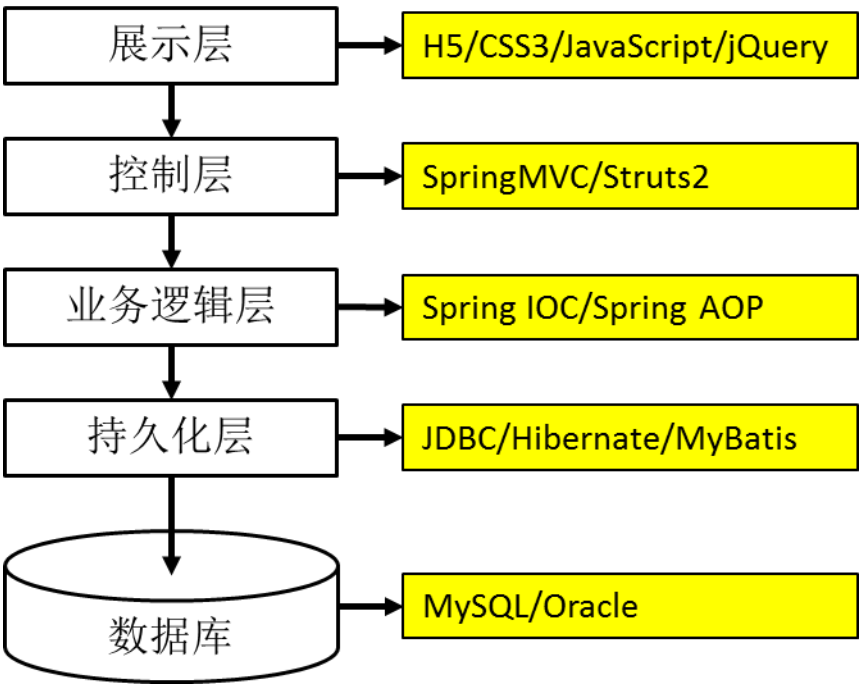
Maven

导言：生产环境下开发不再是一个项目一个工程，而是每一个模块创建一个工程，而多个模块整合在一起就需要使用到像 Maven 这样的构建工具。

1 Why?

1.1 真的需要吗？

Maven 是干什么用的？这是很多同学在刚开始接触 Maven 时最大的问题。之所以会提出这个问题，是因为即使不使用 Maven 我们仍然可以进行 B/S 结构项目的开发。从表述层、业务逻辑层到持久化层再到数据库都有成熟的解决方案——不使用 Maven 我们一样可以开发项目啊？



这里给大家纠正一个误区，Maven 并不是直接用来辅助编码的，它战斗的岗位并不是以上各层。所以我们有必要通过企业开发中的实际需求来看一看哪些方面是我们现有技术的不足。

1.2 究竟为什么？

为什么要使用 Maven？它能帮助我们解决什么问题？

①添加第三方 jar 包

在今天的 JavaEE 开发领域，有大量的第三方框架和工具可以供我们使用。要使用这些 jar 包最简单的方法就是复制粘贴到 WEB-INF/lib 目录下。但是这会导致每次创建一个新的工程就需要将 jar 包重复复制到 lib 目录下，从而造成工作区中存在大量重复的文件，让我们的工程显得很臃肿。

而使用 Maven 后每个 jar 包本身只在本地仓库中保存一份，需要 jar 包的工程只需要以坐标的方式简单的引用一下就可以了。不仅极大的节约了存储空间，让项目更轻巧，更避免了重复文件太多而造成的混乱。

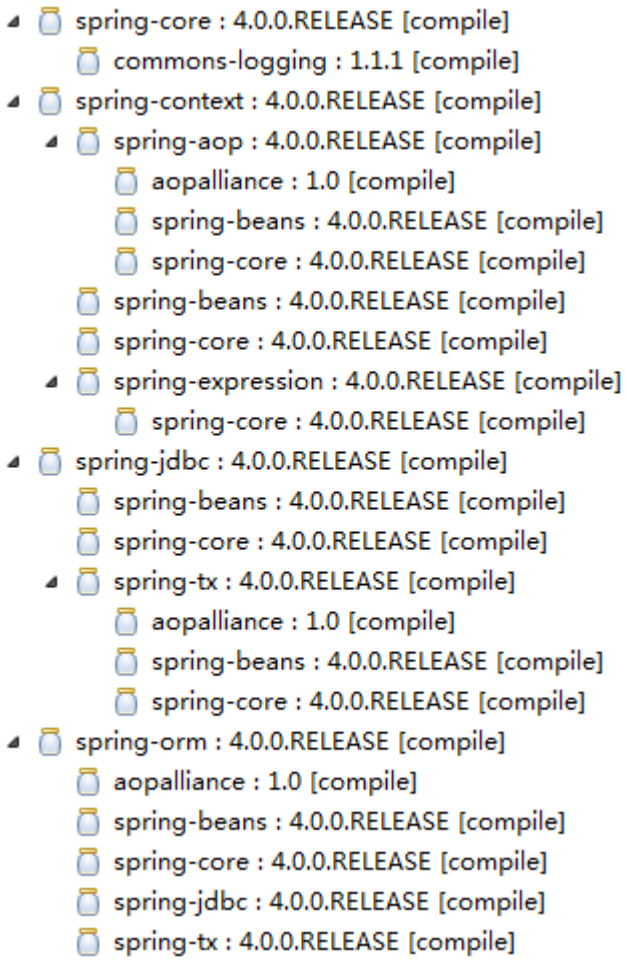
②jar 包之间的依赖关系

jar 包往往不是孤立存在的，很多 jar 包都需要在其他 jar 包的支持下才能够正常工作，我们称之为 jar 包之间的依赖关系。最典型的例子是：commons-fileupload-1.3.jar 依赖于 commons-io-2.0.1.jar，如果没有 IO 包，FileUpload 包就不能正常工作。

