Dubbo 系列笔记之自适应扩展机制

作者：wangning1018
原文链接：https://ld246.com/article/1594714347628
来源网站：链滴

一、引言

上一篇我们说了 Dubbo SPI 的扩展类加载过程以及 Dubbo IOC 的实现，Dubbo 的很多扩展都是基 SPI 来加载的，如 Cluster、Protocol 等等。有时，我们希望有些扩展不在框架启动的时候就加载，是想在扩展方法被调用的时候，通过动态参数按需加载。

这就产生了一个问题，我想调用扩展方法（非静态），但是扩展还没有被加载。也就是说，扩展类还有被 new 出实例，那么就不能调用其方法。想要调用方法，必须提前加载扩展类，而我们不想在框启动时就加载，死循环...

二、理解自适应扩展

对于上面的问题，Dubbo 使用自适应扩展机制解决。同时官方文档中还为我们提供了一个例子帮助解什么是自适应扩展，如下：

- 车轮制造接口 WheelMaker

```
public interface WheelMaker {
    Wheel makeWheel(URL url);
}
```

- WheelMaker 接口的自适应实现类 AdaptiveWheelMaker

```
public class AdaptiveWheelMaker implements WheelMaker {
    public Wheel makeWheel(URL url) {
        if (url == null) {
            throw new IllegalArgumentException("url == null");
        }

        // 1.从 URL 中获取 WheelMaker 名称
```

```
        String wheelMakerName = url.getParameter("Wheel.maker");
        if (wheelMakerName == null) {
            throw new IllegalArgumentException("wheelMakerName == null");
        }

        // 2.通过 SPI 加载具体的 WheelMaker
        WheelMaker wheelMaker = ExtensionLoader
            .getExtensionLoader(WheelMaker.class).getExtension(wheelMakerName);

        // 3.调用目标方法
        return wheelMaker.makeWheel(url);
    }
}
```

AdaptiveWheelMaker 做了三件事情：

1. 从给定参数中获取扩展名称
2. 通过 SPI 加载特定的扩展类
3. 调用扩展类方法返回执行结果

- 接下来，我们来看看汽车制造厂 CarMaker 接口与其实现类。

```
public interface CarMaker {
    Car makeCar(URL url);
}

public class RaceCarMaker implements CarMaker {
    WheelMaker wheelMaker;

    // 通过 setter 注入 AdaptiveWheelMaker
    public setWheelMaker(WheelMaker wheelMaker) {
        this.wheelMaker = wheelMaker;
    }

    public Car makeCar(URL url) {
        Wheel wheel = wheelMaker.makeWheel(url);
        return new RaceCar(wheel, ...);
    }
}
```

注意到 RaceCarMaker 有一个成员变量 wheelMaker ，在程序运行时我们将 AdaptiveWheelMaker 通过 setter 方法注入到 RaceCarMaker 中，这样当传入一个 URL 参数时，就可以调用 AdaptiveWheelMaker 的 makeWheel(URL url) 方法解析 URL 参数，通过 SPI 加载特定的扩展类并调用其方法放一个 Wheel 对象。

如上，就是 Dubbo 的自适应扩展机制的基本原理：**当扩展接口的方法被调用时，生成扩展代理类并过 SPI 加载具体的扩展实现，调用扩展对象的同名方法。**

也就是说，**Dubbo 自适应扩展机制解决了在调用扩展接口方法时，扩展类未被加载方法不能调用的题。**

三、源码一探

接下来，我们跟随 Dubbo 源码，看自适应扩展机制如何实现。

1. @Adaptive 注解

在文章开始就说到 Cluster 是基于 SPI 来加载的，所以看下 Cluster 接口：

```
@SPI(Cluster.DEFAULT)
public interface Cluster {
    String DEFAULT = FailoverCluster.NAME;

    /**
     * Merge the directory invokers to a virtual invoker.
     *
     * @param <T>
     * @param directory
     * @return cluster invoker
     * @throws RpcException
     */
    @Adaptive
    <T> Invoker<T> join(Directory<T> directory) throws RpcException;

    static Cluster getCluster(String name) {
        return getCluster(name, true);
    }

    static Cluster getCluster(String name, boolean wrap) {
        if (StringUtils.isEmpty(name)) {
            name = Cluster.DEFAULT;
        }
        return ExtensionLoader.getExtensionLoader(Cluster.class).getExtension(name, wrap);
    }
}
```

