

Spring 5

笔记本： Spring

创建时间： 2020.08.20 22:00

更新时间： 2020.08.24 15:40

作者： 195330205@qq.com

URL： https://www.matools.com/file/manual/jdk_api_1.8_google/java/lang/reflect/Prox...



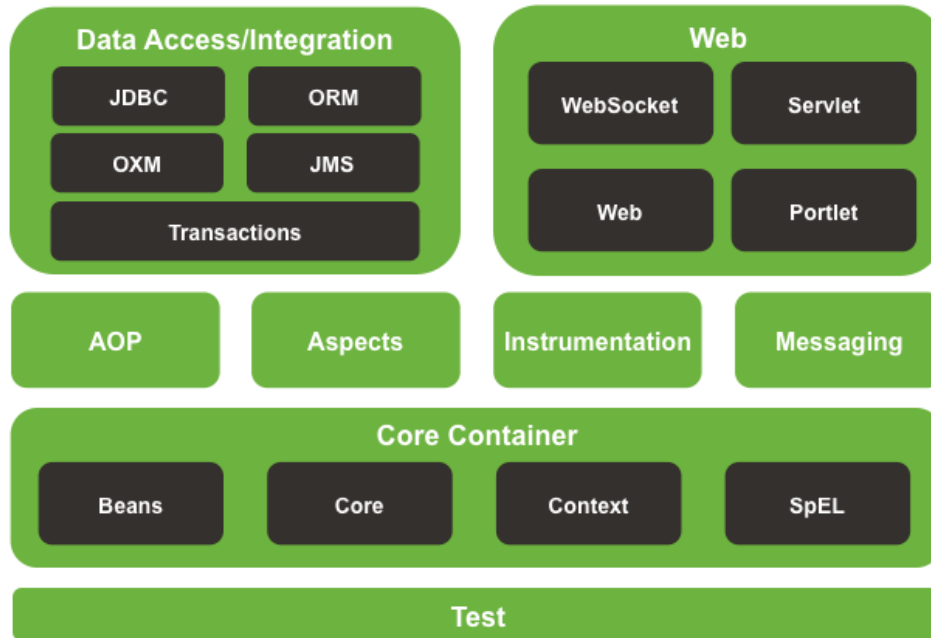
- <https://spring.io>
-

Spring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
 - IOC：控制反转，把创建对象过程交给 Spring 进行管理
 - AOP：面向切面编程，不修改源代码进行功能增强
 4. Spring 特点
 - 方便解耦，简化开发
 - AOP 编程支持
 - 方便程序测试
 - 方便和其他框架进行整合
 - 方便进行事务操作
 - 降低 API 开发难度
-



Spring Framework Runtime



Spring IOC

- > commons-logging-1.1.1.jar
- > spring-beans-5.2.6.RELEASE.jar
- > spring-context-5.2.6.RELEASE.jar
- > spring-core-5.2.6.RELEASE.jar
- > spring-expression-5.2.6.RELEASE.jar
- > spring-aop-5.2.6.RELEASE.jar

1. 什么是 IOC

- IOC (Inversion of Control) : 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
- DI (Dependency Injection) : 依赖注入，就是注入属性
- IoC是一种思想，DI是对IOC思想的具体实现
- 使用 IOC 目的：为了耦合度降低

2. IOC 底层原理

- XML解析
- 反射
- 工厂模式

3. BeanFactory 接口

- IOC 思想基于 IOC 容器完成，IOC 容器底层就是对象工厂
- Spring 提供 IOC 容器实现两种方式：（两个接口）
 - 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

