SSM 框架整合教程：二、Spring——尚硅谷学习笔记 2022 年

二、Spring

1、Spring 简介

1.1、Spring 概述

官网地址：https://spring.io/

> Spring 是最受欢迎的企业级 Java 应用程序开发框架，数以百万的来自世界各地的开发人员使用。
>
> Spring 框架来创建性能好、易于测试、可重用的代码。
>
> Spring 框架是一个开源的 Java 平台，它最初是由 Rod Johnson 编写的，并且于 2003 年 6 月首次在 Apache 2.0 许可下发布。
>
> Spring 是轻量级的框架，其基础版本只有 2 MB 左右的大小。

> Spring 框架的核心特性是可以用于开发任何 Java 应用程序，但是在 Java EE 平台上构建 web 应用程序是需要扩展的。 Spring 框架的目标是使 J2EE 开发变得更容易使用，通过启用基于 POJO 编程模型来促进良好的编程实践。

1.2、Spring 家族

项目列表：https://spring.io/projects

1.3、Spring Framework

Spring 基础框架，可以视为 Spring 基础设施，基本上任何其他 Spring 项目都是以 Spring Framework 为基础的。

1.3.1、Spring Framework 特性

- 非侵入式：使用 Spring Framework 开发应用程序时，Spring 对应用程序本身的结构影响非常小。对领域模型可以做到零污染；对功能性组件也只需要使用几个简单的注解进行标记，完全不会破坏原有结构，反而能将组件结构进一步简化。这就使得基于 Spring Framework 开发应用程序时结构清晰、简洁优雅。
- 控制反转：IOC——Inversion of Control，翻转资源获取方向。把自己创建资源、向环境索取资源变成环境将资源准备好，我们享受资源注入。
- 面向切面编程：AOP——Aspect Oriented Programming，在不修改源代码的基础上增强代码功能。
- 容器：Spring IOC 是一个容器，因为它包含并且管理组件对象的生命周期。组件享受到了容器化的管理，替程序员屏蔽了组件创建过程中的大量细节，极大的降低了使用门槛，大幅度提高了开发效率。
- 组件化：Spring 实现了使用简单的组件配置组合成一个复杂的应用。在 Spring 中可以使用 XML 和 Java 注解组合这些对象。这使得我们可以基于一个个功能明确、边界清晰的组件有条不紊的搭建超大型复杂应用系统。
- 声明式：很多以前需要编写代码才能实现的功能，现在只需要声明需求即可由框架代为实现。
- 一站式：在 IOC 和 AOP 的基础上可以整合各种企业应用的开源框架和优秀的第三方类库。而且 Spring 旗下的项目已经覆盖了广泛领域，很多方面的功能性需求可以在 Spring Framework 的基础上全部使用 Spring 来实现。

1.3.2、Spring Framework 五大功能模块

功能模块	功能介绍
Core Container	核心容器，在 Spring 环境下使用任何功能都必须基于 IOC 容器。
AOP&Aspects	面向切面编程。
Testing	提供了对 junit 或 TestNG 测试框架的整合。
Data Access/ Integration	提供了对数据访问/集成的功能。
Spring MVC	提供了面向 Web 应用程序的集成功能。

2、IOC

2.1、IOC 容器

2.1.1、IOC 思想

IOC：Inversion of Control，翻译过来是**反转控制**。

2.1.1.1 ① 获取资源的传统方式

自己做饭：买菜、洗菜、择菜、改刀、炒菜，全过程参与，费时费力，必须清楚了解资源创建整个过程中的全部细节且熟练掌握。

在应用程序中的组件需要获取资源时，传统的方式是组件**主动**的从容器中获取所需要的资源，在这样的模式下开发人员往往需要知道在具体容器中特定资源的获取方式，增加了学习成本，同时降低了开发效率。

2.1.1.2 ② 反转控制方式获取资源

点外卖：下单、等、吃，省时省力，不必关心资源创建过程的所有细节。

反转控制的思想完全颠覆了应用程序组件获取资源的传统方式：反转了资源的获取方向——改由容器主动的将资源推送给需要的组件，开发人员不需要知道容器是如何创建资源对象的，只需要提供接收资源的方式即可，极大的降低了学习成本，提高了开发的效率。这种行为也称为查找的

被动形式。

2.1.1.3 ③ DI

DI：Dependency Injection，翻译过来是**依赖注入**。

DI 是 IOC 的另一种表述方式：即组件以一些预先定义好的方式（例如：setter 方法）接受来自于容器的资源注入。相对于 IOC 而言，这种表述更直接。

所以结论是：IOC 就是一种反转控制的思想，而 DI 是对 IOC 的一种具体实现。

2.1.2、IOC 容器在 Spring 中的实现

Spring 的 IOC 容器就是 IOC 思想的一个落地的产品实现。IOC 容器中管理的组件也叫做 bean。在创建 bean 之前，首先需要创建 IOC 容器。Spring 提供了 IOC 容器的两种实现方式。

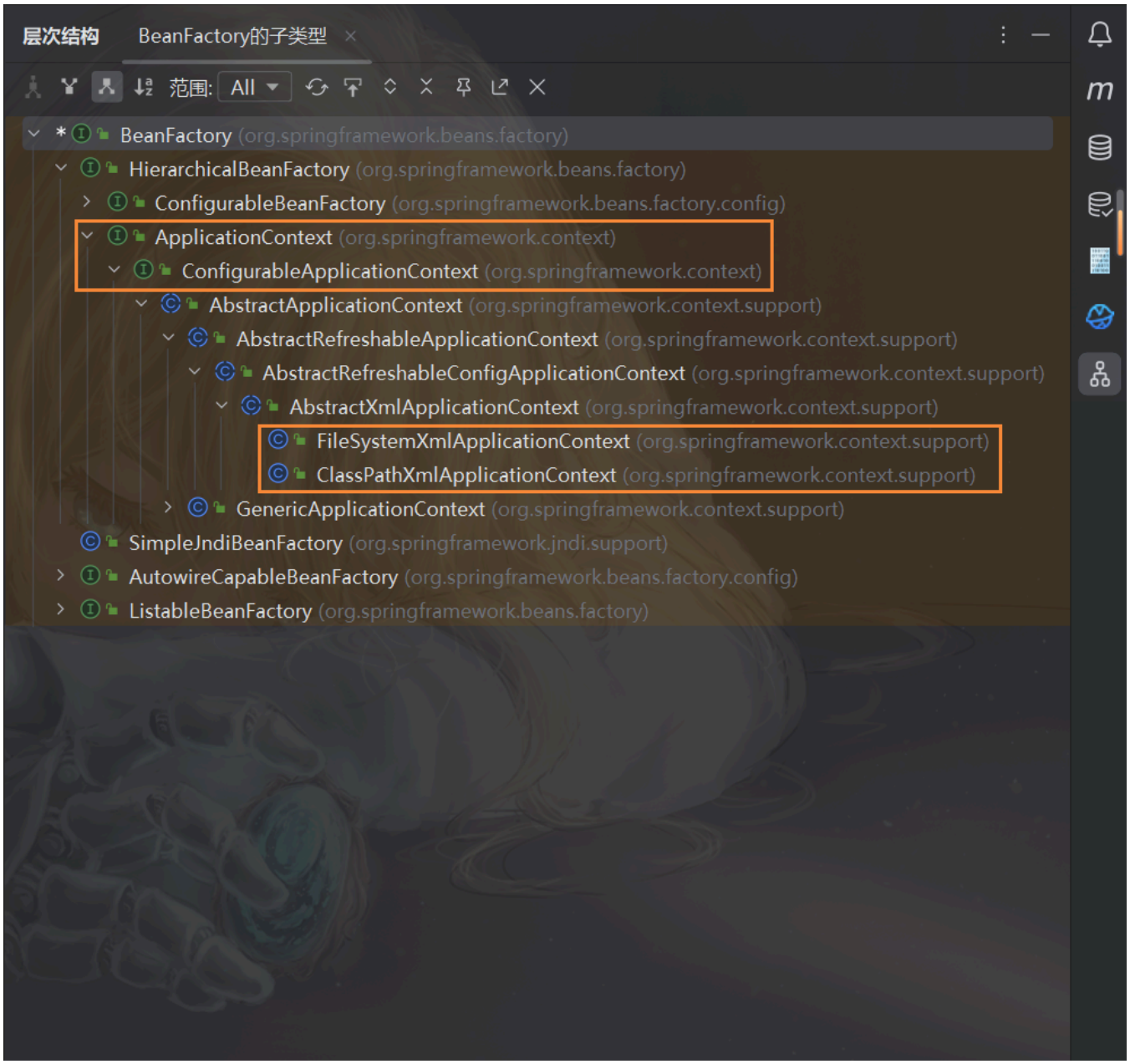
2.1.2.1 ① BeanFactory

这是 IOC 容器的基本实现，是 Spring 内部使用的接口。面向 Spring 本身，不提供给开发人员使用。

2.1.2.2 ② ApplicationContext

BeanFactory 的子接口，提供了更多高级特性。面向 Spring 的使用者，几乎所有场合都使用 ApplicationContext 而不是底层的 BeanFactory。

2.1.2.3 ③ ApplicationContext 的主要实现类



类型名	简介
ClassPathXmlApplicationContext	通过读取类路径下的 XML 格式的配置文件创建 IOC 容器对象。
FileSystemXmlApplicationContext	通过文件系统路径读取 XML 格式的配置文件创建 IOC 容器对象。
ConfigurableApplicationContext	ApplicationContext 的子接口，包含一些扩展方法

类型名	简介
	refresh() 和 close()，让 ApplicationContext 具有启动、关闭和刷新上下文的能力。
WebApplicationContext	专门为 Web 应用准备，基于 Web 环境创建 IOC 容器对象，并将对象引入存入 ServletContext 域中。

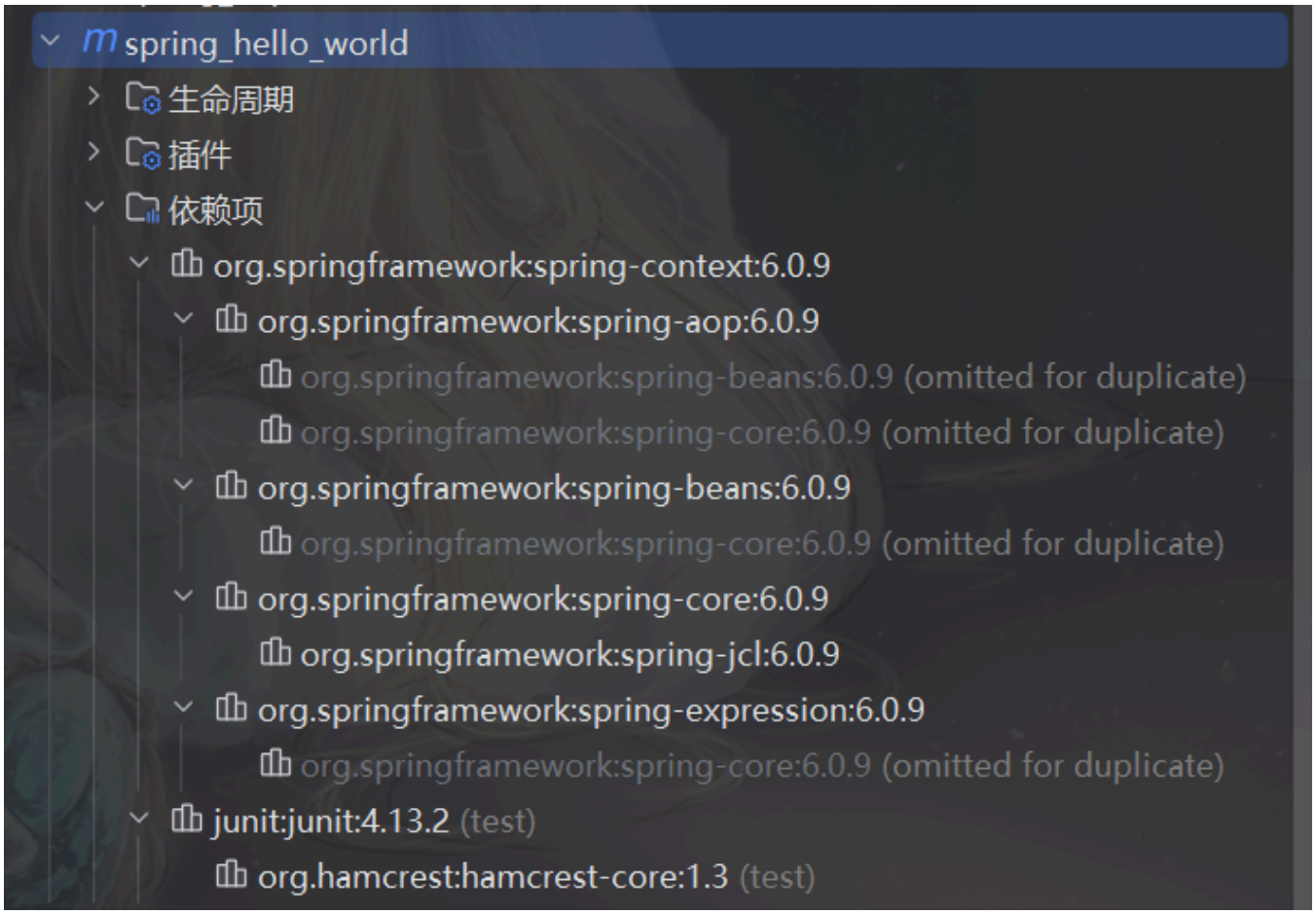
2.2、基于 XML 管理 bean

2.2.1、实验一：入门案例

2.2.1.1 ① 创建 Maven Module

2.2.1.2 ② 引入依赖

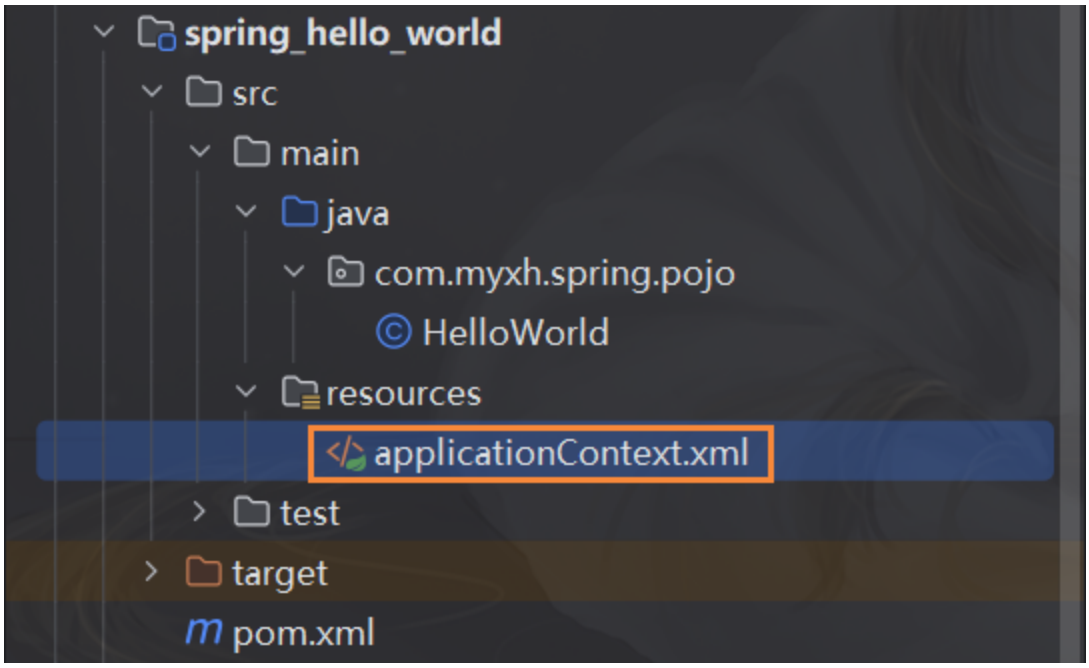
```
<dependencies>
    <!-- 基于 Maven 依赖传递性，导入 spring-context 依赖即可导入当前所需所有 jar 包 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>6.0.9</version>
    </dependency>

    <!-- junit 测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```



2.2.1.3 ③ 创建类 HelloWorld

```java
package com.myxh.spring.pojo;

/**
 * @author MYXH
 * @date 2023/8/22
 */
public class HelloWorld
{
    public void sayHello()
    {
        System.out.println("Hello Spring World!");
    }
}
```

2.2.1.4 ④ 创建 Spring 的配置文件



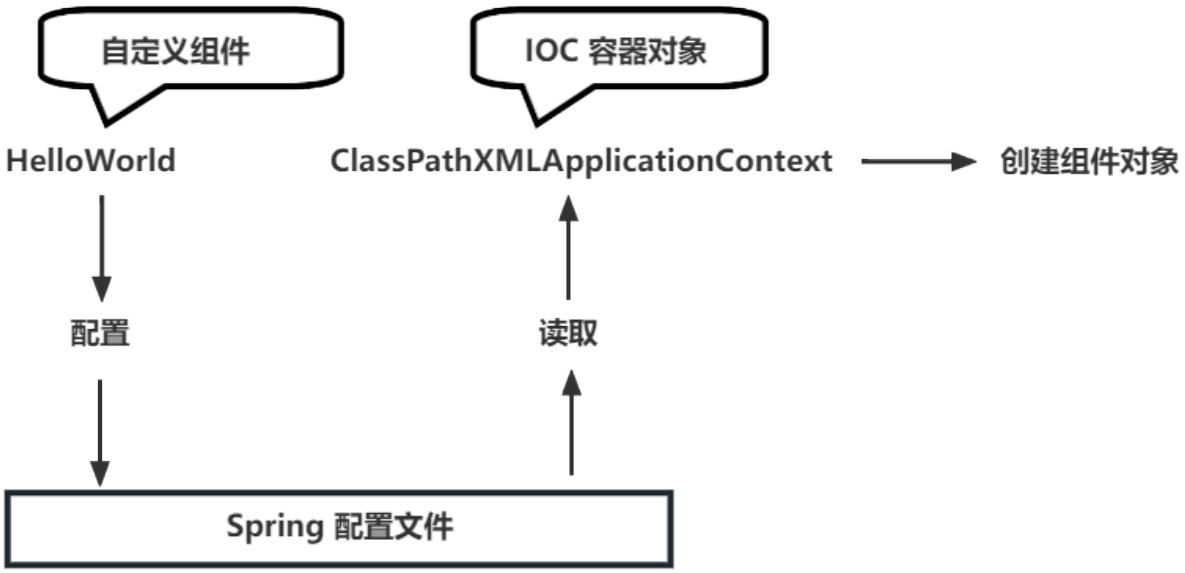
2.2.1.5 ⑤ 在 Spring 的配置文件中配置 bean

```
<?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.
    <!--
    bean: 配置一个 bean 对象，将对象交给 IOC 容器管理
    属性:
    id: bean 的唯一标识，不能重复
    class: 设置 bean 对象所对应的类型
     -->
    <bean id="helloWorld" class="com.myxh.spring.pojo.HelloWorld"/>
</beans>
```

2.2.1.6 ⑥ 创建测试类测试

```java
package com.myxh.spring.test;

import com.myxh.spring.pojo.HelloWorld;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * @author MYXH
 * @date 2023/8/22
 */
public class HelloWorldTest
{
    @Test
    public void testHelloWorld()
    {
        // 获取 IOC 容器
        ApplicationContext ioc = new ClassPathXmlApplicationContext("applicationContext.xml");

        // 获取 IOC 容器中的 bean
        HelloWorld helloWorld = (HelloWorld) ioc.getBean("helloWorld");
        helloWorld.sayHello();
    }
}
```

2.2.1.7 ⑦ 思路



2.2.1.8 ⑧ 注意

Spring 底层默认通过反射技术调用组件类的无参构造器来创建组件对象，这一点需要注意。如果在需要无参构造器时，没有无参构造器，则会抛出下面的异常：

> org.springframework.beans.factory.BeanCreationException: Error creating bean with name ‘helloworld’ defined in class path resource [applicationContext.xml]: Instantiation of bean failed; nested exception is org.springframework.beans.BeanInstantiationException: Failed to instantiate [com.myxh.spring.bean.HelloWorld]: No default constructor found; nested exception is java.lang.NoSuchMethodException: com.myxh.spring.pojo.HelloWorld.<init>()

2.2.2、实验二：获取 bean

2.2.2.1 ① 方式一：根据 id 获取

由于 id 属性指定了 bean 的唯一标识，所以根据 bean 标签的 id 属性可以精确获取到一个组件对象。

上个实验中我们使用的就是这种方式。

```java
@Test
public void testIOC()
{
    // 获取 IOC 容器
    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-ioc.xml");

    // 获取 IOC 容器中的 bean
    Student student1 = (Student) ioc.getBean("student1");
    System.out.println("student1 = " + student1);
}
```

2.2.2.2 ② 方式二：根据类型获取

```java
@Test
public void testIOC()
{
    // 获取 IOC 容器
    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-ioc.xml");

    // 获取 IOC 容器中的 bean
    Student student = ioc.getBean(Student.class);
    System.out.println("student = " + student);
}
```

2.2.2.3 ③ 方式三：根据 id 和类型

```
/**
 * 获取 bean 的三种方式：
 * 1、根据 bean 的 id 获取
 * 2、根据 bean 的类型获取
 * 注意：根据类型获 bean 时，要求 IOC 容器中有且只有一个类型匹配的 bean
 * 若没有任何一个类型匹配的 bean，此时抛出异常：NoSuchBeanDefinitionException
 * 若有多个类型匹配的 bean，此时抛出异常：NoUniqueBeanDefinitionException
 * 3、根据 bean 的 id 和类型获取
 *
 * 结论：
 * 根据类型来获取 bean 时，在满足 bean 唯一性的前提下
 * 其实只是看：对象 instanceof 指定的类型的返回结果
 * 只要返回的是 true 就可以认定为和类型匹配，能够获取到
 * 即通过 bean 的类型、bean 所继承的类的类型、bean 所实现的接口的类型都可以获取 bean
 */
@Test
public void testIOC()
{
    // 获取 IOC 容器
    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-ioc.xml");

    // 获取 IOC 容器中的 bean
    Student student1 = ioc.getBean("student1", Student.class);
    System.out.println("student1 = " + student1);
}
```

2.2.2.4 ④ 注意

当根据类型获取 bean 时，要求 IOC 容器中指定类型的 bean 有且只能有一个。

当 IOC 容器中一共配置了两个：

```
<bean id="student1" class="com.myxh.spring.pojo.Student"/>

<bean id="student2" class="com.myxh.spring.pojo.Student"/>
```

根据类型获取时会抛出异常：

> org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type ‘com.myxh.spring.pojo.Student’ available: expected single matching bean but found 2: student1,student2

2.2.2.5 ⑤ 扩展

如果组件类实现了接口，根据接口类型可以获取 bean 吗？

> 可以，前提是 bean 唯一。

如果一个接口有多个实现类，这些实现类都配置了 bean，根据接口类型可以获取 bean 吗？

> 不行，因为 bean 不唯一。

2.2.2.6 ⑥ 结论

根据类型来获取 bean 时，在满足 bean 唯一性的前提下，其实只是看：『对象 **instanceof** 指定的类型』的返回结果，只要返回的是 true 就可以认定为和类型匹配，能够获取到。

2.2.3、实验三：依赖注入之 setter 注入

2.2.3.1 ① 创建学生类 Student

