Maven教程

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。 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安装和配置

windows环境下安装Maven

想要安装 Apache Maven 在Windows 系统上, 需要下载 Maven 的 zip 文件，并将其解压到你想安装的目录，并配置 Windows 环境变量。
注 Maven 3.2 要求 JDK 1.6 或以上版本

1.下载Apache Maven

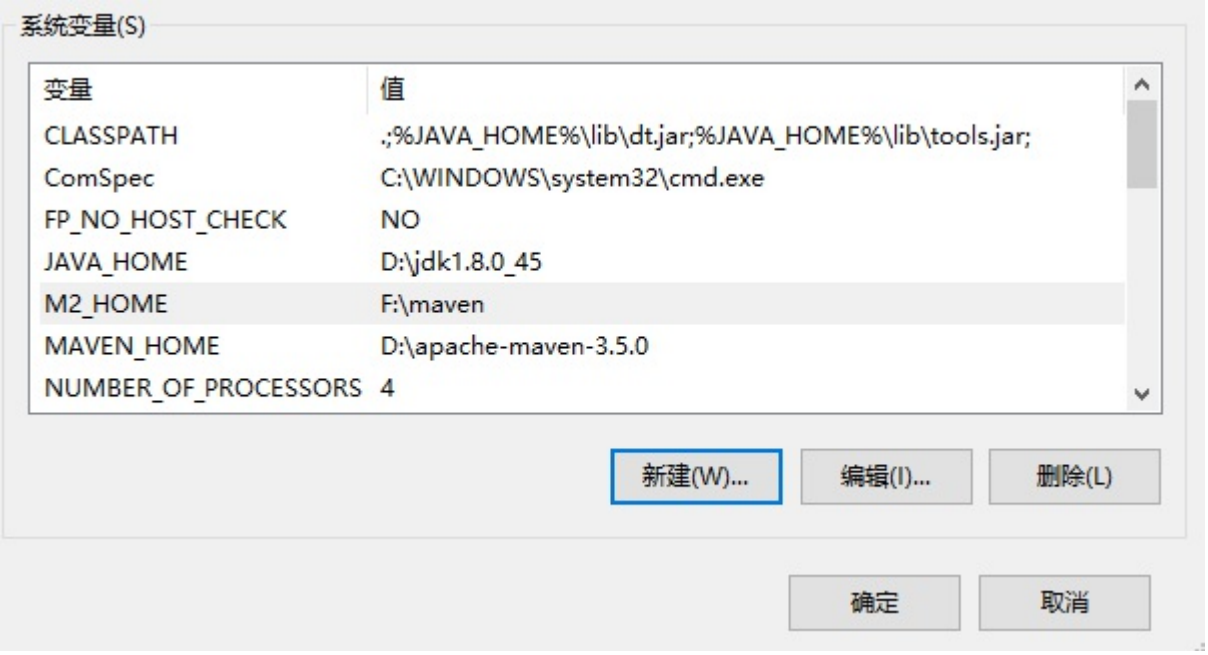
访问 Maven官方网站，打开后找到下载链接，如下：

	Link	Checksum	Signature
Binary tar.gz archive	apache-maven-3.3.3-bin.tar.gz	apache-maven-3.3.3-bin.tar.gz.md5	apache-maven-3.3.3-bin.tar.gz.asc
Binary zip archive	apache-maven-3.3.3-bin.zip	apache-maven-3.3.3-bin.zip.md5	apache-maven-3.3.3-bin.zip.asc
Source tar.gz archive	apache-maven-3.3.3-src.tar.gz	apache-maven-3.3.3-src.tar.gz.md5	apache-maven-3.3.3-src.tar.gz.asc
Source zip archive	apache-maven-3.3.3-src.zip	apache-maven-3.3.3-src.zip.md5	apache-maven-3.3.3-src.zip.asc

