Dubbo扩展源泉之SPI机制深度讲解

1、什么是SPI

1.1、SPI简介

SPI，全称为 Service Provider Interface，是一种服务发现机制。它通过在ClassPath路径下的META-INF/services文件夹查找文件，自动加载文件里所定义的类。 这一机制为很多框架扩展提供了可能，比如在Dubbo、JDBC中都使用到了SPI机制。我们先通过一个很简单的例子来看下它是怎么用的。

简单来说，SPI是一种扩展机制，核心就是将服务配置化，在核心代码不用改动的前提下，通过加载配置文件中的服务，然后根据传递的参数来决定到底走什么逻辑，走哪个服务的逻辑。这样就对扩展是开放的，对修改是关闭的。

1.2、SPI的运用场景

当你在写核心代码的时候，如果某个点有涉及到会根据参数的不同走不同的逻辑的时候，如果没有SPI，你可能会在代码里面写大量的if else代码，这样代码就非常不灵活，假设有一天又新增了一种逻辑，代码里面也要跟着改，这个就违背了开闭原则，SPI的出现就是解决这种扩展问题的，你可以把实现类全部都配置到配置文件中，然后在核心代码里面就只要加载配置文件，然后根据入参跟加载的这些类进行匹配，如果匹配的就走该逻辑，这样如果有一天新增了逻辑，核心代码是不用变的，唯一变的就是自己工程里面的配置文件和新增类，符合了开闭原则。

2、JDK中的SPI机制

前面已经介绍过SPI是什么以及在哪些地方用了，在这里就重点介绍一下SPI在JDK中的实现，下面我们看看具体的

2.1、案例

2.1.1、顶层接口

```
public interface GLog {
    boolean support(String type);
    void debug();
    void info();
}
```

2.1.2、接口实现

```
public class Log4j implements GLog {
    @Override
    public boolean support(String type) {
        return "log4j".equalsIgnoreCase(type);
    }
```

```
    @Override
    public void debug() {
        System.out.println("====log4j.debug======");
    }

    @Override
    public void info() {
        System.out.println("====log4j.info======");
    }
}
```

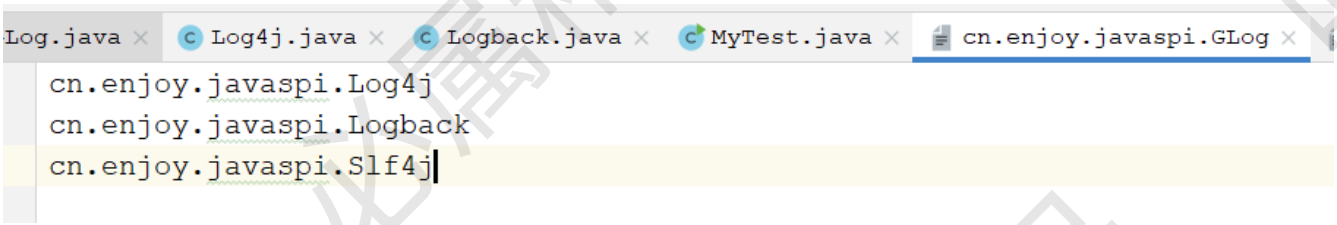
```
public class Logback implements GLog {
    @Override
    public boolean support(String type) {
        return "Logback".equalsIgnoreCase(type);
    }

    @Override
    public void debug() {
        System.out.println("====Logback.debug======");
    }

    @Override
    public void info() {
        System.out.println("====Logback.info======");
    }
}
```

```
public class Slf4j implements GLog {
    @Override
    public boolean support(String type) {
        return "Slf4j".equalsIgnoreCase(type);
    }

    @Override
    public void debug() {
        System.out.println("====Slf4j.debug======");
    }

    @Override
    public void info() {
        System.out.println("====Slf4j.info======");
    }
}
```

