# 第一章 开发框架 maven

## **pom.xml详解**

POM是项目对象模型(Project Object Model)的简称,它是Maven项目中的文件，使用XML表示，名称叫做pom.xml。

该文件用于管理：源代码、配置文件、开发者的信息和角色、问题追踪系统、组织信息、项目授权、项目的url、项目的依赖关系等等。事实上，在Maven世界中，project可以什么都没有，甚至没有代码，但是必须包含pom.xml文件。

## **概览**

下面是一个POM项目中的pom.xml文件中包含的元素。注意，其中的modelVersion是4.0.0，这是当前仅有的可以被Maven2&3同时支持的POM版本，它是必须的。

<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>

<!-- 基本设置 The Basics -->

<groupId>...</groupId>

<artifactId>...</artifactId>

<version>...</version>

<packaging>...</packaging>

<dependencies>...</dependencies>

<parent>...</parent>

<dependencyManagement>...</dependencyManagement>

<modules>...</modules>

<properties>...</properties>

<!-- 构建过程的设置 Build Settings -->

<build>...</build>

<reporting>...</reporting>

<!-- 项目信息设置 More Project Information -->

<name>...</name>

<description>...</description>

<url>...</url>

<inceptionYear>...</inceptionYear>

<licenses>...</licenses>

<organization>...</organization>

<developers>...</developers>

<contributors>...</contributors>

<!-- 环境设置 Environment Settings -->

<issueManagement>...</issueManagement>

<ciManagement>...</ciManagement>

<mailingLists>...</mailingLists>

<scm>...</scm>

<prerequisites>...</prerequisites>

<repositories>...</repositories>

<pluginRepositories>...</pluginRepositories>

<distributionManagement>...</distributionManagement>

<profiles>...</profiles></project>

## **1.1 基本配置**

### **1.1.1 maven的写作相关配置**

一个最简单的pom.xml的定义必须包含modelVersion、groupId、artifactId和version这四个元素，当然这其中的元素也是可以从它的父项目中继承的。在Maven中，使用groupId、artifactId和version组成groupdId:artifactId:version的形式来唯一确定一个项目：

<?xml version="1.0" encoding="UTF-8"?><project xmlns="http://maven.apache.org/POM/4.0.0"

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

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

http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<!--

含义：组织标识，定义了项目属于哪个组，风向标，坐标，或者说若把本项目打包

用途：此名称则是本地仓库中的路径，列如：otowa.user.dao，在M2\_REPO目录下，将是: otowa/user/dao目录

命名规范:项目名称，模块，子模块

-->

<groupId>otowa.user.dao</groupId>

<!--

含义：项目名称也可以说你所模块名称，定义当面Maven项目在组中唯一的ID

用途：例如：user-dao，在M2\_REPO目录下，将是：otowa/user/dao/user-dao目录

命名规范:唯一就好

-->

<artifactId>user-dao</artifactId>

<!--

含义：项目当前的版本号

用途：例如：0.0.1-SNAPSHOT，在M2\_REPO目录下，将是：otowa/user/dao/user-dao/0.0.1-SNAPSHOT目录

-->

<version>0.0.1-SNAPSHOT</version>

<!-- 打包的格式，可以为：pom , jar , maven-plugin , ejb , war , ear , rar , par -->

<packaging>war</packaging>

<!-- 元素声明了一个对用户更为友好的项目名称 -->

<name>maven</name>  
</project>

### **1.1.2 POM之间的关系，集成、聚合与依赖**

我们知道Maven在建立项目的时候是基于Maven项目下的pom.xml进行的，我们项目依赖的信息和一些基本信息都是在这个文件里面定义的。那如果当我们有多个项目要进行，这多个项目有些配置内容是相同的，有些是要彼此关联的，那如果按照传统的做法的话我们就需要在多个项目中都定义这些重复的内容。这无疑是非常耗费时间和不易维护的。好在Maven给我们提供了一个pom的继承和聚合的功能。

对于使用java的人而言，继承这个词大家应该都不陌生。要继承pom就需要有一个父pom，在Maven中定义了超级pom.xml，任何没有申明自己父pom.xml的pom.xml都将默认继承自这个超级pom.xml。

位置：

在Maven 2.xxx版本中，比如maven-2.0.9-uber.jar，打开此Jar文件后，在包包org.apache.maven.project下会有pom-4.0.0.xml的文件。  
但是到了3.xxx版本之后在： aven安装目录下的：/lib/maven-model-builder-version.jar中 \org\apache\maven\mode目录中的pom-4.0.0.xml。

先来看一下这个超级pom.xml的定义：

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

<!--

Licensed to the Apache Software Foundation (ASF) under one

or more contributor license agreements. See the NOTICE file

distributed with this work for additional information

regarding copyright ownership. The ASF licenses this file

to you under the Apache License, Version 2.0 (the

"License"); you may not use this file except in compliance

with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing,

software distributed under the License is distributed on an

"AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY

KIND, either express or implied. See the License for the

specific language governing permissions and limitations

under the License.-->

<!-- START SNIPPET: superpom --><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>

<build>

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

<outputDirectory>${project.build.directory}/classes</outputDirectory>

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

<testOutputDirectory>${project.build.directory}/test-classes</testOutputDirectory>

<sourceDirectory>${project.basedir}/src/main/java</sourceDirectory>

<scriptSourceDirectory>${project.basedir}/src/main/scripts</scriptSourceDirectory>

<testSourceDirectory>${project.basedir}/src/test/java</testSourceDirectory>

<resources>

<resource>

<directory>${project.basedir}/src/main/resources</directory>

</resource>

</resources>

<testResources>

<testResource>

<directory>${project.basedir}/src/test/resources</directory>

</testResource>

</testResources>

<pluginManagement>

<!-- NOTE: These plugins will be removed from future versions of the super POM -->

<!-- They are kept for the moment as they are very unlikely to conflict with lifecycle mappings (MNG-4453) -->

<plugins>

<plugin>

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

<version>1.3</version>

</plugin>

<plugin>

<artifactId>maven-assembly-plugin</artifactId>

<version>2.2-beta-5</version>

</plugin>

<plugin>

<artifactId>maven-dependency-plugin</artifactId>

<version>2.8</version>

</plugin>

<plugin>

<artifactId>maven-release-plugin</artifactId>

<version>2.3.2</version>

</plugin>

</plugins>

</pluginManagement>

</build>

<reporting>

<outputDirectory>${project.build.directory}/site</outputDirectory>

</reporting>

<profiles>

<!-- NOTE: The release profile will be removed from future versions of the super POM -->

<profile>

<id>release-profile</id>

<activation>

<property>

<name>performRelease</name>

<value>true</value>

</property>

</activation>

<build>

<plugins>

<plugin>

<inherited>true</inherited>

<artifactId>maven-source-plugin</artifactId>

<executions>

<execution>

<id>attach-sources</id>

<goals>

<goal>jar</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<inherited>true</inherited>

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

<executions>

<execution>

<id>attach-javadocs</id>

<goals>

<goal>jar</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<inherited>true</inherited>

