# 环境搭建

使用intellij idea新建maven项目：new>project>maven>next>....

在pom.xml 文件中导入依赖

<dependencies>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>5.2.4.RELEASE</version>  
 </dependency>  
</dependencies>

# IOC

控制反转（Inversion of Control，缩写IoC）是指把创建对象的权利交给框架。简单来说就是，以前我们创建对象的时候，是通过 new 关键字来创建的，这种方式是主动的；而通过 IoC，我们把创建对象的任务交给了它，这种方式是被动的，我们只需要列出一个清单，表明需要什么对象，IoC 就会帮我们创建好对象，当我们需要使用时，只需要向 IoC 容器获取即可。

示例：

创建一个类

package com.chq.test;  
  
public class HelloWorld {  
 private String message;  
 public void setMessage(String message){  
 this.message = message;  
 }  
 public void getMessage(){  
 System.*out*.println("Your Message : " + message);  
 }  
}

在resources新建配置文件bean.xml （该文件需要放在类路径下，名称是任意的，但是不能是中文）

<?xml version="1.0" encoding="UTF-8"?>  
<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">  
 <!-- 配置serive -->  
 <bean id="helloWorld" class="com.chq.test.HelloWorld">  
 <property name="message" value="Hello World!"/>  
 </bean>  
</beans>

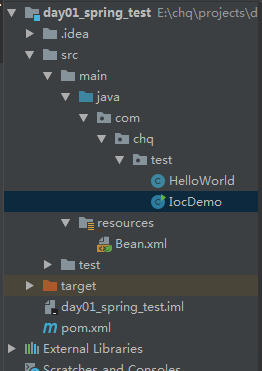
测试代码及运行结果如下：

package com.chq.test;  
  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class IocDemo {  
 public static void main(String[] args) {  
 ApplicationContext ac = new ClassPathXmlApplicationContext("Bean.xml");  
 HelloWorld helloWorld = (HelloWorld) ac.getBean("helloWorld");  
  
 helloWorld.getMessage();  
 }  
}

Your Message : Hello World!

Process finished with exit code 0

目录：



# 依赖注入

依赖注入（Dependency Injection，缩写 DI）是 Spring 框架核心 IoC 的具体实现。当我们把对象的创建交给了 Spring 后，虽然降低了对象间的依赖关系，但是并不会消除。比方说：业务层仍然会调用持久层的方法。那么业务层和持久层的依赖关系，也要靠 Spring 来进行维护。也就是说，当 Spring 在创建业务层对象的时候，会帮我们把持久层对象传入到业务层中，无需我们自己解决。

## 构造注入

通过类中的构造方法给成员变量赋值。这个赋值的操作并不是我们做的，而是通过配置的方式，让 Spring 框架来为我们注入

还是那个类，定义构造函数。

package com.chq.test;  
  
public class HelloWorld {  
 private String message;  
  
 public HelloWorld(String message) {  
 this.message = message;  
 }  
  
// public void setMessage(String message) {  
// this.message = message;  
// }  
  
 public void getMessage() {  
 System.*out*.println("Your Message : " + message);  
 }  
}

将bean.xml中的控制反转修改入下

<bean id="helloWorld" class="com.chq.test.HelloWorld">  
 <constructor-arg name="message" value="Hello World!"></constructor-arg>  
 <!--<property name="message" value="Hello World!"/>-->  
</bean>

这样运行测试类值就已经注入进去了。

## 设值注入

构造注入里注释的就是，在前面控制反转已经实践过了。

# 基于注解的IOC配置

如果要使用注解的话，首先需要在配置文件中导入新的命名空间，并且指定要扫描的包。

<?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">  
  
 <!--告知spring在创建容器时要扫描的包，配置所需要的标签不是在beans的约束中，而是一个名称为context名称空间和约束中-->  
 <context:component-scan base-package="com.chq.test"/>  
</beans>

