Maven

Maven

- 约定配置
- POM
- 构建生命周期
- 构建配置文件
- 仓库
- 插件
- Maven构建Java项目
- Maven项目文档
- Maven快照(SNAPSHOT)
- Maven自动化构建
- Maven依赖管理
- Maven自动化部署

约定配置

目录	目的
${basedir}	存放pom.xml和所有的子目录
${basedir}/src/main/java	项目的java源代码
${basedir}/src/main/resources	项目的资源，比如说property文件，springmvc.xml
${basedir}/src/test/java	项目的测试类，比如说Junit代码
${basedir}/src/test/resources	测试用的资源
${basedir}/src/main/webapp/WEB-INF	web应用文件目录，web项目的信息，比如存放web.xml、本地图片、jsp视图页面
${basedir}/target	打包输出目录
${basedir}/target/classes	编译输出目录
${basedir}/target/test-classes	测试编译输出目录
Test.java	Maven只会自动运行符合该命名规则的测试类
~/.m2/repository	Maven默认的本地仓库目录位置

POM

```
<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/>
    <!-- 父项目的pom.xml文件的相对路径。相对路径允许你选择一个不同的路径。默认值是../pom.xml。Maven首先在构建当前项目的地方寻找父项
        目的pom，其次在文件系统的这个位置（relativePath位置），然后在本地仓库，最后在远程仓库寻找父项目的pom。 -->
    <relativePath/>
  </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，确保稳定性。 -->
  <modelVersion>4.0.0</modelVersion>
  <!--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。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：/com/mycompany/app -->
  <groupId>asia.banseon</groupId>
  <!-- 构件的标识符，它和group ID一起唯一标识一个构件。换句话说，你不能有两个不同的项目拥有同样的artifact ID和groupID；在某个
      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，Maven为项目产生的构件包括：JARs，源 码，二进制发布和WARs等。 -->
  <artifactId>banseon-maven2</artifactId>
  <!--项目产生的构件类型，例如jar、war、ear、pom。插件可以创建他们自己的构件类型，所以前面列的不是全部构件类型 -->
  <packaging>jar</packaging>
  <!--项目当前版本，格式为:主版本.次版本.增量版本-限定版本号 -->
  <version>1.0-SNAPSHOT</version>
  <!--项目的名称，Maven产生的文档用 -->
  <name>banseon-maven</name>
  <!--项目主页的URL，Maven产生的文档用 -->
  <url>http://www.baidu.com/banseon</url>
```

```
  <!-- 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（例如，CDATA中的文本会被解析器忽略，就可以包含HTML标
       签）， 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，你应该修改你自己的索引页文件，而不是调整这里的文档。 -->
  <description>A maven project to study maven.</description>
  <!--描述了这个项目构建环境中的前提条件。 -->
  <prerequisites>
    <!--构建该项目或使用该插件所需要的Maven的最低版本 -->
    <maven/>
  </prerequisites>
  <!--项目的问题管理系统(Bugzilla, Jira, Scarab,或任何你喜欢的问题管理系统)的名称和URL，本例为 jira -->
  <issueManagement>
    <!--问题管理系统（例如jira）的名字， -->
    <system>jira</system>
    <!--该项目使用的问题管理系统的URL -->
    <url>http://jira.baidu.com/banseon</url>
  </issueManagement>
  <!--项目持续集成信息 -->
  <ciManagement>
    <!--持续集成系统的名字，例如continuum -->
    <system/>
    <!--该项目使用的持续集成系统的URL（如果持续集成系统有web接口的话）。 -->
    <url/>
    <!--构建完成时，需要通知的开发者/用户的配置项。包括被通知者信息和通知条件（错误，失败，成功，警告） -->
    <notifiers>
      <!--配置一种方式，当构建中断时，以该方式通知用户/开发者 -->
      <notifier>
        <!--传送通知的途径 -->
        <type/>
        <!--发生错误时是否通知 -->
        <sendOnError/>
        <!--构建失败时是否通知 -->
        <sendOnFailure/>
        <!--构建成功时是否通知 -->
        <sendOnSuccess/>
        <!--发生警告时是否通知 -->
        <sendOnWarning/>
        <!--不赞成使用。通知发送到哪里 -->
        <address/>
        <!--扩展配置项 -->
        <configuration/>
      </notifier>
    </notifiers>
```

```
  </ciManagement>
  <!--项目创建年份，4位数字。当产生版权信息时需要使用这个值。 -->
  <inceptionYear/>
  <!--项目相关邮件列表信息 -->
  <mailingLists>
    <!--该元素描述了项目相关的所有邮件列表。自动产生的网站引用这些信息。 -->
    <mailingList>
      <!--邮件的名称 -->
      <name>Demo</name>
      <!--发送邮件的地址或链接，如果是邮件地址，创建文档时，mailto: 链接会被自动创建 -
->
      <post>banseon@126.com</post>
      <!--订阅邮件的地址或链接，如果是邮件地址，创建文档时，mailto: 链接会被自动创建 -
->
      <subscribe>banseon@126.com</subscribe>
      <!--取消订阅邮件的地址或链接，如果是邮件地址，创建文档时，mailto: 链接会被自动创
建 -->
      <unsubscribe>banseon@126.com</unsubscribe>
      <!--你可以浏览邮件信息的URL -->
      <archive>http:/hi.baidu.com/banseon/demo/dev/</archive>
    </mailingList>
  </mailingLists>
  <!--项目开发者列表 -->
  <developers>
    <!--某个项目开发者的信息 -->
    <developer>
      <!--SCM里项目开发者的唯一标识符 -->
      <id>HELLO WORLD</id>
      <!--项目开发者的全名 -->
      <name>banseon</name>
      <!--项目开发者的email -->
      <email>banseon@126.com</email>
      <!--项目开发者的主页的URL -->
      <url/>
      <!--项目开发者在项目中扮演的角色，角色元素描述了各种角色 -->
      <roles>
        <role>Project Manager</role>
        <role>Architect</role>
      </roles>
      <!--项目开发者所属组织 -->
      <organization>demo</organization>
      <!--项目开发者所属组织的URL -->
      <organizationUrl>http://hi.baidu.com/banseon</organizationUrl>
      <!--项目开发者属性，如即时消息如何处理等 -->
      <properties>
```

```
        <dept>No</dept>
      </properties>
      <!--项目开发者所在时区， -11到12范围内的整数。 -->
      <timezone>-5</timezone>
    </developer>
  </developers>
  <!--项目的其他贡献者列表 -->
  <contributors>
