Maven 是为 Java™ 开发人员提供的一个极为优秀的构建工具，您也可以使用它来管理您的项目生命周期。作为一个生命周期管理工具，Maven 是基于阶段操作的，而不像 Ant 是基于 “任务” 构建的。Maven 完成项目生命周期的所有阶段，包括验证、代码生成、编译、测试、打包、集成测试、安装、部署、以及项目网站创建和部署。

为了更好地理解 Maven 和传统构建工具的不同，我们来看看构建一个 JAR 文件和一个 EAR 文件的过程。使用 Ant，您可能需要定义专有任务来组装每个工件。另一方面，Maven 可以为您完成大部分工作：您只需要告诉它是一个 JAR 文件还是一个 EAR 文件，然后指示它来完成 “打包” 过程。Maven 将会找到所需的资源，然后构建文件。

在本文的 [参考资料](https://www.ibm.com/developerworks/cn/java/j-5things13/#resources) 部分，您将发现大量介绍 Maven 的入门教程。本文的 5 个技巧目的是帮助您解决即将出现的一些问题：使用 Maven 管理您的应用程序的生命周期时，将会出现的编程场景。

**1. 可执行的 JAR 文件**

使用 Maven 构建一个 JAR 文件比较容易：只要定义项目包装为 “jar”，然后执行包装生命周期阶段即可。但是定义一个可执行 JAR 文件却比较麻烦。采取以下步骤可以更高效：

1. 在您定义可执行类的 JAR 的 MANIFEST.MF 文件中定义一个 main 类。（MANIFEST.MF 是包装您的应用程序时 Maven 生成的。）
2. 找到您项目依赖的所有库。
3. 在您的 MANIFEST.MF 文件中包含那些库，便于您的应用程序找到它们。

您可以手工进行这些操作，或者要想更高效，您可以使用两个 Maven 插件帮助您完成：maven-jar-plugin 和 maven-dependency-plugin。

**maven-jar-plugin**

maven-jar-plugin 可以做很多事情，但在这里，我们只对使用它来修改默认 MANIFEST.MF 文件的内容感兴趣。在您的 POM 文件的插件部分添加清单 1 所示代码：

**清单 1. 使用 maven-jar-plugin 修改 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>

所有 Maven 插件通过一个 <configuration> 元素公布了其配置，在本例中，maven-jar-plugin 修改它的 archive 属性，特别是存档文件的manifest 属性，它控制 MANIFEST.MF 文件的内容。包括 3 个元素：

* addClassPath：将该元素设置为 *true* 告知 maven-jar-plugin 添加一个 Class-Path 元素到 MANIFEST.MF 文件，以及在 Class-Path 元素中包括所有依赖项。
* classpathPrefix：如果您计划在同一目录下包含有您的所有依赖项，作为您将构建的 JAR，那么您可以忽略它；否则使用classpathPrefix 来指定所有依赖 JAR 文件的前缀。在清单 1 中，classpathPrefix 指出，相对存档文件，所有的依赖项应该位于 “lib” 文件夹。
* mainClass：当用户使用 lib 命令执行 JAR 文件时，使用该元素定义将要执行的类名。

**maven-dependency-plugin**

当您使用这 3 个元素配置好了 MANIFEST.MF 文件之后，下一步是将所有的依赖项复制到 lib 文件夹。为此，使用 maven-dependency-plugin，如清单 2 所示：

**清单 2. 使用 maven-dependency-plugin 将依赖项复制到库**

<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 有一个 copy-dependencies，目标是将您的依赖项复制到您所选择的目录。本例中，我将依赖项复制到 build 目录下的 lib 目录（project-home/target/lib）。

将您的依赖项和修改的 MANIFEST.MF 放在适当的位置后，您就可以用一个简单的命令启动应用程序：

java -jar jarfilename.jar

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-5things13/#ibm-pcon)

**2. 定制 MANIFEST.MF**

虽然 maven-jar-plugin 允许您修改 MANIFEST.MF 文件的共有部分，但有时候您需要一个更个性化的 MANIFEST.MF。解决方案是双重的：

1. 在一个 “模板” MANIFEST.MF 文件中定义您的所有定制配置。
2. 配置 maven-jar-plugin 来使用您的 MANIFEST.MF 文件，然后使用一些 Maven 配置增强。

