**Spring 5.x..x**

**核心：**

**IOC IoC container**

IOC只是创建对象的容器，只能创建对象，除此之外什么都干不了。

依赖注入是IOC的核心体现

事务 Events

资源 Resources

I18n i18n

验证 Validation

数据绑定 Data Binding

类型转换 Type Conversion

表达式语言 SpEL

面向切面 AOP

**ApplicationContext**

三个常用实现类：（都是用来创建Bean的）

**ClassPathXmlApplicationContext**：它可以加载类路径下的配置文件，要求配置文件必须在类路径下。不在的话，加载不了。(更常用)

**FileSystemXmlApplicationContext**：它可以加载磁盘任意路径下的配置文件(必须有访问权限）

**AnnotationConfigApplicationContext**：它是用于读取注解创建容器的。

**核心容器的两个接口引发出的问题**

**ApplicationContext**: 单例对象适用

它在构建核心容器时，创建对象采取的策略是采用**立即**加载的方式。也就是说，只要一读取完配置文件马上就创建配置文件中配置的对象。

**BeanFactory**: 多例对象适用

它在构建核心容器时，创建对象采取的策略是采用**延迟**加载的方式。也就是说，什么时候根据id获取对象了，什么时候才真正的创建对象。

**三种创建Bean的方式**

*<!--创建Bean的三种方式 -->  
<!--  
 第一种方式：使用默认构造函数创建。  
 在spring的配置文件中使用bean标签，配以id和class属性之后，且没有其他属性和标签时。  
 采用的就是默认构造函数创建bean对象，此时如果类中没有默认构造函数，则对象无法创建。  
-->*<**bean id="ccountService" class="com.yuan.service.imp.AccountServiceImpl"**/>  
  
  
  
  
*<!--  
第二种方式： 使用普通工厂中的方法创建对象（使用某个类中的方法创建对象，并存入spring容器）  
id 本Bean的唯一标识  
class 本Bean的引用类  
factory-bean 引用工厂类Bean的id  
factory-method 引用的工厂方法  
-->*<**bean id="instanceFactory" class="com.itheima.factory.InstanceFactory"**/>  
<**bean id="AccountService" factory-bean="instanceFactory" factory-method="getAccountService"**/>  
  
  
  
*<!--  
 第三种方式：使用工厂中的静态方法创建对象（使用某个类中的静态方法创建对象，并存入spring容器)  
 id 本Bean的唯一标识  
 class 本Bean的引用类  
 factory-method 本Bean的工厂方法名  
-->*<**bean id="accountService"   
 class="com.itheima.factory.StaticFactory"   
 factory-method="getAccountService"**/>

**Bean的作用范围**

<**bean id="accountService" class="com.yuan.service.imp.AccountServiceImpl" scope="prototype"**>  
</**bean**>  
*<!-- bean的作用范围调整  
 bean标签的scope属性：  
 作用：用于指定bean的作用范围  
 取值： 常用的就是单例的和多例的  
 singleton：单例的（默认值）  
 prototype：多例的  
 request：作用于web应用的请求范围  
 session：作用于web应用的会话范围  
 global-session：作用于集群环境的会话范围（全局会话范围），当不是集群环境时，它就是session  
  
  
创建Bean的三种方式 -->*

**Bean的生命周期**

单例对象

出生：当容器创建时对象出生

活着：只要容器还在，对象一直活着

死亡：容器销毁，对象消亡

总结：单例对象的生命周期和容器相同

多例对象

出生：当我们使用对象时spring框架为我们创建

活着：对象只要是在使用过程中就一直活着。

死亡：当对象长时间不用，且没有别的对象引用时，由Java的垃圾回收器回收

**依赖注入**

能注入的数据：有三类

基本类型和String

其他bean类型（在配置文件中或者注解配置过的bean）

复杂类型/集合类型

注入的方式：有三种

第一种：使用构造函数提供

第二种：使用set方法提供

第三种：使用注解提供

**所谓的注入，就是指的是将属性注入到对象中**

**1.构造注入**

*<!--构造注入 constructor-arg-->*<**bean id="accountService1" class="com.yuan.service.imp.AccountServiceImpl1"**>  
 *<!--  
 属性注入可以有3种方法  
 1.index + value 但是index从“0”开始  
 2.name + value name是属性名  
 3.name +ref ref是引用Bean的id  
 -->* <**constructor-arg index="0" value="朱葛青"**/>  
 <**constructor-arg name="money" value="365241"**/>  
 <**constructor-arg name="time" ref="now"**/>  
</**bean**>  
<**bean id="now" class="java.util.Date"**/>

**2.set()注入**

*<!--set()注入 property-->*<**bean id="accountService2" class="com.yuan.service.imp.AccountServiceImpl2"**>  
 <**property name="money" value="5000000"**/>  
 <**property name="name" value="志美"**/>  
 <**property name="time" ref="now"**/>  
</**bean**>

**3.注入集合**

*<!--set()注入之复杂类型注入-->*<**bean id="accountService3" class="com.yuan.service.imp.AccountServiceImpl3"**>  
 <**property name="mySet"**>  
 <**set**>  
 <**value**>AAA</**value**>  
 <**value**>BBB</**value**>  
 <**value**>CCC</**value**>  
 </**set**>  
 </**property**>  
  