    <!--项目的其他贡献者。参见developers/developer元素 -->
    <contributor>
      <name/>
      <email/>
      <url/>
      <organization/>
      <organizationUrl/>
      <roles/>
      <timezone/>
      <properties/>
    </contributor>
  </contributors>
  <!--该元素描述了项目所有License列表。 应该只列出该项目的license列表，不要列出依赖项目的 license列表。如果列出多个license，用户可以选择它们中的一个而不是接受所有license。 -->
  <licenses>
    <!--描述了项目的license，用于生成项目的web站点的license页面，其他一些报表和validation也会用到该元素。 -->
    <license>
      <!--license用于法律上的名称 -->
      <name>Apache 2</name>
      <!--官方的license正文页面的URL -->
      <url>http://www.baidu.com/banseon/LICENSE-2.0.txt</url>
      <!--项目分发的主要方式： repo，可以从Maven库下载 manual， 用户必须手动下载和安装依赖 -->
      <distribution>repo</distribution>
      <!--关于license的补充信息 -->
      <comments>A business-friendly OSS license</comments>
    </license>
  </licenses>
  <!--SCM(Source Control Management)标签允许你配置你的代码库，供Maven web站点和其它插件使用。 -->
  <scm>
    <!--SCM的URL,该URL描述了版本库和如何连接到版本库。欲知详情，请看SCMs提供的URL格式和列表。该连接只读。 -->
    <connection>
```

```
      scm:svn:http://svn.baidu.com/banseon/maven/banseon/banseon-maven2-
trunk(dao-trunk)
    </connection>
    <!--给开发者使用的，类似connection元素。即该连接不仅仅只读 -->
    <developerConnection>
      scm:svn:http://svn.baidu.com/banseon/maven/banseon/dao-trunk
    </developerConnection>
    <!--当前代码的标签，在开发阶段默认为HEAD -->
    <tag/>
    <!--指向项目的可浏览SCM库（例如ViewVC或者Fisheye）的URL。 -->
    <url>http://svn.baidu.com/banseon</url>
  </scm>
  <!--描述项目所属组织的各种属性。Maven产生的文档用 -->
  <organization>
    <!--组织的全名 -->
    <name>demo</name>
    <!--组织主页的URL -->
    <url>http://www.baidu.com/banseon</url>
  </organization>
  <!--构建项目需要的信息 -->
  <build>
    <!--该元素设置了项目源码目录，当构建项目的时候，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。 -->
    <sourceDirectory/>
    <!--该元素设置了项目脚本源码目录，该目录和源码目录不同：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)。 -->
    <scriptSourceDirectory/>
    <!--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，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。 -->
    <testSourceDirectory/>
    <!--被编译过的应用程序class文件存放的目录。 -->
    <outputDirectory/>
    <!--被编译过的测试class文件存放的目录。 -->
    <testOutputDirectory/>
    <!--使用来自该项目的一系列构建扩展 -->
    <extensions>
      <!--描述使用到的构建扩展。 -->
      <extension>
        <!--构建扩展的groupId -->
        <groupId/>
        <!--构建扩展的artifactId -->
        <artifactId/>
        <!--构建扩展的版本 -->
        <version/>
      </extension>
```

```
    </extensions>
    <!--当项目没有规定目标（Maven2 叫做阶段）时的默认值 -->
    <defaultGoal/>
    <!--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，这些资源被包含在最终的打包文件里。 -->
    <resources>
      <!--这个元素描述了项目相关或测试相关的所有资源路径 -->
      <resource>
        <!-- 描述了资源的目标路径。该路径相对target/classes目录（例如${project.build.outputDirectory}）。举个例
            子，如果你想资源在特定的包里(org.apache.maven.messages)，你就必须该元素设置为org/apache/maven /messages。然而，如果你只是想把资源放到源码目录结构里，就不需要该配置。 -->
        <targetPath/>
        <!--是否使用参数值代替参数名。参数值取自properties元素或者文件里配置的属性，文件在filters元素里列出。 -->
        <filtering/>
        <!--描述存放资源的目录，该路径相对POM路径 -->
        <directory/>
        <!--包含的模式列表，例如**/*.xml. -->
        <includes/>
        <!--排除的模式列表，例如**/*.xml -->
        <excludes/>
      </resource>
    </resources>
    <!--这个元素描述了单元测试相关的所有资源路径，例如和单元测试相关的属性文件。 -->
    <testResources>
      <!--这个元素描述了测试相关的所有资源路径，参见build/resources/resource元素的说明 -->
      <testResource>
        <targetPath/>
        <filtering/>
        <directory/>
        <includes/>
        <excludes/>
      </testResource>
    </testResources>
    <!--构建产生的所有文件存放的目录 -->
    <directory/>
    <!--产生的构件的文件名，默认值是${artifactId}-${version}。 -->
    <finalName/>
    <!--当filtering开关打开时，使用到的过滤器属性文件列表 -->
    <filters/>
    <!--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。给定插件的任何本地配置都会覆盖这里的配置 -->
```

```
    <pluginManagement>
      <!--使用的插件列表 。 -->
      <plugins>
        <!--plugin元素包含描述插件所需要的信息。 -->
        <plugin>
          <!--插件在仓库里的group ID -->
          <groupId/>
          <!--插件在仓库里的artifact ID -->
          <artifactId/>
          <!--被使用的插件的版本（或版本范围） -->
          <version/>
          <!--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，只有在
真需要下载时，该元素才被设置成enabled。 -->
          <extensions/>
          <!--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。 -->
          <executions>
            <!--execution元素包含了插件执行需要的信息 -->
            <execution>
              <!--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 -->
              <id/>
