# 从Maven远程存储库下载？

告诉 Maven 来获得 Java.net 的依赖，你需要声明远程仓库在 pom.xml 文件这样：

pom.xml

<repositories>

<repository>

<id>java.net</id>

<url>https://maven.java.net/content/repositories/public/</url>

</repository>

<!—添加JBoss远程仓库的详细信息在“pom.xml”文件中。

<repository>

<id>JBoss repository</id>

<url>http://repository.jboss.org/nexus/content/groups/public/</url>

</repository>

</repositories>

现在，Maven的依赖库查询顺序更改为：

1. 在 Maven 本地资源库中搜索，如果没有找到，进入第 2 步，否则退出。
2. 在 Maven 中央存储库搜索，如果没有找到，进入第 3 步，否则退出。
3. 在java.net Maven的远程存储库搜索，如果没有找到，提示错误信息，否则退出。

# Maven依赖机制

1. 你需要知道 log4j 的 Maven 坐标，例如：

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.14</version>

1. 它会自动下载 log4j 的1.2.14 版本库。如果“version”标签被忽略，它会自动升级库时当有新的版本时。
2. 声明 Maven 的坐标转换成 pom.xml 文件。

<dependencies>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.14</version>

</dependency>

</dependencies>

1. 当 Maven 编译或构建，log4j 的 jar 会自动下载，并把它放到 Maven 本地存储库
2. 所有由 Maven 管理

## 解释说明

看看有什么不同？那么到底在Maven发生了什么？当建立一个Maven的项目，pom.xml文件将被解析，如果看到 log4j 的 Maven 坐标，然后 Maven 按此顺序搜索 log4j 库：

1. 在 Maven 的本地仓库搜索 log4j
2. 在 Maven 中央存储库搜索 log4j
3. 在 Maven 远程仓库搜索 log4j(如果在 pom.xml 中定义)

Maven 依赖库管理是一个非常好的工具，为您节省了大量的工作。

如何找到 Maven 坐标？  
访问 Maven 中心储存库，搜索下载您想要的jar。

# 定制库到Maven本地资源库

例如，kaptcha，它是一个流行的第三方Java库，它被用来生成 “验证码” 的图片，以阻止垃圾邮件，但它不在 Maven 的中央仓库中。

在本教程中，我们将告诉你如何安装 “kaptcha” jar 到Maven 的本地资源库。

## 1. mvn 安装

下载 “kaptcha”，将其解压缩并将 kaptcha-version.jar 复制到其他地方，比如：C盘。发出下面的命令：

mvn install:install-file -Dfile=c:\kaptcha-{version}.jar -DgroupId=com.google.code -DartifactId=kaptcha -Dversion={version} -Dpackaging=jar

现在，“kaptcha” jar被复制到 Maven 本地存储库。

## 2. pom.xml

安装完毕后，就在 pom.xml 中声明 kaptcha 的坐标。

<dependency>

<groupId>com.google.code</groupId>

<artifactId>kaptcha</artifactId>

<version>2.3</version>

</dependency>

## 3. 完成

构建它，现在 “kaptcha” jar 能够从你的 Maven 本地存储库检索了。

# 使用Maven创建Java项目

## 1. 从 Maven 模板创建一个项目

在终端（\* UNIX或Mac）或命令提示符（Windows）中，浏览到要创建 Java 项目的文件夹。键入以下命令：

mvn archetype:generate -DgroupId={project-packaging} -DartifactId={project-name}-DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

这告诉 Maven 来从 maven-archetype-quickstart 模板创建 Java 项目。如果忽视 archetypeArtifactId 选项，一个巨大的 Maven 模板列表将列出。

例如，这里的工作目录是：C:\worksp，执行命令过程时间可能比较久，看个人的网络状况。

C:\worksp>mvn archetype:generate -DgroupId=com.yiibai -DartifactId=NumberGenerate or -DarchetypeArtifactId=maven -archetype-quickstart -DinteractiveMode=false

在上述情况下，一个新的Java项目命名 “NumberGenerator”, 而整个项目的目录结构会自动创建。

**注意**  
有少数用户说 mvn archetype:generate 命令未能生成项目结构。 如果您有任何类似的问题，不用担心，只需跳过此步骤，手动创建文件夹，请参阅步骤2的项目结构。

## 2.Maven目录布局

使用 mvn archetype:generate + maven-archetype-quickstart 模板, 以下项目的目录结构被创建。

NumberGenerator

|-src

|---main

|-----java

|-------com

|---------yiibai

|-----------App.java

|---test

|-----java

|-------com

|---------yiibai

|-----------AppTest.java

|-pom.xml

很简单的，所有的源代码放在文件夹 /src/main/java/, 所有的单元测试代码放入 /src/test/java/.

**注意，请阅读** [Maven标准目录布局](http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html)

附加的一个标准的 pom.xml 被生成。这个POM文件类似于 Ant build.xml 文件，它描述了整个项目的信息，一切从目录结构，项目的插件，项目依赖，如何构建这个项目等，请阅读[POM官方指南](http://maven.apache.org/guides/introduction/introduction-to-the-pom.html)

## 3. Eclipse IDE

为了使它成为一个 Eclipse 项目，在终端进入到 “NumberGenerator” 项目，键入以下命令：

C:\worksp\NumberGenerator>mvn eclipse:eclipse

执行以上命令后，它自动下载更新相关资源和配置信息（需要等待一段时间），并产生 Eclipse IDE所要求的所有项目文件。要导入项目到Eclipse IDE中，选择 “File -> Import… -> General->Existing Projects into Workspace”

图片: 项目导入到 Eclipse IDE中。

## 4. 更新POM

默认的 pom.xml 太简单了，很多时候，你需要添加编译器插件来告诉 Maven 使用哪个 JDK 版本是用来编译项目。（默认JDK1.4，这的确太旧了点）

<plugin>

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

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

<version>2.3.2</version>

<configuration>

<source>1.6</source>

<target>1.6</target>

</configuration>

</plugin>

从更新JUnit 3.8.1到最新的 4.11。

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

<scope>test</scope>

</dependency>

pom.xml – 更新版本

<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.mkyong</groupId>

<artifactId>NumberGenerator</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

<name>NumberGenerator</name>

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

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

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

<version>2.3.2</version>

<configuration>

<source>1.6</source>

<target>1.6</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

在终端，再次发出同样的命令 mvn eclipse:eclipse ,Maven将从Maven中心储存库下载插件项目依赖关系（JUnit），它会自动保存到你的本地仓库。

## 5. 更新业务逻辑

测试驱动开发（TDD），先更新单元测试，以确保应用程序（APP）对象有一个方法来生成包含恰好36位字母表的唯一密钥。

AppTest.java

package com.yiibai;

import org.junit.Assert;

import org.junit.Test;

public class AppTest {

@Test

public void testLengthOfTheUniqueKey() {

App obj = new App();

Assert.assertEquals(36, obj.generateUniqueKey().length());

}

}

完成业务逻辑。

App.java

package com.yiibai;

import java.util.UUID;

/\*\*

\* Generate a unique number

\*

\*/

public class App

{

public static void main( String[] args )

{

App obj = new App();

System.out.println("Unique ID : " + obj.generateUniqueKey());

}

public String generateUniqueKey(){

String id = UUID.randomUUID().toString();

return id;

}

}

## 6. Maven 打包

现在，我们将使用Maven这个项目，并输出编译成一个 “jar” 的文件。 请参考 pom.xml 文件，包元素定义应该包应该输出什么。

pom.xml

<project ...>

<modelVersion>4.0.0</modelVersion>

<groupId>com.yiibai</groupId>

<artifactId>NumberGenerator</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

