二、Spring

1、Spring简介

1.1、Spring概述

官网网址：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 Framwork

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 Framwork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(控制反转)

2.1、IOC容器

2.1.1、IOC思想

IOC：Inversion of Control，翻译过来是反转控制。

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、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。

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。

依赖注入：为类中的属性进行赋值。

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：

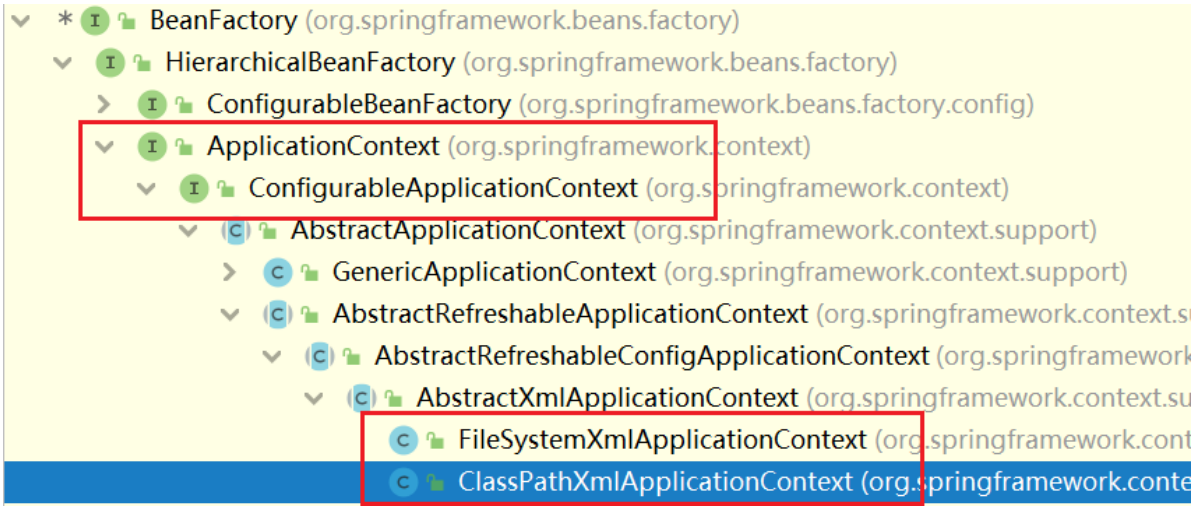
1. BeanFactory

   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. 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。

