
# Maven基础

# 目录 Contents

# 目录 Contents

# 课程约定



## 资料格式

- 配置文件

```
<groupId>com.itheima</groupId>
```

- Java代码

```
 Statement stat = con.createStatement();
```

- 示例

```
<groupId>com.itheima</groupId>
```

- 命令

```
mvn test
```

# 目录 Contents

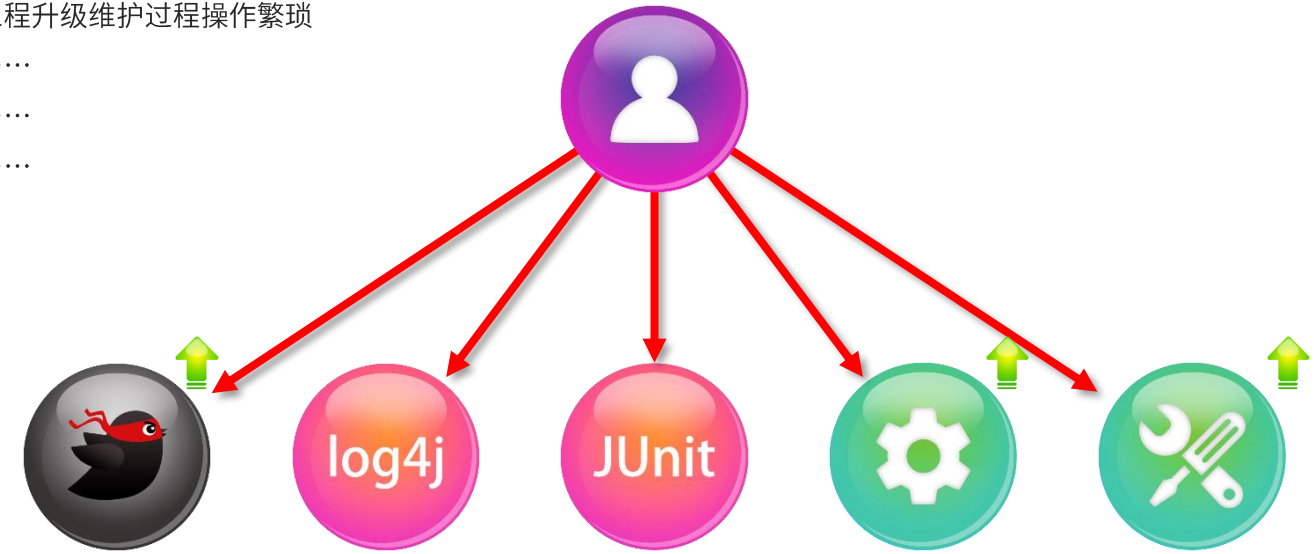
# 目录 Contents

# Maven简介



## 传统项目管理状态分析

- jar包不统一，jar包不兼容
- 工程升级维护过程操作繁琐
- ……
- ……
- ……

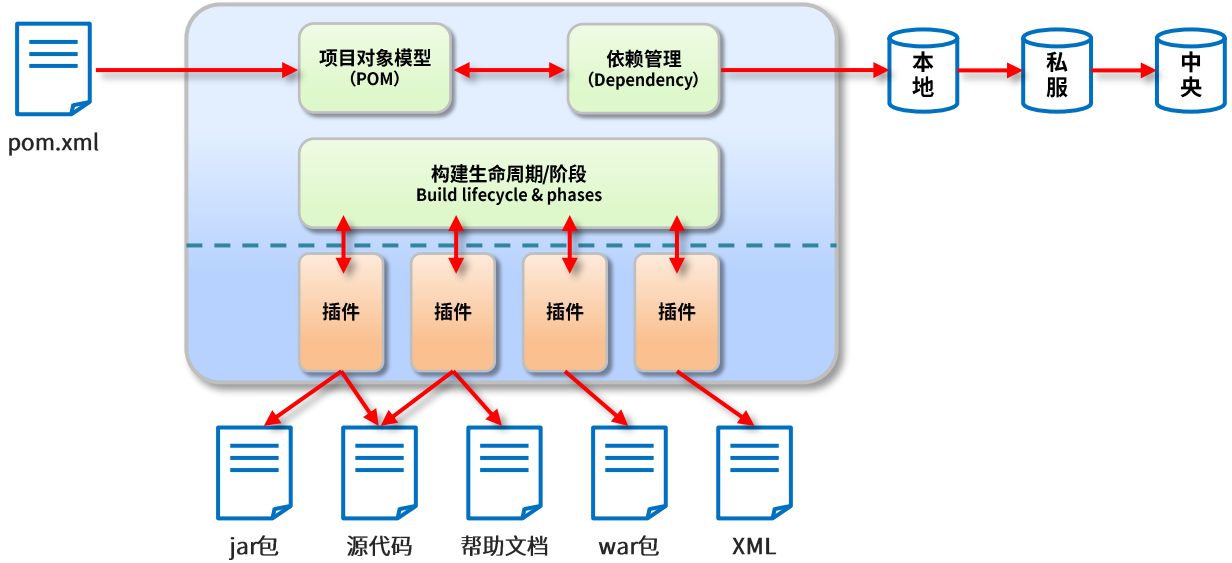


# Maven简介



## Maven是什么

- Maven 的本质是一个项目管理工具，将项目开发和管理过程抽象成一个项目对象模型（POM）
- POM （Project Object Model）：项目对象模型

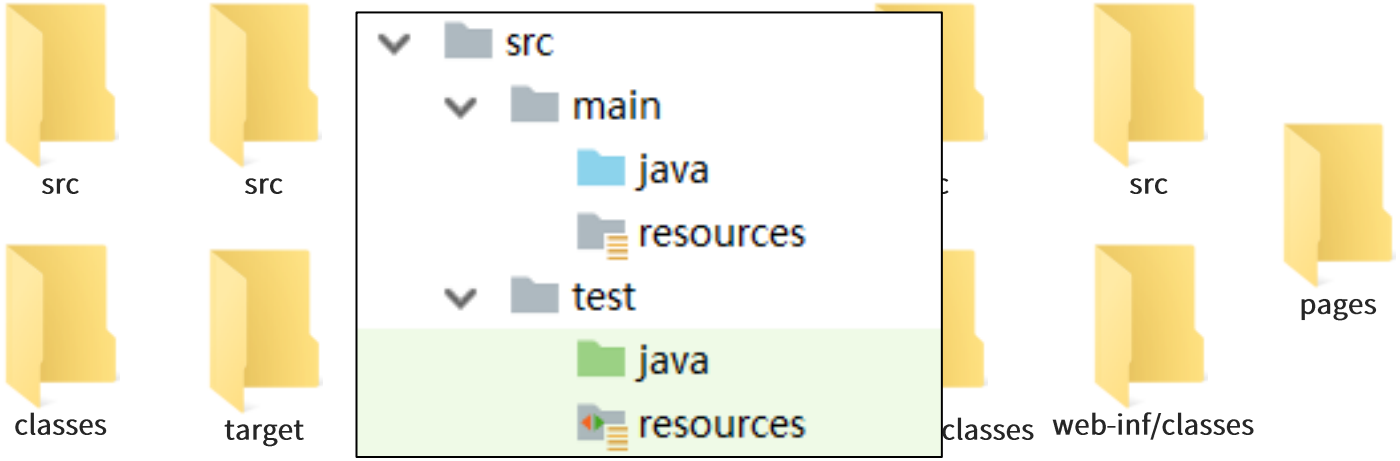


# Maven简介



## Maven的作用

- 项目构建：提供标准的、跨平台的自动化项目构建方式
- 依赖管理：方便快捷的管理项目依赖的资源（jar包），避免资源间的版本冲突问题
- 统一开发结构：提供标准的、统一的项目结构

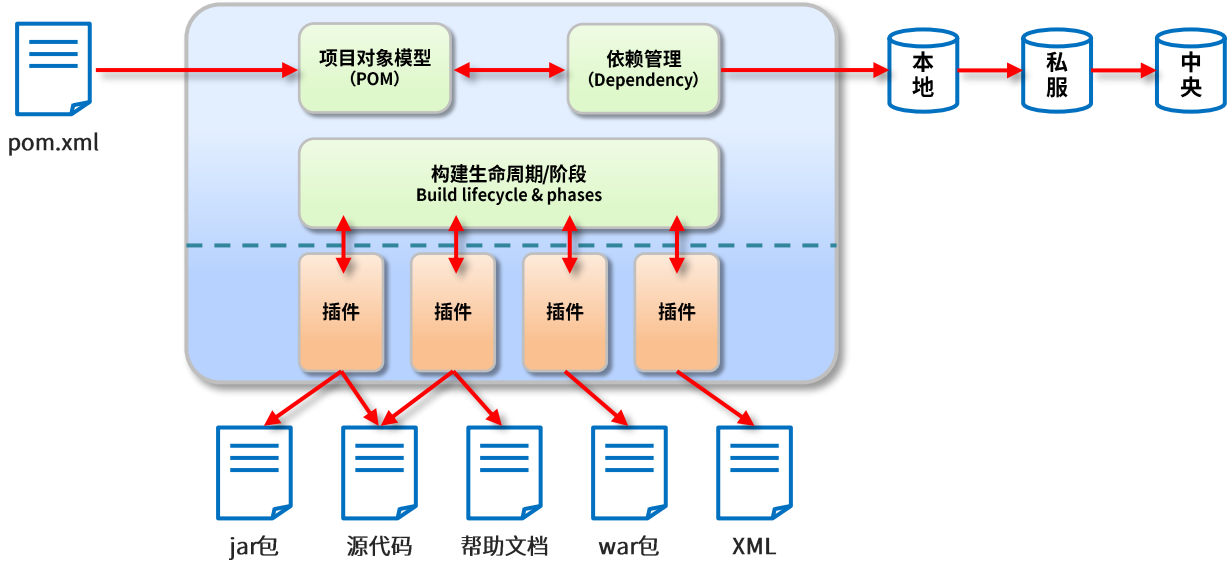


# Maven简介



## 小节

- Maven是什么
- Maven的作用
- POM



# 目录 Contents

# 下载与安装



## Maven下载

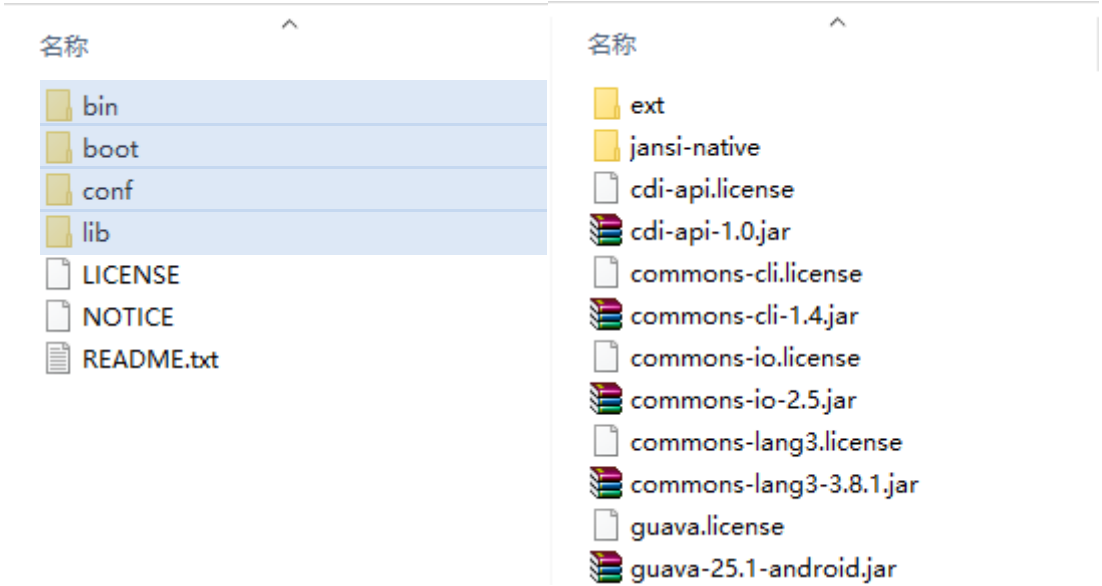
- 官网：http://maven.apache.org/
- 下载地址：http://maven.apache.org/download.cgi

