第 14 章

灵活的构建

本章内容

- Maven 属性
- 构建环境的差异
- 资源过滤
- Maven Profile
- Web 资源过滤
- 在 profile 中激活集成测试
- 小结

一个优秀的构建系统必须足够灵活，它应该能够让项目在不同的环境下都能成功地构建。例如，典型的项目都会有开发环境、测试环境和产品环境，这些环境的数据库配置不尽相同，那么项目构建的时候就需要能够识别所在的环境并使用正确的配置。还有一种常见的情况是，项目开发了大量的集成测试，这些测试运行起来非常耗时，不适合在每次构建项目的时候都运行，因此需要一种手段能让我们在特定的时候才激活这些集成测试。Maven 为了支持构建的灵活性，内置了三大特性，即属性、Profile 和资源过滤。本章介绍如何合理使用这些特性来帮助项目自如地应对各种环境。

14.1 Maven 属性

前面的章节已经简单介绍过 Maven 属性的使用，例如在 5.9.2 节有如代码清单 14-1 所示的代码。

代码清单 14-1 使用 Maven 属性归类依赖

```
<properties>
  <springframework.version>2.5.6</springframework.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${springframework.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>${springframework.version}</version>
    </dependency>
    ...
</dependencies>
```

这可能是最常见的使用 Maven 属性的方式，通过 <properties> 元素用户可以自定义一个或多个 Maven 属性，然后在 POM 的其他地方使用 ${属性名称} 的方式引用该属性，这种做法的最大意义在于消除重复。例如，代码清单 14-1 中本来需要在多个地方重复声明同样的 SpringFramework 版本，现在只在一个地方声明就可以，重复越多，好处就越明显。因为这样不仅减少了日后升级版本的工作量，也能降低错误发生的概率。

这不是 Maven 属性的全部，事实上这只是 6 类 Maven 属性中的一类而已。这 6 类属性分别为：

- **内置属性**：主要有两个常用内置属性——${basedir} 表示项目根目录，即包含 pom.xml 文件的目录；${version} 表示项目版本。
- **POM 属性**：用户可以使用该类属性引用 POM 文件中对应元素的值。例如

${project.artifactId}就对应了 <project> <artifactId> 元素的值，常用的 POM 属性包括：

- ${project.build.sourceDirectory}：项目的主源码目录，默认为 src/main/java/。
- ${project.build.testSourceDirectory}：项目的测试源码目录，默认为 src/test/java/。
- ${project.build.directory}：项目构建输出目录，默认为 target/。
- ${project.outputDirectory}：项目主代码编译输出目录，默认为 target/classes/。
- ${project.testOutputDirectory}：项目测试代码编译输出目录，默认为 target/test-classes/。
- ${project.groupId}：项目的 groupId。
- ${project.artifactId}：项目的 artifactId。
- ${project.version}：项目的 version，与 ${version} 等价。
- ${project.build.finalName}：项目打包输出文件的名称，默认为 ${project.artifactId}-${project.version}。

这些属性都对应了一个 POM 元素，它们中一些属性的默认值都是在超级 POM 中定义的，可以参考 8.5 节。

❑ **自定义属性**：用户可以在 POM 的 <properties> 元素下自定义 Maven 属性。例如：

```
<project>
  ...
  <properties>
    <my.prop>hello</my.prop>
  </properties>
  ...
</project>
```

然后在 POM 中其他地方使用 ${my.prop} 的时候会被替换成 hello。

❑ **Settings 属性**：与 POM 属性同理，用户使用以 settings. 开头的属性引用 settings.xml 文件中 XML 元素的值，如常用的 ${settings.localRepository} 指向用户本地仓库的地址。

❑ **Java 系统属性**：所有 Java 系统属性都可以使用 Maven 属性引用，例如 ${user.home} 指向了用户目录。用户可以使用 mvn help:system 查看所有的 Java 系统属性。

❑ **环境变量属性**：所有环境变量都可以使用以 env. 开头的 Maven 属性引用。例如 ${env.JAVA_HOME} 指代了 JAVA_HOME 环境变量的值。用户可以使用 mvn help:system 查看所有的环境变量。

