provider启动时将所有接口注册到zoo中，并订阅configurators

consumer启动时订阅providers、configurators、routes

consumer、provider启动后，后台会启动定时器，发送统计数据给monitor

# 一、dubbo的spi

## 1. getExtensionLoader

ExtensionLoader实例包含一个type和AdaptiveExtensionFactory【SpiExtensionFactory、SpringExtensionFactory】。

getExtensionLoader(interface<T> type) 为这个参数new一个EntensionLoader，然后缓存起来到EntensionLoader的静态常量中（以供复用）

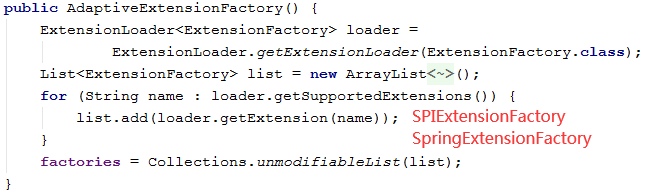
每个EntensionLoader实例在构造方法中会调用：

ExtensionLoader.getExtensionLoader(ExtensionFactory.class).getAdaptiveExtension();

给字段objectFactory赋值（类型是ExtensionFactory，提供dubbo的IOC功能）

(getExtensionLoader(ExtensionFactory.class)自身的objectFactory是null)

ExtensionFactory和Compiler是dubbo唯二没有使用@Adaptive标注方法的类，所以有对应被@Adaptive修饰类名的AdaptiveExtensionFactory来被调用getAdaptiveExtension



**上述代码：** adaptiveExtensionFactory也是通过SPI机制获取其子类，并内部维护两个子类SPIExtensionFactory和SpringExtensionFactory（一个通过SPI机制dubboIOC，另一个是SpringIOC）。

Adaptive……的作用其实就是对两个子类进行包装（装饰者模式）

## 2.getAdaptiveExtension

getAdaptiveExtension() 获取一个扩展的装饰者对象，这个方法有一个规则，如果没有@Adaptive注解修饰该接口的子类（整个Dubbo中只有ExtensionFactory与Compiler有@Adaptive修饰的子类），就会动态创建并编译一个装饰类（仅逻辑包装），例如Protocal$Adaptive对象

**例如：**所有对象在getExtensionLoader()时都会进行初始化里面的objectFactory对象，objectFactory也是用spi但是是使用getAdaptiveExtension()获得用@Adaptive标记的AdaptiveExtensionFactory，AdaptiveExtensionFactory中维护了一个SPIExtensionFactory和一个SpringExtensionFactory。

getAdaptiveExtension() （@Adaptive在一个类上这个类是一个装饰类，在方法上就是代理）

例如：AdaptiveExtensionFactory类上标记了@Adaptive

Protocol的方法上标记了@Adaptive

@Adaptive注解在类和方法上的区别：

注解在类上：代表人工实现编码，即实现了一个装饰类(即设计模式中的装饰者模式)，例如ExtensionFactory

注解在方法上：调用getAdaptiveExtension时自动生成和编译一个动态的adaptive类，例如Protocal$Adaptive

### （2）源码分析

ServiceConfig中会初始化一个Protocal成员

private static final Protocol protocol = ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension();

getAdaptiveExtension()

-createAdaptiveExtension()

--getAdaptiveExtensionClass()

getExtensionClasses()

loadExtensionClasses()

loadDirectory() : 对指定文件夹(dubbo/internal)进行加载（SPI）

loadResource()

loadClass()：【1】对SPI加载的所有类进行逐个缓存（见下面）

--createAdaptiveExtensionClass()：【2】自动生成和编译一个类\*\*\*，仅仅是个代理类

-injectExtension(createAdaptiveExtension().newInstance()):【3】IOC，实现依赖注入

在getAdaptiveExtension()方法的loadExtensionClasses方法会去classpath:dubbo/interval文件夹下寻找带有接口的类，并且SPI加载

【1】在这个SPI加载过程中loadClass，会对SPI文件中所有的类进行遍历，如果满足条件会进行缓存

①cachedAdaptiveClass： 如果遍历的类被@Adaptive修饰，例如AdaptiveExtensionFactory，则保存至该缓存中

②cachedWrapperClasses：如果这个类没有@Adaptive注解修饰，且构造函数的参数类型有这个接口(protocol)，则保存到该缓存中，例如Protocol里面的SPI只有ProtocolFilterWrapper和ProtocolListenerWrapper以及QosProtocolWrapper可以命中

