

Maven 教程



Maven 翻译为"专家"、"内行", 是 Apache 下的一个纯 Java 开发的开源项目。基于项目对象模型（缩写：POM）概念，Maven利用一个中央信息片断能管理一个项目的构建、报告和文档等步骤。

Maven 是一个项目管理工具，可以对 Java 项目进行构建、依赖管理。

Maven 也可被用于构建和管理各种项目，例如 C#，Ruby，Scala 和其他语言编写的项目。Maven 曾是 Jakarta 项目的子项目，现为由 Apache 软件基金会主持的独立 Apache 项目。

阅读本教程前，您需要了解的知识

本教程主要针对初学者，帮助他们学习 Maven 工具的基本功能。完成本教程的学习后你的 Apache Maven 的专业知识将达到中等水平，随后你可以学习更高级的知识了。

阅读本教程，您需要有以下基础：[Java 基础](#)。

Maven 功能

Maven 能够帮助开发者完成以下工作：

- 构建
- 文档生成
- 报告
- 依赖
- SCMs
- 发布
- 分发
- 邮件列表

约定配置

Maven 提倡使用一个共同的标准目录结构，Maven 使用约定优于配置的原则，大家尽可能的遵守这样的目录结构。如下所示：

目录	目的
\${basedir}	存放pom.xml和所有的子目录
\${basedir}/src/main/java	项目的java源代码
\${basedir}/src/main/resources	项目的资源，比如说property文件，springmvc.xml
\${basedir}/src/test/java	项目的测试类，比如说JUnit代码
\${basedir}/src/test/resources	测试用用的资源
\${basedir}/src/main/webapp/WEB-INF	web应用文件目录，web项目的信息，比如存放web.xml、本地图片、jsp视图页面
\${basedir}/target	打包输出目录
\${basedir}/target/classes	编译输出目录
\${basedir}/target/test-classes	测试编译输出目录

目录	目的
Test.java	Maven只会自动运行符合该命名规则的测试类
~/.m2/repository	Maven默认的本地仓库目录位置

Maven 特点

项目设置遵循统一的规则。

任意工程中共享。

依赖管理包括自动更新。

一个庞大且不断增长的库。

可扩展，能够轻松编写 **Java** 或脚本语言的插件。

只需很少或不需要额外配置即可即时访问新功能。

基于模型的构建 – Maven能够将任意数量的项目构建到预定义的输出类型中，如 **JAR**、**WAR** 或基于项目元数据的分发，而不需要在大多数情况下执行任何脚本。

项目信息的一致性站点 – 使用与构建过程相同的元数据，Maven 能够生成一个网站或PDF，包括您要添加的任何文档，并添加到关于项目开发状态的标准报告中。

发布管理和发布发布 – Maven 将不需要额外的配置，就可以与源代码管理系统（如 **Subversion** 或 **Git**）集成，并可以基于某个标签管理项目的发布。它也可以将其发布到分发位置供其他项目使用。Maven能够发布单独的输出，如 **JAR**，包含其他依赖和文档的归档，或者作为源代码发布。

向后兼容性 – 您可以很轻松的从旧版本 Maven 的多个模块移植到 Maven 3 中。

子项目使用父项目依赖时，正常情况子项目应该继承父项目依赖，无需使用版本号，

并行构建 – 编译的速度能普遍提高20 - 50 %。

更好的错误报告 – Maven 改进了错误报告，它为您提供了 **Maven wiki** 页面的链接，您可以点击链接查看错误的完整描述。

Maven 环境配置 ☐



1 篇笔记
#1

☐ 写笔记



Maven 的 Snapshot 版本与 Release 版本

1、Snapshot 版本代表不稳定、尚处于开发中的版本。

2、Release 版本则代表稳定的版本。

3、什么情况下该用 SNAPSHOT?

协同开发时，如果 A 依赖构件 B，由于 B 会更新，B 应该使用 SNAPSHOT 来标识自己。这种做法的必要性可以反证如下：

a. 如果 B 不用 SNAPSHOT，而是每次更新后都使用一个稳定的版本，那版本号就会升得太快，每天一升甚至每小时一升，这就是对版本号的滥用。

b.如果 B 不用 SNAPSHOT,但一直使用一个单一的 Release 版本号，那当 B 更新后，A 可能并不会接受到更新。因为 A 所使用的 repository 一般不会频繁更新 release 版本的缓存（即本地 repository），所以B以不换版本号的方式更新后，A在拿B时发现本地已有这个版本，就不会去远程Repository下载最新的 B

4、不用 Release 版本，在所有地方都用 SNAPSHOT 版本行不行？

不行。正式环境中不得使用 snapshot 版本的库。比如说，今天你依赖某个 snapshot 版本的第三方库成功构建了自己的应用，明天再构建时可能就会失败，因为今晚第三方可能已经更新了它的 snapshot 库。你再次构建时，Maven 会去远程 repository 下载 snapshot 的最新版本，你构建时用的库就是新的 jar 文件了，这时正确性就很难保证了。

任人欺凌小师妹21小时前

反馈/建议



Maven 环境配置

Maven 是一个基于 Java 的工具，所以要做的第一件事情就是安装 JDK。
如果你还未安装 JDK，可以参考我们的 [Java 开发环境配置](#)。

系统要求

项目	要求
JDK	Maven 3.3 要求 JDK 1.7 或以上 Maven 3.2 要求 JDK 1.6 或以上 Maven 3.0/3.1 要求 JDK 1.5 或以上
内存	没有最低要求
磁盘	Maven 自身安装需要大约 10 MB 空间。除此之外，额外的磁盘空间将用于你的本地 Maven 仓库。你本地仓库的大小取决于使用情况，但预期至少 500 MB
操作系统	没有最低要求

检查 Java 安装

操作系统	任务	命令
Windows	打开命令控制台	<pre>c:\> java -version</pre>
Linux	打开命令终端	<pre># java -version</pre>
Mac	打开终端	<pre>\$ java -version</pre>

Maven 下载

Maven 下载地址: <http://maven.apache.org/download.cgi>

	Link
Binary tar.gz archive	apache-maven-3.5.4-bin.tar.gz
Binary zip archive	apache-maven-3.5.4-bin.zip
Source tar.gz archive	apache-maven-3.5.4-src.tar.gz
Source zip archive	apache-maven-3.5.4-src.zip

不同平台下载对应的包:

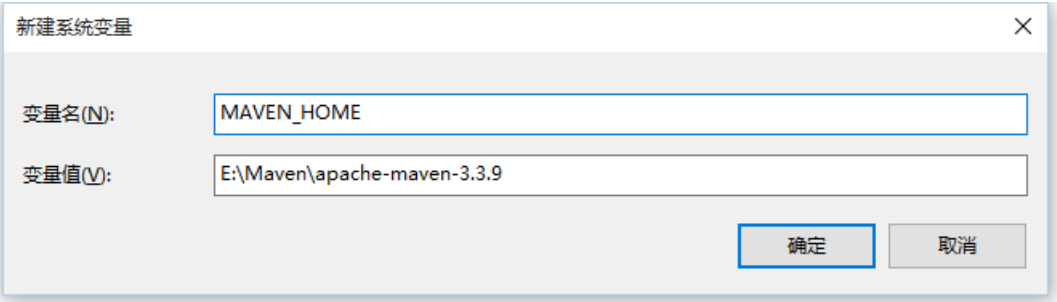
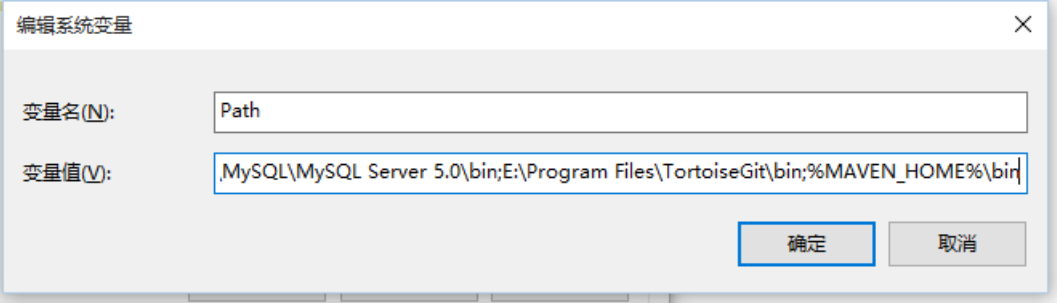
系统	包名
Windows	apache-maven-3.3.9-bin.zip
Linux	apache-maven-3.3.9-bin.tar.gz
Mac	apache-maven-3.3.9-bin.tar.gz

下载包后解压到对应目录：

系统	存储位置 (可根据自己情况配置)
Windows	E:\Maven\apache-maven-3.3.9
Linux	/usr/local/apache-maven-3.3.9
Mac	/usr/local/apache-maven-3.3.9

设置 **Maven** 环境变量

添加环境变量 MAVEN_HOME：

系统	配置
Windows	<div>右键 "计算机"，选择 "属性"，之后点击 "高级系统设置"，点击"环境变量"，来设置环境变量，有以下系统变量需要配置：</div> <div>新建系统变量 MAVEN_HOME，变量值： <code>E:\Maven\apache-maven-3.3.9</code></div> <div></div> <div>编辑系统变量 Path，添加变量值： <code>;%MAVEN_HOME%\bin</code></div> <div></div> <div>注意：注意多个值之间需要有分号隔开，然后点击确定。</div>

Linux

下载解压：

```
# wget http://mirrors.hust.edu.cn/apache/maven/maven-3/3.3.9/binaries/apache-maven-3.3.9-bin.tar.gz

# tar -xvf apache-maven-3.3.9-bin.tar.gz

# sudo mv -f apache-maven-3.3.9 /usr/local/
```

编辑 `/etc/profile` 文件 `sudo vim /etc/profile`，在文件末尾添加如下代码：

```
export MAVEN_HOME=/usr/local/apache-maven-3.3.9

export PATH=${PATH}:${MAVEN_HOME}/bin
```

保存文件，并运行如下命令使环境变量生效：

`# source /etc/profile`

在控制台输入如下命令，如果能看到 **Maven** 相关版本信息，则说明 **Maven** 已经安装成功：

```
# mvn -v
```

下载解压：

```
$ curl -O http://mirrors.hust.edu.cn/apache/maven/maven-3/3.3.9/binaries/apache-maven-3.3.9-bin.tar.gz

$ tar -xvf apache-maven-3.3.9-bin.tar.gz

$ sudo mv -f apache-maven-3.3.9 /usr/local/
```

编辑 `/etc/profile` 文件 `sudo vim /etc/profile`，在文件末尾添加如下代码：

```
export MAVEN_HOME=/usr/local/apache-maven-3.3.9

export PATH=${PATH}:${MAVEN_HOME}/bin
```

保存文件，并运行如下命令使环境变量生效：

```
$ source /etc/profile
```

在控制台输入如下命令，如果能看到 `Maven` 相关版本信息，则说明 `Maven` 已经安装成功：

```
$ mvn -v

Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-11T00:41:47+08:00)

Maven home: /usr/local/apache-maven-3.3.9

Java version: 1.8.0_31, vendor: Oracle Corporation

Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_31.jdk/Contents/Home/jre

Default locale: zh_CN, platform encoding: ISO8859-1

OS name: "mac os x", version: "10.13.4", arch: "x86_64", family: "mac"
```



Maven POM

POM(Project Object Model，项目对象模型) 是 **Maven** 工程的基本工作单元，是一个XML文件，包含了项目的基本信息，用于描述项目如何构建，声明项目依赖，等等。

执行任务或目标时，**Maven** 会在当前目录中查找 **POM**。它读取 **POM**，获取所需的配置信息，然后执行目标。

POM 中可以指定以下配置：

项目依赖
插件
执行目标
项目构建 profile
项目版本
项目开发者列表
相关邮件列表信息

在创建 **POM** 之前，我们首先需要描述项目组 (**groupId**)，项目的唯一ID。

```
<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">
<!-- 模型版本 -->
<modelVersion>4.0.0</modelVersion>
<!-- 公司或者组织的唯一标志，并且配置时生成的路径也是由此生成， 如com.companyname.project-group，maven会将该项目打成的jar包放本地路径： /com/companyname/project-group -->
<groupId>com.companyname.project-group</groupId>
<!-- 项目的唯一ID，一个groupId下面可能多个项目，就是靠artifactId来区分的 -->
<artifactId>project</artifactId>
<!-- 版本号 -->
<version>1.0</version>
</project>
```

所有 **POM** 文件都需要 **project** 元素和三个必需字段：**groupId**，**artifactId**，**version**。

节点	描述
project	工程的根标签。
modelVersion	模型版本需要设置为 4.0。
groupId	这是工程组的标识。它在一个组织或者项目中通常是唯一的。例如，一个银行组织 com.companyname.project-group 拥有所有的和银行相关的项目。
artifactId	这是工程的标识。它通常是工程的名称。例如，消费者银行。 groupId 和 artifactId 一起定义了 artifact 在仓库中的位置。
version	这是工程的版本号。在 artifact 的仓库中，它用来区分不同的版本。例如： <div><pre>com.company.bank:consumer-banking:1.0 com.company.bank:consumer-banking:1.1</pre></div>

父（Super）POM

父（**Super**）**POM**是 **Maven** 默认的 **POM**。所有的 **POM** 都继承自一个父 **POM**（无论是否显式定义了这个父 **POM**）。父 **POM** 包含了一些可以被继承的默认设置。因此，当 **Maven** 发现需要下载 **POM** 中的 依赖时，它会到 **Super POM** 中配置的默认仓库 **http://repo1.maven.org/maven2** 去下载。**Maven** 使用 **effective pom**（**Super pom** 加上工程自己的配置）来执行相关的目标，它帮助开发者在 **pom.xml** 中做尽可能少的配置，当然这些配置可以被重写。

使用以下命令来查看 **Super POM** 默认配置：

```
mvn help:effective-pom
```

接下来我们创建目录 `MVN/project`，在该目录下创建 `pom.xml`，内如如下：

```
<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">
<!-- 模型版本 -->
<modelVersion>4.0.0</modelVersion>
<!-- 公司或者组织的唯一标志，并且配置时生成的路径也是由此生成， 如com.companyname.project-group，maven会将该项目打成的jar
包放本地路径： /com/companyname/project-group -->
<groupId>com.companyname.project-group</groupId>
<!-- 项目的唯一ID，一个groupId下面可能多个项目，就是靠artifactId来区分的 -->
<artifactId>project</artifactId>
<!-- 版本号 -->
<version>1.0</version>
</project>
```

在命令控制台，进入 `MVN/project` 目录，执行以下命令：

Maven 将会开始处理并显示 `effective-pom`。

```
[INFO] Scanning for projects...

Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/2.5/maven-clean-plugin-
2.5.pom

...

[INFO] -----

[INFO] BUILD SUCCESS

[INFO] -----

[INFO] Total time: 01:36 min

[INFO] Finished at: 2018-09-05T11:31:28+08:00

[INFO] Final Memory: 15M/149M

[INFO] -----
```

Effective POM 的结果就像在控制台中显示的一样，经过继承、插值之后，使配置生效。

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ===== -->
<!-- -->
<!-- Generated by Maven Help Plugin on 2012-07-05T11:41:51 -->
<!-- See: http://maven.apache.org/plugins/maven-help-plugin/ -->
<!-- -->
<!-- ===== -->
<!-- ===== -->
<!-- -->
<!-- Effective POM for project -->
<!-- 'com.companyname.project-group:project-name:jar:1.0' -->
<!-- -->
<!-- ===== -->
<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 h
ttp://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.companyname.project-group</groupId>
<artifactId>project</artifactId>
<version>1.0</version>
<build>
```



```
<sourceDirectory>C:\MVN\project\src\main\java</sourceDirectory>
<scriptSourceDirectory>src/main/scripts</scriptSourceDirectory>
<testSourceDirectory>C:\MVN\project\src\test\java</testSourceDirectory>
<outputDirectory>C:\MVN\project\target\classes</outputDirectory>
<testOutputDirectory>C:\MVN\project\target\test-classes</testOutputDirectory>
<resources>
<resource>
<mergeId>resource-0</mergeId>
<directory>C:\MVN\project\src\main\resources</directory>
</resource>
</resources>
<testResources>
<testResource>
<mergeId>resource-1</mergeId>
<directory>C:\MVN\project\src\test\resources</directory>
</testResource>
</testResources>
<directory>C:\MVN\project\target</directory>
<finalName>project-1.0</finalName>
<pluginManagement>
<plugins>
<plugin>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.3</version>
</plugin>
<plugin>
<artifactId>maven-assembly-plugin</artifactId>
<version>2.2-beta-2</version>
</plugin>
<plugin>
<artifactId>maven-clean-plugin</artifactId>
<version>2.2</version>
</plugin>
<plugin>
<artifactId>maven-compiler-plugin</artifactId>
<version>2.0.2</version>
</plugin>
<plugin>
<artifactId>maven-dependency-plugin</artifactId>
<version>2.0</version>
</plugin>
<plugin>
<artifactId>maven-deploy-plugin</artifactId>
<version>2.4</version>
</plugin>
<plugin>
<artifactId>maven-ear-plugin</artifactId>
<version>2.3.1</version>
</plugin>
<plugin>
<artifactId>maven-ejb-plugin</artifactId>
<version>2.1</version>
</plugin>
<plugin>
<artifactId>maven-install-plugin</artifactId>
<version>2.2</version>
</plugin>
<plugin>
<artifactId>maven-jar-plugin</artifactId>
<version>2.2</version>
</plugin>
<plugin>
<artifactId>maven-javadoc-plugin</artifactId>
<version>2.5</version>
</plugin>
<plugin>
<artifactId>maven-plugin-plugin</artifactId>
<version>2.4.3</version>
</plugin>
<plugin>
<artifactId>maven-rar-plugin</artifactId>
<version>2.2</version>
```

```

</plugin>
<plugin>
<artifactId>maven-release-plugin</artifactId>
<version>2.0-beta-8</version>
</plugin>
<plugin>
<artifactId>maven-resources-plugin</artifactId>
<version>2.3</version>
</plugin>
<plugin>
<artifactId>maven-site-plugin</artifactId>
<version>2.0-beta-7</version>
</plugin>
<plugin>
<artifactId>maven-source-plugin</artifactId>
<version>2.0.4</version>
</plugin>
<plugin>
<artifactId>maven-surefire-plugin</artifactId>
<version>2.4.3</version>
</plugin>
<plugin>
<artifactId>maven-war-plugin</artifactId>
<version>2.1-alpha-2</version>
</plugin>
</plugins>
</pluginManagement>
<plugins>
<plugin>
<artifactId>maven-help-plugin</artifactId>
<version>2.1.1</version>
</plugin>
</plugins>
</build>
<repositories>
<repository>
<snapshots>
<enabled>>false</enabled>
</snapshots>
<id>central</id>
<name>Maven Repository Switchboard</name>
<url>http://repo1.maven.org/maven2</url>
</repository>
</repositories>
<pluginRepositories>
<pluginRepository>
<releases>
<updatePolicy>never</updatePolicy>
</releases>
<snapshots>
<enabled>>false</enabled>
</snapshots>
<id>central</id>
<name>Maven Plugin Repository</name>
<url>http://repo1.maven.org/maven2</url>
</pluginRepository>
</pluginRepositories>
<reporting>
<outputDirectory>C:\MVN\project\target\site</outputDirectory>
</reporting>
</project>

```

在上面的 `pom.xml` 中，你可以看到 **Maven** 在执行目标时需要用到的默认工程源码目录结构、输出目录、需要的插件、仓库和报表目录。

Maven 的 `pom.xml` 文件也不需要手工编写。

Maven 提供了大量的原型插件来创建工程，包括工程结构和 `pom.xml`。

POM 标签大全详解

```

<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/4.0.0http://maven.apache.org/maven-v4_0_0.xsd">
<!-- 父项目的坐标。如果项目中没有规定某个元素的值，那么父项目中的对应值即为项目的默认值。 坐标包括group ID, artifact ID和

```

