链滴

Dubbo 源码分析 —【4】SPI 扩展机制 上

作者：zsr251
原文链接：https://ld246.com/article/1543633667235
来源网站：链滴

- Dubbo采用 Microkernel + Plugin 模式，Microkernel 只负责组装 Plugin，Dubbo 自身的功能是通过扩展点实现的，也就是 Dubbo 的所有功能点都可被用户自定义扩展所替换。
- 采用 URL 作为配置信息的统一格式，所有扩展点都通过传递 URL 携带配置信息。

来源

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

JDK 标准的 SPI

详细请参考：Java中SPI机制深入及源码解析

- 主要类： ServiceLoader 默认从 META-INF/services/ 路径下读取文件
- 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，但如果没用上也载，会很浪费资源
- 如果扩展点加载失败，连扩展点的名称都拿不到了

部分核心代码

```
// 判断是否有下一个扩展实现
private boolean hasNextService() {
    if (nextName != null) {
        return true;
    }
    if (configs == null) {
        try {
            String fullName = PREFIX + service.getName();
            // 根据全路径获取配置的扩展实现
            if (loader == null)
                configs = ClassLoader.getSystemResources(fullName);
            else
                configs = loader.getResources(fullName);
        } catch (IOException x) {
            fail(service, "Error locating configuration files", x);
        }
    }
    while ((pending == null) || !pending.hasNext()) {
        if (!configs.hasMoreElements()) {
            return false;
        }
        pending = parse(service, configs.nextElement());
    }
    nextName = pending.next();
    return true;
}
// 获得下一个扩展实现
private S nextService() {
    if (!hasNextService())
        throw new NoSuchElementException();
    String cn = nextName;
```

```
    nextName = null;
    Class<?> c = null;
    try {
        // 根据全限定名加载类
        c = Class.forName(cn, false, loader);
    } catch (ClassNotFoundException x) {
        fail(service,
                "Provider " + cn + " not found");
    }
    if (!service.isAssignableFrom(c)) {
        fail(service,
                "Provider " + cn  + " not a subtype");
    }
    try {
        // 判断是否是 SPI 接口的实现，如果是则加入到提供者列表
        S p = service.cast(c.newInstance());
        providers.put(cn, p);
        return p;
    } catch (Throwable x) {
        fail(service,
                "Provider " + cn + " could not be instantiated",
                x);
    }
    throw new Error();          // This cannot happen
}
```

DUBBO 加强的 SPI

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=扩实现类全限定名，多个实现类用换行符分隔。

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/com.alibaba.ubbo.rpc.Protocol，内容为：

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

实现类内容：

```
package com.alibaba.xxx;

import com.alibaba.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" />
```

示例：https://www.jianshu.com/p/dc616814ce98

https://www.jianshu.com/p/bc523348f519

源码分析

可参考：Dubbo中SPI扩展机制详解

@SPI 、@Adaptive 、@Activate 作用

- @SPI （注解在类上）：@SPI 注解标识了接口是一个扩展点， 属性 value 用来指定默认适配扩点的名称。
- @Activate （注解在类型和方法上）： @Activate 注解在扩展点的实现类上，表示了一个扩展被获取到的的条件，符合条件就被获取，不符合条件就不获取，根据 @Activate 中的 group 、 valu 属性来过滤 。具体参考 ExtensionLoader 中的 getActivateExtension 函数。 可参考 Dubbo SPI之ctivate详解
- @Adaptive （注解在类型和方法上）：
  - 注解在类上， 这个类就是缺省的适配扩展。
  - 注解在接口的方法上，ExtensionLoader根据接口定义动态的生成适配器代码，并实例化这个生的动态类。
  - 可参考 Dubbo SPI之Adaptive详解

核心类 ExtensionLoader

类似与Java的ServiceLoader

最简单使用例子：

```
ExtensionLoader.getExtensionLoader(CompatibleExt.class).getExtension("impl1")
```

