# maven概念

Maven 是一个项目管理和整合工具。Maven 为开发者提供了一套完整的构建生命周期框架。开发团队几乎不用花多少时间就能够自动完成工程的基础构建配置，因为 Maven 使用了一个标准的目录结构和一个默认的构建生命周期。

在有多个开发团队环境的情况下，Maven 能够在很短的时间内使得每项工作都按照标准进行。因为大部分的工程配置操作都非常简单并且可复用，在创建报告、检查、构建和测试自动配置时，Maven 可以让开发者的工作变得更简单。

Maven 的主要目的是为开发者提供

* 一个可复用、可维护、更易理解的工程综合模型
* 与这个模型交互的插件或者工具

Maven 工程结构和内容被定义在一个 xml 文件中 － pom.xml

# 安装配置

## jdk安装

Maven 是一个基于 Java 的工具，所以要做的第一件事情就是安装 JDK

## windows安装，

下载后解压，添加环境变量即ok

maven下载地址：[**http://maven.apache.org/download.html**](http://maven.apache.org/download.html)

## linux安装

**1、配置yum源**

**# sudo yum install -y yum-utils**

**# yum-config-manager --add-repo http://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-maven.repo**

**2、 安装maven**

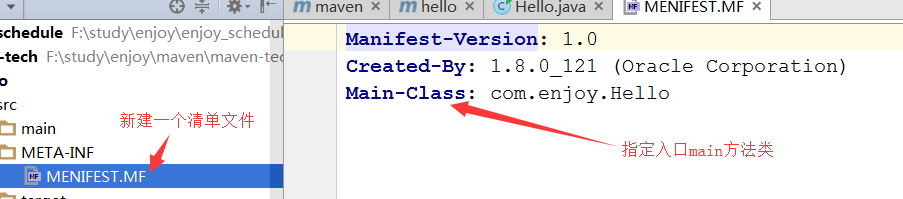