正确使用这些 Maven 属性可以帮助我们简化 POM 的配置和维护工作，下面列举几个常见的 Maven 属性使用样例。

在一个多模块项目中，模块之间的依赖比较常见，这些模块通常会使用同样的 groupId 和 version。因此这个时候就可以使用 POM 属性，如代码清单 14-2 所示。

代码清单 14-2 使用 POM 属性配置依赖

```
<dependencies>
    <dependency>
        <groupId>${project.groupId}</groupId>
        <artifactId>account-email</artifactId>
        <version>${project.version}</version>
    </dependency>
    <dependency>
        <groupId>${project.groupId}</groupId>
        <artifactId>account-persist</artifactId>
        <version>${project.version}</version>
    </dependency>
</dependencies>
```

在代码清单 14-2 中，当前的模块依赖于 account-email 和 account-persist，这三个模块使用同样的 groupId 和 version，因此可以在依赖配置中使用 POM 属性 ${project.groupId} 和 ${project.version}，表示这两个依赖的 groupId 和 version 与当前模块一致。这样，当项目版本升级的时候，就不再需要更改依赖的版本了。

大量的 Maven 插件用到了 Maven 属性，这意味着在配置插件的时候同样可以使用 Maven 属性来方便地自定义插件行为。例如从 10.6 节我们知道，maven-surefire-plugin 运行后默认的测试报告目录为 target/surefire-reports，这实际上就是 ${project.build.directory}/surefire-reports，如果查阅该插件的文档，会发现该插件提供了 reportsDirectory 参数来配置测试报告目录。因此如果想要改变测试报告目录，例如改成 target/test-reports，就可以像代码清单 14-3这样配置。

代码清单 14-3 使用 Maven 属性配置插件

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.5</version>
    <configuration>

    <reportsDirectory>${project.build.directory}/test-reports</reportsDi-
rectory>
    </configuration>
</plugin>
```

从上面的内容中可以看到，Maven 属性能让我们在 POM 中方便地引用项目环境和构建环境的各种十分有用的值，这是创建灵活构建的基础。下面将会结合 profile 和资源过滤，展示 Maven 能够为构建提供的更多的可能性。

14.2 构建环境的差异

在不同的环境中，项目的源码应该使用不同的方式进行构建，最常见的就是数据库配置了。例如在开发的过程中，有些项目会在 src/main/resources/目录下放置带有如下内容的

数据库配置文件：

```
database.jdbc.driverClass = com.mysql.jdbc.Driver
database.jdbc.connectionURL = jdbc:mysql://localhost:3306/test
database.jdbc.username = dev
database.jdbc.password = dev-pwd
```

这本没什么问题，可当测试人员想要构建项目产品并进行测试的时候，他们往往需要使用不同的数据库。这时的数据库配置文件可能是这样的：

```
database.jdbc.driverClass = com.mysql.jdbc.Driver
database.jdbc.connectionURL = jdbc:mysql://192.168.1.100:3306/test
database.jdbc.username = test
database.jdbc.password = test-pwd
```

连接数据库的 URL、用户名和密码都发生了变化，类似地，当项目被发布到产品环境的时候，所使用的数据库配置又是另外一套了。这个时候，比较原始的做法是，使用与开发环境一样的构建，然后在测试或者发布产品之前再手动更改这些配置。这是可行的，也是比较常见的，但肯定不是最好的方法。本书已经不止一次强调，手动往往就意味着低效和错误，因此需要找到一种方法，使它能够自动地应对构建环境的差异。

Maven 的答案是针对不同的环境生成不同的构件。也就是说，在构建项目的过程中，Maven 就已经将这种差异处理好了。

14.3　资源过滤

为了应对环境的变化，首先需要使用 Maven 属性将这些将会发生变化的部分提取出来。在上一节的数据库配置中，连接数据库使用的驱动类、URL、用户名和密码都可能发生变化，因此用 Maven 属性取代它们：

