[IOC（控制反转） 1](#_Toc20009)

[下载spring 2](#_Toc7778)

[依赖注入 3](#_Toc377)

[<bean>标签 6](#_Toc9817)

[Spring AOP 7](#_Toc12538)

[Spring对DAO的支持 10](#_Toc14919)

[Spring对事务的支持 13](#_Toc9220)

[注解 16](#_Toc4687)

[@PostConstruct 16](#_Toc20373)

[@PreDestory 17](#_Toc7997)

[@Lazy 17](#_Toc31239)

[ThreadPoolTaskExecutor 17](#_Toc11223)

// Spring到底是什么玩意啊！AOP，IOC，我从来没有这样讨厌过英文缩写。果然寒假应该把spring全部看完的，我真是个大笨蛋啊！

事实证明，毛都不懂的情况下还硬刚框架会死的很惨！

# IOC（控制反转）

// 有时候也不要把东西想的太复杂，这些概念什么的又不是现在的我该操心的，只要知道它是把控制权转交给容器就好。

// 幼稚的过往。↓

控制权转交，通过一个接口来实现，感觉就是多态啊。一个视频看完之后真是神清气爽，知识点在握的感觉真好。总结起来就是给空父类引用赋值后调用具体的子类实现，这整个过程也叫做依赖注入。但是在Spring这，不通过代码，而是xml实现。

是什么、为什么、怎么办；时间、地点、人物、起因、经过、结果。该告诉我的早就告诉我了，关键在于我几时醒悟过？

IOC是什么？一种设计模式。为什么要IOC？为了解耦。怎么办？创建一个接口，然后在某类内创建一个该接口的引用，并通过某种方式（setters）为该引用赋子类具体实现。上面那个又被叫成依赖注入，也是用的最多的，然而百度百科告诉我，还有依赖查找。

总之，依赖注入大概就这种感觉。

public interface Tester{

void test();

}

public class XXX implements Tester{

// TODO ... ...

}

public class WorkSpace{

private Tester tester;

public void setTester(Tester tester){

this.tester = tester;

}

}

# 下载spring

编程这东西最重要的还是动手，带着这种念头，我第一时间冲往spring官网下载jar包。但，像国足对阵世界杯一样，我竟找不到入口在哪。。。

spring.io -> projects（官网的字体丑到爆）-> spring framework ->

进入github -> 找到downloading spring artifacts ->

最底下有[http://repo.spring.io](http://repo.spring.io/) -> 左侧artifacts选择lib-release-local ->

org -> springframewok -> spring -> 少年快去选你想要的版本啊！

另外这是下全部的方法，里头有19个jar包：

spring-aop spring-context-support

spring-instrument-tomcat spring-oxm

spring-web spring-aspects

spring-core spring-jdbc

spring-struts spring-webmvc

spring-beans spring-expression

spring-jms spring-test

spring-webmvc-portlet spring-context

spring-instrument spring-orm

spring-tx spring-websocket

// 但官网上下的压缩包里却解压出20个（jar，doc，src），因为其中一个模块由两个jar包组成，至于是哪个模块。。。。

但核心jar包，据小峰测试（我就懒得弄了），不算commons的logging还有collection的话，为以下5个：

spring-aop spring-context spring-core spring-expression spring-beans

然后呢，项目怎么写？其实原来怎么写，现在还是怎么写，不过是获取实例的那部分操作被spring承包了。com.jojo.model.Student，com.jojo.test.StudentTest，application-context.xml，写好上述三个文件，最简单的spring demo就完成了。

// 起关键作用的代码就以下几行

<bean id=*"Student"* class=*"com.jojo.model.Student"*>

ApplicationContext ac =

**new** ClassPathXmlApplicationContext("application-context.xml");

Student student = (Student)ac.getBean("Student");

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

# 依赖注入

* 属性注入

<bean id=*"student"* class=*"com.jojo.model.Student”*>

<property name=*"id"* value=*"1"*></property>

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

</bean>

这样获取到的对象会带有一些默认参数，但要求是<property>中的name必须与类内的变量相对应。<property>的原理是set方法，所以要想起效，得保证要注入的那几个成员变量都有set方法。

* 构造方法注入

<bean id=*"Student"* class=*"com.jojo.model.Student"*>

<constructor-arg type=*"Integer"* value=*"23"*></constructor-arg>

<constructor-arg type=*"String"* value=*"sdf"* ></constructor-arg>

</bean>

我们的口号是“严格对应”!