**# yum install -y apache-maven**

# maven初体验

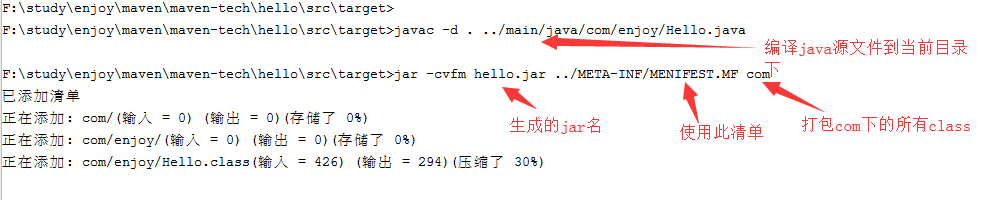
maven主要是做什么的？---打jar、war包和管理jar包

## 原始的javac打包

假如我们有一个Hello.java文件，想要将它打成可执行jar包，一般这样做。



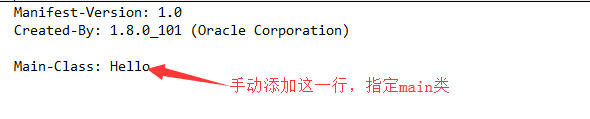
使用javac命令和jar命令打包



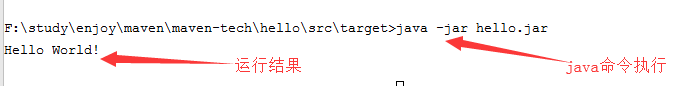
生成的jar包文件，由两部分组成，class文件和META-INF目录，如下：

此时的jar包，是不可直接运行的，需要指定入口main类

进入META-INF目录，编辑MANIFEST.MF文件，如下：



运行此jar包，得到运行结果



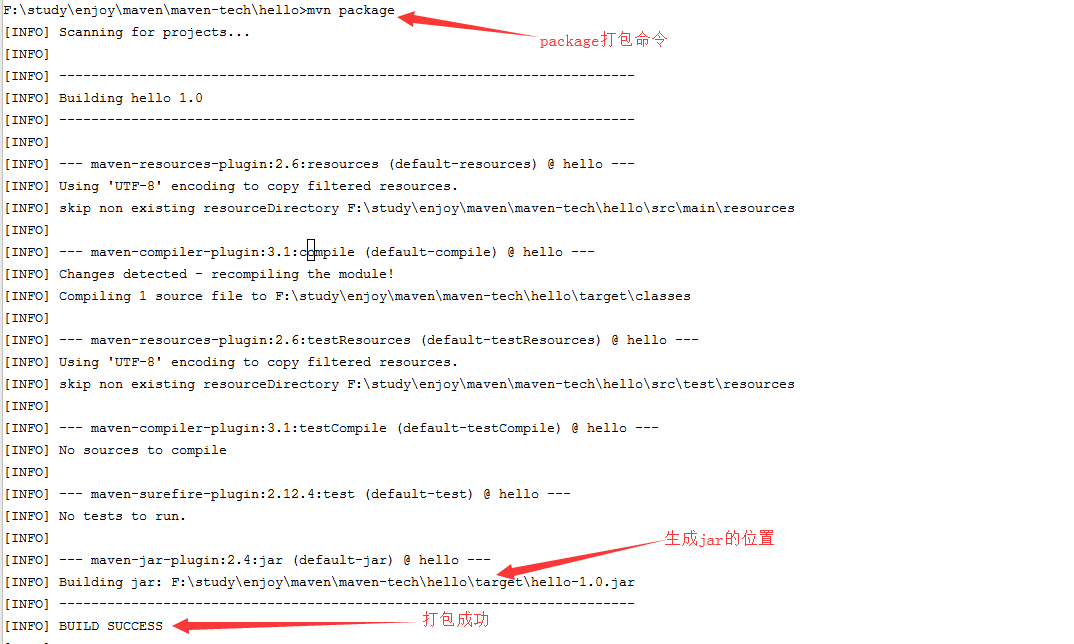
## maven打包

在pom文件内配置入口类

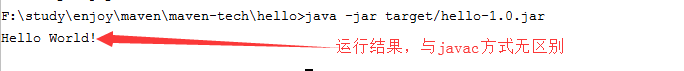


执行maven打包命令

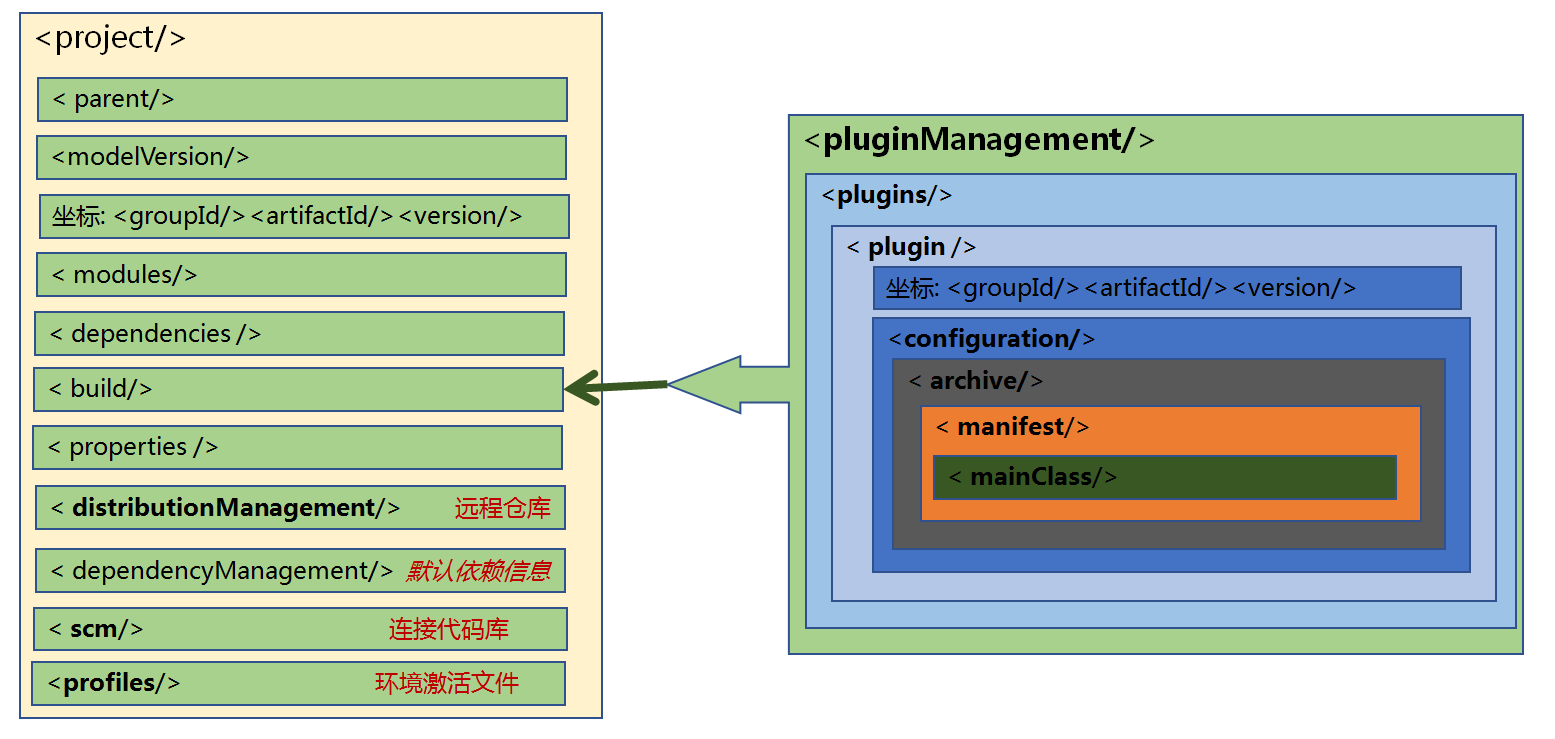
mvn package



运行生成的jar，结果与原始方式无区别



## POM结构



# maven流程

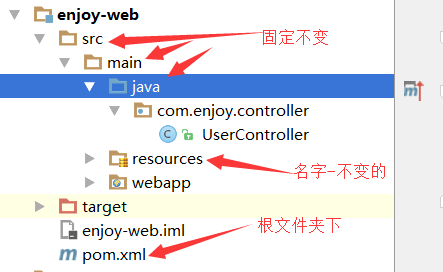
