

Spring6

一、Spring启示录

1.1 OCP开闭原则

1.2 依赖倒置原则DIP

1.3 控制反转IoC

二、Spring概述

2.1 Spring简介

2.2 Spring8大模块

2.3 Spring特点

2.4 本教程软件版本

三、Spring的入门程序

3.1 Spring的下载

3.2 Spring的jar文件

3.3 第一个Spring程序

3.4 第一个Spring程序详细剖析

3.5 Spring6启用Log4j2日志框架

四、Spring对IoC的实现

4.1 IoC 控制反转

4.2 依赖注入

4.2.1 set注入

4.2.2 构造注入

4.3 set注入专题

4.3.1 注入外部Bean

4.3.2 注入内部Bean

4.3.3 注入简单类型

4.3.4 级联属性赋值（了解）

4.3.5 注入数组

4.3.6 注入List集合

4.3.7 注入Set集合

4.3.8 注入Map集合

4.3.9 注入Properties

4.3.10 注入null和空字符串

4.3.11 注入的值中含有特殊符号

4.4 p命名空间注入

4.5 c命名空间注入

4.6 util命名空间

4.7 基于XML的自动装配

4.7.1 根据名称自动装配

4.7.2 根据类型自动装配

4.8 spring引入外部属性配置文件

五、Bean的作用域

5.1 singleton

5.2 prototype

5.3 其它scope

六、GoF之工厂模式

6.1 工厂模式的三种形态

6.2 简单工厂模式

6.3 工厂方法模式

6.4 抽象工厂模式（了解）

七、Bean的实例化方式

7.1 通过构造方法实例化

7.2 通过简单工厂模式实例化

7.3 通过factory-bean实例化

7.4 通过FactoryBean接口实例化

7.5 BeanFactory和FactoryBean的区别

7.5.1 BeanFactory

7.5.2 FactoryBean

7.6 注入自定义Date

八、Bean的生命周期

8.1 什么是Bean的生命周期

8.2 为什么要知道Bean的生命周期

8.3 Bean的生命周期之5步

8.4 Bean生命周期之7步

8.5 Bean生命周期之10步

8.6 Bean的作用域不同，管理方式不同

8.7 自己new的对象如何让Spring管理

九、Bean的循环依赖问题

9.1 什么是Bean的循环依赖

9.2 singleton下的set注入产生的循环依赖

9.3 prototype下的set注入产生的循环依赖

9.4 singleton下的构造注入产生的循环依赖

9.5 Spring解决循环依赖的机理

十、回顾反射机制

10.1 分析方法四要素

10.2 获取Method

10.3 调用Method

10.4 假设你知道属性名

十一、手写Spring框架

第一步：创建模块myspring

第二步：准备好我们要管理的Bean

第三步：准备myspring.xml配置文件

第四步：编写ApplicationContext接口

第五步：编写ClassPathXmlApplicationContext

第六步：确定采用Map集合存储Bean

第七步：解析配置文件实例化所有Bean

第八步：测试能否获取到Bean

第九步：给Bean的属性赋值

第十步：打包发布

第十一步：站在程序员角度使用myspring框架

十二、Spring IoC注解式开发

12.1 回顾注解

12.2 声明Bean的注解

12.3 Spring注解的使用

12.4 选择性实例化Bean

12.5 负责注入的注解

12.5.1 @Value

12.5.2 @Autowired与@Qualifier

12.5.3 @Resource

12.6 全注解式开发

十三、JdbcTemplate

13.1 环境准备

13.2 新增

13.3 修改

13.4 删除

13.5 查询一个对象

13.6 查询多个对象

13.7 查询一个值

13.8 批量添加

13.9 批量修改

13.10 批量删除

13.11 使用回调函数

13.12 使用德鲁伊连接池

十四、GoF之代理模式

14.1 对代理模式的理解

14.2 静态代理

14.3 动态代理

14.3.1 JDK动态代理

14.3.2 CGLIB动态代理

十五、面向切面编程AOP

15.1 AOP介绍

15.2 AOP的七大术语

15.3 切点表达式

15.4 使用Spring的AOP

15.4.1 准备工作

15.4.2 基于AspectJ的AOP注解式开发

实现步骤

通知类型

切面的先后顺序

优化使用切点表达式

全注解式开发AOP

15.4.3 基于XML配置方式的AOP（了解）

15.5 AOP的实际案例：事务处理

15.6 AOP的实际案例：安全日志

十六、Spring对事务的支持

16.1 事务概述

16.2 引入事务场景

第一步：准备数据库表

第二步：创建包结构

第三步：准备POJO类

第四步：编写持久层

第五步：编写业务层

第六步：编写Spring配置文件

第七步：编写表示层（测试程序）

模拟异常

16.3 Spring对事务的支持

Spring实现事务的两种方式

Spring事务管理API

声明式事务之注解实现方式

事务属性

事务属性包括哪些

事务传播行为

事务隔离级别

事务超时

只读事务

设置哪些异常回滚事务

设置哪些异常不回滚事务

事务的全注解式开发

声明式事务之XML实现方式

十七、Spring6整合JUnit5

17.1 Spring对JUnit4的支持

17.2 Spring对JUnit5的支持

十八、Spring6集成MyBatis3.5

18.1 实现步骤

18.2 具体实现

18.3 spring配置文件的import

十九、Spring中的八大模式

19.1 简单工厂模式

19.2 工厂方法模式

19.3 单例模式

19.4 代理模式

19.5 装饰器模式

19.6 观察者模式

19.7 策略模式

19.8 模板方法模式

一、Spring启示录

阅读以下代码：

▼ UserController

Java | 复制代码

```
1 package com.powernode.oa.controller;
2
3 import com.powernode.oa.service.UserService;
4 import com.powernode.oa.service.impl.UserServiceImpl;
5
6 public class UserController {
7
8     private UserService userService = new UserServiceImpl();
9
10    public void login(){
11        String username = "admin";
12        String password = "123456";
13        boolean success = userService.login(username, password);
14        if (success) {
15            // 登录成功
16        } else {
17            // 登录失败
18        }
19    }
20}
21
```

动力节点

动力节点

动力节点

动力节点

动力节点

动力节点

UserServiceImpl

Java | 复制代码

```
1 package com.powernode.oa.service.impl;
2
3 import com.powernode.oa.bean.User;
4 import com.powernode.oa.dao.UserDao;
5 import com.powernode.oa.dao.impl.UserDaoImplForMySQL;
6 import com.powernode.oa.service.UserService;
7
8 public class UserServiceImpl implements UserService {
9
10     private UserDao userDao = new UserDaoImplForMySQL();
11
12     public boolean login(String username, String password) {
13         User user = userDao.selectByUsernameAndPassword(username, password);
14         if (user != null) {
15             return true;
16         }
17         return false;
18     }
19 }
20
```

UserDaoImplForMySQL

Java | 复制代码

```
1 package com.powernode.oa.dao.impl;
2
3 import com.powernode.oa.bean.User;
4 import com.powernode.oa.dao.UserDao;
5
6 public class UserDaoImplForMySQL implements UserDao {
7     public User selectByUsernameAndPassword(String username, String password) {
8         // 连接MySQL数据库, 根据用户名和密码查询用户信息
9         return null;
10    }
11 }
```

可以看出, UserDaoImplForMySQL中主要是连接MySQL数据库进行操作。如果更换到Oracle数据库上, 则需要再提供一个UserDaoImplForOracle, 如下:

UserDaoImplForOracle

Java | 复制代码

```

1 package com.powernode.oa.dao.impl;
2
3 import com.powernode.oa.bean.User;
4 import com.powernode.oa.dao.UserDao;
5
6 public class UserDaoImplForOracle implements UserDao {
7     public User selectByUsernameAndPassword(String username, String password) {
8         // 连接Oracle数据库, 根据用户名和密码查询用户信息
9         return null;
10    }
11 }
12

```

很明显，以上的操作正在进行功能的扩展，添加了一个新的类UserDaoImplForOracle来应付数据库的变化，这里的变化会引起连锁反应吗？当然会，如果想要切换到Oracle数据库上，UserServiceImpl类代码就需要修改，如下：

UserServiceImpl

Java | 复制代码

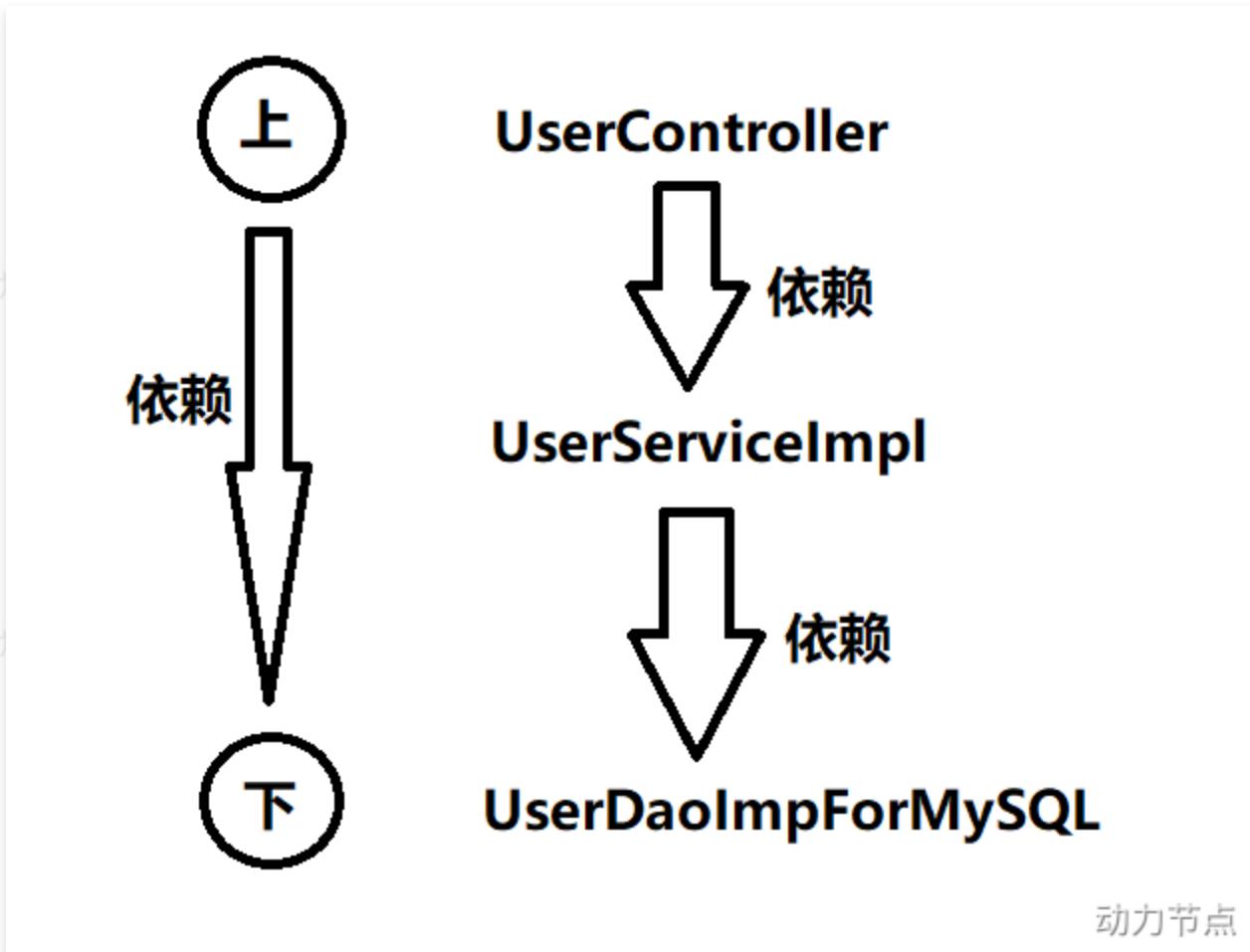
```

1 package com.powernode.oa.service.impl;
2
3 import com.powernode.oa.bean.User;
4 import com.powernode.oa.dao.UserDao;
5 import com.powernode.oa.dao.impl.UserDaoImplForOracle;
6 import com.powernode.oa.service.UserService;
7
8 public class UserServiceImpl implements UserService {
9
10    //private UserDao userDao = new UserDaoImplForMySQL();
11    private UserDao userDao = new UserDaoImplForOracle();
12
13    public boolean login(String username, String password) {
14        User user = userDao.selectByUsernameAndPassword(username, password);
15        if (user != null) {
16            return true;
17        }
18        return false;
19    }
20 }
21

```

1.1 OCP开闭原则

这样一来就违背了开闭原则OCP。开闭原则是这样说的：在软件开发过程中应当对扩展开放，对修改关闭。也就是说，如果在进行功能扩展的时候，添加额外的类是没问题的，但因为功能扩展而修改之前运行正常的程序，这是忌讳的，不被允许的。因为一旦修改之前运行正常的程序，就会导致项目整体要进行全方位的重新测试。这是相当麻烦的过程。导致以上问题的主要原因是：代码和代码之间的耦合度太高。如下图所示：



可以很明显的看出，**上层**是依赖**下层**的。UserController依赖UserServiceImpl，而UserServiceImpl依赖UserDaoImplForMySQL，这样就会导致**下面只要改动，上面必然会受牵连（跟着也会改）**，所谓牵一发而动全身。这样也就同时违背了另一个开发原则：依赖倒置原则。

1.2 依赖倒置原则DIP

依赖倒置原则(Dependence Inversion Principle)，简称DIP，主要倡导面向抽象编程，面向接口编程，不要面向具体编程，让**上层不再依赖下层**，下面改动了，上面的代码不会受到牵连。这样可以大大降低程序的耦合度，耦合度低了，扩展力就强了，同时代码复用性也会增强。（**软件七大开发原则都是在为解耦合服务**）

你可能会说，上面的代码已经面向接口编程了呀：

```
2 usages
public class UserServiceImpl implements UserService {

    //private UserDao userDao = new UserDaoImplForMySQL();
    1 usage          接口
    private UserDao userDao = new UserDaoImplForOracle();

    1 usage
    public boolean login(String username, String password) {
        User user = userDao.selectByUsernameAndPassword(username, password);
        if (user != null) {
            return true;
        }
        return false;
    }
}
```

动力节点

确实已经面向接口编程了，但对象的创建是：new UserDaoImplForOracle()显然并没有完全面向接口编程，还是使用到了具体的接口实现类。什么叫做完全面向接口编程？什么叫做完全符合依赖倒置原则呢？请看以下代码：

```
2 usages
public class UserServiceImpl implements UserService {

    //private UserDao userDao = new UserDaoImplForMySQL();
    //private UserDao userDao = new UserDaoImplForOracle();

    1 usage
    private UserDao userDao;

    1 usage
    public boolean login(String username, String password) {
        User user = userDao.selectByUsernameAndPassword(username, password);
        if (user != null) {
            return true;
        }
        return false;
    }
}
```

动力节点

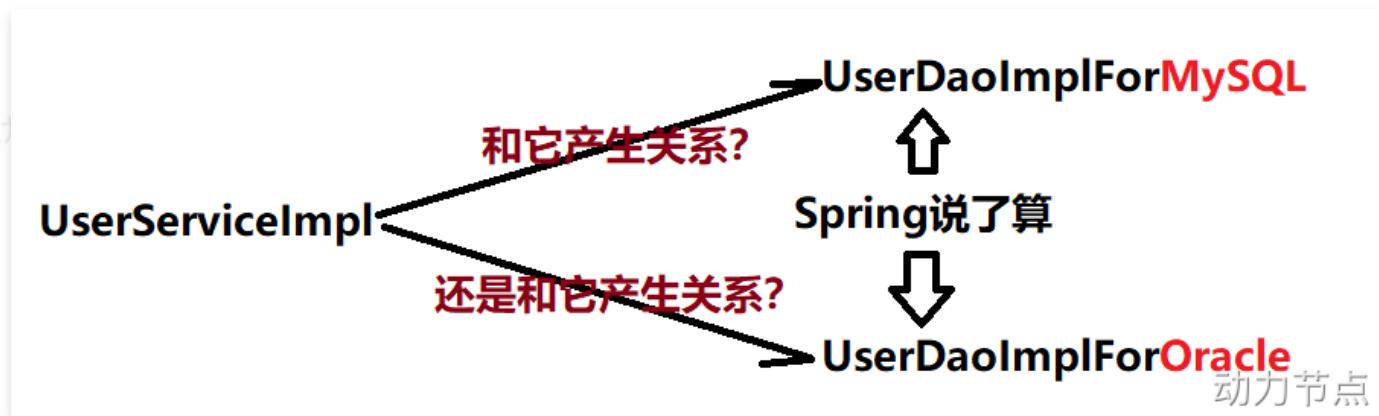
如果代码是这样编写的，才算是完全面向接口编程，才符合依赖倒置原则。那你可能会问，这样userDao是null，在执行的时候就会出现空指针异常呀。你说的有道理，确实是这样的，所以我们要解决这个问题。解决空指针异常的问题，其实就是解决两个核心的问题：

- 第一个问题：谁来负责对象的创建。【也就是说谁来：`new UserDaoImplForOracle()`/`new UserDaoImplForMySQL()`】
- 第二个问题：谁来负责把创建的对象赋到这个属性上。【也就是说谁来把上面创建的对象赋给`userDao`属性】

如果我们把以上两个核心问题解决了，就可以做到既符合OCP开闭原则，又符合依赖倒置原则。

很荣幸的通知你：Spring框架可以做到。

在Spring框架中，它可以帮助我们new对象，并且它还可以将new出来的对象赋到属性上。换句话说，Spring框架可以帮助我们创建对象，并且可以帮助我们维护对象和对象之间的关系。比如：



Spring可以new出来`UserDaoImplForMySQL`对象，也可以new出来`UserDaoImplForOracle`对象，并且还可以让new出来的dao对象和服务对象产生关系（产生关系其实本质上就是给属性赋值）。

很显然，这种方式是将对象的创建权/管理权交出去了，不再使用硬编码的方式了。同时也把对象关系的管理权交出去了，也不再使用硬编码的方式了。像这种把对象的创建权交出去，把对象关系的管理权交出去，被称为控制反转。

1.3 控制反转IoC

控制反转（Inversion of Control，缩写为IoC），是面向对象编程中的一种设计思想，可以用来降低代码之间的耦合度，符合依赖倒置原则。

控制反转的核心是：**将对象的创建权交出去，将对象和对象之间关系的管理权交出去，由第三方容器来负责创建与维护。**

控制反转常见的实现方式：依赖注入（Dependency Injection，简称DI）

通常，依赖注入的实现由包括两种方式：

- set方法注入
- 构造方法注入

而Spring框架就是一个实现了IoC思想的框架。

IoC可以认为是一种**全新的设计模式**，但是理论和时间成熟相对较晚，并没有包含在GoF中。（GoF指的是23种设计模式）

二、Spring概述

2.1 Spring简介



来自百度百科

Spring是一个开源框架，它由Rod Johnson创建。它是为了解决企业应用开发的复杂性而创建的。

从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。

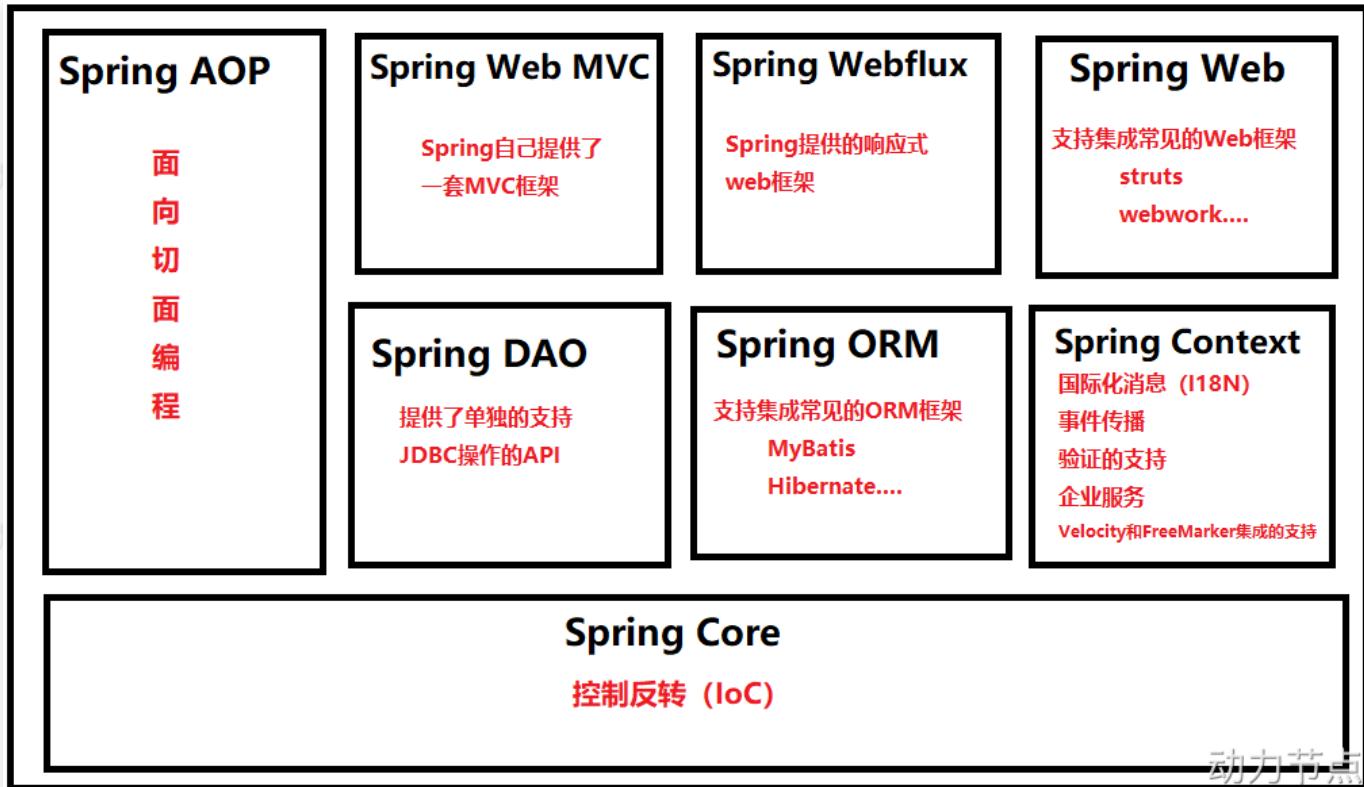
Spring是一个轻量级的控制反转(IoC)和面向切面(AOP)的容器框架。

Spring最初的出现是为了解决EJB臃肿的设计，以及难以测试等问题。

Spring为简化开发而生，让程序员只需关注核心业务的实现，尽可能的不再关注非业务逻辑代码（事务控制，安全日志等）。

2.2 Spring8大模块

注意：Spring5版本之后是8个模块。在Spring5中新增了WebFlux模块。



1. Spring Core模块

这是Spring框架最基础的部分，它提供了依赖注入（DependencyInjection）特征来实现容器对Bean的管理。核心容器的主要组件是 BeanFactory，BeanFactory是工厂模式的一个实现，是任何Spring应用的核心。它使用IoC将应用配置和依赖从实际的应用代码中分离出来。

2. Spring Context模块

如果说核心模块中的BeanFactory使Spring成为容器的话，那么上下文模块就是Spring成为框架的原因。这个模块扩展了BeanFactory，增加了对国际化（I18N）消息、事件传播、验证的支持。另外提供了许多企业服务，例如电子邮件、JNDI访问、EJB集成、远程以及调度（scheduling）服务。也包括了对模版框架例如Velocity和FreeMarker集成的支持。

3. Spring AOP模块

Spring在它的AOP模块中提供了对面向切面编程的丰富支持，Spring AOP模块为基于Spring的应用程序中的对象提供了事务管理服务。通过使用Spring AOP，不用依赖组件，就可以将声明性事务管理集成到应用程序中，可以自定义拦截器、切点、日志等操作。

4. Spring DAO模块

提供了一个JDBC的抽象层和异常层次结构，消除了烦琐的JDBC编码和数据库厂商特有的错误代码解析，用于简化JDBC。

5. Spring ORM模块

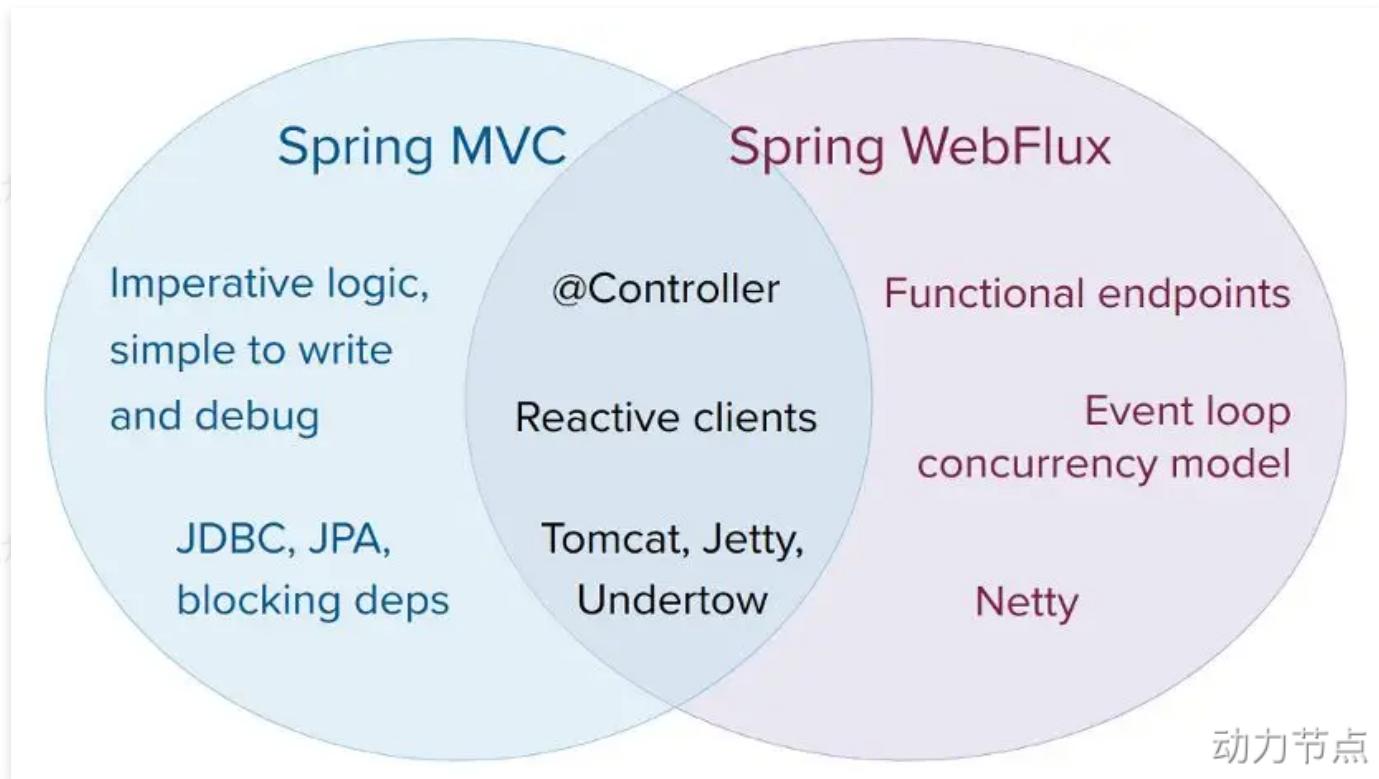
Spring提供了ORM模块。Spring并不试图实现它自己的ORM解决方案，而是为几种流行的ORM框架提供了集成方案，包括Hibernate、JDO和iBATIS SQL映射，这些都遵从 Spring 的通用事务和 DAO 异常层次结构。

6. Spring Web MVC模块

Spring为构建Web应用提供了一个功能全面的MVC框架。虽然Spring可以很容易地与其它MVC框架集成，例如Struts，但Spring的MVC框架使用IoC对控制逻辑和业务对象提供了完全的分离。

7. Spring WebFlux模块

Spring Framework 中包含的原始 Web 框架 Spring Web MVC 是专门为 Servlet API 和 Servlet 容器构建的。反应式堆栈 Web 框架 Spring WebFlux 是在 5.0 版的后期添加的。它是完全非阻塞的，支持反应式流(Reactive Stream)背压，并在Netty, Undertow和Servlet 3.1+容器等服务器上运行。



8. Spring Web模块

Web上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文，提供了 Spring和其他Web框架的集成，比如Struts、WebWork。还提供了一些面向服务支持，例如：实现文件上传的multipart请求。

2.3 Spring特点

1. 轻量

- a. 从大小与开销两方面而言Spring都是轻量的。完整的Spring框架可以在一个大小只有1MB多的

JAR文件里发布。并且Spring所需的处理开销也是微不足道的。

- b. Spring是非侵入式的：Spring应用中的对象不依赖于Spring的特定类。

2. 控制反转

- a. Spring通过一种称作控制反转（IoC）的技术促进了松耦合。当应用了IoC，一个对象依赖的其它对象会通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。你可以认为IoC与JNDI相反——不是对象从容器中查找依赖，而是容器在对象初始化时不等对象请求就主动将依赖传递给它。

3. 面向切面

- a. Spring提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务（例如审计（auditing）和事务（transaction）管理）进行内聚性的开发。应用对象只实现它们应该做的——完成业务逻辑——仅此而已。它们并不负责（甚至是意识）其它的系统级关注点，例如日志或事务支持。

4. 容器

- a. Spring包含并管理应用对象的配置和生命周期，在这个意义上它是一种容器，你可以配置你的每个bean如何被创建——基于一个可配置原型（prototype），你的bean可以创建一个单独的实例或者每次需要时都生成一个新的实例——以及它们是如何相互关联的。然而，Spring不应该被混同于传统的重量级的EJB容器，它们经常是庞大与笨重的，难以使用。

5. 框架

- a. Spring可以将简单的组件配置、组合成为复杂的应用。在Spring中，应用对象被声明式地组合，典型地是在一个XML文件里。Spring也提供了很多基础功能（事务管理、持久化框架集成等等），将应用逻辑的开发留给了你。

所有Spring的这些特征使你能够编写更干净、更可管理、并且更易于测试的代码。它们也为Spring中的各种模块提供了基础支持。

2.4 本教程软件版本

- IDEA工具：2022.1.4
- JDK：Java17 (**Spring6要求JDK最低版本是Java17**)
- Maven：3.8.6
- Spring：6.0.0-M2
- JUnit：4.13.2

三、Spring的入门程序

3.1 Spring的下载

官网地址: <https://spring.io/>

The screenshot shows the official English Spring website at <https://spring.io/>. A red box highlights the 'Projects' dropdown menu in the top navigation bar. A larger red box highlights the 'Spring Framework' option within the dropdown menu. The main content area features a large green and yellow graphic on the left and a central section with the text 'Spring makes cloud native easy'. Below this are sections for 'What Spring can do' with icons for Microservices, Reactive, Cloud, and Web apps.

官网地址（中文）：<http://spring.p2hp.com/>

The screenshot shows the official Chinese Spring website at <http://spring.p2hp.com/>. A red box highlights the 'Projects' dropdown menu in the top navigation bar. A larger red box highlights the 'Spring Framework' option within the dropdown menu. The main content area features a large green and yellow graphic on the left and a central section with the text 'Spring 使应用具备反应性'. Below this are sections for 'Spring 能做什么' with icons for 微服务, 反应性, 云, and Web 应用.

打开Spring官网后，可以看到Spring Framework，以及通过Spring Framework衍生的其它框架：

The screenshot shows the official Spring Framework website. At the top left is the Spring logo. At the top right are links for "Why Spring", "Learn", and "Pro". Below the logo is a navigation bar with the following items: "Spring Boot", "Spring Framework" (which is highlighted with a red border and has a red arrow pointing to it from the text "这是Spring框架, 其他的框架都是基于Spring框架扩展的."), "Spring Data", "Spring Cloud", "Spring Cloud Data Flow", "Spring Security", "Spring for GraphQL", "Spring Session", "Spring Integration", "Spring HATEOAS", "Spring REST Docs", "Spring Batch", "Spring AMQP", "Spring CredHub", "Spring Flo", "Spring for Apache Kafka", "Spring LDAP", "Spring Shell", "Spring StateMachine", "Spring Vault", "Spring Web Flow", and "Spring Web Services". To the right of the navigation bar is a large "Spring Framework" title. Below it is a navigation bar with "OVERVIEW" (highlighted with a green bar), "LEARN", and "SUPPORT". The main content area starts with a paragraph about the Spring Framework providing a comprehensive infrastructure for enterprise applications. It then lists several key components: "plumbing" of enterprise applications so that they can be deployed to any environment without unnecessary ties to specific deployment environments. Below this is a section titled "Support Policy and Migration" with a link to "official Spring Framework support policies". The final section is "Features" with a bulleted list of core technologies, testing, data access, web frameworks, integration, and languages.

这是Spring框架, 其他的框架都是基于Spring框架扩展的。

Support Policy and Migration

For information about minimum requirement policies, please check out the official Spring Framework support policies.

Features

- Core technologies: dependency injection, conversion, SpEL, AOP.
- Testing: mock objects, TestContext framework.
- Data Access: transactions, DAO support, JDBC, JPA, ODBC, JNDI, JMS, JCA, JMX, emulators.
- Spring MVC and Spring WebFlux web frameworks.
- Integration: remoting, JMS, JCA, JMX, emulators.
- Languages: Kotlin, Groovy, dynamic languages.

我们即将要学习的就是Spring Framework。

怎么下载呢?

- 第一步：进入github

The screenshot shows the official Spring Framework website. On the left, there's a sidebar with links to various Spring projects: Spring Boot, Spring Framework (which is highlighted with a black background), Spring Data, Spring Cloud, Spring Cloud Data Flow, Spring Security, Spring for GraphQL, Spring Session, Spring Integration, Spring HATEOAS, Spring REST Docs, and Spring Batch. The main content area has a title 'Spring Framework 5.3.23'. Below it are tabs for 'OVERVIEW' (selected), 'LEARN', and 'SUPPORT'. A section titled 'Documentation' explains that each project has its own documentation. Below this, there are four rows of links for different versions: 5.3.23 (CURRENT, GA), 6.0.0-SNAPSHOT (SNAPSHOT), 6.0.0-M6 (PRE), and 5.3.24-SNAPSHOT (SNAPSHOT). Each row includes a 'Reference Doc.' and 'API Doc.' link. A watermark '动力节点' is visible in the bottom right corner.

- 第二步：找到下图位置，点击超链接

The screenshot shows a GitHub repository page for 'spring-projects/spring-framework'. The URL in the address bar is 'github.com/spring-projects/spring-framework'. The page displays the 'README.md' file. It contains sections for 'Code of Conduct' and 'Access to Binaries'. The 'Access to Binaries' section is highlighted with a red box and a red arrow pointing to the text 'For access to artifacts or a distribution zip, see the [Spring Framework Artifacts](#) wiki page.'

- 第三步：找到下图位置，点击超链接

github.com/spring-projects/spring-framework/wiki/Spring-Framework-Artifacts

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.16</version>
</dependency>
```

Spring Repositories

Snapshot, milestone, and release candidate versions are published to an [Artifactory](#) instance hosted by [JFrog](#). You can use the Web UI at <https://repo.spring.io> to browse the Spring Artifactory, or go directly to one of the repositories listed below.

Snapshots

Add the following to resolve snapshot versions – for example, `5.3.17-SNAPSHOT`:

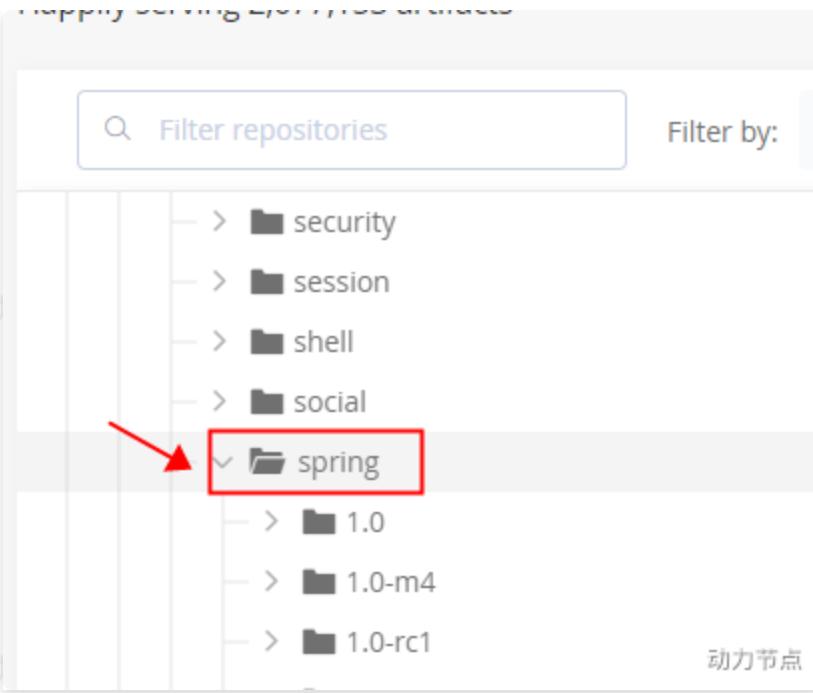
```
<repository>
    <id>repository.spring.snapshot</id>
    <name>Spring Snapshot Repository</name>
    <url>https://repo.spring.io/snapshot</url>
</repository>
```

- 第四步：按照下图步骤操作

repo.spring.io/ui/repos/tree/General/libs-milestone

The screenshot shows the Spring Artifactory interface. On the left, there's a sidebar with icons for Artifactory, Packages, Builds, Artifacts (which is selected and highlighted with a red box), Xray, Distribution, and Pipelines. The main area displays a tree view of repositories. A red box labeled '1' highlights the 'Artifacts' section in the sidebar. A red box labeled '2' highlights the 'release' folder under 'libs-milestone'. A red box labeled '3' highlights the 'org' folder under 'release'. A red box labeled '4' highlights the 'springframework' folder under 'org'. To the right of the tree view, there's a detailed view of the 'springframework' package, showing sections for General (highlighted with a green line), Info, Name:, Package Type, Repository Path, URL to file:, Description:, Package Info, Dependency, and Virtual Repo. The 'Virtual Repo' section contains the text 'Virtual Repo' and '动力节点'.

- 第五步：继续在springframework目录下找下图的spring，点开之后你会看到很多不同的版本



- 第六步：选择对应的版本

Filter repositories Filter by: Package Types Repository Types Sort by: Repository Type

> 5.3.3	5.3.9
> 5.3.4	General Properties
> 5.3.5	Info
> 5.3.6	Name: 5.3.9
> 5.3.7	Repository Path: release/org/springframework/spring/5.3.9/
> 5.3.8	URL to file: https://repo.spring.io/artifactory/release/org/springframework/spring/5.3.9/
> 5.3.9	Created: 14-07-21 07:18:11 +00:00
maven-metadata.xml	Package Information
maven-metadata.xml.sha512	Dependency Declaration
spring-agent	Virtual Repository Associations
spring-aop	Included Repositories
spring-asn	
spring-aspects	
spring-beandoc	
spring-beans	
spring-binding	
spring-context	
spring-context-indexer	
spring-context-support	
spring-core	

- 第七步：点击上图的url

Index of release/org/springframework/spring/5.3.9

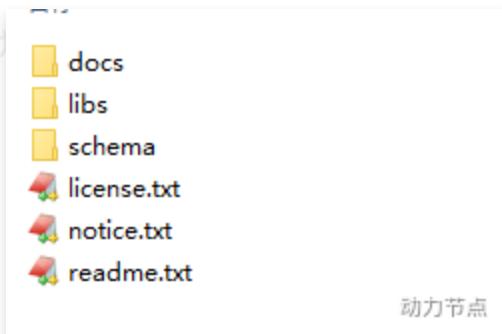
Name	Last Modified
..	
spring-5.3.9-dist.zip	14-07-21 14:49:17 +0800
spring-5.3.9-docs.zip	14-07-21 14:49:15 +0800
spring-5.3.9-schema.zip	14-07-21 14:49:13 +0800
spring-5.3.9.pom	14-07-21 14:49:06 +0800
spring-5.3.9.pom.asc	14-07-21 15:17:35 +0800

Artifactory Online Server at localhost Port 8081

动力节点

点击spring-5.3.9-dist.zip下载spring框架。

将下载的zip包解压：



动力节点

docs: spring框架的API帮助文档

libs: spring框架的jar文件（用spring框架就是用这些jar包）

schema: spring框架的XML配置文件相关的约束文件

3.2 Spring的jar文件

打开libs目录，会看到很多jar包：

spring-core-5.3.9.jar	202
spring-core-5.3.9-javadoc.jar	202
spring-core-5.3.9-sources.jar	202

动力节点

spring-core-5.3.9.jar: 字节码（这个是支撑程序运行的jar包）

spring-core-5.3.9-javadoc.jar: 代码中的注释

spring-core-5.3.9-sources.jar: 源码

我们来看一下spring框架都有哪些jar包：

spring-aop-5.3.9.jar
 spring-aspects-5.3.9.jar
 spring-beans-5.3.9.jar
 spring-context-5.3.9.jar
 spring-context-indexer-5.3.9.jar
 spring-context-support-5.3.9.jar
 spring-core-5.3.9.jar
 spring-expression-5.3.9.jar
 spring-instrument-5.3.9.jar
 spring-jcl-5.3.9.jar
 spring-jdbc-5.3.9.jar
 spring-jms-5.3.9.jar
 spring-messaging-5.3.9.jar
 spring-orm-5.3.9.jar
 spring-oxm-5.3.9.jar
 spring-r2dbc-5.3.9.jar
 spring-test-5.3.9.jar
 spring-tx-5.3.9.jar
 spring-web-5.3.9.jar
 spring-webflux-5.3.9.jar
 spring-webmvc-5.3.9.jar
 spring-websocket-5.3.9.jar

动力节点

JAR文件	描述
spring-aop-5.3.9.jar	这个jar 文件包含在应用中使用Spring 的AOP 特性时所需的类
spring-aspects-5.3.9.jar	提供对AspectJ的支持，以便可以方便的将面向切面的功能集成进IDE中
spring-beans-5.3.9.jar	这个jar 文件是所有应用都要用到的，它包含访问配置文件、创建和管理bean 以及进行Inversion ofControl / Dependency Injection (IoC/DI) 操作相关的所有类。如果应用只需基本的IoC/DI 支持，引入spring-core.jar 及spring-beans.jar 文件就可以了。
spring-context-5.3.9.jar	这个jar 文件为Spring 核心提供了大量扩展。可以找到使用Spring ApplicationContext特性时所需的全部类，JDNI 所需的全部类， instrumentation组件以及校验Validation 方面的相关类。

spring-context-indexer-5.3.9.jar	虽然类路径扫描非常快，但是Spring内部存在大量的类，添加此依赖，可以通过在编译时创建候选对象的静态列表来提高大型应用程序的启动性能。
spring-context-support-5.3.9.jar	用来提供Spring上下文的一些扩展模块,例如实现邮件服务、视图解析、缓存、定时任务调度等
spring-core-5.3.9.jar	Spring 框架基本的核心工具类。Spring 其它组件要都要使用到这个包里的类，是其它组件的基本核心，当然你也可以在自己的应用系统中使用这些工具类。
spring-expression-5.3.9.jar	Spring表达式语言。
spring-instrument-5.3.9.jar	Spring3.0对服务器的代理接口。
spring-jcl-5.3.9.jar	Spring的日志模块。JCL，全称为"Jakarta Commons Logging"，也可称为"Apache Commons Logging"。
spring-jdbc-5.3.9.jar	Spring对JDBC的支持。
spring-jms-5.3.9.jar	这个jar包提供了对JMS 1.0.2/1.1的支持类。JMS是Java消息服务。属于JavaEE规范之一。
spring-messaging-5.3.9.jar	为集成messaging api和消息协议提供支持
spring-orm-5.3.9.jar	Spring集成ORM框架的支持，比如集成 hibernate， mybatis等。
spring-oxm-5.3.9.jar	为主流O/X Mapping组件提供了统一层抽象和封装， OXM是Object Xml Mapping。对象和XML之间的相互转换。
spring-r2dbc-5.3.9.jar	Reactive Relational Database Connectivity (关系型数据库的响应式连接) 的缩写。这个jar文件是Spring对r2dbc的支持。
spring-test-5.3.9.jar	对JUnit等测试框架的简单封装。
spring-tx-5.3.9.jar	为JDBC、Hibernate、JDO、JPA、Beans等提供的一致的声明式和编程式事务管理支持。

spring-web-5.3.9.jar	Spring集成MVC框架的支持，比如集成Struts等。
spring-webflux-5.3.9.jar	WebFlux是 Spring5 添加的新模块，用于 web 的开发，功能和 SpringMVC 类似的，Webflux 使用当前一种比较流程响应式编程出现的框架。
spring-webmvc-5.3.9.jar	SpringMVC框架的类库
spring-websocket-5.3.9.jar	Spring集成WebSocket框架时使用

注意：

如果你只是想用Spring的IoC功能，仅需要引入：spring-context即可。将这个jar包添加到classpath当中。

如果采用maven只需要引入context的依赖即可。

```

▼ spring bean依赖
    XML | 复制代码

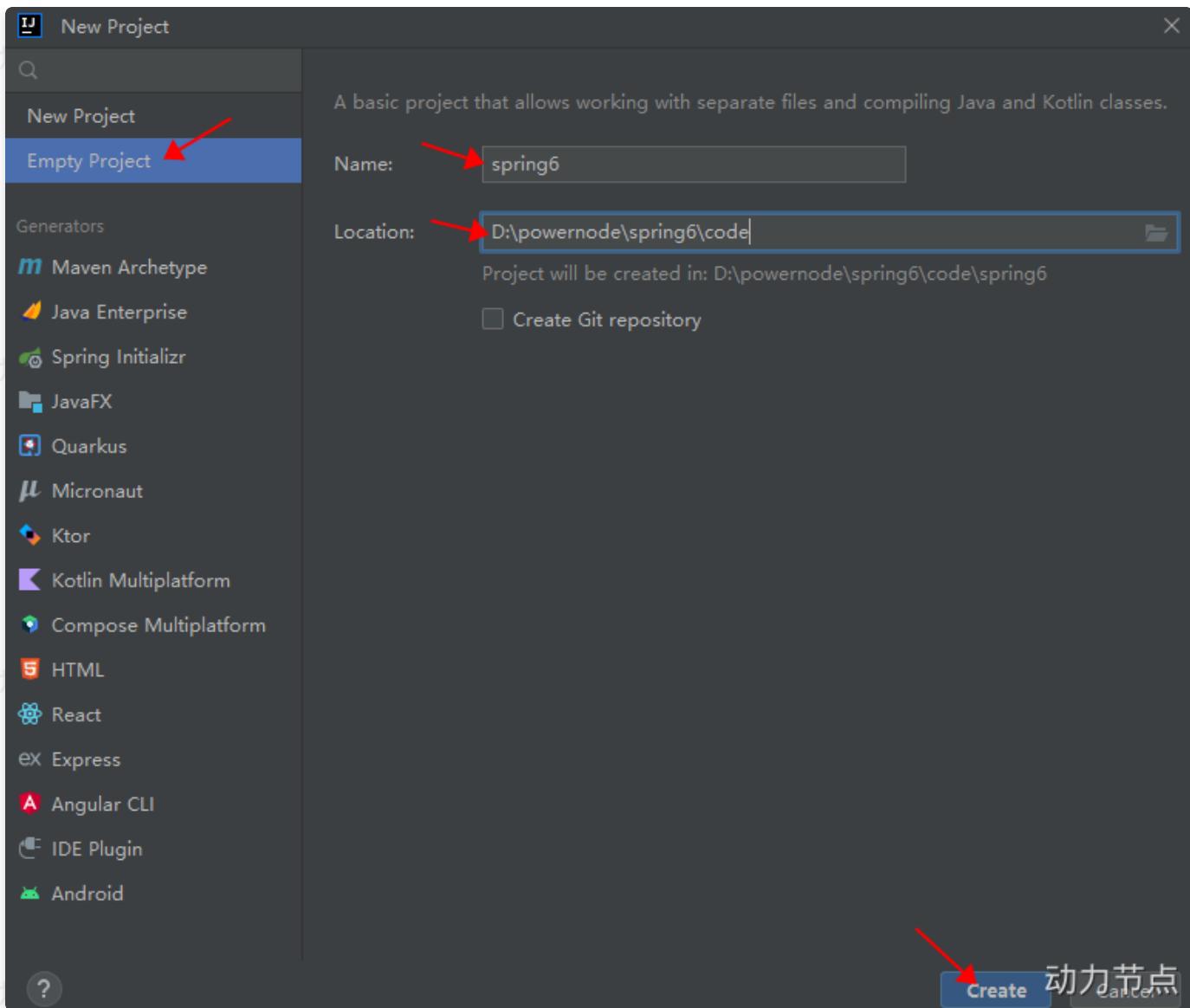
1 <!--Spring6的正式版发布之前，这个仓库地址是需要的-->
2 <repositories>
3   <repository>
4     <id>repository.spring.milestone</id>
5     <name>Spring Milestone Repository</name>
6     <url>https://repo.spring.io/milestone</url>
7   </repository>
8 </repositories>
9
10 <dependencies>
11   <!--spring context依赖：使用的是6.0.0-M2里程碑版-->
12   <dependency>
13     <groupId>org.springframework</groupId>
14     <artifactId>spring-context</artifactId>
15     <version>6.0.0-M2</version>
16   </dependency>
17 </dependencies>

```

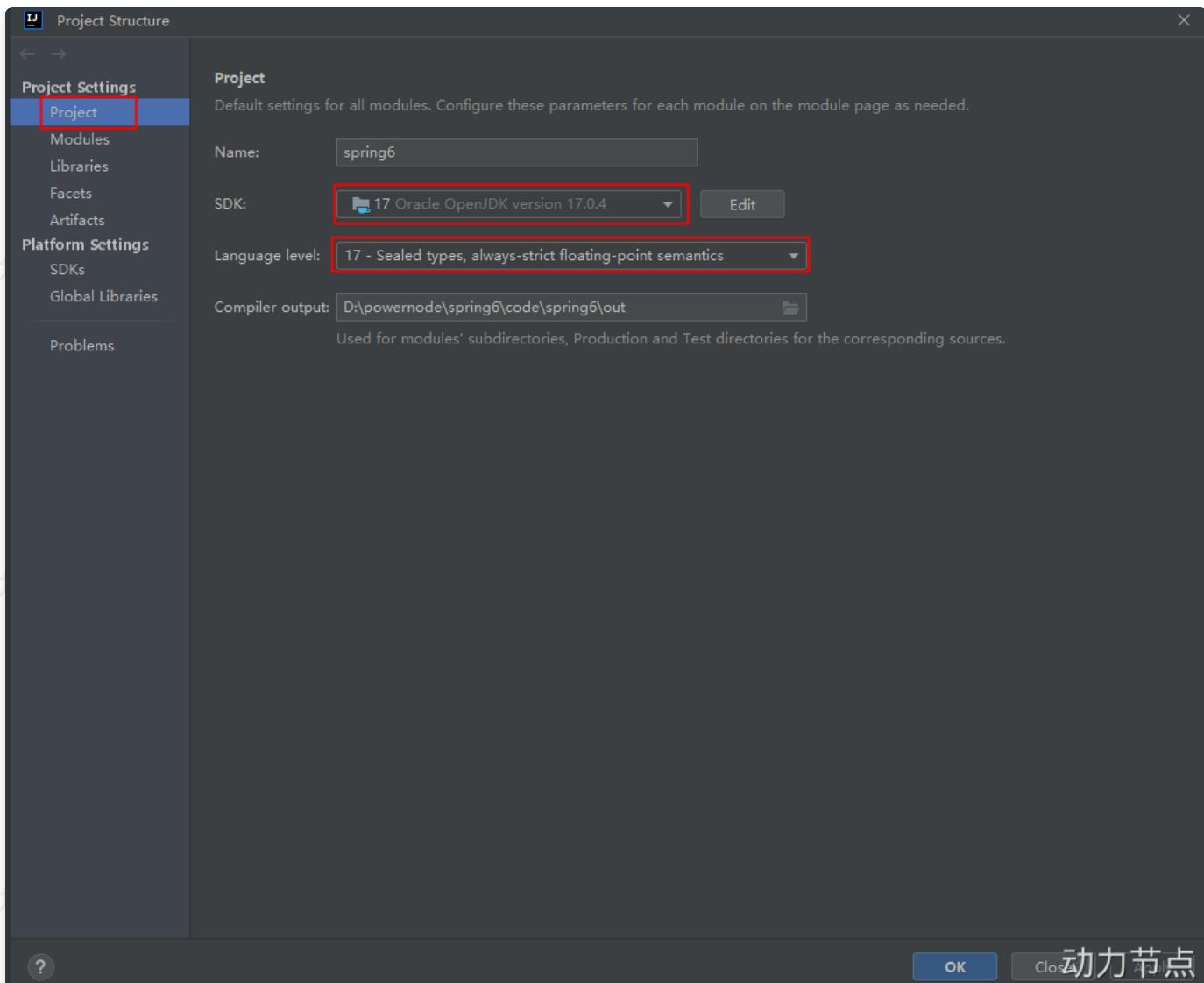
3.3 第一个Spring程序

前期准备：

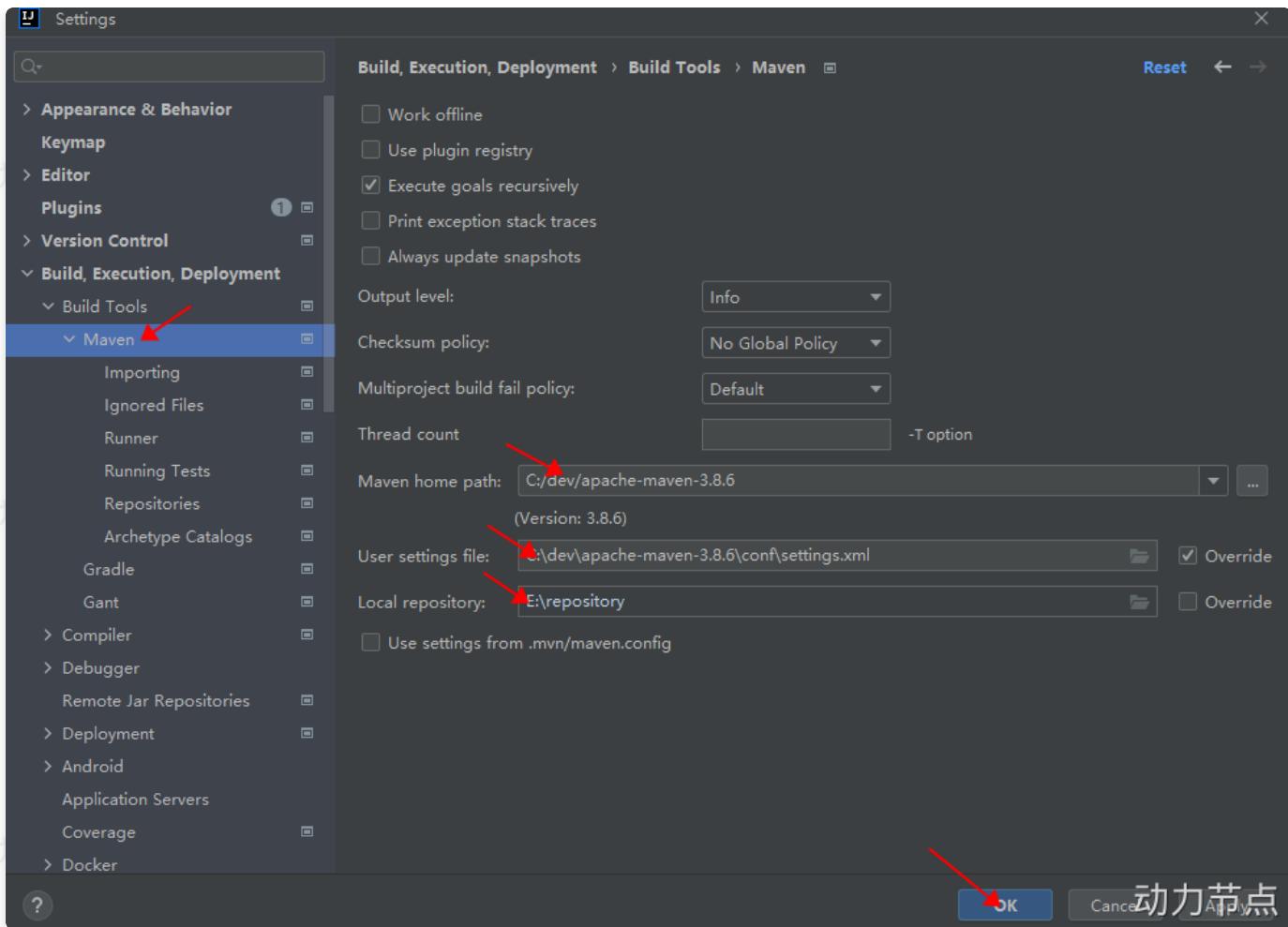
- 打开IDEA创建Empty Project: spring6



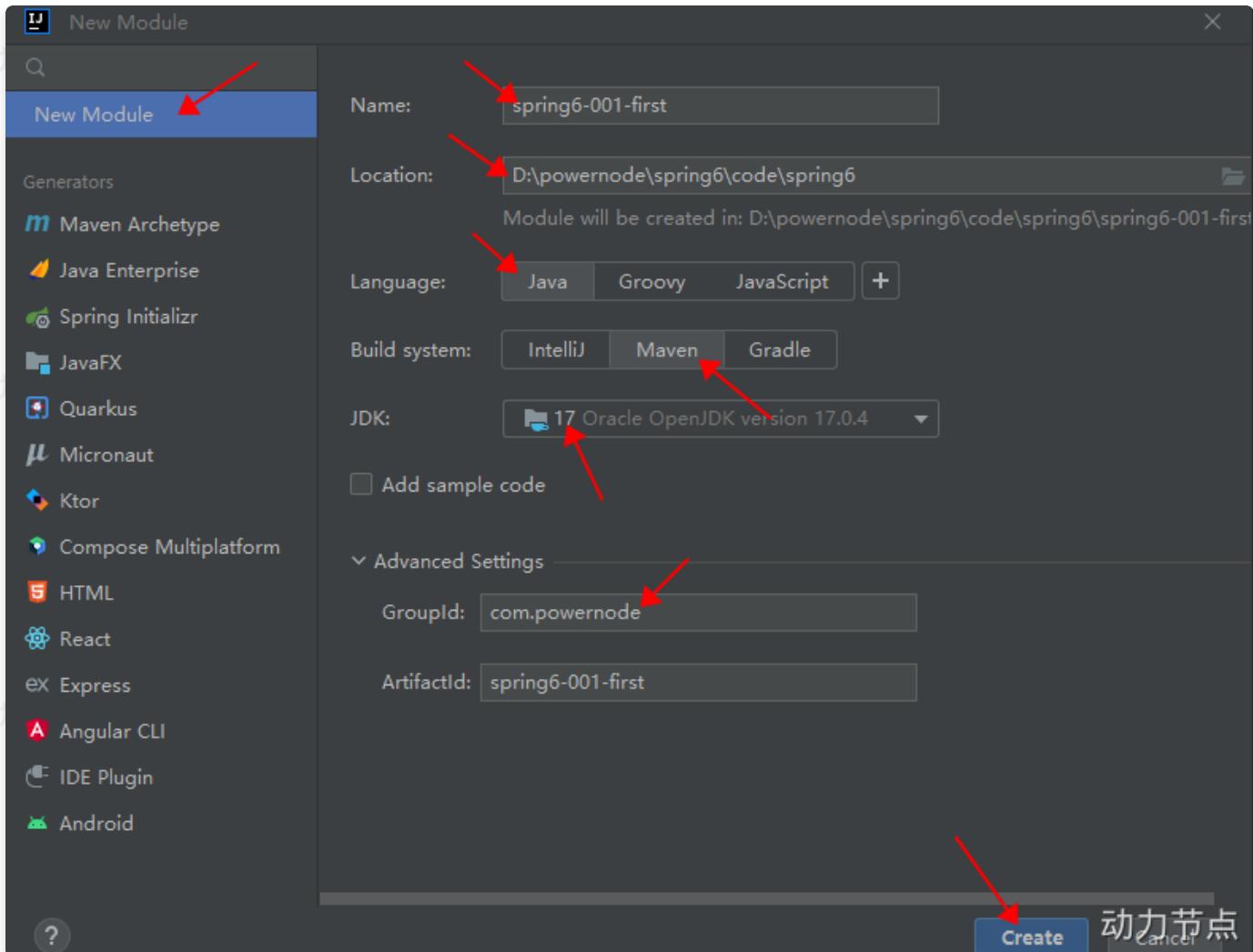
- 设置JDK版本17, 编译器版本17



- 设置IDEA的Maven：关联自己的maven



- 在空的工程spring6中创建第一个模块：spring6-001-first



第一步：添加spring context的依赖，pom.xml配置如下

pom.xml

XML | 复制代码

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>com.powernode</groupId>
8     <artifactId>spring6-001-first</artifactId>
9     <version>1.0-SNAPSHOT</version>
10    <packaging>jar</packaging>
11
12    <repositories>
13      <repository>
14        <id>repository.spring.milestone</id>
15        <name>Spring Milestone Repository</name>
16        <url>https://repo.spring.io/milestone</url>
17      </repository>
18    </repositories>
19
20    <dependencies>
21      <!--spring context依赖-->
22      <dependency>
23        <groupId>org.springframework</groupId>
24        <artifactId>spring-context</artifactId>
25        <version>6.0.0-M2</version>
26      </dependency>
27    </dependencies>
28
29    <properties>
30      <maven.compiler.source>17</maven.compiler.source>
31      <maven.compiler.target>17</maven.compiler.target>
32    </properties>
33
34  </project>
```

