

今まで知らなかった 5 つの事項: Apache Maven のプラグイン

最新の Maven プラグインを習得するためのアドバイス

Alex Theedom

Senior Java developer
Consultant

2018年 3月 22日

Maven を最大限に利用しきれていませんか？それは誰にとっても同じことです！プラグインのゴールが Maven のライフサイクルにどのように対応するのかを学んで、GitHub の Maven Site、Codehaus の Maven Cargo を含め、作成するプロジェクトを次のレベルに引き上げる 4 つのプラグインを導入してください。

Maven をご存知ですか？

Java 開発者にとって、Maven が主要な依存関係管理ツール兼ビルド・ツールとなるのには、十分な理由があります！それは、Maven はソフトウェアのビルド・サイクルを標準化するために、プロジェクトの構造を明確に記述し、プロジェクトをアプリケーションとしてデプロイし、そのアプリケーションを他のプロジェクトと共有するからです。

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が。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 テクノロジーを徹底的に追及する著者たちが 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 Java プログラミングの難題の解決にも役立つ、アドバイスとコツを紹介します。

Maven のあらゆる機能は、Maven で使用する堅牢なプラグイン一式によって実現されます。Maven では、プラグインのそれぞれにゴールがあり、その内部は単なる Java メソッドとなっています。ゴールが実行するのは、プロジェクトのコンパイル、プロジェクトのパッケージ化、ローカルまたはリモート・サーバーへのプロジェクトのデプロイなどのビルド・タスクです。これらのアクティビティーはビルド・ライフサイクルの各フェーズに完璧に対応しています。

Maven にパッケージ化された付属のビルド・プラグインは、デフォルトが事前に構成されていて、すぐに使用できる状態になっています。設定より規約 (Convention-over-configuration) の原則により、特定のタスクの複雑さに応じて構成がスケーリングされるようになっています。つまり、ほとんどのタスクに必要な構成は必要最小限です。

Maven プラグインの動作はカスタマイズすることもできます。Maven の `<configuration>` 要素を使用すれば、簡単にプラグインのデフォルトをオーバーライドして新しい値を設定できます。オーバーライドされるデフォルト値のほとんどは、間違いなく、[コンパイラー・プラグイン](#)の `<source>` と `<target>` の値です。

その証拠に、正しい JVM バージョンを設定するために、リスト 1 の XML を何度 POM に追加したかを考えてください。

リスト 1. コンパイラー・プラグイン内の Java バージョン構成

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.6.1</version>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
  </configuration>
</plugin>
```

Maven ライフサイクルを実行する

プラグインのゴールのそれぞれは、Maven [ライフサイクルの各フェーズ](#)に対応します。mvn compile コマンドを発行すると、mvn ユーティリティーによって、compile ライフサイクル・フェーズにバインドされたゴールのすべてが呼び出されます。compiler プラグインの compile ゴールも、このライフサイクル・フェーズにバインドされているゴールの 1 つです。

フェーズとゴールの関係は 1 対多です。したがって、複数のプラグインのゴールを同じフェーズにバインドすることで、関連するタスクの集合を形成することができます。フェーズは階層型でもあり、あるフェーズが実行されると、[そのフェーズに先行するすべてのフェーズ](#)が実行されます。

この場合、mvn compile コマンドを発行すると、validate フェーズ ([Maven のデフォルトのビルド・ライフサイクルによる最初のフェーズ](#)) が開始され、このフェーズにバインドされたすべてのゴールが呼び出されます。

[Maven](#) と [Google Code](#) の Web サイトで、Maven プラグインのリストが保守されています。便利で時間の節約となる幅広い機能を提供するこれらのプラグインは、ビルド・プロセスに組み込むことができます。これから、私がとりわけ有用だと思ったプラグインをいくつか取り上げて、そのプラグインを最大限に利用する方法を説明します。

1. JAR または WAR にデジタル署名を付ける

下のボタンをクリックして、この例で使用するコードをダウンロードしてください。ダウンロード・ファイルには、単純なアプリケーションと、Jarsigner プラグインが構成された POM ファイルが含まれています。

コードを入手する

JAR や WAR にデジタル署名を付けるのは、重要なタスクです。特に、アプリケーションを配布する場合には、その重要性がさらに増します。ありがたいことに、Java プラットフォームには

jarsigner という名前のアーカイブ署名ユーティリティーが用意されています。このコマンド・ライン・ツールにアーカイブの場所と各種のパラメーター (暗号鍵が格納されているキー・ストアなど) を渡すと、そのアーカイブに署名が付けられます。

jarsigner は、JDK インストール済み環境の `<%JAVA_HOME%/bin #####` 一内にあります。リスト 2 に、このユーティリティーを使用して JAR に署名を付ける方法を示します。

リスト 2. jarsigner ユーティリティーを使用してアーカイブに署名を付ける

