

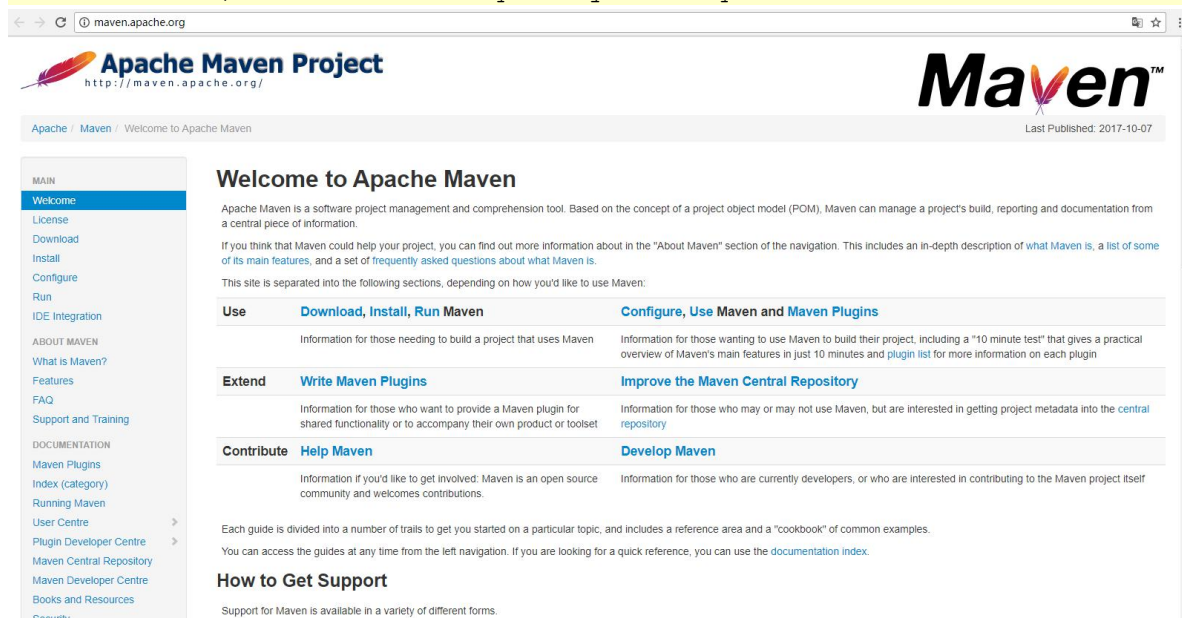
Maven

1 前言

Welcome to Apache Maven

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

If you think that Maven could help your project, you can find out more information about in the "About Maven" section of the navigation. This includes an in-depth description of what Maven is, a list of some of its main features, and a set of frequently asked questions about what Maven is.



Maven 是什么？

Maven 是一个项目管理和综合工具。Maven 提供了开发人员构建一个完整的生命周期框架。开发团队可以自动完成项目的基础工具建设，Maven 使用标准的目录结构和默认构建生命周期。（简单归纳 Maven 负责管理项目开发过程中几乎所有的内容）

在多个开发团队环境时，Maven 可以设置按标准在非常短的时间里完成配置工作。由于大部分项目的设置都很简单，并且可重复使用，Maven 让开发人员的工作更轻松，同时创建报表，检查，构建和测试自动化设置。

Maven 简化和标准化项目建设过程。处理编译，分配，文档，团队协作和其他任务的无缝连接。Maven 增加可重用性并负责建立相关的任务。

Maven 本质上是一个项目管理和理解工具，Maven 提供了开发人员的方式来管理：

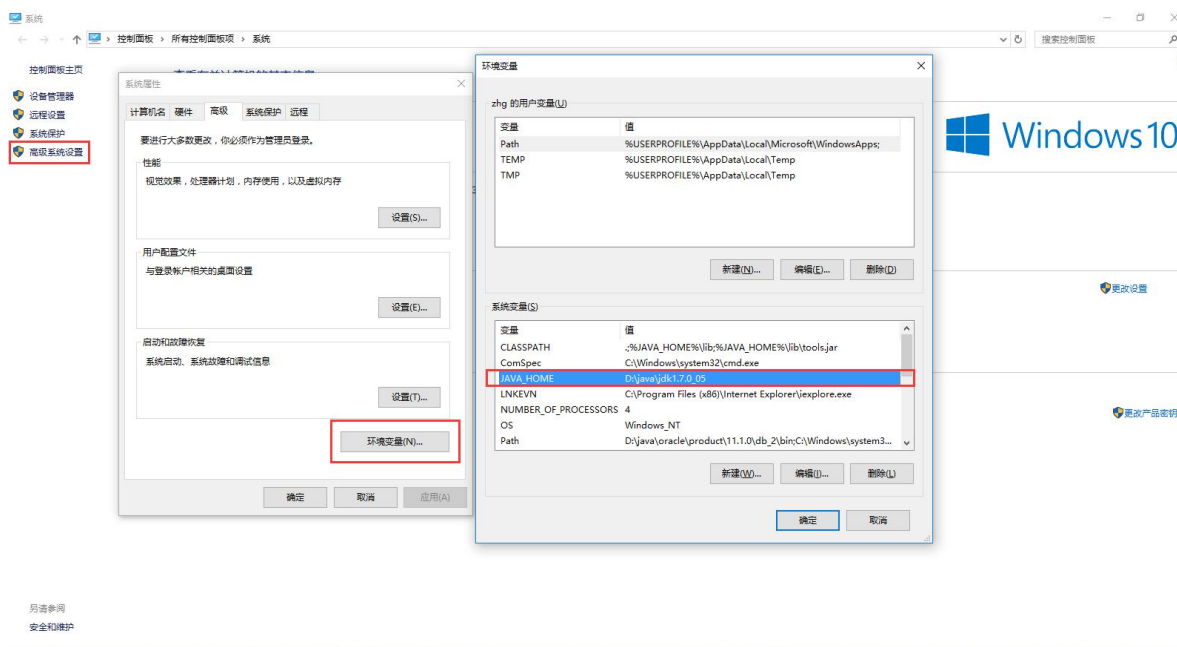
- 构建
- 依赖
- 编译
- 测试
- 打包
- 发布

Maven 的标准工程结构如下：

```
|-- pom.xml (maven的核心配置文件)
|-- src
|   |-- main
|       |-- java (java源代码目录)
|       |-- resources (资源文件目录)
|-- test
|       |-- java (单元测试代码目录)
|-- target (输出目录，所有的输出物都存放在这个目录下)
|   |-- classes (编译后的class文件存放处)
```

