Dubbo之SPI扩展点加载

Dubbo中的一个重要理念是“微内核、可扩展”，其高可扩展性离不开SPI扩展点发现机制的支持，它是对Java原生SPI的增强实现，同时规避了后者的一些不足：

1. 按需加载扩展点具类，避免JDK标准SPI的一次性加载带来的初始化耗时以及扩展点具类没被用上的资源浪费；
2. **AOP** 支持，抽调公共逻辑，对同一扩展点具类做装饰处理，依需要在其方法前后增加逻辑代码；
3. **IoC** 支持，一个扩展点可以直接 setter 注入其它扩展点；

> **NOTE** 为方便阐述，下述将所有 扩展点实现 统称为 扩展点具类 ，也即扩展点的某种具体实现类。

基础

基本使用

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。

> **NOTE** 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

示例：

以扩展 Dubbo 的协议为例，在协议的实现 jar 包内放置文本文件： `META-INF/dubbo/org.apache.dubbo.rpc.Protocol` ，内容为：

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

实现类内容：

```java
package com.alibaba.xxx;

import org.apache.dubbo.rpc.Protocol;

public class XxxProtocol implements Protocol {
    // ...
}
```

配置模块中的配置

Dubbo 配置模块中，扩展点均有对应配置属性或标签，通过配置指定使用哪个扩展实现。比如：

```
<dubbo:protocol name="xxx" />
```

> **NOTE** 扩展点使用单一实例加载（请确保扩展实现的线程安全性），缓存在 `ExtensionLoader` 中。

高级特性

Dubbo中的SPI机制是通过 `ExtensionLoader` 实现的，它的一些高阶特性需要通过注解@Activate和@Adaptive搭配完成。

扩展点自动包装 AOP支持

在《Dubbo与设计模式》一文中，已经对装饰者模式进行过深入的探讨。Dubbo中所谓的扩展点AOP支持，其实现其实是装饰者对被装饰者在行为委派时就其前后增加对应的特性业务代码，将对应扩展点的公共逻辑抽离到装饰者[Wrapper类]。如对某扩展点提供了 `Wrapper装饰者类` ，那么Dubbo会就此扩展点其它 `非Wrapper类实现` 做包装处理，返回是该 `Wrapper类` 的实例。

以上文出现的RPC框架层中的子层协议层扩展点 `Protocol` 为例，Dubbo使用该机制实现了RPC中的拦截链特性，具体请参看《 Dubbo RPC 之 Protocol协议层（二）》一文。

扩展点自动装配 IoC支持

`ExtensionLoader` 在加载扩展点时，会对 `setter` 方法进行判定，如果其入参是另外一个扩展点的话，如果扩展点有多个实现的话，需要采用 `扩展点自适应机制` 来确定使用哪个。

扩展点自适应 @Adaptive

当一个扩展点有多个实现时，依赖它的扩展点需要有某种机制确定到底使用最终使用哪个。Dubbo中被俗称为配置总线的URL，主要用于将配置作为参数在方法调用帧间传递，其背后实际上一个Key-Value[键值对]形式的Map容器， `ExtensionLoader` 利用 `URL + @Adaptive` 这对组合来确定选用哪个实现。

ExtensionLoader 会综合 URL + @Adaptive + 被依赖扩展点接口类 获得 Adaptive 实例，从而使得在方法执行时才决定被选用的其它扩展点具类成为可能，而当前对应 Adaptive 实现是在加载扩展点里动态生成。以 Dubbo 的 `Transporter`扩展点为例：

JAVA

```java
public interface Transporter {
    @Adaptive({"server", "transport"})
    Server bind(URL url, ChannelHandler handler) throws RemotingException;

    @Adaptive({"client", "transport"})
    Client connect(URL url, ChannelHandler handler) throws RemotingException;
}
```

实例中，对于 bind() 方法， Adaptive 实现先查找 server key，如果该 Key 没有值则找 transport key 值，来决定代理到哪个实际扩展点。

扩展点自激活 @Activate

> *对于集合类扩展点，比如：Filter, InvokerListener, ExportListener, TelnetHandler, StatusChecker 等，可以同时加载多个实现，此时，可以用自动激活来简化配置。*

如：

JAVA

```java
import org.apache.dubbo.common.extension.Activate;
import org.apache.dubbo.rpc.Filter;

@Activate // 无条件自动激活
public class XxxFilter implements Filter {
    // ...
}
```

或：

JAVA

```java
import org.apache.dubbo.common.extension.Activate;
import org.apache.dubbo.rpc.Filter;

@Activate("xxx") // 当配置了xxx参数，并且参数为有效值时激活，比如配了cache="lru"，自动激活
CacheFilter。
public class XxxFilter implements Filter {
    // ...
}
```

或：

```java
import org.apache.dubbo.common.extension.Activate;
import org.apache.dubbo.rpc.Filter;

@Activate(group = "provider", value = "xxx") // 只对提供方激活，group可
选"provider"或"consumer"
public class XxxFilter implements Filter {
    // ...
}
```

实现

在具体实现上，概括来讲，Dubbo将整个过程大体分为如下3步：

1. 读取 `classpath` 下的SPI配置文件；
2. 根据配置解析得到扩展点具类及其依赖扩展点具类的Class对象集；
3. 根据当前扩展点所持有的Class对象集按需实例化某个具类，包括注入其它扩展点具类实例；

但为了方便理解，下面剖析具体实现的章节将按照由表入里、依赖前置的原则逐层展开。

获取扩展点加载器

```java
ExtensionLoader.getExtensionLoader(SomeSPI.class).getXXXExtension(... args)
```

每当需要获取一个扩展点实例时，总会调用一段类似如上片段的代码，其中 `ExtensionLoader` 是SPI实现的核心类，是按需延时加载扩展点具类的加载器，每一个被 `@SPI` 标注的扩展点会对应它的一个实例，Dubbo使用一个 `ConcurrentMap<Class<?>, ExtensionLoader<?>>` 类型的Map容器将这种关系缓存起来，避免重复加载。

`ExtensionLoader` 构造函数是私有的，它的实例进行使用工厂方法获得，在实例化时，会对其中两个最为关键的 `final` 型属性赋值， `type` 是表征扩展点的接口类型，而 `objectFactory` 是用于最终获取扩展点实例的工厂。