 <**property name="myArr"**>  
 <**array**>  
 <**value**>AAA</**value**>  
 <**value**>BBB</**value**>  
 <**value**>CCC</**value**>  
 </**array**>  
 </**property**>  
  
 <**property name="myList"**>  
 <**list**>  
 <**value**>list1</**value**>  
 <**value**>list2</**value**>  
 <**value**>list3</**value**>  
 </**list**>  
 </**property**>  
  
 <**property name="myProperties"**>  
 <**props**>  
 <**prop key="testC"**>ccc</**prop**>  
 <**prop key="testD"**>ddd</**prop**>  
 </**props**>  
 </**property**>  
  
 <**property name="myMap"**>  
 <**map**>  
 <**entry key="testA" value="aaa"**></**entry**>  
 <**entry key="testB"**>  
 <**value**>BBB</**value**>  
 </**entry**>  
 </**map**>  
 </**property**>  
</**bean**>

**注解实现IOC**

**用于创建对象的**

@Component:

作用：用于把当前类对象存入spring容器中

属性：value：用于指定bean的id。当我们不写时，它的默认值是当前类名，且首字母改小写

@Controller：一般用在表现层

@Service：一般用在业务层

@Repository：一般用在持久层

以上三个注解他们的作用和属性与Component是一模一样。

他们三个是spring框架为我们提供明确的三层使用的注解，使我们的三层对象更加清晰

**用于注入数据的**

@Autowired:

作用：自动按照类型注入。只要容器中有唯一的一个bean对象类型和要注入的变量类型匹配，就可以注入成功

如果ioc容器中没有任何bean的类型和要注入的变量类型匹配，则报错。

如果Ioc容器中有多个类型匹配时：会匹配到属性名相同的。

出现位置：

可以是变量上，也可以是方法上

细节：

在使用注解注入时，set方法就不是必须的了。

@Qualifier:

作用：在按照类中注入的基础之上再按照名称注入。它在给类成员注入时不能单独使用。但是在给方法参数注入时可以

属性：

value：用于指定注入bean的id。

@Resource

作用：直接按照bean的id注入。它可以独立使用

属性：

name：用于指定bean的id。

以上三个注入都只能注入其他bean类型的数据，而基本类型和String类型无法使用上述注解实现。

另外，集合类型的注入只能通过XML来实现。

@Value

作用：用于注入基本类型和String类型的数据

属性：

value：用于指定数据的值。它可以使用spring中SpEL(也就是spring的el表达式）

SpEL的写法：${表达式}

**用于改变作用范围的**

@Scope

作用：用于指定bean的作用范围

属性：

value：指定范围的取值。常用取值：singleton prototype

**和生命周期相关 了解**

他们的作用就和在bean标签中使用init-method和destroy-methode的作用是一样的

spring 不负责多例对象的销毁方法

@PreDestroy

作用：用于指定销毁方法

@PostConstruct

作用：用于指定初始化方法

**实现CRUD**

1. **XML**

**配置不可缺少的Bean以及注入属性**

*<?***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"  
 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"**>  
  
 <**context:component-scan base-package="com.yuan"**/>  
  
 *<!--service-->* <**bean id="accountService" class="com.yuan.service.imp.AccountServiceImpl"**>  
 <**property name="accountDao" ref="accountDao"**/>  
 </**bean**>  
  
 *<!--dao-->* <**bean id="accountDao" class="com.yuan.dao.imp.AccountDaoImpl"**>  
 <**property name="runner" ref="runner"**/>  
 </**bean**>  
  
 *<!--QueryRunner-->* <**bean id="runner" class="org.apache.commons.dbutils.QueryRunner" scope="prototype"**>  
 <**constructor-arg name="ds" ref="source"**/>  
 </**bean**>  
  
  
 *<!--数据源-->* <**bean id="source" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>  
 *<!--连接数据库四大参数-->* <**property name="driverClass" value="com.mysql.jdbc.Driver"**></**property**>  
 <**property name="jdbcUrl" value="jdbc:mysql:///eesy\_mybatis"**></**property**>  
 <**property name="user" value="root"**></**property**>  
 <**property name="password" value="root"**></**property**>  
 </**bean**>  
  
  
</**beans**>

**2.ANNOCATION**

**在类上加对应注解**

**在属性上加对应注解**

**在配置文件中删掉不需要的Bean标签**

@Service  
**public class** AccountServiceImpl **implements** AccountService {  
  
 @Autowired  
 **private** AccountDao **accountDao**;  
  
