InfoQ Maven专栏（三）——多模块项目的POM重构

January 10th, 2011

在本专栏的上一篇文章POM重构之增还是删中，我们讨论了一些简单实用的POM重构技巧，包括重构的前提——持续集成，以及如何通过添加或者删除内容来提高POM的可读性和构建的稳定性。但在实际的项目中，这些技巧还是不够的，特别值得一提的是，实际的Maven项目基本都是多模块的，如果仅仅重构单个POM而不考虑模块之间的关系，那就会造成无谓的重复。本文就讨论一些基于多模块的POM重构技巧。

重复，还是重复

程序员应该有狗一般的嗅觉，要能嗅到重复这一最常见的坏味道，不管重复披着怎样的外衣，一旦发现，都应该毫不留情地彻底地将其干掉。不要因为POM不是产品代码而纵容重复在这里发酵，例如这样一段代码就有重复：

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactid>spring-beans</artifactId>
  <version>2.5</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactid>spring-context</artifactId>
  <version>2.5</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactid>spring-core</artifactId>
  <version>2.5</version>
</dependency>
```

你会在一个项目中使用不同版本的SpringFramework组件么？答案显然是不会。因此这里就没必要重复写三次<version>2.5</version>，使用Maven属性将2.5提取出来如下：

```
<properties>
  <spring.version>2.5</spring.version>
</properties>
<depencencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactid>spring-beans</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactid>spring-context</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactid>spring-core</artifactId>
    <version>${spring.version}</version>
  </dependency>
</depencencies>
```

现在2.5只出现在一个地方，虽然代码稍微长了点，但重复消失了，日后升级依赖版本的时候，只需要修改一处，而且也能避免漏掉升级某个依赖。

读者可能已经非常熟悉这个例子了，我这里再啰嗦一遍是为了给后面做铺垫，多模块POM重构的目的和该例一样，也是为了消除重复，模块越多，潜在的重复就越多，重构就越有必要。

消除多模块依赖配置重复

考虑这样一个不大不小的项目，它有10多个Maven模块，这些模块分工明确，各司其职，相互之间耦合度比较小，这样大家就能够专注在自己的模块中进行开发而不用过多考虑他人对自己的影响。（好吧，我承认这是比较理想的情况）那我开始对模块A进行编码了，首先就需要引入一些常见的依赖如JUnit、Log4j等等：

```xml
  <dependency>
    <groupId>junit</groupId>
    <artifactid>junit</artifactId>
    <version>4.8.2</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactid>log4j</artifactId>
    <version>1.2.16</version>
  </dependency>
```

我的同事在开发模块B，他也要用JUnit和Log4j（我们开会讨论过了，统一单元测试框架为JUnit而不是TestNG，统一日志实现为Log4j而不是JUL，为什么做这个决定就不解释了，总之就这么定了）。同事就写了如下依赖配置：

```xml
  <dependency>
    <groupId>junit</groupId>
    <artifactid>junit</artifactId>
    <version>3.8.2</version>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactid>log4j</artifactId>
    <version>1.2.9</version>
  </dependency>
```

看出什么问题来没有？对的，他漏了JUnit依赖的scope，那是因为他不熟悉Maven。还有什么问题？对，版本！虽然他和我一样都依赖了JUnit及Log4j，但版本不一致啊。我们开会讨论没有细化到具体用什么版本，但如果一个项目同时依赖某个类库的多个版本，那是十分危险的！OK，现在只是两个模块的两个依赖，手动修复一下没什么问题，但如果是10个模块，每个模块10个依赖或者更多呢？看来这真是一个泥潭，一旦陷进去就难以收拾了。

好在Maven提供了优雅的解决办法，使用继承机制以及dependencyManagement元素就能解决这个问题。注意，是dependencyMananget而非dependencies。也许你已经想到在父模块中配置dependencies，那样所有子模块都自动继承，不仅达到了依赖一致的目的，还省掉了大段代码，但这么做是有问题的，例如你将模块C的依赖spring-aop提取到了父模块中，但模块A和B虽然不需要spring-aop，但也直接继承了。dependencyManagement就没有这样的问题，**dependencyManagement只会影响现有依赖的配置，但不会引入依赖**。例如我们可以在父模块中配置如下：

```xml
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactid>junit</artifactId>
      <version>4.8.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactid>log4j</artifactId>
      <version>1.2.16</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

这段配置不会给任何子模块引入依赖，但如果某个子模块需要使用JUnit和Log4j的时候，我们就可以简化依赖配置成这样：

```xml
 <dependency>
   <groupId>junit</groupId>
   <artifactid>junit</artifactId>
 </dependency>
 <dependency>
   <groupId>log4j</groupId>
   <artifactid>log4j</artifactId>
```

```
  </dependency>
```