2.1.3、文件配置

在resources/META-INF/services目录创建文件，**文件名称必须跟接口的完整限定名相同。**如图：

这个接口文件中配置了该接口的所有实现类的完整限定名，如图：



现在要根据输入的参数来决定到底是走Log4j的逻辑还是Logback的逻辑。

2.1.4、单元测试

```
public class MyTest {

    //不同的入参，对应调用的逻辑是不一样的
    public static void main(String[] args) {
        //这个是我们业务的核心代码，核心代码会根据外部的参数决定要掉哪一个实例
        //可以去读配置文件 properties配置，去决定掉哪个实例
        //jdk api  加载配置文件配置实例
        ServiceLoader<GLog> all = ServiceLoader.load(GLog.class);
        Iterator<GLog> iterator = all.iterator();

        Scanner scanner = new Scanner(System.in);
        String s = scanner.nextLine();

        while (iterator.hasNext()) {
            GLog next = iterator.next();
            //这个实例是不是我们需要掉的
            // 策略模式  当前实例是不是跟入参匹配
            if(next.support(s)) {
                next.debug();
            }
        }
    }
}
```

2.2、总结

大家可以把这个单元测试代码看成是你自己业务中的核心代码，这个代码是不需要改动的，扩展新增的只是接口对应的实现类而已，然后在核心代码中我们只要根据传入的参数去调用不同的逻辑就可以了，这个就是SPI扩展的魅力，核心代码不需要改动。

从上面的测试代码我希望大家还了解一个点，JDK中的SPI也是获取类实例的一种方式，然后配合策略模式就可以根据参数选择实例调用了。

3、dubbo中的SPI机制

dubbo中的spi机制大体上的流程跟jdk中的spi类似的，但是也有很多细节方面不一样，下面我们就重点看看dubbo中的spi机制。

3.1、案例

3.1.1、顶层接口

```
//这个是必须的
@SPI("spring")
public interface ActivateApi {
    @Adaptive
    String todo(String param, URL url);
}
```

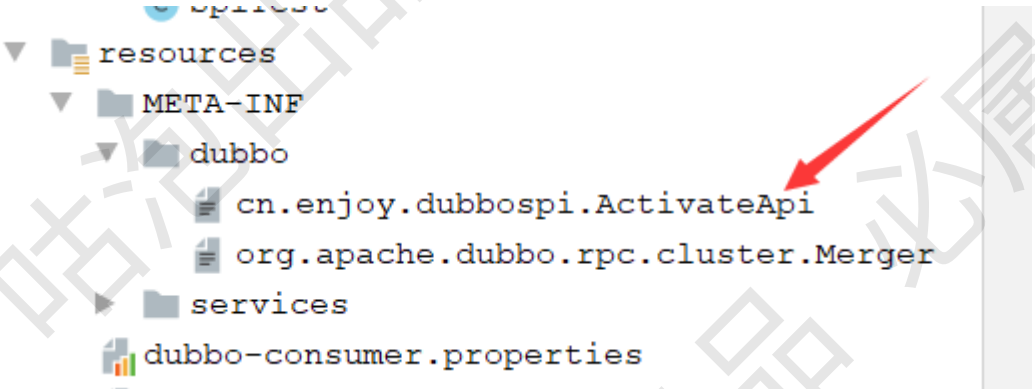
3.1.2、接口实现

```
@Adaptive
public class DubboActivate implements ActivateApi {
    @Override
    public String todo(String param, URL url) {
        return param;
    }
}

public class MybatisActivate implements ActivateApi {

    private ActivateApi  activateApi;

    @Override
    public String todo(String param, URL url) {
        return param;
    }

    public void setActivateApi(ActivateApi activateApi) {
        this.activateApi = activateApi;
        System.out.println(activateApi);
    }
}

public class Rabbitmq1Activate implements ActivateApi {
    @Override
    public String todo(String param, URL url) {
        return param;
    }
}

public class Rabbitmq2Activate implements ActivateApi {
```

```
    @Override
    public String todo(String param, URL url) {
        return param;
    }
}

public class RabbitmqActivate implements ActivateApi {
    @Override
    public String todo(String param, URL url) {
        return param;
    }
}

public class SpringActivate implements ActivateApi {
    @Override
    public String todo(String param, URL url) {
        return param;
    }
}

public class SpringCloudActivate implements ActivateApi {
    @Override
    public String todo(String param, URL url) {
        return param;
    }
}
```