在终端输入 mvn package :

C:\worksp\NumberGenerator> mvn package

它编译，运行单元测试并打包项目成一个 jar 文件，并把它放在 project/target 文件夹。如果出错：**ERROR: Unable to locate the Javac Compiler in:,  C:\Program Files (x86)\Java\jre6\..\lib\tools.jar,Please ensure you are using JDK 1.4 or above and,not a JRE (the com.sun.tools.javac.Main class is required)...**

参考：<http://my.oschina.net/u/1449336/blog/199802>

## 7. 示例

从项目的 jar 文件运行应用程序示例

C:\worksp\NumberGenerator>java -cp target/NumberGenerator-1.0-SNAPSHOT.jar com.yiibai.App

# 使用Maven创建Web应用程序项目

## 1. 从Maven模板创建Web项目

您可以通过使用Maven的maven-archetype-webapp模板来创建一个快速启动Java Web应用程序的项目。在终端(\* UNIX或Mac)或命令提示符(Windows)中，导航至您想要创建项目的文件夹。

键入以下命令：

$ mvn archetype:generate -DgroupId=com.yiibai -DartifactId=CounterWebApp -DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false

具体示例 ：

C:\worksp>mvn archetype:generate -DgroupId=com.yiibai -DartifactId=CounterWebApp -DarchetypeArtifactId=maven-archetype-webapp -DinteractiveMode=false

新的Web项目命名为 “CounterWebApp”，以及一些标准的 web 目录结构也会自动创建。

## 2. 项目目录布局

查看生成的项目结构布局：

.|\_\_\_\_CounterWebApp

||\_\_\_\_pom.xml

||\_\_\_\_src

|||\_\_\_\_main

||||\_\_\_\_resources

||||\_\_\_\_webapp

|||||\_\_\_\_index.jsp

|||||\_\_\_\_WEB-INF

||||||\_\_\_\_web.xml

Maven 产生了一些文件夹，一个部署描述符 web.xml，pom.xml 和 index.jsp。

**注意，**  
请查看[官方Maven标准目录布局指南](http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html)来了解更多。

web.xml – Servlet 2.3 已经比较旧, 建议升级到2.5

## **3. Eclipse IDE 支持**

要导入这个项目到Eclipse中，需要生成一些 Eclipse 项目的配置文件：

3.1、在终端，进入到 “CounterWebApp” 文件夹中，键入以下命令：

C:\worksp>cd CounterWebApp

C:\worksp\CounterWebApp>mvn eclipse:eclipse -Dwtpversion=2.0

**注意，**此选项 -Dwtpversion=2.0 告诉 Maven 将项目转换到 Eclipse 的 Web 项目(WAR)，而不是默认的Java项目(JAR)。为方便起见，以后我们会告诉你如何配置 pom.xml 中的这个 WTP 选项。

3.2 导入到 Eclipse IDE – File -> Import… -> General -> Existing Projects into workspace.

## **4. 更新POM**

在Maven中，Web项目的设置都通过这个单一的pom.xml文件配置。

1. 添加项目依赖 - Spring, logback 和 JUnit
2. 添加插件来配置项目

阅读注释清楚明了。

pom.xml

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

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

xsi:schemaLocation="<http://maven.apache.org/POM/4.0.0>  <http://maven.apache.org/maven-v4_0_0.xsd>">

<modelVersion>4.0.0</modelVersion>

<groupId>com.yiibai</groupId>

<artifactId>CounterWebApp</artifactId>

<packaging>war</packaging>

<version>1.0-SNAPSHOT</version>

<name>CounterWebApp Maven Webapp</name>

<url>**[Error! Hyperlink reference not valid.](http://maven.apache.org%3C/url%3E)** <properties>

<jdk.version>1.7</jdk.version>

<spring.version>4.1.1.RELEASE</spring.version>

<jstl.version>1.2</jstl.version>

<junit.version>4.11</junit.version>

<logback.version>1.0.13</logback.version>

<jcl-over-slf4j.version>1.7.5</jcl-over-slf4j.version>

</properties>

<dependencies>

<!-- Unit Test -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

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

</dependency>

<!-- Spring Core -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

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

<exclusions>

<exclusion>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>jcl-over-slf4j</artifactId>

<version>${jcl-over-slf4j.version}</version>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

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

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

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

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

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

</dependency>

<!-- jstl -->

<dependency>

<groupId>jstl</groupId>

<artifactId>jstl</artifactId>

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

</dependency>

</dependencies>

<build>

<finalName>CounterWebApp</finalName>

<plugins>

    <!-- Eclipse project -->

  <plugin>

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

<artifactId>maven-eclipse-plugin</artifactId>

<version>2.9</version>

<configuration>

        <!-- Always download and attach dependencies source code -->

<downloadSources>true</downloadSources>

<downloadJavadocs>false</downloadJavadocs>

<!-- Avoid type mvn eclipse:eclipse -Dwtpversion=2.0 -->

<wtpversion>2.0</wtpversion>

</configuration>

  </plugin>

  <!-- Set JDK Compiler Level -->

  <plugin>

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

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

<version>2.3.2</version>

<configuration>

<source>${jdk.version}</source>

<target>${jdk.version}</target>

</configuration>

  </plugin>

  <!-- For Maven Tomcat Plugin -->

  <plugin>

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

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

<version>2.2</version>

<configuration>

<path>/CounterWebApp</path>

</configuration>

  </plugin>

</plugins>

</build>

</project>

**注意，**为方便起见，声明 maven-eclipse-plugin，并配置wtpversion 来避免输入参数 -Dwtpversion=2.0。现在，每次使用 mvn eclipse:eclipse，Maven将这个项目导入转换为 Eclipse Web 项目。

#之前

mvn eclipse:eclipse --> Eclipse Java project (JAR)

mvn eclipse:eclipse -Dwtpversion=2.0 --> Eclipse Java web project (WAR)

#之后

mvn eclipse:eclipse --> Eclipse Java web project (WAR)

## **5. 更新源代码**

在这一步中，在上一步配置完pom.xml后，重新执行 mvn eclipse:eclipse 这个命令，我们将创建Spring MVC的一些文件和logback日志框架的文件夹，最终的项目结构如下所示：

.

|\_\_\_\_pom.xml

|\_\_\_\_src

| |\_\_\_\_main

| | |\_\_\_\_java

| | | |\_\_\_\_com

| | | | |\_\_\_\_yiibai

| | | | | |\_\_\_\_controller

| | | | | | |\_\_\_\_BaseController.java

| | |\_\_\_\_resources

| | | |\_\_\_\_logback.xml

| | |\_\_\_\_webapp

| | | |\_\_\_\_WEB-INF

| | | | |\_\_\_\_mvc-dispatcher-servlet.xml

| | | | |\_\_\_\_pages

| | | | | |\_\_\_\_index.jsp

| | | | |\_\_\_\_web.xml

**注意，如果它不存在，需要手动创建文件夹。**

5.1 创建 Spring MVC 的控制器类。

/src/main/java/com/yiibai/controller/BaseController.java

package com.yiibai.controller;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Controller;

import org.springframework.ui.ModelMap;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

@Controller

public class BaseController {

private static int counter = 0;

private static final String VIEW\_INDEX = "index";

private final static org.slf4j.Logger logger = LoggerFactory.getLogger(BaseController.class);

