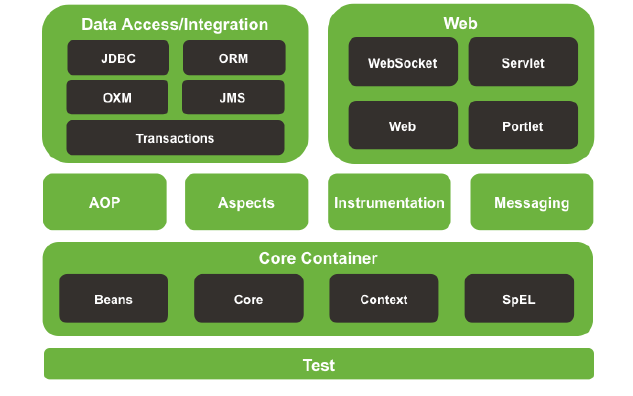
# Spring模块



## Core Container

Core：该模块式是Spring的核心功能，它提供了Spring Ioc容器的实现，这个实现被称为BeanFactory。

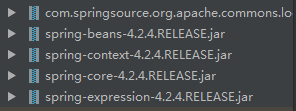
Context：这个模块构建于Core模块之上。它对Core功能进行了扩展，同时，它还提供了高级的Spring Ioc容器的实现，这个实现被称为ApplicationContext。

AOP：这个模块称为Spring AOP，Spring集成了AspectJ作为AOP的一个特定实现，同事还在JVM动态代理/CGLIB的基础上，实现了一个AOP框架，作为Spring集成其他模块的工具。

功能上：

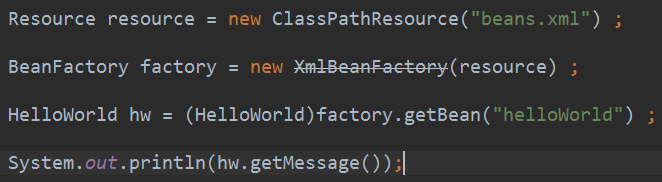
# IOC容器

需要的jar包：

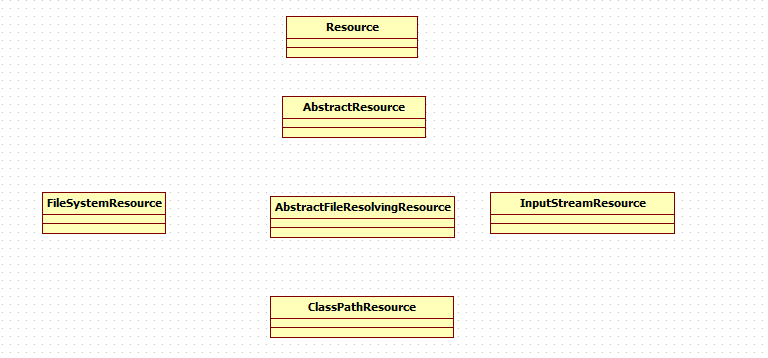


## 实例化容器

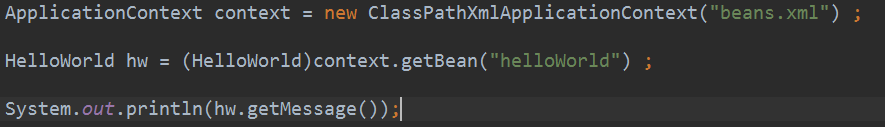
### 实例化BeanFactory



Resource实现类图：



### 实例化ApplicationContext



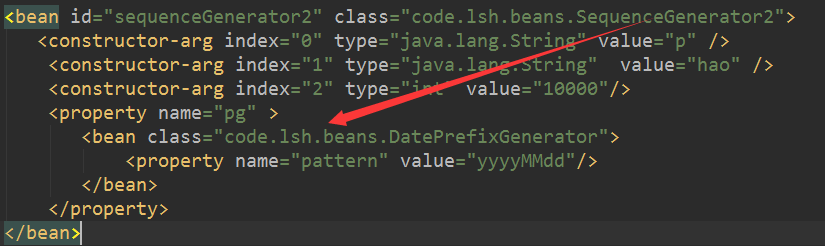
ApplicationContext的一般实现：

1. FlleSystemXmlApplicationContext用于从文件系统中加载XML配制文件。
2. XmlWebApplicationContext只能用于从Web应用程序
3. XmlPortletApplicationContext只能用于门户应用程序。

## Bean的声明方法

### 配置文件

内部bean：当Bean实例仅仅个一个特定的属性使用时，可以声明为内部Bean，内部Bean的声明在<property>和<constructor-arg>标签内。