下载 Maven 的 zip 文件，将它解压到你要安装 Maven 的文件夹。

2.添加 M2_HOME 和 MAVEN_HOME

添加 M2_HOME 和 MAVEN_HOME 环境变量到 Windows 环境变量，MAVEN_HOME指向 Maven 文件夹，M2_HOME指向本地仓库。



3.添加到环境变量 - PATH

更新 PATH 变量，添加 Maven bin 文件夹到 PATH 的最后，如： %M2_HOME%\bin, 这样就可以在命令中的任何目录下运行 Maven 命令了。

4.验证

在命令行中执行

```
mvn –version
```

输出结果：

```
C:\Users\01> mvn -version Apache Maven 3.5.0 (ff8f5e7444045639af65f6095c62210b5713f426; 2017-04-04T03:39:06+08:00) Maven home: D:\apache-maven-3.5.0\bin.. Java version: 1.8.0_45, vendor: Oracle Corporation Java home: D:\jdk1.8.0_45\jre Default locale: zh_CN, platform encoding: GBK OS name: "windows 8.1", version: "6.3", arch: "amd64", family: "windows"
```

说明安装成功

Maven启用代理访问

如果公司建立了防火墙，并使用HTTP代理服务器来阻止用户直接连接到互联网。如果不是用代理就无法下载依赖， 为了使maven能正常工作，必须在setting.xml文件中配置maven代理

找到文件 {maven安装目录}/conf/settings.xml, 并把代理服务器信息配置写入

```
<proxies>
<proxy>
<id>${proxy id}</id>
<active>true</active>
<protocol>http</protocol>
<username>${用户名}</username>
<password>${密码}</password>
<host>${代理主机}</host>
<port>${端口}</port>
<nonProxyHosts>${不需要代理的域名}</nonProxyHosts>
</proxy>
</proxies>
```

Maven相关概念

Maven目录结构

Maven采用约定大于配置原则，maven工程主要由一下几个目录构成:

目录	描述
src/main/java	程序/类库的源码
src/main/resources	程序/类库的资源
src/main/filters	资源过滤文件
src/main/webapp	web application sources - web应用的目录，WEB-INF,js,css等
src/test/java	单元测试java源代码文件
src/test/resources	测试需要的资源库
src/test/filters	测试资源过滤库
src/site	一些文档
pom.xml	工程描述文件
LICENSE.txt	license
README.txt	read me
target/	存放项目构建后的文件和目录，jar包,war包，编译的class文件等；Maven构建时生成的

Maven仓库

首次运行完mvn -version后，会在用户目录下创建一个.m2的目录(比如：C:\Users\当前用户名.m2)，这个目录是maven的“本地仓库”，仓库是maven中一个很重要的概念。

试想一下，我们会在工作中同时创建很多项目，每个项目可能都会引用一些公用的jar包，一种作法是每个项目里，都复制一份这些依赖的jar包，这样显然不好，相同的文件在硬盘上保存了多份，太占用空间，而且这些依赖的jar包的版本也不太好管理(比如某个公用的jar包，从1.0升级到2.0，如果所有引用这个jar包的项目都需要更新，必须一个个项目的修改)。

maven的仓库则很好的解决了这些问题，它在每台机器上创建一个本机仓库，把本机上所有maven项目依赖的jar包统一管理起来，而且这些jar包用“坐标”来唯一标识(注：坐标是另一个重要的概念，后面还会讲到，这里只要简单理解成“唯一识别某个jar包文件名、版本号”的标识即可)，这样所有maven项目就不需要再象以前那样把jar包复制到lib目录中，整个maven项目看起来十分清爽。
有如下几个maven仓库：

- Maven本地资源库:的本地资源库是用来存储项目的依赖库，默认的文件夹是 “.m2” 目录，可能需要将其更改为另一个文件夹。
- Maven中央存储库:中央存储库是 Maven 用来下载所有项目的依赖库的默认位置。
- Maven远程仓库:并非所有的库存储在Maven的中央存储库，很多时候需要添加一些远程仓库来从其他位置，而不是默认的中央存储库下载库。