<artifactId>maven-deploy-plugin</artifactId>

<configuration>

<updateReleaseInfo>true</updateReleaseInfo>

</configuration>

</plugin>

</plugins>

</build>

</profile>

</profiles>

</project>

<!-- END SNIPPET: superpom -->

对于一个pom.xml来说有几个元素是必须定义的，一个是project根元素，然后就是它里面的modelVersion、groupId、artifactId和version。由上面的超级pom.xml的内容我们可以看到pom.xml中没有groupId、artifactId和version的定义，所以我们在建立自己的pom.xml的时候就需要定义这三个元素。和java里面的继承类似，子pom.xml会完全继承父pom.xml中所有的元素，而且对于相同的元素，一般子pom.xml中的会覆盖父pom.xml中的元素，但是有几个特殊的元素它们会进行合并而不是覆盖。这些特殊的元素是：

dependencies

developers

contributors

plugin列表，包括plugin下面的reports

resources

#### 1.1.2.1 继承

##### 被继承项目与继承项目是父子目录关系

现在假设我们有一个项目projectA，它的pom.xml定义如下：

<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/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.tiantian.mavenTest</groupId>

<artifactId>projectA</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

</project>

然后我们有另一个项目projectB，而且projectB是跟projectA的pom.xml文件处于同一个目录下，这时候如果projectB需要继承自projectA的话我们可以这样定义projectB的pom.xml文件。

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

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<parent>

<groupId>com.tiantian.mavenTest</groupId>

<artifactId>projectA</artifactId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<groupId>com.tiantian.mavenTest</groupId>

<artifactId>projectB</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

</project>

由projectB的pom.xml文件的定义我们可以知道，当需要继承指定的一个Maven项目时，我们需要在自己的pom.xml中定义一个parent元素，在这个元素中指明需要继承项目的groupId、artifactId和version。

##### 被继承项目与继承项目的目录结构不是父子关系

当被继承项目与继承项目的目录结构不是父子关系的时候，我们再利用上面的配置是不能实现Maven项目的继承关系的，这个时候我们就需要在子项目的pom.xml文件定义中的parent元素下再加上一个relativePath元素的定义，用以描述父项目的pom.xml文件相对于子项目的pom.xml文件的位置。

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

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<parent>

<groupId>com.tiantian.mavenTest</groupId>

<artifactId>projectA</artifactId>

<version>1.0-SNAPSHOT</version>

<relativePath>../projectA/pom.xml</relativePath>

</parent>

<modelVersion>4.0.0</modelVersion>

<groupId>com.tiantian.mavenTest</groupId>

<artifactId>projectB</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

</project>

#### 1.1.2.2 聚合

对于聚合这个概念搞java的人应该都不会陌生。先来说说我对聚合和被聚合的理解，比如说如果projectA聚合到projectB，那么我们就可以说projectA是projectB的子模块， projectB是被聚合项目，也可以类似于继承那样称为父项目。对于聚合而言，这个主体应该是被聚合的项目。所以，我们需要在被聚合的项目中定义它的子模块，而不是像继承那样在子项目中定义父项目。具体做法是：

* + 修改被聚合项目的pom.xml中的packaging元素的值为pom
  + 在被聚合项目的pom.xml中的modules元素下指定它的子模块项目

对于聚合而言，当我们在被聚合的项目上使用Maven命令时，实际上这些命令都会在它的子模块项目上使用。这就是Maven中聚合的一个非常重要的作用。假设这样一种情况，你同时需要打包或者编译projectA、projectB、projectC和projectD，按照正常的逻辑我们一个一个项目去使用mvn compile或mvn package进行编译和打包，对于使用Maven而言，你还是这样使用的话是非常麻烦的。因为Maven给我们提供了聚合的功能。我们只需要再定义一个超级项目，然后在超级项目的pom.xml中定义这个几个项目都是聚合到这个超级项目的。之后我们只需要对这个超级项目进行mvn compile，它就会把那些子模块项目都进行编译。

##### 被聚合项目(父)和子模块项目(子)在目录结构上是父子关系

<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.tiantian.mavenTest</groupId>

<artifactId>projectA</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>pom</packaging>

<modules>

<module>projectB</module>

</modules>

</project>

由上面的定义我们可以看到被聚合的项目的packaging类型应该为pom，而且一个项目可以有多个子模块项目。  
对于聚合这种情况，我们使用子模块项目的artifactId来作为module的值，表示子模块项目相对于被聚合项目的地址，  
在上面的示例中就表示子模块projectB是处在被聚合项目的子目录下，即与被聚合项目的pom.xml处于同一目录。这里使用的module值是子模块projectB对应的目录名projectB，而不是子模块对应的artifactId。  
这个时候当我们对projectA进行mvn package命令时，实际上Maven也会对projectB进行打包。

##### 被聚合项目与子模块项目在目录结构上不是父子关系

那么当被聚合项目与子模块项目在目录结构上不是父子关系的时候，我们应该怎么来进行聚合呢？还是像继承那样使用relativePath元素吗？答案是非也，具体做法是在module元素中指定以相对路径的方式指定子模块。我们来看下面一个例子。

<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.tiantian.mavenTest</groupId>

<artifactId>projectA</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>pom</packaging>

<modules>

<module>../projectB</module>

</modules>

</project>

注意看module的值是“../projectB”，我们知道“..”是代表当前目录的上层目录，所以它表示子模块projectB是被聚合项目projectA的pom.xml文件所在目录（即projectA）的上层目录下面的子目录，即与projectA处于同一目录层次。  
注意，这里的projectB对应的是projectB这个项目的目录名称，而不是它的artifactId。

##### 依赖关系列表（dependency list）是POM的重要部分

项目之间的依赖是通过pom.xml文件里面的dependencies元素下面的dependency元素进行的。一个dependency元素定义一个依赖关系。在dependency元素中我们主要通过依赖项目的groupId、artifactId和version来定义所依赖的项目。

       先来看一个简单的项目依赖的示例吧，假设我现在有一个项目projectA，然后它里面有对junit的依赖，那么它的pom.xml就类似以下这个样子：

<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.tiantian.mavenTest</groupId>

<artifactId>projectB</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<dependencies>

<dependency>

***<groupId>junit</groupId>***

***<artifactId>junit</artifactId>***

***<version>3.8.1</version>***

<scope>test</scope>

<optional>true</optional>

</dependency>

</dependencies>

</project>

**roupId, artifactId, version:**描述了依赖的项目唯一标志。

**type：**对应于依赖项目的packaging类型，默认是jar。

**scope:**用于限制相应的依赖范围、传播范围。scope的主要取值范围如下：

**test:**

在测试范围有效，它在执行命令test的时候才执行，并且它不会传播给其他模块进行引入，比如 junit,dbunit 等测试框架。

**compile(default 默认):**

这是它的默认值，这种类型很容易让人产生误解，以为只有在编译的时候才是需要的，其实这种类型表示所有的情况都是有用的，包括编译和运行时。而且这种类型的依赖性是可以传递的。

**runtime:**

在程序运行的时候依赖，在编译的时候不依赖。

**provided:**

这个跟compile很类似，但是它表示你期望这个依赖项目在运行时由JDK或者容器来提供。这种类型表示该依赖只有在测试和编译的情况下才有效，在运行时将由JDK或者容器提供。这种类型的依赖性是不可传递的。**比如 javaee**：

**eclipse开发web环境中**是没有javaee必须要手动添加。

**myeclipse新建web项目**会有JavaEE(servlet-api.jar,jsp-api.jar...)web容器依赖的jar包，一般都是做开发的时候才使用。但是myeclipse不会把这些 jar包发布的，lib下你是找不到javaee引入的jar包,因为myeclipse发布项目的时候会忽略它。为什么？因为tomcat容器bin/lib已经存在了这个jar包了。