```
version。 -->
<parent>
<!--被继承的父项目的构件标识符 -->
<artifactId />
<!--被继承的父项目的全球唯一标识符 -->
<groupId />
<!--被继承的父项目的版本 -->
<version />
<!-- 父项目的pom.xml文件的相对路径。相对路径允许你选择一个不同的路径。默认值是../pom.xml。Maven首先在构建当前项目的地方寻找父项目
目的pom，其次在文件系统的这个位置（relativePath位置），然后在本地仓库，最后在远程仓库寻找父项目的pom。 -->
<relativePath />
</parent>
<!--声明项目描述符遵循哪一个POM模型版本。模型本身的版本很少改变，虽然如此，但它仍然是必不可少的，这是为了当Maven引入了新的特性或者其他模型变更的时候，确保稳定性。 -->
<modelVersion>4.0.0</modelVersion>
<!--项目的全球唯一标识符，通常使用全限定的包名区分该项目和其他项目。并且构建时生成的路径也是由此生成， 如com.mycompany.app生成的相对路径为: /com/mycompany/app -->
<groupId>asia.banseaon</groupId>
<!-- 构件的标识符，它和group ID一起唯一标识一个构件。换句话说，你不能有两个不同的项目拥有同样的artifact ID和groupId；在某个特定的group ID下，artifact ID也必须是唯一的。构件是项目产生的或使用的一个东西，Maven为项目产生的构件包括：JARs，源码，二进制发布和WARs等。 -->
<artifactId>banseaon-maven2</artifactId>
<!--项目产生的构件类型，例如jar、war、ear、pom。插件可以创建他们自己的构件类型，所以前面列的不是全部构件类型 -->
<packaging>jar</packaging>
<!--项目当前版本，格式为:主版本.次版本.增量版本-限定版本号 -->
<version>1.0-SNAPSHOT</version>
<!--项目的名称，Maven产生的文档用 -->
<name>banseaon-maven</name>
<!--项目主页的URL，Maven产生的文档用 -->
<url>http://www.baidu.com/banseaon</url>
<!-- 项目的详细描述，Maven 产生的文档用。 当这个元素能够用HTML格式描述时（例如，CDATA中的文本会被解析器忽略，就可以包含HTML标签）， 不鼓励使用纯文本描述。如果你需要修改产生的web站点的索引页面，你应该修改你自己的索引页文件，而不是调整这里的文档。 -->
<description>A maven project to study maven.</description>
<!--描述了这个项目构建环境中的前提条件。 -->
<prerequisites>
<!--构建该项目或使用该插件所需要的Maven的最低版本 -->
<maven />
</prerequisites>
<!--项目的问题管理系统(Bugzilla, Jira, Scarab,或任何你喜欢的问题管理系统)的名称和URL，本例为 jira -->
<issueManagement>
<!--问题管理系统（例如jira）的名字， -->
<system>jira</system>
<!--该项目使用的问题管理系统的URL -->
<url>http://jira.baidu.com/banseaon</url>
</issueManagement>
<!--项目持续集成信息 -->
<ciManagement>
<!--持续集成系统的名字，例如continuum -->
<system />
<!--该项目使用的持续集成系统的URL（如果持续集成系统有web接口的话）。 -->
<url />
<!--构建完成时，需要通知的开发者/用户的配置项。包括被通知者信息和通知条件（错误，失败，成功，警告） -->
<notifiers>
<!--配置一种方式，当构建中断时，以该方式通知用户/开发者 -->
<notifier>
<!--传送通知的途径 -->
<type />
<!--发生错误时是否通知 -->
<sendOnError />
<!--构建失败时是否通知 -->
<sendOnFailure />
<!--构建成功时是否通知 -->
<sendOnSuccess />
<!--发生警告时是否通知 -->
<sendOnWarning />
<!--不赞成使用。通知发送到哪里 -->
<address />
<!--扩展配置项 -->
```

```
<configuration />
</notifier>
</notifiers>
</ciManagement>
<!--项目创建年份，4位数字。当产生版权信息时需要使用这个值。 -->
<inceptionYear />
<!--项目相关邮件列表信息 -->
<mailingLists>
<!--该元素描述了项目相关的所有邮件列表。自动产生的网站引用这些信息。 -->
<mailingList>
<!--邮件的名称 -->
<name>Demo</name>
<!--发送邮件的地址或链接，如果是邮件地址，创建文档时，mailto: 链接会被自动创建 -->
<post>banseon@126.com</post>
<!--订阅邮件的地址或链接，如果是邮件地址，创建文档时，mailto: 链接会被自动创建 -->
<subscribe>banseon@126.com</subscribe>
<!--取消订阅邮件的地址或链接，如果是邮件地址，创建文档时，mailto: 链接会被自动创建 -->
<unsubscribe>banseon@126.com</unsubscribe>
<!--你可以浏览邮件信息的URL -->
<archive>http://hi.baidu.com/banseon/demo/dev/</archive>
</mailingList>
</mailingLists>
<!--项目开发者列表 -->
<developers>
<!--某个项目开发者的信息 -->
<developer>
<!--SCM里项目开发者的唯一标识符 -->
<id>HELLO WORLD</id>
<!--项目开发者的全名 -->
<name>banseon</name>
<!--项目开发者的email -->
<email>banseon@126.com</email>
<!--项目开发者的主页的URL -->
<url />
<!--项目开发者在项目中所扮演的角色，角色元素描述了各种角色 -->
<roles>
<role>Project Manager</role>
<role>Architect</role>
</roles>
<!--项目开发者所属组织 -->
<organization>demo</organization>
<!--项目开发者所属组织的URL -->
<organizationUrl>http://hi.baidu.com/banseon</organizationUrl>
<!--项目开发者属性，如即时消息如何处理等 -->
<properties>
<dept>No</dept>
</properties>
<!--项目开发者所在时区， -11到12范围内的整数。 -->
<timezone>-5</timezone>
</developer>
</developers>
<!--项目的其他贡献者列表 -->
<contributors>
<!--项目的其他贡献者。参见developers/developer元素 -->
<contributor>
<name />
<email />
<url />
<organization />
<organizationUrl />
<roles />
<timezone />
<properties />
</contributor>
</contributors>
<!--该元素描述了项目所有License列表。 应该只列出该项目的license列表，不要列出依赖项目的 license列表。如果列出多个license
，用户可以选择它们中的一个而不是接受所有license。 -->
<licenses>
<!--描述了项目的license，用于生成项目的web站点的license页面，其他一些报表和validation也会用到该元素。 -->
<license>
<!--license用于法律上的名称 -->
<name>Apache 2</name>
```

```
<!--官方的license正文页面的URL -->
<url>http://www.baidu.com/banseon/LICENSE-2.0.txt</url>
<!--项目分发的主要方式: repo, 可以从Maven库下载 manual, 用户必须手动下载和安装依赖 -->
<distribution>repo</distribution>
<!--关于license的补充信息 -->
<comments>A business-friendly OSS license</comments>
</license>
</licenses>
<!--SCM(Source Control Management)标签允许你配置你的代码库, 供Maven web站点和其它插件使用。 -->
<scm>
<!--SCM的URL, 该URL描述了版本库和如何连接到版本库。欲知详情, 请看SCMs提供的URL格式和列表。该连接只读。 -->
<connection>
scm:svn:http://svn.baidu.com/banseon/maven/banseon/banseon-maven2-trunk(dao-trunk)
</connection>
<!--给开发者使用的, 类似connection元素。即该连接不仅仅只读 -->
<developerConnection>
scm:svn:http://svn.baidu.com/banseon/maven/banseon/dao-trunk
</developerConnection>
<!--当前代码的标签, 在开发阶段默认为HEAD -->
<tag />
<!--指向项目的可浏览SCM库 (例如ViewVC或者Fisheye) 的URL。 -->
<url>http://svn.baidu.com/banseon</url>
</scm>
<!--描述项目所属组织的各种属性。Maven产生的文档用 -->
<organization>
<!--组织的全名 -->
<name>demo</name>
<!--组织主页的URL -->
<url>http://www.baidu.com/banseon</url>
</organization>
<!--构建项目需要的信息 -->
<build>
<!--该元素设置了项目源码目录, 当构建项目的时候, 构建系统会编译目录里的源码。该路径是相对于pom.xml的相对路径。 -->
<sourceDirectory />
<!--该元素设置了项目脚本源码目录, 该目录和源码目录不同: 绝大多数情况下, 该目录下的内容 会被拷贝到输出目录(因为脚本是被解释的, 而不是被编译的)。 -->
<scriptSourceDirectory />
<!--该元素设置了项目单元测试使用的源码目录, 当测试项目的时候, 构建系统会编译目录里的源码。该路径是相对于pom.xml的相对路径。 -->
<testSourceDirectory />
<!--被编译过的应用程序class文件存放的目录。 -->
<outputDirectory />
<!--被编译过的测试class文件存放的目录。 -->
<testOutputDirectory />
<!--使用来自该项目的一系列构建扩展 -->
<extensions>
<!--描述使用到的构建扩展。 -->
<extension>
<!--构建扩展的groupId -->
<groupId />
<!--构建扩展的artifactId -->
<artifactId />
<!--构建扩展的版本 -->
<version />
</extension>
</extensions>
<!--当项目没有规定目标 (Maven2 叫做阶段) 时的默认值 -->
<defaultGoal />
<!--这个元素描述了项目相关的所有资源路径列表, 例如和项目相关的属性文件, 这些资源被包含在最终的打包文件里。 -->
<resources>
<!--这个元素描述了项目相关或测试相关的所有资源路径 -->
<resource>
<!-- 描述了资源的目标路径。该路径相对target/classes目录 (例如${project.build.outputDirectory})。举个例子, 如果你想资源在特定的包里(org.apache.maven.messages), 你就必须该元素设置为org/apache/maven /messages。然而, 如果你只是想把资源放到源码目录结构里, 就不需要该配置。 -->
<targetPath />
<!--是否使用参数值代替参数名。参数值取自properties元素或者文件里配置的属性, 文件在filters元素里列出。 -->
<filtering />
<!--描述存放资源的目录, 该路径相对POM路径 -->
<directory />
<!--包含的模式列表, 例如**/*.xml。 -->
<includes />
```

```
<!--排除的模式列表，例如**/*.xml -->
<excludes />
</resource>
</resources>
<!--这个元素描述了单元测试相关的所有资源路径，例如和单元测试相关的属性文件。 -->
<testResources>
<!--这个元素描述了测试相关的所有资源路径，参见build/resources/resource元素的说明 -->
<testResource>
<targetPath />
<filtering />
<directory />
<includes />
<excludes />
</testResource>
</testResources>
<!--构建产生的所有文件存放的目录 -->
<directory />
<!--产生的构件的文件名，默认值是${artifactId}-${version}。 -->
<finalName />
<!--当filtering开关打开时，使用到的过滤器属性文件列表 -->
<filters />
<!--子项目可以引用的默认插件信息。该插件配置项直到被引用时才会被解析或绑定到生命周期。给定插件的任何本地配置都会覆盖这里的配置 -->
<pluginManagement>
<!--使用的插件列表 。 -->
<plugins>
<!--plugin元素包含描述插件所需要的信息。 -->
<plugin>
<!--插件在仓库里的group ID -->
<groupId />
<!--插件在仓库里的artifact ID -->
<artifactId />
<!--被使用的插件的版本（或版本范围） -->
<version />
<!--是否从该插件下载Maven扩展（例如打包和类型处理器），由于性能原因，只有在真需要下载时，该元素才被设置成enabled。 -->
<extensions />
<!--在构建生命周期中执行一组目标的配置。每个目标可能有不同的配置。 -->
<executions>
<!--execution元素包含了插件执行需要的信息 -->
<execution>
<!--执行目标的标识符，用于标识构建过程中的目标，或者匹配继承过程中需要合并的执行目标 -->
<id />
<!--绑定了目标的构建生命周期阶段，如果省略，目标会被绑定到源数据里配置的默认阶段 -->
<phase />
<!--配置的执行目标 -->
<goals />
<!--配置是否被传播到子POM -->
<inherited />
<!--作为DOM对象的配置 -->
<configuration />
</execution>
</executions>
<!--项目引入插件所需要的额外依赖 -->
<dependencies>
<!--参见dependencies/dependency元素 -->
<dependency>
.....
</dependency>
</dependencies>
<!--任何配置是否被传播到子项目 -->
<inherited />
<!--作为DOM对象的配置 -->
<configuration />
</plugin>
</plugins>
</pluginManagement>
<!--使用的插件列表 -->
<plugins>
<!--参见build/pluginManagement/plugins/plugin元素 -->
<plugin>
<groupId />
<artifactId />
```

```

<version />
<extensions />
<executions>
<execution>
<id />
<phase />
<goals />
<inherited />
<configuration />
</execution>
</executions>
<dependencies>
<!--参见dependencies/dependency元素 -->
<dependency>
.....
</dependency>
</dependencies>
<goals />
<inherited />
<configuration />
</plugin>
</plugins>
</build>
<!--在列的项目构建profile，如果被激活，会修改构建处理 -->
<profiles>
<!--根据环境参数或命令行参数激活某个构建处理 -->
<profile>
<!--构建配置的唯一标识符。即用于命令行激活，也用于在继承时合并具有相同标识符的profile。 -->
<id />
<!--自动触发profile的条件逻辑。Activation是profile的开启钥匙。profile的力量来自于它 能够在某些特定的环境中自动使用某些特定的值；这些环境通过activation元素指定。activation元素并不是激活profile的唯一方式。 -->
<activation>
<!--profile默认是否激活的标志 -->
<activeByDefault />
<!--当匹配的jdk被检测到，profile被激活。例如，1.4激活JDK1.4，1.4.0_2，而!1.4激活所有版本不是以1.4开头的JDK。 -->
<jdk />
<!--当匹配的操作系统属性被检测到，profile被激活。os元素可以定义一些操作系统相关的属性。 -->
<os>
<!--激活profile的操作系统的名字 -->
<name>Windows XP</name>
<!--激活profile的操作系统所属家族(如 'windows') -->
<family>Windows</family>
<!--激活profile的操作系统体系结构 -->
<arch>x86</arch>
<!--激活profile的操作系统版本 -->
<version>5.1.2600</version>
</os>
<!--如果Maven检测到某一个属性（其值可以在POM中通过${名称}引用），其拥有对应的名称和值，Profile就会被激活。如果值 字段是空的，那么存在属性名称字段就会激活profile，否则按区分大小写方式匹配属性值字段 -->
<property>
<!--激活profile的属性的名称 -->
<name>mavenVersion</name>
<!--激活profile的属性的值 -->
<value>2.0.3</value>
</property>
<!--提供一个文件名，通过检测该文件的存在或不存在来激活profile。missing检查文件是否存在，如果不存在则激活 profile。另一方面，exists则会检查文件是否存在，如果存在则激活profile。 -->
<file>
<!--如果指定的文件存在，则激活profile。 -->
<exists>/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/
</exists>
<!--如果指定的文件不存在，则激活profile。 -->
<missing>/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/
</missing>
</file>
</activation>
<!--构建项目所需要的信息。参见build元素 -->
<build>
<defaultGoal />
<resources>
<resource>
<targetPath />

```

```
<filtering />
<directory />
<includes />
<excludes />
</resource>
</resources>
<testResources>
<testResource>
<targetPath />
<filtering />
<directory />
<includes />
<excludes />
</testResource>
</testResources>
<directory />
<finalName />
<filters />
<pluginManagement>
<plugins>
<!--参见build/pluginManagement/plugins/plugin元素 -->
<plugin>
<groupId />
<artifactId />
<version />
<extensions />
<executions>
<execution>
<id />
<phase />
<goals />
<inherited />
<configuration />
</execution>
</executions>
<dependencies>
<!--参见dependencies/dependency元素 -->
<dependency>
.....
</dependency>
</dependencies>
<goals />
<inherited />
<configuration />
</plugin>
</plugins>
</pluginManagement>
<plugins>
<!--参见build/pluginManagement/plugins/plugin元素 -->
<plugin>
<groupId />
<artifactId />
<version />
<extensions />
<executions>
<execution>
<id />
<phase />
<goals />
<inherited />
<configuration />
</execution>
</executions>
<dependencies>
<!--参见dependencies/dependency元素 -->
<dependency>
.....
</dependency>
</dependencies>
<goals />
<inherited />
<configuration />
```



```

</plugin>
</plugins>
</build>
<!--模块（有时称作子项目） 被构建成项目的一部分。列出的每个模块元素是指向该模块的目录的相对路径 -->
<modules />
<!--发现依赖和扩展的远程仓库列表。 -->
<repositories>
<!--参见repositories/repository元素 -->
<repository>
<releases>
<enabled />
<updatePolicy />
<checksumPolicy />
</releases>
<snapshots>
<enabled />
<updatePolicy />
<checksumPolicy />
</snapshots>
<id />
<name />
<url />
<layout />
</repository>
</repositories>
<!--发现插件的远程仓库列表，这些插件用于构建和报表 -->
<pluginRepositories>
<!--包含需要连接到远程插件仓库的信息。参见repositories/repository元素 -->
<pluginRepository>
<releases>
<enabled />
<updatePolicy />
<checksumPolicy />
</releases>
<snapshots>
<enabled />
<updatePolicy />
<checksumPolicy />
</snapshots>
<id />
<name />
<url />
<layout />
</pluginRepository>
</pluginRepositories>
<!--该元素描述了项目相关的所有依赖。 这些依赖组成了项目构建过程中的一个个环节。它们自动从项目定义的仓库中下载。要获取更多信息，请看项目依赖机制。 -->
<dependencies>
<!--参见dependencies/dependency元素 -->
<dependency>
.....
</dependency>
</dependencies>
<!--不赞成使用。现在Maven忽略该元素。 -->
<reports />
<!--该元素包括使用报表插件产生报表的规范。当用户执行"mvn site"，这些报表就会运行。 在页面导航栏能看到所有报表的链接。参见reporting元素 -->
<reporting>
.....
</reporting>
<!--参见dependencyManagement元素 -->
<dependencyManagement>
<dependencies>
<!--参见dependencies/dependency元素 -->
<dependency>
.....
</dependency>
</dependencies>
</dependencyManagement>
<!--参见distributionManagement元素 -->
<distributionManagement>
.....

```

```
</distributionManagement>
<!--参见properties元素 -->
<properties />
</profile>
</profiles>
<!--模块（有时称作子项目） 被构建成项目的一部分。列出的每个模块元素是指向该模块的目录的相对路径 -->
<modules />
<!--发现依赖和扩展的远程仓库列表。 -->
<repositories>
<!--包含需要连接到远程仓库的信息 -->
<repository>
<!--如何处理远程仓库里发布版本的下载 -->
<releases>
<!--true或者false表示该仓库是否为下载某种类型构件（发布版，快照版）开启。 -->
<enabled />
<!--该元素指定更新发生的频率。Maven会比较本地POM和远程POM的时间戳。这里的选项是：always（一直），daily（默认，每日），interval: X（这里X是以分钟为单位的时间间隔），或者never（从不）。 -->
<updatePolicy />
<!--当Maven验证构件校验文件失败时该怎么做：ignore（忽略），fail（失败），或者warn（警告）。 -->
<checksumPolicy />
</releases>
<!-- 如何处理远程仓库里快照版本的下载。有了releases和snapshots这两组配置，POM可以在每个单独的仓库中，为每种类型的构件采取不同的策略。例如，可能有人会决定只为开发目的开启对快照版本下载的支持。参见repositories/repository/releases元素 -->
<snapshots>
<enabled />
<updatePolicy />
<checksumPolicy />
</snapshots>
<!--远程仓库唯一标识符。可以用来匹配在settings.xml文件里配置的远程仓库 -->
<id>banseon-repository-proxy</id>
<!--远程仓库名称 -->
<name>banseon-repository-proxy</name>
<!--远程仓库URL，按protocol://hostname/path形式 -->
<url>http://192.168.1.169:9999/repository/</url>
<!-- 用于定位和排序构件的仓库布局类型-可以是default（默认）或者legacy（遗留）。Maven 2为其仓库提供了一个默认的布局；然而，Maven 1.x有一种不同的布局。我们可以使用该元素指定布局是default（默认）还是legacy（遗留）。 -->
<layout>default</layout>
</repository>
</repositories>
<!--发现插件的远程仓库列表，这些插件用于构建和报表 -->
<pluginRepositories>
<!--包含需要连接到远程插件仓库的信息。参见repositories/repository元素 -->
<pluginRepository>
.....
</pluginRepository>
</pluginRepositories>
<!--该元素描述了项目相关的所有依赖。 这些依赖组成了项目构建过程中的一个个环节。它们自动从项目定义的仓库中下载。要获取更多信息，请看项目依赖机制。 -->
<dependencies>
<dependency>
<!--依赖的group ID -->
<groupId>org.apache.maven</groupId>
<!--依赖的artifact ID -->
<artifactId>maven-artifact</artifactId>
<!--依赖的版本号。 在Maven 2里，也可以配置成版本号的范围。 -->
<version>3.8.1</version>
<!-- 依赖类型，默认类型是jar。它通常表示依赖的文件的扩展名，但也有例外。一个类型可以被映射成另外一个扩展名或分类器。类型经常和使用的打包方式对应，尽管这也有例外。一些类型的例子：jar, war, ejb-client和test-jar。如果设置extensions为 true，就可以在 plugin里定义新的类型。所以前面的类型的例子不完整。 -->
<type>jar</type>
<!-- 依赖的分类器。分类器可以区分属于同一个POM，但不同构建方式的构件。分类器名被附加到文件名的版本号后面。例如，如果你想要构建两个单独的构件成JAR，一个使用Java 1.4编译器，另一个使用Java 6编译器，你就可以使用分类器来生成两个单独的JAR构件。 -->
<classifier></classifier>
<!--依赖范围。在项目发布过程中，帮助决定哪些构件被包括进来。欲知详情请参考依赖机制。 - compile：默认范围，用于编译 - provided：类似于编译，但支持你期待jdk或者容器提供，类似于classpath - runtime：在执行时需要使用 - test：用于test任务时使用 - system：需要外在提供相应的元素。通过systemPath来取得 - systemPath：仅用于范围为system。提供相应的路径 - optional：当项目自身被依赖时，标注依赖是否传递。用于连续依赖时使用 -->
<scope>test</scope>
```

