链滴

Spring Boot 启动原理解析

作者：xjlnjut730

原文链接：https://ld246.com/article/1576020530961

来源网站：链滴

与传统的web应用服务部署不同，Spring Boot提供了java -jar 这种方式的一键部署，不再需要单独署tomcat实例，使得部署变得相当简单。这背后究竟是如何实现的呢？要想分析这个问题，我们需先了解Spring Boot打包后的jar文件，究竟是什么样子的。

1. 目录结构分析

我们在IDEA中新建一个Spring Boot工程，叫spring-boot-demo，打包之后，得到一个jar文件：spring-boot-demo-0.0.1-SNAPSHOT.jar，我们用unzip命令解压该jar包后，能够看到对应的目录结构如下图所示：

```
$ tree -L 3
.
├── BOOT-INF
│   ├── classes
│   │   ├── application.properties
│   │   ├── cn
│   │   ├── static
│   │   └── templates
│   └── lib
│       ├── spring-core-5.2.2.RELEASE.jar
│       ├── spring-webmvc-5.2.2.RELEASE.jar
│       ├── ...// 这里略了大量jar包
├── META-INF
│   └── MANIFEST.MF
└── org
    └── springframework
        └── boot
```

我们注意到，工程中的源代码部分编译完成后会进入BOOT-INF/classes文件夹下，工程的依赖会进入OOT-INF/lib目录下。除此之外还有一个META-INF与org/springframework/boot/..的文件夹。看这个，不禁就有疑问，这两个文件夹是干嘛用的?

我们先来看META-INF目录，这个目录下就只有一个MANIFEST.MF文件，我们看下里面的内容：

```
$ cat META-INF/MANIFEST.MF
Manifest-Version: 1.0
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: cn.xiajl.springbootdemo.SpringBootDemoApplication
Spring-Boot-Version: 2.2.2.RELEASE
Spring-Boot-Classes: BOOT-INF/classes/
Spring-Boot-Lib: BOOT-INF/lib/
```

我们可以查询下中oracle对该文件的定义，见参考资料1。我们会发现，只有Manifest-Version、Mai-Class是oracle定义的，Manifest-Version表示jar包的版本号，Main-Class表示jar启动时的启动类其它的Start-Class、Spring-Boot-Version、Spring-Boot-Classes、Spring-Boot-Lib都不在MANIFST.MF的规范里，换句话说，这是SpringBoot自己定义的。

这里有个问题，既然Main-Class是启动类，那么Main-Class为什么是org.springframework.boot.loder.JarLauncher，而不是cn.xiajl.springbootdemo.SpringBootDemoApplication?

我们可以尝试把Main-Class改成工程中的SpringBootDemoApplication类试试，打包回去，尝试看能不能启动：

```
$ vi META-INF/MANIFEST.MF
```

```
$ cat META-INF/MANIFEST.MF
Manifest-Version: 1.0
Start-Class: org.springframework.boot.loader.JarLauncher
Main-Class: cn.xiajl.springbootdemo.SpringBootDemoApplication
Spring-Boot-Version: 2.2.2.RELEASE
Spring-Boot-Classes: BOOT-INF/classes/
Spring-Boot-Lib: BOOT-INF/lib/
$ jar -cvf0M rarpack.jar *
...//略去输出
$ java -jar rarpack.jar
错误: 找不到或无法加载主类 cn.xiajl.springbootdemo.SpringBootDemoApplication
原因: java.lang.ClassNotFoundException: cn.xiajl.springbootdemo.SpringBootDemoApplication
```

很明显启动不了。那看到这里，很自然的就能想到，这个JarLauncher是不是要帮我们解决类路径的题？jar启动的时候，按照oracle的规范似乎要手工指定classpath，并且也没有办法定位到jar包中的BOT-INF/BOOT-Classes路径，只会加载jar包下直接解决的class文件。

2. JarLauncher分析

当思考到这里的时候，我们像往常一样，在Idea里面搜JarLauncher这个类，试图进去看看这个类的码，你会发现找不到。为什么呢？因为这个是插件导进去的，对于gradle编译的工程，是在bootJar段，将spring-boot-loader中类直接解压拷贝到jar包中的。那我们怎么才能看到源码呢？直接在依里加上spring-boot-loader即可：

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    // 仅用于研究spring-boot-loader，实际工程中不需要该依赖
    implementation group: 'org.springframework.boot', name: 'spring-boot-loader', version: '2.
.2.RELEASE'
}
```

打开JarLancher的源码，我们会看一下类的说明：

```
Launcher for JAR based archives. This launcher assumes that dependency jars are included ins
de a /BOOT-INF/lib directory and that application classes are included inside a /BOOT-INF/cla
ses directory.
```

文档很明确地说明了，它是来帮我们加载/BOOT-INF/lib下的jar包与/BOOT-INF/classes下的classe文件。

那它到底是怎么帮我们加载的呢？

我们跟进源码，JarLancher中存在如下main方法：

```
public static void main(String[] args) throws Exception {
    new JarLauncher().launch(args);
}
```

我们来看看launch是怎么实现的：

```
/**
 * Launch the application. This method is the initial entry point that should be
 * called by a subclass {@code public static void main(String[] args)} method.
 * @param args the incoming arguments
```

