Maven实战指南

内容简介

本书由从事多年Java软件工程师培训的资深讲师Noble Yang，结合多年的教学经验，亲自执笔编写。

本书是全国首本，以程序员从零开始研究Maven、熟练使用Maven、到深刻理解Maven为线索的《Maven实战》。

全书以实践为宗旨：一切起源于实践，又回归于实践。

全书的总体分三个层次：

介绍Maven的安装使用。

1）结合实践，介绍Maven的作用和意义

2）详细介绍Maven的安装和同Eclipse的集成配置

3）详细介绍Maven，基于命令的使用，阐述Maven的大致运行过程

4）详细介绍基于Eclipse，使用Maven实现简单Maven工程的构建

5）根据实际需要，详细介绍基于Eclipse，使用Maven实现流行框架工程构建

6）详细介绍为了适合企业开发，搭建archiva私服

介绍Maven的核心概念和运行原理

1）详细介绍Maven的组成架构

2）详细介绍Maven的运行生命周期

3）详细介绍Maven的仓库、依赖和插件

介绍Maven的高级应用

1）详细介绍基于Maven生成项目站点等项目报告文档

2）详细介绍基于Maven测试

3）详细介绍灵活构建Maven项目

4）详细介绍自主开发Maven插件和Archetype

5）常用配置文件的标签附录

前 言

**为什么编写本书**

关于Maven的资料和书籍有很多：有比较简洁，一个帖子全部涉及到；有比较详细的，厚厚的一本书，方方面面都阐述的比较详细，面面俱到；还有些是针对某个方面，进行说明讲解的。等等

纵观所有的资料，以项目实践为需求导向，能剔除不必要过多理解的概念，教程序员快速在项目中上手使用Maven的书籍还是凤毛麟角的。

作为一个对Maven不了解，或者被零碎资料绕了好几天，绕得沮丧的程序员，要的不是这些系统的理论，高深的原理。要的是一看就明白，就能照着一步步上手，使用到项目中去的第一手资料。至于那些底层的原理，系统实现的架构和组件之间的关系，可以暂时没必要全部理解透彻。它们太抽象，太需要静下心来仔细琢磨了。可以在学会了怎么使用，在有运用的切实体会后，再进一步的升华。

这就是我们<<Maven实战指南>>的目的和追求：用简洁实用（尽量少专业术语）的语言，以实际项目为案例，按项目自身的发展为线索，介绍Maven在项目中每个环节的一步步使用，使读者由浅入深的了解Maven，试用Maven，运用Maven，最后实践结合理论，升华把握Maven，一切都是水到渠成。

**本书面向的读者**

因为本书是从零基础研究到深入熟练使用为线索组织的内容，所以最适合准备使用Maven进行开发的程序员。

本书的内容是先实践，再理论，最后又归于实践，所以非常适合Maven快速上手，逐步理解运用。如果您公司的项目马上要启动了，您对Maven的要求暂时不是太高，至少先使用起来，再慢慢熟练的话，本书非常合适您。

当然，对Maven有基本了解，要对Maven要深入了解并且灵活运用到项目中去的程序员和项目管理员，也可以借鉴本书后面部分内容。

**本书的主要内容**

第一章，主要从项目的需要出发，介绍Maven的作用和基于Maven进行项目构建的意义。

第二章、第三章，介绍Maven的安装配置，并且给予命令行构建Maven项目，体验Maven的基本操作过程。

第四章介绍在Eclipse上安装配置m2eclipse插件，并且构建简单maven项目，体验在Eclipse上构建maven项目的过程。

第五章到第八章详细介绍基于Eclipse的m2eclipse插件开发Web应用和基于流行框架，开发企业级Web应用。

第九章详细介绍Maven构建的生命周期和核心概念

第十章到第十三章，介绍对Maven在项目中比较常见的使用

**鸣谢**

目 录

[第一章 课前准备 8](#_Toc5320)

[1.1 项目经理的困惑 8](#_Toc3945)

[1.2 Maven的作用 8](#_Toc1205)

[1.3 研究Maven的步骤 9](#_Toc10933)

[第二章 安装配置Maven 9](#_Toc18104)

[2.1 Maven简介 9](#_Toc10646)

[2.2 安装Maven前的准备 10](#_Toc31099)

[2.3 安装配置Maven 14](#_Toc2237)

[第三章 使用Maven开发第一个案例 16](#_Toc25810)

[3.1 创建Maven项目 17](#_Toc24309)

[3.2 添加样例代码 20](#_Toc408)

[3.3 编写Maven骨架文件 21](#_Toc29180)

[3.4 编译和测试 22](#_Toc13724)

[3.5 生成站点和报告文档 24](#_Toc4934)

[3.6 小结 26](#_Toc3036)

[第四章 基于Eclipse安装Maven插件 28](#_Toc3019)

[4.1 搭建Eclipse+M2Eclipse的必要性 28](#_Toc13209)

[4.2 安装配置M2Eclipse插件 28](#_Toc22349)

[4.3 创建一个Maven项目 35](#_Toc9483)

[4.4 构建Maven项目 39](#_Toc12980)

[4.5 基于M2Eclipse完成所有工作 41](#_Toc192)

[4.6 小结 46](#_Toc24183)

[第五章 基于Maven开发Web应用 46](#_Toc9963)

[5.1 开发Web应用的思路 46](#_Toc15665)

[5.2 实现Web版HelloWorld 47](#_Toc11821)

[5.2.1 安装配置web应用的Archetype Catalog 47](#_Toc30657)

[5.2.2 基于ArcheType向导创建Web工程 49](#_Toc23964)

[5.2.3 编写样例代码 50](#_Toc23478)

[5.2.4 构建Web项目 52](#_Toc11287)

[5.2.5 测试 58](#_Toc3542)

[5.3 基于Maven开发用户模块的CRUD 58](#_Toc5928)

[5.3.1 创建Web工程和初始化数据库 58](#_Toc14834)

[5.3.2 添加相关依赖 60](#_Toc30210)

[5.3.3 添加注册代码 60](#_Toc29303)

[5.3.4 构建项目 83](#_Toc20827)

[5.3.5 测试 83](#_Toc17747)

[5.4 小结 84](#_Toc20277)

[第六章 开发企业级Web应用 84](#_Toc31008)

[6.1 企业web应用简介 84](#_Toc31189)

[6.2 搭建Maven私服 85](#_Toc15560)

[6.3 实现Struts2+Spring+Hibernate框架应用 92](#_Toc9071)

[6.3.1 创建公共POM 92](#_Toc9449)

[6.3.2 实现Hibernate DAO模块 99](#_Toc29789)

[6.3.3 实现 Service模块 117](#_Toc28106)

[6.3.4 实现Struts2 Web模块 125](#_Toc24534)

[6.3.5 整合成SSH 143](#_Toc28074)

[6.4 实现SpringMVC+Spring+MyBatis框架应用 149](#_Toc23416)

[6.4.1 创建公共POM 149](#_Toc17053)

[6.4.2 实现MyBatis DAO模块 153](#_Toc22576)

[6.4.3 实现Spring的Service层模块 163](#_Toc24674)

[6.4.4 实现SpringMVC Web模块 169](#_Toc12090)

[6.4.4 整合SSM 188](#_Toc1491)

[6.5 小结 189](#_Toc3157)

[第七章 生成项目站点 190](#_Toc7898)

[7.1 生成基本站点 190](#_Toc26205)

[7.1.1 简单站点 190](#_Toc32531)

[7.1.2 完善站点信息 192](#_Toc25060)

[7.2 添加插件丰富站点信息 194](#_Toc645)

[7.2.1 JavaDoc插件 194](#_Toc5520)

[7.2.2 源代码插件 195](#_Toc19736)

[7.2.3测试报告插件 197](#_Toc9782)

[7.2.4 源代码分析插件 198](#_Toc26283)

[7.3 个性化站点 199](#_Toc8465)

[7.3.1 修饰外观 199](#_Toc8444)

[7.3.2 自定义页面 205](#_Toc27792)

[7.3.3 国际化 210](#_Toc24006)

[7.4 部署站点 211](#_Toc11390)

[7.5 小结 215](#_Toc16055)

[第八章 版本管理 215](#_Toc7325)

[8.1 简介 215](#_Toc22061)

[8.2 准备点专业术语 215](#_Toc15593)

[8.3 自动版本发布 217](#_Toc26339)

[8.4 GPG签名验证 225](#_Toc17691)

[8.5 小结 232](#_Toc29167)

[第九章 Maven核心概念 232](#_Toc16682)

[9.1 简介 232](#_Toc3522)

[9.2 生命周期 232](#_Toc1748)

[9.2.1 生命周期简介 232](#_Toc5634)

[9.2.2 深入生命周期 233](#_Toc29817)

[9.2.3 调用生命周期阶段 235](#_Toc897)

[9.3 插件 239](#_Toc11216)

[9.3.1 插件的作用和目标 239](#_Toc29853)

[9.3.2 插件同生命周期阶段的绑定 239](#_Toc20284)

[9.3.3 插件参数配置 243](#_Toc29118)

[9.3.4 获取插件信息 244](#_Toc16073)

[9.3.5 调用插件 248](#_Toc5858)

[9.3.6 解析插件 249](#_Toc31261)

[9.4 坐标 252](#_Toc21065)

[9.5 仓库 253](#_Toc14047)

[9.5.1 Maven仓库的定义 253](#_Toc10688)

[9.5.2 仓库的管理方式 254](#_Toc30560)

[9.5.3 仓库的种类 254](#_Toc17191)

[9.5.4 配置远程仓库 258](#_Toc2957)

[9.5.5 快照版本 260](#_Toc4632)

[9.5.6 从仓库中解析依赖的机制 261](#_Toc30323)

[9.5.7 镜像 261](#_Toc23517)

[9.5.8 仓库搜索服务 262](#_Toc17628)

[9.6 依赖 263](#_Toc3102)

[9.6.1 依赖是什么 263](#_Toc1479)

[9.6.2 依赖的配置 264](#_Toc26603)

[9.6.3 依赖的范围 264](#_Toc23510)

[9.6.4 传递性依赖 266](#_Toc23043)

[9.6.5 依赖的调解 267](#_Toc5855)

[9.6.6 排除依赖 269](#_Toc10902)

[9.6.7 归类依赖 269](#_Toc32574)

[9.6.8 优化依赖 270](#_Toc17873)

[9.7 继承和聚合 271](#_Toc2215)

[第十章 Maven测试 279](#_Toc4074)

[10.1 测试简介 279](#_Toc4222)

[10.2 测试框架 279](#_Toc31382)

[10.2.1 JUnit单元测试框架 279](#_Toc11270)

[10.2.2 TestNG测试框架 282](#_Toc28416)

[10.3 Maven测试插件 284](#_Toc27873)

[10.3.1 Surefire插件简介 284](#_Toc30916)

[10.3.2 跳过测试 285](#_Toc30390)

[10.3.3 个性化指定运行测试 286](#_Toc16765)

[10.3.4 包含和排除测试 286](#_Toc22274)

[10.4 测试报告 287](#_Toc10712)

[10.4.1 基本测试报告 287](#_Toc22612)

[10.4.2 测试覆盖率报告 288](#_Toc29787)

[10.5 重用测试代码 289](#_Toc5253)

[第十一章 灵活构件Maven项目 290](#_Toc22330)

[11.1 Maven属性 290](#_Toc26035)

[11.2 需要灵活处理的构建环境 292](#_Toc13804)

[11.3 资源过滤 292](#_Toc5124)

[11.4 Maven的Profile 294](#_Toc2148)

[11.4.2 激活profile配置 294](#_Toc20086)

[11.4.3 profile的种类 297](#_Toc25883)

[11.5 Web资源过滤 298](#_Toc12108)

[11.6 在profile中激活集成测试 299](#_Toc3143)

[11.7 小结 300](#_Toc20702)

[第十二章 自定义Maven插件 300](#_Toc1526)

[12.1 自定义Maven插件简介 301](#_Toc24010)

[12.2 自定义Maven插件案例 301](#_Toc12778)

[12.3 自定义Maven插件的详细说明 309](#_Toc8160)

[第十三章 Archetype扩展 314](#_Toc24425)

[13.1 Archetype使用概述 315](#_Toc27456)

[13.2 自定义Archetype 316](#_Toc29781)

[13.3 Archetype数据库 316](#_Toc28539)

[附录 319](#_Toc9804)

# 第一章 课前准备

关于Maven的资料和书籍有很多：有比较简洁，一个帖子全部涉及到；有比较详细的，厚厚的一本书，方方面面都阐述的比较详细，面面俱到；还有些是针对某个方面，进行说明讲解的。等等

纵观所有的资料，以项目实践为需求导向，能剔除不必要过多理解的概念，教程序员快速在项目中上手使用Maven的书籍还是凤毛麟角的。

作为一个对Maven不了解，或者被零碎资料绕了好几天，绕得沮丧的程序员，要的不是这些系统的理论，高深的原理。要的是一看就明白，就能照着一步步上手，使用到项目中去的第一手资料。至于那些底层的原理，系统实现的架构和组件之间的关系，可以暂时没必要全部理解透彻。它们太抽象，太需要静下心来仔细琢磨了。可以在学会了怎么使用，在有运用的切实体会后，再进一步的升华。

这就是我们<<Maven实战指南>>的目的和追求：用简洁实用（尽量少专业术语）的语言，以实际项目为案例，按项目自身的发展为线索，介绍Maven在项目中每个环节的一步步使用，使读者由浅入深的了解Maven，试用Maven，运用Maven，最后实践结合理论，升华把握Maven，一切都是水到渠成。

## 1.1 项目经理的困惑

一个团队在接到公司下达的项目后，项目经理或项目架构师往往要在团队正式介入以前，做好充分的前期准备，比如：

1. 确定系统架构
2. 收集框架相关的jar包
3. 搭建SSM框架
4. 编写测试代码
5. 寻找框架依赖的jar包
6. 剔除冲突jar包
7. 制订需求设计文档规范
8. 测试文档和代码规范
9. 相关报告文档规范

内容真的很多、很杂，所以必须要进行充分细致的准备。这样一个团队在每承接一个项目的时候，都要重复做类似的事情。这种工作劳动强度大，而且缺少技术含量。

## 1.2 Maven的作用

这些问题，使用Maven都可以直接解决。

1. Maven帮我们统一集中管理好了所有的依赖包，不需要程序员再去寻找。

2.对应第三方组件用到的共同jar，Maven自动帮我们解决重复和冲突的问题。

3.Maven作为一个开放的架构，提供了公共的接口，方便同第三方插件的集成。程序员可以将自己需要的插件，动态地集成到Maven中，从而扩展新的管理功能。

4.Maven可以统一每个项目的构建过程，实现不同项目的兼容性管理。

## 1.3 研究Maven的步骤

为了保持我们的初衷，让读者快速顺利地掌握Maven，本书将按如下步骤，一步步地同大家一起打开Maven的大门。

第一步：安装配置Maven。

第二步：通过Java程序员的经典样例——HelloWord，初步体验Maven。

第三步：使用Eclipse+Maven，开发一个普通的Java Project。

第四步：使用Eclipse+Maven，开发JSP/Servlet Web应用。

第五步：使用Eclipse+Maven，开发基于SSH框架的Web应用。

第六步：使用Eclipse+Maven，开发基于SSM框架的Web应用。

第七步：使用Maven进行项目文档版本控制。

第八步：总结Maven应用的概念和流程。

# 第二章 安装配置Maven

## 2.1 Maven简介

Maven Apache开源组织奉献的一个开源项目。它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)。开发人员只需要做一些简单的配置，就可以批量完成项目的构建、报告和文档的生成工作。

当然，Maven除了是一个优秀的项目构建方面的管理工具之外，还有项目管理相关的其它特殊优势。

比如项目相关的第三方依赖包，这是Java程序员不可能回避的问题。

每个老程序员都会在自己的电脑里面专门有个目录，分类保存过去项目开发过程中使用到的第三方jar包，需要的时候从里面去筛选。新程序员就麻烦了，测试项目的时候，动不动就会遇到“ClassNoFoundException”，经常一整天就在搜索和重启中度过。

Maven就解决了这一问题。它统一管理起来了所有的依赖jar，甚至是不同版本。程序员也可以动态地将自己写好的模块打包成jar包，让它管理。需要的时候，可以直接通过简单的描述文件告诉Maven，它会自动帮程序员找出来，集成到项目中。

本书上的所有案例，都是基于3.3.9版本测试的。

## 2.2 安装Maven前的准备

因为Maven本身就是基于Java写的，所以在安装配置Maven之前，有必要将Java的运行环境安装配置好。

接下来介绍一下Java的安装和配置过程，有基础的读者可以跳过本节，但是要知道一点，Maven3.3.9版本要求的JDK，一定需要JDK1.7或以后的版本。

JDK的安装和配置，主要分如下步骤：

1. 下载合适的JDK安装软件安装

Oracle所有软件的下载连接如下：

<https://www.oracle.com/downloads/index.html>

咱们可以在里面找到需要的JDK安装包，下载Window版本的啊，咱们是在Window操作系统中完成的所有案例。

注意下载跟自己电脑芯片匹配的安装包（32位或64位）。现在的电脑，基本上是64位，建议java相关的软件，统一安装64位的。

下载完后就是安装啦，双击安装文件就行，后面就是下一步一直到结束，当然，中间有要接受协议的步骤，那肯定是直接接受咯，要不安装不了的。

开始的时候，有建议选择安装目录，如果有基础，对自己电脑文件有自己的管理，可以选择自己的目录，为了省去麻烦，建议直接使用默认目录。

安装好后的目录结构，大概如图2-1：

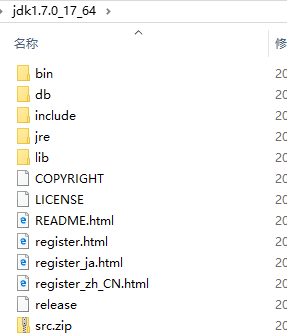


图2-1 JDK安装目录

JDK1.7.0\_17\_64是我安装JDK后的总目录，bin、db、include等，是它里面的子目录和文件。

1. 配置JDK环境变量

JDK的环境变量，目前来说只要配置JAVA\_HOME和PATH就行。

在我的电脑上右键鼠标，选择属性，出现如下图2-2窗口：



图2-2 Window系统信息

点击左边的 高级系统设置，出现如下图2-3窗口：

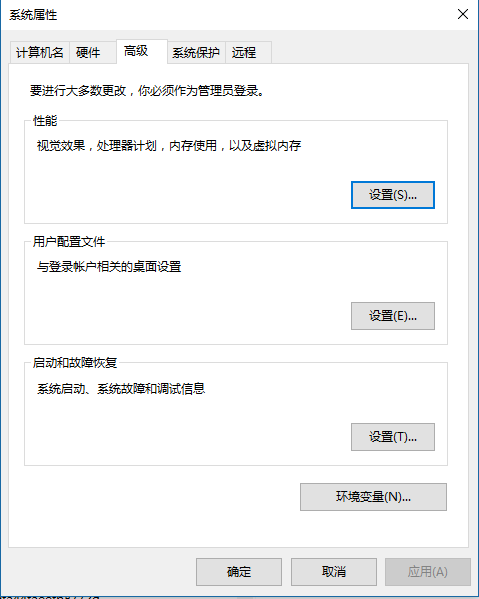


图2-3 系统属性

点击 高级 标签中的 环境变量按钮。显示如下图2-4窗口：

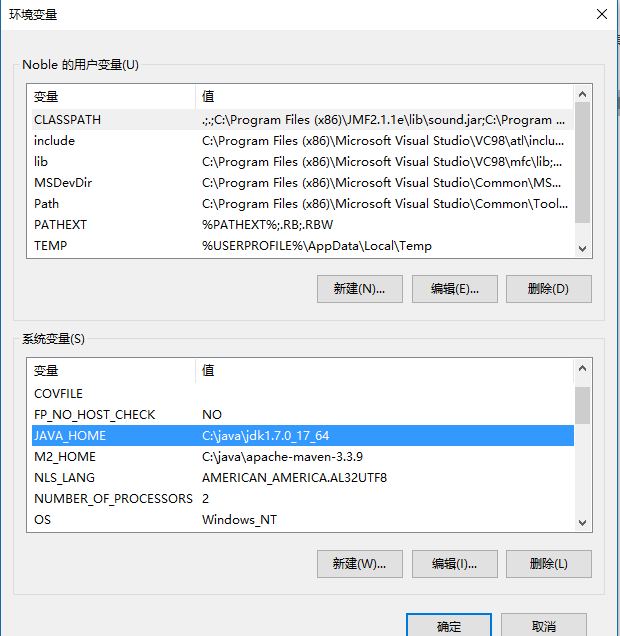


图2-4 系统环境变量

1）配置JAVA\_HOME目录

查看 系统变量 中有没有JAVA\_HOME(上图是我的电脑中有配置过，所以有JAVA\_HOME这项的显示)。

如果没有，点击新建，在弹出的窗口中输入JAVA\_HOME 和 Java的安装目录。

Java的安装目录是从安装盘符到java安装的总目录。如上图，我的java总目录是jdk1.7.0\_17\_64,它是安装在c:\java目录下的，所以JAVA\_HOME的值是 c:\java\jdk1.7.0\_17\_64

1. 配置PATH目录

同样在配置JAVA\_HOME的地方，查找PATH变量，一般是存在的，如图2-5：

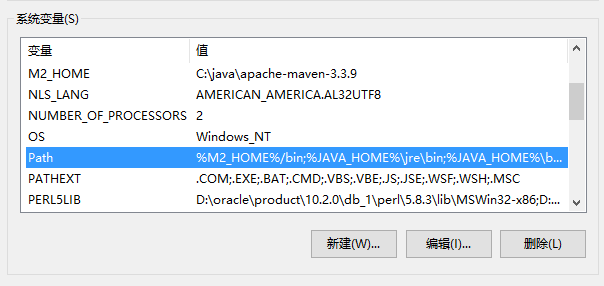


图2-5 系统变量

点击 编辑 按钮，在以前的path值的前面，新添加咱们jdk的bin目录，我的jdk的bin目录是 c:\java\jdk1.7.0\_17\_64\bin,就是前面的JAVA\_HOME目录后面添加 \bin。添加完后，同后面的内容要用分号分开，而且是英文输入法的分号。结果如

c:\java\jdk1.7.0\_17\_64\bin;以前的内容

1. 测试JDK是否安装成功

打开一个cmd窗口，分别输入javac -version 命令和 java -version命令，能出现如下图2-6的Java版本信息显示，就表示JDK安装成功了。



图2-6 Dos命令窗口

## 2.3 安装配置Maven

JDK环境安装好了，接下来就要正式安装配置Maven了，先从如下链接下载Maven。

<http://maven.apache.org/download.cgi>

它是以压缩包提供的，下载binary形式的压缩包就行。Source形式的压缩包是源代码，以后要进一步研究Maven，甚至完善Maven的功能，咱们就可以从source包入手了。现在只要binary的包。

下载完后，就直接用解压工具(比如rar解压软件)解压到自己的一个空目录下，我这里是解压到 c:\java\apache-maven-3.3.9，最后的目录结构图2-7如下:

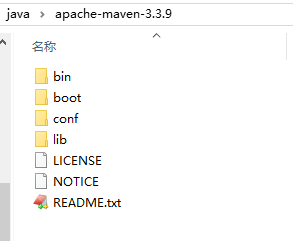


图2-7 Maven安装目录

安装好后，接下来就是对Maven进行配置。

配置内容同JDK的配置内容一样，也是两部分：一是在系统环境变量中配置一个M2\_HOME，另外一个也是将Maven里面的bin目录，添加到PATH环境变量中去。

1. 配置M2\_HOME

同配置JAVA\_HOME一样，在系统环境变量中添加M2\_HOME，参考JDK的JAVA\_HOME配置，在这里就不累赘了，效果图2-8如下：

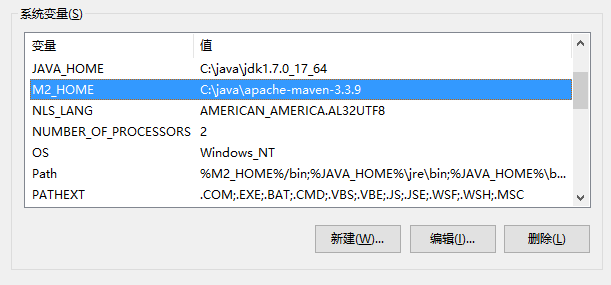


图2-8 设置M2-HOME环境变量

1. 追加PATH

类似前面JDK PATH环境变量的配置，请参考JDK配置，效果图2-9如下：

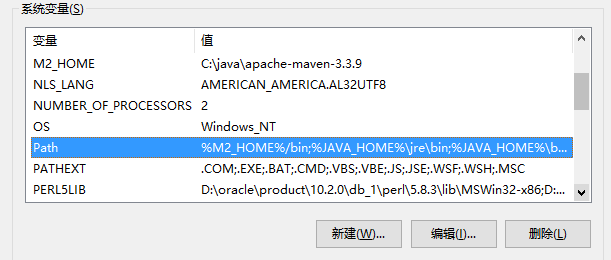


图2-9 Maven Path环境变量

其中%M2\_HOME%\bin; 是作为Maven的path添加上去的。

1. 测试是否安装成功

打开一个行的CMD窗口（注意，一定是要重新打开一个，不能使用配置环境变量之前的旧cmd窗口），输入 mvn -v 命令运行，出现如下图2-10显示，表示安装成功。



图2-10 Maven安装检测

# 第三章 使用Maven开发第一个案例

前面的准备工作做好了，接下来，咱们就开始体验体验一下Maven啦。

既然是体验，就不要搞太复杂了，咱们就HelloWorld一把：编写一个简单的代码，用Maven编译测试一下，同Maven来个感性接触。

当然，因为是感性接触的体验，我们整个编码过程就先不使用Eclipse等方便的开发工具了，用简单编辑器编写代码和配置文件，编译和测试过程统一用maven命令进行了。

**严重声明：后面的操作，没有特殊说明，全部要连外网操作**

**不要问我why，后面你会懂的 ^\_^**

## 3.1 创建Maven项目

Maven项目同以前咱们用Eclipse或其他工具产生的项目一样，有自己的目录结构和特殊的意义。

比如一般有如下目录：

src\main\java 用来存放项目的java源代码

Src\main\resources 用来存放项目相关的资源文件(比如配置文件)

Src\test\java 用来存放项目的测试java源代码

Src\test\resource 用来存放运行测试代码时，所依赖的资源文件

当然，还有一个pom.xml文件，该文件来配置maven管理的所有内容

我们可以按maven的要求，自动创建目录结构，按maven的要求添加项目相关的配置文件。这样确实可以实现，但是太累了，特别是我们刚开始玩maven，就有那么多复杂罗嗦的东西要理解并且用记事本抄写出来，太烦了！

烦是吧，那行，有那么多好心人，用代码将这些要做的事情封装实现了，就同在Eclipse中创建工程的那种图形化导向页面一样（这种效果我们到使用Eclipse+Maven的时候体现），咱们只要按它的步骤输入信息和命令就行，完后自动产生项目架构。

那行，既然有这么个捷径，那咱们开始呗！

不急，前面不是说要输入写信息和命令不是，咱们还是简单了解下有哪些信息和命令吧，特别是信息，这样输入的时候也有个方向不是！

1. 命令

好简单，就是创建项目的命令：create。

要进一步的话，就是要调用哪个软件的create命令，帮咱们创建项目。而且咱们会把这样的软件有个统一的称呼，叫插件（plugin），创建项目的插件咱们叫archetype插件(archetype-plugin)。

1. 信息

我们是要用命令创建项目嘛，那就是项目相关的信息啦。哪些信息呢？

1. groupId 组ID
2. artifactId 构件Id
3. packageName 包名
4. Version 版本

其实packageName和version好理解，我们写的类，肯定要放在一个标准包下或标准包的子包下吧，这个packageName就是指那标准包。Version就是当前代码的版本号咯。

相对来说，组Id和构件Id可能是新接触的。不过无所谓，我们可以直接将它们理解成是一个定位或管理项目代码的层次结构。比如我们程序员，老板看我们加班好辛苦的，要给我们发点辛苦奖金啥的，他是不是在红包上面要写上 项目研发部Maven项组的某某某呀，这里的groupId和artifactId同部门名称和组名称一样，用来唯一确定一个项目（软件、功能）的。每个项目，这两个合起来要唯一，同咱们数据库中的组合主键的唯一确定一条记录一样。所以有些地方，会把这两个描述的信息合起来，叫“坐标”。形象把，坐标不就是唯一定位嘛，至少地理上的坐标用来定位位置，这里定位项目(软件、功能)而已 ^\_^

基本信息介绍完了，咱们就用命令来产生项目吧。

有两种方式可以创建项目（**特别注意，要联外网啊**）

1. 使用命令向导一步步创建项目（简单介绍，不推荐）

1）在硬盘上创建一个空的目录，用来存放maven项目。如：e:\temp\demoMaven

2）打开cmd窗口，用cd命令，切换到demoMaven目录下。如图3-1显示：

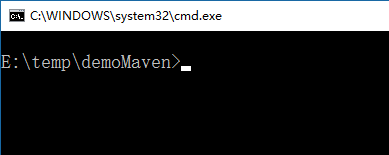


图3-1 Maven项目目录

3）在cmd窗口中输入 mvn archetype:generate 回车

联网初始化一段时间后（我的网络不少于5分钟），会一步步提示输入groupId，artifactId、version、packageName等信息，最后创建成功，而且咱们可以在e:\temp\demoMaven空目录下，发现一个同artifactId一样的目录，这就是创建的项目目录。

1. 在命令中输入所有必要信息直接创建项目

1）在硬盘上创建一个空的目录，用来存放maven项目。如：e:\temp\demoMaven

2）打开cmd窗口，用cd命令，切换到demoMaven目录下。如图3-2显示：

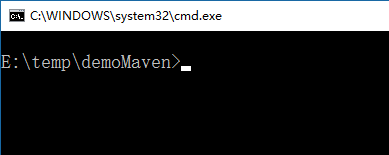


图3-2 Maven项目目录

3）在cmd窗口中输入 如下命令 并回车

mvn org.apache.maven.plugins:maven-archetype-plugin:2.2:create   
 -DgroupId=cn.com.mvnbook.demo   
 -DartifactId=MVNBookTP01  
 -DpackageName=cn.com.mvnbook.demo.tp01

说明：

1. org.apache.maven.plugins:maven-archetype-plugin:2.2 ：  
    使用groupId为org.apache.maven.plugins，artifactId为maven-archetype-plugin，版本为   
    2.2的archetype插件，帮咱们创建工程  
   2、-DgroupId=cn.com.mvnbook.demo  
    指定咱们要创建的工程的groupId
2. -DartifactId=MVNBookTP01 指定咱们工程的artifactId
3. -DpackageName=cn.com.mvnbook.demo.tp01 指定咱们工程代码的标准包

Maven执行命令的时候，会在本地找是否有指定版本的archetype插件，如果没有，就需要联网去下载。所以第一次执行该命令的时候，会在窗口提示信息上面发现很多download。不管怎么样，最后显示的正常现象如下图3-3：

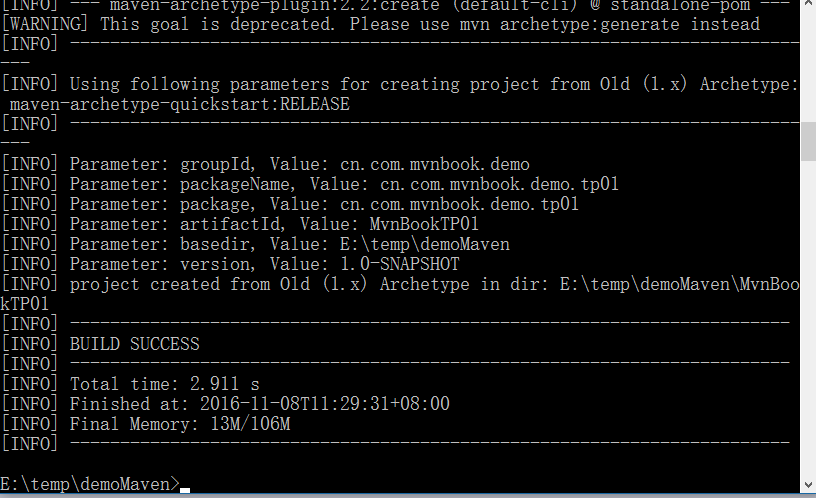


图3-3 Maven创建项目提示

同时，会在咱们的demoMaven目录下创建一个新的MvnBookTP01目录，结构如下图3-4：

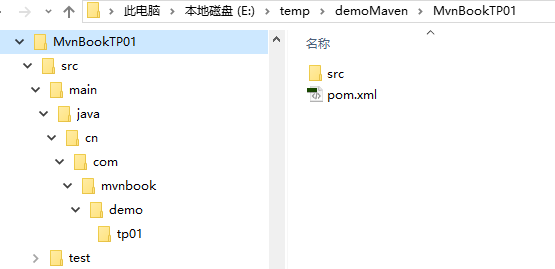


图3-4 简单Maven项目目录结构

到这里，咱们就使用archetype插件，帮我们创建好了第一个工程的架构。详细内容，请查看随书代码里面的[MvnBookTP01](代码/MVNBookTP01)

## 3.2 添加样例代码

为了完成体验，我们需要写两个代码，一个是HelloWorld.java，放在src/main/java目录下；一个是TestHelloWorld.java，用来体现测试的，放在src/test/java目录下。下面就一个个的介绍它们的内容：

1. HelloWorld.java  
   （随书代码[MvnBookTP01\ src\main\java\cn\com\mvnbook\demo\tp01\HelloWorld.java](代码/MVNBookTP01/src/main/java/cn/com/mvnbook/demo/tp01/HelloWorld.java)）

package cn.com.mvnbook.demo.tp01;

/\*\*

\* 这是为了研究Maven，写的第一个java代码

\* 功能很简单，输出一个HelloWorld的问候

\*

\* @author Noble

\* @version 1.0

\* \*/

public class HelloWorld {

/\*\*

\* 输出问候

\* @param name String，说话人名称

\* @return String 格式是：xxx say HelloWorld

\* \*/

public String say(String name){

return name + " say HelloWorld";

}

}

1. TestHelloWorld.java  
   （随书代码  
    [MvnBookTP01\ src\test\java\cn\com\mvnbook\demo\tp01\TestHelloWorld.java](代码/MVNBookTP01/src/test/java/cn/com/mvnbook/demo/tp01/TestHelloWorld.java)）

package cn.com.mvnbook.demo.tp01;

import junit.framework.Assert;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

public class TestHelloWorld {

private HelloWorld hello;

@Before

public void init(){

hello = new HelloWorld();

}

@Test

public void testSay(){

String name = "张三";

String exp = "张三"+" say HelloWorld";

String act = hello.say(name);

Assert.assertEquals(exp, act);

}

@After

public void destory(){

hello = null;

}

}

## 3.3 编写Maven骨架文件

代码写好了，接下来就是要通过配置文件，告诉maven来管理了。怎么告诉maven来管理呢，那就是接下来我们要了解的pom.xml，也是有些地方称呼的骨架文件。该文件创建工程的时候，自动生成在工程目录下：

Pom.xml  
（随书代码[MvnBookTP01\ pom.xml](代码/MVNBookTP01/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.com.mvnbook.demo</groupId> ⑴

<artifactId>MvnBookTP01</artifactId> ⑵

<version>1.0-SNAPSHOT</version> ⑶

<packaging>jar</packaging> ⑷

<name>MvnBookTP01</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId> ⑸

<artifactId>junit</artifactId> ⑹

<version>4.7</version> ⑺

<scope>test</scope> ⑻

</dependency>

</dependencies>

</project>

说明：

⑴创建工程时指定的groupId

⑵创建工程时指定的artifactId

⑶当前工程的版本

⑷工程编译好后，打成jar包安装发布

⑸测试时需要依赖的junit的groupId

⑹测试时需要依赖的junit的artifactId

⑺测试时需要依赖的junit的版本

⑻指定测试依赖的作用范围是测试

## 3.4 编译和测试

1、打开cmd窗口，将目录切换到工程目录下（MvnBookTP01），如图3-5：

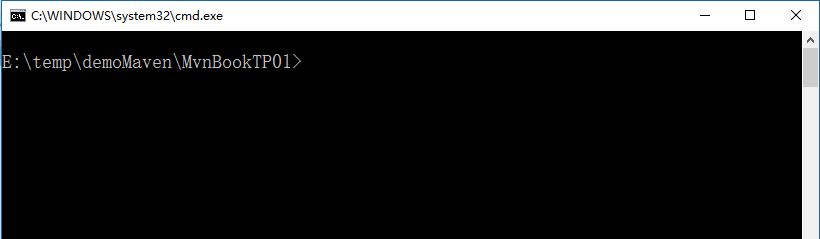


图3-5 Maven工程目录

2、输入 mvn clean 回车，清空以前编译安装过的历史结果，如图3-6：

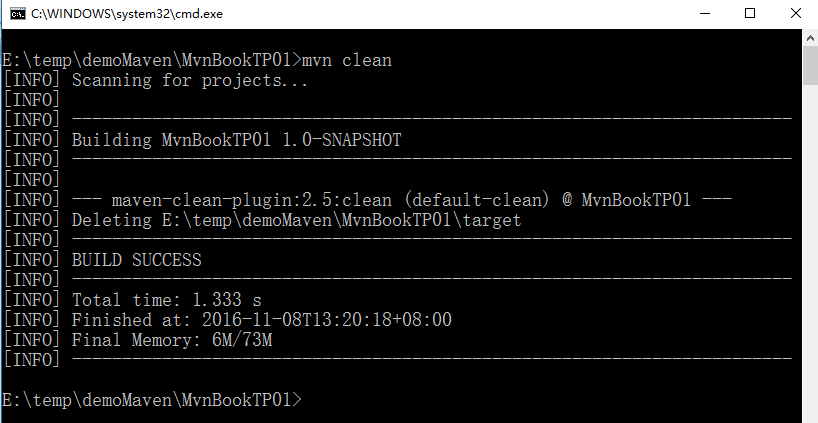


图3-6 mvn clean执行提示

1. 输入 mvn compile 回车，编译源代码，如图3-7：

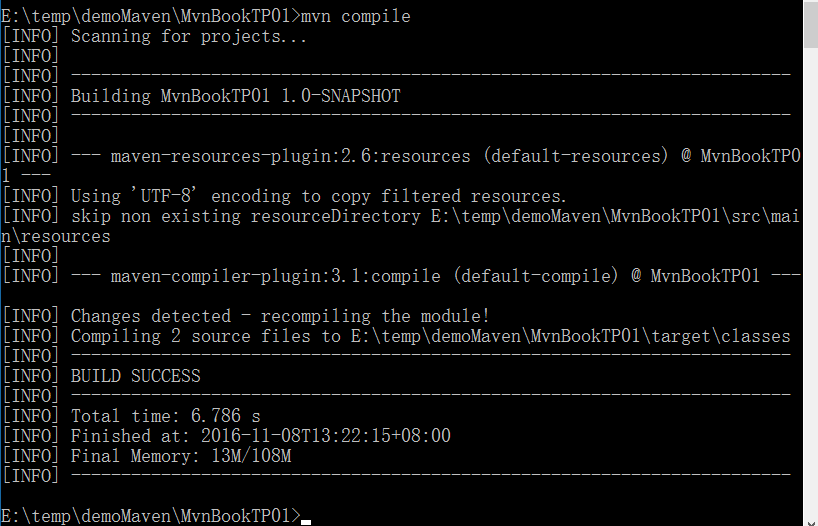


图3-7 mvn compile执行提示

1. 输入 mvn test 回车，运行测试案例进行测试，如图3-8：

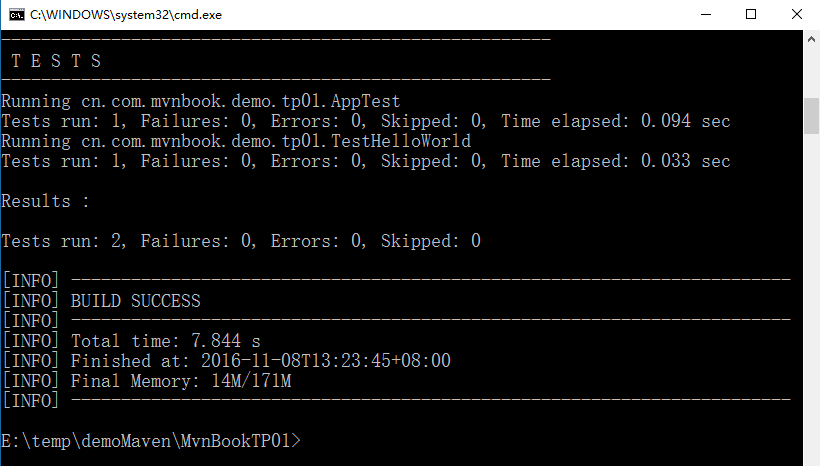


图3-8 mvn test执行提示

1. 输入 mvn install 回车，将当前代码打成jar包，安装到Maven的本地管理目录下，其它maven工程只要指定坐标，就可以使用。如图3-9：

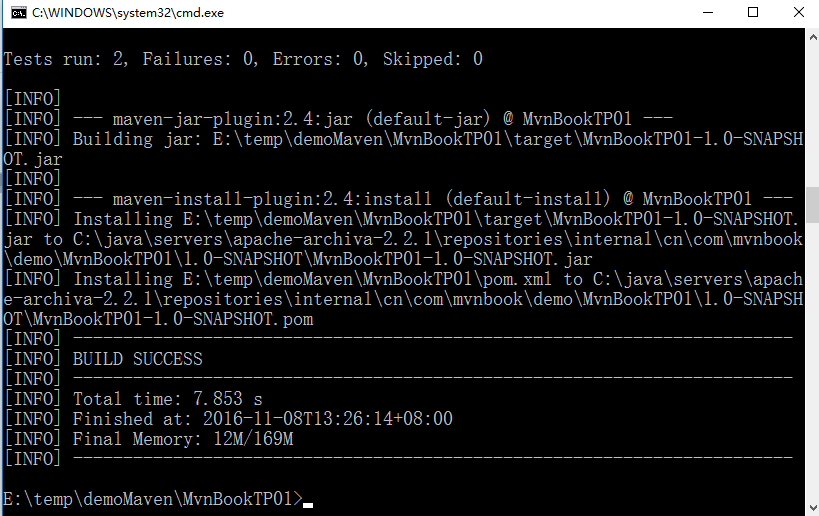


图3-9 mvn install执行提示

其中 [INFO] installing ... 指定了我当前的maven本地构件保存的目录是  
C:\java\servers\apache-archiva-2.2.1\repositories\internal

到现在为止，咱们就把以前编码所最常关心的操作整完了，包括工程的创建、源代码的编写、单元测试代码的编写、代码的编译、测试案例的运行以及最后的打包。

有找到点感觉没 ^\_^

## 3.5 生成站点和报告文档

前面已经完成了一个项目的基本管理过程，接下来咱们再体验体验相关的文档的产生。不玩多了，体验吗，就折腾一下站点信息和javadoc文档意思意思 ^\_^

1、生成站点信息

还在上节用到的cmd窗口下，输入mvn site命令，就会自动生成站点信息。如图3-10：

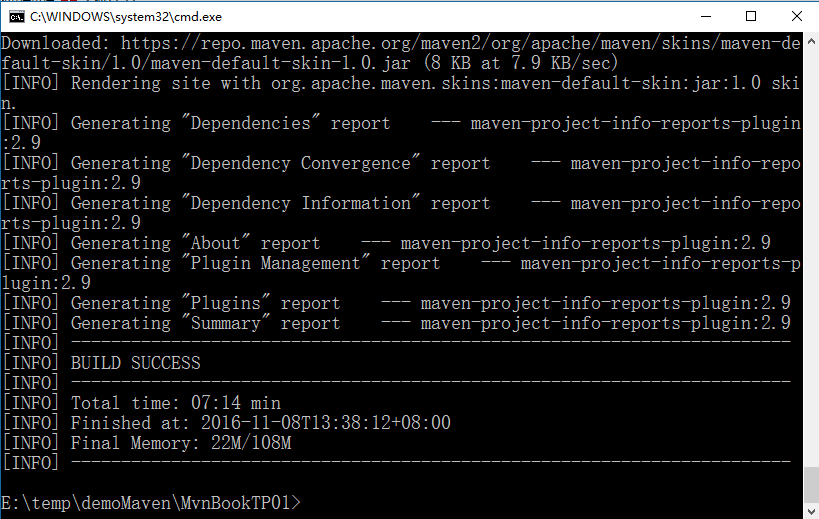


图3-10 mvn site执行提示

执行完后，查看一下工程目录下的target目录，里面自动添加了一个site目录，里面都是站点信息页面，打开其中的index.html（随书代码[MvnBookTP01\target\site\index.html](代码/MVNBookTP01/target/site/index.html)），咱们就可以看到如下类似的页面，里面描述的就是咱们项目相关的信息。如图3-11：

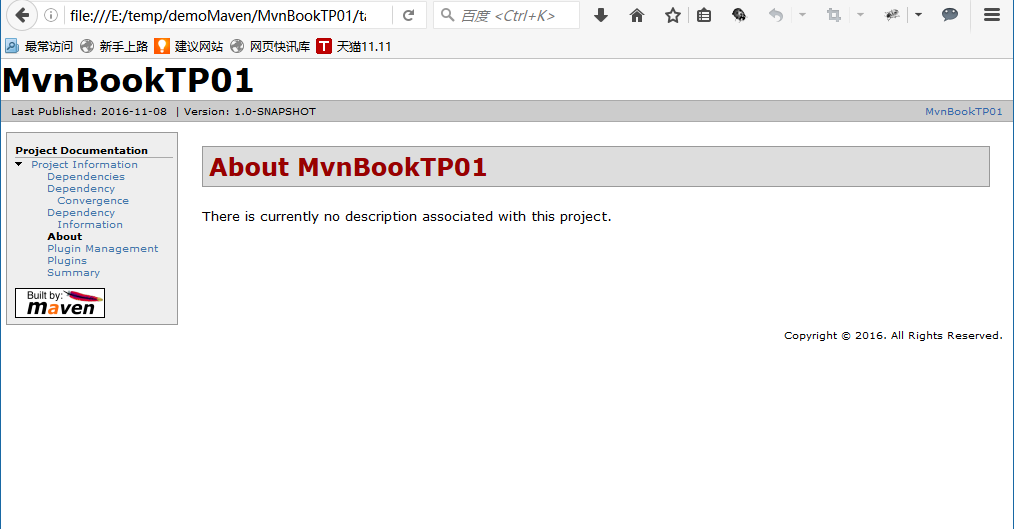


图3-11 maven生成的站点页面

1. 生成API Doc文档

打开cmd窗口，切换到工程目录，输入 mvn javadoc:javadoc回车，会自动身材APIDoc文档。如图3-12：

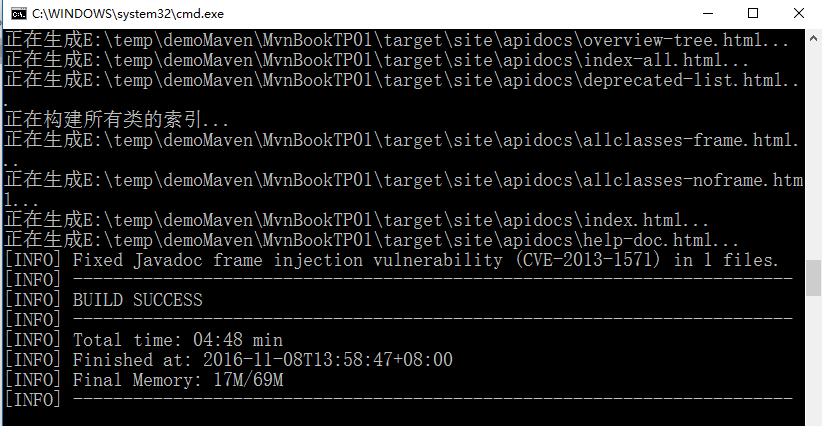


图3-12 mvn javadoc：javadoc执行提示

在工程中自动产生target\site\apidocs目录，里面就是当前工程中代码的API Doc文档。打开index.html（随书代码[MvnBookTP01\target\site\index.html](代码/MVNBookTP01/target/site/apidocs/index.html)）页面如下图3-13：



图3-13 Maven生成的API doc页面

咱们回顾一下JDK的API DOC文档，是不是也就长得这样呢。不错，这就是API DOC帮助文档，只是不是JDK的，是咱们自己写在Java源代码中的javadoc注释生成的。使用咱们组件的人，就不需要看源代码，直接查看咱们刚刚生成的API Doc文档就行了。

## 3.6 小结

到现在，我们已经有切切实实的用Maven折腾出一个简单的java代码了：从创建项目，到编写代码、编译、测试和打成jar包，最后还能像模像样的意思了下文档信息，可以方便第三方程序员学习和使用我们的成功 ^\_^

以前咱们都是一步步的摸石头过河，奔的一个快速整出结果出来的目标进行的，没有太多的讲什么道理和理论。接下来咱们有必要梳理梳理了。

当然，咱们也还是遵循咱们的规矩：整理熟悉的，能搞懂的，整理出来后为后面的研究打下基础。那些搞不懂、难得明白的东西、太抽象的东西，一切丢到后面，慢慢的一口口吃，逐步逐步的再整理再沉淀。

接下来，咱们小结一下如下概念，大家看看是不是那么回事：

1. Maven是什么东东，有什么作用

Maven像不像一个管理者呢，而且是一个很开明开放的管理者？

它啥具体的功能都不做。具体功能都是由相关的插件。当要做某个事情的时候，直接安排使唤一个插件去折腾得。

比如咱们用mvn创建一个工程用的命令：  
 mvn org.apache.maven.plugins:maven-archetype-plugin:2.2:create

不就是在使唤一个groupId叫org.apache.maven.plugins，artifactId叫maven-archetype-plugin，版本为2.2的家伙(插件)去折腾得吗。

同样编译 mvn compile, 测试 mvn test，创建站点信息mvn site,生成doc文档信息mvn javadoc:javadoc，虽然他们没有想创建项目那么明确指定哪些家伙，但是内部都是由mvn从晚上收罗最新的相关组件完成的。从这点看，是不是还体现了maven显示蛮开明的吧。

对头，maven就是这个体系或框架，它能协调最牛的方方面面的插件，完成我们对项目的构建和管理的需要，包括咱们想要的项目的其它管理功能。咱们唯一要做的，是把中意的插件告诉maven，剩下的咱都不用管了。

当然还要注意，pom.xml是咱们同maven进行信息沟通的唯一途径，一个项目能构建得怎么样，成功与否，就靠pom.xml对相关插件信息描述准确充分与否了，所以咱们把这文件又叫maven工程的骨架文件。

1. 插件、依赖和构件的概念

插件，咱们应该不陌生了，以前Eclipse里面不是也可装插件，来方便特点的开发功能嘛。对，maven中的插件，就是能被maven协调，去做特点功能的，由第三方开发的工具软件。

那依赖是什么呢？

唉，依赖，就是依赖嘛^\_^

你看，我们要跑起单元测试，不是就依赖jUnit的jar包嘛，jUnit就是咱们单元测试的一个依赖。说白了，插件是maven的依赖，我们用到的第三方jar包，也是我们项目的依赖。晓得了不！

至于构件嘛，就是可以形成别人依赖的所有的东东。当然咯，要被maven找到的东东呢，maven找不到，那就形同摆设了，它不能被maven协调过来嘛，有什么用？

我们自己写的代码，编译测试打成jar包后，安装到maven本地能找到的地方（仓库），那也是一个个不起眼的构件，了不起吧^\_^

1. 坐标

坐标是啥呢？

前面咱们好像有简单对比地理坐标的概念说过，说白了，坐标，就是maven用来唯一确定某个构件的信息（groupId，artifactId，version和packaging）。你看咱们自己创建的工程不也有这些信息嘛，只要编译打包安装到maven所管理构件的地方，下次只要告诉maven这些信息，maven就能把它们请出来，给别的项目使用，让它们也依赖依赖咱^\_^

1. Maven的仓库

通过对前面概念的介绍，Maven仓库应该晓得是啥东东，起什么作用的了吧。

它就是一个用来存放maven可以找得到的构件的地方。

有两种仓库，一个是本地仓库，还一个就是远程仓库。

本地仓库就是在本地存放构件的地方，默认是咱们电脑的当前用户目录的.m2\repository下，我这是C:\Users\Noble\.m2\repository目录。

远程的就有好多，蛮常用的就是http://mvnrepository.com/。

我们前面为什么强调一定要联网呀，就是刚装的maven环境，是啥子插件都没有的，做啥事都需要从远程仓库中下载下来，才能完成。当然下次一次后，maven会自动保存到本地仓库中，下次再需要的时候，就没必要联网下载了嘛，要咱们自己来，不也会这么管理嘛，毕竟一次性的筷子太耗资源了不是 ^\_^

# 第四章 基于Eclipse安装Maven插件

## 4.1 搭建Eclipse+M2Eclipse的必要性

前面我们用手动方式折腾出了一个HelloWorld，出是出来了，可是太麻烦。

咱不记得命令了怎么办？还有那些插件坐标，哪个记得住呢？要命的是编写代码，用的文档编辑器，一点都不专业，也没点点提示。以前用的Eclipse编代码多好呀，哪里处理问题，里面就提示错误；对不记得的API，也能随时提供帮助。

能不能将对Maven的操作同Eclipse结合起来，在Eclipse上用图形化界面和菜单式命令，协作咱完成对Maven的操作呢？

有需求就有市场，有市场，就有人挖掘并且解决。而且Eclipse本身就是一个开源精神好强烈的开源工具，能很好的接纳集成第三方的插件。

有了这些前提，m2eclipse出来了！

## 4.2 安装配置M2Eclipse插件

1. 下载Eclipse，建议下载开发JavaEE版本的Eclipse IDE，因为我们后面最终是要开发Web应用，下载连接为：

http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/mars2

打开页面如图4-1：



图4-1 Eclipse下载页面

结合整个环境，我这下载的是 Windows 64-bit（**注意是Mars Packages的**），eclipse-jee-mars-2-win32-x86\_64.zip

1. 使用解压工具解压eclipse-jee-mars-2-win32-x86\_64.zip文件，点击解压目录下eclipse目录下的eclipse.exe文件，启动Eclipse如图4-2：



图4-2 Eclipse启动界面

1. 点击菜单栏中的 window->Preferences选项，打开Preferences窗口，如图4-3：

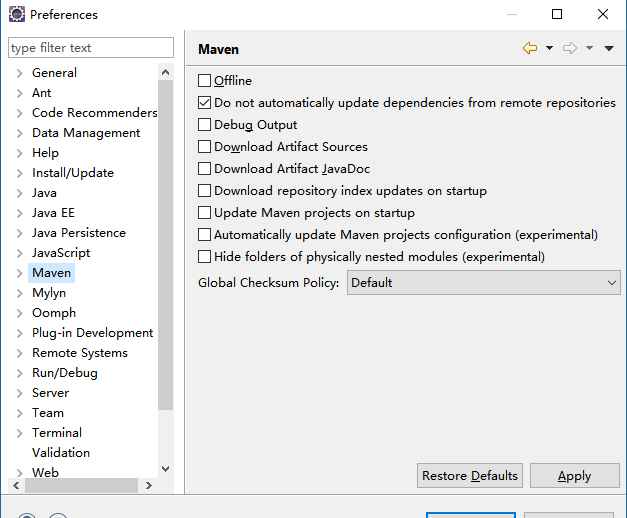


图4-3 Preferences-Maven

很凑巧呢，该版本的Eclipse里面，内置了Maven呢，也就是m2eclipse插件。接下来咱们就没必要自己再去下载m2eclipse插件了，咱只要配置配置就可以使用Maven了。

1. 展开上图中Maven选项，选中其中的Installations，显示如图4-4：

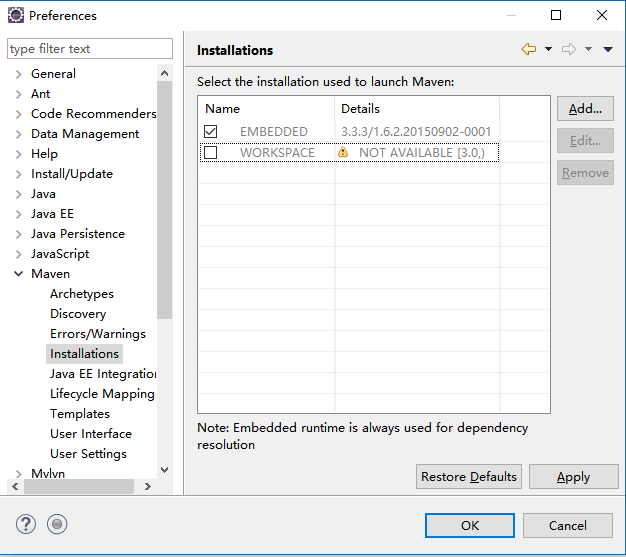


图4-4 Maven-Installations

右边显示的是当前Eclipse内置的Maven。为了整合我们前面自己安装的最新Maven，继续下面步骤。

1. 点击上图右边的 Add按钮，弹出New Maven Runtime窗口，选择咱们装的Maven，如图4-5：

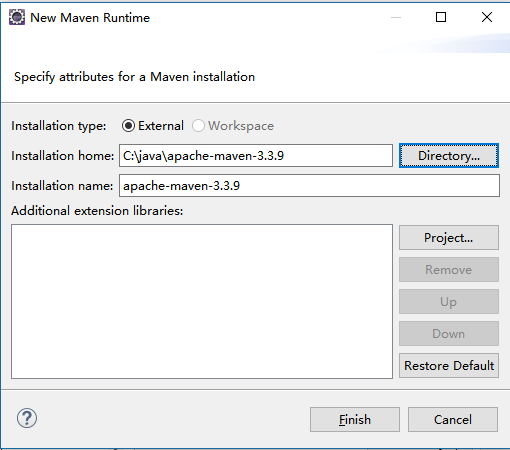


图4-5 设置Maven安装Home目录

在Installation home后面，选择咱们以前安装的Maven目录，点击Finish按钮，回到Preferences窗口，并且选中刚刚添加的Maven。

保险起见，点击 Preferences窗口中的 Apply按钮，保存刚刚的操作。

1. 选中Preferences中Maven里面的User Setting，显示如图4-6：

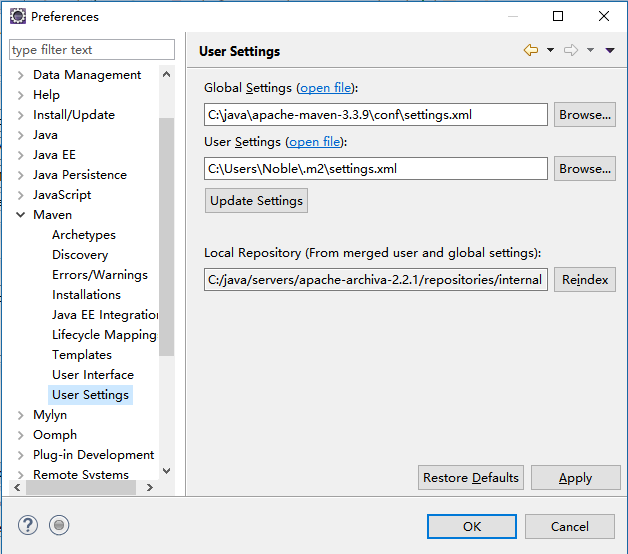


图4-6 设置Maven用户环境

在Global Setting框中选中咱们安装的Maven下的settings.xml

在User Setting框中选中咱们自己登录用户目录下的settings.xml

一般是会默认显示的，如果不正常，自己手动选择。

图片上显示的是我电脑上的位置文件。

1. 点击OK完成设置
2. 点击菜单中的 File->Import选项，打开Import窗口如图4-7：

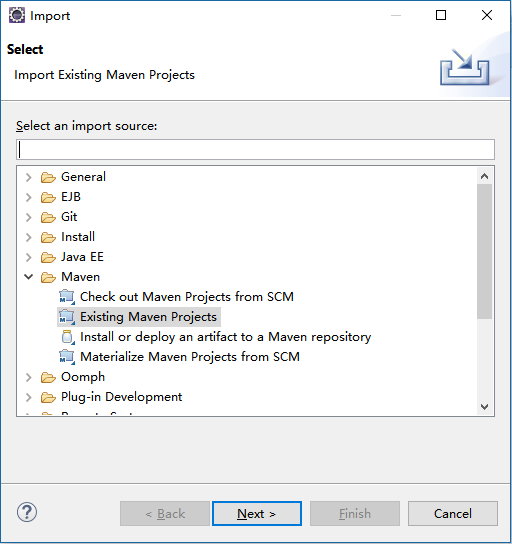


图4-7 导入Maven项目开始界面

1. 选择上图里面的 Maven->Existing Maven Projects选项，点击 Next按钮，在弹出的Import Maven Projects窗口中，选中咱们前面创建的Maven工程目录，如图4-8：

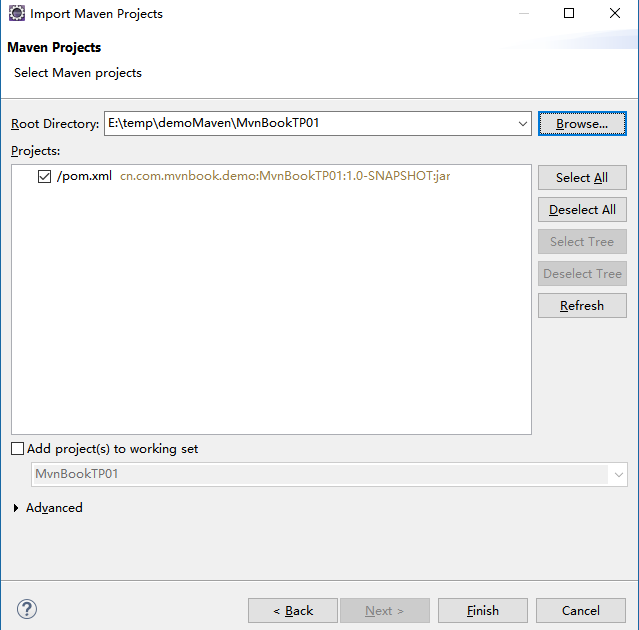


图4-8 选择导入Maven工程

1. 点击 Finish按钮，咱们就可以在Eclipse中看到以前好不容易手动折腾出来的Maven工程，如图4-9：

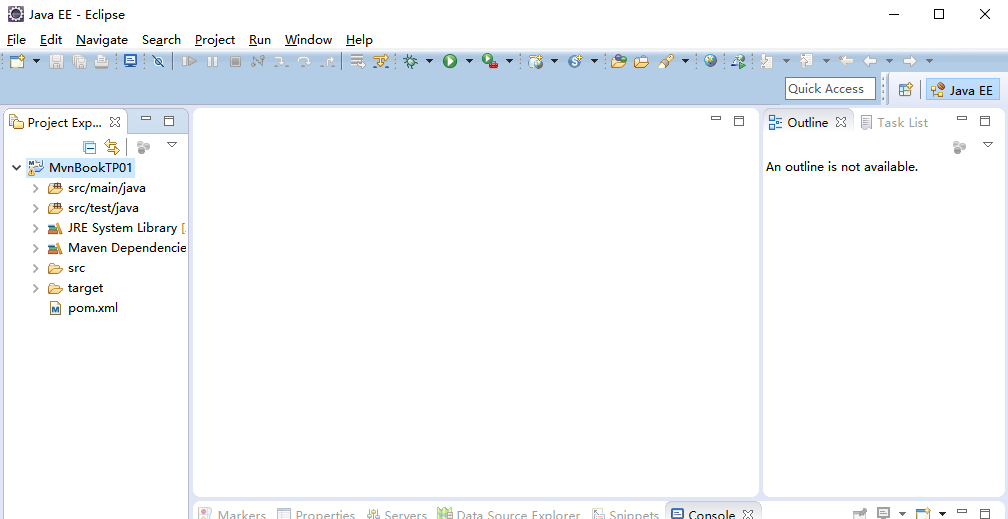


图4-9 Eclipse中Maven工程目录

1. 在Eclipse的工程上，右击鼠标，选择Run As->Maven test，就可以完成咱们一起mvn test命令完成的运行测试代码，在Eclipse的Console窗口中可以看到结果。如图4-10：

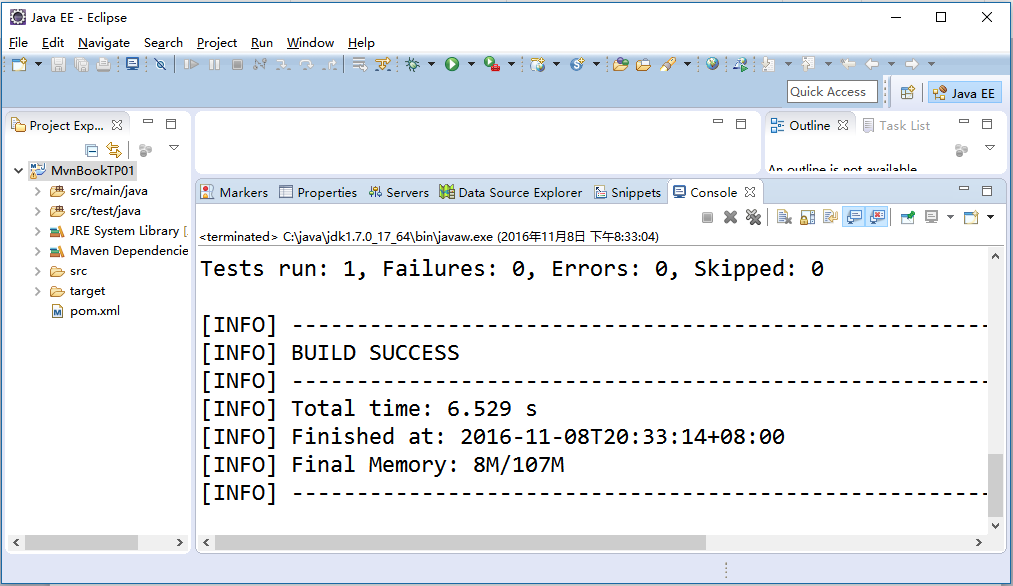


图4-10 运行Maven test提示

## 4.3 创建一个Maven项目

1. 点击Eclipse的File->New->Others，或点击File菜单下的快捷图标，打开New窗口如图4-11：

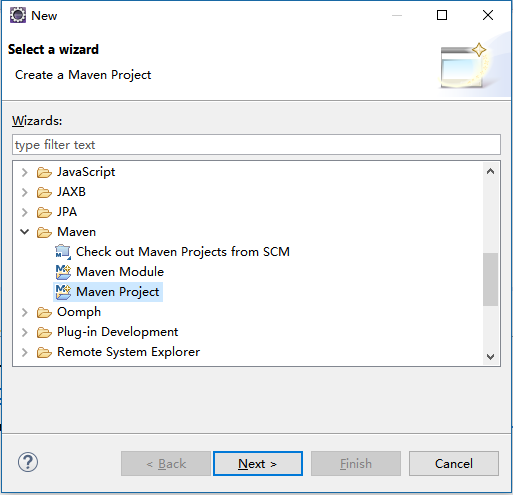


图4-11选择创建Maven工程

1. 选择Maven中的Maven Project，点击 Next 按钮，打开New Maven Project窗口，不做改动，继续点击 Next 按钮，出现如图4-12界面：

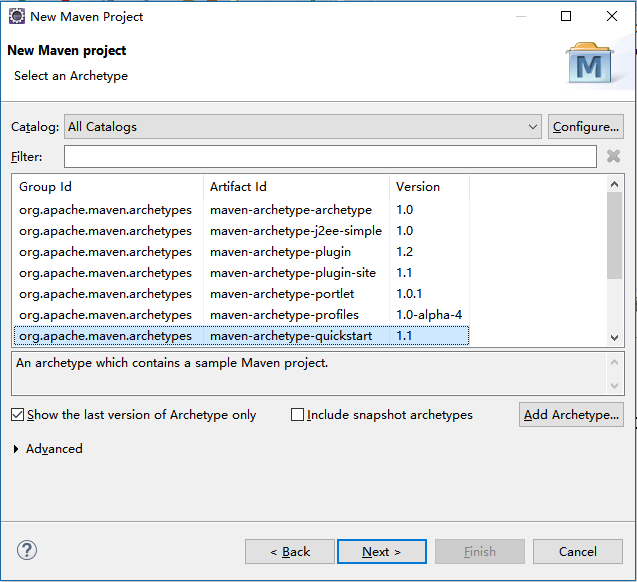


图4-12选择Archetype

列表中显示的是当前常用的archetype插件的groupId，artifactId和版本号，archetype插件，是用来帮咱们创建工程的，这功能我们在前面手动创建工程的时候使用过。

1. 选中1.1版本的quickstart插件，如上图选项。点击 Next 按钮，出现下一个窗口如下图4-13。在窗口中输入咱们新创建工程的groupId、artifactId、packageName和选择版本。