第一步 getExtensionLoader

根据 SPI 接口创建出一个ExtensionLoader实例，如果本地缓存中已存在则使用缓存的，如果不存在实例化一个，与接口相关联放入缓存。

```
// 每一个 SPI 扩展有一个对应的加载类
public static <T> ExtensionLoader<T> getExtensionLoader(Class<T> type) {
    if (type == null)
        throw new IllegalArgumentException("Extension type == null");
    if (!type.isInterface()) {
        throw new IllegalArgumentException("Extension type(" + type + ") is not interface!");
    }
    // 没有使用 @SPI 做注解的接口
    if (!withExtensionAnnotation(type)) {
        throw new IllegalArgumentException ...
```

```
    }
    // 从本地缓存中加载指定 SPI 的加载类
    ExtensionLoader<T> loader = (ExtensionLoader<T>) EXTENSION_LOADERS.get(type);
    if (loader == null) {
        // 如果已存在 则不进行替换，不存在 则存入
        EXTENSION_LOADERS.putIfAbsent(type, new ExtensionLoader<T>(type));
        loader = (ExtensionLoader<T>) EXTENSION_LOADERS.get(type);
    }
    return loader;
}
// 构造方法
private ExtensionLoader(Class<?> type) {
    this.type = type;
    // objectFactory是一个 ExtensionFactory 类型的属性，主要用于加载需要注入的类型的实现
    // objectFactory 主要用在注入那一步，详细说明见注入时候的说明
    // getAdaptiveExtension 方法获取一个运行时自适应的扩展类型 详细介绍会在下文
    // 这里记住非 ExtensionFactory 类型的返回的都是一个AdaptiveExtensionFactory
    objectFactory = (type == ExtensionFactory.class ? null : ExtensionLoader.getExtensionLoade
(ExtensionFactory.class).getAdaptiveExtension());
}
```

第二步 getExtension

根据名称获得扩展实现

```
// 根据名称获得扩展实现
public T getExtension(String name) {
    if (name == null || name.length() == 0)
        throw new IllegalArgumentException("Extension name == null");

    // 如果 名称是 true ，则获得默认的扩展实现
    if ("true".equals(name)) {
        // 如果 @SPI 指定了默认扩展名 则设置 cachedDefaultName
        // cachedClasses 存放 指定路径下名称对应的扩展实现
        // 如果 cachedDefaultName 没有设置，则返回 null
        return getDefaultExtension();
    }
    // 从本地缓存中取一个 实例
    // Holder 在多线程中 防并发
    Holder<Object> holder = cachedInstances.get(name);
    if (holder == null) {
        cachedInstances.putIfAbsent(name, new Holder<Object>());
        holder = cachedInstances.get(name);
    }
    // 容器中 取实例 如果存在则返回 不存在则加载创建
    // 使用二次校验 Holder 中使用了 volatile 防止指令重排引起的并发问题
    Object instance = holder.get();
    if (instance == null) {
        synchronized (holder) {
            instance = holder.get();
            if (instance == null) {
                // 创建真正实例
                instance = createExtension(name);
                holder.set(instance);
```

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

第三步 createExtension

```
// 创建扩展实例
private T createExtension(String name) {
    // 根据名称获得扩展实现类
    // getExtensionClasses  先看下文分析
    Class<?> clazz = getExtensionClasses().get(name);
    // 未找到实现的扩展类 则抛出异常
    if (clazz == null) {
        throw findException(name);
    }
    try {
        T instance = (T) EXTENSION_INSTANCES.get(clazz);
        // 如果本地缓存中不存在 则实例化后放入本地缓存
        if (instance == null) {
            EXTENSION_INSTANCES.putIfAbsent(clazz, clazz.newInstance());
            instance = (T) EXTENSION_INSTANCES.get(clazz);
        }
        // 注入
        injectExtension(instance);
        Set<Class<?>> wrapperClasses = cachedWrapperClasses;
        if (wrapperClasses != null && !wrapperClasses.isEmpty()) {
            for (Class<?> wrapperClass : wrapperClasses) {
                // 使用包装类包装实例 进行注入
                instance = injectExtension((T) wrapperClass.getConstructor(type).newInstance(insta
ce));
            }
        }
        return instance;
    } catch (Throwable t) {
        throw new IllegalStateException ...
    }
}
```

