# 三种创建对象的比较

## BeanFactory: 已过时(接口)

只有在对象被调用时候才会创建对象,在资源匮乏的情况下才会使用Beanfactory

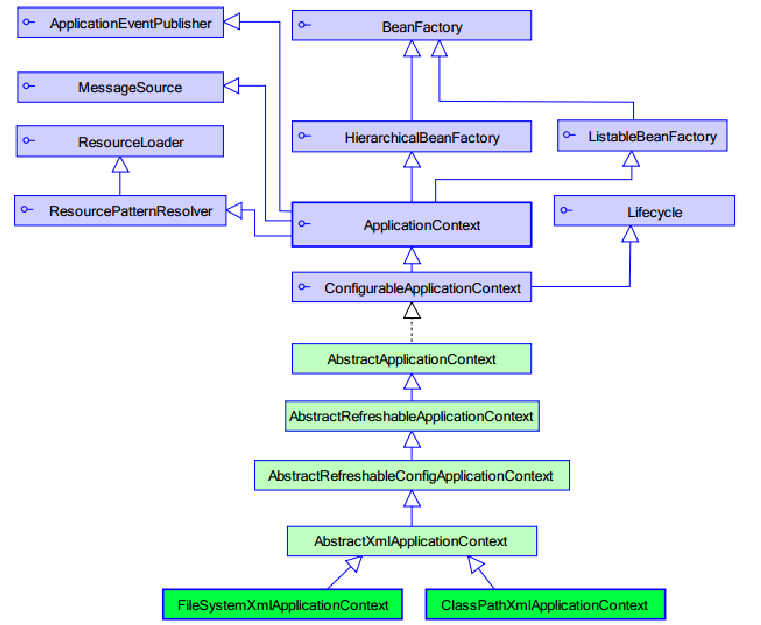
## ClassPathXmlApplicationContext:(Application接口)

加载类路径下 Spring的配置文件.

## FileSystemXmlApplicationContext:

加载本地磁盘下 Spring 的配置文件.





# Spring的Bean实例化三种方式

## 【无参数的构造方法的方式】

<bean id="bean1" class="cn.itcast.spring.demo3.Bean1"></bean>

## 【静态工厂实例化的方式】

静态类->对象id

<bean id="bean2" class="cn.itcast.spring.demo3.Bean2Factory" factory-method="getBean2"/>

