プロジェクト管理: Mavenでもっと簡単に

Javaの次のビルドにプロジェクト管理機能を追加しよう

Charles Chan
Senior Software Developer
Finetix LLC

2003年 4月 08日

AntはJavaプログラムのビルドにおける事実上の標準となっていますが、このツールはプロジェクト管理のタスクに関しては多くの点でまだ不十分です。これに対して、Apache Jakartaプロジェクトの高機能なプロジェクト管理ツールであるMavenは、Antが提供する機能をすべて網羅したうえで、さらに多くの機能を提供します。今回の記事では、Java開発者であるCharles Chan氏が、Mavenの機能を紹介し、Mavenのプロジェクト設定を順を追って説明します。

現在、ほとんどの開発者は、Javaプログラミングを使用するプロジェクトの標準ビルド・ツールはAntであると考えています。しかし、残念ながら、`make` の代替機能としてのAntのプロジェクト管理機能は、ほとんどの開発者のニーズを満たしていません。Antでビルドしたファイルを調べてみても、プロジェクトの依存関係や、その他のメタ情報 (開発者 / 所有者、バージョン、サイトのホームページなど) を知ることは困難です。

Mavenは、プログラムのビルド機能を備えているだけでなく、Antにはない高機能なプロジェクト管理ツールを提供します。Mavenのデフォルトのビルド規則はきわめて再利用性が高いため、単純なプロジェクトであれば、多くの場合Mavenのビルド・スクリプトを2～3行書けばビルドすることができます。Antでは、このような場合にも何十行ものスクリプトが必要です。実際、多くのApache Jakartaプロジェクトでは現在Mavenが使用されています。また、Mavenがプロジェクト指向であることから、企業プロジェクトにおけるMavenの採用率も増加し続けています。

Maven対Ant

それでは、MavenはAntとどのように違うのでしょうか。この質問に答える前に、まずMavenとAntがビルドの問題の異なる2つの側面に焦点を当てていることを強調しておきたいと思います。Antは、Javaテクノロジーを使用した開発プロジェクトにクロス・プラットフォームのビルド機能を提供します。一方、Mavenはプロジェクトの上流工程を扱い、ビルド機能のほとんどはAntから借用しています。つまり、MavenとAntは大きく異なる問題を扱うものであるため、ここでは表1に示すように、2つのツールの同等のコンポーネントについてだけ両者の違いを比較することにします。

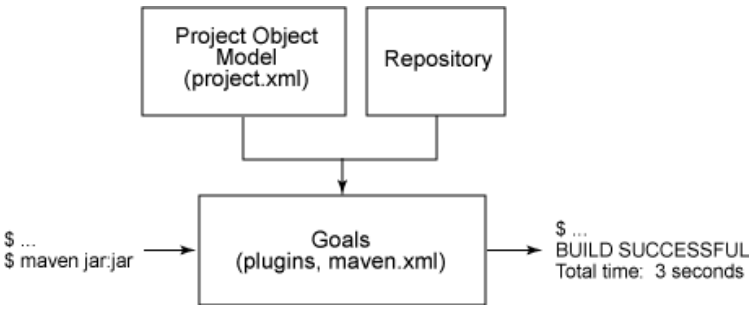
表1. Maven対Ant

	Maven	Ant
標準のビルド・ファイル	`project.xml, maven.xml`	`build.xml`
プロパティーの処理順序	1. `${maven.home}/bin/driver.properties` 2. `${project.home}/project.properties` 3. `${project.home}/build.properties` 4. `${user.home}/build.properties` 5. -Dコマンド・ライン・オプションで定義されたシステム・プロパティー **最後**の定義が優先。	1. -Dコマンド・ライン・オプションで定義されたシステム・プロパティー 2. `<property>` タスクによってロードされたプロパティー **最初**の定義が優先。
ビルド規則	ビルド規則はより動的 (プログラミング言語に類似)。Jellyに基づく実行可能なXML。	ビルド規則はやや静的 (`<script>` タスク使用時を除く (参考文献のチュートリアルを参照)。
拡張言語	プラグインはJelly (XML) で記述。	プラグインはJava言語で記述。
ビルド規則の拡張性	ビルドのゴール (goal) は、`<preGoal>` および`<postGoal>` を定義することにより拡張可能。	ビルド規則は簡単に拡張できない。`<script>` タスクを使用して`<preGoal>` および`<postGoal>` の効果をシミュレートできる。

