# 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"

# web.xml

http://www.cnblogs.com/hafiz/p/5715523.html

## <icon>

## <display-name>

1、Web 应用名称：提供GUI工具可能会用来标记这个特定的Web应用的一个名称

2、示例

<display-name>Tomcat Example</display-name>

## <disciption>

1、Web 应用描述：给出于此相关的说明性文本

2、示例

<disciption>Tomcat Example servlets and JSP pages.</disciption>

## <context-param>

1、上下文参数：声明应用范围内的初始化参数，在servlet里面可以通过 getServletContext().getInitParameter("context/param")得到

2、示例

<context-param>

<param-name>参数名</para-name>

<param-value>参数值</param-value>

<description>参数描述</description>

</context-param>

## <filter>

1、过滤器配置：将一个名字与一个实现javaxs.servlet.Filter接口的类相关联

2、示例

<filter>

<filter-name>setCharacterEncoding</filter-name>

<filter-class>com.myTest.setCharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>GB2312</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>setCharacterEncoding</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

## <listener>

1、监听器配置

2、示例

<listener>

<listerner-class>

org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

## <servlet>

1、Servlet配置

2、示例

<servlet>

<servlet-name>servlet名称</servlet-name>

<servlet-class>servlet类全路径</servlet-class>

<init-param>

<param-name>参数名</param-name>

<param-value>参数值</param-value>

</init-param>

<run-as>

<description>Security role for anonymous access</description>

<role-name>tomcat</role-name>

</run-as>

<load-on-startup>

指定当Web应用启动时，装载Servlet的次序</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>servlet名称</servlet-name>

<url-pattern>映射路径</url-pattern>

</servlet-mapping>

## <session-config>

## <mime-mapping>

## <welcome-file-list>

## <error-page>

## <taglib>

## <resource-env-ref>

## <resource-ref>

## <security-constraint>

## <login-config>

## <security-role>

## <env-entry>

## <ejb-ref>

## <ejb-local-ref>

# pom.xml

## Properties

1、Maven中的Properties就像Ant中的，可以在POM的任何地方通过${X}来返回属性的值，X就表Poperty

2、存在以下5种不同的类型

1. env.X：将返回环境变量的值，如：${env.PATH}返回PATH环境变量的值
2. project.X：POM中的对应元素的值，'.'表示在POM中的路径，如：<project><version>1.0</version></project>可以通过${project.version}获取值
3. settings.X：settings.xml中包含的对应元素的值，'.'表示路径，如： <settings><offline>false</offline></settings>可以通过${settings.offline}获取
4. java.X：所有可以通过java.lang.System.getProperties()获取到的属性在POM属性中都是可用的，如：${java.home}
5. X：在POM的<properties />中设置的元素，如： <properties><someVar>value</someVar></properies>通过${someVar}获取

## build

### BaseBuild元素集合

1、basic elements

1. defaultGoal：执行build任务时，如果没有指定目标，将使用的默认值，如：在命令行中执行mvn，则相当于执行mvn install
2. directory：build目标文件的存放目录，默认在${basedir}/target目录
3. finalName：build目标文件的文件名，默认情况下为${artifactId}-${version}
4. filter：定义\*.properties文件，包含一个properties列表，该列表会应用的支持filter的resources中。也就是说，定义在filter的文件中的"name=value"值对会在build时代替${name}值应用到resources中。Maven的默认filter文件夹是${basedir}/src/main/filters/

2、示例

<build>

<defaultGoal>install</defaultGoal>

<directory>${basedir}/target</directory>

<finalName>${artifactId}-${version}</finalName>

<filters>

<filter>filters/filter1.properties</filter>

</filters>

...

</build>

### resources

1、build的另一个特征是指定你的项目中resources的位置。resources(通常)不是代码，他们不被编译，但是被绑定在你的项目或者用于其它什么原因，例如代码生成

2、元素详解

1. resources：一个resource元素的列表，每一个都描述与项目关联的文件是什么和在哪里
2. targetPath：指定build后的resource存放的文件夹。该路径默认是basedir。通常被打包在JAR中的resources的目标路径为META-INF
3. filtering：true/false，表示为这个resource，filter是否激活
4. directory：定义resource所在的文件夹，默认为${basedir}/src/main/resources
5. includes：指定作为resource的文件的匹配模式，用\*作为通配符
6. excludes：指定哪些文件被忽略，如果一个文件同时符合includes和excludes，则excludes生效
7. testResources：定义和resource类似，但只在test时使用，默认的test resource文件夹路径是${basedir}/src/test/resources，test resource不被部署

3、示例

<build>

...

<resources>

<resource>

<targetPath>META-INF/plexus</targetPath>

<filtering>false</filtering>

<directory>${basedir}/src/main/plexus</directory>

<includes>

<include>configuration.xml</include>

</includes>

<excludes>

<exclude>\*\*/\*.properties</exclude>

</excludes>

</resource>

</resources>

<testResources>

...

</testResources>

...

</build>

### plugins

1、除了groupId:artifactId:version标准坐标，plugin还需要如下属性：

1. extensions：true/false，是否加载plugin的extensions，默认为false
2. inherited：true/false，这个plugin是否应用到该POM的孩子POM，默认true
3. configuration：配置该plugin期望得到的properies，如上面的例子，我们为maven-jar-plugin的Mojo设置了classifier属性
4. dependencies：同base build中的dependencies有同样的结构和功能，但这里是作为plugin的依赖，而不是项目的依赖
5. executions：plugin可以有多个目标，每一个目标都可以有一个分开的配置，甚至可以绑定一个plugin的目标到一个不同的阶段。executions配置一个plugin的目标的execution

2、示例

<build>

...

