

14 | SPI 机制：Dubbo的SPI比JDK的SPI好在哪里？

2023-01-18 何辉 来自北京



天下无鱼

<https://shikey.com/>

课程介绍 >

《Dubbo源码剖析与实战》



讲述：何辉

时长 13:46 大小 12.58M



你好，我是何辉。今天我们来深入研究 Dubbo 源码的第三篇，SPI 机制。

SPI，英文全称是 **Service Provider Interface**，按每个单词翻译就是：服务提供接口。很多开发底层框架经验比较少的人，可能压根就没听过这个 SPI 机制，我们先简单了解一下。

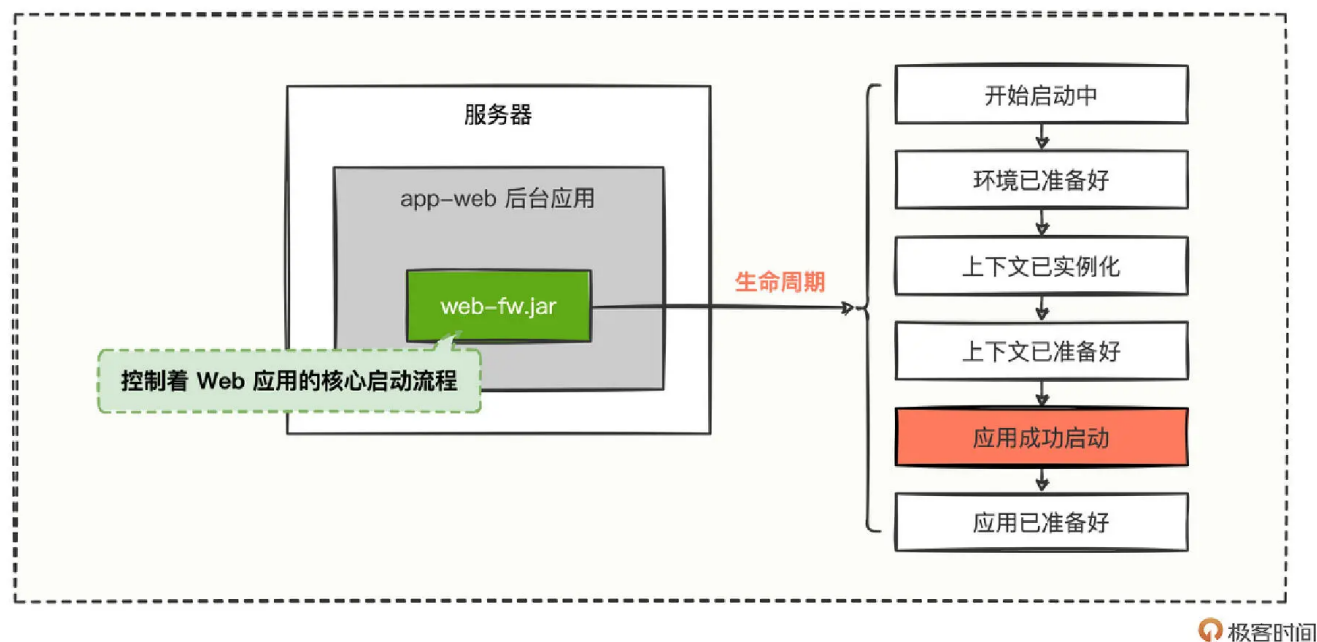
这里的“服务”泛指任何一个可以提供服务的功能、模块、应用或系统，这些“服务”在设计接口或规范体系时，往往会预留一些比较关键的口子或者扩展点，让调用方按照既定的规范去自由发挥实现，而这些所谓的“比较关键的口子或者扩展点”，我们就叫“服务”提供的“接口”。

比较常见的 SPI 机制是 JDK 的 SPI，你可能听过，其实 Dubbo 中也有这样的机制，在前面章节中，我们接触过利用 Dubbo 的 SPI 机制将过滤器的类路径配置到资源目录（resources）下，那为什么有了 JDK 的 SPI 后，还需要有 Dubbo 的 SPI 呢？究竟 Dubbo 的 SPI 比 JDK 的 SPI 好在哪里呢？