Mavenの主要コンポーネント

MavenとAntの違いがわかってきたところで、Mavenの主要なコンポーネントについて考察してみましょう。図1にその主要コンポーネントを示します。

図1. Mavenの主要コンポーネント



プロジェクト・オブジェクト・モデル (POM)

プロジェクト・オブジェクト・モデル (POM) は、プロジェクトのさまざまな側面を記述したものです。POMの実際の表現方法には本質的に制限がありませんが、Mavenの開発者は一般にXMLのプロジェクト・ファイル (project.xml)を使用します。このXMLファイルのフォーマットは、Mavenをインストールしたディレクトリーにあ るXMLスキーマ、maven-project.xsdで定義されています。

一般に、project.xmlファイルは、次の3つのセクションから構成されています。

- プロジェクト管理のセクションには、プロジェクトの組織、開発者のリスト、ソース・コードの場所、バグ追跡システムのURLなどが含まれます。
- プロジェクトの依存関係のセクションには、プロジェクトの依存関係に関する情報が含まれます。現在のMavenの実装 (1.0 beta 8) では、JARファイルの依存関係のみがサポートされています。

- プロジェクトのビルドとレポートのセクションには、ソース・コードのディレクトリー、単体テスト・ケースのディレクトリー、ビルドで生成するレポートなど、プロジェクトのビルド情報が含まれます。

リスト1は、project.xmlファイルの注釈付きのサンプルです。project.xmlファイルの多くの要素はオプションであるため、理解を深めながら徐々にMavenの異なる機能を組み込んでいくことができます。注意 : 以降のコードでは、オプションの要素を "OPTIONAL" として示してあります。

メイン文書には、プロジェクト固有の識別子とグループIDが含まれます。グループIDは、プロジェクトが複数のサブプロジェクトで構成される場合には特に役立ちます。すべてのサブプロジェクトは同じグループIDを持ちながら、それぞれ異なる`<id>` を持ちます。

リスト1. メインのproject.xmlのスケルトン

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- A project file's root element -->
<project>
  <!-- The POM version. This tag is currently unused. -->
  <pomVersion>3</pomVersion>
  <!-- A project group id. If present, the id serves as the project's
        directory name in the repository -->
  <groupId>crayola-group</groupId>
  <!-- A unique project identifier. The project identifier and its
        version number often generate file/directory names during the
        build. For example, a project JAR file follows the
        <id>-<version> naming convention. -->
  <id>crayola</id>
  <!-- A short name for the project -->
  <name>Crayola Professional</name>
  <!-- The project version number. Maven does not enforce a particular
        version numbering scheme. -->
  <currentVersion>0.0.1</currentVersion>
  ...
  <!--
-------------------------------------------------------------- -->
  <!-- Project management section                             -->
  <!--
-------------------------------------------------------------- -->
  ...
  <!--
-------------------------------------------------------------- -->
  <!-- Project dependency section                             -->
  <!--
-------------------------------------------------------------- -->
  ...
  <!--
-------------------------------------------------------------- -->
  <!-- Project build and reports section                      -->
  <!--
-------------------------------------------------------------- -->
  ...
</project>
```

リスト2に示すプロジェクト管理のセクションに含まれる項目は、ほとんどがオプションです。このセクションには、Change LogレポートやDevelopment Activityレポートを作成する場合には特に、開発者 (とその正確なID) を指定します。

リスト2. プロジェクト管理のセクション

