# Spring的基本用法

Spring没有太多的新的东西，它只是抽象了大量的JavaEE应用中的常用代码，将它们抽象成一个框架，通过使用Spring可以大幅度地提高开发效率，并可以保证整个应用具有良好的设计。

Spring框架号称JavaEE应用的一站式解决方案，Spring本身提供了一个设计优良的MVC框架：Spring MVC，使用Spring框架则可以直接使用该MVC框架。但是Spring却没有提供完整的持久层框架——这可以理解成一种“空”，但这种“空”反而是Spring框架的魅力所在。Spring能与大部分持久层框架无缝整合：Hibernate、JPA、MyBatis、甚至直接使用JDBC，随便我们喜欢，无论哪种持久层框架，Spring都会为我们提供无缝的整合以及极好的简化。

从这个意义上看，Spring更像一种中间层容器，Spring向上可以与MVC框架无缝整合，向下可以与各种持久层框架无缝整合，具有强大的生命力。由于Spring框架的特殊地位，所以轻量级JavaEE应用通常都会使用Spring。实际上，轻量级JavaEE这个概念也是由Spring框架衍生出来的，Spring框架暂时都没有较好的替代框架。

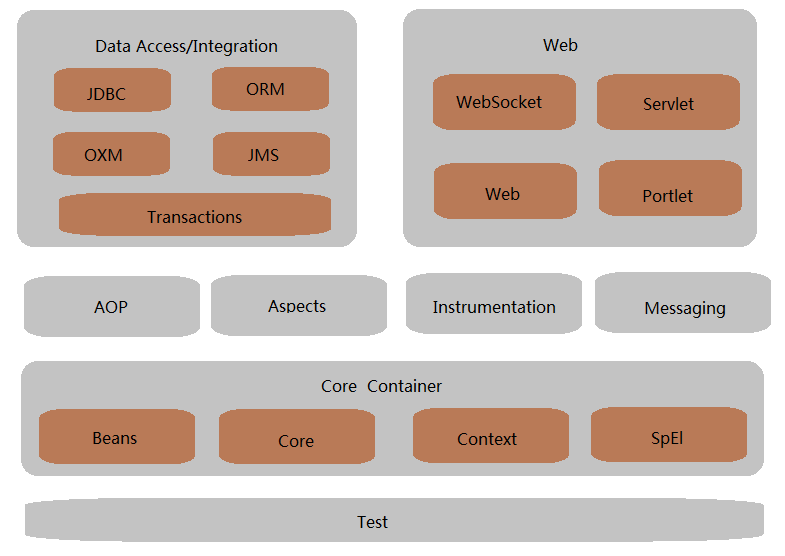
## Spring简介

Spring是一个从实际开发中抽取出来的框架，因此它完成了大量开发中的通用步骤，留给开发者的仅仅是与特定应用相关的部分，从而大大提供了企业级应用的开发效率。

Spring为企业应用的开发提供了一个轻量级的解决方案。该解决方案包括：基于依赖注入的核心机制、基于AOP（Aspect Oriented Programming，面向侧面的程序设计）的声明式事务管理、与各种持久层技术的整合，以及优秀的Web MVC框架等。Spring致力于JavaEE应用各层的解决方案，而不是仅仅专注于某一层的方案。可以说：Spring是企业应用开发的“一站式”选择，Spring贯穿表现层、业务层、持久层。然而，Spring并不想取代那些已有的框架，而是以高度的开发性与它们无缝整合。

总结起来，Spring具有如下优点：

* 低侵入式设计，代码的污染极低
* 独立于各种服务器，基于Spring框架的应用，可以真正实现Write One，Run AngWhere的承诺。
* Spring的IoC容器降低了业务对象替换的复杂性，提高了组件之间的解耦。
* Spring的AOP支持允许将一些通用任务如安全、事务、日志等进行集中处理，从而提供了更好的复用。
* Spring的ORM（Object Relation Mapping，对象关系映射）和DAO（Data Access Object，数据访问对象）提供了与第三方持久层框架的良好整合，并简化了底层的数据库访问。
* Spring的高度开放性，并不强制应用完全依赖Spring，开发者可以自由选用Spring框架的部分或全部。



如上图所示，当使用Spring框架时，必须使用Spring Core Container（即Spring容器），它代表了Spring框架的核心机制，Spring Core Container主要由org.springframework.core、org.springframework.beans、org.springframe

work.context和org.springframework.expression四个包及其子包组成，主要提供Spring IoC容器支持。其中org.spring

framework.expression及其子包是Spring3.0新增的，它提供了Spring Expression Language支持。

## Spring入门

所讲内容基于版本4.0.4版本，使用时也下载该版本。最新版本为：4.3.3。

### Spring的下载与安装

Spring是一个独立的框架，它不需要依赖于任何Web服务器或容器，它既可在独立的JavaSE项目中使用，当然也可以在Java Web项目中使用。下面先介绍如何为Java项目和Java Web项目添加Spring支持。

1. 登录<http://repo.springsource.org/libs-release-local/>站点，该页面显示一个目录列表，读者沿着org->spring

framework->spring路径进入，即可看到Spring框架各个版本的压缩包的下载链接。下载Spring的最新稳定版：4.3.3。

1. 下载完毕，得到一个spring-framework-4.3.3.RELEASE-dist.zip压缩文件，解压该压缩文件得到一个名spring-

framework-4.3.3.RELEASE的文件夹，该文件夹下有如下几个子文件夹。

* docs：该文件夹下存放Spring的相关文档，包含开发指南、API参考文档。
* libs：该目录下的JAR包分为三类——1、Spring框架class文件的JAR包；2、Spring框架源文件的压缩包，文件名以-sources结尾；3、Spring框架API文档的压缩包，文件-javadoc结尾。整个Spring框架由21个模块组成，该目录下将看到Spring为每个模块都提供了三个压缩包。
* schemas：该目录下包含了Spring各种配置文件的XML Schema文档。
* readme.txt、notice.txt、license.txt等说明性文档。

1. 将libs目录下所需要的class文件的JAR包复制添加到项目的类加载路径中——既可通过添加环境变量的

方式来添加，也可以使用Ant或IDE工具来管理应用程序的类加载路径。如果需要发布该应用，则将这些JAR包一同发布既可。如果没有太多要求，建议将libs目录下所有的class文件的JAR包添加进去。

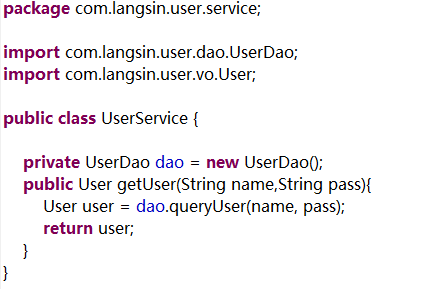
4、除此之外，Spring核心容器必须依赖common-logging的JAR包，因此读者还应该登录<http://commons.apache.org/>站点下载最新的commons-logging工具，下载完成得到一个commons-logging-1.1.3-bin.zip压缩文件，将该文件解压路径下的common-logging-1.1.3.jar也添加到项目的类加载路径中。

## Spring的核心机制—依赖注入

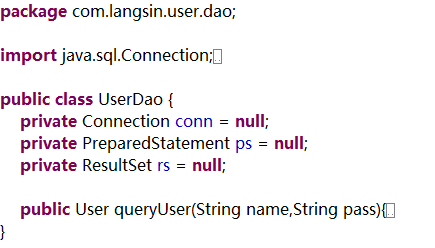
在原始的软件开发过程中，虽然使用了MVC的设计理念，实现了代码的逻辑分层。但是不同的逻辑层之间却是以硬编码的方式组合在一起。软件开发完成后，在实际使用的过程中，软件本身所存在的一些问题也随之暴漏出来。如果需要对软件进行维护升级，工作量往往是非常复杂的。如下代码所示：

示例1：用户注册模块

Service层：

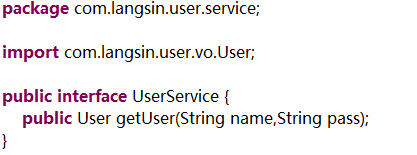


Dao层：

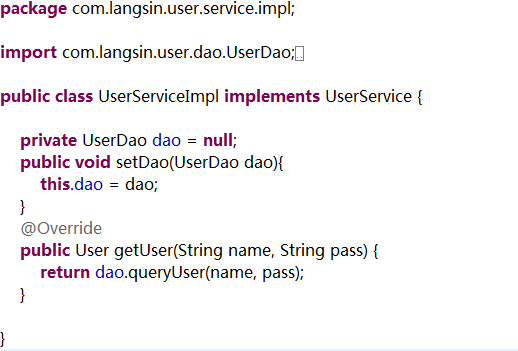


以用户注册为例，软件升级所有的Service层与Dao层的代码组合出全部需要进行改动，工作量势必很大，而如果将软件结构更改如下结构，为Service层和Dao层分别指定接口和创建接口的实现类。

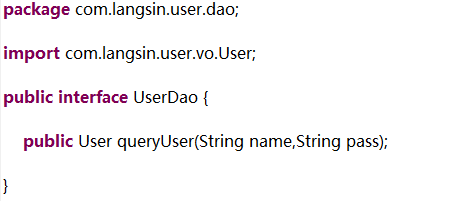
1、创建Service层的接口



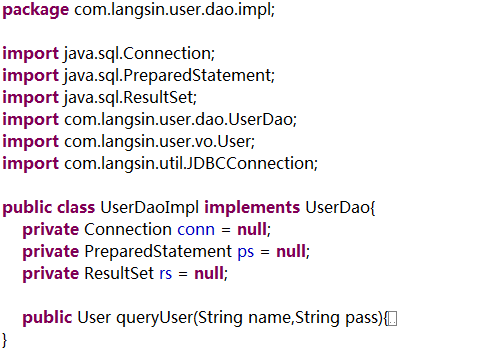
2、创建Service接口的实现类



3、创建Dao接口

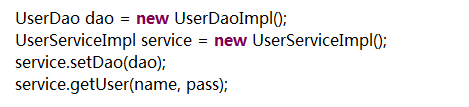


4、创建Dao接口的实现类



如上代码所示，Service调用Dao层时就需要传入一个Dao层的具体的实现类。

测试代码如下：



虽然dao仍然需要创建，但是在结构上却发生了根本的变化，即：

对象由原来程序本身创建，变为了程序接收对象。

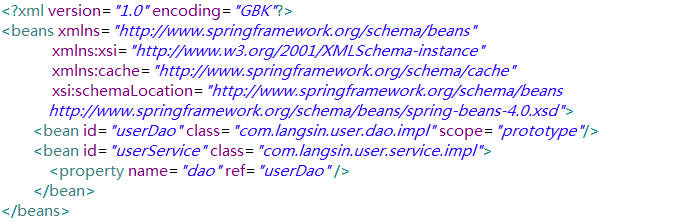
程序员的主要精力放在业务层的实现上。

实现了service层和dao层的解耦工作，service层和dao层实现了分离，没有了直接的依赖关系。

如果dao层发生了业务的改变，原有的程序不需要发生任何改变，只需要将不同的dao实现类传给Service。

Spring框架则完成了完全的解耦，在对象创建时将dao的传入更改为了配置形式。如下所示：

1、Spring的配置文件如下



代码如下：

ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:test.xml");

UserService service = ctx.getBean("userService",UserService.class);

service.getUser(name,pass);

正如前面代码所看到的，程序代码并没有主动为UserService对象的dao成员变量赋值，但执行UserService对象getUser方法时，完全可以正常访问到dao对象，并可以调用dao对象的getUser方法。

由此看见，UserService对象的dao成员变量并不是程序主动设置的，而是由Spring容器负责设置的，程序人员只需要为dao成员变量提供一个setter方法即可，并通过property元素驱动Spring容器用该setter方法完成成员变量的赋值操作。

纵观所有的Java应用，从基于JavaApplication的小应用到多层结构的企业级应用，这些应用中大量存在A对象调用B对象方法的情形，这种情形被Spring称为依赖，即A对象依赖B对象。对于Java应用而言，它们总是由一些互相调用的对象构成的，Spring把这种互相调用的关系称为依赖关系。假如A组件调用B组件的方法，即可称A组件依赖B组件。

Spring框架的核心功能有两个。

* Spring容器作为超级大工厂，负责创建、管理所有的Java对象，这些Java对象被称为Bean。
* Spring容器管理容器中Bean之间的依赖关系，Spring使用一种被称为“依赖注入”的方式来管理Bean之间的依赖关系。

使用依赖注入，不仅可以为Bean注入普通的属性值，还可以注入其他Bean的引用。通过这种依赖注入，JavaEE应用中的各种组件不需要以硬编码方式耦合在一起，甚至无需使用工厂模式。依赖注入达到的效果，非常类似于传说中的“共产主义”，当某个Java实例需要其他Java实例时，系统自动提供所需要的实例，无须程序显示获取。

依赖注入是一种优秀的解耦方式。依赖注入让Spring的Bean以配置文件组织在一起，而不是以硬编码的方式耦合在一起。

### 理解依赖注入

虽然Spring并不是依赖注入的首创者，但是罗德—约翰逊是第一个高度重视以配置文件来管理Java实例的协作关系的人，它给这种方式起了一个名字：控制反转（Inversion of Control，IoC）。在后来的日子里，马丁—富勒(Martine Fowler)为这种方式起了另外一个名称：依赖注入（Dependency Injection）。

因此，不管是依赖注入，还是控制反转，其含义完全相同。当某个Java对象需要调用另外一个Java对象的方法时，在传统模式下通常有如下两种做法。

* 原始做法：调用者主动创建被依赖对象，然后再调用被依赖对象的方法。
* 简单工厂模式：调用者先找到被依赖对象的工厂，然后主动通过工厂去获取被依赖的对象，最后在调用依赖对象的方法。

对于第一种方式，由于调用者需要通过形如“new 构造器()”的代码创建对象，因此必然导致调用者与被依赖对象实现类的硬编码耦合，非常不利于项目的升级的维护。

对于简单工厂的方式，大致需要把握三点：1、调用者面向被依赖对象的接口编程；2、将被依赖的对象的创建交给工厂完成；3、调用者通过工厂来获取被依赖的组件。通过这3点改造，可以保证调用者只需要与被依赖对象的接口耦合，这就避免了类层次的硬编码耦合。这种方式唯一的缺点是，调用组件需要主动通过工厂去获取依赖对象，就会带来调用组件与被依赖对象工厂的耦合。

使用Spring框架之后，调用者无须主动获取被依赖对象，调用者只要被动接受Spring容器为调用者的成员变量赋值即可。由此可见，使用Spring框架之后，调用者获取被依赖对象的方式由原来的主动获取，变成了被动接受，于是就有了罗德—约翰逊称为的控制反转。

从Spring容器的角度来看，Spring容器负责将被依赖对象赋值给调用者的成员变量——相当于为调用者注入它依赖的实例对象所以又有了马丁—富勒的依赖注入。

正因为Spring将被依赖对象注入给了调用者，所以调用者无须主动获取被依赖对象，只要被动等待Spring容器注入即可。由此可见，控制反转和依赖注入其实是同一个行为的两种表达，只是描述角度不用而已。

由此可见使用Spring框架之后的两个主要改变是：

* 程序无须使用new调用构造器去创建对象。所有的Java对象都可以交给Spring容器去创建。
* 当调用者需要调用被依赖对象的方法时，调用者无须主动获取依赖对象，只要等待Spring容器注入即可。

依赖注入通常有如下两种：

* 设值注入：IoC容器使用成员变量的setter方法来注入被依赖对象。
* 构造注入：IoC容器使用构造器来注入被依赖对象。

### 1.3.2 设值注入

设值注入是指IoC容器通过成员变量的setter方法来注入被依赖对象。这种注入方式简单、直观。因而在Spring的依赖注入里大量被使用。

下面示例将会对前面示例进行改写，让其更加规范。Spring推荐面向接口编程。不管是调用者，还是被依赖对象，都应该为之定义接口，程序应该面向它们的接口，而不是面向实现类编程。这样以便后期的升级、维护。

下面先定义一个Person接口，该接口定义了一个Person应该遵循的规范，下面是Person接口的代码：

public interface Person{

void useAxe();

}

下面是Axe接口的代码：

public interface Axe{

public String shop();

}

Spring推荐面向接口编程，这样可以更好地让规范和实现分离，从而提供更好的解耦。对于一个Java EE应用，不管是dao组件，还是业务逻辑组件，都应该先定义一个接口，该接口定义了该组件应该实现的功能，但功能的实现则由其实现类提供。

下面是Person实现类的代码：

public class Worker implements Person{

private Axe axe;

public void setAxe(Axe axe){

this.axe = axe;

}

public void useAxe(){

System.out.println(axe.chop());

}

}

上面程序中的粗体字代码实现了Person接口的useAxe方法，实现该方法时调用了axe的chop方法，这就是典型的以依赖关系。

回顾曾经所写代码，基本上是A调用B，B调用C，C调用D这种方式，而Spring容器的最大作用就是以松耦合的方式来管理这种调用关系。如上面代码Worker类中，它并不知道axe的实例在哪，也不知道实例是如何实现的，它只是需要调用Axe对象的方法，这个Axe实例对象由Spring负责注入。

下面提供一个Axe的实现类：

public class StoneAxe implements Axe{

public String chop(){

return “石斧头砍柴费劲！”;

}

}

到现在为止，程序依然不知道Worker与哪个Axe类实例耦合，Spring也不知道，Spring需要使用XML配置文件指定实例之间的依赖关系。

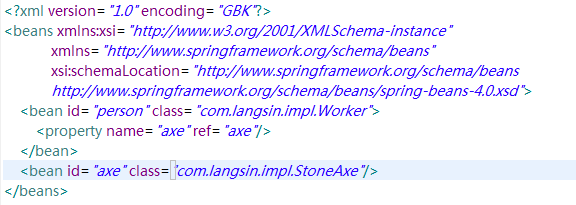
Spring采用XML配置文件，从Spring2.0开始，Spring推荐使用XML Schema来定义配置文件的语义约束。当采用XML Schema来定义配置文件的语义约束时，还可以利用Spring配置文件的扩展型，进一步简化Spring配置。

Spring为基于XML Schema的XML配置文件提供了一些新的标签，这些标签使配置更简单，使用更方便。关于如何使用Spring所提供的新标签，后面会更进一步的介绍。

不仅如此，采用基于XML Schema的XML配置文件时，Spring甚至允许程序员开发自定义的配置文件标签，让其他开发人员在Spring配置文件中使用这些标签，但这些标签通常应该由第三方供应商来完成。对于普通软件开发人员以及普通系统架构师而言，则通常无须开发自定义的Spring配置文件标签。

由于Spring现在已经推荐使用XML Schema作为配置文件的语义约束，因此以下示例全部为使用XML Schema作为配置文件的语义约束。

下面是本示例的配置文件代码：



在配置文件中，Spring配置Bean实例通常会指定两个属性。

* id：指定该Bean的唯一标识，Spring根据id属性值来管理Bean，程序通过id属性值来访问该Bean对象。
* class：指定该Bean的实现类，此处不可再用接口，必须使用实现类，Spring容器会使用XML解析器读取该属性值，并利用反射来创建该实现类的实例。

可以看到Spring管理Bean的灵巧性。Bean与Bean之间依赖关系放在配置文件里组织，而不是写在代码里。通过配置文件的指定，Spring能精确地为每个Bean的成员变量注入值。

Spring会自动检测每个bean定义里的property元素定义，Spring会在调用默认的构造器创建Bean实例之后，立即调用对应setter方法为Bean的成员变量注入值。

每个Bean的id属性是该Bean的唯一标识，程序通过id属性访问Bean，Spring容器也通过Bean的id属性值管理Bean与Bean之间的依赖。

如下代码所示：

package com.langsin.test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.langsin.impl.Person;

public class Test {

public static void main(String[] args) {

**ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");**

**Person per = ctx.getBean("person",Person.class);**

per.useAxe();

}

}

上面程序中的两行粗体字代码实现了创建Spring容器，并通过Spring容器来获取Bean实例。从上面程序中可以看出，Spring容器就是一个巨大的工厂，它可以“生产”出所有类型的Bean实例。程序获取Bean实例的方式是getBean(id属性值)。

主程序调用Person的useAxe方法时，该方法的方法体内需要使用Axe实例，但程序没有任何地方将特定的Person实例和Axe实例耦合在一起。程序没有为Person实例传入Axe实例，Axe实例由Spring在运行期间注入。

Person实例既不需要了解Axe实例的实现类，也无须了解Axe的创建过程。Spring容器根据配置文件的指定，创建Person实例时，不仅创建了Person对象，并为该对象注入它所依赖的Axe实例。

通过这种编码方式，对于以后应用的升级维护都带来了非常大的便利。此时，Person接口、Worker实现类代码无须任何的改变，只需要给出Axe的另一个实现类，修改一下配置文件即可。

package com.langsin.impl;

public class SteelAxe implements Axe {

@Override

public String chop() {

return "钢铁斧头砍柴很快！";

}

}

修改配置文件如下：

<bean id=”axe” class=”com.langsin.impl.SteelAxe” />

程序的再次运行效果如下：



从上面这种切换可以看出，因为Worker实例与具体的Axe的实现类没有任何关系，worker实例仅仅与Axe接口耦合，这就保证了Worker实例与Axe实例之间的松耦合——这也是Spring强调面向接口编程的原因。

Bean与Bean之间的依赖关系由Spring管理，Spring采用setter方法为目标Bean注入所依赖的Bean，这种方式被称为设值注入。

从上面程序示例可以看出，依赖注入以配置文件管理Bean实例之间的耦合，让Bean实例之间的耦合从代码层次分离出来。依赖注入是一种非常优秀的解耦方式。

经过上面的介绍，使用Spring IoC容器的三个基本要点。

1. 应用程序的各个组件面向接口编程，面向接口编程可以将组件之间的耦合关系提升到接口层次，从而有利于项目后期的扩展。
2. 应用程序的各组件不再由程序主动创建，而是由Spring容器来负责产生并初始化。
3. Spring采用配置文件或注解来管理Bean的实现类、依赖关系、Spring容器则根据配置文件或注解，利用反射来创建实例，并为之注入依赖关系。

### 1.3.3 构造注入

前面已经介绍过，通过setter方法为目标Bean注入依赖关系的方式被称为设值注入；另外还有一种注入方式，这种方式在构造实例对象时，已经为其完成依赖关系的初始化，这种通过构造器来完成依赖关系的方式被称为构造注入。

简单来说，就是驱动Spring在底层以反射方式执行带指定参数的构造器，当执行带参数的构造器时，就可以利用构造器参数对成员变量执行初始化——这就是构造注入的本质。

通过前面的介绍，bean元素默认总是驱动Spring调用无参的构造器来创建对象，如果要驱动Spring调用有参数的构造器来创建对象，需要使用constructor-arg子元素，每个constructor-arg子元素代表一个构造参数，如果bean元素包含N个constructor-arg子元素，就会驱动Spring调用带N个参数的构造器来创建对象。

对前面代码中Worker类做简单修改，修改的代码如下：

package com.langsin.impl;

public class Worker implements Person {

private Axe axe;

public Worker(){}

public Worker(Axe axe){

this.axe = axe;

}

@Override

public void useAxe() {

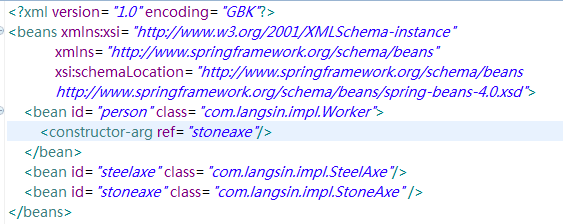
System.out.println(this.axe.chop());

}

}

上面的Worker类没有提供设置axe成员变量的setter方法，仅仅提供了一个带axe参数的构造器，Spring将通过该构造器为Worker注入所依赖的Bean实例。

构造注入的配置文件也需要做简单的修改，为了使用构造注入，还需要使用constructor-arg元素指定构造器的参数，修改后的配置文件如下：



上面配置文件中的粗体代码使用constructor-arg子元素指定了一个构造器参数，该参数类型是Axe，这指定Spring调用Worker类里带一个Axe参数的构造器来创建worker实例。也就是说，上面配置相当于驱动Spring执行如下代码：

String idStr = “person”;

String clazzStr = “com.langsin.impl.Worker”;

String refStr = “stoneaxe”;

Class<?> clazz = Class.forName(clazzStr);

//下面的refClazz代表构造器中参数的class类型，即：container.get(“stoneaxe”).getClass();

Constructor<?> con = clazz.getConstructor(new Class[]{refClazz, refClazz , refClazz ….});

Object person = con.newInstance(new Object[]{container.get(“stoneaxe”)});

container.put(“person”,person);

由于使用了有参数的构造器创建实例，所以当Bean实例被创建完成后，该Bean的依赖关系已经设置完成。该示例的执行效果与设置注入执行的效果完全一样。区别在于：创建Worker实例中的Axe属性的时机不同——设值注入是先通过无参数的构造器创建一个Bean实例，然后调用setter方法注入依赖关系；而构造注入则直接调用有参数的构造器，当Bean实例创建完成后，已经完成了依赖关系的注入。

配置constructor-arg元素时可指定一个index属性，用于指定该构造参数值将作为第几个构造参数值，例如，指定index=”0”表明该构造参数值将作为第一个构造器的参数。



如果在配置文件中通过构造注入来创建Bean，配置代码如下：

<?xml version="1.0" encoding="GBK"?>

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

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

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

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

**<bean id="test" class="com.langsin.test.Test">**

**<constructor-arg value=”hello”/>**

**<constructor-arg value=”23”/>**

**</bean>**

</beans>

当调用含有多个参数的构造器时，配置文件中constructor-age元素又没有指定index属性，那么就按顺序来放入参数，如下代码所示：

Object test = new com.langsin.test.Test(“hello”,”23”);

所以一般情况下，需要按构造器中参数的顺序来写constructor-arg元素，而不需要特别指定index属性。

Spring中构造器参数的类型转换：

由于Spring本身提供了功能强大的类型转换机制，因此如果Test类只包含一个Test(String,int)构造器，那么上面的配置相当于让Spring执行如下代码：

Object test = new com.langsin.test.Test(“hello”,23);

但是，如果Test类既有Test(String,String)，又有Test(String,int)那么上面的配置到底是让Spring执行哪个构造器？答案还是Test(String,String)，因为此时的配置还不够明确，对于<constructor-arg value=”23”/>，Spring只能解析出一个“23”字符串，但它到底需要转换为哪种类型数据——从配置文件中看不出来，只能根据Test类构造器来尝试转换。

为了更明确得指定数据类型，Spring允许为你constructor-arg元素指定一个type属性，例如

**<bean id="test" class="com.langsin.test.Test">**

**<constructor-arg value=”hello” type=”string”/>**

**<constructor-arg value=”23” type=”int”/>**

**</bean>**

此处Spring明确知道此处配置了一个int类型的参数，就会调用Test(String,int)的构造器来创建对象。

## 使用Spring容器

Spring有两个核心接口：BeanFactory和ApplicationContext，其中ApplicationContext是BeanFactory的子接口。它们都可以代表Spring容器，Spring容器是生成Bean实例的工厂，并管理容器中Bean。在基于Spring的Java EE应用中，所有的组件都被当成Bean处理，包括数据源、Hibernate的SessionFactory、事务管理器等。

应用中的所有组件都处于Spring的管理下，都被Spring以Bean的方式管理，Spring负责创建Bean实例，并管理其生命周期。Spring里的Bean都是非常广义的概念，任何的Java对象、Java组件都被当成Bean处理，对于Spring而言，一切Java对象都是Bean。

Java程序面向接口编程，无须关注Bean实例的实现类，但Spring容器负责创建Bean实例。因此在Spring配置文件中必须指定Bean实例的实现类。

### Spring容器

Spring容器最基本的接口就是BeanFactory。BeanFactory负责配置、创建、管理Bean，它有一个子接口：ApplicationContext，因此也被称为Spring上下文。Spring容器还负责管理Bean与Bean之间的依赖关系。

BeanFactory接口包含如下几个基本方法。

* boolean containsBean(String name)：判断Spring容器中是否包含id为name的Bean实例。
* <T> T getBean(Class<T> requiredType)：获取Spring容器中属于required Type类型的、唯一的Bean实例。
* Object getBean(String name)：返回容器中id为name的Bean实例。
* <T> T getBean(String name,Class requiredType)：返回容器中id为name，并且类型为requiredType的Bean。
* Class<?> getType(String name)：返回容器中id为name的Bean的实例的类型。

调用者只需要使用getBean()方法即可获得指定Bean的引用，无须关心Bean的实例化过程。Bean实例的创建、初始化以及依赖关系的注入都由Spring容器完成。

BeanFactory常用的实现类是DefaultListableBeanFactory。

ApplicationContext是Beanfactory的子接口，对于大部分Java EE应用而言，使用它作为Spring容器更方便。其常用实现类是FileSystemXmlApplicationContext、ClassPathXmlApplicationContext和AnnotationConfigApplication

Context。如果在Web应用中使用Spring容器，则通常有XMLWebApplicationContext、AnnocationConfigWebAppli

cationContext两个实现类。

创建Spring容器的实例时，必须提供Spring容器管理的Bean的详细配置信息。Spring的配置信息通常采用XML配置文件来设置，因此，创建BeanFactory实例时，也应该提供XML配置文件作为参数。XML配置文件通常采用Resource对象传入。

大部分JavaEE应用，可在启动Web应用时自动加载ApplicationContext实例，接受Spring管理的Bean无须知道ApplicationContext的存在，一样可以利用ApplicationContext。

对于独立的应用程序，可以通过如下方法来实例化Beanfactory。

//搜索类加载路径下的beans.xml文件创建Resource对象

Resource res = new ClassPathResource(“beans.xml”);

//创建默认的BeanFactory容器

DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();

//让默认的BeanFactory容器加载资源res对应XML配置文件

new XMLBeanDefinitionReader(beanFactory).loadBeanDefinitions(res);

或者采用如下代码来创建BeanFactory：

//搜索文件系统的当前路径下的beans.xml文件创建Resource对象

Resource res = new FileSystemResource(“beans.xml”);

//创建默认的BeanFactory容器

DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();

//让默认的BeanFactory容器加载资源res对应的XML配置文件

new XMLBeanDefinitionReader(beanFactory).loadBeanDefinitions(res);

如果应用需要加载多个配置文件来创建Spring容器，则应该采用BeanFactory的子接口ApplicationContext来创建BeanFactory的实例。ApplicationContext接口包含FileSystemXmlApplicationContext和ClassPathXmlApplication

Context两个常用类。

如果需要同时加载多个XML配置文件来创建Spring容器，则可以采用如下方式：

//以类加载路径下的beans.xml、service.xml文件创建ApplicationContext容器。

ApplicaitonContext context = new ClassPathXmlApplicationContext(“beans.xml”,”service.xml”);

//也可以从文件系统的相对路径或绝对路径来搜索配置文件，如下所示：

ApplicationContext context = new FileSystemXmlApplicationContext(“beans.xml”,”service.xml”);

由于ApplicationContext本身就是BeanFactory的子接口，因此ApplicationContext完全可以作为Spring容器来使用，而且功能更强大。

### 使用ApplicationContext

大部分时候，都不会使用BeanFactory实例作为Spring容器，而是使用ApplicationContext实例作为容器，因此也把Spring容器称为Spring上下文。ApplicationContext是BeanFactory接口的子接口，它增强了BeanFactory的功能。

ApplicationContext允许以声明式方式操作容器，无须手动创建它。可利用如ContextLoader的支持类，在Web应用启动时自动创建ApplicationContext。当然也可采用编程方式创建ApplicationContext。

ApplicationContext除了提供BeanFactory所支持的全部功能外，还有如下额外方法：

* ApplicationContext默认会预初始化所有的singletion Bean，也可通过配置取消预初始化。
* ApplicationContext继承MessageSource接口，因此提供国际化支持。
* 资源访问，比如访问URL和文件。
* 事件机制。
* 同时加载多个配置文件。
* 以声明式方式启动并创建Spring容器。

ApplicationContext包括BeanFactory的全部功能，因此建议优先使用ApplicationContext。除非对于某些内存非常关键的应用才考虑BeanFactory。

当系统创建ApplicationContext容器时，默认会预初始化所有的singletion Bean。也就是说，当Application

Context容器初始化完成后，容器会自动初始化所有的singletion Bean，包括调用构造器创建该Bean的实例。并根据property元素执行setter方法。这意味着：系统前期创建ApplicationContext时将有较大的系统开销，但一旦ApplicationContext初始化完成，程序后面获取singletion Bean实例时将拥有较好的性能。

例如如下配置：

<?xml version="1.0" encoding="GBK"?>

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

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

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

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

<!-- 如果不加任何特殊的配置，该Bean默认是singletion行为的 -->

**<bean id="person" class="com.langsin.impl.Worker">**

**<property name=”test” value=”孙悟空”/>**

**</bean>**

</beans>

上面粗体字代码配置了一个Worker Bean，如果没有任何特殊配置，该Bean就是singleton Bean，ApplicationContext会在容器初始化后，自动调用Worker类的构造器创建Worker Bean，并把“孙悟空”作为参数去调用workerBean的setTest方法。

该程序如下所示：

public class Worker{

public Worker(){

System.out.println(“正在执行Worker无参数的构造器======”);

}

public void setTest(String name){

System.out.println(“调用setTest方法，传入的参数为：”+name);

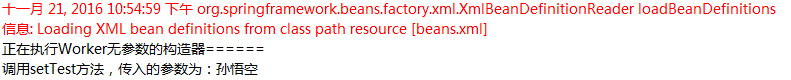
}

}

如果在程序中即使是只写如下一行代码：

ApplicationContext context = new ClassPathXmlApplicationContext(“beans.xml”);

上面代码只是使用ApplicationContext创建了Spring容器，ApplicationContext会自动预初始化容器中的worker Bean——包括调用它的无参构造器，并根据property元素执行setter方法。执行上面代码，可以看出如下效果：



如果将创建Spring容器的代码换成使用BeanFactory作为容器，例如改为如下代码：

Resource res = new ClassPathResource("beans.xml");

DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();

new XmlBeanDefinitionReader(beanFactory).loadBeanDefinitions(res);

上面代码以BeanFactory创建Spring容器，但BeanFactory不会预初始化容器中的Bean，因此执行上面的代码不会看到调用Worker类的构造器、执行worker Bean的setTest方法。



为了阻止Spring容器预初始化容器中的singleton Bean，可以在bean元素指定lazy-init=”true”，该属性用于阻止容器预初始化该Bean。因此，如果为上面bean元素指定了lazy-init=”true”，那么即使是ApplicationContext作为Spring容器，Spring容器也不会预初始化该singleton Bean。

### 1.4.3 ApplicationContext的国际化支持

ApplicationContext接口继承了MessageSource接口，因此具有国际化功能。下面是MessageSource接口中定义的两个用于国际化的方法。

* String getMessage(String code,Object[] args,Locale loc)
* String getMessage(String code,Object[] args,String default,Locale loc)

ApplicationContext正是通过这两个方法来完成国际化的，当程序创建ApplicationContext容器时，Spring自动查找配置文件中名为messageSource的Bean实例，一旦找到这个Bean实例，上述两个方法的调用就被委托给该messageSource Bean。如果没有该Bean，Spring容器会查找其父容器中的messageSource Bean：如果找到，它将被作为messageSource Bean使用。如果无法找到messageSource Bean，系统将会创建一个空的StaticMessageSource Bean，该Bean能接受上述两个方法的调用。

在Spring中配置messageSource Bean时通常使用ResourceBundleMessageSource类。如下配置：

<?xml version="1.0" encoding="GBK"?>

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

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

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

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

**<bean id=”messageSource” class=”org.springframework.context.support.ResourceBundleMessageSource”>**

**<property name=”basenames”>**

**<list>**

**<value>message</value>**

**</list>**

**</property>**

**</bean>**

</beans>

上面文件的粗体字体代码定义了一个messageSource Bean，该Bean实例只指定了一份国际化资源文件，其baseName是message。

然后给出如下两份资源文件，message\_en\_US.properties

hello=welcome,{0}

now=now is:{0}

message\_zh\_CN.properties

hello=欢迎你,{0}

now=现在时间是:{0}

主程序代码如下：

public class SpringTest{

public static void main(String[] args){

ApplicationContext context = new ClassPathXmlApplicationContext(“beans.xml”);

Locale loc = Locale.getDefault();

String hello = context.getMessage(“hello”,new String[]{“孙悟空”},loc);

String time = context.getMessage(“now”,new Object[]{new Date()},loc);

System.out.println(hello);

System.out.println(time);

}

}

上面的程序结果会随着环境不同而改变，在简体中文的环境下，执行结果如下：

欢迎你，孙悟空

现在时间是：当前时间

如果是英文操作系统：

welcome，孙悟空

now is ：当前时间。

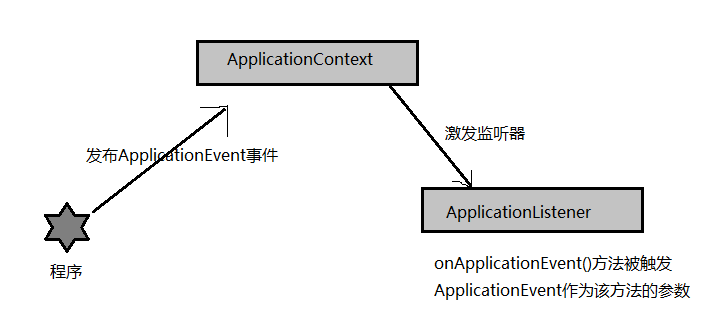
### 1.4.4 ApplicationContext的事件机制

ApplicationContext的事件机制是观察者设计模式的实现，通过ApplicationEvent类和ApplicationListener接口，可以实现ApplicationContext的事件处理。如果容器中有一个ApplicationListener Bean，每当ApplicationContext发布ApplicationEvent时，ApplicationListener Bean将自动被触发。

Spring的事件框架有如下两个重要成员。

* ApplicationEvent：容器事件，必须有ApplicationContext发布。
* ApplicationListener：监听器，可由容器中的任何监听器Bean担任。

实际上，Spring的事件机制与所有的事件机制都基本相似，它们都需要由事件源、事件和事件监听器组成。只是此处的事件源是ApplicationContext，且事件必须由Java程序显示触发。下图为Spring容器的事件机制示意图。



下面程序示范Spring容器的事件机制。程序先定义了一个ApplicationEvent了，其对象就是一个Spring容器事件。ApplicationEvent类的代码如下：

package com.langsin.application;

import org.springframework.context.ApplicationEvent;

public class EmailEvent extends ApplicationEvent {

private String address;

private String text;

public EmailEvent(Object source, String address, String text) {

super(source);

this.address = address;

this.text = text;

}

public EmailEvent(Object source) {

super(source);

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

public String getText() {

return text;

}

public void setText(String text) {

this.text = text;

}

}

上面的EmailEvent类继承了ApplicationEvent，除此之外，它就是一个普通的Java类。只要一个Java类继承了ApplicationEvent基类，那该对象就可作为Spring容器的容器事件。

容器事件的监听器类必须实现ApplicationListener接口，实现接口必须实现如下方法：

package com.langsin.application;

import org.springframework.context.ApplicationEvent;

import org.springframework.context.ApplicationListener;

public class EmailNotifier implements ApplicationListener {

public void onApplicationEvent(ApplicationEvent evt) {

if(evt instanceof EmailEvent){

EmailEvent emailEvent = (EmailEvent)evt;

System.out.println("需要发送邮件的接收地址："+emailEvent.getAddress());

System.out.println("需要发送邮件的正文："+emailEvent.getText());

}else{

System.out.println("其他事件："+evt);

}

}

}

将监听器配置在容器中，配置文件如下：

<?xml version="1.0" encoding="GBK"?>

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

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

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

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

**<bean class="com.langsin.application.EmailNotifier"/>**

</beans>

从上面的粗体字代码可以看出，为Spring容器注册事件监听器，不需要像AWT编程那样采用代码进行编程，只要进行简单的配置即可。只要在Spring中配置一个实现了ApplicationListener接口的Bean。Spring容器就会把这个Bean当成容器事件的事件监听器。

当系统创建Spring容器、加载Spring容器时会自动触发容器事件，容器事件监听器可以监听到这个事件。除此之外，程序也可以调用ApplicationContext的publishEvent()方法来主动触发容器事件。如下主程序使用App

licationContext的publishEvent()来触发事件。

package com.langsin.application;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class EmailTest {

public static void main(String[] args) {

ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

**EmailEvent event = new EmailEvent("test", "test@163.com", "测试发布消息");**

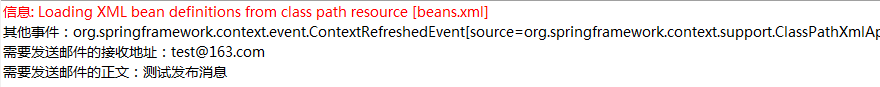
//通过容器对象发布容器事件

**ctx.publishEvent(event);**

}

}

上面程序中的两行粗体代码创建了ApplicationEvent对象，并通过ApplicationContext主动触发了事件，运行上面程序，将看到如下执行结果：



从上面的执行结果可以看出，监听器不仅监听到程序所触发的事件，也监听到容器内置的事件。实际上，如果开发者需要在Spring容器初始化、销毁时回调方法，就可以通过上面的事件监听器来实现。

Spring提供如下几个内置事件。

* ContextRefreshedEvent：ApplicationContext容器初始化或刷新触发该事件。此处的初始化是指，所有的Bean被成功加载，后处理的Bean被检测并激活，所有的singleton Bean被预实例化，ApplicationContext容器已就绪可用。
* ContextStartedEvent：当使用ConfigurableApplicationContext接口的start方法启动ApplicationContext容器时触发该事件。
* ContextClosedEvent：当使用ConfigurableApplicationContext接口的close方法时关闭ApplicationContext容器时触发该事件。
* ContextStoppedEvent：当使用ConfigurableApplicationContext接口的stop方法使ApplicationContext停止时触发该事件。
* RequestHandlerEvent：Web相关的事件，只能应用于使用DispatcherServlet的Web应用中。在使用Spring作为前端的MVC控制器时，当Spring处理用户请求结束后，系统会自动触发该事件。

从Spring4.0.3开始，Spring还新增了SessionConnectedEvent、SessionConnectEvent、SessionDisconnectEvent这三个事件，它们都用于为Spring新增的WebSocket功能服务。

### 1.4.5 让Bean获取Spring容器

前面介绍的示例，都是程序先创建Spring容器，再调用Spring容器的getBean()方法来获取Spring容器中的Bean。在这种访问模式下，程序中总是持有Spring容器的引用。

但是在Web应用中，Spring容器通常采用声明式方式配置产生：开发者只要在web.xml文件中配置一个Listener，该Listener将会负责初始化Spring容器，前端MVC框架可以直接调用Spring容器中的Bean，无须访问Spring容器本身。在这种情况下，容器中的Bean就处于容器管理下，无须主动访问容器，只需接受容器的依赖注入即可。

在某些特殊的情况下，Bean需要实现某个功能，但该功能必须借助于Spring容器才能实现，此时就必须让该Bean先获取Spring容器，然后借助于Spring容器来实现该功能。

为了让Bean获取它所在的Spring容器，可以让该Bean实现BeanFactoryAware接口。BeanFactoryAware接口里只有一个方法。

* setBeanFactory(BeanFactory factory)：该方法有一个参数factory，该参数指向创建它的BeanFactory。

此方法比较特殊，该方法不需要程序员来处理，直接由Spring调用，Spring调用该方法时会将Spring容器作为参数传入该方法。与该接口类似的还有BeanNameAware、ResourceLoaderAware接口，这些接口里都会提供类似的setter方法，这些方法也由Spring负责调用。

与BeanFactoryAware接口类似的有ApplicationContextAware接口，实现该接口的Bean需要实现setApplication

Context(ApplicationContext ctx)：方法——此方法也不是由程序员负责调用，而是由Spring来调用。当Spring容器调用该方法时，它会把自身作为参数传入该方法。

下面示例假设Person类中的sayHi方法必须能输出国际化消息，由于国际化功能需要借助于Spring容器来实现，因此程序就需要让Person类来获取Spring容器对象，让Person类实现ApplicationContextAware接口即可。

示例1：

package com.langsin.impl;

import java.util.Locale;

import org.springframework.beans.BeansException;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ApplicationContextAware;

public class Person implements ApplicationContextAware{

private ApplicationContext ctx = null;

@Override

public void setApplicationContext(ApplicationContext arg0)

throws BeansException {

this.ctx = arg0;

}

public void sayHi(String name){

Locale loc = Locale.getDefault();

String hello = this.ctx.getMessage("hello", new String[]{name},loc);

System.out.println(hello);

}

}

上面的Person类实现了ApplicationContextAware接口，并实现了该接口提供的setApplicationContextAware方法。

Spring容器会检测容器中所有的Bean，如果发现某个Bean实现了ApplicationContextAware接口，Spring容器会在创建该Bean之后，自动调用调用该Bean的setApplicationContextAware方法，会将容器本身作为参数传给该方法，那么在Bean中就可以访问容器本身信息。

将该Bean部署在Spring容器中的XML配置文件如下：

<?xml version="1.0" encoding="GBK"?>

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

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

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

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

<!-- 如果不加任何特殊的配置，该Bean默认是singletion行为的 -->

<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basenames">

<list>

<value>mesg</value>

</list>

</property>

</bean>

<bean id="person" class="com.langsin.impl.Person" />

</beans>

主程序部分进行简单测试，程序先通过实例化的方法来获得ApplicationContext，然后在通过Person Bean来获得BeanFactory，程序如下：

package com.langsin.test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.langsin.impl.Person;

public class Test {

public static void main(String[] args) {

ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

Person person = ctx.getBean("person",Person.class);

person.sayHi("孙悟空");

}

}

## Spring容器中的Bean

从本质上来看，Spring容器就是超级大工厂，Spring容器中的Bean就是该工厂的产品。Spring容器能产生哪些产品，则完全取决于开发者在配置文件中的配置。

对于开发者而言，使用Spring框架主要是做两件事：1、开发Bean。2、配置Bean。对于Spring来说，它要做的就是根据配置文件来创建Bean实例，并调用Bean实例的方法完成“依赖注入”——这就是IoC的本质。这就要求开发者在使用Spring框架时，眼中看到的是“XML配置”，心中想的是“Java代码”。

### Bean的基本定义和Bean别名

beans元素是Spring配置文件的根元素，该元素可以指定如下属性。

* default-lazy-init：指定beans元素下配置的所有Bean默认的延迟初始化行为。
* default-merge：指定beans元素下配置的所有Bean默认的融入行为。
* default-autowire：指定beans元素下配置的所有Bean默认的自动装配行为。
* default-autowire-candidates：指定beans元素下配置的所有Bean是否作为自动装配的候选Bean。
* default-init-method：指定该beans元素下配置的所有Bean默认的初始化方法。
* default-destory-method：指定beans元素下配置的所有Bean默认的回收方法。

