# Spring

尚硅谷Java研究院

版本：V1.0

## 第1章 简介

### 1.1 Spring概述

Spring是一个为简化企业级开发而生的开源框架。使用Spring开发可以将Bean对象，Dao组件对象，Service组件对象等交给Spring容器来管理，这样使得很多复杂的代码在Spring中开发却变得非常的优雅和简洁，有效的降低代码的耦合度，极大的方便项目的后期维护、升级和扩展。

Spring是一个IOC(DI)和AOP容器框架。

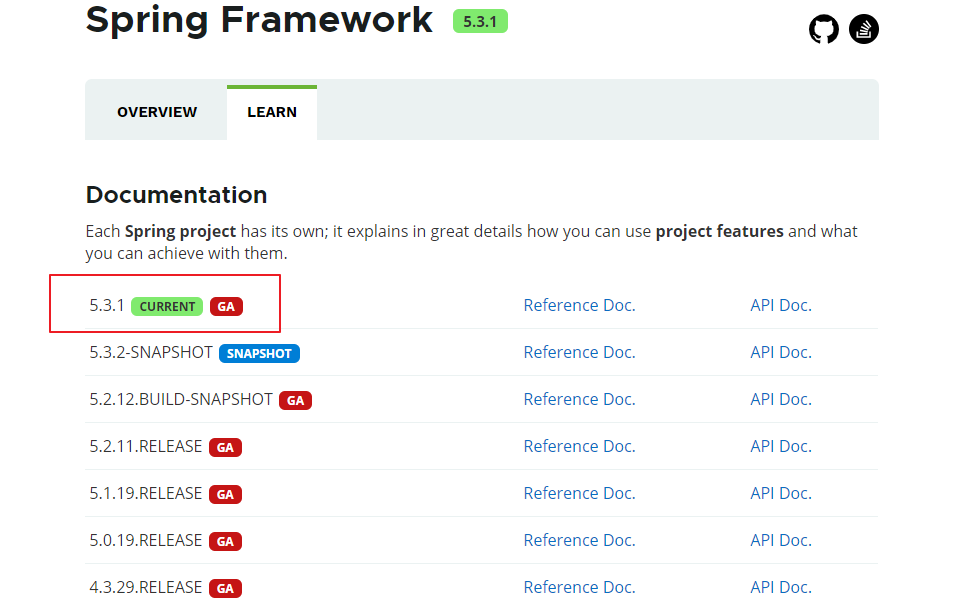
Spring的优良特性：

1. 非侵入式：基于Spring开发的应用中的对象可以不依赖于Spring的API。
2. 控制反转：IOC——Inversion of Control，指的是将对象的创建权交给Spring去创建。使用Spring之前，对象的创建都是由我们自己在代码中new创建。而使用Spring之后。对象的创建都是由给了Spring框架。
3. 依赖注入：DI——Dependency Injection，是指依赖的对象不需要手动调用setXXX方法去设置，而是通过配置赋值。
4. 面向切面编程：AOP——Aspect Oriented Programming，在不修改源代码的基础上进行功能扩展。
5. 容器：Spring是一个容器，因为它包含并且管理应用对象的生命周期。
6. 组件化：Spring实现了使用简单的组件配置组合成一个复杂的应用。在 Spring 中可以使用XML和Java注解组合这些对象。
7. 一站式：在IOC和AOP的基础上可以整合各种企业应用的开源框架和优秀的第三方类库（实际上Spring 自身也提供了表述层的SpringMVC和持久层的JDBCTemplate）。

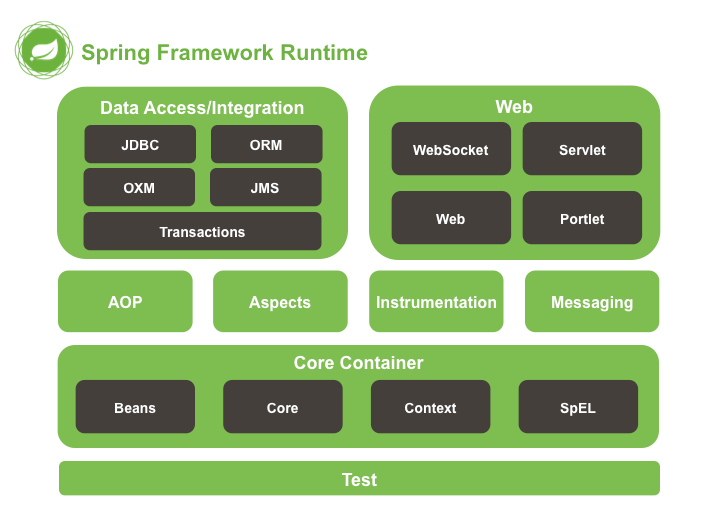
### 1.2 Spring的下载地址

官网：<https://spring.io/>

最新正式发布版下载地址：<https://repo.spring.io/release/org/springframework/spring/>



### 1.3 Spring模块



Spring框架分为五大模块：

1. Core Container：核心容器

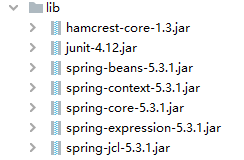
* Beans提供 BeanFactory，它是一个工厂模式的复杂实现。
* Core提供了框架的基本组成部分，包括IOC和DI功能。
* Context建立在由核心和 Bean 模块提供的基础上，它是访问定义和配置的任何对象的媒介。ApplicationContext 接口是上下文模块的重点。
* SpEL在运行时提供了查询和操作一个对象的强大的Spring表达式语言。

1. AOP&Aspects：提供了面向切面编程的实现。
2. DataAccess/Integration：提供了对数据访问/集成的功能。
3. Web：提供了面向Web应用程序的集成功能。
4. Test：提供了对JUnit 或 TestNG 框架的测试功能。

### 1.4 HelloWorld

创建HelloWorld的步骤如下：

1. 创建Java工程，并导入以下jar包

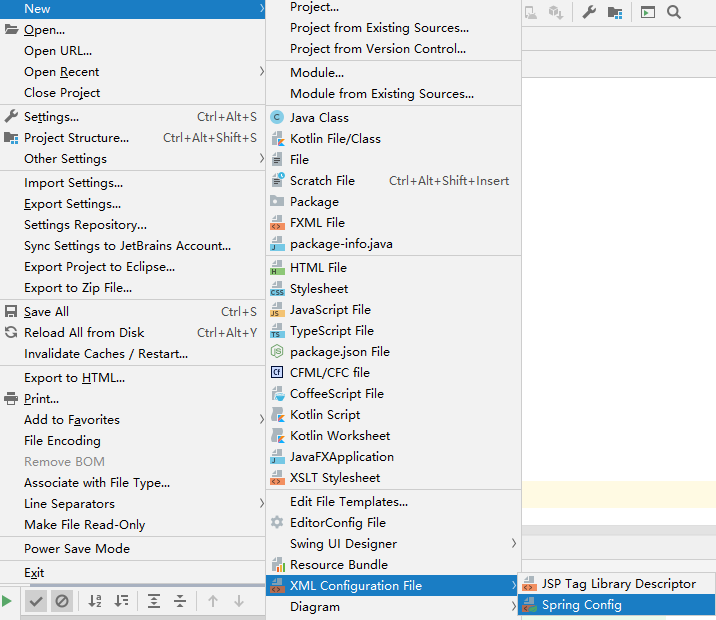


1. 创建Java类HelloWorld

* 声明一个name属性
* 提供setName方法
* 创建sayHello方法

|  |
| --- |
| **public class** HelloWorld {  **private** String **name**;   **public void** setName(String name) {  **this**.**name** = name;  }   **public void** sayHello(){  System.***out***.println(**"Hello :"**+**name**);  } } |

1. 在src目录下创建Spring Config配置文件，并命名为applicationContext.xml



applicationContext.xml文件中的内容

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>   *<!--配置bean  id属性：设置bean的唯一标识  class属性：设置类的全类名，IOC容器通过反射创建对象  -->* <**bean id="helloWorld" class="com.atguigu.spring.helloworld.HelloWorld"**>  <**property name="name" value="Spring"**></**property**>  </**bean**> </**beans**> |

1. 创建单元测试类HelloWorldTest

|  |
| --- |
| **public class** HelloWorldTest {  */\*  测试HelloWorld  \*/* @Test  **public void** testHelloWorld(){  *//1.创建IOC容器对象* ApplicationContext ioc = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  *//2.获取IOC容器中的HelloWorld对象* HelloWorld helloWorld = (HelloWorld) ioc.getBean(**"helloWorld"**);  *//3.调用sayHello方法* helloWorld.sayHello();  } } |

### 1.5 获取bean的方式

1. 根据bean的id获取

* 需要强转

|  |
| --- |
| HelloWorld helloWorld = (HelloWorld) ioc.getBean(**"helloWorld"**); |

1. 根据bean的类型获取

* 如果在IOC容器中有多个该类型的bean则会抛出异常

|  |
| --- |
| HelloWorld helloWorld = ioc.getBean(HelloWorld.**class**); |

1. 根据bean的id和类型获取

|  |
| --- |
| HelloWorld helloWorld = ioc.getBean(**"helloWorld"**, HelloWorld.**class**); |

## 第2章 基于XML的方式管理Bean

### 2.1 IOC和DI

#### 2.1.1 IOC(Inversion of Control)：反转控制

在应用程序中的组件需要获取资源时，传统的方式是组件主动的从容器中获取所需要的资源，在这样的模式下开发人员往往需要知道在具体容器中特定资源的获取方式，增加了学习成本，同时降低了开发效率。

反转控制的思想完全颠覆了应用程序组件获取资源的传统方式：反转了资源的获取方向——改由容器主动的将资源推送给需要的组件，开发人员不需要知道容器是如何创建资源对象的，只需要提供接收资源的方式即可，极大的降低了学习成本，提高了开发的效率。这种行为也称为查找的被动形式。

#### 2.1.2 DI(Dependency Injection)：依赖注入

IOC的另一种表述方式：即组件以一些预先定义好的方式(例如：setter 方法)接受来自于容器的资源注入。相对于IOC而言，这种表述更直接。

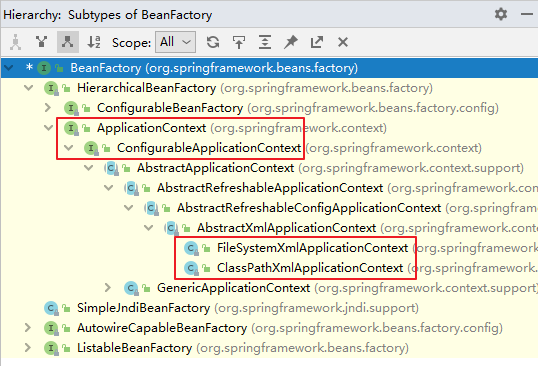
总结: IOC 就是一种反转控制的思想， 而DI是对IOC的一种具体实现。

#### 2.1.3 IOC容器在Spring中的实现

在创建Bean之前，首先需要创建IOC容器。Spring提供了IOC容器的两种实现方式：

1. BeanFactory：IOC容器的基本实现，是Spring内部的使用接口，是面向Spring本身的，不是提供给开发人员使用的。
2. ApplicationContext：BeanFactory的子接口，提供了更多高级特性。面向Spring的使用者，几乎所有场合都使用ApplicationContext而不是底层的BeanFactory。

#### 2.1.4 ApplicationContext的主要实现类



1. ClassPathXmlApplicationContext：对应类路径下的XML格式的配置文件
2. FileSystemXmlApplicationContext：对应文件系统中的XML格式的配置文件
3. ConfigurableApplicationContext：是ApplicationContext的子接口，包含一些扩展方法refresh()和close()，让ApplicationContext具有启动、关闭和刷新上下文的能力。
4. WebApplicationContext：专门为WEB应用而准备的，它允许从相对于WEB根目录的路径中完成初始化工作。

### 2.2 创建Bean

让Spring的IOC容器帮我们创建Bean，只需要在Spring的配置文件中通过bean标签配置即可。

bean标签中常用属性说明：

1. Id属性：给IOC容器中的bean设置的唯一标识。
2. class属性：配置让IOC容器管理的类的全类名，Spring会利用反射技术通过类的无参构造器创建对象。

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>   <**bean id="book" class="com.atguigu.spring.beans.Book"**></**bean**> </**beans**> |

### 2.3 给Bean的属性赋值

#### 2.3.1 setter注入

在bean标签中通过property标签设置bean的属性值，property标签中的属性说明：

* name属性：配置bean的属性名
* value属性：配置bean的属性值
* 还可以通过value子标签配置bean的属性值

|  |
| --- |
| <**bean id="book1" class="com.atguigu.spring.beans.Book"**>  <**property name="id" value="1"**></**property**>  <**property name="title" value="三国演义"**></**property**>  <**property name="author" value="罗贯中"**></**property**>  <**property name="price" value="66.66"**></**property**>  <**property name="sales" value="100"**></**property**>  </**bean**> |

#### 2.3.2 构造器注入

在bean标签中通过constructor-arg标签设置bean的属性值，constructor-arg标签中的属性说明：

* name属性：配置构造器中的参数名
* value属性：配置参数值
* index属性：配置参数在构造器中的索引位置，从0开始
* type属性：配置构造器中参数的类型

1. 只指定value属性，自动匹配合适的构造器

|  |
| --- |
| <**bean id="book2" class="com.atguigu.spring.beans.Book"**>  <**constructor-arg value="2"**></**constructor-arg**>  <**constructor-arg value="水浒传"**></**constructor-arg**>  <**constructor-arg value="施耐庵"**></**constructor-arg**>  <**constructor-arg value="88.88"**></**constructor-arg**>  <**constructor-arg value="100"**></**constructor-arg**> </**bean**> |