注意：打包方式jar。

当加入spring context的依赖之后，会关联引入其他依赖：

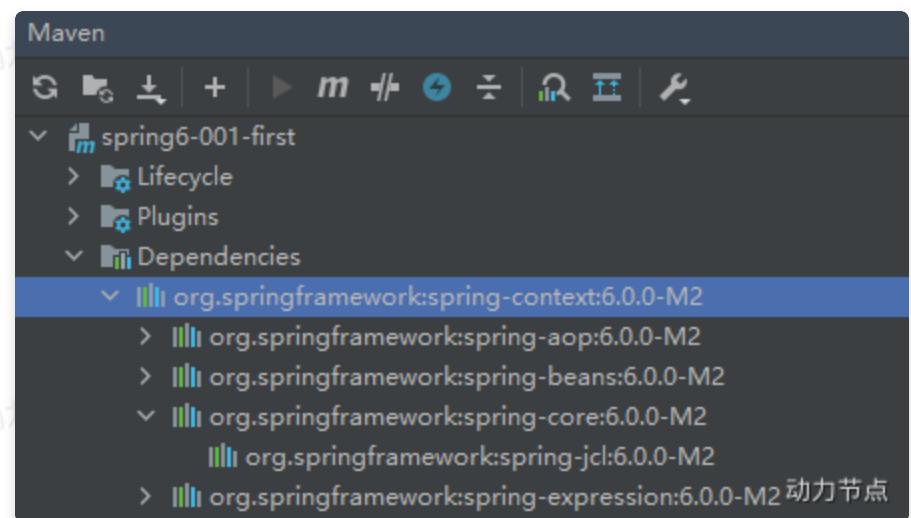
spring aop：面向切面编程

spring beans：IoC核心

spring core：spring的核心工具包

spring jcl: spring的日志包

spring expression: spring表达式



第二步：添加junit依赖

pom.xml

XML | 复制代码

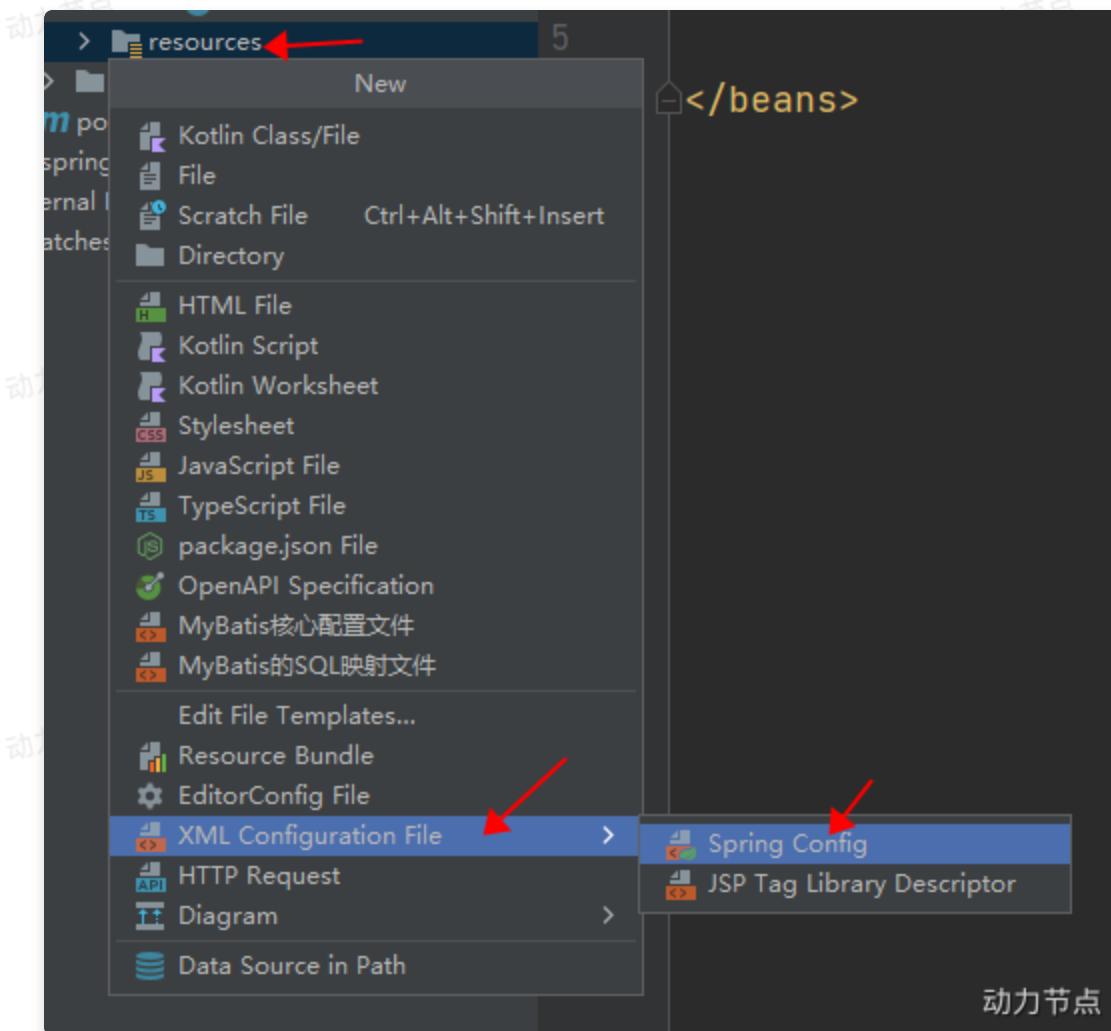
```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>com.powernode</groupId>
8     <artifactId>spring6-001-first</artifactId>
9     <version>1.0-SNAPSHOT</version>
10    <packaging>jar</packaging>
11
12    <repositories>
13      <repository>
14        <id>repository.spring.milestone</id>
15        <name>Spring Milestone Repository</name>
16        <url>https://repo.spring.io/milestone</url>
17      </repository>
18    </repositories>
19
20    <dependencies>
21      <!--spring context依赖-->
22      <dependency>
23        <groupId>org.springframework</groupId>
24        <artifactId>spring-context</artifactId>
25        <version>6.0.0-M2</version>
26      </dependency>
27      <!--junit-->
28      <dependency>
29        <groupId>junit</groupId>
30        <artifactId>junit</artifactId>
31        <version>4.13.2</version>
32        <scope>test</scope>
33      </dependency>
34    </dependencies>
35
36    <properties>
37      <maven.compiler.source>17</maven.compiler.source>
38      <maven.compiler.target>17</maven.compiler.target>
39    </properties>
40
41  </project>
```

第三步：定义bean：User

动力节点

```
▼ User.java Java | 复制代码  
1 package com.powernode.spring6.bean;  
2  
3 /**  
4     * bean, 封装用户信息。  
5     * @author 动力节点  
6     * @version 1.0  
7     * @since 1.0  
8     */  
9 public class User {  
10 }  
11
```

第四步：编写spring的配置文件：beans.xml。该文件放在类的根路径下。



上图是使用IDEA工具自带的spring配置文件的模板进行创建。

配置文件中进行bean的配置。

beans.xml

XML | 复制代码

```
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
6     <bean id="userBean" class="com.powernode.spring6.bean.User"/>
7   </beans>
```

bean的id和class属性：

- **id属性：代表对象的唯一标识。可以看做一个人的身份证号。**
- **class属性：用来指定要创建的java对象的类名，这个类名必须是全限定类名（带包名）。**

第五步：编写测试程序

Spring6Test.java

Java | 复制代码

```
1 package com.powernode.spring6.test;
2
3 import org.junit.Test;
4 import org.springframework.context.ApplicationContext;
5 import org.springframework.context.support.ClassPathXmlApplicationContext;
6
7 public class Spring6Test {
8
9     @Test
10    public void testFirst(){
11        // 初始化Spring容器上下文（解析beans.xml文件，创建所有的bean对象）
12        ApplicationContext applicationContext = new ClassPathXmlApplication
13        context("beans.xml");
14        // 根据id获取bean对象
15        Object userBean = applicationContext.getBean("userBean");
16        System.out.println(userBean);
17    }
18}
```

第七步：运行测试程序

```
Spring6Test.testFirst
Tests passed: 1 of 1 test - 429 ms
Spring6Test (co 429 ms)
  ✓ testFirst      429 ms
  com.powernode.spring6.bean.User@12468a38

Process finished with exit code 0
```

动力节点

3.4 第一个Spring程序详细剖析

```
beans.xml
<bean id="userBean" class="com.powernode.spring6.bean.User"/>

Spring6Test.testFirst()
ApplicationContext applicationContext = new ClassPathXmlApplicationContext("beans.xml");
Object userBean = applicationContext.getBean("userBean");
```

1. bean标签的id属性可以重复吗?

```
Vip.java
package com.powernode.spring6.bean;
/*
 * @author 动力节点
 * @version 1.0
 * @className Vip
 * @since 1.0
 */
public class Vip {
```

beans.xml

XML | 复制代码

```
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
6     <bean id="userBean" class="com.powernode.spring6.bean.User"/>
7     <bean id="userBean" class="com.powernode.spring6.bean.Vip"/>
8 </beans>
```

运行测试程序：

```
public class Spring6Test {
    @Test
    public void testFirst(){
        BeanFactory beanFactory;
        // 初始化Spring容器上下文（解析beans.xml文件，创建所有的bean对象）
        ApplicationContext applicationContext = new ClassPathXmlApplicationContext(configLocation: "beans.xml");
        // 根据id获取bean对象
        Object userBean = applicationContext.getBean(s: "userBean");
        System.out.println(userBean);
    }
}
```

Tests failed: 1 of 1 test - 230 ms
a\jdk-17.0.4\bin\java.exe ...
framework.beans.factory.parsing.BeanDefinitionParsingException: Configuration problem: Bean name 'userBean' is already used in this <beans> element
resource: class path resource [beans.xml]

动力节点

通过测试得出：在spring的配置文件中id是不能重名。

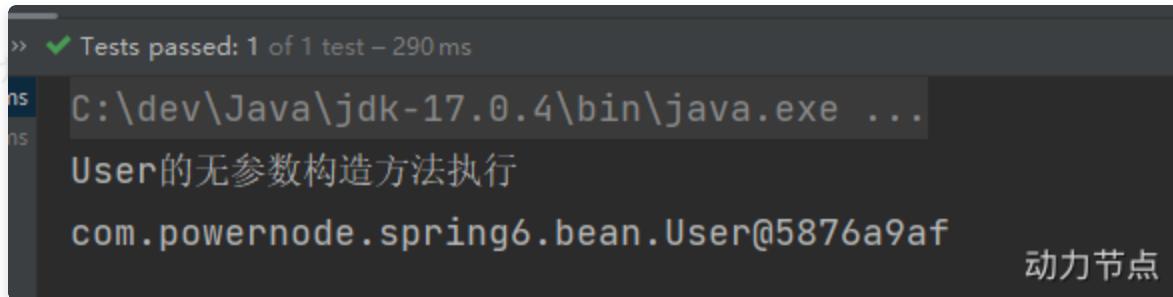
2. 底层是怎么创建对象的，是通过反射机制调用无参数构造方法吗？

```
▼ User.java Java | 复制代码

1 package com.powernode.spring6.bean;
2
3 /**
4  * bean, 封装用户信息。
5  * @author 动力节点
6  * @version 1.0
7  * @since 1.0
8 */
9 public class User {
10    public User() {
11        System.out.println("User的无参数构造方法执行");
12    }
13}
14
```

在User类中添加无参数构造方法，如上。

运行测试程序：



```
» ✓ Tests passed: 1 of 1 test - 290 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
User的无参数构造方法执行
com.powernode.spring6.bean.User@5876a9af      动力节点
```

通过测试得知：创建对象时确实调用了无参数构造方法。

如果提供一个有参数构造方法，不提供无参数构造方法会怎样呢？

```
1 package com.powernode.spring6.bean;
2
3 /**
4  * bean, 封装用户信息。
5  * @author 动力节点
6  * @version 1.0
7  * @since 1.0
8 */
9 public class User {
10    /*public User() {
11        System.out.println("User的无参数构造方法执行");
12    }*/
13
14    public User(String name){
15        System.out.println("User的有参数构造方法执行");
16    }
17}
18
```

运行测试程序：

```
Tests failed: 1 of 1 test - 280 ms

~k.beans.BeanInstantiationException: Failed to instantiate [com.powernode.spring6.bean.User]: No default constructor found; nested exception is
User]: No default constructor found; nested exception is java.lang.NoSuchMethodException: com.powernode.spring6.bean.User.<init>()
```

动力节点

通过测试得知：spring是通过调用类的无参数构造方法来创建对象的，所以要想让spring给你创建对象，必须保证无参数构造方法是存在的。

Spring是如何创建对象的呢？原理是什么？

```
1 // dom4j解析beans.xml文件，从中获取class的全限定类名
2 // 通过反射机制调用无参数构造方法创建对象
3 Class clazz = Class.forName("com.powernode.spring6.bean.User");
4 Object obj = clazz.newInstance();
```

3. 把创建好的对象存储到一个什么样的数据结构当中了呢？

Map<String, Object>

key(id)	value(bean)
"userBean"	User对象
"vipBean"	Vip对象
"studentBean"	Student对象
"userServiceBean"	UserService对象

4. spring配置文件的名字必须叫做beans.xml吗?

```
1 ApplicationContext applicationContext = new ClassPathXmlApplicationContext("beans.xml");
```

通过以上的java代码可以看出，这个spring配置文件名字是我们负责提供的，显然spring配置文件的名字是随意的。

5. 像这样的beans.xml文件可以有多个吗?

再创建一个spring配置文件，起名：spring.xml

spring.xml

XML | 复制代码

```
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         <bean id="vipBean" class="com.powernode.spring6.bean.Vip"/>
6     </beans>
```

Spring6Test.testFirst()

Java | 复制代码

```
1 package com.powernode.spring6.test;
2
3 import org.junit.Test;
4 import org.springframework.context.ApplicationContext;
5 import org.springframework.context.support.ClassPathXmlApplicationContext;
6
7 public class Spring6Test {
8
9     @Test
10    public void testFirst(){
11        // 初始化Spring容器上下文（解析beans.xml文件，创建所有的bean对象）
12        ApplicationContext applicationContext = new ClassPathXmlApplicationContext("beans.xml","spring.xml");
13
14        // 根据id获取bean对象
15        Object userBean = applicationContext.getBean("userBean");
16        Object vipBean = applicationContext.getBean("vipBean");
17
18        System.out.println(userBean);
19        System.out.println(vipBean);
20    }
21 }
22
```

运行测试程序：

```
Tests passed: 1 of 1 test - 276 ms  
C:\dev\Java\jdk-17.0.4\bin\java.exe ...  
User的无参数构造方法执行  
com.powernode.spring6.bean.User@2a798d51  
com.powernode.spring6.bean.Vip@6d763516  
动力节点
```

通过测试得知，spring的配置文件可以有多个，在ClassPathXmlApplicationContext构造方法的参数上传递文件路径即可。这是为什么呢？通过源码可以看到：

```
1 usage  
public ClassPathXmlApplicationContext(String... configLocations) throws BeansException {  
    this(configLocations, refresh: true, (ApplicationContext)null);  
}
```

动力节点

6. 在配置文件中配置的类必须是自定义的吗，可以使用JDK中的类吗，例如：java.util.Date？

```
beans.xml XML | 复制代码  
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  
6      <bean id="userBean" class="com.powernode.spring6.bean.User"/>  
7      <!--<bean id="userBean" class="com.powernode.spring6.bean.Vip"/><!--&gt;<br/>8  
9      <bean id="dateBean" class="java.util.Date"/>  
10 </beans>
```

Spring6Test.testFirst()

Java | 复制代码

```

1 package com.powernode.spring6.test;
2
3 import org.junit.Test;
4 import org.springframework.context.ApplicationContext;
5 import org.springframework.context.support.ClassPathXmlApplicationContext;
6
7 public class Spring6Test {
8
9     @Test
10    public void testFirst(){
11        // 初始化Spring容器上下文（解析beans.xml文件，创建所有的bean对象）
12        ApplicationContext applicationContext = new ClassPathXmlApplication
13        context("beans.xml","spring.xml");
14
15        // 根据id获取bean对象
16        Object userBean = applicationContext.getBean("userBean");
17        Object vipBean = applicationContext.getBean("vipBean");
18        Object dateBean = applicationContext.getBean("dateBean");
19
20        System.out.println(userBean);
21        System.out.println(vipBean);
22        System.out.println(dateBean);
23    }
24}

```

运行测试程序：

```

Tests passed: 1 of 1 test – 311 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
User的无参数构造方法执行
com.powernode.spring6.bean.User@6d763516
com.powernode.spring6.bean.Vip@52bf72b5
Thu Sep 22 15:16:03 CST 2022

```

动力节点

通过测试得知，在spring配置文件中配置的bean可以任意类，只要这个类不是抽象的，并且提供了无参数构造方法。

7. getBean()方法调用时，如果指定的id不存在会怎样？

```
@Test  
public void testFirst(){  
    // 初始化Spring容器上下文（解析beans.xml文件，创建所有的bean对象）  
    ApplicationContext applicationContext = new ClassPathXmlApplicationContext(...configLocations: "beans.xml", "spring.xml");  
  
    // 根据id获取bean对象  
    Object userBean = applicationContext.getBean("hahaBean");  
    Object vipBean = applicationContext.getBean("vipBean");  
    Object dateBean = applicationContext.getBean("dateBean");  
  
    System.out.println(userBean);  
    System.out.println(vipBean);  
    System.out.println(dateBean);  
}
```

动力节点

运行测试程序：

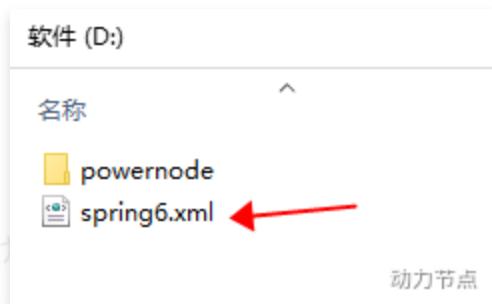
```
Tests failed: 1 of 1 test - 320 ms  
C:\dev\Java\jdk-17.0.4\bin\java.exe ...  
User的无参数构造方法执行  
  
org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'hahaBean' available  
  
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeanDefinition(DefaultListableBeanFactory.java:537)  
    at org.springframework.beans.factory.support.AbstractBeanFactory.getMergedLocalBeanDefinition(AbstractBeanFactory.java:434)  
    at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:375)  
    at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:317)  
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1177)  
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1167)
```

通过测试得知，当id不存在的时候，会出现异常。

8. getBean()方法返回的类型是Object，如果访问子类的特有属性和方法时，还需要向下转型，有其它办法可以解决这个问题吗？

```
1 User user = applicationContext.getBean("userBean", User.class);
```

9. ClassPathXmlApplicationContext是从类路径中加载配置文件，如果没有在类路径当中，又应该如何加载配置文件呢？



spring6.xml

XML | 复制代码

```
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         <bean id="vipBean2" class="com.powernode.spring6.bean.Vip"/>
6     </beans>
```

Java

Java | 复制代码

```
1 ApplicationContext applicationContext2 = new FileSystemXmlApplicationContext("d:/spring6.xml");
2 Vip vip = applicationContext2.getBean("vipBean2", Vip.class);
3 System.out.println(vip);
```

没有在类路径中的话，需要使用FileSystemXmlApplicationContext类进行加载配置文件。

这种方式较少用。一般都是将配置文件放到类路径当中，这样可移植性更强。

10. ApplicationContext的超级父接口BeanFactory。

Java

Java | 复制代码

```
1 BeanFactory beanFactory = new ClassPathXmlApplicationContext("spring.xml");
2 Object vipBean = beanFactory.getBean("vipBean");
3 System.out.println(vipBean);
```

BeanFactory是Spring容器的超级接口。ApplicationContext是BeanFactory的子接口。

3.5 Spring6启用Log4j2日志框架

从Spring5之后，Spring框架支持集成的日志框架是Log4j2.如何启用日志框架：

第一步：引入Log4j2的依赖

pom.xml

XML | 复制代码

```
1 <!--log4j2的依赖-->
2 <dependency>
3   <groupId>org.apache.logging.log4j</groupId>
4   <artifactId>log4j-core</artifactId>
5   <version>2.19.0</version>
6 </dependency>
7 <dependency>
8   <groupId>org.apache.logging.log4j</groupId>
9   <artifactId>log4j-slf4j2-impl</artifactId>
10  <version>2.19.0</version>
11 </dependency>
```

第二步：在类的根路径下提供log4j2.xml配置文件（文件名固定为：log4j2.xml，文件必须放到类根路径下。）

log4j2.xml

XML | 复制代码

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <configuration>
4
5   <loggers>
6     <!--
7       level指定日志级别，从低到高的优先级：
8         ALL < TRACE < DEBUG < INFO < WARN < ERROR < FATAL < OFF
9     -->
10    <root level="DEBUG">
11      <appender-ref ref="spring6log"/>
12    </root>
13  </loggers>
14
15  <appenders>
16    <!--输出日志信息到控制台-->
17    <console name="spring6log" target="SYSTEM_OUT">
18      <!--控制日志输出的格式-->
19      <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss SSS} [%t] %-3le
vel %logger{1024} - %msg%n"/>
20    </console>
21  </appenders>
22
23 </configuration>
```

第三步：使用日志框架

```
1 Logger logger = LoggerFactory.getLogger(FirstSpringTest.class);
2 logger.info("我是一条日志消息");
```

四、Spring对IoC的实现

4.1 IoC 控制反转

- 控制反转是一种思想。
- 控制反转是为了降低程序耦合度，提高程序扩展力，达到OCP原则，达到DIP原则。
- 控制反转，反转的是什么？
 - 将对象的创建权利交出去，交给第三方容器负责。
 - 将对象和对象之间关系的维护权交出去，交给第三方容器负责。
- 控制反转这种思想如何实现呢？
 - DI (Dependency Injection)：依赖注入

4.2 依赖注入

依赖注入实现了控制反转的思想。

Spring通过依赖注入的方式来完成Bean管理的。

Bean管理说的是：Bean对象的创建，以及Bean对象中属性的赋值（或者叫做Bean对象之间关系的维护）。

依赖注入：

- 依赖指的是对象和对象之间的关联关系。
- 注入指的是一种数据传递行为，通过注入行为来让对象和对象产生关系。

依赖注入常见的实现方式包括两种：

- 第一种：set注入
- 第二种：构造注入

新建模块：spring6-002-dependency-injection

4.2.1 set注入

set注入，基于set方法实现的，底层会通过反射机制调用属性对应的set方法然后给属性赋值。这种方式要求属性必须对外提供set方法。

```
pom.xml XML | 复制代码

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6     <groupId>com.powernode</groupId>
7     <artifactId>spring6-002-dependency-injection</artifactId>
8     <version>1.0-SNAPSHOT</version>
9     <packaging>jar</packaging>
10
11    <repositories>
12        <repository>
13            <id>repository.spring.milestone</id>
14            <name>Spring Milestone Repository</name>
15            <url>https://repo.spring.io/milestone</url>
16        </repository>
17    </repositories>
18
19    <dependencies>
20        <dependency>
21            <groupId>org.springframework</groupId>
22            <artifactId>spring-context</artifactId>
23            <version>6.0.0-M2</version>
24        </dependency>
25        <dependency>
26            <groupId>junit</groupId>
27            <artifactId>junit</artifactId>
28            <version>4.13.2</version>
29            <scope>test</scope>
30        </dependency>
31    </dependencies>
32
33    <properties>
34        <maven.compiler.source>17</maven.compiler.source>
35        <maven.compiler.target>17</maven.compiler.target>
36    </properties>
37
38 </project>
```

UserDao

Java | 复制代码

```
1 package com.powernode.spring6.dao;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className UserDao
7  * @since 1.0
8 */
9 public class UserDao {
10
11     public void insert(){
12         System.out.println("正在保存用户数据。");
13     }
14 }
15
```

UserService

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className UserService
9  * @since 1.0
10 */
11 public class UserService {
12
13     private UserDao userDao;
14
15     // 使用set方式注入，必须提供set方法。
16     // 反射机制要调用这个方法给属性赋值的。
17     public void setUserDao(UserDao userDao) {
18         this.userDao = userDao;
19     }
20
21     public void save(){
22         userDao.insert();
23     }
24 }
25
```

spring.xml

XML | 复制代码

```
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
6     <bean id="userDaoBean" class="com.powernode.spring6.dao.UserDao"/>
7
8     <bean id="userServiceBean" class="com.powernode.spring6.service.UserService">
9         <property name="userDao" ref="userDaoBean"/>
10    </bean>
11
12 </beans>
```

测试程序

Java | 复制代码

```
1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.service.UserService;
4 import org.junit.Test;
5 import org.springframework.context.ApplicationContext;
6 import org.springframework.context.support.ClassPathXmlApplicationContext;
7
8 /**
9  * @author 动力节点
10 * @version 1.0
11 * @className DITest
12 * @since 1.0
13 */
14 public class DITest {
15
16     @Test
17     public void testSetDI(){
18         ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
19         UserService userService = applicationContext.getBean("userServiceBean", UserService.class);
20         userService.save();
21     }
22 }
23 }
```

运行结果：

```
Tests passed: 1 of 1 test – 328 ms  
C:\dev\Java\jdk-17.0.4\bin\java.exe ...  
正在保存用户数据。  
|  
动力节点
```

重点内容是，什么原理：

```
spring.xml  
XML | 复制代码  
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         <bean id="userDaoBean" class="com.powernode.spring6.dao.UserDao"/>  
6  
7         <bean id="userServiceBean" class="com.powernode.spring6.service.UserService">  
8             <property name="userDao" ref="userDaoBean"/>  
9         </bean>  
10    </beans>
```

实现原理：

通过property标签获取到属性名：userDao

通过属性名推断出set方法名：setUserDao

通过反射机制调用setUserDao()方法给属性赋值

property标签的name是属性名。

property标签的ref是要注入的bean对象的id。**(通过ref属性来完成bean的装配，这是bean最简单的一种装配方式。装配指的是：创建系统组件之间关联的动作)**

可以把set方法注释掉，再测试一下：

```
ion: Invalid property 'userDao' of bean class [com.powernode.spring6.service.UserService]
```

```
: Bean property 'userDao' is not writable or has an invalid setter method. Does the property exist on the bean class? Does it have an appropriate setter?
```

动力节点

通过测试得知，底层实际上调用了setUserDao()方法。所以需要确保这个方法的存在。

我们现在把属性名修改一下，但方法名还是setUserDao()，我们来测试一下：

UserService

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className UserService
9  * @since 1.0
10 */
11 public class UserService {
12
13     private UserDao aaa;
14
15     // 使用set方式注入，必须提供set方法。
16     // 反射机制要调用这个方法给属性赋值的。
17     public void setUserDao(UserDao userDao) {
18         this.aaa = userDao;
19     }
20
21     public void save(){
22         aaa.insert();
23     }
24 }
25
```

运行测试程序：

动力节点

Tests passed: 1 of 1 test – 297 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

正在保存用户数据。

动力节点

通过测试看到程序仍然可以正常执行，说明property标签的name是：setUserDao()方法名演变得到的。

演变的规律是：

- setUsername() 演变为 username
- setPassword() 演变为 password
- setUserDao() 演变为 userDao
- setUserService() 演变为 userService

另外，对于property标签来说，ref属性也可以采用标签的方式，但使用ref属性是多数的：

spring.xml

XML | 复制代码

```
1 <bean id="userServiceBean" class="com.powernode.spring6.service.UserService">
  2   <property name="userDao">
  3     <ref bean="userDaoBean"/>
  4   </property>
  5 </bean>
```

总结：set注入的核心实现原理：通过反射机制调用set方法来给属性赋值，让两个对象之间产生关系。

4.2.2 构造注入

核心原理：通过调用构造方法来给属性赋值。

OrderDao

Java | 复制代码

```
1 package com.powernode.spring6.dao;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className OrderDao
7  * @since 1.0
8 */
9 public class OrderDao {
10    public void deleteById(){
11        System.out.println("正在删除订单。。。");
12    }
13}
```

OrderService

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.OrderDao;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className OrderService
9  * @since 1.0
10 */
11 public class OrderService {
12     private OrderDao orderDao;
13
14     // 通过反射机制调用构造方法给属性赋值
15     public OrderService(OrderDao orderDao) {
16         this.orderDao = orderDao;
17     }
18
19     public void delete(){
20         orderDao.deleteById();
21     }
22 }
```

spring.xml

XML | 复制代码

```
1 <bean id="orderDaoBean" class="com.powernode.spring6.dao.OrderDao"/>
2 <bean id="orderServiceBean" class="com.powernode.spring6.service.OrderService">
3     <!--index="0"表示构造方法的第一个参数，将orderDaoBean对象传递给构造方法的第一个参数。-->
4     <constructor-arg index="0" ref="orderDaoBean"/>
5 </bean>
```

测试程序

Java | 复制代码

```
1 @Test
2 public void testConstructorDI(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
4     OrderService orderServiceBean = applicationContext.getBean("orderServiceBean", OrderService.class);
5     orderServiceBean.delete();
6 }
```

运行结果如下：

```
Tests passed: 1 of 1 test - 344 ms
c:\dev\Java\jdk-17.0.4\bin\java.exe ...
正在删除订单。。
动力节点
```

如果构造方法有两个参数：

OrderService

Java | 复制代码

```

1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.OrderDao;
4 import com.powernode.spring6.dao.UserDao;
5
6 /**
7  * @author 动力节点
8  * @version 1.0
9  * @className OrderService
10 * @since 1.0
11 */
12 public class OrderService {
13     private OrderDao orderDao;
14     private UserDao userDao;
15
16     // 通过反射机制调用构造方法给属性赋值
17     public OrderService(OrderDao orderDao, UserDao userDao) {
18         this.orderDao = orderDao;
19         this.userDao = userDao;
20     }
21
22     public void delete(){
23         orderDao.deleteById();
24         userDao.insert();
25     }
26 }
27

```

spring配置文件:

spring.xml

XML | 复制代码

```

1 <bean id="orderDaoBean" class="com.powernode.spring6.dao.OrderDao"/>
2
3 <bean id="orderServiceBean" class="com.powernode.spring6.service.OrderService">
4     <!--第一个参数下标是0-->
5     <constructor-arg index="0" ref="orderDaoBean"/>
6     <!--第二个参数下标是1-->
7     <constructor-arg index="1" ref="userDaoBean"/>
8 </bean>
9
10 <bean id="userDaoBean" class="com.powernode.spring6.dao.UserDao"/>

```

执行测试程序:

Tests passed: 1 of 1 test – 456 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

正在删除订单。。

正在保存用户数据。

动力节点

不使用参数下标，使用参数的名字可以吗？

spring.xml

XML | 复制代码

```
1 <bean id="orderDaoBean" class="com.powernode.spring6.dao.OrderDao"/>
2
3 <bean id="orderServiceBean" class="com.powernode.spring6.service.OrderService">
4   <!--这里使用了构造方法上参数的名字-->
5   <constructor-arg name="orderDao" ref="orderDaoBean"/>
6   <constructor-arg name="userDao" ref="userDaoBean"/>
7 </bean>
8
9 <bean id="userDaoBean" class="com.powernode.spring6.dao.UserDao"/>
```

执行测试程序：

Tests passed: 1 of 1 test – 330 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

正在删除订单。。

正在保存用户数据。

动力节点

不指定参数下标，不指定参数名字，可以吗？

spring.xml

XML | 复制代码

```
1 <bean id="orderDaoBean" class="com.powernode.spring6.dao.OrderDao"/>
2 <bean id="orderServiceBean" class="com.powernode.spring6.service.OrderService">
3   <!--没有指定下标，也没有指定参数名字-->
4   <constructor-arg ref="orderDaoBean"/>
5   <constructor-arg ref="userDaoBean"/>
6 </bean>
7
8 <bean id="userDaoBean" class="com.powernode.spring6.dao.UserDao"/>
```

执行测试程序：

```
Tests passed: 1 of 1 test - 353 ms  
C:\dev\Java\jdk-17.0.4\bin\java ...  
正在删除订单。。。  
正在保存用户数据。  
动力节点
```

动力节点

配置文件中构造方法参数的类型顺序和构造方法参数的类型顺序不一致呢？

```
spring.xml  
1 <bean id="orderDaoBean" class="com.powernode.spring6.dao.OrderDao"/>  
2  
3 <bean id="orderServiceBean" class="com.powernode.spring6.service.OrderService">  
4     <!--顺序已经和构造方法的参数顺序不同了-->  
5     <constructor-arg ref="userDaoBean"/>  
6     <constructor-arg ref="orderDaoBean"/>  
7 </bean>  
8  
9 <bean id="userDaoBean" class="com.powernode.spring6.dao.UserDao"/>
```

执行测试程序：

```
Tests passed: 1 of 1 test - 316 ms  
C:\dev\Java\jdk-17.0.4\bin\java.exe ...  
正在删除订单。。。  
正在保存用户数据。  
动力节点
```

动力节点

通过测试得知，通过构造方法注入的时候：

- 可以通过下标
- 可以通过参数名
- 也可以不指定下标和参数名，可以类型自动推断。

Spring在装配方面做的还是比较健壮的。

4.3 set注入专题

4.3.1 注入外部Bean

在之前4.2.1中使用的案例就是注入外部Bean的方式。

```
spring.xml XML 复制代码

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
6     <bean id="userDaoBean" class="com.powernode.spring6.dao.UserDao"/>
7
8     <bean id="userServiceBean" class="com.powernode.spring6.service.UserService">
9         <property name="userDao" ref="userDaoBean"/>
10    </bean>
11
12 </beans>
```

外部Bean的特点：bean定义到外面，在property标签中使用ref属性进行注入。通常这种方式是常用。

4.3.2 注入内部Bean

内部Bean的方式：在bean标签中嵌套bean标签。

```
spring-inner-bean.xml XML 复制代码

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
6     <bean id="userServiceBean" class="com.powernode.spring6.service.UserService">
7         <property name="userDao">
8             <bean class="com.powernode.spring6.dao.UserDao"/>
9         </property>
10    </bean>
11
12 </beans>
```

```
▼ DIITest.testInnerBean() Java | 复制代码

1 @Test
2 public void testInnerBean(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationCont
ext("spring-inner-bean.xml");
4     UserService userService = applicationContext.getBean("userServiceBean",
UserService.class);
5     userService.save();
6 }
```

执行测试程序：

```
✓ Tests passed: 1 of 1 test - 281 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
正在保存用户数据。
动力节点
```

这种方式作为了解。

4.3.3 注入简单类型

我们之前在进行注入的时候，对象的属性是另一个对象。

```
▼ 对象的属性是另一个对象 Java | 复制代码

1 public class UserService{
2
3     private UserDao userDao;
4
5     public void setUserDao(UserDao userDao){
6         this.userDao = userDao;
7     }
8
9 }
```

那如果对象的属性是int类型呢？

▼ 对象的属性是int类型

Java | 复制代码

```
1 public class User{  
2  
3     private int age;  
4  
5     public void setAge(int age){  
6         this.age = age;  
7     }  
8  
9 }
```

可以通过set注入的方式给该属性赋值吗？

- 当然可以。因为只要能够调用set方法就可以给属性赋值。

编写程序给一个User对象的age属性赋值20：

第一步：定义User类，提供age属性，提供age属性的setter方法。

▼ User

Java | 复制代码

```
1 package com.powernode.spring6.beans;  
2  
3 /**  
4  * @author 动力节点  
5  * @version 1.0  
6  * @className User  
7  * @since 1.0  
8  **/  
9 public class User {  
10     private int age;  
11  
12     public void setAge(int age) {  
13         this.age = age;  
14     }  
15  
16     @Override  
17     public String toString() {  
18         return "User{" +  
19             "age=" + age +  
20             '}';  
21     }  
22 }
```

第二步：编写spring配置文件：spring-simple-type.xml

spring-simple-type.xml

XML | 复制代码

```
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     <bean id="userBean" class="com.powernode.spring6.beans.User">
6         <!--如果像这种int类型的属性，我们称为简单类型，这种简单类型在注入的时候要使用value属性，不能使用ref-->
7         <!--<property name="age" value="20"/>-->
8         <property name="age">
9             <value>20</value>
10        </property>
11    </bean>
12 </beans>
```

第三步：编写测试程序

DITest.testSimpleType

Java | 复制代码

```
1 @Test
2 public void testSimpleType(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-simple-type.xml");
4     User user = applicationContext.getBean("userBean", User.class);
5     System.out.println(user);
6 }
```

第四步：运行测试程序

```
Tests passed: 1 of 1 test - 312 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
User{age=20}
```

需要特别注意：如果给简单类型赋值，使用value属性或value标签。而不是ref。

简单类型包括哪些呢？可以通过Spring的源码来分析一下：BeanUtils类

BeanUtils

Java | 复制代码

```
1 public class BeanUtils{  
2     //.....  
3     /**  
4      * Check if the given type represents a "simple" property: a simple va  
5      * lue  
6      * type or an array of simple value types.  
7      * <p>See {@link #isSimpleValueType(Class)} for the definition of <em>  
8      * simple  
9      * value type</em>.  
10     * <p>Used to determine properties to check for a "simple" dependency-  
11     * check.  
12     * @param type the type to check  
13     * @return whether the given type represents a "simple" property  
14     * @see org.springframework.beans.factory.support.RootBeanDefinition#D  
15     * EPENDENCY_CHECK_SIMPLE  
16     * @see org.springframework.beans.factory.support.AbstractAutowireCapa  
17     * bleBeanFactory#checkDependencies  
18     * @see #isSimpleValueType(Class)  
19     */  
20     public static boolean isSimpleProperty(Class<?> type) {  
21         Assert.notNull(type, "'type' must not be null");  
22         return isSimpleValueType(type) || (type.isArray() && isSimpleValue  
23         Type(type.getComponentType()));  
24     }  
25     /**  
26      * Check if the given type represents a "simple" value type: a primiti  
27      * ve or  
28      * primitive wrapper, an enum, a String or other CharSequence, a Numbe  
29      * r, a  
30      * Date, a Temporal, a URI, a URL, a Locale, or a Class.  
31      * <p>{@code Void} and {@code void} are not considered simple value ty  
32      * pes.  
33      * @param type the type to check  
34      * @return whether the given type represents a "simple" value type  
35      * @see #isSimpleProperty(Class)  
36      */  
37     public static boolean isSimpleValueType(Class<?> type) {  
38         return (Void.class != type && void.class != type &&  
39                 (ClassUtils.isPrimitiveOrWrapper(type) ||  
40                  Enum.class.isAssignableFrom(type) ||  
41                  CharSequence.class.isAssignableFrom(type) ||  
42                  Number.class.isAssignableFrom(type) ||
```

```
37     Date.class.isAssignableFrom(type) ||
38     Temporal.class.isAssignableFrom(type) ||
39     URI.class == type ||
40     URL.class == type ||
41     Locale.class == type ||
42     Class.class == type));
43 }
44 //.....
45 }
46 }
```

通过源码分析得知，简单类型包括：

- 基本数据类型
- 基本数据类型对应的包装类
- String或其他的CharSequence子类
- Number子类
- Date子类
- Enum子类
- URI
- URL
- Temporal子类
- Locale
- Class
- 另外还包括以上简单值类型对应的数组类型。

经典案例：给数据源的属性注入值：

假设我们现在要自己手写一个数据源，我们都知道所有的数据源都要实现javax.sql.DataSource接口，并且数据源中应该有连接数据库的信息，例如：driver、url、username、password等。

MyDataSource

Java | 复制代码

```
1 package com.powernode.spring6.beans;
2
3 import javax.sql.DataSource;
4 import java.io.PrintWriter;
5 import java.sql.Connection;
6 import java.sql.SQLException;
7 import java.sql.SQLFeatureNotSupportedException;
8 import java.util.logging.Logger;
9
10 /**
11  * @author 动力节点
12  * @version 1.0
13  * @className MyDataSource
14  * @since 1.0
15  */
16 public class MyDataSource implements DataSource {
17     private String driver;
18     private String url;
19     private String username;
20     private String password;
21
22     public void setDriver(String driver) {
23         this.driver = driver;
24     }
25
26     public void setUrl(String url) {
27         this.url = url;
28     }
29
30     public void setUsername(String username) {
31         this.username = username;
32     }
33
34     public void setPassword(String password) {
35         this.password = password;
36     }
37
38     @Override
39     public String toString() {
40         return "MyDataSource{" +
41                 "driver='" + driver + '\'' +
42                 ", url='" + url + '\'' +
43                 ", username='" + username + '\'' +
44                 ", password='" + password + '\'' +
45                 '}';
46 }
```

```
46     }
47
48     @Override
49     public Connection getConnection() throws SQLException {
50         return null;
51     }
52
53     @Override
54     public Connection getConnection(String username, String password) throws
55     SQLException {
56         return null;
57     }
58
59     @Override
60     public PrintWriter getLogWriter() throws SQLException {
61         return null;
62     }
63
64     @Override
65     public void setLogWriter(PrintWriter out) throws SQLException {
66     }
67
68     @Override
69     public void setLoginTimeout(int seconds) throws SQLException {
70     }
71
72     @Override
73     public int getLoginTimeout() throws SQLException {
74         return 0;
75     }
76
77     @Override
78     public Logger getParentLogger() throws SQLFeatureNotSupportedException
79     {
80         return null;
81     }
82
83     @Override
84     public <T> T unwrap(Class<T> iface) throws SQLException {
85         return null;
86     }
87
88     @Override
89     public boolean isWrapperFor(Class<?> iface) throws SQLException {
90         return false;
91     }
```

```
92 }  
93 }
```

我们给driver、url、username、password四个属性分别提供了setter方法，我们可以使用spring的依赖注入完成数据源对象的创建和属性的赋值吗？看配置文件

```
spring-datasource.xml  
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  
6     <bean id="dataSource" class="com.powernode.spring6.beans.MyDataSource">  
7         <property name="driver" value="com.mysql.cj.jdbc.Driver"/>  
8         <property name="url" value="jdbc:mysql://localhost:3306/spring"/>  
9         <property name="username" value="root"/>  
10        <property name="password" value="123456"/>  
11    </bean>  
12  
13 </beans>
```

测试程序：

```
DItest.testDataSource  
1 @Test  
2 public void testDataSource(){  
3     ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-datasource.xml");  
4     MyDataSource dataSource = applicationContext.getBean("dataSource", MyDataSource.class);  
5     System.out.println(dataSource);  
6 }
```

执行测试程序：

```
Tests passed: 1 of 1 test - 312 ms  
C:\dev\Java\jdk-17.0.4\bin\java.exe ...  
MyDataSource{driver='com.mysql.cj.jdbc.Driver', url='jdbc:mysql://localhost:3306/spring', username='root', password='123456'}
```

你学会了吗？

接下来，我们编写一个程序，把所有的简单类型全部测试一遍：

编写一个类A：

A

Java | 复制代码

```
1 package com.powernode.spring6.beans;
2
3 import java.net.URI;
4 import java.net.URL;
5 import java.time.LocalDate;
6 import java.util.Date;
7 import java.util.Locale;
8
9 /**
10 * @author 动力节点
11 * @version 1.0
12 * @className A
13 * @since 1.0
14 */
15 public class A {
16     private byte b;
17     private short s;
18     private int i;
19     private long l;
20     private float f;
21     private double d;
22     private boolean flag;
23     private char c;
24
25     private Byte b1;
26     private Short s1;
27     private Integer i1;
28     private Long l1;
29     private Float f1;
30     private Double d1;
31     private Boolean flag1;
32     private Character c1;
33
34     private String str;
35
36     private Date date;
37
38     private Season season;
39
40     private URI uri;
41
42     private URL url;
43
44     private LocalDate localDate;
45 }
```

```
46     private Locale locale;
47
48     private Class clazz;
49
50     // 生成setter方法
51     // 生成toString方法
52 }
53
54 enum Season {
55     SPRING, SUMMER, AUTUMN, WINTER
56 }
57 }
```

动力节点

动力节点

动力节点

动力节点

动力节点

动力节点

动力节点

动力节点

```
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
6   <bean id="a" class="com.powernode.spring6.beans.A">
7     <property name="b" value="1"/>
8     <property name="s" value="1"/>
9     <property name="i" value="1"/>
10    <property name="l" value="1"/>
11    <property name="f" value="1"/>
12    <property name="d" value="1"/>
13    <property name="flag" value="false"/>
14
15    <property name="c" value="a"/>
16    <property name="b1" value="2"/>
17    <property name="s1" value="2"/>
18    <property name="i1" value="2"/>
19    <property name="l1" value="2"/>
20    <property name="f1" value="2"/>
21    <property name="d1" value="2"/>
22    <property name="flag1" value="true"/>
23    <property name="c1" value="a"/>
24
25    <property name="str" value="zhangsan"/>
26      <!--注意：value后面的日期字符串格式不能随便写，必须是Date对象toString()方法
执行的结果。-->
27      <!--如果想使用其他格式的日期字符串，就需要进行特殊处理了。具体怎么处理，可以看
后面的课程！！！-->
28    <property name="date" value="Fri Sep 30 15:26:38 CST 2022"/>
29    <property name="season" value="WINTER"/>
30    <property name="uri" value="/save.do"/>
31      <!--spring6之后，会自动检查url是否有效，如果无效会报错。-->
32    <property name="url" value="http://www.baidu.com"/>
33    <property name="localDate" value="EPOCH"/>
34      <!--java.util.Locale 主要在软件的本地化时使用。它本身没有什么功能，更多的是
作为一个参数辅助其他方法完成输出的本地化。-->
35    <property name="locale" value="CHINESE"/>
36    <property name="clazz" value="java.lang.String"/>
37  </bean>
38 </beans>
```

编写测试程序：

▼ DI Test.testAllSimpleType()

Java | 复制代码

```
1 @Test
2 public void testAllSimpleType(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationCont
ext("spring-all-simple-type.xml");
4     A a = applicationContext.getBean("a", A.class);
5     System.out.println(a);
6 }
```

执行结果如下：

```
Tests passed: 1 of 1 test - 369 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
A{b=1, s=1, i=1, l=1, f=1.0, d=1.0, flag=false, c=a, b1=2, s1=2, i1=2, l1=2, f1=2.0, d1=2.0, flag1=true, c1=a, str='zhangsan', date='2023-10-11'}
```

需要注意的是：

- 如果把Date当做简单类型的话，日期字符串格式不能随便写。格式必须符合Date的toString()方法格式。显然这就比较鸡肋了。如果我们提供一个这样的日期字符串：2010-10-11，在这里是无法赋值给Date类型的属性的。
- spring6之后，当注入的是URL，那么这个url字符串是会进行有效性检测的。如果是一个存在的url，那就没问题。如果不存在则报错。

4.3.4 级联属性赋值（了解）

Java | 复制代码

```
1 package com.powernode.spring6.beans;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Clazz
7  * @since 1.0
8 */
9 public class Clazz {
10     private String name;
11
12     public Clazz() {
13     }
14
15     public Clazz(String name) {
16         this.name = name;
17     }
18
19     public String getName() {
20         return name;
21     }
22
23     public void setName(String name) {
24         this.name = name;
25     }
26
27     @Override
28     public String toString() {
29         return "Clazz{" +
30                 "name='" + name + '\'' +
31                 '}';
32     }
33 }
34
```

Student

Java | 复制代码

```
1 package com.powernode.spring6.beans;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Student
7  * @since 1.0
8 */
9 public class Student {
10     private String name;
11     private Clazz clazz;
12
13     public Student() {
14
15
16     public Student(String name, Clazz clazz) {
17         this.name = name;
18         this.clazz = clazz;
19     }
20
21     public void setName(String name) {
22         this.name = name;
23     }
24
25     public void setClazz(Clazz clazz) {
26         this.clazz = clazz;
27     }
28
29     public Clazz getClazz() {
30         return clazz;
31     }
32
33     @Override
34     public String toString() {
35         return "Student{" +
36                 "name='" + name + '\'' +
37                 ", clazz='" + clazz +
38                 '}';
39     }
40 }
41
```

```
spring-cascade.xml
```

XML | 复制代码

```
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
6     <bean id="clazzBean" class="com.powernode.spring6.beans.Clazz"/>
7
8     <bean id="student" class="com.powernode.spring6.beans.Student">
9       <property name="name" value="张三"/>
10
11      <!--要点1：以下两行配置的顺序不能颠倒-->
12      <property name="clazz" ref="clazzBean"/>
13      <!--要点2： clazz属性必须有getter方法-->
14      <property name="clazz.name" value="高三一班"/>
15    </bean>
16 </beans>
```

```
测试程序
```

Java | 复制代码

```
1 @Test
2 public void testCascade(){
3   ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-cascade.xml");
4   Student student = applicationContext.getBean("student", Student.class);
5   System.out.println(student);
6 }
```

运行结果：

```
Tests passed: 1 of 1 test – 328 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
Student{name='张三', clazz=Clazz{name='高三一班'}}|
```

要点：

- 在spring配置文件中，如上，注意顺序。
- 在spring配置文件中， clazz属性必须提供getter方法。

4.3.5 注入数组

当数组中的元素是简单类型：

▼ Person

Java | 复制代码

```
1 package com.powernode.spring6.beans;
2
3 import java.util.Arrays;
4
5 public class Person {
6     private String[] favariteFoods;
7
8     public void setFavariteFoods(String[] favariteFoods) {
9         this.favariteFoods = favariteFoods;
10    }
11
12    @Override
13    public String toString() {
14        return "Person{" +
15                "favariteFoods=" + Arrays.toString(favariteFoods) +
16                '}';
17    }
18 }
19 }
```

▼ spring-array-simple.xml

XML | 复制代码

```
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
5                         http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <bean id="person" class="com.powernode.spring6.beans.Person">
8         <property name="favariteFoods">
9             <array>
10                 <value>鸡排</value>
11                 <value>汉堡</value>
12                 <value>鹅肝</value>
13             </array>
14         </property>
15     </bean>
16 </beans>
```

▼ DITest.testArraySimple()

Java | 复制代码

```
1  @Test
2  public void testArraySimple(){
3      ApplicationContext applicationContext = new ClassPathXmlApplicationCont
ext("spring-array-simple.xml");
4      Person person = applicationContext.getBean("person", Person.class);
5      System.out.println(person);
6  }
```

当数组中的元素是非简单类型：一个订单中包含多个商品。

Goods

Java | 复制代码

```
1 package com.powernode.spring6.beans;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Goods
7  * @since 1.0
8 */
9 public class Goods {
10     private String name;
11
12     public Goods() {
13     }
14
15     public Goods(String name) {
16         this.name = name;
17     }
18
19     public String getName() {
20         return name;
21     }
22
23     public void setName(String name) {
24         this.name = name;
25     }
26
27     @Override
28     public String toString() {
29         return "Goods{" +
30                 "name='" + name + '\'' +
31                 '}';
32     }
33 }
34 }
```

动力节点

动力节点

Order

Java | 复制代码

```
1 package com.powernode.spring6.beans;
2
3 import java.util.Arrays;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className Order
9  * @since 1.0
10 */
11 public class Order {
12     // 一个订单中有多个商品
13     private Goods[] goods;
14
15     public Order() {
16     }
17
18     public Order(Goods[] goods) {
19         this.goods = goods;
20     }
21
22     public void setGoods(Goods[] goods) {
23         this.goods = goods;
24     }
25
26     @Override
27     public String toString() {
28         return "Order{" +
29                 "goods=" + Arrays.toString(goods) +
30                 '}';
31     }
32 }
33
```

spring-array.xml

XML | 复制代码

```

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
6     <bean id="goods1" class="com.powernode.spring6.beans.Goods">
7         <property name="name" value="西瓜"/>
8     </bean>
9
10    <bean id="goods2" class="com.powernode.spring6.beans.Goods">
11        <property name="name" value="苹果"/>
12    </bean>
13
14    <bean id="order" class="com.powernode.spring6.beans.Order">
15        <property name="goods">
16            <array>
17                <!--这里使用ref标签即可-->
18                <ref bean="goods1"/>
19                <ref bean="goods2"/>
20            </array>
21        </property>
22    </bean>
23
24 </beans>

