



黑客派

# JAVA 拾遗 -- 关于 SPI 机制

作者: [gentoo666](#)

原文链接: <https://hacpai.com/article/1513580723264>

来源网站: [黑客派](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

<p>DK 提供的 SPI(Service Provider Interface)机制，可能很多人不太熟悉，因为这个机制是针对商或者插件的，也可以在一些框架的扩展中看到。其核心类 <code>java.util.ServiceLoader</code> 可以在 jdk1.8 的文档中看到详细的介绍。虽然不太常见，但并不代表它不常用，恰恰相反，你无时无刻不在用它。玄乎了，莫急，思考一下你的项目中是否有用到第三方日志包，是否有用到数据库驱动其实这些都和 SPI 有关。再来思考一下，现代的框架是如何加载日志依赖，加载数据库驱动的，你可能会对 class.forName("com.mysql.jdbc.Driver") 这段代码不陌生，这是每个 Java 初学者必定遇到过，但如今的数据库驱动仍然是这样加载的吗？你还能找到这段代码吗？这一切的疑问，将在本篇文章结束后得到解答。</p>

```
<script async src="https://pagead2.googlesyndication.com/pagead/js/adsbygoogle.js"></script>
<!-- 黑客派PC帖子内嵌-展示 -->
<ins class="adsbygoogle" style="display:block" data-ad-client="ca-pub-5357405790190342" data-ad-slot="8316640078" data-ad-format="auto" data-full-width-responsive="true"></ins>
<script>
  (adsbygoogle = window.adsbygoogle || []).push({});
</script>
<p>首先介绍 SPI 机制是个什么东西</p>
<h2 id="实现一个自定义的SPI">实现一个自定义的 SPI</h2>
<h3 id="1-项目结构">1 项目结构</h3>
<p></p>
<ol>
  <li> <p>invoker 是我们的用来测试的主项目。</p> </li>
  <li> <p>interface 是针对厂商和插件商定义的接口项目，只提供接口，不提供实现。</p> </li>
  <li> <p>good-printer,bad-printer 分别是两个厂商对 interface 的不同实现，所以他们会依赖于 interface 项目。</p> </li>
</ol>
<p>这个简单的 demo 就是让大家体验，在不改变 invoker 代码，只更改依赖的前提下，切换 interface 的实现厂商。</p>
<h3 id="2-interface模块">2 interface 模块</h3>
<p><strong>2.1 moe.cnkirit0.spi.api.Printer</strong></p>
<ol>
  <li> <p><code>public interface Printer {</code></p> </li>
  <li> <p><code> void print();</code></p> </li>
  <li> <p><code>}</code></p> </li>
</ol>
<p>interface 只定义一个接口，不提供实现。规范的制定方一般都是比较牛叉的存在，这些接口通常位于 Java, javax 前缀的包中。这里的 Printer 就是模拟一个规范接口。</p>
<h3 id="3-good-printer模块">3 good-printer 模块</h3>
<p><strong>3.1 good-printer\pom.xml</strong></p>
<ol>
  <li> <p>`</p> </li>
  <li> <p><code> </code></p> </li>
  <li> <p><code> moe.cnkirit0</code></p> </li>
  <li> <p><code> interface</code></p> </li>
  <li> <p><code> 1.0-SNAPSHOT</code></p> </li>
  <li> <p><code> </code></p> </li>
  <li> <p>`</p> </li>
</ol>
<p>规范的具体实现类必然要依赖规范接口</p>
<p><strong>3.2 moe.cnkirit0.spi.api.GoodPrinter</strong></p>
<ol>
```

```
<li> <p><code>public class GoodPrinter implements Printer {</code></p> </li>
<li> <p><code> public void print() {</code></p> </li>
<li> <p><code> System.out.println("你是个好人~");</code></p> </li>
<li> <p><code> }</code></p> </li>
<li> <p><code>}</code></p> </li>
</ol>
<p>作为 Printer 规范接口的实现一</p>
<p><strong>3.3 resources\META-INF\services\moe.cnkiriti.spi.api.Printer</strong></p>
<ol>
<li><code>moe.cnkiriti.spi.api.GoodPrinter</code></li>
</ol>
<p>这里需要重点说明，每一个 SPI 接口都需要在自己项目的静态资源目录中声明一个 services 文
件，文件名为实现规范接口的类名全路径，在此例中便是 <code>moe.cnkiriti.spi.api.Printer</code>
，在文件中，则写上一行具体实现类的全路径，在此例中便是 <code>moe.cnkiriti.spi.api.GoodPri
ter</code>。</p>
<p>这样一个厂商的实现便完成了。</p>
<h3 id="4-bad-printer模块">4 bad-printer 模块</h3>
<p>我们在按照和 good-printer 模块中定义的一样的方式，完成另一个厂商对 Printer 规范的实现
</p>
<p><strong>4.1 bad-printer\pom.xml</strong></p>
<ol>
<li> <p>``</p> </li>
<li> <p><code> </code></p> </li>
<li> <p><code> moe.cnkiriti </code></p> </li>
<li> <p><code> interface </code></p> </li>
<li> <p><code> 1.0-SNAPSHOT </code></p> </li>
<li> <p><code> </code></p> </li>
<li> <p>``</p> </li>
</ol>
<p><strong>4.2 moe.cnkiriti.spi.api.BadPrinter</strong></p>
<ol>
<li> <p><code>public class BadPrinter implements Printer {</code></p> </li>
<li></li>
<li> <p><code> public void print() {</code></p> </li>
<li> <p><code> System.out.println("我抽烟，喝酒，蹦迪，但我知道我是好女孩~");</code></
> </li>
<li> <p><code> }</code></p> </li>
<li> <p><code>}</code></p> </li>
</ol>
<p><strong>4.3 resources\META-INF\services\moe.cnkiriti.spi.api.Printer</strong></p>
<script async src="https://pagead2.googlesyndication.com/pagead/js/adsbygoogle.js"></scr
pt>
<!-- 黑客派PC帖子内嵌-展示 -->
<ins class="adsbygoogle" style="display:block" data-ad-client="ca-pub-5357405790190342"
data-ad-slot="8316640078" data-ad-format="auto" data-full-width-responsive="true"></in
>
<script>
  (adsbygoogle = window.adsbygoogle || []).push({});
</script>
<ol>
<li><code>moe.cnkiriti.spi.api.BadPrinter</code></li>
</ol>
<p>这样，另一个厂商的实现便完成了。</p>
<h3 id="5-invoker模块">5 invoker 模块</h3>
```

