Spring5

spring概念

> 1、Spring 是轻量级的开源的 JavaEE 框架
>
> 2、Spring 可以解决企业应用开发的复杂性
>
> 3、Spring 有两个核心部分：IOC 和 Aop
>
> （1）IOC：控制反转，把创建对象过程交给 Spring 进行管理
>
> （2）Aop：面向切面，不修改源代码进行功能增强
>
> 4、Spring 特点
>
> （1）方便解耦，简化开发
>
> （2）Aop 编程支持
>
> （3）方便程序测试
>
> （4）方便和其他框架进行整合
>
> （5）方便进行事务操作
>
> （6）降低 API 开发难度

> **依赖下载：**
>
> 
>
> 
>
> 
>
> **选择合适的版本下载**
>
> 

IOC容器

1、IOC底层原理

1、什么是IOC

（1）控制反转，把对象创建和对象之间的调用过程，交给 Spring 进行管理

（2）使用 IOC 目的：为了耦合度降低

（3）做入门案例就是 IOC 实现————>demo1

2、IOC 底层原理

（1）xml 解析、工厂模式、反射

3、画图讲解 IOC 底层原理

2、IOC接口（BeanFactory 接口）

1、IOC 思想基于 IOC 容器完成，IOC 容器底层就是对象工厂

2、Spring 提供 IOC 容器实现两种方式：（两个接口）

（1）BeanFactory：IOC 容器基本实现，是 Spring 内部的使用接口，不提供开发人员进行使用

加载配置文件时候不会创建对象，在获取对象（使用）才去创建对象

（2）ApplicationContext：BeanFactory 接口的子接口，提供更多更强大的功能，一般由开发人 员进行使用

加载配置文件时候就会把在配置文件对象进行创建

3、ApplicationContext 接口有实现类

ClassPathXmlApplicationContext：项目中src下的xml文件所在路径（**相对路径**）

FileSystemXmlApplicationContext：在某个盘下的xml所在的**绝对路径**

1、什么是 Bean 管理

（1）Bean 管理指的是两个操作 （以下两个操作）

（2）Spring 创建对象

（3）Spirng 注入属性

2、Bean 管理操作有两种方式

（1）基于 xml 配置文件方式实现

（2）基于注解方式实现

3、IOC 操作 Bean 管理（基于 xml 方式）

1、基于 xml 方式创建对象

（1）在 spring 配置文件中，使用 bean 标签，标签里面添加对应属性，就可以实现对象创建

（2）在 bean 标签有很多属性，介绍常用的属性

id 属性：唯一标识

class 属性：类全路径（包类路径）

name属性：作用和和id一样，name可以加一些特殊符号，id不可以（此属性现在用的不多）

（3）创建对象时候，默认也是执行无参数构造方法完成对象创建

```
<!--配置User对象创建
 id：唯一标识
 class：类的全路径    包名+类名
 -->
<bean id="user" class="com.fsir.spring5.User"></bean>
```

2、基于 xml 方式注入属性

（1）DI：依赖注入，就是注入属性

```
        **第一种注入方式：使用 set 方法进行注入**
```

1、创建类，定义属性和对应的 set 方法

```
public class Book {
  //创建属性
  private String bname;
  private String bauthor;
  //创建属性对应的 set 方法
  public void setBname(String bname) {
    this.bname = bname;
  }
  public void setBauthor(String bauthor) {
    this.bauthor = bauthor;
  }
}
```

2、在 spring 配置文件配置对象创建，配置属性注入

```
<!--配置文件中属性注入-->
<bean id="book" class="com.fsir.spring5.Book">
    <!--使用property完成属性注入
        name：类中属性的名称
     value：给此属性注入的值
    -->
 <property name="bname" value="易筋经"></property>
 <property name="bauthor" value="达摩老祖"></property>
</bean>
```

第二种注入方式：有参构造进行注入

1、创建类，定义属性，创建属性对应有参数构造方法

```
public class Orders {
  //属性
  private String oname;
  private String address;
  //有参数构造
  public Orders(String oname,String address) {
      this.oname = oname;
      this.address = address;
  }
}
```

2、在 spring 配置文件中进行配置

```
<!--配置文件中属性注入-->
<bean id="book" class="com.fsir.spring5.Book">
 <!--使用constructor-arg完成有参构造注入
        name：类中有参构造的名称
        index：类中有参构造对应的索引，数字0开始
     value：给对应有参构造注入的值
    -->
    <constructor-arg name="bname" value="易筋经"></constructor-arg>
 <constructor-arg name="bauthor" value="达摩老祖"></constructor-arg>
</bean>
```

第三种注入方式：p名称空间注入（不常用，了解），底层仍然是set方式

1、使用 p 名称空间注入，可以简化基于 xml 配置方式

先添加p名称空间在配置文件中

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

属性注入，在bean标签里面进行操作

```
<bean id="book" class="com.fsir.spring5.Book" p:bname="九阳神功"
p:bauthor="饮酒僧"></bean>
```

IOC操作Bean管理（xml注入其他类型属性：对象）

1、字面量：属性的初始值

1、null值

```
<bean id="book" class="com.fsir.spring5.Book">
    <!--使用property完成属性注入
        name：类中属性的名称
     value：给此属性注入的值
    -->
 <property name="bname" value="易筋经"></property>
 <property name="bauthor" value="达摩老祖"></property>
 <property name="address">
    <null/>
 </property>
</bean>
```

2、属性值包含特殊符号

```
<bean id="book" class="com.fsir.spring5.Book">
    <!--使用property完成属性注入
        name：类中属性的名称
     value：给此属性注入的值
    -->
 <property name="bname" value="易筋经"></property>
 <property name="bauthor" value="达摩老祖"></property>
 <property name="address" value="&lt;&lt;南京&gt;&gt;"></property>
 <!--不想转义也可以用一下方式：把带特殊符号内容写到 <![CDATA[]]>-->
 <property name="address">
    <value>
        <![CDATA[<<南京>>]]>
     </value>
 </property>
</bean>
```

2、注入属性————外部Bean

（1）创建两个类 service 类和 dao 类

（2）在 service 调用 dao 里面的方法

（3）在 spring 配置文件中进行配置

对应demo1中service的实现类

```
//创建UserDao类型属性，生成set方法
 private UserDao dao;

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

 public void add() {
     System.out.println("service add......");
```

```
        //原始方式：  创建UserDao对象
        /*UserDao userDao = new UserDaoImpl();
        userDao.update();*/

        dao.update();
    }
```

对应demo1中bean2.xml配置文件————>测试在testDemo中TestBean文件

```
<!--1、service和dao实现类的对象创建-->
 <bean id="userService"
class="com.fsir.spring5.service.serviceImpl.UserServiceImpl">
        <!--注入userDao对象
            name属性值：对应类里面属性名称
            ref属性：创建userDao对象bean标签id值
        -->
        <property name="dao" ref="userDaoImpl"></property>

 </bean>
 <bean id="userDaoImpl" class="com.fsir.spring5.dao.UserDaoImpl"></bean>
```

3、注入属性————内部Bean

（1）一对多关系：部门和员工

一个部门有多个员工，一个员工属于一个部门

部门是一，员工是多

（2）在实体类之间表示一对多关系，员工表示所属部门，使用对象类型属性进行表示

```
ublic class Emp {
 private String ename;
 private String gender;

 //员工属于某一个部门，使用对象形式表示
  private Dept dept;

 public void setEname(String ename) {
     this.ename = ename;
 }

 public void setGender(String gender) {
     this.gender = gender;
 }

 public Dept getDept() {
     return dept;
 }

 public void setDept(Dept dept) {
     this.dept = dept;
 }

 public void add(){
     System.out.println(ename + "----" + gender + "----" + dept);
 }
}
```

```
public class Dept {
 private String dname;

 public void setDname(String dname) {
     this.dname = dname;
 }

 @Override
 public String toString() {
     return "Dept{" +
             "dname='" + dname + '\'' +
             '}';
 }
}
```

（3）在 spring 配置文件中进行配置————>bean3.xml————>测试类：TestDeptAndEmp

```
<!--内部bean-->
 <bean id="emp" class="com.fsir.spring5.bean.Emp">
     <!--设置两个普通属性-->
     <property name="ename" value="lucy"></property>
     <property name="gender" value="女"></property>
     <!--设置对象l类型属性-->
     <property name="dept">
         <bean id="dept" class="com.fsir.spring5.bean.Dept">
             <property name="dname" value="安保部"></property>
         </bean>
     </property>
 </bean>
```

4、注入属性————级联赋值

```
<!--级联赋值-->
 <bean id="emp" class="com.fsir.spring5.bean.Emp">
     <!--设置两个普通属性-->
     <property name="ename" value="lucy"></property>
     <property name="gender" value="女"></property>
     <!--级联赋值-->
     <property name="dept" ref="dept"></property>
     <!--当外层的dept没有设置属性时，也可以这样设置，同样的效果；但是需要生成get方法-->
     <property name="dept.dname" value="技术部"></property>

 </bean>
 <bean id="dept" class="com.fsir.spring5.bean.Dept">
     <property name="dname" value="财务部"></property>
 </bean>
```

IOC操作Bean管理（xml注入其他类型属性：集合）

对应demo2模块

> 1、注入数组类型属性
>
> 2、注入List集合类型属性
>
> 3、注入Map集合类型属性
>
> 创建类，定义数组、list、map、set 类型属性，生成对应 set 方法

```
public class Stu {
 //数组类型的属性
 private String[] courses;

 //list集合类型
 private List<String> list;

 //setj集合类型
 private Set<String> set;

 //map类型集合
 private Map<String, String> maps;

 public void setCourses(String[] courses) {
     this.courses = courses;
 }

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

 public void setSet(Set<String> set) {
     this.set = set;
 }

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

 public void test(){
     System.out.println(Arrays.toString(courses));
     System.out.println(list);
     System.out.println(set);
     System.out.println(maps);
 }
}
```

```
<!--集合类型属性注入-->
 <bean id="stu" class="com.fsir.spring5.collectionType.beans.Stu">
     <!--数组类型属性注入-->
     <property name="courses">
         <array>
             <value>java课程</value>
             <value>数据库课程</value>
```

```
        </array>
    </property>

    <!--list类型属性注入-->
    <property name="list">
        <list>
            <value>张三</value>
            <value>小三</value>
        </list>
    </property>

    <!--set类型属性注入-->
    <property name="set">
        <set>
            <value>李四</value>
            <value>小四</value>
        </set>
    </property>

    <!--map类型属性注入-->
    <property name="maps">
        <map>
            <entry key="JAVA" value="java"></entry>
            <entry key="PHP" value="php"></entry>
        </map>
    </property>
</bean>
```