**system：**

这种类型跟provided类似，唯一不同的就是这种类型的依赖我们要自己提供jar包，这需要与另一个元素systemPath来结合使用。systemPath将指向我们系统上的jar包的路径，而且必须是给定的绝对路径。

systemPath：上面已经说过了这个元素是在scope的值为system的时候用于指定依赖的jar包在系统上的位置的，而且是绝对路径。该元素必须在依赖的 jar包的scope为system时才能使用，否则Maven将报错。

optional：当该项目本身作为其他项目的一个依赖时标记该依赖为可选项。假设现在projectA有一个依赖性projectB，我们把projectB这个依赖项设为optional，这表示projectB在projectA的运行时不一定会用到。这个时候如果我们有另一个项目projectC，它依赖于projectA，那么这个时候因为projectB对于projectA是可选的，所以Maven在建立projectC的时候就不会安装projectB，这个时候如果projectC确实需要使用到projectB，那么它就可以定义自己对projectB的依赖。当一个依赖是可选的时候，我们把optional元素的值设为true，否则就不设置optional元素。

exclusions：考虑这样一种情况，我们的projectA依赖于projectB，然后projectB又依赖于projectC，但是在projectA里面我们不需要projectB依赖的projectC，那么这个时候我们就可以在依赖projectB的时候使用exclusions元素下面的exclusion排除projectC。这个时候我们可以这样定义projectA对projectB的依赖：

<dependencies>

<dependency>

<groupId>com.tiantian.mavenTest</groupId>

<artifactId>projectB</artifactId>

<version>1.0-SNAPSHOT</version>

<exclusions>

<exclusion>

<groupId>com.tiantian.mavenTest</groupId>

<artifactId>projectC</artifactId>

</exclusion>

</exclusions>

</dependency>

</dependencies>

### **1.1.3 属性**

在pom.xml文件中我们可以使用${propertyName}的形式引用属性。是值的占位符，类似EL，类似ant的属性，比如${X}，可用于pom文件任何赋值的位置。有以下分类：

* env.propertyName：

这种形式表示引用的是环境变量，比如我们需要引用当前系统的环境变量PATH的时候，就可以使用${env.PATH}。

* project.propertyName：

这种形式表示引用的是当前这个pom.xml中project根元素下面的子元素的值。比如我们需要引用当前project下面的version的时候，就可以使用${project.version}。

* settings.propertyName：

这种形式引用的是Maven本地配置文件settings.xml或本地Maven安装目录下的settings.xml文件根元素settings下的元素。比如我们需要引用settings下的本地仓库localRepository元素的值时，我们可以用${settings.localRepository}

* Java System Properties：

java的系统属性，所有在java中使用java.lang.System.getProperties()能够获取到的属性都可以在pom.xml中引用，比如${java.home}。

##### **1.1.3.1 自定义属性段**

pom.xml中properties元素下面的子元素作为属性。假如在pom.xml中有如下一段代码

<properties>

<hello.world>helloWorld</hello.world>

</properties>，

那么我们就可以使用${hello.world}引用到对应的helloWorld。

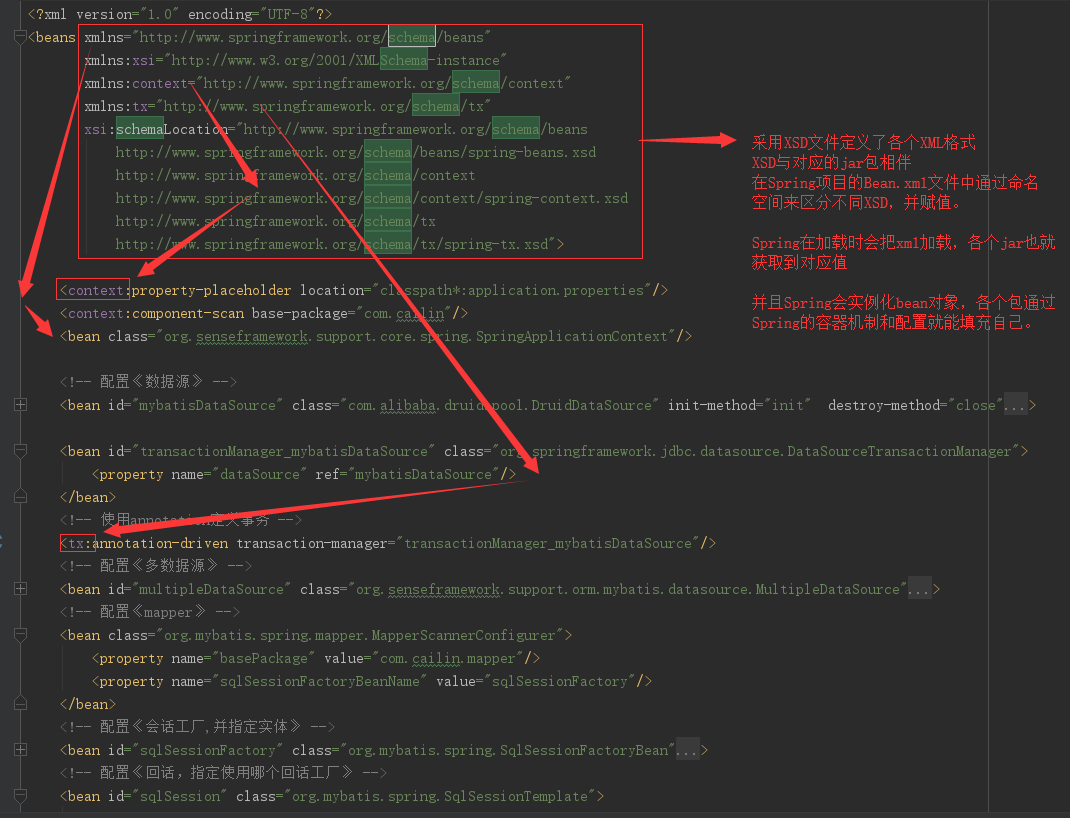
##### **1.1.3.2 dependencyManagement段**

在很多模块的项目中，通过抽象一层作为各个模块的父pom；在父pom中定义DepencyManagement元素。通过它元素来管理jar包的版本，让子项目中引用一个依赖而不用显示的列出版本号。Maven会沿着父子层次向上走，直到找到一个拥有dependencyManagement元素的项目，然后它就会使用在这个dependencyManagement元素中指定的版本号。

# 第二章 开发框架 Spring

## **2.1 Spring的机制**

**org.springframework.context.support.ClassPathXmlApplicationContext与Beans.xml的绑定**



* 一个框架的各个模块（jar）会定义自己的XSD；
* 一个Spring项目会定义自己的bean.xml，在bean.xml中会将框架各个jar的XSD放到不同的命名空间，并在xml中赋值（也就配置了相关jar或者说相关框架）；Spring本身是个依赖注入的容器，代码中标注或bean.xml中定义的bean实例被Spring管理，这些实例对象可以作为其他对象的属性赋值或代码中获取，从而无缝衔接。
* 基于以上两点就能达到一个程序的对象网络通过配置文件就能搭建，因为业务逻辑基于接口，进而可以替换对象实例，达到局部实现的更新而不需整体重新编译。

## **2.2 用来标记stereotype的annotation**

@Component，@Controller，@Repository，@Service。

这四个都在org.springframework.stereotype包下面，后面3个都属于@Component。

可以理解为@Component是@Controller，@Repository，@Service的基类

### **Import org.springframework.stereotype.\***

### **2.2.1 @Component是用来标记任何被Spring管理的组件。**

它表示一个通用注释用于说明一个类是一个spring容器管理的类，即该类已经拉入到spring的管理中了。（把普通pojo实例化到spring容器中，相当于配置文件中的<bean id="" class=""/>）  
举例：  
<context:component-scan base-package=”com.mmnc”>