要有一个严格与xml对应的构造函数，不然会报错。另外上面那个写法不是很好，当有两个String类型时要怎么办？所以还要加上index属性（从0开始）。<constructor-arg>还有一些子标签<list>，<set>，<map>，毕竟有时候会用集合做参数。

* 工厂方法注入

没碰上需求，所以也不是很能理解

1. 静态工厂方法获取实例
2. 非静态

* 来注入一些奇奇怪怪的参数吧

上面只是给bean注入基本类型的变量，现在要注入一个另一个类的对象。

<property name=*"c"* ref=*"A"*></property>

用 ref 属性就好，ref 后跟的值必须也是一个<bean>标签，可以理解为通过一个外部<bean>标签注入。我这么说的意思很明显，当然还可以从内部<bean>标签注入啦~

// 完整实例

<bean id=*"Student"* class=*"com.jojo.model.Student"*>

<property name=*"address"*>

<bean class=*"com.jojo.model.Address"*>

<property name=*"provience"* value=*"江苏"*></property>

</bean>

</property>

</bean>

除此之外有注入内部bean，null值，级联属性，集合类型，有空看。

* 注null

<bean id=*"Student"* class=*"com.jojo.model.Student"*>

<property name=*"address"*>

<null></null>

</property>

</bean>

都说是null了除了类对象，还有谁能用得到？

* 级联

<bean id=*"Student"* class=*"com.jojo.model.Student"*>

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

<property name=*"address.provience"* value=*"江苏"*></property>

</bean>

看着爽，用起来烦，限制太多了。前面讲过了，使用spring时要严格对应，级联这种写法里并没有出现<bean>（要注入的那个），直接使用是会报错的。因此为了正常运行还得加一行：（这种写法，话说我有多久没有看过单例模式了？）

**private** Address address = **new** Address();

* 集合

和级联一样，要正常运行也得先new一个（不一定也有可能啊）。然后<list>，<set>，<map>少年啊，勇敢的 alt+/ 吧。

既然都是键值对，spring还支持**Properties**类型。<props>，神圣的 alt + / 守护者我们。

* 自动注入

<beans>的一个属性default-autowire，有这么几个选项：

byName 如果某<bean>的id与该类成员变量名相同，则自动注入

byType 如果某<bean>的类型（就是class属性的值）与该类成员变量的类型相 同，则自动注入，那么问题来了，存在多个类型相同的<bean>要怎么自 动注入。

constructor 和byType类似，也是通过类型，但注入方式是构造函数（要写构造函数）

自动装配慎用，它屏蔽了装配的细节，容易产生潜在的错误。

* 方法注入

靠<bean>获取对象，默认是单例模式，因此每次获取到的对象都是一模一样的。配置Protype后每次获取到的对象就是新生成的。（这个东西下面还会再讲的）

<bean id=*"student"* class=*"com.jojo.model.Student"* scope=*"prototype"*></bean>