# 下载与安装



## Maven安装

- Maven属于绿色版软件，解压即安装



## Maven环境变量配置

- 依赖Java，需要配置JAVA_HOME
- 设置MAVEN自身的运行环境，需要配置MAVEN_HOME
- 测试环境配置结果

```
MVN
```

# 下载与安装



## 小节

- 下载与安装
- 环境变量配置

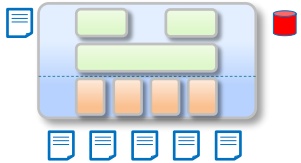
# 目录 Contents

# 目录 Contents

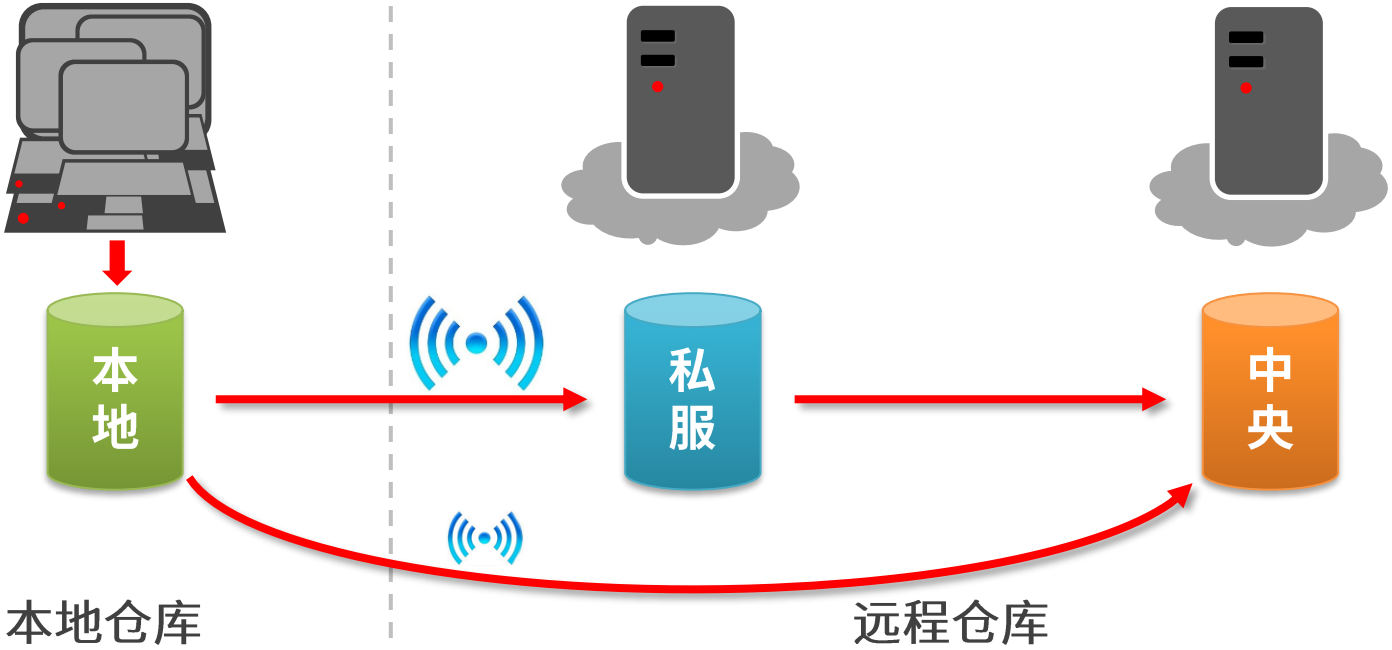
# Maven基础概念



## 仓库



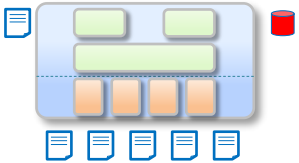
- 仓库：用于存储资源，包含各种jar包



# Maven基础概念



## 仓库

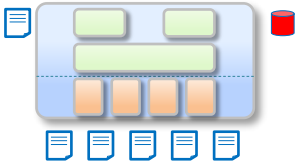


- 仓库：用于存储资源，包含各种jar包
- 仓库分类：
  - 本地仓库：自己电脑上存储资源的仓库，连接远程仓库获取资源
  - 远程仓库：非本机电脑上的仓库，为本地仓库提供资源
    - 中央仓库：Maven团队维护，存储所有资源的仓库
    - 私服：部门/公司范围内存储资源的仓库，从中央仓库获取资源
- 私服的作用：
  - 保存具有版权的资源，包含购买或自主研发的jar
    - 中央仓库中的jar都是开源的，不能存储具有版权的资源
  - 一定范围内共享资源，仅对内部开放，不对外共享

# Maven基础概念



## 小节

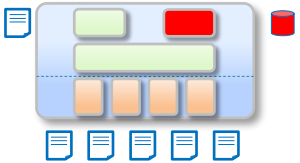


- 仓库的概念与作用
- 仓库的分类
  - 本地仓库
  - 远程仓库
    - 中央仓库
    - 私服

# Maven基础概念

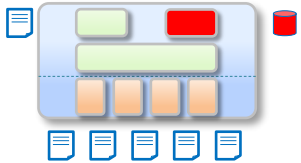


## 坐标



# Maven基础概念





## 坐标

- 什么是坐标?

  Maven中的坐标用于描述仓库中资源的位置

  https://repo1.maven.org/maven2/

- Maven坐标主要组成

  groupId：定义当前Maven项目隶属组织名称（通常是域名反写，例如：org.mybatis）

  artifactId：定义当前Maven项目名称（通常是模块名称，例如CRM、SMS）

  version：定义当前项目版本号

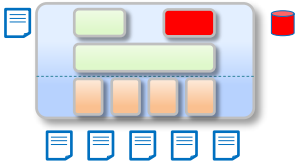
  packaging：定义该项目的打包方式

- Maven坐标的作用

  使用唯一标识，唯一性定位资源位置，通过该标识可以将资源的识别与下载工作交由机器完成

# Maven基础概念





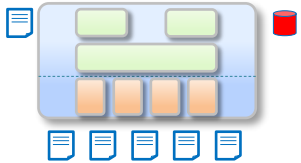
## 小节

- 坐标的概念与作用
- 坐标的组成
  - 组织ID
  - 项目ID
  - 版本号

# Maven基础概念



## 本地仓库配置



- Maven启动后，会自动保存下载的资源到本地仓库
  - 默认位置

```
<localRepository>${user.home}/.m2/repository</localRepository>
```

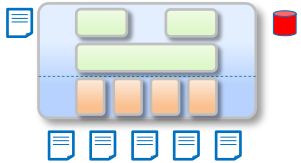
当前目录位置为登录用户名所在目录下的.m2文件夹中

  - 自定义位置

```
<localRepository>D:\maven\repository</localRepository>
```

当前目录位置为D:\maven\repository文件夹中

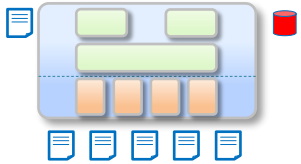
## 远程仓库配置



- Maven默认连接的仓库位置

```
<repositories>
    <repository>
        <id>central</id>
        <name>Central Repository</name>
        <url>https://repo.maven.apache.org/maven2</url>
        <layout>default</layout>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
```

## 镜像仓库配置



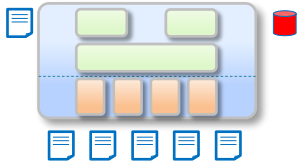
- 在settting文件中配置阿里云镜像仓库

```
<mirrors>
    <!--配置具体的仓库的下载镜像  -->
    <mirror>
        <!-- 此镜像的唯一标识符，用来区分不同的mirror元素  -->
        <id>nexus-aliyun</id>
        <!-- 对哪种仓库进行镜像，简单说就是替代哪个仓库 -->
        <mirrorOf>central</mirrorOf>
        <!-- 镜像名称  -->
        <name>Nexus aliyun</name>
        <!-- 镜像URL  -->
        <url>http://maven.aliyun.com/nexus/content/groups/public</url>
    </mirror>
</mirrors>
```

# Maven基础概念



## 全局setting与用户setting区别

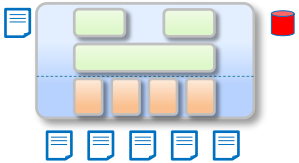


- 全局setttting定义了当前计算器中Maven的公共配置
- 用户setttting定义了当前用户的配置

# Maven基础概念



## 小节



- 配置本地仓库（资源下到哪）
- 配置阿里镜像仓库（资源从哪来）
- settting文件的区别

# 目录 Contents

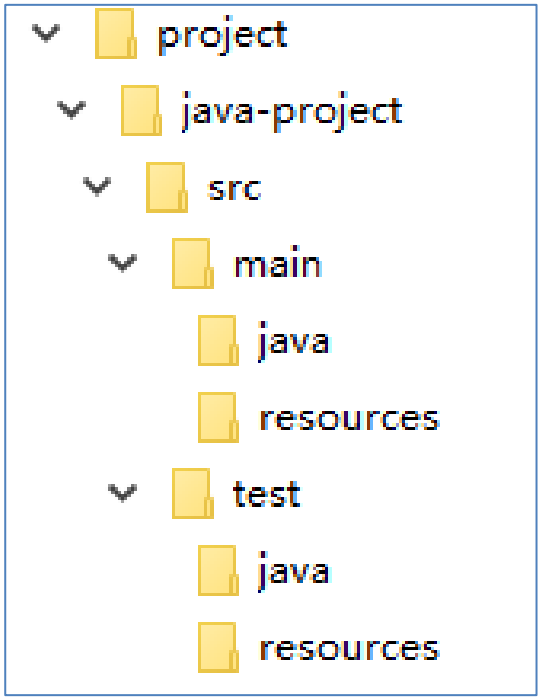
# 目录 Contents

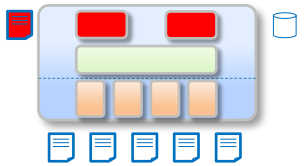
- 第一个Maven项目（手工制作）
  - Maven工程目录结构
  - 构建命令
  - 插件创建工程

# 第一个Maven项目（手工制作）



## Maven工程目录结构

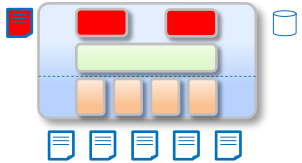


# 第一个Maven项目（手工制作）



## Maven工程目录结构



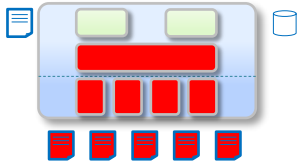
- 在src同层目录下创建pom.xml

```
<?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.itheima</groupId>
    <artifactId>project-java</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
</project>
```

# 第一个Maven项目（手工制作）



## Maven项目构建命令



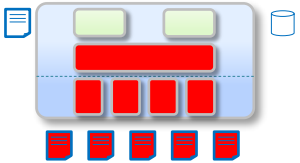
- Maven构建命令使用mvn开头，后面添加功能参数，可以一次执行多个命令，使用空格分隔

```
mvn compile          #编译
mvn clean            #清理
mvn test             #测试
mvn package          #打包
mvn install          #安装到本地仓库
```

# 第一个Maven项目（手工制作）



## 小节

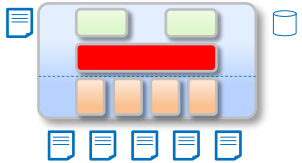


- Maven工程目录结构
- Maven常用项目构建指令

# 第一个Maven项目（手工制作）



## 插件创建工程



- 创建工程

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

- 创建java工程

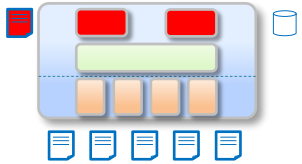
```
mvn archetype:generate -DgroupId=com.itheima -DartifactId=java-project -
DarchetypeArtifactId=maven-archetype-quickstart -Dversion=0.0.1-snapshot -
DinteractiveMode=false
```

