Maven3のはじめかた

Kengo TODA

Published
with GitBook

Table of Contents

概要

build passing

「Maven3のはじめかた」はMaven3.2.5に対応したMavenの入門ならびに応用を助ける日本語資料（開発中）です。 Gitbookにて電子書籍（PDF, ePUB, MOBI）を無償配布しています。 こちらのQRコードからも無償配布ページに行けます。

ライセンスならびに著作権についてははじめにをご覧ください。 英語の資料をお探しの方は、やや古い情報が混ざりますがSonatype社の資料をお薦めします。

はじめに

この資料はMavenでJavaプロジェクトを開発する技術者に向けて整理されたものです。

Mavenはプロジェクト（コードやリソースの集まり）をビルドしてアーティファクト（成果物）を得るためのツールです。 ソースコードのコンパイルに加えて、コンパイルに必要なJARファイルを探したり、単体テストを実行したり、ビルドしたJARファイルを 公開したりといった作業を簡単に行うことができます。またMavenは「テストの前にコンパイル」「JARファイル作成の前にテスト」といった ビルド・ライフサイクル （作業の順番）や、「依存元の前に依存先をビルド」といった 依存関係 をきちんと考えてビルドしてくれます。

このため開発者は細かいことを気にすることなく「何をやりたいのか」だけMavenに伝えれば良くなります。 Mavenを使うことで、誰でも簡単にプロジェクトを同じ手順でビルドできるのです。

なお本ドキュメントはJava開発にある程度なれた個人を想定して書かれており、コンパイル・PATH・CLASSPATH・JARファイル・ 単体テストについての説明は省略しています。

本資料の目的

Mavenは歴史が古いため、情報を公開しているサイトが多数存在します。このためひとたび検索すれば多様な日本語情報を入手できますが、 Maven2の情報が混ざっていたり最新のプラグイン事情を考慮していなかったりと情報の質にバラつきがあります。特にMavenそのものに 詳しくない方には、見つかった情報が今でも有効なのか・利用して良いのかどうかを判別することは難しいと思われます。

本資料は2015年2月時点での最新バージョン、`3.2.5`を前提として情報をまとめています。このため上記の問題を解決し、読者に新鮮な 情報を提供することができると考えています。またCIやJavaEEや"JVM言語"など、近年Mavenと組み合わせることが増えてきた技術との 連携方法についても紹介することで、より実用性の高い助力を目指そうとしています。

皆さんがMavenと親しみ、プロダクト（製品）を効率よく楽しく開発することの一助となれば幸いです。

本資料の構成

本資料は3部構成になる予定です。現時点では第1部ならびに第2部の一部を公開しています。

第1部は入門です。Mavenをこれから利用する方のために、Mavenの利点・導入方法・利用方法・基本用語といった基礎を簡単に紹介しています。 Mavenで管理されたプロジェクトにはじめて参加する開発者が、開発に参加するためのキャッチアップを助けることを目的としています。

第2部は応用です。Mavenをより深く理解し活用することを望む技術者のために、独自プラグインの実装方法・サブモジュールの詳細・ プライベートリポジトリの運用・CIとの連携について紹介する予定です。プロダクト管理の効率向上を支援することを目的としています。

第3部はクックブックです。Java以外の言語をどうビルドするか、サーブレットコンテナやデータベースに依存するコードをどうテストするか、 などの知識を集約できればと思っています。どのような知識にニーズがあるのかわかっていないので、リクエストをお待ちしています。

動作確認環境

`OSX 10.10.2`にて`Oracle JDK8`ならびに`Maven 3.2.5`を使用して動作確認を行っています。 環境に強く依存するコードは含みませんので、JDK8が動く環境であればWindowsであれUNIXであれ問題なく動作すると思われます。

サポート

本資料ならびに本資料が参照するソースコードはGitHubで入手できます。 資料の改善提案や修正依頼などはGitHubのissueあるいはpull requestを通じてご連絡願います。

ライセンス

著作権

謝辞

本資料を作成するきっかけをくださった@ikeike443氏に御礼を申し上げます。 またMavenを継続的に開発・改善してくださっている開発者各位、GitHubというすばらしいフィールドを用意してくださっているGitHub, Inc.、 Javaとそのコミュニティを支えているすべての方に感謝します。1つでも欠けていたらこの資料は産まれませんでした。

Maven入門

この章では、Mavenに初めて触れるJava開発者のためにMavenのインストールから使い方までを説明します。

Mavenのインストール

まず先にJDKをインストールしておいてください。

次にMaven公式サイトから圧縮ファイルをダウンロードし、展開してください。 binディレクトリに実行可能ファイルが含まれていますので、これをPATHに追加すればインストールは完了です。

最後に`mvn --version`を実行して、Mavenが正しく実行されること・意図通りのJDKが認識されていることを確認してください。 参考までに、執筆している環境では以下のような出力が得られました。

```
$ mvn --version
Apache Maven 3.2.5 (12a6b3acb947671f09b81f49094c53f426d8cea1; 2014-12-15T01:29:23+08:00)
Maven home: /usr/local/Cellar/maven/3.2.5/libexec
Java version: 1.8.0_20, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_20.jdk/Contents/Home/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.10.2", arch: "x86_64", family: "mac"
```

このように、Mavenは`mvn`コマンドによって呼び出します。`--version`はMavenやJDKのバージョンを出力するための オプションです。オプションについては適宜解説していきます。

利用されるJDKが意図したものではなかった場合、`JAVA_HOME`環境変数にJDKのホーム（jreディレクトリの親ディレクトリ）を 設定してから再度`mvn --version`を実行して確認してください。

Mavenの使い方

基本的には`mvn package`のように行いたいことを引数に渡して実行するだけです。 この「行いたいこと」をMavenではフェーズ（phase）と呼んでいます。主なphaseをいくつか紹介します。

