链滴

Dubbo 系列笔记之 SPI 实现

作者：wangning1018

原文链接：https://ld246.com/article/1576826201969

来源网站：链滴

一、引言

在讲 Dubbo 的核心流程前，我想先说一下 Dubbo 的 SPI，正是有了这种服务发现机制，让 Dubbo 具有非常好的易拓展性。

SPI 全称为 Service Provider Interface，是一种服务发现机制。我们知道在 Java 中有接口、有实现接口定义了一个服务必须要实现的功能，至于实现的方式可以有多种。如果在编写代码时，不必明确死实现类，具体要使用哪个由使用者灵活声明，那我们的服务就可以有很强的拓展性，SPI 正是为了决这一问题。

在 Java 中的 SPI 实现，具体可以参见我之前写的《Java的SPI简介》。

SPI 的本质是将接口实现类的全限定名配置在文件中，并由服务加载器读取配置文件，加载实现类。样可以在运行时，动态为接口替换实现类。

Dubbo 作为一个扩展性非常好的开源框架，并没有沿用 Java 的 SPI，而是自己实现了一套 Dubbo SP
。

二、特性

Dubbo 为何要自己实现 SPI？其实答案很简单，无非是原有的功能不够完美，不够强大。

Dubbo 改进了 Java SPI 的下列问题：

- JDK 标准的 SPI 会一次性实例化扩展点所有实现，如果有扩展实现初始化很耗时，但如果没用也加载，会很浪费资源。
- 如果扩展点加载失败，连扩展点的名称都拿不到了。比如：JDK 标准的 ScriptEngine，通过 gtName() 获取脚本类型的名称，但如果 RubyScriptEngine 因为所依赖的 jruby.jar 不存在，导致 RuyScriptEngine 类加载失败，这个失败原因被吃掉了，和 ruby 对应不起来，当用户执行 ruby 脚本时会报不支持 ruby，而不是真正失败的原因。

- 增加了对扩展点 IOC 和 AOP 的支持，一个扩展点可以直接 setter 注入其它扩展点。

三、使用示例

使用 SPI 主要有四个重要的点：

- 接口
- 实现
- 实现配置文件

Dubbo 约定配置文件在 META-INF/dubbo/ 目录下。

- 服务加载/调用

Dubbo 对 SPI 机制的实现封装在了 ExtensionLoader 中。

使用方式：

1. 接口

```
package com.aysaml.spi;

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

/**
 * Say hello demo interface.
 *
 * @author wangning
 * @date 2019-12-19
 */
@SPI("demo")
public interface DemoService {

   void sayHello();
}
```

heart 注意：

在扩展的接口上必须加 @SPI 注解。

2. 实现

- 服务 A

```
package com.aysaml.spi.impl;

import com.aysaml.spi.DemoService;

/**
 * Say hello demo impl.
```

```
 *
 * @author wangning
 * @date 2019-12-19
 */
public class DemoServiceImplA implements DemoService {

  public void sayHello() {
    System.out.println("hello! This is A.");
  }
}
```

● 服务 B

```
package com.aysaml.spi.impl;

import com.aysaml.spi.DemoService;

/**
 * Say hello demo impl.
 *
 * @author wangning
 * @date 2019-12-19
 */
public class DemoServiceImplB implements DemoService {

  public void sayHello() {
    System.out.println("Hello! This is B.");
  }
}
```

