* + [首页](https://juejin.im/)
  + [沸点](https://juejin.im/pins)
  + [话题](https://juejin.im/topics)
  + [小册](https://juejin.im/books)
  + [活动](https://juejin.im/events/all)

窗体顶端

窗体底端

* 写文章
* 登录注册

[**mortal同学**](https://juejin.im/user/5bdacfabe51d4509691a5c69)

2019年01月04日阅读 1996

关注

**Spring学习笔记**

**Spring简介**

Spring 是一个开源框架，是一个分层的 JavaEE 一站式框架。

所谓一站式框架是指 Spring 有 JavaEE 开发的每一层解决方案。

* WEB层：SpringMVC
* Service层：Spring的Bean管理，声明式事务
* DAO层：Spring的JDBC模板，ORM模板

优点：

* IOC：方便解耦合
* AOP：对程序进行扩展
* 轻量级框架
* 方便与其他框架整合

[Spring 源码解析](https://blog.csdn.net/lyc_liyanchao/article/details/82891532)

**Spring使用**

Spring开发包解压后的目录介绍：

* docs: Spring 开发规范和API
* libs: Spring jar 包和源代码
* schema: Spring 配置文件的约束

DataAccess 用于数据访问，WEB 用于页面显示，核心容器也就是IOC部分。

**控制反转（IOC）**

控制反转（Inversion of Control）是指将对象的创建权反转（交给）Spring。

使用IOC就需要导入IOC相关的包，也就是上图中核心容器中的几个包：beans,context,core,expression四个包。

**基础概念**

在Java开发中，Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。

传统Java SE程序设计，我们直接在对象内部通过new进行创建对象，是程序主动去创建依赖对象；而IoC是有专门一个容器来创建这些对象，即由Ioc容器来控制对象的创建。

传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而反转则是由容器来帮忙创建及注入依赖对象；为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖对象的获取被反转了。

**实现原理**

传统方式创建对象：

UserDAO userDAO=**new** UserDAO();

复制代码

进一步面向接口编程，可以多态：

UserDAO userDAO=**new** UserDAOImpl();

复制代码

这种方式的缺点是接口和实现类高耦合，切换底层实现类时，需要修改源代码。程序设计应该满足OCP元祖，在尽量不修改程序源代码的基础上对程序进行扩展。此时，可以使用工厂模式：

**class** **BeanFactory**{

**public** **static** UserDAO **getUserDAO**(){

**return** **new** UserDAOImpl();

}

}

复制代码

此种方式虽然在接口和实现类之间没有耦合，但是接口和工厂之间存在耦合。

使用工厂+反射+配置文件的方式，实现解耦，**这也是 Spring 框架 IOC 的底层实现。**

//xml配置文件

//<bean id="userDAO" class="xxx.UserDAOImpl"></bean>

**class** **BeanFactory**{

**public** **static** Object **getBean**(String id){

//解析XML

//反射

Class clazz=Class.forName();

**return** clazz.newInstance();

}

}

复制代码

**IOC XML 开发**

在 docs 文件中包含了 xsd-configuration.hmtl 文件。其中定义了 beans schema。

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

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

xsi:schemaLocation="

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

//在此配置bean

<bean id="userService" class="x.y.UserServiceImpl">

</bean>

</beans>

复制代码

调用类：

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

UserService userService=(UserService)applicationContext.getBean("userService");

userService.save();

复制代码

**IOC 和 DI**

DI 指依赖注入，其前提是必须有 IOC 的环境，Spring 管理这个类的时候将类的依赖的属性注入进来。

例如，在UserServiceImpl.java中：

**public** **class** **UserServiceImpl** **implements** **UserService**{

**private** String name;

**public** **void** **setName**(String name){

**this**.name=name;

}

**public** **void** **save**(){

System.out.println("save "+name);

}

}

复制代码

在配置文件中：

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

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

xsi:schemaLocation="

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

<bean id="userService" class="spring.demo1.UserServiceImpl">

<!--配置依赖的属性-->

<property name="name" value="tony"/>

</bean>

</beans>

复制代码

测试代码：

**@Test**

**public** **void** **demo2**(){

//创建Spring工厂

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

UserService userService=(UserService)applicationContext.getBean("userService");

userService.save();

}

复制代码

运行结果：

save tony

复制代码

可以看到，在配置文件中配置的属性，在 Spring 管理该类的时候将其依赖的属性成功进行了设置。如果不使用依赖注入，则无法使用接口，只能使用实现类来进行设置，因为接口中没有该属性。

**Spring 的工厂类**

* BeanFactory: 老版本的工厂类，在调用getBean()方法时，才会生成类的实例。
* ApplicationContext: 在加载配置文件的时候，就会将 Spring 管理的类都实例化。有两个实现类：
  1. ClassPathXmlApplicationContext: 加载类路径下的配置文件
  2. FileSystemXmlApplicationContext: 加载磁盘下的配置文件

