Пять секретов... Apache Maven

Советы по управлению жизненным циклом проекта с помощью Maven

Стивен Хейнс (steve@javasrc.com)
основатель и генеральный директор
GeekCap Inc.

29.10.2012

С профилями вы, конечно, знакомы, но известно ли вам, что их можно использовать в Maven для решения конкретных задач в различных инструментальных средах? Эта статья из цикла *Пять секретов* содержит пять советов, которые выходят за рамки функции сборки Maven и даже за рамки его основных инструментов для управления жизненным циклом проекта. Они позволят вам повысить производительность труда и облегчат управление приложениями в Maven.

Об этом цикле статей

Вы думаете, что знаете о Java-программировании все? На самом деле большинство разработчиков только скребет по поверхности платформы Java, изучив ее лишь настолько, чтобы выполнять свою работу. В этой постоянной рубрике эксперты по Java-технологии углубляются в недра функциональных возможностей платформы Java, предлагая советы и приемы, которые помогут решить самые заковыристые задачи программирования.

Maven — это отличный инструмент для Java™-разработчиков, который можно использовать и для управления жизненным циклом проектов. Как инструмент управления жизненным циклом Maven работает с этапами, а не «задачами» сборки, как Ant. Maven управляет всеми этапами жизненного цикла проекта, включая валидацию, генерацию кода, компиляцию, тестирование, упаковку, тестирование интеграции, верификацию, установку, развертывание, а также создание и развертывание сайта проекта.

Чтобы почувствовать разницу между Maven и традиционным инструментарием сборки, рассмотрим процесс создания JAR- и EAR-файлов. Чтобы собрать каждый артефакт в Ant, необходимо определить конкретные задачи. Maven же делает за вас большую часть этой работы: вы просто говорите ему, должен ли проект быть JAR- или EAR-файлом, а затем поручаете выполнить этап «упаковки». Maven находит необходимые ресурсы и создает файлы.

Существует множество руководств для начинающих по работе с Maven, некоторые из которых перечислены в разделе Ресурсы к этой статье. Здесь же приведены пять советов, которые помогут сделать следующий шаг: написать сценарии с использованием Maven для управления жизненным циклом приложений.

1. Исполняемые JAR-файлы

Развить навыки по этой теме

Этот материал — часть knowledge path для развития ваших навыков. Смотри Стать Java-программистом

Создать JAR-файл с помощью Maven довольно легко: достаточно определить упаковку проекта как jar и выполнить этап упаковки. Гораздо труднее определить исполняемый JAR-файл. Для этого нужно выполнить следующие шаги.

1. Определите в файле MANIFEST.MF архива JAR класс `main`, определяющий исполняемый класс. (MANIFEST.MF — это файл, который Maven генерирует при упаковке приложения.)
2. Найдите все библиотеки, от которых зависит ваш проект.
3. Включите эти библиотеки в файл MANIFEST.MF, так чтобы классы вашего приложения могли найти их.

Все это можно сделать вручную или, что более эффективно, с помощью двух плагинов Maven: `maven-jar-plugin` и `maven-dependency-plugin`.

maven-jar-plugin

`maven-jar-plugin` делает много вещей, но здесь мы остановимся на его использовании для изменения содержимого файла MANIFEST.MF по умолчанию. Добавьте в разделе плагинов своего файла POM код, приведенный в листинге 1.

Листинг 1. Использование maven-jar-plugin для изменения файла MANIFEST.MF

```
<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>
```

Все плагины Maven показывают свою конфигурацию через элемент `<configuration>`. В этом примере `maven-jar-plugin` изменяет его атрибут `archive` и, в частности, атрибут `manifest`, который управляет содержимым файла MANIFEST.MF. В него входят три элемента:

- `addClassPath`: установка значения *true* этого элемента предписывает `maven-jar-plugin` добавить в файл MANIFEST.MF элемент `Class-Path` и включить в этот элемент `Class-Path` все зависимости.
- `classpathPrefix`: если вы планируете включить все зависимости в тот же каталог, что и создаваемый JAR-файл, то этот элемент можно опустить; в противном случае используйте `classpathPrefix` для указания префиксов всех зависимых JAR-файлов. `classpathPrefix` в листинге 1 указывает, что все зависимости должны быть расположены в папке `lib`, заданной относительно архива.
- `mainClass`: этот элемент используется для определения имени класса, который выполняется при запуске пользователем JAR-файла командой `java -jar`.

maven-dependency-plugin

Настроив файл MANIFEST.MF с помощью этих трех элементов, можно переходить к следующему шагу — копированию всех зависимостей в папку `lib`. Для этого воспользуйтесь плагином `maven-dependency-plugin`, как показано в листинге 2.