- clean （一時ファイルの削除）
- validate （プロジェクトの状態確認）
- compile （プロジェクトのコンパイル）
- test-compile （テストコードのコンパイル）
- test （単体テストの実行）
- package （アーティファクトの作成）
- integration-test （インテグレーションテストの実行）
- verify （アーティファクトの検証）
- install （アーティファクトをローカルリポジトリに配置）
- deploy （アーティファクトをリモートリポジトリに配置）

アーティファクト（artifact）とはjarやwarのような成果物のことを指します。 またリポジトリとは他の開発者とアーティファクトを共有するための保管庫を指します。詳しくはのちほど説明します。

ここでは例として、筆者が開発しているfindbugs pluginプロジェクトをビルドしてみましょう。 以下のコマンドを実行してください。`target`ディレクトリが作成され、中にjarファイルが入っていることが確認できますか？

```
$ git clone https://github.com/KengoTODA/findbugs-plugin.git
$ cd findbugs-plugin
$ mvn package
$ ls ./target
```

targetディレクトリにはアーティファクトだけでなくプロジェクトのclassファイルやテストケースのclassファイル、単体テストの実行結果も配置されていることがわかると思います。 コマンドの引数にはpackage（アーティファクトの作成）しか指定していないのに、Mavenがpackageに必要なcompileやtest-compile、testフェーズも実行してくれていたということです。 この理由はビルド・ライフサイクルの節で解説します。

さて、もうひとつ便利なフェーズを試してみましょう。 `mvn clean`を実行すると、`target`ディレクトリに作成された一時ファイルをディレクトリごと削除します。 実行後にディレクトリが削除されていることを確認してください。

```
$ ls -l
$ mvn clean
$ ls -l
```

Mavenはビルドを高速化するため、既に作成されたclassファイルなどを再利用することがあります。 ビルドを完全にまっさらな状態から実行する場合は、事前にcleanを実行しておきましょう。

なお`target`ディレクトリの削除によってcleanの実行を代替できるように見えますが、一時ファイルがtargetディレクトリ以外の場所に作られることもありえますので、ディレクトリの削除ではなくcleanを利用するように心がけましょう。

Antに慣れている方のための補足

Antでは各targetの間に依存関係を明示することができました。 例えば以下のXMLは、packageを実行する前にcompileを実行しなければならないことを意味しています。

```
<target name="compile">
...
</target>

<target name="package" depends="compile">
...
</target>
```

Mavenではこうしたtargetと依存関係、各targetで実行すべきtaskがデフォルトで用意されていると考えてください。 開発者が行うべきことがJavaプロジェクトのビルドである以上、行うべき作業には他のJavaプロジェクトとの共通点が多いはずです。 Mavenはそれをデフォルトで提供することにより、開発者がファイルに書くべき設定を削減しています。

MavenではAntのtargetがフェーズ、Antのtaskがプラグインに相当します。 各フェーズで実行されるプラグインはデフォルトで割り当てられたものに加え、好みのものを追加することも可能です。

makeやnpmに慣れている方のための補足

Mavenのインストールは`make install`や`npm install -g`とは違い、プロジェクトから実行可能ファイルを作ってPATHに追加することを意味しません。 Mavenのインストールはアーティファクトを作ってプライベートリポジトリに配置することです。

またデフォルトではビルドされたアーティファクトは`java -jar`で実行できません。 マニフェストファイルを適切に作成しアーティファクトに同梱する必要があります。

ビルド・ライフサイクル

さて、Mavenはどのようにしてpackageの実行にcompileやtestが必要だと判断したのでしょうか。 これを理解するには、ビルド・ライフサイクルについて知る必要があります。

ビルド・ライフサイクルとは「コンパイル→テスト→JAR作成」などのビルドにおける作業の順番を定義したものです。 標準でdefaultサイクルとcleanサイクル、siteサイクルが用意されています。 先ほどpackageフェーズ実行時に使ったのが *default*サイクル 、cleanフェーズ実行時に使ったのが *clean*サイクル です。

それぞれのビルド・ライフサイクルは1つ以上のフェーズを含んでいます。 これらのフェーズは順番に並んでいて、あるフェーズが実行されるには それ以前のフェーズが実行済み でなければなりません。 言い換えると、すべてのフェーズはそのひとつ前のフェーズに依存しているということです。

defaultライフサイクル

defaultライフサイクルには以下のように並んだフェーズが含まれています。

1. validate （プロジェクトの状態確認）
2. initialize （ビルドの初期化処理）
3. generate-sources （ソースコードの自動生成）
4. process-sources （ソースコードの自動処理）
5. generate-resources （リソースの自動生成）
6. process-resources （リソースの自動処理）
7. compile （プロジェクトのコンパイル）
8. process-classes （classファイルの自動処理）
9. generate-test-sources （テストコードの自動生成）
10. process-test-sources （テストコードの自動処理）
11. generate-test-resources （テスト用リソースの自動生成）
12. process-test-resources （テスト用リソースの自動処理）
13. test-compile （テストコードのコンパイル）
14. process-test-classes （テスト用classファイルの自動処理）
15. test （単体テストの実行）
16. prepare-package （アーティファクト作成の準備）
17. package （アーティファクトの作成）
18. pre-integration-test （インテグレーションテストの前処理）
19. integration-test （インテグレーションテストの実行）
20. post-integration-test （インテグレーションテストの後処理）
21. verify （アーティファクトの検証）
22. install （アーティファクトをローカルリポジトリに配置）
23. deploy （アーティファクトをリモートリポジトリに配置）

先ほどMavenがコンパイルや自動テストを実行したのは、packageフェーズを実行するために必要なvalidateからprepare-packageまでの16のフェーズすべてを実行したためです。 こうしたフェーズの依存関係によって、Mavenユーザはやりたいことだけを伝えるだけで済むのです。

cleanライフサイクル

前述のdefaultライフサイクルにはcleanフェーズが見当たりません。 cleanフェーズはcleanライフサイクルと呼ばれる他のライフサイクルに属しています。 このライフサイクルには3つのフェーズがあります。