```
database.jdbc.driverClass = ${db.driver}
database.jdbc.connectionURL = ${db.url}
database.jdbc.username = ${db.username}
database.jdbc.password = ${db.password}
```

这里定义了 4 个 Maven 属性：db.driver、db.url、db.username 和 db.password，它们的命名是任意的，读者可以根据自己的实际情况定义最合适的属性名称。

既然使用了 Maven 属性，就应该在某个地方定义它们。14.1 节介绍过如何自定义 Maven 属性，这里要做的是使用一个额外的 profile 将其包裹，如代码清单 14-4 所示。

代码清单 14-4　针对开发环境的数据库配置

```
<profiles>
  <profile>
    <id>dev</id>
    <properties>
      <db.driver>com.mysql.jdbc.Driver</db.driver>
      <db.url>jdbc:mysql://192.168.1.100:3306/test</db.url>
      <db.username>dev</db.username>
```

```
        <db.password>dev-pwd</db.password>
      </properties>
    </profile>
  </profiles>
```

代码清单 14-4 中的 Maven 属性定义与直接在 POM 的 properties 元素下定义并无二致，这里只是使用了一个 id 为 dev 的 profile，其目的是将开发环境下的配置与其他环境区别开来。关于 profile，本章将详细解释。

有了属性定义，配置文件中也使用了这些属性，一切 OK 了吗？还不行。读者要留意的是，Maven 属性默认只有在 POM 中才会被解析。也就是说，${db.username} 放到 POM 中会变成 test，但是如果放到 src/main/resources/目录下的文件中，构建的时候它将仍然还是 ${db.username}。因此，需要让 Maven 解析资源文件中的 Maven 属性。

资源文件的处理其实是 maven-resources-plugin 做的事情，它默认的行为只是将项目主资源文件复制到主代码编译输出目录中，将测试资源文件复制到测试代码编译输出目录中。不过只要通过一些简单的 POM 配置，该插件就能够解析资源文件中的 Maven 属性，即开启资源过滤。

Maven 默认的主资源目录和测试资源目录的定义是在超级 POM 中（可以回顾 8.5 节）。要为资源目录开启过滤，只要在此基础上添加一行 filtering 配置即可，如代码清单 14-5 所示。

代码清单 14-5　为主资源目录开启过滤

```
<resources>
  <resource>
    <directory>${project.basedir}/src/main/resources</directory>
    <filtering>true</filtering>
  </resource>
<resources>
```

类似地，代码清单 14-6 中的配置为测试资源目录开启了过滤。

代码清单 14-6　为测试资源目录开启过滤

```
<testResources>
  <testResource>
    <directory>${project.basedir}/src/test/resources</directory>
    <filtering>true</filtering>
  </testResource>
</testResources>
```

读者可能还会从上述代码中意识到，主资源目录和测试资源目录都可以超过一个，虽然会破坏 Maven 的约定，但 Maven 允许用户声明多个资源目录，并且为每个资源目录提供不同的过滤配置，如代码清单 14-7 所示。

代码清单 14-7　配置多个资源目录

```
<resources>
  <resource>
```

```
    <directory>src/main/resources</directory>
    <filtering>true</filtering>
  </resource>
  <resource>
    <directory>src/main/sql</directory>
    <filtering>false</filtering>
  </resource>
</resources>
```

代码清单 14-7 配置了两个资源目录，其中 src/main/resources 开启了过滤，而 src/main/sql 没有启用过滤。

到目前为止一切基本就绪了，我们将数据库配置的变化部分提取成了 Maven 属性，在 POM 的 profile 中定义了这些属性的值，并且为资源目录开启了属性过滤。最后，只需要在命令行激活 profile，Maven 就能够在构建项目的时候使用 profile 中属性值替换数据库配置文件中的属性引用。运行命令如下：

```
$mvn clean install-Pdev
```

mvn 的 -P 参数表示在命令行激活一个 profile。这里激活了 id 为 dev 的 profile。构建完成后，输出目录中的数据库配置就是开发环境的配置了：