<p>这里的 invoker 便是我们自己的项目了。如果一开始我们想使用厂商 good-printer 的 Printer 现，是需要将其的依赖引入。</p>

```
<ol>
<li> <p>``</p> </li>
<li> <p><code> </code></p> </li>
<li> <p><code> moe.cnkirit0</code></p> </li>
<li> <p><code> interface</code></p> </li>
<li> <p><code> 1.0-SNAPSHOT</code></p> </li>
<li> <p><code> </code></p> </li>
<li> <p><code> </code></p> </li>
<li> <p><code> moe.cnkirit0</code></p> </li>
<li> <p><code> good-printer</code></p> </li>
<li> <p><code> 1.0-SNAPSHOT</code></p> </li>
<li> <p><code> </code></p> </li>
<li> <p>``</p> </li>
</ol>
```

<p><strong>5.1 编写调用主类</strong></p>

```
<ol>
<li> <p><code>public class MainApp {</code></p> </li>
<li> </li>
<li> </li>
<li> <p><code> public static void main(String[] args) {</code></p> </li>
<li> <p><code> ServiceLoader<lt;Printer&gt; printerLoader = ServiceLoader.load(Printer.clas
s);</code></p> </li>
<li> <p><code> for (Printer printer : printerLoader) {</code></p> </li>
<li> <p><code> printer.print();</code></p> </li>
<li> <p><code> }</code></p> </li>
<li> <p><code> }</code></p> </li>
<li> <p><code>}</code></p> </li>
</ol>
```