- 创建web工程

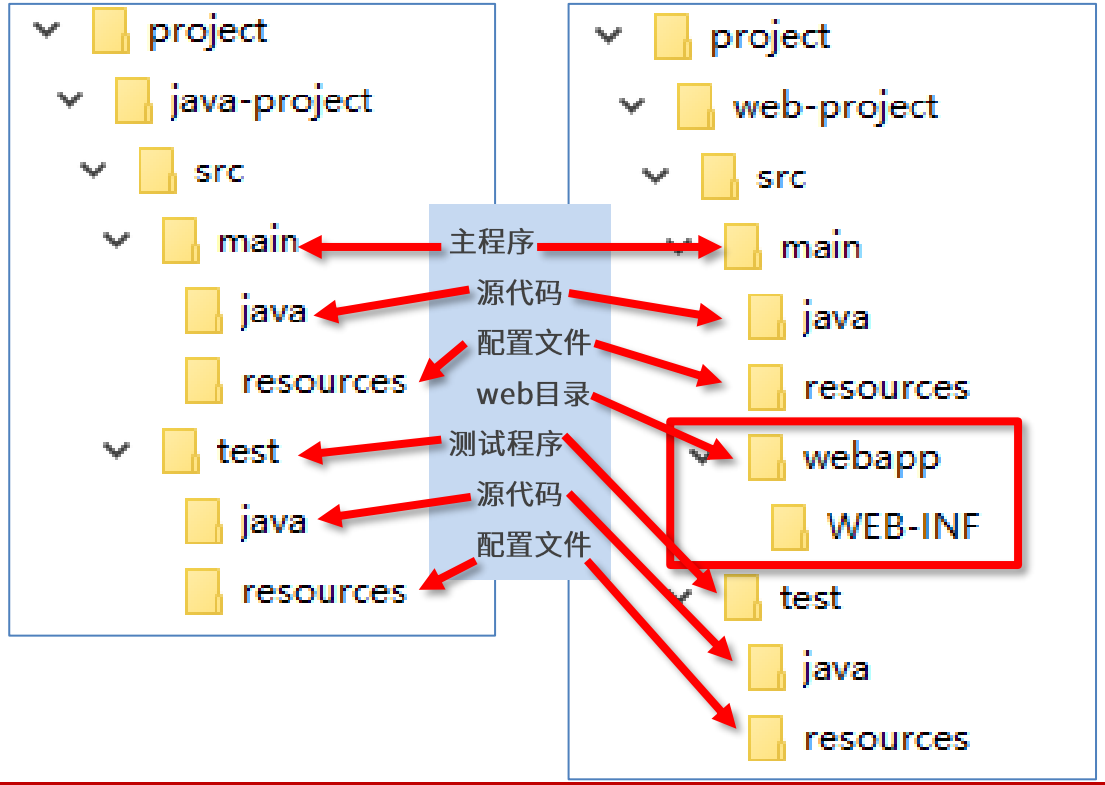
```
mvn archetype:generate -DgroupId=com.itheima -DartifactId=web-project -
DarchetypeArtifactId=maven-archetype-webapp -Dversion=0.0.1-snapshot -
DinteractiveMode=false
```

# 第一个Maven项目（手工制作）





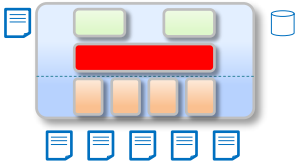
## Maven工程目录结构



# 第一个Maven项目（手工制作）



## 小节



- 插件创建Maven工程
- Maven　web工程目录结构

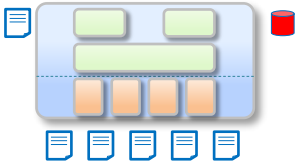
# 目录 Contents

# 目录 Contents

# 第一个Maven项目（IDEA生成）



## 配置Maven

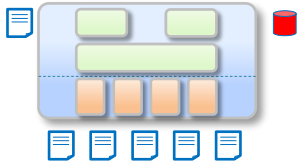


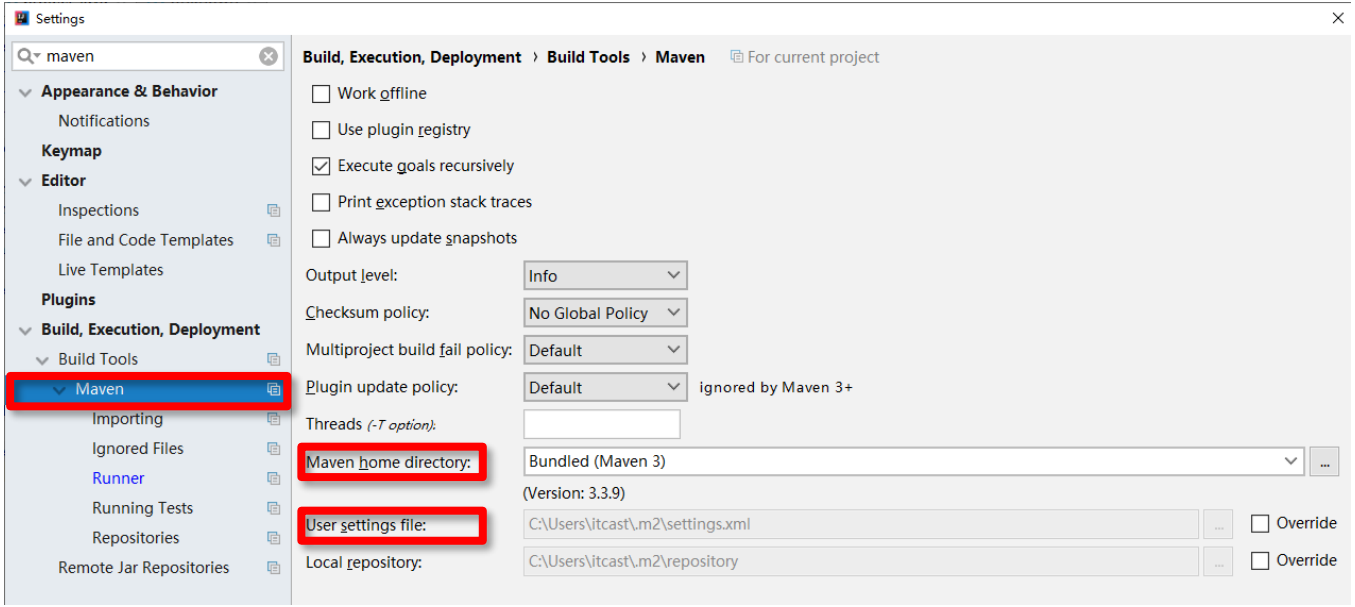
- Idea对3.6.2及以上版本存在兼容性问题，为避免冲突，IDEA中安装使用3.6.1版本