**Spring默认的为所有的bean设置为单例模式，只有当我们特别声明bean的模式时，Spring才会修改对应的bean设置，即使存在嵌套关系bean中，外层声明为prototype，其内部在未声明的情况下，仍然是单例的。**

**给一个singleton的<bean>注入一个prototype的<bean>。怎么搞呢，单单把内部<bean>的scope设置成prototype是没用的，最后获取到的外部<bean>对象从头到脚还是一模一样。**

**这时候就体现出方法注入的好来了，但要改的地方还是比较多的。首先是Model类Student，要将获取Address的getter改成抽象方法，然后改<bean>**

<bean id=*"Student"* class=*"com.jojo.model.Student"*>

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

<lookup-method name=*"getAddress"* bean=*"Address"*/>

</bean>

**同时，***com.jojo.model.Student* **中的getAddress() 方法必须变成 abstract 类型。**

**注意标签的变化<lookup-method>而且属性也变了，是name与bean，name里写方法的名字。用到的机会真是渺茫啊。**

* 方法替换

// 这东西真的用得到吗？

方法替换，就是替换成别的类的方法。

要动的地方也不少，准备一个类，该类要实现MethodReplacer接口，算了我直接贴代码吧。

**import** java.lang.reflect.Method;

**import** org.springframework.beans.factory.support.MethodReplacer;

**public** **class** Student2 **implements** MethodReplacer{

@Override

**public** Object reimplement(Object arg0, Method arg1, Object[] arg2) **throws** Throwable {

// 这里头是要写代码的。

**return** **null**;

}

}

貌似只留这么一个方法就够了，然后是xml

<bean id=*"Student"* class=*"com.jojo.model.Student"*>

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

<replace-method name=*"getAddress"* replacer=*"Student2"*/>

</bean>

<bean id=*"Student2"* class=*"com.jojo.model.Student2"*></bean>

讲道理真的用得到吗？

# <bean>标签

* <bean>之间的关系

~~就目前看来，bean与Java对象对应，spring通过bean来管理对象，程序员通过bean来获取对象，好处在哪我不知道，但既然存在这么一个对应关系，那么对象间的继承、依赖、引用应该也在。~~

然而这种理解方式并不对，<bean>标签之间就是有关系

1. 继承

<bean id=*"Address"* class=*"com.jojo.model.Address"* scope=*"prototype"*>

<property name=*"provience"* value=*"江苏省"*></property>

<property name=*"city"* value=*"苏州"*></property>

</bean>

<bean id=*"abstractStudent"* class=*"com.jojo.model.Student"* abstract=*"true"*>

<property name=*"address"* ref=*"Address"*></property>

</bean>

<bean id=*"student"* parent=*"abstractStudent"*>

这样再获取“student”这个<bean>时，对象的address自动有值。

**Spring允许继承bean的配置，被继承的bean称为父bean，继承这个父bean的bean称为子bean**

**子bean从父bean中继承配置，包括bean的属性配置**

**子bean也可以覆盖从父bean继承过来的配置**

**父bean可以作为配置模版，也可以作为bean实例，若只想把父bean作为模板，可以设置<bean>的abstract属性为true，这样Spring将不会实例化这个bean**

**并不是<bean>元素里的所有属性都会被继承。比如：autowire，abstract等**

**也可以忽略父bean的class属性，让子bean指定自己的类，而共享相同的属性配置，但此时abstract必须设为true**

来自文档，我觉得总结的已经很好了。

1. 依赖

Spring允许用户通过depends-on属性设定<bean>前置依赖的<bean>，前置依赖的<bean>会在本<bean>实例化之前创建好。

如果前置依赖于多个bean，则可以通过逗号，空格或分号来配置bean的名称。

depends-on，嗨，就一属性。

3. 引用

其实这个东西，上面已经讲过了，就是ref之类的属性，一个<bean>引用另一个<bean>

* <bean>的作用范围 scope属性

singleton 单例模式，默认

prototype 每次都返回新的

request 每次HTTP请求都会创建一个新的bean

session 同一个HTTP session 共享一个bean

global session 同一个全局session共享一个bean，一般用于portlet（这玩意是 用来干什么的？）

application 同一个application共享一个bean

# Spring AOP

那么，什么叫面向切面编程呢？

嗯，怎么，你以为我会给出解释？别傻了，最上面那个就是疑问句。能搞清spring aop怎么用就行了。

1. 引入

使用前是要加新jar包的，哈真的是，原本我以为把springframework的全套jar拷进去就完事了，结果spring aop还需要的2个jar包 aopalliance.jar 与 aspectjweaver.jar并不在里头。算上 spring-aop.jar 与 spring-aspects.jar ，要想成功运行一共需要4个jar包。

然后是配置文件：

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/aop*

*[http://www.springframework.org/schema/aop/spring-aop.xsd"](http://www.springframework.org/schema/aop/spring-aop.xsd\">)*[>](http://www.springframework.org/schema/aop/spring-aop.xsd\">)

/\*

\* 为啥已经用 xmlns:aop 指定了一个地址后

\* 还要再在 xsi:schemaLocation 里写一 遍？

\*/

还有就是，嗯哼，划重点。在<beans>标签内部用了xmlns这些属性后就不能在用<!DOCTYPE>了。这个<!DOCTYPE>还是不要用的好，毕竟搞了他以后<beans>一共也就4个子标签<bean>、<import>、<alias>、<description>。