那么问题来了，你知道你所使用的所有 jar 包的依赖关系吗？当你拿到一个新的从未使用过的 jar 包，你如何得知他需要哪些 jar 包的支持呢？如果不了解这个情况，导入的 jar 包不够，那么现有的程序将不能正常工作。再进一步，当你的项目中需要用到上百个 jar 包时，你还会人为的，手工的逐一确认它们依赖的其他 jar 包吗？这简直是不可想象的。

而引入 Maven 后，Maven 就可以替我们自动的将当前 jar 包所依赖的其他所有 jar 包全部导入进来，无需人工参与，节约了我们大量的时间和精力。用实际例子来说明就是：通过 Maven 导入 commons-fileupload-1.3.jar 后，commons-io-2.0.1.jar 会被自动导入，程序员不必了解这个依赖关系。

下图是 Spring 所需 jar 包的部分依赖关系



③获取第三方 jar 包

JavaEE 开发中需要使用到的 jar 包种类繁多，几乎每个 jar 包在其本身的官网上的获取方式都不尽相同。为了查找一个 jar 包找遍互联网，身心俱疲，没有经历过的人或许体会不到这种折磨。不仅如此，费劲心血找的 jar 包里有的时候并没有你需要的那个类，又或者又同名的类没有你要的方法——以不规范的方式获取的 jar 包也往往是不规范的。

使用 Maven 我们可以享受到一个完全统一规范的 jar 包管理体系。你只需要在你的项目中以坐标的方式依赖一个 jar 包，Maven 就会自动从中央仓库进行下载，并同时下载这个 jar 包所依赖的其他 jar 包

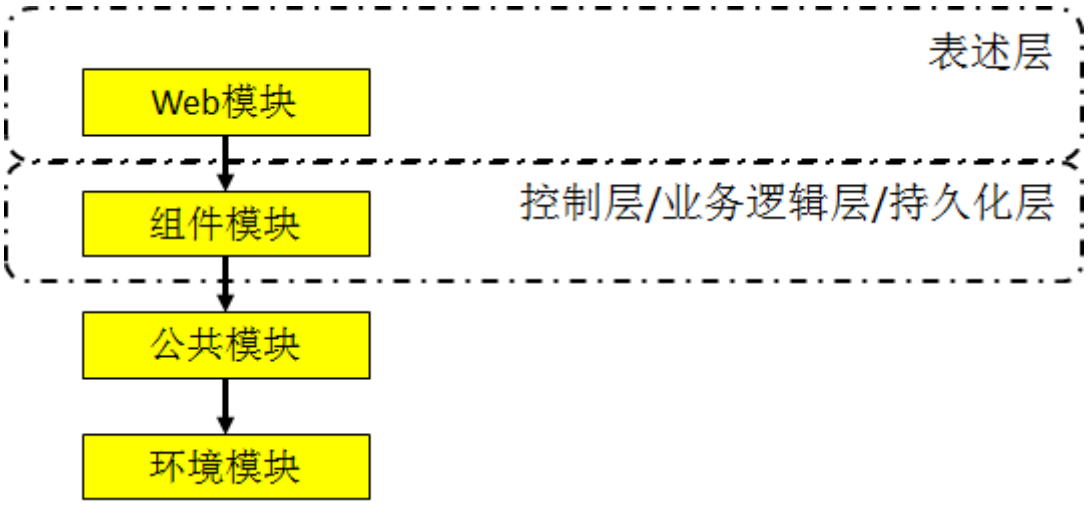
——规范、完整、准确！一次性解决所有问题！

Tips：在这里我们顺便说一下，统一的规范几乎可以说成是程序员的最高信仰。如果没有统一的规范，就意味着每个具体的技术都各自为政，需要以诸多不同的特殊的方式加入到项目中；好不容易加入进来还会和其他技术格格不入，最终受苦的是我们。而任何一个领域的统一规范都能够极大的降低程序员的工作难度，减少工作量。例如：USB 接口可以外接各种设备，如果每个设备都有自己独特的接口，那么不仅制造商需要维护各个接口的设计方案，使用者也需要详细了解每个设备对应的接口，无疑是非常繁琐的。

④将项目拆分成多个工程模块

随着 JavaEE 项目的规模越来越庞大，开发团队的规模也与日俱增。一个项目上千人的团队持续开发很多年对于 JavaEE 项目来说再正常不过。那么我们想象一下：几百上千的人开发的项目是同一个 Web 工程。那么架构师、项目经理该如何划分项目的模块、如何分工呢？这么大的项目已经不可能通过 package 结构来划分模块，必须将项目拆分成多个工程协同开发。多个模块工程中有的是 Java 工程，有的是 Web 工程。

