# Spring学习笔记

## spring是什么

1. Spring 是一个开源框架.
2. Spring 是一个 IOC(DI) 和 AOP 容器框架.
3. 这个框架本身具有轻量（1.资源消耗小2.侵入程度低3.开发较简单）、

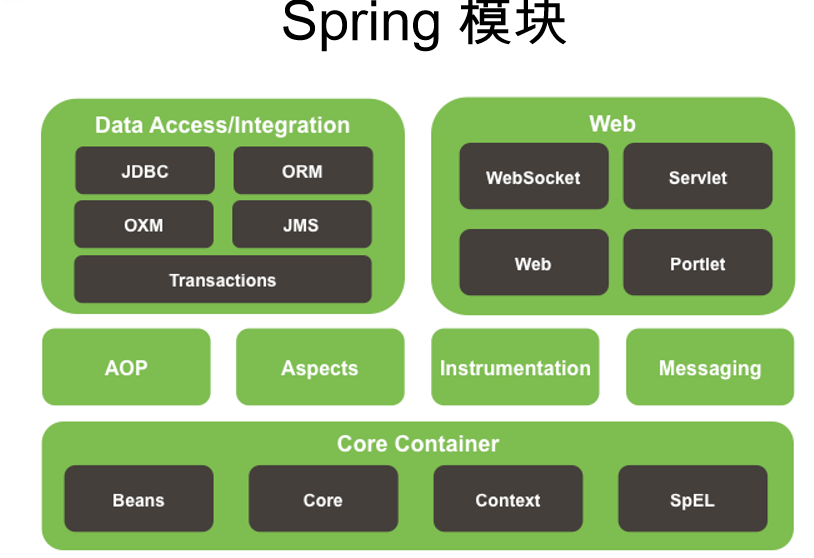
依赖注入(IOC)、

面向切面编程(AOP)的特点。

他是一个容器（包含并且管理应用对象的生命周期

）。

## Spring包含的模块



## spring的使用

### IOC & DI 概述

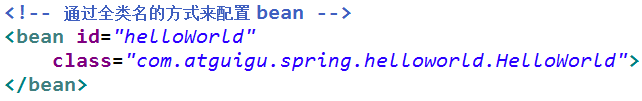
**IOC(Inversion of Control)：**其思想是反转资源获取的方向. 传统的资源查找方式要求组件向容器发起请求查找资源. 作为回应, 容器适时的返回资源. 而应用了 IOC 之后, 则是容器主动地将资源推送给它所管理的组件, 组件所要做的仅是选择一种合适的方式来接受资源. 这种行为也被称为查找的被动形式

**DI(Dependency Injection) — 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)接受来自如容器的资源注入. 相对于 IOC 而言，这种表述更直接

### 配置 bean

#### 在 Spring 的 IOC 容器里配置 Bean

**在 xml 文件中通过 bean 节点来配置 bean**



id：Bean 的名称。

在 IOC 容器中必须是唯一的

若 id 没有指定，Spring 自动将权限定性类名作为 Bean 的名字

id 可以指定多个名字，名字之间可用逗号、分号、或空格分隔

### Spring 提供的IOC容器

在 Spring IOC 容器读取 Bean 配置创建 Bean 实例之前, 必须对它进行实例化. 只有在容器实例化后, 才可以从 IOC 容器里获取 Bean 实例并使用.

**Spring 提供了两种类型的 IOC 容器实现.**

**BeanFactory: IOC 容器的基本实现.**

**ApplicationContext: 提供了更多的高级特性. 是 BeanFactory 的子接口.**

BeanFactory 是 Spring 框架的基础设施，面向 Spring 本身；ApplicationContext 面向使用 Spring 框架的开发者，几乎所有的应用场合都直接使用 ApplicationContext 而非底层的 BeanFactory

无论使用何种方式, 配置文件时相同的.

