Spring

尚硅谷java

V1.0

# Spring入门

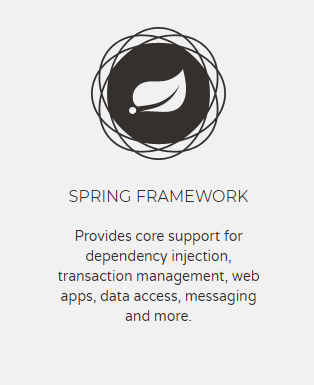
## Spring Framework

**框架**（Framework）：提供某一领域完整的解决方案，使开发更具有工程性、简便性、稳定性，更有效率。

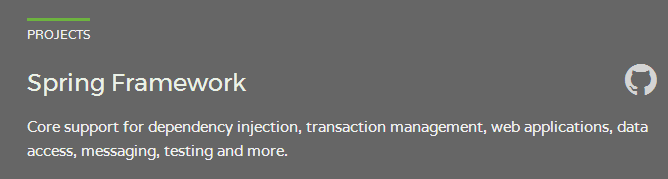
**Spring**是一个开源的简化企业级开发的容器框架。

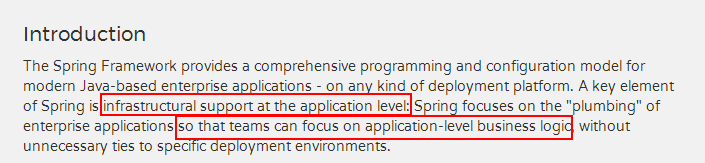
**目的**：简化JavaEE的开发

Spring发展至今，已经产生了关乎JavaEE企业级开发方方面面的框架，我们学习的是Spring Framework



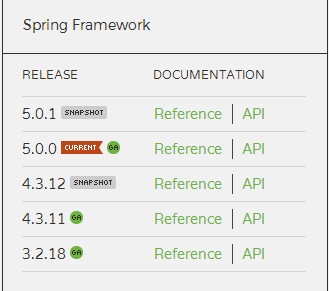
**地址**：https://spring.io/



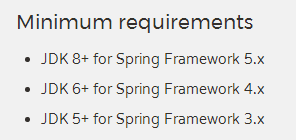


在应用程序层面提供最基础的支持；使开发团队可以专注于应用的业务逻辑开发。

**发行版**：



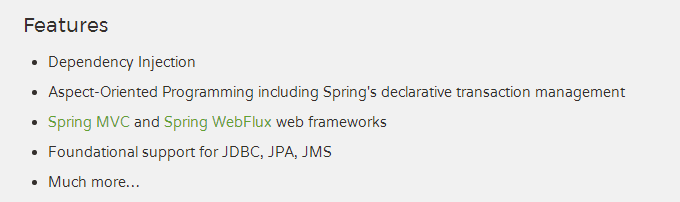
**版本要求**：



## 2. 显著特点

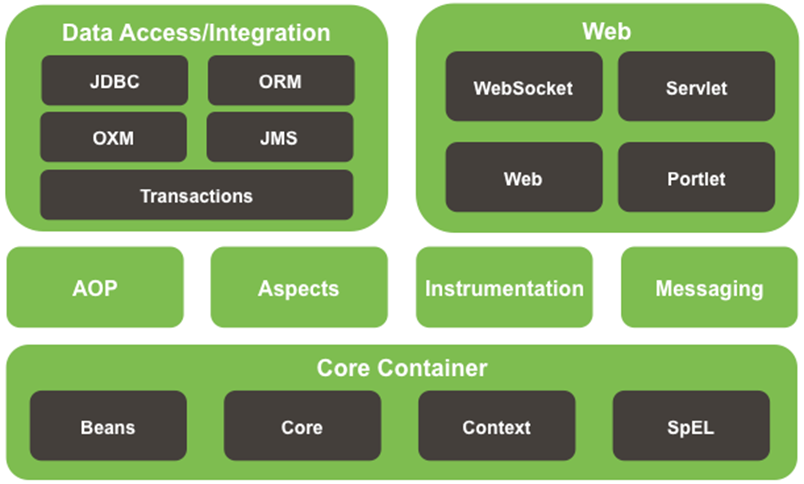
**IOC**（Inversion of control）控制反转 DI

**AOP**(Aspect Oriented Programming)的容器框架



## 功能丰富

几乎覆盖JavaEE开发的方方面面



## 4. Spring的优良特性

[1]**非侵入式**：基于Spring开发的应用中的对象可以不依赖于Spring的API

[2]**依赖注入**：DI——Dependency Injection，反转控制(IOC)最经典的实现。

[3]**面向切面编程**：Aspect Oriented Programming——AOP

[4]**容器**：Spring是一个容器，因为它包含并且管理应用对象的生命周期

[5]**组件化**：Spring实现了使用简单的组件配置组合成一个复杂的应用。在 Spring 中可以使用XML和Java注解组合这些对象。

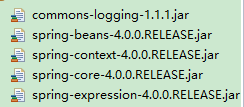
[6]**一站式**：在IOC和AOP的基础上可以整合各种企业应用的开源框架和优秀的第三方类库（实际上Spring 自身也提供了表述层的SpringMVC和持久层的Spring JDBC）。

## 5. Jar包描述：

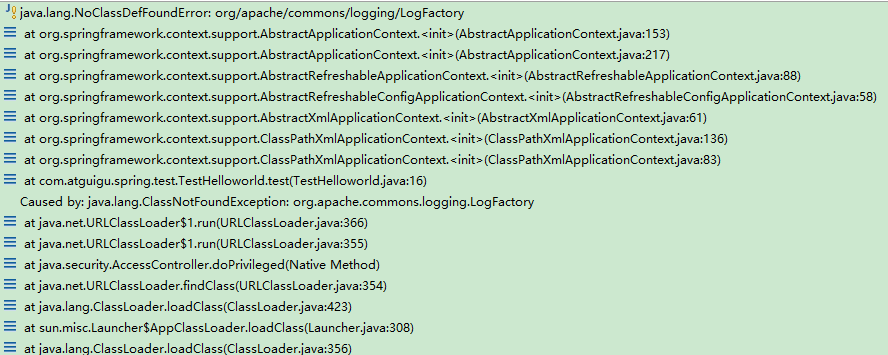
|  |  |
| --- | --- |
| 核心容器模块 | |
| org.springframework spring-beans | bean支持，包括Groovy |
| org.springframework spring-context | 运行时上下文，包括调度和远程调用抽象 |
| org.springframework spring-core | 核心库，被许多其它模块使用 |
| org.springframework spring-expression | Spring表达式语言 |
| AOP+Aspects模块 | |
| org.springframework spring-aop | 基于代理的AOP |
| org.springframework spring-aspects | 基于切面的AspectJ |
| 数据库访问模块 | |
| org.springframework spring-jdbc | JDBC支持包 |
| org.springframework spring-orm | 对象关系映射，包括对JPA和Hibernate支持 |
| org.springframework spring-oxm | 对象XML映射 |
| org.springframework spring-tx | 事务基础，包括对DAO的支持及JCA的集成 |
| org.springframework spring-jms | JMS支持包，包括发送和接收JMS消息的帮助类 |
| web应用开发模块 | |
| org.springframework spring-web | web支持包，包括客户端及web远程调用 |
| org.springframework spring-webmvc | REST web服务及web应用的MVC实现 |
| org.springframework spring-webmvc-portlet | 用于Portlet环境的MVC实现 |
| org.springframework spring-websocket | WebSocket和SockJS实现，包括对STOMP的支持 |
| 单元测试模块 | |
| org.springframework spring-test | 单元测试和集成测试组件 |
| 第三方集成 | |
| org.springframework spring-context-support | 包含用于集成第三方库到Spring上下文的类 |
| 消息框架 | |
| org.springframework spring-messaging | 消息处理的架构和协议 |
| 检测代理 | |
| spring-instrument-4.0.0.RELEASE | JVM引导的检测代理 |
| org.springframework spring-instrument-tomcat | tomcat的检测代理 |

## 6. helloworld

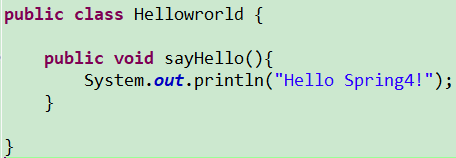
①导包



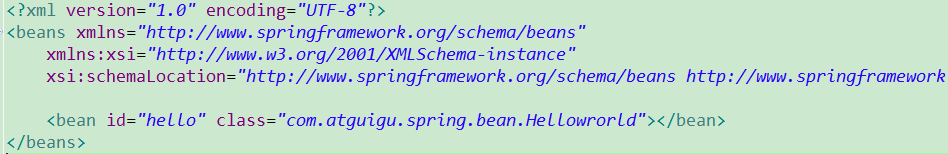
导入的jar包中包括，4个Spring的核心Jar包与其依赖的一个日志框架commons-logging，如果不加入，会报错如下：



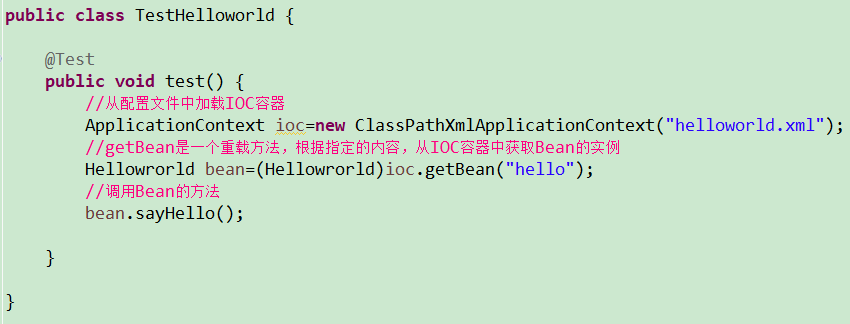
②编写一个类



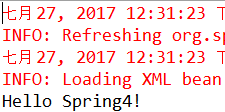
③编写配置文件，在配置文件中配置Bean



④进行测试



⑤测试通过



## 7. helloworld细节

①src目录：源码路径，也称为类路径。所有源码包里面的东西都会被合并在类路径里面：

在 java 工程中是: 项目名/bin/

在 javaweb工程中是 项目名/build/classes/

②使用容器，即 Application（应用上下文）接口来管理组件。Application有两个常用的实现类：

ClassPathXmlApplicationContext：从类路径下的XML配置文件中创建应用上下文对象

FileSystemXmlApplicationContext：从文件系统中的XML配置文件中创建应用上下文对象

③容器一创建就会自动创建配置文件中配置的bean，且默认创建的 bean是单例的

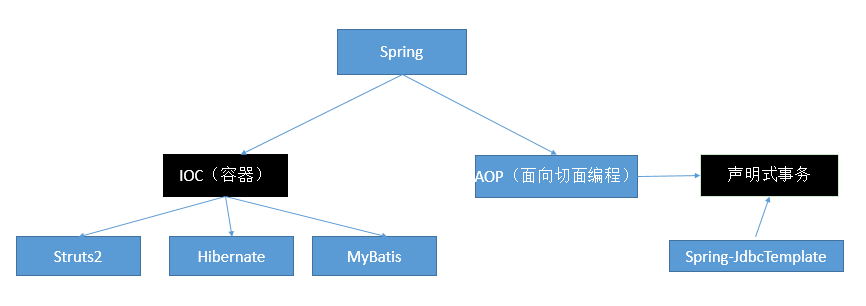
④getBean():从容器中取出bean

getBean(String id):通过id取得Bean，获取Bean的返回值类型是Object，需要强转。如果id不存在，会报错NoSuchBeanDefinitionException

getBean(Class T)：通过类型获取Bean，会返回指定类型的Bean，但是要保证容器中只有唯一此类型的Bean存在，否则会报错NoSuchUniqueBeanException

getBean(String id,Class T):从容器中取出指定id，指定类型的bean。

# 二、IOC



## 1．IOC

**I**nversion **o**f **C**ontrol：控制反转；

控制:资源的获取方式

①传统: 主动地获取资源（需要自己去创建所需的资源）

BookServlet{

BookService bs = new BookService();