那么工程拆分后又如何进行互相调用和访问呢？这就需要用到 Maven 的依赖管理机制。大家请看我们的 Survey 调查项目拆分的情况：



上层模块依赖下层，所以下层模块中定义的 API 都可以为上层所调用和访问。

2 What?

2.1 Maven 简介

Maven 是 Apache 软件基金会组织维护的一款自动化**构建**工具，专注服务于 Java 平台的**项目构建**和**依赖管理**。Maven 这个单词的本意是：专家，内行。读音是['meɪv(ə)n]或['mevn]。

make->ant->maven->gradlew

构建就是以"java源文件"，"框架配置文件"，"jsp","html","图片"等资源为"原材料"，去"生产"一个可以运行的项目的过程

2.2 什么是构建

构建并不是创建，创建一个工程并不等于构建一个项目。要了解构建的含义我们应该由浅入深的从以下三个层面来看：

①纯 Java 代码

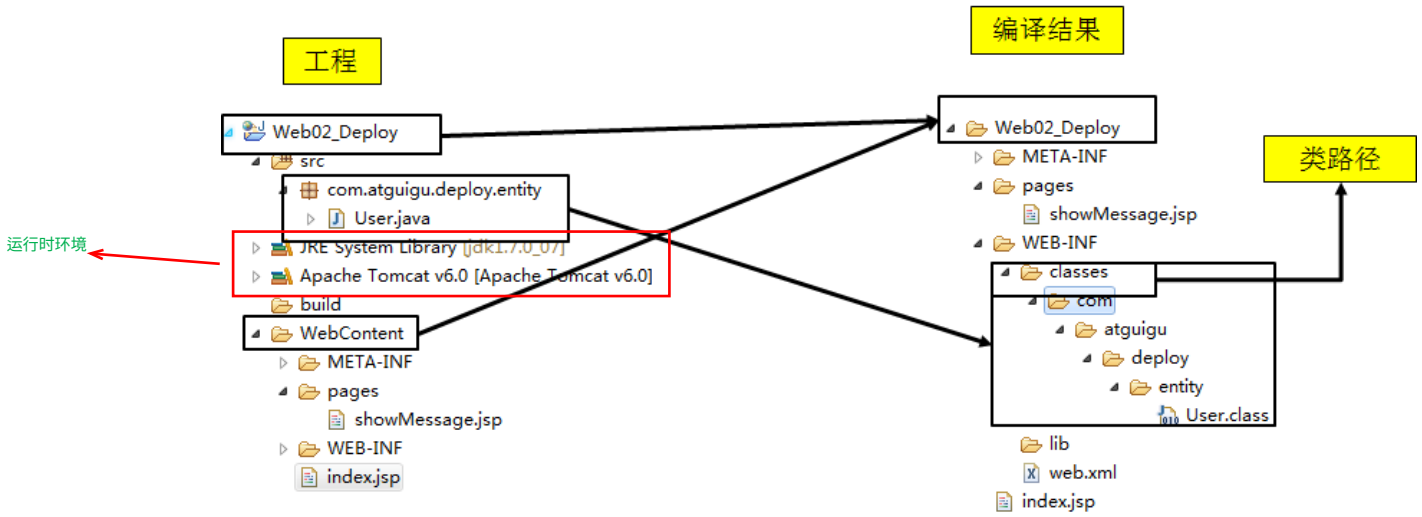
大家都知道，我们 Java 是一门编译型语言，.java 扩展名的源文件需要编译成.class 扩展名的字节码文件才能够执行。所以编写任何 Java 代码想要执行的话就必须经过编译得到对应的.class 文件。

②Web 工程

当我们需要通过浏览器访问 Java 程序时就必须将包含 Java 程序的 Web 工程编译的结果“拿”到服务器上的指定目录下，并启动服务器才行。这个“拿”的过程我们叫部署。

我们可以将未编译的 Web 工程比喻为一只生的鸡，编译好的 Web 工程是一只煮熟的鸡，编译部署的过程就是将鸡炖熟。

Web 工程和其编译结果的目录结构对比见下图：



③实际项目

在实际项目中整合第三方框架，Web 工程中除了 Java 程序和 JSP 页面、图片等静态资源之外，还包括第三方框架的 jar 包以及各种各样的配置文件。所有这些资源都必须按照正确的目录结构部署到服务器上，项目才可以运行。

所以综上所述：构建就是以我们编写的 Java 代码、框架配置文件、国际化等其他资源文件、JSP 页面和图片等静态资源作为“原材料”，去“生产”出一个可以运行的项目的过程。