数组或集合中存对象的方式

```
public class Course {
 //课程名称
 private String cname;

 public void setCname(String cname) {
     this.cname = cname;
 }

 @Override
 public String toString() {
     return "Course{" +
             "cname='" + cname + '\'' +
             '}';
 }
}

public class Stu {
 //数组类型的属性
 private String[] courses;

 //list集合类型
 private List<String> list;

 //setj集合类型
 private Set<String> set;
```

```java
    //map类型集合
    private Map<String, String> maps;

    //学生所学的多门课程
    private List<Course> coursesList;

    public void setCourses(String[] courses) {
        this.courses = courses;
    }

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

    public void setSet(Set<String> set) {
        this.set = set;
    }

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

    public void setCoursesList(List<Course> coursesList) {
        this.coursesList = coursesList;
    }

    public void test(){
        System.out.println(Arrays.toString(courses));
        System.out.println(list);
        System.out.println(set);
        System.out.println(maps);
        System.out.println(coursesList);
    }
}
```

```xml
<!--集合类型属性注入-->
    <bean id="stu" class="com.fsir.spring5.collectionType.beans.Stu">
        <!--数组类型属性注入-->
        <property name="courses">
            <array>
                <value>java课程</value>
                <value>数据库课程</value>
            </array>
        </property>

        <!--list类型属性注入-->
        <property name="list">
            <list>
                <value>张三</value>
                <value>小三</value>
            </list>
        </property>
```

```
    <!--set类型属性注入-->
    <property name="set">
        <set>
            <value>李四</value>
            <value>小四</value>
        </set>
    </property>

    <!--map类型属性注入-->
    <property name="maps">
        <map>
            <entry key="JAVA" value="java"></entry>
            <entry key="PHP" value="php"></entry>
        </map>
    </property>

    <!--list中注入对象-->
    <property name="coursesList">
        <list>
            <ref bean="course1"></ref>
            <ref bean="course2"></ref>
        </list>
    </property>
</bean>

<!--创建多个course对象-->
<bean id="course1" class="com.fsir.spring5.collectionType.beans.Course">
    <property name="cname" value="Spring5框架"></property>
</bean>
<bean id="course2" class="com.fsir.spring5.collectionType.beans.Course">
    <property name="cname" value="Mybatis框架"></property>
</bean>
```

把集合中部分提取出来

在 spring 配置文件中引入名称空间 util

```
<?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
http://www.springframework.org/schema/util/spring-util.xsd">

</beans>
```

```java
public class Book {
 private List<String> list;

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

 public void test(){
     System.out.println(list);
 }


}
```

使用 util 标签完成 list 集合注入提取————>测试在testDemo包内TestcollectionBean

```xml
<!--1 提取 list 集合类型属性注入-->
<util:list id="bookList">
  <value>易筋经</value>
 <value>九阳真经</value>
  <value>九阳神功</value>
</util:list>

 <!--2 提取 list 集合类型属性注入使用-->
 <bean id="book" class="com.fsir.spring5.collectionType.beans.Book">
     <property name="list" ref="bookList"></property>
 </bean>
```

IOC 操作 Bean 管理（FactoryBean:工厂）

Spring有两种类型bean，一种普通bean；另一种工厂bean（FactoryBean）

普通 bean：在配置文件中定义 bean 类型就是返回类型

工厂 bean：在配置文件定义 bean 类型可以和返回类型不一样

第一步 创建类，让这个类作为工厂 bean，实现接口 FactoryBean

第二步 实现接口里面的方法，在实现的方法中定义返回的 bean 类型

```java
public class Mybean implements FactoryBean {

 /**
     * 返回类型，返回的bean实例
     * @return
     * @throws Exception
     */
    @Override
    public Course getObject() throws Exception {
        Course course = new Course();
        course.setCname("abc");
        return course;
    }
```

```
    /**
     * 返回类型
     * @return
     */
    @Override
    public Class<?> getObjectType() {
        return null;
    }

    /**
     * 是否是单例
     * @return
     */
    @Override
    public boolean isSingleton() {
        return false;
    }
}

//测试方法，主要在实现类中返回的什么类型，就可以使用什么类型
@Test
    public void testByBean3(){
        //1、加载spring配置文件；参数为xml所在路径及名称
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean3.xml");
        //2、获取配置文件的对象；参数为配置文件bean标签的id名称
        Course myBean = context.getBean("myBean", Course.class);
        System.out.println(myBean);
    }
```

```
<bean id="myBean"
class="com.fsir.spring5.collectionType.factroyBean.Mybean"></bean>
```

IOC容器-Bean管理（bean的作用域）

1、在 Spring 里面，设置创建 bean 实例是单实例还是多实例

2、在 Spring 里面，默认情况下，bean 是单实例对象

```
 @Test
 public void testByBean2(){
     //1、加载spring配置文件；参数为xml所在路径及名称
     ApplicationContext context = new
ClassPathXmlApplicationContext("bean2.xml");
     //2、获取配置文件的对象；参数为配置文件bean标签的id名称
     Book book = context.getBean("book", Book.class);
     System.out.println(book);
     book.test();

     //测试单实例还是多实例;单实例：指向同一个地址，多实例：指向不同的地址
     Book book1 = context.getBean("book", Book.class);
     System.out.println(book);
 }
```

3、设置单实例还是多实例

（1）在 spring 配置文件 bean 标签里面有属性（scope）用于设置单实例还是多实例　　（2）scope 属性值 第一个值 默认值，singleton，表示是单实例对象 第二个值 prototype，表示是多实例对象

```
<bean id="book" class="com.fsir.spring5.collectionType.beans.Book"
scope="prototype">
     <property name="list" ref="bookList"></property>
 </bean>
```

（3）singleton 和 prototype 区别

第一 singleton 单实例，prototype 多实例

第二 设置 scope 值是 singleton 时候，加载 spring 配置文件时候就会创建单实例对象

设置 scope 值是 prototype 时候，不是在加载 spring 配置文件时候创建 对象，在调用 getBean 方法时候创建多实例对象

request：会将每次创建的对象放在request里面（不常用）

session：会将每次创建的对象放在session里面（不常用）

IOC容器-Bean管理（bean生命周期）

1、生命周期

（1）从对象创建到对象销毁的过程

2、bean 生命周期

（1）通过构造器创建 bean 实例（无参数构造）

（2）为 bean 的属性设置值和对其他 bean 引用（调用 set 方法）

（3）调用 bean 的初始化的方法（需要进行配置初始化的方法）

（4）bean 可以使用了（对象获取到了）

（5）当容器关闭时候，调用 bean 的销毁的方法（需要进行配置销毁的方法）

销毁方法是ClassPathXmlApplicationContext类，Application需要用，要向下转型

 image-20201011214459957

```
public class Orders {
 private String oname;

 public Orders() {
     System.out.println("第一步-->执行无参构造创建bean实例");
 }

 public void setOname(String oname) {
     this.oname = oname;
     System.out.println("第二部-->调用set方法设置属性值");
```

```
    }

    /**
     * 创建执行的初始化方法
     */
    public void initMethod(){
        System.out.println("第三步-->执行初始化的方法");
    }

    /**
     * 创建执行的销毁方法
     */
    public void destroyMethod(){
        System.out.println("最后-->执行销毁方法");
    }
}

//测试方法
    @Test
    public void testBean4(){
        //1、加载spring配置文件；参数为xml所在路径及名称
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean4.xml");
        //2、获取配置文件的对象；参数为配置文件bean标签的id名称
        Orders orders = context.getBean("orders", Orders.class);
        System.out.println("第四步-->获取创建bean实例对象");
        System.out.println(orders);

        //手动销毁bean实例
        ((ClassPathXmlApplicationContext) context).close();
    }
```

```
<bean id="orders" class="com.fsir.spring5.collectionType.beans.Orders" init-
method="initMethod" destroy-method="destroyMethod">
        <property name="oname" value="手机"></property>
    </bean>
```

4、bean 的后置处理器，bean 生命周期有七步

（1）通过构造器创建 bean 实例（无参数构造）

（2）为 bean 的属性设置值和对其他 bean 引用（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 的初始化的方法（需要进行配置初始化的方法）

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

（6）bean 可以使用了（对象获取到了）

（7）当容器关闭时候，调用 bean 的销毁的方法（需要进行配置销毁的方法）

添加后置处理器效果

（1）创建类，实现接口 BeanPostProcessor，创建后置处理器

```
/**
* 后置处理器类
*/
public class MyBeanPost implements BeanPostProcessor {
    @Override
    public Object postProcessBeforeInitialization(Object bean, String
beanName) throws BeansException {
        System.out.println("在初始化前执行的方法");
        return bean;
    }

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

会为文件所有的bean都添加此后置处理器

```
    <!--开启初始化方法和销毁方法-->
    <bean id="orders" class="com.fsir.spring5.collectionType.beans.Orders"
init-method="initMethod" destroy-method="destroyMethod">
        <property name="oname" value="手机"></property>
    </bean>

    <!--配置后之处理器-->
    <bean id="myBeanPost"
class="com.fsir.spring5.collectionType.beans.MyBeanPost"></bean>
```

IOC容器-Bean管理XML方式（自动装配）

1、自动装配

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

```
public class Emp {
 private Dept dept;

 public void setDept(Dept dept) {
     this.dept = dept;
 }

 @Override
 public String toString() {
     return "Emp{" +
             "dept=" + dept +
             '}';
 }

 public void test(){
     System.out.println("dept:\t" + dept);
```

```
 }
}

public class Dept {
 @Override
 public String toString() {
     return "Dept{}";
 }
}

 @Test
 public void testByBean5(){
     //1、加载spring配置文件；参数为xml所在路径及名称
     ApplicationContext context = new
ClassPathXmlApplicationContext("bean5.xml");
     //2、获取配置文件的对象；参数为配置文件bean标签的id名称
     Emp emp = context.getBean("emp", Emp.class);
     System.out.println(emp);