@RequestMapping(value = "/", method = RequestMethod.GET)

public String welcome(ModelMap model) {

model.addAttribute("message", "Welcome");

model.addAttribute("counter", ++counter);

logger.debug("[welcome] counter : {}", counter);

// Spring uses InternalResourceViewResolver and return back index.jsp

return VIEW\_INDEX;

}

@RequestMapping(value = "/{name}", method = RequestMethod.GET)

public String welcomeName(@PathVariable String name, ModelMap model) {

model.addAttribute("message", "Welcome " + name);

model.addAttribute("counter", ++counter);

logger.debug("[welcomeName] counter : {}", counter);

return VIEW\_INDEX;

}

}

5.2 创建Spring配置文件。

/src/main/webapp/WEB-INF/mvc-dispatcher-servlet.xml

<beans xmlns="<http://www.springframework.org/schema/beans>"

xmlns:context="<http://www.springframework.org/schema/context>"

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

xsi:schemaLocation="

<http://www.springframework.org/schema/beans>

<http://www.springframework.org/schema/beans/spring-beans.xsd>         <http://www.springframework.org/schema/context>

<http://www.springframework.org/schema/context/spring-context.xsd>">

<context:component-scan base-package="com.yiibai.controller" />

<bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix">

<value>/WEB-INF/pages/</value>

</property>

<property name="suffix">

<value>.jsp</value>

</property>

</bean>

</beans>

5.3 更新让现有的 web.xml 支持 Servlet 2.5(默认的Servlet2.3 太旧了)， 并且还通过 Spring 监听器 ContextLoaderListener 集成了Spring框架。

/src/main/webapp/WEB-INF/web.xml

<web-app xmlns="<http://java.sun.com/xml/ns/javaee>"

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

xsi:schemaLocation="<http://java.sun.com/xml/ns/javaee>

<http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd>"

version="2.5">

<display-name>Counter Web Application</display-name>

<servlet>

<servlet-name>mvc-dispatcher</servlet-name>

<servlet-class>

                        org.springframework.web.servlet.DispatcherServlet

                </servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>mvc-dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/mvc-dispatcher-servlet.xml</param-value>

</context-param>

<listener>

<listener-class>

                  org.springframework.web.context.ContextLoaderListener

                </listener-class>

</listener>

</web-app>

5.4 移动文件 index.jsp 到 WEB-INF/pages 目录下， 为了保护直接访问。并更新内容：

/src/main/webapp/WEB-INF/pages/index.jsp

5.5 在资源文件夹(resources)中创建 logback.xml 文件

/src/main/resources/logback.xml

<?xml version="1.0" encoding="UTF-8"?>

## **6. Eclipse + Tomcat**

在第5步中创建所有文件以后，这里有一些方法可以用来部署和测试Web项目，我们这里推荐使用6.2中的方法。

6.1 要编译，测试和项目打包成一个WAR文件，输入：

mvn package

一个新的 WAR 文件将在 project/target/CounterWebApp.war产生，只需复制并部署到Tomcat 发布的目录。

6.2 如果想通过 Eclipse 服务器这个项目插件(Tomcat 或其它容器)调试，这里再输入：

mvn eclipse:eclipse

如果一切顺利，该项目的依赖将被装配附加到 Web部署项目。图片: 右键点击 project -> Properties -> Deployment Assembly

6.3 Maven 的 Tomcat 插件声明(加入到 pom.xml)：

pom.xml

<!-- For Maven Tomcat Plugin -->

键入以下命令(有时网络不通畅需要执行2-3次)：

mvn tomcat:run

这将启动Tomcat，部署项目默认在端口8080。

出错：Maven项目下update maven后Eclipse报错：java.lang.ClassNotFoundException: ContextLoaderL

解决方案：

**1.右键点击项目--选择Properties**

选择Deployment Assembly,在右边点击Add按钮，在弹出的窗口中选择Java Build Path Entries

**2.点击Next**，选择Maven Dependencies

**3.点击Finish，**然后可以看到已经把Maven Dependencies添加到Web应用结构中了

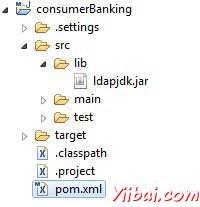
操作完后，重新部署工程，不再报错了。然后我们再到.metadata\.plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps\目录下，发现工程WEB-INF目录下自动生成了lib目录，并且所有的依赖jar包也都已经部署进来。问题因此解决。

# Maven外部依赖

举一个例子，让我们在 [Maven创建项目](http://www.yiibai.com/maven/maven_creating_project.html) 这一章节中创建的项目做以下的修改。

* 添加 lib 文件夹到 src 文件夹
* 复制任何的 jar 到 lib 文件夹。这里使用的是 ldapjdk.jar，这是 LDAP 操作的辅助库。

现在我们的项目结构看起来应该类似下面这样：



在这里，在项目中指定自己所用的库，它可以包含 jar 文件，但是可能无法在任何 Maven 存储库找到，那么需要从部下载。如果代码使用这个 Maven 库但没有办法找到，那么 Maven 构建将会失败，因为它在编译阶段使用指这个库无法下载或无法找到。

要处理这种情况，需要添加外部依赖项，如使用下列方式在 Maven 的 pom.xml 。

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

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

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

http://maven.apache.org/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>

再看上面例子中的第二个依赖元素（dependency），它清除以下有关外部依赖的重要概念。

* 外部依赖（JAR库的位置）可以在 pom.xml 中配置为与其他依赖的方式相同；
* 指定 groupId 同样作为库的名称；
* 指定 artifactId 同样作为库的名称
* 指定范围的系统；
* 指定相系统项目的位置；

# Maven项目文档

本教程学习如何一步到位地创建应用程序的文档。因此现在开始我们进入到  C:\MVN 创建 java 应用程序项目：consumerBanking。 进入到项目文件夹中执行以下命令 mvn 命令。

C:\MVN>mvn site

完成后，在你的项目文件已构建完成。 Maven 会在 target  目录中创建一个 site 目录，

点击 index.html 打开此文档，Maven会使用一个文件处理引擎： Doxia，它将会读取多个源格式并将它们转换为通用文档模型文档。以下几个是常用的格式使用来编写项目文档，这是由 Doxia 解析编写内容。

|  |  |  |
| --- | --- | --- |
| **格式名称** | **描述** | **参考** |
| APT | 纯文本文档格式 | <http://maven.apache.org/doxia/format.html> |
| XDoc | Maven1.x 的文档格式 | <http://jakarta.apache.org/site/jakarta-site2.html> |
| FML | 用于常问问题(FQA)文件 | <http://maven.apache.org/doxia/references/fml-format.html> |
| XHTML | 可扩展HTML | <http://en.wikipedia.org/wiki/XHTML> |

# Maven项目模板

mvn archetype:generate

## 什么是Archetype？

Archetype 是一个 Maven 插件，其任务是按照其模板来创建一个项目结构。在这里我们将使用 *quickstart*原型插件来创建一个简单的 Java应用程序。