1. pre-clean （一時ファイル削除の前処理）
2. clean （一時ファイルの削除）
3. post-clean （一時ファイル削除の後処理）

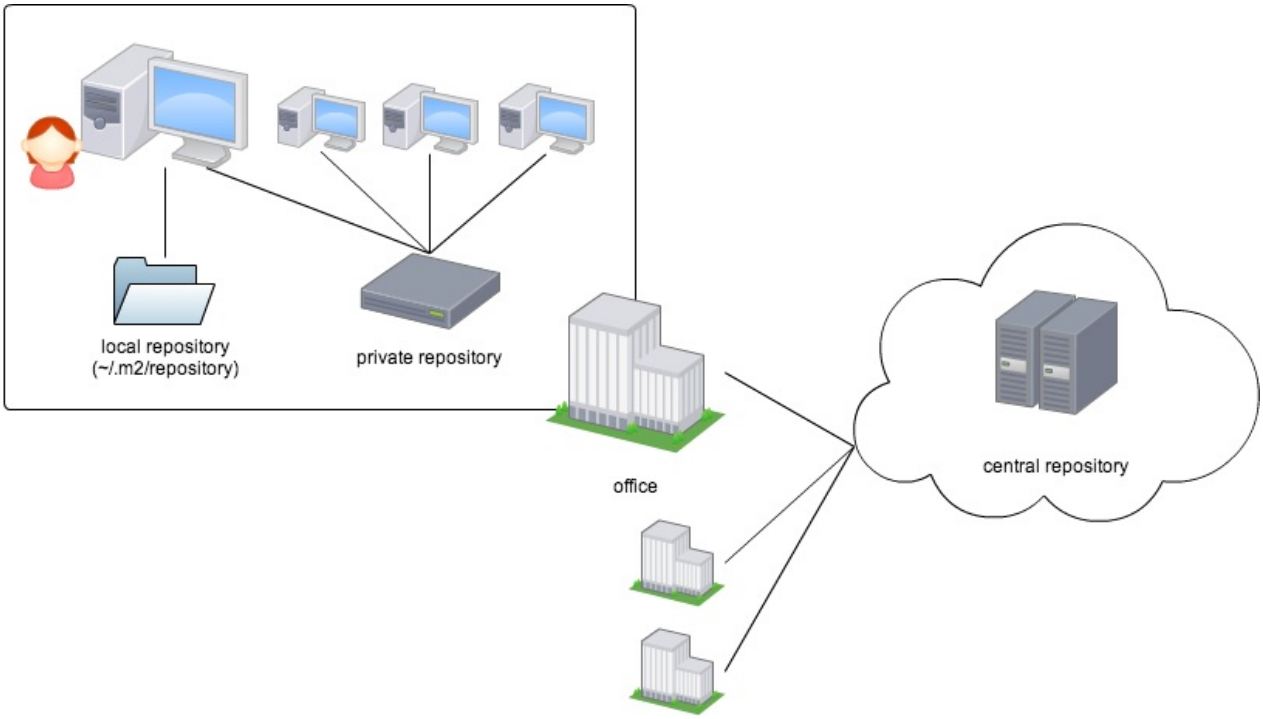
例えば `mvn clean install` と実行すると、まずはcleanライフサイクルが実行され、その次にdefaultライフサイクルが実行されます。 `mvn clean clean` ではcleanライフサイクルが2回実行されます。

pom.xml

プロジェクトのトップディレクトリには通常`pom.xml`という名前の設定ファイルを配置します。 このファイルにはプロジェクトの情報、依存関係、利用するプラグイン、ライセンス、SCMのURLなど様々な情報を記録できます。

Mavenリポジトリ

JARなどの成果物やJavadocをライブラリを整理してまとめておく場所のことを、Mavenリポジトリと呼びます。 セントラルリポジトリ、ローカルリポジトリ、プライベートリポジトリの3種類があります。



セントラルリポジトリとは

インターネットに公開されているリポジトリで、たくさんのライブラリが公開されています。

- Maven central repository

プライベートリポジトリとは

何らかの理由でセントラルリポジトリにライブラリを公開したくない場合、自分でリポジトリを用意して利用することができます。このリポジトリのことをプライベートリポジトリと呼びます。WEBDAVが使えるサーバならなんでもプライベートリポジトリとして使えますが、Apache ArchivaやNexusなどの管理機能を持つウェブアプリケーションを使うと便利です。

なおプライベートリポジトリ以外のリポジトリを表す用語としてリモートリポジトリ（remote repository）があります。

プライベートリポジトリを使うには

プライベートリポジトリをセットアップしたら、pom.xmlに使用するプライベートリポジトリのURLを明記する必要があります。

```
<repository>
  <id>my-repo1</id>
  <name>your custom repo</name>
  <url>http://jarsm2.dyndns.dk</url>
</repository>
```

ローカルリポジトリとは

mvnコマンドを実行したマシンにあるディレクトリのことです。デフォルトでは`~/.m2/repository`が利用されます。他のリポジトリからダウンロードしたライブラリを保管したり、`install`ゴールでJARをインストールしたりするために使われます。

基本的にMavenは、ライブラリを取得ときにまずローカルリポジトリを確認し、そこになかった場合にセントラルリポジトリやプライベートリポジトリを見に行きます。 セントラルリポジトリにもプライベートリポジトリにも公開されていないライブラリを使う場合には、まず `mvn install` でそのライブラリをプライベートリポジトリにインストールしてやるときちんと使うことができます。