Maven POM文件

Maven项目的核心是pom.xml,POM(Project Object Model,项目对象模型)定义了项目的基本信息，用于描述项目如何构建，声明项目依赖，等等，下面是一个简单的pom.xml实例

```
<?xml version="1.0" encoding="UTF-8"?>
<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.xhystc</groupId>
<artifactId>mvntest</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
```

```
<name>mvntest</name>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>
</project>
```

第一行指定了该xml的版本及编码方式，紧接着是project元素，这是所有pom.xml的根元素，它还声明了一些POM相关的命名空间及xsd元素，虽然这些元素不是必须的，但是使用这些属性能够让IDE帮助我们校验POM

- modelVersion:指定了POM模型的版本，对于Maven2及Maven3来说，它只能是4.0.0
- groupId:定义了项目属于哪个组，这个组往往和项目所在的组织或公司存在关联
- artifactId:定义了当前Maven项目在组中唯一的ID，也就是项目的代号
- packaging:定义了当前Maven项目项目的打包方式，默认为jar
- version:指定了项目的版本。SNAPSHOT意为快照，说明该项目还处在开发中，还是不稳定版本。随着项目的发展，version会不断的更新，如升级为1.0、1.1-SNAPSHOT、1.1、2.2
- name:该元素声明了一个对于用户更为友好的项目名称，这不是必须的
- dependencies:该元素制订了项目依赖于哪些模块，依赖机制maven最主要的功能，下一节会重点讲解

Maven依赖

依赖机制是Maven最为用户熟知的特性之一，同时也是Maven所擅长的领域之一。单个项目的依赖管理并不难，但是当你面对包含数百个模块的多模块项目和应用时，Maven能帮你保证项目的高度控制力和稳定性。

最简单的依赖

依赖是使用Maven坐标来定位的，而Maven坐标主要由GAV（groupId, artifactId, version）构成。因此，使用任何一个依赖之前，你都需要知道它的Maven坐标。

```
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.4</version>
</dependency>
```

上例中声明了一个对junit的依赖，它的groupId是junit, artifactId是junit, version是4.4。这一组GAV构成了一个Maven坐标，基于此，Maven就能在本地或者远程仓库中找到对应的junit-4.4.jar文件。

依赖的传递

传递性依赖是Maven2.0的新特性。假设你的项目依赖于一个库，而这个库又依赖于其他库。你不必自己去找出所有这些依赖，你只需要加上你直接依赖的库，Maven会隐式的把这些库间接依赖的库也加入到你的项目中。这个特性是靠解析从远程仓库中获取的依赖库的项目文件实现的。一般的，这些项目的所有依赖都会加入到项目中，或者从父项目继承，或者通过传递性依赖。 传递性依赖的嵌套深度没有任何限制，只是在出现循环依赖时会报错。

依赖范围

依赖范围会影响传递性依赖，同时也会影响项目构建任务中使用的classpath。

依赖管理-依赖范围：

依赖范围（Scope）	对于主代码classpath有效	对于测试代码classpath有效	被打包，对于运行时classpath有效	例子
compile	Y	Y	Y	log4j
test	-	Y	-	junit
provided	Y	Y	-	servlet-api
runtime	-	-	Y	JDBC Driver Implementation

其中依赖范围scope 用来控制依赖和编译，测试，运行的classpath的关系. 主要的是三种依赖关系如下：

- compile： 默认编译依赖范围。对于编译，测试，运行三种classpath都有效。有些依赖在主代码中需要import，在测试代码中也需要import，打包的时候还

