Maven 那点事儿

0. 前言

Jason Van Zyl，在 Java 十大风云人物排行榜上或许会看到他。

这兄弟是干嘛的？

他就是 Maven 的创始人，人们都尊称他为“Maven 他爸”。

毋庸置疑，Jason 也是一个秃顶。James Gosling、Rod Johnson、Gavin King，你们可以告诉我为什么吗？

您曾经是否会遇到这些问题：

* 我们要开发一个 Java 项目，为了保证编译通过，我们会到处去寻找 jar 包。当编译通过了，在运行的时候，却发现ClassNotFoundException，卧槽！还差 jar 包啊？再去找找吧。
* 每个 Java 项目的目录结构都没有一个统一的标准，配置文件到处都是，单元测试代码到底应该放在哪里，没有一个权威的规范。
* 可使用 Ant 作为项目构建工具，它可以自动化地完成编译、测试、打包等任务，确实为我们省了不少事儿，但编写 Ant 的 XML 脚本绝非是一件轻松的事情。

有了 Maven，以上这一切都不再是问题了。

Jason 就是 Java 开发规范的“救世主”！他给我们带来了一种全新的项目构建方式，让我们的开发工作更加高效。

不仅如此，Jason 还是一名“野心家”，他不仅希望每个 Java 开发者都能使用他定义的规范，还要我们都从他家里去获取 jar 包（他家就是 Maven 中央仓库），我们只需告诉他，我们想要的 jar 包具体在什么位置即可（这个位置就是 Maven 坐标）。

看来 Jason 要做的是两件事情：

1. 统一开发规范与工具
2. 统一管理 jar 包

这两件事情他都做到了，而且还做了更多的事情。

工欲善其事，必先利其器。咱们也来玩玩 Maven 这货吧！先得去下载一个。

1. 安装 Maven

Maven 是 Apache 基金会的顶级项目，一般情况下，被 Apache 看中的都不会是烂货。

我们可以从 <http://maven.apache.org/> 下载 Maven 开发包，其实就是一个压缩包，下载完毕后，解压一下，配置一下环境变量就可以用了。真是超简单！

假设我们刚刚下载了一个 apache-maven-3.1.1-bin.zip 文件，现在将其解压到 D:/tool 目录下 。我们不妨将解压后的目录重命名为 Maven，这样Maven 的根目录就是 D:/tool/maven 了。

有两个环境变量可以配置：

* M2\_HOME = D:/tool/maven
* MAVEN\_OPTS = -Xms128m -Xmx512m(me：基于java的貌似都有这样一个配置对虚拟机进行优化)

以上 M2\_HOME 是必须要配置的，如果想让 Maven 跑得更快点，可以根据自己的情况来设置 MAVEN\_OPTS。

现在我们可以打开 cmd，输入：

**mvn -v**

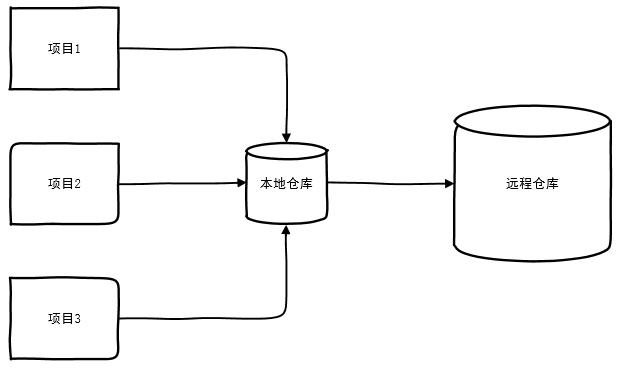
我想您一定会看到一些信息，恭喜您，Maven 安装成功！

在使用 Maven 之前，很有必要了解一下 Maven 到底是怎样管理 jar 包的，这就是 Maven 仓库要干的活了。

2. 了解 Maven 仓库

使用 Maven 给我们带来的最直接的帮助，就是 jar 包得到了统一管理，那么这些 jar 包存放在哪里呢？它们就在您的 本地仓库 中，位于 C:\Users\用户名\.m2 目录下（当然也可以修改这个默认地址）（me：可以通过配置文件settings.xml的localRepository来配置）。