## maven的理想

理想：像一种什么设计模式？---模板方法模式

---------------自动走完标准的构建流程：清理编译測试报告打包部署

---------------统一入口，所有配置在一个pom里搞定

## maven的约定

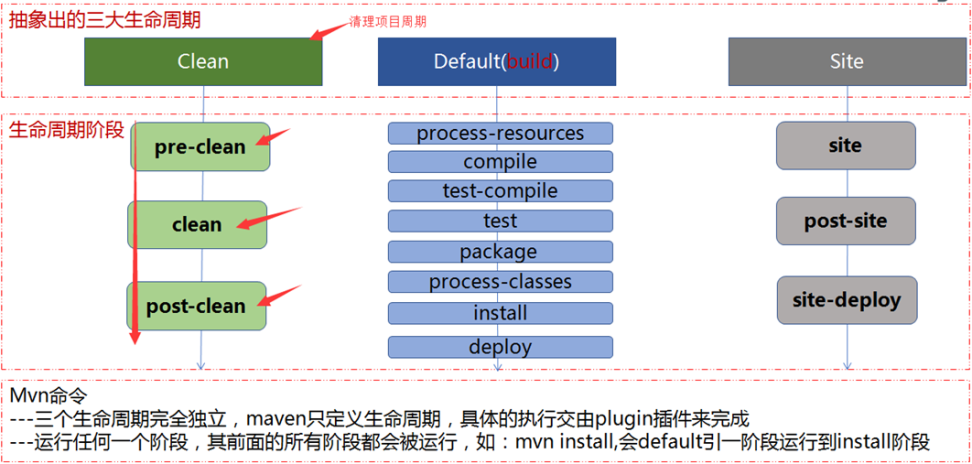


约定的目录（**不可改**）：

|  |
| --- |
| src/main/java –存放项目的.java文件 |
| src/main/resources –存放项目资源文件。比方spring,hibernate配置文件 |
| src/test/java –存放全部測试.java文件，比方JUnit測试类 |
| src/test/resources ---測试资源文件 |
| target ---项目输出位置,编译完毕后的东西放到这里面 |
| pom.xml |

## maven的生命周期

maven的构建生命周期，只是一个抽象规范流程。周期内的每个阶段的具体执行，是在插件里来实现。



各个生命周期预设的定义如下：

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

运行任何一个阶段，都会从其所在生命周期的第一个阶段开始，顺序执行到指定的阶段，如：

mvn package --- 本义：执行default周期的package阶段，maven会自动从process-resources阶段开始运行到package阶段结束