**ApplicationContext 的主要实现类：**

ClassPathXmlApplicationContext：从 类路径下加载配置文件

FileSystemXmlApplicationContext: 从文件系统中加载配置文件

ConfigurableApplicationContext 扩展于 ApplicationContext，新增加两个主要方法：refresh() 和 close()， 让 ApplicationContext 具有启动、刷新和关闭上下文的能力

**ApplicationContext 在初始化上下文时就实例化所有单例的 Bean。**

WebApplicationContext 是专门为 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

### 从 IOC 容器中获取 Bean

调用 ApplicationContext 的 getBean() 方法

### 依赖注入的方式为Bean的属性赋值

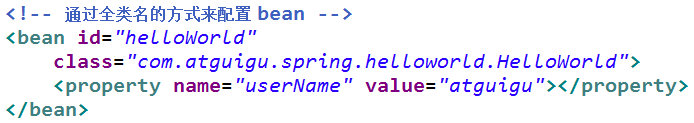
#### Spring 支持 3 种依赖注入的方式

#### 属性注入

属性注入即通过 setter 方法注入Bean 的属性值或依赖的对象

**属性注入使用 <property> 元素, 使用 name 属性指定 Bean 的属性名称，value 属性或 <value> 子节点指定属性值**

**属性注入是实际应用中最常用的注入方式**

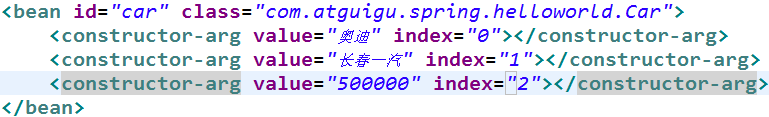


#### 构造器注入

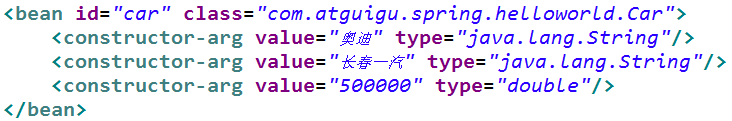
**通过构造方法注入Bean 的属性值或依赖的对象，它保证了 Bean 实例在实例化后就可以使用。**

**构造器注入在 <constructor-arg> 元素里声明属性, <constructor-arg> 中没有 name 属性**

**按索引匹配入参：**



**按类型匹配入参：**



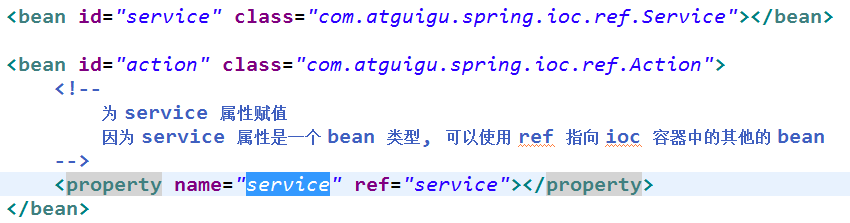
工厂方法注入（很少使用，不推荐）

### E.引用其它 Bean

**组成应用程序的 Bean 经常需要相互协作以完成应用程序的功能. 要使 Bean 能够相互访问, 就必须在 Bean 配置文件中指定对 Bean 的引用**

**在 Bean 的配置文件中, 可以通过 <ref> 元素或 ref 属性为 Bean 的属性或构造器参数指定对 Bean 的引用.**

**也可以在属性或构造器里包含 Bean 的声明, 这样的 Bean 称为内部 Bean**



### F.集合属性

**在 Spring中可以通过一组内置的 xml 标签(例如: <list>, <set> 或 <map>) 来配置集合属性.**

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>

配置 java.util.Set 需要使用 <set> 标签, 定义元素的方法与 List 一样.