**bean标签配置**

* id: 唯一约束，不能出现特殊字符
* name: 理论上可以重复，但是开发中最好不要。可以出现特殊字符

生命周期：

* init-method: bean被初始化的时候执行的方法
* destroy-method: bean被销毁的时候执行的方法

作用范围：

* scope： bean的作用范围，有如下几种，常用的是前两种
  + singleton: 默认使用单例模式创建
  + prototype: 多例
  + request: 在web项目中，spring 创建类后，将其存入到 request 范围中
  + session: 在web项目中，spring 创建类后，将其存入到 session 范围中
  + globalsession: 在web项目中，必须用在 porlet 环境

**属性注入设置**

1. 构造方法方式的属性注入： Car 类在构造方法中有两个属性，分别为 name 和 price。

<bean id="car" class="demo.Car">

<constructor-arg name="name" value="bmw">

<constructor-arg name="price" value="123">

</bean>

复制代码

1. set 方法属性注入： Employee 类在有两个 set 方法，分别设置普通类型的 name 和引用类型的 Car （使用 ref 指向引用类型的 id 或 name）。

<bean id="employee" class="demo.Employee">

<property name="name" value="xiaoming">

<property name="car" ref="car">

</bean>

复制代码

1. P名称空间的属性注入： 首先需要引入p名称空间：

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

//引入p名称空间

xmlns:p="http://www.springframework.org/schema/p"

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

xsi:schemaLocation="

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

</beans>

复制代码

如果是普通属性：

<bean id="car" class="demo.Car" p:name="bmv" p:price="123">

</bean>

复制代码

如果是引用类型：

<bean id="employee" class="demo.Employee" p:name="xiaoming" p:car-ref:"car">

</bean>

复制代码

1. SpEL(Spring Expression Language)属性注入（Spring 3.x以上版本）

<bean id="car" class="demo.Car">

<property name="name" value="#{'xiaoming'}">

<property name="car" ref="#{car}">

</bean>

复制代码

1. 集合类型属性注入：

<bean id="car" class="demo.Car">

<property name="namelist">

<list>

<value>qirui</value>

<value>baoma</value>

<value>benchi</value>

</list>

</property>

</bean>

复制代码

**多模块开发配置**

1. 在加载配置文件的时候，加载多个配置文件
2. 在一个配置文件中引入多个配置文件，通过实现

**IOC 注解开发**

**示例**

1. 引入jar包： 除了要引入上述的四个包之外，还需要引入aop包。
2. 创建 applicationContext.xml ，使用注解开发引入 context 约束（xsd-configuration.html）

<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" xsi:schemaLocation="

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

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

<!-- bean definitions here -->

</beans>

复制代码

1. 组件扫描： 使用IOC注解开发，需要配置组件扫描，也就是哪些包下的类使用IOC的注解。

<context:component-scan base-package="demo1">

复制代码

1. 在类上添加注解
2. 使用注解设置属性的值

属性如果有set方法，将属性注入的注解添加到set方法

属性没有set方法，将注解添加到属性上。

**@Component**("UserDao")//相当于配置了一个<bean> 其id为UserDao,对应的类为该类

**public** **class** **UserDAOImpl** **implements** **UserDAO** {

**@Override**

**public** **void** **save**() {

// TODO Auto-generated method stub

System.out.println("save");

}

}

复制代码

**注解详解**

**1.@Component**

组件注解，用于修饰一个类，将这个类交给 Spring 管理。

有三个衍生的注解，功能类似，也用来修饰类。

* @Controller：修饰 web 层类
* @Service：修饰 service 层类
* @Repository：修饰 dao 层类

**2.属性注入**

* 普通属性使用 @Value 来设置属性的值
* 对象属性使用 @Autowired ，这个注解是按照类型来进行属性注入的。如果希望按照 bean 的名称或id进行属性注入，需要用 @Autowired 和 @Qualifier 一起使用
* 实际开发中，使用 @Resource(name=" ") 来进行按照对象的名称完成属性注入

**3.其他注解**

* @PostConstruct 相当于 init-method，用于初始化函数的注解
* @PreDestroy 相当于 destroy-method，用于销毁函数的注解
* @Scope 作用范围的注解，常用的是默认单例，还有多例 @Scope("prototype")

**IOC 的 XML 和注解开发比较**

* 适用场景：XML 适用于任何场景；注解只适合自己写的类，不是自己提供的类无法添加注解。
* 可以使用 XML 管理 bean，使用注解来进行属性注入

**IOC与传统方式的比较**

1. 获取对象方式：传统通过 new 关键字主动创建一个对象。IOC 方式中，将对象的生命周期交给 Spring 管理，直接从 Spring 获取对象。也就是控制反转————将控制权从自己手中交到了 Spring 手中。

**AOP开发**

AOP 是 Aspect Oriented Programming 的缩写，意为面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术，是OOP的延续。

AOP 能够对程序进行增强，在不修改源码的情况下，可以进行权限校验，日志记录，性能监控，事务控制等。

**也就是说功能分为两大类，一类是核心业务功能，一类是辅助增强功能。两类功能彼此独立进行开发。比如登录功能是核心业务功能，日志功能是辅助增强功能，如果有需要，将日志和登录编制在一起。辅助功能就称为切面，这种能选择性的、低耦合的把切面和核心业务功能结合的编程思想称为切面编程。**