```
  ...
```

```
  <!--
------------------------------------------------------------------- -->
  <!-- Project management section                                    -->
  <!--
------------------------------------------------------------------- -->
  <!-- Details of the organization holding the project. Only the name
       is required. -->
  <organization>
    <name>Markers Inc.</name>
    <url>http://w3.markers.com/</url>
    <logo>http://w3.markers.com/logo/company-logo.gif</logo>
  </organization>
  <!-- (OPTIONAL) Year of inception -->
  <inceptionYear>2003</inceptionYear>
  <!-- (OPTIONAL) Project main package -->
  <package>com.markers.crayola.*</package>
  <!-- (OPTIONAL) Project logo picture (URL) -->
  <logo>http://w3.markers.com/logo/crayola.gif</logo>
  <!-- (OPTIONAL) GUMP repository ID. Useful only if you use GUMP. -->
  <gumpRepositoryId>crayola</gumpRepositoryId>
  <!-- (OPTIONAL) Project description -->
  <description>...</description>
  <!-- (OPTIONAL) Short project description -->
  <shortDescription>...</shortDescription>
  <!-- (OPTIONAL) Project site URL -->
  <url>http://w3.markers.com/crayola</url>
  <!-- (OPTIONAL) Issue-tracking system URL -->

<issueTrackingUrl>http://w3.markers.com/jira/crayola</issueTrackingUrl>
  <!-- (OPTIONAL) Project site address. -->
  <siteAddress>w3.markers.com</siteAddress>
  <!-- (OPTIONAL) Project-site deploy directory (physical location) -->
  <siteDirectory>/www/crayola/site/</siteDirectory>
  <!-- (OPTIONAL) Project distribution directory (physical location) -->

<distributionDirectory>/www/crayola/builds/</distributionDirectory>
  <!-- (OPTIONAL) Project source-repository information -->
  <repository>

<connection>
      scm:cvs:pserver:anoncvs@cvs.markers.com:/home/cvspublic:crayola
    </connection>
    <url>http://cvs.markers.com/viewcvs/crayola/</url>
  </repository>
  <!-- (OPTIONAL) Mailing list information -->
  <mailingLists>
    <mailingList>
      <name>Dev List</name>

<subscribe>dev-subscribe@crayola.markers.com</subscribe>

<unsubscribe>dev-unsubscribe@crayola.markers.com</unsubscribe>
    </mailingList>
    ...
  </mailingLists>
  <!-- Developers involved in this project -->
  <developers>
    <developer>
      <name>John Smith</name>
      <id>jsmith</id>
      <email>jsmith@markers.com</email>
    </developer>
    ...
  </developers>
```

リスト3の情報は、集中型の成果物レポジトリーに付随して、CLASSPATH の設定ミスや依存関係とバージョンの不整合など、ビルドに関するいくつかの問題の排除に使用されます。

リスト3. プロジェクトの依存関係のセクション

```
  <!--
-------------------------------------------------------------- -->
  <!-- Project dependency section                              -->
  <!--
-------------------------------------------------------------- -->
  <dependencies>
    <!-- This project depends on the JAR file "commons-beanutils-1.5.jar"
          in the Maven repository's commons-beanutils/jars subdirectory
          (more about repository later). -->
    <dependency>
      <groupId>commons-beanutils</groupId>
      <artifactId>commons-beanutils</artifactId>
      <version>1.5</version>
    </dependency>
    <!-- This project depends on the JAR file "commons-lib-2.1.jar" in
          the Maven repository's markers/jars subdirectory. -->
    <dependency>
      <groupId>markers</groupId>
      <artifactId>commons-lib</artifactId>
      <version>2.1</version>
    </dependency>
  </dependencies>
```

リスト4に示すプロジェクトのビルドとレポートのセクションには、Mavenのいくつかのプラグインの設定に使用される重要なビルドおよびレポート情報が含まれます。たとえば、サイトの文書が生成されるときに、特定のレポートを組み込んだり除外したりするようにMavenを設定できます。

リスト4. プロジェクトのビルドのセクション

```
  ...
  <!--
-------------------------------------------------------------- -->
  <!-- Project build and reports section                       -->
  <!--
-------------------------------------------------------------- -->
  <build>
    <!-- (OPTIONAL) Build notification email address. -->
    <nagEmailAddress>jsmith@markers.com</nagEmailAddress>
    <!-- (OPTIONAL) Defines where the Java source resides. -->
    <sourceDirectory>src/java</sourceDirectory>
    <!-- (OPTIONAL) Defines where the Java source for unit test-cases
          resides. -->

