第 13 章

版本管理

本章内容

- ❑ 何为版本管理
- ❑ Maven 的版本号定义约定
- ❑ 主干、标签与分支
- ❑ 自动化版本发布
- ❑ 自动化创建分支
- ❑ GPG 签名
- ❑ 小结

一个健康的项目通常有一个长期、合理的版本演变过程。例如 JUnit 有 3.7、3.8、3.8.1、3.8.2、4.0、4.1 等版本。Maven 本身的版本也比较多，如最早的 Maven 1；目前使用最广泛的 Maven 2 有 2.0.9、2.0.10、2.1.0、2.2.0、2.2.1 等各种版本；而最新的 Maven 3 则拥有 3.0-alpha-1、3.0-alpha-2、3.0-alpha-7、3.0-beta-1 等版本。除了这些对外发布的版本之外，6.5 节还介绍了 Maven 特有的快照版本的概念。这些版本中的每个数字代表了什么？alpha、beta 是什么意思？快照版和发布版的区别是什么？我们应该如何科学地管理自己的项目版本？本章将会详细解答这些问题。

阅读本章的时候还需要分清版本管理（Version Management）和版本控制（Version Control）的区别。版本管理是指项目整体版本的演变过程管理，如从 1.0-SNAPSHOT 到 1.0，再到 1.1-SNAPSHOT。版本控制是指借助版本控制工具（如 Subversion）追踪代码的每一个变更。本章重点讲述的是版本管理，但是读者将会看到，版本管理通常也会涉及一些版本控制系统的操作及概念。请在阅读的时候特别留意这两者的关系和区别。

13.1 何为版本管理

6.5 节谈到，为了方便团队的合作，在项目开发的过程中，大家都应该使用快照版本，Maven 能够很智能地处理这种特殊的版本，解析项目各个模块最新的“快照”。快照版本机制促进团队内部的交流，但是当项目需要对外发布时，我们显然需要提供非常稳定的版本，使用该版本应当永远只能够定位到唯一的构件，而不是像快照版本那样，定位的构件随时可能发生变化。对应地，我们称这类稳定的版本为发布版。项目发布了一个版本之后，就进入下一个开发阶段，项目也就自然转换到新的快照版本中。

版本管理关心的问题之一就是这种快照版和发布版之间的转换。项目经过了一段时间的 1.0-SNAPSHOT 的开发之后，在某个时刻发布了 1.0 正式版，然后项目又进入了 1.1-SNAPSHOT 的开发，这个版本可能添加了一些有趣的特性，然后在某个时刻发布 1.1 正式版。项目接着进入 1.2-SNAPSHOT 的开发。由于快照对应了项目的开发过程，因此往往对应了很长的时间，而正式版本对应了项目的发布，因此仅仅代表某个时刻项目的状态，如图 13-1 所示。

图 13-1 快照版和发布版之间的转换

理想的发布版本应当对应了项目某个时刻比较稳定的状态，这包括源代码的状态以及构建的状态，因此这个时候项目的构建应当满足以下的条件：

- **所有自动化测试应当全部通过**。毫无疑问，失败的测试代表了需要修复的问题，因此发布版本之前应该确保所有测试都能得以正确执行。
- **项目没有配置任何快照版本的依赖**。快照版本的依赖意味着不同时间的构建可能会

引入不同内容的依赖，这显然不能保证多次构建能够生成同样的结果。

- **项目没有配置任何快照版本的插件**。快照版本的插件配置可能会在不同时间引入不容内容的 Maven 插件，从而影响 Maven 的行为，破坏构建的稳定性。
- **项目所包含的代码已经全部提交到版本控制系统中**。项目已经发布了，可源代码却不在版本控制系统中，甚至丢失了。这意味着项目丢失了某个时刻的状态，因此这种情况必须避免，版本发布的时候必须确保所有的源代码都已经提交了。

只有上述条件都满足之后，才可以将快照版本更新为发布版本，例如将 1.0-SNAPSHOT 更新为 1.0，然后生成版本为 1.0 的项目构件。

不过这里还缺少一步关键的版本控制操作。如果你了解任何一种版本控制工具，如 Subversion，那就应该能想到项目发布与标签（Tag）的关系。版本控制系统记录代码的每一个变化，通常这些变化都被维护在主干（Trunk）中，但是当项目发布的时候，开发人员就应该使用标签记录这一特殊时刻项目的状态。以 Subversion 为例，日常的变更维护在主干中，包含各种源码版本 r1、r2、…、r284、…。要找到某个时刻的项目状态会比较麻烦，而使用标签就可以明确地将某个源码版本（也就是项目状态）从主干中标记出来，放到单独的位置，这样在之后的任何时刻，我们都能够快速地得到发布版本的源代码，从而能够比较各个版本的差异，甚至重新构建一个同样版本的构件。