 **public** List<Account> findAllAccount() {  
 **return accountDao**.findAllAccount();  
 }  
  
 **public** Account findAccountById(Integer accountId) {  
 **return accountDao**.findAccountById(accountId);  
 }  
  
 **public void** saveAccount(Account account) {  
 **accountDao**.saveAccount(account);  
 }  
  
 **public void** updateAccount(Account account) {  
 **accountDao**.updateAccount(account);  
 }  
  
 **public void** deleteAccount(Integer acccountId) {  
 **accountDao**.deleteAccount(acccountId);  
 }  
}

**注解替代配置文件**

**注解介绍**

spring中的新注解

@Configuration

作用：指定当前类是一个配置类

细节：当配置类作为AnnotationConfigApplicationContext对象创建的参数时，该注解可以不写。

@ComponentScan

作用：用于通过注解指定spring在创建容器时要扫描的包

属性：

value：它和basePackages的作用是一样的，都是用于指定创建容器时要扫描的包。

我们使用此注解就等同于在xml中配置了:

<context:component-scan base-package="com.itheima"></context:component-scan>

@Bean

作用：用于把当前方法的返回值作为bean对象存入spring的ioc容器中

属性:

name:用于指定bean的id。当不写时，默认值是当前方法的名称

细节：

当我们使用注解配置方法时，如果方法有参数，spring框架会去容器中查找有没有可用的bean对象。

查找的方式和Autowired注解的作用是一样的

@Import

作用：用于导入其他的配置类

属性：

@value：用于指定其他配置类的字节码。

当我们使用Import的注解之后，有Import注解的类就父配置类，而导入的都是子配置类

@PropertySource

作用：用于指定properties文件的位置

属性：

value：指定文件的名称和路径。

关键字：classpath，表示类路径下

**实现注解替换配置文件1**

Test类

**public class** XmlTest {  
 *//所有对象* ApplicationContext **app** = **null**;  
 *//dao对象* AccountDao **dao** = **null**;  
 *//service对象* AccountService **service** = **null**;  
  
 @Before  
 **public void** before() {  
 **app** = **new** AnnotationConfigApplicationContext(SpringConfigration.**class**);  
 **dao** = (AccountDao)**app**.getBean(**"accountDaoImpl"**);  
 **service** = (AccountService) **app**.getBean(**"accountServiceImpl"**);  
 }  
  
 @Test  
 **public void** RTest(){  
 List<Account> allAccount = **service**.findAllAccount();  
 **for** (Account account : allAccount) {  
 System.***out***.println(**"account = "** + account);  
 }  
 }

配置类

@Configuration  
@ComponentScan(**"com.yuan"**)  
 **public class** SpringConfigration {  
  
 @Bean(**"runner"**)  
 @Scope(**"prototype"**)  
 **public** QueryRunner getQueryRunner(DataSource dataSource){  
 **return new** QueryRunner(dataSource);  
 }  
  
 @Bean(**"ds"**)  
 **public** DataSource getDataSource (){  
 ComboPooledDataSource ds = **new** ComboPooledDataSource();  
 **try** {  
 ds.setDriverClass(**"com.mysql.jdbc.Driver"**);  
 ds.setJdbcUrl(**"jdbc:mysql:///eesy\_mybatis"**);  
 ds.setUser(**"root"**);  
 ds.setPassword(**"root"**);  
 } **catch** (PropertyVetoException e) {  
 e.printStackTrace();  
 }  
 **return** ds;  
 }  
}

**实现注解替换配置文件2**

Test类不变

主配置类

@Configuration  
@ComponentScan(**"com.yuan"**)  
@Import(value = JDBC.**class**)  
 **public class** SpringConfigration {  
  
}

JDBC配置类

**public class** JDBC {  
 @Bean(**"runner"**)  
 @Scope(**"prototype"**)  
 **public** QueryRunner getQueryRunner(DataSource dataSource){  
 **return new** QueryRunner(dataSource);  
 }  
  
 @Bean(**"ds"**)  
 **public** DataSource getDataSource (){  
 ComboPooledDataSource ds = **new** ComboPooledDataSource();  
 **try** {  
 ds.setDriverClass(**"com.mysql.jdbc.Driver"**);  
 ds.setJdbcUrl(**"jdbc:mysql:///eesy\_mybatis"**);  
 ds.setUser(**"root"**);  
 ds.setPassword(**"root"**);  
 } **catch** (PropertyVetoException e) {  
 e.printStackTrace();  
 }  
 **return** ds;  
 }  
}

**实现注解替换配置文件2(升级)**

主配置文件 （增加了配置文件标签）

@Configuration  
@ComponentScan(**"com.yuan"**)  
@PropertySource(**"classpath:JDBC.properties"**)  
@Import(value = JDBC.**class**)  
 **public class** SpringConfigration {  
  
  
}

JDBC配置文件

**public class** JDBC {  
 @Value(**"${jdbc.driver}"**)  
 **private** String **driver**;  
  
 @Value(**"${jdbc.url}"**)  
 **private** String **url**;  
  
 @Value(**"${jdbc.username}"**)  
 **private** String **username**;  
  
 @Value(**"${jdbc.password}"**)  
 **private** String **password**;  
  
 @Bean(**"runner"**)  
 @Scope(**"prototype"**)  
 **public** QueryRunner getQueryRunner(DataSource dataSource){  
 **return new** QueryRunner(dataSource);  
 }  
  