              <!--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 -->
              <phase/>
              <!--配置的执行目标 -->
              <goals/>
              <!--配置是否被传播到子POM -->
              <inherited/>
              <!--作为DOM对象的配置 -->
              <configuration/>
            </execution>
          </executions>
          <!--项目引入插件所需要的额外依赖 -->
          <dependencies>
            <!--参见dependencies/dependency元素 -->
            <dependency>
              ......
            </dependency>
          </dependencies>
          <!--任何配置是否被传播到子项目 -->
          <inherited/>
          <!--作为DOM对象的配置 -->
          <configuration/>
        </plugin>
      </plugins>
```

```
    </pluginManagement>
    <!--使用的插件列表 -->
    <plugins>
      <!--参见build/pluginManagement/plugins/plugin元素 -->
      <plugin>
        <groupId/>
        <artifactId/>
        <version/>
        <extensions/>
        <executions>
          <execution>
            <id/>
            <phase/>
            <goals/>
            <inherited/>
            <configuration/>
          </execution>
        </executions>
        <dependencies>
          <!--参见dependencies/dependency元素 -->
          <dependency>
            ......
          </dependency>
        </dependencies>
        <goals/>
        <inherited/>
        <configuration/>
      </plugin>
    </plugins>
  </build>
  <!--在列的项目构建profile，如果被激活，会修改构建处理 -->
  <profiles>
    <!--根据环境参数或命令行参数激活某个构建处理 -->
    <profile>
      <!--构建配置的唯一标识符。即用于命令行激活，也用于在继承时合并具有相同标识符的
profile。 -->
      <id/>
      <!--自动触发profile的条件逻辑。Activation是profile的开启钥匙。profile的力量
来自于它 能够在某些特定的环境中自动使用某些特定的值；这些环境通过activation元素指定。
activation元素并不是激活profile的唯一方式。 -->
      <activation>
        <!--profile默认是否激活的标志 -->
        <activeByDefault/>
        <!--当匹配的jdk被检测到，profile被激活。例如，1.4激活JDK1.4，1.4.0_2，
而!1.4激活所有版本不是以1.4开头的JDK。 -->
```

```
        <jdk/>
        <!--当匹配的操作系统属性被检测到，profile被激活。os元素可以定义一些操作系统相关的属性。 -->
        <os>
          <!--激活profile的操作系统的名字 -->
          <name>Windows XP</name>
          <!--激活profile的操作系统所属家族(如 'windows') -->
          <family>Windows</family>
          <!--激活profile的操作系统体系结构 -->
          <arch>x86</arch>
          <!--激活profile的操作系统版本 -->
          <version>5.1.2600</version>
        </os>
        <!--如果Maven检测到某一个属性（其值可以在POM中通过${名称}引用），其拥有对应的名称和值，Profile就会被激活。如果值 字段是空的，那么存在属性名称字段就会激活profile，否则按区分大小写方式匹配属性值字段 -->
        <property>
          <!--激活profile的属性的名称 -->
          <name>mavenVersion</name>
          <!--激活profile的属性的值 -->
          <value>2.0.3</value>
        </property>
        <!--提供一个文件名，通过检测该文件的存在或不存在来激活profile。missing检查文件是否存在，如果不存在则激活 profile。另一方面，exists则会检查文件是否存在，如果存在则激活profile。 -->
        <file>
          <!--如果指定的文件存在，则激活profile。 -->
          <exists>/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/
          </exists>
          <!--如果指定的文件不存在，则激活profile。 -->
          <missing>/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/
          </missing>
        </file>
      </activation>
      <!--构建项目所需要的信息。参见build元素 -->
      <build>
        <defaultGoal/>
        <resources>
          <resource>
            <targetPath/>
            <filtering/>
            <directory/>
            <includes/>
```

```
      <excludes/>
    </resource>
  </resources>
  <testResources>
    <testResource>
      <targetPath/>
      <filtering/>
      <directory/>
      <includes/>
      <excludes/>
    </testResource>
  </testResources>
  <directory/>
  <finalName/>
  <filters/>
  <pluginManagement>
    <plugins>
      <!--参见build/pluginManagement/plugins/plugin元素 -->
      <plugin>
        <groupId/>
        <artifactId/>
        <version/>
        <extensions/>
        <executions>
          <execution>
            <id/>
            <phase/>
            <goals/>
            <inherited/>
            <configuration/>
          </execution>
        </executions>
        <dependencies>
          <!--参见dependencies/dependency元素 -->
          <dependency>
            ......
          </dependency>
        </dependencies>
        <goals/>
        <inherited/>
        <configuration/>
      </plugin>
    </plugins>
  </pluginManagement>
  <plugins>
```

```
            <!--参见build/pluginManagement/plugins/plugin元素 -->
            <plugin>
              <groupId/>
              <artifactId/>
              <version/>
              <extensions/>
              <executions>
                <execution>
                  <id/>