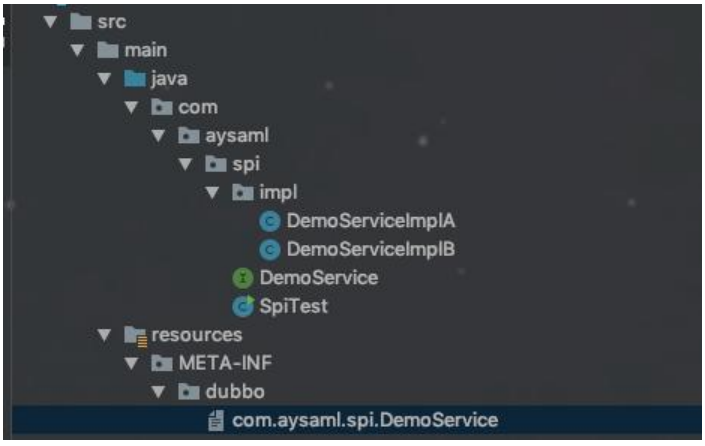
3. 添加配置文件

● 在META-INF/dubbo/ 目录下新建 文件，文件名为接口的全限定名 com.aysaml.spi.DemoServie。

● 在文件中添加实现配置：

```
serviceA = com.aysaml.spi.impl.DemoServiceImplA
serviceB = com.aysaml.spi.impl.DemoServiceImplB
```

如图：



heart 注意：

与 Java 的 SPI 相比配置有下面两点不同

1. 配置目录不同：Java SPI 为 META-INF/services/；Dubbo SPI 为 META-INF/dubbo/

2. 配置方式不同：Java SPI 直接配置为服务实现的全限定名；Dubbo SPI 则采用 key-value 的形，为服务实现指定别名，方便按需加载。内容为：配置名=扩展实现类全限定名，多个实现类用换行分隔。

4. 服务加载调用

```
package com.aysaml.spi;

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

/**
 * The run dubbo SPI test.
 *
 * @author wangning
 * @date 2019-12-19
 */
public class SpiTest {

  public static void main(String[] args) {
      ExtensionLoader<DemoService> loader = ExtensionLoader.getExtensionLoader(DemoServ
ce.class);
      // service A
      DemoService serviceA = loader.getExtension("serviceA");
      serviceA.sayHello();

      // service B
      DemoService serviceB = loader.getExtension("serviceB");
      serviceB.sayHello();
  }
}
```

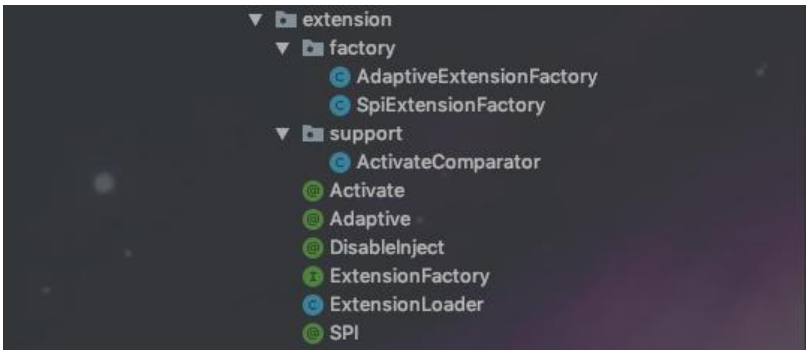
效果如下：

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
十二月 19, 2019 4:47:40 下午 com.alibaba.dubbo.common.logger.LoggerFactory info
信息: using logger: com.alibaba.dubbo.common.logger.jcl.JclLoggerAdapter
hello! This is A.
Hello! This is B.

Process finished with exit code 0
```

四、源码一览

Dubbo 关于 SPI 的实现在 dubbo-common 模块下的 com.alibaba.dubbo.common.extension 包。



在上面演示了 Dubbo SPI 的用法，其中使用ExtensionLoader.getExtensionLoader(DemoService.class) 方法获得了一个 ExtensionLoader 实例。然后通过ExtensionLoader 的 getExtension("service") 方法获得拓展的类实现 ServiceA，这一系列操作都依赖于拓展加载器 ExtensionLoader 。

首先，从 ExtensionLoader 的 getExtensionLoader 方法入手，来说一下 Dubbo SPI 是如何获得拓的类实现的。

- getExtensionLoader 方法