可以看到 Join(Directory<T> directory) 方法上面加了一个 @Adaptive 的注解，该注解如下：

```
/**
 * Provide helpful information for {@link ExtensionLoader} to inject dependency extension ins
ance.
 *
 * @see ExtensionLoader
 * @see URL
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Adaptive {
    String[] value() default {};
}
```

当 Adaptive 注解在类上时，Dubbo 不会为该类生成代理类。注解在方法（接口方法）上时，Dubbo 则会为该方法生成代理逻辑。Adaptive 注解在类上的情况很少，在 Dubbo 中，仅有两个类被 Adaptive 注解了，分别是 AdaptiveCompiler 和 AdaptiveExtensionFactory。此种情况，表示拓展的加载辑由人工编码完成。更多时候，Adaptive 是注解在接口方法上的，表示拓展的加载逻辑需由框架自生成。

通过注释我们可以知道 @Adaptive 是帮助 ExtensionLoader 注入依赖的扩展实例。

2. getAdaptiveExtension( ) 方法

跟随注释我们进入 ExtensionLoader 一探究竟，注意到与自适应拓展相关的有一个关键方法 public T getAdaptiveExtension() ：

```
public T getAdaptiveExtension() {
        // 缓存中取
        Object instance = cachedAdaptiveInstance.get();
    // 缓存未命中
        if (instance == null) {
            if (createAdaptiveInstanceError != null) {
                throw new IllegalStateException("Failed to create adaptive instance: " +
                        createAdaptiveInstanceError.toString(),
                        createAdaptiveInstanceError);
            }

            synchronized (cachedAdaptiveInstance) {
                instance = cachedAdaptiveInstance.get();
                if (instance == null) {
                    try {
                        // 创建自适应扩展实例
                        instance = createAdaptiveExtension();
                        // 加入缓存
                        cachedAdaptiveInstance.set(instance);
                    } catch (Throwable t) {
                        createAdaptiveInstanceError = t;
                        throw new IllegalStateException("Failed to create adaptive instance: " + t.toStri
g(), t);
                    }
                }
            }
        }

        return (T) instance;
    }
```

getAdaptiveExtension( ) 逻辑比较简单，首先从缓存中取，没有取到就调用 createAdaptiveExtensi n( ) 方法创建自适应扩展，然后加入缓存。

接下来我们继续看 createAdaptiveExtension( ) 方法：

```
private T createAdaptiveExtension() {
        try {
            // 获取自适应扩展类，并通过反射实例化
            return injectExtension((T) getAdaptiveExtensionClass().newInstance());
        } catch (Exception e) {
            throw new IllegalStateException("Can't create adaptive extension " + type + ", cause: "
+ e.getMessage(), e);
        }
    }
```

可以看到其中比较关键的一行代码，return injectExtension((T) getAdaptiveExtensionClass().newIn

tance());，它

执行了三个方法，下面说明：

- injectExtension()

这个方法是为了向扩展实例中注入自适应扩展依赖，在 Dubbo 中有两种形式的自适应拓展，一种是ubbo 自动生成的不需要注入；另一种是开发者手动编写的，需要调用这个方法注入依赖。

```
private T injectExtension(T instance) {

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

        try {
            for (Method method : instance.getClass().getMethods()) {
                if (!isSetter(method)) {
                    continue;
                }
                // 加 @DisableInject 注解的方法不需要自动注入扩展
                if (method.getAnnotation(DisableInject.class) != null) {
                    continue;
                }
                // 方法的返回值如果是基本类型，int/long等等，包括一些包装类型 String/Date/Boolean
等等，不是调用注入扩展的方法
                Class<?> pt = method.getParameterTypes()[0];
                if (ReflectUtils.isPrimitives(pt)) {
                    continue;
                }

                try {
                    // 获取 get 方法的属性名
                    String property = getSetterProperty(method);
                    // 通过 SPI 获取扩展实例
                    Object object = objectFactory.getExtension(pt, property);
                    if (object != null) {
                        // 调用 set 方法注入自适应扩展实例
                        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;
    }
```

- getAdaptiveExtensionClass( ) 获取自适应扩展的 Class 对象

```
private Class<?> getAdaptiveExtensionClass() {
```

```
        // 获取所有的扩展实现类，并加入缓存
        getExtensionClasses();
        // 从缓存中取
        if (cachedAdaptiveClass != null) {
            return cachedAdaptiveClass;
        }
        // 获取自适应扩展类
        return cachedAdaptiveClass = createAdaptiveExtensionClass();
    }
```