因此，将项目的快照版本更新至发布版本之后，应当再执行一次 Maven 构建，以确保项目状态是健康的。然后将这一变更提交到版本控制系统的主干中。接着再为当前主干的状态打上标签。以 Subversion 为例，这几个步骤对应的命令如下：

```
$mvn clean install
$svn commit pom.xml -m "prepare to release 1.0"
$svn copy -m "tag release 1.0" \
https://svn.juvenxu.com/project/trunk \
https://svn.juvenxu.com/project/tags/1.0
```

至此，一个版本发布的过程完成了。接下来要做的就是更新发布版本至新的快照版本，如从 1.0 到 1.1-SNAPSHOT。

13.2 Maven 的版本号定义约定

到目前为止，读者应该已经清楚了解了快照版和发布版的区别。现在再深入看一下 1.0、1.1、1.2.1、3.0-beta 这样的版本号后面又遵循了怎样的约定。了解了这样的约定之后，就可以正确地为自己的产品或者项目定义版本号，而你的用户也能了解到隐藏在版本号中的信息。

看一个实际的例子，这里有一个版本：

1.3.4-beta-2

这往往表示了该项目或产品的第一个重大版本的第三个次要版本的第四次增量版本的

beta-2 里程碑。很拗口？那一个个分开解释："1" 表示了该版本是第一个重大版本；"3" 表示这是基于重大版本的第三个次要版本；"4" 表示该次要版本的第四个增量；最后的 "beta-2" 表示该增量的某一个里程碑。

也就是说，Maven 的版本号定义约定是这样的：

<主版本>.<次版本>.<增量版本>-<里程碑版本>

主版本和次版本之间，以及次版本和增量版本之间用点号分隔，里程碑版本之前用连字号分隔。下面解释其中每一个部分的意义：

- **主版本**：表示了项目的重大架构变更。例如，Maven 2 和 Maven 1 相去甚远；Struts 1 和 Struts 2 采用了不同的架构；JUnit 4 较 JUnit 3 增加了标注支持。
- **次版本**：表示较大范围的功能增加和变化，及 Bug 修复。例如 Nexus 1.5 较 1.4 添加了 LDAP 的支持，并修复了很多 Bug，但从总体架构来说，没有什么变化。
- **增量版本**：一般表示重大 Bug 的修复，例如项目发布了 1.4.0 版本之后，发现了一个影响功能的重大 Bug，则应该快速发布一个修复了 Bug 的 1.4.1 版本。
- **里程碑版本**：顾名思义，这往往指某一个版本的里程碑。例如，Maven 3 已经发布了很多里程碑版本，如 3.0-alpha-1、3.0-alpha-2、3.0-beta-1 等。这样的版本与正式的 3.0 相比，往往表示不是非常稳定，还需要很多测试。

需要注意的是，不是每个版本号都必须拥有这四个部分。一般来说，主版本和次版本都会声明，但增量版本和里程碑就不一定了。例如，像 3.8 这样的版本没有增量和里程碑，2.0-beta-1 没有增量。但我们不会看到有人省略次版本，简单地给出主版本显然是不够的。

当用户在声明依赖或插件未声明版本时，Maven 就会根据上述的版本号约定自动解析最新版本。这个时候就需要对版本号进行排序。对于主版本、次版本和增量版本来说，比较是基于数字的，因此 1.5 > 1.4 > 1.3.11 > 1.3.9。而对于里程碑版本，Maven 则只进行简单的字符串比较，因此会得到 1.2-beta-3 > 1.2-beta-11 的结果。这一点需要留意。

13.3 主干、标签与分支

使用版本控制工具时我们都会遇到主干（trunk）、标签（tag）和 branch（分支）的概念。13.1 节已经涉及了主干与标签。这里再详细将这几个概念阐述一下，因为理解它们是理解 Maven 版本管理的基础。

- **主干**：项目开发代码的主体，是从项目开始直到当前都处于活动的状态。从这里可以获得项目最新的源代码以及几乎所有的变更历史。
- **分支**：从主干的某个点分离出来的代码拷贝，通常可以在不影响主干的前提下在这里进行重大 Bug 的修复，或者做一些实验性质的开发。如果分支达到了预期的目的，通常发生在这里的变更会被合并（merge）到主干中。
- **标签**：用来标识主干或者分支的某个点的状态，以代表项目的某个稳定状态，这通

