扩展点加载机制(ExtensionLoader)

标签： dubbo
2015-04-08 21:39 6302人阅读 评论(1) 收藏 举报

分类：

dubbo

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录(?)[+]

概述

来源：

Dubbo的扩展点加载从JDK标准的SPI(Service Provider Interface)扩展点发现机制加强而来。

Dubbo改进了JDK标准的SPI的以下问题：

- 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，但如果没用上也加载，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getName();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RubyScriptEngine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。

约定：

在扩展类的jar包内，放置扩展点配置文件：META-INF/dubbo/接口全限定名，内容为：配置名=扩展实现类全限定名，多个实现类用换行符分隔。

(注意：这里的配置文件是放在你自己的jar包内，不是dubbo本身的jar包内，Dubbo会全ClassPath扫描所有jar包内同名的这个文件，然后进行合并)

扩展Dubbo的协议示例：

在协议的实现jar包内放置文本文件：META-

INF/dubbo/com.alibaba.dubbo.rpc.Protocol，内容为：

```
xxx=com.alibaba.xxx.XxxProtocol
```

实现内容：

```
package com.alibaba.xxx;

import com.alibaba.dubbo.rpc.Protocol;
public class XxxProtocol implemenets Protocol {

    // ...
}
```

注意： 扩展点使用单一实例加载（请确保扩展实现的线程安全性），Cache在ExtensionLoader中

特性

- 扩展点自动包装
- 扩展点自动装配
- 扩展点自适应
- 扩展点自动激活

相关文档可以参考dubbo的官方文档 ， 本文主要通过分析相关的源代码来体会dubbo的扩展点框架提供的特性。

源码分析

dubbo的扩展点框架主要位于这个包下：

com.alibaba.dubbo.common.extension

大概结构如下：

```
com.alibaba.dubbo.common.extension
 |
 |--factory
 |      |--AdaptiveExtensionFactory    #稍后解释
 |      |--SpiExtensionFactory         #稍后解释
 |
 |--support
 |      |--ActivateComparator
 |
 |--Activate  #自动激活加载扩展的注解
 |--Adaptive  #自适应扩展点的注解
 |--ExtensionFactory  #扩展点对象生成工厂接口
 |--ExtensionLoader   #扩展点加载器，扩展点的查找，校验，加载等核心逻辑的实现类
 |--SPI   #扩展点注解
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 1
- 2
- 3
- 4
- 5

- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14

其中最核心的类就是`ExtensionLoader`，几乎所有特性都在这个类中实现，先来看下他的结构：

`ExtensionLoader`没有提供`public`的构造方法，但是提供了一个`public static`的`getExtensionLoader`，这个方法就是获取`ExtensionLoader`实例的工厂方法。

其`public`成员方法中有三个比较重要的方法：

- getActivateExtension： 根据条件获取当前扩展可自动激活的实现
- getExtension： 根据名称获取当前扩展的指定实现
- getAdaptiveExtension : 获取当前扩展的自适应实现

这三个方法将会是我们重点关注的方法；*每一个`ExtensionLoader`实例仅负责加载特定`SPI`扩展的实现*。因此想要获取某个扩展的实现，首先要获取到该扩展对应的`ExtensionLoader`实例，下面我们就来看一下获取`ExtensionLoader`实例的工厂方法`getExtensionLoader`：

```
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 interface!");
    }
    if(!withExtensionAnnotation(type)) { // 只接受使用@SPI注解
注释的接口类型
        throw new IllegalArgumentException("Extension
type(" + type +
                "") is not extension, because WITHOUT @" +
SPI.class.getSimpleName() + " Annotation!");
```

```
    }

    // 先从静态缓存中获取对应的ExtensionLoader实例
    ExtensionLoader<T> loader = (ExtensionLoader<T>)
EXTENSION_LOADERS.get(type);
    if (loader == null) {
        EXTENSION_LOADERS.putIfAbsent(type, new
ExtensionLoader<T>(type)); // 为Extension类型创建
ExtensionLoader实例，并放入静态缓存
        loader = (ExtensionLoader<T>)
EXTENSION_LOADERS.get(type);
    }
    return loader;
}
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

```
1
2
3
4
5
6
7
8
```

- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19

该方法需要一个`Class`类型的参数，该参数表示希望加载的扩展点类型，该参数必须是接口，且该接口必须被`@SPI`注解注释，否则拒绝处理。检查通过之后首先会检查ExtensionLoader缓存中是否已经存在该扩展对应的`ExtensionLoader`，如果有则直接返回，否则创建一个新的`ExtensionLoader`负责加载该扩展实现，同时将其缓存起来。可以看到对于每一个扩展，dubbo中只会有一个对应的`ExtensionLoader`实例。

接下来看下`ExtensionLoader`的私有构造函数：

```
private ExtensionLoader(Class<?> type) {
    this.type = type;

    // 如果扩展类型是ExtensionFactory,那么则设置为null
    // 这里通过getAdaptiveExtension方法获取一个运行时自适应的扩展类型(每个Extension只能有一个@Adaptive类型的实现，如果没有dubbo会动态生成一个类)
    objectFactory = (type == ExtensionFactory.class ? null :
ExtensionLoader.getExtensionLoader(ExtensionFactory.class).
getAdaptiveExtension());
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 1
- 2

- 3
- 4
- 5
- 6
- 7

这里保存了对应的扩展类型，并且设置了一个额外的`objectFactory`属性，他是一个`ExtensionFactory`类型，`ExtensionFactory`主要用于加载扩展的实现：

```
@SPI
public interface ExtensionFactory {

    /**
     * Get extension.
     *
     * @param type object type.
     * @param name object name.
     * @return object instance.
     */
    <T> T getExtension(Class<T> type, String name);

}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13

- 1
- 2
- 3
- 4
- 5
- 6
- 7

- 8
- 9
- 10
- 11
- 12
- 13

同时`ExtensionFactory`也被`@SPI`注解注释，说明他也是一个扩展点，从前面`com.alibaba.dubbo.common.extension`包的结构图中可以看到，dubbo内部提供了两个实现类：`SpiExtensionFactory` 和 `AdaptiveExtensionFactory`，实际上还有一个`SpringExtensionFactory`，不同的实现可以已不同的方式来完成扩展点实现的加载，这块稍后再来学习。从`ExtensionLoader`的构造函数中可以看到，如果要加载的扩展点类型是`ExtensionFactory`是，`object`字段被设置为null。由于`ExtensionLoader`的使用范围有限(基本上局限在`ExtensionLoader`中)，因此对他做了特殊对待：在需要使用`ExtensionFactory`的地方，都是通过对应的自适应实现来代替。

默认的`ExtensionFactory`实现中，`AdaptiveExtensionFactotry`被`@Adaptive`注解注释，也就是它就是`ExtensionFactory`对应的自适应扩展实现(**每个扩展点最多只能有一个自适应实现，如果所有实现中没有被`@Adaptive`注释的，那么dubbo会动态生成一个自适应实现类**)，也就是说，所有对`ExtensionFactory`调用的地方，实际上调用的都是`AdpativeExtensionFactory`，那么我们看下他的实现代码：

```
@Adaptive
public class AdaptiveExtensionFactory implements
ExtensionFactory {

    private final List<ExtensionFactory> factories;

    public AdaptiveExtensionFactory() {
        ExtensionLoader<ExtensionFactory> loader =
ExtensionLoader.getExtensionLoader(ExtensionFactory.class);
        List<ExtensionFactory> list = new
ArrayList<ExtensionFactory>();
        for (String name : loader.getSupportedExtensions())
{ // 将所有ExtensionFactory实现保存起来
            list.add(loader.getExtension(name));
        }
```

```
        factories = Collections.unmodifiableList(list);
    }

    public <T> T getExtension(Class<T> type, String name) {
        // 依次遍历各个ExtensionFactory实现的getExtension方法,
一旦获取到Extension即返回
        // 如果遍历完所有的ExtensionFactory实现均无法找到
Extension,则返回null
        for (ExtensionFactory factory : factories) {
            T extension = factory.getExtension(type, name);
            if (extension != null) {
                return extension;
            }
        }
        return null;
    }

}
```

```
24
25
26
27
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
```

看完代码大家都知道是怎么回事了，这货就相当于一个代理入口，他会遍历当前系统中所有的`ExtensionFactory`实现来获取指定的扩展实现，获取到扩展实现或遍历完所有的`ExtensionFactory`实现。这里调用了`ExtensionLoader`的`getSupportedExtensions`方法来获取`ExtensionFactory`的所有实现，又回到了`ExtensionLoader`类，下面我们就来分析`ExtensionLoader`的几个重要的实例方法。