实际上可将本地仓库理解为“缓存”，因为项目首先会从本地仓库中获取 jar 包，当无法获取指定 jar 包的时候，本地仓库会从 远程仓库（或 中央仓库） 中下载 jar 包，并放入本地仓库中以备将来使用。这个远程仓库是 Maven 官方提供的，可通过 <http://search.maven.org/> 来访问。这样一来，本地仓库会随着项目的积累越来越大。通过下面这张图可以清晰地表达项目、本地仓库、远程仓库之间的关系。



这个结构是否与 Git 的本地仓库与远程仓库有异曲同工之妙呢？（me：完全是两个概念，不具可比性，反而操作的流程方向正相反）

既然 Maven 安装了，那么本地仓库也就有了，下面我们就一起来创建一个 Maven 项目吧。

3. 创建 Maven 项目

我们不妨创建一个 Java Web 项目，只需在 cmd 中输入：

**mvn archetype:generate**

随后 Maven 将下载 Archetype 插件及其所有的依赖插件，这些插件其实都是 jar 包，它们存放在您的 Maven 本地仓库中。

在 终端 中，您会看到几百个 **Archetype（原型）**，可将它理解为项目模板，您得从中选择一个（me：由此可以猜测，idea新建mvn项目的时候，选择项目类型应该是这些模板的类型，然后填写完groupId和artifactId后自动生成了一个简单的模板可能就是下载的这里的模板）。

我们的目标是创建 Java Web 项目，所以您可以选择 maven-archetype-webapp（可以在 cmd 中进行模糊搜索），随后 Maven 会与您进行一些对话（me：即交互模式），Maven 想知道以下信息：

* 项目 Archetype Version（原型版本号）是什么？—— 可选择 1.0 版本
* 项目 groupId（组织名me：这不是包名，是包名的一部分，是组织域名的反写，而这正式包名的全部分） 是什么？—— 可输入 com.smart
* 项目 artifactId（构件名）是什么？—— 可输入 smart-demo
* 项目 version（版本号）是什么？—— 可输入 1.0
* 项目 package（包名）是什么？—— 可输入 com.smart.demo

以上这种方式称为 Interactive Mode（交互模式）。

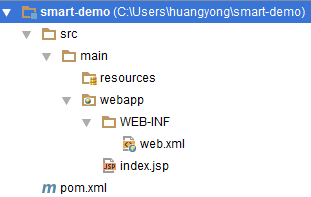
如果您是一位高效人士，或许觉得这样的交互过于繁琐，那么您也可以尝试仅使用一条命令，来完成同样的事情：

**mvn archetype:generate -DinteractiveMode=false -DarchetypeArtifactId=maven-archetype-webapp -DgroupId=com.smart -DartifactId=smart-demo -Dversion=1.0**

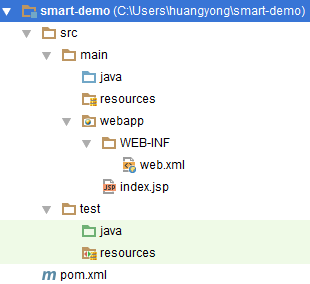
以上这种方式成为 **Batch Mode（批处理模式）**。

当然，还有第三种选择，使用 IDE 来创建 Maven 项目，您可以使用 Eclipse、NetBeans、IDEA 来创建 Maven 项目，操作过程应该是非常简单的（me:这里印证了之前的猜测，实际在使用ide构建maven项目的时候底层也是这些流程）。

您也可以使用 IDEA 直接打开一个 Maven 项目，只需要 File -> Open -> 选择 pom.xml，那么下面您就可以在 IDEA 中开发 Maven 项目了，贴一张图片吧：



