

1、Spring是什么

spring是一个高度灵活的**轻量级框架**，其目的是降低企业级应用开发的复杂度。

MVC设计模式：

model

模型

pojo：封装类，一张表对应着一个实体类

dao：数据访问层，增(insert) 删(delete) 改(update) 查(select)

service：业务处理层，对dao层的增删改查操作做逻辑处理的

```
Dao dao = new Dao();
```

view

html jsp

controller

处理请求和响应结果

访问service

```
Service s = new Service();
```

2、IOC 控制反转

IOC (Inversion of Control)：控制反转，通过容器来控制对象的创建及维护，对象中成员变量的创建及维护。反转就是将对象的控制权转移给容器处理，目的是获得更好的扩展性和可维护性。

```
public class Boy {  
    private Dog dog;  
    public Dog getDog() {  
        return dog;  
    }  
    public void setDog(Dog dog) {  
        this.dog = dog;  
    }  
}  
  
public class Dog {  
    private String dogName;  
    public String getDogName() {  
        return dogName;  
    }  
    public void setDogName(String dogName) {  
        this.dogName = dogName;  
    }  
}
```

```
Dog dog = new Dog();  
dog.setDogName("旺旺");  
Boy boy = new Boy();  
boy.setDog(dog);
```

```
容器 a= new 容器();  
Boy boy = a.get();
```

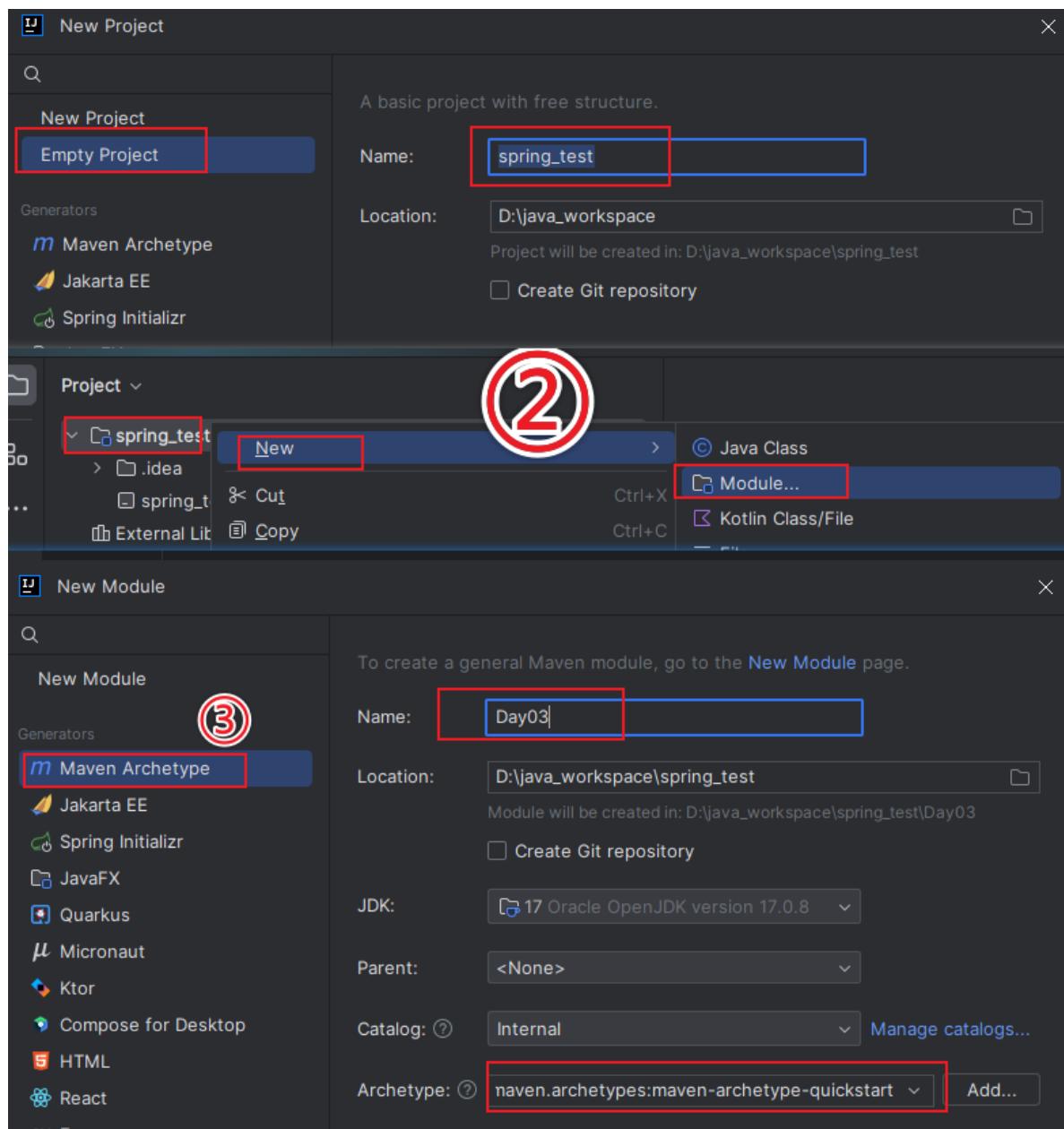
传统的对象创建及维护方式

采用**IOC**来创建与维护对象的方式（**模拟**），对象当中的依赖关系，也依赖于容器处理。

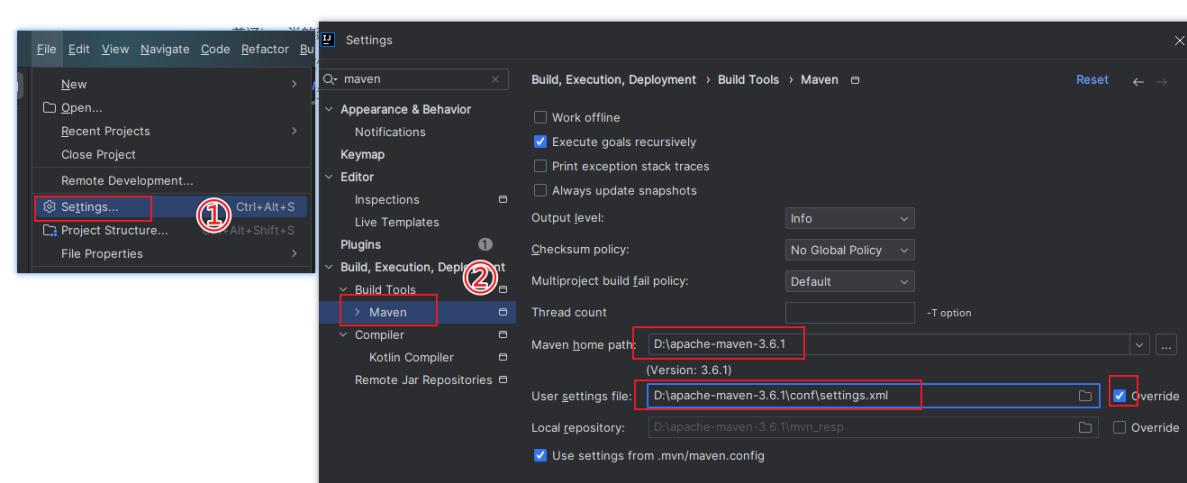


3、入门程序编写

新建maven项目：



配置maven



- 导入包spring核心容器相关依赖

```

16
17     <dependencies>
18         <dependency>
19             <groupId>junit</groupId>
20             <artifactId>junit</artifactId>
21             <version>3.8.1</version>
22             <scope>test</scope>
23         </dependency>
24         <!-- 引入spring容器依赖 -->
25         <dependency>
26             <groupId>org.springframework</groupId>
27             <artifactId>spring-context</artifactId>
28             <version>6.0.0</version>
29         </dependency>
30     </dependencies>

```

Profiles

m Day0303

- Lifecycle
- Plugins
- Dependencies
 - junit:junit:3.8.1 (test)
 - org.springframework:spring-context:6.0.0
 - org.springframework:spring-aop:6.0.0
 - org.springframework:spring-beans:6.0.0
 - org.springframework:spring-core:6.0.0
 - org.springframework:spring-expression:6.0.0

• 普通java类的建立

object

spring_test

Day0303

src

- main
 - java
 - com.zretc
 - controller
 - MainTest
 - pojo
 - Dog
 - Boy
- resources
 - applicationContext.xml
- test

Dog.java

```

1 package com.zretc.pojo;
2
3 public class Dog {
4     // 名字
5     private String dog_name;
6 }
7 usages
8
9

```

Boy.java

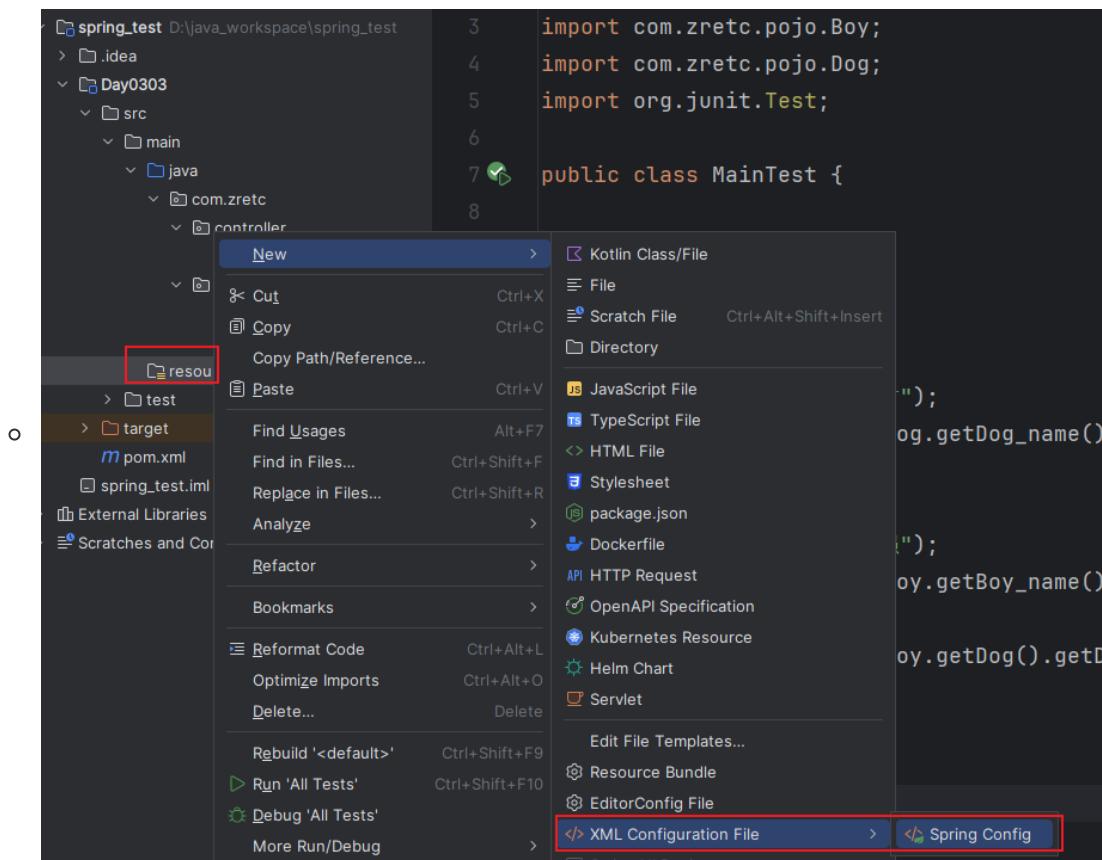
```

3 usages
4 public class Boy {
5     private String boy_name;
6     // 一个男孩有一条狗
7     private Dog dog;
8 }
9 usages