带着这个问题，我们开始今天的学习。

SPI 是怎么来的

首先要明白 SPI 到底能用来做什么，我们还是结合具体的应用场景来思考：



app-web 后台应用引用了一款开源的 web-fw.jar 插件，这个插件的作用就是辅助后台应用的启动，并且控制 Web 应用的核心启动流程，也就是说这个插件控制着 Web 应用启动的整个生命周期。

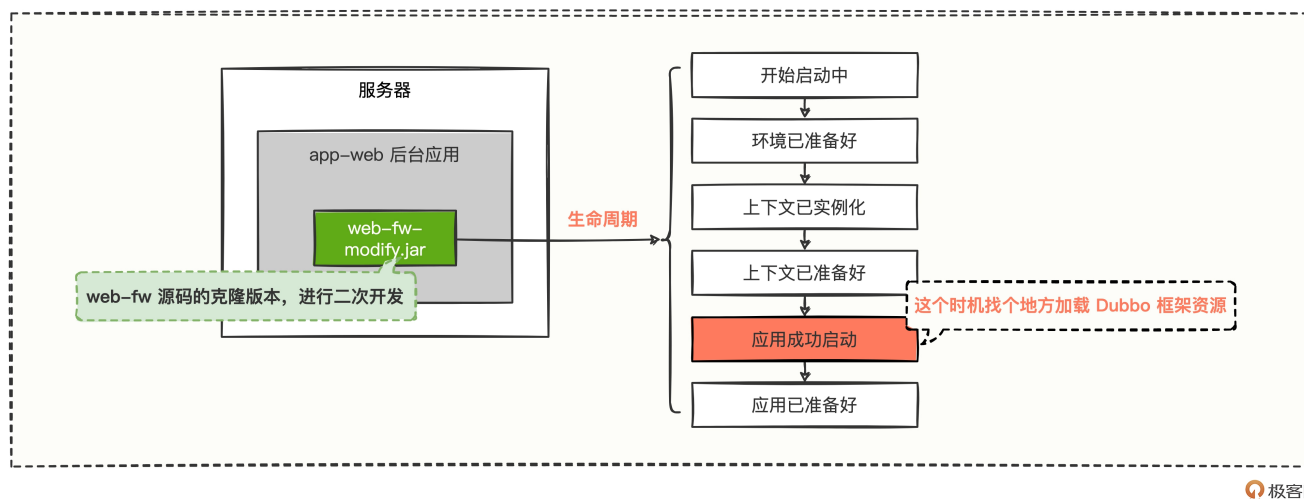
现在，有这样一个需求，在 Web 应用成功启动的时刻，我们需要预加载 Dubbo 框架的一些资源，你会怎么做呢？

你可能会说，这有什么难的。直接去插件翻代码，找到插件生命周期的“应用成功启动”时刻的代码，然后选个合适的位置，写上一段代码来预加载 Dubbo 框架的一些资源，应该可以解决问题了。

思维严谨些，我们看这个需求的背景，开源的第三方插件是一个 jar 包，是不能编辑的，就算在 IDE 编码工具中想办法编辑插件，也顶多只是编辑第三方插件的源码包（web-fw-source.jar），好像根本没啥用。

