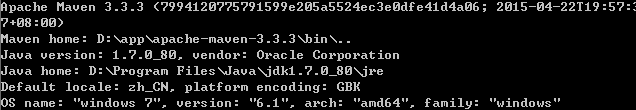
**Maven培训**

1. 安装
2. 安装JDK
3. Maven下载地址：<http://maven.apache.org/download.cgi>
4. 解压
5. 配置环境变量：

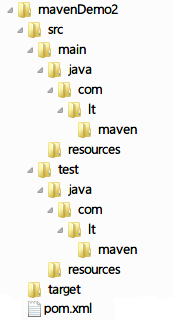
增加MAVEN\_HOME = Maven包解压地址

在PATH里增加%MAVEN\_HOME%\bin;

1. 测试命令：mvn -v 显示如下图则表示成功



1. 常用命令
2. 文件结构：Maven的目录结构如下：



1. 打开pom.xml，添加如下内容

<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.lt</groupId>

<artifactId>mavenDemo2</artifactId>

<packaging>jar</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>mavenDemo2</name>

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

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

</dependency>

</dependencies>

<build>

<finalName>mavenDemo2</finalName>

</build>

<!-- 设置编码 -->

<properties>

<project.build.sourceEncoding>

UTF-8

</project.build.sourceEncoding>

</properties>

</project>

pom.xml文件简单解释（groupId，artifactId，version三个一起描述了项目的唯一标识）：

* groupId - 组名，主项目标识；
* artifactId - 工程名，子项目（模块）标识；
* packaging - 打包方式；
* version - 版本；
* name - 项目描述名；

1. 在 src/main/java/com/lt/maven 下新建 Hello.java 类文件，代码如下：

package com.lt.maven;

public class Hello {

public String sayHello(String name){

return "Hello "+name+"!";

}

}

1. 在 src/test/java/com/lt/maven 下新建 HelloTest.java 测试类文件，代码如下：

package com.lt.maven;

import org.junit.Test;

import static junit.framework.Assert.\*;

public class HelloTest {

@Test

public void testHello(){

Hello hello = new Hello();

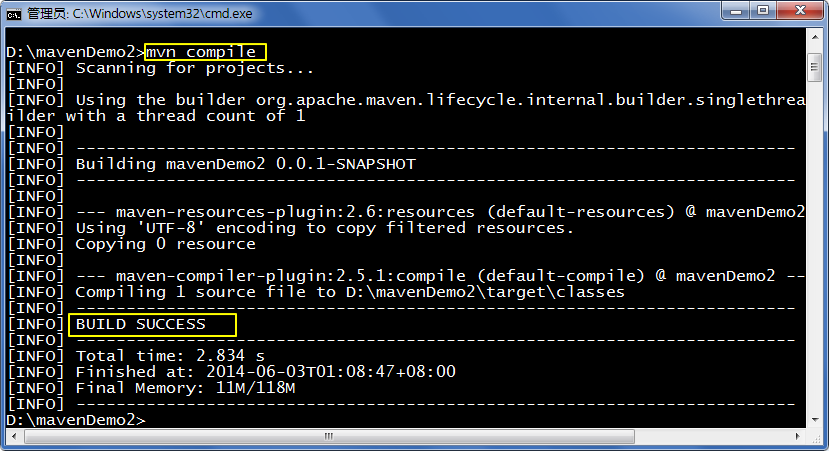
String results = hello.sayHello("coder");

assertEquals("Hello coder!",results);