JAVA

```
public class ExtensionLoader<T> {
    ...
    private static final ConcurrentMap<Class<?>, ExtensionLoader<?>>
        EXTENSION_LOADERS = new ConcurrentHashMap<>();

    private final ExtensionFactory objectFactory;

    private final Class<?> type;

    private ExtensionLoader(Class<?> type) {
        this.type = type;
        objectFactory = (type == ExtensionFactory.class ? null :

ExtensionLoader.getExtensionLoader(ExtensionFactory.class).getAdaptiveExtension());
    }

    private static <T> boolean withExtensionAnnotation(Class<T> type) {
        return type.isAnnotationPresent(SPI.class);
    }

    @SuppressWarnings("unchecked")
    public static <T> ExtensionLoader<T> getExtensionLoader(Class<T> type) {
        if (type == null) {
            throw new IllegalArgumentException("Extension type == null");
        }
        if (!type.isInterface()) {
            throw new IllegalArgumentException("Extension type (" + type + ") is not an
interface!");
        }
        if (!withExtensionAnnotation(type)) {
            throw new IllegalArgumentException("Extension type (" + type +
                    ") is not an extension, because it is NOT annotated with @" +
SPI.class.getSimpleName() + "!");
        }

        ExtensionLoader<T> loader = (ExtensionLoader<T>) EXTENSION_LOADERS.get(type);
        if (loader == null) {
            EXTENSION_LOADERS.putIfAbsent(type, new ExtensionLoader<T>(type));
            loader = (ExtensionLoader<T>) EXTENSION_LOADERS.get(type);
        }
        return loader;
    }
    ...
}
```

获取扩展点实例

获得扩展点加载器后，便可使用该加载器将扩展点具类类型的实例化， `getXXXExtension(… args)` 已经表示可用的实例化方法依据特性要求存在8种形式，为方便进一步讨论，下述细分为两章节加以剖析，将扩展点方法按特性分成两组，前4组用于实例化那种指定一个Key键只能对应一个实现的扩展点具类，后4组则完全用于实例化带有自激活特性的扩展点具类。

实例化具名扩展点具类

除了自激活型扩展点具类，其它类型的扩展点均可以认为是带键[Key]的：a）SPI配置文件的键值对；b） `@SPI` 标注中指定的名称；c） `@Adaptive` 标注在扩展点具类上。

1. `T getAdaptiveExtension()`：若某注解点有一个实现标注了 `@Adaptive`，利用该方法可以直接获取其实例，于一个注解点，Dubbo只允许一个有该标注的实现；
2. `T getDefaultExtension()`：若 `@SPI` 注解带有值，那么Dubbo使用该值可以获取一个默认的扩展点具类；
3. `T getExtension(String name)`：由SPI配置文件中出现的键值对中的键来实例化对应值表示的扩展点具类，实际上 `getDefaultExtension()` 最终也是通过委托它实现；
4. `T getLoadedExtension(String name)`：和 `getExtension(String name)` 的不同之处在于，它只试图去获取已经完成实例化的扩展点具类，如果不存在既有实例，便直接返回 `null` 值；

于具名扩展点实现类来说，需要有个存储其实例的缓存，这个缓存是一个指向 `Holder` 对象的引用，或者是含有它的Map容器，其实例是采用惰性机制进行实例化，其只能实例化一次，一个实现只存在单一的实例。Dubbo中无处不在的并发， `Key` 键所对应的扩展点具类对象自然成了被争用的资源，需要加锁处理。然而在加锁时该对象可能还不存在，因而引入了一个持有它的单元素容器类 `Holder`，加锁前均能获取或者新生成一个对应的 `Holder` 实例，获得锁后再判别对应扩展点具类是否存在实例，不存在则调用 `createAdaptiveExtension()` (c)或者 `createExtension()` (a)+(b)创建对象。

JAVA

```java
public class ExtensionLoader<T> {

    ...
    private volatile Throwable createAdaptiveInstanceError;

    private final Holder<Object> cachedAdaptiveInstance = new Holder<>();

    private final ConcurrentMap<String, Holder<Object>> cachedInstances
        = new ConcurrentHashMap<>();


    public T getAdaptiveExtension() {
        Object instance = cachedAdaptiveInstance.get();
        if (instance == null) {
            if (createAdaptiveInstanceError != null) {
                throw new IllegalStateException("Failed to create adaptive instance: "
+
                        createAdaptiveInstanceError.toString(),
                        createAdaptiveInstanceError);
            }

            synchronized (cachedAdaptiveInstance) {
                instance = cachedAdaptiveInstance.get();
                if (instance == null) {
                    try {
                        instance = createAdaptiveExtension();
                        cachedAdaptiveInstance.set(instance);
                    } catch (Throwable t) {
                        createAdaptiveInstanceError = t;
                        throw new IllegalStateException("Failed to create adaptive
instance: " + t.toString(), t);
                    }
                }
            }
        }

        return (T) instance;
    }

    public T getExtension(String name) {
        if (StringUtils.isEmpty(name)) {
            throw new IllegalArgumentException("Extension name == null");
        }

        //直接使用true表示获取默认扩展点实例
        if ("true".equals(name)) {
            return getDefaultExtension();
        }
        final Holder<Object> holder = getOrCreateHolder(name);
        Object instance = holder.get();
        if (instance == null) {
            synchronized (holder) {
                instance = holder.get();
                if (instance == null) {
                    instance = createExtension(name);
                    holder.set(instance);
```

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

    /**
     * Return default extension, return <code>null</code> if it's not configured.
     */
    public T getDefaultExtension() {
        getExtensionClasses();
        if (StringUtils.isBlank(cachedDefaultName) || "true".equals(cachedDefaultName))
{
            return null;
        }
        //获得cachedDefaultName后，反过来调用getExtension()
        return getExtension(cachedDefaultName);
    }

    public T getLoadedExtension(String name) {
        if (StringUtils.isEmpty(name)) {
            throw new IllegalArgumentException("Extension name == null");
        }
        Holder<Object> holder = getOrCreateHolder(name);
        return (T) holder.get();
    }