```

测试程序：

DITest.testArray()

Java | 复制代码

```

1 @Test
2 public void testArray(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-array.xml");
4     Order order = applicationContext.getBean("order", Order.class);
5     System.out.println(order);
6 }

```

执行结果：

Tests passed: 1 of 1 test – 351 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

Order{goods=[Goods{name='西瓜'}, Goods{name='苹果'}]}
动力节点

要点:

- 如果数组中是简单类型，使用value标签。
- 如果数组中是非简单类型，使用ref标签。

4.3.6 注入List集合

List集合：有序可重复

```
▼ People  
1 package com.powernode.spring6.beans;  
2  
3 import java.util.List;  
4  
5 /**  
6     * @author 动力节点  
7     * @version 1.0  
8     * @className People  
9     * @since 1.0  
10    **/  
11   public class People {  
12       // 一个人有多个名字  
13       private List<String> names;  
14  
15      public void setNames(List<String> names) {  
16          this.names = names;  
17      }  
18  
19      @Override  
20      public String toString() {  
21          return "People{" +  
22                  "names=" + names +  
23                  '}';  
24      }  
25  }
```

Java | 复制代码

spring-collection.xml

XML | 复制代码

```
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
6     <bean id="peopleBean" class="com.powernode.spring6.beans.People">
7         <property name="names">
8             <list>
9                 <value>铁锤</value>
10                <value>张三</value>
11                <value>张三</value>
12                <value>张三</value>
13                <value>狼</value>
14            </list>
15        </property>
16    </bean>
17 </beans>
```

测试程序

Java | 复制代码

```
1 @Test
2 public void testCollection(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-collection.xml");
4     People peopleBean = applicationContext.getBean("peopleBean", People.class);
5     System.out.println(peopleBean);
6 }
```

执行结果：

Tests passed: 1 of 1 test – 308 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

People{names=[铁锤, 张三, 张三, 张三, 狼]}

注意：注入List集合的时候使用list标签，如果List集合中是简单类型使用value标签，反之使用ref标签。

4.3.7 注入Set集合

Set集合：无序不可重复

▼ People

Java | 复制代码

```
1 package com.powernode.spring6.beans;
2
3 import java.util.List;
4 import java.util.Set;
5
6 /**
7  * @author 动力节点
8  * @version 1.0
9  * @className People
10 * @since 1.0
11 */
12 public class People {
13     // 一个人有多个电话
14     private Set<String> phones;
15
16     public void setPhones(Set<String> phones) {
17         this.phones = phones;
18     }
19
20     //.....
21
22     @Override
23     public String toString() {
24         return "People{" +
25                 "phones=" + phones +
26                 ", names=" + names +
27                 '}';
28     }
29 }
30
```

动力节点

动力节点

动力节点

动力节点

```
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
6     <bean id="peopleBean" class="com.powernode.spring6.beans.People">
7         <property name="phones">
8             <set>
9                 <!--非简单类型可以使用ref，简单类型使用value-->
10                <value>110</value>
11                <value>110</value>
12                <value>120</value>
13                <value>120</value>
14                <value>119</value>
15                <value>119</value>
16            </set>
17        </property>
18    </bean>
19 </beans>
```

执行结果：

```
Tests passed: 1 of 1 test – 329 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
People{phones=[110, 120, 119], names=[铁锤, 张三, 张三, 张三, 狼]}  
动力节点
```

要点：

- 使用`<set>`标签
- `set`集合中元素是简单类型的使用`value`标签，反之使用`ref`标签。

4.3.8 注入Map集合

People

Java | 复制代码

```
1 package com.powernode.spring6.beans;
2
3 import java.util.List;
4 import java.util.Map;
5 import java.util.Set;
6
7 /**
8 * @author 动力节点
9 * @version 1.0
10 * @className People
11 * @since 1.0
12 */
13 public class People {
14     // 一个人有多个住址
15     private Map<Integer, String> addrs;
16
17     public void setAddrs(Map<Integer, String> addrs) {
18         this.addrs = addrs;
19     }
20
21     //.....
22
23     @Override
24     public String toString() {
25         return "People{" +
26                 "addrs=" + addrs +
27                 ", phones=" + phones +
28                 ", names=" + names +
29                 '}';
30     }
31
32 }
33
```

spring-collection.xml

XML | 复制代码

```
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
6     <bean id="peopleBean" class="com.powernode.spring6.beans.People">
7         <property name="addrs">
8             <map>
9                 <!--如果key不是简单类型，使用 key-ref 属性-->
10                <!--如果value不是简单类型，使用 value-ref 属性-->
11                <entry key="1" value="北京大兴区"/>
12                <entry key="2" value="上海浦东区"/>
13                <entry key="3" value="深圳宝安区"/>
14            </map>
15        </property>
16    </bean>
17 </beans>
```

执行结果：

```
Tests passed: 1 of 1 test - 328 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
People{addrs={1=北京大兴区, 2=上海浦东区, 3=深圳宝安区}, phones=[110, 120, 119], names=[铁锤, 张三, 动力节点, 张五, 王六]}
```

要点：

- 使用`<map>`标签
- 如果key是简单类型，使用`key`属性，反之使用`key-ref`属性。
- 如果value是简单类型，使用`value`属性，反之使用`value-ref`属性。

4.3.9 注入Properties

`java.util.Properties`继承`java.util.Hashtable`，所以`Properties`也是一个Map集合。

People

Java | 复制代码

```
1 package com.powernode.spring6.beans;
2
3 import java.util.List;
4 import java.util.Map;
5 import java.util.Properties;
6 import java.util.Set;
7
8 /**
9  * @author 动力节点
10 * @version 1.0
11 * @className People
12 * @since 1.0
13 */
14 public class People {
15
16     private Properties properties;
17
18     public void setProperties(Properties properties) {
19         this.properties = properties;
20     }
21
22     //.....
23
24     @Override
25     public String toString() {
26         return "People{" +
27             "properties=" + properties +
28             ", addrs=" + addrs +
29             ", phones=" + phones +
30             ", names=" + names +
31             '}';
32     }
33 }
34 }
```

动力节点

动力节点

spring-collection.xml

XML | 复制代码

```
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
6     <bean id="peopleBean" class="com.powernode.spring6.beans.People">
7         <property name="properties">
8             <props>
9                 <prop key="driver">com.mysql.cj.jdbc.Driver</prop>
10                <prop key="url">jdbc:mysql://localhost:3306/spring</prop>
11                <prop key="username">root</prop>
12                <prop key="password">123456</prop>
13            </props>
14        </property>
15    </bean>
16 </beans>
```

执行测试程序：

```
Tests passed: 1 of 1 test - 312 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
People{properties={password=123456, driver=com.mysql.cj.jdbc.Driver, url=jdbc:mysql://localhost:3306/spring, username=root}}
```

要点：

- 使用`<props>`标签嵌套`<prop>`标签完成。

4.3.10 注入null和空字符串

注入空字符串使用：`<value/>` 或者 `value=""`

注入null使用：`<null/>` 或者 不为该属性赋值

- 我们先来看一下，怎么注入空字符串。

Vip

Java | 复制代码

```
1 package com.powernode.spring6.beans;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Vip
7  * @since 1.0
8 */
9 public class Vip {
10     private String email;
11
12     public void setEmail(String email) {
13         this.email = email;
14     }
15
16     @Override
17     public String toString() {
18         return "Vip{" +
19             "email='" + email + '\'' +
20             '}';
21     }
22 }
23
```

spring-null.xml

XML | 复制代码

```
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
5                         http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <bean id="vipBean" class="com.powernode.spring6.beans.Vip">
8         <!--空串的第一种方式-->
9         <!--<property name="email" value="" />-->
10        <!--空串的第二种方式-->
11        <property name="email">
12            <value/>
13        </property>
14    </bean>
15 </beans>
```

测试程序

Java | 复制代码

```
1 @Test
2 public void testNull(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationCont
ext("spring-null.xml");
4     Vip vipBean = applicationContext.getBean("vipBean", Vip.class);
5     System.out.println(vipBean);
6 }
```

执行结果：

```
Tests passed: 1 of 1 test – 297 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
Vip{email=''}
动力节点
```

- 怎么注入null呢？

第一种方式：不给属性赋值

spring-null.xml

XML | 复制代码

```
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 htt
p://www.springframework.org/schema/beans/spring-beans.xsd">
5
6     <bean id="vipBean" class="com.powernode.spring6.beans.Vip" />
7
8 </beans>
```

执行结果：

```
Tests passed: 1 of 1 test – 287 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
Vip{email='null'}
动力节点
```

第二种方式：使用<null/>

```
spring-null.xml XML | 复制代码

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
6     <bean id="vipBean" class="com.powernode.spring6.beans.Vip">
7       <property name="email">
8         <null/>
9       </property>
10      </bean>
11
12    </beans>
```

执行结果：

```
Tests passed: 1 of 1 test – 287 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
Vip{email='null'}
```

4.3.11 注入的值中含有特殊符号

XML中有5个特殊字符，分别是：<、>、'、"、&

以上5个特殊符号在XML中会被特殊对待，会被当做XML语法的一部分进行解析，如果这些特殊符号直接出现在注入的字符串当中，会报错。

The screenshot shows an XML configuration file in an IDE. The code is as follows:

```
<?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.
<bean id="mathBean" class="com.powernode.spring6.beans.Math">
    <property name="result" value="2 < 3"/>
</bean>
</beans>
```

A tooltip is displayed over the line `<property name="result" value="2 < 3"/>`, containing the message: "The value of attribute "value" associated with an element type "property" must not contain the '<' character." Below the tooltip, there is a brief description of the `property` tag and its usage.

解决方案包括两种：

- 第一种：特殊符号使用转义字符代替。
- 第二种：将含有特殊符号的字符串放到：`<![CDATA[]]>`当中。因为放在CDATA区中的数据不会被XML文件解析器解析。

5个特殊字符对应的转义字符分别是：

特殊字符	转义字符
>	>
<	<
'	'
"	"
&	&

先使用转义字符来代替：

Math

Java | 复制代码

```
1 package com.powernode.spring6.beans;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Math
7  * @since 1.0
8 */
9 public class Math {
10     private String result;
11
12     public void setResult(String result) {
13         this.result = result;
14     }
15
16     @Override
17     public String toString() {
18         return "Math{" +
19                 "result='" + result + '\'' +
20                 '}';
21     }
22 }
23
```

spring-special.xml

XML | 复制代码

```
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     <bean id="mathBean" class="com.powernode.spring6.beans.Math">
6         <property name="result" value="2 < 3"/>
7     </bean>
8 </beans>
```

动力节点

动力节点

测试程序

Java | 复制代码

```
1 @Test
2 public void testSpecial(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationCont
ext("spring-special.xml");
4     Math mathBean = applicationContext.getBean("mathBean", Math.class);
5     System.out.println(mathBean);
6 }
```

执行结果：

```
✓ Tests passed: 1 of 1 test – 297 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
Math{result='2 < 3'}
```

动力节点

我们再来使用CDATA方式：

spring-special.xml

XML | 复制代码

```
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
6     <bean id="mathBean" class="com.powernode.spring6.beans.Math">
7         <property name="result">
8             <!--只能使用value标签-->
9             <value><! [CDATA[2 < 3 ]]></value>
10        </property>
11    </bean>
12
13 </beans>
```

注意：使用CDATA时，不能使用value属性，只能使用value标签。

执行结果：

```
✓ Tests passed: 1 of 1 test – 281 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
Math{result='2 < 3'}
```

动力节点

4.4 p命名空间注入

目的：简化配置。

使用p命名空间注入的前提条件包括两个：

- 第一：在XML头部信息中添加p命名空间的配置信息：

xmlns:p="http://www.springframework.org/schema/p"

- 第二：p命名空间注入是基于setter方法的，所以需要对应的属性提供setter方法。

```
▼ Customer Java | 复制代码

1 package com.powernode.spring6.beans;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Customer
7  * @since 1.0
8 */
9 public class Customer {
10     private String name;
11     private int age;
12
13     public void setName(String name) {
14         this.name = name;
15     }
16
17     public void setAge(int age) {
18         this.age = age;
19     }
20
21     @Override
22     public String toString() {
23         return "Customer{" +
24             "name='" + name + '\'' +
25             ", age=" + age +
26             '}';
27     }
28 }
29
```

spring-p.xml

XML | 复制代码

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:p="http://www.springframework.org/schema/p"
4       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <bean id="customerBean" class="com.powernode.spring6.beans.Customer" p:
8       name="zhangsan" p:age="20"/>
9   </beans>
```

测试程序

Java | 复制代码

```
1 @Test
2 public void testP(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationCont
4         ext("spring-p.xml");
5     Customer customerBean = applicationContext.getBean("customerBean", Cust
6     omer.class);
7     System.out.println(customerBean);
8 }
```

执行结果：

✓ Tests passed: 1 of 1 test – 328 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

Customer{name='zhangsan', age=20}

动力节点

把setter方法去掉：

✖ Tests failed: 1 of 1 test – 329 ms

```
ans.NotWritablePropertyException: Invalid property 'age' of bean class [com.powernode.spring6.
spring6.beans.Customer]: Bean property 'age' is not writable or has an invalid setter method. D
          动力节点
```

所以p命名空间实际上是对set注入的简化。

4.5 c命名空间注入

动力节点

c命名空间是简化构造方法注入的。

使用c命名空间的两个前提条件：

第一：需要在xml配置文件头部添加信息： xmlns:c="http://www.springframework.org/schema/c"

第二：需要提供构造方法。

```
MyTime
Java | 复制代码

1 package com.powernode.spring6.beans;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className MyTime
7  * @since 1.0
8 */
9 public class MyTime {
10     private int year;
11     private int month;
12     private int day;
13
14     public MyTime(int year, int month, int day) {
15         this.year = year;
16         this.month = month;
17         this.day = day;
18     }
19
20     @Override
21     public String toString() {
22         return "MyTime{" +
23                 "year=" + year +
24                 ", month=" + month +
25                 ", day=" + day +
26                 '}';
27     }
28 }
29
```

动力节点

spring-c.xml

XML | 复制代码

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:c="http://www.springframework.org/schema/c"
4       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <!--<bean id="myTimeBean" class="com.powernode.spring6.beans.MyTime"
8           c:year="1970" c:month="1" c:day="1"/><!--&gt;
9
10    &lt;bean id="myTimeBean" class="com.powernode.spring6.beans.MyTime" c:_0=
11      "2008" c:_1="8" c:_2="8"/&gt;
12
13  &lt;/beans&gt;</pre>

```

测试程序

Java | 复制代码

```

1 @Test
2 public void testC(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-c.xml");
4     MyTime myTimeBean = applicationContext.getBean("myTimeBean", MyTime.class);
5     System.out.println(myTimeBean);
6 }
```

执行结果：

Tests passed: 1 of 1 test – 305 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

MyTime{year=2008, month=8, day=8}

动力节点

节点

把构造方法注释掉：

Tests failed: 1 of 1 test – 296 ms

Exception: Error creating bean with name 'myTimeBean' defined in class path resource [spring-c.xml]: Could not resolve matching constructor on bean class [com.powernode.spring6.beans.MyTime].

动力节点

所以，c命名空间是依靠构造方法的。

注意：不管是p命名空间还是c命名空间，注入的时候都可以注入简单类型以及非简单类型。

4.6 util命名空间

使用util命名空间可以让配置复用。

使用util命名空间的前提是：在spring配置文件头部添加配置信息。如下：

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

MyDataSource1

Java | 复制代码

```
1 package com.powernode.spring6.beans;
2
3 import java.util.Properties;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className MyDataSource1
9  * @since 1.0
10 */
11 public class MyDataSource1 {
12     private Properties properties;
13
14     public void setProperties(Properties properties) {
15         this.properties = properties;
16     }
17
18     @Override
19     public String toString() {
20         return "MyDataSource1{" +
21                 "properties=" + properties +
22                 '}';
23     }
24 }
25
```

MyDataSource2

Java | 复制代码

```
1 package com.powernode.spring6.beans;
2
3 import java.util.Properties;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className MyDataSource2
9  * @since 1.0
10 */
11 public class MyDataSource2 {
12     private Properties properties;
13
14     public void setProperties(Properties properties) {
15         this.properties = properties;
16     }
17
18     @Override
19     public String toString() {
20         return "MyDataSource2{" +
21                 "properties=" + properties +
22                 '}';
23     }
24 }
25
```

spring-util.xml

XML | 复制代码

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xmlns:util="http://www.springframework.org/schema/util"
5         xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
6                         http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd">
7
8     <util:properties id="prop">
9         <prop key="driver">com.mysql.cj.jdbc.Driver</prop>
10        <prop key="url">jdbc:mysql://localhost:3306/spring</prop>
11        <prop key="username">root</prop>
12        <prop key="password">123456</prop>
13    </util:properties>
14
15    <bean id="dataSource1" class="com.powernode.spring6.beans.MyDataSource1">
16        <property name="properties" ref="prop"/>
17    </bean>
18
19    <bean id="dataSource2" class="com.powernode.spring6.beans.MyDataSource2">
20        <property name="properties" ref="prop"/>
21    </bean>
22 </beans>
```

测试程序

Java | 复制代码

```
1 @Test
2 public void testUtil(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-util.xml");
4
5     MyDataSource1 dataSource1 = applicationContext.getBean("dataSource1",
6     MyDataSource1.class);
7     System.out.println(dataSource1);
8
9     MyDataSource2 dataSource2 = applicationContext.getBean("dataSource2",
10    MyDataSource2.class);
11    System.out.println(dataSource2);
12 }
```

执行结果：

```
✓ Tests passed: 1 of 1 test - 360 ms  
C:\dev\Java\jdk-17.0.4\bin\java.exe ...  
MyDataSource1{properties={password=123456, driver=com.mysql.cj.jdbc.Driver, url=jdbc:mysql://localhost:3306/spring, username=root}}  
MyDataSource2{properties={password=123456, driver=com.mysql.cj.jdbc.Driver, url=jdbc:mysql://localhost:3306/spring, username=root}}
```

4.7 基于XML的自动装配

Spring还可以完成自动化的注入，自动化注入又被称为自动装配。它可以根据**名字**进行自动装配，也可以根据**类型**进行自动装配。

4.7.1 根据名称自动装配

```
▼ UserDao  
1 package com.powernode.spring6.dao;  
2  
3 /**  
4 * @author 动力节点  
5 * @version 1.0  
6 * @className UserDao  
7 * @since 1.0  
8 */  
9 public class UserDao {  
10  
11     public void insert(){  
12         System.out.println("正在保存用户数据。");  
13     }  
14 }  
15
```

Java | 复制代码

UserService

Java | 复制代码

```

1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className UserService
9  * @since 1.0
10 */
11 public class UserService {
12
13     private UserDao aaa;
14
15     // 这个set方法非常关键
16     public void setAaa(UserDao aaa) {
17         this.aaa = aaa;
18     }
19
20     public void save(){
21         aaa.insert();
22     }
23 }
24

```

Spring的配置文件这样配置：

spring-autowire.xml

XML | 复制代码

```

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
5                         http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <bean id="userService" class="com.powernode.spring6.service.UserService"
8           autowire="byName"/>
9
10    <bean id="aaa" class="com.powernode.spring6.dao.UserDao"/>
11
12 </beans>

```

这个配置起到关键作用：

- UserService Bean中需要添加autowire="byName"， 表示通过名称进行装配。

- UserService类中有一个UserDao属性，而UserDao属性的名字是aaa，**对应的set方法是setAaa()**，正好和UserDao Bean的id是一样的。这就是根据名称自动装配。

```
▼ 测试程序 Java | 复制代码

1 @Test
2 public void testAutowireByName(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-autowire.xml");
4     UserService userService = applicationContext.getBean("userService", UserService.class);
5     userService.save();
6 }
```

执行结果：

```
Tests passed: 1 of 1 test – 328 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
正在保存用户数据。
动力节点
```

我们来测试一下，byName装配是和属性名有关还是和set方法名有关系：

```

1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className UserService
9  * @since 1.0
10 */
11 public class UserService {
12     // 这里没修改
13     private UserDao aaa;
14
15     /*public void setAaa(UserDao aaa) {
16         this.aaa = aaa;
17     }*/
18
19     // set方法名变化了
20     public void setDao(UserDao aaa){
21         this.aaa = aaa;
22     }
23
24     public void save(){
25         aaa.insert();
26     }
27 }
28

```

在执行测试程序：

```

Tests failed: 1 of 1 test - 349 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...

java.lang.NullPointerException: Cannot invoke "com.powernode.spring6.dao.UserDao.insert()" because "this.aaa" is null
    at com.powernode.spring6.service.UserService.save(UserService.java:30)
    at com.powernode.spring6.test.DITest.testAutowireByName(DITest.java:23) <27 internal lines>

```

通过测试得知，aaa属性并没有赋值成功。也就是并没有装配成功。

我们将spring配置文件修改以下：

```
spring-autowire.xml
```

XML | 复制代码

```
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
6   <bean id="userService" class="com.powernode.spring6.service.UserService"
7         autowire="byName"/>
8   <!--这个id修改了-->
9   <bean id="dao" class="com.powernode.spring6.dao.UserDao"/>
10
11 </beans>
```

执行测试程序：



这说明，如果根据名称装配(byName)，底层会调用set方法进行注入。

例如：setAge() 对应的名字是age， setPassword() 对应的名字是password， setEmail() 对应的名字是email。

4.7.2 根据类型自动装配

AccountDao

Java | 复制代码

```
1 package com.powernode.spring6.dao;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className AccountDao
7  * @since 1.0
8 */
9 public class AccountDao {
10    public void insert(){
11        System.out.println("正在保存账户信息");
12    }
13}
14
```

AccountService

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.AccountDao;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className AccountService
9  * @since 1.0
10 */
11 public class AccountService {
12     private AccountDao accountDao;
13
14     public void setAccountDao(AccountDao accountDao) {
15         this.accountDao = accountDao;
16     }
17
18     public void save(){
19         accountDao.insert();
20     }
21 }
22
```

spring-autowire.xml

XML | 复制代码

```

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
6     <!--byType表示根据类型自动装配-->
7     <bean id="accountService" class="com.powernode.spring6.service.AccountService" autowire="byType"/>
8
9     <bean class="com.powernode.spring6.dao.AccountDao"/>
10
11 </beans>

```

测试程序

Java | 复制代码

```

1 @Test
2 public void testAutowireByType(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-autowire.xml");
4     AccountService accountService = applicationContext.getBean("accountService", AccountService.class);
5     accountService.save();
6 }

```

执行结果：

Tests passed: 1 of 1 test – 333 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

正在保存账户信息

动力节点

我们把UserService中的set方法注释掉，再执行：

Tests failed: 1 of 1 test – 325 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

java.lang.NullPointerException: Cannot invoke "com.powernode.spring6.dao.AccountDao.insert()" because "this.accountDao" is null

at com.powernode.spring6.service.AccountService.save(AccountService.java:19)

at com.powernode.spring6.test.DITest.testAutowireByType(DITest.java:24) <27 internal lines>

动力节点

可以看到无论是byName还是byType，在装配的时候都是基于set方法的。所以set方法是必须要提供的。提供构造方法是不行的，大家可以测试一下。这里就不再赘述。

如果byType，根据类型装配时，如果配置文件中有两个类型一样的bean会出现什么问题呢？

```
spring-autowire.xml XML 复制代码

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
6     <bean id="accountService" class="com.powernode.spring6.service.AccountService" autowire="byType"/>
7
8     <bean id="x" class="com.powernode.spring6.dao.AccountDao"/>
9     <bean id="y" class="com.powernode.spring6.dao.AccountDao"/>
10
11 </beans>
```

执行测试程序：

```
Tests failed: 1 of 1 test - 361 ms
@org.junit.Test
public void testAutowireByType() {
    ApplicationContext context = new ClassPathXmlApplicationContext("classpath:beans.xml");
    AccountService accountService = context.getBean("accountService");
    AccountDao accountDao = accountService.getAccountDao();
    assertEquals("x", accountDao.getId());
}

@Test
public void testAutowireByType() {
    ApplicationContext context = new ClassPathXmlApplicationContext("classpath:beans.xml");
    AccountService accountService = context.getBean("accountService");
    AccountDao accountDao = accountService.getAccountDao();
    assertEquals("x", accountDao.getId());
}
```

测试结果说明了，当byType进行自动装配的时候，配置文件中某种类型的Bean必须是唯一的，不能出现多个。

4.8 spring引入外部属性配置文件

我们都知道编写数据源的时候是需要连接数据库的信息的，例如：driver url username password等信息。这些信息可以单独写到一个属性配置文件中吗，这样用户修改起来会更加的方便。当然可以。

第一步：写一个数据源类，提供相关属性。

```
1 package com.powernode.spring6.beans;
2
3 import javax.sql.DataSource;
4 import java.io.PrintWriter;
5 import java.sql.Connection;
6 import java.sql.SQLException;
7 import java.sql.SQLFeatureNotSupportedException;
8 import java.util.logging.Logger;
9
10 /**
11  * @author 动力节点
12  * @version 1.0
13  * @className MyDataSource
14  * @since 1.0
15 */
16 public class MyDataSource implements DataSource {
17     @Override
18     public String toString() {
19         return "MyDataSource{" +
20                 "driver='" + driver + '\'' +
21                 ", url='" + url + '\'' +
22                 ", username='" + username + '\'' +
23                 ", password='" + password + '\'' +
24                 '}';
25     }
26
27     private String driver;
28     private String url;
29     private String username;
30     private String password;
31
32     public void setDriver(String driver) {
33         this.driver = driver;
34     }
35
36     public void setUrl(String url) {
37         this.url = url;
38     }
39
40     public void setUsername(String username) {
41         this.username = username;
42     }
43
44     public void setPassword(String password) {
45         this.password = password;
```

```
46     }
47     //.....
48 }
49 }
50 }
```

第二步：在类路径下新建jdbc.properties文件，并配置信息。

▼ jdbc.properties

Properties | 复制代码

```
1 driver=com.mysql.cj.jdbc.Driver
2 url=jdbc:mysql://localhost:3306/spring
3 username=root
4 password=root123
```

第三步：在spring配置文件中引入context命名空间。

▼ spring-properties.xml

XML | 复制代码

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
6           http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">
7
8 </beans>
```

第四步：在spring中配置使用jdbc.properties文件。

spring-properties.xml

XML | 复制代码

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
6           http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">
7
8   <context:property-placeholder location="jdbc.properties"/>
9
10  <bean id="dataSource" class="com.powernode.spring6.beans.MyDataSource">
11    <property name="driver" value="${driver}"/>
12    <property name="url" value="${url}"/>
13    <property name="username" value="${username}"/>
14    <property name="password" value="${password}"/>
15  </bean>
16 </beans>
```

测试程序：

```
1 @Test
2 public void testProperties(){
3   ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-properties.xml");
4   MyDataSource dataSource = applicationContext.getBean("dataSource", MyDataSource.class);
5   System.out.println(dataSource);
6 }
```

执行结果：

```
Tests passed: 1 of 1 test - 377 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
MyDataSource{driver='com.mysql.cj.jdbc.Driver', url='jdbc:mysql://localhost:3306/spring', username='Administrator', password='root123'}
```

五、Bean的作用域

5.1 singleton

默认情况下，Spring的IoC容器创建的Bean对象是单例的。来测试一下：

```
1 package com.powernode.spring6.beans;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className SpringBean
7  * @since 1.0
8 */
9 public class SpringBean {
10 }
11
```

Java | 复制代码

```
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
6     <bean id="sb" class="com.powernode.spring6.beans.SpringBean" />
7
8 </beans>
```

XML | 复制代码

```
1 @Test
2 public void testScope(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-scope.xml");
4
5     SpringBean sb1 = applicationContext.getBean("sb", SpringBean.class);
6     System.out.println(sb1);
7
8     SpringBean sb2 = applicationContext.getBean("sb", SpringBean.class);
9     System.out.println(sb2);
10 }
```

Java | 复制代码

执行结果：

```
Tests passed: 1 of 1 test – 266 ms  
C:\dev\Java\jdk-17.0.4\bin\java.exe ...  
com.powernode.spring6.beans.SpringBean@5b239d7d  
com.powernode.spring6.beans.SpringBean@5b239d7d  
动力节点
```

通过测试得知：Spring的IoC容器中，默认情况下，Bean对象是单例的。

这个对象在什么时候创建的呢？可以为SpringBean提供一个无参数构造方法，测试一下，如下：

```
SpringBean  
Java | 复制代码  
1 package com.powernode.spring6.beans;  
2  
3 /**  
4 * @author 动力节点  
5 * @version 1.0  
6 * @className SpringBean  
7 * @since 1.0  
8 **/  
9 public class SpringBean {  
10     public SpringBean() {  
11         System.out.println("SpringBean的无参数构造方法执行。");  
12     }  
13 }  
14
```

将测试程序中getBean()所在行代码注释掉：

```
测试程序  
Java | 复制代码  
1 @Test  
2 public void testScope(){  
3     ApplicationContext applicationContext = new ClassPathXmlApplicationCont  
ext("spring-scope.xml");  
4 }
```

执行测试程序：

```
Tests passed: 1 of 1 test – 262 ms  
C:\dev\Java\jdk-17.0.4\bin\java.exe ...  
SpringBean的无参数构造方法执行。  
动力节点
```

通过测试得知， 默认情况下， Bean对象的创建是在初始化Spring上下文的时候就完成的。

5.2 prototype

如果想让Spring的Bean对象以多例的形式存在， 可以在bean标签中指定scope属性的值为：

prototype， 这样Spring会在每一次执行getBean()方法的时候创建Bean对象， 调用几次则创建几次。

spring-scope.xml

XML | 复制代码

```
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
6   <bean id="sb" class="com.powernode.spring6.beans.SpringBean" scope="prototype" />
7
8 </beans>
```

测试程序

Java | 复制代码

```
1 @Test
2 public void testScope(){
3   ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-scope.xml");
4
5   SpringBean sb1 = applicationContext.getBean("sb", SpringBean.class);
6   System.out.println(sb1);
7
8   SpringBean sb2 = applicationContext.getBean("sb", SpringBean.class);
9   System.out.println(sb2);
10 }
```

执行结果：

```
✓ Tests passed: 1 of 1 test – 272 ms
```

```
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
```

SpringBean的无参数构造方法执行。

```
com.powernode.spring6.beans.SpringBean@294425a7
```

SpringBean的无参数构造方法执行。

```
com.powernode.spring6.beans.SpringBean@67d48005动力节点
```

我们可以把测试代码中的getBean()方法所在行代码注释掉：

▼ 测试代码

Java | 复制代码

```
1 @Test
2 public void testScope(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationCont
ext("spring-scope.xml");
4 }
```

执行结果：

```
✓ Tests passed: 1 of 1 test – 317 ms
```

```
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
```

```
Process finished with exit code 0
```

动力节点

可以看到这一次在初始化Spring上下文的时候，并没有创建Bean对象。

那你可能会问：scope如果没有配置，它的默认值是什么呢？默认值是singleton，单例的。

▼ spring-scope.xml

XML | 复制代码

```
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
6     <bean id="sb" class="com.powernode.spring6.beans.SpringBean" scope="sin
gleton" />
7
8 </beans>
```

```
测试程序 Java | 复制代码

1  @Test
2  public void testScope(){
3      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-scope.xml");
4
5      SpringBean sb1 = applicationContext.getBean("sb", SpringBean.class);
6      System.out.println(sb1);
7
8      SpringBean sb2 = applicationContext.getBean("sb", SpringBean.class);
9      System.out.println(sb2);
10 }
```

执行结果：

```
Tests passed: 1 of 1 test – 265 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
SpringBean的无参数构造方法执行。
com.powernode.spring6.beans.SpringBean@5b239d7d
com.powernode.spring6.beans.SpringBean@5b239d7d
```

通过测试得知，没有指定scope属性时，默认是singleton单例的。

5.3 其它scope

scope属性的值不止两个，它一共包括8个选项：

- singleton：默认的，单例。
- prototype：原型。每调用一次getBean()方法则获取一个新的Bean对象。或每次注入的时候都是新对象。
- request：一个请求对应一个Bean。**仅限于在WEB应用中使用。**
- session：一个会话对应一个Bean。**仅限于在WEB应用中使用。**
- global session：**portlet应用中专用的。**如果在Servlet的WEB应用中使用global session的话，和session一个效果。（portlet和servlet都是规范。servlet运行在servlet容器中，例如Tomcat。portlet运行在portlet容器中。）
- application：一个应用对应一个Bean。**仅限于在WEB应用中使用。**
- websocket：一个websocket生命周期对应一个Bean。**仅限于在WEB应用中使用。**

- 自定义scope：很少使用。

接下来咱们自定义一个Scope，线程级别的Scope，在同一个线程中，获取的Bean都是同一个。跨线程则是不同的对象：（以下内容作为了解）

- 第一步：自定义Scope。（实现Scope接口）
 - spring内置了线程范围的类：org.springframework.context.support.SimpleThreadScope，可以直接用。
- 第二步：将自定义的Scope注册到Spring容器中。

```
▼ spring-scope.xml XML | 复制代码

1 ▼ <bean class="org.springframework.beans.factory.config.CustomScopeConfigure
r">
2 ▼   <property name="scopes">
3   <map>
4     <entry key="myThread">
5       <bean class="org.springframework.context.support.SimpleThreadScope"
/>
6     </entry>
7   </map>
8   </property>
9 </bean>
```

- 第三步：使用Scope。

```
▼ spring-scope.xml XML | 复制代码

1 <bean id="sb" class="com.powernode.spring6.beans.SpringBean" scope="myThrea
d" />
```

编写测试程序：

```

1  @Test
2  public void testCustomScope(){
3      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-scope.xml");
4      SpringBean sb1 = applicationContext.getBean("sb", SpringBean.class);
5      SpringBean sb2 = applicationContext.getBean("sb", SpringBean.class);
6      System.out.println(sb1);
7      System.out.println(sb2);
8      // 启动线程
9      new Thread(new Runnable() {
10          @Override
11          public void run() {
12              SpringBean a = applicationContext.getBean("sb", SpringBean.class);
13              SpringBean b = applicationContext.getBean("sb", SpringBean.class);
14              System.out.println(a);
15              System.out.println(b);
16          }
17      }).start();
18  }

```

执行结果:

Tests passed: 1 of 1 test – 454 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

SpringBean的无参数构造方法执行。

com.powernode.spring6.beans.SpringBean@6ed3ccb2

com.powernode.spring6.beans.SpringBean@6ed3ccb2

SpringBean的无参数构造方法执行。

com.powernode.spring6.beans.SpringBean@4a4607ee

com.powernode.spring6.beans.SpringBean@4a4607ee

六、GoF之工厂模式

- 设计模式：一种可以被重复利用的解决方案。

- GoF (Gang of Four) , 中文名——四人组。
- 《Design Patterns: Elements of Reusable Object–Oriented Software》 (即《设计模式》一书) , 1995年由 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 合著。这几位作者常被称为"四人组 (Gang of Four) "。
- 该书中描述了23种设计模式。我们平常所说的设计模式就是指这23种设计模式。
- 不过除了GoF23种设计模式之外, 还有其它的设计模式, 比如: JavaEE的设计模式 (DAO模式、MVC模式等) 。
- GoF23种设计模式可分为三大类:
 - **创建型** (5个) : 解决对象创建问题。
 - 单例模式
 - 工厂方法模式
 - 抽象工厂模式
 - 建造者模式
 - 原型模式
 - **结构型** (7个) : 一些类或对象组合在一起的经典结构。
 - 代理模式
 - 装饰模式
 - 适配器模式
 - 组合模式
 - 享元模式
 - 外观模式
 - 桥接模式
 - **行为型** (11个) : 解决类或对象之间的交互问题。
 - 策略模式
 - 模板方法模式
 - 责任链模式
 - 观察者模式
 - 迭代子模式
 - 命令模式
 - 备忘录模式
 - 状态模式
 - 访问者模式

- 中介者模式
 - 解释器模式
- 工厂模式是解决对象创建问题的，所以工厂模式属于创建型设计模式。这里为什么学习工厂模式呢？这是因为Spring框架底层使用了大量的工厂模式。

6.1 工厂模式的三种形态

工厂模式通常有三种形态：

- 第一种：简单工厂模式（Simple Factory）：不属于23种设计模式之一。简单工厂模式又叫做：静态工厂方法模式。简单工厂模式是工厂方法模式的一种特殊实现。
- 第二种：工厂方法模式（Factory Method）：是23种设计模式之一。
- 第三种：抽象工厂模式（Abstract Factory）：是23种设计模式之一。

6.2 简单工厂模式

简单工厂模式的角色包括三个：

- 抽象产品 角色
- 具体产品 角色
- 工厂类 角色

简单工厂模式的代码如下：

抽象产品角色：

Weapon

Java | 复制代码

```
1 package com.powernode.factory;
2
3 /**
4  * 武器 (抽象产品角色)
5  * @author 动力节点
6  * @version 1.0
7  * @className Weapon
8  * @since 1.0
9  */
10 public abstract class Weapon {
11 /**
12  * 所有的武器都有攻击行为
13 */
14 public abstract void attack();
15 }
```

具体产品角色：

Tank

Java | 复制代码

```
1 package com.powernode.factory;
2
3 /**
4  * 坦克 (具体产品角色)
5  * @author 动力节点
6  * @version 1.0
7  * @className Tank
8  * @since 1.0
9  */
10 public class Tank extends Weapon{
11     @Override
12     public void attack() {
13         System.out.println("坦克开炮! ");
14     }
15 }
16
```

动力节点

动力节点

Fighter

Java | 复制代码

```
1 package com.powernode.factory;
2
3 /**
4  * 战斗机 (具体产品角色)
5  * @author 动力节点
6  * @version 1.0
7  * @className Fighter
8  * @since 1.0
9  */
10 public class Fighter extends Weapon{
11     @Override
12     public void attack() {
13         System.out.println("战斗机投下原子弹! ");
14     }
15 }
16
```

Dagger

Java | 复制代码

```
1 package com.powernode.factory;
2
3 /**
4  * 匕首 (具体产品角色)
5  * @author 动力节点
6  * @version 1.0
7  * @className Dagger
8  * @since 1.0
9  */
10 public class Dagger extends Weapon{
11     @Override
12     public void attack() {
13         System.out.println("砍他丫的! ");
14     }
15 }
16
```

工厂类角色：

WeaponFactory

Java | 复制代码

```
1 package com.powernode.factory;
2
3 /**
4  * 工厂类角色
5  * @author 动力节点
6  * @version 1.0
7  * @className WeaponFactory
8  * @since 1.0
9 */
10 public class WeaponFactory {
11     /**
12      * 根据不同的武器类型生产武器
13      * @param weaponType 武器类型
14      * @return 武器对象
15     */
16     public static Weapon get(String weaponType){
17         if (weaponType == null || weaponType.trim().length() == 0) {
18             return null;
19         }
20         Weapon weapon = null;
21         if ("TANK".equals(weaponType)) {
22             weapon = new Tank();
23         } else if ("FIGHTER".equals(weaponType)) {
24             weapon = new Fighter();
25         } else if ("DAGGER".equals(weaponType)) {
26             weapon = new Dagger();
27         } else {
28             throw new RuntimeException("不支持该武器！");
29         }
30         return weapon;
31     }
32 }
33
```

测试程序（客户端程序）：

动力节点

动力节点

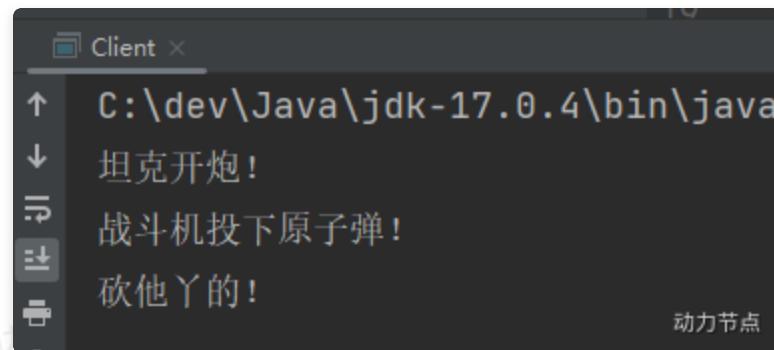
动力节点

```

1 package com.powernode.factory;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Client
7  * @since 1.0
8 */
9 public class Client {
10    public static void main(String[] args) {
11        Weapon weapon1 = WeaponFactory.get("TANK");
12        weapon1.attack();
13
14        Weapon weapon2 = WeaponFactory.get("FIGHTER");
15        weapon2.attack();
16
17        Weapon weapon3 = WeaponFactory.get("DAGGER");
18        weapon3.attack();
19    }
20}
21

```

执行结果：



```

Client x
C:\dev\Java\jdk-17.0.4\bin\java
坦克开炮!
战斗机投下原子弹!
砍他丫的!

```

简单工厂模式的优点：

- 客户端程序不需要关心对象的创建细节，需要哪个对象时，只需要向工厂索要即可，初步实现了责任的分离。客户端只负责“消费”，工厂负责“生产”。生产和消费分离。

简单工厂模式的缺点：

- 缺点1：工厂类集中了所有产品的创造逻辑，形成一个无所不知的全能类，有人把它叫做上帝类。显然工厂类非常关键，不能出问题，一旦出问题，整个系统瘫痪。
- 缺点2：不符合OCP开闭原则，在进行系统扩展时，需要修改工厂类。

Spring中的BeanFactory就使用了简单工厂模式。

6.3 工厂方法模式

工厂方法模式既保留了简单工厂模式的优点，同时又解决了简单工厂模式的缺点。

工厂方法模式的角色包括：

- 抽象工厂角色
- 具体工厂角色
- 抽象产品角色
- 具体产品角色

代码如下：

```
▼ 抽象产品角色 Java | 复制代码

1 package com.powernode.factory;
2
3 /**
4  * 武器类（抽象产品角色）
5  * @author 动力节点
6  * @version 1.0
7  * @className Weapon
8  * @since 1.0
9  */
10 public abstract class Weapon {
11     /**
12      * 所有武器都有攻击行为
13      */
14     public abstract void attack();
15 }
16
```

具体产品角色

Java | 复制代码

```
1 package com.powernode.factory;
2
3 /**
4  * 具体产品角色
5  * @author 动力节点
6  * @version 1.0
7  * @className Gun
8  * @since 1.0
9 */
10 public class Gun extends Weapon{
11     @Override
12     public void attack() {
13         System.out.println("开枪射击! ");
14     }
15 }
16
```

具体产品角色

Java | 复制代码

```
1 package com.powernode.factory;
2
3 /**
4  * 具体产品角色
5  * @author 动力节点
6  * @version 1.0
7  * @className Fighter
8  * @since 1.0
9 */
10 public class Fighter extends Weapon{
11     @Override
12     public void attack() {
13         System.out.println("战斗机发射核弹! ");
14     }
15 }
16
```

▼ 抽象工厂角色

Java | 复制代码

```
1 package com.powernode.factory;
2
3 /**
4  * 武器工厂接口(抽象工厂角色)
5  * @author 动力节点
6  * @version 1.0
7  * @className WeaponFactory
8  * @since 1.0
9  */
10 public interface WeaponFactory {
11     Weapon get();
12 }
13
```

▼ 具体工厂角色

Java | 复制代码

```
1 package com.powernode.factory;
2
3 /**
4  * 具体工厂角色
5  * @author 动力节点
6  * @version 1.0
7  * @className GunFactory
8  * @since 1.0
9  */
10 public class GunFactory implements WeaponFactory{
11     @Override
12     public Weapon get() {
13         return new Gun();
14     }
15 }
16
```

▼ 具体工厂角色

Java | 复制代码

```
1 package com.powernode.factory;
2
3 /**
4  * 具体工厂角色
5  * @author 动力节点
6  * @version 1.0
7  * @className FighterFactory
8  * @since 1.0
9 */
10 public class FighterFactory implements WeaponFactory{
11     @Override
12     public Weapon get() {
13         return new Fighter();
14     }
15 }
16
```

客户端程序：

▼

Java | 复制代码

```
1 package com.powernode.factory;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Client
7  * @since 1.0
8 */
9 public class Client {
10     public static void main(String[] args) {
11         WeaponFactory factory = new GunFactory();
12         Weapon weapon = factory.get();
13         weapon.attack();
14
15         WeaponFactory factory1 = new FighterFactory();
16         Weapon weapon1 = factory1.get();
17         weapon1.attack();
18     }
19 }
20
```

执行客户端程序：

```
C:\dev\Java\jdk-17.0.4\bin
开枪射击!
战斗机发射核弹!
```

动力节点

如果想扩展一个新的产品，只要新增一个产品类，再新增一个该产品对应的工厂即可，例如新增：匕首

增加：具体产品角色

Java | 复制代码

```
1 package com.powernode.factory;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Dagger
7  * @since 1.0
8 */
9 public class Dagger extends Weapon{
10     @Override
11     public void attack() {
12         System.out.println("砍丫的! ");
13     }
14 }
15
```

增加：具体工厂角色

Java | 复制代码

```
1 package com.powernode.factory;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className DaggerFactory
7  * @since 1.0
8 */
9 public class DaggerFactory implements WeaponFactory{
10     @Override
11     public Weapon get() {
12         return new Dagger();
13     }
14 }
15
```

动力节点

客户端程序：

```

1 package com.powernode.factory;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Client
7  * @since 1.0
8 */
9 public class Client {
10    public static void main(String[] args) {
11        WeaponFactory factory = new GunFactory();
12        Weapon weapon = factory.get();
13        weapon.attack();
14
15        WeaponFactory factory1 = new FighterFactory();
16        Weapon weapon1 = factory1.get();
17        weapon1.attack();
18
19        WeaponFactory factory2 = new DaggerFactory();
20        Weapon weapon2 = factory2.get();
21        weapon2.attack();
22    }
23 }
24

```

执行结果如下：

```

com.powernode.factory.Client ×
↑ C:\dev\Java\jdk-17.0.4\
↓
= 开枪射击!
≡ 战斗机发射核弹!
≡ 砍丫的!

```

我们可以看到在进行功能扩展的时候，不需要修改之前的源代码，显然工厂方法模式符合OCP原则。

工厂方法模式的优点：

- 一个调用者想创建一个对象，只要知道其名称就可以了。
- 扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。
- 屏蔽产品的具体实现，调用者只关心产品的接口。

工厂方法模式的缺点：

- 每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。这并不是什么好事。

6.4 抽象工厂模式（了解）

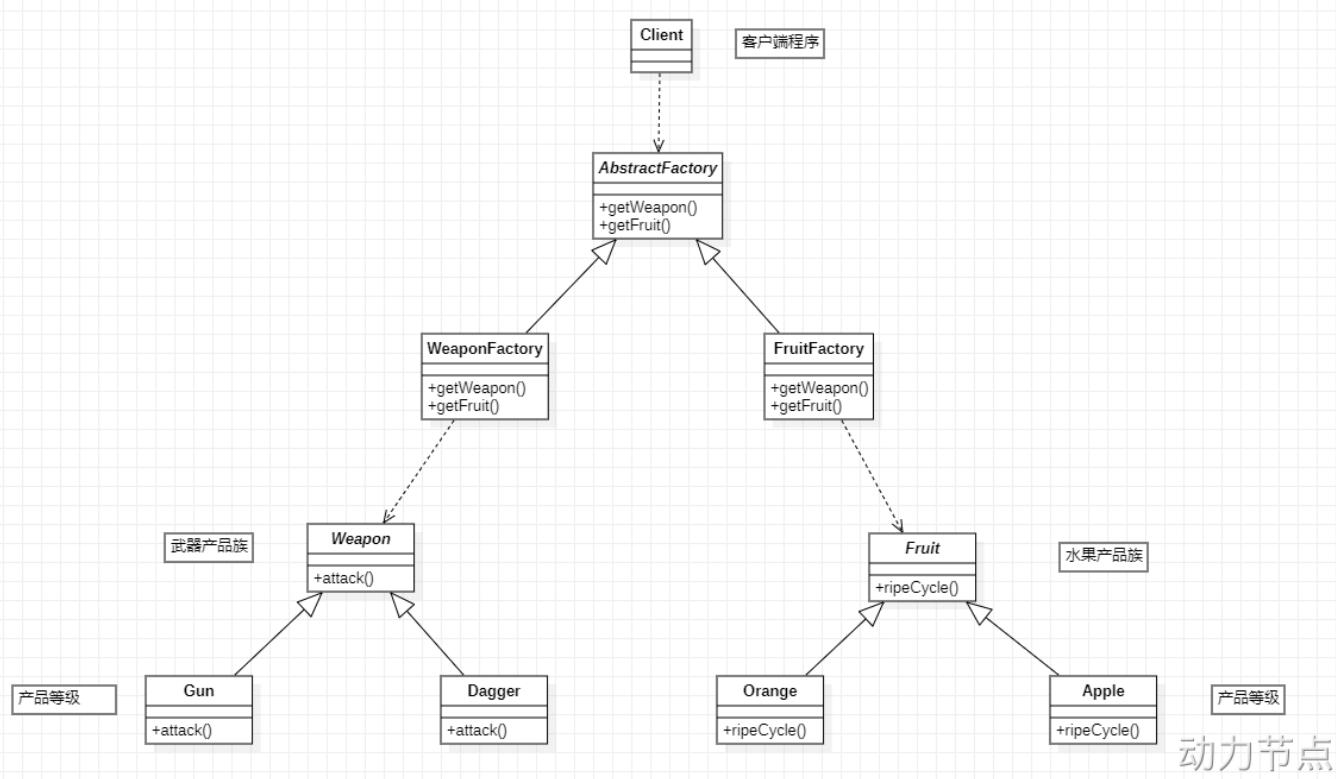
抽象工厂模式相对于工厂方法模式来说，就是工厂方法模式是针对一个产品系列的，而抽象工厂模式是针对多个产品系列的，即工厂方法模式是一个产品系列一个工厂类，而抽象工厂模式是多个产品系列一个工厂类。

抽象工厂模式特点：抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式是指当有多个抽象角色时，使用的一种工厂模式。抽象工厂模式可以向客户端提供一个接口，使客户端在不必指定产品的具体的情况下，创建多个产品族中的产品对象。它有多个抽象产品类，每个抽象产品类可以派生出多个具体产品类，一个抽象工厂类，可以派生出多个具体工厂类，每个具体工厂类可以创建多个具体产品类的实例。每一个模式都是针对一定问题的解决方案，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式针对的是多个产品等级结果。

抽象工厂中包含4个角色：

- 抽象工厂角色
- 具体工厂角色
- 抽象产品角色
- 具体产品角色

抽象工厂模式的类图如下：



抽象工厂模式代码如下：

第一部分：武器产品族

▼
Java

```

1 package com.powernode.product;
2
3 /**
4  * 武器产品族
5  * @author 动力节点
6  * @version 1.0
7  * @className Weapon
8  * @since 1.0
9 */
10 public abstract class Weapon {
11     public abstract void attack();
12 }
13

```

Java | 复制代码

```
1 package com.powernode.product;
2
3 /**
4  * 武器产品族中的产品等级1
5  * @author 动力节点
6  * @version 1.0
7  * @className Gun
8  * @since 1.0
9 */
10 public class Gun extends Weapon{
11     @Override
12     public void attack() {
13         System.out.println("开枪射击! ");
14     }
15 }
16
```

Java | 复制代码

```
1 package com.powernode.product;
2
3 /**
4  * 武器产品族中的产品等级2
5  * @author 动力节点
6  * @version 1.0
7  * @className Dagger
8  * @since 1.0
9 */
10 public class Dagger extends Weapon{
11     @Override
12     public void attack() {
13         System.out.println("砍丫的! ");
14     }
15 }
16
```

第二部分：水果产品族

Java | 复制代码

```
1 package com.powernode.product;
2
3 /**
4  * 水果产品族
5  * @author 动力节点
6  * @version 1.0
7  * @className Fruit
8  * @since 1.0
9 */
10 public abstract class Fruit {
11 /**
12  * 所有果实都有一个成熟周期。
13 */
14 public abstract void ripeCycle();
15 }
16
```

Java | 复制代码

```
1 package com.powernode.product;
2
3 /**
4  * 水果产品族中的产品等级1
5  * @author 动力节点
6  * @version 1.0
7  * @className Orange
8  * @since 1.0
9 */
10 public class Orange extends Fruit{
11     @Override
12     public void ripeCycle() {
13         System.out.println("橘子的成熟周期是10个月");
14     }
15 }
16
```

Java | 复制代码

```
1 package com.powernode.product;
2
3 /**
4  * 水果产品族中的产品等级2
5  * @author 动力节点
6  * @version 1.0
7  * @className Apple
8  * @since 1.0
9 */
10 public class Apple extends Fruit{
11     @Override
12     public void ripeCycle() {
13         System.out.println("苹果的成熟周期是8个月");
14     }
15 }
16
```

第三部分：抽象工厂类

Java | 复制代码

```
1 package com.powernode.factory;
2
3 import com.powernode.product.Fruit;
4 import com.powernode.product.Weapon;
5
6 /**
7  * 抽象工厂
8  * @author 动力节点
9  * @version 1.0
10 * @className AbstractFactory
11 * @since 1.0
12 */
13 public abstract class AbstractFactory {
14     public abstract Weapon getWeapon(String type);
15     public abstract Fruit getFruit(String type);
16 }
17
```

第四部分：具体工厂类

```
1 package com.powernode.factory;
2
3 import com.powernode.product.Dagger;
4 import com.powernode.product.Fruit;
5 import com.powernode.product.Gun;
6 import com.powernode.product.Weapon;
7
8 /**
9  * 武器族工厂
10 * @author 动力节点
11 * @version 1.0
12 * @className WeaponFactory
13 * @since 1.0
14 */
15 public class WeaponFactory extends AbstractFactory{
16
17     public Weapon getWeapon(String type){
18         if (type == null || type.trim().length() == 0) {
19             return null;
20         }
21         if ("Gun".equals(type)) {
22             return new Gun();
23         } else if ("Dagger".equals(type)) {
24             return new Dagger();
25         } else {
26             throw new RuntimeException("无法生产该武器");
27         }
28     }
29
30     @Override
31     public Fruit getFruit(String type) {
32         return null;
33     }
34 }
35
```

```
1 package com.powernode.factory;
2
3 import com.powernode.product.*;
4
5 /**
6  * 水果族工厂
7  * @author 动力节点
8  * @version 1.0
9  * @className FruitFactory
10 * @since 1.0
11 */
12 public class FruitFactory extends AbstractFactory{
13     @Override
14     public Weapon getWeapon(String type) {
15         return null;
16     }
17
18     public Fruit getFruit(String type){
19         if (type == null || type.trim().length() == 0) {
20             return null;
21         }
22         if ("Orange".equals(type)) {
23             return new Orange();
24         } else if ("Apple".equals(type)) {
25             return new Apple();
26         } else {
27             throw new RuntimeException("我家果园不产这种水果");
28         }
29     }
30 }
31
```

第五部分：客户端程序

动力节点

动力节点

```
1 package com.powernode.client;
2
3 import com.powernode.factory.AbstractFactory;
4 import com.powernode.factory.FruitFactory;
5 import com.powernode.factory.WeaponFactory;
6 import com.powernode.product.Fruit;
7 import com.powernode.product.Weapon;
8
9 /**
10 * @author 动力节点
11 * @version 1.0
12 * @className Client
13 * @since 1.0
14 */
15 public class Client {
16     public static void main(String[] args) {
17         // 客户端调用方法时只面向AbstractFactory调用方法。
18         AbstractFactory factory = new WeaponFactory(); // 注意：这里的new WeaponFactory()可以采用 简单工厂模式 进行隐藏。
19         Weapon gun = factory.getWeapon("Gun");
20         Weapon dagger = factory.getWeapon("Dagger");
21
22         gun.attack();
23         dagger.attack();
24
25         AbstractFactory factory1 = new FruitFactory(); // 注意：这里的new FruitFactory()可以采用 简单工厂模式 进行隐藏。
26         Fruit orange = factory1.getFruit("Orange");
27         Fruit apple = factory1.getFruit("Apple");
28
29         orange.ripeCycle();
30         apple.ripeCycle();
31     }
32 }
33
```

执行结果：

```
com.powernode.client.Client ×  
C:\dev\Java\jdk-17.0.4\bin\jav  
开枪射击!  
砍丫的!  
橘子的成熟周期是10个月  
苹果的成熟周期是8个月
```

动力节点

动力节点

抽象工厂模式的优缺点：

- 优点：当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。
- 缺点：产品族扩展非常困难，要增加一个系列的某一产品，既要在AbstractFactory里加代码，又要在具体的里面加代码。

七、Bean的实例化方式

Spring为Bean提供了多种实例化方式，通常包括4种方式。（也就是说在Spring中为Bean对象的创建准备了多种方案，目的是：更加灵活）

- 第一种：通过构造方法实例化
- 第二种：通过简单工厂模式实例化
- 第三种：通过factory-bean实例化
- 第四种：通过FactoryBean接口实例化