//复杂的系统功能可能同时需要创建多个资源

XXXService xx=new xxxService();

public void test01(){

bs.checkout();//

}

②IOC方式：资源不需要自动创建，而是交给一个由应用程序组件所在的容器创建，并自动的注入到需要资源的组件中。

BookServlet{

BookService bs;

public void test01(){

bs.checkout();//

}

}

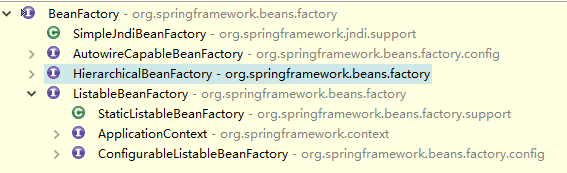
**优点**：开发人员不需要知道容器是如何创建资源对象的，只需要提供接收资源的方式即可，极大的降低了学习成本，提高了开发的效率

## 2. IOC思想在Spring中的实现

IOC在Spring中的典型实现就是容器。容器是Spring框架的核心。在通过IOC容器读取Bean的实例之前，需要先将IOC容器本身实例化。

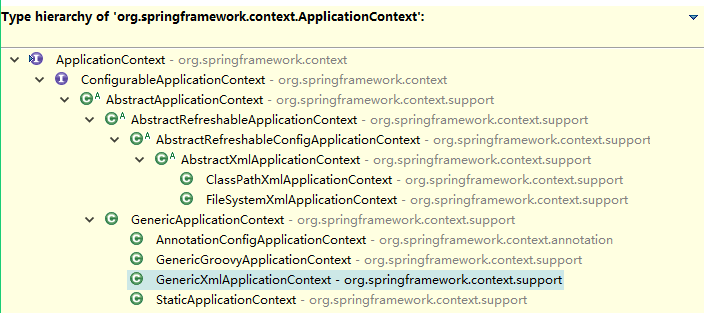
Spring提供了IOC容器的两种实现方式

(1)BeanFactory：IOC容器的基本实现，是Spring内部的基础设施，是面向Spring本身的，不是提供给开发人员使用的



(2)ApplicationContext（应用上下文）：BeanFactory的子接口，提供了更多高级特性。面向Spring的使用者，几乎所有场合都使用ApplicationContext而不是底层的BeanFactory。

ApplicationContext的实现类



## 3. Bean属性的赋值

### 3.1 属性赋值的三种方式

#### 3.1.1 <property>标签赋值

<property>标签通过setXXX()赋值，其中name属性对应的是setXxx()方法中的Xxx。

<bean class=*"com.atguigu.spring.bean.Person"* id=*"gg"*>

<!--使用property标签赋值

name：属性名

value：属性值

-->

<property name=*"name"* value=*"国哥"*/>

<property name=*"age"* value=*"18"*/>

</bean>

#### 3.1.2 构造器赋值

<bean class=*"com.atguigu.spring.bean.Person"* id=*"dg"*>

<constructor-arg type=*"java.lang.Integer"* index=*"1"* value=*"20"*/>

<constructor-arg name=*"name"* value=*"岱哥"*/>

</bean>

name: 构造器参数名 掌握！

value: 参数值。 只有value属性时，默认按照构造器参数顺序赋值

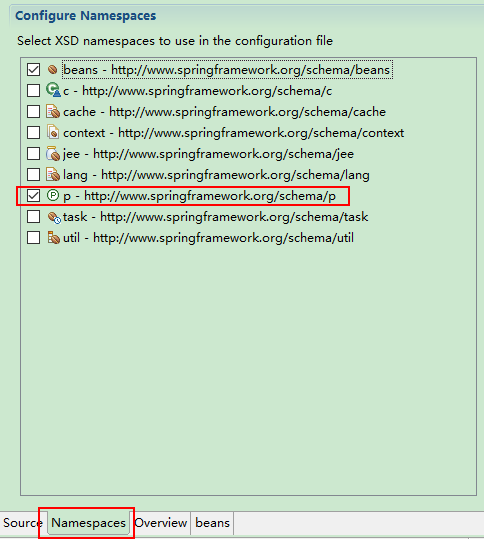
index: 明确指定参数的索引位置

type: 明确指定构造器参数类型

ref: 为引述数据类型赋值

#### 3.1.3 P名称空间赋值

使用p名称空间的前提需要导入p名称空间：



格式：p:xxx(setxXX()的xxx)



### 3.2 可以用值的类型

#### 3.2.1 字面量

字面量表示可以使用字符串表示的值，包括基本数据类型及其包装类，String。

字面量可以通过value属性或<value>标签来指定。

#### 3.2.2 特殊字符

赋值的过程中如果遇到特殊字符，可以使用转义字符或者CDATA块来解决。

<bean class=*"com.atguigu.spring.bean.Person"* id=*"fl"*>

<!--特殊字符：可以使用转义字符来解决 -->

<property name=*"name"* value=*"&lt;苍飞龙&gt;"*/>

<!--使用CDATA包装 -->

<property name=*"gender"* ><value><![CDATA[<汉子>]]></value></property>

</bean>

#### 3.2.3 null值

null值不属于字面量，需要使用<null/>来赋值！

<bean class=*"com.atguigu.spring.bean.Person"* id=*"cfl"*>

<!-- <property name="gender" value="null"></property> -->

<!-- 使用null标签-->

<property name=*"gender"*><null/></property>

</bean>

#### 3.2.4 外部bean

外部bean就是普通的bean，声明在外部，可以通过ref属性结合id直接引用！

<bean class=*"com.atguigu.spring.bean.Person"* id=*"cfl"*>

<property name=*"name"* value=*"&lt;苍飞龙&gt;"*/>

<property name=*"phone"* ref=*"phone"*/>

</bean>

<bean class=*"com.atguigu.spring.bean.Phone"* id=*"phone"*>

<property name=*"brand"* value=*"Mi"*/>

<property name=*"type"* value=*"7"*/>

</bean>

#### 3.2.5 内部bean

<bean class=*"com.atguigu.spring.bean.Person"* id=*"fl"*>

<property name=*"phone"* >

<!-- 内部bean只能被当前的bean所使用，无法通过id获取 -->

<bean class=*"com.atguigu.spring.bean.Phone"* >

<property name=*"brand"* value=*"Mi"*/>

<property name=*"type"* value=*"7"*/>

</bean>

</property>

</bean>

内部bean代表只能被这个bean内部使用的一个bean，无法被外部访问，因此id属性无意义。

### 3.3 集合类型赋值

|  |  |  |
| --- | --- | --- |
| 集合类型 | 赋值标签 | 备注 |
| Array | <array> |  |
| List | <list> | 使用Arraylist实现 |
| Set | <set> | 使用linkedHashSet实现 |
| Map | <map> | 使用linkedHashMap实现；  使用<entry>标签为map中的 元素赋值 |
| Properties | <props> | 使用<prop>标签为其中的元素赋值 |

<bean class=*"com.atguigu.spring.bean.CollectionBean"* id=*"cb"*>

<property name=*"array"*>

<array>

<value>123</value>

<value>你好！</value>

<bean class=*"com.atguigu.spring.bean.Person"* >

<property name=*"name"* value=*"国哥"*/>

</bean>

<!-- 引用一个外部bean -->

<ref bean=*"gg"*/>

<!--拿到一个bean的id -->

<idref bean=*"gg"*/>

</array>

</property>

<property name=*"list"*>

<list>

<bean class=*"com.atguigu.spring.bean.Person"* >

<property name=*"name"* value=*"国哥"*/>

</bean>

<ref bean=*"gg"*/>

</list>

</property>

<property name=*"set"* >

<set>

<bean class=*"com.atguigu.spring.bean.Phone"*>

<property name=*"brand"* value=*"Mi"*/>

<property name=*"type"* value=*"7"*/>

</bean>

</set>

</property>

<property name=*"map"*>

<map>

<entry key=*"国哥"* value-ref=*"gg"*></entry>

<entry key=*"国哥2"*>

<bean class=*"com.atguigu.spring.bean.Person"* >

<property name=*"name"* value=*"国哥"*/>

</bean>

</entry>

</map>

</property>

<property name=*"properties"*>

<props>

<prop key=*"username"*>admin</prop>

<prop key=*"password"*>admin</prop>

</props>

</property>

</bean>

### 3.4 Utils名称空间

上述的集合类型赋值，相当于内部bean，无法被外部的bean引用。

如果集合需要被多个Bean来同时引用；或者需要自定义集合的初始化类型，可以使用Utils名称空间来声明集合。首先需要开启Utils名称空间：

<util:map id=*"myMap"* key-type=*"java.lang.String"*

value-type=*"com.atguigu.spring.bean.Person"*

map-class=*" java.util.HashMap"*>

<entry key=*"国哥"* value-ref=*"gg"*></entry>

<entry key=*"国哥2"*>

<bean class=*"com.atguigu.spring.bean.Person"* >

<property name=*"name"* value=*"国哥"*/>

</bean>

</entry>

</util:map>

### 3.5 级联属性赋值

格式：属性.属性。例如以下的例子，在person对象中为级联的phone属性赋值！

<!-- 级联属性赋值 -->

<bean class=*"com.atguigu.spring.bean.Person"* id=*"cfl"*>

<property name=*"phone"* ref=*"phone"*/>

<!-- name ：属性.属性 -->

<property name=*"phone.type"* value=*"Mix2"*/>

<property name=*"phone.brand"* value=*"Mi"*></property>

</bean>

<bean class=*"com.atguigu.spring.bean.Phone"* id=*"phone"*/>

### 3.6 SPEL

Spring Expression Language，Spring表达式语言，简称SpEL。支持运行时查询并可以操作对象图。

和JSP页面上的**EL**表达式、Struts2中用到的**OGNL**表达式一样，SpEL根据JavaBean风格的**getXxx()、setXxx()方法定义的属性**访问对象图，**完全符合我们熟悉的操作习惯**。

