InfoQ Maven专栏（九）——打包的技巧

August 5th, 2011

“打包”这个词听起来比较土，比较正式的说法应该是”构建项目软件包“，具体说就是将项目中的各种文件，比如源代码、编译生成的字节码、配置文件、文档，按照规范的格式生成归档。最常见的当然就是JAR包和WAR包了，复杂点的例子是Maven官方下载页面的分发包， 它有自定义的格式，方便用户直接解压后就在命令行使用。作为一款”打包工具“，Maven自然有义务帮助用户创建各种各样的包，规范的JAR包和WAR包 自然不再话下，略微复杂的自定义打包格式也必须支持，本文就介绍一些常用的打包案例以及相关的实现方式，除了前面提到的一些包以外，你还能看到如何生成源码包、Javadoc包、以及从命令行可直接运行的CLI包。

Packaging的含义

任何一个Maven项目都需要定义POM元素packaging（如果不写则默认值为jar）。顾名思义，该元素决定了项目的打包方式。实际的情形 中，如果你不声明该元素，Maven会帮你生成一个JAR包；如果你定义该元素的值为war，那你会得到一个WAR包；如果定义其值为POM（比如是一个 父模块），那什么包都不会生成。除此之外，Maven默认还支持一些其他的流行打包格式，例如ejb3和ear。你不需要了解具体的打包细节，你所需要做的就是告诉Maven，”我是个什么类型的项目“，这就是**约定优于配置**的力量。

为了更好的理解Maven的默认打包方式，我们不妨来看看简单的声明背后发生了什么，对一个jar项目执行mvn package操作，会看到如下的输出：

```
[INFO] --- maven-jar-plugin:2.3.1:jar (default-jar) @ git-demo ---
[INFO] Building jar: /home/juven/git_juven/git-demo/target/git-demo-1.2-SNAPSHOT.jar
```

相比之下，对一个war项目执行mvn package操作，输出是这样的：

```
[INFO] --- maven-war-plugin:2.1:war (default-war) @ webapp-demo ---
[INFO] Packaging webapp
[INFO] Assembling webapp [webapp-demo] in [/home/juven/git_juven/webapp-demo/target/webapp-demo
[INFO] Processing war project
[INFO] Copying webapp resources [/home/juven/git_juven/webapp-demo/src/main/webapp]
[INFO] Webapp assembled in [90 msecs]
[INFO] Building war: /home/juven/git_juven/webapp-demo/target/webapp-demo-1.0-SNAPSHOT.war
```

对应于同样的package生命周期阶段，Maven为jar项目调用了maven-jar-plugin，为war项目调用了maven-war-plugin，换言之，**packaging直接影响Maven的构建生命周期**。了解这一点非常重要，特别是当你需要自定义打包行为的时候，你就必须知道去配置哪个插件。一个常见的例子就是在打包war项目的时候排除某些web资源文件，这时就应该配置maven-war-plugin如下：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>2.1.1</version>
    <configuration>
      <webResources>
        <resource>
          <directory>src/main/webapp</directory>
          <excludes>
            <exclude>**/*.jpg</exclude>
          </excludes>
        </resource>
      </webResources>
    </configuration>
  </plugin>
```

源码包和Javadoc包

本专栏的《坐标规划》一文中曾解释过，一个Maven项目只生成一个主构件，当需要生成其他附属构件的时候，就需要用上classifier。源码包和Javadoc包就是附属构件的极佳例子。它们有着广泛的用途，尤其是源码包，当你使用一个第三方依赖的时候，有时候会希望在IDE中直接进入该依赖的源码查看其实现的细节，如果该依赖将源码包发布到了Maven仓库，那么像Eclipse就能通过m2eclipse插件解析下载源码包并关联到你的项目中，十分方便。由于生成源码包是极其常见的需求，因此Maven官方提供了一个插件来帮助用户完成这个任务：

```xml
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-source-plugin</artifactId>
  <version>2.1.2</version>
  <executions>
    <execution>
      <id>attach-sources</id>
      <phase>verify</phase>
      <goals>
        <goal>jar-no-fork</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

类似的，生成Javadoc包只需要配置插件如下：

```xml
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-javadoc-plugin</artifactId>
  <version>2.7</version>
  <executions>
    <execution>
      <id>attach-javadocs</id>
      <goals>
        <goal>jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

为了帮助所有Maven用户更方便的使用Maven中央库中海量的资源，中央仓库的维护者强制要求开源项目提交构件的时候同时提供源码包和Javadoc包。这是个很好的实践，读者也可以尝试在自己所处的公司内部实行，以促进不同项目之间的交流。

可执行CLI包

除了前面提到了常规JAR包、WAR包，源码包和Javadoc包，另一种常被用到的包是在命令行可直接运行的CLI（Command Line）包。默认Maven生成的JAR包只包含了编译生成的.class文件和项目资源文件，而要得到一个可以直接在命令行通过java命令运行的 JAR文件，还要满足两个条件：

- JAR包中的/META-INF/MANIFEST.MF元数据文件必须包含Main-Class信息。
- 项目所有的依赖都必须在Classpath中。

Maven有好几个插件能帮助用户完成上述任务，不过用起来最方便的还是maven-shade-plugin， 它可以让用户配置Main-Class的值，然后在打包的时候将值填入/META-INF/MANIFEST.MF文件。关于项目的依赖，它很聪明地将依 赖JAR文件全部解压后，再将得到的.class文件连同当前项目的.class文件一起合并到最终的CLI包中。这样，在执行CLI JAR文件的时候，所有需要的类就都在Classpath中了。下面是一个配置样例：