1. 所有属性都指定

|  |
| --- |
| <**bean id="book3" class="com.atguigu.spring.beans.Book"**>  <**constructor-arg name="id" value="3"**></**constructor-arg**>  <**constructor-arg name="title" value="红楼梦"**></**constructor-arg**>  <**constructor-arg index="2" value="曹雪芹"**></**constructor-arg**>  <**constructor-arg type="java.lang.Integer" value="100"**></**constructor-arg**> </**bean**> |

#### 2.3.3 特殊值处理

1. 字面量

字面量是用于表达源代码中一个固定值的表示法。基本数据类型及其包装类型、String等类型都可以采取字面值注入的方式。

1. 赋null值

通过<null></null>标签给bean的属性赋null值

1. 赋值中包含特殊字符

若字面值中包含特殊字符，可以使用<![CDATA[ ]]>把字面值包裹起来

|  |
| --- |
| <**bean id="book4" class="com.atguigu.spring.beans.Book"**>  <**property name="id" value="4"**></**property**>  <**property name="title"**>  <**value**><![CDATA[<<西游记>>]]></**value**>  </**property**>  <**property name="author" value="吴承恩"**></**property**>  <**property name="sales"**>  <**null**></**null**>  </**property**> </**bean**> |

#### 2.3.4 使用p名称空间

使用p名称空间需要引入对应的约束，在Idea中根据提示引入即可。

|  |
| --- |
| <**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.xsd"**>  *<!--通过p名称空间配置bean-->*  <**bean id="book5" class="com.atguigu.spring.beans.Book"   p:id="5" p:title="解忧杂货店"   p:author="藤野圭吾"   p:price="33.00"   p:sales="100"**></**bean**>  </**beans**> |

#### 2.3.5 引用外部已声明的bean

当bean的属性是一个其他类型的对象是，可以在property或<constructor-arg>标签中通过ref属性或ref子标签引用IOC容器中配置好的该类型的bean，ref的属性值指定为引用的bean的id值。

|  |
| --- |
| <**bean id="cartItem" class="com.atguigu.spring.beans.CartItem"**>  *<!--引用外部bean-->*  <**property name="book" ref="book1"**></**property**>  <**property name="count" value="10"**></**property**>  <**property name="amount" value="100"**></**property**> </**bean**> |

#### 2.3.6 内部bean

当bean实例仅仅给一个特定的属性使用时，可以将其声明为内部bean。内部bean声明直接包含在<property>或<constructor-arg>元素里，不需要设置id。

|  |
| --- |
| <**bean id="cartItem2" class="com.atguigu.spring.beans.CartItem"**>  <**property name="book"**>  *<!--配置内部bean-->*  <**bean class="com.atguigu.spring.beans.Book"**>  <**property name="id" value="6"**></**property**>  <**property name="title" value="三体"**></**property**>  <**property name="author" value="刘慈欣"**></**property**>  <**property name="price" value="22.00"**></**property**>  <**property name="sales" value="100"**></**property**>  </**bean**>  </**property**>  <**property name="count" value="10"**></**property**>  <**property name="amount" value="100"**></**property**> </**bean**> |

#### 2.3.7 级联属性赋值

当bean的属性是一个对象，我们可以通过配置当前bean的方式给属性中对象的属性赋值，即给属性的属性赋值，这种方式我们称为给级联属性赋值。

|  |
| --- |
| *<!--给级联属性赋值-->*  <**bean id="cartItem3" class="com.atguigu.spring.beans.CartItem"**>  <**property name="book" ref="book1"**></**property**>  *<!--通过给级联属性赋值将book1中的title修改为新三国-->* <**property name="book.title" value="新三国"**></**property**> </**bean**> |

#### 2.3.8 集合属性

当bean的属性是集合类型时，可以通过以下标签进行配置：

* <array></array>：配置数组类型
* <list></list>：配置List类型
* <map><entry></entry></map>：配置Map类型

|  |
| --- |
| *<!--配置集合属性-->*  <**bean id="bookShop" class="com.atguigu.spring.beans.BookShop"**>  <**property name="books"**>  <**list**>  <**ref bean="book1"**></**ref**>  <**ref bean="book2"**></**ref**>  <**ref bean="book3"**></**ref**>  </**list**>  </**property**>  <**property name="map"**>  <**map**>  <**entry key="user1" value="张三"**></**entry**>  <**entry key="user2" value="李四"**></**entry**>  <**entry key="user3" value="王五"**></**entry**>  </**map**>  </**property**> </**bean**> |

#### 2.3.9 集合类型的bean

如果只能将集合对象配置在某个bean内部，则这个集合的配置将不能被重用。我们需要将集合bean的配置拿到外面，供其他bean引用。

配置集合类型的bean需要引入util名称空间。

|  |
| --- |
| <**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.xsd http://www.springframework.org/schema/util https://www.springframework.org/schema/util/spring-util.xsd"**>  *<!--配置集合bean-->*  <**util:list id="listBean"**>  <**bean class="com.atguigu.spring.beans.Book"**>  <**property name="id" value="1"**></**property**>  <**property name="title" value="大秦帝国之裂变"**></**property**>  <**property name="author" value="孙皓晖"**></**property**>  </**bean**>  <**bean class="com.atguigu.spring.beans.Book"**>  <**property name="id" value="2"**></**property**>  <**property name="title" value="大秦帝国之纵横"**></**property**>  <**property name="author" value="孙皓晖"**></**property**>  </**bean**>  <**bean class="com.atguigu.spring.beans.Book"**>  <**property name="id" value="3"**></**property**>  <**property name="title" value="大秦帝国之崛起"**></**property**>  <**property name="author" value="孙皓晖"**></**property**>  </**bean**>  </**util:list**> </**beans**> |

#### 2.3.10 自动装配

手动装配：以value或ref的方式明确指定属性值都是手动装配。

自动装配：根据bean标签的autowire属性指定的装配规则，不需要明确指定，Spring自动将匹配的属性值注入bean中。

自动装配的规则，即autowire的属性值有：

* no或default：不自动装配
* byName：根据bean的属性名称自动装配，以当前bean的属性名作为id从IOC容器中寻找以实现装配。找到则装配，找不到则不装配。
* byType：根据bean的属性类型自动装配。找到一个则装配，找到多个则报错，找不到则不装配。
* constructor：根据bean的属性的构造器自动装配，不推荐使用。

|  |
| --- |
| <**bean id="cartItem4" class="com.atguigu.spring.beans.CartItem" autowire="byName"**></**bean**> |

#### 2.3.11 引入外部属性文件

当bean的配置信息逐渐增多时，查找和修改一些bean的配置信息就变得愈加困难。这时可以将一部分信息提取到bean配置文件的外部，以properties格式的属性文件保存起来，同时在bean的配置文件中引用properties属性文件中的内容，从而实现一部分属性值在发生变化时仅修改properties属性文件即可。这种技术多用于连接数据库的基本信息的配置。

1. 导入druid-1.1.9.jar和mysql-connector-java-5.1.37-bin.jar
2. 在src目录下创建外部属性文件druid.properties

|  |
| --- |
| *#key=value* **jdbc.username**=**root jdbc.password**=**root jdbc.url**=**jdbc:mysql://localhost:3306/test jdbc.driverClassName**=**com.mysql.jdbc.Driver  jdbc.initialSize**=**5 jdbc.maxActive**=**10** |

1. 在Spring的配置文件中通过<context:property-placeholder location=”classpath:druid.properties”/>标签引入外部属性文件”

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**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"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/util  https://www.springframework.org/schema/util/spring-util.xsd  http://www.springframework.org/schema/context  https://www.springframework.org/schema/context/spring-context.xsd"**>  *<!--引入外部属性文件-->* <**context:property-placeholder location="classpath:druid.properties"**></**context:property-placeholder**>  *<!--配置数据源-->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"**>  <**property name="username" value="${jdbc.username}"**></**property**>  <**property name="password" value="${jdbc.password}"**></**property**>  <**property name="url" value="${jdbc.url}"**></**property**>  <**property name="driverClassName" value="${jdbc.driverClassName}"**></**property**>  <**property name="initialSize" value="${jdbc.initialSize}"**></**property**>  <**property name="maxActive" value="${jdbc.maxActive}"**></**property**> </**bean**> |

#### 2.3.12 FactoryBean

Spring中有两种类型的bean，一种是普通bean，另一种是工厂bean，即FactoryBean。普通bean设置的类型就是返回的类型；工厂bean设置的类型可以和返回的类型不一样，其返回的类型通过该工厂bean的getObject方法指定。

创建工厂bean必须实现org.springframework.beans.factory.FactoryBean接口。

创建工厂bean的步骤：

1. 创建一个类实现FactoryBean接口

|  |
| --- |
| **package** com.atguigu.spring.beans;  **import** org.springframework.beans.factory.FactoryBean;  **public class** MyFactoryBean **implements** FactoryBean<Book> {  *//设置返回的对象* @Override  **public** Book getObject() **throws** Exception {  **return new** Book(7,**"生命不息，奋斗不止"**,**"罗永浩"**,11.11,1000);  }   *//设置对象的类型* @Override  **public** Class<?> getObjectType() {  **return** Book.**class**;  }   *//设置当前bean是否是单例的* @Override  **public boolean** isSingleton() {  **return true**;  } } |

1. 在Spring的配置文件中配置工厂bean

|  |
| --- |
| *<!--配置工厂bean-->* <**bean id="myFactoryBean" class="com.atguigu.spring.beans.MyFactoryBean"**></**bean**> |

1. 测试

|  |
| --- |
| */\*  测试工厂bean \*/* @Test **public void** testFactoryBean(){  Book book = (Book) **ioc**.getBean(**"myFactoryBean"**);  System.***out***.println(book); } |

### 2.4 Bean的作用域

在Spring中，可以在bean标签的scope属性里设置bean的作用域，以决定这个bean是单实例的还是多实例的。

默认情况下，Spring只为每个在IOC容器里声明的bean创建唯一一个实例，整个IOC容器范围内都能共享该实例：所有后续的getBean()调用和bean引用都将返回这个唯一的bean实例。该作用域被称为singleton，它是所有bean的默认作用域。

当bean的作用域为单例时，Spring会在IOC容器对象创建时就创建bean的对象实例。而当bean的作用域为prototype时，IOC容器在获取bean的实例时创建bean的实例对象，而且每调用一次getBean()就创建一个实例。

|  |  |
| --- | --- |
| 类别 | 说明 |
| singleton | 在Spring的IOC容器中仅存在一个Bean实例 |
| prototype | 每次调用getBean方法都返回一个新的Bean实例 |
| request | 每次HTTP请求都会创建一个新的Bean实例，该作用域仅适用于WebApplicationContext环境 |
| session | 同一个Session会话共享一个Bean实例，该作用域仅适用于WebApplicationContext环境 |

### 2.5 Bean的生命周期

Spring IOC容器可以管理bean的生命周期，Spring允许在bean生命周期内特定的时间点执行指定的任务。

1. Spring IOC容器对bean的生命周期进行管理的过程：

① 通过构造器或工厂方法创建bean实例

② 为bean的属性设置值和对其他bean的引用

③ bean可以使用了

1. 在配置bean时，可以通过init-method和destroy-method 属性为bean指定初始化和销毁的方法

* 创建Java类

|  |
| --- |
| **package** com.atguigu.spring.beans;  **public class** BeanLife {  **private** String **name**;   **public** BeanLife() {  System.***out***.println(**"BeanLife对象被创建"**);  }   **public void** setName(String name) {  System.***out***.println(**"BeanLife的name属性被赋值"**);  **this**.**name** = name;  }   **public void** initMethod(){  System.***out***.println(**"初始化的方法"**);  }   **public void** destoryMethod() {  System.***out***.println(**"销毁的方法"**);  }   @Override  **public** String toString() {  **return "BeanLife{"** +  **"name='"** + **name** + **'\''** +  **'}'**;  } } |

* 在Spring的配置文件中配置初始化和销毁的方法

|  |
| --- |
| *<!--Bean的生命周期-->* <**bean id="beanLife" class="com.atguigu.spring.beans.BeanLife" init-method="initMethod" destroy-method="destoryMethod"**>  <**property name="name" value="测试Bean的生命周期"**></**property**> </**bean**> |

* 测试

|  |
| --- |
| */\*  测试Bean的生命周期 \*/* @Test **public void** testBeanLife(){  BeanLife beanLife = (BeanLife) **ioc**.getBean(**"beanLife"**);  System.***out***.println(beanLife+**"对象可以使用了"**);  *//关闭容器* ConfigurableApplicationContext coc = (ConfigurableApplicationContext)**ioc**;  coc.close(); } |

* 配置了这两个方法之后的生命周期过程：

① 通过构造器或工厂方法创建bean实例

② 为bean的属性设置值和对其他bean的引用

③ 调用bean的初始化方法

④ bean可以使用了

⑤ 当容器关闭时，调用bean的销毁方法

1. bean的后置处理器

* bean后置处理器允许在调用初始化方法前后对bean进行额外的处理。
* bean后置处理器对IOC容器里的所有bean实例逐一处理，而非单一实例，其典型应用是：检查bean属性的正确性或根据特定的标准更改bean的属性。
* bean后置处理器需要实现接口：

org.springframework.beans.factory.config.BeanPostProcessor。在初始化方法被调用前后，Spring将把每个bean实例分别传递给上述接口的以下两个方法：