上述beans中的属性，可以在每个bean子元素中指定，将属性名default去掉即可。区别是，beans中属性的定义对所有的bean起作用，为bean元素指定这些属性，只对特定Bean起作用；如果两者所指定的属性不一致，bean下指定的属性会覆盖beans下指定的属性。

bean元素是beans元素的子元素，beans元素可以包含多个bean子元素，每个bean子元素定义一个Bean，每个Bean对应Spring容器里的一个Java实例。

定义Bean时，通常需要指定两个属性：id，确定该Bean的唯一标识，class，指定该Bean的具体实现类。如果需要Bean指定别名，有两种方式。

* 定义bean元素时，通过name属性指定别名：如果需要为Bean实例指定多个别名，则可以在name属性中使用逗号、冒号或者空格来分隔多个别名，后面通过任一别名即可访问该Bean实例。
* 通过alias元素为已有的Bean指定别名。

在一些极端的情况下，程序无法在定义Bean时就指定所有的别名，而是需要在其他地方为一个已经存在的Bean指定别名，则可使用alias元素完成，该元素可指定如下两个属性。

1. name：该属性指定一个Bean实例的标识名，表明将为该Bean实例指定别名。
2. alias：指定别名。

示例：

<bean id=”person” class=”……” name=”#abc,@123,abc” />

<alias name=”person” alias=”jack” />

### 1.5.2 容器中Bean的作用域

当通过Spring容器创建一个Bean实例时，不仅可以完成Bean实例的实例化，还可以为Bean指定特定的作用域。Spring支持如下5中作用域。

* singleton：单例模式，在整个Spring IoC容器中，singleton作用域的Bean将只生成一个实例。
* prototype：每次通过容器的getBean方法获取prototype作用域的Bean时，都将生成一个新的Bean实例。
* request：对于一次HTTP请求，request作用域的Bean将只生成一个实例，这意味着，在同一次HTTP请求内，程序每次请求该Bean，得到的是同一个实例。只有在Web应用中使用Spring时，该作用域才有效。
* session：对于一次HTTP会话，session作用域的Bean将只生成一个实例，这意味着，在同一次HTTP会话内，程序每次请求该Bean，得到的是同一个实例。只有在Web应用中使用Spring时，该作用域才有效。
* global session：每个全局的HTTP Session对应一个Bean实例。只有在Web应用中使用Spring时，该作用域才真正有效。

比较常用的是singleton和prototype两种作用域，对于singleton作用域的Bean，每次请求该Bean都将获得相同的实例。容器负责跟踪Bean实例的状态，负责维护Bean实例的生命周期行为；如果一个Bean被设置成prototype作用域，程序每次请求该id的Bean，Spring都会新建一个Bean实例，然后返回给程序。在这种情况下，Spring容器仅仅使用new关键字创建Bean实例，创建完成后就不再跟踪、维护。

如果不指定Bean的作用域，Spring默认使用singleton作用域。Java在创建Java实例时，需要进行内存申请；销毁实例时，需要完成垃圾回收，这些工作都会导致系统开销的增加。因此prototype作用域的Bean的创建、销毁代价比较大。而singleton作用域的Bean实例一旦创建完成可以反复使用。因此尽量避免将Bean设置成prototype作用域。

Spring配置文件通过scope属性指定Bean的作用域，该属性可以接收singleton、prototype、request、session和globalSession五个值，分别代表上面的5个作用域。

<bean id=”p1” class=”com.langsin.impl.Worker”/>

<bean id=”p2” class=”com.langsin.impl.Worker” scope=”prototype” />

上面配置文件定义了两个范围的bean，下面通过程序进行测试：

package com.langsin.test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

public static void main(String[] args) {

ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

System.out.println(ctx.getBean("p1")==ctx.getBean("p1"));

System.out.println(ctx.getBean("p2")==ctx.getBean("p2"));

}

}

执行程序结果如下：



从上面的运行结果来看，对于singleton作用域的Bean，每次请求该id的Bean，都将返回同一个共享实例，因此两次获取的Bean为同一个，但对于prototype作用域的Bean，每次请求该id的Bean都将产生新的实例，因此两次请求获得的Bean实例不相同。

对于request作用域，看如下定义：

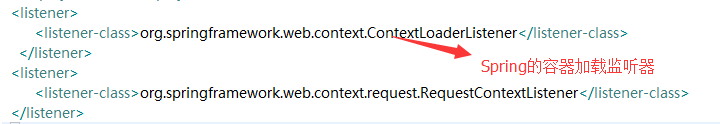
<bean id=”loginAction” class=”com.langsin.login.LoginAction” scope=”request” />

针对每次HTTP请求，Spring容器会根据loginAction Bean定义创建一个新的LoginAction Bean实例，且该实例仅在当前HTTP request内有效。因此，如果程序需要，完全可以自由更改Bean实例内部状态；其他请求所获得的loginAction Bean都不一样，当请求结束后，request作用域的Bean实例都将被销毁。

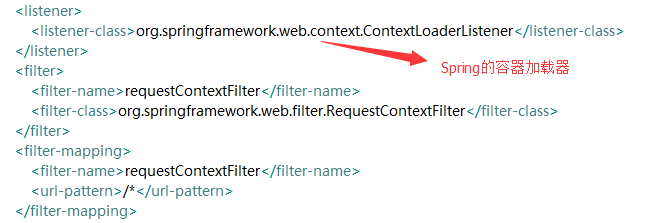
session作用域与request作用域完全类似，区别在于：request作用域的Bean对于每次HTTP请求有效，而session作用域的Bean则对于每次HTTP Session有效。

request和Session作用域只在Web应用中才有效，并且必须在Web应用中增加额外配置才生效。为了让request和session两个作用域生效，必须将HTTP请求对象绑定到Spring的处理器上，这使得具有request和session作用域的Bean实例能够在后面的调用链中被访问到。

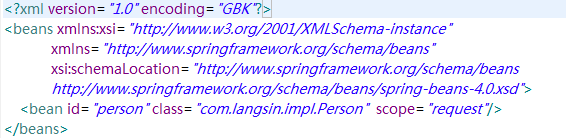
对于能够支持Servlet2.4及更新的规范的Web容器，可以在Web应用的web.xml文件中增加如下Listener配置，该Listener负责使request作用域生效。



对于只支持Servlet2.4以前规范的Web容器，则该容器不支持Listener规范，故无法使用这种配置方式，只能改为使用Filter配置方式。配置片段如下：

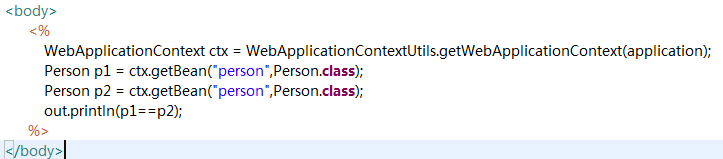


一旦在web.xml中增加了如上配置，程序就可以在Spring配置文件中使用request或session作用域了。下面的配置文件配置了一个实现类为Person的Bean实例，器作用域是request。配置文件代码如下：



这样Spring容器会为每次HTTP生成一个Person实例，当请求响应结束时，该实例也随之消失。

接下来使用JSP脚本来测试该request作用域，脚本如下：



使用浏览器请求该页面，将可以看到页面显示：true。

### 配置依赖

根据前面的介绍，Java应用中各组件相互调用的实质可以归纳为依赖关系，根据注入方式的不同。Bean的依赖注入分为设值注入、构造注入。不管是设值注入还是构造注入，都视为Bean的依赖，接受Spring容器管理，依赖关系的值要么是一个确定的值，要么是Spring容器中其他Bean的引用。

通常不建议使用配置文件管理Bean的基本类型的属性值；通常只使用配置文件管理容器中Bean与Bean之间的依赖关系。

对于singleton作用域的Bean，如果没有强行取消其预初始化行为；系统会在创建Spring容器是预初始化所有的singleton Bean，与此同时，该Bean所依赖的Bean也会一起实例化。

BeanFactory与ApplicationContext实例化容器中Bean的时机不同；前者等到程序需要Bean实例时才创建Bean；而后者在Spring容器被创建时，就会预初始化所有的singletion Bean。因此，有可能程序可以正确的创建BeanFactory的实例，但是在通过容器获取Bean时抛出异常，即创建Bean实例或注入它的依赖时抛出异常。配置错误的延迟出现，会给系统引入不安全因素。而ApplicationContext则默认预实例化所有的singleton作用域Bean，所以ApplicationContext实例化过程比BeanFactory实例化过程的时间和内存开销大，但可以在容器初始化阶段就检验出配置错误。

前面提到Spring作用就是管理Java EE组件，Spring把所有Java对象都称为Bean，所以我们可以把任何Java 类都部署在Spring容器中——只要该Java类具相应的构造器即可。

除此之外，Spring容器可以为任何Java对象注入任何类型的属性——只要该Java对象为该属性提供了对应的setter方法即可。

由于Java实例的属性值可以是各种数据类型，除了基本类型值、字符串类型值等，还可以是其他Java实例，也可以是容器中其他Bean实例，甚至是Java集合、数组等，所以Spring允许通过如下元素为Bean实例的属性指定值。

1. value属性
2. ref属性
3. bean属性
4. list、set、map及props属性

下面详细介绍各种属性。

### 1.5.4 设置普通属性值

value属性用于指定字符串类型、基本类型的属性值，Spring使用XML解析器来解析出这些数据，然后利用java.beans.PropertyEditor完成类型转换：从java.lang.String类型转换为所需的参数值类型。如果目标类型是基本数据类型，通常都可以正确转换。

下面的代码演示value元素确定属性值的情况。假设有如下的Bean类，该Bean类里包含int型和double型两个属性，并为这个两个属性提供对应的的setter方法。

package com.langsin.test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

private int num;

private double dou;

public int getNum() {

return num;

}

public void setNum(int num) {

this.num = num;

}

public double getDou() {

return dou;

}

public void setDou(double dou) {

this.dou = dou;

}

public static void main(String[] args) {

ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

Test t1 = ctx.getBean("test", Test.class);

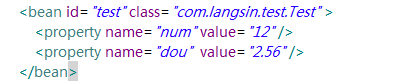
System.out.println(t1.getNum());

System.out.println(t1.getDou());

}

}

上面的测试类Test中含有两个基本数据类型，Spring配置文件使用value属性元素即可为这两个属性指定属性值。配置文件如下：



运行程序可以看到输出结果，表明Spring为这两个属性成功注入了属性值。

value属性主要用于传入字符串、基本类型的属性值。

### 1.5.5 配置合作者Bean

如果需要为Bean设置的属性值是容器中的另一个Bean实例，则应该使用ref元素，使用ref元素时可以指定bean属性，该属性用于引用容器中其他Bean实例的属性值。

如下配置为：

<bean id=”steelAxe” class=”com.langsin.impl.SteelAxe” />

<bean id=”worker” class=”com.langsin.impl.Worker”>

<property name=”axe”>

<ref bean=”steelAxe”/>

</property>

</bean>

与注入普通属性值类似的是，，注入合作者Bean也有一种简洁的写法，看如下配置：

<bean id=”steelAxe” class=”com.langsin.impl.SteelAxe” />

<bean id=”worker” class=”com.langsin.impl.Worker”>

<property name=”axe” ref=”steelAxe” />

</bean>

通过为property元素增加ref属性，一样可以将容器中另一个Bean作为调用setter方法的参数。这种写法更加简洁，配置效果与前面使用ref元素完全相同。

### 1.5.6 使用自动装配注入合作者Bean

Spring能自动装配Bean与Bean之间的依赖关系，即无须使用ref显式指定依赖Bean，而是由Spring容器检查XML配置文件内容，根据某种规则，为调用者Bean注入被依赖的Bean。

Spring的自动装配可以通过beans元素的default-autowire属性指定，该属性对配置文件所有的Bean起作用；也可以通过对bean元素的autowire属性指定，该属性只对该Bean起作用。在同一个Spring容器中完全可以让某些Bean使用自动装配，而另一些Bean不使用自动装配。

自动装配可以减少配置文件的工作量，但降低了依赖关系的透明性和清晰性。

autowire、default-autowire属性可以接受如下值。

* no：不使用自动装配。Bean依赖必须通过ref元素定义。这是默认配置，在比较大的部署环境中不鼓励改变这个配置，显示配置合作者能够得到更清晰的依赖关系。
* byName：根据setter方法名进行自动装配。Spring容器查找容器中的全部Bean，找出其id与setter方法名去掉set前缀，并小写首字母后同名的Bean来完成注入。如果没有找到匹配的Bean实例，则Spring不会进行任何注入。
* byType：根据setter方法的形参类型来自动装配。Spring容器查找容器中全部的Bean，如果正好有一个Bean类型与setter方法的形参类型匹配，就自动注入这个Bean；如果找到多个这样的Bean，就抛出一个异常，如果没有找到这样的Bean，，则什么都不操作，setter方法也不会被调用。
* constructor：与byType类似，区别是用于自动匹配构造器的参数。如果容器不能恰好找到一个与构造器参数类型匹配的Bean，则会抛出异常。
* autodetect：Spring容易根据Bean内部结构，自行决定使用constructor或byType策略。如果找到一个默认的构造函数，那么就会应用byType策略。

#### 1.5.6.1 byName规则

byName规则是指setter方法的方法名与Bean的id进行匹配，假如Bean A的实现类包含setB方法，而Spring的配置文件恰好包含id为b的Bean，则Spring容器会将b实例注入Bean A中。如果容器中没有名字匹配的Bean，Spring则不会做任何事。

看如下配置：

<bean id=”worker” class=”com.langsin.impl.Worker” autowire=”byName”/>

<bean id=”gunDog” class=”com.langsin.test.GunDog” >

<property name=”name” value=”niuniu” />

</bean>

假设对于Worker类恰好有如下setter方法

public void setGunDog(Dog dog){

this.dog = dog;

}

上面setter方法的方法名为setGunDog，Spring容器就会寻找容器中id为gunDog的Bean，如果能找到这样的Bean，该Bean就会作为参数传递给setGunDog方法；如果找不到这样的Bean，Spring就不做任何事。

#### 1.5.6.2 byType规则

byType规则是根据setter方法的参数类型与Bean类型进行匹配。假如A实例有setB(B b)的方法，而Spring的配置文件中恰好有一个类型为B的Bean实例，容器为A注入类型匹配的Bean实例，如果容器中没有类型为B的实例，Spring则不会调用setB方法，但如果容器中包含多个B类型的实例，程序将会抛出异常。

看如下配置：

<bean id=”worker” class=”com.langsin.impl.Worker” autowire=”byType” />

<bean id=”gunDog” class=”com.langsin.test.GunDog”>

<property name=”name” value=”niuniu” />

</bean>

上面的配置文件指定了byType自动装配策略，而且Worker类恰好有如下setter方法。

public void setDog(Dog dog){

this.dog = dog;

}

上面的setter方法的形参类型是Dog，而Spring容器中的com.langsin.test.GunDog类实现了Dog接口，因此Spring会以该Bean为参数来调用worker的setDog方法。

但如果在配置文件中再配置一个如下的Bean

<bean id=”petDog” class=”com.langsin.test.PetDog” >

<property name=”name” value=”yingtao” />

</bean>

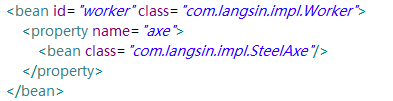
此时，Spring将无法按byType策略进行自动装配：容器中有两个类型为Dog的Bean，Spring无法确定应为worker Bean注入哪个，所以程序抛出异常。

剩下的两种大同小异，就不在演示。

### 1.5.7 注入嵌套Bean

如果某个Bean类A依赖Bean类B，而B不想被Spring直接访问，则可以使用嵌套Bean。不能直接访问的含义是：Spring不能在容器中通过id去获取这个Bean的实例。嵌套Bean就是使用bean元素，配置成property元素或constructor-args的子元素，那么该bean元素配置的Bean仅仅作为setter注入、构造注入的参数。由于容器不能获取嵌套Bean，因此不需要指定id属性。

例如如下配置：



采用上面的配置形式可以保证嵌套Bean不能被容器访问，因此不用担心其他程序修改嵌套Bean。使用嵌套Bean与使用ref引用容器中另一个Bean在本质上是一样的。

Spring框架的本质就是通过XML配置文件来驱动Java代码，当程序调用setter方法或有参数的构造器时，程序总需要传入参数值，随参数类型的不同，Spring配置文件当然也要随之修改。

* 形参类型是基本类型、String、日期等，直接使用value指定字面值即可。
* 形参类型是复合类型（自定义类），那就需要传入一个Java对象作为实参，于是有两种方式：1、使用ref引用容器中的已配置的Bean。2、使用bean元素配置一个嵌套Bean。

除此之外，形参类型还可以是Set、List、Map等集合，也可能是数组类型。接下来介绍在Spring配置文件中配置Set、List、Map、数组等参数值。

### 1.5.8 注入集合

如果需要调用形参类型为集合的setter方法，或调用形参类型为集合的构造器，则可以使用集合元素list、set、map和props分别来设置类型为List、Set、Map和Properties的集合参数值。

下面定义一个包含大量集合属性的Java类，配置文件将会通过list、set等元素来为这些集合属性设置属性值。看如下Java类。

package com.langsin.impl;

import java.util.Arrays;

import java.util.List;

import java.util.Map;

import java.util.Properties;

import java.util.Set;

public class Worker{

private Axe axe;

private List<String> schools;

private Map<String,Axe> phaseAxes;

private Properties health;

private Set<Object> set;

private String[] books;

public Worker(){}

public Worker(Axe axe){

this.axe = axe;

}

public void setAxe(Axe axe){

this.axe = axe;

}

public List<String> getSchools() {

return schools;

}

public void setSchools(List<String> schools) {

this.schools = schools;

}

public Map<String, Axe> getPhaseAxes() {

return phaseAxes;

}

public void setPhaseAxes(Map<String, Axe> phaseAxes) {

this.phaseAxes = phaseAxes;

}

public Properties getHealth() {

return health;

}

public void setHealth(Properties health) {

this.health = health;

}

public Set<Object> getSet() {

return set;

}

public void setSet(Set<Object> set) {

this.set = set;

}

public String[] getBooks() {

return books;

}

public void setBooks(String[] books) {

this.books = books;

}

public Axe getAxe() {

return axe;

}

public void useAxe() {

System.out.println(this.axe.chop());

}

public void test(){

System.out.println(this.schools);

System.out.println(this.phaseAxes);

System.out.println(this.health);

System.out.println(this.set);

System.out.println(Arrays.toString(this.books));

}

}

在上面的Worker类中，定义了5个集合类型的成员变量。下面分别为property元素增加list、set、map和props子元素来配置这些集合类型的参数值。

<?xml version="1.0" encoding="GBK"?>

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

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

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

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

<!-- 如果不加任何特殊的配置，该Bean默认是singletion行为的 -->

<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basenames">

<list>

<value>mesg</value>

</list>

</property>

</bean>

<bean id="stoneAxe" class="com.langsin.impl.StoneAxe" />

<bean id="steelAxe" class="com.langsin.impl.SteelAxe" />

<bean id="worker" class="com.langsin.impl.Worker">

<property name="axe">

<bean class="com.langsin.impl.SteelAxe"/>

</property>

<property name="schools">

**<list value-type="java.lang.String">**

**<value>小学</value>**

**<value>中学</value>**

**<value>大学</value>**

**</list>**

</property>

<property name="phaseAxes">

**<map key-type="java.lang.String" value-type="com.langsin.impl.Axe">**

**<entry key="原始社会" value-ref="stoneAxe" />**

**<entry key="农业社会" value-ref="steelAxe"/>**

**</map>**

</property>

<property name="health">

**<props>**

**<prop key="血压">正常</prop>**

**<prop key="身高">180</prop>**

**</props>**

</property>

<property name="set">

**<set value-type="java.lang.Object">**

**<value>普通字符串</value>**

**<bean class="com.langsin.test.Test"/>**

**<ref bean="stoneAxe"/>**

**</set>**

</property>

<property name="books">

**<array value-type="java.lang.String">**

**<value>HTML</value>**

**<value>Java基础</value>**

**</array>**

</property>

</bean>

<bean id="test" class="com.langsin.test.Test" >

<property name="num" value="12" />

<property name="dou" value="2.56" />

</bean>

</beans>

上面的粗体代码是配置集合类型的参数值的关键代码，从配置文件可以看出，Spring对List集合和数组的处理是一样的，都是用list元素来配置，而且数组还可以使用array元素来配置。

当使用list、set、map等元素配置集合类型的参数值时，还需要配置集合元素。由于集合元素既可以是基本类型值、引用容器中的其他Bean、嵌套Bean或集合属性等，所以list、key和set元素又可以接收如下子元素：

* value：指定集合元素是基本数据类型或字符串类型值。
* ref：指定集合元素是容器中的另一个Bean实例。
* bean：指定集合元素是一个嵌套Bean。
* list、set、map及props，array：指定集合元素又是集合。

props元素用于配置Properties类型的参数值，Properties类型是一种特殊的类型，其key和value都只能是字符串，所以Spring配置Properties类型的参数值比较简单：每个key-value对只要分别给出key和value就足够了——而且key和value都是字符串类型，所以使用如下格式的prop元素就可以了。

<prop key="血压">正常</prop>，其中prop元素的key属性指定key值，prop元素的内容为value值。

当使用map元素配置Map参数值时比较复杂，因为Map集合的每个元素由key、value两个部分组成，所以配置文件中的每个entry配置一组key-value对，其中entry元素支持如下4个属性：

* key：如果Map的key是基本类型值或字符串，则可以使用key属性指定Map key的值。
* key-ref：如果Map的key是容器中的另外一个Bean实例，则使用该属性进行值其他Bean的id。
* value：如果Map value是基本类型值或字符串，则可以使用该属性来指定Map value。
* value-ref：如果Map value是容器的另一个Bean实例，则使用该属性来指定容器中其他Bean的id。

从JDK1.5以后，Java可以使用泛型指定集合元素的类型，则Spring可以通过反射来获取集合元素的类型，这样Spring的类型转换器也会器作用了，同样也可以明确指定泛型的类型。

<property name="phaseAxes">

<map key-type="java.lang.String" value-type="com.langsin.impl.SteelAxe">

<entry key="原始社会" value-ref="stoneAxe" />

<entry key="农业社会" value-ref="steelAxe"/>

</map>

</property>

### 1.5.9 组合属性

Spring还支持组合属性的方式。例如，使用配置文件为样子如eb.person.name的属性设置参数值。为Bean的组合属性设置参数值时，除最后一个属性外，其他属性值都不允许为null。

例如如下代码Bean类：

public class ExampleBean{

private Person person = new Person();

public Person getPerson(){

return this.person;

}

}

上面的ExampleBean里提供了一个person成员变量，该person变量的类型是Person类，Person是一个Java类，Person类里有一个String类型的name成员变量，则可以使用组合属性的方式为ExampleBean的person的name指定值。配置文件如下：

<bean id=”exam” class=”com.langsin.test.ExampleBean”>

**<property name=”person.name” value=”孙悟空”/>**

</bean>

通过使用这种组合属性的方式，Spring允许直接为Bean实例的复合类型的属性指定值。但这种设置方式有一点需要注意：使用组合属性指定参数值时，除了最后一个属性外，其他属性都不能为null，否则将引发NullPointer

Exception异常。例如，上面配置文件为person.name指定参数值，则exam的person一定不能为null。并且还要提供getter方法。

对于这种注入组合属性值的形式，每个property元素依然是让Spring执行一次setter方法，但它不再直接调用该Bean的setter方法，而是需要先调用getter方法，然后再调用setter方法。例如上面粗体，相当于让Spring

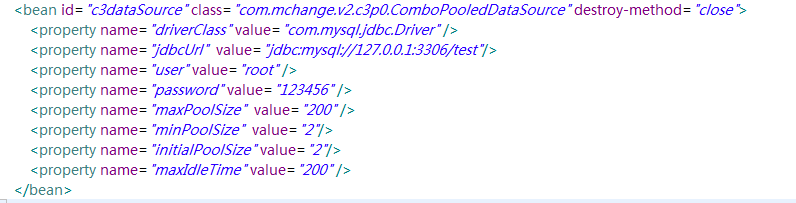
执行如下代码：

exam.getPerson().setName(“孙悟空”);

### 1.5.10 Spring配置数据库连接池

Spring容器对Bean没有特殊要求，甚至不要求该Bean像标准的JavaBean——必须为每个属性提供对应的getter和setter方法。Spring中的Bean是Java实例、Java组件，Spring中的Bean比JavaBean的功能更复杂，用法也更丰富。下面示例把数据源也配置成容器的Bean，该数据源Bean即可用于获取数据库连接。

示例1：C3P0数据源



主程序部分由Spring容器来获取该Bean的实例，获取实例时使用Bean的唯一标识：id属性，id属性是Bean实例在容器中的访问点。下面是主程序代码。

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

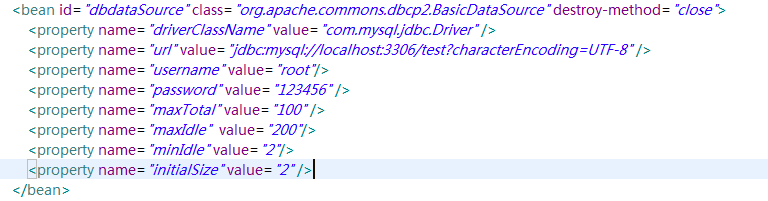
DataSource dataSource = ctx.getBean("c3dataSource",DataSource.class);

Connection conn = dataSource.getConnection();

System.out.println("打印连接对象："+conn);

}

示例2：DBCP数据源



主程序代码如下：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

DataSource dataSource = ctx.getBean("dbdataSource",DataSource.class);

Connection conn = dataSource.getConnection();

System.out.println("打印连接对象："+conn);

}

上面程序从Spring容器中获得了一个DataSource对象，通过该DataSource对象就可以获取简单的数据库连接。从该实例可以看出，Spring的Bean远远超出值对象的JavaBean范畴，Bean可以代表应用中的任何组件、任何资源实例。

虽然Spring对Bean没有特殊要求，但依然建议Spring中的Bean应满足以下几个原则。

* 尽量为每个Bean实现类提供无参的构造器。
* 接受构造注入的Bean，则应提供对应的、带参数的构造函数。
* 接受设值注入的Bean，则应提供对应的setter方法，并不要求提供对应的getter方法。

传统的JavaBean和Spring中的Bean存在如下区别。

* 用处不同：传统的JavaBean更多是作为值对象传递参数：Spring的Bean用于几乎无所不包，任何应用组件都被称为Bean。
* 写法不同：传统的JavaBean作为值对象，要求每个属性都提供getter和setter方法；但Spring的Bean只需要为接受设值注入的属性提供setter方法即可。
* 声明周期不同：传统的JavaBean作为值对象传递，不接受任何容器管理其声明周期；Spring中的Bean由Spring管理其声明周期。

## 1.6 创建Bean的2种方式

在大多数情况下，Spring容器直接通过new关键字调用构造器来创建Bean实例，而class属性指定了Bean实例的实现类。因此，bean元素必须指定Bean实例的class属性，但这并不是实例化Bean的唯一方法。Spring支持使用如下方式来创建Bean。

* 调用构造器创建Bean。
* 调用静态工厂方法创建Bean。
* 调用实例工厂方法创建Bean。

### 1.6.1 使用构造器创建Bean实例

使用构造器来创建Bean实例是最常见的情况，如果不采用构造注入，Spring底层会调用Bean类的无参构造器来创建实例，因此要求该Bean提供无参的构造器。在这种情况下，class元素是必须的，class属性的值就是Bean实例的实现类。Spring对Bean实例的所有属性执行默认初始化，即所有基本类型为0或false，引用类型为null。

接下来，BeanFactory会根据配置文件决定依赖关系，先实例化被依赖的Bean实例然后为注入依赖关系，最后将一个完整的Bean实例返回给程序。

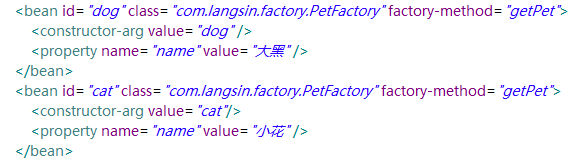
如果采用构造注入，则要求配置文件为bean元素添加constructor-arg子元素，每个constructor-arg子元素配置一个构造器参数，Spring容器使用带对应参数的构造器来创建Bean实例。Spring调用构造器传入的参数即可用于初始化Bean的实例变量，最后也将一个完成的Bean实例返回给程序。

### 1.6.2 使用静态工厂方法创建Bean

使用静态工厂方法创建Bean实例时，class属性也必须指定，但此时class属性并不是指定Bean实例的实现类，而是静态工厂类。除此之外，还需要使用factory-method属性来指定静态工厂方法，Spring将调用静态工厂方法返回一个Bean实例，得到Bean实例后，Spring后面的处理步骤与采用普通方法创建Bean完全一样。

下面示例一个Bean通过静态工厂来创建，所以这个bean元素的class属性就是指定的静态工厂类，factory-method指定工厂的静态方法。如果factory-method需要参数，则需要使用constructor-arg元素传入。

示例代码1：beans.xml



从上面配置文件可以看出，dog和cat两个Bean的配置的class属性和factory-method属性完全相同，这是因为这两个示例都是由同一个静态工厂类、同一个静态方法生产得到。配置这两个Bean示例时指定的静态工厂方法的参数值不同，配置工厂方法的参数值使用constructor-arg子元素。

程序代码2：

//定义一个宠物接口类

package com.langsin.factory;

public interface Pet {

public void testPet();

}

//定义两个宠物实现类

package com.langsin.factory;

public class Dog implements Pet {

private String name;

public void setName(String name){

this.name = name;

}

@Override

public void testPet() {

System.out.println(name+"：狗喜欢吃骨头！");

}

}

package com.langsin.factory;

public class Cat implements Pet {

private String name;

public void setName(String name){

this.name = name;

}

@Override

public void testPet() {

System.out.println(name+"：猫喜欢吃鱼！");

}

}

//定义一个静态工厂类

package com.langsin.factory;

public class PetFactory {

public static Pet getPet(String arg){

if("dog".equals(arg)){

return new Dog();

}else{

return new Cat();

}

}

}

主程序获取Spring容器的cat、dog两个Bean实例的方法依然无须改变，只需要调用Spring容器的getBean方法即可。主程序如下：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

Pet p1 = ctx.getBean("dog", Pet.class);

p1.testPet();

Pet p2 = ctx.getBean("cat", Pet.class);

p2.testPet();

}

指定Spring使用静态工厂方法来创建Bean实例时，Spring先解析配置文件，并根据配置文件指定的信息，通过反射调用静态工厂方法，该静态工厂方法的返回值作为Bean实例。在这个过程中Spring不再负责创建Bean实例，Bean实例是由用户提供的静态工厂类负责创建。

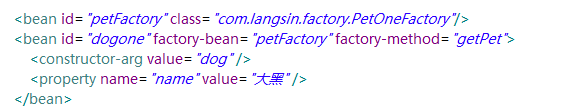
### 1.6.3 调用实例工厂方法创建Bean

实例工厂方法与静态工厂方法只有一点不同：调用静态工厂方法只需要工厂类即可，而调用实例工厂方法则需要工厂实例。所以配置实例工厂方法与配置静态工厂方法基本相似，只有一点区别：配置静态工厂方法使用class指定静态工厂类，而配置实例工厂方法则使用factory-bean指定工厂实例。

采用实例工厂方法创建Bean的bean元素时需要指定如下两个属性：

* factory-bean：该属性的值为工厂Bean的id。
* factory-method：该属性指定实例工厂的工厂方法。

将上一节的程序做下稍微的改动，XML配置文件如下：



上面的PetOneFactory就是一个简单的Pet工厂，getPet方法就是负责生产Pet的工厂方法。单独定义一个bean来生成工厂对象，在下面的bean定义中使用factory-bean属性引用上面的工厂Bean。其他跟静态工厂方式创建Bean完全一致。

## 1.7 深入理解容器中的Bean

Spring框架绝大部分工作都集中在对容器中Bean的管理上，包括管理容器中Bean的生命周期、使用Bean继承等特殊功能。通过深入的管理，应用程序可以更好地使用这些Java组件。

### 1.7.1 抽象Bean与子Bean

在实际开发中，有可能会出现这样的情况：随着项目越来越大，Spring配置文件出现了多个bean，配置具有大致相同的配置信息，只有少量信息不同，这将导致配置文件出现很多重复的内容。如果保留这些配置，则可能导致的问题是：

* 配置文件臃肿。
* 后期难以修改、维护。

为了解决上面问题，可以考虑把多个bean配置中相同的信息提取出来，集中成配置模板——这个配置模板并不是真正的Bean。因此Spring不应该创建该配置模板，于是需要为该bean配置增加abstract属性值为true表示这是个抽象Bean。

抽象Bean不能被实例化，Spring容器不会创建抽象Bean实例。抽象Bean的价值在于被继承，抽象Bean通常作为父Bean被继承。抽象Bean只是配置信息的模板，指定abstract为true即可阻止Spring实例化该Bean，因此抽象Bean可以不指定class属性。

将大部分相同信息配置成抽象Bean之后，将实际的Bean实例配置成该抽象Bean的子Bean即可。子Bean定义可以从父Bean继承实现类、构造参数、属性值能配置信息，除此之外，子Bean配置可以增加新的配置信息，并可以指定新的配置信息覆盖父Bean的定义。

通过为一个bean元素指定parent属性即可指定该Bean是一个子Bean，parent属性指定该Bean所继承的父Bean的id。子Bean无法从父Bean继承如下属性：depends-on、autowire、singleton、scope、lazy-init，这些属性只能从子Bean定义中获取，或采用默认值。

看如下配置：

<bean id=”steelAxe” class=”com.langsin.impl.SteelAxe” />

<bean id=”personTemplete” abstract=”true”>

<property name=”name” value=”zhangsan” />

<property name=”axe” ref=”steelAxe” />

</bean>  
<bean id=”chinese” class=”com.langsin.impl.Chinese” parent=”personTemplete”/>

<bean id=”american” class=”com.langsin.impl.American” parent=”personTemplete”/>

在配置文件中chinese和americanBean都指定了parent=“personTemplete”，表明这两个Bean都可以从父Bean那里继承得到配置信息。虽然这两个Bean没有直接指定proerty子元素，但他们会从personTemplete模板那里继承得到两个property子元素。

### 1.7.2 Bean继承与Java继承的区别

Spring中的Bean继承与Java中的继承截然不同，前者是实例与实例之间参数的延续，后者则是一般到特殊的细化。前者是对象与对象之间的关系，后者则是类与类之间的关系。Spring中Bean的继承和Java中Bean的继承有如下区别：

* Spring中的子Bean和父Bean可以是不同类型，但Java中的继承则可保证子类是一种特殊的父类。
* Spring中Bean的继承是实例之间的关系，因此主要表现为参数值的延续；而Java中的继承是类之间的关系，主要表现为方法、属性的延续。
* Spring中的子Bean不可作为父Bean使用，不具备多态性；Java中的子类实例完全可以当成父类实例使用。

### 1.7.3 容器中的工厂Bean

此处的工厂Bean，与前面介绍的实例工厂Bean，或者静态工厂Bean有所区别：前面那些Bean是标准的工厂模式，Spring只是负责调用工厂方法来创建Bean的实例；此处的工厂Bean是Spring的一种特殊Bean，这种工厂Bean必须实现FactoryBean接口。

FactoryBean接口是工厂Bean的标准接口，把实现FactoryBean接口的工厂Bean部署在容器中后，如果程序通过getBean方法来获取它时，容器返回的不是FactoryBean实现类的实例，而是返回FactoryBean的产品。（即通过工厂所创建的对象被返回）

FactoryBean接口提供如下三个方法：

* T getObject()：负责返回该工厂Bean生成的Java实例。
* Class<?> getObjectType()：返回该工厂Bean生成的Java实例的实现类。
* boolean isSingleton()：表示该工厂Bean生成的Java实例是否是单例模式。

配置FactoryBean与配置普通Bean的定义没有区别，但当程序向Spring容器请求获取该Bean时，容器返回该FactoryBean的产品，而不是返回该FactoryBean本身。所以，实现FactoryBean接口的最大作用在于：Spring容器返回的是该Bean实例的getObject()方法的返回值。而getObject()方法由开发者负责实现，所以返回什么类型就由开发者自己决定。

实例1：

package com.langsin.factory;

import org.springframework.beans.factory.FactoryBean;

public class GetMyObjectFactoryBean implements FactoryBean<Object> {

private String targetClass;

public void setTargetClass(String targetClass){

this.targetClass = targetClass;

}

@Override

**public Object getObject() throws Exception {**

**Class<?> clazz = Class.forName(this.targetClass);**

**return clazz.newInstance();**

**}**

@Override

public Class<?> getObjectType() {

return Object.class;

}

@Override

public boolean isSingleton() {

return false;

}

}

上面的GetMyObjectFactoryBean是一个标准的工厂Bean，该工厂Bean的关键代码为粗体部分，由于程序不需要让GetMyObjectFactoryBean的getObject方法产生的对象是单例的，所以返回为false。下面为配置文件：



从配置文件来看，部署工厂Bean与部署普通Bean其实没有任何区别，同样只需要为该Bean配置id、class属性即可。但Spring对FactoryBean接口的实现类的处理有所不同。Spring容器会自动检测容器中所有的Bean，如果发现某个Bean实现了FactoryBean接口，Spring容器就会在实例化该Bean、根据property执行setter方法之后，额外要调用该Bean的getObject方法，并将返回值作为容器中的Bean。

下面是主程序调用代码：

public static void main(String[] args) throws Exception {

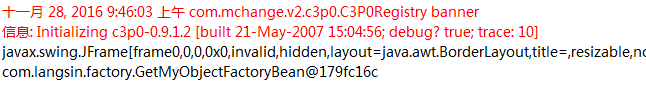
ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

System.out.println(ctx.getBean("myframe"));

System.out.println(ctx.getBean("&myframe"));//在Bean id前面加&符号，会返回FactoryBean本身

}

运行效果如下：



### 1.7.4 强制初始化Bean

在大多数情况下，Bean之间的依赖非常直接，Spring容器返回Bean实例之前，先要完成Bean依赖关系的注入。假如Bean A依赖于Bean B，程序请求Bean A时，Spring容器会自动先自动初始化Bean B，再将Bean B注入Bean A，最后将具备完整依赖的BeanA返回给程序。

在极端的情况下，Bean之间的依赖关系不够直接。比如某个类的初始化块中使用其他Bean，Spring总是先初始化主调Bean，当执行初始化块时，被依赖的Bean可能还没有实例化，此时将引发异常。

为了显示指定被依赖Bean在目标Bean之前初始化，可以使用depends-on属性，该属性可以在初始化主调Bean之前，强制初始化一个或多个Bean。配置文件如下：

<bean id="one" class="com.langsin.factory.One" />

<bean id="two" class="com.langsin.factory.Two" />

如上述没有设置Bean one与Bean two之间的依赖关系，所以Spring容器根据配置的先后关系先加载one再加载two，One、Two两个类中只有构造器，并且只是做了简单的输出操作，代码如下：

示例1：

package com.langsin.factory;

public class One {

public One(){

System.out.println("打印One。。。。。。。。");

}

}

示例2：

package com.langsin.factory;

public class Two {

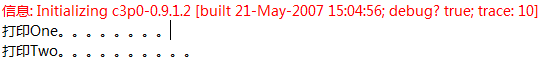
public Two(){

System.out.println("打印Two。。。。。。。。。。");

}

}

执行效果如下：



在Bean one中加入强制依赖为two，例如<bean id="one" class="com.langsin.factory.One" depends-on=”two” />加载关系后，则执行效果如下：



## 1.8 容器中的Bean生命周期

Spring可以管理singleton作用域的Bean的生命周期，Spring可以精确地知道该Bean何时被创建、何时被初始化完成、容器何时准备销毁该Bean实例。

对于prototype作用域的Bean，Spring容器仅仅负责创建，当容器创建Bean实例之后，Bean实例完全交给客户端代码管理，容器不再跟踪其生命周期。每次客户端请求prototype作用域的Bean时，Spring都会产生一个实例交给客户端程序，就不再过问了。

对于singleton作用域的Bean，每次客户端代码请求时都返回同一个共享实例，客户端代码不能控制Bean的销毁，Spring容器负责跟踪Bean实例的产生、销毁。Spring容器可以在创建Bean之后，进行某些通用资源申请；还可以在销毁Bean实例之前，先回收某些资源，比如数据库连接。

对于singleton作用域的Bean，Spring容器知道Bean何时实例化结束、何时销毁，Spring可以管理实例化结束之后和销毁之前的行为，即Bean的存活期间的行为。

### 1.8.1 依赖关系注入之后的行为

Spring提供两种方式在Bean全部属性设置成功后执行特定情况。

* 使用init-method属性。
* 实现InitializingBean接口。

第一种方式，使用init-method属性指定某个方法应在Bean全部依赖关系设置结束后自动执行。使用这种方式不需要将代码与Spring的接口耦合在一起，代码污染小。

第二种方式，让Bean类实现InitializingBean接口，该接口提供一个方法，afterPropertysSet()，Spring容器会在为该Bean注入依赖关系之后，调用该Bean所实现的afterPropertysSet方法。

下面示例中的Bean既实现了InitializingBean接口，也包含了一个普通的初始化方法——在配置文件中将该方法配置成初始化方法。

package com.langsin.init;

import org.springframework.beans.BeansException;

import org.springframework.beans.factory.InitializingBean;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ApplicationContextAware;

public class InitActionBean implements InitializingBean,ApplicationContextAware {

private String name;

public InitActionBean(){

System.out.println("InitActionBean的构造器被调用。。。。");

}

public void setName(String name){

this.name = name;

System.out.println("调用setName方法完成赋值操作。。。。。");

}

@Override

public void setApplicationContext(ApplicationContext arg0)

throws BeansException {

System.out.println("设置Spring容器的对象。。。。。");

}

@Override

**public void afterPropertiesSet() throws Exception {**

**System.out.println("初始化结束后，调用afterPropertiesSet方法。。。。");**

**}**

**public void init(){**

**System.out.println("调用指定的init方法完成初始化之后的操作。。。。");**

**}**

}

上面粗体代码定义了一个init方法，此方法名可以任意指定，然后在配置文件中使用init-method属性指定该方法。afterPropertiesSet方法为实现了InitializingBean接口之后的要实现的方法。配置文件如下：

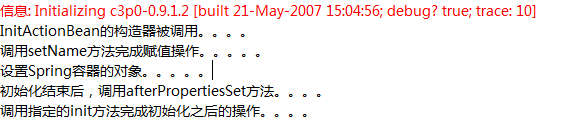


执行下面主程序为：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

}



对于实现了InitializingBean接口的Bean，无须再使用init-method属性来指定初始化方法。配置该Bean实例与配置普通Bean实例完全一样，Spring容器自动检测Bean实例是否实现了特定生命周期接口，并由此决定是否执行生命周期方法。

如果某个Bean实现了InitializingBean接口，当Bean的所有依赖关系被设置完成后，Spring容器会自动调用该Bean实例的afterPorpertiesSet方法。其执行结果与采用init-method属性指定生命周期方法几乎一样，但实现InitializingBean接口污染了代码，是侵入式设计，因此不推荐采用。

### 1.8.2 Bean销毁之前的行为

与定制初始化行为相似，Spring也提供两种方式定制Bean实例销毁之前的特定行为，这两种方式如下：

* 使用destroy-method属性。
* 实现DisposableBean接口。

让Bean类实现DisposableBean接口，需实现接口中声明的destroy方法，该方法在Bean被销毁前调用，第二种方式同样为侵入式设计，不推荐采用。销毁之前的行为与依赖关系注入之后的行为原理一样，不再进行详细叙述。

### 1.8.3 协调作用域不同步的Bean

当两个singleton作用域的Bean存在依赖关系时，或者当prototype作用域的Bean依赖singleton作用域的Bean时，使用Spring提供的依赖注入进行管理即可。

singleton作用域的Bean只有一次初始化的机会，它的依赖关系也只在初始化阶段被设置，当singleton作用域的Bean依赖prototype作用域的Bean时，Spring容器会在初始化singleton作用域的Bean之前，先创建被依赖的prototype Bean，然后才初始化singleton Bean，并将prototype Bean注入到singleton Bean中，这就会导致任何时候通过singleton Bean去访问prototype Bean时，得到的永远是最初那个prototype Bean，任何人访问都将得到相同的信息，这就会产生不同步的现象。解决该问题又如下两种思路：

* 放弃依赖注入：singleton作用域的Bean每次需要prototype作用域的Bean时，主动向容器请求一个新的Bean实例，即可保证每次注入的的都是最新的实例。
* 使用lookup-method元素：使用该元素指定方法注入。

第一种方式显然不合适，将会造成代码的耦合，在通常情况下，推荐采用第二种做法，使用方法注入。

使用lookup子元素可以让Spring容器重写容器中Bean的抽象或者具体方法，返回查找容器中其他Bean的结果，被查找的Bean通常是一个prototype Bean。Spring通过使用JDK动态代理或cglib库修改客户端的二进制码，从而实现上述要求。

为了使用lookup方法注入，大致需要如下两步。

1. 将调用者Bean的实现类定义为抽象类，并定义一个方法来获取被依赖的Bean。
2. 在bean元素中添加lookup-method子元素让Spring容器为调用者Bean提供一个实现类，并实现指定的抽象方法。

示例代码1：singleton Bean

package com.langsin.init;

import com.langsin.factory.Dog;

public abstract class Hunter {

private Dog dog;

public abstract Dog getDog();

public void hunt(){

System.out.println("我的猎狗为："+this.getDog());

}

}

示例代码2：被依赖的prototype Bean

package com.langsin.factory;

public class Dog implements Pet {

private String name;

public void setName(String name){

this.name = name;

}

public String getName(){

return this.name;

}

@Override

public void testPet() {

System.out.println(name+"：狗喜欢吃骨头！");

}

}

示例代码3：配置的xml文件如下：

<bean id="hunDog" class="com.langsin.factory.Dog" scope="prototype"/>

<bean id="hunter" class="com.langsin.init.Hunter">

<lookup-method name="getDog" bean="hunDog"/>

</bean>

运行代码的主程序如下：

示例代码4：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

Hunter hun = ctx.getBean("hunter", Hunter.class);

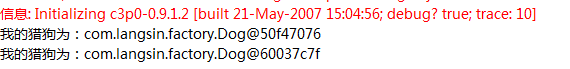
hun.hunt();

hun = ctx.getBean("hunter", Hunter.class);

hun.hunt();