```java
package com.myxh.spring.pojo;

import java.util.Arrays;
import java.util.Map;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public class Student implements Person
{
    private Integer studentId;
    private String studentName;
    private Integer age;
    private String gender;

    public Student()
    {

    }

    public Integer getStudentId()
    {
        return studentId;
    }

    public void setStudentId(Integer studentId)
    {
        this.studentId = studentId;
    }

    public String getStudentName()
    {
        return studentName;
    }

    public void setStudentName(String studentName)
    {
        this.studentName = studentName;
    }

    public Integer getAge()
    {
        return age;
```

```
    }

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

    public String getGender()
    {
        return gender;
    }

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

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

2.2.3.2 ② 配置 bean 时为属性赋值

```
<bean id="student2" class="com.myxh.spring.pojo.Student">
    <!--
    property：通过成员变量的 set 方法进行赋值
    name：设置需要值的属性名（和 set 方法有关）
    value：设置为属性所赋的值
        -->
    <property name="studentId" value="1"/>
    <property name="studentName" value="张三"/>
    <property name="age" value="20"/>
    <property name="gender" value="男"/>
</bean>
```

2.2.3.3 ③ 测试

```
@Test
public void testDI()
{
    // 获取 IOC 容器
    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-ioc.xml");

    // 获取 IOC 容器中的 bean
    Student student2 = ioc.getBean("student2", Student.class);
    System.out.println("student2 = " + student2);
}
```

2.2.4、实验四：依赖注入之构造器注入

2.2.4.1 ① 在 Student 类中添加有参构造

```
public Student(Integer studentId, String studentName, Integer age, String gender)
{
    this.studentId = studentId;
    this.studentName = studentName;
    this.age = age;
    this.gender = gender;
}
```

2.2.4.2 ② 配置 bean

```
<bean id="student3" class="com.myxh.spring.pojo.Student">
    <constructor-arg value="2"/>
    <constructor-arg value="李四"/>
    <constructor-arg value="男"/>
    <constructor-arg value="21" name="age"/>
</bean>
```

> 注意：
>
> constructor-arg 标签还有两个属性可以进一步描述构造器参数：
>
> - index 属性：指定参数所在位置的索引（从 0 开始）。
> - name 属性：指定参数名。

2.2.4.3 ③ 测试

```
@Test
public void testDI()
{
    // 获取 IOC 容器
    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-ioc.xml");

    // 获取 IOC 容器中的 bean
    Student student3 = ioc.getBean("student3", Student.class);
    System.out.println("student3 = " + student3);
}
```

2.2.5、实验五：特殊值处理

2.2.5.1 ① 字面量赋值

> 什么是字面量？
>
> int a = 10;
>
> 声明一个变量 a，初始化为 10，此时 a 就不代表字母 a 了，而是作为一个变量的名字。当我们引用 a 的时候，我们实际上拿到的值是 10。
>
> 而如果 a 是带引号的：‘a’，那么它现在不是一个变量，它就是代表 a 这个字母本身，这就是字面量。所以字面量没有引申含义，就是我们看到的这个数据本身。

```
<!-- 使用value属性给bean的属性赋值时，Spring会把value属性的值看做字面量 -->
<property name="name" value="张三"/>
```

2.2.5.2 ② null 值

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

> 注意：
>
> ```
> <property name="gender" value="null"/>
> ```

> 以上写法，为 name 所赋的值是字符串 null。

2.2.5.2 ③ xml 实体

```
<bean id="student4" class="com.myxh.spring.pojo.Student">
    <property name="studentId" value="3"/>
    <!--
        <: &lt;
        >: &gt;
     -->
    <property name="studentName" value="&lt;王五&gt;"/>
    <property name="age" value="22"/>
    <property name="gender">
        <null/>
    </property>
</bean>
```

2.2.5.3 ④ CDATA 节

```
<bean id="student4" class="com.myxh.spring.pojo.Student">
    <property name="studentId" value="3"/>
    <!--
        CDATA 节其中的内容会原样解析 <![CDATA[ ... ]]>
        CDATA 节是 xml 中一个特殊的标签，因此不能写在一个属性中
        -->
    <property name="studentName">
        <value><![CDATA[<王五>]]></value>
    </property>
    <property name="age" value="22"/>
    <property name="gender">
        <null/>
    </property>
</bean>
```

2.2.6、实验六：为类类型属性赋值

2.2.6.1 ① 创建班级类 Clazz

```java
package com.myxh.spring.pojo;

import java.util.List;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public class Clazz
{
    private Integer clazzId;
    private String clazzName;

    public Clazz()
    {

    }

    public Clazz(Integer clazzId, String clazzName)
    {
        this.clazzId = clazzId;
        this.clazzName = clazzName;
    }

    public Integer getClazzId()
    {
        return clazzId;
    }

    public void setClazzId(Integer clazzId)
    {
        this.clazzId = clazzId;
    }

    public String getClazzName()
    {
        return clazzName;
    }

    public void setClazzName(String clazzName)
    {
        this.clazzName = clazzName;
    }
```

```
    @Override
    public String toString()
    {
        return "Clazz{" +
                "clazzId=" + clazzId +
                ", clazzName='" + clazzName + '\'' +
                '}';
    }
}
```

2.2.6.2 ② 修改 Student 类

在 Student 类中添加以下代码：

```
private Clazz clazz;

public Student(Integer studentId, String studentName, Integer age, String gender, Clazz clazz)
{
    this.studentId = studentId;
    this.studentName = studentName;
    this.age = age;
    this.gender = gender;
    this.clazz = clazz;
}

public Clazz getClazz()
{
    return clazz;
}

public void setClazz(Clazz clazz)
{
    this.clazz = clazz;
}

@Override
public String toString()
{
    return "Student{" +
            "studentId=" + studentId +
            ", studentName='" + studentName + '\'' +
            ", age=" + age +
            ", gender='" + gender + '\'' +
            ", clazz=" + clazz +
            '}';
}
```

2.2.6.3 ③ 方式一：引用外部已声明的 bean

配置 Clazz 类型的 bean：

```
<bean id="clazz1" class="com.myxh.spring.pojo.Clazz">
    <property name="clazzId" value="1"/>
    <property name="clazzName" value="1 班"/>
</bean>
```

为 Student 中的 clazz 属性赋值：

```
<bean id="student5" class="com.myxh.spring.pojo.Student">
    <property name="studentId" value="4"/>
    <property name="studentName" value="赵六"/>
    <property name="age" value="23"/>
    <property name="gender" value="男"/>
    <!-- ref: 引用 IOC 容器中的某个 bean 的 id -->
    <property name="clazz" ref="clazz1"/>
</bean>
```

错误演示：

```
<bean id="student5" class="com.myxh.spring.pojo.Student">
    <property name="studentId" value="4"/>
    <property name="studentName" value="赵六"/>
    <property name="age" value="23"/>
    <property name="gender" value="男"/>
    <!-- ref: 引用 IOC 容器中的某个 bean 的 id -->
    <property name="clazz" value="clazz1"/>
</bean>
```

> 如果错把 ref 属性写成了 value 属性，会抛出异常： Caused by: java.lang.IllegalStateException: Cannot convert value of type ‘java.lang.String’ to required type ‘com.myxh.spring.bean.Clazz’ for property ‘clazz’: no matching editors or conversion strategy found
>
> 意思是不能把 String 类型转换成我们要的 Clazz 类型，说明我们使用 value 属性时，Spring 只把这个属性看做一个普通的字符串，不会认为这是一个 bean 的 id，更不会根据它去找到 bean 来赋值。

2.2.6.4 ④ 方式二：内部 bean

```
<bean id="student5" class="com.myxh.spring.pojo.Student">
    <property name="studentId" value="4"/>
    <property name="studentName" value="赵六"/>
    <property name="age" value="23"/>
    <property name="gender" value="男"/>
    <property name="clazz">
        <!-- 内部 bean，只能在当前 bean 的内部使用，不能直接通过 IOC 容器获取 -->
        <bean id="clazzInner" class="com.myxh.spring.pojo.Clazz">
            <property name="clazzId" value="3"/>
            <property name="clazzName" value="3 班"/>
        </bean>
    </property>
</bean>
```

2.2.6.5 ⑤ 方式三：级联属性赋值

```
<bean id="student5" class="com.myxh.spring.pojo.Student">
    <property name="studentId" value="4"/>
    <property name="studentName" value="赵六"/>
    <property name="age" value="23"/>
    <property name="gender" value="男"/>
    <!-- ref：引用 IOC 容器中的某个 bean 的 id -->
    <property name="clazz" value="clazz1"/>

    <!-- 级联的方式，要保证提前为 Clazz 属性赋值或者实例化-->
    <property name="clazz.clazzId" value="2"/>
    <property name="clazz.clazzName" value="2 班"/>
</bean>

<bean id="clazz1" class="com.myxh.spring.pojo.Clazz">
    <property name="clazzId" value="1"/>
    <property name="clazzName" value="1 班"/>
</bean>
```

2.2.7、实验七：为数组类型属性赋值

2.2.7.1 ① 修改 Student 类

在 Student 类中添加以下代码：

```
private String[] hobby;

public Student(Integer studentId, String studentName, Integer age, String gender, String[] hobby,
{
    this.studentId = studentId;
    this.studentName = studentName;
    this.age = age;
    this.gender = gender;
    this.hobby = hobby;
    this.clazz = clazz;
}

public String[] gethobby()
{
    return hobby;
}
public void sethobby(String[] hobby)
{
    this.hobby = hobby;
}

@Override
public String toString()
{
    return "Student{" +
            "studentId=" + studentId +
            ", studentName='" + studentName + '\'' +
            ", age=" + age +
            ", gender='" + gender + '\'' +
            ", hobby=" + Arrays.toString(hobby) +
            ", clazz=" + clazz +
            '}';
}
```

2.2.7.2 ② 配置 bean

```
<bean id="student5" class="com.myxh.spring.pojo.Student">
    <property name="studentId" value="4"/>
    <property name="studentName" value="赵六"/>
    <property name="age" value="23"/>
    <property name="gender" value="男"/>
    <!-- ref: 引用 IOC 容器中的某个 bean 的 id -->
    <!-- <property name="clazz" ref="clazz1"/> -->

    <!-- 级联的方式，要保证提前为 Clazz 属性赋值或者实例化-->
    <!--
    <property name="clazz.clazzId" value="2"/>
    <property name="clazz.clazzName" value="2 班"/>
        -->

    <property name="clazz">
        <!-- 内部 bean，只能在当前 bean 的内部使用，不能直接通过 IOC 容器获取 -->
        <bean id="clazzInner" class="com.myxh.spring.pojo.Clazz">
            <property name="clazzId" value="3"/>
            <property name="clazzName" value="3 班"/>
        </bean>
    </property>

    <property name="hobby">
        <array>
            <value>游戏</value>
            <value>动漫</value>
            <value>音乐</value>
        </array>
    </property>
</bean>
```

2.2.8、实验八：为集合类型属性赋值

2.2.8.1 ① 为 List 集合类型属性赋值

在 Clazz 类中添加以下代码：

```
private List<Student> students;

public Clazz(Integer clazzId, String clazzName, List<Student> students)
{
    this.clazzId = clazzId;
    this.clazzName = clazzName;
    this.students = students;
}

public List<Student> getStudents()
{
    return students;
}

public void setStudents(List<Student> students)
{
    this.students = students;
}

@Override
public String toString()
{
    return "Clazz{" +
            "clazzId=" + clazzId +
            ", clazzName='" + clazzName + '\'' +
            ", students=" + students +
            '}';
}
```

配置 bean：

```
<bean id="clazz1" class="com.myxh.spring.pojo.Clazz">
    <property name="clazzId" value="1"/>
    <property name="clazzName" value="1 班"/>

    <property name="students">
        <list>
            <ref bean="student1"/>
            <ref bean="student2"/>
            <ref bean="student3"/>
            <ref bean="student4"/>
            <ref bean="student5"/>
        </list>
    </property>
</bean>
```

> 若为 Set 集合类型属性赋值，只需要将其中的 list 标签改为 set 标签即可。

2.2.8.2 ② 为 Map 集合类型属性赋值

创建教师类 Teacher：

```java
package com.myxh.spring.pojo;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public class Teacher
{
    private Integer teacherId;
    private String teacherName;

    public Teacher()
    {

    }

    public Teacher(Integer teacherId, String teacherName)
    {
        this.teacherId = teacherId;
        this.teacherName = teacherName;
    }

    public Integer getTeacherId()
    {
        return teacherId;
    }

    public void setTeacherId(Integer teacherId)
    {
        this.teacherId = teacherId;
    }

    public String getTeacherName()
    {
        return teacherName;
    }

    public void setTeacherName(String teacherName)
    {
        this.teacherName = teacherName;
    }

    @Override
    public String toString()
```

```
    {
        return "Teacher{" +
                "teacherId=" + teacherId +
                ", teacherName='" + teacherName + '\'' +
                '}';
    }
}
```

在 Student 类中添加以下代码：

```
private Map<String, Teacher> teacherMap;

public Student(Integer studentId, String studentName, Integer age, String gender, String[] hobby,
{
    this.studentId = studentId;
    this.studentName = studentName;
    this.age = age;
    this.gender = gender;
    this.hobby = hobby;
    this.clazz = clazz;
    this.teacherMap = teacherMap;
}

public Map<String, Teacher> getTeacherMap()
{
    return teacherMap;
}

public void setTeacherMap(Map<String, Teacher> teacherMap)
{
    this.teacherMap = teacherMap;
}

@Override
public String toString()
{
    return "Student{" +
            "studentId=" + studentId +
            ", studentName='" + studentName + '\'' +
            ", age=" + age +
            ", gender='" + gender + '\'' +
            ", hobby=" + Arrays.toString(hobby) +
            ", clazz=" + clazz +
            ", teacherMap=" + teacherMap +
            '}';
}
```

配置 bean：

```xml
<bean id="student5" class="com.myxh.spring.pojo.Student">
    <property name="studentId" value="4"/>
    <property name="studentName" value="赵六"/>
    <property name="age" value="23"/>
    <property name="gender" value="男"/>
    <!-- ref: 引用 IOC 容器中的某个 bean 的 id -->
    <!-- <property name="clazz" ref="clazz1"/> -->

    <!-- 级联的方式，要保证提前为 Clazz 属性赋值或者实例化-->
    <!--
    <property name="clazz.clazzId" value="2"/>
    <property name="clazz.clazzName" value="2 班"/>
        -->

    <property name="clazz">
        <!-- 内部 bean，只能在当前 bean 的内部使用，不能直接通过 IOC 容器获取 -->
        <bean id="clazzInner" class="com.myxh.spring.pojo.Clazz">
            <property name="clazzId" value="3"/>
            <property name="clazzName" value="3 班"/>
        </bean>
    </property>

    <property name="hobby">
        <array>
            <value>游戏</value>
            <value>动漫</value>
            <value>音乐</value>
        </array>
    </property>

    <property name="teacherMap">
        <map>
            <entry key="1" value-ref="teacher1"/>
            <entry key="2" value-ref="teacher2"/>
        </map>
    </property>
</bean>

<bean id="teacher1" class="com.myxh.spring.pojo.Teacher">
    <property name="teacherId" value="1"/>
    <property name="teacherName" value="MYXH"/>
</bean>

<bean id="teacher2" class="com.myxh.spring.pojo.Teacher">
```

```
    <property name="teacherId" value="2"/>
    <property name="teacherName" value="末影小黑xh"/>
</bean>
```

2.2.8.3 ③ 引用集合类型的 bean

```xml
<bean id="student5" class="com.myxh.spring.pojo.Student">
    <property name="studentId" value="4"/>
    <property name="studentName" value="赵六"/>
    <property name="age" value="23"/>
    <property name="gender" value="男"/>
    <!-- ref: 引用 IOC 容器中的某个 bean 的 id -->
    <!-- <property name="clazz" ref="clazz1"/> -->

    <!-- 级联的方式，要保证提前为 Clazz 属性赋值或者实例化-->
    <!--
    <property name="clazz.clazzId" value="2"/>
    <property name="clazz.clazzName" value="2 班"/>
        -->

    <property name="clazz">
        <!-- 内部 bean，只能在当前 bean 的内部使用，不能直接通过 IOC 容器获取 -->
        <bean id="clazzInner" class="com.myxh.spring.pojo.Clazz">
            <property name="clazzId" value="3"/>
            <property name="clazzName" value="3 班"/>
        </bean>
    </property>

    <property name="hobby">
        <array>
            <value>游戏</value>
            <value>动漫</value>
            <value>音乐</value>
        </array>
    </property>

    <!--
    <property name="teacherMap">
        <map>
            <entry key="1" value-ref="teacher1"/>
            <entry key="2" value-ref="teacher2"/>
        </map>
    </property>
        -->

    <property name="teacherMap" ref="teacherMap"/>
</bean>

<bean id="clazz1" class="com.myxh.spring.pojo.Clazz">
    <property name="clazzId" value="1"/>
```

```
    <property name="clazzName" value="1 班"/>

    <!--
    <property name="students">
        <list>
            <ref bean="student1"/>
            <ref bean="student2"/>
            <ref bean="student3"/>
            <ref bean="student4"/>
            <ref bean="student5"/>
        </list>
    </property>
        -->

    <property name="students" ref="studentList"/>
</bean>

<!-- 配置一个集合类型的 bean，需要使用 util 的约束 -->
<util:list id="studentList">
    <ref bean="student1"/>
    <ref bean="student2"/>
    <ref bean="student3"/>
    <ref bean="student4"/>
    <ref bean="student5"/>
</util:list>

<util:map id="teacherMap">
    <entry key="1" value-ref="teacher1"/>
    <entry key="2" value-ref="teacher2"/>
</util:map>
```

> 使用 util:list、util:map 标签必须引入相应的命名空间，可以通过 idea 的提示功能选择。

2.2.9、实验九：p 命名空间

引入 p 命名空间后，可以通过以下方式为 bean 的各个属性赋值。

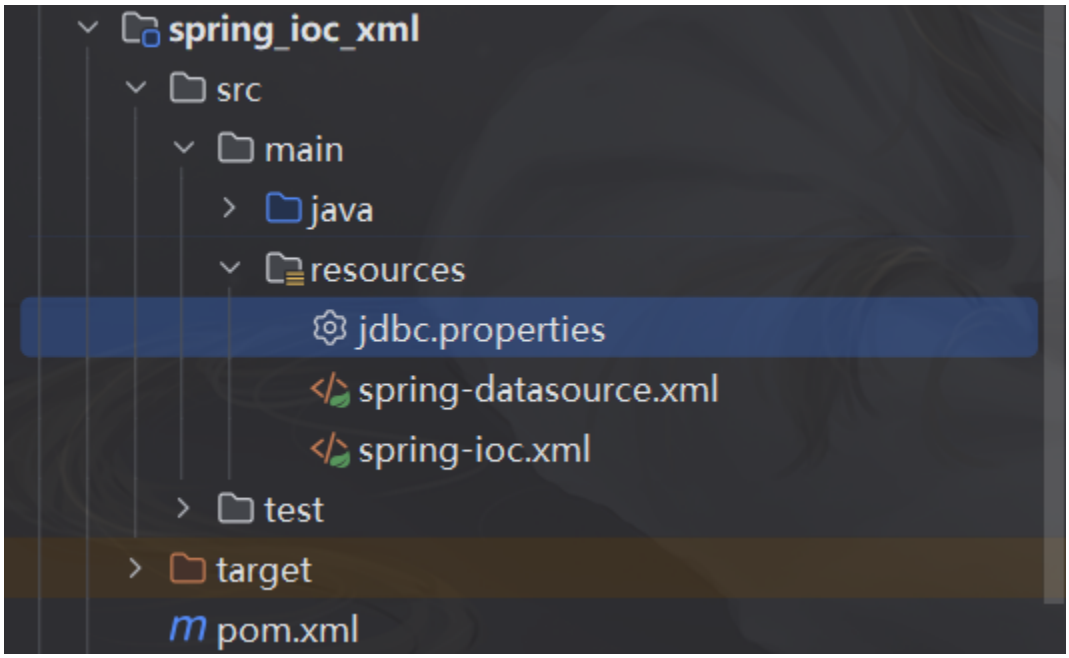
```
<bean id="student6" class="com.myxh.spring.pojo.Student"
      p:studentId="6" p:studentName="钱七" p:age="24" p:gender="男" p:teacherMap-ref="teacherMap"/
```

2.2.10、实验十：引入外部属性文件

2.2.10.1 ① 加入依赖

```
<!-- MySQL驱动 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>

<!-- 数据源 -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.16</version>
</dependency>
```

2.2.10.2 ② 创建外部属性文件




```
jdbc.driver=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
jdbc.username=MYXH
jdbc.password=520.ILY!
```

2.2.10.3 ③ 引入属性文件

```
<!-- 引入外部属性文件 jdbc.properties，之后可以通过 ${key} 的方式访问 value -->
<context:property-placeholder location="jdbc.properties"/>
```

2.2.10.4 ④ 配置 bean

```
<!-- 配置数据源 -->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="${jdbc.driver}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

2.2.10.5 ⑤ 测试

```
package com.myxh.spring.test;

import com.alibaba.druid.pool.DruidDataSource;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.sql.SQLException;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public class DataSourceTest
{
    @Test
    public void testDataSource() throws SQLException
    {
        ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-datasource.xml");
        DruidDataSource dataSource = ioc.getBean(DruidDataSource.class);
        System.out.println("dataSource.getConnection() = " + dataSource.getConnection());
        dataSource.close();
    }
}
```

2.2.11、实验十一：bean 的作用域

2.2.11.1 ① 概念

在 Spring 中可以通过配置 bean 标签的 scope 属性来指定 bean 的作用域范围，各取值含义参加下表：

取值	含义	创建对象的时机
singleton（默认）	在 IOC 容器中，这个 bean 的对象始终为单实例。	IOC 容器初始化时。
prototype	这个 bean 在 IOC 容器中有多个实例。	获取 bean 时。

如果是在 WebApplicationContext 环境下还会有另外两个作用域（但不常用）：

取值	含义
request	在一个请求范围内有效。
session	在一个会话范围内有效。

2.2.11.2 ② 创建类 User

```java
package com.myxh.spring.pojo;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public class User
{
    private Integer id;
    private String username;
    private String password;
    private Integer age;
    private String gender;
    private String email;

    public User()
    {

    }

    public User(Integer id, String username, String password, Integer age, String gender, String e
    {
        this.id = id;
        this.username = username;
        this.password = password;
        this.age = age;
        this.gender = gender;
        this.email = email;
    }

    public Integer getId()
    {
        return id;
    }

    public void setId(Integer id)
    {
        this.id = id;
    }

    public String getUsername()
    {
        return username;
    }
```

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

    public String getPassword()
    {
        return password;
    }

    public void setPassword(String password)
    {
        this.password = password;
    }

    public Integer getAge()
    {
        return age;
    }

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

    public String getGender()
    {
        return gender;
    }

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

    public String getEmail()
    {
        return email;
    }

    public void setEmail(String email)
    {
        this.email = email;
```

```
    }

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

2.2.11.3 ③ 配置 bean