既然插件（web-fw.jar）不能为项目所用，如果替换插件呢？

可是短时间内，我们难以找到可替代的且对工程代码无侵入改造的插件，难不成要把插件（web-fw.jar）的源码下载下来进行二次开发改造么？就像这样：



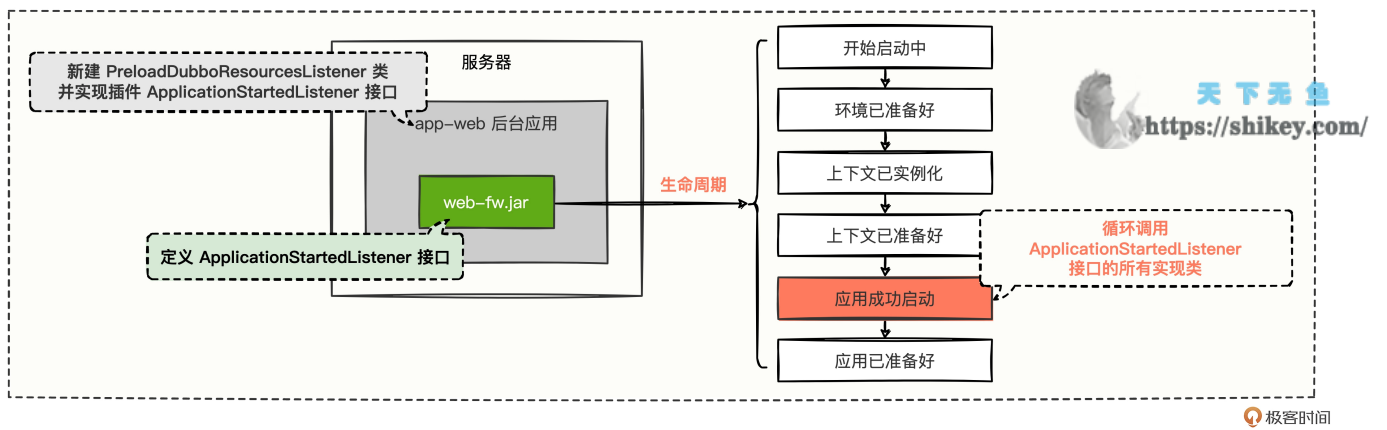
把原来的插件替换为了现在的二次开发插件（web-fw-modify.jar），然后在应用启动成功的地方加载 Dubbo 框架资源。

改造倒不会有什么大的阻碍，但是在工程中，凡是想二次改造开源，不但要考虑开发人员驾驭底层框架的抽象封装能力，还得考虑后续的维护和迭代成本。

比如，如果二次开发的插件（web-fw-modify.jar）发现了一些历史的重大漏洞问题或严重性能问题，总是需要有人维护的，这又得花费大量的人力和时间去研究源码插件。即使解决了插件的漏洞和性能问题，可是市场上的插件（web-fw.jar）却在不断更新增强，新特性越来越多，难道我们又要把新增的差异代码拷贝到二次开发的插件（web-fw-modify.jar）中？

所以，二次开发这个插件后，不但后续工作量没完没了，还跟不上市场上的迭代速度，徒增很多和业务无关的烦恼。不可行。

如果给开发该插件的开源团队提需求呢？好像有点不切实际，像这种企业定制化诉求，人家根本不会搭理。即使搭理了，为了讲究通用性，开源团队也只会提供一个口子，定义一种规范约束，给上层开发人员实现该口子做一些定制化逻辑，一般会这样做：



然后配备代码：

复制代码

```
1 ///////////////////////////////////////////////////
2 // web-fw.jar 插件的启动类，在“应用成功启动”时刻提供一个扩展口子
3 ///////////////////////////////////////////////////
4 public class WebFwBootApplication {
5     // web-fw.jar 插件的启动入口
6     public static void run(Class<?> primarySource, String... args) {
7         // 开始启动中，此处省略若干行代码...
8         // 环境已准备好，此处省略若干行代码...
9         // 上下文已实例化，此处省略若干行代码...
10        // 上下文已准备好，此处省略若干行代码...
11
12        // 应用成功启动
13        onCompleted();
14
15        // 应用已准备好，此处省略若干行代码...
16    }
17
18    // 应用成功启动时刻，提供一个扩展口子
19    private static void onCompleted() {
20        // 加载 ApplicationStartedListener 接口的所有实现类
21        ServiceLoader<ApplicationStartedListener> loader =
22            ServiceLoader.load(ApplicationStartedListener.class);
23        // 遍历 ApplicationStartedListener 接口的所有实现类，并调用里面的 onCompleted
24        Iterator<ApplicationStartedListener> it = loader.iterator();
25        while (it.hasNext()){
26            // 获取其中的一个实例，并调用 onCompleted 方法
27            ApplicationStartedListener instance = it.next();
28            instance.onCompleted();
29        }
30    }
31 }
32
33 ///////////////////////////////////////////////////
34 // web-fw.jar 插件的“应用启动成功的监听器接口”，定制一种接口规范
```

```

35 ///////////////////////////////////////////////////
36 public interface ApplicationStartedListener {
37     // 触发完成的方法
38     void onCompleted();
39 }
40
41 ///////////////////////////////////////////////////
42 // app-web 后台应用的启动类代码
43 ///////////////////////////////////////////////////
44 public class Dubbo14JdkSpiApplication {
45     public static void main(String[] args) {
46         // 模拟 app-web 调用 web-fw 框架启动整个后台应用
47         WebFwBootApplication.run(Dubbo14JdkSpiApplication.class, args);
48     }
49 }
50
51 ///////////////////////////////////////////////////
52 // app-web 后台应用的资源目录文件
53 // 路径为: /META-INF/services/com.hmilyylimh.cloud.jdk.spi.ApplicationStartedLis
54 ///////////////////////////////////////////////////
55 com.hmilyylimh.cloud.jdk.spi.PreloadDubboResourcesListener

