**超级详细的Maven使用教程**

转载[进修的CODER](https://me.csdn.net/lovequanquqn) 最后发布于2018-08-13 13:05:40 阅读数 34897  收藏

展开

**什么是Maven？**

如今我们构建一个项目需要用到很多第三方的类库，如写一个使用Spring的Web项目就需要引入大量的jar包。一个项目Jar包的数量之多往往让我们瞠目结舌，并且Jar包之间的关系错综复杂，一个Jar包往往又会引用其他Jar包，缺少任何一个Jar包都会导致项目编译失败。

以往开发项目时，程序员往往需要花较多的精力在引用Jar包搭建项目环境上，而这一项工作尤为艰难，少一个Jar包、多一个Jar包往往会报一些让人摸不着头脑的异常。

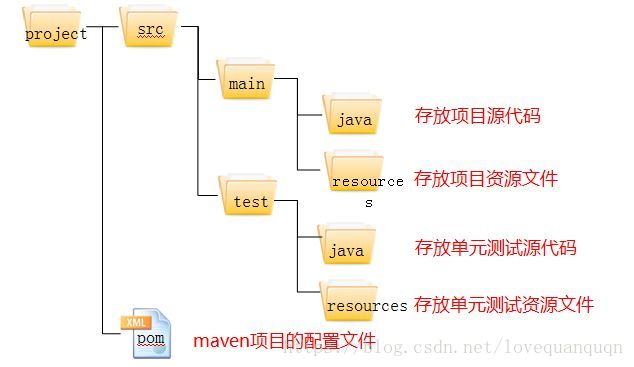
而Maven就是一款帮助程序员构建项目的工具，我们只需要告诉Maven需要哪些Jar 包，它会帮助我们下载所有的Jar，极大提升开发效率。

**安装Maven 和 Maven的Eclipse插件**

<http://blog.csdn.net/qjyong/article/details/9098213>

**Maven规定的目录结构**

若要使用Maven，那么项目的目录结构必须符合Maven的规范，其目录结构如下：



**Maven基本命令**

1. -v:查询Maven版本

本命令用于检查maven是否安装成功。

Maven安装完成之后，在命令行输入mvn -v，若出现maven信息，则说明安装成功。

1. compile：编译

将java源文件编译成class文件

1. test:测试项目

执行test目录下的测试用例

1. package:打包

将项目打成jar包

1. clean:删除target文件夹
2. install:安装

将当前项目放到Maven的本地仓库中。供其他项目使用

**什么是Maven仓库？**

Maven仓库用来存放Maven管理的所有Jar包。分为：本地仓库 和 中央仓库。

* 本地仓库：Maven本地的Jar包仓库。
* 中央仓库： Maven官方提供的远程仓库。

当项目编译时，Maven首先从本地仓库中寻找项目所需的Jar包，若本地仓库没有，再到Maven的中央仓库下载所需Jar包。

**什么是“坐标”？**

在Maven中，坐标是Jar包的唯一标识，Maven通过坐标在仓库中找到项目所需的Jar包。

如下代码中，groupId和artifactId构成了一个Jar包的坐标。

1. <dependency>
2. <groupId>cn.missbe.web.search</groupId>
3. <artifactId>resource-search</artifactId>
4. <packaging>jar</packaging>
5. <version>1.0-SNAPSHOT</version>
6. </dependency>

* groupId:所需Jar包的项目名
* artifactId:所需Jar包的模块名
* version:所需Jar包的版本号

**传递依赖 与 排除依赖**

* 传递依赖：如果我们的项目引用了一个Jar包，而该Jar包又引用了其他Jar包，那么在默认情况下项目编译时，Maven会把直接引用和简洁引用的Jar包都下载到本地。
* 排除依赖：如果我们只想下载直接引用的Jar包，那么需要在pom.xml中做如下配置：(将需要排除的Jar包的坐标写在中)

1. <exclusions>
2. <exclusion>
3. <groupId>cn.missbe.web.search</groupId>
4. <artifactId>resource-search</artifactId>
5. <packaging>pom</packaging>
6. <version>1.0-SNAPSHOT</version>
7. </exclusion>
8. </exclusions>