# 第一个Maven项目（IDEA生成）



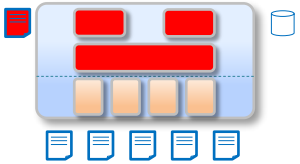
## 配置Maven



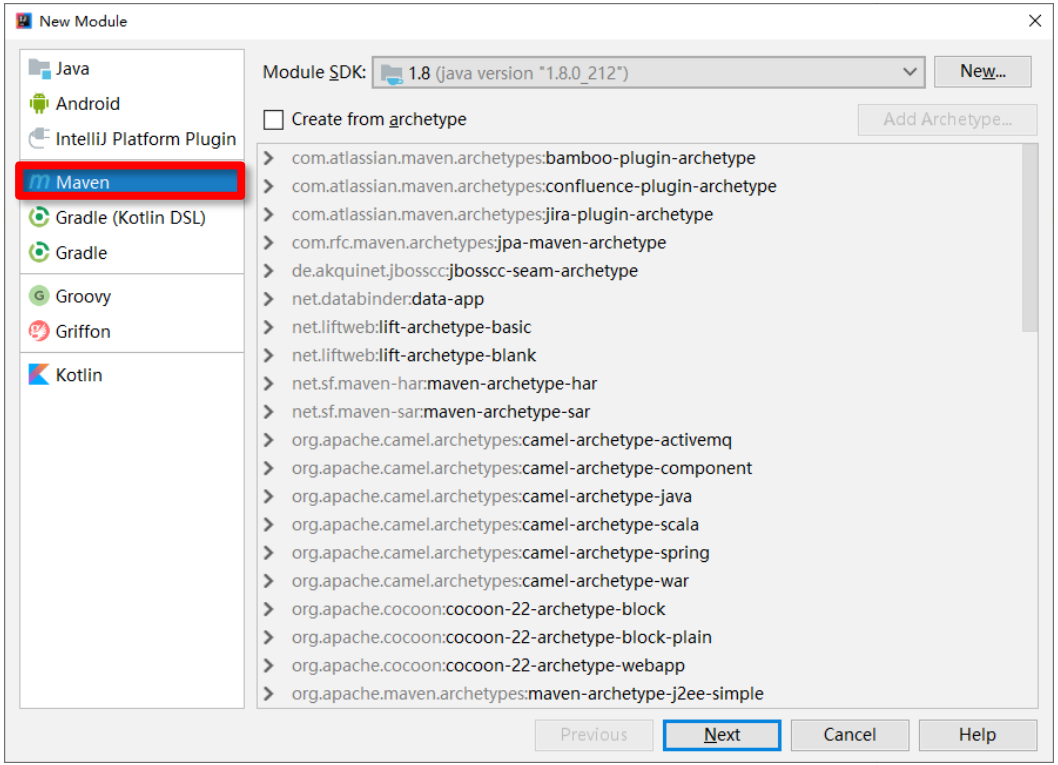


# 第一个Maven项目（IDEA生成）





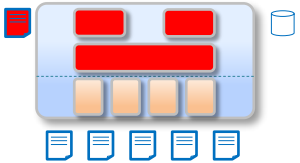
## 手工创建Java项目



# 第一个Maven项目（IDEA生成）



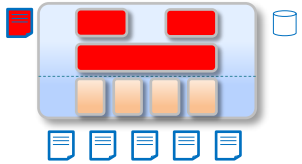
## 小节



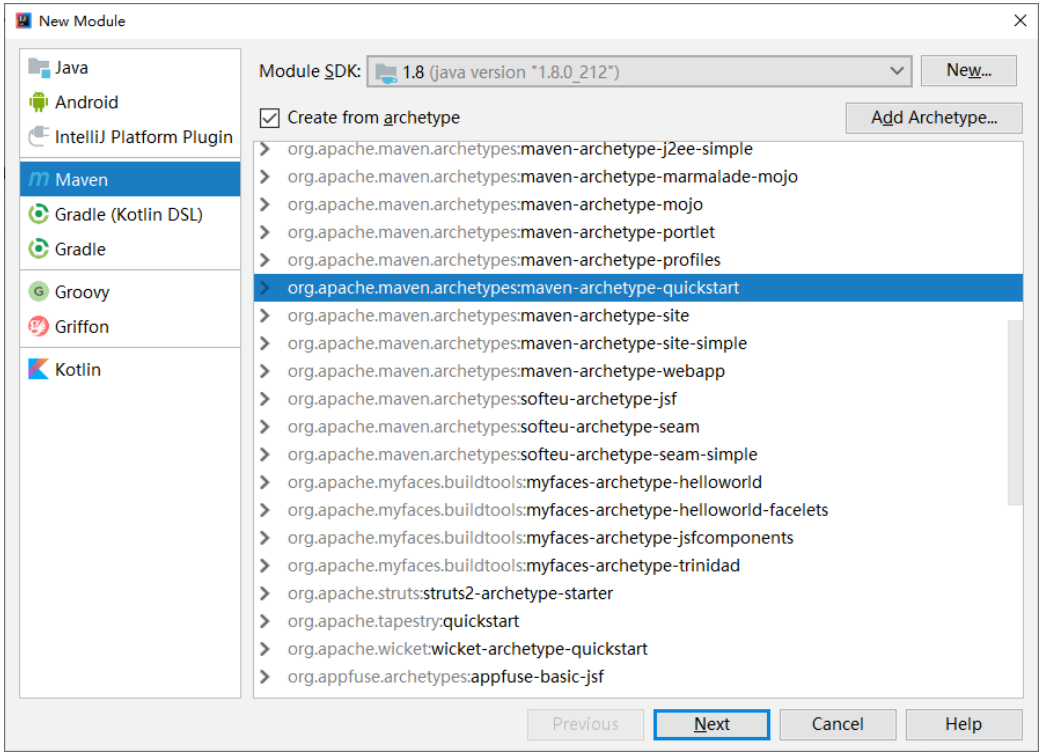
- Maven环境配置
- Maven项目创建
- Maven命令执行

# 第一个Maven项目（IDEA生成）



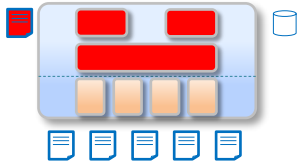


## 原型创建Java项目

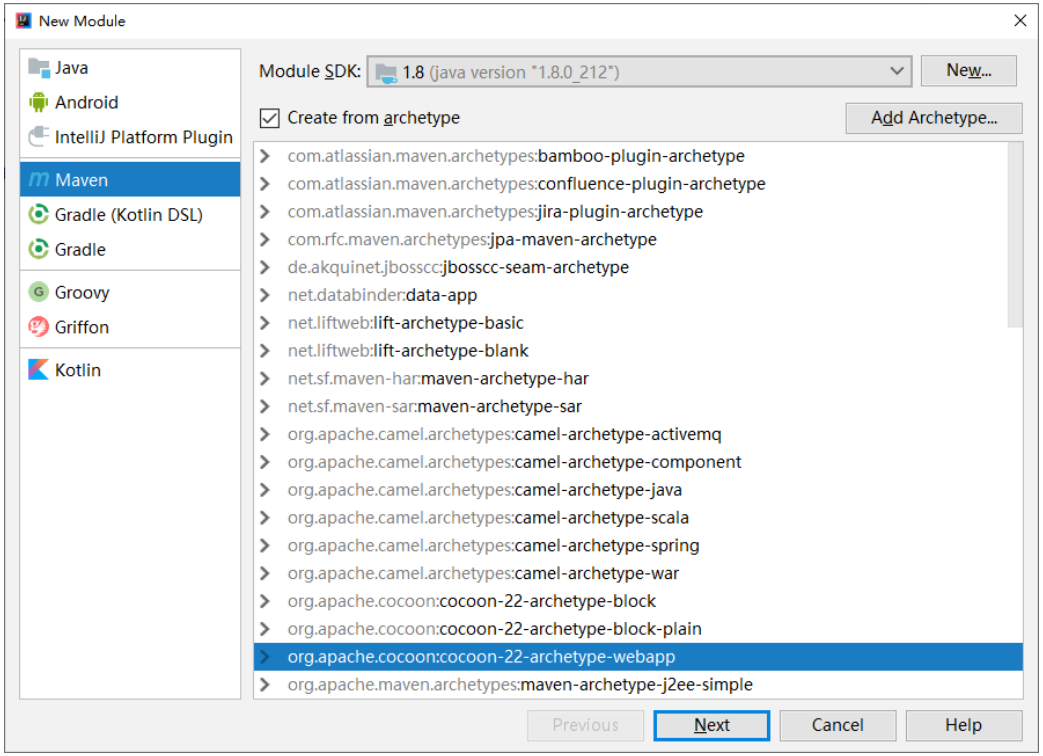


# 第一个Maven项目（IDEA生成）





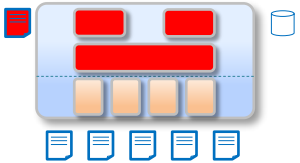
## 原型创建Web项目



# 第一个Maven项目（IDEA生成）



## 小节

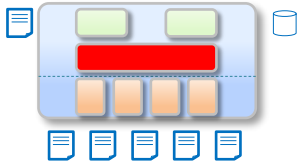


- 使用原型创建java项目
- 使用原型创建web项目

# 第一个Maven项目（IDEA生成）



## 插件



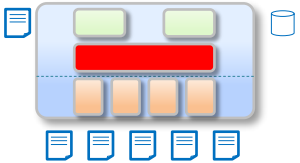
- Tomcat7运行插件

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.tomcat.maven</groupId>
            <artifactId>tomcat7-maven-plugin</artifactId>
            <version>2.1</version>
            <configuration>
                <port>80</port>
                <path>/</path>
            </configuration>
        </plugin>
    </plugins>
</build>
```

# 第一个Maven项目（IDEA生成）



## 小节



- tomcat7插件安装
- 运行web项目

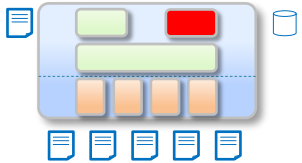
# 目录 Contents

# 目录 Contents

# 依赖管理



## 依赖配置

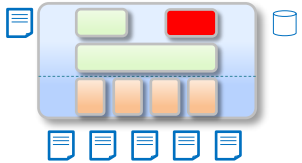


- 依赖指当前项目运行所需的jar，一个项目可以设置多个依赖
- 格式：

```
<!--设置当前项目所依赖的所有jar-->
<dependencies>
  <!--设置具体的依赖-->
  <dependency>
    <!--依赖所属群组id-->
    <groupId>junit</groupId>
    <!--依赖所属项目id-->
    <artifactId>junit</artifactId>
    <!--依赖版本号-->
    <version>4.12</version>
  </dependency>
</dependencies>
```

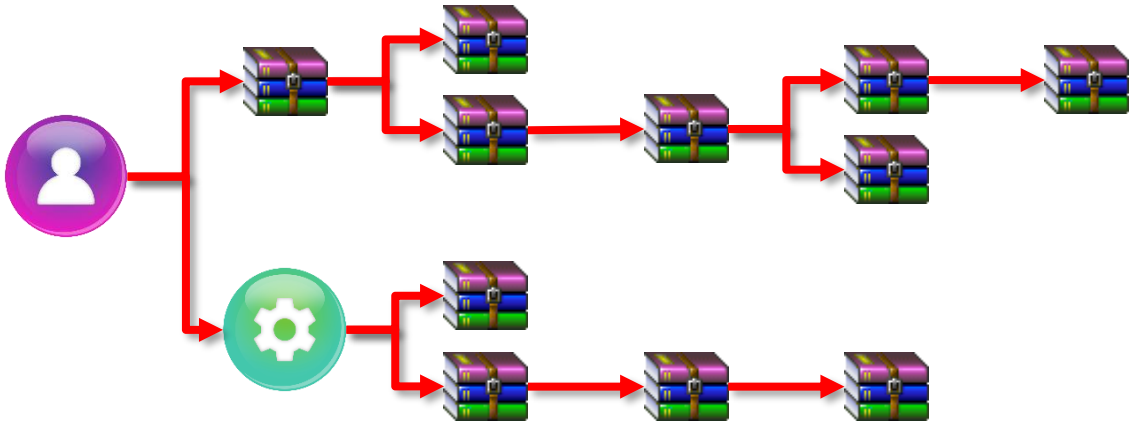
# 依赖管理





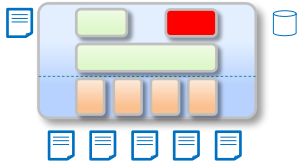
## 依赖传递

- 依赖具有传递性
  - 直接依赖：在当前项目中通过依赖配置建立的依赖关系
  - 间接依赖：被资源的资源如果依赖其他资源，当前项目间接依赖其他资源



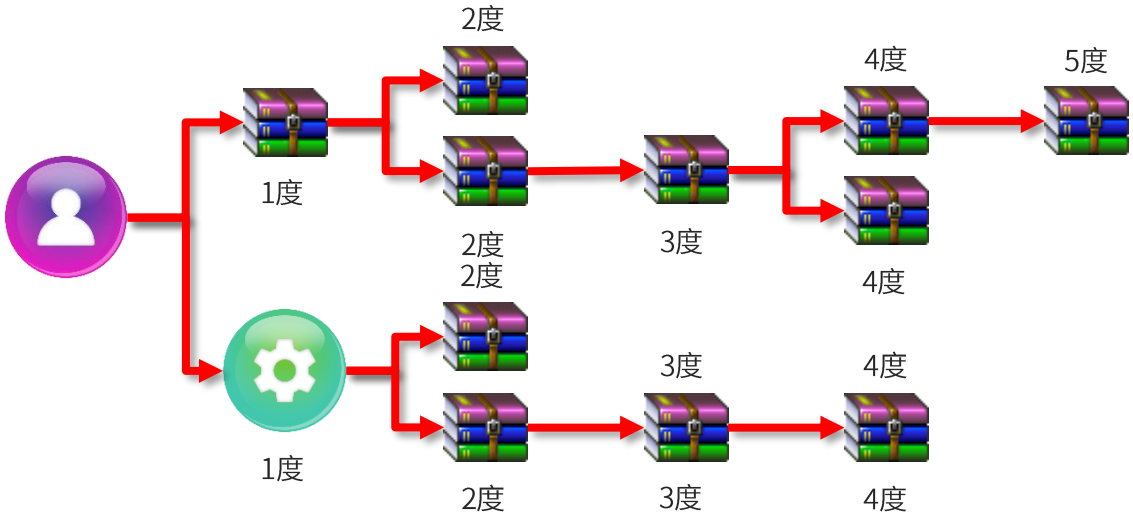
# 依赖管理





## 依赖传递冲突问题

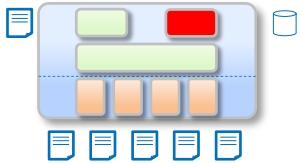
- 路径优先：当依赖中出现相同的资源时，层级越深，优先级越低，层级越浅，优先级越高
- 声明优先：当资源在相同层级被依赖时，配置顺序靠前的覆盖配置顺序靠后的
- 特殊优先：当同级配置了相同资源的不同版本，后配置的覆盖先配置的



# 依赖管理



## 可选依赖



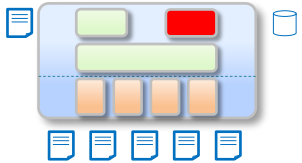
- 可选依赖指对外隐藏当前所依赖的资源——不透明

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <optional>true</optional>
</dependency>
```

# 依赖管理



## 排除依赖



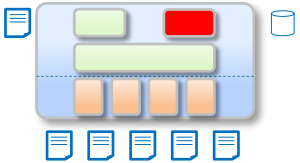
- 排除依赖指主动断开依赖的资源，被排除的资源无需指定版本——不需要

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <exclusions>
    <exclusion>
      <groupId>org.hamcrest</groupId>
      <artifactId>hamcrest-core</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

# 依赖管理



## 小节

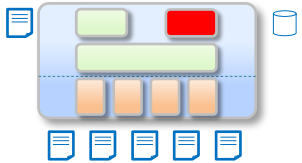


- 依赖管理
- 依赖传递
- 可选依赖（不透明）
- 排除依赖（不需要）

# 依赖管理



## 依赖范围



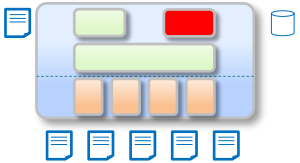
- 依赖的jar默认情况可以在任何地方使用，可以通过scope标签设定其作用范围
- 作用范围
  - 主程序范围有效（main文件夹范围内）
  - 测试程序范围有效（test文件夹范围内）
  - 是否参与打包（package指令范围内）

| scope | 主代码 | 测试代码 | 打包 | 范例 |
| --- | --- | --- | --- | --- |
| compile(默认) | Y | Y | Y | log4j |
| test | | Y | | junit |
| provided | Y | Y | | servlet-api |
| runtime | | | Y | jdbc |

# 依赖管理



## 依赖范围传递性



- 带有依赖范围的资源在进行传递时，作用范围将受到影响

| | compile | test | provided | runtime |
|---|---|---|---|---|
| compile | compile | test | provided | runtime |
| test | | | | |
| provided | | | | |
| runtime | runtime | test | provided | runtime |

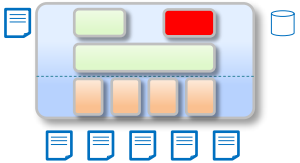
直接依赖

间接依赖

# 依赖管理



## 小节



- 依赖范围
- 依赖范围传递性（了解）

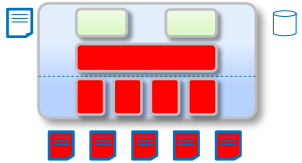
# 目录 Contents

# 目录 Contents

# 生命周期与插件



## 项目构建生命周期

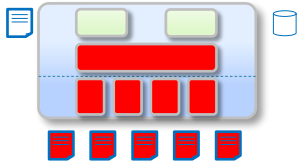


- Maven构建生命周期描述的是一次构建过程经历经历了多少个事件

compile → test-compile → test → package → install

# 生命周期与插件





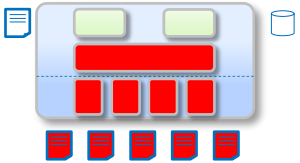
## 项目构建生命周期

- Maven对项目构建的生命周期划分为3套
  - clean：清理工作
  - default：核心工作，例如编译，测试，打包，部署等
  - site：产生报告，发布站点等