}

运行效果如下：



## 1.9 基于XML Schema的简化配置方式

从Spring2.0开始，Spring允许使用基于XML Schema的配置方式来简化Spring配置文件。

Spring用bean元素即可配置所有的Bean实例，而每个设值注入再用一个property元素即可。这种配置方式简单、直观，而且能以相同风格处理所有Bean的配置——唯一的缺点是配置繁琐，当Bean实例的属性足够多，且属性类型复杂时，基于DTD的配置文件将变得更加繁琐。

在这种情况下，Spring提出了使用XML Schema的配置方式。这种配置方式更加简洁，可以对Spring配置文件进行“减肥”，但需要花一些时间来了解这种配置方式。

### 1.9.1 使用p：命名空间简化配置

p:命名空间不需要特定的Schema定义，它直接存在于Spring内核中。与前面采用property元素定义Bean属性不同的是，当导入p:命名空间后，就可直接在bean元素中使用属性来驱动执行setter方法。

假设有如下的持久化类。

package com.langsin.impl;

import java.util.Locale;

import org.springframework.beans.BeansException;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ApplicationContextAware;

public class Person implements ApplicationContextAware{

private ApplicationContext ctx = null;

private Axe axe = null;

private int age;

public void setAge(int age){

this.age = age;

}

public void setAxe(Axe axe){

this.axe = axe;

}

@Override

public void setApplicationContext(ApplicationContext arg0)

throws BeansException {

this.ctx = arg0;

}

public void test(){

System.out.println("今年："+this.age);

System.out.println(this.axe.chop());

}

}

上面的持久化类中有setAxe、setAge两个setter方法可以通过设值注入来驱动，如果采用原来的配置方式，需要使用property元素来驱动它们。但是如果采用p：命名空间，则可直接采用属性来配置它们。配置文件如下：

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

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

**xmlns:p="http://www.springframework.org/schema/p"**

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

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

**<bean id="person" class="com.langsin.impl.Person" p:age="29" p:axe-ref="steelAxe" />**

。。。。以下省略

配置文件beans元素中的粗体字代码用于导入XML Schema里的p：命名空间，配置文件的第二行粗体字代码直接使用属性配置对age、axe执行设值注入，因此axe设值注入的参数需要引用容器中的另外一个已存在的Bean实例。所示使用axe-ref属性，这个属性表示，指定该值不是一个具体的值，而是对另外一个Bean的引用。

### 1.9.2 使用c：命名空间简化配置

p:命名空间主要用于简化设值注入，而c:命名空间则用于简化构造注入。

假设有如下的持久类。

public class Worker implements Person{

private Axe axe;

private int age;

public Worker(Axe axe,int age){

this.age = age;

this.axe = axe;

}

public void useAxe(){

System.out.println(this.axe.chop());

}

}

上面的Worker类的构造器需要两个参数，传统配置是在bean元素中添加两个constructor-arg元素来代表构造器参数；导入c：命名空间后，可以直接使用属性来配置构造器参数。

如下所示：

<beans …………………

**xmlns:c=”http://www.springframework.org/schema/c”**

>

<bean id=”worker” class=”com.langisn.impl.Worker” c:age=”29” c:axe-ref=”steelAxe” />

<bean id=”steelAxe” class=”com.langsin.impl.SteelAxe” />

上面配置文件中粗体部分代码用于导入XML Schema里的c:命名空间，配置文件worker中，直接使用属性配置了age、axe-ref两个构造参数。由于axe构造器参数需要引用容器中另一个已经存在的Bean实例，所以在axe后面追加了“-ref”后缀。

### 1.9.3 使用util：命名空间简化配置

在Spring框架解压缩包的schema\util\路径下包含有util：命名空间的XML Schema文件，为了使用util：命令空间的元素，必须先在Spring配置文件中导入最新的spring-util-4.0.xsd，也就是需要在Spring配置文件中增加如下粗体字配置代码：

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

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

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

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

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

**http://www.springframework.org/schema/util**

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

在util Schema下提供了如下几个元素。

* constant：该元素用于获取指定类的静态Filed的值。
* property-path：该元素用于获取指定对象的getter方法的返回值。
* list：该元素用于定义一个List Bean，支持value、ref、bean等子元素来定义List集合元素。该元素支持如下三个属性。1、id：该属性指定一个名为id的List Bean实例。2、list-class：该属性指定Spring使用哪个List实现类来创建Bean实例。默认使用ArrayList作为实现类。3、scope：指定该List Bean实例的作用域。
* set：该元素用于定义一个Set Bean，支持使用value、ref、bean等元素来定义Set集合元素。该元素支持如下三个属性。同上，set-class属性，默认使用HashSet作为实现类。
* map：该元素用于定义一个Map Bean，支持使用entry来定义Map的key-value对。使用该标签支持如下三个属性。同上，map-class属性，默认使用HashMap作为实现类。
* properties：该元素用于加载一份资源文件，并根据加载的资源文件创建一个Properties Bean实例。使用该标签可以指定如下几个属性。id，同上，scope，同上，location：指定资源文件的位置。

<?xml version="1.0" encoding="GBK"?>

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

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

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

**xmlns:util="http://www.springframework.org/schema/util"**

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

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

**http://www.springframework.org/schema/util**

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

<bean id="stoneAxe" class="com.langsin.impl.StoneAxe" />

<bean id="steelAxe" class="com.langsin.impl.SteelAxe" />

<bean id="worker12" class="com.langsin.impl.Worker" p:axe-ref="steelAxe"

**p:schools-ref="work.schools" p:phaseAxes-ref="work.phase"**

/>

**<util:properties id="work.prop" location="classpath:mesg\_zh\_CN.properties"/>**

**<util:list id="work.schools" list-class="java.util.LinkedList">**

**<value>小学</value>**

**<value>中学</value>**

**<value>大学</value>**

**</util:list>**

**<util:map id="work.phase" map-class="java.util.HashMap" key-type="java.lang.String" value-type="com.langsin.impl.Axe">**

**<entry key="stoneAxe" value-ref="stoneAxe" />**

**</util:map>**

</beans>

## 1.10 Spring3.0提供的表达式语言（SpEL）不讲

Spring表达式语言（简称SpEL）是一种与JSP2的EL功能类似的表达式语言，它可以在运行时查询和操作对象图。与JSP 2的EL相比，SpEL功能更加强大，它甚至支持方法调用和基本字符串模板函数。

SpEL可以独立于Spring容器使用——只是当成简单的表达式语言来使用；也可以在Annotation或XML配置中使用SpEL，这样就可以充分利用SpEL简化Spring的Bean配置。

### 1.10.1 使用Expression接口进行表达式求值

Spring的SpEL可以单独使用，可以使用SpEL对表达式计算、求值。SpEL主要提供了如下三个接口。

* ExpressionParser：该接口的实例负责解析一个SpEL表达式，返回一个Expression对象。对于表达式的解析器对象，Spring提供了一个默认的实现类为：SpelExpressionParser。
* Expression：该接口的实例代表一个表达式。
* EvaluationContext：代表计算表达式值的上下文，（即源头的概念）。当SpEL表达式中含有变量时，程序将需要使用该API来计算表达式的值。此接口Spring框架提供了一个标准的实现类为：StandardEvaluationContext。

Expression实例代表一个表达式，它包含如下方法用于计算，得到表达式的值。

* Object getValue()：计算表达式的值。
* <T> T getValue(Class<T> resultType)：计算表达式的值，尝试将表达式的值当成T类型处理。
* Object getValue(EvaluationContext context)：使用指定的EvaluationContext来计算表达式的值。
* <T> T getValue(EvaluationContext context,Class<T> resultType)：使用指定的EvaluationContext来计算表达式的值，尝试将表达式的值当成T类型处理。
* Object getValue(Object rootObject)：以rootObject作为表达式的root对象来计算表达式的值。
* <T> T getValue(Object rootObject,Class<T> resultType)：以rootObject作为表达式的root对象来计算表达式的值，尝试将表达式的值当成T类型处理。

下面程序示范了如何利用ExpressionParser和Expression来计算表达式的值。

package com.langsin.spel;

import org.springframework.expression.Expression;

import org.springframework.expression.ExpressionParser;

import org.springframework.expression.spel.standard.SpelExpressionParser;

import org.springframework.expression.spel.support.StandardEvaluationContext;

public class Test {

public static void main(String[] args) {

ExpressionParser parser = new SpelExpressionParser();

//接收的字符串为SpEL的表达式，其中单引号所包含部分为表达式模板中的字符串

Expression exp = parser.parseExpression("'Hello World'");

System.out.println(exp.getValue());

//调用方法的表达式

exp = parser.parseExpression("'Hello World'.concat('--孙悟空！')");

System.out.println(exp.getValue());

//调用方法后，再调用返回值的属性的表达式

exp = parser.parseExpression("'Hello World'.getBytes().length");

System.out.println(exp.getValue());

//使用构造器来创建对象

exp = parser.parseExpression("new String('abc').toUpperCase()");

System.out.println(exp.getValue());

//以指定的对象为root（即：根）来计算表达式的值—即root上的数据

Person person = new Person("张三");

exp = parser.parseExpression("name");

System.out.println(exp.getValue(person,String.class));

//使用EvaluationContext为例指定SpEL表达式的root

StandardEvaluationContext ctx = new StandardEvaluationContext();

ctx.setRootObject(person);

System.out.println(exp.getValue(ctx,String.class));

}

}

class Person{

private String name = null;

public Person(String name){

this.name = name;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

上面程序代码使用ExpressionParser多次解析了不同类型的表达式，ExpressionParser调用parseExpression方法将返回一个Expression实例。程序调用Expression对象的getValue方法即可获取该表达式的值。

EvaluationContext代表SpEL计算表达式值的“上下文”（即：所有内容），这个Context对象可以包含多个对象，但只能有一个root根对象。

* setVariable(String name,Object obj)：向EvaluationContext中放入value对象，该对象名为name。为了在SpEL访问EvaluationContext中指定对象，应采用与OGNL类似的格式；

#name

OGNL：对象图导航语言，OGNL是Object-Graph Navigation Language的缩写，它是一种功能强大的表达式语言，通过它简单一致的表达式语法，可以存取对象的任意属性，调用对象的方法，遍历整个对象的结构图，实现字段类型转化等功能。它使用相同的表达式去存取对象的属性。

StandardEvaluationContext提供了如下方法来设置root对象。

* setRootObject(Object root)

### 1.10.2 Bean定义中的表达式语言支持

SpEL的一个重要作用就是扩展Spring容器的功能，允许在Bean定义中使用SpEL。在XML配置文件和Annotation中都可以使用SpEL。在XML配置文件和Annotation中使用SpEL时，在表达式外面增加#{}包围即可。

例如：

public class Author implements Person{

private String name;

private List<String> books;

//省略所有的getter和setter方法

}

上面的Author类需要依赖注入name、books，可以使用SpEL表示进行注入，Spring配置文件将进一步简化。

如下所示：

<bean id=”author” class=”com.langsin.impl.Person”

**p:name =”#{‘张三’}”**

**p:books =”#{{‘Java基础’,’HTML’}}”**

/>

使用SpEL可以在配置文件中调用方法、创建对象，访问其他Bean的属性，凡是SpEL表达式能做的事，都可在配置文件中进行操作。

### 1.10.3 SpEL语法详细叙述

虽然SpEL在功能上大致与JSP 2的EL类似，但SpEL比JSP 2的EL更加强大，下面来学习SpEL所支持的各种语法细节。

#### 1、直接量表达式

直接量表达式是SpEL中最简单的表达式，直接量表达式就是在表达式中使用Java语言支持的直接量，包括字符串、日期、数值、boolan值和null。

例如如下代码：

Expression exp = parser.parseExpression(“’Hello World’”);

System.out.println(exp.getValue(String.class));

exp = parser.parseExpression(“0.23”);

System.out.println(exp.getValue(Double.class));

#### 2、在表达式中创建数组

SpEL表达式直接支持使用静态初始化、动态初始化两种语法来创建数组。例如如下代码片段：

//创建一个数组

Expression exp = parser.parseExpression(“new String[]{‘java’,’html’,’Spring’}”);

System.out.println(exp.getValue());

//创建一个二维数组，表达式中数组后面不需要带括号

exp = parser.parseExpression(“new int[2][4]”);

System.out.println(exp.getValue())

#### 3、在表达式中创建List集合

SpEL直接使用如下语法来创建List集合：

{ele1,ele2,ele3….}

例如如下代码：

//创建一个ExpressionParser对象，用于解析表达式

ExpressionParser parser = new SpelExpressionParser();

Expression exp = parser.parseExpression(“{‘java’,’struts’,’spring’}”);

System.out.println(exp.getValue());

#### 4、在表达式中访问List、Map等集合元素

为了在SpEL中访问List集合的元素，可以使用如下语法格式：

list[index]

为了在SpEL中访问Map集合的元素，可以使用如下语法格式：

map[key]

public static void main(String[] args) throws Exception {

List<String> list = new ArrayList<String>();

list.add("Java");

list.add("Spring");

Map<String,Double> map = new HashMap<String,Double>();

map.put("Java", 80.0);

map.put("Spring", 45.6);

EvaluationContext ctx = new StandardEvaluationContext();

ctx.setVariable("myList", list);

ctx.setVariable("myMap", map);

ExpressionParser parser = new SpelExpressionParser();

System.out.println(parser.parseExpression("#myList[0]").getValue(ctx));

System.out.println(parser.parseExpression("#myMap[Java]").getValue(ctx));

}

#### 5、调用方法

在SpEL中调用方法与在Java代码中调用方法没有任何区别。如以下代码所示：

List<String> list = new ArrayList<String>();

EvaluationContext ctx = new StandardEvaluationContext();

ctx.setVariable("list", list);

ExpressionParser parser = new SpelExpressionParser();

parser.parseExpression("#list.add('abc')").getValue(ctx);

System.out.println(list);

#### 6、算术、比较、逻辑、赋值、三目运算符

与JSP 2类似的是SpEL同样支持算术、比较、逻辑、赋值、三目运算符等各种运算符，在SpEL中使用赋值运算符的功能比较强大，这种赋值可以直接改变表达式所引用的实际对象。

public static void main(String[] args) throws Exception {

List<String> list = new ArrayList<String>();

EvaluationContext ctx = new StandardEvaluationContext();

ctx.setVariable("list", list);

ExpressionParser parser = new SpelExpressionParser();

Expression exp = parser.parseExpression("3+5");

System.out.println(exp.getValue()); //8

parser.parseExpression("#list.add('abc')").getValue(ctx);

System.out.println(list); //[abc]

parser.parseExpression("#list[0]='ABC'").getValue(ctx);

System.out.println(list); //[ABC]

exp = parser.parseExpression("#list.size()>3?'长度大于3':'长度小于3'");

System.out.println(exp.getValue(ctx));//长度小于3

}

#### 7、类型运算符

SpEL提供了一个特殊的运算符：T()，这个运算符用于告诉SpEL将该运算符内的字符串当“类”处理，避免Spring对其进行其他解析。尤其是调用某个类的静态方法时，T()运算符尤其有用。

ExpressionParser parser = new SpelExpressionParser();

Expression exp = parser.parseExpression("T(java.lang.Math).random()");

System.out.println(exp.getValue());

#### 8、调用构造器

SpEL允许在表达式中直接使用new来调用构造器，这种调用可以创建一个Java对象。例如如下代码：

ExpressionParser parser = new SpelExpressionParser();

Expression exp = parser.parseExpression("new javax.swing.JFrame('窗体').setVisible(true)");

System.out.println(exp.getValue());

#### 9、变量

SpEL允许通过EvaluationContext来使用变量，该对象包含了一个setVariable(String name,Object value)方法，该方法用于设置一个变量。

一旦在EvaluationContext中设置了变量，就可以在SpEL中通过#name来访问该变量。前面已经有此示例，所以不再讲解。

需要指出的是，SpEL中有如下两个特殊变量。

* #this：引用SpEL当前正在计算的对象
* #root：引用SpEL的EvaluationContext的root对象。

#### 10、自定义函数

SpEL允许开发者开发自定义函数。类似于JSP 2EL中的自定义函数，所谓自定义函数，也就是为Java方法重新起个名称而已。

通过StandardEvaluationContext的如下方法即可在SpEL注册自定函数。

* registerFunction(String name,Method m)：将m方法注册成自定义函数，该函数的名称为name。

SpEL自定函数的作用并不大，因为SpEL本身已经允许在表达式语言中调用方法，因此将方法重新定义自定义函数的意义不大。

#### 11、Elvis运算符

Elvis运算符只是三目运算符的特殊写法，例如对于如下三目运算符写法：

name!=null?name:”newVal” 🡪简化后：name!=null?:”newVal”

上面的语句使用三目运算符需要将name变量写两次，因此比较繁琐。SpEL允许将上面写法简写为如下形式：

Expression exp = parser.parseExpression("'abc'.equals('abc')?:1");

System.out.println(exp.getValue());

#### 12、安全导航操作

在SpEL中使用如下语句时可能会导致NullPointerException：

foo.bar——Spring配置中的组合属性

如果root对象的foo属性本身已经是null，那么上面表达式尝试访问foo属性的bar属性时自然就会引发异常。

为了避免上面表达式中的NullPointerException异常，SpEL支持如下用法：foo?.bar

上面表达式在计算root对象的foo属性时，如果foo属性为null，计算结果将直接返回null，而不会引发NullPointerException异常，如下代码所示：

System.out.println(parser.parseExpression(“#foo?.bar”).getValue());

#### 13、集合选择

SpEL允许直接对集合进行选择操作，这种选择操作可以根据指定表达式对集合元素进行筛选，只有符合条件的集合元素才会被选择出来。SpEL集合选择的语法格式如下：

collection.?[condition\_expr]

在上面的语法格式中，condition\_expr是一个根据集合元素定义的表达式，只有当该表达式返回true时，对应的集合元素才会被筛选出来。

ExpressionParser parser = new SpelExpressionParser();

EvaluationContext ctx = new StandardEvaluationContext();

List<String> list = new ArrayList<String>();

list.add("中国制造");

list.add("德国制造");

ctx.setVariable("myList", list);

Expression exp = parser.parseExpression("**#myList.?[contains('中国')]**");

System.out.println(exp.getValue(ctx)); //组成一个新的集合返回。

Map<String,Double> map = new HashMap<String,Double>();

map.put(“Java”,66);

map.put(“Math”,77);

ctx.setVariable(“myMap”,map);

exp = parser.parseExpression(“**#myMap.?[value>70]**”);

System.out.println(exp.getValue(ctx));

上面粗体代码所示，这种集合选择既可以对List集合进行筛选，也可以对Map集合进行刷选。当操作List集合时，condition\_expr中访问的每个属性、方法都是以集合元素为主调的；当操作Map集合时，需要显式的用key代表Map Entry的key，用value代表Map Entry的value。

#### 14、集合投影

SpEL允许对集合进行投影运算，这种投影运算将依次迭代每个集合元素，迭代时将根据指定表达式对集合元素进行计算得到一个新的结果，依次将每个结果收集成新的集合，这个新的集合将作为投影运算的结果。

SpEL投影运算的语法格式为：collection.![condition\_rexpr]

示例1：

public static void main(String[] args) throws Exception {

ExpressionParser parser = new SpelExpressionParser();

EvaluationContext ctx = new StandardEvaluationContext();

List<String> list = new ArrayList<String>();

list.add("中国制造");

list.add("德国制造");

ctx.setVariable("myList", list);

Expression exp = parser.parseExpression("#myList.![replace('制造','made')]");

System.out.println(exp.getValue(ctx));

}

运行结果如下：

#### 15、表达式模板

表达式模板有点类似于带占位符的国际化消息。例如{1}这种形式，这种动态改变的值可以借助于SpEL的表达式模板的支持来进行实现。表达式模板的本质是对“直接量表达式”的扩展，它允许在“直接量表达式”中插入一个或多个#{expr}，#{expr}将会被动态的计算粗来。

Person per = new Person(“张三”,22);

Expression expr = parser.parseExpression(“我的名字是#{name}，身高是#{height}”,new TemplateParserContext());

System.out.println(expr.getValue(per));

## 1.11 Spring3.0提供的Java配置管理

Spring3.0为不喜欢使用XML来配置管理Bean的开发者提供了另外一种管理方式，即：使用Java类进行配置管理。

假如有一个类Chinese实现了Person接口，并含有两个成员变量String name、Axe axe，此Chinese Bean如果通过配置文件注入，格式如下：

<bean id=”chin” class=”com.langsin.impl.Chinese”>

<property name=”name” value=”zhangsan” />

<property name=”axe” ref=”steelAxe”/>

</bean>

<bean id=”steelAxe” class=”com.langsin.impl.SteelAxe” />

如果开发者不喜欢使用XML配置文件，Spring3.0允许开发者使用Java类进行配置。

上面的XML配置文件可以替换成如下Java配置类。

package com.langsin.impl;

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

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class AppConfig {

// 配置一个变量personName，其值为：孙悟空

@Value("孙悟空")

String personName;

// 配置一个Bean：Chinese

@Bean(name = "chinese")

public Person person() {

Chinese chin = new Chinese();

chin.setAxe(steelAxe());

chin.setName(personName);

return chin;

}

// 配置一个Bean：Axe

@Bean(name = "steelAxe")

public Axe steelAxe() {

return new SteelAxe();

}

// 配置一个Bean：Axe

@Bean(name = "stoneAxe")

public Axe stoneAxe() {

return new StoneAxe();

}

}

上面的配置文件中使用了Java配置类的三个常用的Spring的Annotation

* @Configuration：用于修饰一个Java配置类。
* @Bean用于修饰一个方法，将该方法的返回值定义成容器中的一个Bean。
* @Value：用于修饰一个Field，用于为该Field配置一个值，相当于配置一个变量。

一旦使用Java配置类来管理Spring容器中的Bean及其依赖关系，此时就需要使用如下方式来创建Spring容器：

ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);

AnnotationConfigApplicationContext类会根据Java配置类来创建Spring容器。该类还提供了一个register方法用于添加Java配置类。

使用Java配置类时，还有如下常用的Annotation。

* @Import：修饰一个Java配置类，用于向当前Java配置类中导入其他Java配置类。
* @Scope：用于修饰一个方法，指定该方法返回的对应的Bean的生命周期。
* @Lazy：用于修饰一个方法，指定该方法返回的对应的Bean是否需要延迟初始化。
* @DependsOn：用于修饰一个方法，指定在初始化该方法对应的Bean之前，先初始化指定的Bean。

对于普通的开发者而言，还是使用XML配置文件管理Bean以及Bean之间的依赖关系更为方便。但这种Java配置类的方式又回退到了Java代码耦合层次，只是将这种耦合集中到一个或多个Java配置类中。这种做法的意义何在？

实际上，Spring提供@Configuration和@Bean并不是为了完全取代XML配置，只是希望它作为XML配置的一种补充。对于Spring框架的开发者来说，Spring框架配置文件的“急剧膨胀”是一个让人头疼的问题，因此Spring框架从2.0开始就不断寻找各种对配置文件“减肥”的方法。

后面所介绍的各种Annotation也都是为了简化Spring配置文件而出现的，但由于Annotation引入时间比较晚，因此在一些特殊的功能的支持上，Annotation还不如XML强大。因此，在目前的多数项目中要么完全使用XML配置方式管理Bean，要么使用以Annotation为主，XML为辅的配置方式管理Bean的配置，想要完全放弃XML配置还是比较难的。

之所以会出现两者共存的情况，主要归纳为三个原因：其一，目前绝大多数采用Spring进行开发的项目，几乎都是基于XML配置方式的，Spring在引入注解的同时，必须保证注解能够与XML和谐共存。其二，由于注解引入较晚，因此功能也没有发展多年的XML强大，对于复杂的配置，注解还难以实现，在一段时间内仍需要XML的配合才能解决问题。其三，Spring的Bean的配置方式与Spring核心模块之间是解耦的，因此，改变配置方式对Spring的框架自身是透明的。Spring可以通过使用Bean后处理器非常方便的增加对于注解的支持。这在技术上非常容易实现。

因此，在实际项目中可能会混合使用XML配置和Java类配置，在这种混合下，以哪种为主都可以。

1. 如果以XML配置为主，就需要让XML配置文件加载Java类配置。例如下面方式：

<context:annotation-config/>

<bean class=”com.langsin.anno.AppConfig”/>

此种情况下需要引入context命名空间。

2、如果以Java类配置为主，就需要让Java配置类能加载XML配置，这就需要借助于@ImportResource注解，如下所示：

@Configuration

@ImportResource(“classpath:/test.xml”)

public class AppConfig{

……….

}

# 2、 深入使用Spring

在上一章中已经介绍了Spring框架的基础内容，详细介绍了Spring容器的核心机制：依赖注入，并介绍了Spring容器对Bean的管理。实际上，上一章介绍的内容是大部分项目的都需要使用的基础部分，很多时候，即使不使用Spring框架，实际项目也都会采用相同的策略。

但Spring框架的功能绝不是只有这些部分，Spring框架允许开发者使用两种后处理器扩展IoC容器，这两种后处理器可以处理IoC容器本身，或者对容器中所有的Bean进行后处理。IoC容器还提供了AOP功能，极好地丰富了Spring容器的功能。

Spring AOP是Spring框架另一个吸引人的地方，AOP本身是一个中非常前言的编程思想，它从动态角度考虑程序运行过程，专门用于处理系统中分布于各个模块中的交叉关注点的问题，能更好地抽离出各个模块的交叉关注点。

Spring的声明式事务管理正是通过AOP来实现的。当然，如果仅仅想使用Spring的声明式事务管理、其实完全无需掌握AOP，但如果希望开发出结构更优雅的应用，例如，集中处理应用的权限控制、系统日志等需求，则应该使用AOP来处理。

除此之外，本章还将详细介绍Spring与JAP框架的整合。

## 2.1 两种后处理器 不讲

Spring框架提供了很好的扩展性，除了可以与各种第三方框架良好整合外，其IoC容器也允许开发者进行扩展，这种扩展甚至无须实现BeanFactory或ApplicationContext接口，而是允许通过两个后处理器对IoC容器进行扩展。Spring提供了两种常用的后处理器。

* + Bean后处理器：这种处理器会对容器中的Bean进行后处理，对Bean进行额外加强。
  + 容器后处理器：这种后处理器对IoC容器进行后处理，用于增强容器功能。

下面将介绍这两种常用的后处理器，以及两种后处理器的相关知识。

### 2.1.1 Bean后处理器

Bean后处理器是一种特殊的Bean，这种特殊的Bean并不对外提供服务，它甚至可以无须id属性，它主要负责对容器中的其他Bean执行后处理，例如为容器中的目标Bean生成代理等，这种Bean被称为Bean后处理器。

Bean后处理器会在Bean实例创建成功之后，对Bean实例进行进一步的增强处理。

Bean后处理器必须实现BeanPostProcessor接口，该接口包含如下两个方法。

* Object postProcessBeforeInitialization(Object bean,String name) throws BeansException：第一个参数是系统即将进行后处理的Bean实例，第二个参数是该Bean的id。
* Object postProcessAfterInitialization(Object bean,String name)，参数同上。

上面两个方法会对容器中指定的Bean进行后处理，会在目标Bean初始化之前、初始化之后分别被回调，这两个方法用于对容器中的Bean实例进行增强处理。

下面将定义一个简单的Bean后处理器，该Bean后处理器将对容器中的其他Bean进行后处理。Bean后处理器代码如下：

代码1：

package com.langsin.beandeal;

import org.springframework.beans.BeansException;

import org.springframework.beans.factory.config.BeanPostProcessor;

public class PostProcessor implements BeanPostProcessor {

@Override

public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {

System.out.println("Bean后处理器在IoC容器初始化Bean："+beanName+"之前对bean进行增强处理。。。。");

return bean;

}

@Override

public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {

if(Worker.class.isInstance(bean)){

Worker worker = (Worker)bean;

worker.setAge("100");

}

System.out.println("Bean后处理器在IoC容器初始化Bean："+beanName+"之后对bean进行增强处理。。。。");

return bean;

}

}

上面类中的两个方法实现了对Bean的增强处理，在Spring容器实例化Bean后，就会依次调用两个方法对Bean进行处理。

下面是Worker类的代码，该类在配置中通过init-method属性指定了初始化之后要执行的方法。

package com.langsin.impl;

public class Worker implements Person{

private Axe axe;

private String age;

private int num;

public Worker(){

}

public void setAxe(Axe axe) {

this.axe = axe;

}

public void setAge(String age) {

this.age = age;

}

public void setNum(int num) {

this.num = num;

}

public Worker(Axe axe,String age){

this.axe = axe;

this.age = age;

}

public Worker(Axe axe,int num){

this.axe = axe;

this.num = num;

}

public void useAxe() {

System.out.println("工人年龄为："+this.age+"，工龄为："+this.num+"在砍树，"+this.axe.chop());

}

}

在配置文件中配置Bean后处理器和配置普通Bean完全一样，但有一点需要指出，在程序开发时基本不需要通过Spring容器获取该Bean后处理器，因此配置时不需要为该Bean后处理器指定id属性值。配置如下：

<?xml version="1.0" encoding="GBK"?>

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

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

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

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

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

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

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

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

<bean id="work" class="com.langsin.impl.Worker" init-method="useAxe" p:axe-ref="steelAxe" p:age="55"/>

<bean class="com.langsin.beandeal.PostProcessor"/>

</beans>

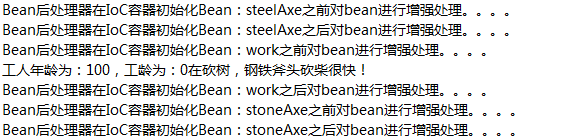
主程序调用如下：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("test.xml");

}

程序运行效果如下：



从上面代码可以看出，该程序无须对Bean后处理器做任何操作，这是因为Spring容器会自动检测容器中所有的Bean，如果发现某个Bean类实现了BeanPostProcessor接口，Spring容器会自动将其注册为Bean后处理器。

通过上面程序的运行可以看出，虽然配置文件通过属性依赖注入的方式为age属性指定了值，但这个值没有发生任何作用，该Worker Bean的age属性被重设置为100，这就是Bean后处理器的作用。

容器中一旦注册了Bean后处理器，Bean后处理器就会自动启动，在容器中每个Bean创建时自动工作，在每个Bean初始化时加入Bean后处理器需要完成的工作。从上面的执行过程可以看出，Bean后处理器两个方法的回调时机如下图所示：



实现BeanPostProcessor接口的Bean后处理器可对Bean进行任何操作，通常用来检查标记接口，或者将Bean包装秤一个Proxy，Spring很多工具类都是通过Bean后处理器完成的。

如果使用BeanFactory作为Spring的容器，则必须手动注册Bean后处理器，程序必须获取Bean的实例，然后手动注册。在这种需求下，就需要为文件中的Bean指定id属性，这样才可以通过Spring容器来进行获取，然后注册。

如下所示：

Resource res = new ClassPathResource(“test.xml”);

DefaultListableBeanFactory ctx = new DefaultListableBeaFactory();

new XmlBeanDefinitionReader(ctx).loadBeanDefinitions(res);

BeanPostProcess bp = (BeanPostProcess)ctx.getBean(“bp”);

ctx.addBeanPostProcess(bp);

### 2.1.2 Bean后处理器的用处

利用Bean后处理器可以完成很多工作，例如生成Proxy。Spring框架本身提供了大量的Bean后处理器，这些后处理器负责对容器中的Bean进行后处理。

下面是Spring提供的两个常用的后处理器。

* BeanNameAutoProxyCreater：根据Bean实例的name属性，创建Bean实例的代理。
* DefaultAdvisorProxyCreator：根据提供的Advisor，对容器中的所有Bean实例创建代理。

上面提供的两个Bean后处理器都用于根据容器中配置的拦截器，创建代理Bean，代理Bean就是对目标Bean进行增强，在目标Bean的基础上进行修改得到新的Bean。

### 2.1.3 容器后处理器

Spring还提供了一种容器后处理器，容器后处理器负责处理容器本身。容器后处理器必须实现BeanFactoryPost

Processor接口，该接口声明了如下一个方法：

* postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)

该方法由Spring容器自动调用，在此方法中可以对Spring容器进行自定义扩展，也可以对Spring容器不做任何处理。

同BeanPostProcessor类似，ApplicationContext可自动检测到容器中的容器后处理器，并且自动注册容器后处理器。但如果使用BeanFactory作为Spring容器，则必须手动调用该容器后处理器来处理BeanFactory容器。

实现了BeanFactoryPostProcessor接口的容器后处理器不仅可以对BeanFactory执行后处理，也可以对ApplicationContext容器执行后处理。容器后处理器还可以用来注册额外的属性编辑器。

Spring提供了如下几个常用的容器后处理器：

* PropertyPlaceholderConfigurer：属性占位符配置器。
* PropertyOverrideConfigurer：重写占位符配置器。
* CustomAutoWrieConfigurer：自定义自动装配的配置器。
* CustomScopeConfigurer：自定义作用域的配置器。

容器后处理器通常用于对于Spring容器进行处理，并且总是在容器实例化其他Bean之前，读取配置文件的元数据，也可以修改这些元数据。

在Spring的配置文件中可以配置多个容器后处理器，在每个容器后处理器中可使用order属性来控制容器后处理器的执行次序。容器后处理器的作用域范围是容器级，它只能对容器本身进行处理，而不对容器中的Bean进行处理。

### 2.1.4 属性占位符配置器

PropertyPlaceholderConfigurer是一个容器后处理器，负责读取属性文件里的属性值，并将这些属性设置成Spring配置文件的数据。例如配置数据库连接，或者配置数据库连接池，只需要将特定的信息，比如连接数据库的URL、用户名、密码等放在特定的属性文件中，当切换数据库时，只需要改属性文件即可，则无需改Spring文件。

如下示例，配置PropertyPlaceholderConfigurer容器后处理器，使用了属性文件中属性值。

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<list>

<value>conn.properties</value>

</list>

</property>

</bean>

<bean id="dbdataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close">

**<property name="driverClassName" value="${driver}" />**

**<property name="url" value="${url}" />**

**<property name="username" value="${username}"/>**

**<property name="password" value="${password}" />**

<property name="maxTotal" value="200" />

<property name="maxIdle" value="200"/>

<property name="minIdle" value="2"/>

<property name="initialSize" value="2" />

</bean>

在上面配置文件中，配置driverClassName、url等信息时，并未直接设置这些属性的属性值，而是设置了${driver}和${url}属性值，这表明Spring容器将从propertyConfigurer指定的属性文件中搜索这些key对应的value，并为该Bean的属性值设置这些value值。

如果导入了context命名空间，则可以使用如下方式来配置该属性占位符。

<context:property-placeholder location=”classpath:conn.properties” />

注意：使用此种方式时，需要将属性local-override赋值为"true"，作用覆盖掉Spring容器中预初始化的本地相关的一些元数据。本地元数据中含有一个key为username的属性，在Spring容器中的容器后处理完成后，拿到的是系统用户名称，如果资源文件conn.properties中也指定了username，则此时的值不会把Spring中已有的元数据覆盖掉，因为local-override属性默认为false，如果指定为true，则可以覆盖。或者将属性文件conn.properties中的key值不要使用username这个不是关键字的类似关键字。

### 2.1.5 重写占位符配置器

PropertyOverrideConfigurer是Spring提供的另一个容器后处理器，这个容器后处理器比PropertyPlaceholder

Configurer更加强大。PropertyOverrideConfigurer的属性文件指定的信息可以直接覆盖Spring配置文件中的元数据。

举例说明：假如Bean A中有个属性name，Spring容器为其注入的值为“zhangsan”，PropertyOverrideConfigurer的属性文件中name的值为“lisi”，那么属性文件中的name属性会覆盖掉Spring配置文件中注入的值。

使用PropertyOverrideConfigurer的属性文件，每条属性应该使用如下格式：

beanId.property=value 针对于下面的work示例，属性文件中的书写格式为：worker.age=55岁

beanId是属性占位符试图覆盖的Bean的id，property是试图覆盖的属性的属性名。例如：对应下面的配置：

<bean id=”worker” class=”com.langsin.impl.Worker”>

<property name=”age” value=”100岁” />

</bean>

属性文件中属性key的格式必须为：worker.age=100岁，此属性文件中所有的key-value对都必须为此格式！

<bean class="org.springframework.beans.factory.config.PropertyOverrideConfigurer">

<property name="locations">

<list>

<value>conn.properties</value>

</list>

</property>

</bean>

<bean id="worker" class="com.langsin.impl.Worker">

<property name="age" value="100" />

</bean>

虽然上面的Spring的配置文件中为id为worker的Bean的age属性依赖注入的值为100，但因为Spring容器中部署了一个PropertyOverrideConfigurer后处理器，此处理器会读取conn.properties文件中的属性，用于覆盖目标Bean的属性。所以对于age属性，此时的age属性不会由Spring容器来决定了。

conn.properties文件的内容为：

worker.age=55

运行下面主程序：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("test.xml");

Worker worker = ctx.getBean("worker", Worker.class);

System.out.println(worker.getAge());

}

执行效果输出为：55。

如果导入了context命名空间，则可以使用如下方式来配置该属性占位符。如下所示：

<?xml version="1.0" encoding="GBK"?>

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

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

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

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

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

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

http://www.springframework.org/schema/beans/spring-beans-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/util

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

<context:property-override location="classpath:conn.properties"/>

<bean id="worker" class="com.langsin.impl.Worker">

<property name="age" value="100" />

</bean>

</beans>

## 2.2 Spring的“零配置”支持

Java是一种跨平台的语言，而XML是跨平台的数据交换格式，所以Java和XML组合应该是最完美的。正是在这种环境下，以前的Java框架都不约而同的选择使用XML作为配置文件。

时至今日，也许是Rails框架的启发，现在的Java框架又都开始对XML配置方式“弃之不顾”，几乎所有的主流Java框架都打算支持“零配置”特性，包括Struts2、Hibernate，以及现在介绍的Spring，都开始支持使用Annotation来代替XML配置文件。

扩展内容：Rails框架首次提出是在2004年7月，它的研发者是26岁的丹麦人David Heinemeier Hansson。不同于已有复杂的Web 开发框架，Rails是一个更符合实际需要而且更高效的Web开发框架。Rails结合了PHP体系的优点（快速开发）和Java体系的优点（程序规整），因此，Rails在其提出后不长的时间里就受到了业内广泛的关注。

提示：Rails框架的处理比较简单，它采用一种所谓的“约定优于配置”的方式，它要求将不同组件放在不同的路径下，而Rails框架中是加载固定路径下的所有组件。

### 2.2.1 搜索Bean类

既然不再使用Spring配置文件来配置任何Bean实例，那么只能由Spring自动搜索某些路径下的Java类，并将这些Java类注册成Bean实例。

Spring没有采用“约定优于配置”的策略，Spring依然要求程序员显示指定搜索哪些路径下的Java类，Spring将会把合适的Java类全部注册成Spring Bean。问题是：Spring如何知道应该把哪些Java类当成Bean类处理？这就需要使用Annotation，Spring通过使用一些特殊的Annotation来标注Bean类。如下所示：

@Component：标注一个普通的Spring Bean类。

@Controller：标注一个控制器组件类。

@Service：标注一个业务逻辑组件类。

@Repository：标注一个DAO组件类。

指定了某些类可以作为Spring Bean类使用后，最后还需要让Spring搜索这些Bean的路径，此时需要配置文件中导入context命名空间，并指定一个简单的搜索路径。

@Component(“chine”) //在指定Bean时，同时为Bean定义一个名称

public class Chinese implements Person{

}

@Component(“steelAxe”)

public class SteelAxe implements Axe{

}

@Component(“stoneAxe”)

public class StoneAxe implements Axe{

}

这些Java类与前面介绍的Bean类没有区别，只是每个Java类都使用了Component标注，表明这些Java类都将作为Spring的Bean类，同时给Bean实例进行了命名。

下面需要在Spring配置文件中指定搜索路径，Spring将会自动搜索该路径下的所有Java类，并根据这些Java类来创建Bean实例。如下配置所示：

<?xml version="1.0" encoding="GBK"?>

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

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

**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/context**

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

<!—自动扫描指定包及其子包下的所有的Bean类 -->

**<context:component-scan base-package=”com.langsin.anno” />**

</beans>

运行下面主程序：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("test.xml");

System.out.println(Arrays.toString(ctx.getBeanDefinitionNames()));

}

上面程序执行后会输出Spring容器中所有的Bean实例的名称，例如chine、steelAxe，如果没有为Bean实例指定名称，Spring则默认以Bean类的首字母小写作为Bean实例名称。

在默认情况下Spring会自动搜索以@Component、@Controller、@Service和@Repository标注的Java类，并将它们当成Spring Bean来处理。

### 2.2.2 指定Bean的作用域

当采用零配置方式来管理Bean实例时，可以使用@Scope Annotation来指定Bean实例的作用域。例如如下Java类代码：

@Scope(“prototype”)

@Component(“chine”)

public class Chinese implements Person{

}

### 2.2.3 使用@Resource配置依赖

@Resource位于javax.annotation包下，是来自于Java EE规范下的一个Annotation，Spring直接借鉴了该Annotation，通过使用该Annotation为目标Bean指定协作者Bean。

@Resource有一个name属性，在默认情况下，Spring将这个值解释为需要被注入的的Bean实例的id。如下示例所示：

@Component

@Scope(“singleton”)

public class Chinese implements Person{

private Axe axe;

**@Resource(name=”steelAxe”)**

public void setAxe(Axe axe){

this.axe = axe;

}

}

@Resource不仅可以修饰setter方法，也可以修饰实例变量，如果使用@Resource修饰实例变量将会更加简单，此时Spring将会直接使用Java EE规范的Field注入，连setter方法都不需要。如下所示：

@Component

@Scope(“singleton”)

public class Chinese implements Person{

**@Resource(name=”steelAxe”)**

private Axe axe;

}

### 2.2.4 使用@PostConstruct和@PreDestroy定制声明周期行为

@PostConstruct和@PreDestroy同样位于javax.annotation包下，也是来自于Java EE规范的两个Annotation，Spring直接借鉴了它们，用于定制Spring容器中Bean的声明周期行为。

前面介绍过Spring生命周期时提供了bean元素的两个属性init-method、destory-method，一个是指定Bean初始化后，调用指定的方法增强初始化。一个是指定Spring容器在销毁Bean之前调用的方法，完成一些处理。

@PostConstruct和@PreDestroy两个注解的作用大致与此相似，它们都用于修饰方法，无须任何属性。如下所示：

@Component

@Scope(“singleton”) //不使用Scope默认为单例行为

public class Chinese implements Person{

@Resource(name=”steelAxe”)

private Axe axe;

public void useAxe(){

System.out.println(this.axe.chop());

}

@PostConstruct

public void init(){

System.out.println(“正在执行初始化的init方法……..”);

}

@PreDestroy

public void close(){

System.out.println(“正在执行销毁之前的close方法……..”);

}

}

## 2.3 资源访问 不讲

创建Spring容器时通常需要访问XML配置文件，除此之外，程序可能有大量地方需要访问各种类型的文件、二进制流等——Spring把这些文件、二级制流等统称为资源。

在Sun提供的标准的API里，资源访问通常由java.net.URL和文件IO来完成，如果需要访问来自网络的资源时，则通常会选择URL类。

URL类可以处理一些常规的资源访问问题，但不能很好的满足所有底层资源访问的需要，比如：暂时还无法在类加载路径或相对application的路径访问资源，而且URL接口功能也不完善，比如检查所指向的资源是否存在就不能实现。

Spring改进了Java资源访问的策略，Spring为资源访问提供了一个Resource接口，该接口提供了更强的资源访问能力。Spring框架本身大量使用了Resource来访问底层资源。

Resource本身是一个接口，是具体资源访问策略的抽象，也是所有资源访问类所实现的接口。Resource接口主要提供了如下几个方法。

* getInputStream()：定位并打开资源，返回资源对应的输入流。每次调用都返回新的输入流。调用者必须负责关闭输入流。
* exists()：返回Resource所指向的资源是否存在。
* isOpen()：返回资源文件是否打开，如果资源文件不能多次读取，每次读取结束时应该显示关闭，以防止资源泄露。
* getDescription()：返回资源的描述信息，用于资源处理出错时输出信息，通常是全限定文件名或实际URL。
* getFile()：返回资源对应的File对象。
* getURL()：返回资源对应的URL对象。

最后两个方法通常无须使用，仅在通过简单方式访问无法实现时，Resource才提供传统的资源访问功能。

Resource接口本身没有提供任何底层资源的实现逻辑，针对不同的底层资源，Spring将会提供不同的Resource实现类，不同的实现类负责不同的资源访问逻辑。

Resource不仅可以在Spring的项目中使用，也可以直接作为资源访问的工具类使用。也就是说：即使不使用Spring框架，也可以使用Resource工具类。

### 2.3.1 Resource实现类

Resource接口时Spring框架资源访问的接口，具体的资源访问由该接口的实现类完成。Spring提供了Resource接口的大量的实现类。

* UrlResource：访问网络资源的实现类。
* ClassPathResource：访问类加载路径里的资源的实现类。
* FileSystemResource：访问文件系统里资源的实现类。
* ServletContextResource：访问相对于ServletContext路径下的资源的实现类。
* InputStreamResource：访问输入流资源的实现类。
* ByteArrayResource：访问字节数组资源的实现类。

针对不同的底层资源，这些Resource实现类提供了相应的资源访问逻辑，并提供便捷的包装，以利于客户端程序的资源访问。

#### 1、访问网络资源

访问网络资源通过UrlResource类实现，UrlResource是URL类的包装，主要用于是先让URL类访问一个资源对象，然后再由它去访问该资源对象进行处理。URL资源通常应该提供标准的协议前缀。例如：file，用于访问文件系统；http，用于通过HTTP协议访问资源；ftp，用于通过FTP协议访问资源等。

UrlResource类实现了Resource接口，对Resource的全部方法提供了实现，完全支持Resource的全部API，下面的代码示范了使用UrlResource访问文件系统资源的示例。

public static void main(String[] args) throws Exception {

**UrlResource resource = new UrlResource("file:src/test.xml");**

System.out.println(resource.getFilename());

System.out.println(resource.getDescription());

System.out.println(resource.exists());

}

上面程序中的粗体字代码使用UrlResource来访问本地磁盘资源，虽然UrlResource是为访问网络资源而设计的，但通过使用file:前缀也可访问本地磁盘资源，如果需要访问网络资源，则可以使用如下两个常用的前缀。

* http：用于访问基于HTTP协议的网络资源。
* ftp：用于访问基于FTP协议的网络资源。

由于UrlResource是对URL的封装，所以两个类支持的前缀应该完全相同。对于使用file:前缀访问的本地的资源文件，该文件的位置应该在应用的当前路径下。

#### 2、访问类加载路径下的资源

ClassPathResource用来访问类加载路径下的资源，相对于其他的Resource实现类，该类的主要优势是方便访问类加载路径下的资源，尤其对于Web应用，ClassPathResource可自动搜索位于WEB-INF/classes下的资源文件，无须使用绝对路径访问。