* postProcessBeforeInitialization(Object, String)
* postProcessAfterInitialization(Object, String)
* 创建Bean的后置处理器

|  |
| --- |
| **package** com.atguigu.spring.beans;  **import** org.springframework.beans.BeansException; **import** org.springframework.beans.factory.config.BeanPostProcessor;  **public class** MyBeanPostProcessor **implements** BeanPostProcessor {   */\*  bean：传入的bean对象  beanName：bean的名称，即IOC容器中配置的bean  \*/* @Override  **public** Object postProcessBeforeInitialization(Object bean, String beanName) **throws** BeansException {  System.***out***.println(**"初始化方法之前执行"**);  **return** bean;  }   @Override  **public** Object postProcessAfterInitialization(Object bean, String beanName) **throws** BeansException {  System.***out***.println(**"初始化方法之后执行"**);  **return** bean;  } } |

* 在Spring的配置文件中配置Bean的后置处理器

|  |
| --- |
| *<!--配置Bean的后置处理器-->* <**bean id="myBeanPostProcessor" class="com.atguigu.spring.beans.MyBeanPostProcessor"**></**bean**> |

* 配置了后置处理器之后的生命周期过程：

①通过构造器或工厂方法创建bean实例

②为bean的属性设置值和对其他bean的引用

③将bean实例传递给bean后置处理器的postProcessBeforeInitialization()方法

④调用bean的初始化方法

⑤将bean实例传递给bean后置处理器的postProcessAfterInitialization()方法

⑥bean可以使用了

⑦当容器关闭时调用bean的销毁方法

## 第3章 基于注解的方式管理Bean

### 3.1 创建Bean

#### 3.1.1 使用注解标识组件

1. @Component

标识一个受Spring IOC容器管理的组件

1. @Repository

标识一个受Spring IOC容器管理的持久化层组件

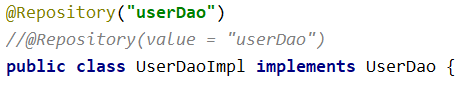
1. @Service

标识一个受Spring IOC容器管理的业务逻辑层组件

1. @Controller

标识一个受Spring IOC容器管理的表述层控制器组件

1. 组件命名规则
   1. 默认情况：使用组件的简单类名首字母小写后得到的字符串作为bean的id
   2. 我们可以使用组件注解的value属性指定bean的id，value属性名可以省略



注意：事实上Spring并没有能力识别一个组件到底是不是它所标记的类型，即使将 @Respository注解用在一个业务逻辑层组件上面也不会产生任何错误，所以@Respository、@Service、@Controller这几个注解仅仅是为了让开发人员自己明确当前的组件扮演的角色。

#### 3.1.2 组件扫描

组件被上述注解标识后还需要通过Spring进行扫描才能够侦测到。

1. 指定被扫描的包

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**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.xsd http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd"**>   *<!--配置自动扫描的包-->* <**context:component-scan base-package="com.atguigu.spring.annotation"  resource-pattern="dao/impl/\*.class"**></**context:component-scan**> </**beans**> |

属性说明：

* + base-package属性指定一个需要扫描的基类包，Spring容器将会扫描这个基类包及其子包中的所有类。
  + 当需要扫描多个包时可以使用逗号分隔。
  + 如果仅希望扫描特定的类而非基包下的所有类，可使用resource-pattern属性过滤特定的类。

1. 通过注解的方式管理Bean还需要在原有jar包的基础上导入spring-aop-5.3.1.jar
2. 设置扫描和不扫描的类

* 设置扫描

|  |
| --- |
| *<!--配置自动扫描的包-->* <**context:component-scan base-package="com.atguigu.spring.annotation" use-default-filters="false"**>  *<!--设置只扫描那个包下的类--> <!-- <context:include-filter type="annotation" expression="org.springframework.stereotype.Repository"/>-->* <**context:include-filter type="assignable" expression="com.atguigu.spring.annotation.dao.UserDao"**/>  </**context:component-scan**> |

注意：让子标签<context:include-filter>起作用的前提需要将父标签的use-default-filters属性设置为false，禁用默认过滤器。

* 设置不扫描

|  |
| --- |
| *<!--配置自动扫描的包-->* <**context:component-scan base-package="com.atguigu.spring.annotation"**>  *<!--设置不扫描那个包下的类--> <!-- <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Repository"/>-->* <**context:exclude-filter type="assignable" expression="com.atguigu.spring.annotation.dao.UserDao"**/>  </**context:component-scan**> |

type属性值说明：

|  |  |  |
| --- | --- | --- |
| 类别 | 示例 | 说明 |
| annotation | com.atguigu.XxxAnnotation | 过滤所有标注了XxxAnnotation的类。这个规则根据目标组件是否标注了指定类型的注解进行过滤。 |
| assignable | com.atguigu.BaseXxx | 过滤所有BaseXxx类的子类。这个规则根据目标组件是否是指定类型的子类的方式进行过滤。 |
| aspectj | com.atguigu.\*Service+ | 所有类名是以Service结束的，或这样的类的子类。这个规则根据AspectJ表达式进行过滤。 |
| regex | com\.atguigu\.anno\.\* | 所有com.atguigu.anno包下的类。这个规则根据正则表达式匹配到的类名进行过滤。 |
| custom | com.atguigu.XxxTypeFilter | 使用XxxTypeFilter类通过编码的方式自定义过滤规则。该类必须实现org.springframework.core.type.filter.TypeFilter接口 |

* component-scan下可以拥有若干个include-filter和exclude-filter子标签

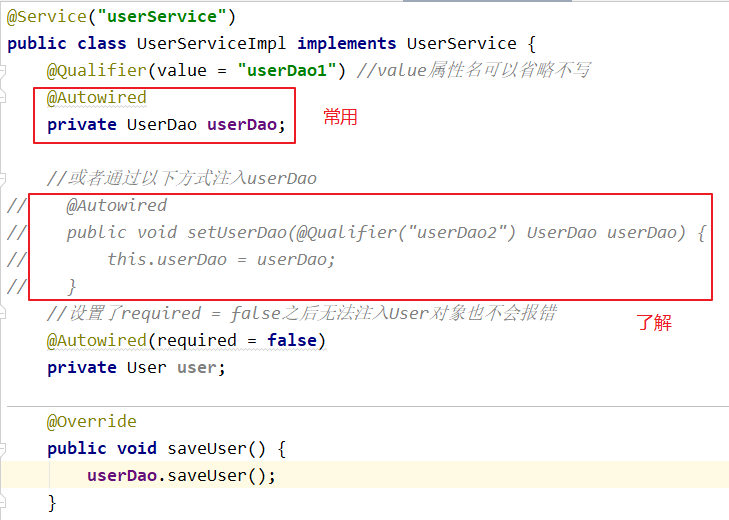
### 3.2 给Bean的属性注入值——自动装配

Controller组件中往往需要用到Service组件的实例，Service组件中往往需要用到Repository组件的实例。Spring可以通过注解的方式帮我们实现属性的装配。

在指定要扫描的包时，<context:component-scan> 元素会自动注册一个bean的后置处理器：AutowiredAnnotationBeanPostProcessor的实例。该后置处理器可以自动装配标记了@Autowired、@Resource或@Inject注解的属性。

1. @Autowired注解

* 构造器、普通字段(即使是非public)、一切具有参数的方法都可以应用@Autowired注解。
* 默认情况下，所有使用@Autowired注解的属性都需要被设置。当Spring找不到匹配的bean装配属性时，会抛出异常。
* 若某一属性允许不被设置，可以设置@Autowired注解的required属性为 false。
* @Autowired注解也可以应用在数组类型的属性上，此时Spring将会把所有匹配的bean进行自动装配。
* @Autowired注解也可以应用在集合属性上，此时Spring读取该集合的类型信息，然后自动装配所有与之兼容的bean。
* @Autowired注解用在java.util.Map上时，若该Map的键值为String，那么 Spring将自动装配与值类型兼容的bean作为值，并以bean的id值作为键。



自动装配（注入）的过程：

* 1. 根据属性的类型实现自动装配。
  2. 如果IOC容器中该属性的类型的bean存在多个，则以属性名作为id从IOC容器中寻找以实现自动装配，找到则装配成功；找不到则抛出异常。
  3. 如果IOC容器中该属性的类型的bean存在多个，并且以属性名作为id从IOC容器中寻找也无法实现自动装配，还可以通过@Qualifier注解指定bean的名称实现自动装配。@Qualifiter注解也可以标注在方法的形参前面。

1. @Resource注解（了解）

@Resource注解要求提供一个bean名称的属性，若该属性为空，则自动采用标注处的变量或方法名作为bean的名称。

1. @Inject注解（了解）

@Inject和@Autowired注解一样也是按类型注入匹配的bean，但没有required属性。

1. @Value注解

注入普通类型的属性，如String类型。

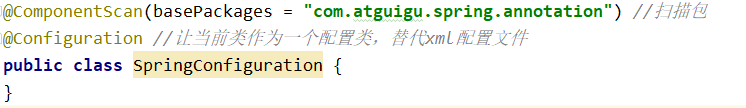
### 3.3 完全注解开发

1. @Configuration

添加了该注解的类将是一个配置类，用来代替xml配置文件

1. @ComponentScan

配置要扫描的包



1. 测试

测试时创建IOC容器的类AnnotationConfigApplicationContext

|  |
| --- |
| */\*  测试Spring的配置类 \*/* @Test **public void** testSpringConfiguration(){  *//创建IOC容器对象* ApplicationContext ioc = **new** AnnotationConfigApplicationContext(SpringConfiguration.**class**);  *//从IOC容器中获取UserService对象* UserService userService = (UserService) ioc.getBean(**"userService"**);  System.***out***.println(userService);  userService.saveUser(); } |

### 3.4 Spring集成Junit

因为Spring提供了对Junit的集成，所以可以在测试类中直接注入IOC容器中的对象，使用此功能需要导入spring-test-5.3.1.jar

1. @ContextConfiguration

指定Spring的配置文件的路径

1. @RunWith

指定Spring环境下运行Junit4的运行器



## 第4章 AOP

### 4.1 引入案例

#### 4.1.1 需求：实现计算器加减乘除功能

1. 接口

|  |
| --- |
| **package** com.atguigu.spring.log;  **public interface** Calculator {  **int** add(**int** a , **int** b);  **int** sub(**int** a , **int** b);  **int** mul(**int** a , **int** b);  **int** div(**int** a , **int** b); } |

1. 实现

|  |
| --- |
| **package** com.atguigu.spring.log;  **public class** CalculatorImpl **implements** Calculator {  @Override  **public int** add(**int** a, **int** b) {  **int** result = a + b;  **return** result;  }   @Override  **public int** sub(**int** a, **int** b) {  **int** result = a - b;  **return** result;  }   @Override  **public int** mul(**int** a, **int** b) {  **int** result = a \* b;  **return** result;  }   @Override  **public int** div(**int** a, **int** b) {  **int** result = a / b;  **return** result;  } } |

#### 4.1.2 功能扩展：在计算结果的前后打印日志

1. 在每个方法计算结果的前后打印日志

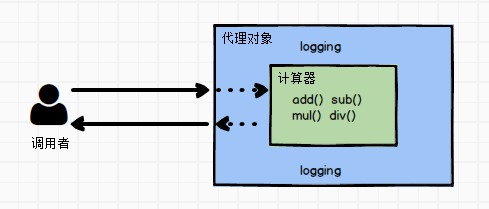
|  |
| --- |
| **package** com.atguigu.spring.log;  **public class** CalculatorImpl2 **implements** Calculator {  @Override  **public int** add(**int** a, **int** b) {  System.***out***.println(**"Logging: The method add begins with ["**+a+**","**+b+**"]"**);  **int** result = a + b;  System.***out***.println(**"Logging: The method add returns "**+result);  **return** result;  }   @Override  **public int** sub(**int** a, **int** b) {  System.***out***.println(**"Logging: The method sub begins with ["**+a+**","**+b+**"]"**);  **int** result = a - b;  System.***out***.println(**"Logging: The method sub returns "**+result);  **return** result;  }   @Override  **public int** mul(**int** a, **int** b) {  System.***out***.println(**"Logging: The method mul begins with ["**+a+**","**+b+**"]"**);  **int** result = a \* b;  System.***out***.println(**"Logging: The method mul returns "**+result);  **return** result;  }   @Override  **public int** div(**int** a, **int** b) {  System.***out***.println(**"Logging: The method div begins with ["**+a+**","**+b+**"]"**);  **int** result = a / b;  System.***out***.println(**"Logging: The method div returns "**+result);  **return** result;  } } |

1. 问题：
2. 代码混乱：越来越多的非业务需求(日志)加入后，原有的业务方法急剧膨胀。每个方法在处理核心逻辑的同时还必须兼顾其他多个关注点。
3. 代码分散: 只是为了满足这个单一需求，就不得不在多个模块（方法）里多次重复相同的日志代码。如果日志需求发生变化，必须修改所有模块。

#### 4.1.3 动态代理

1. 动态代理的原理

代理设计模式的原理：使用一个代理将原本对象包装起来，然后用该代理对象”取代”原始对象。任何对原始对象的调用都要通过代理。代理对象决定是否以及何时将方法调用转到原始对象上。