3.1.3、文件配置

在resources/META-INF/dubbo下面配置接口文件，文件名称必须跟接口完整限定名相同，如图：



接口文件中配置的内容就是该接口的实现类的完整限定名，跟jdk中配置不一样的地方就是，dubbo中可以带key，这个key就是该实现类的映射key，可以根据这个key获取到该key对应的类实例。如图：

```
dubbo=cn.enjoy.dubbospi.DubboActivate
mybatis=cn.enjoy.dubbospi.MybatisActivate
rabbitmq1=cn.enjoy.dubbospi.Rabbitmq1Activate
rabbitmq2=cn.enjoy.dubbospi.Rabbitmq2Activate
rabbitmq=cn.enjoy.dubbospi.RabbitmqActivate
spring=cn.enjoy.dubbospi.SpringActivate
springcloud=cn.enjoy.dubbospi.SpringCloudActivate
```

3.1.4、单元测试

```
@Test
public void adaptive() {
    ActivateApi adaptiveExtension =
ExtensionLoader.getExtensionLoader(ActivateApi.class).getAdaptiveExtension();
    System.out.println(adaptiveExtension.getClass());
}
```

下面我们来介绍一下dubbo中几个非常重要的SPI的API。

3.2、getAdaptiveExtension()

3.2.1、方法功能

该方法是获取到一个接口的实现类，获取的方式有：

1、获取类上有@Adaptive注解的类的实例

2、如果接口的所有实现类都没有@Adaptive注解则dubbo动态生成一个

3.2.2、方法用法

```
//需要获取什么接口的实现类，getExtensionLoader中就传该接口的类型
ActivateApi adaptiveExtension =
ExtensionLoader.getExtensionLoader(ActivateApi.class).getAdaptiveExtension();
```

3.2.3、源码分析

3.2.3.1、ExtensionLoader

ExtensionLoader.getExtensionLoader(ActivateApi.class)，该方法调用是获取一个ExtensionLoader对象，核心就是会根据不同的接口类型创建不同的ExtensionLoader对象，换句话说就是一个接口类型对应这个ExtensionLoader对象。

```
public static <T> ExtensionLoader<T> getExtensionLoader(Class<T> type) {
    if (type == null) {
        throw new IllegalArgumentException("Extension type == null");
    }
    if (!type.isInterface()) {
```

```
        throw new IllegalArgumentException("Extension type (" + type + ") is not an
interface!");
    }
    //如果接口上没有@SPI注解，则报错
    if (!withExtensionAnnotation(type)) {
        throw new IllegalArgumentException("Extension type (" + type +
            ") is not an extension, because it is NOT annotated with @" +
SPI.class.getSimpleName() + "!");
    }
    //从缓存中获取
    ExtensionLoader<T> loader = (ExtensionLoader<T>) EXTENSION_LOADERS.get(type);
    if (loader == null) {
        //每一个@SPI接口类型都会对应一个ExtensionLoader对象
        //这里需要知道，一个接口类型对应着一个ExtensionLoader对象
        EXTENSION_LOADERS.putIfAbsent(type, new ExtensionLoader<T>(type));
        loader = (ExtensionLoader<T>) EXTENSION_LOADERS.get(type);
    }
    return loader;
}
```