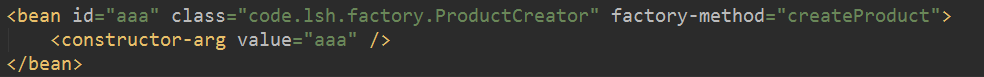
### 静态工厂方法创建Bean

将对象的创建过程封装到静态方法中，当客户端需要创建对象时，它只需要简单地调用这个静态方法就可以了，不需要关心对象的创建过程。

Spring支持静态工厂方法创建Bean，该方法在factory-method中指定。

|  |
| --- |
| public class ProductCreator {  public static Product createProduct(String productId)  {  if("aaa".equals(productId))  {  return new Product("AAA",2.5) ;  }  else if("cdrw".equals(productId))  {  return new Product("CD-RW",1.5) ;  }  throw new IllegalArgumentException("unknow product") ;  } } |

要声明静态方法创建Bean，需要在Bean的class属性里指定拥有该工厂方法的类，同时，在factory-method属性里指定工厂方法的名称，最后使用<constructor-arg>元素为该方法传递方法参数。



### 实例工厂方法创建Bean

目的是将对象的创建的过程封装到另外一个对象实例的方法里。当客户端需要创建对象时，它只需要简单地调用这个静态方法就可以了，不需要关心对象的创建过程。

|  |
| --- |
| public class ProductCreator2 {  private Map<String ,Product> products ;   public void setProducts(Map<String, Product> products){  this.products = products;  }   public Product createProduct(String productId)  {  Product product = products.get(productId) ;  if(product!=null)  {  return product ;  }  throw new IllegalArgumentException("unknow product") ;  } } |

需要在Bean中factory-bean属性里指定拥有该工厂方法的Bean实例，同时，还需要在factory-method属性里指定该工厂方法。

|  |
| --- |
| <bean id="productCreator" class="code.lsh.factory.ProductCreator2">  <property name="products">  <bean class="org.springframework.beans.factory.config.MapFactoryBean">  <property name="targetMapClass" value="java.util.HashMap"></property>  <property name="sourceMap">  <map>  <entry key="aaa">  <bean class="code.lsh.beans.Battery">  <property name="name" value="AAA"></property>  <property name="price" value="2.5"></property>  </bean>  </entry>  <entry key="cdrw">  <bean class="code.lsh.beans.Disc">  <property name="name" value="CD-RW"></property>  <property name="price" value="1.5"></property>  </bean>  </entry>  </map>  </property>  </bean>  </property> </bean> <bean id="aaa" factory-bean="productCreator" factory-method="createProduct">  <constructor-arg value="aaa" /> </bean> <bean id="cdrw" factory-bean="productCreator" factory-method="createProduct">  <constructor-arg value="cdrw" /> </bean> |

### 工厂Bean创建Bean

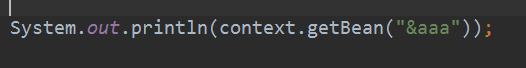
工厂Bean是一种特殊的Bean，工厂Bean与工厂方法相似，Spring在创建Bean时能够分辨出工厂Bean。工厂Bean基本要求是实现FactoryBean接口，Spring提供了一个抽象的摸板类—AbstracFactoryBean。

|  |
| --- |
| public class DiscountFactoryBean extends AbstractFactoryBean<Product> {  private Product product ;  private double discount ;   public Product getProduct(){  return product;  }   public void setProduct(Product product){  this.product = product;  }   public double getDiscount(){  return discount;  }   public void setDiscount(double discount){  this.discount = discount;  }   */\*\*  \*返回目标Bean的类型  \*/* @Override  public Class<?> getObjectType()  {  return product.getClass();  }   */\*\*  \*创建目标Bean实例  \*/* @Override  protected Product createInstance() throws Exception  {  product.setPrice(product.getPrice()\*(1-discount)) ;  return product;  } } |

配置文件：

|  |
| --- |
| <bean id="aaa" class="code.lsh.factory.DiscountFactoryBean">  <property name="product">  <bean class="code.lsh.beans.Battery">  <property name="name" value="AAA" />  <property name="price" value="2.5"></property>  </bean>  </property>  <property name="discount" value="0.2" /> </bean> <bean id="cdrw" class="code.lsh.factory.DiscountFactoryBean">  <property name="product">  <bean class="code.lsh.beans.Disc">  <property name="name" value="CD-RW"></property>  <property name="price" value="1.5"></property>  </bean>  </property>  <property name="discount" value="0.1" /> </bean> |

每当请求一个实现了FactoryBean接口是Bean时，Spring IOC都将使用这个工厂Bean创建目标Bean，并返回目标Bean。如果想要得到工厂Bean本身的实例，那么可以在Bean的前面加上“&”。





