## 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。

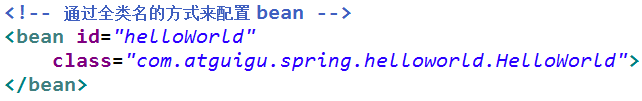
## 配置 bean

### 1. 配置形式：

#### a.基于 XML 文件的方式

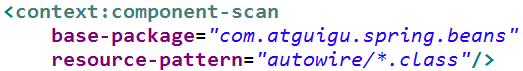
<1>在 xml 文件中通过 bean 节点来配置 bean.

* <2>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

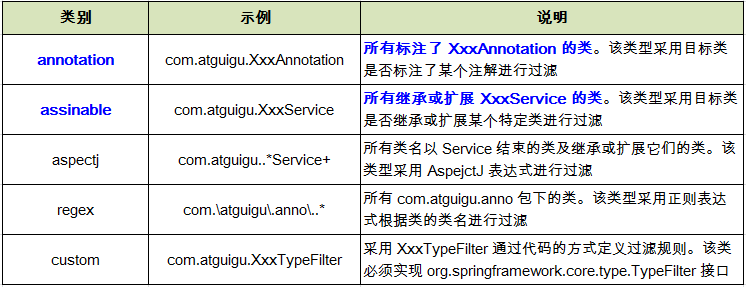


#### b.基于注解的方式

* 组件扫描(component scanning): Spring 能够从 classpath 下自动扫描, 侦测和实例化具有特定注解的组件.
* 特定组件包括:
  + @Component: 基本注解, 标识了一个受 Spring 管理的组件
  + @Respository: 标识持久层组件
  + @Service: 标识服务层(业务层)组件
  + @Controller: 标识表现层组件
* 对于扫描到的组件, **Spring 有默认的命名策略**: 使用非限定类名, 第一个字母小写**. 也可以在注解中通过 value 属性值标识组件的名称**
* <context:component-scan> 元素还会自动注册 AutowiredAnnotationBeanPostProcessor 实例, 该实例可以自动装配具有 **@Autowired 和 @Resource 、@Inject**注解的属性.
* @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 还支持 @Resource 和 @Inject 注解，这两个注解和 @Autowired 注解的功用类似
* @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
* @Inject 和 @Autowired 注解一样也是按类型匹配注入的 Bean， 但没有 reqired 属性
* **建议使用 @Autowired 注解**
* 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：



### 2. Bean 的配置方式：

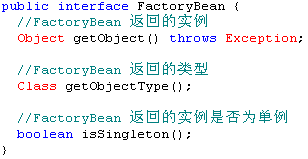
#### a.通过全类名（反射）

#### b.通过工厂方法（静态工厂方法 & 实例工厂方法）

* 调用**静态工厂方法**创建 Bean是将**对象创建的过程封装到静态方法中**. 当客户端需要对象时, 只需要简单地调用静态方法, 而不同关心创建对象的细节.
* 要声明通过静态方法创建的 Bean, 需要在 Bean 的 **class** 属性里指定拥有该工厂的方法的类, 同时在 **factory-method** 属性里指定工厂方法的名称. 最后, 使用 **<constrctor-arg>** 元素为该方法传递方法参数.
* **实例工厂方法**: **将对象的创建过程封装到另外一个对象实例的方法里**. 当客户端需要请求对象时, 只需要简单的调用该实例方法而不需要关心对象的创建细节.
* 要声明通过实例工厂方法创建的 Bean
  + 在 bean 的 **factory-bean** 属性里指定拥有该工厂方法的 Bean
  + 在 **factory-method** 属性里指定该工厂方法的名称
  + 使用 **construtor-arg** 元素为工厂方法传递方法参数

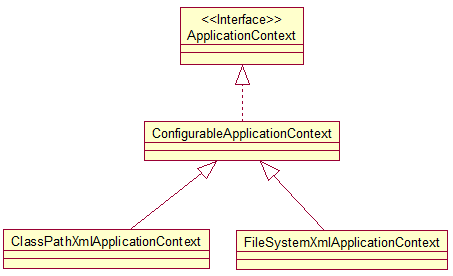
#### c.FactoryBean

* 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



### 3. IOC 容器 BeanFactory & ApplicationContext 概述

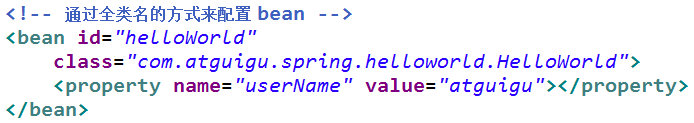
* 在 **Spring IOC 容器**读取 Bean 配置创建 Bean 实例之前, 必须对它进行实例化. 只有在容器实例化后, 才可以从 IOC 容器里获取 Bean 实例并使用.
* Spring 提供了两种类型的 IOC 容器实现.
  + **BeanFactory**: IOC 容器的基本实现.
  + **ApplicationContext**: 提供了更多的高级特性. 是 BeanFactory 的子接口.
  + BeanFactory 是 Spring 框架的基础设施，面向 Spring 本身；ApplicationContext 面向使用 Spring 框架的开发者，**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。
* 调用 ApplicationContext 的 getBean() 方法。