 @Bean(**"ds"**)  
 **public** DataSource getDataSource (){  
 ComboPooledDataSource ds = **new** ComboPooledDataSource();  
 **try** {  
 ds.setDriverClass(**driver**);  
 ds.setJdbcUrl(**url**);  
 ds.setUser(**username**);  
 ds.setPassword(**password**);  
 } **catch** (PropertyVetoException e) {  
 e.printStackTrace();  
 }  
 **return** ds;  
 }  
}

**Spring整合Junit**

1. 添加 Spring-test
2. 使用Junit提供的一个注解把原有的main方法替换了，替换成spring提供的@Runwith
3. 3、告知spring的运行器，spring和ioc创建是基于xml还是注解的，并且说明位置

@ContextConfiguration

ocations：指定xml文件的位置，加上classpath关键字，表示在类路径下

classes：指定注解类所在地位置

@ContextConfiguration(classes = SpringConfiguration.**class**)  
**public class** AccountServiceTest {  
  
 @Autowired  
 **private** IAccountService **as** = **null**;  
  
  
 @Test  
 **public void** testFindAll() {  
 *//3.执行方法* List<Account> accounts = **as**.findAllAccount();  
 **for** (Account account : accounts) {  
 System.***out***.println(account);  
 }  
 }  
}

**AOP名词解释**

**Joinpoint(连接点)**  可以增强的方法**【业务层的方法】**

**Pointcut(切入点)**  **要/已增强的方法**

**Advice(通知/增强)**: **就是增强的内容** 通知的类型：前置通知,后置通知,异常通知,最终通知,环绕通知。

Introduction(引介): 引介是一种特殊的通知在不修改类代码的前提下, Introduction 可以在运行期为类动态地添加一些方 法或 Field。 Target

Target(目标对象): 代理的目标对象。 **【业务层对象】**

Weaving(织入): 是指把增强应用到目标对象来创建新的代理对象的过程。

spring 采用动态代理织入，而 AspectJ 采用编译期织入和类装载期织入**【增强的过程】**

**Proxy（代理）** 一个类被 AOP 织入增强后，就产生一个结果代理类**【Poxy.newPoxyIncetence() 返回的对象】**

**Aspect(切面):** 是切入点和通知（引介）的结合。**【被增强的方法和增强的内容的结合】**

**AOP**

**pom.xml**

*<!--解析AOP切入点-->* <**dependency**>  
 <**groupId**>org.aspectj</**groupId**>  
 <**artifactId**>aspectjweaver</**artifactId**>  
 <**version**>1.8.7</**version**>  
 </**dependency**>  
</**dependencies**>

**Application.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"  
 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"**>  
  
 *<!--配置spring的IOC把service对象配置进来-->*<**bean id="accountService" class="com.yuan.service.imp.AccountServiceImpl"**></**bean**>  
 *<!-- 配置Logger类 -->*<**bean id="logger" class="com.yuan.util.Logger"**></**bean**>  
 *<!--配置AOP-->* <**aop:config**>  
 *<!--配置切面 -->* <**aop:aspect id="logAdvice" ref="logger"**>  
 *<!-- 配置通知的类型，并且建立通知方法和切入点方法的关联-->* <**aop:before method="printLog" pointcut="execution(\* com.yuan.service.imp.\*.\*(..))"**></**aop:before**>  
 </**aop:aspect**>  
 </**aop:config**>  
</**beans**>

**切入点表达式写法**

**关键字：execution(表达式)**

**表达式：**

访问修饰符 返回值 包名.包名.包名...类名.方法名(参数列表)

标准的表达式写法：

访问修饰符可以省略

**public void com.yuan.service.imp.AccountServiceImpl.saveAccount()**

返回值可以使用通配符，表示任意返回值

**\* com.yuan.service.imp.AccountServiceImpl.saveAccount()**

包名可以使用通配符，表示任意包。但是有几级包，就需要写几个\*.

**\* \*.\*.\*.\*.AccountServiceImpl.saveAccount())**

包名可以使用..表示当前包及其子包

**\* \*..AccountServiceImpl.saveAccount()**

类名和方法名都可以使用\*来实现通配

**\* \*..\*.\*()**

参数列表：

可以直接写数据类型：

基本类型直接写名称 int

引用类型写包名.类名的方式 java.lang.String

可以使用通配符表示任意类型，但是必须有参数

可以使用..表示有无参数均可，有参数可以是任意类型

全通配写法：

**\* \*..\*.\*()**

实际开发中切入点表达式的通常写法：

切到业务层实现类下的所有方法

**\* com.yuan.service.imp.\*.\*(..)**

**切面配置(详细)**

**xml**

*<!--配置AOP-->*<**aop:config**>  
 <**aop:pointcut id="pc1" expression="execution(\* com.yuan.service.imp.\*.\*(..))"**/>  
 *<!--配置切面-->* <**aop:aspect id="logAdvice" ref="logger"**>  
  
 *<!--配置前置通知：在切入点方法执行之前执行-->* <**aop:before method="beforePrintLog" pointcut-ref="pc1"**/>  
  
 *<!--配置后置通知 在切入点方法正常执行之后值。它和异常通知永远只能执行一个-->* <**aop:after-returning method="afterReturningPrintLog" pointcut-ref="pc1"**/>  
  