```
jarsigner
-keystore /path/to/keystore.jks
-storepass <password>
-keypass <key password>
YourArchive.jar
alias
```

jarsigner は追加の設定を行わずに簡単に使用できるコマンド・ライン・ツールです。けれども、署名プロセスを自動化して、ビルド・サイクルの `package` フェーズでアーカイブに署名が付けられるようにすると便利だと思いませんか？幸い、それを実現する方法があります。それは、**jarsigner** ユーティリティーをラップする、[Maven Jarsigner プラグイン](#)です。このプラグインの `sign` ゴールをビルドの `package` フェーズにバインドするだけで、自動的に署名が付けられるようになります。

まず始めに、キー・ストアが必要です。既存のキー・ストアがない場合は、Java プラットフォームの **keytool** ユーティリティーを使って作成できます。このユーティリティーは、JDK インストール済み環境の `<%JAVA_HOME%/bin` ディレクトリー内にあります。

キー・ストアを作成する

キー・ストアを作成するには、作成するキー・ストアを保管する場所に移動して、リスト 3 のコマンドを実行します。このコマンドに含まれている `KeyAlias` の部分は、必ず適切な値 (ドメイン名など) で置き換えてください。このリンク先の[公式な Oracle の資料](#)に、このツールが受け入れる構成オプションの詳細な説明が記載されています。

リスト 3. jarsigner ユーティリティー用のキー・ストアを作成する

```
keytool -genkey -alias <KeyAlias> -keyalg RSA -keystore keystore.jks -keysize 2048
```

キー・ストアを作成するプロセスでは、自分と自分が所属する組織に関する一連の質問に答える必要があります。また、セキュアなパスワードも指定しなければなりません。そのパスワードは、次のステップで **Maven Jarsigner プラグイン**を構成するときに必要になります。

Jarsigner をビルド・プロセスに追加する

次は、**Maven Jarsigner プラグイン**をビルド・ライフサイクルに追加してから、`package` フェーズで生成される JAR にデジタル署名を付けるようにプラグインを構成します。`package` フェーズにプラグインを関連付けるのが妥当なわけは、このフェーズでは JAR がすでに作成されて、通常はターゲット・ディレクトリーであるプロジェクトのデフォルト出力ディレクトリー内に格納されているからです。POM ファイルの `<plugins>` セクションに、リスト 4 に記載するコードを追加してください。

リスト 4. Jarsigner プラグインを追加する

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jarsigner-plugin</artifactId>
  <version>1.4</version>
</plugin>
```

JAR に署名を付けるのは、それが作成された後でなければならないため、(sign ゴール内にラップされた) 署名プロセスは package ライフサイクル・フェーズに関連付ける必要があります。それには、リスト 5 のコードをプラグインに追加します。

リスト 5. sign ゴールを package フェーズに関連付ける

```
<executions>
  <execution>
    <id>sign</id>
    <phase>package</phase>
    <goals>
      <goal>sign</goal>
    </goals>
  </execution>
</executions>
```

sign ゴールが JAR にデジタル署名を付けられるように、このゴールに対して、いくつかの構成の詳細を指定しなければなりません。具体的には、<configuration> 要素で囲んだ部分に、キー・ストアの場所、使用する資格情報の別名、ストアのパスワード、資格情報のパスワードを追加する必要があります (リスト 6 を参照)。

リスト 6. セキュリティー資格情報を指定する

```
<configuration>
  <keystore>/location/of/keystore.jks</keystore>
  <alias>KeyAlias</alias>
  <storepass>password</storepass>
  <keypass>password</keypass>
</configuration>
```

この構成は、JAR 署名機能を有効にするために最小限必要な構成です。このプラグインでは、他にもかなり幅広い構成オプションを選択できるようになっていることに注意してください。

最後のステップでは、JAR をビルドして、適切な署名が付けられていることを確認します。コマンド・ラインから、`mvn clean package` を入力して package ゴールを実行します。コンソールの出力で、JAR がビルドされた後、署名が付けられていることを確認できるはずです。図 1 に、出力例を示します。

図 1. アーカイブの署名付けを示す出力

```
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ maven-sampler ---
[INFO] Building jar: C:\Alex\projects\Maven Sampler\target\maven-sampler-1.0.jar
[INFO]
[INFO] --- maven-jarsigner-plugin:1.4:sign (sign) @ maven-sampler ---
[INFO] 1 archive(s) processed
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

JAR はターゲット出力ディレクトリー内に作成されます。ここが、Jarsigner プラグインが JAR を探す場所です。プラグインは指定された資格情報で JAR に署名を付けた後、さらに 2 つのファイルを META-INF ディレクトリーに追加します。それは、ファイル拡張子が .SF の署名ファイルと、ファイル拡張子 .RSA の署名ブロック・ファイルです。

これで、デジタル署名付き JAR を配布できる状態になりました。

デジタル署名を検証する

JAR を配布する前に、適切に署名が付けられていることを検証しましょう。署名の検証は、Maven Jarsigner の `verify` ゴールを使用するか、コマンド・ラインから `jarsigner` ツールを使用することができます。リスト 7 では、コマンド・ライン・ツールを使って、検証するアーカイブ・ファイルを渡しています。

リスト 7. jarsigner ユーティリティーを使用して署名付き JAR を検証する