                  <phase/>
                  <goals/>
                  <inherited/>
                  <configuration/>
                </execution>
              </executions>
              <dependencies>
                <!--参见dependencies/dependency元素 -->
                <dependency>
                  ......
                </dependency>
              </dependencies>
              <goals/>
              <inherited/>
              <configuration/>
            </plugin>
          </plugins>
        </build>
        <!--模块（有时称作子项目） 被构建成项目的一部分。列出的每个模块元素是指向该模块的
目录的相对路径 -->
        <modules/>
        <!--发现依赖和扩展的远程仓库列表。 -->
        <repositories>
          <!--参见repositories/repository元素 -->
          <repository>
            <releases>
              <enabled/>
              <updatePolicy/>
              <checksumPolicy/>
            </releases>
            <snapshots>
              <enabled/>
              <updatePolicy/>
              <checksumPolicy/>
            </snapshots>
            <id/>
```

```
            <name/>
            <url/>
            <layout/>
          </repository>
        </repositories>
        <!--发现插件的远程仓库列表，这些插件用于构建和报表 -->
        <pluginRepositories>
          <!--包含需要连接到远程插件仓库的信息.参见repositories/repository元素 -->
          <pluginRepository>
            <releases>
              <enabled/>
              <updatePolicy/>
              <checksumPolicy/>
            </releases>
            <snapshots>
              <enabled/>
              <updatePolicy/>
              <checksumPolicy/>
            </snapshots>
            <id/>
            <name/>
            <url/>
            <layout/>
          </pluginRepository>
        </pluginRepositories>
        <!--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。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。 -->
        <dependencies>
          <!--参见dependencies/dependency元素 -->
          <dependency>
            ......
          </dependency>
        </dependencies>
        <!--不赞成使用. 现在Maven忽略该元素. -->
        <reports/>
        <!--该元素包括使用报表插件产生报表的规范。当用户执行"mvn site"，这些报表就会运行。 在页面导航栏能看到所有报表的链接。参见reporting元素 -->
        <reporting>
          ......
        </reporting>
        <!--参见dependencyManagement元素 -->
        <dependencyManagement>
          <dependencies>
            <!--参见dependencies/dependency元素 -->
            <dependency>
```

```
            ......
          </dependency>
        </dependencies>
      </dependencyManagement>
      <!--参见distributionManagement元素 -->
      <distributionManagement>
        ......
      </distributionManagement>
      <!--参见properties元素 -->
      <properties/>
    </profile>
  </profiles>
  <!--模块（有时称作子项目） 被构建成项目的一部分。列出的每个模块元素是指向该模块的目录的相对路径 -->
  <modules/>
  <!--发现依赖和扩展的远程仓库列表。 -->
  <repositories>
    <!--包含需要连接到远程仓库的信息 -->
    <repository>
      <!--如何处理远程仓库里发布版本的下载 -->
      <releases>
        <!--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。 -->
        <enabled/>
        <!--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）。 -->
        <updatePolicy/>
        <!--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）。 -->
        <checksumPolicy/>
      </releases>
      <!-- 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，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。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。参见repositories/repository/releases元素 -->
      <snapshots>
        <enabled/>
        <updatePolicy/>
        <checksumPolicy/>
      </snapshots>
      <!--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 -->
      <id>banseon-repository-proxy</id>
      <!--远程仓库名称 -->
      <name>banseon-repository-proxy</name>
      <!--远程仓库URL，按protocol://hostname/path形式 -->
```

```
      <url>http://192.168.1.169:9999/repository/</url>
      <!-- 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）。 -->
      <layout>default</layout>
    </repository>
  </repositories>
  <!--发现插件的远程仓库列表，这些插件用于构建和报表 -->
  <pluginRepositories>
    <!--包含需要连接到远程插件仓库的信息.参见repositories/repository元素 -->
    <pluginRepository>
      ......
    </pluginRepository>
  </pluginRepositories>


  <!--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。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。 -->
  <dependencies>
    <dependency>
      <!--依赖的group ID -->
      <groupId>org.apache.maven</groupId>
      <!--依赖的artifact ID -->
      <artifactId>maven-artifact</artifactId>