# 生命周期与插件



## clean生命周期

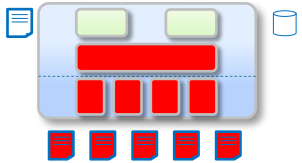


- pre-clean　　执行一些需要在clean之前完成的工作
- clean　　移除所有上一次构建生成的文件
- post-clean　　执行一些需要在clean之后立刻完成的工作

# 生命周期与插件



## default构建生命周期

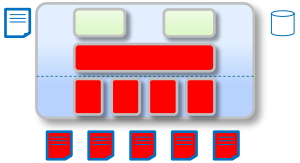


| | |
|---|---|
| validate（校验） | 校验项目是否正确并且所有必要的信息可以完成项目的构建过程。 |
| initialize（初始化） | 初始化构建状态，比如设置属性值。 |
| generate-sources（生成源代码） | 生成包含在编译阶段中的任何源代码。 |
| process-sources（处理源代码） | 处理源代码，比如说，过滤任意值。 |
| generate-resources（生成资源文件） | 生成将会包含在项目包中的资源文件。 |
| process-resources （处理资源文件） | 复制和处理资源到目标目录，为打包阶段最好准备。 |
| **compile**（编译） | 编译项目的源代码。 |
| process-classes（处理类文件） | 处理编译生成的文件，比如说对Java class文件做字节码改善优化。 |
| generate-test-sources（生成测试源代码） | 生成包含在编译阶段中的任何测试源代码。 |
| process-test-sources（处理测试源代码） | 处理测试源代码，比如说，过滤任意值。 |
| generate-test-resources（生成测试资源文件） | 为测试创建资源文件。 |
| process-test-resources（处理测试资源文件） | 复制和处理测试资源到目标目录。 |
| **test-compile**（编译测试源码） | 编译测试源代码到测试目标目录. |
| process-test-classes（处理测试类文件） | 处理测试源码编译生成的文件。 |
| **test**（测试） | 使用合适的单元测试框架运行测试（Juint是其中之一）。 |
| prepare-package（准备打包） | 在实际打包之前，执行任何的必要的操作为打包做准备。 |
| **package**（打包） | 将编译后的代码打包成可分发格式的文件，比如JAR、WAR或者EAR文件。 |
| pre-integration-test（集成测试前） | 在执行集成测试前进行必要的动作。比如说，搭建需要的环境。 |
| integration-test（集成测试） | 处理和部署项目到可以运行集成测试环境中。 |
| post-integration-test（集成测试后） | 在执行集成测试完成后进行必要的动作。比如说，清理集成测试环境。 |
| verify （验证） | 运行任意的检查来验证项目包有效且达到质量标准。 |
| **install**（安装） | 安装项目包到本地仓库，这样项目包可以用作其他本地项目的依赖。 |
| deploy（部署） | 将最终的项目包复制到远程仓库中与其他开发者和项目共享。 |

# 生命周期与插件



## site构建生命周期

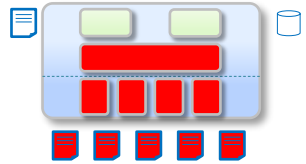


- pre-site　　执行一些需要在生成站点文档之前完成的工作
- site　　生成项目的站点文档
- post-site　　执行一些需要在生成站点文档之后完成的工作，并且为部署做准备
- site-deploy　　将生成的站点文档部署到特定的服务器上

# 生命周期与插件



## default构建生命周期

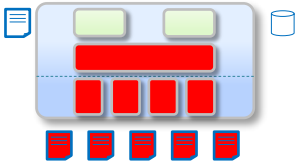


| | |
|---|---|
| validate（校验） | 校验项目是否正确并且所有必要的信息可以完成项目的构建过程。 |
| initialize（初始化） | 初始化构建状态，比如设置属性值。 |
| generate-sources（生成源代码） | 生成包含在编译阶段中的任何源代码。 |
| process-sources（处理源代码） | 处理源代码，比如说，过滤任意值。 |
| generate-resources（生成资源文件） | 生成将会包含在项目包中的资源文件。 |
| process-resources （处理资源文件） | 复制和处理资源到目标目录，为打包阶段最好准备。 |
| compile（编译） | 编译项目的源代码。 |
| process-classes（处理类文件） | 处理编译生成的文件，比如说对Java class文件做字节码改善优化。 |
| generate-test-sources（生成测试源代码） | 生成包含在编译阶段中的任何测试源代码。 |
| process-test-sources（处理测试源代码） | 处理测试源代码，比如说，过滤任意值。 |
| generate-test-resources（生成测试资源文件） | 为测试创建资源文件。 |
| process-test-resources（处理测试资源文件） | 复制和处理测试资源到目标目录。 |
| test-compile（编译测试源码） | 编译测试源代码到测试目标目录. |
| process-test-classes（处理测试类文件） | 处理测试源码编译生成的文件。 |
| test（测试） | 使用合适的单元测试框架运行测试（Juint是其中之一）。 |
| prepare-package（准备打包） | 在实际打包之前，执行任何的必要的操作为打包做准备。 |
| package（打包） | 将编译后的代码打包成可分发格式的文件，比如JAR、WAR或者EAR文件。 |
| pre-integration-test（集成测试前） | 在执行集成测试前进行必要的动作。比如说，搭建需要的环境。 |
| integration-test（集成测试） | 处理和部署项目到可以运行集成测试环境中。 |
| post-integration-test（集成测试后） | 在执行集成测试完成后进行必要的动作。比如说，清理集成测试环境。 |
| verify （验证） | 运行任意的检查来验证项目包有效且达到质量标准。 |
| install（安装） | 安装项目包到本地仓库，这样项目包可以用作其他本地项目的依赖。 |
| deploy（部署） | 将最终的项目包复制到远程仓库中与其他开发者和项目共享。 |

# 生命周期与插件



## 插件

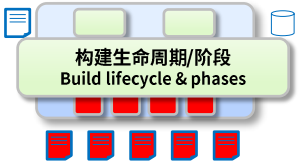


- 插件与生命周期内的阶段绑定，在执行到对应生命周期时执行对应的插件功能
- 默认maven在各个生命周期上绑定有预设的功能
- 通过插件可以自定义其他功能

# 生命周期与插件



## 插件



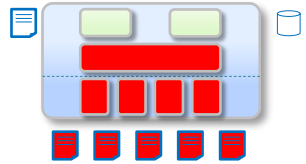

- 插件与生命周期内的阶段绑定，在执行到对应生命周期时执行对应的插件功能
- 默认maven在各个生命周期上绑定有预设的功能
- 通过插件可以自定义其他功能

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-source-plugin</artifactId>
            <version>2.2.1</version>
            <executions>
                <execution>
                    <goals>
                        <goal>jar</goal>
                    </goals>
                    <phase>generate-test-resources</phase>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

# 生命周期与插件



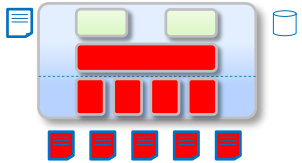
## default构建生命周期



| | |
|---|---|
| validate（校验） | 校验项目是否正确并且所有必要的信息可以完成项目的构建过程。 |
| initialize（初始化） | 初始化构建状态，比如设置属性值。 |
| generate-sources（生成源代码） | 生成包含在编译阶段中的任何源代码。 |
| process-sources（处理源代码） | 处理源代码，比如说，过滤任意值。 |
| generate-resources（生成资源文件） | 生成将会包含在项目包中的资源文件。 |
| process-resources （处理资源文件） | 复制和处理资源到目标目录，为打包阶段最好准备。 |
| compile（编译） | 编译项目的源代码。 |
| process-classes（处理类文件） | 处理编译生成的文件，比如说对Java class文件做字节码改善优化。 |
| generate-test-sources（生成测试源代码） | 生成包含在编译阶段中的任何测试源代码。 |
| process-test-sources（处理测试源代码） | 处理测试源代码，比如说，过滤任意值。 |
| generate-test-resources（生成测试资源文件） | 为测试创建资源文件。 |
| process-test-resources（处理测试资源文件） | 复制和处理测试资源到目标目录。 |
| test-compile（编译测试源码） | 编译测试源代码到测试目标目录. |
| process-test-classes（处理测试类文件） | 处理测试源码编译生成的文件。 |
| test（测试） | 使用合适的单元测试框架运行测试（Juint是其中之一）。 |
| prepare-package（准备打包） | 在实际打包之前，执行任何的必要的操作为打包做准备。 |
| package（打包） | 将编译后的代码打包成可分发格式的文件，比如JAR、WAR或者EAR文件。 |
| pre-integration-test（集成测试前） | 在执行集成测试前进行必要的动作。比如说，搭建需要的环境。 |
| integration-test（集成测试） | 处理和部署项目到可以运行集成测试环境中。 |
| post-integration-test（集成测试后） | 在执行集成测试完成后进行必要的动作。比如说，清理集成测试环境。 |
| verify （验证） | 运行任意的检查来验证项目包有效且达到质量标准。 |
| install（安装） | 安装项目包到本地仓库，这样项目包可以用作其他本地项目的依赖。 |
| deploy（部署） | 将最终的项目包复制到远程仓库中与其他开发者和项目共享。 |

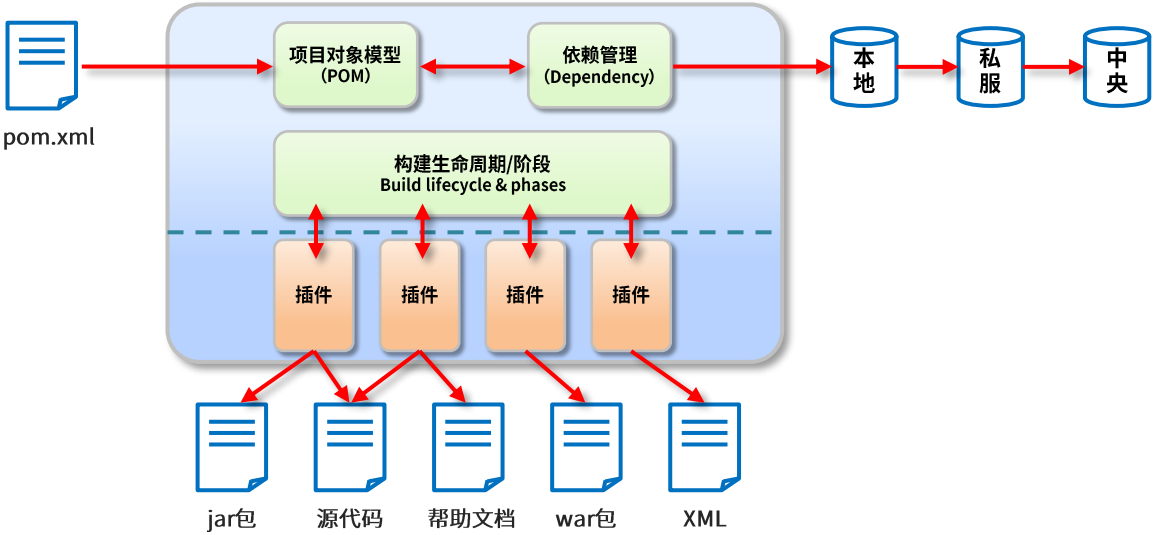
# 生命周期与插件





## 小节

- 生命周期
- 插件




## Maven基础

- Maven简介
- 下载与安装
- Maven基础概念
- 第一个Maven项目（手工制作）
- 第一个Maven项目（IDEA生成）
- 依赖管理
- 生命周期与插件


