

Spring概述

 image-20210724233150742

IOC容器

IOC是什么

1. 控制反转，把对象创建和对象之间的调用过程，都交给Spring进行管理
2. 使用IOC的目的：为了耦合度降低

IOC底层原理

主要是用到了xml解析,工厂模式,反射 的技术

IOC实现的过程

 image-20210726224932975

例子：

```
public void testAdd(){  
    //1. 加载spring配置文件  
    ApplicationContext context =new ClassPathXmlApplicationContext("beans.xml");  
  
    //2. 获取xml中配置的对象,"user"是beans.xml中的id,  
    User user=context.getBean("user", User.class);  
  
    System.out.println(user);  
    user.add ();  
}
```


IOC(接口)

 image-20210802223041054

IOC操作-Bean管理

1. 什么是Bean管理
Bean管理的指的是两个操作：（1）Spring创建对象 （2）Spring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

IOC操作Bean管理(基于xml方式)

1. 基于xml方式创建对象  image-20210802224228985
2. 基于xml方式注入属性
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setter方法
    public void setName(String bname){
        this.bname=bname;
    }

    public void setAuthor(String bauthor){
        this.bauthor=bauthor;
    }
}
```

(2) 在Spring配置文件中配置对象创建，配置属性注入

```
<!--set方法注入属性-->
<bean id="book" class="com.atguigu.spring5.Book">
    <!--使用property完成属性注入
        name: 类里面属性的名称
        value: 向属性注入的值
    -->
    <property name="bname" value="易筋经"></property>
    <property name="bauthor" value="达摩老祖"></property>
</bean>
```

(3) 按照spring的xml创建对象的方式创建对象，并调用相应的方法即可。

第二种方式：使用有参构造器的方法进行注入

(1) 创建类，定义属性，创建带有属性的有参构造方法

```
public class Orders{
    //定义属性
    private String oname;
    private String address;

    //有参的构造方法
    public Orders(String oname, String address){
        this.oname=oname;
        this.address=address;
    }
}
```

(2) 在Spring的配置文件进行配置

```
<!--有参数的构造方法注入-->
<bean id="orders" class="com.atguigu.spring5.Orders">
    <constructor-arg name="oname" value="computer"></constructor-arg>
    <constructor-arg name="address" value="china"></constructor-arg>
</bean>
```

(3)使用Spring的xml创建对象的方式创建对象，并调用相应的方法即可

IOC操作Bean管理(基于xml方式，注入其他类型属性)

1. 字面量

1. null值[在xml中使用null标签注入]

```
<bean id="book" class="com.atguigu.spring5.Book">
    <!--使用property完成属性注入
        name: 类里面属性的名称
        value: 向属性注入的值
    -->
    <property name="bname" value="易筋经"></property>
    <property name="bauthor" value="达摩老祖"></property>
    <!--新的属性是address，为address赋值为null-->
    <!--去除掉property标签中的value属性，并添加null标签-->
    <property name="address" >
        <!--添加null标签，就赋值null值了-->
        <null/>
    </property>
</bean>
```

2. 属性值包含特殊符号

```
<!--属性值包含特殊符号
1.把<>进行转义&lt;&gt;;
2.把带特殊符号内容写到CDATA中
-->
<property address="address">
    <value><![CDATA[<<南京>>]]></value>
</property>
```

2. 注入属性--外部Bean【一般使用外部Bean注入】

1. 创建两个service类和dao类
2. 在service调用dao类的方法
3. 在Spring配置文件中配置

```
<bean id="userService" class="com.atguigu.spring5.service.UserService" >
    <!--注入userDao对象
        name属性：类里面属性的名称
        ref属性：创建userDao对象bean标签的id值
    -->
    <property name="userDao" ref="userDaoImpl"></property>
</bean>
<bean id="userDaoImpl" class="com.atguigu.spring5.dao.UserDaoImpl" >
</bean>
```

3. 注入属性--内部Bean和级联赋值

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】

1. 创建部门和员工的实体类，并在员工的类中注入dept实体类
2. 在Spring配置文件中配置

```

<!--内部bean的方式-->
<bean id="emp" class="com.atguigu.spring5.bean.Emp">
    <!--设置两个普通属性-->
    <property name="ename" value="Lucy"></property>
    <property name="gender" value="female"></property>
    <!--设置对象类型属性-->
    <property name="dept">
        <bean id="dept" class="com.atguigu.spring5.bean.Dept">
            <property name="dname" value="security"></property>
        </bean>
    </property>
</bean>

```

2. 注入属性--级联赋值

1. 第一种方式【类似外部注入的方式】

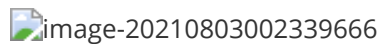
```

<!--级联赋值-->
<bean id="emp" class="com.atguigu.spring5.bean.Emp">
    <!--设置两个普通属性-->
    <property name="ename" value="Lucy"></property>
    <property name="gender" value="female"></property>
    <!--级联赋值-->
    <property name="dept" ref="dept"></property>
</bean>
<bean id="dept" class="com.atguigu.spring5.bean.Dept">
    <property name="dname" value="finance"></property>
</bean>

```

2. 第二种方式[添加这个属性<property name="dept.dname" value="Technology">]

1. 在Emp的类中添加dept的get方法



2. 在Spring配置文件中设置级联赋值

```

<!--级联赋值-->
<bean id="emp" class="com.atguigu.spring5.bean.Emp">
    <!--设置两个普通属性-->
    <property name="ename" value="Lucy"></property>
    <property name="gender" value="female"></property>
    <!--级联赋值-->
    <property name="dept" ref="dept"></property>

    <property name="dept.dname" value="Technology"></property>
</bean>
<bean id="dept" class="com.atguigu.spring5.bean.Dept">
    <property name="dname" value="finance"></property>
</bean>

```

IOC操作Bean管理（xml注入集合属性）

1. 注入数组，List，Map，Set类型的属性,并生成setter方法