 *<!--配置异常通知：在切入点方法执行产生异常之后执行。它和后置通知永远只能执行一个 -->* <**aop:after-throwing method="afterThrowingPrintLog" pointcut-ref="pc1"**/>  
  
 *<!--配置最终通知：无论切入点方法是否正常执行它都会在其后面执行 -->* <**aop:after method="afterPrintLog" pointcut-ref="pc1"**/>  
  
 *<!-- 配置环绕通知 详细的注释请看Logger类中-->* <**aop:around method="aroundPringLog" pointcut-ref="pc1"**></**aop:around**>  
 </**aop:aspect**>  
</**aop:config**>

**java**

*/\*\*  
 \* 用于记录日志的工具类，它里面提供了公共的代码  
 \*/***public class** Logger {  
  
 */\*\*  
 \* 前置通知  
 \*/* **public void** beforePrintLog(){  
 System.***out***.println(**"前置通知Logger类中的beforePrintLog方法开始记录日志了。。。"**);  
 }  
  
 */\*\*  
 \* 后置通知  
 \*/* **public void** afterReturningPrintLog(){  
 System.***out***.println(**"后置通知Logger类中的afterReturningPrintLog方法开始记录日志了。。。"**);  
 }  
 */\*\*  
 \* 异常通知  
 \*/* **public void** afterThrowingPrintLog(){  
 System.***out***.println(**"异常通知Logger类中的afterThrowingPrintLog方法开始记录日志了。。。"**);  
 }  
  
 */\*\*  
 \* 最终通知  
 \*/* **public void** afterPrintLog(){  
 System.***out***.println(**"最终通知Logger类中的afterPrintLog方法开始记录日志了。。。"**);  
 }  
  
 */\*\*  
 \* 环绕通知  
 \* 问题：  
 \* 当我们配置了环绕通知之后，切入点方法没有执行，而通知方法执行了。  
 \* 分析：  
 \* 通过对比动态代理中的环绕通知代码，发现动态代理的环绕通知有明确的切入点方法调用，而我们的代码中没有。  
 \* 解决：  
 \* Spring框架为我们提供了一个接口：ProceedingJoinPoint。该接口有一个方法proceed()，此方法就相当于明确调用切入点方法。  
 \* 该接口可以作为环绕通知的方法参数，在程序执行时，spring框架会为我们提供该接口的实现类供我们使用。  
 \*  
 \* spring中的环绕通知：  
 \* 它是spring框架为我们提供的一种可以在代码中手动控制增强方法何时执行的方式。  
 \*/* **public** Object aroundPringLog(ProceedingJoinPoint pjp){  
 Object rtValue = **null**;  
 **try**{  
 *//得到方法执行所需的参数* Object[] args = pjp.getArgs();  
  
 System.***out***.println(**"Logger类中的aroundPringLog方法开始记录日志了。。。前置"**);  
  
 *//明确调用业务层方法（切入点方法）* rtValue = pjp.proceed(args);  
  
 System.***out***.println(**"Logger类中的aroundPringLog方法开始记录日志了。。。后置"**);  
  
 **return** rtValue;  
 }**catch** (Throwable t){  
 System.***out***.println(**"Logger类中的aroundPringLog方法开始记录日志了。。。异常"**);  
 **throw new** RuntimeException(t);  
 }**finally** {  
 System.***out***.println(**"Logger类中的aroundPringLog方法开始记录日志了。。。最终"**);  
 }  
 }  
}

**注解实现AOP步骤**

**第一步【配置pom】**

*<!--spring核心框架-->*<**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-context</**artifactId**>  
 <**version**>5.0.2.RELEASE</**version**>  
 </**dependency**>  
  
 *<!--解析AOP切入点-->* <**dependency**>  
 <**groupId**>org.aspectj</**groupId**>  
 <**artifactId**>aspectjweaver</**artifactId**>  
 <**version**>1.8.7</**version**>  
 </**dependency**>  
</**dependencies**>

**第二步【配置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spring创建容器时要扫描的包-->* <**context:component-scan base-package="com.yuan"**></**context:component-scan**>  
  
 *<!-- 配置spring开启注解AOP的支持 -->* <**aop:aspectj-autoproxy**></**aop:aspectj-autoproxy**>  
</**beans**>

**第三步【添加IOC注解】**

service层添加 @org.springframework.stereotype.Service(**"accountService"**)

切面层添加 @org.springframework.stereotype.Component(**"logger"**)

标识切面层 @org.aspectj.lang.annotation.Aspect

**第四步【添加AOP注解】**

@Pointcut(**"execution(\* com.yuan.service.imp.\*.\*(..))"**)  
**private void** pt1(){}

在一般通知上添加一般通知的注解

*/\*\*  
 \* 前置通知  
 \*/*@Before(**"pt1()"**)  
**public void** beforePrintLog(){  
 System.***out***.println(**"前置通知Logger类中的beforePrintLog方法开始记录日志了。。。"**);  
}  
  