```
jarsigner -verify target/YourJavaArchive.jar
```

検証に成功すると、「jar verified (JAR が検証されました)」というメッセージが表示されます。

WAR ファイルに署名を付ける

Jarsigner が用意している奥の手は、JAR の署名付けだけではありません。このプラグインは、任意の Java アーカイブ・ファイルにも署名を付けることができます。これには WAR ファイルも含まれます。この仕組みを確かめるために、POM のパッケージング・タイプを JAR から WAR に変更し、[Maven WAR プラグイン](#)を追加して(リスト 8 を参照)、コマンド `mvn clean package` を実行します。

リスト 8. Maven WAR プラグインを追加する

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <version>3.1.0</version>
</plugin>
```

ターゲット・ディレクトリーを確認すると、署名付きの WAR ファイルがあるはずです。

複数のアーカイブに署名を付ける

Jarsigner では、署名を付けるアーカイブが複数ある場合にも対処できます。リスト 9 のコードでは、署名を付ける JAR がある場所を `target/all` として指定し、このディレクトリー内にあるすべての JAR に署名を付けます。

リスト 9. アーカイブ・ディレクトリーを指定する

```
<archiveDirectory>target/all</archiveDirectory>
```

署名を付けるアーカイブを細かく制御する必要がある場合は、リスト 10 に示すように、ワイルドカードを使用して、包含するファイルと除外するファイルを明示的に指定することができます。

リスト 10. アーカイブを選択して包含および除外する

```
<includes>
  <include>*1.0.jar</include>
</includes>
<excludes>
  <exclude>*SNAPSHOT.jar</exclude>
</excludes>
```

2. Maven Cargo を使用してセキュアな Web アプリケーションをデプロイする

下のボタンをクリックして、この例で使用するコードをダウンロードしてください。ダウンロード・ファイルには、単純なアプリケーションと、Cargo プラグインが構成された POM ファイルが含まれています。

コードを入手する

Codehaus の [Cargo プラグイン](#) は、Cargo API のラッパーです。Cargo API は、アプリケーションを構成、起動、停止することと、[サポート対象](#)のコンテナにデプロイするためことを目的としているだけでなく、Java EE の解析、作成、マージにも対応します。

Cargo で特に興味深い特徴となっているのは、コンテナにとらわれない形でサーバー・プロパティーを指定できることです。そのようなプロパティーには、ポート、リモート・セキュリティ資格情報、そして (この記事に最も関連性のある) デプロイするアプリケーションのユーザー・ログイン情報があります。このプラグインは、とりわけよく使われている [14 種類](#)のサーバーを、それぞれの最新エディションまでサポートしています。

Cargo プラグインは、Maven ライフサイクルの `package` フェーズに関連付けると、最も効果を発揮します。また、`run` ゴールを直接参照することで、単独で実行することもできます。単独で実行する場合は、プロジェクトがすでにパッケージ化されて、デプロイできる状態であることが前提となります。

まずは、リスト 11 に示すように、Cargo の `run` ゴールを `package` フェーズに関連付けるところから始めましょう。

リスト 11. Cargo の run ゴールを package フェーズに関連付ける

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <version>1.6.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>run</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Maven との統合が完了したら、次は Cargo によるアプリケーションのデプロイ先になるコンテナを構成する必要があります。Cargo はさまざまなコンテナ・シナリオをサポートします。これには、すでに稼働中のローカル・サーバーまたはリモート・サーバーにコンテナをデプロイするというシナリオも含まれます (必要に応じて、サーバーの起動にも対応します)。あるいは、サポート対象の 14 種類のサーバーのいずれかをダウンロード、インストール、デプロイし、起動するように Cargo を構成することもできます。

これから説明する例では、[IBM WebSphere Liberty](#) を使用します。このサーバーはすでにダウンロードして、`servers` という名前のディレクトリーに抽出しておきました。

コンテナ構成には、少なくともコンテナ ID と、インストール済みサーバーのホーム・ロケーションを指定する必要があります。リスト 12 に、`servers/wlp` ディレクトリー内に格納されている Liberty サーバーの構成を記載します。

リスト 12. コンテナ構成

```
<configuration>
  <container>
    <containerId>liberty</containerId>
    <home>\\path\\to\\servers\\wlp</home>
  </container>
</configuration>
```