**AOP的初衷**

DRY：Dont Repeat Yourself SoC: Sepration of Concerns

* 水平分离：展示层->服务层->持久层
* 垂直分离：模块划分
* 切面分离：分离功能性和非功能性需求

**为何使用AOP**

* 集中处理某一关注点/横切逻辑
* 方便添加/删除关注点
* 侵入性少，增强代码可读性和可维护性

主要应用于权限控制，缓存控制，事务控制，审计日志，性能监控，分布式追踪，异常处理等等。

**底层实现**

JDK 动态代理只能对实现了接口的类产生代理。Cglib 动态代理可以对没有实现接口的类产生代理对象，生成的是子类对象。

**JDK 动态代理**

JDK 动态代理基于接口实现，底层是基于反射。

**public** **interface** **Subject** {

**public** **void** **request**();

}

复制代码

实现类：

**public** **class** **RealSubject** **implements** **Subject** {

**@Override**

**public** **void** **request**() {

System.out.println("real subject request");

}

}

复制代码

JDK 代理：

**public** **class** **JdkProxySubject** **implements** **InvocationHandler** {

**private** RealSubject realSubject;

**public** **JdkProxySubject**(RealSubject realSubject) {

**this**.realSubject = realSubject;

}

**@Override**

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

System.out.println("before");

Object result = **null**;

**try** {

result = method.invoke(realSubject, args);

} **catch** (Exception e) {

**throw** e;

} **finally** {

System.out.println("after");

}

**return** result;

}

}

复制代码

测试类

**public** **class** **Client** {

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

RealSubject rs=**new** RealSubject();

InvocationHandler handler=**new** JdkProxySubject(rs);

Subject subject=(Subject) Proxy.newProxyInstance(rs.getClass().getClassLoader(),rs.getClass().getInterfaces(),handler);

subject.request();

}

}

复制代码

运行结果：

before

real subject request

after

复制代码

在 Client 中，通过调用 Proxy.newProxyInstance() 生成代理对象，其中参数分别是 classLoader, 要代理的接口, 以及代理对象的 InvocationHandler。当调用subject.request()时，实际调用的是JdkProxySubject里的invoke方法。

**Cglib**

Cglib 是第三方开源代码生成类库，可以动态添加类的属性和方法。

JDK 代理基于接口实现，而 Cglib 基于继承实现：

**public** **class** **Hello** {

**public** **void** **hello**(String str){

System.out.println("hello "+str);

}

}

复制代码

该类并没有实现任何接口。通过CGLIB代理实现如下： 首先实现一个MethodInterceptor，方法调用会被转发到该类的intercept()方法。 然后在需要使用该类的时候，通过CGLIB动态代理获取代理对象。

**class** **MyMethodInterceptor** **implements** **MethodInterceptor**{

...

**@Override**

**public** Object **intercept**(Object obj, Method method, Object[] args, MethodProxy proxy) **throws** Throwable {

logger.info("You said: " + Arrays.toString(args));

**return** proxy.invokeSuper(obj, args);

}

}

复制代码

**public** **class** **Test** {

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

Enhancer enhancer = **new** Enhancer();

enhancer.setSuperclass(HelloConcrete.class);

enhancer.setCallback(**new** MyMethodInterceptor());

HelloConcrete hello = (HelloConcrete)enhancer.create();

System.out.println(hello.sayHello("I love you!"));

}

}

复制代码

通过CGLIB的Enhancer来指定要代理的目标对象、实际处理代理逻辑的对象，最终通过调用create()方法得到代理对象，对这个对象所有非final方法的调用都会转发给MethodInterceptor.intercept()方法。

**两者比较**

何时使用JDK还是CGLiB？

* 如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP。
* 如果目标对象实现了接口，可以强制使用CGLIB实现AOP。
* 如果目标对象没有实现了接口，必须采用CGLIB库，Spring会自动在JDK动态代理和CGLIB之间转换。

Spring如何选择用JDK还是CGLiB？

* 当Bean实现接口时，Spring就会用JDK的动态代理。
* 当Bean没有实现接口时，Spring使用CGlib是实现。
* 可以强制使用CGlib（在spring配置中加入<aop:aspectj-autoproxy proxy-target-class="true"/>）。

**Spring 的 AOP 开发（AspectJ 的 XML 方式）**

AspectJ 是一个 AOP 的框架，Spring 引入 AspectJ，基于 AspectJ 进行 AOP 的开发。

**相关术语**

* Joinpoint: 连接点，可以被拦截到的点。指程序运行时允许插入切面的一个点，可以是一个函数、一个包路径、一个类、或者抛出的异常
* Pointcut: 切入点,用于指定连接点的范围，真正被拦截到的点，也就是真正被增强的方法
* Advice: 通知，定义了切面在切点附近的操作,也就是在什么时候做什么事。如 before, after 等。
* Aspect: 切面，切入点和通知的聚合，一般是一个横跨多各类的通用逻辑抽象而成的类，切面类会定义在什么时间、什么地点、做什么事。
* Introduction: 引介，类层面的增强。
* Target: 目标，被增强的对象（类），也就是被切面织入的对象。
* Weaving: 织入是把切面应用到切点对应的连接点的过程。切面在指定连接点被织入到目标对象中。
* Proxy: 代理对象，被增强的对象。