依存関係

プロジェクトをビルドするときに、JDKだけでなくライブラリを必要とすることがあります。このことを「プロジェクトはライブラリに依存している」と表現します。 Mavenではプロジェクトがライブラリに依存していることを以下のように明記できます。

```
<dependency><!-- このプロジェクトはJUnit バージョン4.12に依存している -->
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

Mavenプラグイン

プロジェクトをリポジトリに公開する

この章では、プロジェクトをリモートリポジトリに公開するうえで気にしたいことをまとめています。

バージョンの種類と使い分け

メジャーバージョン、マイナーバージョン、パッチバージョン

バージョンは通常、下記のように数値とピリオドを用いて記載します。Mavenでもこの方法に従います。 ピリオドはいくつでも使うことができますが、特に書式にこだわりがない場合、Semantic Versioning 2.0.0に従い左から順にメジャー・マイナー・パッチとしておくとよいでしょう。

- `1.0`
- `1.2.3`

メジャーバージョンは機能に大きな変更があった場合、マイナーバージョンは後方互換性を保つ変更があった場合、パッチバージョンはバグ修正が行われた場合に大きくなります。 これらの数値は `1.10.12` のように2桁以上になることも可能です。

安定バージョン

もし多くの人に使ってほしい安定版を公開するなら、安定バージョンの公開を検討しましょう。 後述する `maven-release-plugin` を使えば、安定バージョンが満たすべき条件を自動的に検証してくれます。

安定バージョンとは、前述のように数値とピリオドだけで構成されたものです。 反面、実装やインタフェースが安定しない開発途上のものを公開するなら、修飾子を用いて安定版でないことを明示すると良いでしょう。

SNAPSHOT修飾子

最も代表的な修飾子が *SNAPSHOT*修飾子 です。

バージョンにSNAPSHOT修飾子を含めることで、使い手とMavenに対して安定版ではないことを伝えることができます。修飾子は以下のように、半角ダッシュを使ってバージョン番号の後ろに付記します。 大文字でも小文字でも構いませんが、慣習として大文字を使用することが多いようです。

- `1.2.3-SNAPSHOT`
- `1.0-snapshot`

artifactは通常1バージョンにつき1度しかdeployできませんが、SNAPSHOT修飾子を利用したバージョンは同じバージョンで何度も異なるartifactをデプロイすることができます。 これにより、利用者に対して 常に最新の開発版を提供することができます 。

その他の修飾子

Maven3は他にもバージョンに特別な意味を持たせるための修飾子を提供しています。[1] 修飾子を不安定な順（古い順）に並べると、以下のとおりです。 RCはリリース候補（Release Candidate）、SPはサービスパック（Service Pack）の略です。

1. `alpha` （短縮表記として `a` を利用可能）
2. `beta` （短縮表記として `b` を利用可能）
3. `milestone` （短縮表記として `m` を利用可能）
4. `rc` or `cr`
5. `snapshot` （前述）

6. `ga` or `final`（修飾子なしと同様）
7. `sp`

例えばバージョン `1.2.3` の開発過程において、以下の順で修飾子を利用することができます。

- `1.2.3-alpha1`, `1.2.3-alpha2`, ... (`1.2.3` のアルファ版)
- `1.2.3-beta1`, `1.2.3-beta2`, ... (`1.2.3` のベータ版)
- `1.2.3-m1`, `1.2.3-m2`, ... (`1.2.3` のマイルストーン版)
- `1.2.3-rc1`, `1.2.3-rc2`, ... (`1.2.3` のリリース候補版)
- `1.2.3-SNAPSHOT` (1.2.3 の最新開発中バージョン)
- `1.2.3` (`1.2.3` 安定版)
- `1.2.3-sp1`, `1.2.3-sp2`, ... (`1.2.3` バグ修正版)

修飾子を活用している事例としては、org.hibernate:hibernate-coreがわかりやすいでしょう。 バージョンアップではALPHA, BETA, CRとFINAL修飾子を用いたリリースを、バグ修正ではパッチバージョンやSP修飾子を用いたリリースを行っています。

ピリオドとダッシュの違い

このページで紹介している命名方法を利用している限り、この違いを意識する必要はありません。 興味のある方は公式ドキュメントをご覧ください。

> [1]. Wikiにはproposalとして書いてあるが、Maven3では]実装済み機能として利用できる ↩

公開先のMavenリポジトリを指定する

公開先のリモートリポジトリはpom.xmlの`distributionManagement`要素によって指定できます。 例として、Sonatypeのoss-parentsから設定部分を引用してみましょう。

```
<!-- quoted from sonarype/oss-parents -->
<distributionManagement>
  <snapshotRepository>
    <id>sonatype-nexus-snapshots</id>
    <name>Sonatype Nexus Snapshots</name>
    <url>https://oss.sonatype.org/content/repositories/snapshots/</url>
  </snapshotRepository>
  <repository>
    <id>sonatype-nexus-staging</id>
    <name>Nexus Release Repository</name>
    <url>https://oss.sonatype.org/service/local/staging/deploy/maven2/</url>
  </repository>
</distributionManagement>
```

このように、スナップショットバージョンとリリースバージョンで利用するリポジトリは分かれています。 チームにのみスナップショットバージョンを公開したり、リリースバージョンとスナップショットバージョンで公開に必要な権限を変えたりといったことができます。

セントラルリポジトリにライブラリを登録する

セントラルリポジトリには誰でもライブラリを公開することができます。詳細は以下のページを参照してください。

- 【最新版】Maven Central Repository へのライブラリ登録方法 #maven

リポジトリへのアクセス方法の指定

Mavenは基本的にWebDAVでのアクセスを行います。FTPやSSHを使用する場合は対応するプラグインの利用が必要です。

リポジトリへの公開

リポジトリにプロジェクトを公開する主な方法として、`deploy`フェーズとmaven-release-pluginの2つがあります。

deployフェーズ

deployフェーズは作成したパッケージをリモートのプライベートリポジトリに公開します。 これはdefaultライフサイクルの最後のフェーズですので、compile, test, packageといった defaultライフサイクルに含まれるすべての他のフェーズが実行されてから実行されます。

```
$ mvn deploy
```

この方法は主にSNAPSHOTバージョンをリポジトリに公開するときに利用します。 安定バージョンを公開するときは、次に説明する`maven-release-plugin`を利用してください。

maven-relase-plugin

`maven-release-plugin`はプライベートリポジトリへの公開だけでなく、バージョン番号をインクリメントする・ VCSのタグやブランチを編集する・依存するライブラリにSNAPSHOTバージョンが含まれていたらリリースを 中止するなど、安定バージョンの公開に便利な機能を提供します。 リモートリポジトリに安定バージョンを公開する場合は積極的に使いましょう。

使い方は、prepareゴールとperformゴールを順番に呼び出すだけです。 リリースするバージョンなどを対話的に指定するだけで自動的に検証・編集・公開を行います。