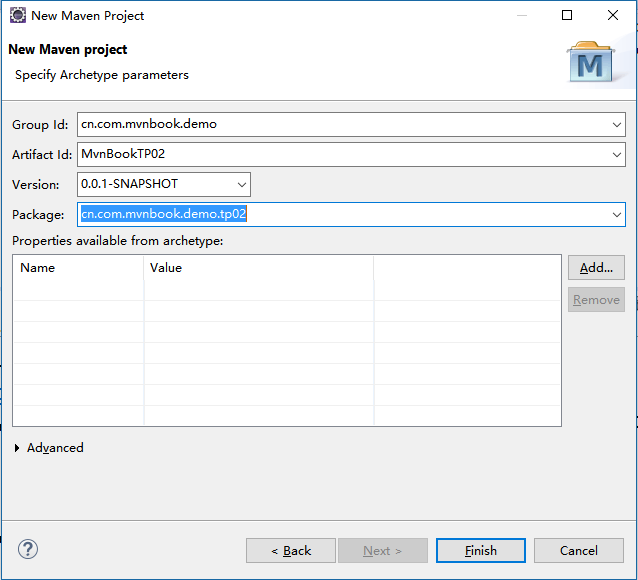


图4-13 Maven工程坐标

1. 点击 Finish 按钮，创建出一个新的Maven项目，如图4-14。

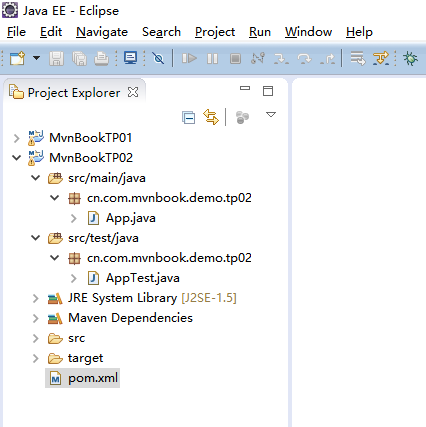


图4-14 新建Maven结构图

咱们就可以分别在src\main\java中和src\test\java中，添加咱们的Java源代码和测试代码。

为了方便，我这里直接将上次手动敲的HelloWorld.java和TestHelloWorld.java复制过来，同时把pom.xml中jUnit的版本改成4.7(默认生成的是3.8.1)

（随书代码[MvnBookTP02](代码/MVNBookTP02)）

到现在为止，咱们就用Eclipse+M2Eclipse插件，完成了Maven项目的创建和相关代码的编写。

## 4.4 构建Maven项目

现在工程创建好，相关的代码也写好了，接下来就是程序员的老套路：清理旧操作、编译源代码、运行测试案例、打包安装了。

以前咱们是用的mvn clean|compile|test|install命令实现的，现在就在Eclipse中，点击命令选项就行。

鼠标右击左边的工程，点击Run As->Maven clean|test|install，如下图4-15，就可以完成该工程清理、测试和打包安装。

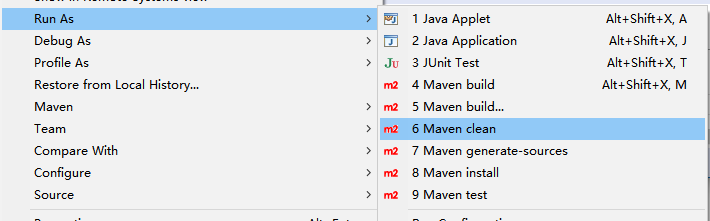


图4-15 基于m2eclipse构建maven项目

当然，这里面缺少一个编译（compile）的选项。

这也是正常的，Eclipse一般都是自动编译的，而且在运行test之前，它都会把所有代码重新编译一遍。

如果我们一定要明确的做编译的动作，也是可以的。选择上图Run As后面的 Maven build...选项，（注意，是后面带了... 的选项哦），弹出如下图4-16，在Goals后面的输入框中输入compile命令，点击Run 按钮，就会执行编译操作。

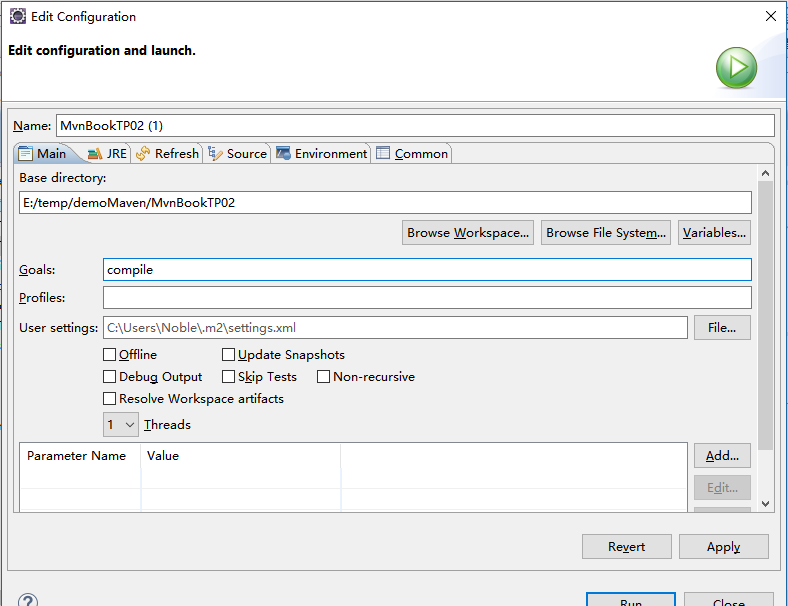


图4-16 Maven Build...界面

到现在为止，咱们就把项目构建相关的动作，都折腾完了。这比以前手动使用mvn命令方便多了吧 ^\_^

那以前咱们有生成过站点文档和API Doc文档啥的呢，在Eclipse+m2eclipse中又怎么体现呢？

不急，咱们下节慢慢道来。

## 4.5 基于M2Eclipse完成所有工作

前面我们有基于Eclipse+m2eclipse完成了工程的基本构建，接下来咱们研究一下怎么样生成相关的文档和报告。

在第三章小结的时候，咱们有体验到这么点：Maven是依赖插件去做具体的事情的。比如创建工程、生成站点和生成JavaDoc API文档，这些都是有专门的插件。

咱们要做的事情，是告诉Maven使用哪个插件，Maven自动协调对应插件去完成特定的工作任务。

在第三章手动操作的时候，咱们是用mvn命令，后面指定插件的坐标(没有自定，mvn自动找仓库中最新的)来告知mvn做什么事，那在Eclipse+m2eclipse环境下，咱们通过什么样的方式，可以告诉maven使用哪些插件做哪些事情呢？

pom.xml骨架文件，对，pom.xml骨架文件就是做这事情的。咱们要做的就是在pom.xml文件中，按它的语法要求指定相关的插件坐标，maven在运行的时候，自动使唤插件完成对应的任务。

那怎么在pom.xml中描述这些插件的坐标，怎么样启动maven，让它可以协调指定的插件做事呢？

别急别急，写配置文件和启动maven，那不是分分钟钟的事情呀^\_^

咱们先了解一下常用的插件有哪些，以及怎么样确定它们的坐标，这样才好在pom.xml中描述吧。

|  |  |  |
| --- | --- | --- |
| 插件名称 | 用途 | 来源 |
| maven-clean-plugin | 清理项目 | Apache |
| maven-compile-plugin | 编译项目 | Apache |
| maven-deploy-plugin | 发布项目 | Apache |
| maven-site-plugin | 生成站点 | Apache |
| maven-surefire-plugin | 运行测试 | Apache |
| maven-jar-plugin | 构建jar项目 | Apache |
| maven-javadoc-plugin | 生成javadoc文件 | Apache |
| maven-surefire-report-plugin | 生成测试报告 | Apache |

稍稍列举几个吧，更多的咱们在后面的附录里面列出。

接下来咱们在咱们的工程里面，体验体验生成站点、运行测试、生成javadoc和生成测试报告插件的使用。

咱们一个个的来，先折腾一下运行测试。

1. 在Maven Repository（仓库）中找到surefire插件的坐标。

用浏览器打开http://mvnrepository.com/，在search输入框中输入surefire-plugin,点击search如下图4-17：

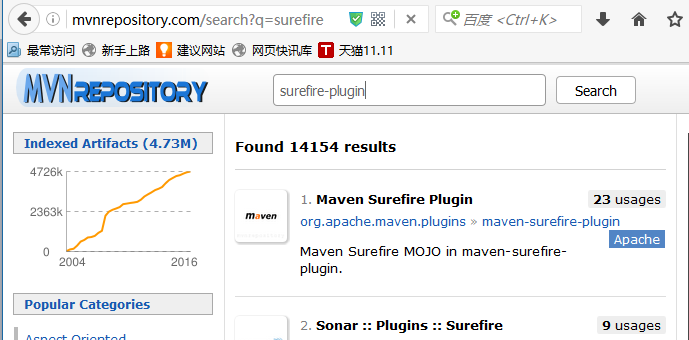


图4-17 中央仓库查询surefire插件

点击maven-surefire-plugin，查看它的所有版本信息，点击想使用的版本（咱们这里使用的2.19.1），会显示该版本的groupId、artifactId等坐标信息。如图4-18：

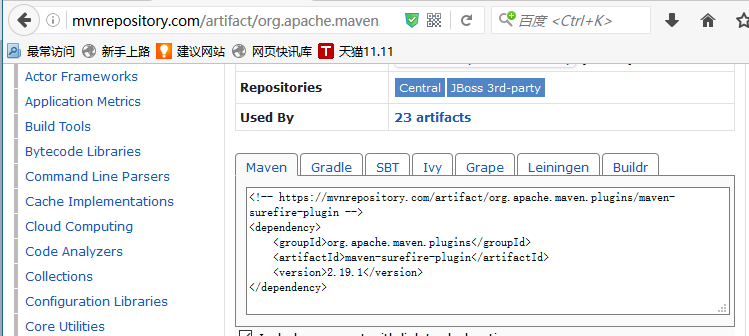


图4-18 surefire插件的坐标页面

1. 在pom.xml中添加surefire插件描述。

在Eclipse中打开pom.xml文件，在pom.xml后面添加一个build标签，里面添加surefire插件的描述信息，具体内容如下，被粗体显示的为新添加的内容。  
（随书代码[MvnBookTP02\ pom.xml](代码/MVNBookTP02/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.com.mvnbook.demo</groupId>

<artifactId>MvnBookTP02</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>MvnBookTP02</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.7</version>

<scope>test</scope>

</dependency>

</dependencies>

**<build>**

**<plugins>**

**<plugin>**

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

**<artifactId>maven-surefire-plugin</artifactId>**

**<version>2.19.1</version>**

**<configuration>**

**<!--设置包含的测试类 -->**

**<includes>**

**<include>\*\*\*\*\*\*</include>**

**</includes>**

**<!-- 设置不进行测试类 -->**

**<excludes>**

**<exclude>Test\*</exclude>**

**</excludes>**

**<!-- 跳过测试阶段，測試類写的有问题也会出错，一般不推荐 -->**

**<!--<skip>true</skip> -->**

**</configuration>**

**</plugin>**

**</plugins>**

**</build>**

</project>

上面内容中，用红色标记的，就是咱们要配置的surefire-plugin的描述，具体的描述方式和说明，上面有注释了，这里就不累赘了。

1. 启动Maven，运行test

鼠标右键工程，选择Run As->Maven test。Maven会自动启动插件进行编译和测试。如果第一次运行测试，我们在控制台会发现如下下载信息，说明Maven将用到咱们刚刚配置的surefire-plugin运行测试代码。

[INFO] --- maven-surefire-plugin:2.19.1:test (default-test) @ MvnBookTP02 ---

[INFO] Downloading: <https://repo.maven.apache.org/maven2/org/apache/maven/surefire/maven-surefire-common/2.19.1/maven-surefire-common-2.19.1.pom>

好了，咱们用surefire-plugin体验了一下，在pom.xml中添加插件，并且启动maven执行插件的过程，解析来我们再重复前面的故事，再折腾一下javadoc-plugin，用来生成javadoc API帮助文档。

1. 查找合适版本的坐标。

在mvnrepository.com中，类似查找surefire-plugin的方式，输入javadoc-plugin查询，找到自己中意版本的坐标信息。

1. 将javadoc-plugin添加到pom.xml中

在pom.xml的plugins标签之间，添加如下内容：

<!-- 项目API doc报告 -->

<plugin>

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

<artifactId>maven-javadoc-plugin</artifactId>

<version>2.7</version>

<configuration>

<aggregate>true</aggregate>

</configuration>

<executions>

<execution>

<id>attach-javadocs</id>

<goals>

<goal>jar</goal>

</goals>

<!--执行maven test的时候运行插件-->

<phase>test</phase>

</execution>

</executions>

</plugin>

这里面有javadoc-plugin的坐标信息外，还有其它配置信息，咱们先不管。中间有红色标记的信息，它的意思是当我们点击Run As->Maven test命令菜单的时候，执行javadoc插件，生成doc帮助文档。

1. 运行javadoc-plugin插件，查看doc ap文档

右键工程 点击 Run As->Maven test，maven自动调用插件生成API文档。在工程的target目录下，会自动产生一个apidocs目录，里面就生成的API文档。

前面介绍了2个插件的坐标查找和配置，后面关于站点和测试报告的，咱们就不再做过程的累赘了，直接贴插件描述的配置，里面有关键点的中文注释，大家把它们移到咱们自己的pom.xml中，测试运行一把，就可以看到对应的执行结果。

剩下插件描述内容：

<!-- 构建项目站点报告插件 -->

<plugin>

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

<artifactId>maven-site-plugin</artifactId>

<version>3.0-beta-3</version>

<configuration>

<!-- 配置站点国际化 -->

<locales>zh\_CN</locales>

<!-- 输出编码 -->

<outputEncoding>GBK</outputEncoding>

</configuration>

</plugin>

<!-- 单元测试报告html -->

<plugin>

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

<artifactId>maven-surefire-report-plugin</artifactId>

<version>2.12.2</version>

<configuration>

<showSuccess>true</showSuccess>

</configuration>

<executions>

<execution>

<id>test-report</id>

<phase>test</phase>

</execution>

</executions>

</plugin>

<!-- 测试覆盖率的报告 -->

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>cobertura-maven-plugin</artifactId>

<version>2.5.1</version>

<configuration>

<formats>

<format>html</format>

<format>xml</format>

</formats>

</configuration>

<executions>

<execution>

<id>cobertura-report</id>

<goals>

<goal>cobertura</goal>

</goals>

<phase>test</phase>

</execution>

</executions>

</plugin>

## 4.6 小结

通过本章的内容介绍，咱们完全可以将用手动搞定的事情，在Eclipse+m2eclipse环境下搞定了。整个操作的思路和目的都是一样的，只是有点不同的是操作形式图形化了，更加方便了，这也是咱们研究m2eclipse的初衷不是。

当然，还有一个变化，那就是咱们是用的pom.xml事先描述好要使用的插件，maven在执行相关命令的时候，自动再协调相关插件，完成对应任务。

也就是说，我们后期还会有怎么样准确描述插件的标记的研究，那是后话。现在咱们只要将在配置过程中，用中文注释过的标记理解清楚就行，后面的后面在去弄，一步步来嘛 ^\_^

# 第五章 基于Maven开发Web应用

前面章节，咱们有研究怎么样在Eclipse环境下，结合m2eclipse创建、编译、测试、打包、安装一个基本的java项目了，当然还包括几个常用文档的生成。但是，目前Java程序员，最常要面对的是JavaWeb应用，那咱们得折腾明白，eclipse+m2eclipse怎么样开发管理JavaWeb应用不是。

本章内容，咱们将弄明白怎么样折腾JavaWeb应用的套路，主要针对的是最基本的JSP/Servlet Web应用。

## 5.1 开发Web应用的思路

一切还是以实际案例进行。不过分两步：

第一步：实现一个简单的jsp/servlet，有点MVC感觉的HelloWorld

在这个环节，咱们主要体现：

1. 怎么搭建创建Web应用工程的环境
2. 怎么样创建Web应用工程
3. Web应用工程的目录结构是怎样的
4. 怎样结合Web服务器，发布Web应用
5. 体验Web应用的开发和发布测试过程

第二步：实现经典的MVC版本的用户CRUD

1. 熟练第一步中的几个方面
2. 结合典型的业务逻辑，实现CRUD

## 5.2 实现Web版HelloWorld

### 5.2.1 安装配置web应用的Archetype Catalog

目标确定好了吧，前面配置好了的Eclipse也启动完备了啦。

好叻，咱们开始创建Web应用了啦。

按前面创建普通Java工程的步骤：

点击 File->New->Others

选择 Create Maven Project，点击 下一步

选中创建Web应用工程的archetype，如图5-1：

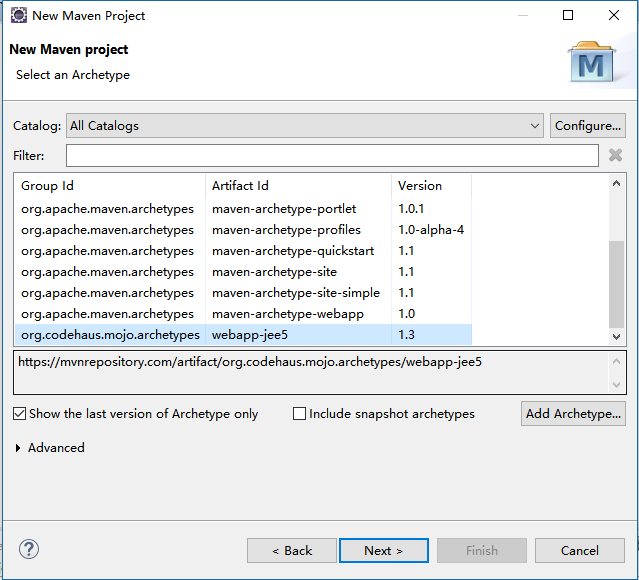


图5-1 选择web archetype

如果没有同我一样的选项，可以选择其它类似的，创建web应用的都可以，比如前面那个叫maven-archetype-webapp的也可以。当然，咱们也可以选择从网上找到坐标后的archetype插件，再安装进去。

怎么安装新的archetype呢，简单着呢，点击上图中的 Add Archetype 按钮，在出现的窗口中输入在网上找到的插件坐标信息。如图5-2：

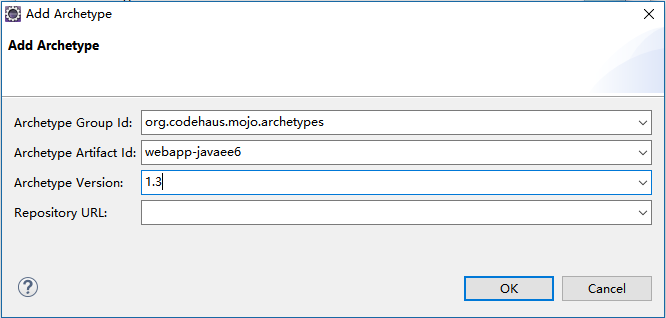


图5-2 添加Archetype

点击OK按钮，Eclipse自动会下载该构件。重新打开创建工程的向导页面，咱们就可以发现新增了刚刚添加的archetype插件。如图5-3：

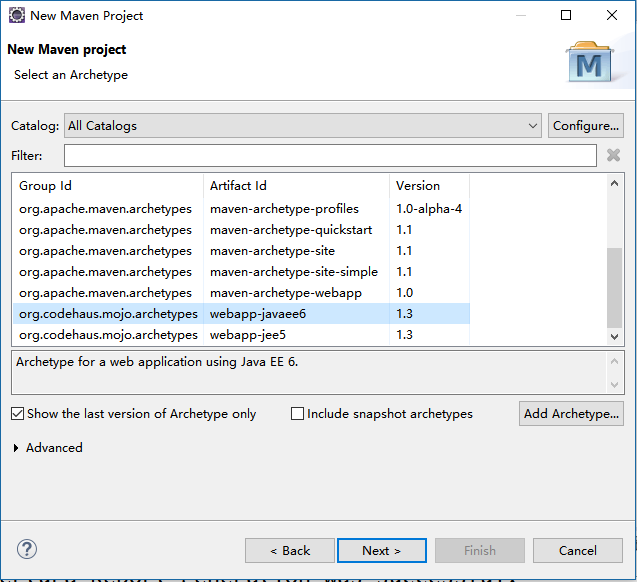


图5-3 选择webapp-javaee6 Archetype

### 5.2.2 基于ArcheType向导创建Web工程

继续上节的创建工程的向导页面，我还是比较念旧jee5的，选中jee5下一步，在下一个界面中输入咱们新创建的Web工程的坐标信息和包名。如图5-4：

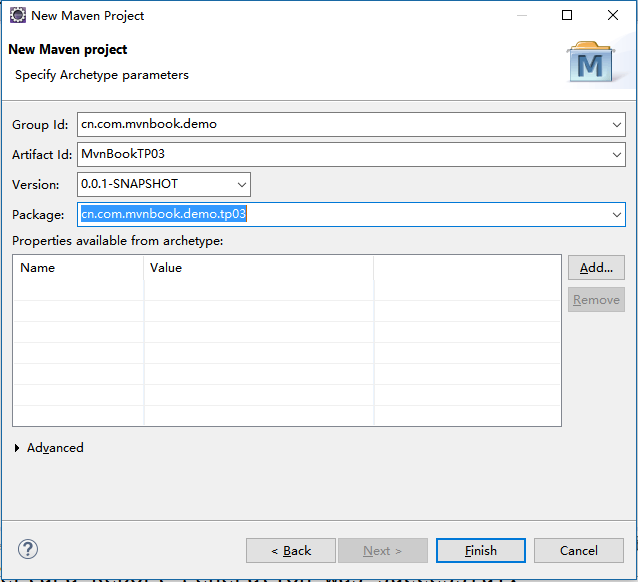


图5-4 maven项目坐标

点击Finish按钮，m2eclipse会自动给咱创建一个Web工程MvnBookTP03，目录结构图5-5如下。其它没什么，就是在src\main目录下，添加了webapp目录，里面有我们久违的Web应用特有的WEB-INF目录,web.xml和index.jsp等。

其中webapp目录和里面的文件以及结构，在maven中也是固定的。

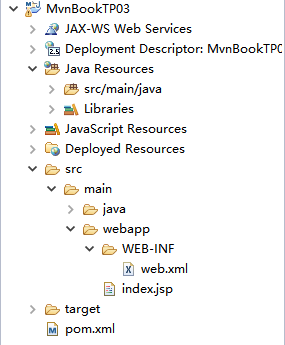


图5-5 Maven Web项目目录结构

咱们就创建好了Web应用工程，可以在里面写咱们的HelloWorld 案例的Servlet和jsp了  
（随书代码[MvnBookTP03](代码/MvnBookTP03)）

### 5.2.3 编写样例代码

工程创建好了，咱们开始写测试代码吧。

在写代码前，咱先罗嗦一句。接下来咱们是写jsp和servlet，是有专门的页面向导，协作咱们快速的创建jsp和servlet代码的架构的。如果您不是使用的我前面介绍的eclipse-jee-mars-2-win32-x86\_64.zip中的Eclipse，或者您使用的是纯Java IDE，那恭喜您，有两个选择：一是在您的路上继续下去，所有代码都手敲出来呗，工具不能帮你生成框架代码嘛；另外一个就是老老实实下载eclipse-jee-mars-2-win32-x86\_64.zip吧 ^\_^

接下来，咱们写3个代码：2个jsp和一个servlet。

Index.jsp,里面显示输入框，能提交我们输入的问候者名称

Welcome.jsp，显示对问候者的问候信息

WelcomeServlet，接收index.jsp发过来的名称，生成问候信息，转给welcome.jsp显示

当然，出来写前面的代码外，咱们还需要配置web.xml，servlet的，不要忘记了。

接下来就开始写代码了。具体写代码过程就不玩了，直接贴代码内容：

1、src\main\webapp\index.jsp  
（随书代码[MvnBookTP03\ src\main\webapp\index.jsp](代码/MvnBookTP03/src/main/webapp/index.jsp)）

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Index JSP</title>

</head>

<body>

<form action="welcomeServlet" method="post">

请输入问候人名:<input type='text' name="name"/><br/>

<input type='submit' value='问候'/>

</form>

</body>

</html>

1. src\main\webapp\welcome.jsp  
   （随书代码[MvnBookTP03\ src\main\webapp\welcome.jsp](代码/MvnBookTP03/src/main/webapp/welcome.jsp)）

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Welcome JSP</title>

</head>

<body>

问候信息：${welcome }

</body>

</html>

1. src\main\java\....\WelcomeServlet.java  
   （随书代码  
    [MvnBookTP03\src\main\java\cn\com\mvnbook\demo\tp03\WelcomeServlet.java](代码/MvnBookTP03/src/main/java/cn/com/mvnbook/demo/tp03/WelcomeServlet.java)）

package cn.com.mvnbook.demo.tp03;

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

/\*\*

\* Servlet implementation class WelcomeServlet

\*/

public class WelcomeServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

/\*\*

\* @see HttpServlet#service(HttpServletRequest request, HttpServletResponse response)

\*/

protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

request.setCharacterEncoding("UTF-8");

String name = request.getParameter("name");

String welcome = "Hello," + name;

request.setAttribute("welcome", welcome);

request.getRequestDispatcher("/index.jsp").forward(request, response);

}

}

1. 在src\main\webapp\WEB-INF\web.xml中添加WecomeServlet的配置  
   （随书代码[MvnBookTP03\src\main\webapp\WEB-INF\web.xml](代码/MvnBookTP03/src/main/webapp/WEB-INF/web.xml)）

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" version="2.5">

<display-name>MvnBookTP03</display-name>

<session-config>

<session-timeout>30</session-timeout>

</session-config>

<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

**<servlet>**

**<description></description>**

**<display-name>WelcomeServlet</display-name>**

**<servlet-name>WelcomeServlet</servlet-name>**

**<servlet-class>cn.com.mvnbook.demo.tp03.WelcomeServlet</servlet-class>**

**</servlet>**

**<servlet-mapping>**

**<servlet-name>WelcomeServlet</servlet-name>**

**<url-pattern>/WelcomeServlet</url-pattern>**

**</servlet-mapping>**

</web-app>

### 5.2.4 构建Web项目

前期的构建过程，同前面基本的java工程一样，根据我们自己的需要，在pom.xml中配置好对应功能的插件，再运行对应的图形化菜单命令就可以了，我们在这里不做重复的说明。

一个Web应用构建好后，不仅仅只是编译打包安装就可以了，还需要将它发布到Web服务器中进行测试调试才行。我们这里主要介绍两种发布到Tomcat7.x服务器，启动测试的方式。在项目开发过程中，我们可以根据自己的需要，选择其中一种：

1. 使用Maven的Jetty插件部署Web
2. 在pom.xml中添加Jetty插件的坐标信息，内容如下：  
   （随书代码[MvnBookTP03\pom.xml](代码/MvnBookTP03/pom.xml)）

<plugin>

<groupId>org.mortbay.jetty</groupId>

<artifactId>maven-jetty-plugin</artifactId>

<version>6.1.26</version>

<configuration>

<webAppSourceDirectory>${basedir}/src/main/webapp</webAppSourceDirectory>

</configuration>

</plugin>

1. 在Eclipse中配置Web服务器运行环境
2. 点击Eclipse的菜单Window->Preferences，打开Preferences窗口，选中左边树Server->Runtime Environment。如下图5-6：

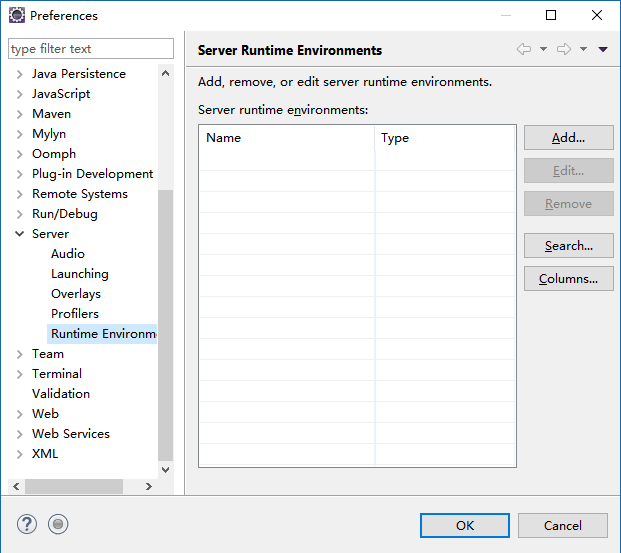


图5-6 Eclipse的web服务器

1. 点击右边的 Add按钮，弹出一个选择服务器的窗口，选中窗口中的Apache->Apache Tomcat v7.0服务器（我的测试环境是Tomcat7.x）。如图5-7：

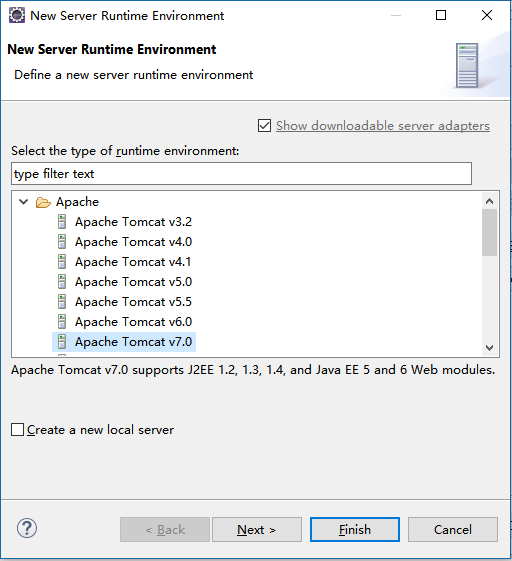


图5-7 添加Tomcat7

1. 点击 Next，进入选择TomatServer配置页面，选择咱们Tomcat的安装目录和JRE运行环境（咱们自己JDK）。如图5-8：

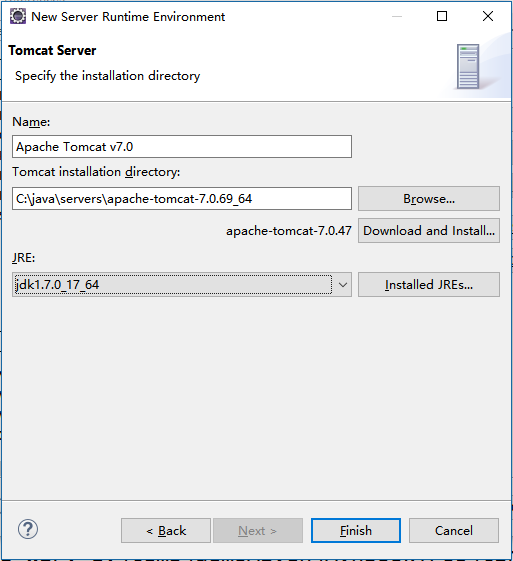


图5-8 添加Tomcat的Java home

1. 点击 Finish 和 OK，关闭所有配置窗口，完成Eclipse中的WebServer配置
2. 运行

右击工程，选择 Run As->Maven build ...，打开自定义launch窗口，在Goals中输入启动的插件名和目标：jetty:run。如图5-9：

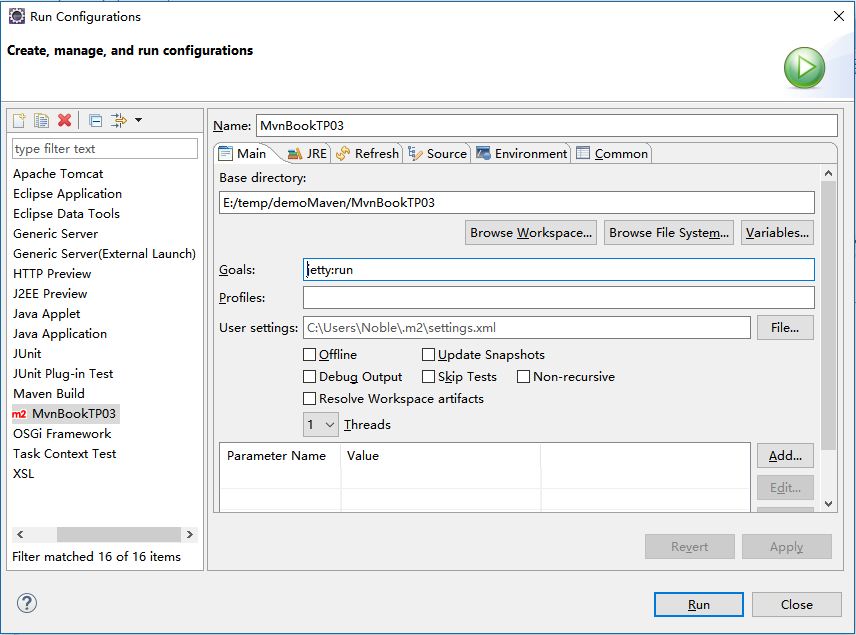


图5-9 运行jetty

点击Run运行一次后，以后每次都可以在 Run As->Maven build选项中选择重复运行。

服务器启动好了，接下来打开浏览器，输入http://localhost:8080/MvnBookTP03/index.jsp，就可以访问第一个页面了。后面的页面访问测试，相信大家有基本的jsp/servlet基础，都可以知道操作了。

1. 使用cargo-maven2-plugin插件部署Web

使用cargo插件相对简单，我们只需要在pom.xml进行配置，指定部署应用必须的信息，再运行Run As->Maven install命令，cargo插件自动会把打成war包的应用，发布到指定的web服务器的发布目录下。

接下来要做的是我们自己启动Web服务器，按以前的方式打开浏览器浏览页面。

Gargo在pom.xml中的插件配置内容如下：  
（随书代码[MvnBookTP03\pom.xml](代码/MvnBookTP03/pom.xml)）

<plugin>

<!-- 指定插件名称及版本号 -->

<groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>

<version>1.4.8</version>

<configuration>

<!--是否说明，操作start、stop等后续操作必须等前面操作完成才能继续 -->

<wait>true</wait>

<!-- 容器的配置 -->

<container>

<!-- 指定tomcat版本 -->

<containerId>tomcat7x</containerId>

<!-- 指定类型：standalone, installed等 -->

<type>installed</type>

<!-- 指定Tomcat的位置，即catalina.home -->

<home>C:\java\servers\apache-tomcat-7.0.69\_64</home>

</container>

<!-- 具体的配置 -->

<configuration>

<!-- 类型，existing:存在 -->

<type>existing</type>

<!-- Tomcat的位置，即catalina.home -->

<home>C:\java\servers\apache-tomcat-7.0.69\_64</home>

</configuration>

<deployables> <!-- 部署设置 -->

<deployable> <!-- 部署的War包名等 -->

<groupId>cn.com.mvnbook.demo</groupId>

<artifactId>MvnBookTP03</artifactId>

<type>war</type>

<properties>

<context>MvnBookTP03</context> <!-- 部署路径 -->

</properties>

</deployable>

</deployables>

<deployer> <!-- 部署配置 -->

<type>installed</type> <!-- 类型 -->

</deployer>

</configuration>

<executions>

<!-- 执行的动作 -->

<execution>

<id>verify-deployer</id>

<phase>install</phase> <!-- 解析install -->

<goals>

<goal>deployer-deploy</goal>

</goals>

</execution>

<execution>

<id>clean-deployer</id>

<phase>clean</phase>

<goals>

<goal>deployer-undeploy</goal>

</goals>

</execution>

</executions>

</plugin>

当我们在工程上点鼠标右键，选择 Run As ->Maven install后，我们就可以在Tomcat7的发布目录下，发现MvnBookTP03.war，启动后它就能自动发布并能被访问。

### 5.2.5 测试

不管前面哪种方式，启动服务器后，打开浏览器，输入http://localhost:8080/MvnBookTP03/index.jsp连接后，会出现如图5-10页面，后面的操作相信大家都能自己完成了。

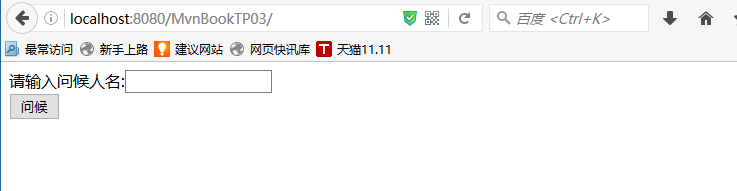


图5-10 index.jsp页面

## 5.3 基于Maven开发用户模块的CRUD

前面介绍了怎么样实现一个简单的Web应用，体现了创建Web应用、编写代码、在pom.xml中配置相关的构件、最后发布测试。接下来咱们再介绍一个经典的实现了CRUD的用户Web管理应用。

### 5.3.1 创建Web工程和初始化数据库

首先，麻烦您按前面章节的方式，创建一个Web工程。因为创建Web工程的套路依旧，我这就不再重复了，直接贴出我这Demo工程的目录结构。如下图5-11：

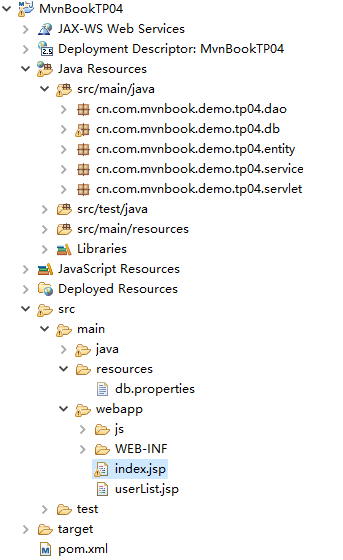


图5-11 Maven Web工程目录

（随书代码[MvnBookTP04](代码/MvnBookTP04)）

创建好了工程，当然还要初始化好数据库啦

我这用的是MySQL数据库，建议各位先安装好数据库，然后创建一个数据库，用如下脚本初始化表。  
（随书代码[mysql\_script.sql](代码/mysql_script.sql)）

CREATE TABLE mvn\_user(

ur\_id int(11) NOT NULL AUTO\_INCREMENT,

ur\_user\_name varchar(255) DEFAULT NULL,

ur\_password varchar(255) DEFAULT NULL,

ur\_age int(11) DEFAULT NULL,

ur\_status varchar(255) DEFAULT NULL,

PRIMARY KEY (ur\_id)

) ENGINE=InnoDB AUTO\_INCREMENT=15 DEFAULT CHARSET=utf8;

INSERT INTO mvn\_user(ur\_user\_name,ur\_password,ur\_age,ur\_status) VALUES ('zhangsan', '123', 11, 'Active');

INSERT INTO mvn\_user(ur\_user\_name,ur\_password,ur\_age,ur\_status) VALUES ('lisi', '123', 13, 'Inactive');

INSERT INTO mvn\_user(ur\_user\_name,ur\_password,ur\_age,ur\_status) VALUES ('wangwu', '123', 13, 'Active');

### 5.3.2 添加相关依赖

在整个Demo应用中，我们需要在创建web工程后，额外需要四个依赖，它们分别是jstl依赖、MySql数据库驱动依赖、jUnit4.7依赖和json-lib依赖，它们的依赖配置文件如下：

（随书代码[MvnBookTP04\pom.xml](代码/MvnBookTP03/pom.xml)）

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.34</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.7</version>

<scope>test</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/net.sf.json-lib/json-lib -->

<dependency>

<groupId>net.sf.json-lib</groupId>

<artifactId>json-lib</artifactId>

<version>2.4</version>

<classifier>jdk15</classifier>

</dependency>

### 5.3.3 添加注册代码

Demo带文件主要有如下几个：

MvnUser.java 用户实体类

DBConnection.java 连接数据库的公共类

MvnUserDAO.java 用户的DAO持久层类

UserService.java 用户服务类

AddUserServlet.java 添加用户Servlet

DeleteUserServlet.java 删除用户Servlet

EditUserServlet.java 修改用户Servlet

SearchUserServlet.java 根据用户Id或用户名查找用户Servlet

SearchUsersServlet.java 查询所有用户Servlet

userList.jsp 显示用户列表jsp

Index.jsp 进入首页（框架jsp）

db.properties 数据库信息配置文件

它们的内容分别如下：

MvnUser.java  
（随书代码  
[MvnBookTP04\MvnBookTP04\src\main\java\cn\com\mvnbook\demo\tp04\entity\MvnUser.java](代码/MvnBookTP04/src/main/java/cn/com/mvnbook/demo/tp04/entity/MvnUser.java)）

package cn.com.mvnbook.demo.tp04.entity;

public class MvnUser {

private int urId;

private String urUserName;

private String urPassword;

private int urAge;

private String urStatus;

public int getUrId() {

return urId;

}

public void setUrId(int urId) {

this.urId = urId;

}

public String getUrUserName() {

return urUserName;

}

public void setUrUserName(String urUserName) {

this.urUserName = urUserName;

}

public String getUrPassword() {

return urPassword;

}

public void setUrPassword(String urPassword) {

this.urPassword = urPassword;

}

public int getUrAge() {

return urAge;

}

public void setUrAge(int urAge) {

this.urAge = urAge;

}

public String getUrStatus() {

return urStatus;

}

public void setUrStatus(String urStatus) {

this.urStatus = urStatus;

}

}

DBConnection.java  
（随书代码  
[MvnBookTP04\MvnBookTP04\src\main\java\cn\com\mvnbook\demo\tp04\db\DBConnection.java](代码/MvnBookTP04/src/main/java/cn/com/mvnbook/demo/tp04/db/DBConnection.java)）

package cn.com.mvnbook.demo.tp04.db;

import java.io.InputStream;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

import java.util.Properties;

/\*\*

\* 获取连接的一个工具类，继承Properties，实现如下封装：<br/>

\* 1 读取db.properties文件<br/>

\* 2 获取Connection连接的方法<br/>

\* 3 关闭资源的方法

\*

\* @author Noble

\* @version 1.0

\* \*/

public class DBConnection extends Properties {

private static DBConnection DB\_CONN = null;

/\*\*

\* 构造器方法，被私有化，封装读取db.properties逻辑

\*

\* \*/

private DBConnection() throws Exception {

InputStream in = DBConnection.class.getClassLoader()

.getResourceAsStream("db.properties");

this.load(in);

// 加载驱动类

Class.forName(this.getProperty("driverName"));

}

/\*\*

\* 单例模式实现，获取DBConnection实例的静态方法

\*

\* @return DBConnection DBConnection实例

\* @throws Exception 初始化db.properties出现问题时，会抛异常

\* \*/

public static DBConnection getInstance() throws Exception {

if (DB\_CONN == null) {

DB\_CONN = new DBConnection();

}

return DB\_CONN;

}

/\*\*

\* 基于驱动和db.properties中配置的连接数据库的信息，创建一个新连接返回

\*

\* @return Connection 创建的新连接对象

\*

\* \*/

public Connection getConnection() {

Connection conn = null;

String url = this.getProperty("url");

String userName = this.getProperty("userName");

String password = this.getProperty("password");

//

try {

conn = DriverManager.getConnection(url, userName, password);

} catch (Exception e) {

throw new RuntimeException("数据库连接错误，请与管理员联系");

}

return conn;

}

/\*\*

\* 关闭操作数据库后的资源

\*

\* @param conn Connection对象

\* @param stmt Statement或Statement的子类对象

\* @param rs ResultSet对象

\* \*/

public void close(Connection conn, Statement stmt, ResultSet rs) {

try {

if (rs != null)

rs.close();

if (stmt != null)

stmt.close();

if (conn != null)

conn.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

MvnUserDAO.java  
（随书代码  
[MvnBookTP04\MvnBookTP04\src\main\java\cn\com\mvnbook\demo\tp04\dao\MvnUserDAO.java](代码/MvnBookTP04/src/main/java/cn/com/mvnbook/demo/tp04/dao/MvnUserDAO.java)）

package cn.com.mvnbook.demo.tp04.dao;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.util.ArrayList;

import java.util.List;

import cn.com.mvnbook.demo.tp04.db.DBConnection;

import cn.com.mvnbook.demo.tp04.entity.MvnUser;

/\*\*

\* MvnUser实体对象的持久层代码，封装了对MvnUser实体对象的CRUD方法

\*

\* @author Noble

\* @version 1.0

\* \*/

public class MvnUserDAO {

/\*\*

\* 在数据库中，添加一个新的MvnUser对象

\*

\* @param user 需要添加的用户实体对象，该对象需要有用户名、密码、年龄和状态属性

\*

\* @return void

\* @throws RuntimeException 添加失败或出现其它意外

\* \*/

public void addUser(MvnUser user) {

DBConnection dbConn = null;

Connection conn = null;

PreparedStatement pstmt = null;

try {

// 获取DBConnection实例

dbConn = DBConnection.getInstance();

String sql = "insert into mvn\_user (ur\_user\_name,ur\_password,ur\_age,ur\_status) values(?,?,?,?)";

// 获取连接对象

conn = dbConn.getConnection();

// 基于连接和sql，获取一个预处理Statement对象

pstmt = conn.prepareStatement(sql);

// 设置sql中占位符的值

pstmt.setString(1, user.getUrUserName());

pstmt.setString(2, user.getUrPassword());

pstmt.setInt(3, user.getUrAge());

pstmt.setString(4, user.getUrStatus());

// 执行预处理

pstmt.executeUpdate();

} catch (Exception e) {

throw new RuntimeException(e);

} finally {

// 关闭资源

if (dbConn != null)

dbConn.close(conn, pstmt, null);

}

}

/\*\*

\* 更新MvnUser对象。该对象中需要设置年龄、状态和id属性，属性和状态是要更新的新值，id为条件

\*

\* @param user 需要更新的MvnUser对象

\*

\* @return void

\* @throws RuntimeException 更新失败或出现其它意外

\* \*/

public void update(MvnUser user) {

DBConnection dbConn = null;

Connection conn = null;

PreparedStatement pstmt = null;

try {

dbConn = DBConnection.getInstance();

String sql = "update mvn\_user set ur\_age=?,ur\_status=? where ur\_id=?";

conn = dbConn.getConnection();

pstmt = conn.prepareStatement(sql);

pstmt.setInt(1, user.getUrAge());

pstmt.setString(2, user.getUrStatus());

pstmt.setInt(3, user.getUrId());

pstmt.executeUpdate();

} catch (Exception e) {

throw new RuntimeException(e);

} finally {

if (dbConn != null)

dbConn.close(conn, pstmt, null);

}

}

/\*\*

\* 删除MvnUser对象，该对象中需要有要删除对象的id属性，id属性为删除条件

\*

\* @param user 要删除的MvnUser对象

\*

\* @return void

\* @throws RuntimeException 删除失败或出现其它意外

\* \*/

public void deleteUser(MvnUser user) {

DBConnection dbConn = null;

Connection conn = null;

PreparedStatement pstmt = null;

try {

dbConn = DBConnection.getInstance();

String sql = "delete from mvn\_user where ur\_id=?";

conn = dbConn.getConnection();

pstmt = conn.prepareStatement(sql);

pstmt.setInt(1, user.getUrId());

pstmt.executeUpdate();

} catch (Exception e) {

throw new RuntimeException(e);

} finally {

if (dbConn != null)

dbConn.close(conn, pstmt, null);

}

}

/\*\*

\* 根据id查询对应的MvnUser对象

\*

\* @param id 要查询的MvnUser对象的id

\* @return MvnUser id对应的MvnUser对象，如果没有对象，返回null

\* @throws RuntimeException 出现意外情况

\* \*/

public MvnUser findUserById(int id) {

MvnUser user = null;

DBConnection dbConn = null;

Connection conn = null;

PreparedStatement pstmt = null;

ResultSet rs = null;

try {

dbConn = DBConnection.getInstance();

String sql = "select \* from mvn\_user where ur\_id=?";

conn = dbConn.getConnection();

pstmt = conn.prepareStatement(sql);

pstmt.setInt(1, id);

rs = pstmt.executeQuery();

if (rs.next()) {

user = new MvnUser();

user.setUrAge(rs.getInt("ur\_age"));

user.setUrId(rs.getInt("ur\_id"));

user.setUrPassword(rs.getString("ur\_password"));

user.setUrStatus(rs.getString("ur\_status"));

user.setUrUserName(rs.getString("ur\_user\_name"));

}

} catch (Exception e) {

throw new RuntimeException(e);

} finally {

if (dbConn != null)

dbConn.close(conn, pstmt, rs);

}

return user;

}

/\*\*

\* 根据用户名查询对应的MvnUser对象

\*

\* @param userName 要查询的MvnUser对象的用户名

\* @return MvnUser 用户对应的MvnUser对象，如果没有对象，返回null

\* @throws RuntimeException 出现意外情况

\* \*/

public MvnUser findUserByUserName(String userName) {

MvnUser user = null;

DBConnection dbConn = null;

Connection conn = null;

PreparedStatement pstmt = null;

ResultSet rs = null;

try {

dbConn = DBConnection.getInstance();

String sql = "select \* from mvn\_user where ur\_user\_name=?";

conn = dbConn.getConnection();

pstmt = conn.prepareStatement(sql);

pstmt.setString(1, userName);

rs = pstmt.executeQuery();

if (rs.next()) {

user = new MvnUser();

user.setUrAge(rs.getInt("ur\_age"));

user.setUrId(rs.getInt("ur\_id"));

user.setUrPassword(rs.getString("ur\_password"));

user.setUrStatus(rs.getString("ur\_status"));

user.setUrUserName(rs.getString("ur\_user\_name"));

}

} catch (Exception e) {

throw new RuntimeException(e);

} finally {

if (dbConn != null)

dbConn.close(conn, pstmt, rs);

}

return user;

}

/\*\*

\* 查找数据库中所有的用户对象，以List集合的形式返回

\*

\* @return List<MvnUser> 所有用户对象的集合

\* @throws RuntimeException 出现意外情况

\* \*/

public List<MvnUser> findUsers() {

List<MvnUser> userList = null;

DBConnection dbConn = null;

Connection conn = null;

PreparedStatement pstmt = null;

ResultSet rs = null;

try {

dbConn = DBConnection.getInstance();

String sql = "select \* from mvn\_user order by ur\_id";

conn = dbConn.getConnection();

pstmt = conn.prepareStatement(sql);

rs = pstmt.executeQuery();

if (rs != null) {

userList = new ArrayList<MvnUser>();

MvnUser user = null;

while (rs.next()) {

user = new MvnUser();

user.setUrAge(rs.getInt("ur\_age"));

user.setUrId(rs.getInt("ur\_id"));

user.setUrPassword(rs.getString("ur\_password"));

user.setUrStatus(rs.getString("ur\_status"));

user.setUrUserName(rs.getString("ur\_user\_name"));

userList.add(user);

}

}

} catch (Exception e) {

throw new RuntimeException(e);

} finally {

if (dbConn != null)

dbConn.close(conn, pstmt, rs);

}

return userList;

}

}

UserService.java  
（随书代码  
[MvnBookTP04\MvnBookTP04\src\main\java\cn\com\mvnbook\demo\tp04\service\UserService.java](代码/MvnBookTP04/src/main/java/cn/com/mvnbook/demo/tp04/service/UserService.java)）

package cn.com.mvnbook.demo.tp04.service;

import java.util.List;

import cn.com.mvnbook.demo.tp04.dao.MvnUserDAO;

import cn.com.mvnbook.demo.tp04.entity.MvnUser;

/\*\*

\* 对用CRUD操作的服务层，封装了CRUD在持久化前的必要业务逻辑

\*

\* @author Noble

\* @version 1.0

\* \*/

public class UserService {

private MvnUserDAO userDAO = new MvnUserDAO();

/\*\*

\* 创建新的用户

\* @param user 要创建的用户对象

\* @return void

\* @throws RuntimeException 当用户信息不全或用户名已经存在的时候，都会抛出异常

\* \*/

public void createUser(MvnUser user) {

// 验证基本的用户信息

if (user == null || user.getUrUserName() == null || user.getUrPassword() == null) {

throw new RuntimeException("用户信息不合法");

}

// 根据用户名查询用户对象

MvnUser u = userDAO.findUserByUserName(user.getUrUserName());

// 如果能查询到用户对象，说明用户已经存在，抛异常

if (u != null) {

throw new RuntimeException(user.getUrUserName() + " 用户已存在");

}

// 调用dao代码，添加一个新用户

userDAO.addUser(user);

}

/\*\*

\* 更新id对应用户的年龄和状态信息

\*

\* @param age 要更新用户的新年龄

\* @param status 要更新用户的新状态

\* @param id 要更新用户的id，这是更新的条件

\*

\* @return void

\* \*/

public void editUser(int age, String status, int id) {

MvnUser user = this.userDAO.findUserById(id);

user.setUrAge(age);

user.setUrStatus(status);

this.userDAO.update(user);

}

public void deleteUser(int id) {

MvnUser user = this.userDAO.findUserById(id);

this.userDAO.deleteUser(user);

}

public MvnUser searchUser(int id) {

MvnUser user = this.userDAO.findUserById(id);

return user;

}

public MvnUser searchUser(String userName) {

MvnUser user = this.userDAO.findUserByUserName(userName);

return user;

}

public List<MvnUser> searchUsers() {

List<MvnUser> userList = this.userDAO.findUsers();

return userList;

}

}

AddUserServlet.java  
（随书代码  
[MvnBookTP04\MvnBookTP04\src\main\java\cn\com\mvnbook\demo\tp04\servlet\AddUserServlet.java](代码/MvnBookTP04/src/main/java/cn/com/mvnbook/demo/tp04/servlet/AddUserServlet.java)）

package cn.com.mvnbook.demo.tp04.servlet;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import cn.com.mvnbook.demo.tp04.entity.MvnUser;

import cn.com.mvnbook.demo.tp04.service.UserService;

/\*\*

\* Servlet implementation class AddUserServlet

\*/

public class AddUserServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

/\*\*

\* @see HttpServlet#service(HttpServletRequest request, HttpServletResponse

\* response)

\*/

protected void service(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// 获取要添加用户的信息

String userName = request.getParameter("userName");

String password = request.getParameter("password");

String ageStr = request.getParameter("age");

String status = request.getParameter("status");

int age = 0;

try {

// 将年龄字符串，转变成数字（数据库中需要数字类型）

age = Integer.parseInt(ageStr);

} catch (Exception e) {

}

// 封装成MvnUser对象

MvnUser user = new MvnUser();

user.setUrAge(age);

user.setUrPassword(password);

user.setUrStatus(status);

user.setUrUserName(userName);

UserService userService = new UserService();

String msg = "添加成功";

try {

// 调用service，创建用户

userService.createUser(user);

} catch (Exception e) {

e.printStackTrace();

msg = "添加失败：" + e.getMessage();

}

// 返回添加后的结果提示信息

response.setContentType("text/html;charset=UTF-8");

PrintWriter out = response.getWriter();

out.print(msg);

}

}

DeleteUserServlet.java  
（随书代码  
[MvnBookTP04\MvnBookTP04\src\main\java\cn\com\mvnbook\demo\tp04\servlet\DeleteUserServlet.java](代码/MvnBookTP04/src/main/java/cn/com/mvnbook/demo/tp04/servlet/DeleteUserServlet.java)）

package cn.com.mvnbook.demo.tp04.servlet;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import cn.com.mvnbook.demo.tp04.service.UserService;

/\*\*

\* Servlet implementation class DeleteUserServlet

\*/

public class DeleteUserServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

/\*\*

\* @see HttpServlet#service(HttpServletRequest request, HttpServletResponse

\* response)

\*/

protected void service(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

String idStr = request.getParameter("id");

int id = 0;

try {

id = Integer.parseInt(idStr);

} catch (Exception e) {

}

UserService userService = new UserService();

String msg = "删除成功";

try {

userService.deleteUser(id);

} catch (Exception e) {

e.printStackTrace();

msg = "删除失败:" + e.getMessage();

}

response.setContentType("text/html;charset=UTF-8");

PrintWriter out = response.getWriter();

out.print(msg);

out.flush();

out.close();

}

}

EditUserServlet.java  
（随书代码  
[MvnBookTP04\MvnBookTP04\src\main\java\cn\com\mvnbook\demo\tp04\servlet\EditUserServlet.java](代码/MvnBookTP04/src/main/java/cn/com/mvnbook/demo/tp04/servlet/EditUserServlet.java)）

package cn.com.mvnbook.demo.tp04.servlet;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import cn.com.mvnbook.demo.tp04.service.UserService;

/\*\*

\* Servlet implementation class EditUserServlet

\*/

public class EditUserServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

String idStr = request.getParameter("id");

String status = request.getParameter("status");

String ageStr = request.getParameter("age");

int id=0,age=0;

try{

id = Integer.parseInt(idStr);

}catch(Exception e){}

try{

age = Integer.parseInt(ageStr);

}catch(Exception e){}

UserService userService = new UserService();

String msg = "修改成功";

try{

userService.editUser(age, status, id);

}catch(Exception e){

e.printStackTrace();

msg = "修改失败:"+e.getMessage();

}

response.setContentType("text/html;charset=UTF-8");

PrintWriter out = response.getWriter();

out.print(msg);

out.flush();

out.close();

}

}

SearchUserServlet.java  
（随书代码  
[MvnBookTP04\MvnBookTP04\src\main\java\cn\com\mvnbook\demo\tp04\servlet\SearchUserServlet.java](代码/MvnBookTP04/src/main/java/cn/com/mvnbook/demo/tp04/servlet/SearchUserServlet.java)）

package cn.com.mvnbook.demo.tp04.servlet;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import cn.com.mvnbook.demo.tp04.entity.MvnUser;

import cn.com.mvnbook.demo.tp04.service.UserService;

import net.sf.json.JSONObject;

/\*\*

\* Servlet implementation class SearchUserServlet

\*/

public class SearchUserServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

protected void service(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// 获取查询的方式（根据id查询还是根据用户名查询）

String type = request.getParameter("type");

UserService userService = new UserService();

MvnUser user = null;

// 根据id查询

if ("byId".equals(type)) {

// 获取id

String idStr = request.getParameter("id");

int id = 0;

try {

id = Integer.parseInt(idStr);

} catch (Exception e) {

}

user = userService.searchUser(id);

} else {

// 根据用户名查询

String userName = request.getParameter("userName");

user = userService.searchUser(userName);

}

// 设置返回的响应为json响应

response.setContentType("text/json;charset=UTF-8");

PrintWriter out = response.getWriter();

// 将查询的用户对象，转变成json格式的字符串，写入响应返回

out.print(JSONObject.fromObject(user));

out.flush();

out.close();

}

}

SearchUsersServlet.java  
（随书代码  
[MvnBookTP04\MvnBookTP04\src\main\java\cn\com\mvnbook\demo\tp04\servlet\addUsersServlet.java](代码/MvnBookTP04/src/main/java/cn/com/mvnbook/demo/tp04/servlet/SearchUsersServlet.java)）

package cn.com.mvnbook.demo.tp04.servlet;

import java.io.IOException;

import java.util.List;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import cn.com.mvnbook.demo.tp04.entity.MvnUser;

import cn.com.mvnbook.demo.tp04.service.UserService;

/\*\*

\* Servlet implementation class SearchUsersServlet

\*/

public class SearchUsersServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

/\*\*

\* @see HttpServlet#service(HttpServletRequest request, HttpServletResponse response)

\*/

protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

UserService userService = new UserService();

List<MvnUser> userList = userService.searchUsers();

// 将查询出的用户集合，写于request属性中

request.setAttribute("userList", userList);

// 转向 userList.jsp页面

request.getRequestDispatcher("/userList.jsp").forward(request, response);

}

}

userList.jsp  
（随书代码  
[MvnBookTP04\MvnBookTP04\src\main\webapp\userList.jsp](代码/MvnBookTP04/src/main/webapp/userList.jsp)）

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<table width="100%" border="1">

<tr>

<td width="51" bgcolor="#CCCCCC"><div align="center"><strong><span class="STYLE2">选择</span></strong></div></td>

<td width="111" bgcolor="#CCCCCC"><div align="center"><strong>序号</strong></div></td>

<td width="137" bgcolor="#CCCCCC"><div align="center"><strong>用户名</strong></div></td>

<td width="105" bgcolor="#CCCCCC"><div align="center"><strong>年龄</strong></div></td>

<td width="101" bgcolor="#CCCCCC"><div align="center"><strong>状态</strong></div></td>

</tr>

<c:forEach items="${userList }" var="\_user" varStatus="status">

<c:choose>

<c:when test="${status.index%2==0 }">

<tr bgcolor="#FFFFFF">

</c:when>

<c:otherwise>

<tr bgcolor="#99FFFF">

</c:otherwise>

</c:choose>

<td><input type="checkbox" name="checkbox" value="${\_user.urId }"></td>

<td>${status.index+1 }</td>

<td>${\_user.urUserName }</td>

<td>${\_user.urAge }</td>

<td>${\_user.urStatus }</td>

</tr>

</c:forEach>

</table>

Index.jsp  
（随书代码  
[MvnBookTP04\MvnBookTP04\src\main\webapp\index.jsp](代码/MvnBookTP04/src/main/webapp/index.jsp)）

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">

<title>测试用户CRUD操作</title>

<script type="text/javascript" src="js/jquery-1.8.2.min.js"></script>

<script type="text/javascript">

$(document).ready(function() {

// 页面加载完后，自动发searchUsersServlet请求，加载到userListDiv中显示

$("#userListDiv").load("searchUsersServlet");

});

// 新增 按钮事件触发函数

function toAdd() {

// 获取addForm中的请求信息

var \_data = $("#addForm").serialize();

//alert(\_data);

// 发添加新用户的Ajax请求

$.ajax({

type : 'post',

url : 'addUserServlet',

data : \_data,

success : function(msg) {

alert(msg);

// 更新最新的用户列表信息

$("#userListDiv").load("searchUsersServlet");

}

});

}

function toEdit() {

var \_data = $("#editForm").serialize();

alert(\_data);

$.ajax({

type : 'post',

url : 'editUserServlet',

data : \_data,

success : function(msg) {

alert(msg);

$("#userListDiv").load("searchUsersServlet");

}

});

}

function toDelete() {

var chks = $("input[name='checkbox']:checked");

if (chks.length == 0) {

alert("请选择要删除的用户");

} else if (chks.length > 1) {

alert("一次只能删除一个用户");

} else {

var to = confirm("您确定要删除选中的用户?");

if (to) {

var \_data = "id=" + chks.val();

$.ajax({

type : 'post',

data : \_data,

url : 'deleteUserServlet',

success : function(msg) {

alert(msg);

$("#userListDiv").load("searchUsersServlet");

}

});

}

}

}

function toShowAdd() {

$("#LayerAdd").show(1000);

}

function toShowEdit() {

//alert($("input[name='checkbox']:checked").length);

var chks = $("input[name='checkbox']:checked");

if (chks.length == 0) {

alert("请选择要编辑的用户");

} else if (chks.length > 1) {

alert("一次只能修改一个用户");

} else {

var \_data = "id=" + chks.val();

$.ajax({

type : 'post',

data : \_data,

url : 'searchUserServlet?type=byId',

dataType : 'json',

success : function(msg) {

$("#editForm #id").val(msg.urId);

$("#editForm #userName").val(msg.urUserName);

$("#editForm #age").val(msg.urAge);

$("#editForm #status").val(msg.urStatus);

//alert($("#editForm #age").val());

$("#LayerEdit").show(1000);

}

});

}

}

function toCloseAdd() {

$("#LayerAdd").hide(1000);

}

function toCloseEdit() {

$("#LayerEdit").hide(1000);

}

</script>

<style type="text/css">

<!--

.STYLE2 {

color: #000000

}

#LayerAdd {

position: absolute;

left: 113px;

top: 183px;

width: 434px;

height: 193px;

z-index: 1;

background-color: #99FFFF;

display: none;

}

#LayerEdit {

position: absolute;

left: 113px;

top: 183px;

width: 434px;

height: 193px;

z-index: 1;

background-color: #99FFFF;

display: none;

}

-->

</style>

</head>

<body>

<div id="LayerAdd">

<form name="addForm" name="addForm" id="addForm" method="post"

action="">

<table width="98%" border="0" align="center" cellpadding="0"

cellspacing="0">

<tr>

<td colspan="2" align="center"><strong><BR>添加新用户<br></strong></td>

</tr>

<tr>

<td width="47%" align="right">用户名：</td>

<td width="53%"><input name="userName" type="text"

id="userName"></td>

</tr>

<tr>

<td align="right">密码：</td>

<td><input name="password" type="password" id="password"></td>

</tr>

<tr>

<td align="right">年龄：</td>

<td><input name="age" type="text" id="age"></td>

</tr>

<tr>

<td colspan="2">&nbsp;</td>

</tr>

<tr>

<td colspan="2" align="center"><input type="button"

name="Submit4" value="添加" onclick="toAdd()"> <input

type="button" name="Submit5" value="关闭" onclick="toCloseAdd()"></td>

</tr>

</table>

</form>

</div>

<div id="LayerEdit">

<form name="editForm" id="editForm" method="post" action="">

<input type="hidden" name="id" id="id" />

<table width="98%" border="0" align="center" cellpadding="0"

cellspacing="0">

<tr>

<td colspan="2" align="center"><strong><br>

修改用户信息<br> </strong></td>

</tr>

<tr>

<td width="47%" align="right">用户名：</td>

<td width="53%"><input name="userName" type="text"

id="userName" readonly="readonly"></td>

</tr>

<tr>

<td align="right">年龄：</td>

<td><input name="age" type="text" id="age"></td>

</tr>

<tr>

<td align="right">状态：</td>

<td><select name="status" id="status">

<option value="Active">Active</option>

<option value="Inactive">Inactive</option>

<option value="Locked">Locked</option>

<option value="Deleted">Deleted</option>

</select></td>

</tr>

<tr>

<td colspan="2">&nbsp;</td>

</tr>

<tr>

<td colspan="2" align="center"><input type="button"

name="Submit4" value="修改" onclick="toEdit()"> <input

type="button" name="Submit5" value="关闭" onclick="toCloseEdit()"></td>

</tr>

</table>

</form>

</div>

<p>&nbsp;</p>

<p>测试用户CRUD页面</p>

<table width="539" border="1">

<tr>

<td colspan="5" align="right"><input type="button" name="Submit"

value="新增" onclick="toShowAdd()"> <input type="submit"

name="Submit2" value="修改" onclick="toShowEdit()"> <input

type="button" name="Submit3" value="删除" onclick="toDelete()"></td>

</tr>

<tr>

<td>

<div id="userListDiv"></div>

</td>

</tr>

</table>

<p>&nbsp;</p>

</body>

</html>

db.properties  
（随书代码  
[MvnBookTP04\MvnBookTP04\src\main\resources\db.properties](代码/MvnBookTP04/src/main/resources/db.properties)）

userName=root

password=root

url=jdbc:mysql://localhost:3306/mvn\_db

driverName=com.mysql.jdbc.Driver

### 5.3.4 构建项目

代码写好了，构建好像没什么特别的呢，基于Eclipse，就是点击Maven builder|test|install等等几个命令菜单而已。关于Web应用特别点的，就是在pom.xml中，添加发布web应用和同web服务器相关的插件，其实这点在前面的简易web案例中有提到，我们这里就直接贴出当前Web应用用到的插件配置。如下：  
（随书代码  
[MvnBookTP04\MvnBookTP04\pom.xml](代码/MvnBookTP04/pom.xml)）

<build>

<plugins>

<plugin>

<groupId>org.mortbay.jetty</groupId>

<artifactId>maven-jetty-plugin</artifactId>

<version>6.1.26</version>

<configuration>

<webAppSourceDirectory>${basedir}/src/main/webapp</webAppSourceDirectory>

</configuration>

</plugin>

<plugin>

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

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

<version>2.0.2</version>

<configuration>

<source>1.5</source>

<target>1.5</target>

</configuration>

</plugin>

</plugins>

</build>

### 5.3.5 测试

在工程上右键 Run As->Maven build...，在Goals后面输入jetty:run目标，运行jetty服务器。在浏览器中输入 http://localhost:8080/MvnBookTP04/index.jsp，运行出来的界面如下图5-12：



图5-12 CRUD首页

后面的操作界面就不截屏了，相信大家都可以点出来，没啥子业务逻辑啦 ^\_^

## 5.4 小结

通过对本章案例的研究，其实是用Maven+Eclipse开发Web应用也蛮简单的，无非刚开始的时候生疏点，有点点新注意安装或配置的地方，以后都很轻松了。

主要是：

1. 在Eclipse中装好自己习惯的archetype构件，帮我们创建好web工程架构
2. 根据应用需要，在pom.xml中配置好需要的依赖
3. 配置好web服务器相关的插件
4. 注意web应用的约定目录结构，和每个目录中放什么样的文件

# 第六章 开发企业级Web应用

## 6.1 企业web应用简介

通过前面的研究，咱们算是能自己独立基于Maven+Eclipse开发JSP/Servlet的Web应用了，但是在企业中，还是会遇到同咱们自己学习中不一样的情况。比如需要使用框架开发；一个项目需要分成多个模块开发，最后合并集成；新项目重用以前搭建好的框架；开发的时候，要减少对外网的依赖（我们一起的开发，都必须时刻同外网连通的）等等。

接下来我们还是以用户CRUD为例子，分别使用Struts+Spring+Hibernate框架和SpringMVC+Spring+MyBatis框架进行实现。

其中包括：

1. Maven私服搭建和使用
2. Maven的聚合管理
3. Maven的继承
4. 以及SSH、SSM两大流行框架的搭建

## 6.2 搭建Maven私服

在我们以前的案例开发过程中，强烈要求同外网保存连接，为什么呢？

因为一旦我们需要的依赖，在本地环境中没有存在的化，maven会很聪明的，自动到网上的资源仓库中查找，并且下载（有些它自动下载最新版本，有的需要我们在pom.xml中指定坐标，maven根据坐标去查找下载）。