**使用方法**

1. 引入相关包
2. 引入配置文件

**<?**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: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 definitions here -->

</beans>

复制代码

1. 编写目标类并配置：

**public** **class** **ProductDaoImpl** **implements** **ProductDao** {

**@Override**

**public** **void** **save**() {

System.out.println("save");

}

**@Override**

**public** **void** **update**() {

System.out.println("update");

}

**@Override**

**public** **void** **find**() {

System.out.println("find");

}

**@Override**

**public** **void** **delete**() {

System.out.println("delete");

}

}

复制代码

<bean id="productDao" class="demo1.ProductDaoImpl"></bean>

复制代码

1. 编写切面类，假设用于权限验证并配置

**public** **class** **MyAspectXML** {

**public** **void** **checkPri**(){

System.out.println("check auth");

}

}

复制代码

<bean id="myAspect" class="demo1.MyAspectXML"></bean>

复制代码

1. 通过AOP配置完成对目标类的增强

<aop:config>

<aop:pointcut expression="execution(\* demo1.ProductDaoImpl.save(..))" id="pointcut1"/>

<aop:aspect ref="myAspect">

<aop:before method="chechPri" pointcut-ref="pointcut1"/>

</aop:aspect>

</aop:config>

复制代码

**通知类型**

1. 前置通知：在目标方法执行前操作，可以获得切入点信息

<aop:before method="chechPri" pointcut-ref="pointcut1"/>

复制代码

**public** **void** **checkPri**(JoinPoint joinPoint){

System.out.println("check auth "+joinPoint);

}

复制代码

1. 后置通知：在目标方法执行后操作，可以获得方法返回值

<aop:after-returning method="writeLog" pointcut-ref="pointcut2" returning="result"/>

复制代码

**public** **void** **writeLog**(Object result){

System.out.println("writeLog "+result);

}

复制代码

1. 环绕通知：在目标方法执行前和后操作，可以阻止目标方法执行

<aop:around method="around" pointcut-ref="pointcut3"/>

复制代码

**public** Object **around**(ProceedingJoinPoint joinPoint) **throws** Throwable{

System.out.println("before");

Object result=joinPoint.proceed();

System.out.println("after");

**return** result;

}

复制代码

1. 异常抛出通知：程序出现异常时操作

<aop:after-throwing method="afterThrowing" pointcut-ref="pointcut4" throwing="ex"/>

复制代码

**public** **void** **afterThrowing**(Throwable ex){

System.out.println("exception "+ex.getMessage());

}

复制代码

1. 最终通知：相当于finally块，无论代码是否有异常，都会执行

<aop:after method="finallyFunc" pointcut-ref="pointcut4"/>

复制代码

**public** **void** **finallyFunc**(){

System.out.println("finally");

}

复制代码

1. 引介通知：不常用

**Spring 切入点表达式**

基于 execution 函数完成

语法：[访问修饰符] 方法返回值 包名.类名.方法名(参数)

其中任意字段可以使用\*代替表示任意值

**Spring 的 AOP 基于 AspectJ 注解开发**

**开发步骤**

1. 引入jar包
2. 设置配置文件：

**<?**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"

xmlns:tx="http://www.springframework.org/schema/tx"

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

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

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

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

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

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

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

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

</beans>

复制代码

1. 编写配置目标类

<bean id="orderDao" class="demo1.OrderDao"></bean>

复制代码

**public** **class** **OrderDao** {

**public** **void** **save**(){

System.out.println("save order");

}

**public** **void** **update**(){

System.out.println("update order");

}

**public** **void** **delete**(){

System.out.println("delete order");

}

**public** **void** **find**(){

System.out.println("find order");

}

}

复制代码

1. 开启aop注解自动代理

<aop:aspectj-autoproxy/>

复制代码

1. 编写切面类并配置

**@Aspect**

**public** **class** **MyAspectAnno** {

**@Before**(value="execution(\* demo1.OrderDao.save(..))")

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

System.out.println("before");

}

}

复制代码

<bean id="myAspect" class="demo1.MyAspectAnno">

复制代码

**注解类型**

注解主要分为三类：

1. @Aspect：表示这是一个切面类
2. @Pointcut：表示切入点，也就是需要增强的类或方法

这个注解需要借助切面表达式组成。切面表达式由三个部分构成：指示器（designators）、通配符（wildcards）以及操作符（operators）。

指示器分为：

* **匹配方法：execution()**

execution(

[modifier-pattern]// 修饰符

ret-type-pattern//返回类型

[declaring-type-pattern]//包名

name-pattern(param-pattern)//方法名

[throws-pattern]//抛出异常声明

)

复制代码

* 匹配注解：@target(),@args(),@within(),@annotation()
* 匹配包/类：within()

**@Pointcut**("within(demo.service.\*)")

**public** **void** **pointcut**(){}

复制代码

* 匹配对象：this(),bean(),target()
* 匹配参数：args()

