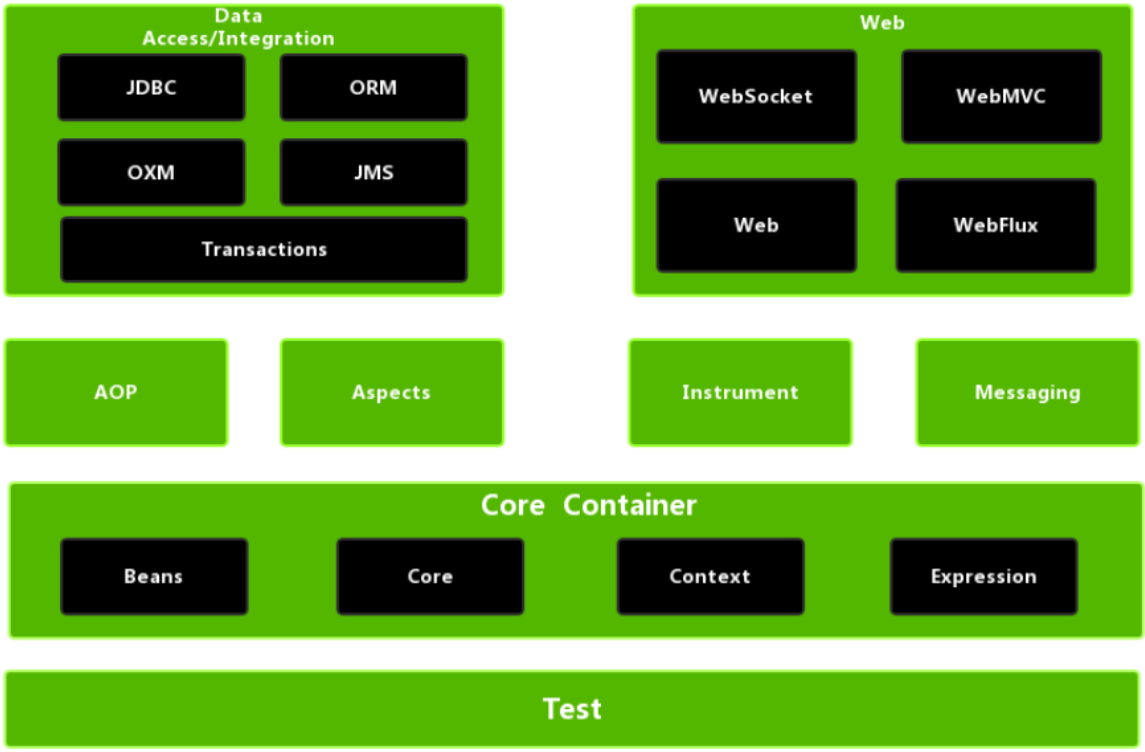
1 Spring框架介绍

1.1 概述

- 定义：轻量级、开源的JavaEE框架。
- 目的：解决企业应用开发的复杂性。

- 两个核心部分：IOC（控制反转）和AOP（面向切面编程）。

1.2 组成部分



我们可以将Spring的组成部分分为八类:

- Data Access/Integration:数据访问或者叫数据整合，主要负责包括对JDBC的封装，对象关系映射，封装JAVA对象与XML之间转换的功能，JAVA消息服务等。
- Web: 网络模块的整合，主要负责包括网络通信，Web相关的开发，反应式Web应用（WebFlux），以及大名鼎鼎的Spring MVC也在这个模块中。
- AOP:面向切面编程，提供AOP（面向切面编程）的实现
- Aspects:提供的对AspectJ框架的整合
- Instrument:整合模块，对服务器的代理接口
- Messaging：为集成messaging api和消息协议提供支持
- Core Container：核心工具包，Spring的重要组成部分，包括对类的处理、应用上下文的处理、以及Spring表达式的处理
- Test：对JUNIT等测试框架的简单封装

1.3 特点

- 方便解耦，简化开发
- AOP编程的支持
- 方便程序的测试，集成Junit
- 方便整合各种其他优秀框架
- 声明式事务的支持
- 降低JavaEE API的使用难度
- Java源码是经典的学习案例

1.4 入门案例

1.4.1 环境搭建

创建Mavn项目，引入Spring5的坐标，当前使用的spring5版本：5.2.8

- **如何引用Spring5的相关jar包?**

访问地址：https://github.com/spring-projects/spring-framework/wiki/Spring-Framework-Artifacts

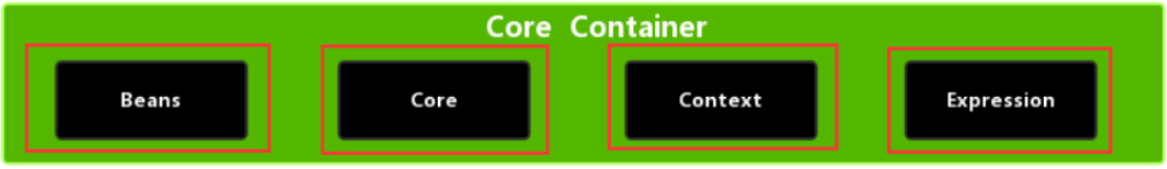
> The Spring Framework is modular and publishes 20+ different jars:

```
spring-aop           spring-context-indexer  spring-instrument  spring-orm    spring-webflux
spring-aspects       spring-context-support  spring-jcl         spring-oxm    spring-webmvc
spring-beans         spring-core             spring-jdbc        spring-test   spring-websocket
spring-beans-groovy  spring-expression       spring-jms         spring-tx
spring-context       spring-framework-bom    spring-messaging   spring-web
```

从图上我们可以看到Spring框架有20多个模块分别在不同的jar包管理，那么我们如果想创建一个简单的spring项目应该引入哪些jar包呢?

需要引入4个核心基础jar包和一个依赖的日志包common-logging、Junit单元测试框架，一共6个包

```
spring-core
spring-aop
spring-context
spring-expression
common-logging
junit
```

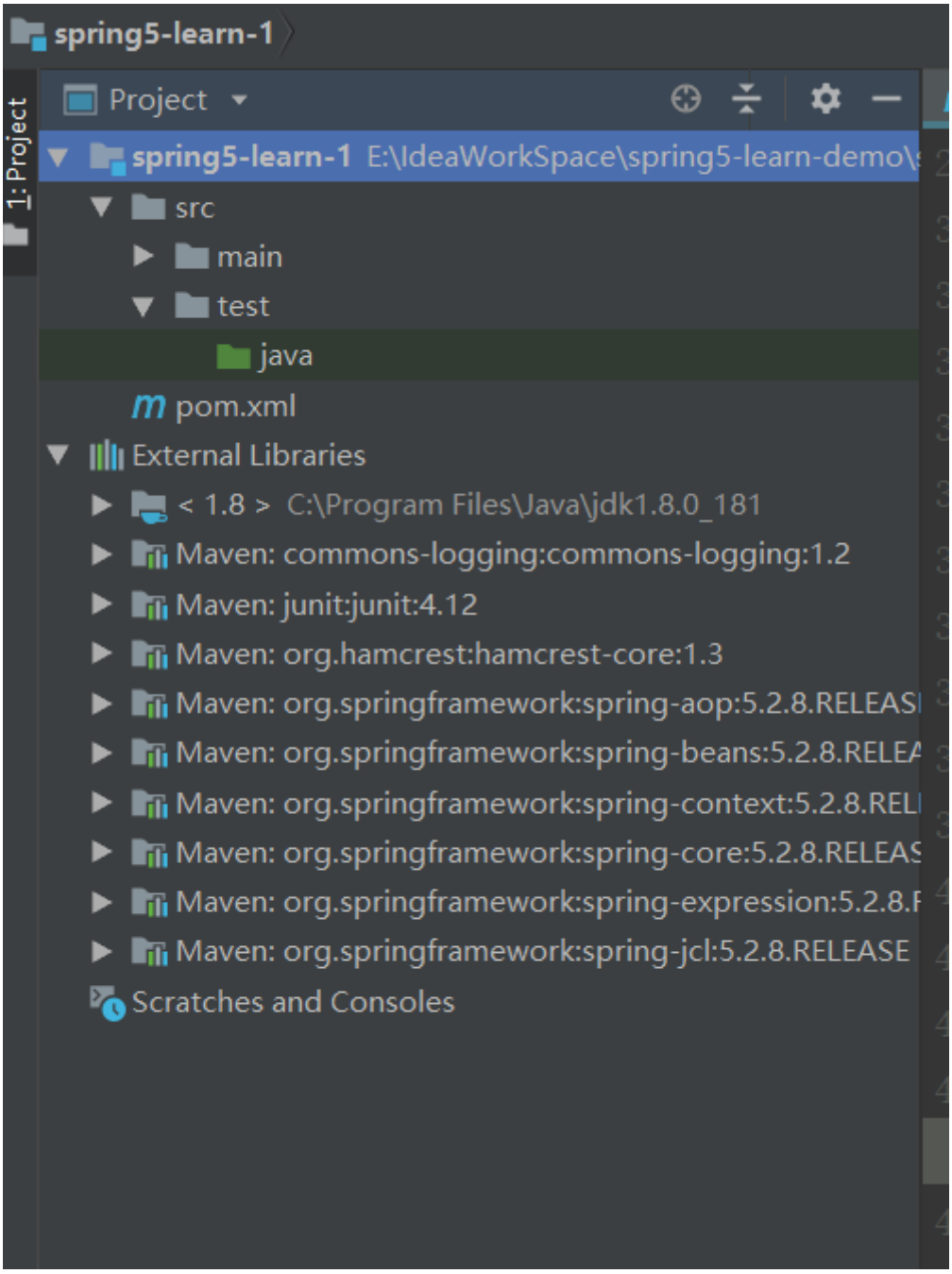


pom具体增加的内容如下:

```
<?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>org.learn.spring5</groupId>
    <artifactId>spring5-learn-1</artifactId>
    <version>1.0-SNAPSHOT</version>

        <dependencies>
            <dependency>
            <!--spring-core-->
                <groupId>org.springframework</groupId>
                <artifactId>spring-core</artifactId>
                <version>5.2.8.RELEASE</version>
            </dependency>
            <!--spring-beans-->
            <dependency>
```

```
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>5.2.8.RELEASE</version>
        </dependency>
            <!--spring上下文包-->
            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-context</artifactId>
                <version>5.2.8.RELEASE</version>
            </dependency>
                <!--spring表达式包-->
            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-expression</artifactId>
                <version>5.2.8.RELEASE</version>
            </dependency>
                <!--日志包-->
            <dependency>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
                <version>1.2</version>
            </dependency>

             <!--Junit单元测试框架-->
            <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>4.12</version>
            </dependency>

        </dependencies>
</project>
```

项目引入的jar包最终如下：



1.4.2 案例演示

用Spring的方式实例化一个对象

1. 创建一个普通的类，在这类中创建一个普通的方法

```
package org.learn.spring5;

/**
 * User类
 */
public class User {

    /**
     * add方法
     */
    public void add(){
    System.out.println("add..........");
```

```
}
}

```

2. 创建Spring配置文件bean1.xml，在配置文件中配置创建的对象

```
<?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">

    <!--通过Spring配置User对象-->
    <bean id="user" class="org.learn.spring5.User"></bean>

</beans>
```

3. 创建测试类

```
package org.learn.spring5;

import org.junit.Test;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring5 {
    @Test
    public void test() {
        //1.加载spring的配置文件
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        //2.获取配置创建的对象
            //第一个参数是配置文件中配置的User对象的ID
        User user = context.getBean("user",User.class);
        System.out.println(user);
        user.add();
    }
}

```

4. 程序执行结果

```
org.learn.spring5.User@2c039ac6
add..........
```

综上所述，我们就已经完成了一个最简单的Spring项目的创建。

2 IOC容器

2.1 IOC是什么?

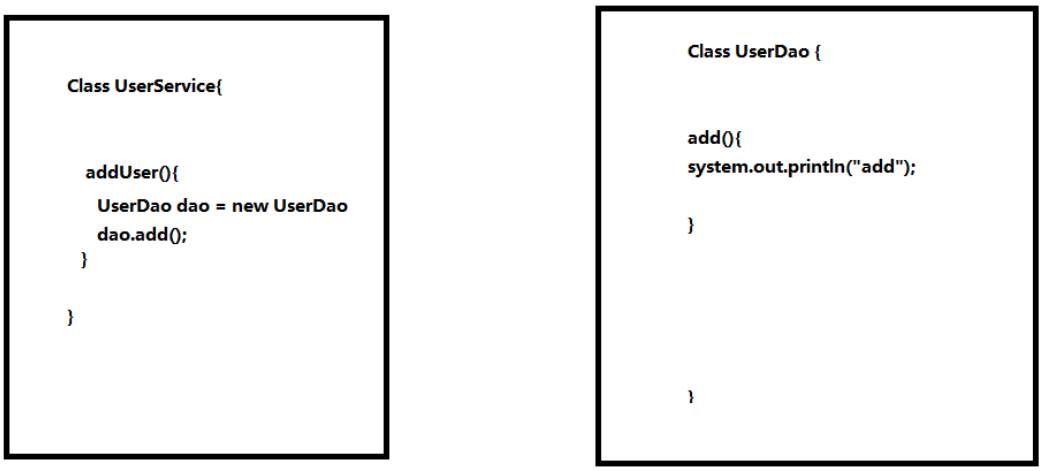
控制反转，把对象的创建和对象间的调用通过Spring去管理，**目的是为了降低程序的耦合度。**

2.2 IOC的底层原理

在讲IOC底层原理实现之前，先思考一个问题：java中的对象间如何调用?