```

public class Stu{
    //1. 数组类型的属性
    private String[] courses;
    //2.List集合类型的属性
    private List<String> list;
    //3.Map集合类型的属性
    private Map<String,String> maps;
    //4.Set集合类型的属性
    private Set<String> sets;
}

```

2. 在Spring的配置文件中配置属性

```

<!--各种属性类型的注入-->
<bean id="stu" class="com.atguigu.spring5.collectiontype.stu">
    <!--数组类型属性的注入-->
    <property name="courses">
        <array>
            <value>java课程</value>
            <value>数据库课程</value>
        </array>
    </property>

    <!--List类型属性的注入-->
    <property name="list">
        <list>
            <value>Tom</value>
            <value>Tony</value>
        </list>
    </property>

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

    <!--Set类型属性的注入-->
    <property name="sets">
        <set>
            <value>MySQL</value>
            <value>Redis</value>
        </set>
    </property>
</bean>

```

3. 在集合里面设置对象类型值

1. 创建Course类，设置cname属性并设置setter方法

```
public class Course{
    private String cname;
    public void setName(String cname){
        this.cname=cname;
    }
}
```

2. 在Stu中注入Course类

```
public class Stu{
    ...
    //注入Course对象的list
    private List<Course> courseList;

    public void setCourseList(List<Course> courseList){
        this.courseList=courseList
    }
}
```

3. 在Spring配置文件配置Course对象的List

```
<!--注入List集合类型，值是对象-->
<property name="courseList">
    <list>
        <ref bean="course1"></ref>
        <ref bean="course2"></ref>
    </list>
</property>


<!--创建多个course对象-->
<bean id="course1" class="com.atguigu.spring5.collectiontype.Course">
    <property name="cname" value="Spring5 framework"></property>
</bean>
<bean id="course2" class="com.atguigu.spring5.collectiontype.Course">
    <property name="cname" value="MyBatis framework"></property>
</bean>
```

4. 把集合注入部分提取出来

1. 在Spring配置文件中引入名称空间util

 image-20210803011150274

2. 使用util标签

 image-20210803011447202

IOC操作Bean管理 (FactoryBean)

1. Spring中有两种bean：一种是普通Bean，一种是工厂Bean(FactoryBean)

2. 普通Bean：在配置文件中定义的Bean类型就是返回的类型

3. 工厂Bean：在配置文件中定义的Bean类型可以和返回类型不一样

1. 创建类，让这个类作为工厂Bean，实现接口FactoryBean

2. 实现接口里面的方法，在实现的方法中定义返回的Bean类型

3. 例子：

(1) 实现FactoryBean接口

```
//实现FactoryBean接口
public class MyBean implements FactoryBean<Course> {

    //定义返回的Bean
    @Override
    public Course getObject() throws Exception {
        Course course =new Course();
        course.setName("abc");
        return course;
    }

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

    @Override
    public boolean isSingleton() {
        return FactoryBean.super.isSingleton();
    }
}
```

(2) 配置xml文件

```
<bean id="myBean" class="com.atguigu.spring5.factorybean.MyBean">
</bean>
```

(3) 测试用例方法

```
import com.atguigu.spring5.User;
import com.atguigu.spring5.collectiontype.Course;
import com.atguigu.spring5.factorybean.MyBean;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestSpring5 {

    @Test
    public void testAdd(){
        //1.加载Spring配置文件
        ApplicationContext context = new
        ClassPathXmlApplicationContext("bean1.xml");

        //2.获取Bean对象
        User user=context.getBean("user", User.class);
        System.out.println(user);
        user.add();
    }
}
```

```

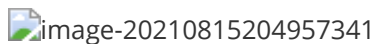
@Test
public void testMybean(){
    ApplicationContext context =new
    ClassPathXmlApplicationContext("bean1.xml");
    Course course =context.getBean("myBean", Course.class);
    System.out.println(course);
}

}

```

IOC操作Bean(Bean的作用域)

1. 在Spring里面，设置创建Bean实例是单例还是多例



在spring中，默认的Bean实例是单例的

2. 如何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) 用户设置单实例还是多实例

scope="singleton" 是单实例，默认值[加载配置文件的时候，就会创建单实例对象]

scope="prototype" 是多实例[在调用getBean的时候，才会创建]

IOC操作Bean管理(Bean的生命周期)

1. Bean生命周期[从开始创建到销毁的过程]

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的使用(对象获取到了)
5. 当容器关闭的时候，调用bean的销毁方法(需要进行配置销毁的方法)

2. 演示Bean生命周期的例子

1. 创建Orders的类

```

package com.atguigu.spring5.Order;

public class Orders {

    private String name;

    public Orders() {
        System.out.println("第一步： 执行无参数构造方法创建Bean实例");
    }

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

    //创建执行初始化的方法
    public void initMethod(){

```



```

        System.out.println("第三步：执行初始化方法");
    }

    //创建执行的销毁的方法
    public void destroyMethod(){
        System.out.println("第五步：执行销毁的方法");
    }
}

```

2. 配置xml文件[要配置init-method和destroy-method 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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="orders" class="com.atguigu.spring5.Order.Orders" init-
method="initMethod" destroy-method="destroyMethod">
        <property name="name" value="abc"></property>
    </bean>
</beans>

```

3. 执行测试方法


```

@Test
public void testOrders(){
    ClassPathXmlApplicationContext context =new
ClassPathXmlApplicationContext("bean1.xml");
    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);

    //默认去调用Orders类中的destroyMethod方法去销毁
    context.close();
}

```

4. 测试用例执行结果

 image-20210815213626715

3. 添加Bean的后置处理器【生命周期发生改变，增加了两步】

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实例传递给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实例传递给Bean后置处理器的方法【postProcessAfterInitialization】**
6. Bean的使用(对象获取到了)
7. 当容器关闭的时候，调用bean的销毁方法(需要进行配置销毁的方法)