### **2.2.2 @Controller用来标记presentation层（比如web controller）。**

的细化的类。（注入服务）

### **2.2.3 @Repository用来标记persistence层（比如DAO）。**

它是对@Component的细化，用来给持久层的类定义一个名字，让Spring根据这个名字关联到这个类。（实现dao访问）  
例如：  
@Repository("userDao")  
public class UserDaoImpl  implements UserDao{  
   ........................................  
}  
声明了UserDaoImpl  在Spring容器中叫userDao这个名字。

### **2.2.4 @Service用来标记service层。**

它是对@Component的细化，用来给服务层的类定义一个名字，让Spring根据这个名字关联到这个类。（注入dao）  
举例：  
Chinese实例化为chinese，如果需要自己改名字则:@Service("你自己改的bean名")。

### **import org.springframework.beans.factory.annotation.**

### **2.2.5 @Override**

它是伪代码,表示重写(当然不写也可以)，不过用它有以下好处:  
1、可以当注释用,方便阅读；  
2、编译器可以给你验证@Override下面的方法名是否是你父类中所有的，如果没有则报错。例如，你如果没写@Override，而你下面的方法名又写错了，这时你的编译器是可以编译通过的，因为编译器以为这个方法是你的子类中自己增加的方法。

关于第二种优点的举例：  
在重写父类的onCreate时，在方法前面加上@Override 系统可以帮你检查方法的正确性。

@Override  
public void onCreate(Bundle savedInstanceState)  
{…….}

这种写法是正确的，如果你写成：

@Override  
public void oncreate(Bundle savedInstanceState)  
{…….}

编译器会报如下错误：The method oncreate(Bundle) of type HelloWorld must override or implement a supertype method，以确保你正确重写onCreate方法（因为oncreate应该为onCreate）。而如果你不加@Override，则编译器将不会检测出错误，而是会认为你为子类定义了一个新方法：oncreate。

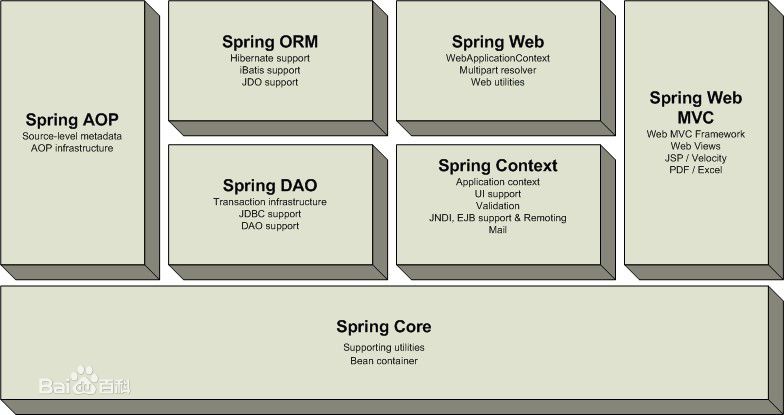
### **2.2.6 @Autowired**

它可以对类成员变量、构造函数进行标注，完成自动装配的工作。 通过 @Autowired的使用来消除属性set方法和构造函数的参数传入。只要配置文件中含有与变量名和构造函数参数名相同ID的bean就会被自动注入。  
举例：  
package com.baobaotao;     
import org.springframework.beans.factory.annotation.Autowired;     
   
public class Boss {     
    @Autowired    
    private Car car;     
    
    @Autowired    
    private Office office;     
    …     
}

### **2.2.7 @Required**

它可以用来指明构造函数中的参数对应的属性为必须

## **2.3、Spring概述**



### **2.3.1 概述**

Spring 是最受欢迎的企业级 Java 应用程序开发框架。数以百万的来自世界各地的开发人员使用 Spring 框架来创建好性能、易于测试、可重用的代码。

Spring 框架是一个开源的 Java 平台，它最初是由 Rod Johnson 编写的，并且 2003 年 6 月首次在 Apache 2.0 许可下发布。

当谈论到大小和透明度时， Spring 是轻量级的。 Spring 框架的基础版本是在 2 MB 左右的。

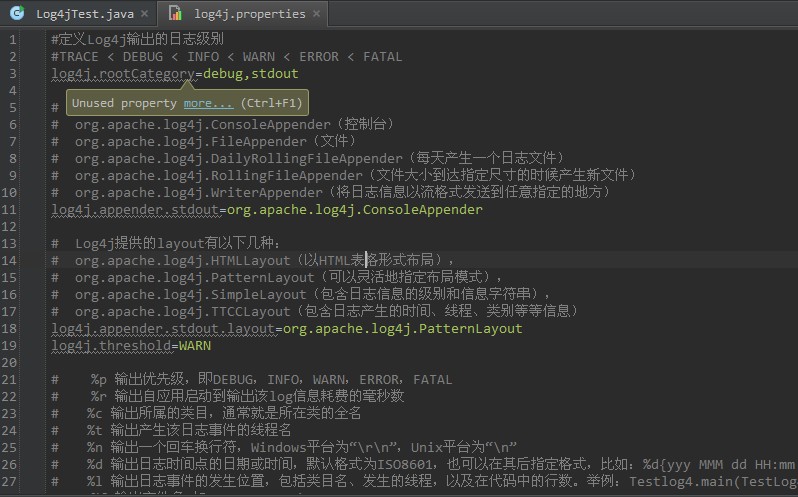
Spring 框架的核心特性可以用于开发任何 Java 应用程序，但是在 Java EE 平台上构建 web 应用程序是需要扩展的。 Spring 框架的目标是使 J2EE 开发变得更容易使用，通过启用基于 POJO 编程模型来促进良好的编程实践。

# 第三章 开发环境 IntelliJ IDEA

## **3.1 基础操作**

### **IntelliJ IDEA Properties 文件代码不高亮 解决方案**

我们在 IDEA 编辑 properties 文件的时候，会发现敲出的代码不会高亮显示，会提示你 “Unused property more... (Ctrl+F1) ”，如下图显示，这是为什么呢？



这是因为IDEA默认对properties文件是进行关键字检查的，虽然我不知道这有啥用，但是很明显对我们来说暂时没用，而且还造成了我们编写properties时没有高亮显示，看着一片灰白色的代码心情一顿烦躁。所以我们要解决这个问题

在setting里面搜索 ”Unused“ 或者 "Inspections"  关键字 将其中  "Properties Files" 的Unused Property  的检查去掉，我们就解决了这个问题

# 第四章 语言基础-Java

## **4.1、语言字典**

### **DAO 数据访问对象，CRUD操作接口实现**

### **CRUD是指在做计算处理时的增加(Create)、读取查询(Read)、更新(Update)和删除(Delete)**

几个单词的首字母简写。crud主要被用在描述软件系统中数据库或者[持久层](https://baike.baidu.com/item/%E6%8C%81%E4%B9%85%E5%B1%82" \t "https://baike.baidu.com/item/crud/_blank)的基本操作功能。

