultiMessageHandler.receive(nettyChannel, response)

HeartbeatHandler.receive(nettyChannel, response)

设置 read 时间戳

如果心跳请求，发送心跳

如果心跳响应 返回

其他下一步 handler 处理

AllChannelHandler, 构建异步执行任务 ChannelEventRunnable, 异步执行

ChannelEventRunnable.receive(nettychannel, response)

DecodeHandler.receive(nettychannel, response)

Decode(response.getResult)

下一步 handler 处理

HeaderExchangeHandler.receive(nettychannel, response)

给 nettychannel 设置 READ\_TIMESTAMP 时间戳

从缓存获取或者构建 HeaderExchangeChannel

handleResponse(headerExchangeChannel, response)如果不是心跳

DefaultFuture.receive(headerExchangeChannel, response)

根据 response 的 id 获取请求时构建的 DefaultFuture

doReceive(response) 赋值 response 对象值, done.signal()唤醒 Condition

前端同步 get 流程

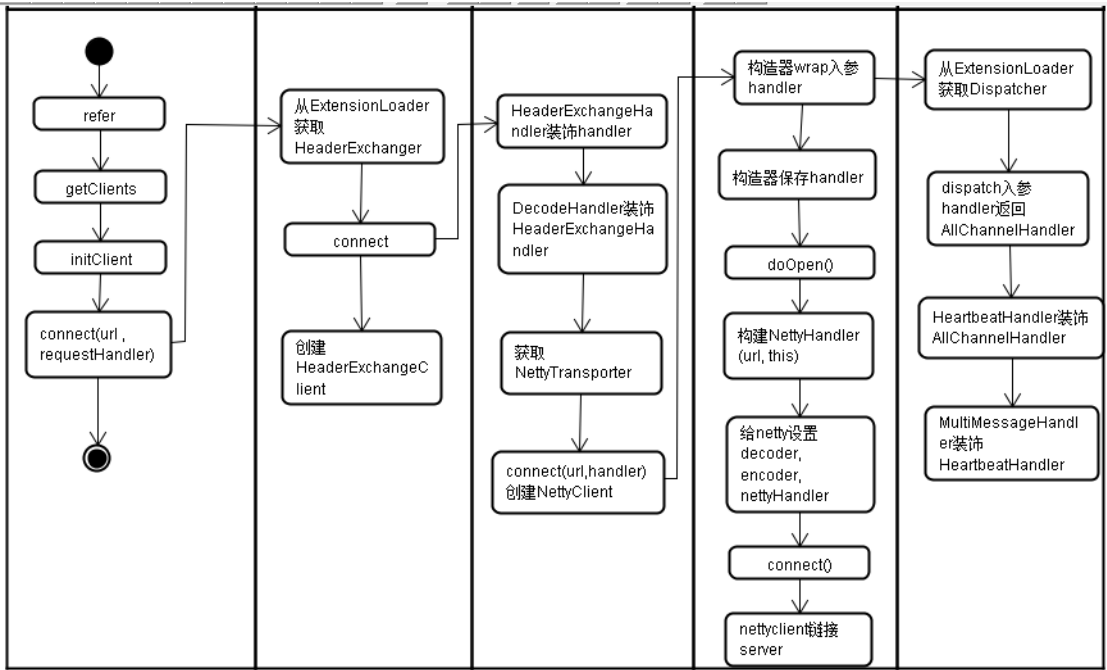
DefaultFuture.get()

done.await(timeout, TimeUnit.MILLISECONDS);