SpEL使用#{…}作为定界符，所有在大框号中的字符都将被认为是SpEL表达式。

SpEL支持字面量的赋值，以及引用其他bean及其属性，调用实例方法和静态方法，并支持一些运算符。

|  |  |
| --- | --- |
| 常用数据 | 格式 |
| 字面量 | #{字面量}，其中字符串使用单引号 |
| 引用bean | #{beanId}，类似于ref |
| 引用bean的属性 | #{beanid.property} |
| 调用bean的方法 | #{beanId.method()} |
| 调用静态方法 | #{T(全类名).xxx()} |

<bean class=*"com.atguigu.spring.bean.Person"* id=*"gg"*>

<!-- 字面量 #{'字符串'} -->

<property name=*"name"* value=*"#{'国哥'}"*/>

<!-- 为引用数据类型赋值 #{id} -->

<property name=*"phone"* value=*"#{phone}"*/>

<!-- 获取一个引用数据类型的属性 #{id.property} -->

<property name=*"gender"* value=*"#{cfl.gender}"*/>

<!-- 调用对象的方法 #{id.xxx()}-->

<!-- <property name="age" value="#{cfl.getAge()}"/> -->

<!-- 调用静态方法： #{T(全类名).xxx()} -->

<property name=*"age"*

value=*"#{T(com.atguigu.spring.bean.CollectionBean).getAge()\*6}"*/>

</bean>

支持运算符运算

①算术运算符：+、-、\*、/、%、^

②字符串连接：+

③比较运算符：<、>、==、<=、>=、lt、gt、eq、le、ge

④逻辑运算符：and, or, not, |

⑤三目运算符：判断条件?判断结果为true时的取值:判断结果为false时的取值

⑥正则表达式：matches

## 4．工厂方式创建Bean

### 4.1 静态工厂创建Bean

静态工厂创建Bean，是通过静态工厂类.静态方法来获取一个Bean。

**public** **class** StaticFactory {

**private** **static** Map<String, Phone> *phones*;

**static**{

*phones*=**new** HashMap<String, Phone>();

*phones*.put("小米", **new** Phone("Mi", "Mix2"));

*phones*.put("华为", **new** Phone("Huawei", "Mate10"));

}

**public** **static** Phone getPhone(String key){

**return** *phones*.get(key);

}

}

传统方式使用的是StaticFactory.getPhone(“xx”);来获取一个Phone对象；

在Spring中配置的原理是一样的！

<!-- StaticFactory.getPhone("小米"); -->

<bean class=*"com.atguigu.spring.factory.StaticFactory"* factory-method=*"getPhone"* id=*"phone1"*>

<constructor-arg value=*"小米"*/>

</bean>

### 4.2 实例工厂创建Bean

实例工厂是通过调用工厂类的实例的工厂方法来获取Bean。

调用方式：实例对象.工厂方法();

**public** **class** InstanceFactory {

**private** Map<String, Phone> phones;

{

phones=**new** HashMap<String, Phone>();

phones.put("小米", **new** Phone("Mi", "Mix2"));

phones.put("华为", **new** Phone("Huawei", "Mate10"));

}

**public** Phone getPhone(String key){

**return** phones.get(key);

}

}

<!-- new InstanceFactory().getPhone("华为"); -->

<!-- 先创建出实例工厂对象 -->

<bean class=*"com.atguigu.spring.factory.InstanceFactory"*

id=*"if"*>

</bean>

<!--factory-bean：实例工厂的id factory-method：工厂方法名-->

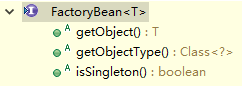
<bean id=*"phone2"* factory-bean=*"if"* factory-method=*"getPhone"*>

<constructor-arg value=*"华为"*/>

</bean>

### 4.3 实现FactoryBean接口

实现Spring框架提供的FactoryBean接口org.springframework.beans.factory.FactoryBean，可以快速地创建出工厂类。



**public** **class** PhoneFactory **implements** FactoryBean<Phone>{

**private** String brand;

**private** String type;

**public** String getBrand() {

**return** brand;

}

**public** **void** setBrand(String brand) {

**this**.brand = brand;

}

**public** String getType() {

**return** type;

}

**public** **void** setType(String type) {

**this**.type = type;

}

//返回指定的对象

@Override

**public** Phone getObject() **throws** Exception {

**return** **new** Phone(brand, type);

}

//返回对象的类型

@Override

**public** Class<?> getObjectType() {

**return** Phone.**class**;

}

//创建对象的模式是否是单例的

@Override

**public** **boolean** isSingleton() {

**return** **false**;

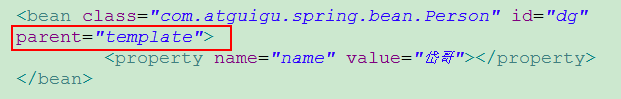
}

}

## 5. Bean的常用属性

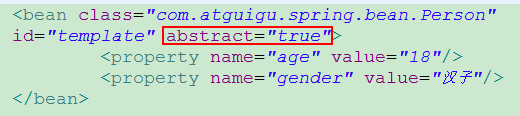
### 5.1 parent

Bean之间配置信息的复制。



### 5.2 abstract

声明一个模板Bean,如果一个Bean作为模版供其他的Bean来继承和复用，那么这个Bean实例化是没有任何意义的，可以使用abstract属性来声明为抽象Bean。



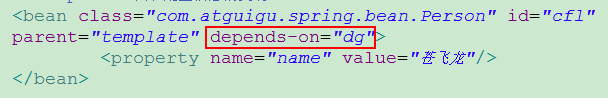
当指定这个属性后，将不能再获取此 bean的实例，否则会报错：

org.springframework.beans.factory.BeanIsAbstractException:

### 5.3 depends-on

使用 deponds-on 属性来指定 bean之间的依赖关系（bean的创建顺序）。

当不显式指定时，bean的默认创建顺序是按照配置文件的注册顺序。



### 5.4 scope

scope代表bean的作用域。

prototype：多实例，容器不会自动创建。每次调用getBean()，都会创建一个新的实例

singleton(默认):单例的，随着容器的创建而创建，且在容器中只有一个实例

session(不用)：web环境下，一次回话创建一次实例

request(不用)：web环境下，一次请求创建一个实例

### 5.5 Bean的生命周期

生命周期：从创建到销毁的过程。

init-method:自定义的初始化方法，会在bean赋值之后调用，不能有参数，但是可以抛出异常。

destroy-method:自定义的初始化方法，会在容器销毁之后调用，不能有参数，但是可以抛出异常。

注意：

①多实例模式的bean的destroy-method不会调用。

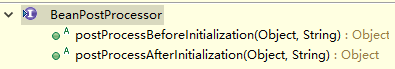
②容器如何关闭：使用ConfigurableApplicationContext类型的close();



单实例：容器创建（bean的构造器）---赋值----init-method()---容器销毁---destroy-method;

多实例：容器创建---getBean(bean的构造器) ---赋值----init-method()---容器销毁

### 5.6 Bean的后置处理器



bean的后置处理器是一个接口，实现此接口后，可以在bean被初始化后，完成一些其他的操作。

不管bean有没有初始化方法，后置处理器都会默认此 bean有初始化方法。

参数说明：

* + - * Object bean:要操作的 bean；
      * String beanName:bean的名称；
      * postProcessBeforeInitialization:在bean初始化方法之前调用；
      * postProcessAfterInitialization：在bean初始化方法之后调用；

执行顺序：

容器创建（bean的构造器）----赋值--postProcessBeforeInitialization--init-method--postProcessAfterInitialization --容器close()---destroy-method

### 5.7 引入外部属性文件

以使用 Spring来配置 C3P0数据源为例：

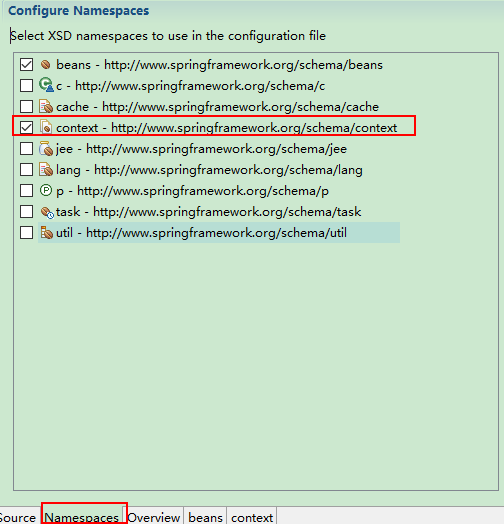
user=root

password=1234

jdbcUrl=jdbc:mysql:///query\_data

driverClass=com.mysql.jdbc.Driver

①首先引入context名称空间



②使用<context:property-placeholder>标签的location属性来制定配置文件的位置

③使用${key}来引入配置文件中的键值对。

<context:property-placeholder location=*"classpath:jdbc.properties"*/>

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<property name=*"user"* value=*"${user}"*/>

<property name=*"password"* value=*"${password}"*/>

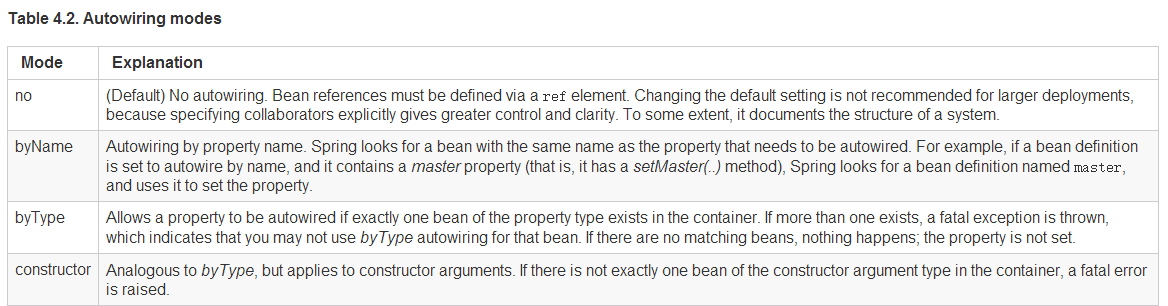
<property name=*"jdbcUrl"* value=*"${jdbcUrl}"*/>

<property name=*"driverClass"* value=*"${driverClass}"*/>

</bean>

注意：在属性文件中，尽量不要使用username作为Key，windows环境下#{username}可能会获取系统用户名。

### 5.8 autowire



之前为自定义的引用数据类型赋值，可以使用ref属性，手动制定装配。

现在我们可以使用autowire属性，此时Spring会自动从容器中查找符合条件的bean，为我们自动赋值。

|  |  |
| --- | --- |
| Autowire属性可选项 | 装配规则 |
| default/no | 不装配 |
| byName | 将属性名作为id来进行匹配；如果id匹配，但是类型不符合，报错类型转换异常。 |
| byType | 参考类型进行装配；如果容器中存在多个同一类型，报错NoUniqueBeanDefinationException |
| constructor | 要求必须有单个赋值参数的构造器；  ①按照类型过滤  ②如果有多个所需类型，继续按照构造器参数名称作为id，继续匹配  ③匹配不上，就不装配，不会报错。 |

