title: 精尽 Dubbo 源码分析 —— 拓展机制 SPI date: 2018-03-04 tags: categories: Dubbo permalink: Dubbo/spi

---

摘要: 原创出处 http://www.iocoder.cn/Dubbo/spi/ 「芋道源码」欢迎转载，保留摘要，谢谢！

- 1. 概述
- 2. 改进
- 3. 代码结构
- 4. ExtensionLoader
  - 4.1 属性
  - 4.2 获得拓展配置
  - 4.3 获得拓展加载器
  - 4.4 获得指定拓展对象
  - 4.5 获得自适应的拓展对象
  - 4.6 获得激活的拓展对象数组
- 5. @SPI
- 6. @Adaptive
- 7. @Activate
- 8. ExtensionFactory
  - 8.1 AdaptiveExtensionFactory
  - 8.2 SpiExtensionFactory
  - 8.3 SpringExtensionFactory
- 666. 彩蛋

> 🙂🙂🙂关注微信公众号：**【芋道源码】**有福利：

> 1. RocketMQ / MyCAT / Sharding-JDBC **所有**源码分析文章列表
> 2. RocketMQ / MyCAT / Sharding-JDBC **中文注释源码 GitHub 地址**
> 3. 您对于源码的疑问每条留言**都**将得到**认真**回复。**甚至不知道如何读源码也可以请教噢**。
> 4. **新的**源码解析文章**实时**收到通知。**每周更新一篇左右**。
> 5. **认真的**源码交流微信群。

---

1. 概述

本文主要分享 **Dubbo 的拓展机制 SPI**。

想要理解 Dubbo ， 理解 Dubbo SPI 是非常必须的。在 Dubbo 中，提供了大量的**拓展点**，基于 Dubbo SPI 机制加载。如下图所示：

5.8 合并结果扩展

5.9 注册中心扩展

5.10 监控中心扩展

5.11 扩展点加载扩展

5.12 动态代理扩展

5.13 编译器扩展

5.14 消息派发扩展

5.15 线程池扩展

5.16 序列化扩展

5.17 网络传输扩展

5.18 信息交换扩展

5.19 组网扩展

2. 改进

在看具体的 Dubbo SPI 实现之前，我们先理解 Dubbo SPI 产生的背景：

> FROM 《Dubbo 开发指南 —— 拓展点加载》
>
> 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：
>
> 1. 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。
> 2. 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。
>
> 3. 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。

- Dubbo 自己实现了一套 SPI 机制，而不是使用 Java 标准的 SPI 。
- 第一点问题，Dubbo 有很多的拓展点，例如 Protocol、Filter 等等。并且每个拓展点有多种的实现，例如 Protocol 有 DubboProtocol、InjvmProtocol、RestProtocol 等等。那么使用 JDK SPI 机制，会初始化无用的拓展点及其实现，造成不必要的耗时与资源浪费。
  - 如果无法理解的胖友，跟着 《Java SPI(Service Provider Interface)简介》 文章，**写多个拓展实现**，就很容易理解了。🙂 这就是概念呀。
- 第二点问题，【TODO 8009】ScriptEngine 没看明白，不影响本文理解。
- 第三点问题，严格来说，这不算问题，**而是增加了功能特性**，在下文我们会看到。

3. 代码结构

Dubbo SPI 在 `dubbo-common` 的 `extension` 包实现，如下图所示：

4. ExtensionLoader

`com.alibaba.dubbo.common.extension.ExtensionLoader`，拓展加载器。这是 Dubbo SPI 的核心。

4.1 属性

```
  1: private static final String SERVICES_DIRECTORY = "META-INF/services/";
  2:
  3: private static final String DUBBO_DIRECTORY = "META-INF/dubbo/";
  4:
  5: private static final String DUBBO_INTERNAL_DIRECTORY = DUBBO_DIRECTORY + "inte
rnal/";
  6:
  7: private static final Pattern NAME_SEPARATOR = Pattern.compile("\\s*[,]+\\s*");
  8:
  9: // ============================== 静态属性 ==============================
 10:
 11: /**
 12:  * 拓展加载器集合
 13:  *
 14:  * key：拓展接口
 15:  */
 16: private static final ConcurrentMap<Class<?>, ExtensionLoader<?>> EXTENSION_LOA
DERS = new ConcurrentHashMap<Class<?>, ExtensionLoader<?>>();
 17: /**
 18:  * 拓展实现类集合
 19:  *
 20:  * key：拓展实现类
 21:  * value：拓展对象。
 22:  *
 23:  * 例如，key 为 Class<AccessLogFilter>
 24:  *  value 为 AccessLogFilter 对象
 25:  */
 26: private static final ConcurrentMap<Class<?>, Object> EXTENSION_INSTANCES = new
 ConcurrentHashMap<Class<?>, Object>();
 27:
 28: // ============================== 对象属性 ==============================
 29:
 30: /**
 31:  * 拓展接口。
 32:  * 例如，Protocol
 33:  */
 34: private final Class<?> type;
 35: /**
 36:  * 对象工厂
 37:  *
```

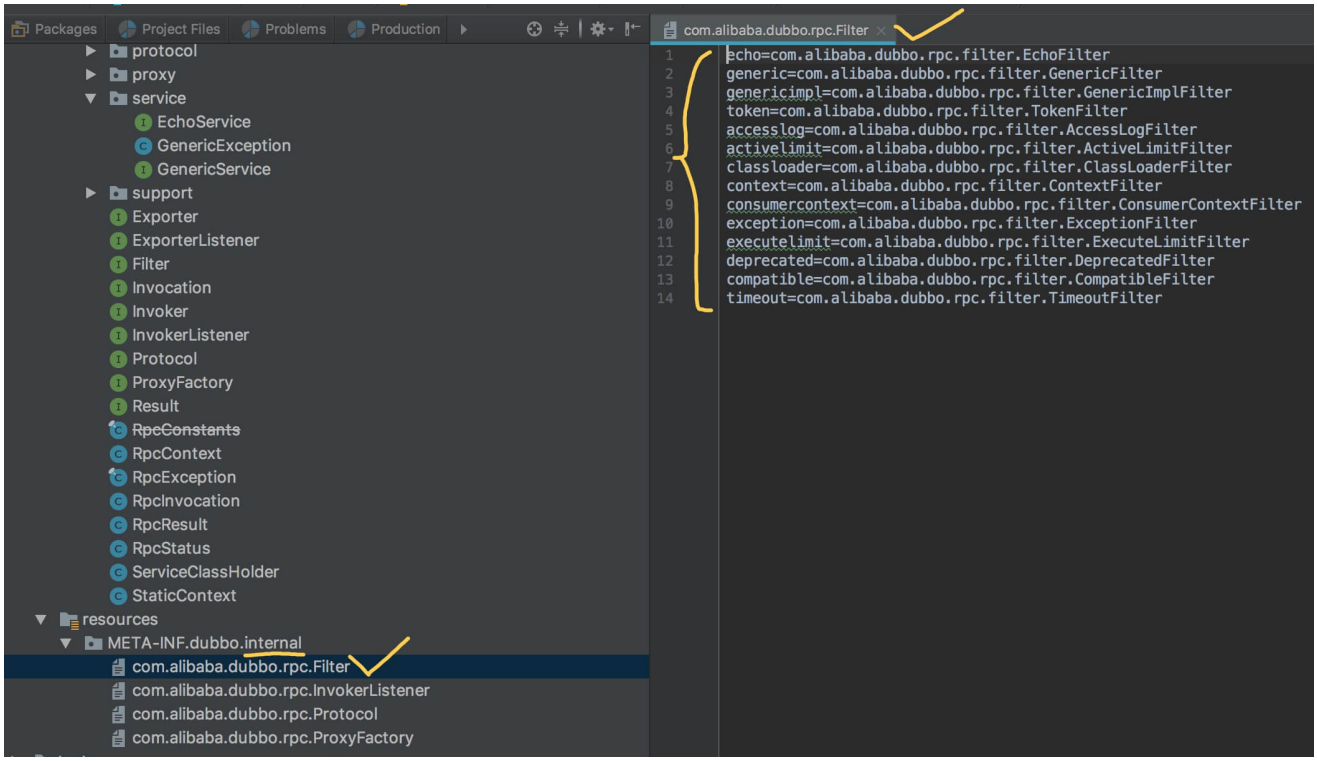
```
 38:  * 用于调用 {@link #injectExtension(Object)} 方法，向拓展对象注入依赖属性。
 39:  *
 40:  * 例如，StubProxyFactoryWrapper 中有 `Protocol protocol` 属性。
 41:  */
 42: private final ExtensionFactory objectFactory;
 43: /**
 44:  * 缓存的拓展名与拓展类的映射。
 45:  *
 46:  * 和 {@link #cachedClasses} 的 KV 对调。
 47:  *
 48:  * 通过 {@link #loadExtensionClasses} 加载
 49:  */
 50: private final ConcurrentMap<Class<?>, String> cachedNames = new ConcurrentHash
Map<Class<?>, String>();
 51: /**
 52:  * 缓存的拓展实现类集合。
 53:  *
 54:  * 不包含如下两种类型：
 55:  *  1. 自适应拓展实现类。例如 AdaptiveExtensionFactory
 56:  *  2. 带唯一参数为拓展接口的构造方法的实现类，或者说拓展 Wrapper 实现类。例如，Prot
ocolFilterWrapper 。
 57:  *   拓展 Wrapper 实现类，会添加到 {@link #cachedWrapperClasses} 中
 58:  *
 59:  * 通过 {@link #loadExtensionClasses} 加载
 60:  */
 61: private final Holder<Map<String, Class<?>>> cachedClasses = new Holder<Map<Str
ing, Class<?>>>();
 62:
 63: /**
 64:  * 拓展名与 @Activate 的映射
 65:  *
 66:  * 例如，AccessLogFilter。
 67:  *
 68:  * 用于 {@link #getActivateExtension(URL, String)}
 69:  */
 70: private final Map<String, Activate> cachedActivates = new ConcurrentHashMap<St
ring, Activate>();
 71: /**
 72:  * 缓存的拓展对象集合
 73:  *
 74:  * key：拓展名
 75:  * value：拓展对象
 76:  *
 77:  * 例如，Protocol 拓展
 78:  *      key：dubbo value：DubboProtocol
 79:  *      key：injvm value：InjvmProtocol
 80:  *
 81:  * 通过 {@link #loadExtensionClasses} 加载
```

```
 82:  */
 83: private final ConcurrentMap<String, Holder<Object>> cachedInstances = new Conc
urrentHashMap<String, Holder<Object>>();
 84: /**
 85:  * 缓存的自适应( Adaptive )拓展对象
 86:  */
 87: private final Holder<Object> cachedAdaptiveInstance = new Holder<Object>();
 88: /**
 89:  * 缓存的自适应拓展对象的类
 90:  *
 91:  * {@link #getAdaptiveExtensionClass()}
 92:  */
 93: private volatile Class<?> cachedAdaptiveClass = null;
 94: /**
 95:  * 缓存的默认拓展名
 96:  *
 97:  * 通过 {@link SPI} 注解获得
 98:  */
 99: private String cachedDefaultName;
100: /**
101:  * 创建 {@link #cachedAdaptiveInstance} 时发生的异常。
102:  *
103:  * 发生异常后，不再创建，参见 {@link #createAdaptiveExtension()}
104:  */
105: private volatile Throwable createAdaptiveInstanceError;
106:
107: /**
108:  * 拓展 Wrapper 实现类集合
109:  *
110:  * 带唯一参数为拓展接口的构造方法的实现类
111:  *
112:  * 通过 {@link #loadExtensionClasses} 加载
113:  */
114: private Set<Class<?>> cachedWrapperClasses;
115:
116: /**
117:  * 拓展名 与 加载对应拓展类发生的异常 的 映射
118:  *
119:  * key：拓展名
120:  * value：异常
121:  *
122:  * 在 {@link #loadFile(Map, String)} 时，记录
123:  */
124: private Map<String, IllegalStateException> exceptions = new ConcurrentHashMap<
String, IllegalStateException>();
```

