今まで知らなかった 5 つの事項: Apache Maven

Maven によってプロジェクトのライフサイクルを管理するためのヒント

Steven Haines
Founder and CEO
GeekCap Inc.

Alex Theedom
Senior Java developer
Consultant

2017年 8月 31日
(初版 2010年 10月 05日)

皆さんはプロファイルについて十分理解しているかもしれませんが、Maven でプロファイルを使用すると、さまざまな環境で特定の動作を実行できることをご存じでしょうか。今回の「今まで知らなかった 5 つの事項」では Maven のビルド機能以外の機能について説明します。そのなかには、プロジェクトのライフサイクルを管理するための基本ツールについての説明も含まれています。この記事で紹介する 5 つのヒントを参考にすれば、生産性を高めることや、Maven で容易にアプリケーションを管理することが可能になります。

このシリーズの他の記事を見る

Java™ 開発者にとって優れたビルド・ツールである Maven を使用すると、プロジェクトのライフサイクルを管理することもできます。ライフサイクル管理ツールとしての Maven は、Ant スタイルのビルド「タスク」の代わりにフェーズをベースに動作します。Maven によって、プロジェクトのライフサイクルのすべてのフェーズを処理することができます (妥当性検証、コード生成、コンパイル、テスト、パッケージ化、統合テスト、動作検証、インストール、デプロイメント、プロジェクト・サイトの作成とデプロイメントなど)。

Maven と従来のビルド・ツールとの違いを理解するためには、JAR ファイルや EAR ファイルのビルド・プロセスについて考える必要があります。Ant を使用する場合には、各成果物を組み立てるための特定のタスクを定義する必要があります。一方 Maven を使用する場合には、ほとんどの作業を Maven が自動的に行ってくれます。単にプロジェクトが JAR ファイルなのか EAR ファイルなのかを Maven に指示し、続いて「パッケージ化」フェーズの処理を指示すればよいのです。あとは Maven が必要なリソースを見つけ、目的のファイルを作成してくれます。

Maven を使い始めるための入門用チュートリアルは数多くあり、その一部を「関連トピック」セクションに挙げてあります。ここで紹介する 5 つのヒントは、その次の段階、つまりMaven を

使ってアプリケーションのライフサイクルを管理する場合のプログラミング・シナリオに役立つものを目指しています。

1. 実行可能な JAR ファイル

この連載について

皆さんは自分が Java プログラミングについて知っていると思うかもしれません。しかし実際には、ほとんどの開発者は Java プラットフォームの表面的な部分しか扱っておらず、当面の作業を完了するために十分なことしか学んでいません。この連載では、Java 技術の専門家が Java プラットフォームのコア機能を深く掘り下げ、非常に厄介なプログラミングの難題を解決するのにも役立つヒントや秘訣を紹介します。

Maven で JAR ファイルをビルドするのは、とても簡単です。単純にプロジェクトを「jar」としてパッケージ化するように指定し、ライフサイクルのパッケージ化フェーズを実行すればよいのです。しかし実行可能な JAR ファイルを定義する際には、少し注意が必要です。実行可能な JAR ファイルを有効に定義するためには、以下のステップに従う必要があります。

1. JAR の MANIFEST.MF ファイルの中で `main` クラスを定義し、この main クラスによって実行可能クラスを定義します (MANIFEST.MF はアプリケーションをパッケージ化する際に Maven によって生成されるファイルです)。
2. プロジェクトが依存するすべてのライブラリーを見つけます。
3. これらのライブラリーを MANIFEST.MF ファイルに含め、アプリケーションのクラスがそれらのライブラリーを見つけられるようにします。

以上をすべて手動で行うことも、あるいはもっと効率的に、`maven-jar-plugin` と `maven-dependency-plugin` という、Maven の 2 つのプラグインを使って行うこともできます。

maven-jar-plugin

`maven-jar-plugin` には多くの機能がありますが、ここではデフォルトの MANIFEST.MF ファイルの内容を変更する機能に焦点を絞ります。POM ファイルのプラグイン・セクションにリスト 1 のコードを追加します。