4. 添加后置处理器的效果例子

1. 创建BeanPostProcessor的实现类

```

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

```

```

public class MyBeanPost implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String
beanName) throws BeansException {
        System.out.println("后置处理器第一步：在初始化之前执行的方法");
        return
BeanPostProcessor.super.postProcessBeforeInitialization(bean, beanName);
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String
beanName) throws BeansException {
        System.out.println("后置处理器第二步：在初始化之后执行的方法");
        return
BeanPostProcessor.super.postProcessAfterInitialization(bean, beanName);
    }
}

```

2. 在配置文件中配置MyBeanPost类

```

<!--配置后置处理器-->
<bean id="myBeanPost" class="com.atguigu.spring5.Order.MyBeanPost">
</bean>

```

3. 调用同样的测试方法


```

@Test
public void testOrders(){
    ClassPathXmlApplicationContext context =new
ClassPathXmlApplicationContext("bean1.xml");
    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);

    //默认去调用Orders类中的destroyMethod方法去销毁
    context.close();
}

```

4. 测试用例的测试结果

image-20210815215849977

IOC操作Bean管理(xml自动装配 不常用)

1. 什么是自动装配

根据指定装配规则(属性名称或者属性类型), spring自动讲匹配的属性值进行注入

2. 在配置Bean的时候, 设置autowire属性值【一个是byName, 一个是byType】

3. 自动装配的例子

1. 创建两个类 Emp 和Dept, 并且Emp中注入Dept

```

public class Emp {
    private Dept dept;

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

    @Override
    public String toString() {
        return "Dept: "+dept;
    }
}

```

2. 在配置文件设置autowire属性[byType只能有一个Dept bean,有多个会报错]

```

<!--    byName自动注入    -->
<bean id="emp" class="com.atguigu.spring5.autoconfig.Emp"
autowire="byName"></bean>

<!--    byType自动注入    -->
<bean id="emp" class="com.atguigu.spring5.autoconfig.Emp"
autowire="byType"></bean>

<bean id="dept" class="com.atguigu.spring5.autoconfig.Dept"></bean>

```

IOC操作Bean管理(外部属性文件)

1. 直接配置数据库连接池

```

<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver">
</property>
    <property name="url" value="jdbc:mysql://localhost:3306/userDb">
</property>
    <property name="username" value="root"></property>
    <property name="password" value="123456"></property>
</bean>

```

2. 引入外部属性文件配置数据库的连接池

1. 创建外部的数据库连接池信息文件

 image-20210831083609300

2. 把外部properties属性文件引入到spring配置文件中

在spring的xml头部添加context命令空间

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

xsi:schemaLocation 中添加"<http://www.springframework.org/schema/context> <http://www.springframework.org/schema/context/spring-context.xsd>"

3. 在spring配置文件中添加[context:property-placeholder](#)

```

<context:property-placeholder location="classpath:jdbc.properties"/>

```

IOC操作Bean管理(基于注解方式)

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. 使用注解的目的：简化xml配置

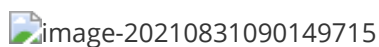
2. Spring针对Bean管理中创建对象提供的注解

1. @Component:所有的类上都能
2. @Service:一般用在service类上
3. @Controller: 一般用在controller类上
4. @Repository: 一般用在DAO类上

注意：上面四个注解功能是一样的，都可以用来创建bean实例。只是为了分层，约定了以上的规则。

3. 基于注解方式创建对象

1. 引入Spring AOP 的jar包



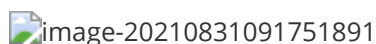
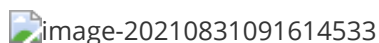
2. 开启组件扫描(xml头部添加context schema信息)

```
<!--开启组件扫描
1. 如果扫描多个包，多个包之间使用逗号隔开
2. 扫描包的上层目录
-->
<context:component-scan base-
package="com.atguigu.spring5.dao,com.atguigu.spring5.service">
</context:component-scan>
```

3. 创建类，在类上面添加创建对象的注解

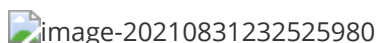
在注解里面value属性值可以省略不写，默认值是类名称，首字母小写。

4. 注解扫描的细节注意事项

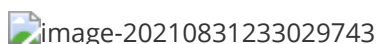


基于注解方式实现属性注入

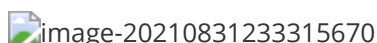
1. @AutoWired: 根据属性类型进行自动装配



2. @Qualifier: 根据属性名称进行注入【要和@AutoWired 一起使用】



3. @Resource: 可以根据类型注入，可以根据名称进行注入【这个是javax.annotation.Resource 类】



4. @Value: 注入普通类型属性




完全注解开发

1. 创建配置类，替代xml配置文件



2. 编写测试类

 image-20210831234426969

AOP部分


AOP概念

1. 什么是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**。

2. 通俗描述: 不通过修改源代码的方式, 在主干功能里面添加新功能

3. 使用登陆的例子说明

 image-20210831235930133

AOP底层原理

AOP底层使用了动态代理[有两种情况的动态代理]

1. 有接口的情况, 使用JDK动态代理

创建接口实现类代理对象, 增强类的方法

 image-20210901000532953

2. 没有接口的情况, 使用CGLIB动态代理

创建子类的代理对象, 增强类的方法

 image-20210901000904928

AOP(JDK动态代理实现)

1. 使用的是java.lang.reflect.proxy类, 利用它的静态方法newProxyInstance(ClassLoader loader,class<?>[] interfaces, InvocationHandler h)

方法中有三个参数:

第一个参数: 类加载器

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

第三个参数: 实现这个接口InvocationHandler, 创建代理对象, 书写增强的方法

2. 编写JDK动态代理实现代码

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,

```

public class UserDaoImp 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

```

package com.atguigu.spring5.aopimp;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.Arrays;

/**
 * @author zrhsmile
 * @create 2021-09-01-21:23
 * @Description:实现jdk动态代理
 */
public class JDKProxy {
    public static void main(String[] args) {