2 Maven 安装配置

2.1 JAVA 环境变量



2.2 MAVEN 环境变量

2.2.1 下载解压

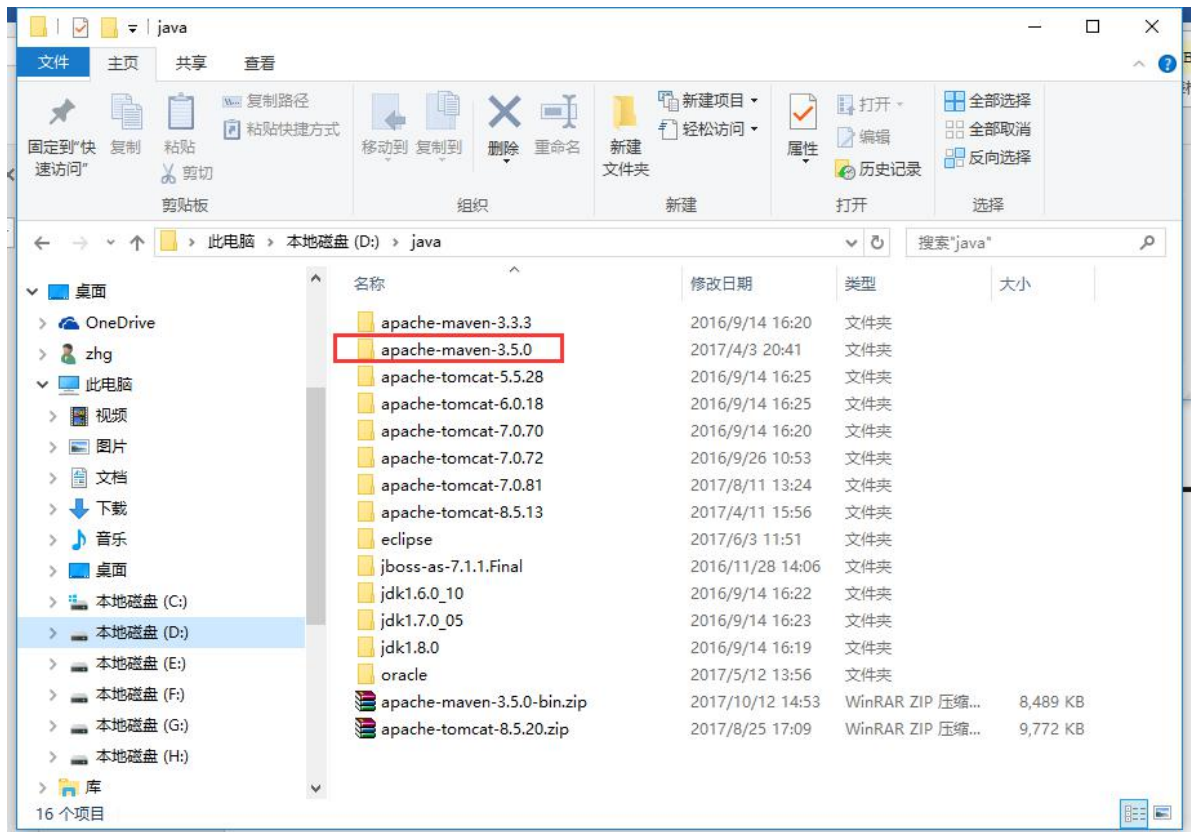
[Maven 官网](#)下载 Maven 安装文件，并解压到指定目录下。

Files

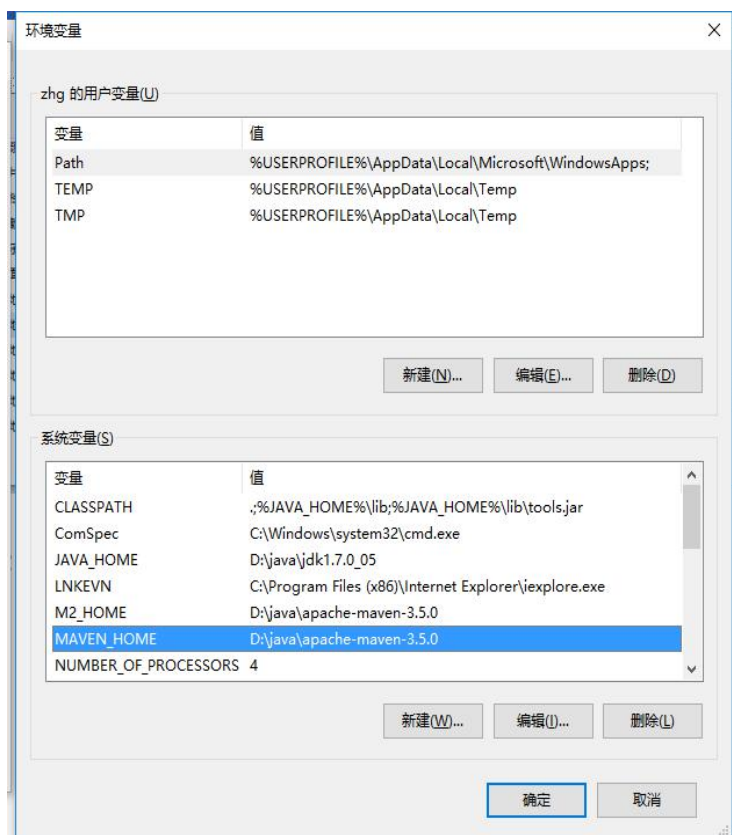
Maven is distributed in several formats for your convenience. Simply pick a ready-made binary distribution archive and follow the [ins](#) yourself.

In order to guard against corrupted downloads/installations, it is highly recommended to [verify the signature](#) of the release bundles a

	Link	Checksum
Binary tar.gz archive	apache-maven-3.5.0-bin.tar.gz	apache-maven-3.5.0-bin.tar.gz.md5
Binary zip archive	apache-maven-3.5.0-bin.zip	apache-maven-3.5.0-bin.zip.md5
Source tar.gz archive	apache-maven-3.5.0-src.tar.gz	apache-maven-3.5.0-src.tar.gz.md5
Source zip archive	apache-maven-3.5.0-src.zip	apache-maven-3.5.0-src.zip.md5



2.2.2 配置环境



注：

在使用 Maven 的过程中，一定要注意 Maven 的版本与 Jdk 的版本相关联。

系统要求

Java开发工具包 (JDK)	<u>Maven 3.3+需要JDK 1.7或更高版本来执行</u> - 它们仍然允许您 通过使用工具链 来构建1.3和其他JDK版本
内存	没有最低要求
硬盘	Maven安装本身需要大约10MB。除此之外，您的本地Maven仓库将使用额外的磁盘空间。本地存储库的大小将根据使用情况而有所不同，但预计至少需要500MB。
操作系统	没有最低要求。启动脚本包含在shell脚本和Windows批处理文件中。

如选择不当的版本后，会出现 Maven 无法使用。

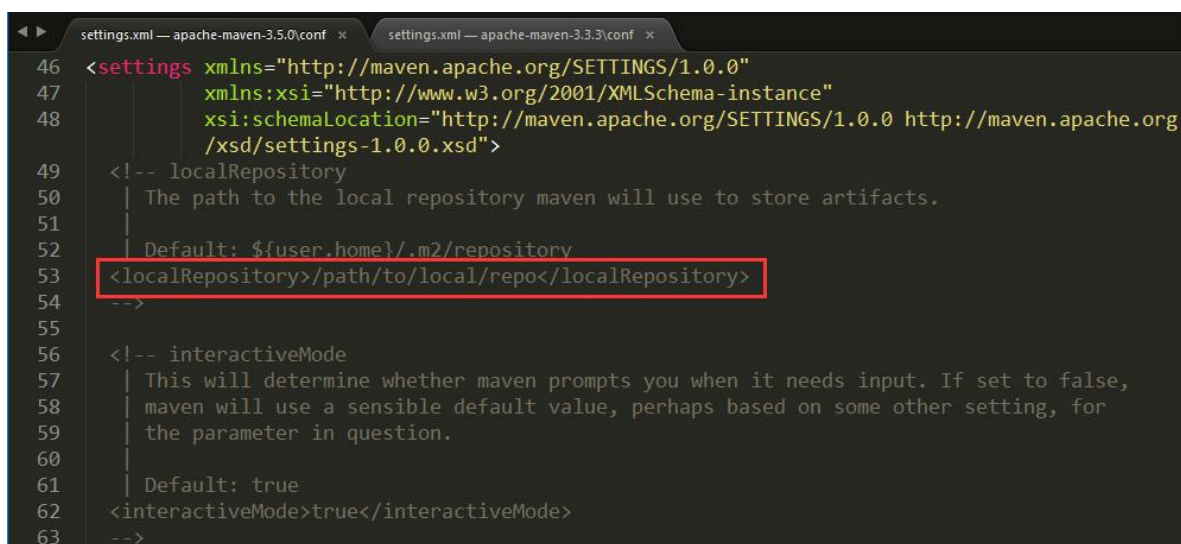
3 Maven 本地资源库

Maven 的本地资源库是用来存储所有项目的依赖关系 (插件 jar 和其他文件, 这些文件被 Maven 下载) 到本地文件夹。很简单, 当你建立一个 Maven 项目, 所有相关文件将被存储在你的 Maven 本地仓库。

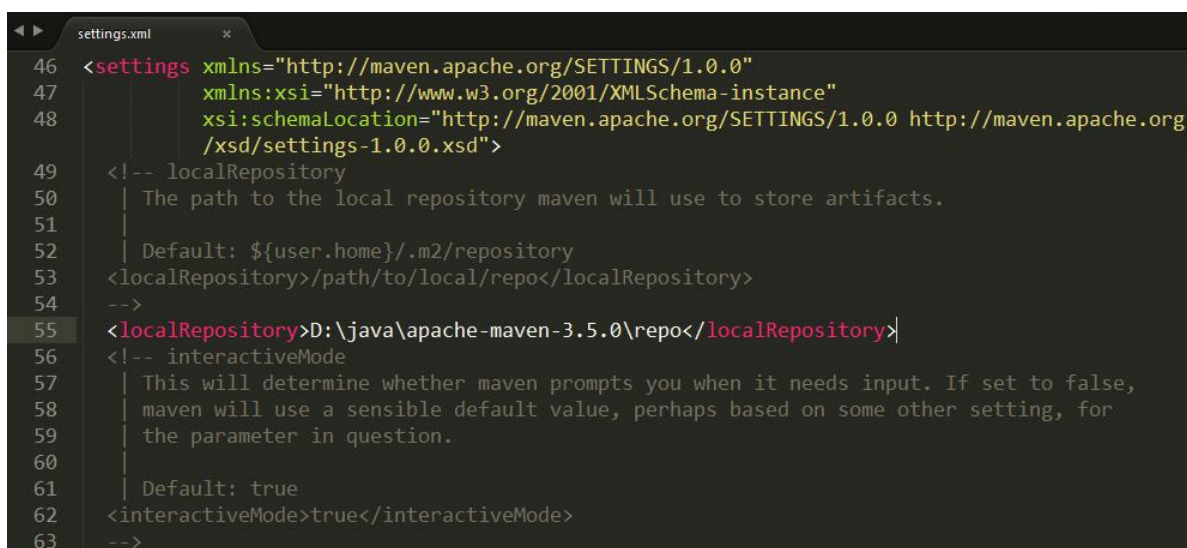
默认情况下, Maven 的本地资源库默认为 .m2 目录文件夹:

1. Linux ~/.m2
2. Windows
 1. XP C:\Documents and Settings\ pcname \.m2
 2. WIN7+ C:\Users\ pcname \.m2

修改 {M2_HOME}/conf/settings.xml



```
46 <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
47     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
48     xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org
49         /xsd/settings-1.0.0.xsd">
50     <!-- localRepository
51         | The path to the local repository maven will use to store artifacts.
52         | Default: ${user.home}/.m2/repository
53     <localRepository>path/to/local/repo</localRepository>
54     -->
55
56     <!-- interactiveMode
57         | This will determine whether maven prompts you when it needs input. If set to false,
58         | maven will use a sensible default value, perhaps based on some other setting, for
59         | the parameter in question.
60         | Default: true
61     <interactiveMode>true</interactiveMode>
62
63     -->
```