<plugins>

<plugin>

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

<artifactId>maven-jar-plugin</artifactId>

<version>2.0</version>

<extensions>false</extensions>

<inherited>true</inherited>

<configuration>

<classifier>test</classifier>

</configuration>

<dependencies>...</dependencies>

<executions>...</executions>

</plugin>

</plugins>

</build>

3、configuration详解

* 如果你的POM有一个parent，它可以从parent的build/plugins或者pluginManagement集成plugin配置
* 为了阐述继承后的关系，考虑如果parent POM中存在如下plugin

<plugin>

<groupId>my.group</groupId>

<artifactId>my-plugin</artifactId>

<configuration>

<items>

<item>parent-1</item>

<item>parent-2</item>

</items>

<properties>

<parentKey>parent</parentKey>

</properties>

</configuration>

</plugin>

* 然后在继承的孩子POM中做如下配置

<pre name="code" class="html"><plugin>

<groupId>my.group</groupId>

<artifactId>my-plugin</artifactId>

<configuration>

<items>

<item>child-1</item>

</items>

<properties>

<childKey>child</childKey>

</properties>

</configuration>

</plugin></pre>

* 这样孩子POM和parent POM中都存在groupId为my.group的plugin，Maven默认的行为将是根据属性名称将两个plugin的configuration的内容进行合并。如果孩子POM中有一个属性，则该属性是有效的，如果孩子POM中没有一个属性，但parent POM中存在，则parent中的属性是有效的
* 根据这些规则，上面的例子在Maven中将得到

<pre name="code" class="html"><plugin>

<groupId>my.group</groupId>

<artifactId>my-plugin</artifactId>

<configuration>

<items>

<item>child-1</item>

</items>

<properties>

<childKey>child</childKey>

<parentKey>parent</parentKey>

</properties>

</configuration>

</plugin></pre>

* 通过在configuration元素中增加combine.children和combine.self属性，孩子POM可以控制Maven怎么合并plugin的configuration
* combine.children="append"表示父POM和子POM的属性合并起来
* combine.self="override"表示子POM的属性完全覆盖父POM的

<pre name="code" class="html"><configuration>

<items combine.children="append">

<!-- combine.children="merge" is the default -->

<item>child-1</item>

</items>

<properties combine.self="override">

<!-- combine.self="merge" is the default -->

<childKey>child</childKey>

</properties>

</configuration></pre>

* 则，现在合并后的效果如下

<pre name="code" class="html"><configuration>

<items combine.children="append">

<item>parent-1</item>

<item>parent-2</item>

<item>child-1</item>

</items>

<properties combine.self="override">

<childKey>child</childKey>

</properties>

</configuration></pre>

4、executions详解

* 假定一项绑定antrun:run目标到verify阶段，我们希望任务响应build文件夹，同时避免传递配置到他的孩子POM。你将得到一个execution

<pre name="code" class="html"><build>

<plugins>

<plugin>

<artifactId>maven-antrun-plugin</artifactId>

<version>1.1</version>

<executions>

<execution>

<id>echodir</id>

<goals>

<goal>run</goal>

</goals>

<phase>verify</phase>

<inherited>false</inherited>

<configuration>

<tasks>

<echo>

Build Dir: ${project.build.directory}</echo>

</tasks>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build></pre>

* 子标签介绍
* id：标识，用于和其他execution区分。当这个阶段执行时，它将以这个形式展示：[plugin:goal execution: id]。在这里为：[antrun:run execution: echodir]
* goals：一个plugin的execution的目标列表
* phase：目标执行的阶段，具体值看Maven的生命周期列表
* inherited：是否继承
* configuration：在指定的目标下的配置

### pluginManagement

1、pluginManagement的元素的配置和plugins的配置是一样的，只是这里的配置只是用于集成，在孩子POM中指定使用

2、示例

* 例如，在父POM中做如下配置

<build>

...

<pluginManagement>

<plugins>

<plugin>

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

<artifactId>maven-jar-plugin</artifactId>

<version>2.2</version>

<executions>

<execution>

<id>pre-process-classes</id>

<phase>compile</phase>

<goals>

<goal>jar</goal>

</goals>

<configuration>

<classifier>pre-process</classifier>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</pluginManagement>

...

</build>

* 则在孩子POM中，我们只需要如下配置，这样就可以大大的简化孩子POM中的配置

<build>

...

<plugins>

<plugin>

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

<artifactId>maven-jar-plugin</artifactId>

</plugin>

</plugins>

...

</build>

### reporting

1、Reporting包含的属性对应到site阶段(见Maven生命周期)。特定的Maven插件能产生定义和配置在reporting元素下的报告

2、示例

* 例如，产生Javadoc报告

<reporting>

<outputDirectory>${basedir}/target/site</outputDirectory>

<plugins>

<plugin>

<artifactId>maven-project-info-reports-plugin</artifactId>

<version>2.0.1</version>

<reportSets>

<reportSet></reportSet>

</reportSets>

</plugin>

</plugins>

</reporting>

* 对于reportSets

<reportSets>

<reportSet>

<id>sunlink</id>

<reports>

<report>javadoc</report>

</reports>

<inherited>true</inherited>

<configuration>

<links>

<link>http://java.sun.com/j2se/1.5.0/docs/api/</link>

</links>

</configuration>

</reportSet>

</reportSets>