## 用于创建对象的注解

### @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 的唯一标识 （Id）。默认值是当前类名，且首字母小写

不需要在bean.xml上配置bean标签，只要在class上面添加@Component即可。

@Component  
public class HelloWorld {

还有三个衍生注解也是用于创建对象，功能与 @Component 一样，这三个注解是 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。

|  |  |
| --- | --- |
| @Controller | 表现层 |
| @Service | 业务层 |
| @Repository | 持久层 |

上面class即使使用这三个注解也是可以的，只是让我们的三层对象更加清晰。

## 用于注入数据的注解

### @Autowired

作用:自动按照类型注入(By Type)

容器首先用变量的类型进行查找，如果容器中有唯一一个 Bean 的类型匹配，那么就直接注入；如果容器中没有任何 Bean 的类型与要注入的变量类型匹配，那么就报错；如果容器中有多个 Bean 的类型匹配，那么容器就会以变量的名称作为 Bean 的 id 进行查找，找到匹配的则注入，未找到则报错。

出现位置:变量或者方法上

细节:当使用注解来注入属性时，set 方法可以省略

@service("accountService")  
public class AccountServiceImpl implements IAccountService {  
 @Autowired  
 private IAccountDao accountDao;  
 // ....  
}

### @Qualifier

作用：在按照类型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，必须和 @Autowired 一起使用。但是在给方法参数注入时可以单独使用。

属性：value ： 用于指定注入 Bean 的唯一标识 （Id）

@service("accountService")  
public class AccountServiceImpl implements IAccountService {  
 @Autowired  
 @Qualifier("accountDao")  
 private IAccountDao accountDao;  
 // ...  
}

### @Resource

作用：自动按照名称注入(By Name)，直接按照 Bean 的 Id 进行注入，它可以独立使用。

属性：name ： 指定注入 Bean 的 id

@service("accountService")  
public class AccountServiceImpl implements IAccountService {  
 // @Autowired  
 // @Qualifier("accountDao")  
 @Resource(name = "accountDao")  
 private IAccountDao accountDao;  
 // ...  
}

### @Value

作用：注入基于类型和 String 类型数据

属性：value ： 用于指定数据的值。它可以使用 Spring 中的 SpEL( Spring 的 el 表达式，写法：${表达式}）

## 用于改变作用范围的

### @Scope

作用：指定 Bean 的作用范围

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

## 和生命周期有关

### @PreDestroy

作用：用于指定销毁方法

### PostConstruct

作用：用于指定初始化方法

# 案例：实现账户的 CRUD 操作

## XML 方式

导入相关依赖

<dependencies>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>5.0.2.RELEASE</version>  
 </dependency>  
 <!-- https://mvnrepository.com/artifact/commons-dbutils/commons-dbutils -->  
 <dependency>  
 <groupId>commons-dbutils</groupId>  
 <artifactId>commons-dbutils</artifactId>  
 <version>1.4</version>  
 </dependency>  
  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.6</version>  
 </dependency>  
  
 <dependency>  
 <groupId>c3p0</groupId>  
 <artifactId>c3p0</artifactId>  
 <version>0.9.1.2</version>  
 </dependency>  
  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
 </dependency>  
</dependencies>

创建数据库表并编写对应实体类

create table account(

id int primary key auto\_increment,

name varchar(40),

balance float

)character set utf8 collate utf8\_general\_ci;

insert into account(name,balance) values('aaa',1000);

insert into account(name,balance) values('bbb',1000);

insert into account(name,balance) values('ccc',1000);

创建实体类

package com.chq.spring.bean;  
  
import java.io.Serializable;  
  
public class Account implements Serializable {  
 private Integer id;  
 private String name;  
 private Double balance;  
  
 public Integer getId() {  
 return id;  
 }  
  
 public void setId(Integer id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public Double getBalance() {  
 return balance;  
 }  
  
 public void setBalance(Double balance) {  
 this.balance = balance;  
 }  
  
 @Override  
 public String toString() {  
 return "Account{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 ", balance=" + balance +  
 '}';  
 }  
}