      <!--依赖的版本号。 在Maven 2里，也可以配置成版本号的范围。 -->
      <version>3.8.1</version>
      <!-- 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。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，
          尽管这也有例外。一些类型的例子：jar，war，ejb-client和test-jar。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。 -->
      <type>jar</type>
      <!-- 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。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，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${java.home}。 -->
      <systemPath></systemPath>
      <!--当计算传递依赖时， 从依赖构件列表里，列出被排除的依赖构件集。即告诉maven你只依赖指定的项目，不依赖项目的依赖。此元素主要用于解决版本冲突问题 -->
      <exclusions>
        <exclusion>
          <artifactId>spring-core</artifactId>
          <groupId>org.springframework</groupId>
        </exclusion>
      </exclusions>
      <!--可选依赖，如果你在项目B中把C依赖声明为可选，你就需要在依赖于B的项目（例如项目A）中显式的引用对C的依赖。可选依赖阻断依赖的传递性。 -->
      <optional>true</optional>
    </dependency>
  </dependencies>
  <!--不赞成使用. 现在Maven忽略该元素. -->
  <reports></reports>
  <!--该元素描述使用报表插件产生报表的规范。当用户执行"mvn site"，这些报表就会运行。在页面导航栏能看到所有报表的链接。 -->
  <reporting>
    <!--true，则，网站不包括默认的报表。这包括"项目信息"菜单中的报表。 -->
    <excludeDefaults/>
    <!--所有产生的报表存放到哪里。默认值是${project.build.directory}/site。 -->
    <outputDirectory/>
    <!--使用的报表插件和他们的配置。 -->
    <plugins>
      <!--plugin元素包含描述报表插件需要的信息 -->
      <plugin>
        <!--报表插件在仓库里的group ID -->
        <groupId/>
        <!--报表插件在仓库里的artifact ID -->
        <artifactId/>
        <!--被使用的报表插件的版本（或版本范围） -->
        <version/>
        <!--任何配置是否被传播到子项目 -->
        <inherited/>
        <!--报表插件的配置 -->
        <configuration/>
        <!--一组报表的多重规范，每个规范可能有不同的配置。一个规范（报表集）对应一个执行目标 。例如，有1，2，3，4，5，6，7，8，9个报表。1，2，5构成A报表集，对应一个执行目标。2，5，8构成B报表集，对应另一个执行目标 -->
        <reportSets>
          <!--表示报表的一个集合，以及产生该集合的配置 -->
```

```
            <reportSet>
              <!--报表集合的唯一标识符，POM继承时用到 -->
              <id/>
              <!--产生报表集合时，被使用的报表的配置 -->
              <configuration/>
              <!--配置是否被继承到子POMs -->
              <inherited/>
              <!--这个集合里使用到哪些报表 -->
              <reports/>
            </reportSet>
          </reportSets>
        </plugin>
      </plugins>
    </reporting>
    <!-- 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），如果group ID和artifact ID以外的一些信息没有描述，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。 -->
    <dependencyManagement>
      <dependencies>
        <!--参见dependencies/dependency元素 -->
        <dependency>
          ......
        </dependency>
      </dependencies>
    </dependencyManagement>
    <!--项目分发信息，在执行mvn deploy后表示要发布的位置。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。 -->
    <distributionManagement>
      <!--部署项目产生的构件到远程仓库需要的信息 -->
      <repository>
        <!--是分配给快照一个唯一的版本号（由时间戳和构建流水号）？还是每次都使用相同的版本号？参见repositories/repository元素 -->
        <uniqueVersion/>
        <id>banseon-maven2</id>
        <name>banseon maven2</name>
        <url>file://${basedir}/target/deploy</url>
        <layout/>
      </repository>
      <!--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，参见distributionManagement/repository元素 -->
      <snapshotRepository>
        <uniqueVersion/>
        <id>banseon-maven2</id>
        <name>Banseon-maven2 Snapshot Repository</name>
```

```
      <url>scp://svn.baidu.com/banseon:/usr/local/maven-snapshot</url>
      <layout/>
    </snapshotRepository>
    <!--部署项目的网站需要的信息 -->
    <site>
      <!--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 -->
      <id>banseon-site</id>
      <!--部署位置的名称 -->
      <name>business api website</name>
      <!--部署位置的URL，按protocol://hostname/path形式 -->
      <url>
        scp://svn.baidu.com/banseon:/var/www/localhost/banseon-web
      </url>
    </site>
    <!--项目下载页面的URL。如果没有该元素，用户应该参考主页。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）。 -->
    <downloadUrl/>
    <!--如果构件有了新的group ID和artifact ID（构件移到了新的位置），这里列出构件的重定位信息。 -->
    <relocation>
      <!--构件新的group ID -->
      <groupId/>
      <!--构件新的artifact ID -->
      <artifactId/>
      <!--构件新的版本号 -->
      <version/>
      <!--显示给用户的，关于移动的额外信息，例如原因。 -->
      <message/>
    </relocation>
    <!-- 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，因为这是工具自动更新的。有效的值有：none（默认），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）。 -->
    <status/>
  </distributionManagement>
  <!--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>。 -->
  <properties/>
</project>
```