#### FactoryBean和BeanFactory的区别

查看资料：http://www.cnblogs.com/aspirant/p/9082858.html

### 静态字段声明Bean

使用内置的工厂BeanFieldRetrievingFactoryBean，还可以使用<util:contant>标签。

### 根据对象属性声明Bean

PropertyPathFactoryBean

### 注解的方式声明Bean

## 依赖检查

可以检验Bean上的某些属性是否被设置，但是属性值为null的情况无法检查。

Spring支持的依赖检查模式：

none：不执行依赖检查，所有属性都可以不设置。

simple：原始类型和集合类型的属性没有设置，则将抛出UnstaisfiedDependencyException异常。

object：对象属性没有设置，则将抛出UnstaisfiedDependencyException异常。

all：任意类型没设置，则将抛出UnstaisfiedDependencyException异常。

RequiredAnnotationBeanPostProcessor后置处理器，它会检查具有@Required注解的属性是否被设置，属性为null无能为力。

## Bean的自动装配

Spring IoC容器可以自动装配Bean，只需要在<bean>中指定autowire属性里指定自动装配的模式。

支持的装配模式：no ,byName,byType,constructor,autodetect。

byType：为每个Bean属性装配与其同类型的Bean，当IoC容器中，有多个同

Bean时，Spring将无法判断那个Bean最合适该属性，将会抛出异常。

byName：装配与其同名的Bean，更具名称装配并非适合所有情况，并不能将目标

Bean与属性名称设置为相同的。

constructor：针对每个构造器的每个参数，首先找出类型与构造器参数一致的Bean，然后再找出具有最匹配参数的构造器。该工作方式与byType相似。对于只有一个构造器的Bean，Spring将试着为每个构造器参数装配一个类型一致的Bean，但是如果Bean拥有多个构造器，这个过程将会很复杂。

@Autowired和@Resource自动装配Bean：

如果使用Bean Factory，则必须注册AutowiredAnnotationBeanPostProcessor实例

或者配置<context:annotation-config/>

@Autowired自动装配具有兼容类型的单个Bean，

@Autowired自动装配具有兼容类型的所有Bean。

用限定符通过类型的自动装配：默认情况下，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，可以和@Qulifier注解搭配，在该注解中指定Bean名称。

@Autowired

@Qulifier(“beanName”)

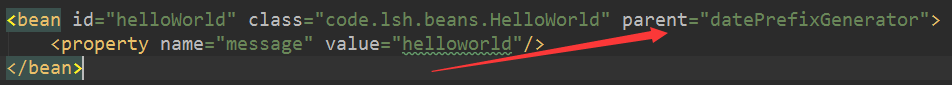
或者@Resource(name=”beanName”)

## 继承Bean

Spring允许将通用的Bean配置抽象出来，组成一个父Bean，继承这个

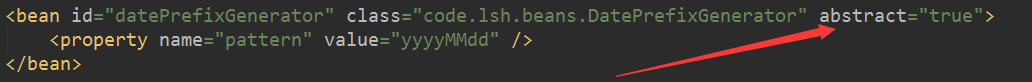
Bean的bean叫做子Bean。父Bean可以作为配置模板。

注意：并不是所有在父bean元素中定义的属性都会被继承，如：autowire



子Bean可以覆盖父Bean中的属性。

如果父Bean只作为模板使用，则父Bean必须设置为abstract属性为true。

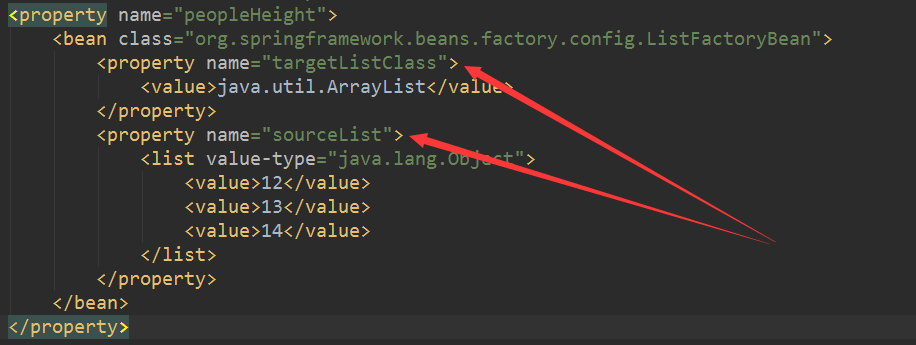


## 工厂Bean

问题：在我们使用集合标签定义集合时，我们不能指定集合的具体实现类，并且，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。

解决方法：使用工厂Bean或者使用<util:list><util:set>和<util:map>

ListFactoryBean，SetFactoryBean，MapFactoryBean。



其他类似。

工厂Bean是用来创建其他Bean。

Utility Schema：

注解扫描：能够从classpath中自动扫描，侦测和实例化具有特定注解的组件。

**基本注解：**@Component标识一般用途的组件，它标识一个受Spring管理的组件。当使用该注解时，我们可以声明单独的XML元素：<context:component-scan base-package=””>让Spring扫描这些组件，在这个元素中指定需要扫描的包，Spring会自动扫描被指定的包和子包。当存在多个需要扫描的包时，可以使用逗号（,）分割它们。注意：该元素还会注册一个AutowiredAnnotationBeanPostProcessor实例，该实例能够自动装配具有@Autowired注解的属性。