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

使用 <props> 定义 java.util.Properties, 该标签使用多个 <prop> 作为子标签. 每个 <prop> 标签必须定义 key 属性

### XML 配置里的 Bean 自动装配

Spring IOC 容器可以自动装配 Bean. 需要做的仅仅是在 <bean> 的 autowire 属性里指定自动装配的模式

byType(根据类型自动装配): 若 IOC 容器中有多个与目标 Bean 类型一致的 Bean. 在这种情况下, 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.

byName(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.

constructor(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

**XML 配置里的 Bean 自动装配的缺点**

在 Bean 配置文件里设置 autowire 属性进行自动装配将会装配 Bean 的所有属性. 然而, 若只希望装配个别属性时, autowire 属性就不够灵活了.

autowire 属性要么根据类型自动装配, 要么根据名称自动装配, 不能两者兼而有之.

一般情况下，在实际的项目中很少使用自动装配功能，因为和自动装配功能所带来的好处比起来，明确清晰的配置文档更有说服力一些

### 继承 Bean 配置

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

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

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

### 依赖 Bean 配置

Spring 允许用户通过 depends-on 属性设定 Bean 前置依赖的Bean，前置依赖的 Bean 会在本 Bean 实例化之前创建好

如果前置依赖于多个 Bean，则可以通过逗号，空格或的方式配置 Bean 的名称

### 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.**

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.**



### 使用外部属性文件

在配置文件里配置 Bean 时, 有时需要在 Bean 的配置里混入系统部署的细节信息(例如: 文件路径, 数据源配置信息等). 而这些部署细节实际上需要和 Bean 配置相分离

**Spring 提供了一个 PropertyPlaceholderConfigurer 的 BeanFactory 后置处理器, 这个处理器允许用户将 Bean 配置的部分内容外移到属性文件中. 可以在 Bean 配置文件里使用形式为 ${var} 的变量, PropertyPlaceholderConfigurer 从属性文件里加载属性, 并使用这些属性来替换变量.**

Spring 还允许在属性文件中使用 ${propName}，以实现属性之间的相互引用。

**注册 PropertyPlaceholderConfigurer**

**Spring 2.5 之后: 可通过 <context:property-placeholder> 元素简化:**

**<beans> 中添加 context Schema 定义**

**在配置文件中加入如下配置:**



### Spring表达式语言：SpEL

Spring 表达式语言（简称SpEL）：是一个支持运行时查询和操作对象图的强大的表达式语言。

语法类似于 EL：SpEL 使用 #{…} 作为定界符，所有在大框号中的字符都将被认为是 SpEL

SpEL 为 bean 的属性进行动态赋值提供了便利

**通过 SpEL 可以实现：**

**通过 bean 的 id 对 bean 进行引用**

**调用方法以及引用对象中的属性**

**计算表达式的值**

**正则表达式的匹配**

**字面量**

字面量的表示：

整数：<property name="count" value="#{5}"/>

小数：<property name="frequency" value="#{89.7}"/>

科学计数法：<property name="capacity" value="#{1e4}"/>

String可以使用单引号或者双引号作为字符串的定界符号：<property name=“name” value="#{'Chuck'}"/> 或 <property name='name' value='#{"Chuck"}'/>

Boolean：<property name="enabled" value="#{false}"/>

**SpEL：引用 Bean、属性和方法（1）**

引用其他对象：

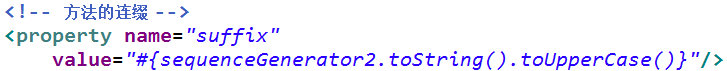


引用其他对象的属性



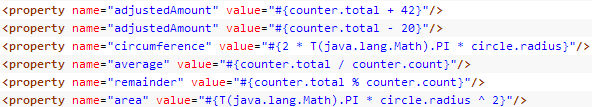
调用其他方法，还可以链式操作





**SpEL支持的运算符号（1）**

算数运算符：+, -, \*, /, %, ^：



加号还可以用作字符串连接：



比较运算符： <, >, ==, <=, >=, lt, gt, eq, le, ge





逻辑运算符号： and, or, not, |



if-else 运算符：?: (ternary), ?: (Elvis)



if-else 的变体



正则表达式：matches



调用静态方法或静态属性：通过 T() 调用一个类的静态方法，它将返回一个 Class Object，然后再调用相应的方法或属性：



### IOC 容器中 Bean 的生命周期方法

Spring IOC 容器可以管理 Bean 的生命周期, Spring 允许在 Bean 生命周期的特定点执行定制的任务.