## 6. DI(依赖注入)

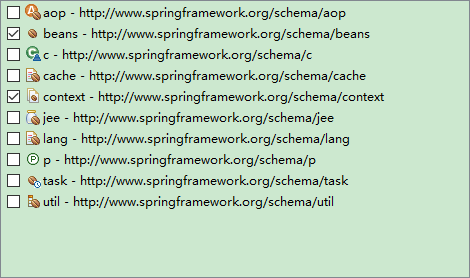
之前我们已经学习了通过配置文件的方式来装配组件。除了配置文件的方式，我们还可以使用注解来实现组件的自动创建和装配。

相对于xml方式的自动装配而言，通过注解的方式配置bean会更加的优雅和简洁，并与MVC的组件化开发理念契合，是开发汇总常用的使用方式

要使用自动装配，还需要导入 AO的包（spring-aop-4.0.0.RELEASE），否则，会报错：java.lang.NoClassDefFoundError: org/springframework/aop/TargetSource

### 6.1 容器自动创建Bean

要使用DI功能，需要首先开启context名称空间：



之后使用<context:component-scan>标签指定要扫描的父路径。

<context:component-scan base-package=*"com.atguigu.spring"* >

之后容器会扫描指定的路径，为标有指定注解的bean，自动注册为spring的bean，支持以下注解：

@Component,： 标识一个普通的对象

@Repository：标识一个持久化层的组件

@Service： 标识一个服务层的组件

@Controller ：标识一个控制层的组件

注意：

①这四个注解功能是一样的。主要是为了给开发人员看，以区别每个类的功能

②base-package:指定扫描的父路径，会自动扫描其下所有子路径

③默认将类名首字母小写作为id，创建组件；还可以通过注解的value属性，手动指定id。

### 6.2 容器自动装配Bean

①在需要自动赋值的属性上标注@Autowired注解，容器会自动扫描容器中的组件，并自动赋值。

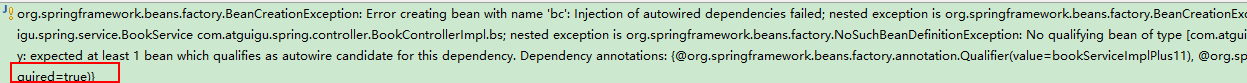


②自动装配的规则：

首先按照类型匹配，如果匹配到单个，就直接装配；

如果匹配到多个，继续按照属性名作为id进一步匹配；

如果匹配不到，报错！报错如下图，原因是@Autowired默认属性required=true，意思就是强制装配。



③取消强制装配：

修改@Autowired注解的required属性为false即可

④指定装配指定id的组件：

在要装配的组件上标注@Quliafier(xxx)后，将按照xxx作为id来进行装配！

⑤一般情况下，需要将@Quliafier和required=false注解结合使用，以防止容器中无法匹配到指定id的bean从而报错。

⑥@Autowired属性还可以标注在方法参数上，此时这个方法会在Bean创建后就立刻执行。

@Autowired

**public** **void** init(BookDao bookDao,@Qualifier("bookServiceImpl2")BookService bookService){

System.***out***.println("BookDao"+bookDao);

System.***out***.println("BookService"+bookService);

}

⑦@Autowired与@Resource的区别

@Resource JSR250 (Common Annotations for Java)这是jsr250规范的实现，通过 ‘CommonAnnotationBeanPostProcessor’ 类实现依赖注入。

与@Autowired的区别：

@Autowired：org.springframework.beans.factory.annotation.Autowired 是Spring框架自己的注解

@Resource：javax.annotation.Resource 是Java的一种标准，扩展性强，可以被EJB等容器框架识别

### 6.3 容器扫描的过滤与包含

在<context:component-scan>中可以通过声明<include-filter>标签来指定只扫描哪些组件；

在<context:component-scan>中可以通过声明<exclude-filter>标签来指定不扫描哪些组件；

<include-filter>和<exclude-filter>可以存在多个；

<include-filter>必须结合use-default-filters="false"属性配合使用才有效；

常见的过滤属性如下：

type： 过滤规则的类型

* + - * annotation：注解，传入注解全类名，所有标注了这个注解的类，都会被过滤掉

按照@Component注解进行过滤，会过滤所有的组件！

* + - * assignable： 签名,传入全类名或者接口名
      * aspectJ ： 参考aspectJ表达式
      * regex： 参考正则表达式
      * custom ： 自定义，需要实现TypeFilter接口

expression：表达式

### 6.4 配置文件和注解开发的选择

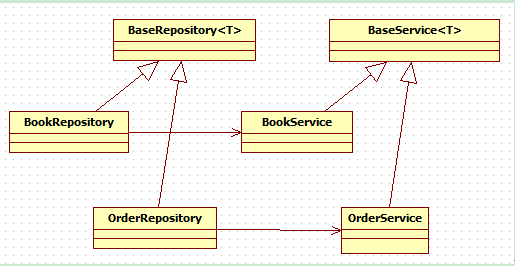
一些简单的业务逻辑，可以使用注解快速开发！

但是，比较重要的类，可能经常做修改的类，我们一般放到配置文件中！

第三方编写的类，只能在配置文件中配置！

具体参考团队开发约束和风格！

## 7. 泛型依赖注入



**结论**：在装配组件时，泛型也是参考的条件之一

# 三、AOP

## 1. AOP是什么

AOP(Aspect-Oriented Programming，面向切面编程)：是一种新的方法论，是对传统 OOP(Object-Oriented Programming，面向对象编程)的补充。

AOP编程操作的主要对象是切面(aspect)，而切面模块化横切关注点。

AOP的好处：

* 每个事物逻辑位于一个位置（分层解耦），代码不分散，便于维护和升级
* 业务模块更简洁，只包含核心业务代码

AOP的作用：AOP可以将一段代码动态地添加到指定对象（目标对象）的指定方法（目标方法）的指定位置运行！

AOP的原理就是动态代理。

## 2. 引入

①计算器接口

**public** **interface** Calculator {

**int** add(**int** i,**int** j);

**int** sub(**int** i,**int** j);

**int** mul(**int** i,**int** j);

**int** div(**int** i,**int** j);

}

②简单的计算器实现

**public** **class** EasyCalculator **implements** Calculator{

@Override

**public** **int** add(**int** i, **int** j) {

**int** result=i+j;

**return** result;

}

@Override

**public** **int** sub(**int** i, **int** j) {

**int** result=i-j;

**return** result;

}

@Override

**public** **int** mul(**int** i, **int** j) {

**int** result=i\*j;

**return** result;

}

@Override

**public** **int** div(**int** i, **int** j) {

**int** result=i/j;

System.***out***.println("目标方法执行了！");

**return** result;

}

}

③需求更改，添加日志记录需求

**public** **class** LoggerCalculator **implements** Calculator{

@Override

**public** **int** add(**int** i, **int** j) {

System.***out***.println("\*\*\*"+"add"+"\*\*\*方法开始执行！参数是："+i+","+j);

**int** result=i+j;

System.***out***.println("\*\*\*"+"add"+"\*\*\*方法结束执行！结果是："+result);

**return** result;

}

@Override

**public** **int** sub(**int** i, **int** j) {

System.***out***.println("["+"sub"+"]方法开始执行！参数是："+i+","+j);

**int** result=i-j;

System.***out***.println("["+"sub"+"]方法结束执行！结果是："+result);

**return** result;

}

@Override

**public** **int** mul(**int** i, **int** j) {

System.***out***.println("["+"mul"+"]方法开始执行！参数是："+i+","+j);

**int** result=i\*j;

System.***out***.println("["+"mul"+"]方法结束执行！结果是："+result);

**return** result;

}

@Override

**public** **int** div(**int** i, **int** j) {

System.***out***.println("["+"div"+"]方法开始执行！参数是："+i+","+j);

**int** result=i/j;

System.***out***.println("["+"div"+"]方法结束执行！结果是："+result);

**return** result;

}

}

特点：业务代码（计算功能）和辅助功能（日志记录）高耦合。后续如果需要维护，会频繁地更改这个类的源代码。

④如果应对需求的频繁更改

可以将频繁修改的辅助业务，抽取为一个独立的方法，后期只需要修改辅助方法，而不需要修改业务代码。

**public** **class** LoggerUtil {

**public** **static** **void** beforeDo(String methodName,Object...args){

System.***out***.println("\*\*\*\*\*\*"+methodName+"\*\*\*\*\*\*方法开始执行！参数是："+Arrays.*asList*(args));

}

**public** **static** **void** afterDo(String methodName,Object result){

System.***out***.println("\*\*\*"+methodName+"\*\*\*方法结束执行！结果是："+result);

}

}

缺点：

①业务代码和辅助功能的代码还是耦合在一起，后期维护性差，可读性差；

②在调用辅助功能的方法时，需要手动传入方法名；

⑤动态代理来解决

动态代理是指客户通过代理对象来调用目标对象的方法，并且是在程序运行时根据需要动态创建目标类的代理对象。

Java中可以通过实现InvocationHandler接口来创建动态代理。

**public** **class** DynamicProxy **implements** InvocationHandler{

//目标对象

**private** Object target;

//获取代理对象

**public** Object getProxy(Object target){

**this**.target=target;

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

}

//代理对象调用目标方法

@Override

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

**throws** Throwable {

//获取方法名

String methodName = method.getName();

//获取参数 参数就是Object[] args

System.***out***.println(methodName+"\*\*\*\*\*\*方法开始执行！参数是："+Arrays.*asList*(args));

//执行目标方法

Object result=**null**;

**try** {

result = method.invoke(target, args);

System.***out***.println(methodName+"方法正常运行结束！结果是："+result);

} **catch** (Exception e) {

e.printStackTrace();

//拿到引起当前异常的异常

Throwable cause = e.getCause();

**if** (**null** !=cause) {

e=(Exception) cause;

}

System.***out***.println(methodName+"方法出现了异常！异常是："+e.getClass().getSimpleName()

+"异常的原因是："+e.getMessage());

}**finally**{

System.***out***.println(methodName+"方法最终结束！");