7.1 通过构造方法实例化

我们之前一直使用的就是这种方式。默认情况下，会调用Bean的无参数构造方法。

User

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className User
7  * @since 1.0
8 */
9 public class User {
10    public User() {
11        System.out.println("User类的无参数构造方法执行。");
12    }
13}
14
```

spring.xml

XML | 复制代码

```
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
5                         http://www.springframework.org/schema/beans/spring-beans.xsd">
6     <bean id="userBean" class="com.powernode.spring6.bean.User"/>
7
8 </beans>
```

```
测试程序 Java | 复制代码

1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.bean.User;
4 import org.junit.Test;
5 import org.springframework.context.ApplicationContext;
6 import org.springframework.context.support.ClassPathXmlApplicationContext;
7
8 /**
9  * @author 动力节点
10 * @version 1.0
11 * @className SpringInstantiationTest
12 * @since 1.0
13 */
14 public class SpringInstantiationTest {
15
16     @Test
17     public void testConstructor(){
18         ApplicationContext applicationContext = new ClassPathXmlApplication
19         context("spring.xml");
20         User user = applicationContext.getBean("userBean", User.class);
21         System.out.println(user);
22     }
23 }
```

执行结果：

```
Tests passed: 1 of 1 test – 421 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
User类的无参数构造方法执行。
com.powernode.spring6.bean.User@5876a9af
```

7.2 通过简单工厂模式实例化

第一步：定义一个Bean

▼ Vip

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Vip
7  * @since 1.0
8 */
9 public class Vip {
10 }
11
```

第二步：编写简单工厂模式当中的工厂类

▼ VipFactory

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className VipFactory
7  * @since 1.0
8 */
9 public class VipFactory {
10     public static Vip get(){
11         return new Vip();
12     }
13 }
14
```

第三步：在Spring配置文件中指定创建该Bean的方法（使用factory-method属性指定）

▼ spring.xml

XML | 复制代码

```
1 <bean id="vipBean" class="com.powernode.spring6.bean.VipFactory" factory-method="get"/>
```

第四步：编写测试程序

测试程序

Java | 复制代码

```
1 @Test
2 public void testSimpleFactory(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationCont
ext("spring.xml");
4     Vip vip = applicationContext.getBean("vipBean", Vip.class);
5     System.out.println(vip);
6 }
```

执行结果：

```
Tests passed: 1 of 1 test – 301 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
com.powernode.spring6.bean.Vip@79e2c065 动力节点
```

7.3 通过factory-bean实例化

这种方式本质上是：通过工厂方法模式进行实例化。

第一步：定义一个Bean

```
1 package com.powernode.spring6.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Order
7  * @since 1.0
8 */
9 public class Order {
10 }
11
```

第二步：定义具体工厂类，工厂类中定义实例方法

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className OrderFactory
7  * @since 1.0
8 */
9 public class OrderFactory {
10    public Order get(){
11        return new Order();
12    }
13}
14
```

XML | 复制代码

```
1 <bean id="orderFactory" class="com.powernode.spring6.bean.OrderFactory"/>
2 <bean id="orderBean" factory-bean="orderFactory" factory-method="get"/>
```

Java | 复制代码

```
1 @Test
2 public void testSelfFactoryBean(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
4     Order orderBean = applicationContext.getBean("orderBean", Order.class);
5     System.out.println(orderBean);
6 }
```

执行结果:

```
✓ Tests passed: 1 of 1 test – 277 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
com.powernode.spring6.bean.Order@35cabb2a
```

动力节点

7.4 通过FactoryBean接口实例化

以上的第三种方式中，factory-bean是我们自定义的，factory-method也是我们自己定义的。

在Spring中，当你编写的类直接实现FactoryBean接口之后，factory-bean不需要指定了，factory-method也不需要指定了。

factory-bean会自动指向实现FactoryBean接口的类，factory-method会自动指向getObject()方法。

第一步：定义一个Bean

```
1 package com.powernode.spring6.bean;  
2  
3 /**  
4 * @author 动力节点  
5 * @version 1.0  
6 * @className Person  
7 * @since 1.0  
8 */  
9 public class Person {  
10 }  
11
```

Java | 复制代码

第二步：编写一个类实现FactoryBean接口

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 import org.springframework.beans.factory.FactoryBean;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className PersonFactoryBean
9  * @since 1.0
10 */
11 public class PersonFactoryBean implements FactoryBean<Person> {
12
13     @Override
14     public Person getObject() throws Exception {
15         return new Person();
16     }
17
18     @Override
19     public Class<?> getObjectType() {
20         return null;
21     }
22
23     @Override
24     public boolean isSingleton() {
25         // true表示单例
26         // false表示原型
27         return true;
28     }
29 }
30
```

第三步：在Spring配置文件中配置FactoryBean

spring.xml

```
1 <bean id="personBean" class="com.powernode.spring6.bean.PersonFactoryBean"/>
```

测试程序：

```

1  @Test
2  public void testFactoryBean(){
3      ApplicationContext applicationContext = new ClassPathXmlApplicationCont
ext("spring.xml");
4      Person personBean = applicationContext.getBean("personBean", Person.cla
ss);
5      System.out.println(personBean);
6
7      Person personBean2 = applicationContext.getBean("personBean", Person.cl
ass);
8      System.out.println(personBean2);
9  }

```

执行结果：

```

Tests passed: 1 of 1 test - 266 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
com.powernode.spring6.bean.Person@79e2c065
com.powernode.spring6.bean.Person@79e2c065
    动力节点

```

FactoryBean在Spring中是一个接口。被称为“工厂Bean”。“工厂Bean”是一种特殊的Bean。所有的“工厂Bean”都是用来协助Spring框架来创建其他Bean对象的。

7.5 BeanFactory和FactoryBean的区别

7.5.1 BeanFactory

Spring IoC容器的顶级对象，BeanFactory被翻译为“Bean工厂”，在Spring的IoC容器中，“Bean工厂”负责创建Bean对象。

BeanFactory是工厂。

7.5.2 FactoryBean

FactoryBean：它是一个Bean，是一个能够辅助Spring实例化其它Bean对象的一个Bean。

在Spring中，Bean可以分为两类：

- 第一类：普通Bean

- 第二类：工厂Bean（记住：工厂Bean也是一种Bean，只不过这种Bean比较特殊，它可以辅助Spring实例化其它Bean对象。）

7.6 注入自定义Date

我们前面说过，`java.util.Date`在Spring中被当做简单类型，简单类型在注入的时候可以直接使用`value`属性或`value`标签来完成。但我们之前已经测试过了，对于`Date`类型来说，采用`value`属性或`value`标签赋值的时候，对日期字符串的格式要求非常严格，必须是这种格式的：`Mon Oct 10 14:30:26 CST 2022`。其他格式是不会被识别的。如以下代码：

```
1 package com.powernode.spring6.bean;
2
3 import java.util.Date;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className Student
9  * @since 1.0
10 */
11 public class Student {
12     private Date birth;
13
14     public void setBirth(Date birth) {
15         this.birth = birth;
16     }
17
18     @Override
19     public String toString() {
20         return "Student{" +
21                 "birth=" + birth +
22                 '}';
23     }
24 }
25
```

Java | 复制代码

spring.xml

XML | 复制代码

```
1 <bean id="studentBean" class="com.powernode.spring6.bean.Student">
2     <property name="birth" value="Mon Oct 10 14:30:26 CST 2002"/>
3 </bean>
```

测试程序

Java | 复制代码

```

1  @Test
2  public void testDate(){
3      ApplicationContext applicationContext = new ClassPathXmlApplicationCont
ext("spring.xml");
4      Student studentBean = applicationContext.getBean("studentBean", Student
.class);
5      System.out.println(studentBean);
6  }

```

执行结果:

Tests passed: 1 of 1 test – 328 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

Student{birth=Fri Oct 11 04:30:26 CST 2002}

动力节点

如果把日期格式修改一下:

spring.xml

XML | 复制代码

```

1 <bean id="studentBean" class="com.powernode.spring6.bean.Student">
2     <property name="birth" value="2002-10-10"/>
3 </bean>

```

执行结果:

Tests failed: 1 of 1 test – 328 ms

eException: Cannot convert value of type 'java.lang.String' to required type 'java.util.Date' for property 'birth':

动力节点

这种情况下，我们就可以使用FactoryBean来完成这个骚操作。

编写DateFactoryBean实现FactoryBean接口：

DateFactoryBean

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 import org.springframework.beans.factory.FactoryBean;
4
5 import java.text.SimpleDateFormat;
6 import java.util.Date;
7
8 /**
9  * @author 动力节点
10 * @version 1.0
11 * @className DateFactoryBean
12 * @since 1.0
13 */
14 public class DateFactoryBean implements FactoryBean<Date> {
15
16     // 定义属性接收日期字符串
17     private String date;
18
19     // 通过构造方法给日期字符串属性赋值
20     public DateFactoryBean(String date) {
21         this.date = date;
22     }
23
24     @Override
25     public Date getObject() throws Exception {
26         SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
27         return sdf.parse(this.date);
28     }
29
30     @Override
31     public Class<?> getObjectType() {
32         return null;
33     }
34 }
35
```

编写spring配置文件：

spring.xml

```
1 <bean id="dateBean" class="com.powernode.spring6.bean.DateFactoryBean">
2   <constructor-arg name="date" value="1999-10-11"/>
3 </bean>
4
5 <bean id="studentBean" class="com.powernode.spring6.bean.Student">
6   <property name="birth" ref="dateBean"/>
7 </bean>
```

执行测试程序：

```
Tests passed: 1 of 1 test – 361 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
Student{birth=Mon Oct 11 00:00:00 CST 1999}
```

动力节点

八、Bean的生命周期

8.1 什么是Bean的生命周期

Spring其实就是一个管理Bean对象的工厂。它负责对象的创建，对象的销毁等。

所谓的生命周期就是：对象从创建开始到最终销毁的整个过程。

什么时候创建Bean对象？

创建Bean对象的前后会调用什么方法？

Bean对象什么时候销毁？

Bean对象的销毁前后调用什么方法？

8.2 为什么要知道Bean的生命周期

其实生命周期的本质是： 在哪个时间节点上调用了哪个类的哪个方法。

我们需要充分的了解在这个生命线上，都有哪些特殊的时间节点。

只有我们知道了特殊的时间节点都在哪，到时我们才可以确定代码写到哪。

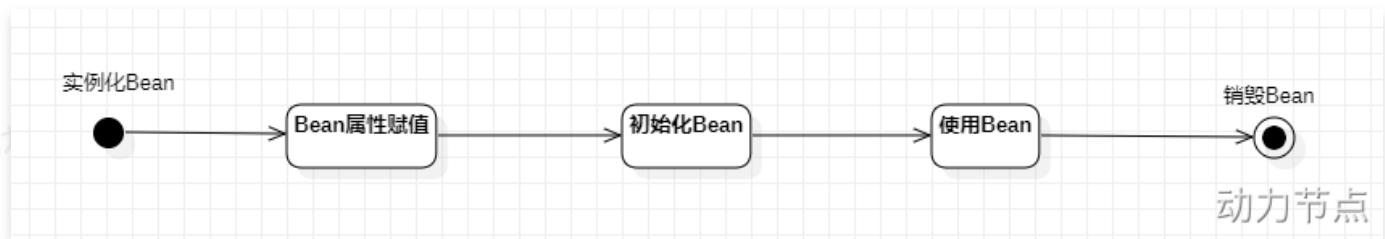
我们可能需要在某个特殊的时间点上执行一段特定的代码，这段代码就可以放到这个节点上。当生命线走到这里的时候，自然会被调用。

8.3 Bean的生命周期之5步

Bean生命周期的管理，可以参考Spring的源码：[AbstractAutowireCapableBeanFactory类的doCreateBean\(\)方法](#)。

Bean生命周期可以粗略的划分为五大步：

- 第一步：实例化Bean
- 第二步：Bean属性赋值
- 第三步：初始化Bean
- 第四步：使用Bean
- 第五步：销毁Bean



编写测试程序：

定义一个Bean

User

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className User
7  * @since 1.0
8 */
9 public class User {
10     private String name;
11
12     public User() {
13         System.out.println("1.实例化Bean");
14     }
15
16     public void setName(String name) {
17         this.name = name;
18         System.out.println("2.Bean属性赋值");
19     }
20
21     public void initBean(){
22         System.out.println("3.初始化Bean");
23     }
24
25     public void destroyBean(){
26         System.out.println("5.销毁Bean");
27     }
28
29 }
30
```

```
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
6     <!--
7         init-method属性指定初始化方法。
8         destroy-method属性指定销毁方法。
9     -->
10    <bean id="userBean" class="com.powernode.spring6.bean.User" init-method="initBean" destroy-method="destroyBean">
11        <property name="name" value="zhangsan"/>
12    </bean>
13
14 </beans>
```

测试程序

Java | 复制代码

```
1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.bean.User;
4 import org.junit.Test;
5 import org.springframework.context.ApplicationContext;
6 import org.springframework.context.support.ClassPathXmlApplicationContext;
7
8 /**
9  * @author 动力节点
10 * @version 1.0
11 * @className BeanLifecycleTest
12 * @since 1.0
13 */
14 public class BeanLifecycleTest {
15     @Test
16     public void testLifecycle(){
17         ApplicationContext applicationContext = new ClassPathXmlApplication
18         context("spring.xml");
19         User userBean = applicationContext.getBean("userBean", User.class)
20 ;
21         System.out.println("4. 使用Bean");
22         // 只有正常关闭spring容器才会执行销毁方法
23         ClassPathXmlApplicationContext context = (ClassPathXmlApplicationC
24 ontext) applicationContext;
25         context.close();
26     }
27 }
```

执行结果:

```
✓ Tests passed: 1 of 1 test – 313 ms
C:\dev\Java\jdk-17.0.4\bin\jav
1. 实例化Bean
2. Bean属性赋值
3. 初始化Bean
4. 使用Bean
5. 销毁Bean
```

需要注意的:

- 第一：只有正常关闭spring容器，bean的销毁方法才会被调用。
- 第二：ClassPathXmlApplicationContext类才有close()方法。
- 第三：配置文件中的init-method指定初始化方法。destroy-method指定销毁方法。

8.4 Bean生命周期之7步

在以上的5步中，第3步是初始化Bean，如果你还想在初始化前和初始化后添加代码，可以加入“Bean后处理器”。

编写一个类实现BeanPostProcessor类，并且重写before和after方法：

```
▼ LogBeanPostProcessor Java | 复制代码
1 package com.powernode.spring6.bean;
2
3 import org.springframework.beans.BeansException;
4 import org.springframework.beans.factory.config.BeanPostProcessor;
5
6 /**
7  * @author 动力节点
8  * @version 1.0
9  * @className LogBeanPostProcessor
10 * @since 1.0
11 */
12 public class LogBeanPostProcessor implements BeanPostProcessor {
13     @Override
14     public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
15         System.out.println("Bean后处理器的before方法执行，即将开始初始化");
16         return bean;
17     }
18
19     @Override
20     public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
21         System.out.println("Bean后处理器的after方法执行，已完成初始化");
22         return bean;
23     }
24 }
25
```

在spring.xml文件中配置“Bean后处理器”：

```
1 <!--配置Bean后处理器。这个后处理器将作用于当前配置文件中所有的bean。-->
2 <bean class="com.powernode.spring6.bean.LogBeanPostProcessor"/>
```

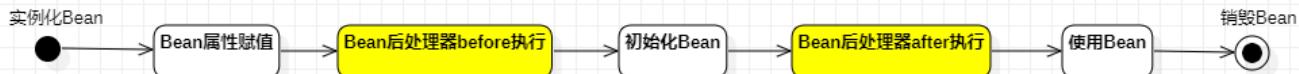
一定要注意：在spring.xml文件中配置的Bean后处理器将作用于当前配置文件中所有的Bean。

执行测试程序：

```
Tests passed: 1 of 1 test – 329 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
1. 实例化Bean
2. Bean属性赋值
Bean后处理器的before方法执行，即将开始初始化
3. 初始化Bean
Bean后处理器的after方法执行，已完成初始化
4. 使用Bean
5. 销毁Bean
```

动力节点

如果加上Bean后处理器的话，Bean的生命周期就是7步了：



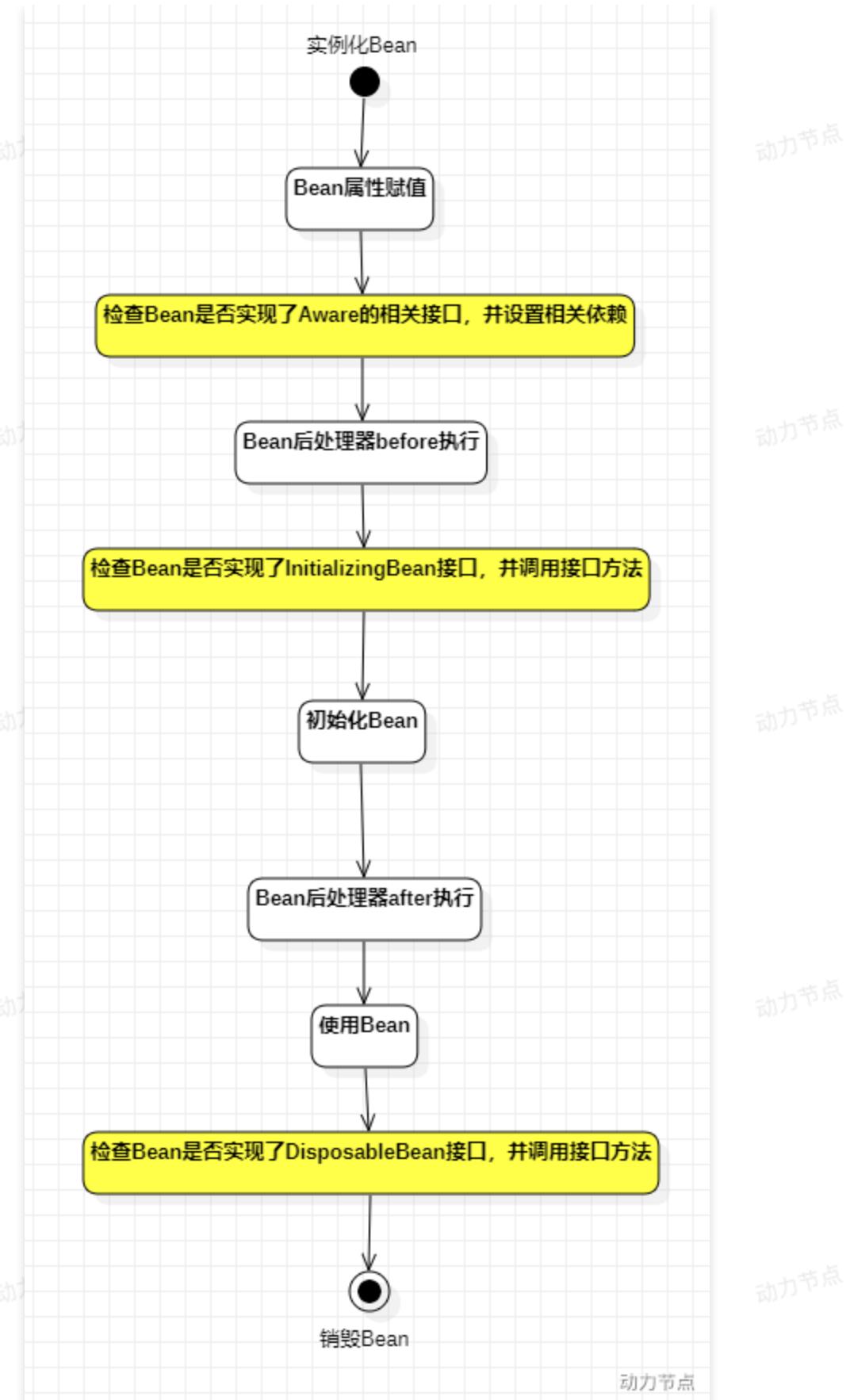
动力节点

8.5 Bean生命周期之10步

如果根据源码跟踪，可以划分更细粒度的步骤，10步：

动力节点

动力节点



上图中检查Bean是否实现了Aware的相关接口是什么意思？

Aware相关的接口包括：BeanNameAware、BeanClassLoaderAware、BeanFactoryAware

- 当Bean实现了BeanNameAware， Spring会将Bean的名字传递给Bean。
- 当Bean实现了BeanClassLoaderAware， Spring会将加载该Bean的类加载器传递给Bean。
- 当Bean实现了BeanFactoryAware， Spring会将Bean工厂对象传递给Bean。

测试以上10步，可以让User类实现5个接口，并实现所有方法：

- BeanNameAware
- BeanClassLoaderAware
- BeanFactoryAware
- InitializingBean
- DisposableBean

代码如下：

```
1 package com.powernode.spring6.bean;
2
3 import org.springframework.beans.BeansException;
4 import org.springframework.beans.factory.*;
5
6 /**
7  * @author 动力节点
8  * @version 1.0
9  * @className User
10 * @since 1.0
11 */
12 public class User implements BeanNameAware, BeanClassLoaderAware, BeanFactoryAware, InitializingBean, DisposableBean {
13     private String name;
14
15     public User() {
16         System.out.println("1.实例化Bean");
17     }
18
19     public void setName(String name) {
20         this.name = name;
21         System.out.println("2.Bean属性赋值");
22     }
23
24     public void initBean(){
25         System.out.println("6.初始化Bean");
26     }
27
28     public void destroyBean(){
29         System.out.println("10.销毁Bean");
30     }
31
32     @Override
33     public void setBeanClassLoader(ClassLoader classLoader) {
34         System.out.println("3.类加载器: " + classLoader);
35     }
36
37     @Override
38     public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
39         System.out.println("3.Bean工厂: " + beanFactory);
40     }
41
42     @Override
43     public void setBeanName(String name) {
```

```
44         System.out.println("3.bean名字: " + name);
45     }
46
47     @Override
48     public void destroy() throws Exception {
49         System.out.println("9.DisposableBean destroy");
50     }
51
52     @Override
53     public void afterPropertiesSet() throws Exception {
54         System.out.println("5.afterPropertiesSet执行");
55     }
56 }
57 }
```

LogBeanPostProcessor

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 import org.springframework.beans.BeansException;
4 import org.springframework.beans.factory.config.BeanPostProcessor;
5
6 /**
7  * @author 动力节点
8  * @version 1.0
9  * @className LogBeanPostProcessor
10 * @since 1.0
11 */
12 public class LogBeanPostProcessor implements BeanPostProcessor {
13     @Override
14     public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
15         System.out.println("4.Bean后处理器的before方法执行，即将开始初始化");
16         return bean;
17     }
18
19     @Override
20     public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
21         System.out.println("7.Bean后处理器的after方法执行，已完成初始化");
22         return bean;
23     }
24 }
25 }
```

执行结果：

Tests passed: 1 of 1 test – 328 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

1. 实例化Bean
2. Bean属性赋值
3. bean名字: userBean
4. 类加载器: jdk.internal.loader.ClassLoaders\$AppClassLoader@63947c6b
5. Bean工厂: org.springframework.beans.factory.support.DefaultListableBeanFactory@147ed70f:
6. Bean后处理器的before方法执行, 即将开始初始化
7. afterPropertiesSet执行
8. 初始化Bean
9. Bean后处理器的after方法执行, 已完成初始化
10. 使用Bean
11. DisposableBean destroy
12. 销毁Bean

动力节点

通过测试可以看出来:

- InitializingBean的方法早于init-method的执行。
- DisposableBean的方法早于destroy-method的执行。

对于SpringBean的生命周期, 掌握之前的7步即可。够用。

8.6 Bean的作用域不同, 管理方式不同

Spring 根据Bean的作用域来选择管理方式。

- 对于singleton作用域的Bean, Spring 能够精确地知道该Bean何时被创建, 何时初始化完成, 以及何时被销毁;
- 而对于 prototype 作用域的 Bean, Spring 只负责创建, 当容器创建了 Bean 的实例后, Bean 的实例就交给客户端代码管理, Spring 容器将不再跟踪其生命周期。

我们把之前User类的spring.xml文件中的配置scope设置为prototype:

spring.xml

XML | 复制代码

```
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
6     <!--
7         init-method属性指定初始化方法。
8         destroy-method属性指定销毁方法。
9     -->
10    <bean id="userBean" class="com.powernode.spring6.bean.User" init-method="initBean" destroy-method="destroyBean" scope="prototype">
11        <property name="name" value="zhangsan"/>
12    </bean>
13
14    <!--配置Bean后处理器。这个后处理器将作用于当前配置文件中所有的bean。-->
15    <bean class="com.powernode.spring6.bean.LogBeanPostProcessor"/>
16
17 </beans>
```

执行测试程序：

```
Tests passed: 1 of 1 test - 328 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
1.实例化Bean
2.Bean属性赋值
3.bean名字: userBean
4.类加载器: jdk.internal.loader.ClassLoaders$AppClassLoader@63947c6b
5.Bean工厂: org.springframework.beans.factory.support.DefaultListableBeanFactory@147ed70f:
6.Bean后处理器的before方法执行，即将开始初始化
7.afterPropertiesSet执行
8.初始化Bean
9.Bean后处理器的after方法执行，已完成初始化
10.使用Bean

Process finished with exit code 0
```

动力节点

通过测试一目了然。只执行了前8步，第9和10都没有执行。

8.7 自己new的对象如何让Spring管理

有些时候可能会遇到这样的需求，某个java对象是我们自己new的，然后我们希望这个对象被Spring容器管理，怎么实现？

```
1 package com.powernode.spring6.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className User
7  * @since 1.0
8 */
9 public class User {
10 }
11
```

```

1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.bean.User;
4 import org.junit.Test;
5 import org.springframework.beans.factory.support.DefaultListableBeanFactory;
6
7 /**
8  * @author 动力节点
9  * @version 1.0
10 * @className RegisterBeanTest
11 * @since 1.0
12 */
13 public class RegisterBeanTest {
14
15     @Test
16     public void testBeanRegister(){
17         // 自己new的对象
18         User user = new User();
19         System.out.println(user);
20
21         // 创建 默认可列表BeanFactory 对象
22         DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
23
24         // 注册Bean
25         factory.registerSingleton("userBean", user);
26         // 从spring容器中获取bean
27         User userBean = factory.getBean("userBean", User.class);
28         System.out.println(userBean);
29     }
30 }
```

执行结果：

Tests passed: 1 of 1 test – 66 ms

```
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
com.powernode.spring6.bean.User@3c5a99da
com.powernode.spring6.bean.User@3c5a99da
```

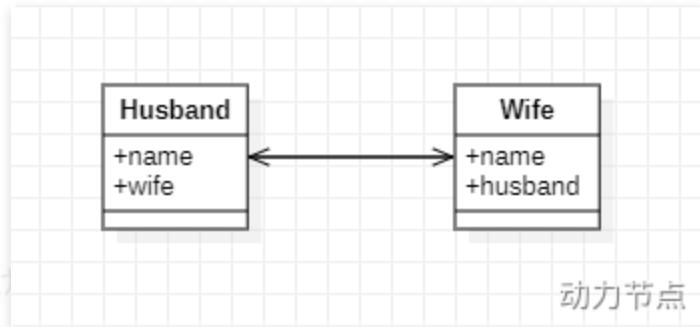
动力节点

九、Bean的循环依赖问题

9.1 什么是Bean的循环依赖

A对象中有B属性。B对象中有A属性。这就是循环依赖。我依赖你，你也依赖我。

比如：丈夫类Husband，妻子类Wife。Husband中有Wife的引用。Wife中有Husband的引用。



```
1 package com.powernode.spring6.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Husband
7  * @since 1.0
8 */
9 public class Husband {
10     private String name;
11     private Wife wife;
12 }
13
```

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Wife
7  * @since 1.0
8 */
9 public class Wife {
10     private String name;
11     private Husband husband;
12 }
13
```

9.2 singleton下的set注入产生的循环依赖

我们来编写程序，测试一下在singleton+setter的模式下产生的循环依赖，Spring是否能够解决？

Husband

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Husband
7  * @since 1.0
8 */
9 public class Husband {
10     private String name;
11     private Wife wife;
12
13     public void setName(String name) {
14         this.name = name;
15     }
16
17     public String getName() {
18         return name;
19     }
20
21     public void setWife(Wife wife) {
22         this.wife = wife;
23     }
24
25     // toString()方法重写时需要注意：不能直接输出wife，输出wife.getName()。要不然
26     // 会出现递归导致的栈内存溢出错误。
27     @Override
28     public String toString() {
29         return "Husband{" +
30                 "name='" + name + '\'' +
31                 ", wife=" + wife.getName() +
32                 '}';
33     }
34 }
```

Wife

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Wife
7  * @since 1.0
8 */
9 public class Wife {
10     private String name;
11     private Husband husband;
12
13     public void setName(String name) {
14         this.name = name;
15     }
16
17     public String getName() {
18         return name;
19     }
20
21     public void setHusband(Husband husband) {
22         this.husband = husband;
23     }
24
25     // toString()方法重写时需要注意：不能直接输出husband，输出husband.getName()。
26     //要不然会出现递归导致的栈内存溢出错误。
27     @Override
28     public String toString() {
29         return "Wife{" +
30                 "name='" + name + '\'' +
31                 ", husband=" + husband.getName() +
32                 '}';
33     }
34 }
```

```
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
6   <bean id="husbandBean" class="com.powernode.spring6.bean.Husband" scope="singleton">
7     <property name="name" value="张三"/>
8     <property name="wife" ref="wifeBean"/>
9   </bean>
10  <bean id="wifeBean" class="com.powernode.spring6.bean.Wife" scope="singleton">
11    <property name="name" value="小花"/>
12    <property name="husband" ref="husbandBean"/>
13  </bean>
14 </beans>
```

```

1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.bean.Husband;
4 import com.powernode.spring6.bean.Wife;
5 import org.junit.Test;
6 import org.springframework.context.ApplicationContext;
7 import org.springframework.context.support.ClassPathXmlApplicationContext;
8
9 /**
10 * @author 动力节点
11 * @version 1.0
12 * @className CircularDependencyTest
13 * @since 1.0
14 */
15 public class CircularDependencyTest {
16
17     @Test
18     public void testSingletonAndSet(){
19         ApplicationContext applicationContext = new ClassPathXmlApplication
19         context("spring.xml");
20         Husband husbandBean = applicationContext.getBean("husbandBean", Hu
20         sband.class);
21         Wife wifeBean = applicationContext.getBean("wifeBean", Wife.class)
22         ;
23         System.out.println(husbandBean);
24         System.out.println(wifeBean);
25     }
26 }
```

执行结果：

```

Tests passed: 1 of 1 test - 330 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
Husband{name='张三', wife=小花}
Wife{name='小花', husband=张三}

```

动力节点

通过测试得知：在singleton + set注入的情况下，循环依赖是没有问题的。Spring可以解决这个问题。

9.3 prototype下的set注入产生的循环依赖

动力节点

我们再来测试一下：prototype+set注入的方式下，循环依赖会不会出现问题？

```
spring.xml XML | 复制代码

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
6      <bean id="husbandBean" class="com.powernode.spring6.bean.Husband" scope="prototype">
7          <property name="name" value="张三"/>
8          <property name="wife" ref="wifeBean"/>
9      </bean>
10     <bean id="wifeBean" class="com.powernode.spring6.bean.Wife" scope="prototype">
11         <property name="name" value="小花"/>
12         <property name="husband" ref="husbandBean"/>
13     </bean>
14 </beans>
```

执行测试程序：发生了异常，异常信息如下：

Caused by: org.springframework.beans.factory.BeanCurrentlyInCreationException: Error creating bean with name 'husbandBean': Requested bean is currently in creation: Is there an unresolvable circular reference?

at

org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:265)

at

org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:199)

at

org.springframework.beans.factory.support.BeanDefinitionValueResolver.resolveReference(BeanDefinitionValueResolver.java:325)

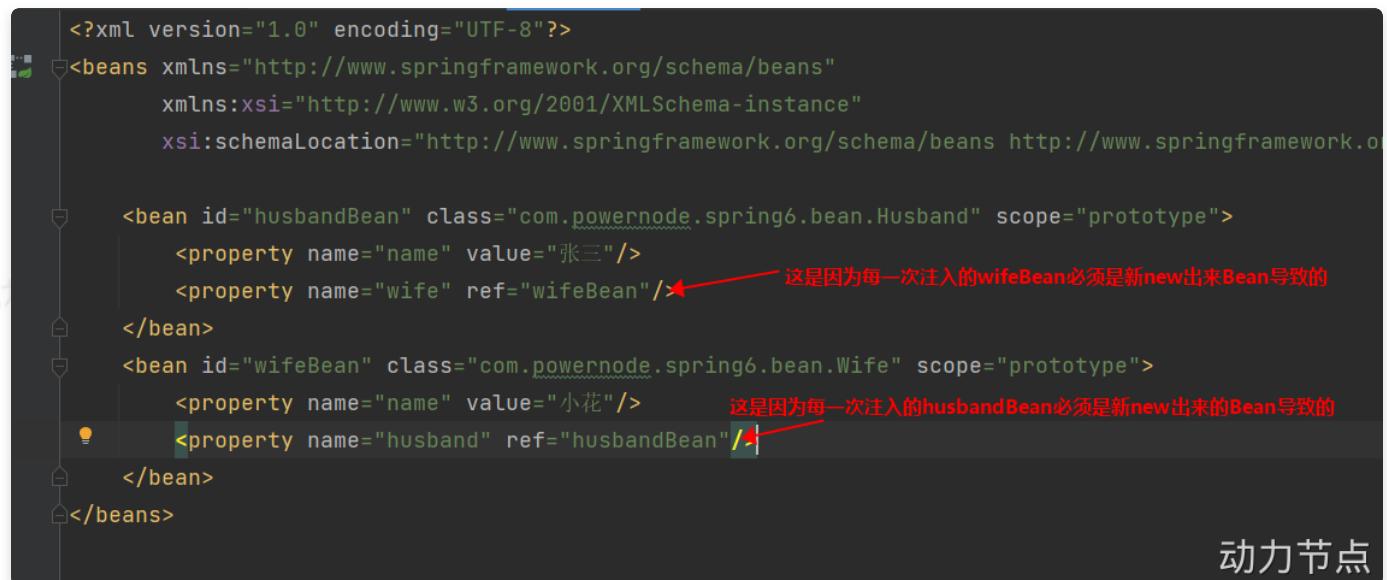
... 44 more

翻译为：创建名为“husbandBean”的bean时出错：请求的bean当前正在创建中：是否存在无法解析的循环引用？

通过测试得知，当循环依赖的**所有Bean**的scope="prototype"的时候，产生的循环依赖，Spring是无法解决的，会出现**BeanCurrentlyInCreationException**异常。

大家可以测试一下，以上两个Bean，如果其中一个是singleton，另一个是prototype，是没有问题的。

为什么两个Bean都是prototype时会出错呢？



```
<?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.xsd">

    <bean id="husbandBean" class="com.powernode.spring6.bean.Husband" scope="prototype">
        <property name="name" value="张三"/>
        <property name="wife" ref="wifeBean"/>-----这是因为每一次注入的wifeBean必须是新new出来Bean导致的-----
    </bean>
    <bean id="wifeBean" class="com.powernode.spring6.bean.Wife" scope="prototype">
        <property name="name" value="小花"/>-----这是因为每一次注入的husbandBean必须是新new出来的Bean导致的-----
        <property name="husband" ref="husbandBean"/>-----
    </bean>
</beans>
```

动力节点

9.4 singleton下的构造注入产生的循环依赖

我们再来测试一下singleton + 构造注入的方式下，spring是否能够解决这种循环依赖。

Husband

Java | 复制代码

```
1 package com.powernode.spring6.bean2;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Husband
7  * @since 1.0
8 */
9 public class Husband {
10     private String name;
11     private Wife wife;
12
13     public Husband(String name, Wife wife) {
14         this.name = name;
15         this.wife = wife;
16     }
17
18     // -----分割线-----
19     public String getName() {
20         return name;
21     }
22
23     @Override
24     public String toString() {
25         return "Husband{" +
26                 "name='" + name + '\'' +
27                 ", wife='" + wife +
28                 '}';
29     }
30 }
31
```

```
1 package com.powernode.spring6.bean2;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Wife
7  * @since 1.0
8 */
9 public class Wife {
10     private String name;
11     private Husband husband;
12
13     public Wife(String name, Husband husband) {
14         this.name = name;
15         this.husband = husband;
16     }
17
18     // -----分割线-----
19     public String getName() {
20         return name;
21     }
22
23     @Override
24     public String toString() {
25         return "Wife{" +
26                 "name='" + name + '\'' +
27                 ", husband=" + husband +
28                 '}';
29     }
30 }
31
```

spring2.xml

XML | 复制代码

```

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
6     <bean id="hBean" class="com.powernode.spring6.bean2.Husband" scope="singleton">
7         <constructor-arg name="name" value="张三"/>
8         <constructor-arg name="wife" ref="wBean"/>
9     </bean>
10
11    <bean id="wBean" class="com.powernode.spring6.bean2.Wife" scope="singleton">
12        <constructor-arg name="name" value="小花"/>
13        <constructor-arg name="husband" ref="hBean"/>
14    </bean>
15 </beans>

```

测试程序

Java | 复制代码

```

1 @Test
2 public void testSingletonAndConstructor(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring2.xml");
4     Husband hBean = applicationContext.getBean("hBean", Husband.class);
5     Wife wBean = applicationContext.getBean("wBean", Wife.class);
6     System.out.println(hBean);
7     System.out.println(wBean);
8 }

```

执行结果：发生了异常，信息如下：

Caused by: org.springframework.beans.factory.BeanCurrentlyInCreationException: Error creating bean with name 'hBean': Requested bean is currently in creation: Is there an unresolvable circular reference?

at

org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.beforeSingletonCreation
(DefaultSingletonBeanRegistry.java:355)

at

org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(DefaultSin

```
gletonBeanRegistry.java:227)
    at
org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:324)
    at
org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:199)
    at
org.springframework.beans.factory.support.BeanDefinitionValueResolver.resolveReference(BeanDefinitionValueResolver.java:325)
...
... 56 more
```

和上一个测试结果相同，都是提示产生了循环依赖，并且Spring是无法解决这种循环依赖的。

为什么呢？

主要原因是因为通过构造方法注入导致的：因为构造方法注入会导致实例化对象的过程和对象属性赋值的过程没有分离开，必须在一起完成导致的。

9.5 Spring解决循环依赖的机理

Spring为什么可以解决set + singleton模式下循环依赖？

根本的原因在于：这种方式可以做到将“实例化Bean”和“给Bean属性赋值”这两个动作分开去完成。

实例化Bean的时候：调用无参数构造方法来完成。**此时可以先不给属性赋值，可以提前将该Bean对象“曝光”给外界。**

给Bean属性赋值的时候：调用setter方法来完成。

两个步骤是完全可以分离开去完成的，并且这两步不要求在同一个时间点上完成。

也就是说，Bean都是单例的，我们可以先把所有的单例Bean实例化出来，放到一个集合当中（我们可以称之为缓存），所有的单例Bean全部实例化完成之后，以后我们再慢慢的调用setter方法给属性赋值。这样就解决了循环依赖的问题。

那么在Spring框架底层源码级别上是如何实现的呢？请看：

5 usages 4 inheritors

```

public class DefaultSingletonBeanRegistry extends SimpleAliasRegistry implements SingletonBeanRegistry {

    Maximum number of suppressed exceptions to preserve.
    1 usage
    private static final int SUPPRESSED_EXCEPTIONS_LIMIT = 100;

    /**
     * Cache of singleton objects: bean name to bean instance.
     */
    22 usages
    private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>( initialCapacity: 256);

    /**
     * Cache of singleton factories: bean name to ObjectFactory.
     */
    6 usages
    private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>( initialCapacity: 16);

    /**
     * Cache of early singleton objects: bean name to bean instance.
     */
    7 usages
    private final Map<String, Object> earlySingletonObjects = new ConcurrentHashMap<>( initialCapacity: 16);

    Set of registered singletons, containing the bean names in registration order.

```

动力节点

在以上类中包含三个重要的属性：

Cache of singleton objects: bean name to bean instance. 单例对象的缓存：key存储bean名称，value存储Bean对象 【一级缓存】

Cache of early singleton objects: bean name to bean instance. 早期单例对象的缓存：key存储bean名称，value存储早期的Bean对象 【二级缓存】

Cache of singleton factories: bean name to ObjectFactory. 单例工厂缓存：key存储bean名称，value存储该Bean对应的ObjectFactory对象 【三级缓存】

这三个缓存其实本质上是三个Map集合。

我们再来看，在该类中有这样一个方法addSingletonFactory()，这个方法的作用是：将创建Bean对象的ObjectFactory对象提前曝光。

Add the given singleton factory for building the specified singleton if necessary.
To be called for eager registration of singletons, e.g. to be able to resolve circular references.
Params: beanName – the name of the bean
singletonFactory – the factory for the singleton object

1 usage

```

protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {
    Assert.notNull(singletonFactory, message: "Singleton factory must not be null");
    synchronized (this.singletonObjects) {
        if (!this.singletonObjects.containsKey(beanName)) {
            this.singletonFactories.put(beanName, singletonFactory); 曝光该Bean对应的ObjectFactory对象
            this.earlySingletonObjects.remove(beanName);
            this.registeredSingletons.add(beanName);
        }
    }
}

```

动力节点

再分析下面的源码：

```
va x C Wife.java x spring2.xml x CircularDependencyTest.java x DefaultSingletonBeanRegistry.java x
params. beanName - the name of the bean to look for
allowEarlyReference - whether early references should be created or not
Returns: the registered singleton object, or null if none found

8 usages
@Nullable
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    // Quick check for existing instance without full singleton lock
    Object singletonObject = this.singletonObjects.get(beanName); 先从一级缓存中获取
    if (singletonObject == null & isSingletonCurrentlyInCreation(beanName)) {
        singletonObject = this.earlySingletonObjects.get(beanName); 一级缓存获取不到时，再从二级缓存中获取
        if (singletonObject == null & allowEarlyReference) {
            synchronized (this.singletonObjects) {
                // Consistent creation of early reference within full singleton lock
                singletonObject = this.singletonObjects.get(beanName);
                if (singletonObject == null) {
                    singletonObject = this.earlySingletonObjects.get(beanName);
                    if (singletonObject == null) { 二级缓存拿不到的话，从三级缓存中获取
                        ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
                        if (singletonFactory != null) { 三级缓存中存放了提前曝光的ObjectFactory对象。
                            singletonObject = singletonFactory.getObject();
                            this.earlySingletonObjects.put(beanName, singletonObject);
                            this.singletonFactories.remove(beanName);
                        }
                    }
                }
            }
        }
    }
    return singletonObject;
}
```

动力节点

从源码中可以看到，spring会先从一级缓存中获取Bean，如果获取不到，则从二级缓存中获取Bean，如果二级缓存还是获取不到，则从三级缓存中获取之前曝光的ObjectFactory对象，通过ObjectFactory对象获取Bean实例，这样就解决了循环依赖的问题。

总结：

Spring只能解决setter方法注入的单例bean之间的循环依赖。ClassA依赖ClassB，ClassB又依赖ClassA，形成依赖闭环。Spring在创建ClassA对象后，不需要等给属性赋值，直接将其曝光到bean缓存当中。在解析ClassA的属性时，又发现依赖于ClassB，再次去获取ClassB，当解析ClassB的属性时，又发现需要ClassA的属性，但此时的ClassA已经被提前曝光加入了正在创建的bean的缓存中，则无需创建新的的ClassA的实例，直接从缓存中获取即可。从而解决循环依赖问题。

十、回顾反射机制

10.1 分析方法四要素

我们先来看一下，不使用反射机制调用一个方法需要几个要素的参与。

有一个这样的类：

```
1 package com.powernode.reflect;
2
3 /**
4 * @author 动力节点
5 * @version 1.0
6 * @className SystemService
7 * @since 1.0
8 */
9 public class SystemService {
10
11     public void logout(){
12         System.out.println("退出系统");
13     }
14
15     public boolean login(String username, String password){
16         if ("admin".equals(username) && "admin123".equals(password)) {
17             return true;
18         }
19         return false;
20     }
21 }
22
```

编写程序调用方法：

```
ReflecTest01 Java | 复制代码

1 package com.powernode.reflect;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className ReflectTest01
7  * @since 1.0
8 */
9 public class ReflectTest01 {
10    public static void main(String[] args) {
11
12        // 创建对象
13        SystemService systemService = new SystemService();
14
15        // 调用方法并接收方法的返回值
16        boolean success = systemService.login("admin", "admin123");
17
18        System.out.println(success ? "登录成功" : "登录失败");
19    }
20}
```

执行结果：



通过以上第16行代码可以看出，调用一个方法，一般涉及到4个要素：

- 调用哪个对象的 (systemService)
- 哪个方法 (login)
- 传什么参数 ("admin", "admin123")
- 返回什么值 (success)

10.2 获取Method

要使用反射机制调用一个方法，首先你要获取到这个方法。

在反射机制中Method实例代表的是一个方法。那么怎么获取Method实例呢？

有这样一个类：

▼ SystemService

Java | 复制代码

```
1 package com.powernode.reflect;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className SystemService
7  * @since 1.0
8 */
9 public class SystemService {
10
11     public void logout(){
12         System.out.println("退出系统");
13     }
14
15     public boolean login(String username, String password){
16         if ("admin".equals(username) && "admin123".equals(password)) {
17             return true;
18         }
19         return false;
20     }
21
22     public boolean login(String password){
23         if("110".equals(password)){
24             return true;
25         }
26         return false;
27     }
28 }
29
```

我们如何获取到 logout()、login(String, String)、login(String) 这三个方法呢？

要获取方法Method，首先你需要获取这个类Class。

▼ 获取类Class的代码

Java | 复制代码

```
1 Class clazz = Class.forName("com.powernode.reflect.SystemService");
```

当拿到Class之后，调用getDeclaredMethod()方法可以获取到方法。

假如你要获取这个方法：login(String username, String password)

获取login(String username, String password)

Java | 复制代码

```
1 Method loginMethod = clazz.getDeclaredMethod("login", String.class, String.class);
```

假如你要获取到这个方法：login(String password)

获取login(String password)

Java | 复制代码

```
1 Method loginMethod = clazz.getDeclaredMethod("login", String.class);
```

获取一个方法，需要告诉Java程序，你要获取的方法的名字是什么，这个方法上每个形参的类型是什么。这样Java程序才能给你拿到对应的方法。

这样的设计也非常合理，因为在同一个类当中，方法是支持重载的，也就是说方法名可以一样，但参数列表一定是不一样的，所以获取一个方法需要提供方法名以及每个形参的类型。

假设你有这样一个方法：

```
1 public void setAge(int age){  
2     this.age = age;  
3 }
```

你要获取这个方法的话，代码应该这样写：

获取setAge(int age)

Java | 复制代码

```
1 Method setAgeMethod = clazz.getDeclaredMethod("setAge", int.class);
```

其中setAge是方法名，int.class是形参的类型。

如果要获取上面的logout方法，代码应该这样写：

获取logout()

Java | 复制代码

```
1 Method logoutMethod = clazz.getDeclaredMethod("logout");
```

因为这个方法形式参数的个数是0个。所以只需要提供方法名就行了。你学会了吗？

10.3 调用Method

要让一个方法调用的话，就关联到四要素了：

- 调用哪个对象的
- 哪个方法
- 传什么参数
- 返回什么值

SystemService

Java | 复制代码

```

1 package com.powernode.reflect;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className SystemService
7  * @since 1.0
8 */
9 public class SystemService {
10
11     public void logout(){
12         System.out.println("退出系统");
13     }
14
15     public boolean login(String username, String password){
16         if ("admin".equals(username) && "admin123".equals(password)) {
17             return true;
18         }
19         return false;
20     }
21
22     public boolean login(String password){
23         if("110".equals(password)){
24             return true;
25         }
26         return false;
27     }
28 }
29

```

假如我们要调用的方法是：login(String, String)

第一步：创建对象（四要素之首：调用哪个对象的）

Java | 复制代码

```

1 Class clazz = Class.forName("com.powernode.reflect.SystemService");
2 Object obj = clazz.newInstance();

```

第二步：获取方法login(String, String)（四要素之一：哪个方法）

```
Method loginMethod = clazz.getDeclaredMethod("login", String.class, String.class);
```

第三步：调用方法

```
Method对象的invoke()方法可以调用方法  
Object retValue = loginMethod.invoke(obj, "admin", "admin123");
```

解说四要素：

- 哪个对象： obj
- 哪个方法： loginMethod
- 传什么参数： "admin", "admin123"
- 返回什么值： retValue

```
测试程序  
package com.powernode.reflect;  
import java.lang.reflect.Method;  
  
/**  
 * @author 动力节点  
 * @version 1.0  
 * @className ReflectTest02  
 * @since 1.0  
 */  
public class ReflectTest02 {  
    public static void main(String[] args) throws Exception{  
        Class clazz = Class.forName("com.powernode.reflectSystemService");  
        Object obj = clazz.newInstance();  
        Method loginMethod = clazz.getDeclaredMethod("login", String.class  
        , String.class);  
        Object retValue = loginMethod.invoke(obj, "admin", "admin123");  
        System.out.println(retValue);  
    }  
}
```

执行结果：

The screenshot shows a terminal window titled "ReflectTest02". The output is:
C:\dev\Java\jdk-17.0.4\bin\java.e
true

那如果调用既没有参数，又没有返回值的logout方法，应该怎么做？

```
1 package com.powernode.reflect;
2
3 import java.lang.reflect.Method;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className ReflectTest03
9  * @since 1.0
10 */
11 public class ReflectTest03 {
12     public static void main(String[] args) throws Exception{
13         Class clazz = Class.forName("com.powernode.reflectSystemService");
14         Object obj = clazz.newInstance();
15         Method logoutMethod = clazz.getDeclaredMethod("logout");
16         logoutMethod.invoke(obj);
17     }
18 }
19
```

执行结果：

The screenshot shows a terminal window titled "ReflectTest03". The output is:
C:\dev\Java\jdk-17.0.4\bi
退出系统

10.4 假设你知道属性名

假设有这样一个类：

```
1 package com.powernode.reflect;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className User
7  * @since 1.0
8 */
9 public class User {
10     private String name;
11     private int age;
12
13     public String getName() {
14         return name;
15     }
16
17     public void setName(String name) {
18         this.name = name;
19     }
20
21     public int getAge() {
22         return age;
23     }
24
25     public void setAge(int age) {
26         this.age = age;
27     }
28
29     @Override
30     public String toString() {
31         return "User{" +
32                 "name='" + name + '\'' +
33                 ", age=" + age +
34                 '}';
35     }
36 }
37
```

你知道以下这几条信息：

- 类名是：com.powernode.reflect.User
- 该类中有String类型的name属性和int类型的age属性。

- 另外你也知道该类的设计符合javabean规范。 (也就是说属性私有化，对外提供setter和getter方法)

你如何通过反射机制给User对象的name属性赋值zhangsan，给age属性赋值20岁。

```
1 package com.powernode.reflect;
2
3 import java.lang.reflect.Method;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className UserTest
9  * @since 1.0
10 */
11 public class UserTest {
12     public static void main(String[] args) throws Exception{
13         // 已知类名
14         String className = "com.powernode.reflect.User";
15         // 已知属性名
16         String propertyName = "age";
17
18         // 通过反射机制给User对象的age属性赋值20岁
19         Class<?> clazz = Class.forName(className);
20         Object obj = clazz.newInstance(); // 创建对象
21
22         // 根据属性名获取setter方法名
23         String setMethodName = "set" + propertyName.toUpperCase().charAt(0)
24             + propertyName.substring(1);
25
26         // 获取Method
27         Method setMethod = clazz.getDeclaredMethod(setMethodName, int.class);
28
29         // 调用Method
30         setMethod.invoke(obj, 20);
31
32         System.out.println(obj);
33     }
34 }
```

执行结果：

```
UserTest  
C:\dev\Java\jdk-17.0.4\bin\java.e  
User{name='null', age=20}
```

给User的name属性赋值zhangsan，这个大家可以尝试自己完成！！！

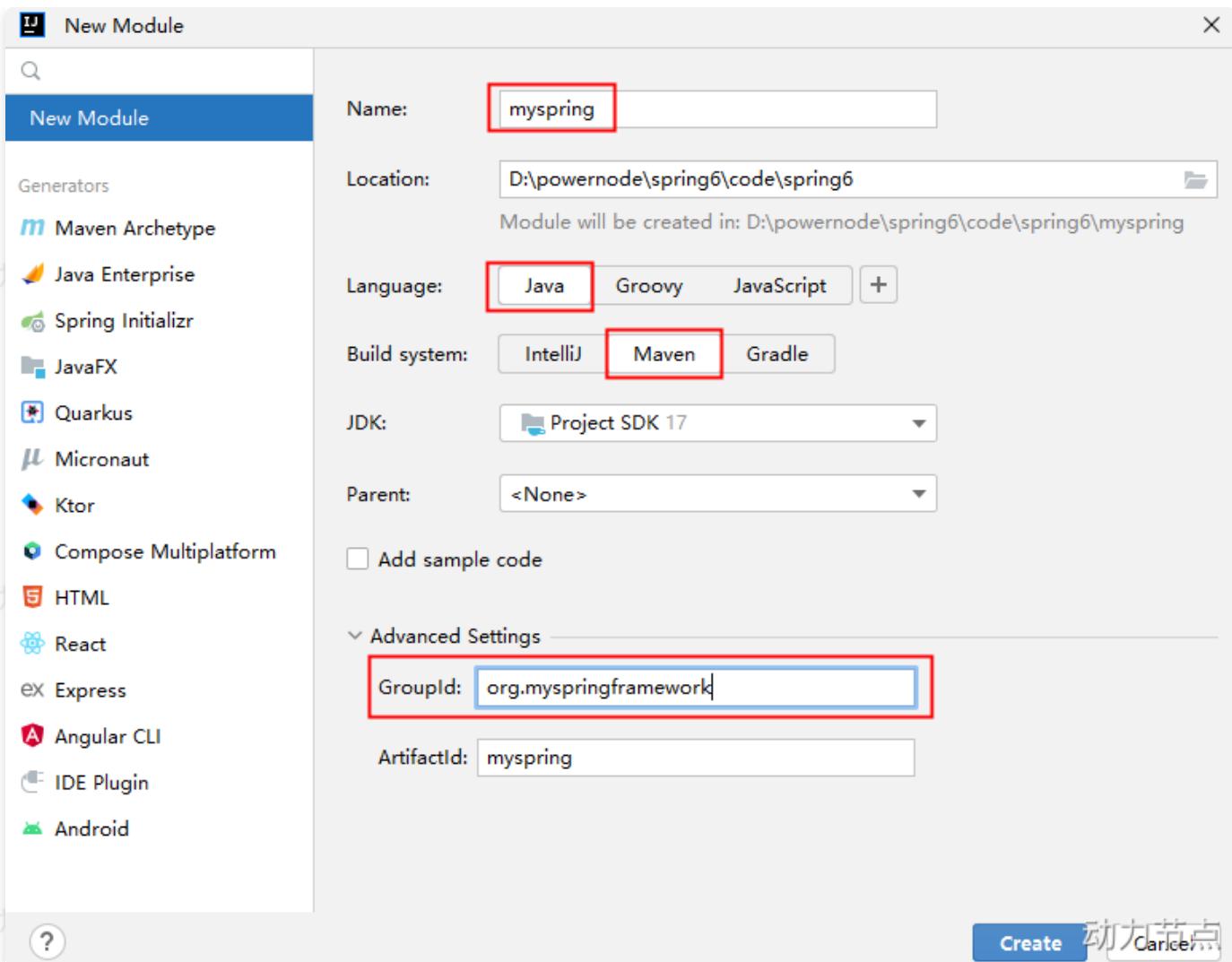
十一、手写Spring框架

Spring IoC容器的实现原理：工厂模式 + 解析XML + 反射机制。

我们给自己的框架起名为：myspring（我的春天）

第一步：创建模块myspring

采用Maven方式新建Module：myspring



打包方式采用jar，并且引入dom4j和jaxen的依赖，因为要使用它解析XML文件，还有junit依赖。

pom.xml

XML | 复制代码

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>org.myspringframework</groupId>
8     <artifactId>myspring</artifactId>
9     <version>1.0.0</version>
10    <packaging>jar</packaging>
11
12    <dependencies>
13        <dependency>
14            <groupId>org.dom4j</groupId>
15            <artifactId>dom4j</artifactId>
16            <version>2.1.3</version>
17        </dependency>
18        <dependency>
19            <groupId>jaxen</groupId>
20            <artifactId>jaxen</artifactId>
21            <version>1.2.0</version>
22        </dependency>
23        <dependency>
24            <groupId>junit</groupId>
25            <artifactId>junit</artifactId>
26            <version>4.13.2</version>
27            <scope>test</scope>
28        </dependency>
29    </dependencies>
30
31    <properties>
32        <maven.compiler.source>17</maven.compiler.source>
33        <maven.compiler.target>17</maven.compiler.target>
34    </properties>
35
36 </project>
```

