

Spring5

课程内容介绍

- 1、Spring 框架概述
- 2、IOC 容器
 - (1) IOC 底层原理
 - (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
- 4、JdbcTemplate
- 5、事务管理
- 6、Spring5 新特性

Spring5 框架概述

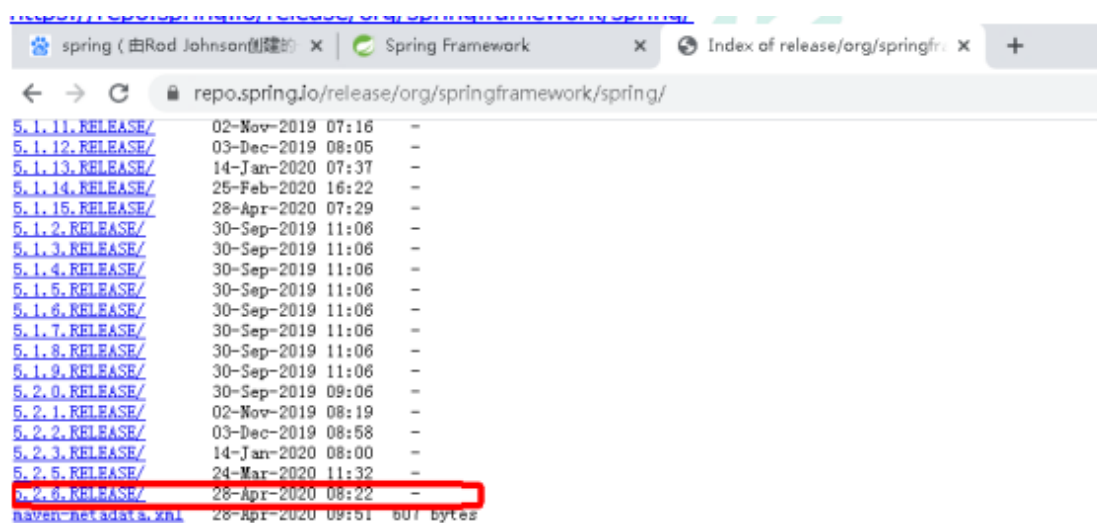
- 1、Spring 是轻量级的开源的 JavaEE 框架
- 2、Spring 可以解决企业应用开发的复杂性
- 3、Spring 有两个核心部分：IOC 和 Aop
 - (1) IOC：控制反转，把创建对象过程交给 Spring 进行管理
 - (2) Aop：面向切面，不修改源代码进行功能增强

