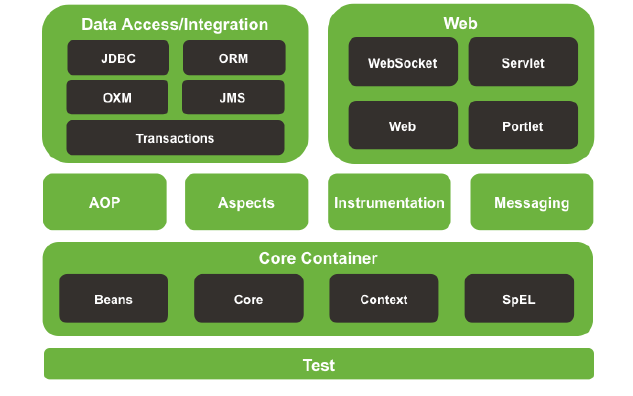
# Spring模块



## 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。

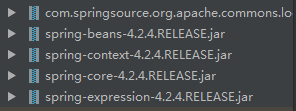
## AOP

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：

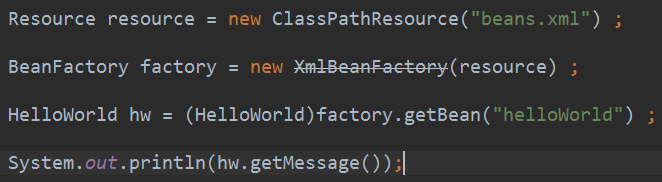
# IOC容器

需要的jar包：

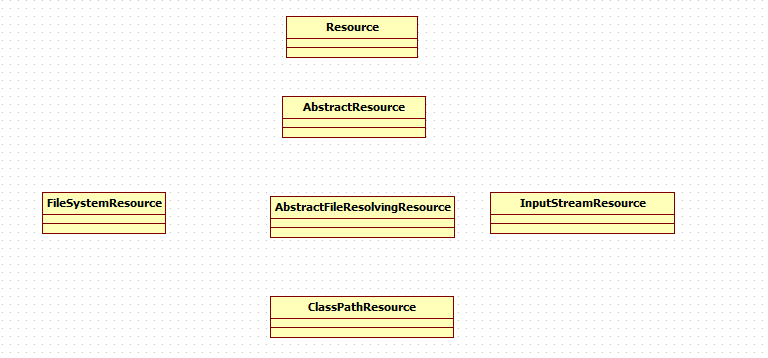


## 实例化容器

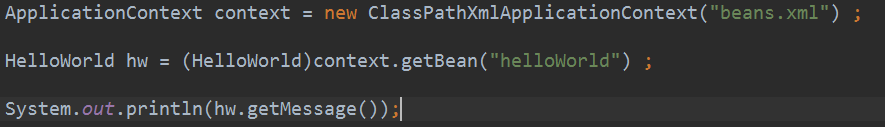
### 实例化BeanFactory



Resource实现类图：



### 实例化ApplicationContext



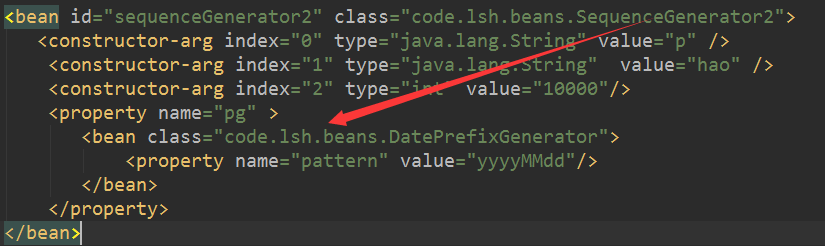
ApplicationContext的一般实现：

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。



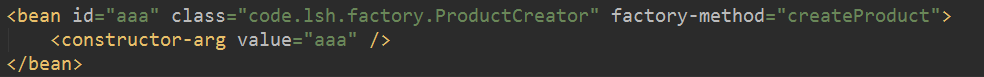
### 静态工厂方法创建Bean

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

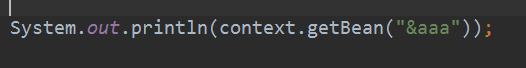
工厂Bean是一种特殊的Bean，工厂Bean与工厂方法相似，Spring在创建Bean时能够分辨出工厂Bean。工厂Bean基本要求是实现FactoryBean接口，Spring提供了一个抽象的摸板类—AbstracFactoryBean。

|  |
| --- |
| 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”)

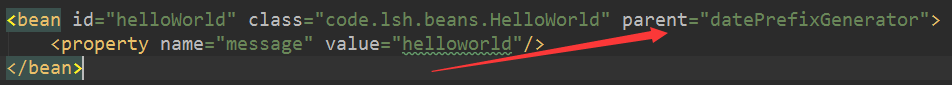
或者@Resource(name=”beanName”)

## 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

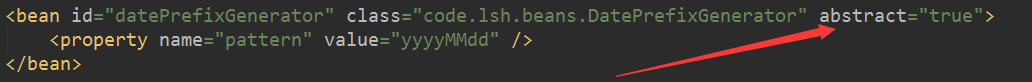
Bean的bean叫做子Bean。父Bean可以作为配置模板。

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。

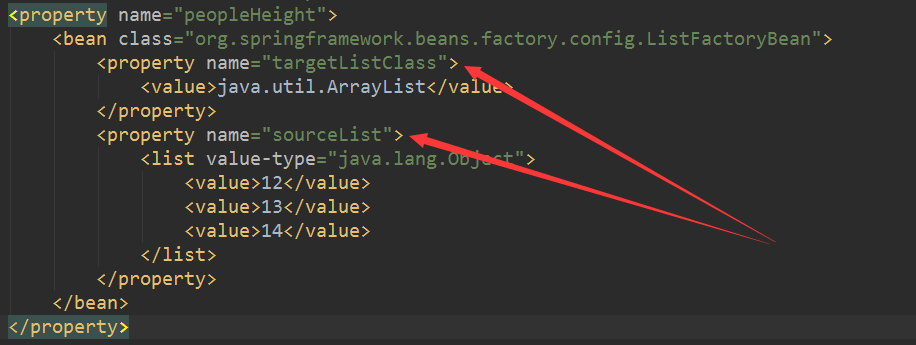


## 工厂Bean

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。

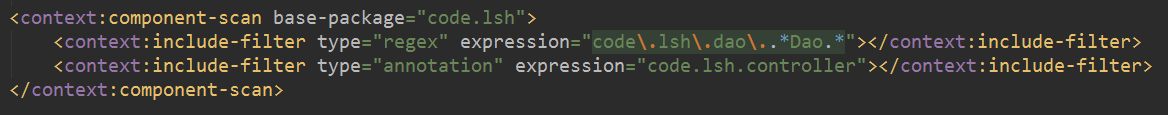
@Service标识了服务层的服务组件。

@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：

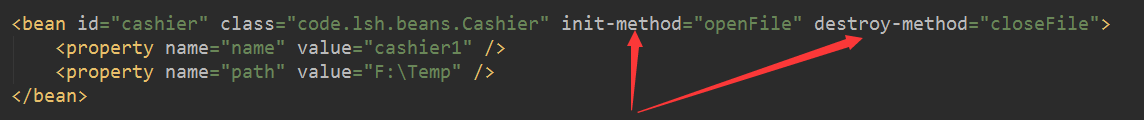
1. 通过构造方法或者工厂方法创建Bean的实例
2. 为Bean的属性设置值和对其他Bean的引用
3. 调用Bean的初始化回调方法
4. 使用Bean
5. 当容器关闭时，调用Bean的销毁回调方法。

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();  } |

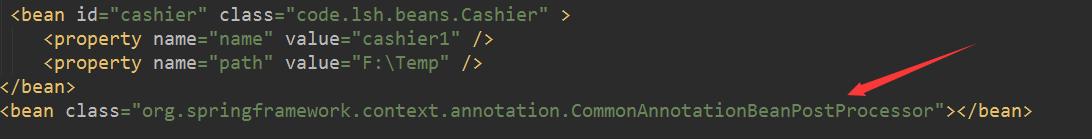
第二种：在配置文件中设置init-method和destory-method属性。



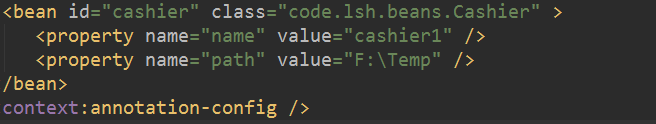
第三种：给初始化方法和销毁的回调方法添加@PostConstruct和@PreDestory注解。