```
     * @throws Exception if the application fails to launch
     */
    protected void launch(String[] args) throws Exception {
        JarFile.registerUrlProtocolHandler();
        ClassLoader classLoader = createClassLoader(getClassPathArchives());
        launch(args, getMainClass(), classLoader);
    }

    /**
     * Create a classloader for the specified URLs.
     * @param urls the URLs
     * @return the classloader
     * @throws Exception if the classloader cannot be created
     */
    protected ClassLoader createClassLoader(URL[] urls) throws Exception {
        return new LaunchedURLClassLoader(urls, getClass().getClassLoader());
    }

    /**
     * Launch the application given the archive file and a fully configured classloader.
     * @param args the incoming arguments
     * @param mainClass the main class to run
     * @param classLoader the classloader
     * @throws Exception if the launch fails
     */
    protected void launch(String[] args, String mainClass, ClassLoader classLoader) throws Except
    on {
        Thread.currentThread().setContextClassLoader(classLoader);
        createMainMethodRunner(mainClass, args, classLoader).run();
    }
```

代码一目了然，Spring Boot会为我们创建一个自定义类加载器LaunchedURLClassLoader，并将其置为线程上下文类加载器。应用执行过程

中，会从LaunchedURLClassLoader进行类加载，这个类加载过程会遵循双亲委派机制，对于父类无加载的类，则由LaunchedURLClassLoader进行加载，LaunchedURLClassLoader加载的路径就是BOOT-INF/lib和BOOT-INF/classes。

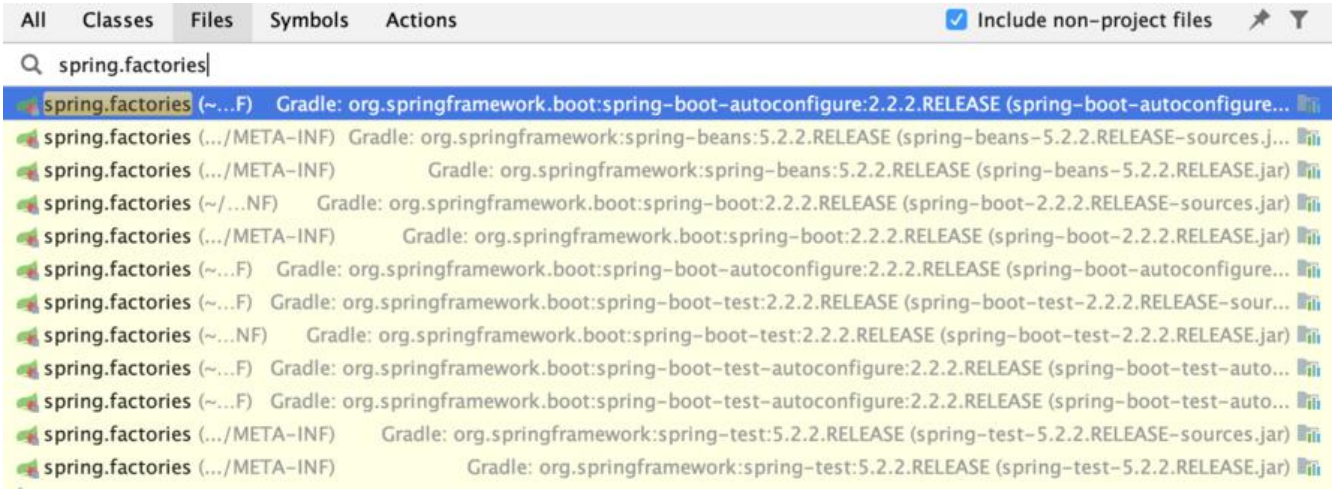
3. 为什么需要BOOT-INF/lib与BOOT-INF/classes?

执行逻辑我们搞清楚了，我们还需要了解一下为什么Spring Boot不按照传统的方式，即像Spring Boo Loader中的类一样，将所有的依赖解压放到jar中呢?是出于什么考虑?

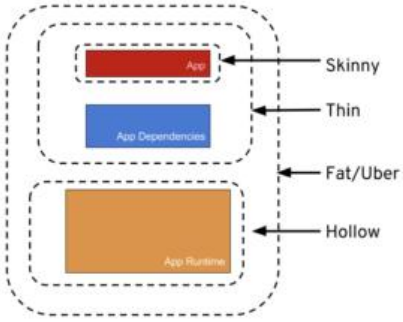
这里其实是有很多工程上的考量，我这边里总结一下：

1. 不同组件之前也有可能存在重名的情况，同样的组件不同版本之间也是不能兼容的。在解压时，无预测知道重名的那些文件，哪个组件的哪个版本会被保留下来。

2. SPI机制规范产生的配置文件很多时候是同名的，比如spring.factories(如下图所示)，这种配置文是不能覆盖的，必须保留多份。



因此，Spring Boot创新性的采用自定义类加载路径的方式来进行。生成的这种带依赖的jar也有一个有的名词，叫FatJar/UberJar，如下图所示(来源：参考资料2)：



参考资料

1. https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html
2. https://dzone.com/articles/the-skinny-on-fat-thin-hollow-and-uber