```

```
Dog.java MainTest.java x
2
3 import com.zretc.pojo.Boy;
4 import com.zretc.pojo.Dog;
5 import org.junit.Test;
6
7 public class MainTest {
8
9     @Test
10    public void test01(){
11        // 创建狗类对象
12        Dog dog = new Dog();
13        dog.setDog_name("旺财");
14        System.out.println(dog.getDog_name());
15        // 创建男孩对象
16        Boy boy = new Boy();
17        boy.setBoy_name("小强");
18        System.out.println(boy.getBoy_name());
19        boy.setDog(dog);
20        System.out.println(boy.getDog().getDog_name());
21    }
}
```

- spring配置文件



- 通过spring内置API接口初始化spring容器，并获取spring 容器管理的java类的实例。

```

applicationContext.xml
<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.xsd">
    <!-- 创建Dog类对象
        id 唯一的表示，不允许重复
        class 通过反射机制找到要创建的类的路径
    -->
    <bean id="dog" class="com.zretec.pojo.Dog"/>
    <!-- 创建boy
        -->
    <bean id="boy" class="com.zretec.pojo.Boy"/>
    <bean id="boy2" class="com.zretec.pojo.Boy"/>

```

```

MainTest.java
@Test
public void test02() {
    // 获取容器
    ApplicationContext ctxt = new ClassPathXmlApplicationContext(configLocation: "applicationContext.xml");
    // 获取dog对象
    Dog dog = (Dog) ctxt.getBean(name: "dog");
    dog.setDog_name("花花");
    System.out.println(dog.getDog_name());

    // 获取boy
    Boy boy = (Boy) ctxt.getBean(name: "boy2");
    boy.setBoy_name("大壮");
    boy.setDog(dog);
    System.out.println(boy.getBoy_name() + ":" + boy.getDog().getDog_name());
}

```

4、实例化对象的方式

4.1 构造器

Spring默认的情况下是调用java类的构造器进行初始化的：

```

public class Dog {
    private String name;
    private int age;
    public Dog() {
        System.out.println("初始化");
    }
    public void setAge(int age) {
        this.age = age;
    }
}
<bean id="dog" class="com.chinasofti.bean.Dog">
    <constructor-arg index="0" value="贵宾犬"></constructor-arg>
    <constructor-arg index="1" value="5"></constructor-arg>
</bean>

```

constructor-arg标签：设定初始化bean的构造方法参数

index：构造方法参数位置，从0开始。

value：参数值。

spring根据constructor-arg标签判断调用哪个构造器初始化对象实例。



```

<beans xmlns="http://www.springframework.org/schema/bean" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.springframework.org/schema/bean http://www.springframework.org/schema/bean/spring.xsd">
    <!-- 创建Dog类对象
        id 唯一的表示，不允许重复
        class 通过反射机制找到要创建的类的路径
    -->
    <bean id="dog666" class="com.zretec.pojo.Dog">
        <constructor-arg index="0" value="花花"/>
    </bean>

    <!-- 无参构造方法实例化bean:创建boy -->
    <bean id="boy" class="com.zretec.pojo.Boy"/>
    <!-- 通过构造方法创建对象 -->
    <bean id="boy2" class="com.zretec.pojo.Boy">
        <!-- 给参数初始化
            index 代表的是参数的下标(索引)，从0开始的
            value 给基本类型的变量初始化
            8个基本数据类型，对应的包装类，String ,StringBuffer,StringBuilder
            ref 给bean组件初始化,从bean标签的id值
        -->
        <constructor-arg index="0" value="小强"/>
        <constructor-arg index="1" ref="dog666"/>
    </bean>
</beans>

```

```

public class MainTest {
    @Test
    public void test() {
        ApplicationContext ctxt = new ClassPathXmlApplicationContext("applicationContext.xml");
        Dog dog = (Dog) ctxt.getBean("boy");
        dog.setDog_name("花花");
        System.out.println(dog.getDog_name());
    }
}

```

4.2 静态工厂

静态工厂模式：工厂类使用静态方法创建对象。

```

public class Product {
    public void desc() {
        System.out.println("调用产品的desc方法");
    }
}

public class StaticFactory {
    public static Product getProduct() {
        return new Product();
    }
}

<bean id="product" class="com.chinasofti.bean.StaticFactory" factory-method="getProduct">
</bean>

```

配置文件：获取静态工厂创建的实例，只需要配置工厂类，及工厂类对应创建实例的方法，Spring会执行静态工厂类的静态方法创建对象，并返回引用。

测试程序

```

Product product = (Product) ac.getBean("product");
product.desc();

```

工厂类生产的对象

静态工厂类



```

package com.zretec.factory;

import com.zretec.pojo.Dog;

// 静态工厂
public class StaticFactory {
    // 生产品
    public static Dog getDog() {
        return new Dog();
    }
}

```

```

public class MainTest {
    @Test
    public void test04() {
        // 获取容器
        ApplicationContext ctxt = new ClassPathXmlApplicationContext("applicationContext.xml");
        Dog dog = (Dog) ctxt.getBean("dog");
        dog.setDog_name("花花");
        System.out.println(dog.getDog_name());
    }
}

```

```

<beans xmlns="http://www.springframework.org/schema/bean" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.springframework.org/schema/bean http://www.springframework.org/schema/bean/spring.xsd">
    <!-- 通过静态工厂实例化狗类对象 -->
    <bean id="dog" class="com.zretec.factory.StaticFactory" factory-method="getDog"/>

```

4.3 普通工厂

普通工厂模式：需要初始化工厂类实例，通过执行工厂类的方法创建产品对象并返回引用。

```
public class Product {  
    public void desc() {  
        System.out.println("调用产品的desc方法");  
    }  
}  
  
public class CustomFactory {  
    public Product getProduct() {  
        return new Product();  
    }  
}  
  
<bean id="customFactory" class="com.chinasofti.bean.CustomFactory">  
</bean>  
<bean id="product" factory-bean="customFactory" factory-method="getProduct">  
</bean>
```

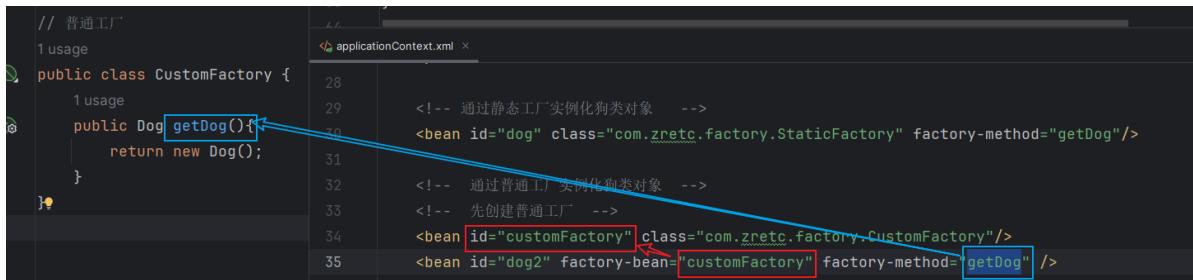
工厂类生产的对象

普通工厂类

配置文件：当通过ac.getBean()获取product实例，spring会创建factory实例，并执行工厂方法，返回对象。

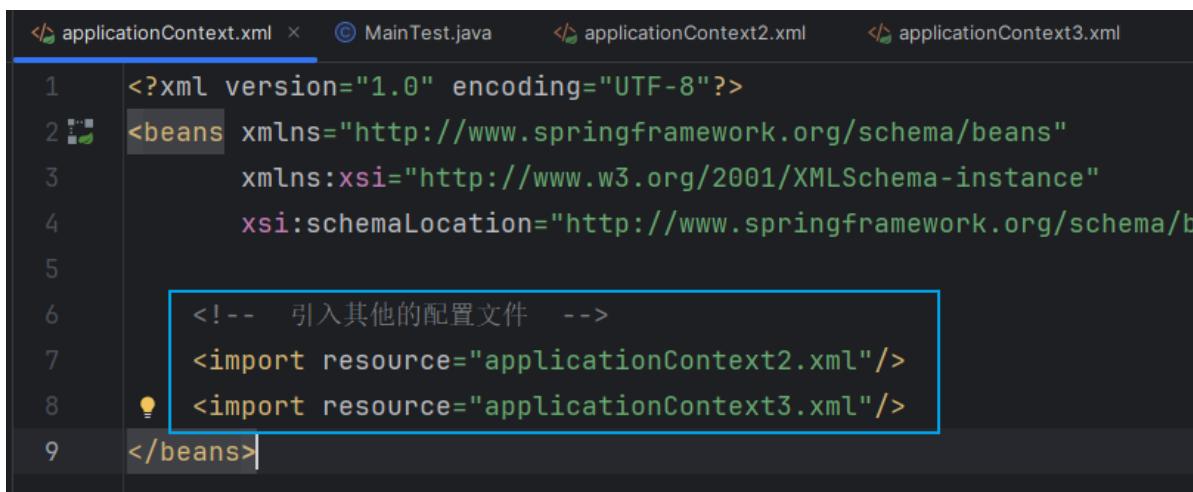
```
Product product = (Product) ac.getBean("product");  
product.desc();
```

测试程序



```
// 普通工厂  
1 usage  
public class CustomFactory {  
    1 usage  
    public Dog getDog(){  
        return new Dog();  
    }  
}  
  
<!-- 通过静态工厂实例化狗类对象 -->  
<bean id="dog" class="com.zreto.factory.StaticFactory" factory-method="getDog"/>  
  
<!-- 通过普通工厂实例化狗类对象 -->  
<!-- 先创建普通工厂 -->  
<bean id="customFactory" class="com.zreto.factory.CustomFactory"/>  
<bean id="dog2" factory-bean="customFactory" factory-method="getDog" />
```

4.4 定义多个配置文件



```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xsi:schemaLocation="http://www.springframework.org/schema/b  
  
<!-- 引入其他的配置文件 -->  
<import resource="applicationContext2.xml"/>  
<import resource="applicationContext3.xml"/>  
</beans>
```

注意：resource值只能为引入配置文件的相对路径（相对于主配置文件）。

5、bean的管理

5.1 bean的作用域与初始化时间

➤ bean的作用域：组件bean有效的时间。通过配置文件设定。

```
<bean id="userDao" class="com.chinasofti.dao.UserDao"  
scope="singleton" lazy-init="true"></bean>
```

- scope取值范围：默认singleton

类 别	说 明
singleton	在 Spring IoC 容器中仅存在一个 Bean 实例，Bean 以单实例的方式存在
prototype	每次从容器中调用 Bean 时，都返回一个新的实例，即每次调用 getBean() 时，相当于执行 new XxxBean() 的操作
request	每次 HTTP 请求都会创建一个新的 Bean。该作用域仅适用于 WebApplicationContext 环境
session	同一个 HTTP Session 共享一个 Bean，不同 HTTP Session 使用不同的 Bean。该作用域仅适用于 WebApplicationContext 环境
globalSession	同一个全局 Session 共享一个 Bean，一般用于 Portlet 应用环境。该作用域仅适用于 WebApplicationContext 环境

bean的作用域和初始化时间：

scope 作用域：

singleton 单实例的，默认（一个类的对象在内存中只有一个）

prototype 多实例的（一个类的对象在内存中有多个）

lazy-init 初始化时间：

立即加载： 默认情况下是 false/default，称为是“立即加载”，

立即加载就是初始化容器时直接创建对象

延迟加载： true，代表的是，“延迟加载”，

只有在 scope="singleton" 时有效，第一次调用 getBean() 时创建对象

多实例是每次调用 getBean() 都会创建一个对象，因此设置成延迟加载没有

意义

5.1.1 bean的作用域

1) 单实例的

```
<bean id="dog1" class="com.zreetc.pojo.Dog" scope="singleton"/>
```

```
ApplicationContext ctxt = new ClassPathXmlApplic  
Dog dog = (Dog) ctxt.getBean(name: "dog1");  
Dog dog2 = (Dog) ctxt.getBean(name: "dog1");  
System.out.println(dog == dog2); //true
```