3.2.3.2、getAdaptiveExtension

核心思想就是先从缓存中拿实例，如果没有才调用createAdaptiveExtension();方法创建实例。创建完成后放入到缓存中。缓存就是在ExtensionLoader中的一个Holder对象，如：Holder cachedAdaptiveInstance = new Holder<>();

```
public T getAdaptiveExtension() {
    //先从缓存中拿实例
    Object instance = cachedAdaptiveInstance.get();
    //DCL 思想
    if (instance == null) {
        if (createAdaptiveInstanceError != null) {
            throw new IllegalStateException("Failed to create adaptive instance: " +
                createAdaptiveInstanceError.toString(),
                createAdaptiveInstanceError);
        }

        synchronized (cachedAdaptiveInstance) {
            instance = cachedAdaptiveInstance.get();
            if (instance == null) {
                try {
                    //创建接口实例的核心方法
                    instance = createAdaptiveExtension();
                    cachedAdaptiveInstance.set(instance);
                } catch (Throwable t) {
                    createAdaptiveInstanceError = t;
                    throw new IllegalStateException("Failed to create adaptive instance: " +
t.toString(), t);
                }
            }
        }
    }
```

```
    return (T) instance;
}
```

3.2.3.3、createAdaptiveExtension

该方法就是创建实例的方法，创建实例并对实例进行IOC属性的依赖注入。

```
private T createAdaptiveExtension() {
    try {
        //injectExtension  是dubbo中的ioc逻辑，对属性进行赋值操作
        return injectExtension((T) getAdaptiveExtensionClass().newInstance());
    } catch (Exception e) {
        throw new IllegalStateException("Can't create adaptive extension " + type + ",
cause: " + e.getMessage(), e);
    }
}
```

3.2.3.4、getAdaptiveExtensionClass

获取到要实例化的类的Class对象并返回

```
private Class<?> getAdaptiveExtensionClass() {
    //核心方法 ，重点看 。建立名称和类的映射关系
    getExtensionClasses();
    //如果有@Adaptive注解的类，则返回该类
    if (cachedAdaptiveClass != null) {
        return cachedAdaptiveClass;
    }
    //动态拼凑类，动态编译生成
    return cachedAdaptiveClass = createAdaptiveExtensionClass();
}
```

3.2.3.5、getExtensionClasses

该方法是一个非常核心的方法，dubbo spi中的很多API都需要先调用这个方法来建立key和class的映射关系。获取映射关系的流程也是先从缓存里面拿，如果缓存没有才读配置文件建立映射关系，并把映射关系缓存起来。

该方法的作用：

1、建立key和class的映射关系

2、给ExtensionLoader里面的很多全局变量赋值，这些全局变量在dubbo spi的api中有用到

```
private Map<String, Class<?>> getExtensionClasses() {
    //先从缓存拿
    Map<String, Class<?>> classes = cachedClasses.get();
    //DCL
    if (classes == null) {
        synchronized (cachedClasses) {
            classes = cachedClasses.get();
            if (classes == null) {
                //从本地文件中加载key和类的关系
                classes = loadExtensionClasses();
```

```
                //把加载到的映射关系缓存起来
                cachedClasses.set(classes);
            }
        }
    }
    return classes;
}
```

3.2.3.6、loadExtensionClasses

该方法就是读取resources/META-INF/dubbo、META-INF/dubbo/internal/下面的配置文件，然后解析配置文件，建立key和class的映射关系，给ExtensionLoader里面的全局变量赋值等功能。

```
private Map<String, Class<?>> loadExtensionClasses() {
    //获取默认的实现类名称并缓存起来
    cacheDefaultExtensionName();

    Map<String, Class<?>> extensionClasses = new HashMap<>();

    //从jdk的spi机制中获取LoadingStrategy 实例
    for (LoadingStrategy strategy : strategies) {
        //加载目录下的文件，建立名称和类的映射关系 。核心逻辑
        loadDirectory(extensionClasses, strategy.directory(), type.getName(),
strategy.preferExtensionClassLoader(), strategy.overridden(), strategy.excludedPackages());
        loadDirectory(extensionClasses, strategy.directory(),
type.getName().replace("org.apache", "com.alibaba"), strategy.preferExtensionClassLoader(),
strategy.overridden(), strategy.excludedPackages());
    }

    return extensionClasses;
}
```