常就是版本发布时的状态。

本书采用 Subversion 作为版本控制系统，如果对上述概念不清晰，请参考开放的《Subversion 与版本控制》（http://svnbook.red-bean.com/）一书。

使用 Maven 管理项目版本的时候，也涉及了很多的版本控制系统操作。下面就以一个实际的例子来介绍这些操作是如何执行的。

图 13-2 下方最长的箭头表示项目的主干，项目最初的版本是 1.0.0-SNAPSHOT，经过一段时间的开发后，1.0.0 版本发布，这个时候就需要打一个标签，图中用一个长条表示。然后项目进入 1.1.0-SNAPSHOT 状态，大量的开发工作都完成在主干中，添加了一些新特性并修复了很多 Bug 之后，项目 1.1.0 发布，同样，这时候需要打另一个标签。发布过后，项目进入 1.2.0-SNAPSHOT 阶段，可这个时候用户报告 1.1.0 版本有一个重大的 Bug，需要尽快修复，我们不能在主干中修 Bug，因为主干有太多的变化，无法在短时间内测试完毕并发布，我们也不能停止 1.2.0-SNAPSHOT 的开发，因此这时候可以基于 1.1.0 创建一个 1.1.1-SNAPSHOT 的分支，在这里进行 Bug 修复，然后为用户发布一个 1.1.1 增量版本，同时打上标签。当然，还不能忘了把 Bug 修复涉及的变更合并到 1.2.0-SNAPSHOT 的主干中。主干在开发一段时间之后，发布 1.2.0 版本，然后进入到新版本 1.3.0-SNAPSHOT 的开发过程中。

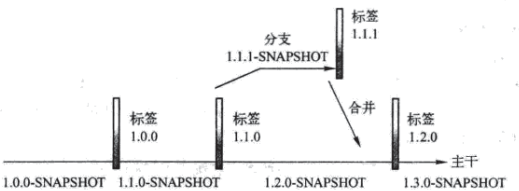

图 13-2 主干、标签和分支与项目版本的关系

图 13-2 展示的是一个典型的项目版本变化过程，这里涉及了快照版与发布版之间的切换、Maven 版本号约定的应用，以及版本控制系统主干、标签和分支的使用。这其实也是一个不成文的行业标准，理解这个过程之后，不仅能够更方便地学习开源项目，也能对项目的版本管理更加标准和清晰。

13.4 自动化版本发布

本章前几节已经详细介绍了版本发布时所需要完成的工作，读者如果愿意，则完全可以手动地执行这些操作，检查是否有未提交代码、是否有快照依赖、更新快照版至发布版、执行 Maven 构建以及为源代码打标签等。事实上，如果对这一过程不是很熟悉，那么还是应该一步一步地操作一遍，以得到最直观的感受。

当熟悉了版本发布流程之后，就会希望借助工具将这一流程自动化。Maven Release Plugin 就提供了这样的功能，只要提供一些必要的信息，它就能帮我们完成上述所有版本发布所涉及的操作。下面介绍如何使用 Maven Release Plugin 发布项目版本。

Maven Release Plugin 主要有三个目标，它们分别为：

- **release:prepare** 准备版本发布，依次执行下列操作：
  - 检查项目是否有未提交的代码。
  - 检查项目是否有快照版本依赖。
  - 根据用户的输入将快照版本升级为发布版。
  - 将 POM 中的 SCM 信息更新为标签地址。
  - 基于修改后的 POM 执行 Maven 构建。
  - 提交 POM 变更。
  - 基于用户输入为代码打标签。
  - 将代码从发布版升级为新的快照版。
  - 提交 POM 变更。
- **release:rollback** 回退 release:prepare 所执行的操作。将 POM 回退至 release:prepare 之前的状态，并提交。需要注意的是，该步骤不会删除 release:prepare 生成的标签，因此用户需要手动删除。
- **release:perform** 执行版本发布。签出 release:prepare 生成的标签中的源代码，并在此基础上执行 mvn deploy 命令打包并部署构件至仓库。

要为项目发布版本，首先需要为其添加正确的版本控制系统信息，这是因为 Maven Release Plugin 需要知道版本控制系统的主干、标签等地址信息后才能执行相关的操作。一般配置项目的 SCM 信息如代码清单 13-1 所示。