### 4. 依赖注入的方式：

#### a.属性注入

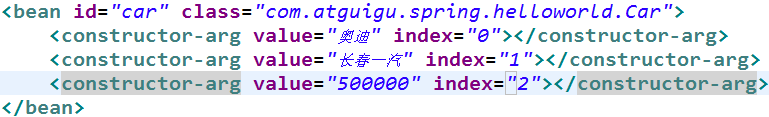


* 属性注入即通过 **setter 方法**注入Bean 的属性值或依赖的对象
* 属性注入使用 <property> 元素, 使用 name 属性指定 Bean 的属性名称，value 属性或 <value> 子节点指定属性值

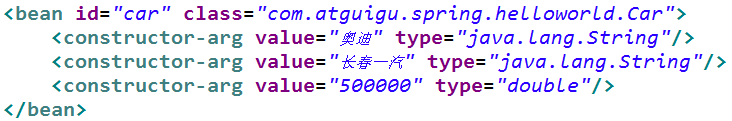
**属性注入是实际应用中最常用的注入方式**

#### b.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：



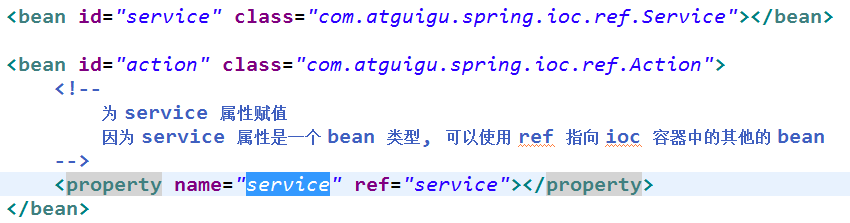
* 按类型匹配入参：



### 5. 注入属性值细节

#### a .引用其它 Bean

* 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,内部 Bean 不能使用在任何其他地方。**
* 可以使用专用的 **<null/>** 元素标签为 Bean 的字符串或其它对象类型的属性注入 null 值
* 和 Struts、Hiberante 等框架一样，**Spring 支持级联属性的配置**。
* 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.
* 使用基本的集合标签定义集合时, **不能将集合作为独立的 Bean 定义, 导致其他 Bean 无法引用该集合, 所以无法在不同 Bean 之间共享集合**.
* 可以使用 util schema 里的集合标签定义独立的集合 Bean. 需要注意的是, 必须在 <beans> 根元素里添加 util schema 定义
* 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

### 6. 自动转配

### 7. bean 之间的关系：

#### a.继承

* **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**

#### b.依赖

* **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**

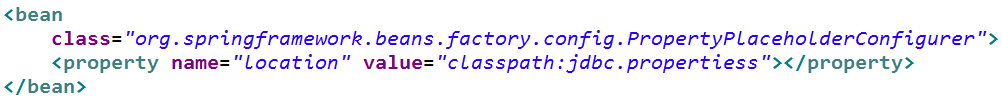
### 8. bean 的作用域：

* 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.

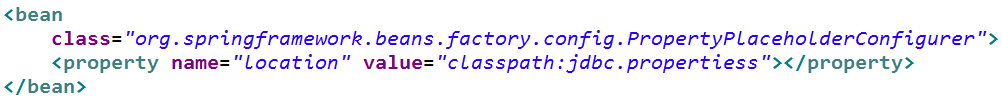


### 9. 使用外部属性文件

* 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。
* Spring 2.0:



* **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:



### 10. spEL

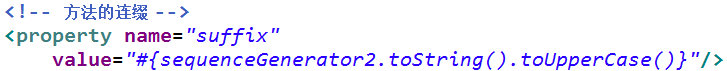
* **Spring 表达式语言**（简称**SpEL**）：是一个**支持运行时查询和操作对象图的强大的表达式语言**。
* **语法类似于 EL**：**SpEL 使用 #{…} 作为定界符，所有在大框号中的字符都将被认为是 SpEL**
* **SpEL 为 bean 的属性进行动态赋值提供了便利**
* 通过 SpEL 可以实现：
  + 通过 bean 的 id 对 bean 进行引用
  + 调用方法以及引用对象中的属性
  + 计算表达式的值
  + 正则表达式的匹配
* 字面量的表示：
  + 整数：<property name="count" value="**#{5}**"/>
  + 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}**"/>
* **引用其他对象：**



* **引用其他对象的属性**

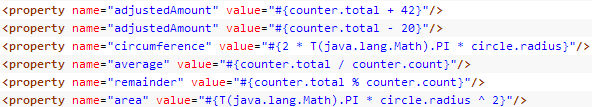


* **调用其他方法，还可以链式操作**





* **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**

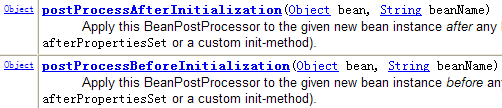


* **调用静态方法或静态属性**：通过 **T()** 调用一个类的静态方法，它将返回一个 Class Object，然后再调用相应的方法或属性：



### 11. IOC 容器中 Bean 的生命周期

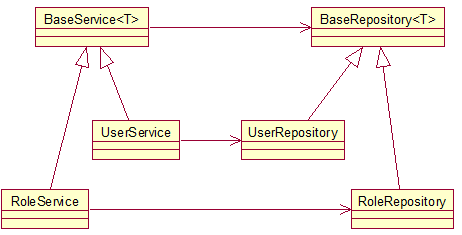
* **Spring IOC 容器可以管理 Bean 的生命周期**, Spring 允许在 Bean 生命周期的特定点执行定制的任务.
* Spring IOC 容器对 Bean 的生命周期进行管理的过程:
  + 通过构造器或工厂方法创建 Bean 实例
  + 为 Bean 的属性设置值和对其他 Bean 的引用
  + **调用 Bean 的初始化方法**
  + Bean 可以使用了
  + **当容器关闭时, 调用 Bean 的销毁方法**
* 在 Bean 的声明里设置 init-method 和 destroy-method 属性, 为 Bean 指定初始化和销毁方法.
* **Bean 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.
* 对Bean 后置处理器而言, 需要实现 接口. 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:



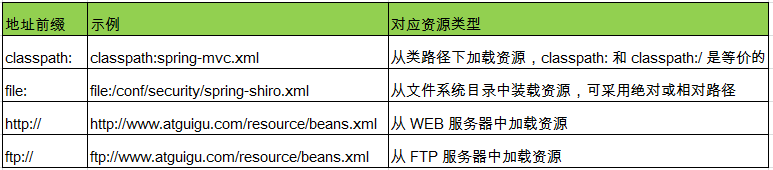
* Spring IOC 容器对 Bean 的生命周期进行管理的过程:
  + 通过构造器或工厂方法创建 Bean 实例
  + 为 Bean 的属性设置值和对其他 Bean 的引用
  + **将 Bean 实例传递给 Bean 后置处理器的 postProcessBeforeInitialization 方法**
  + 调用 Bean 的初始化方法
  + **将 Bean 实例传递给 Bean 后置处理器的 postProcessAfterInitialization方法**
  + Bean 可以使用了
  + 当容器关闭时, 调用 Bean 的销毁方法

### 12. Spring 4.x 新特性：泛型依赖注入

* Spring 4.x 中可以为子类注入子类对应的泛型类型的成员变量的引用



* Spring 允许通过 <import> 将多个配置文件引入到一个文件中，进行配置文件的集成。这样在启动 Spring 容器时，仅需要指定这个合并好的配置文件就可以。
* import 元素的 resource 属性支持 Spring 的标准的路径资源