## 【实例工厂实例化的方式】

工厂

<bean id="bean3Factory" class="cn.itcast.spring.demo3.Bean3Factory"></bean>

工厂bean->对象id

<bean id="bean3" factory-bean="bean3Factory" factory-method="getBean3"></bean>

# Spring 的 Bean 的属性注入

## 【构造方法的方式注入属性】

<!-- 第一种：构造方法的方式 -->

<bean id="car" class="cn.itcast.spring.demo4.Car">

<constructor-arg name="name" value="保时捷"/>

<constructor-arg name="price" value="1000000"/>

</bean>

## 【set 方法的方式注入属性】

<!-- 第二种：set 方法的方式 -->

<bean id="car2" class="cn.itcast.spring.demo4.Car2">

<property name="name" value="奇瑞 QQ"/>

<property name="price" value="40000"/>

</bean>

## 【对象类型的注入】

<!-- 注入对象类型的属性 -->

<bean id="person" class="cn.itcast.spring.demo4.Person">

<property name="name" value="会希"/>

<!-- ref 属性：引用另一个 bean 的 id 或 name -->

<property name="car2" ref="car2"/>

</bean>

## 【名称空间 p 的属性注入的方式:Spring2.x 版本后提供的方式】



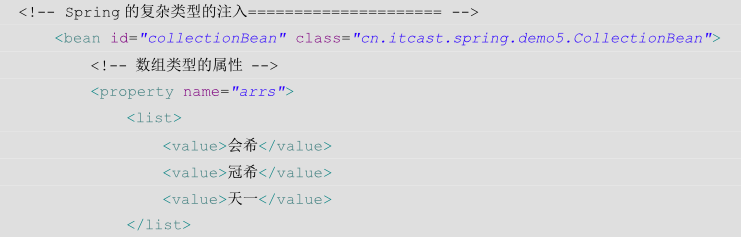


## 【SpEL 的方式的属性注入:Spring3.x 版本后提供的方式】

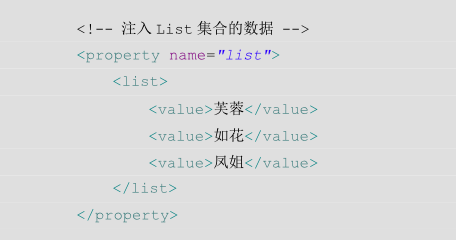


## 【注入复杂类型】

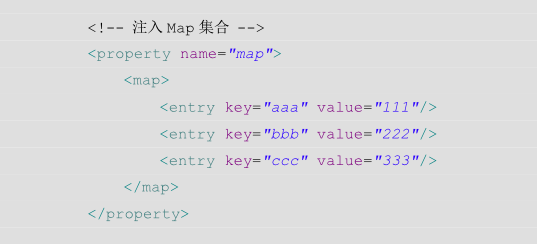
### 数组



### List



### Map



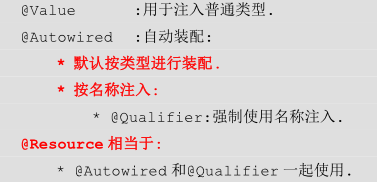
### Properties



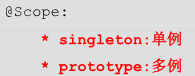
# Spring 的分配置文件的开发



# 属性注入的注解:(使用注解注入的方式,可以不用提供 set 方法.)



# Bean 的作用范围的注解



# Bean 的生命周期的配置作用于方法上

监听bean的变化



# Spring的bean的管理方式比较

# JDK 动态代理增强一个类中方法

**实现一个接口,组合方式注入被代理的对象**

**public** **class** MyJDKProxy **implements** InvocationHandler {

**private** UserDao userDao;

**public** MyJDKProxy(UserDao userDao) { **this**.userDao = userDao;

}

// 编写工具方法：生成代理：

**public** UserDao (){

UserDao userDaoProxy = (UserDao)

Proxy.*newProxyInstance*(userDao.getClass().getClassLoader(), userDao.getClass().getInterfaces(), **this**);

**return** userDaoProxy;

}

@Override

**public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable

{ **if**("save".equals(method.getName())){

System.***out***.println("权限校验================");

}

**return** method.invoke(userDao, args);

}

}

# Cglib 动态代理增强一个类中的方法

实现一个接口,组合方式注入一个被代理的对象

|  |
| --- |
| **public** **class** MyCglibProxy **implements** MethodInterceptor{  **private** CustomerDao customerDao;  **public** MyCglibProxy(CustomerDao customerDao){ **this**.customerDao = customerDao;  }  // 生成代理的方法:  **public** CustomerDao createProxy(){  // 创建Cglib的核心类:  Enhancer enhancer = **new** Enhancer();  // 设置父类:  enhancer.setSuperclass(CustomerDao.**class**);  // 设置回调:  enhancer.setCallback(**this**);  // 生成代理： |
| CustomerDao customerDaoProxy = (CustomerDao) enhancer.create(); **return** customerDaoProxy;  }    @Override   |  | | --- | | proxy |   **public** Object **intercept**(Object , Method method, Object[] args, MethodProxy  methodProxy) **throws** Throwable { **if**("delete".equals(method.getName())){   |  | | --- | | proxy |   Object obj = methodProxy.invokeSuper( , args);  System.***out***.println("日志记录================"); **return** obj;  }     |  | | --- | | proxy |   **return** methodProxy.invokeSuper( , args);  }  } |

# AOP 的开发中的相关术语:

Joinpoint(连接点):所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点.

Pointcut(切入点):所谓切入点是指我们要对哪些Joinpoint进行拦截的定义.

Advice(通知/增强):所谓通知是指拦截到Joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知(切面要完成的功能)

Introduction(引介):引介是一种特殊的通知在不修改类代码的前提下,Introduction可以在运行期为类动态地添加一些方法或Field.

Target(目标对象):代理的目标对象

Weaving(织入):是指把增强应用到目标对象来创建新的代理对象的过程. spring采用动态代理织入，而AspectJ采用编译期织入和类装在期织入

Proxy（代理）:一个类被AOP织入增强后，就产生一个结果代理类

Aspect(切面): 是切入点和通知（引介）的结合

## 引入相应的 jar 包

\* spring的传统AOP的开发的包

spring-aop-4.2.4.RELEASE.jar com.springsource.org.aopalliance-1.0.0.jar

\* aspectJ

的开发包

:

com.springsource.org.aspectj.weaver

-

1.6.8

.RELEASE.jar

spring

-

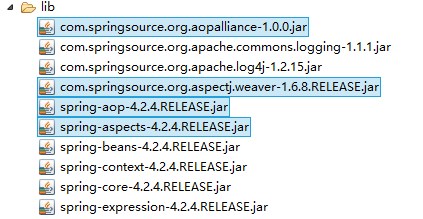
aspect

s

-

4.2.4

.RELEASE.jar



## 引入 Spring 的配置文件

|  |  |  |  |
| --- | --- | --- | --- |
| 引入AOP约束:   |  | | --- | | 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  http://www.springframework.org/schema/aop/spring-aop.xsd"> | | |
| </ | |  | | --- | | beans | | > |