- 第 1 至 5 行：在 `META-INF/dubbo/internal/` 和 `META-INF/dubbo/` 目录下，放置 **接口全限**

定名 配置文件，**每行**内容为：**拓展名=拓展实现类全限定名**。

- `META-INF/dubbo/internal/` 目录下，从名字上可以看出，用于 Dubbo **内部**提供的拓展实现。下图是一个例子：



- `META-INF/dubbo/` 目录下，用于用户**自定义**的拓展实现。
- `META-INF/service/` 目录下，Java SPI 的配置目录。在 「4.2 加载拓展配置」 中，我们会看到 Dubbo SPI 对 Java SPI 做了**兼容**。

- 第 7 行：`NAME_SEPARATOR` ，拓展名分隔符，使用**逗号**。
- **第 9 至 124 行**，我们将属性分成了两类：1）静态属性；2）对象属性。这是为啥呢?
  - 【静态属性】一方面，ExtensionLoader 是 ExtensionLoader 的**管理容器**。一个拓展(拓展接口)对应一个 ExtensionLoader 对象。例如，Protocol 和 Filter **分别**对应一个 ExtensionLoader 对象。
  - 【对象属性】另一方面，一个拓展通过其 ExtensionLoader 对象，加载它的**拓展实现们**。我们会发现多个属性都是 "**cached**" 开头。ExtensionLoader 考虑到性能和资源的优化，读取拓展配置后，会首先进行**缓存**。等到 Dubbo 代码**真正**用到对应的拓展实现时，进行拓展实现的对象的初始化。并且，初始化完成后，也会进行**缓存**。也就是说：
    - 缓存加载的拓展配置
    - 缓存创建的拓展实现对象
- 🙂 胖友先看下属性的代码注释，有一个整体的印象。下面我们在读实现代码时，会进一步解析说明。

考虑到胖友能更好的理解下面的代码实现，推荐先阅读下 《Dubbo 开发指南 —— 扩展点加载》文档，建立下对 ExtensionLoader 特点的初步理解：

- 扩展点自动包装

    - 在「4.4.2 createExtension」详细解析。
- 扩展点自动装配
    - 在「4.4.3 injectExtension」详细解析。
- 扩展点自适应
    - 在「4.5 获得自适应的拓展对象」详细解析。
- 扩展点自动激活
    - 在「4.6 获得激活的拓展对象数组」详细解析。

4.2 获得拓展配置

4.2.1 getExtensionClasses

`#getExtensionClasses()` 方法，获得拓展实现类数组。

```
private final Holder<Map<String, Class<?>>> cachedClasses = new Holder<Map<String,
Class<?>>>();

private volatile Class<?> cachedAdaptiveClass = null;

private Set<Class<?>> cachedWrapperClasses;
```

```
 1: /**
 2:  * 获得拓展实现类数组
 3:  *
 4:  * @return 拓展实现类数组
 5:  */
 6: private Map<String, Class<?>> getExtensionClasses() {
 7:     // 从缓存中，获得拓展实现类数组
 8:     Map<String, Class<?>> classes = cachedClasses.get();
 9:     if (classes == null) {
10:         synchronized (cachedClasses) {
11:             classes = cachedClasses.get();
12:             if (classes == null) {
13:                 // 从配置文件中，加载拓展实现类数组
14:                 classes = loadExtensionClasses();
15:                 // 设置到缓存中
16:                 cachedClasses.set(classes);
17:             }
18:         }
19:     }
20:     return classes;
21: }
```

- `cachedClasses` 属性，缓存的拓展实现类集合。它不包含如下两种类型的拓展实现：
  - **自适应**拓展实现类。例如 AdaptiveExtensionFactory 。
    - 拓展 Adaptive 实现类，会添加到 `cachedAdaptiveClass` 属性中。
  - 带**唯一参数为拓展接口**的构造方法的实现类，或者说拓展 Wrapper 实现类。例如，ProtocolFilterWrapper 。
    - 拓展 Wrapper 实现类，会添加到 `cachedWrapperClasses` 属性中。
  - 总结来说，`cachedClasses` + `cachedAdaptiveClass` + `cachedWrapperClasses` 才是**完整**缓存的拓展实现类的配置。
- 第 7 至 11 行：从缓存中，获得拓展实现类数组。
- 第 12 至 14 行：当缓存不存在时，调用 `#loadExtensionClasses()` 方法，从配置文件中，加载拓展实现类数组。
- 第 16 行：设置加载的实现类数组，到缓存中。

4.2.2 loadExtensionClasses

`#loadExtensionClasses()` 方法，从**多个**配置文件中，加载拓展实现类数组。

```
 1: /**
 2:  * 加载拓展实现类数组
 3:  *
 4:  * 无需声明 synchronized ，因为唯一调用该方法的 {@link #getExtensionClasses()} 已经声明。
 5:  * // synchronized in getExtensionClasses
 6:  *
 7:  * @return 拓展实现类数组
 8:  */
 9: private Map<String, Class<?>> loadExtensionClasses() {
10:     // 通过 @SPI 注解，获得默认的拓展实现类名
11:     final SPI defaultAnnotation = type.getAnnotation(SPI.class);
12:     if (defaultAnnotation != null) {
13:         String value = defaultAnnotation.value();
14:         if ((value = value.trim()).length() > 0) {
15:             String[] names = NAME_SEPARATOR.split(value);
16:             if (names.length > 1) {
17:                 throw new IllegalStateException("more than 1 default extension name on extension " + type.getName()
18:                         + ": " + Arrays.toString(names));
19:             }
20:             if (names.length == 1) cachedDefaultName = names[0];
21:         }
22:     }
23:
24:     // 从配置文件中，加载拓展实现类数组
25:     Map<String, Class<?>> extensionClasses = new HashMap<String, Class<?>>();
```

```
26:     loadFile(extensionClasses, DUBBO_INTERNAL_DIRECTORY);
27:     loadFile(extensionClasses, DUBBO_DIRECTORY);
28:     loadFile(extensionClasses, SERVICES_DIRECTORY);
29:     return extensionClasses;
30: }
```

- 第 10 至 22 行：通过 `@SPI` 注解，获得拓展接口对应的**默认的**拓展实现类名。在 「5. @SPI」 详细解析。
- 第 25 至 29 行：调用 `#loadFile(extensionClasses, dir)` 方法，从配置文件中，加载拓展实现类数组。**注意**，此处配置文件的加载顺序。