```
database.jdbc.driverClass=com.mysql.jdbc.Driver
database.jdbc.connectionURL=jdbc:mysql://localhost:3306/test
database.jdbc.username=dev
database.jdbc.password=dev-pwd
```

14.4　Maven Profile

从前面内容我们看到，不同环境的构建很可能是不同的，典型的情况就是数据库的配置。除此之外，有些环境可能需要配置插件使用本地文件，或者使用特殊版本的依赖，或者需要一个特殊的构件名称。要想使得一个构建不做任何修改就能在任何环境下运行，往往是不可能的。为了能让构建在各个环境下方便地移植，Maven 引入了 profile 的概念。profile 能够在构建的时候修改 POM 的一个子集，或者添加额外的配置元素。用户可以使用很多方式激活 profile，以实现构建在不同环境下的移植。

14.4.1　针对不同环境的 profile

继续以 14.2 节介绍的数据库差异为例，代码清单 14-4 引入了一个针对开发环境的 profile，类似地，可以加入测试环境和产品环境的 profile，如代码清单 14-8 所示。

代码清单 14-8　基于开发环境和测试环境的 profile

```
<profiles>
  <profile>
    <id>dev</id>
    <properties>
```

```
      <db.driver>com.mysql.jdbc.Driver</db.driver>
      <db.url>jdbc:mysql://localhost:3306/test</db.url>
      <db.username>dev</db.username>
      <db.password>dev-pwd</db.password>
    </properties>
  </profile>
  <profile>
    <id>test</id>
    <properties>
      <db.driver>com.mysql.jdbc.Driver</db.driver>
      <db.url>jdbc:mysql://192.168.1.100:3306/test</db.url>
      <db.username>test</db.username>
      <db.password>test-pwd</db.password>
    </properties>
  </profile>
</profiles>
```

同样的属性在两个 profile 中的值是不一样的，dev profile 提供了开发环境数据库的配置，而 test profile 提供的是测试环境数据库的配置。类似地，还可以添加一个基于产品环境数据库配置的 profile。由于篇幅原因，在此不再赘述。

现在，开发人员可以在使用 mvn 命令的时候在后面加上-Pdev 激活 dev profile，而测试人员可以使用-Ptest 激活 test profile。

14.4.2 激活 profile

为了尽可能方便用户，Maven 支持很多种激活 Profile 的方式。

1. 命令行激活

用户可以使用 mvn 命令行参数-P 加上 profile 的 id 来激活 profile，多个 id 之间以逗号分隔。例如，下面的命令激活了 dev-x 和 dev-y 两个 profile：

```
$ mvn clean install -Pdev-x,dev-y
```

2. settings 文件显式激活

如果用户希望某个 profile 默认一直处于激活状态，就可以配置 settings.xml 文件的 activeProfiles 元素，表示其配置的 profile 对于所有项目都处于激活状态，如代码清单14-9所示。

代码清单 14-9 settings 文件显式激活 profile

```
<settings>
  ...
  <activeProfiles>
    <activeProfile>dev-x</activeProfile>
  </activeProfiles>
  ...
</settings>
```

9.5 节就曾经用到这种方式默认激活了一个关于仓库配置的 profile。

3. 系统属性激活

用户可以配置当某系统属性存在的时候，自动激活 profile，如代码清单 14-10 所示。

代码清单 14-10　某系统属性存在时激活 profile

```
<profiles>
  <profile>
    <activation>
      <property>
        <name>test</name>
      </property>
    </activation>
    ...
  </profile>
</profiles>
```

可以进一步配置当某系统属性 test 存在，且值等于 x 的时候激活 profile，如代码清单 14-11所示。

代码清单 14-11　某系统属性存在且值确定时激活 profile

```
<profiles>
  <profile>
    <activation>
      <property>
        <name>test</name>
        <value>x</value>
      </property>
    </activation>
    ...
  </profile>
</profiles>
```

不要忘了，用户可以在命令行声明系统属性。例如：

```
$mvn clean install -Dtest=x
```

因此，这其实也是一种从命令行激活 profile 的方法，而且多个 profile 完全可以使用同一个系统属性来激活。

4. 操作系统环境激活

Profile 还可以自动根据操作系统环境激活，如果构建在不同的操作系统有差异，用户完全可以将这些差异写进 profile，然后配置它们自动基于操作系统环境激活，如代码清单 14-12 所示。

代码清单 14-12　基于操作系统环境激活 profile