//匹配任何只有一个Long参数的方法

**@Pointcut**("args(Long)")

//匹配第一个参数为Long的方法

**@Pointcut**("args(Long,..)")

复制代码

通配符有：

* + 匹配任意数量的字符
  + 匹配指定类及其子列
* .. 用于匹配任意数的子包或参数

运算符主要有&&，||，！也就是与或非三个。

1. Advice: 在什么时候执行增强的方法

* @Before： 前置通知
* @AfterReturning: 后置通知

**@AfterReturning**(value="execution(\* demo1.OrderDao.save(..))",returning="result")

**public** **void** **after**(Object result){

System.out.println("after "+result);

}

复制代码

* @Around：环绕通知

**@Around**(value="execution(\* demo1.OrderDao.save(..))")

**public** Object **around**(ProceedingJoinPoint joinPoint) **throws** Throwable{

System.out.println("before");

Object obj=joinPoint.proceed();

System.out.println("after");

**return** obj;

}

复制代码

* @AfterThrowing: 抛出异常

@AfterThrowing(value="execution(\* demo1.OrderDao.save(..))",throwing="e")

public void afterThrowing(Throwable e){

System.out.println("exception:"+e.getMessage();

}

复制代码

* @After: 最终通知

**@After**(value="execution(\* demo1.OrderDao.save(..))")

**public** **void** **after**(){

System.out.println("finally");

}

复制代码

* @PointCut:切入点注解

**@PointCut**(value="execution(\* demo1.OrderDao.save(..))")

**private** **void** **pointcut1**(){}

复制代码

注解的好处是，只需要维护切入点即可，不用在修改时修改每个注解。

**Spring 的 JDBC 模板**

Spring 对持久层也提供了解决方案，也就是 ORM 模块和 JDBC 的模板。针对 JDBC ，提供了 org.springframework.jdbc.core.JdbcTemplate 作为模板类。

**使用 JDBC 模板**

1. 引入jar包，数据库驱动，Spring 的 jdbc 相关包。
2. 基本使用：

**public** **void** **demo1**(){

//创建连接池

DriverManagerDataSource dataSource=**new** DriverManagerDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql:///spring4");

dataSource.setUsername("root");

dataSource.setPassword("123456");

//创建JDBC模板

JdbcTemplate jdbcTemplate=**new** JdbcTemplate(dataSource);

jdbcTemplate.update("insert into account values (null,?,?)", "xiaoming",1000d);

}

复制代码

1. 将连接池和模板交给 Spring 管理

* 配置文件：

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource;">

<property name="driverClassName" value="com.mysql.jdbc.Driver"></property>

<property name="url" value="jdbc:mysql:///spring4"></property>

<property name="username" value="root"></property>

<property name="password" value="123456"></property>

</bean>

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

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

</bean>

复制代码

* 测试文件：

**@RunWith**(SpringJUnit4ClassRunner.class)

**@ContextConfiguration**("classpath:applicationContext.xml")

**public** **class** **JdbcDemo2** {

**@Resource**(name="jdbcTemplate")

**private** JdbcTemplate jdbcTemplate;

**@Test**

**public** **void** **demo2**(){

jdbcTemplate.update("insert into account values (null,?,?)", "xiaolan",1000d);

}

}

复制代码

**使用开源数据库连接池**

1. 使用 DBCP 的配置：

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver"></property>

<property name="url" value="jdbc:mysql://192.168.66.128/spring4"></property>

<property name="username" value="root"></property>

<property name="password" value="123456"></property>

复制代码

1. 使用 C3P0 的配置：

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

<property name="driverClass" value="com.mysql.jdbc.Driver"></property>

<property name="jdbcUrl" value="jdbc:mysql://192.168.66.128/spring4"></property>

<property name="user" value="root"></property>

<property name="password" value="123456"></property>

</bean>

复制代码

1. 引入外部属性文件

首先建立外部属性文件：

jdbc.driverClass=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://192.168.66.128/spring4

jdbc.username=root

jdbc.password=123456

复制代码

然后对属性文件进行配置：

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

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

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

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

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

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

</bean>

复制代码

**CRUD操作**

insert, update, delete 语句都借助模板的 update 方法进行操作。

**public** **void** **demo**(){

jdbcTemplate.update("insert into account values (null,?,?)", "xiaoda",1000d);

jdbcTemplate.update("update account set name=?,money=? where id=?", "xiaoda",1000d,2);

jdbcTemplate.update("delete from account where id=?", 6);

}

复制代码

查询操作：

**public** **void** **demo3**(){

String name=jdbcTemplate.queryForObject("select name from account where id=?",String.class,5);

**long** count=jdbcTemplate.queryForObject("select count(\*) from account",Long.class);

}

复制代码

将返回的结果封装成为类：

**public** **void** **demo4**(){

Account account=jdbcTemplate.queryForObject("select \* from account where id=?", **new** MyRowMapper(),5);

}

复制代码

其中：

**class** **MyRowMapper** **implements** **RowMapper**<**Account**>{

**@Override**

**public** Account **mapRow**(ResultSet rs, **int** rowNum) **throws** SQLException {

Account account=**new** Account();

account.setId(rs.getInt("id"));

account.setName(rs.getString("name"));

account.setMoney(rs.getDouble("money"));

**return** account;

}

}

复制代码

**Spring的事务管理**

**事务**

事务是指逻辑上的一组操作，组成这组操作的各个单元，要么全部成功，要么全部失败。

具有四个特性：

* 原子性：事务不可分
* 一致性：事务执行前后数据完整性保持一致
* 隔离性：一个事务的执行不应该受到其他事务干扰
* 持久性：一旦事务结束，数据就持久化到数据库