コンテナ ZIP をダウンロードする

別の選択肢として、IBM Web 上に用意されている Liberty ZIP の URL を指定して、Cargo プラグインにサーバーをダウンロードしてインストールさせるという方法もあります。リスト 13 に、その場合のコードを記載します。

リスト 13. Web からコンテナをインストールする

```
<container>
  <containerId>liberty</containerId>
  <zipUrlInstaller>
    <url>
      https://public.dhe.ibm.com/ibmdl/export/pub/
      software/websphere/wasdev/downloads/wlp/17.0.0.2/
      wlp-webProfile7-17.0.0.2.zip
    </url>
    <downloadDir>/path/to/downloadDir</downloadDir>
    <extractDir>/path/to/extractDir</extractDir>
  </zipUrlInstaller>
</container>
```

初めて `run` ゴールを実行すると、ZIP ファイルがダウンロードされて解凍されます。2 回目以降は、`run` ゴールを実行すると、Maven は該当するファイルがすでにダウンロードされてインストールされていることを検出し、そのタスクをスキップします。

これで、必要最小限の構成が完了しました。アプリケーションをサーバーにデプロイすることで、この構成をテストできます。`package` フェーズを呼び出して (`mvn package`)、プロセスを起動してください。これによってアプリケーションのコンパイル、パッケージ化、サーバーへのデプロイが行われた後は、URL: `http://localhost:8080/{artifactid}/index.html` でアプリケーションにアクセスできるようになります。

コンテキスト・ルートをカスタマイズする

デフォルトでは、Web アプリケーションのコンテキスト・ルートの名前には `artifactid` が使われますが、場合によっては、カスタム・ルートを指定しなければならないこともあります。リスト 14 では、コンテキスト・ルートを `home` として定義しています。

リスト 14. コンテキスト・ルートをカスタマイズする

```
<deployables>
  <deployable>
    <properties>
      <context>home</context>
    </properties>
  </deployable>
</deployables>
```

この場合、アプリケーションがデプロイされると、その Web アプリケーションには `localhost:8080/home/index.html` でアクセスできるようになります。

ユーザー・ログインを指定する

サーバーをダウンロードおよびインストールして、アプリケーションをカスタム・コンテキスト・ルートにデプロイするという、かなり堅牢なデプロイメント・プロセスになりました。けれどもこれは、Cargo プラグインが持つ力の発端でしかありません。

構成プロパティを定義する段階になると、Cargo はその実力を発揮してきます。前述のとおり、コンテナのこれらの**構成プロパティ**は、ポート、プロトコル、そして最も興味深いことに、ユーザー・ログインの詳細です。

ユーザーをサンプル・アプリケーションに追加して、サイトにログインさせましょう。ここでは単純にするために、基本認証方式を使用します。基本認証方式の場合、アプリケーションの `web.xml` 内で必要となる構成は最小限で済みます。リスト 15 に、`user-role` と、`WelcomeServlet` Web リソースに対する制限の指定を示します。

リスト 15. `user-role` を宣言する

```
<web-app version="3.1"...>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Welcome Servlet</web-resource-name>
      <url-pattern>/WelcomeServlet/*</url-pattern>
      <http-method>GET</http-method>
    </web-resource-collection>

    <auth-constraint>
      <role-name>user-role</role-name>
    </auth-constraint>
  </security-constraint>

  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>file</realm-name>
  </login-config>

  <security-role>
    <role-name>user-role</role-name>
  </security-role>

</web-app>
```

`user-role` に対応するユーザーを作成するように Cargo を構成できます。この構成は、デプロイメント時に臨機応変に行います。リスト 16 に、`user-role` に対して定義されているユーザー名とパスワードを示します。ユーザー名は `user`、パスワードは `user123!` です。

リスト 16. ユーザー名とパスワードを定義する

```
<configuration>
  <properties>
    <cargo.servlet.users>
      user:user1234!:user-role
    </cargo.servlet.users>
  </properties>
</configuration>
```

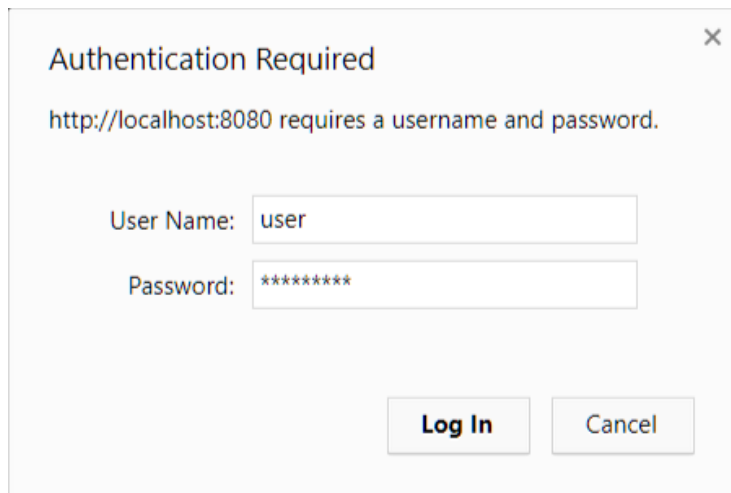
`<cargo.servlet.users>` プロパティのフォーマットは、`username:password:role` です。このセット単位をパイプ記号で区切ることで、単一のプロパティ内で同じロールまたは異なるロールに対して複数のユーザーを宣言できます。

```
username1:password1:role11,...,role1N|username2:password2:role21,...,role2N|...
```