Листинг 2. Использование maven-dependency-plugin для копирования зависимостей в lib

```
            <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` имеет цель `copy-dependencies`, которая копирует зависимости в выбранный каталог. В этом примере я скопировал зависимости в подкаталог `lib` каталога `build` (`project-home/target/lib`).

Имея зависимости и измененный MANIFEST.MF, можно запустить приложение с помощью простой команды:

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

2. Настройка MANIFEST.MF

Плагин `maven-jar-plugin` позволяет изменять общие части файла `MANIFEST.MF`, но иногда нужна более глубокая настройка MANIFEST.MF. Решение состоит из двух частей.

1. Определите все свои специальные конфигурации в файле "шаблона" MANIFEST.MF.
2. Настройте `maven-jar-plugin` на использование файла MANIFEST.MF и введите в него любые настройки Maven.

В качестве примера рассмотрим JAR-файл, содержащий агент Java. Чтобы выполнить агент Java, необходимо определить `Premain-Class` и разрешения. В листинге 3 показано содержимое такого файла MANIFEST.MF.

Листинг 3. Определение Premain-Class в специализированном файле MANIFEST.MF

```
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
```

В листинге 3 указано, что агенту `Premain-Classcom.geekcap.openapm.jvm.agent.Agent` разрешается переопределять и преобразовывать классы. Далее, обновляем `maven-jar-plugin`, включив файл MANIFEST.MF, как показано в листинге 4.

Листинг 4. Включение Premain-Class

```
            <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 стал одним из самых популярных и широко используемых инструментов с открытым исходным кодом для управления жизненным циклом Java-приложений. Версия Maven 3, предложенная в сентябре 2010 года в виде alpha 5, привнесла в Maven некоторые долгожданные изменения. В разделе Ресурсы можно узнать, что нового появилось в Maven 3.

Это интересный пример, потому что в нем определен `Premain-Class`, который позволяет JAR-файлу работать в качестве агента Java, и в то же время содержится `mainClass`, который позволяет JAR-файлу быть исполняемым. В этом конкретном примере я использовал `OpenAPM` (созданный мной инструмент для отслеживания кода), чтобы определить трассировку кода, которая будет записана агентом Java, и пользовательский интерфейс,

который облегчает анализ записанных трассировок. Короче говоря, пример демонстрирует возможности объединения явного файла манифеста с динамическими изменениями.

3. Деревья зависимостей

Одна из наиболее полезных функций Maven — поддержка управления зависимостями: вы просто определяете библиотеки, от которых зависит ваше приложение, а Maven находит их (в локальном или центральном хранилище), загружает и использует для компиляции кода.

В некоторых случаях может потребоваться знание происхождения определенной зависимости — например, если в сборке оказались разные и несовместимые версии одного и того же JAR-файла. В этом случае нужно предотвратить включение в сборку одной версии JAR-файла, но сначала найти зависимости, удерживающие JAR.

Поиск зависимостей оказывается на удивление простым делом, если знать следующую команду:

```
mvn dependency:tree
```

Аргумент `dependency:tree` отображает все прямые зависимости, а затем показывает все подзависимости (а также их подзависимости и т.д.). Например, в листинге 5 приведен отрывок из клиентской библиотеки, требуемой одной из моих зависимостей.

Листинг 5. Дерево зависимостей Maven

```
[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
```

Из листинга 5 видно, что проекту `sis-client` требуются библиотеки `woodstox-core-lgpl` и `easymockclassextension`. Библиотеке `easymockclassextension`, в свою очередь, требуются библиотеки `cglib-nodep` и `objenesis`. В случае возникновения проблем с `objenesis`, таких как наличие двух версий, 1.2 и 1.3, это дерево зависимостей показало бы, что артефакт 1.2 импортирован косвенным путем библиотекой `easymockclassextension`.

Аргумент `dependency:tree` сэкономил мне много часов диагностики ошибочной сборки; надеюсь, что и для вас он сделает то же самое.

4. Использование профилей

Наиболее значительные проекты содержат, по крайней мере, базовую группу инструментальных сред, решающих задачи, связанные с разработкой, контролем качества

(QA), интеграцией и производством. Задача управления всеми этими средами заключается в настройке сборки, которая должна подключаться к нужной базе данных, выполнять нужный набор сценариев и развертывать в каждой среде все необходимые артефакты. С помощью профилей Maven можно делать все это без необходимости составлять четкие инструкции для каждой среды индивидуально.

Ключом служит объединение профилей всех систем с проблемно-ориентированными профилями. Каждый профиль системы определяет конкретные места, сценарии и серверы. Так, в своем файле pom.xml я определяю проблемно-ориентированный профиль `deploywar`, как показано в листинге 6.

Листинг 6. Профиль развертывания

```
    <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>
```

Этот профиль, определяемый идентификатором `deploywar`, выполняет плагин `tomcat-deployer-plugin`, настроенный на подключение к определенному узлу, порту и к определенным учетным данным (имя пользователя/пароль). Вся эта информация задается с помощью переменных, таких как `${deploymentmanagerRestHost}`. Эти переменные определены в моем файле profiles.xml для каждой инструментальной среды, как показано в листинге 7.