如果不考虑隔离性会引发安全性问题：

* 读问题：
  + 脏读：一个事务读到另一个事务未提交的数据
  + 不可重复读：一个事务读到另一个事务已经提交的 update 数据，导致一个事务中多次查询结果不一致
  + 幻读：一个事务读到另一个事务已经提交的 insert 数据，导致一个事务中多次查询结果不一致
* 写问题：
  + 丢失更新

解决读问题：设置事务隔离级别

* Read uncommitted: 未提交读，无法解决任何读问题
* Read committed: 已提交读，解决脏读问题
* Repeatable read: 重复读，解决脏读和不可重复读问题
* Serializable：序列化，解决所有读问题

**事务管理API**

1. PlatformTransactionManager: 平台事务管理器

这是一个接口，拥有多个不同的实现类，如 DataSourceTransactionManager 底层使用了JDBC 管理事务； HibernateTransactionManager 底层使用了 Hibernate 管理事务。

1. TransactionDefinition: 事务定义信息

用于定义事务的相关信息，如隔离级别、超时信息、传播行为、是否只读等

1. TransactionStatus: 事务的状态

用于记录在事务管理过程中，事务的状态的对象。

**上述API的关系**： Spring 在进行事务管理的时候，首先**平台事务管理器**根据**事务定义信息**进行事务管理，在事务管理过程当中，产生各种此状态，将这些状态信息记录到**事务状态**的对象当中。

**事务的传播行为**

事务的传播行为主要解决业务层（Service）方法相互调用的问题，也就是不同的业务中存在不同的事务时，如何操作。

Spring 中提供了7种事务的传播行为，分为三类：

* 保证多个操作在同一个事务中
  + **PROPAGATION\_REQUIRED**: B方法调用A方法，如果A中有事务，使用A中的事务并将B中的操作包含到该事务中；否则新建一个事务，将A和B中的操作包含进来。（默认）
  + PROPAGATION\_SUPPORTS：如果A中有事务，使用A的事务；否则不使用事务
  + PROPAGATION\_MANDATORY：如果A中有事务，使用A的事务；否则抛出异常
* 保证多个操作不在同一个事务中
  + **PROPAGATION\_REQUIRES\_NEW**：如果A中有事务，将其挂起，创建新事务，只包含自身操作。否则，新建一个事务，只包含自身操作。
  + PROPAGATION\_NOT\_SUPPORTED：如果A中有事务，挂起，不使用事务。
  + PROPAGATION\_NEVER：如果A中有事务，抛出异常，也即不能用事务运行。
* 嵌套事务
  + **PROPAGATION\_NESTED**：如果A有事务，按照A的事务执行，执行完成后，设置一个保存点，然后执行B的操作。如果出现异常，可以回滚到最初状态或保存点状态。

**实例**

以转账为例，业务层的DAO层类如下：

**public** **interface** **AccountDao** {

**public** **void** **outMoney**(String from,Double money);

**public** **void** **inMoney**(String to,Double money);

}

**public** **class** **AccountDaoImpl** **extends** **JdbcDaoSupport** **implements** **AccountDao**{

**@Override**

**public** **void** **outMoney**(String from, Double money) {

**this**.getJdbcTemplate().update("update account set money = money - ? where name = ?",money,from);

}

**@Override**

**public** **void** **inMoney**(String to, Double money) {

**this**.getJdbcTemplate().update("update account set money = money + ? where name = ?",money,to);

}

}

**public** **interface** **AccountService** {

**public** **void** **transfer**(String from,String to,Double money);

}

**public** **class** **AccountServiceImpl** **implements** **AccountService** {

**private** AccountDao accountDao;

**public** **void** **setAccountDao**(AccountDao accountDao) {

**this**.accountDao = accountDao;

}

**@Override**

**public** **void** **transfer**(String from, String to, Double money) {

accountDao.outMoney(from, money);

accountDao.inMoney(to, money);

}

}

复制代码

在xml中进行类的配置：

<bean id="accountService" class="tx.demo.AccountServiceImpl">

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

</bean>

<bean id="accountDao" class="tx.demo.AccountDaoImpl">

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

</bean>

复制代码

**事务管理1: 编程式事务管理**

1. 配置平台事务管理器

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

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

</bean>

复制代码

1. 配置事务管理模板类

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

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

</bean>

复制代码

1. 在业务层注入事务管理模板

<bean id="accountService" class="tx.demo1.AccountServiceImpl">

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

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

</bean>

复制代码

1. 编码实现事务管理

//ServiceImpl类中：

**private** TransactionTemplate transactionTemplate;

**@Override**

**public** **void** **transfer**(String from, String to, Double money) {

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

**@Override**

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

accountDao.outMoney(from, money);

accountDao.inMoney(to, money);

}

});

}

复制代码

**声明式事务管理（配置实现，基于AOP思想）**

1. XML 方式的声明式事务管理

* 配置事务管理器

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

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

</bean>

复制代码

* 配置事务通知

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

<tx:attributes>

<tx:method name="transfer" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

复制代码

* 配置aop事务

<aop:config>

<aop:pointcut expression="execution(\* tx.demo2.AccountServiceImpl.\*(..))" id="pointcut1"/>