**依赖范围scope**

在项目发布过程中，帮助决定哪些构件被包括进来。欲知详情请参考依赖机制。      
- compile ：默认范围，用于编译        
- provided：类似于编译，但支持你期待jdk或者容器提供，类似于classpath        
- runtime: 在执行时需要使用        
- test:    用于test任务时使用        
- system: 需要外在提供相应的元素。通过systemPath来取得        
- systemPath: 仅用于范围为system。提供相应的路径        
  - optional:   当项目自身被依赖时，标注依赖是否传递。用于连续依赖时使用

**依赖冲突**

若项目中多个Jar同时引用了相同的Jar时，会产生依赖冲突，但Maven采用了两种避免冲突的策略，因此在Maven中是不存在依赖冲突的。

* 短路优先

1. 本项目——>A.jar——>B.jar——>X.jar
2. 本项目——>C.jar——>X.jar
3. 本项目——>A.jar——>B.jar——>X.jar
4. 本项目——>C.jar——>X.jar

若本项目引用了A.jar，A.jar又引用了B.jar，B.jar又引用了X.jar，并且C.jar也引用了X.jar。

在此时，Maven只会引用引用路径最短的Jar。

* 声明优先

若引用路径长度相同时，在pom.xml中谁先被声明，就使用谁。

**聚合**

1. 什么是聚合？

将多个项目同时运行就称为聚合。

1. 如何实现聚合？

只需在pom中作如下配置即可实现聚合：

1. <modules>
2. <module>web-connection-pool</module>
3. <module>web-java-crawler</module>
4. </modules>

**继承**

1. 什么是继承？

在聚合多个项目时，如果这些被聚合的项目中需要引入相同的Jar，那么可以将这些Jar写入父pom中，各个子项目继承该pom即可。

1. 如何实现继承？

* 父pom配置：将需要继承的Jar包的坐标放入标签即可。

1. <dependencyManagement>
2. <dependencies>
3. <dependency>
4. <groupId>cn.missbe.web.search</groupId>
5. <artifactId>resource-search</artifactId>
6. <packaging>pom</packaging>
7. <version>1.0-SNAPSHOT</version>
8. </dependency>
9. </dependencies>
10. </dependencyManagement>

* 子pom配置：

1. <parent>
2. <groupId>父pom所在项目的groupId</groupId>
3. <artifactId>父pom所在项目的artifactId</artifactId>
4. <version>父pom所在项目的版本号</version>
5. </parent>
6. <parent>
7. <artifactId>resource-search</artifactId>
8. <groupId>cn.missbe.web.search</groupId>
9. <version>1.0-SNAPSHOT</version>
10. </parent>

**使用Maven构建Web项目**

1. New Maven项目：选择WebApp：
2. 若使用JSP，需添加Servlet依赖：

注：Servlet依赖只在编译和测试时使用！

1. <dependency>
2. <groupId>javax.servlet</groupId>
3. <artifactId>javax.servlet-api</artifactId>
4. <version>3.0.1</version>
5. <!-- 只在编译和测试时运行 -->
6. <scope>provided</scope>
7. </dependency>
8. 在Bulid Path中设置resource输出目录：
9. 勾选：Dynamic Web Module
10. 删掉测试目录
11. 在pom中加入jetty的插件，并设置JDK版本：
12. <plugins>
13. <plugin>
14. <groupId>org.apache.maven.plugins</groupId>
15. <artifactId>maven-compiler-plugin</artifactId>
16. <configuration>
17. <source>1.8</source>
18. <target>1.8</target>
19. </configuration>
20. </plugin>
22. <plugin>
23. <groupId>org.eclipse.jetty</groupId>
24. <artifactId>jetty-maven-plugin</artifactId>
25. <version>9.3.10.v20160621</version>
26. <executions>
27. <execution>
28. <phase>package</phase>
29. </execution>
30. </executions>
31. </plugin>
32. </plugins>
33. 运行项目：
34. 输入：jetty:run
35. 访问127.0.0.1:8080