```xml
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>1.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
        <transformers>
          <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResou
            <mainClass>com.juvenxu.mavenbook.HelloWorldCli</mainClass>
          </transformer>
        </transformers>
      </configuration>
```

```
      </execution>
    </executions>
  </plugin>
```

上述例子中的，我的Main-Class是com.juvenxu.mavenbook.HelloWorldCli，构建完成后，对应于一个常规的hello-world-1.0.jar文件，我还得到了一个hello-world-1.0-cli.jar文件。细心的读者可能已经注意到了，这里 用的是cli这个classifier。最后，我可以通过**java -jar hello-world-1.0-cli.jar**命令运行程序。

自定义格式包

实际的软件项目常常会有更复杂的打包需求，例如我们可能需要为客户提供一份产品的分发包，这个包不仅仅包含项目的字节码文件，还得包含依赖以及相关脚本文件以方便客户解压后就能运行，此外分发包还得包含一些必要的文档。这时项目的源码目录结构大致是这样的：

```
pom.xml
src/main/java/
src/main/resources/
src/test/java/
src/test/resources/
src/main/scripts/
src/main/assembly/
README.txt
```

除了基本的pom.xml和一般Maven目录之外，这里还有一个src/main/scripts/目录，该目录会包含一些脚本文件如 run.sh和run.bat，src/main/assembly/会包含一个assembly.xml，这是打包的描述文件，稍后介绍，最后的 README.txt是份简单的文档。

我们希望最终生成一个zip格式的分发包，它包含如下的一个结构：

```
bin/
lib/
README.txt
```

其中bin/目录包含了可执行脚本run.sh和run.bat，lib/目录包含了项目JAR包和所有依赖JAR，README.txt就是前面提到的文档。

描述清楚需求后，我们就要搬出Maven最强大的打包插件：maven-assembly-plugin。 它支持各种打包文件格式，包括zip、tar.gz、tar.bz2等等，通过一个打包描述文件（该例中是src/main /assembly.xml），它能够帮助用户选择具体打包哪些文件集合、依赖、模块、和甚至本地仓库文件，每个项的具体打包路径用户也能自由控制。如下 就是对应上述需求的打包描述文件src/main/assembly.xml：

```
<assembly>
  <id>bin</id>
  <formats>
    <format>zip</format>
  </formats>
  <dependencySets>
    <dependencySet>
      <useProjectArtifact>true</useProjectArtifact>
      <outputDirectory>lib</outputDirectory>
    </dependencySet>
  </dependencySets>
  <fileSets>
    <fileSet>
      <outputDirectory>/</outputDirectory>
      <includes>
        <include>README.txt</include>
      </includes>
    </fileSet>
    <fileSet>
      <directory>src/main/scripts</directory>
      <outputDirectory>/bin</outputDirectory>
      <includes>
        <include>run.sh</include>
        <include>run.bat</include>
```

```
      </includes>
    </fileSet>
  </fileSets>
</assembly>
```

- 首先这个assembly.xml文件的id对应了其最终生成文件的classifier。
- 其次formats定义打包生成的文件格式，这里是zip。因此结合id我们会得到一个名为hello-world-1.0-bin.zip的文件。（假设artifactId为hello-world，version为1.0）
- dependencySets用来定义选择依赖并定义最终打包到什么目录，这里我们声明的一个depenencySet默认包含所有所有 依赖，而useProjectArtifact表示将项目本身生成的构件也包含在内，最终打包至输出包内的lib路径下（由 outputDirectory指定）。
- fileSets允许用户通过文件或目录的粒度来控制打包。这里的第一个fileSet打包README.txt文件至包的根目录下，第二个fileSet则将src/main/scripts下的run.sh和run.bat文件打包至输出包的bin目录下。

打包描述文件所支持的配置远超出本文所能覆盖的范围，为了避免读者被过多细节扰乱思维，这里不再展开，读者若有需要可以去参考这份文档。

最后，我们需要配置maven-assembly-plugin使用打包描述文件，并绑定生命周期阶段使其自动执行打包操作：

```
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>2.2.1</version>
    <configuration>
      <descriptors>
        <descriptor>src/main/assembly/assembly.xml</descriptor>
      </descriptors>
    </configuration>
    <executions>
      <execution>
        <id>make-assembly</id>
        <phase>package</phase>
        <goals>
          <goal>single</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
```

运行**mvn clean package**之后，我们就能在target/目录下得到名为hello-world-1.0-bin.zip的分发包了。

小结

打包是项目构建最重要的组成部分之一，本文介绍了主流Maven打包技巧，包括默认打包方式的原理、如何制作源码包和Javadoc包、如何制作命 令行可运行的CLI包、以及进一步的，如何基于个性化需求自定义打包格式。这其中涉及了很多的Maven插件，当然最重要，也是最为复杂和强大的打包插件 就是maven-assembly-plugin。事实上Maven本身的分发包就是通过maven-assembly-plugin制作的，感兴趣的读 者可以直接查看源码一窥究竟。

本文已经首发于InfoQ中文站，版权所有，原文为《Maven实战（九）——打包的技巧》