被唤醒后从 returnFromResponse()方法获取, response.getResult() 返回

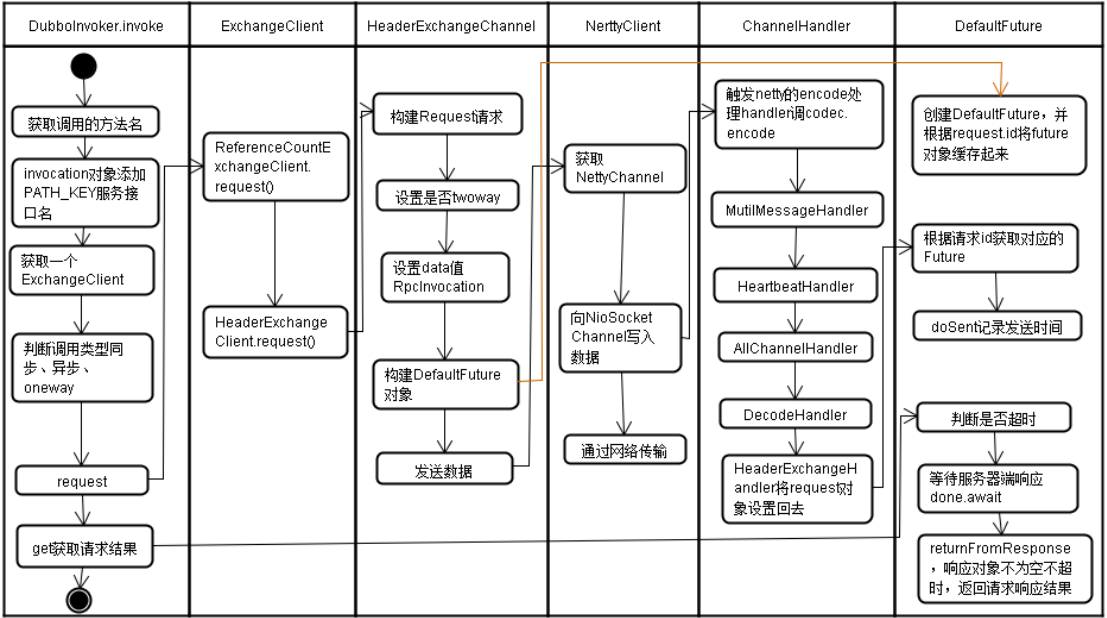
DecodeableRpcResult 对象

下面给出服务消费方连接服务提供方的 connect 过程

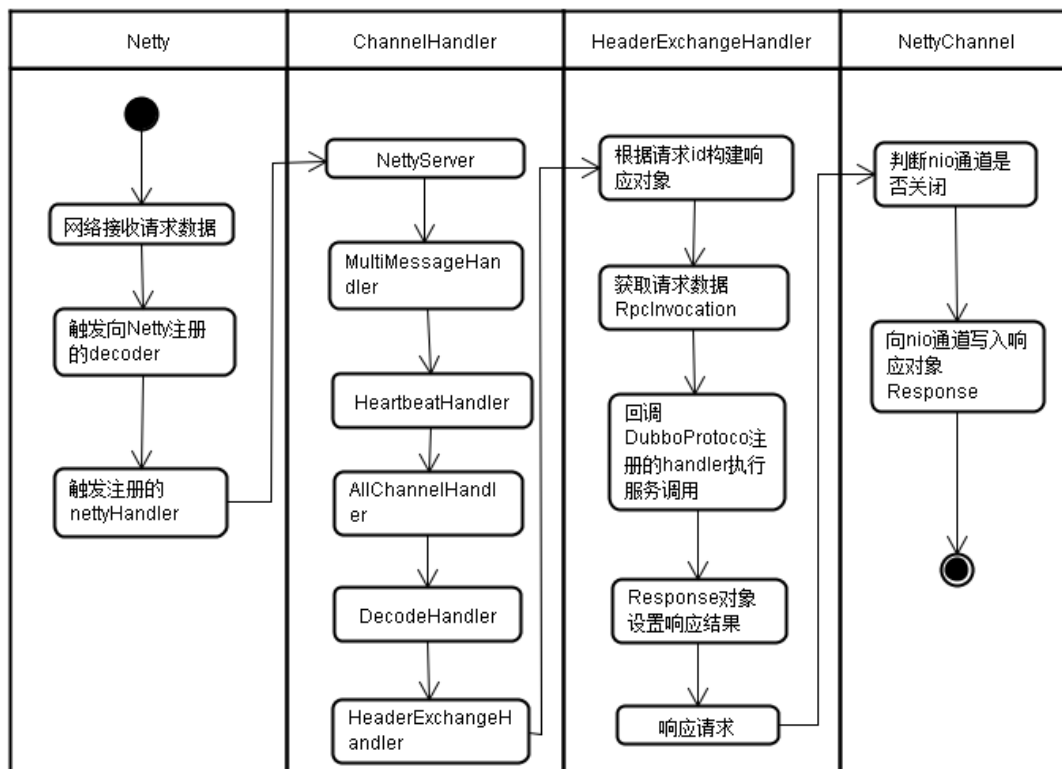


三：请求响应活动图

服务消费方发起远程调用的底层通信



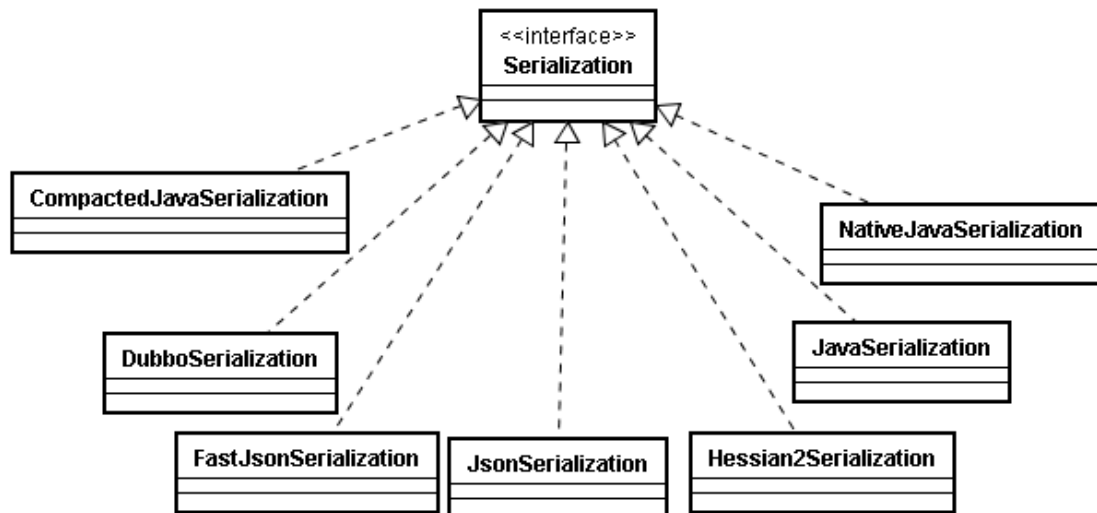
服务提供方接收请求并响应的底层通信



## 第十一章：编码解码

### 一：Serialization 接口定义

序列化：dubbo 提供了一系列的序列化反序列化对象工具。



Serialization 接口定义

```

@SPI("hessian2")
public interface Serialization {
    byte getContentTypeId();
    String getContentType();
    @Adaptive
    ObjectOutput serialize(URL url, OutputStream output)
    throws IOException;

    @Adaptive
    ObjectInput deserialize(URL url, InputStream input)
    throws IOException;
}
  
```

```

DubboSerialization
contentType=1
contentType="x-application/nativejava"
  
```

```

Hessian2Serialization
contentType=2
contentType="x-application/hessian2"
  
```

```

JavaSerialization
contentType=3
contentType="x-application/java"
  
```

```

CompactedJavaSerialization
contentType=4
contentType="x-application/compactedjava"
  
```

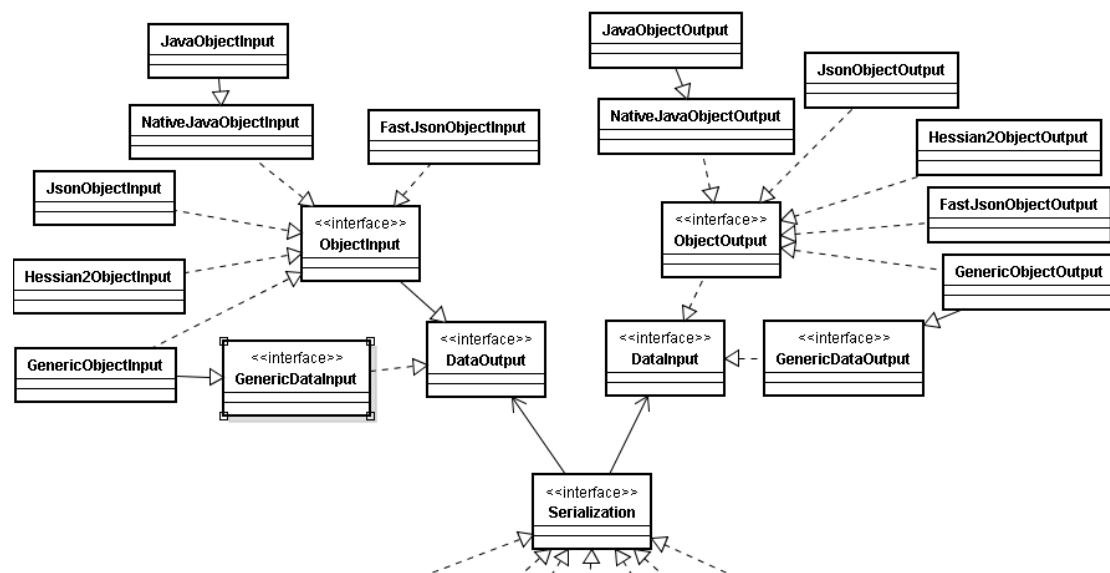
```
JsonSerialization
contentType=5
contentType= "text/json"
```

```
FastJsonSerialization
contentType=6
contentType= "text/json"
```

```
NativeJavaSerialization
contentType=7
contentType= "x-application/nativejava"
```

SPI 注解指定了序列化的默认实现为 hessian2

Serialization 依赖于 jdk 的 OutputStream, InputStream, 因为各具体的序列化工具依赖于 OutputStream, InputStream。又为了屏蔽各个序列化接口对 dubbo 侵入 dubbo 定义统一的 DataOutput DataInput 接口来适配各种序列化工具的输入输出