@Repository标识了持久层DAO组件。

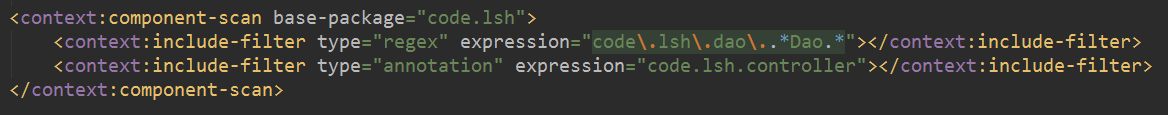
@Service标识了服务层的服务组件。

@Controller标识了表现层的控制器组件。

开启注解扫描：<context:component-scan base-package=””>

过滤扫描的注解组件：默认情况下，Spring将侦测所有使用了特定注解的类，这些注解包括@Component，@Respository，@Service和@Controller，[以及被@Component所注解的定制注解类型，还可以通过应用一个或多个include/exclude过滤器来定制扫描。](mailto:以及被@Component所注解的定制注解类型。Spring支持4)

[Spring支持4](mailto:以及被@Component所注解的定制注解类型。Spring支持4)中类型的过滤器表达式，其中annotation和assignable用于指定需要过滤的注解类型和类/接口。regex和aspectj类型则允许指定正则表达式和AspectJ切入表达式。



## 设置Bean的作用域

Spring有效的Bean的作用域：

singleton：每个IoC容器只创建一个Bean实例。

prototype：每次请求都会创建一个Bean实例。

request：每个Http请求创建一个Bean实例，这个作用域仅在Web应用程序的上下文里有效。

session：每个Http会话创建一个Bean实例，这个作用域仅在Web应用程序的上下文里有效。

## Bean的初始化和销毁

Spring不仅管理Bean的注册，而且IoC容器还管理Bean的生命周期，Spring允许在Bean的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：

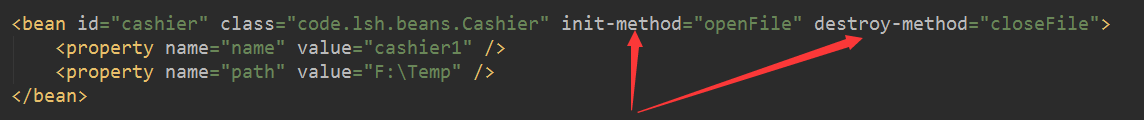
1. 通过构造方法或者工厂方法创建Bean的实例
2. 为Bean的属性设置值和对其他Bean的引用
3. 调用Bean的初始化回调方法
4. 使用Bean
5. 当容器关闭时，调用Bean的销毁回调方法。

有三种方法让Spring分辨出初始化和销毁的回调方法：

第一种：Bean可以实现InitializingBean和DisposableBean这两个生命周期接口，在afterPropertiesSet()方法中实现初始化任务，在destory()方法中实现销毁任务。

|  |
| --- |
| public class Cashier implements InitializingBean , DisposableBean {  private String name ;   private String path ;   private BufferedWriter writer ;   public String getName(){  return name;  }   public void setName(String name){  this.name = name;  }   public String getPath(){  return path;  }   public void setPath(String path){  this.path = path;  }   public BufferedWriter getWriter(){  return writer;  }   public void setWriter(BufferedWriter writer){  this.writer = writer;  }   public void openFile() throws IOException  {  File logFile = new File(path,name+".txt") ;  writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(logFile,true)));  }  public void checkout(Product product) throws IOException{  writer.write(new Date()+"\t"+product.toString()+"\r\n");  writer.flush();  }  public void closeFile() throws IOException{  writer.close();  }   */\*\*  \*实现初始化任务  \** ***@throws*** *Exception  \*/* @Override  public void afterPropertiesSet() throws Exception{  openFile();  }   */\*\*  \*实现销毁任务  \** ***@throws*** *Exception  \*/* @Override  public void destroy() throws Exception{  closeFile();  } |

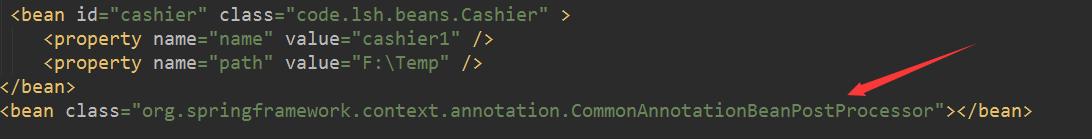
第二种：在配置文件中设置init-method和destory-method属性。



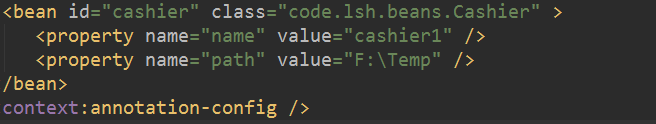
第三种：给初始化方法和销毁的回调方法添加@PostConstruct和@PreDestory注解。