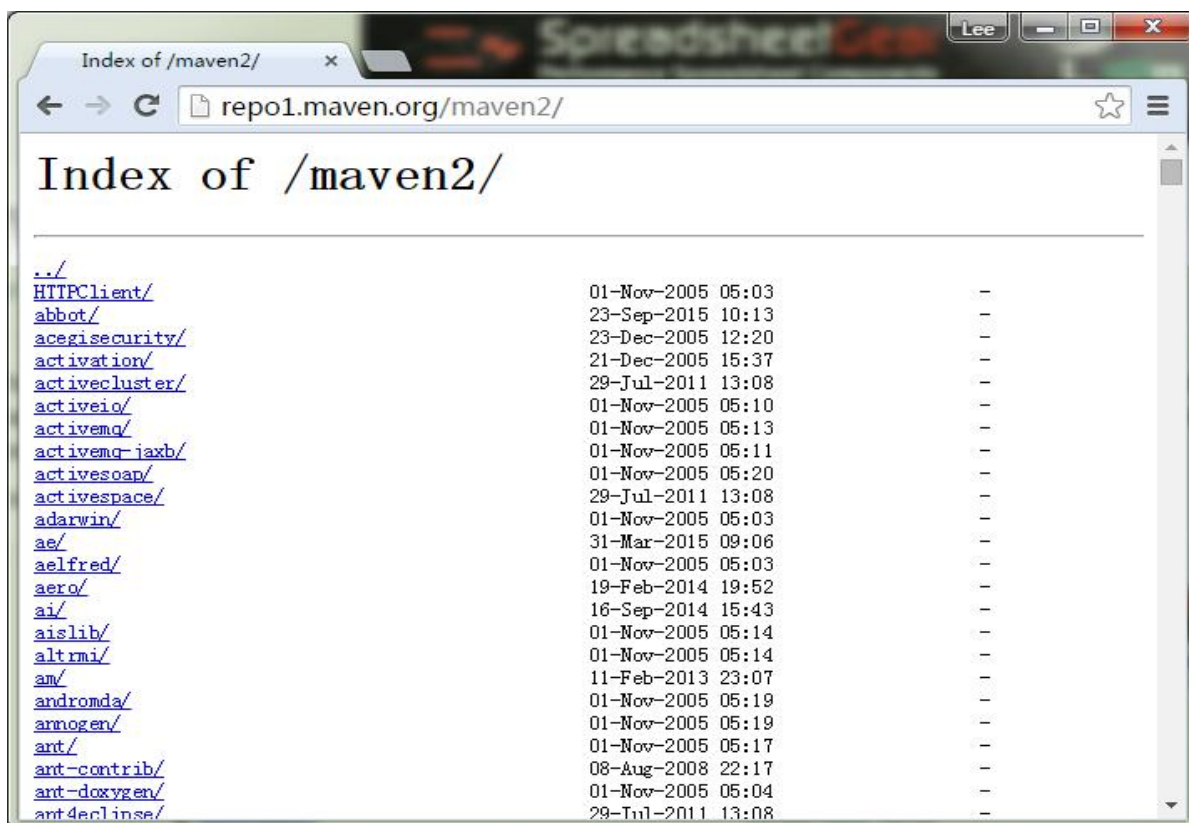


```
46 <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
47     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
48     xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org
49         /xsd/settings-1.0.0.xsd">
50     <!-- localRepository
51         | The path to the local repository maven will use to store artifacts.
52         | Default: ${user.home}/.m2/repository
53     <localRepository>path/to/local/repo</localRepository>
54     -->
55     <localRepository>D:\java\apache-maven-3.5.0\repo</localRepository>
56     <!-- interactiveMode
57         | This will determine whether maven prompts you when it needs input. If set to false,
58         | maven will use a sensible default value, perhaps based on some other setting, for
59         | the parameter in question.
60         | Default: true
61     <interactiveMode>true</interactiveMode>
62
63     -->
```

4 Maven 远程资源库

默认情况下, 当你建立一个 Maven 项目时, Maven 会检查 setting.xml, pom.xml 根据文件中配置以便于确定哪些需要从本地资源库加载, 哪些需要从远程资源库 (中央资源库) 下载。

首先, Maven 将从本地资源库获得 Maven 的本地资源库依赖资源, 如果没有找到, 然后把它会从默认的 Maven 中央存储库 <http://repo1.maven.org/maven2/> 查找下载。



Maven 中心储存库网站已经改版本, 目录浏览可能不再使用。这将直接被重定向到 <http://search.maven.org/>。

在 Maven 中, 当你声明的库不存在于本地存储库中, 也没有不存在于 Maven 中心储存库, 该过程将停止并将错误消息输出到 Maven 控制台。

例如:

```
pom.xml 配置
<dependency>
    <groupId>org.jvnet.localizer</groupId>
    <artifactId>localizer</artifactId>
    <version>1.8</version>
</dependency>
```

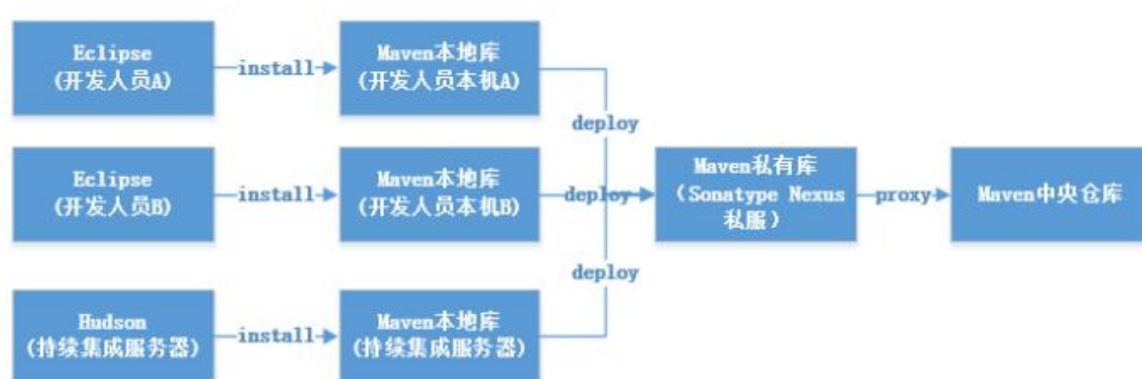

当你建立这个 **Maven** 项目，它将依赖找不到失败并输出错误消息。

```
pom.xml
<repositories>
  <repository>
    <id>java.net</id>
    <url>https://maven.java.net/content/repositories/public/</url>
  </repository>
</repositories>
```

Maven 依赖库查找顺序：

- 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。
- 在 **Maven** 中央存储库搜索，如果没有找到，进入第 3 步，否则退出。
- 在指定的远程存储库搜索，如果没有找到，提示错误信息，否则退出。

5 Maven 私服



项目开发过程中所需的所有构件都需要通过 maven 的中央仓库和第三方的 Maven 仓库下载到本地，而一个团队中的所有人都重复的从 maven 仓库下载构件无疑加大了仓库的负载和浪费了外网带宽，如果网速慢的话，还会影响项目的进程。很多情况下项目的开发都是在内网进行的，连接不到 maven 仓库这时便需要搭建属于自己的 maven 私服，这样既节省了网络带宽也会加速项目搭建的进程，当然前提条件就是你的私服可以连接外网。

5.1 基于 Maven 全局设置

```
<mirrors>
  <!-- mirror
  | Specifies a repository mirror site to use instead of a given repository. The repository that
  | this mirror serves has an ID that matches the mirrorOf element of this mirror. IDs are used
  | for inheritance and direct lookup purposes, and must be unique across the set of mirrors.
  |
  <mirror>
    <id>mirrorId</id>
    <mirrorOf>repositoryId</mirrorOf>
    <name>Human Readable Name for this Mirror.</name>
    <url>http://my.repository.com/repo/path</url>
  </mirror>
-->

<!-- 本机可以连接外网 -->
<mirror>
  <id>alimaven</id>
  <mirrorOf>central</mirrorOf>
  <name>aliyun maven</name>
  <url>http://maven.aliyun.com/nexus/content/repositories/central/</url>
</mirror>
```

```
<!-- 本机不可以连接外网 -->
<mirror>
  <id>javakcmaven</id>
  <mirrorOf>central</mirrorOf>
  <name>javakc maven</name>
  <url>http://192.168.1.99:8081/nexus/content/repositories/central/</url>
</mirror>

</mirrors>
```

5.2 基于 Project 单独设置

私服地址: <http://192.168.1.99:8081/nexus>

pom.xml 中添加私服配置

```
<!-- 配置 maven 私服服务器 start -->
<repositories>
  <repository>
    <id>nexus</id>
    <name>Team Nexus Repository</name>
    <url>http://192.168.1.99:8081/nexus/content/groups/public/</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>nexus</id>
    <name>Team Nexus Repository</name>
    <url>http://192.168.1.99:8081/nexus/content/groups/public/</url>
  </pluginRepository>
</pluginRepositories>
<!-- 配置 maven 私服服务器 end -->
```

6 Maven 常用命令

6.1 创建 Java 项目

```
mvn archetype:create  
    -DgroupId=com.javakc  
    -DartifactId=ssm
```

6.2 创建 Web 项目

```
mvn archetype:create  
    -DgroupId=com.javakc  
    -DartifactId=ssm  
    -DarchetypeArtifactId=maven-archetype-webapp
```

6.3 自动构建项目

```
mvn archetype:generate
```

6.4 编译源代码

```
mvn compile
```

6.5 测试源代码

```
mvn test  
测试执行Junit代码
```

6.6 编译测试代码

```
mvn test-compile
```

6.7 项目打包

```
mvn package  
项目打包jar/war，生成到target目录中
```

6.8 清理项目

```
mvn clean  
清零项目中的临时文件，一般指向target目录中内容
```

6.9 生成 eclipse 项目

```
mvn eclipse:eclipse    生成eclipse环境
```

```
mvn eclipse:clean      删除eclipse环境
```

6.10 生成 idea 项目

```
mvn idea:idea
```

6.11 上传 jar 到 maven

```
mvn install:install-file  
    -Dfile=D:\oracle.jar  
    -DgroupId=com.oracle  
    -DartifactId=ojdbc5  
    -Dversion=10.0.2.1  
    -Dpackaging=jar
```

例如:

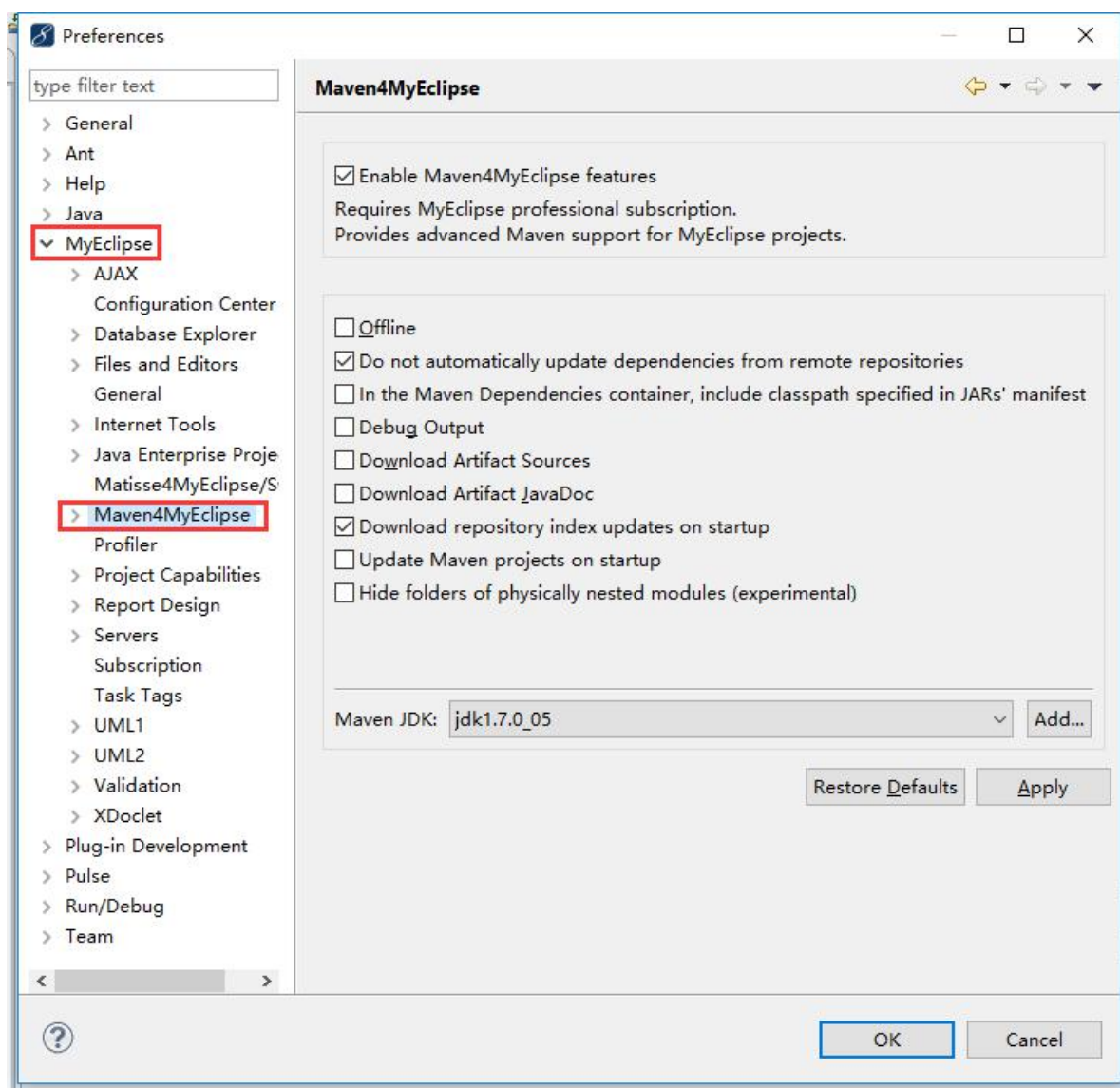
```
mvn
```

```
install:install-file -Dfile=D:\ojdbc5.jar -DgroupId=com.oracle -DartifactId=ojdbc5 -Dversion=11.1.0.5 -Dpackaging=jar
```

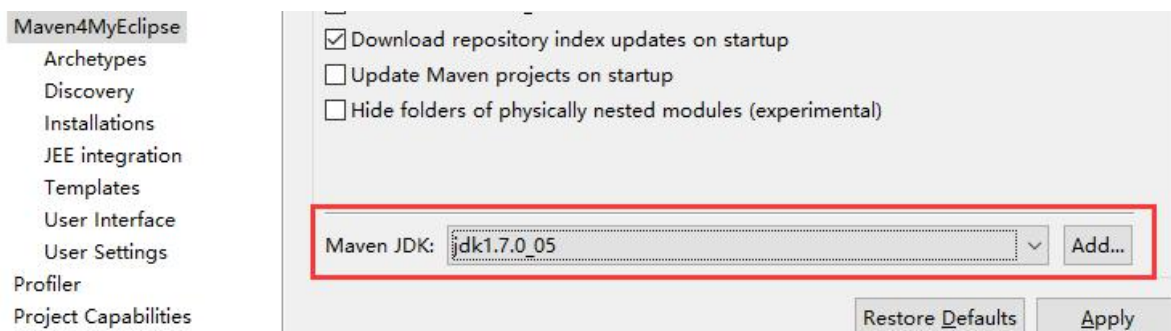
7 Maven 与 IDE 集成

7.1 Eclipse 集成

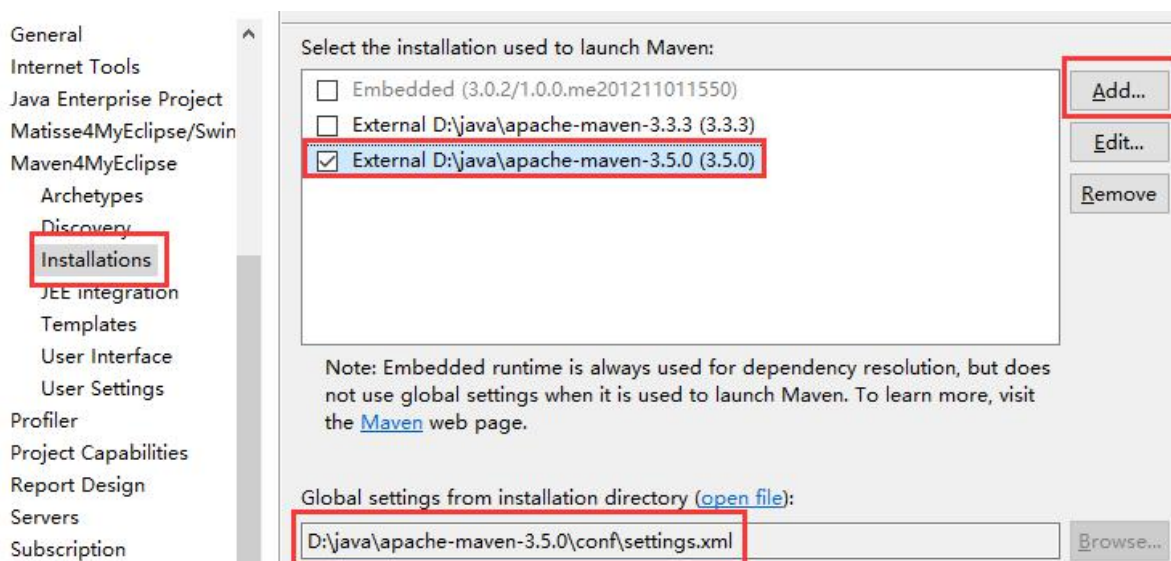
打开 Eclipse 后, Windows→Preferences→MyEclipse→Maven4MyEclipse



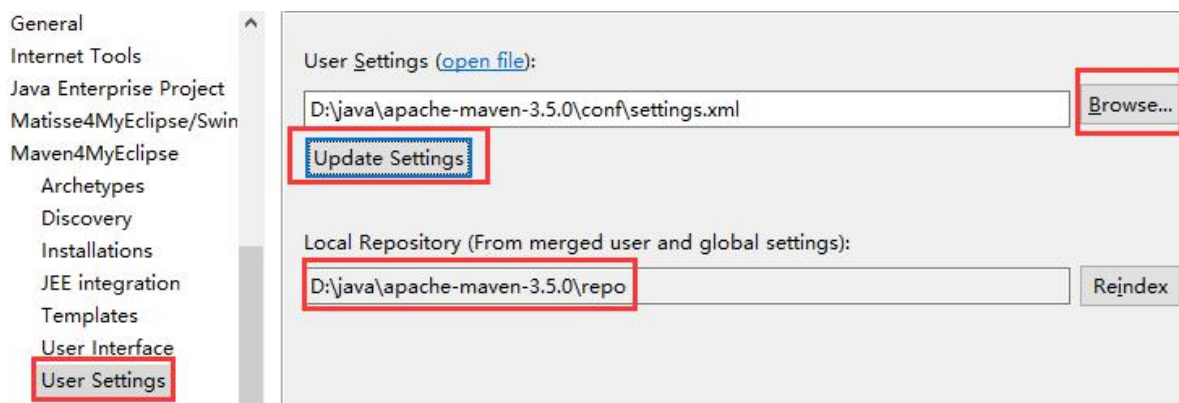
1. 配置 Maven JDK, 不要使用默认的 JDK, 选择自己的.