## **4.2、类库**

### **4.2.1 常用的 Java 核心包 (Java Core Package)**

#### 4.2.1.1 java.lang=Java 编程语言的基本类库

只有此类不需要导入就可以使用

##### **1. object 类**

##### **2. 数学类 (Math)**

##### **3. 数据类型类**

##### **4. 线程类**

##### **5. 字符串类 (String 类和 StringBuffer 类 )**

##### **6. 系统及运行类 (System 类和 Runtime 类 )**

##### **7.  错误和异常处理类 (Throwable 、 Exception 、 Error)**

##### **8.  过程类 (process)**

##### **4.2.1.1.1 Java.lang.annotation**

###### 一、认识注解

注解(Annotation)很重要，未来的开发模式都是基于注解的，JPA是基于注解的，Spring2.5以上都是基于注解的，Hibernate3.x以后也是基于注解的，现在的Struts2有一部分也是基于注解的了，注解是一种趋势，现在已经有不少的人开始用注解了，注解是JDK1.5之后才有的新特性

JDK1.5之后内部提供的三个注解：  
@Deprecated 意思是“废弃的，过时的”   
@Override 意思是“重写、覆盖”   
@SuppressWarnings 意思是“压缩警告”

###### **二、自定义注解类**

package cn.gacl.annotation;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME) //Retention注解决定MyAnnotation注解的生命周期

@Target( { ElementType.METHOD, ElementType.TYPE })

//Target注解决定MyAnnotation注解可以加在哪些成分上，如加在类身上，或者属性身上，或者方法身上等成分

/\*

\* @Retention(RetentionPolicy.SOURCE)

\* 这个注解的意思是让MyAnnotation注解只在java源文件中存在，编译成.class文件后注解就不存在了

\* @Retention(RetentionPolicy.CLASS)

\* 这个注解的意思是让MyAnnotation注解在java源文件(.java文件)中存在，编译成.class文件后注解也还存在，

\* 被MyAnnotation注解类标识的类被类加载器加载到内存中后MyAnnotation注解就不存在了

\*/

public @interface MyAnnotation {

}

#### 4.2.1.2 java.applet=创建 applet 需要的所有类

#### 4.2.1.3 java.awt= 创建用户界面以及绘制和管理图形、图像的类

#### 4.2.1.4 java.io=通过数据流、对象序列以及文件系统实现的系统输入、输出

#### 4.2.1.5 java.net=用于实现网络通讯应用的所有类

#### 4.2.1.6 java.util=集合类、时间处理模式、日期时间工具等各类常用工具包

##### **1. 日期类、日历类（ Data 、 Calendar 、 GregorianCalendar ）**

##### **2. 随机数类（ Random ）**

##### **3. 位运算类（ BitSet ）**

##### **4. 矢量类（ Vector ）**

##### **5. 数据结构类（ Stack ）**

##### **6. 散列表类（ Hashtable ）**

##### **7. StringTokenizer类**

#### 4.2.1.7 java.sql=访问和处理来自于 Java 标准数据源数据的类

#### 4.2.1.8 java.test=以一种独立于自然语言的方式处理文本、日期、数字和消息的类和接口

#### 4.2.1.9 java.security=设计网络安全方案需要的一些类

#### 4.2.1.10 java.beans=开发 Java Beans 需要的所有类

#### 4.2.1.11 java.math=简明的整数算术以及十进制算术的基本函数

#### 4.2.1.12 java.rmi=与远程方法调用相关的所有类

#### 4.2.1.12 java.nio

#### 4.2.1.12 java.time

### **4.2.2 常用的 Java 扩展包 (Java Extension Package)**

#### 4.2.2.1 javax.accessibility=定义了用户界面组件之间相互访问的一种机制

#### 4.2.2.2 javax.naming.\*=为命名服务提供了一系列类和接口

#### 4.2.2.3 javax.swing.\*=提供了一系列轻量级的用户界面组件，是目前 Java 用户界面常用的包

## **4.3、第三方类库**

### **4.3.1 SLF4J(Simple logging Facade for Java)**

SLF4J不是一个真正的日志实现，而是一个抽象层（ [abstraction layer](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html" \t "https://www.cnblogs.com/549294286/p/_blank)），它允许你在后台使用任意一个日志类库。如果是在编写供内外部都可以使用的API或者通用类库，那么你真不会希望使用你类库的客户端必须使用你选择的日志类库。

如果一个项目已经使用了log4j，而你加载了一个类库，比方说 Apache Active MQ——它依赖于于另外一个日志类库logback，那么你就需要把它也加载进去。但如果Apache Active MQ使用了SLF4J，你可以继续使用你的日志类库而无语忍受加载和维护一个新的日志框架的痛苦。

总的来说，SLF4J使你的代码独立于任意一个特定的日志API，这是一个对于开发API的开发者很好的思想。虽然抽象日志类库的思想已经不是新鲜的事物而且Apache commons logging也已经在使用这种思想了，但现在SLF4J正迅速成为Java世界的日志标准。让我们再看看几个使用SLF4J而不是log4j、logback或者java.util.logging的理由。

#### 4.3.1.1 SLF4J工作机制

项目中的日志系统使用的是slf4j + logback。slf4j作为一个简单日志门面，为各种loging APIs（java.util.logging, logback, log4j）提供一个简单统一的接口，有利于维护和各个类的日志处理方式统一。Logback作为一个具体的日志组件，完成具体的日志操作。

**增加pom.xml依赖配置**

<!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-api -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.2</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-log4j12 -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.2</version>

</dependency>

**配置具体日志库配置文件**

### 设置###

log4j.rootLogger = debug,stdout,D,E

### 输出信息到控制抬 ###

log4j.appender.stdout = org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target = System.out

log4j.appender.stdout.layout = org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern = [%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} method:%l%n%m%n

### 输出DEBUG 级别以上的日志到=E://logs/error.log ###

log4j.appender.D = org.apache.log4j.DailyRollingFileAppender

log4j.appender.D.File = D://logs/log.log

log4j.appender.D.Append = true

log4j.appender.D.Threshold = DEBUG

log4j.appender.D.layout = org.apache.log4j.PatternLayout

log4j.appender.D.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss} [ %t:%r ] - [ %p ] %m%n

### 输出ERROR 级别以上的日志到=E://logs/error.log ###

log4j.appender.E = org.apache.log4j.DailyRollingFileAppender

log4j.appender.E.File =D://logs/error.log

log4j.appender.E.Append = true

log4j.appender.E.Threshold = ERROR

log4j.appender.E.layout = org.apache.log4j.PatternLayout

log4j.appender.E.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss} [ %t:%r ] - [ %p ] %m%n

**日志使用样例**

import org.slf4j.Logger;import org.slf4j.LoggerFactory;

public class Log4jTestDemo {

private static Logger logger = LoggerFactory.getLogger(Log4jTestDemo.class); /\*\*

\* @param args

\*/

public static void main(String[] args) {

// System.out.println("This is println message.");

// 记录debug级别的信息

logger.debug("This is debug message.");

// 记录info级别的信息

logger.info("This is info message.");

// 记录error级别的信息

logger.error("This is error message.");

}

}

##### **问题1：SLF4J怎么和具体的日志库勾搭上的**

大家对spring使用的比较多的话，就知道将某个实现类注给其接口的时候，都是需要明确指出的，无论是通过配置文件的方式还是注解的方式。

##### **问题2：什么驱动具体日志库配置文件加载**

我们加上logback的配置文件，仅仅在src/main/resources(相当于classpath)下加logback.xml，发现生成了日志文件(若没有设置日志文件路径，那么日志文件生成在当前工程下)，并且控制台输出结果如下

* 看看LoggerFactory.getLogger()的实现

/\*\*

\* Return a logger named according to the name parameter using the statically

\* bound {@link ILoggerFactory} instance.

