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.ExtensionLoader

(1)static final ConcurrentMap<Class<?>, ExtensionLoader<?>> EXTENSION\_LOADERS

缓存所有已加载类的ExtensionLoader

(2)static final ConcurrentMap<Class<?>, Object> EXTENSION\_INSTANCES

缓存所有类的

(3)Class<?> type

ExtensionLoader的类型，构造函数传入

(4)ExtensionFactory objectFactory

Dubbo的IOC，构造函数传入

(5)ConcurrentMap<Class<?>, String> cachedNames

所有未被@Adaptive、@Activate修饰、且不为包装类

(6)Holder<Map<String, Class<?>>> cachedClasses/ConcurrentMap<String, Holder<Object>> cachedInstances

(5)类的Class与对象/缓存

(7)Holder<Object> cachedAdaptiveInstance/Class<?> cachedAdaptiveClass

@Adaptive类与对象的缓存（SPI中只能用有一个Adaptive类，否则会报错）

(8)Set<Class<?>> cachedWrapperClasses

未被@Adaptive修饰，且有一个构造函数的入参为SPI接口类（包装类AOP）

(9)Map<String, Activate> cachedActivates

所有被@Activate修饰的类

## 2.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）

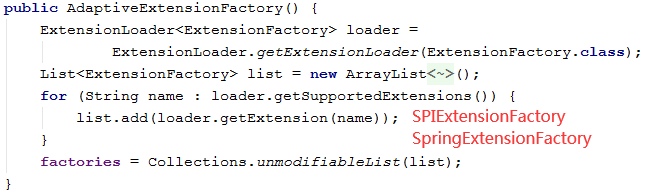
每个EntensionLoader实例在构造方法中会调用：

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）

## 3.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，会对文件中所有的类进行遍历，如果满足条件会进行缓存

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

cachedWrapperClasses：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

cachedActivates：剩下的类包含@Activate注解的。

cachedNames(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），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)

## 4.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出来的对象就是一个包装类）

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就在这个阶段被加入的

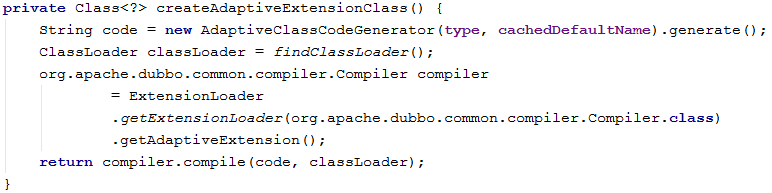
# 二、动态编译

getAdaptiveExtension()

createAdaptiveExtension()

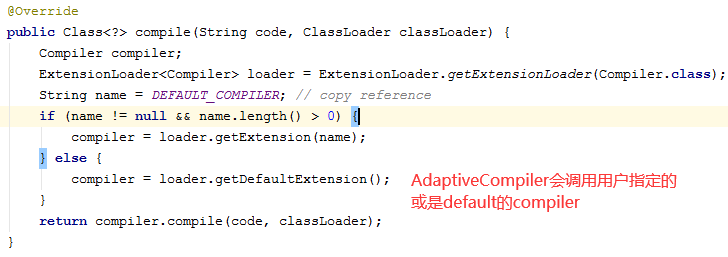
--createAdaptiveExtensionClass()

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

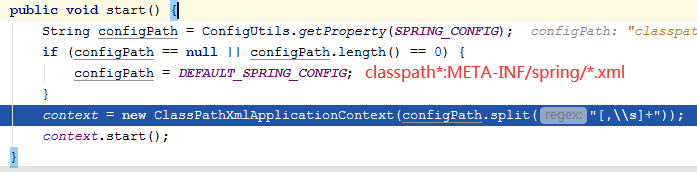
## 1.dubbo启动spring容器

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