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

下载资料

<http://repo.spring.io/release/org/springframework/spring/>

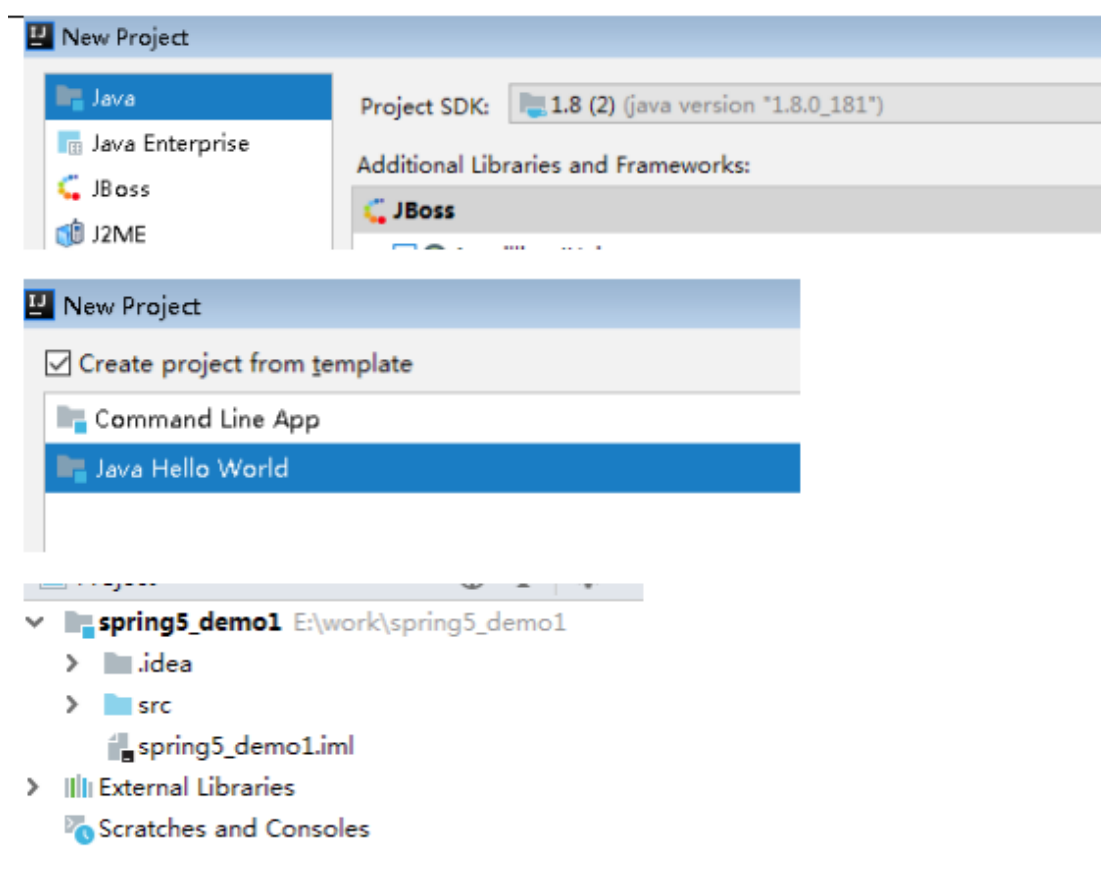


5.1.11.RELEASE/	02-Nov-2019 07:16	-
5.1.12.RELEASE/	03-Dec-2019 08:05	-
5.1.13.RELEASE/	14-Jan-2020 07:37	-
5.1.14.RELEASE/	25-Feb-2020 16:22	-
5.1.15.RELEASE/	28-Apr-2020 07:29	-
5.1.2.RELEASE/	30-Sep-2019 11:06	-
5.1.3.RELEASE/	30-Sep-2019 11:06	-
5.1.4.RELEASE/	30-Sep-2019 11:06	-
5.1.5.RELEASE/	30-Sep-2019 11:06	-
5.1.6.RELEASE/	30-Sep-2019 11:06	-
5.1.7.RELEASE/	30-Sep-2019 11:06	-
5.1.8.RELEASE/	30-Sep-2019 11:06	-
5.1.9.RELEASE/	30-Sep-2019 11:06	-
5.2.0.RELEASE/	30-Sep-2019 09:06	-
5.2.1.RELEASE/	02-Nov-2019 08:19	-
5.2.2.RELEASE/	03-Dec-2019 08:58	-
5.2.3.RELEASE/	14-Jan-2020 08:00	-
5.2.5.RELEASE/	24-Mar-2020 11:32	-
5.2.6.RELEASE/	28-Apr-2020 08:22	-
maven-metadata.xml	28-Apr-2020 09:51	607 bytes

spring-5.2.6.RELEASE-dist > spring-framework-5.2.6.RELEASE >		
帮助(H)		
刻录 新建文件夹		
名称	修改日期	类型
docs	2020/4/28 8:22	文件夹
libs	2020/4/28 8:22	文件夹
schema	2020/4/28 8:22	文件夹
license.txt	2020/4/28 8:14	文本文档
notice.txt	2020/4/28 8:14	文本文档
readme.txt	2020/4/28 8:14	文本文档

基础环境配置

1. 创建工程



2. 添加 4 个 JAR 包



简单使用

1. 创建一个普通类

```
1 package com.pojo;  
2  
3 public class Ordinary1 {  
4  
5     public void ordinaryMethod(){  
6         System.out.println("第一个spring");  
7     }  
8 }
```

2. 创建一个 XML 文件

```
1 <?xml version="1.0" encoding="UTF-8"?>  
2 <beans xmlns="http://www.springframework.org/schema/beans"  
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
4     xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  
5     <!--id:别名 class:类位置-->  
6     <bean id="Ordinary" class="com.pojo.Ordinary1"></bean>  
7 </beans>
```

3. 获取 XML 中创建的类



```
1 package com.test;
2
3 import com.pojo.Ordinary1;
4 import org.junit.Test;
5 import org.springframework.context.ApplicationContext;
6 import org.springframework.context.support.ClassPathXmlApplicationContext;
7
8 public class OrdinaryTest {
9     @Test
10    public void test(){
11        //读取配置文件
12        ApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");
13        //获取Bean并创建类
14        Ordinary1 ordinary = (Ordinary1) context.getBean("Ordinary");
15        //调用类的方法
16        ordinary.ordinaryMethod();
17    }
18 }
```

IOC 容器

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

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

IOC过程

第一步 xml配置文件，配置创建的对象

```
<bean id="dao" class="com.atguigu.UserDao"></bean>
```

进一步降低耦合度

第二步 有service类和dao类，创建工厂类

```
class UserFactory {  
    public static UserDao getDao() {  
        String classValue = class属性值; //1 xml解析  
        Class clazz = Class.forName(classValue); //2 通过反射创建对象  
        return (UserDao)clazz.newInstance();  
    }  
}
```

IOC 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

IOC 操作 Bean 概念

1、什么是 Bean 管理

（0）Bean 管理指的是两个操作

（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)

基于 XML 创建对象

XML

```
<!--XML创建对象 id:别名 class:类路径+类名-->
<bean id="Ordinary" class="com.pojo.Ordinary1"/>
```

JAVA 类

```
public class OrdinaryTest {
    @Test
    public void test(){
        //读取配置文件,并创建类
        ApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");
        //获取类
        Ordinary1 ordinary = (Ordinary1) context.getBean("Ordinary");
        //调用类的方法
        ordinary.ordinaryMethod();
    }
}
```

基于 XML 注入属性

第一种注入方式:通过 set 方法注入

工具类

创建对应的属性以及 set 方法

```
Ordinary1.java x
1  package com.pojo;
2
3  public class Ordinary1 {
4
5      private String ordinaryAttribute;
6
7      public void ordinaryMethod(){
8          System.out.println("第一个spring");
9      }
10
11     public Ordinary1() {
12     }
13
14     public Ordinary1(String ordinaryAttribute) {
15         this.ordinaryAttribute = ordinaryAttribute;
16     }
17     public void setOrdinaryAttribute(String ordinaryAttribute) {
18         this.ordinaryAttribute = ordinaryAttribute;
19     }
20 }
21
```

XML

可以设置多个属性值

```
<!--XML创建对象 id:别名 class:类路径+类名-->
<bean id="Ordinary" class="com.pojo.Ordinary1">
    <!--通过property设置属性 name:属性名 value:属性值-->
    <property name="ordinaryAttribute" value="oneSpring"/>
</bean>
</beans>
```


第二种注入方式:通过构造器注入

工具类

```
5     private String ordinaryAttribute;  
6     private int age;  
7  
8     public void ordinaryMethod() { System.out.println("第一个spring"); }  
11  
12     public Ordinary1() {  
13     }  
14  
15     public Ordinary1(String ordinaryAttribute,int age) {  
16         this.ordinaryAttribute = ordinaryAttribute;  
17         this.age = age;  
18     }  
19     public void setOrdinaryAttribute(String ordinaryAttribute) {  
20         this.ordinaryAttribute = ordinaryAttribute;  
21     }  
22 }  
23  
24     public void setAge(int age) {  
25         this.age = age;  
26     }  
27  
28     public void ordinaryInfo(){  
29         System.out.println("名字"+ordinaryAttribute);  
30         System.out.println("年龄"+age);  
31     }  
32 }
```

XML

```
<bean id="Ordinary" class="com.pojo.Ordinary1">  
    <!--通过constructor-arg使用构造器赋值 name:属性名 value:属性值-->  
    <constructor-arg name="ordinaryAttribute" value="马大厨"/>  
    <constructor-arg name="age" value="13"/>  
</bean>
```