③cachedActivates：剩下的类包含@Activate注解的(例如过滤器Filter)。

④cachedNames(cachedClasses)：如果该类都不满足以上三个缓存

【2】\*\*\*处动态生成的代码如下



动态生成的类会对类中标注@Adaptive方法进行动态生成（包装，装饰者模式），没有标注@Adaptive的方法生成的代码中直接抛出异常

getExtension(String name)

在动态生成的Protocal$Adaptive有

ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(extName)

## 3.getExtension

getExtension(String name) 获取接口指定name子类的包装类（如果SPI配置文件有包装类，那么getExtension出来的对象是 包装类包着包装类…包着name的类）

getExtension(String name):

cachedInstances.get(name)：先从缓存中拿，如果有的话直接返回（没有走下一行）

createExtension(name)

--getExtensionClasses()：把SPI中配置文件全部读出来并分类缓存（2【1】分析）

--injectExtension(T)：这里用来依赖注入（仅支持setter反射依赖注入）

objectFactory.getExtension(pt, property)：为dubbo的IOC提供所有对象

--SPIExtensionFactory. getExtension(type, name)：先通过SPI注入

ExtensionLoader.getExtensionLoader(type)

return loader.getAdaptiveExtension()

--SpringExtensionFactory. getExtension(type, name)：再通过Spring注入

return context.getBean(name);

--injectExtension((T) wrapperClass.getConstructor(type).newInstance(instance));AOP

这边的wrapperClass就是上面在SPI时loadClass筛选出来的包装类

回顾：wrapperClass是不被@Adaptive修饰且构造函数是以接口为参数，

(参见ProtocolFilterWrapper等)

在这个阶段，原来的实现类被包装成了包装类（装饰者模式）

filter…和listener…

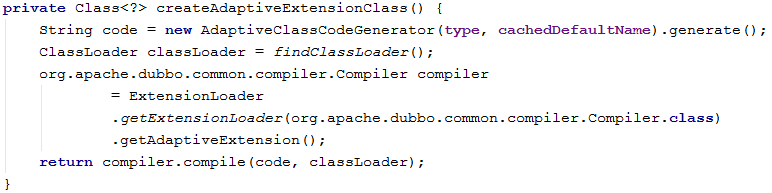
# 二、动态编译

getAdaptiveExtension()

createAdaptiveExtension()

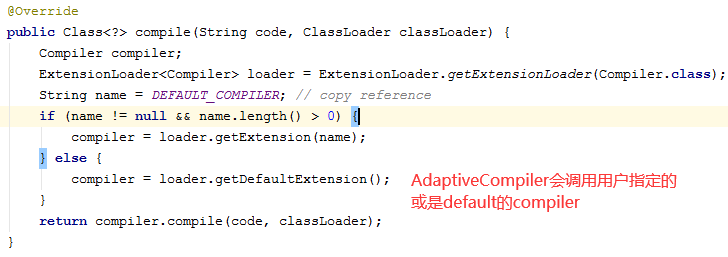
--createAdaptiveExtensionClass()

这一步会自动编译一个类



这里Compiler的获得Adaptive不是生成代理类，而是获得装饰/包装类（@Adaptive修饰的类）。





如果没有指定名字，那么会去取默认的编译器（getDefaultExtension是找到接口中的注释@SPI(value)中value指定的），Compiler上的注释@SPI(“javassist”)

# 三、Dubbo嵌入Spring

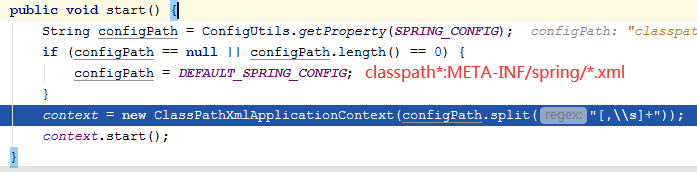
## 1.dubbo启动spring容器

com.alibaba.dubbo.container.Main.main(args); //会读取classpath:dubbo.properties文件

### (1)main方法通过SPI获取容器实例

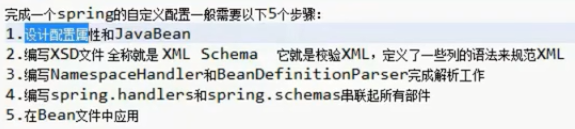


### (2)container.start()方法



默认读取classpath\*:META-INF/spring/\*.xml所有配置并启动

## 2.spring解析dubbo标签



### dubbo中解析标签的类与文件





# 四、服务发布

## 1.服务发布流程（日志）

### (1)暴露本地服务

Export dubbo service com.alibaba.dubbo.demo.DemoService to local registry, dubbo version: , current host: 192.168.142.1