}

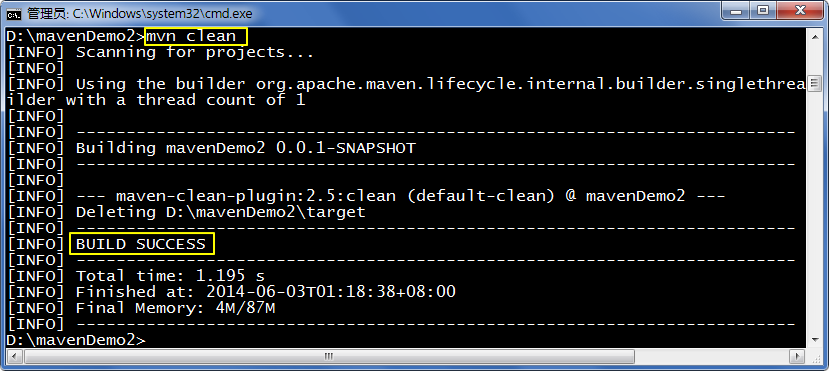
}

1. 编译项目源代码。打开 cmd 窗口，进入 mavenDemo2 目录，执行 “mvn compile” 命令：

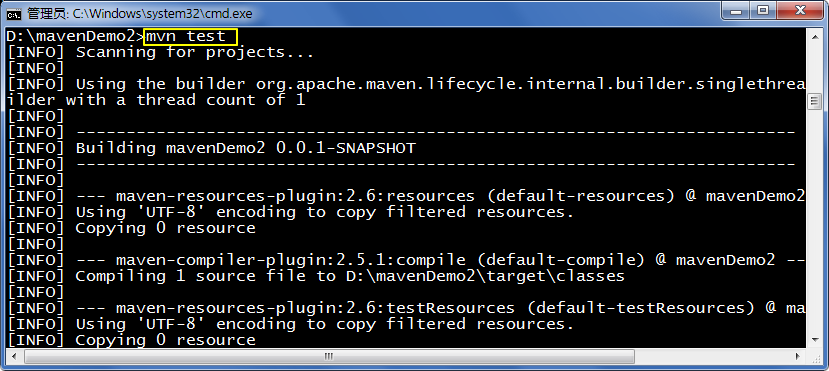


Maven 的各种功能（包括 maven 的命令）都是通过插件的形式来实现的，使用 maven 构建项目，需要联网到 maven [中央仓库](http://repo1.maven.org/maven2/)下载所需插件包，并保存到 maven 仓库，仓库默认位置为：${user.home}/.m2/repository，以后用到就不需要再重新下载。

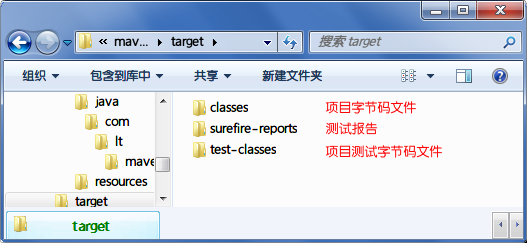
1. 清理项目编译，执行 “mvn clean” 命令，将删除 target 目录：