```
<profiles>
  <profile>
```

```
    <activation>
      <os>
        <name>Windows XP</name>
        <family>Windows</family>
        <arch>x86</arch>
        <version>5.1.2600</version>
      </os>
    </activation>
    ...
  </profile>
</profiles>
```

这里 family 的值包括 Windows、UNIX 和 Mac 等，而其他几项 name、arch、version，用户可以通过查看环境中的系统属性 os. name、os. arch、os. version 获得。

5. 文件存在与否激活

Maven 能够根据项目中某个文件存在与否来决定是否激活 profile，如代码清单 14-13 所示。

代码清单 14-13　基于文件存在与否激活 profile

```
<profiles>
  <profile>
    <activation>
      <file>
        <missing>x.properties</missing>
        <exists>y.properties</exists>
      </file>
    </activation>
    ...
  </profile>
</profiles>
```

6. 默认激活

用户可以在定义 profile 的时候指定其默认激活，如代码清单 14-14 所示。

代码清单 14-14　默认激活 profile

```
<profiles>
  <profile>
    <id>dev</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    ...
  </profile>
</profiles>
```

使用 activeByDefault 元素用户可以指定 profile 自动激活。不过需要注意的是，如果 POM

中有任何一个 profile 通过以上其他任意一种方式被激活了，所有的默认激活配置都会失效。

如果项目中有很多的 profile，它们的激活方式各异，用户怎么知道哪些 profile 被激活了呢？maven-help-plugin 提供了一个目标帮助用户了解当前激活的 profile：

```
$mvn help:active-profiles
```

maven- help- plugin 还有另外一个目标用来列出当前所有的 profile：

```
$mvn help:all-profiles
```

14.4.3　profile 的种类

根据具体的需要，可以在以下位置声明 profile：

- **pom. xml**：很显然，pom. xml 中声明的 profile 只对当前项目有效。
- **用户 settings. xml**：用户目录下 . m2/settings. xml 中的 profile 对本机上该用户所有的 Maven 项目有效。
- **全局 settings. xml**：Maven 安装目录下 conf/settings. xml 中的 profile 对本机上所有的 Maven 项目有效。
- **profiles. xml**（Maven 2）：还可以在项目根目录下使用一个额外的 profiles. xml 文件来声明 profile，不过该特性已经在 Maven 3 中被移除。建议用户将这类 profile 移到 settings. xml 中。

2. 7. 2 节已经解释过，为了不影响其他用户且方便升级 Maven，用户应该选择配置用户范围的 settings. xml，避免修改全局范围的 settings. xml 文件。也正是因为这个原因，一般不会在全局的 settings. xml 文件中添加 profile。

像 profiles. xml 这样的文件，默认是不会被 Maven 安装到本地仓库，或者部署到远程仓库的。因此一般来说应该避免使用，Maven 3 也不再支持该特性。但如果在用 Maven 2，而且需要为几十或者上百个客户执行不同的构建，往 POM 中放置这么多的 profile 可能就不太好。这时可以选择使用 profiles. xml，如代码清单 14-15 所示。

代码清单 14-15　使用 profiles. xml

```
<profiles>
  <profile>
    <id>client-001</id>
    <properties>
      <css.pref>blue.css</css.pref>
    </properties>
  </profile>
  <profile>
    <id>client-002</id>
    <properties>
      <css.pref>red.css</css.pref>
    </properties>
  </profile>
  <profile>
    <id>client-003</id>
```

```
    <properties>
      <css.pref>orange.css</css.pref>
    </properties>
  </profile>
  ...
</profiles>
```

如果是 Maven 3，则应该把这些内容移动到 settings.xml 中。

不同类型的 profile 中可以声明的 POM 元素也是不同的，pom.xml 中的 profile 能够随着 pom.xml 一起被提交到代码仓库中、被 Maven 安装到本地仓库中、被部署到远程 Maven 仓库中。换言之，可以保证该 profile 伴随着某个特定的 pom.xml 一起存在，因此它可以修改或者增加很多 POM 元素，见代码清单 14-16。

代码清单 14-16　POM 中的 profile 可使用的元素