## 使用项目模板

让我们打开命令控制台，进入到 **C:\>MVN**目录，然后执行以下命令 mvn 命令，如下代码所示：

C:\MVN>mvn archetype:generate

按 Enter 键选择默认选项（203：maven-archetype-quickstart）

Maven会要求原型的特定版本

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

Maven会要求填写项目细节信息。如果要使用默认值可直接按回车。也可以通过输入自己的值覆盖它们。

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会要求确认项目的细节信息，可按回车键或按 Y 来确认。

Confirm properties configuration:

groupId: com.companyname.insurance

artifactId: health

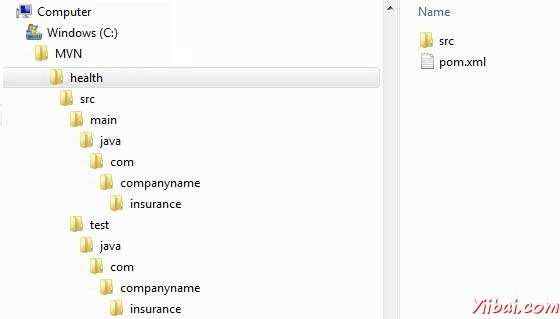
version: 1.0-SNAPSHOT

package: com.companyname.insurance

Y:

## 创建项目

现在进入到 C:\mvn 目录。会看到有一个 java 应用程序项目已创建了，它是在创建项目时给出 artifactId 命名：health 。 Maven 将创建一个标准的目录结构布局，如下图所示：



## 创建pom.xml

Maven 项目中的生成如下所列出 pom.xml 文件，其内容如下：

<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.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>

## 创建App.java

Maven 示例生成 Java 源文件，App.java下面列出项目：

位置：C:\>MVN\health\src\main\java\com\companyname\insurance> App.java

**package com.companyname.insurance;**

**/\*\***

**\* Hello world!**

**\***

**\*/**

**public class App**

**{**

**public static void main( String[] args )**

**{**

**System.out.println( "Hello World!" );**

**}**

**}**

## 创建 AppTest.java

Maven 实例生成 Java 源测试文件，项目中的 AppTest.java 测试文件如下面列出：

位置: C:\ > MVN > health > src > test > java > com > companyname > insurance > AppTest.java

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 );

}

}

就是这样。现在就可以看到 Maven 的功能了。可以使用 maven 单一命令来创建任何类型的项目并开始开发。

# Maven快照

## 什么是快照？

快照（SNAPSHOT ）是一个特殊版本，指出目前开发拷贝。不同于常规版本，Maven 每生成一个远程存储库都会检查新的快照版本。

现在，数据服务团队将在每次发布代码后更新快照存储库为：data-service:1.0-SNAPSHOT 替换旧的 SNAPSHOT jar。

## app-ui pom.xml

app-ui 项目使用数据服务（data-service）的  1.0-SNAPSHOT 

<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 pom.xml

数据服务（data-service）项目对于每一个微小的变化释放 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，在使用快照（SNAPSHOT）时，Maven 自动获取最新的快照版本。不过我们也可以强制使用 -U 切换到任何 maven 命令来下载最新的快照版本。

mvn clean package -U

打开命令控制台，进入到 C: \MVN\app-ui  目录，然后执行以下命令mvn命令。

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

# Maven依赖管理

## 传递依赖发现

这是很通常情况下，当一个库说A就依赖于其他库说B的情况下，另一个项目Ç想用A，则该项目需要使用库中B。Maven提供一些功能来控制传递依赖程度

|  |  |
| --- | --- |
| **Feature** | **描述** |
| Dependency mediation | Determines what version of a dependency is to be used when multiple versions of an artifact are encountered. If two dependency versions are at the same depth in the dependency tree, the first declared dependency will be used. |
| Dependency management | Directly specify the versions of artifacts to be used when they are encountered in transitive dependencies. For an example project C can include B as a dependency in its dependencyManagement section and directly control which version of B is to be used when it is ever referenced. |
| Dependency scope | Includes dependencies as per the current stage of the build |
| Excluded dependencies | Any transitive dependency can be excluede using "exclusion" element. As example, A depends upon B and B depends upon C then A can mark C as excluded. |
| Optional dependencies | Any transitive dependency can be marked as optional using "optional" element. As example, A depends upon B and B depends upon C. Now B marked C as optional. Then A will not use C. |

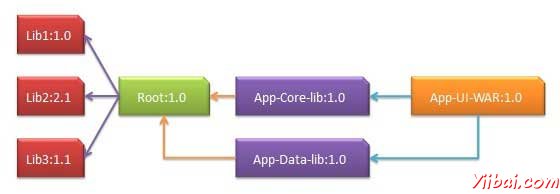
## 依赖范围

传递依赖发现可以使用各种依赖范围如下文所述受到限制

|  |  |
| --- | --- |
| **Scope** | **描述** |
| compile | This scope indicates that dependency is available in classpath of project. It is default scope. |
| provided | This scope indicates that dependency is to be provided by JDK or web-Server/Container at runtime. |
| runtime | This scope indicates that dependency is not required for compilation, but is required during execution. |
| test | This scope indicates that the dependency is only available for the test compilation and execution phases. |
| system | This scope indicates that you have to provide the system path. |
| import | This scope is only used when dependency is of type pom. This scopes indicates that the specified POM should be replaced with the dependencies in that POM's <dependencyManagement> section. |

## 依赖关系管理

通常情况下，我们已经一套项目在一个共同的项目下。在这种情况下，我们可以创造让所有的公共依赖一个共同的POM，然后进行分项目POMS为这个POM父。下面的例子将帮助你理解这个概念



以下是上述的依赖图的细节

* APP-UI-WAR依赖于App-Core-lib和 App-Data-lib。
* Root 是 App-Core-lib 和 App-Data-lib 的父类。
* Root 定义LIB1，LIB2，Lib3作为其依赖部分依赖关系。

从上面的例子中，我们可以学到以下关键概念

* 常见的依赖关系可以用父POM的概念被放置在一个地方。 App-Data-lib 和 App-Core-lib 项目列表在 Root 目录(见Roots包类型。它是POM).
* 不需要Lib1, lib2, Lib3 作为依赖于 App-UI-WAR. Maven使用传递性依赖机制来管理这些细节。

# Maven自动化部署

在项目开发中，通常是部署过程包含以下步骤

* 检入代码在建项目全部进入SVN或源代码库中，并标记它。
* 从SVN下载完整的源代码。
* 构建应用程序。
* 生成输出要么WAR或EAR文件存储到一个共同的网络位置。
* 从网络获取的文件和文件部署到生产现场。
* 更新日期和应用程序的更新版本号的文件。

## 解决

通过结合自动化的部署过程

* Maven构建和释放项目，
* SubVersion源代码库，管理源代码，
* 和远程存储库管理器（Jfrog/ Nexus）来管理项目的二进制文件。

## 更新项目的pom.xml

我们将使用Maven发布插件来创建一个自动释放过程。

例如：bus-core-api 项目POM.xml

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

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

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

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

<modelVersion>4.0.0</modelVersion>

<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中，下面是我们使用的重要元素

|  |  |
| --- | --- |
| **元素** | **描述** |
| SCM | Configures the SVN location from where Maven will check out the source code. |
| Repositories | Location where built WAR/EAR/JAR or any other artifact will be stored after code build is successful. |
| Plugin | maven-release-plugin is configured to automate the deployment process. |

## Maven发布插件

Maven使用确实下列有用的任务*maven-release-plugin*.