3.2.3.6.1、cacheDefaultExtensionName

设置ExtensionLoader中的全局变量cachedDefaultName的值，全局变量值的来源就是接口中@SPI("spring")注解的value值。

```
private void cacheDefaultExtensionName() {
    //获取类上的@SPI注解
    final SPI defaultAnnotation = type.getAnnotation(SPI.class);
    if (defaultAnnotation == null) {
        return;
    }

    String value = defaultAnnotation.value();
    //如果@SPI注解中有value值
    if ((value = value.trim()).length() > 0) {
        String[] names = NAME_SEPARATOR.split(value);
        if (names.length > 1) {
            throw new IllegalStateException("More than 1 default extension name on extension
" + type.getName()
                + ": " + Arrays.toString(names));
```

```
        }
        //把value值设置到 cachedDefaultName，，这个就是默认的实现类
        if (names.length == 1) {
            cachedDefaultName = names[0];
        }
    }
}
```

3.2.3.6.2、loadDirectory

加载目录下的所有文件，建立名称和类的映射关系，设置全局变量的值。循环调用loadResource

```
while (urls.hasMoreElements()) {
    java.net.URL resourceURL = urls.nextElement();
    //加载接口对应的文件的核心方法
    loadResource(extensionClasses, classLoader, resourceURL, overridden, excludedPackages);
}
```

3.2.3.6.3、loadResource

该方法就是对每一个文件进行处理的，建立名称和类的映射关系，设置全局变量的值。对读到的文件中的每一行数据进行处理。

```
private void loadResource(Map<String, Class<?>> extensionClasses, ClassLoader classLoader,
                          java.net.URL resourceURL, boolean overridden, String...
excludedPackages) {
    try {
        try (BufferedReader reader = new BufferedReader(new
InputStreamReader(resourceURL.openStream(), StandardCharsets.UTF_8))) {
            String line;
            String clazz = null;
            //读一行数据
            while ((line = reader.readLine()) != null) {
                final int ci = line.indexOf('#');
                //如果一行数据中有#号，则只要#前面那部分，,后面那部分可以写注释
                if (ci >= 0) {
                    line = line.substring(0, ci);
                }
                line = line.trim();
                if (line.length() > 0) {
                    try {
                        String name = null;
                        //=分割
                        int i = line.indexOf('=');
                        if (i > 0) {
                            //=号前面那部分是key
                            name = line.substring(0, i).trim();
                            //=号后面那部分则是类
                            clazz = line.substring(i + 1).trim();
                        } else {
                            clazz = line;
                        }
                        if (StringUtils.isNotEmpty(clazz) && !isExcluded(clazz,
```

```
excludedPackages)) {
                            //加载类的核心逻辑
                            loadClass(extensionClasses, resourceURL, Class.forName(clazz,
true, classLoader), name, overridden);
                        }
                    } catch (Throwable t) {
                        IllegalStateException e = new IllegalStateException("Failed to load
extension class (interface: " + type + ", class line: " + line + ") in " + resourceURL + ",
cause: " + t.getMessage(), t);
                        exceptions.put(line, e);
                    }
                }
            }
        }
    } catch (Throwable t) {
        logger.error("Exception occurred when loading extension class (interface: " +
            type + ", class file: " + resourceURL + ") in " + resourceURL, t);
    }
}
```

