**Spring框架**

尚硅谷java研究院

版本：V 1.0

# 第1章 Spring概述

## 1.1 Spring概述

1. Spring是一个开源框架
2. Spring为简化企业级开发而生，使用Spring，JavaBean就可以实现很多以前要靠EJB才能实现的功能。同样的功能，在EJB中要通过繁琐的配置和复杂的代码才能够实现，而在Spring中却非常的优雅和简洁。
3. Spring是一个**IOC**(DI)和**AOP**容器框架。
4. 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

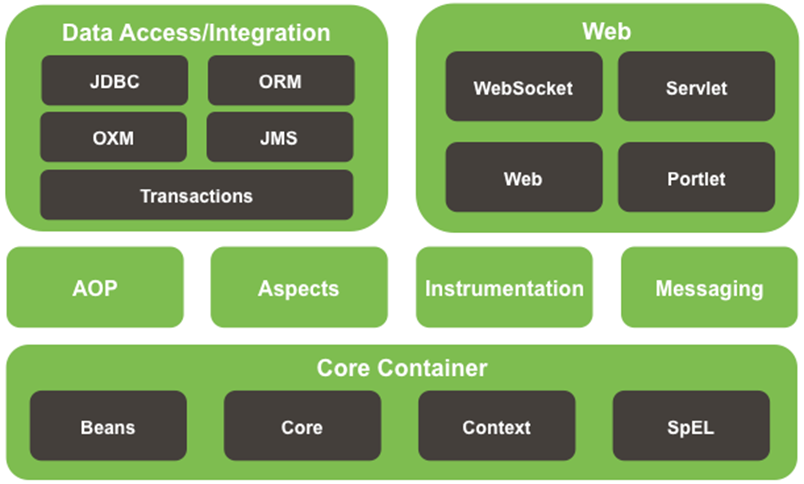
② **依赖注入**：DI——Dependency Injection，反转控制(IOC)最经典的实现。

③ **面向切面编程**：Aspect Oriented Programming——AOP

④ **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

**⑤ 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。

1. **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Spring JDBC）。
2. Spring模块



## 1.2 安装Spring插件【了解】

1. 插件包：springsource-tool-suite-3.4.0.RELEASE-e4.3.1-updatesite.zip
2. 操作步骤：参照《[尚硅谷]\_参考资料：安装Springtools插件.doc》

## 1.3 搭建Spring运行时环境

1. 加入JAR包

① Spring自身JAR包：spring-framework-4.0.0.RELEASE\libs目录下

spring-beans-4.0.0.RELEASE.jar

spring-context-4.0.0.RELEASE.jar

spring-core-4.0.0.RELEASE.jar

spring-expression-4.0.0.RELEASE.jar

② commons-logging-1.1.1.jar

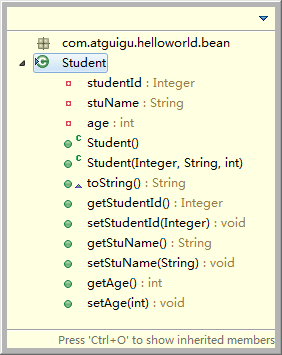
1. 在Spring Tool Suite工具中通过如下步骤创建Spring的配置文件

① File->New->Spring Bean Configuration File

② 为文件取名字 例如：applicationContext.xml

## 1.4 HelloWorld

1. 目标：使用Spring创建对象，为属性赋值
2. 创建Student类



1. 创建Spring配置文件

|  |
| --- |
| <!-- 使用bean元素定义一个由IOC容器创建的对象 -->  <!-- class属性指定用于创建bean的全类名 -->  <!-- id属性指定用于引用bean实例的标识 -->  <bean id=*"student"* class=*"com.atguigu.helloworld.bean.Student"*>  <!-- 使用property子元素为bean的属性赋值 -->  <property name=*"studentId"* value=*"1001"*/>  <property name=*"stuName"* value=*"Tom2015"*/>  <property name=*"age"* value=*"20"*/>  </bean> |

1. 测试：通过Spring的IOC容器创建Student类实例

|  |
| --- |
| //1.创建IOC容器对象  ApplicationContext iocContainer =  **new** ClassPathXmlApplicationContext("helloworld.xml");  //2.根据id值获取bean实例对象  Student student = (Student) iocContainer.getBean("student");  //3.打印bean  System.*out*.println(student); |

# 第2章 IOC容器和Bean的配置

## 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

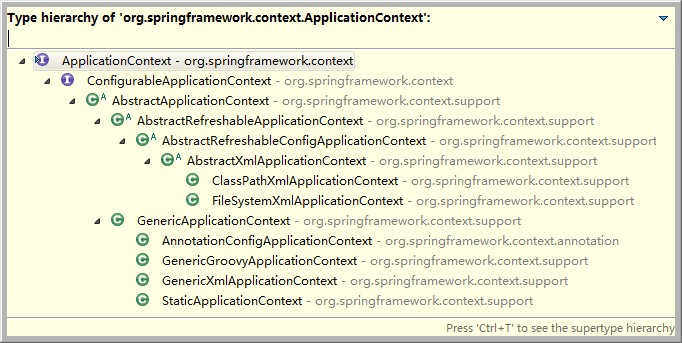
IOC的另一种表述方式：即组件以一些预先定义好的方式(例如：setter 方法)接受来自于容器的资源注入。相对于IOC而言，这种表述更直接。

### 2.1.3 IOC容器在Spring中的实现

1）在通过IOC容器读取Bean的实例之前，需要先将IOC容器本身实例化。

2）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内部的基础设施，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. 在初始化时就创建单例的bean，也可以通过配置的方式指定创建的Bean是多实例的。

### 2.1.5 ConfigurableApplicationContext

1. 是ApplicationContext的子接口，包含一些扩展方法
2. refresh()和close()让ApplicationContext具有启动、关闭和刷新上下文的能力。

### 2.1.6 WebApplicationContext

1. 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 通过类型获取bean

1. 从IOC容器中获取bean时，除了通过id值获取，还可以通过bean的类型获取。但如果同一个类型的bean在XML文件中配置了多个，则获取时会抛出异常，所以同一个类型的bean在容器中必须是唯一的。

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

1. 或者可以使用另外一个重载的方法，同时指定bean的id值和类型

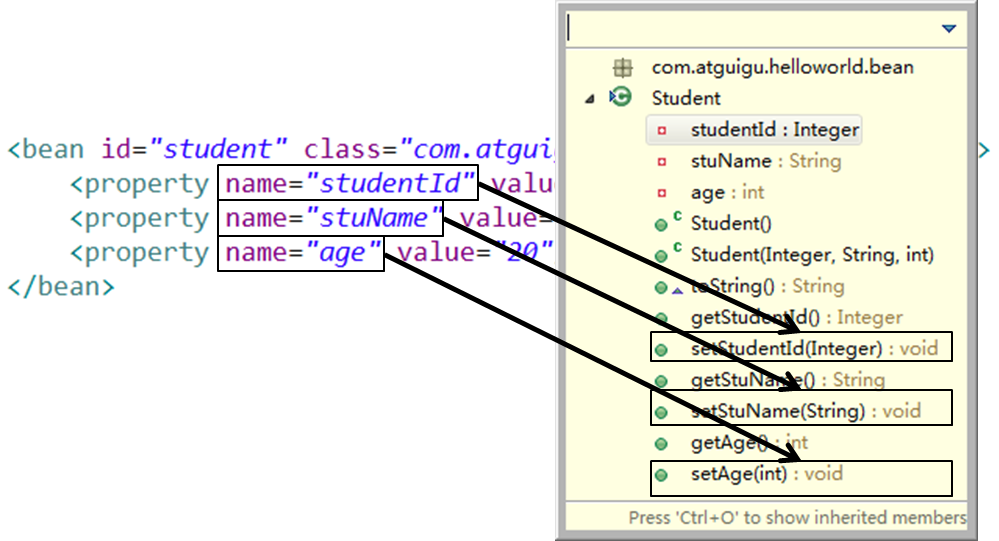
|  |
| --- |
| HelloWorld helloWorld = cxt.getBean(“helloWorld”,HelloWorld. **class**); |

## 2.3 给bean的属性赋值

### 2.3.1 依赖注入的方式

#### 1. 通过bean的setXxx()方法赋值

Hello World中使用的就是这种方式



#### 2. 通过bean的构造器赋值

1. Spring自动匹配合适的构造器

|  |
| --- |
| <bean id=*"book"* class=*"com.atguigu.spring.bean.Book"* >             <constructor-arg value= *"10010"*/>             <constructor-arg value= *"Book01"*/>             <constructor-arg value= *"Author01"*/>             <constructor-arg value= *"20.2"*/>       </bean > |

1. 通过索引值指定参数位置

|  |
| --- |
| <bean id=*"book"* class=*"com.atguigu.spring.bean.Book"* >             <constructor-arg value= *"10010"* index =*"0"*/>             <constructor-arg value= *"Book01"* index =*"1"*/>             <constructor-arg value= *"Author01"* index =*"2"*/>             <constructor-arg value= *"20.2"* index =*"3"*/>       </bean > |

1. 通过类型区分重载的构造器

|  |
| --- |
| <bean id=*"book"* class=*"com.atguigu.spring.bean.Book"* >        <constructor-arg value= *"10010"* index =*"0"* type=*"java.lang.Integer"* />        <constructor-arg value= *"Book01"* index =*"1"* type=*"java.lang.String"* />        <constructor-arg value= *"Author01"* index =*"2"* type=*"java.lang.String"* />        <constructor-arg value= *"20.2"* index =*"3"* type=*"java.lang.Double"* />  </bean > |