4.2.3 loadFile

`#loadFile(extensionClasses, dir)` 方法，从**一个**配置文件中，加载拓展实现类数组。代码如下：

```
/**
 * 缓存的自适应拓展对象的类
 *
 * {@link #getAdaptiveExtensionClass()}
 */
private volatile Class<?> cachedAdaptiveClass = null;

/**
 * 拓展 Wrapper 实现类集合
 *
 * 带唯一参数为拓展接口的构造方法的实现类
 *
 * 通过 {@link #loadExtensionClasses} 加载
 */
private Set<Class<?>> cachedWrapperClasses;

/**
 * 拓展名与 @Activate 的映射
 *
 * 例如，AccessLogFilter。
 *
 * 用于 {@link #getActivateExtension(URL, String)}
 */
private final Map<String, Activate> cachedActivates = new ConcurrentHashMap<String,
 Activate>();

/**
 * 缓存的拓展名与拓展类的映射。
 *
```

```
     * 和 {@link #cachedClasses} 的 KV 对调。
     *
     * 通过 {@link #loadExtensionClasses} 加载
     */
    private final ConcurrentMap<Class<?>, String> cachedNames = new ConcurrentHashMap<C
lass<?>, String>();

    /**
     * 拓展名 与 加载对应拓展类发生的异常 的 映射
     *
     * key：拓展名
     * value：异常
     *
     * 在 {@link #loadFile(Map, String)} 时，记录
     */
    private Map<String, IllegalStateException> exceptions = new ConcurrentHashMap<Strin
g, IllegalStateException>();
```

```
 1: /**
 2:  * 从一个配置文件中，加载拓展实现类数组。
 3:  *
 4:  * @param extensionClasses 拓展类名数组
 5:  * @param dir 文件名
 6:  */
 7: private void loadFile(Map<String, Class<?>> extensionClasses, String dir) {
 8:     // 完整的文件名
 9:     String fileName = dir + type.getName();
10:     try {
11:         Enumeration<java.net.URL> urls;
12:         // 获得文件名对应的所有文件数组
13:         ClassLoader classLoader = findClassLoader();
14:         if (classLoader != null) {
15:             urls = classLoader.getResources(fileName);
16:         } else {
17:             urls = ClassLoader.getSystemResources(fileName);
18:         }
19:         // 遍历文件数组
20:         if (urls != null) {
21:             while (urls.hasMoreElements()) {
22:                 java.net.URL url = urls.nextElement();
23:                 try {
24:                     BufferedReader reader = new BufferedReader(new InputStream
Reader(url.openStream(), "utf-8"));
25:                     try {
26:                         String line;
27:                         while ((line = reader.readLine()) != null) {
28:                             // 跳过当前被注释掉的情况，例如 #spring=xxxxxxxxx
29:                             final int ci = line.indexOf('#');
```

```
30:                             if (ci >= 0) line = line.substring(0, ci);
31:                             line = line.trim();
32:                             if (line.length() > 0) {
33:                                 try {
34:                                     // 拆分，key=value 的配置格式
35:                                     String name = null;
36:                                     int i = line.indexOf('=');
37:                                     if (i > 0) {
38:                                         name = line.substring(0, i).trim();
39:                                         line = line.substring(i + 1).trim();
40:                                     }
41:                                     if (line.length() > 0) {
42:                                         // 判断拓展实现，是否实现拓展接口
43:                                         Class<?> clazz = Class.forName(line, t
rue, classLoader);
44:                                         if (!type.isAssignableFrom(clazz)) {
45:                                             throw new IllegalStateException("E
rror when load extension class(interface: " +
46:                                                     type + ", class line: " +
clazz.getName() + "), class "
47:                                                     + clazz.getName() + "is no
t subtype of interface.");
48:                                         }
49:                                         // 缓存自适应拓展对象的类到 `cachedAdapt
iveClass`
50:                                         if (clazz.isAnnotationPresent(Adaptive
.class)) {
51:                                             if (cachedAdaptiveClass == null) {
52:                                                 cachedAdaptiveClass = clazz;
53:                                             } else if (!cachedAdaptiveClass.eq
uals(clazz)) {
54:                                                 throw new IllegalStateExceptio
n("More than 1 adaptive class found: "
55:                                                         + cachedAdaptiveClass.
getClass().getName()
56:                                                         + ", " + clazz.getClas
s().getName());
57:                                             }
58:                                         } else {
59:                                             // 缓存拓展 Wrapper 实现类到 `cached
WrapperClasses`
60:                                             try {
61:                                                 clazz.getConstructor(type);
62:                                                 Set<Class<?>> wrappers = cache
dWrapperClasses;
63:                                                 if (wrappers == null) {
64:                                                     cachedWrapperClasses = new
 ConcurrentHashSet<Class<?>>();
```

```
 65:                                                 wrappers = cachedWrapperCl
asses;
 66:                                             }
 67:                                             wrappers.add(clazz);
 68:                                         // 缓存拓展实现类到 `extensionClasse
s`
 69:                                         } catch (NoSuchMethodException e)
{
 70:                                             clazz.getConstructor();
 71:                                             // 未配置拓展名，自动生成。例如，
DemoFilter 为 demo 。主要用于兼容 Java SPI 的配置。
 72:                                             if (name == null || name.lengt
h() == 0) {
 73:                                                 name = findAnnotationName(
clazz);
 74:                                                 if (name == null || name.l
ength() == 0) {
 75:                                                     if (clazz.getSimpleNam
e().length() > type.getSimpleName().length()
 76:                                                             && clazz.getSi
mpleName().endsWith(type.getSimpleName())) {
 77:                                                         name = clazz.getSi
mpleName().substring(0, clazz.getSimpleName().length() - type.getSimpleName().lengt
h()).toLowerCase();
 78:                                                     } else {
 79:                                                         throw new IllegalS
tateException("No such extension name for the class " + clazz.getName() + " in the
config " + url);
 80:                                                     }
 81:                                                 }
 82:                                             }
 83:                                             // 获得拓展名，可以是数组，有多个
拓展名。
 84:                                             String[] names = NAME_SEPARATO
R.split(name);
 85:                                             if (names != null && names.len
gth > 0) {
 86:                                                 // 缓存 @Activate 到 `cache
dActivates` 。
 87:                                                 Activate activate = clazz.
getAnnotation(Activate.class);
 88:                                                 if (activate != null) {
 89:                                                     cachedActivates.put(na
mes[0], activate);
 90:                                                 }
 91:                                                 for (String n : names) {
 92:                                                     // 缓存到 `cachedNames`
 93:                                                     if (!cachedNames.conta
```

```
insKey(clazz)) {
 94:                                                     cachedNames.put(cl
azz, n);
 95:                                                 }
 96:                                                 // 缓存拓展实现类到 `ext
ensionClasses`
 97:                                                 Class<?> c = extension
Classes.get(n);
 98:                                                 if (c == null) {
 99:                                                     extensionClasses.p
ut(n, clazz);
100:                                                 } else if (c != clazz)
 {
101:                                                     throw new IllegalS
tateException("Duplicate extension " + type.getName() + " name " + n + " on " + c.g
etName() + " and " + clazz.getName());
102:                                                 }
103:                                             }
104:                                         }
105:                                     }
106:                                 }
107:                             }
108:                         } catch (Throwable t) {
109:                             // 发生异常，记录到异常集合
110:                             IllegalStateException e = new IllegalState
Exception("Failed to load extension class(interface: " + type + ", class line: " +
line + ") in " + url + ", cause: " + t.getMessage(), t);
111:                             exceptions.put(line, e);
112:                         }
113:                     }
114:                 } // end of while read lines
115:             } finally {
116:                 reader.close();
117:             }
118:         } catch (Throwable t) {
119:             logger.error("Exception when load extension class(interfac
e: " +
120:                     type + ", class file: " + url + ") in " + url, t);
121:         }
122:     } // end of while urls
123: }
124: } catch (Throwable t) {
125:     logger.error("Exception when load extension class(interface: " +
126:             type + ", description file: " + fileName + ").", t);
127: }
128: }
```

- 第 9 行：获得完整的文件名(相对路径)。例如：`"META-INF/dubbo/internal/com.alibaba.dubbo.common.extension.ExtensionFactory"` 。
- 第 12 至 18 行：获得文件名对应的所有文件 URL 数组。例如：

