# 入门

## XML简介

### 示例

1、随便摘取一段Spring配置文件作为示例

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

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

xsi:schemaLocation="

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

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

### xmlns

1、xmlns其实是XML Namespace的缩写

### 如何使用xmlns

1、使用语法： xmlns:namespace-prefix="namespaceURI"

1. namespace-prefix为**自定义**前缀，只要在这个XML文档中保证前缀不重复即可
2. namespaceURI是这个前缀对应的XML Namespace的定义

### xmlns和xmlns:xsi有什么不同

1、xmlns表示默认的Namespace，例如

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

* 对于默认的Namespace中的元素，可以不使用前缀

2、xmlns:xsi表示使用xsi作为前缀的Namespace，当然前缀xsi需要在文档中声明

### xsi:schemaLocation有何作用？

1、xsi:schemaLocation属性其实是Namespace为  
http://www.w3.org/2001/XMLSchema-instance里的schemaLocation属性，正是因为我们一开始声明了

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

2、它定义了XML Namespace和对应的XSD(Xml Schema Definition)文档的位置的关系。它的值由一个或多个URI引用对组成，两个URI之间以空白符分隔(空格和换行均可)。第一个URI是定义的XML Namespace的值，第二个URI给出Schema文档的位置，Schema处理器将从这个位置读取Schema文档，该文档的targetNamespace必须与第一个URI相匹配

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

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

## Maven示例

1、示例如下

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

<project xmlns="http://maven.apache.org/POM/4.0.0"

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

xsi:schemaLocation="

http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.sunlands.platform</groupId>

<artifactId>hello-world</artifactId>

<version>1.0-SNAPSHOT</version>

<name>Maven Hello World Project</name>

</project>

2、groupId、artifactId和version是定位一个项目的基本坐标，任何jar、pom或者war都是以给予这些基本的坐标进行区分的

1. groupId定义了项目属于哪个组，组往往和项目所在的组织或公司有关
2. artifactId定义了当前Maven项目在组中的唯一Id

# 坐标和依赖

## 坐标详解

1、五个元素：groupId、artifactId、version、packaging、classifier

2、其中groupId、artifactId、version是必须得，packaging是可选的，classifier是不能直接定义的

### groupId

1、定义当前Maven项目隶属的实际项目

2、Maven项目和实际项目不一定是一对一的关系，例如SpringFramework这个项目，其对应的Maven项目会有很多，例如spring-core、spring-context等

3、一个实际项目往往会被划分为很多模块

4、groupId不应该对应项目隶属的组织或公司。因为一个组织下会有很多实际项目，如果groupId只定义到组织级别，那么artifactId只能定义Maven项目(模块)，那么实际项目这个层将难以定义

5、groupId的表示方式与Java包名的表示方式类似，通常与域名反向一一对应

### artifactId

1、该元素定义实际项目中的一个Maven项目(模块)

2、推荐做法是使用实际项目名称作为artifactId的前缀

### version

1、该元素定义Maven项目当前所处的版本

2、Maven定义了一套完整的版本规范，以及快照(SNAPSHOT)的概念

### packaging

1、该元素定义Maven项目的打包方式

2、首先，打包方式通常与所生成构件的文件扩展名对应

3、当不定义packaing时，默认值为jar

### classifier

1、该元素用来帮助定义构件输出的一些附属构件

2、附属构件与主构件对应

3、不能直接定义项目的classfier，因为附属构件不是项目直接默认生成的，而是由附加的插件帮助生成

# 仓库

## 何为Maven仓库

1、在一台工作站上，可能会有几十个Maven项目，所有项目都使用maven-compiler-plugin，这些项目中大部分用到了log4j，有一小部分用到了Spring Framework，还有另外一小部分用到了Struts2。如果每个有需要的项目中都放置一份复制品，那么不仅造成磁盘空间浪费，也难于管理

2、得益于坐标机制，任何Maven项目使用任何一个构件的方式都是完全相同的。在此基础上，Maven可以在某个位置统一存储所有Maven项目共享的构件，这个统一的位置就是仓库