### 2.3.2 p名称空间

为了简化XML文件的配置，越来越多的XML文件采用属性而非子元素配置信息。Spring 从2.5版本开始引入了一个新的p命名空间，可以通过<bean>元素属性的方式配置Bean 的属性。

使用p命名空间后，基于XML的配置方式将进一步简化。

|  |
| --- |
| <bean  id=*"studentSuper"*  class=*"com.atguigu.helloworld.bean.Student"*  p:studentId=*"2002"* p:stuName=*"Jerry2016"* p:age=*"18"* /> |

### 2.3.3 可以使用的值

#### 1. 字面量

1. 可以使用字符串表示的值，可以通过value属性或value子节点的方式指定
2. 基本数据类型及其封装类、String等类型都可以采取字面值注入的方式
3. 若字面值中包含特殊字符，可以使用<![CDATA[]]>把字面值包裹起来

#### 2. null值

|  |
| --- |
| <bean class=*"com.atguigu.spring.bean.Book"* id=*"bookNull"* >             <property name= *"bookId"* value =*"2000"*/>             <property name= *"bookName"*>                 <null/>             </property>             <property name= *"author"* value =*"nullAuthor"*/>             <property name= *"price"* value =*"50"*/>       </bean > |

#### 3. 给bean的级联属性赋值

|  |
| --- |
| <bean id="action" class="com.atguigu.spring.ref.Action">  <property name="service" ref="service"/>  <!-- 设置级联属性(了解) -->  <property name="service.dao.dataSource" value="DBCP"/>  </bean> |

#### 4. 外部已声明的bean

|  |
| --- |
| <bean id=*"shop"* class=*"com.atguigu.spring.bean.Shop"* >             <property name= *"book"* ref =*"book"*/>       </bean > |

#### 5. 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，不需要设置任何id或name属性

内部bean不能使用在任何其他地方

|  |
| --- |
| <bean id=*"shop2"* class=*"com.atguigu.spring.bean.Shop"* >      <property name= *"book"*>          <bean class= *"com.atguigu.spring.bean.Book"* >             <property name= *"bookId"* value =*"1000"*/>             <property name= *"bookName"* value=*"innerBook"* />             <property name= *"author"* value=*"innerAuthor"* />             <property name= *"price"* value =*"50"*/>          </bean>      </property>  </bean > |

## 2.4 集合属性

在Spring中可以通过一组内置的XML标签来配置集合属性，例如：<list>，<set>或<map>。

### 2.4.1 数组和List

配置java.util.List类型的属性，需要指定<list>标签，在标签里包含一些元素。这些标签 可以通过<value>指定简单的常量值，通过<ref>指定对其他Bean的引用。通过<bean> 指定内置bean定义。通过<null/>指定空元素。甚至可以内嵌其他集合。

数组的定义和List一样，都使用<list>元素。

配置java.util.Set需要使用<set>标签，定义的方法与List一样。

|  |
| --- |
| <bean id=*"shop"* class=*"com.atguigu.spring.bean.Shop"* >             <property name= *"categoryList"*>                 <!-- 以字面量为值的List集合 -->                 <list>                      <value> 历史</value >                      <value> 军事</value >                 </list>             </property>             <property name= *"bookList"*>                 <!-- 以bean的引用为值的List集合 -->                 <list>                      <ref bean= *"book01"*/>                      <ref bean= *"book02"*/>                 </list>             </property>       </bean > |

### 2.4.2 Map

Java.util.Map通过<map>标签定义，<map>标签里可以使用多个<entry>作为子标签。每个条目包含一个键和一个值。

必须在<key>标签里定义键。

因为键和值的类型没有限制，所以可以自由地为它们指定<value>、<ref>、<bean>或<null/>元素。

可以将Map的键和值作为<entry>的属性定义：简单常量使用key和value来定义；bean引用通过key-ref和value-ref属性定义。

|  |
| --- |
| <bean id=*"cup"* class=*"com.atguigu.spring.bean.Cup"*>  <property name=*"bookMap"*>  <map>  <entry>  <key>  <value>bookKey01</value>  </key>  <ref bean=*"book01"*/>  </entry>  <entry>  <key>  <value>bookKey02</value>  </key>  <ref bean=*"book02"*/>  </entry>  </map>  </property>  </bean> |

### 2.4.3 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

|  |
| --- |
| <util:list id=*"bookList"*>  <ref bean=*"book01"*/>  <ref bean=*"book02"*/>  <ref bean=*"book03"*/>  <ref bean=*"book04"*/>  <ref bean=*"book05"*/>  </util:list>  <util:list id=*"categoryList"*>  <value>编程</value>  <value>极客</value>  <value>相声</value>  <value>评书</value>  </util:list> |

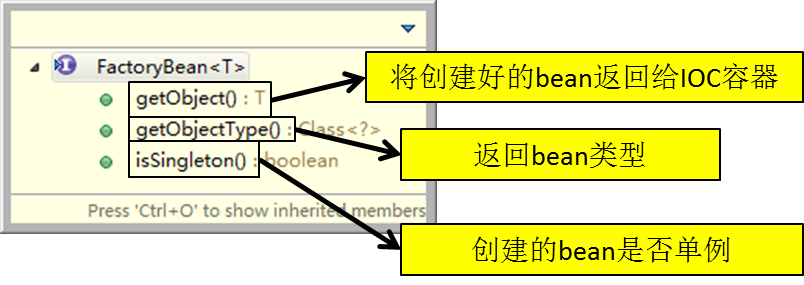
## 2.5 FactoryBean

### 2.5.1 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。

工厂bean跟普通bean不同，其返回的对象不是指定类的一个实例，其返回的是该工 厂bean的getObject方法所返回的对象。

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



|  |
| --- |
| <bean id=*"product"* class=*"com.atguigu.spring.bean.ProductFactory"*>  <property name=*"productName"* value=*"Mp3"* />  </bean> |

## 2.6 bean的高级配置

### 2.6.1 配置信息的继承

#### 1. 背景

查看下面两个Employee的配置，其中dept属性是重复的。

|  |
| --- |
| <bean id=*"dept"* class=*"com.atguigu.parent.bean.Department"*>  <property name=*"deptId"* value=*"100"*/>  <property name=*"deptName"* value=*"IT"*/>  </bean>  <bean id=*"emp01"* class=*"com.atguigu.parent.bean.Employee"*>  <property name=*"empId"* value=*"1001"*/>  <property name=*"empName"* value=*"Tom"*/>  <property name=*"age"* value=*"20"*/>  <!-- 重复的属性值 -->  <property name=*"dept"* ref=*"dept"*/>  </bean>  <bean id=*"emp02"* class=*"com.atguigu.parent.bean.Employee"*>  <property name=*"empId"* value=*"1002"*/>  <property name=*"empName"* value=*"Jerry"*/>  <property name=*"age"* value=*"25"*/>  <!-- 重复的属性值 -->  <property name=*"dept"* ref=*"dept"*/>  </bean> |

#### 2. 配置信息的继承

|  |
| --- |
| <!-- 以emp01作为父bean，继承后可以省略公共属性值的配置 -->  <bean id=*"emp02"* parent=*"emp01"*>  <property name=*"empId"* value=*"1002"*/>  <property name=*"empName"* value=*"Jerry"*/>  <property name=*"age"* value=*"25"*/>  </bean> |

Spring允许继承bean的配置，被继承的bean称为父bean。继承这个父bean的bean 称为子bean

子bean从父bean中继承配置，包括bean的属性配置

子bean也可以覆盖从父bean继承过来的配置

#### 3. 补充说明

父bean可以作为配置模板，也可以作为bean实例。若只想把父bean作为模板，可以设置<bean>的abstract 属性为true，这样Spring将不会实例化这个bean

如果一个bean的class属性没有指定，则必须是抽象bean

并不是<bean>元素里的所有属性都会被继承。比如：autowire，abstract等。

也可以忽略父bean的class属性，让子bean指定自己的类，而共享相同的属性配置。 但 此时abstract必须设为true。

### 2.6.2 bean之间的依赖

有的时候创建一个bean的时候需要保证另外一个bean也被创建，这时我们称前面的bean对后面的bean有依赖。例如：要求创建Employee对象的时候必须创建Department。 这里需要注意的是依赖关系不等于引用关系，Employee即使依赖Department也可以不引用它。

|  |
| --- |
| <bean id=*"emp03"* class=*"com.atguigu.parent.bean.Employee"* depends-on=*"dept"*>  <property name=*"empId"* value=*"1003"*/>  <property name=*"empName"* value=*"Kate"*/>  <property name=*"age"* value=*"21"*/>  </bean> |

## 2.7 bean的作用域★

在Spring中，可以在<bean>元素的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。

## 2.8 bean的生命周期

1. 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。
2. 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 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. 在配置bean时，通过init-method和destroy-method 属性为bean指定初始化和销毁方法
2. bean的后置处理器

① 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

② 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。

③ bean后置处理器时需要实现接口：

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)

1. 添加bean后置处理器后bean的生命周期

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**

## 2.9 引用外部属性文件

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。

### 2.9.1 直接配置

|  |
| --- |
| <!-- 直接配置 -->  <bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <property name=*"user"* value=*"root"*/>  <property name=*"password"* value=*"root"*/>  <property name=*"jdbcUrl"* value=*"jdbc:mysql:///test"*/>  <property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*/>  </bean> |