结果为true，证明是一个对象

2) 多实例的

```
<bean id="dog1" class="com.zreetc.pojo.Dog" scope="prototype"/>
```

```
ApplicationContext ctxt = new ClassPathXmlApplic
Dog dog = (Dog) ctxt.getBean( name: "dog1");
Dog dog2 = (Dog) ctxt.getBean( name: "dog1");
System.out.println(dog == dog2); //false
```

结果为false,证明每次调用getBean()都会创建一个对象

5.1.2 初始化时间

1) 立即加载

```
<bean id="dog1" class="com.zreetc.pojo.Dog" lazy-init="default"/>
```

```
@Test
public void test05() {
    获取容器
    ApplicationContext ctxt = new ClassPathXmlApplica
    Dog() 无参的构造方法。。。.
```

默认情况下是立即加载，在加载容器时就创建对象了

2) 延迟加载

没有调用getBean():

```
<bean id="dog1" class="com.zreetc.pojo.Dog" lazy-init="true"/>
```

```
@Test
public void test05() {
    获取容器
    ApplicationContext ctxt = new ClassPathXmlApplica
```

```
✓ Tests passed: 1 of 1 test – 338 ms
"C:\Program Files\Java\jdk-17\bin\java.exe"
Process finished with exit code 0
```

设置为延迟加载后，初始化容器，并没有创建对象，直接无参构造方法

调用getBean():

```
public void test05() {  
    // 获取容器  
    ApplicationContext ctxt = new ClassPathXmlApplicationCon...  
    Dog dog = (Dog) ctxt.getBean(name: "dog1");  
    Dog dog2 = (Dog) ctxt.getBean(name: "dog1");
```



调用getBean()后才会，创建对象，执行构造方法

5.2 bean初始化和销毁方法

- spring在创建bean实例后，调用bean的初始化方法。spring在容器关闭后，bean不被spring容器调用，进入可垃圾回收阶段，在容器关闭之前，会调用bean的销毁方法(只有scope=singleton的bean才会执行销毁方法)。

```
<bean id="dog" class="com.chinasofti.bean.Dog"  
      scope="singleton" lazy-init="true"  
      init-method="init"  
      destroy-method="destroy">  
    <constructor-arg index="0" value="贵宾犬"/>  
    <constructor-arg index="1" value="5"/>  
</bean>  
  
ClassPathXmlApplicationContext ac = new  
    ClassPathXmlApplicationContext("com/chinasofti/test/test2_4_1/applicationContext.xml");  
Dog dog = (Dog) ac.getBean("dog");  
dog.shout();  
ac.close();
```

```

Dog.java
16     System.out.println(" Dog() 无参的构造方法。。。");
17 }
18
19 no usages
20 public Dog(String dog_name) { this.dog_name = dog_name; }
21
22 // 初始化方法
23 1 usage
24 public void init() { System.out.println("初始化方法。。。"); }
25 // 销毁方法
26 1 usage
27 public void destroy() {
28     System.out.println("销毁方法。。。");
29 }
30
31 }

applicationContext4.xml
18 <!--
19     初始化方法： 创建对象后执行
20     销毁方法： 关闭容器之前执行
21     注意： 必须在单实例模式下有效
22 -->
23 <bean id="dog2" class="com.zretec.pojo.Dog" init-method="init" destroy-method="destroy"/>
24 </beans>

```

```

@Test
public void test05() {
    获取容器
    ClassPathXmlApplicationContext ctxt = new ClassPathXmlApplicationContext(configLocation: "applicationContext4.xml");
    Dog dog = (Dog) ctxt.getBean(name: "dog1");
    Dog dog2 = (Dog) ctxt.getBean(name: "dog1");
    System.out.println(dog == dog2); //false
    Dog dog3 = (Dog) ctxt.getBean("dog2");
    System.out.println(dog3 == dog2);
    关闭容器
    ctxt.close();
}

```

✓ Tests passed: 1 of 1 test – 360 ms

"C:\Program Files\Java\jdk-17\bin\java.exe"

Dog() 无参的构造方法。。。

初始化方法。。。

销毁方法。。。

6、依赖注入 DI

依赖注入：由外部容器动态地将依赖对象注入到另一个对象的组件中。spring采用这种方式为bean的属性赋值。

通俗的说spring容器不仅可以初始化对象，也可以为对象当中的成员变量赋值，初始化成员变量对象以及赋值的过程，不需手动编码，而是由spring容器去做。这种依赖容器初始化对象，并赋值给bean全局变量的方式叫依赖注入。

依赖注入方式

传统的方式：

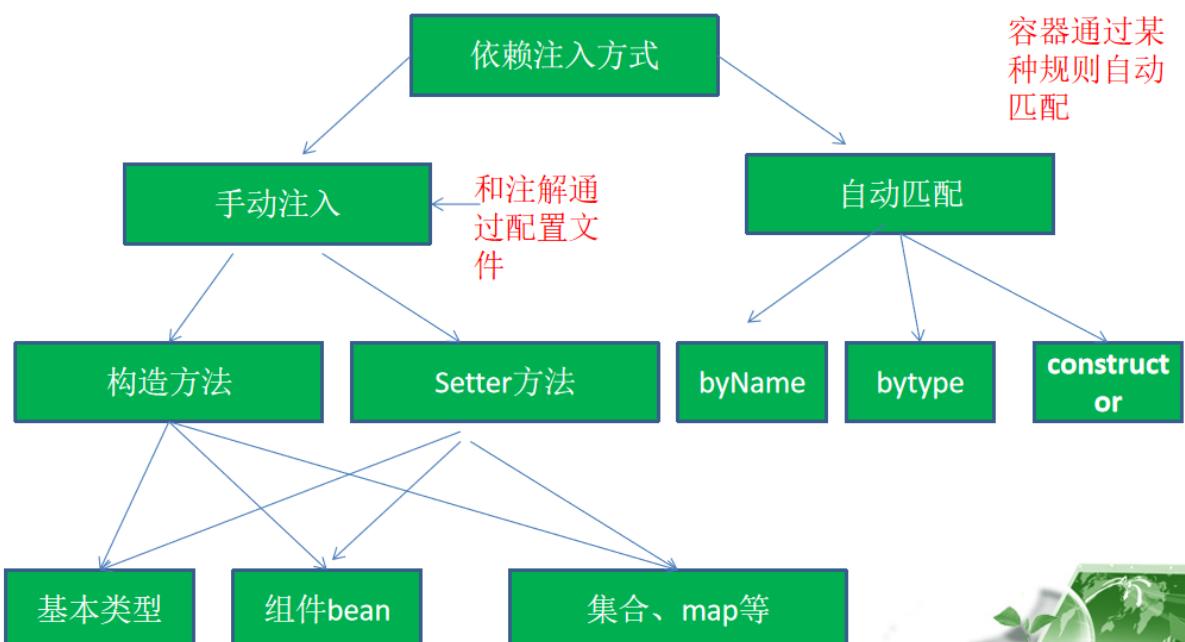
```
Boy boy = new Boy();  
Dog dog = new Dog();  
boy.setDog(dog);
```

Spring容器会分别初始化两个组件，并依据配置的依赖关系，将Boy对象的dog变量赋值。

```
<bean id="boy"  
class="com.chinasofti.entity.Boy">  
    <property name="dog"  
ref="dog"></property>  
</bean>  
<bean id="dog"  
class="com.chinasofti.entity.Dog"></bean>
```



2.5.1 依赖注入



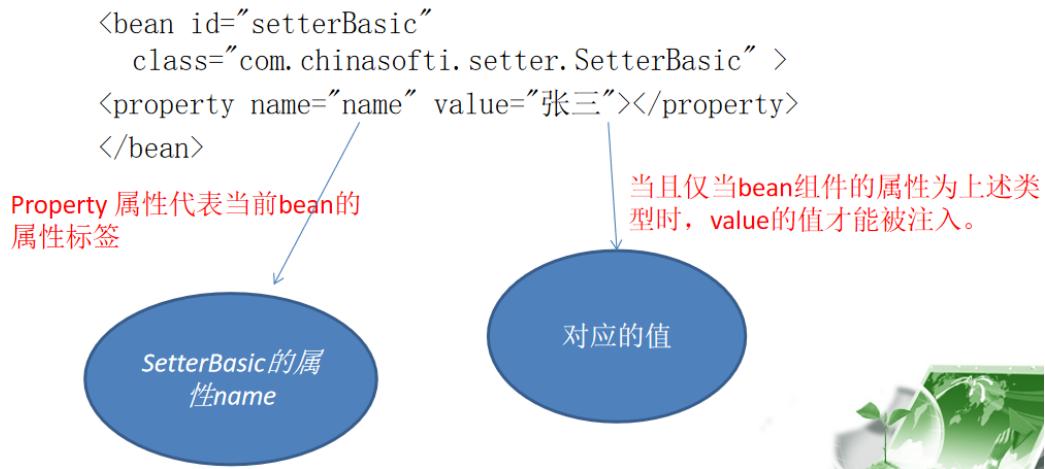
6.1 setter方法注入

setter方法注入：spring容器调用bean组件中属性对应的setter方法完成属性的注入（赋值）。

根据属性的不同类型，可分为如下三种注入情况。

1. 基本类型注入。

- 基本类型注入泛指：
 - 当前bean组件的属性为基本类型
 - byte short int long float double char boolean
 - 基本类型的封装类型
 - Byte Short Integer Long Float Double Character Boolean
 - String、StringBuffer类型，StringBuilder
-



2. spring组件类型注入。

- 当前bean组件的属性为spring容器的其他bean组件，容器可以读取配置文件对当前属性进行赋值。
 - 内部bean

```

<bean id = "setterBean1" class="com.chinasofti.setter.SetterBean">
  <property name="setterBasic1">
    <bean id="setterBasic1"
          class="com.chinasofti.setter.SetterBasic" >
      </bean>
  </property>
</bean>

```

内部bean的缺点是外部的其他bean组件无法访问。（不建议使用）

3. 集合类型注入。

- 当前bean组件的属性为List Set Map、数组、Properties等。
- List注入方式:

```

<bean id = "setterList" class="com.chinasofti.setter.SetterList">
  <property name="list">
    <list>
      <value type="java.lang.String">1</value>
      <ref bean="bean"/>
      <null></null>
    </list>
  </property>
</bean>

```

List中对象的顺序，按照从上到下的赋值顺序，list当中的数据可以为基本类型、spring组件类型、集合类型等。

- Set注入方式

```

<property name="set">
    <set>
        <value type="java.lang.String">1</value>
    <ref bean="bean"/>
    </set>
</property>

```

set当中的数据可以为基本类型、spring组件类型、集合类型等。

- map注入方式

```

<property name="map">
    <map>
        <entry key-ref="bean" value-ref="" />
    <entry key="aa" value="bb" />
    </map>
</property>

```

Map的key值可以使用key=“”去引用基本类型，也可以使用key-ref引用spring组件，value的值类似。

```

<?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">

    <!--
        依赖注入DI:
        给实体类的成员变量初始化
    -->
    <bean id="dog1" class="com.zretc.pojo.Dog">
        <property name="dog_name" value="旺财"/>
    </bean>
    <!-- Setter方式注入，实际上就是调用实体类中的setXXX()方法 -->
    <bean id="boy1" class="com.zretc.pojo.Boy">
        <!-- 1、基本类型的依赖注入 -->
        <property name="boy_name" value="大壮"/>
        <!-- 2、bean类型的依赖注入 -->
        <property name="dog" ref="dog1"/>
        <!-- 内部bean,不建议使用 -->
    <!-- <property name="dog"><!-->
    <!-- <bean id="dog2" class="com.zretc.pojo.Dog"/><!--&gt;
    &lt;!-- &lt;/property><!--&gt;
    &lt;!-- 3、集合 --&gt;
    &lt;property name="list"&gt;
        &lt;list&gt;
            &lt;value type="java.lang.String"&gt;打篮球&lt;/value&gt;
            &lt;value&gt;唱歌&lt;/value&gt;
            &lt;value&gt;180&lt;/value&gt;
            &lt;ref bean="dog1"/&gt;
        &lt;/list&gt;
    &lt;/property&gt;
    &lt;property name="set"&gt;
</pre>