- 第 21 至 24 行：逐个**文件** URL 遍历。
- 第 27 行：逐**行**遍历。
- 第 29 至 32 行：跳过当前被 `"#"` 注释掉的情况，例如 `#spring=xxxxxxxxx` 。
- 第 34 至 40 行：按照 `key=value` 的配置拆分。其中 `name` 为拓展名， `line` 为拓展实现类名。**注意**，上文我们提到过 Dubbo SPI 会兼容 Java SPI 的配置格式，那么按照此处的解析方式， `name` 会为空。这种情况下，拓展名会自动生成，详细见第 71 至 82 行的代码。
- 第 42 至 48 行：判断拓展实现类，需要实现拓展接口。
- 第 50 至 57 行：缓存自适应拓展对象的类到 `cachedAdaptiveClass` 属性。在 「6. @Adaptive」 详细解析。
- 第 59 至 67 行：缓存拓展 Wrapper 实现类到 `cachedWrapperClasses` 属性。
  - 第 61 行：调用 `Class#getConstructor(Class<?>... parameterTypes)` 方法，通过**反射**的方式，参数为拓展接口，判断当前配置的拓展实现类为**拓展 Wrapper 实现类**。若成功（未抛出异常），则代表符合条件。例如，ProtocolFilterWrapper(Protocol protocol) 这个构造方法。
- 第 69 至 105 行：若获得构造方法失败，则代表是普通的拓展实现类，缓存到 `extensionClasses` **变量**中。
  - 第 70 行：调用 `Class#getConstructor(Class<?>... parameterTypes)` 方法，获得参数为空的构造方法。
  - 第 72 至 82 行：未配置拓展名，自动生成。**适用于 Java SPI 的配置方式**。例如，xxx.yyy.DemoFilter 生成的拓展名为 `demo` 。
    - 第 73 行：通过 `@Extension` 注解的方式设置拓展名的方式已经**废弃**，胖友可以无视该方法。
- 第 84 行：获得拓展名。使用逗号进行分割，即多个拓展名可以对应同一个拓展实现类。
- 第 86 至 90 行：缓存 `@Activate` 到 `cachedActivates` 。在 「7. @Activate」 详细解析。
- 第 93 至 95 行：缓存到 `cachedNames` 属性。
- 第 96 至 102 行：缓存拓展实现类到 `extensionClasses` 变量。**注意**，相同拓展名，不能对应多个不同的拓展实现。
- 第 108 至 112 行：若发生异常，记录到异常集合 `exceptions` 属性。

4.2.4 其他方法

如下方法，和该流程无关，胖友可自行查看。

- `#getExtensionClass(name)`

- `#findException(name)`
- `#getExtensionName(extensionInstance)`
- `#getExtensionName(extensionClass)`
- `#getSupportedExtensions()`
- `#getDefaultExtensionName()`
- `#hasExtension(name)`

4.3 获得拓展加载器

在 Dubbo 的代码里，常常能看到如下的代码：

```
ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(name)
```

4.3.1 getExtensionLoader

`#getExtensionLoader(type)` **静态**方法，根据拓展点的接口，获得拓展加载器。代码如下：

```
/**
 * 拓展加载器集合
 *
 * key：拓展接口
 */
 //  【静态属性】
private static final ConcurrentMap<Class<?>, ExtensionLoader<?>> EXTENSION_LOADERS
= new ConcurrentHashMap<Class<?>, ExtensionLoader<?>>();

  1: /**
  2:  * 根据拓展点的接口，获得拓展加载器
  3:  *
  4:  * @param type 接口
  5:  * @param <T> 泛型
  6:  * @return 加载器
  7:  */
  8: @SuppressWarnings("unchecked")
  9: public static <T> ExtensionLoader<T> getExtensionLoader(Class<T> type) {
 10:     if (type == null)
 11:         throw new IllegalArgumentException("Extension type == null");
 12:     // 必须是接口
 13:     if (!type.isInterface()) {
 14:         throw new IllegalArgumentException("Extension type(" + type + ") is no
t interface!");
 15:     }
```

```
 16:     // 必须包含 @SPI 注解
 17:     if (!withExtensionAnnotation(type)) {
 18:         throw new IllegalArgumentException("Extension type(" + type +
 19:                 ") is not extension, because WITHOUT @" + SPI.class.getSimpleN
ame() + " Annotation!");
 20:     }
 21:
 22:     // 获得接口对应的拓展点加载器
 23:     ExtensionLoader<T> loader = (ExtensionLoader<T>) EXTENSION_LOADERS.get(typ
e);
 24:     if (loader == null) {
 25:         EXTENSION_LOADERS.putIfAbsent(type, new ExtensionLoader<T>(type));
 26:         loader = (ExtensionLoader<T>) EXTENSION_LOADERS.get(type);
 27:     }
 28: }
```

- 第 12 至 15 行：必须是接口。
- 第 16 至 20 行：调用 `#withExtensionAnnotation()` 方法，校验必须使用 `@SPI` 注解标记。
- 第 22 至 27 行：从 `EXTENSION_LOADERS` **静态**中获取拓展接口对应的 ExtensionLoader 对象。若不存在，则创建 ExtensionLoader 对象，并添加到 `EXTENSION_LOADERS`。

4.3.2 构造方法

构造方法，代码如下：

```
/**
 * 拓展接口。
 * 例如，Protocol
 */
private final Class<?> type;
/**
 * 对象工厂
 *
 * 用于调用 {@link #injectExtension(Object)} 方法，向拓展对象注入依赖属性。
 *
 * 例如，StubProxyFactoryWrapper 中有 `Protocol protocol` 属性。
 */
private final ExtensionFactory objectFactory;

  1: private ExtensionLoader(Class<?> type) {
  2:     this.type = type;
  3:     objectFactory = (type == ExtensionFactory.class ? null : ExtensionLoader.g
etExtensionLoader(ExtensionFactory.class).getAdaptiveExtension());
  4: }
```

- `objectFactory` 属性，对象工厂，**功能上和 Spring IOC 一致**。
  - 用于调用 `#injectExtension(instance)` 方法时，向创建的拓展注入其依赖的属性。例如，`CacheFilter.cacheFactory` 属性。
  - 第 3 行：当拓展接口非 ExtensionFactory 时(如果不加这个判断，会是一个死循环)，调用 `ExtensionLoader#getAdaptiveExtension()` 方法，获得 ExtensionFactory 拓展接口的**自适应**拓展实现对象。**为什么呢**？在 「8. ExtensionFactory」 详细解析。

4.4 获得指定拓展对象

在 Dubbo 的代码里，常常能看到如下的代码：

```
ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(name)
```

4.4.1 getExtension

`#getExtension()` 方法，返回指定名字的扩展对象。如果指定名字的扩展不存在，则抛异常 IllegalStateException 。代码如下：

```
/**
 * 缓存的拓展对象集合
 *
 * key：拓展名
 * value：拓展对象
 *
 * 例如，Protocol 拓展
 *          key：dubbo value：DubboProtocol
 *          key：injvm value：InjvmProtocol
 *
 * 通过 {@link #loadExtensionClasses} 加载
 */
private final ConcurrentMap<String, Holder<Object>> cachedInstances = new Concurren
tHashMap<String, Holder<Object>>();

  1: /**
  2:  * Find the extension with the given name. If the specified name is not found,
 then {@link IllegalStateException}
  3:  * will be thrown.
  4:  */
  5: /**
  6:  * 返回指定名字的扩展对象。如果指定名字的扩展不存在，则抛异常 {@link IllegalStateExc
eption}.
  7:  *
```

```
 8:  * @param name 拓展名
 9:  * @return 拓展对象
10:  */
11: @SuppressWarnings("unchecked")
12: public T getExtension(String name) {
13:     if (name == null || name.length() == 0)
14:         throw new IllegalArgumentException("Extension name == null");
15:     // 查找 默认的 拓展对象
16:     if ("true".equals(name)) {
17:         return getDefaultExtension();
18:     }
19:     // 从 缓存中 获得对应的拓展对象
20:     Holder<Object> holder = cachedInstances.get(name);
21:     if (holder == null) {
22:         cachedInstances.putIfAbsent(name, new Holder<Object>());
23:         holder = cachedInstances.get(name);
24:     }
25:     Object instance = holder.get();
26:     if (instance == null) {
27:         synchronized (holder) {
28:             instance = holder.get();
29:             // 从 缓存中 未获取到，进行创建缓存对象。
30:             if (instance == null) {
31:                 instance = createExtension(name);
32:                 // 设置创建对象到缓存中
33:                 holder.set(instance);
34:             }
35:         }
36:     }
37:     return (T) instance;
38: }
```

- 第 15 至 18 行：调用 `#getDefaultExtension()` 方法，查询**默认的**拓展对象。在该方法的实现代码中，简化代码为 `getExtension(cachedDefaultName);` 。
- 第 19 至 28 行：从缓存中，获得拓展对象。
- 第 29 至 31 行：当缓存不存在时，调用 `#createExtension(name)` 方法，创建拓展对象。
- 第 33 行：添加创建的拓展对象，到缓存中。

4.4.2 createExtension

`#createExtension(name)` 方法，创建拓展名的拓展对象，并缓存。代码如下：

```
/**
 * 拓展实现类集合
 *
```

```
 * key：拓展实现类
 * value：拓展对象。
 *
 * 例如，key 为 Class<AccessLogFilter>
 *      value 为 AccessLogFilter 对象
 */
private static final ConcurrentMap<Class<?>, Object> EXTENSION_INSTANCES = new Conc
urrentHashMap<Class<?>, Object>();
```

```
 1: /**
 2:  * 创建拓展名的拓展对象，并缓存。
 3:  *
 4:  * @param name 拓展名
 5:  * @return 拓展对象
 6:  */
 7: @SuppressWarnings("unchecked")
 8: private T createExtension(String name) {
 9:     // 获得拓展名对应的拓展实现类
10:     Class<?> clazz = getExtensionClasses().get(name);
11:     if (clazz == null) {
12:         throw findException(name); // 抛出异常
13:     }
14:     try {
15:         // 从缓存中，获得拓展对象。
16:         T instance = (T) EXTENSION_INSTANCES.get(clazz);
17:         if (instance == null) {
18:             // 当缓存不存在时，创建拓展对象，并添加到缓存中。
19:             EXTENSION_INSTANCES.putIfAbsent(clazz, clazz.newInstance());
20:             instance = (T) EXTENSION_INSTANCES.get(clazz);
21:         }
22:         // 注入依赖的属性
23:         injectExtension(instance);
24:         // 创建 Wrapper 拓展对象
25:         Set<Class<?>> wrapperClasses = cachedWrapperClasses;
26:         if (wrapperClasses != null && !wrapperClasses.isEmpty()) {
27:             for (Class<?> wrapperClass : wrapperClasses) {
28:                 instance = injectExtension((T) wrapperClass.getConstructor(typ
e).newInstance(instance));
29:             }
30:         }
31:         return instance;
32:     } catch (Throwable t) {
33:         throw new IllegalStateException("Extension instance(name: " + name + "
, class: " +
34:                 type + ")  could not be instantiated: " + t.getMessage(), t);
35:     }
36: }
```