```
<!-- 仅供system范围使用。注意，不鼓励使用这个元素，并且在新的版本中该元素可能被覆盖掉。该元素为依赖规定了文件系统上的路径。需要绝对路径而不是相对路径。推荐使用属性匹配绝对路径，例如${java.home}。 -->
<systemPath></systemPath>
<!-- 当计算传递依赖时，从依赖构件列表里，列出被排除的依赖构件集。即告诉maven你只依赖指定的项目，不依赖项目的依赖。此元素主要用于解决版本冲突问题 -->
<exclusions>
<exclusion>
<artifactId>spring-core</artifactId>
<groupId>org.springframework</groupId>
</exclusion>
</exclusions>
<!-- 可选依赖，如果你在项目B中把C依赖声明为可选，你就需要在依赖于B的项目（例如项目A）中显式的引用对C的依赖。可选依赖阻断依赖的传递性。 -->
<optional>true</optional>
</dependency>
</dependencies>
<!-- 不赞成使用。现在Maven忽略该元素。 -->
<reports></reports>
<!-- 该元素描述使用报表插件产生报表的规范。当用户执行"mvn site"，这些报表就会运行。在页面导航栏能看到所有报表的链接。 -->
<reporting>
<!-- true，则，网站不包括默认的报表。这包括"项目信息"菜单中的报表。 -->
<excludeDefaults />
<!-- 所有产生的报表存放到哪里。默认值是${project.build.directory}/site。 -->
<outputDirectory />
<!-- 使用的报表插件和他们的配置。 -->
<plugins>
<!-- plugin元素包含描述报表插件需要的信息 -->
<plugin>
<!-- 报表插件在仓库里的group ID -->
<groupId />
<!-- 报表插件在仓库里的artifact ID -->
<artifactId />
<!-- 被使用的报表插件的版本（或版本范围） -->
<version />
<!-- 任何配置是否被传播到子项目 -->
<inherited />
<!-- 报表插件的配置 -->
<configuration />
<!-- 一组报表的多重规范，每个规范可能有不同的配置。一个规范（报表集）对应一个执行目标。例如，有1，2，3，4，5，6，7，8，9个报表。1，2，5构成A报表集，对应一个执行目标。2，5，8构成B报表集，对应另一个执行目标 -->
<reportSets>
<!-- 表示报表的一个集合，以及产生该集合的配置 -->
<reportSet>
<!-- 报表集合的唯一标识符，POM继承时用到 -->
<id />
<!-- 产生报表集合时，被使用的报表的配置 -->
<configuration />
<!-- 配置是否被继承到子POMs -->
<inherited />
<!-- 这个集合里使用到哪些报表 -->
<reports />
</reportSet>
</reportSets>
</plugin>
</plugins>
</reporting>
<!-- 继承自该项目的所有子项目的默认依赖信息。这部分的依赖信息不会被立即解析，而是当子项目声明一个依赖（必须描述group ID和 artifact ID信息），如果group ID和artifact ID以外的一些信息没有描述，则通过group ID和artifact ID匹配到这里的依赖，并使用这里的依赖信息。 -->
<dependencyManagement>
<dependencies>
<!-- 参见dependencies/dependency元素 -->
<dependency>
.....
</dependency>
</dependencies>
</dependencyManagement>
<!-- 项目分发信息，在执行mvn deploy后表示要发布的位置。有了这些信息就可以把网站部署到远程服务器或者把构件部署到远程仓库。 -->
<distributionManagement>
<!-- 部署项目产生的构件到远程仓库需要的信息 -->
```

```

<repository>
<!--是分配给快照一个唯一的版本号（由时间戳和构建流水号）？还是每次都使用相同的版本号？参见repositories/repository元素 -->
<uniqueVersion />
<id>banseon-maven2</id>
<name>banseon maven2</name>
<url>file://${basedir}/target/deploy</url>
<layout />
</repository>
<!--构件的快照部署到哪里？如果没有配置该元素，默认部署到repository元素配置的仓库，参见distributionManagement/repository元素 -->
<snapshotRepository>
<uniqueVersion />
<id>banseon-maven2</id>
<name>Banseon-maven2 Snapshot Repository</name>
<url>scp://svn.baidu.com/banseon:/usr/local/maven-snapshot</url>
<layout />
</snapshotRepository>
<!--部署项目的网站需要的信息 -->
<site>
<!--部署位置的唯一标识符，用来匹配站点和settings.xml文件里的配置 -->
<id>banseon-site</id>
<!--部署位置的名称 -->
<name>business api website</name>
<!--部署位置的URL，按protocol://hostname/path形式 -->
<url>
scp://svn.baidu.com/banseon:/var/www/localhost/banseon-web
</url>
</site>
<!--项目下载页面的URL。如果没有该元素，用户应该参考主页。使用该元素的原因是：帮助定位那些不在仓库里的构件（由于license限制）。 -->
<downloadUrl />
<!--如果构件有了新的group ID和artifact ID（构件移到了新的位置），这里列出构件的重定位信息。 -->
<relocation>
<!--构件新的group ID -->
<groupId />
<!--构件新的artifact ID -->
<artifactId />
<!--构件新的版本号 -->
<version />
<!--显示给用户的，关于移动的额外信息，例如原因。 -->
<message />
</relocation>
<!-- 给出该构件在远程仓库的状态。不得在本地项目中设置该元素，因为这是工具自动更新的。有效的值有：none（默认），converted（仓库管理员从Maven 1 POM转换过来），partner（直接从伙伴Maven 2仓库同步过来），deployed（从Maven 2实例部署），verified（被核实时正确的和最终的）。 -->
<status />
</distributionManagement>
<!--以值替代名称，Properties可以在整个POM中使用，也可以作为触发条件（见settings.xml配置文件里activation元素的说明）。格式是<name>value</name>。 -->
<properties />
</project>

```

☐ Maven 环境配置

Maven 构建生命周期 ☐

☐ 点我分享笔记

反馈/建议

Maven 构建生命周期

Maven 构建生命周期定义了一个项目构建跟发布的过程。

一个典型的 Maven 构建（build）生命周期是由以下几个阶段的序列组成的：

阶段	处理	描述
验证 <code>validate</code>	验证项目	验证项目是否正确且所有必须信息是可用的
编译 <code>compile</code>	执行编译	源代码编译在此阶段完成
测试 <code>Test</code>	测试	使用适当的单元测试框架（例如JUnit）运行测试。
包装 <code>package</code>	打包	创建JAR/WAR包如在 <code>pom.xml</code> 中定义提及的包
检查 <code>verify</code>	检查	对集成测试的结果进行检查，以保证质量达标
安装 <code>install</code>	安装	安装打包的项目到本地仓库，以供其他项目使用
部署 <code>deploy</code>	部署	拷贝最终的工程包到远程仓库中，以共享给其他开发人员和工程

为了完成 `default` 生命周期，这些阶段（包括其他未在上面罗列的生命周期阶段）将被按顺序地执行。

Maven 有以下三个标准的生命周期：

- clean**: 项目清理的处理
- default(或 build)**: 项目部署的处理
- site**: 项目站点文档创建的处理

构建阶段由插件目标构成

一个插件目标代表一个特定的任务（比构建阶段更为精细），这有助于项目的构建和管理。这些目标可能被绑定到多个阶段或者无绑定。不绑定到任何构建阶段的目标可以在构建生命周期之外通过直接调用执行。这些目标的执行顺序取决于调用目标和构建阶段的顺序。

例如，考虑下面的命令：

`clean` 和 `package` 是构建阶段，`dependency:copy-dependencies` 是目标

```
mvn clean dependency:copy-dependencies package
```

这里的 `clean` 阶段将会被首先执行，然后 `dependency:copy-dependencies` 目标会被执行，最终 `package` 阶段被执行。

Clean 生命周期

当我们执行 `mvn post-clean` 命令时，Maven 调用 `clean` 生命周期，它包含以下阶段：

- pre-clean**: 执行一些需要在`clean`之前完成的工作
- clean**: 移除所有上一次构建生成的文件
- post-clean**: 执行一些需要在`clean`之后立刻完成的工作

`mvn clean` 中的 `clean` 就是上面的 `clean`，在一个生命周期中，运行某个阶段的时候，它之前的所有阶段都会被运行，也就是说，`mvn clean` 等同于 `mvn pre-clean clean`，如果我们运行 `mvn post-clean`，那么 `pre-clean`，`clean` 都会被运行。

我们可以通过在上面的 `clean` 生命周期的任何阶段定义目标来修改这部分的操作行为。

在下面的例子中，我们将 `maven-antrun-plugin:run` 目标添加到 `pre-clean`、`clean` 和 `post-clean` 阶段中。这样我们可以在 `clean` 生命周期的各个阶段显示文本信息。

我们已经在 `C:\MVN\project` 目录下创建了一个 `pom.xml` 文件。

```

<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>
<groupId>com.companyname.projectgroup</groupId>
<artifactId>project</artifactId>
<version>1.0</version>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.1</version>
<executions>
<execution>
<id>id.pre-clean</id>
<phase>pre-clean</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>pre-clean phase</echo>
</tasks>
</configuration>
</execution>
<execution>
<id>id.clean</id>
<phase>clean</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>clean phase</echo>
</tasks>
</configuration>
</execution>
<execution>
<id>id.post-clean</id>
<phase>post-clean</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>post-clean phase</echo>
</tasks>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
</project>

```

现在打开命令控制台，跳转到 `pom.xml` 所在目录，并执行下面的 `mvn` 命令。

```
C:\MVN\project>mvn post-clean
```

Maven 将会开始处理并显示 `clean` 生命周期的所有阶段。

```
[INFO] Scanning for projects...
```

```
[INFO] -----
```

```
[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0

[INFO]    task-segment: [post-clean]

[INFO] -----

[INFO] [antrun:run {execution: id.pre-clean}]

[INFO] Executing tasks

    [echo] pre-clean phase

[INFO] Executed tasks

[INFO] [clean:clean {execution: default-clean}]

[INFO] [antrun:run {execution: id.clean}]

[INFO] Executing tasks

    [echo] clean phase

[INFO] Executed tasks

[INFO] [antrun:run {execution: id.post-clean}]

[INFO] Executing tasks

    [echo] post-clean phase

[INFO] Executed tasks

[INFO] -----

[INFO] BUILD SUCCESSFUL

[INFO] -----

[INFO] Total time: < 1 second

[INFO] Finished at: Sat Jul 07 13:38:59 IST 2012

[INFO] Final Memory: 4M/44M

[INFO] -----
```

你可以尝试修改 `mvn clean` 命令，来显示 `pre-clean` 和 `clean`，而在 `post-clean` 阶段不执行任何操作。

Default (Build) 生命周期

这是 **Maven** 的主要生命周期，被用于构建应用，包括下面的 23 个阶段：

生命周期阶段	描述
validate	检查工程配置是否正确，完成构建过程的所有必要信息是否能够获取到。
initialize	初始化构建状态，例如设置属性。
generate-sources	生成编译阶段需要包含的任何源码文件。
process-sources	处理源代码，例如，过滤任何值（ <code>filter any value</code> ）。
generate-resources	生成工程包中需要包含的资源文件。
process-resources	拷贝和处理资源文件到目的目录中，为打包阶段做准备。

生命周期阶段	描述
compile	编译工程源码。
process-classes	处理编译生成的文件，例如 Java Class 字节码的加强和优化。
generate-test-sources	生成编译阶段需要包含的任何测试源代码。
process-test-sources	处理测试源代码，例如，过滤任何值（ filter any values ）。
test-compile	编译测试源代码到测试目的目录。
process-test-classes	处理测试代码文件编译后生成的文件。
test	使用适当的单元测试框架（例如 JUnit ）运行测试。
prepare-package	在真正打包之前，为准备打包执行任何必要的操作。
package	获取编译后的代码，并按照可发布的格式进行打包，例如 JAR 、 WAR 或者 EAR 文件。
pre-integration-test	在集成测试执行之前，执行所需的操作。例如，设置所需的环境变量。
integration-test	处理和部署必须的工程包到集成测试能够运行的环境中。
post-integration-test	在集成测试被执行后执行必要的操作。例如，清理环境。
verify	运行检查操作来验证工程包是有效的，并满足质量要求。
install	安装工程包到本地仓库中，该仓库可以作为本地其他工程的依赖。
deploy	拷贝最终的工程包到远程仓库中，以共享给其他开发人员和工程。

有一些与 **Maven** 生命周期相关的重要概念需要说明：

当一个阶段通过 **Maven** 命令调用时，例如 **mvn compile**，只有该阶段之前以及包括该阶段在内的所有阶段会被执行。

不同的 **maven** 目标将根据打包的类型（**JAR** / **WAR** / **EAR**），被绑定到不同的 **Maven** 生命周期阶段。

在下面的例子中，我们将 **maven-antrun-plugin:run** 目标添加到 **Build** 生命周期的一部分阶段中。这样我们可以显示生命周期的文本信息。

我们已经更新了 **C:\MVN\project** 目录下的 **pom.xml** 文件。

```
<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>
<groupId>com.companyname.projectgroup</groupId>
<artifactId>project</artifactId>
<version>1.0</version>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.1</version>
<executions>
<execution>
<id>id.validate</id>
<phase>validate</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>validate phase</echo>
</tasks>
</configuration>
</execution>
<execution>
<id>id.compile</id>
```



```

<phase>compile</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>compile phase</echo>
</tasks>
</configuration>
</execution>
<execution>
<id>id.test</id>
<phase>test</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>test phase</echo>
</tasks>
</configuration>
</execution>
<execution>
<id>id.package</id>
<phase>package</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>package phase</echo>
</tasks>
</configuration>
</execution>
<execution>
<id>id.deploy</id>
<phase>deploy</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>deploy phase</echo>
</tasks>
</configuration>
</executions>
</plugin>
</plugins>
</build>
</project>

```

现在打开命令控制台，跳转到 `pom.xml` 所在目录，并执行以下 `mvn` 命令。

```
C:\MVN\project>mvn compile
```

Maven 将会开始处理并显示直到编译阶段的构建生命周期的各个阶段。

```

[INFO] Scanning for projects...

[INFO] -----

[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0

[INFO]     task-segment: [compile]

[INFO] -----

```

```
[INFO] [antrun:run {execution: id.validate}]

[INFO] Executing tasks

    [echo] validate phase

[INFO] Executed tasks

[INFO] [resources:resources {execution: default-resources}]

[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!

[INFO] skip non existing resourceDirectory C:\MVN\project\src\main\resources

[INFO] [compiler:compile {execution: default-compile}]

[INFO] Nothing to compile - all classes are up to date

[INFO] [antrun:run {execution: id.compile}]

[INFO] Executing tasks

    [echo] compile phase

[INFO] Executed tasks

[INFO] -----

[INFO] BUILD SUCCESSFUL

[INFO] -----

[INFO] Total time: 2 seconds

[INFO] Finished at: Sat Jul 07 20:18:25 IST 2012

[INFO] Final Memory: 7M/64M

[INFO] -----
```

命令行调用

在开发环境中，使用下面的命令去构建、安装工程到本地仓库

```
mvn install
```

这个命令在执行 **install** 阶段前，按顺序执行了 **default** 生命周期的阶段（**validate**, **compile**, **package**, 等等），我们只需要调用最后一个阶段，如这里是 **install**。

在构建环境中，使用下面的调用来纯净地构建和部署项目到共享仓库中

```
mvn clean deploy
```

这行命令也可以用于多模块的情况下，即包含多个子项目的项目，**Maven** 会在每一个子项目执行 **clean** 命令，然后再执行 **deploy** 命令。

Site 生命周期

Maven Site 插件一般用来创建新的报告文档、部署站点等。

pre-site: 执行一些需要在生成站点文档之前完成的工作

site: 生成项目的站点文档

post-site: 执行一些需要在生成站点文档之后完成的工作，并且为部署做准备

site-deploy: 将生成的站点文档部署到特定的服务器上

这里经常用到的是**site**阶段和**site-deploy**阶段，用以生成和发布Maven站点，这可是Maven相当强大的功能，Manager比较喜欢，文档及统计数据自动生成，很好看。在下面的例子中，我们将 `maven-antrun-plugin:run` 目标添加到 Site 生命周期的所有阶段中。这样我们可以显示生命周期的所有文本信息。

我们已经更新了 `C:\MVN\project` 目录下的 `pom.xml` 文件。

```
<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>
<groupId>com.companyname.projectgroup</groupId>
<artifactId>project</artifactId>
<version>1.0</version>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.1</version>
<executions>
<execution>
<id>id.pre-site</id>
<phase>pre-site</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>pre-site phase</echo>
</tasks>
</configuration>
</execution>
<execution>
<id>id.site</id>
<phase>site</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>site phase</echo>
</tasks>
</configuration>
</execution>
<execution>
<id>id.post-site</id>
<phase>post-site</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>post-site phase</echo>
</tasks>
</configuration>
</execution>
<execution>
<id>id.site-deploy</id>
<phase>site-deploy</phase>
<goals>
<goal>run</goal>
</goals>
</execution>
</build>
</project>
```

```
<configuration>
<tasks>
<echo>site-deploy phase</echo>
</tasks>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
</project>
```

现在打开命令控制台，跳转到 `pom.xml` 所在目录，并执行以下 `mvn` 命令。

```
C:\MVN\project>mvn site
```

Maven 将会开始处理并显示直到 `site` 阶段的 `site` 生命周期的各个阶段。

```
[INFO] Scanning for projects...

[INFO] -----

[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0

[INFO]    task-segment: [site]

[INFO] -----

[INFO] [antrun:run {execution: id.pre-site}]

[INFO] Executing tasks

    [echo] pre-site phase

[INFO] Executed tasks

[INFO] [site:site {execution: default-site}]

[INFO] Generating "About" report.

[INFO] Generating "Issue Tracking" report.

[INFO] Generating "Project Team" report.

[INFO] Generating "Dependencies" report.

[INFO] Generating "Project Plugins" report.

[INFO] Generating "Continuous Integration" report.

[INFO] Generating "Source Repository" report.

[INFO] Generating "Project License" report.

[INFO] Generating "Mailing Lists" report.

[INFO] Generating "Plugin Management" report.

[INFO] Generating "Project Summary" report.

[INFO] [antrun:run {execution: id.site}]

[INFO] Executing tasks
```

[echo] site phase

[INFO] Executed tasks

[INFO] -----

[INFO] BUILD SUCCESSFUL

[INFO] -----

[INFO] Total time: 3 seconds

[INFO] Finished at: Sat Jul 07 15:25:10 IST 2012

[INFO] Final Memory: 24M/149M

[INFO] -----

[Maven POM](#)

Maven 构建配置文件 [Maven POM](#)

[点我分享笔记](#)

反馈/建议

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号: 闽ICP备15012807号-1

[Maven POM](#)

[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Maven 构建生命周期](#)

Maven 仓库 [Maven POM](#)

Maven 构建配置文件

构建配置文件是一系列的配置项的值，可以用来设置或者覆盖 **Maven** 构建默认值。使用构建配置文件，你可以为不同的环境，比如说生产环境（**Production**）和开发（**Development**）环境，定制构建方式。

配置文件在 **pom.xml** 文件中使用 **activeProfiles** 或者 **profiles** 元素指定，并且可以通过各种方式触发。配置文件在构建时修改 **POM**，并且用来给参数设定不同的目标环境（比如说，开发（**Development**）、测试（**Testing**）和生产环境（**Production**）中数据库服务器的地址）。

构建配置文件的类型

构建配置文件大体上有三种类型：

类型	在哪定义
项目级（Per Project）	定义在项目的POM文件pom.xml中
用户级（Per User）	定义在Maven的设置xml文件中 (%USER_HOME%/m2/settings.xml)
全局（Global）	定义在 Maven 全局的设置 xml 文件中 (%M2_HOME%/conf/settings.xml)

配置文件激活

Maven的构建配置文件可以通过多种方式激活。

使用命令控制台输入显式激活。

通过 `maven` 设置。

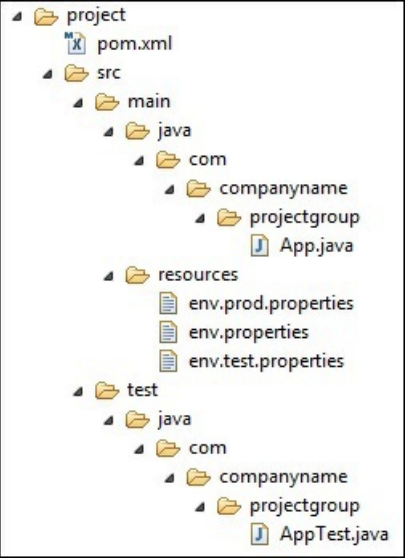
基于环境变量（用户或者系统变量）。

操作系统设置（比如说，Windows系列）。

文件的存在或者缺失。

配置文件激活实例

假定项目结构如下：



其中在src/main/resources文件夹下有三个用于测试文件：

文件名	描述
env.properties	如果未指定配置文件时默认使用的配置。
env.test.properties	当测试配置文件使用时的测试配置。
env.prod.properties	当生产配置文件使用时的生产配置。

注意：这三个配置文件并不是代表构建配置文件的函数，而是用于本次测试的目的；比如，我指定了构建配置文件为 `prod` 时，项目就使用 `envprod.pr` `operties` 文件。

注意：下面的例子仍然是使用 `AntRun` 插件，因为此插件能绑定 `Maven` 生命周期阶段，并通过 `Ant` 的标签不用编写一点代码即可输出信息、复制文件等，经此而已。其余的与本次构建配置文件无关。

1、配置文件激活

`profile` 可以让我们定义一系列的配置信息，然后指定其激活条件。这样我们就可以定义多个 `profile`，然后每个 `profile` 对应不同的激活条件和配置信息，从而达到不同环境使用不同配置信息的效果。

以下实例，我们将 `maven-antrun-plugin:run` 目标添加到测试阶段中。这样可以在不同的 `profile` 中输出文本信息。我们将使用 `pom.xml` 来定义不同的 `profile`，并在命令控制台使用 `maven` 命令激活 `profile`。

`pom.xml` 文件如下：

```
<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.jsoft.test</groupId>
<artifactId>testproject</artifactId>
<packaging>jar</packaging>
<version>0.1-SNAPSHOT</version>
<name>testproject</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>
<profiles>
<profile>
<id>test</id>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.8</version>
<executions>
<execution>
<phase>test</phase>
<goals>
<goal>run</goal>
</goals>
</configuration>
<tasks>
<echo>Using env.test.properties</echo>
<copy file="src/main/resources/env.test.properties" tofile="${project.build.outputDirectory}/env.properties" overwrite="true"/>
</tasks>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
</profile>
<profile>
<id>normal</id>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.8</version>
<executions>
<execution>
<phase>test</phase>
<goals>
<goal>run</goal>
</goals>
</configuration>
<tasks>
<echo>Using env.properties</echo>
<copy file="src/main/resources/env.properties" tofile="${project.build.outputDirectory}/env.properties" overwrite="true"/>
</tasks>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
</profile>
<profile>
<id>prod</id>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.8</version>
<executions>
<execution>
<phase>test</phase>
<goals>
<goal>run</goal>
</goals>
```