        Class[] interfaces= {UserDao.class};

        //创建接口实现类代理对象
        UserDaoImp userDaoImp=new UserDaoImp();
        UserDao dao =
        (UserDao)Proxy.newProxyInstance(JDKProxy.class.getClassLoader(), interfaces, new UserDaoProxy(userDaoImp));
        int result= dao.add(1,2 );
        System.out.println("result: "+result);
    }
}

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

    //1.把创建的是谁的代理对象，把谁传递过来
    //有构造传递
    private Object obj;
    public UserDaoProxy(Object obj){
        this.obj=obj;
    }

    //增强逻辑
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {

```

```

        //方法之前
        System.out.println("方法执行之前..." + method.getName() + " 传递的参数..." + Arrays.toString(args));

        //被增强的方法执行
        Object res=method.invoke(obj,args);

        //方法之后
        System.out.println("方法执行之后..." + obj);

        return res;
    }
}

```

AOP(术语)

1. 连接点

类里面哪些方法可以被增强，这些方法称为连接点

2. 切入点

实际被真正增强的方法，称为切入点

3. 通知(增强)

实际增强的逻辑部分称为通知(增强)

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

4. 切面

是一个动作：把通知应用到切入点的过程

AOP操作(准备)

1. Spring框架一般都是基于AspectJ实现AOP操作

1. 什么是AspectJ

AspectJ不是Spring的组成部分，独立AOP框架，一般把AspectJ和Spring框架一起使用，进行AOP操作

2. 基于AspectJ实现AOP操作

基于xml配置文件实现

基于注解方式实现(使用)

3. 在项目工程里面引入AOP的相关依赖

 image-20210901224349092

4. 切入点表达式

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

语法结构：

execution(【权限修饰符】 【返回类型】 【类全路径】 【方法名称】 (【参数列表】))

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

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

例子2：对com.atguigu.dao.BookDao类里面的所有方法进行增强

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

例子3：对com.atguigu.dao包里面的所有类，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

```
package com.atguigu.spring5.aopano;

/**
 * @author zrhsmile
 * @create 2021-09-01-22:55
 * @Description:
 */
public class User {

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

2. 创建增强类(编写增强逻辑)

在增强类里面，创建方法，让不同方法代表不同的通知类型

```
package com.atguigu.spring5.aopano;

/**
 * @author zrhsmile
 * @create 2021-09-01-23:01
 * @Description:创建User的增强类
 */
public class UserProxy {
    //前置通知
    public void before(){
        System.out.println("before.....");
    }
}
```

3. 进行通知的配置

- 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.atguigu.spring5.aopano">
</context:component-scan>
</beans>
```

- 使用注解创建User和UserProxy对象


 image-20210901235833259

 image-20210901235937853

- 在增强类上面添加注解@Aspect

 image-20210902000038119

- 在Spring配置文件中开启生成代理对象

```
<!--开启Aspect生成代理对象-->
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

4. 配置不同类型的通知

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

```
package com.atguigu.spring5.aopano;

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

/**
 * @author zrhsmile
 * @create 2021-09-01-23:01
 * @Description: 创建User的增强类
 */
@Component
@Aspect //生成代理对象
public class UserProxy {
    //前置通知
    //@Before注解表示作为前置通知
    @Before(value="execution(* com.atguigu.spring5.aopano.User.add(..)")")
    public void before(){
        System.out.println("before.....");
    }

    //最终通知
    @After(value="execution(* com.atguigu.spring5.aopano.User.add(..)")")
```

```

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

    //后置通知（返回通知）
    @AfterReturning(value="execution(*
com.atguigu.spring5.aopano.User.add(..)")
    public void afterReturning(){
        System.out.println("afterReturning.....");
    }

    //异常通知
    @AfterThrowing(value="execution(*
com.atguigu.spring5.aopano.User.add(..)")
    public void afterThrowing(){
        System.out.println("afterThrowing.....");
    }

    //环绕通知
    @Around(value="execution(* com.atguigu.spring5.aopano.User.add(..)")
    public void around(ProceedingJoinPoint proceedingJoinPoint) throws
    Throwable {

        System.out.println("环绕之前.....");

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

        System.out.println("环绕之后.....");
    }
}

```

@After与@AfterReturning之间的区别：@After是方法执行之后执行，@AfterReturning是方法返回结果之后执行。同时存在的时候，@AfterReturning是在@After执行之后执行。

@After不论方法是否有异常，都会执行。@AfterReturning如果方法出现异常不会执行

5. 相同的切入点提取

```

//相同的切入点抽取
@Pointcut(value="execution(* com.atguigu.spring5.aopano.User.add(..)")
    public void pointDemo(){

    }

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

```

6. 有多个增强类对同一个方法进行增强，设置增强类优先级

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

```
/**
```

```

* @author zrhsmile
* @create 2021-09-07-23:38
* @Description:设置Order(数字类型值)确定优先级
*/
@Component
@Aspect
@Order(1)
public class PersonProxy {

    @Before(value="execution(* com.atguigu.spring5.aopanno.User.add(..))")
    public void afterReturning(){
        System.out.println("Person Before.....");
    }
}

```

```

@Component
@Aspect //生成代理对象
@Order(3)
public class UserProxy {

    //相同的切入点抽取
    @Pointcut(value="execution(* com.atguigu.spring5.aopano.User.add(..))")
    public void pointDemo(){

    }
}

```

注解：@Order(1)先执行，@Order(3)后执行。因此，PersonProxy类先执行，UserProxy后执行

7. 完全使用注解开发

创建配置类，不需要创建xml配置文件

```

package com.atguigu.spring5.config;

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

/**
 * @author zrhsmile
 * @create 2021-09-09-15:23
 * @Description:
 */
@Configuration
@ComponentScan(basePackages = {"com.atguigu"})
@EnableAspectJAutoProxy(proxyTargetClass = true)
public class ConfigAop {
}