之后在IoC容器中注册CommonAnnotationBeanPostProcessor的实例来调用这些回

调方法或者配置<context:annotation-config/>



或者：



## 使Bean感知容器

一个设计良好的组件不应该对它的容器产生直接的依赖，有时让Bean能够感知容器的资源也是必要的。

Spring常用的感知接口：

1. org.springframework.beans.factory.BeanNameAware：在IoC容器里配置Bean实例的名称。
2. org.springframework.beans.factory.BeanFactoryAware：当前的Bean Factory，可以调用容器的服务。
3. org.springframework.context. ResourceLoaderAware：资源加载器，通过该接口可以加在外部资源。
4. org.springframework.context.MessageSourceAware：消息源，通过该接口可以解析文本信息。
5. org.springframework.context. ApplicationEventPublisherAware：应用程序事件发布器，通过该接口可以发布应用程序的时间。

事实上，ApplicationContext接口扩展了[MessageSource](mk:@MSITStore:F:\API\Spring3.0.2-RELEASE-API.chm::/org/springframework/context/MessageSource.html)，[ApplicationEventPublisher](mk:@MSITStore:F:\API\Spring3.0.2-RELEASE-API.chm::/org/springframework/context/ApplicationEventPublisher.html)和[ResourceLoader](mk:@MSITStore:F:\API\Spring3.0.2-RELEASE-API.chm::/org/springframework/core/io/ResourceLoader.html)接口，所以只需要感知Application Context就可以访问所有的这些服务。

Spring IoC容器对Bean的生命周期进行管理的过程：

1. 通过构造方法或者工厂方法创建Bean的实例
2. 为Bean的属性设置值和对其他Bean的引用
3. 调用定义在感知接口里的setter方法。
4. 调用Bean的初始化回调方法
5. 使用Bean
6. 当容器关闭时，调用Bean的销毁回调方法。

## 创建Bean的后置处理器

Bean的后置处理器允许调用初始化回调方法前后对Bean进行额外的处理。

特点：它将对IoC容器里所有的Bean实例逐一进行处理，而不是单独一个Bean实例。

基本要求：实现BeanPostProcessor接口，Bean后置处理器可以在初始化回调方法被调用的前后对每个Bean进行处理。

在初始化回调方法被调用的前后，Spring将把每个Bean的实例传递个该接口中方法，具体过程如下：

1. 通过构造方法或者工厂方法创建Bean的实例
2. 为Bean的属性设置值和对其他Bean的引用
3. 调用定义在感知接口里的setter方法。
4. **将Bean实例传递给每个Bean后置处理器的postProcessBeforeInitialization()方法**
5. 调用Bean的初始化回调方法
6. **将Bean实例传递给每个Bean后置处理器的postProcessAfterInitialization ()方法**
7. 使用Bean
8. 当容器关闭时，调用Bean的销毁回调方法。

需求：我们在打开日志之前都要确认日志文件路径是否存在，这样可以避免FileNotFoundException异常。

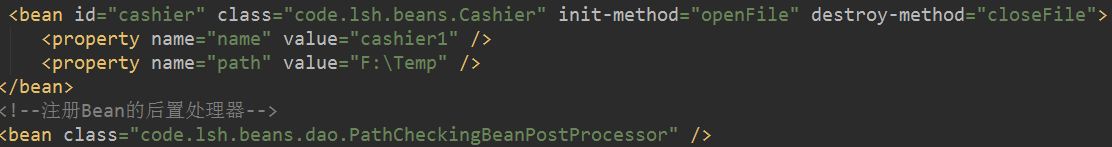
|  |
| --- |
| */\*\*  \* 定义接口，为了使Bean后置处理器能够辨别应该被检查的Bean  \*/* public interface StorageConfig {  public String getPath(); } |

|  |
| --- |
| public class Cashier implements StorageConfig {  private String name ;   private String path ;   private BufferedWriter writer ;   public String getName(){  return name;  }   public void setName(String name){  this.name = name;  }   public String getPath()  {  return path;  }   public void setPath(String path){  this.path = path;  }   public BufferedWriter getWriter(){  return writer;  }   public void setWriter(BufferedWriter writer){  this.writer = writer;  }   public void openFile() throws IOException  {  System.*out*.println("----begin openFile----");  File logFile = new File(path,name+".txt") ;  writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(logFile,true)));  System.*out*.println("----end openFile----");  }   public void checkout(Product product) throws IOException{  writer.write(new Date()+"\t"+product.toString()+"\r\n");  writer.flush();  }  public void closeFile() throws IOException{  System.*out*.println("----begin closeFile----");  writer.close();  System.*out*.println("----end closeFile----");  } } |