<p>ServiceLoader 是 <code>java.util</code> 提供的用于加载固定类路径下文件的一个加载器，正是它加载了对应接口声明的实现类。</p>

<p><strong>5.2 打印结果 1</strong></p>

```
<ol>
<li><code>你是个好人~</code></li>
</ol>
```

<p>如果在后续的方案中，想替换厂商的 Printer 实现，只需要将依赖更换</p>

```
<ol>
<li> <p>``</p> </li>
<li> <p><code> </code></p> </li>
<li> <p><code> moe.cnkirit0</code></p> </li>
<li> <p><code> interface</code></p> </li>
<li> <p><code> 1.0-SNAPSHOT</code></p> </li>
<li> <p><code> </code></p> </li>
<li> <p><code> </code></p> </li>
<li> <p><code> moe.cnkirit0</code></p> </li>
<li> <p><code> bad-printer</code></p> </li>
<li> <p><code> 1.0-SNAPSHOT</code></p> </li>
<li> <p><code> </code></p> </li>
<li> <p>``</p> </li>
</ol>
```

<p>调用主类无需变更代码，这符合开闭原则</p>

<p><strong>5.3 打印结果 2</strong></p>

```
<ol>
```

```
<li><code>我抽烟，喝酒，蹦迪，但我知道我是好女孩~</code></li>
</ol>
<p>是不是很神奇呢？这一切对于调用者来说都是透明的，只需要切换依赖即可！</p>
<h2 id="SPI在实际项目中的应用">SPI 在实际项目中的应用</h2>
<p>先总结下有什么新知识，resources/META-INF/services 下的文件似乎我们之前没怎么接触过，ServiceLoader 也没怎么接触过。那么现在我们打开自己项目的依赖，看看有什么发现。</p>
<ol>
<li></li>
</ol>
<pre><code class="highlight-chroma">在mysql-connector-java-xxx.jar中发现了META-INF\services\java.sql.Driver文件，里面只有两行记录：</code></pre>
```

我们可以分析出，`java.sql.Driver`是一个规范接口，`com.mysql.jdbc.Driver` `com.mysql.fabric.jdbc.DriverMySQLDriver`则是mysql-connector-java-xxx.jar对这个规范的实现接口。

```
</code></pre>
```

```
<ol>
<li> <p><code>com.mysql.jdbc.Driver</code></p> </li>
<li> <p><code>com.mysql.fabric.jdbc.FabricMySQLDriver</code></p> </li>
<li></li>
</ol>
<pre><code class="highlight-chroma">在jcl-over-slf4j-xxxx.jar中发现了META-INF\services\org.apache.commons.logging.LogFactory文件，里面只有一行记录：</code></pre>
```

相信不用我赘述，大家都能理解这是什么含义了

```
</code></pre>
```

```
<ol>
<li> <p><code>org.apache.commons.logging.impl.SLF4JLogFactory</code></p> </li>
<li></li>
</ol>
<pre><code class="highlight-chroma">更多的还有很多，有兴趣可以自己翻一翻项目路径下的些jar包</code></pre>
<script async src="https://pagead2.googlesyndication.com/pagead/js/adsbygoogle.js"></script>
<!-- 黑客派PC帖子内嵌-展示 --&gt;
&lt;ins class="adsbygoogle" style="display:block" data-ad-client="ca-pub-5357405790190342" data-ad-slot="8316640078" data-ad-format="auto" data-full-width-responsive="true"&gt;&lt;/ins&gt;
&lt;script&gt;
  (adsbygoogle = window.adsbygoogle || []).push({});&lt;/script&gt;</pre>
```

<p>既然说到了数据库驱动，索性再多说一点，还记得一道经典的面试题：`Class.forName("com.mysql.jdbc.Driver")`到底做了什么事？</p>