```
    /**
     * 根据接口类型获取拓展加载器
     *
     * @param type
     * @param <T>
     * @return
     */
    @SuppressWarnings("unchecked")
    public static <T> ExtensionLoader<T> getExtensionLoader(Class<T> type) {
        // 参数验证为空
        if (type == null)
            throw new IllegalArgumentException("Extension type == null");
        // 是否是接口
        if (!type.isInterface()) {
            throw new IllegalArgumentException("Extension type(" + type + ") is not interface!");
        }
        // 接口是否加了 @SPI 注解
        if (!withExtensionAnnotation(type)) {
            throw new IllegalArgumentException("Extension type(" + type +
                    ") is not extension, because WITHOUT @" + SPI.class.getSimpleName() + " Annot
tion!");
        }
        // EXTENSION_LOADERS 是一个 map，以接口类型为 key，拓展加载器 ExtensionLoader 为
value。
        // 通过 map 做一个缓存，首先从缓存中取
        ExtensionLoader<T> loader = (ExtensionLoader<T>) EXTENSION_LOADERS.get(type);
        // 缓存未命中
        if (loader == null) {
            // 创建一个拓展加载器，并加入缓存
            EXTENSION_LOADERS.putIfAbsent(type, new ExtensionLoader<T>(type));
            loader = (ExtensionLoader<T>) EXTENSION_LOADERS.get(type);
        }
        return loader;
    }
```

由上成功获取了拓展加载器，接着调用 ExtensionLoader 的 getExtension("serviceA") 方法获得拓的类实现。

- getExtension 方法

```
/**
 * 返回指定名字的拓展类对象。 如果名字未找到则抛出 {@link IllegalStateException} 异常。
 *
 * @param name
 * @return
 */
@SuppressWarnings("unchecked")
public T getExtension(String name) {
    // 参数验证
    if (name == null || name.length() == 0)
        throw new IllegalArgumentException("Extension name == null");
    if ("true".equals(name)) {
        // 若 name 为 true，则获取默认的拓展类实现
        return getDefaultExtension();
    }
    // Holder 作为一个缓存，持有拓展对象，从缓存中获得拓展对象
    Holder<Object> holder = cachedInstances.get(name);
    if (holder == null) {
        cachedInstances.putIfAbsent(name, new Holder<Object>());
        holder = cachedInstances.get(name);
    }
    Object instance = holder.get();
    // 这里比较有意思，锁的双重检查
    if (instance == null) {
        synchronized (holder) {
            instance = holder.get();
            // 如果缓存中没有
            if (instance == null) {
                // 创建类拓展对象
                instance = createExtension(name);
                // 加入缓存
                holder.set(instance);
            }
        }
    }
    return (T) instance;
}
```

接下来继续看 createExtension 方法怎么创建类拓展对象，

- createExtension 方法

```
/**
 * 根据名字创建类拓展对象
 *
 * @param name
 * @return
 */
@SuppressWarnings("unchecked")
private T createExtension(String name) {
```

```
        // 通过名字获取拓展类实现的 class 对象
        Class<?> clazz = getExtensionClasses().get(name);
        // 没有实现类，抛出未找到异常
        if (clazz == null) {
            throw findException(name);
        }
        try {
            // 从缓存中取拓展类对象
            T instance = (T) EXTENSION_INSTANCES.get(clazz);
            // 未命中缓存，创建类拓展对象并加入缓存
            if (instance == null) {
                EXTENSION_INSTANCES.putIfAbsent(clazz, clazz.newInstance());
                instance = (T) EXTENSION_INSTANCES.get(clazz);
            }
            // 通过反射注入依赖的属性
            injectExtension(instance);
            Set<Class<?>> wrapperClasses = cachedWrapperClasses;
            if (wrapperClasses != null && !wrapperClasses.isEmpty()) {
                // 循环创建 Wrapper 拓展对象
                for (Class<?> wrapperClass : wrapperClasses) {
                    // 将 instance 作为参数传给 Wrapper拓展对象的构造方法，并通过反射创建 Wrapper
实例
                    // 然后向 Wrapper 实例中注入依赖，再将 Wrapper 的实例赋值给 instance
                    instance = injectExtension((T) wrapperClass.getConstructor(type).newInstance(ins
ance));
                }
            }
            return instance;
        } catch (Throwable t) {
            throw new IllegalStateException("Extension instance(name: " + name + ", class: " +
                    type + ")  could not be instantiated: " + t.getMessage(), t);
        }
    }
```

如此，再对创建类拓展对象的步骤做一个总结：

1. 获取拓展类实现的所有 class 对象。
2. 通过 clazz.newInstance() 方法创建类拓展对象。
3. 通过反射注入依赖的属性。
4. 将拓展对象包裹在相应的 Wrapper 对象中。

Wrapper 对类拓展对象进行了包装、代理，至于他到底为什么这么做，我们在 Dubbo 的扩展点加载说明。

在上面说创建类拓展对象的步骤中，第一步是创建拓展类的的关键，第三步和第四步是 Dubbo IOC 和 AOP 的实现。下面分别来看看具体是如何实现的：

- createExtension 方法

