Java 打包 FatJar 方法小结

阿里云云栖社区 关注
0.613 2018.08.29 12:47:42 字数 1,698 阅读 6,170

在函数计算(Aliyun FC)中发布一个 Java 函数，往往需要将函数打包成一个 all-in-one 的 zip 包或者 jar 包。Java 中这种打包 all-in-one 的技术常称之为 Fatjar 技术。本文小结一下 Java 里打包 FatJar 的若干种方法。

什么是 FatJar

FatJar 又称作 uber-Jar，是包含所有依赖的 Jar 包。Jar 包中嵌入了除 java 虚拟机以外的所有依赖。我们知道 Java 的依赖分为两种， 零散的 .class 文件和把多个 .class 文件以 zip 格式打包而成 jar 文件。FatJar 是一个 all-in-one Jar 包。FatJar 技术可以让那些用于最终发布的 Jar 便于部署和运行。

三种打包方法

我们知道 .java 源码文件会被编译器编译成字节码.class 文件。Java 虚拟机执行的是 .class 文件。一个 java 程序可以有很多个 .class文件。这些 .class 文件可以由 java 虚拟机的类装载器运行期装载到内存里。java 虚拟机可以从某个目录装载所有的 .class 文件，但是这些零散的.class 文件并不便于分发。所有 java 支持把零散的.class 文件打包成 zip 格式的 .jar 文件，并且虚拟机的类装载器支持直接装载 .jar 文件。

一个正常的 java 程序会有若干个.class 文件和所依赖的第三方库的 jar 文件组成。

1. 非遮蔽方法（Unshaded）

非遮蔽是相对于遮蔽而说的，可以理解为一种朴素的办法。解压所有 jar 文件，再重新打包成一个新的单独的 jar 文件。

借助 Maven Assembly Plugin 都可以轻松实现非遮蔽方法的打包。

Maven Assembly Plugin

Maven Assembly Plugin 是一个打包聚合插件，其主要功能是把项目的编译输出协同依赖，模块，文档和其他文件打包成一个独立的发布包。使用描述符（descriptor）来配置需要打包的物料组合。并预定义了常用的描述符，可供直接使用。

预定义描述符如下

• **bin** 只打包编译结果，并包含 README, LICENSE 和 NOTICE 文件，输出文件格式为 tar.gz, tar.bz2 和 zip。

• **jar-with-dependencies** 打包编译结果，并带上所有的依赖，如果依赖的是 jar 包，jar 包会被解压开，平铺到最终的 uber-jar 里去。输出格式为 jar。

• **src** 打包源码文件。输出格式为 tar.gz, tar.bz2 和 zip。

• **project** 打包整个项目，除了部署输出目录 target 以外的所有文件和目录都会被打包。输出格式为 tar.gz, tar.bz2 和 zip。

除了预定义的描述符，用户也可以指定描述符，以满足不同的打包需求。

打包成 uber-jar，需要使用预定义的 jar-with-dependencies 描述符：

在 pom.xml 中加入如下配置

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>CHOOSE LATEST VERSION HERE</version>
    <configuration>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
    </configuration>
    <executions>
        <execution>
            <id>assemble-all</id>
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Gradle Java plugin

gradle 下打包一个非遮蔽的 jar 包，有不少插件可以用，但是由于 gradle 自身的灵活性，可以直接用 groove 的 dsl 实现。

```
apply plugin: 'java'

jar {
    from {
        (configurations.runtime).collect {
            it.isDirectory() ? it : zipTree(it)
        }
    }
}
```

> 非遮蔽方法会把所有的 jar 包里的文件都解压到一个目录里，然后在打包同一个 fatjar 中。对于复杂应用很可能会碰到同名类相互覆盖问题。

2. 遮蔽方法（Shaded）

遮蔽方法会把依赖包里的类路径进行修改到某个子路径下，这样可以一定程度上避免同名类相互覆盖的问题。最终发布的 jar 也不会带入传递依赖冲突问题给下游。

Maven Shade Plugin

在 pom.xml 中加入如下配置

```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>3.1.1</version>
    <configuration>
        <!-- put your configurations here -->
    </configuration>
    <executions>
        <execution>
        <phase>package</phase>
        <goals>
            <goal>shade</goal>
        </goals>
        </execution>
    </executions>
 </plugin>
```