 }
```

```
<!--实现自动装配
     bean标签属性autowire，配置自动装配
     autowire属性常用两个值
         byName：根据属性名称注入，注入值bean的id值和类属性名称一样
         byType：根据属性类型注入，同一类型，不同id的bean会报错，会出现不知道使用哪个的
情况
 -->
 <bean id="emp" class="com.fsir.spring5.collectionType.autowire.Emp"
autowire="byName">
     <!--手动装配；指定某个属性，注入值-->
     <!--<property name="dept" ref="dept"></property>-->
 </bean>
 <bean id="dept" class="com.fsir.spring5.collectionType.autowire.Dept">
</bean>
```

IOC容器-Bean管理XML方式（外部属性文件）

1、直接配置数据库信息

（1）配置德鲁伊连接池

（2）引入德鲁伊连接池依赖 jar 包

2、引入外部属性文件配置数据库连接池

（1）创建外部属性文件，properties 格式文件，写数据库信息

```
jdbc.driverClass=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql//localhost:3306/mysql2
jdbc.username=root
jdbc.password=root
```

```
<?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">

 <!--直接配置连接池-->
 <!--<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
     &lt;!&ndash;手动注入值的方式&ndash;&gt;
     <property name="driverClassName" value="com.mysql.cj.jdbc.Driver">
</property>
     <property name="url" value="jdbc:mysql://localhost:3306/mysql2">
</property>
     <property name="username" value="root"></property>
     <property name="password" value="root"></property>
 </bean>-->

 <!--引入外部属性文件-->
 <context:property-placeholder location="classpath:jdbc.properties"/>
 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
     <!--引入外部配置文件方式-->
     <property name="driverClassName" value="${jdbc.driverclass}">
</property>
     <property name="url" value="${jdbc.url}"></property>
     <property name="username" value="${jdbc.username}"></property>
     <property name="password" value="${jdbc.password}"></property>
 </bean>

</beans>
```

4、IOC操作Bean管理(基于注解)

1、什么是注解

（1）注解是代码特殊标记，格式：@注解名称(属性名称=属性值, 属性名称=属性值..)　　（2）使用注解，注解作用在类上面，方法上面，属性上面

（3）使用注解目的：简化 xml 配置

2、Spring 针对 Bean 管理中创建对象提供注解

（1）@Component： 普通注解，可以用它创建对象实例

（2）@Service： 一般用在service/业务逻辑层

（3）@Controller： 一般用在web层

（4）@Repository： 一般用在dao层

上面四个注解功能是一样的，都可以用来创建 bean 实例

1、引入依赖：spring-aop

2、开启组件扫描

3、创建类，在类上面添加创建对象注解

```
@Service(value = "userServiceImpl") //等于<bean id="userServiceImpl"
class="com.fsir.spring5.service.serviceImpl.UserServiceImpl"></bean>
public class UserServiceImpl implements UserService {
 @Override
 public void add(){
     System.out.println("service add......");
 }
}


public class TestSpringDemo1 {
 @Test
 public void test(){
     ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
     UserService userService = context.getBean("userServiceImpl",
UserServiceImpl.class);
     System.out.println(userService);
     userService.add();
 }
}
```

```
<?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">

 <!--开启组件扫描
     component-scan:组件扫描
     base-package：指定扫描哪个包下
     扫描多个包，可以以逗号隔开，也可以直接扫描其共有的父级
 -->
<!--    <context:component-scan base-package="com.fsir.spring5.service,
com.fsir.spring5.dao"></context:component-scan>-->
 <!--spring5包下所有类都会扫描-->
 <context:component-scan base-package="com.fsir.spring5">
</context:component-scan>

</beans>
```

IOC容器-Bean管理注解方式（组件扫描配置）

```
<!--
     use-default-filters:false，不使用默认filter，自己配置filter
     include-filter：设置扫描哪些内容；以下写法表示：扫描在spring5包内，包含了
Controller注解的类
```

```
    user-default-filters="fasle" 时候 不写 includ-filter  就不会扫描包
 -->
 <context:component-scan base-package="com.fsir.spring5" use-default-
filters="false">
    <context:include-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
 </context:component-scan>

 <!--
    以下是扫描包内所有内容
    exclude-filter:扫描不包含规则的类；不扫描包含了Controller注解的类
 -->
 <<context:component-scan base-package="com.fsir.spring5">
    <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
 </context:component-scan>
```

IOC容器-Bean管理注解方式（注入属性@Autowired和Qualifier）

（1）@Autowired：根据属性类型进行自动装配 （等于bean标签的autowire自动装配byType）

第一步 把 service 和 dao 对象创建，在 service 和 dao 类添加创建对象注解

第二步 在 service 注入 dao 对象，在 service 类添加 dao 类型属性，在属性上面使用注解

```
@Service(value = "userServiceImpl") //等于<bean id="userServiceImpl"
class="com.fsir.spring5.service.serviceImpl.UserServiceImpl"></bean>
public class UserServiceImpl implements UserService {
 //定义dao类型属性;不需要添加set方法
 @Autowired      //根据类型进行注入
 private UserDao userDao;

 @Override
 public void add(){
    System.out.println("service add......");
    userDao.add();
 }
}

public interface UserDao {
 public void add();
}

@Repository
public class UserDaoImpl implements UserDao {
 @Override
 public void add() {
    System.out.println("dao add......");
 }
}
```

（2）@Qualifier：根据名称进行注入

这个@Qualifier 注解的使用，和上面@Autowired 一起使用

```
@Service(value = "userServiceImpl") //等于<bean id="userServiceImpl"
class="com.fsir.spring5.service.serviceImpl.UserServiceImpl"></bean>
public class UserServiceImpl implements UserService {
 //定义dao类型属性;不需要添加set方法
 @Autowired      //根据类型进行注入
 @Qualifier(value = "userDaoImpl")      //指向dao层具体某个实现类（因为接口的实现
类可以有多个，所以有时候需要指定）
 private UserDao userDao;

 @Override
 public void add(){
     System.out.println("service add......");
     userDao.add();
 }
}

@Repository
//可以带value参数,当指定的名称和默认时的名称不一样，就要在调用处使用@Qualifier指定名称，
否则报错:
//No qualifying bean of type 'com.fsir.spring5.dao.UserDao' available
//@org.springframework.beans.factory.annotation.Qualifier(value="userDaoImpl
1")}
// 默认的value参数是类名首字母小写
public class UserDaoImpl implements UserDao {
 @Override
 public void add() {
     System.out.println("dao add......");
 }
}
```

（3）@Resource：可以根据类型注入，可以根据名称注入（spring官方不建议，因为属于java的注解）

```
@Service(value = "userServiceImpl") //等于<bean id="userServiceImpl"
class="com.fsir.spring5.service.serviceImpl.UserServiceImpl"></bean>
public class UserServiceImpl implements UserService {
 //定义dao类型属性;不需要添加set方法
 /*@Autowired      //根据类型进行注入
 @Qualifier(value = "userDaoImpl")      //指向dao层具体某个实现类（因为接口的实现
类可以有多个，所以有时候需要指定）
 private UserDao userDao;*/

 //@Resource       //根据类型进行注入，默认什么都不写和Autowired一样效果
 @Resource(name = "userDaoImpl")    //根据名称进行注入，dao层实现类的注解上的名称，
没有默认类名首字母小写
 private UserDao userDao;

 @Override
 public void add(){
     System.out.println("service add......");
     userDao.add();
 }
}

@Repository
```

```
//可以带value参数,当指定的名称和默认时的名称不一样，就要在调用处使用@Qualifier指定名称，否则报错：
//No qualifying bean of type 'com.fsir.spring5.dao.UserDao' available
//@org.springframework.beans.factory.annotation.Qualifier(value="userDaoImpl1")}
// 默认的value参数是类名首字母小写
public class UserDaoImpl implements UserDao {
 @Override
 public void add() {
     System.out.println("dao add......");
 }
}
```

（4）@Value：注入普通类型属性；以上三个针对 对象类型

```
@Service(value = "userServiceImpl") //等于<bean id="userServiceImpl" class="com.fsir.spring5.service.serviceImpl.UserServiceImpl"></bean>
public class UserServiceImpl implements UserService {
 //定义dao类型属性;不需要添加set方法
 /*@Autowired      //根据类型进行注入
 @Qualifier(value = "userDaoImpl")      //指向dao层具体某个实现类（因为接口的实现类可以有多个，所以有时候需要指定）
 private UserDao userDao;*/

 //@Resource       //根据类型进行注入，默认什么都不写和Autowired一样效果
 @Resource(name = "userDaoImpl")    //根据名称进行注入，dao层实现类的注解上的名称，没有默认类名首字母小写
 private UserDao userDao;

 @Value("abc")
 private String name;

 @Override
 public void add(){
     System.out.println("service add......" + "----->" + name);
     userDao.add();
 }
}
```

IOC容器-Bean管理注解方式（完全注解开发）

1、创建配置类，替代 xml 配置文件

```
@Configuration      //把当前类作为配置类，代替xml配置文件
@ComponentScan(basePackages = {"com.fsir.spring5"}) //开启扫描组件，扫描包路径
public class SpringConfig {

}
```

2、编写测试类的方法

```
@Test
public void testSpringConfig(){
    //加载配置类：new了一个不同的加载类：AnnotationConfigApplicationContext，配置类名称.class
    ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfig.class);
    UserService userService = context.getBean("userServiceImpl", UserServiceImpl.class);
    System.out.println(userService);
    userService.add();
}
```

AOP

概念：在不修改已经完成的代码情况下增加新的功能

> （1）面向切面编程（方面），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。
>
> （2）通俗描述：不通过修改源代码方式，在主干功能里面添加新功能
>
> （3）例子说明 AOP
>
> 

AOP（底层原理）

> 1、AOP 底层使用动态代理
>
> （1）有两种情况动态代理
>
> 第一种 有接口情况，使用 JDK 动态代理
>
> 创建接口实现类代理对象，增强类的方法
>
> 第二种 没有接口情况，使用 CGLIB 动态代理
>
> 创建子类的代理对象，增强类的方法
>
> 

使用 JDK 动态代理

```
java.lang.reflect.Proxy
static Object newProxyInstance(ClassLoader loader, 类<?>[] interfaces, InvocationHandler h)返回指定接口的代理类的实例，该接口将方法调用分派给指定的调用处理程序。
```

> 方法有三个参数：
>
> 第一参数，类加载器

第二参数，增强方法所在的类，这个类实现的接口，支持多个接口

第三参数，实现这个接口 InvocationHandler，创建代理对象，写增强的部分

代码编写

（1）创建接口，定义方法