```
$ mvn release:prepare release:perform
```

権限不足で公開に失敗したら

リポジトリへの公開に権限が必要な場合、リポジトリの管理者に問い合せて権限を取得する必要があります。 多くの場合は`~/.m2/settings.xml`を編集し、ユーザ名やパスワードなどの認証情報を設定します。

Mavenプラグインを実装する

Mavenは歴史が長く、すでに多数のプラグインが提供されています。自分でプラグインを実装しなくても、多くの作業を自動化できます。 よって、多くの読者はこの章を読み飛ばして構いません。

既存プラグインを改善したい、独自ツール用のプラグインを作成したい、新言語用のプラグインを作成したい、 などのより高度な自動化を実現したい技術者は、プラグインを実装することでそれが実現できる可能性があります。 Mavenプラグインは、多様な環境でも統一された手法によって同一の目的を達成するための良い手段になるでしょう。

Mavenプロジェクトを作成する

この節ではMavenプラグインを実装するためのMavenプロジェクトを作成する方法を紹介します。

pom.xmlを設定する

はじめに、`pom.xml`に以下を設定しましょう。

- `<packaging>`要素を`<project>`要素の直下に追加し、値を*maven-plugin*に設定
- *maven-plugin-api*と*maven-plugin-annotations*に*provided*スコープで依存
- `<artifactId>`要素を*(任意の名前)-maven-plugin*に設定
  - 慣習でありで必須ではありませんが、このように命名することでMavenプラグイン実行時に`-maven-plugin`を省略できます。 例えば*jp.skypencil*グループに属する*sample-maven-plugin*なら、`mvn jp.skypencil:sample`で実行できます。
  - 古い資料では*maven-(任意の名前)-plugin*を使うよう推奨していますが、現在は非推奨ですので使用しないでください。
- *maven-plugin-plugin*に`<execution><id>default-descriptor</id></execution><phase>process-classes</phase></execution>`を追記[1]
- *maven-plugin-plugin*に`<execution><id>generate-helpmojo</id><goals><goal>helpmojo</goal></goals></execution>`を追記[2]

`pom.xml`の概要は以下のようになります。

```
<project>
  <artifactId>sample-maven-plugin</artifactId>
  <packaging>maven-plugin</packaging>

  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-plugin-api</artifactId>
      <version>3.2.5</version><!-- version of Maven -->
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.maven.plugin-tools</groupId>
      <artifactId>maven-plugin-annotations</artifactId>
      <version>3.4</version><!-- version of Maven Plugin Tools -->
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-plugin-plugin</artifactId>
        <version>3.4</version>
        <executions>
          <execution>
            <id>default-descriptor</id>
            <phase>process-classes</phase>
          </execution>
          <execution>
            <id>generate-helpmojo</id>
            <goals>
```

```
            <goal>helpmojo</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

なお、archetypeプラグインを利用するとpom.xmlを自動的に生成してくれます[3]ので、 スクラッチで実装を行う場合はぜひ利用してください。以下のコマンドでMavenプロジェクトの作成を行えます。

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-plugin -Dar
```

Eclipseプロジェクトを作成する

Eclipseで開発を行う場合、以下のコマンドでEclipseプロジェクトの作成を行ってください。 作成後、メニューバーの「ファイル→インポート」から既存のEclipseプロジェクトとして取り込むことができます。

```
mvn eclipse:eclipse
```

あるいはEclipseの `m2e` プラグインを使用することで、MavenプロジェクトをEclipseプロジェクトとして 直接開くことも可能です。詳しくは `m2e` の公式サイトをご確認ください。

TODO 上記サイトのURLを調べる。

> [1]. http://maven.apache.org/plugin-tools/maven-plugin-plugin/examples/using-annotations.html ↩

> [2]. ヘルプ表示用Mojoを自動生成するため ↩

> [3]. TODO 1.2は2015年2月時点での最新版だが、長く更新されていないので、新しいアーキタイプを作成すること ↩

Mojoを作成する

Mavenプラグインの実体は、Mojo（Maven plain Old Java Object）と呼ばれるクラスです。Mavenプラグインは含まれるゴールの数だけMojoを保有します。 このクラスに必要なメソッドとMojoアノテーションを追加することでプラグインの実装を行います。

Mojoは`AbstractMojo`を継承し、`@Mojo`アノテーションで修飾される必要があります。 また`@Execute`アノテーションや`@Parameter`アノテーション、`@Component`アノテーションなどorg.apache.maven.plugins.annotationsパッケージに用意されたアノテーションを使用することもできます。

以下にシンプルなMojoの実装を記載します。

```
import java.io.File;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugins.annotations.Execute;
import org.apache.maven.plugins.annotations.LifecyclePhase;
import org.apache.maven.plugins.annotations.Mojo;
import org.apache.maven.plugins.annotations.Parameter;

@Mojo(
        name = "sample",
        threadSafe = true,
        defaultPhase = LifecyclePhase.COMPILE
)
public class SampleMojo extends AbstractMojo {
    @Parameter(
            property = "project.build.directory",
            required = true
    )
    private File outputDirectory;