测试类

```
public class OrdinaryTest {  
    @Test  
    public void test(){  
        //读取配置文件,并创建类  
        ApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");  
        //获取类  
        Ordinary1 ordinary = (Ordinary1) context.getBean("Ordinary");  
        //调用类的方法  
        ordinary.ordinaryMethod();  
        ordinary.ordinaryInfo();  
    }  
}
```

注入类属性为其他类型

Null 不写 value,在 constructor-arg 中加入 null 标签

```
<!--注入null值-->  
<constructor-arg name="age">  
    <null/>  
</constructor-arg>
```

特殊符号

```
<!--特殊符号-->  
<property name="ordinaryAttribute">  
    <value><![CDATA[<<图书>>]]></value>  
</property>
```

类之间的创建并调用

外部类

Test1

```
1 package com.test;
2
3 import org.springframework.context.ApplicationContext;
4 import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6 public class Test1 {
7
8     private Test2 test2;
9
10    public void setTest2(Test2 test2) {
11        this.test2 = test2;
12    }
13
14    public void test1Method(){
15        System.out.println("这里是test1");
16        test2.test2Method();
17    }
18
19    public static void main(String[] args) {
20        ApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");
21        Test1 test1 = context.getBean("Test1", Test1.class);
22        test1.test1Method();
23    }
24 }
```

Test2

```
1 package com.test;
2
3 public class Test2 {
4
5
6    public void test2Method(){
7        System.out.println("这里是test2的方法");
8    }
9 }
```

XML

```
<!--创建test1的类-->
<bean id="Test1" class="com.test.Test1">
    <!--通过ref调用XML中的Test2给Test1中的Test2赋值-->
    <property name="test2" ref="Test2"/>
</bean>
<!--创建test2的类-->
<bean id="Test2" class="com.test.Test2"/>
```

内部类

案例:一对多,一个员工只有一个部门

Staff

```
public class Staff {
    private String staffName;
    private int staffAge;
    private Department department;

    public void setDepartment(Department department) {
        this.department = department;
    }

    public void setStaffName(String staffName) {
        this.staffName = staffName;
    }

    public void setStaffAge(int staffAge) {
        this.staffAge = staffAge;
    }

    @Override
    public String toString() {
        return "Staff{" +
            "staffName='" + staffName + '\'' +
            ", staffAge=" + staffAge +
            ", department=" + department +
            '}';
    }
}
```

Department

```
public class Department {
    private String departmentId;
    private String departmentName;

    public void setDepartmentId(String departmentId) {
        this.departmentId = departmentId;
    }

    public void setDepartmentName(String departmentName) {
        this.departmentName = departmentName;
    }

    @Override
    public String toString() {
        return "Department{" +
            "departmentId='" + departmentId + '\'' +
            ", departmentName='" + departmentName + '\'' +
            '}';
    }
}
```

XML

```
<!--员工类-->
<bean id="staff" class="com.test.Staff">
    <!--员工名-->
    <property name="staffName" value="王子枫"/>
    <!--员工年龄-->
    <property name="staffAge" value="13"/>
    <!--员工所在的部门-->
    <property name="department">
        <bean class="com.test.Department">
            <!--部门id-->
            <property name="departmentId" value="1"/>
            <!--部门名-->
            <property name="departmentName" value="安保部"/>
        </bean>
    </property>
</bean>
```

Test

```
@org.junit.Test
public void test(){
    ApplicationContext context = new ClassPathXmlApplicationContext( configLocation: "spring.xml");
    Staff staff = context.getBean(s: "staff", Staff.class);
    System.out.println(staff);
}
```

注入数组、集合、map

XML

```
<bean id="collection" class="com.test.CollectionArray">
    <!--数组-->
    <property name="strings">
        <list>
            <value>array1</value>
            <value>array2</value>
        </list>
    </property>
```

```

<!--list 集合-->
<property name="list">
    <list>
        <value>list1</value>
        <value>list2</value>
    </list>
</property>
<!--map 集合-->
<property name="map">
    <map>
        <entry key="map1" value="map1"/>
        <entry key="map2" value="map2"/>
    </map>
</property>
<!--set 集合-->
<property name="set">
    <list>
        <value>set1</value>
        <value>set2</value>
    </list>
</property>
</bean>

```

Test

```

@org.junit.Test
public void test(){
    ApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");
    CollectionArray collection = context.getBean("collection", CollectionArray.class);
    System.out.println(collection);
}

```

集合类注入

XML

```

<!--类集合-->
<bean id="collection" class="com.test.CollectionArray">
    <property name="persons">
        <list>
            <ref bean="person1"/>

```



```

        <ref bean="person2"/>
    </list>
</property>
</bean>

<bean id="person1" class="com.test.Person">
    <property name="name" value="马佳盛"/>
    <property name="age" value="13"/>
</bean>
<bean id="person2" class="com.test.Person">
    <property name="name" value="王子涵"/>
    <property name="age" value="12"/>
</bean>

```

工具类

```

public class Person {
    private String name;
    private int age;

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

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

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

```

存放工具类类

```

public class CollectionArray {

    private List<Person> persons;
}

```

```

public void setPersons(List<Person> persons) {
    this.persons = persons;
}

@Override
public String toString() {
    return "CollectionArray{" +
        "persons=" + persons +
        '}';
}
}

```

Test

```

public class Test {

    @org.junit.Test
    public void test(){
        ApplicationContext context = new
        ClassPathXmlApplicationContext("spring.xml");
        CollectionArray collection = context.getBean("collection",
        CollectionArray.class);
        System.out.println(collection);
    }
}

```

抽取公共部分

XML

```

<!--book 类-->
<bean id="book" class="com.test.Book">
    <!--使用 ref 连接-->
    <property name="bookName" ref="bookList"/>
</bean>

<!--公共部分-->
<util:list id="bookList">

```

```
<value>book1</value>
<value>book2</value>
</util:list>
```

Book

```
public class Book {

    private List<String> bookName;

    public void setBookName(List<String> bookName) {
        this.bookName = bookName;
    }
}
```