方法调用流程

getExtension

```
getExtension(name)
    -> createExtension(name) #如果无缓存则创建
        -> getExtensionClasses().get(name) #获取name对应的扩展类型
        -> 实例化扩展类
        -> injectExtension(instance) # 扩展点注入
        -> instance = injectExtension((T) wrapperClass.getConstructor(type).newInstance(instance)) #循环遍历所有wrapper实现，实例化wrapper并进行扩展点注入
```

- 1
- 2
- 3
- 4
- 5
- 6

- 1
- 2
- 3
- 4
- 5
- 6

getAdaptiveExtension

```
public T getAdaptiveExtension()
    -> createAdaptiveExtension() #如果无缓存则创建
        -> getAdaptiveExtensionClass().newInstance() #获取AdaptiveExtensionClass
            -> getExtensionClasses() # 加载当前扩展所有实现，看是否有实现被标注为@Adaptive
            -> createAdaptiveExtensionClass() #如果没有实现被标注为@Adaptive，则动态创建一个Adaptive实现类
                -> createAdaptiveExtensionClassCode() #动态生成实现类java代码
                -> compiler.compile(code, classLoader) #动态编译java代码，加载类并实例化
        -> injectExtension(instance)
```

- 1
- 2
- 3
- 4

- 5
- 6
- 7
- 8
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

getActivateExtesion

该方法有多个重载方法，不过最终都是调用了三个参数的那一个重载形式。其代码结构也相对剪短，就不需要在列出概要流程了。

详细代码分析

getAdaptiveExtension

从前面`ExtensionLoader`的私有构造函数中可以看出，在选择`ExtensionFactory`的时候，并不是调用`getExtension(name)`来获取某个具体的实现类，而是调用`getAdaptiveExtension`来获取一个自适应的实现。那么首先我们就来分析一下`getAdaptiveExtension`这个方法的实现吧：

```
public T getAdaptiveExtension() {
    Object instance = cachedAdaptiveInstance.get(); // 首先判断是否已经有缓存的实例对象
    if (instance == null) {
        if(createAdaptiveInstanceError == null) {
            synchronized (cachedAdaptiveInstance) {
                instance = cachedAdaptiveInstance.get();
                if (instance == null) {
                    try {
                        instance = createAdaptiveExtension(); // 没有缓存的实例，创建新的AdaptiveExtension实例

cachedAdaptiveInstance.set(instance);
                    } catch (Throwable t) {
                        createAdaptiveInstanceError = t;
```

```
                        throw new
IllegalStateException("fail to create adaptive instance: "
+ t.toString(), t);
                    }
                }
            }
        }
        else {
            throw new IllegalStateException("fail to create
adaptive instance: " +
createAdaptiveInstanceError.toString(),
createAdaptiveInstanceError);
        }
    }

    return (T) instance;
}
```

首先检查缓存的adaptiveInstance是否存在，如果存在则直接使用，否则的话调用`createAdaptiveExtension`方法来创建新的adaptiveInstance并且缓存起来。也就是说对于某个扩展点，每次调用`ExtensionLoader.getAdaptiveExtension`获取到的都是同一个实例。

```
private T createAdaptiveExtension() {
    try {
        return injectExtension((T)
getAdaptiveExtensionClass().newInstance()); // 先获取
AdaptiveExtensionClass，在获取其实例，最后进行注入处理
    } catch (Exception e) {
        throw new IllegalStateException("Can not create
adaptive extenstion " + type + ", cause: " +
e.getMessage(), e);
    }
}
```

在createAdaptiveExtension方法中，首先通过getAdaptiveExtensionClass方法获取到最终的自适应实现类型，然后实例化一个自适应扩展实现的实例，最后进行扩展点注入操作。先看一个getAdaptiveExtensionClass方法的实现：

```
private Class<?> getAdaptiveExtensionClass() {
    getExtensionClasses(); // 加载当前Extension的所有实现,如果有@Adaptive类型，则会赋值为cachedAdaptiveClass属性缓存起来
    if (cachedAdaptiveClass != null) {
        return cachedAdaptiveClass;
    }
    return cachedAdaptiveClass = createAdaptiveExtensionClass(); // 没有找到@Adaptive类型实现，则动态创建一个AdaptiveExtensionClass
}
```

他只是简单的调用了`getExtensionClasses`方法，然后在判adaptiveCalss缓存是否被设置，如果被设置那么直接返回，否则调用`createAdaptiveExntesionClass`方法动态生成一个自适应实现，关于动态生成自适应实现类然后编译加载并且实例化的过程这里暂时不分析，留到后面在分析吧。这里我们