ApplicationContext的主要实现类如下图：

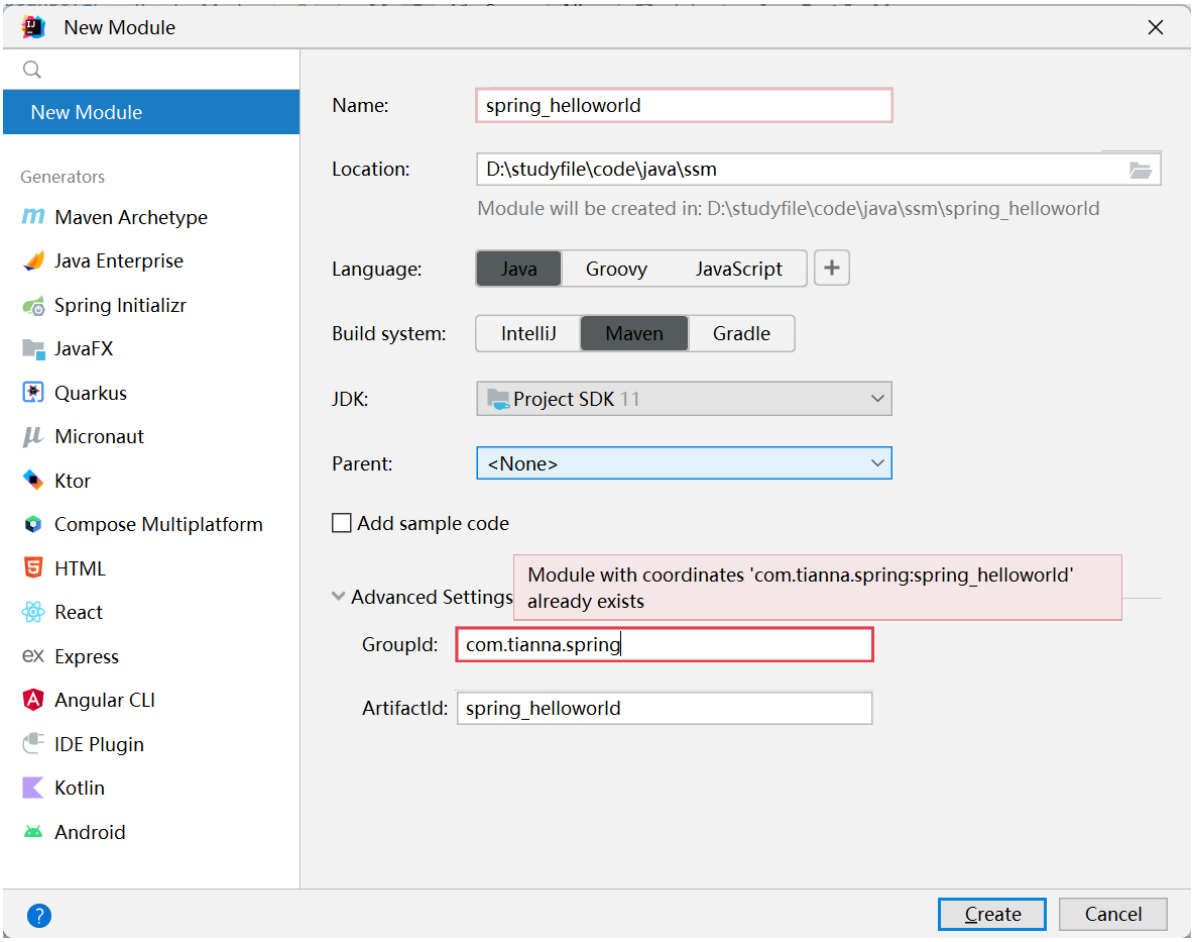


类型名	简介
ClassPathXmlApplicationContext	通过读取类路径下的 XML 格式的配置文件创建 IOC 容器对象（常用）
FileSystemXmlApplicationContext	通过文件系统路径读取 XML 格式的配置文件创建 IOC 容器对象（不常用）
ConfigurableApplicationContext	ApplicationContext 的子接口，包含一些扩展方法 refresh() 和 close()，让 ApplicationContext 具有启动、关闭和刷新上下文的能力。
WebApplicationContext	专门为 Web 应用准备，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