### (2)暴露远程服务+注册到zookeeper

Export dubbo service com.alibaba.dubbo.demo.DemoService to url dubbo://192.168.142.1:20880/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=demo-provider&bean.name=com.alibaba.dubbo.demo.DemoService&bind.ip=192.168.142.1&bind.port=20880&dubbo=2.0.2&generic=false&interface=com.alibaba.dubbo.demo.DemoService&methods=sayHello&pid=9832&side=provider&timestamp=1596558633329, dubbo version: , current host: 192.168.142.1

Register dubbo service com.alibaba.dubbo.demo.DemoService url dubbo://192.168.142.1:20880/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=demo-provider&bean.name=com.alibaba.dubbo.demo.DemoService&bind.ip=192.168.142.1&bind.port=20880&dubbo=2.0.2&generic=false&interface=com.alibaba.dubbo.demo.DemoService&methods=sayHello&pid=9832&side=provider&timestamp=1596558633329 to registry registry://127.0.0.1:2181/com.alibaba.dubbo.registry.RegistryService?application=demo-provider&dubbo=2.0.2&pid=9832&registry=zookeeper&timestamp=1596558633304, dubbo version: , current host: 192.168.142.1

### (3)启动netty

Start NettyServer bind /0.0.0.0:20880, export /192.168.142.1:20880, dubbo version: , current host: 192.168.142.1

### (4)打开连接zk

INFO zookeeper.ClientCnxn: Opening socket connection to server 127.0.0.1/127.0.0.1:2181

### (5)注册到zk

[DUBBO] Register: dubbo://192.168.142.1:20880/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=demo-provider&bean.name=com.alibaba.dubbo.demo.DemoService&dubbo=2.0.2&generic=false&interface=com.alibaba.dubbo.demo.DemoService&methods=sayHello&pid=9832&side=provider&timestamp=1596558633329, dubbo version: , current host: 192.168.142.1

### (6)订阅(监听)zk

[DUBBO] Subscribe: provider://192.168.142.1:20880/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=demo-provider&bean.name=com.alibaba.dubbo.demo.DemoService&category=configurators&check=false&dubbo=2.0.2&generic=false&interface=com.alibaba.dubbo.demo.DemoService&methods=sayHello&pid=9832&side=provider&timestamp=1596558633329, dubbo version: , current host: 192.168.142.1

## 2.源码分析

ServiceBean. onApplicationEvent()：Spring容器初始化完成发布事件给监听器ServiceBean

ServiceConfig.export() ServiceConfig是ServiceBean的父类，这一阶段可完成定时启动

doExport() 进行相参数关检查

doExportUrls() 内for循环表示一个服务可以有多个协议(http/dubbo，默认是dubbo)

loadRegistries(true) 从dubbo.properties中组装registry的URL信息

doExportUrlsFor1Protocol(protocolConfig, registryURLs)每个协议注册暴露

exportLocal(URL url) 本地暴露（见3（1））

## 3.暴露本地服务

### (1)暴露本地服务

暴露在一个jvm中，不用调用zk进行远程通信，例如在在一个服务内，自己调用自己的接口，就没有必要建立网络IP连接进行通信

exportLocal(URL url)

#### ①得到invoker

ProxyFactory$Adaptive.getInvoker(ref, (Class) interfaceClass, local) 这里是动态生成的类

-- extension = ExtensionLoader

.getExtensionLoader(com.alibaba.dubbo.rpc.ProxyFactory.class)

.getExtension(extName);

调用SPI容器中的代理类，默认是javassist的包装类(Wrapper)

-- extension. getInvoker(T proxy, Class<T> type, URL url) （Extension是包装类）

==StubProxyFactoryWrapper. getInvoker(T proxy, Class<T> type, URL url)

这个包装类内部调用了javassist

JavassistProxyFactory.getInvoker(proxy, type, url)

Wrapper wrapper = Wrapper.getWrapper(DemoServiceImpl.class)

makeWrapper(Class<?> c)

return new AbstractProxyInvoker<T>(proxy, type, url)

//这个AbstractProxyInvoker实际上就是wrapper的代理类，而wrapper是demoServiceImpl的代理， ProxyFactory$Adaptive.getInvoker实际上返回的就是demoServiceImpl的代理

#### ②暴露invoker

Protocol$Adaptive.export(Invoker)

ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(extName)

//根据extName得到Protocol的包装类，包装类最内层默认为DubboProtocol

ProtocolFilterWrapper.export(invoker)

ProtocolListenerWrapper.export(invoker)