<unitTestSourceDirectory>test/java</unitTestSourceDirectory>
    <!-- (OPTIONAL) Unit test-case file pattern. -->
    <unitTest>
      <includes>
        <include>**/*Test.java</include>
      </includes>
    </unitTest>
    <!-- (OPTIONAL) Resources packaged inside the JAR file. -->
    <resources/>
    <!-- (OPTIONAL) The reports tag lets you select which reports you
          want generated for your site. In this case, only the checkstyle
          report will generate. -->
  </build>
```

```
  <reports>
    <report>
      maven-checkstyle-plugin
    </report>
  </reports>
```

プロジェクトは、ライブラリーに依存してその機能を果たします。たとえば、あるプロジェクトはログ作成でlog4jに依存し、XSLT変換ではXalanに依存しているかもしれません。また、別のJ2EEプロジェクトでは、WebコンポーネントがEJBコンポーネントに依存してビジネス処理を実行しているということもあるでしょう。Mavenでは、POMの中に様々な依存関係を表現できます。表2に示すタグを使用することにより、project.xmlファイルにそれぞれの依存関係を記述できます。

表2. プロジェクトの依存関係のセクション

- **`groupId`** - リポジトリー内のどのサブディレクトリーに依存関係のファイルが含まれているかを示します。
- **`artifactId`** - 成果物の固有の識別子を示します。
- **`version`** - 依存関係のバージョン番号を表します。
- **`jar`** - (OPTIONAL) 依存関係のJARファイルを表します。ほとんどの場合、JARファイルの名前は依存関係の`<artifactId>` および`<version>` から作成できます。
- **`type`** - (OPTIONAL) jar、distributionなど、依存関係のタイプです。デフォルトはjarです。
- **`url`** - (OPTIONAL) 依存関係のプロジェクトのURLです。このURLは、依存関係がインターネット上のサード・パーティーのライブラリーである場合に特に便利です。

リポジトリー

リポジトリーは、Mavenのもうひとつの主要なコンポーネントです。複数のプロジェクトが存在するJavaを基盤としたサイトでは、多くの場合、サード・パーティーのライブラリーから成る集中リポジトリーによってプロジェクト間の整合性が確保されます。Mavenでは、リポジトリーの構造が標準化され、インターネットおよびイントラネットの任意の場所でサービスを提供するリモート・リポジトリーがサポートされます。リスト5は、リポジトリーの一般的な構造を示しています。

リスト5. リポジトリー