    //cachedInstances本身已经是线程安全的，顾无需重复加锁
    private Holder<Object> getOrCreateHolder(String name) {
        Holder<Object> holder = cachedInstances.get(name);
        if (holder == null) {
            cachedInstances.putIfAbsent(name, new Holder<>());
            holder = cachedInstances.get(name);
        }
        return holder;
    }
    ...
}
```

上述代码中出现了一个声明了 `volatile` 可见性保证的 `Throwable` 类型字段 `createAdaptiveInstanceError`，目的很明显——当多个线程同一时间针对某一特定扩展点调用 `getAdaptiveExtension()` 时，获得锁的线程若遇到异常，可以依靠 `volatile` 第一时间告诉其他参与争用的线程，避免重复执行必然发生错误的代码段。

实例化自激活扩展点具类

另外还存在如下其它4种形如 `getActivateExtension(URL url, … args)` 的方法，用于实例化当前扩展点所有具有自激活特性的实现，上文中已提及于一个扩展点，标注了 `@Activate` 自激活的扩展点具类是可以存在多个，并且它存在3种形式：a）无条件自激活；b）设置 `group="provider" | "consumer"` 限定作用方；c）配置总线中存在Key键@Activate注解中的配置值所对应的配置才激活。

1. `List<T> getActivateExtension(URL url, String key)`

2. `List<T> getActivateExtension(URL url, String key, String group)`
3. `List<T> getActivateExtension(URL url, String[] values)`
4. `List<T> getActivateExtension(URL url, String[] values, String group)`

如果 `@Activate` 注解中没有配置 `group` ，那么当前自激活扩展点具类可以作用于 `provider` 和 `consumer` 双方，否则只能作用在出现在配置中的一方：

JAVA

```java
private boolean isMatchGroup(String group, String[] groups) {
    if (StringUtils.isEmpty(group)) {
        return true;
    }
    if (groups != null && groups.length > 0) {
        for (String g : groups) {
            if (group.equals(g)) {
                return true;
            }
        }
    }
    return false;
}
```

满足Key键的自激活扩展点具类，当前配置总线中需要存在非对应Key键的非空 `false|0|null|N/A` 配置值，其中总线URL中的Key键可能是加了前缀为RPC方法名称"somemethod"+"."的：

JAVA

```java
public class ConfigUtils {
    ...
    public static boolean isNotEmpty(String value) {
        return !isEmpty(value);
    }

    public static boolean isEmpty(String value) {
        return StringUtils.isEmpty(value)
                || "false".equalsIgnoreCase(value)
                || "0".equalsIgnoreCase(value)
                || "null".equalsIgnoreCase(value)
                || "N/A".equalsIgnoreCase(value);
    }
    ...
}

private boolean isActive(String[] keys, URL url) {
    if (keys.length == 0) {
        return true;
    }
    for (String key : keys) {
        for (Map.Entry<String, String> entry : url.getParameters().entrySet()) {
            String k = entry.getKey();
            String v = entry.getValue();
            if ((k.equals(key) || k.endsWith("." + key))
                    && ConfigUtils.isNotEmpty(v)) {
                return true;
            }
        }
    }
    return false;
}
```

于扩展点中接口前声明的 `@Activate` ，其配置中所表示的扩展点具类集合是Dubbo默认支持的，一般会采用首个，当没有找到对应名字的扩展点具类时会采用第二个，以此类推，开发者也可以通过在配置总线设置 `"-"+("default"|"Name4SpiImpl")` 前缀将其指定的扩展点实现排除，若为*-default*，`@Activate` 中的所有配置指定依赖了哪些扩展点具类的Key键均会被忽略。在调用 `getActivateExtension()` 方法时，除非排除，Dubbo会自动加入所有自适配扩展点具类实例，记为ListO，他们会按照配置的 `order` 值的从小到大排序。假如 `values` 中配有 `"default"` ，将自定义加载的其它扩展点具类对象一分为二变成前后ListA和ListB两个集合，那么最终得到的所有扩展点具类实例集合 `ListR=[ListA + ListO + ListB]` 。在没有配 `"default"` 时，可以认为ListA是空值，此时有 `ListR=[ListO + ListB]` 。另外如果values参数中包含了应该在ListO中出现的元素对应扩展点具类名称，那么对应实例将会“移入”到ListA或ListB中，这个特性让Dubbo可以在调用时决定所有依赖扩展点具类实例的最终执行顺序。

JAVA

```
public List<T> getActivateExtension(URL url, String[] values, String group) {
    List<T> exts = new ArrayList<>();
    List<String> names = values == null ? new ArrayList<>(0) : Arrays.asList(values);
    if (!names.contains(REMOVE_VALUE_PREFIX + DEFAULT_KEY)) {
        getExtensionClasses();
        for (Map.Entry<String, Object> entry : cachedActivates.entrySet()) {
            String name = entry.getKey();
            Object activate = entry.getValue();

            String[] activateGroup, activateValue;

            if (activate instanceof Activate) {
                activateGroup = ((Activate) activate).group();
                activateValue = ((Activate) activate).value();
            } else if (activate instanceof com.alibaba.dubbo.common.extension.Activate)
{
                activateGroup = ((com.alibaba.dubbo.common.extension.Activate)
activate).group();
                activateValue = ((com.alibaba.dubbo.common.extension.Activate)
activate).value();
            } else {
                continue;
            }
            if (isMatchGroup(group, activateGroup)
                    //参数中已经出现的扩展点实现实例移入到ListA或ListB中
                    && !names.contains(name)
                    //参数中加了“-”前缀的自适配实现被排除掉
                    && !names.contains(REMOVE_VALUE_PREFIX + name)
                    && isActive(activateValue, url)) {
                exts.add(getExtension(name));
            }
        }
        exts.sort(ActivateComparator.COMPARATOR);
    }
    List<T> usrs = new ArrayList<>();
    for (int i = 0; i < names.size(); i++) {
        String name = names.get(i);
        if (!name.startsWith(REMOVE_VALUE_PREFIX)
                && !names.contains(REMOVE_VALUE_PREFIX + name)) {
            //default之前的实例呈现在ListA位置，自适配的为ListO
            if (DEFAULT_KEY.equals(name)) {
                if (!usrs.isEmpty()) {
                    exts.addAll(0, usrs);
                    usrs.clear();
                }
            } else {
                usrs.add(getExtension(name));
            }
        }
    }
    //加入最后的ListB部分
    if (!usrs.isEmpty()) {
        exts.addAll(usrs);
    }
    //得到[ListA + ListO + ListB]或[ListO + ListB]
```