```

<configuration>
<tasks>
<echo>Using env.prod.properties</echo>
<copy file="src/main/resources/env.prod.properties" tofile="${project.build.outputDirectory}/env.properties" overwrite="true"/>
</tasks>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
</profile>
</profiles>
</project>

```

注意：构建配置文件采用的是 **<profiles>** 节点。

说明：上面新建了三个 **<profiles>**，其中 **<id>** 区分了不同的 **<profiles>** 执行不同的 AntRun 任务；而 AntRun 的任务可以这么理解，AntRun 监听 test 的 Maven 生命周期阶段，当 Maven 执行 test 时，就除了发 AntRun 的任务，任务里面为输出文本并复制文件到指定的位置；而至于要执行哪个 AntRun 任务，此时构建配置文件起到了传输指定的作用，比如，通过命令行参数输入指定的 **<id>**。

执行命令：

```
mvn test -Ptest
```

提示：第一个 test 为 Maven 生命周期阶段，第 2 个 test 为构建配置文件指定的 **<id>** 参数，这个参数通过 **-P** 来传输，当然，它可以是 prod 或者 normal 这些由你定义的 **<id>**。

运行的结果如下：

```

D:\开发工程\Github\5_java_example\maventest\test4\test4\testproject
λ mvn test -Ptest
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building testproject 0.1-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ testproject ---
[WARNING] Using platform encoding (GBK actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] Copying 3 resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ testproject ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ testproject ---
[WARNING] Using platform encoding (GBK actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory D:\开发工程\Github\5_java_example\maventest\test4\test4\testproject\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile) @ testproject ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ testproject ---
[INFO] Surefire report directory: D:\开发工程\Github\5_java_example\maventest\test4\test4\testproject\target\surefire-reports

-----
T E S T S
-----
Running com.jssoft.test.AppTest
Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.004 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-antrun-plugin:1.8:run (default) @ testproject ---
[WARNING] Parameter tasks is deprecated, use target instead
[INFO] Executing tasks

main:
[echo] Using env.test.properties
[copy] Copying 1 file to D:\开发工程\Github\5_java_example\maventest\test4\test4\testproject\target\classes
[INFO] Executed tasks
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.100s
[INFO] Finished at: Tue May 09 04:50:30 CST 2017
[INFO] Final Memory: 12M/491M
[INFO] -----

```

可以看出成功的触发了 AntRun 的任务。并且是对应构建配置文件下的 **<id>** 为 test 的任务。

再测试其余两个命令，结果如下：

□


```

D:\开发工程\GitHub\5_java_example\maventest\test4\test4\testproject
λ mvn test -Pnormal
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building testproject 0.1-SNAPSHOT
[INFO] -----
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ testproject ---
[WARNING] Using platform encoding (GBK actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] Copying 3 resources
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ testproject ---
[INFO] Nothing to compile - all classes are up to date
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ testproject ---
[WARNING] Using platform encoding (GBK actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory D:\开发工程\GitHub\5_java_example\maventest\test4\test4\testproject\src\test\resources
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile) @ testproject ---
[INFO] Nothing to compile - all classes are up to date
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ testproject ---
[INFO] Surefire report directory: D:\开发工程\GitHub\5_java_example\maventest\test4\test4\testproject\target\surefire-reports

-----
T E S T S
-----
Running com.jssoft.test.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.005 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] --- maven-antrun-plugin:1.8:run (default) @ testproject ---
[WARNING] Parameter tasks is deprecated, use target instead
[INFO] Executing tasks

main:
[echo] Using env.properties
[copy] Copying 1 file to D:\开发工程\GitHub\5_java_example\maventest\test4\test4\testproject\target\classes
[INFO] Executed tasks
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.011s
[INFO] Finished at: Tue May 09 04:53:19 CST 2017
[INFO] Final Memory: 12M/491M
[INFO] -----

```

2、通过Maven设置激活配置文件

打开 `%USER_HOME%\.m2` 目录下的 `settings.xml` 文件，其中 `%USER_HOME%` 代表用户主目录。如果 `setting.xml` 文件不存在就直接拷贝 `%M2_HOME%\conf\settings.xml` 到 `.m2` 目录，其中 `%M2_HOME%` 代表 Maven 的安装目录。

配置 `setting.xml` 文件，增加 `<activeProfiles>` 属性：

```

<settings 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/settings-1.0.0.xsd">
...
<activeProfiles>
<activeProfile>test</activeProfile>
</activeProfiles>
</settings>

```

执行命令：

```
mvn test
```

提示 1：此时不需要使用 `-Ptest` 来输入参数了，上面的 `setting.xml` 文件的 `<activeprofile>` 已经指定了 `test` 参数代替了。

提示 2：同样可以使用在 `%M2_HOME%\conf\settings.xml` 的文件进行配置，效果一致。

执行结果：

```

D:\开发工程\GitHub\5_java_example\maventest\test4\test4\testproject
λ mvn test
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building testproject 0.1-SNAPSHOT
[INFO] -----
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ testproject ---
[WARNING] Using platform encoding (GBK actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] Copying 3 resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ testproject ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ testproject ---
[WARNING] Using platform encoding (GBK actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory D:\开发工程\GitHub\5_java_example\maventest\test4\test4\testproject\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile) @ testproject ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ testproject ---
[INFO] Surefire report directory: D:\开发工程\GitHub\5_java_example\maventest\test4\test4\testproject\target\surefire-reports

-----
T E S T S
-----
Running com.jssoft.test.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.002 sec, settings.xml到m2目录, 其中%M2_HOME%代表Maven的安装目录

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-antrun-plugin:1.8:run (default) @ testproject ---
[WARNING] Parameter tasks is deprecated, use target instead
[INFO] Executing tasks

main:
  [echo] Using env.test.properties
  [copy] Copying 1 file to D:\开发工程\GitHub\5_java_example\maventest\test4\test4\testproject\target\classes
[INFO] Executed tasks
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.179s
[INFO] Finished at: Tue May 09 05:03:46 CST 2017
[INFO] Final Memory: 12M/491M
[INFO] -----

```

3、通过环境变量激活配置文件

先把上一步测试的 setting.xml 值全部去掉。

然后在 pom.xml 里面的 <id> 为 test 的 <profile> 节点, 加入 <activation> 节点:

```

<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.jssoft.test</groupId>
  <artifactId>testproject</artifactId>
  <packaging>jar</packaging>
  <version>0.1-SNAPSHOT</version>
  <name>testproject</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>
  <profiles>
    <profile>
      <id>test</id>
      <activation>
        <property>
          <name>env</name>
          <value>test</value>
        </property>
      </activation>
      <build>
        <plugins>
          <plugin>

```

```
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.8</version>
<executions>
<execution>
<phase>test</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>Using env.test.properties</echo>
<copy file="src/main/resources/env.test.properties" tofile="${project.build.outputDirectory}/env.properties" overwrite="true"/>
</tasks>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
</profile>
<profile>
<id>normal</id>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.8</version>
<executions>
<execution>
<phase>test</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>Using env.properties</echo>
<copy file="src/main/resources/env.properties" tofile="${project.build.outputDirectory}/env.properties" overwrite="true"/>
</tasks>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
</profile>
<profile>
<id>prod</id>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.8</version>
<executions>
<execution>
<phase>test</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>Using env.prod.properties</echo>
<copy file="src/main/resources/env.prod.properties" tofile="${project.build.outputDirectory}/env.properties" overwrite="true"/>
</tasks>
</configuration>
</execution>
</executions>
```

```
</plugin>
</plugins>
</build>
</profile>
</profiles>
</project>
```

执行命令：

```
mvn test -Denv=test
```

提示 1: 上面使用 -D 传递环境变量，其中 env 对应刚才设置的 <name> 值，test 对应 <value>。

提示 2: 在 Windows 10 上测试了系统的环境变量，但是不生效，所以，只能通过 -D 传递。

执行结果：

```
D:\开发工程\GitHub\5_java_example\maventest\test4\test4\testproject
λ mvn test -Denv=test
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building testproject 0.1-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ testproject ---
[WARNING] Using platform encoding (GBK actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] Copying 3 resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ testproject ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ testproject ---
[WARNING] Using platform encoding (GBK actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory D:\开发工程\GitHub\5_java_example\maventest\test4\test4\testproject\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile) @ testproject ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ testproject ---
[INFO] Surefire report directory: D:\开发工程\GitHub\5_java_example\maventest\test4\test4\testproject\target\surefire-reports

-----
T E S T S
-----
Running com.jssoft.test.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-antrun-plugin:1.8:run (default) @ testproject ---
[WARNING] Parameter tasks is deprecated, use target instead
[INFO] Executing tasks

main:
[echo] Using env.test.properties
[copy] Copying 1 file to D:\开发工程\GitHub\5_java_example\maventest\test4\test4\testproject\target\classes
[INFO] Executed tasks
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.044s
[INFO] Finished at: Tue May 09 05:08:42 CST 2017
[INFO] Final Memory: 12M/491M
[INFO] -----
```

4、通过操作系统激活配置文件

activation 元素包含下面的操作系统信息。当系统为 windows XP 时，test Profile 将会被触发。

```
<profile>
<id>test</id>
<activation>
<os>
<name>Windows XP</name>
<family>Windows</family>
<arch>x86</arch>
<version>5.1.2600</version>
</os>
</activation>
</profile>
```

现在打开命令控制台，跳转到 pom.xml 所在目录，并执行下面的 mvn 命令。不要使用 -P 选项指定 Profile 的名称。Maven 将显示被激活的 test Profile 的结果。

```
mvn test
```

5、通过文件的存在或者缺失激活配置文件

现在使用 **activation** 元素包含下面的操作系统信息。当 `target/generated-sources/axistools/wsdl2java/com/companyname/group` 缺失时，**test Profile** 将会被触发。

```
<profile>
<id>test</id>
<activation>
<file>
<missing>target/generated-sources/axistools/wsdl2java/
com/companyname/group</missing>
</file>
</activation>
</profile>
```

现在打开命令控制台，跳转到 `pom.xml` 所在目录，并执行下面的 `mvn` 命令。不要使用 `-P` 选项指定 **Profile** 的名称。**Maven** 将显示被激活的 **test Profile** 的结果。

```
mvn test
```

参考：<https://www.cnblogs.com/EasonJim/p/6828743.html>

[Maven 构建生命周期](#)

[Maven 仓库](#)

[点我分享笔记](#)

反馈/建议

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Maven 构建配置文件](#)

[Maven 插件](#)

Maven 仓库

在 **Maven** 的术语中，仓库是一个位置（**place**）。

Maven 仓库是项目中依赖的第三方库，这个库所在的位置叫做仓库。

在 **Maven** 中，任何一个依赖、插件或者项目构建的输出，都可以称之为构件。

Maven 仓库能帮助我们管理构件（主要是 **JAR**），它就是放置所有 **JAR** 文件（**WAR**，**ZIP**，**POM**等等）的地方。

Maven 仓库有三种类型：

本地（**local**）

中央（**central**）

远程（**remote**）

本地仓库

Maven 的本地仓库，在安装 Maven 后并不会创建，它是在第一次执行 maven 命令的时候才被创建。

运行 Maven 的时候，Maven 所需要的任何构件都是直接从本地仓库获取的。如果本地仓库没有，它会首先尝试从远程仓库下载构件至本地仓库，然后再使用本地仓库的构件。

默认情况下，不管Linux还是 Windows，每个用户在自己的用户目录下都有一个路径名为 .m2/respository/ 的仓库目录。

Maven 本地仓库默认被创建在 %USER_HOME% 目录下。要修改默认位置，在 %M2_HOME%\conf 目录中的 Maven 的 settings.xml 文件中定义另一个路径。

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
<localRepository>C:/MyLocalRepository</localRepository>
</settings>
```

当你运行 Maven 命令，Maven 将下载依赖的文件到你指定的路径中。

中央仓库

Maven 中央仓库是由 Maven 社区提供的仓库，其中包含了大量常用的库。

中央仓库包含了绝大多数流行的开源Java构件，以及源码、作者信息、SCM、信息、许可证信息等。一般来说，简单的Java项目依赖的构件都可以在这里下载到。

中央仓库的关键概念：

- 这个仓库由 Maven 社区管理。
- 不需要配置。
- 需要通过网络才能访问。

要浏览中央仓库的内容，maven 社区提供了一个 URL: <http://search.maven.org/#browse>。使用这个仓库，开发人员可以搜索所有可以获取的代码库。

远程仓库

如果 Maven 在中央仓库中也找不到依赖的文件，它会停止构建过程并输出错误信息到控制台。为避免这种情况，Maven 提供了远程仓库的概念，它是开发人员自己定制仓库，包含了所需要的代码库或者其他工程中用到的 jar 文件。

举例说明，使用下面的 pom.xml，Maven 将从远程仓库中下载该 pom.xml 中声明的所依赖的（在中央仓库中获取不到的）文件。

```
<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>
<groupId>com.companyname.projectgroup</groupId>
<artifactId>project</artifactId>
<version>1.0</version>
<dependencies>
<dependency>
<groupId>com.companyname.common-lib</groupId>
<artifactId>common-lib</artifactId>
<version>1.0.0</version>
</dependency>
<dependencies>
<repositories>
<repository>
<id>companyname.lib1</id>
<url>http://download.companyname.org/maven2/lib1</url>
</repository>
<repository>
<id>companyname.lib2</id>
<url>http://download.companyname.org/maven2/lib2</url>
</repository>
</repositories>
</project>
```

Maven 依赖搜索顺序

当我们执行 **Maven** 构建命令时，**Maven** 开始按照以下顺序查找依赖的库：

步骤 1 — 在本地仓库中搜索，如果找不到，执行步骤 2，如果找到了则执行其他操作。

步骤 2 — 在中央仓库中搜索，如果找不到，并且有一个或多个远程仓库已经设置，则执行步骤 4，如果找到了则下载到本地仓库中已被将来引用。

步骤 3 — 如果远程仓库没有被设置，**Maven** 将简单的停滞处理并抛出错误（无法找到依赖的文件）。

步骤 4 — 在一个或多个远程仓库中搜索依赖的文件，如果找到则下载到本地仓库已被将来引用，否则 **Maven** 将停止处理并抛出错误（无法找到依赖的文件）。

Maven 阿里云(Aliyun)仓库

Maven 仓库默认在国外，国内使用难免很慢，我们可以更换为阿里云的仓库。

第一步:修改 **maven** 根目录下的 **conf** 文件夹中的 **setting.xml** 文件，在 **mirrors** 节点上，添加内容如下：

```
<mirrors>
<mirror>
<id>alimaven</id>
<name>aliyun maven</name>
<url>http://maven.aliyun.com/nexus/content/groups/public/</url>
<mirrorOf>central</mirrorOf>
</mirror>
</mirrors>
```

```
<!-- mirror
| Specifies a repository mirror site to use instead of a given repository. In
| this mirror serves has an ID that matches the mirrorOf element of this mirr
| for inheritance and direct lookup purposes, and must be unique across the
|
|
<mirror>
  <id>mirrorId</id>
  <mirrorOf>repositoryId</mirrorOf>
  <name>Human Readable Name for this Mirror.</name>
  <url>http://my.repository.com/repo/path</url>
</mirror>
-->
<mirror>
  <id>alimaven</id>
  <name>aliyun maven</name>
  <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
  <mirrorOf>central</mirrorOf>
</mirror>
</mirrors>
```

第二步: **pom.xml**文件里添加：

```
<repositories>
<repository>
<id>alimaven</id>
<name>aliyun maven</name>
<url>http://maven.aliyun.com/nexus/content/groups/public/</url>
<releases>
<enabled>true</enabled>
</releases>
<snapshots>
<enabled>false</enabled>
</snapshots>
</repository>
</repositories>
```

☐ Maven 构建配置文件

Maven 插件 ☐

☐ 点我分享笔记



Maven 插件

Maven 有以下三个标准的生命周期：

- clean:** 项目清理的处理
- default(或 build):** 项目部署的处理
- site:** 项目站点文档创建的处理

每个生命周期中都包含着一系列的阶段(phase)。这些 phase 就相当于 Maven 提供的统一的接口，然后这些 phase 的实现由 Maven 的插件来完成。我们在输入 mvn 命令的时候 比如 `mvn clean`，clean 对应的就是 Clean 生命周期中的 clean 阶段。但是 clean 的具体操作是由 `maven-clean-plugin` 来实现的。

所以说 Maven 生命周期的每一个阶段的具体实现都是由 Maven 插件实现的。

Maven 实际上是一个依赖插件执行的框架，每个任务实际上是由插件完成。Maven 插件通常被用来：

- 创建 jar 文件
- 创建 war 文件
- 编译代码文件
- 代码单元测试
- 创建工程文档
- 创建工程报告

插件通常提供了一个目标的集合，并且可以使用下面的语法执行：

```
<code>mvn [plugin-name]:[goal-name]</code>
```

例如，一个 Java 工程可以使用 `maven-compiler-plugin` 的 `compile-goal` 编译，使用以下命令：

```
<code>mvn compiler:compile</code>
```

插件类型

Maven 提供了下面两种类型的插件：

类型	描述
Build plugins	在构建时执行，并在 pom.xml 的元素中配置。
Reporting plugins	在网站生成过程中执行，并在 pom.xml 的元素中配置。

下面是一些常用插件的列表：

插件	描述
clean	构建之后清理目标文件。删除目标目录。

插件	描述
compiler	编译 Java 源文件。
surefile	运行 JUnit 单元测试。创建测试报告。
jar	从当前工程中构建 JAR 文件。
war	从当前工程中构建 WAR 文件。
javadoc	为工程生成 Javadoc。
antrun	从构建过程的任意一个阶段中运行一个 ant 任务的集合。

实例

我们已经在我们的例子中大量使用了 **maven-antrun-plugin** 来输出数据到控制台上。请查看 [Maven - 构建配置文件](#) 章节。让我们用一种更好的方式理解这部分内容，在 C:\MVN\project 目录下创建一个 pom.xml 文件。

```
<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>
<groupId>com.companyname.projectgroup</groupId>
<artifactId>project</artifactId>
<version>1.0</version>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.1</version>
<executions>
<execution>
<id>id.clean</id>
<phase>clean</phase>
<goals>
<goal>run</goal>
</goals>
<configuration>
<tasks>
<echo>clean phase</echo>
</tasks>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
</project>
```

接下来，打开命令终端跳转到 pom.xml 所在的目录，并执行下面的 mvn 命令。

```
mvn clean
```

Maven 将开始处理并显示 clean 生命周期的 clean 阶段。

```
[INFO] Scanning for projects...

[INFO] -----

[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0

[INFO] task-segment: [post-clean]

[INFO] -----
```

```
[INFO] [clean:clean {execution: default-clean}]

[INFO] [antrun:run {execution: id.clean}]

[INFO] Executing tasks

    [echo] clean phase

[INFO] Executed tasks

[INFO] -----

[INFO] BUILD SUCCESSFUL

[INFO] -----

[INFO] Total time: < 1 second

[INFO] Finished at: Sat Jul 07 13:38:59 IST 2012

[INFO] Final Memory: 4M/44M

[INFO] -----
```

上面的例子展示了以下关键概念：

插件是在 `pom.xml` 中使用 `plugins` 元素定义的。

每个插件可以有多个目标。

你可以定义阶段，插件会使用它的 `phase` 元素开始处理。我们已经使用了 `clean` 阶段。

你可以通过绑定到插件的目标的方式来配置要执行的任务。我们已经绑定了 `echo` 任务到 `maven-antrun-plugin` 的 `run` 目标。

就是这样，**Maven** 将处理剩下的事情。它将下载本地仓库中获取不到的插件，并开始处理。

[☐ Maven 仓库](#)

[Maven 构建 Java 项目 ☐](#)

[☐ 点我分享笔记](#)

反馈/建议



[☐ Maven 插件](#)

[Maven 构建 & 项目测试 ☐](#)

Maven 构建 Java 项目

Maven 使用原型 `archetype` 插件创建项目。要创建一个简单的 `Java` 应用，我们将使用 `maven-archetype-quickstart` 插件。

在下面的例子中，我们将在 `C:\MVN` 文件夹下创建一个基于 `maven` 的 `java` 应用项目。

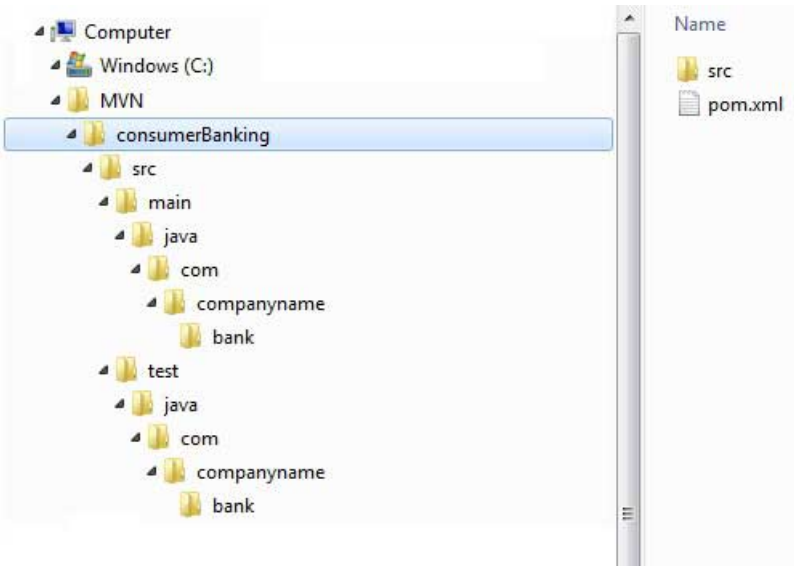
命令格式如下：

```
mvn archetype:generate -DgroupId=com.companyname.bank -DartifactId=consumerBanking -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

参数说明：

- DgroupId: 组织名，公司网址的反写 + 项目名称
- DartifactId: 项目名-模块名
- DarchetypeArtifactId: 指定 ArchetypeId, maven-archetype-quickstart, 创建一个简单的 Java 应用
- DinteractiveMode: 是否使用交互模式

生成的文件夹结构如下：



各个文件夹说明：

文件夹结构	描述
consumerBanking	包含 src 文件夹和 pom.xml
src/main/java contains	java 代码文件在包结构下（com/companyName/bank）。
src/main/test contains	测试代码文件在包结构下（com/companyName/bank）。
src/main/resources	包含了 图片 / 属性 文件（在上面的例子中，我们需要手动创建这个结构）。

在 C:\MVN\consumerBanking\src\main\java\com\companyname\bank 文件夹中，可以看到一个 App.java，代码如下：

App.java