3.2.3.6.4、loadClass

该方法核心就是走了三套逻辑：

1、如果类上有@Adaptive注解

2、如果类是包装类

3、没有@Adaptive注解又不是包装类

```
private void loadClass(Map<String, Class<?>> extensionClasses, java.net.URL resourceURL,
Class<?> clazz, String name,
                       boolean overridden) throws NoSuchMethodException {
    //如果类类型和接口类型不一致，报错
    if (!type.isAssignableFrom(clazz)) {
        throw new IllegalStateException("Error occurred when loading extension class
(interface: " +
            type + ", class line: " + clazz.getName() + "), class "
            + clazz.getName() + " is not subtype of interface.");
    }
    //如果类上面有@Adaptive注解
    if (clazz.isAnnotationPresent(Adaptive.class)) {
        //在这个方法里面 对 cachedAdaptiveClass变量赋值
        cacheAdaptiveClass(clazz, overridden);
    } else if (isWrapperClass(clazz)) {
        //如果是包装类，包装类必然是持有目标接口的引用的，有目标接口对应的构造函数
        //对 cachedWrapperClasses 变量赋值
        cacheWrapperClass(clazz);
    } else {
        //获取类的无参构造函数，如果是包装类，这里会报错，,其实这里包装类走不进来了，包装类先处理的
        clazz.getConstructor();
        //如果没有配置key
        if (StringUtils.isEmpty(name)) {
            //如果类有Extension注解，则是注解的value，如果没注解则是类名称的小写做为name
```

```
            name = findAnnotationName(clazz);
            if (name.length() == 0) {
                throw new IllegalStateException("No such extension name for the class " +
clazz.getName() + " in the config " + resourceURL);
            }
        }

        String[] names = NAME_SEPARATOR.split(name);
        if (ArrayUtils.isNotEmpty(names)) {
            //如果类上面有@Activate注解，则建立名称和注解的映射
            //对 cachedActivates 全局变量赋值
            cacheActivateClass(clazz, names[0]);
            for (String n : names) {
                cacheName(clazz, n);
                //这里 extensionClasses 建立了 key和class的映射
                saveInExtensionClass(extensionClasses, clazz, n, overridden);
            }
        }
    }
}
```

3.2.3.7、createAdaptiveExtensionClass

前面分析类建立映射关系的过程，如果cachedAdaptiveClass属性不为空，也就是有@Adaptive注解的类，则直接返回该类，如果该属性为空，则会走到createAdaptiveExtensionClass方法，该方法的核心作用：

1、自动根据接口中的注解和参数配置生成类的字符串

2、根据类的字符串动态编译生成字节码文件加载到jvm

接口定义的规则

1、接口中的方法必须要有一个方法有@Adaptive注解，要不然会报错

2、方法的参数中必须要有URL参数，要不然会报错

为什么要这样设计

因为这种方式dubbo会自动生成一个类，这个类其实就是一个代理类，但是dubbo并不知道应该都哪个逻辑，也就是这个代理类并不知道走该接口的哪个实现类，所以我们必须用代理对象掉方法的时候用方法的入参告诉dubbo应该走哪个实现类，而选择掉哪个实现类的参数就在URL参数中。所以参数中必须要有URL参数，如下是dubbo生成的代理类：

```
import org.apache.dubbo.common.extension.ExtensionLoader;
public class ActivateApi$Adaptive implements cn.enjoy.dubbospi.ActivateApi {
    public java.lang.String todo(java.lang.String arg0, org.apache.dubbo.common.URL arg1)  {
        if (arg1 == null) throw new IllegalArgumentException("url == null");
        org.apache.dubbo.common.URL url = arg1;
        String extName = url.getParameter("activate.api", "spring");
        if(extName == null) throw new IllegalStateException("Failed to get extension
(cn.enjoy.dubbospi.ActivateApi) name from url (" + url.toString() + ") use
keys([activate.api])");
        cn.enjoy.dubbospi.ActivateApi extension =
(cn.enjoy.dubbospi.ActivateApi)ExtensionLoader.getExtensionLoader(cn.enjoy.dubbospi.Activate
Api.class).getExtension(extName);
        return extension.todo(arg0, arg1);
    }
}
```

从代码中可以看出，是根据url入参来选择接口的某个实例，然后用该实例去掉方法，也就是ActivateApi$Adaptive这种类只是一个代理层，并没有实质性的业务逻辑，是一个根据参数选择实例掉用的过程。