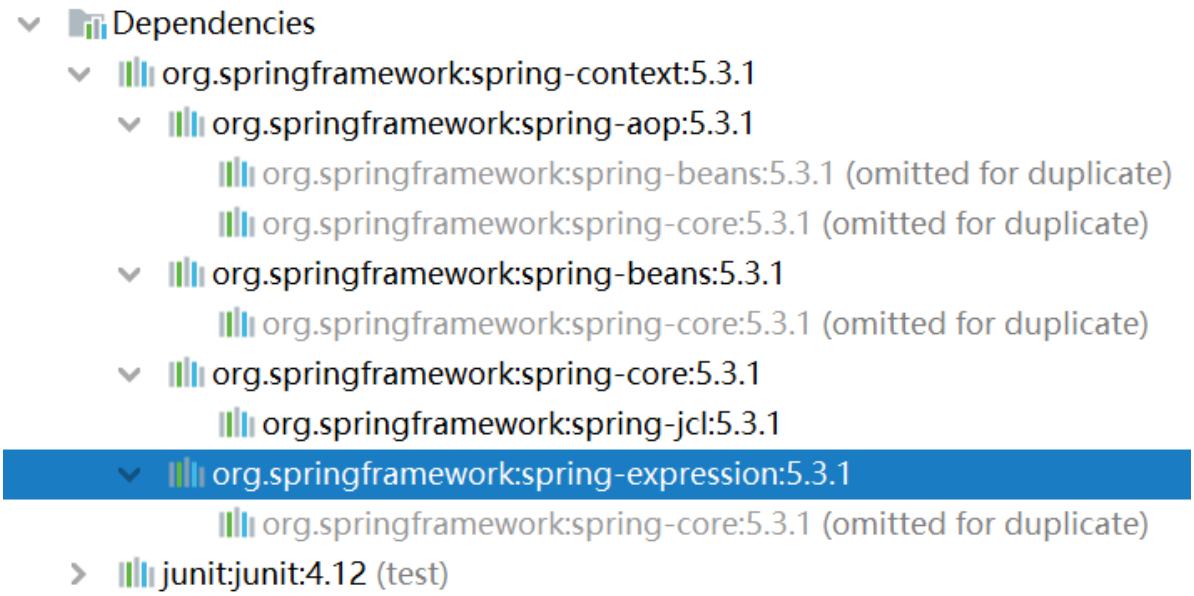
1、创建一个Maven Module



2、在pom.xml文件中引入依赖

```
<packaging>jar</packaging>

<dependencies>
    <!--基于Maven依赖特性，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>5.3.1</version>
    </dependency>
    <!--junit测试-->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

基于Maven依赖特性，在引入`基于Maven依赖特性`时，还会简介引入如下其他的包：



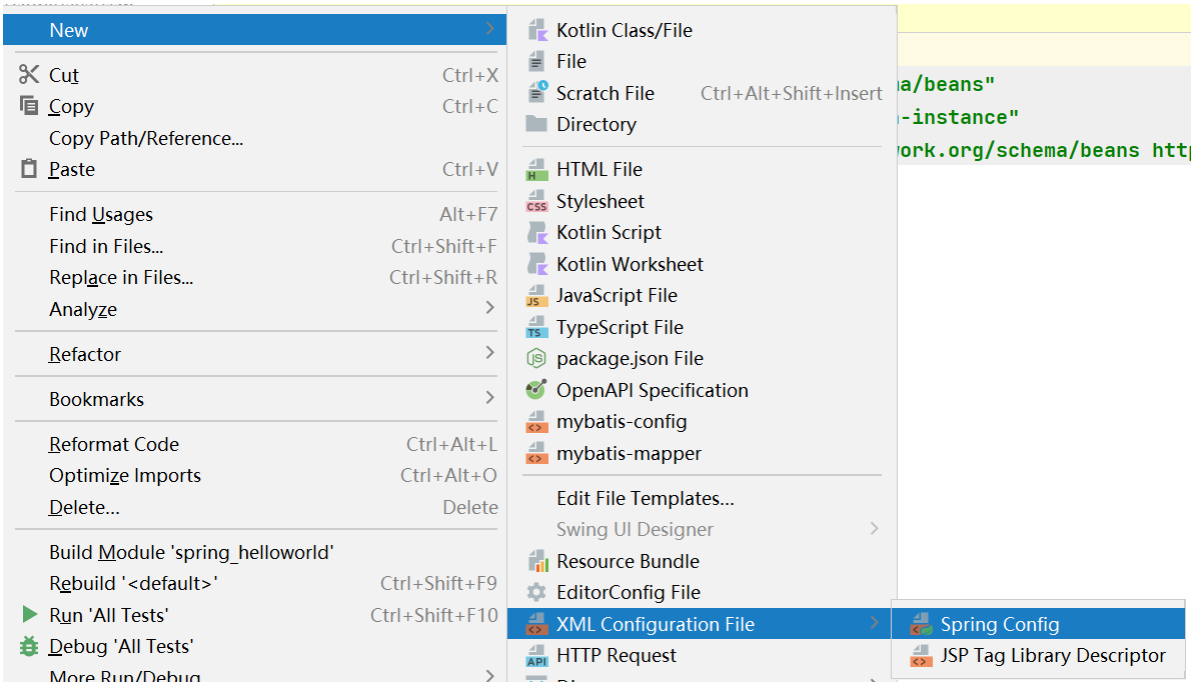
3、 创建类

在`src/main/java`目录下创建包`com.tianna.spring.pojo`，并在该包下创建一个HelloWorld类，如下：

```
public class HelloWorld {
    public void sayHelo(){
        System.out.println("hello spring");
    }
}
```

4、创建Spring的配置文件

在`src/main/resources`目录下按照如下图的方式创建名为`applicationContext`的Spring配置文件。



内容如下：

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

</beans>
```

5、在Spring的配置文件中配置bean

使用bean标签在Spring的配置文件中配置bean，标签及其常用属性的作用如下：

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：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.tianna.spring.pojo.HelloWorld"></bean>
```

6、创建测试类测试

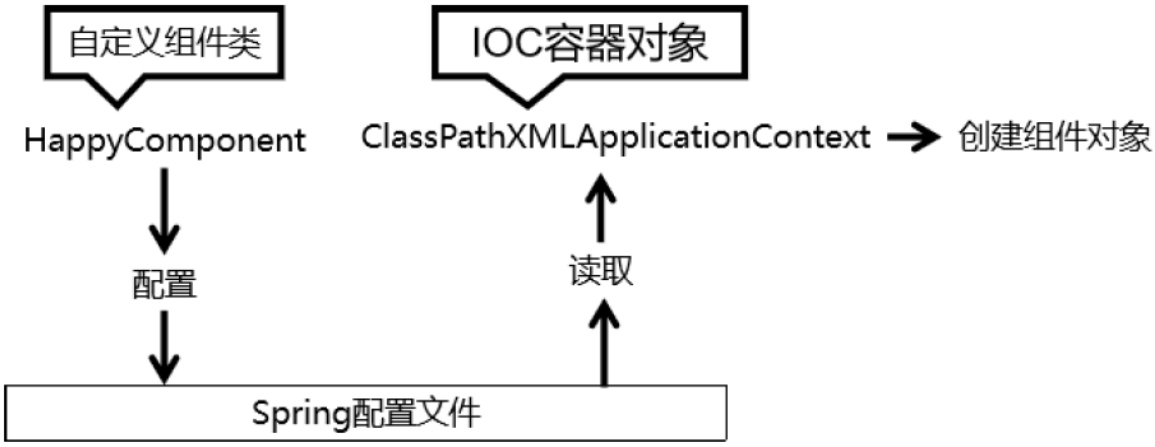
在`src/test/java`目录下创建包`com.tianna.spring.test`，并在该包下创建测试类`test`，测试创建ioc容器，获取ioc容器中的bean，然后调用bean对象中的方法。