那么项目构建的全过程中都包含哪些环节呢？

2.3 构建过程的几个主要环节

①**清理**：删除以前的编译结果，为重新编译做好准备。

②**编译**：将 Java 源程序编译为字节码文件。

③**测试**：针对项目中的关键点进行测试，确保项目在迭代开发过程中关键点的正确性。自动测试，自动调用junit程序

④**报告**：在每一次测试后以标准的格式记录和展示测试结果。

⑤**打包**：将一个包含诸多文件的工程封装为一个压缩文件用于安装或部署。Java 工程对应 jar 包，Web 工程对应 war 包。

⑥**安装**：在 Maven 环境下特指将打包的结果——jar 包或 war 包安装到本地仓库中。

⑦**部署**：将打包的结果部署到远程仓库或将 war 包部署到服务器上运行。

2.4 自动化构建

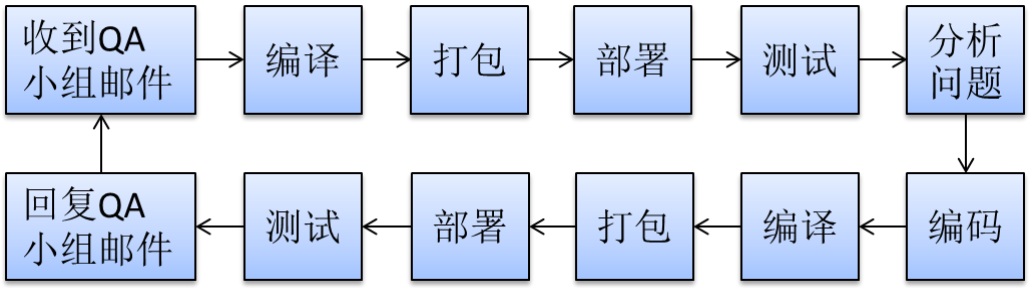
其实上述环节我们在 Eclipse 中都可以找到对应的操作，只是不太标准。那么既然 IDE 已经可以进行构建了我们为什么还要使用 Maven 这样的构建工具呢？我们来看一个小故事：

这是阳光明媚的一天。托马斯向往常一样早早的来到了公司，冲好一杯咖啡，进入了自己的邮箱——很不幸，QA 小组发来了一封邮件，报告了他昨天提交的模块的测试结果——有 BUG。“好吧，反正也不是第一次”，托马斯摇摇头，进入 IDE，运行自己的程序，编译、打包、部署到服务器上，然后按照邮件中的操作路径进行测试。“嗯，没错，这个地方确实有问题”，托马斯说道。于是托马斯开始尝试修复这个 BUG，当他差不多有眉目的时候已经到了午饭时间。

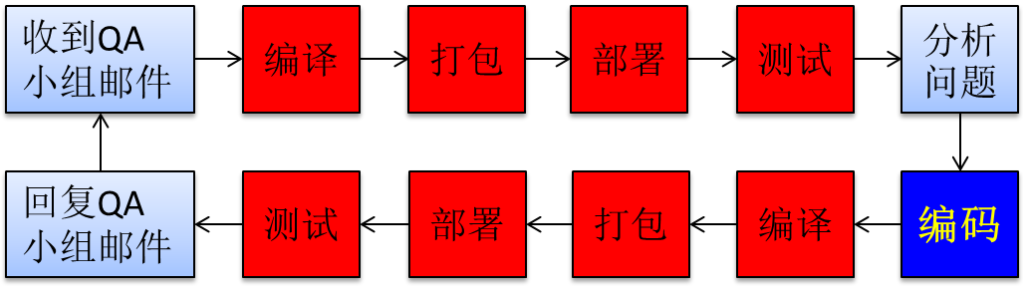
下午继续工作。BUG 很快被修正了，接着托马斯对模块重新进行了编译、打包、部署，测试之后确认没有问题了，回复了 QA 小组的邮件。

一天就这样过去了，明媚的阳光化作了美丽的晚霞，托马斯却觉得生活并不像晚霞那样美好啊。

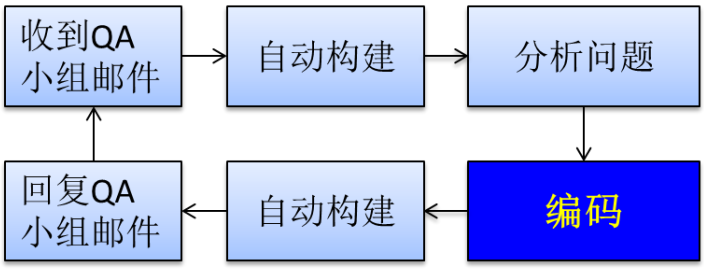
让我们来梳理一下托马斯这一天中的工作内容



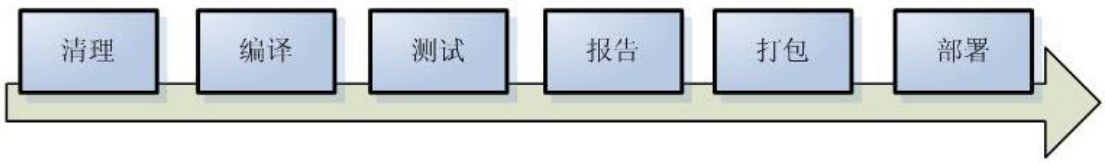
从中我们发现，托马斯的很大一部分时间花在了“编译、打包、部署、测试”这些程式化的工作上面，而真正需要由“人”的智慧实现的分析问题和编码却只占了很少一部分。



能否将这些程式化的工作交给机器自动完成呢？——当然可以！这就是自动化构建。



此时 Maven 的意义就体现出来了，它可以自动的从构建过程的起点一直执行到终点：



2.5 Maven 核心概念

Maven 能够实现自动化构建是和它的内部原理分不开的，这里我们从 Maven 的九个核心概念入手，看看 Maven 是如何实现自动化构建的

①POM
②约定的目录结构
③坐标
④依赖管理
⑤仓库管理
⑥生命周期
⑦插件和目标
⑧继承
⑨聚合

必须进入pom.xml所在的目录

常用Maven命令
注意：执行与构建过程相关的maven命令，必须进入pom.xml所在的目录。
常用命令
1.mvn clean：清理
2.mvn compile：编译主程序
3.mvn test-compile:编译测试程序
4.mvn test：执行测试
5.mvn package :打包
6.mvn install：安装
7.mvn site：生成站点