getExtensionClasses 的实现是为了加载指定路径的文件配置

```
private Map<String, Class<?>> getExtensionClasses() {
    Map<String, Class<?>> classes = cachedClasses.get();
    // 如果本地 缓存中不存在 则初始化 cachedClasses
    // 同样使用的是二次校验实例化
    if (classes == null) {
        synchronized (cachedClasses) {
            classes = cachedClasses.get();
            if (classes == null) {
                classes = loadExtensionClasses();
                cachedClasses.set(classes);
            }
        }
```

```
    }
    return classes;
}

// 加载 SPI 接口 ， 获得配置中的 实现
private Map<String, Class<?>> loadExtensionClasses() {
    ...
            //  设置 @ SPI 指定的默认名称
            if (names.length == 1) cachedDefaultName = names[0];
    ...
    // 加载目录 下的约定文件内容 META-INF/services/ 、 META-INF/dubbo/ 、 META-INF/dub
o/internal/
    // loadDirectory --> loadResource --> loadClass
    Map<String, Class<?>> extensionClasses = new HashMap<String, Class<?>>();
    loadDirectory(extensionClasses, DUBBO_INTERNAL_DIRECTORY, type.getName());
    loadDirectory(extensionClasses, DUBBO_INTERNAL_DIRECTORY, type.getName().replace("o
g.apache", "com.alibaba"));
    ...
    return extensionClasses;
}
//  加载类
private void loadClass(Map<String, Class<?>> extensionClasses, java.net.URL resourceURL, Cl
ss<?> clazz, String name) throws NoSuchMethodException {
    // 判断一个类Class1和另一个类Class2是否相同或是另一个类的超类或接口
    // 与 instanceof 不同之处在于 instanceof 第一个参数需要是实例 isAssignableFrom   可以用类
断
    if (!type.isAssignableFrom(clazz)) {
        throw new IllegalStateException ...
    }
    // 是否 被 @Adaptive 注解 被 @Adaptive 注解的实现类只能有一个，否则抛异常
    if (clazz.isAnnotationPresent(Adaptive.class)) {
        if (cachedAdaptiveClass == null) {
            cachedAdaptiveClass = clazz;
        } else if (!cachedAdaptiveClass.equals(clazz)) {
            throw new IllegalStateException ...
        }
    } else
    // 是否是包装类， 放入本地缓存
    if (isWrapperClass(clazz)) {
        Set<Class<?>> wrappers = cachedWrapperClasses;
        if (wrappers == null) {
            cachedWrapperClasses = new ConcurrentHashSet<Class<?>>();
            wrappers = cachedWrapperClasses;
        }
        wrappers.add(clazz);
    } else {
        clazz.getConstructor();
        // 如果配置中没有指定别名，则查看实现类上 @Extension 注解指定的别名 否则抛异常
        if (name == null || name.length() == 0) {
            name = findAnnotationName(clazz);
            if (name.length() == 0) {
                throw new IllegalStateException ...
            }
        }
```

```
        //  多个别名分割
        String[] names = NAME_SEPARATOR.split(name);
        if (names != null && names.length > 0) {
            Activate activate = clazz.getAnnotation(Activate.class);
            // 如果含有 @Activate 注解，则放入本地缓存中
            if (activate != null) {
                cachedActivates.put(names[0], activate);
            } else {
                ...
            }
            for (String n : names) {
                // 根据实现类找别名 只能存在一个
                if (!cachedNames.containsKey(clazz)) {
                    cachedNames.put(clazz, n);
                }
                Class<?> c = extensionClasses.get(n);
                if (c == null) {
                    extensionClasses.put(n, clazz);
                } else if (c != clazz) {
                    throw new IllegalStateException ...
                }
            }
        }
    }
}
```

injectExtension 根据 set 方法注入

```
private T injectExtension(T instance) {
    try {
        if (objectFactory != null) {
            for (Method method : instance.getClass().getMethods()) {
                        // 必须是 set 方法
                if (method.getName().startsWith("set")
                        // 方法必须只有一个参数
                        && method.getParameterTypes().length == 1
                        // set 方法的修饰符 必须是 public
                        && Modifier.isPublic(method.getModifiers())) {
                    // 获得参数类型
                    Class<?> pt = method.getParameterTypes()[0];
                    try {
                        // 获得属性的名称
                        String property = method.getName().length() > 3 ? method.getName().substri
g(3, 4).toLowerCase() + method.getName().substring(4) : "";
                        Object object = objectFactory.getExtension(pt, property);
                        if (object != null) {
                            method.invoke(instance, object);
                        }
                    } catch (Exception e) {
                        logger.error("fail to inject via method " + method.getName()
                                + " of interface " + type.getName() + ": " + e.getMessage(), e);
                    }
                }
            }
```

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

objectFactory.getExtension 中 objectFactory 是私有构造方法中实例化的

```
objectFactory = (type == ExtensionFactory.class ? null : ExtensionLoader.getExtensionLoader(
xtensionFactory.class).getAdaptiveExtension());
```

具体的方法内容分析，在下一篇

至此关于简单使用分析完毕。