看`getExtensionClassses`方法：

```
private Map<String, Class<?>> getExtensionClasses() {
    Map<String, Class<?>> classes = cachedClasses.get(); // 判断是否已经加载了当前Extension的所有实现类
    if (classes == null) {
        synchronized (cachedClasses) {
            classes = cachedClasses.get();
            if (classes == null) {
                classes = loadExtensionClasses(); // 如果还没有加载Extension的实现，则进行扫描加载,完成后赋值给cachedClasses变量
                cachedClasses.set(classes);
            }
        }
    }
    return classes;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13

- 1
- 2
- 3
- 4
- 5
- 6

在`getExtensionClasses`方法中，首先检查缓存的cachedClasses，如果没有再调用`loadExtensionClasses`方法来加载，加载完成之后就会进行缓存。也就是说对于每个扩展点，其实现的加载只会执行一次。我们看下`loadExtensionClasses`方法：

```
private Map<String, Class<?>> loadExtensionClasses() {
    final SPI defaultAnnotation =
type.getAnnotation(SPI.class);
    if(defaultAnnotation != null) {
        String value = defaultAnnotation.value(); // 解析当前
Extension配置的默认实现名，赋值给cachedDefaultName属性
        if(value != null && (value = value.trim()).length()
> 0) {
            String[] names = NAME_SEPARATOR.split(value);
            if(names.length > 1) { // 每个扩展实现只能配置一个名
称
                throw new IllegalStateException("more than
1 default extension name on extension " + type.getName()
                        + ": " + Arrays.toString(names));
            }
            if(names.length == 1) cachedDefaultName =
names[0];
        }
    }

    // 从配置文件中加载扩展实现类
    Map<String, Class<?>> extensionClasses = new
HashMap<String, Class<?>>();
    loadFile(extensionClasses, DUBBO_INTERNAL_DIRECTORY);
    loadFile(extensionClasses, DUBBO_DIRECTORY);
    loadFile(extensionClasses, SERVICES_DIRECTORY);
    return extensionClasses;
}
```

- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21

从代码里面可以看到，在`loadExtensionClasses`中首先会检测扩展点在`@SPI`注解中配置的默认扩展实现的名称，并将其赋值给`cachedDefaultName`属性进行缓存，后面想要获取该扩展点的默认实现名称就可以直接通过访问`cachedDefaultName`字段来完成，比如`getDefaultExtensionName`方法就是这么实现的。从这里的代码中又可以看到，具体的扩展实现类型，是通过调用`loadFile`方法来加载，分别从一下三个地方加载：

- META-INF/dubbo/internal/
- META-INF/dubbo/
- META-INF/services/

那么这个`loadFile`方法则至关重要了，看看其源代码：

```
private void loadFile(Map<String, Class<?>>
extensionClasses, String dir) {
    String fileName = dir + type.getName(); // 配置文件名称,扫
描整个classpath
    try {
        // 先获取该路径下所有文件
        Enumeration<java.net.URL> urls;
        ClassLoader classLoader = findClassLoader();
        if (classLoader != null) {
            urls = classLoader.getResources(fileName);
        } else {
            urls =
ClassLoader.getSystemResources(fileName);
        }
        if (urls != null) {
            // 遍历这些文件并进行处理
            while (urls.hasMoreElements()) {
                java.net.URL url = urls.nextElement(); // 获
取配置文件路径
                try {
                    BufferedReader reader = new
BufferedReader(new InputStreamReader(url.openStream(),
"utf-8"));
                    try {
                        String line = null;
                        while ((line = reader.readLine())
!= null) { // 一行一行读取(一行一个配置)
                            final int ci =
```