```

```
<set>
    <value type="java.lang.String">打篮球</value>
    <value>唱歌</value>
    <value>180</value>
    <ref bean="dog1"/>
</set>
</property>
<property name="map">
    <!-- 键值对 -->
    <map>
        <entry key="1" value="唱歌"/>
        <entry key-ref="dog1" value-ref="dog1"/>
        <entry key-ref="dog1" value="花花"/>
    </map>
</property>
</bean>
</beans>
```

```
@Test
public void test06() {
    // 获取容器
    ClassPathXmlApplicationContext ctxt = new
    ClassPathXmlApplicationContext("applicationContext5.xml");
    Boy boy = (Boy) ctxt.getBean("boy1");
    System.out.println(boy.getBoy_name());
    System.out.println(boy.getDog().getDog_name());
    List list = boy.getList();
    list.forEach((obj)-> System.out.println(obj));
    Set set = boy.getSet();
    set.forEach((obj)-> System.out.println(obj));
    Map map = boy.getMap();
    Set set1 = map.keySet();
    set1.forEach((obj)-> System.out.println(obj + ":" + map.get(obj)));
}
```

```
✓ Tests passed: 1 of 1 test - 409 ms
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
Dog() 无参的构造方法。。
boy() 无参构造方法。。
大壮
旺财
打篮球
唱歌
180
com.zretc.pojo.Dog@28701274
打篮球
唱歌
180
com.zretc.pojo.Dog@28701274
1:唱歌
com.zretc.pojo.Dog@28701274:花花

Process finished with exit code 0
```

6.2 构造器方式注入

构造器方法注入：spring在初始化bean组件时，调用含参数的构造器对全局变量进行复制，参数类型可以为基本类型、bean组件类型、集合类型，赋值方式与setter方法赋值方式一致。

```
<bean id="boy" class="com.chinasofti.constructor.Boy" >
    <constructor-arg index="0" value="123"></constructor-arg>
    <constructor-arg index="1" value="哈哈" type="java.lang.String">
    </constructor-arg>
    <constructor-arg index="1" ref="beanName"></constructor-arg>
</bean>
```

6.3 自动注入

自动装配（了解知识）

对于自动装配，大家了解一下就可以了，实在不推荐大家使用。例子：

```
<bean id="foo" class="...Foo" autowire="byType">
```

自动装配可以省略<property></property>

autowire属性取值如下

- * **byType** : 按类型装配，可以根据属性的类型，在容器中寻找跟该类型匹配的bean。如果发现多个，那么将会抛出异常。如果没有找到，即属性值为null。
- * **byName**: 按名称装配，可以根据属性的名称，在容器中寻找跟该属性名相同的bean，如果没有找到，即属性值为null。
- * **constructor**与byType的方式类似，不同之处在于它应用于构造器参数。如果在容器中没有找到与构造器参数类型一致的bean，否则将会抛出异常。

7、spring项目程序架构

- 接口：在表面上是由几个没有主体代码的方法定义组成的集合体，有唯一的名称，可以被类或其他接口所实现（或者也可以说继承）。
- 面向接口编程：在系统分析或架构设计中，每个层级的程序并不是直接提供程序服务，而是定义一组接口，通过实现接口来提供功能。面向接口编程实际是面向对象编程的一部分。

最简单的面向接口编程：

接口

```
public interface IntfaceDemo {  
    void insert();  
}
```

实现类

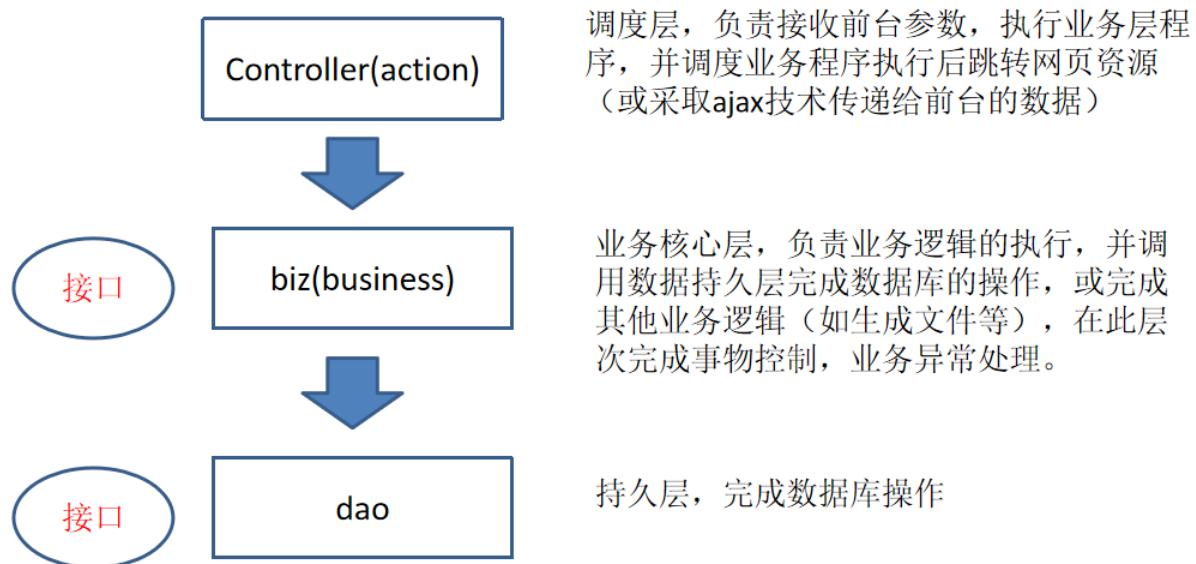
```
public class IntImpl implements IntfaceDemo {  
    public void insert() {  
    }  
}
```



7.1 面向接口编程

- 面向接口编程-优点
- 接口的定义和接口的实现分开。
 - 1) 从分工看，接口定义一般是由架构师来设定，编程人员实现，架构师会根据架构规则、设计规则来制定接口，对编程人员提供了规范。
 - 2) 从实现看，接口的定义时间很短，但接口的实现时间较长，若一个编程人员需要调用其他人员编写的某个方法时，可以采用多态的方式获取接口对象，来调用方法，这样保证团队共同完成开发。
 - 3) 从架构设计看，接口实现分开，程序更清晰，易读。
- 接口可以有多个实现：
 - 1) 如果实现类的业务需要扩展某项功能，可以采用重新实现接口的方式，这样降低了程序的冗余性。
 - 2) 接口的多实现易于通过配置文件的方式配置接口的实现类。

7.2 spring的三层架构模式

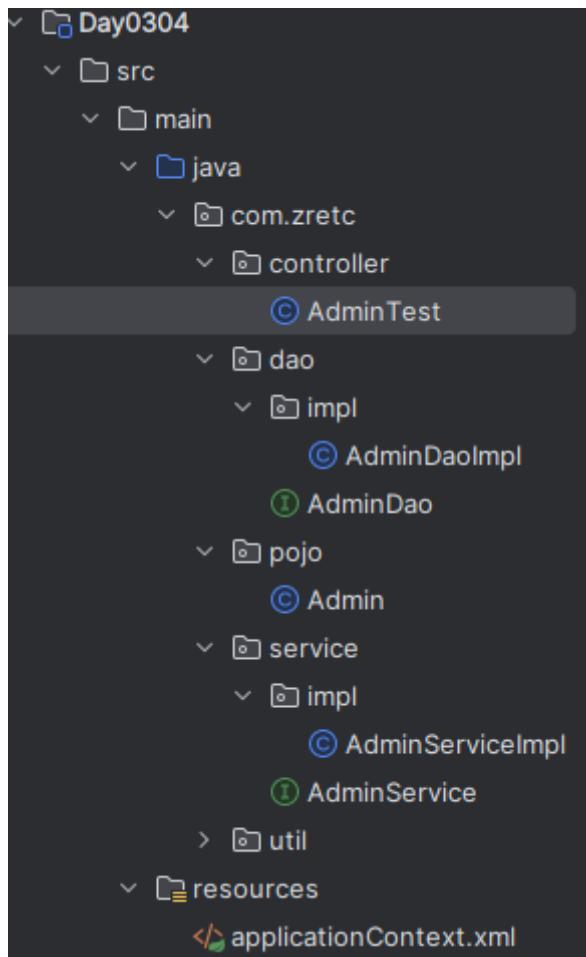


3层架构模式是在mvc设计模式上衍生出来的，每个企业的业务需求、企业类库不同，因此架构模式也不同，spring是鼓励企业在后台程序中使用3层架构模式的。另外以下讲解是我曾经的项目架构设计，如有偏差见谅。



7.3 实现登录功能

7.3.1 项目目录



7.3.2 代码

- pom.xml依赖

```
o <dependencies>
  <!-- 引入spring依赖 -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>6.1.3</version>
  </dependency>
  <!-- 封装: getter setter 有参构造 无参构造 -->
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.30</version>
  </dependency>
  <!-- mysql驱动 -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.25</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
  </dependency>
</dependencies>
```

- pojo层

- ```
@Data
public class Admin {
 private String admin_name;
 private String admin_password;
}
```

- dao层

- ```
public interface AdminDao {  
    Admin login(String admin_name, String admin_password);  
}  
  
package com.zretc.dao.impl;  
  
import com.zretc.dao.AdminDao;  
import com.zretc.pojo.Admin;  
import com.zretc.util.DBUtil;  
import java.sql.*;  
  
public class AdminDaoImpl implements AdminDao {  
    private Connection conn;  
    private PreparedStatement ps;  
    private ResultSet rs;  
    @Override  
    public Admin login(String admin_name, String admin_password) {  
        try {  
            conn = DBUtil.getConn();  
            // 查询  
            ps = conn.prepareStatement("select * from admin where  
admin_name=? and admin_password=?");  
            ps.setString(1, admin_name);  
            ps.setString(2, admin_password);  
            rs = ps.executeQuery();  
            if (rs.next()) {  
                Admin admin = new Admin();  
                admin.setAdmin_name(rs.getString(1));  
                admin.setAdmin_password(rs.getString(2));  
                return admin;  
            }  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
  
        return null;  
    }  
}
```

- service 层

- ```
public interface AdminService {
 Admin login(String admin_name, String admin_password);
}
```

```

}

package com.zretrc.service.impl;

import com.zretrc.dao.AdminDao;
import com.zretrc.dao.impl.AdminDaoImpl;
import com.zretrc.pojo.Admin;
import com.zretrc.service.AdminService;

public class AdminServiceImpl implements AdminService {
 AdminDao dao;

 public void setDao(AdminDaoImpl dao) {
 this.dao = dao;
 }

 @Override
 public Admin login(String admin_name, String admin_password) {
 return dao.login(admin_name, admin_password);
 }
}

```

- DBUtil

```

o package com.zretrc.util;

import java.sql.*;

public class DBUtil {
 private static Connection conn;
 private static PreparedStatement ps;
 private static ResultSet rs;

 // 获取数据库连接对象
 public static Connection getConn() {
 try {
 // 加载驱动
 Class.forName("com.mysql.cj.jdbc.Driver");
 conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/zly", "root",
"root");
 return conn;
 } catch (ClassNotFoundException | SQLException e) {
 throw new RuntimeException(e);
 }
 }

 // 关闭资源
 public static void closeAll(ResultSet rs, PreparedStatement ps,
Connection conn) {
 try {
 if (rs != null) {
 rs.close();
 }
 if (ps != null) {
 ps.close();
 }
 }
 }
}