当我们是一个个体去开发的时候，这是允许的，但是如果我们是一个团队，特别是有历史的团队，开发过很多项目的团队。这样的团队，因为历史悠久，肯定在以前项目中用到了很多构件，这些构件也肯定下载过。如果每次下载都没有保存，或至少保存到每个程序员的本地电脑里面，那就意味着团队中的每个程序员都需要时刻连接外部资源仓库，下载自己的构件资源。全世界那么多程序员，想象一下资源仓库服务器的压力，是不是好大呢。另外网络稳定、速度也快还好说，如果不稳定，很慢，下载一个1M的构件，都要十几分钟，请问程序员怎么开发?太不顺畅了吧。

基于这种情况，我们需要在公司里面搭建一个服务器，程序员需要构件的时候，先看本地有没有，没有，找公司服务器要，公司服务器再没有，再从外网下载。下载后，在公司服务器保存，再自动在程序员本地电脑保存。这样就可以在公司内网，重复使用下载下来的构件，从而减少对外网的依赖。

另外因为是内网，肯定要比外网要稳定快速得多吧。如果贵公司内网比外网还要慢，那麻烦请我这菜鸟去帮你吧 ^\_^

目前常用的搭建Maven私服的服务器有三个：

Apache基金会的Archiva、JFrog的Artifactory和Sonatype的Nexus。作为咱们这刚入门的求学者，就不要太考虑它们有啥子区别，琢磨着选哪个最好了。直接是能用，开源免费就OK，所以接下来咱们一起研究Apache基金会的Archive服务器的搭建。

1. 下载Archiva

下载连接是：<http://archiva.apache.org/download.cgi>

目前最新版本是2.2.1的，有三种内容下载:一个是tar.gz,一个是war包，还有一个是source源代码。

我这下载的是tar.gz包：apache-archiva-2.2.1-bin.tar.gz。

该版本里面包含自己的web服务器，直接解压，独立web服务器启动。

1. 解压Archiva，启动服务器

用压缩工具解压压缩文件，目录结果如下图6-1：

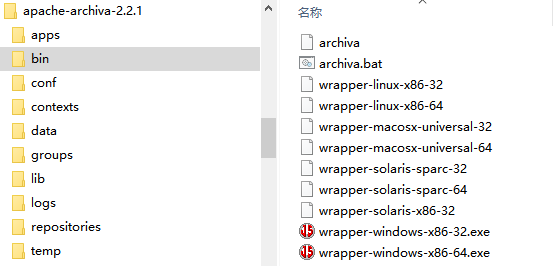


图6-1 Archiva目录结构

bin目录下有个archiva.bat文件，该文件是在win操作系统下的服务启动程序。

安装配置好JDK（1.7以上）

打开cmd窗口，切换到archiva.bat所在的目录，输入archiva console命令回车，启动Archiva服务器。如图6-2：

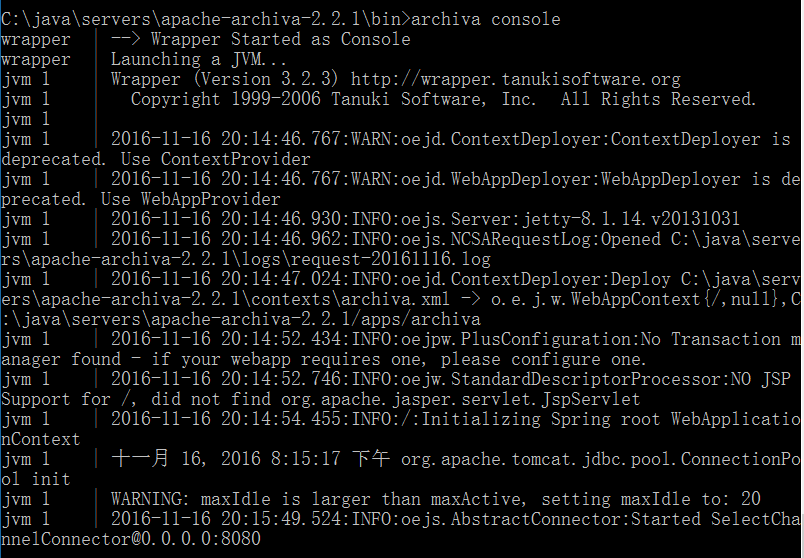


图6-2 archiva启动提示

最后显示当前的Web请求端口是8080

1. 初始化和配置Archiva服务器

1）初始化管理员用户

在浏览器中输入 http://localhost:8080，打开页面如下图6-3：

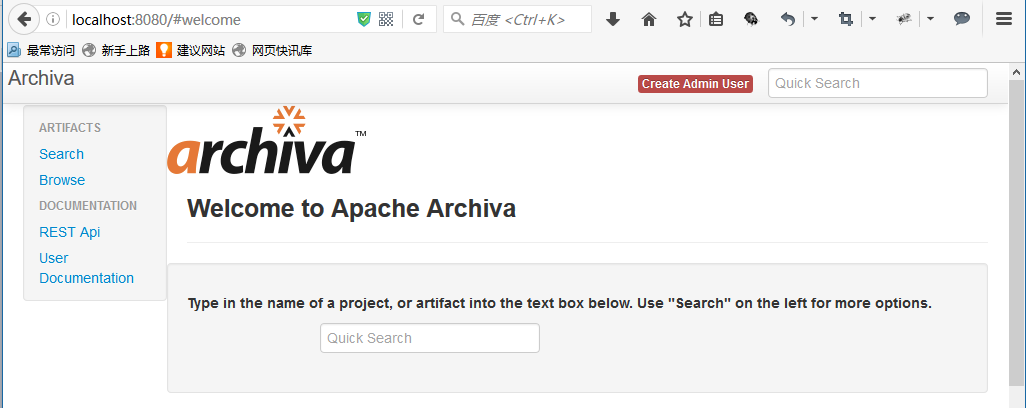


图6-3 Archiva首页

点击右上角的 Create Admin User的红色按钮，在显示的页面上输入管理员的用户名和密码，点保存创建。如图6-4：

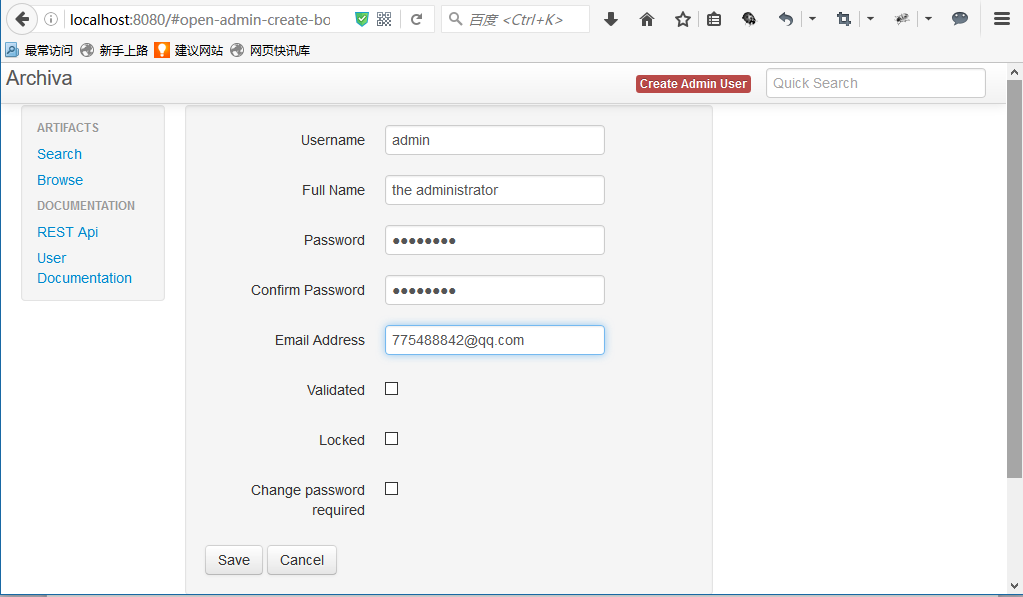


图6-4 创建Admin用户

2）添加其它用户

选择页面左边的Manage，点击Add，在输入框中输入要添加的用户信息，如图6-5：

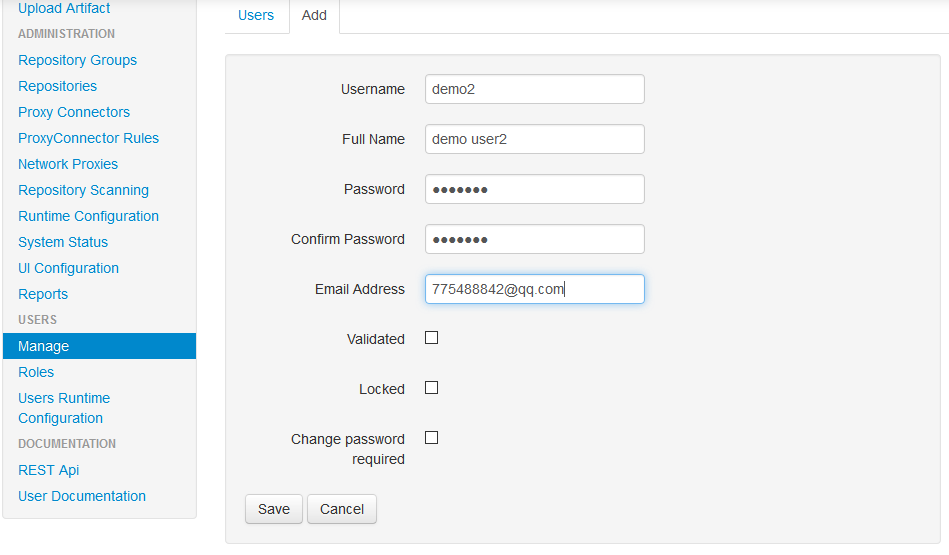


图6-5 添加普通用户

点击 Save按钮，创建一个新的用户。

根据项目团队的需要，可以给每个开发人员创建访问私服的用户名和密码。

3）配置本地仓库位置

选择页面左边 Repositories菜单，页面会显示本地仓库和远程仓库的配置，如图6-6：

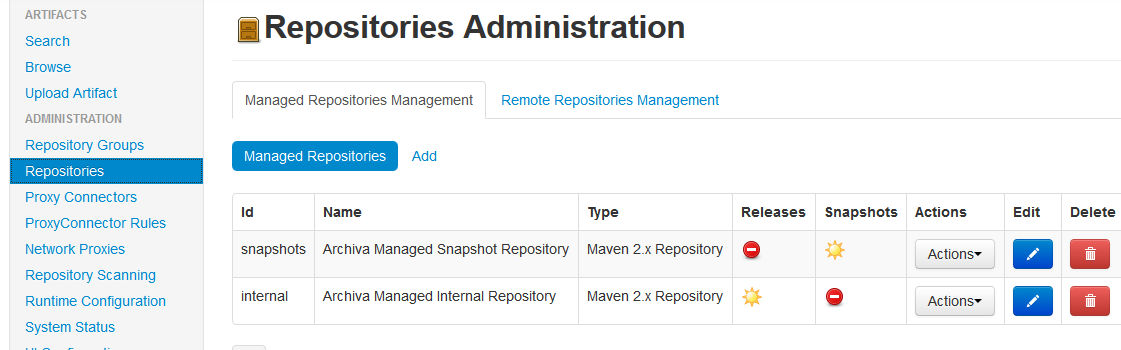


图6-6 管理仓库

点击图上的Add，我们可以添加一个本地仓库位置，具体填写的信息，参考已有的配置就行，当然，咱们也可以修改现在有的，比如id为internal的本地仓库，点击Edit图标，会显示如下界面图6-7：

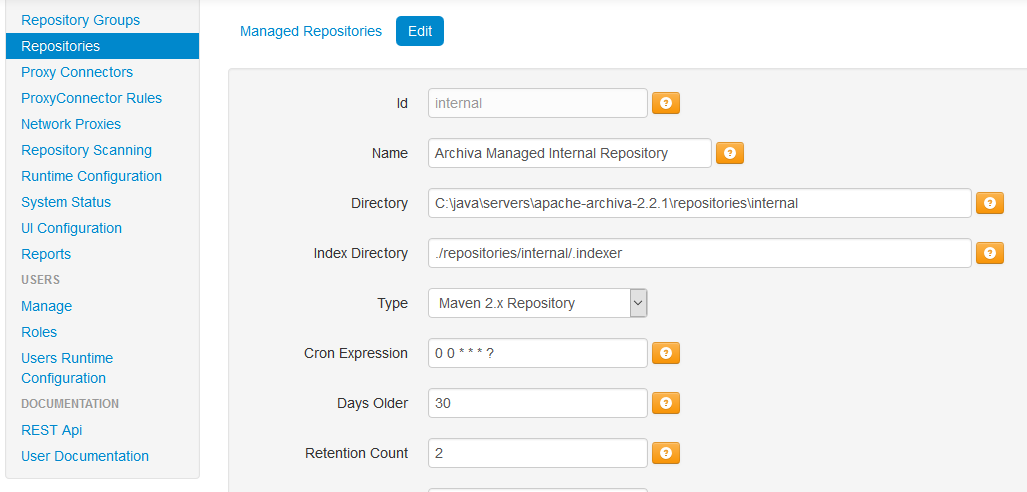


图6-7 修改本地仓库

其它都没什么太重要，主要是id和Directory。

Id在开发人员客户端进行配置的时候，需要使用

Directory是仓库保存构件的路径。

1. 添加远程仓库

Archiva装好后，有个默认的远程仓库，连接是：  
<https://repo.maven.apache.org/maven2>

我们当然可以去发现其它的远程仓库，配置到咱们的私服里面来。需要的时候，Archiva自动会从这些私服中找需要的构件。

比如我们现在配置一个阿里云的远程仓库：

点击当前页面中，Remote Repositories Management中Add按钮，输入阿里云仓库的信息如图6-8：

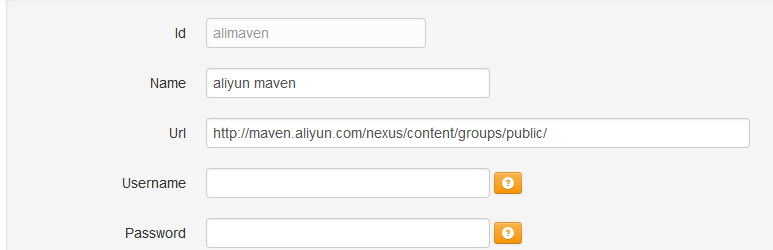


图6-8 添加远程仓库

后面当然还有很几个输入框，这里就截取了关键信息。

Id随便输入，输入后不能再修改的，需要唯一

Name是名称，随意

Url是远程仓库的连接，这就不能随意了 哈哈

Usernam和password是连接url远程仓库的用户名和密码，有些仓库提供公共的共享访问，比如我们现在配置的，它就不需要输入，否则就得获得用户名和密码许可才行。

点击 Save后，出现如下界面图6-9，表示OK

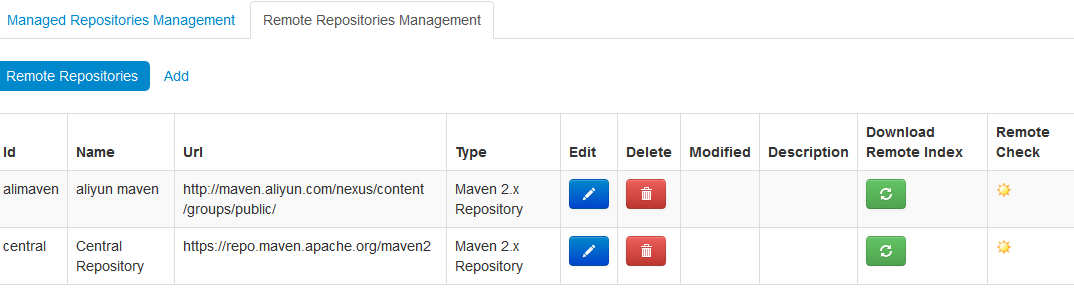


图6-9 仓库显示页面

其中就有我们刚刚添加上去的alimaven。

1. 在开发员端配置对私服的使用

在本地用户目录下的.m2目录下，找到settings.xml,我电脑上目录是 C:\Users\Noble\.m2\settings.xml。如果您电脑是第一次使用，可能没有settings.xml文件，不过没什么，还记得apache-maven-3.3.9的安装目录不，里面有个conf/settings.xml，把这文件copy到用户的.m2目录下就行。  
（随书代码[settings.xml](代码/settings.xml)）

解析来咱们就按步骤，在settings.xml中完成开发员电脑同私服连接的配置。

第一步：配置同服务器的认证信息

根据前面对私服的搭建，咱们应该可以得出结论，私服就是web服务器，里面提供了构件资源，我们可以通过web访问下载下来。既然要连接web服务器访问，而且我们前面好像有对web服务器设置用户吧 ^\_^，所以第一步，是不是应该在我们的本地，配置能访问Web服务器的认证信息（用户名和密码）呢？！

对，就是配置认证信息。

在settings.xml文件中，找到servers标签，在里面添加一个server的认证信息配置，格式如下：

<server>

<id>archivaServer</id>

<username>admin</username>

<password>admin123</password>

</server>

说明：

Id是要认证的服务器名称，因为可以配置多个，它是用来标记服务器的，要唯一

Username和password不用说了吧，就是所谓的用户名和密码，我这里直接是初始化Archiva私服服务器的时候，创建的admin用户和密码。

我们如果要连接多个私服，可以类似的配置多个server，每个server是一个私服的认证信息

第二步：配置要连接的私服服务器信息

前面我们在自己的settings.xml中，配置了连接私服服务器的认证信息，那这认证信息对应哪个私服服务器呢？接下来咱们在settings.xml中配置私服服务器信息。

很简单，直接在settings.xml中找到mirrors标签，在该标签中插入如下内容：

<mirror>

<id>archivaServer</id>

<mirrorOf>\*</mirrorOf>

<name>MyOwnRepo2</name>

<url>http://localhost:8080/repository/internal</url>

</mirror>

说明：

Id是私服映射的标记，该标记要同server中id一样。如果连接私服需要认证信息的化，Maven同过id找对应的server，用server的认证信息进行认证。

mirrorOf 指定哪些内容需要通过私服下载，\*表示所有构件都需要从私服下载

Name是私服的名称，随意取，方便自己的记忆和理解

Url，指定私服的url，注意格式:http://<私服ip/名称>:<web端口>/repository/<仓库id>

就这么两步辛苦，当咱们需要依赖的时候，就直接可以从搭建的私服中获取了。当然，不能保证私服有现成的，不过不要紧，私服自己会去网络中找到对应的依赖给我们，同时它也会在自己的仓库中有备份保存，以备咱们其它开发人员需要。

第三步：配置本地工程的发布

前面我们可以将私服上的依赖下载到我们本地，进行项目开发了，那我本地的模块开发好了，怎么样把它们打包，以构件的形式，发布到私服上去，让同项目组的组员，能以添加其它构件一样的，添加咱们的构件，进一步开发其他项目或模块的功能呢？

这一步就是解决本地项目打包，在私服上发布成构件的问题。

要完成这样的功能，需要做两个事情：编写配置文件和运行发布命令。

1. 编写配置文件

前面两步，咱们都是在[settings.xml](代码/settings.xml)中进行的配置，这步的配置文件不同了，需要在咱们工程的pom.xml文件中，在project标签内，添加distributionManagement配置，指定要发布的目标地（私服）,具体内容如下：

<distributionManagement>

<repository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

</distributionManagement>

说明：

上面的信息，配置了两个仓库url，一个是repository，一个是快照repository。每个respository中，都有一个url。不用过多罗嗦，url就是咱们要发布的私服仓库url，同我们在settings.xml中配置mirror中的url一样。同样，有两个id，这两个id就需要注意了，要同我们在settings.xml中，用server配置的验证信息中的id对应。因为咱们发布无非就是上传文件，上传文件前，需要安全认证的，maven就是通过id，将server中的验证信息发送给私服，私服认证通过了，才允许咱们将本地构件发布上去的。

1. 运行发布命令

前面咱们将配置信息都配置好了，怎么运行命令，按配置信息将工程当构件发布到私服上去呢？简单着呢：

在工程上点击右键 Run As-Maven build...，在弹出框的 Goal后面，输入deploy，点击Run就行，它们就会自动发布到咱们的私服上去。

## 6.3 实现Struts2+Spring+Hibernate框架应用

前面我们研究了基于Archiva的私服搭建，现在我们全项目组就可以在私服下共用Maven开发环境了。接下来我们在Maven环境下，基于Struts2+Spring4.2+Hibernate4.1框架，体验Web应用的开发过程。

为了体现大型项目的模块开发，已经公司中项目模块的解耦和重用，展现出我们Maven开发的优势，我们将按如下步骤进行：

1. 创建三个POM工程，定义好Hibernate、Spring和Struts的基本依赖
2. 创建Service和DAO层的接口模块
3. 创建Service和DAO的实现模块
4. 创建基于Struts的Web模块
5. 整合前面的所有模块，形成一个完整的SSH项目
6. 完善相关的文档插件的配置，进行安装和测试

### 6.3.1 创建公共POM

1. 创建Hibernate的公共POM模块

基于Eclipse的maven-archetype-quickstart创建一个Maven工程（同前面创建基本的Maven工程一样），因为是POM公共模块，我们这里不需要写代码，只需要将Hibernate和相关的依赖配置在pom.xml中，并且在pom.xml中，将packaging方式设置成pom，表示是一个公共的父pom，代码如下：  
（随书代码[Hibernate4MySQLPOM\pom.xml](代码/Hibernate4MySQLPOM/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>Hibernate4MySQLPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

**<packaging>pom</packaging>**

<name>Hibernate4MySQLPOM</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<!-- 3.6.5.Final,3.3.2.GA -->

<project.build.hibernate.version>4.1.0.Final</project.build.hibernate.version>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.7</version>

<scope>test</scope>

</dependency>

<!-- hibernate -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>${project.build.hibernate.version}</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-ehcache</artifactId>

<version>${project.build.hibernate.version}</version>

</dependency>

<dependency>

<groupId>org.hibernate.javax.persistence</groupId>

<artifactId>hibernate-jpa-2.0-api</artifactId>

<version>1.0.0.Final</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.34</version>

</dependency>

</dependencies>

<distributionManagement>

<repository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

</distributionManagement>

</project>

注意pom.xml中的粗体部分 <packaging>pom</packaging>，表示当前的pom是一个独立的pom父模块，可以独立安装到仓库中，可以被其它工程继承使用。

同时注意最后的distributionManagement配置，该配置就可以让咱们的工程以构件的形式发布到指定的私服上去。配置好了怎么发布，请参考前面一节的最后说明。后面每个pom.xml都有一样的配置，作用是一样的！

在工程上点击右键 Run As->Maven install,我就可以把当前pom安装到前面搭建好Archiva私服里面去，安装后，我们可以在Archiva管理界面的Browse导航页中，看到如下图6-10显示的Hibernate4MySQLPOM POM构件。

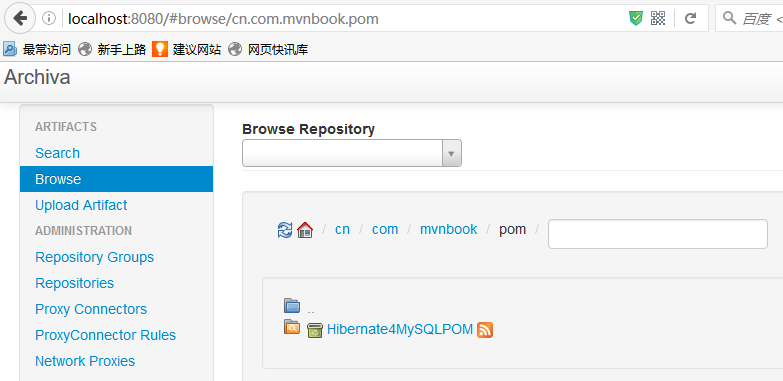


图6-10 浏览Archiva中的构件

1. 创建Spring的公共POM模块

同前面Hibernate的POM创建一样，我们可以创建基于Spring的POM公共构件模块，具体的工程的创建就不演示了，直接贴我这样例里面的pom.xml。  
（随书代码[SpringPOM\pom.xml](代码/SpringPOM/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

**<packaging>pom</packaging>**

<name>SpringPOM</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<!-- 3.2.16.RELEASE,3.1.4.RELEASE -->

<project.build.spring.version>4.2.7.RELEASE</project.build.spring.version>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.7</version>

<scope>test</scope>

</dependency>

<!-- spring -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-aspects -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>5.0.0.Final</version>

</dependency>

</dependencies>

<distributionManagement>

<repository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

</distributionManagement>

</project>

同样注意粗体提示部分，完了 工程上鼠标右键 Run As ->Maven install，安装POM构件图6-11：

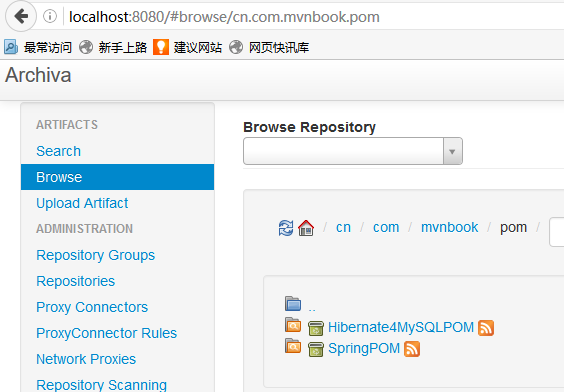


图6-11 显示发布后的构件

1. 创建Struts的公共POM模块

重复前面的故事，不在累赘了，直接贴pom.xml代码和安装pom后的管理界面。只是这里需要注意一下，pom.xml中，除了Struts的依赖之外，还有jsp/servlet的依赖和Struts同Spring集成的插件依赖。

pom.xml  
（随书代码[StrutsPOM\pom.xml](代码/StrutsPOM/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>StrutsPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

**<packaging>pom</packaging>**

<name>StrutsPOM</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<!-- jsp servlet -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>jsp-api</artifactId>

<version>2.1</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<!-- struts2 -->

<!-- https://mvnrepository.com/artifact/org.apache.struts/struts2-core -->

<dependency>

<groupId>org.apache.struts</groupId>

<artifactId>struts2-core</artifactId>

<version>2.3.16</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.struts/struts2-spring-plugin -->

<dependency>

<groupId>org.apache.struts</groupId>

<artifactId>struts2-spring-plugin</artifactId>

<version>2.3.4.1</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.7</version>

<scope>test</scope>

</dependency>

</dependencies>

<distributionManagement>

<repository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

</distributionManagement>

</project>

安装后的Archiva上的POM构件图6-13

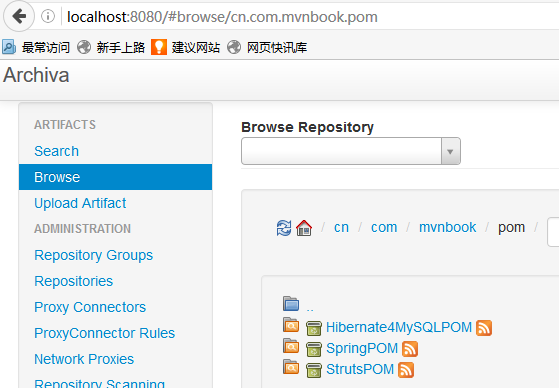


图6-13 StrutsPOM构件

### 6.3.2 实现Hibernate DAO模块

在实际项目中，我们一般会使用面向接口编程，从而实现调用者和被调用者的完全解耦，方便项目的团队开发和后期的扩展。鉴于这样的考虑，关于Hibernate的持久层的实现，分两步进行：第一步定义公共接口和类，第二步基于Hibernate实现接口，详细介绍如下：

第一步：定义公共DAO接口和类

1. 创建一个普通的Maven工程：MvnBookSSHDemo.DAO（随书代码[MvnBookSSHDemo.DAO](代码/MvnBookSSHDemo.DAO)）。目录结构如下图6-14：

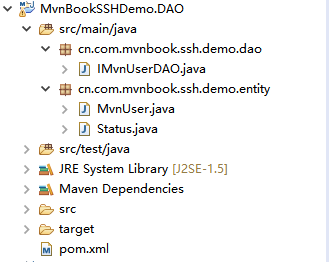


图6-14 DAO项目结构

pom.xml也没有太特别的，内容如下：  
（随书代码[MvnBookSSHDemo.DAO\pom.xml](代码/MvnBookSSHDemo.DAO/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.com.mvnbook.ssh.demo</groupId>

<artifactId>MvnBookSSHDemo.DAO</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>MvnBookSSHDemo.DAO</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.7</version>

<scope>test</scope>

</dependency>

</dependencies>

<distributionManagement>

<repository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

</distributionManagement>

</project>

1. 编写公共接口和代码

在这里有两类代码，一类是实体类（MvnUser），另一类是实体DAO接口(IMvnUserDAO)。因为MvnUser里面有个状态(status)属性，配合它，定义了一个枚举状态类（Status）。具体代码如下：

Status.java  
（随书代码  
[MvnBookSSHDemo.DAO\src\main\java\cn\com\mvnbook\ssh\demo\entity\Status.java](代码/MvnBookSSHDemo.DAO/src/main/java/cn/com/mvnbook/ssh/demo/entity/Status.java)）

package cn.com.mvnbook.ssh.demo.entity;

public enum Status {

ACTIVE("Active"),

INACTIVE("Inactive"),

DELETED("Deleted"),

LOCKED("Locked");

private String status;

private Status(final String status){

this.status = status;

}

public String getStatus() {

return status;

}

public void setStatus(String status) {

this.status = status;

}

public String toString(){

return this.status;

}

}

MvnUser.java  
（随书代码  
[MvnBookSSHDemo.DAO\src\main\java\cn\com\mvnbook\ssh\demo\entity\MvnUser.java](代码/MvnBookSSHDemo.DAO/src/main/java/cn/com/mvnbook/ssh/demo/entity/MvnUser.java)）

package cn.com.mvnbook.ssh.demo.entity;

public class MvnUser {

private int urId;

private String urUserName;

private String urPassword;

private int urAge;

private String urStatus = Status.ACTIVE.getStatus();//Active

public int getUrId() {

return urId;

}

public void setUrId(int urId) {

this.urId = urId;

}

public String getUrUserName() {

return urUserName;

}

public void setUrUserName(String urUserName) {

this.urUserName = urUserName;

}

public String getUrPassword() {

return urPassword;

}

public void setUrPassword(String urPassword) {

this.urPassword = urPassword;

}

public int getUrAge() {

return urAge;

}

public void setUrAge(int urAge) {

this.urAge = urAge;

}

public String getUrStatus() {

return urStatus;

}

public void setUrStatus(String urStatus) {

this.urStatus = urStatus;

}

}

IMvnUserDAO.java  
（随书代码  
[MvnBookSSHDemo.DAO\src\main\java\cn\com\mvnbook\ssh\demo\dao\IMvnUserDAO.java](代码/MvnBookSSHDemo.DAO/src/main/java/cn/com/mvnbook/ssh/demo/dao/IMvnUserDAO.java)）

package cn.com.mvnbook.ssh.demo.dao;

import java.util.List;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

/\*\*

\* MvnUser实体对象的持久层代码，封装了对MvnUser实体对象的CRUD方法

\*

\* @author Noble

\* @version 1.0

\* \*/

public interface IMvnUserDAO {

/\*\*

\* 在数据库中，添加一个新的MvnUser对象

\*

\* @param user 需要添加的用户实体对象，该对象需要有用户名、密码、年龄和状态属性

\*

\* @return void

\* @throws RuntimeException 添加失败或出现其它意外

\* \*/

public void addUser(MvnUser user) ;

/\*\*

\* 更新MvnUser对象。该对象中需要设置年龄、状态和id属性，属性和状态是要更新的新值，id为条件

\*

\* @param user 需要更新的MvnUser对象

\*

\* @return void

\* @throws RuntimeException 更新失败或出现其它意外

\* \*/

public void update(MvnUser user) ;

/\*\*

\* 删除MvnUser对象，该对象中需要有要删除对象的id属性，id属性为删除条件

\*

\* @param user 要删除的MvnUser对象

\*

\* @return void

\* @throws RuntimeException 删除失败或出现其它意外

\* \*/

public void deleteUser(MvnUser user) ;

/\*\*

\* 根据id查询对应的MvnUser对象

\*

\* @param id 要查询的MvnUser对象的id

\* @return MvnUser id对应的MvnUser对象，如果没有对象，返回null

\* @throws RuntimeException 出现意外情况

\* \*/

public MvnUser findUserById(int id) ;

/\*\*

\* 根据用户名查询对应的MvnUser对象

\*

\* @param userName 要查询的MvnUser对象的用户名

\* @return MvnUser 用户对应的MvnUser对象，如果没有对象，返回null

\* @throws RuntimeException 出现意外情况

\* \*/

public MvnUser findUserByUserName(String userName) ;

/\*\*

\* 查找数据库中所有的用户对象，以List集合的形式返回

\*

\* @return List<MvnUser> 所有用户对象的集合

\* @throws RuntimeException 出现意外情况

\* \*/

public List<MvnUser> findUsers() ;

}

1. 安装发布

在工程上鼠标右键 Run As->Maven install,Eclipse会自动将工程代码编译打包，如果没有错误的话，最后以构件的形式，安装在本地仓库中。参考效果图如下图6-15：

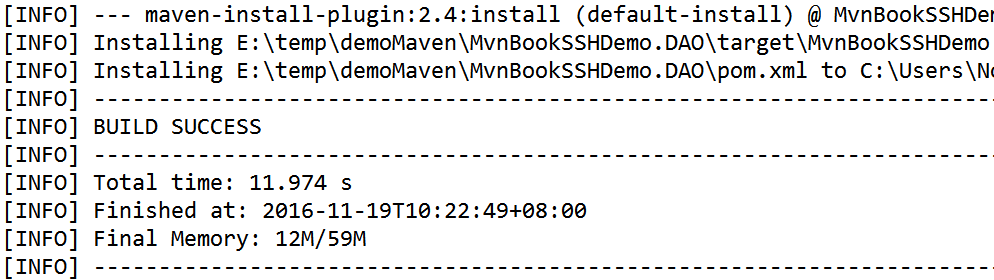


图6-15 安装构件提示

为了方便公司其它开发人员可以使用，我们接下来将该项目以构件的形式，发布到前面搭建好的私服上去，为了发布成功，请按前面的私服介绍，搭建并且启动私服，同时在当前工程的pom.xml中，添加distributionManagement配置，详细参考前面的pom.xml。具体操作和效果图如下。

1. 操作

工程右键选择Run As -> Maven build...

在Goals中输入deploy，点击Run按钮

1. Archiva上关于发布的构件效果图6-16

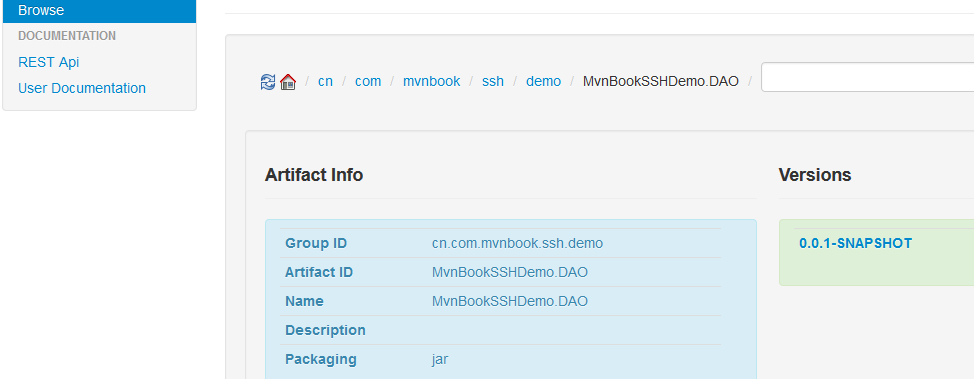


图6-16 DAO构件发布

第二步：基于Hibernate完成DAO接口的实现

团队商量确定好接口，接下来就是对接口的实现和基于接口上的开发工作了。因为有共同的接口，所以这两个工作可以同步进行。这现象同我们现实生活中的电脑的配件一样（硬盘、内存条、CPU、显卡等等），事先定义好标准（插口），不同厂商就可以按同样的标准各自生产，出来后就可以顺利的组装在一起，不用管有哪个厂家，在哪里，用那个流水线生产的。

接下来我们开始研究DAO接口的实现。大概分如下四步进行：

第一步：创建工程，添加相关依赖

这个步骤比较简单，创建工程的方式同以前一样。具体创建的过程，在这里就不重复了。目录结构如下图6-17：

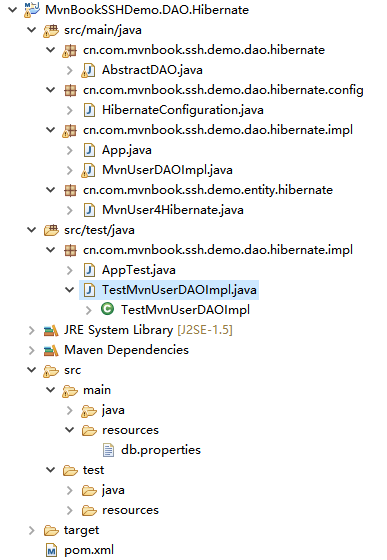


图6-17 Hibernate DAO项目结构

（随书代码[MvnBookSSHDemo.DAO.Hibernate](代码/MvnBookSSHDemo.DAO.Hibernate)）

虽然不重复说明项目的创建过程，但是要注意如下两点：

1. 因为前面我们有创建了公共的Hibernate POM工程，里面有描述好了Hibernate相关的依赖（目的是让所有开发人员重用，不要自己再重复编写），并且有以构件的形式安装发布好了。这里我们要体现的是怎么样继承前面定义好的pom。
2. 同样的，因为我们新工程里面要实现MvnBookSSHDemo.DAO中定义的接口，并且使用到它里面定义的公共类，而且根据前面的介绍，MvnBookSSHDemo.DAO，我们也以构件的形式安装发布到私服了。在这里，我们要体现一下怎么样在自己的工程里面，设置我们团队内部自己发布的构件。

这两点注意事项，主要体现在pom.xml中，pom.xml内容如下：  
（随书代码[MvnBookSSHDemo.DAO.Hibernate\pom.xml](代码/MvnBookSSHDemo.DAO.Hibernate/pom.xml)）

<?xml version="1.0"?>

<project

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"

xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<modelVersion>4.0.0</modelVersion>

**<parent>**

**<groupId>cn.com.mvnbook.pom</groupId>**

**<artifactId>Hibernate4MySQLPOM</artifactId>**

**<version>0.0.1-SNAPSHOT</version>**

**</parent>**

<groupId>cn.com.mvnbook.ssh.demo.dao.hibernate</groupId>

<artifactId>MvnBookSSHDemo.DAO.Hibernate</artifactId>

<name>MvnBookSSHDemo.DAO.Hibernate</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

**<dependency>**

**<groupId>cn.com.mvnbook.ssh.demo</groupId>**

**<artifactId>MvnBookSSHDemo.DAO</artifactId>**

**<version>0.0.1-SNAPSHOT</version>**

**</dependency>**

**<dependency>**

**<groupId>cn.com.mvnbook.pom</groupId>**

**<artifactId>SpringPOM</artifactId>**

**<version>0.0.1-SNAPSHOT</version>**

**<type>pom</type>**

**</dependency>**

</dependencies>

<distributionManagement>

<repository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

</distributionManagement>

</project>

说明：

1、 <parent>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>Hibernate4MySQLPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

这是pom.xml中的第一个粗体内容，它描述的是当前的pom.xml，继承了Hibernate4MySQLPOM构件中定义的pom内容，其中groupId,artifactId和version共同形成构件的坐标。当我们的pom需要继承别人定义好的pom的话，只需要使用如上parent配置指定就行，不过这里的继承，同java中继承一样，只能单继承，而且只能继承packaging类型为pom的构件（这点可以看Hibernate4MySQLPOM中的pom.xml文件，里面的packaging是pom）。

2、 <dependency>

<groupId>cn.com.mvnbook.ssh.demo</groupId>

<artifactId>MvnBookSSHDemo.DAO</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

**<type>pom</type>**

</dependency>

这是在我们pom.xml中第二个粗体内容，描述的是两个依赖。第一个依赖是我们前面定义的DAO接口和公共类的构件依赖，通过查看代码，其实同使用我们从网上找的其它依赖一样。第二个虽然也是使用的我们前面定义的spring的公共pom依赖，但是有点点不同，里面包含了一个<type>pom</type>，这个元素指定的是依赖的packaging类型。依赖的packaging类型默认是jar（我们前面所有pom.xml中没有指定type的情况，都是这样），如果我们pom引用的依赖是pom类型的，就需要在dependency中添加type元素，指定是类型pom，形同这里用到的第二个依赖，否则构建的时候会报错。

第二步：编写实现代码

基于Hibernate的DAO实现代码，主要有如下几个类：

MvnUser4Hibernate.java 该类继承MvnUser类，里面用注解描述了实体信息

AbstractDAO.java 该类定义了实体的公共持久化方法，所有的DAO实现类就继承它

MvnUserDAOImpl.java 该类实现MvnUser实体类的所有持久化方法

HibernateConfiguration.java Hibernate的配置类，描述Hibernate的配置信息，代替  
 hibernate.cfg.xml

db.properties 描述数据库连接信息和hibernate的一些配置信息

各个代码如下：

MvnUser4Hibernate.java  
（随书代码  
[MvnBookSSHDemo.DAO.Hibernate\src\main\java\cn\com\mvnbook\ssh\demo\entity\hibernate\MvnUser4Hibernate.java](代码/MvnBookSSHDemo.DAO.Hibernate/src/main/java/cn/com/mvnbook/ssh/demo/entity/hibernate/MvnUser4Hibernate.java)）

package cn.com.mvnbook.ssh.demo.entity.hibernate;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

@Entity

@Table(catalog="mvn\_db",name="mvn\_user")

public class MvnUser4Hibernate extends MvnUser {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

@Column(name="ur\_id")

public int getUrId() {

// TODO Auto-generated method stub

return super.getUrId();

}

@Column(name="ur\_user\_name",length=20,unique=true,nullable=false)

public String getUrUserName() {

// TODO Auto-generated method stub

return super.getUrUserName();

}

@Column(name="ur\_password",length=10,nullable=false)

public String getUrPassword() {

// TODO Auto-generated method stub

return super.getUrPassword();

}

@Column(name="ur\_age")

public int getUrAge() {

// TODO Auto-generated method stub

return super.getUrAge();

}

@Column(name="ur\_status",length=20,nullable=true)

public String getUrStatus() {

// TODO Auto-generated method stub

return super.getUrStatus();

}

}

AbstractDAO.java  
（随书代码  
[MvnBookSSHDemo.DAO.Hibernate\src\main\java\cn\com\mvnbook\ssh\demo\dao\hibernate\AbstractDAO.java](代码/MvnBookSSHDemo.DAO.Hibernate/src/main/java/cn/com/mvnbook/ssh/demo/dao/hibernate/AbstractDAO.java)）

package cn.com.mvnbook.ssh.demo.dao.hibernate;

import java.io.Serializable;

import java.lang.reflect.ParameterizedType;

import java.util.List;

import org.hibernate.Criteria;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.springframework.beans.factory.annotation.Autowired;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

import cn.com.mvnbook.ssh.demo.entity.hibernate.MvnUser4Hibernate;

public abstract class AbstractDAO<PK extends Serializable, T> {

private final Class<T> persistentClass;

@SuppressWarnings("unchecked")

public AbstractDAO(){

this.persistentClass =(Class<T>) (

(ParameterizedType) this.getClass().getGenericSuperclass()

).getActualTypeArguments()[1];

}

@Autowired

private SessionFactory sessionFactory;

protected Session getSession(){

return sessionFactory.getCurrentSession();

}

@SuppressWarnings("unchecked")

public T getByKey(PK key) {

return (T) getSession().get(persistentClass, key);

}

public void persist(T entity) {

getSession().persist(entity);

}

public void delete(T entity) {

getSession().delete(entity);

}

public void update(T entity){

getSession().merge(entity);

}

public List<T> findAll(){

Criteria cri = this.createEntityCriteria();

cri.setResultTransformer(Criteria.DISTINCT\_ROOT\_ENTITY);//消除重复对象

return cri.list();

}

protected Criteria createEntityCriteria(){

return getSession().createCriteria(persistentClass);

}

}

MvnUserDAOImpl.java  
（随书代码  
[MvnBookSSHDemo.DAO.Hibernate\src\main\java\cn\com\mvnbook\ssh\demo\dao\hibernate\impl\MvnUserDAOImpl.java](代码/MvnBookSSHDemo.DAO.Hibernate/src/main/java/cn/com/mvnbook/ssh/demo/dao/hibernate/impl/MvnUserDAOImpl.java)）

package cn.com.mvnbook.ssh.demo.dao.hibernate.impl;

import java.util.List;

import org.hibernate.Criteria;

import org.hibernate.criterion.Restrictions;

import org.springframework.stereotype.Repository;

import org.springframework.transaction.annotation.Transactional;

import cn.com.mvnbook.ssh.demo.dao.IMvnUserDAO;

import cn.com.mvnbook.ssh.demo.dao.hibernate.AbstractDAO;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

import cn.com.mvnbook.ssh.demo.entity.hibernate.MvnUser4Hibernate;

@Repository("userDAO")

@Transactional// **标记使用事务，为了方便DAO设置，是临时的，正式事务应该设置在服务层**

public class MvnUserDAOImpl extends AbstractDAO<Integer, MvnUser4Hibernate> implements IMvnUserDAO {

public void addUser(MvnUser user) {

MvnUser4Hibernate u = this.convertToMvnUser4Hibernate(user);

super.persist(u);

}

public void update(MvnUser user) {

MvnUser4Hibernate u = this.convertToMvnUser4Hibernate(user);

super.update(u);

}

public void deleteUser(MvnUser user) {

MvnUser4Hibernate u = this.convertToMvnUser4Hibernate(user);

super.delete(u);

}

public MvnUser findUserById(int id) {

return super.getByKey(id);

}

public MvnUser findUserByUserName(String userName) {

Criteria criteria = super.createEntityCriteria();

criteria.add(Restrictions.eq("urUserName", userName));

return (MvnUser)criteria.uniqueResult();

}

public List<MvnUser> findUsers() {

Criteria criteria = super.createEntityCriteria();

return criteria.list();

}

private MvnUser4Hibernate convertToMvnUser4Hibernate(MvnUser user){

MvnUser4Hibernate u = (MvnUser4Hibernate) this.findUserById(user.getUrId());

// 这里不要轻易new一个同已经存在的一样的对象，否则会抛

//org.hibernate.NonUniqueObjectException:

//a different object with the same identifier value was

//already associated withthe session异常

if (u == null) {

u = new MvnUser4Hibernate();

u.setUrId(user.getUrId());

}

u.setUrAge(user.getUrAge());

u.setUrPassword(user.getUrPassword());

u.setUrStatus(user.getUrStatus());

u.setUrUserName(user.getUrUserName());

return u;

}

}

HibernateConfiguration.java  
（随书代码  
[MvnBookSSHDemo.DAO.Hibernate\src\main\java\cn\com\mvnbook\ssh\demo\dao\hibernate\config\HibernateConfiguration.java](代码/MvnBookSSHDemo.DAO.Hibernate/src/main/java/cn/com/mvnbook/ssh/demo/dao/hibernate/config/HibernateConfiguration.java)）

package cn.com.mvnbook.ssh.demo.dao.hibernate.config;

import java.util.Properties;

import javax.sql.DataSource;

import org.hibernate.SessionFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.PropertySource;

import org.springframework.core.env.Environment;

import org.springframework.jdbc.datasource.DriverManagerDataSource;

import org.springframework.orm.hibernate4.HibernateTransactionManager;

import org.springframework.orm.hibernate4.LocalSessionFactoryBean;

import org.springframework.transaction.annotation.EnableTransactionManagement;

//配置类

@Configuration

@EnableTransactionManagement// 要支持事务管理

@ComponentScan({ "cn.com.mvnbook.ssh.demo.dao.hibernate.config" })

@PropertySource(value = { "classpath:db.properties" })// 自动读入的属性文件

public class HibernateConfiguration {

// 自动注入 Spring的环境对象(上下文)

@Autowired

private Environment environment;

// 创建一个SessionFactory

@Bean(name="sessionFactory")

public LocalSessionFactoryBean sessionFactory() {

LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();

// 数据源

sessionFactory.setDataSource(dataSource());

// 指定数据实体类的包

sessionFactory.setPackagesToScan(new String[]   
 { "cn.com.mvnbook.ssh.demo.entity.hibernate" });

// hibernate的属性信息

sessionFactory.setHibernateProperties(hibernateProperties());

return sessionFactory;

}

// 初始化数据源对象

@Bean(name="dataSource")// 将当前方法返回的对象，当成普通Bean对象，放入IOC容器中

public DataSource dataSource() {

DriverManagerDataSource dataSource = new DriverManagerDataSource();

// 设置连接数据库的四要素

dataSource.setDriverClassName(environment.getRequiredProperty("jdbc.driverClassName"));

dataSource.setUrl(environment.getRequiredProperty("jdbc.url"));

dataSource.setUsername(environment.getRequiredProperty("jdbc.username"));

dataSource.setPassword(environment.getRequiredProperty("jdbc.password"));

return dataSource;

}

// 将Hibernate除连接数据库之外的配置，封装到Properties

private Properties hibernateProperties() {

Properties properties = new Properties();

properties.put("hibernate.dialect", environment.getRequiredProperty("hibernate.dialect"));

properties.put("hibernate.show\_sql", environment.getRequiredProperty("hibernate.show\_sql"));

properties.put("hibernate.format\_sql", environment.getRequiredProperty("hibernate.format\_sql"));

properties.put("hibernate.hbm2ddl.auto", environment.getRequiredProperty("hibernate.hbm2ddl.auto"));

return properties;

}

@Bean

@Autowired

public HibernateTransactionManager transactionManager(SessionFactory s) {

HibernateTransactionManager txManager = new HibernateTransactionManager();

txManager.setSessionFactory(s);

return txManager;

}

}

db.properties(存放在工程的 src/main/resources目录下)  
（随书代码[MvnBookSSHDemo.DAO.Hibernate\src\main\resources\db.properties](代码/MvnBookSSHDemo.DAO.Hibernate/src/main/resources/db.properties)）

jdbc.username=root

jdbc.password=root

jdbc.url=jdbc:mysql://localhost:3306/mvn\_db

jdbc.driverClassName=com.mysql.jdbc.Driver

hibernate.dialect = org.hibernate.dialect.MySQLDialect

hibernate.show\_sql = true

hibernate.format\_sql = true

hibernate.hbm2ddl.auto = update

第三步：编写测试代码

测试代码是基于jUnit的，相对比较简单，只有一个类，针对MvnUserDAOImpl.java进行测试，另外有一个spring的配置文件，applicationContext.xml。

需要注意的是，测试的所有代码和资源文件，都是分别放在src/test目录下对应的子目录中的，参考前面提供的工程目录结构图。在Maven中具体文件的存放位置，是有固定约束的（规定死了的）。测试代码和配置文件的内容如下：

TestMvnUserDAOImpl.java  
（随书代码  
[MvnBookSSHDemo.DAO.Hibernate\src\test\java\cn\com\mvnbook\ssh\demo\dao\hibernate\impl\TestMvnUserDAOImpl.java](代码/MvnBookSSHDemo.DAO.Hibernate/src/test/java/cn/com/mvnbook/ssh/demo/dao/hibernate/impl/TestMvnUserDAOImpl.java)）

package cn.com.mvnbook.ssh.demo.dao.hibernate.impl;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import cn.com.mvnbook.ssh.demo.dao.IMvnUserDAO;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

import cn.com.mvnbook.ssh.demo.entity.Status;

import junit.framework.Assert;

public class TestMvnUserDAOImpl {

private IMvnUserDAO userDAO;

private ApplicationContext ctx = null;

@Before

public void init() {

ctx = new ClassPathXmlApplicationContext("applicationContext.xml");

userDAO = (IMvnUserDAO)this.ctx.getBean("userDAO");

//userDAO = (IMvnUserDAO) this.ctx.getBean(MvnUserDAOImpl.class);

}

@Test

public void testAddUser(){

MvnUser user= new MvnUser();

user.setUrAge(11);

user.setUrPassword("11");

user.setUrStatus(Status.ACTIVE.getStatus());

user.setUrUserName("userName11");

this.userDAO.addUser(user);

MvnUser u = this.userDAO.findUserByUserName("userName11");

Assert.assertTrue(u != null && u.getUrAge()==11);

this.userDAO.deleteUser(u);

}

@Test

public void testFindUserById(){

MvnUser user = this.userDAO.findUserById(1);

Assert.assertEquals("zhangsan", user.getUrUserName());

}

@Test

public void testUpdate(){

MvnUser user = this.userDAO.findUserById(1);

user.setUrAge(99);

this.userDAO.update(user);

user = this.userDAO.findUserById(1);

Assert.assertEquals(99, user.getUrAge());

}

@After

public void destory(){

this.userDAO = null;

this.ctx = null;

}

}

applicationContext.xml  
（随书代码  
[MvnBookSSHDemo.DAO.Hibernate\src\test\resources\applicationContext.xml](代码/MvnBookSSHDemo.DAO.Hibernate/src/test/resources/applicationContext.xml)）

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.0.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd

http://www.springframework.org/schema/context

[http://www.springframework.org/schema/context/spring-context-4.0.xsd">](http://www.springframework.org/schema/context/spring-context-4.0.xsd%22%3e)

<!--spring自动扫描组件类的包，包括子包和子子包等-->

<context:component-scan

base-package="cn.com.mvnbook.ssh.demo"></context:component-scan>

</beans>

第四步：测试安装发布

测试：

鼠标右击工程 Run As->Maven test

Maven自动会对jUnit写的测试代码进行测试，并且显示测试结果，如图6-18：

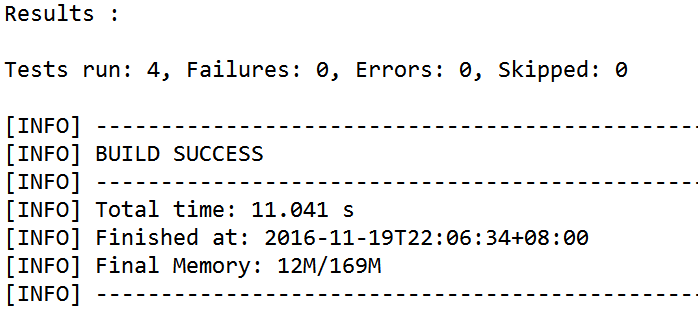


图6-18：测试提示

安装：

鼠标右击工程 Run As ->Maven install

Maven会自动将工程代码编译，运行完测试代码，所有通过后，打包成构件，发布到本地仓库。结果如图6-19：

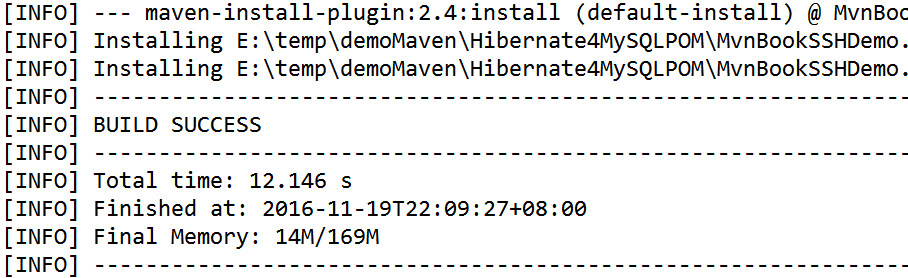


图6-19 DAO Hibernate项目构建提示

发布：

鼠标右击工程 Run As -> Maven build...

在弹出框的Goals输入框中输入deploy，点击Run按钮，Maven会自动将工程构件，发布到指定的私服仓库中，效果图如下显示，但是要注意，一定要在pom.xml中配置distributionManagement，指定发布的位置，如图6-20。

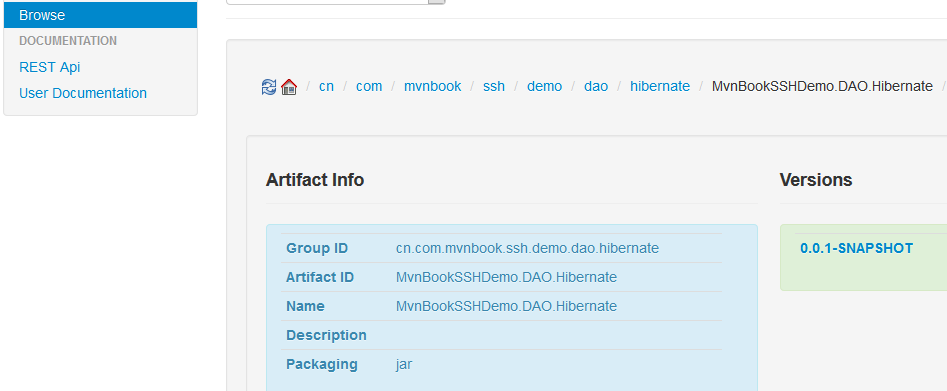


图6-20 DAO Hibernate私服发布

### 6.3.3 实现 Service模块

同DAO层定义的接口类似，前我们将Service的接口定义好，并且发布成一个单独的构件了，解析来由独立的开发人员，在自己的电脑上创建一个新的工程，继承SpringPOM，集成DAO接口的依赖和Service接口的依赖，独立进行Service的实现代码编写和测试。

因为要对Service实现方法进行测试，编码的时候可以面向接口编程，测试的时候，肯定要基于DAO的实现才能操作数据库数据，所以测试的时候，还需要额外的添加前面Hibernate的DAO实现依赖，不过该依赖的score是test，也就是只在测试的时候有效，详细情况，请注意接下来介绍的工程pom.xml中的备注。

接下来，我们按类似Hibernate的DAO实现的思路，介绍Service的实现模块。

第一步：创建工程，配置好pom.xml

同以前一样，创建一个Maven工程，工程目录结构图6-21如下：

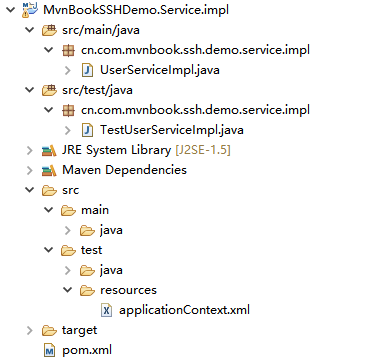


图6-21 Maven Service项目结构

（随书代码[MvnBookSSHDemo.Service.impl](代码/MvnBookSSHDemo.Service.impl)）

根据本节开篇的介绍，我们需要在pom.xml中做如下几个事情：

1. 配置继承SpringPOM构件的信息（里面配置了Spring需要的依赖）
2. 添加DAO接口构件和Service接口构件的依赖
3. 添加Hibernate DAO实现构件的依赖，作用范围是test。

请查看如下pom.xml，注意加粗部分内容和注释，细心的同学会发现里面没有添加DAO接口的构件依赖，只添加Service接口的构件依赖，同我们前面介绍的第二点要求不符合。原因是Service接口构件内部，有配置好对应DAO接口构件的依赖，只要我们在这里配置Service接口构件的依赖，Maven会在加载Service接口构件依赖的同时，自动的连带着将Service接口构件自己内部所需要的其它依赖加进来。

pom.xml  
（随书代码[MvnBookSSHDemo.Service.impl\pom.xml](代码/MvnBookSSHDemo.Service.impl/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

**<!--继承SpringPOM构件-->**

**<parent>**

**<groupId>cn.com.mvnbook.pom</groupId>**

**<artifactId>SpringPOM</artifactId>**

**<version>0.0.1-SNAPSHOT</version>**

**</parent>**

<groupId>cn.com.mvnbook.ssh.demo</groupId>

<artifactId>MvnBookSSHDemo.Service.impl</artifactId>

<packaging>jar</packaging>

<name>MvnBookSSHDemo.Service.impl</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

**<!--Service接口构件依赖-->**

**<dependency>**

**<groupId>cn.com.mvnbook.ssh.demo</groupId>**

**<artifactId>MvnBookSSHDemo.Service</artifactId>**

**<version>0.0.1-SNAPSHOT</version>**

**</dependency>**

**<!--Hibernate DAO实现构件依赖-->**

**<dependency>**

**<groupId>cn.com.mvnbook.ssh.demo.dao.hibernate</groupId>**

**<artifactId>MvnBookSSHDemo.DAO.Hibernate</artifactId>**

**<version>0.0.1-SNAPSHOT</version>**

**<!--作用范围-->**

**<scope>test</scope>**

**</dependency>**

</dependencies>

</project>

第二步：编写Service实现代码

Service的实现代码相对比较简单，只是要有Spring4容器管理相关的基础，因为里面用到Spring内部的组件注解、依赖注入注解和事务管理注解，详情请看代码和Spring4相关的资料。

UserServiceImpl.java  
（随书代码  
[MvnBookSSHDemo.Service.impl\src\main\java\cn\com\mvnbook\ssh\demo\service\impl\UserServiceImpl.java](代码/MvnBookSSHDemo.Service.impl/src/main/java/cn/com/mvnbook/ssh/demo/service/impl/UserServiceImpl.java)）

package cn.com.mvnbook.ssh.demo.service.impl;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Propagation;

import org.springframework.transaction.annotation.Transactional;

import cn.com.mvnbook.ssh.demo.dao.IMvnUserDAO;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

import cn.com.mvnbook.ssh.demo.service.IUserService;

@Service("userService") // 注册成服务组件

@Transactional // 要求启动事务

public class UserServiceImpl implements IUserService {

@Autowired // 自动根据类型注入

@Qualifier("userDAO") // 根据name注入

private IMvnUserDAO userDAO;

@Transactional(propagation=Propagation.REQUIRED)//需要進行事务管理

public void createUser(MvnUser user) {

// 验证输入的用户是否为null

if (user == null) {

throw new RuntimeException("创建的user不能为null");

}

// 验证用户名是否有存在

MvnUser u = this.userDAO.findUserByUserName(user.getUrUserName());

if (u != null) {

throw new RuntimeException(u.getUrUserName() + " 已经存在");

}

this.userDAO.addUser(user);

}

@Transactional(propagation=Propagation.REQUIRED)//需要進行事务管理

public void editUser(int age, String status, int id) {

// 根据id找到以前的用户对象

MvnUser user = this.userDAO.findUserById(id);

// 判断用户是否存在，不存在抛异常，存在就更新

if (user == null) {

throw new RuntimeException("id为" + id + "用户不存在");

} else {

user.setUrAge(age);

user.setUrStatus(status);

this.userDAO.update(user);

}

}

@Transactional(propagation=Propagation.REQUIRED)//需要進行事务管理

public void deleteUser(int id) {

// 根据id找到以前的用户对象

MvnUser user = this.userDAO.findUserById(id);

// 判断用户是否存在，不存在抛异常，存在就删除

if (user == null) {

throw new RuntimeException("id为" + id + "用户不存在");

} else {

this.userDAO.deleteUser(user);

}

}

@Transactional(readOnly=true)//只读，不需要進行事务管理

public MvnUser searchUser(int id) {

MvnUser user = null;

user = this.userDAO.findUserById(id);

return user;

}

@Transactional(readOnly=true)//只读，不需要進行事务管理

public MvnUser searchUser(String userName) {

MvnUser user = null;

user = this.userDAO.findUserByUserName(userName);

return user;

}

@Transactional(readOnly=true)//只读，不需要進行事务管理

public List<MvnUser> searchUsers() {

List<MvnUser> userList = null;

userList = this.userDAO.findUsers();

return userList;

}

}

第三步：编写Service的测试案例代码和必须的配置资源文件

因为测试代码的测试环境是依赖Spring容器的，所以测试部分的内容，除了有针对UserServiceImpl.java的测试案例类之外，需要配置一个applicationContext.xml。而且还要注意，不管是测试类还是测试资源，都需要放在src/test的对应子目录下（看结构图）。

TestUserServiceImpl.java  
（随书代码  
[MvnBookSSHDemo.Service.impl\src\test\java\cn\com\mvnbook\ssh\demo\service\impl\TestUserServiceImpl.java](代码/MvnBookSSHDemo.Service.impl/src/test/java/cn/com/mvnbook/ssh/demo/service/impl/TestUserServiceImpl.java)）

package cn.com.mvnbook.ssh.demo.service.impl;

import java.util.List;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

import cn.com.mvnbook.ssh.demo.entity.Status;

import cn.com.mvnbook.ssh.demo.service.IUserService;

import junit.framework.Assert;

public class TestUserServiceImpl {

private IUserService userService;

private ApplicationContext ctx = null;

@Before

public void init(){

this.ctx = new ClassPathXmlApplicationContext("applicationContext.xml");

this.userService = (IUserService)ctx.getBean("userService");

}

@Test

public void testCreateUser(){

MvnUser user = new MvnUser();

user.setUrAge(11);

user.setUrPassword("11");

user.setUrStatus(Status.ACTIVE.getStatus());

user.setUrUserName("service1");

this.userService.createUser(user);

MvnUser u = this.userService.searchUser("service1");

boolean bool = u != null && u.getUrAge()==11 && u.getUrStatus().equals(Status.ACTIVE.getStatus());

Assert.assertTrue(bool);

// 删除用户

this.userService.deleteUser(u.getUrId());

}

@Test

public void testEditUser(){

MvnUser user = new MvnUser();

user.setUrAge(11);

user.setUrPassword("11");

user.setUrStatus(Status.ACTIVE.getStatus());

user.setUrUserName("service1");

this.userService.createUser(user);

MvnUser u = this.userService.searchUser("service1");

this.userService.editUser(88, Status.INACTIVE.getStatus(), u.getUrId());

u = this.userService.searchUser("service1");

Assert.assertTrue(u.getUrAge()==88 && u.getUrStatus().equals(Status.INACTIVE.getStatus()));

this.userService.deleteUser(u.getUrId());

}

@Test

public void testDeleteUser(){

MvnUser user = new MvnUser();

user.setUrAge(11);

user.setUrPassword("11");

user.setUrStatus(Status.ACTIVE.getStatus());

user.setUrUserName("service1");

this.userService.createUser(user);

MvnUser u = this.userService.searchUser("service1");

this.userService.deleteUser(u.getUrId());

MvnUser u2 = this.userService.searchUser(u.getUrId());

Assert.assertTrue(u != null && u2 == null);

}

@Test

public void testSearchUserById(){

MvnUser user = this.userService.searchUser(1);

Assert.assertNotNull(user);

}

@Test

public void testSearchUserByUserName(){

MvnUser user = this.userService.searchUser("zhangsan");

Assert.assertNotNull(user);

}

@Test

public void testSearchUsers(){

List<MvnUser> userList = this.userService.searchUsers();

Assert.assertTrue(userList != null && userList.size()>0);

}

@After

public void destory(){

this.userService = null;

this.ctx = null;

}

}

applicationContext.xml  
（随书代码  
[MvnBookSSHDemo.Service.impl\src\test\resources\applicationContext.xml](代码/MvnBookSSHDemo.Service.impl/src/test/resources/applicationContext.xml)）

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.0.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-4.0.xsd">

<context:component-scan

base-package="cn.com.mvnbook.ssh.demo"></context:component-scan>

</beans>

第四步：测试 安装 和 发布

这里的测试 安装 和 发布等，同Hibernate DAO实现里面的一样。

1. 测试

工程鼠标右击 Run As -> Maven test

效果图6-22如下：

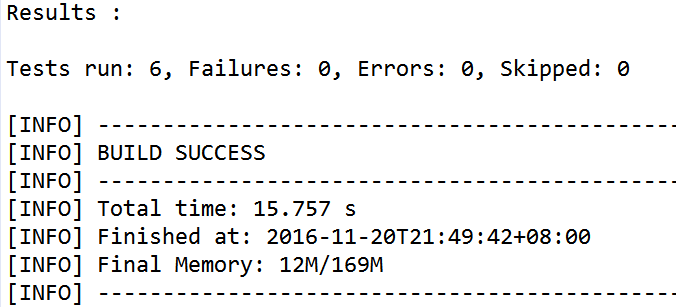


图6-22 Maven Service测试提示

1. 安装

工程鼠标右击 Run As ->Maven Install

效果图6-23如下：

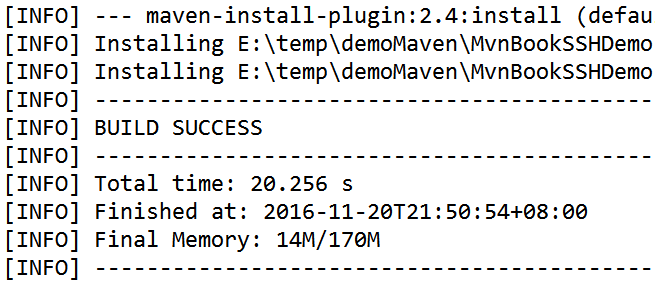


图6-23 Maven Service安装提示

1. 发布

工程鼠标右键 Run As -> Maven build...