需要一起打包上传服务器，则scope是compile。

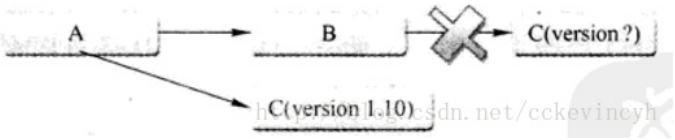
- test：测试依赖范围。只对于测试classpath有效有些代码是测试需要import，而主代码中不需要，我们不需要把junit打包上传到服务器，则scope是test
- provided：已提供依赖范围。对于编译，测试的classpath都有效，但对于运行无效。因为由容器已经提供，例如servlet-ap以servlet-api为例，tomcat中已经提供了servlet-api的jar包，但是本地写代码的时候，只需要import进去，而不需要把servlet-api的jar包打包到服务器上，如果打包了，可能会产生jar包冲突，这个时候用provided。
- runtime:运行时提供,编码时不需要import。例如:jdbc驱动

排除依赖

传递性依赖会给项目隐式地引入很多依赖，着极大简化了项目依赖的管理，但是有些时候这种特性也会带来问题。比如Sun JTA API，Hibernate依赖于这个JAR，但是由于版权的因素，该类库不在中央仓库中，而Apache Geronimo项目有一个对应的实现。这时你就可以排除Sun JAT API，而声明Geronimo的JTA API实现。下面是个简单的例子：

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.cc.maven</groupId>
  <artifactId>project-a</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
  <groupId>com.cc.maven</groupId>
  <artifactId>project-b</artifactId>
   <version>1.0.0</version>
   <exclusions>
     <exclusion>
     <groupId>com.cc.maven</groupId>
      <artifactId>project-c</artifactId>
     </exclusion>
     </exclusions>
   </dependency>
   <dependency>
     <groupId>com.cc.maven</groupId>
     <artifactId>project-b</artifactId>
     </dependency>
  </dependencies>
</project>
```

上述代码中，项目A依赖于项目B，但是由于一些原因，不想引入传递性依赖C，而是自己显式声明对项目C 1.1.0版本的依赖。代码中使用exclusions元素声明排除依赖，exclusions可以包含一个或者多个exclusion子元素，因此可以排除一个或者多个传递性依赖。需要注意的是，声明exclusion的时候只需要groupId和artifactId，而不需要version元素，这是因为只需要groupId和artifactId就能唯一定位依赖图中的某个依赖。换句话说，maven解析后的依赖中，不可能出现groupId和artifactId项目，但是version不同的两个依赖。该例的依赖解析逻辑如下所示：



归类依赖

如果引入的项目包含多个模块，这时所有这些依赖模块的版本都是相同的，可以知道，如果将来想要升级被依赖项目版本的话，这些依赖模块的版本需要一起升级。如果依赖版本硬编码在pom.xml文件里会给升级版本带来不便。使用常量不仅可以使代码变得更加简洁，更重要的是可以避免重复，只需要更改一处，降低了错误的概率。

```
<properties>
<springframework.version>2.5.6</springframework.version>
</properties>
```

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-core</artifactId>
```

```
<version>${springframework.version}</version>
</dependency>
```

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-web</artifactId>
<version>${springframework.version}</version>
</dependency>
```

这里简单使用到了Maven的属性，首先使用properties元素定义了Maven的属性，该例子中定义了一个springframework.version子元素，其值为2.5.6，有了这个属性之后，Maven运行的时候，会将POM中所有${springframework.version}替换成2.5.6，也就是说，可以使用${}的方式来引用Maven的属性，然后将所有Spring Framework依赖的版本值用这一属性引用。这个和在java中常量PI替换3.14是同样的道理，只是语法不同而已。

依赖的继承

实际的项目中，会有一大把的Maven模块，而且你往往发现这些模块有很多依赖是完全相同的，A模块有个对spring的依赖，B模块也有，它们的依赖配置一模一样，同样的groupId, artifactId, version，或者还有exclusions, classifer。重复就意味着潜在的问题，Maven提供的继承机制就是用来消除这种重复的。

定义父POM