*/\*\*  
 \* 后置通知  
 \*/*@AfterReturning(**"pt1()"**)  
**public void** afterReturningPrintLog(){  
 System.***out***.println(**"后置通知Logger类中的afterReturningPrintLog方法开始记录日志了。。。"**);  
}  
*/\*\*  
 \* 异常通知  
 \*/*@AfterThrowing(**"pt1()"**)  
**public void** afterThrowingPrintLog(){  
 System.***out***.println(**"异常通知Logger类中的afterThrowingPrintLog方法开始记录日志了。。。"**);  
}  
  
*/\*\*  
 \* 最终通知  
 \*/*@After(**"pt1()"**)  
**public void** afterPrintLog(){  
 System.***out***.println(**"最终通知Logger类中的afterPrintLog方法开始记录日志了。。。"**);  
}

在环绕通知上添加环绕注解

@Around(**"pt1()"**)  
**public** Object aroundPringLog(ProceedingJoinPoint pjp){  
 Object rtValue = **null**;  
 **try**{  
 *//得到方法执行所需的参数* Object[] args = pjp.getArgs();  
  
 System.***out***.println(**"Logger类中的aroundPringLog方法开始记录日志了。。。前置"**);  
  
 *//明确调用业务层方法（切入点方法）* rtValue = pjp.proceed(args);  
  
 System.***out***.println(**"Logger类中的aroundPringLog方法开始记录日志了。。。后置"**);  
  
 **return** rtValue;  
 }**catch** (Throwable t){  
 System.***out***.println(**"Logger类中的aroundPringLog方法开始记录日志了。。。异常"**);  
 **throw new** RuntimeException(t);  
 }**finally** {  
 System.***out***.println(**"Logger类中的aroundPringLog方法开始记录日志了。。。最终"**);  
 }  
}

结束

**SpringJDBC模板使用**

**public class** JDBCTest {  
 **public static void** main(String[] args) {  
 ClassPathXmlApplicationContext app = **new** ClassPathXmlApplicationContext(**"bean.xml"**);  
 JdbcTemplate jt = (JdbcTemplate) app.getBean(**"jdbcTemlate"**,JdbcTemplate.**class**);  
*// 增* jt.update(**"INSERT INTO ACCOUNT (name,money) VALUES (?,?)"**,**"eee"**,3333f);  
*// 删* jt.update(**"DELETE FROM ACCOUNT WHERE ID = ?"**,5);  
*// 改* jt.update(**"UPDATE ACCOUNT SET name = ? , money = ? where id = ?"**,**"改ddd"**,555f,4);  
*// 查（多行）* List<Account> accounts = jt.query(**"SELECT** *\** **FROM ACCOUNT WHERE MONEY > ?"**, **new** BeanPropertyRowMapper<Account>(Account.**class**), 200f);  
*// 查（一行）* List<Account> account = jt.query(**"SELECT** *\** **FROM ACCOUNT WHERE ID = ?"**, **new** BeanPropertyRowMapper<Account>(Account.**class**), 1);  
*// 查（聚合函数）* Long count = jt.queryForObject(**"SELECT** *COUNT***(***\****) FROM ACCOUNT WHERE MONEY > ?"**, Long.**class**, 5);  
 }  
}  
*/\*\*  
 \* /\*\*  
 \* 定义Account的封装策略  
 \* 用来替换上面的 BeanPropertyRowMapper  
 \*/***class** AccountRowMapper **implements** RowMapper<Account>{  
  
 **public** Account mapRow(ResultSet rs, **int** i) **throws** SQLException {  
 Account accounts = **new** Account();  
 accounts.setId(rs.getInt(**"id"**));  
 accounts.setName(rs.getString(**"name"**));  
 accounts.setMoney(rs.getFloat(**"money"**));  
 **return** accounts;  
 }  
}

**Spring事务**

**基于XML配置文件**

1. **配置事务管理器**
2. **事务通知**
3. **配置切入点表达式**
4. **配置通知和切入点关系**
5. **配置事务属性**
   1. 配置事务的属性
   2. isolation：用于指定事务的隔离级别。默认值是DEFAULT，表示使用数据库的默认隔离级别。
   3. propagation：用于指定事务的传播行为。默认值是REQUIRED，表示一定会有事务，增删改的选择。查询方法可以选择SUPPORTS。
   4. read-only：用于指定事务是否只读。只有查询方法才能设置为true。默认值是false，表示读写。
   5. timeout：用于指定事务的超时时间，默认值是-1，表示永不超时。如果指定了数值，以秒为单位。
   6. rollback-for：用于指定一个异常，当产生该异常时，事务回滚，产生其他异常时，事务不回滚。没有默认值。表示任何异常都回滚。
   7. no-rollback-for：用于指定一个异常，当产生该异常时，事务不回滚，产生其他异常时事务回滚。没有默认值。表示任何异常都回滚。

*<?***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: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/tx  
 http://www.springframework.org/schema/tx/spring-tx.xsd  
 http://www.springframework.org/schema/aop  
 http://www.springframework.org/schema/aop/spring-aop.xsd"**>  
  
*<!-- 配置业务层-->* <**bean id="accountService" class="com.yuan.service.impl.AccountServiceImpl"**>  
 <**property name="accountDao" ref="accountDao"**/>  
 </**bean**>  
  