```
public interface UserDao {
 public int add(int a, int b);

 public String update(String id);
}
```

（2）创建接口实现类，实现方法

```
public class UserDaoImpl implements UserDao {
 @Override
 public int add(int a, int b) {
     return a + b;
 }

 @Override
 public String update(String id) {
     return id;
 }
}
```

（3）使用 Proxy 类创建接口代理对象

```
public class Jdkproxy {
 public static void main(String[] args) {
     //创建接口实现类的代理对象
     //需要实现的接口
     Class[] interfaces = {UserDao.class};
     //所需要的具体对象
     UserDaoImpl userDao = new UserDaoImpl();

     //                          本类加载器                              要实现的接口
接口，可以单独实现或写匿名内部类
//         UserDao dao = (UserDao)
Proxy.newProxyInstance(Jdkproxy.class.getClassLoader(), interfaces, new
UserDaoProxy(userDao));
     UserDao dao = (UserDao)
Proxy.newProxyInstance(Jdkproxy.class.getClassLoader(), interfaces, new
InvocationHandler(){
         private Object obj;

         public InvocationHandler accept(Object obj) {
             this.obj = obj;
             return this;
         }

         @Override
```

```
        public Object invoke(Object o, Method method, Object[] objects)
throws Throwable {
            //执行方法前
            System.out.println("方法执行前执行......" + "----->" +
method.getName() + "----->" + Arrays.toString(objects));

            //被增强的的方法执行;      参数：传递过来的对象，传递过来的参数
            Object res = method.invoke(obj, objects);

            //执行方法后
            System.out.println("方法执行后执行" + "----->" + obj);

            return res;
        }
    }.accept(userDao));

    int add = dao.add(1, 2);
    System.out.println("add方法返回值:\t" + add);
 }
}


//创建代理对象代码
class UserDaoProxy implements InvocationHandler {

 //1、 把创建的是谁的代理对象（具体的实现类），把谁传递过来，例如增强add方法，需要把add所
在的实现类的对象传递过来
 //有参数构造传递
 private Object obj; //传过来的具体实现类对象
 public UserDaoProxy(Object obj){
     this.obj = obj;
 }

 /**
     * 需要增强的逻辑
     * @param o          代理对象
     * @param method     当前的方法
     * @param objects    传递的参数
     * @return
     * @throws Throwable
     */
    @Override
    public Object invoke(Object o, Method method, Object[] objects) throws
Throwable {
        //执行方法前
        System.out.println("方法执行前执行......" + "----->" + method.getName()
+ "----->" + Arrays.toString(objects));

        //被增强的的方法执行;      参数：传递过来的对象，传递过来的参数
        Object res = method.invoke(obj, objects);

        //执行方法后
        System.out.println("方法执行后执行" + "----->" + obj);

        return res;
    }
}
```

AOP术语

1、连接点

类中哪些方法可以被增强，这些方法称为连接点

2、切入点

实际被增强的方法，称为切入点

3、通知

1、实际增强的逻辑部分称为通知（增强）

2、通知的多种类型

前置通知

后置通知

环绕通知

异常通知

最终通知

4、切面

把通知应用到切入点过程

使用 CGLIB 动态代理

Aop准备

1、Spring 框架一般都是基于 AspectJ 实现 AOP 操作

（1）AspectJ 不是 Spring 组成部分，独立 AOP 框架，一般把 AspectJ 和 Spirng 框架一起使用，进行 AOP 操作

2、基于 AspectJ 实现 AOP 操作

（1）基于 xml 配置文件实现

（2）基于注解方式实现（使用）

3、在项目工程里面引入 AOP 相关依赖

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aop</artifactId>
        <version>${org.springframework.version}</version>
</dependency>
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>${org.springframework.version}</version>
</dependency>
```

4、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： execution([权限修饰符] [返回类型] [类全路径] 方法名称)

举例 1：对 com.atguigu.dao.BookDao 类里面的 **add 进行增强**

execution(* com.atguigu.dao.BookDao.add(..))

举例 2：对 com.atguigu.dao.BookDao 类里面的**所有的方法进行增强**

execution(* com.atguigu.dao.BookDao.* (..))

举例 3：对 com.atguigu.dao **包里面所有类，类里面所有方法进行增强**

execution(* com.atguigu.dao.. (..))

AOP 操作（AspectJ 注解）

1、创建类，在类里面定义方法

```
public class User {
 public void add(){
     System.out.println("add......");
 }
}
```

2、创建增强类（编写增强逻辑）

（1）在增强类里面，创建方法，让不同方法代表不同通知类型

```
public class UserProxy {
 /**
     * 前置通知
     */
    public void before(){
        System.out.println("before......");
    }
}
```

3、进行通知的配置

（1）在 spring 配置文件中，开启注解扫描

```
<?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"
       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">

    <!--开启注解扫描-->
    <context:component-scan base-package="com.fsir.spring5.Aop.aopanno">
</context:component-scan>

</beans>
```

（2）使用注解创建 User 和 UserProxy 对象

（3）在增强类上面添加注解 @Aspect

```
@Component
public class User {
    public void add(){
        System.out.println("add......");
    }
}


@Component
@Aspect
public class UserProxy {
    /**
     * 前置通知
     */
    public void before(){
        System.out.println("before......");
    }
}
```

（4）在 spring 配置文件中开启生成代理对象

```
<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">

    <!--开启注解扫描-->
```

```
    <context:component-scan base-package="com.fsir.spring5.Aop.aopanno">
</context:component-scan>

    <!--开启Aspect生成代理对象
        扫描包下，带Aspect注解的类，生成一个代理对象
    -->
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>

</beans>
```

4、配置不同类型的通知

（1）在增强类的里面，在作为通知方法上面添加通知类型注解，使用切入点表达式配置

```
@Component
@Aspect
public class UserProxy {
    /**
     * 前置通知
     * @Before 注解表示作为前置通知
     */
    @Before("execution(* com.fsir.spring5.Aop.aopanno.User.add(..))")
    public void before(){
        System.out.println("before......");
    }

    /**
     * 后置通知（返回通知）
     */
    @AfterReturning("execution(*
com.fsir.spring5.Aop.aopanno.User.add(..))")
    public void afterReturning(){
        System.out.println("afterReturning......");
    }

    /**
     * 异常通知
     *
     * 发生异常，环绕后和最终通知不会执行
     */
    @AfterThrowing("execution(* com.fsir.spring5.Aop.aopanno.User.add(..))")
    public void afterThrowing(){
        System.out.println("afterThrowing......");
    }

    /**
     * 最终通知
     */
    @After("execution(* com.fsir.spring5.Aop.aopanno.User.add(..))")
    public void after(){
        System.out.println("after......");
    }

    /**
     * 环绕通知
     */
```

```
    @Around("execution(* com.fsir.spring5.Aop.aopanno.User.add(..))")
    public void around(ProceedingJoinPoint proceedingJoinPoint) throws
Throwable{
        System.out.println("around前......");

        //被增强的方法执行
        proceedingJoinPoint.proceed();

        System.out.println("around后......");
    }

}

//测试类
public class TestAop {
    @Test
    public void testAopAnno(){
        //1、加载spring配置文件；参数为xml所在路径及名称
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        //2、获取配置文件的对象；参数为配置文件bean标签的id名称
        User user = context.getBean("user", User.class);
        user.add();
    }

}
```

5、相同的切入点抽取：execution的值都是相同的

```
@Component
@Aspect
public class UserProxy {

    /**
     * 相同切入点抽取；value值是其他注解相同的值
     */
    @Pointcut("execution(* com.fsir.spring5.Aop.aopanno.User.add(..))")
    public void pointDemo(){

    }

    /**
     * 前置通知
     * @Before 注解表示作为前置通知
     */
    @Before("pointDemo()")
    public void before(){
        System.out.println("before......");
    }

    /**
     * 后置通知（返回通知）
     */
    @AfterReturning("pointDemo()")
    public void afterReturning(){
        System.out.println("afterReturning......");
    }
```

```
    /**
     * 异常通知
     *
     * 发生异常，环绕后和最终通知不会执行
     */
    @AfterThrowing("pointDemo()")
    public void afterThrowing(){
        System.out.println("afterThrowing.....");
    }

    /**
     * 最终通知
     */
    @After("pointDemo()")
    public void after(){
        System.out.println("after.....");
    }

    /**
     * 环绕通知
     */
    @Around("pointDemo()")
    public void around(ProceedingJoinPoint proceedingJoinPoint) throws
Throwable{
        System.out.println("around前.....");

        //被增强的方法执行
        proceedingJoinPoint.proceed();

        System.out.println("around后.....");
    }

}
```

6、有**多个增强类多同一个方法进行增强**，设置增强类优先级

（1）在增强类上面添加注解 @Order(数字类型值)，数字类型值越小优先级越高

```
@Component      //创建对象实例
@Aspect         //进行切面，生成代理对象
@Order(1)       //增强类优先级，数字越小，等级越高
public class PersonProxy {
    /**
     * 前置通知
     * @Before 注解表示作为前置通知
     */
    @Before("execution(* com.fsir.spring5.Aop.aopanno.User.add(..))")
    public void before(){
        System.out.println("person before.....");
    }
}
```

7、**完全使用注解开发**

（1）创建配置类，不需要创建 xml 配置文件

```
@Configuration
@ComponentScan(basePackages = {"com.fsir.spring5.Aop"})
@EnableAspectJAutoProxy(proxyTargetClass = true)
public class ConfigAop {
}
```

AOP 操作（AspectJ 配置文件）

1、创建两个类，增强类和被增强类，创建方法

```
public class Book {
 public void buy(){
     System.out.println("buy......");
 }
}


public class BookProxy {
 public void before(){
     System.out.println("before......");
 }
}
```

2、在 spring 配置文件中创建两个类对象

```
<!--创建对象-->
 <bean id="book" class="com.fsir.spring5.Aop.aopxml.Book"></bean>
 <bean id="bookProxy" class="com.fsir.spring5.Aop.aopxml.BookProxy"></bean>
```