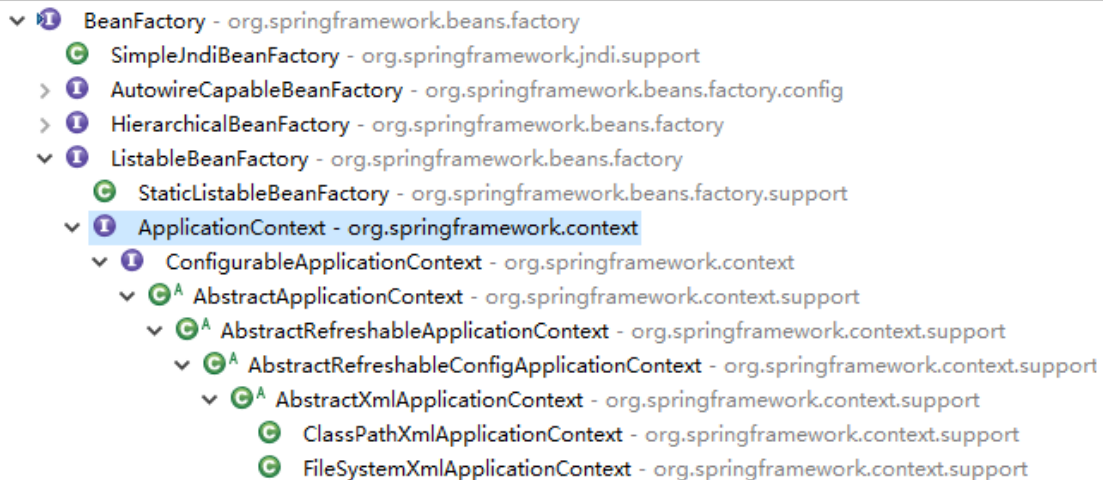
* 加载配置文件时候不会创建对象，在获取对象（使用）才去创建对象

- 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

- ApplicationContext 接口的实现类

Type hierarchy of 'org.springframework.beans.factory.BeanFactory':



4. IOC 操作 Bean 管理

- Bean 管理指的是两个操作：
 - Spring 创建对象
 - Spring 注入属性
- Bean 管理操作有两种方式
 - 基于 xml 配置文件方式实现
 - 基于注解方式实现

基于 xml 配置文件方式实现

基于 xml 方式创建对象

创建对象时候，默认是执行无参数构造方法完成对象创建

```
<!-- User user = new User(); -->
<bean id="user" class="com.clps.spring5.bean.User">
</bean>
```

基于 xml 方式注入属性

1. 使用 set 方法进行注入

Step 1 : 创建类，定义属性和对应的 set 方法

```
public class User {

    private String username;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

}
```

Step 2 : 在 spring 配置文件配置对象创建，配置属性注入

```
<!-- User user = new User(); -->
<bean id="user" class="com.clps.spring5.bean.User">
    <property name="username" value="Tom"></property>
</bean>
```

2. 使用有参构造方法进行注入

Step 1 : 创建类，定义属性，创建属性对应参数构造方法

```
public class Order {

    private String address;

    public Order(String address) {
        this.address = address;
    }

}
```

Step 2 : 在 spring 配置文件中配置

```
<bean id="order" class="com.clps.spring5.bean.Order">
    <constructor-arg name="address" value="Dalian"></constructor-arg>
</bean>
```

3. p 名称空间注入

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- User user = new User(); -->
    <bean id="user" class="com.clps.spring5.bean.User" p:username="Lucy">
        <!-- <property name="username" value="Tom"></property> -->
    </bean>
```

xml 注入其他类型属性

1. 字面量

- null

```
<property name="address">
    <null/>
</property>
```

- 属性值包含特殊符号

```
<property name="address">
    <value><![CDATA[<<大连>>]]></value>
</property>
```

2. 注入属性 - 外部 bean

- 创建两个类 service 类和 dao 类
- 在 service 调用 dao 里面的方法

```

public class UserServiceImpl implements UserService {

    private UserDao userDao;

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

    @Override
    public void updateUserInfo() {
        userDao.updateUserInfo();
    }
}

public class UserDaoImpl implements UserDao {

    @Override
    public void updateUserInfo() {
        System.out.println("Update User Info");
    }
}

```

- 在 spring 配置文件中配置