```
    return exts;
}
```

其它3个变种方法均是对上述这个方法的调用，如下：

JAVA

```
public List<T> getActivateExtension(URL url, String key) {
    return getActivateExtension(url, key, null);
}
public List<T> getActivateExtension(URL url, String[] values) {
    return getActivateExtension(url, values, null);
}
public List<T> getActivateExtension(URL url, String key, String group) {
    String value = url.getParameter(key);
    return getActivateExtension(url, StringUtils.isEmpty(value)
        ? null : COMMA_SPLIT_PATTERN.split(value), group);
}
```

获取扩展点具类元数据

上述关于 ExtensionLoader 的整个源码剖析阐述中，更多关于SPI功能性的，比较浅层次。获取到实例前的最关键一环是得到扩展点的具类信息，也即需要得到产生实例的元数据，上述多次出现的 getExtensionClasses() 正是确保元数据先加载的一个步骤，该方法利用锁和锁的双检形式保证了只会被加载一次，如下源码所示：

JAVA

```
private Map<String, Class<?>> getExtensionClasses() {
    Map<String, Class<?>> classes = cachedClasses.get();
    if (classes == null) {
        synchronized (cachedClasses) {
            classes = cachedClasses.get();
            if (classes == null) {
                classes = loadExtensionClasses();
                cachedClasses.set(classes);
            }
        }
    }
    return classes;
}
```

SPI文件解析

Dubbo的SPI机制中，扩展点实际上是一个接口，其具体实现是由应用配置在 META-INF/dubbo 资源目录中的一个和接口同名的文件中，在编译期间是不知道该具体实现的，只有等到需要用时，才会综合“ 配置总线URL、@SPI、@Adaptive、@Activate ”等信息去决定加载对应的具类，然后由后者获得对应的实例，因此解析RPC配置文件必须前置且很非常关键的一步。

读取SPI配置文件集

第一步需要获取到当前方法调用帧中正在使用的 ClassLoader ，再结合默认给定的SPI配置 classpath 目录和当前扩展点接口类名获取到所有SPI配置文件。一个Java工程中，通常会依赖其它的jar包，和当前工程一样，他们都各自有自己的 classpath 目录，里面除了包含被编译过的class文件外，一般也会含有一些放置在 ./META-INF 目录中的配置文件。当调用 ClassLoader 的 getResources(fileName) 方法或 ClassLoader.getSystemResources(fileName) 时，这些依赖jar包中的配置文件也会一同被获取到，因而需要迭代获得多个SPI配置文件，逐个执行文件解析。

```JAVA
private void loadDirectory(Map<String, Class<?>> extensionClasses, String dir, String
type) {
    String fileName = dir + type;
    try {
        Enumeration<java.net.URL> urls;
        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();
                loadResource(extensionClasses, classLoader, resourceURL);
            }
        }
    } catch (Throwable t) {
        logger.error("Exception occurred when loading extension class (interface: " +
                type + ", description file: " + fileName + ").", t);
    }
}
```

上述代码中的 findClassLoader() ，实际上是调用 ClassUtils.getClassLoader(ExtensionLoader.class) 获得当前 ClassLoader 实例，它总是按照如是顺序其尝试获取该实例：1） Thread.currentThread().getContextClassLoader() ；2） ExtensionLoader.class.getClassLoader() ；3） ClassLoader.getSystemClassLoader() 。在尝试了3种方式还是获取不到一个 ClassLoader 实例时，Dubbo便会直接使用 ClassLoader.getSystemResources(fileName) 。

> META-INF/services/ 、 META-INF/dubbo/ 、 META-INF/dubbo/internal/ *三个值，都是dubbo寻找扩展实现类的配置文件存放路径，也就是我在上述（一）注解@SPI中讲到的以接口全限定名命名的配置文件存放的路径。区别在于* META-INF/services/ *是dubbo为了兼容jdk的SPI扩展机制思想而设存在的，* META-INF/dubbo/internal/ *是dubbo内部提供的扩展的配置文件路径，而* META-INF/dubbo/ *是为了给用户自定义的扩展实现配置文件存放。*

也就是说Dubbo中的SPI配置文件所在classpath的位置是有要求，不能随便放置，具体实现表达如下：

```java
private static final String SERVICES_DIRECTORY = "META-INF/services/";

private static final String DUBBO_DIRECTORY = "META-INF/dubbo/";

private static final String DUBBO_INTERNAL_DIRECTORY = DUBBO_DIRECTORY + "internal/";


/**
 * synchronized in getExtensionClasses
 * */
private Map<String, Class<?>> loadExtensionClasses() {

    //默认扩展点加载，下文将加以阐述
    cacheDefaultExtensionName();

    Map<String, Class<?>> extensionClasses = new HashMap<>();
    loadDirectory(extensionClasses, DUBBO_INTERNAL_DIRECTORY, type.getName());
    loadDirectory(extensionClasses, DUBBO_DIRECTORY, type.getName());
    loadDirectory(extensionClasses, SERVICES_DIRECTORY, type.getName());

    return extensionClasses;
}
```

逐行解析SPI配置文件

接下来便是由SPI配置文件逐行解析出配置信息，如下源码，总体步骤如下：

1. 由参数 `resourceURL` 获取到文件字节流；
2. 使用装饰器 `InputStreamReader` 基于字节流得到字符流；
3. 套上另外一个装饰器 `BufferedReader` 获取到带有缓冲功能的字符流；
4. 逐行读取字符流，将当前行的注释信息忽略，随后取得等号 `"="` 两边的名称和扩展点类名；
5. 针对当前行执行类的元数据加载操作；

JAVA

```
private void loadResource(Map<String, Class<?>> extensionClasses, ClassLoader
classLoader, java.net.URL resourceURL) {
    try {
        try (BufferedReader reader = new BufferedReader(new
InputStreamReader(resourceURL.openStream(), StandardCharsets.UTF_8))) {
            String line;
            while ((line = reader.readLine()) != null) {
                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('=');
                        if (i > 0) {
                            name = line.substring(0, i).trim();
                            line = line.substring(i + 1).trim();
                        }
                        if (line.length() > 0) {
                            loadClass(extensionClasses, resourceURL,
Class.forName(line, true, classLoader), name);
                        }
                    } catch (Throwable t) {
                        IllegalStateException e = new IllegalStateException("Failed to
load extension class (interface: " + type + ", class line: " + line + ") in " +
resourceURL + ", cause: " + t.getMessage(), t);
                        exceptions.put(line, e);
                    }
                }
            }
        }
    } catch (Throwable t) {
        logger.error("Exception occurred when loading extension class (interface: " +
                type + ", class file: " + resourceURL + ") in " + resourceURL, t);
    }
}
```

注：上述代码中使用了Java7中的 `try(…){}catch(…){}` ，会自动完成I/O中资源的回收处理。

IMPORTANT

> Dubbo利用 `Class.forName()` 方法根据扩展点具类全名获取到对应的Class对象具类元数据，它会自动完成类的定位、加载、链接。由于指定 `initialize=true` ，同一具类若没有初始化过，Java会执行Class对象的初始化处理。

JAVA