定义Bean的后置处理器：

|  |
| --- |
| public class PathCheckingBeanPostProcessor implements BeanPostProcessor {  @Override  public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException  {  System.*out*.println("----begin postProcessBeforeInitialization----");  if(bean instanceof StorageConfig)  {  String path = ((StorageConfig)bean).getPath() ;  File file = new File(path);  if(!file.exists())  {  file.mkdirs() ;  }  }  System.*out*.println("----end postProcessBeforeInitialization----");  return bean;  }  @Override  public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException  {  System.*out*.println("----begin postProcessAfterInitialization----");  System.*out*.println("----end postProcessAfterInitialization----");  return bean;  } } |

配置文件：



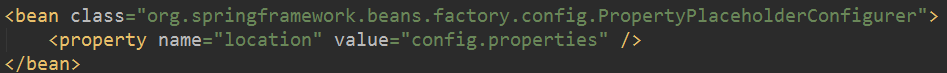
注意：如果Bean依赖于@PostConstruct和@PreDestory，还有用来调用初始化方法的CommonAnnotationBeanPostProcessor实例，则Bean的后置方法无法正常的工作。这是因为默认情况下CommonAnnotationBeanPostProcessor优先级高于Bean的后置处理器。如果要定义Bean后置处理器的处理顺序，可以实现Ordered或者PriorityOrdered接口，在getOrder()方法中返回数序值，这个方法的返回值越低则优先级越高。

## 外部化Bean配置

Spring提供了一个PropertyPlaceholderConfigurer的Bean Factory后置处理器，这个处理器允许用户将Bean配置的部分内容外部化到属性文件中，可以在Bean配置文件里使用${var}的变量，PropertyPlaceholderConfigure将从属性文件中加载属性，并使用这些属性来代替变量。

Bean Factory和Bean的后置处理器的区别时：它的目标是Ioc容器，这个容器可以是Bean Factory，也可以是ApplicationContext，**而不是Bean实例**。它会在IoC容器加载Bean配置之后，创建任何Bean实例之前起作用。

Bean Factory后置处理器的典型用途是在Bean实例化之前改变Bean的配置。



在Spring2.x以上可以通过：



简化PropertyPlaceholderConfigurer的注册。

## 解析文本信息

## 使用应用程序事件进行通信

Spring的Application Context支持Bean之间基于事件进行通信，在基于事件的通信模型里，消息发布组件只负责发布事件，不关心那些组件是消息接受者，同时消息接受者也不关心谁发布了这个事件，能够在同一时刻监听从不同组件发布的多个事件。

所有的事件类必须扩张ApplicationEvent类，接下来，任何Bean都可以调用应用程序事件发布器（application event publisher）的publishEvent()方法发布事件。

对于监听特定事件的Bean，必须实现ApplicationListener接口，并在onApplicationEvent()方法中处理事件。

实现方法：

1. 定义事件

|  |
| --- |
| import org.springframework.context.ApplicationEvent;  import java.util.Date;  public class CheckoutEvent extends ApplicationEvent {   private double amount ;   private Date time ;   public CheckoutEvent(Object source, double amount, Date time)  {  super(source);  this.amount = amount;  this.time = time;  }   public double getAmount(){  return amount;  }   public Date getTime(){  return time;  } } |

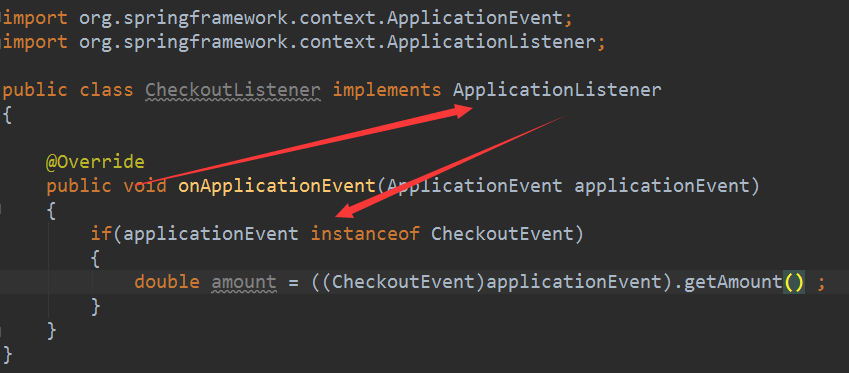
1. 发布事件

要发布事件，只需要创建一个事件的实例，同时，调用应用程序事件发布器的publishEvent()方法，可以通过实现ApplicationEventPublisherAware接口访问应用程序事件发布器。

|  |
| --- |
| import org.springframework.context.ApplicationEventPublisher; import org.springframework.context.ApplicationEventPublisherAware;  import java.util.Date;  public class Cashier implements ApplicationEventPublisherAware {  private ApplicationEventPublisher aep ;   private double total ;   @Override  public void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher)  {  this.aep = applicationEventPublisher ;  }   public void publisher()  {  CheckoutEvent event = new CheckoutEvent(this,total,new Date())  aep.publishEvent(event);  } } |