3 How?

Maven 的核心程序中仅仅定义了抽象的生命周期，而具体的操作则是由 Maven 的插件来完成的。可是 Maven 的插件并不包含在 Maven 的核心程序中，在首次使用时需要联网下载。

下载得到的插件会被保存到本地仓库中。本地仓库默认的位置是：~\.m2\repository。

如果不能联网可以使用我们提供的 RepMaven.zip 解压得到。具体操作参见“Maven 操作指南.txt”。

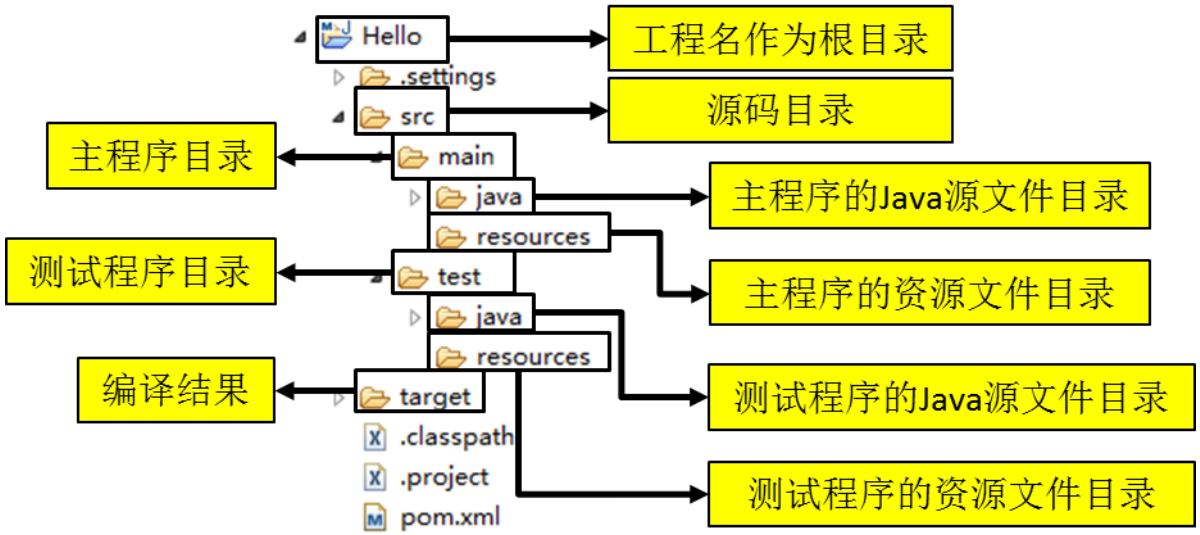
4 约定的目录结构

约定的目录结构对于 Maven 实现自动化构建而言是必不可少的一环，就拿自动编译来说，Maven 必须能找到 Java 源文件，下一步才能编译，而编译之后也必须有一个准确的位置保持编译得到的字节码文件。

我们在开发中如果需要让第三方工具或框架知道我们自己创建的资源在哪，那么基本上就是两种方式：

①通过配置的形式明确告诉它
②基于第三方工具或框架的约定

Maven 对工程目录结构的要求就属于后面的一种。



现在 JavaEE 开发领域普遍认同一个观点：约定>配置>编码。意思就是能用配置解决的问题就不编码，能基于约定的就不进行配置。而 Maven 正是因为指定了特定文件保存的目录才能够对我们的 Java 工程进行自动化构建。

5 POM

Project Object Model：项目对象模型。将 Java 工程的相关信息封装为对象作为便于操作和管理的模型。Maven 工程的核心配置。可以说学习 Maven 就是学习 pom.xml 文件中的配置。

6 坐标

6.1 几何中的坐标

[1]在一个平面中使用 x、y 两个向量可以唯一的确定平面中的一个点。

[2]在空间中使用 x、y、z 三个向量可以唯一的确定空间中的一个点。

6.2 Maven 的坐标

使用如下三个向量在 Maven 的仓库中唯一的确定一个 Maven 工程。

[1]groupid：公司或组织的域名倒序+当前项目名称

[2]artifactId：当前项目的模块名称

[3]version：当前模块的版本

```
<groupId>com.atguigu.maven</groupId>
<artifactId>Hello</artifactId>
<version>0.0.1-SNAPSHOT</version>
```

6.3 如何通过坐标到仓库中查找 jar 包？

[1]将 gav 三个向量连起来

```
com.atguigu.maven+Hello+0.0.1-SNAPSHOT
```

[2]以连起来的字符串作为目录结构到仓库中查找

```
com/atguigu/maven/Hello/0.0.1-SNAPSHOT/Hello-0.0.1-SNAPSHOT.jar
```