我们拿默认的序列化 Hessian2Serialization 来举例来说明

```
public class Hessian2Serialization implements Serialization{
    public ObjectOutput serialize(URL url, OutputStream out)
    throws IOException {
        return new Hessian2ObjectOutput(out);
    }
}
```

```

    }
    public ObjectInput deserialize(URL url, InputStream is)
    throws IOException {
        return new Hessian2ObjectInput(is);
    }
}

```

Hessian2Serialization 构建基于 Hessian 的 Dubbo 接口 Output,Input 实现，Dubbo 是基于 Output, Input 接口操作不需要关心具体的序列化反序列化实现方式。

```

public class Hessian2ObjectOutput implements ObjectOutput{
    private final Hessian2Output mH2o; //构建hessian操作类
    public Hessian2ObjectOutput(OutputStream os) {
        mH2o = new Hessian2Output(os);
        mH2o.setSerializerFactory(Hessian2SerializerFactory.SERIALIZER_FACTORY);
    }
    public void writeObject(Object obj) throws IOException{
        mH2o.writeObject(obj); //利用Hessian序列化对象
    }
    public void writeInt(short v) throws IOException {
        mH2o.writeInt(v); //利用Hessian序列化int类型
    }
    ..... //为了篇幅省略类似操作
}

```

Hessian2ObjectInput 读取 Hessian 序列化的数据使用方式同上面类似就不在堆代码了请自己翻看源代码

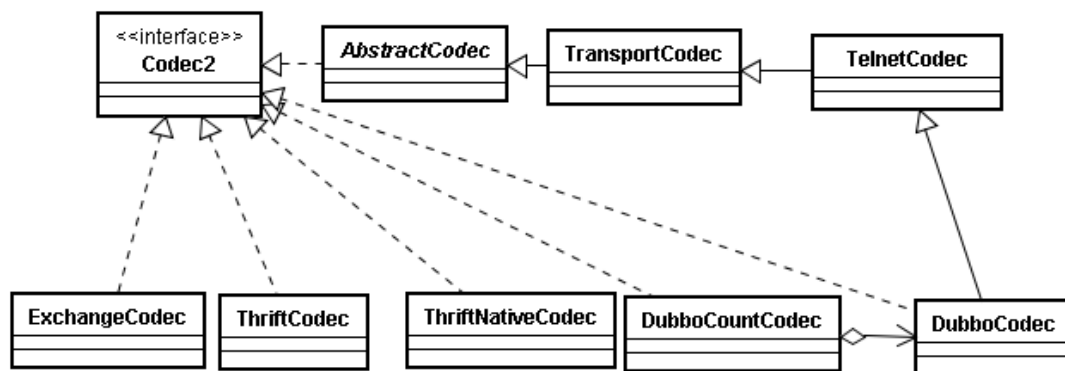
## 二：Codec2 接口定义

Dubbo 的远程调用需要对传输的数据进行编码解码，dubbo 的 Codec2 接口定义了编码解码规范，与废弃的接口 Codec 相比，Codec2 没有依赖 jdk 的输入



输出流，以 dubbo 的 ChannelBuffer 为核心便于更好的整合

```
@SPI
public interface Codec2 {
    @Adaptive({Constants.CODEC_KEY})
    void encode(Channel channel, ChannelBuffer buffer, Object
message) throws IOException;
    @Adaptive({Constants.CODEC_KEY})
    Object decode(Channel channel, ChannelBuffer buffer)
throws IOException;
}
```



### TransportCodec:

传输层的编码解码，比较通用没有具体的协议编码

```
public void encode(Channel channel, ChannelBuffer buffer,
    Object message) throws IOException {
    OutputStream output = new
        ChannelBufferOutputStream(buffer);
    ObjectOutputStream objectOutput =
        getSerialization(channel).serialize(channel.get
        Url(), output);
    encodeData(channel, objectOutput, message);
    objectOutput.flushBuffer();
}
```

1. 构建 ChannelBufferOutputStream，是的 buffer 具有 jdk  
OutputStream 的 api 操作功能，因为序列化工具都是基于 jdkAPI 的
2. getSerialization(channel) 通过 Dubbo 的 SPI 扩展机制得到具体的  
序列化工具
3. encodeData 这里只是将数据序列化后写入传输通道

```

public Object decode(Channel channel, ChannelBuffer buffer)
    throws IOException {
    InputStream input = newChannelBufferInputStream(buffer);
    return decodeData(channel,
        getSerialization(channel).deserialize(channel.
            getUrl(), input));
}

```

1. 构建 ChannelBufferInputStream 是的序列化工具能够通过 jdk 的 api 读取 channelBuffer 数据的功能
2. 通过 Dubbo 的 SPI 扩展机制得到具体的序列化实现进行反序列实现
3. decodeData 这里只是获取反序列化对象

### ExchangeCodec :

交换层是基于请求响应 request/response 的，在传输层之上封装了 Request, Response, ExchangeCodec 层的编码解码就是正对 Request, Response 的编码解码

DubboCodec: 主要是对于 dubbo 的远程调用请求对象 DecodeableRpcInvocation 以及请求返回结果 DecodeableRpcResult 的编码解码。

### 三：编码解码流程

这里把 ExchangeCodec 和 DubboCodec 放一起来讲解 dubbo 传输的底层协议组成以及它的编码解码过程。

#### 传输协议

协议格式<header><body data>

协议头：header 是 16 个字节的定长数据

= 2 //short 类型的 MAGIC = (short) 0xdabb

```

+ 1 // 一个字节的消息标志位，用来表示消息是 request 还是
    //response,twoway 还是 oneway, 是心跳还是正常请求以及采用
    //的序列化反序列化协议

+ 1 //状态位， 消息类型为 response 时，设置请求响应状态

+ 8 //设置消息的 id long 类型

+ 4 //设置消息体 body 长度 int 类型

```

Body data: body 是消息传递的真正内容，body 的占用的字节大小由协议头后四位保存。Body 的内容：

Request.getData() 得到 dubbo 请求消息传输的 body 对象 RpcInvocation，对于请求对象 dubbo 其实是知道需要传输哪些信息的，所以并没有把整个 RpcInvocation 对象序列化传输，而是序列化传输必要字段信息，下面列举出具体信息：

1. dubbo 的版本信息
2. 服务接口名如：com.zhanqiu.DemoService
3. 服务的版本号
4. 调服务的方法名
5. 调服务的方法的参数描述符如：[int.class, boolean[].class, Object.class] => "I[ZLjava/lang/Object;"
6. 遍历传输的参数值逐个序列化
7. 将整个附属信息 map 对象 attachments 序列化

下图是序列化 Request 的 body 的具体代码：

```

protected void encodeRequestData(Channel channel, ObjectOutput out, Object data)
    RpcInvocation inv = (RpcInvocation) data;

    out.writeUTF(inv.getAttachment(Constants.DUBBO_VERSION_KEY, DUBBO_VERSION));
    out.writeUTF(inv.getAttachment(Constants.PATH_KEY));
    out.writeUTF(inv.getAttachment(Constants.VERSION_KEY));

    out.writeUTF(inv.getMethodName());
    out.writeUTF(ReflectUtils.getDesc(inv.getParameterTypes()));
    Object[] args = inv.getArguments();
    if (args != null)
        for (int i = 0; i < args.length; i++){
            out.writeObject(encodeInvocationArgument(channel, inv, i));
        }
    out.writeObject(inv.getAttachments());
}