```
package com.companyname.bank;
/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

打开 C:\MVN\consumerBanking\src\test\java\com\companyname\bank 文件夹，可以看到 Java 测试文件 AppTest.java。

AppTest.java

```
package com.companyname.bank;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
```

```
/**
 * Unit test for simple App.
 */
public class AppTest extends TestCase
{
    /**
     * Create the test case
     *
     * @param testName name of the test case
     */
    public AppTest( String testName )
    {
        super( testName );
    }
    /**
     * @return the suite of tests being tested
     */
    public static Test suite()
    {
        return new TestSuite( AppTest.class );
    }
    /**
     * Rigourous Test :-))
     */
    public void testApp()
    {
        assertTrue( true );
    }
}
```

接下来的开发过程中我们只需要按照上面表格中提到的结构放置好，其他的事情 **Maven** 帮我们将会搞定。

☐ Maven 插件

Maven 构建 & 项目测试 ☐

☐ 点我分享笔记

反馈/建议

Copyright © 2013-2018 菜鸟教程 runoob.com All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

☐ Maven 构建 Java 项目

Maven 引入外部依赖 ☐

Maven 构建 & 项目测试

在上一章节中我们学会了如何使用 **Maven** 创建 **Java** 应用。接下来我们要学习如何构建和测试这个项目。

进入 **C:/MVN** 文件夹下，打开 **consumerBanking** 文件夹。你将看到有一个 **pom.xml** 文件，代码如下：

```
<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.companyname.bank</groupId>
<artifactId>consumerBanking</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<name>consumerBanking</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>
</project>
```

从以上 xml 代码中，可知 Maven 已经添加了 JUnit 作为测试框架。

默认情况下 Maven 添加了一个源码文件 `C:\MVN\consumerBanking\src\main\java\com\companyname\bank\App.java` 和一个测试文件 `C:\MVN\consumerBanking\src\test\java\com\companyname\bank\AppTest.java`。

打开命令控制台，跳转到 `C:\MVN\consumerBanking` 目录下，并执行以下 `mvn` 命令开始构建项目：

```
C:\MVN\consumerBanking>mvn clean package

[INFO] Scanning for projects...

[INFO] -----

[INFO] Building consumerBanking

[INFO]    task-segment: [clean, package]

[INFO] -----

[INFO] [clean:clean {execution: default-clean}]

[INFO] Deleting directory C:\MVN\consumerBanking\target

...

...

...

[INFO] [jar:jar {execution: default-jar}]

[INFO] Building jar: C:\MVN\consumerBanking\target\

consumerBanking-1.0-SNAPSHOT.jar

[INFO] -----

[INFO] BUILD SUCCESSFUL

[INFO] -----

[INFO] Total time: 2 seconds

[INFO] Finished at: Tue Jul 10 16:52:18 IST 2012

[INFO] Final Memory: 16M/89M

[INFO] -----
```

执行完后，我们已经构建了自己的项目并创建了最终的 `jar` 文件，下面是要学习的关键概念：

我们给了 `maven` 两个目标，首先清理目标目录（`clean`），然后打包项目构建的输出为 `jar`（`package`）文件。

打包好的 `jar` 文件可以在 `consumerBanking\target` 中获得，名称为 `consumerBanking-1.0-SNAPSHOT.jar`。

测试报告存放在 `consumerBanking\target\surefire-reports` 文件夹中。

`Maven` 编译源码文件，以及测试源码文件。

接着 **Maven** 运行测试用例。

最后 **Maven** 创建项目包。

```
C:\MVN\consumerBanking\target\classes>java com.companyname.bank.App
```

你可以看到结果：

```
Hello World!
```

添加 **Java** 源文件

接下来我们看看如何添加其他的 **Java** 文件到项目中。打开 `C:\MVN\consumerBanking\src\main\java\com\companyname\bank` 文件夹，在其中创建 **Util** 类 `Util.java`。

Util.java

```
package com.companyname.bank;
public class Util
{
    public static void printMessage(String message){
        System.out.println(message);
    }
}
```

更新 **App** 类来使用 **Util** 类：

App.java

```
package com.companyname.bank;
/**
 * Hello world!
 */
public class App
{
    public static void main( String[] args )
    {
        Util.printMessage("Hello World!");
    }
}
```

现在打开命令控制台，跳转到 `C:\MVN\consumerBanking` 目录下，并执行下面的 **mvn** 命令。

```
C:\MVN\consumerBanking>mvn clean compile
```

在 **Maven** 构建成功之后，跳转到 `C:\MVN\consumerBanking\target\classes` 目录下，并执行下面的 **java** 命令。

```
C:\MVN\consumerBanking\target\classes>java -cp com.companyname.bank.App
```

你可以看到结果：

```
Hello World!
```

☐ [Maven 构建 Java 项目](#)

[Maven 引入外部依赖](#) ☐

☐ [点我分享笔记](#)

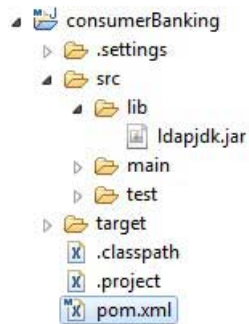
[反馈/建议](#)

Maven 引入外部依赖

如果我们需要引入第三库文件到项目，该怎么操作呢？

pom.xml 的 dependencies 列表列出了我们的项目需要构建的所有外部依赖项。

要添加依赖项，我们一般是先在 src 文件夹下添加 lib 文件夹，然后将你工程需要的 jar 文件复制到 lib 文件夹下。我们使用的是 ldapjdk.jar，它是为 LDAP 操作的一个帮助库：



然后添加以下依赖到 pom.xml 文件中：

```
<dependencies>
<!-- 在这里添加你的依赖 -->
<dependency>
<groupId>ldapjdk</groupId> <!-- 库名称，也可以自定义 -->
<artifactId>ldapjdk</artifactId> <!-- 库名称，也可以自定义 -->
<version>1.0</version> <!-- 版本号 -->
<scope>system</scope> <!-- 作用域 -->
<systemPath>${basedir}\src\lib\ldapjdk.jar</systemPath> <!-- 项目根目录下的 lib 文件夹下 -->
</dependency>
</dependencies>
```

pom.xml 文件完整代码如下：

```
<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.companyname.bank</groupId>
<artifactId>consumerBanking</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<name>consumerBanking</name>
<url>http://maven.apache.org</url>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>ldapjdk</groupId>
<artifactId>ldapjdk</artifactId>
<scope>system</scope>
<version>1.0</version>
```

```
<systemPath>${basedir}\src\lib\ldapjdk.jar</systemPath>
</dependency>
</dependencies>
</project>
```

[Maven 构建 & 项目测试](#)

Maven 项目模板 [□](#)

[□ 点我分享笔记](#)

反馈/建议

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号: 闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Maven 引入外部依赖](#)

Maven 项目文档 [□](#)

Maven 项目模板

Maven 使用 **archetype**(原型) 来创建自定义的项目结构, 形成 **Maven** 项目模板。

在前面章节我们学到 **Maven** 使用下面的命令来快速创建 **java** 项目:

```
mvn archetype:generate
```

什么是 **archetype**?

archetype 也就是原型, 是一个 **Maven** 插件, 准确说是一个项目模板, 它的任务是根据模板创建一个项目结构。我们将使用 **quickstart** 原型插件创建一个简单的 **java** 应用程序。

使用项目模板

让我们打开命令控制台, 跳转到 **C:\> MVN** 目录并执行以下 **mvn** 命令:

```
C:\MVN> mvn archetype:generate
```

Maven 将开始处理, 并要求选择所需的原型:

```
[INFO] Scanning for projects...

[INFO] Searching repository for plugin with prefix: 'archetype'.

[INFO] -----

[INFO] Building Maven Default Project

[INFO]task-segment: [archetype:generate] (aggregator-style)

[INFO] -----

[INFO] Preparing archetype:generate
```


...

600: remote -> org.trailsframework:trails-archetype (-)

601: remote -> org.trailsframework:trails-secure-archetype (-)

602: remote -> org.tynamo:tynamo-archetype (-)

603: remote -> org.wicketstuff.scala:wicket-scala-archetype (-)

604: remote -> org.wicketstuff.scala:wicketstuff-scala-archetype

Basic setup for a project that combines Scala and Wicket,

depending on the Wicket-Scala project.

Includes an example Specs test.)

605: remote -> org.wikbook:wikbook.archetype (-)

606: remote -> org.xaloon.archetype:xaloon-archetype-wicket-jpa-glassfish (-)

607: remote -> org.xaloon.archetype:xaloon-archetype-wicket-jpa-spring (-)

608: remote -> org.xwiki.commons:xwiki-commons-component-archetype

(Make it easy to create a maven project for creating XWiki Components.)

609: remote -> org.xwiki.rendering:xwiki-rendering-archetype-macro

(Make it easy to create a maven project for creating XWiki Rendering Macros.)

610: remote -> org.zkoss:zk-archetype-component (The ZK Component archetype)

611: remote -> org.zkoss:zk-archetype-webapp (The ZK webapp archetype)

612: remote -> ru.circumflex:circumflex-archetype (-)

613: remote -> se.vgregion.javg.maven.archetypes:javg-minimal-archetype (-)

614: remote -> sk.seges.sesam:sesam-annotation-archetype (-)

Choose a number or apply filter

(format: [groupId:]artifactId, case sensitive contains): 203:

按下 **Enter** 选择默认选项 (203:maven-archetype-quickstart)。

Maven 将询问原型的版本

Choose org.apache.maven.archetypes:maven-archetype-quickstart version:

1: 1.0-alpha-1

2: 1.0-alpha-2

3: 1.0-alpha-3

4: 1.0-alpha-4

5: 1.0

6: 1.1

Choose a number: 6:

按下 **Enter** 选择默认选项 (6:maven-archetype-quickstart:1.1)

Maven 将询问项目细节。按要求输入项目细节。如果要使用默认值则直接按 **Enter** 键。你也可以输入自己的值。

Define value for property 'groupId': : com.companyname.insurance

Define value for property 'artifactId': : health

Define value for property 'version': 1.0-SNAPSHOT

Define value for property 'package': com.companyname.insurance

Maven 将要求确认项目细节，按 **Enter** 或按 Y

Confirm properties configuration:

groupId: com.companyname.insurance

artifactId: health

version: 1.0-SNAPSHOT

package: com.companyname.insurance

Y:

现在 Maven 将开始创建项目结构，显示如下:

```
[INFO] -----
[INFO] Using following parameters for creating project
[INFO] from Old (1.x) Archetype: maven-archetype-quickstart:1.1
[INFO] -----
[INFO] Parameter: groupId, Value: com.companyname.insurance
[INFO] Parameter: packageName, Value: com.companyname.insurance
[INFO] Parameter: package, Value: com.companyname.insurance
[INFO] Parameter: artifactId, Value: health
[INFO] Parameter: basedir, Value: C:\MVN
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: C:\MVN\health
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 4 minutes 12 seconds
[INFO] Finished at: Fri Jul 13 11:10:12 IST 2012
```

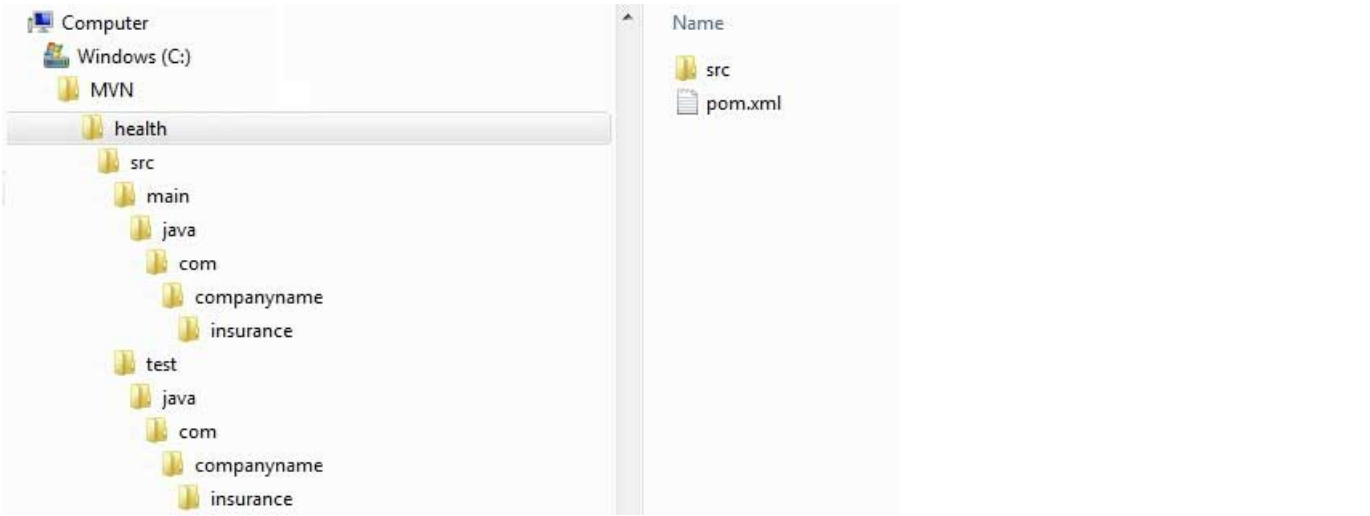
```
[INFO] Final Memory: 20M/90M

[INFO] -----
```

[INFO] -----

创建的项目

现在转到 `C:\>MVN` 目录。你会看到一个名为 `health` 的 `java` 应用程序项目，就像在项目创建的时候建立的 `artifactId` 名称一样。`Maven` 将创建一个有标准目录布局的项目，如下所示：



创建 pom.xml

Maven 为项目自动生成一个 `pom.xml` 文件，如下所示：

```
<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>
<groupId>com.companyname.insurance</groupId>
<artifactId>health</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
<name>health</name>
<url>http://maven.apache.org</url>
<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>
</project>
```

App.java

Maven 会自动生成一个测试的 java 文件 App.java。

路径: C:\MVN\consumerBanking\src\main\java\com\companyname\bank

```
package com.companyname.insurance;
/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
```

```
System.out.println( "Hello World!" );
}
}
```

AppTest.java

Maven 会自动生成一个 java 文件 AppTest.java。

路径为: **C:\MVM\consumerBanking\src\test\java\com\companyname\bank**

```
package com.companyname.insurance;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
/**
 * Unit test for simple App.
 */
public class AppTest
extends TestCase
{
    /**
     * Create the test case
     *
     * @param testName name of the test case
     */
    public AppTest( String testName )
    {
        super( testName );
    }
    /**
     * @return the suite of tests being tested
     */
    public static Test suite()
    {
        return new TestSuite( AppTest.class );
    }
    /**
     * Rigourous Test :-))
     */
    public void testApp()
    {
        assertTrue( true );
    }
}
```

就这样。现在你可以看到 **Maven** 的强大之处。你可以用 **maven** 简单的命令创建任何类型的项目，并且可以启动您的开发。

☐ Maven 引入外部依赖

Maven 项目文档 ☐

☐ 点我分享笔记

反馈/建议

Copyright © 2013-2018 菜鸟教程 runoob.com All Rights Reserved. 备案号: 闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

☐ Maven 项目模板

Maven 快照(SNAPSHOT) ☐

Maven 项目文档

本章节我们主要学习如何创建 **Maven** 项目文档。

比如我们在 **C:/MVN** 目录下，创建了 **consumerBanking** 项目，**Maven** 使用下面的命令来快速创建 **java** 项目：

```
mvn archetype:generate -DgroupId=com.companyname.bank -DartifactId=consumerBanking -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

修改 **pom.xml**，添加以下配置（如果没有的话）：

```
<project>
...
<build>
<pluginManagement>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-site-plugin</artifactId>
<version>3.3</version>
</plugin>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-project-info-reports-plugin</artifactId>
<version>2.7</version>
</plugin>
</plugins>
</pluginManagement>
</build>
...
</project>
```

不然运行 **mvn site** 命令时出现 **java.lang.NoClassDefFoundError: org/apache/maven/doxia/siterenderer/DocumentContent** 的问题，这是由于 **maven-site-plugin** 版本过低，升级到 3.3+ 即可。

打开 **consumerBanking** 文件夹并执行以下 **mvn** 命令。

```
C:\MVN\consumerBanking> mvn site
```

Maven 开始生成文档：

```
[INFO] Scanning for projects...

[INFO] -----

[INFO] Building consumerBanking

[INFO]task-segment: [site]

[INFO] -----

[INFO] [site:site {execution: default-site}]

[INFO] artifact org.apache.maven.skins:maven-default-skin:
checking for updates from central

[INFO] Generating "About" report.

[INFO] Generating "Issue Tracking" report.

[INFO] Generating "Project Team" report.

[INFO] Generating "Dependencies" report.
```

```
[INFO] Generating "Continuous Integration" report.

[INFO] Generating "Source Repository" report.

[INFO] Generating "Project License" report.

[INFO] Generating "Mailing Lists" report.

[INFO] Generating "Plugin Management" report.

[INFO] Generating "Project Summary" report.

[INFO] -----

[INFO] BUILD SUCCESSFUL

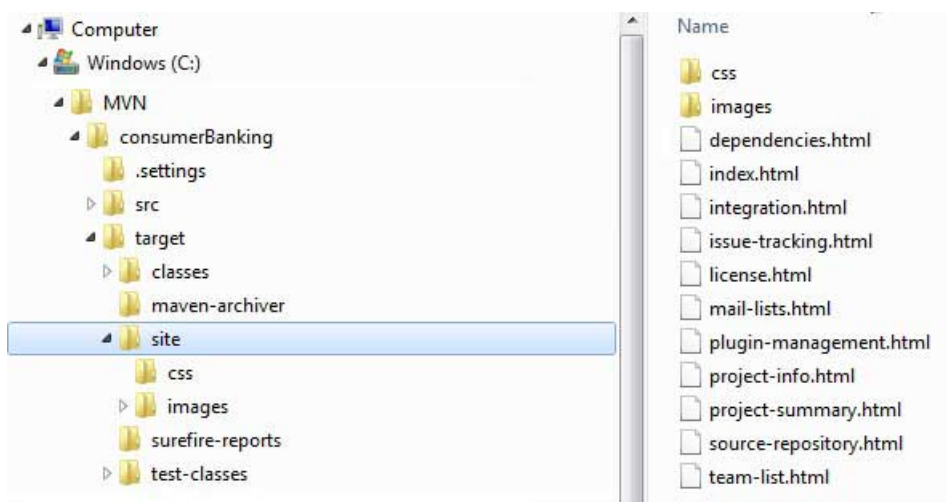
[INFO] -----

[INFO] Total time: 16 seconds

[INFO] Finished at: Wed Jul 11 18:11:18 IST 2012

[INFO] Final Memory: 23M/148M

[INFO] -----
```



打开 **C:\MVN\consumerBanking\target\site** 文件夹。点击 **index.html** 就可以看到文档了。

consumerBanking

Last Published: 2012-07-11

consumerBanking

Project Documentation

Project Information

About

Continuous Integration

Dependencies

Issue Tracking

Mailing Lists

Plugin Management

Project License

Project Summary

Project Team

Source Repository

Built by

maven

Project Summary

Project Information

Field	Value
Name	consumerBanking
Description	-
Homepage	http://maven.apache.org

Project Organization

This project does not belong to an organization.

Build Information

Field	Value
GroupId	com.companyname.bank
ArtifactId	consumerBanking
Version	1.0-SNAPSHOT
Type	jar

© 2012

Maven 使用一个名为 [Doxia](#) 的文档处理引擎来创建文档，它能将多种格式的源码读取成一种通用的文档模型。要为你的项目撰写文档，你可以将内容写成下面几种常用的，可被 [Doxia](#) 转化的格式。

格式名	描述	参考
Apt	纯文本文档格式	http://maven.apache.org/doxia/references/apt-format.html
Xdoc	Maven 1.x 的一种文档格式	http://jakarta.apache.org/site/jakarta-site2.html
FML	FAQ 文档适用	http://maven.apache.org/doxia/references/fml-format.html
XHTML	可扩展的 HTML 文档	http://en.wikipedia.org/wiki/XHTML

☐ Maven 项目模板

Maven 快照(SNAPSHOT) ☐

☐ 点我分享笔记

反馈/建议



☐ Maven 项目文档

Maven 自动化构建 ☐

Maven 快照(SNAPSHOT)

一个大型的软件应用通常包含多个模块，并且通常的场景是多个团队开发同一应用的不同模块。举个例子，设想一个团队开发应用的前端，项目为 `app-ui(app-ui.jar:1.0)`，而另一个团队开发应用的后台，使用的项目是 `data-service(data-service.jar:1.0)`。

现在可能出现的情况是开发 `data-service` 的团队正在进行快节奏的 `bug` 修复或者项目改进，并且他们几乎每隔一天就要发布库到远程仓库。现在如果 `data-service` 团队每隔一天上传一个新版本，那么将会出现下面的问题：

`data-service` 团队每次发布更新的代码时都要告知 `app-ui` 团队。

`app-ui` 团队需要经常地更新他们 `pom.xml` 文件到最新版本。

为了解决这种情况，快照的概念派上了用场。

什么是快照？

快照是一种特殊的版本，指定了某个当前的开发进度的副本。不同于常规的版本，`Maven` 每次构建都会在远程仓库中检查新的快照。现在 `data-service` 团队会每次发布更新代码的快照到仓库中，比如说 `data-service:1.0-SNAPSHOT` 来替代旧的快照 `jar` 包。

项目快照 vs 版本

对于版本，如果 `Maven` 以前下载过指定的版本文件，比如说 `data-service:1.0`，`Maven` 将不会再从仓库下载新的可用的 `1.0` 文件。若要下载更新的代码，`data-service` 的版本需要升到 `1.1`。

快照的情况下，每次 `app-ui` 团队构建他们的项目时，`Maven` 将自动获取最新的快照(`data-service:1.0-SNAPSHOT`)。

app-ui 项目的 pom.xml 文件

`app-ui` 项目使用的是 `data-service` 项目的 `1.0` 快照。