※注意：我们自己的 Maven 工程必须执行安装操作才会进入仓库。安装的命令是：mvn install

7 依赖

Maven 中最关键的部分，我们使用 Maven 最主要的就是使用它的依赖管理功能。要理解和掌握 Maven 的依赖管理，我们只需要解决一下几个问题：

①依赖的目的是什么

当 A jar 包用到了 B jar 包中的某些类时，A 就对 B 产生了依赖，这是概念上的描述。那么如何在项目中以依赖的方式引入一个我们需要的 jar 包呢？

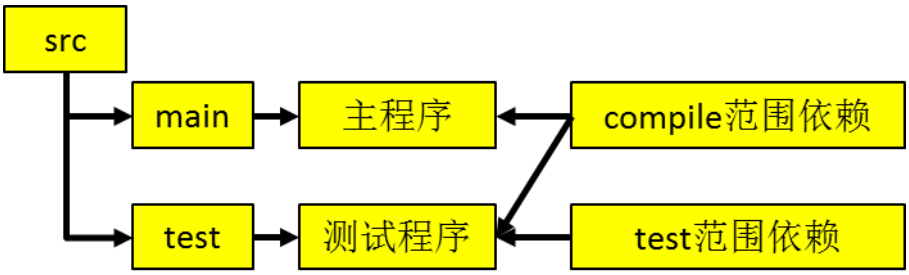
答案非常简单，就是使用 dependency 标签指定被依赖 jar 包的坐标就可以了。

```
<dependency>
    <groupId>com.atguigu.maven</groupId>
    <artifactId>Hello</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <scope>compile</scope>
</dependency>
```

②依赖的范围

大家注意到上面的依赖信息中除了目标 jar 包的坐标还有一个 scope 设置，这是依赖的范围。依赖的范围有几个可选值，我们用得到的是：compile、test、provided 三个。

[1]从项目结构角度理解 compile 和 test 的区别

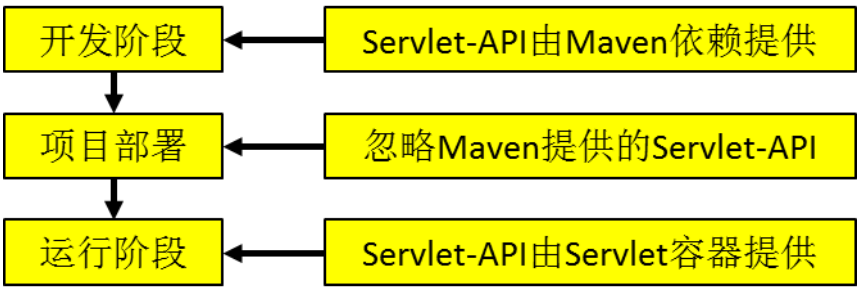


结合具体例子：对于 HelloFriend 来说，Hello 就是服务于主程序的，junit 是服务于测试程序的。

HelloFriend 主程序需要 Hello 是非常明显的，测试程序由于要调用主程序所以也需要 Hello，所以 compile 范围依赖对主程序和测试程序都应该有效。

HelloFriend 的测试程序部分需要 junit 也是非常明显的，而主程序是不需要的，所以 test 范围依赖仅仅对于主程序有效。

[2]从开发和运行这两个不同阶段理解 compile 和 provided 的区别



[3]有效性总结

	compile	test	provided
主程序	√	×	√
测试程序	√	√	√
参与部署	√	×	×

③依赖的传递性 可以传递的依赖不必在每个模块工程中都重复声明，在"最下面"的工程中依赖一次即可。

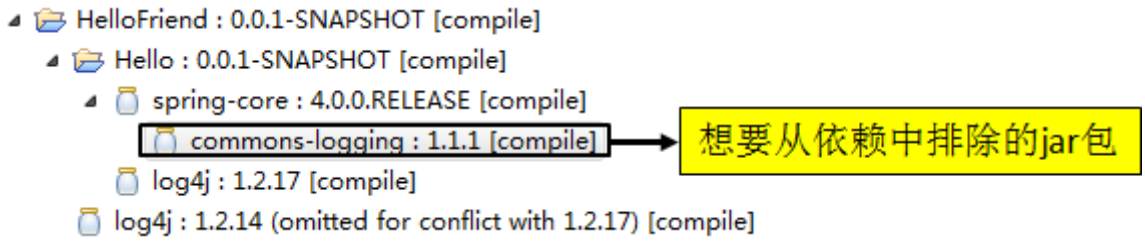
A 依赖 B，B 依赖 C，A 能否使用 C 呢？那要看 B 依赖 C 的范围是不是 compile，如果是则可用，否则不可用。 非complie的范围依赖不能传递

Maven 工程			依赖范围	对 A 的可见性
A	B	C	compile	√
		D	test	×
		E	provided	×

④依赖的排除 什么时候要使用依赖的排除

如果我们在当前工程中引入了一个依赖是 A，而 A 又依赖了 B，那么 Maven 会自动将 A 依赖的 B 引入当前工程，但是个别情况下 B 有可能是一个不稳定版，或对当前工程有不良影响。这时我们可以在引入 A 的时候将 B 排除。