```

```
 }
 if (conn != null) {
 conn.close();
 }
 } catch (SQLException e) {
 throw new RuntimeException(e);
 }
}
```

- application.xml 配置文件

- ```
o   <?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">

        <!-- 创建dao -->
        <bean id="dao" class="com.zre.tc.dao.impl.AdminDaoImpl"/>
        <!-- 创建service -->
        <bean id="service" class="com.zre.tc.service.impl.AdminServiceImpl">
            <!-- 将dao依赖注入到service
                name 属性对应的是实现类中的成员变量的名字
                ref 对应的是bean组件的id
            -->
            <property name="dao" ref="dao"/>
        </bean>
    </beans>
```

- controller层

- ```
• package com.zretc.controller;

import com.zretc.pojo.Admin;
import com.zretc.service.impl.AdminServiceImpl;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class AdminTest {
 // 登录
 @Test
 public void test01(){
 ApplicationContext ctxt = new
ClassPathXmlApplicationContext("applicationContext2.xml");
 // 访问service
 AdminServiceImpl service = (AdminServiceImpl)
ctxt.getBean("service");
 Admin admin = service.login("tom","123");
 if (admin != null){
 System.out.println("登录成功");
 }else{
 System.out.println("登录失败");
 }
 }
}
```

```
 }
}
```

- 运行结果

- 

```
✓ Tests passed: 1 of 1 test - 1 sec 53 ms
"C:\Program Files\Java\jdk-
登录成功
```

## 8、注解配置文件

### 8.1 实例化bean

Spring3.0后为我们引入了组件自动扫描机制，它可以在类路径底下寻找标注了@Component、@Service、@Controller、@Repository注解的类，并把这些类纳入进spring容器中管理。它的作用和在xml文件中使用bean节点配置组件是一样的。

@Component：泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注，例如pojo下

@Service：用于标注业务层组件,service

@Controller：用于标注控制层组件, controller

@Repository：用于标注数据访问组件，dao

```
@Component(value="dog")
public class Dog {
```

```
<bean id="dog" class="com.chinasofti.bean.Dog">
</bean>
```

value对应的就是bean标签中的id,如果不输入id，则默认值当前类首字母小写

- 要使用自动扫描机制，我们需要打开以下配置信息：

- 1、引入AOP的jar包（之前的版本不用）。

```
<dependencies>
 <dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-context</artifactId>
 <version>6.1.3</version>
 </dependency>
 <dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
 <version>8.0.33</version>
 </dependency>
 <dependency>
 <groupId>org.projectlombok</groupId>
 <artifactId>lombok</artifactId>
 <version>1.18.30</version>
 </dependency>
 <dependency>
 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <version>4.13.2</version>
 </dependency>
```

```
</dependencies>
```

- 2、spring配置文件增加context命名空间与xsd的引用，使用注解，需要context组件解析配置文件。

```
■ <?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"

 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">

 <!-- 动态扫描指定包下的所有spring相关的注解 -->
 <context:component-scan base-package="com.zretec.*"/>
</beans>
```

- 其中base-package为需要扫描的包(含子包)。

- 3、增加context的动态扫描

```
■ <context:component-scan base-package="com.chinasofti.bean"></context:component-scan>
```

## 8.2 注解依赖注入

### 8.2.1 setter方法注入

#### 基本类型setter方法注入

- @Value注解

- ```
@Value(value = "18")
private Long age;
```

bean组件类型setter方法注入

- @Resoure注解

- ```
<!-- 初始化和销毁方法--@Resource -->
<dependency>
 <groupId>javax.annotation</groupId>
 <artifactId>javax.annotation-api</artifactId>
 <version>1.3.2</version>
</dependency>
```

- ```
@Resource(name="setterBasic")
private SetterBasic setterBasic;
```

- 在属性上添加注解完成注入，可以省略setter方法

8.2.2 自动匹配

- 默认情况下按照类型匹配
 @Autowired // 根据类型注入，必须找到一个符合条件的对象，否则就会抛出异常
 @Autowired(required = false) // 如果找不到对应的类型就返回null
 AdminDao dao;

- // 按照名字匹配
 @Autowired
 @Qualifier("dao") // dao必须是到层对应的value值
 AdminDao dao;

```
12 usages
@Data
@Component // 相当于在配置文件中写了: <bean id="admin" class="com.zreetc.pojo.Admin'>
public class Admin {
    private String admin_name;
    private String admin_password;
}

@Repository("dao") // 相当于在配置文件中写了: <bean id="dao" class="com.zreetc.dao.impl.AdminDaoImpl'>
@Repository // 相当于在配置文件中写了: <bean id="adminDaoImpl" class="com.zreetc.dao.impl.AdminDaoImpl'>
public class AdminDaoImpl implements AdminDao {
    ● 2 usages

    @Service("service")
    public class AdminServiceImpl implements AdminService {
        // 依赖注入
        @Resource(name="dao")
        // 自动匹配
        // @Autowired // 根据类型注入，必须找到一个符合条件的对象，否则就会抛出异常
        // @Autowired(required = false) // 如果找不到就返回null
        // 按名字匹配
        @Autowired(required = false)
        @Qualifier("dao")
        AdminDao dao;

        public Admin login(String admin_name, String admin_password) {
            Admin admin = dao.login(admin_name, admin_password);
            if (admin != null) {
                System.out.println("登录成功");
            } else {
                System.out.println("登录失败");
            }
            return admin;
        }
    }

    public class AdminTest {
        // 登录
        @Test
        public void test01() {
            ApplicationContext ctxt = new ClassPathXmlApplicationContext(configLocation: "applicationContext2.xml");
            // 访问service
            AdminServiceImpl service = (AdminServiceImpl) ctxt.getBean(name: "service");
            Admin admin = service.login(admin_name: "tom", admin_password: "123"); // 对应的是@Service的值
            if (admin != null) {
                System.out.println("登录成功");
            } else {
                System.out.println("登录失败");
            }
        }
    }
}
```

```
@Data
@Component // 相当于在配置文件中写了: <bean id="admin"
class="com.zreetc.pojo.Admin'>
public class Admin {
    private String admin_name;
    private String admin_password;
}
```

```

@Repository("dao") // 相当于在配置文件中写了: <bean id="dao"
class="com.zretrc.dao.impl.AdminDaoImpl'>
//@Repository // 相当于在配置文件中写了: <bean id="adminDaoImpl"
class="com.zretrc.dao.impl.AdminDaoImpl'>
public class AdminDaoImpl implements AdminDao {
    ...
}

@Service("service")
public class AdminServiceImpl implements AdminService {
    // 依赖注入
    // @Resource(name="dao")

    // 自动匹配
    // @Autowired // 根据类型注入, 必须找到一个符合条件的对象, 否则就会抛出异常
    // @Autowired(required = false) // 如果找不到就返回null

    // 按照名字匹配
    @Autowired(required = false)
    @Qualifier("dao")
    AdminDao dao;
    ...
    ...
}

@Test
public void test01(){
    ApplicationContext ctxt = new
ClassPathXmlApplicationContext("applicationContext2.xml");
    // 访问service
    AdminServiceImpl service = (AdminServiceImpl) ctxt.getBean("service");
    Admin admin = service.login("tom", "123");
    if (admin != null){
        System.out.println("登录成功");
    }else{
        System.out.println("登录失败");
    }
}

```

8.3 初始化和销毁方法配置

`<bean id="dog" class="com.chinasofti.bean.Dog"
scope="singleton" lazy-init="true"
init-method="init"
destroy-method="destory">`

`@PostConstruct`
`private void init() {`
 `System.out.println("调用初始化方法");`

`@PreDestroy`
`private void destory() {`
 `System.out.println("调用销毁方法");`

- 导入依赖

```
○ <!-- 初始化和销毁方法 -->
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```

```
● @Data
@Component // <bean id="dog" class="com.zretc.pojo.Dog"/>
public class Dog {
    // 名字
    @Value("旺财")
    private String dog_name;
}

@Data
@Component
@Scope("prototype") // 作用域, prototype多实例的
public class Boy {
    @Value("小强")
    private String boy_name;
    // 一个男孩有一条狗
    @Autowired
    private Dog dog;

    // 初始化方法
    @PostConstruct // 单实例时在创建对象后执行一次, 多实例时每一次调用getBean()时都会
    执行
    public void initMethod(){
        System.out.println("初始化方法。。。");
    }

    // 销毁方法
    @PreDestroy // 当作用域为多实例时, 不执行销毁方法, 为单实例时, 关闭容器前执行销毁方
    法
    public void destroyMethod(){
        System.out.println("销毁方法");
    }
}

public class MainTest {
    @Test
    public void test01(){
        ClassPathXmlApplicationContext ctxt = new
        ClassPathXmlApplicationContext("applicationContext.xml");
        // 获取Dog
        Dog dog = (Dog) ctxt.getBean("dog");
        // 获取Boy
        Boy boy = (Boy) ctxt.getBean("boy");
        Boy boy2 = (Boy) ctxt.getBean("boy");
        System.out.println(boy.getBoy_name() + "的狗叫" + dog.getDog_name());
        ctxt.close();
    }
}
```

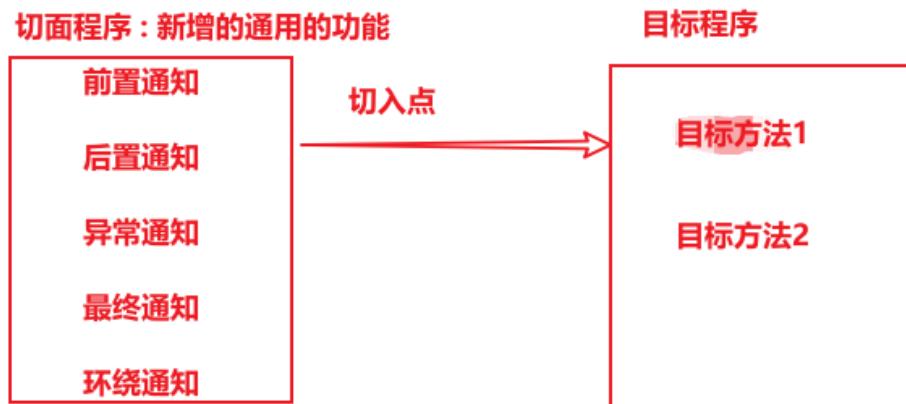
```
}
```

```
• ✓ Tests passed: 1 of 1 test – 694 ms  
"C:\Program Files\Java\jdk-17\bin\java.exe" ...  
初始化方法。。  
初始化方法。。  
小强的狗叫旺财
```

9、AOP(面向切面编程)

AOP的核心组件：

- **切面 (Aspect)**：切面是封装通用业务逻辑的组件，可以作用到其他组件上。
- **切入点 (Pointcut)**：用于指定哪些组件哪些方法使用切面组件，Spring提供表达式来实现该指定。
- **通知 (Advice)**：用于指定组件作用到目标组件的具体位置。
 - AOP前置通知：在目标组件的方法执行前执行的程序。
 - AOP后置通知：在目标组件的方法正常执行并返回参数后执行的程序。
 - AOP异常通知：在目标组件的方法抛出异常信息后执行的程序
 - AOP最终通知：在目标组件的方法正常执行后执行，或在异常通知之前执行。
 - AOP环绕通知：切面程序负责调用目标组件的运行，与struts中的拦截器功能类似，**可以完全取代之前的几个通知**。



通知：新增的功能，在原本功能之前执行还是之后执行，说白了就是执行的位置。

切入点：就是关联目标程序中的目标方法（原功能）使用切面程序中的通知

9.1 代码演示：

引入依赖