### 2.9.2 使用外部的属性文件

#### 1. 创建properties属性文件

|  |
| --- |
| prop.userName=root  prop.password=root  prop.url=jdbc:mysql:///test  prop.driverClass=com.mysql.jdbc.Driver |

#### 2. 引入context名称空间



#### 3.指定properties属性文件的位置

|  |
| --- |
| <!-- 指定properties属性文件的位置 -->  <!-- classpath:xxx 表示属性文件位于类路径下 -->  <context:property-placeholder location=*"classpath:jdbc.properties"*/> |

#### 4.从properties属性文件中引入属性值

|  |
| --- |
| <!-- 从properties属性文件中引入属性值 -->  <bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <property name=*"user"* value=*"****${prop.userName}****"*/>  <property name=*"password"* value=*"****${prop.password}****"*/>  <property name=*"jdbcUrl"* value=*"****${prop.url}****"*/>  <property name=*"driverClass"* value=*"****${prop.driverClass}****"*/>  </bean> |

## 2.10 自动装配

### 2.10.1 自动装配的概念

1. 手动装配：以value或ref的方式**明确指定属性值**都是手动装配。
2. 自动装配：根据指定的装配规则，**不需要明确指定**，Spring**自动**将匹配的属性值**注入**bean中。

### 2.10.2 装配模式

1. 根据**类型**自动装配：将类型匹配的bean作为属性注入到另一个bean中。若IOC容器中有多个与目标bean类型一致的bean，Spring将无法判定哪个bean最合适该属性，所以不能执行自动装配
2. 根据**名称**自动装配：必须将目标bean的名称和属性名设置的完全相同
3. 通过构造器自动装配：当bean中存在多个构造器时，此种自动装配方式将会很复杂。不推荐使用。

### 2.10.3 选用建议

相对于使用注解的方式实现的自动装配，在XML文档中进行的自动装配略显笨拙，在项目中更多的使用注解的方式实现。

## 2.11 通过注解配置bean

### 2.11.1 概述

相对于XML方式而言，通过注解的方式配置bean更加简洁和优雅，而且和MVC组件化开发的理念十分契合，是开发中常用的使用方式。

### 2.11.2 使用注解标识组件

1. 普通组件：@Component

标识一个受Spring IOC容器管理的组件

1. 持久化层组件：@Repository

标识一个受Spring IOC容器管理的持久化层组件

1. 业务逻辑层组件：@Service

标识一个受Spring IOC容器管理的业务逻辑层组件

1. 表述层控制器组件：@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使用组件注解的value属性指定bean的id

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注解用在一个表述层控制器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。

### 2.11.3 扫描组件

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package

|  |
| --- |
| <context:component-scan base-package=*"com.atguigu.component"*/> |

1. 详细说明

①**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。

②当需要扫描多个包时可以使用逗号分隔。

③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，示例：

|  |
| --- |
| <context:component-scan  base-package=*"com.atguigu.component"*  resource-pattern=*"autowire/\*.class"*/> |

④包含与排除

●<context:include-filter>子节点表示要包含的目标类

注意：通常需要与use-default-filters属性配合使用才能够达到“仅包含某些 组件”这样的效果。即：通过将use-default-filters属性设置为false， 禁用默认过滤器，然后扫描的就只是include-filter中的规则指定的 组件了。

●<context:exclude-filter>子节点表示要排除在外的目标类

●component-scan下可以拥有若干个include-filter和exclude-filter子节点

●过滤表达式

|  |  |  |
| --- | --- | --- |
| 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 |

1. JAR包

必须在原有JAR包组合的基础上再导入一个：spring-aop-4.0.0.RELEASE.jar

### 2.11.4 组件装配

1. 需求

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。

1. 实现依据

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

①根据类型实现自动装配。

②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

③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。

④若某一属性允许不被设置，可以设置@Autowired注解的required属性为 false

⑤默认情况下，当IOC容器里存在多个类型兼容的bean时，Spring会尝试匹配bean 的id值是否与变量名相同，如果相同则进行装配。如果bean的id值不相同，通过类 型的自动装配将无法工作。此时可以在@Qualifier注解里提供bean的名称。Spring 甚至允许在方法的形参上标注@Qualifiter注解以指定注入bean的名称。 ⑥@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。

⑦@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。

⑧@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。

1. @Resource

@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

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

# 第3章 AOP前奏

## 3.1 提出问题

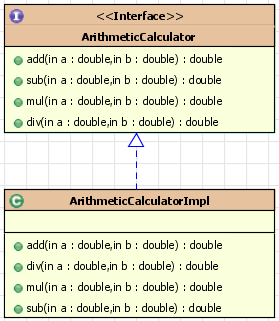
### 3.1.1 情景：数学计算器

1. 要求

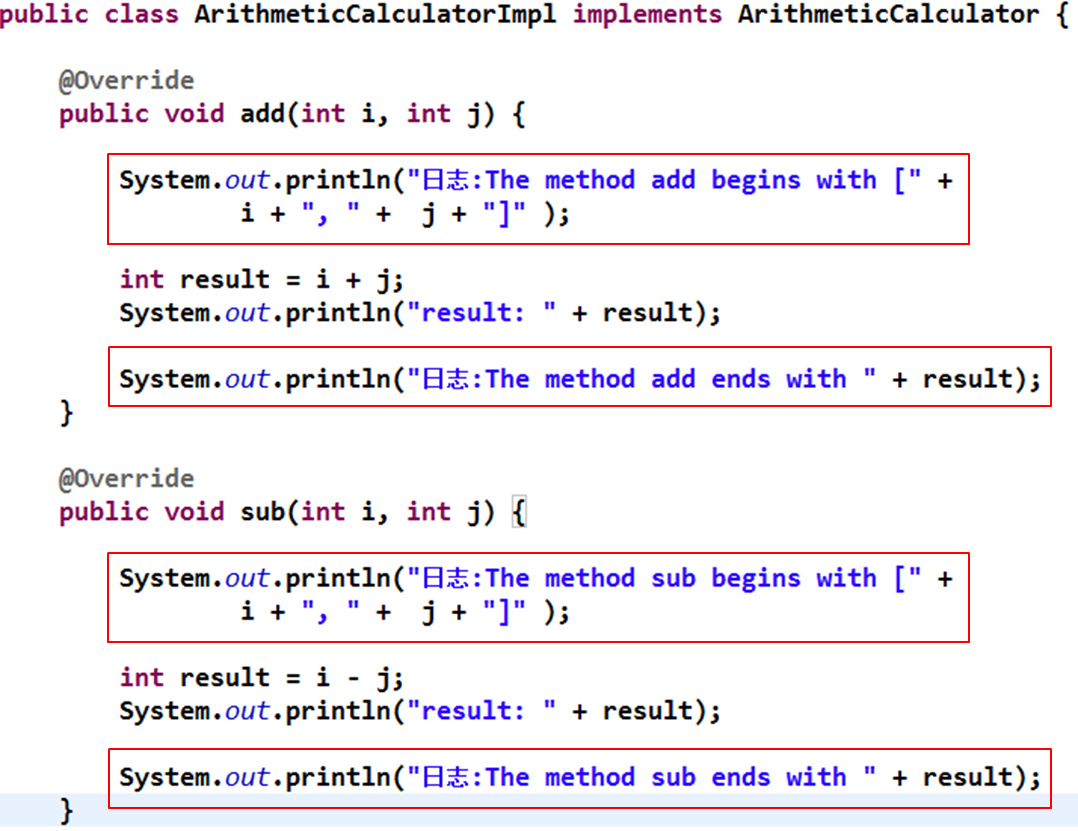
①执行加减乘除运算

②日志：在程序执行期间追踪正在发生的活动

③验证：希望计算器只能处理正数的运算



1. 常规实现



1. 问题

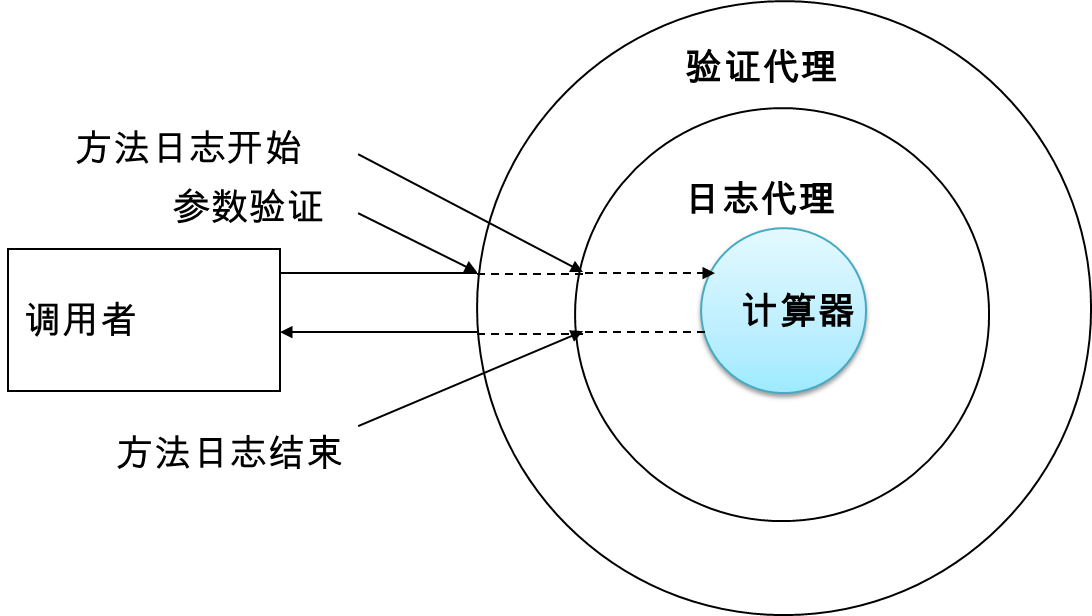
①代码混乱：越来越多的非业务需求(日志和验证等)加入后，原有的业务方法急剧膨胀。每个方法在处理核心逻辑的同时还必须兼顾其他多个关注点。