1. 监听事件

任意在Application Context里定义的Bean，只要它实现了ApplicationListener接口，那么所有的事件都会通知到它。所以，我们必须在onApplicationEvent()方法里对事件进行过滤，以得到想处理的事件



在Application Context中注册这个监听器以监听所有的事件：



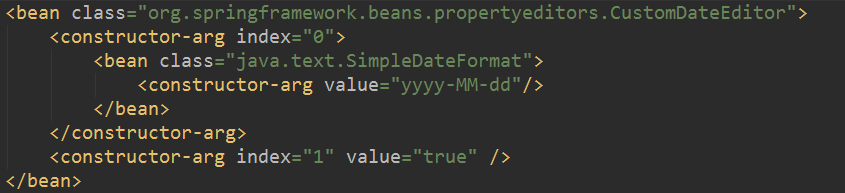
Application Context会辨别出实现了ApplicationListener接口的Bean，然后在有事件发布时通知它们。

## 注册属性编译器

属性编译器是JavaBean API的一个特性，它支持属性值和文本值之间的相互转换。每个属性编译器都是只为一个特定类型的属性所设计的。

Spring IoC容器支持使用属性编译器来简化Bean的配置

日期类型的属性编译器CustomDateEditor，该编译器可以将日期类型的字符串转化成java.util.Date类型的属性。在使用之前必须在Bean配置文件里声明它的一个实例。



在CustomEditorConfigurer实例里注册这个属性编译器。

**常用的数据类型的属性编译器：**org.springframework.beans.propertyeditors.CustomNumberEditor

org.springframework.beans.propertyeditors.ClassEditor

org.springframework.beans.propertyeditors.FileEditor

org.springframework.beans.propertyeditors.LocaleEditor

org.springframework.beans.propertyeditors.StringArrayPropertyEditor

org.springframework.beans.propertyeditors.URLEditor

## 创建定制的属性编译器

可以实现java.beans.PropertyEditor接口，也可以扩展支持类java.beans.PropertyEditorSupport。

## 加载外部资源

Spring的资源加载器提供了一个统一的getResource()方法，使用这个方法可以通过资源路径获取外部资源，可以给路径指定不同的**前缀**以从不同 的位置加载资源。

如：要从文件系统加载资源，可以使用file前缀

要从classpath中加载资源，可以使用classpath前缀

要从网络中加载资源，可以使用url作为前缀。

Spring提供了Resource通用接口，它代表一个外部资源。

|  |
| --- |
| public class BannerLoader implements ResourceLoaderAware {  private ResourceLoader resourceLoader ;   @Override  public void setResourceLoader(ResourceLoader resourceLoader)  {  this.resourceLoader = resourceLoader ;  }   public void showBanner() throws Exception  {  Resource resource = resourceLoader.getResource("file:F:\\Temp\\banner.txt") ;  try(BufferedReader reader = new BufferedReader(new InputStreamReader(resource.getInputStream())))  {  while(true)  {  String line = reader.readLine() ;  if(line!=null)  {  System.*out*.println(line);  }  else break;  }  }  } } |