```
repository
|-- ant                  <-- project group ID -->
|   `-- jars             <-- artifact type, followed by 's',
|                        <-- e.g. jars, wars, ears -->
|       `-- ant-1.5.1.jar <-- actual artifact -->
...
```

リモート・リポジトリーを作成するには、Webサイトにリポジトリーのディレクトリーを配置します。Mavenでは、リモート・リポジトリーの使用が推奨されています。これにより、ユーザーはリモート・リポジトリーを中央で管理でき、プロジェクト間の資源共有の可能性を最大限に引き出すことができます。ビルドごとにファイルをダウンロードするのを避けるため、Mavenは必要な依存関係が最初にダウンロードされるときに、それをローカル・リポジトリーにキャッシュします。Mavenは、リモートとローカルのリポジトリー用に表3のプロパティーを使用します。

表3. リモートおよびローカルのリポジトリーのプロパティー

maven.repo.remote
: リモート・リポジトリーにカンマ区切りの URL リストを指定します。デフォルトでは http://www.ibiblio.org/maven が使用されます。

maven.proxy.host、**maven.proxy.port**、**maven.proxy.username**、**maven.proxy.password**
: ファイアウォールがあり、インターネットにアクセスするためにプロキシー認証が必要な場合は、これらの便利な設定をすぐ使用できます。

maven.repo.local
: ダウンロードされた依存関係のキャッシュ先を指定します。デフォルトは `${MAVEN_HOME}/repository` です。UNIX 環境でリポジトリーのディレクトリーを複数のチームで共有する場合は、開発者の特別なグループを作成して、そのグループに該当リポジトリーへの読み込み / 書き込みアクセスを与えることができます。

ゴール (goal)

MavenのAntタスク

Mavenのゴール定義には任意の有効なAntタスクを含めることができます。このことは、Mavenの早期学習に役立つとともに、Antへの投資を守ることにもなります。

Mavenにおけるゴールは、Antの`target` に似ています。ゴールにもターゲットにも、それを達成するときに実行されるタスクが含まれています。コマンド・ラインで特定のゴールを達成するには、`maven <goal>` と入力します。

定義済みのゴールをすべて表示するには、`maven -g` を使用します。表4に、頻繁に使用されるゴールを示します。

表4. 頻繁に使用されるゴール

- **java:compile** - すべてのJavaソースをコンパイルします。
- **jar** - コンパイルされたソースのJARファイルを作成します。
- **jar:install** - 作成されたJARファイルをローカル・リポジトリーに公開します。これにより、他のプロジェクトもそのJARファイルを参照できるようになります。
- **site** - プロジェクトのサイト文書を作成します。デフォルトのサイト文書には、パッケージ / クラスの依存関係、コーディング・スタイルの適合性、ソース・コードの相互参照、単体テストの結果、javadocsなど、プロジェクトに関する有益な情報が含まれています。生成するレポートのリストはカスタマイズ可能です。
- **site:deploy** - 生成されたサイト文書を配置します。

Mavenのゴールは、拡張可能かつ再利用可能です。このことを考慮に入れて、独自のゴールを作成する前にMavenのサイト (または`${MAVEN_HOME}/plugins`) で、Mavenのプラグインについて確認しておいてください。SourceForgeのMavenプラグイン・プロジェクトも、無料のMavenプラグインの情報を入手できる優れたサイトです (これらの各項目へのリンクについては、参考文献を参照してください)。

目的に合ったゴールが見つからない場合、Mavenには2つの選択肢があります。

- `<preGoal>` または`<postGoal>` を記述して、標準のゴールを拡張する。

- 独自のゴールを作成する。

どちらの場合も、maven.xmlという特別なファイルをプロジェクトのディレクトリーに作成する必要があります。リスト6はmaven.xmlのスケルトンです。

リスト6. maven.xmlのスケルトン

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project xmlns:j="jelly:core">
  ...
  <goal name=...>
    ... build rules, e.g.
    <mkdir dir="${test.result.dir}"/>
    <echo>Executing JUnit tests</echo>
    ...
  </goal>
  ...
  <preGoal name=...>
    ...
  </preGoal>
  <postGoal name=...>
    ...
  </postGoal>
</project>
```

Antに精通した開発者であれば、Mavenのゴール (同様に`preGoal` および`postGoal` も) の定義に任意の有効なAntタスクを含めることができるということが、Mavenの早期学習に役立つとともに、Antへの投資を保護することにもなるということがわかると思います。Antのタスクにより柔軟性を与えるために、MavenではJellyスクリプト言語も使用されています。「Basic Jelly programming」ではサンプルのmaven.xmlファイルを使ってJellyスクリプト言語を紹介していますので、是非参照してください。

<preGoal> と <postGoal> の記述

Ant の `<target>` は、規則を定義した後は事前条件と事後条件が固定されるという点で、`makefile` の規則に似ています。これにより、複数のプロジェクト間でのビルド規則の再利用は難しくなります。たとえば、あるプロジェクトの `compile` ターゲットではソース・ファイルの生成を XDoclet に依存している一方、別のプロジェクトの `compile` ターゲットには何の前提条件もないという場合があるからです。この限界を克服するために、Maven は 2 つの特別なタグ、`<preGoal>` と `<postGoal>` を提供しています。タグ名から想像できると思いますが、`preGoal` は指定されたゴールの前に実行されるべきビルド規則を定義します。これに対して `postGoal` は、指定されたゴールが達成された後で実行されるべきビルド規則を定義します。たとえば、リスト 7 の `preGoal` は、ソース・コードをコンパイルする前に、XDoclet を使用してソース・コードを作成するように Maven に指示しています。

リスト7. preGoalセクションのサンプル

```
  <preGoal name="java:compile">
    <attainGoal name="xdoclet:ejbdoclet"/>
  </preGoal>
```

また、Mavenは、上記のように直接ゴールを達成することが重要であるような場合のために、Antの`<antcall>` タグによく似た`<attainGoal>` タグを提供しています。