```
line.indexOf('#');
                            if (ci >= 0) line =
line.substring(0, ci);
                            line = line.trim();
                            if (line.length() > 0) {
                                try {
                                    String name = null;
                                    int i =
line.indexOf('='); // 等号分割
                                    if (i > 0) {
                                        name =
line.substring(0, i).trim(); // 扩展名称
                                        line =
line.substring(i + 1).trim(); // 扩展实现类
                                    }
                                    if (line.length() > 0)
{
                                        Class<?> clazz =
Class.forName(line, true, classLoader); // 加载扩展实现类
                                        if (!
type.isAssignableFrom(clazz)) { // 判断类型是否匹配
                                            throw new
IllegalStateException("Error when load extension
class(interface: " +
                                                    type +
", class line: " + clazz.getName() + "), class "
                                                    +
clazz.getName() + "is not subtype of interface.");
                                        }
                                        if
(clazz.isAnnotationPresent(Adaptive.class)) { // 判断该实现类
是否@Adaptive,是的话不会放入extensionClasses/cachedClasses缓存

if(cachedAdaptiveClass == null) { // 第一个赋值给
cachedAdaptiveClass属性

cachedAdaptiveClass = clazz;
                                        } else if (!
cachedAdaptiveClass.equals(clazz)) { // 只能有一个@Adaptive实
现,出现第二个就报错了
                                            throw new
```

```
IllegalStateException("More than 1 adaptive class found: "
                                                        +
cachedAdaptiveClass.getClass().getName()
                                                        +
", " + clazz.getClass().getName());
                                                }
                                            } else { // 不是
@Adaptive类型
                                                try {
clazz.getConstructor(type); // 判断是否Wrapper类型
                                                    Set<Class<?
>> wrappers = cachedWrapperClasses;
                                                    if
(wrappers == null) {
cachedWrapperClasses = new ConcurrentHashSet<Class<?>>();
wrappers = cachedWrapperClasses;
                                                    }
wrappers.add(clazz); //放入到Wrapper实现类缓存中
                                                } catch
(NoSuchMethodException e) { //不是Wrapper类型，普通实现类型
clazz.getConstructor();
                                                    if (name ==
null || name.length() == 0) {
                                                        name =
findAnnotationName(clazz);
                                                        if
(name == null || name.length() == 0) {
                                                            if
(clazz.getSimpleName().length() >
type.getSimpleName().length()
&& clazz.getSimpleName().endsWith(type.getSimpleName())) {
name = clazz.getSimpleName().substring(0,
clazz.getSimpleName().length() -
type.getSimpleName().length()).toLowerCase();
```

```
                                                    }
else {

throw new IllegalStateException("No such extension name for
the class " + clazz.getName() + " in the config " + url);
                                                    }
                                                }
                                            }
                                            String[]
names = NAME_SEPARATOR.split(name); // 看是否配置了多个name
                                            if (names
!= null && names.length > 0) {

Activate activate = clazz.getAnnotation(Activate.class); //
是否@Activate类型
                                                if
(activate != null) {

cachedActivates.put(names[0], activate);// 是则放入
cachedActivates缓存
                                                }


                                                // 遍历
所有name
                                                for
(String n : names) {
                                                    if
(! cachedNames.containsKey(clazz)) {

cachedNames.put(clazz, n); // 放入Extension实现类与名称映射缓
存,每个class只对应第一个名称有效
                                                    }


Class<?> c = extensionClasses.get(n);
                                                    if
(c == null) {

extensionClasses.put(n, clazz); // 放入到extensionClasses缓
存,多个name可能对应一个Class
                                                    }
else if (c != clazz) { // 存在重名
```