现在只需要groupId和artifactId，其它元素如version和scope都能通过继承父POM的dependencyManagement得到，如果有依赖配置了exclusions，那节省的代码就更加可观。但重点不在这，重点在于现在能够保证所有模块使用的JUnit和Log4j依赖配置是一致的。而且子模块仍然可以按需引入依赖，如果我不配置dependency，父模块中dependencyManagement下的spring-aop依赖不会对我产生任何影响。

也许你已经意识到了，**在多模块Maven项目中，dependencyManagement几乎是必不可少的，因为只有它是才能够有效地帮我们维护依赖一致性**。

本来关于dependencyManagement我想介绍的也差不多了，但几天前和Sunng的一次讨论让我有了更多的内容分享。那就是在使用dependencyManagement的时候，我们可以不从父模块继承，而是使用特殊的import scope依赖。Sunng将其列为自己的Maven Recipe #0，我再简单介绍下。

我们知道Maven的继承和Java的继承一样，是无法实现多重继承的，如果10个、20个甚至更多模块继承自同一个模块，那么按照我们之前的做法，这个父模块的dependencyManagement会包含大量的依赖。如果你想把这些依赖分类以更清晰的管理，那就不可能了，import scope依赖能解决这个问题。你可以把dependencyManagement放到单独的专门用来管理依赖的POM中，然后在需要使用依赖的模块中通过import scope依赖，就可以引入dependencyManagement。例如可以写这样一个用于依赖管理的POM：

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.juvenxu.sample</groupId>
  <artifactId>sample-dependency-infrastructure</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencyManagement>
    <dependencies>
        <dependency>
          <groupId>junit</groupId>
          <artifactid>junit</artifactId>
          <version>4.8.2</version>
          <scope>test</scope>
        </dependency>
        <dependency>
          <groupId>log4j</groupId>
          <artifactid>log4j</artifactId>
          <version>1.2.16</version>
        </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```

然后我就可以通过非继承的方式来引入这段依赖管理配置：

```
  <dependencyManagement>
    <dependencies>
        <dependency>
          <groupId>com.juvenxu.sample</groupId>
          <artifactid>sample-dependency-infrastructure</artifactId>
          <version>1.0-SNAPSHOT</version>
          <type>pom</type>
          <scope>import</scope>
        </dependency>
    </dependencies>
  </dependencyManagement>

  <dependency>
    <groupId>junit</groupId>
    <artifactid>junit</artifactId>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactid>log4j</artifactId>
  </dependency>
```

这样，父模块的POM就会非常干净，由专门的packaging为pom的POM来管理依赖，也契合的面向对象设计中的单一职责原则。此外，我们还能够创建多个这样的依赖管理POM，以更细化的方式管理依赖。这种做法与面向对象设计中使用组合而非继承也有点相似的味道。

消除多模块插件配置重复

与dependencyManagement类似的，我们也可以使用pluginManagement元素管理插件。一个常见的用法就是我们希望项目所有模块的使用Maven Compiler Plugin的时候，都使用Java 1.5，以及指定Java源文件编码为UTF-8，这时可以在父模块的POM中如下配置pluginManagement：

```
<build>
  <pluginManagement>
    <plugins>
      <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>
          <encoding>UTF-8</encoding>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
```

这段配置会被应用到所有子模块的maven-compiler-plugin中，由于Maven内置了maven-compiler-plugin与生命周期的绑定，因此子模块就不再需要任何maven-compiler-plugin的配置了。

与依赖配置不同的是，通常所有项目对于任意一个依赖的配置都应该是统一的，但插件却不是这样，例如你可以希望模块A运行所有单元测试，模块B要跳过一些测试，这时就需要配置maven-surefire-plugin来实现，那样两个模块的插件配置就不一致了。这也就是说，简单的把插件配置提取到父POM的pluginManagement中往往不适合所有情况，那我们在使用的时候就需要注意了，只有那些普适的插件配置才应该使用pluginManagement提取到父POM中。

关于插件pluginManagement，Maven并没有提供与import scope依赖类似的方式管理，那我们只能借助继承关系，不过好在一般来说插件配置的数量远没有依赖配置那么多，因此这也不是一个问题。

小结

关于Maven POM重构的介绍，在此就告一段落了。基本上如果你掌握了本篇和上一篇Maven专栏讲述的重构技巧，并理解了其背后的目的原则，那么你肯定能让项目的POM变得更清晰易懂，也能尽早避免一些潜在的风险。虽然Maven只是用来帮助你构建项目和管理依赖的工具，POM也并不是你正式产品代码的一部分。但我们也应该认真对待POM，这有点像测试代码，以前可能大家都觉得测试代码可有可无，更不会去用心重构优化测试代码，但随着敏捷开发和TDD等方式越来越被人接受，测试代码得到了开发人员越来越多的关注。因此这里我希望大家不仅仅满足于一个“能用”的POM，而是能够积极地去修复POM中的坏味道。

原创文章，转载请注明出处，本文地址： *http://www.juvenxu.com/2011/01/10/infoq-maven-pom-refactoring-multi-module/*