```

Response.getResult() 获取 Result 对象。根据 Result 对象是否有异常对象序列里化异常对象， 如果没有获取 result.getValue() 此对象为真正返回的业务对象消息的 body 数据， 整体序列化 result.getValue() 返回的对象。

```

protected void encodeResponseData(Channel channel, ObjectOutput out, Object data)
    Result result = (Result) data;

    Throwable th = result.getException();
    if (th == null) {
        Object ret = result.getValue();
        if (ret == null) {
            out.writeByte(RESPONSE_NULL_VALUE);
        } else {
            out.writeByte(RESPONSE_VALUE);
            out.writeObject(ret);
        }
    } else {
        out.writeByte(RESPONSE_WITH_EXCEPTION);
        out.writeObject(th);
    }
}

```

### 编码整体流程:

1. 判断消息类型是 Request, Resonse 如果不是调父类 (可能是 string telnet 类型)
2. 获取序列化方式, 可以同 URL 指定, 没有默认为 hessian 方式
3. 构建存储 header 的字节数组, 大小是 16
4. Header 数组前两位写入 dubbo 协议魔数 (short) 0xdabb
5. Header 数组第三位, 一个字节 4 位与或方式存储,
  - 1) 哪种序列化方式

- 2) 请求还是响应消息
  - 3) 请求时 twoway 还是 oneway
  - 4) 是心跳, 还是正常消息
  - 5) 如果是 response, 响应的状态
6. 获取 buffer 的写入位置 writerIndex, 加上消息头长度 16, 重新设置 buffer 的写入位置, 这里是消息 body 的写入位置, 因为后面是先写 body, 要把 header 的位置空出来
7. 序列化消息 body, (request, response 参考前面的) 写入 buffer
8. 计算消息体大小 writerIndex - startIndex
9. 检查消息体是否超过限制大小
10. 重置 writeIndex 就是第 6 点获取的值
11. 给消息头数组最后四位写入消息 body 长度值 int 类型
12. 向 buffer 写入消息头数据
13. Buffer 设置 writerIndex=savedWriteIndex + HEADER\_LENGTH + len

### 解码整体流程:

1. 从 channel 获取可读取的字节数 readable
2. readable 跟 header\_length 取小构建字节数组 header[]  
  
readable < header\_length 说明不是一个全的 dubbo 协议消息 (所以后面要判断消息头魔数), 或者只是一个 telnet 消息
3. 如果判断 header[] 的第一个和第二个字节不是 dubbo 协议魔数
  - a) 如果可读取字节 readable 大于 header\_length, 重新构建 header[], 读取全部可读取 readable 数据到 header
  - b) 遍历 header 中的字节判断中间是否有 dubbo 协议的魔数 0xdabb, 如

果有说明魔数开始的是 dubbo 协议的消息。

重新设置 buffer 的读取位置从魔数开始

截取 header[] 中从头开始到魔术位置的数据

c) 调父类解码，可能就是 telnet 字符串解码

4. 如果是 dubbo 协议的消息 `readable < header_length` 说明读取

`header[]` 数据不全， 返回 `NEED_MORE_INPUT` 说明需要读取更多数据

5. 读取 `header[]` 最后四位一个 `int` 的数据， `bodydata` 的长度 `len`

6. 计算总消息大小 `tt = len + body_length`

7. 如果可读取数据 `readable < tt`， 数据不够返回 `NEED_MORE_INPUT`

8. 由 `buffer` 和 `len` 构建 `ChannelBufferInputStream`， 后面序列化工具会使用

9. 下面是解码消息体 `body data` 过程

9.1 `header[2]` 第三位获取标志位

9.2 从标志位判断采用哪种序列化方式， 获取具体的序列化工具

9.3 读取 `header[4]` 开始的 8 位， 获取消息的 `id`

9.4 根据标志位判读消息为响应类型

a) 构建 `resonse` 对象 `new Response(id)`

b) 根据标志位如果是心跳， 给 `response` 对象设置 `Event` 类型

c) 从 `header[3]` 获取消息响应状态， 给 `response` 对象设置消息状态

d) 如果是事件消息直接利用反序列化工具读取对象

e) 如果不是构建消息接口 `DecodeableRpcResult result` 序列化工具

读取请求结果并设置到 `result` 的 `value` 属性上

f) 如果响应状态不是 `ok`， 反序列化 `errMessage` 并设置给 `response`

### 9.5 根据标志位判断消息为请求类型

- a) 根据 id 构建 Request(id)
- b) 根据状态位设置请求模式 twoway 还是 oneway
- c) 根据状态位设置请求是否是事件类型
- d) 如果是事件类型直接通过反序列化工具读取
- e) 如果不是事件请求，构建 DecodeableRpcInvocation

通过反序列化工具

读取 dubbo 版本，设置到 invocation 的 map 中

读取 path 服务名，设置到 invocation 的 map 中

读取服务版本，设置到 invocation 的 map 中

读取调用服务的方法， 设置到 invocation 的 methodName 属性上

读取方法描述符， 得到入参类型

遍历读取入参参数值

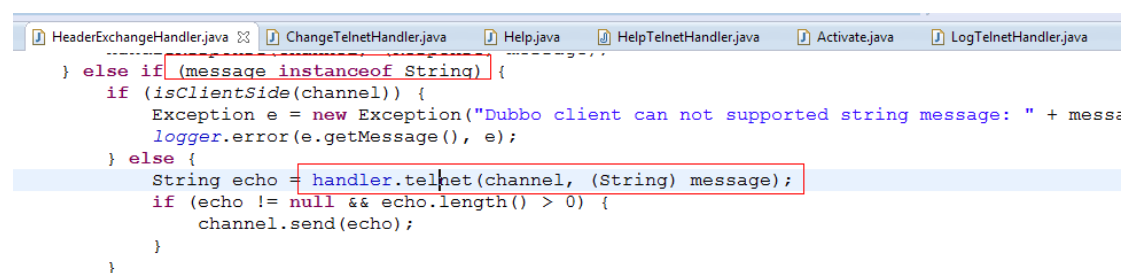
读取 map 对象不为空添加到 invocation 的 map 中

## 第十二章： telnet

Dubbo 提供了 telnet 命令去查看服务功能

```
telnet 127.0.0.1 20880
help
```

这里主要介绍一下 dubbo 实现 telnet 命令的整体实现



```
    } else if (message instanceof String) {
        if (isClientSide(channel)) {
            Exception e = new Exception("Dubbo client can not supported string message: " + message);
            logger.error(e.getMessage(), e);
        } else {
            String echo = handler.telnet(channel, (String) message);
            if (echo != null && echo.length() > 0) {
                channel.send(echo);
            }
        }
    }
}
```