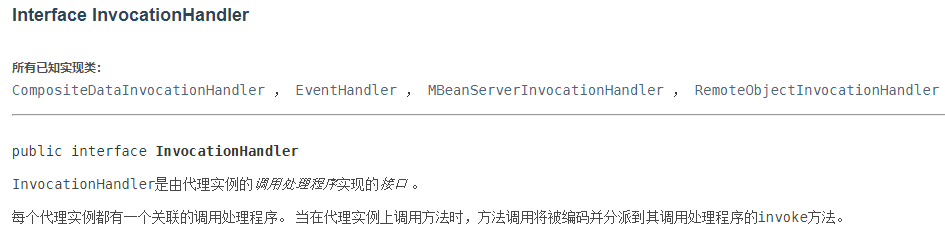
1. 动态代理的方式
   1. 基于接口实现动态代理： JDK动态代理
   2. 基于继承实现动态代理： Cglib、Javassist动态代理
2. JDK动态代理核心类



1. 核心方法



处理器接口



#### 4.1.4 使用JDK动态代理实现日志的打印

1. 创建Java类

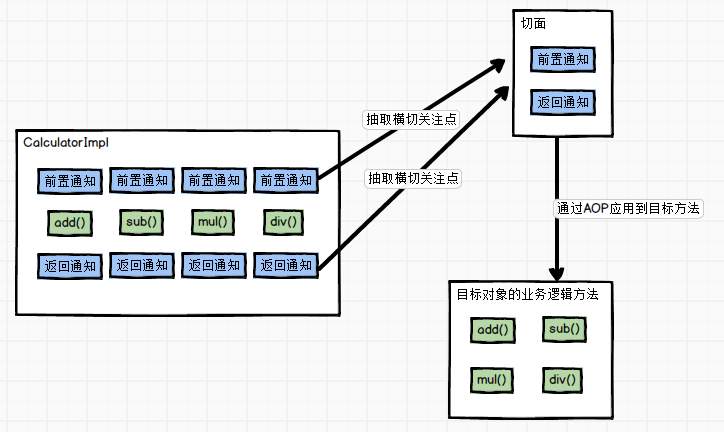
|  |
| --- |
| **package** com.atguigu.spring.log;  **import** java.lang.reflect.InvocationHandler; **import** java.lang.reflect.Method; **import** java.lang.reflect.Proxy; **import** java.util.Arrays;  **public class** LoggingProxy {   **private** Object **target**; *//被代理对象* **public** LoggingProxy(Object target) {  **this**.**target** = target;  }   **public** Object getProxy(){  *//获取代理对象的类加载器* ClassLoader classLoader = **target**.getClass().getClassLoader();  *//获取代理对象实现的接口们* Class<?>[] interfaces = **target**.getClass().getInterfaces();  Object proxy = Proxy.*newProxyInstance*(classLoader, interfaces, **new** InvocationHandler() {  */\*  invoke方法中写扩展的代码  参数说明：  proxy：代理对象  method：执行的目标方法  args：调用目标方法传入的参数  \*/* @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  *//获取方法名* String methodName = method.getName();  System.***out***.println(**"Logging: The method "**+methodName+**"begins with "**+ Arrays.*toString*(args));  *//执行目标方法* Object result = method.invoke(**target**, args);  System.***out***.println(**"Logging: The method "**+methodName+**" returns "**+result);  **return** result;  }  });  **return** proxy;  } } |

1. 测试

|  |
| --- |
| */\*  测试动态代理 \*/* @Test **public void** testProxy(){  Calculator cal = **new** CalculatorImpl();  *//获取代理对象* Calculator calculator = (Calculator) **new** LoggingProxy(cal).getProxy();  *//调用加减乘除的方法* calculator.add(10,2);  calculator.sub(10,2);  calculator.mul(10,2);  calculator.div(10,2); } |

### 4.2 AOP概述

1. AOP(Aspect-Oriented Programming，面向切面编程)：是一种新的方法论，是对传统 OOP(Object-Oriented Programming，面向对象编程)的补充。是一种通过动态代理实现程序功能扩展和统一维护的一种技术。
2. 利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。
3. AOP编程操作的主要对象是切面(aspect)，而切面用于模块化横切关注点（公共功能）。
4. AOP的好处：
   1. 每个事物逻辑位于一个位置，代码不分散，便于维护和升级
   2. 业务模块更简洁，只包含核心业务代码
   3. AOP图解：



### 4.3 AOP术语

1. 横切关注点

从每个方法中抽取出来的同一类非核心业务。

1. 切面(Aspect)

封装横切关注点信息的类，每个关注点体现为一个通知方法。

1. 通知(Advice)

切面必须要完成的各个具体工作

1. 目标(Target)

被通知的对象

1. 代理(Proxy)

向目标对象应用通知之后创建的代理对象

1. 连接点(Joinpoint)

横切关注点在程序代码中的具体位置

1. 切入点(pointcut)

定位连接点的方式。每个类的方法中都包含多个连接点，所以连接点是类中客观存在的事物。如果把连接点看作数据库中的记录，那么切入点就是查询条件——AOP可以通过切入点定位到特定的连接点。切点通过org.springframework.aop.Pointcut 接口进行描述，它使用类和方法作为连接点的查询条件。

### 4.4 AspectJ

1. 简介

Java社区里最完整最流行的AOP框架。

在Spring2.0以上版本中，可以使用基于AspectJ注解或基于XML配置的AOP。

1. @Aspect注解

在Spring中声明切面使用@Aspect注解，而且切面也需要交给IOC容器管理，即切面上也需要添加@Component注解。

当在Spring IOC容器中初始化AspectJ切面之后，Spring IOC容器就会为那些与 AspectJ切面相匹配的bean创建代理。

在AspectJ注解中，切面只是一个带有@Aspect注解的Java类，它往往要包含很多通知，通知是标注有某种注解的简单的Java方法。

1. AspectJ支持5种类型的通知注解：
   1. @Before：前置通知，在方法执行之前执行
   2. @After：后置通知，在方法执行之后执行
   3. @AfterRunning：返回通知，在方法返回结果之后执行
   4. @AfterThrowing：异常通知，在方法抛出异常之后执行
   5. @Around：环绕通知，围绕着方法执行，相当于动态代理的全过程

### 4.5 在Spring中使用AOP的步骤：

1. 在Spring核心包的基础上添加以下jar包

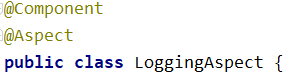
spring-aspects-5.3.1.jar

com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

1. 创建Spring的配置文件，配置自动扫描的包和AspectJ注解支持

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**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:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/aop https://www.springframework.org/schema/aop/spring-aop.xsd"**>    *<!--配置自动扫描的包-->* <**context:component-scan base-package="com.atguigu.spring.aop"**></**context:component-scan**>   *<!--开启AspectJ注解支持-->* <**aop:aspectj-autoproxy**></**aop:aspectj-autoproxy**> </**beans**> |

1. 创建一个类作为切面
   1. 在类上添加@Aspect注解将该类标识为切面
   2. 在类上添加@Component注解将该类交给IOC容器管理



1. 在切面中添加通知方法

|  |
| --- |
| **package** com.atguigu.spring.aop;  **import** org.aspectj.lang.annotation.Aspect; **import** org.aspectj.lang.annotation.Before; **import** org.springframework.stereotype.Component;  @Component @Aspect **public class** LoggingAspect {   *//前置通知，在方法执行之前执行* @Before(value = **"execution(public int com.atguigu.spring.aop.Calculator.\*(int , int ))"**)  **public void** beforeAdvice() {  System.***out***.println(**"在方法执行之前执行！"**);  } } |

1. 测试

|  |
| --- |
| **package** com.atguigu.spring.test;  **import** com.atguigu.spring.aop.Calculator; **import** org.junit.Test; **import** org.springframework.context.ApplicationContext; **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public class** AOPTest {   ApplicationContext **ioc** = **new** ClassPathXmlApplicationContext(**"beans-aop.xml"**);   */\*  测试AOP  \*/* @Test  **public void** testAOP(){  Calculator calculator = (Calculator) **ioc**.getBean(**"calculator"**);  calculator.add(10,2);  calculator.sub(10,2);  calculator.mul(10,2);  calculator.div(10,2);  } } |

测试结果：在add、sub、mul、div方法执行之前执行切面中的beforeAdvice方法

### 4.6 AOP细节

#### 4.6.1 切入点表达式

1. 作用

通过表达式的方式定位一个或多个具体的连接点。

1. 语法格式：

execution([权限修饰符] [返回值类型] [简单类名/全类名] [方法名]([参数列表]))

1. 举例说明:

|  |  |
| --- | --- |
| 表达式 | execution(\*com.atguigu.spring.aop.Calculator.\*(..)) |
| 含义 | Calculator接口中声明的所有方法。  第一个“\*”代表任意修饰符及任意返回值。  第二个“\*”代表任意方法。  “..”匹配任意数量、任意类型的参数。  若目标类、接口与该切面类在同一个包中可以省略包名。 |

|  |  |
| --- | --- |
| 表达式 | execution(public \* Calculator.\*(..)) |
| 含义 | Calculator接口中声明的所有公有方法 |

|  |  |
| --- | --- |
| 表达式 | execution(public int Calculator.\*(..)) |
| 含义 | Calculator接口中声明的返回值为int类型的公有方法 |

|  |  |
| --- | --- |
| 表达式 | execution(public int Calculator.\*(int, ..)) |
| 含义 | 第一个参数为int类型的共有方法。  “..” 匹配任意数量、任意类型的参数。 |

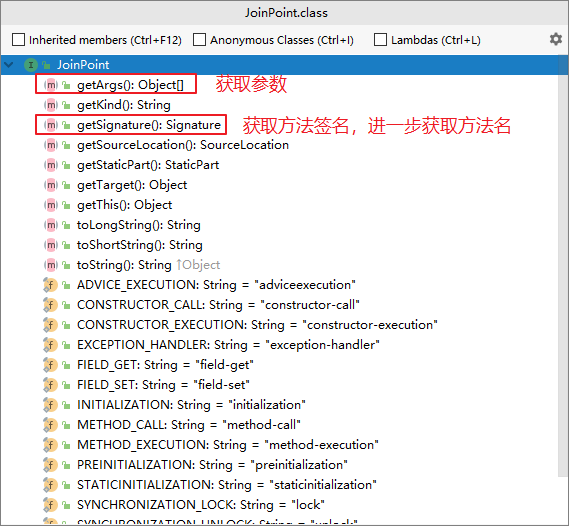
|  |  |
| --- | --- |
| 表达式 | execution(public int Calculator.\*(int, int)) |
| 含义 | 参数类型为int，int类型的方法 |

1. 在AspectJ中，切入点表达式可以通过 “&&”、“||”、“!”等操作符结合起来

|  |  |
| --- | --- |
| 表达式 | execution (\* \*.add(int,..)) || execution(\* \*.sub(int,..)) |
| 含义 | 任意类中第一个参数为int类型的add方法或sub方法 |
| 表达式 | !execution (\* \*.add(int,..)) |
| 含义 | 匹配不是任意类中第一个参数为int类型的add方法 |

#### 4.6.2 连接点细节

切入点表达式通常都会是从宏观上定位一组方法，和具体某个通知的注解结合起来就能够确定对应的连接点。那么就一个具体的连接点而言，我们可能会关心这个连接点的一些具体信息，例如：当前连接点所在方法的方法名、当前传入的参数值等等。这些信息都封装在JoinPoint接口的实例对象中。



#### 4.6.3 通知细节

1. 概述
   * 通知即在具体的连接点上要执行的操作。
   * 一个切面可以包括一个或者多个通知。
   * 通知所使用的注解的值往往是切入点表达式。
2. 前置通知
   * 使用@Before注解标识
   * 在方法执行之前执行的通知。

|  |
| --- |
| *//前置通知* @Before(value = **"execution(public int com.atguigu.spring.aop.Calculator.\*(int , int ))"**) **public void** beforeAdvice(JoinPoint joinPoint) {  *//获取方法名* String methodName = joinPoint.getSignature().getName();  *//获取参数* Object[] args = joinPoint.getArgs();  System.***out***.println(**"Logging: The method "**+methodName+**" begins with "**+ Arrays.*toString*(args)); } |

1. 后置通知
   * 使用@After注解标识
   * 后置通知是在连接点完成之后执行的，即连接点返回结果或者抛出异常的时候。

|  |
| --- |
| *//后置通知* @After(**"execution(public int Calculator.\*(int , int ))"**) **public void** afterAdvice(JoinPoint joinPoint) {  *//获取方法名* String methodName = joinPoint.getSignature().getName();  System.***out***.println(**"Logging: The method "**+methodName+**" ends"**); } |

1. 返回通知
   * 使用@AfterReturning注解标识
   * 无论连接点是正常返回还是抛出异常，后置通知都会执行。如果只想在连接点返回的时候记录日志，应使用返回通知代替后置通知。
     + 在返回通知中，通过@AfterReturning注解的returning属性获取连接点的返回值。该属性的值即为用来传入返回值的参数名称
     + 必须在通知方法的签名中添加一个同名参数。在运行时Spring AOP会通过这个参数传递返回值
     + 原始的切点表达式需要出现在pointcut属性中

|  |
| --- |
| *//返回通知* @AfterReturning(pointcut = **"execution(\* Calculator.\*(int , int ))"**,returning = **"result"**) **public void** returningAdvice(JoinPoint joinPoint , Object result){  *//获取方法名* String methodName = joinPoint.getSignature().getName();  *//获取参数* Object[] args = joinPoint.getArgs();  System.***out***.println(**"Logging: The method "**+methodName+**" returns "**+result); } |