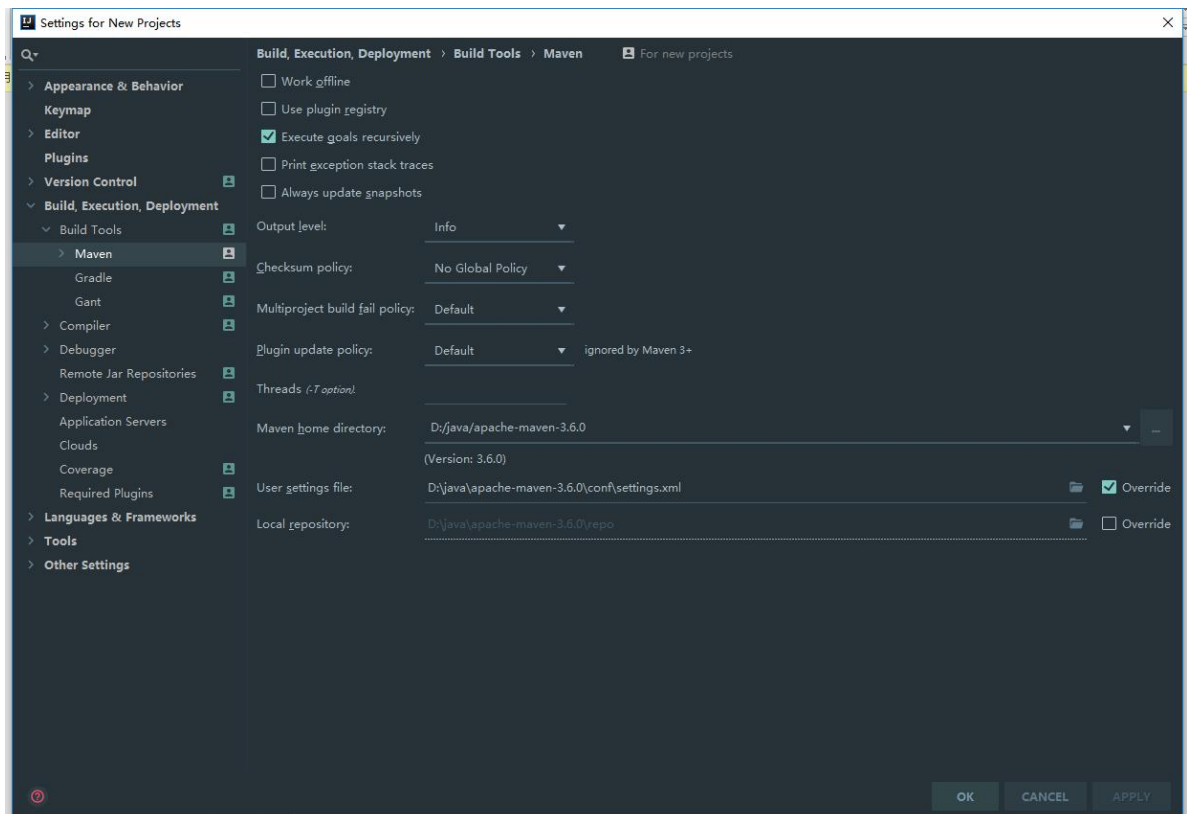
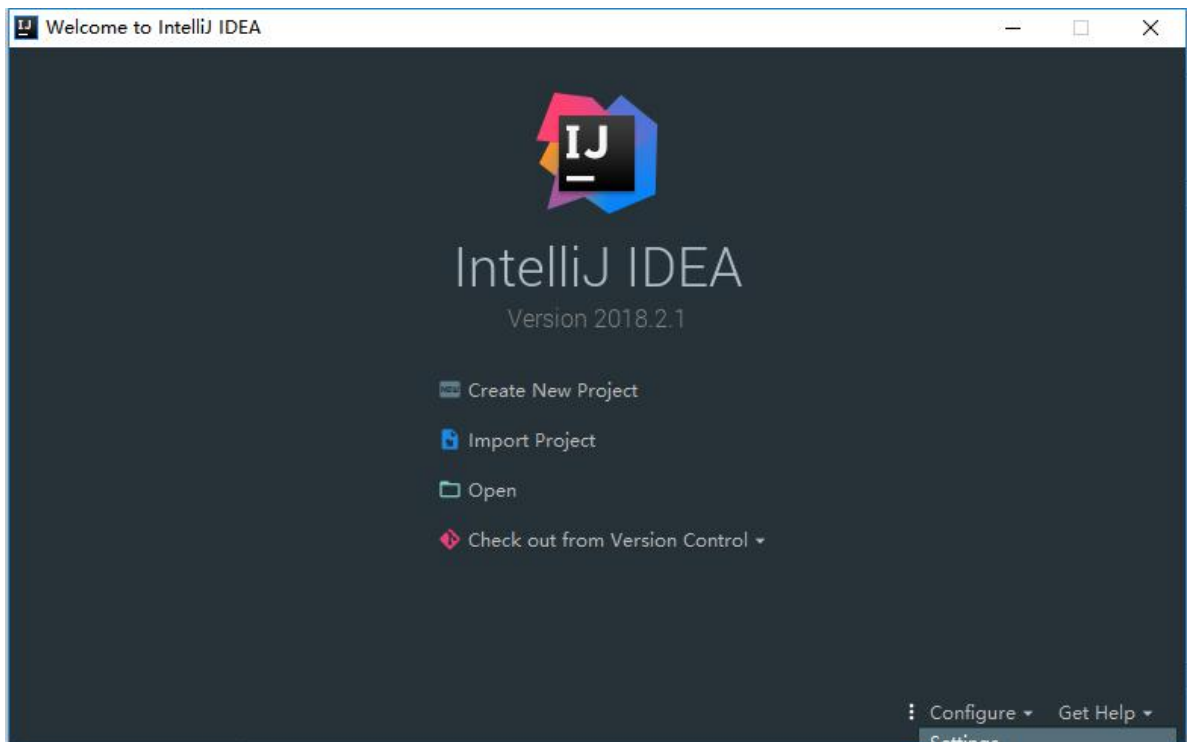
2. 选择 Maven 安装目录



3. 配置 Maven 资源库下载位置 (注:选择自己的 Maven 配置文件 setting.xml)