}

**return** result;

}

}

动态代理的优点：业务方法只包含纯粹的业务功能，而辅助功能编写在代理对象的方法上。实现了分层解耦，可读性强，维护方便。

缺点：

①默认的JDK动态代理只能为接口提供动态代理；

②使用默认的JDK动态代理编写略显复杂（对比Spring）

## 3. AOP的相关概念

动态代理中的核心概念：目标类、代理类（切面）；

AOP中常见的概念：切面、通知、切入点、连接点、目标对象、代理对象、横切关注点

切面： java中的类，切面里面有很多的通知；

通知： 方法（完成某个功能），根据通知之于目标方法执行位置的不同，进行了分类；

前置通知：在目标方法之前执行；

后置通知：在目标方法之后执行

返回通知：目标方法正常返回的时候执行；

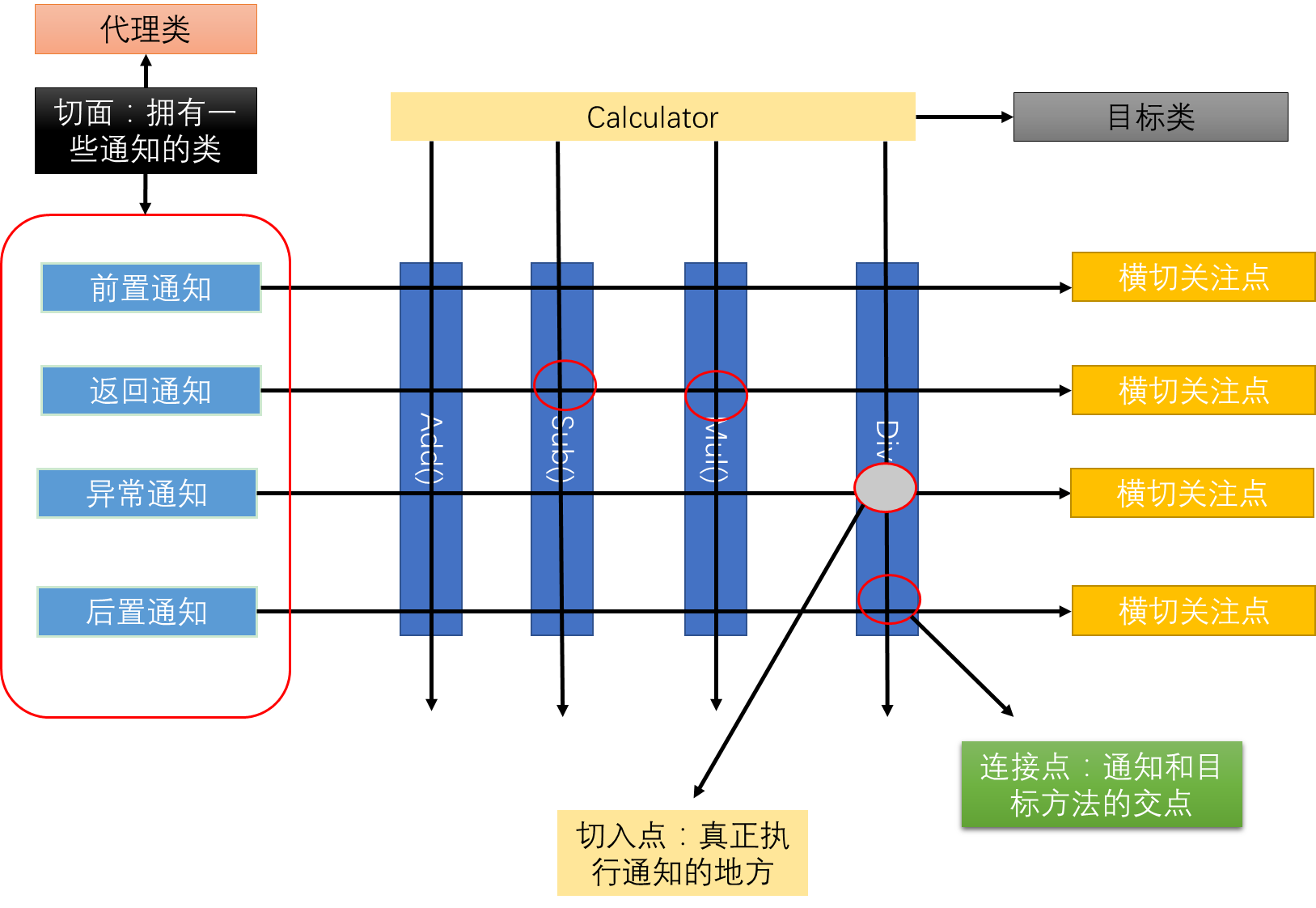
异常通知：目标方法发生异常时执行；

环绕通知：可以模拟上面四种通知的功能，且可以控制目标方法执行！

横切关注点： 通知执行的一个抽象的位置；

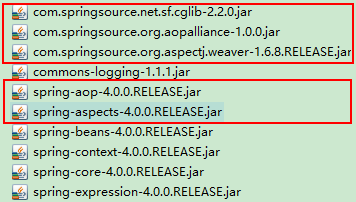
连接点： 通知和目标方法产生关联的一个点；

切入点： 真正执行通知方法的地方；



## 4. Spring实现AOP

①导包



②在配置文件中开启AOP的配置



<!--指定自动扫描的包 -->

<context:component-scan base-package=*"com.atguigu.spring"*/>

<!-- 识别aspectJ风格的注解，并自动地创建代理对象 -->

<aop:aspectj-autoproxy/>

## 5. 使用注解实现AOP

### 5.1 声明切面

使用@Aspect来声明一个切面类。

@Aspect

@Component

**public** **class** LoggerAspect

### 5.2 切入点表达式

#### 5.2.1 声明切入点表达式

**表达式的方式**定位**一个或多个**具体的连接点。

①切入点表达式的语法格式

|  |
| --- |
| execution([权限修饰符] [返回值类型] [简单类名/全类名] [方法名]([参数列表])) |

②在AspectJ中，切入点表达式可以通过 “&&”、“||”、“!”等操作符结合起来。

|  |  |
| --- | --- |
| 表达式 | execution (\* \*.add(int,..)) **||** execution(\* \*.sub(int,..)) |
| 含义 | 任意类中第一个参数为int类型的add方法或sub方法 |

③支持\*、..等通配符

|  |
| --- |
| \*:   * 匹配一或者多个字符 * 匹配任意一个参数（只能匹配1层路径） * 权限位置，可以不写，默认就是public |
| ..   * 匹配任意个数，任意类型的参数 * 匹配多层路径 |
| 最模糊的匹配：\* \*(..)或者可以写为 \* \*.\*(..) |

#### 5.2.2 抽取切入点表达式

使用@Pointcut注解来抽取切入点表达式，其中value属性为其指向的切点表达式。抽取完后，只需要复用此方法，即代表这个切点表达式。

@Pointcut(value="execution(int com.atguigu.spring.impl.EasyCalculator.\*(..))")

**public** **void** myPointCut(){};

### 5.3 声明通知

#### 5.3.1 前置通知

使用@Before注解来声明，其中value属性为必须属性，传入切入点表达式。

@Before(value="execution(int com.atguigu.spring.impl.EasyCalculator.\*(..))")

**public** **void** beforeAdvice(JoinPoint jpPoint){

//获取目标方法名

String methodName = jpPoint.getSignature().getName();

//获取目标方法参数

Object[] args = jpPoint.getArgs();

System.***out***.println("【LA前置通知】"+methodName+"方法开始执行！参数是"+Arrays.*asList*(args));

}

在通知中，可以通过声明一个JoinPoint参数来获取连接点的信息，比如获取方法名和目标参数。

#### 5.3.2 后置通知

使用@After注解来声明一个后置通知！

@After(value = "myPointCut()")

**public** **int** afterAdvice(JoinPoint jpPoint){

String methodName = jpPoint.getSignature().getName();

System.***out***.println("【LA后置通知】"+methodName+"方法最终结束！");

**return** 1;

}

#### 5.3.3 返回通知

使用@AfterReturing注解来声明一个返回通知。

@AfterReturning(value = "myPointCut()",returning="result")

**public** **void** returnAdvice(JoinPoint jpPoint,Object result){

String methodName = jpPoint.getSignature().getName();

System.***out***.println("【LA返回通知】"+methodName+"方法正常运行结束！结果是："+result);

}

注意:对于通知来说，不会对返回值类型进行要求，但是会对通知方法声明的参数进行严格的检查。

我们可以使用returning属性来明确制定我们需要返回值参数。

#### 5.3.4 异常通知

使用@AfterThrowing来声明一个异常通知，使用throwing参数来声明要捕捉的异常对象。

@AfterThrowing(value="myPointCut()",throwing="e")

**public** **void** exceptionAdvice(JoinPoint jpPoint,Exception e){

String methodName = jpPoint.getSignature().getName();

System.***out***.println("【LA异常通知】"+methodName+"方法出现了异常！异常是："+e.getClass().getSimpleName()+"异常的原因是："+e.getMessage());

}

#### 5.3.5 环绕通知

@Around(value="myPointCut()")

**public** Object aroundAdvice(ProceedingJoinPoint pjp){

//环绕通知可以模拟实现上述四种通知的功能！ 类似动态代理的invoke()

//获取方法名

String methodName = pjp.getSignature().getName();

//获取参数列表

Object[] args = pjp.getArgs();

//模拟前置通知

System.***out***.println("【LA环前通知】"+methodName+"方法开始执行！参数是："+Arrays.*asList*(args));

//执行目标方法

Object result=**null**;

**try** {

result = pjp.proceed(args);

//模拟返回通知

System.***out***.println("【LA环返通知】"+methodName+"方法正常运行结束！结果是："+result);

} **catch** (Throwable e) {

//模拟异常通知

System.***out***.println("【LA环异通知】"+methodName+"方法出现了异常！异常是："+e.getClass().getSimpleName()+"异常的原因是："+e.getMessage());

}**finally**{

//模拟后置通知

System.***out***.println("【LA环后通知】"+methodName+"方法最终结束！");

}

**return** result;

}

使用@Aroud来声明一个环绕通知。环绕通过可以通过自己编写代码的形式，模拟前面四种通知的效果。而环绕通知强大的地方或者说区别于前面四种通知的地方在于，环绕通知可以控制目标方法的执行。