- 第 9 至 13 行：获得拓展名对应的拓展实现类。若不存在，调用 `#findException(name)` 方法，抛出异常。
- 第 16 行：从缓存 `EXTENSION_INSTANCES` **静态**属性中，获得拓展对象。
- 第 17 至 21 行：当缓存不存在时，创建拓展对象，并添加到 `EXTENSION_INSTANCES` 中。因为 `#getExtension(name)` 方法中已经加 `synchronized` 修饰，所以此处不用同步。
- 第 23 行：调用 `#injectExtension(instance)` 方法，向创建的拓展注入其依赖的属性。
- 第 24 至 30 行：创建 Wrapper 拓展对象，将 `instance` **包装在其中**。在 《Dubbo 开发指南 —— 扩展点加载》 文章中，如此介绍 Wrapper 类：

> Wrapper 类同样实现了扩展点接口，但是 Wrapper 不是扩展点的真正实现。它的用途主要是用于从 ExtensionLoader 返回扩展点时，包装在真正的扩展点实现外。即从 ExtensionLoader 中返回的实际上是 Wrapper 类的实例，Wrapper 持有了实际的扩展点实现类。
>
> 扩展点的 Wrapper 类可以有多个，也可以根据需要新增。
>
> 通过 Wrapper 类可以把所有扩展点公共逻辑移至 Wrapper 中。新加的 Wrapper 在所有的扩展点上添加了逻辑，有些类似 AOP，即 Wrapper 代理了扩展点。

  - 例如：ListenerExporterWrapper、ProtocolFilterWrapper 。

4.4.3 injectExtension

`#injectExtension(instance)` 方法，注入依赖的属性。代码如下：

```
 1: /**
 2:  * 注入依赖的属性
 3:  *
 4:  * @param instance 拓展对象
 5:  * @return 拓展对象
 6:  */
 7: private T injectExtension(T instance) {
 8:     try {
 9:         if (objectFactory != null) {
10:             for (Method method : instance.getClass().getMethods()) {
11:                 if (method.getName().startsWith("set")
12:                         && method.getParameterTypes().length == 1
13:                         && Modifier.isPublic(method.getModifiers())) { // sett
ing && public 方法
14:                     // 获得属性的类型
15:                     Class<?> pt = method.getParameterTypes()[0];
16:                     try {
17:                         // 获得属性
```

```
 18:                         String property = method.getName().length() > 3 ? meth
od.getName().substring(3, 4).toLowerCase() + method.getName().substring(4) : "";
 19:                         // 获得属性值
 20:                         Object object = objectFactory.getExtension(pt, propert
y);
 21:                         // 设置属性值
 22:                         if (object != null) {
 23:                             method.invoke(instance, object);
 24:                         }
 25:                     } catch (Exception e) {
 26:                         logger.error("fail to inject via method " + method.get
Name()
 27:                                 + " of interface " + type.getName() + ": " + e
.getMessage(), e);
 28:                     }
 29:                 }
 30:             }
 31:         }
 32:     } catch (Exception e) {
 33:         logger.error(e.getMessage(), e);
 34:     }
 35:     return instance;
 36: }
```

- 第 9 行：必须有 `objectFactory` 属性，即 ExtensionFactory 的拓展对象，不需要注入依赖的属性。
- 第 10 至 13 行：反射获得所有的方法，仅仅处理 `public setting` 方法。
- 第 15 行：获得属性的类型。
- 第 18 行：获得属性名。
- 第 20 行：获得**属性值**。**注意**，此处虽然调用的是 `ExtensionFactory#getExtension(type, name)` 方法，实际获取的不仅仅是拓展对象，也可以是 Spring Bean 对象。答案在 「8. ExtensionFactory」 揭晓。
- 第 21 至 24 行：设置属性值。

4.4.4 其他方法

如下方法，和该流程无关，胖友可自行查看。

- `#getLoadedExtension(name)`
- `#getLoadedExtensions()`

4.5 获得自适应的拓展对象

在 Dubbo 的代码里，常常能看到如下的代码：

```
ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension()
```

> 友情提示，胖友先看下 「6. Adaptive」 的内容，在回到此处。

4.5.1 getAdaptiveExtension

`#getAdaptiveExtension()` 方法，获得自适应拓展对象。

```
/**
 * 缓存的自适应( Adaptive )拓展对象
 */
private final Holder<Object> cachedAdaptiveInstance = new Holder<Object>();

/**
 * 创建 {@link #cachedAdaptiveInstance} 时发生的异常。
 *
 * 发生异常后，不再创建，参见 {@link #createAdaptiveExtension()}
 */
private volatile Throwable createAdaptiveInstanceError;
```

```
 1: /**
 2:  * 获得自适应拓展对象
 3:  *
 4:  * @return 拓展对象
 5:  */
 6: @SuppressWarnings("unchecked")
 7: public T getAdaptiveExtension() {
 8:     // 从缓存中，获得自适应拓展对象
 9:     Object instance = cachedAdaptiveInstance.get();
10:     if (instance == null) {
11:         // 若之前未创建报错，
12:         if (createAdaptiveInstanceError == null) {
13:             synchronized (cachedAdaptiveInstance) {
14:                 instance = cachedAdaptiveInstance.get();
15:                 if (instance == null) {
16:                     try {
17:                         // 创建自适应拓展对象
18:                         instance = createAdaptiveExtension();
19:                         // 设置到缓存
20:                         cachedAdaptiveInstance.set(instance);
21:                     } catch (Throwable t) {
22:                         // 记录异常
23:                         createAdaptiveInstanceError = t;
```

```
 24:                         throw new IllegalStateException("fail to create adapti
ve instance: " + t.toString(), t);
 25:                     }
 26:                 }
 27:             }
 28:         // 若之前创建报错，则抛出异常 IllegalStateException
 29:         } else {
 30:             throw new IllegalStateException("fail to create adaptive instance:
 " + createAdaptiveInstanceError.toString(), createAdaptiveInstanceError);
 31:         }
 32:     }
 33:     return (T) instance;
 34: }
```

- 第 9 行：从缓存 `cachedAdaptiveInstance` 属性中，获得自适应拓展对象。
- 第 28 至 30 行：若之前创建报错，则抛出异常 IllegalStateException 。
- 第 14 至 20 行：当缓存不存在时，调用 `#createAdaptiveExtension()` 方法，创建自适应拓展对象，并添加到 `cachedAdaptiveInstance` 中。
- 第 22 至 24 行：若创建发生异常，记录异常到 `createAdaptiveInstanceError` ，并抛出异常 IllegalStateException 。

4.5.2 createAdaptiveExtension

`#createAdaptiveExtension()` 方法，创建自适应拓展对象。代码如下：

```
/**
 * 创建自适应拓展对象
 *
 * @return 拓展对象
 */
@SuppressWarnings("unchecked")
private T createAdaptiveExtension() {
    try {
        return injectExtension((T) getAdaptiveExtensionClass().newInstance());
    } catch (Exception e) {
        throw new IllegalStateException("Can not create adaptive extension " + type
 + ", cause: " + e.getMessage(), e);
    }
}
```

- 调用 `#getAdaptiveExtensionClass()` 方法，获得自适应拓展类。
- 调用 `Class#newInstance()` 方法，创建自适应拓展对象。
- 调用 `#injectExtension(instance)` 方法，向创建的自适应拓展对象，注入依赖的属性。

4.5.3 getAdaptiveExtensionClass

`#getAdaptiveExtensionClass()` 方法，获得自适应拓展类。代码如下：

```
 1: /**
 2:  * @return 自适应拓展类
 3:  */
 4: private Class<?> getAdaptiveExtensionClass() {
 5:     getExtensionClasses();
 6:     if (cachedAdaptiveClass != null) {
 7:         return cachedAdaptiveClass;
 8:     }
 9:     return cachedAdaptiveClass = createAdaptiveExtensionClass();
10: }
```

- 【`@Adaptive` 的第一种】第 6 至 8 行：若 `cachedAdaptiveClass` 已存在，直接返回。的第一种情况。
- 【`@Adaptive` 的第二种】第 9 行：调用 `#createAdaptiveExtensionClass()` 方法，**自动生成**自适应拓展的代码实现，并**编译**后返回该类。