## maven的插件

插件plugin是绑定到生命周期，承担实际功能的组件。mvn运行时，自动关联插件来运行

下图是maven默认的各阶段对应的插件列表：

|  |  |  |  |
| --- | --- | --- | --- |
| **生命周期** | **生命周期阶段** | **插件目标** | **执行任务** |
| clean | pre-clean |  |  |
|  | clean | maven-clean-plugin:clean | 删除项目的输出目录。 |
|  | post-clean |  |  |
| site | pre-site |  |  |
|  | site | maven-site-plugin:site |  |
|  | post-site |  |  |
|  | site-deploy | maven-site-plugin:deploy |  |
| default | process-resources | maven-resources-plugin:resources | 复制主资源文件至主输出目录 |
|  | compile | maven-compiler-plugin:compile | 编译主代码至主输出目录 |
|  | process-test-resources | maven-resources-plugin:testResources | 复制测试资源文件至测试输出目 |
|  | test-compile | maven-compiler-plugin:testCompile | 编译测试代码至测试输出目录 |
|  | test | maven-surefire-plugin:test | 执行测试用例 |
|  | package | maven-jar-plugin:jar（ejb:ejb jar:jar rar:rar war:war） | 创建项目jar包 |
|  | install | maven-install-plugin:install | 将项目输出构件安装到本地仓库 |
|  | deploy | maven-deploy-plugin:deploy | 将项目输出构件部署到远程仓库 |

## 常用Maven命令：

mvn clean  清理

mvn compile 编译主程序

mvn package 打包

mvn install 安装jar到本地库

使用maven命令生成项目（idea和eclipse生成项目最终也是依赖maven插件生成的）

mvn archetype:generate -DgroupId=enjoy -DartifactId=simple -DarchetypeArtifactId=maven-archetype-quickstart -Dversion=1.0

mvn archetype:generate -DgroupId=enjoy -DartifactId=simple-web -DarchetypeArtifactId=maven-archetype-webapp -Dversion=1.0

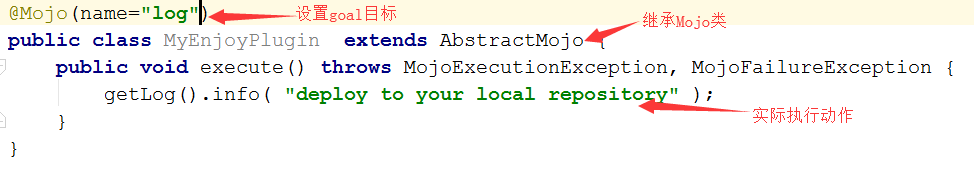
## maven插件开发

可以自定义插件，来扩展maven的功能。插件的开发步骤如下：

### 1、引入maven api的依赖



### 2、编写简单Mojo类---继承AbstractMojo



### 3、执行插件

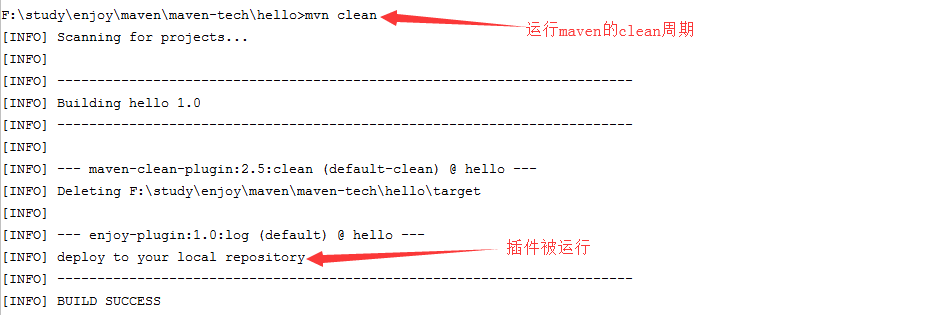
mvn com.enjoy:enjoy-plugin:1.0:log



### 4、关联插件到生命周期来执行



构建项目对应的生命周期



# maven坐标与依赖

## 坐标

--------在数学中, 任何一个坐标可以唯一确定一个**“点”**

Maven 中坐标是Jar包的唯一标识

坐标元素包括groupId、artifactId、version、packaging:

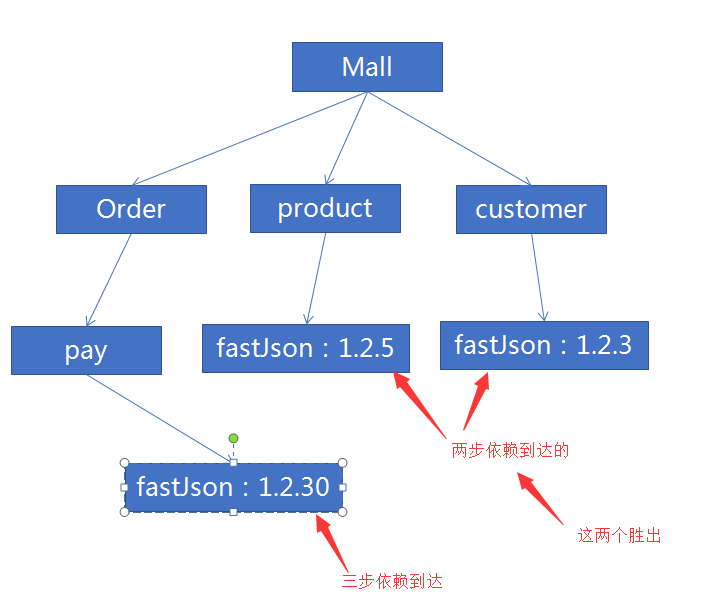
|  |  |  |
| --- | --- | --- |
| 元素 | 描述 | 说明 |
| groupId | 定义当前模块隶属的实际Maven项目 | 中小企业常常直接对应公司/组织 |
| artifactId | 定义实际项目中的一个Maven模块 | 唯一标识一个模块 |
| version | 定义当前项目所处版本 | SNAPSHOT 表示不稳定的版本。 LATEST 指最新发布的版本，可能是个发布版，也可能是一个snapshot版。 RELEASE 指最后一个发布版。 |
| packaging | 定义Maven项目打包方式 | 有jar(默认)、war、pom、maven-plugin等. |
| classifier | 附属构件(如javadoc、sources) | 须有附加插件的帮助 |

## 依赖

依赖即：A --》B，B--》C，C--》D这种项目间的依存关系。

在java的jvm内，依赖的最终表现是，项目A启动时，其依赖的jar包必须都对应放入其classpath路径内

## 依赖传递：



上述过程中，项目Mall归结起来，依赖的fastjson会有三个版本。

而我们的jvm最终肯定只能接受一个版本的jar，所以必须有所取舍

maven默认的取舍规则是：

1、路径最短原则：product和customer里的fastjson引用路径较短，路径为两步 ;pay项目里的fastjson引用路径较长，路径为三步。因此pay中的fastjson被淘汰

2、同路径长度下，谁先声明谁优先： product和customer中的fastjson路径相同，那么就看在pom中是先声明product还是先声明customer，谁先用谁的

## 依赖冲突及解决

在依赖传递里，我们看到，maven根据自己的规则为我们取舍出了一个版本的jar，但此jar版本选择可能会与我们的项目预期不符，

如：我们最终想的版本是fastjson:1.2.30版本-----但它在第一步即被淘汰掉了

当出现此类情况时，我们项目运行可能会出错（项目中使用到了1.2.30版本的特性）--------------- 此问题即是我们常遇到的jar包冲突问题

补救方式 ----- 使用exclusions将product和customer中的fastjson包排除掉，用法如下图：



当发生jar冲突程序报错时，可以使用mvn命令查出项目最终依赖的jar包树，看版本是否是我们预期的：

命令： mvn dependency:tree

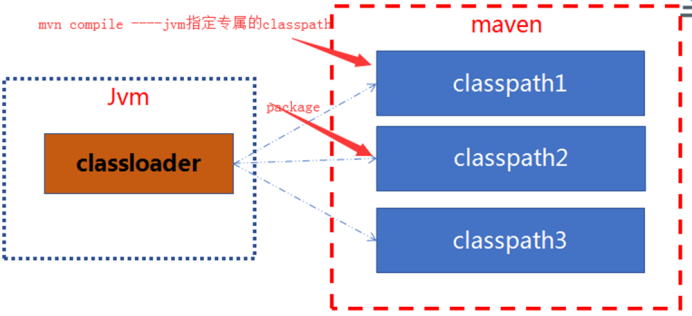


## 依赖范围scope

mvn在运行时，生命周期的不同阶段，会有不同的依赖范围，一般有以下依赖范围scope

- compile ：默认范围，用于编译 ------- 依赖的jar在打包时会包含进去        
- provided：类似于编译，但支持你期待jdk或者容器提供，类似于classpath   -------- 依赖的jar在打包时不会包含进去   
- runtime: 在执行时需要使用      -------依赖的jar在打包时会包含进去  
- test:    用于test任务时使用       -------依赖的jar在打包时不会包含进去  
~~- system: 需要外在提供相应的元素。通过systemPath来取得~~    ----- 一般禁止使用