使用ProceedingJoinPoint类型的对象，不仅可以获取连接点出的方法名和方法参数，还可以通知执行proceed()方法来执行目标方法。

注意：如果存在普通通知和环绕通知结合使用的情况，环绕通知的结果一定要正确返回，异常要正常抛出，否则会影响其他通知的运行！

#### 5.3.6 如何选用通知

两种类型的通知，事实上使用一种即可！而且在真实的开发环境中不会同时存在！

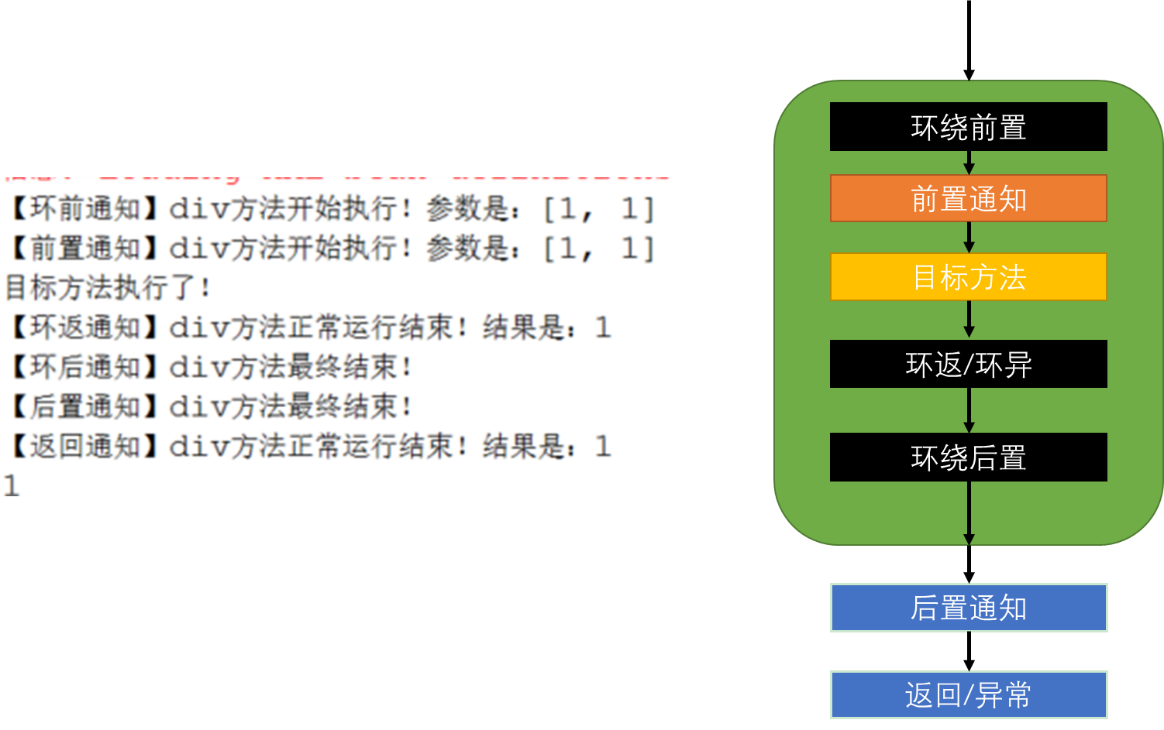
如何用？根据需求而定！

①如果需要对目标方法的运行结果做修改或者校验，只能使用环绕通知！

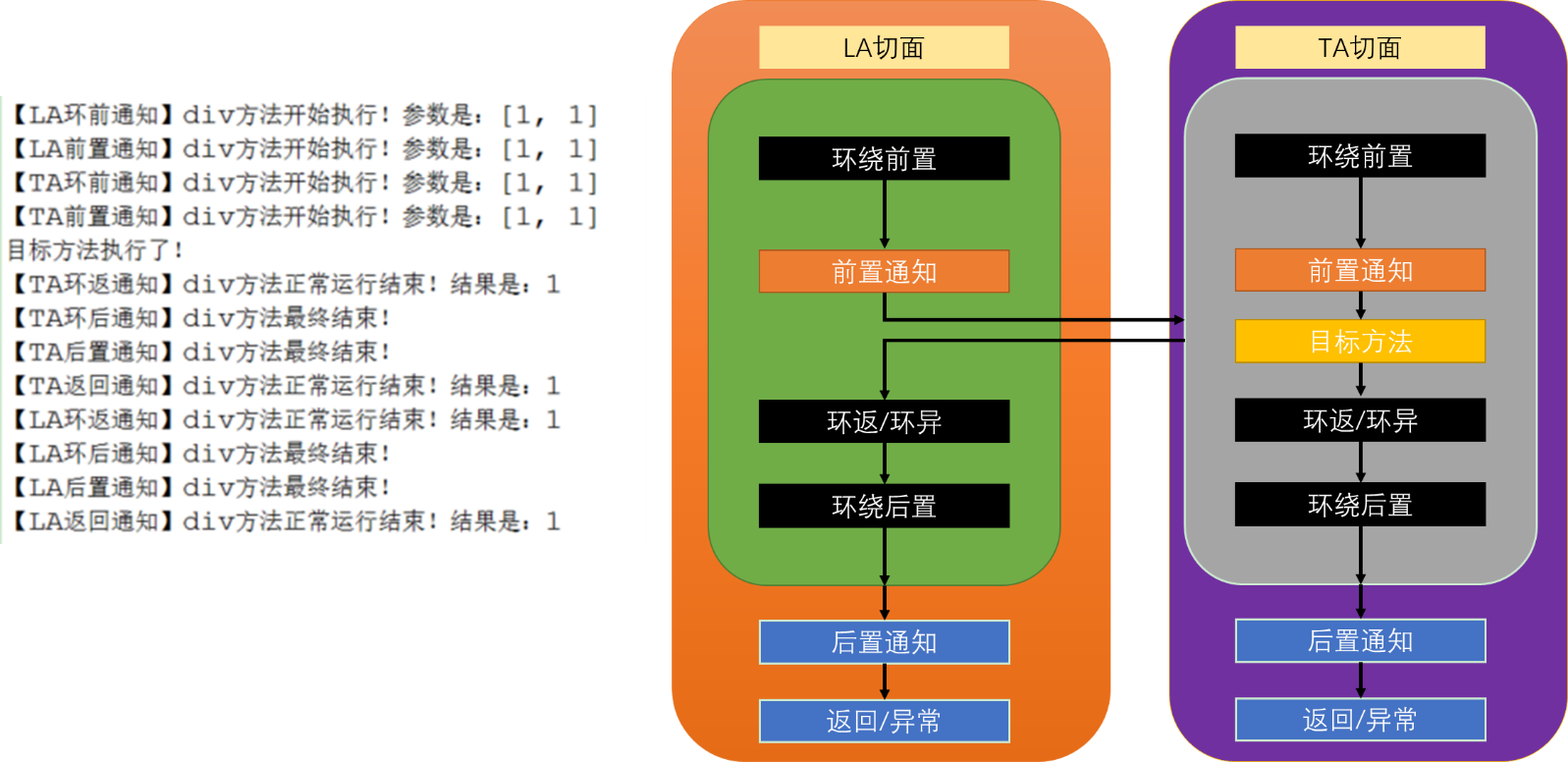
②如果只是一些简单的功能，使用普通的四种通知即可！

## 6. 通知的执行顺序

### 6.1 单个切面通知的执行顺序



### 6.2 多个切面通知的执行顺序



### 6.3 总结

前置优先，环绕先行，先进后出。

### 6.4 切面的优先级

默认是按照切面类首字母在英文字母表的顺序来排名，靠前的优先级高！

使用@Order(int x)可以修改切面的优先级，其中x越小，优先级越高！

## 7. 使用XML配置AOP

<!-- 创建目标对象 -->

<bean class=*"com.atguigu.spring.impl.EasyCalculator"* id=*"ec"*/>

<!-- 创建切面 -->

<bean class=*"com.atguigu.spring.aspects.LoggerAspect"* id=*"la"*/>

<bean class=*"com.atguigu.spring.aspects.TransactionAspect"* id=*"ta"*/>

<!-- 配置切面，将切面和目标对象关联起来 -->

<aop:config>

<!-- 定义一个切入点表达式 -->

<aop:pointcut expression=*"execution(\* com.atguigu.spring.impl.EasyCalculator.\*(..))"* id=*"myPointCut"*/>

<!-- 配置LA切面 -->

<aop:aspect ref=*"la"* order=*"1"*>

<aop:around method=*"aroundAdvice"* pointcut-ref=*"myPointCut"*/>

<aop:before method=*"beforeAdvice"* pointcut-ref=*"myPointCut"*/>

<aop:after method=*"afterAdvice"* pointcut-ref=*"myPointCut"*/>

<aop:after-returning method=*"returnAdvice"* pointcut-ref=*"myPointCut"* returning=*"result"*/>

<aop:after-throwing method=*"exceptionAdvice"* pointcut-ref=*"myPointCut"* throwing=*"e"*/>

</aop:aspect>

<aop:aspect ref=*"ta"* order=*"2"*>

<aop:around method=*"aroundAdvice"* pointcut-ref=*"myPointCut"*/>

<aop:before method=*"beforeAdvice"* pointcut-ref=*"myPointCut"*/>

<aop:after method=*"afterAdvice"* pointcut-ref=*"myPointCut"*/>

<aop:after-returning method=*"returnAdvice"* pointcut-ref=*"myPointCut"* returning=*"result"*/>

<aop:after-throwing method=*"exceptionAdvice"* pointcut-ref=*"myPointCut"* throwing=*"e"*/>

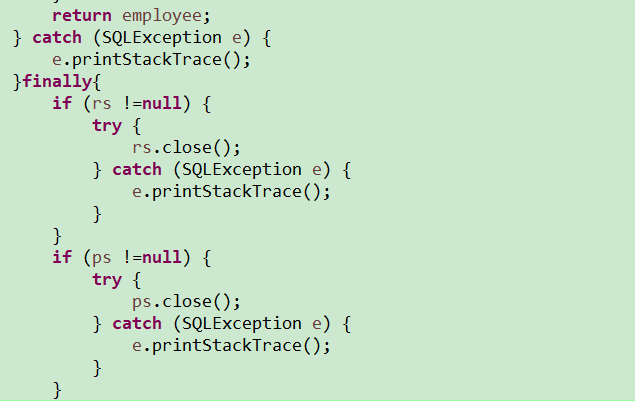
</aop:aspect>

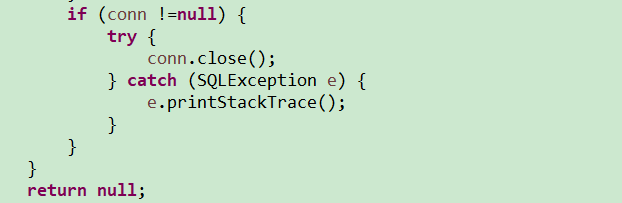
</aop:config>

# 四、JdbcTemplate

## 1. 概述





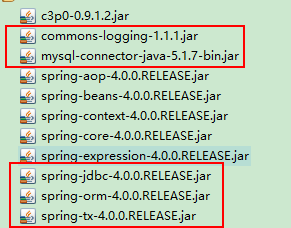


缺点：核心业务，也就是SQL淹没在一堆JDBC的样板式代码中。而使用Spring可以通过模板封装来消除样板代码，让开发人员更加专注于自己的核心逻辑，也就是SQL。

为了使JDBC更加易于使用，Spring在JDBC API上定义了一个抽象层，以此建立一个JDBC存取框架。JdbcTemplate将常用的数据库操作封装为模版方法，开发人员只需要专注于SQL的编写即可。通过这种方式，可以在尽可能保留灵活性的情况下，将数据库存取的工作量降到最低。