// ProtocolFilterWrapper会在这一层对invoker包装Filters责任链

InJvmProtocol.export(invoker)

return new InJvmExporter(invoker,)

exporterMap.put(key, this)

key=”DemoService”, value=InjvmExporter

//将被层层包装的invoker放在一个Exporter中并返回，并最终放到一个Exporter的Map中。

### (2)ProxyFactory的作用

为了获得一个接口的代理类，例如获得一个远程接口的代理

ProxyFactory有两个方法，代表两个作用

getInvoker：针对server端，将服务对象，如DemoServiceImpl包装成一个invoker对象。

getProxy：针对client端，用来创建接口的代理对象，例如DemoService接口。

### (3)wrapper的作用

类似spring的BeanWrapper，它就是包装了一个接口/类，可以通过wrapper对实例对象进行赋值/取值/指定方法的调用

### (4)Invoker的作用

是一个可执行的对象，能根据方法的名称、参数得到相应的执行结果

有一个很重要的方法 Result invoke(Invocation invocation)

Invocation作用是包含了需要执行的方法/参数类型/参数值（远程序列化或是本地传过来的）

它有两个实现类RPCInvocation、MockInvocation

Invoker有三种类型：

①本地执行类的Invoker

②远程通信类的Invoker

③多个远程通信执行类的Invoker聚合成集群版的Invoker

## 4.暴露远程服务

### (1)暴露远程服务

暴露给远程客户端的IP和端口号，通过网络来实现通信

#### ①到RegistryProtocol的流程

proxyFactory.getInvoker(ref服务, 接口, zoo的URL)

//获得一个Invoker，与本地暴露类似，只是其中注册的URL不为本地了，而是远程zoo

Protocol$Adaptive .export(Invoker)

ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(“registry”);

//获得一个RegistryProtocol的包装类

ProtocolFilterWrapper.export(invoker) //这里不会创建Filter的责任链

ProtocolListenerWrapper. export(invoker)

RegistryProtocol. export(invoker) //真正的暴露逻辑

#### ②RegistryProtocol.export(invoker)

doLocalExport(originInvoker)

getCacheKey(originInvoker)//获取提供服务URL的字符串

// dubbo://192.168.142.1:20880/com.alibaba.dubbo.demo.DemoService……

Exporter<T> exporter=protocol.export(invokerDelegete)

==Protocol$Adaptive. export(invokerDelegete) //Url中是Dubbo协议

ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(extName)

//获得的是dubbo协议的包装类

ProtocolFilterWrapper.export() //ProtocolFilterWrapper包的是DubboProtocol

invoker = buildInvokerChain(invoker, XX, “Provider”) //8个Filters包装

ProtocolListenerWrapper.export(invoker);

DubboProtocol.export(invoker) //见③

#### ③DubboProtocol.export(invoker)

String key = serviceKey(url) //组装key:com.alibaba.dubbo.demo.DemoService:20880

DubboExporter<T> exporter = new DubboExporter<T>(invoker, key, exporterMap)

//创建一个exporter

exporterMap.put(key, exporter) //与本地暴露不同，本地暴露key只有接口名，远程还有port

openServer(url)

ExchangeServer server = createServer(url)

ExchangeServer server = Exchangers.bind(url, requestHandler)

ExchangeServer server = getExchanger(“header”); //默认是header

return ExtensionLoader.getExtensionLoader(Exchanger.class).getExtension(“header”);

server.bind(url, ExchangeHandler)

DecodeHandler handler2 = new DecodeHandler(new HeaderExchangeHandler(handler))

//new DecodeHandler/ HeaderExchangeHandler(handler)都是为了赋值handler

Server server = Transporters.bind(url, handler2)

Transporter$Adaptive = ExtensionLoader.getExtensionLoader(Transporter.class).getAdaptiveExtension()

NettyTransporter.bind(url, handler2)

new NettyServer(url, listener)

# 五、Filter责任链

ExportLocal中获得Invoker之后会使用Protocol$Adaptive对Invoker进行export

Protocol$Adaptive.export(Invoker)

ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(extName);

// extName可从Invoker的URL中获得，所以可以通过URL进行设定

//默认获得的是Protocol的包装类，//ProtocolFilterWrapper和ProtocolListenerWrapper包装的InjvmProtocol

ProtocolFilterWrapper.export(Filter责任链包装的Invoker) //见下

buildInvokerChain(invoker, Constants.SERVICE\_FILTER\_KEY, Constants.PROVIDER)

ExtensionLoader //通过SPI扩展获得所有被@Activate修饰的Filter

.getExtensionLoader(Filter.class) //@Activate注解中包含Filter的顺序与作用范围

.getActivateExtension(invoker.getUrl(), key, group)

for(Filter filter : Filters){

利用每个Filter对invoker进行包装

}

return 所有Filter包装的invoker