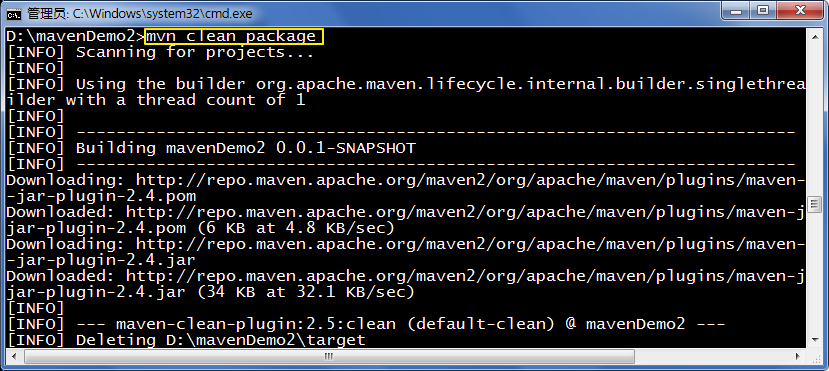
1. 运行测试，执行 “mvn test” 命令：



运行成功，将生成如下文件：

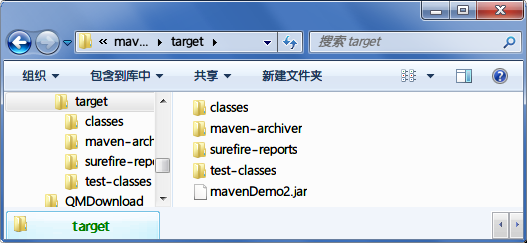


1. 打包项目，执行 “mvn clean package” 组合命令：

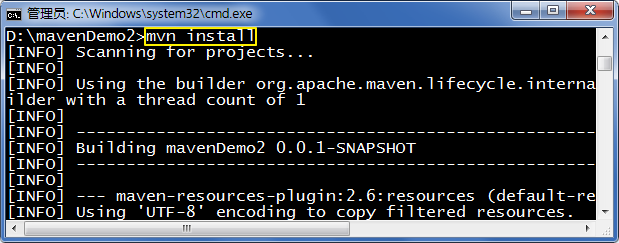


Maven 自动帮我们完成项目的编译、测试、打包。

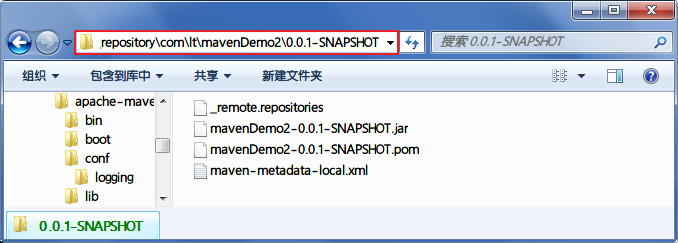
运行成功，将生成如下文件：



1. 执行 “mvn install” 命令：



看到的效果跟mvn clean package命令一样，但是我们打开仓库位置${user.home}/.m2/repository/，可以看到项目被打包发布到了 maven 的仓库，以后其他项目需要依赖到这个项目，就可以通过在 pom.xml 文件中添加依赖来引用：



* maven-metadata-local.xml - 记录了项目元数据，包括 groupId，artifactId，version。
* mavenDemo2-0.0.1-SNAPSHOT.pom - 记录了 mavenDemo2 项目信息以及它的依赖，maven 通过这种方式解决传递依赖，即只要指明了对 mavenDemo2 的依赖，maven 会根据这个文件自动导入 mavenDemo2 对 junit 的依赖。

1. 坐标与依赖
2. 坐标

maven 的所有构件均通过坐标进行组织和管理。maven 的坐标通过 5 个元素进行定义，其中 groupId、artifactId、version 是必须的，packaging 是可选的（默认为jar），classifier 是不能直接定义的。

* **groupId**：定义当前 Maven 项目所属的实际项目，跟 Java 包名类似，通常与域名反向一一对应。
* **artifactId**：定义当前 Maven 项目的一个模块，默认情况下，Maven 生成的构件，其文件名会以 artifactId 开头，如 hibernate-core-3.6.5.Final.jar。
* **version**：定义项目版本。
* **packaging**：定义项目打包方式，如 jar，war，pom，zip ……，默认为 jar。
* **classifier**：定义项 目的附属构件，如 hibernate-core-3.6.6.Final-sources.jar，hibernate-core- 3.6.6.Final-javadoc.jar，其中 sources 和 javadoc 就是这两个附属构件的 classifier。classifier 不能直接定义，通常由附加的插件帮助生成。

1. 使用 Maven 可以方便的管理依赖，如下是一段在 pom.xml 文件中声明依赖的代码示例：

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>3.2.0.RELEASE</version>

<type>jar</type>

<scope>test</scope>

<systemPath>${java.home}/lib/rt.jar</systemPath>

<optional>false</optional>

<exclusions>

<exclusion></exclusion>

</exclusions>

</dependency>

</dependencies>

* type：依赖类型，对应构件中定义的 packaging，可不声明，默认为 jar；
* scope：依赖范围；
* optional：依赖是否可选；
* exclusions：排除传递依赖。