②代码分散: 以日志需求为例，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。

## 3.2 动态代理

### 3.2.1 动态代理的原理

代理设计模式的原理：**使用一个代理将对象包装起来**，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。

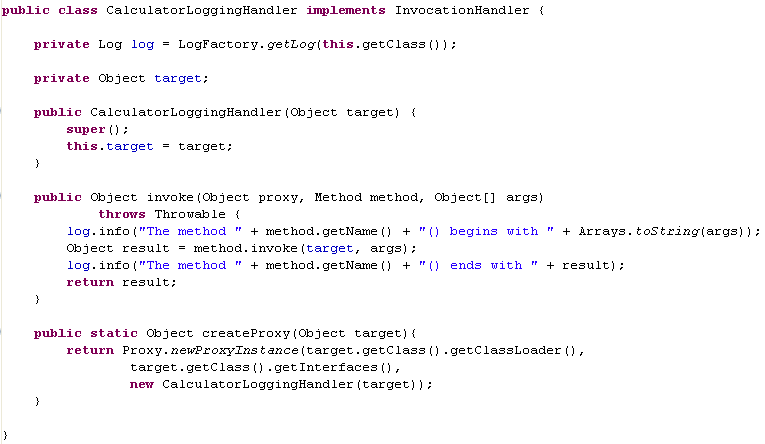


### 3.2.2 动态代理的方式

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

## 3.3 数学计算器的改进

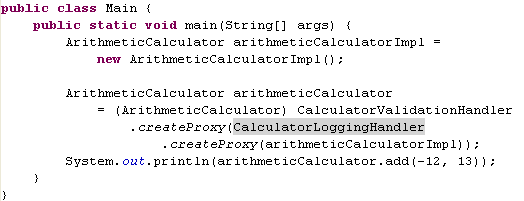
### 3.3.1 日志处理器



### 3.3.2 验证处理器



### 3.3.3 测试代码



### 3.3.4 保存生成的动态代理类

|  |
| --- |
| 在测试方法中加入如下代码：  Properties properties = System.getProperties();  properties.put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true"); |

# 第4章 AOP概述

## 4.1 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

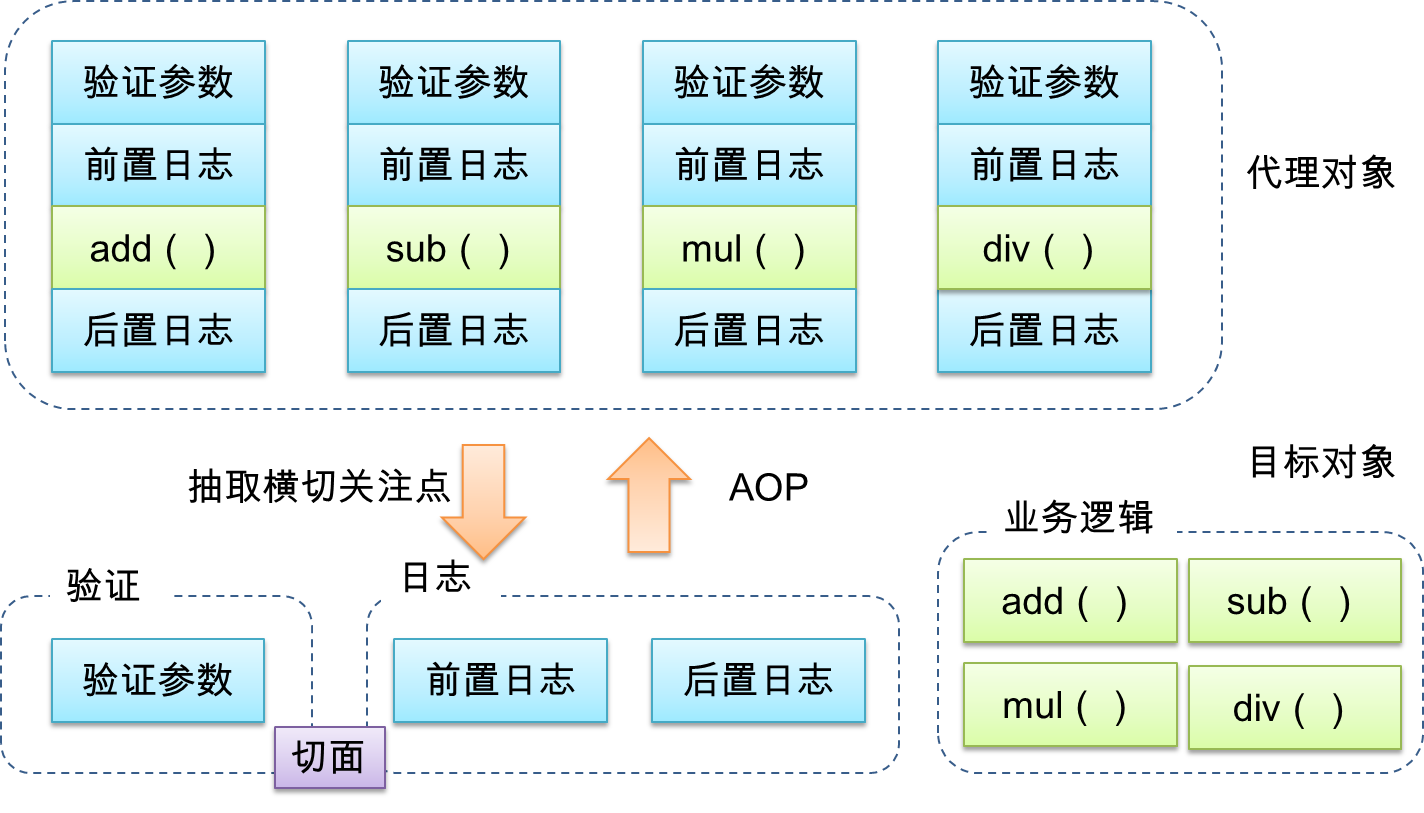
统 OOP(Object-Oriented Programming，面向对象编程)的补充。

1. AOP编程操作的主要对象是切面(aspect)，而切面**模块化横切关注点**。
2. 在应用AOP编程时，仍然需要定义公共功能，但可以明确的定义这个功能应用在哪里，以什么方式应用，并且不必修改受影响的类。这样一来横切关注点就被模块化到特殊的类里——这样的类我们通常称之为“切面”。
3. 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



## 4.2 AOP术语

### 4.2.1 横切关注点

从每个方法中抽取出来的同一类非核心业务。

### 4.2.2 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。

### 4.2.3 通知(Advice)

切面必须要完成的各个具体工作

### 4.2.4 目标(Target)

被通知的对象

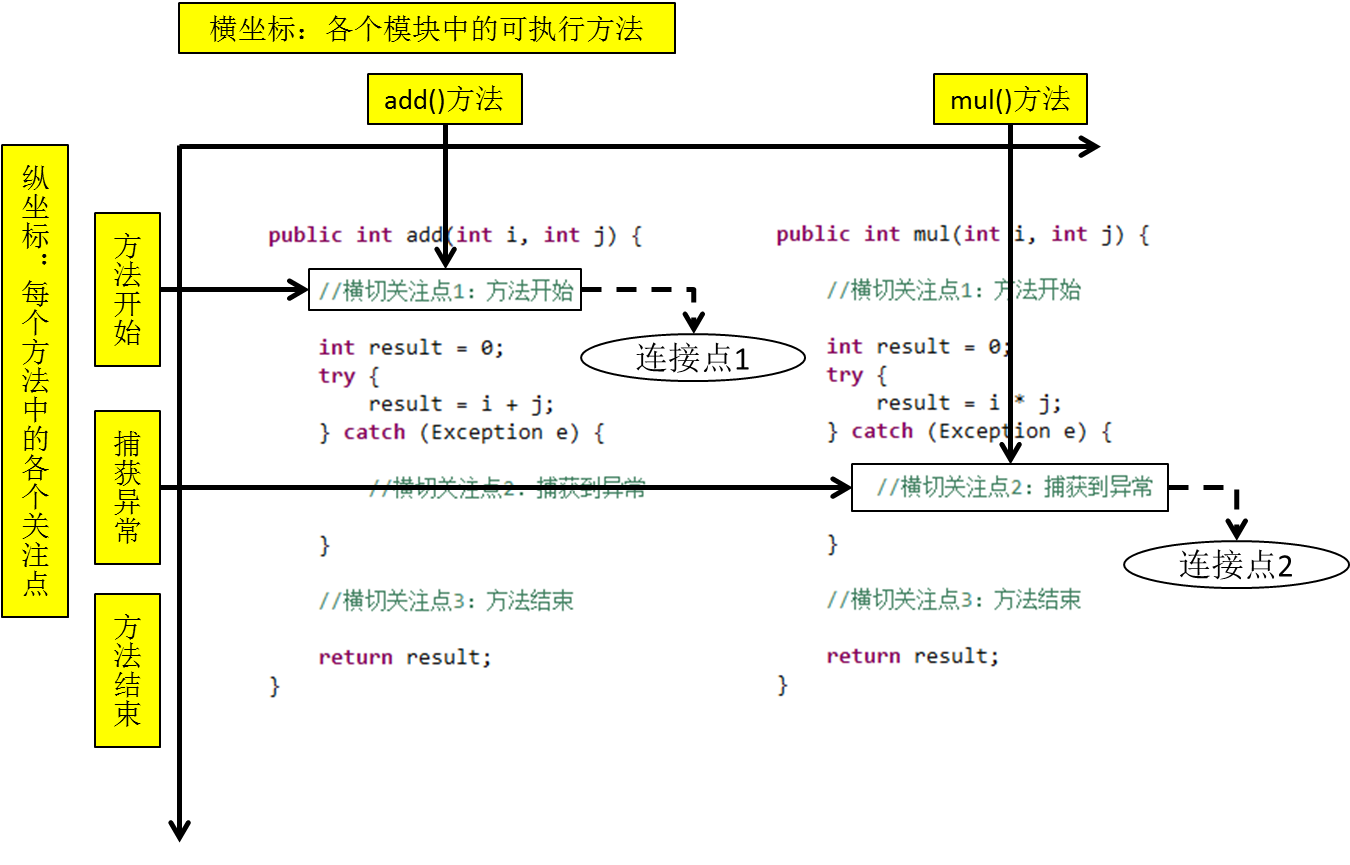
### 4.2.5 代理(Proxy)