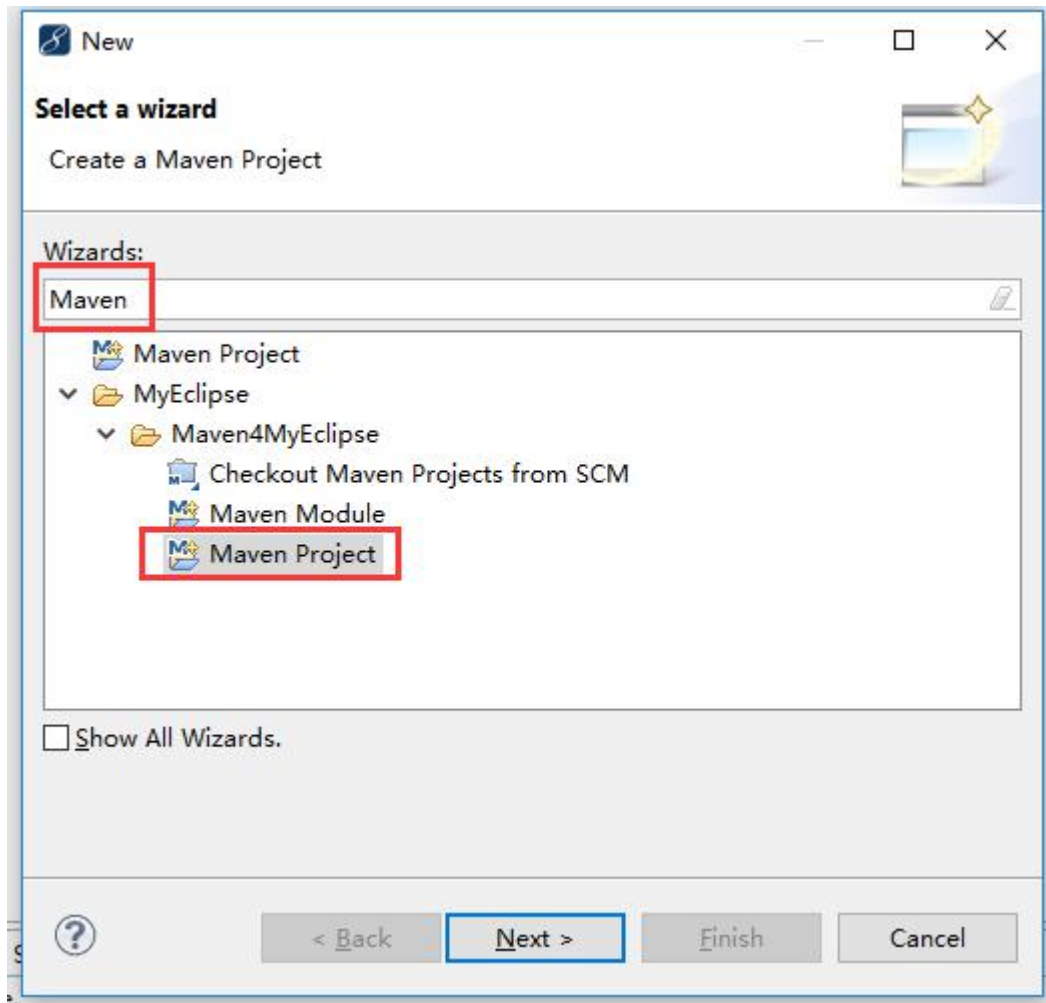
7.2 IDEA 集成



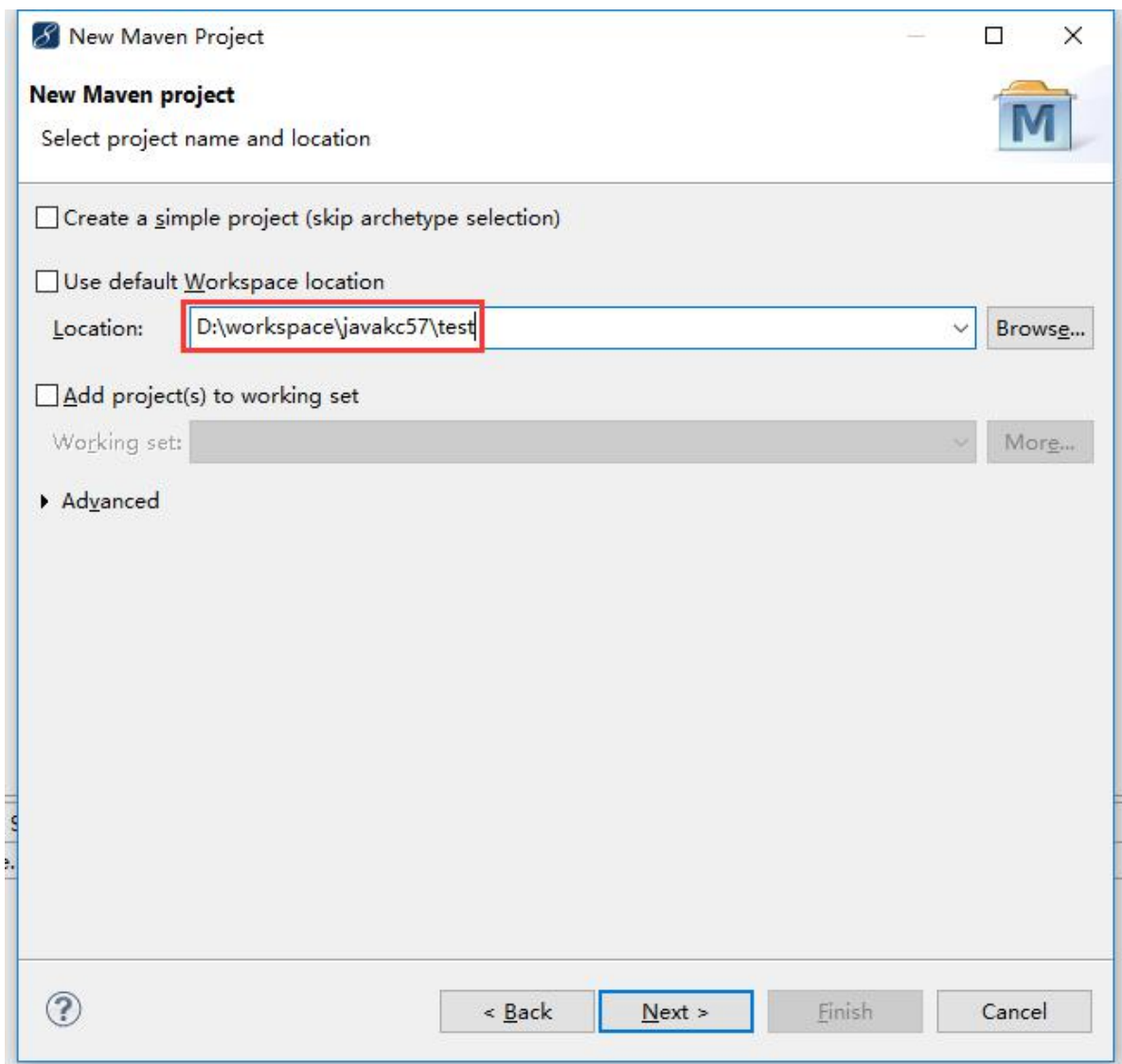
8 Maven 创建 Web 项目

8.1 创建项目

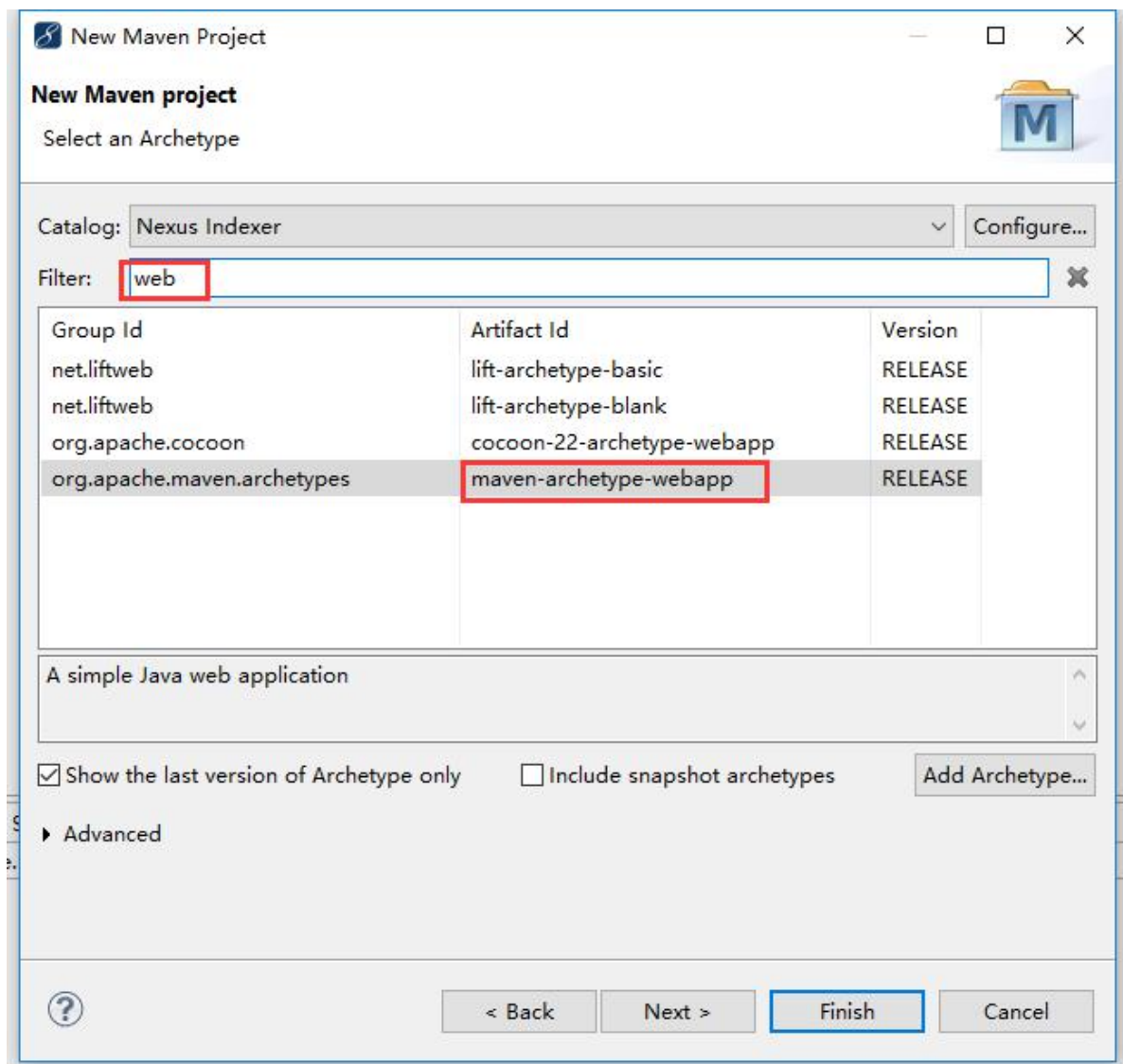
1. 首先 Eclipse 中右键 New，输入 Maven，选择 Maven Project，Next 下一步。



2. 选择 Maven 项目工作空间地址， Next 下一步



3. 过滤 Maven 模板，选择 Web 的 Maven 模板下一步。



4. 输入 Maven Web 项目相关信息

New Maven Project

New Maven project 2

Specify Archetype parameters

Group Id:

Artifact Id:

Version:

Package:

Properties available from archetype:

Name	Value

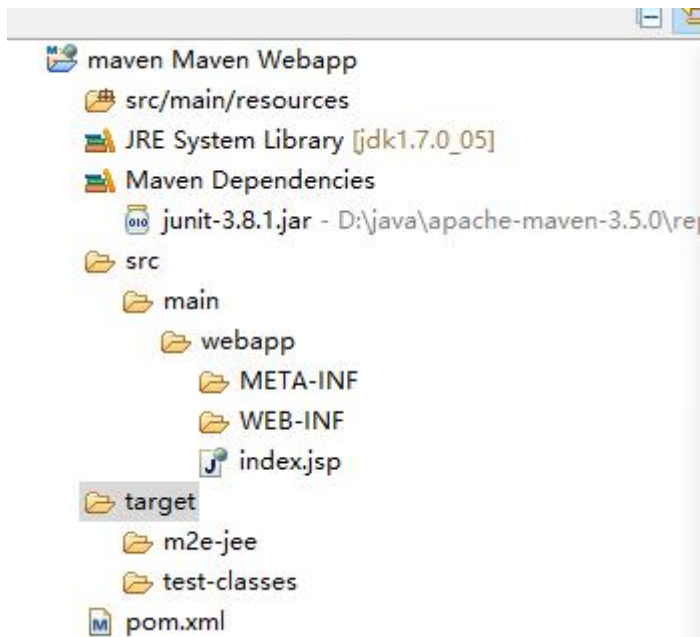
► Advanced

Group Id: 项目名称

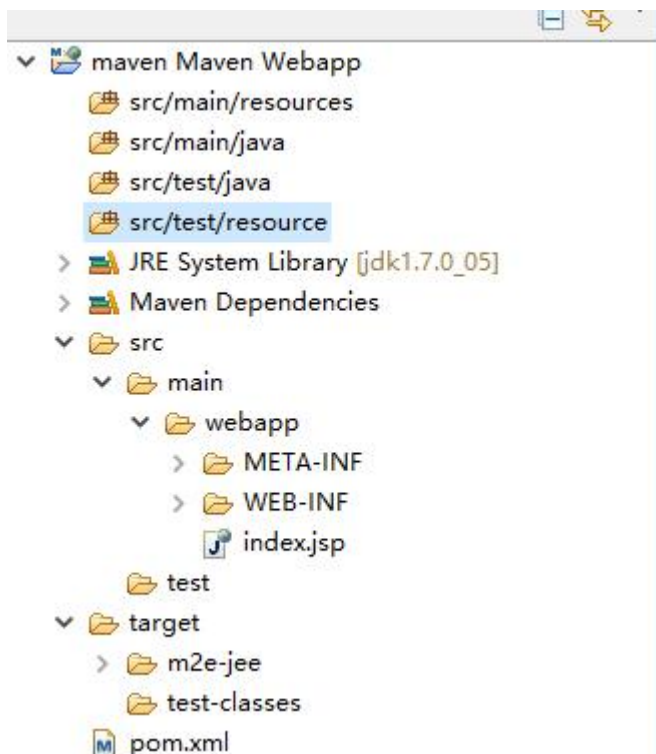
Artifact Id: 项目组织信息

Version: 项目版本号

Package: 包路径 (Artifact+Group 构成)