```
throw new IllegalStateException("Duplicate extension " +
type.getName() + " name " + n + " on " + c.getName() + "
and " + clazz.getName());
                                                                }
                                                            }
                                                        }
                                                    }
                                                }
                                            }
                                        } catch (Throwable t) {
                                            IllegalStateException e
= new IllegalStateException("Failed to load extension
class(interface: " + type + ", class line: " + line + ") in
" + url + ", cause: " + t.getMessage(), t);
                                            exceptions.put(line,
e);
                                        }
                                    }
                                } // end of while read lines
                            } finally {
                                reader.close();
                            }
                        } catch (Throwable t) {
                            logger.error("Exception when load
extension class(interface: " +
                                            type + ", class
file: " + url + ") in " + url, t);
                        }
                    } // end of while urls
                }
            } catch (Throwable t) {
                logger.error("Exception when load extension
class(interface: " +
                        type + ", description file: " + fileName +
").", t);
            }
        }
```

- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73
- 74
- 75
- 76
- 77
- 78
- 79
- 80
- 81
- 82
- 83
- 84
- 85

- 86
- 87
- 88
- 89
- 90
- 91
- 92
- 93
- 94
- 95
- 96
- 97
- 98
- 99
- 100
- 101
- 102
- 103
- 104
- 105
- 106
- 107
- 108
- 109
- 110
- 111
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - 7
  - 8
  - 9
  - 10
  - 11
  - 12
  - 13
  - 14
  - 15

- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56

- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73
- 74
- 75
- 76
- 77
- 78
- 79
- 80
- 81
- 82
- 83
- 84
- 85
- 86
- 87
- 88
- 89
- 90
- 91
- 92
- 93
- 94
- 95
- 96
- 97

- 98
- 99
- 100
- 101
- 102
- 103
- 104
- 105
- 106
- 107
- 108
- 109
- 110
- 111

代码比较长，大概的事情呢就是解析配置文件，获取扩展点实现对应的名称和实现类，并进行分类处理和缓存。当`loadFile`方法执行完成之后，以下几个变量就会被附上值：

- cachedAdaptiveClass : 当前Extension类型对应的AdaptiveExtension类型(只能一个)
- cachedWrapperClasses : 当前Extension类型对应的所有Wrapper实现类型(无顺序)
- cachedActivates : 当前Extension实现自动激活实现缓存(map,无序)
- cachedNames : 扩展点实现类对应的名称(如配置多个名称则值为第一个)

当`loadExtensionClasses`方法执行完成之后，还有一下变量被赋值：

- cachedDefaultName : 当前扩展点的默认实现名称

当`getExtensionClasses`方法执行完成之后，除了上述变量被赋值之外，还有以下变量被赋值：

- cachedClasses : 扩展点实现名称对应的实现类(一个实现类可能有多个名称)

其实也就是说，在调用了`getExtensionClasses`方法之后，当前扩展点对应的实现类的一些信息就已经加载进来了并且被缓存了。后面的许多操作都可以直接通过这些缓存数据来进行处理了。

回到`createAdaptiveExtension`方法，他调用了`getExtesionClasses`方法加载扩展点实现信息完成之后，就可以直接通过判断`cachedAdaptiveClass`缓存字段是否被赋值盘确定当前扩展点是否有默认的AdaptiveExtension实现。如果没有，

那么就调用`createAdaptiveExtensionClass`方法来动态生成一个。在dubbo的扩展点框架中大量的使用了缓存技术。

创建自适应扩展点实现类型和实例化就已经完成了，下面就来看下扩展点自动注入的实现`injectExtension`：

```
private T injectExtension(T instance) {
    try {
        if (objectFactory != null) {
            for (Method method :
instance.getClass().getMethods()) {
                if (method.getName().startsWith("set")
                        &&
method.getParameterTypes().length == 1
                        &&
Modifier.isPublic(method.getModifiers())) {// 处理所有set方法
                    Class<?> pt =
method.getParameterTypes()[0];// 获取set方法参数类型
                    try {
                        // 获取setter对应的property名称
                        String property =
method.getName().length() > 3 ?
method.getName().substring(3, 4).toLowerCase() +
method.getName().substring(4) : "";
                        Object object =
objectFactory.getExtension(pt, property); // 根据类型，名称信息从ExtensionFactory获取
                        if (object != null) { // 如果不为空，说set方法的参数是扩展点类型，那么进行注入
                            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;
    }
```

- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27

这里可以看到，扩展点自动注入的一句就是根据setter方法对应的参数类型和property名称从`ExtensionFactory`中查询，如果有返回扩展点实例，那么就进行注入操作。到这里`getAdaptiveExtension`方法就分析完毕了。

getExtension

这个方法的主要作用是用来获取`ExtensionLoader`实例代表的扩展的指定实现。已扩展实现的名字作为参数，结合前面学习`getAdaptiveExtension`的代码，我们可以推测，这方法中也使用了在调用`getExtensionClasses`方法的时候收集并缓存的数据，其中涉及到名字和具体实现类型对应关系的缓存属性是`cachedClasses`。具体是是否如我们猜想的那样呢，学习一下相关代码就知道了：

```
public T getExtension(String name) {
    if (name == null || name.length() == 0)
        throw new IllegalArgumentException("Extension name
== null");
    if ("true".equals(name)) {  // 判断是否是获取默认实现
        return getDefaultExtension();
    }
    Holder<Object> holder = cachedInstances.get(name);// 缓
存
    if (holder == null) {
        cachedInstances.putIfAbsent(name, new
```

```
Holder<Object>());
        holder = cachedInstances.get(name);
    }
    Object instance = holder.get();
    if (instance == null) {
        synchronized (holder) {
            instance = holder.get();
            if (instance == null) {
                instance = createExtension(name);// 没有缓存
实例则创建
                holder.set(instance);// 缓存起来
            }
        }
    }
    return (T) instance;
}
```

接着看`createExtension`方法的实现：