mvn release:clean

它清除以防工作区的最后一个释放的过程并不顺利。

mvn release:rollback

回滚是为了以防工作空间代码和配置更改的最后一个释放的过程并不顺利。

mvn release:prepare

执行多个操作次数

* 检查是否有任何未提交的本地更改或不
* 确保没有快照依赖
* 更改应用程序的版本并删除快照从版本，以释放
* 更新文件到 SVN.
* 运行测试用例
* 提交修改后POM文件
* 标签代码在subversion中
* 增加版本号和附加快照以备将来发行
* 提交修改后的POM文件到SVN。

mvn release:perform

检查出使用前面定义的标签代码并运行Maven的部署目标来部署战争或内置工件档案库。

让我们打开命令控制台，到 **C: > MVN >bus-core-api**目录并执行以下命令mvn命令。

C:MVNus-core-api>mvn release:prepare

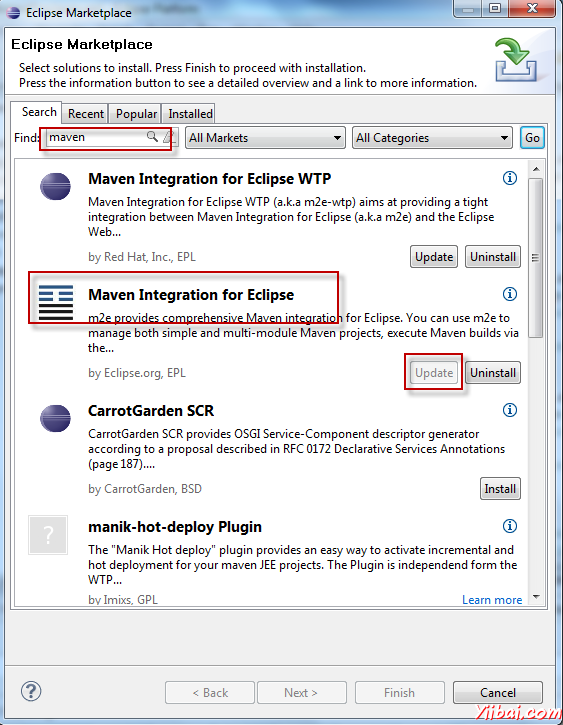
Maven将开始建设该项目。一旦构建成功运行以下命令mvn命令。

C:MVNus-core-api>mvn release:perform

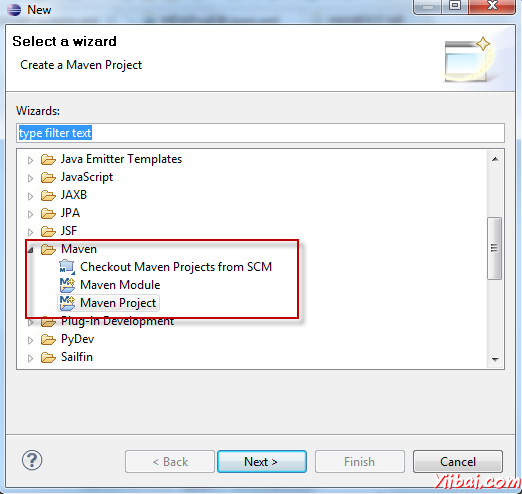
一旦构建成功，您可以在资料库验证上传的JAR文件。

# Eclipse构建Maven项目

**1. 安装m2eclipse插件**  
   要用Eclipse构建Maven项目，我们需要先安装meeclipse插件   
   点击eclipse菜单栏Help->Eclipse Marketplace搜索到插件Maven Integration for Eclipse 并点击安装即可，如下图：