向目标对象应用通知之后创建的代理对象

### 4.2.6 连接点(Joinpoint)

横切关注点在程序代码中的具体体现，对应程序执行的某个特定位置。例如：类某个方法调用前、调用后、方法捕获到异常后等。

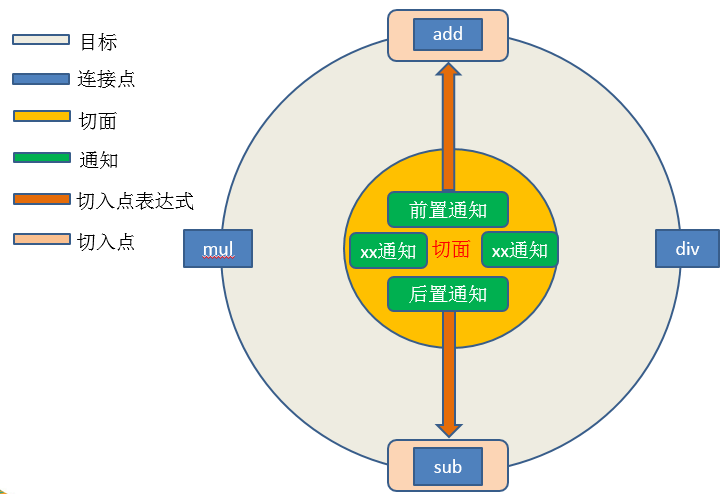
在应用程序中可以使用横纵两个坐标来定位一个具体的连接点：



### 4.2.7 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。

### 4.2.8 图解



## 4.3 AspectJ

### 4.3.1 简介

AspectJ：Java社区里最完整最流行的AOP框架。

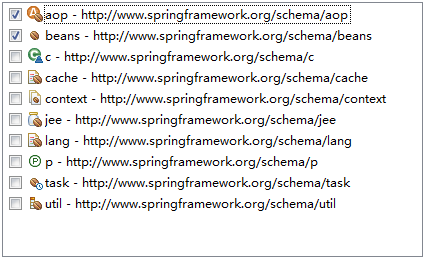
在Spring2.0以上版本中，可以使用基于AspectJ注解或基于XML配置的AOP。

### 4.3.2 在Spring中启用AspectJ注解支持

1. 导入JAR包

* com.springsource.net.sf.cglib-2.2.0.jar
* com.springsource.org.aopalliance-1.0.0.jar
* com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar
* spring-aop-4.0.0.RELEASE.jar
* spring-aspects-4.0.0.RELEASE.jar

1. 引入aop名称空间



1. 配置

<aop:aspectj-autoproxy>

当Spring IOC容器侦测到bean配置文件中的<aop:aspectj-autoproxy>元素时，会自动为 与AspectJ切面匹配的bean创建代理

### 4.3.3用AspectJ注解声明切面

1. 要在Spring中声明AspectJ切面，只需要在IOC容器中将切面声明为bean实例。
2. 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。
3. 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。
4. 通知是标注有某种注解的简单的Java方法。
5. 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

⑥ @Around：环绕通知，围绕着方法执行

# 第5章 AOP细节

## 5.1 切入点表达式

### 5.1.1 作用

通过**表达式的方式**定位**一个或多个**具体的连接点。

### 5.1.2 语法细节

1. 切入点表达式的语法格式

|  |
| --- |
| 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.ArithmeticCalculator.**\***(**..**)) |
| 含义 | ArithmeticCalculator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** \* ArithmeticCalculator.\*(..)) |
| 含义 | ArithmeticCalculator接口的所有公有方法 |

|  |  |
| --- | --- |
| 表达式 | execution(public **double** ArithmeticCalculator.\*(..)) |
| 含义 | ArithmeticCalculator接口中返回double类型数值的方法 |

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

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

3）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 |

### 5.1.3切入点表达式应用到实际的切面类中

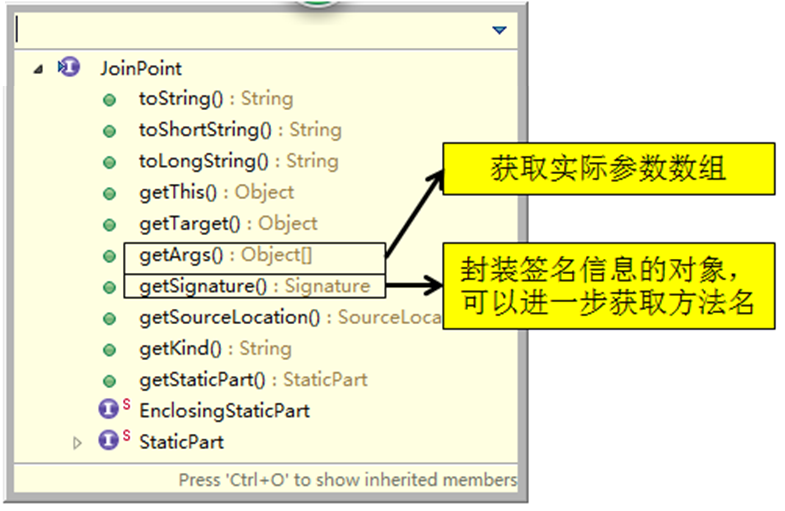


## 5.2 当前连接点细节

### 5.2.1 概述

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。

### 5.2.2 JoinPoint



## 5.3通知

### 5.3.1 概述

1. 在具体的连接点上要执行的操作。
2. 一个切面可以包括一个或者多个通知。
3. 通知所使用的注解的值往往是切入点表达式。

### 前置通知

1. 前置通知：在方法执行之前执行的通知
2. 使用@Before注解

### 5.3.3后置通知

1. 后置通知：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
2. 使用@After注解

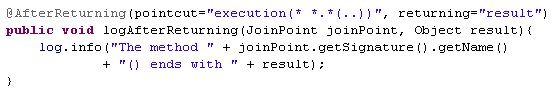
### 5.3.4返回通知

1. 返回通知：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。
2. 使用@AfterReturning注解,在返回通知中访问连接点的返回值

①在返回通知中，只要将returning属性添加到@AfterReturning注解中，就可以访问连接点的返回值。该属性的值即为用来传入返回值的参数名称

②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

③原始的切点表达式需要出现在pointcut属性中

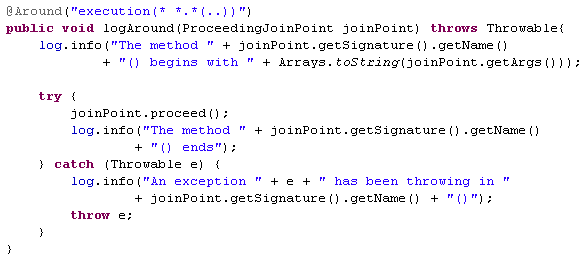


### 5.3.5异常通知

1. 异常通知：只在连接点抛出异常时才执行异常通知
2. 将throwing属性添加到@AfterThrowing注解中，也可以访问连接点抛出的异常。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。
3. 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

### 5.3.6环绕通知

1. 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。
2. 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。
3. 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。
4. 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，否则会出现空指针异常。



## 5.4 重用切入点定义

1. 在编写AspectJ切面时，可以直接在通知注解中书写切入点表达式。但同一个切点表达式可能会在多个通知中重复出现。
2. 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。
3. 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. 其他通知可以通过方法名称引入该切入点



## 指定切面的优先级

1. 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。
2. 切面的优先级可以通过实现Ordered接口或利用@Order注解指定。
3. 实现Ordered接口，getOrder()方法的返回值越小，优先级越高。
4. 若使用@Order注解，序号出现在注解中





# 第6章 以XML方式配置切面

## 6.1 概述

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。

## 6.2 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。



## 6.3 声明切入点

1. 切入点使用<aop:pointcut>元素声明。
2. 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。