## 编写目标类

|  |
| --- |
| 创建接口和类:  **public** **interface** OrderDao{  **public** **void** save();  **public** **void** update();  **public** void delete();  **public** **void** find();  }  **public** **class** OrderDaoImpl **implements** OrderDao {  @Override  **public** **void** save() {  System.***out***.println("保存订单...");  }  @Override  **public** **void** update() {  System.***out***.println("修改订单...");  }  @Override  **public** **void** delete() {  System.***out***.println("删除订单...");  }  @Override  **public** **void** find() {  System.***out***.println("查询订单...");  }  } |

## 目标类的配置

<!-- 目标类================ -->

<bean id=*"orderDao"* class=*"cn.itcast.spring.demo3.OrderDaoImpl"*> </bean>

## 整合 Junit 单元测试

引入spring-test.jar

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml") **public** **class** SpringDemo3 {

@Resource(name="orderDao") **private** OrderDao orderDao;

@Test

**public** **void** demo1(){ orderDao.save(); orderDao.update(); orderDao.delete(); orderDao.find();

}

}

## 通知类型

前置通知 ：在目标方法执行之前执行. 后置通知 ：在目标方法执行之后执行环绕通知 ：在目标方法执行前和执行后执行异常抛出通知：在目标方法执行出现 异常的时候 执行最终通知 ：无论目标方法是否出现异常 最终通知都会 执行.

## 切入点表达式

|  |
| --- |
| execution(表达式) 表达式:  [方法访问修饰符] 方法返回值 包名.类名.方法名(方法的参数)  public \* cn.itcast.spring.dao.\*.\*(..) \* cn.itcast.spring.dao.\*.\*(..)   * cn.itcast.spring.dao.UserDao+.\*(..) * cn.itcast.spring.dao..\*.\*(..) |

## 编写一个切面类

**public** **class** MyAspectXml {

// 前置增强

**public** **void** before(){

System.***out***.println("前置增强===========");

}

}

## 配置完成增强

切入点(定义拦截)指明了连接点(被拦截的方法)

切面就是将代码中指定的方法植入到被代理对象的方法中(通知方法)

|  |
| --- |
| <!-- 配置切面类:是要被植入的代码 -->  <bean id=*"myAspectXml"* class=*"cn.itcast.spring.demo3.MyAspectXml"*></bean>    <!-- 进行aop的配置 -->  <aop:config>  <!-- 配置切入点表达式:哪些类的哪些方法需要进行增强 -->  <aop:pointcut expression=*"execution(\* cn.itcast.spring.demo3.OrderDao.save(..))"* id=*"pointcut1"*/>  <!-- 配置切面 -->  <aop:aspect ref=*"myAspectXml"*>  <aop:before method=*"before"* pointcut-ref=*"pointcut1"*/>  </aop:aspect>  </aop:config> |

**1.2.7.10** 其他的增强的配置：

|  |
| --- |
| <!-- 配置切面类 -->  <bean id=*"myAspectXml"* class=*"cn.itcast.spring.demo3.MyAspectXml"*></bean>    <!-- 进行aop的配置 -->  <aop:config>  <!-- 配置切入点表达式:哪些类的哪些方法需要进行增强 -->  <aop:pointcut expression=*"execution(\* cn.itcast.spring.demo3.\*Dao.save(..))"* id=*"pointcut1"*/>  <aop:pointcut expression=*"execution(\* cn.itcast.spring.demo3.\*Dao.delete(..))"* id=*"pointcut2"*/>  <aop:pointcut expression=*"execution(\* cn.itcast.spring.demo3.\*Dao.update(..))"* id=*"pointcut3"*/>  <aop:pointcut expression=*"execution(\* cn.itcast.spring.demo3.\*Dao.find(..))"* id=*"pointcut4"*/>  <!-- 配置切面 -->  <aop:aspect ref=*"myAspectXml"*>  <aop:before method=*"before"* pointcut-ref=*"pointcut1"*/>  <aop:after-returning method=*"afterReturing"* pointcut-ref=*"pointcut2"*/>  <aop:around method=*"around"* pointcut-ref=*"pointcut3"*/>  <aop:after-throwing method=*"afterThrowing"* pointcut-ref=*"pointcut4"*/>  <aop:after method=*"after"* pointcut-ref=*"pointcut4"*/>  </aop:aspect>  </aop:config> |