4.5.4 createAdaptiveExtensionClassCode

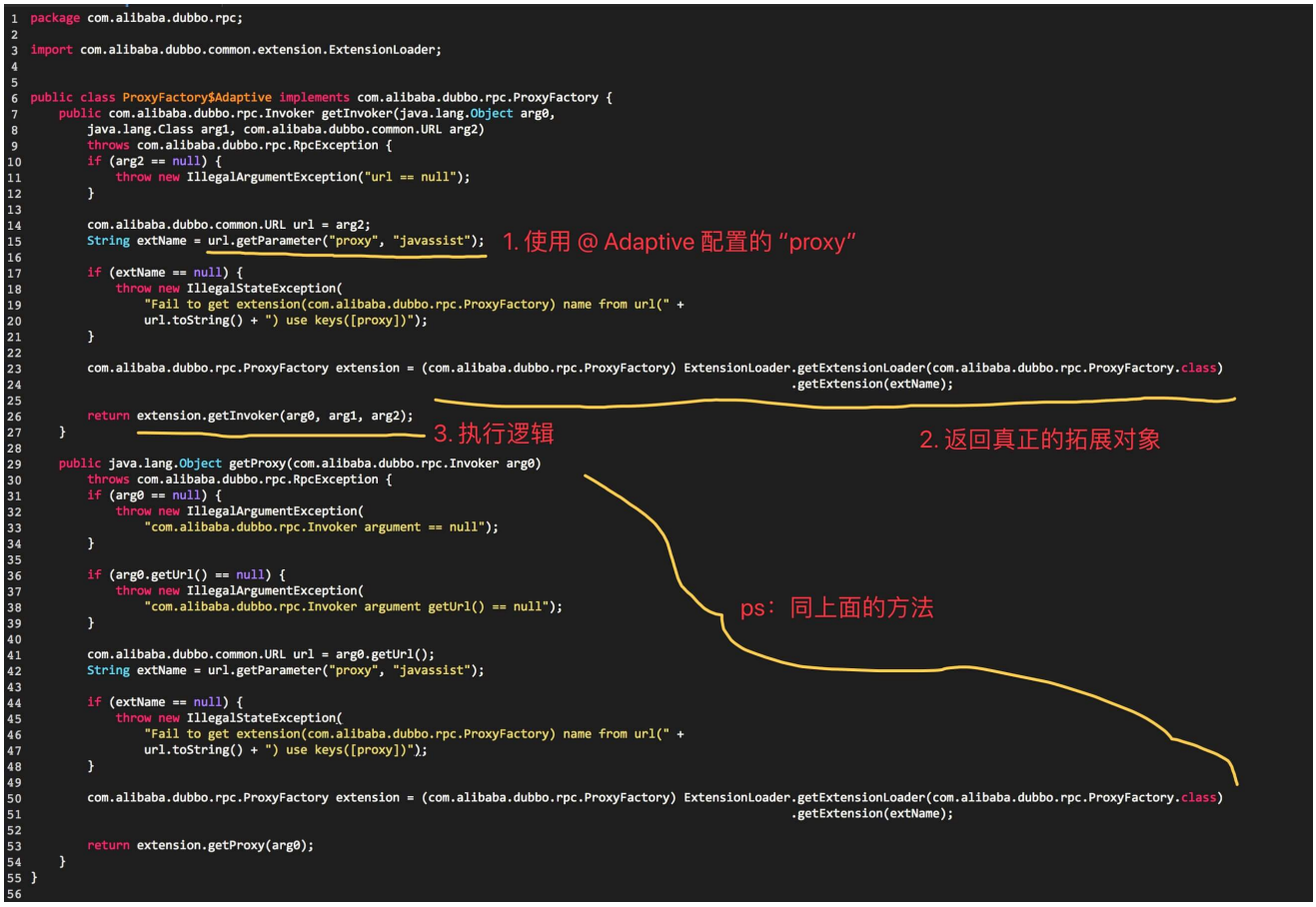
`#createAdaptiveExtensionClassCode()` 方法，自动生成自适应拓展的代码实现，并编译后返回该类。

```
 1: /**
 2:  * 自动生成自适应拓展的代码实现，并编译后返回该类。
 3:  *
 4:  * @return 类
 5:  */
 6: private Class<?> createAdaptiveExtensionClass() {
 7:     // 自动生成自适应拓展的代码实现的字符串
 8:     String code = createAdaptiveExtensionClassCode();
 9:     // 编译代码，并返回该类
10:     ClassLoader classLoader = findClassLoader();
11:     com.alibaba.dubbo.common.compiler.Compiler compiler = ExtensionLoader.getE
xtensionLoader(com.alibaba.dubbo.common.compiler.Compiler.class).getAdaptiveExtensi
on();
12:     return compiler.compile(code, classLoader);
13: }
```

- 第 8 行：调用 `#createAdaptiveExtensionClassCode` 方法，自动生成自适应拓展的代码实现

的字符串。

- 🙂 代码比较简单，已经添加详细注释，胖友点击查看。
- 如下是 ProxyFactory 的自适应拓展的代码实现的字符串生成**例子**




```
1  package com.alibaba.dubbo.rpc;
2
3  import com.alibaba.dubbo.common.extension.ExtensionLoader;
4
5
6  public class ProxyFactory$Adaptive implements com.alibaba.dubbo.rpc.ProxyFactory {
7      public com.alibaba.dubbo.rpc.Invoker getInvoker(java.lang.Object arg0,
8          java.lang.Class arg1, com.alibaba.dubbo.common.URL arg2)
9          throws com.alibaba.dubbo.rpc.RpcException {
10         if (arg2 == null) {
11             throw new IllegalArgumentException("url == null");
12         }
13
14         com.alibaba.dubbo.common.URL url = arg2;
15         String extName = url.getParameter("proxy", "javassist");
16
17         if (extName == null) {
18             throw new IllegalStateException(
19                 "Fail to get extension(com.alibaba.dubbo.rpc.ProxyFactory) name from url(" +
20                 url.toString() + ") use keys([proxy])");
21         }
22
23         com.alibaba.dubbo.rpc.ProxyFactory extension = (com.alibaba.dubbo.rpc.ProxyFactory) ExtensionLoader.getExtensionLoader(com.alibaba.dubbo.rpc.ProxyFactory.class)
24                                                                                             .getExtension(extName);
25
26         return extension.getInvoker(arg0, arg1, arg2);
27     }
28
29     public java.lang.Object getProxy(com.alibaba.dubbo.rpc.Invoker arg0)
30         throws com.alibaba.dubbo.rpc.RpcException {
31         if (arg0 == null) {
32             throw new IllegalArgumentException(
33                 "com.alibaba.dubbo.rpc.Invoker argument == null");
34         }
35
36         if (arg0.getUrl() == null) {
37             throw new IllegalArgumentException(
38                 "com.alibaba.dubbo.rpc.Invoker argument getUrl() == null");
39         }
40
41         com.alibaba.dubbo.common.URL url = arg0.getUrl();
42         String extName = url.getParameter("proxy", "javassist");
43
44         if (extName == null) {
45             throw new IllegalStateException(
46                 "Fail to get extension(com.alibaba.dubbo.rpc.ProxyFactory) name from url(" +
47                 url.toString() + ") use keys([proxy])");
48         }
49
50         com.alibaba.dubbo.rpc.ProxyFactory extension = (com.alibaba.dubbo.rpc.ProxyFactory) ExtensionLoader.getExtensionLoader(com.alibaba.dubbo.rpc.ProxyFactory.class)
51                                                                                             .getExtension(extName);
52
53         return extension.getProxy(arg0);
54     }
55 }
56
```

- 第 9 至 12 行：使用 Dubbo SPI 加载 Compier 拓展接口对应的拓展实现对象，后调用 `Compiler#compile(code, classLoader)` 方法，进行编译。🙂 因为不是本文的重点，后续另开文章分享。

4.6 获得激活的拓展对象数组

在 Dubbo 的代码里，看到使用代码如下：

```
List<Filter> filters = ExtensionLoader.getExtensionLoader(Filter.class).getActivate
Extension(invoker.getUrl(), key, group);
```