```
private T createExtension(String name) {
    Class<?> clazz = getExtensionClasses().get(name); //
getExtensionClass内部使用cachedClasses缓存
    if (clazz == null) {
        throw findException(name);
    }
    try {
        T instance = (T) EXTENSION_INSTANCES.get(clazz); //
从已创建Extension实例缓存中获取
        if (instance == null) {
            EXTENSION_INSTANCES.putIfAbsent(clazz, (T)
clazz.newInstance());
            instance = (T) EXTENSION_INSTANCES.get(clazz);
        }
        injectExtension(instance); // 属性注入

        // Wrapper类型进行包装，层层包裹
        Set<Class<?>> wrapperClasses =
cachedWrapperClasses;
```

```
        if (wrapperClasses != null && wrapperClasses.size()
> 0) {
            for (Class<?> wrapperClass : wrapperClasses) {
                instance = injectExtension((T)
wrapperClass.getConstructor(type).newInstance(instance));
            }
        }
        return instance;
    } catch (Throwable t) {
        throw new IllegalStateException("Extension
instance(name: " + name + ", class: " +
                type + ")  could not be instantiated: " +
t.getMessage(), t);
    }
}
```

从代码中可以看到，内部调用了`getExtensionClasses`方法来获取当前扩展的所有实现，而`getExtensionClassse`方法会在第一次被调用的时候将结果缓存到`cachedClasses`变量中，后面的调用就直接从缓存变量中获取了。这里还可以看到一个缓存`EXTENSION_INSTANCES`，这个缓存是`ExtensionLoader`的静态成员，也就是全局缓存，存放着所有的扩展点实现类型与其对应的已经实例化的实例对象(是所有扩展点，不是某一个扩展点)，也就是说所有的扩展点实现在dubbo中最多都只会有一个实例。

拿到扩展点实现类型对应的实例之后，调用了`injectExtension`方法对该实例进行扩展点注入，紧接着就是遍历该扩展点接口的所有Wrapper来对真正的扩展点实例进行Wrap操作，都是对通过将上一次的结果作为下一个Wrapper的构造函数参数传递进去实例化一个Wrapper对象，最后总返回回去的是Wrapper类型的

实例而不是具体实现类的实例。

这里或许有一个疑问： 从代码中看，不论`instance`是否存在于`EXTENSION_INSTANCE`，都会进行扩展点注入和Wrap操作。那么如果对于同一个扩展点，调用了两次`createExtension`方法的话，那不就进行了两次Wrap操作么?

如果外部能够直接调用`createExtension`方法，那么确实可能出现这个问题。但是由于`createExtension`方法是`private`的，因此外部无法直接调用。而在`ExtensionLoader`类中调用它的`getExtension`方法(只有它这一处调用)，内部自己做了缓存(`cachedInstances`)，因此当`getExtension`方法内部调用了一次`createExtension`方法之后，后面对`getExtension`方法执行同样的调用时，会直接使用`cachedInstances`缓存而不会再去调用`createExtension`方法了。

getActivateExtension

`getActivateExtension`方法主要获取当前扩展的所有可自动激活的实现。可根据入参(values)调整指定实现的顺序，在这个方法里面也使用到`getExtensionClasses`方法中收集的缓存数据。

```
public List<T> getActivateExtension(URL url, String[] values, String group) {
    List<T> exts = new ArrayList<T>();
    List<String> names = values == null ? new ArrayList<String>(0) : Arrays.asList(values); // 解析配置要使用的名称

    // 如果未配置"-default",则加载所有Activates扩展(names指定的扩展)
    if (! names.contains(Constants.REMOVE_VALUE_PREFIX + Constants.DEFAULT_KEY)) {
        getExtensionClasses(); // 加载当前Extension所有实现,会获取到当前Extension中所有@Active实现，赋值给cachedActivates变量
        for (Map.Entry<String, Activate> entry : cachedActivates.entrySet()) { // 遍历当前扩展所有的@Activate扩展
            String name = entry.getKey();
            Activate activate = entry.getValue();
            if (isMatchGroup(group, activate.group())) { // 判断group是否满足,group为null则直接返回true
                T ext = getExtension(name); // 获取扩展示例
```