    @Override
    public void execute() throws MojoExecutionException {
        getLog().info("sample plugin start!");
        getLog().debug("project.build.directory is " + outputDirectory.getAbsolutePath());
    }
}
```

順を追って見ていきましょう。

呼び出されるメソッドを実装する

Mavenプラグインが実行されると、Mojoの`execute`メソッドが実行されます。 プラグインの実際の処理はここに記載します。

実行するフェーズを指定する

`@Mojo`アノテーションの`defaultPhase`オプションによって、実行するフェーズを指定することができます。 この設定はユーザの設定によって上書くことが可能ですが、デフォルト設定で多くの環境で問題なく動作することが望ましいでしょう。

設定を作成する

プラグインは様々なプロジェクトに対応するため、しばしば設定を提供します。 設定は`pom.xml`に記載することができる

ため、同じ設定を複数の環境で容易に共有することができます。 一般的な設定としては以下があります。`

- `<skip>` ... trueならプラグインの実行を行いません。プロジェクト固有のプロパティで指定できることもあります。
- `<outputDirectory>` ... 成果物の出力先ディレクトリです。 `project.build.directory` プロパティが示すディレクトリ、あるいはそのサブディレクトリをデフォルト値として使用するべき[1]です。
- `<encoding>` ... ファイルの読み書きに利用するエンコードです。 `project.build.sourceEncoding` プロパティをデフォルト値として使用できます。

ログを出力する

Mavenプラグインを実装する場合、ログの出力は標準出力や標準エラー出力ではなく、親クラスが提供するロガーを利用します。 これによって、Mavenの `-q` オプション（エラーのみ出力）や `-X` オプション（デバッグログ出力）やログの接頭辞に準拠でき、ログの可読性が向上します。

```
getLog().info( "This is info  level, Maven will print this if user does not use -q option.");
getLog().debug("This is debug level, Maven will print this if user uses -X option.");
getLog().error("This is error level, Maven will print this even if user uses -q option.");
```

処理を高速化するため、デバッグ時にのみ情報を取得・計算することもできます。

```
if (getLog().isDebugEnabled()) {
  getLog().debug(computeDetailedLogMessage());
}
```

なお引数に渡す文字列には、改行コードを含めないことをおすすめします。接頭辞がつかない行ができてしまい、 機械的に処理することが難しくなるためです。複数行の出力を行いたい場合は、メソッドを複数回に分けて 呼び出してください。

マルチスレッド対応の明示

もし作成したプラグインがスレッドセーフである場合は、それを明示することが望ましいと言えます。 Maven3.2.5時点でのマルチスレッドサポートは限定的ですが、これが改善されたときのために用意しておきましょう。

`@Mojo` アノテーションの `threadSafe` プロパティによってスレッドセーフか否かを表現できます。 公式のチェックリストを確認し、スレッドセーフであると言える場合は、 `true` を指定しましょう。

```
@Mojo(
        name = "sample",
        threadSafe = true
)
```

2015年4月時点での公式チェックリストの訳を以下に記載します。

- staticなフィールドや変数を使っている場合、その値がスレッドセーフであることを確認すること。
  - 特に "java.text.Format" のサブクラス (NumeberFormat, DateFormat など) はスレッドセーフでないので、staticなフィールドを介して複数のスレッドから利用されるようなことがないように注意する。
- `components.xml` にシングルトンとして定義されているPlexusコンポーネントを使っている場合、そのコンポーネントはスレッドセーフでなければならない。
- 依存ライブラリに既知の問題が存在しないか確認すること。
- サードパーティ製ライブラリがスレッドセーフであることを確認すること。

つまり、単一JVM内で複数のMojoインスタンスが並列に作成・実行される可能性があるので、それに備える必要があるということです。

動作確認

ここまで来たら、作ったプラグインが正常に動作することを確認してみましょう。 ローカルリポジトリにプラグインをインストールしてから、作成したMojoを実行してみてください。

```
mvn clean install
mvn jp.skypencil:sample-maven-plugin:sample
mvn jp.skypencil:sample-maven-plugin:help
```

[1]. "target"などとハードコードすると、ユーザが `project.build.directory` プロパティを変更した際に動作しなくなります。 ↩

単体テストを作成する

Mavenプラグインの単体テストを作成するために、`maven-plugin-testing-harness`というライブラリが 提供されています。ここではこのライブラリをJUnitと組み合わせて、単体テストを作成する方法を紹介します。

pom.xmlに依存ライブラリを追記する

まずJUnitと`maven-plugin-testing-harness`を使用するために、以下2つの`<dependency>`を`pom.xml`に追加します。

```
<dependencies>
  ...
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.maven.plugin-testing</groupId>
    <artifactId>maven-plugin-testing-harness</artifactId>
    <version>3.3.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

単体テストを作成する

追加したら単体テストの作成を開始できます。`maven-plugin-testing-harness`ではJUnit4の`@Rule`アノテーションを利用しての開発が可能です。 最も単純なテストは以下のようになります。

```
public class SampleMojoTest {
  @Rule
  public MojoRule mojo = new MojoRule();

  @Rule
  public TestResources resources = new TestResources();

  @Test
  public void testMojoHasHelpGoal() throws Exception {
    // src/test/projects/project/pom.xml に書かれた設定を元にMojoインスタンスを作成
    File baseDir = resources.getBasedir("project");
    File pom = new File(baseDir, "pom.xml");

    // 'help' ゴールを実行
    Mojo mojo = mojo.lookupMojo("help", pom);
    mojo.execute();
  }
}
```

`MojoRule`はMojoインスタンスを生成するためのJUnit Ruleです。

`TestResources`は各テストメソッドで固有のリソースを使うためのJUnit Ruleです。`src/test/projects`以下にダミーのMavenプロジェクトが入ったディレクトリを配置しておき、 そのディレクトリ名を`getBasedir()`メソッドに渡すことで、ダミープロジェクトの設定を元に Mojoインスタンスを作成することができます。

つまりこのテストは、ダミープロジェクトの設定を元に作成したMojoインスタンスのメソッドを呼ぶことで プラグインが正常に動作することを確認するためのものです。

期待通りに正常終了することをテストする

前述のとおり、`MojoRule`のインスタンスメソッドを通じて取得した`Mojo`インスタンスの`execute()`メソッドを呼ぶことで、 実際にMavenプラグインを実行できます。`execute()`メソッドが例外を投げずに終了した場合、Mavenプラグインが正常終了したとみなせます。

```
@Test
public void testGoalSucceeds() {
  File baseDir = resources.getBasedir("project");
  File pom = new File(baseDir, "pom.xml");

  Mojo samplePlugin = mojo.lookupMojo("help", pom);
  assertNotNull(samplePlugin);
  samplePlugin.execute();
}
```