1. 异常通知
   * 使用@AfterThrowing注解标识
   * 在异常通知中，通过@AfterThrowing注解的throwing属性获取异常信息。Throwable是所有错误和异常类的顶级父类，所以在异常通知方法可以捕获到任何错误和异常。
   * 如果只对某种特殊的异常类型感兴趣，可以将参数声明为其他异常的参数类型。然后通知就只在抛出这个类型及其子类的异常时才被执行。

|  |
| --- |
| *//异常通知* @AfterThrowing(pointcut = **"execution(\* Calculator.\*(..))"**,throwing = **"e"**) **public void** throwingAdvice(JoinPoint joinPoint,Throwable e){  *//获取方法名* String methodName = joinPoint.getSignature().getName();  System.***out***.println(**"Logging: The method "**+methodName+**" occurs "**+e); } |

1. 环绕通知
   * 通过@Around注解标识
   * 环绕通知是所有通知类型中功能最为强大的，能够全面地控制连接点，甚至可以控制是否执行连接点。
   * 对于环绕通知来说，连接点的参数类型必须是ProceedingJoinPoint。它是 JoinPoint的子接口，允许控制何时执行，是否执行连接点。
   * 在环绕通知中需要明确调用ProceedingJoinPoint的proceed()方法来执行被代理的方法。如果忘记这样做就会导致通知被执行了，但目标方法没有被执行。
   * 注意：环绕通知的方法需要返回目标方法执行之后的结果，即调用 joinPoint.proceed()的返回值，否则会出现异常

|  |
| --- |
| *//环绕通知* @Around(**"execution(\* Calculator.\*(..))"**) **public** Object aroundAdvice(ProceedingJoinPoint proceedingJoinPoint){  *//获取方法名* String methodName = proceedingJoinPoint.getSignature().getName();  *//获取参数* Object[] args = proceedingJoinPoint.getArgs();  Object result=**null**;  **try** {  *//前置通知* System.***out***.println(**"★Logging: The method "**+methodName+**" begins with "**+ Arrays.*toString*(args));  *//执行目标方法* result = proceedingJoinPoint.proceed();  *//返回通知* System.***out***.println(**"★Logging: The method "**+methodName+**" returns "**+result);  } **catch** (Throwable throwable) {  *//异常通知* System.***out***.println(**"★Logging: The method "**+methodName+**" occurs "**+throwable);  throwable.printStackTrace();  }**finally** {  *//后置通知* System.***out***.println(**"★Logging: The method "**+methodName+**" ends"**);  }  **return** result; } |

#### 4.6.4 重用切入点表达式

* 在AspectJ切面中，可以通过@Pointcut注解将一个切入点声明成简单的方法。切入点的方法体通常是空的，因为将切入点定义与应用程序逻辑混在一起是不合理的。
* 其他通知可以通过方法名称引入该切入点
* 切入点方法的访问控制符同时也控制着这个切入点的可见性。如果切入点要在多个切面中共用，最好将它们集中在一个公共的类中。在这种情况下，它们必须被声明为public。在引入这个切入点时，必须将类名也包括在内。如果类没有与这个切面放在同一个包中，还必须包含包名。



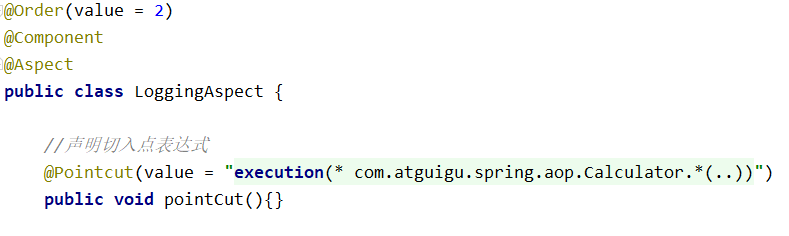
#### 4.6.5 设置切面的优先级

* 在同一个连接点上应用不止一个切面时，除非明确指定，否则它们的优先级是不确定的。
* 切面的优先级可以通过实现Ordered接口或利用@Order注解指定。
* 实现Ordered接口，getOrder()方法的返回值越小，优先级越高。
* 若使用@Order注解，通过注解的value属性指定优先级，值越小优先级越高。

验证参数的切面类：



打印日志的切面类



#### 4.6.6 通过AOP方式实现日志打印的切面类的完整内容

|  |
| --- |
| **package** com.atguigu.spring.aop;  **import** org.aspectj.lang.JoinPoint; **import** org.aspectj.lang.ProceedingJoinPoint; **import** org.aspectj.lang.annotation.\*; **import** org.springframework.core.annotation.Order; **import** org.springframework.stereotype.Component;  **import** java.util.Arrays;  @Order(value = 2) @Component @Aspect **public class** LoggingAspect {   *//声明切入点表达式* @Pointcut(value = **"execution(\* com.atguigu.spring.aop.Calculator.\*(..))"**)  **public void** pointCut(){}   *//前置通知* @Before(value = **"pointCut()"**) *//引用切入点表达式* **public void** beforeAdvice(JoinPoint joinPoint) {  *//获取方法名* String methodName = joinPoint.getSignature().getName();  *//获取参数* Object[] args = joinPoint.getArgs();  System.***out***.println(**"Logging: The method "**+methodName+**" begins with "**+ Arrays.*toString*(args));  }   *//后置通知* @After(**"execution(public int Calculator.\*(int , int ))"**)  **public void** afterAdvice(JoinPoint joinPoint) {  *//获取方法名* String methodName = joinPoint.getSignature().getName();  System.***out***.println(**"Logging: The method "**+methodName+**" ends"**);  }   *//返回通知* @AfterReturning(pointcut = **"execution(\* Calculator.\*(int , int ))"**,returning = **"result"**)  **public void** returningAdvice(JoinPoint joinPoint , Object result){  *//获取方法名* String methodName = joinPoint.getSignature().getName();  *//获取参数* Object[] args = joinPoint.getArgs();  System.***out***.println(**"Logging: The method "**+methodName+**" returns "**+result);  }   *//异常通知* @AfterThrowing(pointcut = **"execution(\* Calculator.\*(..))"**,throwing = **"e"**)  **public void** throwingAdvice(JoinPoint joinPoint,Throwable e){  *//获取方法名* String methodName = joinPoint.getSignature().getName();  System.***out***.println(**"Logging: The method "**+methodName+**" occurs "**+e);  }   *//环绕通知* @Around(**"execution(\* Calculator.\*(..))"**)  **public** Object aroundAdvice(ProceedingJoinPoint proceedingJoinPoint){  *//获取方法名* String methodName = proceedingJoinPoint.getSignature().getName();  *//获取参数* Object[] args = proceedingJoinPoint.getArgs();  Object result=**null**;  **try** {  *//前置通知* System.***out***.println(**"★Logging: The method "**+methodName+**" begins with "**+ Arrays.*toString*(args));  *//执行目标方法* result = proceedingJoinPoint.proceed();  *//返回通知* System.***out***.println(**"★Logging: The method "**+methodName+**" returns "**+result);  } **catch** (Throwable throwable) {  *//异常通知* System.***out***.println(**"★Logging: The method "**+methodName+**" occurs "**+throwable);  throwable.printStackTrace();  }**finally** {  *//后置通知* System.***out***.println(**"★Logging: The method "**+methodName+**" ends"**);  }  **return** result;  } } |

### 4.7 基于XML的方式实现AOP

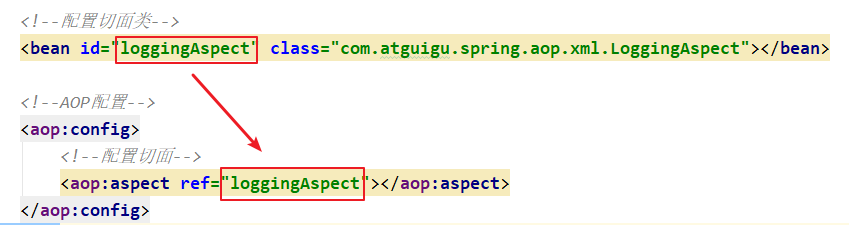
除了使用AspectJ注解声明切面，Spring也支持在bean配置文件中声明切面。这种声明是通过aop名称空间中的XML元素完成的。

正常情况下，基于注解的声明要优先于基于XML的声明。通过AspectJ注解，切面可以与AspectJ兼容，而基于XML的配置则是Spring专有的。由于AspectJ得到越来越多的 AOP框架支持，所以以注解风格编写的切面将会有更多重用的机会。

1. 配置细节

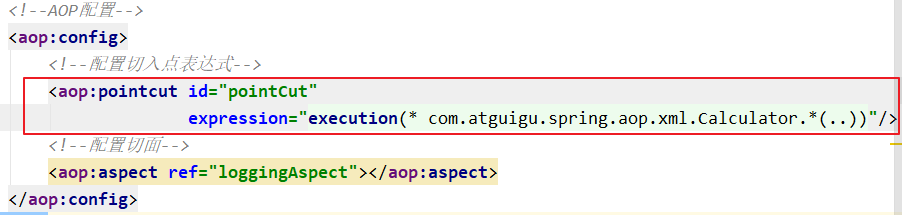
在bean配置文件中，所有的Spring AOP配置都必须定义在<aop:config>元素内部。对于每个切面而言，都要创建一个<aop:aspect>元素来为具体的切面实现引用后端bean实例。

切面bean必须有一个标识符，供<aop:aspect>元素引用。



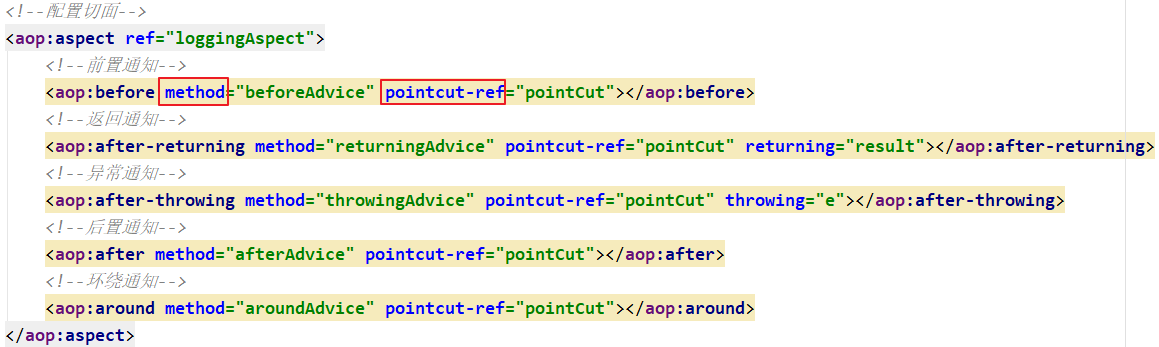
1. 声明切入点

* 切入点使用<aop:pointcut>元素声明。
* 切入点必须定义在<aop:aspect>元素下，或者直接定义在<aop:config>元素下
  + 定义在<aop:aspect>元素下：只对当前切面有效
  + 定义在<aop:config>元素下：对所有切面都有效
* 基于XML的AOP配置不允许在切入点表达式中用名称引用其他切入点。



1. 声明通知

* 在aop名称空间中，每种通知类型都对应一个特定的XML元素。
* 通知元素需要使用<pointcut-ref>来引用切入点，或用<pointcut>直接嵌入切入点表达式。
* method属性指定切面类中通知方法的名称



1. 配置文件中的完整内容

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**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.xsd http://www.springframework.org/schema/aop https://www.springframework.org/schema/aop/spring-aop.xsd"**>   *<!--配置计算器实现类-->* <**bean id="calculator" class="com.atguigu.spring.aop.xml.CalculatorImpl"**></**bean**>   *<!--配置切面类-->* <**bean id="loggingAspect" class="com.atguigu.spring.aop.xml.LoggingAspect"**></**bean**>   *<!--AOP配置-->* <**aop:config**>  *<!--配置切入点表达式-->* <**aop:pointcut id="pointCut"  expression="execution(\* com.atguigu.spring.aop.xml.Calculator.\*(..))"**/>  *<!--配置切面-->* <**aop:aspect ref="loggingAspect"**>  *<!--前置通知-->* <**aop:before method="beforeAdvice" pointcut-ref="pointCut"**></**aop:before**>  *<!--返回通知-->* <**aop:after-returning method="returningAdvice" pointcut-ref="pointCut" returning="result"**></**aop:after-returning**>  *<!--异常通知-->* <**aop:after-throwing method="throwingAdvice" pointcut-ref="pointCut" throwing="e"**></**aop:after-throwing**>  *<!--后置通知-->* <**aop:after method="afterAdvice" pointcut-ref="pointCut"**></**aop:after**>  *<!--环绕通知-->* <**aop:around method="aroundAdvice" pointcut-ref="pointCut"**></**aop:around**>  </**aop:aspect**>  </**aop:config**> </**beans**> |

## 第5章 JdbcTemplate

### 5.1 概述

为了使JDBC更加易于使用，Spring在JDBC API上定义了一个抽象层，以此建立一个JDBC存取框架。

作为Spring JDBC框架的核心，JDBC模板的设计目的是为不同类型的JDBC操作提供模板方法，通过这种方式，可以在尽可能保留灵活性的情况下，将数据库存取的工作量降到最低。