1. 依赖范围

执行不同的 Maven 命令（mvn package，mvn test，mvn install ……），会使用不同的 classpath，Maven 对应的有三套 classpath：编译classpath、测试classpath，运行classpath。scope 选项的值，决定了该依赖构件会被引入到哪一个 classpath 中。

* compile：编译依赖范围，默认值。此选项对编译、测试、运行三种 classpath 都有效，如 hibernate-core-3.6.5.Final.jar，表明在编译、测试、运行的时候都需要该依赖；
* test：测试依赖范围。只对测试有效，表明只在测试的时候需要，在编译和运行时将无法使用该类依赖，如 junit；
* provided：已提供依赖范围。编译和测试有效，运行无效。如 servlet-api ，在项目运行时，tomcat 等容器已经提供，无需 Maven 重复引入；
* runtime：运行时依赖范围。测试和运行有效，编译无效。如 jdbc 驱动实现，编译时只需接口，测试或运行时才需要具体的 jdbc 驱动实现；
* system：系统依赖范围。和 provided 依赖范围一致，需要通过 <systemPath> 显示指定，且可以引用环境变量；
* import：导入依赖范围。使用该选项，通常需要 <type>pom</type>，将目标 pom 的 dependencyManagement 配置导入合并到当前 pom 的  dependencyManagement 元素。

1. 生命周期和插件
2. 三套生命周期

Maven定义了三套生命周期：clean、default、site，每个生命周期都包含了一些阶段（phase）。三套生命周期相互独立，但各个生命 周期中的phase却是有顺序的，且后面的phase依赖于前面的phase。执行某个phase时，其前面的phase会依顺序执行，但不会触发另外两 套生命周期中的任何phase。

1. clean周期
2. pre-clean    ：执行清理前的工作；
3. clean    ：清理上一次构建生成的所有文件；
4. post-clean    ：执行清理后的工作
5. default周期

default生命周期是最核心的，它包含了构建项目时真正需要执行的所有步骤。

1. validate
2. initialize
3. generate-sources
4. process-sources
5. generate-resources
6. process-resources    ：复制和处理资源文件到target目录，准备打包；
7. compile    ：编译项目的源代码；
8. process-classes
9. generate-test-sources
10. process-test-sources
11. generate-test-resources
12. process-test-resources
13. test-compile    ：编译测试源代码；
14. process-test-classes
15. test    ：运行测试代码；
16. prepare-package
17. package    ：打包成jar或者war或者其他格式的分发包；
18. pre-integration-test
19. integration-test
20. post-integration-test
21. verify
22. install    ：将打好的包安装到本地仓库，供其他项目使用；
23. deploy    ：将打好的包安装到远程仓库，供其他项目使用；
24. site周期
25. pre-site
26. site    ：生成项目的站点文档；
27. post-site
28. site-deploy    ：发布生成的站点文档
29. 配置插件

Maven插件高度易扩展，可以方便的进行自定义配置。如：配置maven-compiler-plugin插件编译源代码的JDK版本为1.7：

<plugin>

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

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

<configuration>

<source>1.7</source>

<target>1.7</target>

</configuration>

</plugin>

1. 插件仓库

跟其他构件一样，插件也是根据坐标存储在Maven仓库中。超级POM中Maven配置的默认插件远程仓库如下：

<pluginRepositories>

<pluginRepository>

<id>central</id>

<name>Central Repository</name>

<url>http://repo.maven.apache.org/maven2</url>

<layout>default</layout>

<snapshots>

<enabled>false</enabled>

</snapshots>

<releases>

<updatePolicy>never</updatePolicy>

</releases>

</pluginRepository>

</pluginRepositories>

1. 其他

更多关于Maven细节参考<http://www.cnblogs.com/luotaoyeah/category/584624.html>