テスト内容によっては、実行時にMojoの状態を確認するコードなどを記述する必要があるかもしれません。

期待通りに異常終了することをテストする

設定が異常なときや必要なファイルがないときは、Mavenプラグインが異常終了する必要があります。

JUnitのテストとしては、`execute()`メソッドが`MojoFailureException`あるいは`MojoExecutionException`を投げることを確認するコードを書きます。 次のように`@Test`アノテーションに期待される例外を指定してください。

```
@Test(expected = MojoFailureException.class)
public void testGoalFailsAsExpected() {
  File baseDir = resources.getBasedir("project");
  File pom = new File(baseDir, "pom.xml");

  Mojo samplePlugin = mojo.lookupMojo("help", pom);
  assertNotNull(samplePlugin);
  samplePlugin.execute();
}
```

Wikiによると`MojoExecutionException`は設定に問題がありMojoの実行ができなかったときに、`MojoFailureException`は依存関係やプロジェクトが持つソースコードに問題がありMojoの実行が失敗したときに投げる必要があります。

Javadocに記載されている表現で言い換えると、`MojoExecutionException`はプラグイン提供者が発生を期待しない問題が生じたときにビルドをエラー終了させるために、`MojoFailureException`はプラグイン提供者が期待する問題が生じたときにビルドを失敗させるために使います。 適宜使い分けてください。

ログが正しく呼ばれていることを確認する

Mavenプラグインを使用するユーザは、ログを通じてプラグインからの結果報告や異常通知などを受けます。 このためログが特定の条件下で期待通りに出ることは、ぜひテストで確認・保証しておきたいポイントです。

`Mojo`インタフェースはロガーをセットするメソッドを提供していますので、モックオブジェクトを利用することができます。 以下のコードはMockitoを利用してデバッグログの出力内容を確認するものです。

```
@Test
public void testSampleGoalPrintsOutputDirectory() throws Exception {
  File baseDir = resources.getBasedir("simple");
  File pom = new File(baseDir, "pom.xml");
  Log log = Mockito.mock(Log.class);
```

```
    Mojo samplePlugin = mojo.lookupMojo("sample", pom);
    samplePlugin.setLog(log);
    samplePlugin.execute();
    Mockito.verify(log).debug("outputDirectory is /tmp/target");
  }
```

以上で紹介したように、Mavenプラグインは簡単に単体テストによってテストできます。 複数の動作環境で動作することを保証する意味でも、ソースコードの変更によるバグ混入を未然に防ぐ意味でも、自動テストはプラグイン開発に有用です。 JenkinsのマルチJ構成プロジェクトやJUnitのTheoryと組み合わせるなどして、機能の安定提供に役立ててください。

Mavenリポジトリを通じて配布する

プラグインは通常の成果物と同様、`deploy`フェーズや`maven-release-plugin`でリモートリポジトリにデプロイできます。ユーザに使ってもらうバージョンは`SNAPSHOT`バージョンではなく安定バージョンにするよう心がけましょう。

配布したプラグインを利用する

pom.xmlに利用するプラグインを記載することで、Mavenが自動的にプラグインをダウンロードして利用するようになります。 プラグインがプライベートリポジトリにある場合、`<pluginRepositories>`要素をpom.xmlの`<build>`直下に追加して利用するリポジトリを明示する必要があります。

```
<pluginRepositories>
  <pluginRepository>
    <id>private-repository</id>
    <name>Private Repository</name>
    <url>http://repository.skypencil.jp/</url>
  </pluginRepository>
</pluginRepositories>
```

参考になるプラグイン

以下のプラグインは規模も大きくなく簡単に読むことができます。実装の際に参考にしてください。

- sample-maven-plugin

モジュール

この章では、中〜大規模のMavenプロジェクトをモジュールによって整理する方法についてまとめています。

モジュールでプロジェクトに構造を持ち込む

ある程度の規模になってくると、プロジェクトに構造と制約を持ち込んだほうが開発の効率が上がります。

Javaはクラスをまとめるための仕組みとしてパッケージを提供していますが、もう少し大きな枠組（コンポーネント単位、 サービス単位など）での構造化をしようとするとプロジェクトを分ける必要性が出てきます。

これを普通にやると分割されたプロジェクトの数だけVCSのリポジトリを用意して、pom.xmlを用意して、プロジェクト間の関係を 依存関係として記述して……というかなり複雑かつ管理が難しい方法を採ることになってしまいます。これではプロジェクトの間に 依存ライブラリの不整合が生じたり、ビルド手順が複雑化してメンテナンスが難しくなったりするでしょう。

例としてcore, batch, frontの3つにプロジェクトを分けた場合を考えましょう。ビルド手順は以下のようになります。 どの順番でビルドすれば良いのかをきちんと覚えておかないと、コンパイルエラーになったりひとつ古いバージョンに依存した状態で テストしてしまったりといったトラブルが予想できます。

```
cd go/to/workspace
svn co http://repository/project-core && cd project-core && mvn install
cd go/to/workspace
svn co http://repository/project-batch && cd project-batch && mvn install
cd go/to/workspace
svn co http://repository/project-front && cd project-front && mvn install
```

Mavenは モジュール という、パッケージよりも大きくプロジェクトよりも小さい構造を提供しています。 モジュールの特徴は以下のとおりです。

- プロジェクトに含まれるすべてのモジュールで設定を統一できます。
- バージョン
- リモートリポジトリの場所
- ライセンス
- 利用プラグインのバージョンと設定
- 依存ライブラリのバージョンとスコープ
- etc.
- 関連モジュールを一度にビルドできます。
- Mavenはモジュール間の依存関係を自動的に解析し、モジュールのビルド順を決定
- プロジェクトに含まれるすべてのモジュールから必要なモジュールだけを取り出してビルド可能
- プラグインの適用範囲を制限できます。
- プラグインはモジュール単位で作用するため、プロジェクトの一部分のみ
- 例えば1プロジェクトからjarファイル・warファイル・earファイルを複数個生成可能
- プロジェクト全体に作用するプラグインも存在（aggregateパラメータを利用したpmd:pmdなど）