在打开窗口的 Goals输入框中输入 deploy，点击Run按钮

效果图6-24如下：

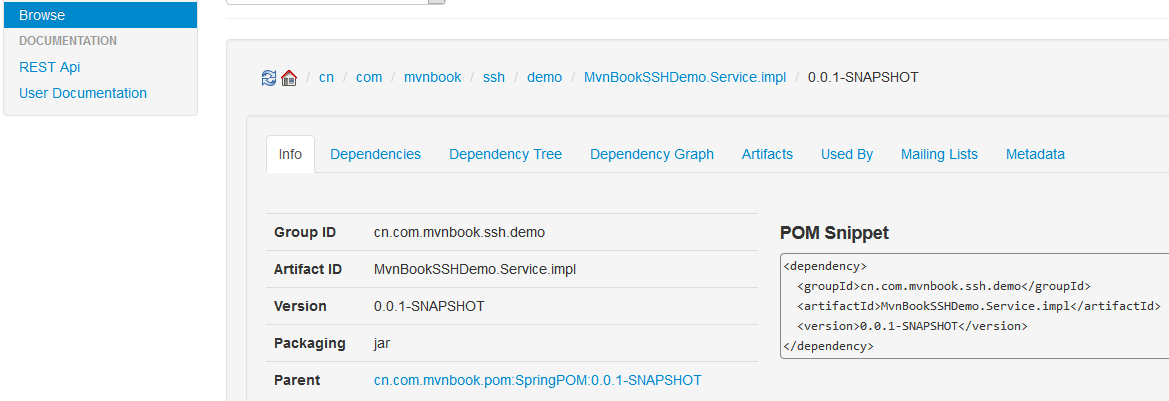


图6-24 Maven Service发布

### 6.3.4 实现Struts2 Web模块

前面我们分别基于Spring和Hibernate实现了Service接口和DAO接口功能，接下来我们基于Struts2实现Web层功能。

根据我们前面用Jsp/Servlet实现，对需求的理解和Struts2开发的相关组件的了解（Struts2需要单独的参考其他资料），Struts2 Web层的代码，需要做如下工作：

1. 实现视图层代码（jsp）

视图层代码同我们以前用jsp/servlet开发的内容一样，两个jsp

Index.jsp 首页框架jsp

userList.jsp 显示用户列表的jsp

1. 编写Action代码

UserAction.java 实现用户CRUD的所有控制逻辑代码

1. Spring容器的配置文件

applicationContext.xml 配置spring容器的初始化组件

1. 编写struts.xml配置文件

完成Struts常量的配置和Action的配置

1. 配置web.xml

配置struts的入口过滤器 和 Spring的初始化Listener

明确了目标，解析来我们按步骤实现功能

第一步：创建web工程

基于webapp的archetypes创建Web工程，我这用的是webapp-jee5，如图6-25：

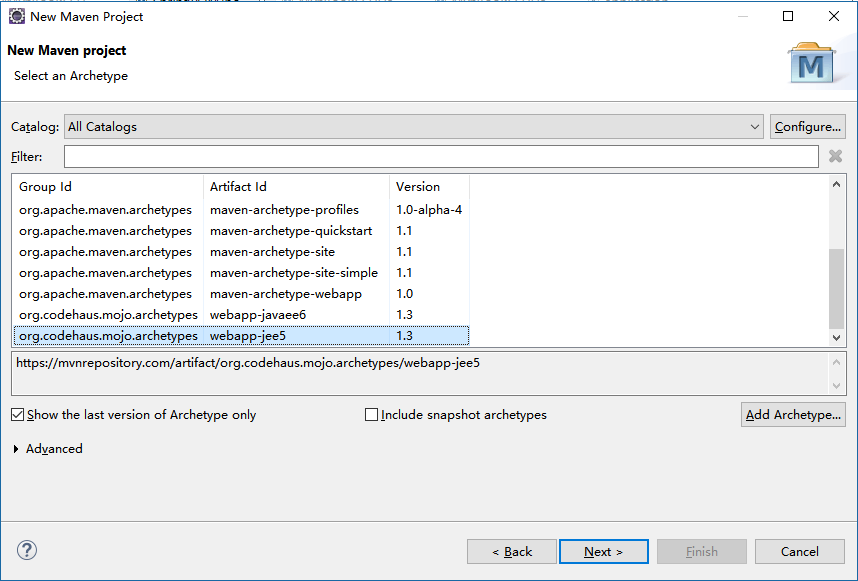


图6-25 选择webapp-jee5创建web应用

点击下一步，在输入框中输入对应信息（根据自己情况来），点击Finish，创建一个Maven的Web工程，仓库目录结构如下图6-26：

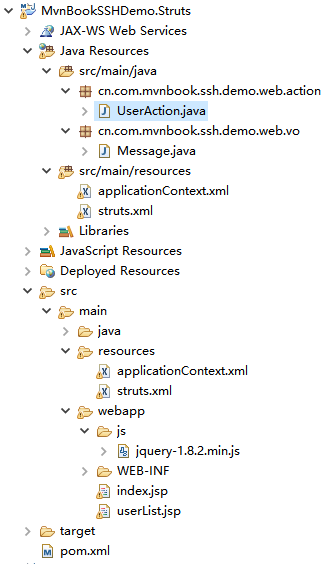


图6-26 Struts web工程结构

（随书代码[MvnBookSSHDemo.Struts](代码/MvnBookSSHDemo.Struts)）

第二步：配置依赖和插件

Pom.xml  
（随书代码[MvnBookSSHDemo.Struts\pom.xml](代码/MvnBookSSHDemo.Struts/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>StrutsPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.com.mvnbook.ssh.demo</groupId>

<artifactId>MvnBookSSHDemo.Struts</artifactId>

**<packaging>war</packaging>**

<name>MvnBookSSHDemo.Struts</name>

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

<dependencies>

<!-- struts json插件 -->

<!-- https://mvnrepository.com/artifact/org.apache.struts/struts2-json-plugin -->

**<dependency>**

**<groupId>org.apache.struts</groupId>**

**<artifactId>struts2-json-plugin</artifactId>**

**<version>2.3.28</version>**

**</dependency>**

<dependency>

<groupId>cn.com.mvnbook.ssh.demo.dao.hibernate</groupId>

<artifactId>MvnBookSSHDemo.DAO.Hibernate</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>cn.com.mvnbook.ssh.demo</groupId>

<artifactId>MvnBookSSHDemo.Service</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>cn.com.mvnbook.ssh.demo</groupId>

<artifactId>MvnBookSSHDemo.Service.impl</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

</dependencies>

<build>

<plugins>

<!-- <plugin> <groupId>org.mortbay.jetty</groupId> <artifactId>maven-jetty-plugin</artifactId>

<version>6.1.26</version> <configuration> <webAppSourceDirectory>${basedir}/src/main/webapp</webAppSourceDirectory>

</configuration> </plugin> -->

<plugin>

<!-- 指定插件名称及版本号 -->

**<groupId>org.codehaus.cargo</groupId>**

**<artifactId>cargo-maven2-plugin</artifactId>**

<version>1.4.8</version>

<configuration>

<wait>true</wait> <!--是否说明，操作start、stop等后续操作必须等前面操作完成才能继续 -->

<container> <!-- 容器的配置 -->

<containerId>tomcat7x</containerId> <!-- 指定tomcat版本 -->

<type>installed</type> <!-- 指定类型：standalone, installed等 -->

<home>C:\java\servers\apache-tomcat-7.0.69\_64</home> <!-- 指定Tomcat的位置，即catalina.home -->

</container>

<configuration> <!-- 具体的配置 -->

<type>existing</type> <!-- 类型，existing:存在 -->

<home>C:\java\servers\apache-tomcat-7.0.69\_64</home> <!-- Tomcat的位置，即catalina.home -->

</configuration>

<deployables> <!-- 部署设置 -->

<deployable> <!-- 部署的War包名等 -->

**<groupId>cn.com.mvnbook.ssh.demo</groupId>**

**<artifactId>MvnBookSSHDemo.Struts</artifactId>**

**<type>war</type>**

**<properties><!-- 部署路径 -->**

**<context>MvnBookSSHDemo</context> </properties>**

</deployable>

</deployables>

<deployer> <!-- 部署配置 -->

<type>installed</type> <!-- 类型 -->

</deployer>

</configuration>

<executions>

<!-- 执行的动作 -->

<execution>

<id>verify-deployer</id>

<phase>install</phase> <!-- 解析install -->

<goals>

<goal>deployer-deploy</goal>

</goals>

</execution>

<execution>

<id>clean-deployer</id>

<phase>clean</phase>

<goals>

<goal>deployer-undeploy</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

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

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

<version>2.0.2</version>

<configuration>

<source>1.5</source>

<target>1.5</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

说明：

相对以前的工程，这里有如下几点不同：

1、当前是webapp工程，packaging要设置成war

2、在操作过程中，添加、修改、删除的返回要用json，所以这里需要集成struts-json依赖

3、本工程不是用的jetty插件发布web应用，用的cargo-maven2-plugin插件，直接发布到指定tomcat的webapps目录下。

4、cargo-maven2-plugin插件的配置说明，在文档中有对应的注释，这里强调要注意的地方是粗体显示部分。groupId和artifactId是当前工程的对应值，context是发布到web服务器中的上下文路径名称（也是打成war包后，war包的名称）

第三步：添加实现代码

1. 视图层代码文件内容

index.jsp  
（随书代码[MvnBookSSHDemo.Struts\src\main\webapp\index.jsp](代码/MvnBookSSHDemo.Struts/src/main/webapp/index.jsp)）

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">

<title>测试用户CRUD操作</title>

<script type="text/javascript" src="js/jquery-1.8.2.min.js"></script>

<script type="text/javascript">

$(document).ready(function() {

// 页面加载完后，自动发searchUsersServlet请求，加载到userListDiv中显示

$("#userListDiv").load("userAction!searchUsers.action");

});

// 新增 按钮事件触发函数

function toAdd() {

// 获取addForm中的请求信息

var \_data = $("#addForm").serialize();

//alert(\_data);

// 发添加新用户的Ajax请求

$.ajax({

type : 'post',

url : 'userAction!addUser.action',

data : \_data,

success : function(msg) {

alert(msg.message);

// 更新最新的用户列表信息

$("#userListDiv").load("userAction!searchUsers.action");

}

});

}

function toEdit() {

var \_data = $("#editForm").serialize();

alert(\_data);

$.ajax({

type : 'post',

url : 'userAction!editUser.action',

data : \_data,

success : function(msg) {

alert(msg.message);

$("#userListDiv").load("userAction!searchUsers.action");

}

});

}

function toDelete() {

var chks = $("input[name='checkbox']:checked");

if (chks.length == 0) {

alert("请选择要删除的用户");

} else if (chks.length > 1) {

alert("一次只能删除一个用户");

} else {

var to = confirm("您确定要删除选中的用户?");

if (to) {

var \_data = "id=" + chks.val();

$.ajax({

type : 'post',

data : \_data,

url : 'userAction!deleteUser.action',

success : function(msg) {

alert(msg);

$("#userListDiv").load("userAction!searchUsers.action");

}

});

}

}

}

function toShowAdd() {

$("#LayerAdd").show(1000);

}

function toShowEdit() {

//alert($("input[name='checkbox']:checked").length);

var chks = $("input[name='checkbox']:checked");

if (chks.length == 0) {

alert("请选择要编辑的用户");

} else if (chks.length > 1) {

alert("一次只能修改一个用户");

} else {

var \_data = "id=" + chks.val();

$.ajax({

type : 'post',

data : \_data,

url : 'userAction!searchUser.action?type=byId',

dataType : 'json',

success : function(msg) {

$("#editForm #id").val(msg.urId);

$("#editForm #userName").val(msg.urUserName);

$("#editForm #age").val(msg.urAge);

$("#editForm #status").val(msg.urStatus);

//alert($("#editForm #age").val());

$("#LayerEdit").show(1000);

}

});

}

}

function toCloseAdd() {

$("#LayerAdd").hide(1000);

}

function toCloseEdit() {

$("#LayerEdit").hide(1000);

}

</script>

<style type="text/css">

<!--

.STYLE2 {

color: #000000

}

#LayerAdd {

position: absolute;

left: 113px;

top: 183px;

width: 434px;

height: 193px;

z-index: 1;

background-color: #99FFFF;

display: none;

}

#LayerEdit {

position: absolute;

left: 113px;

top: 183px;

width: 434px;

height: 193px;

z-index: 1;

background-color: #99FFFF;

display: none;

}

-->

</style>

</head>

<body>

<div id="LayerAdd">

<form name="addForm" name="addForm" id="addForm" method="post"

action="">

<table width="98%" border="0" align="center" cellpadding="0"

cellspacing="0">

<tr>

<td colspan="2" align="center"><strong><BR>添加新用户<br></strong></td>

</tr>

<tr>

<td width="47%" align="right">用户名：</td>

<td width="53%"><input name="user.urUserName" type="text"

id="userName"></td>

</tr>

<tr>

<td align="right">密码：</td>

<td><input name="user.urPassword" type="password" id="password"></td>

</tr>

<tr>

<td align="right">年龄：</td>

<td><input name="user.urAge" type="text" id="age"></td>

</tr>

<tr>

<td colspan="2">&nbsp;</td>

</tr>

<tr>

<td colspan="2" align="center"><input type="button"

name="Submit4" value="添加" onclick="toAdd()"> <input

type="button" name="Submit5" value="关闭" onclick="toCloseAdd()"></td>

</tr>

</table>

</form>

</div>

<div id="LayerEdit">

<form name="editForm" id="editForm" method="post" action="">

<input type="hidden" name="id" id="id" />

<table width="98%" border="0" align="center" cellpadding="0"

cellspacing="0">

<tr>

<td colspan="2" align="center"><strong><br>

修改用户信息<br> </strong></td>

</tr>

<tr>

<td width="47%" align="right">用户名：</td>

<td width="53%"><input name="userName" type="text"

id="userName" readonly="readonly"></td>

</tr>

<tr>

<td align="right">年龄：</td>

<td><input name="age" type="text" id="age"></td>

</tr>

<tr>

<td align="right">状态：</td>

<td><select name="status" id="status">

<option value="Active">Active</option>

<option value="Inactive">Inactive</option>

<option value="Locked">Locked</option>

<option value="Deleted">Deleted</option>

</select></td>

</tr>

<tr>

<td colspan="2">&nbsp;</td>

</tr>

<tr>

<td colspan="2" align="center"><input type="button"

name="Submit4" value="修改" onclick="toEdit()"> <input

type="button" name="Submit5" value="关闭" onclick="toCloseEdit()"></td>

</tr>

</table>

</form>

</div>

<p>&nbsp;</p>

<p>测试用户CRUD页面</p>

<table width="539" border="1">

<tr>

<td colspan="5" align="right"><input type="button" name="Submit"

value="新增" onclick="toShowAdd()"> <input type="submit"

name="Submit2" value="修改" onclick="toShowEdit()"> <input

type="button" name="Submit3" value="删除" onclick="toDelete()"></td>

</tr>

<tr>

<td>

<div id="userListDiv"></div>

</td>

</tr>

</table>

<p>&nbsp;</p>

</body>

</html>

userList.jsp  
（随书代码[MvnBookSSHDemo.Struts\src\main\webapp\userList.jsp](代码/MvnBookSSHDemo.Struts/src/main/webapp/userList.jsp)）

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<table width="100%" border="1">

<tr>

<td width="51" bgcolor="#CCCCCC"><div align="center"><strong><span class="STYLE2">选择</span></strong></div></td>

<td width="111" bgcolor="#CCCCCC"><div align="center"><strong>序号</strong></div></td>

<td width="137" bgcolor="#CCCCCC"><div align="center"><strong>用户名</strong></div></td>

<td width="105" bgcolor="#CCCCCC"><div align="center"><strong>年龄</strong></div></td>

<td width="101" bgcolor="#CCCCCC"><div align="center"><strong>状态</strong></div></td>

</tr>

<c:forEach items="${userList }" var="\_user" varStatus="status">

<c:choose>

<c:when test="${status.index%2==0 }">

<tr bgcolor="#FFFFFF">

</c:when>

<c:otherwise>

<tr bgcolor="#99FFFF">

</c:otherwise>

</c:choose>

<td><input type="checkbox" name="checkbox" value="${\_user.urId }"></td>

<td>${status.index+1 }</td>

<td>${\_user.urUserName }</td>

<td>${\_user.urAge }</td>

<td>${\_user.urStatus }</td>

</tr>

</c:forEach>

</table>

说明：

1、Index.jsp中的js，用到了jQuery，要注意将jQuery的js代码添加到应用中，并且用script应用到页面。

2、index.jsp中的添加函数、修改函数和删除函数，从后台返回的是用json封装的提示信息。

1. Action代码

Message.java :用户封装返回到页面上的操作提示信息  
（随书代码  
[MvnBookSSHDemo.Struts\src\main\java\cn\com\mvnbook\ssh\demo\web\vo\Message.java](代码/MvnBookSSHDemo.Struts/src/main/java/cn/com/mvnbook/ssh/demo/web/vo/Message.java)）

package cn.com.mvnbook.ssh.demo.web.vo;

public class Message {

private String code;

private String message;

public Message(String message) {

super();

this.message = message;

}

public String getCode() {

return code;

}

public void setCode(String code) {

this.code = code;

}

public String getMessage() {

return message;

}

public void setMessage(String message) {

this.message = message;

}

}

UserAction.java  
（随书代码  
[MvnBookSSHDemo.Struts\src\main\java\cn\com\mvnbook\ssh\demo\web\action\UserAction.java](代码/MvnBookSSHDemo.Struts/src/main/java/cn/com/mvnbook/ssh/demo/web/action/UserAction.java)）

package cn.com.mvnbook.ssh.demo.web.action;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Controller;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

import cn.com.mvnbook.ssh.demo.service.IUserService;

import cn.com.mvnbook.ssh.demo.web.vo.Message;

@Controller("userAction")

public class UserAction {

@Autowired

@Qualifier("userService")

private IUserService userService;

private MvnUser user;

private int id;

private String userName;

private String type;

private String status;

private int age;

private Message message;

// 新增用户的执行方法

public String addUser() {

String result = "addSuccess";

message = new Message("添加成功");

try {

// 调用service，创建用户

userService.createUser(user);

} catch (Exception e) {

e.printStackTrace();

message.setMessage("添加失败：" + e.getMessage());

}

return result;

}

// 删除用户的执行方法

public String deleteUser() {

String result = "deleteSuccess";

message = new Message("删除成功");

try {

userService.deleteUser(id);

} catch (Exception e) {

e.printStackTrace();

message.setMessage("删除失败:" + e.getMessage());

}

return result;

}

// 修改用户的执行方法

public String editUser() {

String result = "editSuccess";

this.message = new Message("修改成功!");

try {

userService.editUser(age, status, id);

} catch (Exception e) {

e.printStackTrace();

message.setMessage("修改失败:" + e.getMessage());

}

return result;

}

// 根据id或用户名查找单个用户的执行方法

public String searchUser() {

String result = "searchUserSeuccess";

if ("byId".equals(type)) {

user = this.userService.searchUser(id);

} else {

user = this.userService.searchUser(userName);

}

return result;

}

private List<MvnUser> userList;

// 查询所有用户列表的执行方法

public String searchUsers() {

String result = "searchUsersSuccess";

this.userList = this.userService.searchUsers();

return result;

}

public List<MvnUser> getUserList() {

return userList;

}

public void setUserList(List<MvnUser> userList) {

this.userList = userList;

}

public MvnUser getUser() {

return user;

}

public void setUser(MvnUser user) {

this.user = user;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getUserName() {

return userName;

}

public void setUserName(String userName) {

this.userName = userName;

}

public String getType() {

return type;

}

public void setType(String type) {

this.type = type;

}

public String getStatus() {

return status;

}

public void setStatus(String status) {

this.status = status;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public Message getMessage() {

return message;

}

public void setMessage(Message message) {

this.message = message;

}

}

1. applicationContext.xml  
   （随书代码  
   [MvnBookSSHDemo.Struts\src\main\resources\applicationContext.xml](代码/MvnBookSSHDemo.Struts/src/main/resources/applicationContext.xml)）

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.0.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-4.0.xsd">

<context:component-scan

base-package="cn.com.mvnbook.ssh.demo"></context:component-scan>

</beans>

1. Struts.xml代码  
   （随书代码  
   [MvnBookSSHDemo.Struts\src\main\resources\struts.xml](代码/MvnBookSSHDemo.Struts/src/main/resources/applicationContext.xml)）

<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">

<struts>

<include file="struts-default.xml"></include>

<include file="struts-plugin.xml"></include>

<!-- 从Spring容器中获取Action对象 -->

<constant name="struts.objectFactory" value="spring"></constant>

<!-- 请求和Action方法的动态对应 -->

<constant name="struts.enable.DynamicMethodInvocation" value="true"/>

<package name="t" namespace="/" extends="json-default">

<action name="userAction" class="userAction">

<result name="searchUsersSuccess">/userList.jsp</result>

<result name="searchUserSeuccess" type="json">

<param name="root">user</param>

</result>

<result name="editSuccess" type="json">

<param name="root">message</param>

</result>

<result name="deleteSuccess" type="json">

<param name="root">message</param>

</result>

<result name="addSuccess" type="json">

<param name="root">message</param>

</result>

</action>

</package>

</struts>

1. Web.xml  
   （随书代码  
   [MvnBookSSHDemo.Struts\src\main\webapp\WEB-INF\web.xml](代码/MvnBookSSHDemo.Struts/src/main/resources/applicationContext.xml)）

<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

<display-name>MvnBookSSHDemo.Struts</display-name>

<!-- 启动Spring 容器 -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/classes/applicationContext.xml</param-value>

</context-param>

<!-- struts2的总入口过滤器 -->

<filter>

<filter-name>struts2</filter-name>

<filter-class>

org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter

</filter-class>

</filter>

<filter-mapping>

<filter-name>struts2</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<session-config>

<session-timeout>

30

</session-timeout>

</session-config>

<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

</web-app>

第四步：安装发布测试

在工程上右击鼠标 Run As->Maven install

Maven自动会编译、测试代码，并且打成war包，将war包发布到指定的web服务器的发布目录。

接着我们就可以启动Tomcat服务器，用浏览器进行测试了。浏览器操作过程同前面基于jsp/servlet开发的Demo一样。

### 6.3.5 整合成SSH

按前面的操作，我们就已经分模块，独立的开发实现了各自的功能，并且能将各自的功能封装成构件，安装到本地仓库、发布到公司搭建的私服上面，供需要的地方当依赖构件使用。

中间体现了模块化的思想，同时考虑到框架的依赖配置的共性，我们也独立创建工程（POM），将每个独立框架的依赖配置，都在公共POM工程中设置好，其它要使用的工程，只需要继承它们就行了，不需要再重复的单独配置。比如MvnBookSSHDemo.Struts就是继承StrutsPOM来的。这样体现了Maven开发过程中的继承运用思想。

但是，当我们测试MvnBookSSHDemo.Struts模块功能的时候，发现前面的依赖模块的实现需要修改，这时候需要对修改的模块工程进行独立的编译、测试、打包、安装和发布，然后再测试MvnBookSSHDemo.Struts.

如果依赖的第三方模块很多，这样每次改动，都需要一个个模块进行重复操作的话，很麻烦，而且给你感觉不到是一个整体的项目，很零碎。

为了解决这个文档，Maven里面有个聚合的概念，它能将一个个依赖的模块，聚合成一个大项目（工程），我们只要对这个大项目进行构建操作，Maven自动将聚合的每个小模块一个个的构件，最后形成一个整体。

解析来，我们创建一个项目，将MvnBookSSHDemo的相关模块都聚合到一起，同时构件操作，具体步骤如下：

第一步：创建一个普通工程

这一步没有特别的，就一个普通工程就行，普通的Maven工程，仓库目录结构如下图6-27：

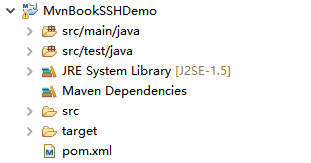


图6-27 聚合工程结构

（随书代码[MvnBookSSHDemo](代码/MvnBookSSHDemo)）

第二步：在pom.xml中，配置每个模块  
（随书代码[MvnBookSSHDemo\pom.xml](代码/MvnBookSSHDemo/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.com.mvnbook.ssh.demo</groupId>

<artifactId>MvnBookSSHDemo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<name>MvnBookSSHDemo</name>

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

**<modules>**

<module>../MvnBookSSHDemo.DAO</module>

**<module>../MvnBookSSHDemo.DAO.Hibernate</module>**

<module>../MvnBookSSHDemo.Service</module>

**<module>../MvnBookSSHDemo.Service.impl</module>**

<module>../MvnBookSSHDemo.Struts</module>

**</modules>**

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

</dependencies>

<build>

<plugins>

</plugins>

</build>

<distributionManagement>

<repository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

</distributionManagement>

</project>

说明：

这个pom.xml中没有太多信息，注意粗体部分的配置，就是将相关的依赖工程，以模块的形式聚合进来。

这些工程，都需要在同一个工作空间下，才能用 ../ 类似的相对路径进行定位应用。本Demo的工程目录结构图6-28如下：

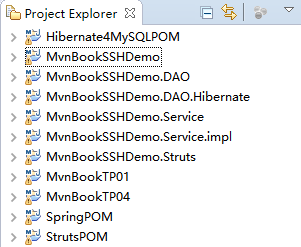


图6-28 MvnBookSSHDemo子模块工程结构

第三步：构建

要对所有模块进行编译、测试、安装、发布的话，我们都可以直接用鼠标右击击MvnBookSSHDemo工程，选择Run As ->Maven clean|test|install|build...|build等命令。

当我们选择Maven install后，Maven自动把整个工程打成MvnBookSSHDemo.war包，发布到Tomcat的webapps目录中。

同样，我们选择Maven build... ，输入deploy点击run的话，在安装的Archiva私服上，就可以浏览到所有的构件，如图6-29：

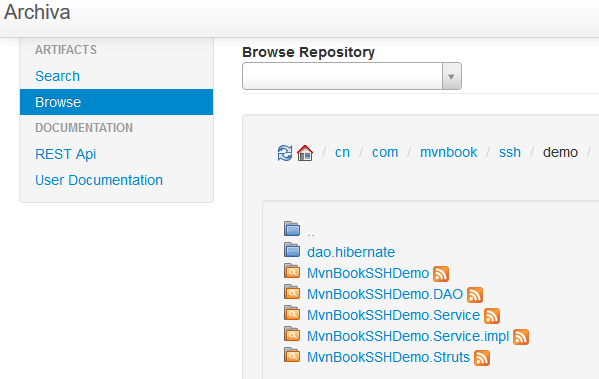


图6-29 MvnBookSSHDemo项目发布

第四步：测试

启动Tomcat，就可以做如下页面的测试操作了，只是请各位注意：我们前面有大家Archiva私服的，如果这个私服在开发的时候启动着，并且私服就搭建在我们自己的电脑上（我们自己在研究，肯定是这样），请将它关闭后再启动测试应用的Tomcat，或者修改测试应用Tomcat的端口，否则会出现端口冲突异常。因为Archiva也是用的Tomcat服务器，默认端口就是8080系列的。

1. 首页，如图6-30

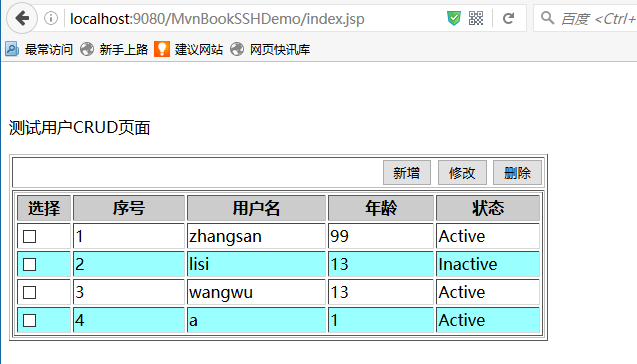


图6-30 首页

1. 添加用户

点击新增按钮，显示添加用户页面图6-31：

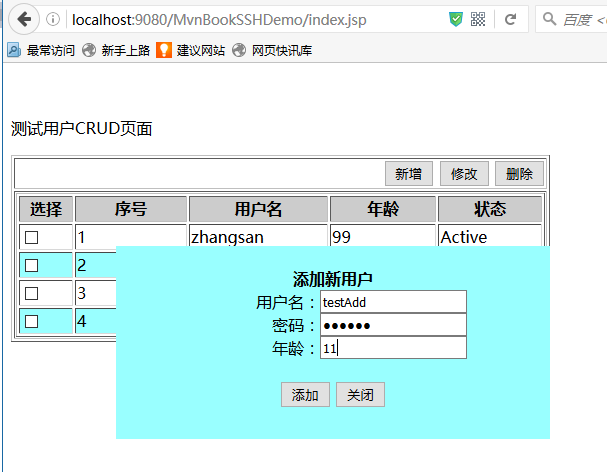


图6-31 添加用户

点击添加后关闭，会查看到新增了一个用户，图6-32：

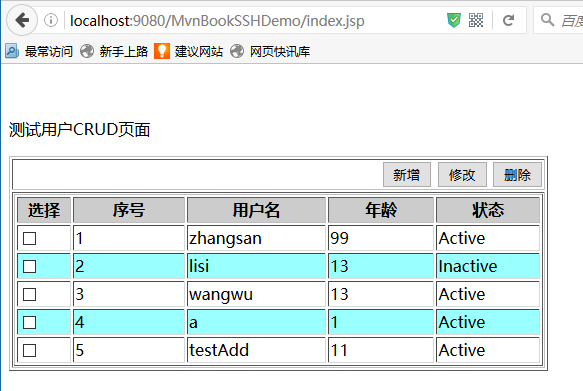


图6-32 查询用户列表

1. 修改用户

选择要修改的用户，点击修改按钮，在如下页面修改提交，实现选择用户信息的修改。如图6-33

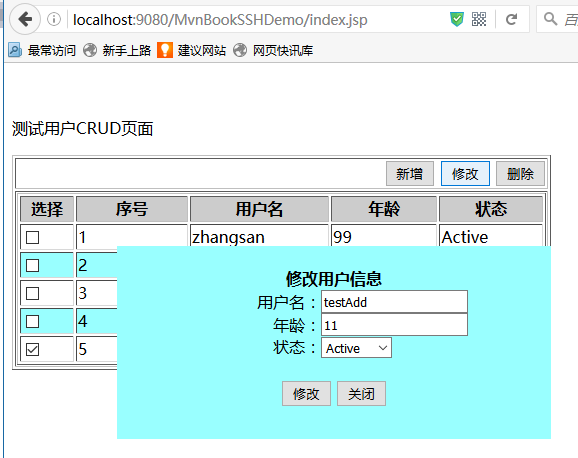


图6-33 修改用户

1. 删除用户

选择要删除的用，点击删除按钮，出现如下页面，点确定就直接删除，如图6-34



图6-34 删除用户

到现在，我们有基于在Maven上面，基于SSH完成了一个用户的CRUD功能，中间还体现了项目的模块思想、面向接口编程思想和Maven的继承、聚合思想。

## 6.4 实现SpringMVC+Spring+MyBatis框架应用

前面研究了基于Maven，使用Struts+Spring+Hibernate(SSH)框架，模仿企业级应用的开发过程，实现了用户的CRUD功能。接下来我们再使用SpringMVC+Spring+MyBatis(SSM)框架，同样模仿企业级应用的开发过程，实现用户的CRUD功能。目的很明确，让大家能直接在企业中，基于SSH和SSM两个流行框架进行项目开发，减少学习到使用之间的转换过程。当然，我们企业中用的是其他的框架，相信大家有了SSH和SSM的基础，应该可以顺利适应的。

### 6.4.1 创建公共POM

为了公司项目有个正常的沉淀和重用，我们先创建基于SSM框架开发的公共POM构件，以免以后项目和开发人员的重复搭建。

基于SSM框架，我们可以独立搭建SpringMVC 、Spring 和 MyBatis三个基本的POM构件，其中Spring的POM，我们有在前面的样例中搭建好了，叫SpringPOM构件，这里就不重复了，这里主要是介绍大家SpringMVC和MyBatis构件POM。

考虑到我们前面已经有介绍过很多Maven工程的创建了，这里就不重复介绍工程的创建步骤，着重介绍关键的配置文件和代码。

1. SpringMVC POM

因为SpringMVC同Spring是以前的，我们有在SpringPOM配置好了，所有在SpringMVC POM中，我们只需要继承SpringPOM好了。

另外，SpringMVC封装的是Web层应用，底层使用的是jsp/servlet技术，所以在SpringMVC POM中，我们需要加入jsp/servlet相关的依赖。

最后，因为我们的用户CRUD样例中，需要有JSON的响应，而且JSON的请求和响应，在实际项目中也是很普通的，所以我们在SpringMVC POM中，也添加了jackson的相关依赖。

pom.xml的详细内容如下：  
（随书代码[SpringPOM\pom.xml](代码/SpringPOM/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<!-- 继承SpringPOM中的pom -->

<parent>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>SpringMVCPOM</artifactId>

<packaging>pom</packaging>

<name>SpringMVCPOM</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<!-- jsp servlet -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>jsp-api</artifactId>

<version>2.1</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<!-- json组件 -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.5.4</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-core</artifactId>

<version>2.5.4</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-annotations</artifactId>

<version>2.5.0</version>

</dependency>

</dependencies>

<distributionManagement>

<repository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

</distributionManagement>

</project>

1. MyBatis POM

考虑到我们的MyBatis最终要同Spring集成，并且是基于MySQL数据库的，在MyBatis的POM中，我们需要定义如下关键的依赖：

1. MyBatis构件
2. MyBatis同Spring集成的构件
3. 连接池构件
4. MySQL数据库驱动构件

MyBatis4MySQLPOM的pom.xml详细内容如下：  
（随书代码 [MyBatis4MySQLPOM \pom.xml](代码/MyBatis4MySQLPOM/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>MyBatis4MySQLPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<name>MyBatis4MySQLPOM</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<!-- 3.6.5.Final,3.3.2.GA -->

<project.build.mybatis.version>3.4.0</project.build.mybatis.version>

</properties>

<dependencies>

<!-- MyBatis -->

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>${project.build.mybatis.version}</version>

</dependency>

<!-- MyBatis同Spring的集成构件 -->

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis-spring</artifactId>

<version>1.3.0</version>

</dependency>

<dependency>

<groupId>org.mybatis.generator</groupId>

<artifactId>mybatis-generator-core</artifactId>

<version>1.3.2</version>

</dependency>

<!-- Datasource连接池 -->

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.4</version>

</dependency>

<!-- MySQL数据库驱动 -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.34</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.7</version>

<scope>test</scope>

</dependency>

</dependencies>

<distributionManagement>

<repository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

</distributionManagement>

</project>

### 6.4.2 实现MyBatis DAO模块

因为同SSH框架实现的用户CRUD功能一样，所以在SSM中，DAO层的接口同SSH中的DAO层接口是一样的，我们这里就不再重新定义了，直接重用前面样例中创建的MvnBookSSHDemo.DAO构件。这里直接创建MyBatis的DAO实现部分工程，对DAO层接口，基于MyBatis进行实现。

第一步：创建MvnBookSSMDemo.DAO.MyBatis工程

MvnBookSSMDemo.DAO.MyBatis的工程，只需要创建Maven的普通java工程就行，目录结构如下图6-35：

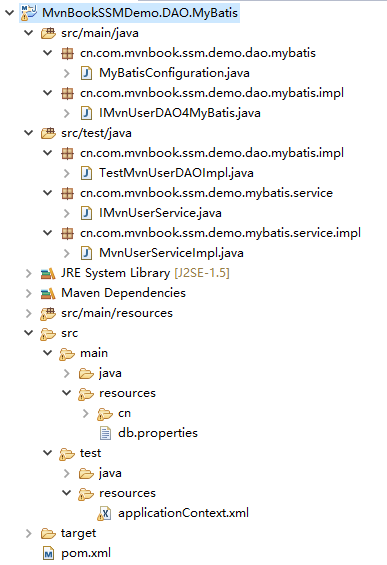


图6-35 MvnBookSSMDemo MyBatis项目结构

（随书代码[MvnBookSSMDemo.DAO.MyBatis](代码/MvnBookSSMDemo.DAO.MyBatis)）

第二步：编写pom.xml

MvnBookSSMDemo.DAO.MyBatis是基于MyBatis实现的，所哟pom先是继承MyBatis4MySQLPOM，然后考虑到用同Spring集成，并且实现的是MvnBookSSHDemo.DAO中定义的接口，所以需要添加SpringPOM和MvnBookSSHDemo.DAO的构件依赖。只是注意一下，因为SpringPOM是pom构件，所以在应用它的依赖的时候，需要指定type为pom，详细情况查看pom.xml。

pom.xml内容如下：  
（随书代码[MvnBookSSMDemo.DAO.MyBatis\pom.xml](代码/MvnBookSSMDemo.DAO.MyBatis/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

**<parent>**

**<groupId>cn.com.mvnbook.pom</groupId>**

**<artifactId>MyBatis4MySQLPOM</artifactId>**

**<version>0.0.1-SNAPSHOT</version>**

**</parent>**

<groupId>cn.com.mvnbook.ssm.demo</groupId>

<artifactId>MvnBookSSMDemo.DAO.MyBatis</artifactId>

<packaging>jar</packaging>

<name>MvnBookSSMDemo.DAO.MyBatis</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<!--SpringPOM构件依赖-->

**<dependency>**

<groupId>cn.com.mvnbook.pom</groupId>

**<artifactId>SpringPOM</artifactId>**

**<version>0.0.1-SNAPSHOT</version>**

**<type>pom</type>**

**</dependency>**

**<!--DAO接口依赖-->**

**<dependency>**

**<groupId>cn.com.mvnbook.ssh.demo</groupId>**

**<artifactId>MvnBookSSHDemo.DAO</artifactId>**

**<version>0.0.1-SNAPSHOT</version>**

</dependency>

</dependencies>

<distributionManagement>

<repository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

</distributionManagement>

</project>

第三步：编写实现代码

基于MyBatis实现的DAO持久层，我们需要写如下代码：

MyBatisConfiguration.java 配置MyBatis的基本信息，包括数据库连接信息

IMvnBookDAO4MyBatis.java 继承前面定义的DAO接口，完成MyBatis接口定义

MvnUserMapper.xml MvnUser实体的映射文件

db.properties 数据库连接信息配置文件

实现代码详细内容如下：

1. MyBatisConfiguration.java  
   （随书代码  
   [MvnBookSSMDemo.DAO.MyBatis\src\main\java\cn\com\mvnbook\ssm\demo\dao\mybatis\MyBatisConfiguration.java](代码/MvnBookSSMDemo.DAO.MyBatis/src/main/java/cn/com/mvnbook/ssm/demo/dao/mybatis/MyBatisConfiguration.java)）

package cn.com.mvnbook.ssm.demo.dao.mybatis;

import java.io.IOException;

import java.util.Properties;

import javax.sql.DataSource;

import org.apache.commons.dbcp.BasicDataSource;

import org.apache.ibatis.io.Resources;

import org.mybatis.spring.SqlSessionFactoryBean;

import org.mybatis.spring.mapper.MapperScannerConfigurer;

import org.springframework.beans.factory.annotation.Autowire;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.core.io.Resource;

import org.springframework.core.io.support.PathMatchingResourcePatternResolver;

import org.springframework.core.io.support.ResourcePatternResolver;

@Configuration//允许注解

@ComponentScan(basePackages = { "cn.com.mvnbook" })//注解类的基本包

public class MyBatisConfiguration {

// 将方法结果按Bean管理，别的地方可以按类型进行装配

@Bean(autowire = Autowire.BY\_TYPE)

public DataSource dataSource() {

// 创建数据源

BasicDataSource dataSource = new BasicDataSource();

Properties prop = null;

try {

// 读资源文件中的数据库连接信息

prop = Resources.getResourceAsProperties("db.properties");

String driverClass = prop.getProperty("jdbc.driverClassName");

String jdbcUrl = prop.getProperty("jdbc.url");

String uname = prop.getProperty("jdbc.username");

String password = prop.getProperty("jdbc.password");

dataSource.setDriverClassName(driverClass);

dataSource.setUrl(jdbcUrl);

dataSource.setUsername(uname);

dataSource.setPassword(password);

dataSource.setRemoveAbandonedTimeout(60);

dataSource.setRemoveAbandoned(true);

dataSource.setLogAbandoned(false);

dataSource.setMinIdle(10);

dataSource.setMinEvictableIdleTimeMillis(30000);

dataSource.setMaxWait(10);

dataSource.setInitialSize(2);

dataSource.setMaxActive(10);

dataSource.setTimeBetweenEvictionRunsMillis(30000);

dataSource.setValidationQuery("SELECT 1");

dataSource.setTestOnReturn(false);

dataSource.setTestOnBorrow(true);

dataSource.setTestWhileIdle(true);

} catch (IOException e) {

e.printStackTrace();

}

return dataSource;

}

// 将返回结果对象定义成一个name为sqlSessionFactory的bean

@Bean(name = "sqlSessionFactory")

public SqlSessionFactoryBean getSqlSesssionFactoryBean() {

SqlSessionFactoryBean bean = new SqlSessionFactoryBean();

bean.setDataSource(dataSource());

/\*ClassPathResource re = new ClassPathResource("myBatisConfig.xml");

bean.setConfigLocation(re);\*/

ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();

Resource[] resources;

try {

//设置MyBatis视图映射文件的路径

resources = resolver.getResources("classpath:cn/com/mvnbook/ssm/demo/dao/mybatis/entity/\*Mapper.xml");

bean.setMapperLocations(resources);

} catch (IOException e) {

e.printStackTrace();

}

return bean;

}

// 创建Mapper和接口的绑定配置器Bean，名称为mapper

@Bean(name = "mapper", autowire = Autowire.BY\_NAME)

public MapperScannerConfigurer getMapperScannerConfigurer() {

MapperScannerConfigurer conf = new MapperScannerConfigurer();

//设置要绑定的接口基本包路径

conf.setBasePackage("cn.com.mvnbook.ssm.demo.dao.mybatis.impl");

// 设置连接数据库的sqlSessionFactory

conf.setSqlSessionFactoryBeanName("sqlSessionFactory");

return conf;

}

}

1. IMvnUserDAO4MyBatis.java  
   （随书代码  
   [MvnBookSSMDemo.DAO.MyBatis\src\main\java\cn\com\mvnbook\ssm\demo\dao\mybatis\impl\IMvnUserDAO4MyBatis.java](代码/MvnBookSSMDemo.DAO.MyBatis/src/main/java/cn/com/mvnbook/ssm/demo/dao/mybatis/impl/IMvnUserDAO4MyBatis.java)）

package cn.com.mvnbook.ssm.demo.dao.mybatis.impl;

import cn.com.mvnbook.ssh.demo.dao.IMvnUserDAO;

public interface IMvnUserDAO4MyBatis extends IMvnUserDAO {

}

1. MvnUserMapper.xml  
   （随书代码  
   [MvnBookSSMDemo.DAO.MyBatis\src\main\resources\cn\com\mvnbook\ssm\demo\dao\mybatis\entity\MvnUserMapper.xml](代码/MvnBookSSMDemo.DAO.MyBatis/src/main/resources/cn/com/mvnbook/ssm/demo/dao/mybatis/entity/MvnUserMapper.xml)）

该xml文件放在src/main/resources/cn/com/mvnbook/ssm/demo/dao/mybatis/entity目录下，注意mapper元素中namespace的值需要是我们定义的DAO接口类，每个执行元素的id需要同接口类中的每个方法名称一一对应。内容如下：

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"   
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="cn.com.mvnbook.ssm.demo.dao.mybatis.impl.IMvnUserDAO4MyBatis">

<resultMap type="cn.com.mvnbook.ssh.demo.entity.MvnUser" id="UserMap">

<id column="ur\_id" property="urId"/>

<result column="ur\_user\_name" property="urUserName" />

<result column="ur\_password" property="urPassword"/>

<result column="ur\_age" property="urAge"/>

<result column="ur\_status" property="urStatus"/>

</resultMap>

<insert id="addUser" useGeneratedKeys="true" keyProperty="urId"   
 parameterType="cn.com.mvnbook.ssh.demo.entity.MvnUser">

insert into mvn\_user (ur\_user\_name,ur\_password,ur\_age,ur\_status)

values (#{urUserName},#{urPassword},#{urAge},#{urStatus})

</insert>

<update id="update" parameterType="cn.com.mvnbook.ssh.demo.entity.MvnUser">

update mvn\_user set ur\_age=#{urAge},ur\_status=#{urStatus} where ur\_id=#{urId}

</update>

<delete id="deleteUser" parameterType="cn.com.mvnbook.ssh.demo.entity.MvnUser">

delete from mvn\_user where ur\_id=#{urId}

</delete>

<select id="findUserById" resultMap="UserMap" parameterType="int">

select \* from mvn\_user where ur\_id=#{id}

</select>

<select id="findUserByUserName" resultMap="UserMap" parameterType="java.lang.String">

select \* from mvn\_user where ur\_user\_name=#{userName}

</select>

<select id="findUsers" resultMap="UserMap">

select \* from mvn\_user order by ur\_id

</select>

</mapper>

1. db.properties

（随书代码  
[MvnBookSSMDemo.DAO.MyBatis\src\main\resources\db.properties](代码/MvnBookSSMDemo.DAO.MyBatis/src/main/resources/db.properties)）

该文件需要放在src/main/resources目录下，内容如下：

jdbc.username=root

jdbc.password=root

jdbc.url=jdbc:mysql://localhost:3306/mvn\_db

jdbc.driverClassName=com.mysql.jdbc.Driver

第四步：编写测试代码

测试的中心代码是TestMvnUserDAOImp.java类，里面基于jUnit对DAO接口中的每个方法做了测试。为了完成它，同时避免MyBatisDAO集成到系统项目中的改动，添加了一个供MvnUserDAOImpl测试的服务类，里面不加业务逻辑，直接调用DAO接口中的方法，只是在类上面添加了事务管理。所以测试代码主要有如下几个部分：

1. IMvnUserService.java

临时测试用的服务层接口

（随书代码  
[MvnBookSSMDemo.DAO.MyBatis\src\test\java\cn\com\mvnbook\ssm\demo\mybatis\service\IMvnUserService.java](代码/MvnBookSSMDemo.DAO.MyBatis/src/test/java/cn/com/mvnbook/ssm/demo/mybatis/service/IMvnUserService.java)）

package cn.com.mvnbook.ssm.demo.mybatis.service;

import java.util.List;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

public interface IMvnUserService {

/\*\*

\* 在数据库中，添加一个新的MvnUser对象

\*

\* @param user

\* 需要添加的用户实体对象，该对象需要有用户名、密码、年龄和状态属性

\*

\* @return void

\* @throws RuntimeException

\* 添加失败或出现其它意外

\*/

public void addUser(MvnUser user);

/\*\*

\* 更新MvnUser对象。该对象中需要设置年龄、状态和id属性，属性和状态是要更新的新值，id为条件

\*

\* @param user

\* 需要更新的MvnUser对象

\*

\* @return void

\* @throws RuntimeException

\* 更新失败或出现其它意外

\*/

public void update(MvnUser user);

/\*\*

\* 删除MvnUser对象，该对象中需要有要删除对象的id属性，id属性为删除条件

\*

\* @param user

\* 要删除的MvnUser对象

\*

\* @return void

\* @throws RuntimeException

\* 删除失败或出现其它意外

\*/

public void deleteUser(MvnUser user);

/\*\*

\* 根据id查询对应的MvnUser对象

\*

\* @param id

\* 要查询的MvnUser对象的id

\* @return MvnUser id对应的MvnUser对象，如果没有对象，返回null

\* @throws RuntimeException

\* 出现意外情况

\*/

public MvnUser findUserById(int id);

/\*\*

\* 根据用户名查询对应的MvnUser对象

\*

\* @param userName

\* 要查询的MvnUser对象的用户名

\* @return MvnUser 用户对应的MvnUser对象，如果没有对象，返回null

\* @throws RuntimeException

\* 出现意外情况

\*/

public MvnUser findUserByUserName(String userName);

/\*\*

\* 查找数据库中所有的用户对象，以List集合的形式返回

\*

\* @return List<MvnUser> 所有用户对象的集合

\* @throws RuntimeException

\* 出现意外情况

\*/

public List<MvnUser> findUsers();

}

1. MvnUserServiceImpl.java

临时测试的服务层实现类  
（随书代码  
[MvnBookSSMDemo.DAO.MyBatis\src\test\java\cn\com\mvnbook\ssm\demo\mybatis\service\impl\MvnUserServiceImpl.java](代码/MvnBookSSMDemo.DAO.MyBatis/src/test/java/cn/com/mvnbook/ssm/demo/mybatis/service/impl/MvnUserServiceImpl.java)）

package cn.com.mvnbook.ssm.demo.mybatis.service.impl;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import cn.com.mvnbook.ssh.demo.dao.IMvnUserDAO;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

import cn.com.mvnbook.ssm.demo.mybatis.service.IMvnUserService;

@Service("userService")

@Transactional

public class MvnUserServiceImpl implements IMvnUserService {

@Autowired

private IMvnUserDAO userDAO;

public void addUser(MvnUser user) {

this.userDAO.addUser(user);

}

public void update(MvnUser user) {

this.userDAO.update(user);

}

public void deleteUser(MvnUser user) {

this.userDAO.deleteUser(user);

}

public MvnUser findUserById(int id) {

return this.userDAO.findUserById(id);

}

public MvnUser findUserByUserName(String userName) {

return this.userDAO.findUserByUserName(userName);

}

public List<MvnUser> findUsers() {

return this.userDAO.findUsers();

}

}

1. TestMvnUserDAOImpl.java

基于jUnit的单元测试类  
（随书代码  
[MvnBookSSMDemo.DAO.MyBatis\src\test\java\cn\com\mvnbook\ssm\demo\dao\mybatis\impl\TestMvnUserDAOImpl.java](代码/MvnBookSSMDemo.DAO.MyBatis/src/test/java/cn/com/mvnbook/ssm/demo/dao/mybatis/impl/TestMvnUserDAOImpl.java)）

package cn.com.mvnbook.ssm.demo.dao.mybatis.impl;

import java.util.List;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

import cn.com.mvnbook.ssh.demo.entity.Status;

import cn.com.mvnbook.ssm.demo.mybatis.service.IMvnUserService;

import junit.framework.Assert;

public class TestMvnUserDAOImpl {

private ApplicationContext ctx;

private IMvnUserService userService;

@Before

public void init(){

this.ctx = new ClassPathXmlApplicationContext("applicationContext.xml");

this.userService = (IMvnUserService)ctx.getBean("userService");

}

@Test

public void testAddUser(){

MvnUser user = new MvnUser();

user.setUrAge(11);

user.setUrPassword("11");

user.setUrStatus(Status.ACTIVE.getStatus());

user.setUrUserName("myBatisUserName");

this.userService.addUser(user);

user = this.userService.findUserByUserName("myBatisUserName");

Assert.assertTrue(user != null && user.getUrAge()==11);

this.userService.deleteUser(user);

}

@Test

public void testDeleteUser(){

MvnUser user = new MvnUser();

user.setUrAge(11);

user.setUrPassword("11");

user.setUrStatus(Status.ACTIVE.getStatus());

user.setUrUserName("myBatisUserName");

this.userService.addUser(user);

user = this.userService.findUserByUserName("myBatisUserName");

this.userService.deleteUser(user);

MvnUser u = this.userService.findUserByUserName("myBatisUserName");

Assert.assertTrue(user != null && u==null);

}

@Test

public void testFindUserById(){

MvnUser user = this.userService.findUserById(1);

Assert.assertEquals(user.getUrUserName(),"zhangsan");

}

@Test

public void testFindByUserName(){

MvnUser user = this.userService.findUserByUserName("zhangsan");

Assert.assertEquals("123", user.getUrPassword());

}

@Test

public void testFindUsers(){

List<MvnUser> userList = this.userService.findUsers();

Assert.assertTrue(userList.size()>0);

}

@After

public void destory(){

this.userService = null;

this.ctx = null;

}

}

1. applicationContext.xml

测试的时候，Spring的配置文件

（随书代码  
[MvnBookSSMDemo.DAO.MyBatis\src\test\resources\applicationContext.xml](代码/MvnBookSSMDemo.DAO.MyBatis/src/test/resources/applicationContext.xml)）

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.0.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-4.0.xsd">

<context:component-scan

base-package="cn.com.mvnbook"></context:component-scan>

</beans>

第五步：测试、安装和发布

同前面的类似：Maven test|install|deploy

### 6.4.3 实现Spring的Service层模块

SSM中的Service层实现，同SSH框架中的Service层实现一样的，唯一不同的是DAO的注入只能用@Autowired根据类型注入，不能用@Qualifier根据名称注入。因为前面MyBatis DAO的实现，使用的MyBatis3的新特征：接口和映射文件自动绑定，没有自己独立实现DAO类，更加没有自己在Spring容器中配置DAO Bean，指定Bean的名称。

MvnBookSSMDemo.Service.Impl工程的实现步骤如下：

第一步：创建MvnBookSSMDemo.Service.Impl工程

只需要创建Maven的简单Java工程，参考目录结构如下图6-36：

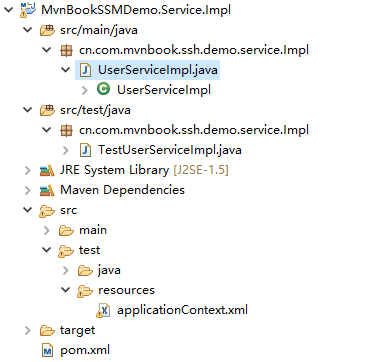


图6-36 MvnBookSSMDemo Service实现项目结构

（随书代码[MvnBookSSMDemo.Service.Impl](代码/MvnBookSSMDemo.Service.Impl)）

第二步：编写pom.xml骨架文件

Pom内容主要体现在如下几个方面：

1. 继承SpringPOM公共构件，避免重复配置对Spring相关构件的依赖
2. 添加MvnBookSSHDemo.DAO接口依赖和MvnBookSSHDemo.Service接口依赖
3. 为了方便测试，添加MvnBookSSMDemo.DAO.MyBatis依赖，scope范围是test

pom.xml的内容如下：  
（随书代码[MvnBookSSMDemo.Service.Impl\pom.xml](代码/MvnBookSSMDemo.Service.Impl/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.com.mvnbook.ssm.demo</groupId>

<artifactId>MvnBookSSMDemo.Service.Impl</artifactId>

<packaging>jar</packaging>

<name>MvnBookSSMDemo.Service.Impl</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>cn.com.mvnbook.ssh.demo</groupId>

<artifactId>MvnBookSSHDemo.DAO</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>cn.com.mvnbook.ssh.demo</groupId>

<artifactId>MvnBookSSHDemo.Service</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>cn.com.mvnbook.ssm.demo</groupId>

<artifactId>MvnBookSSMDemo.DAO.MyBatis</artifactId>

<version>0.0.1-SNAPSHOT</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

第三步：编写实现代码

Service层主要实现用户对象的CRUD功能，同SSH中的Service层的代码类似，唯一不同的是，对DAO对象的注入，只能用byType的自动注入，不能用byName。Service层的代码如下：

UserServiceImpl.java ,实现用户的CRUD功能进行  
（随书代码  
[MvnBookSSMDemo.Service.Impl\src\main\java\cn\com\mvnbook\ssh\demo\service\impl\UserServiceImpl.java](代码/MvnBookSSMDemo.Service.Impl/src/main/java/cn/com/mvnbook/ssh/demo/service/impl/UserServiceImpl.java)）

package cn.com.mvnbook.ssh.demo.service.Impl;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Propagation;

import org.springframework.transaction.annotation.Transactional;

import cn.com.mvnbook.ssh.demo.dao.IMvnUserDAO;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

import cn.com.mvnbook.ssh.demo.service.IUserService;

//@Configuration

@Service("userService") // 注册成服务组件

@Transactional // 要求启动事务

public class UserServiceImpl implements IUserService {

@Autowired // 自动根据类型注入

//@Qualifier("userDAO") // 根据name注入

private IMvnUserDAO userDAO;

@Transactional(propagation=Propagation.REQUIRED)//需要進行事务管理

public void createUser(MvnUser user) {

// 验证输入的用户是否为null

if (user == null) {

throw new RuntimeException("创建的user不能为null");

}

// 验证用户名是否有存在

MvnUser u = this.userDAO.findUserByUserName(user.getUrUserName());

if (u != null) {

throw new RuntimeException(u.getUrUserName() + " 已经存在");

}

this.userDAO.addUser(user);

}

@Transactional(propagation=Propagation.REQUIRED)//需要進行事务管理

public void editUser(int age, String status, int id) {

// 根据id找到以前的用户对象

MvnUser user = this.userDAO.findUserById(id);

// 判断用户是否存在，不存在抛异常，存在就更新

if (user == null) {

throw new RuntimeException("id为" + id + "用户不存在");

} else {

user.setUrAge(age);

user.setUrStatus(status);

this.userDAO.update(user);

}

}

@Transactional(propagation=Propagation.REQUIRED)//需要進行事务管理

public void deleteUser(int id) {

// 根据id找到以前的用户对象

MvnUser user = this.userDAO.findUserById(id);

// 判断用户是否存在，不存在抛异常，存在就删除

if (user == null) {

throw new RuntimeException("id为" + id + "用户不存在");

} else {

this.userDAO.deleteUser(user);

}

}

@Transactional(readOnly=true)//只读，不需要進行事务管理

public MvnUser searchUser(int id) {

MvnUser user = null;

user = this.userDAO.findUserById(id);

return user;

}

@Transactional(readOnly=true)//只读，不需要進行事务管理

public MvnUser searchUser(String userName) {

MvnUser user = null;

user = this.userDAO.findUserByUserName(userName);

return user;

}

@Transactional(readOnly=true)//只读，不需要進行事务管理

public List<MvnUser> searchUsers() {

List<MvnUser> userList = null;

userList = this.userDAO.findUsers();

return userList;

}

}

第四步：编写测试代码

MvnBookSSMDemo.Service.impl的测试代码有两个

1. TestUserServiceImpl.java  
   （随书代码  
   [MvnBookSSMDemo.Service.Impl\src\test\java\cn\com\mvnbook\ssh\demo\service\impl\impl\TestUserServiceImpl.java](代码/MvnBookSSMDemo.Service.Impl/src/test/java/cn/com/mvnbook/ssh/demo/service/impl/TestUserServiceImpl.java)）

实现测试UserServiceImpl.java的jUnit代码，内容如下：

package cn.com.mvnbook.ssh.demo.service.Impl;

import java.util.List;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

import cn.com.mvnbook.ssh.demo.entity.Status;

import cn.com.mvnbook.ssh.demo.service.IUserService;

import junit.framework.Assert;

public class TestUserServiceImpl {

private IUserService userService;

private ApplicationContext ctx = null;

@Before

public void init(){

this.ctx = new ClassPathXmlApplicationContext("applicationContext.xml");

this.userService = (IUserService)ctx.getBean("userService");

}

@Test

public void testCreateUser(){

MvnUser user = new MvnUser();

user.setUrAge(11);

user.setUrPassword("11");

user.setUrStatus(Status.ACTIVE.getStatus());

user.setUrUserName("service1");

this.userService.createUser(user);

MvnUser u = this.userService.searchUser("service1");

boolean bool = u != null && u.getUrAge()==11 && u.getUrStatus().equals(Status.ACTIVE.getStatus());

Assert.assertTrue(bool);

// 删除用户

this.userService.deleteUser(u.getUrId());

}

@Test

public void testEditUser(){

MvnUser user = new MvnUser();

user.setUrAge(11);

user.setUrPassword("11");

user.setUrStatus(Status.ACTIVE.getStatus());

user.setUrUserName("service1");

this.userService.createUser(user);

MvnUser u = this.userService.searchUser("service1");

this.userService.editUser(88, Status.INACTIVE.getStatus(), u.getUrId());

u = this.userService.searchUser("service1");

Assert.assertTrue(u.getUrAge()==88 &&   
 u.getUrStatus().equals(Status.INACTIVE.getStatus()));

this.userService.deleteUser(u.getUrId());

}

@Test

public void testDeleteUser(){

MvnUser user = new MvnUser();

user.setUrAge(11);

user.setUrPassword("11");

user.setUrStatus(Status.ACTIVE.getStatus());

user.setUrUserName("service1");

this.userService.createUser(user);

MvnUser u = this.userService.searchUser("service1");

this.userService.deleteUser(u.getUrId());

MvnUser u2 = this.userService.searchUser(u.getUrId());

Assert.assertTrue(u != null && u2 == null);

}

@Test

public void testSearchUserById(){

MvnUser user = this.userService.searchUser(1);

Assert.assertNotNull(user);

}

@Test

public void testSearchUserByUserName(){

MvnUser user = this.userService.searchUser("zhangsan");

Assert.assertNotNull(user);

}

@Test

public void testSearchUsers(){

List<MvnUser> userList = this.userService.searchUsers();

Assert.assertTrue(userList != null && userList.size()>0);

}

@After

public void destory(){

this.userService = null;

this.ctx = null;

}

}

1. applicationContext.xml  
   （随书代码  
   [MvnBookSSMDemo.Service.Impl\src\test\resources\applicationContext.xml](代码/MvnBookSSMDemo.Service.Impl/src/test/resources/applicationContext.xml)）

在测试的时候，初始化Spring容器。因为只在测试起作用，所以该文件要同前面的TestUserServiceImpl.java代码一样，放在src/test的对应子目录下，内容如下：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.0.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-4.0.xsd">

<context:component-scan

base-package="cn.com.mvnbook"></context:component-scan>

</beans>

第五步：测试 安装 和 发布

右击工程 点击Run As ->Maven test | install | deploy完成测试 安装 和发布，发布完后，Archiva私服上可以看到MvnBookSSMDemo.Service.Impl的构件

### 6.4.4 实现SpringMVC Web模块

用户CRUD模块的SSM Web层实现，我们采用的是SpringMVC4.x版本，用的是零配置方式实现的，所以理解后面的代码实现，需要有SpringMVC 4.x 的开发基础，建议没有接触过的同学，先参考SpringMVC4.0 注解开发web应用的相关资料。

我们如果有做个其它框架的Web层应用开发的话，可以总结出web层基于框架的开发流程，大体上可以抽象如下几个部分：

1. 在web.xml中配置框架的拦截入口，可能是过滤器，也可能是servlet
2. 开发显示层代码，一般是jsp页面
3. 开发控制层代码，实现接收请求数据，调用service处理请求数据，返回结果view层的标记
4. 通过配置文件，描述框架运行时，对请求的处理代码的对应关系和页面转换流程
5. 其它补充

基于SpringMVC4.x，用零配置方式开发Web层，大体上也是同样的流程。只是因为是零配置方式进行的，也就看不到配置文件相关的代码，但是以前用配置文件描述的信息，肯定还是要描述的，只是用注解或其他方式体现。

对SpringMVC Web的实现思路有基本了解后，接下来我们开始实现SSMDemo的web层功能：

第一步：创建Maven的Web工程

前面大部分工程，都是基于Maven的普通Java工程，这里要基于创建web应用的插件，创建web工程，详细过程如下：

1. 点击Eclipse中的创建Maven Project的选项，如图6-37：

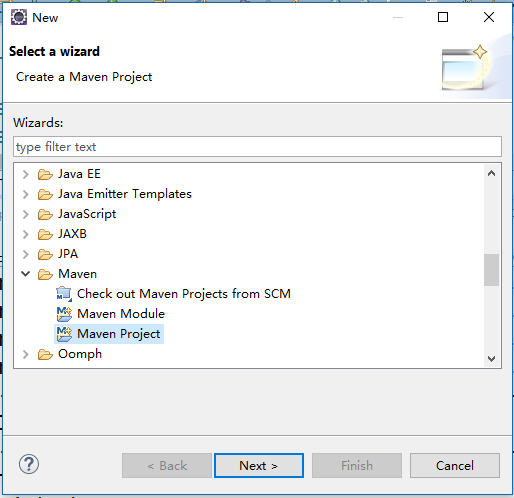


图6-37 创建Maven Project

1. 点击 两个界面的 Next 按钮，进入archetype的选择界面，选择webapp-jee5插件，如图6-38：

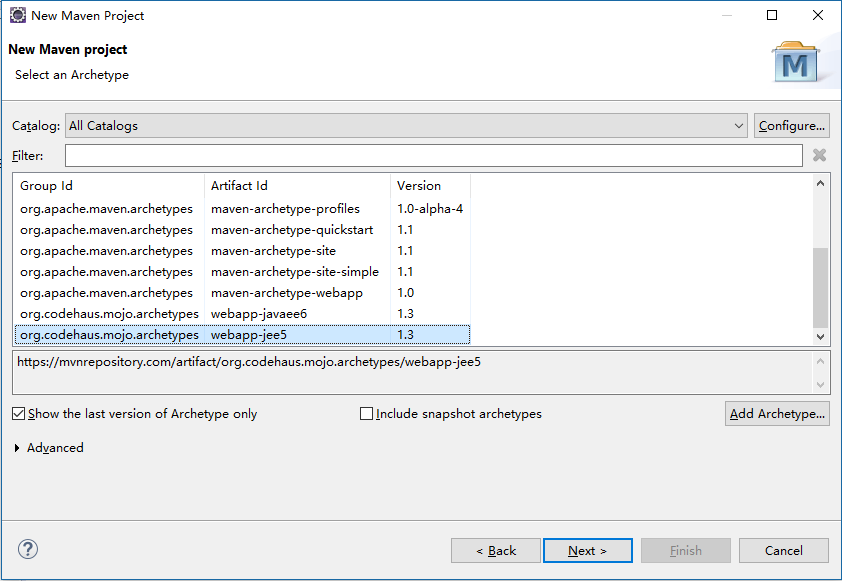


图6-38 选择webapp-jee5

1. 点击 Next按钮，进入类似如下图6-39的界面，在界面中输入工程的groupId、ArtifactId、package，选择一个Version版本。

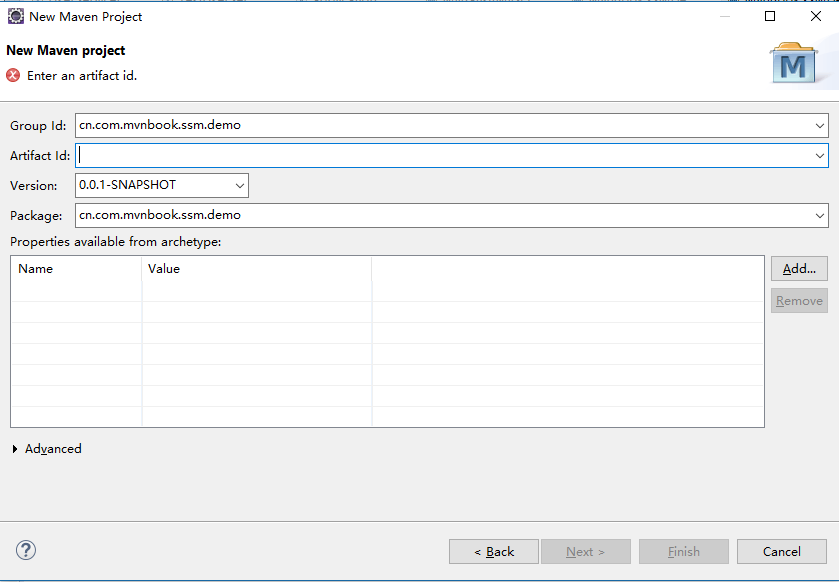


图6-39 输入项目坐标

1. 点击 Finish按钮，创建Web工程，参考目录结构如下图6-40：

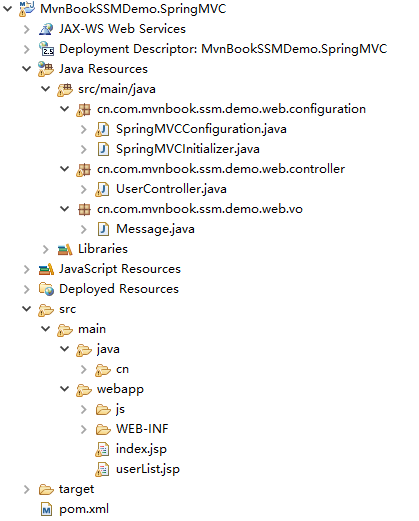


图6-40 MvnBookSSMDemo SpringMVC项目结构

（随书代码[MvnBookSSMDemo.SpringMVC](代码/MvnBookSSMDemo.SpringMVC)）

第二步：编写和完善pom.xml

用户CRUD的模块实现，需要有Spring和Spring web，Spring webmvc的相关jar包，同时还需要我们在前面阶段开发的 MvnBookSSHDAO.DAO和MvnBookSSHService.Service，定义的接口上进行开发，MvnBookSSMDAO.MyBatis和MvnBookSSMService.Service.Impl的实现基础上进行测试，所以pom.xml中，主要体现在对SpringPOM父pom构件的继承和对后面四个构件的依赖。

另外为了发布到Web服务器上运行，需要配置cargo-maven2-plugin插件。

Pom.xml详细内容如下：  
（随书代码[MvnBookSSMDemo.SpringMVC\pom.xml](代码/MvnBookSSMDemo.SpringMVC/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringMVCPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.com.mvnbook.ssm.demo</groupId>

<artifactId>MvnBookSSMDemo.SpringMVC</artifactId>

<packaging>war</packaging>

<name>MvnBookSSMDemo.SpringMVC</name>

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

<dependencies>

<dependency>

<groupId>cn.com.mvnbook.ssh.demo</groupId>

<artifactId>MvnBookSSHDemo.Service</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

<!-- 接口和实现 -->

<dependency>

<groupId>cn.com.mvnbook.ssm.demo</groupId>

<artifactId>MvnBookSSMDemo.Service.Impl</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>cn.com.mvnbook.ssm.demo</groupId>

<artifactId>MvnBookSSMDemo.DAO.MyBatis</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>cn.com.mvnbook.ssh.demo</groupId>

<artifactId>MvnBookSSHDemo.DAO</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

</dependencies>

<build>

<plugins>

<!-- Cargo插件 -->

<plugin>

<!-- 指定插件名称及版本号 -->

<groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>

<version>1.4.8</version>

<configuration>

<wait>true</wait> <!--是否说明，操作start、stop等后续操作必须等前面操作完成才能继续 -->

<container> <!-- 容器的配置 -->

<containerId>tomcat7x</containerId> <!-- 指定tomcat版本 -->

<type>installed</type> <!-- 指定类型：standalone, installed等 -->

<home>C:\java\servers\apache-tomcat-7.0.69\_64</home> <!-- 指定Tomcat的位置，即catalina.home -->

</container>

<configuration> <!-- 具体的配置 -->

<type>existing</type> <!-- 类型，existing:存在 -->

<home>C:\java\servers\apache-tomcat-7.0.69\_64</home> <!-- Tomcat的位置，即catalina.home -->

</configuration>

<deployables> <!-- 部署设置 -->

<deployable> <!-- 部署的War包名等 -->

<groupId>cn.com.mvnbook.ssm.demo</groupId>

<artifactId>MvnBookSSMDemo.SpringMVC</artifactId>

<type>war</type>

<properties>

<context>MvnBookSSMDemo</context> <!-- 部署路径 -->

</properties>

</deployable>

</deployables>

<deployer> <!-- 部署配置 -->

<type>installed</type> <!-- 类型 -->

</deployer>

</configuration>

<executions>

<!-- 执行的动作 -->

<execution>

<id>verify-deployer</id>

<phase>install</phase> <!-- 解析install -->

<goals>

<goal>deployer-deploy</goal>

</goals>

</execution>

<execution>

<id>clean-deployer</id>

<phase>clean</phase>

<goals>

<goal>deployer-undeploy</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

<distributionManagement>

<repository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

</distributionManagement>

</project>

第三步：编写实现代码

SpringMVC Web的代码列表和对应的作用，描述如下：

1. SpringMVCIntializer.java

在Web容器中初始化DispatcherServlet，有Web服务器启动的时候，自己调用初始化，代替以前在web.xml中配置DispatcherServlet的servlet和servlet-mapping

1. SpringMVCConfiguration.java

完成SpringMVC框架运行的基本信息配置，包括视图转换器、类型转换器、多国语言的资源文件、请求路径匹配和不被SpringMVC框架拦截的请求等

1. Message.java

定义VO类，封装页面提示信息，主要是id和message

1. UserController.java

实现用户CRUD的所有控制逻辑代码，包括请求映射注解的描述

1. Index.jsp

CRUD的框架页面，里面包含CRUD的操作按钮和内嵌用户列表的Div，还有每个操作对应的js代码

1. userList.jsp

显示用户列表的页面

1. jquery-3.2.1.min.js

Jquery的內库js，点击CRUD操作的时候，需要实现逻辑js代码，都是基于它来的。该文件放在js目录下，千万不要遗忘。

下面是每个文件的详细内容：

1. SpringMVCInitializer.java  
   （随书代码  
   [MvnBookSSMDemo.SpringMVC\src\main\java\cn\com\mvnbook\ssm\demo\web\configuration\SpringMVCInitializer.java](代码/MvnBookSSMDemo.SpringMVC/src/main/java/cn/com/mvnbook/ssm/demo/web/configuration/SpringMVCInitializer.java)）

package cn.com.mvnbook.ssm.demo.web.configuration;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

// 继承初始化父类，让web服务器能自动识别，初始化SpringMVC Servlet入口组件

// 代替在web.xml中配置org.springframework.web.servlet.DispatcherServlet

public class SpringMVCInitializer extends

AbstractAnnotationConfigDispatcherServletInitializer {

@Override

protected Class<?>[] getRootConfigClasses() {

// 加载SpringMVC的基本配置类

return new Class[] { SpringMVCConfiguration.class };

}

@Override

protected Class<?>[] getServletConfigClasses() {

return null;

}

@Override

protected String[] getServletMappings() {

// 指定SpringMVC的mapping url-pattern

return new String[] { "/" };

}

}

1. SpringMVCConfiguration.java  
   （随书代码  
   [MvnBookSSMDemo.SpringMVC\src\main\java\cn\com\mvnbook\ssm\demo\web\configuration\SpringMVCConfiguration.java](代码/MvnBookSSMDemo.SpringMVC/src/main/java/cn/com/mvnbook/ssm/demo/web/configuration/SpringMVCConfiguration.java)）

package cn.com.mvnbook.ssm.demo.web.configuration;

import java.util.ArrayList;

import java.util.List;

import org.springframework.context.MessageSource;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.support.ResourceBundleMessageSource;

import org.springframework.format.FormatterRegistry;

import org.springframework.http.converter.json.MappingJackson2HttpMessageConverter;

import org.springframework.web.servlet.config.annotation.EnableWebMvc;

import org.springframework.web.servlet.config.annotation.PathMatchConfigurer;

import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;

import org.springframework.web.servlet.config.annotation.ViewResolverRegistry;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

import org.springframework.web.servlet.view.InternalResourceViewResolver;

import org.springframework.web.servlet.view.JstlView;

@Configuration

@EnableWebMvc// 支持SpringMVC

@ComponentScan(basePackages = "cn.com.mvnbook")// SpringMVC自动扫描组件类的基本包

public class SpringMVCConfiguration extends WebMvcConfigurerAdapter{

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

// 注册视图转换器

InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();

viewResolver.setViewClass(JstlView.class);

viewResolver.setPrefix("/");

viewResolver.setSuffix(".jsp");

registry.viewResolver(viewResolver);

}

/\*

\* Configure ResourceHandlers to serve static resources like CSS/ Javascript etc...