编写业务层接口和实现类

package com.chq.spring.service;  
  
import com.chq.spring.bean.Account;  
  
import java.util.List;  
  
public interface IAccountService {  
 */\*\*  
 \* 查询所有账户  
 \** ***@return*** *\*/* List<Account> listAllAccounts();  
  
 */\*\*  
 \* 根据id查询  
 \** ***@param*** *accountId  
 \** ***@return*** *\*/* Account getAccountById(Integer accountId);  
  
 */\*\*  
 \* 保存  
 \** ***@param*** *account  
 \** ***@return*** *\*/* int saveAccount(Account account);  
  
 */\*\*  
 \* 更新  
 \** ***@param*** *account  
 \** ***@return*** *\*/* int updateAccount(Account account);  
  
 */\*\*  
 \* 删除  
 \** ***@param*** *accountId  
 \** ***@return*** *\*/* int removeAccount(Integer accountId);  
}

package com.chq.spring.service;  
  
import com.chq.spring.bean.Account;  
import com.chq.spring.dao.IAccountDao;  
  
import java.util.List;  
  
public class AccountServiceImpl implements IAccountService {  
 private IAccountDao accountDao;  
  
 public void setAccountDao(IAccountDao accountDao) {  
 this.accountDao = accountDao;  
 }  
  
 @Override  
 public List<Account> listAllAccounts() {  
 return accountDao.listAllAccounts();  
 }  
  
 @Override  
 public Account getAccountById(Integer accountId) {  
 return accountDao.getAccountById(accountId);  
 }  
  
 @Override  
 public int saveAccount(Account account) {  
 return accountDao.saveAccount(account);  
 }  
  
 @Override  
 public int updateAccount(Account account) {  
 return accountDao.updateAccount(account);  
 }  
  
 @Override  
 public int removeAccount(Integer accountId) {  
 return accountDao.removeAccount(accountId);  
 }  
}

编写持久层接口和实现类

package com.chq.spring.dao;  
  
import com.chq.spring.bean.Account;  
  
import java.util.List;  
  
public interface IAccountDao {  
 */\*\*  
 \* 查询所有账户  
 \*  
 \** ***@return*** *\*/* List<Account> listAllAccounts();  
  
 */\*\*  
 \* 根据id查询  
 \*  
 \** ***@param*** *accountId  
 \** ***@return*** *\*/* Account getAccountById(Integer accountId);  
  
 */\*\*  
 \* 保存  
 \*  
 \** ***@param*** *account  
 \** ***@return*** *\*/* int saveAccount(Account account);  
  
 */\*\*  
 \* 更新  
 \*  
 \** ***@param*** *account  
 \** ***@return*** *\*/* int updateAccount(Account account);  
  
 */\*\*  
 \* 删除  
 \*  
 \** ***@param*** *accountId  
 \** ***@return*** *\*/* int removeAccount(Integer accountId);  
}

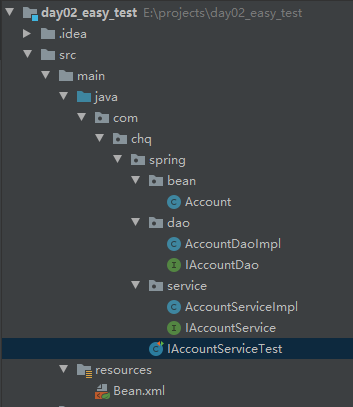
package com.chq.spring.dao;  
  
import com.chq.spring.bean.Account;  
import org.apache.commons.dbutils.QueryRunner;  
import org.apache.commons.dbutils.handlers.BeanHandler;  
import org.apache.commons.dbutils.handlers.BeanListHandler;  
  
import java.sql.SQLException;  
import java.util.List;  
  