```
@Test
public void test(){
    //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");
    //调用方法
    helloworld.sayHelo();
}
```

输出结果为：

```
hello spring
```

7、思路



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。因此需创建一个无参构造器。

2.2.2、实验二：获取bean

获取bean的三种方式：

- 根据bean的id获取
- 根据bean的类型获取
- 根据bean的类型和id获取

1、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。

此时获取的对象为`Object`类型，还需要对其进行类型强制转换。

```
@Test
public void test(){
    //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");
    //调用方法
    helloworld.sayHelo();
}
```

2、根据类型获取（常用该方法）

```
@Test
public void test(){
    //获取ioc容器
    ApplicationContext ioc = new
ClassPathXmlApplicationContext("applicationContext.xml");
    //获取IOC容器中的bean
    HelloWorld helloWorld = ioc.getBean(HelloWorld.class);
    //调用方法
    helloWorld.sayHelo();
}
```

> 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容器中有多个类型匹配的bean时：

```
<bean id="helloworldOne" class="com.tianna.spring.pojo.HelloWorld"></bean>
<bean id="helloworldTwo" class="com.tianna.spring.pojo.HelloWorld"></bean>
```

由于是根据类型获取bean，因此无法确定是获取哪一个bean，因此会产生`NoUniqueBeanDefinitionException`错误。

当IOC容器中没有任何一个类型匹配的bean时：会产生`NoSuchBeanDefinitionException`错误。

3、根据id和类型

此时获取的对象就不需要强制转换为指定类型。

```
@Test
public void test(){
    //获取ioc容器
    ApplicationContext ioc = new
ClassPathXmlApplicationContext("applicationContext.xml");
    //获取IOC容器中的bean
    HelloWorld helloworld = ioc.getBean("helloworld", HelloWorld.class);
    //调用方法
    helloworld.sayHelo();
}
```

4、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

5、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。

2.2.3、实验三、依赖注入之setter注入

使用组件类中的setXXX()方法给组件对象设置属性。

1、首先创建一个学生类

```
public class Student {
    private Integer sid;
    private String sname;
    private Integer age;
    private String gender;

    public Student() {
    }

    public Student(Integer sid, String sname, Integer age, String gender) {
        this.sid = sid;
        this.sname = sname;
        this.age = age;
        this.gender = gender;
    }

    public Integer getSid() {
        return sid;
    }

    public void setSid(Integer sid) {
        this.sid = sid;
    }

    public String getSname() {
        return sname;
    }

    public void setSname(String sname) {
        this.sname = sname;
    }

    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{" +
                "sid=" + sid +
                ", sname='" + sname + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\'' +
                '}';
    }
}
```

2、设置bean时为属性赋值

所使用的标签及其属性为：

property标签：使用组件类中的setXxx()方法给组件对象设置属性。

- name属性：指定属性名（属性名是getXxx()、setXxx()方法定义的，和成员变量无关）
- value属性：指定属性值

```
<bean id="studentOne" class="com.tianna.spring.pojo.Student">
        <property name="sid" value="1001"/>
        <property name="sname" value="张三"/>
        <property name="age" value="23"/>
        <property name="gender" value="男"/>
 </bean>
```

3、测试

对setter注入进行测试：

```
@Test
public void testDI(){
    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-
ioc.xml");
    Student studentOne = ioc.getBean("studentOne",Student.class);
    System.out.println(studentOne);
}
```

结果如下：

```
Student{sid=1001, sname='张三', age=23, gender='男'}
```

2.2.4、实验四：依赖注入之构造器注入

使用组件类中的构造方法给组件对象设置属性。因此组件类中需要添加有参构造方法。

1、在Student类中添加有参构造

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

2、配置bean

所使用的标签及其属性为：

constructor-arg标签：使用构造方法设置参数

- index属性：指定参数所在位置的索引（从0开始），可不设置。
- name属性：指定参数名：可不设置。
- value属性：指定属性值。

```
<!--依赖注入之构造器注入-->
<bean id="studentTwo" class="com.tianna.spring.pojo.Student">
        <constructor-arg value="1002"/>
        <constructor-arg value="李四"/>
        <constructor-arg value="33"/>
        <constructor-arg value="女"/>
</bean>
```

3、测试

对构造器注入进行测试：

```
@Test
public void testDI(){
    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-
ioc.xml");
    Student studentOne = ioc.getBean("studentTwo",Student.class);
    System.out.println(studentOne);
}
```

结果如下：

```
Student{sid=1002, sname='李四', age=33, gender='女'}
```

2.2.5、实验五：特殊值处理

1、字面量赋值

字面量包括：基本数据类型及其对应的包装类、String类型。

对字面量进行赋值时，需要使用`value`属性，Spring会把`value`属性的值看做字面量。

```
<property name="sname" value="张三"/>
```

2、null值

null值使用下面的方法进行赋值：

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

注意：

以下写法，为sname所赋的值是字符串null