构建生命周期

- 在一个生命周期中，运行某个阶段的时候，它之前的所有阶段都会被运行，也就是说，如果执行 mvn clean 将运行pre-clean和clean。
- 插件目标（pluginId:goalId）：一个插件是一个或多个目标的集合，它可以作为单独的目标运行,或者作为一个大的构建的一部分和其它目标一起运行。一个目标是一个明确的任务，一个目标是Maven中的一个工作单元。插件目标可能被绑定到多个阶段或者无绑定。不绑定到任何构建阶段的目标可以在构建生命周期之外通过直接调用执行。这些目标的执行顺序取决于调用目标和构建阶段的顺序。

```
# 这里的 clean 阶段将会被首先执行，然后 dependency:copy-dependencies 目标会被执行，最终 package 阶段被执行。
mvn clean      dependency:copy-dependencies package
```

- 在生命周期的任何阶段定义目标可以修改这部分的操作行为。

Maven标准生命周期

- clean：清理
  - pre-clean
  - clean
  - post-cliean
- default（或build）：部署
  - validate（校验）：校验项目是否正确并且所有必要的信息可以完成项目的构建过程。
  - initialize（初始化）：初始化构建状态，比如设置属性值。
  - generate-sources（生成源代码）：生成包含在编译阶段中的任何源代码。
  - process-sources（处理源代码）：处理源代码，比如说，过滤任意值。
  - generate-resources（生成资源文件）：生成将会包含在项目包中的资源文件。
  - process-resources （处理资源文件）：复制和处理资源到目标目录，为打包阶段最好准备。
  - compile（编译）：编译项目的源代码。
  - process-classes（处理类文件）：处理编译生成的文件，比如说对Java class文件做字节码改善优化。
  - generate-test-sources（生成测试源代码）：生成包含在编译阶段中的任何测试源代码。
  - process-test-sources（处理测试源代码）：处理测试源代码，比如说，过滤任意值。
  - generate-test-resources（生成测试资源文件）：为测试创建资源文件。
  - process-test-resources（处理测试资源文件）：复制和处理测试资源到目标目录。
  - test-compile（编译测试源码）：编译测试源代码到测试目标目录.
  - process-test-classes（处理测试类文件）：处理测试源码编译生成的文件。
  - test（测试）：使用合适的单元测试框架运行测试（Juint是其中之一）。
  - prepare-package（准备打包）：在实际打包之前，执行任何的必要的操作为打包做准备。
  - package（打包）：将编译后的代码打包成可分发格式的文件，比如JAR、WAR或者EAR文件。
  - pre-integration-test（集成测试前）：在执行集成测试前进行必要的动作。比如说，搭建需要的环境。
  - integration-test（集成测试）：处理和部署项目到可以运行集成测试环境中。
  - post-integration-test（集成测试后）：在执行集成测试完成后进行必要的动作。比如说，清理集成测试环境。
  - verify （验证）：运行任意的检查来验证项目包有效且达到质量标准。
  - install（安装）：安装项目包到本地仓库，这样项目包可以用作其他本地项目的依赖。
  - deploy（部署）：将最终的项目包复制到远程仓库中与其他开发者和项目共享。
- site：站点文档创建
  - pre-site
  - site
  - post-site
  - site-deploy

构建配置文件

构建配置文件是一系列的配置项的值，可以用来设置或者覆盖 Maven 构建默认值。

使用构建配置文件，你可以为不同的环境，比如说生产环境（Production）和开发（Development）环境，定制构建方式。

配置文件在 pom.xml 文件中使用 activeProfiles 或者 profiles 元素指定，并且可以通过各种方式触发。配置文件在构建时修改 POM，并且用来给参数设定不同的目标环境（比如说，开发（Development）、测试（Testing）和生产环境（Production）中数据库服务器的地址）。

- 构建配置文件的类型

类型	在哪定义
项目级（Per Project）	定义在项目的POM文件中pom.xml中
用户级（Per User）	定义在Maven的设置xml文件中（%USER_HOME%/.m2/settings.xml）
全局（Global）	定义在Maven全局的设置xml文件中（%M2_HOME%/conf/setting.xml）

- 激活方式
  1. 命令行

```
# -P指定profile的id
mvn test -Ptest
```

```
<!--  pom.xml的profile -->
<profile>
  <!--  命令行启动时通过指定id执行指定的profile -->
  <id>test</id>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.8</version>
        <executions>
          <execution>
```

```
            <phase>test</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <tasks>
                <echo>Using env.test.properties</echo>
                <copy file="src/main/resources/env.test.properties"

 tofile="${project.build.outputDirectory}/env.properties"
overwrite="true"/>
              </tasks>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</profile>
```

2. Maven设置

   通过配置setting.xml的activeProfile则不需要在执行mvn命令时通过-P指定profile

```
<!-- setting.xml -->
<activeProfiles>
  <activeProfile>test</activeProfile>
</activeProfiles>
```

3. 环境变量

```
# 通过-D传递环境变量
mvn test -Denv=test
```

```
<profile>
  <id>test</id>
  <activation>
    <property>
      <!-- 设置环境变量为激活条件 -->
      <name>env</name>
      <value>test</value>
```

```
      </property>
    </activation>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-antrun-plugin</artifactId>
          <version>1.8</version>
          <executions>
            <execution>
              <phase>test</phase>
              <goals>
                <goal>run</goal>
              </goals>
              <configuration>
                <tasks>
                  <echo>Using env.test.properties</echo>
                  <copy file="src/main/resources/env.test.properties"

 tofile="${project.build.outputDirectory}/env.properties"
overwrite="true"/>
                </tasks>
              </configuration>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
```

4. 操作系统

```
<profile>
  <id>test</id>
  <activation>
    <os>
      <name>Windows XP</name>
      <family>Windows</family>
      <arch>x86</arch>
      <version>5.1.2600</version>
    </os>
  </activation>
</profile>
```

5. 文件的存在或缺失

```
<profile>
  <id>test</id>
  <activation>
    <file>
      <!--  缺失missing中的文件时激活    -->
      <missing>target/generated-
sources/axistools/wsdl2java/com/companyname/group</missing>
    </file>
  </activation>
</profile>
```