若出现如下界面，表示成功！

**pom.xml详解**

pom.xml是Maven的核心，你的项目需要什么Jar包就在pom.xml里面配置。当编译项目时Maven读取该文件，并从仓库中下载相应的Jar包。

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0http://maven.apache.org/maven-v4\_0\_0.xsd">
4. <!--父项目的坐标。如果项目中没有规定某个元素的值，
5. 那么父项目中的对应值即为项目的默认值。
6. 坐标包括group ID，artifact ID和 version。-->
7. <parent>
8. <!--被继承的父项目的构件标识符-->
9. <artifactId/>
10. <!--被继承的父项目的全球唯一标识符-->
11. <groupId/>
12. <!--被继承的父项目的版本-->
13. <version/>
14. </parent>
15. <!--声明项目描述符遵循哪一个POM模型版本。模型本身的版本很少改变，虽然如此，
16. 但它仍然是必不可少的，这是为了当Maven引入了新的特性或者其他模型变更的时候，
17. 确保稳定性。-->
18. <modelVersion>4.0.0</modelVersion>
19. <!--项目的全球唯一标识符，通常使用全限定的包名区分该项目和其他项目。
20. 并且构建时生成的路径也是由此生成， 如com.mycompany.app生成的相对路径为：
21. /com/mycompany/app-->
22. <groupId>cn.missbe.web</groupId>
23. <!-- 构件的标识符，它和group ID一起唯一标识一个构件。换句话说，
24. 你不能有两个不同的项目拥有同样的artifact ID和groupID；在某个
25. 特定的group ID下，artifact ID也必须是唯一的。构件是项目产生的或使用的一个东西，
26. Maven为项目产生的构件包括：JARs，源码，二进制发布和WARs等。-->
27. <artifactId>search-resources</artifactId>
28. <!--项目产生的构件类型，例如jar、war、ear、pom。插件可以创建
29. 他们自己的构件类型，所以前面列的不是全部构件类型-->
30. <packaging>war</packaging>
31. <!--项目当前版本，格式为:主版本.次版本.增量版本-限定版本号-->
32. <version>1.0-SNAPSHOT</version>
33. <!--项目的名称, Maven产生的文档用-->
34. <name>search-resources</name>
35. <!--项目主页的URL, Maven产生的文档用-->
36. <url>http:*//www.missbe.cn</url>*
37. <!-- 项目的详细描述, Maven 产生的文档用。 当这个元素能够用HTML格式描述时
38. （例如，CDATA中的文本会被解析器忽略，就可以包含HTML标 签），
39. 不鼓励使用纯文本描述。如果你需要修改产生的web站点的索引页面，
40. 你应该修改你自己的索引页文件，而不是调整这里的文档。-->
41. <description>A maven project to study maven.</description>
42. <!--描述了这个项目构建环境中的前提条件。-->
43. <prerequisites>
44. <!--构建该项目或使用该插件所需要的Maven的最低版本-->
45. <maven/>
46. </prerequisites>
48. <!--构建项目需要的信息-->
49. <build>
50. <!--该元素设置了项目源码目录，当构建项目的时候，
51. 构建系统会编译目录里的源码。该路径是相对于pom.xml的相对路径。-->
52. <sourceDirectory/>
53. <!--该元素设置了项目脚本源码目录，该目录和源码目录不同：
54. 绝大多数情况下，该目录下的内容 会被拷贝到输出目录(因为脚本是被解释的，而不是被编译的)。-->
55. <scriptSourceDirectory/>
56. <!--该元素设置了项目单元测试使用的源码目录，当测试项目的时候，
57. 构建系统会编译目录里的源码。该路径是相对于pom.xml的相对路径。-->
58. <testSourceDirectory/>
59. <!--被编译过的应用程序class文件存放的目录。-->
60. <outputDirectory/>
61. <!--被编译过的测试class文件存放的目录。-->
62. <testOutputDirectory/>
63. <!--使用来自该项目的一系列构建扩展-->
64. <extensions>
65. <!--描述使用到的构建扩展。-->
66. <extension>
67. <!--构建扩展的groupId-->
68. <groupId/>
69. <!--构建扩展的artifactId-->
70. <artifactId/>
71. <!--构建扩展的版本-->
72. <version/>
73. </extension>
74. </extensions>
76. <!--这个元素描述了项目相关的所有资源路径列表，例如和项目相关的属性文件，
77. 这些资源被包含在最终的打包文件里。-->
78. <resources>
79. <!--这个元素描述了项目相关或测试相关的所有资源路径-->
80. <resource>
81. <!-- 描述了资源的目标路径。该路径相对target/classes目录（例如${project.build.outputDirectory}）。举个例 子，如果你想资源在特定的包里(org.apache.maven.messages)，你就必须该元素设置为org/apache/maven /messages。
82. 然而，如果你只是想把资源放到源码目录结构里，就不需要该配置。-->
83. <targetPath/>