独自のゴールの作成

ゴールがプロジェクトに固有の場合は、maven.xmlファイル内に独自のゴールを定義できます。このように自分で定義したゴールは、同じ名前の他のゴールをオーバーライドします。プロジェクトにサブプロジェクトが含まれる場合は、サブプロジェクトもこれらのゴールを継承します。

プラグインの作成

プロジェクト間でゴールを共有するには、そのゴールをプラグインとしてパッケージ化し、Mavenのインストール・ディレクトリーのプラグイン用ディレクトリー (`${MAVEN_HOME}/plugins`) に配置します。一般的なMavenプラグインには、project.xmlおよびplugin.jellyファイルが含まれます。project.xmlファイルには、プラグインのPOMが記述されます。plugin.jellyはmaven.xmlに類似しており、ここにはそのプラグインによって公開されるゴールが含まれます。プラグインは、それ自身のための資源と依存関係を持つことができます。定義済みの変数`${plugin.dir}` により、ユーザーはプラグイン・ディレクトリー内の資源を参照することができます。たとえば、リスト8に示すプラグイン構造では、`${plugin.dir}/dtd/web-app_2_3.dtd` とすることで`plugin.jelly` 内から`web-app_2_3.dtd` にアクセスできます。

リスト8. プラグイン構造のサンプル

```
ejbjar-plugin-1.0
|-- dtd
|   |-- application_1_3.dtd
|   |-- ejb-jar_2_0.dtd
|   |-- web-app_2_3.dtd
|-- plugin.jelly
`-- project.xml
```

Mavenのインストール

最近リリースされたMaven 1.0-beta-8は、1.0の機能をほぼ完全に備えています。Mavenの開発コミュニティーでは毎日バグを修正しているため、何か問題があった場合には、CVS (Concurrent Version System) からためらわずに最新のMavenバージョンを入手し、自分でビルドしてください (手順については参考文献を参照してください)。最新のMavenソース・コードをダウンロードしたら、次のコマンドを起動してMavenをビルドします。

```
ant -f build-bootstrap.xml
(set MAVEN_HOME to where you want Maven to reside and
use Ant 1.5.1 to perform the build)
```

ファイアウォールがある場合は、プロパティー、`maven.proxy.host`、`maven.proxy.port`、`maven.proxy.username`、`maven.proxy.password` を適切に設定してください。デフォルトでは、Mavenのリポジトリーは`${MAVEN_HOME}/repository` にあります。この場所は、`maven.repo.local` プロパティーを新しい場所に変更することによって変更できます。

J2EEプロジェクトのサンプル

プロジェクト・ファイルのサンプル

このJ2EEプロジェクトのサンプルで使用されているコードについては、Mavenプロジェクト・ファイルを参照してください。

ここまでに学んだ知識を活用して、いよいよMavenを実際に使用するときが来ました。このセクションでは、J2EEプロジェクトのサンプルをMavenで設定する方法を説明します。

プロジェクトのディレクトリー・レイアウト

細かい説明に入る前に、まずディレクトリーのレイアウトを説明します。必須とは言いませんが、プロジェクト間で一貫したディレクトリー・レイアウトを持つことは、非常に有益です。これは、1つのプロジェクトに精通した開発者であれば、他のプロジェクト内のディレクトリーも簡単に理解して使用できるためです。さらに重要なことは、一貫したディレクトリーのレイアウトにより、汎用的なビルド規則の作成が可能になることです。

Mavenのディレクトリー・レイアウトのガイドライン (参考文献を参照) は、ほとんどのプロジェクトに適用できます。例として、推奨とは多少異なっていますが、リスト9にレイアウトを示します。

リスト9. プロジェクトのディレクトリー・レイアウトのサンプル

```
project
|
|-- LICENSE.txt
|-- project.properties
|-- maven.xml
|-- project.xml
|-- src
|   `-- java
|       `-- com/....
|   `-- conf
|       `-- Configuration files for staging environments.
|-- test
|   `-- java
|       `-- com/....
|   `-- conf
|       `-- Configuration files for unit testing environments.
`-- xdocs
    `-- index.xml
```