```
<property name="sname" value="null"></property>
```

3、xml实体

在xml文件中，有些符号有这特殊的作用，不能随便使用，例如`<`符号用来定义标签的开始，`>`符号用来定义标签的结束，因此在xml不能直接用于赋值，可以使用XML实体来代替。

给sname属性赋值为<王五>的写法如下。

```

<!--
    常用的xml实体：
        <：&lt;
        >：&gt;
-->
<property name="sname" value="&lt;王五&gt;"/>
```

4、CDATA节

CDATA节其中的内容会原样，使用`<![CDATA[...]]>`将所赋的值包含起来。

```
<property name="sname">
        <value>
                <![CDATA[<王五>]]>
        </value>
</property>
```

> 注意：CDATA节是xml中一个特殊的标签，因此不能写在一个属性中，需要写在标签中。

2.2.6、实验六：为类类型属性赋值

1、创建班级类

创建班级类`Clazz`，该类具有以下属性和方法：

```
public class Clazz {
    private Integer cid;
    private String cname;

    public Clazz() {
    }

    public Clazz(Integer cid, String cname) {
        this.cid = cid;
        this.cname = cname;
    }

    public Integer getCid() {
        return cid;
    }

    public void setCid(Integer cid) {
        this.cid = cid;
    }

    public String getCname() {
        return cname;
    }

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

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

2、修改Student类

在Student类中添加Clazz属性及其set和get方法：

```
private Clazz clazz;

public Clazz getClazz() {
    return clazz;
}

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

3、方式一：引用外部已声明的bean

配置Clazz类型的bean，并在Student中使用`ref`属性为clazz属性赋值

> ref属性：引用IOC容器中某个bean的id，将所对应的bean为属性赋值。

```
<bean id="sstudentSFive1" class="com.tianna.spring.pojo.Student">
        <property name="sid" value="1005"/>
        <property name="sname" value="赵六"/>
        <property name="age" value="27"/>
        <property name="gender" value="男"/>
        <!-- ref属性：引用IOC容器中某个bean的id，将所对应的bean为属性赋值 -->
        <property name="clazz" ref="clazzOne"/>
</bean>

<bean id="clazzOne" class="com.tianna.spring.pojo.Clazz">
        <property name="cid" value="1111"/>
        <property name="cname" value="最强王者班"/>
 </bean>
```

4、方式二：级联赋值

级联的方式，需要保证提前为clazz属性赋值或者实例化。**因此该种方式不常用。**

```
<!--级联赋值-->
<bean id="sstudentSFive2" class="com.tianna.spring.pojo.Student">
        <property name="sid" value="1005"/>
        <property name="sname" value="赵六"/>
        <property name="age" value="27"/>
        <property name="gender" value="男"/>
        <!-- ref属性：引用IOC容器中某个bean的id，将所对应的bean为属性赋值 -->
        <property name="clazz" ref="clazzOne"/>
        <!--级联的方式，要保证提前为clazz属性赋值或者实例化-->
        <property name="clazz.cid" value="2222"/>
        <property name="clazz.cname" value="牛逼班"/>
</bean>

<bean id="clazzOne" class="com.tianna.spring.pojo.Clazz">
        <property name="cid" value="1111"/>
        <property name="cname" value="最强王者班"/>
 </bean>
```

5、方式三、内部bean

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。

```
<!--内部bean-->
<bean id="sstudentSFive3" class="com.tianna.spring.pojo.Student">
        <property name="sid" value="1005"/>
        <property name="sname" value="赵六"/>
        <property name="age" value="27"/>
        <property name="gender" value="男"/>
        <property name="clazz">
                <bean id="clazzInner" class="com.tianna.spring.pojo.Clazz">
                        <property name="cid" value="3333"/>
                        <property name="cname" value="哈哈班"/>
                </bean>
        </property>
</bean>
```

测试方法和结果如下：

```
@Test
public void testDI(){
    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-
ioc.xml");
    Student studentOne = ioc.getBean("studentFive3",Student.class);
    System.out.println(studentOne);
}
```

```
Student{sid=1005, sname='赵六', age=27, gender='男', clazz=Clazz{cid=3333,
cname='哈哈班'}}
```

2.2.7、实验七：为数组类型属性赋值

1、修改Student类

在Student类中添加一个数组类型的属性：

```
private String[] hobby;

public String[] getHobby() {
    return hobby;
}

public void setHobby(String[] hobby) {
    this.hobby = hobby;
}
```

2、配置bean

为数组类型赋值时，需要在`property`标签中使用`array`标签，在`array`标签中：

`value`：若为字面量属性赋值使用该标签。

`ref`：若为类类型属性赋值，使用该标签引用外部bean。

详细代码如下：