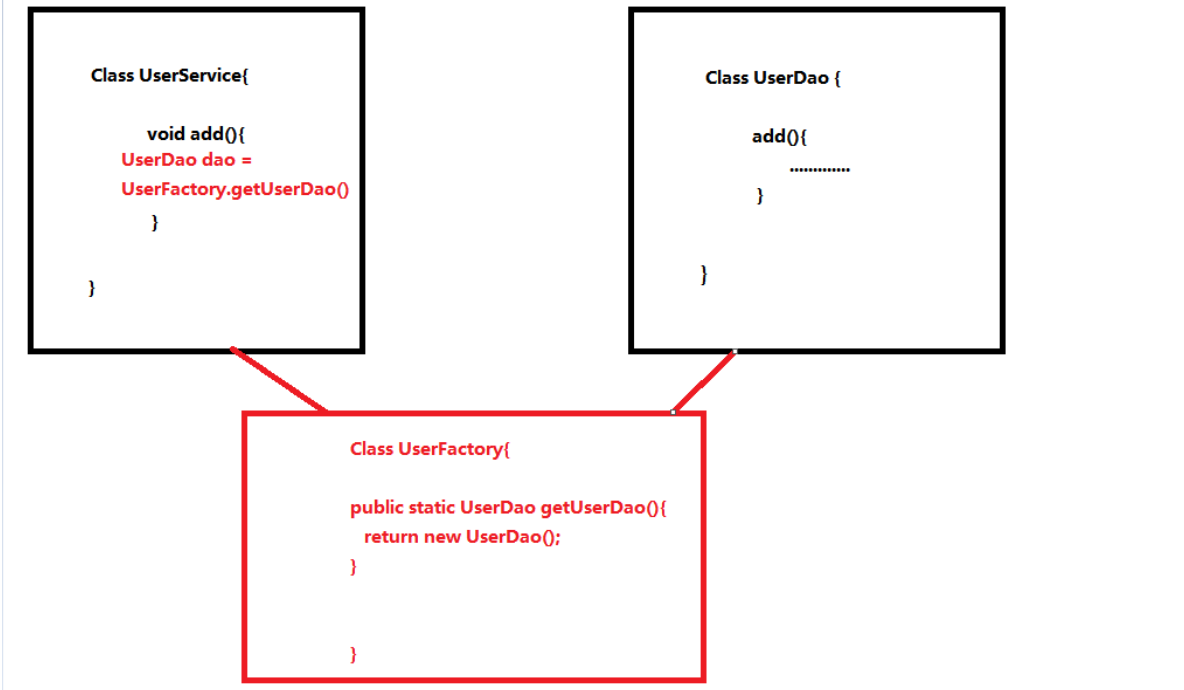
三种对象间调用的方式

方案1：传统的java对象间的调用方式



此方式耦合度较高，当UserDao方法发生变化，UserService中的代码也需要发生改变。

方案2：利用工厂模式解耦



此种方式的优点是，Service与Dao类进行了解耦，缺点是Service与工厂类UserFactory之间还是存在耦合度，如何最大限度的降低耦合度呢?

方案3：IOC创建

IOC的底层实现利用3种JAVA技术

- xml解析
- 工厂模式
- 反射

那么IOC是如何一步步降低你的程序耦合度的呢？我们要先了解IOC的过程：

1)创建xml配置文件，配置创建的对象

```
<bean id="userDao" class="org.learn.spring5.UserDao">
</bean>
```

2)创建工厂类，利用反射机制创建对象

```
class UserFactory{
    public static UserDao getUserDao(){
        String classValue =  class属性值; //利用xml解析xml文件中class属性获取class值org.learn.spring5.UserDao
        Class clazz = Class.forName(classValue); //通过反射创建对象
        return (UserDao)clazz.newInstance();
    }

}
```

从上面程序可以看出利用Spring的IOC方式我们可以最大限度的降低了程序间的耦合度，对象的创建和对象间的调度都可以通过Spring去管理。

2.2.1 IOC接口的介绍

1. IOC思想基于IOC容器完成，IOC容器的底层就是对象工厂
2. IOC容器的实例化，Spring提供了两种方式：（两个接口）

   1）BeanFactory

   IOC容器的基本实现，是Spring内部的使用接口，不提供开发人员进行使用。

   特点：加载配置文件时，不会创建对象，获取对象时才去创建对象。

   ```
   BeanFactory beanFactory = new ClassPathXmlApplicationContext("bean1.xml"); //加载配置文件时，不去创建对象

           User user = beanFactory.getBean("user",User.class);  //此时才去创建对象
   ```

   2）ApplicationContext

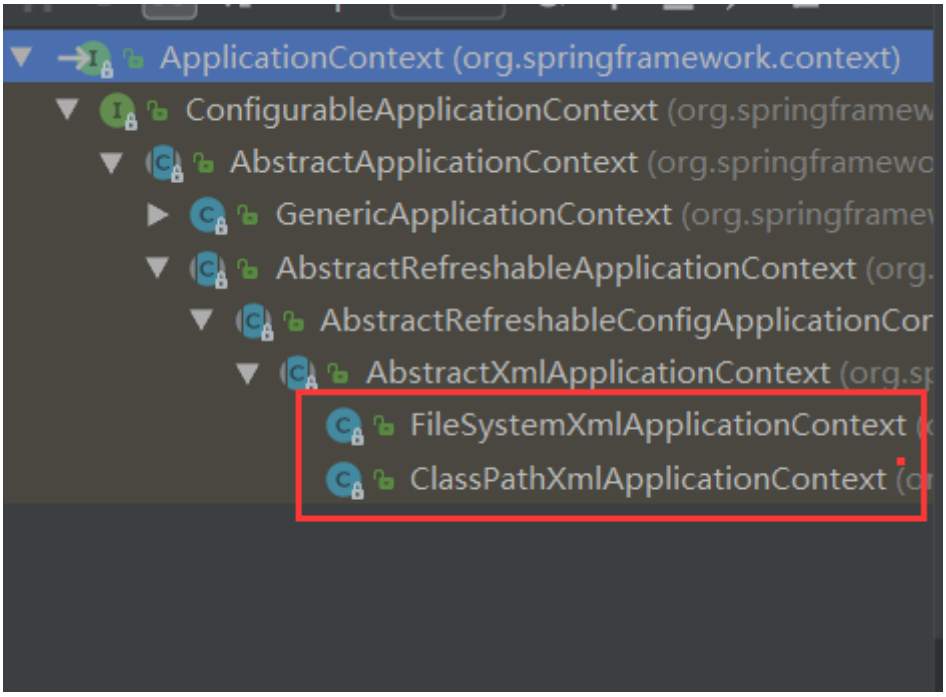
   是BeanFactory的子接口，提供了更多更强大的功能，提供开发人员使用的。

   特点：加载配置文件时，就会创建对象。

```
1  ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");//加载配置文件时，创建对象
2  User user = context.getBean("user",User.class);
```

3. ApplicationContext的实现类介绍

```
1  FileSystemXmlApplicationContext：默认是从系统的盘符下获取文件
2  ClassPathXmlApplicationContext：默认是指项目的classpath路径下面的配置文件
```



2.3 IOC操作Bean管理

什么是Bean管理?

通常是指依据Spring进行的两个操作：

1）Spring创建对象

2）Spring注入属性

Bean管理的两种实现方式：

- 基于XML方式
- 基于注解方式

2.3.1 IOC操作Bean管理（基于XML）

- **基于xml方式创建对象**

在之前入门案例章节我们已经初步认识了Spring基于xml方式创建对象。

```
<!--通过Spring配置User对象-->
<bean id="user" class="org.learn.spring5.User"></bean>
```

基本描述：

1）在Spring配置文件中，使用bean标签，标签里添加相应属性，就可以实现对象的创建。

2）在bean标签中包含许多属性

3）创建对象时，默认也是执行无参构造方法完成对象的创建。

bean标签下常用的几个属性介绍：

> **id**：唯一标识，不能添加特殊符号
>
> **class**：类的全路径，
>
> **name**：定义对象的标识，可以加特殊符号

- **基于xml方式注入属性**

DI:依赖注入，就是注入属性

1. 第一种方式使用set方式注入

   (1)创建类，定义属性和对应的set方法

```
package org.learn.spring5;

public class User {

    private String userName;
    private Integer age;

       public void setUserName(String userName) {
        this.userName = userName;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public void addUser() {

        System.out.println("添加一个用户。。。。");
    }

    public void getUserInfo() {
        System.out.println("userName:" + userName);
        System.out.println("age:" + age);

    }
}


```

(2)在Spring配置文件中，配置对象创建、配置属性注入

```
<?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">
        <!--user对象的创建-->
    <bean id="user" class="org.learn.spring5.User">
        <!--property标签设置set的属性 -->
        <property name="userName" value="joh"></property>
        <property name="age" value="11"></property>
    </bean>

</beans>
```

（3）编写测试程序

```
package org.learn.spring5;


import org.junit.Test;
import org.springframework.beans.factory.BeanFactory;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring5 {
    @Test
    public void test() {
        //1.加载spring的配置文件
        BeanFactory context = new
ClassPathXmlApplicationContext("bean1.xml");
        //2.获取配置创建的对象
        //第一个参数是配置文件中配置的User对象的ID
        User user = context.getBean("user", User.class);
        System.out.println(user);
        user.addUser();
        user.getUserInfo();
    }
}

```

执行测试方法得到结果如下：

```
org.learn.spring5.User@12c8a2c0
添加一个用户。。。。
userName:joh
age:11
```

2. 第二种方式使用构造函数注入

（1）创建类，定义属性和对应的构造方法

```
package org.learn.spring5;
```

```
2
3  public class Order {
4
5      private String orderNo;
6
7      private String address;
8
9      public Order(String orderNo, String address) {
10         this.orderNo = orderNo;
11         this.address = address;
12     }
13
14     public void orderIno(){
15         System.out.println("orderNo:"+orderNo);
16         System.out.println("address:"+address);
17     }
18 }
19
```

(2) 在Spring配置文件中，配置对象创建、配置属性注入

```
1     <!--使用构造的方式注入属性-->
2     <bean id="order" class="org.learn.spring5.Order">
3         <constructor-arg name="orderNo" value="12333"></constructor-arg>
4         <constructor-arg name="address" value="china"></constructor-arg>
5     </bean>
```

(3) 编写测试程序

```
1  @Test
2     public void test2() {
3         //1.加载spring的配置文件
4         BeanFactory context = new
ClassPathXmlApplicationContext("bean1.xml");
5         //2.获取配置创建的对象
6         //第一个参数是配置文件中配置的User对象的ID
7         Order order = context.getBean("order", Order.class);
8         System.out.println(order);
9         order.orderIno();
10    }
```

执行测试方法得到结果如下：

```
1 org.learn.spring5.Order@fcd6521
2 orderNo:12333
3 address:china
```

3. Set方法注入的另一种方式，p命名空间

(1)在Spring配置文件中，配置对象创建、使用P命名空间配置属性注入

```
    <!--使用P命名空间注入属性，简化set属性注入的方式-->
     <bean id="user1" class="org.learn.spring5.User" p:userName="jack"
p:age="15">
     </bean>