```

<bean id="userService" class="com.clps.spring5.service.UserServiceImpl">
    <property name="userDao" ref="userDaoImpl"></property>
</bean>

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

```

3. 注入属性 - 内部 bean

- 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
- 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

```

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

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

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

    public void printEmpInfo() {
        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 + "]";
    }

}

```

```

<bean id="emp" class="com.clps.spring5.bean.Emp">
    <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.clps.spring5.bean.Dept">
            <property name="dname" value="Market"></property>
        </bean>
    </property>

</bean>

```

4. 注入属性 - 级联赋值

```

<bean id="emp" class="com.clps.spring5.bean.Emp">
    <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.clps.spring5.bean.Dept">
    <property name="dname" value="Tech"></property>
</bean>

```

xml 注入集合属性

1. 注入数组类型属性

```

public class Student {

    // 1 数组类型属性
    private String[] courses;

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

}

```



```

<bean id="student" class="com.clps.spring5.bean.Student">

    <!-- array 类型属性注入 -->
    <property name="courses">
        <array>
            <value>Java</value>
            <value>JavaScript</value>
            <value>jQuery</value>
            <value>Angular</value>
            <value>Spring</value>
            <value>Spring</value>
        </array>
    </property>

</bean>

```

2. 注入 List 集合类型属性

```

public class Student {

    // 2 list 集合类型属性
    private List<String> list;

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

}

```

```

<bean id="student" class="com.clps.spring5.bean.Student">

    <!-- list 类型属性注入 -->
    <property name="list">
        <list>
            <value>AAAA</value>
            <value>BBBB</value>
            <value>CCCC</value>
            <value>DDDD</value>
            <value>DDDD</value>
        </list>
    </property>

</bean>

```

3. 注入 Set 集合类型属性

```
public class Student {  
  
    // 4 set 集合类型属性  
    private Set<String> sets;  
  
    public void setSets(Set<String> sets) {  
        this.sets = sets;  
    }  
  
}  
  
<bean id="student" class="com.clps.spring5.bean.Student">  
  
    <!-- set 类型属性注入 -->  
    <property name="sets">  
        <set>  
            <value>Java</value>  
            <value>JavaScript</value>  
            <value>jQuery</value>  
            <value>Angular</value>  
            <value>Spring</value>  
            <value>Spring</value>  
        </set>  
    </property>  
  
</bean>
```

4. 注入 Map 集合类型属性

```
public class Student {  
  
    // 3 map 集合类型属性  
    private Map<String, String> maps;  
  
    public void setMaps(Map<String, String> maps) {  
        this.maps = maps;  
    }  
  
}
```

```
<bean id="student" class="com.clps.spring5.bean.Student">

    <!-- map 类型属性注入 -->
    <property name="maps">
        <map>
            <entry key="k1" value="Java"></entry>
            <entry key="k2" value="JavaScript"></entry>
            <entry key="k3" value="jQuery"></entry>
            <entry key="k4" value="Angular"></entry>
            <entry key="k5" value="Spring"></entry>
            <entry key="k6" value="Spring"></entry>
            <entry key="k6" value="Oracle"></entry>
        </map>
    </property>

</bean>
```

5. 在集合里面设置对象类型值

```
public class Student {

    // 在集合里面设置对象类型值
    private List<Course> courseList;

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

}
```

```

public class Course {

    private String cname;

    public String getCname() {
        return cname;
    }

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

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

}

```

```

<bean id="student" class="com.clps.spring5.bean.Student">

```

```

    <!--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>

```

```

</bean>

```

```

<bean id="course1" class="com.clps.spring5.bean.Course">

```

```

    <property name="cname" value="Java"></property>

```

```

</bean>

```

```

<bean id="course2" class="com.clps.spring5.bean.Course">

```

```

    <property name="cname" value="JavaScript"></property>

```

```

</bean>

```

6. 把集合公共注入部分提取出来

```

public class Book {

    private List<String> list;

    public List<String> getList() {
        return list;
    }

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

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

}

```

```

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

```

```

<bean id="book" class="com.clps.spring5.bean.Book">
    <property name="list" ref="bookList"></property>
</bean>

```