代码清单 13-1 为版本发布配置 SCM 信息

```
<project>
...
  <scm>
    <connection>scm:svn:http://192.168.1.103/app/trunk</connection>
    <developerConnection>scm:svn:https://192.168.1.103/app/trunk</developerCon-
nection>
    <url>http://192.168.1.103/account/trunk</url>
  </scm>...
</project>
```

代码清单 13-1 中的 connection 元素表示一个只读的 scm 地址，而 developerConnection 元素表示可写的 scm 地址，url 则表示可以在浏览器中访问的 scm 地址。为了能让 Maven 识别，connection 和 developerConnection 必须以 scm 开头，冒号之后的部分表示版本控制工具类型（这里是 svn），Maven 还支持 cvs、git 等。接下来才是实际的 scm 地址，该例中的 connection 使用了 http 协议，而 developerConnection 则由于涉及写操作，使用 https 协议进行了保护。

该配置只告诉 Maven 当前代码的位置（主干），而版本发布还要涉及标签操作。因此，

还需要配置 Maven Release Plugin 告诉其标签的基础目录，如代码清单 13-2 所示。

代码清单 13-2 配置 maven-release-plugin 提供标签基础目录

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-release-plugin</artifactId>
    <version>2.0</version>
    <configuration>
      <tagBase>https://192.168.1.103/app/tags/</tagBase>
    </configuration>
</plugin>
```

在执行 release:prepare 之前还有两个注意点：第一，系统必须要提供 svn 命令行工具，Maven 需要 svn 命令行工具执行相关操作，而无法使用图形化的工具，如 TortoiseSVN；第二，POM 必须配置了可用的部署仓库，因为 release:perform 会执行 deploy 操作将构件发布到仓库中。关于如何配置部署仓库可参考 9.6.1 节。

一切就绪之后，在项目根目录下运行如下命令：

```
$mvn release:prepare
```

Maven Release Plugin 开始准备发布版本，如果它检测到项目有未提交的代码，或者项目有快照版的依赖，则会提示出错。如果一切都没问题，则会提示用户输入想要发布的版本号、标签的名称以及新的快照版本号。例如：

```
What is the release version for "App"? (com.juvenxu.mvnbook:app) 1.0.0: :
What is SCM release tag or label for "App"? (com.juvenxu.mvnbook:app) app-1.0.0: :
What is the new development version for "App"? (com.juvenxu.mvnbook:app) 1.0.1-
SNAPSHOT: :1.1.0-SNAPSHOT
```

如果项目的 artifactId 为 app，发布前的版本为 1.0.0-SNAPSHOT，则 Maven Release Plugin 会提示使用发布版本号 1.0.0，使用标签名称 app-1.0.0，新的开发版本为 1.0.1-SNAPSHOT。如果这些模式值正是你想要的，直接按 Enter 键即可，否则就输入想要的值再按 Enter 键，如上例中为新的开发版本输入了值 1.1.0-SNAPSHOT。

基于这些信息，Maven Release Plugin 会将版本从 1.0.0-SNAPSHOT 更新为 1.0.0，并更新 SCM 地址 http://192.168.1.103/app/trunk 至 http://192.168.1.103/app/tags/app-1.0.0。在此基础上运行一次 Maven 构建以防止意外的错误出现，然后将这两个变化提交，并为该版本打上标签，标签地址是 http://192.168.1.103/app/tags/app-1.0.0。即 tagBase 路径加上标签名称。之后，Maven Release Plugin 会将 POM 中的版本信息从 1.0.0 升级到 1.1.0-SNAPSHOT 并提交。

至此，release:prepare 的工作完成。如果这时你发现了一些问题，例如将标签名称配置错了，则可以使用 release:rollback 命令回退发布，Maven Release Plugin 会将 POM 的配置回退到 release:prepare 之前的状态。但需要注意的是，版本控制系统中的标签并不会被删除，也就是说，用户需要手动执行版本控制系统命令删除该标签。

在多模块项目中执行 release:prepare 的时候，默认 maven-release-plugin 会提示用户设定

每个模块发布版本号及新的开发版本号。例如，如果在 account-parent 模块中配置正确的 scm 信息之后进行项目发布，就会看到如下的输出：