```
<bean id="studentSix1" class="com.tianna.spring.pojo.Student">
        <property name="sid" value="1005"/>
        <property name="sname" value="赵六"/>
        <property name="age" value="27"/>
        <property name="gender" value="男"/>
        <!-- ref属性：引用IOC容器中某个bean的id，将所对应的bean为属性赋值 -->
        <property name="clazz" ref="clazzOne"/>
        <property name="hobby">
                <!--为数组类型属性赋值-->
                <array>
                        <value>象棋</value>
                        <value>围棋</value>
                        <value>五子棋</value>
                </array>
        </property>
</bean>
<bean id="clazzOne" class="com.tianna.spring.pojo.Clazz">
        <property name="cid" value="1111"/>
        <property name="cname" value="最强王者班"/>
 </bean>
```

2.2.8、实验八：为集合类型属性赋值

1、为list集合类型属性赋值

在Clazz类中添加list集合类型的属性：

```
private List<Student> students;

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

public Integer getCid() {
    return cid;
}
```

第一种方法：使用`list`标签创建内部bean为list集合类型的属性赋值。其中`list`标签中有`ref`和`value`标签，分别为类类型属性和字面量类型属性赋值。下面演示为引用外部bean为类类型属性赋值：

```
<bean id="clazzTwo1" class="com.tianna.spring.pojo.Clazz">
        <property name="cid" value="3333"/>
        <property name="cname" value="123班"/>
        <property name="students">
                <list>
                        <ref bean="studentOne"></ref>
                        <ref bean="studentTwo"></ref>
                        <ref bean="studentThree"></ref>
                </list>
        </property>
</bean>
```

第二种方法：引用外部list集合类型的bean为list集合类型的属性赋值。

配置一个集合类型的bean，需要使用utils的约束（命名空间），因此需要在约束中添加utils命名空间，添加的命名空间为：

```
xmlns:util="http://www.springframework.org/schema/util"
```

创建一个外部list集合类型的bean，并引用外部bean为list集合类型的属性赋值代码如下：

```
<bean id="clazzTwo2" class="com.tianna.spring.pojo.Clazz">
        <property name="cid" value="3333"/>
        <property name="cname" value="123班"/>
        <property name="students" ref="studentList"/>
</bean>

<util:list id="studentList">
        <ref bean="studentOne"/>
        <ref bean="studentTwo"/>
        <ref bean="studentThree"/>
</util:list>
```

2、为Map集合类型属性赋值

创建教师类：

```
 */
public class Teacher {
    private Integer tid;
    private String tname;

    public Teacher() {
    }

    public Teacher(Integer tid, String tname) {
        this.tid = tid;
        this.tname = tname;
    }
```

```
    public Integer getTid() {
        return tid;
    }

    public void setTid(Integer tid) {
        this.tid = tid;
    }

    public String getTname() {
        return tname;
    }

    public void setTname(String tname) {
        this.tname = tname;
    }

    @Override
    public String toString() {
        return "Teacher{" +
                "tid=" + tid +
                ", tname='" + tname + '\'' +
                '}';
    }
}
```

在Student类中添加Map集合类型的属性，以教师科目为键，教师类为值。

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

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

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

第一种方法：使用`map`标签创建内部bean为map集合类型的属性赋值。`map`标签内为`entry`标签，`entry`中包含了map中的键和值，`entry`标签还有以下四个标签，起作用分别为：

- key：键-类型为字面量类型
- key-ref：键-类型为类类型，引用外部bean
- value：值-类型为字面量类型
- value-ref：值-类型为类类型，引用外部bean

使用内部bean的代码如下：

```
<!--为Map集合类型赋值-->
<bean id="studentSeven1" class="com.tianna.spring.pojo.Student">
        <property name="sid" value="1005"/>
        <property name="sname" value="赵六"/>
```

```
        <property name="age" value="27"/>
        <property name="gender" value="男"/>
        <!--内部bean -->
        <property name="clazz">
                <bean id="clazzInner" class="com.tianna.spring.pojo.Clazz">
                        <property name="cid" value="3333"/>
                        <property name="cname" value="哈哈班"/>
                </bean>
        </property>
        <property name="hobby">
                <array>
                        <value>象棋</value>
                        <value>围棋</value>
                        <value>五子棋</value>
                </array>
        </property>
        <property name="teacherMap">
                <map>
                        <entry key="英语老师" value-ref="teacherOne"/>
                        <entry key="数学老师" value-ref="teacherTwo"/>
                </map>
        </property>
</bean>

<bean id="teacherOne" class="com.tianna.spring.pojo.Teacher">
        <property name="tid" value="10086"/>
        <property name="tname" value="大宝"/>
</bean>
<bean id="teacherTwo" class="com.tianna.spring.pojo.Teacher">
        <property name="tid" value="10087"/>
        <property name="tname" value="小宝"/>
</bean>
```

第二种方法：引用外部map集合类型的bean为map集合类型的属性赋值。

创建一个外部map集合类型的bean，并引用外部bean为list集合类型的属性赋值代码如下：