```

<util:list id="bookList">
    <value>Effective Java Programming Language Guide</value>
    <value>Design Patterns: Elements of Reusable Object-Oriented Software</value>
    <value>Refactoring: Improving the Design of Existing Code</value>
</util:list>

```

FactoryBean

Spring 有两种类型 bean :

1. 普通 bean : 在配置文件中定义 bean 类型就是返回类型
2. 工厂 bean : 在配置文件定义 bean 类型可以和返回类型不一样

Step 1 : 创建类 , 让这个类作为工厂 bean , 实现接口 FactoryBean

```

public class MyFactoryBean implements FactoryBean<Course> {

    // 定义返回 bean
    @Override
    public Course getObject() throws Exception {
        Course course = new Course();
        course.setCname("Angular");
        return course;
    }

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

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

public class Course {

    private String cname;

    public String getCname() {
        return cname;
    }

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

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

```

Step 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

```
<bean id= "myFactoryBean" class="com.clps.spring5.bean.MyFactoryBean"></bean>
```

```

@Test
public void testFactoryBean() {
    Course course = context.getBean("myFactoryBean", Course.class);
    System.out.println(course);
}

```

bean 作用域

- 在 Spring 中，可以设置创建 bean 实例是单实例或多实例
- 默认情况下，bean 是单实例对象
- 设置单实例还是多实例
 - 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
 - scope 属性值及区别
 - singleton：单实例，加载 spring 配置文件时候就会创建单实例对象

```

<bean id="user" class="com.clps.spring5.bean.User" p:username="Lucy" scope="singleton">
    <!-- <property name="username" value="Tom"></property> -->
</bean>

```

- prototype：多实例，不是在加载 spring 配置文件时候创建对象，而是在调用 getBean 方法时候创建多实例对象

```

<bean id="user" class="com.clps.spring5.bean.User" p:username="Lucy" scope="prototype">
    <!-- <property name="username" value="Tom"></property> -->
</bean>

```

- request：web域对象，表示一次请求中
- session：web域对象，表示一次会话中

bean 生命周期

从对象创建到对象销毁的过程

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 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）

```

public class Orders {

    public Orders() {
        System.out.println("1. 通过无参构造器创建 bean 实例");
    }

    private String oname;

    public void setName(String oname) {
        this.oname = oname;
        System.out.println("2. 调用 set 方法设置属性值");
    }

    // 创建执行的初始化的方法
    public void initMethod() {
        System.out.println("3. 执行初始化的方法 initMethod");
    }

    // 创建执行的销毁的方法
    public void destroyMethod() {
        System.out.println("5. 执行销毁的方法 destroyMethod");
    }

}

<bean id="orders" class="com.clps.spring5.lifecycle.Orders"
    init-method="initMethod" destroy-method="destroyMethod">
    <property name="oname" value="Book"></property>
</bean>

```

1. 通过无参构造器创建 bean 实例
2. 调用 set 方法设置属性值
3. 执行初始化的方法 initMethod
4. 获取创建 bean 实例对象
5. 执行销毁的方法 destroyMethod

bean 的后置处理器，bean 生命周期有7步

实现 BeanPostProcessor 接口

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）

```
public class MyBeanPostProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("在初始化方法之前执行");
        return bean;
    }

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

}
```

```
<bean id="myBeanPostProcessor" class="com.clps.spring5.postprocessor.MyBeanPostProcessor"></bean>
```