第二步：准备好我们要管理的Bean

准备好我们要管理的Bean（这些Bean在将来开发完框架之后是要删除的）

注意包名，不要用org.myspringframework包，因为这些Bean不是框架内置的。是将来使用我们框架的程序员提供的。

```
1 package com.powernode.myspring.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Address
7  * @since 1.0
8  **/>
9 public class Address {
10     private String city;
11     private String street;
12     private String zipcode;
13
14     public Address() {
15
16
17     public String getCity() {
18         return city;
19     }
20
21     public void setCity(String city) {
22         this.city = city;
23     }
24
25     public String getStreet() {
26         return street;
27     }
28
29     public void setStreet(String street) {
30         this.street = street;
31     }
32
33     public String getZipcode() {
34         return zipcode;
35     }
36
37     public void setZipcode(String zipcode) {
38         this.zipcode = zipcode;
39     }
40
41     @Override
42     public String toString() {
43         return "Address{" +
44             "city=''" + city + '\'' +
45             ", street=''" + street + '\'' +
```

```
46      ", zipcode=''" + zipcode + '\'' +  
47      '}';  
48  }  
49  
50 }
```

动力节点

动力节点

动力节点

动力节点

动力节点

动力节点

动力节点

动力节点

```
1 package com.powernode.myspring.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className User
7  * @since 1.0
8 */
9 public class User {
10     private String name;
11     private int age;
12     private Address addr;
13
14     public User() {
15     }
16
17     public String getName() {
18         return name;
19     }
20
21     public void setName(String name) {
22         this.name = name;
23     }
24
25     public int getAge() {
26         return age;
27     }
28
29     public void setAge(int age) {
30         this.age = age;
31     }
32
33     public Address getAddress() {
34         return addr;
35     }
36
37     public void setAddress(Address addr) {
38         this.addr = addr;
39     }
40
41     @Override
42     public String toString() {
43         return "User{" +
44             "name='" + name + '\'' +
45             ", age=" + age +
```

```
46             ", addr=" + addr +
47             '}';
48     }
49 }
50 }
```

第三步：准备myspring.xml配置文件

将来在框架开发完毕之后，这个文件也是要删除的。因为这个配置文件的提供者应该是使用这个框架的程序员。

文件名随意，我们这里叫做：myspring.xml

文件放在类路径当中即可，我们这里把文件放到类的根路径下。



The screenshot shows a code editor window with the following details:

- File name: myspring.xml
- Language: XML
- Content:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans>
3
4 <bean id="userBean" class="com.powernode.myspring.bean.User">
5   <property name="name" value="张三"/>
6   <property name="age" value="20"/>
7   <property name="addr" ref="addrBean"/>
8 </bean>
9
10 <bean id="addrBean" class="com.powernode.myspring.bean.Address">
11   <property name="city" value="北京"/>
12   <property name="street" value="大兴区"/>
13   <property name="zipcode" value="1000001"/>
14 </bean>
15
16 </beans>
```

使用value给简单属性赋值。使用ref给非简单属性赋值。

第四步：编写ApplicationContext接口

ApplicationContext接口中提供一个getBean()方法，通过该方法可以获取Bean对象。

注意包名：这个接口就是myspring框架中的一员了。

ApplicationContext接口

Java | 复制代码

```
1 package org.myspringframework.core;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className ApplicationContext
7  * @since 1.0
8  */
9 public interface ApplicationContext {
10 /**
11  * 根据bean的id获取bean实例。
12  * @param beanId bean的id
13  * @return bean实例
14  */
15 Object getBean(String beanId);
16 }
17
```

第五步：编写ClassPathXmlApplicationContext

ClassPathXmlApplicationContext是ApplicationContext接口的实现类。该类从类路径当中加载myspring.xml配置文件。

```
1 package org.myspringframework.core;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className ClassPathXmlApplicationContext
7  * @since 1.0
8  */
9 public class ClassPathXmlApplicationContext implements ApplicationContext{
10     @Override
11     public Object getBean(String beanId) {
12         return null;
13     }
14 }
```

第六步：确定采用Map集合存储Bean

确定采用Map集合存储Bean实例。Map集合的key存储beanId, value存储Bean实例。

Map<String, Object>

在ClassPathXmlApplicationContext类中添加Map<String, Object>属性。

并且在ClassPathXmlApplicationContext类中添加构造方法，该构造方法的参数接收myspring.xml文件。

同时实现getBean方法。

```
▼ ClassPathXmlApplicationContext Java | 复制代码
1 package org.myspringframework.core;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 /**
7  * @author 动力节点
8  * @version 1.0
9  * @className ClassPathXmlApplicationContext
10 * @since 1.0
11 */
12 public class ClassPathXmlApplicationContext implements ApplicationContext{
13     /**
14      * 存储bean的Map集合
15      */
16     private Map<String, Object> beanMap = new HashMap<>();
17
18     /**
19      * 在该构造方法中，解析myspring.xml文件，创建所有的Bean实例，并将Bean实例存放到
20      * Map集合中。
21      * @param resource 配置文件路径（要求在类路径当中）
22      */
23
24     public ClassPathXmlApplicationContext(String resource) {
25
26         @Override
27         public Object getBean(String beanId) {
28             return beanMap.get(beanId);
29         }
30     }
31 }
```

第七步：解析配置文件实例化所有Bean

在ClassPathXmlApplicationContext的构造方法中解析配置文件，获取所有bean的类名，通过反射机制调用无参数构造方法创建Bean。并且将Bean对象存放到Map集合中。

```
▼ ClassPathXmlApplicationContext构造方法 Java | 复制代码

1  /**
2  * 在该构造方法中，解析myspring.xml文件，创建所有的Bean实例，并将Bean实例存放到Map集合中。
3  * @param resource 配置文件路径（要求在类路径当中）
4  */
5  public ClassPathXmlApplicationContext(String resource) {
6      try {
7          SAXReader reader = new SAXReader();
8          Document document = reader.read(ClassLoader.getSystemClassLoader()
9              .getResourceAsStream(resource));
10         // 获取所有的bean标签
11         List<Node> beanNodes = document.selectNodes("//bean");
12         // 遍历集合
13         beanNodes.forEach(beanNode -> {
14             Element beanElt = (Element) beanNode;
15             // 获取id
16             String id = beanElt.attributeValue("id");
17             // 获取className
18             String className = beanElt.attributeValue("class");
19             try {
20                 // 通过反射机制创建对象
21                 Class<?> clazz = Class.forName(className);
22                 Constructor<?> defaultConstructor = clazz.getDeclaredConst
23                 ructor();
24                 Object bean = defaultConstructor.newInstance();
25                 // 存储到Map集合
26                 beanMap.put(id, bean);
27             } catch (Exception e) {
28                 e.printStackTrace();
29             }
30         });
31     } catch (Exception e) {
32         e.printStackTrace();
33     }
34 }
```

第八步：测试能否获取到Bean

编写测试程序。

```
▼ 测试程序 Java | 复制代码

1 package com.powernode.myspring.test;
2
3 import org.junit.Test;
4 import org.myspringframework.core.ApplicationContext;
5 import org.myspringframework.core.ClassPathXmlApplicationContext;
6
7 /**
8 * @author 动力节点
9 * @version 1.0
10 * @className MySpringTest
11 * @since 1.0
12 */
13 public class MySpringTest {
14     @Test
15     public void testMySpring(){
16         ApplicationContext applicationContext = new ClassPathXmlApplication
17         context("myspring.xml");
18         Object userBean = applicationContext.getBean("userBean");
19         Object addrBean = applicationContext.getBean("addrBean");
20         System.out.println(userBean);
21         System.out.println(addrBean);
22     }
23 }
```

执行结果：

```
Tests passed: 1 of 1 test - 149 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe -ea -Didea.test.cyclic
User{name='null', age=0, addr=null}
Address{city='null', street='null', zipcode='null'}
动力节点
```

通过测试Bean已经实例化成功了，属性的值是null，这是我们能够想到的，毕竟我们调用的是无参数构造方法，所以属性都是默认值。

下一步就是我们应该如何给Bean的属性赋值呢？

第九步：给Bean的属性赋值

通过反射机制调用set方法，给Bean的属性赋值。

继续在ClassPathXmlApplicationContext构造方法中编写代码。

```
1 package org.myspringframework.core;
2
3 import org.dom4j.Document;
4 import org.dom4j.Element;
5 import org.dom4j.Node;
6 import org.dom4j.io.SAXReader;
7
8 import java.lang.reflect.Constructor;
9 import java.lang.reflect.Method;
10 import java.util.HashMap;
11 import java.util.List;
12 import java.util.Map;
13
14 /**
15  * @author 动力节点
16  * @version 1.0
17  * @className ClassPathXmlApplicationContext
18  * @since 1.0
19  */
20 public class ClassPathXmlApplicationContext implements ApplicationContext
{
21     /**
22      * 存储bean的Map集合
23      */
24     private Map<String, Object> beanMap = new HashMap<>();
25
26     /**
27      * 在该构造方法中，解析myspring.xml文件，创建所有的Bean实例，并将Bean实例存放
28      * 到Map集合中。
29      * @param resource 配置文件路径（要求在类路径当中）
30      */
31     public ClassPathXmlApplicationContext(String resource) {
32         try {
33             SAXReader reader = new SAXReader();
34             Document document = reader.read(ClassLoader.getSystemClassLoader().getResourceAsStream(resource));
35             // 获取所有的bean标签
36             List<Node> beanNodes = document.selectNodes("//bean");
37             // 遍历集合（这里的遍历只实例化Bean，不给属性赋值。为什么要这样做？）
38             beanNodes.forEach(beanNode -> {
39                 Element beanElt = (Element) beanNode;
40                 // 获取id
41                 String id = beanElt.attributeValue("id");
42                 // 获取className
43                 String className = beanElt.attributeValue("class");
```

```

43             try {
44                 // 通过反射机制创建对象
45                 Class<?> clazz = Class.forName(className);
46                 Constructor<?> defaultConstructor = clazz.getDeclared
47                 Constructor();
48
49                 Object bean = defaultConstructor.newInstance();
50                 // 存储到Map集合
51                 beanMap.put(id, bean);
52             } catch (Exception e) {
53                 e.printStackTrace();
54             }
55         });
56
57         // 再重新遍历集合，这次遍历是为了给Bean的所有属性赋值。
58         // 思考：为什么不在上面的循环中给Bean的属性赋值，而在这里再重新遍历一次
59         // 呢？
60         // 通过这里你是否能够想到Spring是如何解决循环依赖的：实例化和属性赋值分
61         // 开。
62
63         beanNodes.forEach(beanNode -> {
64             Element beanEl = (Element) beanNode;
65             // 获取bean的id
66             String beanId = beanEl.getAttribute("id");
67             // 获取所有property标签
68             List<Element> propertyElts = beanEl.elements("property")
69             ;
70
71             // 遍历所有属性
72             propertyElts.forEach(propertyEl -> {
73                 try {
74                     // 获取属性名
75                     String propertyName = propertyEl.getAttribute(
76                         "name");
77
78                     // 获取属性类型
79                     Class<?> propertyType = beanMap.get(beanId).getCl
80                     ass().getDeclaredField(propertyName).getType();
81
82                     // 获取set方法名
83                     String setMethodName = "set" + propertyName.toUpperCase()
84                     .charAt(0) + propertyName.substring(1);
85
86                     // 获取set方法
87                     Method setMethod = beanMap.get(beanId).getClass()
88                     .getDeclaredMethod(setMethodName, propertyType);
89
90                     // 获取属性的值，值可能是value，也可能是ref。
91                     // 获取value
92                     String propertyValue = propertyEl.getAttributeValue
93                     ("value");
94
95                     // 获取ref
96                     String propertyRef = propertyEl.getAttributeValue(
97                         "ref");
98
99                     Object propertyVal = null;
100                    if (propertyValue != null) {

```

```
81 // 该属性是简单属性
82 String propertyTypeSimpleName = propertyType.
83     getSimpleName();
84
85     switch (propertyTypeSimpleName) {
86         case "byte": case "Byte":
87             propertyVal = Byte.valueOf(propertyVa-
88             lue);
89         break;
90         case "short": case "Short":
91             propertyVal = Short.valueOf(propertyV-
92             alue);
93         break;
94         case "int": case "Integer":
95             propertyVal = Integer.valueOf(propert-
96             yValue);
97         break;
98         case "long": case "Long":
99             propertyVal = Long.valueOf(propertyVa-
100            lue);
101        break;
102        case "float": case "Float":
103            propertyVal = Float.valueOf(propertyV-
104            alue);
105        break;
106        case "double": case "Double":
107            propertyVal = Double.valueOf(property-
108            Value);
109        break;
110        case "boolean": case "Boolean":
111            propertyVal = Boolean.valueOf(propert-
112            yValue);
113        break;
114        case "char": case "Character":
115            propertyVal = propertyValue.charAt(0);
116        break;
117    }
118
119    if (propertyRef != null) {
120        // 该属性不是简单属性
121        setMethod.invoke(beanMap.get(beanId), beanMap.
122            get(propertyRef));
123    }
124 }
```

```

118             } catch (Exception e) {
119                 e.printStackTrace();
120             }
121         });
122     });
123 }
124     } catch (Exception e) {
125         e.printStackTrace();
126     }
127 }
128 }
129 @Override
130 public Object getBean(String beanId) {
131     return beanMap.get(beanId);
132 }
133 }
134

```

重点处理：当property标签中是value怎么办？是ref怎么办？

执行测试程序：

```

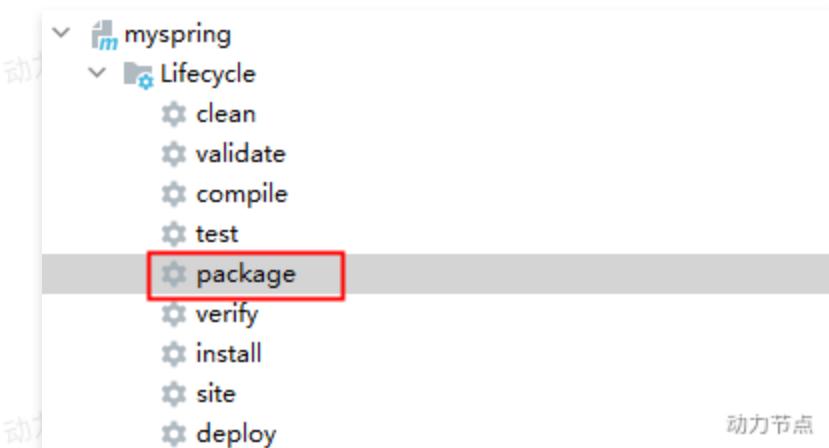
Tests passed: 1 of 1 test – 140 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe -ea -Didea.test.cyclic.buffer.size=1048576 "-javaag
User{name='张三', age=20, addr=Address{city='北京', street='大兴区', zipcode='10000001'}}}
Address{city='北京', street='大兴区', zipcode='10000001'}

```

动力节点

第十步：打包发布

将多余的类以及配置文件删除，使用maven打包发布。



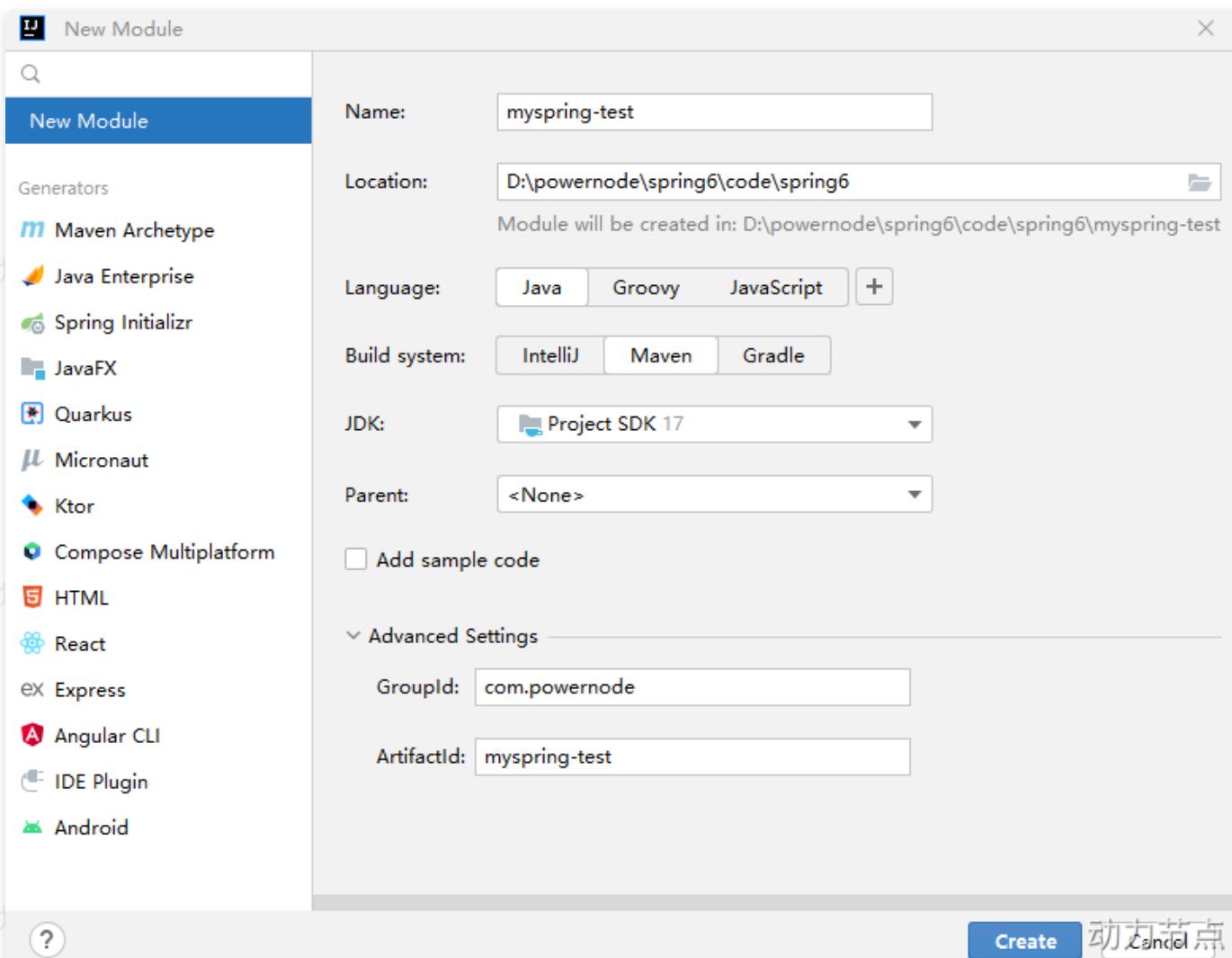
文档 (E:) > repository > org > myspring > myspring > 1.0.0

名称	修改日期	类型
_remote.repositories	2022/9/25 21:42	REPOSITORIES ...
myspring-1.0.0.jar	2022/9/25 21:42	Executable Jar File
myspring-1.0.0.pom	2022/9/25 21:42	POM 文件

动力节点

第十一步：站在程序员角度使用myspring框架

新建模块：myspring-test



引入myspring框架的依赖：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>com.powernode</groupId>
8     <artifactId>myspring-test</artifactId>
9     <version>1.0-SNAPSHOT</version>
10    <packaging>jar</packaging>
11
12    <dependencies>
13        <dependency>
14            <groupId>org.myspringframework</groupId>
15            <artifactId>myspring</artifactId>
16            <version>1.0.0</version>
17        </dependency>
18        <dependency>
19            <groupId>junit</groupId>
20            <artifactId>junit</artifactId>
21            <version>4.13.2</version>
22            <scope>test</scope>
23        </dependency>
24    </dependencies>
25
26    <properties>
27        <maven.compiler.source>17</maven.compiler.source>
28        <maven.compiler.target>17</maven.compiler.target>
29    </properties>
30
31 </project>
```

编写Bean

UserDao

Java | 复制代码

```
1 package com.powernode.myspring.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className UserDao
7  * @since 1.0
8 */
9 public class UserDao {
10    public void insert(){
11        System.out.println("UserDao正在插入数据");
12    }
13}
14
```

UserService

Java | 复制代码

```
1 package com.powernode.myspring.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className UserService
7  * @since 1.0
8 */
9 public class UserService {
10    private UserDao userDao;
11
12    public void setUserDao(UserDao userDao) {
13        this.userDao = userDao;
14    }
15
16    public void save(){
17        System.out.println("UserService开始执行save操作");
18        userDao.insert();
19        System.out.println("UserService执行save操作结束");
20    }
21}
22
```

编写myspring.xml文件

动力节点

myspring.xml

XML | 复制代码

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <beans>
4
5 <bean id="userServiceBean" class="com.powernode.myspring.bean.UserService">
6     <property name="userDao" ref="userDaoBean"/>
7 </bean>
8
9 <bean id="userDaoBean" class="com.powernode.myspring.bean.UserDao"/>
10
11 </beans>
```

编写测试程序

Java | 复制代码

```
1 package com.powernode.myspring.test;
2
3 import com.powernode.myspring.bean.UserService;
4 import org.junit.Test;
5 import org.springframework.core.ApplicationContext;
6 import org.springframework.core.ClassPathXmlApplicationContext;
7
8 /**
9  * @author 动力节点
10 * @version 1.0
11 * @className MySpringTest
12 * @since 1.0
13 */
14 public class MySpringTest {
15
16     @Test
17     public void testMySpring(){
18         ApplicationContext applicationContext = new ClassPathXmlApplication
19             nContext("myspring.xml");
20         UserService userServiceBean = (UserService) applicationContext.get
21             Bean("userServiceBean");
22         userServiceBean.save();
23     }
24 }
```

执行结果

```
Tests passed: 1 of 1 test – 163 ms  
C:\dev\Java\jdk-17.0.4\bin\java.exe -  
UserService开始执行save操作  
UserDao正在插入数据  
UserService执行save操作结束
```

动力节点

十二、Spring IoC注解式开发

12.1 回顾注解

注解的存在主要是为了简化XML的配置。Spring6倡导全注解开发。

我们来回顾一下：

- 第一：注解怎么定义，注解中的属性怎么定义？
- 第二：注解怎么使用？
- 第三：通过反射机制怎么读取注解？

注解怎么定义，注解中的属性怎么定义？

```
自定义注解 Java | 复制代码  
1 package com.powernode.annotation;  
2  
3 import java.lang.annotation.ElementType;  
4 import java.lang.annotation.Retention;  
5 import java.lang.annotation.RetentionPolicy;  
6 import java.lang.annotation.Target;  
7  
8 @Target(value = {ElementType.TYPE})  
9 @Retention(value = RetentionPolicy.RUNTIME)  
10 public @interface Component {  
11     String value();  
12 }
```

以上是自定义了一个注解：Component

该注解上面修饰的注解包括：Target注解和Retention注解，这两个注解被称为元注解。

Target注解用来设置Component注解可以出现的位置，以上代表表示Component注解只能用在类和接口上。

Retention注解用来设置Component注解的保持性策略，以上代表Component注解可以被反射机制读取。

String value(); 是Component注解中的一个属性。该属性类型String，属性名是value。

注解怎么使用？

```
▼ User Java | 复制代码
1 package com.powernode.bean;
2
3 import com.powernode.annotation.Component;
4
5 @Component(value = "userBean")
6 ▼ public class User {
7 }
8
```

用法简单，语法格式：@注解类型名(属性名=属性值, 属性名=属性值, 属性名=属性值.....)

userBean为什么使用双引号括起来，因为value属性是String类型，字符串。

另外如果属性名是value，则在使用的时候可以省略属性名，例如：

```
▼ User Java | 复制代码
1 package com.powernode.bean;
2
3 import com.powernode.annotation.Component;
4
5 //{@Component(value = "userBean")}
6 @Component("userBean")
7 ▼ public class User {
8 }
9
```

通过反射机制怎么读取注解？

接下来，我们来写一段程序，当Bean类上有Component注解时，则实例化Bean对象，如果没有，则不实例化对象。

我们准备两个Bean，一个上面有注解，一个上面没有注解。

有注解的Bean

Java | 复制代码

```
1 package com.powernode.bean;
2
3 import com.powernode.annotation.Component;
4
5 @Component("userBean")
6 public class User {
7 }
```

没有注解的Bean

Java | 复制代码

```
1 package com.powernode.bean;
2
3 public class Vip {
4 }
```

假设我们现在只知道包名：com.powernode.bean。至于这个包下有多少个Bean我们不知道。哪些Bean上有注解，哪些Bean上没有注解，这些我们都不知道，如何通过程序全自动化判断。

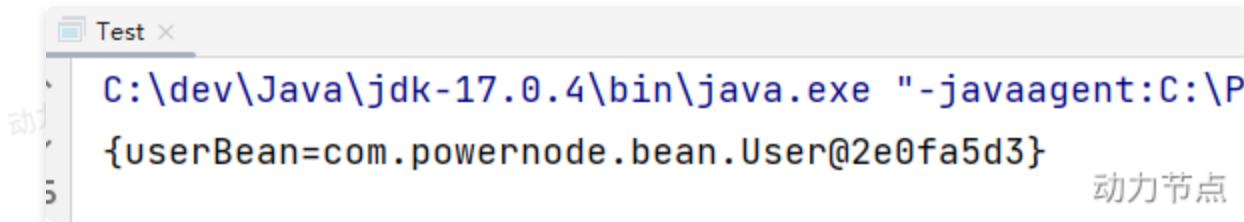
反射解析注解

Java | 复制代码

```
1 package com.powernode.test;
2
3 import com.powernode.annotation.Component;
4
5 import java.io.File;
6 import java.net.URL;
7 import java.util.Arrays;
8 import java.util.Enumeration;
9 import java.util.HashMap;
10 import java.util.Map;
11
12 /**
13 * @author 动力节点
14 * @version 1.0
15 * @className Test
16 * @since 1.0
17 */
18 public class Test {
19     public static void main(String[] args) throws Exception {
20         // 存放Bean的Map集合。key存储beanId。value存储Bean。
21         Map<String, Object> beanMap = new HashMap<>();
22
23         String packageName = "com.powernode.bean";
24         String path = packageName.replaceAll("\\.", "/");
25         URL url = ClassLoader.getSystemClassLoader().getResource(path);
26         File file = new File(url.getPath());
27         File[] files = file.listFiles();
28         Arrays.stream(files).forEach(f -> {
29             String className = packageName + "." + f.getName().split("\\.")[0];
30             try {
31                 Class<?> clazz = Class.forName(className);
32                 if (clazz.isAnnotationPresent(Component.class)) {
33                     Component component = clazz.getAnnotation(Component.class);
34                     String beanId = component.value();
35                     Object bean = clazz.newInstance();
36                     beanMap.put(beanId, bean);
37                 }
38             } catch (Exception e) {
39                 e.printStackTrace();
40             }
41         });
42
43         System.out.println(beanMap);
44     }
45 }
```

```
44     }
45 }
```

执行结果：



```
C:\dev\Java\jdk-17.0.4\bin\java.exe "-javaagent:C:\PowerNode\lib\powernode.jar" {userBean=com.powernode.bean.User@2e0fa5d3}
```

12.2 声明Bean的注解

负责声明Bean的注解，常见的包括四个：

- @Component
- @Controller
- @Service
- @Repository

源码如下：

```
▼ @Component注解
```

```
1 package com.powernode.annotation;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 @Target(value = {ElementType.TYPE})
9 @Retention(value = RetentionPolicy.RUNTIME)
10 public @interface Component {
11     String value();
12 }
```

Java | 复制代码

▼ @Controller注解

Java | 复制代码

```
1 package org.springframework.stereotype;
2
3 import java.lang.annotation.Documented;
4 import java.lang.annotation.ElementType;
5 import java.lang.annotation.Retention;
6 import java.lang.annotation.RetentionPolicy;
7 import java.lang.annotation.Target;
8 import org.springframework.core.annotation.AliasFor;
9
10 @Target({ElementType.TYPE})
11 @Retention(RetentionPolicy.RUNTIME)
12 @Documented
13 @Component
14 public @interface Controller {
15     @AliasFor(
16         annotation = Component.class
17     )
18     String value() default "";
19 }
20
```

▼ @Service注解

Java | 复制代码

```
1 package org.springframework.stereotype;
2
3 import java.lang.annotation.Documented;
4 import java.lang.annotation.ElementType;
5 import java.lang.annotation.Retention;
6 import java.lang.annotation.RetentionPolicy;
7 import java.lang.annotation.Target;
8 import org.springframework.core.annotation.AliasFor;
9
10 @Target({ElementType.TYPE})
11 @Retention(RetentionPolicy.RUNTIME)
12 @Documented
13 @Component
14 public @interface Service {
15     @AliasFor(
16         annotation = Component.class
17     )
18     String value() default "";
19 }
20
```

@Repository注解

Java | 复制代码

```
1 package org.springframework.stereotype;
2
3 import java.lang.annotation.Documented;
4 import java.lang.annotation.ElementType;
5 import java.lang.annotation.Retention;
6 import java.lang.annotation.RetentionPolicy;
7 import java.lang.annotation.Target;
8 import org.springframework.core.annotation.AliasFor;
9
10 @Target({ElementType.TYPE})
11 @Retention(RetentionPolicy.RUNTIME)
12 @Documented
13 @Component
14 public @interface Repository {
15     @AliasFor(
16         annotation = Component.class
17     )
18     String value() default "";
19 }
20
```

通过源码可以看到，@Controller、@Service、@Repository这三个注解都是@Component注解的别名。

也就是说：这四个注解的功能都一样。用哪个都可以。

只是为了增强程序的可读性，建议：

- 控制器类上使用：Controller
- service类上使用：Service
- dao类上使用：Repository

他们都是只有一个value属性。value属性用来指定bean的id，也就是bean的名字。

```
<bean id="userBean" class="com.powernode.spring6.bean.User"/>
```

value属性指定的就是bean的id

```
package com.powernode.spring6.bean;

import org.springframework.stereotype.Component;

@Component(value = "userBean")
public class User {
```

动力节点

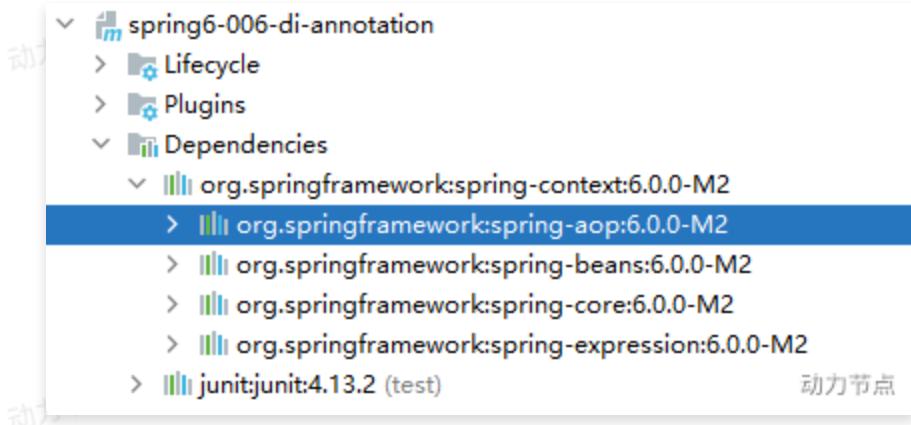
12.3 Spring注解的使用

如何使用以上的注解呢？

- 第一步：加入aop的依赖
- 第二步：在配置文件中添加context命名空间
- 第三步：在配置文件中指定扫描的包
- 第四步：在Bean类上使用注解

第一步：加入aop的依赖

我们可以看到当加入spring-context依赖之后，会关联加入aop的依赖。所以这一步不用做。



第二步：在配置文件中添加context命名空间

spring.xml

XML | 复制代码

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
6   http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">
7
8 </beans>
```

第三步：在配置文件中指定要扫描的包

spring.xml

XML | 复制代码

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
6   http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">
7   <context:component-scan base-package="com.powernode.spring6.bean"/>
8 </beans>
```

第四步：在Bean类上使用注解

User

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 import org.springframework.stereotype.Component;
4
5 @Component(value = "userBean")
6 public class User {
7 }
8
```

编写测试程序：

Java | 复制代码

```
1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.bean.User;
4 import org.junit.Test;
5 import org.springframework.context.ApplicationContext;
6 import org.springframework.context.support.ClassPathXmlApplicationContext;
7
8 public class AnnotationTest {
9     @Test
10    public void testBean(){
11        ApplicationContext applicationContext = new ClassPathXmlApplication
12        nContext("spring.xml");
13        User userBean = applicationContext.getBean("userBean", User.class)
14        ;
15    }
16}
```

执行结果:

Tests passed: 1 of 1 test – 467 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

com.powernode.spring6.bean.User@41fecb8b

动力节点

如果注解的属性名是value，那么value是可以省略的。

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 import org.springframework.stereotype.Component;
4
5 @Component("vipBean")
6 public class Vip {
7 }
```

测试程序

Java | 复制代码

```

1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.bean.Vip;
4 import org.junit.Test;
5 import org.springframework.context.ApplicationContext;
6 import org.springframework.context.support.ClassPathXmlApplicationContext;
7
8 public class AnnotationTest {
9     @Test
10    public void testBean(){
11        ApplicationContext applicationContext = new ClassPathXmlApplication
12        context("spring.xml");
13        Vip vipBean = applicationContext.getBean("vipBean", Vip.class);
14        System.out.println(vipBean);
15    }
16}

```

执行结果：

Tests passed: 1 of 1 test – 391 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

com.powernode.spring6.bean.Vip@51dcb805

如果把value属性彻底去掉，spring会被Bean自动取名吗？会的。并且默认名字的规律是：Bean类名首字母小写即可。

```

1 package com.powernode.spring6.bean;
2
3 import org.springframework.stereotype.Component;
4
5 @Component
6 public class BankDao {
7 }

```

也就是说，这个BankDao的bean的名字为：bankDao

测试一下

Java | 复制代码

```
1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.bean.BankDao;
4 import org.junit.Test;
5 import org.springframework.context.ApplicationContext;
6 import org.springframework.context.support.ClassPathXmlApplicationContext;
7
8 public class AnnotationTest {
9     @Test
10    public void testBean(){
11        ApplicationContext applicationContext = new ClassPathXmlApplication
12        nContext("spring.xml");
13        BankDao bankDao = applicationContext.getBean("bankDao", BankDao.cl
14        ass);
15        System.out.println(bankDao);
16    }
17}
```

执行结果:

```
Tests passed: 1 of 1 test – 500 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
com.powernode.spring6.bean.BankDao@6631f5ca
```

我们将Component注解换成其它三个注解，看看是否可以用：

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 import org.springframework.stereotype.Controller;
4
5 @Controller
6 public class BankDao {
7 }
8
```

执行结果:

Tests passed: 1 of 1 test – 390 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

com.powernode.spring6.bean.BankDao@1a4013

动力节点

剩下的两个注解大家可以测试一下。

如果是多个包怎么办？有两种解决方案：

- 第一种：在配置文件中指定多个包，用逗号隔开。
- 第二种：指定多个包的共同父包。

先来测试一下逗号（英文）的方式：

创建一个新的包：bean2，定义一个Bean类。

bean2包下的Order

```
1 package com.powernode.spring6.bean2;
2
3 import org.springframework.stereotype.Service;
4
5 @Service
6 public class Order {
7 }
```

Java | 复制代码

配置文件修改：

spring.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
6           http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">
7     <context:component-scan base-package="com.powernode.spring6.bean,com.powernode.spring6.bean2"/>
8 </beans>
```

XML | 复制代码

测试程序：

```

1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.bean.BankDao;
4 import com.powernode.spring6.bean2.Order;
5 import org.junit.Test;
6 import org.springframework.context.ApplicationContext;
7 import org.springframework.context.support.ClassPathXmlApplicationContext;
8
9 public class AnnotationTest {
10     @Test
11     public void testBean(){
12         ApplicationContext applicationContext = new ClassPathXmlApplication
13         nContext("spring.xml");
14         BankDao bankDao = applicationContext.getBean("bankDao", BankDao.cl
15         ass);
16         System.out.println(bankDao);
17         Order order = applicationContext.getBean("order", Order.class);
18         System.out.println(order);
19     }
20 }

```

执行结果：

Tests passed: 1 of 1 test – 418 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

com.powernode.spring6.bean.BankDao@15bbf42f

com.powernode.spring6.bean2.Order@550ee7e5

我们再来看看，指定共同的父包行不行：

```
spring.xml XML | 复制代码

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
6   http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">
7     <context:component-scan base-package="com.powernode.spring6"/>
8   </beans>
```

执行测试程序：

```
Tests passed: 1 of 1 test – 398 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
com.powernode.spring6.bean.BankDao@d29f28
com.powernode.spring6.bean2.Order@2fd1433e
```

12.4 选择性实例化Bean

假设在某个包下有很多Bean，有的Bean上标注了Component，有的标注了Controller，有的标注了Service，有的标注了Repository，现在由于某种特殊业务的需要，只允许其中所有的Controller参与Bean管理，其他的都不实例化。这应该怎么办呢？

这里为了方便，将这几个类都定义到同一个java源文件中了

Java | 复制代码

```
1 package com.powernode.spring6.bean3;
2
3 import org.springframework.stereotype.Component;
4 import org.springframework.stereotype.Controller;
5 import org.springframework.stereotype.Repository;
6 import org.springframework.stereotype.Service;
7
8 @Component
9 public class A {
10    public A() {
11        System.out.println("A的无参数构造方法执行");
12    }
13 }
14
15 @Controller
16 public class B {
17    public B() {
18        System.out.println("B的无参数构造方法执行");
19    }
20 }
21
22 @Service
23 public class C {
24    public C() {
25        System.out.println("C的无参数构造方法执行");
26    }
27 }
28
29 @Repository
30 public class D {
31    public D() {
32        System.out.println("D的无参数构造方法执行");
33    }
34 }
35
36 @Controller
37 public class E {
38    public E() {
39        System.out.println("E的无参数构造方法执行");
40    }
41 }
42
43 @Controller
44 public class F {
45    public F() {
```

```
46         System.out.println("F的无参数构造方法执行");
47     }
48 }
49 }
```

我只想实例化bean3包下的Controller。配置文件这样写：

```
spring-choose.xml
```

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
6           http://www.springframework.org/schema/context/spring-context.xsd">
7
8   <context:component-scan base-package="com.powernode.spring6.bean3" use-
9     -default-filters="false">
10    <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
11  </context:component-scan>
12 </beans>
```

use-default-filters="true" 表示：使用spring默认的规则，只要有Component、Controller、Service、Repository中的任意一个注解标注，则进行实例化。

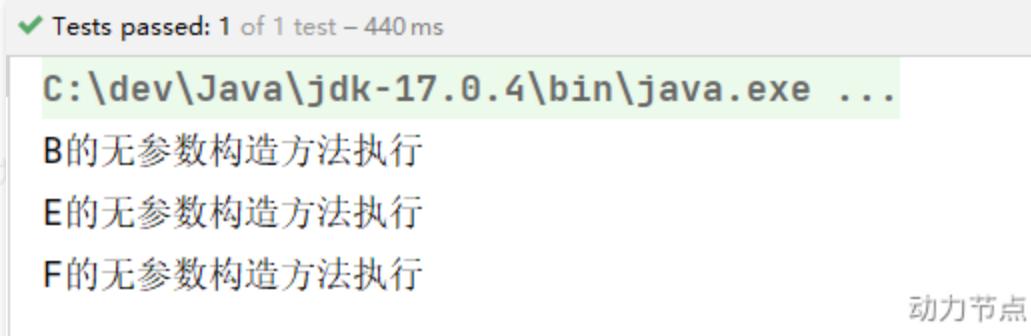
use-default-filters="false" 表示：不再spring默认实例化规则，即使有Component、Controller、Service、Repository这些注解标注，也不再实例化。

<context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/> 表示只有Controller进行实例化。

```
测试程序
```

```
1 @Test
2 public void testChoose(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationCont
4     ext("spring-choose.xml");
5 }
```

执行结果：

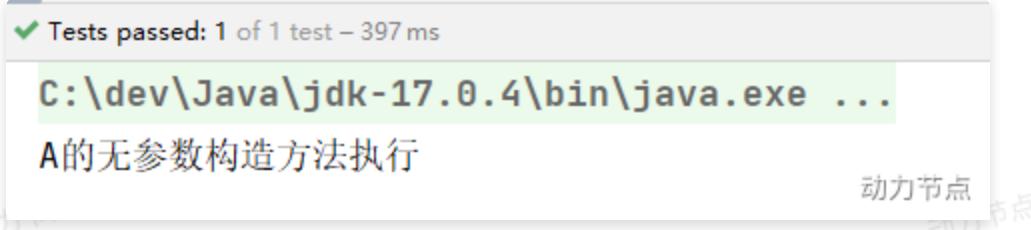


也可以将use-default-filters设置为true（不写就是true），并且采用exclude-filter方式排出哪些注解标注的Bean不参与实例化：

```
spring-choose.xml
```

```
1 <context:component-scan base-package="com.powernode.spring6.bean3">
2   <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Repository"/>
3   <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Service"/>
4   <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
5 </context:component-scan>
```

执行测试程序：



12.5 负责注入的注解

@Component @Controller @Service @Repository 这四个注解是用来声明Bean的，声明后这些Bean将被实例化。接下来我们看一下，如何给Bean的属性赋值。给Bean属性赋值需要用到这些注解：

- @Value
- @Autowired
- @Qualifier
- @Resource

12.5.1 @Value

当属性的类型是简单类型时，可以使用@Value注解进行注入。

```
1 package com.powernode.spring6.bean4;
2
3 import org.springframework.beans.factory.annotation.Value;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class User {
8     @Value(value = "zhangsan")
9     private String name;
10    @Value("20")
11    private int age;
12
13    @Override
14    public String toString() {
15        return "User{" +
16                "name='" + name + '\'' +
17                ", age=" + age +
18                '}';
19    }
20}
21
```

开启包扫描：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xmlns:context="http://www.springframework.org/schema/context"
5         xsi:schemaLocation="http://www.springframework.org/schema/beans
6                         http://www.springframework.org/schema/context
7                         http://www.springframework.org/schema/context/spring-context.xsd">
8     <context:component-scan base-package="com.powernode.spring6.bean4"/>
9 </beans>
```

测试程序

Java | 复制代码

```
1 @Test
2 public void testValue(){
3     ApplicationContext applicationContext = new ClassPathXmlApplicationCont
ext("spring-injection.xml");
4     Object user = applicationContext.getBean("user");
5     System.out.println(user);
6 }
```

执行结果：

```
Tests passed: 1 of 1 test – 413 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
User{name='zhangsan', age=20}
```

动力节点

通过以上代码可以发现，我们并没有给属性提供setter方法，但仍然可以完成属性赋值。

如果提供setter方法，并且在setter方法上添加@Value注解，可以完成注入吗？尝试一下：

```

1 package com.powernode.spring6.bean4;
2
3 import org.springframework.beans.factory.annotation.Value;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class User {
8
9     private String name;
10
11    private int age;
12
13    @Value("李四")
14    public void setName(String name) {
15        this.name = name;
16    }
17
18    @Value("30")
19    public void setAge(int age) {
20        this.age = age;
21    }
22
23    @Override
24    public String toString() {
25        return "User{" +
26                "name='" + name + '\'' +
27                ", age=" + age +
28                '}';
29    }
30}
31

```

执行结果：

Tests passed: 1 of 1 test – 489 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

User{name='李四', age=30}

动力节点

通过测试可以得知，@Value注解可以直接使用在属性上，也可以使用在setter方法上。都是可以的。都可以完成属性的赋值。

为了简化代码，以后我们一般不提供setter方法，直接在属性上使用@Value注解完成属性赋值。

出于好奇，我们再来测试一下，是否能够通过构造方法完成注入：

```
▼ User Java | 复制代码

1 package com.powernode.spring6.bean4;
2
3 import org.springframework.beans.factory.annotation.Value;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class User {
8
9     private String name;
10
11    private int age;
12
13    public User(@Value("隔壁老王") String name, @Value("33") int age) {
14        this.name = name;
15        this.age = age;
16    }
17
18    @Override
19    public String toString() {
20        return "User{" +
21                "name='" + name + '\'' +
22                ", age=" + age +
23                '}';
24    }
25 }
26
```

执行结果：

```
✓ Tests passed: 1 of 1 test – 421 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
User{name='隔壁老王', age=33}
```

通过测试得知：@Value注解可以出现在属性上、setter方法上、以及构造方法的形参上。可见Spring给我们提供了多样化的注入。太灵活了。

12.5.2 @Autowired与@Qualifier

@Autowired注解可以用来注入**非简单类型**。被翻译为：自动连线的，或者自动装配。

单独使用@Autowired注解，**默认根据类型装配**。【默认是byType】

看一下它的源码：

展开 @Autowired源码

Java | 复制代码

```
1 package org.springframework.beans.factory.annotation;
2
3 import java.lang.annotation.Documented;
4 import java.lang.annotation.ElementType;
5 import java.lang.annotation.Retention;
6 import java.lang.annotation.RetentionPolicy;
7 import java.lang.annotation.Target;
8
9 @Target({ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.PARAMETER,
10 ElementType.FIELD, ElementType.ANNOTATION_TYPE})
11 @Retention(RetentionPolicy.RUNTIME)
12 @Documented
13 public @interface Autowired {
14     boolean required() default true;
15 }
```

源码中有两处需要注意：

- 第一处：该注解可以标注在哪里？
 - 构造方法上
 - 方法上
 - 形参上
 - 属性上
 - 注解上
- 第二处：该注解有一个required属性，默认值是true，表示在注入的时候要求被注入的Bean必须是存在的，如果不存在则报错。如果required属性设置为false，表示注入的Bean存在或者不存在都没关系，存在的话就注入，不存在的话，也不报错。

我们先在属性上使用@Autowired注解：

展开 UserDao接口

Java | 复制代码

```
1 package com.powernode.spring6.dao;
2
3 public interface UserDao {
4     void insert();
5 }
```

UserDao实现类

Java | 复制代码

```
1 package com.powernode.spring6.dao;
2
3 import org.springframework.stereotype.Repository;
4
5 @Repository //纳入bean管理
6 public class UserDaoForMySQL implements UserDao{
7     @Override
8     public void insert() {
9         System.out.println("正在向mysql数据库插入User数据");
10    }
11 }
12 
```

UserService

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.stereotype.Service;
6
7 @Service // 纳入bean管理
8 public class UserService {
9
10     @Autowired // 在属性上注入
11     private UserDao userDao;
12
13     // 没有提供构造方法和setter方法。
14
15     public void save(){
16         userDao.insert();
17     }
18 }
```

配置包扫描

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
6   http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">
7     <context:component-scan base-package="com.powernode.spring6.dao,com.powernode.spring6.service"/>
8   </beans>
```

测试程序

```
1 @Test
2 public void testAutowired(){
3   ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring-injection.xml");
4   UserService userService = applicationContext.getBean("userService", UserService.class);
5   userService.save();
6 }
```

执行结果：

```
Tests passed: 1 of 1 test - 410 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
正在向mysql数据库插入User数据
```

以上构造方法和setter方法都没有提供，经过测试，仍然可以注入成功。

接下来，再来测试一下@Autowired注解出现在setter方法上：

```
UserService Java | 复制代码

1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.stereotype.Service;
6
7 @Service
8 public class UserService {
9
10     private UserDao userDao;
11
12     @Autowired
13     public void setUserDao(UserDao userDao) {
14         this.userDao = userDao;
15     }
16
17     public void save(){
18         userDao.insert();
19     }
20 }
21
```

执行结果：

Tests passed: 1 of 1 test – 438 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

正在向mysql数据库插入User数据

动力节点

我们再来看看能不能出现在构造方法上：

```
UserService Java | 复制代码

1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.stereotype.Service;
6
7 @Service
8 public class UserService {
9
10     private UserDao userDao;
11
12     @Autowired
13     public UserService(UserDao userDao) {
14         this.userDao = userDao;
15     }
16
17     public void save(){
18         userDao.insert();
19     }
20 }
21
```

执行结果：

```
Tests passed: 1 of 1 test – 455 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
正在向mysql数据库插入User数据
```

再来看看，这个注解能不能只标注在构造方法的形参上：

```
UserService Java | 复制代码

1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.stereotype.Service;
6
7 @Service
8 public class UserService {
9
10     private UserDao userDao;
11
12     public UserService(@Autowired UserDao userDao) {
13         this.userDao = userDao;
14     }
15
16     public void save(){
17         userDao.insert();
18     }
19 }
20
```

执行结果：

```
Tests passed: 1 of 1 test – 408 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
正在向mysql数据库插入User数据
动力节点
```

还有更劲爆的，当有参数的构造方法只有一个时，@Autowired注解可以省略。

```
UserService Java | 复制代码

1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4 import org.springframework.stereotype.Service;
5
6 @Service
7 public class UserService {
8
9     private UserDao userDao;
10
11     public UserService(UserDao userDao) {
12         this.userDao = userDao;
13     }
14
15     public void save(){
16         userDao.insert();
17     }
18 }
19
```

执行结果：

```
Tests passed: 1 of 1 test – 408 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
正在向mysql数据库插入User数据
```

当然，如果有多个构造方法，@Autowired肯定是不能省略的。

```
UserService
Java | 复制代码

1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4 import org.springframework.stereotype.Service;
5
6 @Service
7 public class UserService {
8
9     private UserDao userDao;
10
11     public UserService(UserDao userDao) {
12         this.userDao = userDao;
13     }
14
15     public UserService(){
16
17     }
18
19     public void save(){
20         userDao.insert();
21     }
22 }
23
```

执行结果：

```
Tests failed: 1 of 1 test – 395 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
java.lang.NullPointerException: Cannot invoke "com.powernode.spring6.dao.UserDao.insert()" because "this.userDao" is null
    at com.powernode.spring6.service.UserService.save(UserService.java:20)
    at com.powernode.spring6.test.AnnotationTest.testAutowired(AnnotationTest.java:15) <27 internal lines> 动力节点
```

到此为止，我们已经清楚@Autowired注解可以出现在哪些位置了。

@Autowired注解默认是byType进行注入的，也就是说根据类型注入的，如果以上程序中，UserDao接口还有另外一个实现类，会出现问题吗？

UserDaoForOracle, 接口另一个实现类

Java | 复制代码

```
1 package com.powernode.spring6.dao;
2
3 import org.springframework.stereotype.Repository;
4
5 @Repository //纳入bean管理
6 public class UserDaoForOracle implements UserDao{
7     @Override
8     public void insert() {
9         System.out.println("正在向Oracle数据库插入User数据");
10    }
11 }
12
```

当你写完这个新的实现类之后，此时IDEA工具已经提示错误信息了：

The screenshot shows the IntelliJ IDEA code editor with the following Java code:

```
1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.stereotype.Service;
6
7 @Service
8 public class UserService {
9
10     2 usages
11     private UserDao userDao;
12
13     @Autowired
14     public void setUserDao(UserDao userDao) {
15         this.userDao = userDao;
16     }
17
18     1 usage
19     public void save(){
20         userDao.insert();
21     }
22 }
```

A tooltip window is open over the `@Autowired` annotation on line 12. The tooltip contains the following text:

Could not autowire. There is more than one bean of 'UserDao' type.

Beans: userDaoForMySQL (UserDaoForMySQL.java)
userDaoForOracle (UserDaoForOracle.java)

Add qualifier Alt+Shift+Enter More actions... Alt+Enter

UserDao userDao

spring6-006-di-annotation

错误信息中说：不能装配， UserDao这个Bean的数量大于1.

怎么解决这个问题呢？**当然要byName, 根据名称进行装配了。**

@Autowired注解和@Qualifier注解联合起来才可以根据名称进行装配，在@Qualifier注解中指定Bean名称。

UserDaoForOracle

Java | 复制代码

```
1 package com.powernode.spring6.dao;
2
3 import org.springframework.stereotype.Repository;
4
5 @Repository // 这里没有给bean起名，默认名字是: userDaoForOracle
6 public class UserDaoForOracle implements UserDao{
7     @Override
8     public void insert() {
9         System.out.println("正在向Oracle数据库插入User数据");
10    }
11 }
12
```

UserService

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.beans.factory.annotation.Qualifier;
6 import org.springframework.stereotype.Service;
7
8 @Service
9 public class UserService {
10
11     private UserDao userDao;
12
13     @Autowired
14     @Qualifier("userDaoForOracle") // 这个是bean的名字。
15     public void setUserDao(UserDao userDao) {
16         this.userDao = userDao;
17     }
18
19     public void save(){
20         userDao.insert();
21     }
22 }
```

执行结果:

Tests passed: 1 of 1 test – 437 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

正在向Oracle数据库插入User数据

动力节点

总结：

- @Autowired注解可以出现在：属性上、构造方法上、构造方法的参数上、setter方法上。
- 当带参数的构造方法只有一个，@Autowired注解可以省略。
- @Autowired注解默认根据类型注入。如果要根据名称注入的话，需要配合@Qualifier注解一起使用。

12.5.3 @Resource

@Resource注解也可以完成非简单类型注入。那它和@Autowired注解有什么区别？

- @Resource注解是JDK扩展包中的，也就是说属于JDK的一部分。所以该注解是标准注解，更加具有通用性。（JSR-250标准中制定的注解类型。JSR是Java规范提案。）
- @Autowired注解是Spring框架自己的。
- @Resource注解默认根据名称装配byName，未指定name时，使用属性名作为name。通过name找不到的话会自动启动通过类型byType装配。
- @Autowired注解默认根据类型装配byType，如果想根据名称装配，需要配合@Qualifier注解一起用。
- @Resource注解用在属性上、setter方法上。
- @Autowired注解用在属性上、setter方法上、构造方法上、构造方法参数上。

@Resource注解属于JDK扩展包，所以不在JDK当中，需要额外引入以下依赖：【**如果是JDK8的话不需要额外引入依赖。高于JDK11或低于JDK8需要引入以下依赖。**】

如果你是Spring6+版本请使用这个依赖

XML | 复制代码

```
1 <dependency>
2   <groupId>jakarta.annotation</groupId>
3   <artifactId>jakarta.annotation-api</artifactId>
4   <version>2.1.1</version>
5 </dependency>
```

一定要注意：**如果你用Spring6，要知道Spring6不再支持JavaEE，它支持的是JakartaEE9。（Oracle把JavaEE贡献给Apache了，Apache把JavaEE的名字改成JakartaEE了，大家之前所接触的所有javax.* 包名统一修改为 jakarta.*包名了。）**

如果你是spring5–版本请使用这个依赖

XML | 复制代码

```
1 <dependency>
2   <groupId>javax.annotation</groupId>
3   <artifactId>javax.annotation-api</artifactId>
4   <version>1.3.2</version>
5 </dependency>
```

@Resource注解的源码如下：

```
4 @Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
5 @Retention(RetentionPolicy.RUNTIME)
6 @Repeatable(Resources.class)
7 public @interface Resource {
8     String name() default ""; // name属性用来接收bean的名称
9
10    String lookup() default "";
11
12    Class<?> type() default Object.class;
13
14    AuthenticationType authenticationType() default Resource.AuthenticationType.CONTAINER;
15
16    boolean shareable() default true;
```

测试一下：

给这个UserDaoForOracle起名xyz

Java | 复制代码

```
1 package com.powernode.spring6.dao;
2
3 import org.springframework.stereotype.Repository;
4
5 @Repository("xyz")
6 public class UserDaoForOracle implements UserDao{
7     @Override
8     public void insert() {
9         System.out.println("正在向Oracle数据库插入User数据");
10    }
11 }
```

在UserService中使用Resource注解根据name注入

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4 import jakarta.annotation.Resource;
5 import org.springframework.stereotype.Service;
6
7 @Service
8 public class UserService {
9
10     @Resource(name = "xyz")
11     private UserDao userDao;
12
13     public void save(){
14         userDao.insert();
15     }
16 }
```

执行测试程序：

```
Tests passed: 1 of 1 test – 429 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
正在向Oracle数据库插入User数据
```

我们把UserDaoForOracle的名字xyz修改为userDao，让这个Bean的名字和UserService类中的 UserDao属性名一致：

UserDaoForOracle

Java | 复制代码

```
1 package com.powernode.spring6.dao;
2
3 import org.springframework.stereotype.Repository;
4
5 @Repository("userDao")
6 public class UserDaoForOracle implements UserDao{
7     @Override
8     public void insert() {
9         System.out.println("正在向Oracle数据库插入User数据");
10    }
11 }
12
```

UserService类中Resource注解并没有指定name

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4 import jakarta.annotation.Resource;
5 import org.springframework.stereotype.Service;
6
7 @Service
8 public class UserService {
9
10     @Resource
11     private UserDao userDao;
12
13     public void save(){
14         userDao.insert();
15     }
16 }
17
```

执行测试程序：

Tests passed: 1 of 1 test – 453 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