```
<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.test.maven</groupId>
<artifactId>parent</artifactId>
<version>1.0.0</version>
<packaging>pom<packaging>
<name>Parent</name>
<dependencies>
...
</dependencies> </project>
```

注 父模块packaging类型必须为pom

子POM继承父POM

```
<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>

<parent>
<groupId>com.test.maven</groupId>
<artifactId>parent</artifactId>
<version>1.0.0</version>
<relativePath>../parent/pom.xml</relativePath>
</parent>

<artifactId>child</artifactId>
<name>child</name>
<dependencies>
...
</dependencies>

</project>
```

可被继承的POM元素:

- groupId: 项目组ID，项目坐标的核心坐标；
- version: 项目版本，项目坐标的核心坐标；
- description: 项目的描述信息；
- organization: 项目的组织信息；
- inceptionYear: 项目的创始年份；
- url: 项目的URL地址；
- developers: 项目的开发者信息；

- contributors: 项目的贡献值和信息；
- distributionManagement: 项目的部署配置；
- issueManagement: 项目的缺陷跟踪系统；
- ciManagement: 项目的持续集成系统信息；
- scm: 项目的版本控制系统信息；
- mailingLists: 项目的邮件列表信息；
- properties: 自定义的Maven属性；
- dependencies: 项目的依赖配置；
- dependencyManagement: 项目的依赖管理配置；
- repositories: 项目的仓库配置；
- build: 包括项目的源码目录配置、输出目录配置、插件配置、插件管理配置等；
- reporting: 包括项目的报告输出目录配置、报告插件配置等。

Maven生命周期

Maven有三套相互独立的生命周期，请注意这里说的是“三套”，而且“相互独立”，初学者容易将Maven的生命周期看成一个整体，其实不然。这三套生命周期分别是：

- Clean Lifecycle 在进行真正的构建之前进行一些清理工作。
- Default Lifecycle 构建的核心部分，编译，测试，打包，部署等等。
- Site Lifecycle 生成项目报告，站点，发布站点。

mvn clean 生命周期

Clean生命周期一共包含了三个阶段：

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

mvn site 生命周期

- pre-site 执行一些需要在生成站点文档之前完成的工作
- site 生成项目的站点文档
- post-site 执行一些需要在生成站点文档之后完成的工作，并且为部署做准备
- site-deploy 将生成的站点文档部署到特定的服务器上

mvn default 生命周期

Default是最重要的生命周期，绝大部分工作都发生在这个生命周期中，主要由一下几个重要阶段构成：

- 验证（validate） - 验证项目是否正确，所有必要的信息可用
- 编译（compile） - 编译项目的源代码
- 测试（test） - 使用合适的单元测试框架测试编译的源代码。这些测试不应该要求代码被打包或部署
- 打包（package） - 采用编译的代码，并以其可分配格式（如JAR）进行打包。
- 验证（verify） - 对集成测试的结果执行任何检查，以确保满足质量标准
- 安装（install） - 将软件包安装到本地存储库中，用作本地其他项目的依赖项
- 部署（deploy） - 在构建环境中完成，将最终的包复制到远程存储库以与其他开发人员和项目共享。

使用Maven

使用maven构建简单java工程

在命令行输入

> mvn archetype:generate

第一次使用需要等待maven 下载相关文件，之后maven会列出大量maven工程模板,然后提示你输入要选择的模版号或过滤名称

> Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains)

输入

```
maven-archetype-quickstart
```

然后选择第一项，显示如下内容

```
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): 1: 1
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:
```

随便选择一个archtype版本，之后依次填写groupId、artifactId、版本号和包名