```
<bean id="studentSeven2" class="com.tianna.spring.pojo.Student">
        <property name="sid" value="1005"/>
        <property name="sname" value="赵六"/>
        <property name="age" value="27"/>
        <property name="gender" value="男"/>
        <!--内部bean -->
        <property name="clazz">
                <bean id="clazzInner" class="com.tianna.spring.pojo.Clazz">
                        <property name="cid" value="3333"/>
                        <property name="cname" value="哈哈班"/>
                </bean>
        </property>
        <property name="hobby">
                <array>
```

```
                        <value>象棋</value>
                        <value>围棋</value>
                        <value>五子棋</value>
                </array>
        </property>
        <property name="teacherMap" ref="teacherMap"></property>
</bean>

<util:map id="teacherMap">
        <entry key="英语老师" value-ref="teacherOne"/>
        <entry key="数学老师" value-ref="teacherTwo"/>
</util:map>

<bean id="teacherOne" class="com.tianna.spring.pojo.Teacher">
        <property name="tid" value="10086"/>
        <property name="tname" value="大宝"/>
</bean>
<bean id="teacherTwo" class="com.tianna.spring.pojo.Teacher">
        <property name="tid" value="10087"/>
        <property name="tname" value="小宝"/>
</bean>
```

> 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

首先需要引入p命名空间：

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

然后利用bean标签中的属性为各个属性赋值，每个属性都有对象的两个属性，分别是带ref和不带ref的，不带ref的说明是为字面量赋值，带ref是引用外部bean的id。

代码如下：

```
<bean id="studentEight" class="com.tianna.spring.pojo.Student"
 p:sid="1008" p:sname="小三" p:teacherMap-ref="teacherMap"/>

<util:map id="teacherMap">
        <entry key="英语老师" value-ref="teacherOne"/>
        <entry key="数学老师" value-ref="teacherTwo"/>
</util:map>
```

2.2.10、实验十：引入外部属性文件

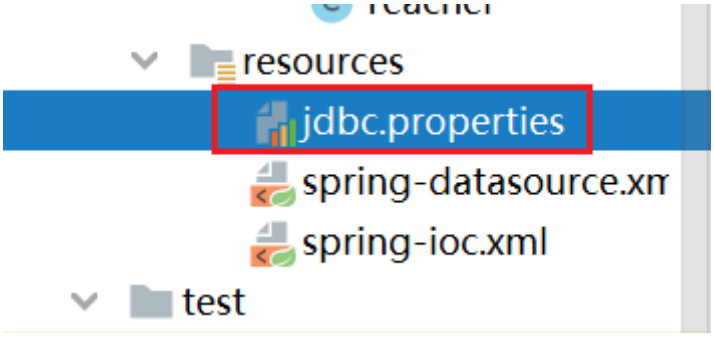
使用Spring来管理数据源，并引入外部属性文件`jdbc.properties`的内容。

1、加入依赖

在pom.xml文件中引入下面的依赖

```
<!--MySQL驱动-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.29</version>
</dependency>
<!--数据源-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.31</version>
</dependency>
```

2、创建一个外部属性文件`jdbc.properties`



其基本内容如下，包括连接数据库的基本信息：

```
jdbc.driver = com.mysql.cj.jdbc.Driver
jdbc.url = jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
jdbc.username = root
jdbc.password = 123456
```

3、引入属性文件

在spring配置文件中引入属性文件，首先需要添加`context`命名空间：

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

在配置文件中引入属性文件代码：

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

4、配置bean

在spring配置文件中创建一个与数据源有关的bean，通过{key}访问属性文件中的值:

```
<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>
```

5、测试

通过从druid获取一个连接对象测试是否能够成功连接数据库。

```
@Test
public void testDataSource() throws SQLException {
    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-
datasource.xml");
    DruidDataSource dataSource = ioc.getBean(DruidDataSource.class);
    System.out.println(dataSource.getConnection());
}
```

2.2.11、实验十一：bean的作用域

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	创建对象的时间
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

下面介绍下使用`singleton`和`prototype`的区别。

当使用`singleton`或默认时为单例模式，此时创建的每一个对象为同一个对象。下面演示一下，将scope属性设置为`singleton`。

```
<bean id="student" class="com.tianna.spring.pojo.Student"
scope="singleton">
    <property name="sid" value="10001"/>
    <property name="sname" value="张三"/>
</bean>
```

下面创建两个对象测试以下是否为同一个对象，因为类重写了`toString`，这里使用`==`来判断两个对象的内存地址是否一样。

```
@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);
}
```

输出结果如下：

```
true
```

由结果可得，当设置为`singleton`或默认时，为单例模式。

当将scope属性设置为`prototype`时为多例模式，即创建的对象是不同的。

```
<bean id="student" class="com.tianna.spring.pojo.Student"
scope="prototype">
    <property name="sid" value="10001"/>
    <property name="sname" value="张三"/>
</bean>
```

下面通过创建两个对象测试以下：