正在向Oracle数据库插入User数据

动力节点

通过测试得知，当@Resource注解使用时没有指定name的时候，还是根据name进行查找，这个name是属性名。

接下来把UserService类中的属性名修改一下：

UserService的属性名修改为userDao2

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4 import jakarta.annotation.Resource;
5 import org.springframework.stereotype.Service;
6
7 @Service
8 public class UserService {
9
10     @Resource
11     private UserDao userDao2;
12
13     public void save(){
14         userDao2.insert();
15     }
16 }
17
```

执行结果：

```
Tests failed: 1 of 1 test – 549 ms

option is org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type 'com.powernode.spring6.dao.UserDao' available: expected single matching bean but found 2: userDaoForMySQL,userDao
    
```

动力节点

根据异常信息得知：显然当通过name找不到的时候，自然会启动byType进行注入。以上的错误是因为 UserDao接口下有两个实现类导致的。所以根据类型注入就会报错。

我们再来看@Resource注解使用在setter方法上可以吗？

UserService添加setter方法并使用注解标注

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4 import jakarta.annotation.Resource;
5 import org.springframework.stereotype.Service;
6
7 @Service
8 public class UserService {
9
10     private UserDao userDao;
11
12     @Resource
13     public void setUserDao(UserDao userDao) {
14         this.userDao = userDao;
15     }
16
17     public void save(){
18         userDao.insert();
19     }
20 }
21
```

注意这个setter方法的方法名，setUserDao去掉set之后，将首字母变小写userDao，userDao就是name
执行结果：

Tests passed: 1 of 1 test – 479 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

正在向Oracle数据库插入User数据

动力节点

当然，也可以指定name：

```
UserService Java | 复制代码

1 package com.powernode.spring6.service;
2
3 import com.powernode.spring6.dao.UserDao;
4 import jakarta.annotation.Resource;
5 import org.springframework.stereotype.Service;
6
7 @Service
8 public class UserService {
9
10     private UserDao userDao;
11
12     @Resource(name = "userDaoForMySQL")
13     public void setUserDao(UserDao userDao) {
14         this.userDao = userDao;
15     }
16
17     public void save(){
18         userDao.insert();
19     }
20 }
21
```

执行结果：

```
Tests passed: 1 of 1 test – 494 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
正在向mysql数据库插入User数据          动力节点
```

一句话总结@Resource注解：默认byName注入，没有指定name时把属性名当做name，根据name找不到时，才会byType注入。byType注入时，某种类型的Bean只能有一个。

12.6 全注解式开发

所谓的全注解开发就是不再使用spring配置文件了。写一个配置类来代替配置文件。

配置类代替spring配置文件

Java | 复制代码

```
1 package com.powernode.spring6.config;
2
3 import org.springframework.context.annotation.ComponentScan;
4 import org.springframework.context.annotation.ComponentScans;
5 import org.springframework.context.annotation.Configuration;
6
7 @Configuration
8 @ComponentScan({"com.powernode.spring6.dao", "com.powernode.spring6.service"})
9 public class Spring6Configuration {
10 }
11
```

编写测试程序：不再new ClassPathXmlApplicationContext()对象了。

```
1 @Test
2 public void testNoXml(){
3     ApplicationContext applicationContext = new AnnotationConfigApplicationContext(Spring6Configuration.class);
4     UserService userService = applicationContext.getBean("userService", UserService.class);
5     userService.save();
6 }
```

执行结果：

Tests passed: 1 of 1 test – 415 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

正在向mysql数据库插入User数据

动力节点

十三、JdbcTemplate

JdbcTemplate是Spring提供的一个JDBC模板类，是对JDBC的封装，简化JDBC代码。

当然，你也可以不用，可以让Spring集成其它的ORM框架，例如：MyBatis、Hibernate等。

接下来我们简单来学习一下，使用JdbcTemplate完成增删改查。

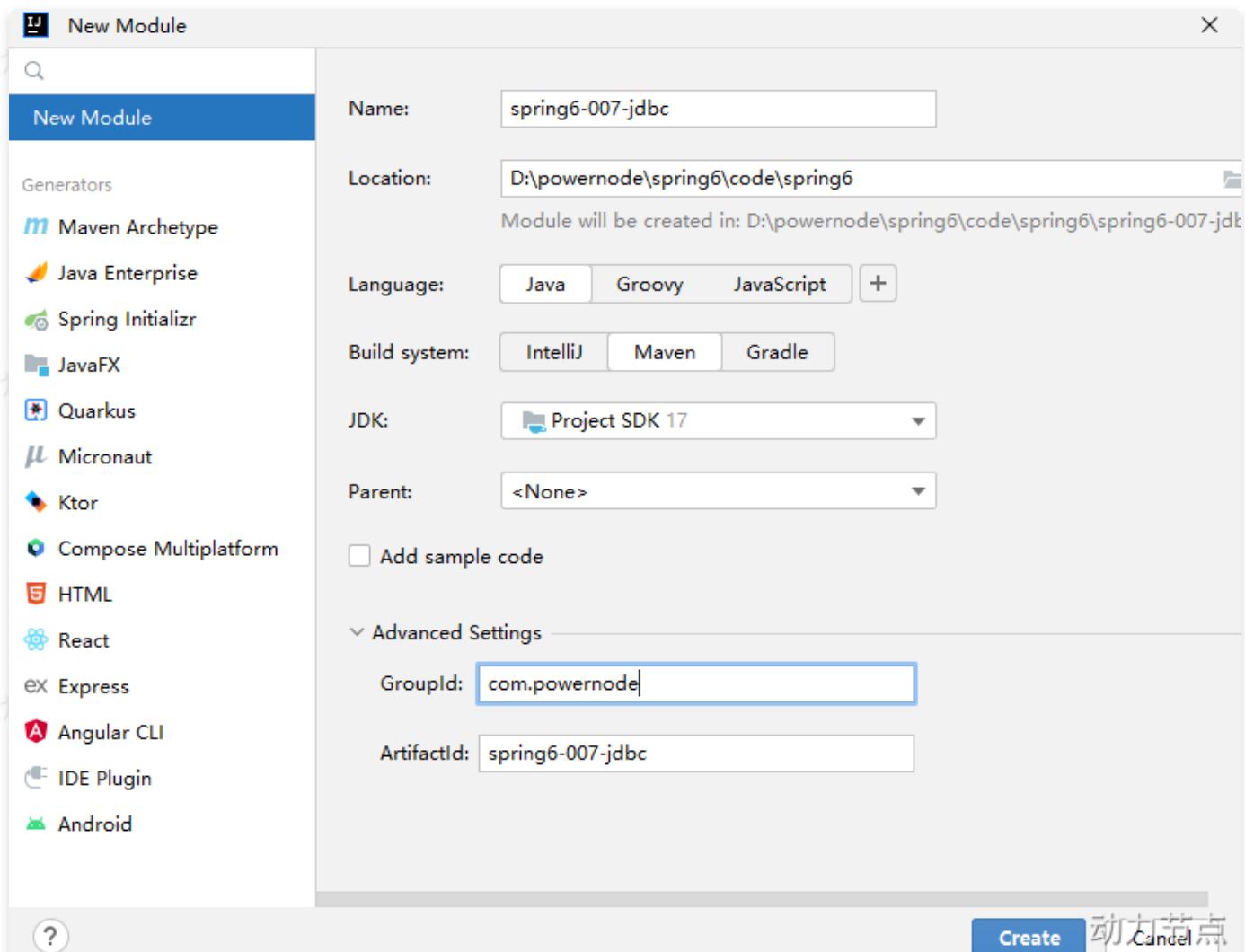
13.1 环境准备

数据库表: t_user

The screenshot shows the MySQL Workbench interface with the database 'spring6' selected. A table named 't_user' is displayed with the following structure:

名	类型	长度	小数点	不是 null	虚拟	键	注释
id	int			<input checked="" type="checkbox"/>	<input type="checkbox"/>	🔑 1	主键自增
real_name	varchar	255		<input type="checkbox"/>	<input type="checkbox"/>		真实姓名
age	int			<input type="checkbox"/>	<input type="checkbox"/>		年龄

IDEA中新建模块: spring6-007-jdbc



引入相关依赖:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>com.powernode</groupId>
8     <artifactId>spring6-007-jdbc</artifactId>
9     <version>1.0-SNAPSHOT</version>
10    <packaging>jar</packaging>
11
12    <repositories>
13      <repository>
14        <id>repository.spring.milestone</id>
15        <name>Spring Milestone Repository</name>
16        <url>https://repo.spring.io/milestone</url>
17      </repository>
18    </repositories>
19
20    <dependencies>
21      <dependency>
22        <groupId>org.springframework</groupId>
23        <artifactId>spring-context</artifactId>
24        <version>6.0.0-M2</version>
25      </dependency>
26      <dependency>
27        <groupId>junit</groupId>
28        <artifactId>junit</artifactId>
29        <version>4.13.2</version>
30        <scope>test</scope>
31      </dependency>
32      <!--新增的依赖:mysql驱动-->
33      <dependency>
34        <groupId>mysql</groupId>
35        <artifactId>mysql-connector-java</artifactId>
36        <version>8.0.30</version>
37      </dependency>
38      <!--新增的依赖: spring jdbc, 这个依赖中有JdbcTemplate-->
39      <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-jdbc</artifactId>
42        <version>6.0.0-M2</version>
43      </dependency>
44    </dependencies>
```

```
45
46      <properties>
47          <maven.compiler.source>17</maven.compiler.source>
48          <maven.compiler.target>17</maven.compiler.target>
49      </properties>
50
51  </project>
```

准备实体类：表t_user对应的实体类User。

```
1 package com.powernode.spring6.bean;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className User
7  * @since 1.0
8 */
9 public class User {
10     private Integer id;
11     private String realName;
12     private Integer age;
13
14     @Override
15     public String toString() {
16         return "User{" +
17             "id=" + id +
18             ", realName='" + realName + '\'' +
19             ", age=" + age +
20             '}';
21     }
22
23     public User() {
24     }
25
26     public User(Integer id, String realName, Integer age) {
27         this.id = id;
28         this.realName = realName;
29         this.age = age;
30     }
31
32     public Integer getId() {
33         return id;
34     }
35
36     public void setId(Integer id) {
37         this.id = id;
38     }
39
40     public String getRealName() {
41         return realName;
42     }
43
44     public void setRealName(String realName) {
45         this.realName = realName;
```

```
46     }
47
48     public Integer getAge() {
49         return age;
50     }
51
52     public void setAge(Integer age) {
53         this.age = age;
54     }
55 }
56 }
```

编写Spring配置文件：

JdbcTemplate是Spring提供好的类，这类的完整类名是：

org.springframework.jdbc.core.JdbcTemplate

我们怎么使用这个类呢？new对象就可以了。怎么new对象，Spring最在行了。直接将这个类配置到Spring配置文件中，纳入Bean管理即可。

```
spring.xml XML 复制代码
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      <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"></bean>
6  </beans>
```

我们来看一下这个JdbcTemplate源码：

```

1 usage
public class JdbcTemplate extends JdbcAccessor implements JdbcOperations {
    private static final String RETURN_RESULT_SET_PREFIX = "#result-set-";
    private static final String RETURN_UPDATE_COUNT_PREFIX = "#update-count-";
    private boolean ignoreWarnings = true;
    private int fetchSize = -1;
    private int maxRows = -1;
    private int queryTimeout = -1;
    private boolean skipResultsProcessing = false;
    private boolean skipUndeclaredResults = false;
    private boolean resultsMapCaseInsensitive = false;

    public JdbcTemplate() {
    }

    public JdbcTemplate(DataSource dataSource) {
        this.setDataSource(dataSource);
        this.afterPropertiesSet();
    }

```

动力节点

```

1 inheritor
public abstract class JdbcAccessor implements InitializingBean {
    private static final boolean shouldIgnoreXml = SpringProperties.getFlag(key: "spring.xml.ignore");
    protected final Log logger = LoggerFactory.getLog(this.getClass());
    @Nullable
    private DataSource dataSource;
    @Nullable
    private volatile SQLExceptionTranslator exceptionTranslator;
    private boolean lazyInit = true;

    public JdbcAccessor() {
    }

```

动力节点

可以看到JdbcTemplate中有一个DataSource属性，这个属性是数据源，我们都知道连接数据库需要Connection对象，而生成Connection对象是数据源负责的。所以我们需要给JdbcTemplate设置数据源属性。

所有的数据源都是要实现javax.sql.DataSource接口的。这个数据源可以自己写一个，也可以用写好的，比如：阿里巴巴的德鲁伊连接池，c3p0，dbcp等。我们这里自己先手写一个数据源。

自己写的数据源

Java | 复制代码

```
1 package com.powernode.spring6.jdbc;
2
3 import javax.sql.DataSource;
4 import java.io.PrintWriter;
5 import java.sql.Connection;
6 import java.sql.DriverManager;
7 import java.sql.SQLException;
8 import java.sql.SQLFeatureNotSupportedException;
9 import java.util.logging.Logger;
10
11 /**
12  * @author 动力节点
13  * @version 1.0
14  * @className MyDataSource
15  * @since 1.0
16  */
17 public class MyDataSource implements DataSource {
18     // 添加4个属性
19     private String driver;
20     private String url;
21     private String username;
22     private String password;
23
24     // 提供4个setter方法
25     public void setDriver(String driver) {
26         this.driver = driver;
27     }
28
29     public void setUrl(String url) {
30         this.url = url;
31     }
32
33     public void setUsername(String username) {
34         this.username = username;
35     }
36
37     public void setPassword(String password) {
38         this.password = password;
39     }
40
41     // 重点写怎么获取Connection对象就行。其他方法不用管。
42     @Override
43     public Connection getConnection() throws SQLException {
44         try {
45             Class.forName(driver);
```

```
46             Connection conn = DriverManager.getConnection(url, username, p
47             assword);
48         } catch (Exception e) {
49             e.printStackTrace();
50         }
51         return null;
52     }
53
54     @Override
55     public Connection getConnection(String username, String password) throws
56     SQLException {
57         return null;
58     }
59
60     @Override
61     public PrintWriter getLogWriter() throws SQLException {
62         return null;
63     }
64
65     @Override
66     public void setLogWriter(PrintWriter out) throws SQLException {
67
68     }
69
70     @Override
71     public void setLoginTimeout(int seconds) throws SQLException {
72
73     }
74
75     @Override
76     public int getLoginTimeout() throws SQLException {
77         return 0;
78     }
79
80     @Override
81     public Logger getParentLogger() throws SQLFeatureNotSupportedException
82     {
83         return null;
84     }
85
86     @Override
87     public <T> T unwrap(Class<T> iface) throws SQLException {
88         return null;
89     }
90
91     @Override
92     public boolean isWrapperFor(Class<?> iface) throws SQLException {
```

```
91         return false;
92     }
93 }
94 }
```

写完数据源，我们需要把这个数据源传递给JdbcTemplate。因为JdbcTemplate中有一个DataSource属性：

spring.xml

XML | 复制代码

```
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
6     <bean id="myDataSource" class="com.powernode.spring6.jdbc.MyDataSource">
7         <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
8         <property name="url" value="jdbc:mysql://localhost:3306/spring6"/>
9         <property name="username" value="root"/>
10        <property name="password" value="123456"/>
11    </bean>
12    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
13        <property name="dataSource" ref="myDataSource"/>
14    </bean>
15 </beans>
```

到这里环境就准备好了。

13.2 新增

编写测试程序：

```
1 package com.powernode.spring6.test;
2
3 import org.junit.Test;
4 import org.springframework.context.ApplicationContext;
5 import org.springframework.context.support.ClassPathXmlApplicationContext;
6 import org.springframework.jdbc.core.JdbcTemplate;
7
8 /**
9  * @author 动力节点
10 * @version 1.0
11 * @className JdbcTest
12 * @since 1.0
13 */
14 public class JdbcTest {
15     @Test
16     public void testInsert(){
17         // 获取JdbcTemplate对象
18         ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
19         JdbcTemplate jdbcTemplate = applicationContext.getBean("jdbcTemplate", JdbcTemplate.class);
20         // 执行插入操作
21         // 注意: insert delete update的sql语句, 都是执行update方法。
22         String sql = "insert into t_user(id,real_name,age) values(?, ?, ?)";
23         int count = jdbcTemplate.update(sql, null, "张三", 30);
24         System.out.println("插入的记录条数: " + count);
25     }
26 }
27
```

update方法有两个参数:

- 第一个参数: 要执行的SQL语句。 (SQL语句中可能会有占位符?)
- 第二个参数: 可变长参数, 参数的个数可以是0个, 也可以是多个。一般是SQL语句中有几个问号, 则对应几个参数。

13.3 修改

测试程序

Java | 复制代码

```
1  @Test
2  public void testUpdate(){
3      // 获取JdbcTemplate对象
4      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
5      JdbcTemplate jdbcTemplate = applicationContext.getBean("jdbcTemplate",
6          JdbcTemplate.class);
6      // 执行更新操作
7      String sql = "update t_user set real_name = ?, age = ? where id = ?";
8      int count = jdbcTemplate.update(sql, "张三丰", 55, 1);
9      System.out.println("更新的记录条数: " + count);
10 }
```

执行结果:

Tests passed: 1 of 1 test – 831 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

更新的记录条数: 1

动力节点

13.4 删除

测试程序

Java | 复制代码

```
1  @Test
2  public void testDelete(){
3      // 获取JdbcTemplate对象
4      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
5      JdbcTemplate jdbcTemplate = applicationContext.getBean("jdbcTemplate",
6          JdbcTemplate.class);
6      // 执行delete
7      String sql = "delete from t_user where id = ?";
8      int count = jdbcTemplate.update(sql, 1);
9      System.out.println("删除了几条记录: " + count);
10 }
```

执行结果:

Tests passed: 1 of 1 test – 804 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

删除了几条记录: 1

动力节点

13.5 查询一个对象

```
1  @Test
2  public void testSelectOne(){
3      // 获取JdbcTemplate对象
4      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
5      JdbcTemplate jdbcTemplate = applicationContext.getBean("jdbcTemplate",
6          JdbcTemplate.class);
7      // 执行select
8      String sql = "select id, real_name, age from t_user where id = ?";
9      User user = jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper
10     <>(User.class), 2);
11      System.out.println(user);
12 }
```

执行结果:

Tests passed: 1 of 1 test – 850 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

User{id=2, realName='张三', age=30}

动力节点

queryForObject方法三个参数:

- 第一个参数: sql语句
- 第二个参数: Bean属性值和数据库记录行的映射对象。在构造方法中指定映射的对象类型。
- 第三个参数: 可变长参数, 给sql语句的占位符问号传值。

13.6 查询多个对象

Java | 复制代码

```
1  @Test
2  public void testSelectAll(){
3      // 获取JdbcTemplate对象
4      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
5      JdbcTemplate jdbcTemplate = applicationContext.getBean("jdbcTemplate",
6          JdbcTemplate.class);
6      // 执行select
7      String sql = "select id, real_name, age from t_user";
8      List<User> users = jdbcTemplate.query(sql, new BeanPropertyRowMapper<>
9          (User.class));
9      System.out.println(users);
10 }
```

执行结果：

```
Tests passed: 1 of 1 test – 850 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
[User{id=2, realName='张三', age=30}, User{id=3, realName='李四', age=20}]
```

13.7 查询一个值

Java | 复制代码

```
1  @Test
2  public void testSelectOneValue(){
3      // 获取JdbcTemplate对象
4      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
5      JdbcTemplate jdbcTemplate = applicationContext.getBean("jdbcTemplate",
6          JdbcTemplate.class);
6      // 执行select
7      String sql = "select count(1) from t_user";
8      Integer count = jdbcTemplate.queryForObject(sql, int.class); // 这里用I
8      nteger.class也可以
9      System.out.println("总记录条数: " + count);
10 }
```

执行结果：

Tests passed: 1 of 1 test – 865 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

总记录条数: 2

动力节点

13.8 批量添加

Java | 复制代码

```
1  @Test
2  public void testAddBatch(){
3      // 获取JdbcTemplate对象
4      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
5      JdbcTemplate jdbcTemplate = applicationContext.getBean("jdbcTemplate",
6          JdbcTemplate.class);
7      // 批量添加
8      String sql = "insert into t_user(id,real_name,age) values(?, ?, ?)";
9
10     Object[] objs1 = {null, "小花", 20};
11     Object[] objs2 = {null, "小明", 21};
12     Object[] objs3 = {null, "小刚", 22};
13     List<Object[]> list = new ArrayList<>();
14     list.add(objs1);
15     list.add(objs2);
16     list.add(objs3);
17
18     int[] count = jdbcTemplate.batchUpdate(sql, list);
19 }
```

执行结果:

Tests passed: 1 of 1 test – 953 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

[1, 1, 1]

动力节点

13.9 批量修改

```
1  @Test
2  public void testUpdateBatch(){
3      // 获取JdbcTemplate对象
4      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
5      JdbcTemplate jdbcTemplate = applicationContext.getBean("jdbcTemplate",
6          JdbcTemplate.class);
6      // 批量修改
7      String sql = "update t_user set real_name = ?, age = ? where id = ?";
8      Object[] objs1 = {"小花11", 10, 2};
9      Object[] objs2 = {"小明22", 12, 3};
10     Object[] objs3 = {"小刚33", 9, 4};
11     List<Object[]> list = new ArrayList<>();
12     list.add(objs1);
13     list.add(objs2);
14     list.add(objs3);
15
16     int[] count = jdbcTemplate.batchUpdate(sql, list);
17     System.out.println(Arrays.toString(count));
18 }
```

执行结果：

```
Tests passed: 1 of 1 test – 883 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
[1, 1, 1]
```

13.10 批量删除

```

1  @Test
2  public void testDeleteBatch(){
3      // 获取JdbcTemplate对象
4      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
5      JdbcTemplate jdbcTemplate = applicationContext.getBean("jdbcTemplate",
6          JdbcTemplate.class);
7      // 批量删除
8      String sql = "delete from t_user where id = ?";
9      Object[] objs1 = {2};
10     Object[] objs2 = {3};
11     Object[] objs3 = {4};
12     List<Object[]> list = new ArrayList<>();
13     list.add(objs1);
14     list.add(objs2);
15     list.add(objs3);
16     int[] count = jdbcTemplate.batchUpdate(sql, list);
17     System.out.println(Arrays.toString(count));
18 }
```

执行结果：

Tests passed: 1 of 1 test - 871 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

[1, 1, 1]

13.11 使用回调函数

使用回调函数，可以参与的更加细节：

```

1  @Test
2  public void testCallback(){
3      // 获取JdbcTemplate对象
4      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
5      JdbcTemplate jdbcTemplate = applicationContext.getBean("jdbcTemplate",
6          JdbcTemplate.class);
6      String sql = "select id, real_name, age from t_user where id = ?";
7
8      User user = jdbcTemplate.execute(sql, new PreparedStatementCallback<User>() {
9          @Override
10         public User doInPreparedStatement(PreparedStatement ps) throws SQLException, DataAccessException {
11             User user = null;
12             ps.setInt(1, 5);
13             ResultSet rs = ps.executeQuery();
14             if (rs.next()) {
15                 user = new User();
16                 user.setId(rs.getInt("id"));
17                 user.setRealName(rs.getString("real_name"));
18                 user.setAge(rs.getInt("age"));
19             }
20             return user;
21         }
22     });
23     System.out.println(user);
24 }

```

执行结果：

```

✓ Tests passed: 1 of 1 test – 858 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
User{id=5, realName='小明', age=21}

```

动力节点

13.12 使用德鲁伊连接池

之前数据源是用我们自己写的。也可以使用别人写好的。例如比较牛的德鲁伊连接池。

第一步：引入德鲁伊连接池的依赖。（毕竟是别人写的）

德鲁伊依赖 pom.xml

XML | 复制代码

```
1 <dependency>
2   <groupId>com.alibaba</groupId>
3   <artifactId>druid</artifactId>
4   <version>1.1.8</version>
5 </dependency>
```

第二步：将德鲁伊中的数据源配置到spring配置文件中。和配置我们自己写的一样。

spring.xml

XML | 复制代码

```
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
6   <bean id="druidDataSource" class="com.alibaba.druid.pool.DruidDataSource">
7     <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
8     <property name="url" value="jdbc:mysql://localhost:3306/spring6"/>
9     <property name="username" value="root"/>
10    <property name="password" value="root"/>
11  </bean>
12
13  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
14    <property name="dataSource" ref="druidDataSource"/>
15  </bean>
16 </beans>
```

测试结果：

Tests passed: 1 of 1 test - 1 sec 36 ms

C:\dev\Java\jdk-17.0.4\bin\java.exe ...

```
10月 13, 2022 3:43:04 下午 com.alibaba.druid.support.logging.JakartaCommonsLoggingImpl info
信息: {dataSource-1} init
User{id=5, realName='小明', age=21}
```

动力节点

十四、GoF之代理模式

动力节点

14.1 对代理模式的理解

生活场景1：牛村的牛二看上了隔壁村小花，牛二不好意思直接找小花，于是牛二找来了媒婆王妈妈。这里面就有一个非常典型的代理模式。牛二不能和小花直接对接，只能找一个中间人。其中王妈妈是代理类，牛二是目标类。王妈妈代替牛二和小花先见个面。（现实生活中的婚介所）【在程序中，对象A和对象B无法直接交互时。】

生活场景2：你刚到北京，要租房子，可以自己找，也可以找链家帮你找。其中链家是代理类，你是目标类。你们两个都有共同的行为：找房子。不过链家除了满足你找房子，另外会收取一些费用的。（现实生活中的房产中介）【在程序中，功能需要增强时。】

西游记场景：八戒和高小姐的故事。八戒要强抢民女高翠兰。悟空得知此事之后怎么做的？悟空幻化成高小姐的模样。代替高小姐与八戒会面。其中八戒是客户端程序。悟空是代理类。高小姐是目标类。那天夜里，在八戒眼里，眼前的就是高小姐，对于八戒来说，他是不知道眼前的高小姐是悟空幻化的，在他内心里这就是高小姐。所以悟空代替高小姐和八戒亲了嘴儿。这是非常典型的代理模式实现的保护机制。**代理模式中有一个非常重要的特点：对于客户端程序来说，使用代理对象时就像在使用目标对象一样。**【在程序中，目标需要被保护时】

业务场景：系统中有A、B、C三个模块，使用这些模块的前提是需要用户登录，也就是说在A模块中要编写判断登录的代码，B模块中也要编写，C模块中还要编写，这些判断登录的代码反复出现，显然代码没有得到复用，可以为A、B、C三个模块提供一个代理，在代理当中写一次登录判断即可。代理的逻辑是：请求来了之后，判断用户是否登录了，如果已经登录了，则执行对应的目标，如果没有登录则跳转到登录页面。【在程序中，目标不但受到保护，并且代码也得到了复用。】

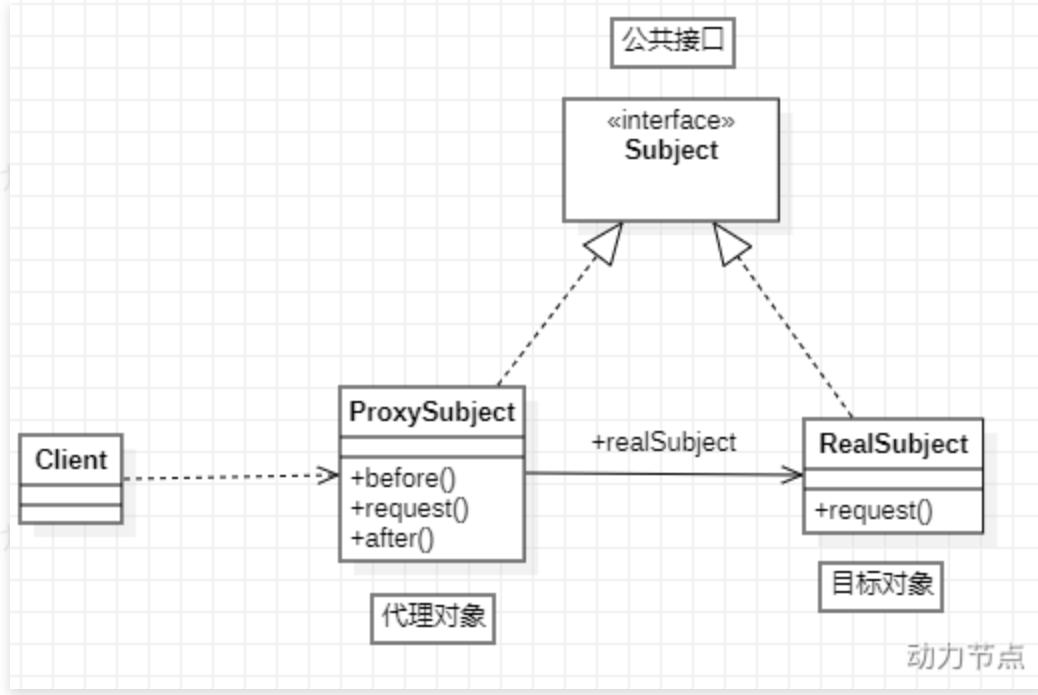
代理模式是GoF23种设计模式之一。属于结构型设计模式。

代理模式的作用是：为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个客户不想或者不能直接引用一个对象，此时可以通过一个称之为“代理”的第三者来实现间接引用。代理对象可以在客户端和目标对象之间起到中介的作用，并且可以通过代理对象去掉客户不应该看到的内容和服务或者添加客户需要的额外服务。通过引入一个新的对象来实现对真实对象的操作或者将新的对象作为真实对象的一个替身，这种实现机制即为代理模式，通过引入代理对象来间接访问一个对象，这就是代理模式的模式动机。

代理模式中的角色：

- 代理类（代理主题）
- 目标类（真实主题）
- 代理类和目标类的公共接口（抽象主题）：客户端在使用代理类时就像在使用目标类，不被客户端所察觉，所以代理类和目标类要有共同的行为，也就是实现共同的接口。

代理模式的类图：



代理模式在代码实现上，包括两种形式：

- 静态代理
- 动态代理

14.2 静态代理

现在有这样一个接口和实现类：

OrderService接口

Java | 复制代码

```
1 package com.powernode.mall.service;
2
3 /**
4  * 订单接口
5  * @author 动力节点
6  * @version 1.0
7  * @className OrderService
8  * @since 1.0
9 */
10 public interface OrderService {
11     /**
12      * 生成订单
13      */
14     void generate();
15
16     /**
17      * 查看订单详情
18      */
19     void detail();
20
21     /**
22      * 修改订单
23      */
24     void modify();
25 }
26
```

OrderService接口的实现类

Java | 复制代码

```
1 package com.powernode.mall.service.impl;
2
3 import com.powernode.mall.service.OrderService;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className OrderServiceImpl
9  * @since 1.0
10 */
11 public class OrderServiceImpl implements OrderService {
12     @Override
13     public void generate() {
14         try {
15             Thread.sleep(1234);
16         } catch (InterruptedException e) {
17             e.printStackTrace();
18         }
19         System.out.println("订单已生成");
20     }
21
22     @Override
23     public void detail() {
24         try {
25             Thread.sleep(2541);
26         } catch (InterruptedException e) {
27             e.printStackTrace();
28         }
29         System.out.println("订单信息如下: *****");
30     }
31
32     @Override
33     public void modify() {
34         try {
35             Thread.sleep(1010);
36         } catch (InterruptedException e) {
37             e.printStackTrace();
38         }
39         System.out.println("订单已修改");
40     }
41 }
42
```

其中Thread.sleep()方法的调用是为了模拟操作耗时。

项目已上线，并且运行正常，只是客户反馈系统有一些地方运行较慢，要求项目组对系统进行优化。于是项目负责人就下达了这个需求。首先需要搞清楚是哪些业务方法耗时较长，于是让我们统计每个业务方法所耗费的时长。如果是你，你该怎么做呢？

第一种方案：直接修改Java源代码，在每个业务方法中添加统计逻辑，如下：

```
1 package com.powernode.mall.service.impl;
2
3 import com.powernode.mall.service.OrderService;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className OrderServiceImpl
9  * @since 1.0
10 */
11 public class OrderServiceImpl implements OrderService {
12     @Override
13     public void generate() {
14         long begin = System.currentTimeMillis();
15         try {
16             Thread.sleep(1234);
17         } catch (InterruptedException e) {
18             e.printStackTrace();
19         }
20         System.out.println("订单已生成");
21         long end = System.currentTimeMillis();
22         System.out.println("耗时" + (end - begin) + "毫秒");
23     }
24
25     @Override
26     public void detail() {
27         long begin = System.currentTimeMillis();
28         try {
29             Thread.sleep(2541);
30         } catch (InterruptedException e) {
31             e.printStackTrace();
32         }
33         System.out.println("订单信息如下: *****");
34         long end = System.currentTimeMillis();
35         System.out.println("耗时" + (end - begin) + "毫秒");
36     }
37
38     @Override
39     public void modify() {
40         long begin = System.currentTimeMillis();
41         try {
42             Thread.sleep(1010);
43         } catch (InterruptedException e) {
44             e.printStackTrace();
45         }
46     }
47 }
```

```
46     System.out.println("订单已修改");
47     long end = System.currentTimeMillis();
48     System.out.println("耗时" +(end - begin) + "毫秒");
49 }
50 }
```

需求可以满足，但显然是违背了OCP开闭原则。这种方案不可取。

第二种方案：编写一个子类继承OrderServiceImpl，在子类中重写每个方法，代码如下：

OrderServiceImpl的子类

Java | 复制代码

```
1 package com.powernode.mall.service.impl;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className OrderServiceImplSub
7  * @since 1.0
8 */
9 public class OrderServiceImplSub extends OrderServiceImpl{
10     @Override
11     public void generate() {
12         long begin = System.currentTimeMillis();
13         super.generate();
14         long end = System.currentTimeMillis();
15         System.out.println("耗时" +(end - begin) + "毫秒");
16     }
17
18     @Override
19     public void detail() {
20         long begin = System.currentTimeMillis();
21         super.detail();
22         long end = System.currentTimeMillis();
23         System.out.println("耗时" +(end - begin) + "毫秒");
24     }
25
26     @Override
27     public void modify() {
28         long begin = System.currentTimeMillis();
29         super.modify();
30         long end = System.currentTimeMillis();
31         System.out.println("耗时" +(end - begin) + "毫秒");
32     }
33 }
```

这种方式可以解决，但是存在两个问题：

- 第一个问题：假设系统中有100个这样的业务类，需要提供100个子类，并且之前写好的创建Service对象的代码，都要修改为创建子类对象。
- 第二个问题：由于采用了继承的方式，导致代码之间的耦合度较高。

这种方案也不可取。

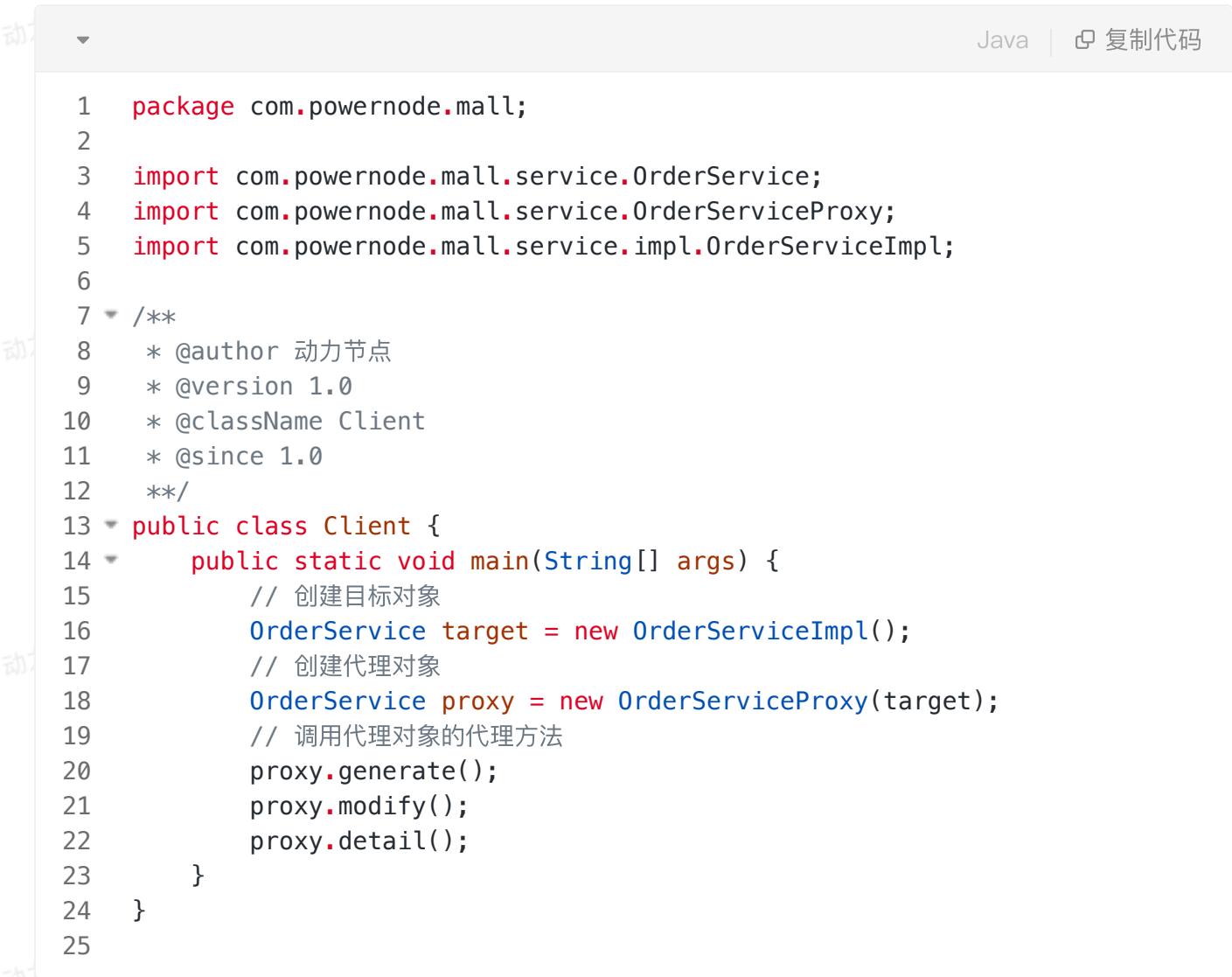
第三种方案：使用代理模式（这里采用静态代理）

可以为OrderService接口提供一个代理类。

```
1 package com.powernode.mall.service;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className OrderServiceProxy
7  * @since 1.0
8  */
9 public class OrderServiceProxy implements OrderService{ // 代理对象
10
11     // 目标对象
12     private OrderService orderService;
13
14     // 通过构造方法将目标对象传递给代理对象
15     public OrderServiceProxy(OrderService orderService) {
16         this.orderService = orderService;
17     }
18
19     @Override
20     public void generate() {
21         long begin = System.currentTimeMillis();
22         // 执行目标对象的目标方法
23         orderService.generate();
24         long end = System.currentTimeMillis();
25         System.out.println("耗时"+(end - begin)+"毫秒");
26     }
27
28     @Override
29     public void detail() {
30         long begin = System.currentTimeMillis();
31         // 执行目标对象的目标方法
32         orderService.detail();
33         long end = System.currentTimeMillis();
34         System.out.println("耗时"+(end - begin)+"毫秒");
35     }
36
37     @Override
38     public void modify() {
39         long begin = System.currentTimeMillis();
40         // 执行目标对象的目标方法
41         orderService.modify();
42         long end = System.currentTimeMillis();
43         System.out.println("耗时"+(end - begin)+"毫秒");
44     }
45 }
```

这种方式的优点：符合OCP开闭原则，同时采用的是关联关系，所以程序的耦合度较低。所以这种方案是被推荐的。

编写客户端程序：



The screenshot shows a Java code editor with the following code:

```
1 package com.powernode.mall;
2
3 import com.powernode.mall.service.OrderService;
4 import com.powernode.mall.service.OrderServiceProxy;
5 import com.powernode.mall.service.impl.OrderServiceImpl;
6
7 /**
8 * @author 动力节点
9 * @version 1.0
10 * @className Client
11 * @since 1.0
12 */
13 public class Client {
14     public static void main(String[] args) {
15         // 创建目标对象
16         OrderService target = new OrderServiceImpl();
17         // 创建代理对象
18         OrderService proxy = new OrderServiceProxy(target);
19         // 调用代理对象的代理方法
20         proxy.generate();
21         proxy.modify();
22         proxy.detail();
23     }
24 }
```

The code implements the Proxy pattern. It defines a `Client` class with a `main` method. Inside the `main` method, it creates a target object of type `OrderService` (which is `OrderServiceImpl` in this case) and a proxy object of type `OrderServiceProxy`. The proxy object wraps the target object, allowing clients to interact with the target object through the proxy.

运行结果：

```
C:\dev\Java\jdk-17.0.4\bin\java
订单已生成
耗时1235毫秒
订单已修改
耗时1025毫秒
订单信息如下: *****
耗时2543毫秒
```

以上就是代理模式中的静态代理，其中OrderService接口是代理类和目标类的共同接口。

OrderServiceImpl是目标类。OrderServiceProxy是代理类。

大家思考一下：如果系统中业务接口很多，一个接口对应一个代理类，显然也是不合理的，会导致类爆炸。怎么解决这个问题？动态代理可以解决。因为在动态代理中可以在内存中动态的为我们生成代理类的字节码。代理类不需要我们写了。类爆炸解决了，而且代码只需要写一次，代码也会得到复用。

14.3 动态代理

在程序运行阶段，在内存中动态生成代理类，被称为动态代理，目的是为了减少代理类的数量。解决代码复用的问题。

在内存当中动态生成类的技术常见的包括：

- JDK动态代理技术：只能代理接口。
- CGLIB动态代理技术：CGLIB(Code Generation Library)是一个开源项目。是一个强大的，高性能，高质量的Code生成类库，它可以在运行期扩展Java类与实现Java接口。它既可以代理接口，又可以代理类，**底层是通过继承的方式实现的**。性能比JDK动态代理要好。**(底层有一个小而快的字节码处理框架ASM。)**
- Javassist动态代理技术：Javassist是一个开源的分析、编辑和创建Java字节码的类库。是由东京工业大学的数学和计算机科学系的 Shigeru Chiba (千叶 滋) 所创建的。它已加入了开放源代码JBoss 应用服务器项目，通过使用Javassist对字节码操作为JBoss实现动态"AOP"框架。

14.3.1 JDK动态代理

我们还是使用静态代理中的例子：一个接口和一个实现类。

OrderService接口

Java | 复制代码

```
1 package com.powernode.mall.service;
2
3 /**
4  * 订单接口
5  * @author 动力节点
6  * @version 1.0
7  * @className OrderService
8  * @since 1.0
9 */
10 public interface OrderService {
11     /**
12      * 生成订单
13      */
14     void generate();
15
16     /**
17      * 查看订单详情
18      */
19     void detail();
20
21     /**
22      * 修改订单
23      */
24     void modify();
25 }
26
```

OrderService接口实现类

Java | 复制代码

```
1 package com.powernode.mall.service.impl;
2
3 import com.powernode.mall.service.OrderService;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className OrderServiceImpl
9  * @since 1.0
10 */
11 public class OrderServiceImpl implements OrderService {
12     @Override
13     public void generate() {
14         try {
15             Thread.sleep(1234);
16         } catch (InterruptedException e) {
17             e.printStackTrace();
18         }
19         System.out.println("订单已生成");
20     }
21
22     @Override
23     public void detail() {
24         try {
25             Thread.sleep(2541);
26         } catch (InterruptedException e) {
27             e.printStackTrace();
28         }
29         System.out.println("订单信息如下: *****");
30     }
31
32     @Override
33     public void modify() {
34         try {
35             Thread.sleep(1010);
36         } catch (InterruptedException e) {
37             e.printStackTrace();
38         }
39         System.out.println("订单已修改");
40     }
41 }
42
```

我们在静态代理的时候，除了以上一个接口和一个实现类之外，是不是要写一个代理类 UserServiceProxy呀！在动态代理中UserServiceProxy代理类是可以动态生成的。这个类不需要写。我们直接写客户端程序即可：

```
客户端程序 Java | 复制代码

1 package com.powernode.mall;
2
3 import com.powernode.mall.service.OrderService;
4 import com.powernode.mall.service.impl.OrderServiceImpl;
5
6 import java.lang.reflect.Proxy;
7
8 /**
9  * @author 动力节点
10 * @version 1.0
11 * @className Client
12 * @since 1.0
13 */
14 public class Client {
15     public static void main(String[] args) {
16         // 第一步：创建目标对象
17         OrderService target = new OrderServiceImpl();
18         // 第二步：创建代理对象
19         OrderService orderServiceProxy = Proxy.newProxyInstance(target.getClass().getClassLoader(), target.getClass().getInterfaces(), 调用处理器对象);
20         // 第三步：调用代理对象的代理方法
21         orderServiceProxy.detail();
22         orderServiceProxy.modify();
23         orderServiceProxy.generate();
24     }
25 }
```

以上第二步创建代理对象是需要大家理解的：

```
OrderService orderServiceProxy = Proxy.newProxyInstance(target.getClass().getClassLoader(),
target.getClass().getInterfaces(), 调用处理器对象);
```

这行代码做了两件事：

- 第一件事：在内存中生成了代理类的字节码
- 第二件事：创建代理对象

Proxy类全名：java.lang.reflect.Proxy。这是JDK提供的一个类（所以称为JDK动态代理）。主要是通过这个类在内存中生成代理类的字节码。

其中newProxyInstance()方法有三个参数：

- 第一个参数：类加载器。在内存中生成了字节码，要想执行这个字节码，也是需要把这个字节码加载到内存当中的。所以要指定使用哪个类加载器加载。
- 第二个参数：接口类型。代理类和目标类实现相同的接口，所以要通过这个参数告诉JDK动态代理生成的类要实现哪些接口。
- 第三个参数：调用处理器。这是一个JDK动态代理规定的接口，接口全名：
java.lang.reflect.InvocationHandler。显然这是一个回调接口，也就是说调用这个接口中方法的程序已经写好了，就差这个接口的实现类了。

所以接下来我们要写一下java.lang.reflect.InvocationHandler接口的实现类，并且实现接口中的方法，代码如下：

```
▼ TimerInvocationHandler Java | 复制代码
1 package com.powernode.mall.service;
2
3 import java.lang.reflect.InvocationHandler;
4 import java.lang.reflect.Method;
5
6 /**
7  * @author 动力节点
8  * @version 1.0
9  * @className TimerInvocationHandler
10 * @since 1.0
11 */
12 public class TimerInvocationHandler implements InvocationHandler {
13     @Override
14     public Object invoke(Object proxy, Method method, Object[] args) throws
15             Throwable {
16         return null;
17     }
}
```

InvocationHandler接口中有一个方法invoke，这个invoke方法上有三个参数：

- 第一个参数：Object proxy。代理对象。设计这个参数只是为了后期的方便，如果想在invoke方法中使用代理对象的话，尽管通过这个参数来使用。
- 第二个参数：Method method。目标方法。
- 第三个参数：Object[] args。目标方法调用时要传的参数。

我们将来肯定是要调用“目标方法”的，但要调用目标方法的话，需要“目标对象”的存在，“目标对象”从哪儿来呢？我们可以给TimerInvocationHandler提供一个构造方法，可以通过这个构造方法传过来“目标对象”，代码如下：

TimerInvocationHandler

Java | 复制代码

```
1 package com.powernode.mall.service;
2
3 import java.lang.reflect.InvocationHandler;
4 import java.lang.reflect.Method;
5
6 /**
7  * @author 动力节点
8  * @version 1.0
9  * @className TimerInvocationHandler
10 * @since 1.0
11 */
12 public class TimerInvocationHandler implements InvocationHandler {
13     // 目标对象
14     private Object target;
15
16     // 通过构造方法来传目标对象
17     public TimerInvocationHandler(Object target) {
18         this.target = target;
19     }
20
21     @Override
22     public Object invoke(Object proxy, Method method, Object[] args) throws
23     Throwable {
24         return null;
25     }
26 }
```

有了目标对象我们就可以在invoke()方法中调用目标方法了。代码如下：

TimerInvocationHandler

Java | 复制代码

```
1 package com.powernode.mall.service;
2
3 import java.lang.reflect.InvocationHandler;
4 import java.lang.reflect.Method;
5
6 /**
7  * @author 动力节点
8  * @version 1.0
9  * @className TimerInvocationHandler
10 * @since 1.0
11 */
12 public class TimerInvocationHandler implements InvocationHandler {
13     // 目标对象
14     private Object target;
15
16     // 通过构造方法来传目标对象
17     public TimerInvocationHandler(Object target) {
18         this.target = target;
19     }
20
21     @Override
22     public Object invoke(Object proxy, Method method, Object[] args) throws
23     Throwable {
24         // 目标执行之前增强。
25         long begin = System.currentTimeMillis();
26         // 调用目标对象的目标方法
27         Object retValue = method.invoke(target, args);
28         // 目标执行之后增强。
29         long end = System.currentTimeMillis();
30         System.out.println("耗时"+(end - begin)+"毫秒");
31         // 一定要记得返回哦。
32         return retValue;
33     }
34 }
```

到此为止，调用处理器就完成了。接下来，应该继续完善Client程序：

客户端程序

Java | 复制代码

```
1 package com.powernode.mall;
2
3 import com.powernode.mall.service.OrderService;
4 import com.powernode.mall.service.TimerInvocationHandler;
5 import com.powernode.mall.service.impl.OrderServiceImpl;
6
7 import java.lang.reflect.Proxy;
8
9 /**
10 * @author 动力节点
11 * @version 1.0
12 * @className Client
13 * @since 1.0
14 */
15 public class Client {
16     public static void main(String[] args) {
17         // 创建目标对象
18         OrderService target = new OrderServiceImpl();
19         // 创建代理对象
20         OrderService orderServiceProxy = (OrderService) Proxy.newProxyInstance(target.getClass().getClassLoader(),
21             target.getClass().getInterfaces(),
22             new TimerInvocationHandler(target));
23         // 调用代理对象的代理方法
24         orderServiceProxy.detail();
25         orderServiceProxy.modify();
26         orderServiceProxy.generate();
27     }
28 }
29
```

大家可能会比较好奇：那个InvocationHandler接口中的invoke()方法没看见在哪里调用呀？

注意：当你调用代理对象的代理方法的时候，注册在InvocationHandler接口中的invoke()方法会被调用。也就是上面代码第24 25 26行，这三行代码中任意一行代码执行，注册在InvocationHandler接口中的invoke()方法都会被调用。

执行结果：

```
com.powernode.mall.Client >
C:\dev\Java\jdk-17.0.4\bin\java
订单信息如下: *****
耗时2550毫秒
订单已修改
耗时1016毫秒
订单已生成
耗时1235毫秒
```

学到这里可能会感觉有点懵，折腾半天，到最后这不是还得写一个接口的实现类吗？没省劲儿呀？

你要这样想就错了!!!!

我们可以看到，不管你有多少个Service接口，多少个业务类，这个TimerInvocationHandler接口是不是只需要写一次就行了，代码是不是得到复用了！！！！

而且最重要的是，以后程序员只需要关注核心业务的编写了，像这种统计时间的代码根本不需要关注。因为这种统计时间的代码只需要在调用处理器中编写一次即可。

到这里，JDK动态代理的原理就结束了。

不过我们看以下这个代码确实有点繁琐，对于客户端来说，用起来不方便：

```
public class Client {
    public static void main(String[] args) {
        // 创建目标对象
        OrderService target = new OrderServiceImpl();
        // 创建代理对象
        OrderService orderServiceProxy = (OrderService) Proxy.newProxyInstance(target.getClass().getClassLoader(),
                target.getClass().getInterfaces(),
                new TimerInvocationHandler(target));
        // 调用代理对象的代理方法
        orderServiceProxy.detail();
```

我们可以提供一个工具类：ProxyUtil，封装一个方法：

▼ 工具类

Java | 复制代码

```
1 package com.powernode.mall.util;
2
3 import com.powernode.mall.service.TimerInvocationHandler;
4
5 import java.lang.reflect.Proxy;
6
7 /**
8 * @author 动力节点
9 * @version 1.0
10 * @className ProxyUtil
11 * @since 1.0
12 */
13 public class ProxyUtil {
14     public static Object newProxyInstance(Object target) {
15         return Proxy.newProxyInstance(target.getClass().getClassLoader(),
16             target.getClass().getInterfaces(),
17             new TimerInvocationHandler(target));
18     }
19 }
```

这样客户端代码就不需要写那么繁琐了：

客户端程序

Java | 复制代码

```
1 package com.powernode.mall;
2
3 import com.powernode.mall.service.OrderService;
4 import com.powernode.mall.service.TimerInvocationHandler;
5 import com.powernode.mall.service.impl.OrderServiceImpl;
6 import com.powernode.mall.util.ProxyUtil;
7
8 import java.lang.reflect.Proxy;
9
10 /**
11 * @author 动力节点
12 * @version 1.0
13 * @className Client
14 * @since 1.0
15 */
16 public class Client {
17     public static void main(String[] args) {
18         // 创建目标对象
19         OrderService target = new OrderServiceImpl();
20         // 创建代理对象
21         OrderService orderServiceProxy = (OrderService) ProxyUtil.newProxy
22             Instance(target);
23         // 调用代理对象的代理方法
24         orderServiceProxy.detail();
25         orderServiceProxy.modify();
26         orderServiceProxy.generate();
27     }
28 }
```

执行结果：

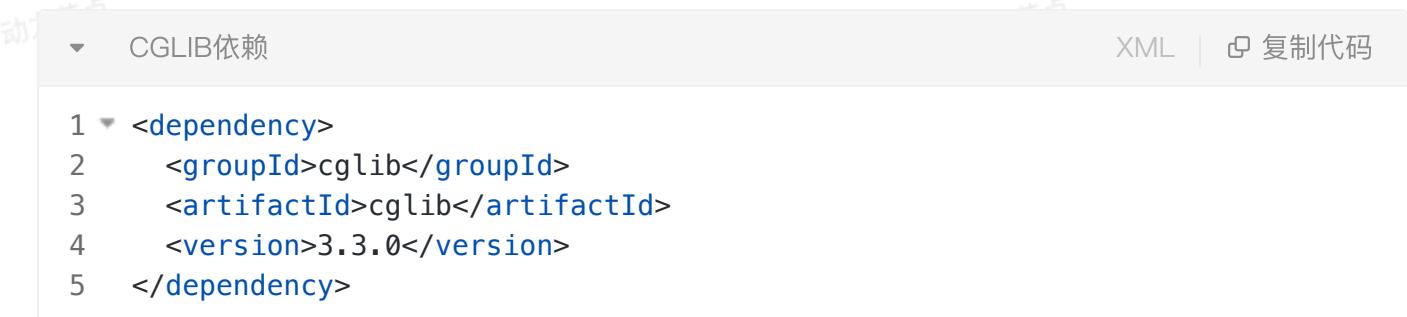
The screenshot shows a terminal window with the title bar "com.powernode.mall.Client". The output of the program is displayed in the terminal:

```
C:\dev\Java\jdk-17.0.4\b
订单信息如下: *****
耗时2542毫秒
订单已修改
耗时1015毫秒
订单已生成
耗时1235毫秒
```

14.3.2 CGLIB动态代理

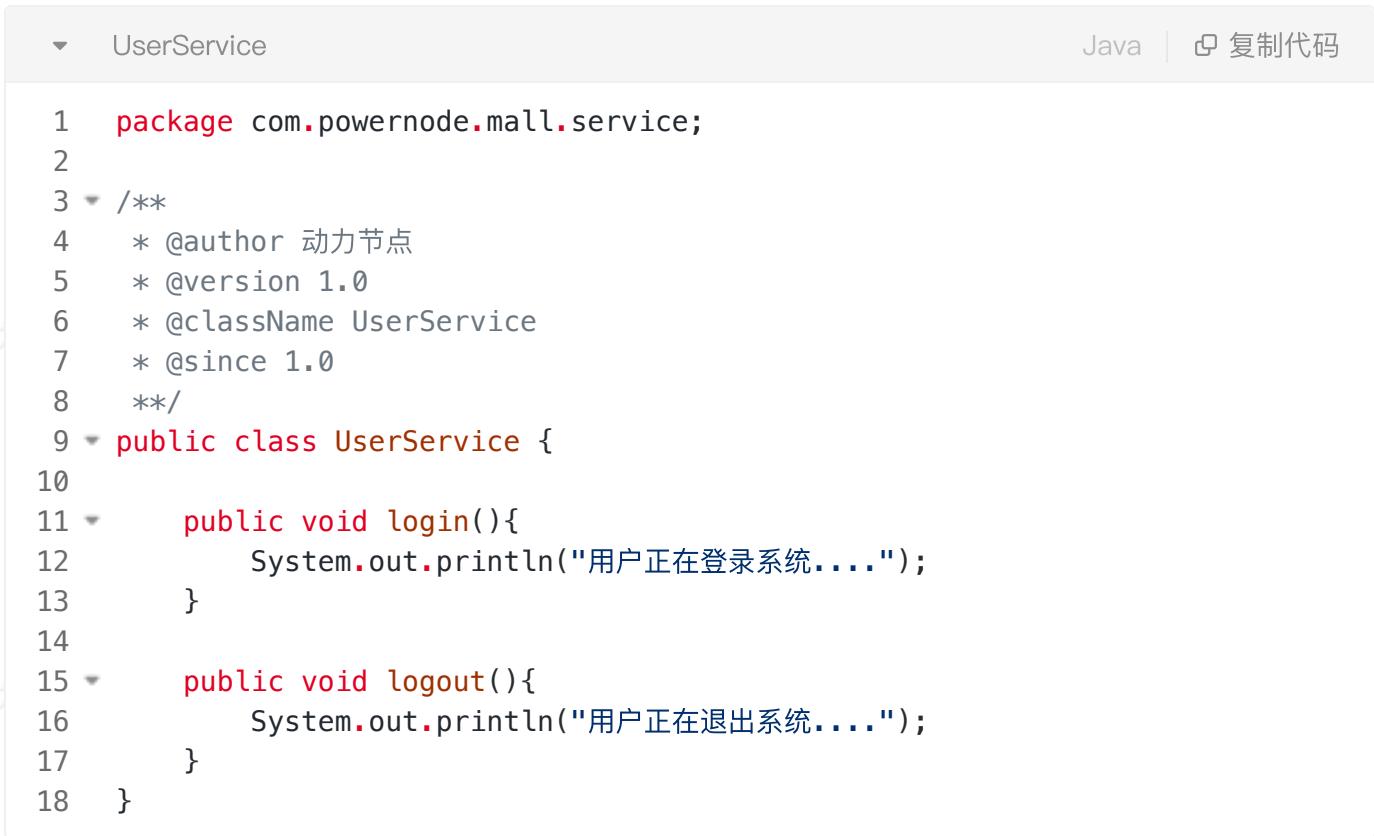
CGLIB既可以代理接口，又可以代理类。底层采用继承的方式实现。所以被代理的目标类不能使用final修饰。

使用CGLIB，需要引入它的依赖：