```
Define value for property 'groupId': com.mvntest
Define value for property 'artifactId': mvntest
Define value for property 'version' 1.0-SNAPSHOT: : 1.0
Define value for property 'package' com.mvntest: : com.mvntest.demo
Confirm properties configuration:
groupId: com.mvntest
artifactId: mvntest
version: 1.0
package: com.mvntest.demo
Y: : y
```

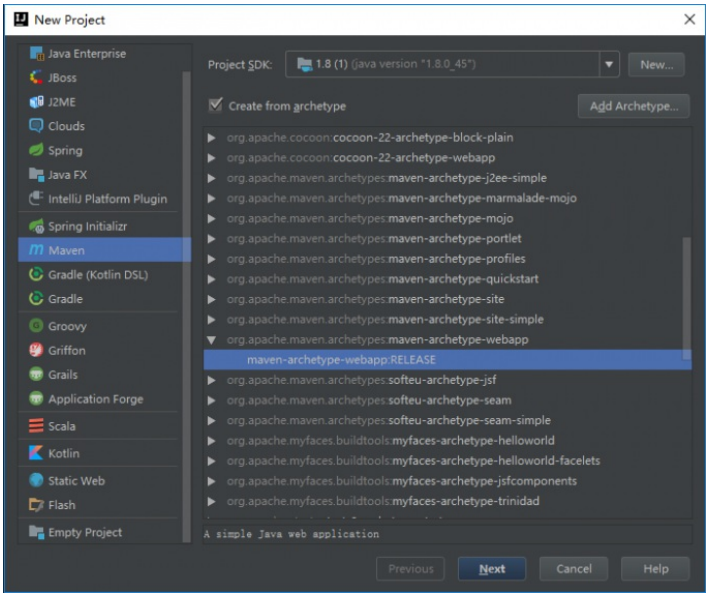
确认后可以看到当前目录下生成了maven工程的目录结构