安装成成之后我们在Eclipse菜单栏中点击File->New->Other,在弹出的对话框中会看到如下图所示：

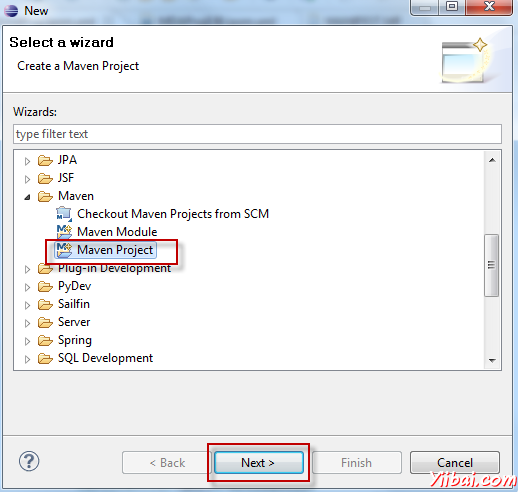


**2. 构建Maven项目**

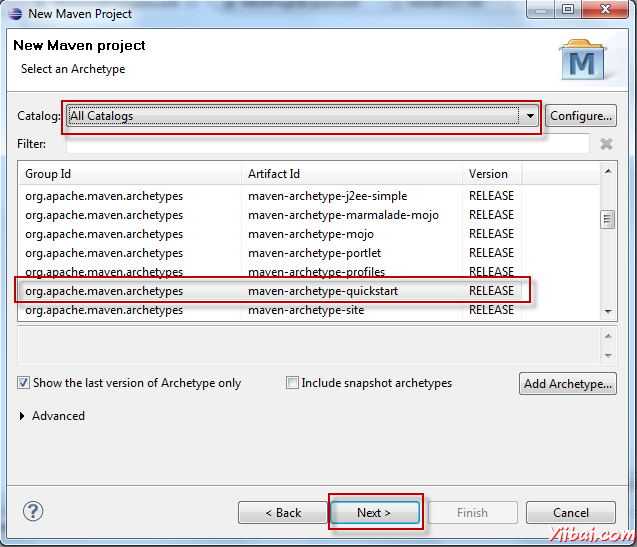
      以eclipse3.6为例

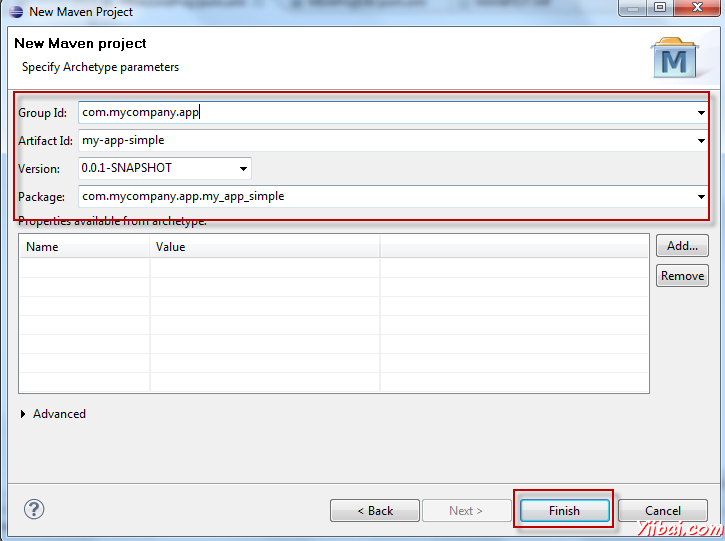
      1）创建简单Maven项目

           点击Eclipse菜单栏File->New->Ohter->Maven得到如下图所示对话框：

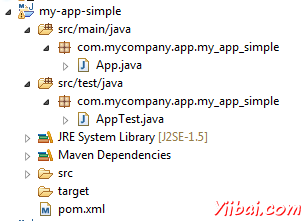


选中Maven Project并点击Next，到下一个对话框继续点击Next得到如下对话框

  
 如图示操作，选择maven-archetype-quickstart，点击Next

  
 按图示填写好groupId, artfactId,version等信息，点击Finish。

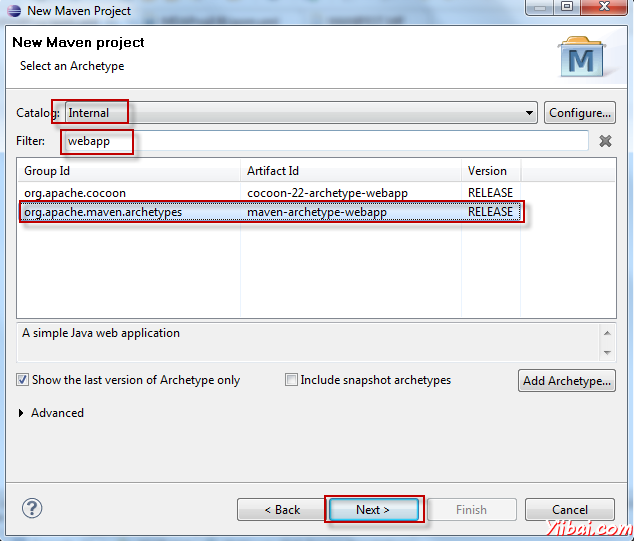
由此我们成功创建了一个简单的Maven项目，项目结构如图所示



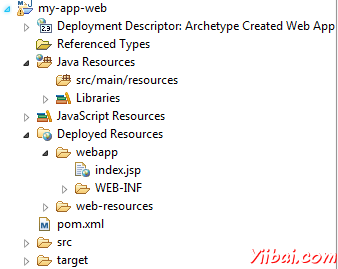
     2）创建Maven web项目

     操作跟创建简单Maven项目类似，点击Eclipse菜单File->New->Other->Maven->Maven Project

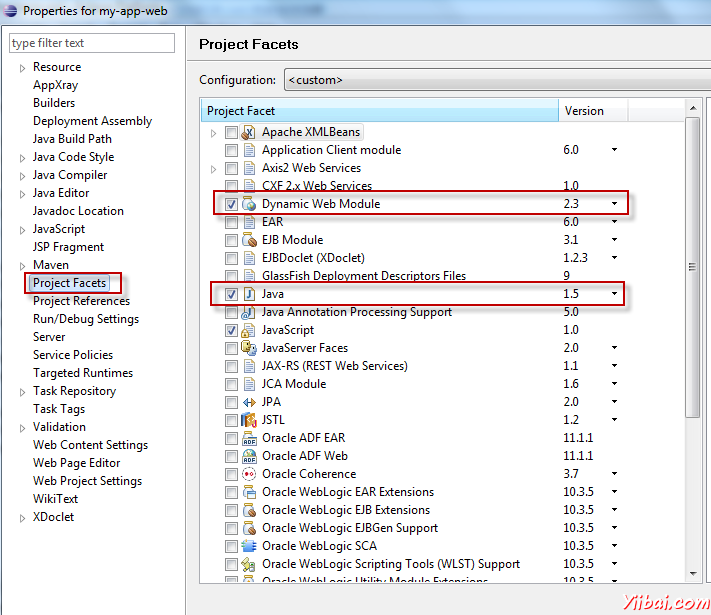
     在选择maven-archetype的界面进行如下操作：

  
 点击Next,填写好相应的groupId,artifactId,version等信息，点击Finish

得到的Maven web项目结构如下图所示：



 右击项目，点击Properties->Project Facets

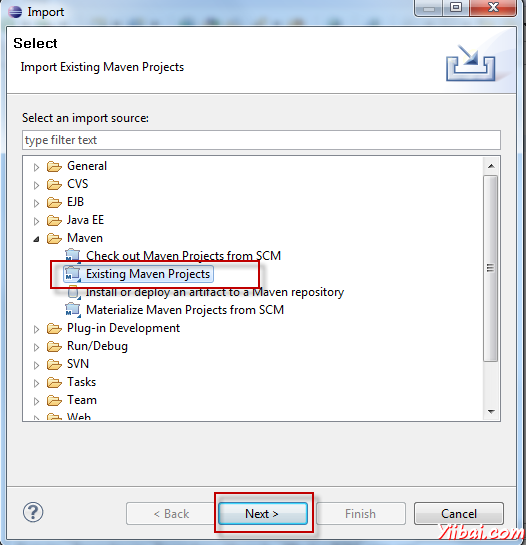


如上图可以看到项目为web2.3 java1.5 当然我们也可以改成我们所需要的版本，打开xml文件my-app-web/.settings/org.eclipse.wst.common.project.facet.core.xml，进行修改即可：

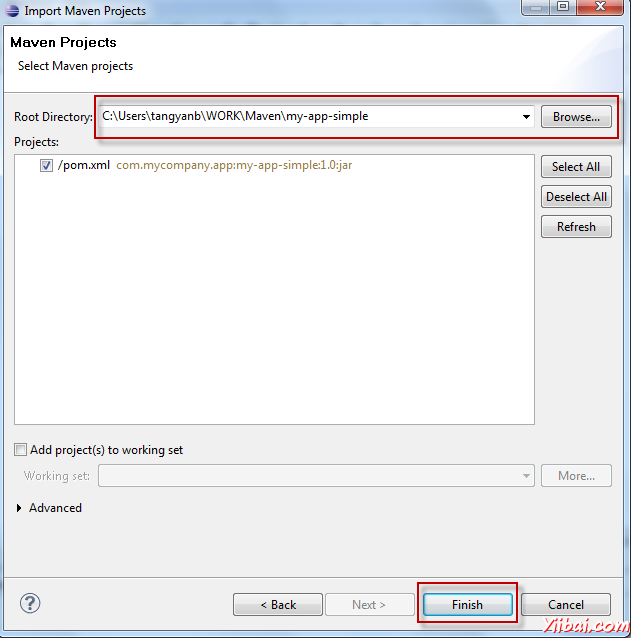
**Xml代码  藏代码**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<faceted-project>**
3. **<fixed** facet="wst.jsdt.web"**/>**
4. **<installed** facet="java" version="1.5"**/>**
5. **<installed** facet="jst.web" version="2.3"**/>**
6. **<installed** facet="wst.jsdt.web" version="1.0"**/>**
7. **</faceted-project>**