 *<!-- 配置账户的持久层-->* <**bean id="accountDao" class="com.yuan.dao.impl.AccountDaoImpl"**>  
 <**property name="dataSource" ref="dataSource"**/>  
 </**bean**>  
  
 *<!-- 配置数据源-->* <**bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"**>  
 <**property name="driverClassName" value="com.mysql.jdbc.Driver"**></**property**>  
 <**property name="url" value="jdbc:mysql://localhost:3306/spring"**></**property**>  
 <**property name="username" value="root"**></**property**>  
 <**property name="password" value="root"**></**property**>  
 </**bean**>  
  
 *<!--1. 配置事务管理器 -->* <**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  
 <**property name="dataSource" ref="dataSource"**></**property**>  
 </**bean**>  
  
 *<!--2.配置事务通知-->* <**tx:advice id="txAdvice" transaction-manager="transactionManager"**>  
 *<!--5.配置事务属性-->* <**tx:attributes**>  
 <**tx:method name="\*" propagation="REQUIRED" read-only="false"**/>  
 </**tx:attributes**>  
 </**tx:advice**>  
  
 <**aop:config**>  
 *<!--3、配置AOP中的通用切入点表达式-->* <**aop:pointcut id="accountPT" expression="execution(\* com.yuan.service.impl.\*.\*(..))"**/>  
 *<!--4.配置通知和切入点关系-->* <**aop:advisor advice-ref="txAdvice" pointcut-ref="accountPT"**/>  
 </**aop:config**>  
</**beans**>

**基于注解**

1. **xml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"  
 xmlns:tx="http://www.springframework.org/schema/tx"  
 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/tx  
 http://www.springframework.org/schema/tx/spring-tx.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spring创建容器时要扫描的包-->* <**context:component-scan base-package="com.yuan"**/>  
 *<!-- 配置JdbcTemplate-->* <**bean class="org.springframework.jdbc.core.JdbcTemplate" id="jdbcTemplate"**>  
 <**property name="dataSource" ref="dataSource"**/>  
 </**bean**>  
  
 *<!-- 配置数据源-->* <**bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"**>  
 <**property name="driverClassName" value="com.mysql.jdbc.Driver"**></**property**>  
 <**property name="url" value="jdbc:mysql://localhost:3306/spring"**></**property**>  
 <**property name="username" value="root"**></**property**>  
 <**property name="password" value="root"**></**property**>  
 </**bean**>  
  
 *<!-- spring中基于注解 的声明式事务控制配置步骤  
 1、配置事务管理器  
 2、开启spring对注解事务的支持  
 3、在需要事务支持的地方使用@Transactional注解  
 -->  
 <!-- 配置事务管理器 -->* <**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  
 <**property name="dataSource" ref="dataSource"**/>  
 </**bean**>  
  
 *<!-- 开启spring对注解事务的支持-->* <**tx:annotation-driven transaction-manager="transactionManager"**/>  
</**beans**>

1. **Service层**

@Service(**"accountService"**)  
@Transactional(propagation= Propagation.***SUPPORTS***)*//只读型事务的配置***public class** AccountServiceImpl **implements** IAccountService {  
  
 @Autowired  
 **private** IAccountDao **accountDao**;  
  
 **public** Account findAccountById(Integer accountId) {  
 **return accountDao**.findAccountById(accountId);  
  
 }  
  
 */\*\*  
 \*  
 \** ***@param sourceName*** *转成账户名称  
 \** ***@param targetName*** *转入账户名称  
 \** ***@param money*** *转账金额  
 \** ***@Transactional*** *需要的是读写型事务配置  
 \*/  
 //需要的是读写型事务配置* **public void** transfer(String sourceName, String targetName, Float money) {  
 System.***out***.println(**"transfer...."**);  
 *//2.1根据名称查询转出账户* Account source = **accountDao**.findAccountByName(sourceName);  
 *//2.2根据名称查询转入账户* Account target = **accountDao**.findAccountByName(targetName);  
 *//2.3转出账户减钱* source.setMoney(source.getMoney()-money);  
 *//2.4转入账户加钱* target.setMoney(target.getMoney()+money);  
 *//2.5更新转出账户* **accountDao**.updateAccount(source);  
  
 **int** a = 0/0;  
  
 *//2.6更新转入账户* **accountDao**.updateAccount(target);  
 }  
}

1. **Dao层**

*/\*\*  
 \* 账户的持久层实现类  
 \*/*@Repository(**"accountDao"**)  
**public class** AccountDaoImpl **implements** IAccountDao {  
  
 @Autowired  
 **private** JdbcTemplate **jdbcTemplate**;  
  
 **public** Account findAccountById(Integer accountId) {  
 List<Account> accounts = **jdbcTemplate**.query(**"select** *\** **from account where id = ?"**,**new** BeanPropertyRowMapper<Account>(Account.**class**),accountId);  
 **return** accounts.isEmpty()?**null**:accounts.get(0);  
 }  
  
 **public** Account findAccountByName(String accountName) {  
 List<Account> accounts = **jdbcTemplate**.query(**"select** *\** **from account where name = ?"**,**new** BeanPropertyRowMapper<Account>(Account.**class**),accountName);  
 **if**(accounts.isEmpty()){  
 **return null**;  
 }  
 **if**(accounts.size()>1){  
 **throw new** RuntimeException(**"结果集不唯一"**);  
 }  
 **return** accounts.get(0);  
 }  
  