**ClassPathResource resource = new ClassPathResource(“classpath:test.xml”);**

System.out.println(resource.getFilename());

System.out.println(resource.getDescription());

System.out.println(resource.exists());

上面粗体代码用于访问类加载路径下的test.xml文件，对比前面的示例程序，两个程序除了资源访问的代码有所区别之外其他代码基本一致，这就是Spring资源访问的优势：Spring的资源访问消除了底层资源访问的差异，允许以一致的方式来访问不同的底层资源。

ClassPathResource实例可以使用构造器显示的创建，但更多的时候都是隐式创建，当执行Spring的某个方法时，该方法接受一个代表资源路径的字符串参数，当Spring识别该字符串参数中包含classpath:前缀后，系统将自动创建ClassPathResource。

#### 3、访问文件系统资源

Spring提供的FileSystemResource类用于访问文件系统资源。使用FileSystemResource来访问文件系统资源并没有太大的优势，因为Java提供的File类也可以用于访问文件系统资源。

使用FileSystemResource也可消除底层资源访问的差异，程序通过统一的Resource API来进行资源访问。如下所示：

// 默认从文件系统的当前路径加载book.xml资源

FileSystemResource res = new FileSystemResource(“src/test.xml”);

// 获取资源的简单信息

System.out.println(resource.getFilename());

System.out.println(resource.getDescription());

System.out.println(resource.exists());

与前两种使用Resource进行资源访问的区别在于：资源字符串确定的资源，位于本地系统内，而且无须使用任何前缀。

#### 4、访问应用相关资源

Spring提供了ServletContextResource类来访问Web Context下的资源，ServletContextResource构造器接受一个代表资源位置的字符串参数，该资源位置是相对于Web应用跟路径的位置。

使用ServletContextResource访问的资源，也可以通过文件IO访问或者URL访问。通过File访问时要求资源必须是被解压出来的，而且在本地系统文件中的；但使用ServletContextResource进行访问时无须关心资源是否被解压缩出来还是在JAR文件中，都可通过Servlet访问。

下面把test.xml文件放在Web应用中的WEB-INF路径下，然后通过JSP页面来直接访问该test.xml文件。需要指出的是，在默认情况下，JSP不能直接访问WEB-INF路径下的任何资源。所以应用中的JSP需要使用Servlet

ContextResource来访问该资源。

<body>

<%

**ServletContextResource src = new ServletContextResource(application,"WEB-INF/web.xml");**

SAXReader reader = new SAXReader();

Document doc = reader.read(src.getFile());

Element root = doc.getRootElement();

out.println("<h3>"+root.getName()+"</h3>");

List<?> list = root.elements();

for(int i=0;i<list.size();i++){

Element el = (Element)list.get(i);

out.println("<h3>"+el.getName()+"</h3>");

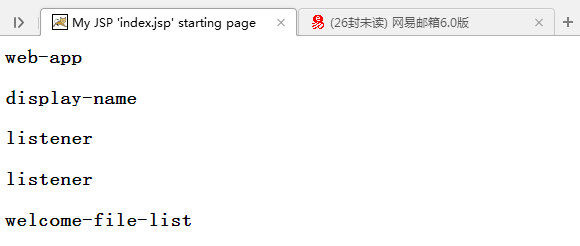
}

%>

</body>

上面程序中的粗体字代码指定应用从Web Context下的WEB-INF路径下读取web.xml配置文件，并且将内容在JSP页面上进行输出展示。

启动Tomcat，再打开浏览器访问该JSP页面，执行效果如下：



### 2.3.2 ResourceLoader接口和ResourceLoaderAware接口

前面介绍了实现不同策略的资源Resource的实现类，下面来看如何直接通过Spring容器来获取资源访问对象。Spring提供了如下两个标志性接口。

* ResourceLoader：该接口实现类的实例可以获得一个Resource实例。
* ResourceLoaderAware：该接口实现类的实例将获得一个ResourceLoader的引用。

在ResourceLoader接口中声明了如下方法：

* Resource getResource(String location)：该接口仅包含这个方法，此方法用于返回一个Resource的实例对象。ApplicationContext的实现类都实现了ResourceLoader接口，因此ApplicationContext可用于直接获取Resource实例。

ApplicationContext实例获取Resource实例时，默认采用与ApplicationContext相同的资源访问策略，如下：

Resource res = ctx.getResource(“path/path/test.xml”);

从此代码无法确定Spring容器使用哪个实现类来访问指定的资源，Spring采用和容器相同的策略来访问资源。即：如果ApplicationContext是FileSystemXmlApplicationContext，res就是FileSystemResource；如果ApplicationContext是ClassPathXmlApplicationContext，res就是ClassPathResource；如果ApplicationContext是XmlWebApplicationContext，res就是ServletContextResource实例。

从上面程序可以看出，当Spring应用需要进行资源访问时，实际上并不需要直接使用Resource实现类，而是调用Spring容器的getResource()方法来获取资源。ResourceLoader将会负责选择Resource的实现类，也就是确定具体的资源访问策略。

**ApplicationContext ctx = new ClassPathXmlApplicationContext(“beans.xml”);**

**//ApplicationContext ctx = new FileSystemXmlApplicationContext(“src/beans.xml”);**

**Resource res = ctx.getResource(“test.xml”);**

System.out.println(res.getFilename());

System.out.println(res.getDescription());

执行结果如下：

beans.xml

class path resource [beans.xml]

切换Spring的容器类，执行效果如下：

beans.xml

file [D:\MyEclipse\workspaces\spring-test\src\test.xml]

从执行结果可以看出，程序的Resource实现类发生了改变，变为使用FileSystemResource实现类。

以下是常见的前缀及对应的访问策略。

* classpath：以ClassPathResource实例访问类加载路径下的资源。
* file：以UrlResource实例访问本地系统的资源，默认以当前应用为根路径。
* http：以UrlResource实例访问基于HTTP协议的网络资源。
* 无前缀：由ApplicationContext的实现类决定访问策略。

ResourceLoaderAware完全类似于Spring提供的BeanFactoryAware、BeanNameAware接口，ResourceLoaderAware接口也提供了一个setResourceLoader()方法，该方法由Spring容器负责调用，Spring容器会将一个ResourceLoader对象作为参数传递给此方法。

如果把实现ResourceLoaderAware接口的Bean类部署在Spring容器中，Spring容器会将自身当成Resource

Loader作为setResourceLoader()方法的参数传入。由于ApplicationContext的实现类都实现了ResourceLoader接口，Spring容器自身完全可以当成ResourceLoader使用。

下面通过示例验证实现ResourceLoaderAware接口得到的ResourceLoader与通过Spring容器得到的是否为同一个：

示例1：

package com.langsin.resource;

import org.springframework.context.ResourceLoaderAware;

import org.springframework.core.io.ResourceLoader;

public class TestBean implements ResourceLoaderAware{

private ResourceLoader loader = null;

@Override

public void setResourceLoader(ResourceLoader loader) {

this.loader = loader;

}

public ResourceLoader getLoader(){

return this.loader;

}

}

示例2：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new FileSystemXmlApplicationContext("src/test.xml");

TestBean tb = ctx.getBean("loader", TestBean.class);

ResourceLoader loader = tb.getLoader();

System.out.println(loader==ctx);

}

### 2.3.3 使用Resource作为属性

前面介绍的Spring提供的资源访问策略，要么直接使用Resource的实现类，要么通过Spring容器来获取资源。除此之外，Spring还提供了一种更为简单的方式来访问资源。即：当应用程序中的Bean实例需要访问资源时，只需要将Resource定义为成员变量，通过Spring容器直接进行设值注入。

示例1：

package com.langsin.resource;

import java.util.List;

import org.dom4j.Document;

import org.dom4j.Element;

import org.dom4j.io.SAXReader;

import org.springframework.core.io.Resource;

public class TestBean{

private Resource resource = null;

public void setResource(Resource resource){

this.resource = resource;

}

public void parseXml() throws Exception{

SAXReader reader = new SAXReader();

Document doc = reader.read(this.resource.getFile());

List<?> list = doc.getRootElement().elements();

for(int i=0;i<list.size();i++){

Element el = (Element)list.get(i);

System.out.println(el.getName());

}

}

}

示例2：

<?xml version="1.0" encoding="GBK"?>

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

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

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

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

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

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

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

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

<bean id="work" class="com.langsin.impl.Worker" />

<bean id="loader" class="com.langsin.resource.TestBean" p:resource="classpath:test.xml" />

</beans>

示例3：主程序如下：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new FileSystemXmlApplicationContext("src/test.xml");

TestBean tb = ctx.getBean("loader", TestBean.class);

tb.parseXml();

}

采用依赖注入，允许动态配置资源文件位置，无须将资源文件位置写在代码中，当资源文件位置发生变化时，无须改动程序，直接修改配置文件即可。

### 2.3.4 在ApplicationContext中使用资源

不管以怎样的方式创建ApplicationContext实例，都需要为ApplicationContext指定配置文件，Spring允许使用一份或多份XML配置文件。

当程序创建ApplicationContext实例时，通常也是以Resource的方式来访问配置文件，所以ApplicationContext完全支持ClassPathResource、FileSystemResource、ServletContextResource等资源访问方式。ApplicationContext确定资源访问策略通常有两种方式。

* 使用ApplicationContext实现类指定访问策略。
* 使用前缀指定访问策略。

#### 1、使用ApplicationContext实现类指定访问策略

创建ApplicationContext对象时，通常可以使用如下三个实现类。

* ClassPathXmlApplicationContext：对应使用ClassPathResource进行资源访问。
* FileSystemXmlApplicationContext：对应使用FileSystemResource进行资源访问。
* XmlWebApplicationContext：对应使用ServletContextResource进行资源访问。

当使用ApplicationContext的不同实现类时，就意味着Spring使用相应的资源访问策略。

#### 2、使用前缀指定访问策略

Spring也允许使用前缀来指定资源访问策略，例如如下代码所示：

ApplicationContext ctx = new FileSystemXmlApplicationContext(“classpath:test.xml”);

虽然上面的代码采用了FileSystemXmlApplicationContext实现类，但程序依然从类加载路径下搜索test.xml配置文件，而不是从本地文件系统的当前路径下搜索。相应的，还可以使用http:、ftp:等前缀，用来确定对应的资源访问策略。

public static void main(String[] args) throws Exception {

**ApplicationContext ctx = new FileSystemXmlApplicationContext("classpath:test.xml");**

System.out.println(ctx);

**Resource res = ctx.getResource("beans.xml");**

System.out.println(res.exists());

}

上面程序中的粗体字代码创建Spring容器时，系统将从类加载路径下搜索test.xml；但是第二行粗体代码使用Spring容器获取访问资源时，依然采用的是FileSystemResource实现类，这与FileSystemXmlApplicationContext的访问策略是一致的。这表明：通过classpath:前缀指定资源访问策略仅仅对当次访问有效，程序后面进行资源访问时，还是会根据Spring容器的实现类来选择对应的资源访问策略。

因此，如果程序需要使用Spring容器访问资源，建议采用对应的Spring容器实现类来加载配置文件，而不是通过前缀来指定资源访问策略。

#### 3、classpath\*:前缀的用法

classpath\*:前缀提供了加载多个XML配置文件的能力，当使用classpath\*:前缀来指定XML配置文件时，系统将搜索类加载路径，找出所有与文件名匹配的文件，分别加载文件中的配置定义，最后合并成一个Application

Context。如下代码所示：

ApplicationContext ctx = new FileSystemXmlApplicationContext("classpath\*:beans.xml");

将配置文件beans.xml分别放在应用的classes路径下，再将另一个beans.xml放在classes/xml文件夹下，程序实例化时会将两个配置文件都加载到，如果放入的Spring配置文件名称不是beans.xml的，容器不加载，因为前缀中指定的只加载beans.xml文件。

另外，还有一种可以一次性加载多个配置文件的方式，即：指定配置文件时使用统配符。如下所示：

ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:test\*.xml");

上面代码指定从类加载路径搜索配置文件，且搜索所有以test开头的XML配置文件，也可以写成\*.xml将表示搜索类加载路径下的所有的XML配置文件。

ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath\*: \*.xml");

上面代码表示搜索类加载路径下，以及子路径下的所有的配置文件。

## 2.4 Spring的AOP

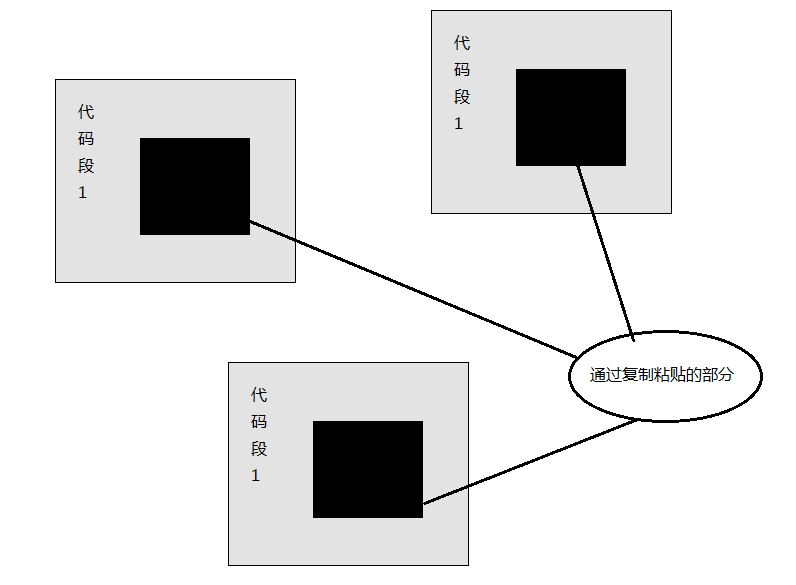
AOP（Aspect Orient Programming），也就是面向切面编程，作为面向对象编程的一种补充，AOP已经成为一种比较成熟的编程方式。其实AOP的发展时间并不是太长，甚至在国内的翻译也不太统一。AOP和OOP互为补充，OOP将程序分解成各个层次的对象，而AOP将程序运行过程分解成各个切面。可以这样理解：OOP是从静态角度考虑程序结构，而AOP是从动态角度考虑程序运行过程。

### 2.4.1 为何需要AOP

在传统的OOP编程中以对象为核心，整个软件由一系列相互依赖的对象组成，而这些对象将被抽象成一个个类，并允许使用类继承来管理类与类之间一般到特殊的关系。随着软件规模的增大，应用的逐渐升级，慢慢出现了一些OOP很难解决的问题。

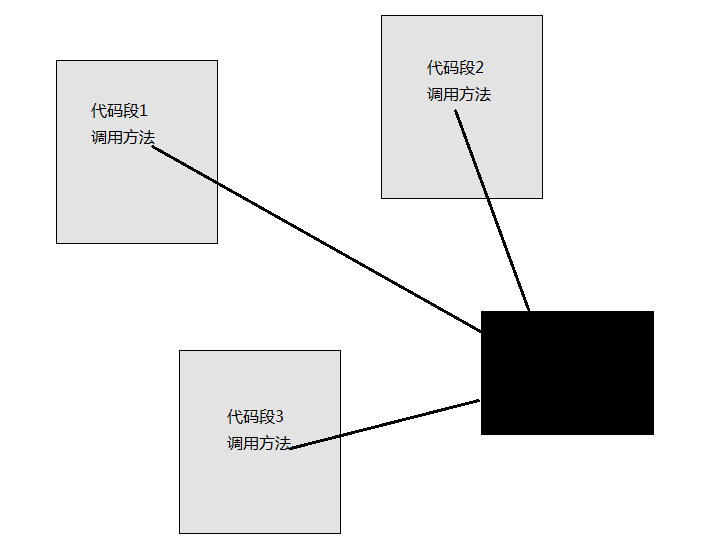
面向对象可以通过分析，然后抽象出一系列具有一定属性与行为的类，并通过这些类之间的协作来形成一个完整的软件功能。由于类可以继承，因此可以把具有相同功能或相同特性的属性抽象到一个层次分明的类结构体系中。随着软件规范的不断扩大，专业化分工越来越细致，以及OOP应用实践的不断增多，随之也暴漏出现了一些OOP无法很好解决的问题。

假设系统中有三段完全相同的代码，这些代码通常会采用“复制”、“粘贴”的方式来完成，通过这种方式开发出来的软件示意图如下所示：



采用上图方式实现的软件结构，在软件开发期可能会觉得无所谓，但如果某一天需要需更改深颜色所代表的代码部分时，这就意味着需要打开3份源码，甚至更多去修改，那工作将会变的非常繁杂。

为了解决这个问题，通常会将上图中深色代码部分定义成一个方法，然后在三个代码段中分别调用该方法即可。这这种方式下，软件系统的结构如下图所示：



采用上图所示的软件系统，如果需要修改深色代码部分，只要修改一个地方即可。不管整个系统中有多少个地方调用了该方法，程序无须修改这些地方，只需要修改被调用的方法即可，通过这种方式，大大降低了软件后期维护的复杂度。

对于上图所示的三个地方，依然需要显示调用深色代码表示的方法，这样做能够解决大部分应用场景。如果程序希望实现更好的解耦，即：希望代码段1、2、3无须直接调用深颜色方法，而让它执行，该如何解决？

因为软件系统需求变更是很频繁的事情，系统前期设计方法1、方法2、方法3时只是实现了核心业务功能，过了一段时间，可能需要为方法1、方法2、方法3都增加事务控制；又过了一段时间，客户提出方法1、方法2、方法3需要进行用户合法性验证，只有合法的用户才能执行这些方法；有过了一段时间，客户提出方法1、方法2、方法3应该增加日志记录；又过了一段时间，客户提出……面对这样的情况，应该怎么处理？通常两种做法：

* 根据需求说明书，直接拒绝客户要求。
* 拥抱需求，满足客户的需求。

第一种做法显然不好，客户是上帝，开发者应该尽量满足客户的需求。通常会采用第二种做法，那么该如何解决？是不是每次都先定义一个新方法，然后修改方法1、方法2、方法3的源代码，增加调用新方法？这样做的工作量也不小。此时就希望有一种特殊的方法：只要是实现新的方法，然后无须在方法1、方法2、方法3中显示调用它，系统会“自动”在方法1、方法2、方法3中调用这个特殊的新方法。

在编程过程中，没有所谓自动的事情，任何事情都是代码驱动的。这里的自动是指无须开发者关系，有系统来驱动。

上面的想法听起来很神奇，甚至有一些不切合实际，但其实完全可以实现，实现这个需求的技术就是AOP。AOP专门用于处理系统中分布于各个模块中的交叉关注点的问题，在Java EE应用中，常常通过AOP来处理一些具有横切性质的系统级服务，如事务管理、安全检查、缓存、对象池管理等，AOP已经成为一种非常常用的解决方案。

### 2.4.2 使用AspectJ实现AOP

AspectJ是一个基于Java语言的AOP框架，提供了强大的AOP功能，其他很多AOP框架都借鉴或采纳其中的一些思想。由于Spring4.0的AOP与AspectJ进行了很好的集成，因此掌握AspectJ是学习Spring AOP的基础。

AspectJ是Java语言的一个AOP实现，其主要包括两部分：一个部分定义了如何表达、定义AOP编程中的语法规范，通过这套语法规范，可以方便地用AOP来解决Java语言中存在的交叉关注点的问题；另一个部分是工具部分，包括编译器，调试工具等。

AspectJ是最早的、功能比较强大的AOP实现之一，对整套AOP机制都有较好的实现，很多其他的语言的AOP实现，也借鉴或采纳了AspectJ中很多设计。在Java领域，AspectJ中的很多语法结构基本上已经成为AOP领域的标准。

从Spring2.0开始，Spring AOP已经引入了对AspectJ的支持，并允许直接使用AspectJ进行AOP编程，而Spring自身的AOP API也努力与AspectJ保持一致。因此，学习Spring AOP必然需要从AspectJ开始，因为它是Java领域最流行的AOP解决方案。即使不使用Spring框架，也可以直接使用AspectJ进行AOP编程。

AspectJ是Eclipse下面的一个开源子项目，其最新的版本为1.9.0版本为jdk1.9发布。我们下载的是AspectJ1.8.8版本，是2016年发布的版本。

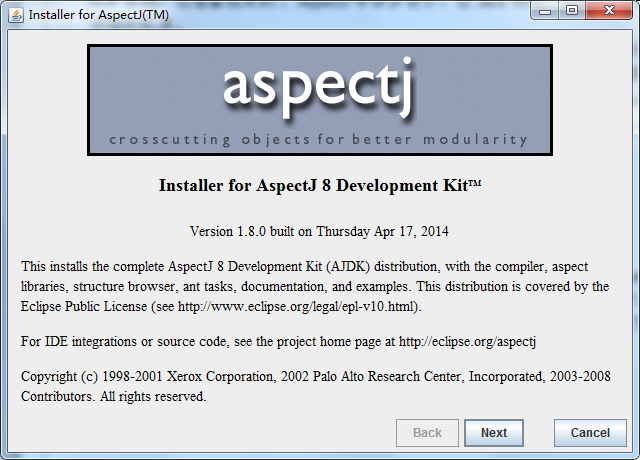
#### 1、下载和安装AspectJ

下载和安装AspectJ按如下步骤：

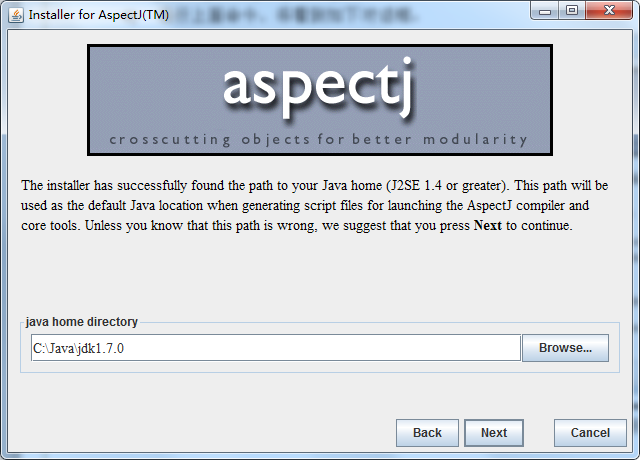
1. 登录<http://www.eclipse.org/aspectj/downloads.php>站点，下载AspectJ的最新稳定版本
2. 下载完成后得到一个aspectj-1.8.0.jar文件，文件名中的1.8.0表示AspectJ的版本号。
3. 启动命令行窗口，进入aspectj-1.8.0.jar文件所在的路径，输入如下命令：

java -jar aspectj-1.8.0.jar

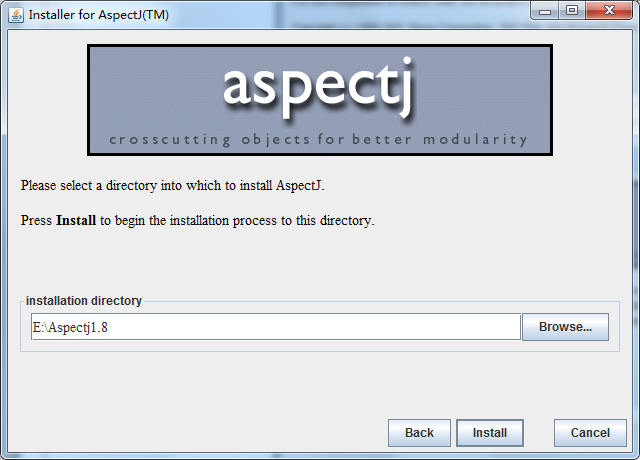
1. 运行上面命令，将看到如下对话框：



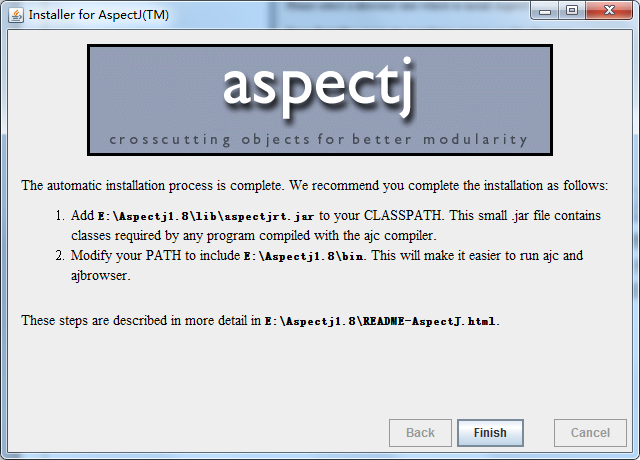
1. 点击“Next”按钮，出现如下对话框，该对话框用于选择JDK的安装路径。



1. 如果JDK安装路径正确，点击“Next”按钮；否则应该通过“Browse”按钮来选择JDK安装路径。然后点解“Next”按钮，出现如下对话框。用于选择AspectJ的安装路径。



1. AspectJ是“纯绿色”软件，安装AspectJ的实质是解压一个压缩包，并不需要向Windows注册表，系统路径里添加任何“垃圾”信息，因此保留AspectJ安装后的文件夹，即使以后重装系统，AspectJ也不会受任何的影响。点击“Install”按钮，程序开始安装AspectJ，安装结束后出现如下对话框，再点击“Next”按钮，弹出安装完成对话框。



1. 安装完成AspectJ之后，系统还应该将E:\AspectJ1.8\bin路径添加到PATH环境变量中，将E:\AspectJ\lib\

aspectjrt.jar添加到CLASSPATH环境变量中。

#### 2、AspectJ使用入门

成功安装了AspectJ之后，将会在E:\AspectJ1.8路径下看到如下文件结构。

* bin：该路径下存放了aj、aj5、ajc、ajdoc、ajbrowser等命令，其中ajc命令最常用，它的作用类似于javac，用于对普通的Java类进行编译时增强。
* docs：该路径下存放了AspectJ的使用说明、参考手册、API文档等内容。
* lib：该路径下的4个JAR文件是AspectJ的核心类库。
* 相关授权文件。

AspectJ的用法非常简单，类似于JDK编译、运行Java程序一样。下面通过一个简单的程序来示范AspectJ的用法。

首先编写两个简单的Java类，这两个Java类用于模拟系统中的业务组件，实际上无论多少个类，AspectJ的处理方式都是一样的。

public class Hello{

public void foo(){

System.out.println(“执行Hello组件的foo()方法”);

}

public int addUser(String name,String pass){

System.out.println(“执行Hello组件的addUser()方法添加用户：”+name);

return 20;

}

}

另一个组件类：

public class World{

public void bar(){

System.out.println(“执行World组件的bar()方法”);

}

}

下面使用一个主程序来模拟系统调用两个业务组件的三个业务方法。

public class AspectJTest{

public static void main(String[] args){

Hello hello = new Hello();

hello.foo();

hello.addUser(“孙凯”,”123456”);

World wor = new World();

wor.bar();

}

}

使用最原始的javac.exe命令来编译这三个源程序，然后使用java.exe命令来执行AspectJTest类，执行结果没有任何悬念，程序显示如下：

**执行Hello组件的foo()方法**

**执行Hello组件的addUser()方法添加用户：孙凯**

**执行World组件的bar()方法**

假设现在用户要求在执行所有业务方法之前执行权限检查，如果使用传统的编程方式，开发者必须先定义一个权限检查的方法，然后由此打开每个业务方法，并修改业务方法的源代码，增加调用权限检查的方法——但这种方式需要对所有业务组件的每个业务方法都进行修改，因此不仅容易引入新的错误，而且维护成本相当大。

如果使用AspectJ的AOP支持，则只需要添加如下特殊的“Java类”即可。

public aspect AuthAspect{

//指定在执行com.langsin.test包中任意类的任意方法之前执行下面代码

//第一个星号表示返回值不限；第二个星号表示类名不限

//第三个星号表示方法名不限，圆括号中..两个点表示任意个数、类型不限的形参

**before(): execution(\* com.langsin.test.\*.\*(..))**{

System.out.println(“模拟进行权限检查…”);

}

}

通过上面程序可以发现，类文件中不是使用class、interface、enum在定义Java类，而是使用aspect，但并不表示Java又新增了关键字。上面的AuthAspect根本不是一个Java类，所以aspect也不是Java支持的关键字，它只是AspectJ才能识别的关键字。

上面的粗体字也不是方法，它只是在执行某些类的某些方法之前，AspectJ将会自动先调用该代码块中的代码。因为aspect不是Java支持的关键字，所以Java无法识别AuthAspect.java文件的内容，所以使用ajc.bat来编译上面的Java程序

ajc –d . \*.java

可以把ajc.bat理解成增强版的javac.exe命令，都用于编译Java程序，区别是ajc命令可以识别AspectJ的语法。-d代表要放置的生成的class的类路径，后面的圆点表示以当前目录为标准，自动创建源文件的package的目录，并将生成的class文件放入到生成的目录中，如果不写“.”那么必须明确书写要放置的路径，然后根据书写的目录为标准，创建源文件的package目录，然后将生成的class文件，放入到生成的目录中。后面的\*.java表示编译当前目录下的所有Java源程序文件。

再运行AspectJTest类的main函数，程序运行效果如下：

**模拟进行权限检查…**

**执行Hello组件的foo()方法**

**模拟进行权限检查…**

**执行Hello组件的addUser()方法添加用户：孙凯**

**模拟进行权限检查…**

**执行World组件的bar()方法**

从上面的运行结果来看，完全不需要对Hello.java、World.java等业务组件进行任何修改，但同时又可以满足客户的需求。

如果客户再提出新的需求，比如需要在执行所有的业务方法之后，增加记录日志的功能，那也很简单，只需要再定义一个LogAspect就可以了，程序如下：

public aspect LogAspect{

// 定义一个Pointcut，其名称为logPointcut，该pointcut代表了后面给出的切入点表达式

// 这样就可以复用该切入点表达式

**pointcut logPointcut() :execution(\* com.langsin.test.\*.\*(..));**

**after():logPointcut()**{

System.out.println(“模拟记录日志…”);

}

}

//如果不定义切入点，则为 after():execution(\* com.langsin.test.\*.\*(..))

上面程序中的粗体字代码定义了一个pointcut:logPointuct()，这种用法就是为后面的切入点表达式起个名称，方便后面复用这个切入点表达式， 假如程序中有多个代码块需要使用切入点表达式，则可以写成如下格式：

after():logPointcut(){

//代码块。。。。。。。。。。

}

before():logPointcut(){

//代码块。。。。。。。。。。

}

这些代码块都可以直接复用此处定义的切入点logPointcut，而不是反复书写繁琐的切入点表达式：**execution(\* com.langsin.test.\*.\*(..));。**

再次使用如下命令来编译上面的Java程序：

**模拟进行权限检查…**

**执行Hello组件的foo()方法**

**模拟记录日志…**

**模拟进行权限检查…**

**执行Hello组件的addUser()方法添加用户：孙凯**

**模拟记录日志…**

**模拟进行权限检查…**

**执行World组件的bar()方法**

**模拟记录日志…**

加入现在需要在业务组件的所有业务方法之前启动事务，并在方法执行结束时关闭事务，同样只要定义如下TxAspect即可。

public aspect TxAspect{

//

Object around():call(\* com.langsin.test.\*.\*(..)){

System.out.println(“模拟开启事务….”);

// 回调原来的目标方法

**Object rvt = proceed();**

System.out.println(“模拟结束事务….”);

return rvt;

}

}

上面的粗体字代码指定proceed()代表回调原来的目标方法，这样位于proceed()代码之前的代码就会被添加在目标方法之前，位于proceed()代码之后的代码就会被添加在目标方法之后。

如果再次使用jac.bat命令来编译上面所有的Java类，并执行AspectJTest，此时将会发现系统中两个业务组件所包含的业务方法已经变得很强大，但并未修改过Hello.java、World.java的源代码——这就是AspectJ的作用：开发者无须修改源代码，但又可以为这些组件的方法添加新的功能。

通过Java的反编译工具，反编译前面程序生成的Hello.class、World.class文件，将会发现此时的Hello.class和World.class文件已经不是由Hello.java、World.java文件编译得到的，Hello.class、World.class里新增了很多内容——这表明AspectJ在编译时已增强了Hello.class、World.class类的功能，因此AspectJ通常被称为编译时增强的AOP框架。

AOP要达到的效果是，保证在程序员不修改源代码的前提下，为系统中业务组件的多个业务方法添加某种通用功能。但AOP的本质是，依然要去修改业务组件的多个业务方法的源代码——只是这个修改由AOP框架完成，程序员不需要修改。

AOP实现可分为两类：

* 静态AOP实现：AOP框架在编译阶段对程序进行修改，即：实现对目标类的增强，生成静态的AOP代理类，即生成的\*.class已经被改掉了，需要特定的编译器。以AspectJ为代表。
* 动态AOP实现：AOP框架在运行阶段生成AOP代理，即：在内存中以JDK动态代理或cglib动态地生成AOP代理类，以实现对目标对象的增强。以Spring AOP为代表。

一般来说，静态AOP实现具有较好的性能，但需要使用特殊的编译器。动态AOP实现是纯Java实现，因此无须特殊的编译器，但是通常性能较差。

### 2.4.3 AOP的基本概念

AOP从程序运行角度考虑程序的流程，提取业务处理过程的切面。AOP面向的是程序运行中各个步骤，希望以更好的方式来组合业务处理的各个步骤。

AOP框架并不与特定的代码耦合，AOP框架能处理程序执行中特定的切入点pointcut，而不与某个具体类耦合。AOP框架具有如下两个特征。

* 各步骤之间的良好隔离性。
* 源代码无关性。

下面是关于面向切面编程的一些术语。

* 切面（Aspect）：切面用于组织多个Advice（增强处理），Advice放在切面中定义。即：被加入执行的代码块。
* 连接点（Joinpoint）：程序执行过程中明确的点，比如方法的调用，或者异常的抛出。在Spring AOP中，连接点总是方法的调用。
* 增强处理（Advice）：AOP框架在特定的切入点上执行的增强处理。增强处理有“around”、“before”、“after”等类型。
* 切入点（Pointcut）：可以插入增强处理的连接点。简而言之，当某个连接点满足指定要求时，该连接点将被添加增强处理，该连接点也就变成了切入点。例如如下代码：

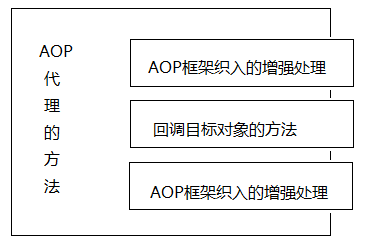
pointcut xxxPointcut():execution(void H\*.say\*)

每个方法被调用都只是连接点，但如果方法属于H开头的类，且方法以say开头，则该方法的执行将变成切入点。也可以认为符合条件的连接点即为切入点。如何用表达式来定义切入点是AOP的核心，Spring默认使用AspectJ切入点的语法。

* 引入：将方法或字段添加到被处理的类中。
* 目标对象：被AOP框架进行增强处理的对象，也被称为增强对象。如果AOP框架采用的是动态AOP实现，那么该对象就是一个被代理的对象。
* AOP代理：AOP框架创建的对象，简单地说，代理就是对目标对象的增强。Spring中的AOP代理可以是JDK动态代理，也可以是cglib(Code Generation Library：代码生成库)代理。前者为实现接口的目标对象的代理，后者为不实现接口的目标对象的代理。
* 织入（Weaving）：将增强处理添加到目标对象中，并创建一个被增强的对象的过程就是织入。织入有两种方式——编译时增强和运行时增强。

由前面的介绍知道，AOP代理就是由AOP框架动态生成一个对象，该对象可作为目标对象使用。AOP代理包含了目标对象的全部方法，但AOP代理中的方法与目标对象的方法存在差异——AOP方法在特定的切入点加入了增强处理，并回调了目标对象的方法。

AOP代理所包含的方法与目标对象的方法示意图如下所示：



### 2.4.4 Spring的AOP支持

Spring中的AOP代理由Spring的IoC容器负责生成、管理，其依赖关系也由IoC容器负责管理。因此，AOP代理可以直接使用容器中的其他Bean实例作为目标，这种关系可由IoC容器的依赖注入提供。Spring默认使用Java动态代理来创建AOP代理，这样就可以为任何接口实例创建代理了。

Sring也可以使用cglib代理，在需要代理类而不是代理接口的时候，Spring会自动切换为使用cglib代理。但Spring推荐使用面向接口编程，因此业务对象通常都会实现一个或多个接口，此时默认将使用JDK动态代理，但也可强制使用cglib代理。

Spring AOP使用纯Java实现。它不需要专门的编译过程。因此它可以在所有的Java Web容器或应用服务器中运行良好。

Spring目前仅支持将方法调用作为连接点(Joinpoint)，如果需要把对成员变量的访问和更新也作为增强处理的连接点，则可以考虑使用AspectJ。

Spring实现AOP方法跟其他框架不同。Spring并不是要提供最完整的AOP实现，Spring侧重于AOP实现与IoC容器之间的整合，用于帮助解决企业级开发中常见问题。

因此，Spring的AOP通常和Spring的IoC容器一起使用，Spring的AOP从来没有打算通过提供一种全面的AOP解决方案来与AspectJ竞争。Spring的AOP采用基于动态代理的AOP实现方案，而AspectJ则采用编译时增强的解决方案。

Spring2.0可以无缝地整合Spring AOP、IoC和AspectJ，使得所有的AOP应用完全融入基于Spring的框架中，这样的集成不会影响Spring AOP API或者AOP Alliance API，SpringAOP保持了向下兼容性，依然允许直接使用Spring AOP API来完成AOP编程。

如果掌握上面AOP的相关概念，那么进行 AOP编程将会变得非常简单。纵观所有的AOP编程，就会发现需要程序员参与的只有三个部分。

* 定义普通业务组件。
* 定义切入点，一个切入点可能横切多个业务组件。
* 定义增强处理，增强处理就是在AOP框架中为普通业务组件织入的处理动作。

定义业务组件，无须额外说明，就是普通的业务开发。那么进行AOP编程的关键就是定义切入点和定义增强处理。一旦定义了合适的切入点和增强处理，AOP框架将会自动生成AOP代理，而AOP代理的方法大致有如下公式：

**AOP代理的方法 = 增强处理 + 目标方法**

Spring1.X采用自身提供的AOP API来定义切入点和增强处理，但目前这种方式已经过时了。现在通常建议使用AspectJ方法来定义切入点和增强处理，在这种方式下，Spring依然有如下两种选择来定义切入点和增强处理。

* 基于注解的“零配置”方式：使用@Aspect、@Pointcut等注解来标注切入点的增强处理。
* 基于XML配置文件的管理方式：使用Spring配置文件来定义切入点和增强处理。

### 2.4.5 基于注解的“零配置”方式

AspectJ允许使用注解定义切面、切入点和增强处理，而Spring框架则可以识别这些注解并根据注解来生成AOP代理。Spring只是使用了和AspectJ 1.5一样的注解，但并没有使用AspectJ的编译器或者织入器，底层依然使用的是Spring AOP，依然是在运行时动态生成AOP代理，并不依赖AspectJ的编译器或者织入器。

为了启用Spring对@AspectJ切面配置的支持，并保证Spring容器中的目标Bean被一个或多个切面自动增强，必须在Spring配置文件中配置如下片段：

<?xml version="1.0" encoding="GBK"?>

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

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

**xmlns:aop="http://www.springframework.org/schema/aop"**

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

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

**http://www.springframework.org/schema/aop**

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

<!-- 启动@AspectJ支持 -->

**<aop:aspectj-autoproxy/>**

</beans>

如果希望完全启动Spring的“零配置”功能，则需要安装Spring的“零配置”方式进行配置。

如果不喜欢用Spring的XML Schema的配置方式，即<aop:aspectj-autoproxy/>这种方式，则应该在Spring配置文件中增加以下片段来启用@AspectJ支持。

**<bean class=”org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoPoxyCreater”/>**

AnnotationAwareAspectJAutoPoxyCreater类是一个Bean后处理器，该Bean后处理器将会为容器中所有的Bean生成AOP代理。

为了在Spring应用中启动@AspectJ支持，还需要在应用的类加载路径下增加两个AspectJ库：asectjweaver.jar和aspectjrt.jar，直接使用AspectJ安装路径下lib目录中增加的两个JAR文件即可。除此之外，Spring AOP还需要依赖一个aopalliance.jar。

#### 1、定义切面Bean

当启动了@AspectJ支持后，只要在Spring容器中配置一个带@AspectJ注解的Bean，Spring将会自动识别该Bean，并将该Bean作为切面处理。

在Spring容器中配置切面Bean，即：带@AspectJ注解的Bean，与配置普通Bean没有任何区别，一样使用bean元素进行配置，一样支持使用依赖注入来配置属性值；如果启动了Spring的“零配置”特性，一样可以让Spring自动搜索指定路径下的切面Bean。

使用@Aspect标注一个Java类，该Java类将会作为切面Bean，如下面的代码所示。

示例1：

@Aspect

public class LoginAction{

//定义该类的业务内容

}

切面类和其他类一样可以有方法、成员变量定义、区别是还可能包括切入点、增强处理定义。

当使用@Aspect来修饰一个Java类之后，Spring将不会把该Bean当成组件Bean处理，因此负责自动增强的后处理Bean将会略过该Bean，不会对该Bean进行任何增强处理。

所以开发时无须担心使用@Aspect定义的切面类被增强处理，当Spring容器检测到某个Bean类使用了@Aspect修饰之后，Spring容器不会对该Bean类进行增强。

#### 2、定义Before增强处理

在一个切面类里使用@Before来修饰一个方法时，该方法将作为一个Before增强处理。使用@Before修饰时，通常需要指定一个value属性值，该属性值指定一个切入点表达式（既可以是一个已有的切入点，也可以直接定义切入点表达式），用于指定该增强处理将被织入哪些切入点。

下面的Java类里使用了@Before定义了一个Before增强处理。

示例1：

package com.langsin.aop;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

@Aspect

public class AuthAspect {

// 匹配com.langsin.anno包下的所有类的所有方法，所有方法的执行作为切入点

@Before("execution(\* com.langsin.anno.\*.\*(..))")

public void authority(){

System.out.println("模拟进行权限检查操作！");

}

}

上面程序中使用@Aspect修饰了AuthAspect类，表示该类为一个切面类，在该类里定义了一个authority()方法，这个方法本来没有任何特殊之处，但因为使用了@Before来标注该方法后，此方法就变成了一个Before增强处理。

上面程序使用了@Before注解时，直接指定了切入点表达式，指定匹配com.langsin.anno包下所有类的所有方法的执行作为切入点。

下面在com.langsin.anno包下定义两个类：HelloImpl实现类，Hello接口类，就是单纯的业务组件类，并使用@Component注解进行了修饰。下面是HelloImpl代码

示例2：

public interface Hello{

public void foo();

public void addUser(String name,String pass);

}

package com.langsin.anno;

import org.springframework.stereotype.Component;

@Component("hello")

public class HelloImpl implements Hello {

@Override

public void foo() {

System.out.println("执行Hello组件的foo方法");

}

@Override

public void addUser(String name, String pass) {

System.out.println("执行Hello组件的addUser方法");

}

}

从上面的HelloImpl代码来看，只是一个纯净的Java类，它丝毫不知道将被谁来进行增强，也不知道将被进行怎样的增强——正是因为这种什么都不知道，才是AOP的最大魅力所在，即：目标类可以被无限增强。

在Spring配置文件中配置自动搜索Bean组件、自动搜索切面类，Spring AOP自动对Bean组件进行增强，下面是Spring配置文件代码。

<?xml version="1.0" encoding="GBK"?>

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

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

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

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

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

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

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

http://www.springframework.org/schema/beans/spring-beans-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/aop

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

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

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

<context:component-scan base-package="com.langsin.anno,com.langsin.aop">

**<context:include-filter type="annotation" expression="org.aspectj.lang.annotation.Aspect"/>**

</context:component-scan>

<!-- 启动@AspectJ支持 -->

<aop:aspectj-autoproxy />

</beans>

主程序示例3：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:test.xml");

Hello hel = ctx.getBean("hello", Hello.class);

hel.foo();

hel.addUser("张三", "123456");

}

执行效果如下：



注意：使用Before增强处理只能在目标方法执行之前织入增强，如果Before增强处没有任何特殊处理，目标方法总会自动执行，如果Before处理需要阻止目标方法的执行，可通过抛出异常来实现。Before增强处理执行时，目标方法还没有被执行，所以Before增强处无法访问目标方法的返回值。

#### 3、定义AfterReturning增强处理

类似于使用@Before注解可修饰Before增强处理，使用@AfterReturning可修饰AfterReturning增强处理，AfterReturning增强处理将在目标方法正常完成后被织入。

使用@AfterReturning注解可指定如下两个常用属性。

* pointcut/value：这个两个属性的作用是一样的，都用于指定该切入点对应的切入表达式。一样既可以是个已有的切入点，也可以直接定义切入点表达式。当指定了pointcut属性值后，value属性将会被覆盖。
* returning：该属性值指定一个形参名，用于表示Advice方法中可以定义与此同名的形参，该形参可以用于访问目标方法的返回值。除此之外，在Advice方法中定义该形参时指定的类型，会限制目标方法必须返回指定类型的值或没有返回值。

下面程序定义了一个AfterReturning增强处理。

package com.langsin.aop;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.Aspect;

@Aspect

public class LoginAspect {

@AfterReturning(returning="rvt",pointcut="execution(\* com.langsin.anno.\*.\*(..))")

public void login(Object rvt){

System.out.println("获取目标的返回值："+rvt);

System.out.println("模拟记录日志操作！");

}

}

正如上面的程序中看到的，程序中使用@AfterReturning注解时，指定了一个returning属性，该属性值为rvt，这表明允许在Advice方法中，即login方法中定义名为rvt的形参，程序可通过rvt形参来访问目标方法的返回值。

运行下面主程序：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:test.xml");

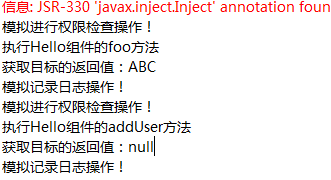
Hello hel = ctx.getBean("hello", Hello.class);

hel.foo();

hel.addUser("张三", "123456");

}

运行效果如下：



@AfterReturning注解的returning属性所指定的形参名必须对应增强处理中的一个形参名，当目标方法执行返回后，返回值作为相应的参数值传入增强处理的方法

使用returning属性还有一个额外的作用：它可以用于限定切入点只匹配具有对应返回值类型的方法，假如在上面的login方法中定义的rvt形参的类型是String，则该切入点只匹配com.langsin.anno包下返回值类型为String或没有返回值的方法。上面login方法的rvt形参类型是Object，则表明切入点匹配人任何返回值类型的方法。