\*

\*/

@Override

public void addResourceHandlers(ResourceHandlerRegistry registry) {

// 设置不需要经过SpringMVC框架过滤的资源，比如js css 和 images等

registry.addResourceHandler("/static/\*\*")

.addResourceLocations("/static/");

registry.addResourceHandler("/js/\*\*")

.addResourceLocations("/js/");

}

/\*

\* Configure Converter to be used.

\* In our example, we need a converter to convert string values[Roles] to UserProfiles in newUser.jsp

\*/

@Override

public void addFormatters(FormatterRegistry registry) {

//registry.addConverter(roleToUserProfileConverter);

}

/\*\*

\* Configure MessageSource to lookup any validation/error message in internationalized property files

\*/

@Bean

public MessageSource messageSource() {

// 添加资源文件（多国语言）

ResourceBundleMessageSource messageSource =   
 new ResourceBundleMessageSource();

messageSource.setBasename("messages");

return messageSource;

}

/\*\*Optional. It's only required when handling '.' in @PathVariables which otherwise ignore everything after last '.' in @PathVaidables argument.

\* It's a known bug in Spring [https://jira.spring.io/browse/SPR-6164], still present in Spring 4.1.7.

\* This is a workaround for this issue.

\*/

@Override

public void configurePathMatch(PathMatchConfigurer matcher) {

matcher.setUseRegisteredSuffixPatternMatch(true);

}

}

1. Message.java  
   （随书代码  
   [MvnBookSSMDemo.SpringMVC\src\main\java\cn\com\mvnbook\ssm\demo\web\vo\Message.java](代码/MvnBookSSMDemo.SpringMVC/src/main/java/cn/com/mvnbook/ssm/demo/web/vo/Message.java)）

package cn.com.mvnbook.ssm.demo.web.vo;

public class Message {

private String code;

private String message;

public Message(String message) {

super();

this.message = message;

}

public String getCode() {

return code;

}

public void setCode(String code) {

this.code = code;

}

public String getMessage() {

return message;

}

public void setMessage(String message) {

this.message = message;

}

}

1. UserController.java  
   （随书代码  
   [MvnBookSSMDemo.SpringMVC\src\main\java\cn\com\mvnbook\ssm\demo\web\controller\UserController.java](代码/MvnBookSSMDemo.SpringMVC/src/main/java/cn/com/mvnbook/ssm/demo/web/controller/UserController.java)）

package cn.com.mvnbook.ssm.demo.web.controller;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.ResponseBody;

import org.springframework.web.servlet.ModelAndView;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

import cn.com.mvnbook.ssh.demo.service.IUserService;

import cn.com.mvnbook.ssm.demo.web.vo.Message;

@Controller("userController")

@RequestMapping("/user")

public class UserController {

@Autowired

@Qualifier("userService")

private IUserService userService;

@RequestMapping("/add")

@ResponseBody

public Message addUser(MvnUser user) {

Message msg = new Message("添加成功");

try {

this.userService.createUser(user);

} catch (Exception e) {

e.printStackTrace();

msg.setMessage("添加失败:" + e.getMessage());

}

return msg;

}

@RequestMapping("/delete")

@ResponseBody

public Message deleteUser(int id) {

Message msg = new Message("删除成功");

try {

this.userService.deleteUser(id);

} catch (Exception e) {

e.printStackTrace();

msg.setMessage("删除失败:" + e.getMessage());

}

return msg;

}

@RequestMapping("/edit")

@ResponseBody

public Message editUser(@RequestParam("status") String status, @RequestParam("age") int age,@RequestParam("id") int id) {

Message msg = new Message("修改成功");

try {

this.userService.editUser(age, status, id);

} catch (Exception e) {

e.printStackTrace();

msg.setMessage("修改失败:" + e.getMessage());

}

return msg;

}

@RequestMapping("/searchUser")

@ResponseBody

public MvnUser searchUser(@RequestParam("id") int id) {

MvnUser user = this.userService.searchUser(id);

return user;

}

@RequestMapping("/searchUsers")

public ModelAndView searchUsers() {

ModelAndView mv = new ModelAndView();

mv.setViewName("userList");

mv.addObject("userList", this.userService.searchUsers());

return mv;

}

}

1. Index.jsp  
   （随书代码  
   [MvnBookSSMDemo.SpringMVC\src\main\webapp\index.jsp](代码/MvnBookSSMDemo.SpringMVC/src/main/webapp/index.jsp)）

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">

<title>测试用户CRUD操作</title>

<script type="text/javascript" src="js/jquery-1.8.2.min.js"></script>

<script type="text/javascript">

$(document).ready(function() {

// 页面加载完后，自动发searchUsersServlet请求，加载到userListDiv中显示

$("#userListDiv").load("user/searchUsers");

});

// 新增 按钮事件触发函数

function toAdd() {

// 获取addForm中的请求信息

var \_data = $("#addForm").serialize();

//alert(\_data);

// 发添加新用户的Ajax请求

$.ajax({

type : 'post',

url : 'user/add',

data : \_data,

dataType:'json',// 指定返回数据的格式：json

success : function(msg) {

alert(msg.message);

// 更新最新的用户列表信息

$("#userListDiv").load("user/searchUsers");

}

});

}

function toEdit() {

var \_data = $("#editForm").serialize();

alert(\_data);

$.ajax({

type : 'post',

url : 'user/edit',

data : \_data,

dataType:'json',

success : function(msg) {

alert(msg.message);

$("#userListDiv").load("user/searchUsers");

}

});

}

function toDelete() {

var chks = $("input[name='checkbox']:checked");

if (chks.length == 0) {

alert("请选择要删除的用户");

} else if (chks.length > 1) {

alert("一次只能删除一个用户");

} else {

var to = confirm("您确定要删除选中的用户?");

if (to) {

var \_data = "id=" + chks.val();

$.ajax({

type : 'post',

data : \_data,

url : 'user/delete',

dataType:'json',

success : function(msg) {

alert(msg);

$("#userListDiv").load("user/searchUsers");

}

});

}

}

}

function toShowAdd() {

$("#LayerAdd").show(1000);

}

function toShowEdit() {

//alert($("input[name='checkbox']:checked").length);

var chks = $("input[name='checkbox']:checked");

if (chks.length == 0) {

alert("请选择要编辑的用户");

} else if (chks.length > 1) {

alert("一次只能修改一个用户");

} else {

var \_data = "id=" + chks.val();

$.ajax({

type : 'post',

data : \_data,

url : 'user/searchUser',

dataType:'json',

//contentType:'application/json;charset=UTF-8',

success : function(msg) {

$("#editForm #id").val(msg.urId);

$("#editForm #userName").val(msg.urUserName);

$("#editForm #age").val(msg.urAge);

$("#editForm #status").val(msg.urStatus);

//alert($("#editForm #age").val());

$("#LayerEdit").show(1000);

}

});

}

}

function toCloseAdd() {

$("#LayerAdd").hide(1000);

}

function toCloseEdit() {

$("#LayerEdit").hide(1000);

}

</script>

<style type="text/css">

<!--

.STYLE2 {

color: #000000

}

#LayerAdd {

position: absolute;

left: 113px;

top: 183px;

width: 434px;

height: 193px;

z-index: 1;

background-color: #99FFFF;

display: none;

}

#LayerEdit {

position: absolute;

left: 113px;

top: 183px;

width: 434px;

height: 193px;

z-index: 1;

background-color: #99FFFF;

display: none;

}

-->

</style>

</head>

<body>

<div id="LayerAdd">

<form name="addForm" name="addForm" id="addForm" method="post"

action="">

<table width="98%" border="0" align="center" cellpadding="0"

cellspacing="0">

<tr>

<td colspan="2" align="center"><strong><BR>添加新用户<br></strong></td>

</tr>

<tr>

<td width="47%" align="right">用户名：</td>

<td width="53%"><input name="urUserName" type="text"

id="userName"></td>

</tr>

<tr>

<td align="right">密码：</td>

<td><input name="urPassword" type="password" id="password"></td>

</tr>

<tr>

<td align="right">年龄：</td>

<td><input name="urAge" type="text" id="age"></td>

</tr>

<tr>

<td colspan="2">&nbsp;</td>

</tr>

<tr>

<td colspan="2" align="center"><input type="button"

name="Submit4" value="添加" onclick="toAdd()"> <input

type="button" name="Submit5" value="关闭" onclick="toCloseAdd()"></td>

</tr>

</table>

</form>

</div>

<div id="LayerEdit">

<form name="editForm" id="editForm" method="post" action="">

<input type="hidden" name="id" id="id" />

<table width="98%" border="0" align="center" cellpadding="0"

cellspacing="0">

<tr>

<td colspan="2" align="center"><strong><br>

修改用户信息<br> </strong></td>

</tr>

<tr>

<td width="47%" align="right">用户名：</td>

<td width="53%"><input name="userName" type="text"

id="userName" readonly="readonly"></td>

</tr>

<tr>

<td align="right">年龄：</td>

<td><input name="age" type="text" id="age"></td>

</tr>

<tr>

<td align="right">状态：</td>

<td><select name="status" id="status">

<option value="Active">Active</option>

<option value="Inactive">Inactive</option>

<option value="Locked">Locked</option>

<option value="Deleted">Deleted</option>

</select></td>

</tr>

<tr>

<td colspan="2">&nbsp;</td>

</tr>

<tr>

<td colspan="2" align="center"><input type="button"

name="Submit4" value="修改" onclick="toEdit()"> <input

type="button" name="Submit5" value="关闭" onclick="toCloseEdit()"></td>

</tr>

</table>

</form>

</div>

<p>&nbsp;</p>

<p>测试用户CRUD页面</p>

<table width="539" border="1">

<tr>

<td colspan="5" align="right"><input type="button" name="Submit"

value="新增" onclick="toShowAdd()"> <input type="submit"

name="Submit2" value="修改" onclick="toShowEdit()"> <input

type="button" name="Submit3" value="删除" onclick="toDelete()"></td>

</tr>

<tr>

<td>

<div id="userListDiv"></div>

</td>

</tr>

</table>

<p>&nbsp;</p>

</body>

</html>

1. userList.jsp  
   （随书代码  
   [MvnBookSSMDemo.SpringMVC\src\main\webapp\userList.jsp](代码/MvnBookSSMDemo.SpringMVC/src/main/webapp/userList.jsp)）

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<table width="100%" border="1">

<tr>

<td width="51" bgcolor="#CCCCCC"><div align="center"><strong><span class="STYLE2">选择</span></strong></div></td>

<td width="111" bgcolor="#CCCCCC"><div align="center"><strong>序号</strong></div></td>

<td width="137" bgcolor="#CCCCCC"><div align="center"><strong>用户名</strong></div></td>

<td width="105" bgcolor="#CCCCCC"><div align="center"><strong>年龄</strong></div></td>

<td width="101" bgcolor="#CCCCCC"><div align="center"><strong>状态</strong></div></td>

</tr>

<c:forEach items="${userList }" var="\_user" varStatus="status">

<c:choose>

<c:when test="${status.index%2==0 }">

<tr bgcolor="#FFFFFF">

</c:when>

<c:otherwise>

<tr bgcolor="#99FFFF">

</c:otherwise>

</c:choose>

<td><input type="checkbox" name="checkbox" value="${\_user.urId }"></td>

<td>${status.index+1 }</td>

<td>${\_user.urUserName }</td>

<td>${\_user.urAge }</td>

<td>${\_user.urStatus }</td>

</tr>

</c:forEach>

</table>

第四步：编译 测试 安装 发布 和 启动服务器

编译、测试、安装和发布，从Maven角度来说，同前面模块项目的操作和目的一样。

编译：调用编译器编译Java代码

测试：基于jUnit运行测试案例，输出测试结果报告

安装：将工程编译后的结果，打成对应的包（pom|jar|war），并且形成构件，安装到本地仓库中。

发布：将生成的构件，发布到局域网里面的私服上或远程仓库。

具体操作都是，右击工程 Run As->Maven test|install|deploy

发布好后，在Tomcat的webapps发布目录下，会有MvnBookSSMDemo.war文件，启动Tomcat，就可以自动发布成web应用。

详细的浏览器操作效果，我们整合在一起后，最后说明。

### 6.4.5 整合SSM

前面我们以分模块的方式，实现了每个功能，包括 公共DAO、Service接口的定义，基于MyBatis的DAO实现，Service的独立实现，还有基于SpringMVC的web层实现，并且能够集成到SpringMVC的web层代码中，共同完成测试。

但是为了方便管理，接下来我们有必要创建一个工程，将前面独立实现的各个模块管理起来，这样每次编译、测试、安装和发布的时候，都能基于Maven实现自动同步。

整合SSM模块的具体步骤如下：

第一步：创建工程

只需要创建一个简单的java工程，按当初自己的设计，输入groupId和artifactId和版本。我们的demo分别是cn.com.mvnbook.ssm.demo, MvnBookSSMDemo和0.0.1-SNAPSHOT。  
（随书代码[MvnBookSSMDemo](代码/MvnBookSSMDemo)）

第二步：配置pom.xml

打开pom.xml中，在里面添加包含的模块，并且设置packaging为pom。

我们前面介绍的是基于编辑器直接编写pom.xml，这里介绍基于Eclipse的图形化界面，基于向导界面修改pom.xml（添加模块）  
（随书代码[MvnBookSSMDemo\pom.xml](代码/MvnBookSSMDemo/pom.xml)）

1. 双击 pom.xml文件，我们可以通过点击不同标签，选择打开方式。如图6-41：



图6-41 pom.xml

1. 如果选择的是pom.xml，显示的是pom.xml源文件，我们可以按以前的方式，基于源代码修改pom.xml。现在我们选择Overview标签，显示界面如下图6-42：

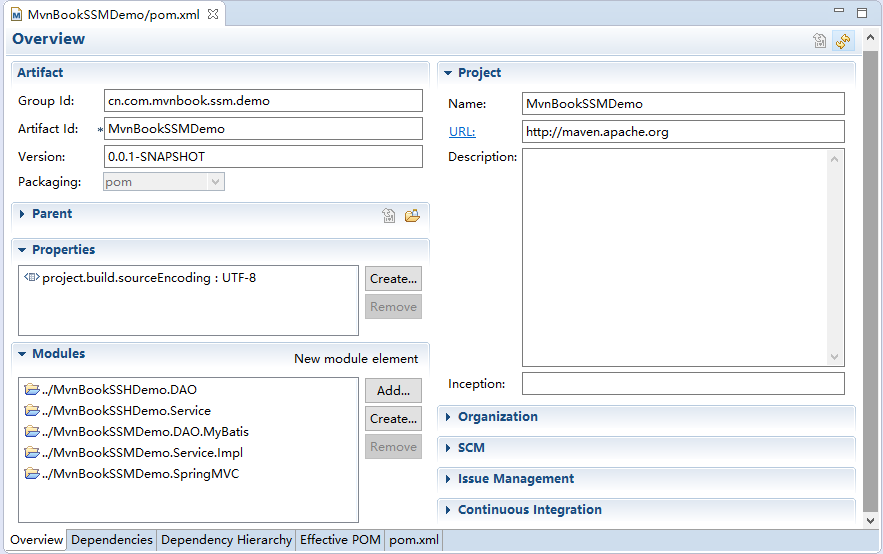


图6-42 可视化pom.xml

1. 点击 Modules 范围内的 Add按钮，就可以通过图形界面，选择需要添加的模块。保存就完成了模块的集成。

第三步：编译、测试、安装 和 发布

这个步骤，同前面SSH集成后操作的步骤一样，请参考SSH集成里面最后的对应步骤。

## 6.5 小结

到现在，我们基本上亲自经历了常用的两个框架SSH和SSM的开发过程，回看开发过程，它们都有共同的规律和特点：

1. 分析清楚项目层次结构。

我们这里比较简单，就分web层、service层和dao层 三层结构。在实际项目中，可以把这三层交叉分开，比如按功能分不同模块。项目越大，分得越多。但是有个总原则：每个模块之间尽量解耦。

1. 定义好框架的公共依赖pom

一般是将常用的每个框架和所依赖的构件，封装到一个pom工程中，供其他的项目继承或引用。

1. 定义好每个层次的接口和公共类
2. 针对每个层次的接口进行实现

因为是需要测试的，所以在测试的时候，需要引用其它模块接口的实现，不过可以将这些依赖的scope设置成test，表示只是在测试的时候引入，打包发布的时候不用考虑（系统总集成的时候，再根据具体情况，有选择的总体集成起来）

1. 创建一个整合工程，将所有模块集成起来，统一管理测试

将所有的接口和具体的实现模块，集成在一起

# 第七章 生成项目站点

Maven是一个自动化构建工具，也是一个公共依赖管理工具，这两点，我们在前面的样例项目开发过程中，已经有亲自的体验。Maven除了是构件工具和依赖工具外，它还能帮助集中管理项目相关的信息，促进项目团队之间的交流。

在pom中包含了项目的各种信息，比如项目描述、版本控制系统信息、许可证信息、开发者信息等等，我们可以用Maven，将这些信息生成成web页面，然后将这些web页面发不到公司的服务器上，让团队成员随时了解到最新的项目信息。同时，我们还可以在pom中动态的添加插件，生成包括JavaDoc文档、测试报告、测试覆盖率报告、代码规范报告等等，集成到web站点中，统一发布。

这章主要介绍怎么样基于Maven自动生成项目站点。

## 7.1 生成基本站点

### 7.1.1 简单站点

在MvnBookSSMDemo项目的pom.xml中，添加站点插件(maven-site-plugin)的配置，内容如下：  
（随书代码[MvnBookSSMDemo\pom.xml](代码/MvnBookSSMDemo/pom.xml)）

<build>

<pluginManagement>

<plugins>

<plugin>

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

<artifactId>maven-site-plugin</artifactId>

<version>3.4</version>

</plugin>

</plugins>

</pluginManagement>

</build>

注意：这块内容需要直接放在project元素里面

右击工程 Run As ->Maven build...，在弹出的配置窗口的Goals后面，输入site，表示执行创建站点命令，点击 run 按钮，在控制窗口中会显示如下内容，表示生成成功。

[INFO] Relativizing decoration links with respect to project URL: http://maven.apache.org

[INFO] Rendering site with org.apache.maven.skins:maven-default-skin:jar:1.0 skin.

[INFO] Generating "Dependency Convergence" report --- maven-project-info-reports-plugin:2.9:dependency-convergence

[INFO] Generating "Dependency Information" report --- maven-project-info-reports-plugin:2.9:dependency-info

[INFO] Generating "Distribution Management" report --- maven-project-info-reports-plugin:2.9:distribution-management

[INFO] Generating "About" report --- maven-project-info-reports-plugin:2.9:index

[INFO] Generating "Project Modules" report --- maven-project-info-reports-plugin:2.9:modules

[INFO] Generating "Plugin Management" report --- maven-project-info-reports-plugin:2.9:plugin-management

[INFO] Generating "Plugins" report --- maven-project-info-reports-plugin:2.9:plugins

[INFO] Generating "Summary" report --- maven-project-info-reports-plugin:2.9:summary

[INFO] ------------------------------------------------------------------------

[INFO] Reactor Summary:

[INFO]

[INFO] MvnBookSSHDemo.DAO ................................. SUCCESS [02:15 min]

[INFO] MvnBookSSHDemo.Service ............................. SUCCESS [ 5.350 s]

[INFO] MvnBookSSMDemo.DAO.MyBatis ......................... SUCCESS [ 29.677 s]

[INFO] MvnBookSSMDemo.Service.Impl ........................ SUCCESS [ 13.300 s]

[INFO] MvnBookSSMDemo.SpringMVC ........................... SUCCESS [ 25.718 s]

[INFO] MvnBookSSMDemo ..................................... SUCCESS [01:32 min]

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 05:02 min

[INFO] Finished at: 2016-11-25T11:01:23+08:00

[INFO] Final Memory: 32M/314M

[INFO] ------------------------------------------------------------------------

Maven运行完后，我们可以在项目的target/site目录下，找到Maven生成的站点文件，包括index.html，project-info.html，modules.html, plugin-management.html，plugins.html等文件，还有css和images两个目录。

使用浏览器打开index.html页面，我们看到如下图7-1页面：

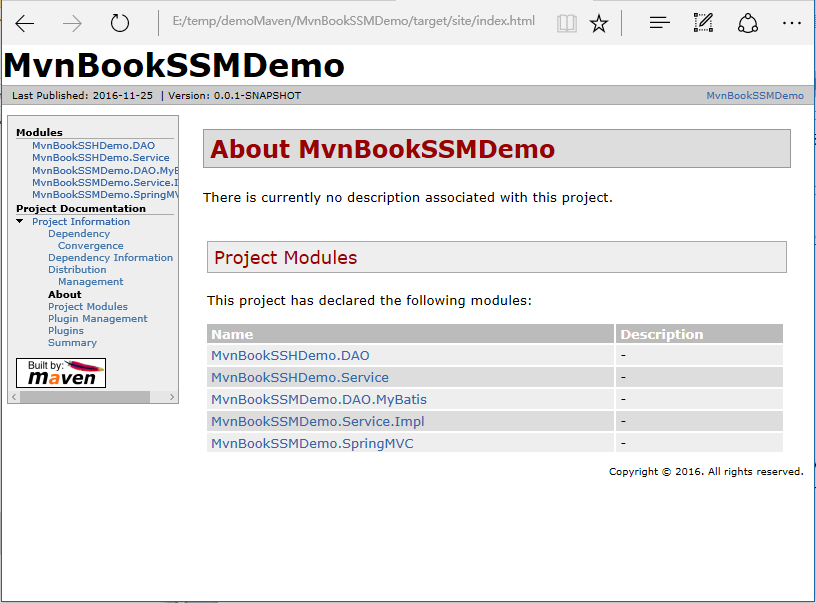


图7-1 MvnBookSSMDemo 站点首页

从上图可以看出，左边导航栏包含项目的各类信息链接，包括项目简介、项目模块、项目管理的插件、项目用到的依赖以及保护的子模块等信息。

当然，如果项目同本书的案例一样，是一个聚合项目（包含多个模块）的话，点击modules中的连接是找不到子页面的。这是因为每个模块的报告，都生成在自己目录的target/site目录下，每个模块的物理位置是独立的，我们只要把它们都发布到站点服务器里面，它们放在一起，使用的是相对路径访问，就可以连接到了。

如果暂时没有搭建web站点服务器，又想看到子模块的连接效果的话，我们可以在生成站点文件的时候，指定输出到一个目录，这样它们就可以相互连接上。具体操作是：

右击工程 Run As -> Maven build... ，在弹出的配置窗口的Goals中，输入 site:stage -DstagingDirectory=e:\temp\site

这样，就可以将站点文件生成到指定目录。这样有点麻烦，就是要每个模板独立生成站点信息，并且按相对路径，生成到不同的目录下，才能准确的连接上。

比如总工程MvnBookSSMDemo的目标目录是e:\temp\site，那它里面的MvnBookSSHDemo

.DAO模块的目录应该是 e:\temp\site\MvnBookSSHDemo.DAO，而MvnBookSSMDemo.DAO.MyBatis的目录应该是e:\temp\site\MvnBookSSMDemo.DAO.MyBatis，其它的模块目录按同样规律指定目标目录，最后才能连接成一个总体。

### 7.1.2 完善站点信息

在Maven3中，生成站点信息是由maven-site-plugin插件自动完成的，至于站点里面的那些项目项的连接，那是由一个maven-project-info-reports-plugin的插件负责产生的，该插件内置在maven-site-plugin中。所以我们不用做太多的配置，就能生成大量的项目信息。接下来我们介绍一下这些信息报告：

1. about（关于）：项目的一些整体描述信息
2. Continuous Integration（持续集成）:项目持续集成服务器信息
3. Dependencies（依赖）:项目依赖信息，包括传递依赖、依赖图、依赖许可证以及依赖文件的大小、所包含的类的数目等等。
4. Depedency Convergence（依赖收敛）:只针对包含有多个模块的项目生成，提供一些依赖健康状态的分析信息，比如各模块使用的依赖版本是否一致、项目中是否有SNAPSHOT依赖等。
5. Dependency Management（依赖管理）:基于项目的依赖管理配置生成的报告信息
6. Issue Tracking（问题跟踪）:项目bug跟踪系统的信息
7. Mailing Lissts（邮件列表）:项目邮件列表信息
8. Plugin Management（插件管理）:项目使用到的插件列表信息
9. Project License（项目许可证）:项目许可证信息
10. Project Summary（项目概述）:项目的坐标、名称、描述、版本等信息
11. Project Team（项目团队）:项目团队信息
12. Source Repository（源代码仓库）:项目的源代码仓库信息。

上面列的信息，都是根据自己项目的实际情况和pom中的已有配置描述生成的，Maven不会自己凭空产生。也就是说，我们虽然在上面列出了十二项信息，但是我们未必能在自己的站点文件中看到所有信息，因为有些信息，我们在pom文件中没有配置。

比如我们有如下pom.xml  
（随书代码[MvnBookSSMDemo\pom.xml](代码/MvnBookSSMDemo/pom.xml)）

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.com.mvnbook.ssm.demo</groupId>

<artifactId>MvnBookSSMDemo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<name>MvnBookSSMDemo</name>

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

... ...

<organization>

<name>Organization</name>

<url>http://www.mvnbook.com</url>

</organization>

<issueManagement>

<system>admin</system>

<url>http://www.mvnbook.com/issue</url>

</issueManagement>

<ciManagement>

<system>admin</system>

<url>http://www.mvnbook.com/continuous</url>

</ciManagement>

<description>这是MvnBook的SSM Demo集成项目</description>

</project>

Maven-site-plugin就会根据ciManagement的配置，生成持续集成服务器信息；根据issueManagement的配置，生成bug跟踪系统的信息；根据description元素配置，生成项目的描述信息；根据organization元素的配置，生成组织机构信息等等。

其它情况，我们在这里就不详细介绍了，后面有相关的章节和附录，会全面介绍pom.xml的元素，请参考后面的章节和最后的附录

## 7.2 添加插件丰富站点信息

除了前面我们介绍的项目信息报告为，Maven还提供了大量的报告插件，这些插件有Maven社区的程序员和团体免费提供，我们只要稍稍的配置，就可以让Maven自动的帮我们生成相关的丰富报告。接下来我们介绍写比较常用的插件。稍微要注意一下的是，这些生成报告的插件，在pom中配置的位置和方式，同我们以前用到的插件有点不一样，我们以前的插件是配置在project->builder->plugins中的，现在的报告插件，需要配置在project->reporting->plugins中。

### 7.2.1 JavaDoc插件

我们对JavaDocs应该是比较熟悉的，开始学Java不久，我们就需要会查看JDK的Docs文档，以便查找和理解常用类和方法，从而在自己代码中使用。这里，我们通过配置maven-javadoc-plugin插件，调用JDK的javadoc工具，基于项目的源代码生成JavaDocs文档。建议在Maven远程仓库中找到最新的javadoc插件版本，配置到我们的pom中去。因为我们现在开发的项目，基本上是模块式开发，最后用一个聚合项目，把各个模块合在一起。这样生成javadoc文档的时候，希望直接在聚合项目上执行maven命令，将包含的各个模块的所有javadoc文档都生成，不想每个模块一个个的运行maven命令生成文档后再手动合并，相信大家都会打成这样的一致共识。maven-javadoc-plugin插件的最新版本，就帮我们实现了这样的目的：只要在聚合工程上运行插件，能自动的将包含的所有模块的javadoc文档都生成出来。

在浏览器中输入http://mvnrepository.com/,search窗口中输入maven-javadoc-plugin,我们就可以找到maven-javadoc-plugin插件的所有版本，我们这里选择的是2.10.3版本，在pom.xml中的project标签下，添加如下内容，完成maven-javadoc-plugin的配置  
（随书代码[MvnBookSSMDemo\pom.xml](代码/MvnBookSSMDemo/pom.xml)）

<reporting>

<plugins>

<plugin>

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

<artifactId>maven-javadoc-plugin</artifactId>

<version>2.10.3</version>

</plugin>

</plugins>

</reporting>

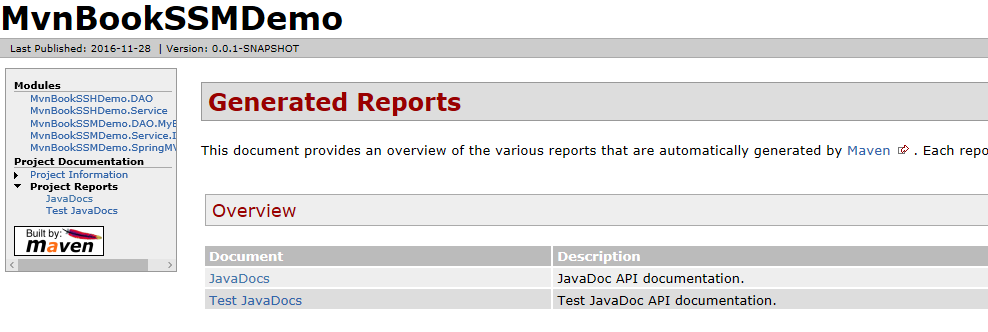
右键MvnBookSSMDemo工程，选择Run As->maven build...，在Goals后面输入site，点击Run按钮，Maven在生成站点的时候，自动调用maven-javadoc-plugin插件，生成JavaDocs文档，该文档位置在MvnBookSSMDemo工程的target/site下，名称为apidocs的目录，里面的内容就是javadocs的文档网页。打开site目录下的index.html页面，里面的左边导航，会多一个Project Resports导航链接，如下图7-2：  


图7-2 javadoc报告

展开Project Reports导航链接，里面就有JavaDocs和Test JavaDocs两部分内容。JavaDocs是我们项目源代码的文档，Test JavaDocs是我们项目中，测试代码的doc文档。如下是我们MvnBookSSMDemo项目的JavaDocs文档的样例图7-3：

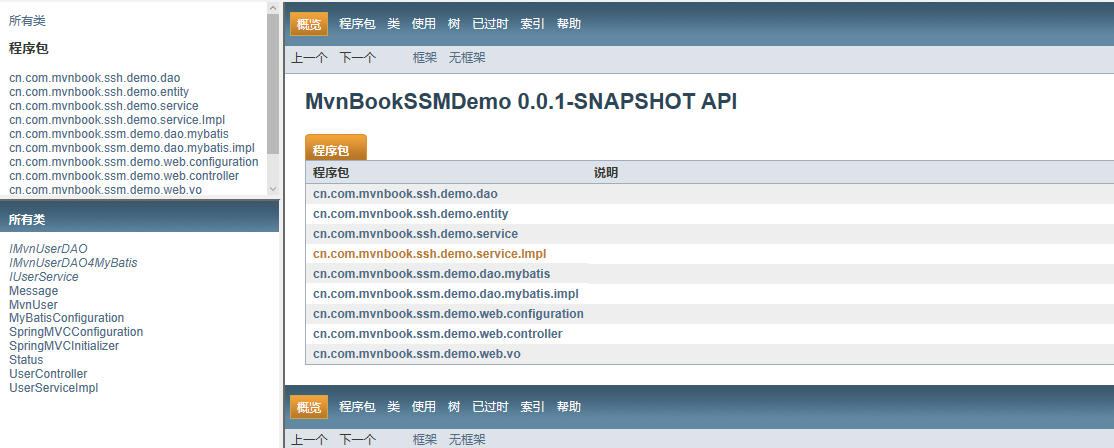


图7-3 javaDocs文档

### 7.2.2 源代码插件

前面我们介绍了根据源代码，利用JavaDoc工具生成JavaDocs文档的插件，能帮我们自动的生成项目的帮助文档。这样其他开发人员，就可以通过版主文档，了解我们有哪些代码，每个代码有哪些方法，以及每个方法的功能、参数、返回等信息，从而了解怎么调用。但是，如果我们能用查看帮助文档的方式，查看到源代码的话，这样对源代码的阅读和理解、包括发现缺陷和bug是很有帮助的。接下来我们介绍一下maven-jxr-plugin插件，该插件可以实现源代码的生成。

首先，同查找maven-javadoc-plugin的方式一样，在http://mvnrepository.com里面找到maven-jxr-plugin合适的版本，我们这里使用的是2.4版本。

接着在pom.xml中，project->reporting->plugins里面，添加maven-jxr-plugin插件的依赖，相关源代码如下：  
（随书代码[MvnBookSSMDemo\pom.xml](代码/MvnBookSSMDemo/pom.xml)）

<plugin>

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

<artifactId>maven-jxr-plugin</artifactId>

<version>2.4</version>

</plugin>

右击工程，选择Run As->Maven build ...，在Goals后面输入site，如果前面有通过build...配置过site的话，直接Maven build，选择配置过的site执行。完成后，我们就可以在site的index.html的project reports导航链接中，发现有Source Xref和Test Source Xref子链接，如下图7-4：

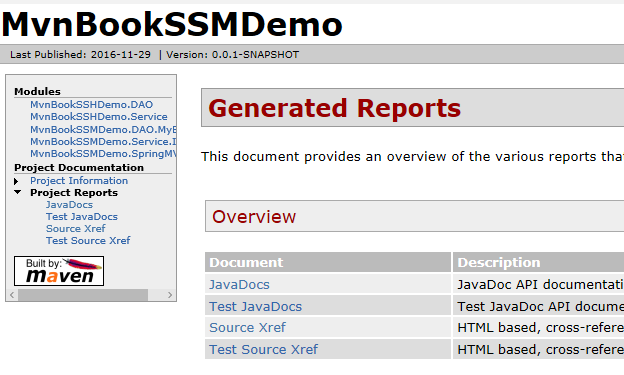


图7-4 源码报告

点击Source Xref进去，我们就可以以网页的形式，查看所有类和接口的源代码。如下图7-5，就是查看的cn.com.mvnbook.ssh.demo.dao.IMvnUserDAO.java的源代码。



图7-5 IMvnUserDAO源码文档

有意思的是，每个源代码页面的最顶上，有个View Javadoc链接，点击该链接，我们可以直接跳到基于maven-javadoc-plugin插件生成的JavaDocs文档上面去，友情实现了源代码和对应JavaDocs文档的直接链接。

### 7.2.3测试报告插件

在前面的每个案例中，处理编写正常的项目源代码外，基本上都有基于jUnit框架，编写了对应的测试案例代码，并且通过运行mvn test命令，来运行测试案例代码进行测试（前期是直接使用的mvn test命令，后期基本上是右击工程，选择Run As->Maven test运行的）。其实这个过程的内部，是由Maven自动调用maven-surefire-plugin插件，由maven-surefire-plugin插件启动jUnit单元测试框架，运行的测试案例代码。运行完后，在工程的target目录下，自动生成了surefire-reports目录，里面有当前测试的测试报告。报告的文件有两个，一个是txt文件，还一个是xml文件，如我们有在MvnBookSSH.service.impl工程中写过的TestUserServiceImpl.java测试案例，运行后的测试报告就是cn.com.mvnbook.ssh.demo.service.Impl.TestUserServiceImpl.txt和TEST-cn.com.mvnbook.ssh.demo.service.Impl.TestUserServiceImpl.xml。TestUserServiceImpl.txt内容在下面有罗列，TestUserServiceImpl.xml的内容就不罗列了，大家可以查看自己工程中运行测试后对应的代码，主要描述的是测试相关的参数信息。

TestUserServiceImp.xml内容：

-------------------------------------------------------------------------------

Test set: cn.com.mvnbook.ssh.demo.service.Impl.TestUserServiceImpl

-------------------------------------------------------------------------------

Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 9.18 sec

通过查看maven-surefire-plugin插件生成的测试报告，对比我们前面介绍的JavaDocs和源代码报告，是不是感觉太弱了，如果能将项目中的每个测试报告，以同样web网页的形式，集成到站点报告中去，是不是更好更统一了呢。接下来我们介绍怎么样将maven-surefire-plugin插件生成的报告，集成到项目站点中去。

首先要说明一下，maven-surefire-plugin插件已经内置在Maven中，运行mvn test命令的时候，Maven会自动的寻找最新版本的插件，启动jUnit框架，运行对应的测试案例，并且生成txt版本的测试报告。

要将测试结果形成web页面形式的报告，并且集成到站点中去的话，我们可以使用cobertura插件（cobertura-maven-plugin）。

同样的方式，首先我们在http://mvnrepository.com站点中找到中意的版本，我们这里使用的是2.6。在project->reporting->plugins中，添加cobertura-maven-plugin的坐标，代码如下：  
（随书代码[MvnBookSSMDemo.Service.Impl\pom.xml](代码/MvnBookSSMDemo.Service.Impl/pom.xml)）

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>cobertura-maven-plugin</artifactId>

<version>2.6</version>

</plugin>

右击工程，选择Run As->Maven build... ，在Goals后面输入site，点击运行，就会生成测试代码覆盖率报告，在站点文档的index.html中，有导航链接直接链入，有点美中不足的是，当前版本还不支持聚合项目中模块报告的生成。

如下图7-6是我们对UserServiceImpl.java代码进程测试的测试覆盖报告：



图7-6 UserServiceImpl的测试覆盖报告

### 7.2.4 源代码分析插件

前面介绍了三个比较常用，也比较容易理解的插件，接下来，我们再介绍一个对Java源代码进行分析，形成报告的插件，maven-pmd-plugin插件。它能找出我们写的源代码中的问题，并且形成报告，供我们进一步改进源代码。maven-pmd-plugin可以发现源代码中潜在的bug、无用代码、可优化代码、重复代码以及过于复杂的表达式等。如果要对pmd插件有个详细的了解，大家可以访问http://pmd.sourceforge.net站点。

为了让pmd能帮我们生成分析报告，操作步骤同其它报告操作步骤一样，首先确定要使用的版本，和其它坐标信息；接着在pom的project->reporting->plugins中添加pmd插件，源代码如下：  
（随书代码[MvnBookSSMDemo.Service.Impl\pom.xml](代码/MvnBookSSMDemo.Service.Impl/pom.xml)）

<plugin>

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

<artifactId>maven-pmd-plugin</artifactId>

<version>2.5</version>

</plugin>

右击工程，选择Run As->Maven build...，执行site，就能在target/site目录下，发现pmd生成的报告文档。打开site下面首页index.html，点击Project Reports导航树中的PMD Report链接，就可以查看内容。下图7-7是关于MvnBookSSMDemo.Service.Impl工程中代码的pmd报告截图：

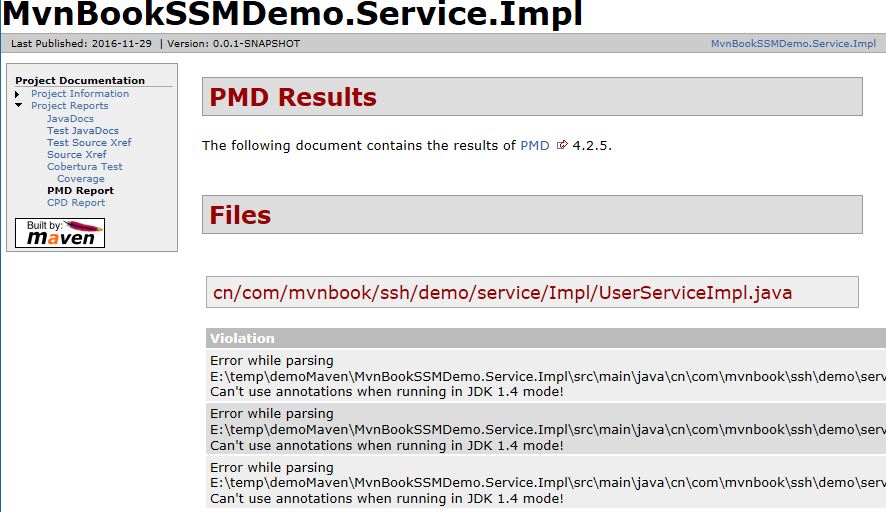


图7-7 UserServiceImpl的PMD

要对代码进行缺陷分析，肯定需要实现定义分析规则，这样才能按规则分析出结果。PMD包含有很多分析规则，我们可以访问http://pmd.sourceforge.net/rules/index.html，查看这些规则。PMD默认使用的是rulesets/basic.xml, rulesets/unusedcode.xml和rulesets/imports.xml。如果要使用其它规则，我们可以在pom的pmd插件配置文件中进行描述指定。代码模板如下：  
（随书代码[MvnBookSSMDemo.Service.Impl\pom.xml](代码/MvnBookSSMDemo.Service.Impl/pom.xml)）

<plugin>

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

<artifactId>maven-pmd-plugin</artifactId>

<version>2.5</version>

<configuration>

<rulesets>

<ruleset>rulesets/braces.xml</ruleset>

<ruleset>rulesets/strings.xml</ruleset>

</rulesets>

</configuration>

</plugin>

## 7.3 个性化站点

通过前面的介绍，我们可以感知到，基于Maven生成站点是非常方便灵活的，功能也很强大。当然这样归功于Maven的open精神，它可以包含随意的插件，通过集成大量插件的不同功能，从而体现Maven作为整体的强大。但是，如果每个项目的站点信息都长得一样的话，这样的世界也太单调了，而且不同的站点，再怎么样，它们的logo总该长得不一样吧。其实Maven有提供了让我们自己定义站点外观的方式，接下来，为了让我们的站点更同我们的项目和公司环境融合，开始介绍怎么样自定义站点的外观。

### 7.3.1 修饰外观

1. 站点描述符

为了定义站点外观，我们必须准备一个名称为site.xml的站点描述文件，默认情况下，site.xml需要放在src/site目录下。在site.xml中，描述我们需要在自定义站点的个性化信息。

我们创建一个简单的site.xml站点描述符文件，先做一个基本的体验，内容如下：  
（随书代码[MvnBookSSMDemo.Service.Impl\src\site\site.xml](代码/MvnBookSSMDemo.Service.Impl/src/site/site.xml)）

<?xml version="1.0" encoding="UTF-8"?>

<project name="MvnBookSSMDemo">

<bannerLeft>

<name>MvnBookSSMDemo.Service.Impl</name>

<src>images/logo.png</src>

<href>http://cyedu.ke.qq.com</href>

</bannerLeft>

<body>

<menu ref="reports"></menu>

</body>

<skin>

<groupId>com.googlecode.fluido-skin</groupId>

<artifactId>fluido-skin</artifactId>

<version>1.3</version>

</skin>

</project>

在上面的描述符文件中，我们定义站点的头部图片logo.png,导航栏菜单项reports和站点的皮肤。表现出的效果图7-8如下：

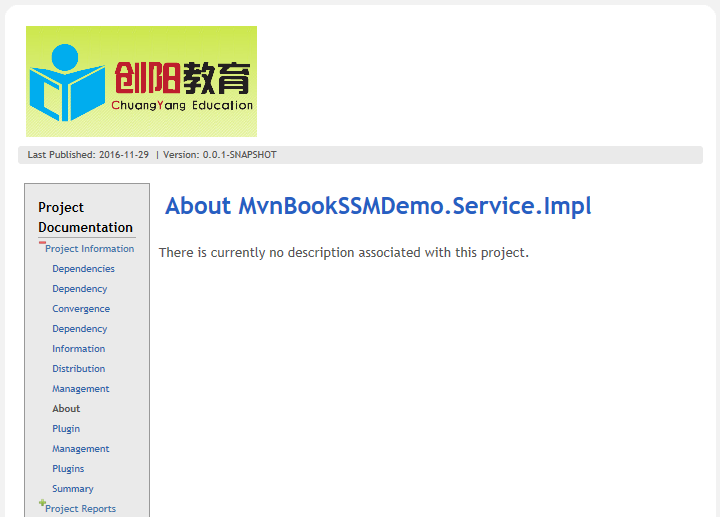


图7-8 站点logo

到现在为止，我们有基本体验了自定义站点描述符的编写，接下来我们详细介绍各类可以在站点描述符中定义的内容。

1. 头部内容和外观

默认情况下，Maven站点的标题显示的是pom中name的值，我们可以在站点描述符文件中进行指定，设置project的name属性，如果有设置，站点的标题就是name的值。比如：

<?xml version="1.0" encoding="UTF-8"?>

<project name="MvnBookSSMDemo 站点">

... ...

</project>

显示效果如下图7-9：

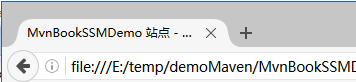


图7-9 站点名称

在缺省情况下，头部左边会显示项目的名称，但是为我们可以使用bannerLeft，配置左边要显示的图片。通常在这位置，显示公司的logo。同样的，我们也可以使用bannerRight配置显示右边的图片。具体样例代码如下：  
（随书代码[MvnBookSSMDemo.Service.Impl\src\site\site.xml](代码/MvnBookSSMDemo.Service.Impl/src/site/site.xml)）

<bannerLeft>

<name>MvnBookSSMDemo</name>

<src>images/logo.png</src>

<href>http://cyedu.ke.qq.com</href>

</bannerLeft>

<bannerRight>

<name>Apache Org</name>

<src>images/apache-maven-project.png</src>

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

</bannerRight>

左边配置的是公司logo，右边配置的是Apache的Maven标志条幅

效果如下图7-10：



图7-10 站点条幅

这里需要注意一下的是，左边的图片是本地图片。站点的本地资源，需要放在工程的src/site/resources目录下。比如说我们左边图片，就是放在src/site/resources/images目录下的。

除了标题和头部条幅图片外，Maven提供了让我们配置是否显示站点最近发布时间和版本的方式。样例代码如下：

<project name="MvnBookSSMDemo 站点">

... ...

<version position="right"></version>

<publishDate position="left"></publishDate>

... ...

</project>

其中postion的值包括none、left、right、navigation-top、navigation-bottom和bottom，它们分别表示不显示、左边、右边、导航栏上方、导航栏下方和底部。

当然，Maven站点还支持面包屑导航，配置代码如下：

<project name="MvnBookSSMDemo 站点">

... ...

<body>

<breadcrumbs>

<item name="cyedu" href="http://cyedu.ke.qq.com"></item>

<item name="Maven" href="http://maven.apache.org"></item>

</breadcrumbs>

...

</body>

... ...

</project>

显示效果如图7-11（包含前面配置的发布日期和版本）：



图7-11 站点头日期

皮肤

我们除了可以定义站点的标题、横幅图片和面包屑导航外面，我们还可以选择站点的皮肤，让我们的站点同别人的站点有不同的外观表现。

自定义站点的皮肤，一般分两步：

第一步：选择要使用的站点皮肤构件；

第二步：在站点描述符文件的shin元素中，使用选择的皮肤构件。

目前，Maven官方提供了三款皮肤，它们分别是：

org.apache.maven.skins:maven-classic-skin

Org.apache.maven.skins:maven-default-skin

Org.apache.maven.skins:maven-stylus-skin

其中maven-default-skin是站点的默认皮肤，我们可以在中央仓库中查询这些皮肤的最新版本。

当然，处理官方的皮肤构件，还有大量的第三方用户创建的站点皮肤，比如在GoogleCode上，就有个名为fluido-skin的皮肤。我们如果感兴趣，可以去网路上收罗一下。

我们这里以maven-stylus-skin为例子，体现一下皮肤构件的使用。

大家可以自己到http://mvnrepository.com找到自己中意的版本，我们这里使用的是1.5版本，在site.xml中的样例配置如下：  
（随书代码[MvnBookSSMDemo.Service.Impl\src\site\site.xml](代码/MvnBookSSMDemo.Service.Impl/src/site/site.xml)）

<project name="MvnBookSSMDemo 站点">

... ...

<skin>

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

<artifactId>maven-stylus-skin</artifactId>

<version>1.5</version>

</skin>

... ...

</project>

右击工程，选择Run As->Maven build...，执行site，生成的站点如下，大家可以比较出它跟前面的显示完全不一样的风格了，这就是皮肤的效果如图7-12。

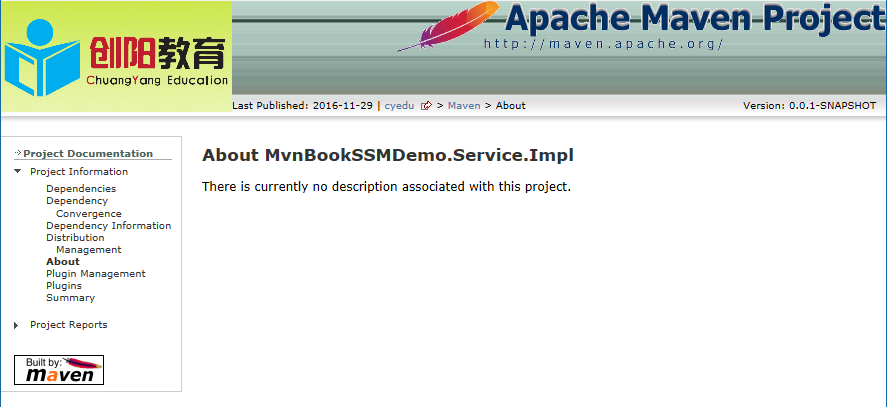


图7-12 皮肤效果

导航栏

当然，我们不仅仅可以根据自己的需要，指定横幅图片和皮肤，体现我们自己的个性需要，同时，如果觉得左边的导航需要调整的话，我们也可以根据自己的意图进行调整的。

调整左边导航栏，是通过编辑body元素下的menu子元素实现的，如下是一个调整左边导航的样例代码：  
（随书代码[MvnBookSSMDemo.Service.Impl\src\site\site.xml](代码/MvnBookSSMDemo.Service.Impl/src/site/site.xml)）

<?xml version="1.0" encoding="UTF-8"?>

<project name="MvnBookSSMDemo 站点">

... ...

<body>

... ...

<menu ref="parent"/>

<menu ref="modules"/>

<menu ref="reports"/>

<menu name="第一个菜单">

<item name="简介" href="introduction.html"></item>

<item name="常见问题" href="faq.html"></item>

</menu>

<menu name="第二个菜单">

<item name="test1" href="test1.html"></item>

<item name="test2" href="test2.html"></item>

</menu>

</body>

... ...

</project>

上面的代码，我们定义了五个菜单，前面三个是直接沿用Maven站点默认生成的页面，parent表示包含父模块链接菜单，modules包含子模块链接，reports表示包含项目报告菜单。

根据site.xml配置生成的站点如图7-13：

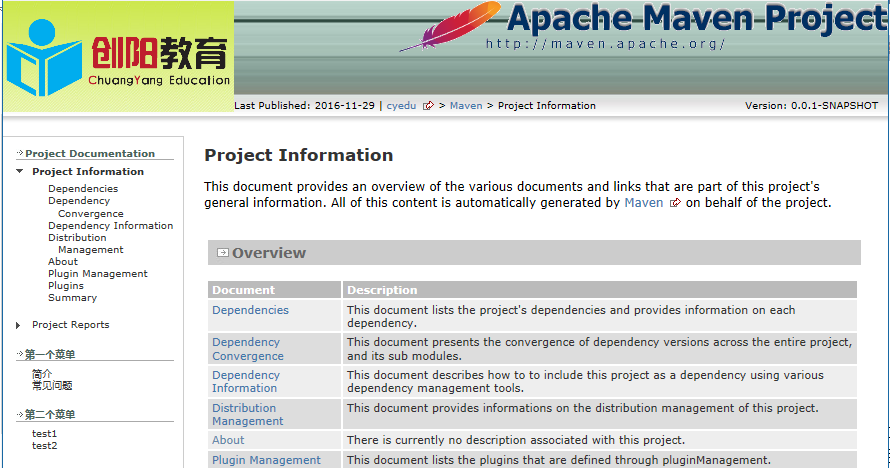


图7-13 站点自定义菜单

### 7.3.2 自定义页面

前面我们介绍了个性化横幅图片、菜单、发布时间版本等，接下来我们介绍怎么样在站点中定义自己的页面。

比如上面自定义栏目中，有个第一个菜单，里面有简介和常见问题两个链接，虽然我们有指定链接所链上的文件，但是我们根据就没有这文件，现在点击链接，肯定不可能有内容显示的。那怎么样自己定义页面，让它们同我们自定义链接整合起来呢？

到目前为止，Maven对APT和FML两个格式的文档支持比较好，接下来分别介绍怎么样基于APT和FML格式自定义站点页面文档，同我们自定义的菜单整合起来。

APT是一种类似维基的文档格式，是Almost Plain Text的简写，我们可以用它快速创建简单有结构丰富的文档。

接下来我们基于APT，实现一个简单的介绍页面，页面效果如下图7-14：



图7-14 自定义页面

生成如上图的apt代码introduction.apt代码如下：  
（随书代码[MvnBookSSMDemo.Service.Impl\src\site\introduction.apt](代码/MvnBookSSMDemo.Service.Impl/src/site/introduction.apt)）

1 ----------------

2 Demo简介

3 ----------------

4 Noble Yang

5 ----------------

6 2016-11-29

7 ----------------

8

9Maven是什么?

10

11 Maven是一个构件项目工具、是一个依赖管理工具、是一个站点报告生成工具...

12 总之，就是一个项目开发过程中的大管家！

13

14Maven实战指南的宗旨是什么?

15

16 用简洁的语言、简单的逻辑、简单的概念、简洁的步骤，直接接受Maven的使用

17

18Maven实战的特点

19

20\* 简单

21

22 简单的案例、简单的语言、简洁的操作步骤

23

24\* 项目实例

25

26 直接以实现项目为目的

27

28\* 简单的java工程

29

30\*\* 简单的Web应用

31

32\*\*\* 简单的SSH应用

33

34\*\*\* 简单的SSM应用

为了方便说明，我们给代码的每行添加了一个行号说明。

1、第1到7行是apt文档的头部分，该部分是可选的，主要描述标题、作者和发布日期等信息，关于这部分，有如下几个规则：

1. 每行开头，必须有至少两个空格
2. 每行文字之间，需要有一个---字符行，而且至少三个-字符
3. 第8行是一个空格行，标题和文本内容部分，需要用空行隔开
4. 第9、14、18行为每个段落的标题行，前面不能有空格
5. 第10、15、19行是分隔标题和内容的空格行
6. 第11、16行为标题后面的文本行，每个文本行前面需要空两个空格
7. 第20、24、28为列表行，相当于在html里面的li元素，注意\*号定格，同文字后面要空一个空格
8. 第30、32、34行为第28行定义的列表项下面的子项，\*号越多，表明层次越深

上面是我们根据例子的需要，对apt格式的一个简单说明。其实从效果上来看，我们完全可以使用html元素实现，但是如果用html实现的话，我们肯定需要使用很多html标签进行描述的，相对html来说，apt就简单多了，用些固定的约定和几个简单符号，就可以实现想要的效果。

当然，我们这里介绍的只是一部分apt规则，还有有序列表、表格、分页、链接等等，我们可以查看http://maven.apache.org/doxia/references/apt-format.html进一步了解，这里就不一一列到了。

类似常见问题的页面一样，有很多页面内容有很多标题和内容组成，在页面的开始，是这些标题的导航链接。如下图7-15：



图7-15 导航链接

这样的格式比较简单，有规律，我们就可以使用fml格式的文档描述。

接下来我们通过介绍faq.fml的内容，说明一下fml格式的大体规则。

faq.fml的内容如下：  
（随书代码[MvnBookSSMDemo.Service.Impl\src\site\faq.fml](代码/MvnBookSSMDemo.Service.Impl/src/site/faq.fml)）

<?xml version="1.0" encoding="UTF-8"?>

<faqs xmlns="http://maven.apache.org/FML/1.0.1"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/FML/1.0.1 http://maven.apache.org/xsd/fml-1.0.1.xsd"

title="常见问题"

toplink="false">

<part id="general">

<title>基本问题</title>

<faq id="whats-foo">

<question>

Maven是什么?

</question>

<answer>

<p>由骨架文件配置好所有的插件和依赖，再用固定命令构建项目的工具</p>

<source>

<![CDATA[

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.com.mvnbook.ssm.demo</groupId>

<artifactId>MvnBookSSMDemo.Service.Impl</artifactId>

<packaging>jar</packaging>

... ...

</project>

]]>

</source>

<p>上面是pom.xml骨架文件的简单样例</p>

</answer>

</faq>

<faq id="whats-bar">

<question>

What is Bar?

</question>

<answer>

<p>some markup goes here</p>

</answer>

</faq>

</part>

<part id="install">

<title>Installation</title>

<faq id="how-install">

<question>

How do I install Foo?

</question>

<answer>

<p>some markup goes here</p>

</answer>

</faq>

</part>

</faqs>

说明：

fml文档是一个标准的xml文档，根元素是faqs

faqs元素有个title属性，描述文档的主题

faqs中可以包含多个part元素，每个有个id属性，用来唯一标记

每个part元素，封装一个内容的大类，它有title子元素，描述大类主题

1. part元素里面，再包含多个faq元素，包含每个问题的题目和内容。question封装问题的题目，answer封装问题的答案。考虑到问题答案除了有文字，还会有代码，提供了p元素封装答案的文字说明，source封装答案的辅助代码。

### 7.3.3 国际化

作为一个中国的普通读者，还是更希望、也更方便查看中文文档。我们前面生成的站点信息都是英文版的，有没有一种方式，能生成我们本地语言（中文）的呢？

Maven-site-plugin插件支持本地化站点的生成，但是要作如下四个方面的准备和配置：

1. 所有的项目文档，包括源代码和所有资源配置文件，需要用utf-8格式保存
2. 在pom.xml中配置指定maven-site-plugin使用utf-8格式读取源代码和文档的编码
3. 在pom.xml中配置指定maven-site-plugin按utf-8格式输出站点文档
4. 在maven-site-plugin的插件元素中，添加configuration，用locals指定本地语言zh\_CN。

第2、3方面，我们在pom.xml的project中，添加如下配置可以实现  
（随书代码[MvnBookSSMDemo.Service.Impl\pom.xml](代码/MvnBookSSMDemo.Service.Impl/pom.xml)）

<properties>

<!--读取源代码和文档的编码-->

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<!--输出站点文档的编码-->

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

</properties>

第4方面，我们可以在pom.xml中添加如下配置实现：

<build>

... ...

<plugins>

<plugin>

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

<artifactId>maven-site-plugin</artifactId>

<version>3.4</version>

<configuration>

<!--汉字语言环境-->

<locales>zh\_CN</locales>

</configuration>

</plugin>

</plugins>

</build>

完成前面介绍的配置后，新站点页面如下图7-16：



图7-16 站点多国语言

## 7.4 部署站点

现在我们基本能生成我们自己想要的站点和报告文档，但是这些文档，我们只能在本地浏览。当然，我们也可以将这些文档自己copy到公司的web服务器上去，让团队成员和允许访问的其它人可以时时跟踪查看，只是这么手动copy有点麻烦，而且每次更新站点后，都要手动复制，想我们MvnBookSSMDemo那样，有多个模块平行开发，最后聚合形成整体项目的开发模式，就更麻烦了，因为每次更新都要每个模块都要重新copy到服务器上去，效率太低了。

接下来我们介绍一下怎么样自动部署站点文件。

Maven支持多种协议的站点部署，包括FTP、SCP和DAV。不顾是哪种，目标和思路都是一样的：由maven将所有的站点文档，按照配置信息，copy到指定的服务器位置。FTP和SCP我们就不介绍了，接下来我们介绍基于DAV协议，将站点自动部署到Tomcat7 Web服务器上去，Tomcat服务器是我们Java程序员开发web应用常用的服务器，所以以它为例子。

在具体使用DAV协议部署站点前，我们先简单了解一下DAV的概念。

DAV是什么呢？

首先很明显，它是一个协议。

DAV是一个在HTTP协议的基础上进行扩展了的web通讯协议，我们可以通过这个协议，完成对网络文件管理的工作，具体内容包括支持client远程锁定web服务器上的文件；支持远程查找、定位Web服务器上的文件；支持创建、复制 和 移动Web服务器上的文件。

了解DAV的概念后，我们就可以自然想象得到，要实现maven基于DAV协议的站点文档在Tomcat7上的自动部署，需要做两个大方向的实现：第一个方面，在Tomcat7上配置DAV服务；另外一个方面是在我们开发电脑上，配置maven信息，让maven可以自动将生成的站点文档发布到dav服务上去。

接下来我们分别从这两个方面进行介绍：

1. 在Tomcat7上大家dav服务

Tomcat7 Web 服务器默认是支持dav协议的，我们需要做的是，在Tomcat7 Web服务器上搭建一个dav服务，该服务可以完成在Web服务器中，基于dav协议的文件操作。能实现dav的服务类，在Tomcat7中已经有提供了，它是org.apache.catalina.servlets.WebdavServlet类。我们接下来要做的是，将该Servlet发布成dav服务，并且设置好用户名和密码，让客户端能基于这个服务，操作Tomcat7 Web 服务器里面的文件。具体操作如下：

1. 搭建DAV服务

A、在Tomcat7的应用发布目录webapps里面，创建一个目录（site），这个目录是一个新应用的上下文路径，站点文件将被发布到这里。

B、在新建的应用路径（site）下，创建WEB-INF目录，并且按web标准，创建一个web.xml文件

C、在web.xml中，部署org.apache.catalina.servlets.WebdavServlet Servlet类，并且配置好servlet-mapping，拦截所有的请求。详细代码和说明，请看下面代码清单：

<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

<servlet>

<servlet-name>webdav</servlet-name>

<servlet-class>org.apache.catalina.servlets.WebdavServlet</servlet-class>

<init-param>

<param-name>debug</param-name>

<param-value>0</param-value>

</init-param>

<init-param>

<param-name>listings</param-name>

<param-value>false</param-value>

</init-param>

<init-param>

<param-name>readonly</param-name>

<!--允许修改删除-->

<param-value>false</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>webdav</servlet-name>

<url-pattern>/\*</url-pattern>

</servlet-mapping>

<display-name>site</display-name>

<session-config>

<session-timeout>

30

</session-timeout>

</session-config>

<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

</web-app>

启动web服务器，dav服务就会自动启动，只要有权限，客户端就可以通过这个dav服务，管理Tomcat7 Web服务器上的文档。

设置Tomcat7的用户名和密码

为了让客户端顺利的通过搭建好的dav服务，管理服务器上的文档，我们需要在Tomcat7上配置管理web应用的用户名和密码。这样dav通讯的时候，就可以通过用户名和密码的认证，拥有对应的管理权限。

打开TomcatHome/conf/tomcat-users.xml，在里面添加两个角色，分别是manager-gui和admin-gui，代码如下：

<role rolename="manager-gui"/>

<role rolename="admin-gui"/>

再在该文件中添加一个用户名和密码，给它赋予manager-gui和admin-gui两个角色。用户名和密码随意取。样例代码如下：

<user username="both" password="admin123" roles="manager-gui,admin-gui"/>

这样maven客户端配置好用户名both和密码admin123后，就可以在操作web服务器上的文件前，认证通过后获取所有管理权限。

配置maven的dav自动部署

关于maven的配置，我们大概要做如下几个方面的工作：

1）配置maven客户端访问dav服务器前的安全认证信息

打开用户目录下的[.m2/settings.xml](代码/settings.xml)文件，在settings->servers元素下，添加一个server元素，描述我们在Tomcat7中配置好的both用户和对应的密码，用一个id标记，这id名称需要记住，下一步在pom.xml中要对应配置的。样例配置代码如下：

<server>

<id>siteServer</id>

<username>both</username>

<password>admin123</password>

</server>

我们这里配置的id名称叫siteServer

2）在Maven工程的pom.xml中配置站点部署信息

打开要部署站点的maven工程的骨架文件(pom.xml)，在project->distributionManagement中，添加一个site元素，描述要部署的目标url和对应的认证服务id，我们的MvnBookSSMDemo.Service.Impl工程的配置代码如下：  
（随书代码[MvnBookSSMDemo.Service.Impl\pom.xml](代码/MvnBookSSMDemo.Service.Impl/pom.xml)）

<project>

...

<distributionManagement>

<site>

<id>siteServer</id>

<url>dav:http://127.0.0.1:9080/site/MvnBookSSMDemo.Service.Impl</url>

</site>

...