传智播客旗下高端IT教育品牌


# Maven高级

# 目录 Contents

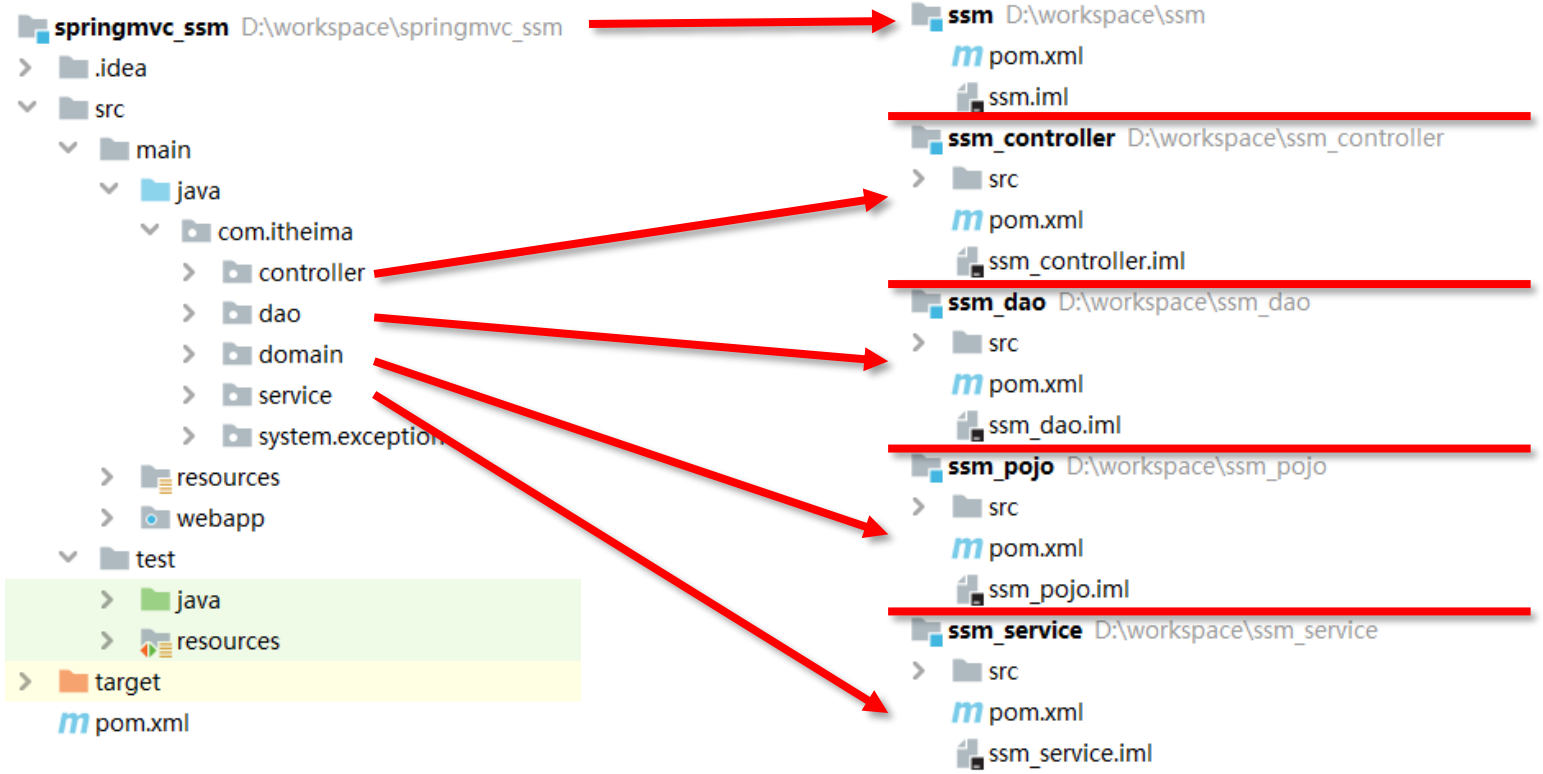
# 目录 Contents

# 目录 Contents

# 分模块开发与设计



## 工程模块与模块划分



# 分模块开发与设计



## ssm_pojo拆分

- 新建模块
- 拷贝原始项目中对应的相关内容到ssm_pojo模块中
  - 实体类（User）
  - 配置文件（无）

# 分模块开发与设计



## ssm_dao拆分

- 新建模块
- 拷贝原始项目中对应的相关内容到ssm_dao模块中
  - 数据层接口（UserDao）
  - 配置文件：保留与数据层相关配置文件(3个)
    - 注意：分页插件在配置中与SqlSessionFactoryBean绑定，需要保留
  - pom.xml：引入数据层相关坐标即可，删除springmvc相关坐标
    - spring
    - mybatis
    - spring 整合mybatis
    - mysql
    - druid
    - pagehelper
    - 直接依赖ssm_pojo（对ssm_pojo模块执行install指令，将其安装到本地仓库）

## ssm_service拆分

- 新建模块
- 拷贝原始项目中对应的相关内容到ssm_service模块中
  - 业务层接口与实现类（UserService、UserServiceImpl）
  - 配置文件：保留与数据层相关配置文件(1个)
  - pom.xml：引入数据层相关坐标即可，删除springmvc相关坐标
    - spring
    - junit
    - spring 整合junit
    - 直接依赖ssm_dao（对ssm_dao模块执行install指令，将其安装到本地仓库）
    - 间接依赖ssm_pojo（由ssm_dao模块负责依赖关系的建立）
  - 修改service模块spring核心配置文件名，添加模块名称，格式：applicationContext-service.xml
  - 修改dao模块spring核心配置文件名，添加模块名称，格式：applicationContext-dao.xml
  - 修改单元测试引入的配置文件名称，由单个文件修改为多个文件

## ssm_control拆分

- 新建模块（使用webapp模板）
- 拷贝原始项目中对应的相关内容到ssm_controller模块中
  - 表现层控制器类与相关设置类（UserController、异常相关……）
  - 配置文件：保留与表现层相关配置文件(1个）、服务器相关配置文件（1个）
  - pom.xml：引入数据层相关坐标即可，删除springmvc相关坐标
    - spring
    - springmvc
    - jackson
    - servlet
    - tomcat服务器插件
    - 直接依赖ssm_service（对ssm_service模块执行install指令，将其安装到本地仓库）
    - 间接依赖ssm_dao、ssm_pojo
  - 修改web.xml配置文件中加载spring环境的配置文件名称，使用*通配，加载所有applicationContext-开始的配置文件

# 分模块开发与设计



## 小节

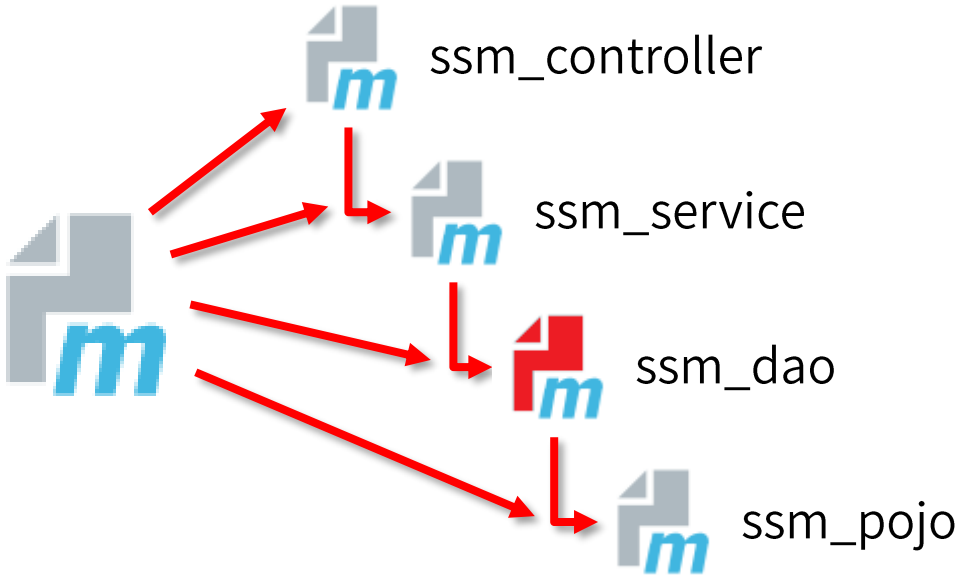
- 分模块开发
  - 模块中仅包含当前模块对应的功能类与配置文件
  - spring核心配置根据模块功能不同进行独立制作
  - 当前模块所依赖的模块通过导入坐标的形式加入当前模块后才可以使用
  - web.xml需要加载所有的spring核心配置文件

# 目录 Contents

# 聚合



## 多模块构建维护




本地仓库

# 聚合

- 作用：聚合用于快速构建maven工程，一次性构建多个项目/模块。
- 制作方式：
  - 创建一个空模块，打包类型定义为pom

```
<packaging>pom</packaging>
```

  - 定义当前模块进行构建操作时关联的其他模块名称

```
<modules>
    <module>../ssm_controller</module>
    <module>../ssm_service</module>
    <module>../ssm_dao</module>
    <module>../ssm_pojo</module>
</modules>
```

- 注意事项：参与聚合操作的模块最终执行顺序与模块间的依赖关系有关，与配置顺序无关

# 聚合



## 小节

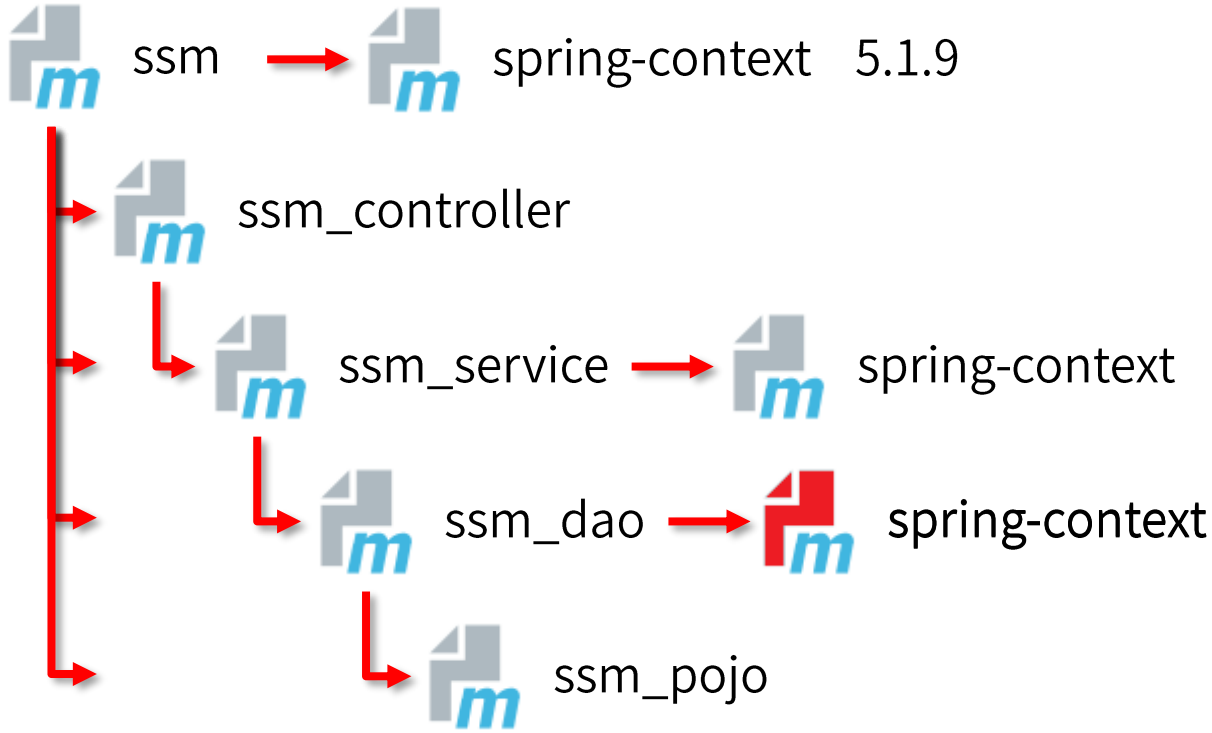
- 聚合的作用
- 聚合的配置方式
- 模块的类型
  - pom
  - war
  - jar