## 2. 环境准备

①导入数据库驱动、以及Spring相关jar包



②数据库连接配置文件

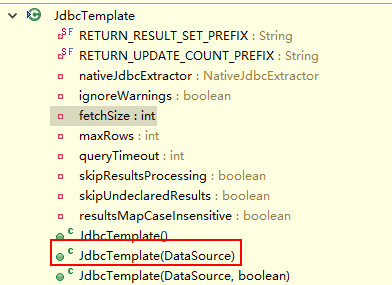
user=root

password=root

jdbcUrl=jdbc:mysql:///query\_data

driverClass=com.mysql.jdbc.Driver

③在容器中配置JdbcTemplate



<context:component-scan base-package=*"com.atguigu.spring"*/>

<!-- 从外部文件引入数据库配置信息 -->

<context:property-placeholder location=*"classpath:db.properties"*/>

<!-- 准备C3P0数据源 -->

<bean class=*"com.mchange.v2.c3p0.ComboPooledDataSource"* id=*"dataSource"*>

<property name=*"user"* value=*"${user}"*/>

<property name=*"password"* value=*"${password}"*/>

<property name=*"jdbcUrl"* value=*"${jdbcUrl}"*/>

<property name=*"driverClass"* value=*"${driverClass}"*/>

</bean>

<!-- 创建一个JdbcTemplate，它需要一个数据源 -->

<bean class=*"org.springframework.jdbc.core.JdbcTemplate"* id=*"jt"*>

<constructor-arg name=*"dataSource"* ref=*"dataSource"*/>

</bean>

<!-- 创建一个带具名参数的JdbcTempate -->

<bean class=*"org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate"* id=*"npjt"*>

<constructor-arg name=*"dataSource"* ref=*"dataSource"*/>

</bean>

④建库建表

CREATE DATABASE `jdbc\_template`;

USE `jdbc\_template`;

DROP TABLE IF EXISTS `employee`;

CREATE TABLE `employee` (

`emp\_id` INT(11) NOT NULL AUTO\_INCREMENT,

`emp\_name` CHAR(100) DEFAULT NULL,

`salary` DOUBLE DEFAULT NULL,

PRIMARY KEY (`emp\_id`)

) ENGINE=INNODB AUTO\_INCREMENT=7 DEFAULT CHARSET=gb2312;

INSERT INTO `employee`(`emp\_id`,`emp\_name`,`salary`) VALUES

(1,'Susan',5000.23),(2,'Julian',4234.77),(3,'Papu',9034.51),(4,'Babala',8054.33),(5,'Kasier',6039.11),(6,'Owen',7714.11);

## 3. 常用方法

### 3.1 查询单条记录



@Test

**public** **void** testQueryForObject() {

String sql="select emp\_id id,emp\_name name,salary,date from employee where emp\_id =?";

//BeanPropertyRowMapper: 定义如何按照一定的规则将数据库的字段一一封装为bean的属性

RowMapper<Employee> rowMapper=**new** BeanPropertyRowMapper<Employee>(Employee.**class**);

Employee employee = jt.queryForObject(sql, rowMapper, 1);

System.***out***.println(employee);}

### 3.2 查询多条记录



@Test

**public** **void** testQueryForList() {

String sql="select emp\_id id,emp\_name name,salary from employee where emp\_id >?";

//BeanPropertyRowMapper: 定义如何按照一定的规则将数据库的字段一一封装为bean的属性

RowMapper<Employee> rowMapper=**new** BeanPropertyRowMapper<Employee>(Employee.**class**);

List<Employee> result = jt.query(sql, rowMapper, 2);

System.***out***.println(result);

}

### 3.3 查询单一值



@Test

**public** **void** testQueryForSingleValue() {

//单一值： 如记录数、最大值、最小值

String sql="select MAX(salary) from employee";

Double maxSalary = jt.queryForObject(sql, Double.**class**);

System.***out***.println("最大薪水："+maxSalary);

}

### 3.4 增删改

  
@Test

**public** **void** testUpdate() {

String sql="update employee set salary=? where emp\_id=?";

jt.update(sql, 6666.66,1);

}

### 3.5 批量增删改



@Test

**public** **void** testBatchUpdate() {

String sql="insert into employee(emp\_name,salary) value(?,?)";

//封装多条记录的参数进来

List<Object[]> batchArgs=**new** ArrayList<Object[]>();

batchArgs.add(**new** Object[]{"Tom",2222.22});

batchArgs.add(**new** Object[]{"Jerry",2222.23});

jt.batchUpdate(sql, batchArgs);

}

## 4. 在Dao中注入JdbcTemplate

@Repository

**public** **class** EmployeeDao {

@Autowired

**private** JdbcTemplate jt;

**public** **void** getDao(){

System.***out***.println(jt);

}

}

# 五、声明式事务

## 1. 引入

### 1.1 事务

●在JavaEE企业级开发的应用领域，为了保证数据的**完整性**和**一致性**，必须引入数据库事务的概念，所以事务管理是企业级应用程序开发中必不可少的技术。

●事务就是一组由于逻辑上紧密关联而合并成一个整体(工作单元)的多个数据库操作，这些操作**要么都执行**，**要么都不执行**。

●事务的四个关键属性(ACID)

○**原子性**(atomicity)：“原子”的本意是“**不可再分**”，事务的原子性表现为一个事务中涉及到的多个操作在逻辑上缺一不可。事务的原子性要求事务中的所有操作要么都执行，要么都不执行。

○**一致性**(consistency)：“一致”指的是数据的一致，具体是指：所有数据都处于**满足业务规则的一致性状态**。一致性原则要求：一个事务中不管涉及到多少个操作，都必须保证事务执行之前数据是正确的，事务执行之后数据仍然是正确的。如果一个事务在执行的过程中，其中某一个或某几个操作失败了，则必须将其他所有操作撤销，将数据恢复到事务执行之前的状态，这就是**回滚**。

○**隔离性**(isolation)：在应用程序实际运行过程中，事务往往是并发执行的，所以很有可能有许多事务同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。隔离性原则要求多个事务在**并发执行过程中不会互相干扰**。

○**持久性**(durability)：持久性原则要求事务执行完成后，对数据的修改**永久的保存**下来，不会因各种系统错误或其他意外情况而受到影响。通常情况下，事务对数据的修改应该被写入到持久化存储器中。

### 1.2 编程式事务

[1]获取数据库连接Connection对象

Connection connection=JDBCUtils.getConnection();

[2]取消事务的自动提交

connection.setAutoCommit(false);

[3]执行操作

try{

doSomething();

[4]正常完成操作时手动提交事务

connection.commit();

}catch(Exception e){

[5]执行失败时回滚事务

connection.rollback();

}finally{

[6]关闭相关资源

connection.close()

}

缺点：

编程式事务管理需要将事务管理代码**嵌入到业务方法中**来控制事务的提交和回滚。

相对于**核心业务**而言，事务管理的代码显然属于**非核心业务**，在使用编程的方式管理事务时，必须在每个事务操作中包含额外的事务管理代码。

如果多个模块都使用同样模式的代码进行事务管理，显然会造成较大程度的**代码冗余**。

解决：

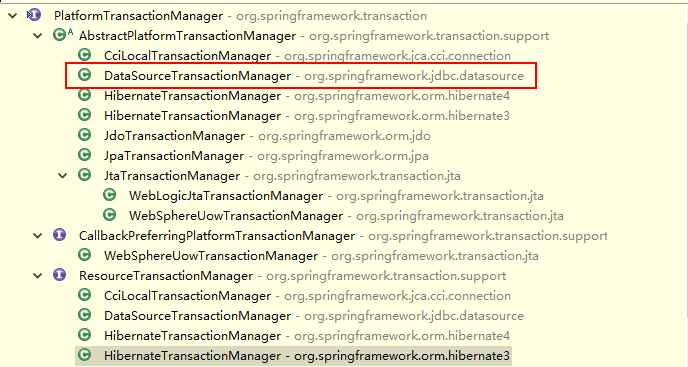
事务管理代码的固定模式作为一种横切关注点，可以通过AOP方法模块化，进而借助Spring AOP框架实现声明式事务管理。

Spring在不同的事务管理API之上定义了一个抽象层，通过配置的方式使其生效，从而让应用程序开发人员不必了解事务管理API的底层实现细节，就可以使用Spring的事务管理机制。

Spring既支持编程式事务管理，也支持声明式的事务管理。

### 1.3.Spring提供的事务管理器

Spring的核心事务管理抽象是PlatformTransactionManager。它为事务管理封装了一组独立于技术的方法。而实际开发中，需要根据Dao层的具体实现来选择相应的实现。



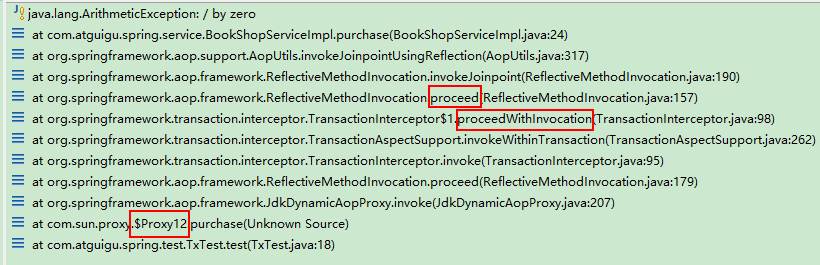
### 1.4 声明式事务的原理

声明式事务的原理是AOP，而AOP的原理是动态代理。当在制定的业务方法上面加上@Transactional注解后，再调用业务方法时，实际上是调用容器中此方法的代理对象来执行。

没有标上@Transactional注解时，报错的信息：



使用声明式事务后：



### 1.5 实验的准备工作

①建库建表

CREATE TABLE **book** (

isbn VARCHAR (50) PRIMARY KEY,

book\_name VARCHAR (100),

price INT

) ;

CREATE TABLE **book\_stock** (

isbn VARCHAR (50) PRIMARY KEY,

stock INT,

CHECK (stock > 0)

) ;

CREATE TABLE **account** (

username VARCHAR (50) PRIMARY KEY,

balance INT,

CHECK (balance > 0)

) ;

INSERT INTO account (`username`,`balance`) VALUES ('Tom',100000);

INSERT INTO account (`username`,`balance`) VALUES ('Jerry',150000);

INSERT INTO book (`isbn`,`book\_name`,`price`) VALUES ('ISBN-001','book01',100);

INSERT INTO book (`isbn`,`book\_name`,`price`) VALUES ('ISBN-002','book02',200);

INSERT INTO book (`isbn`,`book\_name`,`price`) VALUES ('ISBN-003','book03',300);

INSERT INTO book (`isbn`,`book\_name`,`price`) VALUES ('ISBN-004','book04',400);

INSERT INTO book (`isbn`,`book\_name`,`price`) VALUES ('ISBN-005','book05',500);

INSERT INTO book\_stock (`isbn`,`stock`) VALUES ('ISBN-001',1000);