```
<?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.
    <!--
    scope: 设置 bean 的作用域
    scope="singleton|prototype"
    singleton（单例）：表示获取该 bean 所对应的对象都是同一个
    prototype（多例）：表示获取该 bean 所对应的对象不是同一个
      -->
    <bean id="student" class="com.myxh.spring.pojo.Student" scope="prototype">
        <property name="studentId" value="1"/>
        <property name="studentName" value="张三"/>
        <property name="age" value="20"/>
        <property name="gender" value="男"/>
    </bean>
</beans>
```

2.2.11.4 ④ 测试

```
package com.myxh.spring.test;

import com.myxh.spring.pojo.Student;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public class ScopeTest
{
    @Test
    public void testScope()
    {
        ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-scope.xml");
        Student student1 = ioc.getBean(Student.class);
        Student student2 = ioc.getBean(Student.class);
        System.out.println("(student1 == student2) = " + (student1 == student2));
    }
}
```

2.2.12、实验十二：bean 的生命周期

2.2.12.1 ① 具体的生命周期过程

- bean 对象创建（调用无参构造器）。
- 给 bean 对象设置属性。
- bean 对象初始化之前操作（由 bean 的后置处理器负责）。
- bean 对象初始化（需在配置 bean 时指定初始化方法）。
- bean 对象初始化之后操作（由 bean 的后置处理器负责）。
- bean 对象就绪可以使用。
- bean 对象销毁（需在配置 bean 时指定销毁方法）。
- IOC 容器关闭。

2.2.12.2 ② 修改类 User

```java
package com.myxh.spring.pojo;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public class User
{
    private Integer id;
    private String username;
    private String password;
    private Integer age;
    private String gender;
    private String email;

    public User()
    {
        System.out.println("生命周期 1: 实例化");
    }

    public User(Integer id, String username, String password, Integer age, String gender, String e
    {
        this.id = id;
        this.username = username;
        this.password = password;
        this.age = age;
        this.gender = gender;
        this.email = email;
    }

    public Integer getId()
    {
        return id;
    }

    public void setId(Integer id)
    {
        System.out.println("生命周期 2: 依赖注入");
        this.id = id;
    }

    public String getUsername()
    {
        return username;
```

```
    }

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

    public String getPassword()
    {
        return password;
    }

    public void setPassword(String password)
    {
        this.password = password;
    }

    public Integer getAge()
    {
        return age;
    }

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

    public String getGender()
    {
        return gender;
    }

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

    public String getEmail()
    {
        return email;
    }

    public void setEmail(String email)
    {
```

```
        this.email = email;
    }

    @Override
    public String toString()
    {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", password='" + password + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\'' +
                ", email='" + email + '\'' +
                '}';
    }

    public void initMethod()
    {
        System.out.println("生命周期 3: 初始化");
    }

    public void destroyMethod()
    {
        System.out.println("生命周期 4: 销毁");
    }
}
```

> 注意其中的 initMethod()和 destroyMethod()，可以通过配置 bean 指定为初始化和销毁的方法。

2.2.12.3 ③ 配置 bean

```xml
<?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.
    <bean id="user" class="com.myxh.spring.pojo.User" init-method="initMethod" destroy-method="des
        <property name="id" value="1"/>
        <property name="username" value="MYXH"/>
        <property name="password" value="520.ILY!"/>
        <property name="age" value="21"/>
        <property name="gender" value="男"/>
        <property name="email" value="1735350920@qq.com"/>
    </bean>
</bean>
```

2.2.12.4 ④ 测试

```java
package com.myxh.spring.test;

import com.myxh.spring.pojo.User;
import org.junit.Test;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public class LifeCycleTest
{
    /**
     * bean 的生命周期:
     * 1、实例化
     * 2、依赖注入
     * 3、后置处理器的 postProcessBeforeInitialization
     * 4、初始化，需要通过 bean 的 init-method 属性指定初始化的方法
     * 5、后置处理器的 postProcessAfterInitialization
     * 6、IOC 容器关闭时销毁，需要通过 bean 的 destroy-method 属性指定销毁的方法
     * <p>
     * bean 的后置处理器会在生命周期的初始化前后添加额外的操作
     * 需要实现 BeanPostprocessor 接口且配置到 IOC 容器中
     * 要注意的是，bean 后置处理器不是单独针对某一个 bean 生效，而是针对 IOC 容器中所有 bean 都会执行
     * <p>
     * 注意:
     * 若 bean 的作用域为单例时，生命周期的前三个步骤会在获取 IOC 器时执行
     * 若 bean 的作用域为多例时，生命周期的前三个步骤会在获取 bean 时执行
     */
    @Test
    public void testLifeCycle()
    {
        /*
          ConfigurableApplicationContext ApplicationContext 的子接口，其中扩展了刷新和关闭容器的方法
         */
        ConfigurableApplicationContext ioc = new ClassPathXmlApplicationContext("spring-lifecycle.
        User user = ioc.getBean(User.class);
        System.out.println("user = " + user);
        ioc.close();
    }
}
```

2.2.12.5 ⑤ bean 的后置处理器

bean 的后置处理器会在生命周期的初始化前后添加额外的操作，需要实现 BeanPostProcessor 接口，且配置到 IOC 容器中，需要注意的是，bean 后置处理器不是单独针对某一个 bean 生效，而是针对 IOC 容器中所有 bean 都会执行。

创建 bean 的后置处理器：

```java
package com.myxh.spring.processor;

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

/**
 * @author MYXH
 * @date 2023/8/24
 */
public class MyBeanPostProcessor implements BeanPostProcessor
{
    /**
     * 此方法在 bean 的生命周期初始化之前执行
     *
     * @param bean     新的 bean 实例
     * @param beanName bean 的名称
     * @return 要使用的bean实例，原始实例或已包装实例；
     * 如果为 null，则不会调用后续 BeanPostProcessors
     * @throws org.springframework.beans.BeansException 如果出现错误
     */
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansExcept
    {
        System.out.println("MyBeanPostProcessor -> 后置处理器 postProcessBeforeInitialization");

        return BeanPostProcessor.super.postProcessBeforeInitialization(bean, beanName);
    }

    /**
     * 此方法在 bean 的生命周期初始化之后执行
     *
     * @param bean     新的 bean 实例
     * @param beanName bean 的名称
     * @return 要使用的 bean 实例，原始实例或已包装实例；
     * 如果为 null，则不会调用后续 BeanPostProcessors
     * @throws org.springframework.beans.BeansException 如果出现错误
     */
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansExcepti
    {
        System.out.println("MyBeanPostProcessor -> 后置处理器 postProcessAfterInitialization");

        return BeanPostProcessor.super.postProcessAfterInitialization(bean, beanName);
    }
```

```
}
```

在 IOC 容器中配置后置处理器：

```
<bean id="myBeanPostProcessor" class="com.myxh.spring.processor.MyBeanPostProcessor"/>
```

2.2.13、实验十三：FactoryBean

2.2.13.1 ① 简介

FactoryBean 是 Spring 提供的一种整合第三方框架的常用机制。和普通的 bean 不同，配置一个 FactoryBean 类型的 bean，在获取 bean 的时候得到的并不是 class 属性中配置的这个类的对象，而是 getObject()方法的返回值。通过这种机制，Spring 可以帮我们把复杂组件创建的详细过程和繁琐细节都屏蔽起来，只把最简洁的使用界面展示给我们。

将来我们整合 Mybatis 时，Spring 就是通过 FactoryBean 机制来帮我们创建 SqlSessionFactory 对象的。

```java
/*
 * Copyright 2002-2020 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.beans.factory;

import org.springframework.lang.Nullable;

/**
 * Interface to be implemented by objects used within a {@link BeanFactory} which
 * are themselves factories for individual objects. If a bean implements this
 * interface, it is used as a factory for an object to expose, not directly as a
 * bean instance that will be exposed itself.
 *
 * <p><b>NB: A bean that implements this interface cannot be used as a normal bean.</b>
 * A FactoryBean is defined in a bean style, but the object exposed for bean
 * references ({@link #getObject()}) is always the object that it creates.
 *
 * <p>FactoryBeans can support singletons and prototypes, and can either create
 * objects lazily on demand or eagerly on startup. The {@link SmartFactoryBean}
 * interface allows for exposing more fine-grained behavioral metadata.
 *
 * <p>This interface is heavily used within the framework itself, for example for
 * the AOP {@link org.springframework.aop.framework.ProxyFactoryBean} or the
 * {@link org.springframework.jndi.JndiObjectFactoryBean}. It can be used for
 * custom components as well; however, this is only common for infrastructure code.
 *
 * <p><b>{@code FactoryBean} is a programmatic contract. Implementations are not
 * supposed to rely on annotation-driven injection or other reflective facilities.</b>
 * {@link #getObjectType()} {@link #getObject()} invocations may arrive early in the
 * bootstrap process, even ahead of any post-processor setup. If you need access to
 * other beans, implement {@link BeanFactoryAware} and obtain them programmatically.
```

```
 *
 * <p><b>The container is only responsible for managing the lifecycle of the FactoryBean
 * instance, not the lifecycle of the objects created by the FactoryBean.</b> Therefore,
 * a destroy method on an exposed bean object (such as {@link java.io.Closeable#close()})
 * will <i>not</i> be called automatically. Instead, a FactoryBean should implement
 * {@link DisposableBean} and delegate any such close call to the underlying object.
 *
 * <p>Finally, FactoryBean objects participate in the containing BeanFactory's
 * synchronization of bean creation. There is usually no need for internal
 * synchronization other than for purposes of lazy initialization within the
 * FactoryBean itself (or the like).
 *
 * @author Rod Johnson
 * @author Juergen Hoeller
 * @since 08.03.2003
 * @param <T> the bean type
 * @see org.springframework.beans.factory.BeanFactory
 * @see org.springframework.aop.framework.ProxyFactoryBean
 * @see org.springframework.jndi.JndiObjectFactoryBean
 */
public interface FactoryBean<T> {

    /**
     * The name of an attribute that can be
     * {@link org.springframework.core.AttributeAccessor#setAttribute set} on a
     * {@link org.springframework.beans.factory.config.BeanDefinition} so that
     * factory beans can signal their object type when it can't be deduced from
     * the factory bean class.
     * @since 5.2
     */
    String OBJECT_TYPE_ATTRIBUTE = "factoryBeanObjectType";


    /**
     * Return an instance (possibly shared or independent) of the object
     * managed by this factory.
     * <p>As with a {@link BeanFactory}, this allows support for both the
     * Singleton and Prototype design pattern.
     * <p>If this FactoryBean is not fully initialized yet at the time of
     * the call (for example because it is involved in a circular reference),
     * throw a corresponding {@link FactoryBeanNotInitializedException}.
     * <p>As of Spring 2.0, FactoryBeans are allowed to return {@code null}
     * objects. The factory will consider this as normal value to be used; it
     * will not throw a FactoryBeanNotInitializedException in this case anymore.
```

```
     * FactoryBean implementations are encouraged to throw
     * FactoryBeanNotInitializedException themselves now, as appropriate.
     * @return an instance of the bean (can be {@code null})
     * @throws Exception in case of creation errors
     * @see FactoryBeanNotInitializedException
     */
    @Nullable
    T getObject() throws Exception;

    /**
     * Return the type of object that this FactoryBean creates,
     * or {@code null} if not known in advance.
     * <p>This allows one to check for specific types of beans without
     * instantiating objects, for example on autowiring.
     * <p>In the case of implementations that are creating a singleton object,
     * this method should try to avoid singleton creation as far as possible;
     * it should rather estimate the type in advance.
     * For prototypes, returning a meaningful type here is advisable too.
     * <p>This method can be called <i>before</i> this FactoryBean has
     * been fully initialized. It must not rely on state created during
     * initialization; of course, it can still use such state if available.
     * <p><b>NOTE:</b> Autowiring will simply ignore FactoryBeans that return
     * {@code null} here. Therefore, it is highly recommended to implement
     * this method properly, using the current state of the FactoryBean.
     * @return the type of object that this FactoryBean creates,
     * or {@code null} if not known at the time of the call
     * @see ListableBeanFactory#getBeansOfType
     */
    @Nullable
    Class<?> getObjectType();

    /**
     * Is the object managed by this factory a singleton? That is,
     * will {@link #getObject()} always return the same object
     * (a reference that can be cached)?
     * <p><b>NOTE:</b> If a FactoryBean indicates to hold a singleton object,
     * the object returned from {@code getObject()} might get cached
     * by the owning BeanFactory. Hence, do not return {@code true}
     * unless the FactoryBean always exposes the same reference.
     * <p>The singleton status of the FactoryBean itself will generally
     * be provided by the owning BeanFactory; usually, it has to be
     * defined as singleton there.
     * <p><b>NOTE:</b> This method returning {@code false} does not
     * necessarily indicate that returned objects are independent instances.
```

```
     * An implementation of the extended {@link SmartFactoryBean} interface
     * may explicitly indicate independent instances through its
     * {@link SmartFactoryBean#isPrototype()} method. Plain {@link FactoryBean}
     * implementations which do not implement this extended interface are
     * simply assumed to always return independent instances if the
     * {@code isSingleton()} implementation returns {@code false}.
     * <p>The default implementation returns {@code true}, since a
     * {@code FactoryBean} typically manages a singleton instance.
     * @return whether the exposed object is a singleton
     * @see #getObject()
     * @see SmartFactoryBean#isPrototype()
     */
    default boolean isSingleton() {
        return true;
    }

}
```

2.2.13.2 ② 创建类 UserFactoryBean

```java
package com.myxh.spring.factory;

import com.myxh.spring.pojo.User;
import org.springframework.beans.factory.FactoryBean;

/**
 * @author MYXH
 * @date 2023/8/24
 * @description
 * FactoryBean 是一个接口，需要创建一个类实现该接口
 * 其中有三个方法：
 * getObject()：通过一个对象交给 IOC 容器管理
 * getObjectType()：设置所提供对象的类型
 * isSingleton()：所提供的对象是否单例
 * 当把 FactoryBean 的实现类配置为 bean 时，会将当前类中 getObject() 所返回的对象交给 IOC 容器管理
 */
public class UserFactoryBean implements FactoryBean
{
    @Override
    public Object getObject() throws Exception
    {
        return new User();
    }

    @Override
    public Class<?> getObjectType()
    {
        return User.class;
    }
}
```

2.2.13.3 ③ 配置 bean

```xml
<?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.
    <bean class="com.myxh.spring.factory.UserFactoryBean"/>
</beans>
```

2.2.13.4 ④ 测试

```java
package com.myxh.spring.test;

import com.myxh.spring.pojo.User;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public class FactoryBeanTest
{
    @Test
    public void testFactoryBean()
    {
        ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-factory.xml");
        User user = ioc.getBean(User.class);
        System.out.println("user = " + user);

    }
}
```

2.2.14、实验十四：基于 xml 的自动装配

> 自动装配：
>
> 根据指定的策略，在 IOC 容器中匹配某一个 bean，自动为指定的 bean 中所依赖的类类型或接口类型属性赋值。

2.2.14.1 ① 场景模拟

创建类 UserController

```java
package com.myxh.spring.controller;

import com.myxh.spring.pojo.User;
import com.myxh.spring.service.UserService;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public class UserController
{
    private UserService userService;

    public UserService getUserService()
    {
        return userService;
    }

    public void setUserService(UserService userService)
    {
        this.userService = userService;
    }

    /**
     * 保存用户信息
     *
     * @param user 一个用户信息
     */
    public void saveUser(User user)
    {
        userService.saveUser(user);
    }
}
```

创建接口 UserService

```java
package com.myxh.spring.service;

import com.myxh.spring.pojo.User;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public interface UserService
{
    /**
     * 保存用户信息
     *
     * @param user 一个用户信息
     */
    void saveUser(User user);
}
```

创建类 UserServiceImpl 实现接口 UserService

```java
package com.myxh.spring.service.impl;

import com.myxh.spring.dao.UserDao;
import com.myxh.spring.pojo.User;
import com.myxh.spring.service.UserService;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public class UserServiceImpl implements UserService
{
    private UserDao userDao;

    public UserDao getUserDao()
    {
        return userDao;
    }

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

    /**
     * 保存用户信息
     *
     * @param user 一个用户信息
     */
    @Override
    public void saveUser(User user)
    {
        userDao.saveUser(user);
    }
}
```

创建接口 UserDao

```java
package com.myxh.spring.dao;

import com.myxh.spring.pojo.User;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public interface UserDao
{
    /**
     * 保存用户信息
     *
     * @param user 一个用户信息
     */
    void saveUser(User user);
}
```

创建类 UserDaoImpl 实现接口 UserDao

```java
package com.myxh.spring.dao.impl;

import com.myxh.spring.dao.UserDao;
import com.myxh.spring.pojo.User;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public class UserDaoImpl implements UserDao
{
    /**
     * 保存用户信息
     *
     * @param user 一个用户信息
     */
    @Override
    public void saveUser(User user)
    {
        System.out.println("保存用户信息成功");
    }
}
```

2.2.14.2 ② 配置 bean

> 使用 bean 标签的 autowire 属性设置自动装配效果。
>
> 自动装配方式：byType
>
> byType：根据类型匹配 IOC 容器中的某个兼容类型的 bean，为属性自动赋值。
>
> 若在 IOC 中，没有任何一个兼容类型的 bean 能够为属性赋值，则该属性不装配，即值为默认值 null。
>
> 若在 IOC 中，有多个兼容类型的 bean 能够为属性赋值，则抛出异常 NoUniqueBeanDefinitionException。

```
<bean id="user" class="com.myxh.spring.pojo.User">
       <property name="id" value="1"/>
       <property name="username" value="MYXH"/>
       <property name="password" value="520.ILY!"/>
       <property name="age" value="21"/>
       <property name="gender" value="男"/>
       <property name="email" value="1735350920@qq.com"/>
   </bean>

   <!--
   <bean id="userController" class="com.myxh.spring.controller.UserController">
        <property name="userService" ref="userService"/>
   </bean>

   <bean id="userService" class="com.myxh.spring.service.impl.UserServiceImpl">
        <property name="userDao" ref="userDao"/>
   </bean>

   <bean id="userDao" class="com.myxh.spring.dao.impl.UserDaoImpl"/>
    -->

   <bean id="userController" class="com.myxh.spring.controller.UserController" autowire="byType"/

   <bean id="userService" class="com.myxh.spring.service.impl.UserServiceImpl" autowire="byType"/

   <bean id="userDao" class="com.myxh.spring.dao.impl.UserDaoImpl"/>
```

> 自动装配方式：byName

> byName：将自动装配的属性的属性名，作为 bean 的 id 在 IOC 容器中匹配相对应的 bean 进行赋值。

```
<bean id="userController" class="com.myxh.spring.controller.UserController" autowire="byName"/>

<bean id="userService" class="com.myxh.spring.service.impl.UserServiceImpl" autowire="byName"/>

<bean id="userService2" class="com.myxh.spring.service.impl.UserServiceImpl" autowire="byName"/>

<bean id="userDao" class="com.myxh.spring.dao.impl.UserDaoImpl"/>

<bean id="userDao2" class="com.myxh.spring.dao.impl.UserDaoImpl"/>
```

2.2.14.3 ③ 测试

```java
package com.myxh.spring.test;

import com.myxh.spring.controller.UserController;
import com.myxh.spring.pojo.User;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public class AutowireByXmlTest
{
    /**
     * 自动装配：
     * 根据指定的策略，在 IOC 容器中匹配某个 bean，自动为 bean 中的类类型的属性或接口类型的属性赋值
     * 可以通过 bean 标签中的 autowire 属性设置自动装配的策略
     * <p>
     * 自动装配的策略：
     * 1、no、default：表示不装配，即 bean 中的属性不会自动匹配某个 bean 为属性赋值，此时属性使用默认值
     * 2、byType：根据要赋值的属性的类型，在 IOC 容器中匹配某个 bean，为属性赋值
     * 注意：
     * ① 若通过类型没有找到任何一个类型匹配的 bean，此时不装配，属性使用默认值
     * ② 若通过类型找到了多个类型匹配的 bean，此时会抛出异常：NoUniqueBeanDefinitionException
     * 总结：当使用 byType 实现自动装配时，IOC 容器中有且只有一个类型匹配的 bean 能够为属性赋值
     * 3、byName：将要赋值的属性的属性名作为 bean 的 id 在 IOC 容器中匹配 bean，为属性赋值
     * 总结：当类型匹配 bean 有多个时，此时可以使用 byName 实现自动装配
     */
    @Test
    public void testAutowire()
    {
        ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-autowire-xml.xml");
        User user = ioc.getBean(User.class);
        System.out.println("user = " + user);
        UserController userController = ioc.getBean(UserController.class);
        System.out.println("userController = " + userController);
        userController.saveUser(user);
    }
}
```

2.3、基于注解管理 bean

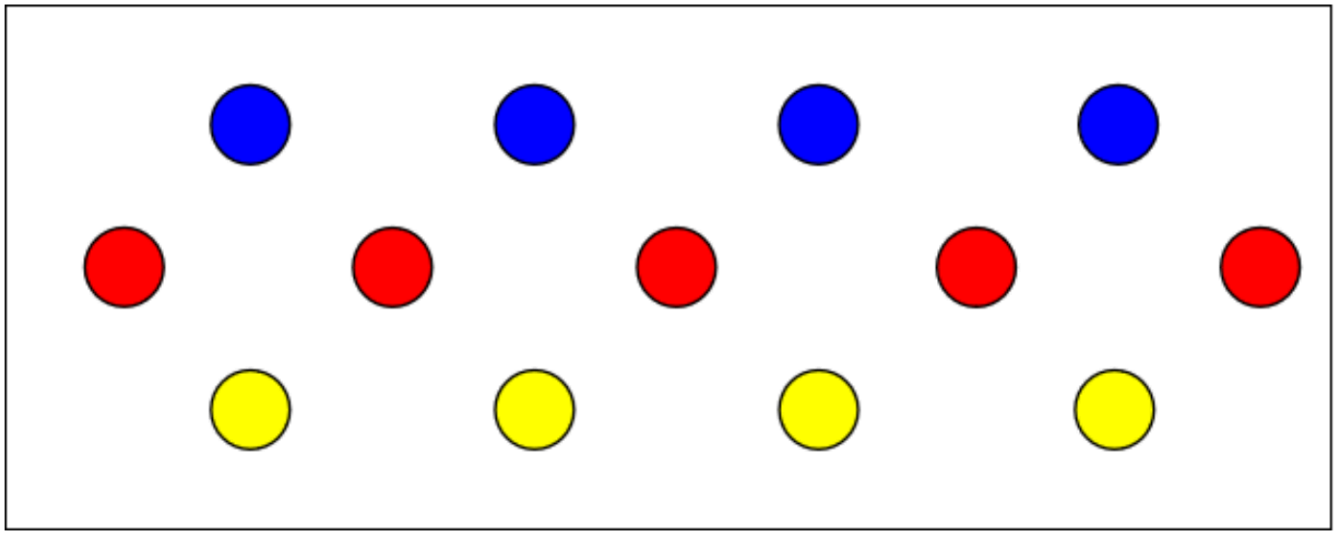
2.3.1、实验一：标记与扫描

2.3.1.1 ① 注解

和 XML 配置文件一样，注解本身并不能执行，注解本身仅仅只是做一个标记，具体的功能是框架检测到注解标记的位置，然后针对这个位置按照注解标记的功能来执行具体操作。

本质上：所有一切的操作都是 Java 代码来完成的，XML 和注解只是告诉框架中的 Java 代码如何执行。

举例：元旦联欢会要布置教室，蓝色的地方贴上元旦快乐四个字，红色的地方贴上拉花，黄色的地方贴上气球。



班长做了所有标记，同学们来完成具体工作。墙上的标记相当于我们在代码中使用的注解，后面同学们做的工作，相当于框架的具体操作。

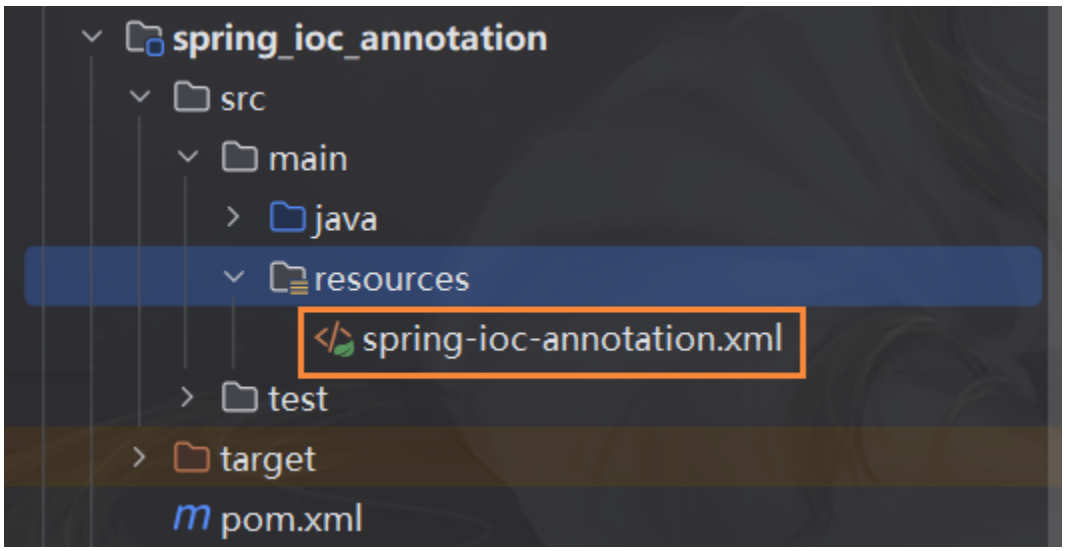
2.3.1.2 ② 扫描

Spring 为了知道程序员在哪些地方标记了什么注解，就需要通过扫描的方式，来进行检测。然后根据注解进行后续操作。

2.3.1.3 ③ 新建 Maven Module