```
<project>
  <repositories></repositories>
  <pluginRepositories></pluginRepositories>
  <distributionManagement></distributionManagement>
  <dependencies></dependencies>
  <dependencyManagement></dependencyManagement>
  <modules></modules>
  <properties></properties>
  <reporting></reporting>
  <build>
    <plugins></plugins>
    <defaultGoal></defaultGoal>
    <resources></resources>
    <testResources></testResources>
    <finalName></finalName>
  </build>
</project>
```

从代码清单 14-16 中可以看到，可供 pom 中 profile 使用的元素非常多，在 pom profile 中用户可以修改或添加仓库、插件仓库以及部署仓库地址；可以修改或者添加项目依赖；可以修改聚合项目的聚合配置；可以自由添加或修改 Maven 属性；添加或修改项目报告配置；pom profile 还可以添加或修改插件配置、项目资源目录和测试资源目录配置以及项目构件的默认名称。

与 pom.xml 中的 profile 对应的，是其他三种外部的 profile，由于无法保证它们能够随着特定的 pom.xml 一起被分发，因此 Maven 不允许它们添加或者修改绝大部分的 pom 元素。举个简单的例子。假设用户 Jack 在自己的 settings.xml 文件中配置了一个 profile，为了让项目 A 构建成功，Jack 在这个 profile 中声明几个依赖和几个插件，然后通过激活该 profile 将项目构建成功了。但是，当其他人获得项目 A 的源码后，它们并没有 Jack settings.xml 中的 profile，因此它们无法构建项目，这就导致了构建的移植性问题。为了避免这种问题的出现，Maven 不允许用户在 settings.xml 的 profile 中声明依赖或者插件。事实上，在 pom.xml 外部的 profile 只能够声明如代码清单 14-17 所示几个元素。

代码清单 14-17　POM 外部的 profile 可使用的元素

```
<project>
  <repositories></repositories>
  <pluginRepositories></pluginRepositories>
  <properties></properties>
</project>
```

现在不用担心 POM 外部的 profile 会对项目产生太大的影响了，事实上这样的 profile 仅仅能用来影响到项目的仓库和 Maven 属性。

14.5　Web 资源过滤

14.3 节介绍了如何开启资源过滤，在 Web 项目中，资源文件同样位于 src/main/resources/目录下，它们经处理后会位于 WAR 包的 WEB-INF/classes 目录下，这也是 Java 代码编译打包后的目录。也就是说，这类资源文件在打包过后位于应用程序的 classpath 中。Web 项目中还有另外一类资源文件，默认它们的源码位于 src/main/webapp/目录，经打包后位于 WAR 包的根目录。例如，一个 Web 项目的 css 源码文件在 src/main/webapp/css/目录，项目打包后可以在 WAR 包的 css/目录下找到对应的 css 文件。这一类资源文件称做 web 资源文件，它们在打包过后不位于应用程序的 classpath 中。

与一般的资源文件一样，web 资源文件默认不会被过滤。开启一般资源文件的过滤也不会影响到 web 资源文件。

不过有的时候，我们可能希望在构建项目的时候，为不同的客户使用不一样的资源文件（例如客户的 logo 图片不同，或者 css 主题不同）。这时可以在 web 资源文件中使用 Maven 属性，例如用 ${client.logo} 表示客户的 logo 图片，用 ${client.theme} 表示客户的 css 主题。然后使用 profile 分别定义这些 Maven 属性的值，如代码清单 14-18 所示。

代码清单 14-18　针对不同客户 web 资源的 profile

```
<profiles>
  <profile>
    <id>client-a</id>
    <properties>
      <client.logo>a.jpg</client.logo>
      <client.theme>red</client.theme>
    </properties>
  </profile>
  <profile>
    <id>client-b</id>
    <properties>
      <client.logo>b.jpg</client.logo>
      <client.theme>blue</client.theme>
    </properties>
  </profile>
</profiles>
```

最后需要配置 maven-war-plugin 对 src/main/webapp/这一 web 资源目录开启过滤，如代码清单 14-19 所示。

代码清单 14-19 为 web 资源目录 src/main/webapp/开启过滤

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <version>2.1-beta-1</version>
  <configuration>
    <webResources>
      <resource>
        <filtering>true</filtering>
        <directory>src/main/webapp</directory>
        <includes>
            <include>**/*.css</include>
            <include>**/*.js</include>
        </includes>
      </resource>
    </webResources>
  </configuration>
</plugin>
```