① 定义在<aop:aspect>元素下：只对当前切面有效

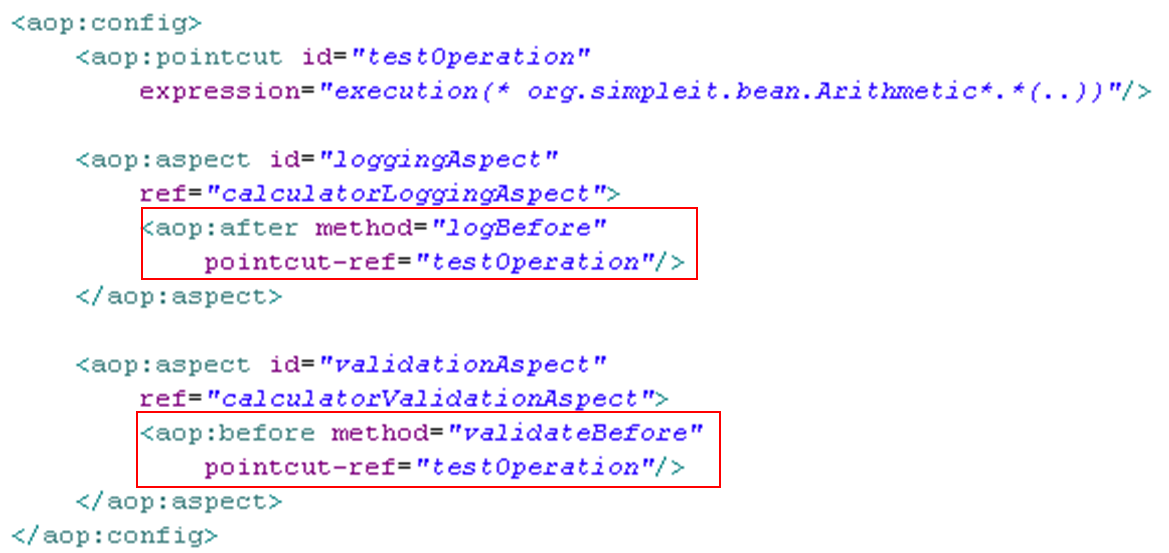
② 定义在<aop:config>元素下：对所有切面都有效

1. 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。



## 6.4 声明通知

1. 在aop名称空间中，每种通知类型都对应一个特定的XML元素。
2. 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。
3. method属性指定切面类中通知方法的名称



# 第7章 JdbcTemplate

## 7.1 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。

## 7.2 环境准备

### 7.2.1导入JAR包

1. IOC容器所需要的JAR包

commons-logging-1.1.1.jar

spring-beans-4.0.0.RELEASE.jar

spring-context-4.0.0.RELEASE.jar

spring-core-4.0.0.RELEASE.jar

spring-expression-4.0.0.RELEASE.jar

1. JdbcTemplate所需要的JAR包

spring-jdbc-4.0.0.RELEASE.jar

spring-orm-4.0.0.RELEASE.jar

spring-tx-4.0.0.RELEASE.jar

1. 数据库驱动和数据源

c3p0-0.9.1.2.jar

mysql-connector-java-5.1.7-bin.jar

### 7.2.2创建连接数据库基本信息属性文件

|  |
| --- |
| user=root  password=root  jdbcUrl=jdbc:mysql:///query\_data  driverClass=com.mysql.jdbc.Driver  initialPoolSize=30  minPoolSize=10  maxPoolSize=100  acquireIncrement=5  maxStatements=1000  maxStatementsPerConnection=10 |

### 7.2.3在Spring配置文件中配置相关的bean

1. 数据源对象

|  |
| --- |
| <context:property-placeholder location=*"classpath:jdbc.properties"*/>  <bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <property name=*"user"* value=*"${user}"*/>  <property name=*"password"* value=*"${password}"*/>  <property name=*"jdbcUrl"* value=*"${jdbcUrl}"*/>  <property name=*"driverClass"* value=*"${driverClass}"*/>  <property name=*"initialPoolSize"* value=*"${initialPoolSize}"*/>  <property name=*"minPoolSize"* value=*"${minPoolSize}"*/>  <property name=*"maxPoolSize"* value=*"${maxPoolSize}"*/>  <property name=*"acquireIncrement"* value=*"${acquireIncrement}"*/>  <property name=*"maxStatements"* value=*"${maxStatements}"*/>  <property name=*"maxStatementsPerConnection"*  value=*"${maxStatementsPerConnection}"*/>  </bean> |

1. JdbcTemplate对象

|  |
| --- |
| <bean id=*"template"*  class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"dataSource"*/>  </bean> |

## 7.3 持久化操作

1. 增删改

JdbcTemplate.update(String, Object...)

1. 批量增删改

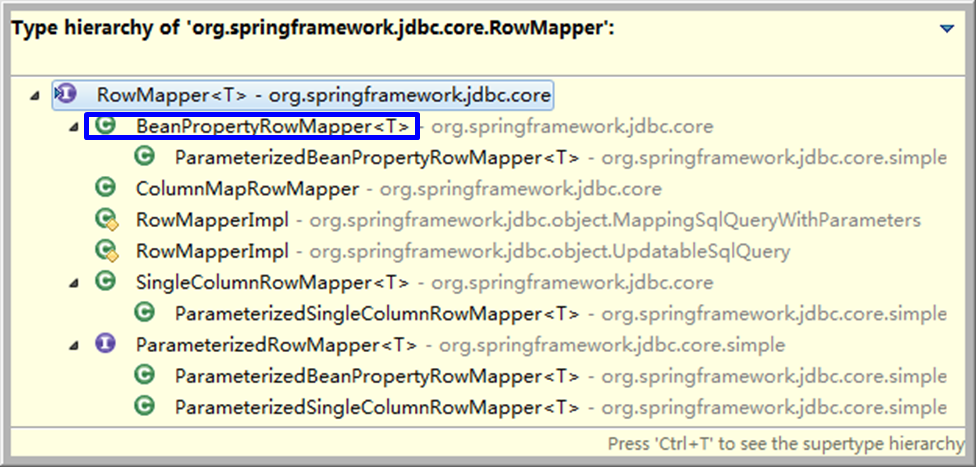
JdbcTemplate.batchUpdate(String, List<Object[]>)

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

1. 查询单行

JdbcTemplate.queryForObject(String, RowMapper<Department>, Object...)



1. 查询多行

JdbcTemplate.query(String, RowMapper<Department>, Object...)

RowMapper对象依然可以使用BeanPropertyRowMapper

1. 查询单一值

JdbcTemplate.queryForObject(String, Class, Object...)

## 7.4 使用具名参数的JdbcTemplate

1. 关于具名参数

在Hibernate的HQL查询中我们体验过具名参数的使用，相对于基于位置的参数，具名参数具有更好的可维护性，在SQL语句中参数较多时可以考虑使用具名参数。

在Spring中可以通过NamedParameterJdbcTemplate类的对象使用带有具名参数的SQL语句。

1. 通过IOC容器创建NamedParameterJdbcTemplate对象

|  |
| --- |
| <!-- 配置可以使用具名参数的JDBCTemplate类对象 -->  <bean  id=*"namedTemplate"*  class=*"org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate"*>  <!-- 没有无参构造器，必须传入数据源或JdbcTemplate对象 -->  <constructor-arg ref=*"dataSource"*/>  </bean> |

1. 具名参数在SQL语句中的格式

|  |
| --- |
| INSERT INTO depts (dept\_name) VALUES (**:deptName**) |

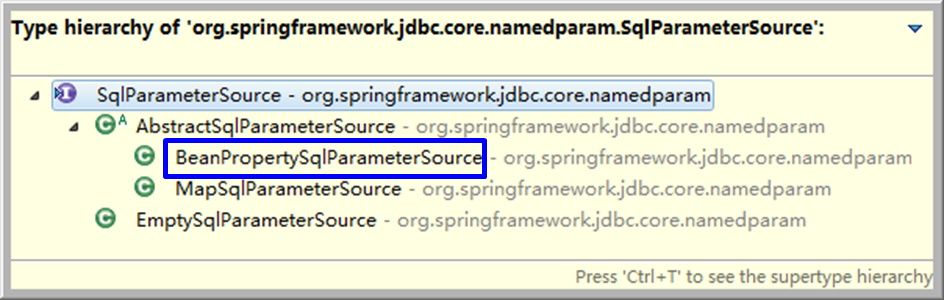
1. 具名参数传入

①通过Map对象传入

NamedParameterJdbcTemplate.update(String sql, Map<String, ?> map)

Map的键是参数名，值是参数值

② 通过SqlParameterSource对象传入



|  |
| --- |
| String sql = "INSERT INTO depts (dept\_name) VALUES (:deptName)";  Department department = **new** Department(**null**, "YYY", **null**);  SqlParameterSource sqlParameterSource =  **new** BeanPropertySqlParameterSource(department);  namedTemplate.update(sql, sqlParameterSource); |

## 7.5 使用JdbcTemplate实现Dao

1. 通过IOC容器自动注入

**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。

|  |
| --- |
| **@Repository**  public class EmployeeDao {    **@Autowired**  private JdbcTemplate **jdbcTemplate**;    public Employee get(Integer id){  //…  }  } |

# 第8章 声明式事务

## 8.1事务概述

1. 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。
2. 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**。
3. 事务的四个关键属性(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。