```
<dependencies>  
    <dependency>  
        <groupId>org.springframework</groupId>
```

```

<artifactId>spring-context</artifactId>
<version>6.1.3</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
</dependency>
<!-- AOP面向切面编程依赖 -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.9.1</version>
</dependency>
</dependencies>

```

配置头文件，加入aop的命名空间

```

<?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:context="http://www.springframework.org/schema/context"
       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/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">
</beans>

```

增加了AOP的命名空间与AOP的xsd规范，如不需要注解，可以省略context.xsd

目标程序

```

// 目标程序
public class Target {
    // 目标方法
    public String save(String name){
        System.out.println("目标方法被执行了。。。");
        return name;
    }
}

```

切面程序：存储各种通知

在类型为环绕通知的切面程序函数中，参数为org.aspectj.lang.ProceedingJoinPoint是JoinPoint的子类，

扩展了JoinPoint类，提供了proceed()函数，该函数的作用是调用目标组件，并返回目标组件返回的值。

```
package com.zretc.pojo;
```

```

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;

// 切面程序
public class AspectTest {
    // 通知： 新功能在目标方法的具体执行位置。
    // 前置通知： 在目标方法之前执行就是前置通知
    public void doBefore(){
        System.out.println("前置通知。。。");
    }

    // 后置通知： 在目标方法之后执行就是后置通知，如果目标方法发生异常则不执行
    // Joinpoint 该接口中封装了目标程序中目标方法的相关信息
    // retval 返回值
    public void doAfterReturn(JoinPoint jp, Object retval){
        System.out.println("后置通知。。。" + retval);
        System.out.println("目标方法中的参数：" + jp.getArgs()[0]);
        System.out.println("获取目标对象：" + jp.getTarget());
        System.out.println("获取目标方法的反射对象：" + jp.getSignature());
    }

    // 最终通知：在目标方法执行完以后执行，一定会执行的。
    public void doAfter(){
        System.out.println("最终通知。。。");
    }

    // 异常通知：在目标方法发生异常时执行
    public void doException(Exception e){ // e 就是目标方法发生的异常对象
        System.out.println("异常通知。。。");
    }

    // 环绕通知：可以代替以上几种通知
    public void doAround(ProceedingJoinPoint pjp){
        try {
            System.out.println("前置通知");
            // 目标对象
            Object obj = pjp.proceed();
            System.out.println("后置通知");
        } catch (Throwable e) {
            System.out.println("异常通知");
        } finally {
            System.out.println("最终通知");
        }
    }

    // 通知总结：前置通知，目标方法，如果有异常执行异常通知，不执行后置通知，最终通知
    // 如果没有异常，执行后置通知，不执行异常通知
}

```

配置文件，配置通知

```

<?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"
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">

<!-- 创建目标对象 -->
<bean id="target" class="com.zretc.pojo.Target"/>
<!-- 创建切面程序 -->
<bean id="aspectTest" class="com.zretc.pojo.AspectTest"/>

<!-- 配置AOP -->
<aop:config>
    <!-- 配置切入点 : 关联目标程序中的目标方法
        expression 切入点的表达式
        execution 函数式表达式, 第一个* 代表的是返回值的类型是任意的,
                  第二个* 代表的是参数的类型是任意的
    -->
    <aop:pointcut id="pointcut"
        expression="execution(* com.zretc.pojo.Target.save(*))"/>
    <!-- 配置切面 -->
    <aop:aspect ref="aspectTest">
        <!-- 通知: 新功能执行的具体位置 -->
        <!-- 前置通知 -->
        <!-- <aop:before method="doBefore" pointcut-ref="pointcut"/>-->
        <!-- 后置通知 -->
        <!-- <aop:after-returning method="doAfterReturn" returning="retval"-->
        <!-- 最终通知 -->
        <!-- <aop:after method="doAfter" pointcut-ref="pointcut"/>-->
        <!-- 异常通知 -->
        <!-- <aop:after-throwing method="doException" pointcut-ref="pointcut"-->
        <!-- throwing="e"/>-->
        <!-- 环绕通知 -->
        <aop:around method="doAround" pointcut-ref="pointcut"/>
    </aop:aspect>
</aop:config>
</beans>
```

测试类：

```
public class MainTest {  
    @Test  
    public void test01(){  
        ApplicationContext ctxt = new  
ClassPathXmlApplicationContext("applicationContext.xml");  
        // 获取目标程序  
        Target target = (Target) ctxt.getBean("target");  
        System.out.println(target.save("小强"));  
    }  
}
```

```
✓ Tests passed: 1 of 1 test - 859 ms  
"C:\Program Files\Java\jdk-17\bin\java.e  
前置通知  
目标方法被执行了。。。  
后置通知  
最终通知  
null|
```

9.2 通知总结

执行顺序:

发生异常:

前置通知
目标方法
异常通知
最终通知

未发生异常:

前置通知
目标方法
后置通知
最终通知

注意: 1、异常通知与后置通知不能同时存在
2、如果使用环绕通知就把前几种通知注释掉, 两种方式选择其中一种

9.3 切入点

- 切入点表达式用于声明spring容器中哪些组件的函数是目标函数, 也就是切面程序要作用到哪些组件的哪些函数上。
- 常用的切入点表达式分为:
 - 按类匹配: 匹配的java类中全部函数作为目标函数, 使用 `within` 关键字。

- 按**函数匹配**: 匹配的函数作为目标函数, 使用 execution 关键字。
- 按**bean的id匹配**: 匹配的bean中全部函数作为目标组件。使用 bean 关键字

9.3.1 按类匹配

匹配到类

```
<aop:pointcut id="targetPintcut" expression="within(com.chinasofti.Target)"/>
```

匹配到包下的类

```
<aop:pointcut id="targetPintcut" expression="within(com.chinasofti.*)"/>
```

匹配到包下及子包下的类

```
<aop:pointcut id="targetPintcut" expression="within(com..*)"/>
```

9.3.2 按函数匹配

完整写法

返回类型 类的路径 类名 函数名 参数类型 (用, 分开)

```
<aop:pointcut id="targetPintcut" execution(string  
com.chinasofti.Target.save(String)) />
```

任意返回类型

```
<aop:pointcut id="targetPintcut" execution(* com.chinasofti.Target.save(String))  
/>
```

任意返回类型下指定包下任意类

```
<aop:pointcut id="targetPintcut" execution(* com.chinasofti.*.save(String)) />
```

任意返回类型下指定包下任意类任意函数

```
<aop:pointcut id="targetPintcut" execution(* com.chinasofti.*.*(String)) />
```

任意返回类型下指定包或子包下任意类任意函数任意参数

```
<aop:pointcut id="targetPintcut" execution(* com..*.*(..)) />
```

9.3.3 按bean的id匹配

根据bean组件名称匹配

```
<aop:pointcut id="targetPintcut" expression="bean(target)"/>
```

根据bean组件名称 (含通配符) 匹配

```
<aop:pointcut id="targetPintcut" expression="bean(target*)"/>
```

代码演示:

```
<?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"  
       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">
```

```

<!-- 创建目标对象 -->
<bean id="target" class="com.zretc.pojo.Target"/>

<!-- 创建切面程序 -->
<bean id="aspectTest" class="com.zretc.pojo.AspectTest"/>

<!-- 配置AOP -->
<aop:config>
    <!-- 配置切入点： 关联目标程序中的目标方法 -->
    <!-- <aop:pointcut id="pointcut" expression="execution(* com.zretc.pojo.Target.save(*))"/>-->
        <!-- 按照函数匹配：
            返回值是任意的，com包及其子包下的任意类中的任意方法，参数的个数类型是任意的
            -->
    <!-- <aop:pointcut id="pointcut" expression="execution(* com..*.*(..))"/>-->
        <!-- 按照类匹配：
            com 这个包及其子包下的任意类
            -->
    <!-- <aop:pointcut id="pointcut" expression="within(com..*)"/>-->

        <!-- 按照bean匹配
            bean组件的id 以 target开头的
            -->
        <aop:pointcut id="pointcut" expression="bean(target*)"/>
        <!-- 配置切面 -->
        <aop:aspect ref="aspectTest">
            <!-- 通知： 新功能执行的具体位置 -->
            <!-- 前置通知 -->
            <!-- <aop:before method="doBefore" pointcut-ref="pointcut"/>-->
            <!-- 后置通知 -->
            <!-- <aop:after-returning method="doAfterReturn" returning="retVal"-->
            >
            <!-- pointcut-ref="pointcut"/>-->
            <!-- 最终通知 -->
            <!-- <aop:after method="doAfter" pointcut-ref="pointcut"/>-->
            <!-- 异常通知 -->
            <!-- <aop:after-throwing method="doException" pointcut-ref="pointcut"-->
            >
            <!-- throwing="e"/>-->

            <!-- 环绕通知 -->
            <aop:around method="doAround" pointcut-ref="pointcut"/>
        </aop:aspect>
    </aop:config>
</beans>

```

9.4 AOP的注解配置

@Aspect : 切面

@Before("execution(* com..*(..))") 前置通知

```
@AfterReturning(pointcut = "within(com..*)",returning = "retVal") 后置通知
```

```
@After("bean(target)") 最终通知
```

```
@AfterThrowing(pointcut = "within(com..*)",throwing = "e") 异常通知
```

```
@Around("within(com..*)") 环绕通知
```

```
<?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"
    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">

    <!-- 动态扫描 -->
    <context:component-scan base-package="com.zretc.*"/>
    <!-- 自动代理 AOP 相关的注解 -->
    <aop:aspectj-autoproxy/>
</beans>
```

目标程序:

```
package com.zretc.pojo;

import org.springframework.stereotype.Component;

// 目标程序
@Component
public class Target { // 默认bean的id 是类名的首字母小写
    // 目标方法
    public String save(String name) {
        System.out.println("目标方法被执行了。。。");
        //      int a = 10 / 0;
        return name;
    }
}
```

切面程序

```
package com.zretc.pojo;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;

// 切面程序
```

```

@Aspect
@Component
public class AspectTest {
    // 通知： 新功能在目标方法的具体执行位置。
    // 前置通知：在目标方法之前执行就是前置通知
    @Before("execution(* com..*(..))") // 按照函数匹配，返回值是任意的，com包及其子包下的任意类中的任意方法，参数是任意的
    public void doBefore(){
        System.out.println("前置通知。。。");
    }

    // 后置通知：在目标方法之后执行就是后置通知，如果目标方法发生异常则不执行
    // Joinpoint 该接口中封装了目标程序中目标方法的相关信息
    // retval 返回值
    @AfterReturning(pointcut = "within(com..*)",returning = "retval")
    public void doAfterReturn(JoinPoint jp, Object retval){
        System.out.println("后置通知。。。" + retval);
        System.out.println("目标方法中的参数：" + jp.getArgs()[0]);
        System.out.println("获取目标对象：" + jp.getTarget());
        System.out.println("获取目标方法的反射对象：" + jp.getSignature());
    }

    // 最终通知：在目标方法执行完以后执行，一定会执行的。
    @After("bean(target)")
    public void doAfter(){
        System.out.println("最终通知。。。");
    }

    // 异常通知：在目标方法发生异常时执行
    @AfterThrowing(pointcut = "within(com..*)",throwing = "e")
    public void doException(Exception e){ // e 就是目标方法发生的异常对象
        System.out.println("异常通知。。。");
    }

    // 环绕通知：可以代替以上几种通知
    //     @Around("within(com..*)")
    //     public void doAround(ProceedingJoinPoint pjp){
    //         try {
    //             System.out.println("前置通知");
    //             // 目标对象
    //             Object obj = pjp.proceed();
    //             System.out.println("后置通知");
    //         } catch (Throwable e) {
    //             System.out.println("异常通知");
    //         } finally {
    //             System.out.println("最终通知");
    //         }
    //     }