代码清单 14-19 中声明了 web 资源目录 src/main/webapp（这也是默认的 web 资源目录），然后配置 filtering 开启过滤，并且使用 includes 指定要过滤的文件，这里是所有 css 和 js 文件。读者可以模仿上述配置添加额外的 web 资源目录，选择是否开启过滤，以及包含或者排除一些该目录下的文件。

配置完成后，可以选择激活某个 profile 进行构建，如 mvn clean install -Pclinet -a，告诉 web 资源文件使用 logo 图片 a.jpg，使用 css 主题 red。

14.6 在 profile 中激活集成测试

很多项目都有大量的单元测试和集成测试，单元测试的粒度较细，运行较快，集成测试粒度较粗，运行比较耗时。在构建项目或者做持续集成的时候，我们都应当尽量运行所有的测试用例，但是当集成测试比较多的时候，高频率地运行它们就会变得不现实。因此有一种更为合理的做法。例如，每次构建时只运行所有的单元测试，因为这不会消耗太多的时间（可能小于 5 分钟），然后以一个相对低一点的频率执行所有集成测试（例如每天 2 次）。

TestNG 中组的概念能够很好地支持单元测试和集成测试的分类标记。例如，可以使用如下的标注表示一个测试方法属于单元测试：

```
@Test(groups={"unit"})
```

然后使用类似的标注表示某个测试方法为集成测试：

```
@Test(groups={"integration"})
```

使用上述方法可以很方便清晰地声明每个测试方法所属的类别。下面的工作就是告诉 Maven 默认只执行所有的单元测试，只在特定的时候才执行集成测试，见代码清单 14-20 所示。

代码清单 14-20　在 profile 中配置执行 TestNG 测试组

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.5</version>
        <configuration>
          <groups>unit</groups>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <profiles>
    <profile>
      <id>full</id>
      <build>
        <plugins>
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.5</version>
            <configuration>
              <groups>unit,integration</groups>
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>
<project>
```

如果读者对 Maven 集成 TestNG 不熟悉，请先回顾 10.7 节。代码清单 14-20 中首先配置了 maven-surefire-plugin 执行 unit 测试组，也就是说默认 Maven 只会执行单元测试。如果想要执行集成测试，就需要激活 full profile，在这个 profile 中配置了 maven-surefire-plugin 执行 unit 和 integration 两个测试组。

有了上述配置，用户就可以根据实际情况配置持续集成服务器。例如，每隔 15 分钟检查源码更新，如有更新则进行一次默认构建，即只包含单元测试。此外，还可以配置一个定时的任务。例如，每天执行两次，执行一个激活 full profile 的构建，以包含所有的集成测试。

从该例中可以看到，profile 不仅可以用来应对不同的构建环境以保持构建的可移植性，还可以用来分离构建的一些较耗时或者耗资源的行为，并给予更合适的构建频率。

14.7 小结

项目构建过程中一个常常需要面对的问题就是不同的平台环境差异，这可能是操作系统的差异、开发平台和测试平台的差异、不同客户之间的差异。

为了应对这些差异，Maven 提供了属性、资源过滤以及 profile 三大特性。Maven 用户可以在 POM 和资源文件中使用 Maven 属性表示那些可能变化的量，通过不同 profile 中的属性值和资源过滤特性为不同环境执行不同的构建。

读者需要区分 Web 项目中一般资源文件和 web 资源文件，前者是通过 maven-resources-plugin 处理的，而后者通过 maven-war-plugin 处理。

本章还详细介绍了 profile，包括各种类别 profile 的特点，以及激活 profile 的多种方式。除此之外，本章还贯穿了几个实际的示例，相信它们能够帮助读者理解什么才是灵活的构建。