```

(2)编写测试程序

```
 /**
      * P命名空间方式注入属性
      */
     @Test
     public void test3() {
         //1.加载spring的配置文件
         BeanFactory context = new
ClassPathXmlApplicationContext("bean1.xml");
         //2.获取配置创建的对象
         //第一个参数是配置文件中配置的User对象的ID
         User user = context.getBean("user1", User.class);
         System.out.println(user);
         user.addUser();
         user.getUserInfo();
     }
```

执行测试方法得到结果

```
org.learn.spring5.User@6dde5c8c
添加一个用户。。。。
userName:jack
age:15
```

4. 注入空值和特殊符号

1）空值

```
        <!--向属性值中设置空值-->
        <property name="email">
            <null />
        </property>
```

2）特殊符号

```
  <!--属性值包含特殊符号 <>
    1.转义<>
    2.CDATA
    -->
    <property name="email">
        <value>
            <![CDATA[<<email@qq.com>>]]>
        </value>
    </property>
```

5. 注入属性-外部bean

场景：UserService和UserDao的关系

步骤：1）创建UserService类和UserDao接口、UserDAOImpl实现类

```
package org.learn.spring5.service;

import org.learn.spring5.dao.UserDao;

public class UserService {

    //创建userDao属性，生成set方法

    private UserDao userDao;

    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }

    public void userAdd() {
        //调用 userDao对象的userAdd方法
        System.out.println("service userAdd");
        userDao.userAdd();

    }
}

```

```
package org.learn.spring5.dao;

public interface UserDao {

    int userAdd();
}

```

```
package org.learn.spring5.dao.impl;

import org.learn.spring5.dao.UserDao;

public class UserDaoImpl implements UserDao {
    public int userAdd() {
        System.out.println("UserDao userAdd");
        return 0;
    }
}

```

2）在bean2.xml配置文件中创建对象UserDao和UserDao，并且在UserService对象中注入UserDao属性

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"


```

```
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--创建对象UserService-->
  <bean id="userService" class="org.learn.spring5.service.UserService">
      <!--注入外部bean-->
      <!--ref属性：是UserDaoImpl对象的id值-->
      <property name="userDao" ref="userDaoImpl"></property>
  </bean>
    <!--创建对象UserDaoImpl-->
    <bean id="userDaoImpl"
class="org.learn.spring5.dao.impl.UserDaoImpl"></bean>
</beans>
```

3）创建测试程序

```
    /**
     *注入属性-外部bean
     */
    @Test
    public void test4() {
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean2.xml");
       UserService userService =  context.getBean("userService",
UserService.class);
       userService.userAdd();
    }
```

执行测试方法得到的结果

```
service userAdd
UserDao userAdd
```

6. 注入属性-内部bean

场景：一对多关系： 班级和学生的关系

一个班级有多个学生，一个学生只属于一个班级

在实体类中表示一对多的关系

1）创建类和属性

```
package org.learn.spring5.bean;

//班级类
public class Classes {

    public void setName(String name) {
        this.name = name;
    }

    private String name;

    @Override
```

```
    public String toString() {
        return "Classes{" +
                "name='" + name + '\'' +
                '}';
    }
}

```

```
package org.learn.spring5.bean;

//学生类
public class Student {

    //学生名称
    private String studentName;
    //学生的年龄
    private String age;

    public void setStudentName(String studentName) {
        this.studentName = studentName;
    }

    public void setAge(String age) {
        this.age = age;
    }

    public void setClasses(Classes classes) {
        this.classes = classes;
    }

    //学生所属的班级
    private Classes classes;


    public void add() {
        System.out.println("studentName:" + studentName + "----" +
"age:" + age);
        System.out.println("classes:" + classes);
    }
}

```

2）创建配置文件

```
<?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:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--注入属性-内部bean-->

    <!--创建对象-->
    <bean id="student" class="org.learn.spring5.bean.Student">
        <!--设置对象属性-->
```

```
        <property name="studentName" value="xiaoming"></property>
        <property name="age" value="15"></property>
        <!--配置内部bean-->
        <property name="classes">
            <bean id="classes" class="org.learn.spring5.bean.Classes">
                <property name="name" value="初三一班"></property>

            </bean>

        </property>
    </bean>


</beans>
```

3）创建测试程序

```
    /**
     * 注入属性-内部bean
     */
    @Test
    public void test5() {
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean3.xml");
        Student student = context.getBean("student", Student.class);
        student.add();
    }
```

执行测试方法得到的结果

```
studentName:xiaoming----age:15
classes:Classes{name='初三一班'}
```

7. 注入属性-级联赋值

方式1：

配置文件中设置

```
  <!--注入属性-级联赋值-->
    <!--方式1-->
    <!--创建对象-->
    <bean id="student" class="org.learn.spring5.bean.Student">
        <!--设置对象属性-->
        <property name="studentName" value="xiaoming"></property>
        <property name="age" value="15"></property>
        <!--级联赋值-->
        <property name="classes" ref="classes"></property>
    </bean>
    <bean id="classes" class="org.learn.spring5.bean.Classes">
        <property name="name" value="初三一班"></property>
    </bean>

```

方式2：

Student类中class属性增加get方法

```
    //学生所属的班级
    private Classes classes;


    public Classes getClasses() {
        return classes;
    }
```

配置文件中设置

```
  <bean id="student" class="org.learn.spring5.bean.Student">
        <!--设置对象属性-->
        <property name="studentName" value="xiaoming"></property>
        <property name="age" value="15"></property>
        <!--级联赋值-->
        <property name="classes" ref="classes"></property>
        <property name="classes.name" value="初三二班"></property>
    </bean>
    <bean id="classes" class="org.learn.spring5.bean.Classes">
    </bean>
```

8. IOC操作Bean管理-注入集合属性

1）注入数组类型属性

2）注入集合类型属性

3）注入Map类型属性

4）集合里设置对象的值

第一步：创建类

```
package org.learn.spring5.bean;


import java.lang.reflect.Array;
import java.util.Arrays;
import java.util.List;
import java.util.Map;

//班级类
public class Classes {

    //数组类型
    private String[] student;
    //集合类型
    private List<String> list;
    //map类型
    private Map<String, String> maps;

    public void setStudents(List<Student> students) {
        this.students = students;
    }

```

```
    //对象类型
    private List<Student> students;


    public void setStudent(String[] student) {
        this.student = student;
    }

    public void setList(List<String> list) {
        this.list = list;
    }

    public void setMaps(Map<String, String> maps) {
        this.maps = maps;
    }


    public void test() {
        System.out.println(Arrays.toString(student));
        System.out.println(list);
        System.out.println(maps);
        System.out.println(students);

    }
}


```

```
package org.learn.spring5.bean;


//學生類
public class Student {

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    //学生名称
    private String name;

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    //年龄
    private int age;
}

```

第二步：创建Spring配置文件

```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:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--IOC操作Bean管理-注入集合属性-->

    <bean id="classes" class="org.learn.spring5.bean.Classes">
            <!--数组类型的属性注入-->
    <property name="student">
        <array>
            <value>zhangsan</value>
            <value>lisi</value>
            <value>wangwu </value>
        </array>

    </property>
            <!--集合类型的属性注入-->
        <property name="list">
            <list>
                <value>list1</value>
                <value>list2</value>
                <value>list3</value>
            </list>
        </property>
            <!--map类型的属性注入-->
        <property name="maps">
            <map>
                <entry key="k1" value="v1"></entry>
                <entry key="k2" value="v2"></entry>
                <entry key="k3" value="v3"></entry>
            </map>
        </property>
        <!--集合里是对象类型的值-->
        <property name="students">
            <list>
                <ref bean="student1"></ref>
                <ref bean="student2"></ref>
            </list>
        </property>
    </bean>


    <bean id="student1" class="org.learn.spring5.bean.Student">
        <property name="name" value="xiaoming"></property>
        <property name="age" value="15"></property>
    </bean>
    <bean id="student2" class="org.learn.spring5.bean.Student">
        <property name="name" value="xiaohong"></property>
        <property name="age" value="16"></property>
    </bean>
</beans>
```

第三步：创建测试程序

```
    /**
     * IOC操作Bean管理-注入集合属性
     */
    @Test
    public void test() {

        ApplicationContext context = new ClassPathXmlApplicationContext("bean1.xml");
        Classes classes = context.getBean("classes", Classes.class);
        classes.test();
    }
```

测试程序执行结果：

```
[zhangsan, lisi, wangwu]
[list1, list2, list3]
{k1=v1, k2=v2, k3=v3}
[Student{name='xiaoming', age=15}, Student{name='xiaohong', age=16}]

```

5）把集合属性抽取成公共部分

```
<?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:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/util https://www.springframework.org/schema/util/spring-util.xsd">


    <!--IOC操作Bean管理-提取list集合属性-->
    <!-- 提取list集合属性-->
    <util:list id="list1">
        <value>list1</value>
        <value>list2</value>
        <value>list3</value>
    </util:list>

    <bean id="classes" class="org.learn.spring5.bean.Classes">
        <!--数组类型的属性注入-->
        <property name="student">
            <array>
                <value>zhangsan</value>
                <value>lisi</value>
                <value>wangwu</value>
            </array>

        </property>
        <!--集合类型的属性注入,引入公共的属性-->
```

```
        <property name="list" ref="list1"></property>

        <!--map类型的属性注入-->
        <property name="maps">
            <map>
                <entry key="k1" value="v1"></entry>
                <entry key="k2" value="v2"></entry>
                <entry key="k3" value="v3"></entry>
            </map>
        </property>
        <!--集合里是对象类型的值-->
        <property name="students">
            <list>
                <ref bean="student1"></ref>
                <ref bean="student2"></ref>
            </list>
        </property>
    </bean>


    <bean id="student1" class="org.learn.spring5.bean.Student">
        <property name="name" value="xiaoming"></property>
        <property name="age" value="15"></property>
    </bean>
    <bean id="student2" class="org.learn.spring5.bean.Student">
        <property name="name" value="xiaohong"></property>
        <property name="age" value="16"></property>
    </bean>
</beans>
```

9. IOC操作Bean管理-工厂bean

Spring 有两种类型的Bean，一种是普通bean，另一种是工厂bean（FactoryBbean）

普通bean：在配置文件中定义的类型和返回类型一致

工厂bean：在配置文件中定义的类型可以和返回类型不一致

工厂bean的实现步骤

1）创建普通类Student

```
public class Student {

    private String name;

    public void setName(String name) {
        this.name = name;
    }
}

```

2）创建自定义工厂类并实现接口FactoryBean

```java
import org.springframework.beans.factory.FactoryBean;

public class MyBean implements FactoryBean<Student> {
    //定义返回的bean
    public Student getObject() throws Exception {
        Student student = new Student();
        student.setName("zhangsan");
        return student;
    }

    public Class<?> getObjectType() {
        return null;
    }
}

```

3）创建Spring的配置文件

```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">

    <!--自定义工厂类-->
    <bean id="myBean" class="MyBean"></bean>
</beans>
```

4）创建测试类

```java
    /**
     * IOC操作Bean管理-工厂bean
     */
    @Test
    public void test() {
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        Student student = context.getBean("myBean", Student.class);
        System.out.println(student);
    }
```

测试程序执行结果：

```
Student@23d2a7e8
```

10. IOC操作Bean管理-bean的作用域

定义：在Spring中，设置创建bean实例是单实例还是多实例

默认情况下，bean是单实例对象

设置bean的作用域

scope属性：

singleton，表示单实例对象，默认值，加载Spring配置文件时，就会创建对象。

```
<bean id="classes" class="Classes" scope="singleton">
</bean>
```

prototype,表示多实例，在调用getBean方法时创建多实例的对象

```
<bean id="classes" class="Classes" scope="prototype">
</bean>
```

request，表次一次请求，每次HTTP请求都会创建一个新的Bean，适用于WebApplicationContext环境。

session，表示一次session，同一个Session共享一个Bean实例。不同Session使用不同的实例。

11. bean的生命周期

- 生命周期的定义：从对象的创建到对象的销毁过程称为bean的生命周期。
- bean的生命周期过程：

1）通过构造器创建bean实例（无参构造）

2）为bean属性设置值（调用set方法）

3）调用bean的初始化方法

4）bean对象获取使用

5）容器关闭时，调用bean的销毁方法

- 演示生命周期

1）创建类及对应的生命周期的方法

```
//学生类
public class Student {
    public Student() {
        System.out.println("第一步：无参构造方法");
    }

    public void setName(String name) {
        this.name = name;
        System.out.println("第二步：调用set方法");
    }