创建后的目录结构。



标准目录，使用 Maven 创建 Web 项目后没有的目录则手动添加。

5. 为 Web 项目引入相关 jar 包

Maven 项目创建完毕后，标准的 pom.xml 文件内容。当我们需要使用一些相关技术来去完善项目是，则需要修改 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/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.javakc</groupId>
    <artifactId>maven</artifactId>
    <packaging>war</packaging>
    <version>0.0.1-SNAPSHOT</version>
    <name>maven 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>maven</finalName>
    </build>

</project>
```

例如当我们需要在项目中集成 SpringMvc 是，修改 pom.xml 文件添加 spring-mvc 的相关配置。

```
<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.javakc</groupId>
```

```

<artifactId>maven</artifactId>
<packaging>war</packaging>
<version>0.0.1-SNAPSHOT</version>
<name>maven Maven Webapp</name>
<url>http://maven.apache.org</url>

<dependencies>

    <!-- springMvc start -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>4.0.2.RELEASE</version>
    </dependency>
    <!-- springMvc end -->

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <finalName>maven</finalName>
</build>

</project>

```

保存后，Maven 立即检索并引入相关 jar 到项目中。

```

> JRE System Library [jdk1.7.0_05]
v Maven Dependencies
  > spring-webmvc-4.0.2.RELEASE.jar - D:\java\apache-maven-3.5.0\repo\org\springframework\spring-webmvc\4.0.2.RELEASE
  > spring-beans-4.0.2.RELEASE.jar - D:\java\apache-maven-3.5.0\repo\org\springframework\spring-beans\4.0.2.RELEASE
  > spring-context-4.0.2.RELEASE.jar - D:\java\apache-maven-3.5.0\repo\org\springframework\spring-context\4.0.2.RELEASE
  > spring-aop-4.0.2.RELEASE.jar - D:\java\apache-maven-3.5.0\repo\org\springframework\spring-aop\4.0.2.RELEASE
  > aopalliance-1.0.jar - D:\java\apache-maven-3.5.0\repo\org\springframework\spring-aop\4.0.2.RELEASE
  > spring-core-4.0.2.RELEASE.jar - D:\java\apache-maven-3.5.0\repo\org\springframework\spring-core\4.0.2.RELEASE
  > commons-logging-1.1.3.jar - D:\java\apache-maven-3.5.0\repo\commons-logging\commons-logging\1.1.3
  > spring-expression-4.0.2.RELEASE.jar - D:\java\apache-maven-3.5.0\repo\org\springframework\spring-expression\4.0.2.RELEASE
  > spring-web-4.0.2.RELEASE.jar - D:\java\apache-maven-3.5.0\repo\org\springframework\spring-web\4.0.2.RELEASE
  > junit-3.8.1.jar - D:\java\apache-maven-3.5.0\repo\org\springframework\spring-web\4.0.2.RELEASE
  > junit-3.8.1.jar - D:\java\apache-maven-3.5.0\repo\org\springframework\spring-web\4.0.2.RELEASE
  > src
  > target
  pom.xml

```

再例如我需要在项目中使用 Mybatis 技术。

```
<dependencies>

    <!-- mybatis start -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.2.8</version>
    </dependency>
    <!-- mybatis end -->

</dependencies>
```

项目中便有 Maven 自动引入相应 jar 包。



接下来我们看一下关于一个 ssm 项目 Maven 是如何配置的。

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.javakc</groupId>
    <artifactId>maven</artifactId>
    <packaging>war</packaging>
    <version>0.0.1-SNAPSHOT</version>
    <name>maven Maven Webapp</name>
    <url>http://maven.apache.org</url>

    <!-- 用来设置引入 jar 包版本号 start -->
    <properties>
```

```
<!-- spring 版本号 -->
<spring.version>4.3.12.RELEASE</spring.version>
<!-- mybatis 版本号 -->
<mybatis.version>3.4.5</mybatis.version>
<!-- mybatis 整合 spring 包 -->
<mybatis-spring.version>1.3.1</mybatis-spring.version>
<!-- druid 数据源 -->
<druid.version>1.1.5</druid.version>
<!-- 驱动包版本 -->
<oracle.version>10.2.0.4.0</oracle.version>
<!-- 日志文件管理包版本 -->
<slf4j.version>1.7.7</slf4j.version>
<log4j.version>1.2.17</log4j.version>
</properties>
<!-- 用来设置引入 jar 包版本号 end -->

<dependencies>

<!-- spring 核心包 start-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-oxm</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${spring.version}</version>
</dependency>
```

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${spring.version}</version>
</dependency>
<!-- spring 核心包 end-->

<!-- mybatis/spring 核心包 start-->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>${mybatis.version}</version>
</dependency>
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>${mybatis-spring.version}</version>
</dependency>
<!-- mybatis/spring 核心包 end-->
<!-- 数据库连接池 start -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>${druid.version}</version>
</dependency>
<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc14</artifactId>
    <version>${oracle.version}</version>
```



```
</dependency>
<!-- 数据库连接池 end -->

<!-- 日志包导入 start-->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>${log4j.version}</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>${slf4j.version}</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>${slf4j.version}</version>
</dependency>
<!-- 日志包导入 end-->

<!-- springmvc-json 解析 start -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.9.2</version>
</dependency>
<!-- springmvc-json 解析 start -->

<!-- apache 工具类包 start -->
<dependency>
    <groupId>commons-dbcp</groupId>
    <artifactId>commons-dbcp</artifactId>
    <version>1.2.2</version>
</dependency>
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.1</version>
</dependency>
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.4</version>
```

```
</dependency>
<dependency>
  <groupId>commons-codec</groupId>
  <artifactId>commons-codec</artifactId>
  <version>1.9</version>
</dependency>
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.3.2</version>
</dependency>
<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi</artifactId>
  <version>3.14</version>
</dependency>
<!-- apache 工具类包 end -->

<!-- servlet 组件支持 start -->
<dependency>
  <groupId>javax</groupId>
  <artifactId>javace-api</artifactId>
  <version>7.0</version>
</dependency>
<!-- JSTL 标签类 -->
<dependency>
  <groupId>jstl</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
<!-- servlet 组件支持 end -->

<!-- 单元测试工具类 start -->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>3.8.1</version>
  <scope>test</scope>
</dependency>
<!-- 单元测试工具类 end -->

</dependencies>

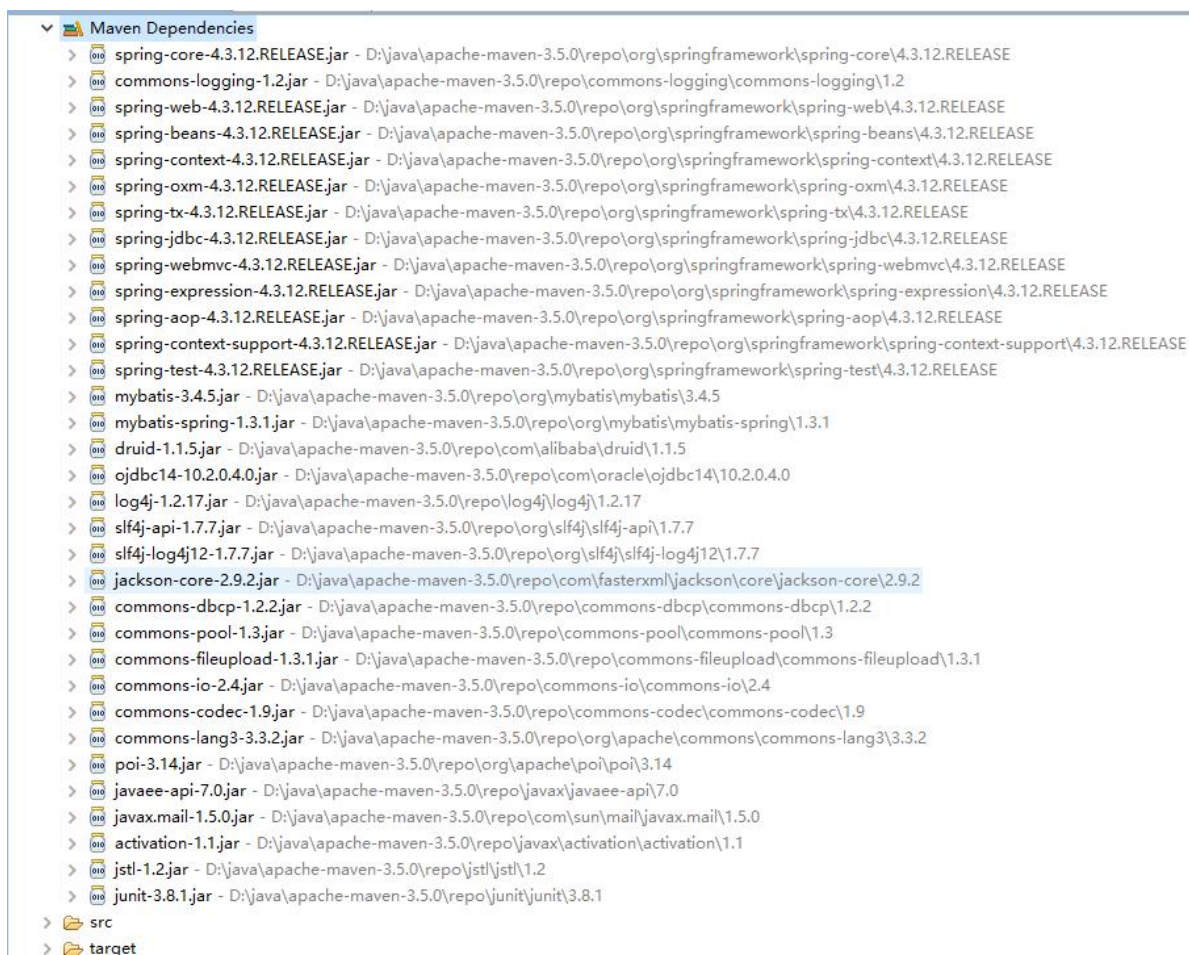
<build>
```

```
<finalName>maven</finalName>

</build>

</project>
```

上述 pom.xml 配置后由 Maven 自动构建的项目 jar。



maven 官网网址搜索需要用的 jar 包

<http://mvnrepository.com>

私服 maven nexus 搜索使用

<http://192.168.1.99:8081/nexus>

8.2 集成 tomcat

pom.xml 引入 tomcat 插件依赖