</distributionManagement>

其中id值为siteServer，需要同我们在第一步中配置的server的id一致。Url以dav:开头，表示使用dav协议，发布到127.0.0.1服务器的，上下文为site的应用下，服务器的端口为9080（Tomcat的默认端口为8080，具体端口请关注启动服务器后显示的http-port ）。

如果当前部署的站点是一个独立的站点的话，url只要到site上下文路径就行了，表示把站点文件，发布到site的根目录下。如果当前的站点只是一个module站点，需要聚合到一个大站点去的，这样，我们就需要指定当前module站点在总站点中的相对路径。比如我们这里的MvnBookSSMDemo.Service.Impl, MvnBookSSMDemo.DAO.MyBatis, MvnBookSSMDemo.SpringMVC将用来聚合到MvnBookSSMDemo中去的，所以在它们的url中，需要在site后面加上自己的相对路径，自己的工程名称。比如样例中，MvnBookSSMDemo.Service.Impl的url为dav:http://127.0.0.1:9080/site/MvnBookSSMDemo.Service.Impl。这样聚合后才能集成在一起。

同时再加上如下代码，指定webdav构建：

<project>

...

<build>

<extensions>

<extension>

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

<artifactId>wagon-webdav-jackrabbit</artifactId>

<version>1.0-beta-7</version>

</extension>

</extensions>

</build>

...

</project>

按前面的配置好后，右击MvnBookSSMDemo工程，选择Run As->Maven build...，先运行site:site（生成站点信息，包括每个模块）,再运行site:deploy（部署站点）,在浏览器中输入http://localhost:9080/site/index.html，我们就可以看到如下图7-17的综合站点页面：

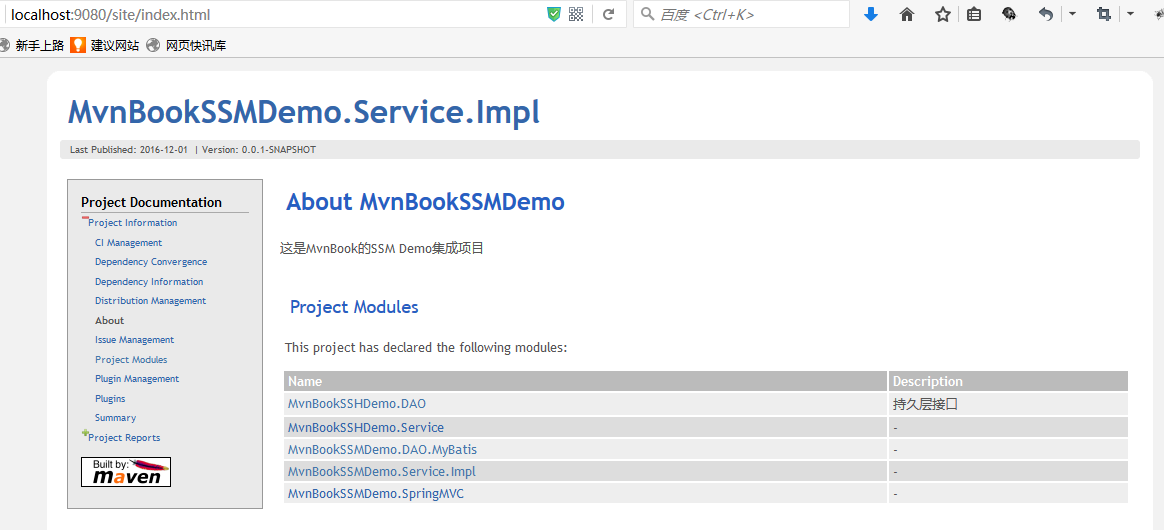


图7-17 站点发布后首页

## 7.5 小结

本章详细介绍了如果使用Maven生成站点，美化站点和自定义站点页面，同时还介绍了使用常用的几个报告插件，生成对应报告，同站点合并在一起。这些插件的功能都是独立的，需要其它功能，我们可以在中央仓库中找到后，自由的整合到我们的站点中。

另外还对站点的汉化做了简单的介绍，最后详细介绍了怎么基于dav协议，在Tomcat7上自动发布站点。

# 第八章 版本管理

## 8.1 简介

一个正常的项目开发，是一个比较长期的发展过程，这个过程，有着每个项目的生命周期和各个功能的里程碑，我们一般会把这些阶段和里程碑确定成一个个的版本，以便对整个项目有个历程的管理和阶段目标的控制实现，那我们可以怎么样科学的管理项目的版本呢，我们接下来就讨论这个问题。

在正式进行研究前，我们有必要澄清一下两个概念：版本管理 和 版本控制。

版本管理是指对项目的整体版本的演变过程进行管理，如从1.0到1.1，再到1.2,2.0等。版本控制是借助第三方的版本控制工具，追踪管理项目文档（包括代码）的每一个变更。

我们接下来主要研究方向是版本管理，而不是版本控制，请大家先注意版本管理和版本控制的区别，已经我们的研究目标。

## 8.2 准备点专业术语

为了方便后面的版本管理的理解，我们先了解一下相关的专业术语，也就是概念。

1. 快照

在项目开发过程中，为了方便团队成员的合作，解决模块间相互依赖和时时更新问题，我们对每个模块构建的时候，输出成临时性的版本，叫快照版本，这种版本定位的构件文件，会随着开发的进展，不断的变法更新，同时Maven对同一个快照构件的依赖，也会同步更新，方便团队内部的相互用到的依赖，都是最新的。

1. 发布版本

项目开发到一定的阶段后，就需要向团队外部发布一个比较稳定的版本，这个版本构件所对应的构件文件，是固定的，不会发生变化。就算后期有更多新功能要继续开发，开发完后也不会改变当前发布版本的内容，这样的版本我们叫发布版本。

1. 版本管理

项目开发过程中，团队内部会随着项目的进展，各个模块发布最新的快照版本，但是开发到一定的阶段，我们需要将快照版本定位成一个发布版本，对团队外部进行发布，同时，在这个定位版本的基础上进行再续开发，开发过程中又形成不断更新的快照，到一定阶段后，再发布一个定位的发布版本，以此重复进行，直到最后项目的完成。这个过程中快照版本和发布版本的切换管理，就是版本管理。也就是说，版本管理关系的一个核心问题，就是要科学的解决快照和发布版本之间的切换问题。

理想的发布版本，应该是在项目进展到一个比较稳定状态的时刻，这种稳定状态包括源代码的状态和构件的状态，所以一般构建一个稳定版本的条件有下面几个方面：

1. 所有的测试案例应该全部通过。

这点应该是可以理解的，自己的测试案例都不能通过，说明有预期的bug，这样带有明显bug的版本发布是没有意义的。

1. 项目中没有配置任何快照版本的依赖。

发布版本相对快照版本最显著的区别就是稳定，如果自己包含有对快照构件的依赖，依赖的基础都不是稳定的，怎么能谈得上自己的稳定呢？

1. 项目中没有配置任何快照版本的插件

同上面的道理一样

1. 项目中所有的文档（包含代码）都有提交到版本控制系统

一个稳定版本的发布，不仅仅表示项目进入了一个相对稳定的阶段，而且还必须保证相关的文档能齐备保存，以便以后能准确的回到这个阶段。如果要发布的文档没有全部集中提交到版本控制系统中，意味着一不小心，文档就会残缺，这样的版本就没法原因回滚。

1. 版本号的约定

为了方便团队交流，Maven将版本号约定为四个部分组成，主版本、次版本、增量版本和里程碑版本，按如下格式共同形成一个版本号：

<主版本>.<次版本>.<增量版本>-<里程碑版本>

主版本：表示项目的重大架构的变更。比如Struts1和Struts2，它们的架构体系都不同，jUnit4和jUnit3，一个全面支持注解，另外一个就没有支持。

次版本：表示有较大的功能增加和变法，或者比较全面系统的bug修复等。

增量版本：表示有正大bug的修复

里程碑版本：表明一个版本的里程碑，注意，是一个版本内部的历程碑。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。

当然需要注意的是，不是每个版本号都必须由这四个部分组成，有些版本号就可以没有增量版本和里程碑。

1. 主干、分支、标签

主干：项目开发的主体，也是主线，关键历程。从这里可以获取项目的最新代码和绝大部分的变更历史。

分支：从主线某个点分离出去的一段分支。在一个特别时间点的时候，既要不影响项目的总体（主线）进度，又需要同步修改某些重要bug、或实现特殊功能、或实验性开发，就可以创建一个分支独立进行。分支达到预期效果后，需要将分支里面的变更合并到主线中去。

标签：用来标记分支和主线中，进展到某个状态的点，代表项目进展到某个阶段或某个相对比较稳定的状态，实际项目中，这种状态往往就是版本发布的状态。

## 8.3 自动版本发布

根据前面的介绍，我们可以手动的按照前面的规则和步骤，完成检查是否有未提交代码、是否有快照依赖、更新快照版本到发布版本、执行Maven构建构件、为版本控制器上的源代码打上版本标记等等。

开始的时候，我们会对这个过程会比较新鲜，能重复多次一步步的操作，但是做熟悉了，多了就会比较烦，重复流程太枯燥了，这时候就会希望有工具能帮我们自动的完成这些操作。Maven Release Plugin插件就可以满足我们的需要，实现所有的版本控制发布流程的自动化。接下来我们就介绍一下如果使用Maven Release Plugin插件，并且结合SVN版本控制器来发布项目版本。

Maven Release Plugin一共有三个目标，它们分别是：

1. release:prepare

准备版本发布，按流程顺序执行如下操作：

1. 检查项目是否有未提交的代码
2. 检查项目是否有快照版本依赖
3. 根据用户的输入将快照版本升级为发布版本
4. 将pom中的scm形象更新为标签地址
5. 基于修改后的pom执行maven构建
6. 提交pom变更
7. 基于用户输入的代码打标签
8. 将代码从发布版本升级成新的快照版本
9. 提交pom变更
10. release:rollback

回退release:prepare所执行的操作。

将pom回退到release:prepare之前的状态，并且提交。

需要注意的是，该步骤不会删除release:prepare生成的标签，因此需要我们手动的删除。

1. release:perform

执行版本发布。

签出release:prepare生成的标签中的源代码，并且在这基础上执行mvn deploy命令，打包并且将构件部署到仓库。

了解了Maven Release Plugin，接下来我们就开始介绍怎么使用Maven Release Plugin，在SVN版本控制器上，基于Maven自动发布版本。具体操作步骤如下：

1. 准备好环境

主要有三个软件或插件需要事先安装配置好。

1. SVN服务器

Maven Release Plugin要基于版本控制系统进行代码的签入签出并且打标签的，所以版本控制系统的服务器软件肯定是要安装的，我们这里以SVN为例子，服务器安装的是VisualSVN-Server软件，测试用的版本是3.5.3。大家可以到官网上下载VisualSVN-Server-3.5.3-x64.msi安装软件，官网链接是：

1. SVN命令行工具

安装可以基于命令行，执行svn命令，完成代码管理的工具。注意，一定要安装这样的工具，因为Maven在进行svn操作的时候，是基于svn命令进行操作的，即使安装了可视化SVN客户端，比如TortoiseSVN都是没有用的。笔者在测试本书的Demo的时候，因为美誉注意这点，很是花了一会功夫的。我们这里安装的是Slik-Subversion，大家可以在SVN官网下载Slik-Subversion-1.9.4-x64.zip进行安装。

1. Eclipse的SVN插件

因为我们要基于Eclipse的Maven项目，基于SVN进行代码的签入签出等操作，所以还需要在Eclipse中安装好SVN插件。建议安装跟自己Eclipse匹配的最新插件，以免出现不必要的麻烦。我这里的测试Demo，使用的是eclipse-java-mars-R-win32-x86\_64 Eclipse，大家可以从官网上下载 eclipse-java-mars-R-win32-x86\_64.zip包，插件用的是Subversive-SVN Team Provider 4.0.2。我们可以在Eclipse中点击Help->Eclipse Marketplace...，输入关键字svn search，找到对应的Subversive版本安装，安装好后，需要在插件里面安装好对应的连接器。点击Eclipse中Window->Preference，选择Team->SVN，显示的连接器如下图8-1：

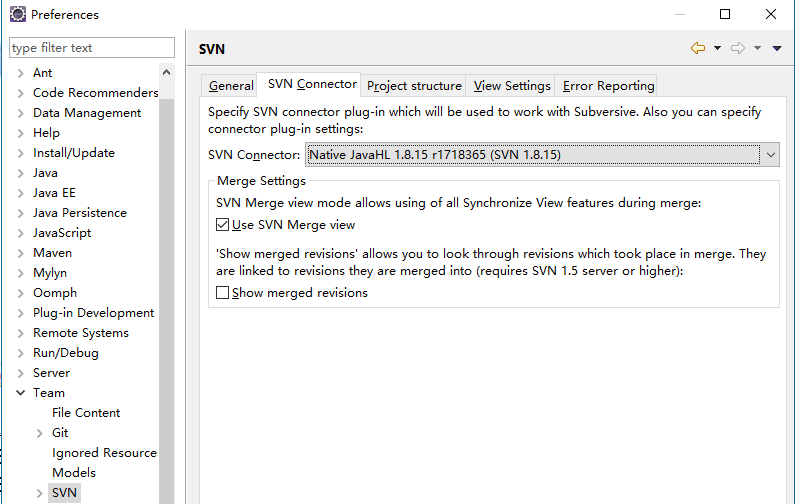


图8-1 SVN连接器

1. 在SVN服务器中创建好仓库，并且设置好用户名和密码

关于SVN的这方面的操作，我们就不累赘了，大家可以参考一下SVN相关的资料。跟我们测试Demo相关的url为：https://Noble-PC:8443/svn/MvnBookSSMDemo/trunk/svnDemo，用户名和密码为noble

1. 创建样例项目，在pom中配置相关信息
2. 在Eclipse中创建一个简单的Maven工程MvnBookTestSVN
3. 在pom.xml中配置版本控制系统信息，样例配置如下：

<project>

...

<scm>

<developerConnection>scm:svn:https://Noble-PC:8443/svn/MvnBookSSMDemo/trunk/svnDemo</developerConnection>

<connection>scm:svn:http://Noble-PC:8443/svn/MvnBookSSMDemo/trunk/svnDemo</connection>

<url>http://Noble-PC:8443/svn</url>

</scm>

</project>

developerConnection元素描述科协的scm地址

Connection元素描述的是只读的scm地址

Url表示浏览器可以直接访问的src地址。

为了让Maven识别，connection和developerConnection必须以scm开头，冒号后面的部分表示版本控制工具的类型(我们这里用的是svn)，Maven除了支持svn外，还支持cvs和git，后面才是实际的scm地址。在我们的例子中，connection使用的是http协议，developerConnection因为涉及到写操作，所有用的是https协议，以便有保护作用。

配置maven-release-plugin插件，样例配置如下：

<build>

<plugins>

...

<plugin>

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

<artifactId>maven-release-plugin</artifactId>

<version>2.5.3</version>

<configuration>

<tagBase>  
 <https://Noble-PC:8443/svn/MvnBookSSMDemo/tags/svnDemo> </tagBase>

<username>noble</username>

<password>noble</password>

</configuration>

</plugin>

</plugins>

</build>

tagBase指定的是标签的基本目录

Username和password是svn的用户名和密码

配置好私服的发布信息，样例配置如下：

<project>

...

<distributionManagement>

<repository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

</distributionManagement>

</project>

配合私服的访问的权限认证，需要在settings.xml中配置私服的用户认证信息：

<server>

<id>archivaServer</id>

<username>admin</username>

<password>admin123</password>

</server>

基于Maven自动发布版本

在Eclipse中创建一个Repository Location

点击Window->show view->Others->SVN Repositories，打开SVN Repositories View。  
 如图8-2：

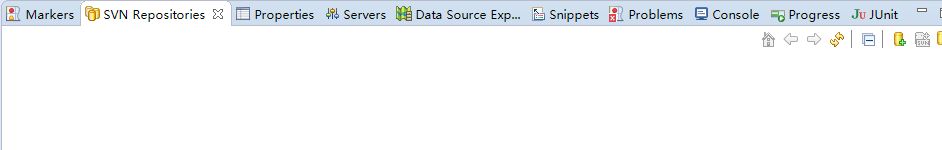


图8-2 SVN Repositories视图窗口

右击空白区域 new->Repository Location，打开创建Repository Location的窗口，在里面输入svn的url，用户名和密码，并且选上保存认证用户名和密码的选框，如图8-3：

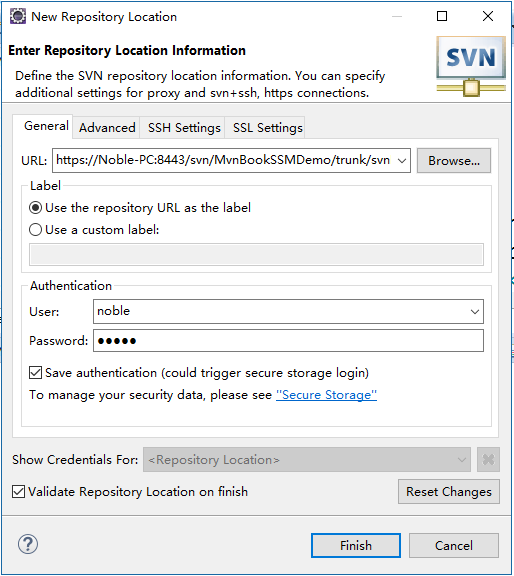


图8-3 设置SVN连接信息

点击Finish按钮，创建一个Repository Location

将MvnBookTestSVN Share到SVN服务器中（添加到SVN）

右击MvnBookTestSVN工程，选择Team->Share Projects，选中已经存在的Repository Location，如图8-4：

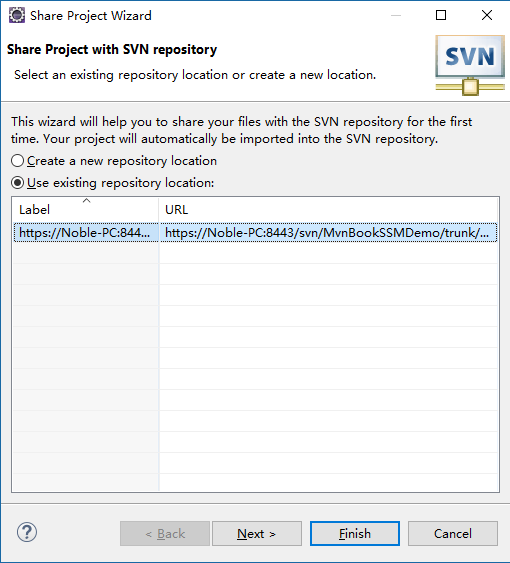


图8-4 选择SVNRepository

点击 Finish按钮，再点击接下来窗口中的 确定按钮，完成TestSVN工程的添加，如下是添加到SVN后的工程样例图8-5：

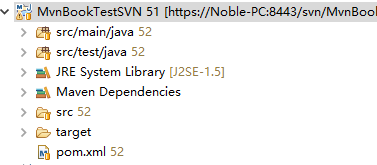


图8-5 SVN管理后的工程视图

同样，我们也可以在VisualSVN Server的控制界面中，查看到刚刚加入的MvnBookTestSVN，如图8-6：

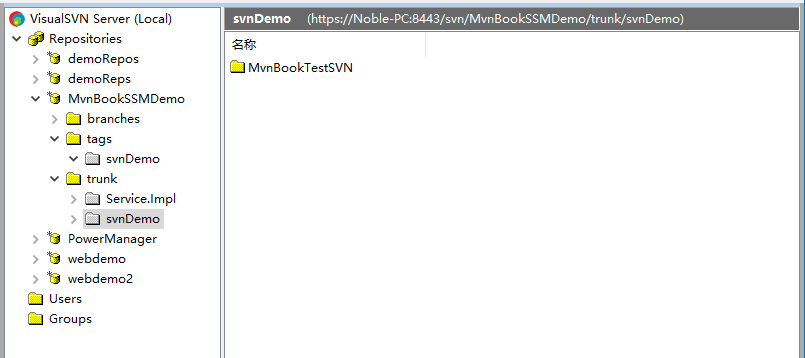


图8-6 VisualSVN Server Repositories

在Eclipse中修改几次代码，并且给予SVN提交到服务器中去

右击工程 Run As->Maven build...，在Goals后面输入release:prepare，如图8-7：

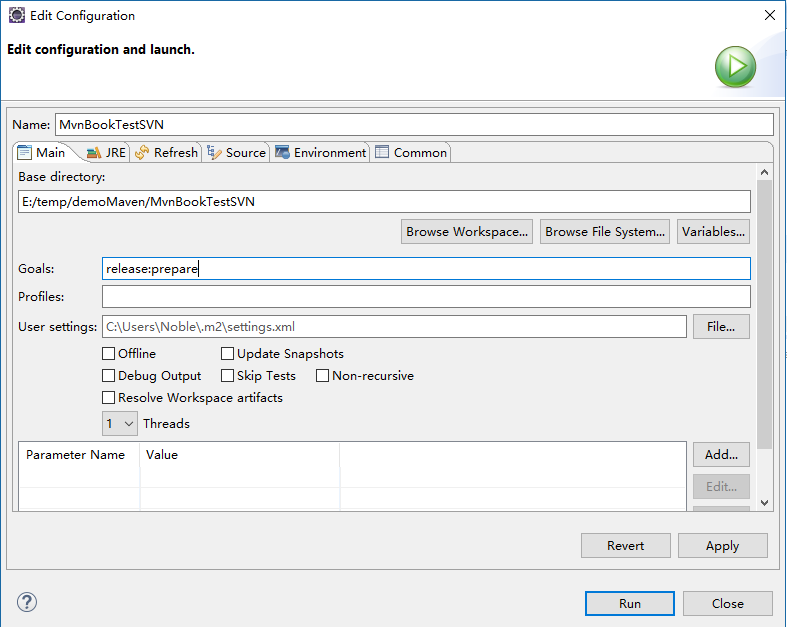


图8-7 配置m2eclipse版本发布

点击Run按钮，控制台输入如下信息，表示release:prepare成功：

[INFO] Release preparation complete.

[INFO] -----------------------------------------

[INFO] BUILD SUCCESS

[INFO] -----------------------------------------

[INFO] Total time: 39.948 s

[INFO] Finished at: 2016-12-13T23:06:45+08:00

[INFO] Final Memory: 13M/107M

[INFO] ------------------------------------------

右击MvnBookTestSVN项目，Run As->Maven Build...，在Goals中输入release:perform，点击Run按钮（同上图中输入release:prepare操作一样），将TestSVN工程正式发布成第一个发布版本，并部署到私服中。

我们就可以在svn服务器中的tags目录中，找到刚刚发布的一个新版本，如图8-8：

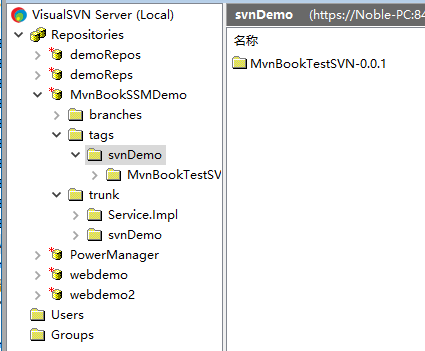


图8-8 发布后的服务器查看视图

同样，同过Eclipse中SVN插件，也可以查看到更新图（Revision Graph）如下图8-9：

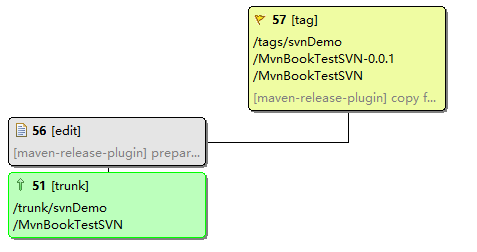


图8-9 SVN版本更新视图

而且我们查看pom.xml中的version，会发现version值自动从0.0.1-SNAPSHOT变成了0.0.2-SNAPSHOT。

重复前面的操作，我们又可以基于Maven Release Plugin插件发布第二个版本，从如下图8-10的Revision Graph可以清晰看出中间的变化，同样，pom中version也自动变成了下一个版本。

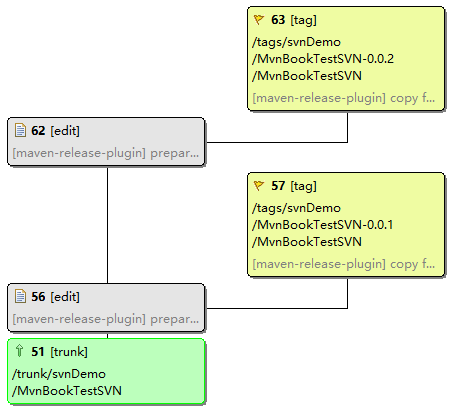


图8-9 插件更新视图

## 8.4 GPG签名验证

基于网络的开源项目，给我们带来在公共标准的基础上的自由发挥，并且能很好的给每个自愿人士提供了共享贡献的机会，同时也因为大众化，也给我们使用共享的程序员或团队带来了不确定的安全性问题。当我们从中央仓库下载第三方构件的时候，下载的文件有可能级被另外一个人篡改过，从而给我们代码不必要的破坏，为了确定下载的内容是正确的，我们一般在发布自己构件的同时，还会发布一个签名认证文件，使用者在使用下载的第三方构件前，先通过签名验证后，确定没有被篡改再安心使用。PGP(Pretty Good Privacy)就是这么个认证签名技术。接下来我们介绍怎么样使用PGP技术，为发布的Maven构件签名，从而提高项目的安全性。

GnuPG，简称GPG，来至http://www.gnupg.org，是PGP标准的一个免费实现，不管是Linux还是Windows平台，都可以使用。GPGneng 帮助我们为文件生成签名，管理密匙以及验证签名等。

接下来我们介绍一下使用GPG实现文件签名，并且验证签名文件。分两个阶段介绍：第一阶段介绍GPG的手动操作，第二阶段介绍怎么样基于Maven插件对Maven项目文件进行签名。

1. 手动操作GPG

使用GPG一般需要如下几个步骤：

1. 下载安装GPG

访问http://www.gnupg.org/download，下载适合自己操作系统平台的安装程序，我们这里下载的是Windows平台的gpg4win-2.3.3.exe。

点击安装gpg4win-2.3.3.exe，具体过程同安装普通软件一样。

安装完后，打开cmd窗口，输入 gpg --version，出现如下信息，表示安装成功：

gpg (GnuPG) 2.0.30 (Gpg4win 2.3.3)

libgcrypt 1.6.6

Copyright (C) 2015 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law.

Home: C:/Users/Noble/AppData/Roaming/gnupg

Supported algorithms:

Pubkey: RSA, RSA, RSA, ELG, DSA

Cipher: IDEA, 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH,

CAMELLIA128, CAMELLIA192, CAMELLIA256

Hash: MD5, SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224

Compression: Uncompressed, ZIP, ZLIB, BZIP2

生成密钥对

在使用GPG之前，我们先要准备一个密钥对：一个私钥，一个公钥。接下来才能使用私钥对文件进行签名，将公钥分发到公钥服务器，供其他用户下载，其它用户就可以使用公钥对签名进行验证。

在cmd命令行中，输入 gpg --gen-key命令生成密钥对。

在GPG执行过程中，会提示如下几个信息：

生成密钥类型

Please select what kind of key you want:

(1) RSA and RSA (default)

(2) DSA and Elgamal

(3) DSA (sign only)

(4) RSA (sign only)

Your selection?

通过输入1或直接回车（默认），选择第一项

RSA Keys的大小

RSA keys may be between 1024 and 4096 bits long.

What keysize do you want? (2048)

输入一个介于1024到4096直接的整数，或直接回车（默认2048），我们这里直接回车，选择的是2048。

密钥有效期

Please specify how long the key should be valid.

0 = key does not expire

<n> = key expires in n days

<n>w = key expires in n weeks

<n>m = key expires in n months

<n>y = key expires in n years

输入密钥有效时长，默认是0，表示永不过期，输入一个数字n，表示有效期为n天，当然也可是输入nw、nm、ny，分别表示n周、n月和n年。我们这里选择的是直接回车，表示永不过期。

提示前面的选择是否正确（是否确认）

Is this correct? (y/N)

输入y，表示确认，输入回车或n，表示要重新输入新的有效期

接下来的信息，是为了生存GPG的唯一用户ID的信息：

输入开发者或团队名

Real name:

作为演示，我们这里输入NobleForMvnBook

联系邮箱地址

Email address:

[作为演示，我们这里输入3310435058@qq.com](mailto:作为演示，我们这里输入3310435058@qq.com)

输入备注信息

Comment:

作为演示，我们这里输入this is a demo for MvnBook

这时候会提示如下信息，显示生成的USER-ID：

You selected this USER-ID:

"NobleForMvnBook (this is a demo for MvnBook) [<3310435058@qq.com>"](mailto:%3c3310435058@qq.com%3e%22)

中间NobleForMvnBook (this is a demo for MvnBook) [<3310435058@qq.com>为USER](mailto:%3c3310435058@qq.com%3e为USER) ID

修改生成USER ID的信息

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit?

输入N、C、E，分别用来修改名称、备注和邮件地址信息

输入Q表示退出

输入O表示进入下一步，我们这里输入O回车

输入私钥密码

我们这里输入自己的密码，作为演示，我们这里输入的noble123

接下来GPG会提示如下信息，表示密钥对已经生成：

generator a better chance to gain enough entropy.

gpg: C:/Users/Noble/AppData/Roaming/gnupg/trustdb.gpg: trustdb created

gpg: key 25C6CAD0 marked as ultimately trusted

public and secret key created and signed.

gpg: checking the trustdb

gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model

gpg: depth: 0 valid: 1 signed: 0 trust: 0-, 0q, 0n, 0m, 0f, 1u

pub 2048R/25C6CAD0 2016-12-18

Key fingerprint = A1BB E48B 8003 8C72 2954 781A 8280 BE47 25C6 CAD0

uid [ultimate] NobleForMvnBook (this is a demo for MvnBook) <3310435058@qq.com>

sub 2048R/F01A5633 2016-12-18

其中C:/Users/Noble/AppData/Roaming/gnupg/trustdb.gpg表示生成的位置等信息。

查看公钥和私钥信息

在cmd命令行窗口中输入gpg --list-keys，查看本地公钥信息，列表如下：

C:/Users/Noble/AppData/Roaming/gnupg/pubring.gpg

------------------------------------------------

pub 2048R/25C6CAD0 2016-12-18

uid [ultimate] NobleForMvnBook (this is a demo for MvnBook) <3310435058@qq.com>

sub 2048R/F01A5633 2016-12-18

第一行显示公钥文件和所在的位置

pub 行描述的是 公钥大小(2048)/公钥id(25C6CAD0) 公钥产生日期(2016-12-18)

uid行描述的是有名称、备注和邮件地址组成的字符串

Sub 表述的是子钥(我们可以不用关心)

在cmd命令行窗口中输入gpg --list-secret-keys，查看本地私钥信息，列表如下：

C:/Users/Noble/AppData/Roaming/gnupg/secring.gpg

------------------------------------------------

sec 2048R/25C6CAD0 2016-12-18

uid NobleForMvnBook (this is a demo for MvnBook) <3310435058@qq.com>

ssb 2048R/F01A5633 2016-12-18

第一行表示密钥文件和所在位置

sec描述密钥大小(2048)、id(25C6CAD0)和产生日期(2016-12-18)

uid描述的是名称、备注和邮件地址组成的字符串

Ssb描述的是密钥的子钥(我们可以不关心)

给文件创建签名文件

打开cmd命令行窗口，切换到IMvnBookDAO.java文件所在的目录

输入gpg -ab IMvnBookDAO.java命令，输入我们前面生成密钥时输入的密码noble123，gpg会在当前目录下，生成一个名字叫IMvnBookDAO.java.asc的签名文件。

分发公钥文件

为了让我们的用户能方便的获取我们的公钥文件，对下载下去的文件进行验证，我们需要将公钥文件，发布到公共的公钥服务器上去，如hkp://pgp.mit.edu是美国麻省理工学院提供的公钥服务器。

打开cmd命令行窗口，将目录切换到公钥文件所在的目录，输入如下命令:

Gpg --keyserver hkp://pgp.mit.edu --send-keys 25C6CAD0，将公钥文件分发到公钥服务器

Hkp://pgp.mit.edu是公钥服务器名称

25C6CAD0是我们要发布的公钥id（前面生成的密钥对中的公钥）

显示如下信息，表示发布成功：

gpg: sending key 25C6CAD0 to hkp server pgp.mit.edu

有一点需要说明的是，我们只需要往一个服务器上发布公钥就行，其它公钥服务器会自动同步。

导入公钥服务器上的公钥

我为了验证下载下来的文件是否准确，我们需要先从公钥服务器上下载对应的公钥，导入到本地gpg服务器中，才能使用gpg完成对下载文件的验证。

在cmd命令行窗口中，输入gpg --keyserver hkp://pgp.mit.edu --recv-keys 25C6CAD0，下载25C6CAD0对应的公钥。

显示如下信息：

gpg: requesting key 25C6CAD0 from hkp server pgp.mit.edu

gpg: key 25C6CAD0: "NobleForMvnBook (this is a demo for MvnBook) <3310435058@qq.com>" not changed

gpg: Total number processed: 1

gpg: unchanged: 1

因为我本地已经有这个公钥，所有下载后提示没有改变。

使用公钥验证下载下来的文件

打开cmd命令行窗口，切换到下载文件所在的目录（原始文件和签名文件），输入如下命令 gpg --verify IMvnBookDAO.java.asc，使用签名验证IMvnBookDAO.java文件，显示如下：

gpg: assuming signed data in 'IMvnUserDAO.java'

gpg: Signature made 12/18/16 10:36:47 中国标准时间 using RSA key ID 25C6CAD0

gpg: Good signature from "NobleForMvnBook (this is a demo for MvnBook) <3310435058@qq.com>" [ultimate]

到现在为止，我们基本上完成了GPG的安装、签名、分发和验证的流程。以后的Maven项目就可以直接使用我们现在生成的密钥对对发布文件签名。接下来我们介绍怎么样基于Maven的GPG插件，自动完成构件签名。

基于Maven插件使用GPG

每次手动对Maven构件进行签名，并且将签名部署到Maven仓库中去，是一个烦，还没技术含量的工作，为了从这样简单重复的工作脱离开来，Maven提供了一个叫GPG的插件，能帮我们做所有的事情。我们只需要在pom.xml中做对应的配置，如：

<project>

...

<plugins>

...

<plugin>

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

<artifactId>maven-gpg-plugin</artifactId>

<version>1.6</version>

<executions>

<execution>

<id>signArtifact</id>

<phase>verify</phase>

<goals>

<goal>sign</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

配置好后，使用Mvn命令，就可以完成签名并且发布了。当然有个前提，那就是GPG需要安装好的，也就是说，能在命令行中能执行gpg命令。

当然，在实际项目过程中，对日常的SNAPSHOT构件进行签名就没有太大意义了，而且耗费资源。那有什么办法可以避免这点，只在版本正式发布的时候签名呢？

当然是可以了，在超级pom中有个release-profile，该profile只有在maven属性performRelease为true的时候，才会被激活，而release:perform执行的时候，会把该属性的值设置成true，这个时机刚好是项目进行版本发布的实际。所以，我们可以在settings.xml或pom中创建如下代码，实现只是在发布正式版本的时候，对正式版本进行签名：

<profiles>

<profile>

<id>release-sign-artifacts</id>

<activation>

<property>

<name>performRelease</name>

<value>true</value>

</property>

</activation>

<build>

<plugins>

<plugin>

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

<artifactId>maven-gpg-plugin</artifactId>

<version>1.6</version>

<executions>

<execution>

<id>signArtifact</id>

<phase>verify</phase>

<goals>

<goal>sign</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</profile>

</profiles>

另外需要注意的是，因为Maven Release Plugin有个bug，release:perform执行过程中签名，可能会导致进程永久挂起，为了避免这种情况的发生，我们可以在Maven Release Plugin中，提供一个mavenExecutorId配置，整体样例配置如下：

<build>

<plugins>

<plugin>

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

<artifactId>maven-release-plugin</artifactId>

<version>2.5.3</version>

<configuration>

<tagBase>https://Noble-PC:8443/svn/MvnBookSSMDemo/tags/svnDemo</tagBase>

<branchBase>https://Noble-PC:8443/svn/MvnBookSSMDemo/branches/svnDemo</branchBase>

<username>noble</username>

<password>noble</password>

**<mavenExecutorId>forked-path</mavenExecutorId>**

</configuration>

</plugin>

<plugin>

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

<artifactId>maven-gpg-plugin</artifactId>

<version>1.6</version>

<executions>

<execution>

<id>signArtifact</id>

<phase>verify</phase>

<goals>

<goal>sign</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

到这里，比较规范、自动签名的配置就完成了，当Maven执行release:perform发布项目版本的时候，maven gpg plugin就会自动执行，对构建进行签名。在执行的过程中，会提示输入私钥的密码。

## 8.5 小结

项目开发完了，需要进行版本发布，这章详细介绍了版本发布，包括快照版本同发布版本之间的切换、各种版本的更新以及版本号的内部规则和意义、还有Maven同版本控制系统之间的关系。理解版本转换同scm操作的关系后，还介绍了配置使用Maven Release plugin插件，自动化版本发布等操作。最后，还介绍了在版本发布的时候，使用GPG为构件提供签名，从而为项目开发提供更安全得保证。

# 第九章 Maven核心概念

## 9.1 简介

前面，我们根据项目的进展，介绍了基于Maven的一步步操作，中间根据实际情况的需要，零星的提到了一些概念，比如构建、插件、依赖、发布、安装... ... 等等。

完成前面章节的研究，我们基本上能安装流程，一步步的完成一个项目的从创建工程到测试、安装发布，甚至创建站点报告等所有工作。但是还是有美中不足的，比如为什么这么操作，中间的原理是什么，相关操作的关联是什么，它们的完整体系是怎样的？

等等这些，我们有必要了解清楚了，这样才能更深刻的理解我们按步骤做事情背后的意义，将行动和理论结合起来，使整个工作更完美。

## 9.2 生命周期

### 9.2.1 生命周期简介

在我们研究Maven之前，项目构建的生命周期概念就已经事实存在了，软件开发人员每天都要对项目进行清理、编译、测试、打包以及安装部署。虽然每个软件开发人员都做相关的事情，但公司和公司之间、项目和项目之间，往往目的一样，实现的形式各式各样。有的项目基于IDE工具完成编译、打包和发布，比如MyEclipse和Eclipse Java EE，有些是软件开发人员自己编写脚本，对项目进行自定义构件，比如ant脚本（当然，ant脚本本身也是各写个的，个个不一样）。这些都是特别具有个性化和针对性的，到下一个项目后，又需要改造成新项目所需要的样子。造成了一种事情是一样，又不得不重头再来的囧态：感觉是一样，又不能重用，必须得重写。

Maven通过学习、分析、反思和总结以前工作中对项目的构建过程，抽象出来了一个适合于所有项目的构建生命周期，并且把它们统一规范出来了。具体包括清理、初始化、编译、测试、打包、集成测试、验证、部署和生成站点。这些步骤，几乎时候所有的项目，也就是说，所有项目的管理构建过程，都可以对应到这个生命周期上来。

需要注意的是，Maven中项目的构建生命周期，只是Maven根据实际情况，抽象提炼出来的一个统一标准和规范，它是不能做具体的事情的。也就是说，Maven是没有提供一个编译器，能在编译阶段，帮我们编译源代码的。

既然Maven不做具体事情，那具体事情由谁做呢？好的思想、创意，最终都需要在做具体事件的实践中执行才有结果啦。

是的，Maven只是规定了生命周期的各个阶段和步骤，具体事情，由集成到Maven中的插件完成。比如我们前面介绍的生成站点，就是由maven-site-plugin插件完成的。Maven在项目的构建过程中，只是在方向和步骤上面做了管理和协调的作用。

Maven这种做事的方式，有点像在公司中的报销制度和具体报销。行政或其他部门，根据公司的具体情况，规定好报销的流程。比如软件开发部的报销流程是：报销人领取报销表、填写报销单、组长签字、项目经理签字、人事签字、财务报销。这个流程相当于是Maven中规划好的生命周期，规定了每个阶段和前后关系。哪个开发人员要报销某笔差旅费的话，就必须按对应的流程，找具体的人做具体的事情。比如找行政领取报销单、自己填写报销单、找自己的组长签字等等，这个过程中的人，就是能做具体事情的插件。

当然，如果要专业点的对比的话，Maven的这种做事方式，又同设计模式中的模板模式很像：

Maven就是那个抽象的模板，在里面定义好了每个步骤的抽象方法，在一个执行方法中，按流程完成了这些每个步骤的抽象方法的调用。具体要执行的话，需要写个子类，实现这些步骤的每个方法，在实现里面，自己和调用第三的项目完成具体的功能。这里对每个方法的实现，就可以理解成是插件，前面的抽象方法，可以理解成生命周期的每个阶段，那个完成对每个抽象方法的调用的执行方法，就是Maven对生命周期中每个阶段的管理。

前面是对Maven生命周期的概念性的理解，实际上，Maven就是按这种思路，设计出了一种插件机制，具体实现完成这种思想。Maven在生命周期的每个阶段，都设计了插件接口，我们可以在这接口上，根据项目的实际需要，绑定第三方的插件，做该阶段应该完成的任务，从而保证所有Maven项目的构建过程的标准化，还能让用户自己通过配置第三方的或自己的插件来定义自己个性的构建。当然，Maven对大多数构建阶段，绑定了默认的插件，通过这样的默认绑定，又简化和稳定了实际项目的构建。

### 9.2.2 深入生命周期

前面我们介绍了Maven生命周期的大概概念和思路，在方向上对生命周期有个概括的了解，接下来我们详细介绍Maven生命周期的各个阶段，以及它们的意义和关系。

Maven拥有三套独立的生命周期，它们分别是clean、default和site。Clean生命周期的目的是清理项目；default生命周期的目的是构建项目；site生命周期的目的是建立项目站点。

每个生命周期，又包含多个阶段，这些阶段，在执行的时候，是有固定顺序的，后面的阶段，一定要等前面的阶段执行完后才能被执行，我们同Maven，就可以通过调用这么生命周期的阶段，来完成相互沟通。比如clean生命周期，它就包含有pre-clean、clean和post-clean三个阶段。我们调用pre-clean的时候，只有pre-clean阶段被执行；我们调用clean的时候，先执行pre-clean，再执行clean阶段；同样，当我们调用post-clean的时候，Maven自动先执行pre-clean、再执行clean，最后执行post-clean。

相对于生命周期每个阶段，执行有固定的前后顺序，并且后面阶段的执行，一定要有前面阶段已经执行的前提，clean、default和site这三套生命周期直接，倒是相互独立，不存在谁是谁的前提。比如我们执行clean，同是否有执行default和site没有关系，同样的，执行default，一样同是否有执行clean和site没有关系。

接下来我们详细介绍每套生命周期的各个阶段。

1. clean生命周期

Clean生命周期的目的是清理项目，它包括下面三个阶段：

1. pre-clean 执行清理前需要完成的工作
2. clean 清理上一次构建过程中生成的文件，比如编译后的class文件等
3. post-clean 执行清理后需要完成的工作
4. Default生命周期

Default生命周期定义了构建项目时所需要的执行步骤，它是所有生命周期中最核心部分，包含的阶段如下，比较常用重点的阶段，我们这里用粗体标记：

1. Validate 验证项目结构是否正常，必要的配置文件是否存在
2. Initialize 做构建前的初始化操作，比如初始化参数、创建必要的目录等等
3. generate-sources 产生在编译过程中需要的源代码
4. process-sources 处理源代码，比如过滤值
5. **generate-resources** 产生主代码中的资源在classpath中的包
6. **process-resources** 将资源文件copy到classpath的对应包中
7. **Compile** 编译项目中的源代码
8. process-classes 产生编译过程中生成的文件
9. generate-test-sources 产生编译过程中测试相关的代码
10. process-test-srouces 处理测试代码
11. **generate-test-resources** 产生测试中资源在classpath中的包
12. **process-test-resources** 将测试资源copy到classpath中
13. **test-compile** 编译测试代码
14. process-test-classes 产生编译测试代码过程的文件
15. **Test** 运行测试案例
16. prepare-package 处理打包前需要初始化的准备工作
17. Package 将编译后的class和资源打包成压缩文件，比如rar
18. pre-integration-test 做好集成测试前的准备工作，比如集成环境的参数设置
19. integration-test 集成测试
20. post-integration-test 完成集成测试后的收尾工作，比如清理集成环境的值
21. Verify 检测测试后的包是否完好
22. **Install** 将打包的组件，以构件的形式，安装到本地依赖仓库中，以便共享给本地的其它项目
23. **Deploy** 运行集成和发布环境，将测试后的最终包，以构件的方式发布到远程仓库中，一遍所有程序员可以共享

这些阶段的详细介绍内容，我们可以参考链接：

<http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

1. Site生命周期

Site生命周期的目的是建立和发布项目站点，Maven可以基于pom所描述的信息，自动生成项目的站点，同时还可以根据需要，生成相关的报告文档，集成在站点中，方便团队交流和发布项目信息。Site生命周期包括如下结果阶段：

1. pre-site 执行生成站点前需要做的准备工作
2. Site 生成站点文档
3. post-site 执行生成站点后需要收尾的工作
4. site-deploy 将生成的站点，发布到服务器上

### 9.2.3 调用生命周期阶段

前面介绍了每套生命周期的各个阶段，那我们怎么样通知maven执行生命周期的哪个阶段呢？

有两种方式可以同Maven进行交互，一种是用mvn命令后面指定阶段名称；一种是在m2eclipse中，有对应的Run As菜单命令。

其实这两种方式，我们都有在前面的章节中介绍过怎么使用，只是前面只是介绍的怎么操作，没有对应上具体的理论。接下来我们分别梳理一下这两种方式：

1. mvn命令行指定执行周期阶段

这种方式，都是在cmd窗口中执行的，前提条件是要配置好安装的maven环境变量（path），并且将当前目录切换到了maven工程目录下。后面的每个命令的例子，都是基于  
MvnBookSSMDemo.Service.Impl工程进行的，它的当前目录是  
E:\temp\demoMaven\MvnBookSSMDemo.Service.Impl。

1. mvn clean：调用clean生命周期的clean阶段，实际执行的是clean生命周期的pre-clean和clean阶段。如图9-1：

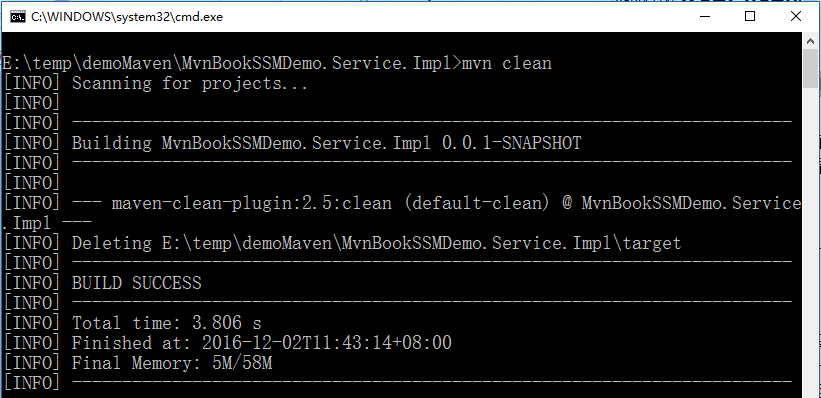


图9-1 mvn clean提示

1. mvn test：该命令调用default生命周期中的test阶段。实际执行的阶段包括validate、initialize、generate-sources...compile... test-compile、process-test-classes、test，也就是把default生命周期中，从开始到test的所有阶段都执行完了，而且是按顺序。最后运行效果图9-2如下：

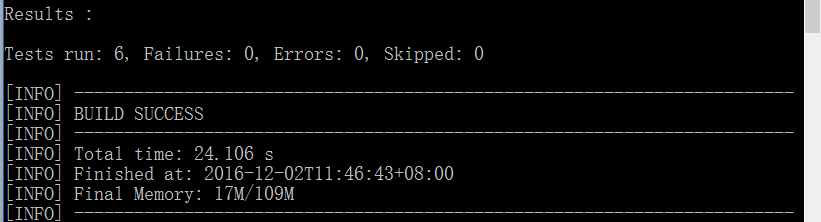


图9-2 mvn test提示

1. mvn clean install：该命令调用clean生命周期的clean阶段和default生命周期的install阶段。实际执行clean生命周期中的pre-clean、clean两个阶段，和default生命周期中从开始的validate到install的所有阶段。该命令结合了两个生命周期，在实际项目构建中，每执行一个行的构建，先清理一下以前构建的旧文件，是一个比较好的习惯。效果图9-3如下：

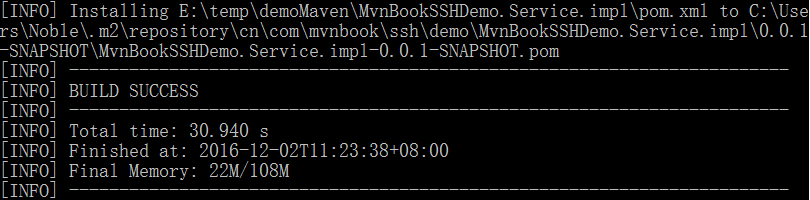


图9-3 mvn clean install提示

1. mvn clean deploy site-deploy：该命令调用clean生命周期中的pre-clean、clean阶段，default生命周期中从validate到deploy的所有阶段，以及site生命周期中的pre-site、site、post-site和site-deploy阶段。最后效果是把该项目编译好、测试、打包发布到远程仓库，同时还将生成好的站点发布到了站点服务器。命令执行后的效果是：
2. 命令行效果图9-4

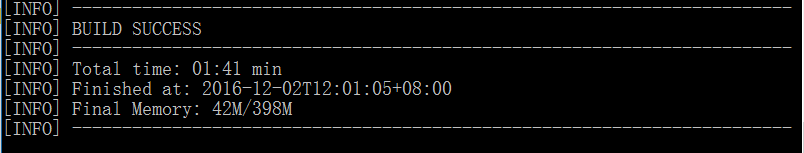


图9-4 mvn clean deploy site-deploy提示

1. Archiva私服构件查询如图9-5

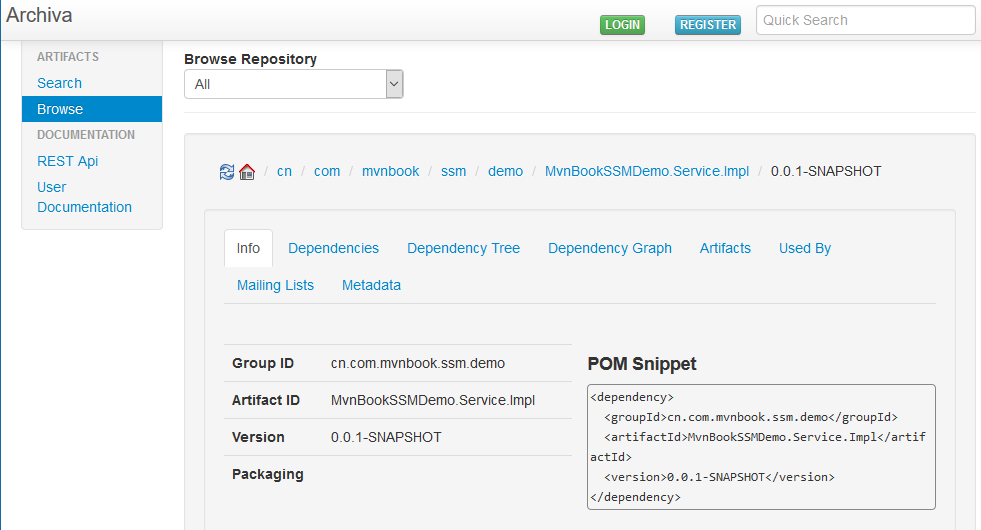


图9-5 私服发布构件

1. 站点服务器访问页面如图9-6：



图9-6 发布到服务器上的站点

1. M2eclipse指定执行周期阶段

在Eclipse里面，基于m2eclipse执行生命周期的阶段思路，同前面基于命令操作是一样的，不同的是m2eclipse帮我们用图形化界面的形式展现出来了，操作起来比较人性化。具体操作如下：

右击工程，选择Run As，后面就会显示常用的Maven执行生命周期阶段的命令，如图9-7：

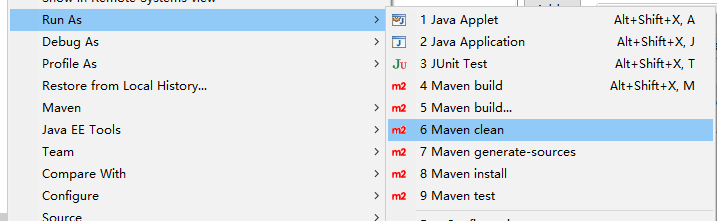


图9-6 m2eclipse运行选项

其中Maven clean，就等同命令行的mvn clea；Maven generate-sources，等同命令行的mvn generate-sources；Maven install，等同命令行的 mvn install；Maven test，等同命令行的mvn test。除了这些常用的命令，我们可以直接选择执行外，我们也可以自定义自己的Maven执行菜单命令。

点击上图的Maven build...，出现如下图9-7界面：

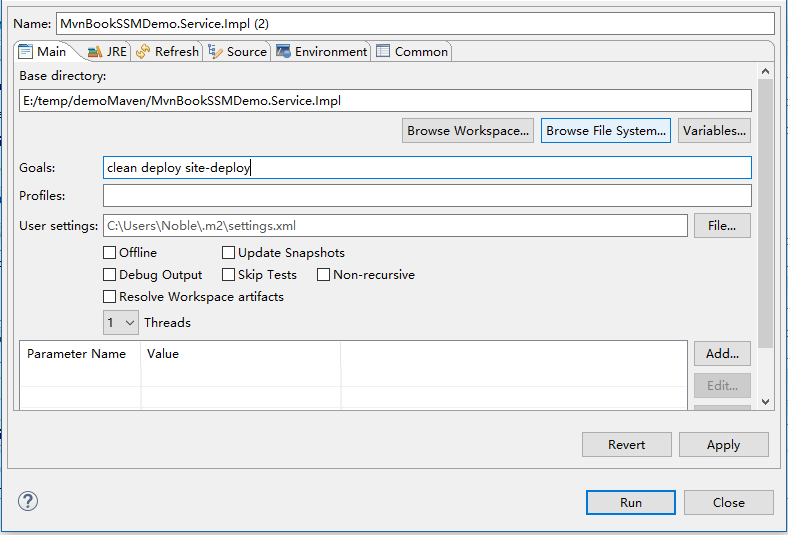


图9-7 Maven Build... 配置目标

在Goals后面输入框中，输入我们要执行的阶段名称，如图上显示，点击Run，同执行mvn clean deploy site-deploy的效果一样。

而且首次运行完后，以后选择 Run As->Maven build选项，可以选择前面有输入的生命周期阶段重复执行。

## 9.3 插件

### 9.3.1 插件的作用和目标

通过对Maven生命周期的了解，我们知道Maven只是对项目的构建过程进行了统一的抽象定义和管理，至于每个阶段由谁来做，Maven自己不去实现，而是丢给对应的插件去完成。这就是插件的作用，完成构建过程中某个或某几个阶段的任务。比如maven-compile-plugin，就可以完成在compile阶段的，java源代码的编译任务。

但是从插件本身来说，为了能够代码重用，同时有更多功能的封装，一个插件可以实现生命周期多个阶段的任务，比如maven-dependency-plugin就可以实现十多个功能：分析项目的依赖功能；列出项目的依赖树；分析依赖的来源等等。为方便指定执行插件的某个功能，我们将插件的每个功能叫目标。这样我们就可以实现 在哪个阶段，执行哪个插件的哪个目标。比如 dependency:analyze，表示maven-dependency-plugin的分析目标；dependency:tree表示maven-dependency-plugin的列出依赖的目标等等。

### 9.3.2 插件同生命周期阶段的绑定

前面有对插件的作用和目标进行了解了，但是我们最终的目标是要将插件等功能集成到maven生命周期的相关阶段中去，让maven构建工程的时候，构建到相关阶段的时候，自动调用插件完成指定的任务。那我们怎样让maven的生命周期同插件实现相互绑定，来完成实际的构建任务呢？

比如maven-compile-plugin插件的compile目标能完成编译任务，而且Maven的default生命周期的compile阶段，定义好要实现源代码编译的，我们怎么样将maven-compile-plugin插件的compile目标，绑定到default生命周期的compile阶段，让maven构建项目，到compile阶段的时候，能自动执行插件的compile目标呢？

实现生命周期的阶段同插件目标的绑定，一共有两种方式：内置绑定和自定义绑定。接下来我们分别研究一下这两种绑定。

1. 内置绑定

为了让用户使用Maven方便，少配置甚至不用配置，就可以使用Maven构建项目，Maven安装好后，自动为生命周期的主要阶段，绑定了很多插件的目标。当我们通过命令或图形界面，执行生命周期的某个阶段的时候，对应的插件目标就会自动执行，从而完成既定的任务。

Maven-clean-plugin插件有个目标叫clean，它的作用是删除项目构建的时候，产生的输出目录。Maven-clean-plugin的clean目标，就被默认绑定在clean生命周期的clean阶段，也就是说，当我们执行 mvn clean的时候，maven在clean阶段，自动调用maven-clean-plugin的clean目标，删除构建的输出目录。

同样的，maven-site-plugin插件有两个目标：site目标，用来生成项目站点；deploy目标，用来将生成的站点，发布到站点服务器上去。Maven默认将site目标，绑定在site生命周期的site阶段，将deploy目标，绑定在site生命周期的site-deploy阶段。

Default生命周期就比较复杂了，不仅仅复杂在有那么多的阶段，还复杂在java可以打成不同的包：jar、war和ear等包，不同包在不同的阶段是不一样的绑定。想想都不想研究太深了，详细情况，请参考下面链接文档的介绍：  
<http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

我们这，就简单复制了一下该文档上面列出的阶段和插件目标的对应表：

1. jar包war包的绑定

|  |  |
| --- | --- |
| process-resources | resources:resources |
| compile | compiler:compile |
| process-test-resources | resources:testResources |
| test-compile | compiler:testCompile |
| test | surefire:test |
| package | ejb:ejb *or* ejb3:ejb3 *or* jar:jar *or* par:par *or* rar:rar *or* war:war |
| install | install:install |
| deploy | deploy:deploy |

1. Ear包绑定

|  |  |
| --- | --- |
| generate-resources | ear:generate-application-xml |
| process-resources | resources:resources |
| package | ear:ear |
| install | install:install |
| deploy | deploy:deploy |

1. Maven-plugin包绑定

|  |  |
| --- | --- |
| generate-resources | plugin:descriptor |
| process-resources | resources:resources |
| compile | compiler:compile |
| process-test-resources | resources:testResources |
| test-compile | compiler:testCompile |
| test | surefire:test |
| package | jar:jar *and* plugin:addPluginArtifactMetadata |
| install | install:install |
| deploy | deploy:deploy |

1. Pom包绑定

|  |  |
| --- | --- |
| package | site:attach-descriptor |
| install | install:install |
| deploy | deploy:deploy |

当然，我们也可以通过运行maven命令，在命令提示信息中，查看到阶段和插件目标的绑定。比如在cmd命令行窗口中敲maven compile，我们可以查看到如下日志：

[INFO] Scanning for projects...

[INFO]

[INFO] ------------------------------------------------------------------------

[INFO] Building MvnBookSSMDemo.Service.Impl 0.0.1-SNAPSHOT

[INFO] ------------------------------------------------------------------------

[INFO]

[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ MvnBookSSMDemo.Service.Impl ---

[INFO] Using 'UTF-8' encoding to copy filtered resources.

[INFO] skip non existing resourceDirectory E:\temp\demoMaven\MvnBookSSMDemo.Service.Impl\src\main\resources

[INFO]

[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ MvnBookSSMDemo.Service.Impl ---

[INFO] Nothing to compile - all classes are up to date

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 2.782 s

[INFO] Finished at: 2016-12-02T16:59:28+08:00

[INFO] Final Memory: 11M/108M

[INFO] ------------------------------------------------------------------------

根据日志，我们可以查看出defaut-resources阶段，绑定的是maven-resources-plugin:2.6:resources目标，default-compile阶段，绑定的是maven-compile-plugin:3.1:compile目标。

自定义绑定

除了Maven内置的绑定外，我们也可以指定在某个阶段，绑定某个插件的哪个目标，这样就使得maven在构建项目的时候，能执行我们需要的更多特殊任务。

比如有时候我们希望在构建工程的时候，能将源代码打成jar包(安装jdk的时候，我们是可以选择安装src.jar的，这样我们可以学习jdk api的源代码)，这样的任务，maven是没有帮我们内置绑定到生命周期的阶段上去的。这就需要我们自己配置了。Maven-source-plugin里面有个jar-no-forkmub ，能帮我们将项目中的主代码，打成jar文件，我们就可以将该目标，绑定到default生命周期的指定阶段是上去，比如verify阶段就可以的。这样就可以在执行完集成测试后，安装构建前，创建源代码jar包。这样的配置，我们可以加在pom.xml中，参考pom.xml配置代码如下：

<project>

...

<build>

<plugins>

...

<plugin>

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

<artifactId>maven-source-plugin</artifactId>

<version>3.0.0</version>

<executions>

<execution>

<id>att-sources</id>

<phase>verify</phase>

<goals>

<goal>jar-no-fork</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

...

</project>

在build下的plugins里面配置了一个插件，名称叫maven-source-plugin，它的groupId为org.apache.maven.plugins,artifactId为maven-source-plugin，version为3.0.0。这里需要说明一下的是，我们自定义使用插件，尽量使用费快照版本，这样就可以避免因为插件的版本的不稳定，从而影响构建的不稳定。

除了要指定要绑定的插件外，我们还需要通过executions下的execution子元素，配置执行任务，指定任务的id和任务要执行的目标，还有绑定到生命周期的哪个阶段。phrase元素配置的是绑定的阶段（verify），goals中的goal描述的目标jar-no-fork。

在cmd命令行窗口中输入 mvn verify命令，我们可以看到如下信息输出：

[INFO]

[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ MvnBookSSMDemo.Service.Impl ---

[INFO]

[INFO] --- maven-source-plugin:3.0.0:jar-no-fork (att-sources) @ MvnBookSSMDemo.Service.Impl

最后表示执行了id为att-sources，插件目标为maven-source-plugin:3.0.0:jar-no-fork的任务。在工程的target目录下，会发现有个  
MvnBookSSMDemo.Service.Impl-0.0.1-SNAPSHOT-sources.jar文件，这里面就是工程中的所有主代码。

当然，如果我们比较细心的话，会发现不在phase中指定要绑定的生命周期阶段，也会执行同样的结果。比如我们把pom.xml文件中的phase这行删除，再执行一次mvn verify命令，会发现同样在verify阶段，执行maven-source-plugin:jar-no-fork目标。为什么呢？原因是很多插件的目标，在插件编写的时候，就已经指定了默认的绑定阶段。虽然我们没有在execution中用phase元素指定绑定目标的阶段，它一样的可以默认绑定。

为了了解插件绑定的默认生命周期阶段，可以运行如下命令查看：

mvn help:describe -Dplugin=org.apache.maven.plugins:maven-source-plugin:3.0.0 -Ddetail

我们可以查看到该插件的详细信息。在这些信息中，我们可以看到关于jar-no-fork的描述如下：

...

source:jar-no-fork

Description: This goal bundles all the sources into a jar archive. This

goal functions the same as the jar goal but does not fork the build and is

suitable for attaching to the build lifecycle.

Implementation: org.apache.maven.plugins.source.SourceJarNoForkMojo

Language: java

Bound to phase: package

...

通过Bound to phase:package可以了解到，jar-no-fork默认绑定的生命周期阶段是package

最后需要说明一下的是，我们在给不同的生命周期阶段绑定不同的插件目标后，这些目标的执行，自然是按阶段的顺序逐个执行，如果我们在一个阶段上，绑定了多个目标，效果会如何呢？很简单，都执行，而且是按插件声明的顺序执行！

### 9.3.3 插件参数配置

完成插件目标同生命周期的阶段的绑定后，maven在构建工程时，在构建到对应阶段后，会自动执行绑定插件的目标任务了，但是有很多情况，我们需要给即将执行的目标制定参数，让执行的任务更加适合当前项目的需要，而且几乎所有的maven插件目标，都有一些参数可以设置。我们可以通过命令行和pom配置两种方式，给这些目标设置比较合适的参数值。接下来我们分别介绍这两种配置参数的方式。

1. 命令行配置参数

在maven命令中，使用-D后面接参数名称=参数值的方式，配置目标参数。

比如maven-surefire-plugin插件中，提供了一个maven.test.skip参数，当他的值为true的时候，就不会执行test案例。具体语法是：

Mvn install -Dmaven.test.skip=true

执行这命令，我们可以从输入的info信息中查看到，没有测试案例部分信息，也就是没有执行测试。

1. Pom配置参数

对于有些参数，项目创建好后，目标每次执行的时候都不需要改变，这时候比较好的方式是把这些值配置pom.xml中，这样就省去每次构建的时候，都需要输入的麻烦。

通过在命令行中输入：

Mvn help:describe -Dplugin=org.apache.maven.plugins:maven-compiler-plugin:3.5.1 -Ddetail命令，我们会发现compile目标中有一堆参数，其中有如下内容：

source (Default: 1.5)

User property: maven.compiler.source

The -source argument for the Java compiler.

staleMillis (Default: 0)

User property: lastModGranularityMs

Sets the granularity in milliseconds of the last modification date for

testing whether a source needs recompilation.

target (Default: 1.5)

User property: maven.compiler.target

The -target argument for the Java compiler.

这里有source和target两个参数的介绍，我们可以通过在pom.xml中，做如下配置，指定这两个参数的值：

<project>

...

<build>

<plugins>

<plugin>

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

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

<version>3.5.1</version>

<configuration>

<source>1.5</source>

<target>1.5</target>

</configuration>

</plugin>

...

</plugins>

</build>

...

</project

在configuration元素作用，用参数名称做为子元素，配置对应参数的值

<source>1.5</source> <target>1.5</target>，通过这两个配置，指定编译java1.5的源代码，生成于JVM1.5兼容的字节码文件，也就是class文件。

当然，前面这种配置，是给maven-compiler-plugin插件配置的一个全局参数值，也就是说不管是使用maven-compiler-plugin编译工程代码，还是测试代码或其他的，它们都会使用source=1.5，target=1.5这两个值。那如果我们需要给特点的任务指定特定的值怎么办呢？很简单，直接在我们配置任务的execution中，以前我们了解到可以通过id指定任务id，phase指定绑定的生命周期阶段，goals中的goal指定插件的目标外，还可以直接在execution中添加configuration元素，内容同前面的一样：

<configuration>

<source>1.5</source>

<target>1.5</target>

</configuration>

这样的两值，就只对当前任务有效了。

### 9.3.4 获取插件信息

到现在为止，我们基本上掌握了怎么配置使用插件，但是仅仅是如何配置实用插件是不够完美的。因为毕竟我们在书本上介绍的插件是有限的，而且对每个插件的使用，只是根据需要，使用的有限的，有代表意义的目标。在实际项目中，我们可能需要使用更多更合适的插件，那到底要使用哪些插件呢？所以首先得找到自己认为合适的插件，再了解这些插件的合适配置情况，还有相关参数的详细情况。

由于插件非常多，而且随着不同个人和不同组织的分享，插件的数量每天都在增加。加上大部分插件都没有完善的帮助文档，造成了我们要找到一个正确的插件，并且很好的配置起来使用，其实也不是一件简单容易的事情。

接下来我们介绍一下查找插件信息的地方和方法，大家可以以它们为参考，找到自己合适的插件配置使用，并且积累下来。

1. 在线查找插件

目前插件基本上都来源于两个地方，一个是Apache，还一个就是Codehaus。鹦鹉Maven本身就是Apache软件基金会的，所有在Apache上有很多Maven的官方插件，而且每天有很多人在使用这些插件，这些插件都经过了很多项目实际考验的，所以它们比较文档可靠。通过访问http://maven.apache.org/plugins/index.html页面，我们可以看到所有插件的列表信息，点击进去，可以进一步了解每个插件的详细信息，当然，我们也可以通过http://maven.-  
apache.org/maven2/org/apache/maven/pugins/链接下载中意的插件。

处理Apache官方插件外，托管在Codehaus上的Mojo项目，也提供了大量丰富的Maven插件，我们可以通过http://mojo.codehaus.org/plugins.html访问详细列表。同样，我们可以通过http://repository.codehaus.org/org/codehaus/mojo下载找到的插件。有点缺陷的是，这些插件的文档和可靠性相对不是很好，在使用过程中，如果遇到问题，往往需要自己查看源代码确认修复。

当然，我们在附录里面，也有整理了一些常用的插件，大家可以结合附录介绍和Apache以及Codehaus上的文档介绍，综合的把握和配置使用对应的插件。

1. 使用maven-help-plugin查看插件

除了通过访问在线文档，了解某个插件的详细信息外，其实我们还可以借助maven-help-plugin插件来获取插件的详细信息。比如在cmd命令行窗口中运行如下命令：

Mvn help:describe -Dplugin=org.apache.maven.plugins:maven-site-plugin:3.4 -Ddetail

我们就可以查看到maven-site-plugin插件3.4版本的详细信息，内容很多，如下列出的是开始的几个基本信息：

Name: Apache Maven Site Plugin

Description: The Maven Site Plugin is a plugin that generates a site for the

current project.

Group Id: org.apache.maven.plugins

Artifact Id: maven-site-plugin

Version: 3.4

Goal Prefix: site

This plugin has 9 goals:

site:attach-descriptor

Description: Adds the site descriptor (site.xml) to the list of files to be

installed/deployed.