**Spring IOC 容器对 Bean 的生命周期进行管理的过程:**

**通过构造器或工厂方法创建 Bean 实例**

**为 Bean 的属性设置值和对其他 Bean 的引用**

**调用 Bean 的初始化方法**

**Bean 可以使用了**

**当容器关闭时, 调用 Bean 的销毁方法**

在 Bean 的声明里设置 **init-method 和 destroy-method** 属性, 为 Bean 指定初始化和销毁方法.

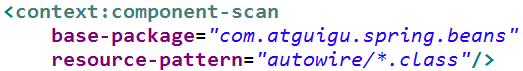
### N.在 classpath 中扫描组件

**当在组件类上使用了特定的注解之后, 还需要在 Spring 的配置文件中声明 <context:component-scan> ：**

**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里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，示例：

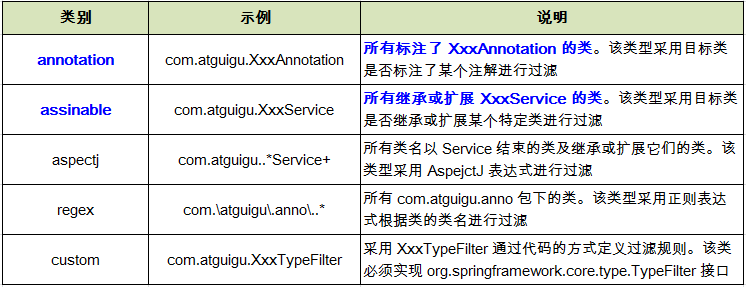


<context:include-filter> 子节点表示要包含的目标类

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

<context:component-scan> 下可以拥有若干个 <context:include-filter> 和 <context:exclude-filter> 子节点

**<context:include-filter> 和 <context:exclude-filter> 子节点支持多种类型的过滤表达式：**



### O.使用 @Autowired 自动装配 Bean

@Autowired 注解自动装配具有兼容类型的单个 Bean属性

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@Authwired 注解

**默认情况下, 所有使用 @Authwired 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 @Authwired 注解的 required 属性为 false**

**默认情况下, 当 IOC 容器里存在多个类型兼容的 Bean 时, 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 允许对方法的入参标注 @Qualifiter 已指定注入 Bean 的名称**

@Authwired 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.

@Authwired 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.

@Authwired 注解用在 java.util.Map 上时, 若该 Map 的键值为 String, 那么 Spring 将自动装配与之 Map 值类型兼容的 Bean, 此时 Bean 的名称作为键值

## Spring AOP

### AOP 简介

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.

AOP 的主要编程对象是切面(aspect), 而切面模块化横切关注点.

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对象(切面)里.

**AOP 的好处:**

**每个事物逻辑位于一个位置, 代码不分散, 便于维护和升级**

**业务模块更简洁, 只包含核心业务代码.**

**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**

### AOP 术语

**切面(Aspect):**  横切关注点(跨越应用程序多个模块的功能)被模块化的特殊对象

**通知(Advice):** 切面必须要完成的工作

**目标(Target):** 被通知的对象

**代理(Proxy):** 向目标对象应用通知之后创建的对象

**连接点（Joinpoint）：**程序执行的某个特定位置：如类某个方法调用前、调用后、方法抛出异常后等。连接点由两个信息确定：方法表示的程序执行点；相对点表示的方位。例如 ArithmethicCalculator#add() 方法执行前的连接点，执行点为 ArithmethicCalculator#add()； 方位为该方法执行前的位置