```
└─mvntest
    └─src
        ├─main
        │  └─java
        │      └─com
        │          └─xhystc
        │              └─mvntest
        └─test
            └─java
                └─com
                    └─xhystc
                        └─mvntest
```

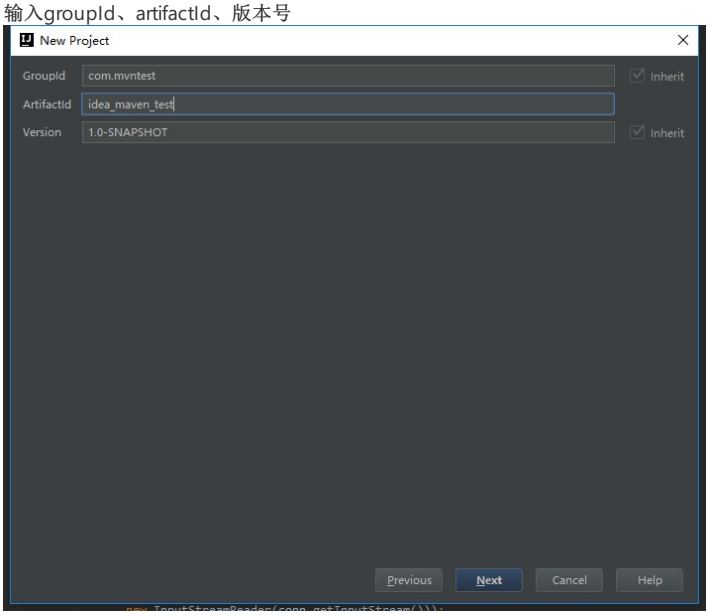
目录中包含了pom.xml文件，名为App的主类，以及一个Junit单元测试。
在目录下可以使用mvn test进行测试，使用mvn package命令打包

在IntelliJ IDEA中使用maven

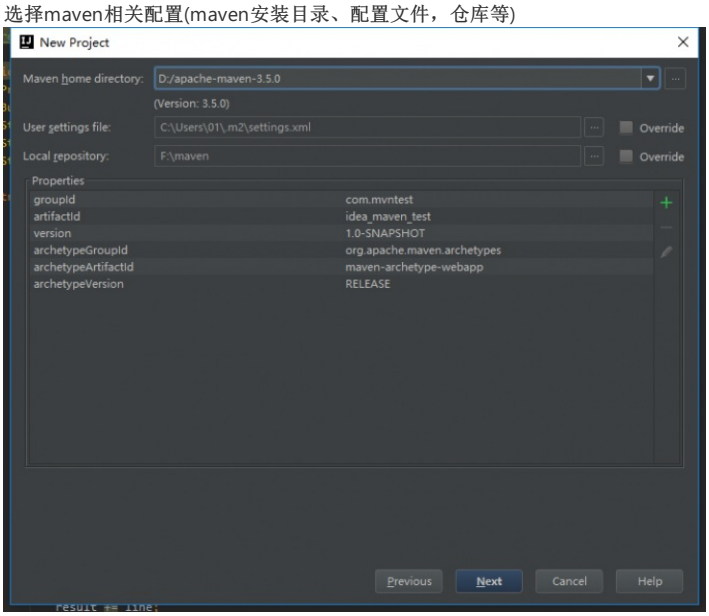
在idea中新建maven工程，这次我们选择weapp工程模版



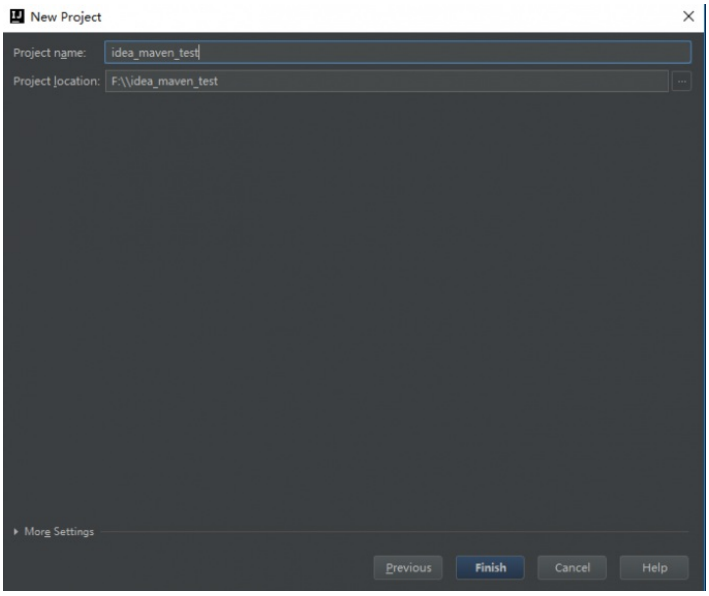
输入groupId、artifactId、版本号



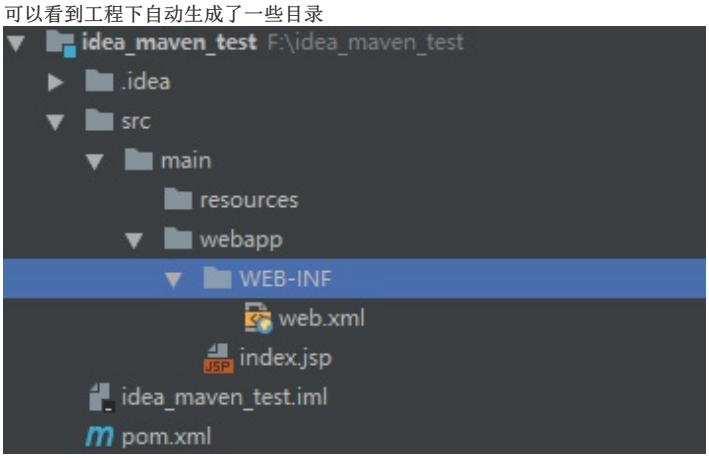
选择maven相关配置(maven安装目录、配置文件，仓库等)



填写工程名、所在目录，完成



可以看到工程下自动生成了一些目录

