Java SPI 机制详解

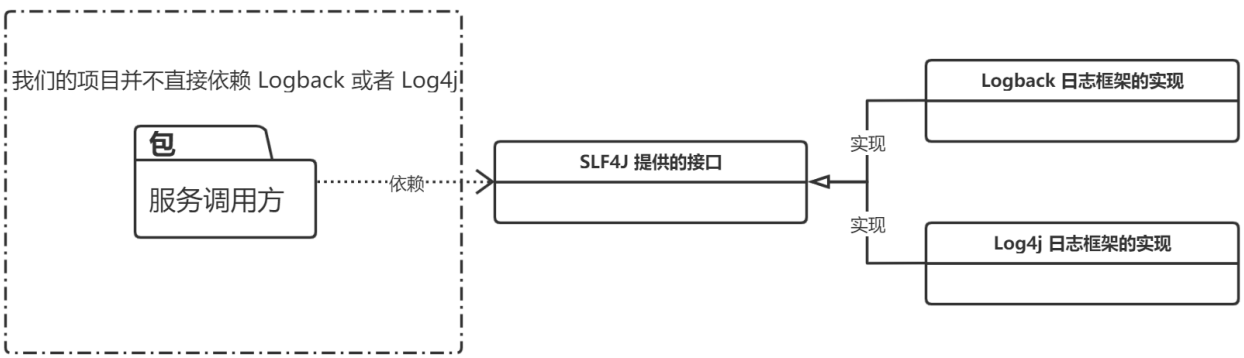
> 最近参与了一个新项目，里面使用到了 SPI，所以自己也重新学习了一下 SPI，在这里做一个小总结。
>
> 本文使用 IDEA 进行打包 jar，所需的工具：
>
> - IDEA 2021
> - JDK 11

1、什么是 SPI

SPI 全称为 Service Provider Interface，在很多框架都有使用，例如 JDBC、SLF4J、Dubbo 等，是一种基于接口编程的方式。

SPI 将服务接口和具体的服务实现分离开，将服务使用者和服务实现者解耦，提升程序的扩展性、可维护性。修改或者替换服务实现并不需要修改服务使用者。

拿最常用的日志框架 SLF4J 举例：SLF4J 的后端可以是 Logback、Log4j2 等，而且可以换来换去，连项目里面的代码都不要更改，也就是换个 Jar 包的事情（Maven、Gradle 改一下依赖而已），而这种就是靠 SPI 实现的。



SPI 的最重要的作用就是再无需修改原始代码库的情况下，通过使用“插件”的方式新增新的功能实现、修改或者移除旧的功能实现。

Service： 服务，也就是为上层应用程序提供特定的功能的访问。服务可以自定义功能接口以及使用接口的方式，定义的一组公关接口和抽象类。

Service Provider Interface： 服务提供的接口抽象类。

> SLF4J : 就是一个服务 Service，为应用程序提供了使用日志功能的访问
>
> ```
> Logger log = LoggerFactory.getLogger(XX.class);
> log.info("输出 info 的日志")
> ```
>
> 同样， SLF4J 也提供了一组接口类，这就是 Service Provider Interface。

Service Provider：实现 Service Provider Interface，也就是实现具体的功能。

> 例如 Logback，实现了 SLF4J 的接口，这样子就可以通过 SLF4J 来输出日志。达到了应用程序和具体的日志输出框架的解耦。

2、SPI 示例

这里通过一个简单的 日志输出 的示例，来详解 SPI 以及 SPI 的工作原理。

2.1、实现 Service Provider Interface

在 IDEA 新建一个普通的 Java 项目，service-provider-interface, 目录结构如下

```
└── src
    ├── META-INF
    │   └── MANIFEST.MF
    └── org
        └── spi
            └── service
                ├── Logger.java
                ├── LoggerService.java
                ├── Main.java
                └── MyServicesLoader.java
```

新建 Logger.java，这个就是 SPI，Service provider interface 就是服务提供的接口，Service Provider 服务提供实现这个接口。

```java
package org.spi.service;
/**
 * 这个是 Service provider interface (SPI): 服务提供商接口
 */
public interface Logger {
    void info(String msg);
    void debug(String msg);
}
```