可以将Spring的JdbcTemplate看作是一个小型的轻量级持久化层框架，和我们之前使用过的DBUtils风格非常接近。

### 5.2 环境搭建

#### 5.2.1 导入jar包

1. Spring的核心包

spring-beans-5.3.1.jar

spring-context-5.3.1.jar

spring-core-5.3.1.jar

spring-expression-5.3.1.jar

spring-jcl-5.3.1.jar

spring-aop-5.3.1.jar

1. JdbcTemplate需要的jar包

spring-jdbc-5.3.1.jar

spring-orm-5.3.1.jar

spring-tx-5.3.1.jar

1. 连接数据库的驱动jar包和数据源

mysql-connector-java-5.1.37-bin.jar

druid-1.1.9.jar

1. 单元测试的jar包

junit-4.12.jar

hamcrest-core-1.3.jar

#### 5.2.2 创建数据库和表

在MySQL中执行资料中的jdbc\_template.sql文件生成对应的数据库和表。

#### 5.2.3 创建druid.properties文件

|  |
| --- |
| *#key=value* **jdbc.username**=**root jdbc.password**=**root jdbc.url**=**jdbc:mysql://localhost:3306/jdbc\_template jdbc.driverClassName**=**com.mysql.jdbc.Driver  jdbc.initialSize**=**5 jdbc.maxActive**=**10** |

#### 5.2.4 创建Spring的配置文件

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**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.xsd http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd"**>   *<!--引入外部属性文件-->* <**context:property-placeholder location="classpath:druid.properties"**></**context:property-placeholder**>   *<!--配置数据源-->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"**>  <**property name="username" value="${jdbc.username}"**></**property**>  <**property name="password" value="${jdbc.password}"**></**property**>  <**property name="url" value="${jdbc.url}"**></**property**>  <**property name="driverClassName" value="${jdbc.driverClassName}"**></**property**>  <**property name="initialSize" value="${jdbc.initialSize}"**></**property**>  <**property name="maxActive" value="${jdbc.maxActive}"**></**property**>  </**bean**>    *<!--配置JdbcTemplate-->* <**bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>  *<!--配置数据源属性-->* <**property name="dataSource" ref="dataSource"**></**property**>  </**bean**> </**beans**> |

### 5.3 操作数据库

1. 增删改
   * JdbcTemplate.update(String, Object...)

|  |
| --- |
| **public class** JdbcTemplateTest {   ApplicationContext **ioc** = **new** ClassPathXmlApplicationContext(**"beans-jdbc.xml"**);  JdbcTemplate **jdbcTemplate** = (JdbcTemplate) **ioc**.getBean(**"jdbcTemplate"**);   */\*  测试增删改  \*/* @Test  **public void** testUpdate(){  *//写sql语句* String sql = **"insert into employee(last\_name,email,salary) values(?,?,?)"**;  *//调用JdbcTemplate中的update方法* **jdbcTemplate**.update(sql,**"雷军"**,**"leijun@xiaomi.xom"**,9999.00);  } } |

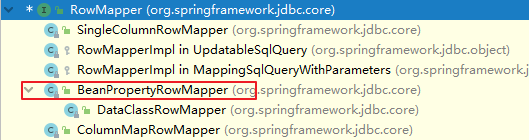
1. 批量增删改
   * JdbcTemplate.batchUpdate(String, List<Object[]>)
     + Object[]封装了SQL语句每一次执行时所需要的参数
     + List集合封装了SQL语句多次执行时的所有参数

|  |
| --- |
| */\*  测试批量增删改 \*/* @Test **public void** testBatchUpdate(){  *//写sql语句* String sql = **"insert into employee(last\_name,email,salary) values(?,?,?)"**;  *//创建一个List* List<Object[]> batchArgs = **new** ArrayList<>();  Object[] arg1 = **new** Object[]{**"李彦宏"**,**"liyanhong@baidu.com"**,8888.00};  Object[] arg2 = **new** Object[]{**"刘强东"**,**"liuqiangdong@jd.com"**,7777.00};  Object[] arg3 = **new** Object[]{**"张一鸣"**,**"zhangyiming@douyin.com"**,6666.00};  batchArgs.add(arg1);  batchArgs.add(arg2);  batchArgs.add(arg3);  *//调用JdbcTemplate中的批处理方法* **jdbcTemplate**.batchUpdate(sql,batchArgs); } |

1. 查询单一值
   * JdbcTemplate.queryForObject(String, Class, Object...)

|  |
| --- |
| */\*  测试获取一个单一值 \*/* @Test **public void** testGetSingonValue(){  *//写sql语句* String sql = **"select** *avg***(salary) from employee"**;  *//调用JdbcTemplate中的queryForObject方法* Double avgSalary = **jdbcTemplate**.queryForObject(sql, Double.**class**);  System.***out***.println(avgSalary); } |

1. 查询单行
   * JdbcTemplate.queryForObject(String, RowMapper<T>, Object...)



|  |
| --- |
| */\*  测试获取一行数据 \*/* @Test **public void** testGetOne(){  *//写sql语句* String sql = **"select id,last\_name lastName,email,salary from employee where id = ?"**;  *//创建RowMapper对象* RowMapper<Employee> rowMapper = **new** BeanPropertyRowMapper<>(Employee.**class**);  *//调用JdbcTemplate中的queryForObject方法* Employee employee = **jdbcTemplate**.queryForObject(sql, rowMapper, 1);  System.***out***.println(employee); } |

1. 查询多行
   * JdbcTemplate.query(String, RowMapper<T>, Object...)
   * RowMapper对象依然使用BeanPropertyRowMapper

|  |
| --- |
| */\*  测试获取多行数据 \*/* @Test **public void** testGetAll(){  *//写sql语句* String sql = **"select id,last\_name lastName,email,salary from employee"**;  *//创建RowMapper对象* RowMapper<Employee> rowMapper = **new** BeanPropertyRowMapper<>(Employee.**class**);  *//调用JdbcTemplate中的query方法* List<Employee> employees = **jdbcTemplate**.query(sql, rowMapper);  *//遍历* **for** (Employee employee : employees) {  System.***out***.println(employee);  } } |

### 5.4 通过注入JdbcTemplate实现Dao

JdbcTemplate类是线程安全的，所以可以在IOC容器中声明它的单个实例，并将这个实例注入到所有的Dao实例中。

1. 在Spring的配置文件中添加配置自动扫描的包

|  |
| --- |
| *<!--配置自动扫描的包-->* <**context:component-scan base-package="com.atguigu.spring.jdbc"**></**context:component-scan**> |

1. 创建EmployeeDao及实现类并注入JdbcTemplate

|  |
| --- |
| **package** com.atguigu.spring.jdbc.dao.impl;  **import** com.atguigu.spring.jdbc.bean.Employee; **import** com.atguigu.spring.jdbc.dao.EmployeeDao; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.jdbc.core.BeanPropertyRowMapper; **import** org.springframework.jdbc.core.JdbcTemplate; **import** org.springframework.jdbc.core.RowMapper; **import** org.springframework.stereotype.Repository;  @Repository(**"employeeDao"**) **public class** EmployeeDaoImpl **implements** EmployeeDao {   @Autowired  **private** JdbcTemplate **jdbcTemplate**;   @Override  **public** Employee getEmployeeById(Integer id) {  *//写sql语句* String sql = **"select id,last\_name lastName,email,salary from employee where id = ?"**;  *//创建RowMapper对象* RowMapper<Employee> rowMapper = **new** BeanPropertyRowMapper<>(Employee.**class**);  *//调用JdbcTemplate中的queryForObject方法* Employee employee = **jdbcTemplate**.queryForObject(sql, rowMapper, id);  **return** employee;  } } |

1. 测试

|  |
| --- |
| */\*  测试EmployeeDao中的的方法 \*/* @Test **public void** testEmployeeDao(){  EmployeeDao employeeDao = (EmployeeDao) **ioc**.getBean(**"employeeDao"**);  Employee employeeById = employeeDao.getEmployeeById(1);  System.***out***.println(employeeById); } |

## 第6章 声明式事务

### 6.1 事务概述

在JavaEE企业级开发的应用领域，为了保证数据的完整性和一致性，必须引入数据库事务的概念，所以事务管理是企业级应用程序开发中必不可少的技术。

事务就是一组由于逻辑上紧密关联而合并成一个整体(工作单元)的多个数据库操作，这些操作要么都执行，要么都不执行。

事务的四个特性(ACID)

1. 原子性(atomicity)：“原子”的本意是“不可再分”，事务的原子性表现为一个事务中涉及到的多个操作在逻辑上缺一不可。事务的原子性要求事务中的所有操作要么都执行，要么都不执行。
2. 一致性(consistency)：“一致”指的是数据的一致，具体是指：所有数据都处于满足业务规则的一致性状态。一致性原则要求：一个事务中不管涉及到多少个操作，都必须保证事务执行之前数据是正确的，事务执行之后数据仍然是正确的。如果一个事务在执行的过程中，其中某一个或某几个操作失败了，则必须将其他所有操作撤销，将数据恢复到事务执行之前的状态，这就是回滚。
3. 隔离性(isolation)：在应用程序实际运行过程中，事务往往是并发执行的，所以很有可能有许多事务同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。隔离性原则要求多个事务在并发执行过程中不会互相干扰。
4. 持久性(durability)：持久性原则要求事务执行完成后，对数据的修改永久的保存下来，不会因各种系统错误或其他意外情况而受到影响。通常情况下，事务对数据的修改应该被写入到持久化存储器中。

### 6.2 Spring的事务管理

#### 6.2.1 编程式事务管理

* 使用原生的JDBC API进行事务管理的步骤：
  1. 获取数据库连接Connection对象
  2. 取消事务的自动提交
  3. 执行操作
  4. 正常完成操作时手动提交事务
  5. 执行失败时回滚事务
  6. 关闭相关资源
* 评价

使用原生的JDBC API实现事务管理是所有事务管理方式的基石，同时也是最典型的编程式事务管理。编程式事务管理需要将事务管理代码嵌入到业务方法中来控制事务的提交和回滚。在使用编程的方式管理事务时，必须在每个事务操作中包含额外的事务管理代码。相对于核心业务而言，事务管理的代码显然属于非核心业务，如果多个模块 都使用同样模式的代码进行事务管理，显然会造成较大程度的代码冗余。

#### 6.2.2 声明式事务管理

Spring既支持编程式事务管理，也支持声明式的事务管理。

大多数情况下声明式事务比编程式事务管理更好：它将事务管理代码从业务方法中分离出来，以声明的方式来实现事务管理。

事务管理代码的固定模式作为一种横切关注点，可以通过AOP方法模块化，进而借助Spring AOP框架实现声明式事务管理。

Spring在不同的事务管理API之上定义了一个抽象层，通过配置的方式使其生效，从而让应用程序开发人员不必了解事务管理API的底层实现细节，就可以使用Spring的事务管理机制。

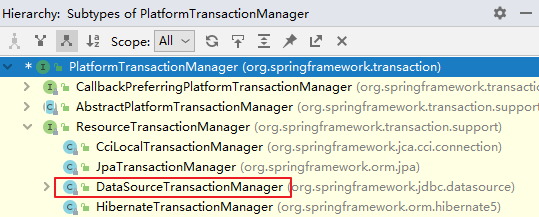
#### 6.2.3 Spring提供的事务管理器

Spring从不同的事务管理API中抽象出了一整套事务管理机制，让事务管理代码从特定的事务技术中独立出来。开发人员通过配置的方式进行事务管理，而不必了解其底层是如何实现的。

Spring的核心事务管理抽象是PlatformTransactionManager接口，它为事务管理封装了一组独立于技术的方法。无论使用Spring的哪种事务管理策略(编程式或声明式)，事务管理器都是必须的。

事务管理器可以以普通的bean的形式声明在Spring IOC容器中。

事务管理器的主要实现类：



1. DataSourceTransactionManager：在应用程序中只需要处理一个数据源，而且通过JDBC存取。
2. HibernateTransactionManager：用Hibernate框架存取数据库

### 6.3 引入案例

#### 6.3.1 创建数据库和表

在MySQL中执行资料中的spring\_transaction.sql文件生成对应的数据库和表。

#### 6.3.2 需求：购买图书

1. 接口

* BookShopDao

|  |
| --- |
| **public interface** BookShopDao {  *//根据图书的书号获取图书的价格* Double getBookPrice(String isbn);  *//根据图书的书号更新图书库存，一次只能买一本* **void** updateBookStock(String isbn);  *//根据用户的id和图书的价格更新用户的余额* **void** updateUserBalance(**int** userId , Double bookPrice); } |

* BookShopService

|  |
| --- |
| **public interface** BookShopService {  *//购买图书* **void** purchase(**int** userId , String isbn); } |

1. 创建Spring的配置文件配置数据源和JdbcTemplate

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**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.xsd http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd"**>   *<!--配置自动扫描的包-->* <**context:component-scan base-package="com.atguigu.spring.tx"**></**context:component-scan**>   *<!--引入外部属性文件-->* <**context:property-placeholder location="classpath:druid.properties"**></**context:property-placeholder**>   *<!--配置数据源-->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"**>  <**property name="username" value="${jdbc.username}"**></**property**>  <**property name="password" value="${jdbc.password}"**></**property**>  <**property name="url" value="${jdbc.url}"**></**property**>  <**property name="driverClassName" value="${jdbc.driverClassName}"**></**property**>  <**property name="initialSize" value="${jdbc.initialSize}"**></**property**>  <**property name="maxActive" value="${jdbc.maxActive}"**></**property**>  </**bean**>   *<!--配置JdbcTemplate-->* <**bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>  *<!--配置数据源属性-->* <**property name="dataSource" ref="dataSource"**></**property**>  </**bean**> </**beans**> |