ユーザー名とパスワードをユーザー・ロールに割り当てたら、次は、アプリケーションをパッケージ化してデプロイします。たった 1 つのコマンド `mvn clean package` を実行するだけで、Web アプリケーションがコンパイルおよびパッケージ化されて、Liberty コンテナにデプロイされます。さらに、サーバーも起動されます。

URL `http://localhost:8080/home/welcomeServlet` にアクセスすると、図 2 に示すログイン・ボックスが表示されます。Cargo プラグインで構成したユーザー名とパスワードを入力し、「Log In (ログイン)」ボタンをクリックしてアプリケーションにログインしてください。

図 2. ログイン・ダイアログボックス

A screenshot of a web browser's authentication dialog box. The title bar says "Authentication Required" with a close button (X) on the right. The main text says "http://localhost:8080 requires a username and password." Below this, there are two input fields: "User Name:" with the text "user" entered, and "Password:" with "*****" entered. At the bottom, there are two buttons: "Log In" and "Cancel".

3. カスタム JAR をローカル・リポジトリにインストールする

プロジェクトに JAR を含める従来の方法は、プロジェクト内にディレクトリーを作成し、そのディレクトリーがクラス・パス上で指定されるようにすることです。Maven を使用する場合、この方法に従う必要はありません。Maven では、プロジェクトと JAR を同じ場所に配置する必要はなく、JAR は独立したローカル・レポジトリ内に格納されます。したがって、プロジェクトで、これらの JAR に対する依存関係を指定する必要があります。

ローカル・リポジトリには、[Maven の中央リポジトリ](#)からダウンロードされた依存関係が取り込まれます。中央リポジトリは Maven のマザーシップであり、評判の高い、よく使われている JAR が何千も格納されています。けれども、必要な JAR が Maven の中央リポジトリ内になかったとしたら、どうでしょう？ その場合、もちろん従来のソリューションに頼って、JAR を直接プロジェクトに含めることはできますが、そうすると、Maven の依存性管理のメリットが生かされないこととなります。

従来のソリューションに頼るのではなく、そのような場合には、`maven install #####`を使用して、JAR をローカル Maven リポジトリにインストールすることができます。それにはまず、この[プラグインをダウンロード](#)し、このプラグインに含まれる `install-file` ゴールを実行します。これから説明する例は、前の例とは少々異なり、ゴールを Maven ライフサイクルのフェーズに関連付けることはしません。この場合は、Maven コマンド・ライン・ツール内でゴールを指定します。

`install-file` ゴールを実行するには、5 つのパラメーターを使用して、JAR ファイルの場所、`group`、`artifactid`、バージョン番号、パッケージ・タイプを指定する必要があります。リスト 17 に、これらのパラメーターからなる Maven コマンドの構造を示します。

リスト 17. JAR をローカル Maven リポジトリにインストールする

```
mvn install:install-file
-Dfile=PATH/TO/YOUR-JAR.jar
-DgroupId=GROUP-ID
-DartifactId=ARTIFACT-ID
-Dversion=VERSION
-Dpackaging=jar
```

インストールする JAR には、上記のコマンドに示されているパラメーターの値を指定する必要があります。指定する値は任意で選べます。コマンドの実行が完了した後は、新しいファイルがローカル Maven リポジトリ内に格納されていることを確認できます。リポジトリの `/.m2/repository` ディレクトリに移動して、`GROUP-ID` という名前のフォルダーを見つけてください。groupId の値が `com.readlearncode` のような逆ドメイン名の場合は、`com` ディレクトリを探してから、`readlearncode` ディレクトリに移動すればよいだけです。そのディレクトリで、成果物のバージョン番号が付いたフォルダーを見つけて、そのフォルダー内にインストールされている JAR を確認してください。JAR の場所は、`/.m2/repository/com/readlearncode/4.0/MyJar.jar` のようになります。

これで、すべてのプロジェクトからこの JAR を使用できるようになりました。この JAR を依存関係として POM に追加するには、通常の方法を使えます (リスト 18 を参照)。

リスト 18. JAR を依存関係として指定する

```
<dependency>
  <groupId>GROUP-ID</groupId>
  <artifactId>ARTIFACT-ID</artifactId>
  <version>VERSION</version>
</dependency>
```