其实这个目录结构还不太完备，我们需要手工添加几个目录上去，最终的目录结构看起来是这样的：



我们手工创建了三个目录：

1. src/main/java
2. src/test/java
3. src/test/resources

为什么自动生成的目录不完备？确实挺无语的，我们就不要去纠结了。不过有必要稍微解释一下这个 Maven 目录规范：

* main 目录下是项目的主要代码，test 目录下存放测试相关的代码（me：现在大部分规范的项目也是这样的目录结构）。
* 编译输出后的代码会放在target 目录下（该目录与 src 目录在同一级别下，这里没有显示出来）（me：gradle编译的默认目录是build）。
* java 目录下存放 Java 代码，resources 目录下存放配置文件。
* webapp 目录下存放 Web 应用相关代码。
* pom.xml 是 Maven 项目的配置文件。

其中 pom.xml 称为 Project Object Model（项目对象模型），它用于描述整个 Maven 项目，所以也称为 Maven 描述文件。

可见 pom.xml 才是理解 Maven 的关键点，很有必要看看它到底长什么样。

4. 理解 pom.xml

当您打开自动生成的 pom.xml，或许会感觉到可读性不太好，有必要做一下格式化，经过整理后是这样的：

<project xmlns="http://maven.apache.org/POM/4.0.0"

         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

         http://maven.apache.org/maven-v4\_0\_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.smart</groupId>

    <artifactId>smart-demo</artifactId>

    <version>1.0</version>

    <packaging>war</packaging>

    <name>smart-demo Maven Webapp</name>

    <url>http://maven.apache.org</url>

    <dependencies>

        <dependency>

            <groupId>junit</groupId>

            <artifactId>junit</artifactId>

            <version>3.8.1</version>

            <scope>test</scope>

        </dependency>

    </dependencies>

    <build>

        <finalName>smart-demo</finalName>

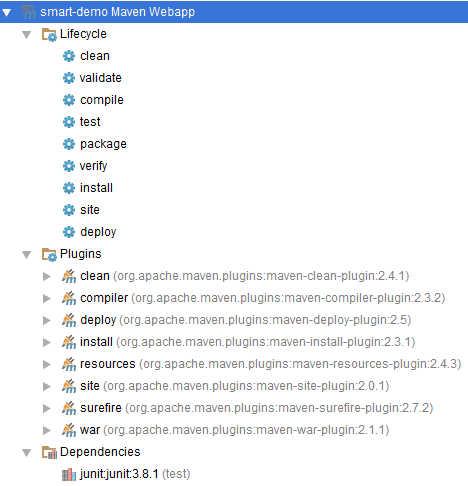
    </build>

</project>

从上往下简要说明一下：

* modelVersion：这个是 POM 的版本号，现在都是 4.0.0 ，必须得有，但不需要修改。
* groupId、artifactId、version：分别表示 Maven 项目的组织名、构件名、版本号，它们三个合起来就是 Maven 坐标，根据这个坐标可以在 Maven 仓库中对应唯一的 Maven 构件。
* packaging：表示该项目的打包方式，war 表示打包为 war 文件，默认为 jar，表示打包为 jar 文件。
* name、url：表示该项目的名称与 URL 地址，意义不大，可以省略。
* dependencies：定义该项目的依赖关系，其中每一个 dependency 对应一个 Maven 项目，可见 Maven 坐标再次出现，还多了一个 scope，表示作用域（下面会描述）。
* build：表示与构建相关的配置，这里的 finalName 表示最终构建后的名称 smart-demo.war，这里的 finalName 还可以使用另一种方式来定义（下面会描述）。