<aop:advisor advice-ref="txAdvice" pointcut-ref="pointcut1"/>

</aop:config>

复制代码

1. 注解方式

* 配置事务管理器，和上方一致
* 开启事务管理的注解：

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

复制代码

* 在使用事务的类上添加一个注解@Transactional

**关注下面的标签，发现更多相似文章**

[Java](https://juejin.im/tag/Java" \t "_blank)

[**mortal同学**](https://juejin.im/user/5bdacfabe51d4509691a5c69)

[发布了 48 篇专栏 ·](https://juejin.im/user/5bdacfabe51d4509691a5c69/posts)获得点赞 289 · 获得阅读 14,598

关注

[安装掘金浏览器插件](https://juejin.im/extension/?utm_source=juejin.im&utm_medium=post&utm_campaign=extension_promotion)

打开新标签页发现好内容，掘金、GitHub、Dribbble、ProductHunt 等站点内容轻松获取。快来安装掘金浏览器插件获取高质量内容吧！

评论

[sanlulu](https://juejin.im/user/57ccd04d165abd0068f35568)

总结的挺好的

1年前

回复

[kanjiushi](https://juejin.im/user/575660f2d342d3006b4b2247)

不懂java,请教一下楼主spring和spring boot的关系是什么？

1年前

回复

[Jackdream](https://juejin.im/user/5be244dde51d4520fc446a6e)

JAVA工程师

学习了

1年前

回复

[溢性循环](https://juejin.im/user/5a61d6bc6fb9a01cab28560f)

再更新下spring boot的配置就更好了

1年前

回复

**相关推荐**

* + [专栏](https://juejin.im/post/5e7f150d51882573b3309ceb" \t "_blank)

[Jay\_huaxiao](https://juejin.im/user/5cc588845188252dd006cc4e)

* + 22小时前
  + [Java](https://juejin.im/tag/Java)

[**Java程序员必备：序列化全方位解析**](https://juejin.im/post/5e7f150d51882573b3309ceb)

* + **22**
  + [专栏](https://juejin.im/post/5e98159ae51d45470027417b" \t "_blank)

[Guide哥](https://juejin.im/user/59fbb2daf265da4319559f3a)

* + 3天前
  + [Java](https://juejin.im/tag/Java)[IntelliJ IDEA](https://juejin.im/tag/IntelliJ%20IDEA)

[**第一弹！安排！安利10个让你爽到爆的IDEA必备插件！**](https://juejin.im/post/5e98159ae51d45470027417b)

* + **128**
  + [**12**](https://juejin.im/post/5e98159ae51d45470027417b#comment)
  + [专栏](https://juejin.im/post/5e9c56e4f265da47b35c7d5f" \t "_blank)

[风平浪静如码](https://juejin.im/user/5d9ecfc6f265da5b991d5a45)

* + 9小时前
  + [Java](https://juejin.im/tag/Java)

[**一篇文章解析Redis的数据结构和对象系统是怎么设计的？**](https://juejin.im/post/5e9c56e4f265da47b35c7d5f)

* + **5**
  + [专栏](https://juejin.im/post/5e9c536d6fb9a03c6d3db164" \t "_blank)

[mghio](https://juejin.im/user/5bbccb4b6fb9a05ce95c7a81)

* + 10小时前
  + [Java](https://juejin.im/tag/Java)

[**如何在亿级数据中判断一个元素是否存在？**](https://juejin.im/post/5e9c536d6fb9a03c6d3db164)

* + **2**
  + [**1**](https://juejin.im/post/5e9c536d6fb9a03c6d3db164#comment)
  + [专栏](https://juejin.im/post/5e999ff0518825738e2180e0" \t "_blank)

[六脉神剑](https://juejin.im/user/5dd75dc1f265da7dc9557eae)

* + 1天前
  + [Java](https://juejin.im/tag/Java)

[**多线程之按顺序打印结果**](https://juejin.im/post/5e999ff0518825738e2180e0)

* + **5**
  + [专栏](https://juejin.im/post/5e9c45bb6fb9a03c854630a2" \t "_blank)

[Java从入坑到入土](https://juejin.im/user/5e6caa3c6fb9a07cd80f3937)

* + 10小时前
  + [Java](https://juejin.im/tag/Java)

[**Java并发篇—线程池入门扫盲指南**](https://juejin.im/post/5e9c45bb6fb9a03c854630a2)

* + **2**
  + [专栏](https://juejin.im/post/5e9bb182f265da47ca6932af" \t "_blank)

[go4it](https://juejin.im/user/58ae3672ac502e006c881d65)

* + 21小时前
  + [Java](https://juejin.im/tag/Java)

[**聊聊canal的Position**](https://juejin.im/post/5e9bb182f265da47ca6932af)

* + [专栏](https://juejin.im/post/5e9903dfe51d454714427e56" \t "_blank)

[Java中文社群](https://juejin.im/user/5a62ee8af265da3e290c4aa4)

* + 2天前
  + [Java](https://juejin.im/tag/Java)