    //初始化方法
    public void initMethod(){
        System.out.println("第三步：执行初始化方法");
    }


    public void destoryMethod(){
        System.out.println("第五步：执行销毁的方法");
    }
```

```
    //学生的名称
    private String name;

}

```

2）创建Spring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/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">


    <!--bean的生命周期演示-->

    <!--实例化对象-->
    <bean id="student" class="Student" init-method="initMethod" destroy-
method="destoryMethod">

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

    </bean>


</beans>
```

3）创建测试程序

```
   /**
     * IOC操作Bean管理-生命周期
     */
    @Test
    public void test() {
        //ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");

        Student student = context.getBean("student", Student.class);
        System.out.println("第四步：获取创建bean实例对象");
        System.out.println(student);
        //手动让bean实例销毁
        context.close();
    }
```

测试程序执行结果：

```
第一步：无参构造方法
第二步：调用set方法
第三步：执行初始化方法
第四步：获取创建bean实例对象
Student@4a87761d
第五步：执行销毁的方法
```

- bean的后置处理器，bean的生命周期过程（七步）

1）通过构造器创建bean实例（无参构造）

2）为bean属性设置值（调用set方法）

3）把bean实例传递bean后置处理器的方法postProcessBeforeInitialization

4）调用bean的初始化方法

5）把bean实例传递bean后置处理器的方法postProcessAfterInitialization

6）bean对象获取使用

7）容器关闭时，调用bean的销毁方法

- 添加后置处理后的生命周期演示

1）创建自定义的后置处理器类

```
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

/**
 * 实现后置处理器
 */
public class MyBeanPost implements BeanPostProcessor {
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {

        System.out.println("在初始化之前执行的方法");
        return bean;
    }

    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("在初始化之后执行的方法");
        return bean;
    }
}

```

2）创建Spring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/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--bean的生命周期演示-->
    <!--实例化对象-->
    <bean id="student" class="Student" init-method="initMethod" destroy-method="destoryMethod">
        <property name="name" value="zhangsan"></property>
    </bean>

    <!--配置后置处理,会对当前所有对象都添加后置处理器-->
```

```
    <bean id="myBeanPost" class="MyBeanPost"></bean>

</beans>
```

测试程序执行结果：

```
第一步：无参构造方法
第二步：调用set方法
在初始化之前执行的方法
第三步：执行初始化方法
在初始化之后执行的方法
第四步：获取创建bean实例对象
Student@37374a5e
第五步：执行销毁的方法
```

12. 基于xml方式-自动装配

根据指定装配规则，（属性的名称或者属性的类型），Spring自动将匹配的属性值进行注入

```
    <!--实现自动装配
            属性：autowire，配置自动装配
            byName：按属性名称自动装配
            byType：按属性类型自动装配
        -->
    <bean id="student" class="Student" autowire="byName">
    </bean>
    <bean id="classes" class="Classes"></bean>
```

13. 基于xml方式-引入外部的属性文件

场景：spring配置jdbc数据库连接文件的引入

步骤：1）模拟创建数据库连接池类MyDataSource

```
//模拟数据库连接池类
public class MyDataSource {

//定义数据库连接属性
    //驱动名称
    private String driverClassName;
    //连接地址
    private String url;
    //用户名
    private String userName;
    //密码
    private String password;


    public void setDriverClassName(String driverClassName) {
        this.driverClassName = driverClassName;
    }

    public void setUrl(String url) {
        this.url = url;
    }
```

```
22
23      public void setUserName(String userName) {
24          this.userName = userName;
25      }
26
27      public void setPassword(String password) {
28          this.password = password;
29      }
30
31
32
33      @Override
34      public String toString() {
35          return "MyDataSource{" +
36                  "driverClassName='" + driverClassName + '\'' +
37                  ", url='" + url + '\'' +
38                  ", userName='" + userName + '\'' +
39                  ", password='" + password + '\'' +
40                  '}';
41      }
42  }
43
```

2）创建jdbc.properties文件

```
1  prop.driverClass=com.mysql.jdbc.Driver
2  prop.url=jdbc:mysql://localhost:3306/userDb
3  prop.userName=root
4  prop.password=123456
```

3）创建Spring文件文件，引入context命名空间并且设置引入外部的配置文件

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xmlns:context="http://www.springframework.org/schema/context"
5         xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/context
   https://www.springframework.org/schema/context/spring-context.xsd">
6
7
8      <!--引入外部的配置文件-->
9      <context:property-placeholder location="classpath:jdbc.properties" />
10     <!--注入属性-->
11     <bean id="myDataSource" class="MyDataSource">
12         <property name="driverClassName" value="${prop.driverClass}">
   </property>
13         <property name="url" value="${prop.url}"></property>
14         <property name="userName" value="${prop.userName}"></property>
15         <property name="password" value="${prop.password}"></property>
16
17     </bean>
18
19
20 </beans>
```

4）编写测试程序

```
    /**
     * IOC操作Bean管理-引入外部属性文件
     */
    @Test
    public void test() {
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        MyDataSource myDataSource =
context.getBean("myDataSource",MyDataSource.class);
        System.out.println(myDataSource);
    }
```

5）测试程序执行结果

```
MyDataSource{driverClassName='com.mysql.jdbc.Driver',
url='jdbc:mysql://localhost:3306/userDb', userName='root', password='123456'}
```

2.3.2 IOC操作Bean管理（基于注解）

2.3.2.1 什么是注解?

(1)代码里特殊的标记，格式：@注解名称(属性名称=属性值..)

(2)可以在类、方法、属性上添加注解。

2.3.2.2 使用注解的目的?

可以简化xml配置，使程序变得更简洁、优雅

2.3.2.3 注解的运用

- 创建对象提供的注解
  - @Componet
  - @Service
  - @Controller
  - @Repository

  特点：上面四个注解的功能都是一样的，都可以用来创建bean实例，建议使用在对应的功能层次中

注解	含义
@Component	最普通的组件，可以被注入到spring容器进行管理
@Repository	作用于持久层
@Service	作用于业务逻辑层
@Controller	作用于表现层，Controller类上

- 属性注入提供的注解：

  @Autowired

  @Qualifier

  @Resource

  @Value

  特点：@Autowired、@Qualifier、@Resource可以用于注入对象类型属性，@Value可以用户注入普通类型的属性。

注解	含义
@Autowired	根据属性类型进行自动装配
@Qualifier	根据属性名称进行注入
@Resource	可以根据属性、名称进行注入
@Value	注入普通类型的属性

2.3.2.4 基于注解实现创建对象

1）引入依赖

```
<!-- spring-aop-->
       <dependency>
           <groupId>org.springframework</groupId>
           <artifactId>spring-aop</artifactId>
           <version>5.2.8.RELEASE</version>
       </dependency>
```

2）开启组件扫描

```
<?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
https://www.springframework.org/schema/context/spring-context.xsd">

    <!--开启组件扫描
    base-package属性：设置要扫描的包
    -->
    <context:component-scan base-package="org.leanr.spring5">
</context:component-scan>
</beans>
```

3）类上添加注解

```
package org.leanr.spring5.service;

import org.springframework.stereotype.Service;

```

```java
/**
 * 类上增加创建对象的注解
 * - @Componet
 * - @Service
 * - @Controller
 * - @Repository
 * value属性可以不设置，默认是类的名称，首字母小写，
 */
@Service(value = "userService")
public class UserService {
    public void add(){
        System.out.println("adddddddd");
    }
}


```

4）编写测试类

```java
package org.leanr.spring5;

import org.junit.Test;
import org.leanr.spring5.bean.User;
import org.leanr.spring5.service.UserService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring5 {
    /**
     * IOC操作Bean管理-注解方式创建对象
     */
    @Test
    public void test() {
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        UserService userService = context.getBean("userService",
UserService.class);
        System.out.println(userService);
        userService.add();
    }


}

```

测试类执行结果：

```
org.leanr.spring5.service.UserService@7920ba90
adddddddd
```

2.3.2.5 组件扫描配置的细节

```
    <!--use-default-filters属性表示：是否开启默认的filter来扫描包下的内容
     默认是true，如果设置为false，则需要设置指定的规则
     例如下面设置的意思是，注解是@Controller在org.leanr.spring5包下扫描是生效的，其他注解类不会被扫描到
     -->
    <context:component-scan base-package="org.leanr.spring5" use-default-filters="false">
        <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>
</beans>
```

2.3.2.6 基于注解实现属性的注入

第一步注解方式创建对象

第二步在类中注入外部属性，并添加属性的注解

1)@Autowired

```
@Service(value = "userService")
public class UserService {
    //注入属性的注解@Autowired，根据属性类型自动装入
    @Autowired
    private UserDao userDao;
    public void add(){
        System.out.println("userService add");

        userDao.add();
    }
```

2)@Qualifier

根据属性名称进行注入

```
/**
 * UserDao的实现类
 */
@Repository(value = "userDaoImpl2")
public class UserDaoImpl implements UserDao {
    public int add() {

        System.out.println("UserDaoImpl add ");
        return 1;
    }
}
```

```java
@Service(value = "userService")
public class UserService {
    //注入属性的注解@Autowired，根据属性类型自动装入
    @Autowired
    @Qualifier(value = "userDaoImpl2")
    private UserDao userDao;
    public void add(){
        System.out.println("userService add");

        userDao.add();
    }
```

3)@Resource

既可以根据类型也可以名称注入

```java
@Service(value = "userService")
public class UserService {
    //注入属性的注解@Autowired，根据属性类型自动装入
    //@Autowired
    //@Qualifier(value = "userDaoImpl2")
    //@Resource  //根据类型注入
    @Resource(value="userDaoimpl2")  //根据名称注解
    private UserDao userDao;
    public void add(){
        System.out.println("userService add");

        userDao.add();
    }
```

4)@Value

针对普通类型的属性注入

```java
   @Value(value = "zhangsan")
    private String userName;
```

2.3.2.7 纯注解开发

1）创建配置类，替代xml配置文件

```java
package org.leanr.spring5.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
//Configuration，让Spring知道这是个配置类
@Configuration
@ComponentScan(basePackages = {"org.leanr.spring5"})
public class SpringConfig {
}

```

2）编写测试类

```java
  /**
     * 完全注解开发
     */
    @Test
    public void test3(){
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(SpringConfig.class);
        UserService userService = context.getBean("userService",
UserService.class);
        userService.add();
    }
```