如果用树形图来表达 pom.xml，那么会更加清晰：



可见，除了项目的基本信息（Maven 坐标、打包方式等）以外，每个 pom.xml 都应该包括：

1. Lifecycle（生命周期）
2. Plugins（插件）
3. Dependencies（依赖）

Lifecycle 是项目构建的生命周期，它包括 9 个 Phase（阶段）。

大家知道，Maven 是一个核心加上多个插件的架构（me：apache是一个核心多个模块），而这些插件提供了一系列非常重要的功能，这些插件会在许多阶段里发挥重要作用。

|  |  |  |
| --- | --- | --- |
| **阶段** | **插件** | **作用** |
| clean | clean | 清理自动生成的文件，也就是 target 目录 |
| validate | 由 Maven 核心负责 | 验证 Maven 描述文件是否有效 |
| compile | compiler、resources | 编译 Java 源码 |
| test | compiler、surefire、resources | 运行测试代码 |
| package | war | 项目打包，就是生成构件包，也就是打 war 包 |
| verify | 由 Maven 核心负责 | 验证构件包是否有效 |
| install | install | 将构件包安装到本地仓库 |
| site | site | 生成项目站点，就是一堆静态网页文件，包括 JavaDoc |
| deploy | deploy | 将构件包部署到远程仓库 |

以上表格中所出现的插件名称实际上是插件的别名（或称为前缀），比如：compiler 实际上是 org.apache.maven.plugins:maven-compiler-plugin:2.3.2，这个才是 Maven 插件的完全名称。

每个插件又包括了一系列的 **Goal（目标）**，以 compiler 插件为例，它包括以下目标：

* compiler:help：用于显示 compiler 插件的使用帮助。
* compiler:compile：用于编译 main 目录下的 Java 代码。
* compiler:testCompile：用于编译 test 目录下的 Java 代码。

可见，插件目标才是具体干活的人，一个插件包括了一个多个目标，一个阶段可由零个或多个插件来提供支持。

我们可以在 pom.xml 中定义一些列的项目依赖（构件包），每个构件包都会有一个 **Scope（作用域）**，它表示该构件包在什么时候起作用，包括以下五种：

1. compile：默认作用域，在编译、测试、运行时有效
2. test：对于测试时有效
3. runtime：对于测试、运行时有效
4. provided：对于编译、测试时有效，但在运行时无效
5. system：与 provided 类似，但依赖于系统资源

可用一张矩阵表格来表示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 作用域 | 编译时有效 | 测试时有效 | 运行时有效 | 示例 |
| compile | **√** | **√** | **√** | smart-framework.jar |
| test |  | **√** |  | junit.jar |
| runtime |  | **√** | **√** | mysql-connector-java.jar |
| provided | **√** | **√** |  | servlet-api.jar |
| system | **√** | **√** |  | JDK 的 rt.jar |

如果您想开发一个 Smart 应用，可参考如下 pom.xml：

<project xmlns="http://maven.apache.org/POM/4.0.0"