```
1 <dependency>
2   <groupId>cglib</groupId>
3   <artifactId>cglib</artifactId>
4   <version>3.3.0</version>
5 </dependency>
```

我们准备一个没有实现接口的类，如下：



```
1 package com.powernode.mall.service;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className UserService
7  * @since 1.0
8 */
9 public class UserService {
10
11     public void login(){
12         System.out.println("用户正在登录系统....");
13     }
14
15     public void logout(){
16         System.out.println("用户正在退出系统....");
17     }
18 }
```

使用CGLIB在内存中为UserService类生成代理类，并创建对象：

客户端程序

Java | 复制代码

```
1 package com.powernode.mall;
2
3 import com.powernode.mall.service.UserService;
4 import net.sf.cglib.proxy.Enhancer;
5
6 /**
7  * @author 动力节点
8  * @version 1.0
9  * @className Client
10 * @since 1.0
11 */
12 public class Client {
13     public static void main(String[] args) {
14         // 创建字节码增强器
15         Enhancer enhancer = new Enhancer();
16         // 告诉cglib要继承哪个类
17         enhancer.setSuperclass(UserService.class);
18         // 设置回调接口
19         enhancer.setCallback(方法拦截器对象);
20         // 生成源码，编译class，加载到JVM，并创建代理对象
21         UserService userServiceProxy = (UserService)enhancer.create();
22
23         userServiceProxy.login();
24         userServiceProxy.logout();
25
26     }
27 }
```

和JDK动态代理原理差不多，在CGLIB中需要提供的不是InvocationHandler，而是：

net.sf.cglib.proxy.MethodInterceptor

编写MethodInterceptor接口实现类：

TimerMethodInterceptor

Java | 复制代码

```
1 package com.powernode.mall.service;
2
3 import net.sf.cglib.proxy.MethodInterceptor;
4 import net.sf.cglib.proxy.MethodProxy;
5
6 import java.lang.reflect.Method;
7
8 /**
9  * @author 动力节点
10 * @version 1.0
11 * @className TimerMethodInterceptor
12 * @since 1.0
13 */
14 public class TimerMethodInterceptor implements MethodInterceptor {
15     @Override
16     public Object intercept(Object target, Method method, Object[] objects
17 , MethodProxy methodProxy) throws Throwable {
18         return null;
19     }
}
```

MethodInterceptor接口中有一个方法intercept(), 该方法有4个参数:

第一个参数: 目标对象

第二个参数: 目标方法

第三个参数: 目标方法调用时的实参

第四个参数: 代理方法

在MethodInterceptor的intercept()方法中调用目标以及添加增强:

TimerMethodInterceptor

Java | 复制代码

```
1 package com.powernode.mall.service;
2
3 import net.sf.cglib.proxy.MethodInterceptor;
4 import net.sf.cglib.proxy.MethodProxy;
5
6 import java.lang.reflect.Method;
7
8 /**
9  * @author 动力节点
10 * @version 1.0
11 * @className TimerMethodInterceptor
12 * @since 1.0
13 */
14 public class TimerMethodInterceptor implements MethodInterceptor {
15     @Override
16     public Object intercept(Object target, Method method, Object[] objects
17 , MethodProxy methodProxy) throws Throwable {
18         // 前增强
19         long begin = System.currentTimeMillis();
20         // 调用目标
21         Object retValue = methodProxy.invokeSuper(target, objects);
22         // 后增强
23         long end = System.currentTimeMillis();
24         System.out.println("耗时" + (end - begin) + "毫秒");
25         // 一定要返回
26         return retValue;
27     }
28 }
```

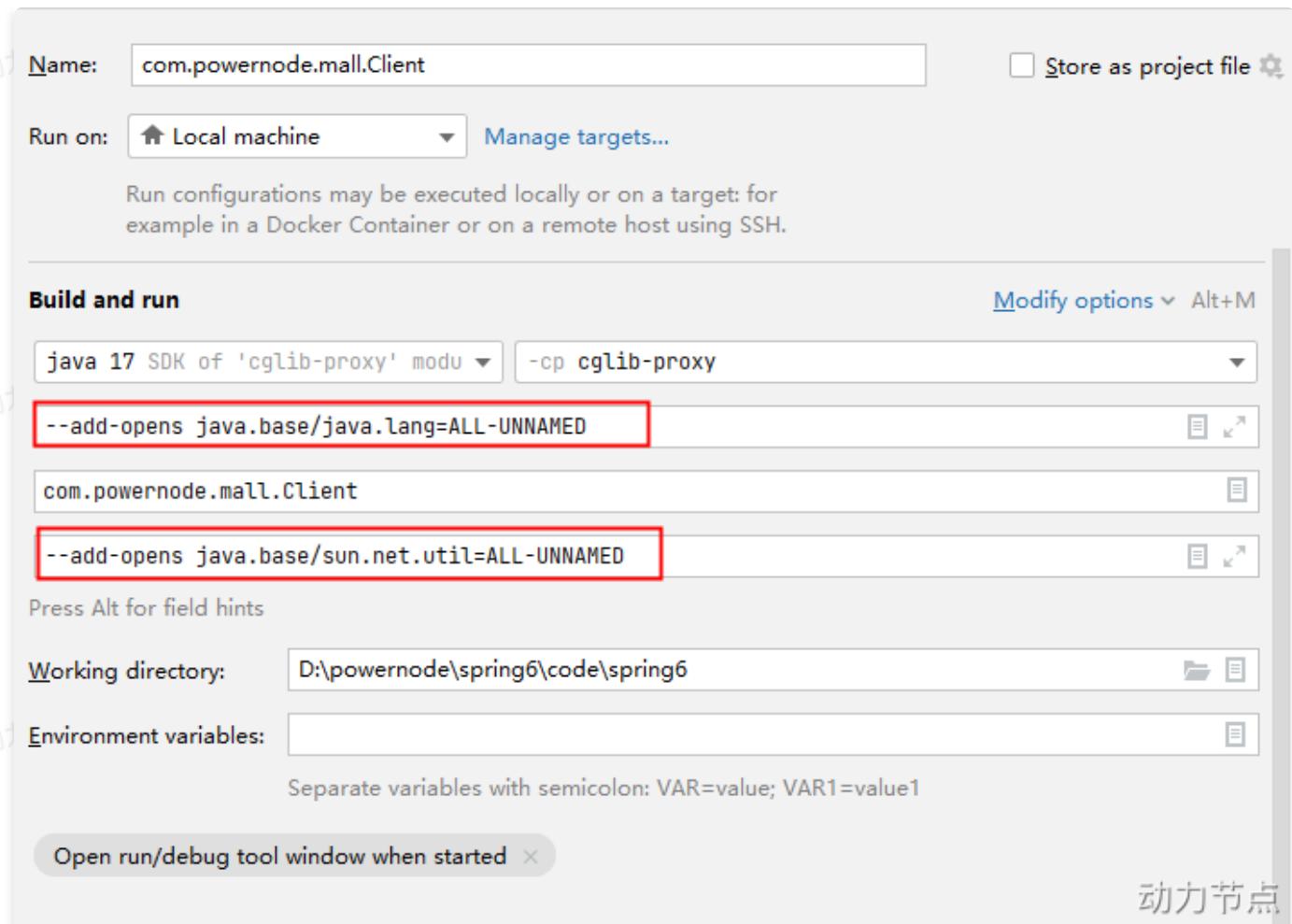
回调已经写完了，可以修改客户端程序了：

客户端程序

Java | 复制代码

```
1 package com.powernode.mall;
2
3 import com.powernode.mall.service.TimerMethodInterceptor;
4 import com.powernode.mall.service.UserService;
5 import net.sf.cglib.proxy.Enhancer;
6
7 /**
8 * @author 动力节点
9 * @version 1.0
10 * @className Client
11 * @since 1.0
12 */
13 public class Client {
14     public static void main(String[] args) {
15         // 创建字节码增强器
16         Enhancer enhancer = new Enhancer();
17         // 告诉cglib要继承哪个类
18         enhancer.setSuperclass(UserService.class);
19         // 设置回调接口
20         enhancer.setCallback(new TimerMethodInterceptor());
21         // 生成源码，编译class，加载到JVM，并创建代理对象
22         UserService userServiceProxy = (UserService)enhancer.create();
23
24         userServiceProxy.login();
25         userServiceProxy.logout();
26
27     }
28 }
```

对于高版本的JDK，如果使用CGLIB，需要在启动项中添加两个启动参数：



- --add-opens java.base/java.lang=ALL-UNNAMED
- --add-opens java.base/sun.net.util=ALL-UNNAMED

执行结果：

```
com.powernode.mall.Client
C:\dev\Java\jdk-17.0.4\bin'
用户正在登录系统....
耗时13毫秒
用户正在退出系统....
耗时0毫秒
```

十五、面向切面编程AOP

IoC使软件组件松耦合。AOP让你能够捕捉系统中经常使用的功能，把它转化成组件。

AOP (Aspect Oriented Programming) : 面向切面编程，面向方面编程。（AOP是一种编程技术）

AOP是对OOP的补充延伸。

AOP底层使用的就是动态代理来实现的。

Spring的AOP使用的动态代理是：JDK动态代理 + CGLIB动态代理技术。Spring在这两种动态代理中灵活切换，如果是代理接口，会默认使用JDK动态代理，如果要代理某个类，这个类没有实现接口，就会切换使用CGLIB。当然，你也可以强制通过一些配置让Spring只使用CGLIB。

15.1 AOP介绍

一般一个系统当中都会有一些系统服务，例如：日志、事务管理、安全等。这些系统服务被称为：**交叉业务**

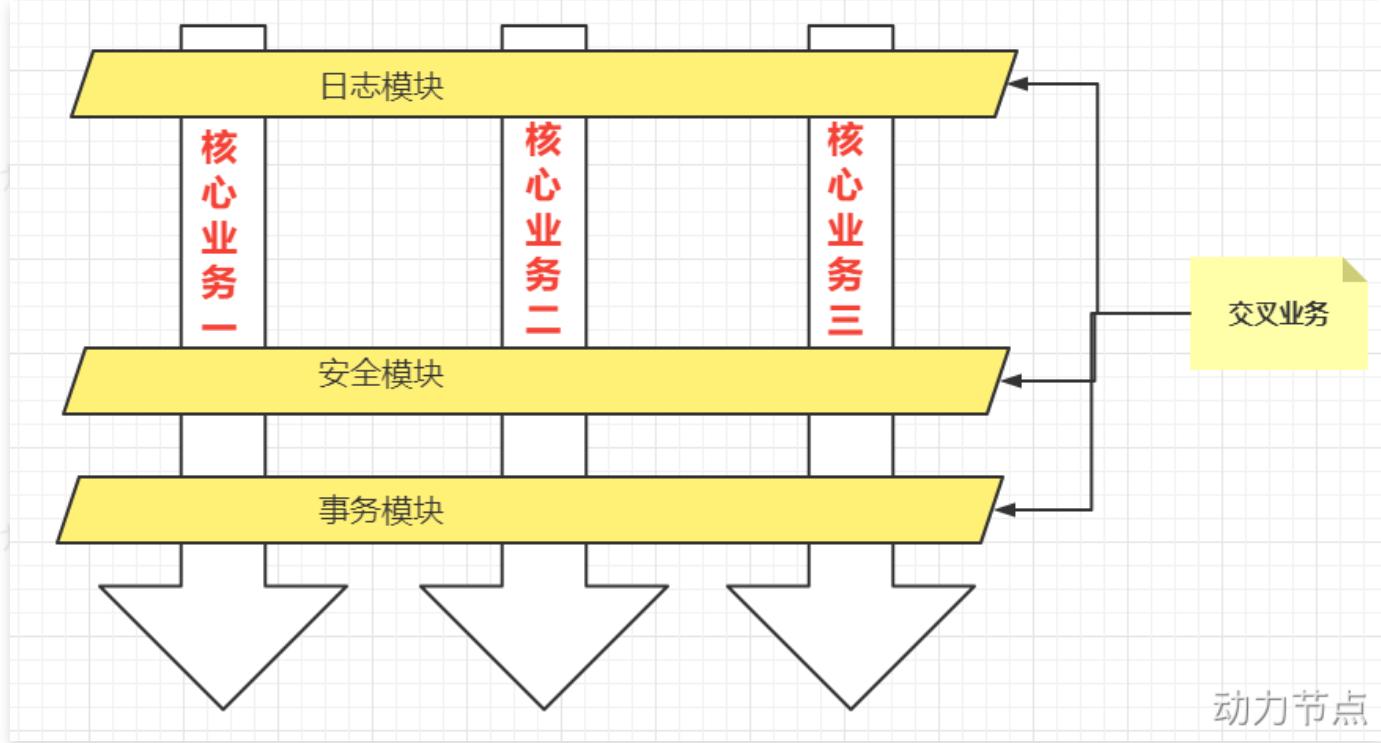
这些**交叉业务**几乎是通用的，不管你是做银行账户转账，还是删除用户数据。日志、事务管理、安全，这些都是需要做的。

如果在每一个业务处理过程当中，都掺杂这些交叉业务代码进去的话，存在两方面问题：

- 第一：交叉业务代码在多个业务流程中反复出现，显然这个交叉业务代码没有得到复用。并且修改这些交叉业务代码的话，需要修改多处。
- 第二：程序员无法专注核心业务代码的编写，在编写核心业务代码的同时还需要处理这些交叉业务。

使用AOP可以很轻松的解决以上问题。

请看下图，可以帮助你快速理解AOP的思想：



用一句话总结AOP：将与核心业务无关的代码独立的抽取出来，形成一个独立的组件，然后以横向交叉的方式应用到业务流程当中的过程被称为AOP。

AOP的优点：

- 第一：代码复用性增强。
- 第二：代码易维护。
- 第三：使开发者更关注业务逻辑。

15.2 AOP的七大术语

```

1  public class UserService{
2      public void do1(){
3          System.out.println("do 1");
4      }
5      public void do2(){
6          System.out.println("do 2");
7      }
8      public void do3(){
9          System.out.println("do 3");
10     }
11     public void do4(){
12         System.out.println("do 4");
13     }
14     public void do5(){
15         System.out.println("do 5");
16     }
17     // 核心业务方法
18     public void service(){
19         do1();
20         do2();
21         do3();
22         do5();
23     }
24 }
```

• 连接点 Joinpoint

- 在程序的整个执行流程中，**可以织入**切面的位置。方法的执行前后，异常抛出之后等位置。

• 切点 Pointcut

- 在程序执行流程中，**真正织入**切面的方法。（一个切点对应多个连接点）

• 通知 Advice

- 通知又叫增强，就是具体你要织入的代码。

- 通知包括：

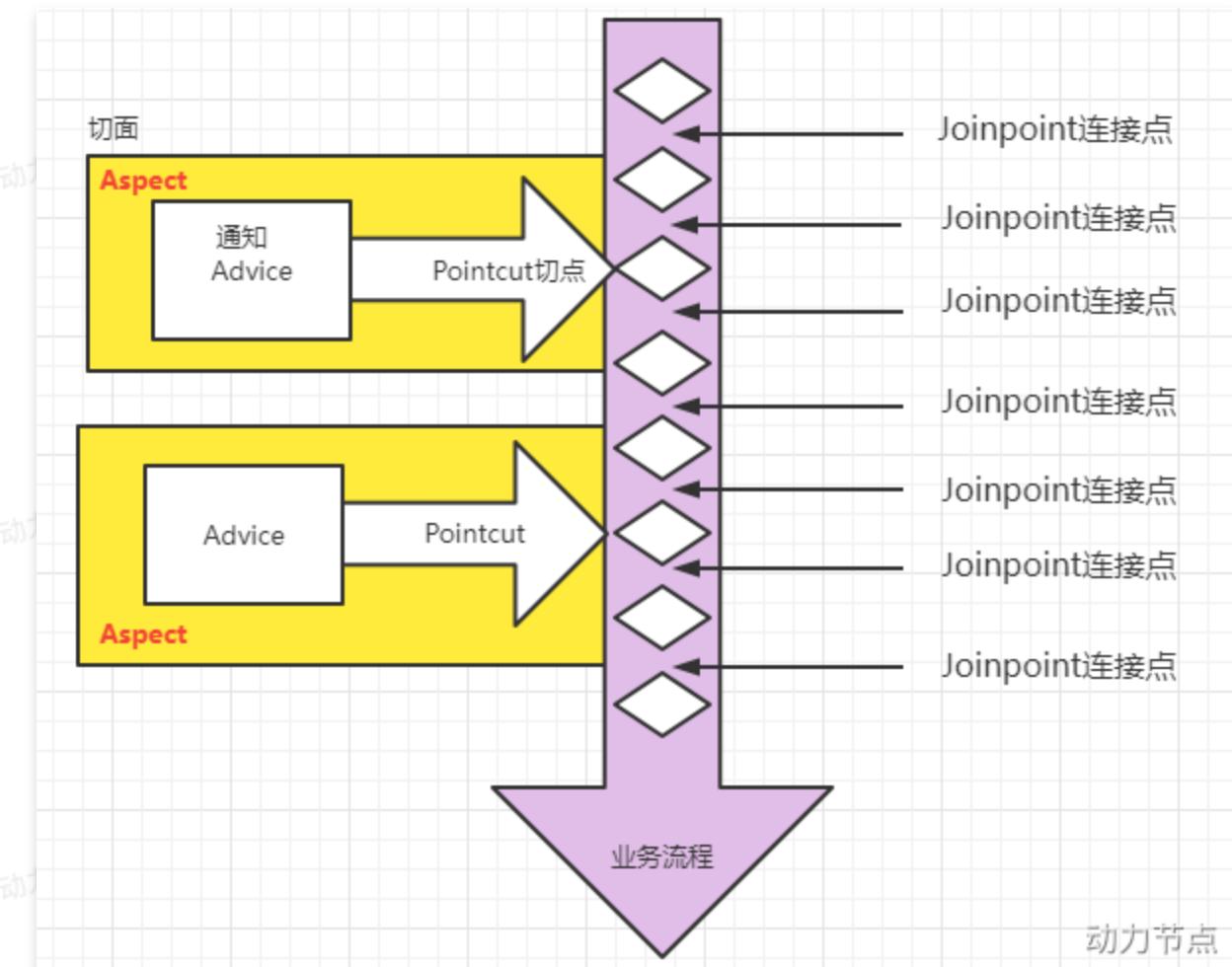
- 前置通知
- 后置通知
- 环绕通知
- 异常通知
- 最终通知

• 切面 Aspect

- 切点 + 通知就是切面。

- 织入 Weaving
 - 把通知应用到目标对象上的过程。
- 代理对象 Proxy
 - 一个目标对象被织入通知后产生的新对象。
- 目标对象 Target
 - 被织入通知的对象。

通过下图，大家可以很好的理解AOP的相关术语：



15.3 切点表达式

切点表达式用来定义通知（Advice）往哪些方法上切入。

切入点表达式语法格式：

```
1 execution([访问控制权限修饰符] 返回值类型 [全限定类名]方法名(形式参数列表) [异常])
```

访问控制权限修饰符：

- 可选项。
- 没写，就是4个权限都包括。
- 写public就表示只包括公开的方法。

返回值类型：

- 必填项。
- * 表示返回值类型任意。

全限定类名：

- 可选项。
- 两个点“..”代表当前包以及子包下的所有类。
- 省略时表示所有的类。

方法名：

- 必填项。
- *表示所有方法。
- set*表示所有的set方法。

形式参数列表：

- 必填项
- () 表示没有参数的方法
- (...) 参数类型和个数随意的方法
- (*) 只有一个参数的方法
- (*, String) 第一个参数类型随意，第二个参数是String的。

异常：

- 可选项。
- 省略时表示任意异常类型。

理解以下的切点表达式：

```
▼ service包下所有的类中以delete开始的所有方法 Java | 复制代码
1 execution(public * com.powernode.mall.service.*.delete*(...))

▼ mall包下所有的类的所有的方法 Java | 复制代码
1 execution(* com.powernode.mall...*(...))

▼ 所有类的所有方法 Java | 复制代码
1 execution(* *(...))
```

15.4 使用Spring的AOP

Spring对AOP的实现包括以下3种方式：

- 第一种方式：Spring框架结合AspectJ框架实现的AOP，基于注解方式。
- 第二种方式：Spring框架结合AspectJ框架实现的AOP，基于XML方式。
- 第三种方式：Spring框架自己实现的AOP，基于XML配置方式。

实际开发中，都是Spring+AspectJ来实现AOP。所以我们重点学习第一种和第二种方式。

什么是AspectJ？（Eclipse组织的一个支持AOP的框架。AspectJ框架是独立于Spring框架之外的一个框架，Spring框架用了AspectJ）

AspectJ项目起源于帕洛阿尔托（Palo Alto）研究中心（缩写为PARC）。该中心由Xerox集团资助，Gregor Kiczales领导，从1997年开始致力于AspectJ的开发，1998年第一次发布给外部用户，2001年发布1.0 release。为了推动AspectJ技术和社团的发展，PARC在2003年3月正式将AspectJ项目移交给了Eclipse组织，因为AspectJ的发展和受关注程度大大超出了PARC的预期，他们已经无力继续维持它的发展。

15.4.1 准备工作

使用Spring+AspectJ的AOP需要引入的依赖如下：

pom.xml

XML | 复制代码

```
1 <!--spring context依赖-->
2 <dependency>
3   <groupId>org.springframework</groupId>
4   <artifactId>spring-context</artifactId>
5   <version>6.0.0-M2</version>
6 </dependency>
7 <!--spring aop依赖-->
8 <dependency>
9   <groupId>org.springframework</groupId>
10  <artifactId>spring-aop</artifactId>
11  <version>6.0.0-M2</version>
12 </dependency>
13 <!--spring aspects依赖-->
14 <dependency>
15   <groupId>org.springframework</groupId>
16   <artifactId>spring-aspects</artifactId>
17   <version>6.0.0-M2</version>
18 </dependency>
```

Spring配置文件中添加context命名空间和aop命名空间

spring-aspectj-aop-annotation.xml

XML | 复制代码

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xmlns:aop="http://www.springframework.org/schema/aop"
6   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
7           http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd
8           http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">
9
10 </beans>
```

15.4.2 基于AspectJ的AOP注解式开发

实现步骤

第一步：定义目标类以及目标方法

▼ 目标类

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 // 目标类
4 public class OrderService {
5     // 目标方法
6     public void generate(){
7         System.out.println("订单已生成！");
8     }
9 }
```

第二步：定义切面类

▼ 切面类

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import org.aspectj.lang.annotation.Aspect;
4
5 // 切面类
6 @Aspect
7 public class MyAspect {
8 }
```

第三步：目标类和切面类都纳入spring bean管理

在目标类OrderService上添加**@Component**注解。

在切面类MyAspect类上添加**@Component**注解。

第四步：在spring配置文件中添加组建扫描

spring-aspectj-aop-annotation.xml

XML | 复制代码

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xmlns:context="http://www.springframework.org/schema/context"
5         xmlns:aop="http://www.springframework.org/schema/aop"
6         xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
7                         http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd
8                         http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">
9     <!--开启组件扫描-->
10    <context:component-scan base-package="com.powernode.spring6.service"/>
11 </beans>
```

第五步：在切面类中添加通知

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import org.springframework.stereotype.Component;
4 import org.aspectj.lang.annotation.Aspect;
5
6 // 切面类
7 @Aspect
8 @Component
9 public class MyAspect {
10     // 这就是需要增强的代码（通知）
11     public void advice(){
12         System.out.println("我是一个通知");
13     }
14 }
15
```

第六步：在通知上添加切点表达式

通知+切点=切面

Java | 复制代码

```

1 package com.powernode.spring6.service;
2
3 import org.aspectj.lang.annotation.Before;
4 import org.springframework.stereotype.Component;
5 import org.aspectj.lang.annotation.Aspect;
6
7 // 切面类
8 @Aspect
9 @Component
10 public class MyAspect {
11
12     // 切点表达式
13     @Before("execution(* com.powernode.spring6.service.OrderService.*(..))")
14     // 这就是需要增强的代码（通知）
15     public void advice(){
16         System.out.println("我是一个通知");
17     }
18 }
```

注解@Before表示前置通知。

第七步：在spring配置文件中启用自动代理

spring-aspectj-aop-annotation.xml

XML | 复制代码

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xmlns:context="http://www.springframework.org/schema/context"
5         xmlns:aop="http://www.springframework.org/schema/aop"
6         xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
7                         http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd
8                         http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">
9     <!--开启组件扫描-->
10    <context:component-scan base-package="com.powernode.spring6.service"/>
11    <!--开启自动代理-->
12    <aop:aspectj-autoproxy proxy-target-class="true"/>
13 </beans>
```

<aop:aspectj-autoproxy proxy-target-class="true"/> 开启自动代理之后，凡事带有@Aspect注解的bean都会生成代理对象。

proxy-target-class="true" 表示采用cglib动态代理。

proxy-target-class="false" 表示采用jdk动态代理。默认值是false。即使写成false，当没有接口的时候，也会自动选择cglib生成代理类。

测试程序：

```
1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.service.OrderService;
4 import org.junit.Test;
5 import org.springframework.context.ApplicationContext;
6 import org.springframework.context.support.ClassPathXmlApplicationContext;
7
8 public class AOPTest {
9     @Test
10    public void testAOP(){
11        ApplicationContext applicationContext = new ClassPathXmlApplication
12        nContext("spring-aspectj-aop-annotation.xml");
13        OrderService orderService = applicationContext.getBean("orderServ
14        ce", OrderService.class);
15        orderService.generate();
16    }
17}
```

运行结果：

```
✓ Tests passed: 1 of 1 test – 275 ms
/Library/Java/JavaVirtualMachine
我是一个通知
订单已生成!
```

通知类型

通知类型包括：

- 前置通知：@Before 目标方法执行之前的通知
- 后置通知：@AfterReturning 目标方法执行之后的通知
- 环绕通知：@Around 目标方法之前添加通知，同时目标方法执行之后添加通知。
- 异常通知：@AfterThrowing 发生异常之后执行的通知
- 最终通知：@After 放在finally语句块中的通知

接下来，编写程序来测试这几个通知的执行顺序：

动力节点

MyAspect

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import org.aspectj.lang.ProceedingJoinPoint;
4 import org.aspectj.lang.annotation.*;
5 import org.springframework.stereotype.Component;
6
7 // 切面类
8 @Component
9 @Aspect
10 public class MyAspect {
11
12     @Around("execution(* com.powernode.spring6.service.OrderService.*(..))")
13     public void aroundAdvice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
14         System.out.println("环绕通知开始");
15         // 执行目标方法。
16         proceedingJoinPoint.proceed();
17         System.out.println("环绕通知结束");
18     }
19
20     @Before("execution(* com.powernode.spring6.service.OrderService.*(..))")
21     public void beforeAdvice(){
22         System.out.println("前置通知");
23     }
24
25     @AfterReturning("execution(* com.powernode.spring6.service.OrderService.*(..))")
26     public void afterReturningAdvice(){
27         System.out.println("后置通知");
28     }
29
30     @AfterThrowing("execution(* com.powernode.spring6.service.OrderService.*(..))")
31     public void afterThrowingAdvice(){
32         System.out.println("异常通知");
33     }
34
35     @After("execution(* com.powernode.spring6.service.OrderService.*(..))")
36     public void afterAdvice(){
37         System.out.println("最终通知");
38     }
39 }
```

```
40 }
```

▼ 目标类和目标方法

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import org.springframework.stereotype.Component;
4
5 // 目标类
6 @Component
7 public class OrderService {
8     // 目标方法
9     public void generate(){
10         System.out.println("订单已生成! ");
11     }
12 }
13
```

▼ 测试程序

Java | 复制代码

```
1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.service.OrderService;
4 import org.junit.Test;
5 import org.springframework.context.ApplicationContext;
6 import org.springframework.context.support.ClassPathXmlApplicationContext;
7
8 public class AOPTest {
9     @Test
10    public void testAOP(){
11        ApplicationContext applicationContext = new ClassPathXmlApplication
12          Context("spring-aspectj-aop-annotation.xml");
13        OrderService orderService = applicationContext.getBean("orderServ
14          ie", OrderService.class);
15        orderService.generate();
16    }
17 }
```

执行结果：

```
Tests passed: 1 of 1 test – 285 ms  
/Library/Java/JavaVirtualMachines/jdk-17.jdk/Contents/Home/bin/java -jar /Users/zhengzhiwei/IdeaProjects/Spring6-AspectJ/target/Spring6-AspectJ-0.0.1-SNAPSHOT.jar  
环绕通知开始  
前置通知  
订单已生成!  
后置通知  
最终通知  
环绕通知结束
```

动力节点

通过上面的执行结果就可以判断他们的执行顺序了，这里不再赘述。

结果中没有异常通知，这是因为目标程序执行过程中没有发生异常。我们尝试让目标方法发生异常：

让目标方法执行过程中发生异常

Java | 复制代码

```
1 package com.powernode.spring6.service;  
2  
3 import org.springframework.stereotype.Component;  
4  
5 // 目标类  
6 @Component  
7 public class OrderService {  
8     // 目标方法  
9     public void generate(){  
10         System.out.println("订单已生成! ");  
11         if (1 == 1) {  
12             throw new RuntimeException("模拟异常发生");  
13         }  
14     }  
15 }
```

再次执行测试程序，结果如下：

```
! Tests failed: 1 of 1 test – 278ms  
/Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/Cor  
环绕通知开始  
前置通知  
订单已生成!  
异常通知  
最终通知  
  
java.lang.RuntimeException: 模拟异常发生
```

动力节点

通过测试得知，当发生异常之后，最终通知也会执行，因为最终通知@After会出现在finally语句块中。

出现异常之后，**后置通知和环绕通知的结束部分**不会执行。

切面的先后顺序

我们知道，业务流程当中不一定只有一个切面，可能有的切面控制事务，有的记录日志，有的进行安全控制，如果多个切面的话，顺序如何控制：**可以使用@Order注解来标识切面类，为@Order注解的value指定一个整数型的数字，数字越小，优先级越高。**

再定义一个切面类，如下：

另一个切面类，并设置优先级

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import org.aspectj.lang.ProceedingJoinPoint;
4 import org.aspectj.lang.annotation.*;
5 import org.springframework.core.annotation.Order;
6 import org.springframework.stereotype.Component;
7
8 @Aspect
9 @Component
10 @Order(1) //设置优先级
11 public class YourAspect {
12
13     @Around("execution(* com.powernode.spring6.service.OrderService.*(..))")
14     public void aroundAdvice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
15         System.out.println("YourAspect环绕通知开始");
16         // 执行目标方法。
17         proceedingJoinPoint.proceed();
18         System.out.println("YourAspect环绕通知结束");
19     }
20
21     @Before("execution(* com.powernode.spring6.service.OrderService.*(..))")
22     public void beforeAdvice(){
23         System.out.println("YourAspect前置通知");
24     }
25
26     @AfterReturning("execution(* com.powernode.spring6.service.OrderService.*(..))")
27     public void afterReturningAdvice(){
28         System.out.println("YourAspect后置通知");
29     }
30
31     @AfterThrowing("execution(* com.powernode.spring6.service.OrderService.*(..))")
32     public void afterThrowingAdvice(){
33         System.out.println("YourAspect异常通知");
34     }
35
36     @After("execution(* com.powernode.spring6.service.OrderService.*(..))")
37     public void afterAdvice(){
38         System.out.println("YourAspect最终通知");
39     }
}
```

40 }

41

动力节点

▼ 设置切面类MyAspect的优先级

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import org.aspectj.lang.ProceedingJoinPoint;
4 import org.aspectj.lang.annotation.*;
5 import org.springframework.core.annotation.Order;
6 import org.springframework.stereotype.Component;
7
8 // 切面类
9 @Component
10 @Aspect
11 @Order(2) //设置优先级
12 public class MyAspect {
13
14     @Around("execution(* com.powernode.spring6.service.OrderService.*(..))")
15     public void aroundAdvice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
16         System.out.println("环绕通知开始");
17         // 执行目标方法。
18         proceedingJoinPoint.proceed();
19         System.out.println("环绕通知结束");
20     }
21
22     @Before("execution(* com.powernode.spring6.service.OrderService.*(..))")
23     public void beforeAdvice(){
24         System.out.println("前置通知");
25     }
26
27     @AfterReturning("execution(* com.powernode.spring6.service.OrderService.*(..))")
28     public void afterReturningAdvice(){
29         System.out.println("后置通知");
30     }
31
32     @AfterThrowing("execution(* com.powernode.spring6.service.OrderService.*(..))")
33     public void afterThrowingAdvice(){
34         System.out.println("异常通知");
35     }
36
37     @After("execution(* com.powernode.spring6.service.OrderService.*(..))")
38     public void afterAdvice(){
39         System.out.println("最终通知");
```

```
40      }
41
42 }
```

执行测试程序：

```
✓ Tests passed: 1 of 1 test – 294 ms
```

```
/Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/Contents/H
YourAspect环绕通知开始
YourAspect前置通知
环绕通知开始
前置通知
订单已生成!
后置通知
最终通知
环绕通知结束
YourAspect后置通知
YourAspect最终通知
YourAspect环绕通知结束
```

动力节点

通过修改@Order注解的整数值来切换顺序，执行测试程序：

```
✓ Tests passed: 1 of 1 test – 310 ms
```

```
/Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/Contents/H
环绕通知开始
前置通知
YourAspect环绕通知开始
YourAspect前置通知
订单已生成!
YourAspect后置通知
YourAspect最终通知
YourAspect环绕通知结束
后置通知
最终通知
环绕通知结束
```

动力节点

优化使用切点表达式

观看以下代码中的切点表达式：

```
1 package com.powernode.spring6.service;
2
3 import org.aspectj.lang.ProceedingJoinPoint;
4 import org.aspectj.lang.annotation.*;
5 import org.springframework.core.annotation.Order;
6 import org.springframework.stereotype.Component;
7
8 // 切面类
9 @Component
10 @Aspect
11 @Order(2)
12 public class MyAspect {
13
14     @Around("execution(* com.powernode.spring6.service.OrderService.*(..))")
15     public void aroundAdvice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
16         System.out.println("环绕通知开始");
17         // 执行目标方法。
18         proceedingJoinPoint.proceed();
19         System.out.println("环绕通知结束");
20     }
21
22     @Before("execution(* com.powernode.spring6.service.OrderService.*(..))")
23     public void beforeAdvice(){
24         System.out.println("前置通知");
25     }
26
27     @AfterReturning("execution(* com.powernode.spring6.service.OrderService.*(..))")
28     public void afterReturningAdvice(){
29         System.out.println("后置通知");
30     }
31
32     @AfterThrowing("execution(* com.powernode.spring6.service.OrderService.*(..))")
33     public void afterThrowingAdvice(){
34         System.out.println("异常通知");
35     }
36
37     @After("execution(* com.powernode.spring6.service.OrderService.*(..))")
38     public void afterAdvice(){
39         System.out.println("最终通知");
```

```
40      }
41
42  }
43
```

缺点是：

- 第一：切点表达式重复写了多次，没有得到复用。
- 第二：如果要修改切点表达式，需要修改多处，难维护。

可以这样做：将切点表达式单独的定义出来，在需要的位置引入即可。如下：

```
1 package com.powernode.spring6.service;
2
3 import org.aspectj.lang.ProceedingJoinPoint;
4 import org.aspectj.lang.annotation.*;
5 import org.springframework.core.annotation.Order;
6 import org.springframework.stereotype.Component;
7
8 // 切面类
9 @Component
10 @Aspect
11 @Order(2)
12 public class MyAspect {
13
14     @Pointcut("execution(* com.powernode.spring6.service.OrderService.*(..))")
15     public void pointcut(){}
16
17     @Around("pointcut()")
18     public void aroundAdvice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
19         System.out.println("环绕通知开始");
20         // 执行目标方法。
21         proceedingJoinPoint.proceed();
22         System.out.println("环绕通知结束");
23     }
24
25     @Before("pointcut()")
26     public void beforeAdvice(){
27         System.out.println("前置通知");
28     }
29
30     @AfterReturning("pointcut()")
31     public void afterReturningAdvice(){
32         System.out.println("后置通知");
33     }
34
35     @AfterThrowing("pointcut()")
36     public void afterThrowingAdvice(){
37         System.out.println("异常通知");
38     }
39
40     @After("pointcut()")
41     public void afterAdvice(){
42         System.out.println("最终通知");
43     }
}
```

```
44  
45 }  
46
```

使用@Pointcut注解来定义独立的切点表达式。

注意这个@Pointcut注解标注的方法随意，只是起到一个能够让@Pointcut注解编写的位置。

执行测试程序：

The terminal window displays the following output:

```
Tests passed: 1 of 1 test - 310 ms  
/Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/Contents/Ho  
环绕通知开始  
前置通知  
YourAspect环绕通知开始  
YourAspect前置通知  
订单已生成!  
YourAspect后置通知  
YourAspect最终通知  
YourAspect环绕通知结束  
后置通知  
最终通知  
环绕通知结束
```

动力节点

全注解式开发AOP

就是编写一个类，在这个类上面使用大量注解来代替spring的配置文件，spring配置文件消失了，如下：

```
1 package com.powernode.spring6.service;  
2  
3 import org.springframework.context.annotation.ComponentScan;  
4 import org.springframework.context.annotation.Configuration;  
5 import org.springframework.context.annotation.EnableAspectJAutoProxy;  
6  
7 @Configuration  
8 @ComponentScan("com.powernode.spring6.service")  
9 @EnableAspectJAutoProxy(proxyTargetClass = true)  
10 public class Spring6Configuration {  
11 }
```

测试程序也变化了：

Java | 复制代码

```
1 @Test
2 public void testAOPWithAllAnnotation(){
3     ApplicationContext applicationContext = new AnnotationConfigApplication
4         Context(Spring6Configuration.class);
5     OrderService orderService = applicationContext.getBean("orderService",
6     OrderService.class);
7     orderService.generate();
8 }
```

执行结果如下：

```
✓ Tests passed: 1 of 1 test – 310 ms
/Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/Contents/Home
环绕通知开始
前置通知
YourAspect环绕通知开始
YourAspect前置通知
订单已生成!
YourAspect后置通知
YourAspect最终通知
YourAspect环绕通知结束
后置通知
最终通知
环绕通知结束
```

动力节点

15.4.3 基于XML配置方式的AOP（了解）

第一步：编写目标类

不添加@Component注解

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 // 目标类
4 public class VipService {
5     public void add(){
6         System.out.println("保存vip信息。");
7     }
8 }
```

第二步：编写切面类，并且编写通知

不添加@Component注解

Java | 复制代码

```
1 package com.powernode.spring6.service;
2
3 import org.aspectj.lang.ProceedingJoinPoint;
4
5 // 负责计时的切面类
6 public class TimerAspect {
7
8     public void time(ProceedingJoinPoint proceedingJoinPoint) throws Throw
9         able {
10         long begin = System.currentTimeMillis();
11         //执行目标
12         proceedingJoinPoint.proceed();
13         long end = System.currentTimeMillis();
14         System.out.println("耗时"+(end - begin)+"毫秒");
15     }
16 }
```

第三步：编写spring配置文件

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xmlns:aop="http://www.springframework.org/schema/aop"
6   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
7                           http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd
8                           http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">
9
10    <!--纳入spring bean管理-->
11    <bean id="vipService" class="com.powernode.spring6.service.VipService"
12      />
13    <bean id="timerAspect" class="com.powernode.spring6.service.TimerAspect"/>
14
15    <!--aop配置-->
16    <aop:config>
17      <!--切点表达式-->
18      <aop:pointcut id="p" expression="execution(* com.powernode.spring
19       6.service.VipService.*(..))"/>
20      <!--切面-->
21      <aop:aspect ref="timerAspect">
22        <!--切面=通知 + 切点-->
23        <aop:around method="time" pointcut-ref="p"/>
24      </aop:aspect>
25    </aop:config>
26  </beans>
```

测试程序：

```

1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.service.VipService;
4 import org.junit.Test;
5 import org.springframework.context.ApplicationContext;
6 import org.springframework.context.support.ClassPathXmlApplicationContext;
7
8 public class AOPTest3 {
9
10    @Test
11    public void testAOPXml(){
12        ApplicationContext applicationContext = new ClassPathXmlApplication
13        nContext("spring-aop-xml.xml");
14        VipService vipService = applicationContext.getBean("vipService", V
15        ipService.class);
16        vipService.add();
17    }

```

执行结果：

```

✓ Tests passed: 1 of 1 test – 275 ms
/Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/
保存vip信息。
耗时4毫秒

Process finished with exit code 0

```

15.5 AOP的实际案例：事务处理

项目中的事务控制是在所难免的。在一个业务流程当中，可能需要多条DML语句共同完成，为了保证数据的安全，这多条DML语句要么同时成功，要么同时失败。这就需要添加事务控制的代码。例如以下伪代码：

▼ 伪代码

Java | 复制代码

```
1 class 业务类1{
2     public void 业务方法1(){
3         try{
4             // 开启事务
5             startTransaction();
6
7             // 执行核心业务逻辑
8             step1();
9             step2();
10            step3();
11            ....
12
13            // 提交事务
14            commitTransaction();
15        }catch(Exception e){
16            // 回滚事务
17            rollbackTransaction();
18        }
19    }
20    public void 业务方法2(){
21        try{
22            // 开启事务
23            startTransaction();
24
25            // 执行核心业务逻辑
26            step1();
27            step2();
28            step3();
29            ....
30
31            // 提交事务
32            commitTransaction();
33        }catch(Exception e){
34            // 回滚事务
35            rollbackTransaction();
36        }
37    }
38    public void 业务方法3(){
39        try{
40            // 开启事务
41            startTransaction();
42
43            // 执行核心业务逻辑
44            step1();
45            step2();
```

```
46             step3();
47             ....
48
49             // 提交事务
50             commitTransaction();
51         }catch(Exception e){
52             // 回滚事务
53             rollbackTransaction();
54         }
55     }
56 }
57
58 class 业务类2{
59     public void 业务方法1(){
60         try{
61             // 开启事务
62             startTransaction();
63
64             // 执行核心业务逻辑
65             step1();
66             step2();
67             step3();
68             ....
69
70             // 提交事务
71             commitTransaction();
72         }catch(Exception e){
73             // 回滚事务
74             rollbackTransaction();
75         }
76     }
77     public void 业务方法2(){
78         try{
79             // 开启事务
80             startTransaction();
81
82             // 执行核心业务逻辑
83             step1();
84             step2();
85             step3();
86             ....
87
88             // 提交事务
89             commitTransaction();
90         }catch(Exception e){
91             // 回滚事务
92             rollbackTransaction();
93     }
```

```
94    }
95    public void 业务方法3(){
96        try{
97            // 开启事务
98            startTransaction();
99
100           // 执行核心业务逻辑
101           step1();
102           step2();
103           step3();
104           ....
105
106           // 提交事务
107           commitTransaction();
108       }catch(Exception e){
109           // 回滚事务
110           rollbackTransaction();
111       }
112   }
113 }
//.....
```

可以看到，这些业务类中的每一个业务方法都是需要控制事务的，而控制事务的代码又是固定的格式，都是：

```
1 try{
2     // 开启事务
3     startTransaction();
4
5     // 执行核心业务逻辑
6     //.....
7
8     // 提交事务
9     commitTransaction();
10    }catch(Exception e){
11        // 回滚事务
12        rollbackTransaction();
13    }
```

这个控制事务的代码就是和业务逻辑没有关系的“**交叉业务**”。以上伪代码当中可以看到这些交叉业务的代码没有得到复用，并且如果这些交叉业务代码需要修改，那必然需要修改多处，难维护，怎么解决？可以采用AOP思想解决。可以把以上控制事务的代码作为环绕通知，切入到目标类的方法当中。接下来我们做一下这件事，有两个业务类，如下：

银行账户的业务类

Java | 复制代码

```
1 package com.powernode.spring6.biz;
2
3 import org.springframework.stereotype.Component;
4
5 @Component
6 // 业务类
7 public class AccountService {
8     // 转账业务方法
9     public void transfer(){
10         System.out.println("正在进行银行账户转账");
11     }
12     // 取款业务方法
13     public void withdraw(){
14         System.out.println("正在进行取款操作");
15     }
16 }
17
```

订单业务类

Java | 复制代码

```
1 package com.powernode.spring6.biz;
2
3 import org.springframework.stereotype.Component;
4
5 @Component
6 // 业务类
7 public class OrderService {
8     // 生成订单
9     public void generate(){
10         System.out.println("正在生成订单");
11     }
12     // 取消订单
13     public void cancel(){
14         System.out.println("正在取消订单");
15     }
16 }
```

注意，以上两个业务类已经纳入spring bean的管理，因为都添加了@Component注解。

接下来我们给以上两个业务类的4个方法添加事务控制代码，使用AOP来完成：

事务切面类

Java | 复制代码

```
1 package com.powernode.spring6.biz;
2
3 import org.aspectj.lang.ProceedingJoinPoint;
4 import org.aspectj.lang.annotation.Around;
5 import org.aspectj.lang.annotation.Aspect;
6 import org.springframework.stereotype.Component;
7
8 @Aspect
9 @Component
10 // 事务切面类
11 public class TransactionAspect {
12
13     @Around("execution(* com.powernode.spring6.biz..*(..))")
14     public void aroundAdvice(ProceedingJoinPoint proceedingJoinPoint){
15         try {
16             System.out.println("开启事务");
17             // 执行目标
18             proceedingJoinPoint.proceed();
19             System.out.println("提交事务");
20         } catch (Throwable e) {
21             System.out.println("回滚事务");
22         }
23     }
24 }
25
```

你看，这个事务控制代码是不是只需要写一次就行了，并且修改起来也没有成本。编写测试程序：

```
1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.biz.AccountService;
4 import com.powernode.spring6.biz.OrderService;
5 import com.powernode.spring6.service.Spring6Configuration;
6 import org.junit.Test;
7 import org.springframework.context.ApplicationContext;
8 import org.springframework.context.annotation.AnnotationConfigApplicationContext;
9
10 public class AOPTest2 {
11     @Test
12     public void testTransaction(){
13         ApplicationContext applicationContext = new AnnotationConfigApplicationContext(Spring6Configuration.class);
14         OrderService orderService = applicationContext.getBean("orderService", OrderService.class);
15         AccountService accountService = applicationContext.getBean("accountService", AccountService.class);
16         // 生成订单
17         orderService.generate();
18         // 取消订单
19         orderService.cancel();
20         // 转账
21         accountService.transfer();
22         // 取款
23         accountService.withdraw();
24     }
25 }
```

执行结果：

```
>> ✓ Tests passed: 1 of 1 test – 250 ms  
ns /Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/  
ns  
开启事务  
正在生成订单  
提交事务  
开启事务  
正在取消订单  
提交事务  
开启事务  
正在进行银行账户转账  
提交事务  
开启事务  
正在进行取款操作  
提交事务
```

动力节点

通过测试可以看到，所有的业务方法都添加了事务控制的代码。

15.6 AOP的实际案例：安全日志

需求是这样的：项目开发结束了，已经上线了。运行正常。客户提出了新的需求：凡事在系统中进行修改操作的，删除操作的，新增操作的，都要把这个记录下来。因为这几个操作是属于危险行为。例如有业务类和业务方法：

用户业务类

Java | 复制代码

```
1 package com.powernode.spring6.biz;
2
3 import org.springframework.stereotype.Component;
4
5 @Component
6 //用户业务
7 public class UserService {
8     public void getUser(){
9         System.out.println("获取用户信息");
10    }
11    public void saveUser(){
12        System.out.println("保存用户");
13    }
14    public void deleteUser(){
15        System.out.println("删除用户");
16    }
17    public void modifyUser(){
18        System.out.println("修改用户");
19    }
20 }
```

商品业务类

Java | 复制代码

```
1 package com.powernode.spring6.biz;
2
3 import org.springframework.stereotype.Component;
4
5 // 商品业务类
6 @Component
7 public class ProductService {
8     public void getProduct(){
9         System.out.println("获取商品信息");
10    }
11    public void saveProduct(){
12        System.out.println("保存商品");
13    }
14    public void deleteProduct(){
15        System.out.println("删除商品");
16    }
17    public void modifyProduct(){
18        System.out.println("修改商品");
19    }
20 }
21
22
```

注意：已经添加了@Component注解。

接下来我们使用aop来解决上面的需求：编写一个负责安全的切面类

▼ 负责安全的切面类

Java | 复制代码

```
1 package com.powernode.spring6.biz;
2
3 import org.aspectj.lang.JoinPoint;
4 import org.aspectj.lang.annotation.Aspect;
5 import org.aspectj.lang.annotation.Before;
6 import org.aspectj.lang.annotation.Pointcut;
7 import org.springframework.stereotype.Component;
8
9 @Component
10 @Aspect
11 public class SecurityAspect {
12
13     @Pointcut("execution(* com.powernode.spring6.biz..save*(..))")
14     public void savePointcut(){}
15
16     @Pointcut("execution(* com.powernode.spring6.biz..delete*(..))")
17     public void deletePointcut(){}
18
19     @Pointcut("execution(* com.powernode.spring6.biz..modify*(..))")
20     public void modifyPointcut(){}
21
22     @Before("savePointcut() || deletePointcut() || modifyPointcut()")
23     public void beforeAdvice(JoinPoint joinpoint){
24         System.out.println("XXX操作员正在操作"+joinpoint.getSignature().getName()+"方法");
25     }
26 }
27
```

测试程序

Java | 复制代码

```
1  @Test
2  public void testSecurity(){
3      ApplicationContext applicationContext = new AnnotationConfigApplication
nContext(Spring6Configuration.class);
4      UserService userService = applicationContext.getBean("userService", Us
erService.class);
5      ProductService productService = applicationContext.getBean("productSer
vice", ProductService.class);
6      userService.getUser();
7      userService.saveUser();
8      userService.deleteUser();
9      userService.modifyUser();
10     productService.getProduct();
11     productService.saveProduct();
12     productService.deleteProduct();
13     productService.modifyProduct();
14 }
```

执行结果:

```
✓ Tests passed: 1 of 1 test – 269 ms
/Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/Co
获取用户信息
XXX操作员正在操作saveUser方法
保存用户
XXX操作员正在操作deleteUser方法
删除用户
XXX操作员正在操作modifyUser方法
修改用户
获取商品信息
XXX操作员正在操作saveProduct方法
保存商品
XXX操作员正在操作deleteProduct方法
删除商品
XXX操作员正在操作modifyProduct方法
修改商品

Process finished with exit code 0
```

动力节点

十六、Spring对事务的支持

16.1 事务概述

- 什么是事务

- 在一个业务流程当中，通常需要多条DML（insert delete update）语句共同联合才能完成，这多条DML语句必须同时成功，或者同时失败，这样才能保证数据的安全。
- 多条DML要么同时成功，要么同时失败，这叫做事务。
- 事务：Transaction (tx)

- 事务的四个处理过程：

- 第一步：开启事务 (start transaction)
- 第二步：执行核心业务代码
- 第三步：提交事务（如果核心业务处理过程中没有出现异常）(commit transaction)
- 第四步：回滚事务（如果核心业务处理过程中出现异常）(rollback transaction)

- 事务的四个特性：

- A 原子性：事务是最小的工作单元，不可再分。
- C 一致性：事务要求要么同时成功，要么同时失败。事务前和事务后的总量不变。
- I 隔离性：事务和事务之间因为有隔离性，才可以保证互不干扰。
- D 持久性：持久性是事务结束的标志。

16.2 引入事务场景

以银行账户转账为例学习事务。两个账户act-001和act-002。act-001账户向act-002账户转账10000，必须同时成功，或者同时失败。（一个减成功，一个加成功，这两条update语句必须同时成功，或同时失败。）

连接数据库的技术采用Spring框架的JdbcTemplate。

采用三层架构搭建：



模块名: spring6-013-tx-bank (依赖如下)

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>com.powernode</groupId>
8     <artifactId>spring6-013-tx-bank</artifactId>
9     <version>1.0-SNAPSHOT</version>
10    <packaging>jar</packaging>
11
12    <!--仓库-->
13    <repositories>
14        <!--spring里程碑版本的仓库-->
15        <repository>
16            <id>repository.spring.milestone</id>
17            <name>Spring Milestone Repository</name>
18            <url>https://repo.spring.io/milestone</url>
19        </repository>
20    </repositories>
21
22    <!--依赖-->
23    <dependencies>
24        <!--spring context-->
25        <dependency>
26            <groupId>org.springframework</groupId>
27            <artifactId>spring-context</artifactId>
28            <version>6.0.0-M2</version>
29        </dependency>
30        <!--spring jdbc-->
31        <dependency>
32            <groupId>org.springframework</groupId>
33            <artifactId>spring-jdbc</artifactId>
34            <version>6.0.0-M2</version>
35        </dependency>
36        <!--mysql驱动-->
37        <dependency>
38            <groupId>mysql</groupId>
39            <artifactId>mysql-connector-java</artifactId>
40            <version>8.0.30</version>
41        </dependency>
42        <!--德鲁伊连接池-->
43        <dependency>
44            <groupId>com.alibaba</groupId>
```

```

45         <artifactId>druid</artifactId>
46         <version>1.2.13</version>
47     </dependency>
48     <!--@Resource注解-->
49     <dependency>
50         <groupId>jakarta.annotation</groupId>
51         <artifactId>jakarta.annotation-api</artifactId>
52         <version>2.1.1</version>
53     </dependency>
54     <!--junit-->
55     <dependency>
56         <groupId>junit</groupId>
57         <artifactId>junit</artifactId>
58         <version>4.13.2</version>
59         <scope>test</scope>
60     </dependency>
61   </dependencies>
62 
63   <properties>
64     <maven.compiler.source>17</maven.compiler.source>
65     <maven.compiler.target>17</maven.compiler.target>
66   </properties>
67 
68 </project>

```

第一步：准备数据库表

表结构：

名	类型	长度	小数点	不是 null	虚拟	键	注释
actno	varchar	255		<input checked="" type="checkbox"/>	<input type="checkbox"/>		账号
balance	double			<input type="checkbox"/>	<input type="checkbox"/>		余额

表数据：

actno	balance
act-001	50000
act-002	0

第二步：创建包结构

com.powernode.bank.pojo
com.powernode.bank.service
com.powernode.bank.service.impl
com.powernode.bank.dao
com.powernode.bank.dao.impl

第三步：准备POJO类

```
1 package com.powernode.bank.pojo;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Account
7  * @since 1.0
8 */
9 public class Account {
10     private String actno;
11     private Double balance;
12
13     @Override
14     public String toString() {
15         return "Account{" +
16                 "actno='" + actno + '\'' +
17                 ", balance=" + balance +
18                 '}';
19     }
20
21     public Account() {
22     }
23
24     public Account(String actno, Double balance) {
25         this.actno = actno;
26         this.balance = balance;
27     }
28
29     public String getActno() {
30         return actno;
31     }
32
33     public void setActno(String actno) {
34         this.actno = actno;
35     }
36
37     public Double getBalance() {
38         return balance;
39     }
40
41     public void setBalance(Double balance) {
42         this.balance = balance;
43     }
44 }
45
```