```
    /**
     * 获取所有的拓展类 class 对象
     *
     * @return
     */
```

```
    private Map<String, Class<?>> getExtensionClasses() {
        // cachedClasses 缓存了所有的拓展类 class 对象
        // 首先从缓存中取
        Map<String, Class<?>> classes = cachedClasses.get();
        if (classes == null) {
            synchronized (cachedClasses) {
                classes = cachedClasses.get();
                if (classes == null) {
                    // 从配置文件中读取拓展类 class 对象
                    classes = loadExtensionClasses();
                    // 加入缓存
                    cachedClasses.set(classes);
                }
            }
        }
        return classes;
    }
```

这个方法逻辑比较简单，首先从缓存中取，没有取到就从配置文件中读取拓展类，然后加入缓存。接来我们继续看怎么从配置文件中读取拓展类：

- loadExtensionClasses 方法

```
    /**
     * 从配置文件中读取拓展类
     *
     * @return
     */
    private Map<String, Class<?>> loadExtensionClasses() {
        // 这个 type 从哪来的呢？type 是当前拓展类的接口类型，在获取拓展类加载器传入的
        // 获取接口的 @SPI 注解，这里就是当初为什么拓展类接口必须标注 @SPI 的原因，哈哈哈
        final SPI defaultAnnotation = type.getAnnotation(SPI.class);
        if (defaultAnnotation != null) {
            // 获取 @SPI 注解的 value , 即定义的拓展名称
            String value = defaultAnnotation.value();
            // 对名字一系列校验，不能有特殊字符什么的
            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 extensi
n " + type.getName()
                            + ": " + Arrays.toString(names));
                }
                // 设置默认的名称
                if (names.length == 1) cachedDefaultName = names[0];
            }
        }

        Map<String, Class<?>> extensionClasses = new HashMap<String, Class<?>>();
        // 从三种不同的目录下读取加载，其中还包含了 Java SPI 的路径
        loadDirectory(extensionClasses, DUBBO_INTERNAL_DIRECTORY);
        loadDirectory(extensionClasses, DUBBO_DIRECTORY);
        loadDirectory(extensionClasses, SERVICES_DIRECTORY);
        return extensionClasses;
```

```
    }
```

同样，这个方法也比较简单，首先获取 @SPI 注解，对注解的 value 处理，设置拓展默认名称，然后别从不同的文件夹路径读取加载拓展类。

所以，接下来看从不同路径读取加载拓展类方法的实现：

- loadDirectory 方法

```
/**
 * 从指定目录加载指定的拓展类
 *
 * @param extensionClasses
 * @param dir
 */
private void loadDirectory(Map<String, Class<?>> extensionClasses, String dir) {
    // 路径加接口全限定名
    String fileName = dir + type.getName();
    try {
        // 用来保存配置文件链接
        Enumeration<java.net.URL> urls;
        // 获取类加载器
        ClassLoader classLoader = findClassLoader();
        if (classLoader != null) {
            urls = classLoader.getResources(fileName);
        } else {
            urls = ClassLoader.getSystemResources(fileName);
        }
        if (urls != null) {
            while (urls.hasMoreElements()) {
                java.net.URL resourceURL = urls.nextElement();
                // 循环加载资源
                loadResource(extensionClasses, classLoader, resourceURL);
            }
        }
    } catch (Throwable t) {
        logger.error("Exception when load extension class(interface: " +
                type + ", description file: " + fileName + ").", t);
    }
}
```

这个方法就是通过类加载器获取对应的所有配置的资源链接，然后通过 loadResource 方法加载资源接下来继续看 loadResource 方法：

- loadResource 方法