```

AOP操作(AspectJ配置文件)[只是了解]

1. 创建两个类，增强类与被增强类，创建方法

```
public class Book {

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

}
```

```
/**
 * @author zrhsmile
 * @create 2021-09-07-23:53
 * @Description:Book的增强类
 */
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bean对象-->
<bean id="book" class="com.atguigu.spring5.aopxml.Book"></bean>
<bean id="bookProxy" class="com.atguigu.spring5.aopxml.BookProxy"></bean>
```

3. 在spring配置文件中配置切入点

```
<!--配置aop config-->
<aop:config>
    <!--切入点设置-->
    <aop:pointcut id="proxy" expression="execution(*
com.atguigu.spring5.aopxml.Book.buy())"/>
    <!--配置切面-->
    <aop:aspect>
        <!--增强作用在具体的方法上-->
        <aop:before method="before" pointcut-ref="proxy"></aop:before>
    </aop:aspect>
</aop:config>
```

Data Access/Integration部分


JdbcTemplate(概念和准备)

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操作。

2. 准备工作

1. 引入相应的jar包

image-20210910145429657

2. 在spring配置文件中配置数据库连接池

```

<!--配置数据源信息-->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
destroy-method="close">
    <property name="url" value="jdbc:mysql:///user_db" />
    <property name="username" value="root" />
    <property name="password" value="root" />
    <property name="driverClassName" value="com.mysql.jdbc.Driver"
/>
</bean>

```

3. 在spring配置文件中，创建JdbcTemplate 对象，注入DataSource对象

```

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

```

4. 创建dao,service对象，并在spring的配置文件中配置组件扫描
BookDao的接口文件

```

import org.springframework.stereotype.Repository;

/**
 * @author zrhsmile
 * @create 2021-09-10-15:10
 * @Description:
 */
public interface BookDao {
    public void insert();
}

```

BookDaoImpl的实现类文件

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

/**
 * @author zrhsmile
 * @create 2021-09-10-15:10
 * @Description:
 */
@Repository
public class BookDaoImpl implements BookDao {

    //注入JdbcTemplate
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Override
    public void insert() {

```

```
}  
}
```

BookService的类文件

```
import com.atguigu.spring5.dao.BookDao;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
/**  
 * @author zrhsmile  
 * @create 2021-09-10-15:12  
 * @Description:  
 */  
@Service  
public class BookService {  
  
    //注入bookDao  
    @Autowired  
    private BookDao bookDao;  
}
```

在spring文件中配置组件扫描

```
<!--配置组件扫描-->  
    <context:component-scan base-package="com.atguigu">  
</context:component-scan>
```

JdbcTemplate操作数据(添加)

1. 创建数据库表和实体类

```
/**  
 * @author zrhsmile  
 * @create 2021-09-10-16:35  
 * @Description:  
 */  
public class Book {  
    private String bookid;  
    private String bookName;  
    private String bookStatus;  
  
    public String getBookid() {  
        return bookid;  
    }  
  
    public void setBookid(String bookid) {  
        this.bookid = bookid;  
    }  
  
    public String getBookName() {  
        return bookName;  
    }  
  
    public void setBookName(String bookName) {
```

```

        this.bookName = bookName;
    }

    public String getBookStatus() {
        return bookStatus;
    }

    public void setBookStatus(String bookStatus) {
        this.bookStatus = bookStatus;
    }
}

```

2. 编写service和dao

service层

```

/**
 * @author zrhsmile
 * @create 2021-09-10-15:12
 * @Description:
 */
@Service
public class BookService {

    //注入bookDao
    @Autowired
    private BookDao bookDao;

    //添加方法
    public void addBook(Book book){
        bookDao.addBook(book);
    }
}

```

dao层

```

import com.atguigu.spring5.entity.Book;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

/**
 * @author zrhsmile
 * @create 2021-09-10-15:10
 * @Description:
 */
@Repository
public class BookDaoImpl implements BookDao {

    //注入JdbcTemplate
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Override
    public void addBook(Book book) {
        //1. 创建sql语句
        String sql="insert into t_book value(?,?,?)";
    }
}

```

```

        //2.调用JdbcTemplate的update(sql,args)方法实现
        Object[] args={book.getBookid(), book.getBookName(),
        book.getBookStatus()};
        int addCount= jdbcTemplate.update(sql, args);
        System.out.println(addCount);
    }
}

```

3. 编写测试类

```

package com.atguigu.spring5.test;

import com.atguigu.spring5.entity.Book;
import com.atguigu.spring5.service.BookService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * @author zrhsmile
 * @create 2021-09-10-16:46
 * @Description:
 */
public class TestBook {

    public void testJdbcTemplateAddBook(){
        //1.加载spring配置文件，并获取bookService
        ApplicationContext context =new
        ClassPathXmlApplicationContext("bean1.xml");
        BookService bookService = context.getBean("bookService",
        BookService.class);

        //2.创建book对象并设置属性
        Book book= new Book();
        book.setBookid("1");
        book.setBookName("aa");
        book.setBookStatus("bbb");

        //3.调用bookService的addBook进行添加
        bookService.addBook(book);
    }
}

```

JdbcTemplate操作数据库(修改和删除)

1. 编写service类和daoImpl类的实现

service层


```

//修改的方法
public void updateBook(Book book){
    bookDao.updateBook(book);
}

//删除的方法
public void deleteBook(String id){
    bookDao.deleteBook(id);
}

```

daoImpl层

```

import com.atguigu.spring5.entity.Book;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

/**
 * @author zrhsmile
 * @create 2021-09-10-15:10
 * @Description:
 */
@Repository
public class BookDaoImpl implements BookDao {

    //注入JdbcTemplate
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Override
    public void updateBook(Book book) {
        //1.编写sql语句
        String sql="update t_book set book_name=?,book_status=? where book_id=?";

