# 一、dubbo的SPI

SPI是一种服务发现机制，JDK自带有服务发现机制，作用是自动加载位于指定文件夹下的配置文件，并加载其中所配置的类，例如：自动加载classpath下的数据库驱动类等。

Dubbo也有一套自身的服务发现机制，相较于jdk的SPI，dubboSPI支持懒加载，AOP、IOC，以及对异常的正确报出。

dubbo的SPI依赖于ExtensionLoader类，这个类同样是通过指定文件夹下所配置的文件扩展加载组件。

## 1.扩展流程



## 2.相关注解

dubbo通过三个注解@SPI、@Adaptive、@Activate对接口的实现类进行动态扩展。

@SPI(String value):

标注在接口上，代表这个接口的实现类可被扩展，value是该扩展接口默认实现类缩写。

@Adaptive(String[] value):

若标注在一个实现类上(不能标注在接口上)，表明这个类是一个自适应的装饰类，在调用getAdaptiveExtension()时不会动态生成编译装饰类。

若标注在接口的方法上，表明该方法需要被动态生成，在调用getAdaptiveExtension()时会动态生成并编译一个装饰类(类名以“$Adaptive结尾”)，没有标记@Adaptive注解的方法默认抛出异常。

其中的value为动态生成类需要过滤url的字段，会将@SPI中value和@Adaptive中的value对url先后进行过滤。

@Activate

@Adaptive是获取一个适配类，而@Activate则可以获取一批适配类，在impl上加上@Adaptive注解在调用Extension. getActivateExtension(URL url, String key)可以获得url满足条件的实现类。

## 3.ExtensionLoader相关api

ExtensionLoader是dubboSPI的核心类，相关API会对标注@SPI接口的类进行动态扩展

### 1)getExtensionLoader(Interface) 静态方法



### 2)ExtensionLoader的<init>

<init>中会保存interface.class，并且初始化内部的objectFactory字段，调用的是

ExtensionLoader.getExtensionLoader(ExtensionFactory.class).getAdaptiveExtension()【1】方法来初始化objectFactory

AdaptiveExtensionFactory被@Adaptive标记并且满足SPI扩展，所以【1】方法获得的其实是AdaptiveExtensionFactory实例。

### 3)AdaptiveExtensionFactory的作用

AdaptiveExtensionFactory是装饰类(同样也是单例类)，其内部组合了SpringExtensionFactory和SpiExtensionFactory(分别提供了Spring/SPI的IOC)

### 4)getExtension(key) 实例方法

通过getExtensionLoader(Interface)获得ExtensionLoader的实例之后，可以调用getExtension(key)获取SPI配置的key对应interface的实现类。



【1】中主要分成四类：

1）被@Adaptive标记的类

2）含有一个参数为interface的构造函数的实现类（wrapper类）

3）被@Activate标记的类

4）其他类与3）中的类

### 5)getAdaptiveExtension() 实例方法

如果调用该方法的扩展点接口，没有实现类被@Adaptive标记，则会为其自动生成一个装饰类，装饰类会对url进行筛选出符合条件的属性，完成getExtension()的调用。



# 二、容器启动

## 1.dubbo的容器

Main.main(args)会自动启动配置在dubbo.properties中的容器，默认为spring容器。spring容器启动时依据spring的扩展机制，会对dubbo相关标签进行加载。

## 2.dubbo内嵌于spring

spring的扩展加载beanDefinition机制：在classpath:META-INF下放文件spring.handlers、spring.schemas、dubbo.xsd三个文件

spring.handlers是dubbo解析<dubbo/……>标签的parse类，以key-value形式存在，key是名称空间，value是对应类的权限定名。

dubbo.xsd是dubbo在xml文件中的配置规则。

spring.schemas记录spring扩展的配置规则的名字



【1】每个标签都对应了一个Config类(例如<dubbo:service>最终会被解析成ServiceBean)

# 三、服务发布

## 1.dubbo服务启动的时机

dubbo是在spring容器初始化之后被激活的，因为ServiceBean实现了ApplicationListener接口(监听者模式)，具体流程如下所示



ServiceBean在onApplicationEvent()方法执行了export()方法对服务进行暴露

## 2.暴露流程



【1】中暴露，会将每个Service在每个Registry(注册中心)暴露，多对多关系。

### (1)本地暴露

若本地引用时发起网络连接，则毫无意义且浪费资源，所以dubbo首先会进行本地暴露避免本地调用时的网络连接。

1)本地暴露通过ProxyFactory创建一个Wrapper代理类，代理ref，并用AbstractProxyInvoker包装，用来为每个ref的方法做方法名与参数的校验。AbstractProxyInvoker中保存了“//injvm:/……”的URL、接口名。

2)调用InJvmProtocol.export(AbstractProxyInvoker)进行本地暴露得到InjvmExporter，不会打开端口。同时将暴露的InjvmExporter在InjvmProtocol的exporterMap缓存。