当服务器端接收到的消息类型是 string 的时候回调用到 TelnetHandler 的 telnet 方法中

TelnetHandlerAdapter 类会从接收的字符串解析出命令，根据 dubbo 的 spi 扩展机制获取对应的 TelnetHandler 实现

```
if (command.length() > 0) {
    if (extensionLoader.hasExtension(command)) {
        try {
            String result = extensionLoader.getExtension(command).telnet(channel, message);
            if (result == null) {
                return null;
            }
            buf.append(result);
        } catch (Throwable t) {
            buf.append(t.getMessage());
        }
    }
}
```

在 com.alibaba.dubbo.remoting.telnet.TelnetHandler 多个文件中有如下配置

```
clear=com.alibaba.dubbo.remoting.telnet.support.command.ClearTelnetHandler
exit=com.alibaba.dubbo.remoting.telnet.support.command.ExitTelnetHandler
help=com.alibaba.dubbo.remoting.telnet.support.command.HelpTelnetHandler
```



```
lpTelnetHandler
```

```
。 。 。 。 。 。
```

对于 `telnet` 功能的实现方式跟其他的功能类似，由于每个 `TelnetHandler` 实现太细了，这里对有兴趣的读者自己翻看源码

## 第十四章：监控

Dubbo 发布代码中，自带了一个简易的监控中心实现。对于一般的小业务这个监控中心应该能够满足需求，对于那些大业务量的大公司一般都会有自己的监控中心，更加丰富的功能如常用的报警短信通知等等。这章讲解分析使得读者能够了解一般的监控中心实现，也使得有自己接入监控中心需求的大概知道如何集成自己的监控中心实现。下面我们就以 dubbo 自带的监控中心开始讲解。

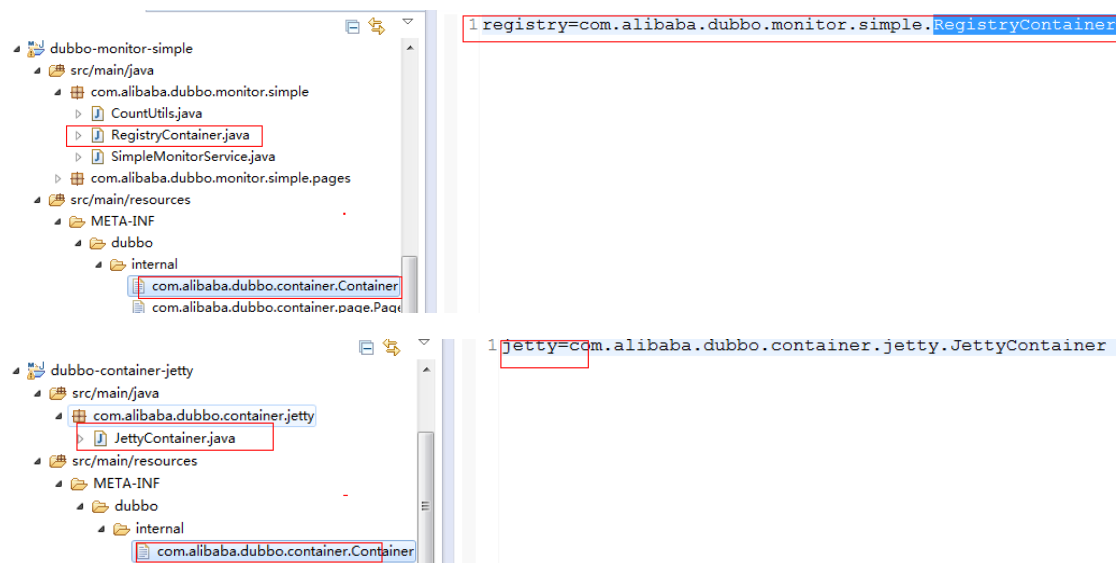
### 一：监控中心

1. 监控中心启动，我们先看下 dubbo 的属性文件

```
dubbo.container=log4j,spring,registry,jetty
dubbo.application.name=simple-monitor
dubbo.application.owner=
dubbo.registry.address=zookeeper://127.0.0.1:2181
dubbo.protocol.port=7070
dubbo.jetty.port=8080
dubbo.jetty.directory=${user.home}/monitor
dubbo.charts.directory=${dubbo.jetty.directory}/charts
dubbo.statistics.directory=${user.home}/monitor/statistics
```

相比于 `provider`，`consumer` 的启动注册中心多了 `registry`，`jetty` 容器

## 启动



它们都是基于 dubbo 的 spi 扩展机制的。

SpringContainer 容器启动就是加载 classpath\*:META-INF/spring/

\*.xml spring 的配置文件

```
<bean id="monitorService"
class="com.alibaba.dubbo.monitor.simple.SimpleMonitorService">
    <property name="statisticsDirectory"
value="${dubbo.statistics.directory}" />
    <property name="chartsDirectory"
value="${dubbo.charts.directory}" />
</bean>
<dubbo:application name="${dubbo.application.name}"
owner="${dubbo.application.owner}" />
<dubbo:registry address="${dubbo.registry.address}" />
<dubbo:protocol name="dubbo" port="${dubbo.protocol.port}" />
<dubbo:service
interface="com.alibaba.dubbo.monitor.MonitorService"
ref="monitorService" delay="-1" />
<dubbo:reference id="registryService"
interface="com.alibaba.dubbo.registry.RegistryService" />
```

## 2. SimpleMonitorService

监控中心配置了监控服务的实现 SimpleMonitorService, 并且作为一个普通的 dubbo 服务暴露到注册中心, 供服务的提供者和服务的消费方调用, 将服务提供者和服务的消费方的调用数据保存到监控中心。

## 监控服务的接口定义

```
public interface MonitorService {  
    /**  
        * 监控数据采集。  
  
        * 1. 支持调用次数统计：  
  
count://host/interface?application=foo&method=foo&provider=10.20.153.11:20880&success=12&failure=2&elapsed=135423423  
  
        * 1.1 host,application,interface,group,version,method  
记录监控来源主机，应用，接口，方法信息。  
  
        * 1.2 如果是消费者发送的数据，加上provider地址参数，反之，加上  
来源consumer地址参数。  
  
        * 1.3 success,faulure,elapsed 记录距上次采集，调用的成功次数，失败次数，成功调用总耗时，平均时间将用总耗时除以成功次数。  
        *  
        * @param statistics  
        */  
    void collect(URL statistics);  
    /**  
        * 监控数据查询。  
  
        * 1. 支持按天查询：  
  
count://host/interface?application=foo&method=foo&side=provider&view=chart&date=2012-07-03  
  
        * 1.1 host,application,interface,group,version,method  
查询主机，应用，接口，方法的匹配条件，缺失的条件表示全部，host用0.0.0.0表示全部。  
  
        * 1.2 side=consumer,provider 查询由调用的哪一端采集的数据，缺省为都查询。  
  
        * 1.3 缺省为view=summary，返回全天汇总信息，支持view=chart  
表示返回全天趋势图表图片的URL地址，可以进接嵌入其它系统的页面上展示。  
  
        * 1.4 date=2012-07-03 指定查询数据的日期，缺省为当天。
```

```

    *
    * @param query
    * @return statistics
    */
    List<URL> lookup(URL query);
}