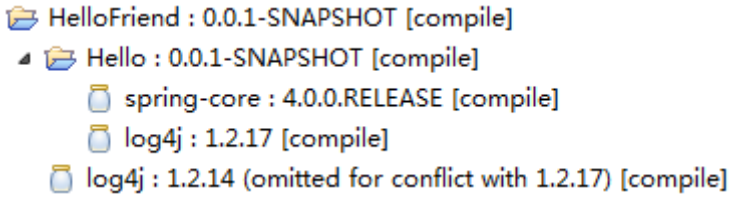
[1]情景举例



[2]配置方式

```
<dependency>
    <groupId>com.atguigu.maven</groupId>
    <artifactId>HelloFriend</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <type>jar</type>
    <scope>compile</scope>
    <exclusions>
        <exclusion>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

[3]排除后的效果



⑤统一管理所依赖 jar 包的版本

对同一个框架的一组 jar 包最好使用相同的版本。为了方便升级框架，可以将 jar 包的版本信息统一提取出来

[1]统一声明版本号

```
<properties>
    <atguigu.spring.version>4.1.1.RELEASE</atguigu.spring.version>
</properties>
```

其中 atguigu.spring.version 部分是自定义标签。

[2]引用前面声明的版本号

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

```
    </dependency>
……
</dependencies>
```

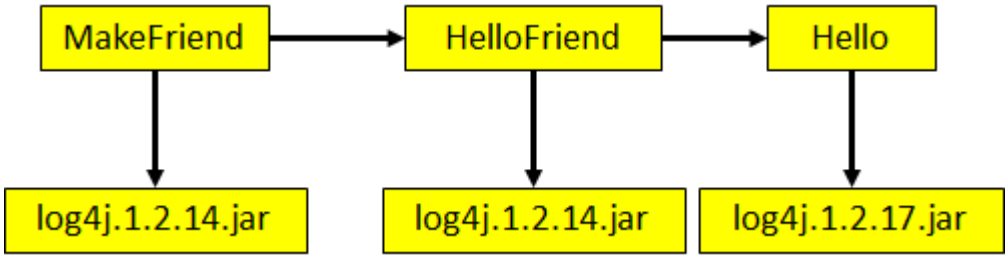
[3]其他用法

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

⑥依赖的原则：解决 jar 包冲突

[1]路径最短者优先

路径按箭头数 一个箭头代表一个路径



[2]路径相同时先声明者优先

这里“声明”的先后顺序指的是 dependency 标签配置的先后顺序。

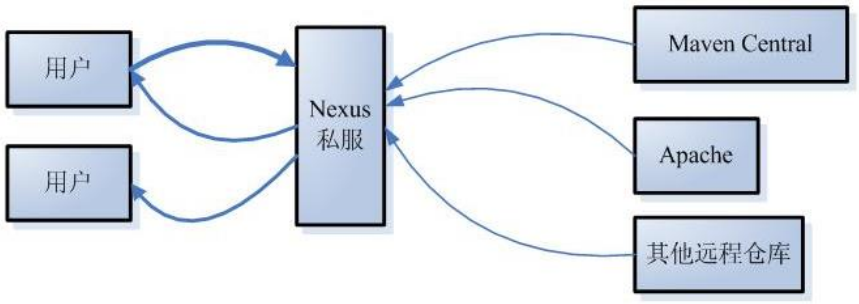
8 仓库

8.1 分类

[1]本地仓库：为当前本机电脑上的所有 Maven 工程服务。

[2]远程仓库

(1)私服：架设在当前局域网环境下，为当前局域网范围内的所有 Maven 工程服务。



(2)中央仓库：架设在 Internet 上，为全世界所有 Maven 工程服务。

(3)中央仓库的镜像：架设在各个大洲，为中央仓库分担流量。减轻中央仓库的压力，同时更快的

响应用户请求。

8.2 仓库中的文件

[1]Maven 的插件

[2]我们自己开发的项目的模块

[3]第三方框架或工具的 jar 包

※不管是什么样的 jar 包，在仓库中都是按照坐标生成目录结构，所以可以通过统一的方式查询或依赖。

9 生命周期

9.1 什么是 Maven 的生命周期？

●Maven 生命周期定义了各个构建环节的执行顺序，有了这个清单，Maven 就可以自动化的执行构建命令了。

●Maven 有三套相互独立的生命周期，分别是：

①Clean Lifecycle 在进行真正的构建之前进行一些清理工作。

②Default Lifecycle 构建的核心部分，编译，测试，打包，安装，部署等等。

③Site Lifecycle 生成项目报告，站点，发布站点。

它们是相互独立的，你可以仅仅调用 clean 来清理工作目录，仅仅调用 site 来生成站点。当然你也可以直接运行 mvn clean install site 运行所有这三套生命周期。

每套生命周期都由一组阶段(Phase)组成，我们平时在命令行输入的命令总会对应于一个特定的阶段。比如，运行 mvn clean，这个 clean 是 Clean 生命周期的一个阶段。有 Clean 生命周期，也有 clean 阶段。

9.2 Clean 生命周期

Clean 生命周期一共包含了三个阶段：

①pre-clean 执行一些需要在 clean 之前完成的工作

②clean 移除所有上一次构建生成的文件

③post-clean 执行一些需要在 clean 之后立刻完成的工作

9.3 Site 生命周期

①pre-site 执行一些需要在生成站点文档之前完成的工作

②site 生成项目的站点文档

③post-site 执行一些需要在生成站点文档之后完成的工作，并且为部署做准备

④site-deploy 将生成的站点文档部署到特定的服务器上

这里经常用到的是 site 阶段和 site-deploy 阶段，用以生成和发布 Maven 站点，这可是 Maven 相当强大的功能，Manager 比较喜欢，文档及统计数据自动生成，很好看。

9.4 Default 生命周期

Default 生命周期是 Maven 生命周期中最重要的一个，绝大部分工作都发生在这个生命周期中。这里，只解释一些比较重要和常用的阶段：

validate

generate-sources

process-sources

generate-resources
process-resources 复制并处理资源文件，至目标目录，准备打包。
compile 编译项目的源代码。
process-classes
generate-test-sources
process-test-sources
generate-test-resources
process-test-resources 复制并处理资源文件，至目标测试目录。
test-compile 编译测试源代码。
process-test-classes
test 使用合适的单元测试框架运行测试。这些测试代码不会被打包或部署。
prepare-package
package 接受编译好的代码，打包成可发布的格式，如 JAR。
pre-integration-test
integration-test
post-integration-test
verify
install 将包安装至本地仓库，以让其它项目依赖。
deploy 将最终的包复制到远程的仓库，以让其它开发人员与项目共享或部署到服务器上运行。

9.5 生命周期与自动化构建

运行任何一个阶段的时候，它前面的所有阶段都会被运行，例如我们运行 mvn install 的时候，代码会被编译，测试，打包。这就是 Maven 为什么能够自动执行构建过程的各个环节的原因。此外，Maven 的插件机制是完全依赖 Maven 的生命周期的，因此理解生命周期至关重要。

10 插件和目标

●Maven 的核心仅仅定义了抽象的生命周期，具体的任务都是交由插件完成的。
●每个插件都能实现多个功能，每个功能就是一个插件目标。
●Maven 的生命周期与插件目标相互绑定，以完成某个具体的构建任务。
例如：compile 就是插件 maven-compiler-plugin 的一个目标；pre-clean 是插件 maven-clean-plugin 的一个目标。

11 继承

11.1 为什么需要继承机制？

由于非 compile 范围的依赖信息是不能在“依赖链”中传递的，所以有需要的工程只能单独配置。例如：

Hello	<dependency> <groupId>junit</groupId> <artifactId>junit</artifactId> <version>4.0</version> <scope>test</scope> </dependency> 由于test范围的依赖不能传递，所以必然会分散在各个模块工程中，很容易造成版本不一致
HelloFriend	<dependency> <groupId>junit</groupId> <artifactId>junit</artifactId> <version>4.0</version>

12

操作步骤
1.创建一个Maven工程作为父工程。注意：打包的方式是pom
2.在子工程中声明对父工程的引用
3.将子工程的坐标中与父工程坐标中重复的内容删除
4.在夫工程中统一junit的依赖
5.在子工程中删除junit依赖的部分版本号部分

	<scope>test</scope> </dependency>
MakeFriend	<dependency> <groupId>junit</groupId> <artifactId>junit</artifactId> <version>4.0</version> <scope>test</scope> </dependency>

此时如果项目需要将各个模块的 junit 版本统一为 4.9，那么到各个工程中手动修改无疑是非常不可取的。使用继承机制就可以将这样的依赖信息统一提取到父工程模块中进行统一管理。

需求：统一管理各个模块工程中对junit依赖的版本
解决思路：将junit依赖版本统一提取到"父"工程中，在子工程中声明依赖时不指定版本

11.2 创建父工程　只在pom.xml文件中做版本控制，其他没啥操作　第一步

创建父工程和创建一般的 Java 工程操作一致，唯一需要注意的是：打包方式处要设置为 pom。

第二步

11.3 在子工程中引用父工程