下面直接看 createAdaptiveExtensionClass() 方法：

```
 private Class<?> createAdaptiveExtensionClass() {
        // 通过 AdaptiveClassCodeGenerator 构建自适应扩展类代码
        String code = new AdaptiveClassCodeGenerator(type, cachedDefaultName).generate();
        // 获取类加载器
        ClassLoader classLoader = findClassLoader();
        // 获取编译器实现类
        org.apache.dubbo.common.compiler.Compiler compiler = ExtensionLoader.getExtension
oader(org.apache.dubbo.common.compiler.Compiler.class).getAdaptiveExtension();
        // 编译代码，生成 Class 对象
        return compiler.compile(code, classLoader);
    }
```

到这里，我们可以知道，**Dubbo 的自适应扩展机制中自己生成了自适应扩展的代理类**，如此，不需框架启动阶段就通过 SPI 加载所有的实现类，就可以在运行期通过动态参数调用扩展方法。

3. AdaptiveClassCodeGenerator 如何生成自适应扩展代码

AdaptiveClassCodeGenerator 类大约有 400 行的代码，这也是 Dubbo 自适应扩展机制的核心，下来会用较长的篇幅来说明。

3.1 AdaptiveClassCodeGenerator 构造方法

```
public AdaptiveClassCodeGenerator(Class<?> type, String defaultExtName) {
        this.type = type;
        this.defaultExtName = defaultExtName;
    }
```

其中有两个参数，type 为扩展类的接口类型，defaultExtName 为 @SPI 中指定的默认扩展。

3.2 generate() 方法

接下来，我们从 generate() 方法入手：

```
public String generate() {
        // 判断类方法是否有 @Adaptive 注解
        if (!hasAdaptiveMethod()) {
            throw new IllegalStateException("No adaptive method exist on extension " + type.get
ame() + ", refuse to create the adaptive class!");
        }
        // 类结构生成，package/import/class
        StringBuilder code = new StringBuilder();
```

```
        // package + type 所在包
        code.append(generatePackageInfo());
        // import + ExtensionLoader全限定名
        code.append(generateImports());
        // public class + type简单名称 + $Adaptive + implements + type全限定名 + {
        code.append(generateClassDeclaration());
        // 生成方法
        Method[] methods = type.getMethods();
        for (Method method : methods) {
            code.append(generateMethod(method));
        }
        code.append("}");

        if (logger.isDebugEnabled()) {
            logger.debug(code.toString());
        }
        return code.toString();
    }
```

继续以 Cluster 为例，除生成的方法外代码如下：

```
package org.apache.dubbo.rpc.cluster;
import org.apache.dubbo.common.extension.ExtensionLoader;
public class Cluster$Adaptive implements org.apache.dubbo.rpc.cluster.Cluster {
    // 省略方法代码
}
```

3.3 generateMethod(Method method) 方法

继续看 generateMethod(Method method) 如何生成方法：

```
private String generateMethod(Method method) {
        // 返回类型
        String methodReturnType = method.getReturnType().getCanonicalName();
        // 方法名
        String methodName = method.getName();
        // 生成方法内容
        String methodContent = generateMethodContent(method);
        // 参数
        String methodArgs = generateMethodArguments(method);
        // 异常
        String methodThrows = generateMethodThrows(method);
        return String.format(CODE_METHOD_DECLARATION, methodReturnType, methodName,
methodArgs, methodThrows, methodContent);
    }
```

- 生成方法内容：

```
private String generateMethodContent(Method method) {
        // 获取方法的 @Adaptive 注解
        Adaptive adaptiveAnnotation = method.getAnnotation(Adaptive.class);
        StringBuilder code = new StringBuilder(512);
        // 若注解为空则抛出异常
        if (adaptiveAnnotation == null) {
```

```
            return generateUnsupported(method);
        } else {
            // 获取 URL 参数的下标
            int urlTypeIndex = getUrlTypeIndex(method);

            // 参数列表中存在 URL 类型的参数
            if (urlTypeIndex != -1) {
                // 为空抛出异常校验，url 赋值
                code.append(generateUrlNullCheck(urlTypeIndex));
            } else {
                // 没有找到 URL 参数，调用类似 getUrl 的 getter 方法获取
                code.append(generateUrlAssignmentIndirectly(method));
            }
            // 获取 @Adaptive 注解的 value 值
            String[] value = getMethodAdaptiveValue(adaptiveAnnotation);
            // 参数总是否有 Invocation 类型的
            boolean hasInvocation = hasInvocationArgument(method);
            // Invocation 类型参数空值校验
            code.append(generateInvocationArgumentNullCheck(method));
            // ***生成拓展名获取逻辑***
            code.append(generateExtNameAssignment(value, hasInvocation));
            // check extName == null?
            code.append(generateExtNameNullCheck(value));
            // 生成使用 SPI 加载扩展类代码
            code.append(generateExtensionAssignment());

            // return statement
            code.append(generateReturnAndInvocation(method));
        }

        return code.toString();
    }
```