```

注: lookup 方面可能在开源过程中依赖了阿里的什么系统, 并没有具体的实现, 如果使用着需要此功能则需要根据接口定义自己实现

MonitorService 的 dubbo 默认实现 SimpleMonitorService

Collect 方法被远程调用后将数据 url (传过来的 url 包含监控需要的数据) 保存到一个阻塞队列中 BlockingQueue<URL>

启动定时任务将统计日志记录到本地,

```

String filename = ${user.home}/monitor/statistics
                + "/" + day
                + "/" +
statistics.getServiceInterface()
                + "/" + statistics.getParameter(METHOD)
                + "/" + consumer
                + "/" + provider
                + "/" + type + "." + key

```

monitor ▶ statistics ▶ 20141117 ▶ com.alibaba.dubbo.demo.DemoService ▶ sayHello ▶ 10.33.37.6 ▶ 10.33.37.6				
(H)				
刻录 新建文件夹				
名称	修改日期	类型	大小	
consumer.concurrent	2014/11/17 17:51	CONCURRENT ...	3 KB	
consumer.elapsed	2014/11/17 17:51	ELAPSED 文件	4 KB	
consumer.failure	2014/11/17 17:51	FAILURE 文件	3 KB	
consumer.max.concurrent	2014/11/17 17:51	CONCURRENT ...	3 KB	
consumer.max.elapsed	2014/11/17 17:51	ELAPSED 文件	3 KB	
consumer.success	2014/11/17 17:51	SUCCESS 文件	4 KB	
provider.concurrent	2014/11/17 17:51	CONCURRENT ...	3 KB	
provider.elapsed	2014/11/17 17:51	ELAPSED 文件	4 KB	
provider.failure	2014/11/17 17:51	FAILURE 文件	3 KB	
provider.max.concurrent	2014/11/17 17:51	CONCURRENT ...	3 KB	
provider.max.elapsed	2014/11/17 17:51	ELAPSED 文件	3 KB	
provider.success	2014/11/17 17:51	SUCCESS 文件	4 KB	

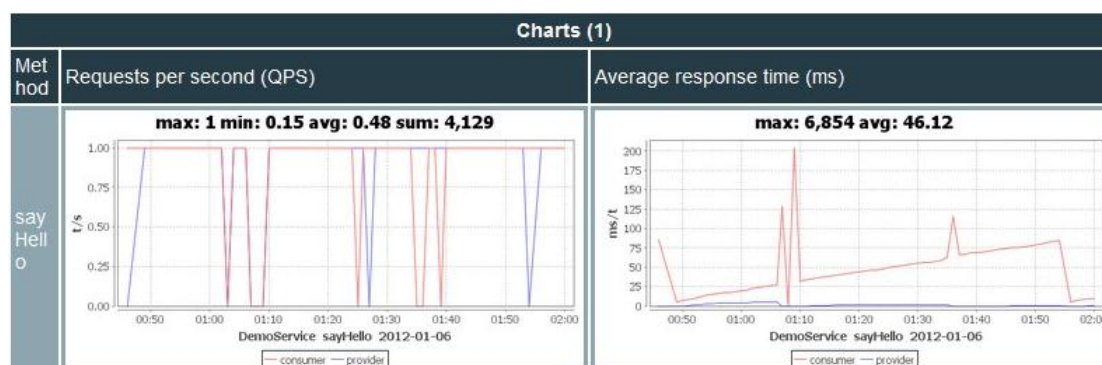
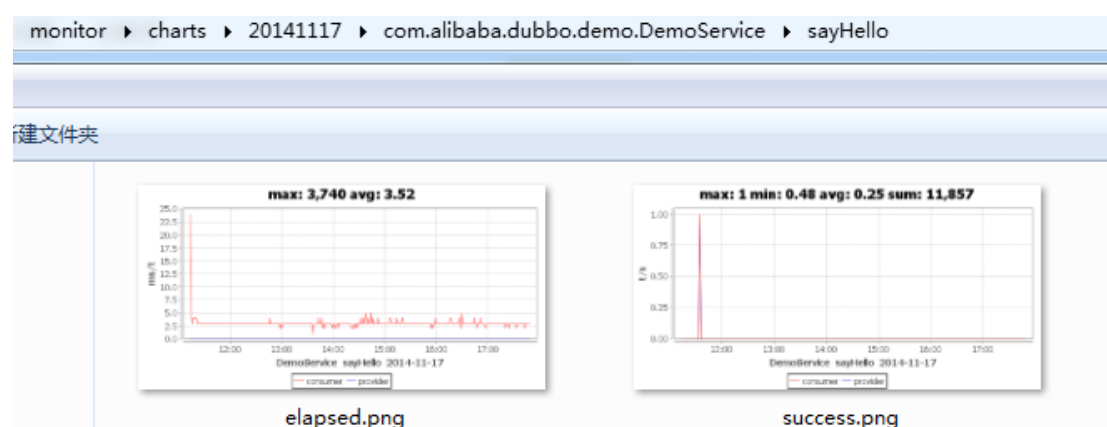
这是文件在本地存储的格式

consumer.elapsed		
1	1114	749
2	1115	123
3	1116	114
4	1117	142
5	1118	131
6	1119	137

文件内容如图保存时间 方法消费耗时

3. 起定时任务利用 JFreeChart 绘制图表, 保存路径

`${user.home}\monitor\charts\date\interfaceName\methodName`

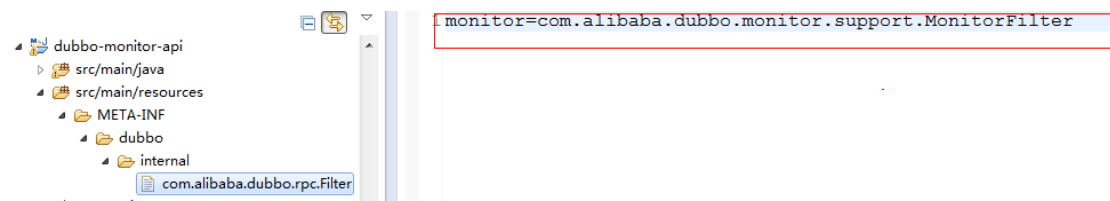


## 二：产生监控数据

注册中心暴露了 MonitorService 服务，它是被谁调用的呢，监控中心的数据是从哪里来呢，下面我们看下服务提供方与服务的消费方式如何介入监控中心的。在服务的提供方跟消费方的 dubbo 配置加入如下配置