例如，考虑一个包含 Java 代理的 JAR 文件。要运行一个 Java 代理，需要定义 Premain-Class 和设置许可。清单 3 展示了这样一个 MANIFEST.MF 文件的内容：

**清单 3. 在一个定制的 MANIFEST.MF 文件中定义 Premain-Class**

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

在 [清单 3](https://www.ibm.com/developerworks/cn/java/j-5things13/#listing3) 中，我已指定 Premain-Class - com.geekcap.openapm.jvm.agent.Agent 被授权许可来对类进行重定义和再转换。接下来，我更新maven-jar-plugin 代码来包含 MANIFEST.MF 文件。如清单 4 所示：

**清单 4. 包含 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 已确立了它作为一种最流行和最易使用的开源 Java 生命周期管理工具的地位。Maven 3，2010 年 9 月升级为 alpha 5，带来一些期待已久的改进。在 [参考资料](https://www.ibm.com/developerworks/cn/java/j-5things13/#resources)部分寻找 Maven 的新功能。

这是一个很有趣的示例，因为它既定义了一个 Premain-Class — 允许 JAR 文件作为一个 Java 代理运行，也有一个 mainClass — 允许它作为一个可执行的 JAR 文件运行。在这个特殊的例子中，我使用 OpenAPM（我已构建的一个代码跟踪工具）来定义将被 Java 代理和一个用户界面记录的代码跟踪。简而言之，这个示例展示一个显式清单文件与动态修改相结合的力量。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-5things13/#ibm-pcon)

**3. 依赖项树**

Maven 一个最有用的功能是它支持依赖项管理：您只需要定义您应用程序依赖的库，Maven 找到它们、下载它们、然后使用它们编译您的代码。

必要时，您需要知道具体依赖项的来源 — 这样您就可以找到同一个 JAR 文件的不同版本的区别和矛盾。这种情况下，您将需要防止将一个版本的 JAR 文件包含在您的构建中，但是首先您需要定位保存 JAR 的依赖项。

一旦您知道下列命令，那么定位依赖项往往是相当容易的：

mvn dependency:tree

dependency:tree 参数显示您的所有直接依赖项，然后显示所有子依赖项（以及它们的子依赖项，等等）。例如，清单 5 节选自我的一个依赖项所需要的客户端库：

**清单 5. Maven 依赖项树**

[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

在 [清单 5](https://www.ibm.com/developerworks/cn/java/j-5things13/#listing5) 中您可以看到 sis-client 项目需要 woodstox-core-lgpl 和 easymockclassextension 库。easymockclassextension 库反过来需要 cglib-nodep 库和 objenesis 库。如果我的 objenesis 出了问题，比如出现两个版本，1.2 和 1.3，那么这个依赖项树可能会向我显示，1.2 工件是直接由 easymockclassextension 库导入的。

dependency:tree 参数为我节省了很多调试时间，我希望对您也同样有帮助。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-5things13/#ibm-pcon)

**4. 使用配置文件**

多数重大项目至少有一个核心环境，由开发相关的任务、质量保证（QA）、集成和生产组成。管理所有这些环境的挑战是配置您的构建，这必须连接到正确的数据库中，执行正确的脚本集、并为每个环境部署正确的工件。使用 Maven 配置文件让您完成这些任务，而无需为每个环境分别建立明确指令。

关键在于环境配置文件和面向任务的配置文件的合并。每个环境配置文件定义其特定的位置、脚本和服务器。因此，在我的 pox.xml 文件中，我将定义面向任务的配置文件 “deploywar”，如清单 6 所示：

**清单 6. 部署配置文件**

<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>

这个配置文件（通过 ID “deploywar” 区别）执行 tomcat-deployer-plugin，被配置来连接一个特定主机和端口，以及指定用户名和密码证书。所有这些信息使用变量来定义，比如 ${deploymentmanagerRestHost}。这些变量在我的 profiles.xml 文件中定义，如清单 7 所示：

**清单 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>

**部署 Maven 配置文件**

在 [清单 7](https://www.ibm.com/developerworks/cn/java/j-5things13/#listing7) 的 profiles.xml 文件中，我定义了两个配置文件，并根据 env （环境）属性的值激活它们。如果 env 属性被设置为 dev，则使用开发部署信息。如果 env 属性被设置为 qa，那么将使用 QA 部署信息，等等。