新建 LoggerService.java , 这个就是服务 Service，为使用者提供特定功能的访问。

> ServiceLoader 这个类后面会介绍

```java
package org.spi.service;
public class LoggerService {
    private static final LoggerService SERVICE = new LoggerService();

    private final Logger logger;

    private final List<Logger> loggerList;


    private LoggerService(){
        ServiceLoader<Logger> loader = ServiceLoader.load(Logger.class);
        List<Logger> list = new ArrayList<>();
        for(Logger log: loader){
            list.add(log);
        }
        // LoggerList 是所有 ServiceProvider
        loggerList = list;
        if(!list.isEmpty()){
            // Logger 只取一个
            logger = list.get(0);
        }else {
            logger = null;
        }
    }

    public static LoggerService getService(){
        return SERVICE;
    }

    public void info(String msg){
        if(logger == null) {
            System.out.println("info 中没有发现 Logger 服务提供者");
        }else{
            logger.info(msg);
        }
    }

    public void debug(String msg){
        if(loggerList.isEmpty()){
            System.out.println("debug 中没有发现 Logger 服务提供者");
        }
        loggerList.forEach(log -> log.debug(msg));
    }
}
```

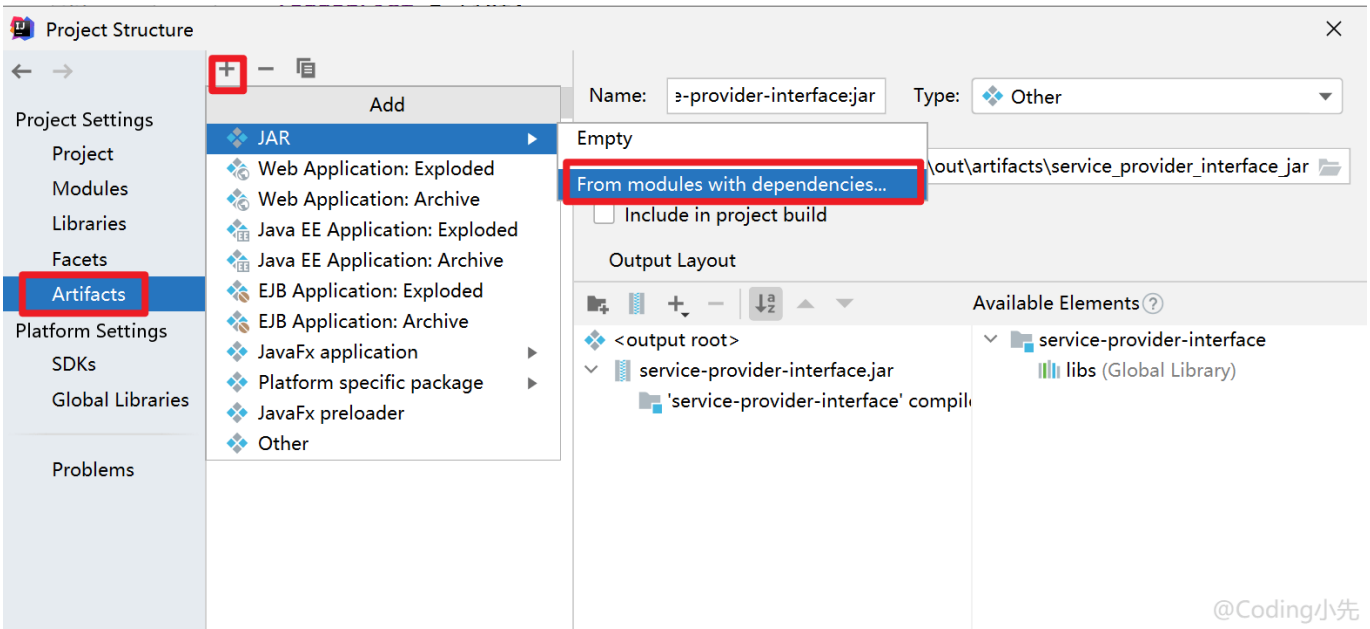
新建 Main.java 类，也就是服务使用者，这里为了方便展示就没有新建一个新项目了