    // 通知总结：前置通知，目标方法，如果有异常执行异常通知，不执行后置通知，最终通知
    // 如果没有异常，执行后置通知，不执行异常通知
}

```

10、spring集成jdbc

Spring提供了一个**工具类JdbcTemplate**，该类对jdbc的操作进行了轻量级别的封装。

优点如下：

- 直接使用sql语句操作数据库，效率很高。
- 支持数据库的分区，这是数据量大的项目的实现方案，hibernate无法实现。
- 使用java语言拼sql语句，效率很高，这是ibatis无法达到的。

缺点如下：

- sql语句直接写在java程序中，很难管理，但部分企业进行了封装，实现了sql语句的简单管理。编码量大。

10.1 JdbcTemplate

JdbcTemplate简介：封装了操作数据库的各种方法，该类包含一个**dataSource属性（数据源）**，只有在初始化数据源的情况下才能调用JdbcTemplate的方法。

数据源：数据源为主流连接池的数据源对象（例如C3P0，DBCP数据库连接池），因此在使用JdbcTemplate前，要创建数据源对象。



10.1.1 实现步骤

引入依赖

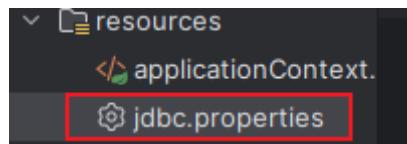
```
<dependencies>
    <!-- spring 依赖 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>6.1.3</version>
    </dependency>
    <!-- 驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>
    <!-- 数据源 -->
```

```

<dependency>
    <groupId>c3p0</groupId>
    <artifactId>c3p0</artifactId>
    <version>0.9.1.2</version>
</dependency>
<!-- JdbcTemplate 依赖 -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>6.1.3</version>
</dependency>
<!-- 测试 -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
</dependency>
</dependencies>

```

创建属性文件:



```

jdbc.driver=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/zly
jdbc.user=root
jdbc.password=root

```

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"
       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">

    <!-- 加载jdbc.properties文件 -->
    <context:property-placeholder location="jdbc.properties"/>
    <!-- 数据源 c3p0 -->
    <bean id="ds" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="jdbcUrl" value="${jdbc.url}"/>
        <property name="driverClass" value="${jdbc.driver}"/>
        <property name="user" value="${jdbc.user}"/>
        <property name="password" value="${jdbc.password}"/>
    
```

```

<property name="initialPoolSize" value="3"/>
<property name="maxPoolSize" value="10"/>
<property name="minPoolSize" value="1"/>
<property name="acquireIncrement" value="3"/>
<property name="maxIdleTime" value="60"/>
</bean>

<!-- JdbcTemplate工具类 对数据库中数据进行增删改查 -->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <!-- 依赖注入数据源 -->
    <property name="dataSource" ref="ds"/>
</bean>

<!-- 创建dao层对象 -->
<bean id="deptDao" class="com.zretec.dao.DeptDao">
    <!-- 依赖注入JdbcTemplate工具类 -->
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
</bean>

<!-- 创建service层对象 -->
<bean id="deptService" class="com.zretec.service.DeptService">
    <!-- 依赖注入dao -->
    <property name="deptDao" ref="deptDao"/>
</bean>
</beans>

```

实体类:

```

@Data
public class Dept {
    private Integer dept_id;
    private String dept_name;
    private String dept_loc;
}

```

Dao层接口

```

public interface DeptDao {
    int insertDept(Dept dept);
    int deleteDept(Integer dept_id);
    int updateDept(Dept dept);
    List<Dept> selectAll();
    Dept selectById(Integer dept_id);
}

```

10.1.2 JdbcTemplate常用API接口

- update: 方法可以执行一次sql语句完成数据更新操作（增删改）

```
int update(String sql, Object[] obs);
```

- 返回值: 执行本次操作更新的数据数量。

- 参数说明:

sql: sql语句, 可以带预处理。

Object[]: 预处理参数。 (可选)

- Object query(String sql, Object[] obs, ResultSetExtractor rs)

- 返回值: 返回rs接口的实现类中实现方法返回值 (一般用来查询一个对象)。

- 参数说明:

sql: sql语句, 可以带预处理。

Object[]: 预处理参数。 (可选)

rs: 开发者需要实现的接口, 接口中包含一个抽象方法

```
public Object extractData(ResultSet rs) throws SQLException,  
    DataAccessException {  
    return null;  
}
```

参数说明:

rs: ResultSet对象, 由spring容器在执行sql时创建, 并传递到该方法中, 开发者根据rs获取对象并封装数据, 最后返回封装好的数据。

- List query(String sql, Object[] obs, RowMapper rm)

- 参数说明:

sql: sql语句, 可以带预处理。

Object[]: 预处理参数。 (可选)

rm: 开发者需要实现的接口, 接口中包含一个抽象方法



```
public Object mapRow(ResultSet rs, int arg1)
```

mapRow的参数说明:

rs: 正在迭代的ResultSet。

arg1: 已经迭代了多少次。

返回值: 将rm接口的实现类中实现方法的返回值封装到一个List中, 并返回。

Dao层实现类

```
package com.zretc.dao;

import com.zretc.pojo.Dept;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

public class DeptDaoImpl implements DeptDao {
    private JdbcTemplate jdbcTemplate;

    public JdbcTemplate getJdbcTemplate() {
        return jdbcTemplate;
    }

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Override
    public int insertDept(Dept dept) {
        return jdbcTemplate.update("insert into dept
values(default,?,?)",dept.getDept_name(),dept.getDept_loc());
    }

    @Override
    public int deleteDept(Integer dept_id) {
        return jdbcTemplate.update("delete from dept where dept_id=?",dept_id);
    }

    @Override
    public int updateDept(Dept dept) {
        return jdbcTemplate.update("update dept set dept_name=?,dept_loc=? where
dept_loc=?",
                dept.getDept_name(),dept.getDept_loc(),dept.getDept_id());
    }

    @Override
    public List<Dept> selectAll() {
        return jdbcTemplate.query("select * from dept", new RowMapper<Dept>() {
            @Override
            public Dept mapRow(ResultSet rs, int rowNum) throws SQLException {
                Dept dept = new Dept();
                dept.setDept_id(rs.getInt(1));
                dept.setDept_name(rs.getString(2));
                dept.setDept_loc(rs.getString(3));
                return dept;
            }
        });
    }

    @Override
    public Dept selectById(Integer dept_id) {
        // 注意：第二个参数必须是数组，不是可变参数
    }
}
```

```
        return jdbcTemplate.query("select * from dept where dept_id=?", new
Object[dept_id], new ResultSetExtractor<Dept>() {
    @Override
    public Dept extractData(ResultSet rs) throws SQLException,
DataAccessException {
        if (rs.next()){
            Dept dept = new Dept();
            dept.setDept_id(rs.getInt(1));
            dept.setDept_name(rs.getString(2));
            dept.setDept_loc(rs.getString(3));
            return dept;
        }
        return null;
    }
});

}

}
```

service层接口

```
public interface DeptService {
    int insertDept(Dept dept);
    int deleteDept(Integer dept_id);
    int updateDept(Dept dept);
    List<Dept> selectAll();
    Dept selectById(Integer dept_id);
}
```

service层实现类

```
package com.zrecc.service;

import com.zrecc.dao.DeptDaoImpl;
import com.zrecc.pojo.Dept;

import java.util.List;

public class DeptServiceImpl implements DeptService{
    DeptDaoImpl deptDao;

    public DeptDaoImpl getDeptDao() {
        return deptDao;
    }

    public void setDeptDao(DeptDaoImpl deptDao) {
        this.deptDao = deptDao;
    }

    @Override
    public int insertDept(Dept dept) {
        return deptDao.insertDept(dept);
    }
}
```

```

@Override
public int deleteDept(Integer dept_id) {
    return deptDao.deleteDept(dept_id);
}

@Override
public int updateDept(Dept dept) {
    return deptDao.updateDept(dept);
}

@Override
public List<Dept> selectAll() {
    return deptDao.selectAll();
}

@Override
public Dept selectById(Integer dept_id) {
    return deptDao.selectById(dept_id);
}
}

```

配置文件中创建dao层对象和service层对象，病依赖注入

```

<!-- 创建dao层对象 -->
<bean id="deptDao" class="com.zretrc.dao.DeptDaoImpl">
    <!-- 依赖注入JdbcTemplate工具类 -->
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
</bean>

<!-- 创建service层对象 -->
<bean id="deptService" class="com.zretrc.service.DeptServiceImpl">
    <!-- 依赖注入dao -->
    <property name="deptDao" ref="deptDao"/>
</bean>

```

测试类：

```

public class MainTest {
    @Test
    public void test01(){
        ApplicationContext ctxt = new
ClassPathXmlApplicationContext("applicationContext.xml");
        // 访问service
        DeptServiceImpl service = (DeptServiceImpl) ctxt.getBean("deptService");
        int rows = service.deleteDept(9);
        if (rows > 0){
            System.out.println("删除成功");
        }else{
            System.out.println("删除失败");
        }
        List<Dept> depts = service.selectAll();
        depts.forEach(dept -> System.out.println(dept));
    }
}

```

```
}
```

```
✓ Tests passed: 1 of 1 test - 1 sec 296 ms
```

```
删除成功
```

```
Dept(dept_id=1, dept_name=销售部, dept_loc=一楼)
```

```
Dept(dept_id=2, dept_name=市场部, dept_loc=二楼)
```

```
Dept(dept_id=7, dept_name=行政部, dept_loc=三楼)
```

10.1.3 注解

```
@Repository
public class DeptDaoImpl implements DeptDao {
    @Autowired
    private JdbcTemplate jdbcTemplate;
    .....

@Service("deptService")
public class DeptServiceImpl implements DeptService{
    @Autowired
    DeptDaoImpl deptDao;
    .....
```

配置文件:

```
<!-- 动态扫描 -->
<context:component-scan base-package="com.zretec.*"/>
```

11、jdbc的事务控制

11.1 事务概述

事务是一组操作的执行单元，相对于数据库操作来讲，事务管理的是一组SQL指令，比如增加，修改，删除等。**事务的一致性**，要求，这个事务内的操作必须全部执行成功，如果在此过程中出现了差错，比如有一条SQL语句没有执行成功，那么这一组操作都将全部回滚

简单来说：就是将多个sql语句看成是一个整体，要么都执行，要么都不执行。

11.2 事务的四大特性

事务的四大特性ACID：

- atomic(原子性):要么都发生，要么都不发生。

- consistent(一致性):数据应该不被破坏。
- isolate(隔离性):用户间操作不相混淆
- durable(持久性):永久保存,例如保存到数据库中等

张三 要给 李四转账 1000 元

张三 1001 ==> 1

李四 1 ==> 1001

11.3 事务管理方式

Spring提供了两种事务管理方式:

- **编程式事务管理**

- 通过**程序代码**来控制你的事务何时开始, 何时结束等, 结果是控制的颗粒度更细, 但需要手写程序, 另外在团队合作开发时, 会出现事物管理混乱。

Hibernate的事务操作:

```
public void save(){
    Session session = sessionFactory.getCurrentSession();
    session.beginTransaction();
    Info info = new Info("传智播客");
    info.setContent("国内实力最强的java培训机构");
    session.save(info);
    session.getTransaction().commit();
}
```

JDBC的事务操作:

```
Connection conn = null;
try {
    .....
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("update person where name='叶天'");
    conn.commit();
    .....
} catch (Exception e) {
    conn.rollback();
} finally {
    conn.close();
}
```

- **声明式事务管理**

- 在Spring中, 你只需要在**Spring配置文件**中做一些配置, 即可将数据库的访问纳入到事务管理中, 解除了和代码的耦合, 这是对应用代码影响最小的选择。当你不需要事务管理的时候, 可以直接从Spring配置文件中移除该设置