通过注册中心 `<dubbo:monitor protocol="registry" />`

或者直连 `<dubbo:monitor address="127.0.0.1:7070" />`



在构建服务的调用链的时候有如上基于监控的扩展，下面我们就来看下这个类  
`@Activate(group = {Constants.PROVIDER, Constants.CONSUMER})`  
//此过滤器在服务的提供方，服务的消费方应用中被激活，也就是起作用

```
public class MonitorFilter implements Filter {  
    private MonitorFactory monitorFactory;  
    public Result invoke(Invoker<?> invoker,  
Invocation invocation) throws RpcException {  
        if  
(invoker.getUrl().hasParameter(Constants.MONITOR_KEY)) {  
            //有注监控中心处理  
  
            1. 获取invoker的调用上下文  
            2. 记录起始时间戳  
            3. 并发计数加一  
            try {  
                4. 调用调用链的下一步  
                5. 采集调用信息  
            } finally {  
                6. 并发计数减一  
            }  
        } else {  
            //没有配置监控中心，直接往下调用  
  
            return invoker.inovke(invocation);  
        }  
    }  
}
```

上面第 5 点信息采集

1. 计算调用耗时
2. 获取并发数
3. 获取服务名称
4. 获取方法名
5. 判断是服务消费方监控还是服务提供方监控
6. 由工厂类 `monitorFactory.getMonitor(监控 url)`，获取 `DubboMonitor` 对象，

构建调用监控中心服务的的 Url， url 中包括了监控中心所需的监控信息

```
monitor.collect(new URL(Constants.COUNT_PROTOCOL,
    NetUtils.getLocalHost(), localPort,
    service + "/" + method,
    MonitorService.APPLICATION, application,
    MonitorService.INTERFACE, service,
    MonitorService.METHOD, method,
    remoteKey, remoteValue,
    error ? MonitorService.FAILURE :
    MonitorService.SUCCESS, "1",
    MonitorService.ELAPSED,
    String.valueOf(elapsed),
    MonitorService.CONCURRENT,
    String.valueOf(concurrent),
    Constants.INPUT_KEY, input,
    Constants.OUTPUT_KEY, output));
```

DubboMonitor 是调用监控中心的服务的封装，之所以没有直接调监控中心而是通过 DubboMonitor 调用，是因为监控是附加功能，不应该影响主链路更不应该损害主链路的新能，DubboMonitor 采集到数据后通过任务定时调用监控中心服务将数据提交到监控中心。

### 三：RegistryContainer

监控中心 refer 引用了注册中心暴露的 RegistryService 服务，主要是被下面的 RegistryContainer 使用的。

RegistryContainer 主要是到注册中心收集服务，分组，版本信息，并注册回调当注册中心数据发生变化时候更新到监控中心

下面看下 RegistryContainer 的 start 方法流程：

1. 通过 SpringContainer 获取前面初始化的 RegistryService，得到其实是对注册中心的一个远程代理服务
2. 构建订阅注册中心数据的 URL，看可以看出下面的 url 是订阅服务提供者和服务消费者的所有服务

```
subscribeUrl = new URL(Constants.ADMIN_PROTOCOL,
    NetUtils.getLocalHost(), 0, "",
    Constants.INTERFACE_KEY, Constants.ANY_VALUE, //所有服务
    Constants.GROUP_KEY, Constants.ANY_VALUE, //所有分组
    Constants.VERSION_KEY, Constants.ANY_VALUE, //所有版本
    Constants.CLASSIFIER_KEY, Constants.ANY_VALUE, //所有分类
```

```
Constants.CATEGORY_KEY, Constants.PROVIDERS_CATEGORY +  
", " + Constants.CONSUMERS_CATEGORY, //服务的提供者和服  
务的消费者
```

```
Constants.CHECK_KEY, String.valueOf(false)); //不检查
```

3. 调注册中心服务registry.subscribe(subscribeUrl, listener)订阅所有数据，NotifyListener在监控中心暴露为回调服务，由注册中心回调回调接口NotifyListener实现的功能主要是按服务提供者和服务的消费者分类，收集服务名称，服务的url，服务提供方或者消费方的系统相关信息。同时提供了一系列方法供注册中心调用查询。

#### 四：JettyContainer

监控中心将采集到的信息通过内置jetty来展现给用户，这里为了不依赖与jsp, velocity, freemarker等一些编写web应用的技术，采用在servlet中将html, css, js打印出来

JettyContainer的start方法启动了内置的jettyweb容器

将监控中心访问的本地文件目录设置到ResourceFilter中，并设置这个filter的访问映射到jetty中，ResourceFilter主要是读取本地保存的JFreeChart绘制的图片到浏览器中去。

将监控中心的前置控制器PageServlet，以及这个servlet的访问映射配置到jetty中。之所以叫PageServlet为前置控制器，就像其他的mvc框架一样用来分发具体的业务类

PageServlet的init初始化方法在web容器启动的时候加载所有的页面处理器PageHandler，用来根据不同的请求生成不同的页面，前面说过这里页面html都是通过java代码打印出来的。

PageServlet的init方法加载所有PageHandler时会判断PageHandler上是否有@Menu注解，将有注解的PageHandler加入集合，以被HomePageHandler用来生成主页以及各个页面的uri

PageServlet的doGet, doPost接收浏览器请求，请求以xx.html形式，xx就是PageHandler扩展的key，找到对应的PageHandler绘制对应的页面返回给浏览器。

```
@Menu(name = "Home", desc = "Home page.", order = Integer.MIN_VALUE)
```

```
//有注解 name 跟desc属性都是在页面中展示给用户看的
```

```
public class HomePageHandler implements PageHandler {  
    public Page handle(URL url) {  
        List<List<String>> rows = new ArrayList<List<String>>();  
        for (PageHandler handler : PageServlet.getInstance().getMenus())  
        {
```

```
            String uri =
```

```
ExtensionLoader.getExtensionLoader(PageHandler.class).getExtensionName  
e(handler); //这个uri其实就是PageHandler扩展配置的key， 页面中用它来  
请求选择具体的handler绘制 //出具体的page
```



```

        Menu menu = handler.getClass().getAnnotation(Menu.class);
        List<String> row = new ArrayList<String>();
        row.add("<a href=\"" + uri + ".html\">" + menu.name() +
"</a>");
        row.add(menu.desc());
        rows.add(row);
    }
    return new Page("Home", "Menus", new String[] {"Menu Name",
"Menu Desc"}, rows); //一个Page实体就是一个页面，这里包含所有主要
HomePage的页面内容
    }
}

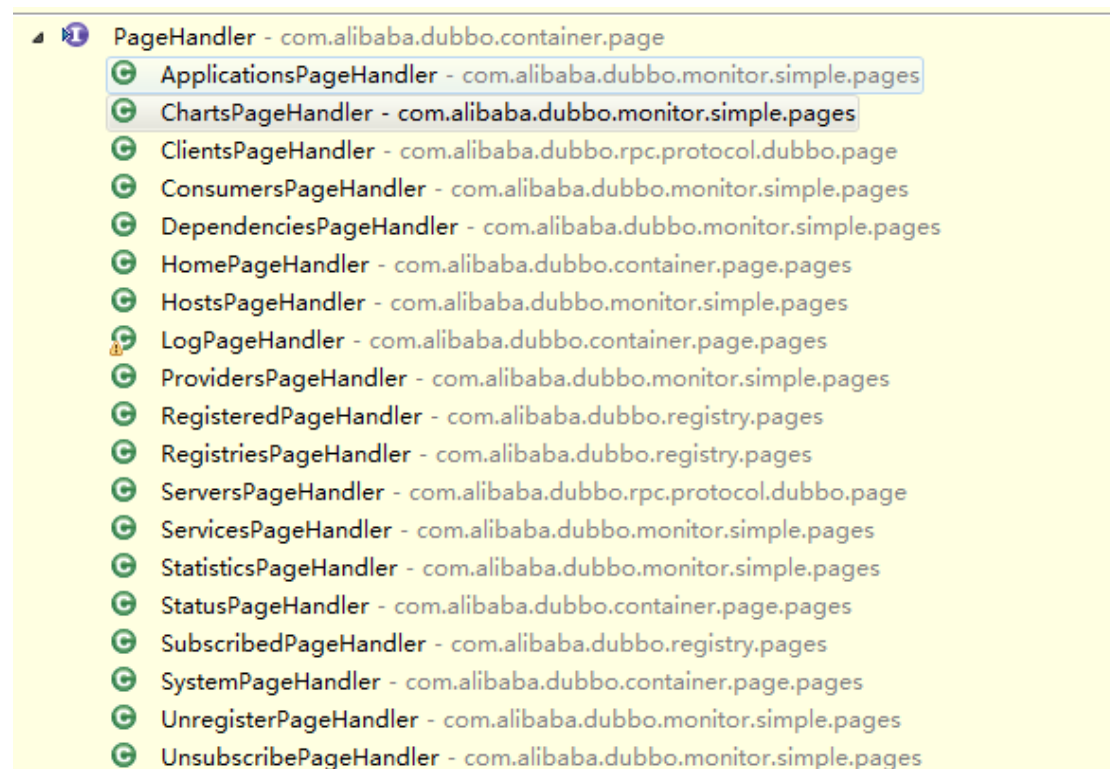
```