4.6.1 getExtensionLoader

`#getExtensionLoader(url, key, group)` 方法，获得符合自动激活条件的拓展对象数组。

```
1: /**
2:  * This is equivalent to {@code getActivateExtension(url, url.getParameter(key
).split(","), null)}
```

```
 3:  *
 4:  * 获得符合自动激活条件的拓展对象数组
 5:  *
 6:  * @param url   url
 7:  * @param key   url parameter key which used to get extension point names
 8:  *              Dubbo URL 参数名
 9:  * @param group group
10:  *              过滤分组名
11:  * @return extension list which are activated.
12:  * @see #getActivateExtension(com.alibaba.dubbo.common.URL, String[], String)
13:  */
14: public List<T> getActivateExtension(URL url, String key, String group) {
15:     // 从 Dubbo URL 获得参数值
16:     String value = url.getParameter(key);
17:     // 获得符合自动激活条件的拓展对象数组
18:     return getActivateExtension(url, value == null || value.length() == 0 ? nu
ll : Constants.COMMA_SPLIT_PATTERN.split(value), group);
19: }
20:
21: /**
22:  * Get activate extensions.
23:  *
24:  * 获得符合自动激活条件的拓展对象数组
25:  *
26:  * @param url    url
27:  * @param values extension point names
28:  * @param group  group
29:  * @return extension list which are activated
30:  * @see com.alibaba.dubbo.common.extension.Activate
31:  */
32: public List<T> getActivateExtension(URL url, String[] values, String group) {
33:     List<T> exts = new ArrayList<T>();
34:     List<String> names = values == null ? new ArrayList<String>(0) : Arrays.as
List(values);
35:     // 处理自动激活的拓展对象们
36:     // 判断不存在配置 `"-name"` 。例如, <dubbo:service filter="-default" /> , 代
表移除所有默认过滤器。
37:     if (!names.contains(Constants.REMOVE_VALUE_PREFIX + Constants.DEFAULT_KEY)
) {
38:         // 获得拓展实现类数组
39:         getExtensionClasses();
40:         // 循环
41:         for (Map.Entry<String, Activate> entry : cachedActivates.entrySet()) {
42:             String name = entry.getKey();
43:             Activate activate = entry.getValue();
44:             if (isMatchGroup(group, activate.group())) { // 匹配分组
45:                 // 获得拓展对象
46:                 T ext = getExtension(name);
```

```
47:                 if (!names.contains(name) // 不包含在自定义配置里。如果包含，会在下面的代码处理。
48:                         && !names.contains(Constants.REMOVE_VALUE_PREFIX + name) // 判断是否配置移除。例如 <dubbo:service filter="-monitor" />，则 MonitorFilter 会被移除
49:                         && isActive(activate, url)) { // 判断是否激活
50:                     exts.add(ext);
51:                 }
52:             }
53:         }
54:         // 排序
55:         Collections.sort(exts, ActivateComparator.COMPARATOR);
56:     }
57:     // 处理自定义配置的拓展对象们。例如在 <dubbo:service filter="demo" /> ，代表需要加入 DemoFilter （这个是笔者自定义的）。
58:     List<T> usrs = new ArrayList<T>();
59:     for (int i = 0; i < names.size(); i++) {
60:         String name = names.get(i);
61:         if (!name.startsWith(Constants.REMOVE_VALUE_PREFIX) && !names.contains(Constants.REMOVE_VALUE_PREFIX + name)) { // 判断非移除的
62:             // 将配置的自定义在自动激活的拓展对象们前面。例如，<dubbo:service filter="demo,default,demo2" /> ，则 DemoFilter 就会放在默认的过滤器前面。
63:             if (Constants.DEFAULT_KEY.equals(name)) {
64:                 if (!usrs.isEmpty()) {
65:                     exts.addAll(0, usrs);
66:                     usrs.clear();
67:                 }
68:             } else {
69:                 // 获得拓展对象
70:                 T ext = getExtension(name);
71:                 usrs.add(ext);
72:             }
73:         }
74:     }
75:     // 添加到结果集
76:     if (!usrs.isEmpty()) {
77:         exts.addAll(usrs);
78:     }
79:     return exts;
80: }
```

- 第 16 行：从 Dubbo URL 获得参数值。例如说，若 XML 配置 Service `<dubbo:service filter="demo, demo2" />` ，并且在获得 Filter 自动激活拓展时，此处就能解析到 `value=demo,demo2` 。另外， `value` 可以根据**逗号**拆分。
- 第 18 行：调用 `#getActivateExtension(url, values, group)` 方法，获得符合自动激活条件的拓展对象数组。

- 第 35 至 56 行：处理自动激活的拓展对象们。
  - `#isMatchGroup(group, groups)` 方法，匹配分组。
  - `#isActive(Activate, url)` 方法，是否激活，通过 Dubbo URL 中是否存在参数名为 `@Activate.value` ，并且参数值非空。
- 第 57 至 74 行：处理自定义配置的拓展对象们。
- 第 75 至 78 行：将 `usrs` 合并到 `exts` **尾部**。
- 🙂 代码比较简单，胖友直接看注释。

4.6.2 ActivateComparator

`com.alibaba.dubbo.common.extension.support.ActivateComparator` ，自动激活拓展对象排序器。

- 🙂 代码比较简单，胖友直接看注释。

5. @SPI

`com.alibaba.dubbo.common.extension.@SPI` ，扩展点接口的标识。代码如下：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface SPI {

    /**
     * default extension name
     */
    String value() default "";

}
```

- `value` ，默认拓展实现类的名字。例如，Protocol 拓展接口，代码如下：

  ```
  @SPI("dubbo")
  public interface Protocol {
      // ... 省略代码
  }
  ```

  - 其中 `"dubbo"` 指的是 DubboProtocol ， Protocol 默认的拓展实现类。

6. @Adaptive

`com.alibaba.dubbo.common.extension.@Adaptive` ，自适应拓展信息的标记。代码如下：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Adaptive {

    /**
     * Decide which target extension to be injected. The name of the target extensi
on is decided by the parameter passed
     * in the URL, and the parameter names are given by this method.
     * <p>
     * If the specified parameters are not found from {@link URL}, then the default
 extension will be used for
     * dependency injection (specified in its interface's {@link SPI}).
     * <p>
     * For examples, given <code>String[] {"key1", "key2"}</code>:
     * <ol>
     * <li>find parameter 'key1' in URL, use its value as the extension's name</li>
     * <li>try 'key2' for extension's name if 'key1' is not found (or its value is
empty) in URL</li>
     * <li>use default extension if 'key2' doesn't appear either</li>
     * <li>otherwise, throw {@link IllegalStateException}</li>
     * </ol>
     * If default extension's name is not give on interface's {@link SPI}, then a n
ame is generated from interface's
     * class name with the rule: divide classname from capital char into several pa
rts, and separate the parts with
     * dot '.', for example: for {@code com.alibaba.dubbo.xxx.YyyInvokerWrapper}, i
ts default name is
     * <code>String[] {"yyy.invoker.wrapper"}</code>. This name will be used to sea
rch for parameter from URL.
     *
     * @return parameter key names in URL
     */
    /**
     * 从 {@link URL }的 Key 名，对应的 Value 作为要 Adapt 成的 Extension 名。
     * <p>
     * 如果 {@link URL} 这些 Key 都没有 Value ，使用 缺省的扩展（在接口的{@link SPI}中
设定的值）。<br>
     * 比如，<code>String[] {"key1", "key2"}</code>，表示
     * <ol>
     *      <li>先在URL上找key1的Value作为要Adapt成的Extension名；
     *      <li>key1没有Value，则使用key2的Value作为要Adapt成的Extension名。
```

```
     *      <li>key2没有Value，使用缺省的扩展。
     *      <li>如果没有设定缺省扩展，则方法调用会抛出{@link IllegalStateException}。
     * </ol>
     * <p>
     * 如果不设置则缺省使用Extension接口类名的点分隔小写字串。<br>
     * 即对于Extension接口 {@code com.alibaba.dubbo.xxx.YyyInvokerWrapper} 的缺省值为
 <code>String[] {"yyy.invoker.wrapper"}</code>
     *
     * @see SPI#value()
     */
    String[] value() default {};

}
```

`@Adaptive` 注解，可添加**类**或**方法**上，分别代表了两种不同的使用方式。

> 友情提示：一个拓展接口，有且仅有一个 Adaptive 拓展实现类。

- 第一种，标记在**类**上，代表**手动实现**它是一个拓展接口的 Adaptive 拓展实现类。目前 Dubbo 项目里，只有 ExtensionFactory 拓展的实现类 AdaptiveExtensionFactory 有这么用。详细解析见 「8.1 AdaptiveExtensionFactory」 。
- 第二种，标记在拓展接口的**方法**上，代表**自动生成代码实现**该接口的 Adaptive 拓展实现类。
  - `value` ，从 Dubbo URL 获取参数中，使用键名(Key)，获取键值。该值为**真正的**拓展名。
    - 自适应拓展实现类，会获取拓展名对应的**真正**的拓展对象。通过该对象，执行真正的逻辑。
    - 可以设置**多个**键名(Key)，顺序获取直到**有值**。若最终获取不到，使用**默认拓展名**。
  - 在 「4.5.4 createAdaptiveExtensionClassCode」 详细解析。

7. @Activate

`com.alibaba.dubbo.common.extension.@Activate` ，自动激活条件的标记。代码如下：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Activate {
    /**
     * Activate the current extension when one of the groups matches. The group pas
```

```
sed into
     * {@link ExtensionLoader#getActivateExtension(URL, String, String)} will be us
ed for matching.
     *
     * @return group names to match
     * @see ExtensionLoader#getActivateExtension(URL, String, String)
     */
    /**
     * Group过滤条件。
     * <br />
     * 包含{@link ExtensionLoader#getActivateExtension}的group参数给的值，则返回扩展。
     * <br />
     * 如没有Group设置，则不过滤。
     */
    String[] group() default {};

    /**
     * Activate the current extension when the specified keys appear in the URL's p
arameters.
     * <p>
     * For example, given <code>@Activate("cache, validation")</code>, the current
extension will be return only when
     * there's either <code>cache</code> or <code>validation</code> key appeared in
 the URL's parameters.
     * </p>
     *
     * @return URL parameter keys
     * @see ExtensionLoader#getActivateExtension(URL, String)
     * @see ExtensionLoader#getActivateExtension(URL, String, String)
     */
    /**
     * Key过滤条件。包含{@link ExtensionLoader#getActivateExtension}的URL的参数Key中有
，则返回扩展。
     * <p/>
     * 示例：<br/>
     * 注解的值 <code>@Activate("cache,validatioin")</code>,
     * 则{@link ExtensionLoader#getActivateExtension}的URL的参数有<code>cache</code>
Key，或是<code>validatioin</code>则返回扩展。
     * <br/>
     * 如没有设置，则不过滤。
     */
    String[] value() default {};