```
<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>
  <groupId>app-ui</groupId>
  <artifactId>app-ui</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>
  <name>health</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>data-service</groupId>
      <artifactId>data-service</artifactId>
      <version>1.0-SNAPSHOT</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

data-service 项目的 pom.xml 文件

`data-service` 项目为每次小的改动发布 `1.0` 快照。

```
<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>
  <groupId>data-service</groupId>
  <artifactId>data-service</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>health</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
</project>
```

虽然，快照的情况下，`Maven` 在日常工作中会自动获取最新的快照，你也可以在任何 `maven` 命令中使用 `-U` 参数强制 `maven` 现在最新的快照构建。

```
mvn clean package -U
```

让我们打开命令控制台，去到 `C:\> MVN > app-ui` 目录，然后执行下面的 `mvn` 命令。


```
C:\MVN\app-ui>mvn clean package -U
```

Maven 将在下载 **data-service** 最新的快照之后，开始构建项目。

```
[INFO] Scanning for projects...

[INFO] -----

[INFO] Building consumerBanking

[INFO]    task-segment: [clean, package]

[INFO] -----

[INFO] Downloading data-service:1.0-SNAPSHOT

[INFO] 290K downloaded.

[INFO] [clean:clean {execution: default-clean}]

[INFO] Deleting directory C:\MVN\app-ui\target

[INFO] [resources:resources {execution: default-resources}]

[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!

[INFO] skip non existing resourceDirectory C:\MVN\app-ui\src\main\
resources

[INFO] [compiler:compile {execution: default-compile}]

[INFO] Compiling 1 source file to C:\MVN\app-ui\target\classes

[INFO] [resources:testResources {execution: default-testResources}]

[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!

[INFO] skip non existing resourceDirectory C:\MVN\app-ui\src\test\
resources

[INFO] [compiler:testCompile {execution: default-testCompile}]

[INFO] Compiling 1 source file to C:\MVN\app-ui\target\test-classes

[INFO] [surefire:test {execution: default-test}]

[INFO] Surefire report directory: C:\MVN\app-ui\target\
surefire-reports

-----

T E S T S

-----

Running com.companyname.bank.AppTest
```

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.027 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [jar:jar {execution: default-jar}]

[INFO] Building jar: C:\MVN\app-ui\target\
app-ui-1.0-SNAPSHOT.jar

[INFO] -----

[INFO] BUILD SUCCESSFUL

[INFO] -----

[INFO] Total time: 2 seconds

[INFO] Finished at: Tue Jul 10 16:52:18 IST 2012

[INFO] Final Memory: 16M/89M

[INFO] -----

[Maven 项目文档](#)

[Maven 自动化构建](#)

[点我分享笔记](#)

[反馈/建议](#)

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号: 闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Maven 快照\(SNAPSHOT\)](#)

[Maven 依赖管理](#)

自动化构建定义了这样一种场景: 在一个项目成功构建完成后, 其相关的依赖工程即开始构建, 这样可以保证其依赖项目的稳定。

比如一个团队正在开发一个项目 `bus-core-api`, 并且有其他两个项目 `app-web-ui` 和 `app-desktop-ui` 依赖于这个项目。

`app-web-ui` 项目使用的是 `bus-core-api` 项目的 1.0 快照:

```
<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>
<groupId>app-web-ui</groupId>
<artifactId>app-web-ui</artifactId>
<version>1.0</version>
<packaging>jar</packaging>
<dependencies>
<dependency>
<groupId>bus-core-api</groupId>
<artifactId>bus-core-api</artifactId>
<version>1.0-SNAPSHOT</version>
</dependency>
</dependencies>
</project>
```

app-desktop-ui 项目使用的是 bus-core-api 项目的 1.0 快照:

```
<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>
<groupId>app-desktop-ui</groupId>
<artifactId>app-desktop-ui</artifactId>
<version>1.0</version>
<packaging>jar</packaging>
<dependencies>
<dependency>
<groupId>bus-core-api</groupId>
<artifactId>bus-core-api</artifactId>
<version>1.0-SNAPSHOT</version>
</dependency>
</dependencies>
</project>
```

bus-core-api 项目:

```
<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>
<groupId>bus-core-api</groupId>
<artifactId>bus-core-api</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
</project>
```

现在 app-web-ui 和 app-desktop-ui 项目的团队要求不管 bus-core-api 项目何时变化, 他们的构建过程都应当可以启动。

使用快照可以确保最新的 bus-core-api 项目被使用, 但要达到上面的要求, 我们还需要做一些额外的工作。

可以使用两种方式:

在 bus-core-api 项目的 pom 文件中添加一个 post-build 目标操作来启动 app-web-ui 和 app-desktop-ui 项目的构建。

使用持续集成 (CI) 服务器, 比如 Hudson, 来自行管理构建自动化。

使用 Maven

修改 bus-core-api 项目的 pom.xml 文件。

```
<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>
<groupId>bus-core-api</groupId>
<artifactId>bus-core-api</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
<build>
<plugins>
<plugin>
```

```
<artifactId>maven-invoker-plugin</artifactId>
<version>1.6</version>
<configuration>
<debug>true</debug>
<pomIncludes>
<pomInclude>app-web-ui/pom.xml</pomInclude>
<pomInclude>app-desktop-ui/pom.xml</pomInclude>
</pomIncludes>
</configuration>
<executions>
<execution>
<id>build</id>
<goals>
<goal>run</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
<build>
</project>
```

打开命令控制台，切换到 `C:\>MVN>bus-core-api` 目录下，然后执行以下命令。

```
C:\MVN\bus-core-api>mvn clean package -U
```

执行完命令后，**Maven** 将开始构建项目 `bus-core-api`。

```
[INFO] Scanning for projects...

[INFO] -----

[INFO] Building bus-core-api

[INFO]    task-segment: [clean, package]

[INFO] -----

...

[INFO] [jar:jar {execution: default-jar}]

[INFO] Building jar: C:\MVN\bus-core-ui\target\

bus-core-ui-1.0-SNAPSHOT.jar

[INFO] -----

[INFO] BUILD SUCCESSFUL

[INFO] -----
```

`bus-core-api` 构建成功后，**Maven** 将开始构建 `app-web-ui` 项目。

```
[INFO] -----

[INFO] Building app-web-ui

[INFO]    task-segment: [package]

[INFO] -----

...
```

```
[INFO] [jar:jar {execution: default-jar}]

[INFO] Building jar: C:\MVN\app-web-ui\target\
app-web-ui-1.0-SNAPSHOT.jar

[INFO] -----

[INFO] BUILD SUCCESSFUL

[INFO] -----
```

app-web-ui 构建成功后，Maven 将开始构建 app-desktop-ui 项目。

```
[INFO] -----

[INFO] Building app-desktop-ui

[INFO] task-segment: [package]

[INFO] -----

...

[INFO] [jar:jar {execution: default-jar}]

[INFO] Building jar: C:\MVN\app-desktop-ui\target\
app-desktop-ui-1.0-SNAPSHOT.jar

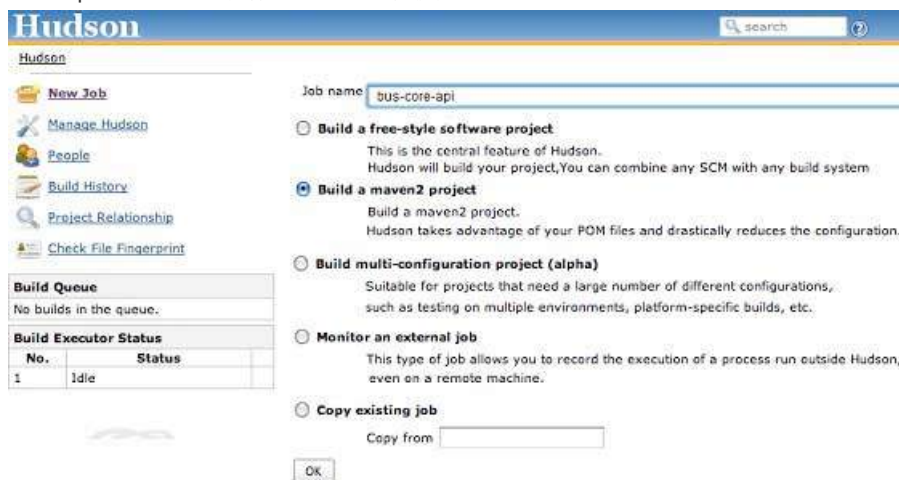
[INFO] -----

[INFO] BUILD SUCCESSFUL

[INFO] -----
```

使用持续集成服务器（CI）

如果使用 CI 服务器更，我们每次的一个新项目，比如说实例中的 **app-mobile-ui**，添加为依赖 **bus-core-api** 项目时，开发者就不需要更新 **bus-core-api** 项目的 **pom**。Hudson 将会借助 Maven 的依赖管理功能实现工程的自动化创建。



Hudson 把每个项目构建当成一次任务。在一个项目的代码提交到 SVN（或者任何映射到 Hudson 的代码管理工具）后，Hudson 将开始项目的构建任务，并且一旦此构建任务完成，Hudson 将自动启动其他依赖的构建任务（其他依赖项目的构建）。

在上面的例子中，当 **bus-core-ui** 源代码在 SVN 更新后，Hudson 开始项目构建。一旦构建成功，Hudson 自动地查找依赖的项目，然后开始构建 **app-web-ui** 和 **app-desktop-ui** 项目。

Maven 依赖管理

Maven 一个核心的特性就是依赖管理。当我们处理多模块的项目（包含成百上千个模块或者子项目），模块间的依赖关系就变得非常复杂，管理也变得很困难。针对此种情形，**Maven** 提供了一种高度控制的方法。

可传递性依赖发现

一种相当常见的情况，比如说 **A** 依赖于其他库 **B**。如果，另外一个项目 **C** 想要使用 **A**，那么 **C** 项目也需要使用库 **B**。

Maven 可以避免去搜索所有所需库的需求。**Maven** 通过读取项目文件（**pom.xml**），找出它们项目之间的依赖关系。

我们需要做的只是在每个项目的 **pom** 中定义好直接的依赖关系。其他的事情 **Maven** 会帮我们搞定。

通过可传递性的依赖，所有被包含的库的图形会快速的的增长。当有重复库时，可能出现的情形将会持续上升。**Maven** 提供一些功能来控制可传递的依赖的程度。

范围	描述
编译阶段	该范围表明相关依赖是只在项目的类路径下有效。默认取值。
供应阶段	该范围表明相关依赖是由运行时的 JDK 或者 网络服务器提供的。
运行阶段	该范围表明相关依赖在编译阶段不是必须的，但是在执行阶段是必须的。
测试阶段	该范围表明相关依赖只在测试编译阶段和执行阶段。
系统阶段	该范围表明你需要提供一个系统路径。
导入阶段	该范围只在依赖是一个 pom 里定义的依赖时使用。同时，当前项目的 POM 文件的 部分定义的依赖关系可以取代某特定的 POM 。

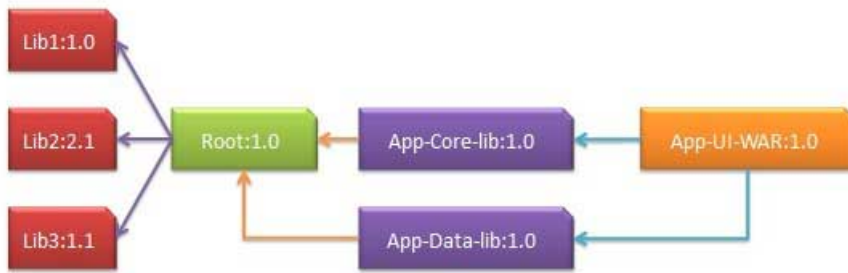
依赖范围

传递依赖发现可以通过使用如下的依赖范围来得到限制：

范围	描述
编译阶段	该范围表明相关依赖是只在项目的类路径下有效。默认取值。
供应阶段	该范围表明相关依赖是由运行时的 JDK 或者 网络服务器提供的。
运行阶段	该范围表明相关依赖在编译阶段不是必须的，但是在执行阶段是必须的。
测试阶段	该范围表明相关依赖只在测试编译阶段和执行阶段。
系统阶段	该范围表明你需要提供一个系统路径。
导入阶段	该范围只在依赖是一个 pom 里定义的依赖时使用。同时，当前项目的 POM 文件的 部分定义的依赖关系可以取代某特定的 POM 。

依赖管理

通常情况下，在一个共通的项目下，有一系列的项目。在这种情况下，我们可以创建一个公共依赖的 pom 文件，该 pom 包含所有的公共的依赖关系，我们称其为其他子项目 pom 的 pom 父。接下来的一个例子可以帮助你更好的理解这个概念。



接下来是上面依赖图的详情说明：

App-UI-WAR 依赖于 App-Core-lib 和 App-Data-lib。

Root 是 App-Core-lib 和 App-Data-lib 的父项目。

Root 在它的依赖部分定义了 Lib1、lib2 和 Lib3 作为依赖。

App-UI-WAR 的 pom.xml 文件代码如下：

```
<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>
  <groupId>com.companyname.groupname</groupId>
  <artifactId>App-UI-WAR</artifactId>
  <version>1.0</version>
  <packaging>war</packaging>
  <dependencies>
  <dependency>
    <groupId>com.companyname.groupname</groupId>
    <artifactId>App-Core-lib</artifactId>
    <version>1.0</version>
  </dependency>
</dependencies>
<dependencies>
<dependency>
  <groupId>com.companyname.groupname</groupId>
  <artifactId>App-Data-lib</artifactId>
  <version>1.0</version>
</dependency>
</dependencies>
</project>
```

App-Core-lib 的 pom.xml 文件代码如下：

```
<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">
  <parent>
    <artifactId>Root</artifactId>
    <groupId>com.companyname.groupname</groupId>
    <version>1.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.companyname.groupname</groupId>
  <artifactId>App-Core-lib</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>
</project>
```

App-Data-lib 的 pom.xml 文件代码如下：

```
<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">
<parent>
<artifactId>Root</artifactId>
<groupId>com.companyname.groupname</groupId>
<version>1.0</version>
</parent>
<modelVersion>4.0.0</modelVersion>
<groupId>com.companyname.groupname</groupId>
<artifactId>App-Data-lib</artifactId>
<version>1.0</version>
<packaging>jar</packaging>
</project>
```

Root 的 pom.xml 文件代码如下：

```
<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>
<groupId>com.companyname.groupname</groupId>
<artifactId>Root</artifactId>
<version>1.0</version>
<packaging>pom</packaging>
<dependencies>
<dependency>
<groupId>com.companyname.groupname1</groupId>
<artifactId>Lib1</artifactId>
<version>1.0</version>
</dependency>
</dependencies>
<dependencies>
<dependency>
<groupId>com.companyname.groupname2</groupId>
<artifactId>Lib2</artifactId>
<version>2.1</version>
</dependency>
</dependencies>
<dependencies>
<dependency>
<groupId>com.companyname.groupname3</groupId>
<artifactId>Lib3</artifactId>
<version>1.1</version>
</dependency>
</dependencies>
</project>
```

现在当我们构建 App-UI-WAR 项目时，Maven 将通过遍历依赖关系图找到所有的依赖关系，并且构建该应用程序。

通过上面的例子，我们可以学习到以下关键概念：

公共的依赖可以使用 pom 父的概念被统一放在一起。App-Data-lib 和 App-Core-lib 项目的依赖在 Root 项目里列举了出来（参考 Root 的包类型，它是一个 POM）。

没有必要在 App-UI-W 里声明 Lib1, lib2, Lib3 是它的依赖。Maven 通过使用可传递的依赖机制来实现该细节。

☐ Maven 自动化构建

Maven 自动化部署 ☐

☐ 点我分享笔记

反馈/建议

Maven 自动化部署

项目开发过程中，部署的过程包含需如下步骤：

将所的项目代码提交到 **SVN** 或者代码库中并打上标签。

从 **SVN** 上下载完整的源代码。

构建应用。

存储构建输出的 **WAR** 或者 **EAR** 文件到一个常用的网络位置下。

从网络上获取文件并且部署文件到生产站点上。

更新文档并且更新应用的版本号。

问题描述

通常情况下上面的提到开发过程中会涉及到多个团队。一个团队可能负责提交代码，另一个团队负责构建等等。很有可能由于涉及的人为操作和多团队环境的原因，任何一个步骤都可能出错。比如，较旧版本没有在网络机器上更新，然后部署团队又重新部署了较早的构建版本。

解决方案

通过结合以下方案来实现自动化部署：

使用 **Maven** 构建和发布项目

使用 **SubVersion**，源码仓库来管理源代码

使用远程仓库管理软件（**Jfrog**或者**Nexus**）来管理项目二进制文件。

修改项目的 pom.xml

我们将会使用 **Maven** 发布的插件来创建一个自动化发布过程。

例如，**bus-core-api** 项目的 **pom.xml** 文件代码如下：

```
<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>
  <groupId>bus-core-api</groupId>
  <artifactId>bus-core-api</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <scm>
    <url>http://www.svn.com</url>
    <connection>scm:svn:http://localhost:8080/svn/jrepo/trunk/
Framework</connection>
    <developerConnection>scm:svn:${username}/${password}@localhost:8080:
common_core_api:1101:code</developerConnection>
  </scm>
  <distributionManagement>
    <repository>
      <id>Core-API-Java-Release</id>
      <name>Release repository</name>
      <url>http://localhost:8081/nexus/content/repositories/
Core-API-Release</url>
    </repository>
  </distributionManagement>
  <build>
```

```
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-release-plugin</artifactId>
<version>2.0-beta-9</version>
<configuration>
<useReleaseProfile>>false</useReleaseProfile>
<goals>deploy</goals>
<scmCommentPrefix>[bus-core-api-release-checkin]-<
/scmCommentPrefix>
</configuration>
</plugin>
</plugins>
</build>
</project>
```

在 pom.xml 文件中，我们常用到的一些重要元素节点如下表所示：

元素节点	描述
SCM	配置 SVN 的路径，Maven 将从该路径下将代码取下来。
repository	构建的 WAR 或 EAR 或 JAR 文件的位置，或者其他源码构建成功后生成的构件的存储位置。
Plugin	配置 maven-release-plugin 插件来实现自动部署过程。

Maven Release 插件

Maven 使用 maven-release-plugin 插件来完成以下任务。

```
mvn release:clean
```

清理工作空间，保证最新的发布进程成功进行。

```
mvn release:rollback
```

在上次发布过程不成功的情况下，回滚修改的工作空间代码和配置保证发布过程成功进行。

```
mvn release:prepare
```

执行多种操作：

- 检查本地是否存在还未提交的修改
- 确保没有快照的依赖
- 改变应用程序的版本信息用以发布
- 更新 POM 文件到 SVN
- 运行测试用例
- 提交修改后的 POM 文件
- 为代码在 SVN 上做标记
- 增加版本号和附加快照以备将来发布
- 提交修改后的 POM 文件到 SVN

```
mvn release:perform
```

将代码切换到之前做标记的地方，运行 Maven 部署目标来部署 WAR 文件或者构建相应的结构到仓库里。

打开命令终端，进入到 C:\>MVN>bus-core-api 目录下，然后执行如下的 mvn 命令。

```
C:\MVN\bus-core-api>mvn release:prepare
```

Maven 开始构建整个工程。构建成功后即可运行如下 mvn 命令。

```
C:\MVN\bus-core-api>mvn release:perform
```

构建成功后，你就可以可以验证在你仓库下上传的 JAR 文件是否生效。

❏ 点我分享笔记

反馈/建议



Maven Web 应用

本章节我们将学习如何使用版本控制系统 Maven 来管理一个基于 web 的项目，如何创建、构建、部署已经运行一个 web 应用。

创建 Web 应用

我们可以使用 maven-archetype-webapp 插件来创建一个简单的 Java web 应用。

打开命令控制台，进入到 C:\MVN 文件夹，然后执行以下的 mvn 命令：

```
C:\MVN>mvn archetype:generate -DgroupId = com.companyname.automobile -DartifactId = trucks -DarchetypeArtifactId = maven-archetype-webapp -DinteractiveMode = false
```