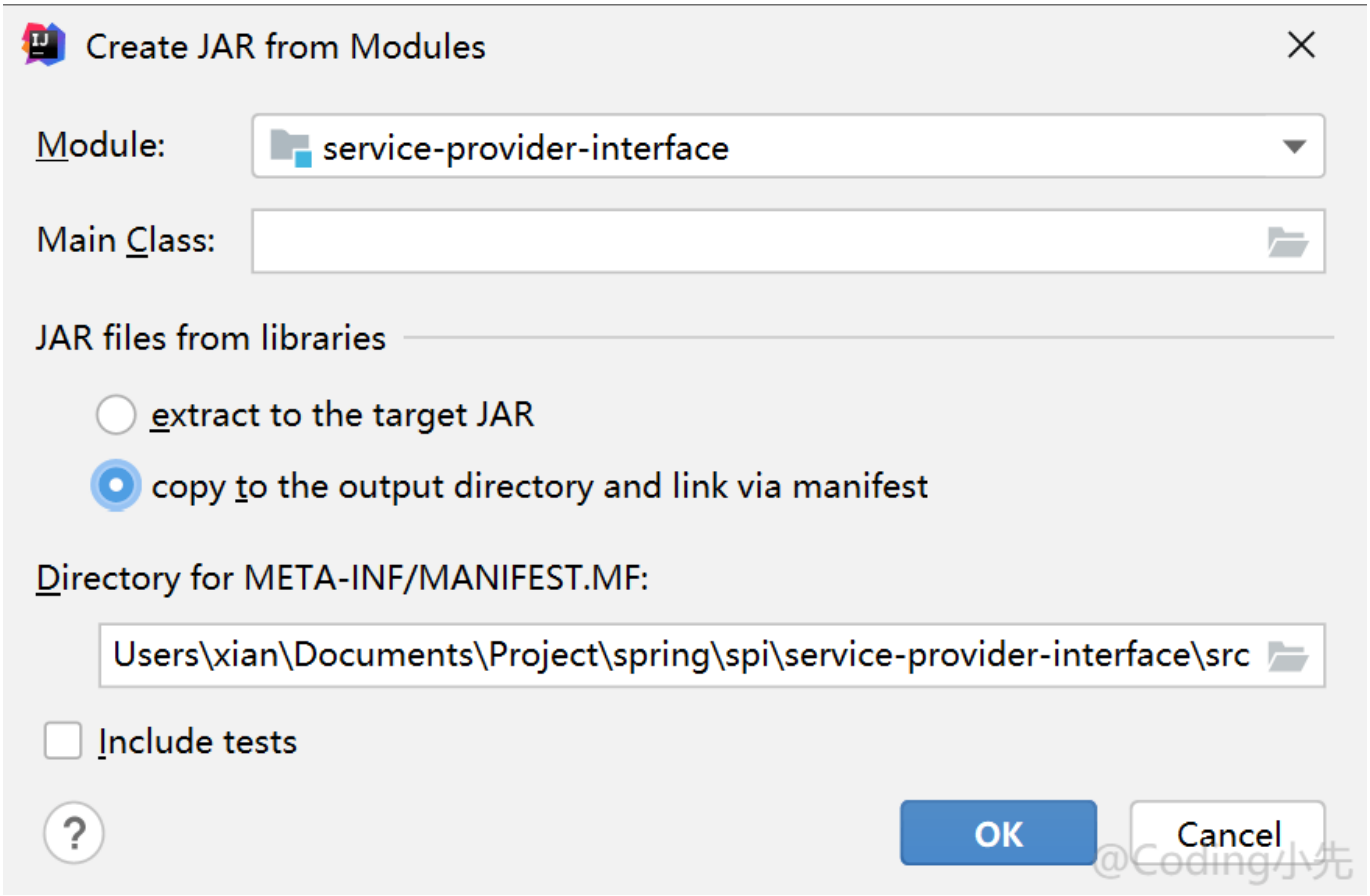
```java
package org.spi.service;

public class Main {
    public static void main(String[] args) {
        LoggerService service = LoggerService.getService();

        service.info("哈哈哈");
        service.debug("嘻嘻嘻");
    }
}
```

直接运行，会看到输出：

```
info 中没有发现 Logger 服务提供者
debug 中没有发现 Logger 服务提供者
```

将整个项目打包成 jar，这里通过 IDEA 的方式打包，打开 Project Structure 的设置界面，设置完成后运行 "Build" -> "Build Artifacts"。