#### 4、定义AfterThrowing增强处理

使用@AfterThrowing注解可修饰AfterThrowing增强处理，AfterThrowing增强处理主要用于处理程序中未处理的异常。

使用@AfterThrowing注解时可指定如下两个常用属性。

* pointcut/value：这个两个属性的作用是一样的，都用于指定该切入点对应的切入表达式。一样既可以是个已有的切入点，也可以直接定义切入点表达式。当指定了pointcut属性值后，value属相将会被覆盖。
* throwing：该属性值指定一个形参名，用于表示Advice方法中可以定义与此同名的形参，该形参可用于访问目标方法抛出的异常。除此之外，在Advice方法中定义该形参时指定的类型，会限制目标方法抛出指定类型的异常。

下面定义一个使用AfterThrowing增强处理的切面类

package com.langsin.aop;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Aspect;

@Aspect

public class RepairAspect {

@AfterThrowing(throwing="able",pointcut="execution(\* com.langsin.anno.HelloImpl.foo(..))")

public void dealAction(Throwable able){

System.out.println("只有执行anno包下的HelloImpl类的foo方法时才进行增强");

System.out.println("目标方法中抛出的异常为："+able.getMessage());

System.out.println("模拟Advice对异常进行修复....");

}

}

在上面的程序中看到，程序中使用@AfterThrowing注解时指定了一个throwing属性，该属性值为able，这表示在dealAction方法中需要定义一个参数名为able的形参，程序可通过该形参访问目标方法抛出的异常。

将前面的HelloImpl类做一些修改，用于模拟程序抛出异常，如下所示：

package com.langsin.anno;

import org.springframework.stereotype.Component;

@Component("hello")

public class HelloImpl implements Hello {

@Override

public String foo() {

System.out.println("执行Hello组件的foo方法");

int num = 12;

if(num%2==0){

throw new NullPointerException("抛出空指针异常！");

}

return "ABC";

}

@Override

public void addUser(String name, String pass) {

System.out.println("执行Hello组件的addUser方法");

}

}

运行主程序如下：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:test.xml");

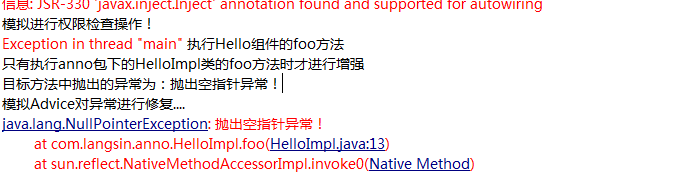
Hello hel = ctx.getBean("hello", Hello.class);

hel.foo();

hel.addUser("张三", "123456");

}

该示例的主程序，效果如下：



正如上图所示，AOP的AfterThrowing处理虽然可以对目标方法的异常进行处理，但这种处理与直接使用catch捕获不同，catch捕获意味着完全处理该异常，如果catch块中没有重新抛出新异常，则该方法可以正常结束；而AfterThrowing处理虽然也处理了该异常，但它不能完全处理该异常，该异常依然会传播到上一级调用者，在本例中直接传播到JVM，所以程序中止。

#### 5、After增强处理

Spring还提供了一个After增强处理，它与AfterReturning增强处理有点相似，但也有区别。

* AfterReturning增强处理只有在目标方法成功后才会被织入。
* After增强处理不管目标方法如何结束，包括成功完成和遇到异常中止两种情况，它都会被织入。

因为不管一个方法是如何结束的，After增强处理都会被织入，因此After增强处理必须准备处理正常返回和异常返回两种情况。这种增强处理通常用于释放资源操作。

使用@After注解修饰一个方法，即可将该方法转成After处理。使用@After注解时需要指定一个value属性，该属性值用于指定该增强处理被织入的切入点，既可以是一个已有的切入点，也可以是直接指定切入点表达式。

下面的程序定义一个After增强处理。

package com.langsin.aop;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Aspect;

@Aspect

public class RepairAspect {

@AfterThrowing(throwing="able",pointcut="execution(\* com.langsin.anno.HelloImpl.foo(..))")

public void dealAction(Throwable able){

System.out.println("只有执行anno包下的HelloImpl类的foo方法时才进行增强");

System.out.println("目标方法中抛出的异常为："+able.getMessage());

System.out.println("模拟Advice对异常进行修复....");

}

@After("execution(\* com.langsin.anno.HelloImpl.foo(..))")

public void release(){

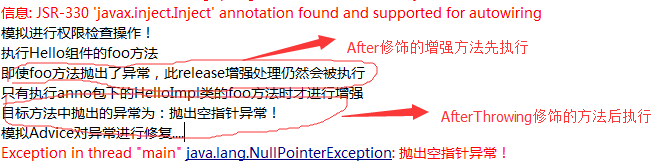
System.out.println("即使foo方法抛出了异常，此release增强处理仍然会被执行");

}

}

上面程序中的粗体字代码定义了一个After增强处理，不管切入点的目标方法如何结束，该增强处理都会被织入，而且此类中的两个增强处理都织入到目标方法中了，只不过一个使用AfterThrowing修饰，一个使用After修饰，但是After织入会被先执行，其他后执行。

指向效果如下：



#### 6、Around增强处理

@Around注解用于修饰Around增强处理，Around增强处理是功能比较强大的增强处理，它近似等于Before增强处理和AfterReturning增强处理的总和，Around增强处理既可在执行目标方法之前织入增强动作，也可在执行目标方法之后织入增强动作。使用@Around注解时需要指定一个value属性，该属性指定该增强处理被织入的切入点。

与Before增强处理、AfterReturning增强处理不同的是，Around增强处理可以决定目标方法在什么时候执行，如何执行，甚至可以完全阻止目标方法的执行。

Around增强处理可以改变执行目标方法的参数值，也可以改变执行目标方法之后的返回值。

Around增强处理的功能虽然强大，但通常需要在线程安全的环境下使用。因此，如果使用普通的Before增强处理、AfterReturning增强处理就能解决的问题，就没有必要使用Around增强处理了。如果需要目标方法执行之前和执行之后共享某种状态数据，则应该考虑使用Around增强处理；尤其是需要改变目标方法的返回值时，则只能使用Around增强处理了。

当定义一个Around增强处理方法时，该方法的第一个参数必须是ProceedingJoinPoint类型，在增强处理方法体内，调用ProceedingJoinPoint参数的proceed()方法才会执行目标方法，这就是Around增强处理可以完全控制目标方法的执行时机、如何执行的关键；如果程序没有调用ProceedingJoinPoint参数的proceed()方法，则目标方法不会被执行。

调用proceed()方法时，还可以传入一个Object[]对象作为参数，该数组中的值将被传入目标方法作为执行方法的实参。

下面程序定义了一个Around增强处理。

package com.langsin.aop;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

@Aspect

public class RepairAspect {

@Around(value="execution(\* com.langsin.anno.HelloImpl.sum(..))")

public int aroundAspect(ProceedingJoinPoint point) throws Throwable{

System.out.println("执行目标之前开始模拟事务......");

// 获取目标方法原始的调用参数

Object[] args = point.getArgs();

if(args!=null){

for(int i=0;i<args.length;i++){

args[i] = (Integer)args[i]+2;

}

}

// 改变参数值后调用目标方法

System.out.println("开始执行目标方法......");

**Object rvt = point.proceed(args);**

System.out.println("执行目标之后模拟事务结束.......");

// 更改执行目标方法之后的返回值信息

**return (Integer)rvt\*2;**

}

}

在上面的切面类中包含了一个Around增强处理：aroundAspect()方法，方法中的粗体代码用于回调目标方法，回调目标方法时传入了一个args数组，但这个数组是执行目标方法的原始参数被修改后的结果，这样就实现了对调用参数的修改；第二行粗体代码用于改变目标方法的返回值。

执行主程序代码：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("test.xml");

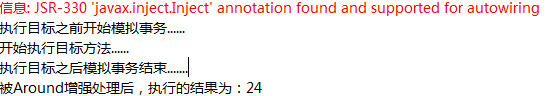
Hello hello = ctx.getBean("hello", Hello.class);

int result = hello.sum(3, 5);

System.out.println("被Around增强处理后，执行的结果为："+result);

}

执行效果如下：



从上图可以看出，使用Around增强处理可以取得对目标方法最大的控制权，既可以完全控制目标方法的执行，也可以改变执行目标方法的参数，还可以改变目标方法的返回的值。

当调用ProceedingJoinPoint的proceed()方法时，传入的Object[]参数值将作为目标方法的参数，如果传入的数组长度与目标方法所需要参数的个数不相等，或者数组元素与目标方法所需要参数的类型不匹配，程序就会抛出异常。

#### 7、访问目标方法的参数

访问目标方法最简单的做法是定义增强处理方法时将第一个参数定义为JoinPoint类型，当该增强处理方法被调用时，该JoinPoint参数就代表了织入增强处理的连接点。JoinPoint里包含了如下几个常用的方法。

* Object[] getArgs()：返回执行目标方法时的参数。
* Signature getSignature()：返回被增强的方法的相关信息。
* Object getTarget()：返回被织入增强处理的目标对象。
* Object getThis()：返回AOP框架为目标对象生成的代理对象。

通过使用这些方法就可以访问目标方法的相关信息。

示例1：

package com.langsin.aop;

import java.util.Arrays;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

@Aspect

public class RepairAspect {

@Before("execution(\* com.langsin.anno.Hello.addUser(..))")

public void authority(JoinPoint jp){

System.out.println("Before：被增强的方法名为："+jp.getSignature().getName());

System.out.println("Before：目标方法的参数："+Arrays.toString(jp.getArgs()));

System.out.println("Before：被织入增强的目标对象为："+jp.getTarget());

}

@AfterReturning(returning="rvt",pointcut="execution(\* com.langsin.anno.HelloImpl.foo(..))")

public void login(JoinPoint jp,Object rvt){

System.out.println("AfterReturning：获取目标方法的返回值："+rvt);

System.out.println("AfterReturning：被增强的方法名为："+jp.getSignature().getName());

System.out.println("AfterReturning：目标方法的参数："+Arrays.toString(jp.getArgs()));

System.out.println("AfterReturning：被织入增强的目标对象为："+jp.getTarget());

}

@After("execution(\* com.langsin.anno.HelloImpl.foo(..))")

public void release(JoinPoint jp){

System.out.println("After：目标方法结束后释放资源！");

System.out.println("After：被增强的方法名为："+jp.getSignature().getName());

System.out.println("After：目标方法的参数："+Arrays.toString(jp.getArgs()));

System.out.println("After：被织入增强的目标对象为："+jp.getTarget());

}

@Around(value="execution(\* com.langsin.anno.HelloImpl.sum(..))")

public int aroundAspect(ProceedingJoinPoint point) throws Throwable{

System.out.println("Around：执行目标之前开始模拟事务......");

// 获取目标方法原始的调用参数

Object[] args = point.getArgs();

if(args!=null){

for(int i=0;i<args.length;i++){

args[i] = (Integer)args[i]+2;

}

}

// 改变参数值后调用目标方法

System.out.println("Around：开始执行目标方法......");

Object rvt = point.proceed(args);

System.out.println("Around：执行目标之后模拟事务结束.......");

// 更改执行目标方法之后的返回值信息

return (Integer)rvt\*2;

}

}

运行下面主程序：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("test.xml");

Hello hello = ctx.getBean("hello", Hello.class);

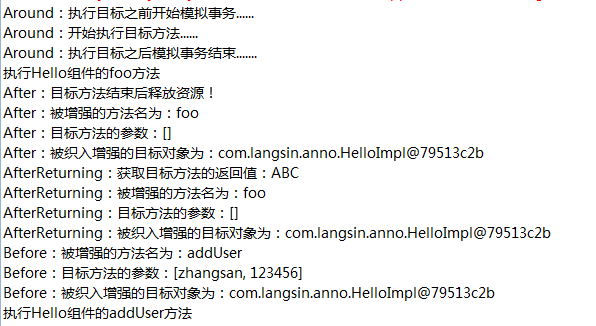
hello.sum(3, 5);

hello.foo();

hello.addUser("zhangsan", "123456");

}

执行效果如下：



Spring AOP采用和AspectJ一样的优先顺序来织入增强处理：在“进入”连接点时，具有最高优先级的增强处理将先被织入，在“退出”连接点时，具有最高优先级的增强处理会最后被织入。

当不同切面里的两个增强处理需要在同一个接入点被织入时，Spring AOP将以随机的顺序来织入这两个增强处理。如果应用需要指定不同切面类里增强处理的优先级，Spring提供了如下两个解决方案。

* 让切面类实现org.springframework.core.Ordered接口，该接口声明了一个int getOrder()方法，该方法的返回值越小，则优先级越高。
* 直接使用@Order注解来修饰一个切面类，该注解中含有一个int类型的value属性，该属性值越大，优先级越高。

同一个切面类里的两个相同类型的增强处理在同一个连接点被织入时，Spring AOP将以随机的顺序来织入这两个增强处理，程序没有办法控制它们的织入顺序。如果确实需要它们以固有的顺序被织入，只能有两种处理方式：1、将两个增强处理合并成一个增强处理。2、将两个增强处理分别定义在两个切面类中，然后通过指定切面类的级别来指定顺序。

如果只需要访问目标方法的参数，Spring还提供了一种更简单的方法：可以在指定切入点的表达式中使用args表达式来绑定目标方法的参数。如果在一个args表达式中指定了一个或多个参数，则该切入点将只匹配具有对应形参的方法，并且目标方法的参数值将被传入到增强处理方法中。

示例1：

@Aspect

@Order(value=1)

public class AcessAspect{

@AfterReturning(returning=”rvt”,pointcut=”execution(\* com.langsin.impl.HelloImpl.addUser(..)) && **args(arg0)**”)

public void access(Object rvt,**String arg0**){

System.out.println(“调用目标方法的第一个参数：”+arg0);

System.out.println(“获取目标方法的返回值：”+rvt);

}

}

使用args表达式有如下两个作用。

* 提供了一种简单的方法来访问目标方法的参数。
* 对切入点表达式增加额外的限制。

除此之外，使用args表达式时还可以使用如下形式：args(name,age,..)，这表明在增强处理方法中可以通过name，age来访问目标方法的参数。注意args表达式括号中的两个点“..”，它表示可匹配更多参数，如果该args表达式对应的增强处理方法名为：

@AfterReturning(returning=”rvt”,pointcut=”execution(\* com.langsin.impl.HelloImpl.addUser(..)) && args(name,age,..)”)

public void access(Object rvt,String name,int age,Date birth)

这就意味着只要目标方法的第一个参数时String类型，第二个参数时int类型，则该方法就可以匹配该切入点。

#### 8、定义切入点

在切面类中定义增强处理方法时，很少通过直接使用定义切入点表达式的方式来指定切入点，AspectJ和Spring都允许先定义一个切入点，所谓的定义一个切入点，其实质就是为一个切入点表达式起一个名称，从而允许在多个增强处理中重用该名称。

切入点定义包含两个部分：

* 一个切入点表达式
* 一个包含名字和任意参数的方法签名

其中切入点表达式用于指定该切入点和哪些方法进行匹配，包含名字和任意参数的方法签名将作为该切入点的名称。

在@AspectJ风格的AOP中，切入点签名采用一个普通的方法定义来提供，此方法体通常为空，并且该方法的返回值必须为void，因为这只是定义一个切入点名称，不需要有返回值，切点表达式需要使用@Pointcut注解来标注，如下所示：

// 使用@Pointcut注解定义一个切入点名称

@Pointcut(“execution(\* transfer(..))”)

private void myPointcut(){}

切入点表达式，实际为@Pointcut注解的的值，是正规的AspectJ切入点表达式。一旦采用上面代码片段定义了名称为anyTransferName的切入点之后，程序就可以多次重复使用该切入点。甚至可以在其他切面类、其他包的切面类使用该切入点。

如果需要使用本切面类中的切入点，则可以在使用@Before、@After、@Around等注解定义增强处理时，使用pointcut或value属性值引用已有的切入点。如下代码所示：

示例1：不带参数的切入点

@AfterReturning(**pointcut=”myPointcut()”**,returning=”retval”)

public void writeLog(Object retval,String mesg){

// 省略的代码

}

示例2：带参数的切入点的定义及引用（注意：在增强处理的方法上，定义的形参列表要保持一致）

@Pointcut("execution(\* foo(..)) && bean(hello) && args(num,age)") // 定义切入点

public void fooPointcut(String num,String age){

}

@After(value="fooPointcut(num,age)") // 引用切入点

public void fooAfter(String num,String age){

System.out.println("执行After增强处理。。。。");

}

如果需要使用其他切面类中的切入点时，则其他切面类中的切入点不能使用private修饰，要严格遵循Java的访问权限修饰符的访问规则。而且在使用@Before、@After、@Around等注解中pointcut或value引用已有的切入点是，必须添加类名前缀。如下所示：

示例1：在同一个切面类中

@Pointcut("execution(\* com.langsin.anno.HelloImpl.foo(..))")

public void loginPointCut(){}

@AfterReturning(pointcut="loginPointCut()",returning="rvt")

public void login(JoinPoint jp,Object rvt){

System.out.println("AfterReturning：获取目标方法的返回值："+rvt);

System.out.println("AfterReturning：被增强的方法名为："+jp.getSignature().getName());

System.out.println("AfterReturning：目标方法的参数："+Arrays.toString(jp.getArgs()));

System.out.println("AfterReturning：被织入增强的目标对象为："+jp.getTarget());

}

示例2：在不同切面类中

package com.langsin.aop;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Pointcut;

@Aspect

public class PointCutDefine {

@Pointcut("execution(\* com.langsin.anno.HelloImpl.foo(..))")

public void loginPointCut(){}

}

@Aspect

public class RepairAspect {

@AfterReturning(pointcut="PointCutDefine.loginPointCut()",returning="rvt")

public void login(JoinPoint jp,Object rvt){

System.out.println("AfterReturning：获取目标方法的返回值："+rvt);

System.out.println("AfterReturning：被增强的方法名为："+jp.getSignature().getName());

System.out.println("AfterReturning：目标方法的参数："+Arrays.toString(jp.getArgs()));

System.out.println("AfterReturning：被织入增强的目标对象为："+jp.getTarget());

}

}

#### 9、切入点指示符

前面定义切入点表达式时大量使用了execution表达式，其中execution就是一个切入点指示符。Spring AOP仅支持部分AspectJ的切入点指示符，但Spring AOP还额外支持一个bean切入点指示符。

不仅如此，因为Spring AOP只支持使用方法调用作为连接点，所以Spring AOP的切入点指示符仅匹配方法执行的连接点。

完整的AspectJ切入点语言支持大量的切入点指示符，但是Spring AOP并不支持它们，Spring AOP不支持的切入点指示符有call、get、set、preinitialization、staticinitialization、initalization、handler、adviceexecution、cflow、withincode、cflowbelow、if、@this和@withincode。一旦在Spring AOP中使用这些指示符，将会导致抛出Illegal

ArgumentException异常。

Spring AOP一共支持如下几种切入点指示符。

* execution：用于匹配执行方法的连接点，这是Spring AOP中主要的切入点指示符。该切入点的用法也相对复杂，execution的格式如下：

execution(modifier-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)

上面格式中的execution是不变的，用于作为execution表达式的开头，整个表达式中各部分的解释如下：

* modifier-pattern：指定方法的修饰符，支持通配符，该部分可省略
* ret-type-pattern：指定方法的返回值类型，支持通配符，可以使用“\*”通配符来匹配所有的返回值类型。
* declaring-type-pattern：指定方法所属的类，支持通配符，该部分可省略，但一般不省略。
* name-pattern：用于匹配指定的方法名，支持统配符，可以使用“\*”通配符来匹配所有方法。
* param-pattern：指定方法声明中的形参列表，支持两个通配符，即“\*”和“..”，其中“\*”表示一个任意类型的参数，而“..”表示零个或多个任意类型的参数。
* throws-pattern：指定方法声明抛出的异常，支持通配符，该部分可省略。
* within：用于限定匹配特定类型的连接点，当使用Spring AOP的时候，只能匹配方法执行的连接点。

例如如下几个within表达式：

// 指定com.langsin.impl包中任意的连接点

within(com.langsin.impl.\*)

// 指定com.langsin.impl包及其子包下的任意的连接点

within(com.langsin.impl..\*)

* this：用于限定AOP代理必须是指定类型的实例，匹配该对象的所有连接点。当使用Spring AOP的时候，只能匹配方法执行的连接点。

// 匹配实现了com.langsin.impl.Hello接口的AOP代理的所有连接点

// Spring AOP中只是方法执行的连接点

this(com.langsin.impl.Hello)

* target：用于限定目标对象必须是指定类型的实例，匹配该对象的所有连接点。当使用Spring AOP的时候，只能匹配方法执行的连接点。

// 匹配实现了com.langsin.impl.Hello接口的目标对象的所有连接点

// 在Spring AOP中只是方法执行的连接点

target(com.langsin.impl.Hello)

* args：用于对连接点的参数类型进行限制，要求参数类型是指定类型的实例。当使用Spring AOP的时候，只能匹配方法执行的连接点。

@After(value="execution(\* com.langsin.anno.HelloImpl.foo(..)) && args(num,age)")

public void fooAfter(int num,String age){

System.out.println("执行After增强处理。。。。");

}

如上例所示：args匹配方法形参列表为num和age的连接点，其中num和age的形参数据类型，由增强处理方法的形参类型来决定。

另外String AOP还提供了一个名为bean的切入点指示符，它用于限定只匹配指定Bean实例内的连接点。当然Spring AOP中只能使用方法执行作为连接点。

* bean：用于限定只匹配指定Bean实例内的连接点，实际上只能使用方法执行作为连接点。定义bean表达式时需要传入Bean的id或name，表示只匹配该Bean实例内的连接点。支持使用“\*”通配符。

// 匹配tradeService Bean实例内的连接点

bean(tradeService)

// 匹配名以service结尾的Bean实例内的连接点

bean(\*Service)

bean切入点表达式时Spring AOP额外支持的，并不是AspectJ所支持的切入点指示符。这个指示符对Spring框架非常实用：它可以明确指定为Spring的哪个Bean织入增强。

示例1：

@Before("execution(\* addUser(..)) && bean(hello)")

public void authority(JoinPoint jp){

System.out.println("Before：被增强的方法名为："+jp.getSignature().getName());

System.out.println("Before：目标方法的参数："+Arrays.toString(jp.getArgs()));

System.out.println("Before：被织入增强的目标对象为："+jp.getTarget());

}

#### 10、组合切入点表达式

Spring支持使用如下三个逻辑运算符来组合切入点表达式。

* &&：要求连接点同时匹配两个切入点表达式。
* ||：只要连接点匹配任意一个切入点表达式。
* ！：要求连接点不匹配指定的切入点表达式。

execution(\* com.langsin.anno.HelloImpl.foo(..)) && args(name,age,..)

上面的pointcut属性指定的切入点表达式需要满足两个条件。

* 匹配com.langsin.anno.HelloImpl.foo方法
* 被匹配的方法的第一个参数为name的类型，第二参数为age的类型，数据类型有增强处理方法自己决定。

### 2.4.6 基于XML配置文件的管理方式

除了使用Annotation注解方式来定义切面、切入点和增强处理外，Spring AOP也允许直接使用XML配置文件来定义管理它们。

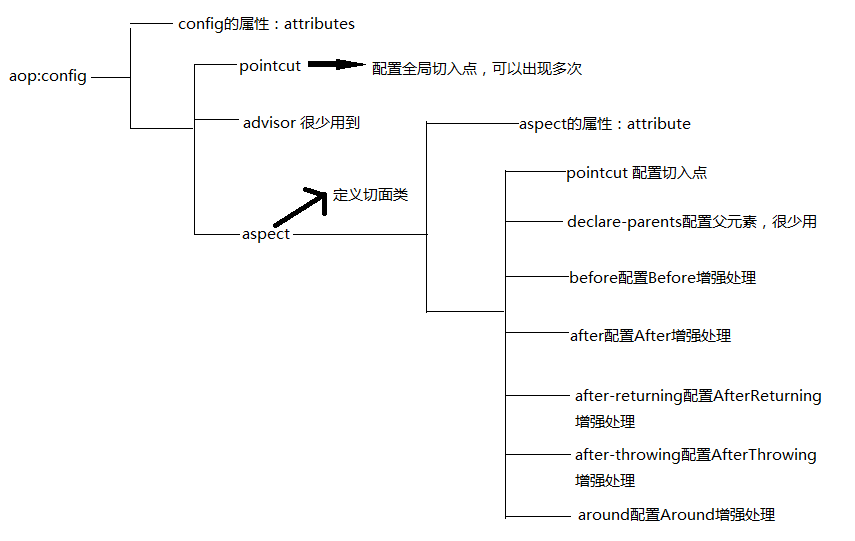
使用XML配置方式与前面介绍的@AspectJ方式的实质是一样的，同样需要指定相关信息：配置切面、切入点、增强处理所需要的信息完全一样，只是提供这些信息的位置不同而已。使用XML配置方式时是通过XML文件来提供这些信息的；而使用@AspectJ方式时则通过Annotation来提供这些信息。

使用XML配置，可以在配置文件中清晰得看出系统中存在哪些切面。但使用XML配置方式，也存在如下几个缺点。

* 使用XML配置方式不能将切面、切入点、增强处理等封装到一个地方。如果需要查看切面、切入点和增强处理，必须同时结合Java文件和XML配置文件来查看；但使用@AspectJ的注解方式时，只需要一个单独的切面类文件即可看到切面、切入点和增强处理的全部信息。
* XML配置方式比@AspectJ方式有更多的限制：仅支持“singleton”切面Bean，不能在XML中组合多个命名连接点的声明。

除此之外，@AspectJ切面还有一个优点，就是能被Spring AOP和@AspectJ同时支持，如果某天需要将应用改为使用AspectJ来实现AOP，使用@AspectJ将非常容易迁移到基于AspectJ的AOP实现中。相比之下，选择使用@AspectJ注解方式优势更大。

在Spring配置文件中，所有的切面，切入点和增强处理都必须定义在aop:config元素内部。beans元素下可以包含多个aop:config元素，一个aop:config元素可以包含pointcut:advisor和aspect元素，并且这三个元素必须按照彼此顺序来定义。关于aop:config元素所包含的子元素如下图所示：



在上图中已经非常清楚的绘制出aop:config元素下能包含三个有序的子元素：pointcut、advisor和aspect，其中aspect下可以包含多个子元素，通过使用这些子元素就可以在XML文件中配置切面、切入点和增强处理了。

#### 1、配置切面

使用aop:aspect元素定义切面时，其实质就是将一个已有的Spring Bean转换成切面Bean，所以需要先定义一个普通的Spring Bean。因为切面Bean可以当成一个普通的Spring Bean来配置，所以完全可以为该切面Bean配置依赖注入。当切面Bean定义完成后，通过aop:aspect元素中使用ref属性引用该Bean，就可以将该Bean转换成一个切面Bean了。

配置aop:aspect元素时可以指定如下三个属性。

* id：定义该切面的标识名。
* ref：用于将ref属性所引用的普通Bean转换成切面Bean。
* order：指定该切面Bean的优先级，该属性的作用与前面@AspectJ中的@Order注解、Ordered接口的作用完全一样，order属性值越大，该切面的优先级越高。

如下配置片段定义了一个切面：

<aop:config>

**<aop:aspect id=”afterAdvice” ref=”afterAdviceBean” >**

**……**

**</aop:aspect>**

</aop:config>

<bean id=”**afterAdviceBean**” class=”com.langsin.AfterAdvice”/>

上面配置文件中的粗体字代码将Spring容器中的afterAdviceBean转换成一个切面Bean，该切面Bean的id为afterAdvice。

由于Spring支持将切面Bean当成普通Bean来管理，所以完全可以利用依赖注入来管理切面Bean，管理切面Bean的属性值、依赖关系等。

#### 2、配置增强处理

与使用@AspectJ一样，使用XML一样可以配置Before、After、AfterReturning、AfterThrowing和Around五中增强处理，而且完全支持和@AspectJ完全一样的语义。

使用XML配置增强处理分别依赖如下几个元素。

* aop:before：配置Before增强处理。
* aop:after：配置After增强处理。
* aop:after-returning：配置AfterReturning增强处理。
* aop:after-throwing：配置AfterThrowing增强处理。
* aop:around：配置Around增强处理。

上面这些元素都不支持使用子元素，但通常可指定如下属性。

* pointcut：指定一个切入点表达式，Spring将在匹配该表达式的连接点时织入增强处理。
* pointcut-ref：指定一个已经存在的切入点名称，通常pointcut和pointcut-ref两个属性只需使用一个。
* method：指定一个方法名，指定将切面Bean的该方法转换成增强处理。
* throwing：该属性只对after-throwing元素有效，用于指定一个形参名，AfterThrowing增强处理方法可通过该形参访问目标方法所抛出的异常。
* returning：该属性只对after-returning元素有效，用于指定一个形参名，AfterReturning增强处理方法可以通过该形参访问目标方法的返回值。

如果选择使用XML配置方式来配置增强处理，所以切面类里定义切面、切入点和增强处理的Annotation就可以全部删除了。

当定义切入点表达式时，XML配置方式和@AspectJ注解方式支持完全相同的切入点指示符，即切入点表达式，一样可以支持execution、within、args、this、target、bean等切入点指示符。

XML配置方式和@AspectJ注解方式一样支持组合切入点表达式，但XML配置方式不再使用简单的&&、||和！作为组合运算符，而是使用如下三个组合运算符：and、or、not。

下面程序定义一个简单的切面类，该切面类只是将前面的@AspectJ示例中切面类中的Anntation删除。

package com.langsin.aop;

import java.util.Arrays;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.Pointcut;

public class RepairAspect {

public void authority(JoinPoint jp){

System.out.println("Before：被增强的方法名为："+jp.getSignature().getName());

System.out.println("Before：目标方法的参数："+Arrays.toString(jp.getArgs()));

System.out.println("Before：被织入增强的目标对象为："+jp.getTarget());

}

public void login(JoinPoint jp,Object rvt){

System.out.println("AfterReturning：获取目标方法的返回值："+rvt);

System.out.println("AfterReturning：被增强的方法名为："+jp.getSignature().getName());

System.out.println("AfterReturning：目标方法的参数："+Arrays.toString(jp.getArgs()));

System.out.println("AfterReturning：被织入增强的目标对象为："+jp.getTarget());

}

public void release(JoinPoint jp){

System.out.println("After：目标方法结束后释放资源！");

System.out.println("After：被增强的方法名为："+jp.getSignature().getName());

System.out.println("After：目标方法的参数："+Arrays.toString(jp.getArgs()));

System.out.println("After：被织入增强的目标对象为："+jp.getTarget());

}

public void fooAfterThrowing(Throwable able,String num,String age){

}

public int aroundAspect(ProceedingJoinPoint point) throws Throwable{

System.out.println("Around：执行目标之前开始模拟事务......");

// 获取目标方法原始的调用参数

Object[] args = point.getArgs();

if(args!=null){

for(int i=0;i<args.length;i++){

args[i] = (Integer)args[i]+2;

}

}

// 改变参数值后调用目标方法

System.out.println("Around：开始执行目标方法......");

Object rvt = point.proceed(args);

System.out.println("Around：执行目标之后模拟事务结束.......");

// 更改执行目标方法之后的返回值信息

return (Integer)rvt\*2;

}

}

上面的RepairAspect几乎是一个普通的Java类，类中大部分方法的第一个参数都是JoinPoint类型，主要是用于访问连接点的相关信息，使用这个参数主要是为了获取目标方法名、参数值等信息。

将上面的Java Bean配置成一个切面Bean，XML配置信息如下：

<?xml version="1.0" encoding="GBK"?>

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

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

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

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

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

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

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

http://www.springframework.org/schema/beans/spring-beans-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/aop

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

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

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

<aop:config>

<aop:aspect id="aspect" ref="**repairAspect**" order="1">

<aop:after-throwing method="fooAfterThrowing" throwing="able" pointcut="execution(\* com.langsin.anno.HelloImpl.foo(..)) and args(num,age)"/>

</aop:aspect>

</aop:config>

<bean id="**repairAspect**" class="com.langsin.aop.RepairAspect" />

<bean id="hello" class="com.langsin.anno.HelloImpl" />

<bean id="worker" class="com.langsin.impl.Worker" p:resource="classpath:conn.properties"/>

</beans>

在上面的配置文件中，已经没有了<aop:aspectj-autoproxy />元素的配置，因为使用纯Spring的AOP来支持，所以就不需要启动@AspectJ的支持了。因此启动@AspectJ的支持是为了能够让Spring识别AspectJ中的增强注解而已，所以需要启动@AspectJ的支持，现在不需要用注解了，则可以在XML中去掉此配置信息。

#### 3、配置切入点

类似于@AspectJ方式，允许定义切入点来重用切入点表达式，XML配置方式也可通过定义切入点来重用切入点表达式，Spring提供了aop:pointcut元素来定义切入点。当把aop:pointcut元素作为aop:config元素的子元素定义时，表明该切入点可被多个切面共享；当把aop:pointcut元素作为aop:aspect的子元素定义时，表明该切点只能在该切面中有效。

配置aop:pointcut元素时通常需要指定如下两个属性：

* id：指定该切入点的标识名。
* expression：指定该切入点关联的切入点表达式。

如下配置片段定义了一个简单的切入点：

<aop:pointcut id=”mypointcut” expression=”execution(\* com.langsin.impl.HelloImpl.foo(..))”/>

上面的配置片段即可作为aop:config的子元素，用于配置全局切入点；也可作为aop:aspect的子元素，用于配置仅对该切面有效的切入点。

除此之外，如果程序已经使用注解定义了切入点，在aop:pointcut元素中指定切入点表达式时还有另外一种用法，看如下配置片段：

<aop:config>

<aop:pointcut id=”mypointcut” expression=”com.langsin.impl.HelloImpl.foo()”/>

</aop:config>

下面的程序定义了一个AfterThrowing增强处理，包含该增强处理的切面类如下。

示例1：

package com.langsin.aop;

import org.aspectj.lang.annotation.AfterThrowing;

public class RepairAspect {

@AfterThrowing(throwing="able",pointcut="execution(\*com.langsin.anno.HelloImpl.foo(..)) && args(num,age,..)")

public void fooAfterThrowing(Throwable able,String num,String age){

}

}

下面的配置文件将负责配置该Bean，并将该Bean实例转换成切面Bean，并使用配置共享切入点的方式配置切入点。

示例2：

<?xml version="1.0" encoding="GBK"?>

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

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

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

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

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

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

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

<aop:config>

**<aop:pointcut expression="execution(\* com.langsin.anno.HelloImpl.foo(..)) and args(num,age)" id="mypoint"/>**

<aop:aspect id="aspect" ref="repairAspect" order="1">

<aop:after-throwing method="fooAfterThrowing" throwing="able" **pointcut-ref="mypoint"**/>

</aop:aspect>

</aop:config>

<bean id="repairAspect" class="com.langsin.aop.RepairAspect" />

<bean id="hello" class="com.langsin.anno.HelloImpl" />

<bean id="worker" class="com.langsin.impl.Worker" p:resource="classpath:conn.properties"/>

</beans>

## 2.5 Spring3.1新增的缓存机制

Spring3.1新增了一种全新的缓存机制，这种缓存机制与Spring容器无缝地整合在一起，可以对容器中的任意Bean或者Bean的方法增加缓存。Spring的缓存机制非常灵活，它可以对容器中的任意Bean或Bean的任意方法进行缓存，因此这种缓存机制可以在Java EE应用的任何层次上进行缓存。

Spring缓存同样不是一种具体的缓存实现方案，它底层同样需要依赖EhCache、Guava等具体的缓存工具。但这也正是Spring缓冲机制的优势，应用程序只要面向Spring缓冲API编程，应用底层的缓存实现可以在不同的缓存实现之间自由切换，应用程序无须任何改变，只要对配置文件略作修改即可。

### 2.5.1 启用Spring缓存

Spring配置文件专门为缓存提供了一个cache:命名空间，为了启用Spring缓存，需要在配置文件中导入cache:命名空间。导入cache:命名空间与前面介绍的导入util:、context:命名空间的方式完全一样。如下所示：

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

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

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

导入cache:命名空间后，启用Spring缓存还需要两步：

1、在Spring配置文件中添加<cache:annotation-driven cache-manager=”缓存管理器ID”/>，该元素指定Spring根据注解来启用Bean级别或方法级别的缓存。

2、针对不同的缓存实现配置对应的缓存管理器。

对于上面两步，其中第1步非常简单，使用cache:annotation-driven元素时可通过cache-manager属性显示指定容器中缓存管理器的ID；该属性的默认值为cacheManager，如果容器中缓存管理器的ID设置为cacheManager，则可以省略cache:annotation-driven的cache-manager属性。

第2步稍微有些复杂，由于Spring底层可使用大部分主流的Java缓存工具，而不同的缓存工具所需的配置也不同，因此略微有点繁琐。下面以EhCache为例来介绍Spring缓存的配置。

1. **EhCache缓存实现的配置**

Ehcache是一种广泛使用的开源的Java分布式缓存。主要面向通用缓存、Java EE和轻量级容器三者提供缓存技术支持。它具有内存和磁盘存储、缓存加载器、缓存扩展、缓存异常处理程序；一个gzip缓存servlet过滤器；支持REST和SOAP api等特点。

在配置EhCache缓存实现之前，首先需要将EhCache缓存的JAR包添加到项目的类加载路径中。只要将ehcache-core-2.4.3.jar和s1f4j-api-1.6.1.jar复制到项目类加载路径下即可。其中ehcache-core-2.4.3.jar是EhCache的核心JAR包，而slf4j-api-1.6.1.jar则是缓存工具所使用的日志工具。

为了使用EhCache，同样需要在应用的类加载路径下添加一个ehcache.xml配置文件。例如，使用如下ehcache.xml配置文件。

<?xml version=”1.0” encoding=”gbk”?>

<ehcache>

<diskStore path=”java.io.tmpdir”>

<defaultCache maxElementsInMemory=”10000”

eternal=”false”

timeToIdleSeconds=”120”

timeToLiveSeconds=”120”

maxElementsOnDisk=”10000000”

diskExpriyThreadIntervalSeconds=”120”

memoryStoreEvictionPolicy=”LRU”/>

<!-- 配置名为users的缓冲区 -->

<cache name=”users”

maxElementsInMemory=”10000”

eternal=”false”

overflowToDisk=”true”

timeToIdleSeconds=”300”

timeToLiveSeconds=”600” />

</ehcache>

对于上面的配置文件中属性的说明如下：

* maxElementsInMemory：设置缓存中最多可以放多少个对象。
* maxElementsOnDisk：设置硬盘中最多可以放多少个对象。
* eternal：设置缓存是否永久有效。
* timeToIdleSeconds：设置缓存的对象多少秒没有被使用就会被清理掉。
* timeToLiveSeconds：设置缓存的对象在过期之前可以缓存多少秒。
* diskPersistent：设置缓存是否被持久化到硬盘中，保存路径由diskStore元素指定。
* overflowToDisk：设置缓存异常后是否存入硬盘中。
* diskExpriyThreadIntervalSeconds：指定专门用于清除过期对象的监听线程的轮询时间。
* memoryStoreEvictionPolicy：当内存缓存达到最大，有新的element加入的时候，移除缓存中element的策略。默认是LRU（Least Recently Used）近期最少使用，可选的有LFU（Least Frequently Used）最不常使用的，FIFO（First Input First Output）先进先出。

Spring使用EhCacheCacheManager作为EhCache缓存实现的缓存管理器，因此只要该对象配置在Spring容器中，它就可以作为缓存管理器使用，但EhCacheCacheManager底层需要依赖一个net.sf.ehcache.CacheManager作为实际的缓存管理器。

为了将net.sf.ehcache.CacheManager加入到Spring容器的管理之下，Spring提供了EhCacheManagerFactory

Bean工厂Bean，，该工厂Bean实现了FactoryBean<CacheManager>接口，当程序把EhCacheManagerFactoryBean部署在Spring容器中，并通过Spring容器请求该工厂Bean时，实际返回的是它的产品，CacheManager对象。

因此，为了在Spring配置文件中配置基于EhCache的缓存管理器，只要增加如下两段配置即可。

<!-- 启动Spring的缓存管理 -- >

**<cache:annotation-driven cache-manager="ehCacheManager"/>**

<bean id=”**cacheManager**” class=”org.springframework.cache.ehcache.EhCacheManagerFactoryBean”

p:configLocation=”classpath:ehcache.xml” p:shared=”false”/>

<bean id=”ehCacheManager” class=”org.springframework.cache.ehcache.EhCacheManager”

**p:cacheManager-ref=”cacheManager”**/>

上面配合文件中配置的第一个Bean是工厂Bean，它用于配置EhCache的CacheManager实例的生成，第二个Bean才是为Spring的缓存，配置的基于EhCache的缓存管理器，该缓存管理器需要依赖于CacheManager，因此需要将第一个Bean注入到第二Bean中，如上述粗体代码。

配置好上面缓存管理器之后，接下来就可以使用注解来驱动Spring将缓存数据放入指定缓存区了。

### 2.5.2 使用@Cacheable执行缓存

@Cacheable可用于修饰类或修饰方法，当使用@Cacheable修饰类时，用于告诉Spring在类级别上进行缓存，程序调用该类的实例的任何方法是都需要缓存，而且共享同一个缓存区；当使用@Cacheable修饰方法时，用于告诉Spring在方法级别上进行缓存，只有当程序调用该方法时才需要缓存。

1. **类级别的缓存**

使用@Cacheable修饰类时，就可控制Spring在类级别进行缓存，这样当程序调用该类的任意方法时，只要传入的参数相同，Spring就会使用缓存。

例如如下示例1：

package com.langsin.anno;

import org.springframework.cache.annotation.Cacheable;

import org.springframework.stereotype.Service;

@Service("userService")

**@Cacheable(value="users")**

public class UserServiceImpl implements UserService {

@Override

public User getUserByNameAndAge(String name, int age) {

System.out.println("执行UserService的getUserByNameAndAge方法");

return new User(name,age);

}

@Override

public User getAdminUser(String name, int age) {

System.out.println("执行UserService的getAdminUser方法");

return new User(name,age);

}

}

上面程序中的粗体字代码指定UserServiceImpl进行类级别的缓存，这样程序中调用该类的任意方法时，只要传入的参数相同，Spring就会缓存。

缓存的意义：当程序第一次调用该类实例的某个方法是，Spring缓存机制会将该方法返回的数据放入指定缓冲区，就是@Cacheable注解的value属性值所指定的缓冲区，以后程序调用该类的实例的任何方法时，只要传入的参数相同，Spring将不会真正执行该方法，而是直接利用缓存区中的数据。

例如如下程序，示例2：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("test.xml");

UserService service = ctx.getBean("userService", UserService.class);

**User user1 = service.getUserByNameAndAge("zhangsan", 22);**

**User user2 = service.getAdminUser("zhangsan", 22);**

System.out.println(user1==user2);

}

上面程序的两行粗体字代码先后调用了UserServiceImpl的两个不同方法，但由于程序传入的方法参数相同，因此Spring不会真正执行第二次调用的方法，而是直接复用缓存区中的数据。

类级别的缓存默认以所有方法参数作为key来缓存方法返回的数据，同一个类不管调用哪个方法，只要调用方法时传入的参数相同，Spring都会直接利用缓存区中的数据。

使用@Cacheable是可指定如下几个属性：

* value：必需属性。该属性可指定多个缓存区的名字，用于指定将方法返回值放入指定的缓冲区内。
* key：通过SpEL表达式显示指定缓存的key。
* condition：该属性指定一个返回boolean值的SpEL表达式，只有当该表达式返回true时，Spring才会缓存方法返回值。
* unless：该属性指定一个返回boolean的SpEL表达式，当该表达式返回true时，Spring就不缓存方法返回值。

例如，将上面程序中UserServiceImpl该为如下格式：

@Service("userService")

**@Cacheable(value="users",key=”#name”)**

public class UserServiceImpl implements UserService {

// 省略的内容

}

显示指定以name参数作为缓存的key，只要调用的方法具有相同的name参数，Spring缓存机制就会生效。

condition属性与unless属性的功能基本相似，但规则恰好相反，当condition指定的条件为true时，Spring缓存机制才会执行缓存，当unless属性指定的条件为true时，Spring缓存机制就不执行缓存。

@Service("userService")

**@Cacheable(value="users",condition=”#age<100”)**

public class UserServiceImpl implements UserService {

// 省略的内容

}

1. **方法级别的缓存**

使用@Cacheable修饰方法时，就可控制Spring在方法级别进行缓存，这样当程序调用该方法时，只要传入的参数相同，Spring就会使用缓存。

示例如下：

package com.langsin.anno;

import org.springframework.cache.annotation.Cacheable;

import org.springframework.stereotype.Service;

@Service("userService")

public class UserServiceImpl implements UserService {

@Override

**@Cacheable(value="users1")**

public User getUserByNameAndAge(String name, int age) {

System.out.println("执行UserService的getUserByNameAndAge方法");

return new User(name,age);

}

@Override

**@Cacheable(value="users1")**

public User getAdminUser(String name, int age) {

System.out.println("执行UserService的getAdminUser方法");

return new User(name,age);

}

}

上面两行粗体代码指定两个方法分别使用不同的缓存区，这意味着这两个方法都会缓存，但由于使用了不同的缓存区，因此它们不能共享缓存数据，如果缓存区相同，即都为user1或者user2，则可以共享缓存数据，如果参数相同，则其中一个不能执行。

### 2.5.3 使用@CacheEvict清除缓存

被@CacheEvict注解修饰的方法可用于清除缓存，使用@CacheEvict注解时可指定如下属性。

* vlaue：必需属性。用于指定该方法用于清除哪个缓存区的数据。
* allEntries：该属性指定是否清空整个缓存区。
* beforeInvocation：该属性指定是否在执行方法之前清除缓存。默认是在方法成功之后才清除缓存。
* condition：该属性指定一个SpEL表达式，只有当该表达式为true时才清除缓存。
* key：通过SpEL表达式显示指定缓存的key。

为UserServiceImpl类增加两个方法，分别用于清空指定缓存和清空缓存，如下代码所示：

package com.langsin.anno;

import org.springframework.cache.annotation.CacheEvict;

import org.springframework.cache.annotation.Cacheable;

import org.springframework.stereotype.Service;

@Service("userService")

public class UserServiceImpl implements UserService {

@Cacheable(value="users")

public User getUserByNameAndAge(String name, int age) {

System.out.println("执行UserService的getUserByNameAndAge方法");

return new User(name,age);

}

@Cacheable(value="users")

public User getAdminUser(String name, int age) {

System.out.println("执行UserService的getAdminUser方法");

return new User(name,age);

}

**@CacheEvict(value="users")**

**public void evictUser(String name, int age) {**

**System.out.println("只要此方法执行就清除name和age对应的缓存");**

**}**

**@CacheEvict(value="users",allEntries=true)**

**public void evictAll(String name, int age) {**

**System.out.println("只要此方法执行就清除整个缓存");**

**}**

}

上面程序中第一个@CacheEvict注解只是指定了value=”user”，这表明该注解用于清除users缓存区中的数据，程序将会根据传入的name、age参数清除对应的数据。第二个@CacheEvict注解则指定了allEntries=true，这表明该方法将会清空整个users缓存区。