Test

```
public class Test {

    @org.junit.Test
    public void test(){
        ApplicationContext context = new
        ClassPathXmlApplicationContext("spring.xml");
        Book book = context.getBean("book", Book.class);
        System.out.println(book);
    }
}
```

工厂 Bean

- 1、Spring 有两种类型 bean，一种普通 bean，另外一种工厂 bean（FactoryBean）
- 2、普通 bean：在配置文件中定义 bean 类型就是返回类型
- 3、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

自定义工厂类

```
public class MyFactoryBean implements FactoryBean<Person> {

    @Override
    public Person getObject() throws Exception {
        return new Person();
    }

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

    @Override
    public boolean isSingleton() {
        return false;
    }
}
```

工具类

```
public class Person {
    private String name;
    private int age;

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

```

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

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

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Person() {
    }
}

```

XML

```

<bean id="myFactory" class="com.test.MyFactoryBean"/>

```

Test

```

public class Test {

    @org.junit.Test
    public void test(){
        ApplicationContext context = new
        ClassPathXmlApplicationContext("spring.xml");
        Person person = context.getBean("myFactory", Person.class);
        System.out.println(person);
    }
}

```

Bean 的作用域

使用 xml 创建 bean 的时候默认是单例模式,只能创建一个对象

如果想创建多个对象需要在 Bean 标签中加入 scope 属性值为 prototype

```
<bean id="myFactory" class="com.test.MyFactoryBean" scope="prototype"/>
```

Scope 有两个值:

Prototype:当调用 getBean 方法的时候创建对象

Singleton:当加载配置文件的时候创建单例对象

还有 2 个属性:

Request:创建时放到 request 域中

Session:创建时放到会话里

Bean 的生命周期

无后置处理器

- (1).无参构造器
- (2).属性赋值
- (3).初始化方法
- (4).创建对象
- (5).销毁方法

工具类

```
public class Person {
    private String name;

    public Person() {
        System.out.println("第一步无参构造器");
    }

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        System.out.println("第二步属性赋值");
        this.name = name;
    }

    public void initMethod(){
        System.out.println("第三步初始化方法");
    }

    public void destroy() {
        System.out.println("第五步销毁对象");
    }

    @Override
    public String toString() {
        return "Person{" +
            "name=" + name + "\" +
            '>';
    }
}
```

XML

```
<!--init:初始化方法      destroy:销毁方法-->
<bean id="person" class="com.test.Person" init-method="initMethod" destroy-
method="destroy">
    <property name="name" value="马佳盛"/>
</bean>
```

Test

```
public class Test {  
  
    @org.junit.Test  
    public void test(){  
        ClassPathXmlApplicationContext context = new  
ClassPathXmlApplicationContext("spring.xml");  
        Person person = context.getBean("person", Person.class);  
        System.out.println("第四步创建对象");  
        context.close();           //执行销毁方法  
    }  
}
```

有后置处理器

- (1).无参构造器
- (2).属性赋值
- (3). 将 Bean 实例传入到后置处理器
postProcessBeforeInitialization 方法中
- (4).初始化方法
- (5). 将 Bean 实例传入到后置处理器
postProcessAfterInitialization 方法中
- (6).创建对象
- (7).销毁方法

工具类

```
public class Person {  
    private String name;  
  
    public Person() {  
        System.out.println("第一步无参构造器");  
    }  
  
    public Person(String name) {
```



```

        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        System.out.println("第二步属性赋值");
        this.name = name;
    }

    public void initMethod(){
        System.out.println("第四步初始化方法");
    }

    public void destroy() {
        System.out.println("第七步销毁对象");
    }

    @Override
    public String toString() {
        return "Person{" +
            "name=" + name + "\n" +
            '}';
    }
}

```

后置转换器类

```

public class MyPost implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
        System.out.println("第三步 postProcessBeforeInitialization 方法");
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        System.out.println("第五步 postProcessAfterInitialization 方法");
        return bean;
    }
}

```

```
}
```

XML

```
<!--init:初始化方法      destroy:销毁方法-->
<bean id="person" class="com.test.Person" init-method="initMethod" destroy-
method="destroy">
    <property name="name" value="马佳盛"/>
</bean>

<!--当该类继承 BeanPostProcessor 接口的时候 Spring 自动识别为后置转换器,为所
有 XML 中创建的 Bean 加载后置转换器-->
<bean id="myPost" class="com.test.MyPost"/>
```

Test

```
public class Test {

    @org.junit.Test
    public void test(){
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("spring.xml");
        Person person = context.getBean("person", Person.class);
        System.out.println("第六步创建对象");
        context.close();          //执行销毁方法
    }
}
```

自动装配(自动注入属性值)

部门类

```
public class Dest {
    private String dName;

    public Dest() {
    }

    @Override
    public String toString() {
        return "Dest{" +
            "dName=" + dName +
            '}';
    }
}
```

```
}
```

员工类

```
public class Emp {  
    private Dest dest;  
  
    public Dest getDest() {  
        return dest;  
    }  
  
    public void setDest(Dest dest) {  
        this.dest = dest;  
    }  
  
    @Override  
    public String toString() {  
        return "Emp{" +  
            "dest=" + dest +  
            '}';  
    }  
}
```

XML

```
<!--自动装配(自动赋值) byName:类中属性值和 XML 中 id 值一样    byType:根据属性类型赋值-->  
<bean id="emp" class="com.test.Emp" autowire="byName"/>  
  
<bean id="dest" class="com.test.Dest"/>
```

Tset

```
public class Test {  
  
    @org.junit.Test  
    public void test(){  
        ClassPathXmlApplicationContext context = new  
ClassPathXmlApplicationContext("spring.xml");  
        Emp emp = context.getBean("emp", Emp.class);  
        System.out.println(emp);  
    }  
}
```

XML 读取配置文件

Properties

```
driverClassName=com.mysql.jdbc.Driver  
url=jdbc:mysql://localhost:3306/text  
username=root  
password=Qwer1234
```

XML