```
/**
 * 加载拓展类实现
 *
 * @param extensionClasses
 * @param classLoader
 * @param resourceURL
 */
private void loadResource(Map<String, Class<?>> extensionClasses, ClassLoader classLoa
```

```
er, java.net.URL resourceURL) {
        try {
            // 读取配置文件
            BufferedReader reader = new BufferedReader(new InputStreamReader(resourceURL.o
enStream(), "utf-8"));
            try {
                String line;
                while ((line = reader.readLine()) != null) {
                    // 注释处理
                    final int ci = line.indexOf('#');
                    if (ci >= 0) line = line.substring(0, ci);
                    line = line.trim();
                    if (line.length() > 0) {
                        try {
                            String name = null;
                            // 等号下标
                            int i = line.indexOf('=');
                            if (i > 0) {
                                // 等号之前是拓展类名字
                                name = line.substring(0, i).trim();
                                // 等号之后是拓展的实现类
                                line = line.substring(i + 1).trim();
                            }
                            if (line.length() > 0) {
                                // 通过 Class.forName 反射加载拓展类，具体的下面看
                                loadClass(extensionClasses, resourceURL, Class.forName(line, true, classLo
der), name);
                            }
                        } catch (Throwable t) {
                            IllegalStateException e = new IllegalStateException("Failed to load extension
class(interface: " + type + ", class line: " + line + ") in " + resourceURL + ", cause: " + t.getMes
age(), t);
                            exceptions.put(line, e);
                        }
                    }
                }
            } finally {
                reader.close();
            }
        } catch (Throwable t) {
            logger.error("Exception when load extension class(interface: " +
                    type + ", class file: " + resourceURL + ") in " + resourceURL, t);
        }
    }
```

loadResource 方法主要处理配置文件的读取、解析和拓展实现类的加载。下面我们接着看 loadClass 方法都做了什么：

- loadClass

```
    /**
     * 各种缓存操作
     *
     * @param extensionClasses
```

```
     * @param resourceURL
     * @param clazz
     * @param name
     * @throws NoSuchMethodException
     */
    private void loadClass(Map<String, Class<?>> extensionClasses, java.net.URL resourceURL,
Class<?> clazz, String name) throws NoSuchMethodException {
        // 校验加载的类是否是当前接口的实现
        if (!type.isAssignableFrom(clazz)) {
            throw new IllegalStateException("Error when load extension class(interface: " +
                    type + ", class line: " + clazz.getName() + "), class "
                    + clazz.getName() + "is not subtype of interface.");
        }
        // 是否有 @Adaptive 注解
        // @Adaptive 是什么呢，与自适应扩展有关，这个后面看
        if (clazz.isAnnotationPresent(Adaptive.class)) {
            if (cachedAdaptiveClass == null) {
                // 缓存
                cachedAdaptiveClass = clazz;
            } else if (!cachedAdaptiveClass.equals(clazz)) {
                // 只能有一个自适应扩展类
                throw new IllegalStateException("More than 1 adaptive class found: "
                        + cachedAdaptiveClass.getClass().getName()
                        + ", " + clazz.getClass().getName());
            }
            // Wrapper 类型的
        } else if (isWrapperClass(clazz)) {
            Set<Class<?>> wrappers = cachedWrapperClasses;
            if (wrappers == null) {
                // 缓存
                cachedWrapperClasses = new ConcurrentHashSet<Class<?>>();
                wrappers = cachedWrapperClasses;
            }
            wrappers.add(clazz);
            // 普通类型的拓展类
        } else {
            clazz.getConstructor();
            // 拓展类名字为空
            if (name == null || name.length() == 0) {
                // 从 @Extension 注解获取名字，或者直接使用类型小写作为名字
                name = findAnnotationName(clazz);
                if (name.length() == 0) {
                    throw new IllegalStateException("No such extension name for the class " + clazz.
etName() + " in the config " + resourceURL);
                }
            }
            String[] names = NAME_SEPARATOR.split(name);
            if (names != null && names.length > 0) {
                Activate activate = clazz.getAnnotation(Activate.class);
                // 类如果存在 @Activate 注解
                if (activate != null) {
                    // 加入其缓存
                    cachedActivates.put(names[0], activate);
                }
```