Gradle Shadow plugin

Gradle shadow plugin 使用非常简单，简单声明插件后就可以生效。

```
plugins {
  id 'com.github.johnrengelman.shadow' version '2.0.4'
  id 'java'
}

shadowJar {
   include '*.jar'
   include '*.properties'
```

> 遮蔽方法依赖修改 class 的字节码，更新依赖文件的包路径达到规避同名同包类冲突的问题，但是改名也会带来其他问题，比如代码中使用 Class.forName 或 ClassLoader.loadClass 装载的类，Shade Plugin 是感知不到的。同名文件覆盖问题也没法杜绝，比如META-INF/services/javax.script.ScriptEngineFactory不属于类文件，但是被覆盖后会出现问题。

3. 嵌套方法（Jar of Jars）

还是一种办法就是在 jar 包里嵌套其他 jar，这个方法可以彻底避免解压同名覆盖的问题，但是这个方法不被 JVM 原生支持，因为 JDK 提供的 ClassLoader 仅支持装载嵌套 jar 包的 class 文件。所以这种方法需要自定义 ClassLoader 以支持嵌套 jar。

Onejar Maven Plugin

One-JAR 就是一个基于上面嵌套 jar 实现的工具。onejar-maven-plugin 是社区基于 onejar 实现的 maven 插件。

```
<plugin>
    <groupId>com.jolira</groupId>
    <artifactId>onejar-maven-plugin</artifactId>
    <version>1.4.4</version>
    <executions>
        <execution>
            <goals>
                <goal>one-jar</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Spring boot plugin

One-JAR 有点年久失修，好久没有维护了，Spring Boot 提供的 Maven Plugin 也可以打包 Fatjar，支持非遮蔽和嵌套的混合模式，并且支持 maven 和 gradle 。

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <layout>ZIP</layout>
        <requiresUnpack>
            <dependency>
                <groupId>org.jruby</groupId>
                <artifactId>jruby-complete</artifactId>
            </dependency>
        </requiresUnpack>
    </configuration>
</plugin>
```

```
plugins {
    id 'org.springframework.boot' version '2.0.4.RELEASE'
}

bootJar {
    requiresUnpack '**/jruby-complete-*.jar'
}
```

requiresUnpack 参数可以定制那些 jar 不希望被解压，采用嵌套的方式打包到 Fatjar 内部。

其打包后的内部结构为

```
example.jar
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-BOOT-INF
    +-classes
    |  +-mycompany
    |     +-project
    |        +-YourClasses.class
    +-lib
       +-dependency1.jar
       +-dependency2.jar
```

应用的类文件被放置到 BOOT-INF/classes 目录，依赖包被放置到 BOOT-INF/lib 目录。

查看 META-INF/MANIFEST.MF 文件，其内容为

```
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: com.mycompany.project.MyApplication
```

启动类是固定的 org.springframework.boot.loader.JarLauncher，应用程序的入口类需要配置成 Start-Class。这样做的目的主要是为了支持嵌套 jar 包的类装载，替换掉默认的 ClassLoader。

> 但是函数计算 Java Runtime 需要的 jar 包是一种打包结构，在服务端运行时会解压开，./lib 目录加到 classpath 中，单不会调用 Main-Class。所以自定义 ClassLoader 是不生效的，所以不要使用嵌套 jar 结构，除非在入口函数指定重新定义 ClassLoader 或者 classpath 以支持 BOOT-INF/classes 和 BOOT-INF/lib 这样的定制化的类路径。

小结

插件	构建平台	工作机制
maven-assembly-plugin	maven	Unshaded
Gradle Java plugin	gradle	Unshaded
maven-shade-plugin	maven	Shaded
com.github.johnrengelman.shadow	gradle	Shaded
Onejar	ant, maven	Jar of Jars
Spring boot plugin	maven, gradle	Unshaded, Jar of Jars

单从 Fatjar 的角度看， Spring boot maven/gradle 做得最精致。但是 jar 包内部的自定义路径解压开以后和函数计算是不兼容的。所以如果用于函数计算打包，建议使用 Unshaded 或者 Shared 的打包方式，但是需要自己注意文件覆盖问题。

本文作者：倚贤

阅读原文