         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

         http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <properties>

        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

        <smart.version>1.0</smart.version>

    </properties>

    <groupId>com.smart</groupId>

    <artifactId>smart-demo</artifactId>

    <version>1.0</version>

    <packaging>war</packaging>

    <dependencies>

        <!-- JUnit -->

        <dependency>

            <groupId>junit</groupId>

            <artifactId>junit</artifactId>

            <version>4.11</version>

            <scope>test</scope>

        </dependency>

        <!-- MySQL -->

        <dependency>

            <groupId>mysql</groupId>

            <artifactId>mysql-connector-java</artifactId>

            <version>5.1.25</version>

            <scope>runtime</scope>

        </dependency>

        <!-- Servlet -->

        <dependency>

            <groupId>javax.servlet</groupId>

            <artifactId>javax.servlet-api</artifactId>

            <version>3.0.1</version>

            <scope>provided</scope>

        </dependency>

        <!-- JSTL -->

        <dependency>

            <groupId>javax.servlet</groupId>

            <artifactId>jstl</artifactId>

            <version>1.2</version>

            <scope>runtime</scope>

        </dependency>

        <!-- Smart -->

        <dependency>

            <groupId>com.smart</groupId>

            <artifactId>smart-framework</artifactId>

            <version>${smart.version}</version>

        </dependency>

    </dependencies>

    <build>

        <plugins>

            <!-- Compile -->

            <plugin>

                <groupId>org.apache.maven.plugins</groupId>

                <artifactId>maven-compiler-plugin</artifactId>

                <version>2.5.1</version>

                <configuration>

                    <source>1.6</source>

                    <target>1.6</target>

                </configuration>

            </plugin>

            <!-- Test -->

            <plugin>

                <groupId>org.apache.maven.plugins</groupId>

                <artifactId>maven-surefire-plugin</artifactId>

                <version>2.15</version>

                <configuration>

                    <skipTests>true</skipTests>

                </configuration>

            </plugin>

            <!-- Package -->

            <plugin>

                <groupId>org.apache.maven.plugins</groupId>

                <artifactId>maven-war-plugin</artifactId>

                <version>2.4</version>

                <configuration>

                    <warName>${project.artifactId}</warName>

                </configuration>

            </plugin>

            <!-- Tomcat -->

            <plugin>

                <groupId>org.apache.tomcat.maven</groupId>

                <artifactId>tomcat7-maven-plugin</artifactId>

                <version>2.2</version>

            </plugin>

        </plugins>

    </build>

</project>

以上 pom.xml 大致解释一下：

* 我们可使用 properties 来定义一些配置属性，例如：project.build.sourceEncoding（项目构建源码编码方式），可设置为 UTF-8，可防止中文乱码。也可定义相关构件包版本号，例如：smart.version，便于日后统一升级。
* 建议使用最新版本的 JUnit，通过 Archetype 自动生成的 JUnit 太老了（3.8.1），可改为最新版（4.11）。
* 因为没必要使用 MySQL 客户端的 API，它仅仅在运行时有效，所以我们将 MySQL 构件包的作用域设置为 runtime。
* 因为我们只想在代码中使用 Servlet API，而不想将它所对应的 jar 包放入 WEB-INF 的 lib 目录下，所以我们可设置 Servlet 构件包的作用域为 provided。
* 为了保证在 JDK 1.6 运行，我们可配置 maven-compiler-plugin 插件，设置输入源码为 1.6，编译输出的字节码也为 1.6。
* 如果想跳过测试，可配置 maven-surefire-plugin 插件，将 skipTests 设置为 true。
* 如果想配置生成的 war 包为 artifactId，可修改 maven-war-plugin 插件，将 warName 修改为 ${project.artifactId}，这样就无需再配置 finalName 了。
* 如果想通过 Maven 将应用部署到 Tomcat 中，可使用 tomcat7-maven-plugin 插件，可使用 mvn tomcat7:run-war 命令来运行 war 包。

5. 使用 Maven 命令

前面我们已经使用了几个 Maven 命令，例如：mvn archetype:generate，mvn tomcat7:run-war 等。其实，可使用两种不同的方式来执行 Maven 命令：

方式一：**mvn <插件>:<目标> [参数]**

方式二：**mvn <阶段>**

现在我们接触到的都是第一种方式，而第二种方式才是我们日常中使用最频繁的，例如：

* mvn clean：清空输出目录（即 target 目录）
* mvn compile：编译源代码
* mvn package：生成构件包（一般为 jar 包或 war 包）
* mvn install：将构件包安装到本地仓库
* mvn deploy：将构件包部署到远程仓库
* mvn war:war
* mvn jar:jar

执行 Maven 命令需要注意的是：必须在 Maven 项目的根目录处执行，也就是当前目录下一定存在一个名为 pom.xml 的文件。

6. 后记

Maven 使 Java 开发更加规范化与自动化，其实 Maven 那点事远远不止这些，如果您掌握了以上这些基础知识，再去学习 Maven 的高级特性，我想一定会是一件非常轻松的事情。

推荐大家使用 [OSC Maven](http://maven.oschina.net/)，它是国内 Maven 的镜像站点，使用它可加快构件包的下载速度，从而提升您的开发效率，可阅读《[使用 OSC Maven 仓库](http://my.oschina.net/huangyong/blog/180189)》这篇文章来学会如何使用 OSC Maven。

感谢您阅读本文，感谢您对“[Java 那点事儿](http://my.oschina.net/huangyong/blog/161419)”与“[Smart Framework](http://my.oschina.net/huangyong/blog/158380)”的支持与鼓励！