For Maven-2.x this is enabled by default only when the project has pom

packaging since it will be used by modules inheriting, but this can be

enabled for other projects packaging if needed.

This default execution has been removed from the built-in lifecycle of

Maven 3.x for pom-projects. Users that actually use those projects to

provide a common site descriptor for sub modules will need to explicitly

define this goal execution to restore the intended behavior.

Implementation: org.apache.maven.plugins.site.SiteDescriptorAttachMojo

Language: java

Bound to phase: package

当然，如果我们不想查看太多，只是想具体了解插件的某个目标，可以用-Dgoal=目标的方式，查看指定目标的信息，比如运行如下命令，我们可以查看site插件的site目标信息。

Mvn help:describe -Dplugin=site -Dgoal=site -Ddetail

输出信息：

[INFO] Mojo: 'site:site'

site:site

Description: Generates the site for a single project.

Note that links between module sites in a multi module build will not work,

since local build directory structure doesn't match deployed site.

Implementation: org.apache.maven.plugins.site.render.SiteMojo

Language: java

Available parameters:

attributes

Additional template properties for rendering the site. See Doxia Site

Renderer.

generatedSiteDirectory (Default: ${project.build.directory}/generated-site)

Directory containing generated documentation. This is used to pick up

other source docs that might have been generated at build time.

generateProjectInfo (Default: true)

User property: generateProjectInfo

Whether to generate the summary page for project reports:

project-info.html.

generateReports (Default: true)

User property: generateReports

Convenience parameter that allows you to disable report generation.

generateSitemap (Default: false)

User property: generateSitemap

Generate a sitemap. The result will be a 'sitemap.html' file at the site

root.

inputEncoding (Default: ${project.build.sourceEncoding})

User property: encoding

Specifies the input encoding.

locales (Default: en)

User property: locales

A comma separated list of locales to render. The first valid token will

be the default Locale for this site.

moduleExcludes

Module type exclusion mappings ex: fml -> \*\*/\*-m1.fml (excludes fml files

ending in '-m1.fml' recursively)

The configuration looks like this:

<moduleExcludes>

<moduleType>filename1.ext,\*\*/\*sample.ext</moduleType>

<!-- moduleType can be one of 'apt', 'fml' or 'xdoc'. -->

<!-- The value is a comma separated list of -->

<!-- filenames or fileset patterns. -->

<!-- Here's an example: -->

<xdoc>changes.xml,navigation.xml</xdoc>

</moduleExcludes>

outputDirectory (Default: ${project.reporting.outputDirectory})

User property: siteOutputDirectory

Directory where the project sites and report distributions will be

generated.

outputEncoding (Default: ${project.reporting.outputEncoding})

User property: outputEncoding

Specifies the output encoding.

relativizeDecorationLinks (Default: true)

User property: relativizeDecorationLinks

Make links in the site descriptor relative to the project URL. By

default, any absolute links that appear in the site descriptor, e.g.

banner hrefs, breadcrumbs, menu links, etc., will be made relative to

project.url.

Links will not be changed if this is set to false, or if the project has

no URL defined.

saveProcessedContent

Whether to save Velocity processed Doxia content (\*.<ext>.vm) to

${generatedSiteDirectory}/processed.

siteDirectory (Default: ${basedir}/src/site)

Directory containing the site.xml file and the source for hand written

docs (one directory per Doxia-source-supported markup types): see Doxia

Markup Languages References).

skip (Default: false)

User property: maven.site.skip

Set this to 'true' to skip site generation and staging.

templateFile

User property: templateFile

The location of a Velocity template file to use. When used, skins and the

default templates, CSS and images are disabled. It is highly recommended

that you package this as a skin instead.

validate (Default: false)

User property: validate

Whether to validate xml input documents. If set to true, all input

documents in xml format (in particular xdoc and fml) will be validated

and any error will lead to a build failure.

xdocDirectory (Default: ${basedir}/xdocs)

Alternative directory for xdoc source, useful for m1 to m2 migration

Deprecated. use the standard m2 directory layout

说明：

命令中的 -Dplugin=site，通过指定插件的前缀，来指定要查看的插件名称。同写成-Dplugin=org.apache.maven.plugins:maven-site-plugin:3.6是一样的意思

-Dgoal=site 指定要查看的目标，名称是site

-Ddetail 表示要查看详细信息。

### 9.3.5 调用插件

一般我们是通过Maven在构建工程的时候，自己调用执行配置好的插件目标任务的，当然，我们也可以用命令行之间执行插件。比如我们前面查看插件信息的命令，就是在调用help插件的describe目标，来完成的查看目标查看信息的任务的。同样，我们运行如下命令：

Mvn dependency:tree

就是在命令行中执行maven-dependency-plugin的tree目标，列出当前项目的依赖树，结果内容如下：

[INFO] cn.com.mvnbook.ssm.demo:MvnBookSSMDemo.Service.Impl:jar:0.0.1-SNAPSHOT

[INFO] +- cn.com.mvnbook.ssh.demo:MvnBookSSHDemo.DAO:jar:0.0.1-SNAPSHOT:compile

[INFO] +- cn.com.mvnbook.ssh.demo:MvnBookSSHDemo.Service:jar:0.0.1-SNAPSHOT:compile

[INFO] +- cn.com.mvnbook.ssm.demo:MvnBookSSMDemo.DAO.MyBatis:jar:0.0.1-SNAPSHOT:test

[INFO] | +- cn.com.mvnbook.pom:SpringPOM:pom:0.0.1-SNAPSHOT:test

[INFO] | +- org.mybatis:mybatis:jar:3.4.0:test

[INFO] | +- org.mybatis:mybatis-spring:jar:1.3.0:test

[INFO] | +- org.mybatis.generator:mybatis-generator-core:jar:1.3.2:test

[INFO] | +- commons-dbcp:commons-dbcp:jar:1.4:test

[INFO] | | \- commons-pool:commons-pool:jar:1.5.4:test

[INFO] | \- mysql:mysql-connector-java:jar:5.1.34:test

[INFO] +- junit:junit:jar:4.7:test

[INFO] +- org.springframework:spring-core:jar:4.2.7.RELEASE:compile

[INFO] | \- commons-logging:commons-logging:jar:1.2:compile

[INFO] +- org.springframework:spring-aop:jar:4.2.7.RELEASE:compile

[INFO] | \- aopalliance:aopalliance:jar:1.0:compile

[INFO] +- org.springframework:spring-beans:jar:4.2.7.RELEASE:compile

[INFO] +- org.springframework:spring-context:jar:4.2.7.RELEASE:compile

[INFO] | \- org.springframework:spring-expression:jar:4.2.7.RELEASE:compile

[INFO] +- org.springframework:spring-context-support:jar:4.2.7.RELEASE:compile

[INFO] +- org.springframework:spring-web:jar:4.2.7.RELEASE:compile

[INFO] +- org.springframework:spring-webmvc:jar:4.2.7.RELEASE:compile

[INFO] +- org.springframework:spring-aspects:jar:4.2.7.RELEASE:compile

[INFO] | \- org.aspectj:aspectjweaver:jar:1.8.9:compile

总结起来，使用命令行执行maven插件的语法如下：

maven <插件名称|前缀>:<目标> [-D参数名=参数值 ...]

### 9.3.6 解析插件

前面一节，我们输入mvn dependency:tree命令，查看当前工程的依赖树，用到的是depedency插件实现的。有没有感觉到疑惑，我们说插件，使用插件，一般要指定插件的坐标信息（groupId、artifactId、version）才能唯一指定一个插件的，为什么我们这里只是输入了dependency就可以指定是使用的maven-dependency-plugin插件呢？

其实这是Maven为了方便我们，提供的一种简单的方式，使用插件的前缀，来指定插件。接下来我们详细介绍一下Maven的运行机制，来从本质上把握mvn命令的语法。

插件仓库

同依赖构件一样，插件构件也是基于坐标存储在Maven仓库中的。在需要的时候，Maven先从本地仓库中查找插件，如果不存，就从远程仓库查找。找到插件后，下载到本地仓库使用。

需要注意的是，Maven会区别对待依赖的远程仓库和插件的远程仓库。我们以前在setting中配置的远程仓库，只是Maven用来找依赖的，而插件的远程仓库，是内置的，一般直接在http://repo1.maven.org/maven2中都可以找到，要自己单独配置插件的远程仓库的话，需要使用pluginRepositories->pluginRepository进行配置。比如下面配置代码，是默认插件仓库的配置

<pluginRepositories>

<pluginRepository>

<id>central</id>

<name>Maven Plugin Repository</name>

<url>http://repo1.maven.org/maven2</url>

<layout>default</layout>

<snapshots>

<enabled>false</enabled>

</snapshots>

<releases>

<updatePolicy>never</updatePolicy>

</releases>

</pluginRepository>

</pluginRepositories>

如果我们实在是在中央仓库中，找不到需要的插件，可以模仿上面的代码，配置自己的远程插件仓库。

插件默认的groupId

在我们使用插件，或者在pom中配置插件的时候，如果使用的是Maven官方插件的话，是可以不指定groupId的，因为这些插件的groupId都是一样的，都是org.apache.maven.plugins。

Maven对于官方groupId是可以不在配置文件中明确配置的，也就是说，如果我们没有配置指定groupId的话，maven默认认为是org.apache.maven.plugins。这样就可以简化我们部分的配置。

当然为了保险起见，建议大家不要太嫌麻烦，还是自己指定groupId比较直观些。

解析插件的版本

同样的目的，为了简化插件的配置和使用，我们可以不指定插件的版本。

如果没有指定插件的版本的话，maven对版本出来的方式有两个：

使用maven超级pom中，给核心插件设置的版本

如果插件不属于核心插件范畴，maven会去检测所有仓库中的版本，最终选择lastest版本。也就是说选择最新版本，而且这个最新版本，不排除是快照版本。我们是知道的，快照版本是有稳定性缺陷的。

解析插件的前缀

前面有提到，为了简化我们对插件的调用，我们可以在命令行中使用前缀指明要执行的插件，现在解释一下，Maven是如果根据插件的前缀，找到真正的插件的。

插件前缀与groupId:artifactId是一一对应的，这种对应关系，同DNS服务器中，域名同ip地址是一一对应的一样，这种对应关系保存在仓库的元数据中，这样的元数据为groupId/maven-metadata.xml。要准确的解析到插件，我们还需要解释一下这里的groupId。前面我们有介绍过，目前绝大部分插件都是放在http://repo1.maven.org和http://repository.codehaus.org中的，它们的groupId对应的是arg.apache.maven.plugins和org.-codehaus.mojo，Maven在解析插件仓库元数据的时候，会默认使用org.apache.maven.plugins和org.codehaus.mojo两个groupId，也就是说，Maven会自动检测http://repo1.maven.org/maven2/org/apache/maven/plugins/maven-metadata.xml和http://rep-  
sitory.codehaus.org/org/codehaus/mojo/maven-metadata.xml中的元数据。当然，我们也可以告诉Maven从其他仓库中查找，只要在settings.xml中做如下配置：

<settings>

<pluginGroups>

<pluginGroup>cn.com.demo.plugins</pluginGroup>

</pluginGroups>

</settings>

这样配置后，Maven不仅仅会检测org/apache/maven/plugins/maven-metadata.xml、org/codehaus/mojo/maven-metadata.xml，还会检测cn/com/demo/plugins/maven-metadata.xml。

那插件仓库数据的内容是什么样的呢，下面列出了org.apache.mave.plugins中groupId部分的内容：

<metadata>

<plugins>

<plugin>

<name>Maven Clean Plugin</name>

<prefix>clean</prefix>

<artifactId>maven-clean-plugin</artifactId>

</plugin>

<plugin>

<name>Maven Compiler Plugin</name>

<prefix>compiler</prefix>

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

</plugin>

<plugin>

<name>Maven Dependency Plugin</name>

<prefix>dependency</prefix>

<artifactId>maven-dependency-plugin</artifactId>

</plugin>

...

</plugins>

</metadata>

从上面的内容我们可以发现，maven-clean-plugin的前缀为clean，maven-compiler-  
plugin前缀是compiler，maven-dependency-plugin的前缀是dependency。当Maven解析到compiler:compile命令后，它首先基于默认的groupId归并所有插件仓库的元数据arg/apache/maven/plugins/maven-metadata.xml;接着检查归并后的元数据，找到对应的artifactId为maven-compiler-plugin；接下来再结合当前元数据的groupId为org.apache.maven.plugins，最后找到仓库中最新的version，从而就可以得到一个插件的完整坐标信息。如果第一个metadata.xml中没有找到目标插件，就用同样的流程，找其它的metadata.xml，包括我们自己定义的metadata.xml。如果所有的地方都没有找到对应的前缀，这就以报错的形式结束了。

## 9.4 坐标

通过前面内容的学习和练习，我们发现要完成一个项目的开发和构件，总是要使用到地方的内容（我们叫构件），而且这些构件已经被我们的Maven仓库管理好了，不管是在本地仓库、私服还是远程仓库、中央仓库中。总之，就是被仓库管理了。也就是说，这些仓库中有很多构件需要管理，那我们要中间的哪些，具体的哪几个，就需要先从里面精确的找到，才能够使用起来。

那Maven是通过什么方式精确的找到自己想要的构件的呢？

其实怎么找到，前面已经介绍了，那就是通过构件的坐标去唯一定位查找的。反过来也就是说，Maven仓库中，是用坐标为标记，一一对应的管理好每个构件的。

这就是坐标的作用，它用来唯一标记仓库中的每个构件，通过这些标记，我们可以唯一的从仓库中找到对应的构件，从而使用起来。

咱们知道构件的作用了，它就是唯一对应构件的这么一个数据或一组值，那坐标由哪些信息组成呢？

一个完整的坐标信息，它由groupId、artifactId、version、packaging、classifier组成，如下是一个简单的坐标定义：

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>4.2.7.RELEASE</version>

<packaging>jar</packaging>

这是junit的坐标，下面详细介绍一下各个元素：

1. groupId

定义当前Maven项目从属的实际项目。关于groupId的理解：

1. Maven项目和实际项目不一定是一一对应的，比如SpringFramework这实际项目，它对应的Maven项目就有很多，如spring-core、spring-context、spring-security等等。造成这样的原因，是Maen中有模块的概念，所以一个实际项目经常会被划分成很多模块。
2. groupId不应该同开发项目的公司或组织对应，原因比较好理解，一个公司和一个组织会开发很多实际项目，如果我们用groupId对应公司和组织，那artifactId就只能是对于每个实际项目了，而再下去的模块，就没法描述了，而往往项目中的每个模块，是以单独的形式形成构件，以便其它项目重复聚合使用。
3. groupId的表述形式同Java包名的表述方式类似，通常与域名反向一一对应。
4. artifactId

定义实际项目中的一个Maven项目（实际项目中的一个模块）。

推荐命名的方式为：实际项目名称-模块名称。

比如org.springframework是实际项目名称，而我们现在用的是其中的核心模块，它的artifactId为spring-core。

1. Version

定义Maven项当前所处的版本。如上的描述，用的是4.2.7.RELEASE版本。需要注意的是，Maven中对版本号的定义，是由一套规范的，具体规范，请参考版本管理章节的介绍。

1. Packaging

定义Maven项目的打包方式。

打包方式通常与所生成的构件文件的扩展名对应，比如.jar, .ear, .war, .pom等。另外，打包方式是同工程构建的生命周期对应的，比如jar打包同war打包，使用的命令是不相同的。最后需要注意的是，我们可以不指定packaging，这时候maven会自动默认成jar。

1. Classifier

定义构件输出的附属构件。

附属构件同主构件是一一对应的，比如上面的spring-core-4.2.7.RELEASE.jar，是spring-core Maven spring-core项目的主构件，Maven spring-core项目除了可以生成上面的主构件外面，也可以生成spring-core-4.2.7.RELEASE-javadoc.java和spring-core-4.2.7.RELEASE-sources.jar这样的附属构件。这时候，javadoc和sources就是这两个附属构件的classifier。这样就为主构件的每个附属构件也定义了一个唯一的坐标。

最后需要特别注意的是，不能直接定义一个Maven项目的classifier，因为附属构件不是由maven项目构建的时候，直接默认生成的，而是由附加的其它插件生成的。

前面介绍的组成坐标的5个要素中，groupId、artifactId和version是必须的，packaging是可选的，默认是jar，而classifier是不能直接定义的。同时，Maven项目的构件文件名，同坐标也是有对应关系的，一般对于规则是artifactId-version[-classifier].packaging。

了解这些信息后，我们应该对坐标同构件一一对应的，已经怎么对应的还有个比较全面的了解了。

## 9.5 仓库

根据前面的样例实现，我们知道，坐标和依赖是构件在Maven中的一个标记，而构件的真正存在的形式是文件，Maven是通过仓库来统一管理这些文件的。接下来我们详细介绍一下Maven仓库，从底层更加理解、并高效的使用Maven。

### 9.5.1 Maven仓库的定义

在Maven中，所有的依赖、插件以及Maven项目构建完的输出，都是以构件的形式存在的，都叫构件。任何一个构件，都是由一组坐标信息唯一标识的。

在一个开发电脑中，有可能存在很多Maven项目，比如我们前面介绍的那么多样例代码，它们都是分布在不同的Maven项目中的。这些Maven项目，肯定都有用到compiler插件，处理这个插件外，还有很多功能的构件和自己特有的构件。比如MvnBookSSHDemo.Struts里面就用到了struts2构件，MvnBookSSH.Spring中就用到了Spring相关的构件，MvnBookSSM.SpringMVC中用到了Spring-web构件等等，而且这些直接依赖中，又会引入很多间接依赖，中间也肯定有交叉的构件依赖在引用。

如果同我们以前的开发模式一样，将各自用到的依赖对应的构件文件，都体现到自己的对应目录，比如lib目录下的话，我们会发现同样的文件，会在很多工程里面都重复存在，这样不仅仅造成了大量的磁盘空间浪费，还不方便统一管理，文件的复制等操作，也降低了项目构建的速度。

Maven使用的坐标机制，就帮我们解决了这问题。它是怎么解决的呢？

Maven在某个位置，统一存储了所有Maven项目用到的构件，这些构件都是共享的，当某个Maven项目要使用哪些构件的时候，就直接通过构件的坐标，引用共享的构件，不需要copy到每个Maven工程的独立物理目录中去。这个统一的位置，我们就叫它仓库。Maven的仓库，实际上就是Maven构件的公共仓库，在实际的Maven项目中，不在各自存储它们的依赖维基，它们只需要指明这些依赖的坐标，需要的时候（编译、测试、运行），由Maven自动根据坐标找到对应的构件后使用。

为了完全实现重用，Maven项目构建完毕后生成的构件，也可以安装和或部署到仓库中，供其他Maven项目，以坐标的新式指明使用。

### 9.5.2 仓库的管理方式

我们前面有介绍过，构件最终是以文件的新式存在的。既然是以文件的新式存在，那就少不了目录。Maven仓库，说白了，就是以固定规则的目录结构，把公共的构件保存在一个固定的存储位置，需要的时候，就可以按这规则，找到这些具体的构件文件。那这些规则是怎么样的呢，它同我们声明在pom.xml中，依赖的坐标是什么关系呢？

接下来，我们就以一个具体的例子，来分析一下构件在仓库中保存的目录结构规则，以及它同坐标的关系。

比如我们有个这样的构件：groupId=cn.com.mvnbook.demo、artifactId=MvnBookSSMDem-  
o.SpringMVC、version=0.0.1-SNAPSHOT、classifier=jdk17、packaging=jar，它对应的路径按如下步骤，逐个生成：

1. 生成groupId对应的第一部分路径：将groupId中的点分隔符，转换成路径分割符。该例子中，groupId=cn.com.mvnbook.demo转换成cn/com/mvnbook/demo/目录。
2. 生成artifactId对应的第二部分路径：在第一部分路径的基础上，加一层同artifactId名称对应的一样的目录。比如例子中的artifactId=MvnBookSSMDemo.SpringMVC，加上前面的groupId对应的目录，最后的结果是cn/com/mvnbook/demo/MvnBookSSMDemo.SpringMVC
3. 生成version对应的第三部分路径：同artifactId的转换类似，直接在前面的基础上，追加同version同名的目录。比如例子中version=0.0.1-SNAPSHOT，生成的目录就是  
   cn/com/mvnbook/demo/MvnBookSSMDemo.SpringMVC/0.0.1-SNAPSHOT。到这一步，构件文件所在的目录基本形成。
4. 前面三步，根据groupId、artifactId和version，生成了构件文件在仓库目录下的子目录，接下来就开始按规则生成构件的文件名。构件的文件名，首先是由artifactId和version按<artifactId>-<version>组成第一部分。比如我们例子的文件名第一部分就是MvnBookSSM-  
   Demo.SpringMVC-0.0.1-SNAPSHOT。
5. 如果构件有classifier的话，在第四步生成的文件名的基础上，用连接符-再连接classifier名字，我们的例子的文件名变成：MvnBookSSMDemo.SpringMVC-0.0.1-SNAPSHOT-  
   jdk17。
6. 最后再判断构件的packaging，在前面的文件名称后面添加packaging的值对应的后缀名。比如我们的例子，最后生成的目录和文件名是：cn/com/mvnbook/demo/Mvn-  
   BookSSMDemo.SpringMVC/0.0.1-SNAPSHOT/MvnBookSSMDemo.SpringMVC-0.0.1-SNAPSHOT-  
   jdk17.jar。

到现在为止，我们就介绍完了Maven是按什么规律，将坐标信息转换成构件目录和文件名，最终保存在仓库中的。明白这规律后，Maven根据坐标从仓库中找构件，就只是前面生成构件目录和文件名的一个逆过程了。

其实，我们也可按这样的规律，在自己的本地仓库或自己的私服仓库中，找到对应坐标的具体构件的文件。

### 9.5.3 仓库的种类

Maven的存放构件的仓库分两种：本地仓库和远程仓库。Maven根据坐标寻找构件的时候，先查看本地仓库，如果本地仓库存在坐标对应的构件，就直接使用；如果本地仓库不存在所需要的构件，或者需要查看是否有更新的构件版本，Maven就会去远程仓库查找，发现需要的构件后，下载到本地仓库再使用。如果本地仓库和远程仓库都没有找到需要的构件，Maven就报错。

远程仓库又可以分三种：一种是中央仓库；一种是私服；还有一种就是其他公共仓库。

中央仓库是Maven核心自带的远程仓库，它包含了绝大部分开源的构件，在默认配置下，当本地仓库没有Maven需要的构件的时候，都会尝试从中央仓库下载。

私服是另外一种特殊的远程仓库，为了节省带宽，节约下载构件的时间，在局域网内架设一个私有的仓库服务器，用来代理所有的外部远程仓库，局域网里面的项目还能部署到私服上供其他项目使用。

除了中央仓库和私服外，还有很多其它公开的远程仓库，常见的有java.net Maven库(http://download.java.net/maven/2/2)和JBoss Maven库(http://repository.jboss.com/maven2/)等。

为了更具体的对仓库的理解，更深刻的理解Maven的运行工程，更方便熟练的使用Maven开发应用，我们详细研究各种仓库。

1. 本地仓库

Maven在根据坐标查找依赖的构件的时候，先是在本地仓库中查找。默认情况下，不管是Windows操作系统，还是Linux操作系统，每个用户在自己的用户目录下，都有一个路径名为.m2/repository/的目录，这个目录就是Maven的本地仓库目录。比如我的用户名是Noble，我电脑上的默认本地仓库的目录就是：C:\Users\Noble\.m2\repository\。

一般为了方便文件的管理，我们会希望自定义本地仓库的目录。这也是可以的，可以编辑~/.m2/settings.xml文件，设置里面的localRepository元素的值，就可以改变Maven本地仓库的默认位置。如：

<settings>

...

<localRepository>  
 C:/java/servers/apache-archiva-2.2.1/repositories/internal  
 </localRepository>

...

</settings>

这样，本地仓库的目录就是C:/java/servers/apache-archiva-2.2.1/repositories/internal了。

另外需要注意一下的是，默认情况下，~/.m2/settings.xml文件是不存在的，我们需要从Maven的安装目录中，复制$M2\_HOME/conf/settings.xml文件到~/.m2/目录下，再进行编辑。当然如果嫌麻烦的话，可以直接修改$M2\_HOME/conf/settings.xml文件，效果也是一样的，但是不建议修改，因为Maven目录下的settings.xml是全局的，也就是每个用户都共享，而~/.m2/settings.xml只是当前用户起作用，修改后不会影响其它用户。

一个构件只有存在本地仓库后，才能被Maven项目使用到，那构件如果保存到本地仓库呢？最常见的有两种方式，一个是以依赖的形成，从远程仓库下载到本地仓库；一种是将本地项目编译打包后，安装到本地仓库；

Maven在构建项目的时候，如果在本地仓库中没有找到需要的构件，自动会从远程仓库下载，安装到本地仓库的，至于本地项目后构建成项目安装到本地仓库，也不复杂：在项目目录下，使用mvn install命令，就可以构建当前项目成构件，并且安装到本地仓库中去。

1. 远程仓库

安装好Maven后，如果不执行任何Maven命令的话，本地仓库目录是不存在的。当我们输入第一条Maven命令后，Maven才会创建本地仓库，然后根据配置和需要，从远程仓库下载对应的构件，到本地仓库，以备我们的Maven项目需要的时候使用。

对我们来说，本地仓库只会有一个，而远程仓库可以有很多，当然，具体可以操作哪些远程仓库，这也需要我们做对应的配置。

1. 中央仓库

由于最原始的本地仓库是空的，Maven必须知道至少一个远程仓库，才能执行Maven的命令，并且下载需要的构件，这个远程仓库是默认的，也就是不需要我们专门配置的，我们把它叫做中央仓库。也就是说，中央仓库就是一个默认的远程仓库。

我们可以打开$M2\_HONE/lib/maven-model-builder-3.3.9.jar，里面有org/apache/maven/  
model/pom-4.0.0.xml文件（注意jar文件的3.3.9是当前Maven的版本号，要根据自己的版本号找对应的jar文件），里面的有如下内容：

<project>

<modelVersion>4.0.0</modelVersion>

<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>

<pluginRepositories>

<pluginRepository>

<id>central</id>

<name>Central Repository</name>

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

<layout>default</layout>

<snapshots>

<enabled>false</enabled>

</snapshots>

<releases>

<updatePolicy>never</updatePolicy>

</releases>

</pluginRepository>

</pluginRepositories>

...

</project>

中间repository配置的是依赖的默认中央仓库，pluginRepository配置的是插件的默认中央仓库，很凑巧，它们都是 https://repo.maven.apache.org/maven2。所有的Maven项目，都会继承这个pom.xml，所以我们把这个pom叫超级pom，同Java中的Object类一样，也是所有的类都自动继承Object类，而Object类也叫超级类。

中央仓库包含了这世界上绝大多数流行的开源Java构件，已经对应的源代码、作者信息、scm、信息许可证等，每天都接受全世界Java程序员N N次的访问，可以想象出没有它，Java程序员的世界将变得怎么样个黑暗。由于中央仓库中包含了差不多所有流行的构件，所以一个简单的Maven工程所需要的依赖构件，都可以直接从中央仓库中找到并且下载下来，所以我们安装好Maven后，基本上不用做太多的额外配置，就可以直接开发Maven项目。

1. 私服

前面我们有简单介绍了，私服是一个特殊的远程仓库，它是架设在局域网内的仓库服务器，它是一个代理外网的一个远程仓库，供局域网内部的Maven用户使用。

当局域网内部的Maven用户需要构件的时候，先是从自己的本地仓库中查找，没有找到，就在私服上面查找，还没找到，就从外部的远程仓库查找并下载。这时候需要注意一下，没有私服的时候，Maven是直接把从外部远程仓库下载下来的构件，保存到本地仓库中，供当前的Maven工程和其它的本地Maven工程使用，现在有私服了，从外部远程仓库下载下来的构件，会先将构件保存一份在私服，再在Maven用户的本地仓库中保存一份。保存在Maven用户本地仓库中的构件，可以实现一个用户多个Maven工程的共享使用，保存在私服中的构件，可以实现多个Maven用户，在局域网内共享使用。这样做，有如下几个方面的优势：

1. 节省外部带宽

建立私服可以减少Maven开发团队的开支，大量的对外部仓库的重复请求，会浪费太多的网络流量，利用私服代理后，每个构件只要消耗一次必须的流量，局域网中任何一个Maven用户要使用某个已经使用过的构件的时候，就不需要再请求远程仓库，只需要从私服中直接获取就行了。

1. 提高Maven的构件效率

不停的请求外面仓库，下载构件，不仅仅会浪费流量，还很耗时，特别是网络比较拥、上网速度比较慢的时候，搭建私服就不用频繁的请求外部远程仓库了，而且局域网的速度一般要比外网速度快多了的。

1. 方便部署第三方构件

如果某个构件没法从外部远程仓库获取到的话，没有私服，相对会比较麻烦的。如果有了私服，我们可以将它们部署到私服里面，这样局域网内的Maven用户就可以找到对应的构件了。比如Oracle驱动程序，因为版权没有放开，在外部远程仓库中是没有部署的，我们就可以在自己局域网内部的私服里面进行部署，这样基于网内部的Maven用户就可以使用Oracle驱动程序构件了，这样也不会影响Oracle的版权。

1. 提高Maven的稳定性，更方便控制管理

Maven构件如果高度依赖远程仓库的话，一旦外网不稳定，比如中断，就会直接影响我们的Maven项目的构件，甚至不能构件。使用了私服后，因为私服中已经有保存了大量的构件，即使外网断了，只要当前的构建任务中，没有使用到一个以前构建没有用过的构件，都可以照常工作。

另外，一些私服软件还提供了很多额外的管理功能：比如权限管理、RELEASE/SNAPSHOT区分等，管理员可以对私服进行更多的控制管理。

1. 降低中央仓库的负荷压力

通过对全面内容的了解，搭建私服，能降低中央仓库的压力，让中央仓库更好的服务与哪些迫切且真正需要的用户，应该不难理解的。我们少请求一次中央仓库，就给别人都留出了一次快速请求的机会。

通过对前面的了解，我们可以体会得出，在一个团队开发中，搭建Maven私服是一个重要的环境，前面有章节专门以archiva为例子，介绍了怎么样搭建Archiva私服。大家现在可以回顾以前的操作步骤，更加理解私服的意义。

### 9.5.4 配置远程仓库

虽然我们可以从中央仓库中找到绝大部分流行的构件，但是毕竟不是所有。对那些在中央仓库中没有的构件，而我们又要使用怎么办呢？我们可以在pom.xml中添加另外一个远程仓库，从而从这个新的远程仓库中，下载中央仓库中没有的构件。比如我们要将jboss Maven远程仓库，添加到我们的Maven工程的话，需要在Maven工程的pom.xml中，添加如下配置：

<project>

...

<repositories>

<repository>

<id>jboss</id>

<name>JBoss Maven Repository</name>

<url>http://repository.jboss.com/maven2/</url>

<releases>

<enabled>true</enabled>

</releases>

<snapshots>

<enabled>false</enabled>

</snapshots>

<layout>default</layout>

</repository>

</repositories>

...

</project>

在repositories元素下，可以使用repository子元素声明一个或多个远程仓库。该例子中，配置了一个id为jboss，名称为JBoss Maven Repository的仓库。在pom中，我们可以配置多个仓库，每个仓库的id需要唯一，而且我们需要注意一下的是，Maven的超级pom中，有默认配置了一个中央仓库的，它的id为central。请不要在配置一个一样的id，否则新的配置会覆盖原理的中央仓库的配置。在repository中，有个url元素，该元素是指定当前配置的远程仓库的地址，一般来说，都是基于http协议的，我们可以在浏览器中打开仓库地址，浏览仓库中的构件。

另外，配置中的releases和snapshots元素，也是比较重要的元素，它们用来控制Maven对发布版本的构件和快照版本构件的下载。当它们的子元素enabled的值配置成true或false的时候，表示开启或关闭对应版本的构件下载。在实际项目中，我们一般开启releases版本的构件下载，屏蔽snapshots版本的构件下载，因为snapshots版本的构件不稳定，我们的Maven工程引用了snapshots版本的构件，会直接影响我们自己Maven工程构建的稳定性。

至于上面例子中的layout元素，值为default，表示仓库布局是Maven2和Maven3的默认布局，而不是Maven1的布局。

当然，我们也可以在releases和snapshots元素中，添加updatePolicy和checksumPolicy两个子元素，进一步指定仓库的控制使用。

比如：

<releases>

<enabled>true</enabled>

<updatePolicy>daily</updatePolicy>

<checksumPolicy>ignore</checksumPolicy>

</release>

updatePolicy配置Maven从远程仓库检测更新的频率，默认值为daily，表示每天检测异常。还可以配置其它的值：never--从不检测更新；always--每次构建都检测更新；interfal：X--每隔X分钟检测一次更新。

checksumPolicy配置检测校验和文件的策略。当构件被部署到Maven仓库的时候，自动会部署对应的校验和文件。在下载构件的时候，Maven会严重校验和文件，如果失败了怎么办？当checksumPolicy的值为warn时，Maven会执行构件是输出警告信息；如果是fail，Maven会直接中止构件，提示构件失败；如果是ignore，Maven会忽略校验的错误，继续构建Maven项目。checksumPolicy的默认值是warn。

配置好了远程仓库，那我们怎么样将我们自己的Maven项目构建成的构件，发布到远程仓库中去呢？

比如前面我们搭建了私服，我们怎么样将Maven项目的每个模块，以构件的形式部署在私服上，供其它Maven项目去集成使用。

当然也是可以的，只是需要我们在pom.xml中，使用distributionManagement配置部署信息就可以了，样例配置如下：

<distributionManagement>

...

<repository>

<id>archivaServer</id>

<name>Archiva Releases</name>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<name>Archiva Snapshots</name>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

...

</distributionManagement>

distributionManagement中包含repository和snapshotRepository两个子元素，repository表示帆布版本的构件仓库，snapshotRepository表示快照版本的构件仓库。Id和name，分别是仓库的唯一标记和名称。

在pom.xml中完成了上面类似的配置后，使用maven命令mvn deploy，Maven就会自动将构建输出的构件，部署到对应的仓库中。

不管是我们现在的在远程仓库中部署构件，还是从远程仓库中下载依赖构件，实质上都是对服务器进行访问。有些服务器访问是需要权限认证的，只有认证通过后的用户，才能发请求访问服务器，特别是添加、修改和删除服务器上的文件。

那我们怎么样配置，让Maven能顺利的自动访问那些需要权限认证后才能访问的远程仓库呢？

同配置仓库信息和远程私服部署信息不同，它们都是在pom.xml中进行配置的，需要在settings.xml中进行配置。因为pom是被提交到代码仓库中，供所有成员访问的，而settings.xml一般只放在本地机器，因此在settings.xml中配置认证信息更安全。比如如下配置，就是我们配置的访问前面搭建的Archiva私服服务器的安全认证信息：

<settings>

...

<servers>

...

<server>

<id>archivaServer</id>

<username>admin</username>

<password>admin123</password>

</server>

...

</servers>

...

</settings>

其中username和password是服务器中安全认证的用户名和密码信息。Id为认证服务器的唯一标记，这个标记，需要同pom.xml中distributionManagement里面配置的仓库的id要对应起来，表示访问某个仓库的地址的话，需要先根据id找到server的认证信息认证，才能有权限访问。

### 9.5.5 快照版本

在Maven中，任何一个项目和构件，都必须有自己的版本。版本的值可能是1.0.0、1.0-  
alpha-4、1.3-SNAPSHOT等等，其中1.0.0、1.0-alpha-4是稳定的发布版本，而1.3-SNAPSHOT为不稳定的快照版本。

Maven为什么要添加一个快照版本的控制呢？

假设张三在开发用户管理模块的1.1版本，该版本还没有正式发布，同用户管理模块以前开发的，还有权限管理模块，它是由李四在单独开发。其中权限管理模块的功能，是依赖用户管理模块的。在开发过程中，张三要经常要将最新的用户管理模块构建输出，交给李四，让他对权限管理模块进行开发集成和调试。这种问题，如果由我们自己手动控制的话，相对比较麻烦，Maven基于快照机制，就能自动帮我们解决这个问题。

基于Maven的快照机制，张三只需要将用户管理模块的版本设置成1.1-SNAPSHOT，然后发不到私服中，在发布过程中，Maven会自动为构件打上时间戳，比如1.1-20161211.111111-11，表示2016年12月11日11点11分11秒的第11次的快照。游离这个时间戳，Maven就能随时找到仓库中用户管理构件1.1-SNAPSHOT版本的最新文件。这时，李四配置对用户管理模块的1.1-SNAPSHOT版本的依赖，当他构件权限管理模块的时候，Maven会自动从仓库中检测用户管理1.1-SNAPSHOT的最新构件，发现最新构件后就自动下载。Maven默认情况下，每天检测一次（具体实际情况，由参考配置的updatePolicy控制）,当然，我们也可以使用 mvn -U强制让Maven检测更新。如：mvn clean install -U。

基于这样的机制，张三在构建成功后，将构件发布到仓库，李四可以完全考虑用户管理模块的构件，并且他还能确保随时得到用户管理模块的最新可用的快照构件，这些所有的一切，都有Maven自动完成。

快照版本只应该在开发团队内部的项目或模块之间依赖使用，因为这时，团队成员对这些快照版本的依赖具有完全的理解和控制权利。项目部应该依赖任何团队外部的快照版本依赖，由于快照版本的不稳定性，这样的依赖会造成潜在的危险。也就是说，即使项目构建这次成功了，由于外部的快照版本依赖会随时间改变而再次更新的并发生变法，下次构建的时候，很有可能会失败。

### 9.5.6 从仓库中解析依赖的机制

前面我们介绍了Maven的依赖机制，那些构件是放在仓库中的，那Maven是根据什么规则从仓库中解析这些依赖构件的呢？

Maven在寻找项目需要的依赖的顺序是，先在本地仓库中查找，如果没有找到，再找远程仓库，找到后下载；如果依赖的版本为快照版本，Maven除了找到对应的构件，还会自动找最新的快照。这个找依赖的过程大概如下：

1. 当依赖的范围是system的时候，Maven直接从本地文件系统中解析构件。
2. 根据依赖坐标，计算仓库路径，尝试直接从本地仓库寻找构件，如果发现对应的构件，就解析成功。
3. 如果在本地仓库不存在相应的构件，而且一来的版本是发布版本，就遍历所有的远程仓库，发现后，下载并解析使用。
4. 如果依赖的版本是RELEASE或LATEST，就基于更新策略读取所有远程仓库的元数据文件:groupId/artifactId/maven-metadata.xml将其与本地仓库的对应源数据文件合并，计算出RELEASE或LATEST的真实值，然后各级该值检查本地仓库，或者从远程仓库下载。
5. 如果依赖的版本是SNAPSHOT，就基于更新策略读取所有远程仓库的元数据文件，将它与本地仓库对应的元数据合并，得到最新快照版本的值，然后根据该值检查本地仓库，或从远程仓库下载。
6. 如果最后解析得到的构件版本包含有时间戳，先将该文件下载下来，再将文件名中时间戳信息删除，剩下SNAPSHOT并使用（以非时间戳的形式使用）。

### 9.5.7 镜像

如果仓库A能提供仓库B存储的所有构件服务，我们就将A叫做B的镜像。比如http://  
maven.net.cn/content/groups/public就是中央仓库http://repo1.maven.org/maven2/在中国的镜像。由于地理位置的因素，该镜像往往能够提供比中央仓库更快的服务。所以，为了提高我们Maven的构件效率，我们可以通过配置配置文件，用镜像代替中央参考。如修改settings.xml如下：

<settings>

...

<mirrors>

<mirror>

<id>maven.net.cn</id>

<name>中央仓库在中国的镜像</name>

<url>http://maven.net.cn/content/groups/public/</url>

<mirrorOf>central</mirrorOf>

</mirror>

...

</mirrors>

...

</settings>

上面代码中，mirrorOf的值为central，表示该配置为id为central仓库的镜像，也就是中央仓库的镜像。任何对中央仓库的请求，都会转向到这个镜像，我们也可以用同样的方式配置其它仓库的镜像。另外三个元素：id、name和url同我们以前配置仓库信息一样，表示镜像的唯一标记、名称和地址。同样的，如果镜像服务器需要认证的话，也可以根据这个id配置一个对应的仓库认证。

其实在实际工作中，关于镜像有一个最常见的用法，那就是结合私服使用。由于私服是用来代所有的外部公共仓库的，包括中央仓库，所以对于团队内部的Maven用户来说，使用一个私服地址，就等于使用了所有的外部仓库，这样就可以将对外部远程仓库的访问配置，都集成到私服上来，从而简化Maven本身的配置。为达到这样的目标，我们可以配置一个如下内容的镜像：

<settings>

...

<mirrors>

<mirror>

<id>internal-repository</id>

<name>Internal Repository Manager</name>

<url>http://192.168.1.207:8080/repository/internal</url>

<mirrorOf>\*</mirrorOf>

</mirror>

...

</mirrors>

...

</settings>

上面配置信息中，mirrorOf的值为\*，表示是所有Maven仓库的镜像，任何对远程仓库的请求，都会转向到207这台电脑的私服上去。如果私服需要认证，统一配置一个id为internal-repository的server就可以了。

当然，关于mirrorOf还有一些特别的配置方式，一遍适合更多的需求：

1. <mirrorOf>\*</mirrorOf>：匹配所有的远程仓库
2. <mirrorOf>external:\*</mirrorOf>：匹配所有的远程仓库，使用localhost的除外、使用file://协议的除外。也就是说，匹配所有非本地的远程仓库。
3. <mirrorOf>r1,r2</mirrorOf>：匹配指定的几个远程仓库，每个仓库之间用逗号隔开。
4. <mirrorOf>\*,!r1,r2</mirrorOf>：匹配除了指定仓库外的所有仓库，!后面的仓库，是被排除外的。

### 9.5.8 仓库搜索服务

在实际开发过程中，我们可能只知道需要使用的构件项目名称，但是在Maven依赖配置中，一定要指定详细的坐标信息，这时候，我们就可以使用仓库的搜索服务，根据构件的关键字来查找Maven的详细坐标。接下来介绍几个常用的Maven仓库搜索服务，希望能帮助我们在实际开发过程中，快速找到自己想要的构件坐标。

MVNRepository

地址：<http://mvnrepository.com>

MVNRepository的界面相对比较简洁清新，提供基于关键字搜索、依赖声明代码片段、构件下载、遗骸与被依赖关系信息、构件所包含信息等功能。如下是MVNRepository的界面图9-8：

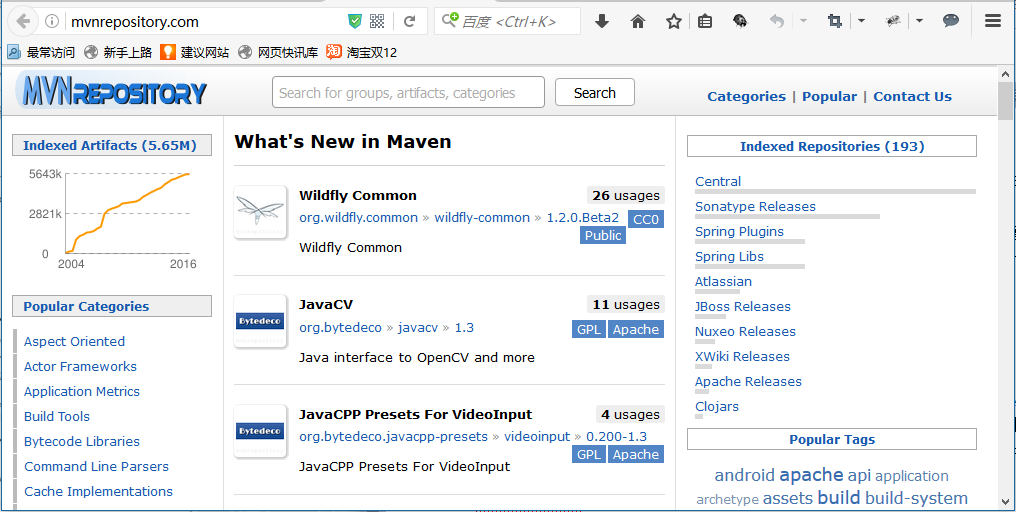


图9-8 中央仓库首页

## 9.6 依赖

### 9.6.1 依赖是什么

前面我们有用坐标一一对应的描述了构件，并且保存在仓库中了。那用坐标描述好后，把它们放在仓库中的作用是什么呢？有什么用？当其它项目需要在这些构件的基础上，开发实现其它功能的时候，我们就可以没必要自己再重新实现一遍了，直接指定坐标，告诉Maven，将坐标对应的构件，从仓库中找出来，集成到我们的新项目中就可以了。这时候，引入的构件，就是我们新项目的依赖。

依赖一般分两个层次理解，第一个就是我们在Maven项目的pom.xml中配置所需要构件的坐标，也就是配置依赖。还一个就是Maven在构建项目的时候，根据坐标从仓库中找到坐标所对应的构件文件，并且把它们引入到Maven项目中来，也就是Maven引用。

第二个层次，由Maven构件的时候自己搞定，前面我们也介绍了Maven基于坐标找要执行的插件的思路，实际上插件，本身就是一个特殊的构件，找插件的思路，也就是依赖查找的思路。我们需要把握的，更多的是第一层次，怎么样配置依赖，已经用配置文件很好的指定所需要的依赖。以及指定依赖内部的关系和优化等等。

### 9.6.2 依赖的配置

掌握依赖，从配置开始，接下来我们介绍一下依赖的配置。依赖是配置在pom.xml中的，如下是关于依赖配置的大概内容：

<project>

...

<dependencies>

<dependency>

<groupId>...</groupId>

<artifactId>...</artifacted>

<version>...</version>

<type>...</type>

<scope>...</scope>

<optional>...</optional>

<exclusions>

<exclusion>...</exclusion>

</exclusions>

</dependency>

...

</dependencies>

...

</project>

通过前面依赖配置样例，我们会发现，依赖配置中处理构件的必须的坐标信息，groupId、artifactId和version之外，还有其它的元素，接下来我们简单介绍一下这些元素的作用，详细的意义，在后面章节中逐个介绍。

1. groupId、artifactId和version：依赖的基本坐标，对于任何依赖，基本坐标是最基本、最重要的，因为Maven是根据坐标找依赖的。
2. Type：依赖的类型，同项目中的packaging对应。大部分情况不需要声明，默认是jar。
3. Scope：依赖的范围，详细情况后面介绍
4. Optional：标记依赖是否可选，详细情况后面介绍。
5. Exclusion：排除传递性依赖，详细情况后面介绍。

大部分情况，我们只需要指定坐标相关的基本信息就行了，但是有些情况需要使用其它元素指定其它信息，这里就不详细介绍了，后面接着会逐个详细介绍。

### 9.6.3 依赖的范围

前面有提到了，在配置依赖的时候，scope来指定依赖的范围。那依赖的范围到底是什么意思，它到底有哪些选项，怎么使用呢？接下来我们将逐个讨论。

刚开始学习Java的时候，我们就知道Java中有个环境变量叫classpath，Java编译器编译编译源代码的时候，需要基于classpath查找第三方的类文件，JVM运行代码的时候，需要基于classpath查找需要的类文件，才能加载到内存执行。

Maven在编译项目主代码的时候，使用的是一套classpath，主代码编译时需要的依赖，就需要添加到这个classpath中去；Maven在编译和执行测试代码的时候，会使用一套classpath，这个动作需要的依赖，需要添加到这个classpath中去；Maven项目具体运行的时候，又有一个独立的classpath，同样运行时需要的依赖，肯定也要加到这个classpath中。这些classpath，就是我们依赖的范围。

依赖的范围，就是用来控制这三种classpath的关系（编译classpath、测试classpath和运行classpath），接下来我们分别介绍依赖的范围的名称和意义：

1. Compile：编译依赖范围。如果我们在配置的时候没有指定的话，就默认使用这个范围。使用该依赖范围的依赖，对编译、测试、运行三种classpath都有效。
2. Test：测试依赖范围。使用该依赖范围的依赖，只对测试classpath有效，在编译主代码或运行项目的时候，这种依赖是无效的。
3. Provided：已提供依赖范围。使用次依赖范围的依赖，只在编译和测试classpath的时候有效，运行项目的时候是无效的。比如web应用中的servlet-api，编译和测试的时候就需要该依赖，运行的时候，因为容器中自带了servlet-api，就没必要使用了。如果使用了，反而有可能出现版本不一致的冲突。
4. Runtime：运行时依赖范围。使用该范围的依赖，只对测试和运行的classpath有效，但在编译主代码时是无效的。比如JDBC驱动实现类，就需要在运行测试和运行主代码时候使用，编译的时候，只需要JDBC接口就行。
5. System：系统依赖范围。该范围同classpath的关系，同provided一样。但是，使用system访问的依赖时，必须通过systemPath元素，显式的指定依赖文件的路径。因为该依赖不是通过Maven仓库解析的，而且同本节系统绑定，可能会影响构建的可移植性，所有建议谨慎使用。

如下是一个使用system范围的一个案例：

<dependency>

<groupId>xxx</groupId>

<artifactId>xxx</artifactId>

<version>xx</version>

<scope>system</scope>

<systemPath>e:/xxxx/xxx/xx.jar</systemPath>

</dependency>

1. Import:导入依赖范围。该依赖范围不会对三种classpath产生实际的影响。它的作用是将其它模块定义好的dependencyManagement导入合并到当前Maven项目pom的dependencyManagement中。比如有个SpringPOM Maven工程，它的pom中的dependencyManagement配置如下：

<project>

...

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

...

<dependencyManagement>

<dependencies>

<!-- spring -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

</dependencies>

</dependencyManagement>

...

</project>

接下来我们创建一个新的Maven工程Second，想将First工程中，pom中定义的dependencyManagement原样合并过来，除了可以copy、继承之外，我们还可以写如下代码，将它们import进去：

<dependencyManagement>

<dependencies>

<dependency>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

### 9.6.4 传递性依赖

在我们使用Maven之前，如果要基于Spring框架开发项目，除了要加入Spring框架的jar包外，还需要将Spring框架所用到的第三方jar包加入。否则可以编译通过，但是运行的时候就会出现classNotFound异常。为了解决这样的问题，一般有两种方式：一种是下载spring的dependencies.zip包，将里面的所有jar包都导入到工程；另外一种是根据运行时的报错信息，确定哪些类没有，再将包含这些类的jar包下载下来导入。

第一种方式虽然可以一次性解决所有需要jar包的导入问题，但是我们查看工程的jar包会发现，有好多多余的jar包，这些多余的jar包不仅仅加大了我们项目的体积，还有可能同其它框架所导入的jar包有版本冲突。

第二种方式虽然不会有多余的jar包存在，但是要根据每次启动的错误，一个个的找到jar包，在导入，想象如果有10个jar包，就要启动10次，查看10次错误分别导入，有多麻烦。

Maven的传递依赖机制就帮我们解决这样的问题。

当我们项目基于Spring框架实现的时候，只需要将spring的依赖配置到pom的依赖元素就行，至于spring框架所依赖的第三方jar包，我们不用处理，Maven自己通过检测spring框架的依赖信息，将它们导入到我们的项目中来。而且只会导入spring框架所需要的，不会导入多余的依赖。

也就是说，Maven会解析我们项目中的每个直接依赖的pom，将那些必要的间接依赖，以传递依赖的形式，引入到我们的项目中。

当然，传递依赖在将间接依赖引入我们项目的过程中，也有它自己的规则和范围。那这个规则和范围是什么样子的呢？

这个规则和范围，是同去我们前面介绍的依赖范围紧密关联的。

现在有三个项目 A、B和C项目，假设A依赖B，B依赖C，我们把A对B的依赖叫第一直接依赖，B对C是第二直接依赖，而A对C是传递依赖（通过B传递的）。中间A到B第一直接依赖的范围和B到C第二直接依赖的范围，就共同决定了A到C的传递依赖范围。它们的影响效果，就如下表格。坐标第一列为第一直接依赖的范围，第一行表示第二直接依赖的范围，中间的交叉点为共同影响后的传递依赖范围。

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

通过前面的表格，我们可以得出如下规律：

当第二直接依赖为compile的时候，传递依赖同第一直接依赖一致

1. 当第二直接依赖为test的时候，没有传递依赖
2. 当第二直接依赖为provided的时候，值将第一直接依赖中的provided以provided的形式传递
3. 当第二直接依赖为runtime的时候，传递依赖的范围，基本上同第一直接依赖的范围一样，compile除外，compile的传递依赖范围为runtime。

### 9.6.5 依赖的调解

在我们使用Maven自动提供的传递依赖后，可以解决我们对应依赖管理，特别是间接依赖管理的繁琐问题，但是，当多个直接依赖，都带来了同一个间接依赖，而且是不同版本的间接依赖的时候，就会引起重复依赖，甚至包冲突问题。

象这样的问题，Maven在传递依赖的时候是按什么规则来的呢？在这样的规则基础上，我们作为开发人员，在实践中有哪些好的方式利用它们呢？接下来我们分别进行介绍。

先讨论Maven的传递依赖中的规则：

1. 依赖调解原则

Maven依赖调整原则有两个：一个是路径最近优先原则；一个是第一声明优先原则；而且是路径优先原则搞不定的时候，再使用声明优先原则。

比如我们有个项目A，它有两个依赖：A->B->C->T(1.0)，A->D->T(2.0)。我们会发现，A最终对T1.0和T2.0都有间接依赖，这时候Maven会自动判断它的路径，发现T2.0的路径长度为2，T1.0的路径长度为3，以最短路径为原则，将T2.0引入当前项目A。

如果我们有个项目A，它有两个依赖：A->B->T(1.0)，A->C->T(2.0)。这时候两条路径都是一样的长度2，那Maven是到底把那个引入到项目A呢？这时候Maven会判断哪个依赖，在pom.xml中先声明，选择引入先声明的依赖。

1. 可选依赖

在实际项目中，存在一些比较特殊的依赖。比如数据访问层模块，对数据库驱动的依赖就比较特殊了。DAO层要范围数据库的时候，需要加入数据库驱动依赖，而且不同数据库，驱动依赖是不一样的。如果我们设计DAO层的时候，考虑并且是按跨数据库标准实现的，这就来了一个新问题，我们是在pom.xml中配置MySQL驱动依赖呢，还是配置Oracle驱动依赖？或者两个都配置？

其实仔细想想，前面三种选项都不合适，单独配置MySQL或Oracle，这样就不能跨数据库了，两个数据库都配置，驱动之间就会有冲突，或有多余的依赖。

这时候，我们就直接把这两个数据库驱动的依赖都设置成可选依赖，代码如下：

<dependencies>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.34</version>

<optional>true</optional>

</dependency>

<dependency>

<groupId>oracle</groupId>

<artifactId>ojdbc14</artifactId>

<version>10.2.0.4</version>

<optional>true</optional>

</dependency>

</dependencies>

在应用项目中，我们再具体指定使用哪个依赖，如：

<dependencies>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.34</version>

</dependency>

</dependencies>

需要说明的是，在实际项目中，建议不要使用可选依赖。虽然可选依赖满足了我们对一个模块的特征多样性，同时还提供了机会让我们选择其中的一种，但是在实际配置过程中，好像不仅没有减少我们的配置代码，还加多了重复copy的机会。同时从我们提倡的面向对象分析和设计的思路来说，也是建议遵循单一职责原则，也就是一个类只有一个功能，不要揉合太多的功能，不方便理解、开发和维护。所以实际项目中，我们一般是对不同数据库的驱动单独创建一个Maven工程，各自什么自己对驱动的依赖，其它项目需要基于哪个数据库进行操作的话，引用对应的Maven的工程以来就行，用传递依赖，引入需要的数据库驱动依赖。

### 9.6.6 排除依赖

Maven的传递依赖能自动的帮我们将间接依赖引入到项目中来，这样极大的简化了我们项目中的依赖管理，但是，有时候这种自动化也会带来麻烦。比如Maven可能帮我们自动引入快照版的依赖，而快照版的依赖是不稳定的，这时候我们就需要避免引入快照版。这时候就需要用一种方式告知Maven，排除快照版的依赖引入，这种做法就是排除依赖。那怎么实现排除依赖呢？

其实实现排除依赖还是比较简单的，在直接依赖的配置里面，添加exclusions->exclusion元素，指定要排除依赖的groupId和artifactId就行，如下面代码：

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>${project.build.hibernate.version}</version>

<exclusions>

<exclusion>

<groupId>xxx</groupId>

<artifactId>xxx</artifactId>

</exclusion>

</exclusions>

</dependency>

上面的代码的含义就是，在引入hibernate-core直接依赖的时候，不要引入exclusion中指定的groupId为xxx，artifactId为xxx的构件。需要注意的是，这里没有version，排除依赖是排除指定groupId和artifactId的所有版本的依赖。

### 9.6.7 归类依赖

在我们引用依赖的时候，很多情况需要引入一个Maven项目的多个模块，这些模块都应该是相同的版本。比如我们在spring框架下开发应用，就需要同时引用org.springframework的spring-core、spring-context、spring-context-support等模块。可以想象的出，这些模块肯定是统一个版本，如果我们在每个依赖里面都分别用groupId、artifactId和version具体指明的话，下次升级，需要将2.5版本升级成3.0版本，这样就需要将org.springframework的每个模块的版本都统一更改，这样做很容易出现不一致情况，很容易出错。

为了避免出现这种情况，我们可以在pom.xml中定义一个属性名称，描述版本的值，接下来在每个version中，用特殊的语法引用这个属性名称，实际引入的时候，有maven将属性改成对应的值。这样就可以统一版本，也方便修改。具体样例代码如下：

<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/xsd/maven-4.0.0.xsd">

...

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<!-- 3.2.16.RELEASE,3.1.4.RELEASE -->

<project.build.spring.version>4.2.7.RELEASE</project.build.spring.version>

</properties>

<dependencies>

<!-- spring -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

...

</dependencies>

...

</project>

### 9.6.8 优化依赖

在开发软件过程中，我们需要通过重构等方式，不断的优化自己的代码，使其变得更简洁、灵活、高效。同样的，我们也应该对Maven项目的依赖了然于胸，并对其进行诱发。比如去除多余的，显式的配置必要的依赖等。

通过对前面章节的了解，我们理解Maven定位依赖的方式、传递依赖的规则、还有怎么样排除依赖等。但是要实现这些动作，还必须对我们项目中的依赖有全面的了解，这样才能更有效的达到我们的目的。

接下来介绍一下查看依赖的相关命令。

Mvn dependency：list 列出所有的依赖列表

Mvn dependency：tree 以树形结构方式，列出依赖和层次关系

Mvn dependency：analyze 分析主代码、测试代码编译的依赖

## 9.7 继承和聚合

我们在设计java类的时候，如果发现很多类都有公共的行为和特征的话，很自然的会将这些公共特征和行为，提炼到一个类中，我们把这个类叫父类，其它有这些特征和行为的类里面，就不要重复定义了，二只要继承个父类就行，这样至少解决了代码重复编写的问题，从而简化了子类的代码。

同样的，在Maven项目里面，我们经常需要在pom.xml中配置很多信息，比如坐标信息、依赖、插件等等，随着Maven项目的模块化，我们发现很多内容都是重复的，为了保持一致，尽量的用复制粘贴的方式编写，这样虽然保证了一致性，但是这只是一次性的，如果要修改的话，这就需要在很多的pom.xml中重复的查找修改了，非常麻烦。

为了解决这样的问题，Maven借鉴了面向对象的思想，支持继承。

也就是说，我们可以将多个项目要用的配置，单独用一个pom类型的工程定义好，其它有重复使用这些配置的Maven项目，就可以在继承公共pom项目的基础上，再扩展自己个性化的信息。

比如我们可以定义一个Maven项目，它的pom.xml配置如下：

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<name>SpringPOM</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<!-- 3.2.16.RELEASE,3.1.4.RELEASE -->

<project.build.spring.version>4.2.7.RELEASE</project.build.spring.version>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.7</version>

<scope>test</scope>

</dependency>

<!-- spring -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

...

</dependencies>

<distributionManagement>

<repository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

</distributionManagement>

</project>

在这个pom中定义自己的坐标和jUint、springframework的依赖。注意packaging的值为pom，因为我们只是在这里定义共用的依赖等信息，不写代码，以备其它工程继承的，必须packaging为pom类型。

在另外一个使用springframework框架的Maven工程的pom.xml中，可以写成如下样式：

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<!-- 继承SpringPOM中的pom -->

<parent>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>SpringMVCPOM</artifactId>

<packaging>pom</packaging>

<name>SpringMVCPOM</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<!-- jsp servlet -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>jsp-api</artifactId>

<version>2.1</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<!-- json组件 -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.5.4</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-core</artifactId>

<version>2.5.4</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-annotations</artifactId>

<version>2.5.0</version>

</dependency>

</dependencies>

<distributionManagement>

<repository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/internal</url>

</repository>

<snapshotRepository>

<id>archivaServer</id>

<url>http://localhost:8080/repository/snapshots</url>

</snapshotRepository>

</distributionManagement>

</project>

注意里面的

<parent>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

这就指定了当前的SpringMVC工程，是继承了SpringPOM工程来的。也就是说，虽然我们在SpringMVC工程里面没有定义对Spring组件相关的依赖，但是因为继承了SpringPOM，所有它一样的有引入了Spring组件的依赖。

pom.xml中可以定义那么多元素，到底有哪些元素可以继承呢？下面是可以继承的元素列表和简单说明：

1. groupId：项目组Id，项目坐标的核心元素
2. Version：项目版本，项目坐标的核心元素
3. Description：项目的描述信息
4. Organization：项目的组织信息
5. inceptionYear：项目的创始年份
6. Url：项目的url地址
7. Developers：项目的开发者信息
8. Contributors：项目的贡献者信息
9. distributionManagement：项目的部署配置
10. issueManagement：项目的bug跟踪系统信息
11. ciManagement：项目的持续集成系统信息
12. Scm：项目的版本控制系统信息
13. mailingLists：项目的右击列表信息
14. Properties：自定义的Maven属性
15. Dependencies：项目的依赖配置
16. dependencyManagement：项目的依赖管理配置
17. Repositories：项目的仓库配置
18. Build：包括项目的源代码目录配置、输出目录配置、插件配置、插件管理配置等
19. Reporting：包括项目的报告输出目录配置、报告插件配置等

这里有个比较特别的元素，dependencyManagement元素，根据前面的简介，我们知道它是依赖管理元素，也就是说，用来管理依赖的。因为在实际项目中，它有特殊意义，而且能够被继承，我们这样多介绍一点。

一个Maven项目要直接引用某个依赖，我们都是在dependencies中，使用dependency描述要引用依赖的坐标信息来完成的，这样就达到了一个要什么，直接写什么的效果，决定权都在是否用dependency指定了引用构件的坐标。但是实际项目管理过程中，我们可以有个全局的管理，也就是说，整个项目要引用的依赖，事先分析整理好，形成一个全局的集合，当某个Maven模块需要具体引用哪些依赖的时候，直接在这集合中指定若干个。这样就可以实现整个项目依赖的全局管理，不至于零碎的分布在每个Maven模块中去，最后用到了哪些依赖都需要逐个统计，更加不用说对依赖进行合理的优化。

对的，基于这样的考虑，我们就可以在一个公共的pom.xml中，使用dependencyManagement元素，将所有的依赖都声明管理好，如下代码：

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<name>SpringPOM</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<!-- 3.2.16.RELEASE,3.1.4.RELEASE -->

<project.build.spring.version>4.2.7.RELEASE</project.build.spring.version>

</properties>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.7</version>

<scope>test</scope>

</dependency>

<!-- spring -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

...

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>5.0.0.Final</version>

</dependency>

<!-- jsp servlet -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

...

<!-- json组件 -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.5.4</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-core</artifactId>

<version>2.5.4</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-annotations</artifactId>

<version>2.5.0</version>

</dependency>

</dependencies>

</dependencyManagement>

...

</project>

在项目中如果要使用前面在SpringPOM中用dependencyManagement管理的Spring-core构件的话，只需继承SpringPOM，然后在自己的dependencies中指明spring-core就行，代码如下：

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<!-- 继承SpringPOM中的pom -->

<parent>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>SpringMVCPOM</artifactId>

<packaging>pom</packaging>

<name>SpringMVCPOM</name>

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-core</artifactId>**

**</dependency>**

...

</dependencies>

...

</project>

注意粗体部分，dependency里面是没有指定version的，而且还不用配置scope，因为这些信息已经在SpringPOM中配置好了，我们这里只要通过groupId和artifactId指明是在dependencyManagement中配置的哪个，其它就自动明了了。虽然使用这种依赖管理机制，不能减少太多的pom配置，不过还是建议大家使用这种方式。原因应该大家能理解：在父pom中使用dependencyManagement声明依赖统一管理了项目中使用到的依赖种类、版本，每个子项目就不会出现额外的多余依赖，特别是没有优化的依赖和同一个构件的不同版本了。

当然了，我们在子项目中，处理可以用copy、继承的方式，将定义在父项目中的dependencyManagement中管理的构件信息，合并到当前Maven工程中来外，也可以用依赖作用范围import，将它们合并过来，样例代码如下：

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringDemo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>cn.com.mvnbook.pom</groupId>

<artifactId>SpringPOM</artifactId>

<version>0.0.1-SNAPSHOT</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

</project>

前面我们介绍了继承，能将公共的信息定义在一个pom中，通过集成来减少重复配置和统一标准的效果。同样的，根据我们前面基于SSH和SSM框架实现的用户CRUD功能来分析，实际项目中会将多个功能分别用不同的模块单独实现。一个项目用不同模块单独实现，确实能给我们项目开发的独立性提供了方便，但是构建测试就不一样了，如果没有特殊的解决方案，就需要每个模块独立的管理，这样比较麻烦。

所以对于继承，Maven里面还有个聚合机制。聚合机制能用一个独立的Maven工程，里面没有代码，只有pom.xml，将相关Module模块合并起来，这样每个构建过程（命令）只要在独立的pom Maven工程上进行，Maven自动会将包含的每个模块工程同步构建。

其实我们前面的SSM和SSH样例就是按这样的思路实现的，下面是SSM聚合项目中的pom.xml：

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.com.mvnbook.ssm.demo</groupId>

<artifactId>MvnBookSSMDemo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<name>MvnBookSSMDemo</name>

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

**<modules>**

**<module>../MvnBookSSHDemo.DAO</module>**

**<module>../MvnBookSSHDemo.Service</module>**

**<module>../MvnBookSSMDemo.DAO.MyBatis</module>**

**<module>../MvnBookSSMDemo.Service.Impl</module>**

**<module>../MvnBookSSMDemo.SpringMVC</module>**

**</modules>**

...

</project>

通过modules和module，将相关的独立Maven模块工程配置在一起就完成了Maven模块的聚合，就可以统一的实现所有相关Maven模块的管理了。

# 第十章 Maven测试

## 10.1 测试简介

软件测试时软件开发过程中的一个重要组成部分，是贯穿整个软件开发生命周期、对软件产品（包括阶段性产品）进行验证和确认的活动过程，其目的是尽快尽早的发现在软件产品中存在的各自问题（与用户需求和预先的定义不一致）。

从不同角度，我们可以对软件测试（软件测试技术）有不同的分类：

从是否需要执行被测试软件的角度，我们可以把软件测试分为静态测试和动态测试；

从是否针对软件结构与算法进行测试的角度，我们可以把软件测试分为白盒测试和黑盒测试；

从测试的阶段角度，我们可以把软件测试分为单元测试、集成测试、系统测试和验收测试；

不管是什么测试，都基本上有如下几个环节（阶段）：制订测试计划、 分析设计测试、开发测试、运行测试、整理测试报告。

我们这里主要研究的是怎样基于jUnit和TestNG测试框架编写测试案例、基于Maven高效率的执行测试代码和形成规范的测试报告。

## 10.2 测试框架

目前常用的Java单元测试框架，是JUnit和在JUnit基础上进一步扩展的TestNG，为了能很好的在Maven中完成测试案例的执行和形成测试报告，我们先研究一下怎么样在JUnit和TestNG框架下编写测试代码。

### 10.2.1 JUnit单元测试框架

JUnit是由Erich Gamma和Kent Beck编写的一个回归测试框架，是一个开放源代码的Java测试框架，我们可以在它的基础上编写和运行可重复的测试。JUnit框架有如下几个特点：

1. 使用断言测试期望结果
2. 能共享共同测试数据
3. 方便组册和运行测试
4. 支持图形化和文件的测试运行器

jUnit测试框架的安装比较简单，只需要下载JUnit的最新压缩包，在本地解压后，配置好JUNIT\_HOME环境变量，并且在CLASSPATH目录中，追加好jUnit的jar包就行。

对于IDE环境的用户，只需要将jUnit的jar包，添加到项目的build path中就行了。

接下来我们回顾梳理（以前样例中有编写，只是没有系统介绍）一下怎么样在一个Maven项目中，基于jUnit编写测试案例。

在Maven项目中，基于jUnit编写测试案例，一般要做两个方面的事情：一是在pom.xml中添加jUnit依赖；一是基于jUnit规范，编写测试代码。

如下是MvnBookSSMDemo.Service.Impl项目中，关于jUnit的配置和代码：

1. 在pom.xml中的jUnit依赖配置  
   （随书代码[MvnBookSSMDemo.Service.Impl\pom.xml](代码/MvnBookSSMDemo.Service.Impl/pom.xml)）

<project>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.7</version>

<scope>test</scope>

</dependency>

... ...

</dependencies>

... ...