例えば2013年9月現在、Jenkinsはcore,maven-plugin,testといった合計7つのモジュールで構成されています。

マルチモジュール構成プロジェクト用実行時オプション

マルチモジュール構成のプロジェクトにおける`mvn`コマンドの使い方を紹介していきます。

すべてのモジュールをビルドする

すべてのモジュールをinstallしたい場合は、普段どおりで構いません。モジュールのビルド順はMavenが計算してくれます。

```
mvn install
```

特定のモジュールだけをビルドする

`--projects`オプションを使うことで、特定のモジュールだけをビルドすることができます。 例えば次のコマンドはcoreモジュールだけをinstallします。

```
mvn install --projects core
```

なお`--projects`の代わりに`-pl`と短縮して書くことも可能です。

指定モジュールが依存しているモジュールもすべてビルドする

`--projects`オプションだけでは指定モジュールが依存しているモジュールはビルドされないため、それらの最新版を使っての ビルドにはなりません。依存モジュールのビルドも必要な場合には、`--also-make`オプションを利用しましょう。Mavenはモジュール間の依存関係を自動的に計算して、依存モジュールをどの順にビルドするべきかも判断してくれます。

例えばcoreモジュールとそれが依存するモジュールだけinstallしたい場合は、以下のようにコマンドを実行します。

```
mvn install --also-make --projects core
```

なお`--also-make`は`-am`と短縮して書くことも可能です。

指定モジュールに依存しているモジュールもすべてビルドする

自分がモジュールに加えた変更が他のモジュールに悪影響を与えていないことを確認するために、変更したモジュールに依存するすべてのモジュールをビルドしたいこともあるでしょう。その場合は`--also-make-dependents`を使用します。

```
mvn install --also-make-dependents --projects core
```

なお`--also-make-dependents`は`-amd`と短縮して書くことも可能です。

途中のモジュールからビルドを再開（**--resume-from**オプショ

ン）

ビルドが失敗してそれを修正した場合、ビルドを最初からやり直す必要はありません。修正したモジュールからビルドを再開することで、 ビルドにかかる時間を短縮することができます。ビルドを開始したいモジュールを`--resume-from`で指定してください。

```
mvn install --also-make --projects core --resume-from test
```

なお`--resume-from`は`-rf`と短縮して書くことも可能です。

並列ビルド（**--threads**オプション）

Maven3では複数モジュールを同時にビルドする`--threads`オプションが提供されています。 非対応のプラグインがあること、実験的な実装であることに注意が必要ですが、CPUがボトルネックになっている ビルドを高速化したい場合には検討できるかもしれません。

- Parallel builds in Maven 3

困ったときの逆引き

この章ではMaven利用時にありがちな問題について、調査方法と解決方法を説明します。

環境によって結果が異なる場合

実行環境によって結果が異なる場合、まずはJDKのバージョンやLocaleなどに環境差がないかどうか確認します。 以下のコマンドで環境依存の情報を一括確認できますので、利用してください。

```
mvn -v
```

またファイルパスによる問題なら、以下も確認してください。

- Windowsではバックスラッシュ（または円記号）、UNIX系ではスラッシュを[ファイル区切り文字][1]に利用します。Mavenの設定では一括してスラッシュを利用して構いませんが、必要に応じて `${file.separator}` を利用できます。
- Windowsではファイルのパスが長すぎると問題になることがあります。 Javaはパッケージとフォルダ階層を等しくする必要があるため、この制限に引っかかる可能性があります。 プロジェクトはドライブ直下など、パスが浅い場所に作成するようにしてください。

これでも解決できない場合は、Mavenの設定に相違がないか確認するとよいでしょう。Mavenは実行環境に応じて 設定を切り替える機能を提供しているため、これによって結果が異なってしまっているのかもしれません。 Mavenは原因究明に便利なプラグインを提供していますので、以下に紹介します。

まずはmaven-help-pluginのactive-profilesゴールです。 これは実行時に有効化されているMavenプロファイルを一覧で表示してくれます。有効化されているMavenプロファイルに差があれば、そこに原因があるかもしれません。 このゴールを利用するには、以下のように `mvn` コマンドを実行します：

```
mvn org.apache.maven.plugins:maven-help-plugin:2.2:active-profiles
```

次にmaven-help-pluginのeffective-pomゴールです。 `pom.xml` は継承や `settings.xml` やMavenプロファイルなど、様々な要因によって要素が書き換えられますが、 このゴールはそうした変動要因を考慮した、最終的な設定を出力してくれます。このゴールを利用するには、以下のように `mvn` コマンドを実行します：

```
mvn org.apache.maven.plugins:maven-help-plugin:2.2:effective-pom
```

最後にmaven-help-pluginのeffective-settingsゴールです。 これはMavenリポジトリのミラーサイトなど、 `pom.xml` に書かれていない（プロジェクトに依存しない）設定を参照するために使用します。 このゴールを利用するには、以下のように `mvn` コマンドを実行します：

```
mvn org.apache.maven.plugins:maven-help-plugin:2.2:effective-settings
```

この他にもmaven-help-pluginは 環境依存の原因究明に便利なゴールを多数提供していますので、一度見ておくとよいでしょう。

Mavenプラグインに設定がきちんと渡っているか不安な場合

`mvn` コマンドを実行するときに、`-X` オプションを加えてみてください。 デバッグログが出力され、各プラグインがどのような設定を受け取ったか表示されるようになります。

デバッグログの出力はビルドの実行を低速化させ、ログファイルを大きくしますので、原因究明するときにのみ 使うようにするとよいでしょう。

ClassNotFoundExceptionが出る場合

TODO

> [1]. ディレクトリ名を区切るための文字のこと。Windowsの場合は`C:\\Foo\Bar\Baz.txt`のようにバックスラッシュで区切り、UNIX系の場合は`/tmp/Foo/Bar/Baz.txt`のようにスラッシュで区切ります。 ↩

サンプルプロジェクト

1. 単純なMavenプロジェクト
2. 独自Mavenプラグインの実装