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。

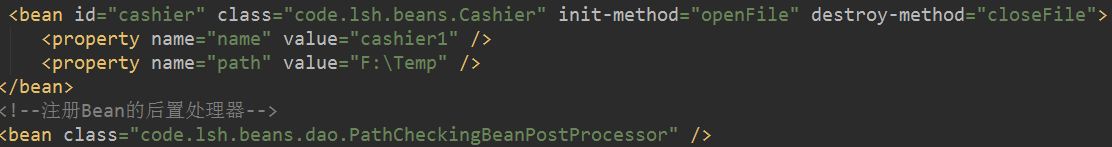
|  |
| --- |
| */\*\*  \* 定义接口，为了使Bean后置处理器能够辨别应该被检查的Bean  \*/* public interface StorageConfig {  public String getPath(); } |

|  |
| --- |
| 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：



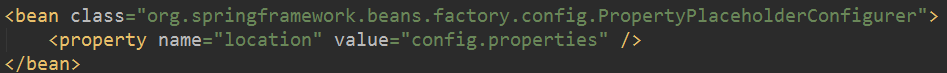
注意：如果Bean依赖于@PostConstruct和@PreDestory，还有用来调用初始化方法的CommonAnnotationBeanPostProcessor实例，则Bean的后置方法无法正常的工作。这是因为默认情况下CommonAnnotationBeanPostProcessor优先级高于Bean的后置处理器。如果要定义Bean后置处理器的处理顺序，可以实现Ordered或者PriorityOrdered接口，在getOrder()方法中返回数序值，这个方法的返回值越低则优先级越高。

## 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;  } } |

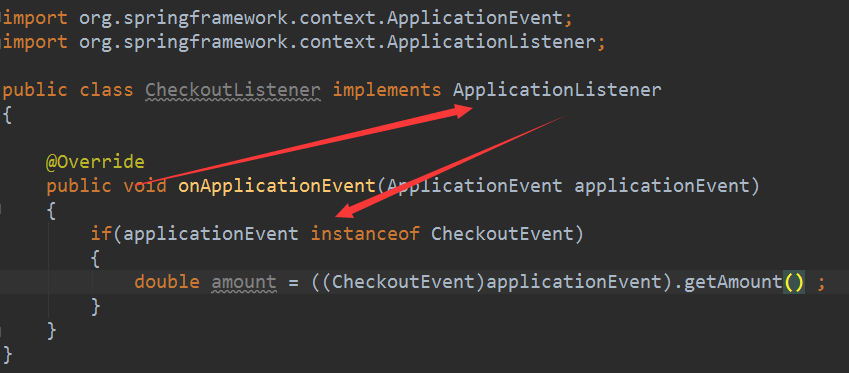
1. 发布事件

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：



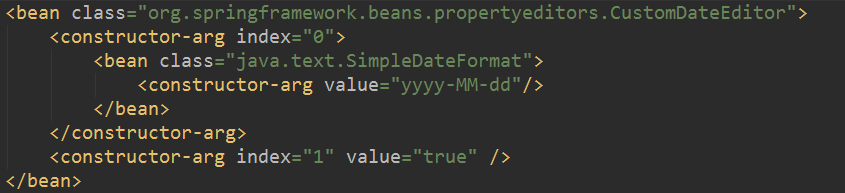
Application Context会辨别出实现了ApplicationListener接口的Bean，然后在有事件发布时通知它们。

## 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

## 加载属性文件

### 注解方式

Spring 提供了注解@PropertySource来加载属性文件，它的使用比较简单。

public @interface PropertySource {  
 String name() default "";  
  
 String[] value();  
  
 boolean ignoreResourceNotFound() default false;  
}

name：字符串，配置这次属性配置的名称

value：字符串数组，可以配置多个属性文件。

ignoreResourceNotFound：boolean值，默认为false，其含义为如果找不到对应的属性文件是否尽心忽略处理，由于默认为false，所以在默认的情况下找不到对应的配置文件就会抛出异常。

案例：在classpath路径下配置database-config.properties文件

jdbc.database.driver=com.mydql.jdbc.Driver  
jdbc.database.url=jdbc:mysql://localhost:3306/spring  
jdbc.database.username=root  
jdbc.database.password=123456

使用注解：

@Configuration  
@PropertySource(value={"classpath:database-config.properties"},ignoreResourceNotFound=true)  
public class ApplicaitonProperties  
{  
 public static void main(String[] args)  
 {  
 ApplicationContext context = new AnnotationConfigApplicationContext(ApplicaitonProperties.class) ;  
 String url = context.getEnvironment().getProperty("jdbc.database.url");  
 System.*out*.println(url);  
 }  
}

Spring推荐使用一个属性文件解析类进行处理，它就是PropertySourcesPlaceholderConfigurer，使用它就意味着允许Spring解析对应的属性文件，并通过占位符区引用对应的配置。

第一步：

import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Component;  
@Component  
public class DataBaseBean  
{  
 @Value("${jdbc.database.driver}")  
 private String driver = null;  
  
 @Value("${jdbc.database.url}")  
 private String url = null ;  
  
 @Value("${jdbc.database.username}")  
 private String username = null ;  
  
 @Value("${jdbc.database.password}")  
 private String password = null;  
  
 @Override  
 public String toString(){  
 return "DataBaseBean{"+"driver='"+driver+'\''+", url='"+url+'\''+", username='"+username+'\''+", password='"+password+'\''+'}';  
 }  
}

第二步：

import code.lsh.beans.DataBaseBean;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.annotation.\*;  
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;  
  
@Configuration  
@PropertySource(value={"classpath:database-config.properties"},ignoreResourceNotFound=true)  
@ComponentScan(basePackages={"code.lsh"})  
public class ApplicaitonProperties  
{  
 @Bean  
 public PropertySourcesPlaceholderConfigurer propertySourcesPlaceholderConfigurer()  
 {  
 return new PropertySourcesPlaceholderConfigurer();  
 }  
  
 public static void main(String[] args)  
 {  
 ApplicationContext context = new AnnotationConfigApplicationContext(ApplicaitonProperties.class) ;  
 DataBaseBean dbb = (DataBaseBean)context.getBean("dataBaseBean");  
 System.*out*.println(dbb.toString());  
 }  
}

### 配置文件方式

第一种方式：

<context:component-scan base-package="code.lsh"/>  
<context:property-placeholder ignore-resource-not-found="true" location="classpath:database-config.properties"/>

ignore-resource-not-found属性代表着是否允许文件不存在，如果设置为true，则是允许不存在，默认值为false，不允许文件不存在，如果我、不存在，Spring就会抛出异常。location是一个配置文件 的路径，它可以配置单个文件，也可以配置多个文件，多个文件之间要使用逗号隔开。

第二种方式：

<bean class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">  
 <property name="locations">  
 <array>  
 <value>classpath:database-config.properties</value>  
 </array>  
 </property>  
 <property name="ignoreResourceNotFound" value="true"/>  
</bean>

## 条件化转配Bean

在某些条件下不需要装配Bean，Spring提供了注解@Conditional去配置，通过它可以配置一个或者多个类，只是这些类都需要实现Condition接口。

public class DataSourceCondition implements Condition  
{  
 @Override  
 public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata annotatedTypeMetadata)  
 {  
 //获取上下文环境  
 Environment env = conditionContext.getEnvironment() ;  
 //判断是否存在关于数据源的基础配置  
 return env.containsProperty("jdbc.database.driver") && env.containsProperty("jdbc.database.url") &&  
 env.containsProperty("jdbc.database.username") && env.containsProperty("jdbc.database.password") ;  
 }  
}

# AOP面向切面编程

AOP并不是Spring框架特有的，Spring只是支持AOP编程的框架之一，每一额框架对AOP的支持各有特点，有些AOP能够对方法的参数进行拦截，有些AOP对方法进行拦截，而SpringAOP是一种基于方法拦截的AOP。

Spring中有4种方式去实现AOP的拦截功能：

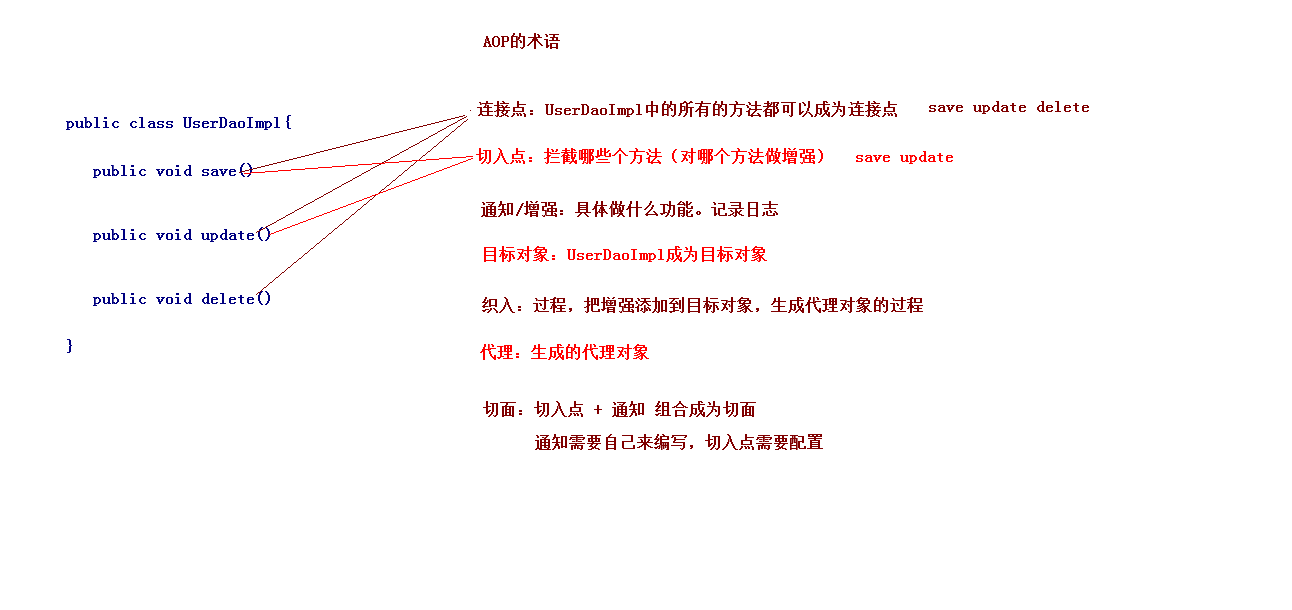
1. 使用ProxyFactoryBean和对应的接口实现AOP
2. 使用XML配置AOP
3. 使用@AspectJ注解驱动切面
4. 使用AspectJ注入切面。