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

"http://www.springframework.org/dtd/spring-beans.dtd">

2. 例子

首先写个类

**import** org.aspectj.lang.JoinPoint;

**public** **class** StudentAspect {

**public** **void** doBefore(JoinPoint jp){

System.***out***.println(jp.getClass().getName() + "插进去了");

}

}

然后弄配置文件

<bean id=*"studentAspect"* class=*"com.jojo.aspect.StudentAspect"*></bean>

<aop:config>

<aop:aspect ref=*"studentAspect"*>

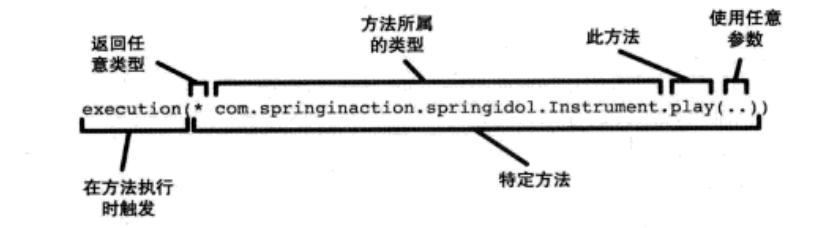
<aop:before method=*"doBefore"* pointcut=*"execution(\* com.jojo.service.\*.\*(..))"*/>

</aop:aspect>

</aop:config>

这是一个最简单的 spring aop 前置通知，根据配置service包下任意类的任意方法在执行前都会输出一个字符串。

切点表达式也是个大坑，这东西。。。唉碰到了再说吧



execution是必写的，括号也是必须的，\* 就是通配符，(..)表示方法的任意参数，别问为什么就这么规定的。我上面那种写法是因为简单，没什么东西要填，一般都是下面这种写法：

<aop:config>

<aop:aspect ref=*"studentAspect"*>

<aop:pointcut expression=*"execution(\* com.jojo.service.\*.\*(..))"* id=*"aspect"*/>

<aop:before method=*"doBefore"* pointcut-ref=*"aspect"*/>

</aop:aspect>

</aop:config>

我对起名字这事恨得深沉。。。。

1. 各种通知

有关于spring aop，我所能懂的也就是这些通知了。

* 前置通知

<aop:before method=*"doBefore"* pointcut-ref=*"aspect"*/>

method属性不一定非得写“doBefore”，准备用StudentAspect类里的哪个方法就写哪个方法。

* 后置通知

<aop:after method=*"doAfter"* pointcut-ref=*"aspect"*/>

方法结束后通知。

* 环绕通知

<aop:around method=*"doAround"* pointcut-ref=*"aspect"*/>

方法开始前通知，开始后通知。和前置通知与环绕通知是重复的，所以要么用环绕要么用前置与后置。环绕通知的代码有点不同：

**public** Object doAround(ProceedingJoinPoint pjp) **throws** Throwable{

Object result = **null**;

System.***out***.println("类：" + pjp.getTarget().getClass().getSimpleName());

System.***out***.println("方法：" + pjp.getSignature().getName());

System.***out***.println("开始了");

result = pjp.proceed();

System.***out***.println("返回值：" + result);

System.***out***.println("结束了");

**return** result;

}

// 透过反射，这个类在你面前等于没穿衣服。

你看这玩意是有返回值的，因为它会执行方法，方法的运行是在 pjp.proceed(); 之后，就像 Struts 一样。

另外这玩意的优先级，怎么说呢，前置，环绕，后置一起写的话，spring 会先执行前置再执行环绕最后执行后置。

* 返回通知

<aop:after-returning method=*"doAfterReturn"* pointcut-ref=*"aspect"*/>

方法快返回前通知一下，没什么特别的。

* 异常通知

<aop:after-throwing method=*"doAfterThrowing"* throwing=*"ex"* pointcut-ref=*"aspect"*/>

有异常时通知一下，xml不一样代码也不一样，突出一个异常信息。throwing属性的值要与方法的形参相同，不然报错，这里就是“ex”。

**public** **void** doAfterThrowing(JoinPoint jp, Throwable ex){

System.***out***.println("哇呀呀呀，来了");

System.***out***.println("他就是" + ex.getMessage());

}

# Spring对DAO的支持

// 思来想去，怎么都觉得这东西我用不到

* Spring JDBC

给我文档我就能飞？

Spring的尿性是管理<bean>，因此程序员可以放心大胆的去思考需求，也就是写业务代码。而其他的比如说

daoFactory.openConnection();

daoFactory.beginTransaction();

daoFactory.createXXXDao();