1. 创建实现类

* BookShopDaoImpl

|  |
| --- |
| **package** com.atguigu.spring.tx.dao.impl;  **import** com.atguigu.spring.tx.dao.BookShopDao; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.jdbc.core.JdbcTemplate; **import** org.springframework.stereotype.Repository;  @Repository(**"bookShopDao"**) **public class** BookShopDaoImpl **implements** BookShopDao {   @Autowired  **private** JdbcTemplate **jdbcTemplate**;   @Override  **public** Double getBookPrice(String isbn) {  *//写sql语句* String sql = **"select price from book where isbn = ?"**;  *//调用JdbcTemplate中的queryForObject方法* Double price = **jdbcTemplate**.queryForObject(sql, Double.**class**,isbn);  **return** price;  }   @Override  **public void** updateBookStock(String isbn) {  *//写sql语句* String sql = **"update book set stock = stock - 1 where isbn = ?"**;  *//调用JdbcTemplate中的update方法* **jdbcTemplate**.update(sql,isbn);  }   @Override  **public void** updateUserBalance(**int** userId, Double bookPrice) {  *//写sql语句* String sql = **"update account set balance = balance - ? where id = ?"**;  *//调用JdbcTemplate中的update方法* **jdbcTemplate**.update(sql,bookPrice,userId);  } } |

* BookShopServiceImpl

|  |
| --- |
| **package** com.atguigu.spring.tx.service.impl;  **import** com.atguigu.spring.tx.dao.BookShopDao; **import** com.atguigu.spring.tx.service.BookShopService; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.stereotype.Service;  @Service(**"bookShopService"**) **public class** BookShopServiceImpl **implements** BookShopService {   @Autowired  **private** BookShopDao **bookShopDao**;   @Override  **public void** purchase(**int** userId, String isbn) {  *//获取图书的价格* Double bookPrice = **bookShopDao**.getBookPrice(isbn);  *//更新图书的库存* **bookShopDao**.updateBookStock(isbn);  *//更新用户账户的余额* **bookShopDao**.updateUserBalance(userId,bookPrice);  } } |

1. 测试

|  |
| --- |
| **package** com.atguigu.spring.tx.test;  **import** com.atguigu.spring.tx.dao.BookShopDao; **import** com.atguigu.spring.tx.service.BookShopService; **import** org.junit.Test; **import** org.springframework.context.ApplicationContext; **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public class** TransactionTest {   ApplicationContext **ioc** = **new** ClassPathXmlApplicationContext(**"beans-tx.xml"**);   */\*  测试BookShopDao中的方法  \*/* @Test  **public void** testBookShopDao(){  BookShopDao bookShopDao = (BookShopDao) **ioc**.getBean(**"bookShopDao"**);  *//获取图书价格* Double bookPrice = bookShopDao.getBookPrice(**"1001"**);  System.***out***.println(bookPrice);  *//更新图书库存* bookShopDao.updateBookStock(**"1001"**);  *//更新用户账户余额* bookShopDao.updateUserBalance(1,bookPrice);  }   */\*  测试BookShopServie中的方法  \*/* @Test  **public void** testBookShopService(){  BookShopService bookShopService = (BookShopService) **ioc**.getBean(**"bookShopService"**);  bookShopService.purchase(1,**"1001"**);  } } |

1. 出现的问题

当用户账户余额不足时会抛出异常，但是图书的库存却会减去一本！

1. 解决方案

添加事务

### 6.4 基于注解方式的声明式事务

#### 6.4.1 购买图书案例中添加Spring的声明式事务

1. 导入AOP相关的jar包

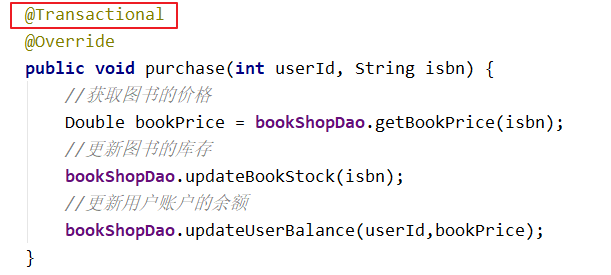
spring-aspects-5.3.1.jar

com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

1. 在Spring的配置文件中添加以下配置

|  |
| --- |
| *<!--配置事务管理器-->* <**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  *<!--配置数据源属性-->* <**property name="dataSource" ref="dataSource"**></**property**>  </**bean**>   *<!--开启事务注解支持  当事务管理器的id是transactionManager时，可以省略指定transaction-manager属性  --> <!-- <tx:annotation-driven transaction-manager="transactionManager"></tx:annotation-driven>-->* <**tx:annotation-driven**></**tx:annotation-driven**> |

1. 在需要添加事务的方法上添加@Transactional注解



#### 6.4.2 事务的传播行为

1. 简介

当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。

事务传播属性可以在@Transactional注解的propagation属性中定义。

1. Spring的7种传播行为

|  |  |
| --- | --- |
| 传播属性 | 描述 |
| REQUIRED | 如果有事务在运行，当前的方法就在这个事务内运行；否则就启动一个新的事务，并在自己的事务内运行。 |
| REQUIRES\_NEW | 当前的方法必须启动新事务，并在自己的事务内运行；如果有事务正在运行，应该将它挂起。 |
| SUPPORTS | 如果有事务在运行，当前的方法就在这个事务内运行，否则可以不运行在事务中。 |
| NOT\_SUPPORTED | 当前的方法不应该运行在事务中，如果有运行的事务将它挂起 |
| MANDATORY | 当前的方法必须运行在事务中，如果没有正在运行的事务就抛出异常。 |
| NEVER | 当前的方法不应该运行在事务中，如果有正在运行的事务就抛出异常。 |
| NESTED | 如果有事务正在运行，当前的方法就应该在这个事务的嵌套事务内运行，否则就启动一个新的事务，并在它自己的事务内运行。 |

1. 传播行为测试

* 创建Cashier接口

|  |
| --- |
| **package** com.atguigu.spring.tx.service;  **import** java.util.List;  **public interface** Cashier {  *//为多本书结账* **void** checkout(**int** userId , List<String> ids); } |

* 创建CashierImpl实现类

|  |
| --- |
| **package** com.atguigu.spring.tx.service.impl;  **import** com.atguigu.spring.tx.service.BookShopService; **import** com.atguigu.spring.tx.service.Cashier; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.stereotype.Service; **import** org.springframework.transaction.annotation.Transactional;  **import** java.util.List;  @Service(**"cashier"**) **public class** CashierImpl **implements** Cashier {   @Autowired  **private** BookShopService **bookShopService**;   @Transactional  @Override  **public void** checkout(**int** userId, List<String> ids) {  *//遍历所有图书的id* **for** (String id : ids) {  *//调用BookShopService中购买图书的方法* **bookShopService**.purchase(userId,id);  }  } } |

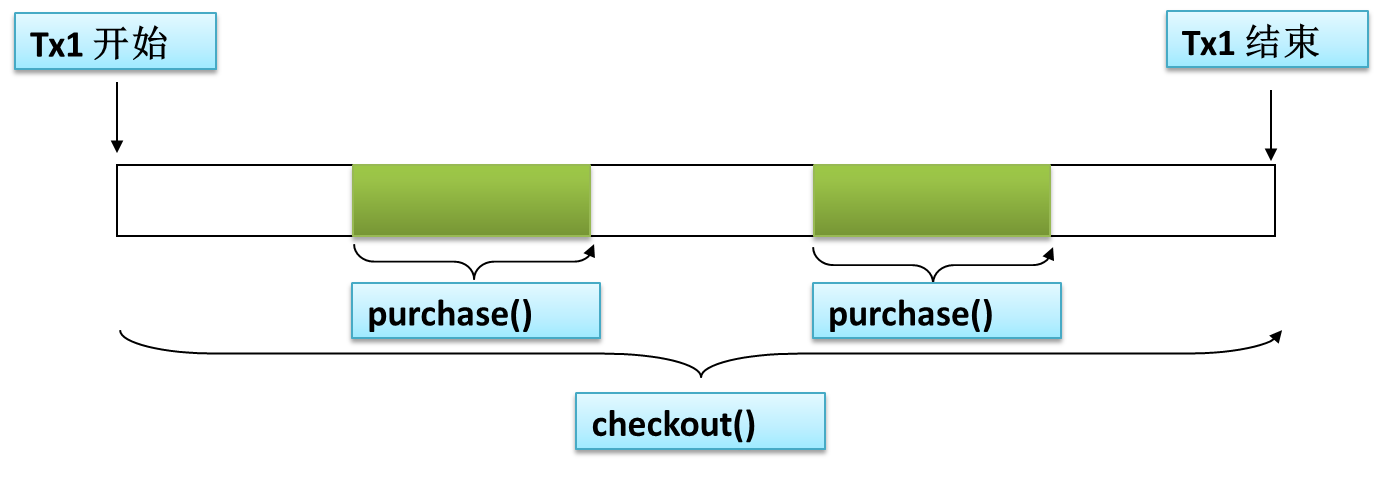
此时，purchase方法就运行在了一个开启了事务的checkout方法中

* 测试

|  |
| --- |
| */\*  测试去结账里的方法 \*/* @Test **public void** testCashier(){  Cashier cashier = (Cashier) **ioc**.getBean(**"cashier"**);  *//创建List* List<String> ids = **new** ArrayList<>();  ids.add(**"1001"**);  ids.add(**"1002"**);  cashier.checkout(1,ids); } |

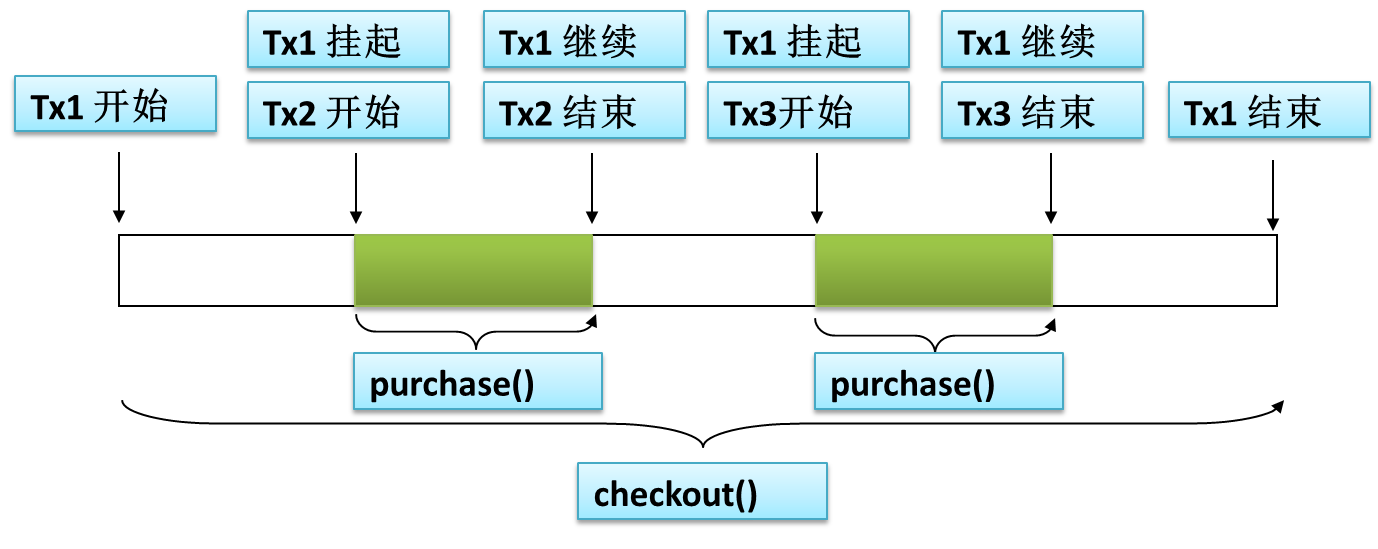
* 常用传播行为说明
  + REQUIRED

当bookService的purchase()方法被另一个事务方法checkout()调用时，它默认会在现有的事务内运行。这个默认的传播行为就是REQUIRED。因此在checkout()方法的开始和终止边界内只有一个事务。这个事务只在checkout()方法结束的时候被提交，结果用户一本书都买不了。



* + REQUIRES\_NEW

表示该方法必须启动一个新事务，并在自己的事务内运行。如果有事务在运行，就应该先挂起它。



#### 6.4.3 事务的隔离级别

1. 数据库事务并发问题
   1. 脏读

①Transaction01将某条记录的AGE值从20修改为30。

②Transaction02读取了Transaction01更新后的值：30。

③Transaction01回滚，AGE值恢复到了20。

④Transaction02读取到的30就是一个无效的值。

简单来说就是你读到了别人更新但未提交的数据。

* 1. 不可重复度

①Transaction01读取了AGE值为20。

②Transaction02将AGE值修改为30并提交。

③Transaction01再次读取AGE值为30，和第一次读取不一致。

简单来说就是你两次读取的值不可能重复。

* 1. 幻读

①Transaction01读取了STUDENT表中的一部分数据。

②Transaction02向STUDENT表中插入了新的行。

③Transaction01读取了STUDENT表时，多出了一些行。

简单来说就是你两次读取的表中的记录不一样，好像出现幻觉似的。

1. 隔离级别

数据库系统必须具有隔离并发运行各个事务的能力，使它们不会相互影响，避免各种并发问题。一个事务与其他事务隔离的程度称为隔离级别。SQL标准中规定了多种事务隔离级别，不同隔离级别对应不同的干扰程度，隔离级别越高，数据一致性就越好，但并发性越弱。

隔离级别一共有四种：

* 1. 读未提交：READ UNCOMMITTED

允许Transaction01读取Transaction02未提交的修改。

* 1. 读已提交：READ COMMITTED、

要求Transaction01只能读取Transaction02已提交的修改。

* 1. 可重复读：REPEATABLE READ

确保Transaction01可以多次从一个字段中读取到相同的值，即Transaction01执行期间禁止其它事务对这个字段进行更新。

* 1. 串行化：SERIALIZABLE

确保Transaction01可以多次从一个表中读取到相同的行，在Transaction01执行期间，禁止其它事务对这个表进行添加、更新、删除操作。可以避免任何并发问题，但性能十分低下。

