InfoQ Maven专栏（二）——POM重构之增还是删

December 31st, 2010

重构是广大开发者再熟悉不过的技术，在Martin Fowler的《重构——改善既有代码的设计》一书中，其定义为“重构(名词):对软件内部结构的一种调整,目的是在不改变软件之可察行为前提下,提高其可理解性,降低其修改成本.”以及“重构(动词):使用一系列重构准则(手法),在不改变软件之可察行为前提下,调整其结构.”。重构能够改善软件设计，使代码更易读，更容易找出bug，并帮助你更快速地编码。较之于一般的代码来说，Maven的POM简单很多，不过随着项目的成长，模块的增多，POM的内容也会变多，这个时候，我们可以对POM进行重构，在保持构建成功的前提下，简化POM内容，使其更简洁易懂。

前提

大家都知道，如果没有单元测试为前提，对代码进行重构是非常危险的。同样，在重构POM之前，项目应该有足够的自动化测试保证POM重构不会破坏构建。例如，重构POM的时候可能会删除或添加依赖，造成依赖版本变化，依赖范围变化，插件版本变化等等，这些变化可能会导致项目编译失败，或者测试失败。在自动化测试及构建的基础上，最好能够有持续集成环境，以保证POM的修改可能造成的问题能够及时的被发现和修复。笔者目前工作的项目有一个对应的持续集成任务，该任务基于Hudson，每10分钟检查一次SCM，如果发现变更则构建项目，并反馈结果。这样，我就不用担心自己修改POM会引入潜在的问题。

增还是删

有时候这个答案是很显然的，当你的POM中存在一些依赖或者插件配置，但实际代码没有用到这些配置的时候，应该尽早删掉它们以免给人带来困惑。

还有一种常见的情况，我们可以删掉一些POM的元素，例如POM中配置了继承，当前模块与父模块使用同样的groupId和version时，就可以将<groupId>和<version>元素删除，因为它们可以从父模块继承而来，重复配置没有什么意义。

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.juvenxu.sample</groupId>
    <artifactId>sample-parent</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <artifactId>sample-foo</artifactId>
  <packaging>jar</packaging>
...
</project>
```

上述配置sample-foo就没有groupId和version，需要注意的是，artifactId是不能被删除的，因为该元素不能也不应该被继承，父子模块应当使用不同的artifactId值。

除了删之外，有些时候我们还需要在POM中增加一些XML元素，目的是为了让POM更清晰易读，且保证Maven构建的稳定性。考虑如下的插件配置：

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.5</source>
    <target>1.5</target>
  </configuration>
</plugin>
```

虽然没有groupId及version，但这段配置是完全合法的。当插件没有groupId配置的时候，Maven会认为它是官方插件而自动使用org.apache.maven.plugins作为groupId，当插件没有version配置的时候，Maven则会使用最新的版本（Maven 2会使用最新的版本，包括SNAPSHOT，而Maven 3则只使用最新的非SNAPSHOT版本）。这段配置有两个问题，首先，如果让一个不熟悉Maven的开发者来看这段配置，他会感到费解，groupId和version究竟是什么呢？这与不清晰的代码是一个意思，有时候一些程序员为了让代码更短，会采用一些奇怪的语法和变量名，虽然代码量是少了，但沟通成本增加了，得不偿失。其次，让Maven猜测版本会有潜在的风险，因

为插件的最新版本可能会变化，而这种变化对于Maven使用者来说通常是隐藏的，特别是在Maven 2中，甚至可能引入SNAPSHOT版本的插件，这是非常危险的。基于这两个原因，使用插件的时候，我们应当配置清楚groupId和version，如：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>2.3.2</version>
  <configuration>
    <source>1.5</source>
    <target>1.5</target>
  </configuration>
</plugin>
```

基于类似的原因，在配置项目依赖的时候，我们也应当一直显式地写明依赖版本，以避免Maven在不同的时刻引入不同版本的依赖而导致项目构建的不稳定。

除了上面提到的增删点之外，Maven官方还提供了一个非常有用的Maven Dependency Plugin来帮助我们分析项目中哪些依赖配置应该删除，那些依赖配置应该增加。Maven Dependency Plugin的analyze目标能够帮助分析项目依赖，例如运行命令 **mvn dependency:analyze** ，可以看到如下输出：

```
[INFO] --- maven-dependency-plugin:2.1:analyze (default-cli) @ sample-bar ---
[WARNING] Used undeclared dependencies found:
[WARNING]    org.springframework:spring-context:jar:2.5.6:compile
[WARNING] Unused declared dependencies found:
[WARNING]    org.springframework:spring-core:jar:2.5.6:compile
[WARNING]    org.springframework:spring-beans:jar:2.5.6:compile
[INFO] ------------------------------------------------------------------------
```

这里的 Used undeclared dependencies 是指那些在项目中直接使用到的，但没有在POM中配置的依赖。例如该例中可能项目中的一些类有关于spring-context的Java import声明，但spring-context这个依赖实际是通过传递性依赖进入classpath的，这就意味着潜在的风险。一般来说我们对直接依赖的版本变化会比较清楚，因为那是我们自己直接配置的，但对于传递性依赖的版本变化，就会比较模糊，当这种变化造成构建失败的时候，就很难找到原因。因此我们应当增加这些 Used undeclared dependencies 。

依赖分析还提供了 Unused declared dependencies 供我们参考，这表示那些我们配置了，但并未直接使用的依赖。需要注意的是，对于这些依赖，我们不该直接简单地删除。由于dependency:analyze只分析编译主代码和测试代码使用的依赖，一些执行测试和运行时的依赖它发现不了，因此还需要人工分析。通常情况，Unused declared dependencies 还是能帮助我们发现一些无用的依赖配置。

最后，还一些重要的POM内容通常被大多数项目所忽略，这些内容不会影响项目的构建，但能方便信息的沟通，它们包括项目URL，开发者信息，SCM信息，持续集成服务器信息等等，这些信息对于开源项目来说尤其重要。对于那些想了解项目的人来说，这些信息能他们帮助找到想要的信息，基于这些信息生成的Maven站点也更有价值。相关的POM配置很简单，如：

```
<project>
  <description>...</description>
  <url>...</url>
  <licenses>...</licenses>
  <organization>...</organization>
  <developers>...</developers>
  <issueManagement>...</issueManagement>
  <ciManagement>...</ciManagement>
  <mailingLists>...</mailingLists>
  <scm>...</scm>
</project>
```

小结

无论是对POM内容进行增还是删，其目的都是一样的，就是*为了让POM更清晰易懂且让构建更稳定*。从这点来说，POM重构与一般的代码重构是类似的。需要谨记的是，重构的前提是完善的自动化测试和持续集成。本文介绍的单个POM规模的重构，下篇文章笔者会介绍多模块项目的POM重构等内容。

原创文章，转载请注明出处，本文地址： http://www.juvenxu.com/2010/12/31/infoq-maven-pom-refactoring-add-or-delete/