        //2.调用jdbcTemplate的update(sql,args)方法实现更改
        Object[] args=
        {book.getBookName(),book.getBookStatus(),book.getBookid()};
        int update = jdbcTemplate.update(sql, args);
        System.out.println(update);
    }

    @Override
    public void deleteBook(String id) {
        String sql="delete from t_book where book_id=?";
        int deleteCount = jdbcTemplate.update(sql, id);
        System.out.println(deleteCount);
    }
}

```

2. 测试修改和删除方法

```

@Test
public void testJdbcTemplateUpdateBook(){
    ApplicationContext context =new
    ClassPathXmlApplicationContext("bean1.xml");
}

```

```

        BookService bookService = context.getBean("bookService",
BookService.class);
        Book book= new Book();
        book.setBookid("1");
        book.setBookName("dd");
        book.setBookStatus("eee");
        bookService.updateBook(book);
    }

    @Test
    public void testJdbcTemplateDeleteBook(){
        ApplicationContext context =new
ClassPathXmlApplicationContext("bean1.xml");
        BookService bookService = context.getBean("bookService",
BookService.class);
        String id ="1";
        bookService.deleteBook(id);
    }

```

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，返回某个值
2. 使用JdbcTemplate实现查询返回某个值，使用的是JdbcTemplate的queryForObject(String sql,Class requiredType)方法

service层

```

//查询表的数据记录总数的方法
public int findCount(){
    return bookDao.findCount();
}

```

daoImpl层

```

@Override
public int findCount() {
    //1.编写sql语句
    String sql="select count(1) from t_book";
    //2.使用JdbcTemplate.JdbcTemplate的queryForObject(String sql,Class<T>
requiredType)方法
    Integer count = jdbcTemplate.queryForObject(sql, Integer.class);
    return count;
}

```

JdbcTemplate操作数据库(查询返回对象)

1. 场景：查询图书的详情
2. JdbcTemplate实现查询返回对象;使用JdbcTemplate.queryForObject(String sql, RowMapper rowMapper, Object... args) 其中，这个RowMapper是一个接口，返回不同类型的数据，使用这个接口里面实现对数据的封装

service 层

```
//查询之后，返回Book对象
public Book findOne(String id){
    return bookDao.findOne(id);
}
```

Daoimpl层

```
@Override
public Book findOne(String id) {
    //1.编写sql语句
    String sql="select * from t_book where book_id=?";
    //2.使用JdbcTemplate.queryForObject(String sql, RowMapper<T>
    rowMapper, Object... args)
    Book book = jdbcTemplate.queryForObject(sql,new
    BeanPropertyRowMapper<Book>(Book.class),id);
    return book;
}
```

JdbcTemplate操作数据库(查询返回集合)

1. 返回集合，使用JdbcTemplate.query(String sql, RowMapper rowMapper)方法

service层

```
//查询之后，返回Book对象的集合
public List<Book> findAll(){
    return bookDao.findAll();
}
```

daoimpl层

```
@Override
public List<Book> findAll() {
    //1.编写sql语句
    String sql="select * from t_book";
    //2.使用JdbcTemplate.query(String sql, RowMapper<T> rowMapper)方法
    List<Book> bookList = jdbcTemplate.query(sql,new
    BeanPropertyRowMapper<Book>(Book.class));
    return bookList;
}
```

JdbcTemplate操作数据库(批量添加)

1. 批量操作：操作表里面多条记录
2. JdbcTemplate实现批量添加操作，使用的是JdbcTemplate.batchUpdate(String sql, List<Object[]> batchArgs)方法

service层：

```
//批量数据添加
public void batchAdd(List<Object[]> batchArgs){
    bookDao.batchAdd(batchArgs);
}
```

daoImpl层

```
@Override
public void batchAdd(List<Object[]> batchArgs) {
    //1.编写sql语句
    String sql="insert into t_book values(?,?,?)";
    //2.使用JdbcTemplate.batchUpdate(String sql, List<Object[]>
batchArgs)方法
    int[] ints = jdbcTemplate.batchUpdate(sql, batchArgs);
    System.out.println(Arrays.toString(ints));
}
```

测试用例

```
@Test
public void testJdbcTemplateBatchAdd(){
    //加载spring的配置文件，并获取bookService对象
    ApplicationContext context =new
ClassPathXmlApplicationContext("bean1.xml");
    BookService bookService = context.getBean("bookService",
BookService.class);

    //创建需要批量添加的对象数组
    List<Object[]> batchArgs=new ArrayList<>();
    Object[] o1={"3","c++","abc"};
    Object[] o2={"4","java","def"};
    Object[] o3={"5","php","ghi"};
    batchArgs.add(o1);
    batchArgs.add(o2);
    batchArgs.add(o3);

    //调用service的batchAdd方法
    bookService.batchAdd(batchArgs);
}
```

JdbcTemplate操作数据库(批量更新和删除)

1. 还是调用JdbcTemplate的JdbcTemplate.batchUpdate(String sql, List<Object[]> batchArgs)方法，只是sql语句改成需要更新或者删除的sql。
2. 测试用例中，数组改成sql所需要的内容。

Transaction事务部分

事务概念

1. 什么是事务

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。

典型场景：银行转账

2. 事务四个特性(ACID)

- 原子性(**Atomicity**): 指事务是一个不可分割的工作单位, 事务中的操作要么全部成功, 要么全部失败
- 一致性(**Consistency**): 事务必须使数据库从一个一致性状态变换到另外一个一致性状态
- 隔离性(**Isolation**): 多个用户并发访问数据库时, 数据库为每一个用户开启的事务, 不能被其他事务的操作数据所干扰, 多个并发事务之间要相互隔离
- 持久性(**Durability**): 一个事务一旦被提交, 它对数据库中数据的改变就是永久性的, 接下来即使数据库发生故障也不应该对其有任何影响。


事务操作(搭建事务操作环境)

事务操作的步骤

 image-20210912143955975

事务操作(Spring事务管理介绍)

1. 事务一般要添加到JavaEE三层结构里面Service层(业务逻辑层)
2. 有两种方式: 编程式事务管理和声明式事务管理(一般使用这种)
3. 声明式事务管理
 1. **基于注解的方式[一般使用这种]**
 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. 在Spring事务管理的API
 1. 提供了一个接口, 代表事务管理器, 这个接口针对不同的框架提供了不同的实现类
 2. **这个接口是**PlatformTransactionManager, Mybatis和JdbcTemplate是DataSourceTransactionManger实现类

 image-20210912144830861

事务操作(注解声明式事务管理)

1. 在Spring配置文件中配置事务管理器