4. GitHub ベースの Maven リポジトリを作成する

下のボタンをクリックして、この例で使用するコードをダウンロードしてください。ダウンロード・ファイルには、単純なアプリケーションと、`site-maven-plugin` が構成された POM ファイルが含まれています。

コードを入手する

Maven リポジトリを GitHub 上でホストする理由は、ライセンスの問題、プライバシー、有料ホスティング・サービスにかかるコストなど、さまざまにあります。どのような理由で GitHub ベースの Maven リポジトリをホストすることにしたかに関わらず、セットアップするのは簡単です。

まず、既存の GitHub リポジトリ内でブランチを作成し、そのブランチに直接、プロジェクトの成果物をインストールします。これにより、Maven ライフサイクルの `deploy` フェーズにバインドされるので、プロジェクトをデプロイすると、リモート・リポジトリが自動的に更新されます。

GitHub 資格情報を追加する

GitHub リポジトリ内に Maven ブランチを作成した後は、GitHub 資格情報を Maven の `settings.xml` ファイルに追加する必要があります。このファイルは通常、`/.m2` ディレクトリ内

にあります。settings.xml ファイル内の <servers> 要素で囲んだ部分に、新規サーバーを定義して、ユーザー名とパスワードを指定します。リスト 19 に一例を示します (この例では、新規サーバーには github という名前を付けています)。

リスト 19. github リポジトリをサーバーとして構成する

```
<servers>
<server>
  <id>github</id>
  <username>USERNAME_OR_EMAIL</username>
  <password>PASSWORD</password>
</server>
</servers>
```

2 要素認証

2 要素認証が有効にされている場合、「GitHub Settings (GitHub の設定)」を使用して [パーソナル・アカウント・トークン](#) を作成する必要があります。このトークンをコピーして、<password> 要素で囲んだ部分に直接貼り付けます。セキュリティを強化するために、パスワードを暗号化することもできます。その場合は、このリンク先の [Maven Web](#) サイトで説明されている手順に従ってください。

Maven サイト・プラグイン

Maven POM 内では、GitHub の [site-maven-plugin](#) を使用することになります。最初に必要な作業は、ファイルのコミット元とするベース・ディレクトリを設定することです。それにはまず、リスト 20 の例に示すように、ステージング・リポジトリを作成して、プロジェクトのターゲット・ディレクトリ内に配置します。この例では、ステージング・リポジトリに repo-stage という名前を付けています。

次は、このプラグインを deploy フェーズに関連付けるために、maven-deploy-plugin 内で、リポジトリの場所を <altDeploymentRepository> で囲んで指定します。リスト 20 のコード・スニペットに、その方法が示されています。

リスト 20. 代替ローカル・リポジトリを指定する

```
<plugin>
  <artifactId>maven-deploy-plugin</artifactId>
  <version>2.8.2</version>
  <configuration>
    <altDeploymentRepository>
      Repo.stage::default::file://
        ${project.build.directory}/repo-stage
    </altDeploymentRepository>
  </configuration>
</plugin>
```

最後に site-maven-plugin を構成して、このプラグインを deploy フェーズに関連付けます。それには、プラグインに、前に作成した GitHub サーバーの id を渡します。それに加え、リポジトリの名前と所有者、プロジェクト成果物をコミットするブランチ、コミットに関するコメントも渡す必要があります。また、Jekyll をオフにして、GitHub ページが生成されないようにする必要があります。

リスト 21 に、`site-maven-plugin` の完全な構成を記載します。ブランチのリポジトリ名は `dependency` となっていることに注意してください (このブランチが存在しなければ、GitHub が作成します)。

リスト 21. サイト・プラグインを構成する

```
<plugin>
  <groupId>com.github.github</groupId>
  <artifactId>site-maven-plugin</artifactId>
  <version>0.12</version>
  <configuration>
    <!-- Github settings -->
    <server>github</server>
    <repositoryName>MavenSampler</repositoryName>
    <repositoryOwner>atheedom</repositoryOwner>
    <branch>refs/heads/dependency</branch>
    <message>
      Artifacts for
      ${project.name}/${project.artifactId}/${project.version}
    </message>
    <noJekyll>true</noJekyll>
    <!-- Deployment values -->
    <outputDirectory>
      ${project.build.directory}/repo-stage
    </outputDirectory>
    <includes>
      <include>*/*</include>
    </includes>
  </configuration>
  <executions>
    <execution>
      <phase>deploy</phase>
      <goals>
        <goal>site</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

残る作業は、コマンド `mvn clean deploy` を実行することだけです。コンソールで、プラグインがリポジトリのファイルを GitHub アカウントにアップロードし、コミットを作成し、それを `dependency` ブランチにプッシュする一連のフローを確認できます。図 3 に、コンソールの出力例を示します。

図 3. コミットが成功した場合のコンソールの出力