```
<plugins>
  <!-- tomcat 插件控制 -->
  <plugin>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <version>2.2</version>
    <configuration>
      <!-- 配置 tomcat 端口号 -->
      <port>8080</port>
      <!-- 配置 tomcat 项目名 -->
      <path>/test</path>
      <!-- 配置编码格式 -->
      <uriEncoding>UTF-8</uriEncoding>
    </configuration>
  </plugin>
  <!-- maven 插件控制 -->
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
      <source>1.8</source>
      <target>1.8</target>
      <encoding>utf-8</encoding>
    </configuration>
  </plugin>
</plugins>
```

9 Maven pom 配置

POM 代表项目对象模型。它是 Maven 中工作的基本单位，这是一个 XML 文件。它始终保存在该项目基本目录中的 pom.xml 文件。

POM 包含的项目是使用 Maven 来构建的，它用来包含各种配置信息。

POM 也包含了目标和插件。在执行任务或目标时，Maven 会使用当前目录中的 POM。它读取 POM 得到所需要的配置信息，然后执行目标。部分的配置可以在 POM 使用如下：

```
<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>
  <!-- 项目组织唯一名称 -->
  <groupId>com.javakc</groupId>
  <!-- 项目名称 -->
  <artifactId>maven</artifactId>
  <!-- 打包形式 war,jar 等 -->
  <packaging>war</packaging>
  <!-- 项目版本号 -->
  <version>0.0.1-SNAPSHOT</version>
  <name>maven Maven Webapp</name>
  <url>http://maven.apache.org</url>

  <!-- 项目 jar 引入管理 -->
  <dependencies>
    <!-- 引入指定的 jar 包 -->
    <dependency>
      <groupId>junit</groupId><!-- junit 组织唯一名称 -->
      <artifactId>junit</artifactId><!-- junit 项目名称 -->
      <version>3.8.1</version><!-- junit 版本号，不写自动下载最新版本 -->
      <scope>test</scope><!-- 标示仅在测试使用，打包不引入 -->
    </dependency>
  </dependencies>

  <!-- 构建项目的相关配置 -->
  <build>
    <finalName>maven</finalName><!-- 构建时的项目名称 -->
    <directory>${basedir}/target</directory><!-- 构建时项目存储路径 -->
  </build>

</project>
```

```
<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.javakc.app 生成的相对路径为：com/javakc/app -->

```
<groupId>com.javakc</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>app</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>maven</name>
```

<!-- 项目主页的 URL，Maven 产生的文档用 -->

```
<url>http://www.javakc.com/app</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 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/banseon</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>javakc@163com</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>javakc@163.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>javakc@163.com</unsubscribe>
    <!-- 你可以浏览邮件信息的 URL -->
    <archive>http://www.javakc.com/com/javakc/app/</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>javakc</name>
    <!-- 项目开发者的 email -->
    <email>javakc@163.com</email>
    <!-- 项目开发者的主页的 URL -->
    <url />
    <!-- 项目开发者在项目中扮演的角色, 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://www.javakc.com/</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>-8</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.javakc.com/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。 -->
<!-- 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.javakc.com/javakc</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
            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/javakc/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/javakc/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 就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>
```


10 pom.xml 详解

10.1 常用 jar 包

servlet-api

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
```

```
</dependency>
```

jsp-api

```
<dependency>
  <groupId>javax.servlet.jsp</groupId>
  <artifactId>jsp-api</artifactId>
  <version>2.2</version>
  <scope>provided</scope>
</dependency>
```

gson

```
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.8.5</version>
</dependency>
```

10.2 scope 标签

1. **compile**: 默认值 他表示被依赖项目需要参与当前项目的编译，还有后续的测试，运行周期也参与其中，是一个比较强的依赖。打包的时候通常需要包含进去

2. **test**: 依赖项目仅仅参与测试相关的工作，包括测试代码的编译和执行，不会被打包，例如: junit

3. **runtime**: 表示被依赖项目无需参与项目的编译，不过后期的测试和运行周期需要其参与。与 compile 相比，跳过了编译而已。例如 JDBC 驱动，适用运行和测试阶段

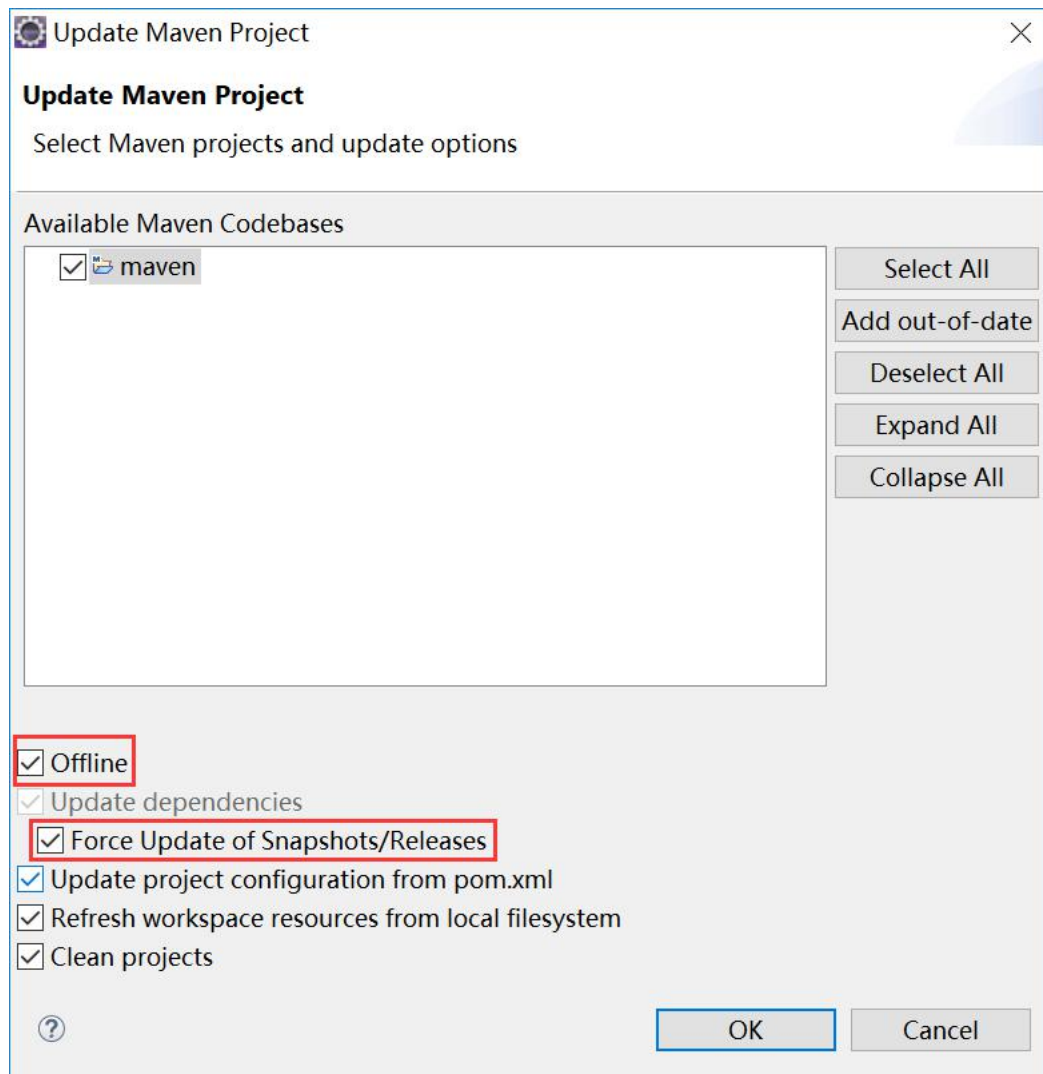
4. **provided**: 打包的时候可以不用包进去，别的设施会提供。事实上该依赖理论上可以参与编译，测试，运行等周期。相当于 compile，但是打包阶段做了 exclude 操作

5. **system**: 从参与度来说，和 provided 相同，不过被依赖项不会从 maven 仓库下载，而是从本地文件系统拿。需要添加 systemPath 的属性来定义路径

11 Maven 常见问题

11.1 pom.xml 文件报错

- 1.选择 Update Maven Project, 勾选 Offline, Force Update of...



3. 从本地 Maven 存储中找到该 jar，删除后重新更新。

11.2 web 项目版本

maven 创建 webapp 项目默认的 jdk 为 1.5，web 版本为 2.3。

1. 修改 maven 配置文件

```
<profile>
  <id>jdk-1.8</id>
```

```
<activation>
  <activeByDefault>true</activeByDefault>
  <jdk>1.8</jdk>
</activation>
<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
<maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
</properties>
</profile>
```

通过以上配置，设置 maven 构建项目是选择 jdk 版本为 1.8

2. 修改项目的配置文件

找到当前项目的目录的 .settings 文件夹，找到 .settings 文件夹下面的一个配置文件 org.eclipse.wst.common.project.facet.core.xml 修改该文件

修改前：

```
<installed facet="jst.web" version="2.3"/>
```

修改后：

```
<installed facet="jst.web" version="3.1"/>
```

通过上述两个配置，项目 update 更新 maven 即可！

11.3 java.util.ZipException

到 maven 的 repository 目录下搜 aether*****in-progress (可以搜 aether 或者 in-progress 都行) 文件，如果存在，把这个文件对应的版本目录删除，刷新一下项目重新部署打包即可。