リスト 1. maven-jar-plugin を使って MANIFEST.MF を変更する

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <configuration>
        <archive>
            <manifest>
                <addClasspath>true</addClasspath>
                <classpathPrefix>lib/</classpathPrefix>
                <mainClass>com.mypackage.MyClass</mainClass>
            </manifest>
        </archive>
    </configuration>
</plugin>
```

Maven のプラグインはすべて、そのプラグインの構成を `<configuration>` 要素を使って公開します。この例では `maven-jar-plugin` プラグインの `archive` 属性が変更されており、具体的にはそのアーカイブ内の `manifest` 属性が変更されています。manifest 属性は、MANIFEST.MF ファイルの内容を制御します。manifest 属性には以下の 3 つの要素が含まれています。

- addClassPath: この要素を true に設定すると、MANIFEST.MF ファイルに Class-Path 要素を追加し、その Class-Path 要素にすべての依存関係を含めるように、maven-jar-plugin に対して指示がされます。
- classpathPrefix: ビルド対象の JAR と同じディレクトリーにすべての依存関係を含める場合には、この要素を省略することができます。それ以外の場合には、classpathPrefix を使って、従属するすべての JAR ファイルの接頭辞を指定します。リスト 1 の classpathPrefix では、アーカイブに対する相対フォルダー lib にすべての依存関係を配置する必要があることが指定されています。
- mainClass: ユーザーが java -jar コマンドを使って JAR ファイルを実行する場合には、この要素を使って実行対象クラスの名前を指定します。

maven-dependency-plugin

これらの 3 つの要素によって MANIFEST.MF を構成したら、次のステップとして、実際にすべての依存関係を lib フォルダーにコピーします。そのためには maven-dependency-plugin を使用します (リスト 2)。

リスト 2. maven-dependency-plugin を使って依存関係を lib にコピーする

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <executions>
        <execution>
            <id>copy</id>
            <phase>install</phase>
            <goals>
                <goal>copy-dependencies</goal>
            </goals>
            <configuration>
                <outputDirectory>
                  ${project.build.directory}/lib
                </outputDirectory>
            </configuration>
        </execution>
    </executions>
</plugin>
```

maven-dependency-plugin には、指定されたディレクトリーに依存関係をコピーする、copy-dependencies というゴールがあります。この例では、build ディレクトリーの下にある lib ディレクトリー (project-home/target/lib) に依存関係をコピーしています。

依存関係を用意でき、MANIFEST.MF の変更が終わると、下記の単純なコマンドを使ってアプリケーションを起動することができます。

```
java -jar jarfilename.jar
```

2. MANIFEST.MF をカスタマイズする

maven-jar-plugin によって MANIFEST.MF ファイルの共通部分を変更することはできますが、もっとカスタマイズした MANIFEST.MF が必要な場合もあります。その場合には、以下の 2 つのステップで対処することができます。

1. カスタムの構成をすべて、MANIFEST.MF ファイルという「テンプレート」の中で定義します。
2. その MANIFEST.MF を使うように `maven-jar-plugin` を構成し、Maven のカスタマイズによって強化します。

例えば、Java エージェントを含む JAR ファイルを考えてみてください。Java エージェントを実行するためには、その Java エージェントによって `Premain-Class` とアクセス権を定義する必要があります。リスト 3 は、そうした MANIFEST.MF ファイルの内容を示しています。

リスト 3. カスタムの MANIFEST.MF ファイルの中で Premain-Class を定義する

```
Manifest-Version: 1.0
Premain-Class: com.geekcap.openapm.jvm.agent.Agent
Can-Redefine-Classes: true
Can-Retransform-Classes: true
Can-Set-Native-Method-Prefix: true
```

リスト 3 では、クラスの再定義と再変換を行うための権限を `Premain-Classcom.geekcap.openapm.jvm.agent.Agent` に与えるように指定しています。次に、`maven-jar-plugin` を更新し、この MANIFEST.MF ファイルを含めます (リスト 4)。