```
<!--创建德鲁伊池对象并使用 EL 表达式读取配置文件注入属性-->  
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">  
    <property name="driverClassName" value="${driverClassName}"/>  
    <property name="url" value="${url}"/>  
    <property name="username" value="${username}"/>  
    <property name="password" value="${password}"/>  
</bean>  
<!--需要导入头文件-->  
<context:property-placeholder location="jdbcPro.properties"/>
```

IOC 操作 Bean 管理(基于注解)

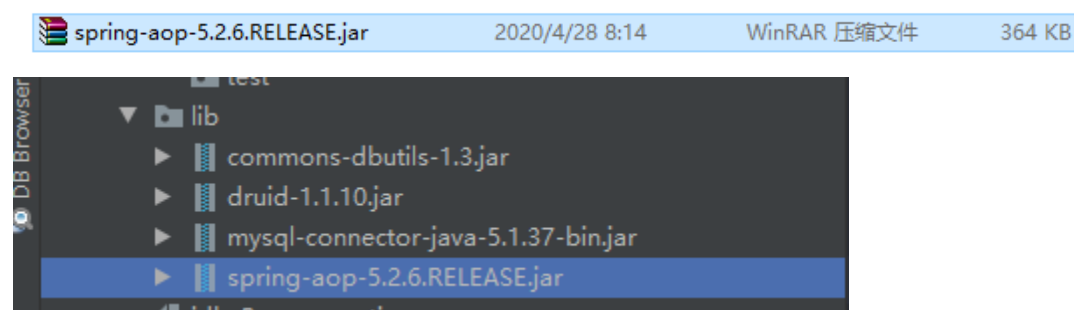
什么是注解

- (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

Spring 针对 Bean 管理中创建对象提供注解

- (1) @Component 普通类注解
 - (2) @Service service 层类注解
 - (3) @Controller servlet 层类注解
 - (4) @Repository dao 层类注解
- * 上面四个注解功能是一样的，都可以用来创建 bean 实例

导入 JAR 包



简单的使用

XML

```
<!--开启扫描注解,扫描文件为 com.test 文件下的所有文件-->
<context:component-scan base-package="com.test"/>
```

工具类

```
/*
    value 可以省略,默认为类名第一个字母小写
*/

@Component(value = "scanTest")
public class ScanTest {
    public void info(){
        System.out.println("这里是 scanTest");
    }
}
```

测试

```
public class Test {

    @org.junit.Test
    public void test1(){
        ApplicationContext context = new
        ClassPathXmlApplicationContext("spring.xml");
        ScanTest scanTest = context.getBean("scanTest", ScanTest.class);
        scanTest.info();
    }
}
```

扫描筛选

```
<!--关闭默认扫描,不扫描全部的类,只扫描包含 Controller 类-->
<context:component-scan base-package="com.test" use-default-filters="false">
    <context:include-filter type="annotation"
        expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
```

```
</context:component-scan>
<!--不扫描包含 Controller 的类-->
<context:component-scan base-package="com.test">
    <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
```

属性注入

@Autowired 根据类型

@Qualifier 根据 name 与上边同时使用

@Resource 可以根据类型,可以根据 name,不加参数为类型,
加(name="name")为根据 name

以上为类对象注入

普通类型注入

@value 注入普通类型

类属性(@Autowired @Qualifier @Resource)

XML

```
<!--扫描 Service 和 Dao 层文件-->
<context:component-scan base-package="com.service,com.dao"/>
```

DAO

```
@Repository(value = "userDAOimpl")
public class UserDAOimpl implements UserDAO{
    @Override
    public void add() {
        System.out.println("UserDao add");
    }
}
```

```
}  
}
```

Service

```
@Service  
public class UserServiceImpl implements UserService{  
  
    第一种方式  
    @Autowired  
    @Qualifier("userDAOimpl")  
    -----  
    第二种方式  
    @Resource    根据类型  
    @Resource(name = "userDAOimpl") 根据 name  
    private UserDAO userdao = new UserDAOimpl();  
  
    @Override  
    public void add() {  
        System.out.println("service add....");  
        userdao.add();  
    }  
}
```

普通属性注入(@value)

```
@Value("哈哈")  
private String name;
```

完全注解开发不用 XML 文件

1. 创建扫描类

```
@Configurable  
@ComponentScan(basePackages = {"com.service","com.dao" })  
public class scanSPring {  
}
```

2. 读取扫描类