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。切入点是一组连接点匹配的表达式。同一个切入点表达式可能会在多个通知中重复出现。
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。在AspectJ注解中，切面只是一个带有@Aspect注解的Java类。通知是个标注有某种通知注解的简单Java方法。



切入点表达式：

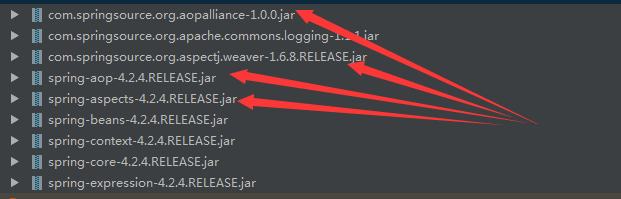
execution()无法省略，public可以省略，返回值类型可以使用通配符（\*）代替，

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。

导入jar包：



## ProxyFactoryBean配置AOP

### 通知模块

1. 前置通知，前置通知在方法执行之前执行，可以通过实现MethodBeforeAdvice接口创建它，在before方法里，能够获取目标方法的细节以及参数。

*/\*\*  
 \* 前置通知  
 \*/*public class LoggingBeforeAdvice implements MethodBeforeAdvice  
{  
 @Override  
 public void before(Method method, Object[] args, Object target) throws Throwable  
 {  
  
 }  
}

1. 后置通知，通过实现AfterReturningAdvice接口

*/\*\*  
 \*后置通知  
 \*/*public class LoggingAfterAdvice implements AfterReturningAdvice  
{  
 @Override  
 public void afterReturning(Object returnValue, Method method, Object[] args, Object target) throws Throwable  
 {  
 }  
}

1. 异常通知，对于异常通知必须实现ThrowAdvice接口，注意，该接口并没有声明任何方法，这样就能够在不同的方法里处理不同的异常了，不过，每个处理方法的名称必须是afterThrowing。异常的类型由方法的参数类型决定。

*/\*\*  
 \* 异常通知  
 \*/*public class LoggingThrowsAdvice implements ThrowsAdvice  
{  
 */\*\*  
 \*处理算术异常  
 \*/* public void afterThrowing(IllegalArgumentException e)  
 {  
  
 }  
  
 */\*\*  
 \*处理索引越界异常  
 \*/* public void afterThrowing(ArrayIndexOutOfBoundsException e){}  
}

1. 环绕通知，环绕通知必须实现MethodInterceptor接口，这个接口是AOP联盟了保持不同的AOP框架之间兼容性而定义的。

*/\*\*  
 \* 环绕通知  
 \*/*public class LoggingAroundAdvice implements MethodInterceptor  
{  
 @Override  
 public Object invoke(MethodInvocation methodInvocation) throws Throwable  
 {  
 Object result = methodInvocation.proceed();  
 return result;  
 }  
}

如：配置前置通知：打印方法名称和参数

第一步：定义接口

public interface ArithmeticCalculator  
{  
 public double add(double a ,double b) ;  
}

第二步：接口实现类

public class ArithmeticCalculatorImpl implements ArithmeticCalculator  
{  
 @Override  
 public double add(double a, double b){  
 return (a+b);  
 }  
}

第三步：创建前置通知

*/\*\*  
 \* 前置通知  
 \*/*public class LoggingBeforeAdvice implements MethodBeforeAdvice  
{  
 @Override  
 public void before(Method method, Object[] args, Object target) throws Throwable  
 {  
 System.*out*.println("The method "+method.getName()+"() begins with "+Arrays.*toString*(args));  
 }  
}

第四部：配置文件

|  |
| --- |
| <bean id="arithmeticCalculator" class="code.lsh.aspect.proxy.ArithmeticCalculatorImpl"></bean> <!--前置通知类--> <bean id="loggingBeforeAdvice" class="code.lsh.aspect.proxy.LoggingBeforeAdvice"></bean> <!--配置切入点--> <bean id="methodPointcut" class="org.springframework.aop.support.NameMatchMethodPointcut">  <property name="mappedName" value="add"></property> </bean> <!--关联切入点和通知--> <bean id="methodNameAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">  <!--切入点-->  <property name="pointcut" ref="methodPointcut"></property>  <!--通知-->  <property name="advice" ref="loggingBeforeAdvice"></property> </bean> <!--创建代理--> <bean id="proxyFactoryBean" class="org.springframework.aop.framework.ProxyFactoryBean">  <property name="target" ref="arithmeticCalculator"></property>  <property name="interceptorNames">  <list>  <value>methodNameAdvisor</value>  </list>  </property> </bean> |

第四步：测试

### 切入点模块

切入点（pointcut）通常以表达式的形式出现，能够匹配特定的程序执行点来通知，

Spring提供了一些列切入点类匹配程序的连接点。要定义切入点，可以简单地在Bean配置文件中声明这些类型的Bean。如果发现这些内置的切入点不能满足需要，可以扩展StaticMethodMatcherPointcut或DynamicMethodMatcherPointcut编写自己的切入点类。前者可以静态类和方法信息匹配连接点。而后者还可以通过动态的参数值匹配连接点。

1. 方法名切入点

如果只想增强一个单独的方法，可以使用NameMatchMethodPointcut通过方法名称静态地匹配它。

<bean id="methodPointcut" class="org.springframework.aop.support.NameMatchMethodPointcut">  
 <property name="mappedName" value="add"></property>  
</bean>

切入点必须与通知关联起来才能表明在哪里应用这些通知，可以使用DefaultPointcutAdvisor类简单地切入点和通知关联起来。

<bean id="" class="org.springframework.aop.support.DefaultPointcutAdvisor">  
 <property name="pointcut" ref="methodPointcut"></property>  
 <property name="advice" ref="loggingBeforeAdvice"></property>  
</bean>

如果想在方法名称切入点中里匹配多个方法，可以在mappedNames属性里设置这些方法。

Spring提供了一些方便的增强器类，可以将切入点声明合并到增强器里：

1. 正则表达式切入点

RegexpMethodPointcut里指定一个或多个正则表达式。

1. AspectJ表达式切入点

使用AspectJExpressionPointcutAdvisor

<bean id="" class="org.springframework.aop.aspectj.AspectJExpressionPointcut">  
 <property name="expression">  
 <value>execution(\* \*.\*To\*(..))</value>  
 </property>  
 <property name="advice">  
 <ref bean="loggingBeforeAdvice"></ref>  
 </property>  
</bean>

## XML配置AOP

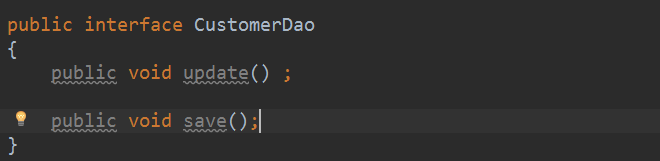
|  |  |
| --- | --- |
| AOP配置元素 | 用途 |
| aop:advisor |  |
| aop:aspect |  |
| aop:before |  |
| aop:after |  |
| aop:around |  |
| aop:after-returning |  |
| aop:after-throwing |  |
| aop:config |  |
| aop:declare-parents |  |
| aop:pointcut |  |

约束文件：

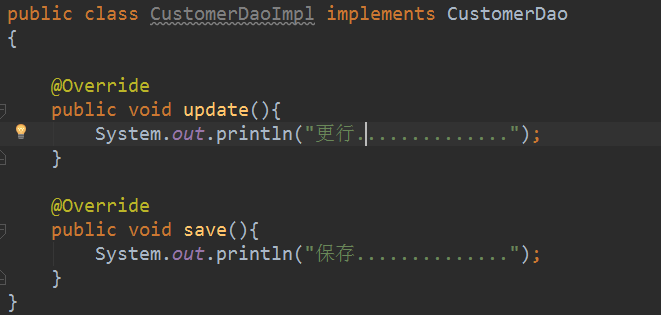
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xmlns:aop="http://www.springframework.org/schema/aop"  
http://www.springframework.org/schema/context/spring-context-4.2.xsd  
http://www.springframework.org/schema/aop/spring-aop-4.2.xsd">  
</beans>

案例：

Dao层：



Dao层实现类：



|  |
| --- |
| <!--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：

通知类型：

前置通知：在目标类方法执行之前执行，或者在动态代理反射原有对象方法或者执行环绕通知前执行的通知功能。

org.springframework.aop.MethodBeforeAdvice



后置通知：在方法执行完毕后执行，或者在动态代理反射原有对象方法或者执行环绕通知后执行的通知功能，无论是否抛出异常，它都会被执行。



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。



## 使用@AspectJ

|  |  |  |
| --- | --- | --- |
| 注解 | 通知 | 备注 |
| @Before | 在被代理对象的方法之前先调用 | 前置通知 |
| @Around | 将被代理对象的方法封装起来，并用环绕通知取代它 | 环绕通知，它将覆盖原有的方法，但是允许你通过反射调用原有方法。 |
| @After | 在被代理对象的方法后调用 | 后置通知 |
| @AfterRet0.urning | 在被代理对象的方法正常返回后调用 | 返回通知，要求代理对象的方法执行过程中没有发生异常 |
| @AfterThrowing | 在被代理对象的方法抛出异常后调用 | 异常通知，要求被代理对象的方法执行过程中产生异常。 |