## 8.2 Spring事务管理

### 8.2.1编程式事务管理

1. 使用原生的JDBC API进行事务管理

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

1. 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**。

### 8.2.2 声明式事务管理

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。

Spring既支持编程式事务管理，也支持声明式的事务管理。

### 8.2.3 Spring提供的事务管理器

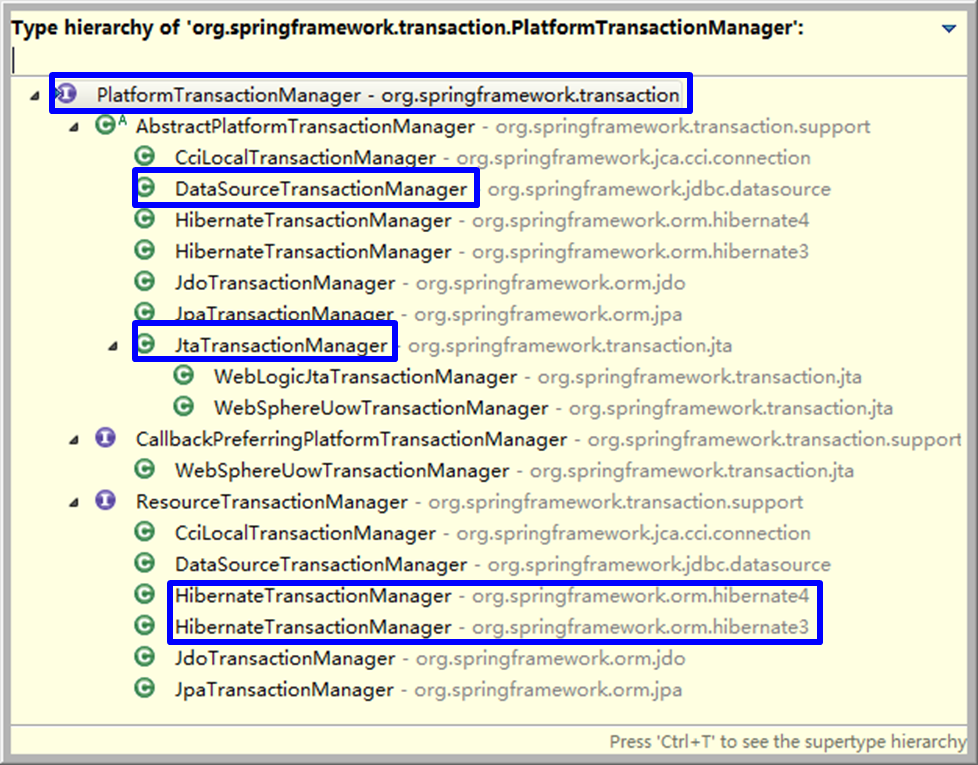
Spring从不同的事务管理API中抽象出了一整套事务管理机制，让事务管理代码从特定的事务技术中独立出来。开发人员通过配置的方式进行事务管理，而不必了解其底层是如何实现的。

Spring的核心事务管理抽象是PlatformTransactionManager。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。

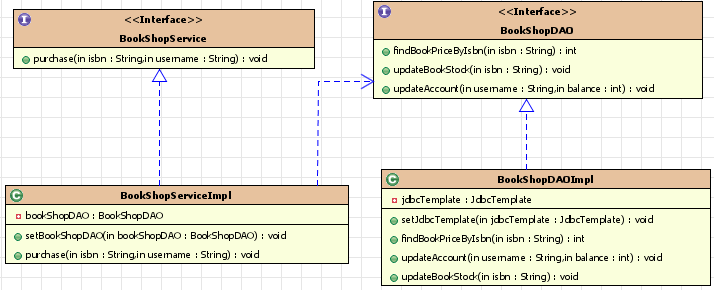
### 8.2.4事务管理器的主要实现

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. JtaTransactionManager：在JavaEE应用服务器上用JTA(Java Transaction API)进行事务管理
3. HibernateTransactionManager：用Hibernate框架存取数据库



## 8.3 测试数据准备

### 8.3.1 需求



### 8.3.2 数据库表

|  |
| --- |
| CREATE TABLE **book** (  isbn VARCHAR (50) PRIMARY KEY,  book\_name VARCHAR (100),  price INT  ) ;  CREATE TABLE **book\_stock** (  isbn VARCHAR (50) PRIMARY KEY,  stock INT,  CHECK (stock > 0)  ) ;  CREATE TABLE **account** (  username VARCHAR (50) PRIMARY KEY,  balance INT,  CHECK (balance > 0)  ) ;  INSERT INTO account (`username`,`balance`) VALUES ('Tom',100000);  INSERT INTO account (`username`,`balance`) VALUES ('Jerry',150000);  INSERT INTO book (`isbn`,`book\_name`,`price`) VALUES ('ISBN-001','book01',100);  INSERT INTO book (`isbn`,`book\_name`,`price`) VALUES ('ISBN-002','book02',200);  INSERT INTO book (`isbn`,`book\_name`,`price`) VALUES ('ISBN-003','book03',300);  INSERT INTO book (`isbn`,`book\_name`,`price`) VALUES ('ISBN-004','book04',400);  INSERT INTO book (`isbn`,`book\_name`,`price`) VALUES ('ISBN-005','book05',500);  INSERT INTO book\_stock (`isbn`,`stock`) VALUES ('ISBN-001',1000);  INSERT INTO book\_stock (`isbn`,`stock`) VALUES ('ISBN-002',2000);  INSERT INTO book\_stock (`isbn`,`stock`) VALUES ('ISBN-003',3000);  INSERT INTO book\_stock (`isbn`,`stock`) VALUES ('ISBN-004',4000);  INSERT INTO book\_stock (`isbn`,`stock`) VALUES ('ISBN-005',5000); |

## 8.4 初步实现

1. 配置文件

|  |
| --- |
| <!-- 配置事务管理器 -->  <bean id=*"transactionManager"*  class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <property name=*"dataSource"* ref=*"dataSource"*/>  </bean>  <!-- 启用事务注解 -->  <tx:annotation-driven transaction-manager=*"transactionManager"*/> |

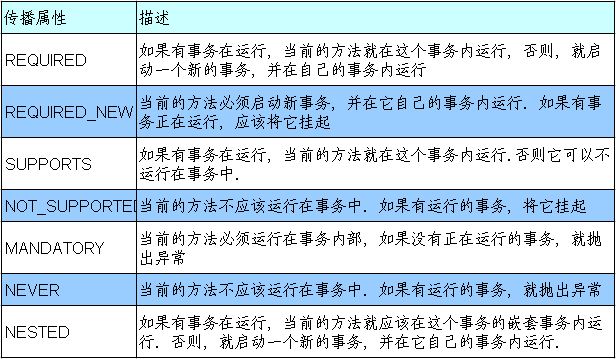
1. 在需要进行事务控制的方法上加注解 @Transactional

## 8.5 事务的传播行为

### 8.5.1 简介

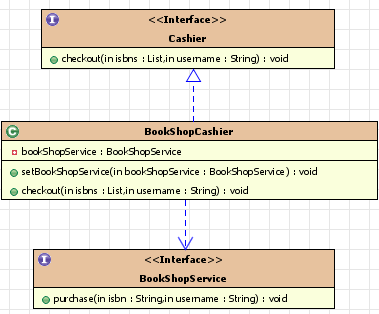
当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。