     3）导入Maven项目

          在Eclipse project explorer中右击，在弹出框中选择import，得到如下图所示：            


      选择Existing Maven Projects，并点击Next，得到如下图所示对话框：

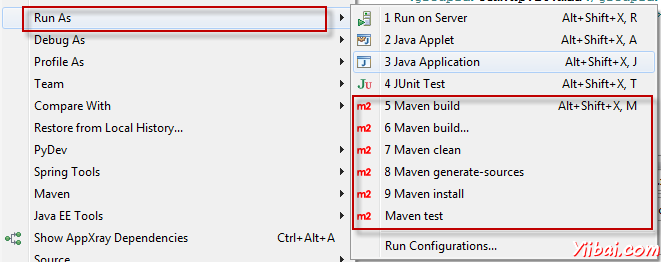


选择一个已经创建好的Maven项目，并点击Finish。

由此，导入Maven项目成功

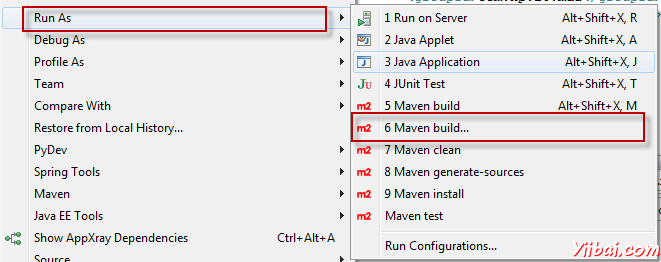
**3. 运行Maven命令**

    右击项目，点击Run as，如下图：

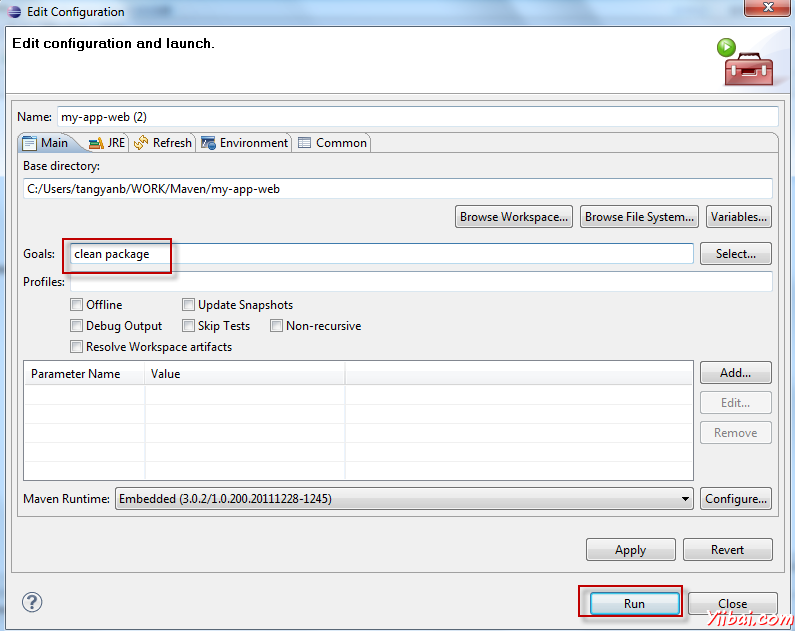


 即可看到有很多现有的maven命令，点击即可运行，并在控制台可以看到运行信息

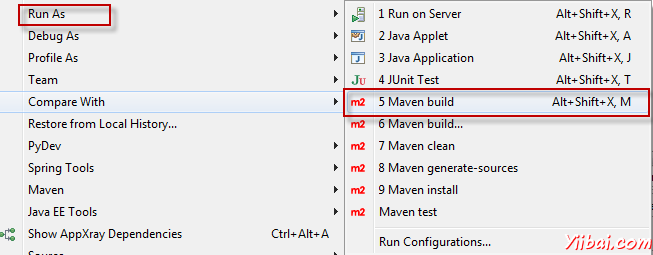
 如果你想运行的maven命令在这里没有找到，点击Maven build创建新的命令，操作如下图所示：

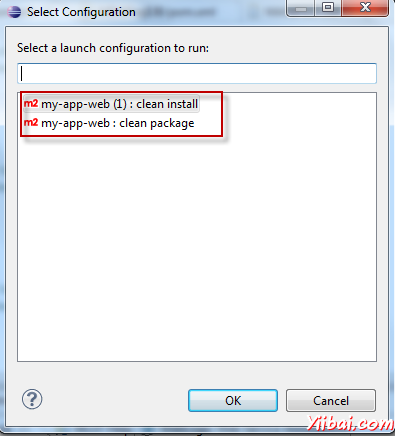


 如下图填入Maven命令，点击Run即可



新增的maven命令可以通过如下方式找到，并再次运行：





# 部署基于Maven的war文件到Tomcat

## 1. Tomcat 7 示例

这个例子说明了如何在Tomcat7打包并部署WAR文件。

1.1 Tomcat 认证  
添加具有角色管理器GUI和管理脚本的用户。

%TOMCAT7\_PATH%/conf/tomcat-users.xml

<?xml version='1.0' encoding='utf-8'?>

<tomcat-users>

<role rolename="manager-gui"/>

<role rolename="manager-script"/>

<user username="admin" password="password" roles="manager-gui,manager-script" />

</tomcat-users>

**1.2 Maven 认证**  
添加在上面Maven 设置文件的 Tomcat 用户，是之后Maven使用此用户来登录Tomcat服务器。

%MAVEN\_PATH%/conf/settings.xml

<?xml version="1.0" encoding="UTF-8"?>

<settings ...>

<servers>

<server>

<id>TomcatServer</id>

<username>admin</username>

<password>password</password>

</server>

</servers>

</settings>

**1.3 Tomcat7 Maven 插件**  
声明一个Maven的Tomcat插件。

pom.xml

<plugin>

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

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

<version>2.2</version>

<configuration>

<url>http://localhost:8080/manager/text</url>

<server>TomcatServer</server>

<path>/yiibaiWebApp</path>

</configuration>

</plugin>

**怎么运行的？**  
在部署过程中，它告诉 Maven 通过部署 WAR 文件Tomcat服务器， “http://localhost:8080/manager/text” , 在路径“/yiibaiWebApp“上, 使用 “TomcatServer” (settings.xml) 用户名和密码来进行认证。

**1.4 发布到Tomcat**  
以下的命令是用来操纵Tomcat WAR文件。

mvn tomcat7:deploy

mvn tomcat7:undeploy

mvn tomcat7:redeploy

**示例**

> mvn tomcat7:deploy

## 2. Tomcat 6 示例

这个例子说明了如何在Tomcat6打包和部署WAR文件。这些步骤和Tomcat7是一样的，只是部署URL和命令名称不同。

2.1 Tomcat 认证

%TOMCAT6\_PATH%/conf/tomcat-users.xml

<?xml version='1.0' encoding='utf-8'?>

<tomcat-users>

<role rolename="manager-gui"/>

<role rolename="manager-script"/>

<user username="admin" password="password" roles="manager-gui,manager-script" />

</tomcat-users>

**2.2 Maven 认证**

%MAVEN\_PATH%/conf/settings.xml

<?xml version="1.0" encoding="UTF-8"?>

<settings ...>

<servers>

<server>

<id>TomcatServer</id>

<username>admin</username>

<password>password</password>

</server>

</servers>

</settings>

**2.3 Tomcat6 Maven 插件**

pom.xml

<plugin>

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

<artifactId>tomcat6-maven-plugin</artifactId>

<version>2.2</version>

<configuration>

<url>http://localhost:8080/manager</url>

<server>TomcatServer</server>

<path>/yiibaiWebApp</path>

</configuration>

</plugin>

**2.4 发布到Tomcat**

mvn tomcat6:deploy

mvn tomcat6:undeploy

mvn tomcat6:redeploy

**示例**

> mvn tomcat6:deploy

pom.xml

groupId🡪

artifactId🡪

version🡪0.0.1-SNAPSHOT

x.y.z（x:架构变化，y:分支变化，z:分支的更新）-里程碑—>SNAPSHOT，alpha（项目内部测试版），beta（使用人员试用），Release（释放版本），GA（可靠版本）

版本管理：

如：cms0.0.1-SNAPSHOT🡪cms0.0.1-Release|增加功能并大变化🡪cms1.0.1-SNAPSHOT

|重新调整🡪cms0.0.1-SNAPSHOT

|cms0.1.1-release

SVN是版本控制工具。

src

main

java

package

resource

test

java

package

resources

target

classes

mvn:compile

mvn:test

mvn:clean

mvn:install

mvn:package

<dependencies>

<!-- service -->

<dependency>

<groupId>esm.sdk</groupId>

<artifactId>sdk-java-api</artifactId>

<version>${project.parent.version}</version>

</dependency>

</dependencies>

<groupId>org.json</groupId>

<artifactId>json</artifactId>

<version>20160212</version>

<scope>value</scope>

scope中的值有：

1. test范围指的是测试范围有效，在编译和打包时都不会有这个依赖。不会把依赖传递过去。
2. compile范围指的是编译范围有效，在编译和打包时都会将依赖存储进去
3. provided依赖，在编译和测试的过程有效，最后生成war包时不会加入：诸如：service-api，因为servlet-api在tomcat等服务器已经存在了，如果再打包会冲突。
4. runtime在运行的时候依赖，在编译的时候不依赖。