这是部署文件的命令：

mvn -Pdeploywar -Denv=dev clean install

-Pdeploywar 标记通知要明确包含 deploywar 配置文件。-Denv=dev 语句创建一个名为 env 的系统属性，并将其值设为 dev，这激活了开发配置。传递 -Denv=qa 将激活 QA 配置。

[**回页首**](https://www.ibm.com/developerworks/cn/java/j-5things13/#ibm-pcon)

**5. 定制 Maven 插件**

Maven 有十几个预构建插件供您使用，但是有时候您只想找到自己需要的插件，构建一个定制的 Maven 插件比较容易：

1. 用 POM packaging 创建一个新项目，设置为 “maven-plugin”。
2. 包括一个 maven-plugin-plugin 调用，可以定义您的公布插件目标。
3. 创建一个 Maven 插件 “mojo” 类 （一个扩展 AbstractMojo 的类）。
4. 为类的 Javadoc 做注释来定义目标，并为每个将被作为配置参数公布的变量做注解。
5. 实现一个 execute() 方法，该方法在调用您的插件是将被调用。

例如，清单 8 显示了一个定制插件（为了部署 Tomcat）的相关部分：

**清单 8. TomcatDeployerMojo.java**

package net.fpic.maven.plugins;

import java.io.File;

import java.util.StringTokenizer;

import net.fpic.tomcatservice64.TomcatDeploymentServerClient;

import org.apache.maven.plugin.AbstractMojo;

import org.apache.maven.plugin.MojoExecutionException;

import com.javasrc.server.embedded.CommandRequest;

import com.javasrc.server.embedded.CommandResponse;

import com.javasrc.server.embedded.credentials.Credentials;

import com.javasrc.server.embedded.credentials.UsernamePasswordCredentials;

import com.javasrc.util.FileUtils;

/\*\*

\* Goal that deploys a web application to Tomcat

\*

\* @goal deploy

\* @phase install

\*/

public class TomcatDeployerMojo extends AbstractMojo

{

/\*\*

\* The host name or IP address of the deployment server

\*

\* @parameter alias="host" expression="${deploy.host}" @required

\*/

private String serverHost;

/\*\*

\* The port of the deployment server

\*

\* @parameter alias="port" expression="${deploy.port}" default-value="58020"

\*/

private String serverPort;

/\*\*

\* The username to connect to the deployment manager (if omitted then the plugin

\* attempts to deploy the application to the server without credentials)

\*

\* @parameter alias="username" expression="${deploy.username}"

\*/

private String username;

/\*\*

\* The password for the specified username

\*

\* @parameter alias="password" expression="${deploy.password}"

\*/

private String password;

/\*\*

\* The name of the source artifact to deploy, such as target/pos.war

\*

\* @parameter alias="artifactSource" expression=${deploy.artifactSource}"

\* @required

\*/

private String artifactSource;

/\*\*

\* The destination name of the artifact to deploy, such as ROOT.war.

\* If not present then the

\* artifact source name is used (without pathing information)

\*

\* @parameter alias="artifactDestination"

\* expression=${deploy.artifactDestination}"

\*/

private String artifactDestination;

public void execute() throws MojoExecutionException

{

getLog().info( "Server Host: " + serverHost +

", Server Port: " + serverPort +

", Artifact Source: " + artifactSource +

", Artifact Destination: " + artifactDestination );

// Validate our fields

if( serverHost == null )

{

throw new MojoExecutionException(

"No deployment host specified, deployment is not possible" );

}

if( artifactSource == null )

{

throw new MojoExecutionException(

"No source artifact is specified, deployment is not possible" );

}

...

}

}

在这个类的头部，@goal 注释指定 MOJO 执行的目标，而 @phase 指出目标执行的阶段。除了一个映射到一个有真实值的系统属性的表达式之外，每个公布的属性有一个 @phase 注释，通过将被执行的参数指定别名。如果属性有一个 @required 注释，那么它是必须的。如果它有一个default-value，那么如果没有指定的话，将使用这个值。在 execute() 方法中，您可以调用 getLog() 来访问 Maven 记录器，根据记录级别，它将输出具体消息到标准输出设备。如果插件发生故障，抛出一个 MojoExecutionException 将导致构建失败。