```
<dependencies>
    <!-- 基于 Maven 依赖传递性，导入 spring-context 依赖即可导入当前所需所有 jar 包 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>6.0.9</version>
    </dependency>

    <!-- junit 测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

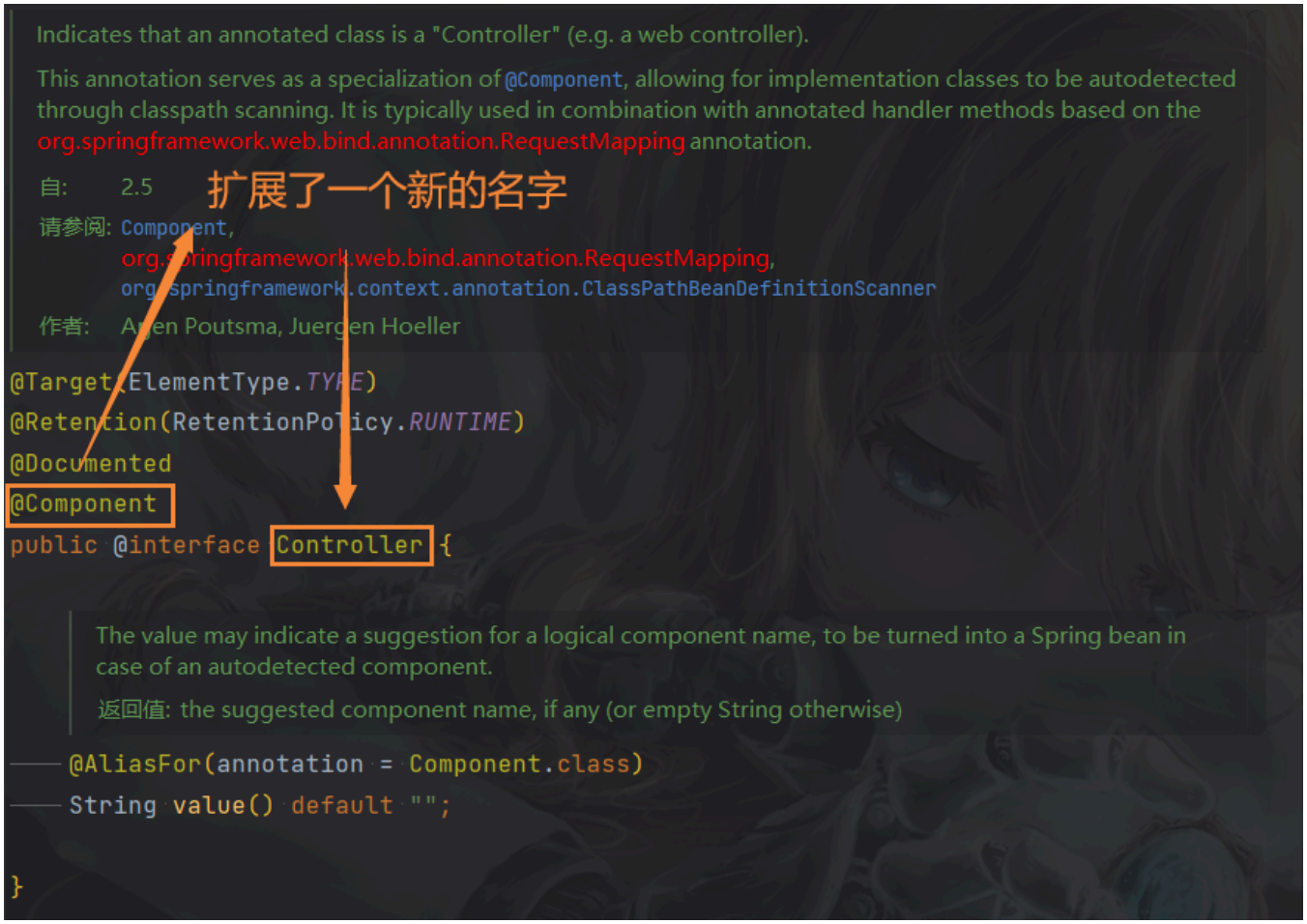
2.3.1.4 ④ 创建 Spring 配置文件



2.3.1.5 ⑤ 标识组件的常用注解

> @Component：将类标识为普通组件。
> @Controller：将类标识为控制层组件。
> @Service：将类标识为业务层组件。
> @Repository：将类标识为持久层组件。

问：以上四个注解有什么关系和区别？



通过查看源码我们得知，@Controller、@Service、@Repository 这三个注解只是在 @Component 注解的基础上起了三个新的名字。

对于 Spring 使用 IOC 容器管理这些组件来说没有区别。所以@Controller、@Service、@Repository 这三个注解只是给开发人员看的，让我们能够便于分辨组件的作用。

注意：虽然它们本质上一样，但是为了代码的可读性，为了程序结构严谨我们肯定不能随便胡乱标记。

2.3.1.6 ⑥ 创建组件

创建控制层组件

```java
package com.myxh.spring.controller;

import org.springframework.stereotype.Controller;

/**
 * @author MYXH
 * @date 2023/8/24
 */
@Controller
public class UserController
{
    public UserController()
    {

    }
}
```

创建接口 UserService

```java
package com.myxh.spring.service;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public interface UserService
{

}
```

创建业务层组件 UserServiceImpl

```java
package com.myxh.spring.service.impl;

import com.myxh.spring.service.UserService;
import org.springframework.stereotype.Service;

/**
 * @author MYXH
 * @date 2023/8/24
 */
@Service
public class UserServiceImpl implements UserService
{

}
```

创建接口 UserDao

```java
package com.myxh.spring.dao;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public interface UserDao
{

}
```

创建持久层组件 UserDaoImpl

```java
package com.myxh.spring.dao.impl;

import com.myxh.spring.dao.UserDao;
import org.springframework.stereotype.Repository;

/**
 * @author MYXH
 * @date 2023/8/24
 */
@Repository
public class UserDaoImpl implements UserDao
{

}
```

2.3.1.7 ⑦ 扫描组件

情况一：最基本的扫描方式

```xml
<!-- 扫描组件 -->
<context:component-scan base-package="com.myxh.spring"/>
```

情况二：指定要排除的组件

```xml
<!--
    context:exclude-filter：排除扫描
    type：设置排除扫描的方式
    type="annotation|assignable"
    annotation：根据注解的类型进行排除，expression 需要设置排除的注解的全类名
    assignable：根据类的类型进行排除，expression 需要设置排除的类的全类名
-->
<context:component-scan base-package="com.myxh.spring">
        <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Contr
        <context:exclude-filter type="assignable" expression="com.myxh.spring.controller.UserContr
</context:component-scan>
```

情况三：仅扫描指定组件

```
<!--
    context:include-filter: 包含扫描
    注意：需要在 context:component-scan 标签中设置 use-default-filters="false"
    use-default-filters="true"（默认），所设置的包下所有的类都需要扫描，此时可以使用排除扫描
    use-default-filters="false"，所设置的包下所有的类都不需要扫描，此时可以使用包含扫描
-->
<context:component-scan base-package="com.myxh.spring" use-default-filters="false">
    <context:include-filter type="annotation" expression="org.springframework.stereotype.Component
    <context:include-filter type="annotation" expression="org.springframework.stereotype.Controlle
    <context:include-filter type="annotation" expression="org.springframework.stereotype.Service"/
    <context:include-filter type="annotation" expression="org.springframework.stereotype.Repositor
</context:component-scan>
```

2.3.1.8 ⑧ 测试

```
@Test
public void testIoc()
{
    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-ioc-annotation.xml");
    User user = ioc.getBean("user", User.class);
    System.out.println("user = " + user);
    UserController userController = ioc.getBean("userController", UserController.class);
    System.out.println("userController = " + userController);
    UserService userService = ioc.getBean("userServiceImpl", UserService.class);
    System.out.println("userService = " + userService);
    UserDao userDao = ioc.getBean("userDaoImpl", UserDao.class);
    System.out.println("userDao = " + userDao);
}
```

2.3.1.9 ⑨ 组件所对应的 bean 的 id

在我们使用 XML 方式管理 bean 的时候，每个 bean 都有一个唯一标识，便于在其他地方引用。现在使用注解后，每个组件仍然应该有一个唯一标识。

> 默认情况
>
> 类名首字母小写就是 bean 的 id。例如：UserController 类对应的 bean 的 id 就是 userController。
>
> 自定义 bean 的 id 可通过标识组件的注解的 value 属性设置自定义的 bean 的 id。

```
// 默认为 userServiceImpl
@Service("userService")
public class UserServiceImpl implements UserService
{

}
```

2.3.2、实验二：基于注解的自动装配

2.3.2.1 ① 场景模拟

> 参考基于 xml 的自动装配。
>
> 在 UserController 中声明 UserService 对象。
>
> 在 UserServiceImpl 中声明 UserDao 对象。

2.3.2.2 ② @Autowired 注解

在成员变量上直接标记@Autowired 注解即可完成自动装配，不需要提供 setXxx()方法。以后我们在项目中的正式用法就是这样。

```java
package com.myxh.spring.controller;

import com.myxh.spring.pojo.User;
import com.myxh.spring.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

/**
 * @author MYXH
 * @date 2023/8/24
 */
@Controller
public class UserController
{
    @Autowired
    private  UserService userService;

    /**
     * 保存用户信息
     *
     * @param user 一个用户信息
     */
    public void saveUser(User user)
    {
        userService.saveUser(user);
    }
}
```

```java
package com.myxh.spring.service;

import com.myxh.spring.pojo.User;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public interface UserService
{
    /**
     * 保存用户信息
     *
     * @param user 一个用户信息
     */
    void saveUser(User user);
}
```

```java
package com.myxh.spring.service.impl;

import com.myxh.spring.dao.UserDao;
import com.myxh.spring.pojo.User;
import com.myxh.spring.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

/**
 * @author MYXH
 * @date 2023/8/24
 */
@Service
public class UserServiceImpl implements UserService
{
    @Autowired
    private UserDao userDao;

    /**
     * 保存用户信息
     *
     * @param user 一个用户信息
     */
    @Override
    public void saveUser(User user)
    {
        userDao.saveUser(user);
    }
}
```

```java
package com.myxh.spring.dao;

import com.myxh.spring.pojo.User;

/**
 * @author MYXH
 * @date 2023/8/24
 */
public interface UserDao
{
    /**
     * 保存用户信息
     *
     * @param user 一个用户信息
     */
    void saveUser(User user);
}
```

```java
package com.myxh.spring.dao.impl;

import com.myxh.spring.dao.UserDao;
import com.myxh.spring.pojo.User;
import org.springframework.stereotype.Repository;

/**
 * @author MYXH
 * @date 2023/8/24
 */
@Repository
public class UserDaoImpl implements UserDao
{
    /**
     * 保存用户信息
     *
     * @param user 一个用户信息
     */
    @Override
    public void saveUser(User user)
    {
        System.out.println("保存用户信息成功");
    }
}
```

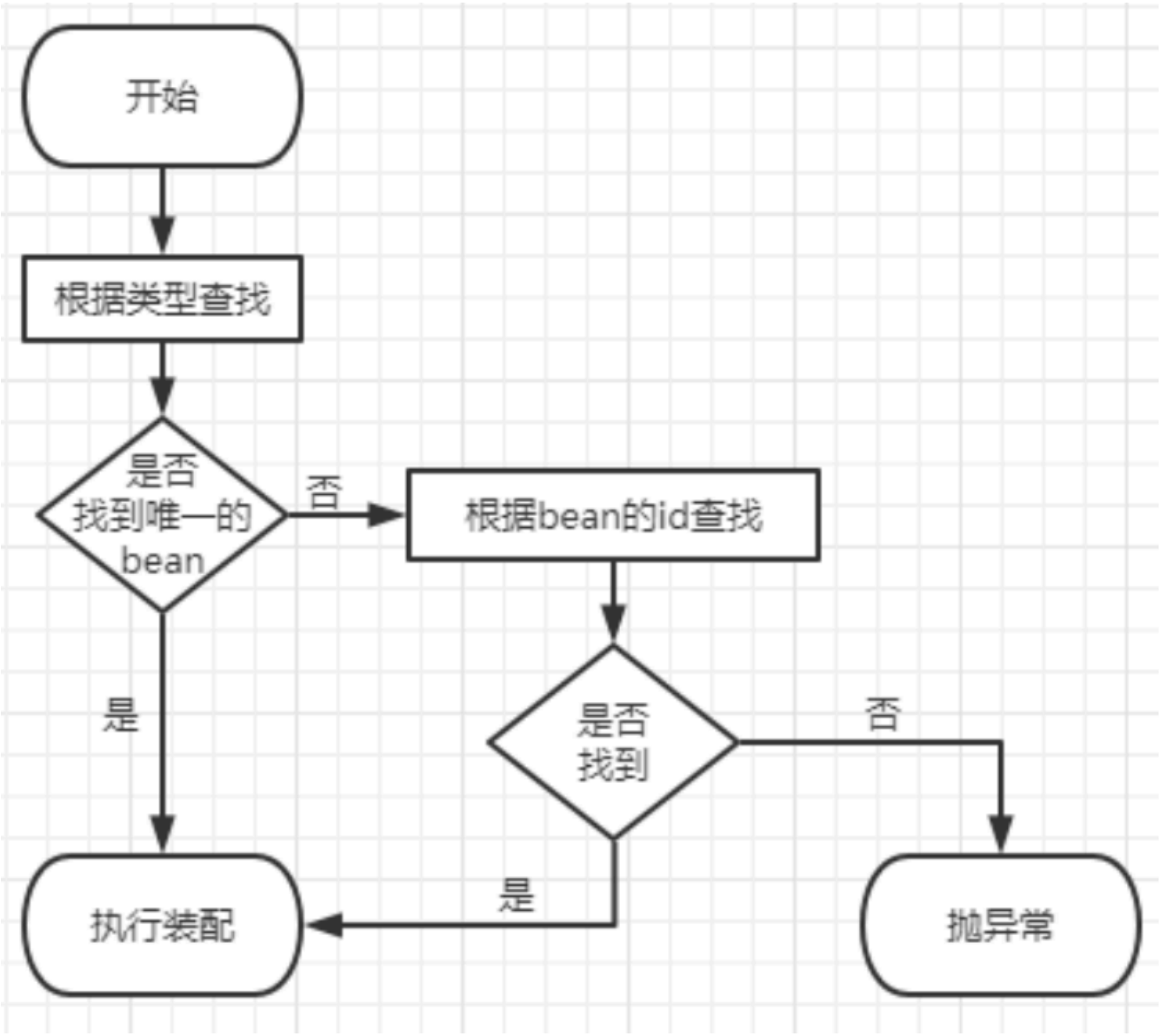
2.3.2.3 ③ @Autowired 注解其他细节

> @Autowired 注解可以标记在构造器和 set 方法上

```
package com.myxh.spring.controller;

import com.myxh.spring.pojo.User;
import com.myxh.spring.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

/**
 * @author MYXH
 * @date 2023/8/24
 */
@Controller
public class UserController
{
    private  UserService userService;

    public UserController()
    {

    }

    @Autowired
    public UserController(UserService userService)
    {
        this.userService = userService;
    }


    /**
     * 保存用户信息
     *
     * @param user 一个用户信息
     */
    public void saveUser(User user)
    {
        userService.saveUser(user);
    }
}
```

```java
package com.myxh.spring.controller;

import com.myxh.spring.pojo.User;
import com.myxh.spring.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

/**
 * @author MYXH
 * @date 2023/8/24
 */
@Controller
public class UserController
{
    private  UserService userService;

    public UserService getUserService()
    {
        return userService;
    }

    @Autowired
    public void setUserService(UserService userService)
    {
        this.userService = userService;
    }

    /**
     * 保存用户信息
     *
     * @param user 一个用户信息
     */
    public void saveUser(User user)
    {
        userService.saveUser(user);
    }
}
```

2.3.2.4 ④ @Autowired 工作流程



- 首先根据所需要的组件类型到 IOC 容器中查找
    - 能够找到唯一的 bean：直接执行装配。
    - 如果完全找不到匹配这个类型的 bean：装配失败。
    - 和所需类型匹配的 bean 不止一个
        - 没有@Qualifier 注解：根据@Autowired 标记位置成员变量的变量名作为 bean 的 id 进行匹配。
        - 能够找到：执行装配。
        - 找不到：装配失败。
        - 使用@Qualifier 注解：根据@Qualifier 注解中指定的名称作为 bean 的 id 进行匹配。

- 能够找到：执行装配。
- 找不到：装配失败。

```java
package com.myxh.spring.controller;

import com.myxh.spring.pojo.User;
import com.myxh.spring.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

/**
 * @author MYXH
 * @date 2023/8/24
 */
@Controller
public class UserController
{
    // @Autowired(required = false)
    @Qualifier("userServiceImpl")
    private  UserService userService;

    /**
     * 保存用户信息
     *
     * @param user 一个用户信息
     */
    public void saveUser(User user)
    {
        userService.saveUser(user);
    }
}
```

> @Autowired 中有属性 required，默认值为 true，因此在自动装配无法找到相应的 bean 时，会装配失败。
>
> 可以将属性 required 的值设置为 false，则表示能装就装，装不上就不装，此时自动装配的属性为默认值。
>
> 但是实际开发时，基本上所有需要装配组件的地方都是必须装配的，用不上这个属性。

3、AOP

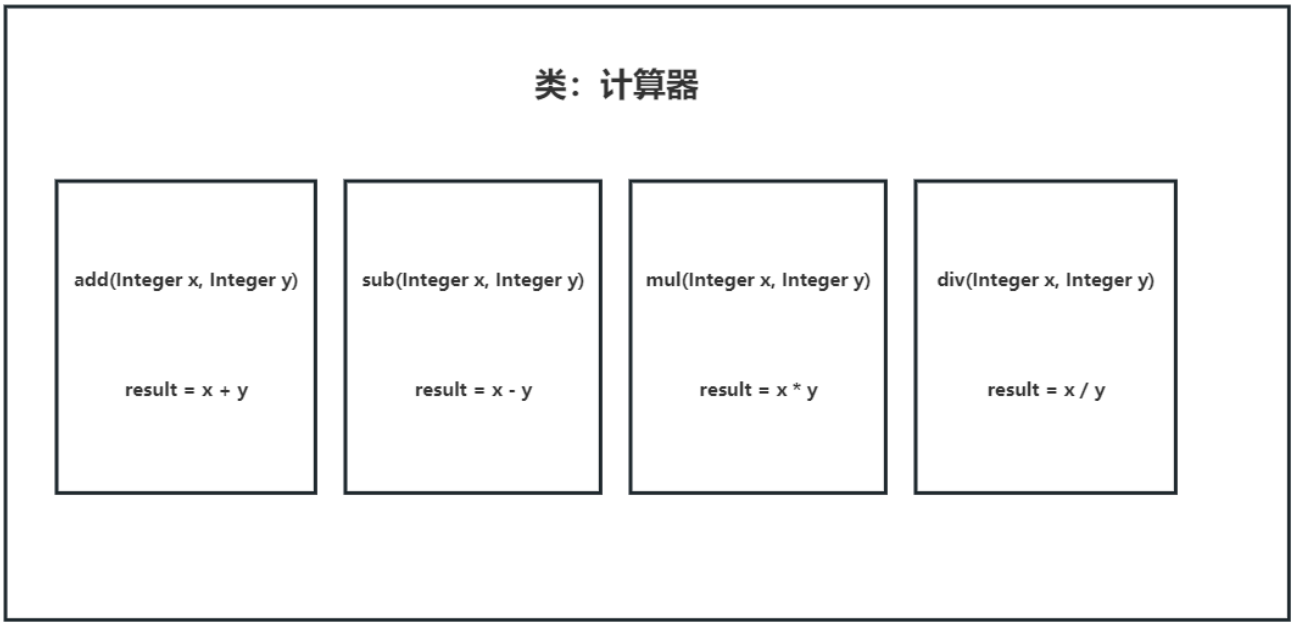
3.1、场景模拟

3.1.1、声明接口

声明计算器接口 Calculator，包含加减乘除的抽象方法。

```java
package com.myxh.spring.aop.annotation;

/**
 * @author MYXH
 * @date 2023/8/25
 */
public interface Calculator
{
    /**
     * 加法计算
     *
     * @param x 被加数
     * @param y 加数
     * @return 和
     */
    Integer add(Integer x, Integer y);

    /**
     * 减法计算
     *
     * @param x 被减数
     * @param y 减数
     * @return 差
     */
    Integer sub(Integer x, Integer y);

    /**
     * 乘法计算
     *
     * @param x 被乘数
     * @param y 乘数
     * @return 积
     */
    Integer mul(Integer x, Integer y);

    /**
     * 除法计算
     *
     * @param x 被除数
     * @param y 除数
     * @return 商
     */
    Integer div(Integer x, Integer y);
}
```