public class AccountDaoImpl implements IAccountDao {  
 private QueryRunner runner;  
  
 public void setRunner(QueryRunner runner) {  
 this.runner = runner;  
 }  
  
 @Override  
 public List<Account> listAllAccounts() {  
 try {  
 return runner.query("SELECT \* FROM account", new BeanListHandler<Account>(Account.class));  
 } catch (SQLException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
 @Override  
 public Account getAccountById(Integer accountId) {  
 try {  
 return runner.query("SELECT \* FROM account WHERE id = ?", accountId, new BeanHandler<Account>(Account.class));  
 } catch (SQLException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
 @Override  
 public int saveAccount(Account account) {  
 try {  
 return runner.update("INSERT INTO account(name,balance) VALUES(?,?)", account.getName(), account.getBalance());  
 } catch (SQLException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
 @Override  
 public int updateAccount(Account account) {  
 try {  
 return runner.update("UPDATE account SET name = ?,balance = ? WHERE id = ?", account.getName(), account.getBalance(), account.getId());  
 } catch (SQLException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
 @Override  
 public int removeAccount(Integer accountId) {  
 try {  
 return runner.update("DELETE FROM account WHERE id = ?", accountId);  
 } catch (SQLException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}

编写 Spring 配置文件bean.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"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd">  
 <!-- 配置service -->  
 <bean id="accountService" class="com.chq.spring.service.AccountServiceImpl">  
 <!-- 注入dao -->  
 <property name="accountDao" ref="accountDao"></property>  
 </bean>  
  
 <!-- 配置dao -->  
 <bean id="accountDao" class="com.chq.spring.dao.AccountDaoImpl">  
 <!-- 注入QueryRunner -->  
 <property name="runner" ref="runner"></property>  
 </bean>  
  
 <!-- 配置QueryRunner -->  
 <bean id="runner" class="org.apache.commons.dbutils.QueryRunner" scope="prototype">  
 <!-- 注入数据源 -->  
 <constructor-arg name="ds" ref="dataSource"></constructor-arg>  
 </bean>  
  
 <!-- 配置数据源 -->  
 <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  
 <property name="driverClass" value="com.mysql.jdbc.Driver"></property>  
 <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/db\_demo"></property>  
 <property name="user" value="root"></property>  
 <property name="password" value=""></property>  
 </bean>  
</beans>

测试代码如下

package com.chq.spring;  
  
import com.chq.spring.bean.Account;  
import com.chq.spring.service.IAccountService;  
import org.junit.Assert;  
import org.junit.Before;  
import org.junit.Test;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
import java.util.List;  
  
public class IAccountServiceTest {  
 private ApplicationContext ac;  
 private IAccountService service;  
  
 @Before  
 public void init() {  
 // 获取容器  
 ac = new ClassPathXmlApplicationContext("bean.xml");  
 // 获取service对象  
 service = ac.getBean("accountService", IAccountService.class);  
 }  
  
 @Test  
 public void testListAllAccounts() {  
 List<Account> accounts = service.listAllAccounts();  
 accounts.forEach(System.*out*::println);  
 }  
  
 @Test  
 public void testGetAccountById() {  
 Account account = service.getAccountById(1);  
 System.*out*.println(account);  
 }  
  
 @Test  
 public void testSaveAccount() {  
 Account account = new Account();  
 account.setName("test");  
 account.setBalance(2000.0);  
 int row = service.saveAccount(account);  
 Assert.*assertEquals*(1, row);  
 }  
  
 @Test  
 public void testUpdateAccount() {  
 Account account = service.getAccountById(7);  
 account.setBalance(1000.0);  
 int row = service.updateAccount(account);  
 Assert.*assertEquals*(1, row);  
 }  
  
 @Test  
 public void testRemoveAccount() {  
 int row = service.removeAccount(7);  
 Assert.*assertEquals*(1, row);  
 }  
}