```
                for (String n : names) {
                    if (!cachedNames.containsKey(clazz)) {
                        // 加入 names 缓存
                        cachedNames.put(clazz, n);
                    }
                    Class<?> c = extensionClasses.get(n);
                    if (c == null) {
                        // class 缓存
                        extensionClasses.put(n, clazz);
                    } else if (c != clazz) {
                        throw new IllegalStateException("Duplicate extension " + type.getName() + " n
me " + n + " on " + c.getName() + " and " + clazz.getName());
                    }
                }
            }
        }
    }
```

可以看到这个方法主要是在加载完拓展类之后做的各种类型的缓存。

Dubbo SPI 拓展类的加载过程我们就了解完了，过程其实比较简单，一步一步看下来比较容易理解，果遇到不懂的地方可以结合调试来理解。

五、Dubbo IOC

Dubbo 的依赖注入目前仅有一种，就是通过对象的 setter 方法进行注入。逻辑比较简单，首先获取展类的 setter 方法，然后再通过 objectFactory 的 getExtension 的方法获取依赖的拓展类，然后通反射调用 setter 方法进行依赖注入。下面看下代码实现。

```
    /**
     * 依赖注入
     *
     * @param instance
     * @return
     */
    private T injectExtension(T instance) {
        try {
            // objectFactory 是拓展类的工厂，可以通过它的 getExtension 获取扩展类
            // 在扩展类加载器的构造方法中初始化，有两种实现 SpiExtensionFactory 和 SpringExtensi
nFactory
            // SpiExtensionFactory 用于获取自适应的拓展；SpringExtensionFactory 用来从 Spring
获取所需的拓展类
            if (objectFactory != null) {
                for (Method method : instance.getClass().getMethods()) {
                    // Dubbo 通过 setter 实现依赖注入
                    // 获取本扩展类的 set 方法
                    if (method.getName().startsWith("set")
                            && method.getParameterTypes().length == 1
                            && Modifier.isPublic(method.getModifiers())) {
                        // 有 @DisableInject 注解的 set 不需要依赖注入
                        if (method.getAnnotation(DisableInject.class) != null) {
                            continue;
                        }
```

```
                    // 获取参数 class 对象
                    Class<?> pt = method.getParameterTypes()[0];
                    try {
                        // 获取属性名，也就是 set 方法后面的第一个字母变小写 0.0
                        String property = method.getName().length() > 3 ? method.getName().subst
ing(3, 4).toLowerCase() + method.getName().substring(4) : "";
                        // 通过拓展工厂获取拓展类
                        Object object = objectFactory.getExtension(pt, property);
                        if (object != null) {
                            // 通过反射调用 set 方法注入
                            method.invoke(instance, object);
                        }
                    } catch (Exception e) {
                        logger.error("fail to inject via method " + method.getName()
                                + " of interface " + type.getName() + ": " + e.getMessage(), e);
                    }
                }
            }
        }
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    }
    return instance;
}
```

六、总结

关于 Dubbo SPI 的基本使用和它的加载拓展类的过程到现在就了解完了，Dubbo 对 Java SPI 的功做了扩展，实现了可以按需加载拓展类实现，节省了资源，同时增加了 IOC 和 AOP 特性 。限于篇，其实关于 Dubbo SPI 还有一部分比较重要的没有说，即 Dubbo SPI 的扩展点自适应机制。那么扩点自适应机制到底是什么呢？我们将会在下篇做讨论。