```
<!--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"/>
</bean>
```

2. 在Spring配置文件中, 开启事务注解
 1. 在Spring配置文件中引入名称空间tx

```
<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">
```

2. 在Spring配置文件中配置事务注解

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

3. 在Service类上面(或者在Service类里面的方法上面)添加事务注解

在类上添加@Transactional注解开启注解，这个类中的所有的方法都开启了事务。

```
package com.atguigu.spring5.service;

import com.atguigu.spring5.dao.UserDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/**
 * @author zrhsmile
 * @create 2021-09-12-15:06
 * @Description:在类上添加@Transactional注解开启事务
 */
@Service
@Transactional
public class UserService {

    @Autowired
    private UserDao userDao;

    /**
     * 转账方法
     */
    public void accountMoney(){
        userDao.accountMoney();
    }
}
```

在方法上添加@Transactional注解开启注解，那么只会在这个方法上面开启了事务。

```
package com.atguigu.spring5.service;
```

```

import com.atguigu.spring5.dao.UserDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/**
 * @author zrhsmile
 * @create 2021-09-12-15:06
 * @Description:在方法上添加@Transactional注解开启事务
 */
@Service

public class UserService {

    @Autowired
    private UserDao userDao;

    /**
     * 转账方法
     */
    @Transactional
    public void accountMoney(){
        userDao.accountMoney();
    }
}

```


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-20210912151740092

1. propagation：事务传播行为

1. 多事务方法直接进行调用，这个过程中事务是如何进行管理的

2.  image-20210912160535342

3.  image-20210912160732125

4. Service类中配置这个事务

```

package com.atguigu.spring5.service;

import com.atguigu.spring5.dao.UserDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

/**
 * @author zrhsmile
 * @create 2021-09-12-15:06
 * @Description:添加@Transactional注解，并设置propagation为REQUIRED
 */
@Service
@Transactional(propagation = Propagation.REQUIRED)
public class UserService {

```

```

@Autowired
private UserDao userDao;

/**
 * 转账方法
 */
public void accountMoney(){
    userDao.accountMoney();
}
}


```

2. isolation:事务隔离级别

1. 事务特有特性为隔离性，多事务操作之间不会产生影响。不考虑隔离性产生很多很多问题

1. 有三个读问题：脏读，不可重复读，幻读

2. **脏读：一个未提交的事务读取到另一个未提交事务的数据**


image-20210912161812069

3. **不可重复读：一个未提交的事务读取到另一个提交事务的修改数据**

image-20210912162054279

4. **幻读：一个未提交的事务读取到另一个提交事务的添加数据**

2. 解决：通过设置事务隔离级别，解决读问题

image-20210912162441402

3. 在Service中设置设置隔离级别isolation = Isolation.REPEATABLE_READ[默认]

```

@Service
@Transactional(propagation = Propagation.REQUIRED,isolation =
Isolation.REPEATABLE_READ)
public class UserService {

    @Autowired
    private UserDao userDao;

    /**
     * 转账方法
     */
    public void accountMoney(){
        userDao.accountMoney();
    }
}

```

3. timeout:超时时间

1. 事务需要在一定时间内提交， 如果不提交进行回滚
2. 默认是-1,设置时间以秒为单位进行计算
3. 在Service类中设置 timeout = 5

```

@Service
@Transactional(timeout = 5, propagation =
Propagation.REQUIRED,isolation = Isolation.REPEATABLE_READ)

```



```
public class UserService {

    @Autowired
    private UserDao userDao;

    /**
     * 转账方法
     */
    public void accountMoney(){
        userDao.accountMoney();
    }

}
```

4. readOnly: 是否只读

1. 读: 查询操作 写: 添加修改删除操作
2. readOnly默认值是false, 表示可以查询, 可以添加修改删除操作
3. 设置readOnly值是true, 则只能进行查询操作

5. rollbackFor: 回滚

1. 设置出现哪些异常进行事务回滚

6. 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

```
<!--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"/>
</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="accountMoney" 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(*
com.atguigu.spring5.service.UserService.*(..))"/>
    <!--配置切面-->
    <aop:advisor advice-ref="txadvice" pointcut-ref="pt"/>
</aop:config>
```

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

```
import org.springframework.transaction.TransactionManager;
import
org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;

/**
 * @author zrhsmile
 * @create 2021-09-12-16:50
 * @Description:
 */
@Configuration //配置类
@ComponentScan(basePackages = "com.atguigu")//组件扫描
@EnableTransactionManagement //开启事务
public class TxConfig {
    //创建数据库连接池
    @Bean
    public DruidDataSource getDruidDataSource(){
        DruidDataSource dataSource =new DruidDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql:///user_db");
        dataSource.setUsername("root");
        dataSource.setPassword("root");
    }


    //创建JdbcTemplate对象，并注入datasource
    @Bean
    public JdbcTemplate getJdbcTemplate(DruidDataSource dataSource){
        //到IOC容器中根据类型找到dataSource
        JdbcTemplate jdbcTemplate=new JdbcTemplate();
        //注入dataSource
        jdbcTemplate.setDataSource(dataSource);
        return jdbcTemplate;
    }