```
public class test {  
  
    @Test  
    public void test1(){  
        ApplicationContext context = new  
AnnotationConfigApplicationContext(scanSPring.class);  
        UserServiceImpl userService = context.getBean("userServiceImpl",  
        UserServiceImpl.class);  
        userService.add();  
    }  
}
```

AOP

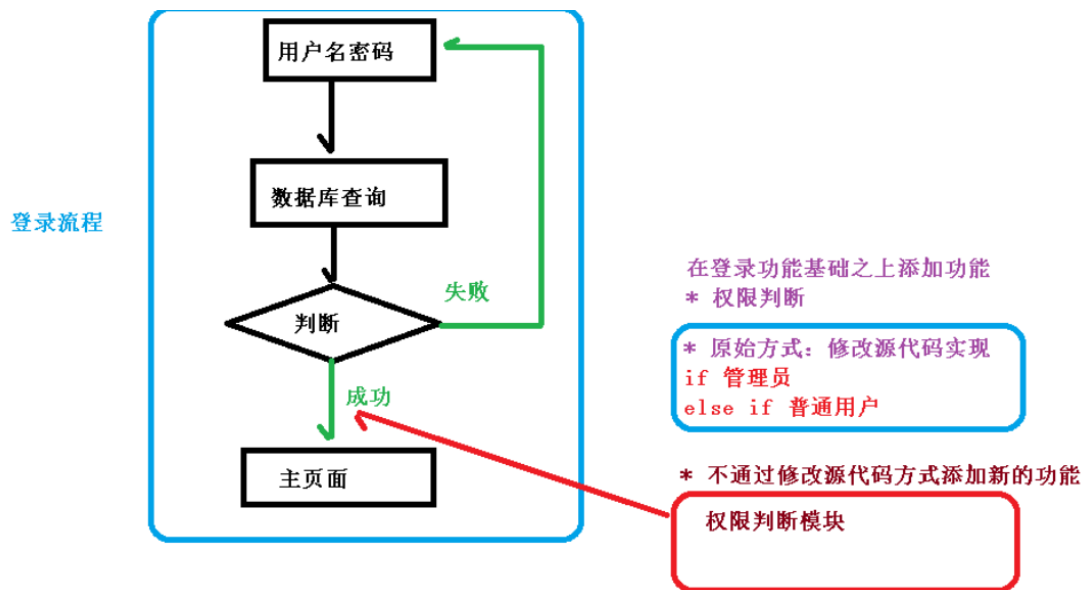
概念

在不修改源代码的情况下添加业务

(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) 使用登录例子说明 **AOP**

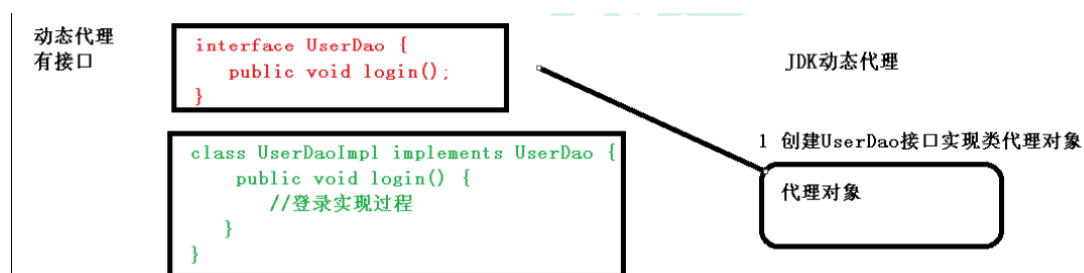


底层原理

AOP 底层使用动态代理(分为 2 种)

第一种:有接口的情况下

创建接口实现类代理对象，增强类的方法



第二种:无接口的情况下

创建子类的代理对象，增强类的方法

动态代理
没有接口情况

```
class User {  
    public void add() {  
        .....  
    }  
}
```

子类

```
class Person extends User {  
    public void add() {  
        super.add();  
        //增强逻辑  
    }  
}
```

CGLIB动态代理

1 创建当前类子类的代理对象

代理对象

原生动态代理模式

1. 创建公共接口
2. 创建被代理类实现公共接口
3. 创建代理类实现 `InvocationHandler` 接口并在实现方法中添加业务
4. 测试方法

代码

公共接口(创建了加、减方法)

```
public interface PublicInterface {  
  
    int add(int a,int b);  
  
    int reduce(int a,int b);  
}
```

被代理类(只是返回了结果)

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

    @Override
    public int reduce(int a, int b) {
        return a-b;
    }
}
```

代理类

```
public class PassIveProxyClass implements InvocationHandler {

    //创建一个接受被代理类的对象
    private Object obj;

    //使用构造器赋值
    public PassIveProxyClass(Object obj){
        this.obj = obj;
    }

    //重写方法
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
    Throwable {
        //添加执行方法前业务
        System.out.println("当前执行的方法是"+method.getName() +"参数为
        "+args.toString());
        //执行原生业务 第一个为被代理类对象 第二个参数为 args
        Object invoke = method.invoke(obj, args);
        //添加执行完方法后的业务
        System.out.println("执行完成");
        //返回执行结果
        return invoke;
    }
}
```

测试

```
public class Test {  
  
    public static void main(String[] args) {  
        //创建一个 Class 数组封装被代理类对象  
        Class[] publicInterfaceClasses = {  
            PublicInterface.class  
        };  
        //创建被代理类对象  
        InitiativeProxyClass initiat = new InitiativeProxyClass();  
        //调用 proxy.newProxyInstance 完成代理  args1:类加载器  args2:  
被代理类 Class 数组  args3:代理类  
        PublicInterface publicInterface =  
(PublicInterface)Proxy.newProxyInstance(Test.class.getClassLoader(),  
                                           publicInterfaceClasses,  
new PassIveProxyClass(initiat));  
        //调用被代理类方法  
        int add = publicInterface.add(1, 3);  
        System.out.println(add);  
    }  
}
```

AOP 术语

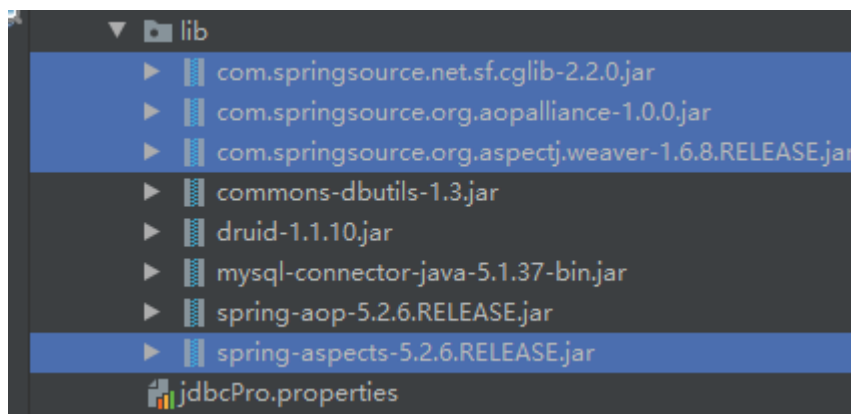
1. 连接点:类的哪些方法可以被增强,这些方法成为连接点
2. 切入点:实际被真正增强的方法,叫做切入点
3. 通知(增强)
 - a) 实际添加的业务
 - b) 通知分为多种类型

- i. 前置通知:原生代码前执行的业务
- ii. 后置通知:原生代码后执行的业务
- iii. 环绕通知:原生代码前后都有的业务
- iv. 异常通知:原生代码发生异常执行的业务
- v. 最终通知:不管什么最终都运行的业务