84. <!--是否使用参数值代替参数名。参数值取自properties元素或者文件里配置的属性，
85. 文件在filters元素里列出。-->
86. <filtering/>
87. <!--描述存放资源的目录，该路径相对POM路径-->
88. <directory/>
89. <!--包含的模式列表，例如\*\**/\*.xml.-->*
90. *<includes/>*
91. *<!--排除的模式列表，例如\*\*/*\*.xml-->
92. <excludes/>
93. </resource>
94. </resources>
95. <!--这个元素描述了单元测试相关的所有资源路径，例如和单元测试相关的属性文件。-->
96. <testResources>
97. <!--这个元素描述了测试相关的所有资源路径，参见build/resources/resource元素的说明-->
98. <testResource>
99. <targetPath/><filtering/><directory/><includes/><excludes/>
100. </testResource>
101. </testResources>
102. <!--构建产生的所有文件存放的目录-->
103. <directory/>
104. <!--产生的构件的文件名，默认值是${artifactId}-${version}。-->
105. <finalName/>
106. <!--当filtering开关打开时，使用到的过滤器属性文件列表-->
107. <filters/>
108. <!--子项目可以引用的默认插件信息。该插件配置项直到被引用时才会被解析或绑定到生命周期。
109. 给定插件的任何本地配置都会覆盖这里的配置-->
110. <pluginManagement>
111. <!--使用的插件列表 。-->
112. <plugins>
113. <!--plugin元素包含描述插件所需要的信息。-->
114. <plugin>
115. <!--插件在仓库里的group ID-->
116. <groupId/>
117. <!--插件在仓库里的artifact ID-->
118. <artifactId/>
119. <!--被使用的插件的版本（或版本范围）-->
120. <version/>
121. <!--是否从该插件下载Maven扩展（例如打包和类型处理器），由于性能原因，
122. 只有在真需要下载时，该元素才被设置成enabled。-->
123. <extensions/>
124. <!--在构建生命周期中执行一组目标的配置。每个目标可能有不同的配置。-->
125. <executions>
126. <!--execution元素包含了插件执行需要的信息-->
127. <execution>
128. <!--执行目标的标识符，用于标识构建过程中的目标，或者匹配继承过程中需要合并的执行目标-->
129. <id/>
130. <!--绑定了目标的构建生命周期阶段，如果省略，目标会被绑定到源数据里配置的默认阶段-->
131. <phase/>
132. <!--配置的执行目标-->
133. <goals/>
134. <!--配置是否被传播到子POM-->
135. <inherited/>
136. <!--作为DOM对象的配置-->
137. <configuration/>
138. </execution>
139. </executions>
140. <!--项目引入插件所需要的额外依赖-->
141. <dependencies>
142. <!--参见dependencies/dependency元素-->
143. <dependency>
144. ......
145. </dependency>
146. </dependencies>
147. <!--任何配置是否被传播到子项目-->
148. <inherited/>
149. <!--作为DOM对象的配置-->
150. <configuration/>
151. </plugin>
152. </plugins>
153. </pluginManagement>
154. <!--使用的插件列表-->
155. <plugins>
156. <!--参见build/pluginManagement/plugins/plugin元素-->
157. <plugin>
158. <groupId/><artifactId/><version/><extensions/>
159. <executions>
160. <execution>
161. <id/><phase/><goals/><inherited/><configuration/>
162. </execution>
163. </executions>
164. <dependencies>
165. <!--参见dependencies/dependency元素-->
166. <dependency>
167. ......
168. </dependency>
169. </dependencies>
170. <goals/><inherited/><configuration/>
171. </plugin>
172. </plugins>
173. </build>
174. <!--模块（有时称作子项目） 被构建成项目的一部分。
175. 列出的每个模块元素是指向该模块的目录的相对路径-->
176. <modules/>
177. <!--发现依赖和扩展的远程仓库列表。-->
178. <repositories>
179. <!--包含需要连接到远程仓库的信息-->
180. <repository>
181. <!--如何处理远程仓库里发布版本的下载-->
182. <releases>
183. <!--true或者false表示该仓库是否为下载某种类型构件（发布版，快照版）开启。 -->
184. <enabled/>
185. <!--该元素指定更新发生的频率。Maven会比较本地POM和远程POM的时间戳。这里的选项是：always（一直），daily（默认，每日），interval：X（这里X是以分钟为单位的时间间隔），或者never（从不）。-->