在这个方法中开始就执行了获取 URL 参数的逻辑。我们知道在 Dubbo 中 URL 主要作用是为扩展点传递数据，在 URL 中除了一些比较重要的值外，使用键值对的形式传递。

组成 URL 的具体参数：

- protocol：一般是 dubbo 中的各种协议 如：dubbo thrift http zk
- username/password：用户名/密码
- host/port：主机/端口
- path：接口名称
- parameters：参数键值对

回到这里讲的 Dubbo 自适应扩展机制，这里 **URL 中携带了要执行的目标扩展名称。**

3.4 生成扩展名获取逻辑方法 generateExtNameAssignment(...)

在上面生成方法内容的代码中有一个方法是 generateExtNameAssignment(String[] value, boolean hasInvocation) 。

```
private String generateExtNameAssignment(String[] value, boolean hasInvocation) {
```

```
        String getNameCode = null;
        // value 是 @Adaptive 的值，主要逻辑是生成从 URL 中获取扩展名的代码
        for (int i = value.length - 1; i >= 0; --i) {
            if (i == value.length - 1) {
                // defaultExtName 是 @SPI 中指定的默认扩展名
                if (null != defaultExtName) {
                    // 上面也说了 URL 的组成，protocol 可以直接使用 get 方法获取，其他的要从参数 m
p 中取
                    if (!"protocol".equals(value[i])) {
                        if (hasInvocation) {
                            getNameCode = String.format("url.getMethodParameter(methodName, \"%
\", \"%s\")", value[i], defaultExtName);
                        } else {
                            getNameCode = String.format("url.getParameter(\"%s\", \"%s\")", value[i], d
faultExtName);
                        }
                    } else {
                        getNameCode = String.format("( url.getProtocol() == null ? \"%s\" : url.getPro
ocol() )", defaultExtName);
                    }
                } else {
                    if (!"protocol".equals(value[i])) {
                        if (hasInvocation) {
                            getNameCode = String.format("url.getMethodParameter(methodName, \"%
\", \"%s\")", value[i], defaultExtName);
                        } else {
                            getNameCode = String.format("url.getParameter(\"%s\")", value[i]);
                        }
                    } else {
                        getNameCode = "url.getProtocol()";
                    }
                }
            } else {
                if (!"protocol".equals(value[i])) {
                    if (hasInvocation) {
                        getNameCode = String.format("url.getMethodParameter(methodName, \"%s\",
\"%s\")", value[i], defaultExtName);
                    } else {
                        getNameCode = String.format("url.getParameter(\"%s\", %s)", value[i], getNa
eCode);
                    }
                } else {
                    getNameCode = String.format("url.getProtocol() == null ? (%s) : url.getProtocol()"
 getNameCode);
                }
            }
        }

        return String.format(CODE_EXT_NAME_ASSIGNMENT, getNameCode);
    }
```

代码的分支虽然多，但是只做了一件事情，生成获取扩展名的代码。根据不同情况，生成的代码例子以直观的看下面：

```
String extName = (url.getProtocol() == null ? "dubbo" : url.getProtocol());
```

或

```
String extName = url.getMethodParameter(methodName, "loadbalance", "random");
```

亦或是

```
String extName = url.getParameter("client", url.getParameter("transporter", "netty"));
```

四、总结

上面用了较长的篇幅分析了生成扩展代理类的代码，其实只要知道两方面阅读起来很简单。 Dubbo 自适应扩展为了做什么：**在运行时动态调用扩展方法。**以及怎么做的：**生成扩展代理类。**代理类中根据 URL 获取扩展名，使用 SPI 加载扩展类，并调用同名方法，返回执行结果。