```
public static Class<?> forName(String name, boolean initialize, ClassLoader loader)
```

扩展点具类元数据加载

`loadClass()` 是为扩展点准备好其所有实现具类的元数据，也即表示具类的 `Class<?>对象` ，有了它们才能进一步获取到具类的实例。

类加载，也即 `loadClass()` 所表示的这个过程，由于特性要求涉及到不少技术点，整个过程比较复杂，按惯例，还是化繁为简，先零后整，逐个击破。

> **IMPORTANT**
>
> `ExtensionLoader` 会为每一个扩展点准备它的一个实例， `loadClass()` 这一环已经是在加载扩展点具类信息了，但所有的具类信息都是汇总在同一个 `ExtensionLoader` 实例下加以管理的。

扩展点的实例惰性加载的第一步是获取具类元数据，第二步才是根据这些准备好的元数据按照当前配置总线要求完成某个具类的实例化操作。上述已经提到，第一步对于当前 `type` 扩展点只会执行一次，第二步则是每一个具类会执行一次实例的初始化操作，后续需要他们的时候均能直接从缓存提取到。

具类的加载过程采用的是分治思想，按照当前具类的类注解@Adaptive、@Activate、@Extension、构造函数是否入参为当前扩展点接口类拆解成4种具类的加载过程：1）自适配具类；2）装饰型具类；3）自激活型具类；4） 一般具类。

在SPI配置文件中，可以根据需要就对应扩展点具类以 `","` 作为分隔符赋予多个名字。JDK中标准的SPI配置是没有Key键的，为了兼容， `ExtensionLoader` 使用 `findAnnotationName()` 方法创建Key，规则是若具类配置了 `@Extension` 注解，取其名称，否则取具类本身的名称或者去除扩展点接口名后缀后得到的小写字符串。

整个具类元数据的总体加载过程如下：

JAVA

```java
private void loadClass(Map<String, Class<?>> extensionClasses, java.net.URL
resourceURL, Class<?> clazz, String name) throws NoSuchMethodException {
    //具类必须是扩展点接口的实现类
    if (!type.isAssignableFrom(clazz)) {
        throw new IllegalStateException("Error occurred when loading extension class
(interface: " +
                type + ", class line: " + clazz.getName() + "), class "
                + clazz.getName() + " is not subtype of interface.");
    }

    if (clazz.isAnnotationPresent(Adaptive.class)) {
    //自适配具类元数据加载
        cacheAdaptiveClass(clazz);
    } else if (isWrapperClass(clazz)) {
    //装饰型具类元数据加载
        cacheWrapperClass(clazz);
    } else {
        clazz.getConstructor();//确保有一个无参构造函数，没有则报错

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

        String[] names = NAME_SEPARATOR.split(name);
        if (ArrayUtils.isNotEmpty(names)) {
            //自激活型具类元数据加载
            cacheActivateClass(clazz, names[0]);

            for (String n : names) {
                //一般具类元数据加载
                cacheName(clazz, n);
                saveInExtensionClass(extensionClasses, clazz, n);
            }
        }
    }
}

private String findAnnotationName(Class<?> clazz) {
    org.apache.dubbo.common.Extension extension =
clazz.getAnnotation(org.apache.dubbo.common.Extension.class);
    if (extension != null) {
        return extension.value();
    }

    String name = clazz.getSimpleName();
    if (name.endsWith(type.getSimpleName())) {
        name = name.substring(0, name.length() - type.getSimpleName().length());
    }
    return name.toLowerCase();
}
```

每一种细分具类都有对应的缓存装载其类型元数据，也相应需要配合一些验重逻辑，下述分子章节进一步阐述：

自适配具类

保证只会有一个标注了 `@Adaptive` 注解的具类，注意它使用一个声明了 `volatile` 可见性的属性进行存取。

JAVA

```
private volatile Class<?> cachedAdaptiveClass = null;

/**
* cache Adaptive class which is annotated with <code>Adaptive</code>
 */
private void cacheAdaptiveClass(Class<?> clazz) {
    if (cachedAdaptiveClass == null) {
        cachedAdaptiveClass = clazz;
    } else if (!cachedAdaptiveClass.equals(clazz)) {
        throw new IllegalStateException("More than 1 adaptive class found: "
                + cachedAdaptiveClass.getName()
                + ", " + clazz.getName());
    }
}
```

装饰型具类

可以层层装饰，因此使用 `Set` 作为集合容器。代码中以当前扩展点的接口类信息作为入参调用 `clazz.getConstructor(type)` 便轻松判断当前具类是否为装饰类。

JAVA

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

/**
 * cache wrapper class
 * <p>
 * like: ProtocolFilterWrapper, ProtocolListenerWrapper
 */
private void cacheWrapperClass(Class<?> clazz) {
    if (cachedWrapperClasses == null) {
        cachedWrapperClasses = new ConcurrentHashSet<>();
    }
    cachedWrapperClasses.add(clazz);
}

/**
 * test if clazz is a wrapper class
 * <p>
 * which has Constructor with given class type as its only argument
 */
private boolean isWrapperClass(Class<?> clazz) {
    try {
        clazz.getConstructor(type);
        return true;
    } catch (NoSuchMethodException e) {
        return false;
    }
}
```

自激活型具类

可以有多个，比如 `Filter` ，和一般的具类缓存类的元数据 `Class<?>` 对象不同，它只缓存“名字”和“@Activate对象”的键值信息。存取容器Map的值应该 `Activate` 类型，但因该注解有两个，为兼容，委曲求全，被声明成了Object类型。

JAVA

```
private final Map<String, Object> cachedActivates = new ConcurrentHashMap<>();

/**
 * cache Activate class which is annotated with <code>Activate</code>
 * <p>
 * for compatibility, also cache class with old alibaba Activate annotation
 */
private void cacheActivateClass(Class<?> clazz, String name) {
    Activate activate = clazz.getAnnotation(Activate.class);
    if (activate != null) {
        cachedActivates.put(name, activate);
    } else {
        // support com.alibaba.dubbo.common.extension.Activate
        com.alibaba.dubbo.common.extension.Activate oldActivate =
clazz.getAnnotation(com.alibaba.dubbo.common.extension.Activate.class);
        if (oldActivate != null) {
            cachedActivates.put(name, oldActivate);
        }
    }
}
```