第一步：选择连接点

Spring是方法级别的AOP框架，而我们主要以某个类的方法作为连接点。

public interface RoleService  
{  
 public void printRole(Role role) ;  
}

*/\*\*  
 \* 连接点  
 \*/*@Component  
public class RoleServiceImpl implements RoleService  
{  
  
 @Override  
 public void printRole(Role role)  
 {  
 System.*out*.println(role.toString());  
 }  
}

第二步：创建切面

Spring中使用注解@Aspect注解一个类，则Spring IOC容器就会认为这是一个切面了

@Aspect  
public class RoleAspect  
{  
 @Before("execution(code.lsh.aspect.aop.aspectj.RoleServiceImpl.printRole(..))")  
 public void befor()  
 {  
 System.*out*.println("------------befor-------------");  
 }  
  
 @After("execution(code.lsh.aspect.aop.aspectj.RoleServiceImpl.printRole(..))")  
 public void after()  
 {  
 System.*out*.println("------------after-------------");  
 }  
  
 @AfterReturning("execution(code.lsh.aspect.aop.aspectj.RoleServiceImpl.printRole(..))")  
 public void afterReturning()  
 {  
 System.*out*.println("------------afterReturning-------------");  
 }  
  
 @AfterThrowing("execution(code.lsh.aspect.aop.aspectj.RoleServiceImpl.printRole(..))")  
 public void afterThrowing()  
 {  
 System.*out*.println("------------afterThrowing-------------");  
 }  
}

为了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，然后访问连接点的细节。

@After("execution(public \* code.lsh.aspect.aop.aspectj.RoleServiceImpl.printRole(..))")  
public void after(JoinPoint joinPoint)  
{  
 //获取连接点名  
 String methodName = joinPoint.getSignature().getName() ;  
 //获取连接点参数列表  
 Object[] args = joinPoint.getArgs();  
 System.*out*.println("------------after-------------");  
 //连接点类型(Spring AOP只有method-execution)  
 System.*out*.println(joinPoint.getKind());  
 //方法签名  
 System.*out*.println(joinPoint.getSignature().getDeclaringTypeName());  
 //方法名  
 System.*out*.println(joinPoint.getSignature().getName());  
 //参数值  
 System.*out*.println(joinPoint.getArgs());  
 //目标对象  
 System.*out*.println(joinPoint.getTarget().getClass().getName());  
 //代理对象  
 System.*out*.println(joinPoint.getThis().getClass().getName());  
}

### 指定切面优先级

同一个连接点上应用了不止一个切面时，除非明确指定，否则它们的优先级时不确定的。切面的优先级可以通过实现Ordered接口或使用@Order注解指定。

实现Ordered接口：getOrder方法返回值越小，则优先级就越高。

@Override  
public int getOrder(){  
 return 0;  
}

第三步：定义切入点

1.方法签名模式

execution(code.lsh.aspect.aop.aspectj.RoleServiceImpl.printRole(..))

execution：代表执行方法的时候触发

\*：代表任意返回类型的方法

code.lsh.aspect.aop.aspectj.RoleServiceImpl：代表类的全限定名。

(..)：任意参数。

AspectJ的指示器：

|  |  |
| --- | --- |
| AspectJ指示器 | 描述 |
| arg() | 限制连接点匹配参数为指定类型的方法 |
| @args() | 限制连接点匹配参数为指定注解标注的执行方法 |
| execution | 用于匹配连接点的执行方法 |
| this() | 限制连接点匹配AOP代理的Bean，引用为指定类型的类 |
| target | 限制连接点匹配被代理对象为指定的类型 |
| @target | 限制连接点匹配特定的执行对象，这些对象符合指定的注解类型 |
| within() | 限制连接点匹配指定的包 |
| @within() | 限制连接点匹配指定的类型 |
| @annotation | 限制匹配带有指定注解的连接点 |

2.类型签名模式

with(code.lsh.aspect.aspectJ.\*)：在指定包内的所有方法执行连接点的匹配。

with(code.lsh.aspect.aspectJ..\*)：匹配包及其子包内部的连接点

with(code.lsh.aspect.aspectJ.RoleServiceImpl)：匹配指定类内部方法执行连接点

with(RoleServiceImpl)：如果目标类与这个切面在同一个包中，可以省略包名。

with(RoleService+)：添加一个+，可以与实现指定接口的所有类内部方法执行连接点匹配。

@AfterReturning("within(code.lsh.aspect.aop.aspectj.RoleServiceImpl)")  
public void afterReturning2()  
{  
 System.*out*.println("------------afterReturning-------------");  
}

3.Bean名称模式

@AfterReturning("bean(code.lsh.aspect.aop.aspectj.RoleServiceImpl)")  
public void afterReturning2()  
{  
 System.*out*.println("------------afterReturning-------------");  
}

4.合并切入点表达式

在AspectJ中，切入点表达式可以通过操作符&&（逻辑与），||（逻辑或）和!（逻辑非）结合起来。

5.声明切入点参数

利用某些特殊的切入点表达式，以声明的方式访问连接点信息，如：表达式target()和args()捕获目标对象和当前连接点的参数值，并将它们公开为切入点参数，这些参数通过同名参数传递到通知方法中。

第四步：测试AOP

@Configuration  
@EnableAspectJAutoProxy //启用AspectJ框架的自动代理  
@ComponentScan(basePackages={"code.lsh"})  
public class AopConfig  
{  
 @Bean  
 public RoleAspect getRoleAspect()  
 {  
 return new RoleAspect();  
 }  
  
 public static void main(String[] args)  
 {  
 ApplicationContext context = new AnnotationConfigApplicationContext(AopConfig.class) ;  
 RoleService roleService = (RoleService)context.getBean(RoleService.class) ;  
 Role role = new Role();  
 role.setId(10);role.setRoleName("hao");role.setNote("you are name?");  
 roleService.printRole(role);  
 System.*out*.println("-------------");  
 role = null ;  
 roleService.printRole(role);  
 }  
}

环绕通知是Spring AOP中最强大的通知，它可以同时实现前置通知和后置通知。它保留了调度被代理对象原有方法的功能。

修改切面：

@Around("print()")  
public void around(ProceedingJoinPoint jp)  
{  
 System.*out*.println("------------around before-------------");  
 try{  
 jp.proceed();  
 }catch(Throwable throwable){  
 throwable.printStackTrace();  
 }  
 System.*out*.println("------------around after-------------");  
}  
  
@Pointcut("execution(public \* code.lsh.aspect.aop.aspectj.RoleServiceImpl.printRole(..))")  
public void print(){}

## 引入

引入（introduction）时AOP中一种特殊的通知类型，它通过为接口提供实现类，允许对象动态地实现接口，就像对象已经在运行时扩展了实现类一样。此外，还能同时为对象引入多个实现类和接口，这与多继承的效果相同。

比如printRole方法要求，如果当角色为空时不再打印，那么就需要引入一个新的检测器对其尽心检测。

/\*  
\* 检验器  
\* \*/  
public interface RoleVerifier  
{  
 public boolean verify(Role role);  
}

*/\*\*  
 \* 检验器实现类  
 \*/*public class RoleVerifierImpl implements RoleVerifier  
{  
 @Override  
 public boolean verify(Role role){  
 return role!=null;  
 }  
}

修改切面（RoleAspect.java）：添加如下内

@DeclareParents(value="code.lsh.aspect.aop.aspectj.RoleServiceImpl",defaultImpl=RoleVerifierImpl.class)  
public RoleVerifier roleVerifier ;

@DeclareParents注解类型value属性表示哪些类是这个引入的目标，也就是在RoleServiceImpl中引入一个新的接口，引入的接口有被注解域的类型决定。defaultImpl属性中指定这个新接口使用的实现类。

@Configuration  
@EnableAspectJAutoProxy //启用AspectJ框架的自动代理  
@ComponentScan(basePackages={"code.lsh"})  
public class AopConfig  
{  
 @Bean  
 public RoleAspect getRoleAspect()  
 {  
 return new RoleAspect();  
 }  
  
 public static void main(String[] args)  
 {  
 ApplicationContext context = new AnnotationConfigApplicationContext(AopConfig.class) ;  
 RoleService roleService = (RoleService)context.getBean(RoleService.class) ;  
 RoleVerifier roleVerifier = (RoleVerifier) roleService ;  
 Role role = new Role();  
 role.setId(10);role.setRoleName("hao");role.setNote("you are name?");  
 if(roleVerifier.verify(role))  
 {  
 roleService.printRole(role,10);  
 }  
  