```
// bean生命周期 和 bean后置处理器
@Test
public void testLifeCycleAndPostProcessor() {
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext9.xml");
    Orders orders = context.getBean("orders", Orders.class);
    System.out.println("4. 获取创建 bean 实例对象 ");
    // IOC容器关闭, bean实例销毁
    context.close();
}
```

1. 通过无参构造器创建 bean 实例
2. 调用 set 方法设置属性值
在初始化方法之前执行
3. 执行初始化的方法 initMethod
在初始化方法之后执行
4. 获取创建 bean 实例对象
5. 执行销毁的方法 destroyMethod

基于注解方式实现

注解

- 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...)
- 注解可以作用于 类、属性、方法 上面
- 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

- **@Component** 被IoC容器管理的组件
- **@Controller** 表述层控制器组件
- **@Service** 业务逻辑层组件
- **@Repository** 持久化层组件，DAO

* 上面四个注解功能是一样的，都可以用来创建 bean 实例

基于注解方式实现对象创建

Step 1：引入依赖，spring-aop-5.2.6.RELEASE.jar

Step 2：开启组件扫描

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       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">

    <context:component-scan base-package="com.clps.spring5.annotation.bean"></context:component-scan>

</beans>
```

Step 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
- User : user

```
// 被IoC容器管理的组件
@Component
public class User {

    public void createUser() {
        System.out.println("User is created");
    }

}
```

基于注解方式实现属性注入

@Autowired：默认byType，根据属性类型进行自动装配，不需要 set 方法

Step 1：用添加注解的方式创建 service 和 dao 对象

```

@Repository(value="asfdfgsdfgsdfgsdfgsdf")
public class UserDaoImpl implements UserDao {

    @Override
    public void updateUserInfo() {
        System.out.println("Update User Info");
    }

}

```

Step 2 : 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

```

// <bean id="userService" class="com.clps.spring5.annotation.service.UserServiceImpl"></bean>
@Service(value="userService")
public class UserServiceImpl implements UserService {

    // 添加注入属性注解，不需要添加 set 方法
    @Autowired
    private UserDao userDao;

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

    @Override
    public void updateUserInfo() {
        userDao.updateUserInfo();
    }

}

```

@Qualifier : 默认byName，根据名称进行注入，需要和@Autowired一起使用

```

@Repository(value="asfdfgsdfgsdfgsdfgsdf")
public class UserDaoImpl implements UserDao {

    @Override
    public void updateUserInfo() {
        System.out.println("Update User Info");
    }

}

```

```
// <bean id="userService" class="com.clps.spring5.annotation.service.UserServiceImpl"></bean>
@Service(value="userService")
public class UserServiceImpl implements UserService {

    // 添加注入属性注解，不需要添加 set 方法
    @Autowired
    @Qualifier(value="asfdffgsdfgsdfgsdf")
    private UserDao userDao;

    @Override
    public void updateUserInfo() {
        userDao.updateUserInfo();
    }
}
```

@Resource：可以根据类型byType注入，也可以根据名称byName注入是JDK1.6支持的注解，javax.annotation.Resource，名称可以通过name属性进行指定，如果没有指定name属性，当注解写在字段上时，默认取字段名，按照名称查找，如果注解写在setter方法上默认取属性名进行装配。当找不到与名称匹配的bean时才按照类型进行装配。但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配。

```
@Repository(value="asfdffgsdfgsdfgsdf")
public class UserDaoImpl implements UserDao {

    @Override
    public void updateUserInfo() {
        System.out.println("Update User Info");
    }

}
```

byType :

```
// <bean id="userService" class="com.clps.spring5.annotation.service.UserServiceImpl"></bean>
@Service(value="userService")
public class UserServiceImpl implements UserService {

    // 添加注入属性注解，不需要添加 set 方法
    @Resource
    private UserDao userDao;

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

    @Override
    public void updateUserInfo() {
        userDao.updateUserInfo();
    }
}
```

byName :

```
// <bean id="userService" class="com.clps.spring5.annotation.service.UserServiceImpl"></bean>
@Service(value="userService")
public class UserServiceImpl implements UserService {

    @Resource(name="asdfdfgsdfgsdfgdfgsdf")
    private UserDao userDao;

    @Override
    public void updateUserInfo() {
        userDao.updateUserInfo();
    }

}
```