\*

\* @param name The name of the logger.

\* @return logger

\*/

public static Logger getLogger(String name) {

// 获取日志工厂

ILoggerFactory iLoggerFactory = getILoggerFactory();

// 返回日志实例

return iLoggerFactory.getLogger(name);

}

* 看看getILoggerFactory方法

/\*\*

\* Return the {@link ILoggerFactory} instance in use.

\* <p/>

\* <p/>

\* ILoggerFactory instance is bound with this class at compile time. // 编译时绑定工厂实例

\*

\* @return the ILoggerFactory instance in use

\*/

public static ILoggerFactory getILoggerFactory() {

if (INITIALIZATION\_STATE == UNINITIALIZED) {

INITIALIZATION\_STATE = ONGOING\_INITIALIZATION;

// 执行初始化 performInitialization();

}

switch (INITIALIZATION\_STATE) {

case SUCCESSFUL\_INITIALIZATION:

// 若初始化成功，则返回日志工厂

return StaticLoggerBinder.getSingleton().getLoggerFactory();

case NOP\_FALLBACK\_INITIALIZATION:

return NOP\_FALLBACK\_FACTORY;

case FAILED\_INITIALIZATION:

throw new IllegalStateException(UNSUCCESSFUL\_INIT\_MSG);

case ONGOING\_INITIALIZATION:

// support re-entrant behavior.

// See also http://bugzilla.slf4j.org/show\_bug.cgi?id=106

return TEMP\_FACTORY;

}

throw new IllegalStateException("Unreachable code");

}

* 跟进performInitialization方法

private final static void performInitialization() {

bind();

if (INITIALIZATION\_STATE == SUCCESSFUL\_INITIALIZATION) {

versionSanityCheck();

}

}

* 跟进bind方法

private final static void bind() {

try {

// 从classpath获取可能的日志绑定者，就是找出所有slf4j的实现，并将它们的资源路径存放到staticLoggerBinderPathSet

Set<URL> staticLoggerBinderPathSet = findPossibleStaticLoggerBinderPathSet();

// 若有多个(多余1个)绑定者，就是从classpath中找到了多个slf4j的实现，那么就打印警告。这个方法就不跟进了，感兴趣的自己跟进

reportMultipleBindingAmbiguity(staticLoggerBinderPathSet);

// the next line does the binding 真正的绑定，将具体的实现绑定到slf4j

StaticLoggerBinder.getSingleton();

// 修改初始化状态为初始化成功

INITIALIZATION\_STATE = SUCCESSFUL\_INITIALIZATION;

// 报告真实的绑定信息

reportActualBinding(staticLoggerBinderPathSet);

fixSubstitutedLoggers();

} catch (NoClassDefFoundError ncde) {

// 若有多个绑定者，则会抛此异常，Java虚拟机在编译时能找到合适的类，而在运行时不能找到合适的类导致的错误

//，jvm不知道用哪个StaticLoggerBinder

String msg = ncde.getMessage();

if (messageContainsOrgSlf4jImplStaticLoggerBinder(msg)) {

INITIALIZATION\_STATE = NOP\_FALLBACK\_INITIALIZATION;

Util.report("Failed to load class \"org.slf4j.impl.StaticLoggerBinder\".");

Util.report("Defaulting to no-operation (NOP) logger implementation");

Util.report("See " + NO\_STATICLOGGERBINDER\_URL

+ " for further details.");

} else {

failedBinding(ncde);

throw ncde;

}

} catch (java.lang.NoSuchMethodError nsme) {

String msg = nsme.getMessage();

if (msg != null && msg.indexOf("org.slf4j.impl.StaticLoggerBinder.getSingleton()") != -1) {

INITIALIZATION\_STATE = FAILED\_INITIALIZATION;

Util.report("slf4j-api 1.6.x (or later) is incompatible with this binding.");

Util.report("Your binding is version 1.5.5 or earlier.");

Util.report("Upgrade your binding to version 1.6.x.");

}

throw nsme;

} catch (Exception e) {

failedBinding(e);

throw new IllegalStateException("Unexpected initialization failure", e);

}

}

* 跟进findPossibleStaticLoggerBinderPathSet方法

// We need to use the name of the StaticLoggerBinder class, but we can't reference

// the class itself.

private static String STATIC\_LOGGER\_BINDER\_PATH = "org/slf4j/impl/StaticLoggerBinder.class";

// 从classpath找出所有slf4j的实现，并记录下它们的资源路径

private static Set<URL> findPossibleStaticLoggerBinderPathSet() {

// use Set instead of list in order to deal with bug #138

// LinkedHashSet appropriate here because it preserves insertion order during iteration

// 用LinkedHashSet能够保证插入的顺序

Set<URL> staticLoggerBinderPathSet = new LinkedHashSet<URL>();

try {

ClassLoader loggerFactoryClassLoader = LoggerFactory.class

.getClassLoader();

Enumeration<URL> paths;

if (loggerFactoryClassLoader == null) {

paths = ClassLoader.getSystemResources(STATIC\_LOGGER\_BINDER\_PATH);

} else {

paths = loggerFactoryClassLoader

.getResources(STATIC\_LOGGER\_BINDER\_PATH);

}

while (paths.hasMoreElements()) {

// path的值 jar:file:/D:/repository/ch/qos/logback/logback-classic/1.1.7/

//logback-classic-1.1.7.jar!/org/slf4j/impl/StaticLoggerBinder.class

URL path = (URL) paths.nextElement();

staticLoggerBinderPathSet.add(path);

}

} catch (IOException ioe) {

Util.report("Error getting resources from path", ioe);

}

return staticLoggerBinderPathSet;

}

### **4.3.2 org.apache.commons:commons-lang3**

#### 4.3.2.1 概述

Apache Commons Lang 库提供了标准Java库函数里所没有提供的Java核心类的操作方法。 Apache Commons Lang 为java.lang API提供了大量的辅助工具，尤其是在String操作方法，基础数值方法，对象引用，并发行，创建及序列化，系统属性方面。

Lang3.0及其后续版本使用的包名为 org.apache.commons.lang3 ，而之前的版本为 org.apache.commons.lang ，允许其在被使用的同时作为一个较早的版本。

Apache Commons Lang 3.3 API 包列表：

org.apache.commons.lang3

org.apache.commons.lang3.builder

org.apache.commons.lang3.concurrent

org.apache.commons.lang3.event

org.apache.commons.lang3.exception

org.apache.commons.lang3.math

org.apache.commons.lang3.mutable

org.apache.commons.lang3.reflect

org.apache.commons.lang3.text

org.apache.commons.lang3.text.translate

org.apache.commons.lang3.time

org.apache.commons.lang3.tuple

#### 4.3.2.2 使用说明

##### **4.3.2.2.1 org.apache.commons.lang3**

此包提供了高度可重用静态的工具方法，主要是对 java.lang 类的一些补充。

由于此包中方法绝大多数都为静态的，因此不需要创建实例化相应的对象，而是通过类名直接调用需要的方法。

ArrayUtils 是一个对数组进行特殊处理的类。当然 jdk 中的 Arrays 是有一些功能的， Array 也提供了一些动态访问 java 数组的方法，这里的 ArrayUtils 扩展提供了更多的功能。