<p>先思考下：自己会怎么回答？</p>

<p>都知道 `Class.forName` 与类加载机制有关，会触发执行 `com.mysql.jdbc.Driver` 类中的静态方法，从而使主类加载数据库驱动。如果再追问，为什么它的静态块没有自动触发？可答：因为数据库驱动的特殊性质，JDBC 规范中明确要求 Driver 类必须向 DriverManager 注册自己，导致其必须由 `Class.forName` 手动触发，这可以在 `java.sql.Driver` 中得到解释。完美了吗？还没，来到最新的 DriverManager 源码中，可以看到这样的注释，翻译如下：</p>

```
<blockquote>
```

<p><code>DriverManager</code> 类的方法 <code>getConnection</code> 和 <code>get

drivers</code> 已经得到提高以支持 Java Standard Edition Service Provider 机制。 JDBC 4.0 Drivers 必须包括 <code>META-INF/services/java.sql.Driver</code> 文件。此文件包含 <code>java.sql.Driver</code> 的 JDBC 驱动程序实现的名称。例如，要加载 <code>my.sql.Driver</code> 类，<code>META-INF/services/java.sql.Driver</code> 文件需要包含下面的条目： </p>

```
<blockquote>
</blockquote>
<ol>
<li><code> my.sql.Driver</code> </li>
</ol>
<p>应用程序不再需要使用 <code>Class.forName()</code> 显式地加载 JDBC 驱动程序。当前用 <code>Class.forName()</code> 加载 JDBC 驱动程序的现有程序将在不作修改的情况下继续工作。 </p>
```

```
</blockquote>
<p>可以发现，Class.forName 已经被弃用了，所以，这道题目的最佳回答，应当是和面试官牵扯到 ava 中的 SPI 机制，进而聊聊加载驱动的演变历史。 </p>
```

```
<p><strong>java.sql.DriverManager</strong></p>
<ol>
<li> <p><code>public Void run() {</code></p> </li>
<li></li>
<li> <p><code> ServiceLoader<Driver> loadedDrivers = ServiceLoader.load(Driver.class);</code></p> </li>
<li> <p><code> Iterator<Driver> driversIterator = loadedDrivers.iterator();</code></p> </li>
<li></li>
<li> <p><code> try{</code></p> </li>
<li> <p><code> while(driversIterator.hasNext()) {</code></p> </li>
<li> <p><code> driversIterator.next();</code></p> </li>
<li> <p><code> }</code></p> </li>
<li> <p><code> } catch(Throwable t) {</code></p> </li>
<li> <p><code> // Do nothing</code></p> </li>
<li> <p><code> }</code></p> </li>
<li> <p><code> return null;</code></p> </li>
<li> <p><code>}</code></p> </li>
</ol>
```

<p>当然那，本节的内容还是主要介绍 SPI，驱动这一块这是引申而出，如果不太理解，可以多去翻翻 jdk1.8 中 Driver 和 DriverManager 的源码，相信会有不小的收获。 </p>

## SPI 在扩展方面的应用

<p>SPI 不仅仅是为厂商指定的标准，同样也为框架扩展提供了一个思路。框架可以预留出 SPI 接口这样可以在不侵入代码的前提下，通过增删依赖来扩展框架。前提是，框架得预留出核心接口，也就本例中 interface 模块中类似的接口，剩下的适配工作便留给了开发者。 </p>

```
<script async src="https://pagead2.googlesyndication.com/pagead/js/adsbygoogle.js"></script>
```

```
<!-- 黑客派PC帖子内嵌-展示 -->
```

```
<ins class="adsbygoogle" style="display:block" data-ad-client="ca-pub-5357405790190342" data-ad-slot="8316640078" data-ad-format="auto" data-full-width-responsive="true"></ins>
```

```
<script>
```

```
  (adsbygoogle = window.adsbygoogle || []).push({});
```

```
</script>
```

<p>例如我的上一篇文章中介绍的 motan 中 Filter 的扩展，便是采用了 SPI 机制，熟悉这个设定之后再去回头去了解一些框架的 SPI 扩展就不会太陌生了。 </p>