 }  
}

使用强制类型转换之后就可以把roleServie转化为RoleVerifier对象。

# 数据库事务管理

## Spring数据库事务管理器设置

Spring中支持事务的是TransactionTemplate模板，它是Spring提供的事物管理器的模板。

|  |
| --- |
| private PlatformTransactionManager transactionManager;  public <T> T execute(TransactionCallback<T> action) throws TransactionException {  //使用自定义的事物管理器  if (this.transactionManager instanceof CallbackPreferringPlatformTransactionManager) {  return ((CallbackPreferringPlatformTransactionManager)this.transactionManager).execute(this, action);  } else {//系统默认管理器  //获取事物状态  TransactionStatus status = this.transactionManager.getTransaction(this);  Object result;  try {  //回调接口方法  result = action.doInTransaction(status);  } catch (RuntimeException var5) {  //回滚异常方法  this.rollbackOnException(status, var5);  //抛出异常  throw var5;  } catch (Error var6) {  this.rollbackOnException(status, var6);  throw var6;  } catch (Exception var7) {  this.rollbackOnException(status, var7);  //抛出无法捕获的异常  throw new UndeclaredThrowableException(var7, "TransactionCallback threw undeclared checked exception");  }  //提交事物  this.transactionManager.commit(status);  return result;  } } |

源码中我们可以从看出事物的创建，提交和会馆是通过PlatformTransactionManage接口完成的。当事物产生异常时会回滚异常，在默认的实现中所有的异常都会回滚。

Spring多种事物管理器：

PlatformTransactionManager是所有Spring事物管理器的通用接口，Spring内置了几个实现，用于不同的事物管理API。

1. 如果应用程序只需要一个数据源，并且通过JDBC进行存取，

DataSourceTransactionManager能够满足我们的需求。

1. 如果是在JavaEE应用服务器上用JTA(Java Transaction API)进行事物管理，就

应该使用JtaTransactionManager从应用服务器查找事物。Spring提供支持分布式事物管理实现类。

（3）如果是用对象/关系映射框架存取数据库，就应该为这个框架选择相应的事物管理器。

## 用事物管理器API编程式管理事物

|  |
| --- |
| public class TransactionJdbcBookShop extends JdbcDaoSupport implements BookShop {  private PlatformTransactionManager transactionManager ;   public void setTransactionManager(PlatformTransactionManager transactionManager)  {  this.transactionManager = transactionManager ;  }  @Override  public void purchase(String isbn, String username)  {  TransactionDefinition def = new DefaultTransactionDefinition() ;  TransactionStatus statue = transactionManager.getTransaction(def) ;  try  {  /\*  \* 执行具体的SQL语句  \* \*/  //提交事物  transactionManager.commit(statue);  }catch(Exception e){  //回滚事物  transactionManager.rollback(statue);  e.printStackTrace();  }   } } |

配置文件：

|  |
| --- |
| <!--配置数据源--> <bean id="dataSource" class="org.springframework.jdbc.datasource.SimpleDriverDataSource">  <property name="username" value="root"></property>  <property name="password" value="lsh159"></property>  <property name="url" value="jdbc:mysql://localhost:3306/spring"></property>  <property name="driverClass" value="com.mysql.jdbc.Driver"></property> </bean> <!--配置数据源事物管理器--> <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource"></property> </bean> <bean id="bookShop" class="code.lsh.service.impl.TransactionJdbcBookShop">  <property name="transactionManager" ref="transactionManager"></property> </bean> |

## 用事物模板编程式管理事物

事物模板是在事务管理器上创建的，事物模板执行封装了事物代码的事物回调对象，我们可以独立类或者内部类的形式来实现回调接口，如果实现为内部类，就必须将方法参数声明为final，以供内部类访问。

TransactionTemplate template = new TransactionTemplate(transactionManager) ;  
template.execute(new TransactionCallbackWithoutResult(){  
 @Override  
 protected void doInTransactionWithoutResult(TransactionStatus transactionStatus)  
 {  
 */\*\*  
 \* 具体的SQL语句  
 \*/* }  
});

事物模板可以接受TransactionCallback或者TransactionCallbackWithoutResult接口的事物回调对象，在回调对象执行期间，如果它抛出未受检的异常（如RuntimeException和DataAccessException），或者在TransactionStatus参数上显示地调用setRollbackOnly()方法，事物就会回滚，否则，回调对象完成之后就会提交事物。

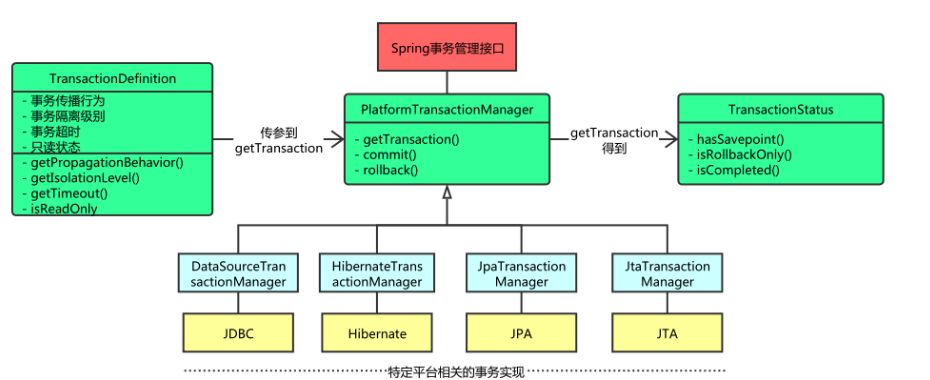
## 声明式管理事物

声明式事物是一种约定型事物，在大部分情况下，当时用数据库事物时，大部分的场景是在代码发生了异常时。需要回滚事物。而不发生异常时就提交事物，从而保证数据库数据的一致性。Spring给了一个约定，当我们的业务方法不发生异常时，Spring就会让事物管理器提交事物。而发生异常时，则让事物管理器回滚事物。

声明式事物管理将事物管理代码从业代码中分离出来，以声明的方式来实现事物管理。事物管理作为一种横切关注点，可以通过AOP方法模块化。Spring通过AOP框架支持声明式事物管理。

Spring支持的事物属性包括：传播行为，隔离级别，回滚规则，事物超时以及只读。

声明式事物允许自定义事物接口-----TransactionDefinition，它可以由XML或者@Transactional进行配置。



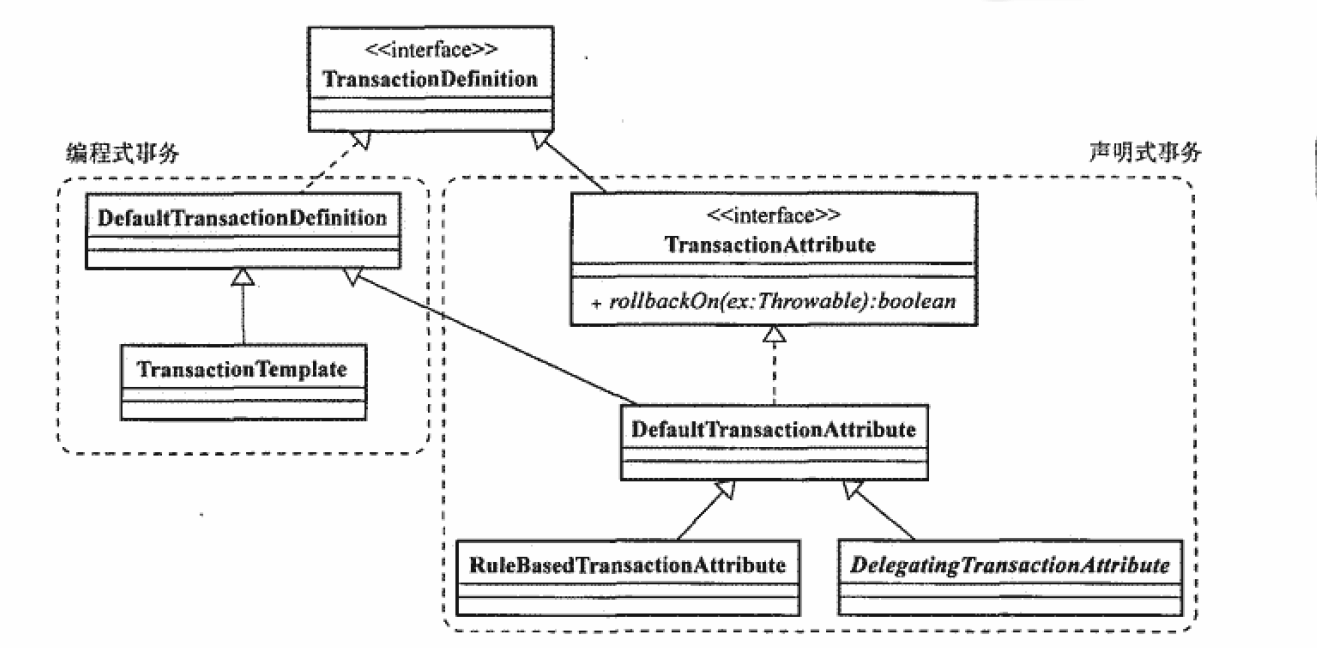
### @Transactional配置项

|  |  |  |
| --- | --- | --- |
| 配置项 | 含义 | 备注 |
| value | 定义事物管理器 | 同时Spring IOC容器中的Bean id，这个Bean需要实现PlatformTransactionManager接口 |
| transactionManager | 同上 | 同上 |
| isolation | 隔离级别 | 这是一个数据库在多个事物同时存在时的概念，默认值取数据库默认隔离级别 |
| propagation | 传播行为 | 传播行为是方法之间调用的问题。默认值为Propagation.REQUIRED |
| timeout | 超时时间 | 单位为秒，当超时时，会引用异常，默认会导致事物会馆 |
| readOnly | 是否开启只读事物 | 默认值为false |
| rollbakFor | 回滚事物的异常类定义 | 也就是当只有当方法产生所定义异常时，才回滚事物，否则就提交事物 |
| rollbackForClassName | 回滚事物的异常类名定义 | 同rollbackFor，只是使用类名称定义 |
| noRollbackFor | 当产生哪些异常不回滚事物 | 当产生所定义异常时，Spring将继续提交事物。 |
| noRollbackForClassName | 同noRollbackFor | 同noRollbackFor，只是使用类的名称定义 |