```



代码中定义了一个应用启动成功的监听器接口（`ApplicationStartedListener`），接着 `app-web` 自定义一个预加载 Dubbo 资源监听器（`PreloadDubboResourcesListener`）来实现该接口。

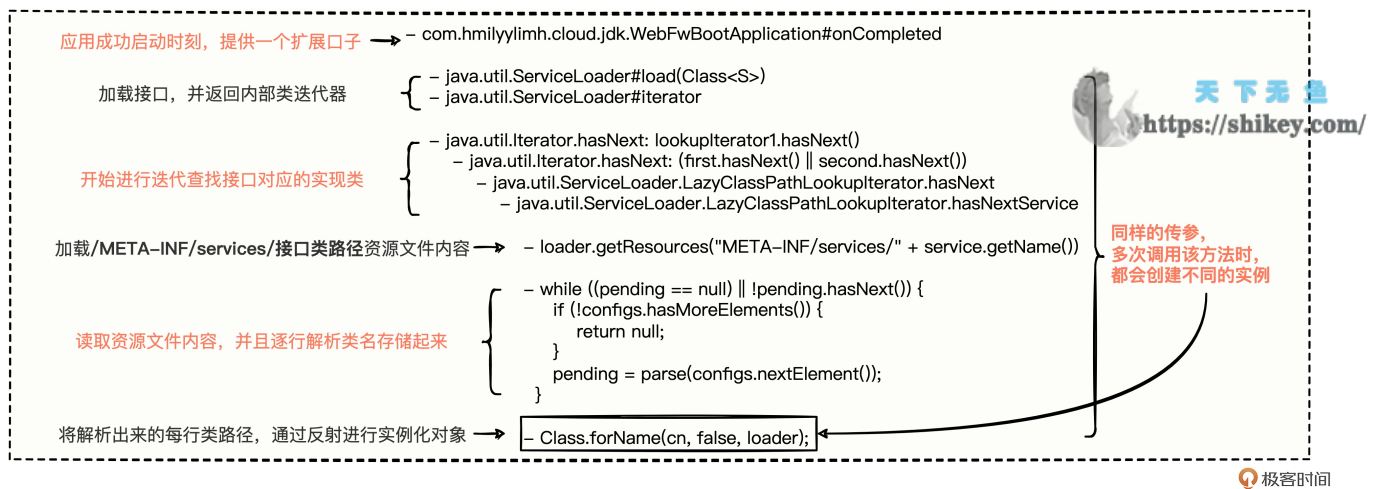
在插件应用成功启动的时刻，会寻找 `ApplicationStartedListener` 接口的所有实现类，并将所有实现类全部执行一遍，这样，插件既提供了一种子口的规范约束，又能满足业务诉求在应用成功启动时刻做一些事情。

其实插件在指定标准接口规范的这件事情上，就是 `SPI` 的思想体现，只不过是 `JDK` 通过 `ServiceLoader` 实现了这套思想，也就是我们耳熟能详的 `JDK SPI` 机制。

JDK SPI

好奇的你，一定想知道，这么神乎其神的 `JDK SPI` 底层到底是怎么实现的？为什么就能通过底层逻辑直接驱动上层按照规范进行编码呢？

我们可以到 `ServiceLoader` 里面看个究竟，同样的，我也总结了大致的核心代码流程逻辑：



源码流程主要有三块。

- 第一块, 将接口传入到 `ServiceLoader.load` 方法后, 得到了一个内部类的迭代器。
- 第二块, 通过调用迭代器的 `hasNext` 方法, 去读取“/META-INF/services/ 接口类路径”这个资源文件内容, 并逐行解析出所有实现类的类路径。
- 第三块, 将所有实现类的类路径通过“`Class.forName`”反射方式进行实例化对象。