@Value : 注入普通类型属性

```
// 被IoC容器管理的组件
@Component
public class User {

    @Value(value="Tom")
    private String username;

    public void createUser() {
        System.out.println("User is created");
    }

    @Override
    public String toString() {
        return "User [username=" + username + "]";
    }

}
```

组件扫描细节配置

- 如果扫描多个包，多个包使用逗号隔开

```
<context:component-scan base-package="com.clps.spring5.annotation.dao,com.clps.spring5.annotation.service"></context:component-scan>
```

- 扫描包上层目录

```
<context:component-scan base-package="com.clps.spring5.annotation"></context:component-scan>
```

- include-filter

设置扫描哪些内容，use-default-filters="false"（默认：true）表示现在不使用默认 filter，而自己配置 filter，context:include-filter

```
<!-- 扫描com.clps.spring5包下所有带有@Controller注解的类 -->
<context:component-scan base-package="com.clps.spring5" use-default-filters="false">
    <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller" />
</context:component-scan>
```

- exclude-filter

设置哪些内容不进行扫描，context:exclude-filter

```
<!-- 不扫描com.clps.spring5包下所有带有@Controller注解的类 -->
<context:component-scan base-package="com.clps.spring5">
    <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller" />
</context:component-scan>
```

完全使用注解开发

1. 创建配置类，替代 xml 配置文件

```
package com.clps.spring5.annotation.config;

import org.springframework.context.annotation.ComponentScan;

@Configuration
@ComponentScan(basePackages = { "com.clps.spring5.annotation" })
public class ApplicationContextConfig {

}
```

2. 加载配置类

```
// 加载配置类，完全使用注解开发
ApplicationContext context = new AnnotationConfigApplicationContext(ApplicationContextConfig.class);

@Test
public void testAnnotation() {
    User user = context.getBean("user", User.class);
    user.createUser();
}
```

在实际开发中，会使用 SpringBoot

Spring AOP

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
- 不通过修改源代码方式，在主干功能里面添加新功能
- AOP 是在程序运行期的一种行为，是指在程序运行期动态地将某段功能代码切入或织入到指定方法和指定位置的一种编程方式
- 在使用 AOP 之前，系统级业务和主业务逻辑业务交织耦合在一起，程序的可扩展性和可维护性很差，并且有代码冗余
- 使用 AOP 可以解耦合（系统级业务和主业务逻辑业务之间的耦合），提高程序的可扩展性和可维护性，并减少代码冗余

AOP 底层使用**动态代理**

有两种情况的动态代理：

1. 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

2. 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

JDK 动态代理

使用 Proxy 类里面的方法创建代理对象

<https://docs.oracle.com/javase/8/docs/api/index.html>

java.lang.reflect

Class Proxy

java.lang.Object

java.lang.reflect.Proxy

static Object

newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)

Returns an instance of a proxy class for the specified interfaces that dispatches method invocations to the specified invocation handler.

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。

ClassLoader loader：类加载器

类<?>[] interfaces：增强方法所在的类，这个类实现的接口，支持多个接口

InvocationHandler h：实现这个接口 InvocationHandler，创建代理对象，写增强的部分

1. 创建接口，定义方法

```
public interface UserDao {  
  
    String add(String name);  
  
}
```

2. 创建接口实现类，实现方法

```
public class UserDaoImpl implements UserDao {  
  
    @Override  
    public String add(String name) {  
        System.out.println("add username");  
        return name;  
    }  
  
}
```