使用如下主程序：

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("test.xml");

UserService service = ctx.getBean("userService", UserService.class);

User user1 = service.getUserByNameAndAge("zhangsan", 22);

// 此方法执行将移除缓存区中的user对象

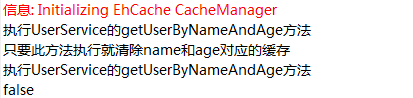
**service.evictUser("zhangsan", 22);**

User user2 = service.getUserByNameAndAge("zhangsan", 22);

System.out.println(user1==user2);

}

执行效果如下：



## 2.6 Spring的事务

Spring的事务管理不需要与任何特定的事务API耦合。对不同的持久层访问技术，编程式事务提供了一致的事务编程风格，通过模板化操作一致性地管理事务。声明式事务基于Spring AOP实现，但并不需要开发者真正精通AOP技术就可以容易地使用Spring的声明式事务管理。

### 2.6.1 Spring支持的事务策略

Java EE应用的传统事务有两种策略：全局事务和局部事务。全局事务由应用服务器管理，需要底层服务器的JTA支持。局部事务和底层采用的持久化技术有关，当采用JDBC持久化技术时，需要使用Connection对象来操作事务；采用Hibernate持久化技术时，需要使用Session对象来操作事务。

全局事务可以跨多个事务性资源；使用局部事务，应用服务器不需要参与事务管理，因此不能保证跨多个事务性资源的事务的正确性。实际开发时大部分应用都使用单一的事务性资源。

下图对比了JTA全局事务、JDBC局部事务、Hibernate事务的事务操作代码。

JTA全局事务 JDBC局部事务 Hibernate事务

Transaction tx = ctx.lookup(..); Connection conn = getConnection(); Session sess = getSession();

conn.setAutoCommit(false); Transaction tx = sess.beginTransaction()

//业务实现 //业务实现 //业务实现

if 正常 if 正常 if 正常

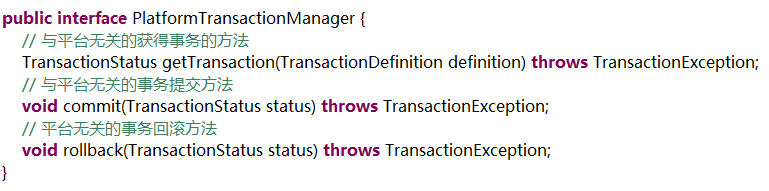
tx.commit(); conn.commit(); tx.commit();

if 失败 if 失败 if 失败

tx.rollback(); conn.rollback(); tx.rollback();

从上图可以看出，当采用传统的事务编程策略时，程序代码必然和具体的事务操作代码耦合，这样造成的后果是；当应用程序需要在不同的事务策略之间切换时，开发者必须手动修改程序代码。如果使用Spring事务策略，就可以改变这种情况。

Spring事务策略是通过PlatformTransactionManager接口体现的，该接口是Spring事务策略的核心。该接口的源代码如下：



PlatformTransactionManager是一个与任何事务策略分离的接口，随着底层不同事务策略的切换，应用必须采用不同的实现类。PlatformTransactionManager接口没有与任何事务性资源捆绑在一起，它可以适用于任何的事务策略，结合Spring的IoC容器，可以向PlatformTransactionManager注入相关的平台的特性。

PlatformTransactionManager接口有许多不同的实现类，应用程序面向与平台无关的接口编程，当底层采用不同的持久层技术时，系统只需要使用不同的PlatformTransactionManager的实现类即可——而这种切换通常由Spring容器负责管理，应用程序既无须与具体的事务API耦合，也无须与特定实现类耦合，从而将应用和持久化技术、事务API彻底分离开来。

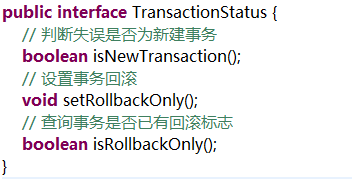
在PlatformTransactionManager接口内，包含一个getTransaction(TransactionDefinition definition)方法，该方法根据TransactionDefinition参数返回一个TransactionStatus对象。TransactionStatus对象表示一个事务，该对象被关联在当前执行的线程上。

getTransaction(TransactionDefinition definition)返回的TransactionStatus对象，可能是一个新的事务，也可能是一个已经存在的事务对象。如果当前执行的线程已经处于事务管理下，则返回当前线程的事务对象，否则创建一个新的事务对象返回。

TransactionDefinition接口定义了一个事务规则，该接口必须指定如下几个属性值。

* 事务隔离：当前事务和其他事务的隔离程度。例如，这个事务能否看到其他事务未提交的数据等。
* 事务传播：通常在事务中执行的代码都会在当前事务中运行。但是，如果一个事务上下文已经存在，有几个选项可以指定该事务性方法的执行行为。
* 事务超时：事务在超时前能运行多久，也就是事务的最长持续时间。如果事务一直没有被提交或回滚，将在超出该时间后，系统自动回滚。
* 只读状态：只读事务不修改任何数据。只读事务是非常有用的优化。

TransactionStatus代表事务本身，它提供了简单的控制事务执行和查询事务状态的方法，这些方法在所有的事务API中都是相同的。TransactionStatus接口的源代码如下：



Spring具体的事务管理由PlatformTransactionManager的不同实现类来完成。在Spring容器中配置管理类时，必须针对不同的环境提供不同的实现类。

下面提供了不同的持久层访问环境及其对应的PlatformTransactionManager实现类的配置。

### 2.6.2 声明式事务管理

**1、使用XML配置的方式**

Spring的事务管理，提供了一个DataSourceTransactionManager类，用于进行声明式事务管理类，该类在org.springframework.jdbc.datasource包下，该类中包含一个dataSource属性用于配置一个数据源。配置如下

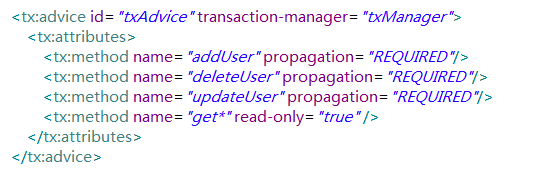


事务管理器在配置时引用了dataSource对象，此对象为数据库连接池对象，表示事务管理器对事务的管理是通过连接池对象中的连接进行管理。事务管理器是通过消息通知的方式起作用，所以需要为事务管理来配置一个事务通知消息，即事务管理器在接收到一个消息通知时才会开启事务的操作。配置如下：

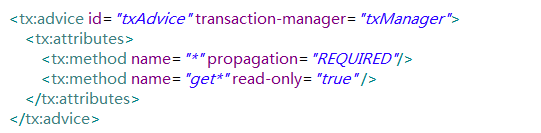
配置通知消息需要使用<tx:advice>标签，此标签包含id、transaction-manager两个重要属性。

* id：消息通知的唯一标识ID
* transaction-manager：消息通知发出后交由哪个事务管理器去处理。

在<tx:advice>标签下含有<tx:attributes>标签，用于配置在执行什么方法时才会触发消息发送。配置方法使用<tx:method>方法，如下所示：

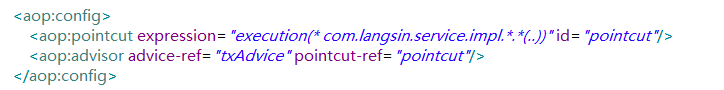


在配置方法时可以使用统配符的方式，即\*，所以在开发时建议如果是查询数据方法以get开头，那么上面配置就可以改为如下格式：



以上为消息通知的配置，表示在调用XXX方法时，如果在消息通知的配置管理内，则会触发消息推送，advice会将消息告知给相应的事务管理器，去执行事务管理。消息通知配置仅仅是定义了在遇到XXX方法执行时会触发消息通知，但是在整个应用中要执行的方法太多，不可能让Spring去监控所有的方法，这样会大大降低应用的响应效率。所以需要有这样一种机制，即在合理的情况下来启动消息通知。

在软件开发过程中，持久层的操作是dao层，而事务管理是对dao层的事务管理，而dao层是对service层提供服务的，而service层在业务处理时有可能会因为业务稍微复杂，需要调用几个dao来完成整体的业务流程，这时就涉及到事务，即一个dao失败就需要全部失败，全部成功才能成功。所以，启动消息通知的操作就需要在servcie层来完成。但是service层时纯粹的业务层，而消息的触发相当于公共业务，所以建议使用aop来完成此操作。配置如下：

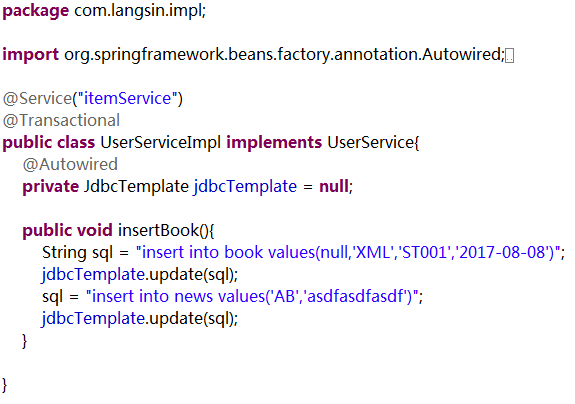


**2、使用@Transaction的方式**

使用注解的方式，配置更为简洁，可以去掉事务的消息通知配置，只需要在Spring的配置文件中，加入事务驱动即可，然后在需要使用事务的方法或Service层类中使用@Transaction注解即可，如下所示：



Service层类或方法使用注解，如下所示：



# 3、 MyBatis的基本用法

MyBatis是一个简化和实现了 Java 数据持久化层(persistence layer)的开源框架，它抽象了大量的JDBC冗余代码，并提供了一个简单易用的API和数据库交互。

MyBatis的前身是ibatis，ibatis于2002年由ClintonBegin创建。MyBatis3是ibatis的全新设计，支持注解和Mapper。

MyBatis流行的主要原因在于它的简单性和易使用性。在Java应用程序中，数据持久化层涉及到的工作有：将从数据库查询到的数据生成所需要的Java对象；将Java对象中的数据通SQL持久化到数据库中。

MyBatis通过抽象底层的JDBC代码，自动化SQL结果集产生Java对象、Java对象的数据持久化数据库中的过程使得对SQL的使用变得容易。

## 3.1 JDBC编程的问题

在Java EE应用开发中，必不可免的要与数据库打交道，与数据库的交互一般都是通过定一个DAO层（数据库访问）来完成这个操作。定义DAO层的主要是原因是为了解决在开发过程中业务与数据的分离，可以在不同的层次上解决不同的问题，而不至于让所有代码冗余在一起，不利于后期的维护。

但是，即使在开发中进行了分层，使用单纯的Java的JDBC编程仍然存在大量的问题。如下代码所示：

try {

Class.forName("com.mysql.jdbc.Driver");

**String url = "jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8";**

**String user = "root";**

**String password = "123456";**

**Connection conn = DriverManager.getConnection(url, user, password);**

**String sql = "select \* from user where user\_name=?";**

ps = conn.prepareStatement(sql);

**ps.setString(1, name);**

rs = ps.executeQuery();

while(rs.next()){

us = new User();

**us.setUserId(rs.getString("user\_id"));**

**us.setUserName(rs.getString("user\_name"));**

**us.setUserPass(rs.getString("user\_pass"));**

**us.setCname(rs.getString("cname"));**

}

conn.close();

} catch (Exception e) {

e.printStackTrace();

}

如上述粗体代码中所示，在进行JDBC编程时就会存在如下问题：

1、数据库的连接，使用时就需要连接，不需要就关闭，频繁的打开与关闭连接，势必会造成数据库资源的浪费，降低数据库的性能。

解决方案：数据库连接池。

2、将要执行的SQL语句以硬编码的方式写入到程序中，如果需求发生变动，则需要更改代码，不利于系统的维护。

解决方案：将SQL语句配置在XML文件中，即使SQL语句更改，也不需要更改Java源程序。

3、如果执行的SQL中含有占位符，则需要向执行SQL语句的PreparedStatement对象中传入参数，如果查询条件发生更改，则占位符的数目也需要改动，那么传入的参数个数会随之发生变动。

解决方案：将执行SQL语句中的占位符、参数，同样配置在XML文件中。

4、执行查询后得到的ResultSet结果集中，在获取数据时将数据表的字段名称，同样以硬编码的方式写死在程序中，不利于系统的维护。

解决方案：将查询的结果集，自动映射为Java对象。

以上所有问题的解决，就是数据库持久层框架存在的意义，而mybatis这个访问数据库持久层框架就很好的解决了以上所有的问题。

## 3.2 MyBatis简介

MyBatis本是apache的一个开源项目iBatis, 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis 。它支持普通 SQL查询，存储过程和高级映射的优秀持久层框架。MyBatis 消除了几乎所有的JDBC代码和参数的手工设置以及结果集的检索。MyBatis 使用简单的 XML或注解用于配置和原始映射，将接口和 Java 的POJOs（Plain Old Java Objects，普通的Java对象）映射成数据库中的记录。

### 3.2.1 MyBatis与Hibernate的区别

1、hibernate是全自动，而mybatis是半自动。hibernate完全可以通过对象关系模型实现对数据库的操作，拥有完整的JavaBean对象与数据库的映射结构来自动生成sql。而mybatis仅有基本的字段映射，对象数据以及对象实际关系仍然需要通过手写sql来实现和管理。

2、hibernate数据库移植性远大于mybatis。hibernate通过它强大的映射结构和sql语言，大大降低了对象与数据库（oracle、mysql等）的耦合性，而mybatis由于需要手写sql，因此与数据库的耦合性直接取决于程序员写sql的方法，如果sql不具通用性而用了很多某数据库特性的sql语句的话，移植性也会随之降低很多，成本很高。

3、hibernate拥有完整的日志系统，mybatis则欠缺一些。hibernate日志系统非常健全，涉及广泛，包括：sql记录、关系异常、优化警告、缓存提示、脏数据警告等；而mybatis则除了基本记录功能外，功能薄弱很多。

4、mybatis相比hibernate需要关心很多细节。hibernate配置要比mybatis复杂的多，学习成本也比mybatis高。但也正因为mybatis使用简单，才导致它要比hibernate关心很多技术细节。mybatis由于不用考虑很多细节，开发模式上与传统jdbc区别很小，因此很容易上手并开发项目，但忽略细节会导致项目前期bug较多，因而开发出相对稳定的软件很慢，而开发出软件却很快。hibernate则正好与之相反。但是如果使用hibernate很熟练的话，实际上开发效率丝毫不差于甚至超越mybatis。

5、sql直接优化上，mybatis要比hibernate方便很多。由于mybatis的sql都是写在xml里，因此优化sql比hibernate方便很多。而hibernate的sql很多都是自动生成的，无法直接维护sql；虽有hql，但功能还是不及sql强大，见到报表等变态需求时，hql也歇菜，也就是说hql是有局限的；hibernate虽然也支持原生sql，但开发模式上却与orm不同，需要转换思维，因此使用上不是非常方便。总之写sql的灵活度上hibernate不及mybatis。

6、Hibernate有更好的二级缓存机制，可以使用第三方缓存。而MyBatis的二级缓存机制不佳。

### 3.2.2 MyBatis的作用及特点

mybatis是一个持久层框架，它的主要作用就是可以让程序开发人员将主要精力放在SQL语句的编写上，通过mybatis提供的映射方式，自由灵活的创建出满足业务需求的各种SQL语句，不管是单表查询，还是多表查询，都可以很好的完成。而这种灵活生成是一种半自动化的生成，即还是需要程序人员进行编写SQL。

如果要执行的SQL语句中存在占位符，那么在执行之前，需要向执行该SQL语句的PreparedStatement对象传入参数。mybatis可以将参数进行自动的输入映射，查询之后的结果集，mybatis可以灵活的映射成Java对象，这就是mybatis最为重要的输入映射和输出映射。

如果使用mybatis框架，则需要去它的官方网站下载，该地址为：<https://github.com/mybatis/mybatis-3/releases>

目前最新版本为3.4.1版。

下面来看一下mybatis的结构示意图，如下图所示：

SqlMapConfig.xml,是mybatis的全局配置文件，配置了数据源、事务等mybatis的运行环境，同时还需需要大量的mapper.xml映射文件，来配置mybatis的映射输入和映射输出。

mybatis操作数据库需要使用SqlSession会话，而SqlSession会话的创建需要依赖一个SqlSessionFactory会话工厂，通过工厂会话来创建SqlSession对象。

SqlSession会话主要用于对数据库进行操作，包括一系列的增、删、改、查操作，但是SqlSession并不直接对数据库进行操作，而是底层通过一个Executor执行器来完成。

输入参数类型：

在statement对象中封装了执行SQL语句的参数对象，参数对象可以是任何的java类型，包括基本数据类型和引用类型。

输入参数类型：

在statement对象中封装了执行SQL语句的参数对象，参数对象可以是任何的java类型，包括基本数据类型和引用类型。

Executor执行器，SqlSession内部就是通过它来完成数据库的执行操作。Executor执行器的运行需要依赖下面的对象。

mapped statement，底层封装对象，Executor执行器执行数据库操作时，需要知道要执行的SQL语句是什么？执行SQL语句传入的参数是什么？以及完成查询后返回的Java类型是什么？都封装在这个对象中。

数据库

## 3.3 入门程序

以用户订单表为例，通过mybatis框架完成最简单的增、删、改、查操作。

### 3.3.1 设置mybatis的运行环境

下载最新版本的mybatis运行jar包，解压开里面会有一个mybatis-3.4.1.jar，在该目录下会存在一个同名的PDF文件，为该版本的mybatis的操作指南。lib目录为mybatis运行所依赖的jar包，同样需要配置到工程中来。

在该lib目录中ant为构建工具包，cglib动态代理工具包，log4j日志输出jar包，ognl对象图导航语言包，log4j日志工具包，slf4j简单日志包。mybatis所依赖的log4j日志工具包，可以使软件开发人员在使用mybatis进行开发时输出日志，以便于跟踪除错。

**1、配置log4j日志配置文件**

在工程下建立一个config目录，此下配置log4j.properties属性文件，内容如下：

# 设置log4j的日志级别，开发环境下为DEBUG，生产环境下为info或error

log4j.rootLogger=DEBUG,stdout

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

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

log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n

**2、配置mybatis的全局配置文件sqlMapConfig.xml**

此配置文件，文件名称不固定，可由开发人员自己指定，同样配置在config目录下，如下所示：

<?xml version="1.0" encoding="GBK"?>

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

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

<configuration>

<!-- 以下配置不需掌握，和Spring整合后，以下配置将被废除 -->

<environments default="development">

<environment id="development">

<transactionManager type="JDBC" />

<dataSource type="POOLED">

<property name="dirver" value="org.mysql.jdbc.Driver"/>

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

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

<property name="password" value="123456"/>

</dataSource>

</environment>

</environments>

</configuration>

### 3.3.2 根据用户ID进行查询

**1、创建配置SQL语句的mapper.xml映射文件**

创建一个根据用户ID进行查询的示例，在mybatis中需要为此开发操作准备一个映射文件，比如现在是进行用户查询，因此创建一个user.xml配置文件，格式如下：

<?xml version="1.0" encoding="GBK"?>

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

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

<!-- 命名空间的作用是对SQL进行分类管理，起到隔离SQL的作用，如果使用mapper代理，namespace有特殊重要作用 -->

<mapper namespace="user">

<!-- 通过select元素配置查询语句，id属性值，可以理解为标识映射文件中的SQL，因为最终将SQL语句封装到mappedStatement对象中去，所以id为statement的id ，在此SQL中，使用#{}表示占位符，花括号中的值为userId表示要接收的参数，参数的名称为userId，如果输入的参数是基本数据类型，#{}中的参数可以为任意名称，但是一般都遵循开规范些形参名，parameterType为SQL要接收的参数的数据类型，而resultType为select进行查询后，返回的数据类型，因为是单条查询，所以映射为一个Java Bean对象-->

<select id="findUserById" parameterType=”int” resultType=”com.langsin.user.vo.User”>

select \* from user where user\_id=#{userId}

</select>

</mapper>

**2、创建mapper.xml映射文件对应的Java Bean**

Java Bean对象User代码如下：

package com.langsin.user.vo;

public class User {

private String user\_id;

private String user\_name;

private String role\_id;

private String user\_pass;

private String cname;

private String user\_image;

….省略getter和setter方法

}

注意：在创建对应的JavaBean时，mybatis默认是以数据库表中字段列名与Java Bean中的成员变量名进行对应的，所以在定义成员变量名是需要与表字段的列名保持一致。

**3、配置mapper.xml文件**

为了让配置的mapper.xml文件起作用，那么需要在mybatis的全局配置文件sqlMapConfig.xml中加载该文件，如下所示：

<?xml version="1.0" encoding="GBK"?>

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

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

<configuration>

<!-- 以下配置不需掌握，和Spring整合后，以下配置将被废除 -->

<environments default="development">

<environment id="development">

<transactionManager type="JDBC" />

<dataSource type="POOLED">

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

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

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

<property name="password" value="123456"/>

</dataSource>

</environment>

</environments>

<mappers>

<mapper resource="sqlmap/user.xml" />

</mappers>

</configuration>

**4、创建测试方法**

新建一个测试包，在该包下建立测试类，因为使用JUnit测试，所以需要在工程中加入JUnit的jar包。

主程序代码如下：

package com.mybatis.test;

import java.io.InputStream;

import org.apache.ibatis.io.Resources;

import org.apache.ibatis.session.SqlSession;

import org.apache.ibatis.session.SqlSessionFactory;

import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import org.junit.Test;

import com.langsin.user.vo.User;

public class MyBatisTest {

// 根据ID来查询唯一条记录，返回一个Java对象

@Test

public void getUserByIdTest() throws Exception{

String resource = "sqlMapConfig.xml";

InputStream stream = Resources.getResourceAsStream(resource);

// 创建一个SqlSession的会话工厂

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(stream);

// 创建SqlSession会话对象

SqlSession session = factory.openSession();

// 第一个为statement的Id，第二个parameter的参数

User user = session.selectOne("**user.findUserById**", "000010");

System.out.println(user);

// 释放资源

session.close();

}

}

在上述程序中，通过SqlSession对象的selectOne方法进行查询单条记录，此方法需要两个参数，第一个参数为statementId，在配置mapper.xml文件时（user.xml），select元素的id属性，为SQL语句的标识，但是mapper的配置SQL在执行时，最终会被mybatis封装到一个statement对象中去，而这个id也就是statement对象的id，这个参数的字符格式为：命名空间+”.”+属性ID的方式，例如：user.findUserById。

### 3.3.3 根据用户名称进行模糊查询

**1、首先在mapper.xml文件中使用select元素添加SQL语句，如下所示：**

<select id="findUserByName" parameterType="String" resultType="com.langsin.user.vo.User">

select \* from user where cname like '%${value}%'

</select>

在上述的SQL语句中没有使用#{}占位符的方式，因为这里要进行模糊匹配查询，要用到%号，要进行字符串拼接，所以使用了’%${value}%’，花括号中此时只能使用value，而不能使用其他变量名，使用${value}的方式，表示对参数不做任何修饰直接进行SQL拼接，但这样做具有一定的危险性，会导致程序安全性能降低，原因就是SQL注入，就是因为使用了字符串拼接。**因此不建议使用**。

同样也可以使用#{}占位符的方式，只不过需要程序开发人员在对传入的参数进行处理，更改为”%参数%”的方式。

1. **创建mapper.xml文件对应的JavaBean**

此步省略，以为在根据ID查询中已经完成

**3、配置mapper.xml文件**

此步操作同样省略，原因同上。

**4、创建测试方法**

@Test

public void getUserByName() throws Exception{

String resource = "sqlMapConfig.xml";

InputStream stream = Resources.getResourceAsStream(resource);

// 创建一个SqlSession的会话工厂

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(stream);

// 创建SqlSession会话对象

SqlSession session = factory.openSession();

// 第一个为statement的Id，第二个parameter的参数

List<User> list = session.selectList("user.findUserByName", "%张三%");

System.out.println(list);

// 释放资源

session.close();

}

### 3.3.4 用户添加操作

使用mybatis进行插入操作，与根据查询操作类似，都是相同的流程，只不过需要配置的mapper.xml文件，以及运行的主程序不同而已。因此下面示例，省略了2、3步的内容。

**1、使用insert元素配置SQL语句，内容如下：**

<insert id="addUser" parameterType="com.langsin.user.vo.User">

insert into user values(#{user\_id},#{user\_name},#{role\_id},#{user\_pass},#{cname},#{user\_image})

</insert>

在上面配置文件中，parameterType类型为Java 对象，所以在占位符#{}中指定Java对象的属性名称，来接收Java对象的属性值，mybatis同样是通过OGNL对象导航视图的方式获取对象的属性值。

**4、添加测试方法**

@Test

public void insertUser() throws Exception{

String resource = "sqlMapConfig.xml";

InputStream stream = Resources.getResourceAsStream(resource);

// 创建一个SqlSession的会话工厂

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(stream);

// 创建SqlSession会话对象

SqlSession session = factory.openSession();

// 创建一个Java对象作为参数传入

User user = new User();

user.setUser\_id("US00000001");

user.setUser\_name("liubei");

user.setCname("刘备");

int num = session.insert("user.addUser", user);

System.out.println("影响行数为："+num);

// 提交事务

session.commit();

// 释放资源

session.close();

}

#### 1、 返回自增长主键

mysql在执行insert插入提交之前，会自动生成一个自增长主键，通过mysql的提供的函数last\_insert\_id()可以获取到该主键值，但是此函数的执行必须要在insert语句执行之后才可以，所以如果在mybatis中如果想要获取刚插入的子增长主键记录，则在mapper.xml配置文件中，修改如下：

<insert id="addUser" parameterType="**com.langsin.user.vo.User**">

<selectKey keyProperty="**user\_id**" order="AFTER" **resultType**=”Integer”>

select last\_insert\_id()

</selectKey>

insert into user values(#{user\_id},#{user\_name},#{role\_id},#{user\_pass},#{cname},#{user\_image})

</insert>

keyProperty属性的作用：是将查询之后得到的主键值设置到parameterType的对应的属性中。order的作用是指定，selectKey元素的查询是在SQL语句之前还是之后执行，可选值为：BEFORE、AFTER

**注意：只适用于自增长主键**

#### 2、 返回非自增长主键

在mysql中调用uuid()函数可以获取一个唯一的随机的字符串，长度为36位，可以利用此函数的值当做表的主键。那么如果要想使用mybatis来获取此值，执行思路如下：

1. 首先调用select uuid()函数生成字符串
2. 将字符串的值赋值到parameterType参数的属性中，然后执行插入SQL语句

所以mapper.xml配置文件修改如下：

<insert id="addUser" parameterType="com.langsin.user.vo.User">

<selectKey keyProperty="user\_id" **order**="BEFORE" **resultType**="String">

select uuid()

</selectKey>

insert into user values(#{user\_id},#{user\_name},#{role\_id},#{user\_pass},#{cname},#{user\_image})

</insert>

在上面的配置中需要注意，执行uuid函数的操作需要在执行insert语句之前，因此order的值为before，resultType为执行查询之后的返回值类型String。

### 3.3.5 删除用户操作

在mapper.xml配置文件中，添加如下配置：

<delete id="deleteUserById" parameterType="String">

delete from user where user\_id=#{user\_id}

</delete>

使用delete元素完成删除操作。

### 3.3.6 更新用户操作

在mapper.xml配置文件中，添加如下配置：

<update id="updateUser" parameterType="com.langsin.user.vo.User">

update user set user\_id=#{user\_id},user\_name=#{user\_name},user\_pass=#{user\_pass},cname=#{cname} where user\_id=#{user\_id}

</update>

使用update元素完成删除操作。

### 3.3.7 #{}与${}的区别

#{}符号mybatis的占位符，相当于jdbc中的？

${}符号mybatis的拼接符，对传入的参数不做任何修饰，进行简单的字符串拼接。

### 3.3.8 selectOne与selectList的区别

* selectOne方法只能用于发回一条记录。
* selectList用于返回一个List集合，对于selectOne返回一条记录的也可使用selectList，只不过相当于集合中只有一条记录而已。

## 3.4 MyBatis开发Dao

在Java企业级应用的开发中，与数据库交互的持久层的开发Dao是不可缺少的一个环节，使用mybatis开发dao环节如下。

首先定义一个该功能模块的dao接口，这个无须多说，无非是该功能下的一系列方法的声明而已。接下来主要是该dao接口的实现。

使用mybatis进行数据库的开发主要涉及到以下几个类：

* SqlSessionFactory：SqlSession会话的创建工厂，mybatis主要是利用该类完成SqlSession对象的创建即可。因此该类只需要一个就可以，在Spring的IoC容器中考虑使用单例模式管理。
* SqlSessionFactoryBuilder：SqlSessionFactory的创建需要依赖此工具类进行创建，因此只需要当成一个工具类使用即可，不需要Spring进行管理。
* SqlSession：mybatis主要使用该类完成一系列的数据库驱动操作，包括增加、删除、修改、查询。但是SqlSession是一个线程不安全的，因此使用时为了防止出现并发访问导致的数据状态信息不一致的情况，需要将SqlSession定义成方法局部变量来使用。

示例1：Dao接口定义

package com.langsin.user.dao;

import java.util.List;

import com.langsin.user.vo.User;

public interface UserDao {

// 通过id来查询用户

public User getUserById(String id) throws Exception;

// 通过用户名来查询用户

public List<User> getUserByName(String name) throws Exception;

// 根据id删除用户

public int deleteUserById(String id) throws Exception;

// 更新用户信息

public int updateUser(User user) throws Exception;

}

示例2：接口Dao的实现类

package com.langsin.user.dao;

import java.util.List;

import org.apache.ibatis.session.SqlSession;

import org.apache.ibatis.session.SqlSessionFactory;

import com.langsin.user.vo.User;

public class UserDaoImpl implements UserDao {

private SqlSessionFactory factory = null;

// 通过构造注入的方式完成

public UserDaoImpl(SqlSessionFactory factory){

this.factory = factory;

}

@Override

public User getUserById(String id) throws Exception {

SqlSession session = this.factory.openSession();

User user = session.selectOne("user.findUserById", id);

session.close();

return user;

}

@Override

public List<User> getUserByName(String name) throws Exception {

SqlSession session = this.factory.openSession();

List<User> list = session.selectList("user.findUserByName", name);

session.close();

return list;

}

@Override

public int deleteUserById(String id) throws Exception {

SqlSession session = this.factory.openSession();

int result = session.delete("user.deleteUserById", id);

session.commit();

session.close();

return result;

}

@Override

public int updateUser(User user) throws Exception {

SqlSession session = this.factory.openSession();

int result = session.delete("user.upateUser", user);

session.commit();

session.close();

return result;

}

}

测试用例3：创建一个JUnit的测试类

package com.mybatis.test;

import java.io.InputStream;

import org.apache.ibatis.io.Resources;

import org.apache.ibatis.session.SqlSessionFactory;

import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import org.junit.Before;

import org.junit.Test;

import com.langsin.user.dao.UserDao;

import com.langsin.user.dao.UserDaoImpl;

import com.langsin.user.vo.User;

public class UserDaoImplTest {

private SqlSessionFactory factory = null;

@Before

public void setUp() throws Exception {

String resource = "sqlMapConfig.xml";

InputStream stream = Resources.getResourceAsStream(resource);

// 创建一个SqlSession的会话工厂

this.factory = new SqlSessionFactoryBuilder().build(stream);

}

@Test

public void testGetUserById() throws Exception {

UserDao dao = new UserDaoImpl(factory);

User user = dao.getUserById("US00000001");

System.out.println(user);

}

}

从上面使用mybatis开发的dao被称为原始dao的开发，也是目前企业中常用的方式，但这种使用存在以下几个问题：

1. dao接口的实现类中存在大量的重复代码，是否可以将代码提取出来，减轻开发的工作量。
2. 调用SqlSession方法时，将mapper配置中的statement的id硬编码到程序中。
3. 调用SqlSession方法时传入的变量，由于SqlSession使用了泛型 ，即使变量传入错误，也不会引发编译异常，不利于程序人员的开发。

由于以上问题的存在，所以使用mybatis开发一个优秀的dao层结构，可以使用mapper代理的方式。

## 3.5 MyBatis的mapper代理

使用mybatis实现mapper接口的代理，同样需要编写mapper.xml配置文件。而且mapper.xml配置文件需要遵循一定的开发规范才可以：

1、mapper.xml配置文件的namespace必须指定mapper接口的地址。如下所示：

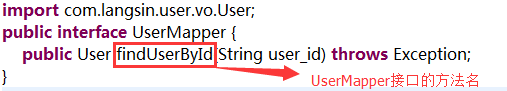
package com.langsin.user.dao;

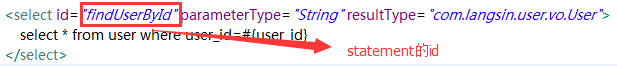
public interface UserMapper {

}



2、mapper接口中方法名和mapper.xml中statement的id一致。如下所示：





3、mapper接口中方法的输入参数要与mapper.xml配置文件中的parameterType保持一致。

4、mapper接口中方法的返回值类型要与mapper.xml配置文件中的resultType保持一致。

只有遵循了如上所述的开发规范，mybatis框架才可以自动的实现mapper接口的动态代理对象。通过sqlSession 的getMapper方法，即可获得接口的动态代理对象，此方法执行需要接口的Class对象作为参数。如果查询返回的是结果集合，只需要将mapper接口中声明的返回值类型是List集合即可。这样mybatis的实现的动态代理对象内部就能判断在进行查询时是调用selectOne还是selectList方法了。

如下所示UserMapper接口1：

package com.langsin.user.dao;

import java.util.List;

import com.langsin.user.vo.User;

public interface UserMapper {

public User findUserById(String user\_id) throws Exception;

public List<User> findUserByName(String cname) throws Exception;

}

测试类2：

package com.mybatis.test;

import java.io.InputStream;

import java.util.List;

import org.apache.ibatis.io.Resources;

import org.apache.ibatis.session.SqlSession;

import org.apache.ibatis.session.SqlSessionFactory;

import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import org.junit.Before;

import org.junit.Test;

import com.langsin.user.dao.UserMapper;

import com.langsin.user.vo.User;

public class UserMapperTest {

private SqlSessionFactory factory = null;

@Before

public void setUp() throws Exception {

String resource = "sqlMapConfig.xml";

InputStream stream = Resources.getResourceAsStream(resource);

// 创建一个SqlSession的会话工厂

this.factory = new SqlSessionFactoryBuilder().build(stream);

}

@Test

public void findUserByIdTest() throws Exception {

SqlSession session = this.factory.openSession();

**UserMapper mapper = session.getMapper(UserMapper.class);**

User user = mapper.findUserById("000010");

System.out.println(user);

sesson.close();

}

@Test

public void findUserByNameTest() throws Exception {

SqlSession session = this.factory.openSession();

**UserMapper mapper = session.getMapper(UserMapper.class);**

List<User> list = mapper.findUserByName("张三");

System.out.println(list);

sesson.close();

}

}

## 3.6 SqlMapConfig.xml配置详解

SqlMapConfig.xml是mybatis的全局配置文件，目前已经学习了environments元素，其下面配置environment元素用于配置mybatis的运行环境。

* transactionManager元素用于配置事务管理。
* dataSource元素用于配置数据源，type属性用于指定数据源类型，值为pooled表示数据库连接池。
* property元素用于配置数据库连接池的属性信息，以硬编码的形式配置在文件中。

SqlMapConfig.xml配置文件还提供了Properties属性元素，可以很好的解决此问题。

### 3.6.1 properties属性加载

properties元素用于加载一个属性文件。通过属性resource属性配置，如下所示：



为了防止属性名称冲突，属性值一般定义为xxx.xxx的形式，例如db.properties属性文件用于配置数据库连接，格式如下：



### 3.6.2 settings全局参数配置

settings元素用于配置mybatis框架运行时的一些性能调优的参数的配置，如果配置不当会影响整个框架的运行。例如：开启二级缓存、开启延迟加载等。

|  |  |  |  |
| --- | --- | --- | --- |
| **setting设置** | **Description描述** | **验证值** | **default** |
| **cacheEnabled** | 在全局范围内启用或禁用缓存配置任何映射器在此配置下。 | true | false | true |
| **lazyLoadingEnabled** | 在全局范围内启用或禁用延迟加载。禁用时，所有协会将热加载。 | true | false | true |
| **aggressiveLazyLoading** | 启用时，有延迟加载属性的对象将被完全加载后调用懒惰的任何属性。否则，每一个属性是按需加载。 | true | false | true |
| **multipleResultSetsEnabled** | 允许或不允许从一个单独的语句（需要兼容的驱动程序）要返回多个结果集。 | true | false | true |
| **useColumnLabel** | 使用列标签，而不是列名。在这方面，不同的驱动有不同的行为。参考驱动文档或测试两种方法来决定你的驱动程序的行为如何。 | true | false | true |
| **useGeneratedKeys** | 允许JDBC支持生成的密钥。兼容的驱动程序是必需的。此设置强制生成的键被使用，如果设置为true，一些驱动会不兼容性，但仍然可以工作。 | true | false | true |
| **autoMappingBehavior** | 指定MyBatis的应如何自动映射列到字段/属性。NONE自动映射。 PARTIAL只会自动映射结果没有嵌套结果映射定义里面。 FULL会自动映射的结果映射任何复杂的（包含嵌套或其他）。 | none  partial  full | partial |
| **defaultExecutorType** | 配置默认执行人。SIMPLE执行人确实没有什么特别的。 REUSE执行器重用准备好的语句。 BATCH执行器重用语句和批处理更新。 | simple  reuse  batch | simple |
| **defaultStatementTimeout** | 设置驱动程序等待一个数据库响应的秒数。 | Any positive integer | not set  (null) |
| **safeRowBoundsEnabled** | 允许使用嵌套的语句RowBounds。 | true | false | false |
| **localCacheScope** | MyBatis的使用本地缓存，以防止循环引用，并加快反复嵌套查询。默认情况下（SESSION）会话期间执行的所有查询缓存。如果localCacheScope=STATMENT本地会话将被用于语句的执行，只是没有将数据共享之间的两个不同的调用相同的SqlSession。 | session  statement | session |
| **dbcTypeForNull** | 指定为空值时，没有特定的JDBC类型的参数的JDBC类型。有些驱动需要指定列的JDBC类型，但其他像NULL，VARCHAR或OTHER的工作与通用值。 | JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER | oter |
| **lazyLoadTriggerMethods** | 指定触发延迟加载的对象的方法。 | A method name list separated by commas | equals,clone,hashCode,toString |
| **defaultScriptingLanguage** | 指定所使用的语言默认为动态SQL生成。 | A type alias or fully qualified class name. | org.apache.ibatis.scripting.xmltags.XMLDynamicLanguageDriver |
| **callSettersOnNulls** | 指定如果setter方法​​或地图的put方法时，将调用检索到的值是null。它是有用的，当你依靠Map.keySet（）或null初始化。注意原语（如整型，布尔等）不会被设置为null。 | true | false | false |
| **logPrefix** | 指定的前缀字串，MyBatis将会增加记录器的名称。 | Any String | not set |
| **logImpl** | 指定MyBatis的日志实现使用。如果此设置是不存在的记录的实施将自动查找。 | SLF4J | LOG4J | LOG4J2 | JDK\_LOGGING | COMMONS\_LOGGING | STDOUT\_LOGGING | NO\_LOGGING | not set |
| **proxyFactory** | 指定代理工具，MyBatis将会使用创建懒加载能力的对象。 | cglib  javassist |  |

### 3.6.3 typeAliases别名

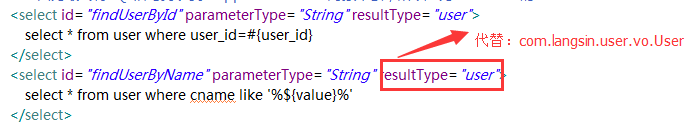
类型别名是为Java类型命名一个短的名字。它只和XML配置有关，只用来减少全限定类型的多于部分，便于开发。

**1、单个别名定义**

例如如下格式：在SqlMapConfig.xml文件中定义别名

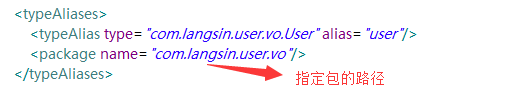


在mapper.xml文件中引用别名

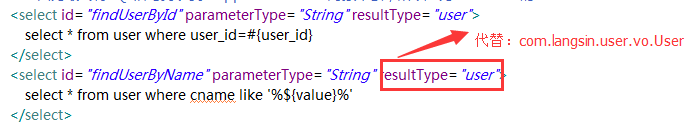


**2、批量别名定义 （经常使用方式）**

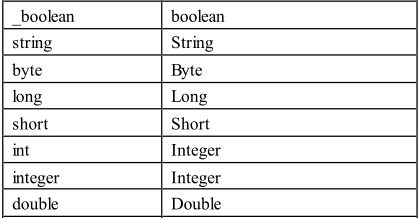
使用package元素，指定一个包的路径，mybatis自动扫描包中的Java类，别名就是类名，首字母大小写都可以。例如如下格式：

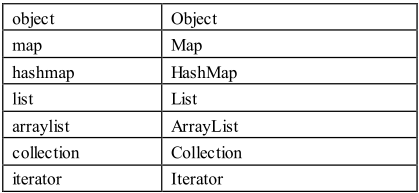
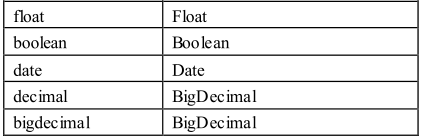


在mapper.xml文件中引用别名。



mybatis支持的内置别名



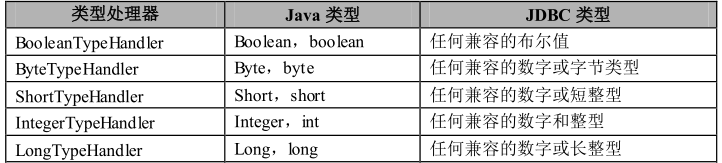


### 3.6.4 typeHandlers类型处理器

无论是mybatis在预处理语句中设置一个参数，还是从结果集中取出一个值时，类型处理器被用来将获取的值以合适的方式转换成Java类型。

通常情况下，mybatis提供的类型处理器满足日常需要，不需要自定义。

mybatis支持的自定义类型：





例如：



简单来讲，typeHandler类型处理器的作用就是将配置文件中的预定义的字符串，转换成一种何时的Java对象。

### 3.6.5 mapper映射配置

**1、通过resource属性加载mapper配置文件**

在SqlMapConfig.xml全局配置文件中，在mappers元素下，使用mapper加载映射配置，可以使用resource属性加载单个配置文件，文件路径是在类加载路径下开始的。如下所示：



**2、通过class属性加载mapper配置文件**

在mapper元素中，可以通过class属性指定mapper接口的方式，来加载映射，但是要遵循一定的规范。

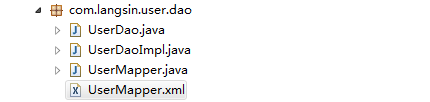
* 首先必须遵循mapper代理的规范，如statement的id与mapper接口的方法名一致等。
* mapper的接口名必须与mapper文件名保持一致。
* 必须在同一文件夹下。

如下所示：

mapper接口的加载方式：



mapper配置文件的位置：



实际上，使用mybatis框架加载mapper配置文件时，如果发现是通过mapper元素的class属性的方式去加载，则会在该接口的所在包下扫描mapper文件并去加载该配置文件。如果mapper配置文件与mapper接口名不一致，则不加载。

**3、批量加载mapper映射（推荐使用方式）**

通过使用package元素的name属性指定mapper接口所在的包名，mybatis自动扫描该包下的所有的mapper接口，并查找与接口名相同的mapper配置文件进行加载，前提是也需要遵循上面2中的规范。

如下所示：



这是项目开发所推荐的使用方式。

## 3.7 输入映射

上面利用mybatis所做的这些增加、删除、修改、查询的操作中，传入参数的输入映射都是简单的Java类型，比如int、string等，稍微复杂一点的也就是一个简单的自定义的Java Bean对象即（POJO—Plain Ordinary Java Object），如果为了支持一个复杂的多表联合查询、嵌套子查询等，需要传入的参数可能不是针对一个表的查询条件，在这种情况下，就需要指定自定义包装类型的POJO或者传入一个hashMap对象。

### 3.7.1 使用hashMap集合

如果在执行SQL语句时，传入的参数较为复杂，是针对好几个表的查询条件，而在开发时针对每张表都会定义一个与之对应的Java Bean对象。但现在确实多表的操作，而且条件也是针对不同表中的字段进行过滤查询，这时可以考虑使用HashMap集合作为parameterType类型传入，其中#{}中的字符为map的key值即可。

如下所示：

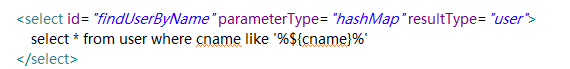
public interface UserMapper {

public User findUserById(String user\_id) throws Exception;

**public List<User> findUserByName(Map<String,String> map) throws Exception;**

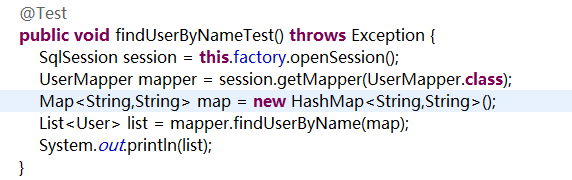
}

mapper.xml文件如下：



注意：如果不小心${}括号中的字符串没有写入map中的key值，或者不小心些错了，mybatis也不会报错，而不是相当于没有这个查询条件，即查询全部内容。

测试代码如下：



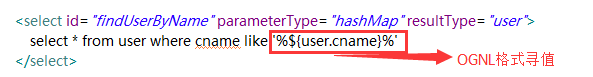
在Java基础集合框架的学习中，我们知道HashMap的key值和value值都可以是任意的Java类型，绝大部分情况下，key值都是字符串，而value值则是花样繁多。在mybatis中，如果parameterType的传入映射是hashMap而value值则是一个Java对象，不是String简单字符串时，可以使用OGNL的方式在占位符中寻值。