3、在 spring 配置文件中配置切入点

```
<?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">

 <!--创建对象-->
 <bean id="book" class="com.fsir.spring5.Aop.aopxml.Book"></bean>
 <bean id="bookProxy" class="com.fsir.spring5.Aop.aopxml.BookProxy"></bean>

 <!--配置aop增强-->
 <aop:config>
     <!--配置切入点-->
```

```
    <aop:pointcut id="p" expression="execution(*
com.fsir.spring5.Aop.aopxml.Book.buy(..))"/>

    <!--配置切面-->
    <aop:aspect ref="bookProxy">
        <!--配置具体作用的方法-->
        <aop:before method="before" pointcut-ref="p" />
    </aop:aspect>
 </aop:config>

</beans>
```

测试类

```
public class TestAop {
@Test
 public void testAopXml(){
    //1、加载spring配置文件；参数为xml所在路径及名称
    ApplicationContext context = new
ClassPathXmlApplicationContext("bean2.xml");
    //2、获取配置文件的对象；参数为配置文件bean标签的id名称
    Book book = context.getBean("book", Book.class);
    book.buy();
 }

}
```

jdbcTemplate

概念

1、什么是 JdbcTemplate

（1）Spring 框架对 JDBC 进行封装，使用 JdbcTemplate 方便实现对数据库操作

2、引入相关 jar 包

（1）在 spring 配置文件配置数据库连接池

```
<!-- 数据库连接池 -->
 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
       destroy-method="close">
    <property name="url" value="jdbc:mysql://springtemplate" />
    <property name="username" value="root" />
    <property name="password" value="root" />
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
 </bean>
```

（2）配置 JdbcTemplate 对象，注入 DataSource

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

（3）创建 service 类，创建 dao 类，在 dao 注入 jdbcTemplate 对象

```
<!--开启组件扫描-->
 <context:component-scan base-package="com.fsir.spring5">
</context:component-scan>
```

```
@Service
public class BookServiceImpl implements BookService {
 //注入dao
 @Autowired
 private BookDao bookDao;

}



@Repository
public class BookDaoImpl implements BookDao {
 //注入jdbcTemplate
 @Autowired
 private JdbcTemplate jdbcTemplate;

}
```

JdbcTemplate 操作数据库（添加）

1、对应数据库创建实体类

```
public class User {

  private Long UserId;

  private String userName;

  private String uStatus;

  public Long getUserId() {
      return UserId;
  }

  public void setUserId(Long userId) {
      UserId = userId;
  }

  public String getUserName() {
      return userName;
```

```
    }

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

    public String getuStatus() {
        return uStatus;
    }

    public void setuStatus(String uStatus) {
        this.uStatus = uStatus;
    }
}
```

2、编写 service 和 dao

（1）在 dao 进行数据库添加操作

（2）调用 JdbcTemplate 对象里面 update 方法实现添加操作

update(String sql, Object... args)

有两个参数

第一个参数：sql 语句

第二个参数：可变参数，设置 sql 语句值

```
//service层
@Service
public class BookServiceImpl implements BookService {
    //注入dao
    @Autowired
    private BookDao bookDao;

    /**
     * 添加方法
     * @param user
     */
    public void addBook(User user) {
        bookDao.add(user);
    }
}


//dao层
@Repository
public class BookDaoImpl implements BookDao {
    //注入jdbcTemplate
    @Autowired
    private JdbcTemplate jdbcTemplate;

    /**
     * 添加方法
     * @param user
     */
    public void add(User user) {
        //创建sql语句
```

```
        String sql = "insert into t_book values(?, ?, ?)";
        //将可变参数，存到数组中
        Object[] args = {user.getUserId(), user.getUserName(),
user.getuStatus()};
        //调用方法;可进行增加，修改，删除操作
        int update = jdbcTemplate.update(sql, args);
        System.out.println("update影响行数:\t" + update);
    }
}
```

3、测试类

```
```

JdbcTemplate 操作数据库（修改和删除）

1、service和dao层

```
@Service
public class BookServiceImpl implements BookService {
 //注入dao
 @Autowired
 private BookDao bookDao;

 /**
     * 添加方法
     * @param user
     */
    @Override
    public void addBook(User user) {
        bookDao.add(user);
    }

    /**
     * 修改方法
     * @param user
     */
    @Override
    public void updateBook(User user) {
        bookDao.updateBook(user);
    }

    /**
     * 删除方法
     * @param id
     */
    @Override
    public void delBook(Long id) {
        bookDao.delBook(id);
    }
}
```

```java
@Repository
public class BookDaoImpl implements BookDao {
    //注入jdbcTemplate
    @Autowired
    private JdbcTemplate jdbcTemplate;

    /**
     * 添加方法
     * @param user
     */
    @Override
    public void add(User user) {
        //创建sql语句
        String sql = "insert into t_book values(?, ?, ?)";
        //将可变参数，存到数组中
        Object[] args = {user.getUserId(), user.getUserName(),
user.getuStatus()};
        //调用方法;可进行增加，修改，删除操作
        int update = jdbcTemplate.update(sql, args);
        System.out.println("update插入影响行数:\t" + update);
    }


    /**
     * 修改方法
     * @param user
     */
    @Override
    public void updateBook(User user) {
        //创建sql语句
        String sql = "update t_book set username = ?, ustatus = ? where
user_id = ?";
        //将可变参数，存到数组中
        Object[] args = {user.getUserName(), user.getuStatus(),
user.getUserId()};
        //调用方法;可进行增加，修改，删除操作
        int update = jdbcTemplate.update(sql, args);
        System.out.println("update修改影响行数:\t" + update);
    }


    /**
     * 删除方法
     * @param id
     */
    @Override
    public void delBook(Long id) {
        //创建sql语句
        String sql = "delete from t_book where user_id = ?";
        //调用方法;可进行增加，修改，删除操作
        int update = jdbcTemplate.update(sql, id);
        System.out.println("update删除影响行数:\t" + update);
    }
}
```

2、测试

```
    @Test
    public void testJdbcTemplateInsertAndDel(){
        //1、加载spring配置文件；参数为xml所在路径及名称
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        //2、获取配置文件的对象；参数为配置文件bean标签的id名称
        BookServiceImpl jdbcTemplate = context.getBean("bookServiceImpl",
BookServiceImpl.class);
        //修改
//        jdbcTemplate.updateBook(new User(1L, "javascript", "abc"));

        //删除
        jdbcTemplate.delBook(1L);
    }
```

JdbcTemplate 操作数据库（查询返回某个值）

1、查询表里面有多少条记录，返回是某个值

2、使用 JdbcTemplate 实现查询返回某个值代码

queryForObject(String sql, Class requiredType)

有两个参数

第一个参数：sql 语句

第二个参数：返回类型 Class

```
 /**
     * 查询表中中条数
     * @return
     */
    @Override
    public Integer findCount() {
        return bookDao.findCount();
    }


    /**
     * 查询表中中条数
     * @return
     */
    @Override
    public Integer findCount() {
        //创建sql语句
        String sql = "select count(*) from t_book";
        //调用查询方法;参数：sql语句，返回类型的类

        return jdbcTemplate.queryForObject(sql, Integer.class);
    }


//测试
```

```
    @Test
    public void testJdbcTemplateQueryCount(){
        //1、加载spring配置文件；参数为xml所在路径及名称
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        //2、获取配置文件的对象；参数为配置文件bean标签的id名称
        BookServiceImpl jdbcTemplate = context.getBean("bookServiceImpl",
BookServiceImpl.class);
        //查询总条数
        Integer count = jdbcTemplate.findCount();
        System.out.println("共:\t" + count + "条记录");

    }
```

JdbcTemplate 操作数据库（查询返回对象）

1、场景：查询图书详情

2、JdbcTemplate 实现查询返回对象

queryForObject(String sql, RowMapper rowMapper, Object... args)

有三个参数

第一个参数：sql 语句

第二个参数：RowMapper 是接口，针对返回不同类型数据，使用这个接口里面实现类完成 数据封装

第三个参数：sql 语句值

```
 /**
     * 查询后的数据，封装到对象总
     * @param id
     * @return
     */
    @Override
    public User findOne(Long id) {
        return bookDao.findOne(id);
    }


    /**
     * 查询后的数据，封装到对象总
     * @param id
     * @return
     */
    @Override
    public User findOne(Long id) {
        //创建sql语句
        String sql = "select * from t_book where user_id = ?";
        //调用查询方法;参数：sql语句，将返回类型的类封装到BeanPropertyRowMapper对
象，                    传递的参数
```

```
        User user = jdbcTemplate.queryForObject(sql, new
BeanPropertyRowMapper<>(User.class), id);
        return user;
    }


//测试
@Test
    public void testJdbcTemplateQueryReturnEntity(){
        //1、加载spring配置文件；参数为xml所在路径及名称
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        //2、获取配置文件的对象；参数为配置文件bean标签的id名称
        BookServiceImpl jdbcTemplate = context.getBean("bookServiceImpl",
BookServiceImpl.class);
        //查询总条数
        User one = jdbcTemplate.findOne(1L);
        System.out.println(one);

    }
```

JdbcTemplate 操作数据库（查询返回集合）

1、场景：查询图书列表分页...