事务的传播行为可以由传播属性指定。Spring定义了7种类传播行为。



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

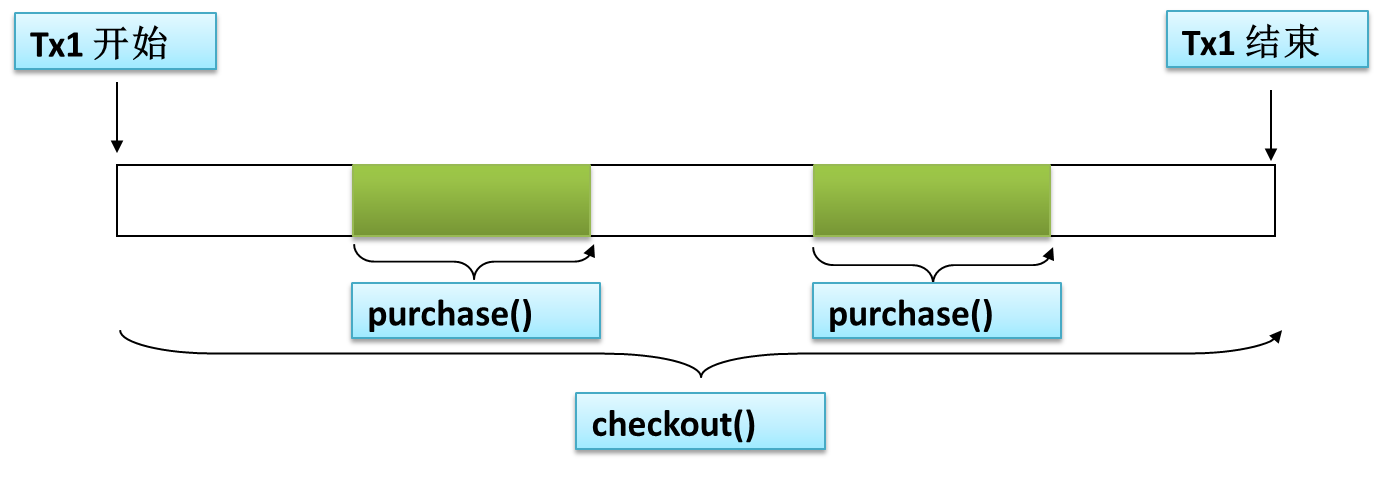
### 8.5.2 测试



1. . 说明

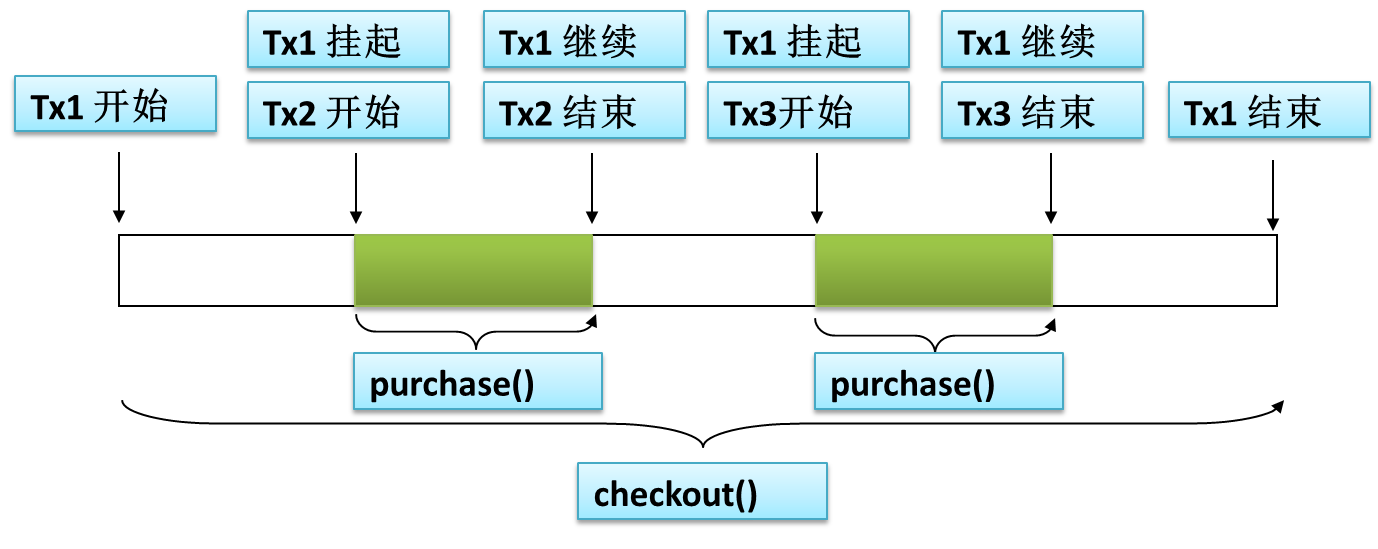
①REQUIRED传播行为

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。



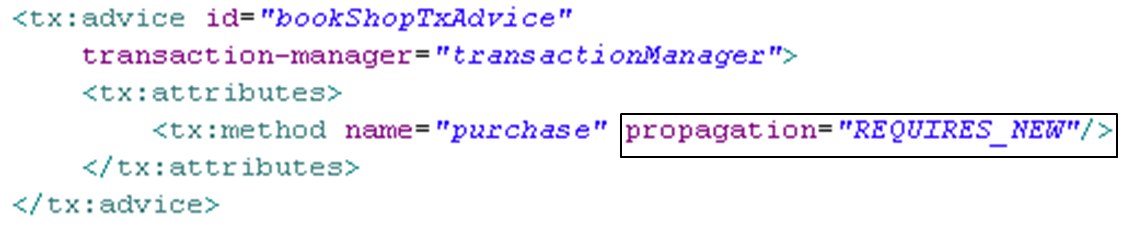
②. REQUIRES\_NEW传播行为

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。



### 8.5.3 补充

在Spring 2.x事务通知中，可以像下面这样在<tx:method>元素中设定传播事务属性。



## 8.6 事务的隔离级别

### 8.6.1 数据库事务并发问题

假设现在有两个事务：Transaction01和Transaction02并发执行。

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。

1. 不可重复读

①Transaction01读取了AGE值为20。

②Transaction02将AGE值修改为30。

③Transaction01再次读取AGE值为30，和第一次读取不一致。

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。

### 8.6.2 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。

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 | √ | √ |

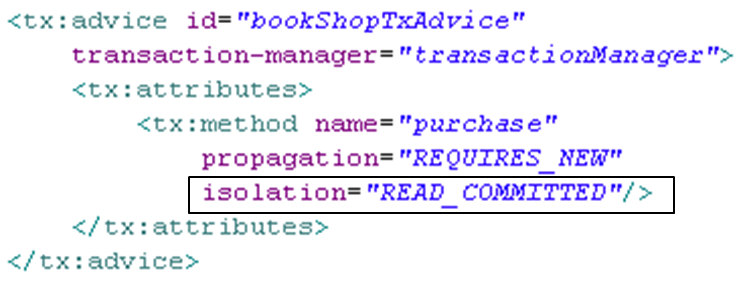
### 8.6.3 在Spring中指定事务隔离级别

1. 注解

用@Transactional注解声明式地管理事务时可以在@Transactional的isolation属性中设置隔离级别

1. XML

在Spring 2.x事务通知中，可以在<tx:method>元素中指定隔离级别



## 8.7 触发事务回滚的异常

### 8.7.1默认情况

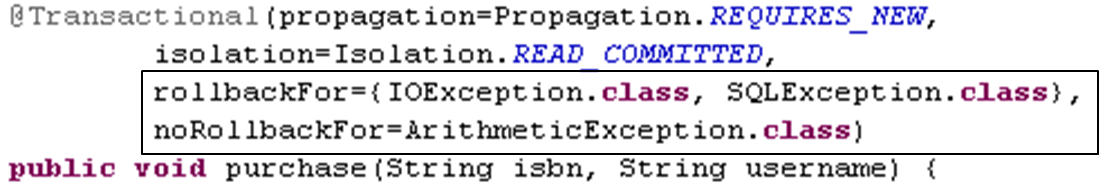
捕获到RuntimeException或Error时回滚，而捕获到编译时异常不回滚。

### 8.7.2设置途经

1. 注解@Transactional 注解

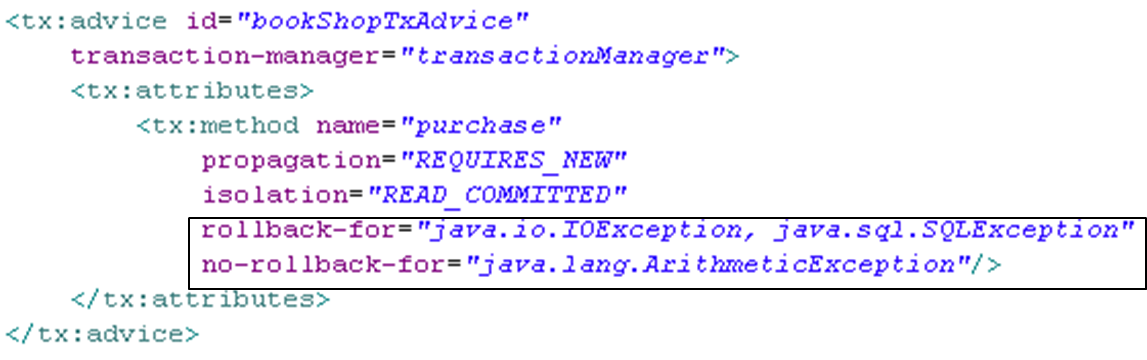
① rollbackFor属性：指定遇到时必须进行回滚的异常类型，可以为多个

② noRollbackFor属性：指定遇到时不回滚的异常类型，可以为多个



1. XML

在Spring 2.x事务通知中，可以在<tx:method>元素中指定回滚规则。如果有不止一种异常则用逗号分隔。



## 8.8 事务的超时和只读属性

### 8.8.1简介

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。

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。

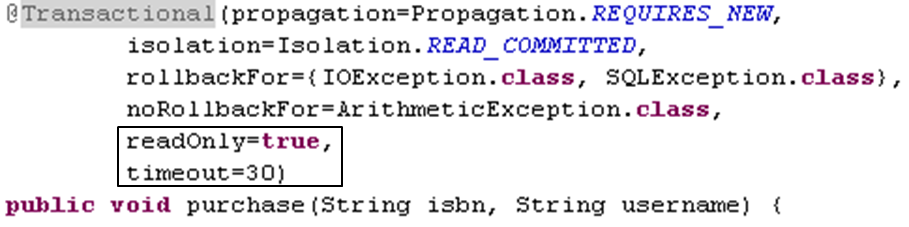
超时事务属性：事务在强制回滚之前可以保持多久。这样可以防止长期运行的事务占用资源。

只读事务属性: 表示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。

### 8.8.2设置

1. 注解

@Transaction注解



1. XML

在Spring 2.x事务通知中，超时和只读属性可以在<tx:method>元素中进行指定



## 8.9 基于XML文档的声明式事务配置

|  |
| --- |
| <!-- 配置事务切面 -->  <aop:config>  <aop:pointcut  expression="execution(\* com.atguigu.tx.component.service.BookShopServiceImpl.purchase(..))"  id="txPointCut"/>  <!-- 将切入点表达式和事务属性配置关联到一起 -->  <aop:advisor advice-ref="myTx" pointcut-ref="txPointCut"/>  </aop:config>    <!-- 配置基于XML的声明式事务 -->  <tx:advice id="myTx" transaction-manager="transactionManager">  <tx:attributes>  <!-- 设置具体方法的事务属性 -->  <tx:method name="find\*" read-only="true"/>  <tx:method name="get\*" read-only="true"/>  <tx:method name="purchase"  isolation="READ\_COMMITTED"  no-rollback-for="java.lang.ArithmeticException,java.lang.NullPointerException"  propagation="REQUIRES\_NEW"  read-only="false"  timeout="10"/>  </tx:attributes>  </tx:advice> |