186. <updatePolicy/>
187. <!--当Maven验证构件校验文件失败时该怎么做：ignore（忽略），fail（失败），或者warn（警告）。-->
188. <checksumPolicy/>
189. </releases>
190. <!-- 如何处理远程仓库里快照版本的下载。有了releases和snapshots这两组配置，
191. POM就可以在每个单独的仓库中，为每种类型的构件采取不同的 策略。
192. 例如，可能有人会决定只为开发目的开启对快照版本下载的支持。
193. 参见repositories/repository/releases元素 -->
194. <snapshots>
195. <enabled/><updatePolicy/><checksumPolicy/>
196. </snapshots>
197. <!--远程仓库唯一标识符。可以用来匹配在settings.xml文件里配置的远程仓库-->
198. <id>banseon-repository-proxy</id>
199. <!--远程仓库名称-->
200. <name>banseon-repository-proxy</name>
201. <!--远程仓库URL，按protocol:*//hostname/path形式-->*
202. <url>http:*//192.168.1.169:9999/repository/</url>*
203. <!-- 用于定位和排序构件的仓库布局类型-可以是default（默认）或者legacy（遗留）。Maven 2为其仓库提供了一个默认的布局；然 而，Maven 1.x有一种不同的布局。我们可以使用该元素指定布局是default（默认）还是legacy（遗留）。-->
204. <layout>default</layout>
205. </repository>
206. </repositories>
207. <!--发现插件的远程仓库列表，这些插件用于构建和报表-->
208. <pluginRepositories>
209. <!--包含需要连接到远程插件仓库的信息.参见repositories/repository元素-->
210. <pluginRepository>
211. ......
212. </pluginRepository>
213. </pluginRepositories>
215. <!--该元素描述了项目相关的所有依赖。 这些依赖组成了项目构建过程中的一个个环节。
216. 它们自动从项目定义的仓库中下载。要获取更多信息，请看项目依赖机制。-->
217. <dependencies>
218. <dependency>
219. <!--依赖的group ID-->
220. <groupId>org.apache.maven</groupId>
221. <!--依赖的artifact ID-->
222. <artifactId>maven-artifact</artifactId>
223. <!--依赖的版本号。 在Maven 2里, 也可以配置成版本号的范围。-->
224. <version>3.8.1</version>
225. <!-- 依赖类型，默认类型是jar。它通常表示依赖的文件的扩展名，但也有例外
226. 。一个类型可以被映射成另外一个扩展名或分类器。类型经常和使用的打包方式对应，
227. 尽管这也有例外。一些类型的例子：jar，war，ejb-client和test-jar。
228. 如果设置extensions为 true，就可以在 plugin里定义新的类型。所以前面的类型的例子不完整。-->
229. <type>jar</type>
230. <!-- 依赖的分类器。分类器可以区分属于同一个POM，但不同构建方式的构件。
231. 分类器名被附加到文件名的版本号后面。例如，如果你想要构建两个单独的构件成 JAR，
232. 一个使用Java 1.4编译器，另一个使用Java 6编译器，你就可以使用分类器来生成两个单独的JAR构件。-->
233. <classifier></classifier>
234. <!--依赖范围。在项目发布过程中，帮助决定哪些构件被包括进来。欲知详情请参考依赖机制。
235. - compile ：默认范围，用于编译
236. - provided：类似于编译，但支持你期待jdk或者容器提供，类似于classpath
237. - runtime: 在执行时需要使用
238. - test: 用于test任务时使用
239. - system: 需要外在提供相应的元素。通过systemPath来取得
240. - systemPath: 仅用于范围为system。提供相应的路径
241. - optional: 当项目自身被依赖时，标注依赖是否传递。用于连续依赖时使用-->
242. <scope>test</scope>
243. <!--仅供system范围使用。注意，不鼓励使用这个元素，
244. 并且在新的版本中该元素可能被覆盖掉。该元素为依赖规定了文件系统上的路径。
245. 需要绝对路径而不是相对路径。推荐使用属性匹配绝对路径，例如${java.home}。-->
246. <systemPath></systemPath>
247. <!--当计算传递依赖时， 从依赖构件列表里，列出被排除的依赖构件集。
248. 即告诉maven你只依赖指定的项目，不依赖项目的依赖。此元素主要用于解决版本冲突问题-->
249. <exclusions>
250. <exclusion>
251. <artifactId>spring-core</artifactId>
252. <groupId>org.springframework</groupId>
253. </exclusion>
254. </exclusions>
255. <!--可选依赖，如果你在项目B中把C依赖声明为可选，你就需要在依赖于B的项目（例如项目A）中显式的引用对C的依赖。可选依赖阻断依赖的传递性。-->
256. <optional>true</optional>
257. </dependency>
258. </dependencies>