3、实际的Maven项目不再各自存储其依赖文件，它们只需要声明这些依赖的坐标

4、为了实现重用，项目构件完毕后生成的构件也可以安装或部署到仓库中，供其他项目使用

## 仓库的布局

1、任何一个构件都有其唯一的坐标，根据这个坐标可以定义其在仓库中的唯一存储路径，这便是Maven仓库布局方式

groupId/artifactId/version/artifactId-version.packaging

2、例如log4j/log4j/1.2.15/log4j-1.2.15.jar

## settings

1、Maven可以选择配置$M2\_HOME/conf/settings.xml或者~/.m2/settings.xml

* 前者是全局范围的，整台机器上的所有用户都会直接受到该配置的影响
* 后者是用户范围的，只有当前用户才会受到该配置的影响

2、推荐用用户范围的settings.xml

1. 不同用户不会相互影响
2. 便于Maven更新升级

## 仓库的分类

1、对于Maven来说，仓库只分类两类：本地仓库和远程仓库

2、当Maven根据坐标寻找构件时

1. 它首先会查看本地仓库，如果本地仓库存在此构件，则直接使用
2. 如果本地方库不存在此构件，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，发现需要的构件之后，下载到本地仓库再使用

### 本地仓库

1、默认是~/.m2/repository

2、可以在settings中进行更改

<settings>

<localRespository>D:\java\respository</localRespository>

</settings>

### 远程仓库

1、一般地，对于Maven来说，每个用户只有一个本地仓库，但是可以配置多个远程仓库

### 中央仓库

1、中央仓库是一个默认的远程仓库

2、中央仓库包含了绝大多数流行的开源Java构件，以及源码、作者信息、SCM信息、许可证信息等

3、一般来说一个简单Maven项目所需要的依赖构件都能从中央仓库下载到，这也揭示了Maven为什么能做到开箱即用

### 私服

1、私服是特殊远程仓库，它是架设在局域网内的仓库服务，私服代理广域网上的远程仓库，供局域网内的Maven用户使用，当Maven需要下载构件的时候，它从私服请求，如果私服上不存在该构件，则从外部的远程仓库下载，缓存在私服上后，再为Maven的下载请求提供服务

2、优势

1. 节省自己的外网带宽
2. 加速Maven构建
3. 部署第三方构件
4. 提高稳定性，增强控制
5. 降低中央仓库的负荷

# 生命周期和插件

1、除了坐标、依赖以及仓库之外，Maven另外两个核心概念是生命周期和插件

## 何为生命周期

1、Maven从大量项目和构件工具中学习和反思，然后总结了一套高度完善的、易扩展的生命周期

2、生命周期包含了

1. 项目的清理
2. 初始化
3. 编译
4. 测试
5. 打包
6. 集成测试
7. 验证
8. 部署
9. 站点生成

3、Maven的生命周期是抽象的，这意味着生命周期本身不做任何实际的工作，在Maven的设计中，实际的任务(例如编译源代码)都由插件来完成。这种思想与设计模式中的模板方法(Template Method)非常类似

4、生命周期抽象了构建的各个步骤，定义了他们的次序，但没有提供具体实现

* 如果让用户来实现，那么又会导致用户在做重复的工作
* 因此设计了插件机制，每个构建步骤都可以绑定一个或者多个插件行为，而且Maven为大多数构建步骤编写并绑定了默认插件，例如针对编译的插件有maven-compiler-plugin，针对测试的插件有maven-surefire-plugin等

5、Maven定义的生命周期和插件机制一方面保证了所有Maven项目有一致的构件标准，另一方面又通过默认插件简化和稳定了实际项目的构建。另外此机制还提供了足够的扩展空间，用户可以通过配置现有插件或者自行编写插件来自定义构建行为

## 生命周期详解

### 三套生命周期

1、Maven拥有三套相互独立的生命周期，它们分别为clean、default和site

1. clean生命周期的目的是清理项目
2. default生命周期的目的是构建项目
3. site生命周期的目的是建立项目站点

### clean生命周期

1、clean生命周期的目的是清理项目，包含三个阶段

1. pre-clean执行一些清理前需要完成的工作
2. clean清理上一次构建生成的文件
3. post-clean执行一些清理后需要完成的工作

### default生命周期

1、default生命周期定义了真正构建时所需要执行的所有步骤，它是所有生命周期中最核心的部分，包含如下阶段

* validate
* initialize
* generate-sources
* process-sources
* generate-resources
* process-resources
* compile
* process-classes
* generate-test-sources
* process-test-sources
* generate-test-resources
* process-test-resources
* test-compile
* process-test-classes
* test
* prepare-package
* package
* pre-integration-test
* integration-test
* post-integration-test
* verify
* install
* deploy

### site生命周期

### 命令行与生命周期