3 AOP

3.1 基本概念

AOP,面向切面编程,利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

3.2 主要意图

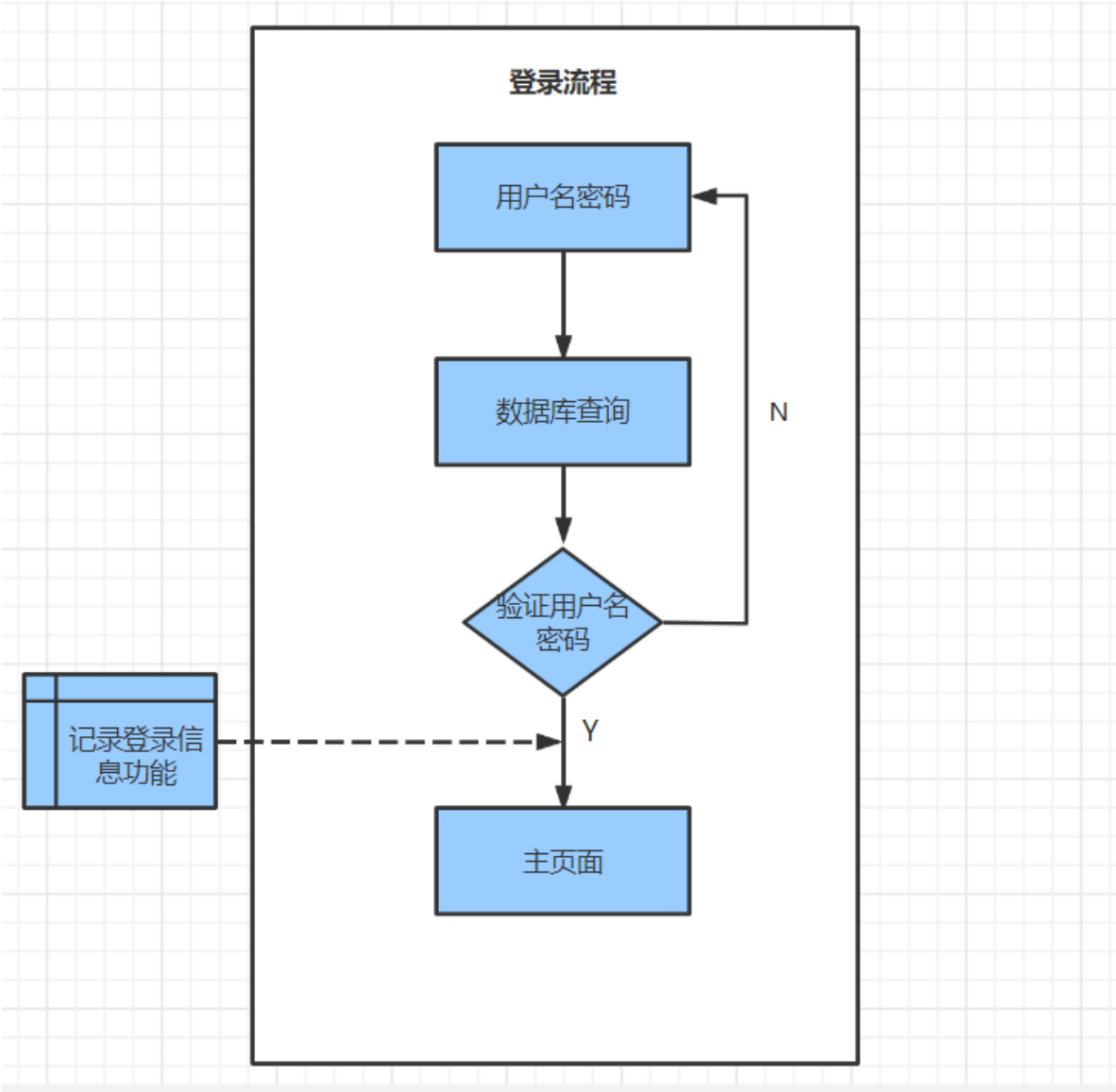
将日志记录，性能统计，安全控制，事务处理，异常处理等代码从业务逻辑代码中划分出来，通过对这些行为的分离，我们希望可以将它们独立到非指导业务逻辑的方法中，进而改变这些行为的时候不影响业务逻辑的代码。

核心思想：不通过修改源代码方式添加新功能。

实际案例分享：

例如我们现在有一个用户登录的功能，现在需要增加一个新的功能，就是登录完成后增加登录信息的记录的功能，记录的信息包括登录名称、登录时间、登录的ip地址、登录的操作系统等内容。我们会有两种方式实现，一种是修改源代码，在登录功能代码中添加记录用户登录信息功能，另外一种方式是通过AOP的方式不改变源代码添加登录信息记录的功能。

AOP实现方式如下图：



3.3 AOP的底层实现原理

AOP底层使用动态代理实现：

动态代理实现有两种方式

1）被代理的对象有接口情况

使用JDK动态代理实现

场景：

```
Interface UserDao {
int login();
}

Class UserDaoImpl implements UserDao{

public int login(){
//登录过程
}

}
```

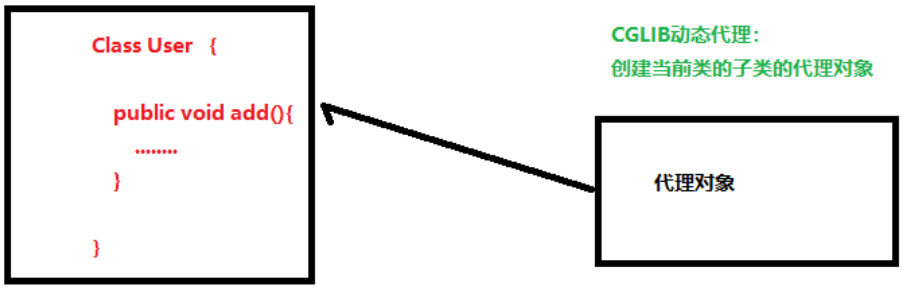
不修改UserDaoImpl类的login方法的情况下增强当前类的功能，使用JDK动态代理

创建UserDao接口实现类代理对象

2）被代理的对象没有接口的情况

使用CGLIB动态代理实现

场景：



AOP(JDK动态代理)实现原理

编写JDK动态代理代码：

1）创建接口UserDao和接口实现类UserDaoImpl

```java
package org.learn.spring5.dao;

public interface UserDao {
    int add();

    int update(Integer id);

}

```

```java
package org.learn.spring5.dao.impl;

import org.learn.spring5.dao.UserDao;
```

```
4
5  /**
6   * UserDao的实现类
7   */
8  public class UserDaoImpl implements UserDao {
9
10
11      public int add() {
12          System.out.println("UserDaoImpl add ");
13          return 1;
14      }
15
16      public int update(Integer id) {
17
18          System.out.println("UserDaoImpl  update");
19          return id;
20      }
21 }
22
```

2）创建MyInvocationHandler类实现InvocationHandler，该类用于实现代理类调用的方法增强的那部分的处理。增强处理的内容是在invoke方法中，在代理类调用方法时，执行invoke方法。

```
1  package org.learn.spring5.dao.impl;
2
3  import java.lang.reflect.InvocationHandler;
4  import java.lang.reflect.Method;
5
6  //创建代理对象调用的程序代码
7  public class MyInvocationHandler implements InvocationHandler {
8      private Object object;
9      //有参构造传递
10     public MyInvocationHandler(Object object){
11         this.object = object;
12     }
13
14     //增强的逻辑
15     //第一个参数：类加载
16     //第二个参数：增强方法所在的类，这个类实现的接口，支持多个接口
17     //第三个参数：实现这个接口InvocationHandler，创建代理对象，写增强的方法
18     public Object invoke(Object proxy, Method method, Object[] args) throws
   Throwable {
19
20         System.out.println("方法之前处理");
21
22         //被增强的方法执行
23         Object invoke = method.invoke(object,args);
24
25         System.out.println("方法之后处理");
26
27
28         return invoke;
29     }
30 }
31
```

3）利用Proxy类生成代理对象的实例，调用相应的被代理类的方法。

```
import org.learn.spring5.dao.UserDao;
import org.learn.spring5.dao.impl.MyInvocationHandler;
import org.learn.spring5.dao.impl.UserDaoImpl;

import java.lang.reflect.Proxy;

class TestProxy {

    public static void main(String[] args) {
        //被增强的类，真实的角色
        UserDaoImpl userDaoImpl = new UserDaoImpl();
        Class[] interfaces = {UserDao.class};  //接口
        //动态生成代理对象的实例并返回，传递谁就生产谁的代理对象
        UserDao dao = (UserDao)
Proxy.newProxyInstance(TestProxy.class.getClassLoader(),
userDaoImpl.getClass().getInterfaces(), new
MyInvocationHandler(userDaoImpl));
        int res = dao.update(1);
        System.out.println(res);
    }
}

```

3.4 AOP操作术语

- 连接点

  类里面哪些方法可以被增强，这些方法就被称为连接点。

- 切入点

  实际被增强的方法，称之为切入点

- 通知（增强）

  实际增强的内容被称为通知

  通知有多种类型：

  前置通知

  后置通知

  环绕通知

  异常通知

  最终通知

- 切面

  通知应用到切入点的过程

- 切入点表达式

  （1）切入点表达式作用：知道对哪个类里面的哪个方法进行增强

  （2）语法结构：

  excution (<权限修饰符><返回类型><类全路径><方法名称>(<参数列表>))

（3）示例：对org.learn.spring5.UserDao类的add方法进行增强

excution(* org.learn.spring5.UserDao.add(..))

示例：对org.learn.spring5.UserDao类的所有方法进行增强

excution(* org.learn.spring5.UserDao.*(..))

3.5 AOP具体操作

Spring框架一般是基于AspectJ实现AOP操作

什么是AspectJ?

AspectJ是独立的框架，可以和Spring一起用，完成Spring AOP的操作。

基于AspectJ实现Spring AOP具体实现方式分为两种：

- 基于XML配置文件
- 基于注解方式

实现具体操作前的准备工作：

1）环境搭建，引入相关依赖

```
    <!-- spring-aop-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>5.2.8.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>net.sourceforge.cglib</groupId>
            <artifactId>com.springsource.net.sf.cglib</artifactId>
            <version>2.2.0</version>
        </dependency>

        <dependency>
            <groupId>aopalliance</groupId>
            <artifactId>aopalliance</artifactId>
            <version>1.0</version>
        </dependency>

        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.9.2</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>5.2.8.RELEASE</version>
        </dependency>
```

基于注解方式实现AOP的具体操作

1. 开启注解扫描和开启Aspect生成代理对象

```
package org.learn.spring5.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

//Configuration，让Spring知道这是个配置类
@Configuration
@ComponentScan(basePackages = {"org.learn.spring5"})
@EnableAspectJAutoProxy
public class SpringConfig {
}

```

2. 创建类，在类里定义方法，并通过注解的方式完成对象的创建。

```
package org.learn.spring5.service.impl;

import org.learn.spring5.service.UserService;
import org.springframework.stereotype.Service;

@Service  //1
public class UserServiceImpl implements UserService {

    //2
    public void add() {
        System.out.println("add");
    }

    public void del() {
        System.out.println("del");
    }

    public void update() {
        System.out.println("update");
    }

    public void query() {
        System.out.println("query");
    }
}

```

3. 对类中的方法进行增强：

1）创建一个增强的类

2）实现增强的逻辑

3）并通过注解的方式完成该对象的创建

4）添加通知的注解配置@Aspect

5）配置不同类型的通知，在增强的类中，在作为通知方法上面增加通知类型注解，使用切入点表达式配置

具体代码如下：

```
package org.learn.spring5.service.impl;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

//增强类，编写增强的方法
@Component      //(3)
@Aspect   //(4)
public class UserServiceProxy  {    //(1)

    //前置通知
    @Before(value ="execution(*
org.learn.spring5.service.impl.UserServiceImpl.add(..))")   //(5)
    public void before(){
        System.out.println("before");  //(2)
    }
  //后置通知
    @After(value ="execution(*
org.learn.spring5.service.impl.UserServiceImpl.add(..))")
    public void after(){
    System.out.println("After");
}
    //最终通知
    @AfterReturning(value ="execution(*
org.learn.spring5.service.impl.UserServiceImpl.add(..))")
    public void afterReturning(){
        System.out.println("afterReturning");
    }
    //异常通知
    @AfterThrowing(value ="execution(*
org.learn.spring5.service.impl.UserServiceImpl.add(..))")
    public void afterThrowing(){

        System.out.println("AfterThrowing");
    }
    //环绕通知
    @Around(value ="execution(*
org.learn.spring5.service.impl.UserServiceImpl.add(..))")
    public void around(ProceedingJoinPoint proceedingJoinPoint){
        System.out.println("环绕之前");
        try {
            proceedingJoinPoint.proceed();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
        System.out.println("环绕之后");
    }
}

```

4. 测试类编写：

```
import org.junit.Test;
import org.learn.spring5.config.SpringConfig;
```

```
import org.learn.spring5.service.UserService;
import org.learn.spring5.service.impl.UserServiceImpl;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class TestSpring5 {

    /**
     * Spring AOP实现
     */
    @Test
    public void test1(){
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(SpringConfig.class);
        UserService userService = context.getBean("userServiceImpl",
UserService.class);
        userService.add();

    }

}
```

- **对公共切入点抽取**

例如上面的例子，5种切入点表达式切入的都是同一个方法，表达式都是一样的，那么如何将这行代码抽取成公共的代码进行调用呢?

```
package org.learn.spring5.service.impl;

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

//增强类，编写增强的方法
@Component
@Aspect
public class UserServiceProxy {

    //相同的切入点抽取
    @Pointcut(value = "execution(*
org.learn.spring5.service.impl.UserServiceImpl.add(..))")
    public void pointCunt(){


    }

    //前置通知
    @Before(value = "pointCunt()")
    public void before(){
    System.out.println("before");
}

    //后置通知
    @After(value = "pointCunt()")
    public void after(){
```

```
    System.out.println("After");
}
    //最终通知
    @AfterReturning(value = "pointCunt()")
    public void afterReturning(){
        System.out.println("afterReturning");
    }
    //异常通知
    @AfterThrowing(value = "pointCunt()")
    public void afterThrowing(){

        System.out.println("AfterThrowing");
    }
    //环绕通知
    @Around(value = "pointCunt()")
    public void around(ProceedingJoinPoint proceedingJoinPoint){
        System.out.println("环绕之前");
        try {
            proceedingJoinPoint.proceed();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
        System.out.println("环绕之后");
    }

}