在跟踪源码的过程中, 我还发现了一个问题, 当我们使用 `ServiceLoader` 的 `load` 方法执行多次时, 会不断创建新的实例对象。你可以这样编写代码验证:

复制代码

```
1 public static void main(String[] args) {  
2     // 模拟进行 3 次调用 load 方法并传入同一个接口  
3     for (int i = 0; i < 3; i++) {  
4         // 加载 ApplicationStartedListener 接口的所有实现类  
5         ServiceLoader<ApplicationStartedListener> loader  
6             = ServiceLoader.load(ApplicationStartedListener.class);  
7         // 遍历 ApplicationStartedListener 接口的所有实现类, 并调用里面的 onCompleted  
8         Iterator<ApplicationStartedListener> it = loader.iterator();  
9         while (it.hasNext()){  
10            // 获取其中的一个实例, 并调用 onCompleted 方法  
11            ApplicationStartedListener instance = it.next();  
12            instance.onCompleted();  
13        }  
14    }  
15 }
```


代码中，尝试调用 3 次 `ServiceLoader` 的 `load` 方法，并且每一次传入的都是同一个接口，运行编写好的代码，打印出如下信息：



 复制代码

```
1 预加载 Dubbo 框架的一些资源, com.hmilyylimh.cloud.jdk.spi.PreloadDubboResourcesList
2 预加载 Dubbo 框架的一些资源, com.hmilyylimh.cloud.jdk.spi.PreloadDubboResourcesList
3 预加载 Dubbo 框架的一些资源, com.hmilyylimh.cloud.jdk.spi.PreloadDubboResourcesList
```

打印出来的日志信息，验证了我们的猜想，每次调用 `load` 方法传入同一个接口的话，打印出来的引用地址都不一样，说明创建出了多个实例对象。

你可能会问，**创建出多个实例对象，有什么问题呢？**

如果同一个接口使用 `load` 方法的次数非常少，而且每次调用 `load` 方法创建出新的实例对象并不影响你的业务逻辑，其实问题也不大，顶多就是占用了一点内存，这点毛毛雨的内存占用可以忽略不计。

但是，如果调用 `load` 方法的频率比较高，那每调用一次其实就在做读取文件 -> 解析文件 -> 反射实例化这几步，不但会影响磁盘 IO 读取的效率，还会明显增加内存开销，我们并不想看到。

而且 `load` 方法，每次在调用时想拿到其中一个实现类，使用起来也非常不舒服，因为我们不知道想要的实现类，在迭代器的哪个位置，只有遍历完所有的实现类，才能找到想要的那个。假如项目中，有很多业务逻辑都需要获取指定的实现类，那将会充斥着各色各样针对 `load` 进行遍历的代码并比较，无形中，我们又悄悄产生了很多雷同代码。

JDK SPI 的问题

所以，JDK 的 SPI 创建出多个实例对象，总结起来有两个问题。

- 问题一，使用 `load` 方法频率高，容易影响 IO 吞吐和内存消耗。
- 问题二，使用 `load` 方法想要获取指定实现类，需要自己进行遍历并编写各种比较代码。

我们看看如何解决。

针对问题一，引发的关键在于大量的磁盘 IO 操作以及大量的实例对象产生，那有什么办法可以降低一下磁盘的频繁操作和对象的大量产生呢？



在“[缓存操作](#)”那一讲中也有方法被大量调用，我们的尝试是看看是否可以缓存起来。有 N 次调用，如果第一次通过读取文件、解析文件、反射实例化拿到接口的所有实现类并缓存起来，后面 N - 1 次就可以直接从缓存读取，大大降低了各种耗时的操作，性能有质的提升。

针对问题二，每次需要遍历找到想要的实现类，说白了其实就是 $O(n)$ 的时间复杂度，那有啥办法可以降低到 $O(1)$ 的时间复杂度呢？

从 $O(n)$ 降低到 $O(1)$ ，如果你对算法有过一定了解，想必也想到了，可以以空间换时间，叠加哈希算法进行快速寻址查找，基本上就可以做到了。那回忆下，Java 基础知识中哪些常用的工具类，可以辅助我们做到 $O(1)$ 检索复杂度呢？