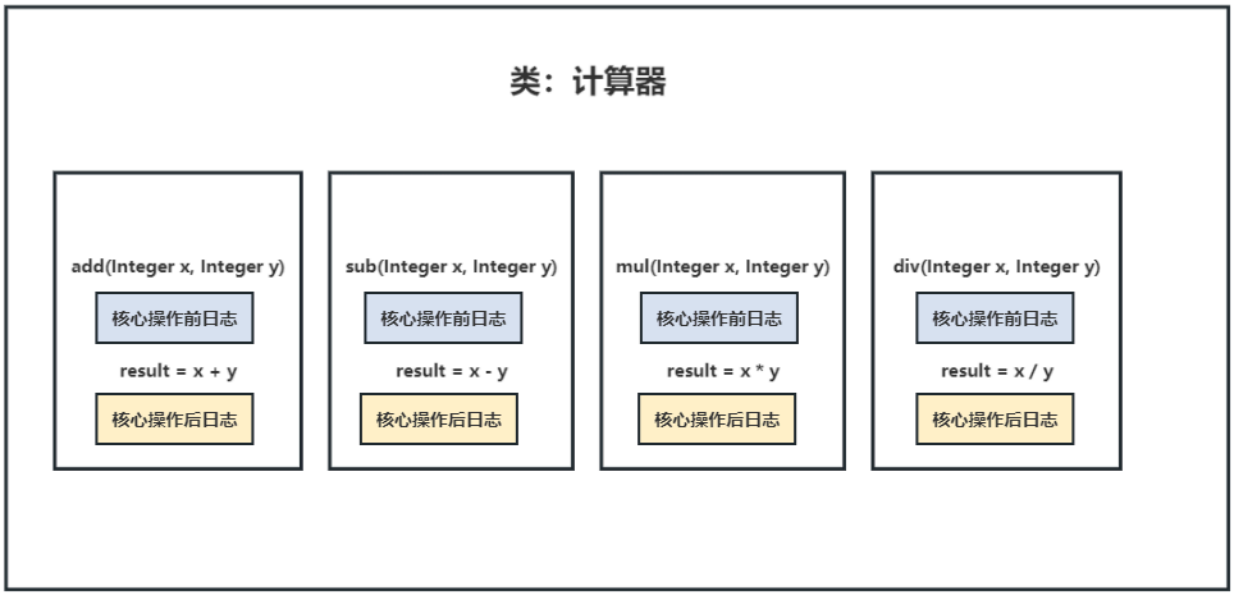
3.1.2、创建实现类


类：计算器
add(Integer x, Integer y)
result = x + y
sub(Integer x, Integer y)
result = x - y
mul(Integer x, Integer y)
result = x * y
div(Integer x, Integer y)
result = x / y

```java
package com.myxh.spring.aop.annotation;

import org.springframework.stereotype.Component;

/**
 * @author MYXH
 * @date 2023/8/25
 */
@Component
public class CalculatorImpl implements Calculator
{
    /**
     * 加法计算
     *
     * @param x 被加数
     * @param y 加数
     * @return 和
     */
    @Override
    public Integer add(Integer x, Integer y)
    {
        Integer result = x + y;
        System.out.println("方法内部: result = " + result);

        return result;
    }

    /**
     * 减法计算
     *
     * @param x 被减数
     * @param y 减数
     * @return 差
     */
    @Override
    public Integer sub(Integer x, Integer y)
    {
        Integer result = x - y;
        System.out.println("方法内部: result = " + result);

        return result;
    }

    /**
```

```
     * 乘法计算
     *
     * @param x 被乘数
     * @param y 乘数
     * @return 积
     */
    @Override
    public Integer mul(Integer x, Integer y)
    {
        Integer result = x * y;
        System.out.println("方法内部: result = " + result);

        return result;
    }

    /**
     * 除法计算
     *
     * @param x 被除数
     * @param y 除数
     * @return 商
     */
    @Override
    public Integer div(Integer x, Integer y)
    {
        Integer result = x / y;
        System.out.println("方法内部: result = " + result);

        return result;
    }
}
```

3.1.3、创建带日志功能的实现类



```java
package com.myxh.spring.proxy;

/**
 * @author MYXH
 * @date 2023/8/25
 */
public class CalculatorImpl implements Calculator
{
    /**
     * 加法计算
     *
     * @param x 被加数
     * @param y 加数
     * @return 和
     */
    @Override
    public Integer add(Integer x, Integer y)
    {
        System.out.println("[日志] 方法: add, 参数: [" + x + ", " + y + "]");

        Integer result = x + y;
        System.out.println("方法内部: result = " + result);

        System.out.println("[日志] 方法: add, 结果: " + result);

        return result;
    }

    /**
     * 减法计算
     *
     * @param x 被减数
     * @param y 减数
     * @return 差
     */
    @Override
    public Integer sub(Integer x, Integer y)
    {
        System.out.println("[日志] 方法: sub, 参数: [" + x + ", " + y + "]");

        Integer result = x - y;
        System.out.println("方法内部: result = " + result);

        System.out.println("[日志] 方法: sub, 结果: " + result);
```

```java
        return result;
    }

    /**
     * 乘法计算
     *
     * @param x 被乘数
     * @param y 乘数
     * @return 积
     */
    @Override
    public Integer mul(Integer x, Integer y)
    {
        System.out.println("[日志] 方法: mul, 参数: [" + x + ", " + y + "]");

        Integer result = x * y;
        System.out.println("方法内部: result = " + result);

        System.out.println("[日志] 方法: mul, 结果: " + result);

        return result;
    }

    /**
     * 除法计算
     *
     * @param x 被除数
     * @param y 除数
     * @return 商
     */
    @Override
    public Integer div(Integer x, Integer y)
    {

        System.out.println("[日志] 方法: div, 参数: [" + x + ", " + y + "]");

        Integer result = x / y;
        System.out.println("方法内部: result = " + result);

        System.out.println("[日志] 方法: div, 结果: " + result);

        return result;
    }
```

```
}
```

3.1.4、提出问题

3.1.4.1 ① 现有代码缺陷

针对带日志功能的实现类，我们发现有如下缺陷：

- 对核心业务功能有干扰，导致程序员在开发核心业务功能时分散了精力。
- 附加功能分散在各个业务功能方法中，不利于统一维护。

3.1.4.2 ② 解决思路

解决这两个问题，核心就是：解耦。我们需要把附加功能从业务功能代码中抽取出来。

3.1.4.3 ③ 困难

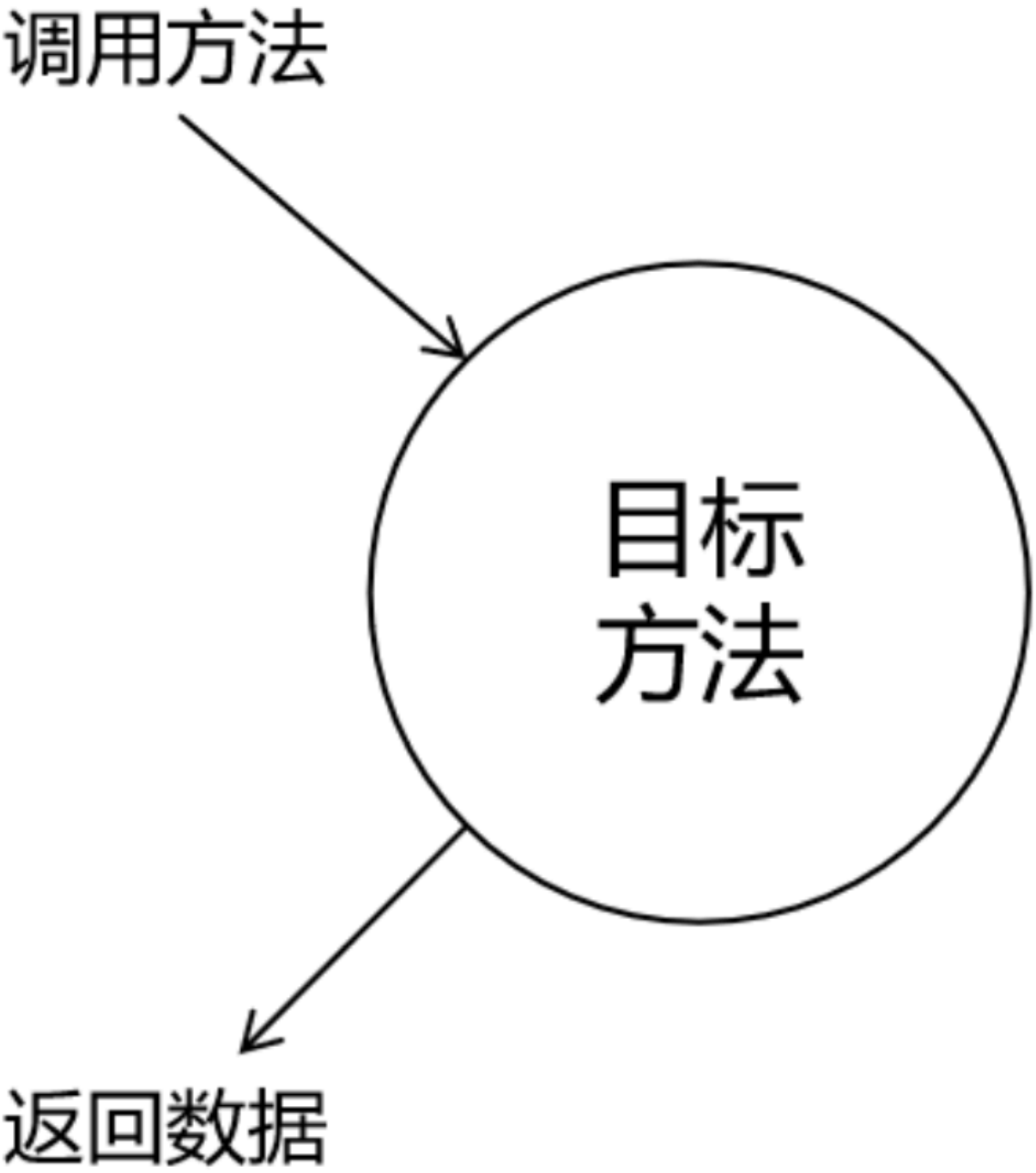
解决问题的困难：要抽取的代码在方法内部，靠以前把子类中的重复代码抽取到父类的方式没法解决。所以需要引入新的技术。

3.2、代理模式

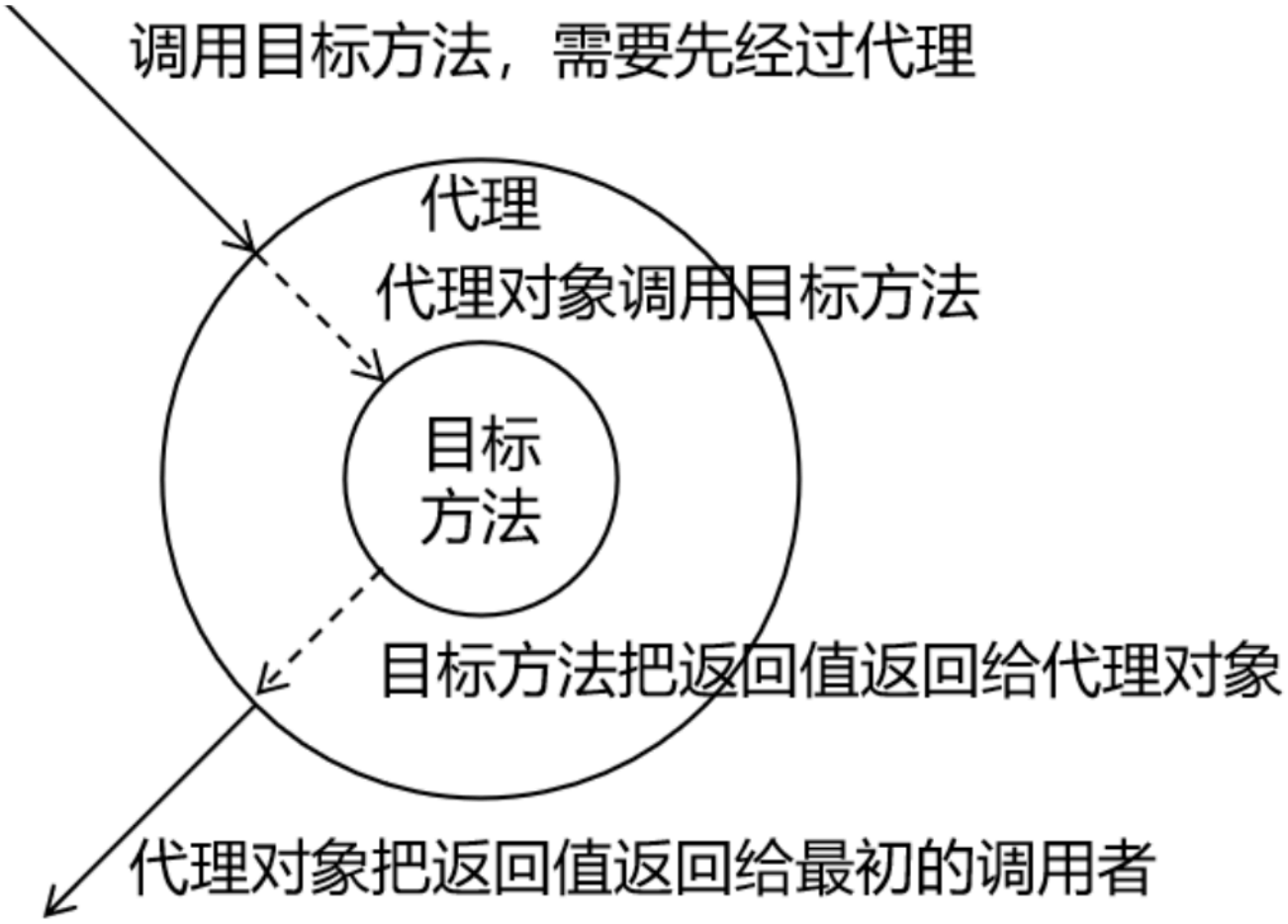
3.2.1、概念

3.2.1.1 ① 介绍

二十三种设计模式中的一种，属于结构型模式。它的作用就是通过提供一个代理类，让我们在调用目标方法的时候，不再是直接对目标方法进行调用，而是通过代理类**间接**调用。让不属于目标方法核心逻辑的代码从目标方法中剥离出来——**解耦**。调用目标方法时先调用代理对象的方法，减少对目标方法的调用和打扰，同时让附加功能能够集中在一起也有利于统一维护。



使用代理后：



3.2.1.2 ② 生活中的代理

- 广告商找大明星拍广告需要经过经纪人。
- 合作伙伴找大老板谈合作要约见面时间需要经过秘书。
- 房产中介是买卖双方的代理。

3.2.1.3 ③ 相关术语

- 代理：将非核心逻辑剥离出来以后，封装这些非核心逻辑的类、对象、方法。
- 目标：被代理“套用”了非核心逻辑代码的类、对象、方法。

3.2.2、静态代理

创建静态代理类：

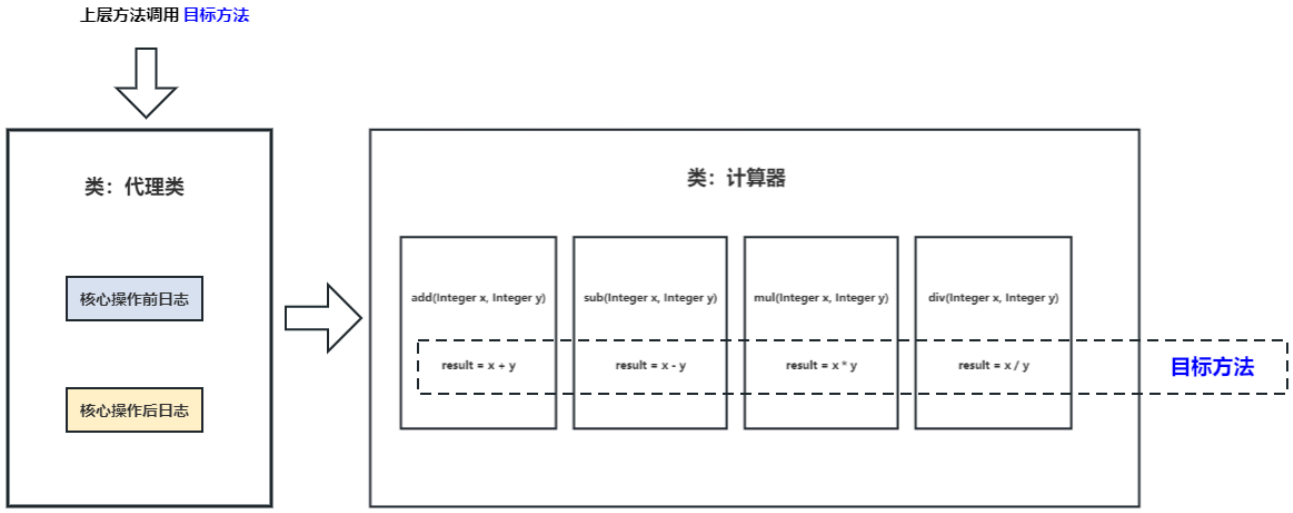
```java
package com.myxh.spring.proxy;

/**
 * @author MYXH
 * @date 2023/8/25
 */
public class CalculatorStaticProxy implements Calculator
{
    private CalculatorImpl target;

    public CalculatorStaticProxy(CalculatorImpl target)
    {
        this.target = target;
    }

    /**
     * 加法计算
     *
     * @param x 被加数
     * @param y 加数
     * @return 和
     */
    @Override
    public Integer add(Integer x, Integer y)
    {
        System.out.println("[日志] 方法: add, 参数: [" + x + ", " + y + "]");

        Integer result = target.add(x, y);

        System.out.println("[日志] 方法: add, 结果: " + result);

        return result;
    }

    /**
     * 减法计算
     *
     * @param x 被减数
     * @param y 减数
     * @return 差
     */
    @Override
    public Integer sub(Integer x, Integer y)
    {
```

```
        System.out.println("[日志] 方法: sub, 参数: [" + x + ", " + y + "]");

        Integer result = target.sub(x, y);

        System.out.println("[日志] 方法: sub, 结果: " + result);

        return result;
    }

    /**
     * 乘法计算
     *
     * @param x 被乘数
     * @param y 乘数
     * @return 积
     */
    @Override
    public Integer mul(Integer x, Integer y)
    {
        System.out.println("[日志] 方法: mul, 参数: [" + x + ", " + y + "]");

        Integer result = target.mul(x, y);

        System.out.println("[日志] 方法: mul, 结果: " + result);

        return result;
    }

    /**
     * 除法计算
     *
     * @param x 被除数
     * @param y 除数
     * @return 商
     */
    @Override
    public Integer div(Integer x, Integer y)
    {
        System.out.println("[日志] 方法: div, 参数: [" + x + ", " + y + "]");

        Integer result = target.div(x, y);

        System.out.println("[日志] 方法: div, 结果: " + result);
```

```
        return result;
    }
}
```

> 静态代理确实实现了解耦，但是由于代码都写死了，完全不具备任何的灵活性。就拿日志功能来说，将来其他地方也需要附加日志，那还得再声明更多个静态代理类，那就产生了大量重复的代码，日志功能还是分散的，没有统一管理。
>
> 提出进一步的需求：将日志功能集中到一个代理类中，将来有任何日志需求，都通过这一个代理类来实现。这就需要使用动态代理技术了。

3.2.3、动态代理



生产代理对象的工厂类：

```java
package com.myxh.spring.proxy;

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

/**
 * @author MYXH
 * @date 2023/8/25
 */
public class ProxyFactory
{
    private final Object target;

    public ProxyFactory(Object target)
    {
        this.target = target;
    }

    public Object getProxy()
    {
        /*
          ClassLoader loader: 指定加载动态生成的代理类的类加载器
          Class<?>[] interfaces: 获取目标对象实现的所有接口的 class 对象的数组
          InvocationHandler h: 设置代理类中的抽象方法如何重写
         */
        ClassLoader classLoader = this.getClass().getClassLoader();
        Class<?>[] interfaces = target.getClass().getInterfaces();
        InvocationHandler handler = new InvocationHandler()
        {
            /**
             * proxy 表示代理对象，method 表示要执行的方法，args 表示要执行的方法的参数列表
             *
             * @param proxy 调用该方法的代理实例
             *
             * @param method 与代理实例上调用的接口方法相对应的 method 实例。
             *               Method 对象的声明类将是在其中声明该方法的接口，该接口可能是代理类通过其继承
             *
             * @param args 一个对象数组，包含在代理实例上的方法调用中传递的参数值，如果接口方法不接受参数
             *             基元类型的参数被包装在适当的基元包装类的实例中，例如 java.lang.Integer 或 ja
             *
             * @return 从代理实例上的方法调用返回的值。
```

```
             * 如果接口方法声明的返回类型是基元类型，则此方法返回的值必须是相应基元包装类的实例；否则，它必
             * 如果此方法返回的值为 null，并且接口方法的返回类型为基元，则方法调用将在代理实例上引发 Null
             * 如果此方法返回的值与如上所述接口方法声明的返回类型不兼容，则方法调用将在代理实例上引发 Clas
             * @throws Throwable 从代理实例上的方法调用中抛出的异常。
             * 异常的类型必须可分配给接口方法的 throws 子句中声明的任何异常类型，或者可分配给未检查的异常
             * 如果此方法抛出了一个检查的异常，但该异常不可分配给接口方法的 throw 子句中声明任何异常类型，
             */
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
            {
                Object result = null;

                try
                {
                    System.out.println("[日志] 方法: " + method.getName() + ", 参数: " + Arrays.toSt

                    result = method.invoke(target, args);

                    System.out.println("[日志] 方法: " + method.getName() + ", 结果: " + result);
                } catch (IllegalAccessException | IllegalArgumentException | InvocationTargetExcep
                {
                    System.out.println("[日志] 方法: " + method.getName() + ", 异常: " + e);
                    e.printStackTrace();
                } finally
                {
                    System.out.println("[日志] 方法: " + method.getName() + ", 方法执行完毕");
                }

                return result;
            }
        };

        return Proxy.newProxyInstance(classLoader, interfaces, handler);
    }
}
```