PageHandler的在com.alibaba.dubbo.container.page.PageHandler文件中的扩展配置

```

index=com.alibaba.dubbo.container.page.pages.HomePageHandler
providers=com.alibaba.dubbo.monitor.simple.pages.ProvidersPageHandler
consumers=com.alibaba.dubbo.monitor.simple.pages.ConsumersPageHandler
。。。。
```

下面截图看下dubbo大概提供了哪些扩展



下面截几张图看看监控中心的页面。



## 第十五章： 管理控制台

Dubbo 的 dubbo-admin 模块是后台管理系统，它的 MVC 框架式基于 webx3 的，webx 是阿里巴巴开源出来基于页面驱动的 mvc 框架，webx 在阿里内部广泛使用。Webx 是基于 turbine 发展而来逐渐成为一个功能强大扩展性强的 mvc 框架，利用约定大于配置原则，虽说使用简单，但是也有很多潜在规则在里面。Webx 作为除了阿里以外一个小众框架学习起来还是很有成本的，再说 dubbo 中又对 webx 进行一层 restful 改造，使大家即时按官方文档 <http://www.openwebx.org/> 学习也很难很快看懂。这里不做 webx 的相关介绍，因为一时半会没法讲，主要讲下 dubbo-admin 基本流程，方便有的朋友对 webx 不爽想迁移到 springmvc 或者 struts 之上。

### dubbo-admin web 应用配置

属性文件路径 `dubbo\dubbo-admin\src\main\webapp\WEB-INF\dubbo.properties`

`dubbo.registry.address=zookeeper://127.0.0.1:2181`

`dubbo.admin.root.password=root`

`dubbo.admin.guest.password=guest`

这里需要指定注册中心地址，因为对于服务的治理需要从注册中心获取相关信息，如提供者、消费者、路由信息、权重等等

这里后台权限验证也简单，用户也可以修改针对数据库，或者登陆中心的权限验证。

### Spring 的 bean 配置路径：

`dubbo\dubbo-admin\src\main\resources\META-INF\spring\dubbo-admin.xml`

下面我们来几个比较关键的配置

```
<dubbo:registry address="${dubbo.registry.address}"
check="false" file="false" />
<dubbo:reference id="registryService"
    interface="com.alibaba.dubbo.registry.RegistryService" check="false" />
<bean id="userService"
    class="com.alibaba.dubbo.governance.service.impl.UserServiceImpl">
    <property name="rootPassword"
value="${dubbo.admin.root.password}" />
    <property name="guestPassword"
value="${dubbo.admin.guest.password}" />
</bean>
```

```
<bean id="governanceCache"
      class="com.alibaba.dubbo.governance.sync.RegistryServerSync" />
```

- 1) <Dubbo:registry/>用来指定注册中心的地址，这个由 properties 文件中提供
- 2) 引用 registryService 服务， 这个在 zookeeper 为注册中心其实并没有引用什么远程的 registryService 服务，而是创建了代理通过 ZookeeperRegistry 来操作 zookeeper 的节点数据
- 3) RegistryServerSync 这个 bean 用来订阅同步注册中心数据

RegistryServerSync 实现了 InitializingBean 接口，这个接口是 spring 提供的一个回调在 spring 初始化 bean 的时当 bean 的参数都被设置的时候调用，这个方法实现为：registryService.subscribe(SUBSCRIBE, this); 向注册中心订阅

```
URL SUBSCRIBE = new URL(Constants.ADMIN_PROTOCOL, NetUtils.getLocalHost(), 0, "",
    Constants.INTERFACE_KEY, Constants.ANY_VALUE,
    Constants.GROUP_KEY, Constants.ANY_VALUE,
    Constants.VERSION_KEY, Constants.ANY_VALUE,
    Constants.CLASSIFIER_KEY, Constants.ANY_VALUE,
    Constants.CATEGORY_KEY, Constants.PROVIDERS_CATEGORY + ","
        + Constants.CONSUMERS_CATEGORY + ","
        + Constants.ROUTERS_CATEGORY + ","
        + Constants.CONFIGURATORS_CATEGORY,
    Constants.ENABLED_KEY, Constants.ANY_VALUE,
    Constants.CHECK_KEY, String.valueOf(false));
```

如图 url 订阅了所有的信息

同时 RegistryServerSync 实现了 NotifyListener 接口，这个接口用来当注册中心数据发生变化后回调订阅用户更新信息，是注册中心反向推送的实现。

RegistryServerSync 的 notify(urls)实现主要是分类缓存注册中心信息，供页面是使用

- 4) OverrideServiceImpl 实现 覆盖注册中心 url，主要流程如下  

```
registryService.unregister(oldOverride);
registryService.register(newOverride);
```

 当动态配置，负载均衡，权重配置等会调用此接口更新注册中心 url
- 5) RouteServiceImpl 用来新增，修改，删除路由信息  
 更新路由的流程如下，跟上面其实一样，也是覆盖注册中心路由信息  

```
registryService.unregister(oldRoute);
registryService.register(route.toUrl());
```

启动后台服务

Dubbo 内嵌 jetty 作为后台服务的 web 容器，测试的利用 mvn 命令很方便启动 debug

到 dubbo\dubbo\dubbo-admin 目录下执行 mvn jetty:run 或者 mvn debug jetty:run

默认访问端口是 8080

<http://localhost:8080/index.htm>