**切点（pointcut）：**每个类都拥有多个连接点：例如 ArithmethicCalculator 的所有方法实际上都是连接点，即连接点是程序类中客观存在的事务。AOP 通过切点定位到特定的连接点。类比：连接点相当于数据库中的记录，切点相当于查询条件。切点和连接点不是一对一的关系，一个切点匹配多个连接点，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。

### 在 Spring 中启用 AspectJ 注解支持

要在 Spring 应用中使用 AspectJ 注解, 必须在 classpath 下包含 AspectJ 类库: aopalliance.jar、aspectj.weaver.jar 和 spring-aspects.jar

将 aop Schema 添加到 <beans> 根元素中.

要在 Spring IOC 容器中启用 AspectJ 注解支持, 只要在 Bean 配置文件中定义一个空的 XML 元素 <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.

### 用 AspectJ 注解声明切面

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. 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.

通知是标注有某种注解的简单的 Java 方法.

AspectJ 支持 5 种类型的通知注解:

**@Before: 前置通知, 在方法执行之前执行**

**@After: 后置通知, 在方法执行之后执行**

**@AfterRunning: 返回通知, 在方法返回结果之后执行**

**@AfterThrowing: 异常通知, 在方法抛出异常之后**

**@Around: 环绕通知, 围绕着方法执行**

### 利用方法签名编写 AspectJ 切入点表达式

最典型的切入点表达式时根据方法的签名来匹配各种方法:

execution \* com.atguigu.spring.ArithmeticCalculator.\*(..): 匹配 ArithmeticCalculator 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. .. 匹配任意数量的参数. 若目标类与接口与该切面在同一个包中, 可以省略包名.

execution public \* ArithmeticCalculator.\*(..): 匹配 ArithmeticCalculator 接口的所有公有方法.

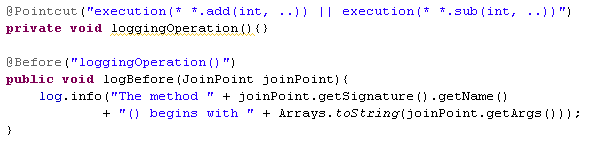
execution public double ArithmeticCalculator.\*(..): 匹配 ArithmeticCalculator 中返回 double 类型数值的方法

execution public double ArithmeticCalculator.\*(double, ..): 匹配第一个参数为 double 类型的方法, .. 匹配任意数量任意类型的参数

execution public double ArithmeticCalculator.\*(double, double): 匹配参数类型为 double, double 类型的方法.

### 合并切入点表达式

在 AspectJ 中, 切入点表达式可以通过操作符 &&, ||, ! 结合起来.

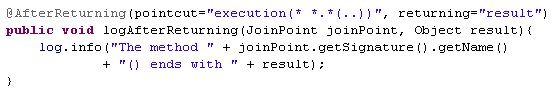


**可以在通知方法中声明一个类型为 JoinPoint 的参数. 然后就能访问链接细节. 如方法名称和参数值.**

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.

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.