## 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.

### 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。

### 3.AOP使用方法

1.SpringAOP

1).加入jar包：

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

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

2).在配置文件中加入aop的命名空间

3).基于注解方式：

a.在配置文件中加入如下配置

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

b.把横切关注点的代码抽象到切面的类中。

i.切面首先是一个IOC中的bean，即加入@Component注解

ii.切面还需要加入@Aspect注解

c.在类中声明各种通知

i.声明一个方法

ii.在方法前加入@Before注解

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

### 4.在 Spring 中启用 AspectJ 注解支持

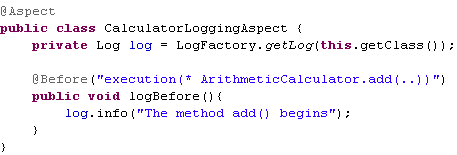
* 要在 Spring 应用中使用 AspectJ 注解, **必须在 classpath 下包含 AspectJ 类库**: aopalliance.jar、aspectj.weaver.jar 和 spring-aspects.jar
* **将 aop Schema 添加到 <beans> 根元素中**.
* 要在 Spring IOC 容器中启用 AspectJ 注解支持, 只要**在 Bean 配置文件中定义一个空的 XML 元素 <aop:aspectj-autoproxy>**
* 当 Spring IOC 容器侦测到 Bean 配置文件中的 <aop:aspectj-autoproxy> 元素时, 会自动为与 AspectJ 切面匹配的 Bean 创建代理.

### 5.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

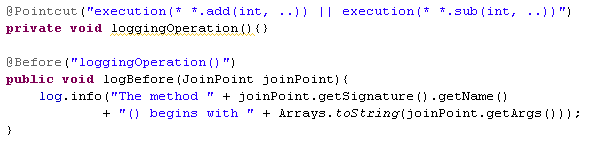
### 6.前置通知

* 前置通知:在方法执行之前执行的通知
* 前置通知使用 @Before 注解, 并将切入点表达式的值作为注解值.



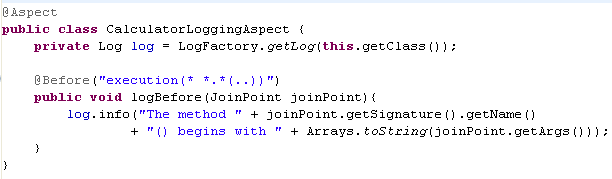
### 7.利用方法签名编写 AspectJ 切入点表达式

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



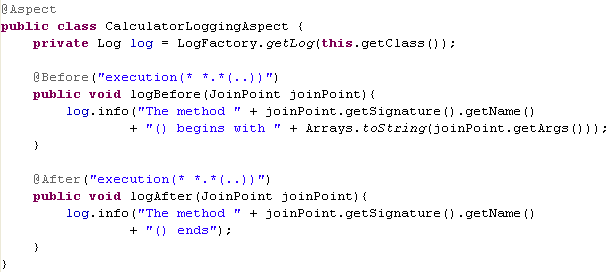
### 8.让通知访问当前连接点的细节

* 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.



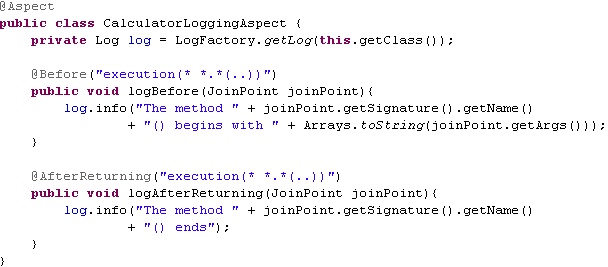
### 9.后置通知

* 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, 下面的后置通知记录了方法的终止.
* 一个切面可以包括一个或者多个通知.

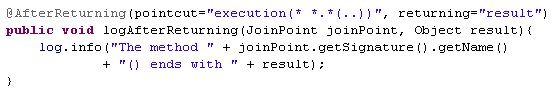


### 10.返回通知

* **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**.

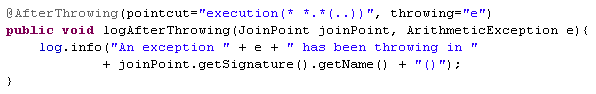


* 在返回通知中, 只要将 **returning** 属性添加到 @AfterReturning 注解中, 就可以访问连接点的返回值. 该属性的值即为用来传入返回值的参数名称.
* 必须在通知方法的签名中添加一个**同名参数**. 在运行时, Spring AOP 会通过这个参数传递返回值.
* **原始的切点表达式需要出现在 pointcut 属性中**



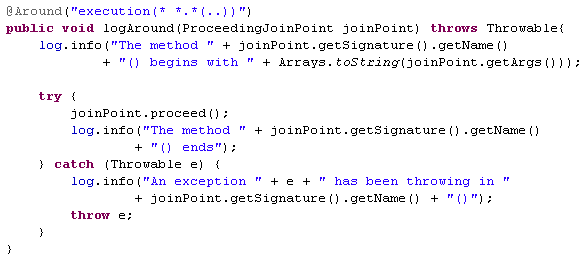
### 11.异常通知

* 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.