```

- **一个方法有多个增强类，设置优先级可以确定执行的顺序**

@order()注解，括号里的值越小，优先级越高，先执行

```
package org.learn.spring5.service.impl;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

//增强类2
@Component
@Aspect
@Order(3)
public class UserServiceProxy2 {

    @Before(value ="execution(*
org.learn.spring5.service.impl.UserServiceImpl.add(..))")
    public void before(){
    System.out.println("增强类2 before");
}
}

```

基于XML配置文件方式实现AOP的具体操作

1. 创建类，被增强类和增强类

```
package org.learn.spring5.service;

public interface UserService {
    void add();
    void del();
    void update();
    void query();
}

```

```
package org.learn.spring5.service.impl;

import org.learn.spring5.service.UserService;


 public class UserServiceImpl implements UserService {

    public void add() {
        //int i=10/0;
        System.out.println("add");
    }
    public void del() {
        System.out.println("del");
    }

    public void update() {
        System.out.println("update");
    }

    public void query() {
        System.out.println("query");
    }
}

```

```
package org.learn.spring5.service.impl;

import org.aspectj.lang.ProceedingJoinPoint;

//增强类，编写增强的方法

public class UserServiceProxy {
    public void before(){
    System.out.println("before");
}
    public void after(){
    System.out.println("After");
}
    public void afterReturning(){
        System.out.println("afterReturning");
```

```
    }
    public void afterThrowing(){
        System.out.println("AfterThrowing");
    }
    public void around(ProceedingJoinPoint proceedingJoinPoint){
        System.out.println("环绕之前");
        try {
            proceedingJoinPoint.proceed();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
        System.out.println("环绕之后");
    }

}

```

2. 配置XML文件 bean1.xml

1）创建对象

2）配置AOP部分

```
<?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
https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!--Spring AOP操作，基于XML实现方式-->

    <!--对象创建-->
    <bean id="userServiceImpl"
class="org.learn.spring5.service.impl.UserServiceImpl"></bean>
    <bean id="userServiceProxy"
class="org.learn.spring5.service.impl.UserServiceProxy"></bean>

    <!-- 配置AOP增强-->
    <aop:config>
        <!--切入点-->
        <aop:pointcut id="p" expression="execution(*
org.learn.spring5.service.impl.UserServiceImpl.add(..))"/>
        <!--配置切面-->
        <aop:aspect ref="userServiceProxy">
            <aop:before method="before" pointcut-ref="p"/>
            <aop:after method="after" pointcut-ref="p"/>
            <aop:after-returning method="afterReturning" pointcut-ref="p"/>
            <aop:after-throwing method="afterThrowing" pointcut-ref="p"/>
            <aop:around method="around" pointcut-ref="p"/>
        </aop:aspect>
    </aop:config>

</beans>
```

3. 编写测试类

```
import org.junit.Test;
import org.learn.spring5.service.UserService;
import org.learn.spring5.service.impl.UserServiceImpl;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring5 {

    /**
     * Spring AOP实现 基于XML方式
     */
    @Test
    public void test1(){
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        UserService userService = context.getBean("userServiceImpl",
UserService.class);
        userService.add();
    }

}

```

4 JdbcTemplate

4.1 什么是JdbcTemplate?

Spring 框架对JDBC进行封装，使用JDBCTemplate方便实现对数据库的才做。

4.2 如何使用jdbcTemplate?

4.2.1 准备工作

- 引用jar包

```
        <!--jdbcTemplate相关依赖-->

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.2.8.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>5.2.8.RELEASE</version>
        </dependency>

        <dependency>
```

```
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>5.2.8.RELEASE</version>
        </dependency>
        <!--mysql 数据库驱动-->
        <!-- Mysql驱动包 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.13</version>
        </dependency>

        <!--数据库连接池-->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.14</version>
        </dependency>
```

- 配置数据库连接池

```
    <context:component-scan base-package="org.learn.spring5">
</context:component-scan>
    <!--引入外部的配置文件-->
    <context:property-placeholder location="classpath:jdbc.properties"/>
    <!--配置数据库连接-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
destroy-method="close">
        <property name="url" value="${prop.url}"></property>
        <property name="username" value="${prop.userName}"></property>
        <property name="password" value="${prop.password}"></property>
        <property name="driverClassName" value="${prop.driverClass}">
</property>
    </bean>
```

- 配置jdbcTemplate对象，注入DataSource

```
    <!--JdbcTemplate对象创建,注入dataSource对象-->
    <bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
        <!--注入dataSource对象-->
        <property name="dataSource" ref="dataSource"></property>
    </bean>
```

- 创建jdbc.properties

```
prop.driverClass=com.mysql.cj.jdbc.Driver
prop.url=jdbc:mysql://127.0.0.1:3306/spring5-demo?
characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2B8&zeroDateTimeBehavi
or=CONVERT_TO_NULL
prop.userName=root
prop.password=123456
```

- 创建service类，dao类，在dao类注入jdbcTemplate对象

```
package org.learn.spring5.service.impl;

import org.learn.spring5.dao.impl.UserDaoImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserServiceImpl implements UserService {
    //注入UserDao的对象
    @Autowired
    private UserDaoImpl userDao;

}

```

```
package org.learn.spring5.dao.impl;

import org.learn.spring5.service.impl.UserServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;

public class UserDaoImpl implements UserDao{

    //注入JdbcTemplate對象
    @Autowired
    private JdbcTemplate jdbcTemplate;
}

```

- 编写实体类User

```
package org.learn.spring5.entiy;

public class User {
    //id
    private Integer id;
    //用户名
    private String userName;
    //用户状态
    private Integer status;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUserName() {
```

```
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public Integer getStatus() {
        return status;
    }

    public void setStatus(Integer status) {
        this.status = status;
    }
}
```

- 创建数据和表结构

```
DROP TABLE IF EXISTS `t_user`;
CREATE TABLE `t_user`  (
  `id` int(11) NOT NULL,
  `user_name` varchar(22) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL,
  `status` int(11) NULL DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;

SET FOREIGN_KEY_CHECKS = 1;
```

4.2.2 具体操作

1. JdbcTemplate操作数据库-添加功能

   1）在UserDaoImpl类里添加add方法

   2）调用JdbcTemplate对象的update方法

```
package org.learn.spring5.dao.impl;

import org.learn.spring5.entiy.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class UserDaoImpl implements UserDao{

    //注入JdbcTemplate對象
    @Autowired
    private JdbcTemplate jdbcTemplate;

```

```
    public void add(User user) {
        //1.创建sql語句
        String sql="insert into t_user values(?,?,?)";
        //2.調用方法实现
        Object[] args=
{user.getId(),user.getUserName(),user.getStatus()};
        int update = jdbcTemplate.update(sql, args);
        System.out.println(update);
    }
}

```

3)编写测试程序

```
import org.junit.Test;
import org.learn.spring5.entiy.User;
import org.learn.spring5.service.impl.UserService;
import org.learn.spring5.service.impl.UserServiceImpl;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring5 {
    /**
     * 操作JdbcTemplate，添加操作
     */
    @Test
    public void testAdd() {
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        UserService userService = context.getBean("userServiceImpl",
UserServiceImpl.class);
        User user = new User();
        user.setId(1);
        user.setUserName("张三");
        user.setStatus(1);
        userService.addUser(user);

    }


}

```

程序返回结果1，标识数据添加成功！

2. JdbcTemplate操作数据库-修改功能

```
  public int update(User user) {
       String sql = "update t_user set user_name =?,status=? where id =?";
       Object[] args = {user.getUserName(), user.getStatus(), user.getId()};
       int update = jdbcTemplate.update(sql, args);

       return update;
   }
```

编写测试程序

```
    //JdbcTemplate修改操作
    @Test
    public void testUpdate() {
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        UserService userService = context.getBean("userServiceImpl",
UserServiceImpl.class);
        User user = new User();
        user.setId(1);
        user.setUserName("张四");
        user.setStatus(0);
        userService.update(user);
    }
```

3. JdbcTemplate操作数据库-查询功能

1）查询返回某一个值

```
    public int queryCount() {
        String sql = "select count(*) from t_user";
        Integer count = jdbcTemplate.queryForObject(sql, Integer.class);
        return count;
    }
```

编写测试程序

```
   //JdbcTemplate 查询操作，返回用户的数量
    @Test
    public void testQueryCount() {
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        UserService userService = context.getBean("userServiceImpl",
UserServiceImpl.class);
        int count = userService.queryCount();
        System.out.println(count);
    }
```

2）查询返回对象

```
    public User findOne(Integer id) {
        String sql = "select * from t_user where id =?";
        User user = jdbcTemplate.queryForObject(sql, new
BeanPropertyRowMapper<User>(User.class), id);
        return user;
    }
```

编写测试程序

```
    //JdbcTemplate 查询操作，返回对象根据id
    @Test
    public void testQueryById() {
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        UserService userService = context.getBean("userServiceImpl",
UserServiceImpl.class);
        User user = userService.queryById(1);
        System.out.println(user);
    }
```

3）查询返回集合

```
    //JdbcTemplate 查询操作，返回集合对象
    @Test
    public void testFindAll() {
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        UserService userService = context.getBean("userServiceImpl",
UserServiceImpl.class);
        List<User> all = userService.findAll();
        System.out.println(all);
    }
```

编写测试程序

```
    //JdbcTemplate 查询操作，返回集合对象
    @Test
    public void testFindAll() {
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        UserService userService = context.getBean("userServiceImpl",
UserServiceImpl.class);
        List<User> all = userService.findAll();
        System.out.println(all);
    }
```

4. JdbcTemplate操作数据库-删除功能

```
    public int del(Integer id) {
        String sql = "delete from t_user where id =?";
        int update = jdbcTemplate.update(sql, id);
        return update;
    }
```

编写测试程序

```
    //JdbcTemplate删除操作
    @Test
    public void testDel() {
        ApplicationContext context = new ClassPathXmlApplicationContext("bean1.xml");
        UserService userService = context.getBean("userServiceImpl", UserServiceImpl.class);
        userService.del(1);
    }
```

5. JdbcTemplate操作数据库-批量添加

```
    public void batchAdd(List<Object[]> users) {
        String sql = "insert into t_user values(?,?,?)";
        int[] ints = jdbcTemplate.batchUpdate(sql, users);
        System.out.println(Arrays.asList(ints));

    }
```

编写测试程序

```
  //JdbcTemplate 批量添加操作
    @Test
    public void batchAdd() {
        ApplicationContext context = new ClassPathXmlApplicationContext("bean1.xml");
        UserService userService = context.getBean("userServiceImpl", UserServiceImpl.class);
        List<Object[]> list = new ArrayList<Object[]>();
        Object[] o1 = {"4","name1",0};
        Object[] o2 = {"5","name2",0};
        Object[] o3 = {"6","name3",0};
        list.add(o1);
        list.add(o2);
        list.add(o3);
        userService.batchAdd(list);
    }
```

5. JdbcTemplate操作数据库-批量修改和删除

```java
//批量修改操作
  public void batchUpdate(List<Object[]> users) {
        String sql = "update t_user set user_name =?,status=? where id =?";
        int[] ints = jdbcTemplate.batchUpdate(sql, users);
        System.out.println(Arrays.asList(ints).toString());

    }
//批量删除操作
    public void batchDel(List<Object[]> users) {
        String sql = "delete from t_user where id =?";
        int[] ints = jdbcTemplate.batchUpdate(sql, users);
        System.out.println(ints);
    }
```

编写测试程序

```java
//JdbcTemplate 批量修改操作
    @Test
    public void batchUpdate() {
        ApplicationContext context = new ClassPathXmlApplicationContext("bean1.xml");
        UserService userService = context.getBean("userServiceImpl", UserServiceImpl.class);
        List<Object[]> list = new ArrayList<Object[]>();
        Object[] o1 = {"namexiugai", 1, 4};
        Object[] o2 = {"name2xiugai", 1, 5};
        Object[] o3 = {"name3xiugai", 1, 6};
        list.add(o1);
        list.add(o2);
        list.add(o3);
        userService.batchUpdate(list);
    }

    //JdbcTemplate 批量删除操作
    @Test
    public void batchDel() {
        ApplicationContext context = new ClassPathXmlApplicationContext("bean1.xml");
        UserService userService = context.getBean("userServiceImpl", UserServiceImpl.class);
        List<Object[]> list = new ArrayList<Object[]>();
        Object[] o1 = {4};
        Object[] o2 = {5};
        Object[] o3 = {6};
        list.add(o1);
        list.add(o2);
        list.add(o3);
        userService.batchDel(list);
    }

```

5 事务管理

5.1 什么是事务?