...

daoFactory.endTransaction();

daoFactory.closeConnection();

这种不得不写，但麻烦的要死，且没技术成就感的代码就不用去写了。其实我觉得大部分业务代码也很无趣，相当无趣。

Spring解放程序员双手的关键是 application-context.xml （jar包就不谈了），配置完毕，然后用注解或是XXXUtil去享受高潮。

下面，凭着文档，与一丢丢印象，用spring JDBC来操作个数据库看看。

第一步，贴jar包。 // 暴力一点，全贴了

第二步，搞配置文件。

从官方文档那贴过来的，当然最终肯定不止这么点，先写着。

<?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:aop=*"http://www.springframework.org/schema/aop"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd"*>

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp2.BasicDataSource"* destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClassName}"*/>

<property name=*"url"* value=*"${jdbc.url}"*/>

<property name=*"username"* value=*"${jdbc.username}"*/>

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

</bean>

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

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

</bean>

</beans>

这几行是为了找到存放参数的 .properties ，不过真正写的时候顺序还是要反过来。

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

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*

完整的xml

<?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:aop=*"http://www.springframework.org/schema/aop"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

<!-- 配置DataSource -->

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

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp2.BasicDataSource"* destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClassName}"*/>

<property name=*"url"* value=*"${jdbc.url}"*/>

<property name=*"username"* value=*"${jdbc.username}"*/>

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

</bean>

<!-- 用来连接、操作数据库的类 -->

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

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

</bean>

<!-- 要用的DAO -->

<bean id=*"studentDao"* class=*"com.jojo.dao.StudentDaoImpl"*>

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

</bean>

<!-- 最终被使用的Service，这吊东西为什么这么烦... -->

<bean id=*"studentService"* class=*"com.jojo.service.StudentServiceImpl"*>

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

</bean>

</beans>

最后要用的只是StudentService，中间却经历了JdbcTemplate，StudentDao。另外由于使用<property>标签的 ref 属性，所以每个类里头都要写上对应的 setXXX() 方法，不够美观。

Spring推崇面向接口编程，因此无论 DAO 还是 Service 都有一个接口，这种东西只有真正接触到大型项目才会觉得有用啊，我这个学习例子是看不出来的。

第三步，写代码。

。。。我现在觉得注解好用了，但人啊，有时候就要倔强，把 xml 死磕到底。

**public** **class** StudentDaoImpl **implements** StudentDao {

**private** JdbcTemplate jdbcTemplate;

// set注入

**public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {

**this**.jdbcTemplate = jdbcTemplate;

}

@Override

**public** **int** add(Student student) {

String sql = "insert into t\_student values(null, ?, ?)";

Object[] parameters = { student.getName(), student.getAge() };

**return** jdbcTemplate.update(sql, parameters);

}

}

测试：

**public** **static** **void** main(String[] args) {

@SuppressWarnings("resource")

ApplicationContext ac = **new** ClassPathXmlApplicationContext("applicationContext.xml");

StudentService studentService = (StudentService)ac.getBean("studentService");

Student student = **new** Student();

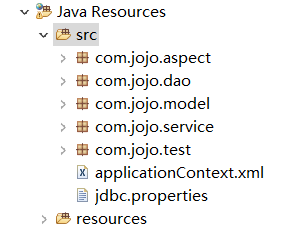
student.setAge(1);

student.setName("jojo");

System.***out***.println(studentService.add(student));

}

这里我觉得有难度的是如何读取 applicationContext.xml，上面的能正常运行是因为我把这些配置文件都放在同一文件夹下。



我要是放在resources文件夹下，我要怎么读呢？正式开发肯定是要把配置文件放在专门的文件夹下的，不然不乱套了。

其他的，就先到这吧，这块一时半会的也用不到。

# Spring对事务的支持

妈的，搞得好像我真的知道该怎么实时回滚一样，还不是调调接口，扯淡！

* 编程式事务管理

!!! 这个例子举得不恰当

首先是xml：

<?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:aop=*"http://www.springframework.org/schema/aop"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

<!-- 配置DataSource -->

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

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp2.BasicDataSource"* destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClassName}"*/>