リスト 4. Premain-Class を含める

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <configuration>
                    <archive>
                        <manifestFile>
                          src/main/resources/META-INF/MANIFEST.MF
                        </manifestFile>
                        <manifest>
                            <addClasspath>true</addClasspath>
                            <classpathPrefix>lib/</classpathPrefix>
                            <mainClass>
                              com.geekcap.openapm.ui.PerformanceAnalyzer
                            </mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
```

Maven 3

Maven 2 は、最もよく知られ、最もよく使われているオープンソースの Java ライフサイクル管理ツールの 1 つとして、その地位を確立しています。2010年 9月 5日にアルファ 5 にプロモートされた Maven 3 では、長年 Maven に熱望されていた変更が実現されています。Maven 3 ではどんな点が新しいのかについては、「関連トピック」セクションを参照してください。

これは興味深い例です。この例では、JAR ファイルを Java エージェントとして動作させる `Premain-Class` を定義すると同時に、実行可能な JAR ファイルとして Java エージェントを実行させる `mainClass` もあるからです。この具体例では、`OpenAPM` (私が作成したコード・トレース・ツール) を使ってコード・トレースを定義しています。コード・トレースは、この Java エージェントとユーザー・インターフェースによって記録され、記録されたトレースの分析が行われます。要するに、この例は明示的なマニフェスト・ファイルと動的な変更との組み合わせによる強力さを示しているのです。

3. 依存関係ツリー

Maven で最も便利な機能の 1 つとして、依存関係の管理機能があります。アプリケーションが依存しているライブラリーを単純に指定すると、Maven は (ローカルまたは中央のリポジトリーで) それらのライブラリーを見つけてダウンロードし、それらのライブラリーを使ってコードをコンパイルします。

場合によると、特定の依存関係の起源について知る必要が出てくるかもしれません (例えば、ビルドの中にある JAR ファイルの別バージョンで、他とは互換性のない JAR ファイルを見つけるようなことになった場合)。そうした場合、あるバージョンの JAR ファイルがビルドに含まれないようにする必要がありますが、その前にまず、その JAR が関わる依存関係を見つける必要があります。

以下のコマンドを理解していると、驚くほど容易に依存関係を見つけることができます。

```
mvn dependency:tree
```

`dependency:tree` 引数を指定すると、直接の依存関係がすべて表示され、さらにその下位の依存関係もすべて表示されます (そして、さらにその下位の依存関係も表示され・・・といった具合に続きます)。例えばリスト 5 は、クライアント・ライブラリーから依存関係の 1 つが必要としている部分を抜粋したものです。

リスト 5. Maven の依存関係ツリー

```
[INFO] ------------------------------------------------------------------------
[INFO] Building Client library for communicating with the LDE
[INFO]    task-segment: [dependency:tree]
[INFO] ------------------------------------------------------------------------
[INFO] [dependency:tree {execution: default-cli}]
[INFO] com.lmt.pos:sis-client:jar:2.1.14
[INFO] +- org.codehaus.woodstox:woodstox-core-lgpl:jar:4.0.7:compile
[INFO] |  \- org.codehaus.woodstox:stax2-api:jar:3.0.1:compile
[INFO] +- org.easymock:easymockclassextension:jar:2.5.2:test
[INFO] |  +- cglib:cglib-nodep:jar:2.2:test
[INFO] |  \- org.objenesis:objenesis:jar:1.2:test
```

リスト 5 を見るとわかるように、`sis-client` プロジェクトには `woodstox-core-lgpl` ライブラリーと `easymockclassextension` ライブラリーが必要です。そして `easymockclassextension` ライブラリーには `cglib-nodep` ライブラリーと `objenesis` ライブラリーが必要です。`objenesis` に問題がある場合 (例えば 1.2 と 1.3 という 2 つのバージョンがあるなど)、この依存関係ツリーには、1.2 の成果物が `easymockclassextension` ライブラリーによって間接的にインポートされていることが示されています。

`dependency:tree` 引数を利用することで、私は問題のあるビルドのデバッグに要する時間を大幅に節約できた経験があります。この引数が皆さんにも役立つことを祈っています。

4. 環境に応じてビルドする

大規模なプロジェクトでは、ほとんどの場合、最低限のコアとなる一連の環境があり、それらの環境は、開発、品質管理 (QA)、統合、本番、などに関連するタスクで構成されています。こうし

た環境すべてを管理する場合に難題となるのは、ビルドの構成をすることです。ビルドをするには、適切なデータベースに接続し、適切な一連のスクリプトを実行し、それぞれの環境にしかるべき成果物をすべてデプロイする必要があるからです。Maven のプロファイルを使用すると、それぞれの環境に対する明示的な指示を作成しなくても、ビルドの構成をするためのすべてを行うことができます。

重要な点は、環境のプロファイルとタスク指向のプロファイルとを組み合わせる点にあります。それぞれの環境プロファイルでは、その環境の具体的な場所、スクリプト、サーバーを定義します。つまり私の `pom.xml` ファイルの場合であれば、タスク指向のビルドをリスト 6 のように定義します。

リスト 6. デプロイメント・ビルド