如下所示：

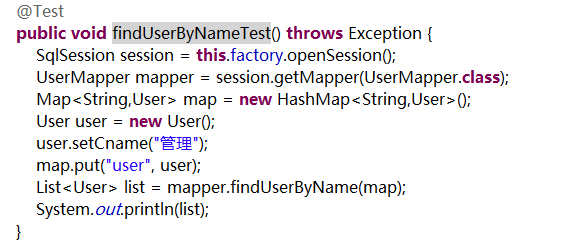
示例代码1：接口定义



示例代码2：mapper.xml配置



测试代码3：

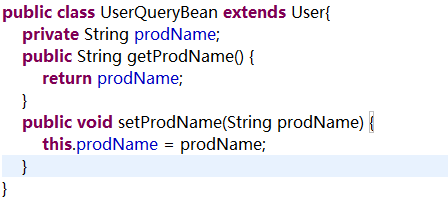


### 3.7.2 使用POJO自定义类型

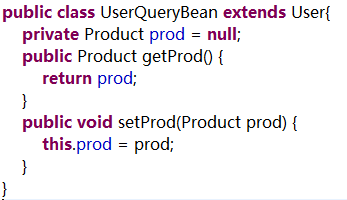
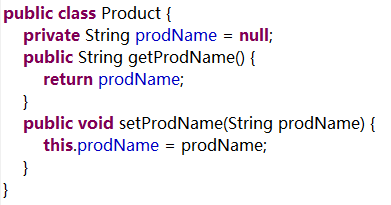
如果不喜欢使用hashMap集合作为传入参数parameterType的输入映射，mybatis还为程序开发人员提供了另外一种方式，即自定义POJO类型，但最底层原理都是相通的。

例如：查询购买某种商品的所有的“王”姓用户的信息，则查询条件既有User表中信息，又有Product商品表中信息。在此种情况下，不要修改任意的一个VO对象即与表一一对应的Java Bean对象，而是在原有基础上扩展即可。

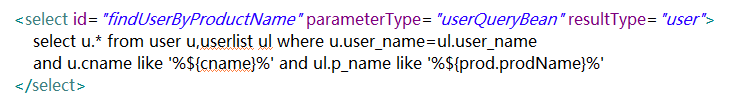
代码1：在User Bean基础上扩展



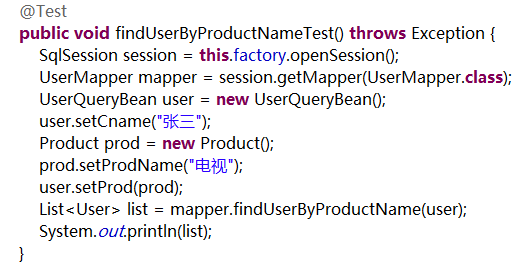
在上面代码中，既可以定义String类型变量，也可以组装一个Product对象都可以，如下所示：

代码2：mapper.xml配置文件如下



代码3：测试代码如下



## 3.8 输出映射

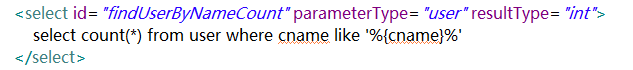
mybatis的输出映射支持两种形式，一种形式为resultType，一种形式为resultMap。

### 3.8.1 resultType映射

之前所做操作返回值类型映射都是使用resultType形式完成的，不管是返回单个Java对象还是返回Java的List集合，resultType都是指向一个Java Bean对象。只不过具体是返回一个对象还是返回一个集合必须在mapper接口的方法的返回值类型中指明。这是因为mybatis在生成的动态代理对象中，是根据返回值类型来确定是调用selectOne方法还是selectList方法。

需要注意的是，Java Bean的属性名称必须与查询结果的字段表保持一致才可以映射成功，如果对应不一致的，则Java Bean对应的属性的值基本类型为0或false，引用类型为null。

resultType属性还可以指定为简单类型，如Java的基本数据类型或String字符串。例如获取查询结果的所有记录的总条数。



resultType属性还可以指定为hashMap，这个很好理解。无非就是将查询的结果的每条记录，都映射为一个HashMap对象，字段名为key值，字段值为value值。

注意：如果是执行联合查询的结果，查询数据是从多个表中筛选出来的，而且字段名也进行了重命名，则可以考虑使用resultType的属性为hashMap。因为使用hashMap不需要考虑字段名，映射为hashMap对象时，是以最终的查询结果的字段名为准作为key值对应的。

### 3.8.2 resultMap映射

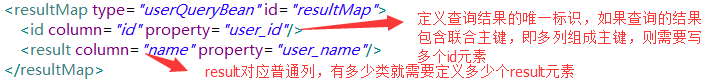
在mybatis框架中，使用resultMap属性完成高级输出结果的映射。例如查询结果的列名与POJO对象的属性名不一致，可以通过定义一个resultMap对查询结果列名和POJO属性名之间做一个映射关系。

首先定义一个resultMap元素，该元素包含两个主要属性，type和id，type表示该resultMap元素所对应的POJO的类型，id表示resultMap的唯一标识名。在resultMap元素下还包括两个重要子元素。

* id元素：查询结果集的唯一标识列，即主键列。
* result元素：查询结果集的普通列。

两个元素都包含相同的属性：其中主要使用column和property属性，column表示查询结果的列名，property表示resultMap元素type属性所对应的POJO的属性名。

如下所示：



定义完成后，则可以在其他insert、select元素中的resultMap属性中引用该resultMap定义的高级映射。

如下所示：



在上面mapper.xml配置中，select元素的resultMap属性指定了resultMap元素的id属性值，通过关联，select语句查询后返回的数据映射类型为userQueryBean所对应的POJO类型。

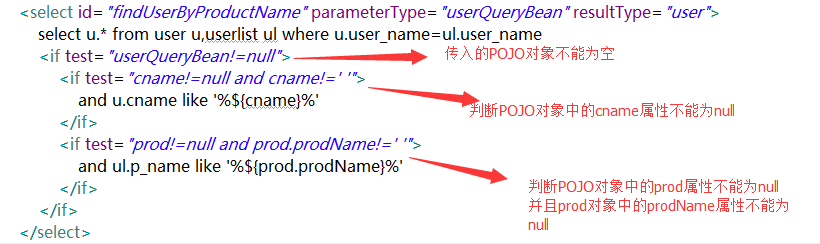
总结：使用resultMap进行输出映射，只有查询出来的列名与POJO中的属性名一致，才可以映射成。如果不一致，通过定义一个resultMap对列名和属性名之间作一个映射也可以完成。

## 3.9 动态SQL

mybatis的核心就是对SQL语句进行灵活的操作，甚至还可以通过表达式进行判断，对SQL语句进行灵活的拼装，组成最终的SQL，这就是动态SQL的含义。

例如，在进行复杂查询时，查询条件可能是多个，也可能是一个，也就是说，查询条件不确定。需要根据查询条件，来确定最终执行的SQL语句。

使用mybatis的动态SQL时，需要标签元素的支持，最常用的为where元素和if元素，如下所示：

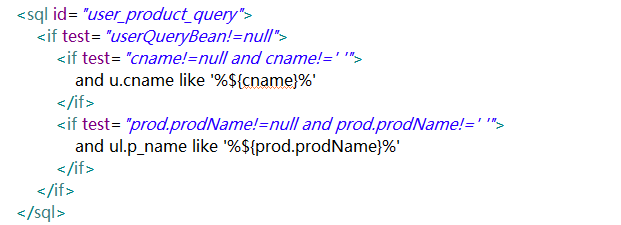


在if判断中，判断是否为空时，单引号中间不能使用空格隔开，直接书写两个单引号即可。上例中没有使用where元素，是因为上述SQL中u.user\_name=ul.user\_name是必要的连接条件，所以前面肯定会有where关键字。

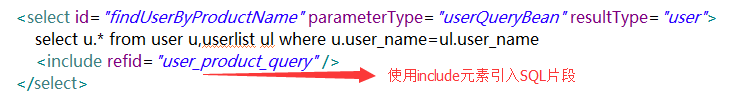
### 3.9.1 引用SQL片段

在mapper.xml配置文件中如果存在大量的负责查询而且查询条件相同，那么则可以抽取成一个SQL片段，在下面的SQL中引用该片段即可。

例如：定义一个SQL片段，id属性用于标识该SQL片段的唯一性



示例2：引用该SQL片段



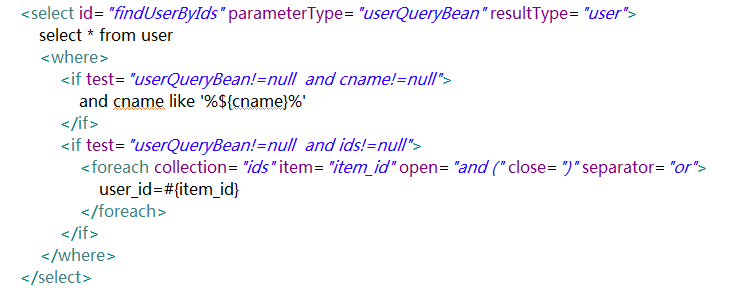
通过SQL片段的定义，不仅可以减少mapper.xml配置文件的书写，而且方便程序人员的开发。

### 3.9.2 使用foreach遍历

在进行复杂SQL查询时，往往会遇到这样一种情况，对于某个列的值，好几个值都符合条件，使用SQL书写格式如下：

1. **where column=’value’ or column=’value2’ or column=’value3’…..**

执行此SQL时传入的参数就需要是个集合，在mybatis中，对于这种情况的处理，使用foreach元素解决。



foreach标签用于遍历传入的集合，属性如下：

* collection：指定要遍历的集合对象
* item：定义标识指向每次遍历时得到的对象
* open：开始遍历时要拼接的字符串
* close：结束遍历时要拼接的字符串
* separator：遍历两个对象中间要拼接的字符串

如果ids不为null，最终SQL拼接完成如下：

and (user\_id=’value1’ or user\_id=’value2’ or user\_id=’value3’…)

**2、where column in(‘value1’,’value2’,’value3’,…)**

**<foreach collection=”ids” item=”item\_id” open=”and user\_id in(” close=”)” separator=”,”>**

**#{item\_id}**

**</foreach>**

# 4、 MyBatis的高级应用

在本章节中，会讲解mybatis的高级映射、一级缓存、二级缓存以及mybatis与Spring的整合。同时会以具体的订单商品数据模型为例，通过mybatis框架演示在企业中进行软件开发的实际开发过程。首先来分析一下用户订单商品的数据模型。

## 4.1 订单商品数据模型

在订单商品数据模型中，存在以下四张表，用户表user，用户订单表orders，订单明细表orderdetail，商品表items。四张表之间的关系如下图所示：

订单明细：orderdetail

detail\_id：订单明细主键

order\_num：订单号

item\_id：商品外键

订单表：orders

order\_id：订单主键

user\_id：用户外键

一对多

一对一

一对一

一对多

一对多

一对一

商品明细：items

item\_id：商品主键

用户表：user

user\_id：用户主键

下面给出四张表结构的详细定义，出于演示，每张表仅定义少量字段。

用户表：user

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **列名** | **数据类型** | **是否为空** | **主键** | **外键** | **注释** |
| user\_id | varchar2（40） | 否 | 是 |  | 用户主键 |
| user\_name | varchar2（40） | 否 |  |  | 用户名 |
| user\_pass | varchar2（20） | 否 |  |  | 用户密码 |
| cname | varchar2（20） | 否 |  |  | 昵称 |
| user\_img | varchar2（100） | 是 |  |  | 用户头像 |

订单表：orders

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **列名** | **数据类型** | **是否为空** | **主键** | **外键** | **注释** |
| order\_id | varchar2（40） | 否 | 是 |  | 订单主键 |
| user\_id | varchar2（40） | 否 |  | 是 | 用户外键 |
| numer | varchar2（20） | 否 |  |  | 订单号 |
| order\_date | date | 否 |  |  | 订单日期 |
| payment | char（1） | 否 |  |  | 是否付款 |

订单明细表：orderdetail

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **列名** | **数据类型** | **是否为空** | **主键** | **外键** | **注释** |
| detail\_id | varchar2（40） | 否 | 是 |  | 订单明细主键 |
| order\_num | varchar2（40） | 否 |  | 是 | 订单号 |
| item\_id | varchar2（20） | 否 |  | 是 | 商品外键 |
| buy\_num | int | 否 |  |  | 购买数量 |

商品表：items

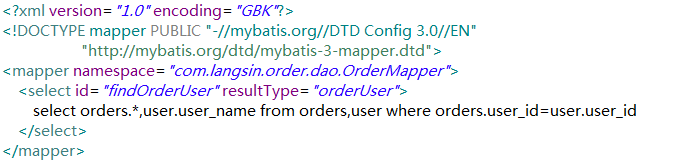
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **列名** | **数据类型** | **是否为空** | **主键** | **外键** | **注释** |
| item\_id | varchar2（40） | 否 | 是 |  | 商品主键 |
| item\_name | varchar2（40） | 否 |  |  | 商品名称 |
| price | double | 否 |  |  | 商品价格 |
| description | varchar2（50） | 否 |  |  | 商品描述 |
| item\_img | varchar2（100） | 是 |  |  | 用户图片 |

## 4.2 一对一关联查询

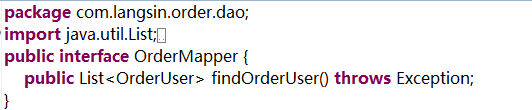
使用mybatis实现一对一关联查询，操作内容为：订单信息查询，并关联查询创建订单的用户的信息。对于返回查询结果信息可以由resultType或resultMap实现。

### 4.2.1 resultType实现

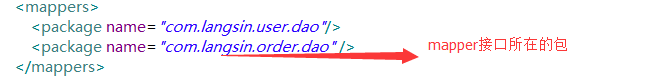
1、编写查询SQL语句如下，暂时不考虑查询条件，查询所有符合条件记录，OrderMapper.xml配置文件如下：



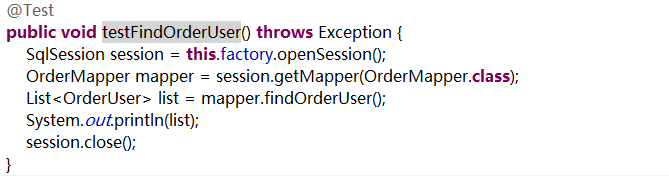
2、定义mapper接口OrderMapper.java，接口与mapper配置文件在同一个包下



3、通过mapper元素加载mapper配置文件，可以通过resource属性、class指定接口方式、package指定mapper接口所在包的形式都可以。下面通过package的形式加载。



4、下面就可以通过JUnit进行测试，测试代码如下：



### 4.2.2 resultMap实现

之前我们已经学习了resultMap的简单使用，对resultMap进行了一定的简单了解。下面来学习resultMap的高级使用方式，以及为何要使用resultMap方式。

例如上例中使用resultType的实现，再进行关联查询时，对于关联表中的字段只取出几列而已，那么在开发时可以选择一个较多的列POJO对象为基础进行扩展，例如A表与B表进行关联查询，A表查询所有列，B表查询几列，那就需要以A表对应的POJO对象为基础进行扩展。即：class QueryABean extends A 。

但是如果在进行关联查询时，几乎两个表中所有的列都被当做查询结果，那么在扩展类中再定义另外一个表中所有的列，显然不是很合适了，这种情况下最常见的方式，就是把另外一个表所对应的POJO对象组合进来，这时使用mybatis进行输出映射就只能使用resultMap元素了，这就是resultMap的高级使用。

1、扩展类OrderUser

package com.langsin.order.vo;

import com.langsin.user.vo.User;

public class OrderUser extends Order {

private User user = null;

public User getUser() {

return user;

}

public void setUser(User user) {

this.user = user;

}

}

2、mapper配置文件

在配置映射时，必须设定唯一标识ID，例如配置订单信息，订单的唯一标识为：order\_id



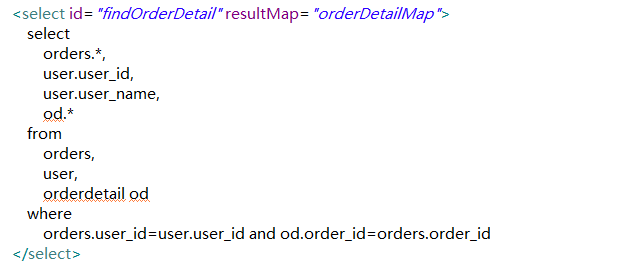
3、测试主程序自己编写并测试

## 4.3 一对多关联查询

使用mybatis实现一对多关联查询，操作内容为：查询订单表并关联查询订单明细表。在没有使用框架时，针对于这种情况，一般我们都是service层进行业务处理，首先组装一个订单扩展类，在订单扩展类中定义一个List集合来存放此订单所对应的订单明细。即：先调用dao层查询订单表，然后再根据订单表与订单明细表的主外键关联字段，再调用订单明细的dao再查询订单明细，最终完成数据组装。

通过mybatis框架collection集合，可以很轻松的完成此项工作。

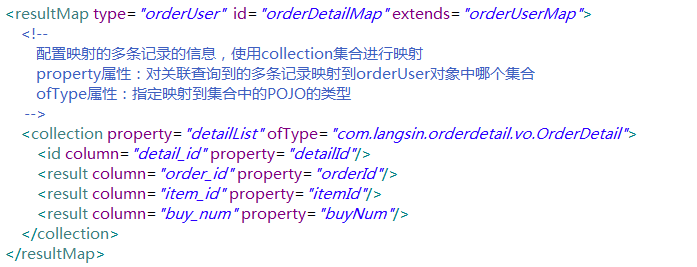
1、mapper配置文件中查询SQL如下：



2、mapper配置文件中resultMap的映射配置



在进行配置映射时，与查询订单明细管理用户信息配置的resultMap映射配置大部分相同所以可以使用extends属性继承一对一查询的映射配置，如下所示：



3、在mapper接口中定义查询方法的声明

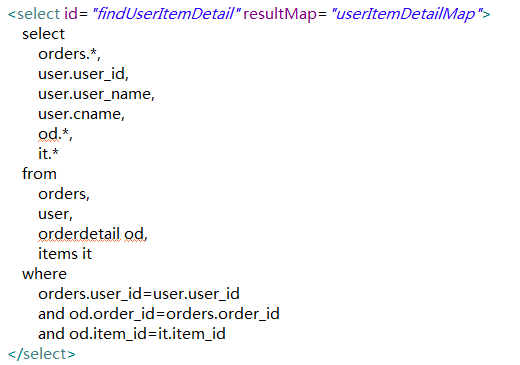
4、定义测试主方法

## 4.4 多对多关联查询

使用mybatis实现多对多关联查询，操作内容为：查询用户及用户购买的商品的信息。

首先进行分析，查询主表为用户表user

通过4.1节订单商品数据模型图可以看出，用户表与商品表没有直接的关联关系，但可以通过中间表：订单表orders、订单明细表orderdetail、商品表item进行关联。SQL如下



下面开始进行配置映射关系，因为是多对多的查询，所以应该如下定义类

1. 将用户信息映射到User对象中
2. 在User对象中定义List<OrderUser> orderList订单列表属性，将用户的订单映射到orderList中
3. Order中添加List<OrderDetail> detailList订单明细列表属性，将订单的明细映射到detailList中
4. 在OrderDetail中添加Item item商品属性，将商品明细映射到Item中

public class User{

// 省略其他用户信息，用户名、地址等

private List<Order> orderList;

}

public class Order{

// 省略其他订单信息，订单号、是否付款等

private List<OrderDetail> detailList;

}

public class Detial{

// 省略其他订单明细信息，购买数量

private Item item;

}

public class Item{

// 定义商品明细，商品名称、价格等

}

mapper.xml配置文件如下：

.

上面映射配置将用户购买的商品的信息全部到User对象中，这种是比较特殊的映射，如果在页面呈现，信息展示效果如下示意图：

用户名：XXX 联系方式：XXXXXXX 地址：XXXXXX

——————————————————————————————————————————————————————

订单号：XXXXX01 是否付款：是/否

订单明细：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 商品名称 | 商品价格 | 描述 | 图片 | 品牌 | 购买数量 |
| 三星笔记本 | 4000 | 一般 |  | 三星 | 1台 |

——————————————————————————————————————————————————————

订单号：XXXXX02 是否付款：是/否

订单明细：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 商品名称 | 商品价格 | 描述 | 图片 | 品牌 | 购买数量 |
| 联想笔记本 | 4000 | 一般 |  | 联想 | 1台 |
| 华硕硬盘 | 800 | 不错 |  | 华硕 | 20个 |

——————————————————————————————————————————————————————

通过上图的展示，配置映射出的User对象的数据结构完全可以很轻易的完成这种数据的展示。

如果仅仅是将用户购买的商品信息明细清单，例如：用户名、地址、购买数量、购买商品时间等、用表格的方式直接展示出来如下所示：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 用户名 | 地址 | 商品名称 | 购买数量 | 购买时间 | 品牌 |
| 张丽 | 山东济南长清XXXX | 海尔洗衣机 | 1台 | 2016-03-12 | 海尔 |

针对上面的需求，就使用resultType将查询到的结果记录映射到一个扩展的POJO中，很简单的实现清单明细的功能，这也是多对多的体现。

## 4.5 resultType与resultMap总结

1、对于单表的查询基本上都用resultType实现即可。

1. 对于一对一的联合查询，如果作为查询结果的第二张的表字段不多，定义一个扩展类，然后使用resultType实现即可。
2. 如果一对一的联合查询，两个表的字段都大量的使用作为查询结果，那么在一个类中，组合另外一个类，使用resultMap进行映射实现。使用association将关联信息映射到一个对象中。
3. 对于一对多，多对多的联合查询，如果查询结果数据特殊使用resultMap，使用collection将关联信息映射到一个集合中去，如果只是针对明细的数据，建议定义扩展类，使用resultType映射实现。

## 4.5 MyBatis的延迟加载

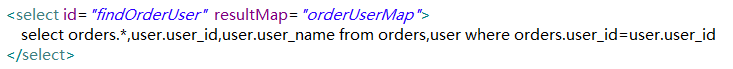
通过前面的技术点学习，可以知道通过resultMap元素可以实现mybatis的高级映射输出，association可以实现一对一高级映射，collection可以实现一对多高级映射，而association和collection还具有延迟加载的功能。

所谓的延迟加载就是先查询主表信息，主表信息查询完毕后，再按照主表相关数据完成关联表信息的查询，这就是延迟加载。即：在SQL查询中，先查询一部分（主表信息），再查询另一部分（关联表信息）。

比如前面操作的查询用户订单数据，并关联查询用户的数据信息，先查询主表用户订单的信息，查询完成后，再按照主表的数据，按需要去查询关联表中的数据。

### 4.5.1 延迟加载的SQL拆分

在实现查询订单表并关联查询用户表的SQL查询中，SQL语句书写形式如下：

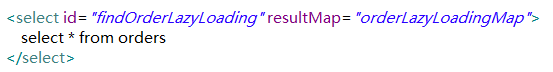


对上面的SQL语句进行拆分，按照延迟加载的本质，即先查询主表，再查询关联表，拆分如下：

1. **select \* from orders**
2. **select \* from user where user\_id=?**

### 4.5.2 在mapper中组装SQL

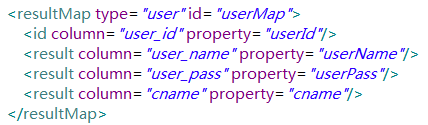
将拆分出是SQL在mapper.xml文件中配置出来，并使用resultMap元素完成组装。如下所示：



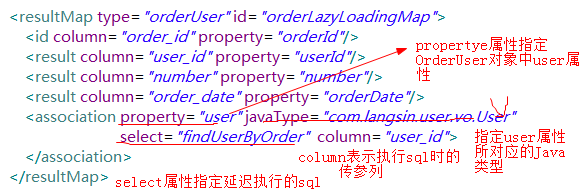
使用resultMap高级映射实现延迟加载，所以上面的select查询的输出映射为resultMap



需要延迟执行的SQL语句，返回类型仍然是个resultMap，之所以使用resultMap是因为表中的列名与Java对象User中的属性名不一致，所以无法直接使用resultType，而需要使用resultMap进行映射。如下所示：



下面使用resultMap元素将两个SQL语句组装起来，如下所示：



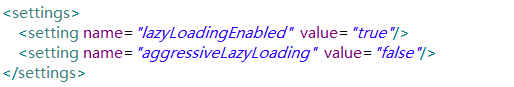
在通过mybatis执行SQL查询时，调用时只会执行mapper接口中的findOrderLazyLoading方法，那么关联的findUserByOrder方法在什么时候执行呢？

在执行findOrderLazyLoading方法时返回的肯定是一个List<OrderUser>的集合，当调用OrderUser对象的getUser方法时，mybatis才会去执行findUserByOrder对应的statement的SQL去查询User对象。

### 4.5.3 配置mybatis的延迟加载

在mybatis中默认是关闭延迟加载的，如果要实现延迟加载，需要在SqlMapConfig.xml全局配置文件中打开延迟加载，使用settings元素进行设置，更改如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **设置项** | **描述** | **允许值** | **默认值** |
| lazyLoadingEnabled | 全局性设置懒加载。如果设为‘false’，则所有相关联的都会被初始化加载。 | true | false | false |
| aggressiveLazyLoading | 当设置为‘true’的时候，懒加载的对象可能被任何懒属性全部加载。否则，每个属性都按需加载。 | true | false | true |



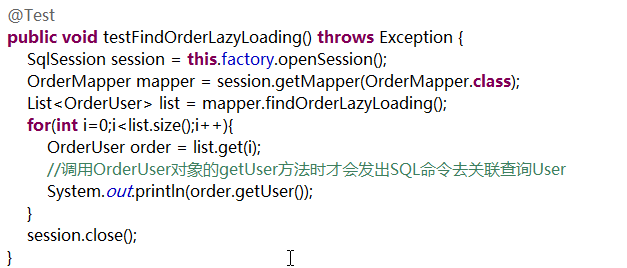
### 4.5.4 mybatis实现延迟

通过上面一系列的配置，其实已经实现了延迟，在下面就是简单的定义mapper接口以及，通过mapper代理完成测试而已。

mapper接口定义如下：



JUnit的测试主程序如下：



### 4.5.5 深入了解延迟加载

对于延迟加载的本质，无非就是将一个复杂的SQL拆分成简单的SQL，进行查询。这个简单的SQL并不一定就是单表查询，也可以是关联查询，只要是个简单查询即可。

主表查询完成后，如果需要再去查询关联表的相关数据。那么也就是说即使不使用mybatis提供的延迟加载功能，也可以实现自己的延迟加载。如上例所示，实现方法如下：

1、定义两个mapper接口

* 查询订单表的方法findOrder
* 查询用户表的的方法findUserByOrder

2、在service层即业务层按需要去查询关联的用户表即可。

延迟加载的本质：先从单表查询、需要时再从关联表去关联查询，大大提高 数据库性能，因为查询单表要比关联查询多张表性能更加优越。

关于延迟加载的使用规则如下：

* 当只有部分记录需要关联查询其它信息时，此时可按需延迟加载，需要关联查询时再向数据库发出sql，以提高数据库性能。
* 当全部需要关联查询信息时，此时不用延迟加载，直接将关联查询信息全部返回即可，可使用resultType或resultMap完成映射。

## 4.6 MyBatis的查询缓存

mybatis提供了查询缓存功能，用于减轻数据压力，提高数据库性能。mybatis提供了一级缓存和二级缓存，如下图所示：

sqlSession

sqlSession

sqlSession

一级缓存

一级缓存

一级缓存

Mappe（namespace）二级缓存

从上图可以看出：

一级缓存是存在于一个sqlSession中的，而sqlSession就是操作数据库的一个会话对象，在对象中实际存储了一个hashMap的数据结构用于存储缓存数据。不同的sqlSession之间的缓存数据互不影响。

二级缓存是mapper级别的缓存，当多个sqlSession操作同一个mapper配置中的SQL语句时，多个sqlSession可以共用二级缓存，二级缓存是跨sqlSession的。

### 4.6.1 一级缓存

mybatis一级缓存的作用域是同一个sqlSession，在同一个sqlSession中两次执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。当一个sqlSession结束后该sqlSession中的一级缓存也就不存在了。mybatis默认开启一级缓存。

如下图所示：

SqlSession会话

写入

清除

读取

SqlSession

一级缓存区域

hashMap

第一次查询id为1的用户

写入

修改、添加、删除用户操作执行commit

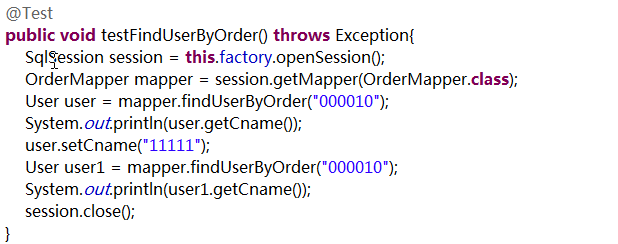
第二次查询id为1的用户

* 第一次执行查询id为1的用户的信息操作时，先在缓存中查找，如果没有，则执行数据库查询操作，并将从数据库中查询到的用户数据信息，并放入到一级缓存。

mybatis内部存储缓存使用一个HashMap，key为hashCode+sqlId+Sql语句。value为从查询出来映射生成的java对象

* 如果sqlSession执行insert、update、delete等操作commit提交后会清空一级缓存，这样做的目的是为了让缓存中存储的数据是最新的信息，避免脏读。
* 第二次执行查询id为1的用户的信息操作时，先去缓存中查找，如果缓存中有，则直接从缓存中获取用户信息。

测试代码如下：



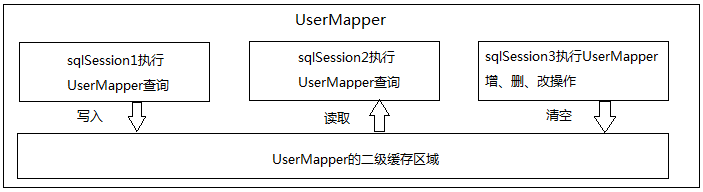
注意：在正式开发中，是将mybatis和Spring进行整合开发，事务是在service层进行控制，一个service方法中包含很多mapper方法的调用。

在service方法开始执行时，就会开启事务，在开启事务的过程中就会创建SqlSession对象，因此期间不管是调用哪个mapper的方法，都是走一级缓存，有就从缓存中取，没有就从数据库中查询。方法结束时，事务关闭，同时清空一级缓存。

因此在同一service方法中是走一级缓存，如果是两次调用service方法，不走一级缓存，因为方法结束，SqlSession也就关闭，一级缓存就会被清空。

### 4.6.2 二级缓存

在mybatis中允许多个SqlSession对象共享一个缓存区域，只不过这个缓存区域并一定在内存中，也可能是存储硬盘空间内，这个共享区域就是mybatis的二级缓存。mybatis同样适用hashMap这种数据结构来存储二级缓存中保存的数据。 如下图所示：



从上图中可以看出，mybatis的二级缓存是根据mapper配置文件的namespace命名空间进行划分的，相同namespace的查询数据操作放在同一个缓存区中。即用户的查询操作的数据是放在UserMapper的缓存区中，订单查询操作的数据是放在OrderMapper的缓存区中。

如果两个用户的SqlSession会话都是执行同一个UserMapper接口中的方法，并且都去查询用户数据，每次查询都会先从缓存区中查找，如果找不到从数据库中查询，查询到的数据写入到二级缓存。

任何一个用户的SqlSession 执行insert、update、delete等操作commit提交后都会清空该mapper下的二级缓存区域。

在mybatis中二级缓存是默认关闭的，如果要开启mybatis的二级缓存，配置如下：

**1、在SqlMapConfig.xml全局配置文件中加入setting信息**



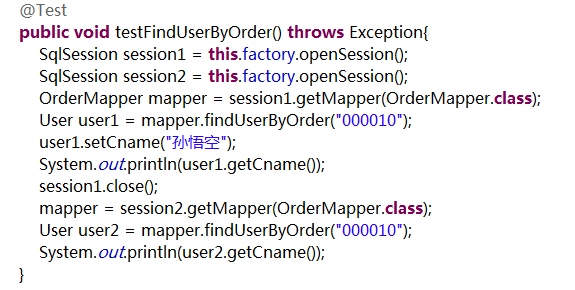
**2、在需要开启的二级缓存的mapper.xml中加入cache标签**

只要加入cache标签就表示开启二级缓存

二级缓存需要查询结果映射的pojo对象实现java.io.Serializable接口实现序列化和反序列化操作，注意如果存在父类、成员pojo都需要实现序列化接口。因为mybatis实现的二级缓存，数据并不一定存储在内存中，也有可能是其他位置，比如硬盘、网络空间等。

在SqlSession会话中，如果会话没有结束，数据只会存储于一级缓存中，如果此SqlSession的会话结束并且此命名空间的mapper开启了二级缓存，这时数据才会写入到二级缓存中。

**测试代码如下：**



## 4.7 MyBatis整合Ehcache

EhCache 是一个纯Java的进程内缓存框架，是一种广泛使用的开源Java分布式缓存，具有快速、精干等特点，是Hibernate中默认的缓存提供者。不使用分布缓存，缓存的数据在各个服务器上单独存储，不方便系统开发。所以要使用分布式缓存对缓存数据进行集中管理。分布式缓存示意图如下：

客户端

系统工程

服务器1

系统工程

服务器1

系统工程

服务器1

对缓存数据进行集中管理

使用分布式缓存框架redis、ehcache、memcached

### 4.7.1 mybatis整合ehcache的原理

mybatis提供二级缓存Cache接口，如下所示：



mybatis针对此接口提供了一个默认的实现类，如下所示：



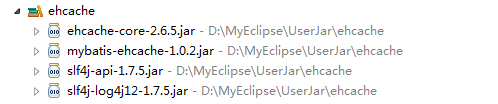
虽然mybatis也实现提供了二级缓存，但是mybatis的特长是SQL操作，而不是缓存管理，比如实现分布式缓存，mybatis就不支持此功能。为了提高缓存的性能将mybatis与第三方的缓存数据库框架整合即可，如ehcache、redis、memcache等。

通过上图可以看出只要实现了mybatis的cache接口就可以实现mybatis的二级缓存数据。

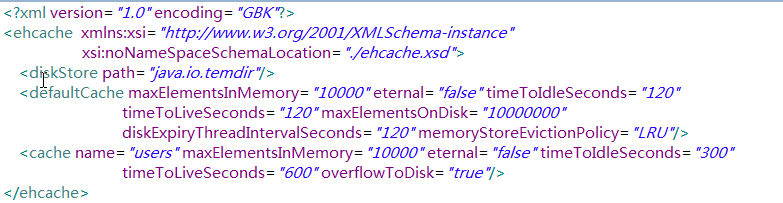
### 4.7.2 mybatis整合ehcache三部曲

根据实现mybatis的二级缓存的原理，整合ehcache只需要完成以下三步即可：

**1、引入缓存的依赖包**



**2、在classpath下添加ehcache.xml配置文件**



**3、开启ehcache缓存**

在mapper.xml中开启二级缓存是通过添加cache元素，cache元素提供了一个type属性，此属性指定一个实现了mybatis的Cache接口的实现类。对于ehcache缓存框架，它提供了一个mybatis的Cache接口的实现类，只需要指向此类即可。



同样也可根据需求调整缓存参数：

<cache type="org.mybatis.caches.ehcache.EhcacheCache" >

<property name="timeToIdleSeconds" value="3600"/>

<property name="timeToLiveSeconds" value="3600"/>

<!-- 同ehcache参数maxElementsInMemory -->

<property name="maxEntriesLocalHeap" value="1000"/>

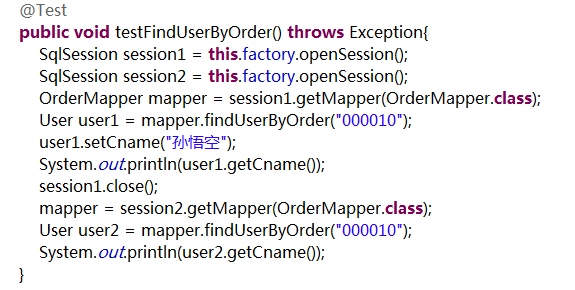
<!-- 同ehcache参数maxElementsOnDisk -->

<property name="maxEntriesLocalDisk" value="10000000"/>

<property name="memoryStoreEvictionPolicy" value="LRU"/>

</cache>

测试主程序如下：



## 4.8 二级缓存的应用场景及局限性

**1、应用场景**

对于访问次数较多的查询请求并且用户对查询的实时性要求不高，此时可以采用mybatis二级缓存技术降低数据库的访问量，提高访问速度。比如：耗时较高的统计分析SQL，电话账单查询SQL等。

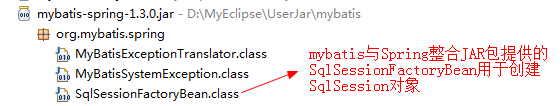
实现方法如下：通过设置刷新间隔时间，由mybatis每隔一段时间自动清空缓存，根据数据变化频率设置缓存刷新间隔flushInterval，比如设置为30分钟、60分钟等，根据需求设置。

**2、局限性**

mybatis二级缓存对细粒度的数据级别的缓存实现不好，比如如下需求：对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次都能查询最新的商品信息，此时如果使用mybatis的二级缓存就无法实现当一个商品变化时只刷新该商品的缓存信息而不刷新其它商品的信息，因为mybaits的二级缓存区域以mapper为单位划分，当一个商品信息变化会将所有商品信息的缓存数据全部清空。解决此类问题需要在业务层根据需求对数据有针对性缓存。

## 4.9 Spring整合MyBatis

在mybatis框架的学习中，最重要的两个类就是SqlSessionFactory和SqlSession会话对象，SqlSession由工厂SqlSessionFactory工厂创建，因此整合Spring与MyBatis进行整合时，就需要由Spring容器以单例模式来管理SqlSessionFactory。但是在mybatis框架的学习中，SqlSessionFactory是由SqlSessionFactoryBuilder工厂创建对象来建立的，那么在进行整合时SqlSessionFactory是由Spring和MyBatis的整合JAR包来提供。如下所示：



在Spring的配置文件中配置该Bean，该Bean有两个重要field，一个是configLocation用于指定mybatis的全局配置文件SqlMapConfig.xml，一个是dataSource属性，用于指定一个数据库连接池，所以在spring配置文件中配置一个数据库连接池，以C3P0为例，如下所示：



### 4.9.1 整合下原始dao开发

**1、以原始dao的开发方式配置mybatis**

在上面的配置文件中配置了一个userDao的DAO持久层类，此类需要继承SqlSessionDaoSupport类，该类提供了一个setSqlSessionFactory的方法，可以把配置的SqlSessionFactory Bean对象作为参数赋值进去，通过SqlSessionDaoSupport类提供的getSqlSession方法来获取一个SqlSession会话。如下所示：

示例代码1：

package com.langsin.user.dao;

import java.util.List;

import org.apache.ibatis.session.SqlSession;

import org.mybatis.spring.support.SqlSessionDaoSupport;

import com.langsin.user.vo.User;

public class UserDaoImpl extends SqlSessionDaoSupport implements UserDao {

@Override

public User getUserById(String id) throws Exception {

SqlSession session = this.getSqlSession();

User user = session.selectOne("users.findUserById", id);

return user;

}

}

**2、创建JUnit测试类**

package com.springmybatis.test;

import org.junit.Before;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.langsin.user.dao.UserDao;

import com.langsin.user.vo.User;

public class UserDaoImplTest {

private ApplicationContext context = null;

@Before

public void setUp() throws Exception {

this.context = new ClassPathXmlApplicationContext("classpath:applicationContext.xml");

}

@Test

public void testUserDaoImpl() throws Exception {

UserDao dao = this.context.getBean("userDao", UserDao.class);

User user = dao.getUserById("000010");

System.out.println(user);

}

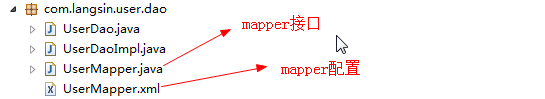
}

### 4.9.2 整合下mapper代理开发

使用Spring容器管理mapper的动态代理对象仍然需要遵循mybatis创建mapper动态代理对象的规范，即：1、 mapper的配置文件与mapper接口名保持一致，mapper配置文件的命名空间是执行mapper接口的全限定名；2、mapper配置文件的SQL的statement的id要与mapper接口的方法名保持一致；3、输入参数的类型要与mapper接口的方法名中参数类型保持一致；4、发回的数据类型要与mapper接口的方法的返回值类型一致。

如下所示：

**1、创建mapper.java与mapper.xml**

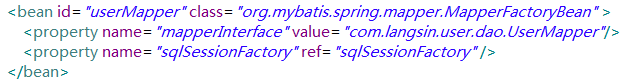


**2、在Spring中配置mapper的代理对象**

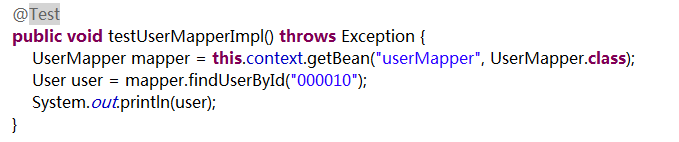
在Spring的容器中配置的都是具体的实现类，而mapper接口是没有实现类的，这时需要mybatis和spring的整合JAR包下的MapperFactoryBean来帮助生成Mapper的动态代理对象。如下所示：



配置MapperFactoryBean类时需要注入两个重要的属性，mapperInterface属性：指定mapper接口并为之生成mapper动态代理对象；sqlSessionFactory属性：指定创建SqlSession会话对象的SqlSessionFactory类。如下所示：



**3、创建JUnit测试主程序**

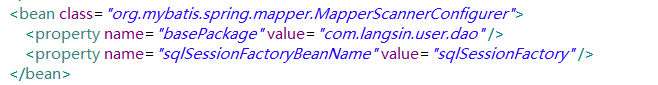


### 4.9.3 整合下批量扫描mapper接口

通过Spring配置mapper的动态代理，唯一的缺点就是需要对每个mapper进行配置，一旦项目工程越来越大，在mybatis的配置文件中mapper的配置就会变的非常庞大，甚至是繁琐，这不是Spring所期望的，下面可以借助于mybatis和spring的整合JAR包中的MapperScannerConfigurer类进行批量扫描mapper接口，如下所示：

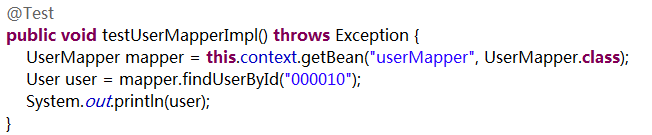


配置MapperScannerConfigurer类时需要注入两个重要的属性，basePackage属性：指定要扫描的包路径下的Mapper接口；sqlSessionFactoryBeanName属性：在MapperScannerConfigurer类中是String类型的成员变量，赋值为Spring配置文件中的SqlSessionFactory Bean配置的id属性值，用此id属性值来寻找SqlSessionFactory对象。如下所示：



如果需要扫描多个package，需要在value属性中使用英文逗号隔开。

创建JUnit测试程序：



## 4.10 逆向工程

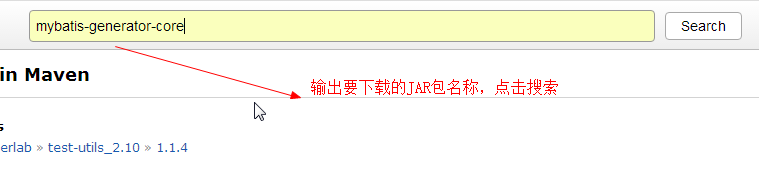
在使用mybatis框架进行数据库持久层开发时，需要程序开发人员手动编写SQL语句、Mapper.xml配置文件，Mapper接口，POJO类，针对于数据库中的单表的操作，mybatis提供了mybatis-generator-core-bundle这样一个JAR包可以帮助程序开发人员自动生成上述所需要手动编写的程序或配置文件，这就称为逆向工程。

一般情况下，软件开发都是功能设计、类图、生成数据库表这样一个流程完成，而mybatis是通过数据库中表来生成类、配置文件、接口等，是反向的操作，所以称为逆向工程。

### 4.10.1 下载逆向工程

**1、下载逆向工程**

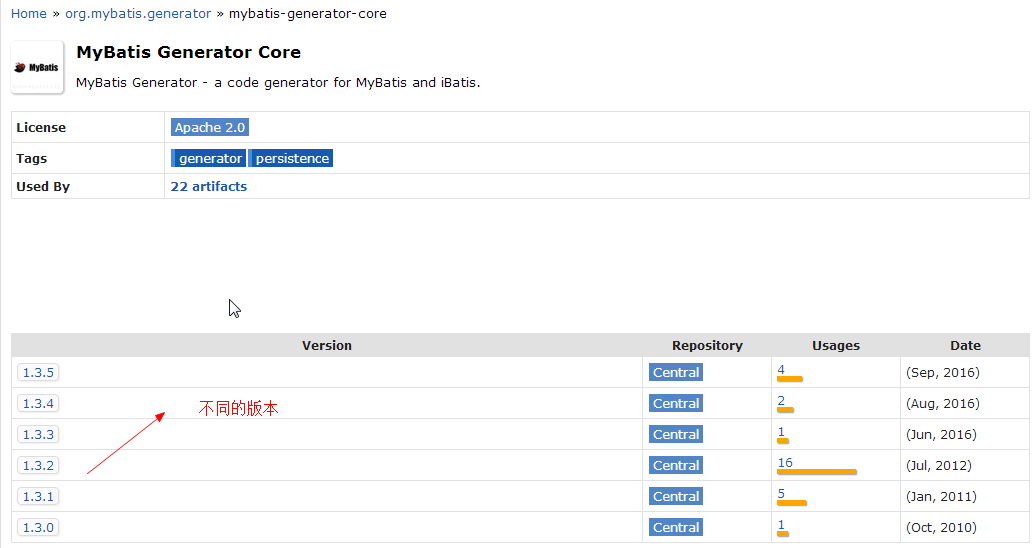
所谓的逆向工程实际就是一个工具JAR包，可利用JAR自动生成相应的代码文件，可以从MVN依赖工厂地址下载所需要的JAR包，地址为：<http://mvnrepository.com/> ，此地址可以下载绝大部分进行JAVA开发所需要的第三方提供的各种不同版本的开发JAR包。如下图所示：