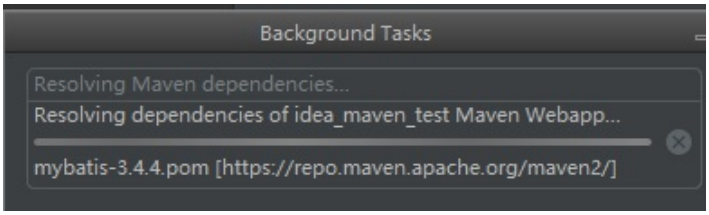


我们在pom.xml中添加一些webapp常用的依赖

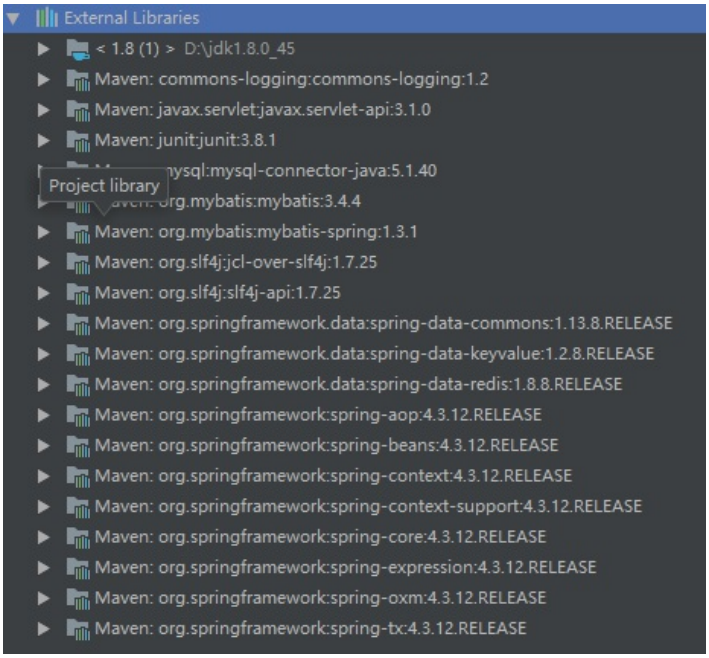
```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.4.4</version>
</dependency>
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>1.3.1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework.data/spring-data-redis -->
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
    <version>1.8.8.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.40</version>
</dependency>
```

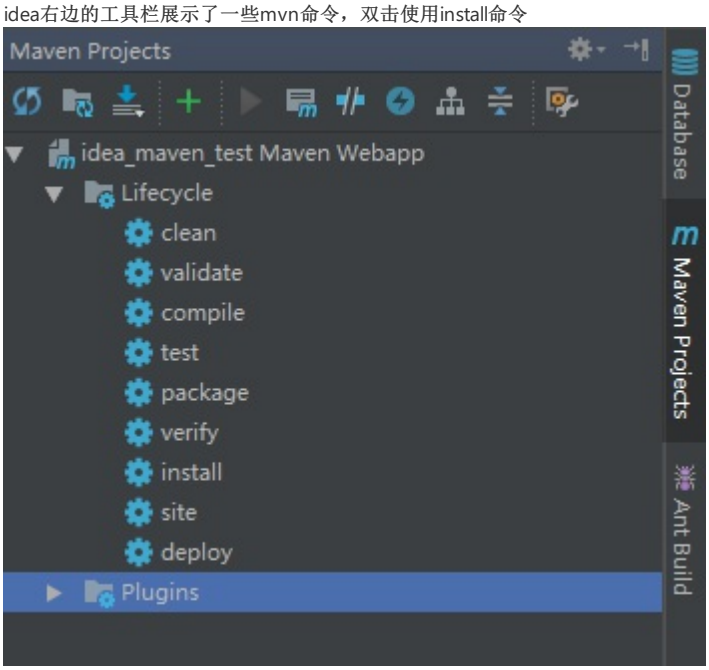
可以看到idea自动开始下载依赖



左边工程目录可以查看用了哪些maven依赖



idea右边的工具栏展示了一些mvn命令，双击使用install命令



maven开始工作

${工程目录}/target目录下生成了打包后的war文件