3. 使用 Proxy 类创建接口代理对象


```

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

public class JDKProxy {

    public static void main(String[] args) {
        // 创建接口实现类代理对象
        Class<?>[] interfaces = { UserDao.class };

        UserDao userDao = new UserDaoImpl();

        // UserDao userDaoProxy = (UserDao) Proxy.newProxyInstance(JDKProxy.class.getClassLoader(), interfaces, new UserDaoProxy(userDao));

        UserDao userDaoProxy = (UserDao) Proxy.newProxyInstance(JDKProxy.class.getClassLoader(), interfaces, new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                // 方法执行之前
                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(userDao, args);

                // 方法执行之后
                System.out.println("方法执行之后: " + userDao);

                return res;
            }
        });

        String result = userDaoProxy.add("Tom");
        System.out.println("result:" + result);
    }
}

```

AOP 术语

1. 连接点

类里面哪些方法可以被增强

```
// 被增强类
@Component
public class User {

    // 连接点
    public void add() {
        System.out.println("User add method is invoked");
    }

    // 连接点
    public void delete() {
    }

    // 连接点
    public void update() {
    }

    // 连接点
    public void query() {
    }
}
```

2. 切入点：**@PointCut, JoinPoint joinPoint**

实际被增强的方法

```

// 被增强类
@Component
public class User {

    // 连接点
    // 切入点
    public void add() {
        System.out.println("User add method is invoked");
    }

    // 连接点
    public void delete() {
    }

    // 连接点
    public void update() {
    }

    // 连接点
    public void query() {
    }

}

```

切入点表达式：**@PointCut("execution(public int)")**

- 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]))

Sample 1：对 com.clps.dao.UserDao 类里面的 add 进行增强

execution(* com.clps.dao.UserDao.add(..))

Sample 2：对 com.clps.dao.UserDao类里面的所有的方法进行

增强

execution(* com.clps.dao.UserDao.*(..))

Sample 3：对 com.clps.dao 包里面所有类，类里面所有方法进行

增强

execution(* com.clps.dao.*.*(..))

3. 通知（增强）

实际增强的逻辑部分

通知 Advice :

- 前置通知 : **@Before**
- 后置通知 : **@After**
- 环绕通知 : **@Around, around(ProceedingJoinPoint proceedingJoinPoint)**
- 返回通知 : **@AfterReturning(value="pointCut()", returning="result")**
- 异常通知 : **@AfterThrowing(value="pointCut()", throwing="exception")**

```
// 增强类
@Component
@Aspect    // 生成代理对象
public class UserProxy {

    @Pointcut(value = "execution(* com.clps.spring5.aop.annotation.User.add(..))")
    public void commonPointCut() {

    }

    @Before(value = "execution(* com.clps.spring5.aop.annotation.User.add(..))")
    public void before() {
        System.out.println("前置通知: @Before");
    }

    @After(value = "commonPointCut()")
    public void after() {
        System.out.println("后置通知: @After");
    }

    @Around(value = "execution(* com.clps.spring5.aop.annotation.User.add(..))")
    public void around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
        System.out.println("环绕通知: @Around --- Before");
        proceedingJoinPoint.proceed();
        System.out.println("环绕通知: @Around --- After");
    }

    @AfterReturning(value = "execution(* com.clps.spring5.aop.annotation.User.add(..))")
    public void afterReturning() {
        System.out.println("返回通知: @AfterReturning");
    }

    @AfterThrowing(value = "execution(* com.clps.spring5.aop.annotation.User.add(..))")
    public void afterThrowing() {
        System.out.println("异常通知: @AfterThrowing");
    }

}
```







4. 切面 : **@Aspect**

把通知应用到切入点的过程

基于 AspectJ 实现 AOP

Spring 框架一般都是基于 AspectJ 实现 AOP 操作
AspectJ 不是 Spring 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
spring-aspects-5.2.6.RELEASE.jar