[**别再问我 new 字符串创建了几个对象了！我来证明给你看！**](https://juejin.im/post/5e9903dfe51d454714427e56)

* + **62**
  + [**17**](https://juejin.im/post/5e9903dfe51d454714427e56#comment)
  + [专栏](https://juejin.im/post/5e93c34be51d4546cf778291" \t "_blank)

[HollisChuang](https://juejin.im/user/56d79d3f128fe100513b924a)

* + 6天前
  + [Java](https://juejin.im/tag/Java)

[**Java工程师成神之路 | 2020正式版**](https://juejin.im/post/5e93c34be51d4546cf778291)

* + **185**
  + [**13**](https://juejin.im/post/5e93c34be51d4546cf778291#comment)

关于作者

[**mortal同学**](https://juejin.im/user/5bdacfabe51d4509691a5c69)

获得点赞289

文章被阅读14,598

掘金小册

[JVM 字节码从入门到精通](https://juejin.im/book/5c25811a6fb9a049ec6b23ee)

[新人价 ￥9.95](https://juejin.im/book/5c25811a6fb9a049ec6b23ee)

~~[￥19.9](https://juejin.im/book/5c25811a6fb9a049ec6b23ee)~~

[JavaScript 设计模式核⼼原理与应⽤实践](https://juejin.im/book/5c70fc83518825428d7f9dfb)

[新人价 ￥9.95](https://juejin.im/book/5c70fc83518825428d7f9dfb)

~~[￥19.9](https://juejin.im/book/5c70fc83518825428d7f9dfb)~~

新人专享好礼

**送你45元买小册**

立即领取

相关文章

[Java 多线程和高并发面试题](https://juejin.im/post/5cb4551ef265da03b858443d" \t "_blank)

[16](https://juejin.im/post/5cb4551ef265da03b858443d" \t "_blank)

[1](https://juejin.im/post/5cb4551ef265da03b858443d" \t "_blank)

[JVM+GC 面试题](https://juejin.im/post/5cb872a2e51d456e79545c6c" \t "_blank)

[56](https://juejin.im/post/5cb872a2e51d456e79545c6c" \t "_blank)

[3](https://juejin.im/post/5cb872a2e51d456e79545c6c" \t "_blank)

[Java并发5：ConcurrentHashMap](https://juejin.im/post/5c26146f5188250d104cd49f" \t "_blank)

[30](https://juejin.im/post/5c26146f5188250d104cd49f" \t "_blank)

[0](https://juejin.im/post/5c26146f5188250d104cd49f" \t "_blank)

[Java并发编程实战笔记3：基础构建模块](https://juejin.im/post/5c19aa876fb9a049fe3516f0" \t "_blank)

[24](https://juejin.im/post/5c19aa876fb9a049fe3516f0" \t "_blank)

[0](https://juejin.im/post/5c19aa876fb9a049fe3516f0" \t "_blank)

[Java并发6：阻塞队列，Fork/Join框架](https://juejin.im/post/5c2c52d9e51d455405559378" \t "_blank)

[16](https://juejin.im/post/5c2c52d9e51d455405559378" \t "_blank)

[1](https://juejin.im/post/5c2c52d9e51d455405559378" \t "_blank)

目录

* [**Spring简介**](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-0)
  + [Spring使用](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-1)
* [**控制反转（IOC）**](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-2)
  + [基础概念](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-3)
  + [实现原理](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-4)
  + [IOC XML 开发](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-5)
    - [IOC 和 DI](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-6)
    - [Spring 的工厂类](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-7)
    - [bean标签配置](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-8)
    - [属性注入设置](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-9)
    - [多模块开发配置](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-10)
  + [IOC 注解开发](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-11)
    - [示例](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-12)
    - [注解详解](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-13)
    - [1.@Component](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-14)
    - [2.属性注入](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-15)
    - [3.其他注解](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-16)
    - [IOC 的 XML 和注解开发比较](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-17)
    - [IOC与传统方式的比较](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-18)
* [**AOP开发**](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-19)
  + [AOP的初衷](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-20)
  + [为何使用AOP](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-21)
  + [底层实现](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-22)
    - [JDK 动态代理](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-23)
    - [Cglib](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-24)
    - [两者比较](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-25)
  + [Spring 的 AOP 开发（AspectJ 的 XML 方式）](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-26)
    - [相关术语](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-27)
    - [使用方法](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-28)
    - [通知类型](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-29)
    - [Spring 切入点表达式](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-30)
  + [Spring 的 AOP 基于 AspectJ 注解开发](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-31)
    - [开发步骤](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-32)
    - [注解类型](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-33)
* [**Spring 的 JDBC 模板**](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-34)
  + [使用 JDBC 模板](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-35)
    - [使用开源数据库连接池](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-36)
  + [CRUD操作](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-37)
* [**Spring的事务管理**](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-38)
  + [事务](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-39)
  + [事务管理API](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-40)
  + [事务的传播行为](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-41)
  + [实例](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-42)
  + [事务管理1: 编程式事务管理](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-43)
  + [声明式事务管理（配置实现，基于AOP思想）](https://juejin.im/post/5c2d6ecae51d4558873c7a69#heading-44)

分享