```
                // 排除names指定的扩展;并且如果names中没有指定移除该扩展(-name)，且当前url匹配结果显示可激活才进行使用
                if (! names.contains(name)
                        && !
names.contains(Constants.REMOVE_VALUE_PREFIX + name)
                        && isActive(activate, url)) {
                    exts.add(ext);
                }
            }
        }
        Collections.sort(exts,
ActivateComparator.COMPARATOR); // 默认排序
    }

    // 对names指定的扩展进行专门的处理
    List<T> usrs = new ArrayList<T>();
    for (int i = 0; i < names.size(); i ++) { // 遍历names指定的扩展名
        String name = names.get(i);
        if (!
name.startsWith(Constants.REMOVE_VALUE_PREFIX)
                && !
names.contains(Constants.REMOVE_VALUE_PREFIX + name)) { // 未设置移除该扩展
            if (Constants.DEFAULT_KEY.equals(name)) { // default表示上面已经加载并且排序的exts,将排在default之前的Activate扩展放置到default组之前,例如:ext1,default,ext2
                if (usrs.size() > 0) { // 如果此时user不为空，则user中存放的是配置在default之前的Activate扩展
                    exts.addAll(0, usrs); // 注意index是0，放在default前面
                    usrs.clear(); // 放到default之前，然后清空
                }
            } else {
                T ext = getExtension(name);
                usrs.add(ext);
            }
        }
    }
    if (usrs.size() > 0) { // 这里留下的都是配置在default之后的
```

```
            exts.addAll(usrs); // 添加到default排序之后
        }
        return exts;
    }
```

- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
    - 1
    - 2
    - 3
    - 4
    - 5
    - 6
    - 7
    - 8
    - 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32

总结

基本上将dubbo的扩展点加载机制学习了一遍，有几点可能需要注意的地方：

- 每个`ExtensionLoader`实例只负责加载一个特定扩展点实现
- 每个扩展点对应最多只有一个`ExtensionLoader`实例
- 对于每个扩展点实现，最多只会有一个实例
- 一个扩展点实现可以对应多个名称(逗号分隔)
- 对于需要等到运行时才能决定使用哪一个具体实现的扩展点，应获取其自使用扩展点实现(AdaptiveExtension)
- `@Adaptive`注解要么注释在扩展点`@SPI`的方法上，要么注释在其实现类的类定义上
- 如果`@Adaptive`注解注释在`@SPI`接口的方法上，那么原则上该接口所有方法都应该加`@Adaptive`注解(自动生成的实现中默认为注解的方法抛异常)
- 每个扩展点最多只能有一个被AdaptiveExtension
- 每个扩展点可以有多个可自动激活的扩展点实现(使用`@Activate`注解)
- 由于每个扩展点实现最多只有一个实例，因此扩展点实现应保证线程安全
- 如果扩展点有多个Wrapper，那么最终其执行的顺序不确定(内部使用`ConcurrentHashSet`存储)

TODO：

- 学习一下动态生成`AdaptiveExtension`类的实现过程

官方文档描述动态生成的`AdaptiveExtension`代码如下：

```
package <扩展点接口所在包>;

public class <扩展点接口名>$Adpative implements <扩展点接口> {
    public <有@Adaptive注解的接口方法>(<方法参数>) {
        if(是否有URL类型方法参数?) 使用该URL参数
        else if(是否有方法类型上有URL属性) 使用该URL属性
        # <else 在加载扩展点生成自适应扩展点类时抛异常，即加载扩展点失败！>

        if(获取的URL == null) {
            throw new IllegalArgumentException("url == null");
        }

        根据@Adaptive注解上声明的Key的顺序，从URL获致Value，作为实际扩展点名。
        如URL没有Value，则使用缺省扩展点实现。如没有扩展点， throw new IllegalStateException("Fail to get extension");

        在扩展点实现调用该方法，并返回结果。
    }

    public <有@Adaptive注解的接口方法>(<方法参数>) {
        throw new UnsupportedOperationException("is not adaptive method!");
    }
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13

规则如下：

- 先在URL上找`@Adaptive`注解指定的Extension名；
- 如果不设置则缺省使用Extension接口类名的点分隔小写字串(即对于Extension接口com.alibaba.dubbo.xxx.YyyInvokerWrapper的缺省值为String[] {“yyy.invoker.wrapper”})。
- 使用默认实现(@SPI指定)，如果没有设定缺省扩展，则方法调用会抛出IllegalStateException。

1