261. <!-- 继承自该项目的所有子项目的默认依赖信息。这部分的依赖信息不会被立即解析,
262. 而是当子项目声明一个依赖（必须描述group ID和 artifact ID信息），
263. 如果group ID和artifact ID以外的一些信息没有描述，
264. 则通过group ID和artifact ID 匹配到这里的依赖，并使用这里的依赖信息。-->
265. <dependencyManagement>
266. <dependencies>
267. <!--参见dependencies/dependency元素-->
268. <dependency>
269. ......
270. </dependency>
271. </dependencies>
272. </dependencyManagement>
273. <!--项目分发信息，在执行mvn deploy后表示要发布的位置。
274. 有了这些信息就可以把网站部署到远程服务器或者把构件部署到远程仓库。-->
275. <distributionManagement>
276. <!--部署项目产生的构件到远程仓库需要的信息-->
277. <repository>
278. <!--是分配给快照一个唯一的版本号（由时间戳和构建流水号）？
279. 还是每次都使用相同的版本号？参见repositories/repository元素-->
280. <uniqueVersion/>
281. <id>banseon-maven2</id>
282. <name>banseon maven2</name>
283. <url>file:*//${basedir}/target/deploy</url>*
284. <layout/>
285. </repository>
286. <!--构件的快照部署到哪里？如果没有配置该元素，默认部署到repository元素配置的仓库，
287. 参见distributionManagement/repository元素-->
288. <snapshotRepository>
289. <uniqueVersion/>
290. <id>banseon-maven2</id>
291. <name>Banseon-maven2 Snapshot Repository</name>
292. <url>scp:*//svn.baidu.com/banseon:/usr/local/maven-snapshot</url>*
293. <layout/>
294. </snapshotRepository>
295. <!--部署项目的网站需要的信息-->
296. <site>
297. <!--部署位置的唯一标识符，用来匹配站点和settings.xml文件里的配置-->
298. <id>banseon-site</id>
299. <!--部署位置的名称-->
300. <name>business api website</name>
301. <!--部署位置的URL，按protocol:*//hostname/path形式-->*
302. <url>
303. scp:*//svn.baidu.com/banseon:/var/www/localhost/banseon-web*
304. </url>
305. </site>
306. <!--项目下载页面的URL。如果没有该元素，用户应该参考主页。
307. 使用该元素的原因是：帮助定位那些不在仓库里的构件（由于license限制）。-->
308. <downloadUrl/>
309. <!-- 给出该构件在远程仓库的状态。不得在本地项目中设置该元素，
310. 因为这是工具自动更新的。有效的值有：none（默认），
311. converted（仓库管理员从 Maven 1 POM转换过来），partner（直接从伙伴Maven 2仓库同步过来），deployed（从Maven 2实例部 署），verified（被核实时正确的和最终的）。-->
312. <status/>
313. </distributionManagement>
314. <!--以值替代名称，Properties可以在整个POM中使用，也可以作为触发条件（见settings.xml配置文件里activation元素的说明）。格式是<name>value</name>。-->
315. <properties/>
316. </project>