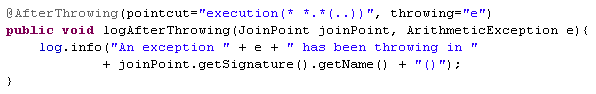
原始的切点表达式需要出现在 pointcut 属性中



只在连接点抛出异常时才执行异常通知

**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超类. 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.



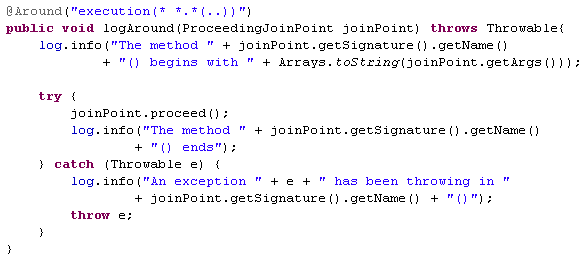
### 环绕通知

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, 否则会出现空指针异常**



### 指定切面的优先级

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, 序号出现在注解中



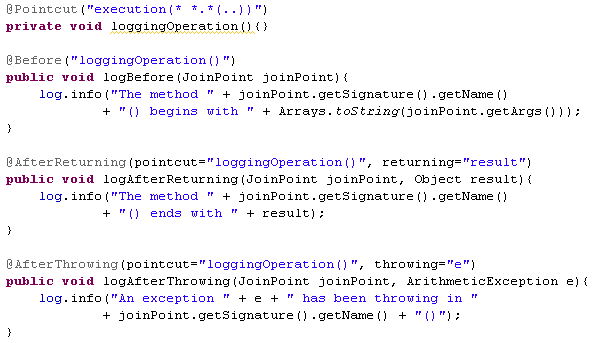
### 重用切入点定义（pointcut）

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.

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.

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.

其他通知可以通过方法名称引入该切入点.



### 用基于 XML 的配置声明切面

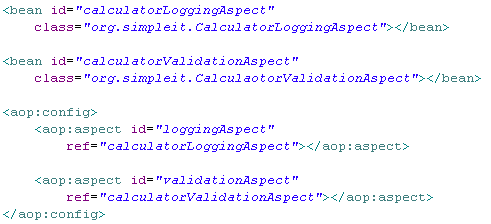
除了使用 AspectJ 注解声明切面, Spring 也支持在 Bean 配置文件中声明切面. 这种声明是通过 aop schema 中的 XML 元素完成的.

**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.**

当使用 XML 声明切面时, 需要在 <beans> 根元素中导入 aop Schema

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 必须有一个标示符, 供 <aop:aspect> 元素引用



### 基于 XML ---- 声明切入点

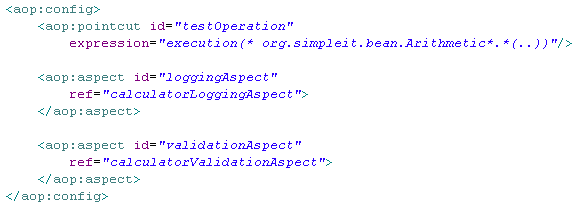
切入点使用 <aop:pointcut> 元素声明

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.



## spring事务

### 事务的属性（ACID）

**原子性(atomicity):** 事务是一个原子操作, 由一系列动作组成. 事务的原子性确保动作要么全部完成要么完全不起作用.

**一致性(consistency):** 一旦所有事务动作完成, 事务就被提交. 数据和资源就处于一种满足业务规则的一致性状态中.

**隔离性(isolation):** 可能有许多事务会同时处理相同的数据, 因此每个事物都应该与其他事务隔离开来, 防止数据损坏.

**持久性(durability):** 一旦事务完成, 无论发生什么系统错误, 它的结果都不应该受到影响. 通常情况下, 事务的结果被写到持久化存储器中

### B.Spring 中的事务管理

作为企业级应用程序框架, Spring 在不同的事务管理 API 之上定义了一个抽象层. 而应用程序开发人员不必了解底层的事务管理 API, 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.**

**编程式事务管理:** 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. 在编程式管理事务时, 必须在每个事务操作中包含额外的事务管理代码.

**声明式事务管理:** 大多数情况下比编程式事务管理更好用. 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. 事务管理作为一种横切关注点, 可以通过 AOP 方法模块化. Spring 通过 Spring AOP 框架支持声明式事务管理.

**Spring 的核心事务管理抽象是** 

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.