```
What is the release version for "Account Parent"? (com.juvenxu.mvnbook.account:
account-parent) 1.0.0: :
What is the release version for "Account Email"? (com.juvenxu.mvnbook.account:ac-
count-email) 1.0.0: :
What is the release version for "Account Persist"? (com.juvenxu.mvnbook.account:
account-persist) 1.0.0: :
What is the release version for "Account Captcha"? (com.juvenxu.mvnbook.account:
account-captcha) 1.0.0: :
What is the release version for "Account Service"? (com.juvenxu.mvnbook.account:
account-service) 1.0.0: :
What is the release version for "Account Web"? (com.juvenxu.mvnbook.account:ac-
count-web) 1.0.0: :
What is SCM release tag or label for "Account Parent"? (com.juvenxu.mvnbook.account:ac-
count-parent) account-parent-1.0.0: :
What is the new development version for "Account Parent"? (com.juvenxu.mvnbook.account:
account-parent) 1.0.1-SNAPSHOT: :
What is the new development version for "Account Email"? (com.juvenxu.mvnbook.account:
account-email) 1.0.1-SNAPSHOT: :
What is the new development version for "Account Persist"? (com.juvenxu.mvnbook.account:
account-persist) 1.0.1-SNAPSHOT: :
What is the new development version for "Account Captcha"? (com.juvenxu.mvnbook.account:
account-captcha) 1.0.1-SNAPSHOT: :
What is the new development version for "Account Service"? (com.juvenxu.mvnbook.account:
account-service) 1.0.1-SNAPSHOT: :
What is the new development version for "Account Web"? (com.juvenxu.mvnbook.account:ac-
count-web) 1.0.1-SNAPSHOT: :
```

在很多情况下，我们会希望所有模块的发布版本以及新的 SNAPSHOT 开发版本都保持一致。为了避免重复确认，maven-release-plugin 提供了 autoVersionSubmodules 参数。例如运行下面的命令后，maven-release-plugin 就会自动为所有子模块使用与父模块一致的发布版本和新的 SNAPSHOT 版本：

```
$mvn release:prepare -DautoVersionSubmodules=true
```

如果检查下来 release: prepare 的结果没有问题，标签和新的开发版本都是正确的，可以执行如下发布执行命令：

```
$mvn release:perform
```

该命令将标签中的代码签出，执行 mvn deploy 命令构建刚才准备的 1.0.0 版本，并部署到仓库中。至此，版本 1.0.0 正式发布完成。由于它已经被部署到了 Maven 仓库中，其他人可以方便地配置对它的依赖。

细心的读者可能会发现，如果你所发布项目的打包类型为 jar，在执行 release:perform 之后，不仅项目的主构件会被生成并发布到仓库中，基于该主构件的-sources. jar 和-javadoc. jar 也会生成并发布。对于你的用户来说，这无疑是非常方便的，他们不仅能够下载你的主构件，还能够得到项目的源码和 Javadoc。那么，release: perform 是怎样生成-sources. jar和-javadoc. jar 的呢？

8.5节介绍过，所有Maven项目的POM都继承自超级POM，而如果打开超级POM，就能发现如代码清单13-3所示内容。

代码清单13-3 超级POM中sources和javadoc的配置

```
<profiles>
  <profile>
    <id>release-profile</id>

    <activation>
      <property>
        <name>performRelease</name>
        <value>true</value>
      </property>
    </activation>

    <build>
      <plugins>
        <plugin>
          <inherited>true</inherited>
          <artifactId>maven-source-plugin</artifactId>
          <executions>
            <execution>
              <id>attach-sources</id>
              <goals>
                <goal>jar</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
        <plugin>
          <inherited>true</inherited>
          <artifactId>maven-javadoc-plugin</artifactId>
          <executions>
            <execution>
              <id>attach-javadocs</id>
              <goals>
                <goal>jar</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
        <plugin>
          <inherited>true</inherited>
          <artifactId>maven-deploy-plugin</artifactId>
          <configuration>
            <updateReleaseInfo>true</updateReleaseInfo>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

超级 POM 中定义了一个名为 release-profile 的 Maven Profile，Profile 是指一段在特定情况下被激活并更改 Maven 行为的配置，本书后续会有专门的章节详细阐述。这里看到 activate 元素下有一个名为 performRelease、值为 true 的属性配置，这表示当 Maven 运行时，如果运行环境中有 performRelease 属性且值为 true 的时候，该 Profile 就被激活。也就是说，该 Profile 下的配置会得到应用。那么，什么情况下 Maven 运行环境中会有名为 performRelease、值为 true 的属性呢？可以在命令行指定。例如：

```
$ mvn clean install -DperformRelease=true
```

但是，读者可能已经猜到了，在执行 release:perform 的时候，Maven Release Plugin 会自动生成值为 true 的 performRelease 属性。这时，超级 POM 中的 release-profile 就会被激活。