### (2)远程暴露

远程暴露时，也是生成invoker后进行暴露，生成invoker的流程与本地暴露类似，只是传入的url为RegistryUrl(包括需要注册的信息(dubbo:/ip:port……))

再对上述生成的invoker进行远程暴露(通过RegistryProtocol.export()封装远程暴露)



【1】注册时会在zookeeper上创建持久节点/dubbo/interfaceName/provider，在该持久节点下面创建临时节点记录provider相关信息：协议名:/ip:port/interfaceName……等

【2】注册监听/dubbo/interfaceName/configurators，动态配置中心可以从此配置provider相关信息。当dubbo的监听器监听到该节点变化之后，会调用notify()方法对exporter在运行时进行动态改变

# 四、引用服务

dubbo同样生成了一个invoker作为consumer的引用，但由于引用服务时consumer端没有服务(Service)的实现类(impl)，所以依然使用动态代理将invoker封装成impl

## 1.服务引用时机(FactoryBean)

spring容器启动后内部会有一个ReferenceBean对象，ReferenceBean实现了spring中的FactoryBean接口，在调用applicationContext.getBean()方法时会调用FactoryBean.getObject()。

## 未命名文件 (3)2.引用流程

①获得与zookeeper的连接client，保存于RegistryDirectory对象中，并利用RegistryDirectory完成对zookeeper进行注册

②RegistryDirectory(实现了NotifyLstener)对zookeeper进行订阅，内部使用ChildListener进行订阅，当ChildListener检测到改变，会激活RegistryDirectory的Notify方法。

③RegistryDirectory的Notify方法会对多个invoker(这里的invoker对应的是服务)进行创建或改变。第一次订阅的时候会拉取providers全量信息，并调用notify()方法创建invoker对象，invoker对象中维护了与provider的TCP长连接Exchanger ( 默认懒加载，在第一次调用时生成 ) 。

总之：RegistryDirectory管理了invoker，并在运行期对invoker进行动态更改。

④生成一个外部的invoker(一个Service一个invoker)，该invoker是MockClusterWrapper，内部包了一个FailbackClusterInvoker(默认)，FailbackClusterInvoker内部包了RegistryDirectory。

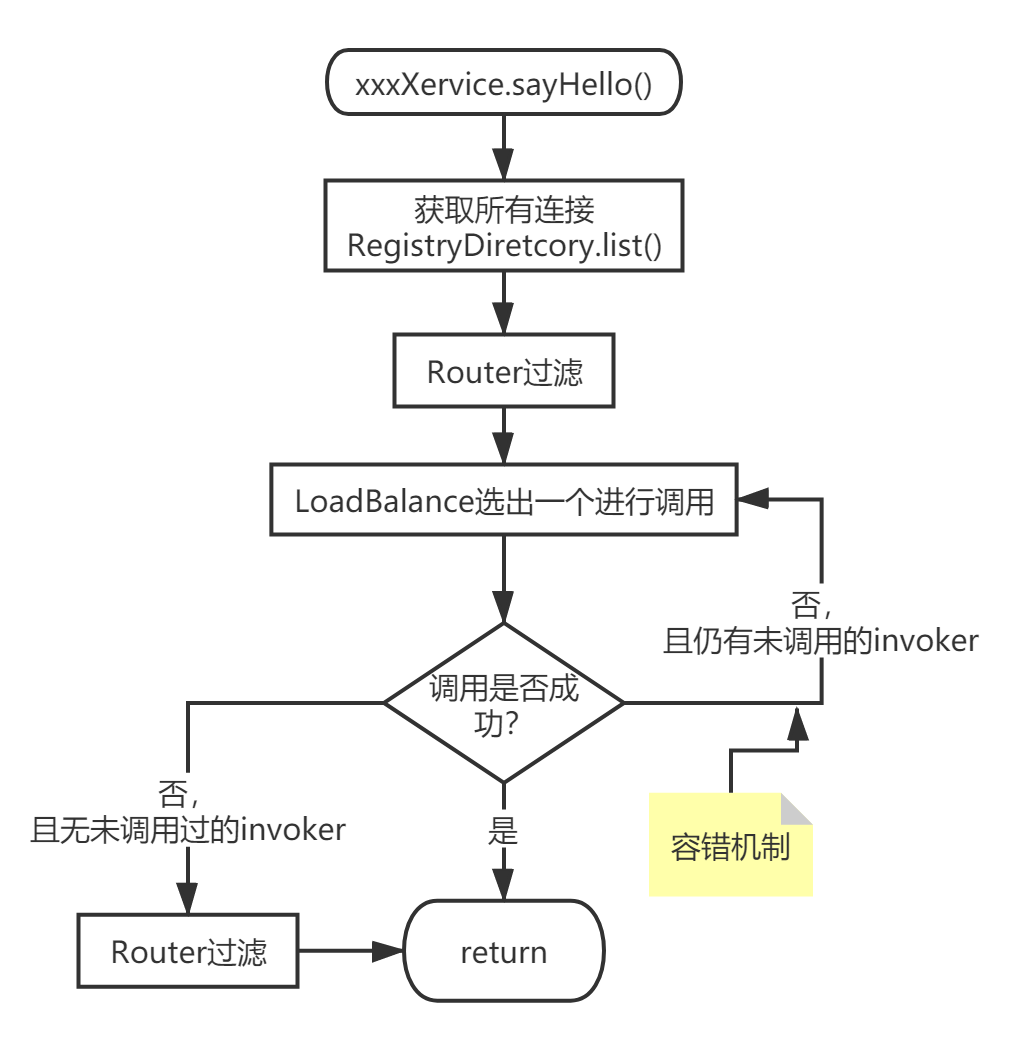
结论：利用FailbackClusterInvoker将RegistryDirectory中的多个invoker伪装成一个invoker