```
[INFO] --- maven-deploy-plugin:2.8.2:deploy (default-deploy) @ maven-sampler ---
[INFO] Using alternate deployment repository repo.stage::default::file:///C:/Alex/projects/Maven Sampler/target/repo-stage
Uploading: file:///C:/Alex/projects/Maven Sampler/target/repo-stage/com/readlearncode/maven-sampler/1.0/maven-sampler-1.0.jar
Uploaded: file:///C:/Alex/projects/Maven Sampler/target/repo-stage/com/readlearncode/maven-sampler/1.0/maven-sampler-1.0.jar (4 KB at 255.9 KB/sec)
Uploading: file:///C:/Alex/projects/Maven Sampler/target/repo-stage/com/readlearncode/maven-sampler/1.0/maven-sampler-1.0.pom
Uploaded: file:///C:/Alex/projects/Maven Sampler/target/repo-stage/com/readlearncode/maven-sampler/1.0/maven-sampler-1.0.pom (7 KB at 1022.8 KB/sec)
Downloading: file:///C:/Alex/projects/Maven Sampler/target/repo-stage/com/readlearncode/maven-sampler/maven-metadata.xml
Uploading: file:///C:/Alex/projects/Maven Sampler/target/repo-stage/com/readlearncode/maven-sampler/maven-metadata.xml
Uploaded: file:///C:/Alex/projects/Maven Sampler/target/repo-stage/com/readlearncode/maven-sampler/maven-metadata.xml (306 B at 42.7 KB/sec)
[INFO]
[INFO] --- site-maven-plugin:0.12:site (default) @ maven-sampler ---
[INFO] Creating 9 blobs
[INFO] Creating tree with 10 blob entries
[INFO] Creating commit with SHA-1: 27fbed0a7c5a3cc2402c95390b208ec1c09c89a9
[INFO] Creating reference refs/heads/dependency starting at commit 27fbed0a7c5a3cc2402c95390b208ec1c09c89a9
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 40.185 s
[INFO] Finished at: 2017-08-02T15:42:36+01:00
[INFO] Final Memory: 20M/166M
[INFO] -----
```

すべてが正常に機能していることを確認するために、GitHub リポジトリにアクセスして、dependency ブランチを選択してください。プロジェクトのすべての成果物が表示されていれば、すべて問題なく機能していることとなります (図 4 を参照)。

図 4. GitHub 内の Maven リポジトリのスナップショット

```
C:\Alex\projects\Maven Sampler>keytool -genkey -alias readlearncode.com -keyalg RSA -keystore keystore.jks -keysize 2048
Enter keystore password:
Re-enter new password:
What is your first and last name?
[Unknown]: Alex Theedom
What is the name of your organizational unit?
[Unknown]: readlearncode.com
What is the name of your organization?
[Unknown]: readlearncode.com
What is the name of your City or Locality?
[Unknown]: London
What is the name of your State or Province?
[Unknown]: London
What is the two-letter country code for this unit?
[Unknown]: UK
Is CN=Alex Theedom, OU=readlearncode.com, O=readlearncode.com, L=London, ST=London, C=UK correct?
[no]: y

Enter key password for <readlearncode.com>
(RETURN if same as keystore password):
Re-enter new password:
```

成果物が確実に GitHub にコミットされていることを確認した後は、あなたが作成した成果物を頼りにしたいという他の開発者と新しいリポジトリを共有してください。

GitHub をリポジトリとして追加する

GitHub Maven リポジトリを、サブスクライバーのプロジェクトで利用できるリモート・リポジトリとして指定するには、まず、サブスクライバーが自分の POM 内でこの GitHub Maven リポジトリを宣言する必要があります (リスト 22 を参照)。その上で、プロジェクトの成果物を通常の方法で指定すれば、サブスクライバーで利用できるようになります (リスト 23 を参照)。

リスト 22. GitHub をリモート・リポジトリとして指定する

```
<repositories>
  <repository>
    <id>PROJECT_NAME-dependency</id>
    <url>
      https://raw.github.com/
      GITHUB_USERNAME/PROJECT_NAME/dependency
    </url>
    <snapshots>
      <enabled>true</enabled>
      <updatePolicy>always</updatePolicy>
    </snapshots>
  </repository>
</repositories>
```

リスト 23. プロジェクトの依存関係