3.2.4、测试

```
package com.myxh.spring.proxy.test;

import com.myxh.spring.proxy.Calculator;
import com.myxh.spring.proxy.CalculatorImpl;
import com.myxh.spring.proxy.ProxyFactory;
import org.junit.Test;

/**
 * @author MYXH
 * @date 2023/8/25
 */
public class ProxyTest
{
    /**
     * 动态代理有两种：
     * 1、jdk 动态代理，要求必须有接口，最终生成的代理类和目标类实现相同的接口
     * 在 com.sun.proxy 包下，类名为 $proxy2
     * 2、cglib 动态代理，最终生成的代理类会继承目标类，并且和目标类在相同的包下
     */
    @Test
    public void testProxy()
    {
        // CalculatorStaticProxy proxy = new CalculatorStaticProxy(new CalculatorImpl());
        // proxy.add(1, 2);

        ProxyFactory proxyFactory = new ProxyFactory(new CalculatorImpl());
        Calculator proxy = (Calculator) proxyFactory.getProxy();
        proxy.add(1, 2);
        // proxy.div(1, 0);
    }
}
```

3.3、AOP 概念及相关术语

3.3.1、概述

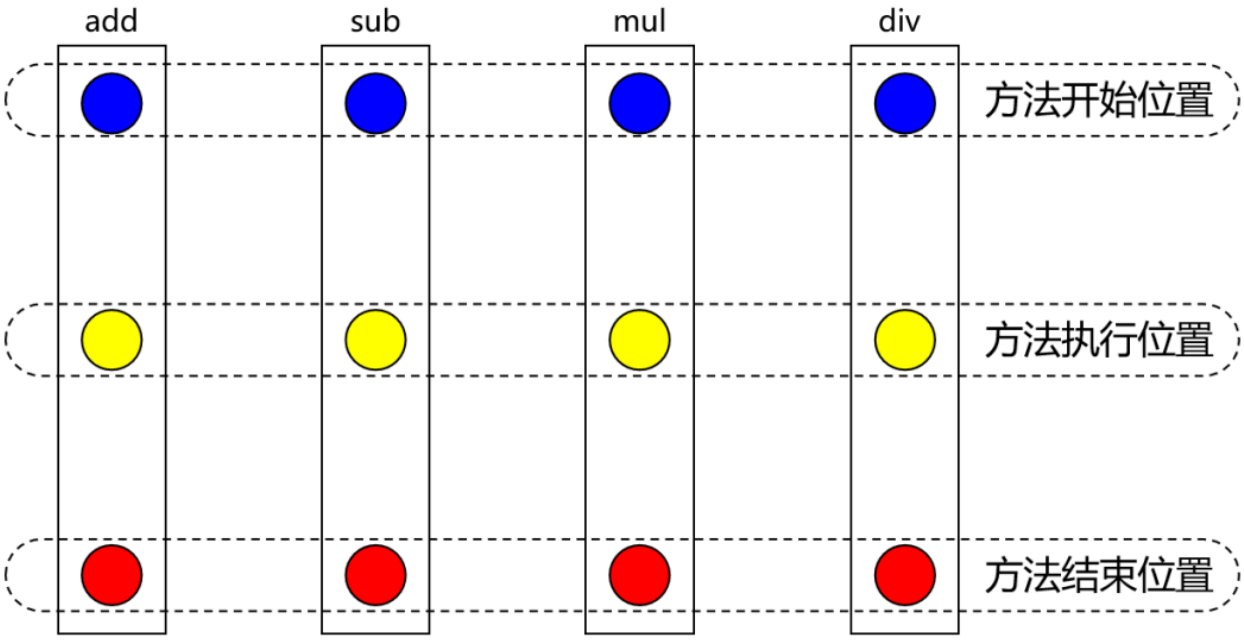
AOP（Aspect Oriented Programming）是一种设计思想，是软件设计领域中的面向切面编程，它是面向对象编程的一种补充和完善，它以通过预编译方式和运行期动态代理方式实现在不修改源代码的情况下给程序动态统一添加额外功能的一种技术。

3.3.2、相关术语

3.3.2.1 ① 横切关注点

从每个方法中抽取出来的同一类非核心业务。在同一个项目中，我们可以使用多个横切关注点对相关方法进行多个不同方面的增强。

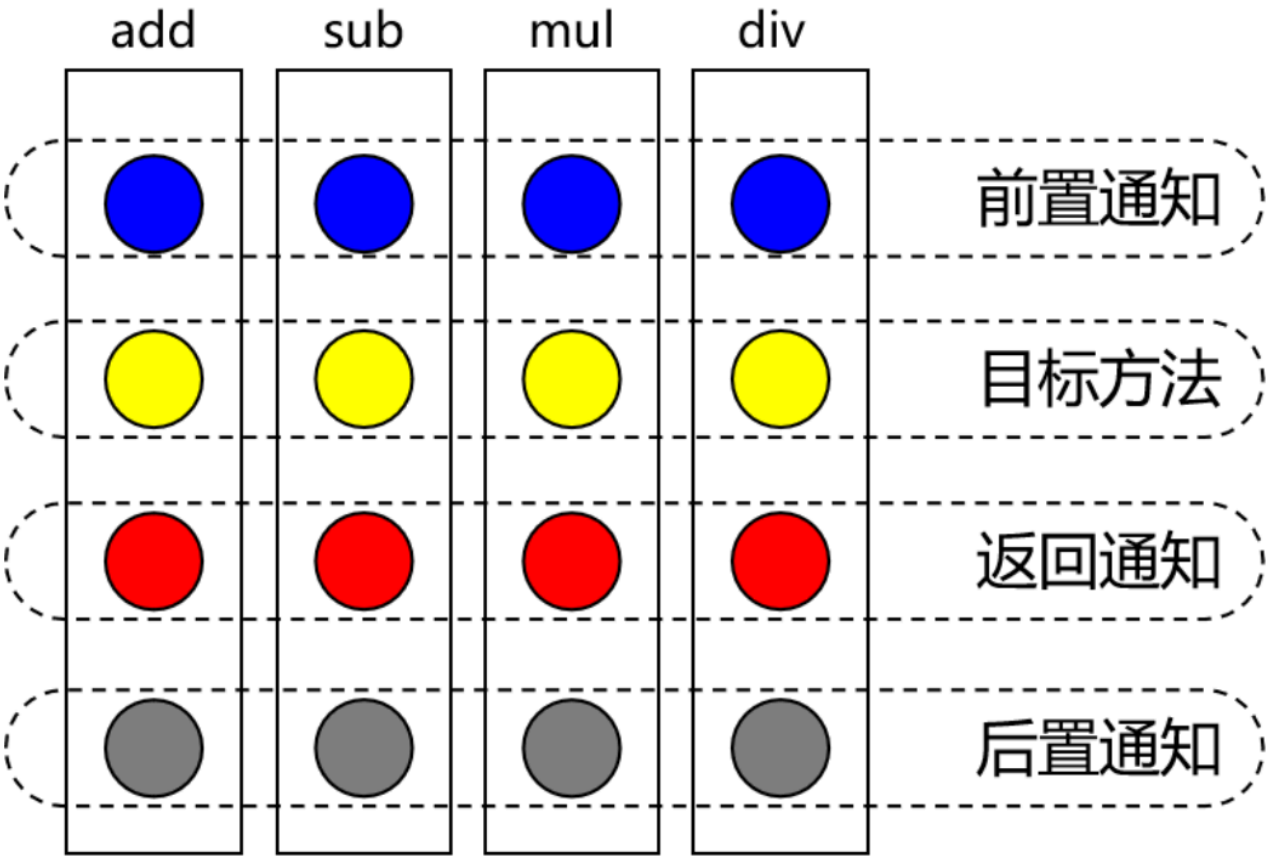
这个概念不是语法层面天然存在的，而是根据附加功能的逻辑上的需要：有十个附加功能，就有十个横切关注点。



3.3.2.2 ② 通知

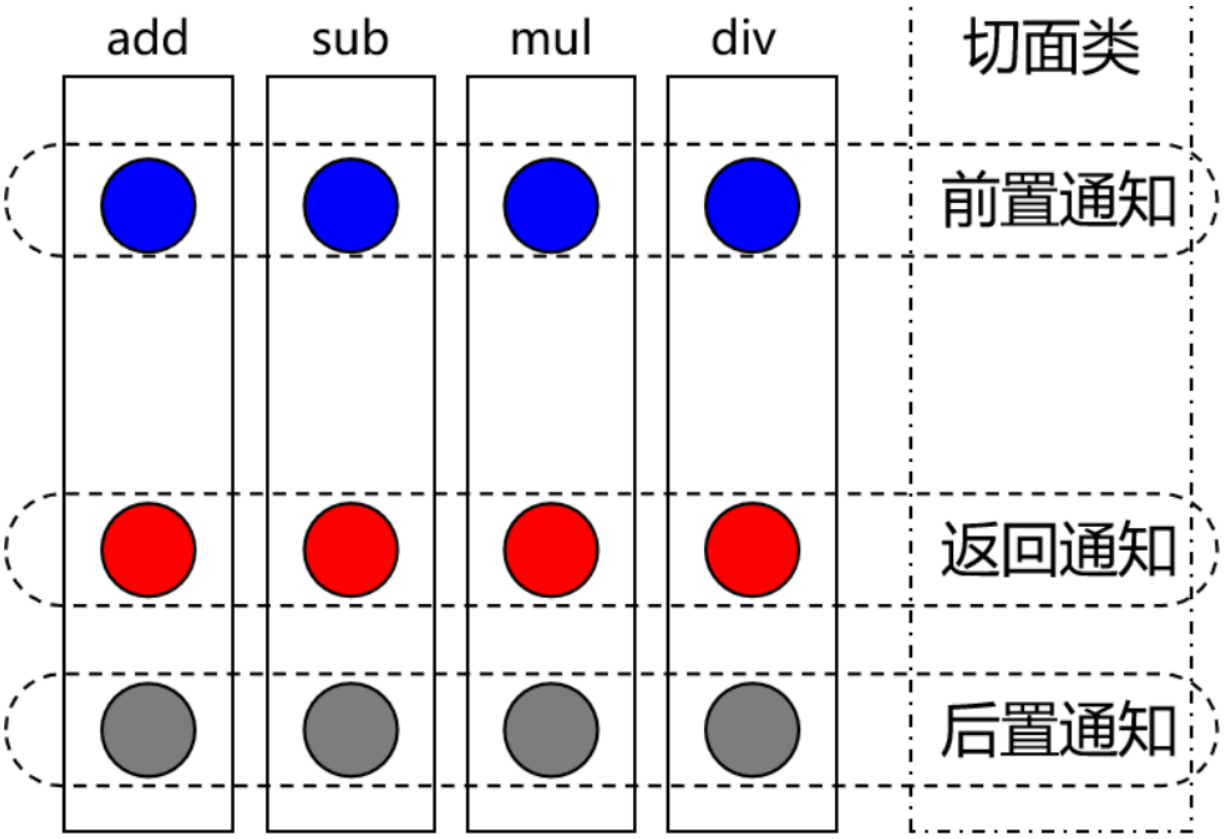
每一个横切关注点上要做的事情都需要写一个方法来实现，这样的方法就叫通知方法。

- 前置通知：在被代理的目标方法**前**执行。
- 返回通知：在被代理的目标方法**成功结束**后执行（**寿终正寝**）。
- 异常通知：在被代理的目标方法**异常结束**后执行（**死于非命**）。
- 后置通知：在被代理的目标方法**最终结束**后执行（**盖棺定论**）。
- 环绕通知：使用 try…catch…finally 结构围绕**整个**被代理的目标方法，包括上面四种通知对应的所有位置。



3.3.2.3 ③ 切面

封装通知方法的类。



3.3.2.4 ④ 目标

被代理的目标对象。

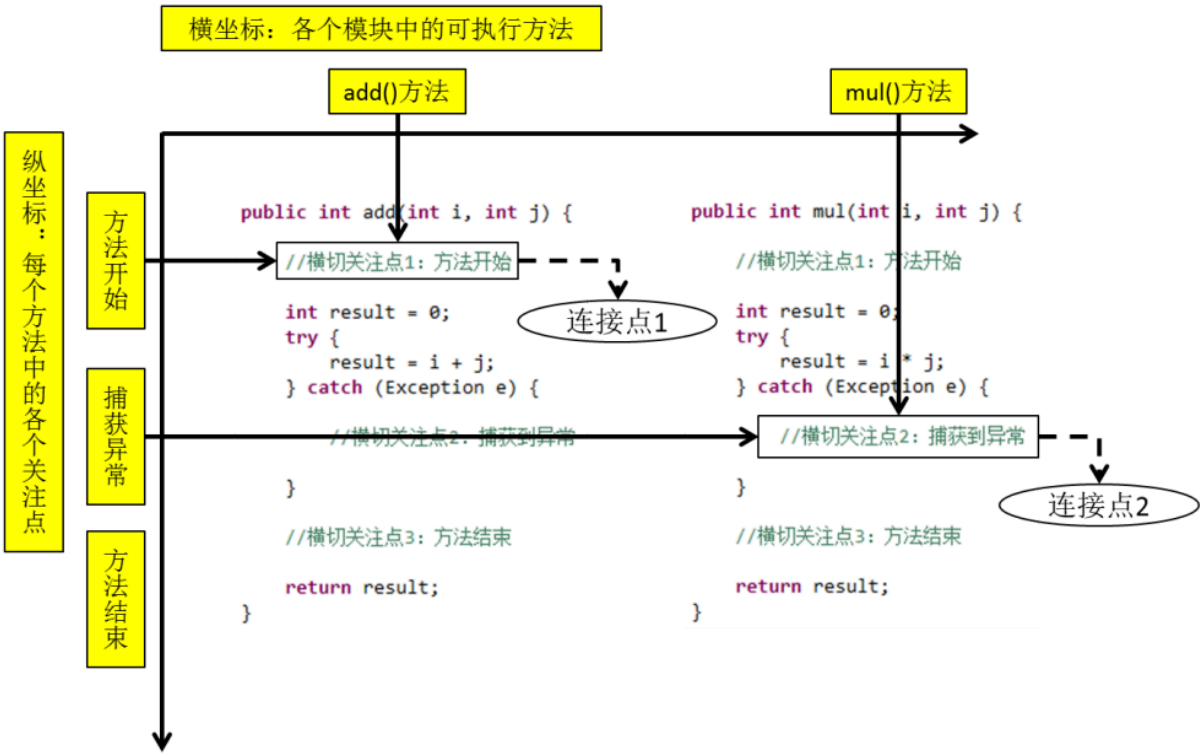
3.3.2.5 ⑤ 代理

向目标对象应用通知之后创建的代理对象。

3.3.2.6 ⑥ 连接点

这也是一个纯逻辑概念，不是语法定义的。

把方法排成一排，每一个横切位置看成 x 轴方向，把方法从上到下执行的顺序看成 y 轴，x 轴和 y 轴的交叉点就是连接点。



3.3.2.7 ⑦ 切入点

定位连接点的方式。

每个类的方法中都包含多个连接点，所以连接点是类中客观存在的事物（从逻辑上来说）。

如果把连接点看作数据库中的记录，那么切入点就是查询记录的 SQL 语句。

Spring 的 AOP 技术可以通过切入点定位到特定的连接点。

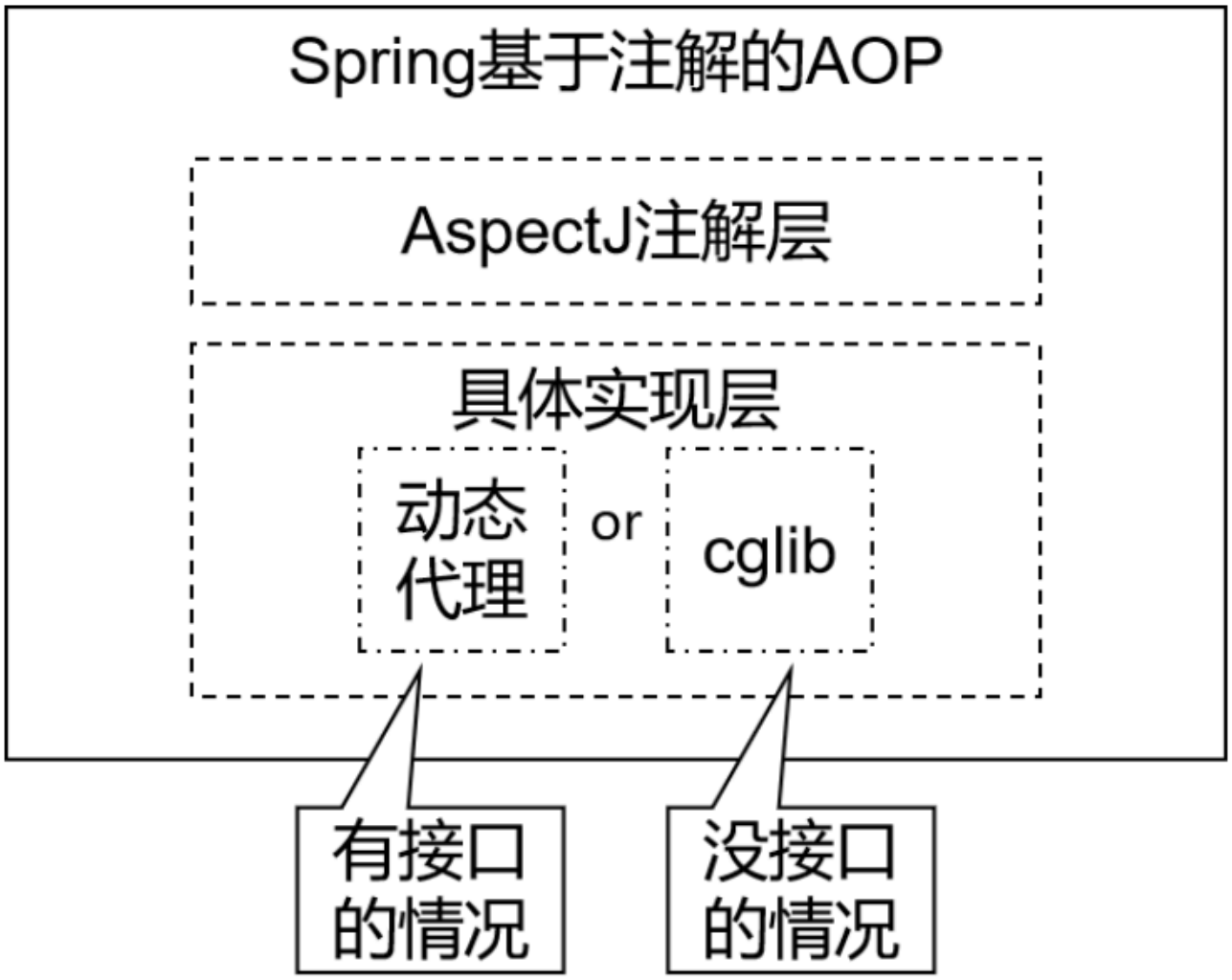
切点通过 org.springframework.aop.Pointcut 接口进行描述，它使用类和方法作为连接点的查询条件。

3.3.3、作用

- 简化代码：把方法中固定位置的重复的代码**抽取**出来，让被抽取的方法更专注于自己的核心功能，提高内聚性。
- 代码增强：把特定的功能封装到切面类中，看哪里有需要，就往上套，被**套用**了切面逻辑的方法就被切面给增强了。

3.4、基于注解的 AOP

3.4.1、技术说明



- 动态代理（InvocationHandler）：JDK 原生的实现方式，需要被代理的目标类必须实现接口。因为这个技术要求**代理对象和目标对象实现同样的接口**（兄弟两个拜把子模式）。
- cglib：通过**继承被代理的目标类**（认干爹模式）实现代理，所以不需要目标类实现接口。
- AspectJ：本质上是静态代理，**将代理逻辑“织入”被代理的目标类编译得到的字节码文件**，所以最终效果是动态的。weaver 就是织入器。Spring 只是借用了 AspectJ 中的注解。

3.4.2、准备工作

3.4.2.1 ① 添加依赖

在 IOC 所需依赖基础上再加入下面依赖即可：

```
<!-- spring-aspects 会帮我们传递过来 aspectjweaver -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>6.0.9</version>
</dependency>
```

3.4.2.2 ② 准备被代理的目标资源

接口：

```java
package com.myxh.spring.aop.annotation;

/**
 * @author MYXH
 * @date 2023/8/28
 */
public interface Calculator
{
    /**
     * 加法计算
     *
     * @param x 被加数
     * @param y 加数
     * @return 和
     */
    Integer add(Integer x, Integer y);

    /**
     * 减法计算
     *
     * @param x 被减数
     * @param y 减数
     * @return 差
     */
    Integer sub(Integer x, Integer y);

    /**
     * 乘法计算
     *
     * @param x 被乘数
     * @param y 乘数
     * @return 积
     */
    Integer mul(Integer x, Integer y);

    /**
     * 除法计算
     *
     * @param x 被除数
     * @param y 除数
     * @return 商
     */
    Integer div(Integer x, Integer y);
}
```

实现类：

```java
package com.myxh.spring.aop.annotation;

import org.springframework.stereotype.Component;

/**
 * @author MYXH
 * @date 2023/8/28
 */
@Component
public class CalculatorImpl implements Calculator
{
    /**
     * 加法计算
     *
     * @param x 被加数
     * @param y 加数
     * @return 和
     */
    @Override
    public Integer add(Integer x, Integer y)
    {
        Integer result = x + y;
        System.out.println("方法内部: result = " + result);

        return result;
    }

    /**
     * 减法计算
     *
     * @param x 被减数
     * @param y 减数
     * @return 差
     */
    @Override
    public Integer sub(Integer x, Integer y)
    {
        Integer result = x - y;
        System.out.println("方法内部: result = " + result);

        return result;
    }

    /**
```

```
     * 乘法计算
     *
     * @param x 被乘数
     * @param y 乘数
     * @return 积
     */
    @Override
    public Integer mul(Integer x, Integer y)
    {
        Integer result = x * y;
        System.out.println("方法内部: result = " + result);

        return result;
    }

    /**
     * 除法计算
     *
     * @param x 被除数
     * @param y 除数
     * @return 商
     */
    @Override
    public Integer div(Integer x, Integer y)
    {
        Integer result = x / y;
        System.out.println("方法内部: result = " + result);

        return result;
    }
}
```