J2EEプロジェクトは一般に、WARファイル、EJB JARファイル、およびEARファイルを生成します。各プロジェクトには独自の依存関係とソース・ファイルが含まれているため、プロジェクトはそれぞれ個別に作成する必要があります。一般に、このプロジェクト / サブプロジェクトの関係は、サブプロジェクトをマスター・プロジェクトのサブディレクトリーとして保存することによって構築されます。リスト10 にここで使用するレイアウトを示します。

リスト10. MavenにおけるJ2EEプロジェクトの上位ディレクトリー・レイアウト

```
j2ee-project
|
|-- project.xml       - Produces the EAR file
|
|-- util-subproject
|   |
|   `-- project.xml   - Produces the Utility JAR file
|
|-- ejb-subproject
|   |
|   `-- project.xml   - Produces the EJB JAR file
|
`-- web-subproject
    |
    `-- project.xml   - Produces the WAR file
```

プロジェクトの継承

プロジェクトの継承は、POMがマスターPOMを継承する、ある意味でオブジェクトの継承に似た機能です。この機能は、プロジェクト間の差異 (主に依存関係の) が非常に小さいこの例では特に重要です。プロジェクト管理のセクションは、マスターのproject.xmlで集中管理できます。プロジェクトの継承機能を使用するには、project.xml内の`<extend>` タグを使用します (リスト2の「プロジェクト・ファイルのサンプル」を参照してください)。

サンプル・プロジェクトのゴール

POMの定義が終わったので、次はゴールを作成します。これらのゴールではPOMで定義されたプロパティーが使用されるため、先へ進む前にまず「プロジェクト・ファイルのサンプル」のproject.xmlを理解してください。

Utilityサブプロジェクト

Utilityサブプロジェクトは、クラスを含むJARファイルをソース・ディレクトリーに作成します。この要件は、デフォルトの`jar:jar` ゴールによって達成されるため、ここではカスタムのゴールは必要ありません。

WebサブプロジェクトとEJBサブプロジェクトは、ともにUtilityサブプロジェクトに依存しているため、これらのサブプロジェクトを作成する前に`jar:install` ゴールを呼び出して、Utilityサブプロジェクトの JARファイルをローカル・リポジトリーに配置しておく必要があります。これにより、WARサブプロジェクトとEJBサブプロジェクトは依存関係を適切に解決できます。

Webサブプロジェクト

Webサブプロジェクトは、ソース・ディレクトリーにあるクラス、`jsp` ディレクトリーにあるJSPファイル、そして`conf` ディレクトリーにあるweb.xmlファイルを含んだweb.xmlファイルを作成します。デフォルトの`war:war` ゴールにあるのは、プロジェクトのディレクトリー・レイアウトのより単純なものです。このゴールを再利用するために、次のようにその動作をカスタマイズします。

1. プロジェクトのproject.propertiesファイルで、`maven.war.src` および`maven.war.webxml` プロパティーをそれぞれ`${maven.build.dir}/webapp` および`${maven.src.dir}/conf/web.xml` に設