默认的依赖范围为compile。

4.Maven的依赖特性：

A🡪l1.0

B🡪l2.0

C🡪A,B

A与L直接依赖

C与L间接依赖

当依赖级别相同的时候，哪一个先写，则依赖哪一个版本。

当依赖级别不相同的时候，采用依赖级别最短的那个。

当包产生冲突时，排除某个依赖：

<version>20160212</version>

<exclusions>

<exclusion>

<groupId>commons-logging</groupId>

</exclusion>

</exclusions>

1. Maven的聚合和继承：

聚合，是模块的位置。

两个项目外边有一个聚合的pom.xml文件

导入三个模块聚合在一块：

<modules>

<module>../user-core</module>

<module>../user-log</module>

<module>../user-service</module>

</module>

继承，绝对路径为pom.xml文件。

<parent>

<groupId>esm.sdk</groupId>

<artifactId>sdk-java-super-pom</artifactId>

<version>2.0-SNAPSHOT</version>

<relativePath>../user-parent/pom.xml</relativePath>

</parent>

依赖管理：

<dependencyManagement></dependencyManagement>

1.利用maven骨架自动生成项目：

mvn archetype:generate –DgroupId=zttc.itat.maven –DartifactId=maven-ch03 –Dversion=0.0.1-SNAPSHOT

2.数据库测试用dbunit.

7.Nexus的介绍和安装

A—>core

B—>service

C—>action

采用Sonatype的Nexus可以用来做私服。

修改bin/jsw/com/wrapper.conf中的xxx=java

执行：nexus intall安装。

Localhost:8081

JQuery MiniUI和EXT都是重量级框架。

mvn:deploy—提交到SNAPSHOT，RELEASE，3rd Party（第三方）。

1. 配置项目不从中央工厂下载，可以在项目的pom.xml文件中设置，也可以在maven的settings.xml文件中设置。
2. pom.xml中设置：

<repositories>  
 <repository>  
 <id>nexus</id>  
 <name>Nexus Repository</name>  
 <url>http://localhost:8081/nexus/content/groups/public/</url>

<release>

<enable>true</enable>

</release>

<!—snapshot默认是关闭的，需要手动开启—>

<snapshot>

<enable>true</enable>

</snapshot>  
 </repository>  
</repositories>

1. setting.xml中设置：

<profile>

<repositories>

<repository>

<id>nexus</id>

<name>Nexus Repository</name>

<url>http://localhost:8081/nexus/content/groups/public/</url>

<release>

<enable>true</enable>

</release>

<!--snapshot默认是关闭的，需要手动开启-->

<snapshot>

<enable>true</enable>

</snapshot>

</repository>

</repositories>

<id>jdk-1.7</id>

<activation>

<activeByDefault>true</activeByDefault>

<jdk>1.7</jdk>

</activation>

<properties>

<maven.compiler.source>1.7</maven.compiler.source>

<maven.compiler.target>1.7</maven.compiler.target>

<maven.compiler.compilerVersion>1.7</maven.compiler.compilerVersion>

</properties>

</profile>

</profiles>

<activeProfiles>

<!--只有激活才能生效-->

<activeProfile>alwaysActiveProfile</activeProfile>

<activeProfile>anotherAlwaysActiveProfile</activeProfile>

</activeProfiles>

当nexus关闭后，就会去中央工厂下载。在maven-model-builder-3.0.5.jar下有指定。

<!--工厂的镜像，只要mirrorOf的工厂要访问，都会自动来找镜像，如果镜像无法访问就不会再去中央工厂，使用\*表示所有的工厂都使用这个镜像访问.

<mirror>

<id>mirrorId</id>

<mirrorOf>\*</mirrorOf>

<name>Human Readable Name for this Mirror.</name>

<url>http://localhost:8081 /repo/path</url>

</mirror>

-->

9.发布项目到Nexus中。

a. 在maven的setting.xml中设置：

<server>  
 <id>user-aaa</id>  
 <username>repouser</username>  
 <password>repouser/password>  
</server>

b.在项目的pom.xml文件中配置。

<distributionManagement>  
 <repository>  
 <id>user-aaa</id>  
 <name>user release resp</name>  
 <url>http://localhost:8081/nexus/content/repositories/release/</url>  
 </repository>  
</distributionManagement>

10.maven的生命周期。

clean 和 compile和site的生命周期。

clean的生命周期

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

compile的生命周期

validate

generate-sources

process-sources

generate-resource

process-resources 复制并处理资源文件，至目标目录，准备打包。

compile 编译项目的源代码

process-classes

generate-test-sources

process-test-sources

generate-test-resources

process-test-resource 复制并处理资源文件，至目标测试目录。

test-compile 编译测试源代码

process-test-classes

test 使用合适的单元测试框架运行测试。这些测试代码不会被打包或部署。

prepare-package

package 接收编译好的代码，打包成可发布的格式，如：JAR。

pre-integration-test

integration-test

post-integration-test

verify

install 将包安装至本地仓库，以让其他项目依赖。

deploy 将最终的包复制到远程的仓库，以让其它开发人员与项目共享。

1. site

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程，并且为部署做准备。

对源文件打包，把源文件打包到jar中：clean source:jar

<build>  
 <finalName>HelloWorld</finalName>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-source-plugin</artifactId>  
 <version>2.2.1</version>  
 <executions>  
 <execution>  
 <phase>compile</phase>  
 <goals><goal>jar</goal><goal>test-jar</goal></goals>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
</build>

11. maven插件基础及应用

仓库和插件生命周期

本地仓库、私有仓库、中央仓库

举例几个插件的应用：

<build>

<pluginManagement>

<plugins>

<plugin>

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

<artifactId>maven-rar-plugin</artifactId>

<version>2.3</version>

<executions>

<execution>

<goals><goal>rar</goal></goals>

<phase>package</phase>

</execution>

</executions>

<configuration>

<includeJar>true</includeJar>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>sql-maven-plugin</artifactId>

<version>1.5</version>

<!-- 使用插件依然可以指定相应的依赖 -->

<dependencies>

<dependency>

<groupId>mysql</groupId>

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

<version>5.1.24</version>

</dependency>

</dependencies>

<configuration>

<driver>com.mysql.jdbc.Driver</driver>

<url></url>

<username></username>

<password></password>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

12. 测试（TestNG，JUnit）

在src/test/java中以Test\*\*开头、\*\*Test、\*\*TestCase等命名方式的类才能被测试到。

在maven-surefire-plugin中查看有哪些配置参数。

采用mvn xxx –DskipTests就可以跳过测试。

mvn xxx –Dtest-Hello.java-只测试Hello.java

利用cobertura进行覆盖测试。

<build>

<finalName>rest\_test</finalName>

<defaultGoal>compile</defaultGoal>

<plugins>

<plugin>

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

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

<version>3.1</version>

<configuration>

<source>1.7</source>

<target>1.7</target>

<compilerArgs>

<arg>-verbose</arg>

<arg>-Xlint:unchecked</arg>

</compilerArgs>

<!-- 此处设置了skip等于设置把所有的测试都编译都跳过，如果测试类写的有问题，也略过 -->

<!-- <skip>true</skip> -->

</configuration>

</plugin>

</plugins>

</build>

Maven:隐式变量

Cargo：可以远程发布项目。