3.4.3、创建切面类并配置

```java
package com.myxh.spring.aop.annotation;

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

import java.util.Arrays;

/**
 * @author MYXH
 * @date 2023/8/28
 * @description
 * 1、在切面中，需要通过指定的注解将方法标识为通知方法
 * {@code @Before}：前置通知，在目标对象方法执行之前执行
 * {@code @After}：后置通知，在目标对象方法的 finally 字句中执行
 * {@code @AfterReturning}：返回通知，在目标对象方法返回值之后执行
 * {@code @AfterThrowing}：异常通知，在目标对象方法的 catch 子句中执行
 * {@code @Around}：环绕通知，环绕通知的方法的返回值一定要和目标对象方法的返回值一致
 * <p>
 * 2、切入点表达式：设置在标识通知的注解的 value 属性中
 * {@code execution(public Integer com.myxh.spring.aop.annotation.CalculatorImpl.add(Integer, Inte
 * {@code execution(* com.myxh.spring.aop.annotation.CalculatorImpl.*(..))}
 * 第一个 * 表示任意的访问修饰符和返回值类型
 * 第二个 * 表示类中任意的方法
 * .. 表示任意的参数列表
 * 类的地方也可以使用 *，表示包下所有的类
 * <p>
 * 3、获取连接点的信息
 * 在通知方法的参数位置，设置 JoinPoint 类型的参数，就可以获取连接点所对应方法的信息
 * {@code
 * // 获取连接点所对应方法的签名信息
 * Signature signature = joinPoint.getSignature();
 * <p>
 * // 获取连接点所对应方法的方法名
 * String name = signature.getName();
 * <p>
 * // 获取连接点所对应方法的参数
 * Object[] args = joinPoint.getArgs();
 * }
 */
@Component
// 将当前组件标识为切面
```

```java
@Aspect
public class LoggerAspect
{
    // @Before("execution(public Integer com.myxh.spring.aop.annotation.CalculatorImpl.add(Integer
    @Before("execution(* com.myxh.spring.aop.annotation.CalculatorImpl.*(..))")
    public void beforeAdviceMethod(JoinPoint joinPoint)
    {
        // 获取连接点所对应方法的签名信息
        Signature signature = joinPoint.getSignature();

        // 获取连接点所对应方法的方法名
        String name = signature.getName();

        // 获取连接点所对应方法的参数
        Object[] args = joinPoint.getArgs();

        System.out.println("[LoggerAspect] 方法: " + name + ", 参数: " + Arrays.toString(args));
    }

    @After("execution(* com.myxh.spring.aop.annotation.CalculatorImpl.*(..))")
    public void afterAdviceMethod(JoinPoint joinPoint)
    {
        // 获取连接点所对应方法的签名信息
        Signature signature = joinPoint.getSignature();

        // 获取连接点所对应方法的方法名
        String name = signature.getName();

        System.out.println("[LoggerAspect] 方法: " + name + ", 方法执行完毕");
    }

    /**
     * 在返回通知中若要获取目标对象方法的返回值
     * 只需要通过 {@code @AfterReturning} 注解的 returning 属性
     * 就可以将通知方法的某个参数指定为接收目标对象方法的返回值的参数
     *
     * @param joinPoint 连接点
     * @param result 结果
     */
    @AfterReturning(value = "execution(* com.myxh.spring.aop.annotation.CalculatorImpl.*(..))", re
    public void afterReturningAdviceMethod(JoinPoint joinPoint, Object result)
    {
        // 获取连接点所对应方法的签名信息
        Signature signature = joinPoint.getSignature();
```

```
        // 获取连接点所对应方法的方法名
        String name = signature.getName();

        System.out.println("[LoggerAspect] 方法: " + name + ", 结果: " + result);
    }

    /**
     * 在异常通知中若要获取目标对象方法的异常
     * 只需要通过 {@code @AfterThrowing} 注解的 throwing 属性
     * 就可以将通知方法的某个参数指定为接收目标对象方法出现的异常的参数
     *
     * @param joinPoint 连接点
     * @param exception 异常
     */
    @AfterThrowing(value = "execution(* com.myxh.spring.aop.annotation.CalculatorImpl.*(..))", thr
    public void afterThrowingAdviceMethod(JoinPoint joinPoint, Throwable exception)
    {
        // 获取连接点所对应方法的签名信息
        Signature signature = joinPoint.getSignature();

        // 获取连接点所对应方法的方法名
        String name = signature.getName();

        System.out.println("[LoggerAspect] 方法: " + name + ", 异常: " + exception);
    }

    /**
     * 环绕通知的方法的返回值一定要和目标对象方法的返回值一致
     *
     * @param joinPoint 连接点
     * @return result 结果
     */
    @Around("execution(* com.myxh.spring.aop.annotation.CalculatorImpl.*(..))")
    public Object aroundAdviceMethod(ProceedingJoinPoint joinPoint)
    {
        Object result = null;

        try
        {
            System.out.println("环绕通知 -> 前置通知");

            // 表示目标对象方法的执行
            result  = joinPoint.proceed();
```

```
            System.out.println("环绕通知 -> 返回通知");
        }
        catch (Throwable e)
        {
            System.out.println("环绕通知 -> 异常通知");
            e.printStackTrace();
        }
        finally
        {
            System.out.println("环绕通知 -> 后置通知");
        }

        return result;
    }
}
```

在 Spring 的配置文件中配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop" xmlns="http://www.springframework.org
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.
    <!--
    AOP 的注意事项:
    切面类和目标类都需要交给 IOC 器管理
    切面类必须通过 @Aspect 注解标识为一个切面
    在 Spring 的配置文件中设置 <aop:aspectj-autoproxy/> 开启基于注解的 AOP
     -->
    <!-- 扫描组件 -->
    <context:component-scan base-package="com.myxh.spring.aop.annotation"/>

    <!-- 开启基于注解的 AOP -->
    <aop:aspectj-autoproxy/>
</beans>
```

3.4.4、各种通知

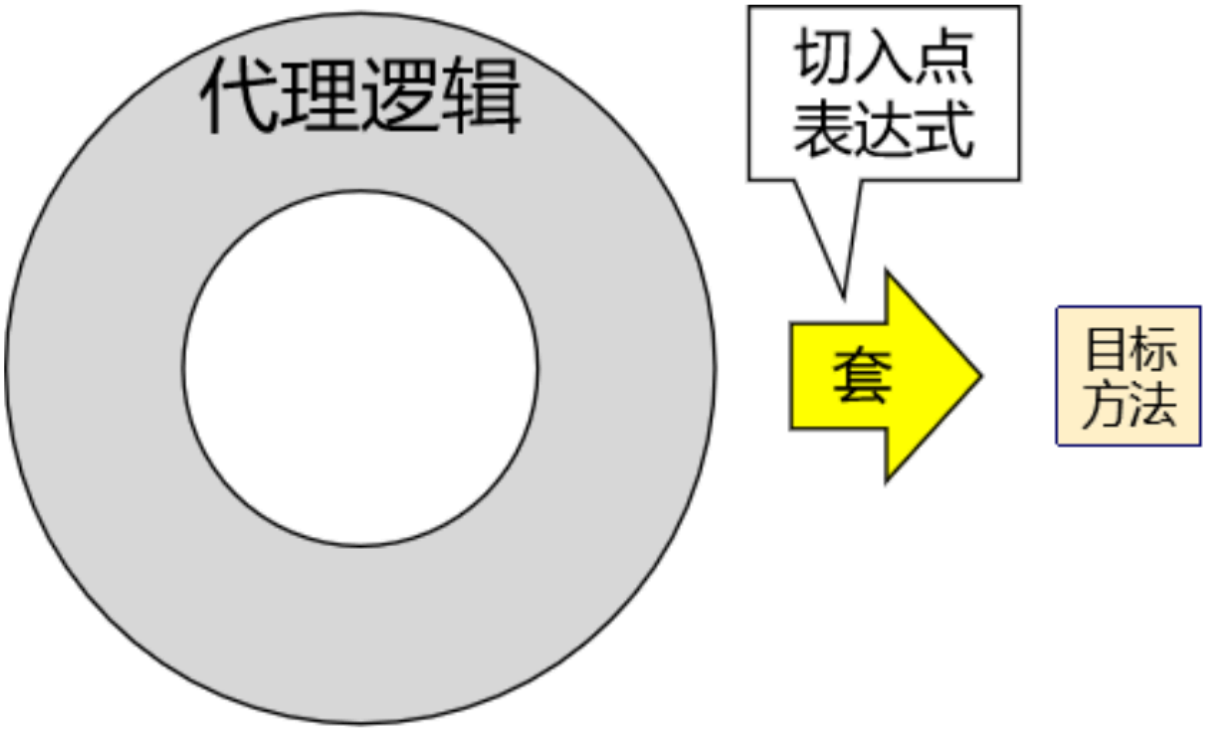
- 前置通知：使用@Before 注解标识，在被代理的目标方法**前**执行。
- 返回通知：使用@AfterReturning 注解标识，在被代理的目标方法**成功结束**后执行（**寿终正寝**）。

- 异常通知：使用@AfterThrowing 注解标识，在被代理的目标方法**异常结束**后执行（**死于非命**）。
- 后置通知：使用@After 注解标识，在被代理的目标方法**最终结束**后执行（**盖棺定论**）。
- 环绕通知：使用@Around 注解标识，使用 try…catch…finally 结构围绕**整个**被代理的目标方法，包括上面四种通知对应的所有位置。

> 各种通知的执行顺序：
>
> - Spring 版本 5.3.x 以前：
>   - 前置通知
>   - 目标操作
>   - 后置通知
>   - 返回通知或异常通知
> - Spring 版本 5.3.x 以后：
>   - 前置通知
>   - 目标操作
>   - 返回通知或异常通知
>   - 后置通知

3.4.5、切入点表达式语法

3.4.5.1 ① 作用

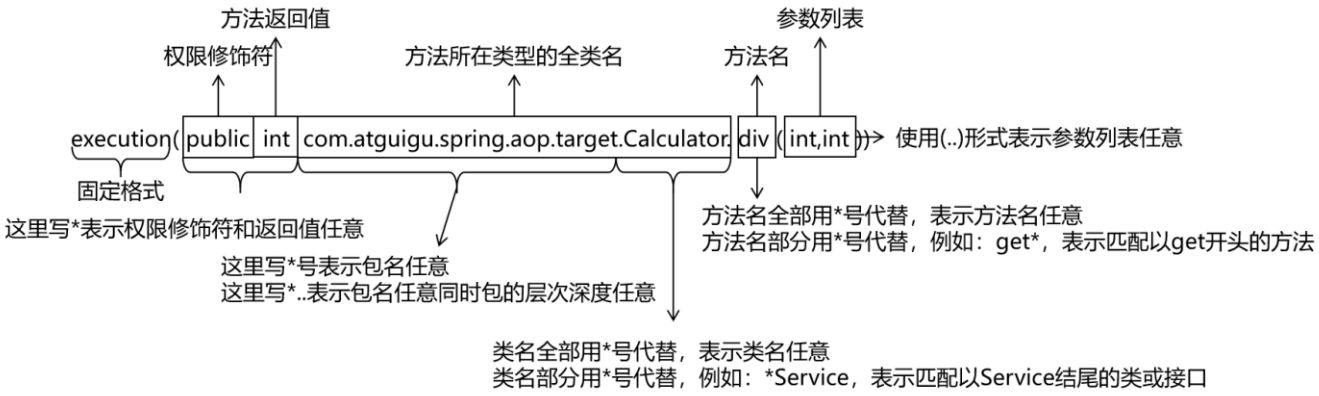


3.4.5.2 ② 语法细节

- 用 * 号代替“权限修饰符”和“返回值”部分表示“权限修饰符”和“返回值”不限。
- 在包名的部分，一个“*”号只能代表包的层次结构中的一层，表示这一层是任意的。
    - 例如：*.Hello 匹配 com.Hello，不匹配 com.myxh.Hello。
- 在包名的部分，使用“..”表示包名任意、包的层次深度任意。
- 在类名的部分，类名部分整体用 * 号代替，表示类名任意。
- 在类名的部分，可以使用 * 号代替类名的一部分。
    - 例如：*Service 匹配所有名称以 Service 结尾的类或接口。
- 在方法名部分，可以使用 * 号表示方法名任意。
- 在方法名部分，可以使用 * 号代替方法名的一部分。
    - 例如：*Operation 匹配所有方法名以 Operation 结尾的方法。
- 在方法参数列表部分，使用 (..) 表示参数列表任意。
- 在方法参数列表部分，使用 (int, ..) 表示参数列表以一个 int 类型的参数开头。
- 在方法参数列表部分，基本数据类型和对应的包装类型是不一样的。
    - 切入点表达式中使用 int 和实际方法中 Integer 是不匹配的。
- 在方法返回值部分，如果想要明确指定一个返回值类型，那么必须同时写明权限修

饰符。

- 例如：execution(public int ..Service.*(.., int)) 正确。
- 例如：execution(int ..Service.*(.., int)) 错误。



3.4.6、重用切入点表达式

3.4.6.1 ① 声明

```
// @Pointcut 声明一个公共的切入点表达式
@Pointcut("execution(* com.myxh.spring.aop.annotation.CalculatorImpl.*(..))")
public void pointCut()
{

}
```

3.4.6.2 ② 在同一个切面中使用

```
// @Before("execution(public Integer com.myxh.spring.aop.annotation.CalculatorImpl.add(Integer, Integer))")
// @Before("execution(* com.myxh.spring.aop.annotation.CalculatorImpl.*(..))")
@Before("pointCut()")
public void beforeAdviceMethod(JoinPoint joinPoint)
{
    // 获取连接点所对应方法的签名信息
    Signature signature = joinPoint.getSignature();

    // 获取连接点所对应方法的方法名
    String name = signature.getName();

    // 获取连接点所对应方法的参数
    Object[] args = joinPoint.getArgs();

    System.out.println("[LoggerAspect] 方法: " + name + ", 参数: " + Arrays.toString(args));
}
```

3.4.6.3 ③ 在不同切面中使用

```
package com.myxh.spring.aop.annotation;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

/**
 * @author MYXH
 * @date 2023/8/28
 */
@Component
@Aspect
public class ValidateAspect
{
    // @Before("execution(* com.myxh.spring.aop.annotation.CalculatorImpl.*(..))")
    @Before("com.myxh.spring.aop.annotation.LoggerAspect.pointCut()")
    public void beforeMethod()
    {
        System.out.println("ValidateAspect -> 前置通知");
    }
}
```

3.4.7、获取通知的相关信息

3.4.7.1 ① 获取连接点信息

获取连接点信息可以在通知方法的参数位置设置 JoinPoint 类型的形参。

```
// @Before("execution(public Integer com.myxh.spring.aop.annotation.CalculatorImpl.add(Integer, Int
// @Before("execution(* com.myxh.spring.aop.annotation.CalculatorImpl.*(..))")
@Before("pointCut()")
public void beforeAdviceMethod(JoinPoint joinPoint)
{
    // 获取连接点所对应方法的签名信息
    Signature signature = joinPoint.getSignature();

    // 获取连接点所对应方法的方法名
    String name = signature.getName();

    // 获取连接点所对应方法的参数
    Object[] args = joinPoint.getArgs();

    System.out.println("[LoggerAspect] 方法: " + name + ", 参数: " + Arrays.toString(args));
}
```

3.4.7.2 ② 获取目标方法的返回值

@AfterReturning 中的属性 returning，用来将通知方法的某个形参，接收目标方法的返回值。

```java
/**
 * 在返回通知中若要获取目标对象方法的返回值
 * 只需要通过 {@code @AfterReturning} 注解的 returning 属性
 * 就可以将通知方法的某个参数指定为接收目标对象方法的返回值的参数
 *
 * @param joinPoint 连接点
 * @param result 结果
 */
@AfterReturning(value = "pointCut()", returning = "result")
public void afterReturningAdviceMethod(JoinPoint joinPoint, Object result)
{
    // 获取连接点所对应方法的签名信息
    Signature signature = joinPoint.getSignature();

    // 获取连接点所对应方法的方法名
    String name = signature.getName();

    System.out.println("[LoggerAspect] 方法: " + name + ", 结果: " + result);
}
```

3.4.7.3 ③ 获取目标方法的异常

@AfterThrowing 中的属性 throwing，用来将通知方法的某个形参，接收目标方法的异常。

```java
/**
 * 在异常通知中若要获取目标对象方法的异常
 * 只需要通过 {@code @AfterThrowing} 注解的 throwing 属性
 * 就可以将通知方法的某个参数指定为接收目标对象方法出现的异常的参数
 *
 * @param joinPoint 连接点
 * @param exception 异常
 */
@AfterThrowing(value = "pointCut()", throwing = "exception")
public void afterThrowingAdviceMethod(JoinPoint joinPoint, Throwable exception)
{
    // 获取连接点所对应方法的签名信息
    Signature signature = joinPoint.getSignature();

    // 获取连接点所对应方法的方法名
    String name = signature.getName();

    System.out.println("[LoggerAspect] 方法: " + name + ", 异常: " + exception);
}
```

3.4.8、环绕通知

```
/**
 * 环绕通知的方法的返回值一定要和目标对象方法的返回值一致
 *
 * @param joinPoint 连接点
 * @return result 结果
 */
@Around("pointCut()")
public Object aroundAdviceMethod(ProceedingJoinPoint joinPoint)
{
    Object result = null;

    try
    {
        System.out.println("环绕通知 -> 前置通知");

        // 表示目标对象方法的执行
        result  = joinPoint.proceed();

        System.out.println("环绕通知 -> 返回通知");
    }
    catch (Throwable e)
    {
        System.out.println("环绕通知 -> 异常通知");
        e.printStackTrace();
    }
    finally
    {
        System.out.println("环绕通知 -> 后置通知");
    }

    return result;
}
```

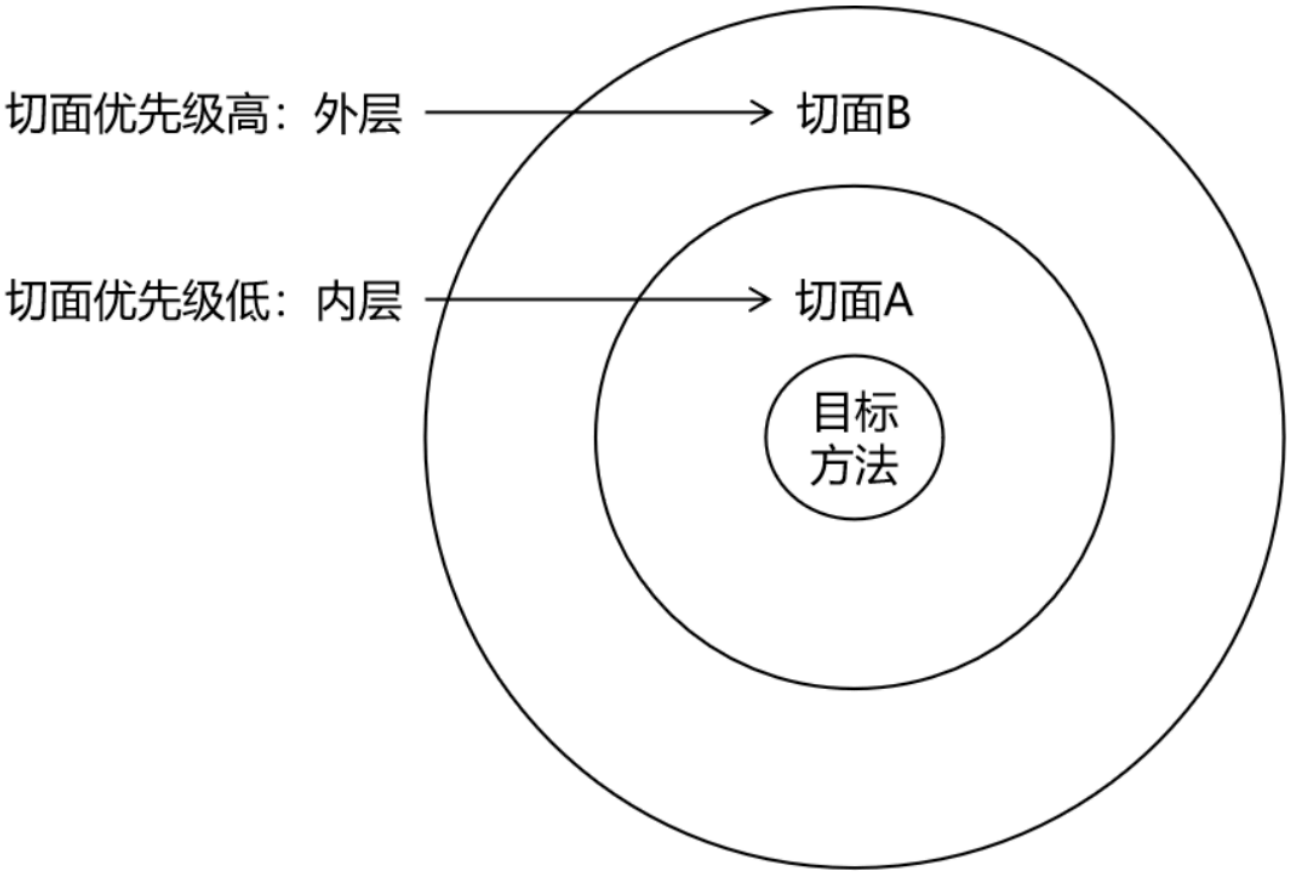
3.4.9、切面的优先级

相同目标方法上同时存在多个切面时，切面的优先级控制切面的**内外嵌套**顺序。

- 优先级高的切面：外面。
- 优先级低的切面：里面。

使用@Order 注解可以控制切面的优先级：

- @Order(较小的数)：优先级高。
- @Order(较大的数)：优先级低。



3.5，基于 XML 的 AOP（了解）

3.5.1、准备工作

参考基于注解的 AOP 环境。

3.5.2、实现

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop" xmlns="http://www.springframework.org
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.
    <!-- 扫描组件 -->
    <context:component-scan base-package="com.myxh.spring.aop.xml"/>

    <aop:config>
        <!-- 设置一个公共的切入点表达式 -->
        <aop:pointcut id="pointCut" expression="execution(* com.myxh.spring.aop.xml.CalculatorImpl

        <!-- 将 IOC 容器中的某个 bean 设置为切面 -->
        <aop:aspect ref="loggerAspect">
            <aop:before method="beforeAdviceMethod" pointcut-ref="pointCut"/>
            <aop:after method="afterAdviceMethod" pointcut-ref="pointCut"/>
            <aop:after-returning method="afterReturningAdviceMethod" returning="result" pointcut-r
            <aop:after-throwing method="afterThrowingAdviceMethod" throwing="exception" pointcut-r
            <aop:around method="aroundAdviceMethod" pointcut-ref="pointCut"/>
        </aop:aspect>

        <aop:aspect ref="validateAspect" order="1">
            <aop:before method="beforeMethod" pointcut-ref="pointCut"/>
        </aop:aspect>
    </aop:config>
</beans>
```

4、声明式事务

4.1、JdbcTemplate

4.1.1、简介

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

4.1.2、准备工作

4.1.2.1 ① 加入依赖

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.myxh.spring</groupId>
    <artifactId>spring_transaction</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <!-- 基于 Maven 依赖传递性，导入 spring-context 依赖即可导入当前所需所有 jar 包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>6.0.9</version>
        </dependency>

        <!-- Spring 持久化层支持 jar 包 -->
        <!-- Spring 在执行持久化层操作、与持久化层技术进行整合过程中，需要使用 orm、jdbc、tx 三个 jar 包
        <!-- 导入 orm 包就可以通过 Maven 的依赖传递性把其他两个也导入 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>6.0.6</version>
        </dependency>

        <!-- Spring 测试相关 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>6.0.6</version>
        </dependency>

        <!-- junit 测试 -->
        <dependency>
            <groupId>junit</groupId>
```

```
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>

        <!-- spring-aspects 会帮我们传递过来 aspectjweaver -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>6.0.9</version>
        </dependency>