没错，List 的 index 查找、Map 的 hash 查找，都可以做到 $O(1)$ 检索复杂度，这两个集合工具类，我们平时经常用，但都没发现有这么神奇的效果。

那 List 和 Map 两个工具，这里该用哪个呢？

这个好区分，需要从一个接口的所有实现类中，查找某个实现类，既然要寻找，那肯定不知道要找的实现类在第几个位置，因此 List 不可行，剩下的就只有 Map 了，Map 支持通过不同的一个对象获取另外一个对象，类似通过别名获取另外一个对象，比较符合常规查找操作。

好，到这里，我们梳理下两个问题的结论。

- 结论一，增加缓存，来降低磁盘 IO 访问以及减少对象的生成。
- 结论二，使用 Map 的 hash 查找，来提升检索指定实现类的性能。

Dubbo SPI

于是，为了弥补我们分析的 JDK SPI 的不足，Dubbo 也定义出了自己的一套 SPI 机制逻辑，既要通过 $O(1)$ 的时间复杂度来获取指定的实例对象，还要控制缓存创建出来的对象，做到按需加载获取指定实现类，并不会像 JDK SPI 那样一次性实例化所有实现类。

正因为有了这些改进，Dubbo 设计出了一个 ExtensionLoader 类，实现了 SPI 思想，也被称为 Dubbo SPI 机制。



在代码层面使用起来也很方便，你看这里的代码：

复制代码

```
1  //////////////////////////////////////
2  // Dubbo SPI 的测试启动类
3  //////////////////////////////////////
4  public class Dubbo14DubboSpiApplication {
5      public static void main(String[] args) {
6          ApplicationModel applicationModel = ApplicationModel.defaultModel();
7          // 通过 Protocol 获取指定像 ServiceLoader 一样的加载器
8          ExtensionLoader<IDemoSpi> extensionLoader = applicationModel.getExtensi
9
10         // 通过指定的名称从加载器中获取指定的实现类
11         IDemoSpi customSpi = extensionLoader.getExtension("customSpi");
12         System.out.println(customSpi + ", " + customSpi.getDefaultPort());
13
14         // 再次通过指定的名称从加载器中获取指定的实现类，看看打印的引用是否创建了新对象
15         IDemoSpi customSpi2 = extensionLoader.getExtension("customSpi");
16         System.out.println(customSpi2 + ", " + customSpi2.getDefaultPort());
17     }
18 }
19
20 //////////////////////////////////////
21 // 定义 IDemoSpi 接口并添加上了 @SPI 注解，
22 // 其实也是在定义一种 SPI 思想的规范
23 //////////////////////////////////////
24 @SPI
25 public interface IDemoSpi {
26     int getDefaultPort();
27 }
28
29 //////////////////////////////////////
30 // 自定义一个 CustomSpi 类来实现 IDemoSpi 接口
31 // 该 IDemoSpi 接口被添加上了 @SPI 注解，
32 // 其实也是在定义一种 SPI 思想的规范
33 //////////////////////////////////////
34 public class CustomSpi implements IDemoSpi {
35     @Override
36     public int getDefaultPort() {
37         return 8888;
38     }
39 }
40
41 //////////////////////////////////////
42 // 资源目录文件
43 // 路径为：/META-INF/dubbo/internal/com.hmilyylimh.cloud.dubbo.spi.IDemoSpi
```

```
44 ///////////////////////////////////////////////////  
45 customSpi=com.hmilyylimh.cloud.dubbo.spi.CustomSpi
```



看这里的使用，主要有三大步骤。

- 第一，定义一个 `IDemoSpi` 接口，并在该接口上添加 `@SPI` 注解。
- 第二，定义一个 `CustomSpi` 实现类来实现该接口，然后通过 `ExtensionLoader` 的 `getExtension` 方法传入指定别名来获取具体的实现类。
- 最后，在“/META-INF/services/com.hmilyylimh.cloud.dubbo.spi.IDemoSpi”这个资源文件中，添加实现类的类路径，并为类路径取一个别名（`customSpi`）。

