https://blog.csdn.net/lovequanquqn/article/details/81627807

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。

2. compile：编译

   将 java 源文件编译成 class 文件

3. test:测试项目

   执行 test 目录下的测试用例

4. package:打包

   将项目打成 jar 包

5. clean:删除 target 文件夹
6. 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。

```
<dependency>
    <groupId>cn.missbe.web.search</groupId>
    <artifactId>resource-search</artifactId>
```

```
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
</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)

```
<exclusions>
    <exclusion>
        <groupId>cn.missbe.web.search</groupId>
        <artifactId>resource-search</artifactId>
        <packaging>pom</packaging>
        <version>1.0-SNAPSHOT</version>
    </exclusion>
</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

1. 本项目——>A.jar——>B.jar——>X.jar
2. 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。

2. 如何实现聚合？

   只需在 pom 中作如下配置即可实现聚合：

```
<modules>
    <module>web-connection-pool</module>
    <module>web-java-crawler</module>
</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。

2. 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。

```
<dependencyManagement>
    <dependencies>
        <dependency>
          <groupId>cn.missbe.web.search</groupId>
          <artifactId>resource-search</artifactId>
          <packaging>pom</packaging>
```

```
            <version>1.0-SNAPSHOT</version>
          </dependency>
    </dependencies>
</dependencyManagement>
```

- 子 pom 配置:

```
<parent>
        <groupId>父 pom 所在项目的 groupId</groupId>
        <artifactId>父 pom 所在项目的 artifactId</artifactId>
        <version>父 pom 所在项目的版本号</version>
</parent>
 <parent>
        <artifactId>resource-search</artifactId>
        <groupId>cn.missbe.web.search</groupId>
        <version>1.0-SNAPSHOT</version>
</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！

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
    <!-- 只在编译和测试时运行 -->
    <scope>provided</scope>
</dependency>
```

1. 在 Bulid Path 中设置 resource 输出目录；
2. 勾选：Dynamic Web Module
3. 删掉测试目录
4. 在 pom 中加入 jetty 的插件，并设置 JDK 版本：

```
<plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
```

```
        </plugin>

        <plugin>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-maven-plugin</artifactId>
            <version>9.3.10.v20160621</version>
            <executions>
                <execution>
                    <phase>package</phase>
                </execution>
            </executions>
        </plugin>
</plugins>
```

1. 运行项目：
2. 输入：jetty:run
3. 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。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0http://maven.apache.org/maven-v4_0_0.xsd">
    <!--父项目的坐标。如果项目中没有规定某个元素的值，
那么父项目中的对应值即为项目的默认值。
坐标包括 group ID，artifact ID 和 version。-->
    <parent>
     <!--被继承的父项目的构件标识符-->
     <artifactId/>
     <!--被继承的父项目的全球唯一标识符-->
     <groupId/>
     <!--被继承的父项目的版本-->
     <version/>
 </parent>
 <!--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，
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>
47.
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>
75.
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>
214.
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。-->
```

```
            <classifier></classifier>
            <!--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。
                - 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-->
            <scope>test</scope>
            <!--仅供 system 范围使用。注意，不鼓励使用这个元素，
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。
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}。-->
            <systemPath></systemPath>
            <!--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。
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-->
            <exclusions>
             <exclusion>
                    <artifactId>spring-core</artifactId>
                    <groupId>org.springframework</groupId>
                </exclusion>
            </exclusions>
            <!--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。-->
            <optional>true</optional>
        </dependency>
    </dependencies>


 <!-- 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,
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>
```

参考原文：Maven 使用详解

本节我们介绍 Maven 的排除依赖、归类依赖和优化依赖。

排除依赖

Maven 的传递依赖能自动将间接依赖引入项目中来，这样极大地简化了项目中的依赖管理，

但是，有时候这种自动化也会带来麻烦。

比如 Maven 可能会自动引入快照版本的依赖，而快照版本的依赖是不稳定的，这时候就

需要避免引入快照版本。这样的话需要用一种方式告知 Maven 排除快照版本的依赖引入，这种做法就是排除依赖。那怎么实现排除依赖呢?

其实实现排除依赖还是比较简单的，在直接依赖的配置里面添加 exclusions→exclusion 元素，指定要排除依赖的 groupId 和 artifactId 就行，如下面代码所示。

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>${project.build.hibernate.version}</version>
    <exclusions>
        <exclusion>
            <groupId>xxx</groupId>
            <artifactId>xxx</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

归类依赖

在引用依赖的时候，很多情况需要引入一个 Maven 项目的多个模块，这些模块都应该是相同的版本。比如，用户在 Spring 框架下开发应用，就需要同时引用 org.springframework 的 spring-core、spring-context、spring-context-support 等模块。

可以想象，这些模块肯定是统一的版本，如果在每个依赖里面都分别用 groupId、artifactId 和 version 具体指明的话，例如下次升级，需要将 2.5 版本升级成 3.0 版本，这样就需要将 org.springframework 的每个模块的版本都统一更改，这样做很容易出现不一致的情况，就很容易出错。

为了避免出现这种情况，可以在 pom.xml 中定义一个属性名称描述版本的值。接下来在每个 version 中，用特殊的语法引用这个属性名称。实际引入的时候，由 Maven 将属性改成对应的值。这样就可以统一版本，也方便修改。具体样例代码如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <!-- 3.2.16.RELEASE,3.1.4.RELEASE -->

<project.build.spring.version>4.2.7.RELEASE</project.build.spring.version>
    </properties>

    <dependencies>
        <!-- spring -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${project.build.spring.version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>${project.build.spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
```

```
            <version>${project.build.spring.version}</version>
        </dependency>
        ...
    </dependencies>
    ...
</project>
```

优化依赖

程序员在软件开发过程中，需要通过重构等方式不断优化代码，使其变得更简洁、灵活、高效。同样，用户也应该对 Maven 项目的依赖了然于胸，并对其进行优化。

通过教程前面的了解，可以理解 Maven 定位依赖的方式、传递依赖的规则以及怎么样排除依赖等。但是要实现这些动作，还必须对项目中的依赖有全面的了解，这样才能更有效地达到目的。

接下来介绍一下查看依赖的相关命令。

- Mvn dependency:list，列出所有的依赖列表。
- Mvn dependency:tree，以树形结构方式，列出依赖和层次关系。
- Mvn dependency:analyze，分析主代码、测试代码编译的依赖。