仓库

顺序：本地→中央→远程

- 本地（local)

```
<!--  setting.xml配置本地仓库位置   -->
<localRepository>/home/data/maven-repository</localRepository>
```

- 中央（central）

```
<mirrors>
  <!--  setting.xml 设置阿里云镜像   -->
  <mirror>
    <id>alimaven</id>
    <name>aliyun maven</name>
```

```
        <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
        <mirrorOf>central</mirrorOf>
      </mirror>
      <!--  setting.xml 设置私服仓库   -->
      <mirror>
          <id>ln-bss</id>
          <mirrorOf>*</mirrorOf>
          <url>http://127.0.0.1:8088/repository/maven-public/</url>
        </mirror>
    </mirrors>
```

- 远程（remote）

```
    <repositories>
        <!--  pom 文件中配置远程仓库 -->
        <repository>
          <id>ln-bss</id>
          <name>ln-bss</name>
          <url>http://132.194.41.68:8088/repository/maven-public/</url>
        </repository>
      </repositories>
```

插件

生命周期的每个阶段都是由插件实现的，插件提供了一个目标的集合

```
mvn [plugin-name]:[goal-name]
```

- 插件类型

类型	描述
Build plugins	在构建时执行，并在 pom.xml 的 元素中配置。
Reporting plugins	在网站生成过程中执行，并在 pom.xml 的 元素中配置。

- 常见插件列表

插件	描述
clean	构建之后清理目标文件。删除目标目录。
compiler	编译 Java 源文件。

surefile	运行 JUnit 单元测试。创建测试报告。
jar	从当前工程中构建 JAR 文件。
war	从当前工程中构建 WAR 文件。
javadoc	为工程生成 Javadoc。
antrun	从构建过程的任意一个阶段中运行一个 ant 任务的集合。

- 实例

```
<build>
  <plugins>
    <!-- 使用maven-antrun-plugin在clean阶段输出指定内容 -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-antrun-plugin</artifactId>
      <version>1.1</version>
      <executions>
        <execution>
          <!--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 -->
          <id>id.clean</id>
          <!--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 -->
          <!--这里指定在clean阶段 -->
          <phase>clean</phase>
          <goals>
            <!-- 目标为maven-antrun-plugin的run阶段 -->
            <goal>run</goal>
          </goals>
          <configuration>
            <tasks>
              <!-- maven-antrun-plugin的run阶段绑定了echo任务，会在clean阶段打印clean phase -->
              <echo>clean phase</echo>
            </tasks>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Maven构建Java项目

使用maven-archetype-quickstart构建一个Java项目

archetype 也就是原型，是一个 Maven 插件，准确说是一个项目模板，它的任务是根据模板创建一个项目结构。

```
# 不传入参数将要求选择所需的原型、原型版本、项目细节
mvn archetype:generate "-DgroupId=com.companyname.bank" "-
DartifactId=consumerBanking" "-DarchetypeArtifactId=maven-archetype-
quickstart" "-DinteractiveMode=false"
```

- **-DgourpId**: 组织名，公司网址的反写 + 项目名称
- **-DartifactId**: 项目名-模块名
- **-DarchetypeArtifactId**: 指定 ArchetypeId，maven-archetype-quickstart，创建一个简单的 Java 应用
- **-DinteractiveMode**: 是否使用交互模式

文件夹结构	描述
consumerBanking	包含 src 文件夹和 pom.xml
src/main/java contains	java 代码文件在包结构下（com/companyName/bank）。
src/main/test contains	测试代码文件在包结构下（com/companyName/bank）。
src/main/resources	包含了 图片 / 属性 文件（在上面的例子中，我们需要手动创建这个结构）。

Maven项目文档

Maven 使用一个名为 Doxia的文档处理引擎来创建文档，它能将多种格式的源码读取成一种通用的文档模型。要为你的项目撰写文档，你可以将内容写成下面几种常用的，可被 Doxia 转化的格式。

格式名	描述	参考
Apt	纯文本文档格式	http://maven.apache.org/doxia/references/apt-format.html
Xdoc	Maven 1.x 的一种文档格式	http://jakarta.apache.org/site/jakarta-site2.html
FML	FAQ 文档适用	http://maven.apache.org/doxia/references/fml-format.html

XHTML	可扩展的 HTML 文档	http://en.wikipedia.org/wiki/XHTML

maven-site-plugin 插件 ， pom文件中做相应配置以后可以生成以下信息

- 关于（about）： 项目描述
- 持续集成（Continuous Integeration）： 项目持续化集成服务器信息；
- 依赖（Dependencies）： 项目依赖信息，包括传递性依赖、依赖图、依赖许可证以及依赖文件的大小、所包含的类的数目；
- 依赖收敛（Dependency Convergence）： 针对多个模块项目生成，提供一些依赖健康状况分析；
- 依赖管理（Dependency Management）： 基于项目的依赖管理生成的报告；
- 问题追踪（Issue Tracking）： 项目问题追踪系统信息；
- 邮件列表（Mailing Lists）： 项目的邮件列表信息；
- 插件管理（Plugin Management）： 项目所有项目插件的列表；
- 项目许可证（Project License）： 项目许可证信息；
- 项目概述（Project Summary）： 项目概述包括坐标、名称、描述等；
- 项目团队（Project Team）： 项目团队信息；
- 源码仓库（Source Repository）： 项目的源码仓库信息；

还可以使用其他插件生成其他报告

JavaDocs、Source Xref、CheckStyle、ChangeLog、Cobertura

还可以自定义外观、国际化