这个 Profile 配置了 3 个 Maven 插件，maven-sources-plugin 的 jar 目标会为项目生成 -source.jar文件，maven-javadoc-plugin 的 jar 目标会为项目生成-javadoc.jar 文件，而 maven-deploy-plugin 的 update-release-info 配置则会在部署的时候更新仓库中的元数据，告诉仓库该版本是最新的发布版。每个插件配置中值为 true 的 inherited 元素则表示该插件配置可以被子 POM 继承。

在日常的快照开发过程中，往往没有必要每次都生成-source.jar 和-javadoc.jar，但是当项目发布的时候，这些文件就显得十分重要。超级 POM 中的 release-profile 就是为了这种情形而设计的。需要注意的是，这种隐式的配置对于不熟悉 Maven 的用户来说可能会显得十分令人费解，因此将来的 Maven 版本中可能会从超级 POM 中移除这段配置，所以如果用户希望在发布版本时自动生成-sources.jar 和-javadoc.jar，最好还是在自己的 POM 中显式地配置这些插件。

13.5 自动化创建分支

13.4 节介绍了如何使用 Maven Release Plugin 自动化版本发布，如果回顾一下图 13-2，就会发现分支创建的操作还没有具体涉及。本节就继续基于实际的样例讲解如何自动化创建分支。

在图 13-2 中可以看到，在正式发布版本 1.1.0 的同时，还可以创建一个分支用来修复将来这个版本可能遇到的重大 Bug。这个过程可以手工完成，例如使用 svn copy 操作将主干代码复制到一个名为 1.1.x 的分支中，然后修改分支中的 POM 文件，升级其版本为 1.1.1-SNAPSHOT，这会涉及很多 Subversion 操作。

使用 Maven Release Plugin 的 branch 目标，它能够帮我们自动化这些操作：

- 检查本地有无未提交的代码。
- 为分支更改 POM 的版本，例如从 1.1.0-SNAPSHOT 改变成 1.1.1-SNAPSHOT。
- 将 POM 中的 SCM 信息更新为分支地址。
- 提交以上更改。
- 将主干的代码复制到分支中。

- 修改本地代码使其回退到分之前的版本（用户可以指定新的版本）。
- 提交本地更改。

当然，为了让 Maven Release Plugin 为我们工作，和版本发布一样，必须在 POM 中提供正确的 SCM 信息。此外，由于分支操作会涉及版本控制系统里的分支地址，因此还要为 Maven Release Plugin 配置分支基础目录，如代码清单 13-4 所示。

代码清单 13-4 配置 maven-release-plugin 提供分支基础目录

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-release-plugin</artifactId>
    <version>2.0</version>
    <configuration>
        <tagBase>https://192.168.1.103/app/tags/</tagBase>
        <branchBase>https://192.168.1.103/app/branches/</branchBase>
    </configuration>
</plugin>
```

然而 tagBase 和 branchBase 并非是一定要配置的。如果为版本控制仓库使用了标准的 Subversion 布局，即在平行的 trunk/tags/branches 目录下分别放置项目主干代码、标签代码和分支代码，那么 Maven Release Plugin 就能够自动根据主干代码位置计算出标签及分支代码位置，因此你就可以省略这两项配置。

理解了创建分支所将执行的实际行为后，就可以在项目目录下运行如下命令以创建分支：

```
$mvn release:branch -DbranchName=1.1.x \
-DupdateBranchVersions=true -DupdateWorkingCopyVersions=false
```

上述命令中使用了 Maven Release Plugin 的 branch 目标，-DbranchName=1.1.x 用来配置所要创建的分支的名称，-DupdateBranchVersions=true 表示为分支使用新的版本，-DupdateWorkingCopyVersions=false 表示不更新本地代码（即主干）的版本。运行上述命令之后，Maven 会提示输入分支项目的版本。例如：