然后运行这段简短的代码，打印出日志。

复制代码

```
1 com.hmilyylimh.cloud.dubbo.spi.CustomSpi@143640d5, 8888  
2 com.hmilyylimh.cloud.dubbo.spi.CustomSpi@143640d5, 8888
```

从日志中可以看到，再次通过别名去获取指定的实现类时，打印的实例对象的引用是同一个，说明 `Dubbo` 框架做了缓存处理。而且整个操作，我们只通过一个简单的别名，就能从 `ExtensionLoader` 中拿到指定实现类，确实简单方便。

当你了解了 `SPI` 的思想，也在代码层面感受了它的功效，至于如何应用，就非常清晰了，主要是通过定制底层规范接口，在不同的业务场景，封装底层逻辑不变性，提供扩展点给到上层应用做不同的自定义开发实现，既可以用来提供框架扩展，也可以用来替换组件。

总结

今天，我们从一个 `Web` 应用预加载 `Dubbo` 框架资源的案例开始，思考如何实现需求。

从修改源码插件，到让开源团队提供扩展口子，逐渐理解了 `SPI` 思想，并且通过使用 `JDK` 中的 `ServiceLoader` 关键类实现了 `SPI` 思想，也就是 `JDK SPI` 机制。

`JDK SPI` 的使用三部曲。

- 首先，定义一个接口。
- 然后，定义一些类来实现该接口，并将多个实现类的类路径添加到“/META-INF/services/ 接口类路径”文件中。
- 最后，使用 `ServiceLoader` 的 `load` 方法加载接口的所有实现类。

但 JDK SPI 在高频调用时，可能会出现磁盘 IO 吞吐下降、大量对象产生和查询指定实现类的 $O(n)$ 复杂度等问题，采用缓存 + Map 的组合方式来解决，就是 Dubbo SPI 的核心思想。

Dubbo SPI 的使用步骤三部曲：

- 首先，同样也是定义一个接口，但是需要为该接口添加 `@SPI` 注解。
- 然后，定义一些类来实现该接口，并将多个实现类的类路径添加到“/META-INF/services/ 接口类路径”文件中，并在文件中为每个类路径取一个别名。
- 最后，使用 `ExtensionLoader` 的 `getExtension` 方法就能拿到指定的实现类使用。

思考题

了解了 JDK SPI 和 Dubbo SPI，还有 Spring SPI，留 2 个小作业给你。

- 研究下 Dubbo SPI 的底层加载逻辑是怎样的。
- 总结下 Spring SPI 的使用步骤是怎样的。

期待看到你的思考，如果觉得今天的内容对你有帮助，也欢迎分享给身边的朋友一起讨论。我们下一讲见。

13 思考题参考

上一期的问题是让你尝试研究下 Spring 与 Mybatis 是怎么有机结合起来使用的。

1. 通过

`org.springframework.context.annotation.ClassPathBeanDefinitionScanner`，找到关于 mybatis 的实现类
`org.mybatis.spring.mapper.ClassPathMapperScanner`。

2. 在 `org.mybatis.spring.mapper.ClassPathMapperScanner#doScan` 中通过调用 `super.doScan`，让 Spring 帮忙扫描出一堆的 `BeanDefinition` 集合，并且修改 `BeanDefinition` 集合。



3. 通过


`org.springframework.beans.factory.support.AbstractBeanDefinition#setBeanClass` 设置 `org.mybatis.spring.mapper.MapperFactoryBean` 类，目的是方便将来创建代理类。

4. 在 `org.mybatis.spring.mapper.MapperFactoryBean#getObject` 方法中创建核心代理类 `org.apache.ibatis.binding.MapperProxy`。

5. 最后在 `org.apache.ibatis.binding.MapperProxy#invoke` 方法中，`SqlSession` 处理业务代码增删改查的 SQL 业务逻辑。

到这里，我们就从源码层面弄明白了 Spring 与 Mybatis 是怎么有机结合起来使用的。

分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享

 赞 3  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 13 | 集成框架：框架如何与Spring有机结合？

下一篇 15 | Wrapper机制：Wrapper是怎么降低调用开销的？

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。