```
<dependency>
  <groupId>com.readlearncode</groupId>
  <artifactId>maven-sampler</artifactId>
  <version>1.0</version>
</dependency>
```

これで、作成した依存関係を手軽な公開 Maven リポジトリにデプロイして、世界中と共有できるようになりました。

5. さらに時間を節約するためのアドバイスとコツ

Apache Maven はかなり広く使われているため、まだ知られていないアドバイスとコツを探すのは金鉱を探すようなものです。そこで、私が極めて重宝と思った、選り抜きのアドバイスとコツを紹介して締めくくことにします。これらの中には、皆さんの時間の節約とフラストレーションの軽減に役立つものもあるはずです。

使っていない依存関係のための「ルンバ」

新しい API やテクノロジーを試してみる段階になると尚更のこと、開発作業は楽しくエキサイティングなものです。新しい API やテクノロジーを迅速かつ簡単に導入するという点でも、Maven は役立ちます。依存関係をコピーして POM ファイルに貼り付けるだけで、その依存関係を使い始めることができるからです。その一方、開発者は時として興奮するあまり、未使用の依存関係を削除し忘れることがあります。あるいは、削除するのが面倒だからと、放置する場合もあるでしょう。この悪習慣によって、そのうち POM が膨れ上がってくると、デプロイメントの時間が長引くことになります。

そこで私が提案する最初のアドバイスは、ロボット掃除機「[ルンバ](#)」に相当する Maven のコマンドです。リスト 24 のコマンドを実行するだけで、Maven はプロジェクトのコードを分析して未使用の依存関係を特定します。

リスト 24. 未使用の依存関係を特定する

```
dependency:analyze
```

ローカル・リポジトリをパージする

何かにつけ、Maven リポジトリをパージする必要が生じます。破損が原因であったり、IDE のインデックス付けに非常に時間がかかっていることが原因であったりします。理由は何であれ、リスト 25 のコードを実行することで、リポジトリの中身をすべて削除できます。

リスト 25. ローカル・リポジトリをパージする

```
mvn dependency:purge-local-repository
```

このコードはリポジトリをパージした後、現在のプロジェクトに必要なすべての依存関係を自動的にダウンロードします。プロジェクトは生まれ変わったような気分になって、再び最大限のパフォーマンスを発揮してくれることでしょう。

タイムスタンプを付けること！

ビルドにタイムスタンプを追加すると、非常に役立つことがあります。Maven 2 で導入されたビルド変数 `maven.build.timestamp` は、ビルドの開始時点を示します。この変数は、POM 内の任意の場所で変数 `${maven.build.timestamp}` と併せて使用できます。

このタイムスタンプは Java [SimpleDateFormat](#) クラスを使用してフォーマット化され、`<maven.build.timestamp.format>` プロパティを使用して宣言されます。リスト 26 に、タイムスタンプの使用例を記載します。

リスト 26. タイムスタンプのフォーマット

```
<properties>
  <maven.build.timestamp.format>
    yyyy-MM-dd'T'HH:mm:ss'Z'
  </maven.build.timestamp.format>
</properties>
```

プロパティをエコー出力する

Maven プロパティの値を知りたいと思ったことはありませんか？その場合は、推測するのではなく、[maven-antrun-plugin](#) を使用して確かめることができます。この簡潔なツールは、渡されたプロパティが何であろうと、そのプロパティをコンソールにエコー出力します。

リスト 27 の例では、`antrun` の `run` ゴールを Maven の `validate` フェーズ (デフォルト・ライフサイクルの最初のフェーズ) に関連付けて、構成セクション内の `<echo>` 要素を使用してプロパティの値を出力しています。

リスト 27. プロパティをコンソールにエコー出力する

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.1</version>
  <executions>
    <execution>
```

```
<phase>validate</phase>
<goals>
  <goal>run</goal>
</goals>
<configuration>
  <tasks>
    <echo>
      settings.localRepository = ${settings.localRepository}
    </echo>
    <echo>
      project.build.directory = ${project.build.directory}
    </echo>
  </tasks>
</configuration>
</execution>
</executions>
</plugin>
```

これで、これからは `mvn validate` を使用するだけで、`validate` フェーズを実行して、コンソールに出力されたプロパティ値を確認することができます。

まとめ

Maven は単なるビルド・ツールではなく、その数々のプラグインのおかげで、幅広い開発タスクを単純化および自動化することができます。この記事では、そのうちのわずかなプラグインを紹介し、プラグインを利用して開発ライフサイクルをより効率的かつ生産的にするためのアドバイスを提供しました。Maven とそのプロジェクト・ライフサイクルについて、さらに詳しく調べるには、このリンク先の記事「[今まで知らなかった 5 つの事項: Apache Maven](#)」を参照してください。記事の内容は、Maven 3 に合わせて最近更新したばかりです。

関連トピック： [この記事の完全なコードを入手する](#) [今まで知らなかった 5 つの事項: Apache Maven](#) [Maven 3 リリース・ノート](#)

著者について

Alex Theedom



May 2017

© Copyright IBM Corporation 2018

(www.ibm.com/legal/copytrade.shtml)

商標

(www.ibm.com/developerworks/jp/ibm/trademarks/)