2、调用 JdbcTemplate 方法实现查询返回集合

query(String sql, RowMapper rowMapper, Object... args)

有三个参数

第一个参数：sql 语句

第二个参数：RowMapper 是接口，针对返回不同类型数据，使用这个接口里面实现类完成 数据封装

第三个参数：sql 语句值

```
 /**
     * 查询表中所有的记录,封装在list集合中
     * @return
     */
    @Override
    public List<User> findAll() {
        return bookDao.findAll();
    }


    /**
     * 查询表中所有的记录,封装在list集合中
     * @return
     */
    @Override
```

```
    public List<User> findAll() {
        //创建sql语句
        String sql = "select * from t_book";
        //调用查询方法;参数：sql语句，将返回类型的类封装到BeanPropertyRowMapper对象， 第三个参数没有则不写
        List<User> query = jdbcTemplate.query(sql, new BeanPropertyRowMapper<>(User.class));
        return query;
    }


//测试
    @Test
    public void testJdbcTemplateQueryReturnCollection(){
        //1、加载spring配置文件；参数为xml所在路径及名称
        ApplicationContext context = new ClassPathXmlApplicationContext("bean1.xml");
        //2、获取配置文件的对象；参数为配置文件bean标签的id名称
        BookServiceImpl jdbcTemplate = context.getBean("bookServiceImpl", BookServiceImpl.class);
        //查询总条数
        List<User> all = jdbcTemplate.findAll();
        System.out.println(all);

    }
```

JdbcTemplate 操作数据库（批量操作）

1、批量操作：操作表里面多条记录

2、JdbcTemplate 实现批量添加操作

batchUpdate(String sql, List<Object[]> batchArgs)

有两个参数

第一个参数：sql 语句

第二个参数：List 集合，添加多条记录数据

```
 /**
     * 批量添加
     * @param list
     * @return
     */
    @Override
    public Integer batchAdd(List<Object[]> list) {
        return bookDao.batchInsert(list);
    }

        /**
     * 批量添加
     * @param list
     * @return
     */
```

```
    @Override
    public Integer batchInsert(List<Object[]> list) {
        //创建sql语句
        String sql = "insert into t_book values(?, ?, ?)";
        int[] ints = jdbcTemplate.batchUpdate(sql, list);
        System.out.println(Arrays.toString(ints));
        return ints.length;
    }


//测试
    @Test
    public void testJdbcTemplateBatchInsert(){
        //1、加载spring配置文件；参数为xml所在路径及名称
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        //2、获取配置文件的对象；参数为配置文件bean标签的id名称
        BookServiceImpl jdbcTemplate = context.getBean("bookServiceImpl",
BookServiceImpl.class);

        List<Object[]> list = new ArrayList<>();
        Object[] o1 = {3, "java", "a"};
        Object[] o2 = {4, "c++", "b"};
        Object[] o3 = {5, "Mysql", "c"};
        list.add(o1);
        list.add(o2);
        list.add(o3);
        Integer integer = jdbcTemplate.batchAdd(list);
        System.out.println("共影响:\t" + integer + "行");

    }
```

3、JdbcTemplate 实现批量修改操作

```
    /**
     * 批量修改
     * @param list
     * @return
     */
    @Override
    public Integer batchUpdate(List<Object[]> list) {
        return bookDao.batchUpdate(list);
    }


    /**
     * 批量修改
     * @param list
     * @return
     */
    @Override
    public Integer batchUpdate(List<Object[]> list) {
        //创建sql语句
        String sql = "update t_book set username = ?, ustatus = ? where
user_id = ?";
        int[] ints = jdbcTemplate.batchUpdate(sql, list);
        System.out.println(Arrays.toString(ints));
```

```
        return ints.length;
    }


//测试
    @Test
    public void testJdbcTemplateBatchInsert(){
        //1、加载spring配置文件；参数为xml所在路径及名称
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        //2、获取配置文件的对象；参数为配置文件bean标签的id名称
        BookServiceImpl jdbcTemplate = context.getBean("bookServiceImpl",
BookServiceImpl.class);

        List<Object[]> list = new ArrayList<>();

        //批量修改
        Object[] o1 = {"java", "a--", 3};
        Object[] o2 = {"c++", "b--", 4};
        Object[] o3 = {"Mysql", "c--", 5};
        list.add(o1);
        list.add(o2);
        list.add(o3);
        //批量插入
        Integer integer = jdbcTemplate.batchUpdate(list);
        System.out.println("共影响:\t" + integer + "行");
    }
```

4、JdbcTemplate 实现批量删除操作

```
    /**
     * 批量删除
      * @param list
     * @return
     */
    @Override
    public Integer batchDel(List<Object[]> list) {
        return bookDao.batchDel(list);
    }


    /**
     * 批量删除
     * @param list
     * @return
     */
    @Override
    public Integer batchDel(List<Object[]> list) {
        //创建sql语句
        String sql = "delete from t_book where user_id = ?";
        int[] ints = jdbcTemplate.batchUpdate(sql, list);
        System.out.println(Arrays.toString(ints));
        return ints.length;
    }


//测试
```

```
    @Test
    public void testJdbcTemplateBatchInsert(){
        //1、加载spring配置文件；参数为xml所在路径及名称
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean1.xml");
        //2、获取配置文件的对象；参数为配置文件bean标签的id名称
        BookServiceImpl jdbcTemplate = context.getBean("bookServiceImpl",
BookServiceImpl.class);

        List<Object[]> list = new ArrayList<>();

        //批量删除
        Object[] o1 = {3};
        Object[] o2 = {4};
        Object[] o3 = {5};
        list.add(o1);
        list.add(o2);
        list.add(o3);
        //批量插入
        Integer integer = jdbcTemplate.batchDel(list);
        System.out.println("共影响:\t" + integer + "行");
    }
```

事务管理

事务操作（事务概念）

1、什么事务

（1）事务是数据库操作最基本单元，逻辑上一组操作，要么都成功，如果有一个失败所有操 作都失败

（2）典型场景：银行转账

lucy 转账 100 元 给 mary

lucy 少 100，mary 多 100

2、事务四个特性（ACID）

（1）原子性 ： 要么都成功，一条失败则全部失败

（2）一致性：操作前后总量不发生改变

（3）隔离性：多事务之间不会相互影响

（4）持久性：操作完成，事务提交之后，数据库数据发生改变。

 image-20201013210251372

操作环境

1、创建数据库表，添加记录

 image-20201013210532792

2、创建 service，搭建 dao，完成对象创建和注入关系

（1）service 注入 dao，在 dao 注入 JdbcTemplate，在 JdbcTemplate 注入 DataSource

```
@Service
public class USerService {
 @Autowired      //注入dao
 private UserDao userDao;
}

@Repository
public class UserDaoImpl implements UserDao {
 @Autowired
 private JdbcTemplate jdbcTemplate;
}
```

3、在 dao 创建两个方法：多钱和少钱的方法，在 service 创建方法（转账的方法）

```
@Service
public class USerService {
 @Autowired      //注入dao
 private UserDao userDao;

 /**
     * 转账的方法
     */
    public void accountMoney(){
        //lucy少100
        userDao.reduceMoney();
        //mary多100
        userDao.addMoney();
    }

}
```

4、上面代码，如果正常执行没有问题的，但是如果代码执行过程中出现异常，有问题

 image-20201013213305165

 image-20201013213315409

（1）上面问题如何解决呢?

使用事务进行解决

（2）事务操作过程

 image-20201013213554488

事务操作（Spring 事务管理介绍）

> 1、事务添加到 JavaEE 三层结构里面 Service 层（业务逻辑层）
>
> 2、在 Spring 进行事务管理操作
>
> （1）有两种方式：编程式事务管理(例如try...catch,不推荐代码过多)和声明式事务管理（使用）
>
> 3、声明式事务管理
>
> （1）基于注解方式（使用）
>
> （2）基于 xml 配置文件方式
>
> 4、在 Spring 进行声明式事务管理，底层使用 AOP 原理
>
> 5、Spring 事务管理 API： PlatformTransactionManager
>
> （1）提供一个接口，代表事务管理器，这个接口针对不同的框架提供不同的实现类
>
>  image-20201013214358260

事务操作（注解声明式事务管理）

> 1、在 spring 配置文件配置事务管理器

```
<?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">

 <!--开启组件扫描-->
 <context:component-scan base-package="com.fsir.spring5">
</context:component-scan>

 <!-- 数据库连接池 -->
 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
       destroy-method="close">
     <property name="url" value="jdbc:mysql://localhost/springtemplate?
useUnicode=true&amp;characterEncoding=UTF-8&amp;serverTimezone=UTC" />
     <property name="username" value="root" />
     <property name="password" value="root" />
     <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
 </bean>

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

 <!--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>

</beans>
```

2、在 spring 配置文件，开启事务注解

（1）在 spring 配置文件引入名称空间 tx

（2）开启事务注解

```
<?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">

 <!--开启组件扫描-->
 <context:component-scan base-package="com.fsir.spring5">
</context:component-scan>

 <!-- 数据库连接池 -->
 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
       destroy-method="close">
    <property name="url" value="jdbc:mysql://localhost/springtemplate?
useUnicode=true&amp;characterEncoding=UTF-8&amp;serverTimezone=UTC" />
    <property name="username" value="root" />
    <property name="password" value="root" />
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
 </bean>

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

 <!--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>

 <!--开启事务主键-->
 <tx:annotation-driven transaction-manager="transactionManager">
</tx:annotation-driven>
```

```
</beans>
```

3、在 service 类上面（或者 service 类里面方法上面）添加事务注解

（1）**@Transactional，这个注解添加到类上面，也可以添加方法上面**

（2）如果把这个注解添加类上面，这个类里面所有的方法都添加事务

（3）如果把这个注解添加方法上面，为这个方法添加事务

事务操作（声明式事务管理参数配置）

1、在 service 类上面添加注解@Transactional，在这个注解里面可以配置事务相关参数

 image-20201013215529263

2、propagation：事务传播行为

 image-20201013220038164

 image-20201013220038164

（1）**多事务方法**(对数据库表数据进行变化的操作)直接进行调用，这个过程中事务 是如何进行管理的

3、ioslation：事务隔离级别

（1）事务有特性称为隔离性，多事务操作之间不会产生影响。不考虑隔离性产生很多问题
（2）有三个读问题：脏读、不可重复读、虚（幻）读

（3）脏读：一个未提交事务读取到另一个未提交事务的数据

 image-20201013220952884

（4）不可重复读：一个未提交事务读取到另一提交事务修改数据

 image-20201013221322474

（5）虚读：一个未提交事务读取到另一提交事务添加数据

（6）解决：通过设置事务隔离级别，解决读问题

```
//默认使用propagation = Propagation.REQUIRED，mysql默认可重复读
@Transactional(propagation = Propagation.REQUIRED, isolation =
Isolation.REPEATABLE_READ)
```

 image-20201013221513581

4、timeout：超时时间

（1）事务需要在一定时间内进行提交，如果不提交进行回滚

（2）默认值是 -1 ，设置时间以秒单位进行计算

5、readOnly：是否只读

（1）读：查询操作，写：添加修改删除操作

（2）readOnly **默认值 false**，表示可以查询，可以添加修改删除操作

（3）**设置 readOnly 值是 true，设置成 true 之后，只能查询**

6、rollbackFor：回滚

（1）设置出现哪些异常进行事务回滚

7、noRollbackFor：不回滚

（1）设置出现哪些异常不进行事务回滚

事务操作（XML 声明式事务管理）

1、在 spring 配置文件中进行配置

第一步 配置事务管理器

第二步 配置通知

第三步 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: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">