```
mvn site
```

Maven快照(SNAPSHOT)

快照是一种特殊的版本，指定了某个当前的开发进度的副本。不同于常规的版本，Maven 每次构建都会在远程仓库中检查新的快照。

对于版本，如果 Maven 以前下载过指定的版本文件，比如说 1.0，Maven 将不会再从仓库下载新的可用的 1.0 文件。若要下载更新的代码，版本需要升到1.1。

快照的情况下，每次 构建项目时，Maven 将自动获取最新的快照。

Maven自动化构建

自动化构建定义了这样一种场景: 在一个项目成功构建完成后，其相关的依赖工程即开始构建，这样可以保证其依赖项目的稳定。

两种方法:

- 使用Maven

```
<plugin>
  <artifactId>maven-invoker-plugin</artifactId>
  <version>1.6</version>
  <configuration>
    <debug>true</debug>
    <pomIncludes>
      <!--  app-web-ui和app-desktop-ui依赖于当前项目，构建完当前项目后，会以此构建app-web-ui和app-desktop-ui  -->
      <pomInclude>app-web-ui/pom.xml</pomInclude>
      <pomInclude>app-desktop-ui/pom.xml</pomInclude>
    </pomIncludes>
  </configuration>
  <executions>
    <execution>
      <id>build</id>
      <goals>
        <goal>run</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

- 使用持续集成（CI）服务器

Maven依赖管理

- 可传递性依赖发现

功能	功能描述
依赖调节	决定当多个手动创建的版本同时出现时，哪个依赖版本将会被使用。 如果两个依赖版本在依赖树里的深度是一样的时候，第一个被声明的依赖将会被使用。
依赖管理	直接的指定手动创建的某个版本被使用。例如当一个工程 A 在自己的依赖管理模块包含工程 B，即 B 依赖于 A， 那么 A 即可指定在 B 被引用时所使用的版本。
依赖范	包含在构建过程每个阶段的依赖。

围	
依赖排除	任何可传递的依赖都可以通过 "exclusion" 元素被排除在外。举例说明，A 依赖 B， B 依赖 C，因此 A 可以标记 C 为 "被排除的"。
依赖可选	任何可传递的依赖可以被标记为可选的，通过使用 "optional" 元素。例如：A 依赖 B，B 依赖 C。因此，B 可以标记 C 为可选的， 这样 A 就可以不再使用 C。

- 依赖范围

范围	描述
编译阶段	该范围表明相关依赖是只在项目的类路径下有效。默认取值。
供应阶段	该范围表明相关依赖是由运行时的 JDK 或者 网络服务器提供的。
运行阶段	该范围表明相关依赖在编译阶段不是必须的，但是在执行阶段是必须的。
测试阶段	该范围表明相关依赖只在测试编译阶段和执行阶段。
系统阶段	该范围表明你需要提供一个系统路径。
导入阶段	该范围只在依赖是一个 pom 里定义的依赖时使用。同时，当前项目的POM 文件的 部分定义的依赖关系可以取代某特定的 POM。

```
<!--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>
```

Maven自动化部署

元素节点	描述

SCM	配置代码管理的路径，Maven 将从该路径下将代码取下来。
repository	构建的 WAR 或 EAR 或JAR 文件的位置，或者其他源码构建成功后生成的构件的存储位置。
Plugin	配置 maven-release-plugin 插件来实现自动部署过程。

```
<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">
   <modelVersion>4.0.0</modelVersion>
   <groupId>bus-core-api</groupId>
   <artifactId>bus-core-api</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>jar</packaging>
   <scm>
      <url>http://www.svn.com</url>

 <connection>scm:svn:http://localhost:8080/svn/jrepo/trunk/Framework</conne
ction>
      <developerConnection>scm:svn:${username}/${password}@localhost:8080:
      common_core_api:1101:code</developerConnection>
   </scm>
   <distributionManagement>
      <repository>
         <id>Core-API-Java-Release</id>
         <name>Release repository</name>
         <url>http://localhost:8081/nexus/content/repositories/Core-Api-
Release</url>
      </repository>
   </distributionManagement>
   <build>
      <plugins>
         <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-release-plugin</artifactId>
            <version>2.0-beta-9</version>
            <configuration>
               <useReleaseProfile>false</useReleaseProfile>
               <goals>deploy</goals>
               <scmCommentPrefix>[bus-core-api-release-checkin]-
</scmCommentPrefix>
            </configuration>
         </plugin>
```

```
        </plugins>
    </build>
</project>
```

Maven Release 插件

```
# 清理工作空间，保证最新的发布进程成功进行。
mvn release:clean
```

```
# 在上次发布过程不成功的情况下，回滚修改的工作空间代码和配置保证发布过程成功进行。
mvn release:rollback
```

```
# 执行多种操作：
#   检查本地是否存在还未提交的修改
#   确保没有快照的依赖
#   改变应用程序的版本信息用以发布
#   更新 POM 文件到 SVN
#   运行测试用例
#   提交修改后的 POM 文件
#   为代码在 SVN 上做标记
#   增加版本号和附加快照以备将来发布
#   提交修改后的 POM 文件到 SVN
mvn release:prepare
```

```
# 将代码切换到之前做标记的地方，运行 Maven 部署目标来部署 WAR 文件或者构建相应的结构到仓库里。
mvn release:perform
```