导入相关依赖：

- >  com.springsource.net.sf.cglib-2.2.0.jar
- >  com.springsource.org.aopalliance-1.0.0.jar
- >  com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar
- >  commons-logging-1.1.1.jar
- >  spring-aop-5.2.6.RELEASE.jar
- >  spring-aspects-5.2.6.RELEASE.jar
- >  spring-beans-5.2.6.RELEASE.jar
- >  spring-context-5.2.6.RELEASE.jar
- >  spring-core-5.2.6.RELEASE.jar
- >  spring-expression-5.2.6.RELEASE.jar

基于注解方式实现

Step 1：创建被增强类，在类里面定义方法

```
// 被增强类
public class User {

    public void add() {
        System.out.println("User add method is invoked");
    }

}
```

Step 2：创建增强类（代理类），在增强类里面，创建方法，编写增强逻辑，让不同方法代表不同通知类型

```
// 增强类
public class UserProxy {

    public void before() {
        System.out.println("前置通知: @Before");
    }

}
```

Step 3：进行通知的配置

1. 在 spring 配置文件中，开启注解扫描

```
<!-- 开启注解扫描 -->  
<context:component-scan base-package="com.clps.spring5.aop.annotation"></context:component-scan>
```

2. 使用注解创建 User 和 UserProxy 对象

```
// 被增强类  
@Component  
public class User {  
  
    public void add() {  
        System.out.println("User add method is invoked");  
    }  
  
}  
  
// 增强类  
@Component  
public class UserProxy {  
  
    public void before() {  
        System.out.println("前置通知: @Before");  
    }  
  
}
```

3. 在增强类上面添加注解 @Aspect

```
// 增强类  
@Component  
@Aspect    // 生成代理对象  
public class UserProxy {
```

4. 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>
```

Step 4：配置不同类型的通知，在增强类里面，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

```
// 增强类
@Component
@Aspect    // 生成代理对象
public class UserProxy {

    @Pointcut(value = "execution(* com.clps.spring5.aop.annotation.User.add(..))")
    public void commonPointCut() {

    }

    @Before(value = "execution(* com.clps.spring5.aop.annotation.User.add(..))")
    public void before() {
        System.out.println("前置通知: @Before");
    }

    @After(value = "commonPointCut()")
    public void after() {
        System.out.println("后置通知: @After");
    }

    @Around(value = "execution(* com.clps.spring5.aop.annotation.User.add(..))")
    public void around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
        System.out.println("环绕通知: @Around --- Before");
        proceedingJoinPoint.proceed();
        System.out.println("环绕通知: @Around --- After");
    }

    @AfterReturning(value = "execution(* com.clps.spring5.aop.annotation.User.add(..))")
    public void afterReturning() {
        System.out.println("返回通知: @AfterReturning");
    }

    @AfterThrowing(value = "execution(* com.clps.spring5.aop.annotation.User.add(..))")
    public void afterThrowing() {
        System.out.println("异常通知: @AfterThrowing");
    }

}
```

Step 5：抽取相同的切入点

```
@Pointcut(value = "execution(* com.clps.spring5.aop.annotation.User.add(..))")
public void commonPointCut() {

}

}
```

```

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

@After(value = "commonPointCut()")
public void after() {
    System.out.println("后置通知: @After");
}

```

Step 6：设置多个增强类的优先级

在增强类上面添加注解 @Order(数字)，数字类型值越小优先级越高

```

// 增强类
@Component
@Aspect    // 生成代理对象
@Order(2)
public class UserProxy {

// 增强类
@Component
@Aspect    // 生成代理对象
@Order(1)
public class UserProxy2 {

```

完全使用注解开发

1. 创建配置类，替代 xml 配置文件

```

@Configuration
@ComponentScan(basePackages = { "com.clps.spring5.aop.annotation" })
@EnableAspectJAutoProxy(proxyTargetClass = true)
public class ApplicationContextConfig {

}

```

2. 加载配置类


```
ApplicationContext context = new AnnotationConfigApplicationContext(ApplicationContextConfig.class);

@Test
public void testUser() {
    User user = context.getBean("user", User.class);
    user.add();
}
```

基于 xml 配置文件方式实现