</project>

1. TestUserServiceImpl.java类  
   （随书代码  
   [MvnBookSSMDemo.Service.Impl\src\test\java\cn\com\mvnbook\ssh\demo\service\impl\TestUserServiceImpl.java](代码/MvnBookSSMDemo.Service.Impl/src/test/java/cn/com/mvnbook/ssh/demo/service/impl/TestUserServiceImpl.java)）

package cn.com.mvnbook.ssh.demo.service.Impl;

import java.util.List;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import cn.com.mvnbook.ssh.demo.entity.MvnUser;

import cn.com.mvnbook.ssh.demo.entity.Status;

import cn.com.mvnbook.ssh.demo.service.IUserService;

import junit.framework.Assert;

public class TestUserServiceImpl {

private IUserService userService;

private ApplicationContext ctx = null;

@Before

public void init(){

this.ctx = new ClassPathXmlApplicationContext("applicationContext.xml");

this.userService = (IUserService)ctx.getBean("userService");

}

@Test

public void testCreateUser(){

MvnUser user = new MvnUser();

user.setUrAge(11);

user.setUrPassword("11");

user.setUrStatus(Status.ACTIVE.getStatus());

user.setUrUserName("service1");

this.userService.createUser(user);

MvnUser u = this.userService.searchUser("service1");

boolean bool = u != null && u.getUrAge()==11 && u.getUrStatus().equals(Status.ACTIVE.getStatus());

Assert.assertTrue(bool);

// 删除用户

this.userService.deleteUser(u.getUrId());

}

@Test

public void testEditUser(){

MvnUser user = new MvnUser();

user.setUrAge(11);

user.setUrPassword("11");

user.setUrStatus(Status.ACTIVE.getStatus());

user.setUrUserName("service1");

this.userService.createUser(user);

MvnUser u = this.userService.searchUser("service1");

this.userService.editUser(88, Status.INACTIVE.getStatus(), u.getUrId());

u = this.userService.searchUser("service1");

Assert.assertTrue(u.getUrAge()==88  
 && u.getUrStatus().equals(Status.INACTIVE.getStatus()));

this.userService.deleteUser(u.getUrId());

}

@Test

public void testDeleteUser(){

MvnUser user = new MvnUser();

user.setUrAge(11);

user.setUrPassword("11");

user.setUrStatus(Status.ACTIVE.getStatus());

user.setUrUserName("service1");

this.userService.createUser(user);

MvnUser u = this.userService.searchUser("service1");

this.userService.deleteUser(u.getUrId());

MvnUser u2 = this.userService.searchUser(u.getUrId());

Assert.assertTrue(u != null && u2 == null);

}

@Test

public void testSearchUserById(){

MvnUser user = this.userService.searchUser(1);

Assert.assertNotNull(user);

}

@Test

public void testSearchUserByUserName(){

MvnUser user = this.userService.searchUser("zhangsan");

Assert.assertNotNull(user);

}

@Test

public void testSearchUsers(){

List<MvnUser> userList = this.userService.searchUsers();

Assert.assertTrue(userList != null && userList.size()>0);

}

@After

public void destory(){

this.userService = null;

this.ctx = null;

}

}

Pom.xml中的jUnit依赖配置，我们在这里就不过多重复了，这里主要说明一下测试代码的注意事项：

1. 在Maven项目中，测试代码有专门的默认目录：src/test/java
2. 一般测试案例代码的包，同要测试的目标类的包一样
3. 测试代码的类的命名，一般是 Test+目标测试类的类名
4. 测试代码中的方法，一般是有三种：

1）使用@Before标记的，实现初始化执行测试代码需要的资源

2）使用@Test标记的，跟测试目标类的每个方法一一对应的测试代码

3）使用@After标记的，完成测试后需要释放的资源

1. 测试方法的逻辑一般是：

1）准备好测试数据

2）根据测试工具和用户需求（目标代码的实现），确定期望结果

3）执行测试方法，获取实际结果

4）断言实际结果是否同期望结果一致

### 10.2.2 TestNG测试框架

TestNG是一个测试框架，是一个开源的自动化测试框架。很多地方把TestNG理解成JUnit、特别是JUnit4的下一代。实际上它不只是简单的扩展JUnit，它是一个灵感源于JUnit，目的是为了更优于JUnit的自动测试框架，同JUnit是独立的，由Cedric Beust创建的开源框架。

TestNG消除了大部分旧框架的限制，使开发人员能够编写更加灵活、更加强大的测试，而且很大程度上借鉴了Java注解，使我们的测试代码更好的同Java新特征整合。

TestNG相对其它测试框架，有如下自身的特点：

1. 使用简单的注解说明测试方法
2. TestNG使用Java和面向对象编程
3. 支持综合测试
4. 独立的编译时间、独立的运行测试代码的配置和数据
5. 灵活的运行时配置
6. 支持测试组设置和运行
7. 支持依赖测试、并行测试、负载测试和局部测试
8. 灵活的插件API
9. 支持多线程测试

在Maven项目中编写和运行TestNG是比较方便的。首先要移除以前在POM中配置的jUnit依赖，添加TestNG依赖，如下代码：

<dependency>

<groupId>org.testng</groupId>

<artifactId>testng</artifactId>

<version>5.9</version>

<scope>test</scope>

<classifier>jdk15</classifier>

</dependency>

同jUnit类似，TestNG的依赖范围是test。另外，TestNG使用classifier jdk15和jdk14为不同的Java平台提供支持。

接下来就是在测试代码中，将以前引用的jUnit的注解、类改成TestNG的。注解名称和类名都一样，只是包名不同，常用的类如下：

org.testng.annotations.Test 测试方法的注解

org.testng.annotations.BeforeMethod 测试方法前运行的方法注解

org.testng.annotations.AfterMethod 测试方法运行后执行的方法注解

org.testng.annotations.BeforeClass 所有测试方法运行执行执行的方法注解

org.testng.annotations.AfterClass 所有测试方法运行后执行的方法注解

org.testng.Assert 断言类

同运行jUnit一样，直接用mvn test命令，Maven就会自动执行符合命名模式的测试类。

TestNG除了可以同jUnit一样，自动执行符合命名模式的测试类外，还可以通过testng.xml配置文件配置需要运行的测试集合。如我们可以在Maven项目的根目录下创建一个testng.xml文件如下：

<?xml version=’1.0’ encoding=’utf-8’ ?>

<suite name=”TestSuite” verbose=”1”>

<test name=”test1”>

<classes>

<class name=”cn.com.mvnbook.demo.TestNGDemo”/>

</classes>

</test>

</suite>

同时，在maven-surefire-plugin中，声明testng.xml，代码如下：

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>2.16</version>

<configuration>

<suiteXmlFiles>

<suiteXmlFile>testng.xml</suiteXmlFile>

</suiteXmlFiles>

</configuration>

</plugin>

另外，TestNG相对jUnit来说，还有一个大的优势是，可以使用注解的方式，对测试方法进行分组标记，运行的时候，指定只执行哪个组的测试方法，或哪些组测试方法。

如：

@Test{groups = {“group1”,”group2”}}

表示将对应的方法加入group1组和group2组

接下来，我们可以在maven-surefire-plugin插件中，配置运行哪些组。如

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>2.16</version>

<configuration>

<groups>group1,group3</groups>

</configuration>

</plugin>

表示执行当前TestNG的时候，只会执行group1和group2两个组的测试方法。

## 10.3 Maven测试插件

前面我们介绍了两个比较流行的测试框架。在Maven项目中，我们基于jUnit或TestNG编写好了测试代码，接下来怎么执行，并且形成测试报告呢？具体执行测试代码，就需要靠maven-surefire-plugin插件来实现了。

### 10.3.1 Surefire插件简介

Maven本身虽然不是测试框架，但是Maven能够在构建执行到特定的生命周期阶段的时候，能通过调用插件来执行基于jUnit和TestNG编写好的测试用例。这个插件就是maven-  
surefire-plugin插件，它能很好的兼容jUnit系列和TestNG测试框架。

在Maven的default生命周期的test阶段，绑定的默认插件是maven-surefire-plugin，这是一个内置绑定，在默认情况下，maven-surefire-plugin的test阶段，会自动执行测试源代码路径下的所有符合命名模式的测试类。符合命名模式的规范是：

1. \*\*/Test\*.java：所有命名为Test开头的java类
2. \*\*/\*Test.java：所有命名为Test结尾的java类
3. \*\*/\*TestCase.java：所有命名为TestCase结尾的java类

只要测试类符合上面册命名模式，Maven都会自动的运行它们，我们不需要再定义测试集合来声明要执行哪些测试类。

如果我们的测试代码是基于TestNG框架的，还可以通过配置文件，灵活的指定需要运行的测试案例类。具体的写法，请参考前面关于TestNG的介绍。

### 10.3.2 跳过测试

在实际项目中，我们喜欢Maven的构建过程暂时跳过测试环境（不运行测试案例），我们可以在执行mvn命令的后面，通过添加skipTests参数实现。如：

Mvn package -DskipTests

执行这命令，我们就可以跳过测试阶段。

虽然我们可以通过该方式跳过测试阶段，但是这种操作和思路是不提倡的，因为发布和使用一个没有测试的构件，是有很大不可预料的风险的。

我们除了可以在mvn命令后面，通过指定skipTests参数标明跳过测试外，还可以在pom中的maven-surefire-plugin插件配置中进行声明。如：

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>2.16</version>

<configuration>

<skipTests>true</skipTests>

</configuration>

</plugin>

我们除了可以通过上面介绍的方式，在Maven构建过程中，跳过执行测试代码，我们还可以直接跳过对测试的编译。

同样的，我们可以通过命令行指定，也可以通过pom的配置文件指定。

Maven命令指定如：

Mvn package -Dmaven.test.skip=true

其中maven.test.skip参数，同时控制maven-compile-plugin和maven-surefire-plugin两个插件的行为：既跳过了测试代码的编译，也跳过了测试代码的运行。

pom配置文件指定如：

<plugin>

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

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

<version>2.1</version>

<configuration>

<skip>true</skip>

</configuration>

</plugin>

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>2.16</version>

<configuration>

<skip>true</skip>

</configuration>

</plugin>

### 10.3.3 个性化指定运行测试

在实际项目中，根据需要，我们可能需要灵活的指定执行某些测试案例，这时候，我们可以通过mvn命令的test参数来实现。如：

Mvn test -Dtest=TestDemo

表示只执行TestDemo测试类。

同样，我们还可以在test参数中，用\*通配符指定执行符合规则的所有测试类。如：

Mvn test -Dtest=Test\*Demo

表示执行类名为Test开头，Demo结尾的所有测试类。

如果我们不嫌麻烦，也可以通过test参数，明确指定要执行的测试类名。如：

Mvn test -Dtest=TestDemo1,TestDemo2

表示执行类名为TestDemo1和TestDemo2的所有测试类。注意，指定的多个测试类之间，用逗号隔开。

同样，我们也可以将\*通配符和明确指定测试类的两种方式结合起来，以达到更灵活的效果。如：

Mvn test -Dtest=TestA\*,TestDemo1,TestDemo2

表示运行类名以TestA开头的所有的测试类，和TestDemo1、TestDemo2测试类。

最后需要说明的是，如果maven-surefire-plugin根据test参数，找不到任何匹配的测试类的话，会报测试失败，从而导致构建失败。如

Mvn test -Dtest

这样的命令肯定会找不到一个测试类，就会导致构建失败。

### 10.3.4 包含和排除测试

Maven提倡约定优于配置原则，所以我们在写测试代码的时候，尽量按规范的模式，给测试类起名字。但是有时候，就难免会出现一些不符合模式的测试类名，而这些测试类又需要执行，我们可以通过如下方式，在maven-surefire-plugin中进行配置实现：

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>2.16</version>

<configuration>

<includes>

<include>\*\*/Demo\*.java</include>

</includes>

</configuration>

</plugin>

通过上面的配置，surefire插件就会自动执行所有命名以Demo开头的测试类，同样的，我们可以通过excludes描述排除哪些测试类。如：

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>2.16</version>

<configuration>

<excludes>

<exclude>\*\*/\*abc.java</exclude>

<exclude>\*\*/Temp\*.java</exclude>

</excludes>

</configuration>

</plugin>

表示不执行所有命名以abc结尾的测试类和以Test开头的测试类。

## 10.4 测试报告

在Maven构建过程中，我们除了可以通过查看命令行的提示信息，了解测试状况和结果外，还可以使用Maven的相关插件，生成专业统一的测试报告，方便归档、查看和提交测试状况和结果。

### 10.4.1 基本测试报告

缺省情况下，maven-surefire-plugin插件会在Maven项目的target/surefire-reports目录下生成两种格式的错误报告：一个是简单文本格式；一个是与JUnit兼容的xml格式。下面是MvnBookSSMDemo.Service.Impl项目中TestUserServiceImpl测试案例运行后的报告样例：

1. 文本测试报告内容

-------------------------------------------------------------------------------

Test set: cn.com.mvnbook.ssh.demo.service.Impl.TestUserServiceImpl

-------------------------------------------------------------------------------

Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.906 sec

Xml测试报告内容

<?xml version="1.0" encoding="UTF-8" ?>

<testsuite failures="0" time="4.773" errors="0" skipped="0" tests="6"   
 name="cn.com.mvnbook.ssh.demo.service.Impl.TestUserServiceImpl">

<properties>

<property name="java.runtime.name" value="Java(TM) SE Runtime Environment"/>

... ...

</properties>

<testcase time="2.669"   
 classname="cn.com.mvnbook.ssh.demo.service.Impl.TestUserServiceImpl"   
 name="testEditUser"/>

... ...

</testsuite>

如果要生成html的测试报告，我们需要在pom.xml中添加maven-surefire-report-plugin插件的配置，样例配置代码如下：

<build>

<plugins>

<plugin>

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

<artifactId>maven-surefire-report-plugin</artifactId>

<version>2.12.2</version>

<configuration>

<showSuccess>false</showSuccess>

</configuration>

</plugin>

</plugins>

</build>

运行测试后，Maven会自动生成html版的测试报告，内容同文本和xml版本的一样，只是展现的形式是html，方便项目相关人员和客户查看。

### 10.4.2 测试覆盖率报告

测试报告主要报告的是测试代码的运行结果是否正确，至于对软件的测试质量，要依靠测试代码本身的设计和实现。设计和实现得比较详细周到的话，测试的质量就高，否则就会有很多情况没有测试到，造成软件的bug不能被及时发现。

所以为了提高软件本身的质量，我们除了要设计编写测试案例代码进行测试之外，还必须对测试案例的代码测试的范围进行一个把握，从而进一步的保证软件代码的质量。所以测试覆盖率也是衡量软件代码质量的一个重要参考指标。

Cobertura是一个优秀的测试覆盖率统计工具，在Maven项目中，我们可以通过集成cobertura-maven-plugin插件，再执行简单的maven命令：mvn cobertura:cobertura，就可以生成测试覆盖率报告。

当然，要能顺利的生成测试覆盖率报告，我们需要在pom.xml中配置Cobertura插件，样例配置如下：

<build>

<plugins>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>cobertura-maven-plugin</artifactId>

<version>2.5.1</version>

</plugin>

</plugins>

</build>

Cobertura处理有前面介绍的生成测试报告的命令外，还有如下命令：

常用命令

　　mvn cobertura:help 查看cobertura插件的帮助

　　mvn cobertura:clean 清空cobertura插件运行结果

　　mvn cobertura:check 运行cobertura的检查任务

　　mvn cobertura:cobertura 运行cobertura的检查任务并生成报表，报表生成在target/site/cobertura目录下

　　cobertura:dump-datafile Cobertura Datafile Dump Mojo

　　mvn cobertura:instrument Instrument the compiled classes

　　在target文件夹下出现了一个site目录，下面是一个静态站点，里面就是单元测试的覆盖率报告

## 10.5 重用测试代码

在项目开发过程中，我们经常要将公共的功能代码打包共享给其它模块重复使用。还有在历史项目开发过程中，沉淀下的公共框架代码，我们更加需要形成独立的jar，以便以后的类似项目中重复实现。我们可以使用mvn package命令，将这些代码大虫包，构建成构件，发布到仓库中共享，以便其它项目可以做依赖构件使用。

同样的，对那些有良好设计，能够重复使用在项目的不同模块中、甚至不同项目中的测试代码，我们也需要打包形成构件，以便当依赖一样重复使用，从而减少编写测试代码的工作量。而mvn package只会对主代码和资源文件进行打包安装和部署，对测试代码和资源文件是不会处理的。

为了实现将测试代码和资源文件打包安装和部署，我们可以在pom中配置maven-jar  
-plugin插件，配置代码如下：

<plugin>

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

<artifactId>maven-jar-plugin</artifactId>

<version>2.2</version>

<executions>

<execution>

<goals>

<goal>test-jar</goal>

</goals>

</execution>

</executions>

</plugin>

Maven-jar-plugin有两个目标：一个是jar；一个是test-jar。Jar目标有内置绑定到maven的default生命周期的package阶段。当我们对maven工程进行构建的时候，自动执行，将项目的主代码和资源文件进行打包。Test-jar目标没有内置绑定，所以需要我们自己在插件配置中显式的声明该目标，从而达到在maven工程构建的时候，同时将测试代码和资源文件打包。如上声明，test-jar目标是默认绑定到default生命周期的package阶段，所以当我们运行mvn clean package命令的时候，能同时将主代码和测试代码分别打包。

最后需要说明一下的是，当我们使用测试构件依赖的时候，需要指定依赖的type为test-jar，样例代码如下：

<dependency>

<groupId>...</groupId>

<artifactId>...</artifactId>

<version>...</version>

<type>test-jar</type>

<score>test</score>

</dependency>

# 第十一章 灵活构件Maven项目

在项目开发过程中，一般都会有开发环境、测试环境和正式运行环境，这些环境的数据库配置一般都不一样，那么项目构建的时候，就需要根据不同的环境，选择对应的配置。同样的，一个项目在开发过程中，肯定会形成大量的基础测试，每次运行完所有的测试案例是一个庞大的工程，做全面测试的时候肯定都要运行，而如果每次构建项目的时候运行的话，就有点不合适了，这时候就需要有一种方式，能让我们灵活的指定在特定的时候执行对应的测试案例。

为了应对这些在实际项目中频繁遇到的问题，Maven提供了动态灵活构建的机制。Maven提供属性、资源过滤和profile三大特征，我们可以在pom和资源文件中使用Maven属性表示可能变法的量，通过不同的profile中的属性值和资源过滤特征，为不同环境执行不同的构建。接下来我们将介绍怎么样使用Maven的这三个特征，实现项目在不同环境下的灵活构建的配置和操作。

## 11.1 Maven属性

对Maven属性的使用，其实我们应该不陌生的，在我们前面的样例工程的pom文件中，都有使用过，比如SpringPOM工程中的pom.xml有如下代码：

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.build.spring.version>4.2.7.RELEASE</project.build.spring.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>${project.build.spring.version}</version>

</dependency>

.... ....

</dependencies>

在上面的代码中，我们就在properties中定义了project.build.spring.version属性，描述了Spring的4.2.7.RELEASE版本信息，然后再依赖的version中使用${project.build.spring.version}使用，从而统一Spring所有相关构件的版本，不仅仅能降低错误的发生概率，而且还能减少升级版本的工作量（不用每个构件都去修改版本，只要修改属性描述的版本就行）。

前面只是Maven属性中的一种体现形式，在Maven中，一共提供了六类属性，它们分别是：

1. 内置属性

在Maven中，主要有两个常用的内置属性，它们分别是${basedir}和${version}属性。${basedir}表示项目的根目录，也就是包含pom.xml文件的目录；${version}表示项目的版本。

1. POM属性

我们可以通过POM属性，引用POM文件中对应元素的值，比如${project.artifactId}就对应project->artifactId元素的值。常用的POM属性包括：

1. ${project.build.sourceDirectory}:项目的主源码目录，默认是src/main/java
2. ${project.build.testSourceDirectory}:项目的测试源码目录，默认是src/test/java
3. ${project.build.directory}:项目构建输出目录，默认是target
4. ${project.outputDirectory}:项目主代码编译输出目录，默认是target/classes
5. ${project.testOutputDirector}:项目测试代码编译输出目录，默认是target/testclasses
6. ${project.groupId}:项目的groupId
7. ${project.artifactId}:项目的artifactId
8. ${project.version}:项目的版本
9. ${project.build.finalName}:项目答辩输出的文件名称，默认为${project.artifactId}-  
   ${project.version}。

这些属性，都在pom中有对应的元素，它们中的一些属性的默认值，都在超级pom中有定义，详细情况可以参考超级pom.xml。Maven3的超级pom文件在$MAVEN\_HOME/lib/maven-model-builder-x.x.x.jar中的org/apache/maven/model目录下，文件名为pom-4.0.0.xml。

1. 自定义属性

我们可以在pom的properties中，定义我们自己的Maven属性，然后在后面重复使用，同我们在前面提到的SpringPOM的pom.xml中，定义的spring的版本信息一样。

1. Settings属性

Settings属性同POM属性是一样的，我们可以用以settings.开头的属性，引用settings.xml文件中xml元素的值。如使用${settings.localRepository}指向用户本地仓库的地址。

1. Java系统属性

所有的Java系统属性，我们都可以通过Maven属性引用使用，比如${user.home}指向的就是用户目录。我们可以通过使用mvn help:system查看所有的Java系统属性，如：

PROGRAMW6432=C:\Program Files

COMMONPROGRAMW6432=C:\Program Files\Common Files

WINDOWS\_TRACING\_LOGFILE=C:\BVTBin\Tests\installpackage\csilogfile.log

PROCESSOR\_ARCHITECTURE=AMD64

VISUALSVN\_SERVER=C:\Program Files (x86)\VisualSVN Server\

CLASSWORLDS\_LAUNCHER=org.codehaus.plexus.classworlds.launcher.Launcher

PROGRAMDATA=C:\ProgramData

... ...

1. 环境变量属性

所有的环境变量，我们都可以用以evn.开头的Maven属性引用。比如${evn.JAVA\_HOME}就指向引用了JAVA\_HOME环境变量的值。同查看Java系统属性一样，我们可以使用命令mvn help:system查看到所有的环境变量。

在Maven项目中，适当的使用用这些Maven属性，可以简化我们的pom的配置和维护工作，而且能将公共的信息很好的统一起来，避免不必要的不一致错误。

## 11.2 需要灵活处理的构建环境

在不同的环境里面，项目的源代码需要使用不同的方式进行构建，从而适应对应环境的需要。比如数据库配置，在开发过程中，我们会将数据库的配置信息放在src/main/resources目录下，配置的是基于oracle数据库驱动，使用Oracle数据库中用户名和密码，连接Oracle数据库，但是为了方便，测试的时候，我们需要基于MySQL数据库驱动，使用MySQL用户名和密码，连接MySQL数据库。为了解决这样的矛盾问题，我们可以在测试的时候，将resources中的连接数据库的信息，改成MySQL数据库相关的信息，正式构件发布前，把数据库的配置信息改成Oracle相关的连接信息。

平常的时候，我们就是经常这么操作的，但是，如果有一种能自动实现切换的方法的话，那这种方式是不是就严重过时了呢？

是的，Maven就针对这种情况提供了自动切换的方式方法。

## 11.3 资源过滤

为了适应环境的变化，我们需要使用Maven属性，将这些将会变化的部分提出出来，用一个特殊的方式--Maven属性描述它们，这个道理同我们在代码中定义变量，用一个变量描述一个值一样。比如上面提到的数据库配置，我们就可以在配置文件中，用Maven属性代替以前写死的配置信息，如：

db.jdbc.driverClass=${db.driver}

db.jdbc.connectionURL=${db.url}

db.jdbc.username=${db.username}

db.jdbc.password=${db.password}

这样我们就定义四个Maven属性，它们的名称分别是db.driver, db.url, db.username, db.password。这些名称，我们可以根据自己的需要和习惯自定义。

在配置文件中，我们用${xxx}的方式，用到了Maven属性，用它们代替以前写死在配置文件的连接数据的信息，接下来，我们就需要在某个地方给这些Maven属性赋予真实的，能用来连接数据库的值了。

我们可以按前面章节定义Maven属性的方式，在pom.xml的properties中定义Maven属性，并且给它们赋值。但是不方便根据环境的变化而自动切换，我们这里介绍一下用profile的方式进行定义。如：

<profiles>

<profile>

<id>dev\_evn</id>

<properties>

<db.driver>com.mysql.jdbc.Driver<db.driver>

<db.url>jdbc:mysql://localhost:3306/test<db/url>

<db.username>root</db.username>

<db.password>root</db.password>

</properties>

</profile>

</profiles>

这里的声明方式，同在properties元素中声明的方式实际上是一样的，唯一不同的是，把它们放在profile中进了包装，同时用一个名称为dev\_evn的id标识了，这样就可以将开发环境的配置，同其它环境的配置区分开来。

现在定义了Maven属性，同时在配置文件中也使用这些定义好的Maven属性，是不是就可以正常构建起来了呢？当然还不行的，因为Maven属性，只能在pom.xml中才能被解析出来。也就是说，${db.username}在pom中，可以被Maven解析成root，但是如果放在资源目下：src/main/resources目录下的文件中的话，Maven在构建的时候，是不能将${db.username}转变成root。这时候就需要想办法让Maven能解析资源文件中的Maven属性。

在Maven中，对资源文件进行处理的工作，是由maven resource plugin插件完成的，它默认的工作是将项目主资源文件复制到主代码编译输出目录中，将测试资源文件，复制到测试代码编译输出目录中。当然，我们也可以通过一些简单的pom配置，让maven Resource plugin插件能解析资源文件中的Maven属性，也就是开启资源过滤功能。

Maven是在超级POM中定义的主资源目录和测试资源目录，为了开启资源过滤功能，我们需要在pom中添加一个filtering配置就行，代码如下：

<resources>

<resource>

<directory>${project.basedir}/src/main/resources</directory>

<filtering>true</filtering>

</resource>

</resources>

<testResources>

<testResource>

<directory>${project.basedir}/src/test/resources</directory>

<filtering>true</filtering>

</testResource>

</testResources>

根据前面的配置，实际上我们可以实现多个主资源目录和测试资源目录的配置，虽然会稍稍破坏Maven的部分约束。而且我们可以根据需要，将部分资源目录的过滤功能开启，而其它的资源目录的过滤功能屏蔽。

按照我们项目的需要，遵循前面的规则，在pom中配置好后，就可以在mvn命令中，用-P参数指定激活指定id对应的profile，进行动态构建了。比如执行：

Mvn clean install -Pdev\_evn

Maven在执行的时候，先激活id为dev\_evn，基于配置好的Maven属性，将配置的数据库连接信息，构建到Maven项目中去。

## 11.4 Maven的Profile

为了实现不同环境构建的不同需求，我们使用到了Profile。因为profile能够在构建的时候修改POM的一个子集，或者添加额外的配置元素。接下来我们详细研究一下Maven中对profile的配置和激活。

**11.4.1 针对不同环境的profile的配置**

为了体现不同环境的不同构建，我们需要配置好不同环境的profile，如下代码：

<profiles>

<profile>

<id>dev\_evn</id>

<properties>

<db.driver>com.mysql.jdbc.Driver<db.driver>

<db.url>jdbc:mysql://localhost:3306/test<db/url>

<db.username>root</db.username>

<db.password>root</db.password>

</properties>

</profile>

<profile>

<id>test\_evn</id>

<properties>

<db.driver>com.mysql.jdbc.Driver<db.driver>

<db.url>jdbc:mysql://localhost:3306/test\_db<db/url>

<db.username>root</db.username>

<db.password>root</db.password>

</properties>

</profile>

</profiles>

在两个不同profile中，配置了同样的属性，不一样的值，按照前面的介绍，我们在开发的时候，可以是用mvn命令后面添加-Pdev\_evn激活dev\_evn profile，在测试的时候，使用mvn命令后面添加-Ptest\_evn激活test\_evn profile

### 11.4.2 激活profile配置

在Maven中，我们可以选用如下的多种方式激活Profile

1. 命令行激活

我们可以在mvn命令行中，添加参数-P，指定要激活的profile的id，如果一次要激活多个profile，可以用逗号分开一起激活。如：

mvn clean install -Pdev\_env,test\_evn

这个命令就同时激活了id为dev\_evn和test\_evn的两个profile

1. Settings文件显示激活

如果我们希望某个profile默认一直处于激活状态，就可以在settings.xml中，配置activeProfiles元素，指定某个profile为默认激活状态，配置如下：

<settings>

...

<activeProfiles>

<activeProfile>dev\_evn</activeProfile>

</activeProfiles>

...

</settings>

1. 系统属性激活

我们可以通过配置当某个系统属性存在时，激活profile，代码如下：

<profiles>

<profile>

...

<activation>

<property>

<name>profileProperty</name>

</property>

</activation>

</profile>

</profiles>

甚至，我们还可以进一步配置某个属性的值为什么时激活，如：

<profiles>

<profile>

...

<activation>

<property>

<name>profileProperty</name>

<value>dev</value>

</property>

</activation>

</profile>

</profiles>

这样我们就可以在mvn中用-D参数来指定激活，如：

Mvn clean install -DprofileProperty=dev

表示激活属性名称为profileProperty，值为dev的profile

实际上这也是一种命令激活profile的方法，只是用的是-D参数指定激活的属性和值，而前面的是用的-P参数指定激活的profile的id而已。

1. 操作系统环境激活

我们可以通过配置，指定不同操作系统的信息，实现不同操作系统做不同的构建。如：

<profiles>

<profile>

<activation>

<os>

<name>Window XP</name>

<family>Windows</family>

<arch>x86</arch>

<version>5.1.2600</version>

</os>

</activation>

</profile>

</profiles>

Familyd的值是Window、UNIX或Mac，name为操作系统名称，arch为操作系统的架构，version为操作系统的版本，具体的值，我们尅通过查看环境中的系统属性os.name，os.arch和os.version获取。

1. 文件存在与否激活

当然，我们也可以通过配置判断某个文件存在与否，来决定是否激活profile，配置如下：

<profiles>

<profile>

<activation>

<file>

<missing>t1.properties</missing>

<exists>t2.properties</exists>

</file>

</activation>

</profile>

</profiles>

1. 默认激活

最后，我们还可以配置一个默认的激活profile，如：

<profiles>

<profile>

<activation>

<activeByDefault>true</activeByDefault>

</activation>

</profile>

</profiles>

需要注意的是，如果pom中有任何一个profile，通过其它方式被激活的话，所有配置成默认激活的profile都会自动失效。

我们可以使用如下命令查看当前激活的profile：

Mvn help:active-profiles

也可以用如下命令，查看所有的profile:

Mvn help:all-profiles

### 11.4.3 profile的种类

前面我们介绍了profile的意义和激活方式，那在Maven中有哪些profile，怎么配置了，接下来我们一起研究一下。

根据profile配置的位置不同，我们可以将profile分成如下几种：

1. Pom.xml

Pom.xml中声明的profile只对当前项目有效

1. 用户settings.xml

在用户目录下的.m2/settings.xml中的profile，对本机上的该用户的所有Maven项目有效。

1. 全局settings.xml

在Maven安装目录下conf/settings.xml中配置的profile，对本机上所有项目都有效。

为了不影响其它用户，且方便升级Maven，我们一般配置自己的settings.xml，不要轻易修改全局的settings.xml。同样的道理，我们一般不需要修改全局settings.xml中的profile。

不同类型的profile中可以声明的pom元素是不一样的，pom.xml中的profile能够随同pom.xml一起提交到代码仓库中，被Maven安装到本地仓库里面，并且能被部署到远程Maven仓库中。也就是说，可以保证profile伴随特定的pom.xml一起存在，所有它可以修改或者添加很多pom元素，如：

<project>

<repositories></repositories>

<pluginRepositories></pluginRepositories>

<dependencies></dependencies>

<dependencyManagement></dependencyManagement>

<modules></modules>

<properties></properties>

<reporting></reporting>

<build>

<plugins></plugins>

<defaultGoal></defaultGoal>

<resources></resources>

<testResources></testResources>

<finalName></finalName>

</build>

</project>

如上所示，在pom中的profile元素比较多，可以添加和修改仓库、插件参考以及部署仓库的地址；可以IU个、添加项目依赖；可以修改聚合项目的聚合配置；可以自由添加或修改Maven属性；可以添加或修改项目报告配置；可以添加或修改插件配置、项目资源目录、测试资源目录配置和项目构建的默认名称等等。

除了pom中的profile外，其它外部的profile，可以配置的元素相对就少些，因为那些外部profile无非保证同项目中的pom.xml一起发布。如果在外部profile中配置了项目依赖，开发用户可以在本地可以很好的编译构建，但是因为依赖配置没有随同pom.xml一起发布部署到仓库中，别的用户下载了该项目后，就会因为缺少依赖而构建失败。为了避免这样的不一致情况，所以很多在pom的profile可以出现的元素，不允许在外部profile中出现。

在外部profile可以声明的元素如下：

<project>

<repositories></repositories>

<pluginRepositories></pluginRepositories>

<properties></properties>

</project>

这些外部profile元素，不足以影响项目的正常构建，只会影响项目的仓库和Maven属性。

## 11.5 Web资源过滤

在Web项目中，在src/main/resources目录下的资源文件，会打包到war的WEB-INF/classes目录下，也是java代码编译后的class所在的目录。换句话说，这些资源文件打包后，会放在应用程序的classpath目录中。另外还有src/main/webapp目录下的资源文件，打包后会放在war的根目录下。

同一般的资源文件一样，web资源文件默认是不会被过滤的，即使开启一般资源文件的过滤功能，也不会影响web资源文件。

不过有时候，我们希望在构建项目的时候，为不同客户使用不一样的资源文件，比如客户的logo，就需要为不同客户构建不同的logo图片。这时候，我们就可以在web资源文件中使用Maven属性来达到灵活构建的效果。

比如我们可以用${client.logo}表示客户的图片名称，然后再profile中做如下定义：

<profiles>

<profile>

<id>client-cyedu</id>

<properties>

<client.logo>cyedu.jpg</client.logo>

</properities>

</profile>

<profile>

<id>client-maven</id>

<properties>

<client.logo>maven.jpg</client.logo>

</properities>

</profile>

</profiles>

接下来在pom中配置maven-war-plugin插件，对src/main/webapp资源目录开启过滤功能，如：

<plugin>

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

<artifactId>maven-war-plugin</artifactId>

<version>2.6</version>

<configuration>

<webResources>

<resource>

<filtering>true</filtering>

<directory>src/main/webapp</directory>

<includes>

<include>\*\*/\*.css</include>

<include>\*\*/\*.js</include>

</includes>

</resource>

</webResources>

</configuration>

</plugin>

如上代码，使用directory指定要开启过滤功能的目录为src/main/webapp，使用includes元素，指定要过滤的文件，这里是所有的css和js文件。

完成上面的配置，使用命令mvn clean install -Pclient-cyedu构件表示使用cyedu.jpg图片，使用命令mvn clean install -Pclient-maven构建，表示使用maven.jpg图片。

## 11.6 在profile中激活集成测试

在正规项目开发过程中，开发人员和测试人员需要编写大量的测试案例，对项目代码测试。这些测试案例一般分为单元测试和集成测试。单元测试是对单个类中的一个个方法进行测试，比较细粒具体，运行速度快；集成测试功能模块级测试，比较宏观，涉及的内容比较多，运行速度比较慢。在项目后期的集成和系统测试阶段，频繁的执行集成测试案例是需要的，但是在平常开发过程中，还自动执行集成测试案例的话，就有点不现实了。

这样我们就可以利用TestNG中的组的概念，为所有的单元测试标记成单元测试组，系统测试标记成系统测试组，然后在Maven指定执行哪些组，达到灵活执行必要的测试案例效果，提高测试效率。具体样例代码如下：

1. 使用Test标签对测试案例分组：

@Test(groups={“unit”})// 指定unit组的测试方法

@Test(groups={“integration”})// 指定integration组的测试方法

1. Maven配置

<project>

<build>

<plugins>

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>2.16</version>

<configuration>

<groups>unit</groups>

</configuration>

</plugin>

</plugins>

<profiles>

<profile>

<id>full</id>

<build>

<plugins>

<plugin>

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

<artifactId>maven-surfire-plugin</artifactId>

<version>2.16</version>

<configuration>

<groups>unit, integration</groups>

</configuration>

</plugin>

</plugins>

</build>

</profile>

</profiles>

</build>

</project>

如上代码，在maven-surefire-plugin中，配置了默认执行unit测试组，如果要执行全面测试的话，我们可以激活full profile，执行unit和integration两个测试组。

## 11.7 小结

项目构建过程中，我们常常要处理不同的平台环境差异，可能是操作系统不一样可能是开发平台不一样、可能是测试平台不一样，也可能不同客户又不一样。

为了灵活的处理这些差异，Maven提供了属性、资源过滤或profile三种不同的处理方式。我们可以在pom和资源文件中使用Maven的属性，描述那些可能变化的内容（等同于在语言中用变量描述不同的值一样），通过不同的profile的属性值和资源过滤等特征，为不同环境执行不同的构建。

# 第十二章 自定义Maven插件

通过前面对Maven的研究和实战，我们可以感觉到Maven是一个系统管理框架或体系，专注管理构建的生命周期和各个阶段，真正做事的，是绑定到各个阶段的Maven插件。每个插件具有一个或一个以上的目标，我们可以将这些插件的目标绑定到Maven生命周期的各个阶段中，或直接从命令行运行这些插件的目标。

我们可以从Apache和其它的参考Maven中心仓库获取，当然，这些插件基本上能满足我们大部分需求，但是在特殊情况下，有些极个别的功能，可能在中心仓库中找不到，这时候怎么办呢？一个是耐心的等待别人做好了插件分享；一个是我们自己动手，完成一个实现该功能的插件。

等待是需要耐心的，如果当前功能要求的很急，没有时间让我们验证自己的耐心，那咱们就只能亲自动手写插件了。

其实编写插件也并不是很难很复杂的事情，接下来我们开始研究一下，怎么样编写看似高深的、能绑定到Maven生命周期的阶段中自动被调用执行的Maven插件。

## 12.1 自定义Maven插件简介

为了方便大家对编写Maven插件的方向和过程有个总体的了解，我们先介绍一下编写Maven插件的基本步骤：

1. 创建Maven项目

插件的功能肯定需要编写java类的，所以插件本身就是一个Maven项目。当然，相对我们以前研究的Maven项目，插件项目有它特殊点：packaging必须是maven-plugin类型，我们可以通过maven-archetype-plugin快速创建一个Maven插件项目。

1. 编写插件目标

每个插件都至少包含一个目标，每个目标对应一个独立的Java类，我们把这种类叫Mojo类（对象）。Mojo类必须继承AbstractMojo父类

1. 设置目标的配置点

大部分Maven插件和它的目标都是可以配置的，根据需要，我们可以在编写Mojo的时候，给它设置好可以配置的参数。

1. 编写逻辑代码，实现目标功能

用java代码，实现插件的功能

1. 处理错误和日志

当Mojo运行的时候发生异常是，我们需要根据情况，控制Maven的运行状况，并且用代码实现必要的日志输出，为用户提供必要的提示信息。

1. 测试插件

编写测试案例，绑定（或命令行）执行插件

## 12.2 自定义Maven插件案例

为了快速的体会自定义Maven插件的过程，接下来我们实现一个简单的Hello Maven插件，功能很简单：输出Hello World插件。具体步骤和操作如下：

1. 创建Maven插件项目
2. 创建Maven新项目，选择maven-archetype-plugin项目向导。如图12-1：

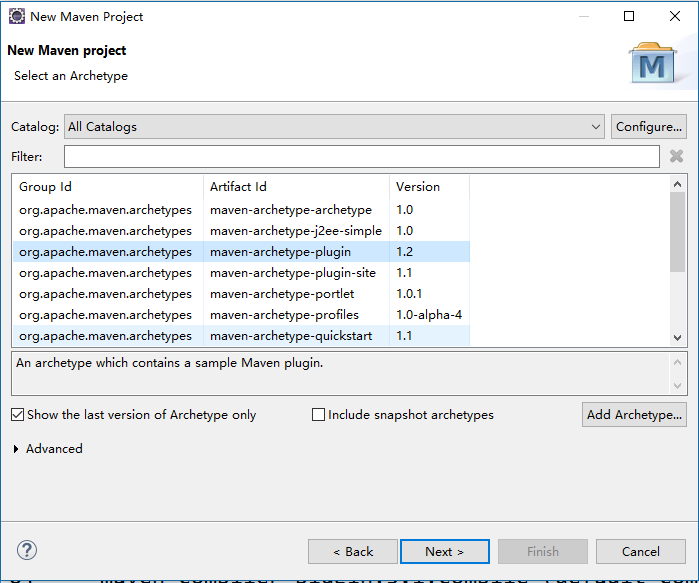


图12-1 选择archetype

1. 点击上图 Next 按钮，进入设置Maven插件参数界面，输入要创建的插件的groupId、artifactId和版本，还有确实包名。如图12-2：

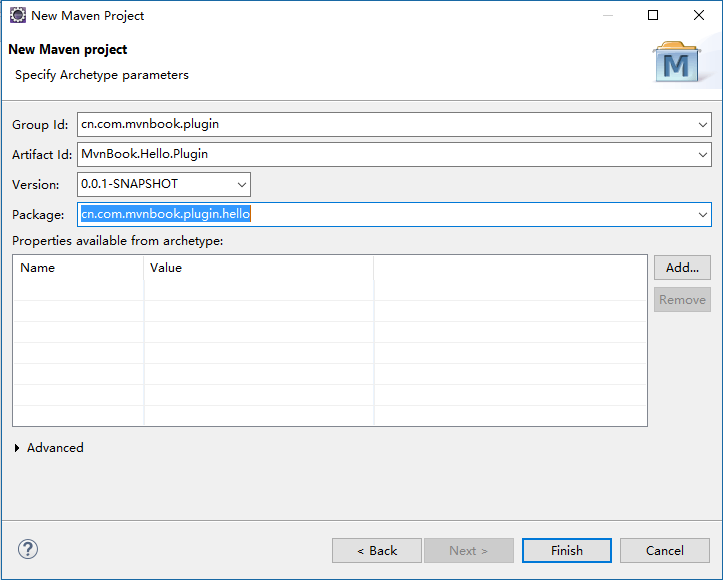


图12-2 输入坐标

1. 点击上图的 Finish 按钮，archetype插件会自动帮我们创建好一个Maven插件项目。因为我们现在用的Maven是3.x版本的，所有有必要调整一下我们的Maven插件项目必须依赖的maven-plugin-api的版本：从2.x改成3.x，我们这里是用的是3.3.9版本。样例项目中的pom配置如下，注意粗体标识部分。

<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/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>cn.com.mvnbook.plugin</groupId>

<artifactId>MvnBook.Hello.Plugin</artifactId>

<version>0.0.1-SNAPSHOT</version>

**<packaging>maven-plugin</packaging>**

<name>MvnBook.Hello.Plugin Maven Plugin</name>

<!-- FIXME change it to the project's website -->

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

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

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

<artifactId>maven-plugin-api</artifactId>

**<version>3.3.9</version>**

</dependency>

<dependency>

<groupId>org.apache.maven.plugin-tools</groupId>

<artifactId>maven-plugin-annotations</artifactId>

<version>3.2</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.codehaus.plexus</groupId>

<artifactId>plexus-utils</artifactId>

<version>3.0.8</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.7</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

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

<version>3.2</version>

<configuration>

<goalPrefix>MvnBook.Hello.Plugin</goalPrefix>

<skipErrorNoDescriptorsFound>true</skipErrorNoDescriptorsFound>

</configuration>

<executions>

<execution>

<id>mojo-descriptor</id>

<goals>

<goal>descriptor</goal>

</goals>

</execution>

<execution>

<id>help-goal</id>

<goals>

<goal>helpmojo</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

<profiles>

<profile>

<id>run-its</id>

<build>

<plugins>

<plugin>

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

<artifactId>maven-invoker-plugin</artifactId>

<version>1.7</version>

<configuration>

<debug>true</debug>

<cloneProjectsTo>${project.build.directory}/it</cloneProjectsTo>

<pomIncludes>

<pomInclude>\*/pom.xml</pomInclude>

</pomIncludes>

<postBuildHookScript>verify</postBuildHookScript>

<localRepositoryPath>${project.build.directory}/local-repo</localRepositoryPath>

<settingsFile>src/it/settings.xml</settingsFile>

<goals>

<goal>clean</goal>

<goal>test-compile</goal>

</goals>

</configuration>

<executions>

<execution>

<id>integration-test</id>

<goals>

<goal>install</goal>

<goal>integration-test</goal>

<goal>verify</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</profile>

</profiles>

</project>

到现在为止，我们基本上创建好了Maven插件项目。

创建插件目标类

在cn.com.mvnbook.plugin.hello包下，创建Java类HelloName，继承AbstractMojo，并且在类上面，使用@goal指定该Mojo的目标名称为name，样例代码如下：

package cn.com.mvnbook.plugin.hello;

import org.apache.maven.plugin.AbstractMojo;

import org.apache.maven.plugin.MojoExecutionException;

import org.apache.maven.plugin.MojoFailureException;

/\*\*

\* @goal name

\* \*/

public class HelloName extends AbstractMojo {

public void execute() throws MojoExecutionException, MojoFailureException {

}

}

设置目标的配置点

在前面的代码基础上，添加一个name属性，用来接收运行插件的时候，传过来的name参数，并且使用@parameter将name属性标注成配置点。代码如下：

package cn.com.mvnbook.plugin.hello;

import org.apache.maven.plugin.AbstractMojo;

import org.apache.maven.plugin.MojoExecutionException;

import org.apache.maven.plugin.MojoFailureException;

/\*\*

\* @goal name

\* \*/

public class HelloName extends AbstractMojo {

/\*\*

\* @parameter expression="${name}"

\* @required

\* @readonly

\* \*/

String name;

public void execute() throws MojoExecutionException, MojoFailureException {

}

}

编写实现代码

在HelloName类中的execute方法中，添加咱们插件要实现的逻辑代码。咱们的HelloName的功能很简单，只是打印一下问候，所以代码如下：

package cn.com.mvnbook.plugin.hello;

import org.apache.maven.plugin.AbstractMojo;

import org.apache.maven.plugin.MojoExecutionException;

import org.apache.maven.plugin.MojoFailureException;

/\*\*

\* @goal name

\* \*/

public class HelloName extends AbstractMojo {

/\*\*

\* @parameter expression="${name}"

\* @required

\* @readonly

\* \*/

String name;

public void execute() throws MojoExecutionException, MojoFailureException {

System.out.println(“Hello ” + name);

}

}

处理错误日志

我们插件的目的是打印对name的问候，如name为空，或是空字符串，这样的问候就没什么意义了，我们就需要中止问候，并且提示以日志方式提示用户，样例代码如下：

package cn.com.mvnbook.plugin.hello;

import org.apache.maven.plugin.AbstractMojo;

import org.apache.maven.plugin.MojoExecutionException;

import org.apache.maven.plugin.MojoFailureException;

/\*\*

\* @goal name

\* \*/

public class HelloName extends AbstractMojo {

/\*\*

\* @parameter expression="${name}"

\* @required

\* @readonly

\* \*/

String name;

public void execute() throws MojoExecutionException, MojoFailureException {

If(this.name == null || this.name.trim().equals(“”)){

// 异常处理

throw new MojoExecutionException(“name参数必须设置有效的值”);

}else{

System.out.println(“Hello ”+ name);

// 日志处理

this.getLog().info(“这是运行日志提示: 执行完问候” );

}

}

}

到现在为止，我们就完成了HelloName Maven插件的编写。要进行插件测试，我们还需要将咱们自定义插件安装好，如果要共享给其它开发人员使用的话，还必须安装到咱们的私服或外面的中央仓库中去。

运行mvn install，就可以完成在本地仓库的安装，如果要安装私服或中央仓库，请仓库前面关于archiva私服相关的介绍。

测试插件

运行插件有两种方式：命令行运行和在别的Maven项目中配置运行。

命令行运行插件

在命令行中输入如下命令：

Mvn cn.com.mvnbook.plugin:MvnBook.Hello.Plugin:name -Dname=zhangsan

命令执行效果如下：

[INFO] ------------------------------------------------------------------------

[INFO]

[INFO] --- MvnBook.Hello.Plugin:0.0.1-SNAPSHOT:name (default-cli) @ MvnBookDemoPlugin ---

Hello lisi

[INFO] 这是运行日志提示：执行完问候

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 1.139 s

[INFO] Finished at: 2017-01-08T11:55:48+08:00

[INFO] Final Memory: 8M/108M

[INFO] ------------------------------------------------------------------------

项目配置运行插件

创建一个简单的Maven项目，在它的pom中添加我们的HelloName插件如下：

<build>

<plugins>

<plugin>

<groupId>cn.com.mvnbook.plugin</groupId>

<artifactId>MvnBook.Hello.Plugin</artifactId>

<version>0.0.1-SNAPSHOT</version>

<executions>

<execution>

<goals>

<goal>name</goal>

</goals>

<phase>test</phase>

</execution>

</executions>

<configuration>

<name>lisi</name>

</configuration>

</plugin>

</plugins>

</build>

执行mvn test，我们可以看到如下日志：

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]

[INFO] --- MvnBook.Hello.Plugin:0.0.1-SNAPSHOT:name (default) @ MvnBookDemoPlugin ---

Hello lisi

[INFO] 这是运行日志提示：执行完问候

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

## 12.3 自定义Maven插件的详细说明

前面我们实现了一个简单自定义Maven插件的编写和测试，在代码中，我们用到了@goal和@parameter，分别用来标记插件的目标和参数，接下来，我们详细介绍一下编写Maven插件要用到的标记

**12.3.1 Mojo标记**

自定义Maven插件的常用标记：

1. @goal <name>

这是自定义Maven插件Mojo代码中，唯一必须声明的标记，用来生命该Mojo的目标名称。

1. @phase <name>

声明默认将该目标绑定到default生命周期的某个阶段，这样在配置使用该插件目标的时候，就可以不声明phase。

1. @requiresDependecyResolution <scope>

声明运行该Mojo之前，必须解析哪些范围的依赖。比如maven-surefire-plugin的test目标中，就用@requireDependecyResolution test标注，必须解析完测试访问的所有依赖（compile、test和runtime）。该标记的默认值是runtime

1. @requiresProject <true/false>

声明该目标是不是必须在一个Maven项目中运行，默认是true。

大部分maven插件的目标，都需要依赖一个项目才能执行的，但有些目标是例外的。比如maven-help-plugin插件中的system目标，它是用来显示系统属性和环境变量属性信息的，就不必要强制依赖一个项目才能运行，所以就用@requiresProject false声明。

1. @requiresDirectInvoction <true/false>

声明该目标是否只能使用命令行调用。默认值是false，既可以在命令行中调用，也可以在pom中配置绑定生命周期阶段。如果是true的话，就只支持在命令行中执行，不小心在pom进行配置，绑定生命周期阶段的话，Maven会抛异常。

1. @requiresOnline <true/false>

声明Maven是不是必须是在线状态，默认是false

1. @requiresReport <true/false>

声明是否要求项目报告已经生成，默认是false

1. @aggregator

在多模块的Maven项目中，声明该目标是否只在顶层模块构建的时候执行。如maven-javadoc-plugin的aggregator-jar，就使用@aggregator标记了，它只会在顶层项目生成一个已经聚合的JavaDoc文档

1. @execute goal=”<goal>”

声明执行该目标之前，先执行指定的目标。

如果该目标是自己插件的另外一个目标，直接goal=”目标名”就行

如果该目标是另外一个插件的目标，就需要写成 goal=”目标前缀:目标名”

1. @execute phase=”<phase>”

声明在执行该目标之前，Maven先运行到当前生命周期的指定阶段

1. @execute lifecycle=”<lifecycle>” phase=”<phase>”

声明执行目标前，Maven运行到指定生命周期的指定阶段

**12.3.2 Mojo参数**

在Mojo中，一般都会有一个或多个Mojo参数需要配置，我们会用@parameter标记。

Maven执行boolean、int、float、String、Date、File和URL等单值累出的参数，多值类型参数包括数组、Collection、Map、Properties等。下面我们一个个介绍它们的配置。

1. Boolean（boolean和Boolean）

标记形式：

/\*\*

\*@parameter

\*/

private boolean testBoolean;

Pom中的配置

<testBoolean>value</testBoolean>

1. Int(Integer,long,Long,short,Short,byte,Byte)

标记形式：

/\*\*

\*@parameter

\*/

private int testInt;

Pom中的配置

<testInt>value</testInt>

1. Float (Float,double,Double)

标记形式：

/\*\*

\*@parameter

\*/

private double testDouble;

Pom中的配置

<testDouble>value</testDouble>

1. String (StringBuffer,char,Character)

标记形式：

/\*\*

\*@parameter

\*/

private String testString;

Pom中的配置

<testString>value</testString>

1. Date (yyyy-MM-dd hh:mm:ss.S a or yyyy-MM-dd hh:mm:ssa)

标记形式：

/\*\*

\*@parameter

\*/

private Date testDate;

Pom中的配置

<testDate>value</testDate>

1. File

标记形式：

/\*\*

\*@parameter

\*/

private File testFile;

Pom中的配置

<testFile>value</testFile>

1. URL

标记形式：

/\*\*

\*@parameter

\*/

private URL testUrl;

Pom中的配置

<testUrl>value</testUrl>

1. 数组

标记形式：

/\*\*

\*@parameter

\*/

Private String[] test

pom中的配置

<includes>

<include>abc</include>

<include>efg</include>

</includes>

Collection

标记形式：

/\*\*

\*@parameter

\*/

Private String[] tests

pom中的配置

<tests>

<test>abc</test>

<test>efg</test>

</tests>

Map

标记形式：

/\*\*

\*@parameter

\*/

Private Map test

pom中的配置

<test>

<key1>value1</key1>

<key2>value2</key2>

</test>

Properties

标记形式：

/\*\*

\*@parameter

\*/

Private Properties tests;

Pom中的配置：

<tests>

<property>

<name>name1</name>

<value>value1</value>

</property>

<property>

<name>name2</name>

<value>value2</value>

</property>

</tests>

除了直接使用@parameter标记配置的Mojo属性外，我们还可以使用@parameter的其它属性，进行进一步的详细声明：

@parameter alias=”<aliasName>”

通过alias指定Mojo参数的别名。

/\*\*

\*@parameter alias=”username”

\*/

Private String testUserName;

Pom中的配置：

<username>value</username>

@parameter expression=”${attributeName}”

读取属性的值给参数赋值。如：

/\*\*

\*@parameter expression=”${userName}”

\*/

Private String testUserName

我们可以通过在pom中配置userName参数赋值，也可以在命令行中使用-DuserName=value进行赋值。

@parameter default-value=”value/${attributeName}”

支持对Mojo属性赋予初始值

除了@parameter标记外，我们还可以使用@required和@readonly配合标记属性

@required

表示Mojo参数是必须的，如果使用了该标记，则必须配置Mojo值，否则会报错

@readonly

表示Mojo参数是只读的，用户不能通过配置修改。

## 12.4 自定义Maven插件中的错误处理和日志

在Mojo的执行方法后面，支持MojoExecutionExcetion和MojoFailureException两种异常。

如果运行抛出MojoFailureException异常，会显示“Build Failure”错误信息，表示可以预期的错误。

如果运行抛出MojoExecutationException异常，会显示“Build Error”错误信息，表示为预期的错误。

除了前面的两个异常定义，Mojo中还提供了相关方法，可以输出不同等级的日志，用户可以通过这些日志输出，更详细的把握执行状况。

我们可以调用父类AbstractMojo的getLog()方法，获取Log对象，输入四个等级的日志信息。从低到高分别是：

1. Debug：调试基本的日志
2. Info：消息基本的日志
3. Warn：警告级别的日志
4. Error：错误级别的日志

为了输出上面各个级别的信息，分别提供了三种方法

Void debug(CharSequence ch);

Void debug(CharSequence ch,Throwable error);

Void debug(Throwable error);

# 第十三章 Archetype扩展

通过前面章节的研究，我们了解了可以使用Maven插件，将对Maven项目的构建逻辑绑定到Maven构建生命周期的对应阶段，从而完成固定的构建动作。比如编译源代码、编译测试代码、运行测试代码，甚至生成站点文档等等。一般情况下，我们通过指定对应插件的坐标，就可以从仓库中找到、下载插件使用就行了，如果自己想在构建的过程中，完成自己个性化的功能，我们也可以根据Maven插件的规范，自己定义实现插件使用，还可以将自己写好的插件，发布到仓库中，供其他开发人员使用。

Maven除了有完成构建的插件外，还有一种插件：Archetype插件。它的作用是生成Maven项目骨架（项目的目录结构和pom.xml）。我们只要给对应的Archetype插件，提供简单的基本信息，比如groupId、artifactId和version，它就可以生成项目的基本目录结构和pom文件。比如maven-archetype-quickstart插件，就是一个快速创建简单Maven工程的Archetype插件。

有很多开源项目，如Appfuse和Apache Wicket，都提供了Archetype插件，方便开发人员快速的创建项目。

当然，如果我们的公司或项目组的Maven项目，有自己个性化的配置和目录结构的话，我们也可以自己定义一个Archetype插件，发布给所有开发人员使用。

有了公共的Archetype插件后，不仅仅能让开发人员快速简单的创建Maven项目，而且还可以强制所有开发人员遵循统一的项目结构和配置约定，从而统一Maven项目的规范和标准。

## 13.1 Archetype使用概述

Archetype不是Maven的核心特征，作为插件，当我们要使用它的时候，我们需要输入完整的插件坐标，由Maven根据坐标下载对应的插件运行。虽然只是一个插件，因为使用的比较广泛，主要的IDE（Eclipse、NetBeans和IDEA）在集成Maven的时候，都有着重集成了archetype，以方便开发人员快速的创建Maven项目。

对Archetype插件使用是比较简单的。如果使用的是IDE，比如Eclipse，直接基于向导界面，就而已引导选择和使用对应的Archetype插件创建工程。如果使用命令行，也就直接在cmd命令行窗口中，输入 mvn 插件坐标：目标 就行，必要的时候再加参数。比如使用maven-archetype-plugin插件创建Maven项目，只需要输入如下命令就行：

Mvn archetype:generate

其中archetype是maven-archetype-plugin的简称（别名）,generate是maven-archetype-plugin插件创建Maven项目的目标名称。Maven接收到archetype命令后，自动到http://maven.apache.org/archetype/maven-archetype-plugin上下载最新的插件（maven3自动下载最新的稳定版本），然后运行generate目标，创建项目。

在中央仓库中，有很多archetype插件，我们这里对应创建简单maven工程、创建maven web工程和创建maven 框架工程，分别介绍一个代表：

1. maven-archetype-quikstart

Maven-archetype-quickstart应该是最常用的archetype，在我们输入命令行mvn archetype:generate的时候，如果没有指定使用哪个archetype，默认就是使用的quickstart。使用maven-archetype-quickstart生成的项目比较简单：

1. pom.xml，包含有JUnit的依赖声明
2. src/main/java 主代码目录以及一个名为App的java类
3. src/main/test 测试代码目录以及一个名为AppTest的JUnit测试用例类

如果我们需要创建一个全新的Maven项目，我们就可以使用该archetype生成项目架构，再在该架构的基础上进行对应的修改，比如添加依赖，添加resources目录等等，从而省去手动创建pom以及目录结构的麻烦。

1. Maven-archetype-webapp

Maven-archetype-webapp是一个创建Maven War项目的Archetype，它能帮我们创建一个Web应用的基本目录结构和必须的web.xml。使用maven-archetype-webapp生成如下内容：

1. pom.xml packaging的值为war，带有JUnit的依赖声明
2. src/main/webapp 目录
3. src/main/webapp/index.jsp 文件
4. src/main/webapp/WEB-INF/web.xml 文件
5. AppFuse Archetype

AppFuse是一个集成了很多开源工具的项目，它有Matt Raible开发，能帮助我们快速高效的创建Maven项目。目前AppFuse已经集成了最流行的开源工具，比如Spring、Struts、JPA、Hibernate、MyBatis等等。

AppFuse为我们提供了大量的Archetype，方便我们创建各种类型的项目。针对不同的显示层框架，可以分为：

1. appfuse-\*-jsf:基于JSF的Archetype
2. Appfus-\*-spring:基于SpringMVC的Archetype
3. Appfuse-\*-struts:基于Struts2的Archetype
4. Appfuse-\*-tapestry:基于Tapestry的Archetype

每一种archetype，又分3个Archetype，分别为light、basic和modular。Light只包含简单的骨架；basic包含一些用户管理及安全方面的特性；modular会生成多模块的项目，其中core模块包含持久层和业务逻辑层代码，web模块为view层代码。

## 13.2 自定义Archetype

## 13.3 Archetype数据库

在我们用Archetype创建项目的时候，如果没有指定具体的Archetype插件的坐标，maven-archetype-plugin会提供一个Archetype列表给我们选择，在基于m2eclipse插件的Eclipse中，创建项目的时候，也会可视化的显示所有的Archetype插件供我们选择。那这些列表内容怎么类的，来源于哪里呢？这就涉及到了Archetype数据库。

**13.3.1 Archetype数据库简介**

Archetype数据库实际上就是archetype-catalog.xml文件，里面描述了对应archetype插件的坐标。当我们使用maven-archetype-plugin创建Maven项目的时候，如果没有指定具体的插件坐标，maven-archetype-plugin就会读取archetype-catalog.xml中的信息，形成列表给我们选择使用。

如下是一个简单的archetype-catalog.xml样例，包含了webapp-jee5和webapp-javaee6两个Archetype：

<?xml version="1.0" encoding="UTF-8"?>

<archetype-catalog xsi:schemaLocation="http://maven.apache.org/plugins/maven-archetype-plugin/archetype-catalog/1.0.0 http://maven.apache.org/xsd/archetype-catalog-1.0.0.xsd"

xmlns="http://maven.apache.org/plugins/maven-archetype-plugin/archetype-catalog/1.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<archetypes>

<archetype>

<groupId>org.codehaus.mojo.archetypes</groupId>

<artifactId>webapp-jee5</artifactId>

<version>1.3</version>

<repository>https://mvnrepository.com/artifact/org.codehaus.mojo.archetypes/webapp-jee5</repository>

</archetype>

<archetype>

<groupId>org.codehaus.mojo.archetypes</groupId>

<artifactId>webapp-javaee6</artifactId>

<version>1.3</version>

<repository></repository>

</archetype>

</archetypes>

</archetype-catalog>

通过上面的archetype-catalog.xml的了解，Archetype也是通过坐标（groupId、artifactId、version）唯一定位的。另外还有一个repository元素，指定查找（下载）Archetype的位置，默认是maven的缺省中央仓库。

archetype-catalog.xml能够提供Archetype的坐标和url连接等信息，那maven-archetype  
-plugin可以从那里读到archetype-catalog.xml文件呢？

Maven-archetype-plugin能读到的archetype-catalog.xml有如下几种：

Interal

Maven-archetype-plugin内置的Archetype Catalog，大约有五六十个archetype

Local

用户本地的Archetype Catalog，目录是 用户/.m2/archetype-catalog.xml。缺省情况，该文件不存在，我们可以在对应目录下添加

Remote

Maven中央仓库的Archetype Catalog，具体位置是http://repo1.maven.org/maven2  
/archetype-Catalog.xml。

File

指定本地任何位置的archetype -catalog.xml。

http

使用HTTP协议，指定网络中的远程archetype-catalog.xml

如果我们执行maven命令的话，可以在mvn archetype:generate后面，使用archetypeCatalog参数，指定maven-archetype-plugin使用的Catalog。如：

Mvn archetype:generate -DarchetypeCatalog=file://c:/work/temp/archetype-catalog.xml

在执行mvn archetype:generate时，我们可以使用archetypeCatalog参数指定多个xml文件，中间用逗号隔开。同时，maven3的archetypeCatalog的缺省值是remote,local。

**13.3.2 使用本地Archetype数据库**

我们不仅可以基于mavne-archetype-plugin插件，使用在archetype-catalog.xml中描述的插件，创建maven工程外，也可以使用maven-archetype-plugin中的crawl目标，搜索指定仓库中的archetype，生成archetype-catalog.xml。比如：

Mvn archetype:crawl -Drepository=e:/test/repository   
 -Dcatalog=e:/temp/archtype-catalog.xml

Maven会自动遍历repository中的archetype，生成catalog文件到catalog指定的位置。如果没有指定repository参数，archetype插件自动搜索settings.xml中，定义的localRepository目录。

这样，我们就可以将任意地方的archetype（包括网络中的），都集中本地的一个archetype-catalog.xml描述，从而避免在创建工程的和，需要动态的访问多个资源，形成archetype列表。

## 13.4 m2eclipse中配置Archetype Catalogs

基于项目开发的实际需要，我们需要在m2eclipse开发工具中，配置Archetype Catalogs，方便基于界面向导的操作。下面我们介绍一种装好了M2Eclipse的Eclipse中的操作方式：

1. 点击Eclipse中的window菜单，选择Preferences选项，展开左边的Maven，选中Archetypes，如图13-1：

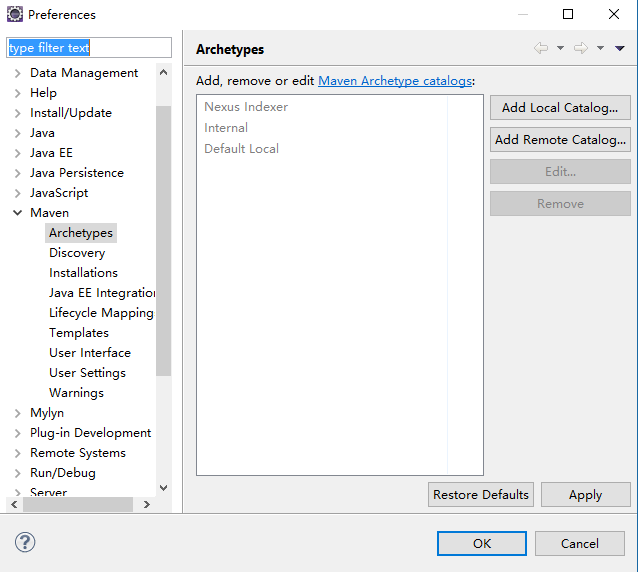


图13-1 Maven的Archetypes列表

1. 点击 Add Local Catalog... 按钮，在如下图13-2界面中，输入界面中，选择要加入的本地archetype-catalog.xml文件和描述，就可以完成本地Catalogs的添加。

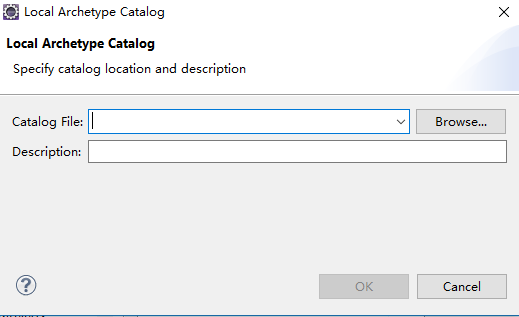


图13-2添加本地Archetype数据库

同样，点击Add Remote Catalog... 按钮，在弹出的对话框中，输入远程archetype-  
catalogs.xml的地址和描述，可以完成一个remote Catalog的添加。

# 附录

## POM元素

1. Project pom.xml根元素
2. Parent 继承元素
3. Modules 聚合模块元素
4. groupId 坐标元素之组id
5. artifactId 坐标元素之构件id
6. Version 坐标元素之版本
7. Packaging 坐标元素之包类型（文件类型）
8. Name 项目名称
9. Description 项目描述
10. Organization 组织名称
11. Licenses ->license 许可证信息
12. mailingLists ->mailingList 邮件列表
13. Developers ->developer 开发者
14. Contributions->contribution 贡献着
15. issueManagement 问题跟踪系统
16. ciManagement 持续集成系统
17. Scm 版本控制系统
18. Prerequistites->maven 要求Maven的最低版本，默认是2.0
19. Build -> sourceDirectory 主源代码目录
20. Build->scriptSourceDirectory 脚本源代码目录
21. Build ->testSourceDirectory 测试源码目录
22. Build -> outputDirectory 主代码输出目录
23. Build -> testOutputDirectory 测试代码输出目录
24. Build -> resources ->resource 主资源目录
25. Build -> testResources ->testResource 测试资源目录
26. Build -> finalName 输出主构件名称
27. Build -> directory 输出目录
28. Build -> filters ->filter 通过properties文件定义资源过滤属性
29. Build ->extensions -> extension 扩展Maven的核心
30. Build -> pluginManagement 插件管理
31. Build ->plugins -> plugin 插件
32. Profiles ->profile POM Profile
33. distributionManagement ->repository 发布版本部署仓库
34. distributionManagement->snapshotRepository 快照版本部署仓库
35. distributionManagement ->site 站点部署
36. Repositories ->repository 仓库
37. pluginRepositories -> pluginRepository 插件仓库
38. Dependencies ->dependency 依赖
39. dependencyManagement 依赖管理
40. Properties Maven属性
41. Reporting->plugins 报告插件

## Setting元素

1. settings settings.xml的根元素
2. localRepository 本地仓库
3. interactiveMode Maven是否与用户交互，默认是true
4. Offline 离线模式，默认是false
5. pluginGroups ->pluginGroup 插件组
6. Servers ->server 下载与部署仓库的认证信息
7. Mirrors ->mirror 仓库镜像
8. Proxies ->proxy 代理
9. Profiles -> profile Setting Profile
10. activeProfiles ->activeProfile 激活Profile