## 纯注解方式

如果想要使用纯注解，那么我们就需要将配置文件中还存在的配置也使用注解配置，这时候就需要一些新的注解了。

@Configuration 注解

作用：用于指定当前类是一个 Spring 配置类，当创建容器时会从该类上加载注解

属性 value ： 用于指定配置类的字节码

注意：获取容器时需要使用AnnotationApplicationContext(有 @Configuration 注解的类.class)

@Configuration  
public class SpringConfiguration {  
// ...  
}

@ComponentScan 注解

作用：用于指定 Spring 在初始化容器时要扫描的包。相当于<context:component-scan base-package="com.chq"/>

属性： value / basePackages ：两者都是用来指定要扫描的包

@Configuration  
@ComponentScan(basePackages = "com.chq")  
public class SpringConfiguration {  
// ...  
}

@Bean

作用：该注解只能写在方法上，表明使用此方法创建一个对象，并且放入 Spring 容器

属性：name ：用于指定 Bean的 id ，默认值是当前方法的名称

细节：当我们使用该注解配置方法时，如果方法有参数，那么 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 一样

@Configuration  
@ComponentScan(basePackages = "com.chq")  
public class SpringConfiguration {  
 /\*\*  
 \* 创建一个 QueryRunner 对象并且放入 ioc 容器中，多例  
 \*  
 \* @return  
 \*/  
 @Bean(name = "runner")  
 @Scope("prototype")  
 public QueryRunner createQueryRunner(DataSource dataSource) {  
 return new QueryRunner(dataSource);  
 }  
   
 /\*\*  
 \* 创建一个数据源并且放入 ioc 容器中  
 \*  
 \* @return  
 \*/  
 @Bean(name = "dataSource")  
 public DataSource createDataSource() {  
 try {  
 ComboPooledDataSource ds = new ComboPooledDataSource();  
 ds.setDriverClass("com.mysql.jdbc.Driver");  
 ds.setJdbcUrl("jdbc:mysql://localhost:3306/db\_demo");  
 ds.setUser("root");  
 ds.setPassword("123456");  
 return ds;  
 } catch (PropertyVetoException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}

这样可以删除bean.xml文件了，然后将

ApplicationContext ac = new ClassPathXmlApplicationContext("Bean.xml")

修改为：

ApplicationContext ac = new AnnotationApplicationContext(SpringConfiguration.class)

@PropertySource 注解

属性 value[] ：用于指定配置文件的位置

细节：如果配置文件是在类路径下，需要写上 classpath:

# AOP

## AOP 的相关概念

AOP 即面向切面编程，英文全称为 Aspect Oriented Programming。简单的说它就是把我们程序重复的代码抽取出来，在需要执行的时候，使用动态代理的技术，在不修改源码的基础上，对我们的已有方法进行增强。

作用：在程序运行期间，不修改源码对已有方法进行增强。

优势：

减少重复代码

提高开发效率

维护方便

## 术语

Joinpoint (连接点)

所谓连接点是指那些被拦截到的方法，比如说，我们的业务层中的方法都是连接点，因为我们对业务层中的所有方法都进行了拦截

Pointcut (切入点)