第四步：编写持久层

```
1 package com.powernode.bank.dao;
2
3 import com.powernode.bank.pojo.Account;
4
5 /**
6  * @author 动力节点
7  * @version 1.0
8  * @className AccountDao
9  * @since 1.0
10 */
11 public interface AccountDao {
12
13     /**
14      * 根据账号查询余额
15      * @param actno
16      * @return
17      */
18     Account selectByActno(String actno);
19
20     /**
21      * 更新账户
22      * @param act
23      * @return
24      */
25     int update(Account act);
26
27 }
28
```

```
1 package com.powernode.bank.dao.impl;
2
3 import com.powernode.bank.dao.AccountDao;
4 import com.powernode.bank.pojo.Account;
5 import jakarta.annotation.Resource;
6 import org.springframework.jdbc.core.BeanPropertyRowMapper;
7 import org.springframework.jdbc.core.JdbcTemplate;
8 import org.springframework.stereotype.Component;
9
10 /**
11  * @author 动力节点
12  * @version 1.0
13  * @className AccountDaoImpl
14  * @since 1.0
15 */
16 @Repository("accountDao")
17 public class AccountDaoImpl implements AccountDao {
18
19     @Resource(name = "jdbcTemplate")
20     private JdbcTemplate jdbcTemplate;
21
22     @Override
23     public Account selectByActno(String actno) {
24         String sql = "select actno, balance from t_act where actno = ?";
25         Account account = jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<>(Account.class), actno);
26         return account;
27     }
28
29     @Override
30     public int update(Account act) {
31         String sql = "update t_act set balance = ? where actno = ?";
32         int count = jdbcTemplate.update(sql, act.getBalance(), act.getActno());
33         return count;
34     }
35 }
36
```

第五步：编写业务层

```
1 package com.powernode.bank.service;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className AccountService
7  * @since 1.0
8 */
9 public interface AccountService {
10
11     /**
12      * 转账
13      * @param fromActno
14      * @param toActno
15      * @param money
16      */
17     void transfer(String fromActno, String toActno, double money);
18 }
19
```

```
1 package com.powernode.bank.service.impl;
2
3 import com.powernode.bank.dao.AccountDao;
4 import com.powernode.bank.pojo.Account;
5 import com.powernode.bank.service.AccountService;
6 import jakarta.annotation.Resource;
7 import org.springframework.stereotype.Service;
8
9 /**
10 * @author 动力节点
11 * @version 1.0
12 * @className AccountServiceImpl
13 * @since 1.0
14 */
15 @Service("accountService")
16 public class AccountServiceImpl implements AccountService {
17
18     @Resource(name = "accountDao")
19     private AccountDao accountDao;
20
21     @Override
22     public void transfer(String fromActno, String toActno, double money) {
23         // 查询账户余额是否充足
24         Account fromAct = accountDao.selectByActno(fromActno);
25         if (fromAct.getBalance() < money) {
26             throw new RuntimeException("账户余额不足");
27         }
28         // 余额充足，开始转账
29         Account toAct = accountDao.selectByActno(toActno);
30         fromAct.setBalance(fromAct.getBalance() - money);
31         toAct.setBalance(toAct.getBalance() + money);
32         int count = accountDao.update(fromAct);
33         count += accountDao.update(toAct);
34         if (count != 2) {
35             throw new RuntimeException("转账失败，请联系银行");
36         }
37     }
38 }
39
```

第六步：编写Spring配置文件

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
6                           http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">
7
8     <context:component-scan base-package="com.powernode.bank"/>
9
10    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
11        <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
12        <property name="url" value="jdbc:mysql://localhost:3306/spring6"/>
13        <property name="username" value="root"/>
14        <property name="password" value="root"/>
15    </bean>
16
17    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
18        <property name="dataSource" ref="dataSource"/>
19    </bean>
20
21 </beans>
```

第七步：编写表示层（测试程序）

Java | 复制代码

```
1 package com.powernode.spring6.test;
2
3 import com.powernode.bank.service.AccountService;
4 import org.junit.Test;
5 import org.springframework.context.ApplicationContext;
6 import org.springframework.context.support.ClassPathXmlApplicationContext;
7
8 /**
9  * @author 动力节点
10 * @version 1.0
11 * @className BankTest
12 * @since 1.0
13 */
14 public class BankTest {
15     @Test
16     public void testTransfer(){
17         ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
18         AccountService accountService = applicationContext.getBean("accountService", AccountService.class);
19         try {
20             accountService.transfer("act-001", "act-002", 10000);
21             System.out.println("转账成功");
22         } catch (Exception e) {
23             e.printStackTrace();
24         }
25     }
26 }
27 }
28 }
```

执行结果:

```
Tests passed: 1 of 1 test - 1 sec 279 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
10月 23, 2022 12:01:00 下午 com.alibaba.druid.sup
信息: {dataSource-1} init
转账成功
```

动力节点

数据变化:

动力节点

动力节点

对象 t_act @spring6 (localhost) - 表

开始事务 文本 筛选 排序 导入 导出

actno	balance
act-001	40000
act-002	10000

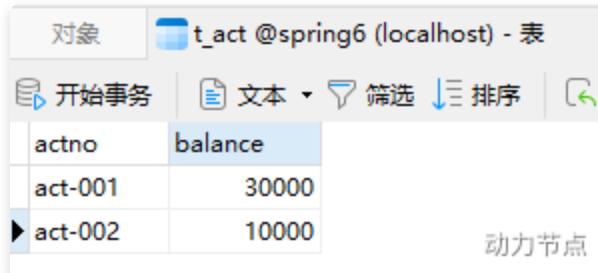
模拟异常

```
1 package com.powernode.bank.service.impl;
2
3 import com.powernode.bank.dao.AccountDao;
4 import com.powernode.bank.pojo.Account;
5 import com.powernode.bank.service.AccountService;
6 import jakarta.annotation.Resource;
7 import org.springframework.stereotype.Service;
8
9 /**
10 * @author 动力节点
11 * @version 1.0
12 * @className AccountServiceImpl
13 * @since 1.0
14 */
15 @Service("accountService")
16 public class AccountServiceImpl implements AccountService {
17
18     @Resource(name = "accountDao")
19     private AccountDao accountDao;
20
21     @Override
22     public void transfer(String fromActno, String toActno, double money) {
23         // 查询账户余额是否充足
24         Account fromAct = accountDao.selectByActno(fromActno);
25         if (fromAct.getBalance() < money) {
26             throw new RuntimeException("账户余额不足");
27         }
28         // 余额充足，开始转账
29         Account toAct = accountDao.selectByActno(toActno);
30         fromAct.setBalance(fromAct.getBalance() - money);
31         toAct.setBalance(toAct.getBalance() + money);
32         int count = accountDao.update(fromAct);
33
34         // 模拟异常
35         String s = null;
36         s.toString();
37
38         count += accountDao.update(toAct);
39         if (count != 2) {
40             throw new RuntimeException("转账失败，请联系银行");
41         }
42     }
43 }
44
```

执行结果：

```
Tests passed: 1 of 1 test – 1 sec 187 ms  
C:\dev\Java\jdk-17.0.4\bin\java.exe ...  
10月 23, 2022 12:03:14 下午 com.alibaba.druid.support.logging.JakartaCommonsLoggingImpl info  
信息: {dataSource-1} init  
java.lang.NullPointerException Create breakpoint : Cannot invoke "String.toString()" because "s" is null  
    at com.powernode.bank.service.impl.AccountServiceImpl.transfer(AccountServiceImpl.java:36)  
    at com.powernode.spring6.test.BankTest.testTransfer(BankTest.java:20) <27 internal lines>
```

数据库表中数据：



actno	balance
act-001	30000
act-002	10000

丢了1万。

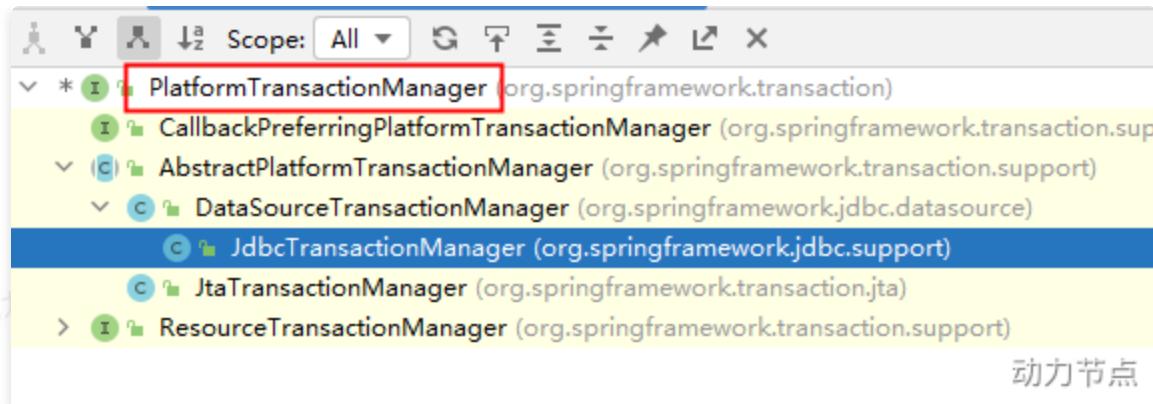
16.3 Spring对事务的支持

Spring实现事务的两种方式

- 编程式事务
 - 通过编写代码的方式来实现事务的管理。
- 声明式事务
 - 基于注解方式
 - 基于XML配置方式

Spring事务管理API

Spring对事务的管理底层实现方式是基于AOP实现的。采用AOP的方式进行了封装。所以Spring专门针对事务开发了一套API，API的核心接口如下：



PlatformTransactionManager接口：spring事务管理器的核心接口。在Spring6中它有两个实现：

- DataSourceTransactionManager：支持JdbcTemplate、MyBatis、Hibernate等事务管理。
- JtaTransactionManager：支持分布式事务管理。

如果要在Spring6中使用JdbcTemplate，就要使用DataSourceTransactionManager来管理事务。
(Spring内置写好了，可以直接用。)

声明式事务之注解实现方式

- 第一步：在spring配置文件中配置事务管理器。

```

1 <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
2   <property name="dataSource" ref="dataSource"/>
3 </bean>

```

- 第二步：在spring配置文件中引入tx命名空间。

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xmlns:tx="http://www.springframework.org/schema/tx"
6   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
7                                     http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd
8                                     http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd">

```

- 第三步：在spring配置文件中配置“事务注解驱动器”，开始注解的方式控制事务。

XML | 复制代码

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

- 第四步：在service类上或方法上添加@Transactional注解

在类上添加该注解，该类中所有的方法都有事务。在某个方法上添加该注解，表示只有这个方法使用事务。

```
1 package com.powernode.bank.service.impl;
2
3 import com.powernode.bank.dao.AccountDao;
4 import com.powernode.bank.pojo.Account;
5 import com.powernode.bank.service.AccountService;
6 import jakarta.annotation.Resource;
7 import org.springframework.stereotype.Service;
8 import org.springframework.transaction.annotation.Transactional;
9
10 /**
11  * @author 动力节点
12  * @version 1.0
13  * @className AccountServiceImpl
14  * @since 1.0
15 */
16 @Service("accountService")
17 @Transactional
18 public class AccountServiceImpl implements AccountService {
19
20     @Resource(name = "accountDao")
21     private AccountDao accountDao;
22
23     @Override
24     public void transfer(String fromActno, String toActno, double money) {
25         // 查询账户余额是否充足
26         Account fromAct = accountDao.selectByActno(fromActno);
27         if (fromAct.getBalance() < money) {
28             throw new RuntimeException("账户余额不足");
29         }
30         // 余额充足，开始转账
31         Account toAct = accountDao.selectByActno(toActno);
32         fromAct.setBalance(fromAct.getBalance() - money);
33         toAct.setBalance(toAct.getBalance() + money);
34         int count = accountDao.update(fromAct);
35
36         // 模拟异常
37         String s = null;
38         s.toString();
39
40         count += accountDao.update(toAct);
41         if (count != 2) {
42             throw new RuntimeException("转账失败，请联系银行");
43         }
44     }
45 }
```

当前数据库表中的数据：

actno	balance
act-001	30000
act-002	10000

执行测试程序：

```
Tests passed: 1 of 1 test - 1 sec 400 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
10月 23, 2022 2:08:56 下午 com.alibaba.druid.support.logging.JakartaCommonsLoggingImpl info
信息: {dataSource-1} init
java.lang.NullPointerException Create breakpoint : Cannot invoke "String.toString()" because "s" is null
    at com.powernode.bank.service.impl.AccountServiceImpl.transfer(AccountServiceImpl.java:38) <4 inter
    at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:343) 动力节点
    at org.springframework.aop.framework.ReflectiveMethodInvocation.invokeJoinpoint(ReflectiveMethodInv...
```

虽然出现异常了，再次查看数据库表中数据：

actno	balance
act-001	30000
act-002	10000

通过测试，发现数据没有变化，事务起作用了。

事务属性

事务属性包括哪些

```

public @interface Transactional {
    @AliasFor("transactionManager")
    String value() default "";
}

    @AliasFor("value")
    String transactionManager() default "";

    String[] label() default {};

    Propagation propagation() default Propagation.REQUIRED;

    Isolation isolation() default Isolation.DEFAULT;

    int timeout() default -1;

    String timeoutString() default "";

    boolean readOnly() default false;

    Class<? extends Throwable>[] rollbackFor() default {};

    String[] rollbackForClassName() default {};

    Class<? extends Throwable>[] noRollbackFor() default {};

    String[] noRollbackForClassName() default {};
}

```

动力节点

事务中的重点属性：

- 事务传播行为
- 事务隔离级别
- 事务超时
- 只读事务
- 设置出现哪些异常回滚事务
- 设置出现哪些异常不回滚事务

事务传播行为

什么是事务的传播行为？

在service类中有a()方法和b()方法，a()方法上有事务，b()方法上也有事务，当a()方法执行过程中调用了b()方法，事务是如何传递的？合并到一个事务里？还是开启一个新的事务？这就是事务传播行为。

事务传播行为在spring框架中被定义为枚举类型：

```
6 package org.springframework.transaction.annotation;
7
8 public enum Propagation {
9     REQUIRED( value: 0 ),
10    SUPPORTS( value: 1 ),
11    MANDATORY( value: 2 ),
12    REQUIRES_NEW( value: 3 ),
13    NOT_SUPPORTED( value: 4 ),
14    NEVER( value: 5 ),
15    NESTED( value: 6 );
```

一共有七种传播行为：

- REQUIRED：支持当前事务，如果不存在就新建一个（默认）【没有就新建，有就加入】
- SUPPORTS：支持当前事务，如果当前没有事务，就以非事务方式执行 【有就加入，没有就不管了】
- MANDATORY：必须运行在一个事务中，如果当前没有事务正在发生，将抛出一个异常 【有就加入，没有就抛异常】
- REQUIRES_NEW：开启一个新的事务，如果一个事务已经存在，则将这个存在的事务挂起
【不管有没有，直接开启一个新事务，开启的新事务和之前的事务不存在嵌套关系，之前事务被挂起】
- NOT_SUPPORTED：以非事务方式运行，如果有事务存在，挂起当前事务 【不支持事务，存在就挂起】
- NEVER：以非事务方式运行，如果有事务存在，抛出异常 【不支持事务，存在就抛异常】
- NESTED：如果当前正有一个事务在进行中，则该方法应当运行在一个嵌套式事务中。被嵌套的事务可以独立于外层事务进行提交或回滚。如果外层事务不存在，行为就像REQUIRED一样。
【有事务的话，就在这个事务里再嵌套一个完全独立的事务，嵌套的事务可以独立的提交和回滚。没有事务就和REQUIRED一样。】

在代码中设置事务的传播行为：

```
Java | 复制代码  
1 @Transactional(propagation = Propagation.REQUIRED)
```

可以编写程序测试一下传播行为：

```
1号service Java | 复制代码  
1 @Transactional(propagation = Propagation.REQUIRED)  
2 public void save(Account act) {  
3  
4     // 这里调用dao的insert方法。  
5     accountDao.insert(act); // 保存act-003账户  
6  
7     // 创建账户对象  
8     Account act2 = new Account("act-004", 1000.0);  
9     try {  
10         accountService.save(act2); // 保存act-004账户  
11     } catch (Exception e) {  
12     }  
13     // 继续往后进行我当前1号事务自己的事儿。  
14 }  
15 }
```

```
2号service Java | 复制代码  
1 @Override  
2 // @Transactional(propagation = Propagation.REQUIRED)  
3 @Transactional(propagation = Propagation.REQUIRES_NEW)  
4 public void save(Account act) {  
5     accountDao.insert(act);  
6     // 模拟异常  
7     String s = null;  
8     s.toString();  
9  
10    // 事儿没有处理完，这个大括号当中的后续也许还有其他的DML语句。  
11 }
```

一定要集成Log4j2日志框架，在日志信息中可以看到更加详细的信息。

事务隔离级别

事务隔离级别类似于教室A和教室B之间的那道墙，隔离级别越高表示墙体越厚。隔音效果越好。

数据库中读取数据存在的三大问题：（三大读问题）

- 脏读：读取到没有提交到数据库的数据，叫做脏读。
- 不可重复读：在同一个事务当中，第一次和第二次读取的数据不一样。
- 幻读：读到的数据是假的。

事务隔离级别包括四个级别：

- 读未提交：READ_UNCOMMITTED
 - 这种隔离级别，存在脏读问题，所谓的脏读(dirty read)表示能够读取到其它事务未提交的数据。
- 读提交：READ_COMMITTED
 - 解决了脏读问题，其它事务提交之后才能读到，但存在不可重复读问题。
- 可重复读：REPEATABLE_READ
 - 解决了不可重复读，可以达到可重复读效果，只要当前事务不结束，读取到的数据一直都是相同的。但存在幻读问题。
- 序列化：SERIALIZABLE
 - 解决了幻读问题，事务排队执行。不支持并发。

大家可以通过一个表格来记忆：

隔离级别	脏读	不可重复读	幻读
读未提交	有	有	有
读提交	无	有	有
可重复读	无	无	有
序列化	无	无	无

在Spring代码中如何设置隔离级别？

隔离级别在spring中以枚举类型存在：

```
public enum Isolation {  
    DEFAULT( value: -1 ),  
    READ_UNCOMMITTED( value: 1 ),  
    READ_COMMITTED( value: 2 ),  
    REPEATABLE_READ( value: 4 ),  
    SERIALIZABLE( value: 8 );
```

Java | 复制代码

```
1 @Transactional(isolation = Isolation.READ_COMMITTED)
```

测试事务隔离级别：READ_UNCOMMITTED 和 READ_COMMITTED

怎么测试：一个service负责插入，一个service负责查询。负责插入的service要模拟延迟。

IsolationService1

Java | 复制代码

```
1 package com.powernode.bank.service.impl;
2
3 import com.powernode.bank.dao.AccountDao;
4 import com.powernode.bank.pojo.Account;
5 import jakarta.annotation.Resource;
6 import org.springframework.stereotype.Service;
7 import org.springframework.transaction.annotation.Isolation;
8 import org.springframework.transaction.annotation.Transactional;
9
10 /**
11  * @author 动力节点
12  * @version 1.0
13  * @className IsolationService1
14  * @since 1.0
15 */
16 @Service("i1")
17 public class IsolationService1 {
18
19     @Resource(name = "accountDao")
20     private AccountDao accountDao;
21
22     // 1号
23     // 负责查询
24     // 当前事务可以读取到别的事务没有提交的数据。
25     //@Transactional(isolation = Isolation.READ_UNCOMMITTED)
26     // 对方事务提交之后的数据我才能读取到。
27     @Transactional(isolation = Isolation.READ_COMMITTED)
28     public void getByActno(String actno) {
29         Account account = accountDao.selectByActno(actno);
30         System.out.println("查询到的账户信息: " + account);
31     }
32
33 }
34
```

IsolationService2

Java | 复制代码

```
1 package com.powernode.bank.service.impl;
2
3 import com.powernode.bank.dao.AccountDao;
4 import com.powernode.bank.pojo.Account;
5 import jakarta.annotation.Resource;
6 import org.springframework.stereotype.Service;
7 import org.springframework.transaction.annotation.Transactional;
8
9 /**
10 * @author 动力节点
11 * @version 1.0
12 * @className IsolationService2
13 * @since 1.0
14 */
15 @Service("i2")
16 public class IsolationService2 {
17
18     @Resource(name = "accountDao")
19     private AccountDao accountDao;
20
21     // 2号
22     // 负责insert
23     @Transactional
24     public void save(Account act) {
25         accountDao.insert(act);
26         // 睡眠一会
27         try {
28             Thread.sleep(1000 * 20);
29         } catch (InterruptedException e) {
30             e.printStackTrace();
31         }
32     }
33
34 }
```

测试程序

动力节点

动力节点

Java | 复制代码

```
1  @Test
2  public void testIsolation1(){
3      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
4      IsolationService1 i1 = applicationContext.getBean("i1", IsolationService1.class);
5      i1.getByActno("act-004");
6  }
7
8  @Test
9  public void testIsolation2(){
10     ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring.xml");
11     IsolationService2 i2 = applicationContext.getBean("i2", IsolationService2.class);
12     Account act = new Account("act-004", 1000.0);
13     i2.save(act);
14 }
```

通过执行结果可以清晰的看出隔离级别不同，执行效果不同。

事务超时

代码如下：

Java | 复制代码

```
1  @Transactional(timeout = 10)
```

以上代码表示设置事务的超时时间为10秒。

表示超过10秒如果该事务中所有的DML语句还没有执行完毕的话，最终结果会选择回滚。

默认值-1，表示没有时间限制。

这里有个坑，事务的超时时间指的是哪段时间？

在当前事务当中，最后一条DML语句执行之前的时间。如果最后一条DML语句后面有很多业务逻辑，这些业务代码执行的时间不被计入超时时间。

▼ 以下代码的超时不会被计入超时时间

Java | 复制代码

```
1 @Transactional(timeout = 10) // 设置事务超时时间为10秒。
2 public void save(Account act) {
3     accountDao.insert(act);
4     // 睡眠一会
5     try {
6         Thread.sleep(1000 * 15);
7     } catch (InterruptedException e) {
8         e.printStackTrace();
9     }
10 }
```

▼ 以下代码超时时间会被计入超时时间

Java | 复制代码

```
1 @Transactional(timeout = 10) // 设置事务超时时间为10秒。
2 public void save(Account act) {
3     // 睡眠一会
4     try {
5         Thread.sleep(1000 * 15);
6     } catch (InterruptedException e) {
7         e.printStackTrace();
8     }
9     accountDao.insert(act);
10 }
```

当然，如果想让整个方法的所有代码都计入超时时间的话，可以在方法最后一行添加一行无关紧要的DML语句。

只读事务

代码如下：

```
1 @Transactional(readOnly = true)
```

将当前事务设置为只读事务，在该事务执行过程中只允许select语句执行，delete insert update均不可执行。

该特性的作用是：启动spring的优化策略。提高select语句执行效率。

如果该事务中确实没有增删改操作，建议设置为只读事务。

设置哪些异常回滚事务

代码如下：

```
1 @Transactional(rollbackFor = RuntimeException.class)
```

表示只有发生RuntimeException异常或该异常的子类异常才回滚。

设置哪些异常不回滚事务

代码如下：

```
1 @Transactional(noRollbackFor = NullPointerException.class)
```

表示发生NullPointerException或该异常的子类异常不回滚，其他异常则回滚。

事务的全注解式开发

编写一个类来代替配置文件，代码如下：

```
1 package com.powernode.bank;
2
3 import com.alibaba.druid.pool.DruidDataSource;
4 import org.springframework.context.annotation.Bean;
5 import org.springframework.context.annotation.ComponentScan;
6 import org.springframework.context.annotation.Configuration;
7 import org.springframework.jdbc.core.JdbcTemplate;
8 import org.springframework.jdbc.datasource.DataSourceTransactionManager;
9 import org.springframework.transaction.annotation.EnableTransactionManagement;
10
11 import javax.sql.DataSource;
12
13 /**
14 * @author 动力节点
15 * @version 1.0
16 * @className Spring6Config
17 * @since 1.0
18 */
19 @Configuration
20 @ComponentScan("com.powernode.bank")
21 @EnableTransactionManagement
22 public class Spring6Config {
23
24     @Bean
25     public DataSource getDataSource(){
26         DruidDataSource dataSource = new DruidDataSource();
27         dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
28         dataSource.setUrl("jdbc:mysql://localhost:3306/spring6");
29         dataSource.setUsername("root");
30         dataSource.setPassword("root");
31         return dataSource;
32     }
33
34     @Bean(name = "jdbcTemplate")
35     public JdbcTemplate getJdbcTemplate(DataSource dataSource){
36         JdbcTemplate jdbcTemplate = new JdbcTemplate();
37         jdbcTemplate.setDataSource(dataSource);
38         return jdbcTemplate;
39     }
40
41     @Bean
42     public DataSourceTransactionManager getDataSourceTransactionManager(DataSource dataSource){
43 }
```

```
44     DataSourceTransactionManager dataSourceTransactionManager = new Da  
45     taSourceTransactionManager();  
46     dataSourceTransactionManager.setDataSource(dataSource);  
47     return dataSourceTransactionManager;  
48 }  
49 }
```

测试程序如下：

```
1  @Test  
2  public void testNoXml(){  
3      ApplicationContext applicationContext = new AnnotationConfigApplication  
4          Context(Spring6Config.class);  
5      AccountService accountService = applicationContext.getBean("accountSer  
6          vice", AccountService.class);  
7      try {  
8          accountService.transfer("act-001", "act-002", 10000);  
9          System.out.println("转账成功");  
10     } catch (Exception e) {  
11         e.printStackTrace();  
12     }  
13 }
```

执行结果：

```
Tests passed: 1 of 1 test – 1 sec 303 ms  
C:\dev\Java\jdk-17.0.4\bin\java.exe ...  
10月 23, 2022 3:49:25 下午 com.alibaba.druid.support.logging.JakartaCommonsLoggingImpl info  
信息: {dataSource-1} init  
java.lang.NullPointerException Create breakpoint : Cannot invoke "String.toString()" because "s" is null  
at com.powernode.bank.service.impl.AccountServiceImpl.transfer(AccountServiceImpl.java:41) <4 int  
at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:343)  
at org.springframework.aop.framework.ReflectiveMethodInvocation.invokeJoinpoint(ReflectiveMethod  
at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMe
```

数据库表中数据：

对象 t_act @spring6 (localhost) - 表

开始事务 文本 筛选 排序 导出

actno	balance
act-001	30000
act-002	10000

声明式事务之XML实现方式

配置步骤：

- 第一步：配置事务管理器
- 第二步：配置通知
- 第三步：配置切面

记得添加aspectj的依赖：

pom.xml

XML | 复制代码

```
1 <!--aspectj依赖-->
2 <dependency>
3   <groupId>org.springframework</groupId>
4   <artifactId>spring-aspects</artifactId>
5   <version>6.0.0-M2</version>
6 </dependency>
```

Spring配置文件如下：

记得添加aop的命名空间。

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xmlns:tx="http://www.springframework.org/schema/tx"
6   xmlns:aop="http://www.springframework.org/schema/aop"
7   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
8                               http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd
9                               http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd
10                          http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">
11
12     <context:component-scan base-package="com.powernode.bank"/>
13
14     <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
15       <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
16       <property name="url" value="jdbc:mysql://localhost:3306/spring6"/>
17       <property name="username" value="root"/>
18       <property name="password" value="root"/>
19     </bean>
20
21     <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
22       <property name="dataSource" ref="dataSource"/>
23     </bean>
24
25     <!--配置事务管理器-->
26     <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
27       <property name="dataSource" ref="dataSource"/>
28     </bean>
29
30     <!--配置通知-->
31     <tx:advice id="txAdvice" transaction-manager="txManager">
32       <tx:attributes>
33         <tx:method name="save*" propagation="REQUIRED" rollback-for="java.lang.Throwable"/>
34         <tx:method name="del*" propagation="REQUIRED" rollback-for="java.lang.Throwable"/>
35         <tx:method name="update*" propagation="REQUIRED" rollback-for="java.lang.Throwable"/>
```

```

36          <tx:method name="transfer*" propagation="REQUIRED" rollback-fo
37          r="java.lang.Throwable"/>
38      </tx:attributes>
39  </tx:advice>
40
41  <!--配置切面-->
42  <aop:config>
43      <aop:pointcut id="txPointcut" expression="execution(* com.powernod
e.bank.service..*(...))"/>
44      <!--切面 = 通知 + 切点-->
45      <aop:advisor advisor-ref="txAdvice" pointcut-ref="txPointcut"/>
46  </aop:config>
47
</beans>

```

将AccountServiceImpl类上的@Transactional注解删除。

编写测试程序：

```

1  @Test
2  public void testTransferXml(){
3      ApplicationContext applicationContext = new ClassPathXmlApplicationCon
text("spring2.xml");
4      AccountService accountService = applicationContext.getBean("accountSer
vice", AccountService.class);
5      try {
6          accountService.transfer("act-001", "act-002", 10000);
7          System.out.println("转账成功");
8      } catch (Exception e) {
9          e.printStackTrace();
10     }
11 }

```

执行结果：

```

Tests passed: 1 of 1 test - 1 sec 512 ms
C:\dev\Java\jdk-17.0.4\bin\java.exe ...
10月 23, 2022 3:29:59 下午 com.alibaba.druid.support.logging.JakartaCommonsLoggingImpl info
信息: {dataSource-1} init
java.lang.NullPointerException Create breakpoint : Cannot invoke "String.toString()" because "s" is null
+   at com.powernode.bank.service.impl.AccountServiceImpl.transfer(AccountServiceImpl.java:40) <4 in
    at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:343)
    at org.springframework.aop.framework.ReflectiveMethodInvocation.invokeJoinpoint(ReflectiveMethodInvocation.java:151)
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:178)
    at org.springframework.aop.framework.interceptor.MethodInvocationProceedingJoinPoint.proceed(MethodInvocationProceedingJoinPoint.java:85)

```

数据库表中记录：

对象 t_act @spring6 (localhost) - 表

开始事务 文本 筛选 排序 导入

actno	balance
act-001	30000
act-002	10000

通过测试可以看到配置XML已经起作用了。

十七、Spring6整合JUnit5

17.1 Spring对JUnit4的支持

准备工作：

pom.xml

XML | 复制代码

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>com.powernode</groupId>
8     <artifactId>spring6-015-junit</artifactId>
9     <version>1.0-SNAPSHOT</version>
10    <packaging>jar</packaging>
11
12    <!--仓库-->
13    <repositories>
14        <!--spring里程碑版本的仓库-->
15        <repository>
16            <id>repository.spring.milestone</id>
17            <name>Spring Milestone Repository</name>
18            <url>https://repo.spring.io/milestone</url>
19        </repository>
20    </repositories>
21
22    <dependencies>
23        <!--spring context依赖-->
24        <dependency>
25            <groupId>org.springframework</groupId>
26            <artifactId>spring-context</artifactId>
27            <version>6.0.0-M2</version>
28        </dependency>
29        <!--spring对junit的支持相关依赖-->
30        <dependency>
31            <groupId>org.springframework</groupId>
32            <artifactId>spring-test</artifactId>
33            <version>6.0.0-M2</version>
34        </dependency>
35        <!--junit4依赖-->
36        <dependency>
37            <groupId>junit</groupId>
38            <artifactId>junit</artifactId>
39            <version>4.13.2</version>
40            <scope>test</scope>
41        </dependency>
42    </dependencies>
43
44    <properties>
```

```
45      <maven.compiler.source>17</maven.compiler.source>
46      <maven.compiler.target>17</maven.compiler.target>
47  </properties>
48
49 </project>
```

动力节点

声明Bean

Java | 复制代码

```
1 package com.powernode.spring6.bean;
2
3 import org.springframework.beans.factory.annotation.Value;
4 import org.springframework.stereotype.Component;
5
6 /**
7  * @author 动力节点
8  * @version 1.0
9  * @className User
10 * @since 1.0
11 */
12 @Component
13 public class User {
14
15     @Value("张三")
16     private String name;
17
18     @Override
19     public String toString() {
20         return "User{" +
21                 "name='" + name + '\'' +
22                 '}';
23     }
24
25     public String getName() {
26         return name;
27     }
28
29     public void setName(String name) {
30         this.name = name;
31     }
32
33     public User() {
34     }
35
36     public User(String name) {
37         this.name = name;
38     }
39 }
40
```

spring.xml

XML | 复制代码

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
6   http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">
7     <context:component-scan base-package="com.powernode.spring6.bean"/>
8   </beans>
```

单元测试：

```
1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.bean.User;
4 import org.junit.Test;
5 import org.junit.runner.RunWith;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.test.context.ContextConfiguration;
8 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
9
10 /**
11  * @author 动力节点
12  * @version 1.0
13  * @className SpringJUnit4Test
14  * @since 1.0
15  */
16 @RunWith(SpringJUnit4ClassRunner.class)
17 @ContextConfiguration("classpath:spring.xml")
18 public class SpringJUnit4Test {
19
20     @Autowired
21     private User user;
22
23     @Test
24     public void testUser(){
25         System.out.println(user.getName());
26     }
27 }
28
```

执行结果如下：

```
Tests passed: 1 of 1 test - 5 ms  
C:\dev\Java\jdk-17.0.4\bin\java.exe ...  
10月 24, 2022 4:56:26 下午 org.springframework.tes  
信息: Loaded default TestExecutionListener class n  
10月 24, 2022 4:56:26 下午 org.springframework.tes  
信息: Using TestExecutionListeners: [org.springfra  
张三| 动力节点
```

Spring提供的方便主要是这几个注解：

```
@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration("classpath:spring.xml")
```

在单元测试类上使用这两个注解之后，在单元测试类中的属性上可以使用@Autowired。比较方便。

17.2 Spring对JUnit5的支持

引入JUnit5的依赖，Spring对JUnit支持的依赖还是：spring-test，如下：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>com.powernode</groupId>
8     <artifactId>spring6-015-junit</artifactId>
9     <version>1.0-SNAPSHOT</version>
10    <packaging>jar</packaging>
11
12    <!--仓库-->
13    <repositories>
14        <!--spring里程碑版本的仓库-->
15        <repository>
16            <id>repository.spring.milestone</id>
17            <name>Spring Milestone Repository</name>
18            <url>https://repo.spring.io/milestone</url>
19        </repository>
20    </repositories>
21
22    <dependencies>
23        <!--spring context依赖-->
24        <dependency>
25            <groupId>org.springframework</groupId>
26            <artifactId>spring-context</artifactId>
27            <version>6.0.0-M2</version>
28        </dependency>
29        <!--spring对junit的支持相关依赖-->
30        <dependency>
31            <groupId>org.springframework</groupId>
32            <artifactId>spring-test</artifactId>
33            <version>6.0.0-M2</version>
34        </dependency>
35        <!--junit5依赖-->
36        <dependency>
37            <groupId>org.junit.jupiter</groupId>
38            <artifactId>junit-jupiter</artifactId>
39            <version>5.9.0</version>
40            <scope>test</scope>
41        </dependency>
42    </dependencies>
43
44    <properties>
```

```
45      <maven.compiler.source>17</maven.compiler.source>
46      <maven.compiler.target>17</maven.compiler.target>
47  </properties>
48
49 </project>
```

▼ 单元测试类

Java | 复制代码

```
1 package com.powernode.spring6.test;
2
3 import com.powernode.spring6.bean.User;
4 import org.junit.jupiter.api.Test;
5 import org.junit.jupiter.api.extension.ExtendWith;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.test.context.ContextConfiguration;
8 import org.springframework.test.context.junit.jupiter.SpringExtension;
9
10
11 @ExtendWith(SpringExtension.class)
12 @ContextConfiguration("classpath:spring.xml")
13 public class SpringJUnit5Test {
14
15     @Autowired
16     private User user;
17
18     @Test
19     public void testUser(){
20         System.out.println(user.getName());
21     }
22 }
```

在JUnit5当中，可以使用Spring提供的以下两个注解，标注到单元测试类上，这样在类当中就可以使用@Autowire注解了。

@ExtendWith(SpringExtension.class)

@ContextConfiguration("classpath:spring.xml")

十八、Spring6集成MyBatis3.5

18.1 实现步骤

- 第一步：准备数据库表

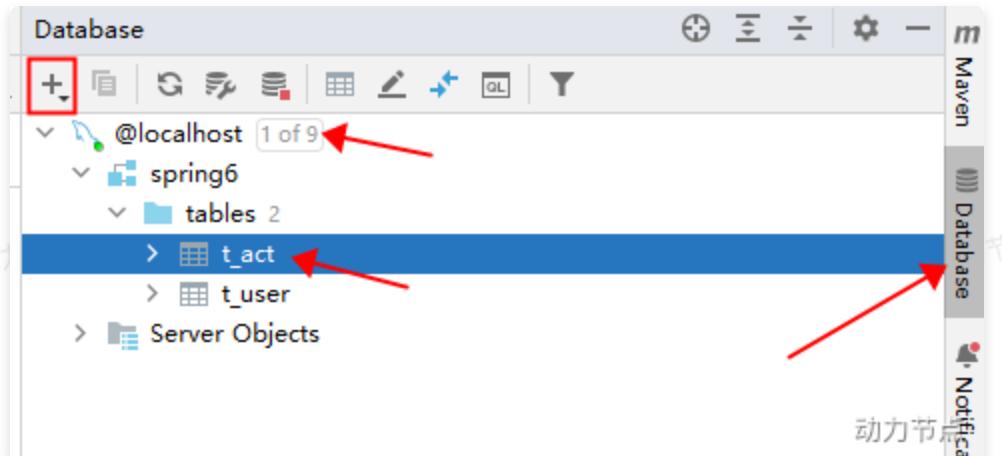
- 使用t_act表（账户表）
- 第二步：IDEA中创建一个模块，并引入依赖
 - spring-context
 - spring-jdbc
 - mysql驱动
 - mybatis
 - mybatis-spring：**mybatis提供的与spring框架集成的依赖**
 - 德鲁伊连接池
 - junit
- 第三步：基于三层架构实现，所以提前创建好所有的包
 - com.powernode.bank.mapper
 - com.powernode.bank.service
 - com.powernode.bank.service.impl
 - com.powernode.bank.pojo
- 第四步：编写pojo
 - Account，属性私有化，提供公开的setter getter和toString。
- 第五步：编写mapper接口
 - AccountMapper接口，定义方法
- 第六步：编写mapper配置文件
 - 在配置文件中配置命名空间，以及每一个方法对应的sql。
- 第七步：编写service接口和服务接口实现类
 - AccountService
 - AccountServiceImpl
- 第八步：编写jdbc.properties配置文件
 - 数据库连接池相关信息
- 第九步：编写mybatis-config.xml配置文件
 - 该文件可以没有，大部分的配置可以转移到spring配置文件中。
 - 如果遇到mybatis相关的系统级配置，还是需要这个文件。
- 第十步：编写spring.xml配置文件
 - 组件扫描
 - 引入外部的属性文件
 - 数据源

- SqlSessionFactoryBean配置
 - 注入mybatis核心配置文件路径
 - 指定别名包
 - 注入数据源
- Mapper扫描配置器
 - 指定扫描的包
- 事务管理器DataSourceTransactionManager
 - 注入数据源
- 启用事务注解
 - 注入事务管理器
- 第十一步：编写测试程序，并添加事务，进行测试

18.2 具体实现

- 第一步：准备数据库表

连接数据库的工具有很多，除了之前我们使用的navicat for mysql之外，也可以使用IDEA工具自带的 DataBase插件。可以根据下图提示自行配置：



The screenshot shows a MySQL Workbench interface with a query editor titled "t_act". The results pane displays a table with two rows:

	actno	balance
1	act-001	50000
2	act-002	0

- 第二步：IDEA中创建一个模块，并引入依赖

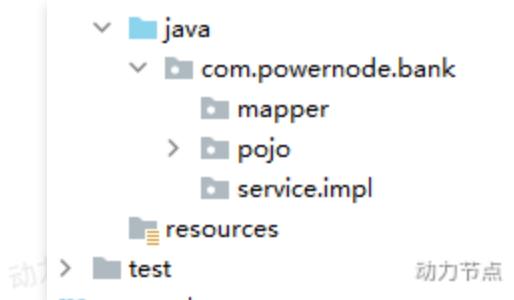
```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>com.powernode</groupId>
8     <artifactId>spring6-016-sm</artifactId>
9     <version>1.0-SNAPSHOT</version>
10    <packaging>jar</packaging>
11
12    <!--仓库-->
13    <repositories>
14        <!--spring里程碑版本的仓库-->
15        <repository>
16            <id>repository.spring.milestone</id>
17            <name>Spring Milestone Repository</name>
18            <url>https://repo.spring.io/milestone</url>
19        </repository>
20    </repositories>
21
22    <dependencies>
23        <dependency>
24            <groupId>org.springframework</groupId>
25            <artifactId>spring-context</artifactId>
26            <version>6.0.0-M2</version>
27        </dependency>
28        <dependency>
29            <groupId>org.springframework</groupId>
30            <artifactId>spring-jdbc</artifactId>
31            <version>6.0.0-M2</version>
32        </dependency>
33        <dependency>
34            <groupId>mysql</groupId>
35            <artifactId>mysql-connector-java</artifactId>
36            <version>8.0.30</version>
37        </dependency>
38        <dependency>
39            <groupId>org.mybatis</groupId>
40            <artifactId>mybatis</artifactId>
41            <version>3.5.11</version>
42        </dependency>
43        <dependency>
44            <groupId>org.mybatis</groupId>
```

```

45      <artifactId>mybatis-spring</artifactId>
46      <version>2.0.7</version>
47    </dependency>
48    <dependency>
49      <groupId>com.alibaba</groupId>
50      <artifactId>druid</artifactId>
51      <version>1.2.13</version>
52    </dependency>
53    <dependency>
54      <groupId>junit</groupId>
55      <artifactId>junit</artifactId>
56      <version>4.13.2</version>
57      <scope>test</scope>
58    </dependency>
59  </dependencies>
60
61  <properties>
62    <maven.compiler.source>17</maven.compiler.source>
63    <maven.compiler.target>17</maven.compiler.target>
64  </properties>
65
66</project>

```

- 第三步：基于三层架构实现，所以提前创建好所有的包



- 第四步：编写pojo

```
1 package com.powernode.bank.pojo;
2
3 /**
4  * @author 动力节点
5  * @version 1.0
6  * @className Account
7  * @since 1.0
8 */
9 public class Account {
10     private String actno;
11     private Double balance;
12
13     @Override
14     public String toString() {
15         return "Account{" +
16                 "actno='" + actno + '\'' +
17                 ", balance=" + balance +
18                 '}';
19     }
20
21     public Account() {
22     }
23
24     public Account(String actno, Double balance) {
25         this.actno = actno;
26         this.balance = balance;
27     }
28
29     public String getActno() {
30         return actno;
31     }
32
33     public void setActno(String actno) {
34         this.actno = actno;
35     }
36
37     public Double getBalance() {
38         return balance;
39     }
40
41     public void setBalance(Double balance) {
42         this.balance = balance;
43     }
44 }
45
```

- 第五步：编写mapper接口

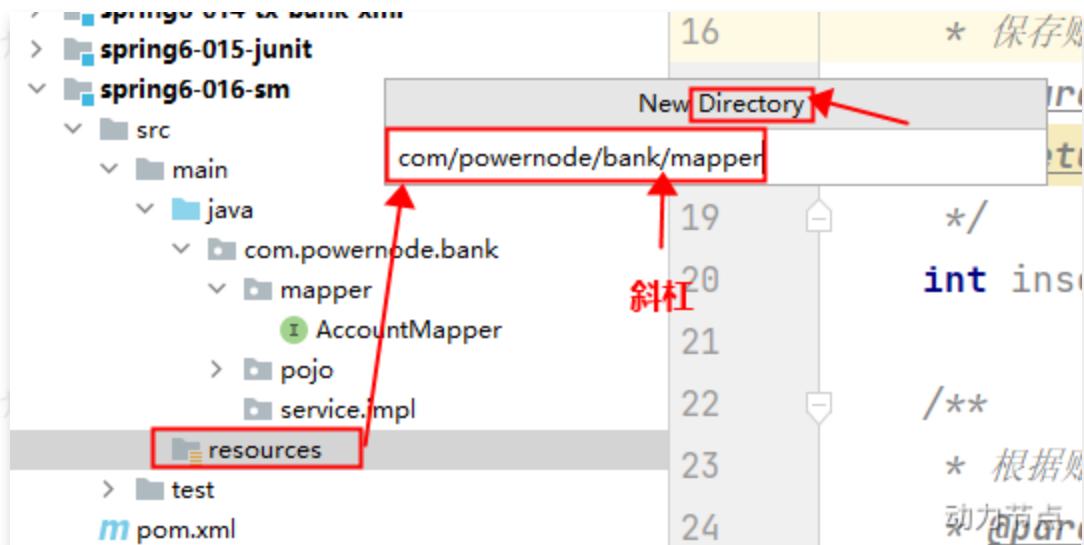
动力节点

```
1 package com.powernode.bank.mapper;
2
3 import com.powernode.bank.pojo.Account;
4
5 import java.util.List;
6
7 /**
8 * @author 动力节点
9 * @version 1.0
10 * @className AccountMapper
11 * @since 1.0
12 */
13 public interface AccountMapper {
14
15     /**
16      * 保存账户
17      * @param account
18      * @return
19      */
20     int insert(Account account);
21
22     /**
23      * 根据账号删除账户
24      * @param actno
25      * @return
26      */
27     int deleteByActno(String actno);
28
29     /**
30      * 修改账户
31      * @param account
32      * @return
33      */
34     int update(Account account);
35
36     /**
37      * 根据账号查询账户
38      * @param actno
39      * @return
40      */
41     Account selectByActno(String actno);
42
43     /**
44      * 获取所有账户
45      * @return
```

```
46     */
47     List<Account> selectAll();
48 }
49 }
```

- 第六步：编写mapper配置文件

一定要注意，按照下图提示创建这个目录。注意是斜杠不是点儿。在resources目录下新建。并且要和Mapper接口包对应上。



如果接口叫做AccountMapper， 配置文件必须是AccountMapper.xml

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <mapper namespace="com.powernode.bank.mapper.AccountMapper">
6   <insert id="insert">
7     insert into t_act values(#{actno}, #{balance})
8   </insert>
9   <delete id="deleteByActno">
10    delete from t_act where actno = #{actno}
11  </delete>
12  <update id="update">
13    update t_act set balance = #{balance} where actno = #{actno}
14  </update>
15  <select id="selectByActno" resultType="Account">
16    select * from t_act where actno = #{actno}
17  </select>
18  <select id="selectAll" resultType="Account">
19    select * from t_act
20  </select>
21 </mapper>
```

- 第七步：编写service接口和service接口实现类

注意编写的service实现类纳入IoC容器管理：

AccountService接口

Java | 复制代码

```
1 package com.powernode.bank.service;
2
3 import com.powernode.bank.pojo.Account;
4
5 import java.util.List;
6
7 /**
8 * @author 动力节点
9 * @version 1.0
10 * @className AccountService
11 * @since 1.0
12 */
13 public interface AccountService {
14     /**
15      * 开户
16      * @param act
17      * @return
18      */
19     int save(Account act);
20
21     /**
22      * 根据账号销户
23      * @param actno
24      * @return
25      */
26     int deleteByActno(String actno);
27
28     /**
29      * 修改账户
30      * @param act
31      * @return
32      */
33     int update(Account act);
34
35     /**
36      * 根据账号获取账户
37      * @param actno
38      * @return
39      */
40     Account getByActno(String actno);
41
42     /**
43      * 获取所有账户
44      * @return
45      */
```

```
46     List<Account> getAll();
47
48     /**
49      * 转账
50      * @param fromActno
51      * @param toActno
52      * @param money
53      */
54     void transfer(String fromActno, String toActno, double money);
55
56 }
```

动力节点

动力节点

动力节点

动力节点

动力节点

动力节点

动力节点

动力节点

AccountServiceImpl, 注入AccountMapper

Java | 复制代码

```
1 package com.powernode.bank.service.impl;
2
3 import com.powernode.bank.mapper.AccountMapper;
4 import com.powernode.bank.pojo.Account;
5 import com.powernode.bank.service.AccountService;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.stereotype.Service;
8
9 import java.util.List;
10
11 /**
12  * @author 动力节点
13  * @version 1.0
14  * @className AccountServiceImpl
15  * @since 1.0
16  */
17 @Transactional
18 @Service("accountService")
19 public class AccountServiceImpl implements AccountService {
20
21     @Autowired
22     private AccountMapper accountMapper;
23
24     @Override
25     public int save(Account act) {
26         return accountMapper.insert(act);
27     }
28
29     @Override
30     public int deleteByActno(String actno) {
31         return accountMapper.deleteByActno(actno);
32     }
33
34     @Override
35     public int update(Account act) {
36         return accountMapper.update(act);
37     }
38
39     @Override
40     public Account getByActno(String actno) {
41         return accountMapper.selectByActno(actno);
42     }
43
44     @Override
45     public List<Account> getAll() {
```

```

46         return accountMapper.selectAll();
47     }
48
49     @Override
50     public void transfer(String fromActno, String toActno, double money) {
51         Account fromAct = accountMapper.selectByActno(fromActno);
52         if (fromAct.getBalance() < money) {
53             throw new RuntimeException("余额不足");
54         }
55         Account toAct = accountMapper.selectByActno(toActno);
56         fromAct.setBalance(fromAct.getBalance() - money);
57         toAct.setBalance(toAct.getBalance() + money);
58         int count = accountMapper.update(fromAct);
59         count += accountMapper.update(toAct);
60         if (count != 2) {
61             throw new RuntimeException("转账失败");
62         }
63     }
64 }
65

```

- 第八步：编写jdbc.properties配置文件

放在类的根路径下

▼ jdbc.properties	Properties	复制代码
1 jdbc.driver=com.mysql.cj.jdbc.Driver 2 jdbc.url=jdbc:mysql://localhost:3306/spring6 3 jdbc.username=root 4 jdbc.password=root		

- 第九步：编写mybatis-config.xml配置文件

放在类的根路径下，只开启日志，其他配置到spring.xml中。

▼ mybatis-config.xml	XML	复制代码
1 <?xml version="1.0" encoding="UTF-8" ?> 2 <!DOCTYPE configuration 3 PUBLIC "-//mybatis.org//DTD Config 3.0//EN" 4 "http://mybatis.org/dtd/mybatis-3-config.dtd"> 5 <configuration> 6 <settings> 7 <setting name="logImpl" value="STDOUT_LOGGING"/> 8 </settings> 9 </configuration>		

- 第十步：编写spring.xml配置文件

注意：当你在spring.xml文件中直接写标签内容时，IDEA会自动给你添加命名空间

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd">
6
7   <!--组件扫描-->
8   <context:component-scan base-package="com.powernode.bank"/>
9
10  <!--外部属性配置文件-->
11  <context:property-placeholder location="jdbc.properties"/>
12
13  <!--数据源-->
14  <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
15    <property name="driverClassName" value="${jdbc.driver}"/>
16    <property name="url" value="${jdbc.url}"/>
17    <property name="username" value="${jdbc.username}"/>
18    <property name="password" value="${jdbc.password}"/>
19  </bean>
20
21  <!--SqlSessionFactoryBean-->
22  <bean class="org.mybatis.spring.SqlSessionFactoryBean">
23    <!--mybatis核心配置文件路径-->
24    <property name="configLocation" value="mybatis-config.xml"/>
25    <!--注入数据源-->
26    <property name="dataSource" ref="dataSource"/>
27    <!--起别名-->
28    <property name="typeAliasesPackage" value="com.powernode.bank.pojo"/>
29  </bean>
30
31  <!--Mapper扫描器-->
32  <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
33    <property name="basePackage" value="com.powernode.bank.mapper"/>
34  </bean>
35
36  <!--事务管理器-->
37  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
38    <property name="dataSource" ref="dataSource"/>
```

```
39     </bean>
40
41     <!--开启事务注解-->
42     <tx:annotation-driven transaction-manager="txManager"/>
43
44 </beans>
```

- 第十一步：编写测试程序，并添加事务，进行测试

Java | 复制代码

```
1 package com.powernode.spring6.test;
2
3 import com.powernode.bank.service.AccountService;
4 import org.junit.Test;
5 import org.springframework.context.ApplicationContext;
6 import org.springframework.context.support.ClassPathXmlApplicationContext;
7
8 /**
9  * @author 动力节点
10 * @version 1.0
11 * @className SMTTest
12 * @since 1.0
13 */
14 public class SMTTest {
15
16     @Test
17     public void testSM(){
18         ApplicationContext applicationContext = new ClassPathXmlApplication
19         context("spring.xml");
20         AccountService accountService = applicationContext.getBean("accoun
21         tService", AccountService.class);
22         try {
23             accountService.transfer("act-001", "act-002", 10000.0);
24             System.out.println("转账成功");
25         } catch (Exception e) {
26             e.printStackTrace();
27             System.out.println("转账失败");
28         }
29     }
}
```

最后大家别忘了测试事务！！！！

18.3 spring配置文件的import

spring配置文件有多个，并且可以在spring的核心配置文件中使用import进行引入，我们可以将组件扫描单独定义到一个配置文件中，如下：

```
▼ common.xml XML 复制代码

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd">
6
7   <!--组件扫描-->
8   <context:component-scan base-package="com.powernode.bank"/>
9
10 </beans>
```

然后在核心配置文件中引入：

```
▼ spring.xml XML 复制代码

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"
5   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd">
6
7   <!--引入其他的spring配置文件-->
8   <import resource="common.xml"/>
9
10 </beans>
```

注意：在实际开发中，service单独配置到一个文件中，dao单独配置到一个文件中，然后在核心配置文件中引入，养成好习惯。

十九、Spring中的八大模式

19.1 简单工厂模式

BeanFactory的getBean()方法，通过唯一标识来获取Bean对象。是典型的简单工厂模式（静态工厂模式）；

19.2 工厂方法模式

FactoryBean是典型的工厂方法模式。在配置文件中通过factory-method属性来指定工厂方法，该方法是一个实例方法。

19.3 单例模式

Spring用的是双重判断加锁的单例模式。请看下面代码，我们之前讲解Bean的循环依赖的时候见过：

```
8 usages
@Nullable
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    // Quick check for existing instance without full singleton lock
    Object singletonObject = this.singletonObjects.get(beanName);
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
        singletonObject = this.earlySingletonObjects.get(beanName);
        if (singletonObject == null && allowEarlyReference) {
            synchronized (this.singletonObjects) {
                // Consistent creation of early reference within full singleton lock
                singletonObject = this.singletonObjects.get(beanName);
                if (singletonObject == null) {
                    singletonObject = this.earlySingletonObjects.get(beanName);
                    if (singletonObject == null) {
                        ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
                        if (singletonFactory != null) {
                            singletonObject = singletonFactory.getObject();
                            this.earlySingletonObjects.put(beanName, singletonObject);
                            this.singletonFactories.remove(beanName);
                        }
                    }
                }
            }
        }
    }
    return singletonObject;
}
```

19.4 代理模式

Spring的AOP就是使用了动态代理实现的。

19.5 装饰器模式

JavaSE中的IO流是非常典型的装饰器模式。

Spring中配置DataSource的时候，这些dataSource可能是各种不同类型的，比如不同的数据库：Oracle、SQL Server、MySQL等，也可能是不同的数据源：比如apache提供的org.apache.commons.dbcp.BasicDataSource、spring提供的org.springframework.jndi.JndiObjectFactoryBean等。

这时，能否在尽可能少修改原有类代码下的情况下，做到动态切换不同的数据源？此时就可以用到装饰者模式。

Spring根据每次请求的不同，将dataSource属性设置成不同的数据源，以到达切换数据源的目的。

Spring中类名中带有：Decorator和Wrapper单词的类，都是装饰器模式。

19.6 观察者模式

定义对象间的一对多的关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动更新。Spring中观察者模式一般用在listener的实现。

Spring中的事件编程模型就是观察者模式的实现。在Spring中定义了一个ApplicationListener接口，用来监听Application的事件，Application其实就是ApplicationContext， ApplicationContext内置了几个事件，其中比较容易理解的是：ContextRefreshedEvent、ContextStartedEvent、ContextStoppedEvent、ContextClosedEvent

19.7 策略模式

策略模式是行为性模式，调用不同的方法，适应行为的变化，强调父类的调用子类的特性。

getHandler是HandlerMapping接口中的唯一方法，用于根据请求找到匹配的处理器。

比如我们自己写了AccountDao接口，然后这个接口下有不同的实现类：AccountDaoForMySQL，AccountDaoForOracle。对于service来说不需要关心底层具体的实现，只需要面向AccountDao接口调用，底层可以灵活切换实现，这就是策略模式。

19.8 模板方法模式

Spring中的JdbcTemplate类就是一个模板类。它就是一个模板方法设计模式的体现。在模板类的模板方法execute中编写核心算法，具体的实现步骤在子类中完成。