其他的一些类的用途：

AnnotationUtils 用于辅助处理注释实例。

CharSetUtils 用于操作字符集实例。

CharUtils 用于操作字符基本类型及字符类对象。

StringUtils 用于实现对字符串的操作，处理null输入。

##### **4.3.2.2.2 org.apache.commons.lang3.builder**

辅助实现 equals(Object) ， toString() ， hashCode() , 和 compareTo(Object) 方法， 在这个包里面一共有7个类：

CompareToBuilder : 用于辅助实现 Comparable.compareTo(Object) 方法；

EqualsBuilder : 用于辅助实现 Object.equals(Object) 方法；

HashCodeBuilder : 用于辅助实现 Object.hashCode() 方法；

ToStringBuilder : 用于辅助实现 Object.toString() 方法；

ReflectionToStringBuilder : 使用反射机制辅助实现 Object.toString() 方法；

ToStringStyle : 辅助 ToStringBuilder 控制输出格式；

StandardToStringStyle : 辅助 ToStringBuilder 控制标准格式。

在我们的日常编码过程当中，经常会使用到比较两个对象是否相等、比较大小、取hash、获取对象信息等。但是在实现这些方法的具体代码当中，稍微有点不注意就会出现一些BUG，而且有些往往还非常难以发现，因此 org.apache.commons.lang3.builder中提供的这些用于辅助实现上述功能的方法就比较好了，有了这些类，就可以更好、更快、更方便的实现上述方法。

##### **4.3.2.2.3 org.apache.commons.lang3.time**

用于提供操作时间（Date）和日期（Duration）的方法和类，在这个包里面一共有7个类：

DateFormatUtils ： 提供格式化日期和时间的功能及相关常量，

DateUtils ： 在Calendar和Date的基础上提供更方便的访问，

DurationFormatUtils ： 提供格式化时间跨度的功能及相关常量，

FastDateFormat ： 为java.text.SimpleDateFormat提供一个的线程安全的替代类，

FastDateParser ： 为java.text.SimpleDateFormat提供一个的线程安全的替代类，

FastDatePrinter ： 为java.text.SimpleDateFormat提供一个的线程安全的替代类，

StopWatch ： 提供一套方便的计时器的API。

这些包除了 StopWatch ，其他的因为都是不可变的，所以是\_\_线程安全\_\_的，此包包含了一些操作时间的基础工具。更 ApacheCommons Lang 中的其他的大部分类一样，此包中的方法基本均为 static 方法，应该\_\_直接使用类名调用\_\_相应的方法予以实现相应的功能。

### **4.3.3 org.projectlombok:lombok**

#### 4.3.3.1 概述

ombok是一个可以通过简单的注解的形式来帮助我们简化消除一些必须有但显得很臃肿的 Java 代码的工具，简单来说，比如我们新建了一个类，然后在其中写了几个字段，然后通常情况下我们需要手动去建立getter和setter方法啊，构造函数啊之类的，lombok的作用就是为了省去我们手动创建这些代码的麻烦，它能够在我们编译源码的时候自动帮我们生成这些方法。

lombok能够达到的效果就是在源码中不需要写一些通用的方法，但是在编译生成的字节码文件中会帮我们生成这些方法，这就是lombok的神奇作用。

虽然有人可能会说IDE里面都自带自动生成这些方法的功能，但是使用lombok会使你的代码看起来更加简洁，写起来也更加方便。

#### 4.3.3.2 maven安装

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.16.16</version>

</dependency>

#### 4.3.3.3 使用样例

##### @NonNull：修饰函数参数，判空减少函数里面参数验证代码

**使用lombok**

import lombok.NonNull;

public class NonNullExample extends Something {

private String name;

public NonNullExample(@NonNull Person person) {

super("Hello");

this.name = person.getName();

}

}

**不使用lombok**

public class NonNullExample extends Something {

private String name;

public NonNullExample(@NonNull Person person) {

super("Hello");

if (person == null) {

throw new NullPointerException("person");

}

this.name = person.getName();

}

}

##### @Cleanup：修饰对有close函数变量，在函数结束时自动调用

##### @Getter/@Setter: 自动生成Getter/Setter方法

##### @NoArgsConstructor: 自动生成无参数构造函数

##### @AllArgsConstructor: 自动生成全参数构造函数

##### @...

### **4.3.4 commons-beanutils**

#### 4.3.4.1 概述

是Apache开源组织提供的用于操作JAVA BEAN的工具包。使用commons-beanutils，我们可以很方便的对bean对象的属性进行操作。今天为大家介绍一下该包的常用方法。

#### 4.3.4.2 maven安装

<dependency>

<groupId>commons-beanutils</groupId>

<artifactId>commons-beanutils</artifactId>

<version>1.9.3</version>

</dependency>

#### 4.3.4.3 使用样例

##### MethodUtils:通过反射访问对象的方法并且执行方法。

public void test() throws Exception

{

UserInfo userInfo = new UserInfo();

// 通过方法名和参数类型获得可访问方法

Method method = MethodUtils.getAccessibleMethod(UserInfo.class,"setUserId", String.class);

method.invoke(userInfo, "jianggujin");

// 可以直接通过invokeMethod执行方法

MethodUtils.invokeMethod(userInfo, "setUserName", "蒋固金");

System.out.println(userInfo);

}

##### ConstructorUtils:通过反射提供了构造方法相关的便捷操作方法

public void test() throws Exception

{

Constructor<UserInfo> constructor = ConstructorUtils.getAccessibleConstructor(UserInfo.class, new Class[] {String.class, String.class });

System.out.println(constructor.newInstance("jianggujin", "蒋固金"));

// 更简洁的写法

UserInfo userInfo = ConstructorUtils.invokeConstructor(UserInfo.class,new String[] { "jianggujin", "蒋固金" });

System.out.println(userInfo);

}

##### PropertyUtils:通过反射提供了对象属性的便捷操作方法

public void test() throws Exception

{

UserInfo userInfo = new UserInfo("jianggujin", "蒋固金");

UserInfo copyed = new UserInfo();

PropertyUtils.copyProperties(copyed, userInfo);

System.out.println(copyed);

System.out.println(PropertyUtils.describe(userInfo));

PropertyUtils.setProperty(userInfo, "userId", "gjjiang");

System.out.println(userInfo);

}

##### BeanUtils:通过反射提供了Bean对象的便捷操作方法

public void test() throws Exception

{

UserInfo userInfo = new UserInfo();

HashMap<String, String> properties = new HashMap<String, String>();

properties.put("userId", "jianggujin");

properties.put("userName", "蒋固金");

BeanUtils.populate(userInfo, properties);

System.out.println(userInfo);

}

##### ConvertUtils:提供了数据类型相互转换的方法

public void test() throws Exception

{

ConvertUtils.register(new Converter()

{

@Override

public <T> T convert(Class<T> arg0, Object arg1)

{

try

{

return (T) new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse((String) arg1);

}

catch (ParseException e)

{

return null;

}

}

}, Date.class);

System.out.println(ConvertUtils.convert("2016-04-09 12:41:00", Date.class));

}

### **4.3.5 org.mybatis:mybatis**

#### 4.3.5.1 概述

MyBatis 本是[apache](https://baike.baidu.com/item/apache" \t "https://baike.baidu.com/item/MyBatis/_blank)的一个开源项目[iBatis](https://baike.baidu.com/item/iBatis" \t "https://baike.baidu.com/item/MyBatis/_blank), 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis 。2013年11月迁移到Github。

iBATIS一词来源于“internet”和“abatis”的组合，是一个基于Java的[持久层](https://baike.baidu.com/item/%E6%8C%81%E4%B9%85%E5%B1%82" \t "https://baike.baidu.com/item/MyBatis/_blank)框架。iBATIS提供的持久层框架包括SQL Maps和Data Access Objects（DAOs）