⑤对MockClusterWrapper生成代理类(访问失败时返回mock值)

# 五、集群容错

集群容错皆在consumer端执行，包括路由选择(router)、负载均衡(loadBalance)、容错机制(cluster)

执行大致流程如下 (以下的容错机制使用默认的failover )



## 1.容错机制cluster

每个cluster可以伪装一个clusterInvoker给客户端调用，clusterInvoker中放了RegistryDirectory。

failover：失败重新loadbalance一个进行调用，可以设置retries次数选择重试的次数

failfast：若失败，直接返回异常，不重试

failsafe：若失败，返回null

failback：若失败，返回空值，但是后台会调用一个线程池对错误重新定时调用

forking：同时调用多个服务一起运行，返回第一个结果，可以设置forks最大并行数

loadbalance选择出forks个provider放入线程池运行，默认同步调用，先运行完的线程将数据放入一个BlockingQueue，主线程调用blockingQueue.poll() timeout秒，返回先调用完的数据给用户(包括异常)。

broadcast：广播调用所有可用的服务，若出现一个失败，则报错。不需要负载均衡

## 2.LoadBalance

负载均衡其实并不是直接调用LoadBalance.select()进行选择，而是通过AbstractClusterInvoker.select()引用到LoadBalance.select() (这里的AbstractClusterInvoker是集群容错类，用来伪装多个invoker的)

AbstractClusterInvoker.select()中首先会根据url中传的参数判断是否开启了可用性检测以及粘滞连接

Dubbo的负载均衡有四种：

Random：按权重随机select一个provider，容易重复调用同一个节点

LeastActive：最小活跃数调用

一致性Hash：相同参数的调用总是发给同一台机器

RoundRobin：有普通轮询和滑动轮询，dubbo中采用的是滑动轮询算法。

介绍一下滑动轮询算法：

例如：三台机器A、B、C的权重分别是1:2:3，总权重为6

第一次：0:0:0 每个节点加上自身权重1:2:3 max=3 调用C 调用者减6变成1:2:-3

第二次：1:2:-3 每个节点加上自身权重2:4:0 max=4 调用B 调用者减6变成2:-2:0

第三次：2:-2:0 每个节点加上自身权重3:0:3 max=3 调用A 调用者减6变成-3:0:3

第四次：-3:0:3 每个节点加上自身权重-2:2:6 max=3 调用C 调用者减6变成-2:2:0

第五次：-2:2:0 每个节点加上自身权重-1:4:3 max=4 调用B 调用者减6变成-1:-2:3

第六次：-1:-2:3 每个节点加上自身权重0:0:6 max=3 调用C 调用者减6变成0:0:0

..........

以上可以看到，滑动轮询避免了普通轮询的有概率对同一节点反复调用的问题。

# 六、过滤器Filter

## 1.包装逻辑

Filter在dubbo中其实是通过SPI机制进行自动包装的，易于扩展。

①过滤器构造时机：进行暴露/引用服务时，对invoker进行包装

②在调用Protocol$Adaptive(dubbo为Protocol动态生成的包装类，见第一章)时会为XXXProtocol包装两个Wrapper：ProtocolFilterWrapper、ProtocolListenerWrapper；前者就是构造过滤器的时机，将invoker包装成具有定制化功能的调用。

构造的逻辑在

buildInvokerChain(invoker, Constants.SERVICE\_FILTER\_KEY, provider/consumer)中

## 2.@Activate注解

该注解应用于Filter的子类上，@Activate中有几个属性：group用来标记该Filter的作用范围，provider/consumer；其他属性皆用于对该Filter的排序。

对于provider的Filter有：EchoFilter、ContextFilter、GenericFilter、ExecuteLimitFilter、ActiveLimitFilter、TraceFilter、AccessLogFilter......

对于consumer的Filter有：GenericImplFilter、ConsumerContextFilter、DeprecatedFilter、TokenFilter、CacheFilter......

## 3.流程