执行完后 Maven 将开始处理，并且创建完整的于 Java Web 项目的目录结构。

```
[INFO] Scanning for projects...

[INFO] Searching repository for plugin with prefix: 'archetype'.

[INFO] -----

[INFO] Building Maven Default Project

[INFO]    task-segment: [archetype:generate] (aggregator-style)

[INFO] -----

[INFO] Preparing archetype:generate

[INFO] No goals needed for project - skipping

[INFO] [archetype:generate {execution: default-cli}]

[INFO] Generating project in Batch mode
```

```
[INFO] -----

[INFO] Using following parameters for creating project

from Old (1.x) Archetype: maven-archetype-webapp:1.0

[INFO] -----

[INFO] Parameter: groupId, Value: com.companyname.automobile

[INFO] Parameter: packageName, Value: com.companyname.automobile

[INFO] Parameter: package, Value: com.companyname.automobile

[INFO] Parameter: artifactId, Value: trucks

[INFO] Parameter: basedir, Value: C:\MVN

[INFO] Parameter: version, Value: 1.0-SNAPSHOT

[INFO] project created from Old (1.x) Archetype in dir: C:\MVN\trucks

[INFO] -----

[INFO] BUILD SUCCESSFUL

[INFO] -----

[INFO] Total time: 16 seconds

[INFO] Finished at: Tue Jul 17 11:00:00 IST 2012

[INFO] Final Memory: 20M/89M

[INFO] -----
```

执行完后，我们可以在 C:\MVN 文件夹下看到 trucks 项目，查看项目的目录结构：



Maven 目录结构是标准的，各个目录作用如下表所示：

文件夹结构	描述
trucks	包含 src 文件夹和 pom.xml 文件。
src/main/webapp	包含 index.jsp 文件和 WEB-INF 文件夹。
src/main/webapp/WEB-INF	包含 web.xml 文件
src/main/resources	包含图片、properties资源文件。

pom.xml 文件代码如下：

```
<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.companyname.automobile</groupId>
<artifactId>trucks</artifactId>
<packaging>war</packaging>
<version>1.0-SNAPSHOT</version>
<name>trucks 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>trucks</finalName>
</build>
</project>
```

接下来我们打开 C:\> MVN > trucks > src > main > webapp > 文件夹，可以看到一个已经创建好的 index.jsp 文件，代码如下：

```
<html>
<body>
<h2>Hello World!</h2>
</body>
</html>
```

构建 Web 应用

打开命令控制台，进入 C:\MVN\trucks 目录，然后执行下面的以下 mvn 命令：

```
C:\MVN\trucks>mvn clean package
```

Maven 将开始构建项目：

```
[INFO] Scanning for projects...

[INFO] -----

[INFO] Building trucks Maven Webapp

[INFO]    task-segment: [clean, package]

[INFO] -----

[INFO] [clean:clean {execution: default-clean}]

[INFO] [resources:resources {execution: default-resources}]

[WARNING] Using platform encoding (Cp1252 actually) to
copy filtered resources,i.e. build is platform dependent!

[INFO] Copying 0 resource

[INFO] [compiler:compile {execution: default-compile}]

[INFO] No sources to compile

[INFO] [resources:testResources {execution: default-testResources}]

[WARNING] Using platform encoding (Cp1252 actually) to
copy filtered resources,i.e. build is platform dependent!
```

```
[INFO] skip non existing resourceDirectory
C:\MVN\trucks\src\test\resources

[INFO] [compiler:testCompile {execution: default-testCompile}]

[INFO] No sources to compile

[INFO] [surefire:test {execution: default-test}]

[INFO] No tests to run.

[INFO] [war:war {execution: default-war}]

[INFO] Packaging webapp

[INFO] Assembling webapp[trucks] in [C:\MVN\trucks\target\trucks]

[INFO] Processing war project

[INFO] Copying webapp resources[C:\MVN\trucks\src\main\webapp]

[INFO] Webapp assembled in[77 msec]

[INFO] Building war: C:\MVN\trucks\target\trucks.war

[INFO] -----

[INFO] BUILD SUCCESSFUL

[INFO] -----

[INFO] Total time: 3 seconds

[INFO] Finished at: Tue Jul 17 11:22:45 IST 2012

[INFO] Final Memory: 11M/85M

[INFO] -----
```

部署 Web 应用

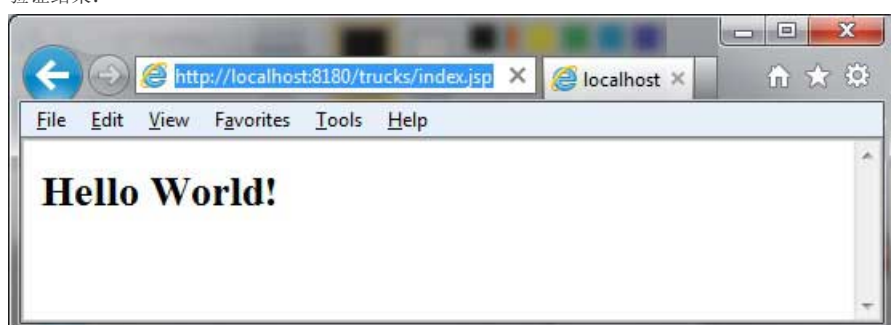
打开 C:\< MVN < trucks < target < 文件夹，找到 trucks.war 文件，并复制到你的 web 服务器的 web 应用目录，然后重启 web 服务器。

测试 Web 应用

访问以下 URL 运行 web 应用：

```
http://localhost:8180/trucks/index.jsp
```

验证结果：



□ 点我分享笔记

反馈/建议

Copyright © 2013-2018 菜鸟教程 runoob.com All Rights Reserved. 备案号: 闽ICP备15012807号-1

首页 HTML CSS JS 本地书签

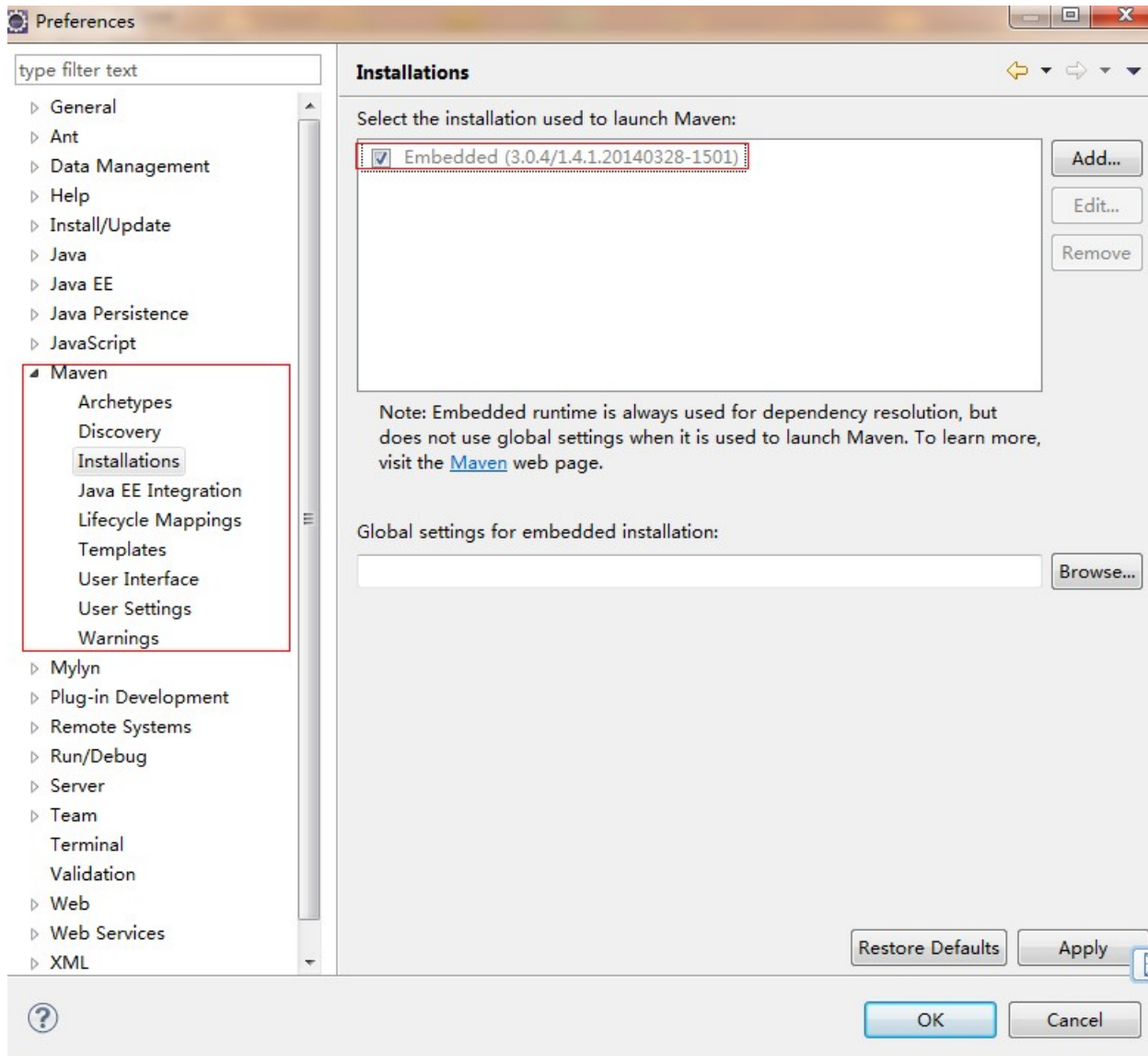
□ Maven Web 应用

Maven NetBeans □

Maven Eclipse

Eclipse 提供了一个很好的插件 [m2eclipse](#)，该插件能将 Maven 和 Eclipse 集成在一起。

在最新的 Eclipse 中自带了 Maven，我们打开，Windows->Preferences，如果出现下面的画面：



下面列出 m2eclipse 的一些特点：

可以在 Eclipse 环境中运行 Maven 的目标文件。

可以使用其自带的控制台在 **Eclipse** 中直接查看 **Maven** 命令的输出。

可以在 **IDE** 下更新 **Maven** 的依赖关系。

可以使用 **Eclipse** 开展 **Maven** 项目的构建。

Eclipse 基于 **Maven** 的 **pom.xml** 来实现自动化管理依赖关系。

它解决了 **Maven** 与 **Eclipse** 的工作空间之间的依赖，而不需要安装到本地 **Maven** 的存储库（需要依赖项目在同一个工作区）。

它可以自动地从远端的 **Maven** 库中下载所需要的依赖以及源码。

它提供了向导，为建立新 **Maven** 项目，**pom.xml** 以及在已有的项目上开启 **Maven** 支持。

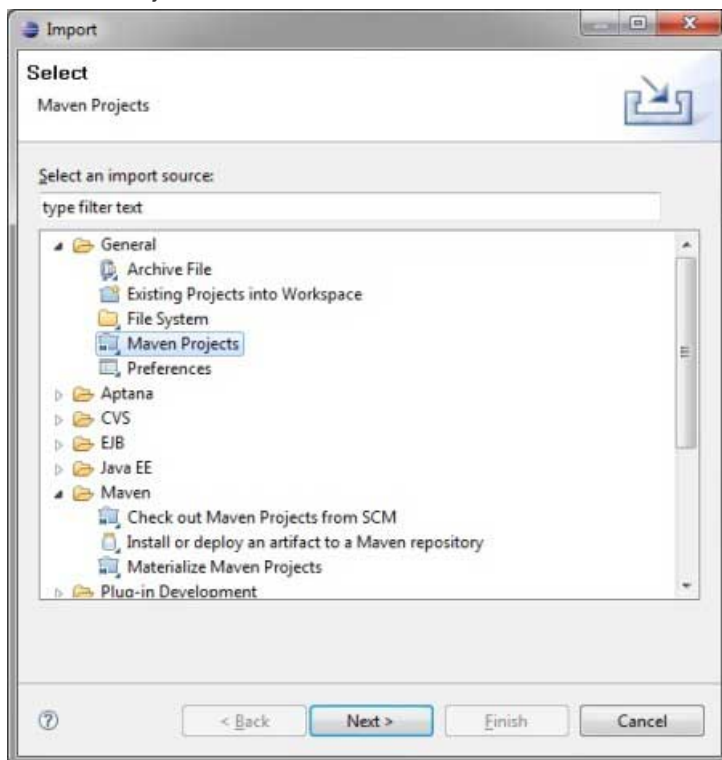
它提供了远端的 **Maven** 存储库的依赖的快速搜索。

在 **Eclipse** 中导入一个 **Maven** 的项目

打开 **Eclipse**

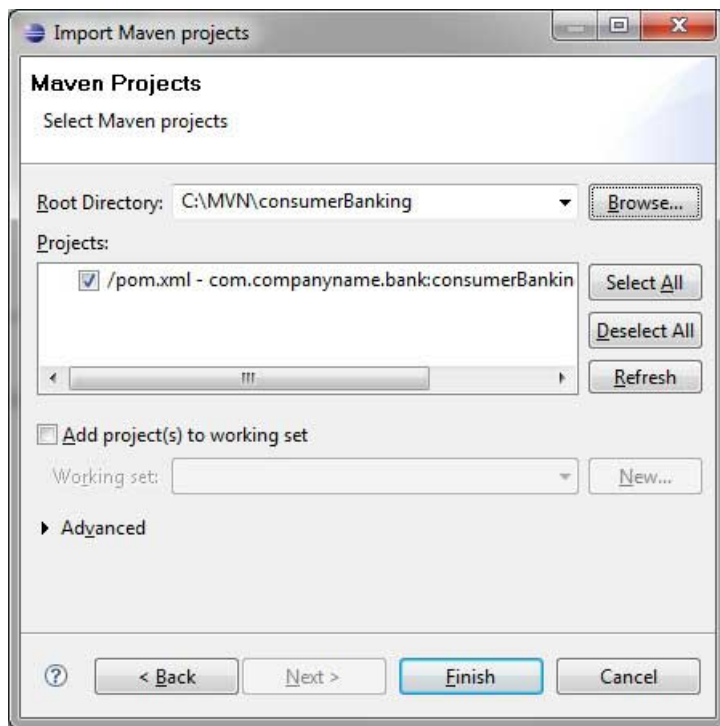
选择 **File > Import > option**

选择 **Maven Projects** 选项。点击 **Next** 按钮。

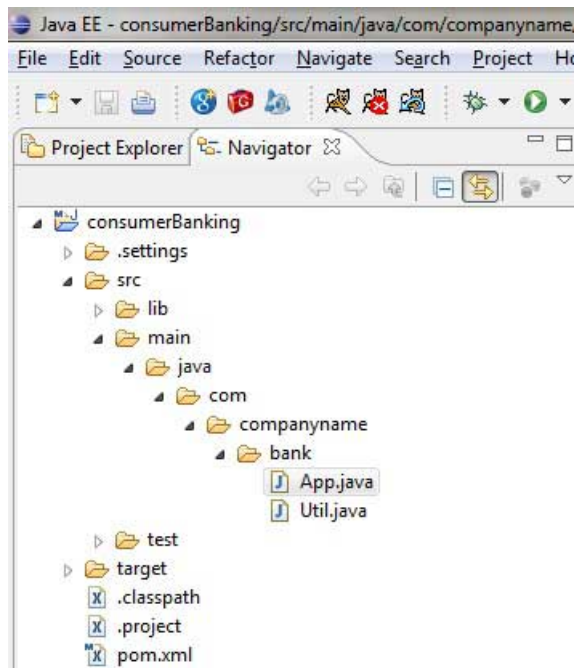


选择项目的路径，即使用 **Maven** 创建一个项目时的存储路径。假设我们创建了一个项目： **consumerBanking**. 通过 **Maven 构建 Java 项目** 查看如何使用 **Maven** 创建一个项目。

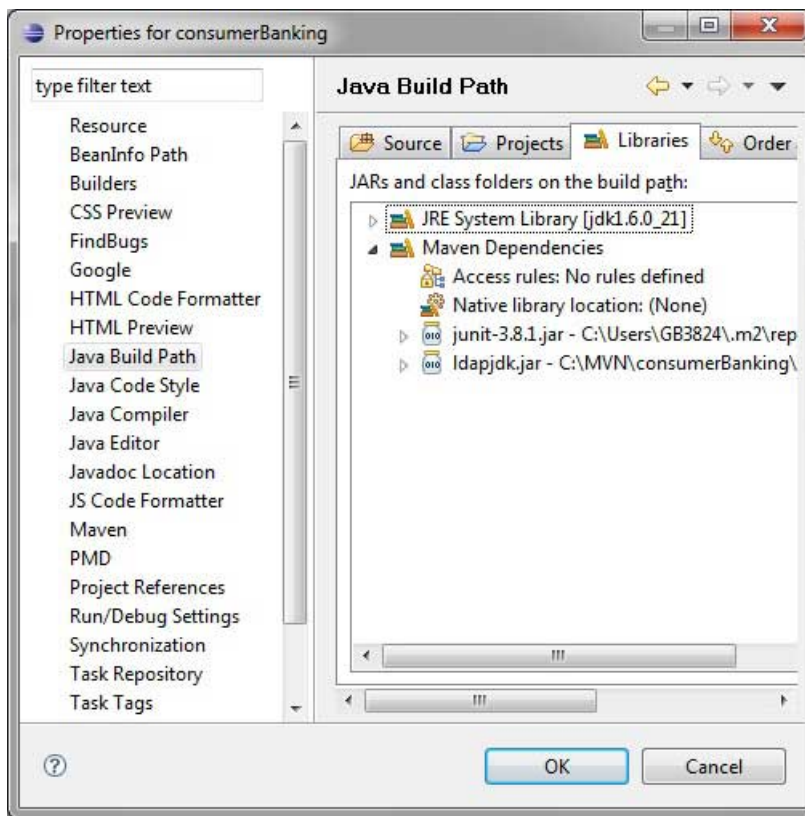
点击 **Finish** 按钮。



现在，你可以在 Eclipse 中看到 Maven 项目。



看一下 consumerBanking 项目的属性，你可以发现 Eclipse 已经将 Maven 所依赖的都添加到了它的构建路径里了。



好了，我们来使用 **Eclipse** 的编译功能来构建这个 **Maven** 项目。

右键打开 **consumerBanking** 项目的上下文菜单

选择 **Run** 选项

然后选择 **maven package** 选项

Maven 开始构建项目，你可以在 **Eclipse** 的控制台看到输出日志。

```
[INFO] Scanning for projects...

[INFO] -----

[INFO] Building consumerBanking

[INFO]

[INFO] Id: com.companyname.bank:consumerBanking:jar:1.0-SNAPSHOT

[INFO] task-segment: [package]

[INFO] -----

[INFO] [resources:resources]

[INFO] Using default encoding to copy filtered resources.

[INFO] [compiler:compile]

[INFO] Nothing to compile - all classes are up to date

[INFO] [resources:testResources]

[INFO] Using default encoding to copy filtered resources.

[INFO] [compiler:testCompile]

[INFO] Nothing to compile - all classes are up to date
```

[INFO] [surefire:test]

[INFO] Surefire report directory:

C:\MVN\consumerBanking\target\surefire-reports

T E S T S

Running com.companyname.bank.AppTest

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.047 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [jar:jar]

[INFO] -----

[INFO] BUILD SUCCESSFUL

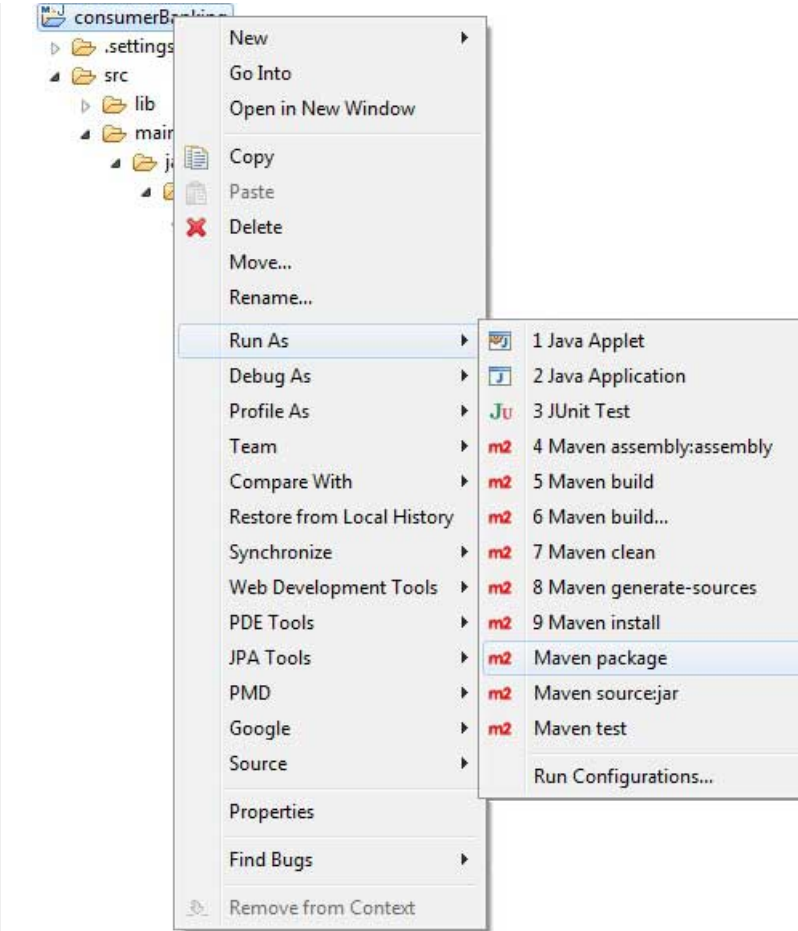
[INFO] -----

[INFO] Total time: 1 second

[INFO] Finished at: Thu Jul 12 18:18:24 IST 2012

[INFO] Final Memory: 2M/15M

[INFO] -----



现在，右键点击 **App.java**，选择 **Run As** 选项。选择 **As Java App**
你将看到如下结果：

```
Hello World!
```

☐ Maven Web 应用

Maven NetBeans ☐

☐ 点我分享笔记

反馈/建议

Maven NetBeans

NetBeans 6.7 及更新的版本已经内置了 Maven。对于以前的版本，可在插件管理中心获取 Maven 插件。此例中我们使用的是 NetBeans 6.9。关于 NetBeans 的一些特性如下：