#### 4.3.5.2 maven安装

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.4.4</version>

</dependency>

#### 4.3.5.3 使用过程（百度）

##### 两种配置文件

第一种:mybatis的配置文件: mybatis-config.xml,其中包括数据库连接信息，类型别名等等   
         特点: 名字一般是固定的、位置是src下面

示例

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

           <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

           "http://mybatis.org/dtd/mybatis-3-config.dtd">

           <configuration>

             <typeAliases>

               <typeAlias alias="Student" type="com.briup.pojo.Student" />

             </typeAliases>

             <environments default="development">

               <environment id="development">

                 <transaction Manager type="JDBC" />

                  <dataSource type="POOLED">

                   <!--

                   <property name="driver" value="com.mysql.jdbc.Driver" />

                   <property name="url" value="jdbc:mysql://localhost:3306/test" />

                   -->

                   <property name="driver" value="oracle.jdbc.driver.OracleDriver" />

                   <property name="url" value="jdbc:oracle:thin:@127.0.0.1:1521:XE" />

                   <property name="username" value="test" />

                   <property name="password" value="test" />

                 </dataSource>

               </environment>

             </environments>

             <mappers>

               <mapper resource="com/briup/pojo/StudentMapper.xml" />

             </mappers>

           </configuration>

第二种:mybatis的映射文件:XxxxxMapper.xml,这个xml文件中包括Xxxx类所对应的数据库表的各种增删改查sql语句   
            特点:   
                名字一般为XxxxMapper.xml,Xxxx是对应类的名字   
                位置不固定,一般放到一个专门的package里面   
            实例:

<?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="com.briup.pojo.StudentMapper">

           <resultMap type="Student" id="StudentResult">

             <id property="id" column="id" />

             <result property="name" column="name" />

             <result property="email" column="email" />

           </resultMap>

           <select id="findAllStudents" resultMap="StudentResult">

                 SELECT \* FROM STUDENTS

           </select>

           <select id="findStudentById" parameterType="int" resultType="Student">

              SELECT ID AS STUDID, NAME, EMAIL

                  FROM STUDENTS WHERE ID=#{Id}

           </select>

           <insert id="insertStudent" parameterType="Student">

              INSERT INTO STUDENTS(ID,NAME,EMAIL)

                  VALUES(#{id },#{name},#{email})

           </insert>

         </mapper>

#### 4.3.5.4 项目使用过程

##### 第一步：定义与数据表对应模型

**import** org.senseframework.support.orm.mybatis.annotations.Column;  
**import** org.senseframework.support.orm.mybatis.annotations.Table;

@Table(table = **"log\_auth"**)  
@Data  
**public class** LogAuth {  
 @Column  
 **private** String **id**;  
 @Column(column = **"service\_id"**)  
 **private** String **serviceId**;  
 @Column(column = **"operate\_id"**)  
 **private** String **operateId**;  
 @Column  
 **private** String **caller**;

}

##### 第二步：基于模型定义与数据表对应Mapper

**import** com.iflytek.musicsearch.vspp.model.LogAuth;  
**import** org.apache.ibatis.annotations.Mapper;  
**import** org.senseframework.support.orm.mybatis.mysql.MysqlMapper;

@Mapper  
**public interface** LogAuthMapper **extends** MysqlMapper<LogAuth> {  
}

##### 第三步：基于Mapper定义DAO对象

**import** org.senseframework.support.orm.mybatis.mysql.AbstractMySqlDao;

**public abstract class** BasicDao<T> **extends** AbstractMySqlDao<T> {  
}

**public class** LogAuthDao **extends** BasicDao<LogAuth>{  
 @Override  
 **protected** Class mapperClass() {  
 **return** LogAuthMapper.**class**;  
 }  
}

### **4.3.6 Google Guava（google出版的Java基础库）**

#### 4.3.6.1 概述

什么是Guava？Google Guava源于2007年的"Google Collections Library"，提供使用Java集合的工具，已经进化为Java开发者的基础工具箱。Guava拥有对很多人有用的一些东西。如与字符串、集合、并发、I/O和反射的协作的类。Function接口提供了Java函数式编程的能力，并且极大地简化了代码。Suppier接口帮助creational pattern（23种设计模式之一）。但是，Guava不仅仅是从Java中提取的一些样板的抽象化，或者一些我们都认为在Java中自带的便利方法。Guava涉及编写良好的代码，促使代码更有弹性、更加简洁。所有建议不仅仅是使用Guava，更要读一下它的源码，学习运行的原理。然后将同样的准则用于自己所写的代码之中。最后，祝大家学的愉快！

#### 4.3.6.2 maven安装

<dependency>

<groupId>com.google.guava</groupId>

<artifactId>guava</artifactId>

<version>12.0.1</version>

</dependency>

#### 4.3.6.3 使用

##### 第一季，介绍Guava，此外给出使用Guava的例子；

##### 第二季，基本的Guava工具，包括使用字符串和对象的基本功能；

##### 第三季，Guava函数式变成，介绍Guava提供的函数式编程特性；

##### 第四季，使用集合，包括增强现有Java集合功能的工具和类；

##### 第五季，并发，展示了如何通过使用Guava的并发抽象化帮助我们更好的编写并发代码；

##### 第六季，Guava缓存，介绍Guava缓存，包括一个自加载缓存；

##### 第七季，EventBus类，通过它完成基于事件的编程；

##### 第八季，使用文件类，展示了Guava如何极大地简化文件的读写；

##### 第九季，零碎工作覆盖了规避null的Optional类，哈希函数，BloomFilter数据结构。

### **4.3.7 org.apache.avro（序列化库）**

### **4.3.8 JPA（Java Persistence API）**

#### 4.3.8.1 概述

JPA是Java Persistence API的简称，中文名Java持久层API，是JDK 5.0注解或XML描述对象－关系表的映射关系，并将运行期的实体[对象持久化](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1%E6%8C%81%E4%B9%85%E5%8C%96" \t "https://baike.baidu.com/item/JPA/_blank)到数据库中。

Sun引入新的JPA ORM规范出于两个原因：其一，简化现有Java EE和Java SE应用开发工作；其二，Sun希望整合ORM技术，实现天下归一。

#### 4.3.8.2 maven安装

<dependency>

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

<artifactId>hibernate-jpa-2.1-api</artifactId>

<version>1.0.0.Final</version>

</dependency>

### **4.3.9 com.alibaba:druid(数据库连接池)**

#### 4.3.9.1 概述

Druid是Java语言中最好的数据库连接池。Druid能够提供强大的监控和扩展功能。

#### 4.3.9.2 maven安装

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.0.29</version>

</dependency>

# 第五章 周边设施

## **5.1 MariaDB**

MySQL之父Widenius先生离开了Sun之后，觉得依靠Sun/Oracle来发展MySQL，实在很不靠谱，于是决定从新开发代码全部开源免费关系型数据库，这就是MariaDB。在navicat中操作mariaDB的界面和提示符还是mysql端口也是3306 外人看起来除了数据库名字改了其他和mysql完全一致，但是看一下性能比较，最近在美国mariaDB很火。我觉得oracle是不是故意把mysql做的很差？

mariaDB在各方面都是mysql创新和提高版本 而不是简单的替代品