事务是数据库操作最基本单元，逻辑上的一组操作，要么都成功，如果有一个失败所有操作都失败。

5.2 事务特性(ACID)

原子性：不可分割，要么成功，要么都失败。

一致性：操作之前和操作之后的总量是不变得。

隔离性：多事务间不影响。

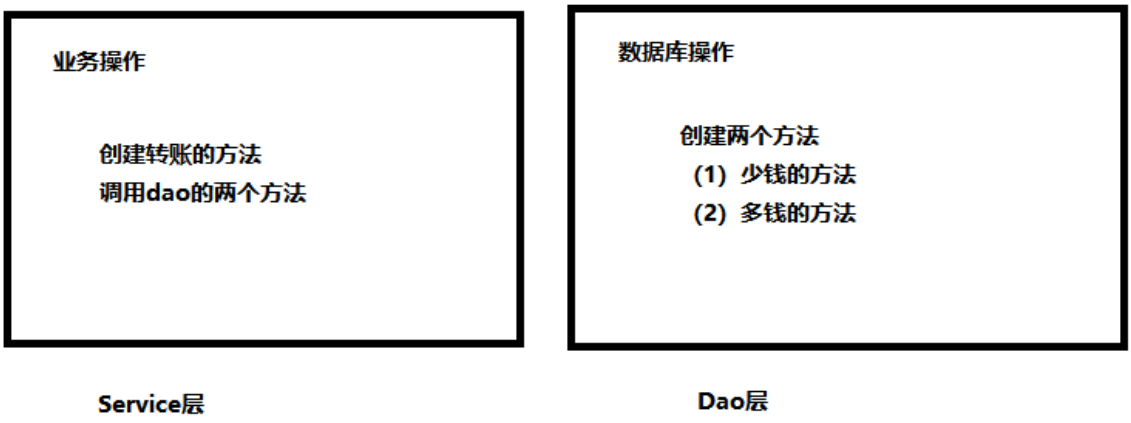
持久性：事务提交后，表中数据发生持久变化。

5.3 事务操作

典型场景：银行转账

场景描述：小明去银行转账给小红100元，小明执行转账方法，方法执行后，小明的账户减少100元，小红的账户增加100元。

搭建事务操作的环境



- 创建数据库表，并添加记录

```
DROP TABLE IF EXISTS `t_account`;
CREATE TABLE `t_account`  (
  `id` int(11) NOT NULL,
  `user_name` varchar(55) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '用户名',
  `money` decimal(11, 2) NULL DEFAULT NULL COMMENT '账户金额',
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;

-- ----------------------------
-- Records of t_account
-- ----------------------------
INSERT INTO `t_account` VALUES (1, '小明', 1000.00);
INSERT INTO `t_account` VALUES (2, '小红', 1000.00);

```

```
SET FOREIGN_KEY_CHECKS = 1;

```

创建UserService接口和实现类、UserDao接口和实现类，配置Spring的XML文件

```
package org.learn.spring5.service;

public interface UserService {

    public void transferAccount();
}

```

```
package org.learn.spring5.service.impl;

import org.learn.spring5.dao.UserDao;
import org.learn.spring5.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDao userDao;
    public void transferAccount() {
        //小明转账100，减少100元
        userDao.reduceMoney();
        //小红账户增加100元
        userDao.addMoney();
    }
}
```

```
package org.learn.spring5.dao.impl;

import org.learn.spring5.dao.UserDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class UserDaoImpl implements UserDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;
    public int reduceMoney() {
    String sql=" update t_account set money=money-100 where user_name ='小
明'";
        Object[] args = {};
        int update = jdbcTemplate.update(sql, args);

        return update;
    }

```

```
    public int addMoney() {
        String sql=" update t_account set money=money+100 where user_name ='小红'";
        Object[] args = {};
        int update = jdbcTemplate.update(sql, args);


        return update;
    }
}

```

```
package org.learn.spring5.dao;

public interface UserDao {

    //账户金额减少的方法
    public int reduceMoney();
    //账户金额增加的方法
    public int addMoney();
}

```

```
package org.learn.spring5.dao.impl;

import org.learn.spring5.dao.UserDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class UserDaoImpl implements UserDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;
    public int reduceMoney() {
    String sql=" update t_account set money=money-100 where user_name ='小明'";
        Object[] args = {};
        int update = jdbcTemplate.update(sql, args);

        return update;
    }

    public int addMoney() {
        String sql=" update t_account set money=money+100 where user_name ='小红'";
        Object[] args = {};
        int update = jdbcTemplate.update(sql, args);


        return update;
    }
}
```

```

```

```
<?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
https://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.learn.spring5">
</context:component-scan>
    <!--引入外部的配置文件-->
    <context:property-placeholder location="classpath:jdbc.properties"/>
    <!--配置数据库连接-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
destroy-method="close">
        <property name="url" value="${prop.url}"></property>
        <property name="username" value="${prop.userName}"></property>
        <property name="password" value="${prop.password}"></property>
        <property name="driverClassName" value="${prop.driverClass}">
</property>
    </bean>

    <!--JdbcTemplate对象创建,注入dataSource对象-->
    <bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
        <!--注入dataSource对象-->
        <property name="dataSource" ref="dataSource"></property>
    </bean>

</beans>
```

从上面代码我们可以分析出，Service层用于处理业务，具体有两部操作，小明转账，账户减少100元，和小红账户增加100元

Dao层用于操作数据库，分别执行了小明账户减少100和小红账户增加100的数据操作，**如果我们没有使用事务发生异常会发生什么问题呢?**

假如在处理如上业务逻辑时1.小明转账，账户减少100，当执行完这步操作时，网络突然中断，没有执行下一步操作，这时候数据库会执

行小明账户减少100元的操作，结果是小明的账户减少100，而小红的账户金额没有发生变化。这种情况明显是错误的。正常情况应该是如果转账成功，小明账户减少100，小红账户增加100。如果发生异常，小明和小红的账户金额应该保持不变。如何处理这个问题呢？事务的引入就可以解决这个问题，让我们在回顾一下事务的定义：逻辑上的一组操作，要么都成功，如果有一个失败所有操作都失败。

事务场景引入

增加事务的步骤

在要增加事务的方法内添加如下的代码流程

1、开启事务

2、进行业务操作

3、如果没有发生异常，事务提交

4、如果在处理业务逻辑时出现异常，事务回滚

```
    @Autowired
     private UserDao userDao;
     public void transferAccount() {
         try{
         //1.开启事务
         //2.业务逻辑的处理
         //小明转账100，减少100元
         userDao.reduceMoney();
         //小红账户增加100元
         userDao.addMoney();
         }catch(Exception e){
         //3.如果存在异常，事务回滚
         }
         //4.事务提交

     }
```

思考一个问题，如果系统中存在很多功能，每个功能的业务逻辑处理的Service类都需要增加事务代码的编写，会发生什么问题?

第一，代码会变得非常臃肿，冗余。

第二，业务方法功能变得复杂而不清晰。

然而，Spring集成了对事务的管理，恰恰解决了这些问题，大大减少了程序员的代码量，也对事务有了很好的管理控制。业务层只需要专

心业务的处理。

5.4 如何用Spring实现事务的管理?

1、事务要添加到三层结构里的Service层（业务逻辑层）

2、Spring进行事务管理操作有两种方式

1）编程式事务管理（不建议使用，该方式是在程序里编写事务相关的代码）

2）声明式事务管理（建议使用，通过配置的方式操作事务）

3、声明式事务管理具体实现有两种方式

1）基于注解的方式（推荐使用）

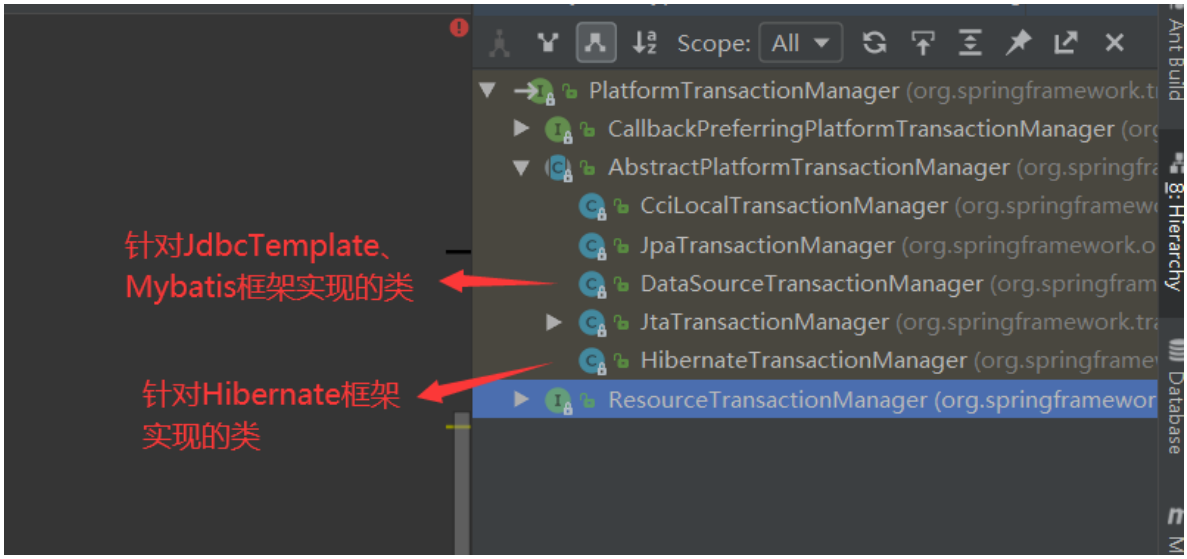
2）基于xml配置文件的方式

4、在Spring进行声明式事务管理，底层使用的AOP原理

5、Springs事务管理相关API

1）提供一个接口，代表事务管理器，这个接口针对不同的数据持久框架提供不同的实现类

```
PlatformTransactionManager
```



- **基于注解声明式事务管理的具体实现**

1、在Spring配置文件配置事务管理器

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

2、在Spring配置文件开启事务注解

1）增加命名空间

```
<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: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
https://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd">
```

2）增加开启事务注解的设置

```
   <!--开启事务的注解-->
    <tx:annotation-driven transaction-manager="transactionManager">
</tx:annotation-driven>
```

3、在Service类或者该类的方法上增加事务注解@Transactional

类上添加，表明事务在该类所有的方法生效

方法上添加，表明事务只在该方法上生效

```java
package org.learn.spring5.service.impl;

import org.learn.spring5.dao.UserDao;
import org.learn.spring5.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
@Transactional
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDao userDao;
    public void transferAccount() {

        //小明转账100，减少100元
        userDao.reduceMoney();
        //小红账户增加100元
        userDao.addMoney();
    }
}

```

4.编写测试程序

```java
import org.junit.Test;
import org.learn.spring5.service.UserService;
import org.learn.spring5.service.impl.UserServiceImpl;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring5 {



    /**
     * 模拟转账操作(事务管理的引入)
     */
    @Test
    public void testAccount() {
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        UserService userService = context.getBean("userServiceImpl",
UserServiceImpl.class);
        userService.transferAccount();//转账操作
    }




}

```

我们在UserServiceImpl类的转账方法里模拟一个异常，让程序出错，看看我们的事务配置的是否生效，正常情况应该是如果程序出现异常，service类对应的方法中执行数据操作的方法将会全部失败，数据不发生变化。

@Transactional注解相关属性的介绍

1）propagation：传播行为，描述由某一个事务方法被嵌套另一个方法时事务如何传播。

传播行为属性值设置有七种如下：

传播属性	描述
REQUIRED	如果有事务在运行，当前的方法就在这个事务内运行，否则，就启动一个新的事务，并在自己的事务内运行
REQUIRED_NEW	当前的方法必须启动新事务，并在它自己的事务内运行．如果有事务正在运行，应该将它挂起
SUPPORTS	如果有事务在运行，当前的方法就在这个事务内运行．否则它可以不运行在事务中．
NOT_SUPPORTE	当前的方法不应该运行在事务中．如果有运行的事务，将它挂起
MANDATORY	当前的方法必须运行在事务内部，如果没有正在运行的事务，就抛出异常
NEVER	当前的方法不应该运行在事务中．如果有运行的事务，就抛出异常
NESTED	如果有事务在运行，当前的方法就应该在这个事务的嵌套事务内运行．否则，就启动一个新的事务，并在它自己的事务内运行．