        <!-- MySQL驱动 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.33</version>
        </dependency>

        <!-- 数据源 -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.2.16</version>
        </dependency>
    </dependencies>
</project>
```

4.1.2.2 ② 创建 jdbc.properties

```
jdbc.driver=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
jdbc.username=MYXH
jdbc.password=520.ILY!
```

4.1.2.3 ③ 配置 Spring 的配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.
    <!-- 引入外部属性文件 jdbc.properties -->
    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!-- 配置数据源 -->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${jdbc.driver}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <!-- 配置 JdbcTemplate -->
    <bean class="org.springframework.jdbc.core.JdbcTemplate">
        <!-- 装配数据源 -->
        <property name="dataSource" ref="dataSource"/>
    </bean>
</beans>
```

4.1.3、测试

4.1.3.1 ① 在测试类装配 JdbcTemplate

```
package com.myxh.spring.test;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

/**
 * @author MYXH
 * @date 2023/8/29
 */
// 指定当前测试类在 Spring 的测试环境中执行，此时就可以通过注入的方式直接获取 IOC 容器中的 bean
@RunWith(SpringJUnit4ClassRunner.class)
// 设置 spring 测试环境的配置文件
@ContextConfiguration("classpath:spring-jdbc.xml")
public class JdbcTemplateTest
{
    @Autowired
    private JdbcTemplate jdbcTemplate;
}
```

4.1.3.2 ② 测试增删改功能

```
@Test
public void testInsert()
{
    String sql = "insert into t_user (id, username, password, age, gender, email) values (null, ?,
    int result = jdbcTemplate.update(sql, "test", "test", 18, "男", "test@@qq.com");
    System.out.println("result = " + result);
}
```

4.1.3.3 ③ 查询一条数据为实体类对象

```
@Test
public void testGetUserById()
{
    String sql = "select * from t_user where id = ?;";
    User user = jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<>(User.class), 1);
    System.out.println("user = " + user);
}
```

4.1.3.4 ④ 查询多条数据为一个 list 集合

```
@Test
public void testGetAllUser()
{
    String sql = "select * from t_user;";
    List<User> userList = jdbcTemplate.query(sql, new BeanPropertyRowMapper<>(User.class));
    userList.forEach(System.out::println);
}
```

4.1.3.5 ⑤ 查询单行单列的值

```
@Test
public void testGetCount()
{
    String sql = "select count(*) from t_user;";
    Integer count = jdbcTemplate.queryForObject(sql, Integer.class);
    System.out.println("count = " + count);
}
```

4.2、声明式事务概念

4.2.1、编程式事务

事务功能的相关操作全部通过自己编写代码来实现：

```
Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/ssm", "MYXH", "52

try
{
    // 开启事务：关闭事务的自动提交
    connection.setAutoCommit(false);

    // 核心操作

    // 提交事务
    connection.commit();
}
catch(Exception e)
{
    // 回滚事务
    connection.rollBack();
}
finally
{
    // 释放数据库连接
    connection.close();
}
```

编程式的实现方式存在缺陷：

- 细节没有被屏蔽：具体操作过程中，所有细节都需要程序员自己来完成，比较繁琐。
- 代码复用性不高：如果没有有效抽取出来，每次实现功能都需要自己编写代码，代码就没有得到复用。

4.2.2、声明式事务

既然事务控制的代码有规律可循，代码的结构基本是确定的，所以框架就可以将固定模式的代码抽取出来，进行相关的封装。

封装起来后，我们只需要在配置文件中进行简单的配置即可完成操作。

- 好处 1：提高开发效率。
- 好处 2：消除了冗余的代码。
- 好处 3：框架会综合考虑相关领域中在实际开发环境下有可能遇到的各种问题，进行了健壮性、性能等各个方面的优化。

所以，我们可以总结下面两个概念：

- **编程式**：**自己写代码**实现功能。
- **声明式**：通过**配置**让**框架**实现功能。

4.3、基于注解的声明式事务

4.3.1、准备工作

4.3.1.1 ① 加入依赖

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.myxh.spring</groupId>
    <artifactId>spring_transaction</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <!-- 基于 Maven 依赖传递性，导入 spring-context 依赖即可导入当前所需所有 jar 包 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>6.0.9</version>
        </dependency>

        <!-- Spring 持久化层支持 jar 包 -->
        <!-- Spring 在执行持久化层操作、与持久化层技术进行整合过程中，需要使用 orm、jdbc、tx 三个 jar 包
        <!-- 导入 orm 包就可以通过 Maven 的依赖传递性把其他两个也导入 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>6.0.6</version>
        </dependency>

        <!-- Spring 测试相关 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>6.0.6</version>
        </dependency>

        <!-- junit 测试 -->
        <dependency>
            <groupId>junit</groupId>
```

```
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>

        <!-- spring-aspects 会帮我们传递过来 aspectjweaver -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>6.0.9</version>
        </dependency>

        <!-- MySQL驱动 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.33</version>
        </dependency>

        <!-- 数据源 -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.2.16</version>
        </dependency>
    </dependencies>
</project>
```

4.3.1.2 ② 创建 jdbc.properties

```
jdbc.driver=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
jdbc.username=MYXH
jdbc.password=520.ILY!
```

4.3.1.3 ③ 配置 Spring 的配置文件

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.
    <!-- 扫描组件 -->
    <context:component-scan base-package="com.myxh.spring"/>

    <!-- 引入外部属性文件 jdbc.properties -->
    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!-- 配置数据源 -->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${jdbc.driver}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <!-- 配置 JdbcTemplate -->
    <bean class="org.springframework.jdbc.core.JdbcTemplate">
        <!-- 装配数据源 -->
        <property name="dataSource" ref="dataSource"/>
    </bean>
</bean>
```

4.3.1.4 ④ 创建表

```
# 创建图书表 t_book
CREATE TABLE IF NOT EXISTS t_book
(
    book_id   INT(11) NOT NULL AUTO_INCREMENT COMMENT '主键',
    book_name VARCHAR(20)            DEFAULT NULL COMMENT '图书名称',
    price     DECIMAL(8, 2) UNSIGNED DEFAULT NULL COMMENT '价格',
    stock     INT(10) UNSIGNED       DEFAULT NULL COMMENT '库存（无符号）',
    PRIMARY KEY (book_id)
) ENGINE = InnoDB
  AUTO_INCREMENT = 3
  DEFAULT CHARSET = utf8;

# 添加图书表 t_book 的数据
insert into t_book(book_id, book_name, price, stock)
values (1, '三体', 80, 100),
       (2, '小王子', 50, 100);

# 创建书店用户表 t_bookstore_user
CREATE TABLE t_bookstore_user
(
    user_id  INT(11) NOT NULL AUTO_INCREMENT COMMENT '主键',
    username VARCHAR(20)            DEFAULT NULL COMMENT '用户名',
    balance  DECIMAL(8, 2) UNSIGNED DEFAULT NULL COMMENT '余额（无符号）',
    PRIMARY KEY (user_id)
) ENGINE = InnoDB
  AUTO_INCREMENT = 2
  DEFAULT CHARSET = utf8;

# 添加书店用户表 t_bookstore_user 的数据
INSERT INTO t_bookstore_user(user_id, username, balance)
VALUES (1, 'MYXH', 50);
```

4.3.1.5 ⑤ 创建组件

创建 BookController：

```java
package com.myxh.spring.controller;

import com.myxh.spring.service.BookService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

/**
 * @author MYXH
 * @date 2023/8/29
 */
@Controller
public class BookController
{
    @Autowired
    private BookService bookService;

    public void buyBook(Integer userId, Integer bookId)
    {
        bookService.buyBook(userId, bookId);
    }
}
```

创建接口 BookService：

```java
package com.myxh.spring.service;

/**
 * @author MYXH
 * @date 2023/8/29
 */
public interface BookService
{
    /**
     * 买书
     *
     * @param userId 用户 id
     * @param bookId 图书 id
     */
    void buyBook(Integer userId, Integer bookId);
}
```

创建实现类 BookServiceImpl：

```java
package com.myxh.spring.service.impl;

import com.myxh.spring.dao.BookDao;
import com.myxh.spring.service.BookService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;

/**
 * @author MYXH
 * @date 2023/8/29
 */
@Service
public class BookServiceImpl implements BookService
{
    @Autowired
    private BookDao bookDao;

    /**
     * 买书
     *
     * @param userId 用户 id
     * @param bookId 图书 id
     */
    @Override
    public void buyBook(Integer userId, Integer bookId)
    {
        // 查询图书的价格
        BigDecimal price = bookDao.getPriceByBookId(bookId);

        // 更新图书的库存
        bookDao.updateStock(bookId);

        // 更新用户的余额
        bookDao.updateBalance(userId, price);
    }
}
```

创建接口 BookDao：

```java
package com.myxh.spring.dao;

import java.math.BigDecimal;

/**
 * @author MYXH
 * @date 2023/8/29
 */
public interface BookDao
{
    /**
     * 根据图书的 id 查询图书的价格
     *
     * @param bookId 图书 id
     * @return 图书价格
     */
    BigDecimal getPriceByBookId(Integer bookId);

    /**
     * 更新图书的库存
     *
     * @param bookId 图书 id
     */
    void updateStock(Integer bookId);

    /**
     * 更新用户的余额
     *
     * @param userId 用户 id
     * @param price  图书价格
     */
    void updateBalance(Integer userId, BigDecimal price);
}
```

创建实现类 BookDaoImpl：

```java
package com.myxh.spring.dao.impl;

import com.myxh.spring.dao.BookDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import java.math.BigDecimal;

/**
 * @author MYXH
 * @date 2023/8/29
 */
@Repository
public class BookDaoImpl implements BookDao
{
    @Autowired
    private JdbcTemplate jdbcTemplate;

    /**
     * 根据图书的 id 查询图书的价格
     *
     * @param bookId 图书 id
     * @return 图书价格
     */
    @Override
    public BigDecimal getPriceByBookId(Integer bookId)
    {
        String sql = "select price from t_book where book_id = ?;";
        BigDecimal price = jdbcTemplate.queryForObject(sql, BigDecimal.class, bookId);

        return price;
    }

    /**
     * 更新图书的库存
     *
     * @param bookId 图书 id
     */
    @Override
    public void updateStock(Integer bookId)
    {
        String sql = "update t_book set stock = stock - 1 where book_id = ?;";
        jdbcTemplate.update(sql,bookId);
```

```
    }

    /**
     * 更新用户的余额
     *
     * @param userId 用户 id
     * @param price  图书价格
     */
    @Override
    public void updateBalance(Integer userId, BigDecimal price)
    {
        String sql = "update t_bookstore_user set balance = balance - ? where user_id = ?;";
        jdbcTemplate.update(sql,price, userId);
    }
}
```

4.3.2、测试无事务情况

4.3.2.1 ① 创建测试类

```
package com.myxh.spring.test;

import com.myxh.spring.controller.BookController;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

/**
 * @author MYXH
 * @date 2023/8/29
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:tx-annotation.xml")
public class TxByAnnotationTest
{
    @Autowired
    private BookController bookController;

    @Test
    public void testBuyBook()
    {
        bookController.buyBook(1,1);
    }
}
```

4.3.2.2 ② 模拟场景

用户购买图书，先查询图书的价格，再更新图书的库存和用户的余额。

假设用户 id 为 1 的用户，购买 id 为 1 的图书。

用户余额为 50，而图书价格为 80。

购买图书之后，用户的余额为-30，数据库中余额字段设置了无符号，因此无法将-30 插入到余额字段。

此时执行 sql 语句会抛出 SQLException。

4.3.2.3 ③ 观察结果

因为没有添加事务，图书的库存更新了，但是用户的余额没有更新。

显然这样的结果是错误的，购买图书是一个完整的功能，更新库存和更新余额要么都成功要么都失败。

4.3.3、加入事务

4.3.3.1 ① 添加事务配置

在 Spring 的配置文件中添加配置：

```
<!-- 配置事务管理器 -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionMana
    <property name="dataSource" ref="dataSource"/>
</bean>

<!--
开启事务的注解驱动
将使用 @Transactional 注解所标识的方法或类中所有的方法使用事务进行管理
transaction-manager 属性设置事务管理器的 id
若事务管理器的 bean 的 id 默认为 transactionManager，则该属性可以不写
    -->
<tx:annotation-driven transaction-manager="transactionManager"/>
```

注意：导入的名称空间需要 **tx 结尾**的那个。

4.3.3.2 ② 添加事务注解

因为 service 层表示业务逻辑层，一个方法表示一个完成的功能，因此处理事务一般在 service

层处理。

在 BookServiceImpl 的 buybook()添加注解@Transactional。

4.3.3.3 ③ 观察结果

由于使用了 Spring 的声明式事务，更新库存和更新余额都没有执行。

4.3.4、@Transactional 注解标识的位置

@Transactional 标识在方法上，咋只会影响该方法。

@Transactional 标识的类上，咋会影响类中所有的方法。

4.3.5、事务属性：只读

4.3.5.1 ① 介绍

对一个查询操作来说，如果我们把它设置成只读，就能够明确告诉数据库，这个操作不涉及写操作。这样数据库就能够针对查询操作来进行优化。

4.3.5.2 ② 使用方式

```java
package com.myxh.spring.service.impl;

import com.myxh.spring.dao.BookDao;
import com.myxh.spring.service.BookService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;

/**
 * @author MYXH
 * @date 2023/8/29
 */
@Service
public class BookServiceImpl implements BookService
{
    @Autowired
    private BookDao bookDao;

    /**
     * 买书
     *
     * @param userId 用户 id
     * @param bookId 图书 id
     */
    @Override

    @Transactional(readOnly = true)
    public void buyBook(Integer userId, Integer bookId)
    {
        // 查询图书的价格
        BigDecimal price = bookDao.getPriceByBookId(bookId);

        // 更新图书的库存
        bookDao.updateStock(bookId);

        // 更新用户的余额
        bookDao.updateBalance(userId, price);
    }
}
```

4.3.5.3 ③ 注意

对增删改操作设置只读会抛出下面异常：

> Caused by: java.sql.SQLException: Connection is read-only. Queries leading to data modification are not allowed

4.3.6、事务属性：超时

4.3.6.1 ① 介绍

事务在执行过程中，有可能因为遇到某些问题，导致程序卡住，从而长时间占用数据库资源。而长时间占用资源，大概率是因为程序运行出现了问题（可能是 Java 程序或 MySQL 数据库或网络连接等等）。

此时这个很可能出问题的程序应该被回滚，撤销它已做的操作，事务结束，把资源让出来，让其他正常程序可以执行。

概括来说就是一句话：超时回滚，释放资源。

4.3.6.2 ② 使用方式

```java
package com.myxh.spring.service.impl;

import com.myxh.spring.dao.BookDao;
import com.myxh.spring.service.BookService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;

/**
 * @author MYXH
 * @date 2023/8/29
 */
@Service
public class BookServiceImpl implements BookService
{
    @Autowired
    private BookDao bookDao;

    /**
     * 买书
     *
     * @param userId 用户 id
     * @param bookId 图书 id
     */
    @Override
    @Transactional(
            // readOnly = true
            timeout = 3
    )
    public void buyBook(Integer userId, Integer bookId)
    {
        try
        {
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e)
        {
            e.printStackTrace();
        }

        // 查询图书的价格
        BigDecimal price = bookDao.getPriceByBookId(bookId);

        // 更新图书的库存
```

```
        bookDao.updateStock(bookId);

        // 更新用户的余额
        bookDao.updateBalance(userId, price);
    }
}
```

4.3.6.3 ③ 观察结果

执行过程中抛出异常：

> org.springframework.transaction.**TransactionTimedOutException**: Transaction timed out: deadline was Fri Aug 29 12:00:00 CST 2023

4.3.7、事务属性：回滚策略

4.3.7.1 ① 介绍

声明式事务默认只针对运行时异常回滚，编译时异常不回滚。

可以通过@Transactional 中相关属性设置回滚策略

- rollbackFor 属性：需要设置一个 Class 类型的对象。
- rollbackForClassName 属性：需要设置一个字符串类型的全类名。
- noRollbackFor 属性：需要设置一个 Class 类型的对象。
- rollbackFor 属性：需要设置一个字符串类型的全类名。

4.3.7.2 ② 使用方式

```java
package com.myxh.spring.service.impl;

import com.myxh.spring.dao.BookDao;
import com.myxh.spring.service.BookService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;

/**
 * @author MYXH
 * @date 2023/8/29
 */
@Service
public class BookServiceImpl implements BookService
{
    @Autowired
    private BookDao bookDao;

    /**
     * 买书
     *
     * @param userId 用户 id
     * @param bookId 图书 id
     */
    @Override
    @Transactional(
            // readOnly = true
            // timeout = 3
            noRollbackFor = ArithmeticException.class
            // noRollbackForClassName = "java.lang.ArithmeticException"
    )
    public void buyBook(Integer userId, Integer bookId)
    {
        /*
        try
        {
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e)
        {
            e.printStackTrace();
        }
         */
```

```
        // 查询图书的价格
        BigDecimal price = bookDao.getPriceByBookId(bookId);

        // 更新图书的库存
        bookDao.updateStock(bookId);

        // 更新用户的余额
        bookDao.updateBalance(userId, price);

        System.out.println(1 / 0);
    }
}
```

4.3.7.3 ③ 观察结果

虽然购买图书功能中出现了数学运算异常（ArithmeticException），但是我们设置的回滚策略是，当出现 ArithmeticException 不发生回滚，因此购买图书的操作正常执行。

4.3.8、事务属性：事务隔离级别

4.3.8.1 ① 介绍

数据库系统必须具有隔离并发运行各个事务的能力，使它们不会相互影响，避免各种并发问题。一个事务与其他事务隔离的程度称为隔离级别。SQL 标准中规定了多种事务隔离级别，不同隔离级别对应不同的干扰程度，隔离级别越高，数据一致性就越好，但并发性越弱。

隔离级别一共有四种：

- 读未提交：READ UNCOMMITTED

允许 Transaction01 读取 Transaction02 未提交的修改。

- 读已提交：READ COMMITTED

要求 Transaction01 只能读取 Transaction02 已提交的修改。

- 可重复读：REPEATABLE READ

确保 Transaction01 可以多次从一个字段中读取到相同的值，即 Transaction01 执行期间禁止其它事务对这个字段进行更新。

- 串行化：SERIALIZABLE

确保 Transaction01 可以多次从一个表中读取到相同的行，在 Transaction01 执行期间，禁止其它事务对这个表进行添加、更新、删除操作。可以避免任何并发问题，但性能十分低下。

各个隔离级别解决并发问题的能力见下表：

隔离级别	脏读	不可重复读	幻读
READ UNCOMMITTED	有	有	有
READ COMMITTED	无	有	有
REPEATABLE READ	无	无	有
SERIALIZABLE	无	无	无

各种数据库产品对事务隔离级别的支持程度：

隔离级别	Oracle	MySQL
READ UNCOMMITTED	×	√
READ COMMITTED	√(默认)	√
REPEATABLE READ	×	√(默认)
SERIALIZABLE	√	√

4.3.8.2 ② 使用方式

```
// 使用数据库默认的隔离级别
@Transactional(isolation = Isolation.DEFAULT)

// 读未提交
@Transactional(isolation = Isolation.READ_UNCOMMITTED)

// 读已提交
@Transactional(isolation = Isolation.READ_COMMITTED)

// 可重复读
@Transactional(isolation = Isolation.REPEATABLE_READ)

// 串行化
@Transactional(isolation = Isolation.SERIALIZABLE)
```

4.3.9、事务属性：事务传播行为

4.3.9.1 ① 介绍

当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。

4.3.9.2 ② 测试

创建接口 CheckoutService：

```
package com.myxh.spring.service;

/**
 * @author MYXH
 * @date 2023/8/29
 */
public interface CheckoutService
{
    /**
     * 结账
     *
     * @param userId 用户 id
     * @param bookIds 一些图书 id
     */
    void checkout(Integer userId, Integer[] bookIds);
}
```

创建实现类 CheckoutServiceImpl：

```java
package com.myxh.spring.service.impl;

import com.myxh.spring.service.BookService;
import com.myxh.spring.service.CheckoutService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

/**
 * @author MYXH
 * @date 2023/8/29
 */
@Service
public class CheckoutServiceImpl implements CheckoutService
{
    @Autowired
    private BookService bookService;

    /**
     * 结账
     *
     * @param userId 用户 id
     * @param bookIds 一些图书 id
     */
    @Override
    // @Transactional
    public void checkout(Integer userId, Integer[] bookIds)
    {
        for (Integer bookId : bookIds)
        {
            bookService.buyBook(userId,bookId);
        }
    }
}
```

在 BookController 中添加方法：

```java
@Autowired
private CheckoutService checkoutService;

public void checkout(Integer[] bookIds, Integer userId)
{
    checkoutService.checkout(bookIds, userId);
}
```

在数据库中将用户的余额修改为 100 元。

4.3.9.3 ③ 观察结果

可以通过@Transactional 中的 propagation 属性设置事务传播行为。

修改 BookServiceImpl 中 buyBook()上，注解@Transactional 的 propagation 属性。

@Transactional(propagation = Propagation.REQUIRED)，默认情况，表示如果当前线程上有已经开启的事务可用，那么就在这个事务中运行。经过观察，购买图书的方法 buyBook()在 checkout()中被调用，checkout()上有事务注解，因此在此事务中执行。所购买的两本图书的价格为 80 和 50，而用户的余额为 100，因此在购买第二本图书时余额不足失败，导致整个 checkout()回滚，即只要有一本书买不了，就都买不了。

@Transactional(propagation = Propagation.REQUIRES_NEW)，表示不管当前线程上是否有已经开启的事务，都要开启新事务。同样的场景，每次购买图书都是在 buyBook()的事务中执行，因此第一本图书购买成功，事务结束，第二本图书购买失败，只在第二次的 buyBook()中回滚，购买第一本图书不受影响，即能买几本就买几本。

4.4、基于 XML 的声明式事务

4.3.1、场景模拟

参考基于注解的声明式事务。

4.3.2、修改 Spring 配置文件

将 Spring 配置文件中去掉 tx:annotation-driven 标签，并添加配置：

```xml
<!-- 配置事务管理器 -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionMana
    <property name="dataSource" ref="dataSource"/>
</bean>

<!-- 配置事务通知 -->
<tx:advice id="tx" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="buyBook" propagation="REQUIRES_NEW"/>
        <tx:method name="checkout"/>
        <tx:method name="*"/>
    </tx:attributes>
</tx:advice>

<!-- 配置切入点表达式 -->
<aop:config>
    <aop:advisor advice-ref="tx" pointcut="execution(* com.myxh.spring.service.impl.*.*(..))"/>
</aop:config>
```

> 注意：基于 xml 实现的声明式事务，必须引入 aspectJ 的依赖。
>
> ```xml
> <!-- spring-aspects 会帮我们传递过来 aspectjweaver -->
> <dependency>
>     <groupId>org.springframework</groupId>
>     <artifactId>spring-aspects</artifactId>
>     <version>6.0.9</version>
> </dependency>
> ```