```
What is the branch version for "app"? (com.juvenxu.mvnbook:app) 1.1.1-SNAPSHOT: :
```

用户根据自己的需要为分支输入新的版本后按 Enter 键，Maven 就会处理其余的操作。最后，用户就能在源码库中找到 Maven 创建的分支，如 https://192.168.1.103/app/branches/1.1.x/。在这里，POM 中的版本已经升级到了 1.1.1-SNAPSHOT。

13.6 GPG 签名

当从中央仓库下载第三方构件的时候，你可能会想要验证这些文件的合法性，例如它们是由开源项目官方发布的，并且没有被篡改过。同样地，当发布自己项目给客户使用的时候，你的客户也会想要验证这些文件是否是由你的项目组发布的，且没有被恶意篡改过。

PGP（Pretty Good Privacy）就是这样一个用来帮助提高安全性的技术。PGP 最常用来给电子邮件进行加密、解密以及提供签名，以提高电子邮件交流的安全性。本节介绍如何使用 PGP 技术为发布的 Maven 构件签名，为项目增强安全性。

13.6.1 GPG 及其基本使用

GnuPG（简称 GPG，来自 http://www.gnupg.org/）是 PGP 标准的一个免费实现，无论是类 UNIX 平台还是 Windows 平台，都可以使用它。GPG 能够帮助我们为文件生成签名、管理密钥以及验证签名等。

首先，访问 http://www.gnupg.org/download/ 并下载对应自己平台的 GPG 分发包，按照官方的文档将 GPG 安装完毕，运行如下命令检查安装：

```
juven@ juven-ubuntu:~ $ gpg --version
gpg (GnuPG) 1.4.9Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3 + : GNU GPL version 3 or later
...
```

在使用 GPG 之前，先得为自己准备一个密钥对，即一个私钥和一个公钥。之后才可以使用私钥对文件进行签名，并且将公钥分发到公钥服务器供其他用户下载，用户可以使用公钥对签名进行验证。

使用如下命令生成密钥对：

```
juven@ juven-ubuntu:~ $ gpg --gen-key
```

GPG 会问你密钥的类型、大小和有效时间，通常使用默认的值即可。GPG 还会要求你输入自己的名称、电子邮件地址和对密钥的注释，这些内容会被包含在公钥中并被你的用户看到，因此务必正确填写。最后，还可以提供一个密码来保护密钥，这不是强制性的，但通常最好提供以防止别人得到你的密钥后恶意使用。你将来需要使用私钥和密码为文件提供签名，因此一定要认证保护它们。

现在已经有了密钥对，就可以在命令行中查看它们（其他导入到本地机器的密钥也会被显示），如下面的命令可用来列出所有公钥：

```
juven@ juven-ubuntu:~ $ gpg --list-keys
/home/juven/.gnupg/pubring.gpg
------------------------------
pub  1024D/C6EED57A  2010-01-13
uid                  Juven Xu (Juven Xu works at Sonatype) juven@ sonatype.com
sub  2048g/D704745C 2010-01-13
```

这里的/home/juven/.gnupg/pubring.gpg 表示公钥存储的位置。以 pub 开头的一行显示公钥的长度（1024D）、ID（C6EED57A）以及创建日期（2010-01-13）。下一行显示了公钥的 UID，也就是一个由名称、注释和邮件地址组成的字符串。最后一行显示的子钥不用关心。

类似地，下面的命令用来列出本机私钥：

```
juven@ juven-ubuntu:~ $ gpg --list-secret-keys
/home/juven/.gnupg/secring.gpg
------------------------------
sec  1024D/C6EED57A  2010-01-13
uid                  Juven Xu (Juven Xu works at Sonatype)
ssb  2048g/D704745C  2010-01-13
```

对GPG的公私钥有了基本的了解之后，就可以使用如下命令为任意文件创建一个ASCII格式的签名：

```
juven@ juven-ubuntu:~ $ gpg -ab temp.java
```

这里的-a选项告诉GPG创建ASCII格式的输出，而-b选项则告诉GPG创建一个独立的签名文件。如果你的私钥拥有密码，这个时候就需要输入密码。如果私钥没有密码，那么只要他人获得了你的私钥，就能够以你的名义对任何内容进行签名，这是非常危险的。

在该例中，GPG会创建一个名为temp.java.asc的签名文件，这时就可以将这个后缀名为.asc的签名文件连同原始文件一起分发给你的用户。如果你的用户已经导入了你的公钥，就可以运行如下命令验证原始文件：

```
$ gpg --verify temp.java.asc
```

为了能让你的用户获取公钥并验证你分发的文件，需要将公钥分发到公钥服务器中。例如，hkp://pgp.mit.edu是美国麻省理工学院提供的公钥服务器，运行如下命令可将公钥分发到该服务器中：

```
$ gpg --keyserver hkp://pgp.mit.edu --send-keys  C6EED57A
```

这里的--keyserver选项用来指定分发服务器的地址，--send-keys用来指定想要分发公钥的ID。你可以罗列本地公钥来查看它们的ID。需要注意的是，公钥会在各个公钥服务器中被同步，因此你不需要重复地往各个服务器分发同一公钥。

现在，你的用户可以将服务器上的公钥导入到本地机器：