<property name=*"url"* value=*"${jdbc.url}"*/>

<property name=*"username"* value=*"${jdbc.username}"*/>

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

</bean>

<!-- 事务管理器 --> **主要多了下面这几个东西**

<bean id=*"transactionManager"*

class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

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

</bean>

<!-- 事务管理模板 -->

<bean id=*"transactionTemplate"* class=*"org.springframework.transaction.support.TransactionTemplate"*>

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

</bean>

<!-- 用来连接、操作数据库的类 -->

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

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

</bean>

<!-- 要用的DAO -->

<bean id=*"studentDao"* class=*"com.jojo.dao.StudentDaoImpl"*>

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

</bean>

<!-- 最终被使用的Service，加了事务后又变长了 -->

<bean id=*"studentService"* class=*"com.jojo.service.StudentServiceImpl"*>

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

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

</bean>

</beans>

代码：

**public** **class** StudentServiceImpl **implements** StudentService {

**private** StudentDao studentDao;

**private** TransactionTemplate transactionTemplate;

// set注入

**public** **void** setStudentDao(StudentDao studentDao) {

**this**.studentDao = studentDao;

}

**public** **void** setTransactionTemplate(TransactionTemplate transactionTemplate) {

**this**.transactionTemplate = transactionTemplate;

}

@Override

**public** **int** add(Student student) {

**int** result = 0;

transactionTemplate.execute(**new** TransactionCallbackWithoutResult() {

@Override

**protected** **void** doInTransactionWithoutResult(TransactionStatus arg0) {

studentDao.add(student);

}

});

**return** result;

}

}

这就是为什么这个例子不好了，因为上面那种写法，withoutResult，可我这个函数明明是有返回值的。但这种内部类都搞出来的玩意，唉，编程式事务管理用的不多啊。

* 声明式事务管理

面向切面编程，把事务切进去。

老规矩，xml：

<?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:aop=*"http://www.springframework.org/schema/aop"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:tx=*"http://www.springframework.org/schema/tx"* // 这边新增了 tx

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx.xsd"*>

<!-- 配置DataSource -->

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

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp2.BasicDataSource"* destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClassName}"*/>

<property name=*"url"* value=*"${jdbc.url}"*/>

<property name=*"username"* value=*"${jdbc.username}"*/>

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

</bean>

<!-- 事务管理器 -->

<bean id=*"transactionManager"*

class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

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

</bean>

<!-- 事务通知 --> // 新增的

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

<tx:attributes>

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

</tx:attributes>

</tx:advice>

<!-- 切事务 --> // 新增的

<aop:config>

<aop:advisor advice-ref=*"txAdvice"* pointcut=*"execution(\* com.jojo.service.\*.\*(..))"*/>

</aop:config>

<!-- 用来连接、操作数据库的类 -->

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

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

</bean>

<!-- 要用的DAO -->

<bean id=*"studentDao"* class=*"com.jojo.dao.StudentDaoImpl"*>

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

</bean>

<!-- 最终被使用的Service，这吊东西为什么这么烦... -->

<bean id=*"studentService"* class=*"com.jojo.service.StudentServiceImpl"*>

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

// 这里注意，transactionTemplate没了

</bean>

</beans>

没有代码

上面那个是在 xml 里配置，还可以靠注解配置

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

@Transactional

大不大，爽不爽

* 事务的传播行为

因为每个service 方法都有事务，因此当一个service 方法调用另外一个service 方法的时，就出现了事务的嵌套

在Spring 中，通过配置Propagation，来定义事务传播行为

<tx:method name=*"\*"* propagation=*"REQUIRED"*/> 就这个

*REQUIRED* 支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。

*SUPPORTS* 支持当前事务，如果当前没有事务，就以非事务方式执行。

*MANDATORY* 支持当前事务，如果当前没有事务，就抛出异常。

*REQUIRES\_NEW* 新建事务，如果当前存在事务，把当前事务挂起。

*NOT\_SUPPORTED* 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

*NEVER* 以非事务方式执行，如果当前存在事务，则抛出异常。

*NESTED* 还没百度

总之一般就用 *REQUIRED* ，不会有人闲着没事用别的的

# 注解

## @PostConstruct

加在方法上，程序启动时便会加载。

## @PreDestory

没用过

## @Lazy

加在类上，懒加载，也就是这个类的实例只有真正用到的时候才会加载。

Spring，让模块与模块独立，互不干扰，互不影响。

# ThreadPoolTaskExecutor

**public** **class** Work {

**public** **static** **void** main(String[] args) {

ThreadPoolTaskExecutor executor = **new** ThreadPoolTaskExecutor();

executor.setMaxPoolSize(15);

executor.setCorePoolSize(5);