注意：注解@Transactional的底层实现时Spring AOP技术，而Spring AOP技术使用的动态代理，这意味着对于静态方法和非public方法，注解@Transactional是失效的

自调用失效：自调用就是一个类的方法去调用自身另外一个方法的过程。

### TransactionDefinition



TransatcionDefinition类继承层次图

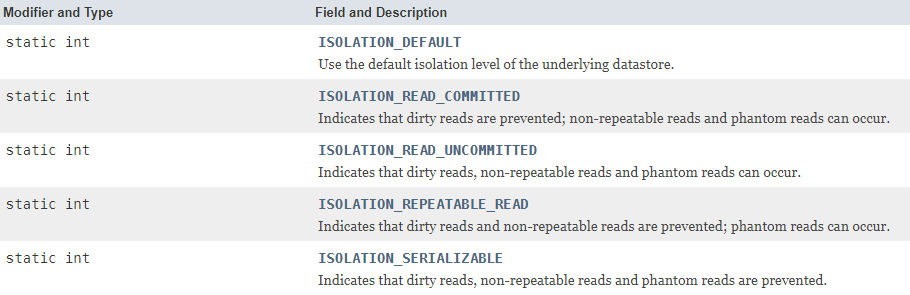
‘TransactionTemplate是Spring提供的进行编程式模板方法类，它直接继承与DefaultTransactionDefinition，我们在使用该模板类时，可以直接使用该模板类提供的事物控制属性。’

TransactionAttribute：继承TransactionDefinition，在DefaultTransactionDefinition的基础上增加了rollbackOn()方法，DefaultTransactionDefinition的实现，指定了当异常类型为uncheck exception的情况下就回滚事物。

RuleBasedTransactionAttribute：允许我们同时指定过个回滚规则，这些规则包括RollbackRuleAttribute或NoRollbackRuleAttribute的List形式提供，RulrBasedTransactionAttribute的rollbackOn将使用传入的异常类型与这些回滚规则进行匹配，然后再决定是否回滚事物。

@Override  
public boolean rollbackOn(Throwable ex) {  
 if (*logger*.isTraceEnabled()) {  
 *logger*.trace("Applying rules to determine whether transaction should rollback on " + ex);  
 }  
  
 RollbackRuleAttribute winner = null;  
 int deepest = Integer.*MAX\_VALUE*;  
  
 if (this.rollbackRules != null) {  
 for (RollbackRuleAttribute rule : this.rollbackRules) {  
 int depth = rule.getDepth(ex);  
 if (depth >= 0 && depth < deepest) {  
 deepest = depth;  
 winner = rule;  
 }  
 }  
 }  
 if (*logger*.isTraceEnabled()) {  
 *logger*.trace("Winning rollback rule is: " + winner);  
 }  
  
 // User superclass behavior (rollback on unchecked) if no rule matches.  
 if (winner == null) {  
 *logger*.trace("No relevant rollback rule found: applying default rules");  
 return super.rollbackOn(ex);  
 }  
  
 return !(winner instanceof NoRollbackRuleAttribute);  
}

#### 隔离级别



ISOLATION\_DEFAULT：使用后端数据库默认的隔离级别

ISOLATION\_READ\_UNCOMMITTED：最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读

ISOLATION\_READ\_COMMITTED：允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生

ISOLATION\_REPEATABLE\_READ：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生

ISOLATION\_SERIALIZABLE：最高的隔离级别，完全服从ACID的隔离级别，确保阻止脏读、不可重复读以及幻读，也是最慢的事务隔离级别，因为它通常是通过完全锁定事务相关的数

在企业会选择读/写提交的方式设置事物，这样有助于提高并发，有压制了脏读，

但是对对于数据一致性问题并没有解决。

注解@Transactional隔离级别的默认值为Isolation.DEFAULT，其含义是默认的，

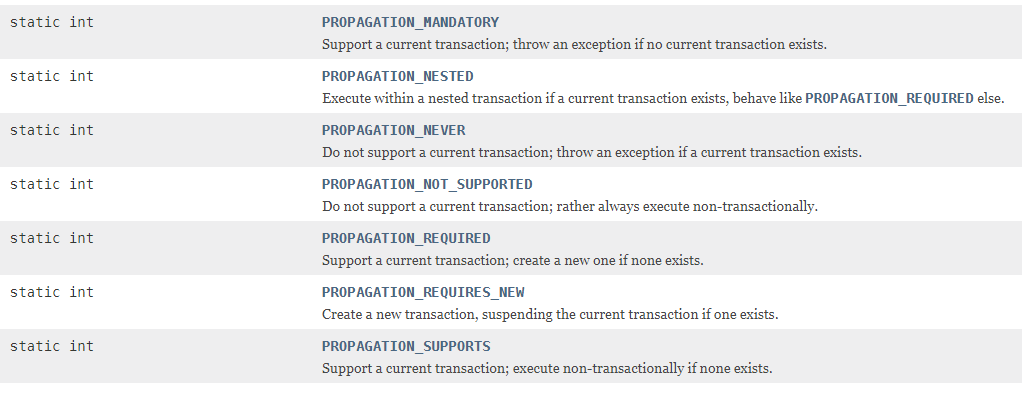
随数据库默认值的变化而变化，因为对于不同的数据库而言，隔离级别的支持是不

样的。

#### 传播行为

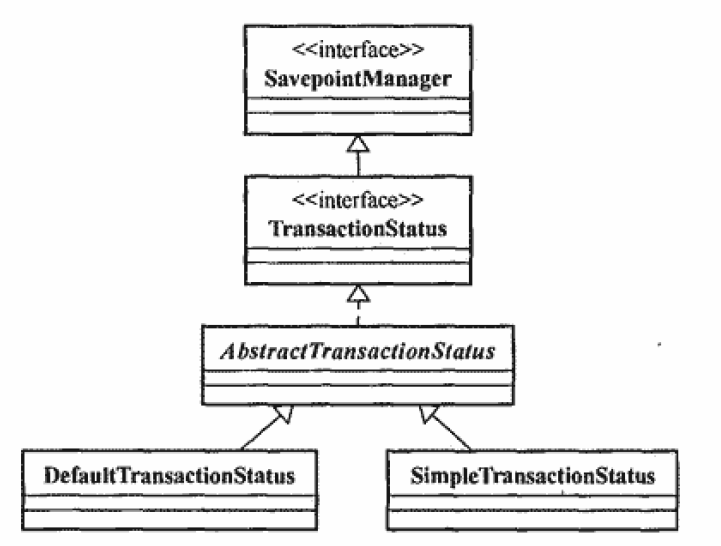
传播行为是指方法之间的调用事物策略的问题。在大部分的情况下，我们希望事物能够同时成功或者同时失败。

Spring中传播行为的类型为，是通过一个枚举类型定义，这个枚举类是Propagation。



|  |  |  |
| --- | --- | --- |
| 传播行为 | 含义 | 备注 |
| REQUIRED | 当方法调用时，如果不存在当前事物，那么就创建事物，如果之前的方法已经存在事物了，那么就沿用之前的事物 | 这是Spring默认的传播行为 |
| SUPPORTS | 当方法调用时，如果不存在当前事物，那么不启用事物，如果存在当前事物，那么就沿用当前事物 |  |
| MANDATORY | 方法必须在事物内运行 | 如果不存在当前事物，那么抛出异常 |
| REQUIRES\_NEW | 无论是否存在当前事物，方法都会在新的事物中运行 | 也就是事物管理器会打开新的事物运行该方法 |
| NOT\_SUPPORTED | 不支持事物，如果不存在当前事物也不会创建事物，如果存在当前事物，则挂其它，直到该方法结束后才恢复当前事物 | 适用于那些不需要事物的SQL |
| NEVER | 不支持事物，只有在没有事物的环境中才能运行它 | 如果方法存在当前事物，则抛出异常 |
| NESTED | 嵌套事物，也就是调用方法如果抛出异常只回滚自己内部执行的SQL，而不回滚主方法的SQL | 它是实现存在两种情况，如果当前数据库支持保存点（savepoint），那么它就会在当前事物上使用保存点技术；如果发生异常则将方法内执行的SQL回滚到保存点上，而不是全部回滚，否则就等同于REQUIRES\_NEW创建新的事物运行方法代码 |