```
$ gpg --keyserver hkp://pgp.mit.edu --recv-keys  C6EED57A
```

上述就是一个基本的签名、分发并验证的流程，在使用Maven发布项目的时候，可以使用GPG为发布文件提供签名。现在读者应该已经知道如何手工完成这一步骤了，下面介绍如何使用Maven GPG Plugin自动化签名这一步骤。

13.6.2 Maven GPG Plugin

手动地对Maven构件进行签名并将这些签名部署到Maven仓库中是一件耗时的体力活。而使用Maven GPG Plugin只需要提供几行简单的配置，它就能够帮我们自动完成签名这一工作。

在使用Maven GPG Plugin之前，首先需要确认命令行下的gpg是可用的，然后如代码清单13-5所示配置POM。

代码清单 13-5 配置 maven-gpg-plugin 为项目提供签名

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-gpg-plugin</artifactId>
        <version>1.0</version>
        <executions>
          <execution>
            <id>sign-artifacts</id>
            <phase>verify</phase>
            <goals>
              <goal>sign</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

然后就可以使用一般的 mvn 命令签名并发布项目构件：

```
$mvn clean deploy -Dgpg.passphrase=yourpassphrase
```

如果不提供 -Dgpg. passphrase 参数，运行时就会要求输入密码。

如果有一些已经发布了但没有被签名的文件，你仍然想对其签名并发布到 Maven 仓库中，上述方式显然是行不通的，因为 POM 已经不允许被修改。好在 Maven GPG Plugin 为此提供了另外一个目标。例如：

```
$mvn gpg:sign-and-deploy-file
> -DpomFile=target/myapp-1.0.pom
> -Dfile=target/myapp-1.0.jar
> -Durl=http://oss.sonatype.org/service/local/staging/deploy/maven2/
> -DrepositoryId=sonatype_oss
```

在这里可以指定要签名的 POM 及相关文件、Maven 仓库的地址和 ID，Maven GPG Plugin 就会帮你签名文件并部署到仓库中。

读者可以想到，GPG 签名这一步骤只有在项目发布时才显得必要，对日常的 SNAPSHOT 构件进行签名不仅没有多大的意义，反而会比较耗时。因此，只需要配置 Maven PGP Plugin 在项目发布的时候运行，那么如何判断项目发布呢？回顾代码清单 13-3，在超级 POM 中有一个 release-profile，该 Profile 只有在 Maven 属性 performRelease 为 true 的时候才被激活，而 release:perform 执行的时候，就会将该属性置为 true，这正是项目进行版本发布的时刻。因此，类似地，可以在 settings. xml 或者 POM 中创建如代码清单 13-6 所示 Profile。

代码清单 13-6 配置自动激活的 Profile 对项目进行签名

```
<profiles>
  <profile>
    <id>release-sign-artifacts</id>
    <activation>
      <property>
        <name>performRelease</name>
        <value>true</value>
      </property>
    </activation>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-gpg-plugin</artifactId>
          <version>1.0</version>
          <executions>
            <execution>
              <id>sign-artifacts</id>
              <phase>verify</phase>
              <goals>
                <goal>sign</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

最后需要一提的是，由于一个已知的 Maven Release Plugin 的 Bug，release:perform 执行过程中签名可能会导致进程永久挂起。为了避免该情况，用户需要为 Maven Release Plugin 提供 mavenExecutorId 配置，如代码清单 13-7 所示。

代码清单 13-7 配置 maven-release-plugin 避免签名时永久挂起

```
<plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-release-plugin</artifactId>
      <version>2.0</version>
      <configuration>
            <tagBase>https://192.168.1.103/app/tags/</tagBase>
            <branchBase>https://192.168.1.103/app/branches/</branchBase>
            <mavenExecutorId>forked-path</mavenExecutorId>
      </configuration>
</plugin>
```

至此，一个较为规范的自动化签名配置就完成了。当执行 release:perform 发布项目版本的时候，maven-gpg-plugin 会被自动调用对构件进行签名。当然，这个时候你需要根据命令行提示输入私钥密码。

13.7 小结

项目开发到一定阶段后，就必然要面对版本发布的问题，本章介绍了 Maven 的版本管理方式，包括快照版和发布版之间的转换、各种版本号的意义以及项目版本与版本控制系统（如 Subversion）之间的关系。理解了版本转换与 SCM 操作的关系后，就可以使用 Maven Release Plugin 自动化版本发布和创建分支等操作。本章最后介绍了如何在版本发布的时候使用 GPG 为构件提供签名，以提供更强的安全性。