AOP 准备工作

导入 JAR 包

spring-aop-5.2.6.RELEASE.jar	2020/4/28 8:15	WinRAR 压缩文件	47 KB
com.springsource.net.sf.cglib-2.2.0.jar	2018/12/24 20:59	WinRAR 压缩文件	320 KB
com.springsource.org.aopalliance-1.0.0.jar	2018/12/24 20:59	WinRAR 压缩文件	5 KB
com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar	2018/12/24 20:59	WinRAR 压缩文件	1,604 KB



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:execution([权限修饰符][返回类型][类全路径][方法名

称][参数列表])

举例一:对 com.test 类里面的 add 进行增强

```
execution(* com.test.add(..))
```

举例二:对 com.test 类里面的所有方法进行增强

```
execution(* com.test.*(..))
```

举例三:对 com 包里面的所有类进行增强

```
Exection(* com.*.*(..))
```

AOP 操作(注解完成动态代理)

1. 创建被代理类

```
public class BProxyClass {  
  
    public void eat(){  
        System.out.println("BProxyClass 正在吃东西");  
    }  
}
```

2. 创建代理类

```
public class ZProxyClass {  
  
    public void sports(){  
        System.out.println("做运动");  
    }  
}
```

```
}
```

3. 配置文件通知配置

开启扫描

```
<!--扫描 Service 和 Dao 层文件-->
<context:component-scan base-package="com.Action"/>
<!--开启代理模式-->
<aop:aspectj-autoproxy/>
```

配置注解

```
@Component //被代理对象
public class BProxyClass {

@Component
@Aspect //代理对象
public class ZProxyClass {
```

4. 配置不通类型通知

```
@Component
@Aspect //代理对象
public class ZProxyClass {

    //前置通知 value="execution(返回值类型,包下的哪个类.什么方法(参数列表))"
    @Before(value = "execution(* com.AnnotationProxy.BProxyClass.eat(..))")
    public void sports(){
        System.out.println("前置通知");
    }

    //后置通知 如果出异常不执行
    @AfterReturning(value = "execution(* com.AnnotationProxy.BProxyClass.eat())")
    public void afterReturning() {
        System.out.println("后置通知");
    }

    //异常通知 不出异常不通知
    @AfterThrowing(value = "execution(*
```



```

com.AnnotationProxy.BProxyClass.eat())"
    public void afterThrowing(){
        System.out.println("异常通知");
    }

    //最终通知
    @After(value = "execution(* com.AnnotationProxy.BProxyClass.eat())")
    public void after(){
        System.out.println("最终通知");
    }

    //环绕通知
    @Around(value = "execution(* com.AnnotationProxy.BProxyClass.eat())")
    public void around(ProceedingJoinPoint proceedingJoinPoint){
        System.out.println("前置通知");
        try {
            proceedingJoinPoint.proceed();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
        System.out.println("后置通知");
    }
}

```

5. 提取公共的 value

```

@Pointcut(value = "execution(* com.AnnotationProxy.BProxyClass.eat())")
public void publicValue() {

}

```

这样如果下面有和这个 **value** 相同的就可以使用

```

    @Before(value = "publicValue()")
    public void sports(){
        System.out.println("前置通知");
    }

```

6. 如果多个代理对象完成同一个被代理类的方法,可以使用 优先级限制

在类的上面添加注解 **@order(1)** 数字越小越有先

```
@Component
@Aspect //代理对象
@Order(1)
public class ZProxyClass {
```

7. 测试

```
public class Test {
    public static void main(String[] args) {
        ApplicationContext context = new
        ClassPathXmlApplicationContext("spring.xml");
        BProxyClass bProxyClass =
        context.getBean("bProxyClass",BProxyClass.class);
        bProxyClass.eat();
    }
}
```

AOP 操作(基于 XML)

1、 创建两个类，增强类和被增强类，创建方法

2、 在 spring 配置文件中创建两个类对象

<!--创建对象-->

<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 增强-->

<aop:config>

<!--切入点-->

<aop:pointcut id="p" expression="execution(*

com.atguigu.spring5.aopxml.Book.buy(..))"/>

```
<!--配置切面-->
<aop:aspect ref="bookProxy">
<!--增强作用在具体的方法上-->
<aop:before method="before" pointcut-ref="p"/>
</aop:aspect>
</aop:config>
```

完全注解开发





(1) 创建配置类，不需要创建 xml 配置文件

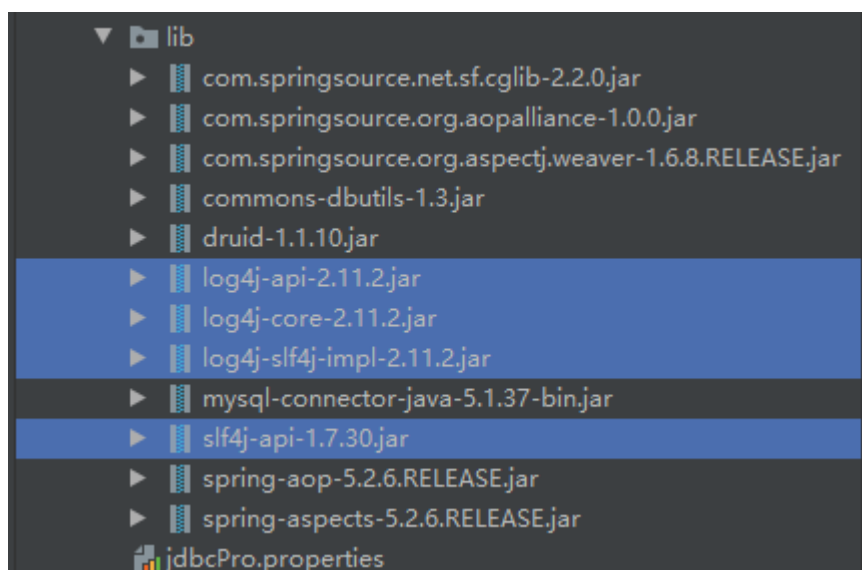
```
@Configuration
@ComponentScan(basePackages = {"com.AnnotationProxy"})
//开启扫描
@EnableAspectJAutoProxy(proxyTargetClass = true)// 开 启
自动代理默认为 True
public class ConfigAop {
}
```