可以通过 NetBeans 来运行 Maven 目标。

可以使用 NetBeans 自身的控制台查看 Maven 命令的输出。

可以更新 Maven 与 IDE 的依赖。

可以在 NetBeans 中启动 Maven 的构建。

NetBeans 基于 Maven 的 pom.xml 来实现自动化管理依赖关系。

NetBeans 可以通过自己的工作区解决 Maven 的依赖问题，而无需安装到本地的 Maven 仓库，虽然需要依赖的项目在同一个工作区。

NetBeans 可以自动从远程 Maven 库上下载需要的依赖和源码。

NetBeans 提供了创建 Maven 项目，pom.xml 文件的向导。

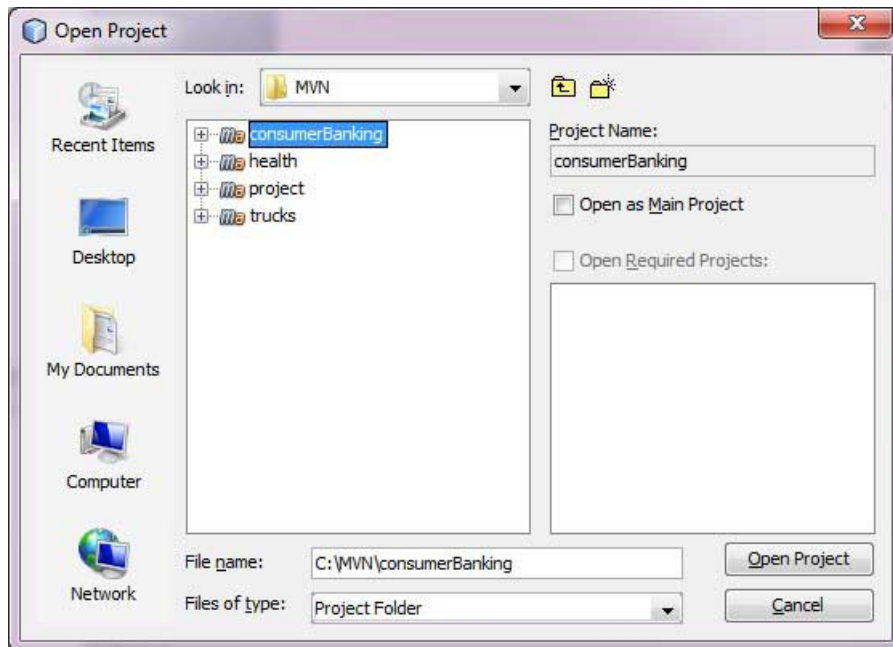
NetBeans 提供了关于 Maven 仓库的浏览器，使您可以查看本地存储库和注册在外部的 Maven 仓库。

在 NetBeans 里打开一个 Maven 项目

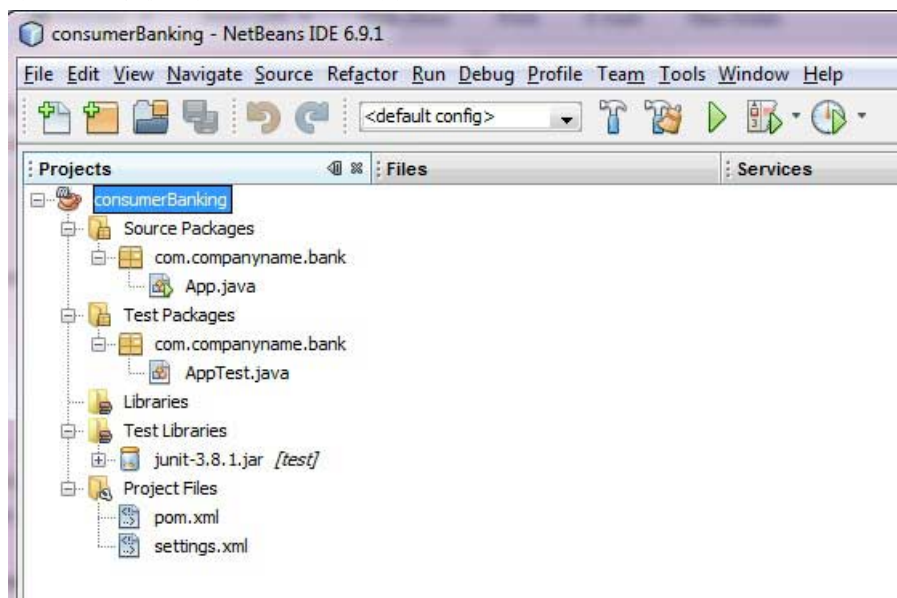
打开 NetBeans

选择 **File Menu > Open Project** 选项

选择项目的路径，即使用 Maven 创建一个项目时的存储路径。假设我们创建了一个项目：consumerBanking. 通过 [Maven 构建 Java 项目](#) 查看如何使用 Maven 创建一个项目。



目前为止，你已经可以在 NetBeans 里看到 Maven 项目了。看一下 consumerBanking 项目的 Libraries 和 Test Libraries. 你可以发现 NetBeans 已经将 Maven 所依赖的都添加到了它的构建路径里了。



在 **NetBeans** 里构建一个 **Maven** 项目

好了，我们来使用 **NetBeans** 的编译功能来构建这个 **Maven** 项目

右键点击 **consumerBanking** 项目打开上下文菜单。

选择 "**Clean and Build**" 选项

□

Maven 将会开始构建该项目。你可以在 **NetBeans** 的终端里查看输出的日志信息：

```
NetBeans: Executing 'mvn.bat -Dnetbeans.execution=true clean install'

NetBeans:      JAVA_HOME=C:\Program Files\Java\jdk1.6.0_21

Scanning for projects...

-----

Building consumerBanking

    task-segment: [clean, install]

-----

[clean:clean]

[resources:resources]

[WARNING] Using platform encoding (Cp1252 actually)
to copy filtered resources, i.e. build is platform dependent!

skip non existing resourceDirectory C:\MVN\consumerBanking\src\main\resources

[compiler:compile]

Compiling 2 source files to C:\MVN\consumerBanking\target\classes

[resources:testResources]

[WARNING] Using platform encoding (Cp1252 actually)
to copy filtered resources, i.e. build is platform dependent!

skip non existing resourceDirectory C:\MVN\consumerBanking\src\test\resources

[compiler:testCompile]

Compiling 1 source file to C:\MVN\consumerBanking\target\test-classes

[surefire:test]

Surefire report directory: C:\MVN\consumerBanking\target\surefire-reports

-----

T E S T S

-----

Running com.companyname.bank.AppTest
```

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.023 sec
```

```
Results :
```

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

```
[jar:jar]
```

```
Building jar: C:\MVN\consumerBanking\target\consumerBanking-1.0-SNAPSHOT.jar
```

```
[install:install]
```

```
Installing C:\MVN\consumerBanking\target\consumerBanking-1.0-SNAPSHOT.jar
```

```
to C:\Users\GB3824\.m2\repository\com\companyname\bank\consumerBanking\
1.0-SNAPSHOT\consumerBanking-1.0-SNAPSHOT.jar
```

```
-----
BUILD SUCCESSFUL
```

```
-----
Total time: 9 seconds
```

```
Finished at: Thu Jul 19 12:57:28 IST 2012
```

```
Final Memory: 16M/85M
-----
```

在 NetBeans 里运行应用程序

现在，右键点击 **App.java** 文件。选择 **Run File** 选项。你可以在终端看到如下结果：

```
NetBeans: Executing 'mvn.bat -Dexec.classpathScope=runtime

-Dexec.args=-classpath %classpath com.companyname.bank.App

-Dexec.executable=C:\Program Files\Java\jdk1.6.0_21\bin\java.exe

-Dnetbeans.execution=true process-classes

org.codehaus.mojo:exec-maven-plugin:1.1.1:exec'

NetBeans:      JAVA_HOME=C:\Program Files\Java\jdk1.6.0_21

Scanning for projects...

-----

Building consumerBanking

    task-segment: [process-classes,

    org.codehaus.mojo:exec-maven-plugin:1.1.1:exec]
```

[resources:resources]

[WARNING] Using platform encoding (Cp1252 actually)

to copy filtered resources, i.e. build is platform dependent!

skip non existing resourceDirectory C:\MVN\consumerBanking\src\main\resources

[compiler:compile]

Nothing to compile - all classes are up to date

[exec:exec]

Hello World!

BUILD SUCCESSFUL

Total time: 1 second

Finished at: Thu Jul 19 14:18:13 IST 2012

Final Memory: 7M/64M

☐ Maven Eclipse

Maven IntelliJ ☐

☐ 点我分享笔记



反馈/建议

Copyright © 2013-2018 菜鸟教程 runoob.com All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

☐ Maven NetBeans

Maven IntelliJ

IntelliJ IDEA 已经内建了对 **Maven** 的支持。我们在此例中使用的是 **IntelliJ IDEA** 社区版 11.1。

IntelliJ IDEA 的一些特性列出如下：

- 可以通过 **IntelliJ IDEA** 来运行 **Maven** 目标。

- 可以在 **IntelliJ IDEA** 自己的终端里查看 **Maven** 命令的输出结果。

可以在 IDE 里更新 Maven 的依赖关系。

可以在 IntelliJ IDEA 中启动 Maven 的构建。

IntelliJ IDEA 基于 Maven 的 pom.xml 来实现自动化管理依赖关系。

IntelliJ IDEA 可以通过自己的工作区解决 Maven 的依赖问题，而无需安装到本地的 Maven 仓库，虽然需要依赖的项目在同一个工作区。

IntelliJ IDEA 可以自动从远程 Maven 仓库上下载需要的依赖和源码。

IntelliJ IDEA 提供了创建 Maven 项目，pom.xml 文件的向导。

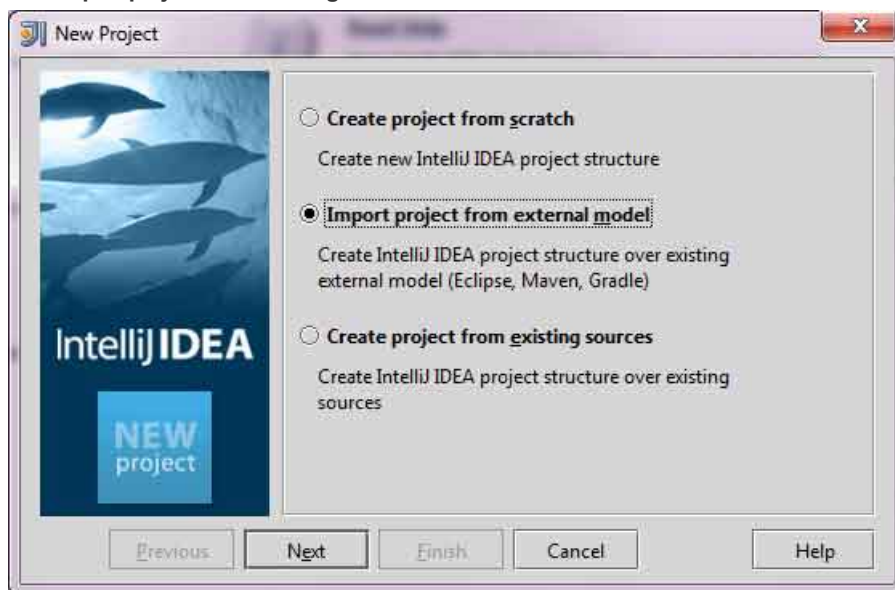
在 IntelliJ IDEA 里创建一个新的项目

使用新建项目向导来导入一个 Maven 项目。

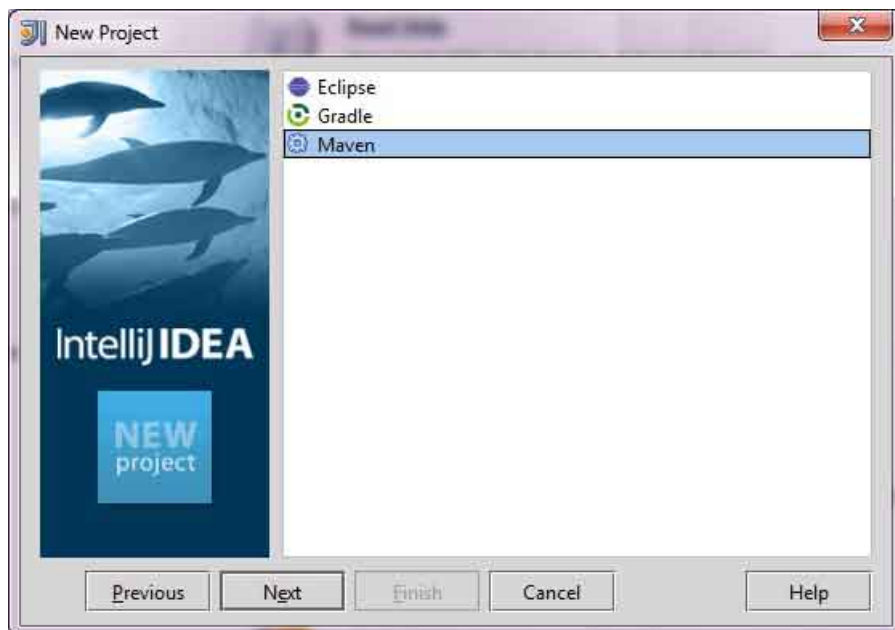
打开 IntelliJ IDEA。

选择 **File Menu > New Project** 选项。

选择 **import project from existing model** 选项。



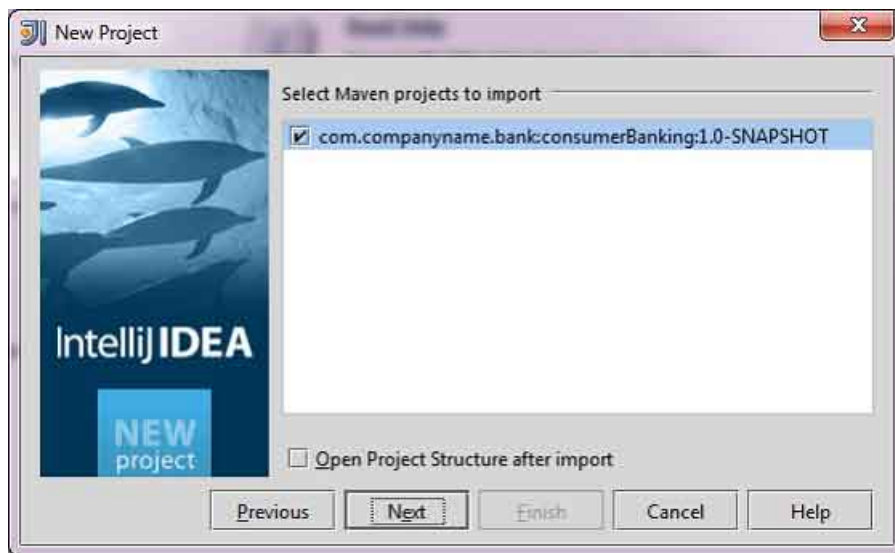
选择 **Maven** 选项。



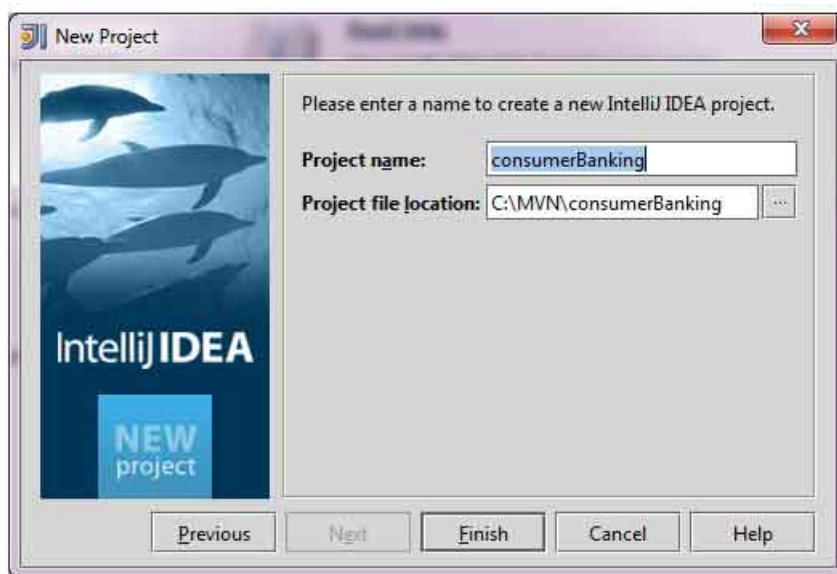
选择项目路径，即使用 Maven 创建一个项目时的存储路径。假设我们创建了一个项目 **consumerBanking**。通过 [Maven 构建 Java 项目](#) 查看如何使用 Maven 创建一个项目。

□

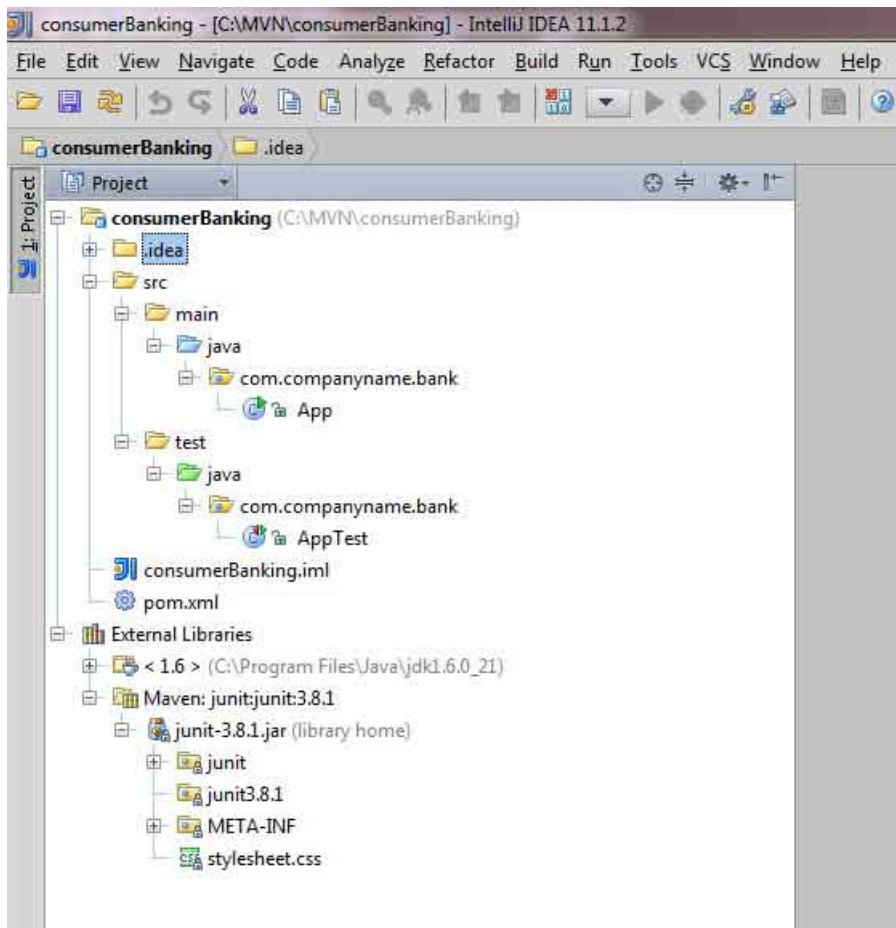
选择要导入的 Maven 项目。



输入项目名称，点击 "finish"。



现在，我们可以在 IntelliJ IDEA 里看到 Maven 项目了。看一下 consumerBanking 项目的 Libraries 和 Test Libraries，你可以发现 IntelliJ IDEA 已经将 Maven 所依赖的都添加到了它的构建路径里了。



在 IntelliJ IDEA 里构建一个 Maven 项目

好了，接下来我们来使用 IntelliJ IDEA 的编译功能来构建这个 Maven 项目。

选中 consumerBanking 项目。

选择 **Buid menu > Rebuild Project** 选项。

你可以在 IntelliJ IDEA 的终端里看到构建过程输出的log:

```
4:01:56 PM Compilation completed successfully
```

在 IntelliJ IDEA 里运行应用程序

选中 consumerBanking 项目。

右键点击 App.java 弹出上下文菜单。

选择 **Run App.main()**。

□

你将会在 IntelliJ IDEA 的终端下看到如下运行结果:

```
"C:\Program Files\Java\jdk1.6.0_21\bin\java"
-Didea.launcher.port=7533
"-Didea.launcher.bin.path=
C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 11.1.2\bin"
-Dfile.encoding=UTF-8
-classpath "C:\Program Files\Java\jdk1.6.0_21\jre\lib\charsets.jar;
```

```
C:\Program Files\Java\jdk1.6.0_21\jre\lib\deploy.jar;

C:\Program Files\Java\jdk1.6.0_21\jre\lib\javaws.jar;

C:\Program Files\Java\jdk1.6.0_21\jre\lib\jce.jar;

C:\Program Files\Java\jdk1.6.0_21\jre\lib\jsse.jar;

C:\Program Files\Java\jdk1.6.0_21\jre\lib\management-agent.jar;

C:\Program Files\Java\jdk1.6.0_21\jre\lib\plugin.jar;

C:\Program Files\Java\jdk1.6.0_21\jre\lib\resources.jar;

C:\Program Files\Java\jdk1.6.0_21\jre\lib\rt.jar;

C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\dnsns.jar;

C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\localedata.jar;

C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\sunjce_provider.jar;

C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\sunmscapi.jar;

C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\sunpkcs11.jar

C:\MVN\consumerBanking\target\classes;

C:\Program Files\JetBrains\

IntelliJ IDEA Community Edition 11.1.2\lib\idea_rt.jar"

com.intellij.rt.execution.application.AppMain com.companyname.bank.App

Hello World!

Process finished with exit code 0
```

☐ Maven NetBeans

☐ 点我分享笔记

反馈/建议