11.4 spring的事务管理器

- **spring的事务管理器:** spring没有直接管理事务, 只是开发了事物管理器调用第三方组件完成事物控制。

事务管理器实现	目标
org.springframework.jdbc.datasource.DataSourceTransactionManager	在单一的JDBC Datasource中的管理事务
org.springframework.orm.hibernate3.HibernateTransactionManager	当持久化机制是hibernate时, 用它来管理事务
org.springframework.jdo.JdoTransactionManager	当持久化机制是Jdo时, 用它来管理事务。
org.springframework.transaction.jta.JtaTransactionManager	使用一个JTA实现来管理事务。在一个事务跨越多个资源时必须使用
org.springframework.orm.obj.PersistenceBrokerTransactionManager	当apache的obj用作持久化机制时, 用它来管理事务。

- Spring控制事物的方式: spring控制事物是以**bean组件的函数为单位的**, 如果一个函数正常执行完毕, 该函数内的全部数据库操作按照一次事物提交, 如果抛出异常, 全部回滚。
- 事物的传播策略: 如两个bean组件都由spring控制事物, 且组件的函数之间存在调用关系, 即(bean1 函数a 调用了 bean2 函数b), spring提供了一组配置方式供开发者选择, 这些配置方式称为事物的传播策略。

传播行为	意义
REQUIRED	业务方法需要在一个事务中运行。如果方法运行时，已经处在一个事务中，那么加入到该事务，否则为自己创建一个新的事务
NOT_SUPPORTED	声明方法不需要事务。如果方法没有关联到一个事务，容器不会为它开启事务。如果方法在一个事务中被调用，该事务会被挂起，在方法调用结束后，原先的事务便会恢复执行
REQUIRESNEW	属性表明不管是否存在事务，业务方法总会为自己发起一个新的事务。如果方法已经运行在一个事务中，则原有事务会被挂起，新的事务会被创建，直到方法执行结束，新事务才算结束，原先的事务才会恢复执行
MANDATORY	该属性指定业务方法只能在一个已经存在的事务中执行，业务方法不能发起自己的事务。如果业务方法在没有事务的环境下调用，容器就会抛出例外。
SUPPORTS	这一事务属性表明，如果业务方法在某个事务范围内被调用，则方法成为该事务的一部分。如果业务方法在事务范围外被调用，则方法在没有事务的环境下执行
Never	指定业务方法绝对不能在事务范围内执行。如果业务方法在某个事务中执行，容器会抛出例外，只有业务方法没有关联到任何事务，才能正常执行
NESTED	如果一个活动的事务存在，则运行在一个嵌套的事务中。如果没有活动事务，则按REQUIRED属性执行。它使用了一个单独的事务，这个事务拥有多个可以回滚的保存点。内部事务的回滚不会对外部事务造成影响。它只对DataSourceTransactionManager事务管理器起效

- 数据库的隔离级别：

- **脏读**:一个事务读取了另一个事务改写但还未提交的数据,如果这些数据被回滚，则读到的数据是无效的。
- **不可重复读**: 在同一事务中，多次读取同一数据返回的结果有所不同。换句话说就是，后续读取可以读到另一事务已提交的更新数据。相反，“可重复读”在同一事务中多次读取数据时，能够保证所读数据一样，也就是，后续读取不能读到另一事务已提交的更新数据。
- **幻读**: 一个事务读取了几行记录后，另一个事务插入一些记录，幻读就发生了。再后来的查询中，第一个事务就会发现有些原来没有的记录。

隔离级别	含义
DEFAULT	使用后端数据库默认的隔离级别(spring中的的选择项)
READ_UNCOMMITTED	允许你读取还未提交的改变了的数据。可能导致脏、幻、不可重复读
READ_COMMITTED	允许在并发事务已经提交后读取。可防止脏读，但幻读和不可重复读仍可发生
REPEATABLE_READ	对相同字段的多次读取是一致的，除非数据被事务本身改变。可防止脏、不可重复读，但幻读仍可能发生。
SERIALIZABLE	完全服从ACID的隔离级别，确保不发生脏、幻、不可重复读。这是在所有的隔离级别中最慢的，它是典型的通过完全锁定在事务中涉及的数据表来完成的。

- 只读事物与读写事物：

- 与所访问的数据库以及数据库驱动程序相关，并不一定是一个强制选项（例如在只读事物中去更新事物时允许的）。但若在事物中声明了只读事物，将会暗示数据库驱动程序和数据库系统，这个事物并不包含更改数据的操作，那么驱动程序和数据库就有可能根据这种情况对该事物进行一些特定的优化，比方说不安排相应的数据库锁，不记录回滚日志等，以减轻事物对数据库的压力，毕竟事物也是要消耗数据库的资源的。

- 使用场景：

- **只读事物：单纯的数据库查询**
- **读写事物：对数据进行增删改的操作。**

11.5 使用spring声明式事物控制分为如下几步

事物用到了AOP需要导入相关依赖

```
<!-- aop面向切面编程 -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.9.1</version>
</dependency>
```

在配置文件头信息上增加tx命名空间与tx.xsd：声明式事物控制需要引入tx标签，因此要引入此xsd文件。

```
<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" <-- tx 命名空间 -->
    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
<-- tx.xsd 引入 -->
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd">
```

```
<?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>
```

配置事物控制管理器：jdbc的事物控制管理器为DataSourceTransactionManager

```
<!-- 事务管理器 -->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- 依赖注入数据源 -->
    <property name="dataSource" ref="ds"/>
</bean>
```

配置事物通知：配置哪些函数委托spring进行事物管理，以及事物管理的隔离级别、传播行为、是否只读事物属性。建议大家在需要更新数据的函数上配置隔离级别为数据库默认级别，传播行为采用required，读写事物。而只是查询的函数使用只读事物，效率更高。□

```

<!-- 配置事物通知，即配置jdbc的事物管理器头 -->
<tx:advice id="trAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="insert*" isolation="DEFAULT" propagation="REQUIRED" read-only="false"/>
        <tx:method name="update*" isolation="DEFAULT" propagation="REQUIRED" read-only="false"/>
        <tx:method name="delete*" isolation="DEFAULT" propagation="REQUIRED" read-only="false"/>
        <tx:method name="find*" read-only="true"/>
        <tx:method name="load*" | read-only="true"/>
    </tx:attributes>
</tx:advice>

```

```

<!-- 配置事物通知 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <!--
            * 代表的是以。。。开的头，例如insertDept insertEmp
            isolation="DEFAULT" 事务的隔离级别是默认的
            propagation="REQUIRED" 事务的传播策略
            read-only="false" 只读还是读写操作
                false代表的是读写，DML
                true 代表的是只读，DQL
        -->
        <tx:method name="insert*" isolation="DEFAULT" propagation="REQUIRED"
read-only="false"/>
            <tx:method name="delete*" isolation="DEFAULT" propagation="REQUIRED"
read-only="false"/>
            <tx:method name="update*" isolation="DEFAULT" propagation="REQUIRED"
read-only="false"/>
            <tx:method name="select*" read-only="true"/>
    </tx:attributes>
</tx:advice>

```

配置事物的切入点：也就是spring的哪些组件要配置事物通知。

在spring的三层架构中，建议把事物控制放在service层。

```

<!-- 配置事物的切入点 -->
<aop:config>
    <!-- 切入点 -->
    <aop:pointcut id="txPointcut" expression="within(com.zretc.service.*)" />
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut" />
</aop:config>

```

service实现类中抛出异常：

在添加的功能中存在删除的功能，同时抛出异常，这样就会被事务管理器拦截，并回滚。

```

@Override
public int insertDept(Dept dept) {
    deletedDept(12);
    // 抛出异常
    int i = 10/0;
    return deptDao.insertDept(dept);
}

```

测试类

```
@Test
public void test02() {
    ApplicationContext ctxt = new
classPathXmlApplicationContext("applicationContext.xml");
    // 访问service
    // 动态代理必须针对接口：
    // DK动态代理的原理是根据定义好的规则，用传入的接口创建一个新类，这就是为什么采用动态
    // 代理时为什么只能用接口引用指向代理，而不能用传入的类引用执行动态类。
    DeptService service = (DeptService) ctxt.getBean("deptServiceImpl");
    Dept dept = (Dept) ctxt.getBean("dept");
    dept.setDept_name("406教室");
    dept.setDept_loc("406");
    service.insertDept(dept);
}
```

WHERE ORDER BY			
	dept_id	dept_name	dept_loc
1		销售部	一楼
2		市场部	二楼
3	12	406教室	406

运行后，抛出异常，数据库没有编号，回滚了

```
信息: Initializing c3p0 pool... com.mchange.v2.c3p0.ComboPooledDataSource [ acquireIncrement -> 3, acq
java.lang.ArithmetricException: / by zero
    at com.zretc.service.DeptServiceImpl.insertDept(DeptServiceImpl.java:27) <4 internal lines>
    at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:351)
```

WHERE ORDER BY			
	dept_id	dept_name	dept_loc
1		销售部	一楼
2		市场部	二楼
3	12	406教室	406

11.6 使用注解完成事物控制

- **@Transactional**// 在接口上声明后。代表这个类下的所有方法都会开启事务
- 配置文件上增加

- ```

<!--1 启用spring的自动扫描功能-->
<context:component-scan base-package="com.zretc.*"/>

<!--5 采用@Transactional注解方式使用事务 -->
<tx:annotation-driven transaction-manager="transactionManager" />

```

```

<?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">

 <!-- 加载jdbc.properties文件 -->
 <context:property-placeholder location="jdbc.properties"/>
 <!-- 数据源 c3p0 -->
 <bean id="ds" class="com.mchange.v2.c3p0.ComboPooledDataSource">
 <property name="jdbcurl" value="${jdbc.url}"/>
 <property name="driverClass" value="${jdbc.driver}"/>
 <property name="user" value="${jdbc.user}"/>
 <property name="password" value="${jdbc.password}"/>
 <property name="initialPoolSize" value="3"/>
 <property name="maxPoolSize" value="10"/>
 <property name="minPoolSize" value="1"/>
 <property name="acquireIncrement" value="3"/>
 <property name="maxIdleTime" value="60"/>
 </bean>

 <!-- JdbcTemplate工具类 对数据库中数据进行增删改查 -->
 <bean id="jdbcTemplate"
 class="org.springframework.jdbc.core.JdbcTemplate">
 <!-- 依赖注入数据源 -->
 <property name="dataSource" ref="ds"/>
 </bean>

 <!-- 事务管理器 -->
 <bean id="transactionManager"
 class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
 <!-- 依赖注入数据源 -->
 <property name="dataSource" ref="ds"/>
 </bean>

 <!-- 动态扫描 -->
 <context:component-scan base-package="com.zretc.*"/>
 <!-- 事务注解的驱动 -->

```

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

- 在业务层的方法上增加@**transactional**注解。

```
//在需要被事物管理的方法上配置注解
@Transactional(propagation=Propagation.REQUIRED,isolation=Isolation.DEFA
ULT,readonly=true)
public void save () throws Exception {
}
```

```
@Transactional // 在接口上声明后。代表这个类下的所有方法都会开启事务
public interface DeptService {
```

```
 @Transactional(isolation = Isolation.DEFAULT,propagation =
Propagation.REQUIRED,readonly = false)
 int insertDept(Dept dept);
 @Transactional(isolation = Isolation.DEFAULT,propagation =
Propagation.REQUIRED,readonly = false)
 int deleteDept(Integer dept_id);
 @Transactional(isolation = Isolation.DEFAULT,propagation =
Propagation.REQUIRED,readonly = false)
 int updateDept(Dept dept);
 @Transactional(readOnly = true)
 List<Dept> selectAll();
 @Transactional(readOnly = true)
 Dept selectById(Integer dept_id);
}
```