### TransactionStatus

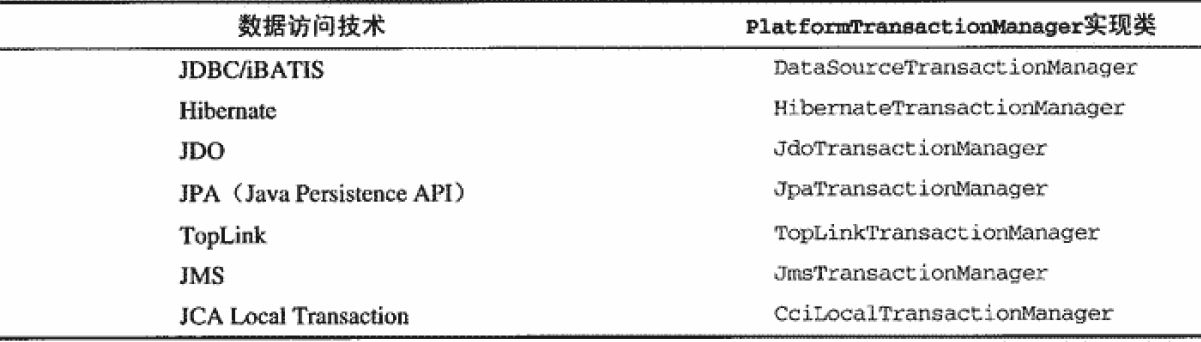


TransactionStatus接口定义了整个事物处理过程的事物状态

# PlatformTransactionManager

PlatformTransactionManager是实现类可划分为面向局部事物和面向全局事物两个分支。

**面向局部事物实现类：**Spring为各个数据访问技术提供了现成的PlatformTransactionManager实现支持，如图：



**面向全局事物实现类：**JtaTransactionManager

**需要了解的概念：**

**transaction object：**transaction object承载了当前事物的必要信息，PlatformTransactionManager实现类可以根据transaction object所提供的信息来决定如何处理当前事物。

**TransactionSynchronization：**可以注册到事物处理过程的回调接口，它就像事物处理的监听器，当事物处理的某些规定发生时，会调用TransactionSynchronization上的一些方法来执行相应的回调逻辑。如在事物完成后清理相应的系统资源等操作。

**TransactionSynchronizationManager：**管理TransactionSynchronization，当前事物状态以及具体的事物资源。

### AbstractPlatformTransactionManager

作为DataSourceTransactionManager父类，以模板方法的形式封装了固定的事物处理逻辑，而只将于事物资源相关的操作以protected或者abstract方法的形式留给DataSourceTransactionManager来实现。作为模板方法的父类，AbstractPlaformTransactionManager替各个子类实现了一下固定的事物内部处理逻辑：

1. 判定是否存在当前事物，然后根据判断结果执行不同的处理逻辑
2. 结合是否在当前事物的情况，根据TransactionDefinition中指定的传播行为的不同语义执行后继逻辑。
3. 根据情况挂起或者恢复事物。
4. 提交事物之前检查readOnly字段是否被设置，如果是，以事物回滚替代事物的提价。
5. 在事物回滚的情况下，清理并恢复事物状态。
6. 如果事物的Synchronization处于active状态，在事物处理的规定时间触发注册的Synchronization回调接口。

具体方法实现：

|  |
| --- |
| @Override public final TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException {  Object transaction = doGetTransaction();  // Cache debug flag to avoid repeated checks.  boolean debugEnabled = logger.isDebugEnabled();  if (definition == null) {  // Use defaults if no transaction definition given.  definition = new DefaultTransactionDefinition();  }  if (isExistingTransaction(transaction)) {  // Existing transaction found -> check propagation behavior to find out how to behave.  return handleExistingTransaction(definition, transaction, debugEnabled);  }  // Check definition settings for new transaction.  if (definition.getTimeout() < TransactionDefinition.TIMEOUT\_DEFAULT) {  throw new InvalidTimeoutException("Invalid transaction timeout", definition.getTimeout());  }  // No existing transaction found -> check propagation behavior to find out how to proceed.  if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_MANDATORY) {  throw new IllegalTransactionStateException(  "No existing transaction found for transaction marked with propagation 'mandatory'");  }  else if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_REQUIRED ||  definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_REQUIRES\_NEW ||  definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_NESTED) {  SuspendedResourcesHolder suspendedResources = suspend(null);  if (debugEnabled) {  logger.debug("Creating new transaction with name [" + definition.getName() + "]: " + definition);  }  try {  boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION\_NEVER);  DefaultTransactionStatus status = newTransactionStatus(  definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);  doBegin(transaction, definition);  prepareSynchronization(status, definition);  return status;  }  catch (RuntimeException ex) {  resume(null, suspendedResources);  throw ex;  }  catch (Error err) {  resume(null, suspendedResources);  throw err;  }  }  else {  // Create "empty" transaction: no actual transaction, but potentially synchronization.  if (definition.getIsolationLevel() != TransactionDefinition.ISOLATION\_DEFAULT && logger.isWarnEnabled()) {  logger.warn("Custom isolation level specified but no actual transaction initiated; " +  "isolation level will effectively be ignored: " + definition);  }  boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION\_ALWAYS);  return prepareTransactionStatus(definition, null, true, newSynchronization, debugEnabled, null);  } } |

解析：

（1）Object transaction = doGetTransaction();//获取transaction object，以判断是否存在当前事物

* 获取transaction objct类型因具体实现类不同而不同，DataSourceTransactionManager实现：

|  |
| --- |
| @Override protected Object doGetTransaction() {  DataSourceTransactionObject txObject = new DataSourceTransactionObject();  txObject.setSavepointAllowed(isNestedTransactionAllowed());  ConnectionHolder conHolder =  (ConnectionHolder) TransactionSynchronizationManager.*getResource*(this.dataSource);  txObject.setConnectionHolder(conHolder, false);  return txObject; } |

AbstractPlatformTransactionManager不需要知道具体实现类返回的transaction obejct具体类型是什么，因为最终对transaction object的依赖都将通过方法参数进行传递，只要具体的实现类在取得transaction object参数后知道如何转型就行，所以，该方法的返回值为Object类型。

* doGetTransaction()是getTransaction(TransactionDefinition)模板方法公开给子类来实现的abstract类型方法。DataSourceTransactionManager在实现该方法时，会从TransactionSynchronizationManager获取绑定的资源，然后添加到DataSourceTransactionObject之后返回。

（2）获取Log类的debug信息，避免之后代码重复

boolean debugEnabled = logger.isDebugEnabled();  
（3）检车TransactionDefinition参数合法性

if (definition == null) {  
 definition = new DefaultTransactionDefinition();  
 }  
如果definition参数为null，则创建一个DefaultTransactionDefinition实例以提供默认的事物定义数据。

（4）根据定义的transaction object判断是否存在当前事物，根据判断结果采取不同的处理方式：

if (isExistingTransaction(transaction)) {  
 // Existing transaction found -> check propagation behavior to find out how to behave.  
 return handleExistingTransaction(definition, transaction, debugEnabled);  
 }  
默认的情况下，isExistingTransaction返回false,该方法需要具体的子类根据情况进行重写，对于DataSourceTransactionManager，它会根据传入的transaction所记载的信息进行判断：

@Override  
protected boolean isExistingTransaction(Object transaction) {  
 DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;  
 return (txObject.getConnectionHolder() != null && txObject.getConnectionHolder().isTransactionActive());  
}

* handleExistingTransaction(TransactionDefinition definition ,Object transaction ,boolean debugEnable)

|  |
| --- |
| private TransactionStatus handleExistingTransaction(  TransactionDefinition definition, Object transaction, boolean debugEnabled)  throws TransactionException {   if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_NEVER) {  throw new IllegalTransactionStateException(  "Existing transaction found for transaction marked with propagation 'never'");  }   if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED) {  if (debugEnabled) {  logger.debug("Suspending current transaction");  }  //挂起当前事物  Object suspendedResources = suspend(transaction);  //  boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION\_ALWAYS);  return prepareTransactionStatus(  definition, null, false, newSynchronization, debugEnabled, suspendedResources);  }   if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_REQUIRES\_NEW) {  if (debugEnabled) {  logger.debug("Suspending current transaction, creating new transaction with name [" +  definition.getName() + "]");  }  //挂起当前事物  SuspendedResourcesHolder suspendedResources = suspend(transaction);  try {  boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION\_NEVER);  DefaultTransactionStatus status = newTransactionStatus(  definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);  //开启新的事物  doBegin(transaction, definition);  prepareSynchronization(status, definition);  return status;  }  catch (RuntimeException beginEx) {  resumeAfterBeginException(transaction, suspendedResources, beginEx);  throw beginEx;  }  catch (Error beginErr) {  resumeAfterBeginException(transaction, suspendedResources, beginErr);  throw beginErr;  }  }   if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_NESTED) {  //检测nestedTransactionAllowed属性  if (!isNestedTransactionAllowed()) {  throw new NestedTransactionNotSupportedException(  "Transaction manager does not allow nested transactions by default - " +  "specify 'nestedTransactionAllowed' property with value 'true'");  }  if (debugEnabled) {  logger.debug("Creating nested transaction with name [" + definition.getName() + "]");  }  //默认返回true，即默认使用Savepoint创建嵌套事物  if (useSavepointForNestedTransaction()) {  // Create savepoint within existing Spring-managed transaction,  // through the SavepointManager API implemented by TransactionStatus.  // Usually uses JDBC 3.0 savepoints. Never activates Spring synchronization.  DefaultTransactionStatus status =  prepareTransactionStatus(definition, transaction, false, false, debugEnabled, null);  status.createAndHoldSavepoint();  return status;  }  else {  // Nested transaction through nested begin and commit/rollback calls.  // Usually only for JTA: Spring synchronization might get activated here  // in case of a pre-existing JTA transaction.  boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION\_NEVER);  DefaultTransactionStatus status = newTransactionStatus(  definition, transaction, true, newSynchronization, debugEnabled, null);  doBegin(transaction, definition);  prepareSynchronization(status, definition);  return status;  }  }   // Assumably PROPAGATION\_SUPPORTS or PROPAGATION\_REQUIRED.  if (debugEnabled) {  logger.debug("Participating in existing transaction");  }  /\*\*  \*如果需要检查事物状态匹配情况，则对当前存在事物与传入的defintion中定义的隔离级别与ReadOnly属性  \*进行检查，如果数据不吻合，则抛出异常  \*/  if (isValidateExistingTransaction()) {  //验证隔离级别  if (definition.getIsolationLevel() != TransactionDefinition.ISOLATION\_DEFAULT) {  Integer currentIsolationLevel = TransactionSynchronizationManager.getCurrentTransactionIsolationLevel();  if (currentIsolationLevel == null || currentIsolationLevel != definition.getIsolationLevel()) {  Constants isoConstants = DefaultTransactionDefinition.constants;  throw new IllegalTransactionStateException("Participating transaction with definition [" +  definition + "] specifies isolation level which is incompatible with existing transaction: " +  (currentIsolationLevel != null ?  isoConstants.toCode(currentIsolationLevel, DefaultTransactionDefinition.PREFIX\_ISOLATION) :  "(unknown)"));  }  }  //验证read-onlu  if (!definition.isReadOnly()) {  if (TransactionSynchronizationManager.isCurrentTransactionReadOnly()) {  throw new IllegalTransactionStateException("Participating transaction with definition [" +  definition + "] is not marked as read-only but existing transaction is");  }  }  }  boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION\_NEVER);  return prepareTransactionStatus(definition, transaction, false, newSynchronization, debugEnabled, null); } |