    //创建事务管理器
    @Bean
    public DataSourceTransactionManager
getTransactionManager(DruidDataSource dataSource){
        DataSourceTransactionManager transactionManager =new
DataSourceTransactionManager();
        transactionManager.setDataSource(dataSource);
        return transactionManager;
    }
}
```

Spring5框架新功能

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. Spring5框架自带了通用的日志封装
 1. Spring5已经移除了Log4jConfigListener 官方建议使用Log4j2
 2. Spring5整合Log4j2,

第一步：需要这几个jar包

image-20210912170738398

第二步：创建Log4j2.xml

```
<?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是, 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, 只有定了logger并引入appender, appender才会生效-->
    <!--root:用于指定项目的根日志, 如果没有单独值丁Logger, 则会使用root作为默认的
日志输出-->
    <Loggers>
        <Root level="info">
            <AppenderRef ref="Console"/>
        </Root>
    </Loggers>
</Configuration>
```

3. Spring5框架核心容器支持支持@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, 属性可以为空, 参数可以为空。


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


```
//函数式风格创建对象, 交给Spring进行管理
@Test
public void testGenericApplicationContext(){
    //1. 创建GenericApplicationContext对象
    GenericApplicationContext context = new GenericApplicationContext();
    //2. 调用context的方法进行对象注册
    context.refresh();
    context.registerBean("user1", User.class, ()->new User() );
    //3. 获取在spring注册的对象
    User user=(User)context.getBeanDefinition("user1");
    System.out.println(user);
}
```

5. Spring5支持整合JUnit5

1. 整合JUnit4

1. 引入Spring针对测试的相关依赖

image-20210912180019816

image-20210912180633745

2. 创建测试类，使用注解方式完成

```
package com.atguigu.spring5.test;


import com.atguigu.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;

/**
 * @author zrhsmile
 * @create 2021-09-12-18:03
 * @Description:
 */
@RunWith(SpringJUnit4ClassRunner.class) //单元测试框架
@ContextConfiguration("classpath:bean1.xml") //加载配置文件
public class JTest4 {
    @Autowired
    private UserService userService;

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

2. 整合Junit5

1. 引入Junit5的jar包

 image-20210912210347248

2. 创建测试类， import org.junit.jupiter.api.Test; 【这个是junit5的测试类】

```
package com.atguigu.spring5.test;

import com.atguigu.spring5.service.UserService;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;

/**
 * @author zrhsmile
 * @create 2021-09-12-21:04
 * @Description: 使用@ExtendWith和@ContextConfiguration这两个注解
 */
@ExtendWith(SpringExtension.class)
@ContextConfiguration("classpath:bean1.xml")
public class JTest5 {

    private UserService userService;
```

```

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

```

这两个注解可以用@SpringJUnitConfig(locations = "classpath:bean1.xml") 这个替换掉，这个是复合注解

Spring5框架新功能(Webflux)

1. SpringWebflux介绍

是Spring5添加新的模块，用于web开发的，功能与SpringMVC类似的，Webflux使用当前一种比较流程响应式出现的框架

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Servlet 3.1以后才支持。核心是基于Reactor的相关API实现。

异步非阻塞

- **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
- **阻塞和非阻塞**针对被调用者，被调用收到请求之后，做完请求人后之后才给出反馈就是阻塞；收到请求之后马上给出反馈然后再去做事情就是非阻塞。

Webflux特点：

- 非阻塞式：在有限资源，提高系统吞吐量和伸缩性，以Reactor为基础实现响应式编程
- 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

比较SpringMVC与Webflux

 image-20210912212857980

区别：

- 两个框架都可以使用朱恩杰方式，都运行在Tomcat等容器中
- SpringMVC采用命令式编程，Webflux采用异步响应式编程

2. 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。

- Java8及其之前版本

提供的观察者模式两个类Observer和Observable

```

public class ObserverDemo extends Observable{
    public static void main(String[] args){
        ObserverDemo observer=new ObserverDemo();
        //添加观察者
        observer.addObserver((o,arg)->{
            System.out.println("发生了变化")
        })
        observer.addObserver((o,arg)->{
            System.out.println("收到了请求，数据发生了变化")
        })
        //数据变化
    }
}

```

```
observer.setChanged();
observer.notifyObservers();
    }
}
```

3. 响应式编程(Reactor实现)

- 响应式编程操作中, Reactor是满足Reactive规范框架
- 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
- 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的同时, 也会把错误信息传递给订阅者
- 【内容需要补充】

4. SpringWebFlux执行流程和核心API

SpringWebflux基于Reactor, 默认容器是Netty,Netty是高性能, NIO框架, 异步非阻塞的框架

- BIO模式[阻塞式]


image-20210912221325763

- Netty是基于非阻塞式NIO, 模式如下:

image-20210912221603353

SpringWebflux执行过程和SpringMVC 相似的

- SpringWebflux核心控制器DispatchHandler,实现的接口是WebHandler
- 接口WebHandler有一个方法handle, 内部实现如下

image-20210912222155168

SpringWebflux里面DispatcherHandler, 负责请求的处理

- HandlerMapping: 其你去查询到处理的方法
- HandlerAdapter: 真正负责请求处理
- HandlerResultHandler: 相应结果处理

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实现方式(基于注解编程模型)

【需要补充】

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实现方式(基于函数式编程模型)

在使用函数式编程模型操作的时候, 需要自己初始化服务器

基于函数式编程模型的时候, 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(处理请求生成相应的函数)。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。

SpringWebflux请求和相应不再是ServletRequest和ServletResponse,而是ServerRequest和ServerResponse

【需要补充】

Spring5框架总结

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

IOC容器

1. IOC底层原理(工厂,反射)
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)**

AOP

1. AOP底层原理：动态代理，有接口(JDK动态代理),没有接口(CGLIB代理)
2. 基于ASpect实现AOP操作

JdbcTemplate

1. 使用JdbcTemplate实现数据库CRUD操作
2. 使用JdbcTemplate实现数据库批量操作

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

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