默认是REQUIRED

```
1 @Transactional(propagation = Propagation.REQUIRED)
```

2）isolation：隔离级别，解决隔离性的问题，多事务（并发操作）操作之间不会产生影响

如果不考虑隔离性，会出现很多问题。

例如出现三个读的问题，

1）脏读，一个未提交事务读取到另一个未提交事务的数据。

2）不可重复读，一个未提交的事务读取到另一个已提交事务中修改的数据。

3）虚（幻）读，一个未提交的事务读取到另一个已提交事务中添加的数据。

Spring事务管理中隔离级别的设置就是针对如上问题进行处理的，具体设置的值如下：

英文	中文	更新丢失	脏读	不可重复读	幻读
Read Uncommited	读未提交	不会出现	会出现	会出现	会出现
Read Commited	读已提交	不会出现	不会出现	会出现	会出现
Repeatable Read	可重复读	不会出现	不会出现	不会出现	会出现
Serializable	串行化	不会出现	不会出现	不会出现	不会出现

```
@Transactional(propagation = Propagation.REQUIRED ,isolation = Isolation.REPEATABLE_READ)
```

3）timeout：超时时间，事务要在一个规定时间内提交，如果不提交就会超时，事务就会回滚，默认值为-1，表示不超时。
4）readOnly：是否只读，默认值false，如果 设置true，只能查询，不能添加、删除、修改操作。

```
@Transactional(readOnly = false,propagation = Propagation.REQUIRED ,isolation = Isolation.REPEATABLE_READ)
```

5）rollbackFor：回滚，出现哪些异常进行事务回滚。
6）noRollbackFor：不回滚，出现哪些异常，事务不进行事务回滚。

```
@Transactional(readOnly = false,propagation = Propagation.REQUIRED ,isolation = Isolation.REPEATABLE_READ,rollbackFor = Exception.class)
```

- **基于XML配置文件声明式事务管理的具体实现**

在Spring的配置文件中进行配置，具体内容如下：

1）配置事务管理

2）配置通知

3）配置切入点和切面

```
<?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:tx="http://www.springframework.org/schema/tx"
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
https://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
https://www.springframework.org/schema/aop/spring-aop.xsd">

    <context:component-scan base-package="org.learn.spring5">
</context:component-scan>
    <!--引入外部的配置文件-->
    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!--开启事务的注解-->

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


```

```
    <!--配置数据库连接-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
destroy-method="close">
        <property name="url" value="${prop.url}"></property>
        <property name="username" value="${prop.userName}"></property>
        <property name="password" value="${prop.password}"></property>
        <property name="driverClassName" value="${prop.driverClass}">
</property>
    </bean>

    <!--JdbcTemplate对象创建,注入dataSource对象-->
    <bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
        <!--注入dataSource对象-->
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!--1.创建事务管理器-->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <!--注入数据源-->
        <property name="dataSource" ref="dataSource"></property>
    </bean>
    <!--2.配置通知-->
    <tx:advice id="txadvice">
        <!--配置事务参数-->
        <tx:attributes>
            <!--指定哪种规则的方法上面添加事务-->
            <tx:method name="transferAccount" propagation="REQUIRED"/>
        </tx:attributes>
    </tx:advice>

    <!--3.配置切入点和切面-->
    <aop:config>
        <!--配置切入点-->
<aop:pointcut id="pt" expression="execution(*
org.learn.spring5.service.impl.UserServiceImpl.*(..))"></aop:pointcut>
        <!--配置切面-->
        <aop:advisor advice-ref="txadvice" pointcut-ref="pt"></aop:advisor>
    </aop:config>

</beans>
```

- **完全注解事务管理实现**

添加配置类SpringConfig，删除xml配置文件

```
package org.learn.spring5.config;

import com.alibaba.druid.pool.DruidDataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import
org.springframework.transaction.annotation.EnableTransactionManagement;
```

```
10
11  @Configuration
12  @ComponentScan(basePackages = "org.learn.spring5")  //开启组件扫描
13  @EnableTransactionManagement  //开启事务
14  public class SpringConfig {
15
16      //创建数据库连接池
17      @Bean
18      public DruidDataSource getDruidDataSource(){
19          DruidDataSource druidDataSource = new DruidDataSource();
20          druidDataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
21          druidDataSource.setUrl("jdbc:mysql://127.0.0.1:3306/spring5-demo?
    characterEncoding=utf8&useSSL=false&serverTimezone=GMT%2B8&zeroDateTimeBeha
    vior=CONVERT_TO_NULL");
22          druidDataSource.setUsername("root");
23          druidDataSource.setPassword("123456");
24          return  druidDataSource;
25      }
26      //创建JdbcTemplate
27      @Bean
28      public JdbcTemplate JdbcTemplate(DruidDataSource druidDataSource){
29          JdbcTemplate jdbcTemplate = new JdbcTemplate();
30          jdbcTemplate.setDataSource(druidDataSource);
31          return  jdbcTemplate;
32      }
33      @Bean
34      public DataSourceTransactionManager
    getDataSourceTransactionManager(DruidDataSource druidDataSource){
35          DataSourceTransactionManager transactionManager = new
    DataSourceTransactionManager();
36          transactionManager.setDataSource(druidDataSource);
37
38          return transactionManager;
39      }
40
41  }
42
```

编写测试程序

```
1       /**
2        * 模拟转账操作(完全註解形式)
3        */
4       @Test
5       public void testAccount() {
6           ApplicationContext context = new
    AnnotationConfigApplicationContext(SpringConfig.class);
7           UserService userService = context.getBean("userServiceImpl",
    UserService.class);
8           userService.transferAccount();//转账操作
9       }
10
```

6 Spring5新特性

JDK 8+和Java EE7+以上版本

- 整个框架的代码基于java8
- 通过使用泛型等特性提高可读性
- 对java8提高直接的代码支撑
- 运行时兼容JDK9
- Java EE 7API需要Spring相关的模块支持
- 运行时兼容Java EE8 API
- 取消的包,类和方法
- 包 beans.factory.access
- 包 dbc.support.nativejdbc
- 从spring-aspects 模块移除了包mock.staicmock,不在提AnnotationDrivenStaticEntityMockingControl支持
- 许多不建议使用的类和方法在代码库中删除

核心特性

JDK8的增强：

- 访问Resuouce时提供getFile或和isFile防御式抽象
- 有效的方法参数访问基于java 8反射增强
- 在Spring核心接口中增加了声明default方法的支持一贯使用JDK7 Charset和StandardCharsets的增强
- 兼容JDK9
- Spring 5.0框架自带了通用的日志封装
- 持续实例化via构造函数(修改了异常处理)
- Spring 5.0框架自带了通用的日志封装
- spring-jcl替代了通用的日志，仍然支持可重写
- 自动检测log4j 2.x, SLF4J, JUL（java.util.Logging）而不是其他的支持
- 访问Resuouce时提供getFile或和isFile防御式抽象
- 基于NIO的readableChannel也提供了这个新特性

核心容器

- 支持候选组件索引(也可以支持环境变量扫描)
- 支持@Nullable注解
- 函数式风格GenericApplicationContext/AnnotationConfigApplicationContext
- 基本支持bean API注册
- 在接口层面使用CGLIB动态代理的时候，提供事物，缓存，异步注解检测
- XML配置作用域流式
- Spring WebMVC
- 全部的Servlet 3.1 签名支持在Spring-provied Filter实现
- 在Spring MVC Controller方法里支持Servlet4.0 PushBuilder参数
- 多个不可变对象的数据绑定(Kotlin/Lombok/@ConstructorPorties)
- 支持jackson2.9
- 支持JSON绑定API
- 支持protobuf3
- 支持Reactor3.1 Flux和Mono

6.1 整合日志框架

6.1.1 spring5整合log4j2日志工具

首先我们还是用事务管理中创建的项目，也就是转账的例子给它增加记录日志的功能。

第一步，引入相关依赖

```
<!-- log4j2日志相关jar包引用-->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.30</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-api</artifactId>
            <version>2.11.2</version>
        </dependency>

        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>2.11.2</version>
        </dependency>

        <!--用于slf4j与log4j2保持桥接 -->
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-slf4j-impl</artifactId>
            <version>2.11.2</version>
        </dependency>
```

第二步，创建log4j2.xml配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
       日志输出级别，共有8个级别，按照从低到高为：All < Trace < Debug < Info < Warn < Error < Fatal < OFF.
-->
<Configuration status="warn">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
        </Console>
    </Appenders>
    <Loggers>
        <Root level="info">
            <AppenderRef ref="Console"/>
        </Root>
    </Loggers>
</Configuration>
```

第三步，创建Logger对象，输入日志信息。

```java
package org.learn.spring5.service.impl;

import org.learn.spring5.dao.UserDao;
import org.learn.spring5.service.UserService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;


@Service
public class UserServiceImpl implements UserService {

    private static final Logger log=
LoggerFactory.getLogger(UserServiceImpl.class);

    @Autowired
    private UserDao userDao;
    public void transferAccount() {
        log.info("执行了 transferAccount 方法");
        //小明转账100，减少100元
        userDao.reduceMoney();
        int i = 10/0;
        //小红账户增加100元
        userDao.addMoney();
    }
}

```

执行测试程序

```java
import org.junit.Test;
import org.learn.spring5.service.UserService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring5 {


    /**
     * Spring5整合log4j2
     */
    @Test
    public void testAccount() {
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        UserService userService = context.getBean("userServiceImpl",
UserService.class);
        userService.transferAccount();//转账操作
    }



}

```

返回结果

```
19:05:51.899 [main] INFO  com.alibaba.druid.pool.DruidDataSource - {dataSource-1} inited
19:05:52.254 [main] INFO  org.learn.spring5.service.impl.UserServiceImpl - 执行了 transferAccount 方法
```

结果表明，我们的日志信息按照log4j2配置文件设置的格式成功输出，log4j2框架整合成功。

6.2 Nullable注解和函数式风格编程

@NonNull

使用在字段，方法参数或方法的返回值。表示不能为空

@NonNullFields

使用在包级别，并且是该包下类的字段不能为空。

当一个类中的字段使用了太多的NonNull时可以考虑使用@NonNullFields注解，使用该注解必须先定义一个名为`package-info.java`的文件，例如：

package-info.java

```
@NonNullApi
@NonNullFields
package org.springframework.mail;

import org.springframework.lang.NonNullApi;
import org.springframework.lang.NonNullFields;
```

@Nullable

使用在字段，方法参数或方法的返回值。表示可以为空。

当一个类的包被`@NonNullFields`或`@NonNullApi`注解，而我们想要从包级别指定的非null约束中免除某些字段，方法，返回值时可以使用`@Nullable`

@NonNullApi

和`@NonNullFields`一样使用在包级别，但是区别是它作用是该包下的类的方法参数和返回值不能为空

当一个类中的方法参数和返回值使用了太多的NonNull时可以考虑使用@NonNullFields注解，使用该注解必须先定义一个名为`package-info.java`的文件，形式同上。

- **函数式风格编程**

函数式风格创建对象，并交给Spring进行管理

1.创建GenericApplicationContext对象

2.调用context的registerBean方法注册对象

3.获取在Srping注册的对象

```
 @Test
    public void test2() {
        //1.手动实例化对象
        User user = new User();
        //2.创建GenericApplicationContext对象
        GenericApplicationContext context = new
GenericApplicationContext();
        context.refresh();
        //3.调用context的registerBean方法注册对象
        context.registerBean("user1", User.class, () -> new User());
        User bean = (User) context.getBean("user1");
        System.out.println(bean);

    }
```

6.3 整合JUnit5单元测试框架

第一步，引入相关依赖

```
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>5.2.8.RELEASE</version>
        </dependency>
            <!-- junit5 -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>RELEASE</version>
            <scope>compile</scope>
        </dependency>
```

第二步，创建测试类，添加junit5注解

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.learn.spring5.service.UserService;
import org.learn.spring5.service.impl.UserServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;

@ExtendWith(SpringExtension.class)
@ContextConfiguration("classpath:bean1.xml")
public class TestSpring5 {

    @Autowired
    private UserService userService;
    /**
     * Spring5整合junit5
     */
    @Test
```

```
    public void testAccount() {
        userService.transferAccount();//转账操作
    }


}

```