 **public void** updateAccount(Account account) {  
 **jdbcTemplate**.update(**"update account set name=?,money=? where id=?"**,account.getName(),account.getMoney(),account.getId());  
 }  
}

**JK:**

Bean容器工厂模式。

Spring5 要求 JDK8及以上 TomCat8.5及以上

Spring整合Junit需要4.1.2及以上版本。

动态

一Poxy

**public class** Client {  
  
 **public static void** main(String[] args) {  
 **final** Producer producer = **new** Producer();  
  
 */\*\*  
 \* 动态代理：  
 \* 特点：字节码随用随创建，随用随加载  
 \* 作用：不修改源码的基础上对方法增强  
 \* 分类：  
 \* 基于接口的动态代理  
 \* 基于子类的动态代理  
 \* 基于接口的动态代理：  
 \* 涉及的类：Proxy  
 \* 提供者：JDK官方  
 \* 如何创建代理对象：  
 \* 使用Proxy类中的newProxyInstance方法  
 \* 创建代理对象的要求：  
 \* 被代理类最少实现一个接口，如果没有则不能使用  
 \* newProxyInstance方法的参数：  
 \* ClassLoader：类加载器  
 \* 它是用于加载代理对象字节码的。和被代理对象使用相同的类加载器。固定写法。  
 \* Class[]：字节码数组  
 \* 它是用于让代理对象和被代理对象有相同方法。固定写法。  
 \* InvocationHandler：用于提供增强的代码  
 \* 它是让我们写如何代理。我们一般都是些一个该接口的实现类，通常情况下都是匿名内部类，但不是必须的。  
 \* 此接口的实现类都是谁用谁写。  
 \*/* IProducer proxyProducer = (IProducer) Proxy.*newProxyInstance*(producer.getClass().getClassLoader(),  
 producer.getClass().getInterfaces(),  
 **new** InvocationHandler() {  
 */\*\*  
 \* 作用：执行被代理对象的任何接口方法都会经过该方法  
 \* 方法参数的含义  
 \** ***@param proxy*** *代理对象的引用  
 \** ***@param method*** *当前执行的方法  
 \** ***@param args*** *当前执行方法所需的参数  
 \** ***@return*** *和被代理对象方法有相同的返回值  
 \** ***@throws*** *Throwable  
 \*/* **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  
 *//提供增强的代码* Object returnValue = **null**;  
  
 *//1.获取方法执行的参数* Float money = (Float)args[0];  
  
 *//2.判断当前方法是不是销售* **if**(**"saleProduct"**.equals(method.getName())) {  
 returnValue = method.invoke(producer, money\*0.8f);  
 }  
 **return** returnValue;  
 }  
 });  
 proxyProducer.saleProduct(10000f);  
 }  
}

二Cglib

**public class** Client {  
  
 **public static void** main(String[] args) {  
 **final** Producer producer = **new** Producer();  
  
 */\*\*  
 \* Cglib动态代理：  
 \* 特点：字节码随用随创建，随用随加载  
 \* 作用：不修改源码的基础上对方法增强  
 \* 分类：  
 \* 基于接口的动态代理  
 \* 基于子类的动态代理  
 \* 基于子类的动态代理：  
 \* 涉及的类：Enhancer  
 \* 提供者：第三方cglib库  
 \* 如何创建代理对象：  
 \* 使用Enhancer类中的create方法  
 \* 创建代理对象的要求：  
 \* 被代理类不能是最终类  
 \* create方法的参数：  
 \* Class：字节码  
 \* 它是用于指定被代理对象的字节码。  
 \*  
 \* Callback：用于提供增强的代码  
 \* 它是让我们写如何代理。我们一般都是些一个该接口的实现类，通常情况下都是匿名内部类，但不是必须的。  
 \* 此接口的实现类都是谁用谁写。  
 \* 我们一般写的都是该接口的子接口实现类：MethodInterceptor  
 \*/* Producer cglibProducer = (Producer) Enhancer.*create*(producer.getClass(), **new** MethodInterceptor() {  
 */\*\*  
 \* 执行调被调用的任何方法都会经过该方法  
 \** ***@param proxy*** *\** ***@param method*** *\** ***@param args*** *\* 以上三个参数和基于接口的动态代理中invoke方法的参数是一样的  
 \** ***@param methodProxy*** *：当前执行方法的代理对象  
 \** ***@return*** *\** ***@throws*** *Throwable  
 \*/* **public** Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) **throws** Throwable {  
 *//提供增强的代码* Object returnValue = **null**;  
  
 *//1.获取方法执行的参数* Float money = (Float) args[0];  
 *//2.判断当前方法是不是销售* **if** (**"saleProduct"**.equals(method.getName())) {  
 returnValue = method.invoke(producer, money \* 0.8f);  
 }  
 **return** returnValue;  
 }  
 });  
  
 cglibProducer.saleProduct(12000f);  
 cglibProducer.afterService(120f);  
  
  
 }  
}