### C.Spring 中的事务管理器的不同实现

**1.在应用程序中只需要处理一个数据源, 而且通过 JDBC 存取**

****

**2.在 JavaEE 应用服务器上用 JTA(Java Transaction API) 进行事务管理**

****

**3.用 Hibernate 框架存取数据库**

****

**4.事务管理器以普通的 Bean 形式声明在 Spring IOC 容器中**

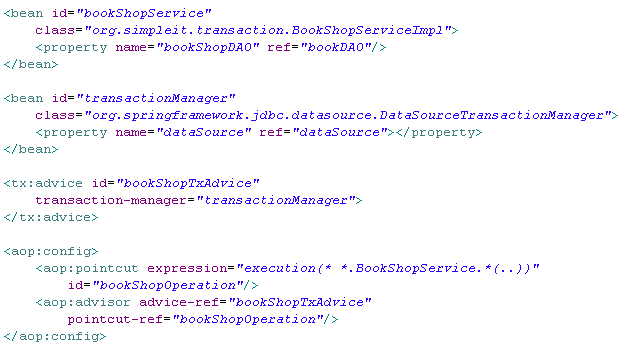
### D.用事务通知声明式地管理事务

事务管理是一种横切关注点

为了在 Spring 2.x 中启用声明式事务管理, 可以通过 tx Schema 中定义的 <tx:advice> 元素声明事务通知, 为此必须事先将这个 Schema 定义添加到 <beans> 根元素中去.

声明了事务通知后, 就需要将它与切入点关联起来. 由于事务通知是在 <aop:config> 元素外部声明的, 所以它无法直接与切入点产生关联. 所以必须在 <aop:config> 元素中声明一个增强器通知与切入点关联起来.

由于 Spring AOP 是基于代理的方法, 所以只能增强公共方法. 因此, 只有公有方法才能通过 Spring AOP 进行事务管理.



### 用 @Transactional 注解声明式地管理事务

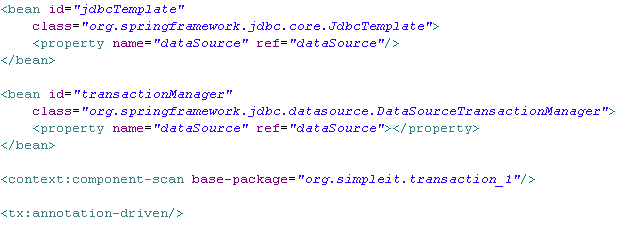
除了在带有切入点, 通知和增强器的 Bean 配置文件中声明事务外, Spring 还允许简单地用 @Transactional 注解来标注事务方法.

**为了将方法定义为支持事务处理的, 可以为方法添加 @Transactional 注解. 根据 Spring AOP 基于代理机制, 只能标注公有方法.**

**可以在方法或者类级别上添加 @Transactional 注解. 当把这个注解应用到类上时, 这个类中的所有公共方法都会被定义成支持事务处理的.**

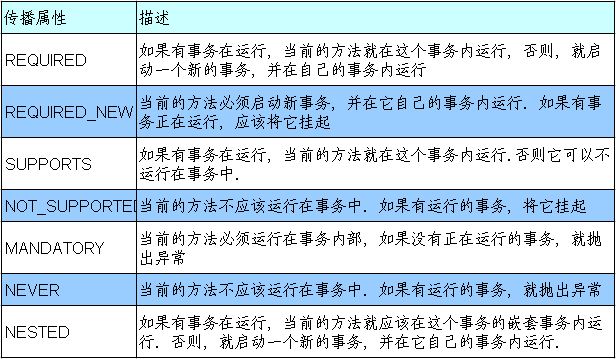
**在 Bean 配置文件中只需要启用 <tx:annotation-driven> 元素, 并为之指定事务管理器就可以了.**

如果事务处理器的名称是 transactionManager, 就可以在<tx:annotation-driven> 元素中省略 transaction-manager 属性. 这个元素会自动检测该名称的事务处理器.