**for** (**int** i = 0; i < 100; i++) {

executor.execute(**new** SimpleEvent(i + "事件"));

}

}

}

**class** SimpleEvent **implements** Runnable {

**private** String name;

**public** SimpleEvent(String name) {

**super**();

**this**.name = name;

}

@Override

**public** **void** run() {

System.***out***.println(name + "被执行");

}

}

Exception in thread "main" java.lang.IllegalStateException: ThreadPoolTaskExecutor not initialized

at org.springframework.util.Assert.state(Assert.java:70)

at org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor.getThreadPoolExecutor(ThreadPoolTaskExecutor.java:260)

at org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor.execute(ThreadPoolTaskExecutor.java:291)

at com.jojo.util.Work.main(Work.java:18)

直接 new 然后用，结果报错了，不过看源码，spring的线程池好像是引用的java自身的先用java自带的吧。

ExecutorService executor = Executors.*newFixedThreadPool*(15);

**for** (**int** i = 0; i < 100; i++) {

executor.execute(**new** SimpleEvent(i + "事件"));

}

20181023，上面那段代码之所以报错，是因为线程池没有初始化。。。。。

我脑子为什么就这么轴，不肯看一眼日志，明明都标了ThreadPoolTaskExecutor not initialized

# BeanPostProcessor

完整路径：[org](eclipse-javadoc:%E2%98%82=demo/D:\\/Maven\\/repo\\/m2\\/org\\/springframework\\/spring-beans\\/4.3.10.RELEASE\\/spring-beans-4.3.10.RELEASE.jar<org).[springframework](eclipse-javadoc:%E2%98%82=demo/D:\\/Maven\\/repo\\/m2\\/org\\/springframework\\/spring-beans\\/4.3.10.RELEASE\\/spring-beans-4.3.10.RELEASE.jar<org.springframework).[beans](eclipse-javadoc:%E2%98%82=demo/D:\\/Maven\\/repo\\/m2\\/org\\/springframework\\/spring-beans\\/4.3.10.RELEASE\\/spring-beans-4.3.10.RELEASE.jar<org.springframework.beans).[factory](eclipse-javadoc:%E2%98%82=demo/D:\\/Maven\\/repo\\/m2\\/org\\/springframework\\/spring-beans\\/4.3.10.RELEASE\\/spring-beans-4.3.10.RELEASE.jar<org.springframework.beans.factory).[config](eclipse-javadoc:%E2%98%82=demo/D:\\/Maven\\/repo\\/m2\\/org\\/springframework\\/spring-beans\\/4.3.10.RELEASE\\/spring-beans-4.3.10.RELEASE.jar<org.springframework.beans.factory.config).BeanPostProcessor

是什么：实现该接口后，其它bean的初始化前及初始化后，都可以执行一些额外的操作。

为什么：<https://www.cnblogs.com/jyyzzjl/p/5417418.html>

怎么做：实现接口内的方法，然后注入到容器中就行了

// 配置文件

<bean id=*"testBeanPostProcessor"* class=*"com.jojo.spring.processor.TestBeanPostProcessor"*></bean>

<!-- 随便写的bean，用来测试这个Processor -->

<bean id=*"errr"* class=*"java.lang.String"*></bean>

// 相当简单的processor

**public** **class** TestBeanPostProcessor **implements** BeanPostProcessor {

@Override

**public** Object postProcessBeforeInitialization(Object bean, String beanName) **throws** BeansException {

System.***out***.println(bean.getClass().getName() + "初始化开始");

System.***out***.println("其名为：" + beanName);

**return** bean;

}

@Override

**public** Object postProcessAfterInitialization(Object bean, String beanName) **throws** BeansException {

System.***out***.println(bean.getClass().getName() + "初始化完毕");

System.***out***.println("其名为：" + beanName);

**return** bean;

}

}

// 异常简单的main方法

**public** **static** **void** main(String[] args) **throws** Exception {

ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext("config/applicationContext-bean.xml");

}

// 输出结果