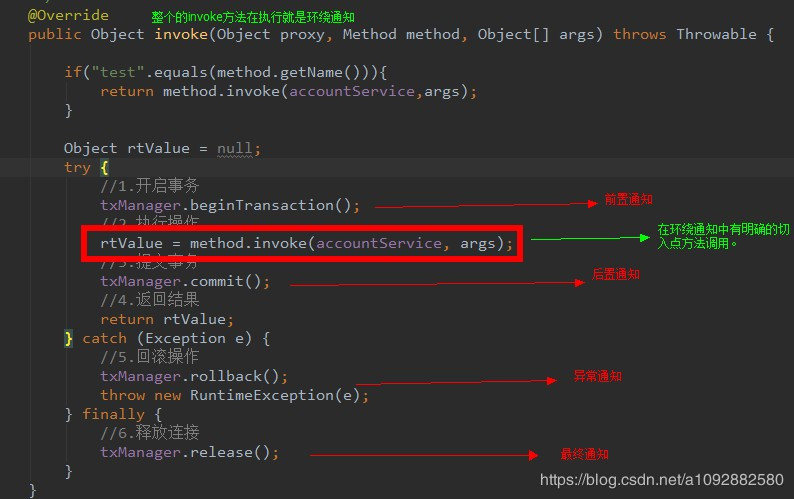
所谓切入点是指我们要对哪些 Joinpoint 进行拦截的定义，简单来说就是被增强的方法

比如说，我们在业务层新增了一个方法 test() ，但是在代理工厂中，我们不对该方法进行增强，而是直接放行，那么此时的 test() 就不是切入点，仅仅是一个连接点，而其他的方法都被事务管理，也就是切入点

Advice (通知/ 增强)

所谓通知是指拦截到方法之后所要做的事情，简单来说就是对切入点进行的增强操作

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 (目标对象)

被代理的对象

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

Proxy (代理)

一个类被 AOP 织入增强后，就产生一个结果代理类

Aspect (切面)

是切入点和通知（引介）的结合

## Spring 中基于 XML 的 AOP 配置

导入 aspectjweaver 依赖

<dependency>  
<groupId>org.aspectj</groupId>  
<artifactId>aspectjweaver</artifactId>  
<version>1.9.5</version>  
</dependency>

在配置文件bean.xml中导入 AOP 的名称空间

<?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">

将事务管理器类 TransactionManager 作为通知类用于增强，编写配置文件

<!-- 这里省略了 service 、 dao 、 QueryRunner 、 数据源 、 工具类等配置，参照之前的配置文件 -->  
  
 <!-- 配置事务管理器 -->  
<bean id="txManager" class="cn.ykf.utils.TransactionManager">  
<property name="connectionUtils" ref="connectionUtils"></property>  
</bean>  
  
 <!-- 配置aop -->  
<aop:config>  
<aop:pointcut id="pt1" expression="execution(\* cn.ykf.service.impl.\*.\*(..))"/>  
<aop:aspect id="txAdvice" ref="txManager">  
 <!-- 前置通知：开启事务，在切入点方法执行之前执行 -->  
 <aop:before method="beginTransaction" pointcut-ref="pt1"></aop:before>  
 <!-- 后置通知：提交事务，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="commit" pointcut-ref="pt1"></aop:after-returning>  
 <!-- 异常通知：回滚事务，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="rollback" pointcut-ref="pt1"></aop:after-throwing>  
 <!-- 最终通知：释放连接，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="release" pointcut-ref="pt1"></aop:after>  
</aop:aspect>  
</aop:config>

# JdbcTemplate

导入依赖

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.chq.test</groupId>  
 <artifactId>day04\_spring\_jdbc</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 <packaging>jar</packaging>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>5.2.4.RELEASE</version>  
 </dependency>  
  
 <!-- 使用JdbcTemplate -->  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-jdbc</artifactId>  
 <version>5.0.2.RELEASE</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-tx</artifactId>  
 <version>5.0.2.RELEASE</version>  
 </dependency>  
  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.6</version>  
 </dependency>  
 </dependencies>  
  
</project>

编写实体类

package com.chq.test.bean;  
  
import java.io.Serializable;  
  
public class Account implements Serializable {  
 private Integer id;  
 private String name;  
 private Double balance;  
  
 public Integer getId() {  
 return id;  
 }  
  
 public void setId(Integer id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public Double getBalance() {  
 return balance;  
 }  
  
 public void setBalance(Double balance) {  
 this.balance = balance;  
 }  
  
 @Override  
 public String toString() {  
 return "Account{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 ", balance=" + balance +  
 '}';  
 }  
}