Листинг 7. profiles.XML

```
        <!-- 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>
```

Развертывание профилей Maven

В файле profiles.xml в листинге 7 я определил два профиля и активизировал их на основе значения свойства (среды) `env`. Если свойству `env` присвоить значение `dev`, то будет использоваться информация для развертывания среды разработки. Если свойству `env` присвоить значение `qa`, то будет использоваться информация для развертывания среды QA, и т.д.

Вот команда для развертывания файла:

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

Флаг `-Pdeploywar` предписывает Maven явно включить профиль `deploywar`. Оператор `-Denv=dev` создает системное свойство с именем `env` и присваивает ему значение `dev`, которое активирует конфигурацию среды разработки. Передача `-Denv=qa` активировала бы конфигурацию среды QA.

5. Специальные плагины Maven

Maven предоставляет в распоряжение программиста десятки готовых плагинов, но иногда может потребоваться какой-нибудь специальный плагин. Создать специальный плагин в Maven легко.

1. Создайте новый проект, настроив упаковку POM на `maven-plugin`.
2. Включите вызов `maven-plugin-plugin`, определяющий цели вашего плагина.
3. Создайте класс плагина Maven `mojo` (класс, расширяющий `AbstractMojo`).
4. Добавьте комментарии Javadoc для этого класса, чтобы определить цели, и для переменных, которые будут служить в качестве параметров конфигурации.
5. Реализуйте метод `execute()`, который будет вызываться при вызове плагина.

В качестве примера в листинге 8 показаны соответствующие части специального плагина для развертывания Tomcat.

Листинг 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;

/**
 * Цель, развертывающая Web-приложение в Tomcat
 *
 * @goal deploy
 * @phase install
 */
public class TomcatDeployerMojo extends AbstractMojo
{
 /**
  * Имя хоста или IP-адрес сервера развертывания
  *
  * @parameter alias="host" expression="${deploy.host}" @required
  */
 private String serverHost;

 /**
  * Порт сервера развертывания
  *
  * @parameter alias="port" expression="${deploy.port}" default-value="58020"
  */
 private String serverPort;

 /**
  * Имя пользователя для подключения к менеджеру развертывания (если этот
  * параметр опущен, то плагин попытается развернуть приложение на сервере
  * без учетных данных)
  *
  * @parameter alias="username" expression="${deploy.username}"
  */
 private String username;

 /**
  * Пароль для указанного имени пользователя
  *
```

```
  * @parameter alias="password" expression="${deploy.password}"
  */
 private String password;

 /**
  * Имя исходного артефакта для развертывания, например, target/pos.war
  *
  * @parameter alias="artifactSource" expression=${deploy.artifactSource}"
  * @required
  */
 private String artifactSource;

 /**
  * Имя места назначения для развертывания артефакта, например, ROOT.war.
  * При его отсутствии используется имя исходного артефакта (без
  * информации о пути)
  *
  * @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 );

     // Проверка полей
     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" );
     }

        ...
   }
}
```

В заголовке класса комментарий `@goal` определяет цель, которую выполняет этот MOJO, а `@phase` определяет этап, на котором выполняется цель. Каждое указанное свойство имеет аннотацию `@parameter`, которая содержит псевдоним, по которому этот параметр будет выполняться, а также выражение, которое указывает на системное свойство, содержащее фактическое значение. Если свойство имеет аннотацию `@required`, то оно обязательно. Если оно имеет аннотацию `default-value`, то это значение будет использоваться по умолчанию. В методе `execute()` можно вызвать `getLog()`, чтобы получить доступ к регистратору событий Maven, который в зависимости от уровня регистрации выводит указанное сообщение на стандартное устройство вывода. В случае ошибки плагина сообщение `MojoExecutionException` приведет к прерыванию процесса сборки.

Заключение

Maven можно использовать только для сборки, но в своем лучшем проявлении это инструмент управления жизненным циклом проекта. Эта статья знакомит читателя с пятью

малоизвестными функциями, которые помогают использовать Maven с большей пользой. Подробнее о Maven см. в разделе Ресурсы.

Далее в цикле *Пять секретов* будет дано пять советов по созданию красивых пользовательских интерфейсов в Swing, так что следите за новыми выпусками.

Об авторе

Стивен Хейнс

Стивен Хайнс (Steven Haines) - технический архитектор ioko и основатель компании GeekCap. Написал три книги по Java-программированию и анализу производительности, а также несколько сотен статей и десяток официальных технических документов. Выступал на таких отраслевых конференциях, как JBoss World и STPCon; преподавал Java-программирование в Калифорнийском университете в Ирвине и в Learning Tree University. Проживает в пригороде Орландо (штат Флорида, США).