```
<profiles>
    <profile>
        <id>deploywar</id>
        <build>
            <plugins>
                <plugin>
                    <groupId>net.fpic</groupId>
                    <artifactId>tomcat-deployer-plugin</artifactId>
                    <version>1.0-SNAPSHOT</version>
                    <executions>
                        <execution>
                            <id>pos</id>
                            <phase>install</phase>
                            <goals>
                                <goal>deploy</goal>
                            </goals>
                            <configuration>
                                <host>${deploymentManagerRestHost}</host>
                                <port>${deploymentManagerRestPort}</port>
                                <username>${deploymentManagerRestUsername}</username>
                                <password>${deploymentManagerRestPassword}</password>
                                <artifactSource>
                                  address/target/addressservice.war
                                </artifactSource>
                            </configuration>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
    </profile>
</profiles>
```

このビルドが実行するのは `tomcat-deployer-plugin` です。このプラグインは、特定のホスト、ポート、特定のユーザー名とパスワードの資格情報に接続するように構成されています。これらの情報はすべて、`${deploymentmanagerRestHost}` などの変数を使って定義されています。これらの変数はプロファイルの中で、環境ごとに定義されています (リスト 7)。

リスト 7. 環境プロファイル

```
    <!-- Defines the development deployment information -->
    <profile>
        <id>dev</id>
        <activation>
            <property>
```

```
                    <name>env</name>
                    <value>dev</value>
                </property>
            </activation>
            <properties>
                <deploymentManagerRestHost>10.50.50.52</deploymentManagerRestHost>
                <deploymentManagerRestPort>58090</deploymentManagerRestPort>
                <deploymentManagerRestUsername>myusername</deploymentManagerRestUsername>
                <deploymentManagerRestPassword>mypassword</deploymentManagerRestPassword>
            </properties>
        </profile>

        <!-- Defines the QA deployment information -->
        <profile>
            <id>qa</id>
            <activation>
                <property>
                    <name>env</name>
                    <value>qa</value>
                </property>
            </activation>
            <properties>
                <deploymentManagerRestHost>10.50.50.50</deploymentManagerRestHost>
                <deploymentManagerRestPort>58090</deploymentManagerRestPort>
                <deploymentManagerRestUsername>
                  myotherusername
                </deploymentManagerRestUsername>
                <deploymentManagerRestPassword>
                  myotherpassword
                </deploymentManagerRestPassword>
            </properties>
        </profile>
```

Maven のプロファイルをデプロイする

リスト 7 のプロファイルの中では 2 つのプロファイルを定義し、プロファイル名として設定された値に基づいて、該当するプロファイルが有効になるようにしています。選択されたプロファイルが `dev` であれば、開発用のデプロイメント情報が使用され、選択されたプロファイルが `qa` であれば、QA 用のデプロイメント情報が使用されるといった具合です。

下記は開発環境にデプロイするためのコマンドです。

```
mvn -Pdeploywar -Denv=dev clean install
```

`-Pdev` フラグによって、デプロイメント構成を有効にするように Maven に指示します。`-Pdev` ではなく `-Pqa` を渡したとしたら、QA 用の構成が有効になります。

5. カスタムの Maven プラグイン

Maven には自由に使用できるビルド済みのプラグインが何十とありますが、場合によるとカスタムのプラグインが必要になることがあります。カスタムの Maven プラグインは以下の手順で簡単に作成することができます。

1. 新しいプロジェクトを作成し、POM を `maven-plugin` としてパッケージ化するように設定します。
2. 公開されたプラグインのゴールを定義するように、`maven-plugin-plugin` の呼び出しを含めます。

3. Maven プラグインの `mojo` クラス (`AbstractMojo` を継承するクラス) を作成します。
4. このクラスにアノテーションを付けて、構成パラメーターとして公開するゴールと変数を定義します。
5. `execute()` メソッドを実装します。プラグインを呼び出すと、この `execute()` メソッドが呼び出されます。

例えばリスト 8 は、Tomcat をデプロイするために設計されたカスタム・プラグインの関連部分を示しています。

リスト 8. TomcatDeployerMojo.java