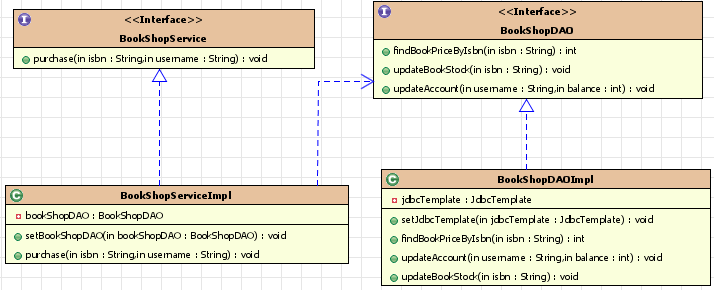
INSERT INTO book\_stock (`isbn`,`stock`) VALUES ('ISBN-002',2000);

INSERT INTO book\_stock (`isbn`,`stock`) VALUES ('ISBN-003',3000);

INSERT INTO book\_stock (`isbn`,`stock`) VALUES ('ISBN-004',4000);

INSERT INTO book\_stock (`isbn`,`stock`) VALUES ('ISBN-005',5000);

②业务逻辑



③service与dao

public interface BookShopService {

//根据isbn号与用户名进行买书操作

public void purchase(String isbn,String username);

//对书籍进行调价行为

public void managerBook(String isbn,int price);

}

public interface BookShopDao {

//根据isbn号查询对应书籍的价格

public int findBookPriceByIsbn(String isbn);

//根据isbn号更新指定书籍的库存

public void updateBookStock(String isbn);

//更新指定用户的账户金额

public void updateAccount(String username,int balance);

//更新指定书籍的价格

public void updateBookPrice(String isbn,int price);

}

## 2. 实现声明式事务

步骤：

[1]在IOC容器中配置事务管理器的bean

[2]开启基于注解的事务管理功能，并指定当前要使用的事务管理器对象

[3]在要进行事务管理的方法上加@Transactional

①导包



②配置

<context:component-scan base-package=*"com.atguigu.spring"*/>

<!-- 从外部文件引入数据库配置信息 -->

<context:property-placeholder location=*"classpath:db.properties"*/>

<!-- 准备C3P0数据源 -->

<bean class=*"com.mchange.v2.c3p0.ComboPooledDataSource"* id=*"dataSource"*>

<property name=*"user"* value=*"${user}"*/>

<property name=*"password"* value=*"${password}"*/>

<property name=*"jdbcUrl"* value=*"${jdbcUrl}"*/>

<property name=*"driverClass"* value=*"${driverClass}"*/>

</bean>

<!-- 创建一个JdbcTemplate，它需要一个数据源 -->

<bean class=*"org.springframework.jdbc.core.JdbcTemplate"* id=*"jt"*>

<constructor-arg name=*"dataSource"* ref=*"dataSource"*/>

</bean>

<!-- 根据自己的Dao层使用的组件，配置相应的事务管理器 -->

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

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

</bean>

<!-- 开启基于注解的声明式事务 -->

<tx:annotation-driven transaction-manager=*"dt"*/>

③在要使用事务的方法上面，加上@Transactional注解

## 3. @Transactional的常用属性

## 3.1 异常处理

运行时异常（非检查异常）： 直接回滚

noRollbackFor：指定不回滚异常的全类名，多个使用，隔开

noRollbackForClassName:指定不回滚异常的类名

编译时异常（检查异常）： 默认不回滚

rollbackForClassName： 指定哪些异常回滚，传入异常的类名

rollbackFor：指定哪些异常回滚，传入异常类型

@Transactional(noRollbackForClassName={"java.lang.ArithmeticException"},rollbackFor={FileNotFoundException.**class**})

**public** **void** purchase(String isbn, String username) **throws** FileNotFoundException {

## 3.2 readonly

通常在查询方法中，声明readOnly=true，声明一个方法为只读方法，此方法只能从数据库查询数据，而不能有增删改的行为，否则会报错！

作用：Spring对声明了readOnly=true的方法，会做底层优化，加快查询效率！

## 3.3 timeout

int timeout() default -1： 事务的超时属性，默认是-1，代表永不超时。单位是秒. 如果指定时间内，方法没有执行完，则回滚事务！

## 3.4 isolation



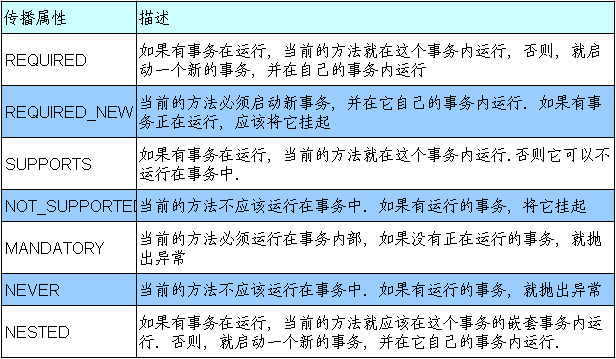
Isolation isolation() default Isolation.DEFAULT;事务的隔离级别，代表多事务并发操作时引起的问题！

隔离级别越高，数据的安全性，一致性越好，但是效率越低。

DEFAULT:采取数据库默认的隔离级别，如果是Mysql就代表可重复读。

## 3.5 propagation

propagation(传播行为): 当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。事务的传播行为可以由传播属性指定。Spring定义了7种类传播行为。



常用的有Required、和Required\_New。默认情况下是Required: 如果当前方法有事务运行，就在当前方法的事务中运行。否则开启自己的新事务！Required\_New: 不管当前方法有没有事务，都开启自己的新事务！

## 4. 基于XML的事务配置

<context:component-scan base-package=*"com.atguigu.spring"*/>

<!-- 从外部文件引入数据库配置信息 -->

<context:property-placeholder location=*"classpath:db.properties"*/>

<!-- 准备C3P0数据源 -->

<bean class=*"com.mchange.v2.c3p0.ComboPooledDataSource"* id=*"dataSource"*>

<property name=*"user"* value=*"${user}"*/>

<property name=*"password"* value=*"${password}"*/>

<property name=*"jdbcUrl"* value=*"${jdbcUrl}"*/>

<property name=*"driverClass"* value=*"${driverClass}"*/>

</bean>

<!-- 创建一个JdbcTemplate，它需要一个数据源 -->

<bean class=*"org.springframework.jdbc.core.JdbcTemplate"* id=*"jt"*>

<constructor-arg name=*"dataSource"* ref=*"dataSource"*/>

</bean>

<!-- 根据自己的Dao层使用的组件，配置相应的事务管理器 -->

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

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

</bean>

<!-- 开启基于注解的声明式事务 -->

<!-- <tx:annotation-driven transaction-manager="dt"/> -->

<!-- 基于XML开启声明式事务的配置 -->

<!-- 声明一个tx:advice(事务通知) -->

<tx:advice id=*"myTx"* transaction-manager=*"dt"*>

<!-- 配置事务的属性 -->

<tx:attributes>

<!-- 配置哪些方法需要声明式事务 -->

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

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

<tx:method name=*"purchase"* timeout=*"3"*/>

<!-- \* 相当于只要被切入点表达式切到的方法，都加上@Transactional -->

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

</tx:attributes>

</tx:advice>

<!-- 配置BookShopServiceImplPlus方法的事务属性 -->

<tx:advice id=*"myTx2"* transaction-manager=*"dt"*>

<!-- 配置事务的属性 -->

<tx:attributes>

<!-- 配置哪些方法需要声明式事务 -->

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

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

<tx:method name=*"purchase"* timeout=*"6"*/>

<!-- \* 相当于只要被切入点表达式切到的方法，都加上@Transactional -->

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

</tx:attributes>

</tx:advice>

<aop:config>

<!-- 通过切入点表达式，将事务管理器切入到目标方法 -->

<aop:pointcut expression=*"execution(\* com.atguigu.spring.service.BookShopServiceImpl.\*(..))"* id=*"myPointCut"*/>

<!-- 定义另外一组事务策略 -->

<aop:pointcut expression=*"execution(\* com.atguigu.spring.service.BookShopServiceImplPlus.\*(..))"* id=*"myPointCut2"*/>

<aop:advisor advice-ref=*"myTx"* pointcut-ref=*"myPointCut"*/>

<aop:advisor advice-ref=*"myTx2"* pointcut-ref=*"myPointCut2"*/>

</aop:config>

# 六、Web中使用Spring

## 1.在web中使用Spring的思路

Spring是个容器，用来创建和管理对象（为对象的属性赋值）

Web下使用Spring：

### 1.1获取容器

①获取容器。容器对象要保证，一个web应用中，只有一个容器对象，且随着web应用的加载而创建

②将容器放入到ServletContext中，也就是最大的域对象中，保证随时可以获取。

上述步骤，可以通过Listenner实现！Spring已经提供好了这个监听器（contextLoadListenner）。

### 1.2让容器创建web应用中的资源

创建web应用时，按照MVC三层结构创建：

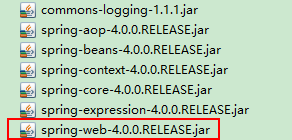
Servlet------Service------Dao

Servlet是请求第一次访问的时候，由Tomcat（servlet 容器）创建！

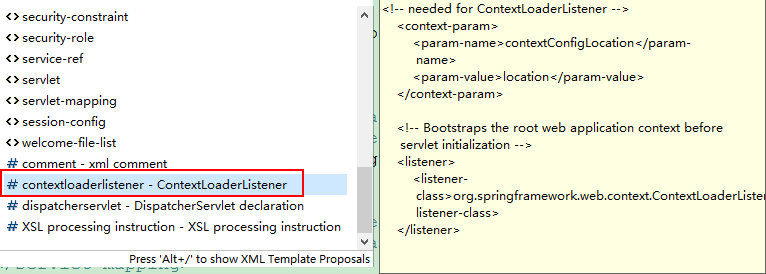
注意：service,dao层我们可以使用Spring的DI，但是Servlet中的属性，要手动赋值！

## 2.使用Spring的ContextLoaderListener

①导包



②在web.xml中配置：



<context-param>

<!-- spring配置文件的位置 -->

<param-name>contextConfigLocation</param-name>

<param-value>classpath:beans.xml</param-value>

</context-param>

<listener>

<!-- spring提供的监听器类 -->

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

③编写配置文件



④创建组件

Service,Dao使用DI创建

Servlet，必须手动装配属性！

步骤：①拿到ServletContext域对象;

②Spring提供了工具方法，从当前域对象，获取容器：

③调用getBean()，装配对象

注意：不能在构造器中，操作，报错：空指针异常！重写init()方法实现！

@Override

**public** **void** init() **throws** ServletException {

//拿到ServletContext对象

ServletContext sc = getServletContext();

//拿到容器

WebApplicationContext context = WebApplicationContextUtils.*getWebApplicationContext*(sc);

//从容器中取出对象手动赋值

bs=context.getBean(BookService.**class**);}