每个scope实际上是配置了一个不同的classpath，jvm根据选择不同的classpath来达到依赖不同

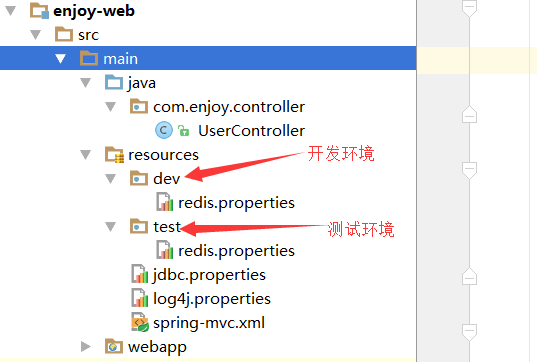


# 环境激活---profiles的使用

在springmvc项目中，开发/测试/线上三个不同环境，配置文件往往也不同。

打包时需要对配置文件做出选择---maven提供了profiles机制供我们使用

## profiles的场景



这个选择，实际发生在default生命周期的resource阶段---maven-resources-plugin执行过程里

## 定义profiles

为了指导插件将对应的resource文件打入classpath里，先定出profiles



此定义即指，当mvn命令执行时，我们需要通过 -P dev或者-P test方式传入我们的意图：

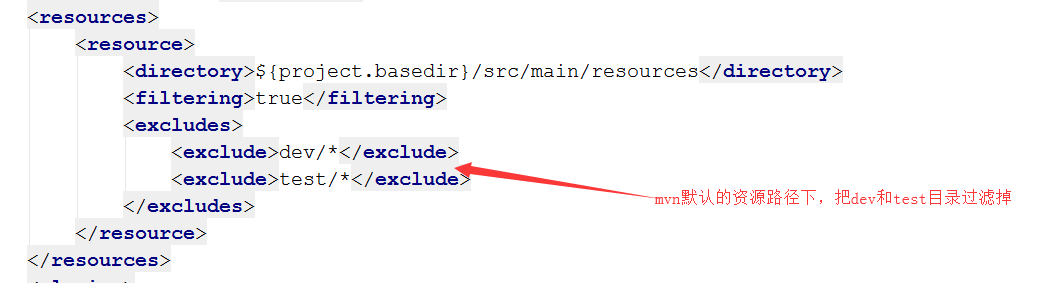
dev/test选择，会导致properties里的变量值含义不同，我们主要关注package.environment变量

## 资源插件的配置指定



配置maven-resources-plugin插件执行时，要复制的目录资源

## mvn约定的资源中需要过滤掉环境目录



需要将mvn约定的资源目录里，过滤掉环境目录

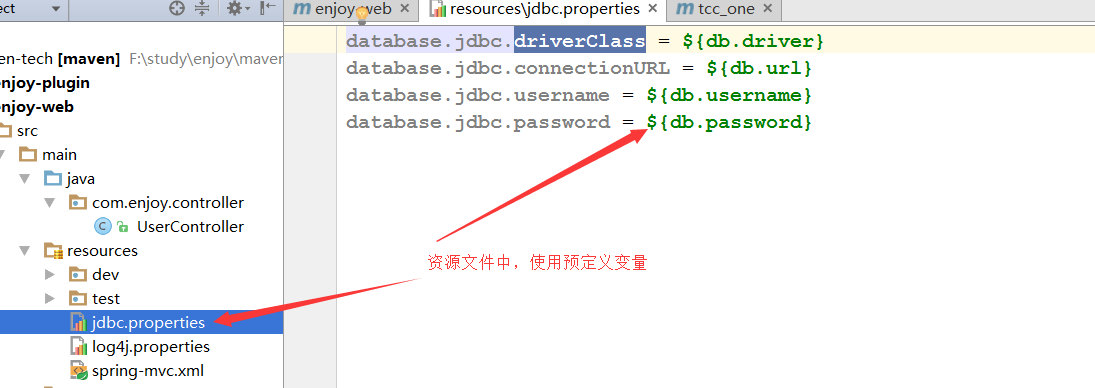
## 小属性更轻便的用法

对于简单的属性，我们可以选择更轻便的用法

### 直接在环境中定义属性值



### 项目属性文件配置



### 约定的资源启用替换过滤