```
package net.fpic.maven.plugins;

import java.io.File;
import java.util.StringTokenizer;

import net.fpic.tomcatservice64.TomcatDeploymentServerClient;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;

import com.javasrc.server.embedded.CommandRequest;
import com.javasrc.server.embedded.CommandResponse;
import com.javasrc.server.embedded.credentials.Credentials;
import com.javasrc.server.embedded.credentials.UsernamePasswordCredentials;
import com.javasrc.util.FileUtils;

/**
 * Goal that deploys a web application to Tomcat
 *
 * @goal deploy
 * @phase install
 */
public class TomcatDeployerMojo extends AbstractMojo
{
 /**
  * The host name or IP address of the deployment server
  *
  * @parameter alias="host" expression="${deploy.host}" @required
  */
 private String serverHost;

 /**
  * The port of the deployment server
  *
  * @parameter alias="port" expression="${deploy.port}" default-value="58020"
  */
 private String serverPort;

 /**
  * The username to connect to the deployment manager (if omitted then the plugin
  * attempts to deploy the application to the server without credentials)
  *
  * @parameter alias="username" expression="${deploy.username}"
  */
 private String username;

 /**
  * The password for the specified username
  *
  * @parameter alias="password" expression="${deploy.password}"
  */
 private String password;
```

```
 /**
  * The name of the source artifact to deploy, such as target/pos.war
  *
  * @parameter alias="artifactSource" expression=${deploy.artifactSource}"
  * @required
  */
 private String artifactSource;

 /**
  * The destination name of the artifact to deploy, such as ROOT.war.
  * If not present then the
  * artifact source name is used (without pathing information)
  *
  * @parameter alias="artifactDestination"
  *   expression=${deploy.artifactDestination}"
  */
 private String artifactDestination;

    public void execute() throws MojoExecutionException
    {
     getLog().info( "Server Host: " + serverHost +
               ", Server Port: " + serverPort +
               ", Artifact Source: " + artifactSource +
               ", Artifact Destination: " + artifactDestination );

     // Validate our fields
     if( serverHost == null )
     {
      throw new MojoExecutionException(
        "No deployment host specified, deployment is not possible" );
     }
     if( artifactSource == null )
     {
      throw new MojoExecutionException(
        "No source artifact is specified, deployment is not possible" );
     }

         ...
   }
}
```

クラスごとに、`@Mojo` アノテーションの値 `name` でこの MOJO が実行するゴールを指定し、そのゴールを実行するフェーズを lifecyclePhase で指定します。公開される各プロパティーには、エイリアスと式を指定するための `@parameter` アノテーションがあります。エイリアスはパラメーターが実行される際に使用され、式は実際の値を含むシステム・プロパティーにマッピングするために使用されます。プロパティーに値 `required=true` が設定されている場合は、そのプロパティーは必須プロパティーです。`defaultValue` が設定されているとしたら、値が指定されない場合はその `defaultValue` が使われます。`execute()` メソッドの中で `getLog()` を呼び出すと、Maven のロガーを利用することができます。ロギング・レベルによりますが、Maven のロガーによって、指定されたメッセージを標準出力デバイスに出力することができます。プラグインが失敗すると、`MojoExecutionException` がスローされ、ビルドが失敗します。

まとめ

Maven を単なるビルドに使用することもできますが、Maven の最も優れた点は、プロジェクトのライフサイクル管理ツールとして使用できることです。この記事では、Maven をより有効に使うための、あまり知られていない 5 つの機能を紹介しました。さらに Maven について学ぶためには、「**関連トピック**」セクションを参照してください。

著者について

Steven Haines

Steven Haines は ioko の技術アーキテクトであり、GeekCap Inc. の設立者でもあります。彼は Java プログラミングとパフォーマンス分析に関する本を 3 冊執筆しており、また数百本の記事や 10 本を超えるホワイトペーパーも執筆しています。また彼は JBoss World や STPCon などの業界のカンファレンスでの講演経験もあり、以前はカリフォルニア大学アーバイン校 (University of California, Irvine) と Learning Tree University で Java プログラミングを教えていました。彼はフロリダ州オーランドの近郊に住んでいます。

Alex Theedom

May 2017