* 1. 各个隔离级别解决并发问题的能力见下表

|  |  |  |  |
| --- | --- | --- | --- |
|  | 脏读 | 不可重复读 | 幻读 |
| READ UNCOMMITTED | 有 | 有 | 有 |
| READ COMMITTED | 无 | 有 | 有 |
| REPEATABLE READ | 无 | 无 | 有 |
| SERIALIZABLE | 无 | 无 | 无 |

* 1. 各种数据库产品对事务隔离级别的支持程度

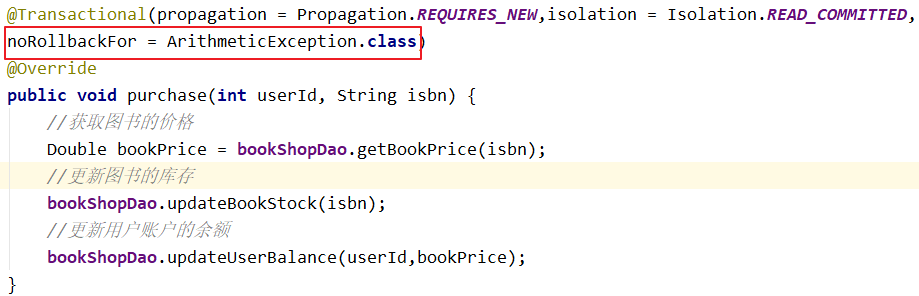
|  |  |  |
| --- | --- | --- |
|  | Oracle | MySQL |
| READ UNCOMMITTED | × | √ |
| READ COMMITTED | √(默认) | √ |
| REPEATABLE READ | × | √(默认) |
| SERIALIZABLE | √ | √ |

1. 在Spring中指定事务隔离级别

在@Transactional的isolation属性中设置隔离级别

#### 6.4.4 @Transactional注解中的其他属性

1. 触发事务回滚的异常
   1. 默认情况下，捕获到RuntimeException或Error时回滚，而捕获到编译时异常不回滚。
   2. @Transactional注解中设置回滚的属性
      1. rollbackFor或rollbackForClassName属性：指定遇到时必须进行回滚的异常类型，可以为多个。
      2. noRollbackFor或noRollbackForClassName属性：指定遇到时不回滚的异常类型，可以为多个。



1. 事务的超时时间

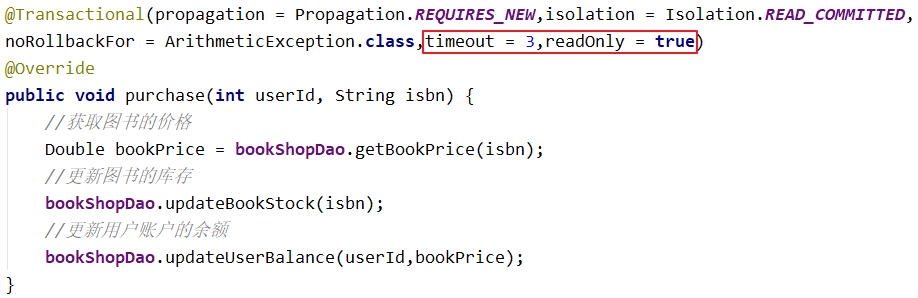
由于事务可以在行和表上获得锁，因此长事务会占用资源，并对整体性能产生影响。

timeout超时事务属性可以设置事务在强制回滚之前可以保持多久。这样可以防止长期运行的事务占用资源。

1. 只读属性

如果一个事务只读取数据但不做修改，数据库引擎可以对这个事务进行优化。

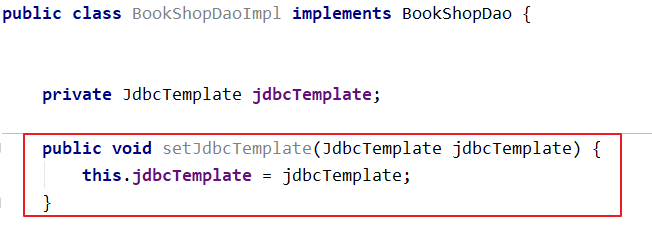
readOnly只读事务属性可以设置这个事务只读取数据但不更新数据, 这样可以帮助数据库引擎优化事务。



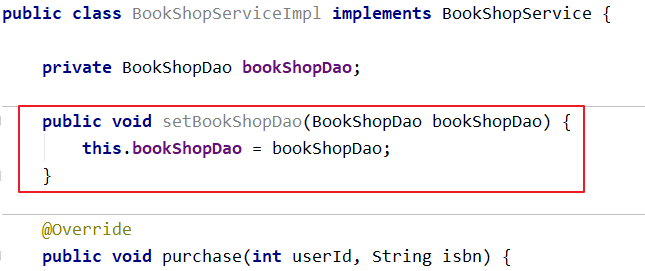
### 6.5 基于XML方式的声明式事务

#### 6.5.1 修改实现类

* BookShopDaoImpl



* BookShopServiceImpl



#### 6.5.2 重新创建一个Spring的配置文件

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**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:aop="http://www.springframework.org/schema/aop" xmlns:tx="http://www.springframework.org/schema/tx"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/aop https://www.springframework.org/schema/aop/spring-aop.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd"**>   *<!--引入外部属性文件-->* <**context:property-placeholder location="classpath:druid.properties"**></**context:property-placeholder**>   *<!--配置数据源-->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"**>  <**property name="username" value="${jdbc.username}"**></**property**>  <**property name="password" value="${jdbc.password}"**></**property**>  <**property name="url" value="${jdbc.url}"**></**property**>  <**property name="driverClassName" value="${jdbc.driverClassName}"**></**property**>  <**property name="initialSize" value="${jdbc.initialSize}"**></**property**>  <**property name="maxActive" value="${jdbc.maxActive}"**></**property**>  </**bean**>   *<!--配置JdbcTemplate-->* <**bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>  *<!--配置数据源属性-->* <**property name="dataSource" ref="dataSource"**></**property**>  </**bean**>   *<!--配置BookShopDaoImpl-->* <**bean id="bookShopDao" class="com.atguigu.spring.tx.xml.BookShopDaoImpl"**>  <**property name="jdbcTemplate" ref="jdbcTemplate"**></**property**>  </**bean**>   *<!--配置BookShopServiceImpl-->* <**bean id="bookShopService" class="com.atguigu.spring.tx.xml.BookShopServiceImpl"**>  <**property name="bookShopDao" ref="bookShopDao"**></**property**>  </**bean**>   *<!--配置事务管理器-->* <**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  *<!--配置数据源属性-->* <**property name="dataSource" ref="dataSource"**></**property**>  </**bean**>   *<!--配置声明式事务-->* <**tx:advice id="tx" transaction-manager="transactionManager"**>  *<!--设置添加事务的方法-->* <**tx:attributes**>  *<!--设置查询的方法的只读属性为true-->* <**tx:method name="find\*" read-only="true"**/>  <**tx:method name="get\*" read-only="true"**/>  <**tx:method name="purchase" propagation="REQUIRES\_NEW" isolation="READ\_COMMITTED"**></**tx:method**>  </**tx:attributes**>  </**tx:advice**>   *<!--AOP配置-->* <**aop:config**>  *<!--配置切入点表达式-->* <**aop:pointcut id="pointCut"  expression="execution(\* com.atguigu.spring.tx.xml.BookShopServiceImpl.purchase(..))"**/>  *<!--将事务方法和切入点表达式关联起来-->* <**aop:advisor advice-ref="tx" pointcut-ref="pointCut"**></**aop:advisor**>  </**aop:config**>  </**beans**> |

#### 6.5.3 测试

|  |
| --- |
| **public class** Transaction\_XMLTest {   ApplicationContext **ioc** = **new** ClassPathXmlApplicationContext(**"beans-tx-xml.xml"**);   */\*  测试BookShopServie中的方法  \*/* @Test  **public void** testBookShopService(){  BookShopService bookShopService = (BookShopService) **ioc**.getBean(**"bookShopService"**);  bookShopService.purchase(1,**"1001"**);  } } |

注意：由于是从基于注解的方式配置声明式事务哪儿复制的接口和类，导包时一定不要导错，否则会出现类型转换异常。

## 第7章 Spring5的新特性

### 7.1 新特性简介

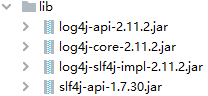
1. 整个 Spring5 框架的代码基于 Java8，运行时兼容 JDK9，许多不建议使用的类和方法在代码库中删除。
2. Spring 5.0 框架自带了通用的日志封装。
3. 支持@Nullable 注解。
4. Spring5 支持整合 JUnit5。

### 7.2 Spring 5.0 框架自带了通用的日志封装

Spring5 已经移除 Log4jConfigListener，官方建议使用 Log4j2。

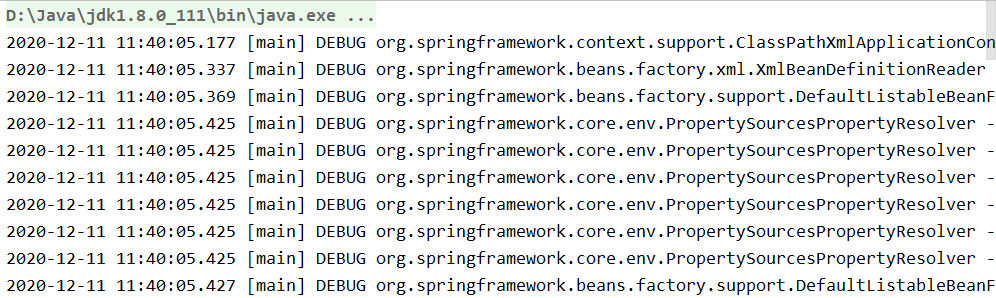
Spring5 框架整合 Log4j2步骤：

1. 导入以下jar包



1. 创建log4j2.xml配置文件

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?> <!--日志级别以及优先级排序: OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE > ALL --> <!--Configuration后面的status用于设置log4j2自身内部的信息输出，可以不设置，当设置成trace时，可以看到log4j2内部各种详细输出-->* <**configuration status="INFO"**>  *<!--先定义所有的appender-->* <**appenders**>  *<!--输出日志信息到控制台-->* <**console name="Console" target="SYSTEM\_OUT"**>  *<!--控制日志输出的格式-->* <**PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"**/>  </**console**>  </**appenders**>  *<!--然后定义logger，只有定义了logger并引入的appender，appender才会生效-->  <!--root：用于指定项目的根日志，如果没有单独指定Logger，则会使用root作为默认的日志输出-->* <**loggers**>  <**root level="DEBUG"**>  <**appender-ref ref="Console"**/>  </**root**>  </**loggers**> </**configuration**> |

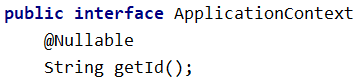
1. 测试

### 7.3 支持@Nullable 注解

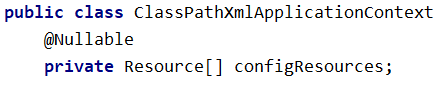
@Nullable 注解可以使用在方法上面，属性上面，参数前面，表示方法返回可以为空，属性值可以为空，参数值可以为空。

此注解通常用来消除NullPointerException

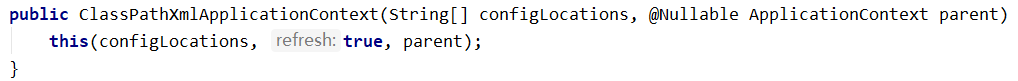
1. 注解用在方法上面，方法返回值可以为空



1. 注解用在属性上面，属性可以为空



1. 注解用在参数前面，参数可以为空



### 7.4 整合 JUnit5

1. 导入jar包
   1. Spring5的测试包

spring-test-5.3.1.jar

* 1. Junit5相关jar包

junit-jupiter-api-5.7.0.jar

junit-jupiter-api-5.7.0.jar

junit-platform-commons-1.7.0.jar

opentest4j-1.2.0.jar

1. 创建测试类
   1. 使用@ExtendWith注解和@ContextConfiguration注解

|  |
| --- |
| **package** com.atguigu.spring.tx.test;  **import** com.atguigu.spring.tx.dao.BookShopDao; **import** org.junit.jupiter.api.Test; **import** org.junit.jupiter.api.extension.ExtendWith; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.test.context.ContextConfiguration; **import** org.springframework.test.context.junit.jupiter.SpringExtension;  @ExtendWith(SpringExtension.**class**) @ContextConfiguration(locations = **"classpath:beans-tx.xml"**) **public class** Junit5Test {   @Autowired  **private** BookShopDao **bookShopDao**;   */\*  测试Spring整合Junit5  \*/* @Test  **public void** testJunit5(){  System.***out***.println(**bookShopDao**);  } } |

* 1. 使用复合注解@SpringJUnitConfig

|  |
| --- |
| **package** com.atguigu.spring.tx.test;  **import** com.atguigu.spring.tx.dao.BookShopDao; **import** org.junit.jupiter.api.Test; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.test.context.junit.jupiter.SpringJUnitConfig;  @SpringJUnitConfig(locations = **"classpath:beans-tx.xml"**) **public class** Junit5Test {   @Autowired  **private** BookShopDao **bookShopDao**;   */\*  测试Spring整合Junit5  \*/* @Test  **public void** testJunit5(){  System.***out***.println(**bookShopDao**);  } } |