12.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**



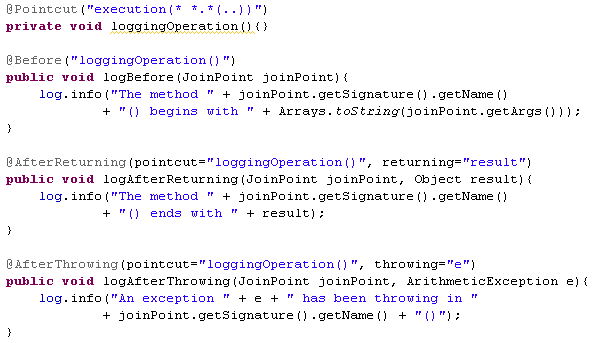
### 12.指定切面的优先级

* 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



### 13.重用切入点定义

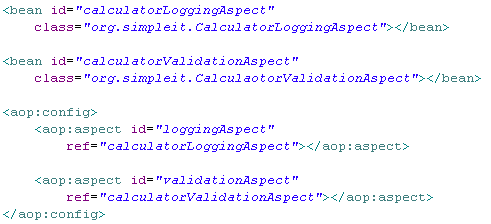
* 在编写 AspectJ 切面时, 可以直接在通知注解中书写切入点表达式. 但同一个切点表达式可能会在多个通知中重复出现.
* 在 AspectJ 切面中, 可以**通过 @Pointcut 注解将一个切入点声明成简单的方法. 切入点的方法体通常是空的**, 因为将切入点定义与应用程序逻辑混在一起是不合理的.
* **切入点方法的访问控制符同时也控制着这个切入点的可见性**. 如果切入点要在多个切面中共用, 最好将它们集中在一个公共的类中. 在这种情况下, 它们必须被声明为 public. 在引入这个切入点时, 必须将类名也包括在内. 如果类没有与这个切面放在同一个包中, 还必须包含包名.
*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



### 14.基于 XML 的配置声明切面

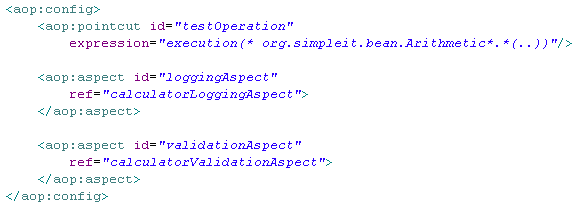
#### a.声明切面

* 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



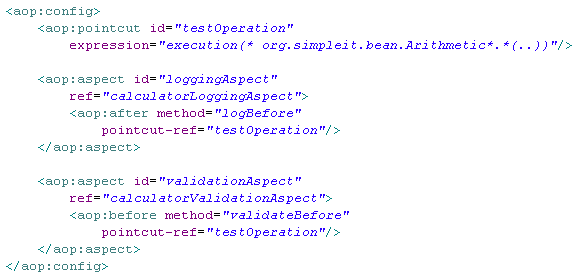
#### b.声明切入点

* 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.



#### c.声明通知

* 在 aop Schema 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.



## 事物：

### 添加事务注解

1.使用 propagation 指定事务的传播行为, 即当前的事务方法被另外一个事务方法调用时如何使用事务, 默认取值为 REQUIRED, 即使用调用方法的事务,REQUIRES\_NEW: 事务自己的事务, 调用的事务方法的事务被挂起.

2.使用 isolation 指定事务的隔离级别, 最常用的取值为 READ\_COMMITTED

3.默认情况下 Spring 的声明式事务对所有的运行时异常进行回滚. 也可以通过对的.属性进行设置. 通常情况下去默认值即可.

4.使用 readOnly 指定事务是否为只读. 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. 若真的事一个只读取数据库值的方法, 应设置 readOnly=true

5.使用 timeout 指定强制回滚之前事务可以占用的时间.

### 添加事物XML

<!-- 1. 配置事务管理器 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"></property>

</bean>

<!-- 2. 配置事务属性 -->

<tx:advice id="txAdvice" transaction-manager="transactionManager">

<tx:attributes>

<!-- 根据方法名指定事务的属性 -->

<tx:method name="purchase" propagation="REQUIRES\_NEW"/>

<tx:method name="get\*" read-only="true"/>

<tx:method name="find\*" read-only="true"/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

<!-- 3. 配置事务切入点, 以及把事务切入点和事务属性关联起来 -->

<aop:config>

<aop:pointcut expression="execution(\* com.atguigu.spring.tx.xml.service.\*.\*(..))"

id="txPointCut"/>

<aop:advisor advice-ref="txAdvice" pointcut-ref="txPointCut"/>

</aop:config>