```
<parent>
    <!-- 父工程坐标 -->
    <groupId>...</groupId>
    <artifactId>...</artifactId>
    <version>...</version>
    <relativePath>从当前目录到父项目的 pom.xml 文件的相对路径</relativePath>
</parent>
```

```
<parent>
    <groupId>com.atguigu.maven</groupId>
    <artifactId>Parent</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <!-- 指定从当前子工程的pom.xml文件出发，查找父工程的pom.xml的路径 -->
    <relativePath>../Parent/pom.xml</relativePath>
</parent>
```

此时如果子工程的 groupId 和 version 如果和父工程重复则可以删除。

11.4 在父工程中管理依赖

将 Parent 项目中的 dependencies 标签，用 dependencyManagement 标签括起来

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.9</version>
            <scope>test</scope>
```

```
        </dependency>
    </dependencies>
</dependencyManagement>
```

在子项目中重新指定需要的依赖，删除范围和版本号

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
    </dependency>
</dependencies>
```

12 聚合

聚合就是一键安装 mvn install

12.1 为什么要使用聚合？

将多个工程拆分为模块后，需要手动逐个安装到仓库后依赖才能够生效。修改源码后也需要逐个手动进行 clean 操作。而使用了聚合之后就可以批量进行 Maven 工程的安装、清理工作。

12.2 如何配置聚合？

聚合工程随便是那个，都行 随便放到那个工程项目中都行

在总的聚合工程中使用 modules/module 标签组合，指定模块工程的相对路径即可

```
<modules>
    <module>../Hello</module>
    <module>../HelloFriend</module>
    <module>../MakeFriends</module>
</modules>
```

顺序无所谓，会自动调整

13 Maven 酷站

我们可以到 http://mvnrepository.com/搜索需要的 jar 包的依赖信息。