一般具类

可以有多个，缓存了“名称”和“类的元数据 Class<?> 对象”的键值信息。如下 ExtensionLoader 还反向缓存了他们间的关系，由于一般具类可以有多个“名称”，因此会存在多个“名称”指向同一个 Class<?> 对象”的情况。

JAVA

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

private final ConcurrentMap<Class<?>, String> cachedNames = new ConcurrentHashMap<>();

/**
 * cache name
 */
private void cacheName(Class<?> clazz, String name) {
    if (!cachedNames.containsKey(clazz)) {
        cachedNames.put(clazz, name);
    }
}

/**
 * put clazz in extensionClasses
 */
private void saveInExtensionClass(Map<String, Class<?>> extensionClasses, Class<?>
clazz, String name) {
    Class<?> c = extensionClasses.get(name);
    if (c == null) {
        extensionClasses.put(name, clazz);
    } else if (c != clazz) {
        throw new IllegalStateException("Duplicate extension " + type.getName() + "
name " + name + " on " + c.getName() + " and " + clazz.getName());
    }
}
```

实例化扩展点具类

有了各扩展点具类的元数据信息 `Class<?>` 后，创建其实例就比较简单了。对于所有类型的具类来说，获得其实例实际上需要三步：

1. 检查是否存在一个对应的实例，有直接返回，没有则继续第二步；
2. 使用 `Class<?>` 生成一个对象；
3. 注入处理：业务特性需求，有些具类需要依赖其它的扩展点接口，从而添加了相应的 `setter` ，需要给第二步生成的对象注入被依赖扩展点具类实例；

扩展点具类实例化处理

注：自适配具类存在两种形式，一种是开发人员使用 `@Adaptive` 注解的扩展点实现，另一类是由Dubbo根据扩展点中其方法中的的 `@Adaptive` 注解来生成的扩展点实现。整个实例化过程有点不一样，另起章节讨论

开发者为Dubbo提供了扩展点具类后，需要加入到相应SPI扩展点配置文件中，否则就成了孤魂野鬼不会发生作用，要用它时也找不到。键值对的硬性要求这一点，确保了尽管有多种类型的具类，但他们的实例获取方式是一致的，使用的基本都是 `createExtension()` 。

> **NOTE**
>
> Dubbo要求同一个扩展点的不同具类需要配置不同的名称，否则会强行抛错。

Dubbo的SPI规定一个具类只能存在一个实例，也就是说扩展点使用的是单例模式，因此声明了一个全局 `ConcurrentMap<Class<?>, Object>` 类型的容器缓存类到对象间的关系，确保唯一性。

仔细阅读过上文的，不难看出扩展点的具类的构造函数只能存在两种形式：1）无参；2）类型为当前扩展点接口的单一入参。前者简单调用 `clazz.newInstance()` 便能获取到实例 `instance` 。

后一种形式，也就是上文提及的用于支持类AOP特性的 `Wrapper装饰者类` ，它的目的是用于装饰前一种无参构造函数扩展点具类，不会为其单独创建实例，借助缓存 `cachedWrapperClasses` ，`ExtensionLoader` 挨个遍历其元素，对 `instance` 加以层层包装并完成注入处理。

实例创建过程中有点特殊的是，无论对应具类的实例是从缓存中获取的，还是新创建的，都会执行自动装配和自动包装这两个步骤，具体原因不明。

JAVA

```
private static final ConcurrentMap<Class<?>, Object> EXTENSION_INSTANCES
    = new ConcurrentHashMap<>();

private T createExtension(String name) {
    Class<?> clazz = getExtensionClasses().get(name);
    if (clazz == null) {
        throw findException(name);
    }
    try {
        T instance = (T) EXTENSION_INSTANCES.get(clazz);
        if (instance == null) {
            EXTENSION_INSTANCES.putIfAbsent(clazz, clazz.newInstance());
            instance = (T) EXTENSION_INSTANCES.get(clazz);
        }
        //自动装配
        injectExtension(instance);

        //自动包装
        Set<Class<?>> wrapperClasses = cachedWrapperClasses;
        if (CollectionUtils.isNotEmpty(wrapperClasses)) {
            for (Class<?> wrapperClass : wrapperClasses) {
                //1.取得装饰者扩展点具类
                //2.使用type获取构造函数
                //3.将instance传入调用构造函数创建示例，完成包装处理
                instance = injectExtension((T)
wrapperClass.getConstructor(type).newInstance(instance));
            }
        }
        return instance;
    } catch (Throwable t) {
        throw new IllegalStateException("Extension instance (name: " + name + ", class:
" +
                type + ") couldn't be instantiated: " + t.getMessage(), t);
    }
}
```

上文中有两处关于异常的点比较特殊， `throw findException(name)` 和 `exceptions.put(line, e)` ，不难看出 `ExtensionLoader` 将加载具类元数据时候出现的错误延迟到创建实例时才抛出，系统运行期间这两个时段可能是紧接着发生的，也可能前后相距比较大的时差，上文可以找到线索。

IMPORTANT

扩展点具类的实例创建过程相当简短精悍，自动装配、包装处理的风轻云淡，在研读相关实现源码时，有时会因为忽略掉这些细节而迷失方向，有些断片般的迷离。从SPI的自动包装的实现过程可以看出，一个扩展点，如果有多个包装类，那么在实例化的时候，它的任意其它非包装类扩展点具类，均会被这些类包装一遍，最后返回一个经过了层层包裹的对象，并且每一层都已经完成依赖注入处理，返回实例的最终行为是它们的总和。

依赖注入处理

上述用到的注入方法 `injectExtension()` ，是完成自动装配的，简单讲就是遍历目标对象的所有 `setter` 方法，若方法入参不为基本类型，便试图调用 `objectFactory.getExtension()` 获取一个扩展点实例，取到了则给设入处理。

```java
private T injectExtension(T instance) {

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

    try {
        for (Method method : instance.getClass().getMethods()) {
            if (!isSetter(method)) {
                continue;
            }
            if (method.getAnnotation(DisableInject.class) != null) {
                continue;
            }

            //获取入参类型
            Class<?> pt = method.getParameterTypes()[0];
            if (ReflectUtils.isPrimitives(pt)) {
                continue;
            }

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

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

private String getSetterProperty(Method method) {
    return method.getName().length() > 3 ?
        method.getName().substring(3, 4).toLowerCase()
            + method.getName().substring(4) : "";
}