 <!--开启组件扫描-->
 <context:component-scan base-package="com.fsir.spring5">
</context:component-scan>

 <!-- 数据库连接池 -->
 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
       destroy-method="close">
     <property name="url" value="jdbc:mysql://localhost/springtemplate?
useUnicode=true&amp;characterEncoding=UTF-8&amp;serverTimezone=UTC" />
     <property name="username" value="root" />
     <property name="password" value="root" />
     <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
 </bean>

 <!--JdbcTemplate-->
 <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
     <!--注入datasource-->
     <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">
        <tx:attributes>
            <!--指定某种规则的方法上添加事务/可写指定方法名-->
            <tx:method name="accountMoney" propagation="REQUIRED"/><!--指定某一
个方法-->
            <!--<tx:method name="account*"/>--><!--指定以account方法开头的所有方法-
->
        </tx:attributes>
    </tx:advice>

    <!--3、配置切入点和切面-->
    <aop:config>
        <!--配置切入点，表示service中所有的方法都切入-->
        <aop:pointcut id="pt" expression="execution(*
com.fsir.spring5.service.UserService.*(..))"/>
        <!--配置切面，         指定某个通知-->           <!--切入点名称-->
        <aop:advisor advice-ref="txadvice" pointcut-ref="pt"></aop:advisor>
    </aop:config>

</beans>
```

事务操作（完全注解声明式事务管理）

1、创建配置类，使用配置类替代 xml配置文件

```
package com.fsir.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;

import javax.sql.DataSource;

/**
 * @author fsir
 * @Company com.fsir.spring5.config
 * @date 2020-10-13  22:33
 */
@Configuration                                          //将此类作为配置类
@ComponentScan(basePackages = "com.fsir.spring5")           //包扫描路径
@EnableTransactionManagement                            //开启事务注解
public class TxConfig {
    /**
     * 创建数据库连接池
```

```
     * @return
     */
    @Bean
    public DruidDataSource getDruidDataSource(){
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost/springtemplate?
useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC");
        dataSource.setUsername("root");
        dataSource.setPassword("root");
        return dataSource;
    }

    /**
     * 创建JdbcTemplate对象
     * @param dataSource
     * @return
     */
    @Bean
    public JdbcTemplate getJdbcTemplate(DataSource dataSource){
        JdbcTemplate jdbcTemplate = new JdbcTemplate();
        //参数为数据库连接池方法名；将参数作用在方法上，就不要多次调用多次创建了
        jdbcTemplate.setDataSource(dataSource);
        return jdbcTemplate;
    }

    /**
     * 创建事务管理器
     * @param dataSource
     * @return
     */
    @Bean
    public DataSourceTransactionManager
getDataSourceTransactionManager(DataSource dataSource){
        DataSourceTransactionManager transactionManager = new
DataSourceTransactionManager();
        transactionManager.setDataSource(dataSource);
        return transactionManager;
    }

}




//测试
/**
     * 完全注解的方式，开启事务管理---一定记得service也要加注解
     */
    @Test
    public void testAccountByTxConfig(){
        ApplicationContext context = new
AnnotationConfigApplicationContext(TxConfig.class);
        UserService bean = context.getBean("userService",
UserService.class);
        bean.accountMoney();

    }
```

Spring5新特性

1、整个 Spring5 框架的代码基于 Java8，运行时兼容 JDK9，许多不建议使用的类和方 法在代码库中删除

2、Spring 5.0 框架自带了通用的日志封装

（1）**Spring5 已经移除 Log4jConfigListener，官方建议使用 Log4j2 ，否则降级只5版本之下**

（2）Spring5 框架整合 Log4j2

第一步 引入 jar 包

第二步 创建 log4j2.xml 配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!--日志级别以及优先级排序: OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE > 
ALL -->
<!--Configuration后面的status用于设置log4j2自身内部的信息输出，可以不设置，当设置成
trace时，可以看到log4j2内部各种详细输出-->
<configuration status="INFO">
<!--先定义所有的appender-->
<appenders>
  <!--输出日志信息到控制台-->
  <console name="Console" target="SYSTEM_OUT">
      <!--控制日志输出的格式-->
      <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level 
%logger{36} - %msg%n"/>
  </console>
</appenders>
<!--然后定义logger，只有定义了logger并引入的appender，appender才会生效-->
<!--root：用于指定项目的根日志，如果没有单独指定Logger，则会使用root作为默认的日志输出-
->
<loggers>
  <root level="info">
      <appender-ref ref="Console"/>
  </root>
</loggers>
</configuration>
```

3、Spring5 框架核心容器支持@Nullable 注解

（1）@Nullable 注解可以使用在方法上面，属性上面，参数上面，表示方法返回可以为空，属性值可以 为空，参数值可以为空

（2）注解用在方法上面，方法返回值可以为空

（3）注解使用在方法参数里面，方法参数可以为空

（4）注解使用在属性上面，属性值可以为空

4、Spring5 核心容器支持函数式风格 GenericApplicationContext

```
 @Test
 public void testGenericApplicationContext(){
     //1、创建GenericApplicationContext对象
     GenericApplicationContext context = new GenericApplicationContext();
     //2、调用context的方法注册对象
     context.refresh();
     //3、可以先指定创建的对象在spring里面的名称，在lambda里面创建的对象，完成注册
     context.registerBean("user", User.class, () -> new User());
     //4、获取在spring注册的对象;在User对象没有注册到spring的情况，里面放类的全路径
//         User user = (User)
context.getBean("com.fsir.spring5.entity.User");
     User user = (User) context.getBean("user");//指定beanname名称后，可以使用
名称

     System.out.println(user);

 }
```

5、Spring5 支持整合 JUnit5

（1）整合 JUnit4

第一步 引入 Spring 相关针对测试依赖

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-test -->
     <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-test</artifactId>
         <version>${org.springframework.version}</version>
         <scope>test</scope>
     </dependency>
```

第二步 创建测试类，使用注解方式完成

```
import com.fsir.spring5.service.UserService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;


@RunWith(SpringJUnit4ClassRunner.class)      //使用junit4进行测试
@ContextConfiguration(locations = {"classpath:bean1.xml"})//加载配置文件
public class TestJunit4 {
 @Autowired
 private UserService userService;

 @Test
 public void test1(){
     userService.accountMoney();
 }

}
```

（2）Spring5 整合 JUnit5

第一步 引入 JUnit5 的 jar 包

```
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.7.0</version>
    </dependency>
```

第二步 创建测试类，使用注解完成

（3）使用一个复合注解替代上面两个注解完成整合

```
import com.fsir.spring5.service.UserService;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.test.context.junit.jupiter.SpringJUnitConfig;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;


/*@ExtendWith(SpringExtension.class)          //spring5的junit注解
@ContextConfiguration("classpath:bean1.xml")//加载配置文件*/

@SpringJUnitConfig(locations = {"classpath:bean1.xml"})     //可取代上面两个配置
public class TestJunit5 {
 @Autowired
 private UserService userService;

 @Test
 public void test1(){
     userService.accountMoney();
 }

}
```

spring5框架新功能（Webflux）

1、SpringWebflux 介绍

（1）是 Spring5 添加新的模块，用于 web 开发的，功能和 SpringMVC 类似的，Webflux 使用当前一种比较流程响应式编程出现的框架

（2）使用传统 web 框架，比如 SpringMVC，这些基于 Servlet 容器，Webflux 是一种异步非阻塞的框架，异步非阻塞的框架在 Servlet3.1 以后才支持，核心是基于 Reactor 的相关 API 实现的。

（3）解释什么是异步非阻塞

异步和同步

非阻塞和阻塞

上面都是针对对象不一样

异步和同步针对调用者，调用者发送请求，如果等着对方回应之后才去做其他事情就是同 步，如果发送请求之后不等着对方回应就去做其他事情就是异步

阻塞(做完在反馈)和非阻塞(反馈后再做)针对被调用者，被调用者收到请求之后，做完请求任务之后才给出反馈就是阻 塞，受到请求之后马上给出反馈然后再去做事情就是非阻塞

（4）Webflux 特点：

第一 非阻塞式：在有限资源下，提高系统吞吐量和伸缩性，以 Reactor(java9及之后的版本) 为基础实现响应式编程 ——>Flow取代了Observable，Observer(这两个只能算是伪 编程)

第二 函数式编程：Spring5 框架基于 java8，Webflux 使用 Java8 函数式编程方式实现路由请求

（5）比较 SpringMVC

第一 两个框架都可以使用注解方式，都运行在 Tomcat 等容器中

第二 SpringMVC 采用命令式编程，Webflux 采用异步响应式编程

2、响应式编程（Java 实现）

（1）什么是响应式编程

响应式编程是一种面向数据流和变化传播的编程范式。这意味着可以在编程语言中很方便 地表达静态或动态的数据流，而相关的计算模型会自动将变化的值通过数据流进行传播。

电子表格程序就是响应式编程的一个例子。单元格可以包含字面值或类似"=B1+C1"的公 式，而包含公式的单元格的值会依据其他单元格的值的变化而变化。

（2）Java8 及其之前版本

提供的观察者模式两个类 Observer 和 Observable

```
import java.util.Observable;
import java.util.concurrent.Flow;

public class ObserverDemo extends Observable {
 public static void main(String[] args) {
     ObserverDemo observerDemo = new ObserverDemo();
     //添加观察者
     observerDemo.addObserver((o, arg) -> System.out.println("发生了变化"));

     observerDemo.addObserver((o, arg) -> System.out.println("手动被观察者通
知，准备改变"));

     observerDemo.setChanged();//进行监控,数据变化
     observerDemo.notifyObservers();//进行通知
 }
}

//java9及之后的版本实现响应式编程
class Main{
 public static void main(String[] args) {
     //被订阅后，触发                                订阅
     Flow.Publisher<String> publisher = subscriber -> {
```

```
            subscriber.onNext("1");
            subscriber.onNext("2");
            subscriber.onError(new RuntimeException("出错了"));
            subscriber.onComplete();
        };

        publisher.subscribe(new Flow.Subscriber<>() {

            @Override
            public void onSubscribe(Flow.Subscription subscription) {//取消订阅
                subscription.cancel();
            }

            @Override
            public void onNext(String s) {//收到的数据
                System.out.println(s);
            }