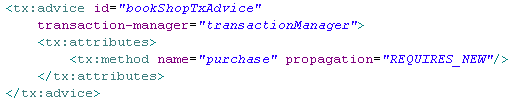
### \*F. Spring 支持的事务传播行为

**默认为required**





### G.基于xml、 Spring 2.x 配置事务通知中配置传播属性



### H.并发事务所导致的问题

**脏读:** 对于两个事物 T1, T2, T1 读取了已经被 T2 更新但 还没有被提交的字段. 之后, 若 T2 回滚, T1读取的内容就是临时且无效的.

**不可重复读**:对于两个事物 T1, T2, T1 读取了一个字段, 然后 T2 更新了该字段. 之后, T1再次读取同一个字段, 值就不同了.

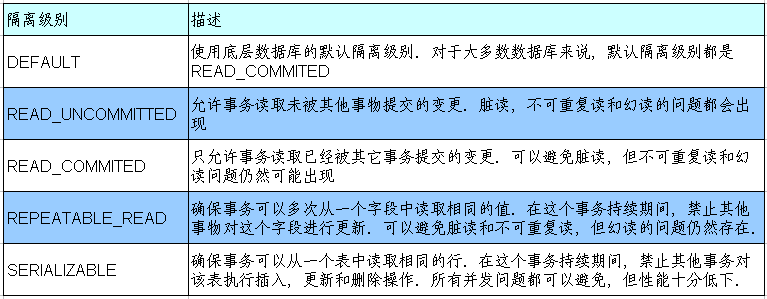
**幻读:**对于两个事物 T1, T2, T1 从一个表中读取了一个字段, 然后 T2 在该表中插入了一些新的行. 之后, 如果 T1 再次读取同一个表, 就会多出几行.

### \*I. 事务的隔离级别

**从理论上来说, 事务应该彼此完全隔离, 以避免并发事务所导致的问题. 然而, 那样会对性能产生极大的影响, 因为事务必须按顺序运行.**

在实际开发中, 为了提升性能, 事务会以较低的隔离级别运行.

事务的隔离级别可以通过隔离事务属性指定



**事务的隔离级别要得到底层数据库引擎的支持, 而不是应用程序或者框架的支持.**

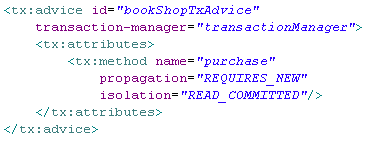
**Oracle 支持的 2 种事务隔离级别：READ\_COMMITED , SERIALIZABLE**

**Mysql 支持 4 中事务隔离级别.**

**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.**



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



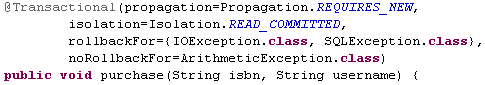
### J.设置回滚事务属性

**默认情况下只有未检查异常(RuntimeException和Error类型的异常)会导致事务回滚. 而受检查异常不会.**

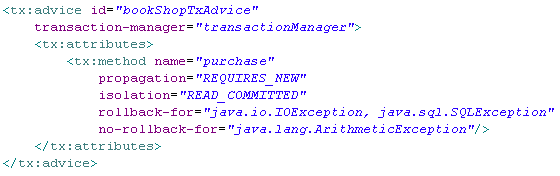
事务的回滚规则可以通过 @Transactional 注解的 rollbackFor 和 noRollbackFor 属性来定义. 这两个属性被声明为 Class[] 类型的, 因此可以为这两个属性指定多个异常类.

**rollbackFor: 遇到时必须进行回滚**

**noRollbackFor: 一组异常类，遇到时必须不回滚**



**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, 用逗号分隔.**



### K. 超时和只读属性

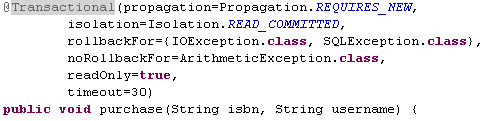
由于事务可以在行和表上获得锁, 因此长事务会占用资源, 并对整体性能产生影响.

如果一个事物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.**

**超时和只读属性可以在 @Transactional 注解中定义.超时属性以秒为单位来计算.**



**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.**