3.2.3.8、injectExtension

前面我们分析到已经生成了一个实例了，该方法就是对该实例进行属性的依赖注入操作，其实是掉该实例的setxxx方法进行参数赋值

```
private T injectExtension(T instance) {

    if (objectFactory == null) {
        return instance;
    }

    try {
        //对实例中的setXXX方法进行属性注入
        for (Method method : instance.getClass().getMethods()) {
            if (!isSetter(method)) {
                continue;
            }
            /**
             * Check {@link DisableInject} to see if we need auto injection for this
property
             */
            //如果有DisableInject注解，则不注入
            if (method.getAnnotation(DisableInject.class) != null) {
                continue;
            }
            Class<?> pt = method.getParameterTypes()[0];
            if (ReflectUtils.isPrimitives(pt)) {
                continue;
            }

            try {
                //根据方法名称获取该方法的属性名称
                String property = getSetterProperty(method);
                //获取需要依赖注入的值
```

```
                Object object = objectFactory.getExtension(pt, property);
                if (object != null) {
                    //反射赋值
                    method.invoke(instance, object);
                }
            } catch (Exception e) {
                logger.error("Failed to inject via method " + method.getName()
                    + " of interface " + type.getName() + ": " + e.getMessage(), e);
            }

        }
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    }
    return instance;
}
```

但是这里要强调的是，对调setxxx方法，参数值的来源

1、通过dubbo的spi方式获取到参数值

2、通过spring容器的getbean获取到参数值

dubbo spi的方式获取值

其实这种方法就是调用了getAdaptiveExtension方法获取到了实例

```
public class SpiExtensionFactory implements ExtensionFactory {

    @Override
    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().isEmpty()) {
                return loader.getAdaptiveExtension();
            }
        }
        return null;
    }

}
```

spring容器的方式获取值

这种方式就是掉getBean获取到容器中的实例

```
public class SpringExtensionFactory implements ExtensionFactory, Lifecycle {
    private static final Logger logger =
LoggerFactory.getLogger(SpringExtensionFactory.class);

    private static final Set<ApplicationContext> CONTEXTS = new
ConcurrentHashSet<ApplicationContext>();

    public static void addApplicationContext(ApplicationContext context) {
```

```
        CONTEXTS.add(context);
        if (context instanceof ConfigurableApplicationContext) {
            ((ConfigurableApplicationContext) context).registerShutdownHook();
            // see https://github.com/apache/dubbo/issues/7093
            DubboShutdownHook.getDubboShutdownHook().unregister();
        }
    }

    public static void removeApplicationContext(ApplicationContext context) {
        CONTEXTS.remove(context);
    }

    public static Set<ApplicationContext> getContexts() {
        return CONTEXTS;
    }

    // currently for test purpose
    public static void clearContexts() {
        CONTEXTS.clear();
    }

    @Override
    @SuppressWarnings("unchecked")
    public <T> T getExtension(Class<T> type, String name) {

        //SPI should be get from SpiExtensionFactory
        if (type.isInterface() && type.isAnnotationPresent(SPI.class)) {
            return null;
        }

        for (ApplicationContext context : CONTEXTS) {
            T bean = BeanFactoryUtils.getOptionalBean(context, name, type);
            if (bean != null) {
                return bean;
            }
        }

        //logger.warn("No spring extension (bean) named:" + name + ", try to find an
extension (bean) of type " + type.getName());

        return null;
    }

    @Override
    public void initialize() throws IllegalStateException {
        clearContexts();
    }

    @Override
    public void start() throws IllegalStateException {
        // no op
    }
```

```
    @Override
    public void destroy() {
        clearContexts();
    }
}
```

3.2.4、总结

getAdaptiveExtension方法是dubbo spi中一个分成核心的方法，做了两件事情，一个是获取实例，一个是在ExtensionLoader中赋值了全局变量，这个变量会用到其他api中。实例获取也分为两种，获取类上有@Adaptive注解的类的实例，一个是dubbo自动生成的代理实例，代理实例会根据接口配置来生成，接口的方法必须要有一个有@Adaptive注解，方法的入参必须要与URL参数。