定します。これらのプロパティーは、`war:war` にWebソース (JSPページ、静的HTMLページ、イメージなど) とweb.xml ファイルを探す場所を指示します。
2. すべてのJSPファイルを`${maven.build.dir}/webapp` にコピーする`preGoal` を定義します。次のmaven.xmlは、この動作を実行するものです。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project>
  <preGoal name="war:init">
    <copy todir="${maven.build.dir}/webapp">
      <fileset dir="${maven.src.dir}/jsp" include="*.jsp"/>
    </copy>
  </preGoal>
</project>
```

`war:war` ゴールを起動するときに、Utility JARファイルとcommons-beanutils JARファイルがWARファイルにパッケージされることに注目してください。Mavenは、project.xmlファイルの依存関係のセクションにある`war.bundle.jar` プロパティーから、WARファイルに含めるファイルを認識します。

EJBサブプロジェクト

EJB JARファイルのパッケージ化は、JARファイルのパッケージ化に似ています。プロジェクトの設定がデフォルトの`ejb` ゴールと同じでない場合は、前述の「Webサブプロジェクト」で説明した手法を使用できます。この場合は、`conf` ディレクトリーから`${maven.build.dir}/ejb/META-INF` ディレクトリーにejb-jar.xmlをコピーし、`maven.ejb.src` プロパティーを`${maven.build.dir}/ejb` に設定します。

依存しているJARファイルをEJB JARのマニフェスト・クラスパスに追加するには、依存関係のセクションの`ejb.manifest.classpath` プロパティーを使用します。

マスター (EAR) プロジェクト

サブディレクトリーの正常なコンパイルと配置が完了したら (`jar:install`、`war:install`、および`ejb:install` の各ゴールを使用)、最後にEARファイルを作成します。依存関係のプロパティー、`ear.bundle.jar`、`ear.bundle.ejb`、および`ear.bundle.war` は、EARファイルに含めるファイルを`ear` プラグインに指示します (Maven 1.0-beta-8では、依存関係のタイプとしてWARファイルはサポートされていないため、EARプラグインはWARファイルを適切にパッケージ化できません。これを避けるには、`postGoal` を使用して手動でEARファイルを更新します)。

リアクター: サブプロジェクトの自動的なビルド

J2EEプロジェクトのビルドには、かなりの労力が必要です。プロジェクトに変更があるたびに同じ手順を繰り返すことは、時間がかかり、エラーも発生しやすくなります。このような問題を緩和するために、Mavenのリアクター機能はサブプロジェクトを正しい順序で自動的にビルドし、時間を節約してミスを減少させます。

リスト11のmaven.xmlは、リアクターの定義方法の1つです。

リスト11. リアクター定義のサンプル

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project default="all"
         xmlns:m="jelly:maven">
  <goal name="all">
    <m:reactor basedir="${basedir}"
               includes="*/project.xml"
               goals="install"
               banner="Building"
               ignoreFailures="false"/>
  </goal>
</project>
```

リアクターは、最初に `basedir` ディレクトリーでproject.xmlファイルを探し、次に`install` ゴールを起動します。実行順序は、各プロジェクトの依存関係のセクションによって異なります。また、リアクターは一般に、マスター・プロジェクトのmaven.xmlファイルに定義します。ゴールはサブプロジェクトに継承されるため、ゴールの名前を選択するときには注意してください。

積極的に関わろう

Mavenは豊富な機能を備えた製品ですが、まだベータ段階です。そのため、バグが続出する可能性もあります。だからと言って、慌てないで下さい。バグへの回答を得るための最善の方法は、Mavenのメーリング・リストのアーカイブ (参考文献を参照) で関連のある投稿を見ることです。関連する投稿がなければ、メーリング・リストに読者自身の質問を投稿し、アドバイスを求めてみてください。このメーリング・リストのメンバーは、多くの場合とても協力的です。

バグを正式に報告するには、Mavenプロジェクトの問題追跡システム (参考文献を参照) にアクセスしてください。

いったんMavenに慣れてしまえば、ほとんどの疑問に対する回答がプラグインの実装の中に見つかるということもあり得ます。このような専門的なレベルに達し、かつMavenが選択肢であると考える読者は、是非このコミュニティーにパッチを送付し、Mavenの今後の発展をお手伝いください。

まとめ

現代のプロジェクトは複雑化しています。そのため、私たちはこの複雑性を表現し、処理することのできるツールを求めています。Mavenは、プロジェクト・オブジェクト・モデルと強力なXMLスクリプト言語を組み合わせることで、私たちにそのようなツールを提供します。この記事では、Mavenの`goal` のメカニズムを使用してPOMを定義し、プロジェクトをビルドする方法を学習しました。また、Jellyを使用してビルドの動作をカスタマイズするいくつかの方法についても考察しました。そして、最後に、J2EEのサンプル・プロジェクトでこのような概念を実際に適用しました。皆さんがMavenをダウンロードし、積極的に試してくれることを期待してやみません。

著者は、この記事の校閲者であるJason van Zyl氏に感謝の意を表します。

著者について

Charles Chan

Charles Chanは、カナダのトロントで活躍する独立のコンサルタントです。彼の関心領域は、分散システムやハイ・パフォーマンス・コンピューティング、国際化、ソフトウェア設計パターンなど多岐に渡っています。時間のある時には、オープンソース・コミュニティーに貢献しています。