实现持久层

package com.chq.test.dao;  
  
import com.chq.test.bean.Account;  
  
public interface AccountDao {  
 Account findById(int id);  
}

package com.chq.test.dao;  
  
import com.chq.test.bean.Account;  
import org.springframework.jdbc.core.BeanPropertyRowMapper;  
import org.springframework.jdbc.core.support.JdbcDaoSupport;  
  
import java.util.List;  
  
public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao{  
  
 public Account findById(int id) {  
 List<Account> list = getJdbcTemplate().query("select \* from account where id=?",new BeanPropertyRowMapper<Account>(Account.class),id);  
 return list.get(0);  
 }  
}

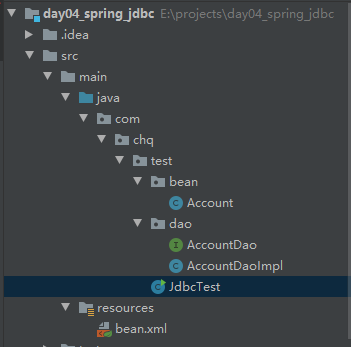
配置数据源

<?xml version="1.0" encoding="UTF-8"?>  
<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="accountDao" class="com.chq.test.dao.AccountDaoImpl">  
 <property name="dataSource" ref="dataSource"></property>  
 </bean>

<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/db\_demo"></property>  
 <property name="username" value="root"></property>  
 <property name="password" value=""></property>  
 </bean>  
</beans>

测试

package com.chq.test;  
  
import com.chq.test.bean.Account;  
import com.chq.test.dao.AccountDao;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class JdbcTest {  
 public static void main(String[] args) {  
 ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml");  
 AccountDao accountDao = ac.getBean("accountDao",AccountDao.class);  
 Account account = accountDao.findById(1);  
 System.*out*.println(account);  
 }  
}



# Spring 中的事务控制

## 基本介绍

Spring 框架为我们提供了一组事务控制的接口，这组接口是在 spring-tx-5.0.2.RELEASE.jar 中，因此我们需要导入相关依赖

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-tx</artifactId>  
 <version>5.0.2.RELEASE</version>  
</dependency>

## Spring 中事务控制的 API 介绍

### PlatformTransactionManager 接口

|  |  |
| --- | --- |
| 常用方法 | 作用 |
| TransactionStatus getTransaction(@Nullable TransactionDefinition definition) | 获取事务状态信息 |
| void commit(TransactionStatus status) | 提交事务 |
| void rollback(TransactionStatus status) | 回滚事务 |

在开发我们一般使用以下实现类:

org.springframework.jdbc.datasource.DataSourceTransactionManager

使用Spring JDBC 或者 Mybatis 进行持久化数据时使用

### TransactionDefinition 接口

TransactionDefinition 接口是事务的定义信息对象，提供以下方法

常用方法 作用

String getName() 获取事务的名称

int getIsolationLevel() 获取事务的隔离级别

int getPropagationBehavior 获取事务的传播行为

int getTimeout() 获取事务的超时时间

boolean isReadOnly() 获取事务是否只读

### TransactionStatus 接口

常用方法 作用

void flush() 刷新事务

boolean hasSavepoint() 获取是否存在存储点

boolean isCompleted() 获取事务是否完成

boolean isNewTransaction() 获取事务是否为新的事务

boolean isRollbackOnly() 获取事务是否回滚

void setRollbackOnly() 设置回滚

## 基于 XML 的声明式事务控制

注意表需要为innodb结构

导入相关依赖

<dependencies>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>5.2.4.RELEASE</version>  
 </dependency>  
  
 <!-- 使用JdbcTemplate -->  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-jdbc</artifactId>  
 <version>5.0.2.RELEASE</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-tx</artifactId>  
 <version>5.0.2.RELEASE</version>  
 </dependency>  
  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.6</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.aspectj</groupId>  
 <artifactId>aspectjweaver</artifactId>  
 <version>1.8.7</version>  
 </dependency>  
</dependencies>