Resource ResourceLoader ResourceLoaderAware ResourcePatternResolver

PathMatchingResourcePatternResolver

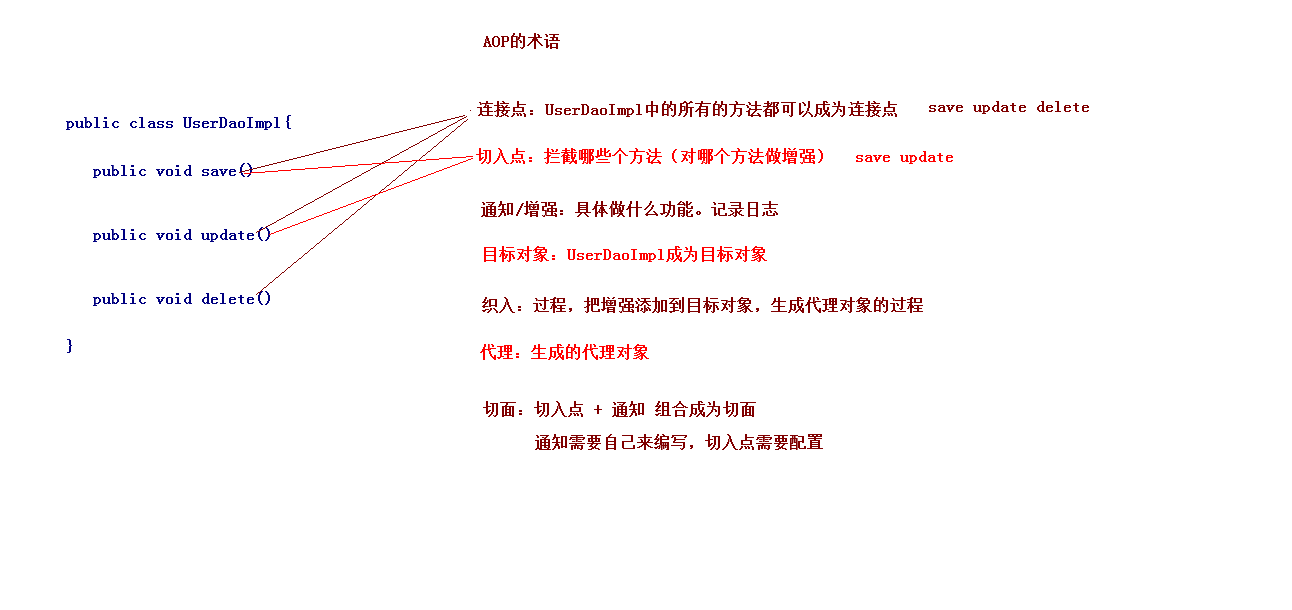
# AOP

AOP采取了横向抽取机制，取代了传统纵向继承体系重复代码（性能监控，事物管理，安全检测，缓存）

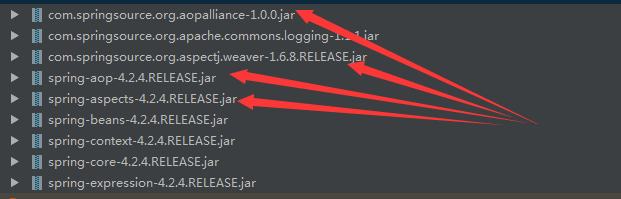
通过预编译方式和运行期动态代理实现程序功能的同意维护的一种技术。

AOP相关术语：

1. Joinpoint（连接点）：所谓的连接点是指那些被拦截的点，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。
2. Pointcut（切入点）：所谓的切入点指我们对哪些Joinpoint进行拦截的定义。
3. Advice（通知/增强）：所谓的通知是指拦截到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。
4. Introduction（引介）：引介是一种特殊的通知在不修改代码的前提下，Introduction可以在运行期为类动态地添加一些方法或Field。
5. Target（目标对象）：代理的目标对象。
6. Weaving（织入）：是指吧增强应用到目标对象来创建的代理对象的过程。
7. Proxy（代理）：一个类被AOP织入增强后，就产生一个结果代理类。
8. Aspect（切面）：是切入点和通知的结合。

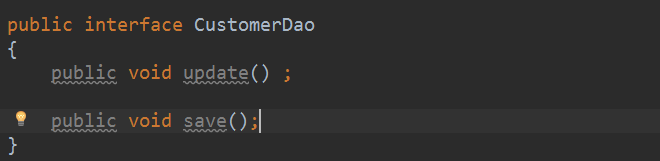


导入jar包：

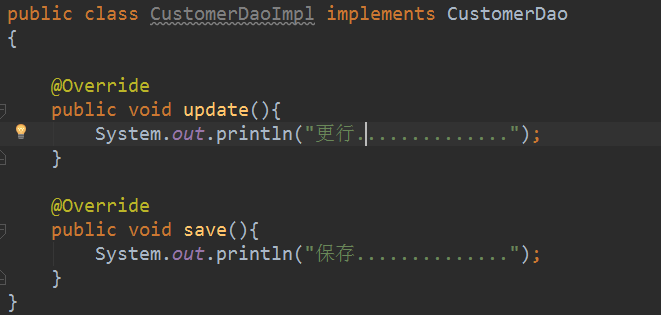


案例：

Dao层：



Dao层实现类：



配置文件：

|  |
| --- |
| <!--配置dao--> <bean id="customerDao" class="code.lsh.dao.impl.CustomerDaoImpl"></bean> <!--配置增强类--> <bean id="myaspectXml" class="code.lsh.aspect.MyaspectXml"></bean> <!--配置AOP--> <aop:config>  <!--配置切面类：切入点+通知-->  <aop:aspect ref="myaspectXml">  <!--前置通知-->  <aop:before method="log" pointcut="execution(public void code.lsh.dao.impl.CustomerDaoImpl.save())"></aop:before>  </aop:aspect> </aop:config> |

测试类：

切入点表达式：

execution()无法省略，public可以省略，返回值类型可以使用通配符（\*）代替，

包名可以使用通配符（\*）代替，也可以简写\*..\*代替。

类名也可以\*代替，但不建议，方法名不建议缩写。

方法参数可以使用\*代替，也可以使用..代替任意参数。

通知类型：

前置通知：在目标类方法执行之前执行



后置通知：在方法执行完毕后执行。



最终通知：在目标类的方法执行之后执行，如果程序出行异常，也会执行。



异常抛出通知：在抛出异常时执行。



环绕通知：方法执行前后执行。默认情况下，目标对象方法无法正常执行，需要手动让目标对象的方法执行。

解决方法：需要在增强的方法中添加ProceedingJoinPoint类型方法参数，在需要执行目标对象时，调用ProceedingJoinPoint的proceed()方法。