2.2、实现 Service Provider

> 这里将实现上文的 Logger 接口

新建项目 service-provider，目录结构如下，其中 service-provider-interface.jar 是上文项目打包成的 jar 包。需要在 “Project Structure” 设置中将这个 jar 包添加到项目。

```
.
├── libs
│   └── service-provider-interface.jar
└── src
    ├── META-INF
    │   ├── MANIFEST.MF
    │   └── services
    │       └── org.spi.service.Logger
    └── org
        └── spi
            └── provider
                └── Logback.java
```

新建 Logback.java： 实现 Logger.java

```java
package org.spi.provider;
public class Logback implements Logger {
    @Override
    public void info(String msg) {
        System.out.println("Logback info 的输出" + msg);
    }

    @Override
    public void debug(String msg) {
        System.out.println("Logback debug 的输出" + msg);
    }
}
```

在 src 目录下新建 META-INF/services 的文件夹，新建 org.spi.service.Logger 的文件。

> 这个文件名就是上文 Logger 接口类的 包名+类名。
>
> 内容就是实现 Logger 接口的类的 包名+类名。
>
> 这个是 JDK 的 ServiceLoader 规定的

```
org.spi.provider.Logback
```

参考上文，打包成 jar 包。

2.3、使用 Service Provider

回到 service-provider-interface 的项目，将 service-provider 的 jar 包通过 “Project Structure”引入，重新运行 Mian.java，此时输出变成了：

```
Logback info 的输出哈哈哈
Logback debug 的输出嘻嘻嘻
```

通过使用 SPI 的方式，可以发现服务(LoggerService)、服务提供者（Logback）两者之间的耦合度非常低：

例如需要替换 Logback，换一个新的实现 LogbackPlus，只需替换 jar 即可，就如同 SLF4J 一样。

> 也就是 LoggerService.info() 的方式，只使用一个 ServiceProvider。

例如某天新的需求，需要（Logback）日志输出到 控制台 的同时，还需要发送到 Kafka，那么只需要新加一个发送到 Kafka 功能的 jar 即可，而无需修改原有的 Logback

> 也就是 LoggerService.debug() 的使用方式，同时调用所有 Service Provider 的实现。

3、ServiceLoader 的作用

前面提到过 ServiceLoader 这个类的作用会在后面解析，这是 JDK 提供的一个类，其功能就是从读取 所有 jar 包下的 META-INF/services/ 的文件，然后通过反射的方式实例化 ServiceProvider 。

> 这也是为什么上文的 service-provider 需要 org.spi.service.Logger 文件的原因。

这里实现一个简易版的 ServiceLoader 。

```
public class MyServicesLoader<S> {
    // 对应服务的接口
    private final Class<S> service;
    // 暂存所有的 Provider 的实例
    private final List<S> providers = new ArrayList<>();
    private final ClassLoader classLoader;

    public static <S> MyServicesLoader<S> load(Class<S> service){
        return new MyServicesLoader<>(service);
    }

    private MyServicesLoader(Class<S> service){
        this.service = service;
        this.classLoader = Thread.currentThread().getContextClassLoader();
        doLoad();
    }

    private void doLoad(){
        try {
            // 读取所有 jar 包的 META-INF/services/" + service.getName() 文件
            Enumeration<URL> urls = classLoader.getResources("META-
INF/services/" + service.getName());
            // 遍历
            while(urls.hasMoreElements()) {
                URL u = urls.nextElement();
                // 打印路径 path
                System.out.println("File =" + u.getPath());
                // 读取 path 路径下的内容
                URLConnection uc = u.openConnection();
                uc.setUseCaches(false);
                InputStream inputStream = uc.getInputStream();

                BufferedReader reader = new BufferedReader(new
InputStreamReader(inputStream));
                String className = reader.readLine();
                while (className != null){
                    Class<?> clazz = Class.forName(className, false,
classLoader);
                    if(service.isAssignableFrom(clazz)){
                        // 反射出实例
                        Constructor<? extends S> ctor = (Constructor<? extends
S>) clazz.getConstructor();
                        S instance = ctor.newInstance();
                        providers.add(instance);
                    }
                    className = reader.readLine();
                }
```

```
            }
        } catch (Exception ignored){ }
    }

    public List<S> getProviders() {
        return providers;
    }
}
```

如何使用

```
MyServicesLoader<Logger> loggers = MyServicesLoader.load(Logger.class);

loggers.getProviders().forEach(provider -> provider.info("这个是
MyServicesLoader "));
```

总结

这里主要详解了 Java SPI 机制，结合代码食用效果更佳呢！