实体类

package com.chq.test.bean;  
  
import java.io.Serializable;  
  
public class Account implements Serializable {  
 private Integer id;  
 private String name;  
 private Double balance;  
  
 public Integer getId() {  
 return id;  
 }  
  
 public void setId(Integer id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public Double getBalance() {  
 return balance;  
 }  
  
 public void setBalance(Double balance) {  
 this.balance = balance;  
 }  
  
 @Override  
 public String toString() {  
 return "Account{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 ", balance=" + balance +  
 '}';  
 }  
}

持久层

package com.chq.test.dao;  
  
import com.chq.test.bean.Account;  
  
public interface AccountDao {  
  
 int updataeBalance(Account account);  
}

package com.chq.test.dao;  
  
import com.chq.test.bean.Account;  
import org.springframework.jdbc.core.support.JdbcDaoSupport;  
  
  
public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao{  
  
 public int updataeBalance(Account account) {  
 return getJdbcTemplate().update("update account set balance=? where id=?",account.getBalance(),account.getId());  
 }  
}

实现层

package com.chq.test.service;  
  
public interface AccountService {  
 void updateBalance();  
  
}

package com.chq.test.service.impl;  
  
import com.chq.test.bean.Account;  
import com.chq.test.dao.AccountDao;  
import com.chq.test.service.AccountService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
@Service("accountService")  
public class AccountServiceImpl implements AccountService {  
  
 @Autowired  
 private AccountDao accountDao;  
  
 @Override  
 public void updateBalance() {  
 Account account = new Account();  
 account.setId(1);  
 account.setBalance(4000D);  
 accountDao.updataeBalance(account);  
// int i = 1/0;  
 Account accountt = new Account();  
 accountt.setId(2);  
 accountt.setBalance(4000D);  
 accountDao.updataeBalance(accountt);  
 }  
}

Bean.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"  
 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/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">  
  
 <!-- 配置要扫描的包 -->  
 <context:component-scan base-package="com.chq.test"/>  
  
 <bean id="accountDao" class="com.chq.test.dao.AccountDaoImpl">  
 <property name="dataSource" ref="dataSource"></property>  
 </bean>  
  
  
 <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/db\_demo"></property>  
 <property name="username" value="root"></property>  
 <property name="password" value=""></property>  
 </bean>  
  
 <!-- 配置事务管理器 -->  
 <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dataSource"/>  
 </bean>  
  
 <!-- 配置事务的通知 -->  
 <tx:advice id="txAdvice" transaction-manager="transactionManager">  
 <!-- 配置事务的属性 -->  
 <tx:attributes>  
 <!-- 所有方法都有事务控制，但是优先级低于 "get\*" -->  
 <tx:method name="\*" propagation="REQUIRED" read-only="false"/>  
 <!-- 所有查询方法都为只读事务，传播行为：支持事务 -->  
 <tx:method name="get\*" propagation="SUPPORTS" read-only="true"/>  
 </tx:attributes>  
 </tx:advice>  
  
 <!-- 配置aop -->  
 <aop:config>  
 <!-- 配置切入点表达式 -->  
 <aop:pointcut id="pt1" expression="execution(\* com.chq.test.service.impl.\*.\*(..))"/>  
 <!-- 建立切入点表达式和事务通知的关系 -->  
 <aop:advisor advice-ref="txAdvice" pointcut-ref="pt1"></aop:advisor>  
 </aop:config>  
</beans>

测试类

package com.chq.test;  
  
import com.chq.test.service.AccountService;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class JdbcTest {  
  
 public static void main(String[] args) {  
 ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml");  
 AccountService accountService = ac.getBean("accountService", AccountService.class);  
 accountService.updateBalance();  
 }  
}