Spring5 新功能

整合日志框架

导入 jar 包

 log4j-api-2.11.2.jar	2019/2/19 13:39	WinRAR 压缩文件	261 KB
 log4j-core-2.11.2.jar	2020/5/18 22:23	WinRAR 压缩文件	1,592 KB
 log4j-slf4j-impl-2.11.2.jar	2020/5/18 22:36	WinRAR 压缩文件	23 KB
 slf4j-api-1.7.30.jar	2020/2/13 15:51	WinRAR 压缩文件	41 KB



创建 log4j2.xml 配置文件(名字固定,直接复制进去即可)

```
<?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>
```

@nullable 和支持 lambda 表达式

(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

```
@Nullable  
String getId();
```

(3) 注解使用在方法参数里面，方法参数可以为空

```
public <T> void registerBean(@Nullable String beanName,  
    this.reader.registerBean(beanClass, beanName, suppli  
}
```

(4) 注解使用在属性上面，属性值可以为空

```
@Nullable  
private String bookName;
```

Lambda 表达式

//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.getBean("com.atguigu.spring5.test.User");

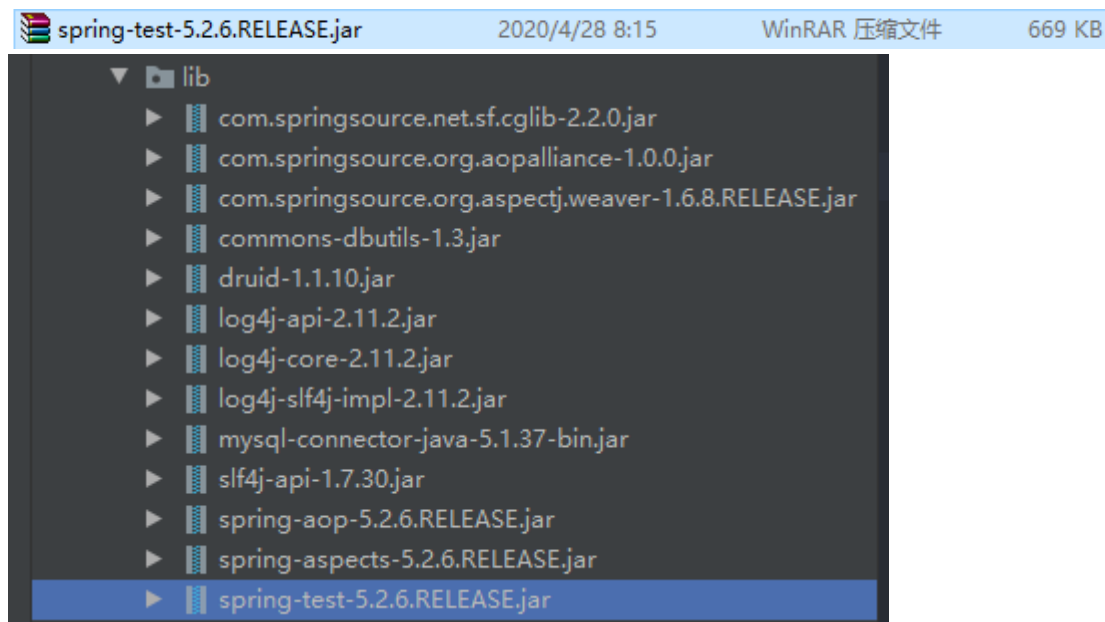
User user = (User)context.getBean("user1");

System.out.println(user);

}
```

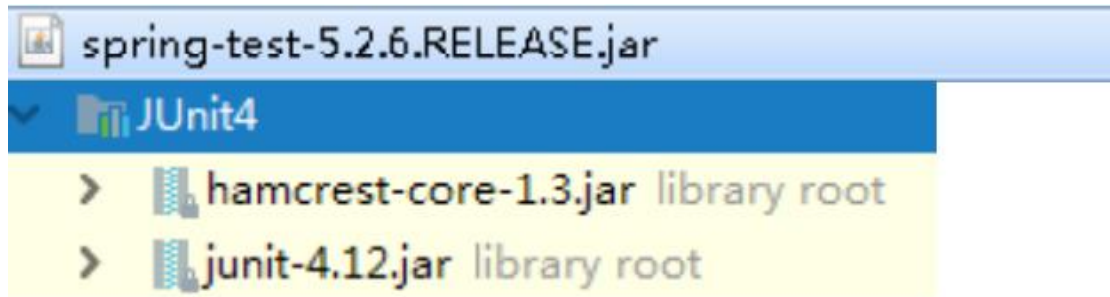
Junit5 整合框架

导入 jar 包



整合 Junit4 演示

第一步 引入 Spring 相关测试依赖



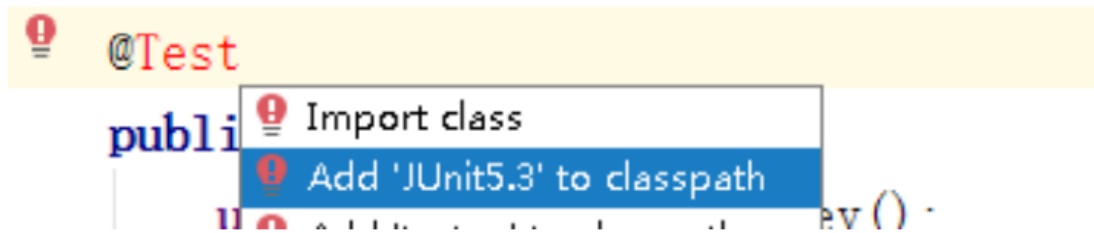
第二步 创建测试类，使用注解方式完成

```
@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();
    }
}
```

整合 Junit5 演示

第一步 引入 JUnit5 的 jar 包



第二步 创建测试类，使用注解完成

```
@SpringJUnitConfig(locations = "classpath:bean1.xml")
public class JTest5 {
    @Autowired
    private UserService userService;

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