    /**
     * Relative ordering info, optional
     *
     * @return extension list which should be put before the current one
     */
```

```
    /**
     * 排序信息，可以不提供。
     */
    String[] before() default {};

    /**
     * Relative ordering info, optional
     *
     * @return extension list which should be put after the current one
     */
    /**
     * 排序信息，可以不提供。
     */
    String[] after() default {};

    /**
     * Absolute ordering info, optional
     *
     * @return absolute ordering info
     */
    /**
     * 排序信息，可以不提供。
     */
    int order() default 0;
}
```

- 对于可以被框架中自动激活加载扩展，`@Activate` 用于配置扩展被自动激活加载条件。比如，Filter 扩展，有多个实现，使用 `@Activate` 的扩展可以根据**条件**被自动加载。
  - 这块的例子，可以看下 《Dubbo 开发指南 —— 扩展点加载》「扩展点自动激活」 文档提供的。
- 🙂 分成过滤条件和排序信息**两类属性**，胖友看下代码里的注释。
- 在 「4.6 获得激活的拓展对象数组」 详细解析。

8. ExtensionFactory

`com.alibaba.dubbo.common.extension.ExtensionFactory` ，拓展工厂接口。代码如下：

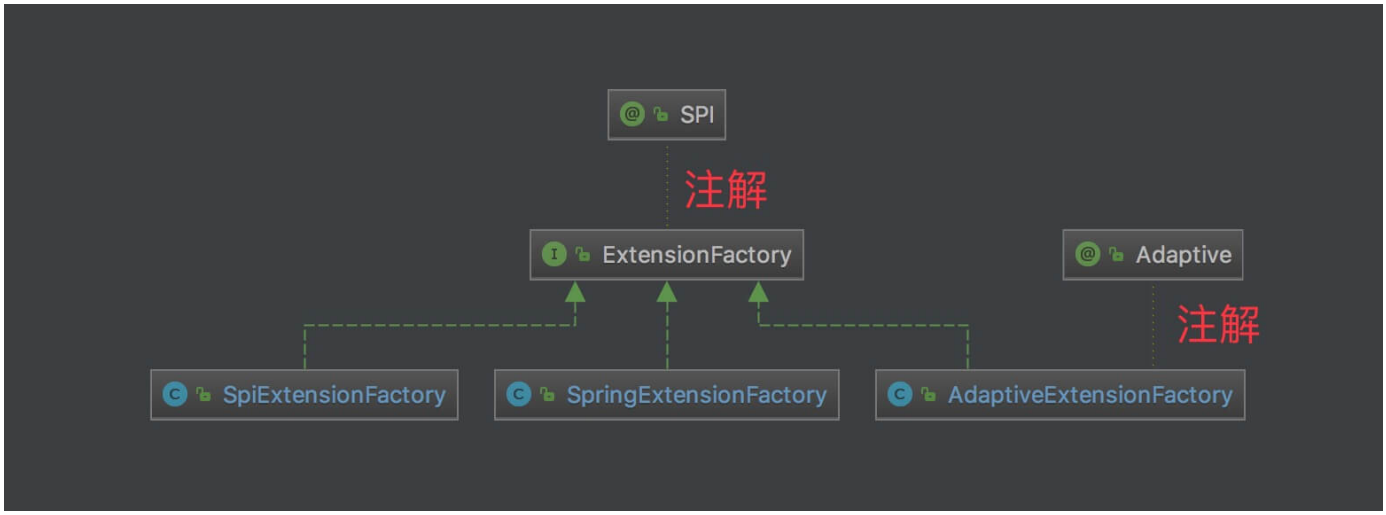
```
/**
 * ExtensionFactory
 *
 * 拓展工厂接口
 */
@SPI
```

```java
public interface ExtensionFactory {

    /**
     * Get extension.
     *
     * 获得拓展对象
     *
     * @param type object type. 拓展接口
     * @param name object name. 拓展名
     * @return object instance. 拓展对象
     */
    <T> T getExtension(Class<T> type, String name);

}
```

- ExtensionFactory 自身也是拓展接口，基于 Dubbo SPI 加载具体拓展实现类。
- `#getExtension(type, name)` 方法，在 「4.4.3 injectExtension」 中，获得拓展对象，向创建的拓展对象**注入依赖属性**。在实际代码中，我们可以看到不仅仅获得的是拓展对象，也可以是 Spring 中的 Bean 对象。
- ExtensionFactory 子类类图如下：



8.1 AdaptiveExtensionFactory

`com.alibaba.dubbo.common.extension.factory.AdaptiveExtensionFactory` ，自适应 ExtensionFactory 拓展实现类。代码如下：

```java
1: @Adaptive
2: public class AdaptiveExtensionFactory implements ExtensionFactory {
3:
4:     /**
5:      * ExtensionFactory 拓展对象集合
6:      */
```

```
 7:     private final List<ExtensionFactory> factories;
 8:
 9:     public AdaptiveExtensionFactory() {
10:         // 使用 ExtensionLoader 加载拓展对象实现类。
11:         ExtensionLoader<ExtensionFactory> loader = ExtensionLoader.getExtensio
nLoader(ExtensionFactory.class);
12:         List<ExtensionFactory> list = new ArrayList<ExtensionFactory>();
13:         for (String name : loader.getSupportedExtensions()) {
14:             list.add(loader.getExtension(name));
15:         }
16:         factories = Collections.unmodifiableList(list);
17:     }
18:
19:     public <T> T getExtension(Class<T> type, String name) {
20:         // 遍历工厂数组，直到获得到属性
21:         for (ExtensionFactory factory : factories) {
22:             T extension = factory.getExtension(type, name);
23:             if (extension != null) {
24:                 return extension;
25:             }
26:         }
27:         return null;
28:     }
29:
30: }
```

- `@Adaptive` 注解，为 ExtensionFactory 的**自适应**拓展实现类。
- **构造**方法，使用 ExtensionLoader 加载 ExtensionFactory 拓展对象的实现类。若胖友没自己实现 ExtensionFactory 的情况下，`factories` 为 SpiExtensionFactory 和 SpringExtensionFactory 。
- `#getExtension(type, name)` 方法，遍历 `factories` ，调用其 `#getExtension(type, name)` 方法，直到获得到属性值。

8.2 SpiExtensionFactory

`com.alibaba.dubbo.common.extension.factory.SpiExtensionFactory` ，SPI ExtensionFactory 拓展实现类。代码如下：

```
public class SpiExtensionFactory implements ExtensionFactory {

    /**
     * 获得拓展对象
     *
     * @param type object type. 拓展接口
```

```
     * @param name object name. 拓展名
     * @param <T> 泛型
     * @return 拓展对象
     */
    public <T> T getExtension(Class<T> type, String name) {
        if (type.isInterface() && type.isAnnotationPresent(SPI.class)) { // 校验是 @
SPI
            // 加载拓展接口对应的 ExtensionLoader 对象
            ExtensionLoader<T> loader = ExtensionLoader.getExtensionLoader(type);
            // 加载拓展对象
            if (!loader.getSupportedExtensions().isEmpty()) {
                return loader.getAdaptiveExtension();
            }
        }
        return null;
    }

}
```

8.3 SpringExtensionFactory

`com.alibaba.dubbo.config.spring.extension.SpringExtensionFactory`，Spring ExtensionFactory 拓展实现类。代码如下：

```
public class SpringExtensionFactory implements ExtensionFactory {

    /**
     * Spring Context 集合
     */
    private static final Set<ApplicationContext> contexts = new ConcurrentHashSet<A
pplicationContext>();

    public static void addApplicationContext(ApplicationContext context) {
        contexts.add(context);
    }

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

    @Override
    @SuppressWarnings("unchecked")
    public <T> T getExtension(Class<T> type, String name) {
        for (ApplicationContext context : contexts) {
            if (context.containsBean(name)) {
```

```
                // 获得属性
                Object bean = context.getBean(name);
                // 判断类型
                if (type.isInstance(bean)) {
                    return (T) bean;
                }
            }
        }
        return null;
    }

}
```

- `#getExtension(type, name)` 方法，遍历 `contexts` ，调用其 `ApplicationContext#getBean(name)` 方法，获得 Bean 对象，直到成功并且值类型正确。

8.3.1 例子

DemoFilter 是笔者实现的 Filter 拓展实现类，代码如下：

```
public class DemoFilter implements Filter {

    private DemoDAO demoDAO;

    @Override
    public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcExcep
tion {
        return invoker.invoke(invocation);
    }

    public DemoFilter setDemoDAO(DemoDAO demoDAO) {
        this.demoDAO = demoDAO;
        return this;
    }
}
```

- DemoDAO ，笔者在 Spring 中声明对应的 Bean 对象。

```
<bean id="demoDAO" class="com.alibaba.dubbo.demo.provider.DemoDAO" />
```

- 在 「4.4.3 injectExtension」 中，会调用 `#setDemoDAO(demo)` 方法，将 DemoFilter 依赖的属性 `demoDAO` 注入。

666. 彩蛋

比想象中的长的多的多。初始理解会比较辛苦，梳理干净后实际很简单。

能够耐心到此处的胖友，为你点赞。👍👍👍