# 目录 Contents

# 继承



## 模块依赖关系维护



## 继承

- 作用：通过继承可以实现在子工程中沿用父工程中的配置
  - maven中的继承与java中的继承相似，在子工程中配置继承关系
- 制作方式：
  - 在子工程中声明其父工程坐标与对应的位置

```
<!--定义该工程的父工程-->
<parent>
    <groupId>com.itheima</groupId>
    <artifactId>ssm</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!--填写父工程的pom文件-->
    <relativePath>../ssm/pom.xml</relativePath>
</parent>
```

## 继承依赖定义

- 在父工程中定义依赖管理

```
<!--声明此处进行依赖管理-->
<dependencyManagement>
    <!--具体的依赖-->
    <dependencies>
        <!--spring环境-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.9.RELEASE</version>
        </dependency>
    <dependencies>
<dependencyManagement>
```

## 继承依赖使用

- 在子工程中定义依赖关系，无需声明依赖版本，版本参照父工程中依赖的版本

```
<dependencies>
    <!--spring环境-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
    </dependency>
</dependencies>
```

## 继承的资源

- groupId：项目组ID，项目坐标的核心元素
- version：项目版本，项目坐标的核心因素
- 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：项目的贡献者信息
- distributionManagement：项目的部署配置
- issueManagement：项目的缺陷跟踪系统信息
- ciManagement：项目的持续集成系统信息
- scm：项目的版本控制系统西溪
- malilingLists：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

## 继承与聚合

- 作用
  - 聚合用于快速构建项目
  - 继承用于快速配置
- 相同点：
  - 聚合与继承的pom.xml文件打包方式均为pom，可以将两种关系制作到同一个pom文件中
  - 聚合与继承均属于设计型模块，并无实际的模块内容
- 不同点：
  - 聚合是在当前模块中配置关系，聚合可以感知到参与聚合的模块有哪些
  - 继承是在子模块中配置关系，父模块无法感知哪些子模块继承了自己

# 继承



## 小节

- 继承的作用
- 继承的内容
- 依赖继承
  - 定义在父工程中
  - 使用在子工程中（无需配置version）

# 目录 Contents

## 版本统一的重要性

```xml
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.1.9.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.5.3</version>
</dependency>
<!--spring整合jdbc-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>5.2.1.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.2.0.RELEASE</version>
</dependency>
```

```java
String name = "Jock";
System.out.println(name);
```

```java
String spring_version = "5.1.9.RELEASE";
System.out.println(spring_version);
```

# 属性



## 属性类别

1. 自定义属性
2. 内置属性
3. Setting属性
4. Java系统属性
5. 环境变量属性

## 属性类别：自定义属性

- 作用
  - 等同于定义变量，方便统一维护
- 定义格式：

```
<!--定义自定义属性-->
<properties>
    <spring.version>5.1.9.RELEASE</spring.version>
    <junit.version>4.12</junit.version>
</properties>
```

- 调用格式：

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

## 属性类别：内置属性

- 作用
  - 使用maven内置属性，快速配置
- 调用格式：

```
${basedir}
${version}
```

## 属性类别：Setting属性

- 作用
  - 使用Maven配置文件setting.xml中的标签属性，用于动态配置
- 调用格式：

```
${settings.localRepository}
```

## 属性类别：Java系统属性

- 作用
  - 读取Java系统属性
- 调用格式

```
${user.home}
```

- 系统属性查询方式

```
mvn help:system
```

## 属性类别：环境变量属性

- 作用
  - 使用Maven配置文件setting.xml中的标签属性，用于动态配置
- 调用格式

```
${env.JAVA_HOME}
```

- 环境变量属性查询方式

```
mvn help:system
```

# 属性



## 小节

- 属性的作用
- 五种属性值获取方式
- 自定义属性配置与使用方式

# 目录 Contents

# 版本管理



## 工程版本区分


1.0-SNAPSHOT


1.1-SNAPSHOT


1.2-SNAPSHOT


2.0-SNAPSHOT


2.1-SNAPSHOT

# 版本管理



## 工程版本

- SNAPSHOT（快照版本）
  - 项目开发过程中，为方便团队成员合作，解决模块间相互依赖和时时更新的问题，开发者对每个模块进行构建的时候，输出的临时性版本叫快照版本（测试阶段版本）
  - 快照版本会随着开发的进展不断更新
- RELEASE（发布版本）
  - 项目开发到进入阶段里程碑后，向团队外部发布较为稳定的版本，这种版本所对应的构件文件是稳定的，即便进行功能的后续开发，也不会改变当前发布版本内容，这种版本称为发布版本

# 版本管理



## 工程版本号约定

- 约定规范：
  - <主版本>.<次版本>.<增量版本>.<里程碑版本>
  - 主版本：表示项目重大架构的变更，如：spring5相较于spring4的迭代
  - 次版本：表示有较大的功能增加和变化，或者全面系统地修复漏洞
  - 增量版本：表示有重大漏洞的修复
  - 里程碑版本：表明一个版本的里程碑（版本内部）。这样的版本同下一个正式版本相比，相对来说不是很稳定，有待更多的测试
- 范例：
  - 5.1.9.RELEASE

# 版本管理



## 小节

- 工程版本
  - RELEASE
  - SNAPSHOT

# 目录 Contents

# 资源配置



## 资源配置多文件维护

```
<!--定义自定义属性-->
<properties>
    <spring.version>5.1.9.RELEASE</spring.version>
    <junit.version>4.12</junit.version>
</properties>
```

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/ssm_db
jdbc.username=root
jdbc.password=itheima
```

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

## 配置文件引用pom属性

- 作用
  - 在任意配置文件中加载pom文件中定义的属性
- 调用格式

```
${jdbc.url}
```

- 开启配置文件加载pom属性

```
<!--配置资源文件对应的信息-->
<resources>
    <resource>
        <!--设定配置文件对应的位置目录，支持使用属性动态设定路径-->
        <directory>${project.basedir}/src/main/resources</directory>
        <!--开启对配置文件的资源加载过滤-->
        <filtering>true</filtering>
    </resource>
</resources>
```

# 资源配置



## 小节

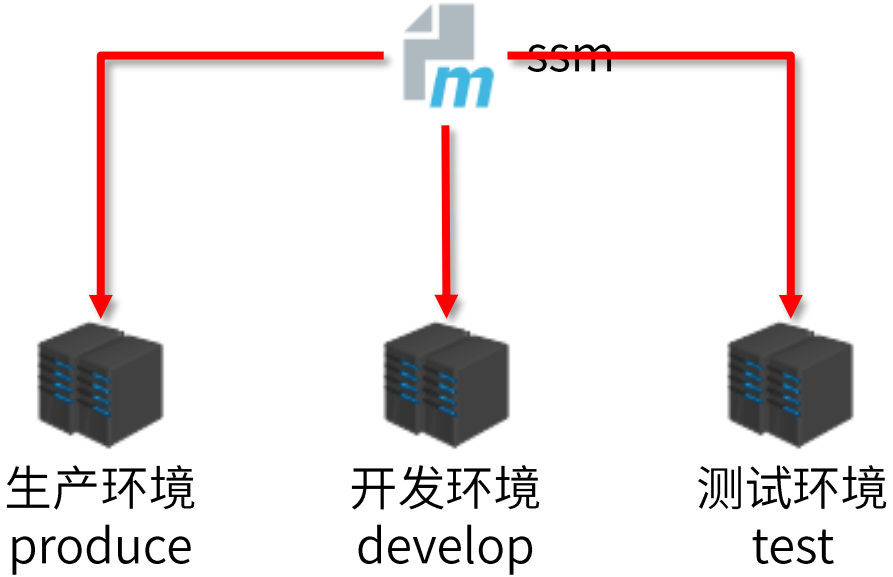
- 配置文件中读取pom属性值
  - 在pom文件中设定配置文件路径
  - 开启加载pom属性过滤功能
  - 使用${属性名}格式引用pom属性

# 目录 Contents

# 多环境开发配置



## 多环境兼容



# 多环境开发配置



## 多环境配置

```
<!--创建多环境-->
<profiles>
    <!--定义具体的环境：生产环境-->
    <profile>
        <!--定义环境对应的唯一名称-->
        <id>pro_env</id>
        <!--定义环境中专用的属性值-->
        <properties>
            <jdbc.url>jdbc:mysql://127.1.1.1:3306/ssm_db</jdbc.url>
        </properties>
        <!--设置默认启动-->
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
    </profile>
    <!--定义具体的环境：开发环境-->
    <profile>
        <id>dev_env</id>
        ……
    </profile>
</profiles>
```

# 多环境开发配置



## 加载指定环境

- 作用
  - 加载指定环境配置
- 调用格式

```
mvn 指令 -P 环境定义id
```

- 范例

```
mvn install -P pro_env
```

# 多环境开发配置



## 小节

- 多环境开发配置
  - 配置多环境
  - 执行构建命令并指定加载对应环境配置信息

# 目录 Contents

# 跳过测试（了解）



## 跳过测试环节的应用场景

- 整体模块功能未开发
- 模块中某个功能未开发完毕
- 单个功能更新调试导致其他功能失败
- 快速打包
- ……

# 跳过测试



## 使用命令跳过测试

- 命令

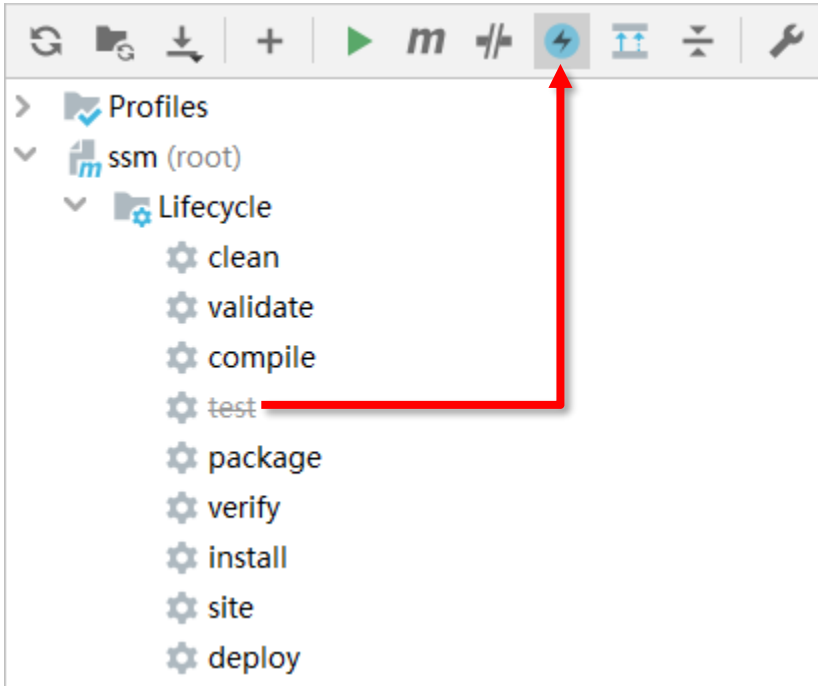
```
mvn 指令 -D skipTests
```

- 注意事项
  - 执行的指令生命周期必须包含测试环节

# 跳过测试



## 使用界面操作跳过测试



# 跳过测试



## 使用配置跳过测试

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.1</version>
    <configuration>
        <skipTests>true</skipTests><!--设置跳过测试-->
        <includes> <!--包含指定的测试用例-->
            <include>**/User*Test.java</include>
        </includes>
        <excludes><!--排除指定的测试用例-->
            <exclude>**/User*TestCase.java</exclude>
        </excludes>
    </configuration>
</plugin>
```