//含有一个入参的public型set方法
private boolean isSetter(Method method) {
    return method.getName().startsWith("set")
            && method.getParameterTypes().length == 1
            && Modifier.isPublic(method.getModifiers());
}
```

上述代码中说明，可以使用 @DisableInject 注解显示告知当前具类忽略掉扩展点注入处理。另外可以认为 objectFactory.getExtension() 实际上就是调用了某扩展点所对应的 ExtensionLoader 实例由入参类型获得的 getExtension(name) 方法。

自适配具类的实例化

自适配具类中，由于其 Class<?> 对象是单独使用 cachedAdaptiveClass 缓存的，因而其实例化相对比较直接。但是当前扩展点若没有具类注解 @Adaptive ， ExtensionLoader 会使用拼接字符串的方式动态生成当前扩展点接口实现的全部代码，随后完成其编译操作获得所生成具类的元数据——一个 Class<?> 对象。

JAVA

```
private T createAdaptiveExtension() {
    try {
        return injectExtension((T) getAdaptiveExtensionClass().newInstance());
    } catch (Exception e) {
        throw new IllegalStateException("Can't create adaptive extension " + type + ",
cause: " + e.getMessage(), e);
    }
}

private Class<?> getAdaptiveExtensionClass() {
    getExtensionClasses();
    if (cachedAdaptiveClass != null) {
        return cachedAdaptiveClass;
    }

    //没有提供@Adaptive标注的具类时，动态创建代理具类
    return cachedAdaptiveClass = createAdaptiveExtensionClass();
}

private Class<?> createAdaptiveExtensionClass() {
    //生成代码
    String code = new AdaptiveClassCodeGenerator(type, cachedDefaultName).generate();

    //获取classloader
    ClassLoader classLoader = findClassLoader();

    //使用SPI获取用于编译字符串得到类的Compiler
    org.apache.dubbo.common.compiler.Compiler compiler =

ExtensionLoader.getExtensionLoader(org.apache.dubbo.common.compiler.Compiler.class)
            .getAdaptiveExtension();

    //完成字符串到Class<?>对象的转换处理
    return compiler.compile(code, classLoader);
}
```

其中 AdaptiveClassCodeGenerator 以 @SPI 注解配置的值作为默认值生成一个后缀为"$Adaptive"的代理具类，该代理具类会将扩展点接口中标注了 @Adaptive 的方法委托给通过 ExtensionLoader 获取到的当前扩展点具类实例的同名方法。

假设我们定义了如下一个扩展点接口：

JAVA

```java
package org.apache.dubbo.common.extension.ext2
@SPI
public interface EgSpi {
    @Adaptive({"key_first","key_second","key_third"})
    String echo(UrlHolder holder, String s);

    String bang(URL url, int i);

    @Adaptive
    void conn(URL url, int timeout);
}
```

经 AdaptiveClassCodeGenerator 处理后会生成下述代码，为了便于阅读，对代码进行了美化处理。

JAVA

```java
package org.apache.dubbo.common.extension.ext2;
import org.apache.dubbo.common.extension.ExtensionLoader;
public class EgSpi$Adaptive implements EgSpi {
    public String echo(UrlHolder holder, String str) {
        if (holder == null) throw new IllegalArgumentException(
                "UrlHolder argument == null");
        if (holder.getUrl() == null) throw new IllegalArgumentException(
                "UrlHolder argument getUrl() == null");

        URL url = holder.getUrl();
        String extName = url.getParameter("key_first",
            url.getParameter("key_second", url.getParameter("key_third")));

        if (extName == null)
            throw new IllegalStateException(
                    "Failed to get extension (EgSpi) name from url ("
                            + url.toString() + ") use keys([eg.spi])");
        EgSpi extension = (EgSpi) ExtensionLoader.getExtensionLoader
                (EgSpi.class).getExtension(extName);
        return extension.echo(holder, str);
    }

    public String bang(URL url, int i) {
        throw new UnsupportedOperationException(
                "The method public abstract String EgSpi.bang(URL,int)" +
                        " of interface EgSpi is not adaptive method!");
    }

    public void conn(URL urlArg, int timeout) {
        if (urlArg == null) throw new IllegalArgumentException("url == null");
        URL url = urlArg;
        String extName = url.getParameter("eg.spi", "test");
        if (extName == null)
            throw new IllegalStateException(
                    "Failed to get extension (EgSpi) name from url ("
                            + url.toString() + ") use keys([eg.spi])");
        EgSpi extension = (EgSpi) ExtensionLoader.getExtensionLoader
                (EgSpi.class).getExtension(extName);
        extension.conn(urlArg, timeout);
    }
}
```

从上述生成代码可以看出：

1. 扩展点中没有声明 `@Adaptive` 注解的方法，对应生成代理具类的方法只会简单抛错处理；
2. 针对已声明 `@Adaptive` 注解的方法，正常生成代理逻辑：
   a. Dubbo会找到首个类型为**URL**的入参_{传入配置总线}获取目标具类的实例；
   b. 如果没有类型为**URL**的入参，会试图迭代所有入参，挨个试探是否能含有 `getUrl()` 方法，若有则通过它获取到**URL**配置总线；

c. Dubbo的SPI机制中，每一个扩展点具类都会有一个唯一对应的Key键，扩展点调用方需要使用该Key键通过 `ExtensionLoader` 动态获取到扩展点实例，调用方可以借助配置总线URL传入Key键，在URL中需要有合适的配置项， `@Adaptive` 注解的值正是为此提供支持的，值可以配置多个，按顺序取用，直到首次找到可用的扩展点实例为止；

d. 如果 `@Adaptive` 没有配置值，则当前扩展点所在配置总线中配置项字符串型的Key键被设定为：取扩展点接口名，按词分割，取小写，最后以"."拼接；

e. 配置总线中若没有相应配置项，则使用由 `@SPI` 注解得到名词 `cachedDefaultName` ，如果该值为null，会因 `extName == null` 抛错；

3. 不管是直接还是间接获取的 `URL` 配置总线，均不能为 `null` ；

4. 生成类的包名和扩展点所处的包名一致；

自适配具类的源码生成

源码生成总体实现上并没有多复杂，基本原理是利用当前扩展点接口本身信息和 `@Adaptive` 注解信息生成扩展点代理具类，就其各个方法，从入参寻找到配置总线URL，以 `@Adaptive` 的值作为配置项从中取到目标扩展点具类的名称， `ExtensionLoader` 使用该名称动态地获得扩展点实例，从而最终得以将生成具类的当前委托给目标具类。

类声明

声明代理具类的实现比较简单，简单的对应关系如下：

1. `package` 语句：和扩展点同包， `type.getPackage().getName()`
2. 引入 `ExtensionLoader` 依赖： `ExtensionLoader.class.getName()`
3. 类声明： `String.format("public class %s$Adaptive implements %s", type.getSimpleName(), type.getCanonicalName())`

方法签名生成

众所周知，总体上Java的一个方法包含了方法名、出参、入参、方法体，以及抛出异常声明异常也是一种出参，其中方法名取的就是当前扩展点接口的名称—— `method.getName()` 。凡是参数，具有自己的类型，Java中可以统一用 `Class<?>` 表示，类型可能含有泛型等复杂组成，因而生成代理具类源码时需要有完整的文本表示，也即需要使用 `.getCanonicalName()` 取得，另外一个好处是由于它使用的是全名，就避免了复杂的 `import` 导入处理。

Java中的出参是单一的，直接使用 `method.getReturnType().getCanonicalName()` ，而入参和异常出参稍微复杂点，如下：

JAVA