doBegin(Object transaction，TransactionDefinition definition)方法我抽象方法，需要具体的子类来实现，在DataSourceTransactionManager中：

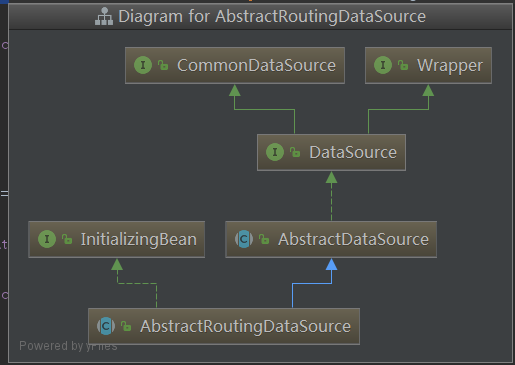
|  |
| --- |
| protected void doBegin(Object transaction, TransactionDefinition definition) {  DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;  Connection con = null;   try {  if (txObject.getConnectionHolder() == null ||  txObject.getConnectionHolder().isSynchronizedWithTransaction()) {  Connection newCon = this.dataSource.getConnection();  if (logger.isDebugEnabled()) {  logger.debug("Acquired Connection [" + newCon + "] for JDBC transaction");  }  txObject.setConnectionHolder(new ConnectionHolder(newCon), true);  }   txObject.getConnectionHolder().setSynchronizedWithTransaction(true);  con = txObject.getConnectionHolder().getConnection();   Integer previousIsolationLevel = DataSourceUtils.prepareConnectionForTransaction(con, definition);  txObject.setPreviousIsolationLevel(previousIsolationLevel);   // Switch to manual commit if necessary. This is very expensive in some JDBC drivers,  // so we don't want to do it unnecessarily (for example if we've explicitly  // configured the connection pool to set it already).  if (con.getAutoCommit()) {  txObject.setMustRestoreAutoCommit(true);  if (logger.isDebugEnabled()) {  logger.debug("Switching JDBC Connection [" + con + "] to manual commit");  }  con.setAutoCommit(false);  }  txObject.getConnectionHolder().setTransactionActive(true);   int timeout = determineTimeout(definition);  if (timeout != TransactionDefinition.TIMEOUT\_DEFAULT) {  txObject.getConnectionHolder().setTimeoutInSeconds(timeout);  }   // Bind the session holder to the thread.  if (txObject.isNewConnectionHolder()) {  TransactionSynchronizationManager.bindResource(getDataSource(), txObject.getConnectionHolder());  }  }   catch (Throwable ex) {  if (txObject.isNewConnectionHolder()) {  DataSourceUtils.releaseConnection(con, this.dataSource);  txObject.setConnectionHolder(null, false);  }  throw new CannotCreateTransactionException("Could not open JDBC Connection for transaction", ex);  } } |

（5）如果isExistingTransaction(transaction)方法返回false，即不存在当前事物情况下，当definition中定义的传播行为为PROPAGATION\_MANDATORY的时候，抛出异常，因为不存在当前事物。当definition中定义的传播行为是PROPAGATION\_REQUIRED，PROPAGATION\_REQUIRES\_NEW或者PROPAGATION\_NESTED的时候，开启新的事务。

# Spring JDBC

## 整合多数据源(主从分离)

Spring提供了对DataSource的实现：



AbstractRoutingDataSource是DataSource的抽象实现，在我们调用getConnection()方法时，会路由到一个目标数据源。

我们只需要继承该类，并重写determineCurrentLookupKey()方法即可

public class DynamicDataSource extends AbstractRoutingDataSource  
{  
 @Override  
 protected Object determineCurrentLookupKey()  
 {  
 return DynamicDataSourceHolder.*getDbType*();  
 }  
}

定义DynamicDataSourceHolder.java

public class DynamicDataSourceHolder  
{  
 private static Logger *log* = LoggerFactory.*getLogger*(DynamicDataSourceHolder.class) ;  
   
 private static ThreadLocal<String> *contextHolder* = new ThreadLocal<>();  
  
 */\*\*  
 \* 主数据源  
 \*/* public static String *DB\_MASTER* = "master" ;  
  
 */\*\*  
 \* 从数据源  
 \*/* public static String *DB\_SLAVE* = "slave" ;  
   
 public static String getDbType()  
 {  
 String db = *contextHolder*.get();  
 if(db == null)  
 {  
 db = *DB\_MASTER* ;  
 }  
 return db ;  
 }  
  
 public static void setDbType(String key)  
 {  
 *log*.debug("设置的数据源为："+key);  
   
 *contextHolder*.set(key);  
 }  
  
 public static void clearDbType()  
 {  
 *contextHolder*.remove();  
 }  
}

修改配置文件spring-dao.xml：

<!--配置数据库连接池 -->  
 <bean id="abstractDataSource" abstract="true" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">  
 <property name="driverClass" value="com.mysql.jdbc.Driver" />  
 <!--c3p0私有属性 -->  
 <property name="maxPoolSize" value="30"/>  
 <property name="minPoolSize" value="10"/>  
 <property name="autoCommitOnClose" value="false"/>  
 <property name="checkoutTimeout" value="10000"/>  
 <!--当获取连接失败后重试的次数 -->  
 <property name="acquireRetryAttempts" value="2"/>  
 </bean>  
<!--主数据源-->  
<bean id="master" parent="abstractDataSource">  
 <property name="jdbcUrl" value="jdbc:mysql://192.168.253.120:3306/o2o"/>  
 <property name="user" value="root"/>  
 <property name="password" value="lsh159"/>  
</bean>  
<!--从数据源-->  
<bean id="slave" parent=" abstractDataSource ">  
 <property name="jdbcUrl" value="jdbc:mysql://192.168.253.110:3306/o2o"/>  
 <property name="user" value="o2o"/>  
 <property name="password" value="lsh159"/>  
</bean>  
<!--配置动态数据源-->  
<bean id="dynamicDataSource" class="code.lsh.dao.split.DynamicDataSource">  
 <property name="targetDataSources">  
 <map>  
 <entry key="master" value-ref="master"></entry>  
 <entry key="slave" value-ref="slave"></entry>  
 </map>  
 </property>  
</bean>  
<!--配置懒加载：只有在SQL语句执行时，才能确定使用那一个数据源 -->  
<bean id="dataSource" class="org.springframework.jdbc.datasource.LazyConnectionDataSourceProxy">  
 <property name="targetDataSource" ref="dynamicDataSource" />  
</bean>  
  
 <!--配置SqlSessionFactory对象 -->  
 <bean id="sqlSessionFactoryName" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <property name="dataSource" ref="dataSource" /><!--注入数据库连接池 -->  
 <property name="configLocation" value="classpath:mybatis-config.xml"/><!--配置mybatis全局配置文件：mybatis.xml -->  
 <property name="typeAliasesPackage" value="code.lsh.entity"/><!--扫描entiry包 使用别名 -->  
 <property name="mapperLocations" value="classpath:mapper/\*.xml"/>  
 </bean>  
   
 <!--配置扫描Dao接口包，动态实现dao接口，注入到spring容器中 -->  
 <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  
 <property name="sqlSessionFactoryBeanName" value="sqlSessionFactoryName"/>  
 <property name="basePackage" value="code.lsh.dao" />  
 </bean>

配置MyBatis拦截器：