# 跳过测试



## 小节

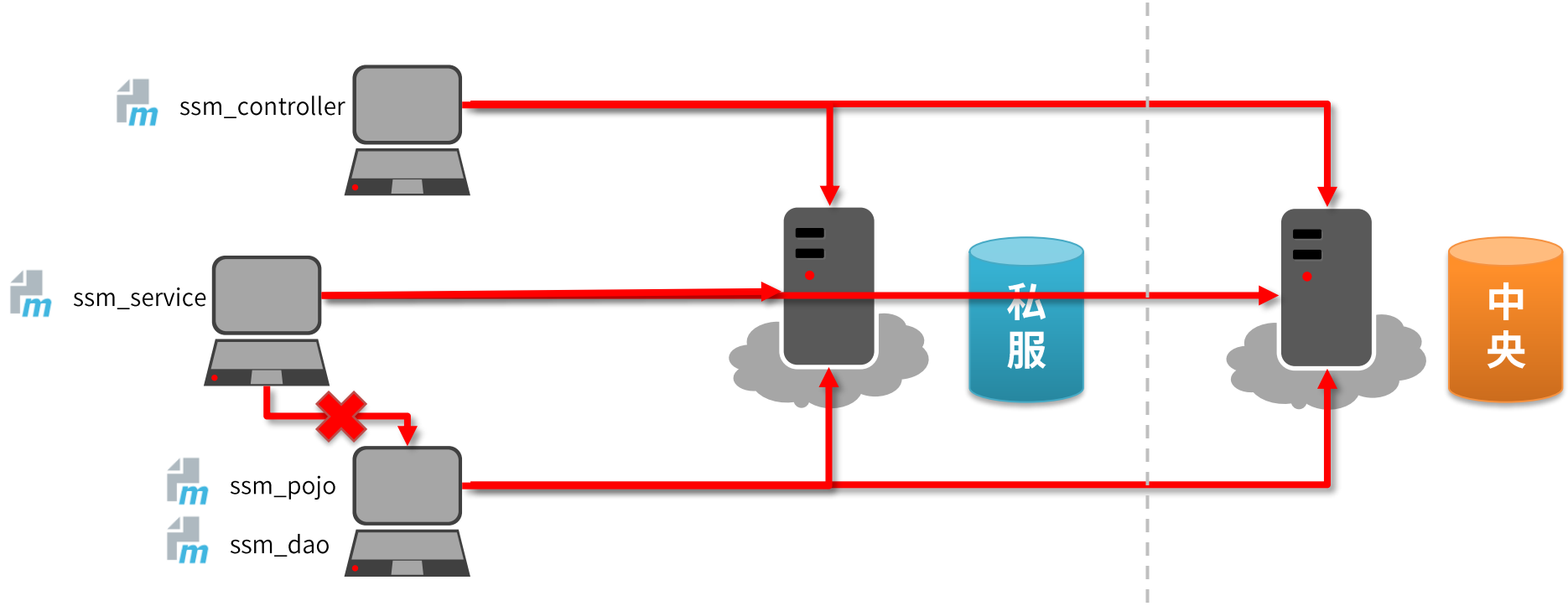
- 三种跳过测试的方式

# 目录 Contents

# 私服



## 分模块合作开发



# 私服



## Nexus

- Nexus是Sonatype公司的一款maven私服产品
- 下载地址：https://help.sonatype.com/repomanager3/download

# Nexus安装、启动与配置

- 启动服务器（命令行启动）

```
nexus.exe /run nexus
```

- 访问服务器（默认端口：8081）

```
http://localhost:8081
```

- 修改基础配置信息
  - 安装路径下etc目录中nexus-default.properties文件保存有nexus基础配置信息，例如默认访问端口
- 修改服务器运行配置信息
  - 安装路径下bin目录中nexus.vmoptions文件保存有nexus服务器启动对应的配置信息，例如默认占用内存空间

# 私服



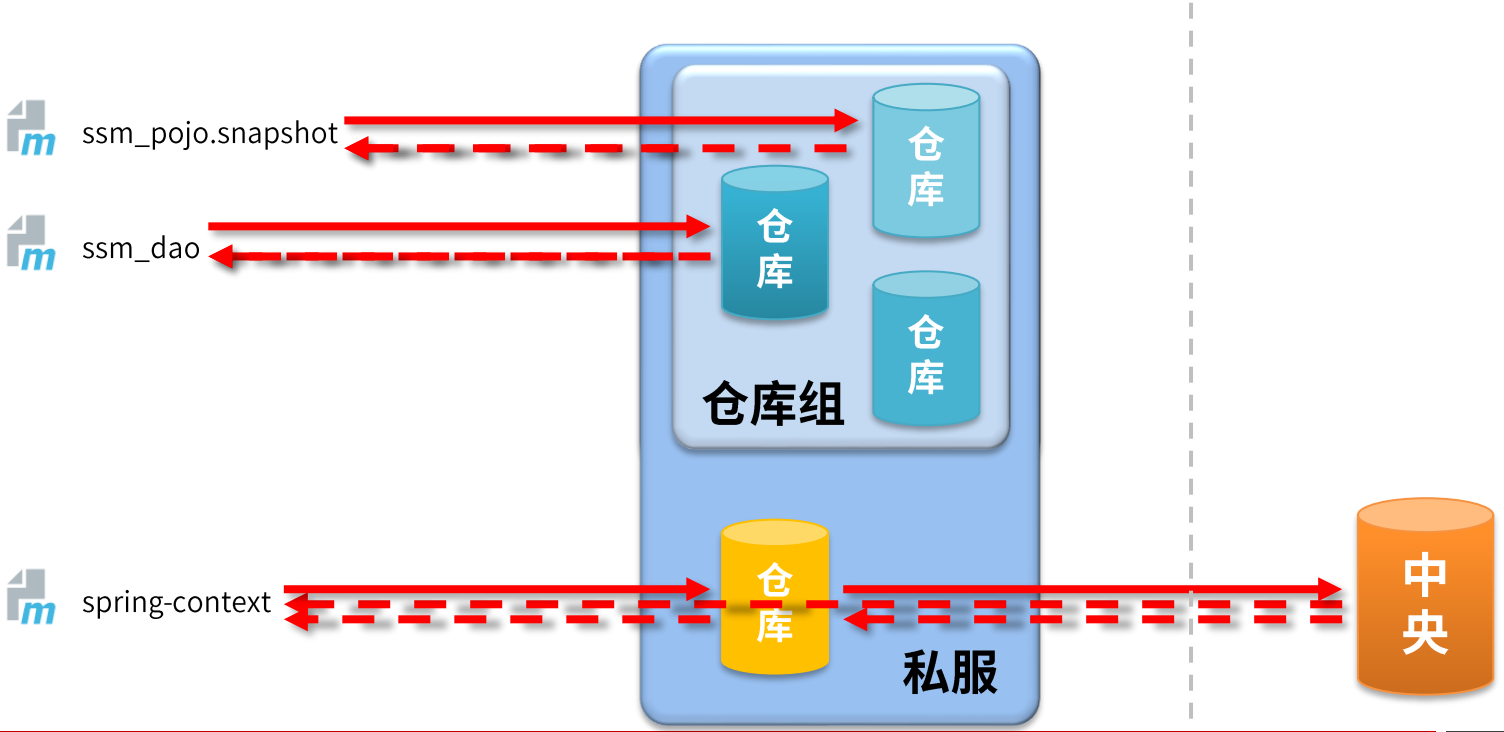
## 小节

- nexus私服安装、访问、配置

# 私服



## 私服资源获取



## 仓库分类

- 宿主仓库hosted
  - 保存无法从中央仓库获取的资源
    - 自主研发
    - 第三方非开源项目
- 代理仓库proxy
  - 代理远程仓库，通过nexus访问其他公共仓库，例如中央仓库
- 仓库组group
  - 将若干个仓库组成一个群组，简化配置
  - 仓库组不能保存资源，属于设计型仓库

## 资源上传

- 上传资源时提供对应的信息
  - 保存的位置（宿主仓库）
  - 资源文件
  - 对应坐标

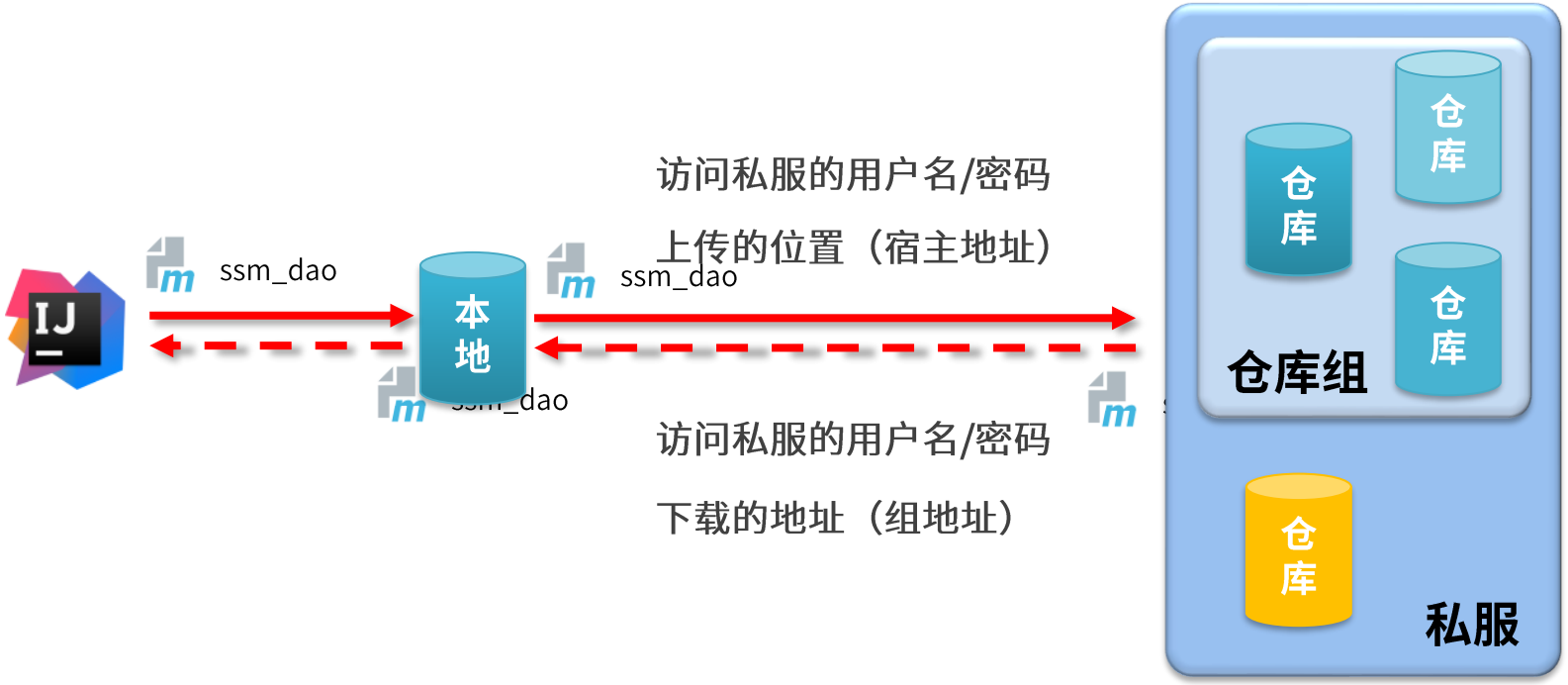
## 小节

- 仓库分类
  - 宿主仓库
  - 代理仓库
  - 仓库组

# 私服



## idea环境中资源上传与下载



## 访问私服配置（本地仓库访问私服）

- 配置本地仓库访问私服的权限（setting.xml）

```
<servers>
    <server>
        <id>heima-release</id>
        <username>admin</username>
        <password>admin</password>
    </server>
    <server>
        <id>heima-snapshots</id>
        <username>admin</username>
        <password>admin</password>
    </server>
</servers>
```

- 配置本地仓库资源来源（setting.xml）

```
<mirrors>
    <mirror>
        <id>nexus-heima</id>
        <mirrorOf>*</mirrorOf>
        <url>http://localhost:8081/repository/maven-public/</url>
    </mirror>
</mirrors>
```

## 访问私服配置（项目工程访问私服）

- 配置当前项目访问私服上传资源的保存位置（pom.xml）

```
<distributionManagement>
    <repository>
        <id>heima-release</id>
        <url>http://localhost:8081/repository/heima-release/</url>
    </repository>
    <snapshotRepository>
        <id>heima-snapshots</id>
        <url>http://localhost:8081/repository/heima-snapshots/</url>
    </snapshotRepository>
</distributionManagement>
```

- 发布资源到私服命令

```
mvn deploy
```

# 私服



## 小节

- 资源发布
  - 设置私服访问权限
  - 设置资源上传路径（私服宿主仓库地址）
  - 设置资源下载路径（私服仓库组地址）
- 发布命令


## Maven基础

- 分模块开发与设计
- 聚合
- 继承
- 属性
- 版本管理
- 资源配置
- 多环境开发配置
- 跳过测试
- 私服


传智播客旗下高端IT教育品牌