```
//generate method arguments
private String generateMethodArguments(Method method) {
    Class<?>[] pts = method.getParameterTypes();
    return IntStream.range(0, pts.length)
                    .mapToObj(i -> String.format(CODE_METHOD_ARGUMENT,
pts[i].getCanonicalName(), i))
                    .collect(Collectors.joining(", "));
}

//generate method throws
private String generateMethodThrows(Method method) {
    Class<?>[] ets = method.getExceptionTypes();
    if (ets.length > 0) {
        String list =
Arrays.stream(ets).map(Class::getCanonicalName).collect(Collectors.joining(", "));
        return String.format(CODE_METHOD_THROWS, list);
    } else {
        return "";
    }
}

private String generateReturnAndInvocation(Method method) {
    String returnStatement = method.getReturnType().equals(void.class) ? "" : "return
";

    String args = IntStream.range(0, method.getParameters().length)
            .mapToObj(i -> String.format(CODE_EXTENSION_METHOD_INVOKE_ARGUMENT, i))
            .collect(Collectors.joining(", "));

    return returnStatement + String.format("extension.%s(%s);\n", method.getName(),
args);
}
```

方法体生成

方法体生成的代码涉及细节稍微比较多，下面挑拣几个重要的点阐述下：

先说说解决获取URL的问题，如下源码所示，大概思路是遍历所有入参，对每个入参的所有方法逐个检查，如果找到返回类型为URL且非 `static` 的 `public` 型无参 `getter` 方法，则直接：

JAVA

```java
private String generateUrlAssignmentIndirectly(Method method) {
    Class<?>[] pts = method.getParameterTypes();

    // find URL getter method
    for (int i = 0; i < pts.length; ++i) {
        for (Method m : pts[i].getMethods()) {
            String name = m.getName();
            if ((name.startsWith("get") || name.length() > 3)
                    && Modifier.isPublic(m.getModifiers())
                    && !Modifier.isStatic(m.getModifiers())
                    && m.getParameterTypes().length == 0
                    && m.getReturnType() == URL.class) {
                return generateGetUrlNullCheck(i, pts[i], name);
            }
        }
    }

    // getter method not found, throw
    throw new IllegalStateException("Failed to create adaptive class for interface " +
type.getName()
                    + ": not found url parameter or url attribute in parameters of
method " + method.getName());

}

/**
 * 1, test if argi is null
 * 2, test if argi.getXX() returns null
 * 3, assign url with argi.getXX()
 */
private String generateGetUrlNullCheck(int index, Class<?> type, String method) {
    // Null point check
    StringBuilder code = new StringBuilder();
    code.append(String.format("if (arg%d == null) throw new
IllegalArgumentException(\"%s argument == null\");\n",
            index, type.getName()));
    code.append(String.format("if (arg%d.%s() == null) throw new
IllegalArgumentException(\"%s argument %s() == null\");\n",
            index, method, type.getName(), method));

    code.append(String.format("%s url = arg%d.%s();\n", URL.class.getName(), index,
method));
    return code.toString();
}
```

NOTE

java中的访问修饰符总共有12个，分别是 pubic、private、protected、static、final、synchronized、volatile、transient、native、interface、abstract、strictfp，他们可以汇总在一个2字节的int类型变量加以表达，`0000 0000 0000 0000`，从低到高，分别各占一位，元素含有该修饰符就标记为1，否则为0，比如某个方法加了 synchronized 修饰符，那第6位便是1。Method对象中有一个 modifiers 变量，2字节的int类型，它告知外界这个方法的可见性、是否为static等。这样便可以使用高效的位操作判别方法的特性，比如检测方法是否是static的：

JAVA

```
//0x00000008 = 0000 0000 0000 1000

//The {@code int} value representing the {@code static}
modifier.
public static final int STATIC = 0x00000008;

public static boolean isStatic(int mod) {
    return (mod & STATIC) != 0;
}
```

上文已经提到对于没有提供值的 @Adaptive 方法注解，Dubbo会默认生成一个配置项，如下所示：

JAVA

```
private String[] getMethodAdaptiveValue(Adaptive adaptiveAnnotation) {
    String[] value = adaptiveAnnotation.value();
    // value is not set, use the value generated from class name as the key
    if (value.length == 0) {
        String splitName = StringUtils.camelToSplitName(type.getSimpleName(), ".");
        value = new String[]{splitName};
    }
    return value;
}
```

最后在 @Adaptive 注解中，如果出现了 "protocol"，如果其出现在前面，会优先以它作为扩展点的名称获取扩展点实例，否则只有没有在URL中找到对应配置值是才使用它。如下述代码片段所示：

```java
//①场景：@Adaptive({"protocol", "key2"})
String extName = url.getProtocol() == null ? (url.getParameter("key2")) :
url.getProtocol();

//②场景：@Adaptive({"key1", "protocol"})
String extName = url.getParameter("key1", url.getProtocol());

//③场景：@Adaptive({"key1", "protocol", "key2"})
String extName = url.getParameter("key1", url.getProtocol() == null ?
(url.getParameter("key2")) : url.getProtocol());
```

关于Dubbo SPI，最重要的一环这里没有涉及，就代码生成之后的动态编译处理。搞Java业务开发，这些比较深入的技能点，一般是没法接触到的，下文我将带大家去探访。

完结