点击查询后，在查询结果列表页面，单击要下载的JAR包，如下所示：



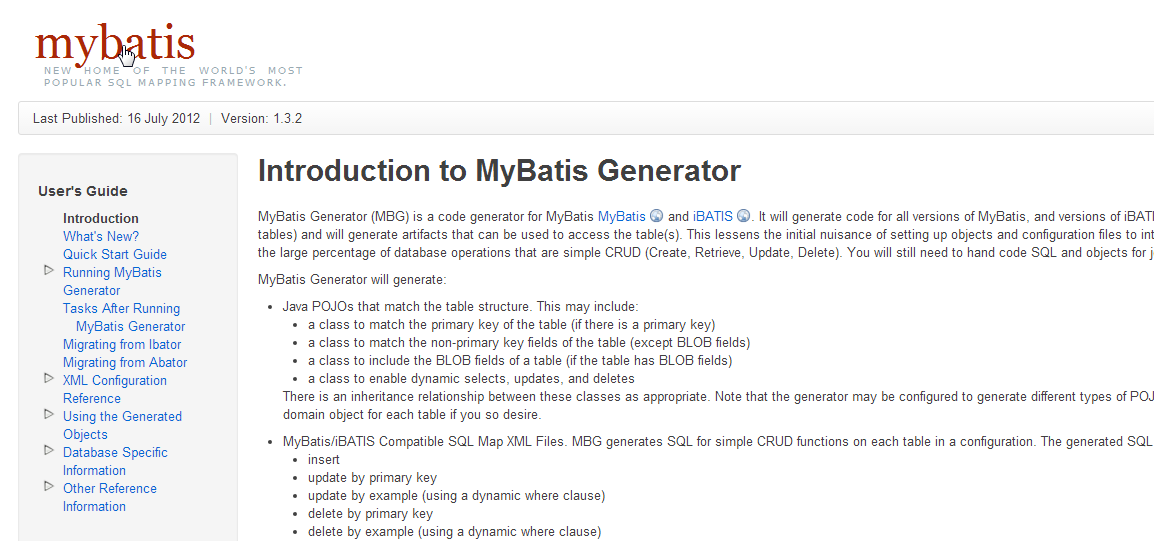
单击连接，进入下载页面，此页面包含以往的各个不同版本，选择需要的版本下载，如下所示：



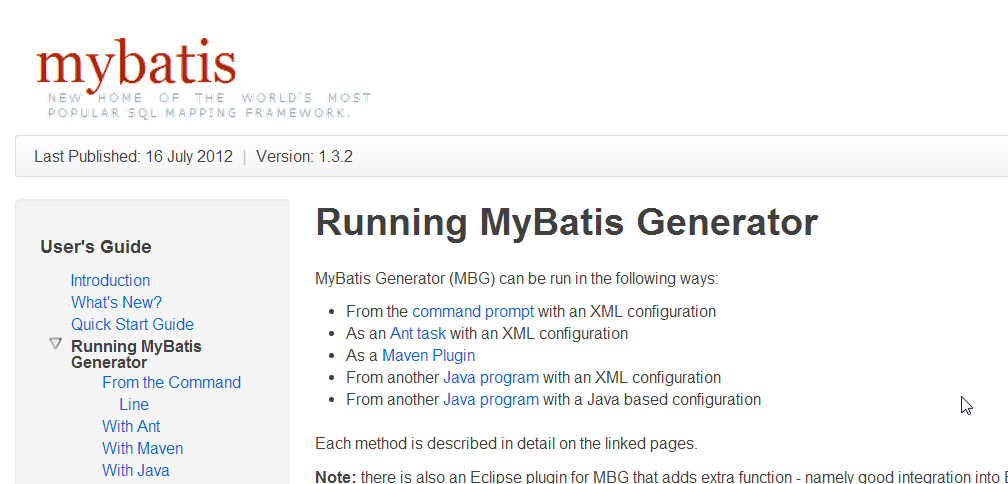
下载完成后解压即可。

### 4.10.2 使用逆向工程

从上述地址下载的只是运行所依赖的JAR包，如果需要下载原工程以及使用说明文档需要去官网下载，目前mybatis项目时托管在GitHub下，地址为：<https://github.com/mybatis/generator/releases>，最新版本为：1.3.5。下载ZIP文件，下载完成后解压里面含有一个doc文件，为说明文档，打开index.html如下：



单击左侧链接：Running MyBatis Generator命令，运行mybatis生成工具的方式，进入如下界面

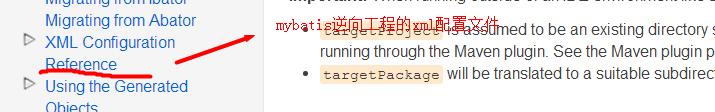


在此页面，程序开发人员可以使用如下几种方式来运行mybatis的逆向工具：

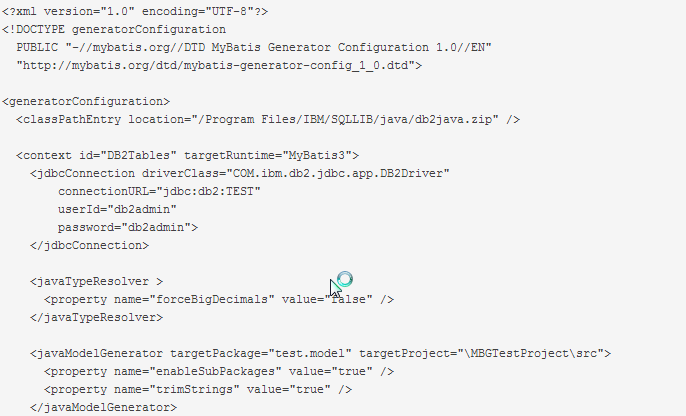
1. 使用命令行输入命令，基于XML配置运行
2. 使用Ant工具，基于XML配置运行
3. 使用Maven工具
4. 使用Java项目的方式基于XML配置运行（重点）
5. 仅仅使用Java项目以及最基础的配置运行

**第4种方式是推荐方式，因为可以不受开发工具的限制，不受插件的限制。只要可以运行Java项目的工具就可以使用该工具来运行mybatis的逆向工具来生成代码。**

首先建立一个普通的Java项目，最重要的还是MyBatis Generator的配置文件，如下所示：



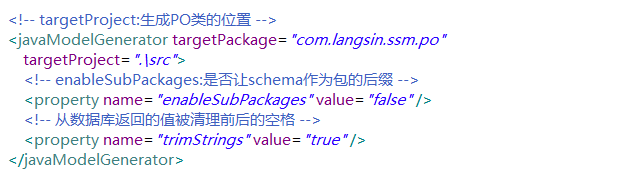
单击链接，XML配置模板如下：



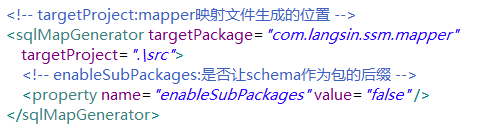
对上述配置文件分析如下：



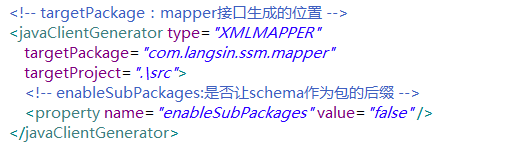
jdbcConnection元素：配置数据库连接



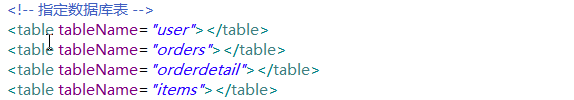
javaModelGenerator元素：生成PO类，并指定PO类的包



sqlMapGenerator元素：生成Mapper映射文件

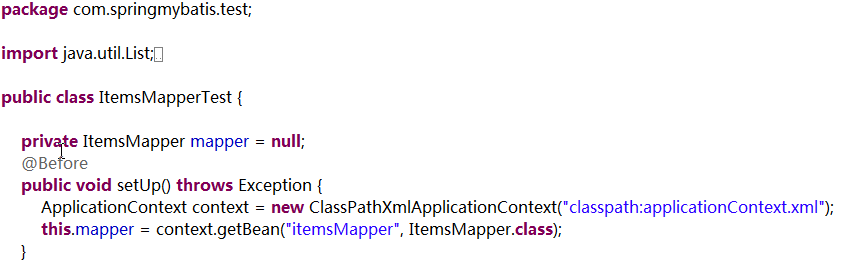


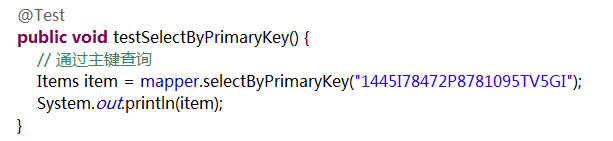
javaClientGenerator元素：生成Mapper接口

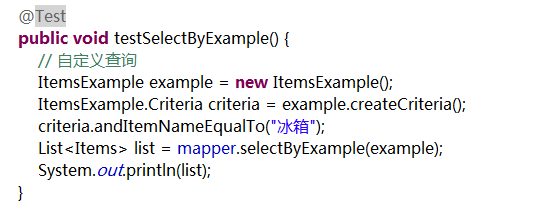


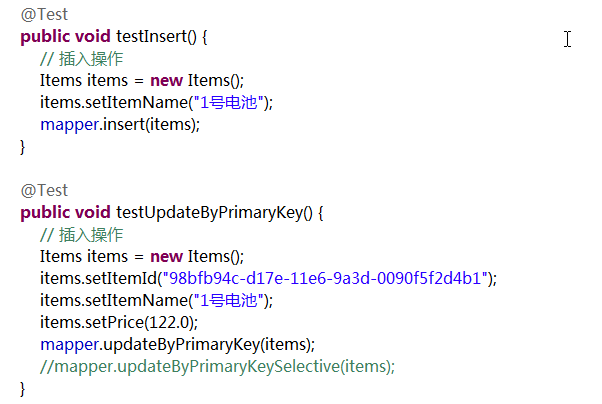
table元素：指定对数据库中哪些表完成代码生成。

**测试生成代码，将生成的代码复制到项目中来，生成JUnit测试类**









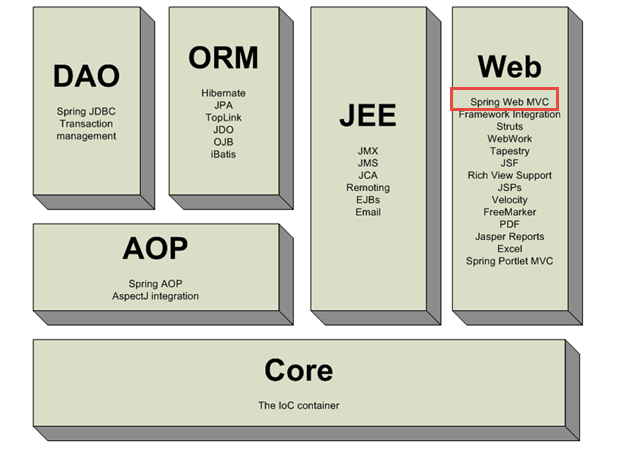
在上述update方法中，updateByPrimaryKey与updateByPrimaryKeySelective方法的区别如下：

* updateByPrimaryKey：根据主键更新，先查询后更新，如果对象中包含属性为null，字段也会被更新为null。
* updateByPrimaryKeySelective：根据主键更新，不需要提前查询，只做更新操作，如果对象中包含属性为null，则不更新此字段，适合批量更新操作。

# SpringMVC基本用法

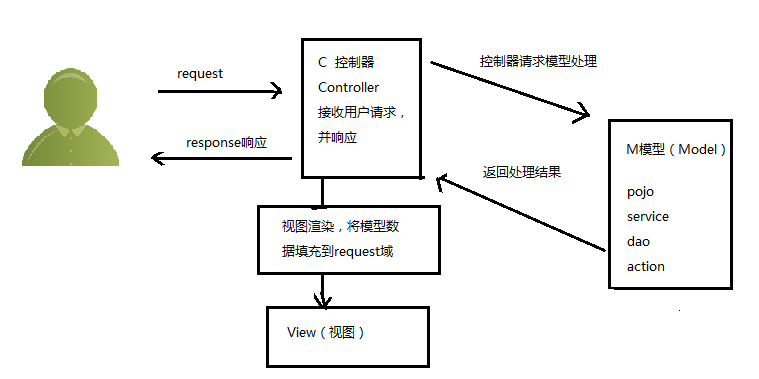
SpringMVC是Spring框架的一个模块，SpringMVC和Spring无需通过中间整合层进行整合。SpringMVC是一个基于MVC的web框架，即Spring Web MVC。

Spring Web MVC和Struts2都属于表现层的框架，它是Spring框架的一部分，我们可以从Spring的整体结构中看得出来：



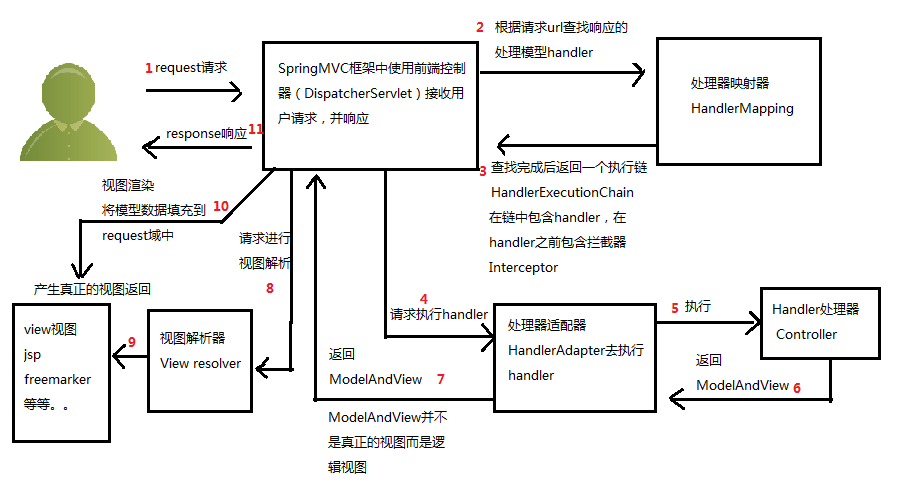
## 5.1 Web MVC原理

首先来一下MVC，MVC是一种设计模式，在大量的软件开发过程中所积累出来的经验，然后根据经验抽象出来的一种开发模式。下面来看一下MVC设计模式B/S架构的系统下的应用：



## 5.2 SpringMVC框架原理

从上图中看到MVC的基本框架后，那么下面学习SpringMVC框架就有了一个大体的思路。因为SpringMVC就是MVC模型的一种框架。所以学习SpringMVC框架主要就是来看SpringMVC中的C（控制层）、M（模型层）、V（视图层）的各种技术实现即可。如下图所示：



根据上图所示，使用SpringMVC的步骤如下：

1. 发起请求到前端控制器DsipatcherServlet
2. 前端控制前DispatcherServlet请求HandlerMapping查找Handler，（XML配置或注解）
3. 处理器映射器HandlerMapping向前端控制器返回handler
4. 前端控制器调用处理器适配器去执行handler
5. 处理器适配器执行handler
6. handler执行完成后返回给处理器适配器ModelAndView（SpringMVC框架的一个底层对象，包括model、view）
7. 处理器适配器向前端控制器返回ModelAndView
8. 前端控制器请求视图解析器去进行视图解析，根据逻辑视图名解析成真正的视图视图（jsp）
9. 视图解析器向前端控制器返回view

10、前端控制器进行视图渲染，将模型数据（在ModelAndView）填充到request域中。

11、前端控制器向用户响应结果。

以上步骤涉及到的组件：

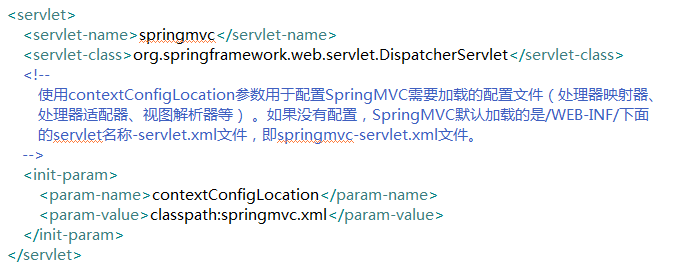
1. 前端控制器DispatcherServlet，作用：接收请求，响应结果，相当于转发**（不需要程序人员开发）**
2. 处理器映射器HandlerMapping，作用：根据请求url查找handler**（不需要程序人员开发）**
3. 适配器处理器HandlerAdapter，作用：按照特定的规则去执行handler，所谓的特定规则，即HandlerAdapter指定的规则。**（不需要程序人员开发）**
4. Handler处理器。**（需要程序人员开发）**
5. 视图解析器Viewresolver，作用：进行视图解析，根据逻辑视图名解析成真正的视图view**（不需要程序人员开发）**
6. 视图View，作用：View是一个接口，实现类支持不同的View类型（jsp、freemarker等）**（需要程序人员开发，View不需要但是页面需要开发）**

## 5.3 SpringMVC入门程序

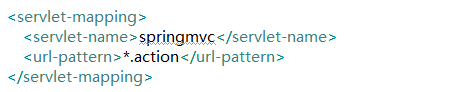
以商品订单管理为例进行测试开发，涉及表用户表、订单表、订单明细、商品明细表。

### 第一步：配置SpringMVC的前端控制器

SpringMVC的前端控制器是个Servlet，所以在web应用的web.xml文件中来配置该前端控制器，如下图所示：



servlet配置完成后，需要对该servlet类来配置相应的servlet-mapping如下图所示：



url-pattern的配置有两种方式：

第一种：\*.action；访问以.action结尾的任何请求都由DispatcherServlet进行解析

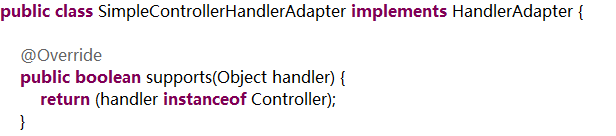
第二种：/；所有访问的地址全部由DispatcherServlet进行解析，对于静态文件的解析需要配置不让DispatcherServlet进行解析，使用此种方式可以实现RESTful风格的url。

### 第二步：配置处理器适配器

在classpath下的springmvc.xml文件中配置处理器映射器。springmvc框架是Spring的一个内部模块，所以springmvc.xml文件肯定需要同spring的配置文件兼容，即保持相同的头信息。如下所示：

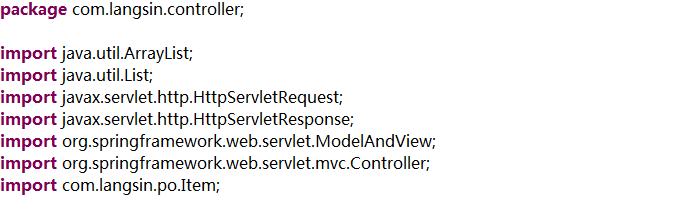


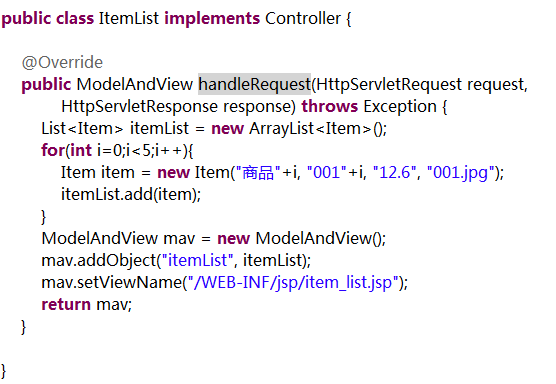
配置完成适配器，配置器的作用就是来执行相应的Handler处理器，那么怎么编写代码才算是一个Handler处理器，通过SimpleControllerHandlerAdapter的原码查看，此类中包含如下一个方法supports，如下所示：



通过此方法可以得知，只有实现了Controller接口的类，才能够被Handler适配器当成是Handler处理器来进行调用。所以进入下一步，编写Handler。

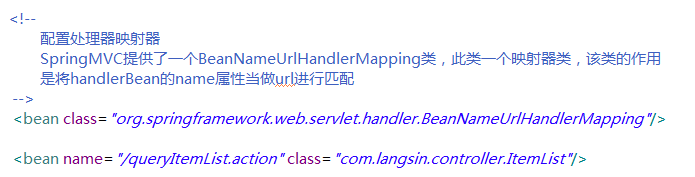
### 第三步：编写Handler





### 第四步：配置处理器映射器

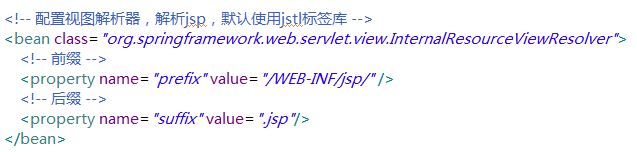
在springmvc.xml文件中配置处理器映射器，如下所示：



上图中的queryItemList.action就是应用中访问的url路径。

### 第五步：配置视图解析器

在springmvc.xml文件中配置是解析器，如下所示：



prefix属性：执行访问路径中固定的前缀路径。

suffix属性：执行访问路径中固定的后缀路径。

使用此种方式：第三步中的代码可修改如下：

**XXX.setViewName(“item\_list”);**

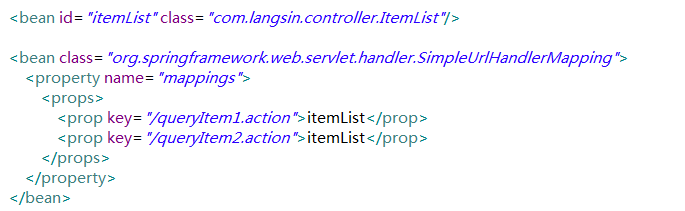
## 5.4 非注解的处理器映射器、适配器

**处理器映射器：**

通过上面的入门程序中，对于处理器映射器BeanNameUrlHandlerMapping就是一个非注解的处理器映射器，只不过这个映射器稍微有点缺陷。缺陷就是不能对bean的url映射进行集中管理。

SpringMVC还提供了另外一个非注解的处理器映射器SimpleUrlHandlerMapping，这个映射器可以实现对bean

的url的集中管理，首先配置一个handler Bean通过bean ID与property元素中的prop子元素关联，key属性配置的是访问的业务的url，prop元素内容为bean ID，表示当访问到/queryItem1.action业务时，SpringMVC通过映射器查询到itemListhand，然后将它交由Handler适配器去执行。如下所示：



注意：对于SpringMVC来说，可以让多个处理器映射器共存，由前端控制器判断用户请求的url能让哪些映射器来处理，就让哪个正确的的映射器处理。

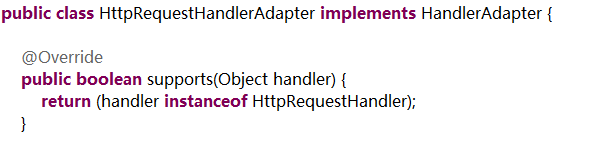
**适配器：**

在上面的入门程序中，org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter就是一个非注解的适配器。此适配器要求所有的Handler处理器必须实现Controller接口。使用方式参考入门程序。

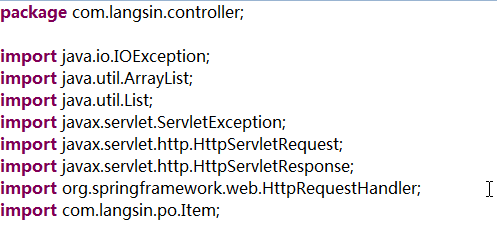
org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter是SpringMVC提供的另外一种非注解的处理器适配器，配置如下图所示：



查看该类的原码如下：



此适配器要求Handler处理器必须实现HttpRequestHandler接口，测试类如下所示：



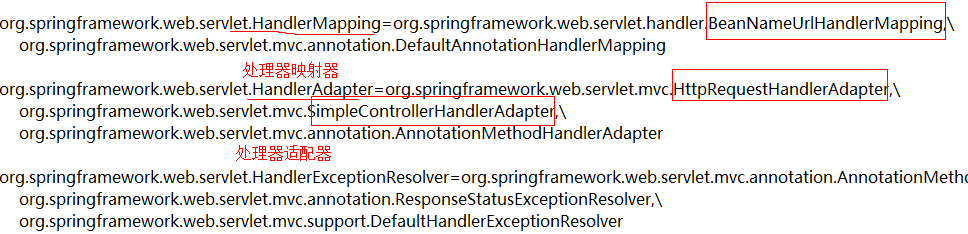


通过上面测试类的编写，大家会发现这种编程非常类似JSP、Servlet中Servlet的代码编写。大家可能会认为这种方式不如上面实现Controller接口的handler更加直观。但是这种方式却比较灵活，因为实现Controller接口的handler返回值就是一个ModelAndView对象，如果需要服务器端响应后返回json格式的数据，返回image对象数据，它就不能实现。而使用第二种方式，可以通过修改response对象，设置响应格式，所以在开发时可根据需要实现不同类型的handler。

**总结：在SpringMVC中多个处理器映射器、适配器是可以共存的。通过用户访问的url路径映射到一个具体的handler对象后，SpringMVC来确定此handler的类型来决定此Handler应该交由哪种适配器去调用执行。**

**注意：如果在springmvc.xml文件即使不配置处理器映射器、处理器适配器，程序仍然可以正常执行，原因如下：在前端控制器DispatcherServlet所在的包org.springframework.web.servlet下包含一个DispatcherServlet.properites属性文件，在该文件中配置了大量的处理器映射器、处理器适配器、视图解析器等组件，如果在springmvc.xml配置文件中没有配置映射器、适配器，那么SpringMVC就使用该属性文件中默认配置的组件。**

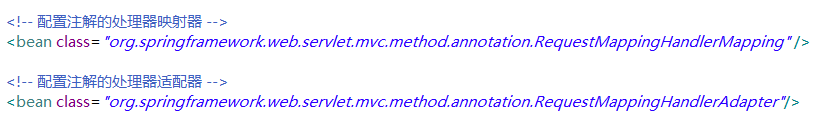
**如下图所示：（截取属性文件部分内容 ）**



在上面的配置文件中同样存在使用注解的配置，AnnotationMethodHandlerAdapter、DefaultAnnotationhadnerMapping，只不过这种注解是比较原始的使用方式，在SpringMVC3.1之前使用，在SpringMVC3.1之后使用另一种注解处理器、适配器。

## 5.5 注解处理器映射器、适配器

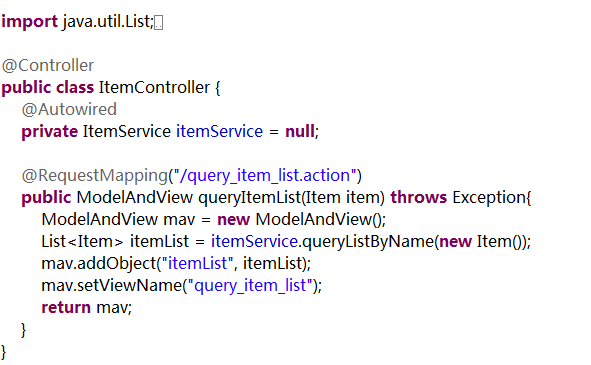
### 5.5.1 配置注解的映射器、适配器



在3.1之后的注解的处理器映射器、适配器添加了很多的优化以及参数设置，所以在开发时，推荐使用3.1之后的版本。

在SpringMVC中提供了一个mvc:annotation-driven元素，用于简化映射器、适配器在springmvc.xml文件中的配置。不仅可以替代上面两行的配置，而且mvc:annotation-driven的驱动类还默认加载了多个参数绑定的方法，比如json转换解析器就默认加载。所以开发时推荐使用该方式。

### 5.5.2 开发注解的处理器Handler



加载该Handler可以使用Spring的批量加载扫描的方式，如下所示：



小结：

处理器映射器：

非注解的处理器映射器（了解）

注解的处理器映射器（掌握）

对标记@Controller注解的类，Spring容器自动加载该Bean并管理，对于该Bean中的使用@RequestMapping注解进行修饰的方法进行映射。[在@RequestMapping里边定义映射的url路径。使用注解的映射器不需要在xml中配置url和Handler](mailto:在@RequestMapping里边定义映射的url路径。使用注解的映射器不需要在xml中配置url和Handler)之间的映射关系。

处理器适配器：

非注解的处理器适配（了解）

注解的处理器适配器（掌握）

注解的适配器与注解的映射器是配对使用，也就是说注解的适配器不能使用非注解映射进行映射。

## 5.6 Spring+SpringMVC+Mybatis整合开发

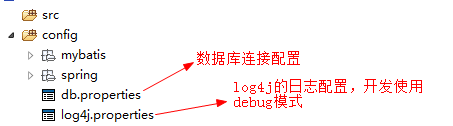
通过SpringMVC入门程序的学习，以及对非注解、注解处理器映射器、处理器适配器的了解，大体知道了SpringMVC的基本的开发流程，下面就以商品查询功能模块为示例，来学习Spring框架、SpringMVC框架、MyBatis框架的整合开发。

### 1、引入JAR包

首先将应用所需要的JAR包引入到项目中，包括log4j、mysql、dbcp、commons-io、spring、mybatis等Jar文件。

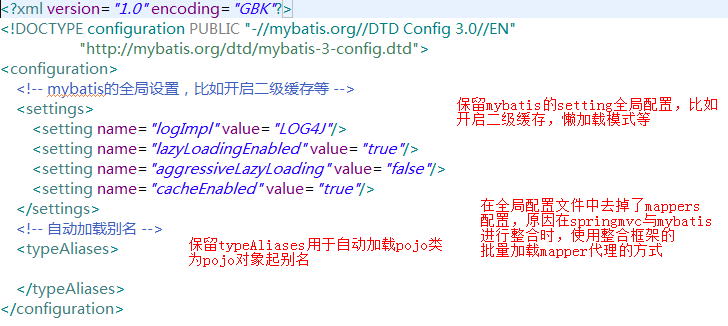
### 2、配置资源文件

配置相关的资源文件，比如log4j、数据库链接配置等，如下图所示：



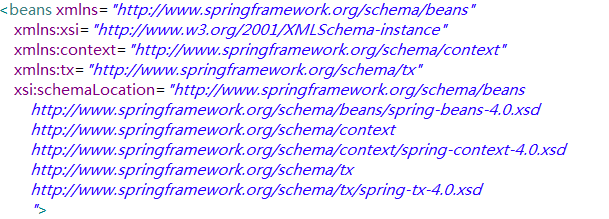
### 3、mybatis的sqlMapConfig.xml

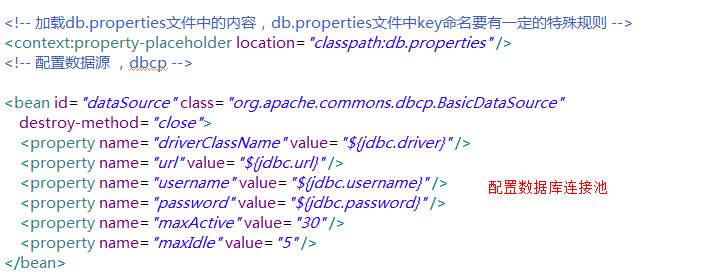
编写mybatis框架的配置文件sqlMapConfig.xml，在使用Spring框架进行整合开发时会加入不同框架的配置文件，为了能够方便管理，所以在开发时需要遵循一定的规范，即不同框架的配置文件，放在同的文件夹下单独管理。例如，mybatis的配置文件存储路径为：config/mybatis/sqlMapConfig.xml，如下图所示：

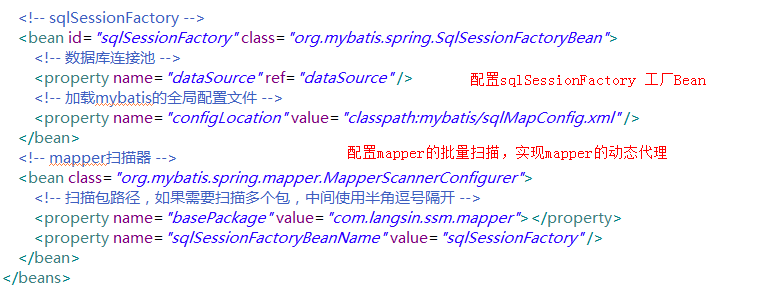


### 4、Spring与MyBatis整合

将Spring与mybatis进行整合，配置文件为applicationContext-dao.xml，如下图所示：

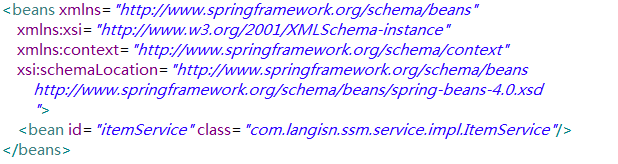






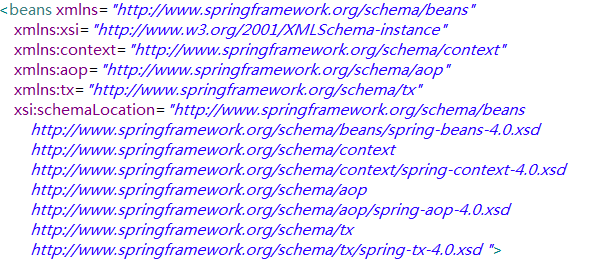
### 5、配置应用的Service层

使用Spring框架的配置文件，applicationContext-service.xml文件来配置管理应用的所有Service层。

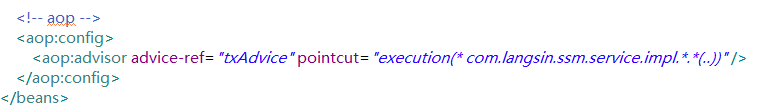


### 6、配置事务管理

事务管理器：对mybatis数据库操作进行事务控制，spring使用jdbc的事务控制类实行事务管理，如下所示：

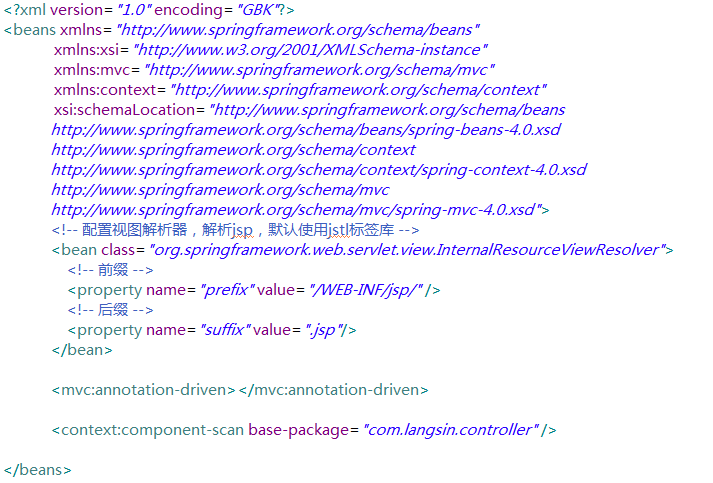






### 7、配置SpringMVC控制层

SpringMVC的配置文件，主要配置视图解析器，处理器映射器、处理器适配器，在配置文件中使用mvc:annotation-dirven驱动来加载映射器与适配器。如下图所示：



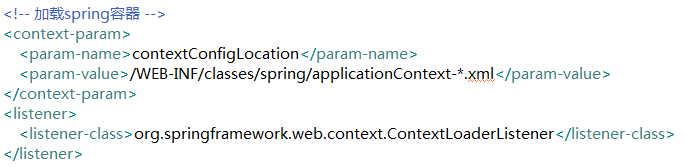
### 8、配置Spring容器（重要）

以上所有配置文件配置完成后，在web.xml文件中通过配置SpringMVC的前端控制器，则可以完成将springmvc.xml配置文件加载到应用的容器中去，如下所示：



此操作仅仅是完成了SpringMVC模块的加载，但是Spring容器还没有加载到Web应用中来，所以下一步需要将Spring容器加载到Web应用的容器中，通过Spring框架提供的org.springframework.web.context.

ContextLoaderListener监听器来完成，此监听器在启动时会自动记载web.xml配置文件中的context-param元素中指定contextConfigLocation的value值，如下所示：



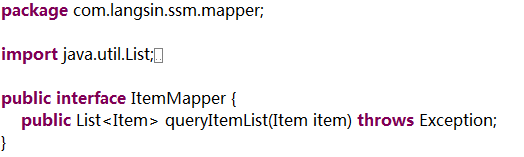
到此SpringMVC+Spring+Mybatis的整合全部结束！

## 5.7 商品查询调试

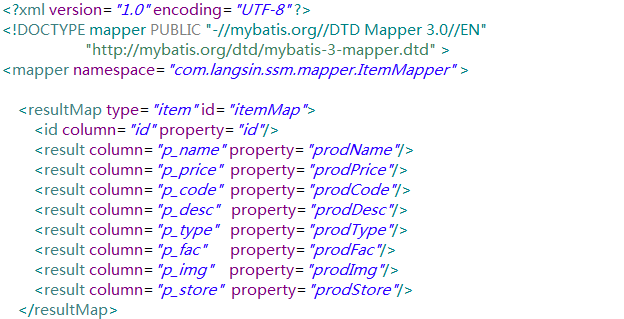
整合工作完成后，下面来开发商品管理功能模块来进行测试。

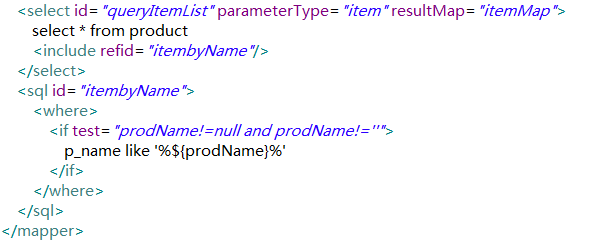
### 5.7.1 dao开发

1、mapper接口开发以及mapper.xml文件如下所示：



mapper.xml文件如下所示：

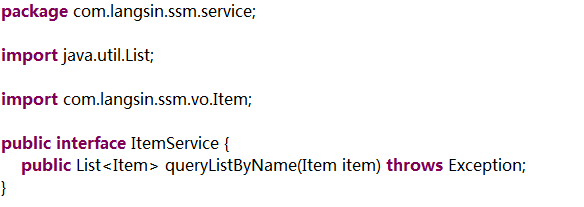




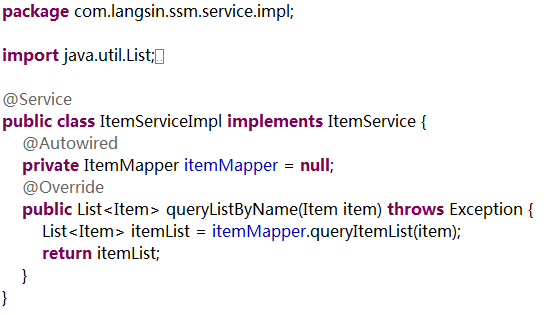
### 5.7.2 service层开发

在应用中对Service层使用xml的配置方式进行管理，而不使用spring的自动加载，这是因为在Handler层（Controller）层中对Service层的定义是通过接口定义的，例如ItemService itemservice = null;变量名为itemService而如果使用自动装配的话，那么对于Service层的所有的定义，变量名全部进行了限定。使用xml配置的方式反而更加灵活。如下所示：

1、接口定义



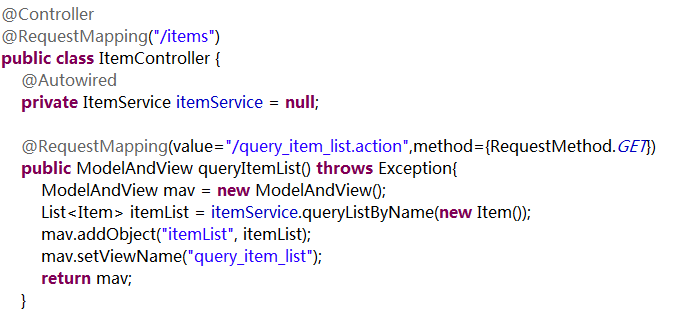
2、实现类定义



在Spring的配置文件中，配置该Java Bean。对于Service中成员变量itemMapper要遵循一定的规范，Mapper是动态代理的，在Spring的dao配置文件中整合的时候已经被Spring容器加载进来，动态代理的Mapper的Bean Id为类名的首字母小写。而且在进行自动注入时需要通过byName的方式完成，所以变量名要写成Bean Id的值。

### 5.7.3 Controller层开发

下面来编写Controller层（Handler层），如下所示：



在Controller类中，使用注解@Controller进行修饰，由Spring框架完成自动加载，用户在访问该url：query\_item\_list.action时，SpringMVC根据此处理器映射器找到该类，然后再将该类交由SpirngMVC的处理器适配器与执行该类，最后返回一个view。

## 5.8 商品信息修改

在上面的示例中仅仅是完成最简单的操作，如果需要完成稍微复杂一点的开发，比如限制请求方式、往后台传入参数等，这就需要利用SpringMVC框架提供的注解@RequestMapping，以及处理器Controller层中方法的返回值以及传入参数来完成。本节示例—商品信息修改将利用这几点技术来完成，首先来看一下个技术点的内容。

### 5.8.1 @RequestMapping

通过RequestMapping注解可以定义不同的处理器映射规则。

* URL路径映射：@RequestMapping(value="/item")或@RequestMapping("/item），value的值是数组，可以将多个url映射到同一个方法。
* 窄化请求映射：在class上添加@RequestMapping(url)指定通用请求前缀， 限制此类下的所有方法请求url必须以请求前缀开头，通过此方法对url进行分类管理。

例如：

@RequestMapping放在类名上边，设置请求前缀

@Controller

@RequestMapping("/item")

方法名上边设置请求映射url：

@RequestMapping放在方法名上边，如下：

@RequestMapping("/queryItem ")

访问地址为：/item/queryItem

* 请求方法限定：

限定GET方法

@RequestMapping(method = RequestMethod.GET)

如果通过Post访问则报错：

HTTP Status 405 - Request method 'POST' not supported

例如：

@RequestMapping(value="/editItem",method=RequestMethod.GET)

限定POST方法

@RequestMapping(method = RequestMethod.POST)

如果通过Post访问则报错：

HTTP Status 405 - Request method 'GET' not supported

GET和POST都可以

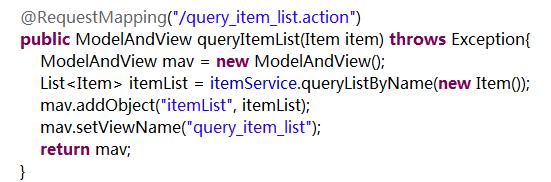
@RequestMapping(method={RequestMethod.GET,RequestMethod.POST})

### 5.8.2 Controller方法的返回值

在Controller层中，响应处理请求的方法，可以定义如下几种类型，返回一个ModelAndView或者返回一个String字符串，或者使用void修饰。

* 返回ModelAndView

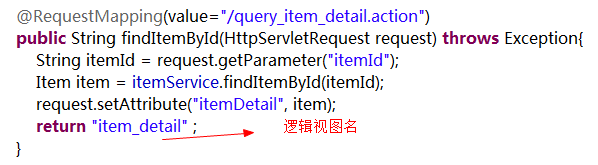
controller方法中定义ModelAndView对象并返回，对象中可添加model数据、指定view。如下所示：



* 返回String字符串

使用字符串作为返回值类型时，Controler层中的方法可以实现如下三种功能：1、返回逻辑视图名2、实现重定向3、实现请求转发

**1、返回逻辑视图名**



controller方法返回字符串可以指定逻辑视图名，通过视图解析器解析为物理视图地址。

**物理视图地址=视图解析器前缀+逻辑视图名+视图解析器后缀**

**2、实现重定向**

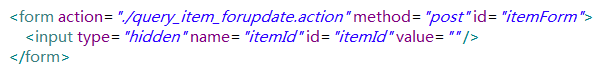
重定向的原理与Servlet类中的重定向原理一致，问题是在SpringMVC中将数据模型model放 request域中，而重定向实际是发起了两次请求，那么在第一次请求中获取的数据，是无法与第二次的request进行共享，所以在重定向时，一般做法是让SpringMVC重新执行一次action动作，例如：商品修改完成后，返回到列表页面。

示例如下：



选中一条记录进行修改，然后将获取到的商品ID放入一个form表单的隐藏域中，然后将隐藏域提交即可，隐藏域与修改的JQuery代码如下：

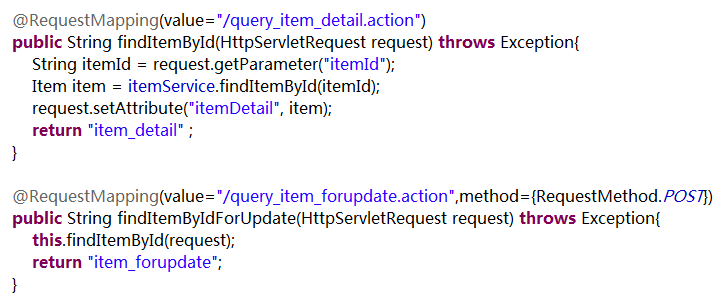
**隐藏域：**



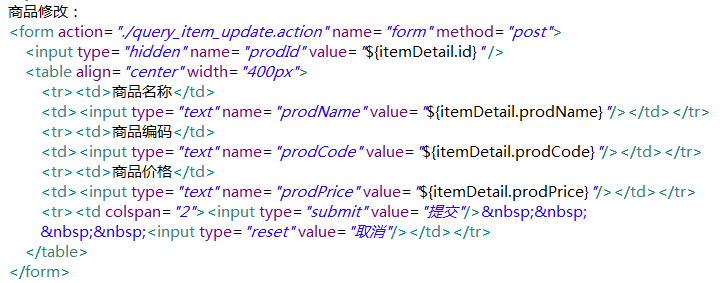
**JQuery代码：**



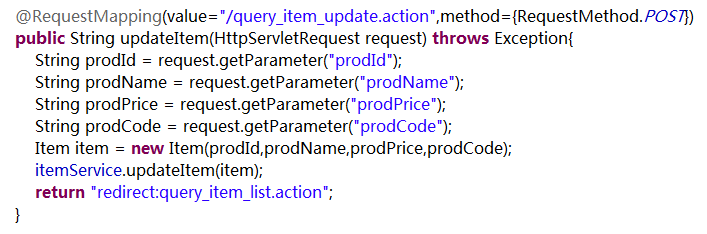
最终请求到query\_item\_forupdate，对应的Controller处理方法如下：



因为在Controller中已经存在一个根据ID来获取商品的方法，所以调用上面已有的方法，但是因为上面的方法返回的视图名不是query\_item\_forupdate要转到的页面，所以需要重写return语句。返回页面代码如下：



重点：在本次提价后完成重定向操作！

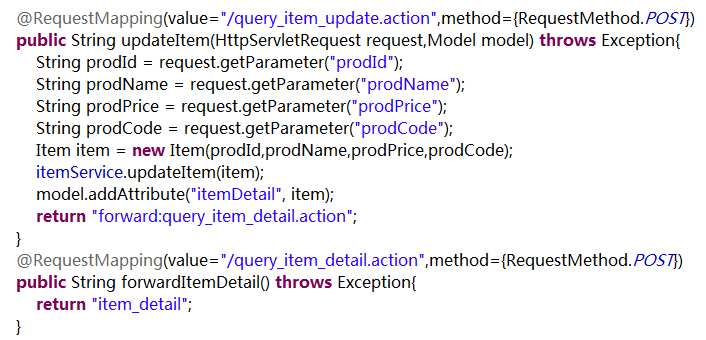


在返回的重定向的字符串中，redirect后面与冒号紧挨着，不能出现空格字符串，慎重！

**3、实现请求转发**

请求转发与Servlet中的请求转发一致，服务器的多次处理在一个请求范围内，所以数据模型model可以实现共享。如下所示：

在上例中加入商品修改完成后，转入详细页面，则代码如下：



在上面方法updateItem中返回的字符串为：forward：query\_item\_detail.action路径，也可以直接写成要访问的JSP页面，但是这样要考虑路径的问题，所以简单来做可以返回一个映射后的url，如query\_item\_detail.action。