            @Override
            public void onError(Throwable throwable) {//异常
                System.out.println("出错了");
            }

            @Override
            public void onComplete() {//完成
                System.out.println("publish complete");
            }
        });

    }
}
```

3、响应式编程（Reactor 实现）

（1）响应式编程操作中，Reactor 是满足 Reactive 规范框架

（2）Reactor 有两个核心类，Mono 和 Flux，这两个类实现接口 Publisher，提供丰富操作 符。Flux 对象实现发布者，返回 N 个元素；Mono 实现发布者，返回 0 或者 1 个元素

（3）Flux 和 Mono 都是数据流的发布者，使用 Flux 和 Mono 都可以发出三种数据信号：**元素值，错误信号，完成信号**，错误信号和完成信号都代表终止信号，终止信号用于告诉 订阅者数据流结束了，错误信号终止数据流同时把错误信息传递给订阅者

（4）代码演示 Flux 和 Mono

第一步 引入依赖

```
<dependency>
<groupId>io.projectreactor</groupId>
<artifactId>reactor-core</artifactId>
<version>3.1.5.RELEASE</version>
</dependency>
```

第二步 编程代码

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class TestRector {
 public static void main(String[] args) {
     //just方法直接声明
     Flux.just(1, 2, 3, 4);//Flux发送n个元素(可变参数)
     Mono.just(1);//发送一个元素，方法一个元素

     //其他方法
     Integer[] array = {1, 2, 3, 4};
     Flux.fromArray(array);

     List<Integer> list = Arrays.asList(array);
     Flux.fromIterable(list);

     Stream<Integer> stream = list.stream();//将list集合转换为stream流
     Flux.fromStream(stream);
 }
}
```

（5）三种信号特点

错误信号和完成信号都是终止信号，不能共存的

如果没有发送任何元素值，而是直接发送错误或者完成信号，表示是空数据流

如果没有错误信号，没有完成信号，表示是无限数据流

（6）调用 just 或者其他方法只是声明数据流，数据流并没有发出，只有进行订阅之后才会触 发数据流，不订阅什么都不会发生的

```
public class TestRector {
 public static void main(String[] args) {
     //just方法直接声明
     Flux.just(1, 2, 3, 4).subscribe(System.out::println);//Flux发送n个元素(可变参数),再进行订阅
     Mono.just(1).subscribe(System.out::println);//发送一个元素，方法一个元素
 }
}
```

（7）操作符

对数据流进行一道道操作，成为操作符，比如工厂流水线

第一 map 元素映射为新元素

第二 flatMap 元素映射为流

把每个元素转换流，把转换之后多个流合并大的流

4、SpringWebflux 执行流程和核心 API

> SpringWebflux 基于 Reactor，默认使用容器是 Netty，Netty 是高性能的 NIO 框架，异步非阻塞的框架
>
> （1）Netty
>
> BIO
>
> NIO
>
> （2）SpringWebflux 执行过程和 SpringMVC 相似的
>
> SpringWebflux 核心控制器 DispatchHandler，实现接口 WebHandler
>
> 接口 WebHandler 有一个方法
>
> （3）SpringWebflux 里面 DispatcherHandler，负责请求的处理
>
> HandlerMapping：请求查询到处理的方法
>
> HandlerAdapter：真正负责请求处理
>
> HandlerResultHandler：响应结果处理
>
> （4）SpringWebflux 实现函数式编程，两个接口：RouterFunction（路由处理） 和 HandlerFunction（处理函数）

5、SpringWebflux（基于注解编程模型）

SpringWebflux 实现方式有两种：注解编程模型和函数式编程模型

使用注解编程模型方式，和之前 SpringMVC 使用相似的，只需要把相关依赖配置到项目中，SpringBoot 自动配置相关运行容器，默认情况下使用 Netty 服务器

第一步 创建 SpringBoot 工程，引入 Webflux 依赖:spring5-webflux-demo8项目

第二步 配置启动端口号

```
<dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-webflux</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>


server.port=8081
```

第三步 创建包和相关类

实体类

```
public class User {
    private String name;
    private String gender;
    private Integer age;

    public User() {
    }

    public User(String name, String gender, Integer age) {
        this.name = name;
        this.gender = gender;
        this.age = age;
    }

    public String getName() {
        return name;
    }

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

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public Integer getAge() {
        return age;
    }

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

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

创建接口定义操作的方法

接口实现类

```
public interface UserService {
```

```
    /**
     * 根据id查询用户
     * @param id
     * @return
     */
    Mono<User> getUserById(int id);

    /**
     * 查询所有用户
     * @return
     */
    Flux<User> getAllUser();

    /**
     * 添加用户
     * @param userMono
     * @return
     */
    Mono<Void> saveUSerInfo(Mono<User> userMono);
}


public class UserServiceImpl implements UserService {
    //创建map集合存储数据
    private final Map<Integer, User> users = new HashMap<>();

    public UserServiceImpl() {
        this.users.put(1, new User("lucy", "男", 20));
        this.users.put(2, new User("mary", "女", 30));
        this.users.put(3, new User("jack", "女", 50));
    }

    /**
     * 根据id查询用户
     * @param id
     * @return
     */
    @Override
    public Mono<User> getUserById(int id) {
        return Mono.justOrEmpty(this.users.get(id));
    }

    /**
     * 查询所有用户,查询多个
     * @return
     */
    @Override
    public Flux<User> getAllUser() {
        return Flux.fromIterable(this.users.values());
    }

    /**
     * 添加用户
     * @param userMono
     * @return
     */
    @Override
    public Mono<Void> saveUSerInfo(Mono<User> userMono) {
```

```
        return userMono.doOnNext(person -> {
            //给map集合存值
            int id = users.size() + 1;
            users.put(id, person);
        }).then(Mono.empty());
    }
}
```

创建 controller

```
@RestController
public class UserConteroller {
    //注入service
    @Autowired
    private UserService userService;

    /**
     * 根据id查询用户
     * @param id
     * @return
     */
    @GetMapping("/user/{id}")
    public Mono<User> getUserId(@PathVariable int id){
        return userService.getUserById(id);
    }

    /**
     * 查询所有用户,查询多个
     * @return
     */
    @GetMapping("/user")
    public Flux<User> getUsers(){
        return userService.getAllUser();
    }

    /**
     * 添加用户
     * @return
     */
    @GetMapping("/saveUser")
    public Mono<Void> saveUser(@RequestBody User user){
        Mono<User> userMono = Mono.just(user);
        return userService.saveUSerInfo(userMono);
    }
}
```

说明

SpringMVC 方式实现，同步阻塞的方式，基于 SpringMVC+Servlet+Tomcat

SpringWebflux 方式实现，异步非阻塞 方式，基于 SpringWebflux+Reactor+Netty

6、SpringWebflux（基于函数式编程模型）

（1）在使用函数式编程模型操作时候，需要自己初始化服务器

（2）基于函数式编程模型时候，有两个核心接口：RouterFunction（实现路由功能，请求转发给对应的 handler）和 HandlerFunction（处理请求生成响应的函数）。核心任务定义两个函数式接口的实现并且启动需要的服务器。

（3）SpringWebflux 请求和响应不再是 ServletRequest 和 ServletResponse，而是 ServerRequest 和 ServerResponse

第一步 把注解编程模型工程复制一份，保留 entity 和 service 内容

第二步 创建 Handler（具体实现方法）

第三步 初始化服务器，编写 Router

创建路由的方法

```
//1、创建Router路由
public RouterFunction<ServerResponse> routerFunction(){
    UserServiceImpl userService = new UserServiceImpl();
    UserHandler handler = new UserHandler(userService);

    return RouterFunctions.route(
            GET("/users/{id}").and(accept(APPLICATION_JSON)),
handler::getUserById)
            .andRoute(GET("/user").and(accept(APPLICATION_JSON)),
handler::getUsers);

}
```

创建服务器完成适配

```
//2、创建服务器完成适配
public void createRouterServer(){
    //路由饿hanlder适配
    RouterFunction<ServerResponse> route = routerFunction();
    HttpHandler httpHandler = toHttpHandler(route);
    ReactorHttpHandlerAdapter adapter = new
ReactorHttpHandlerAdapter(httpHandler);


    //创建服务器
    HttpServer httpServer = HttpServer.create();
    httpServer.handle(adapter).bindNow();
}
```

最终调用

```
public static void main(String[] args) throws IOException {
    Server server = new Server();
    server.createRouterServer();
    System.out.println("enter to exit");

    System.in.read();
}
```

（4）使用 WebClient 调用

```
public class Client {
 public static void main(String[] args) {
     //调用服务器地址
     WebClient webClient = WebClient.create("http://localhost:4131");

     //根据id查询
     String id = "1";
     User userResult = webClient.get().uri("/users/{id}",
id).accept(MediaType.APPLICATION_JSON).retrieve().bodyToMono(User.class).blo
ck();
     System.out.println(userResult.getName());

     //查询所有
     Flux<User> results =
webClient.get().uri("/user").accept(MediaType.APPLICATION_JSON).retrieve().b
odyToFlux(User.class);
     results.map(stu ->
stu.getName()).buffer().doOnNext(System.out::println).blockFirst();
 }
}
```

最后

1、Spring 框架概述

（1）轻量级开源 JavaEE 框架，为了解决企业复杂性，两个核心组成：IOC 和 AOP

（2）Spring5.2.6 版本

2、IOC 容器

（1）IOC 底层原理（工厂、反射等）

（2）IOC 接口（BeanFactory）

（3）IOC 操作 Bean 管理（基于 xml）

（4）IOC 操作 Bean 管理（基于注解）

3、Aop

（1）AOP 底层原理：动态代理，有接口（JDK 动态代理），没有接口（CGLIB 动态代理）

（2）术语：切入点、增强（通知）、切面

（3）基于 AspectJ 实现 AOP 操作

4、JdbcTemplate

（1）使用 JdbcTemplate 实现数据库 curd 操作

（2）使用 JdbcTemplate 实现数据库批量操作

5、事务管理

（1）事务概念

（2）重要概念（传播行为和隔离级别）

（3）基于注解实现声明式事务管理

（4）完全注解方式实现声明式事务管理

6、Spring5 新功能

（1）整合日志框架

（2）@Nullable 注解

（3）函数式注册对象

（4）整合 JUnit5 单元测试框架

（5）SpringWebflux 使用