```
@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);
}
```

输出结果如下：

```
false
```

2.2.12、实验十二：bean的生命周期

bean具体的生命周期过程：

1、实例化（无参构造器）

2、依赖注入（调用set方法给对象设置属性）

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`属性指定初始化的方法。

> 注意
>
> 1、其中的initMethod()和destroyMethod()，需要通过配置bean指定为初始化和销毁的方法。
>
> 2、第三步和第四步过程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，需要进行创建和配置才能生效，在后续会详细介绍。

下面通过代码演示bean的生命周期。

首先创建一个User类，并在其无参构造方法，其中一个属性的set方法中输出当前过程，然后创建初始化和销毁的方法（这两个方法需要在bean的`init-method`和`destroy-method`属性进行指定）。

```
public class User {
    private Integer id;
    private String username;
    private String password;
    private Integer age;

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

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

    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;
    }

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

    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");
    }
}
```

在spring配置文件中配置bean，并使用`init-method`和`destroy-method`属性指定初始化和销毁方法。

```
<!-- 使用init-method属性指定初始化方法 -->
<!-- 使用destroy-method属性指定销毁方法 -->
<bean id="user" class="com.tianna.spring.pojo.User" init-
method="initMethod" destroy-method="destroyMethod">
    <property name="id" value="1"/>
    <property name="username" value="admin"/>
    <property name="password" value="123456"/>
    <property name="age" value="23"/>
</bean>
```

下面通过测试方法测试生命周期：

```
@Test
public void test(){
    //ConfigurableApplicationContext是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.xml");
    User user = ioc.getBean(User.class);
    System.out.println(user);
    //IOC容器关闭时才会销毁
    ioc.close();
}
```

输出结果如下：

```
生命周期1：实例化
生命周期2：依赖注入
生命周期3：初始化
User{id=1, username='admin', password='123456', age=23}
生命周期4：销毁
```

bean的作用域会对生命周期产生影响

1. 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容器时执行。
2. 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。

当bean的作用域为单例时，spring配置文件中bean的配置如下：

```
<bean id="user" scope="singleton" class="com.tianna.spring.pojo.User" init-
method="initMethod" destroy-method="destroyMethod">
    <property name="id" value="1"/>
    <property name="username" value="admin"/>
    <property name="password" value="123456"/>
    <property name="age" value="23"/>
</bean>
```

下面进行测试，测试时只执行获取IOC容器的方法，将获取bean的方法去掉：

```
@Test
public void test(){
    ConfigurableApplicationContext ioc = new
ClassPathXmlApplicationContext("spring-lifecycle.xml");
   /* User user = ioc.getBean(User.class);
    System.out.println(user);
    ioc.close();*/
}
```

输出结果如下：

```
生命周期1：实例化
生命周期2：依赖注入
生命周期3：初始化
```

可见在单例下，实例化、依赖注入、初始化均在获取IOC容器时执行。

当bean的作用域为多例时，spring配置文件中bean的配置如下：

```
<bean id="user" scope="prototype" class="com.tianna.spring.pojo.User" init-
method="initMethod" destroy-method="destroyMethod">
    <property name="id" value="1"/>
    <property name="username" value="admin"/>
    <property name="password" value="123456"/>
    <property name="age" value="23"/>
</bean>
```

下面进行测试，测试时只执行获取IOC容器的方法，将获取bean的方法去掉：

```
@Test
public void test(){
    ConfigurableApplicationContext ioc = new
ClassPathXmlApplicationContext("spring-lifecycle.xml");
   /* User user = ioc.getBean(User.class);
    System.out.println(user);
    ioc.close();*/
}
```

输出结果如下：

```

```

输出结果为空，将获取bean的代码添加后，再次执行测试代码：

```
@Test
public void test(){
    ConfigurableApplicationContext ioc = new
ClassPathXmlApplicationContext("spring-lifecycle.xml");
    User user = ioc.getBean(User.class);
    System.out.println(user);
    ioc.close();
}
```

输出结果如下：

```
生命周期1：实例化
生命周期2：依赖注入
生命周期3：初始化
User{id=1, username='admin', password='123456', age=23}
```

可见在单例下，实例化、依赖注入、初始化均在获取bean时执行。

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

首先在com.tianna.pojo.process包下创建一个bean的后置处理器`MyBeanPostProcessor`

```
package com.tianna.spring.process;

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

/**
 * @author tiancn
 * @date 2022/8/7 21:13
 */
public class MyBeanPostProcessor implements BeanPostProcessor {
    @Override
    public Object postProcessBeforeInitialization(Object bean, String
beanName) throws BeansException {
        //此方法在bean的生命周期初始化之前执行
        System.out.println("MyBeanPostProcessor-->后置处理器
postProcessBeforeInitialization");
        return
BeanPostProcessor.super.postProcessBeforeInitialization(bean, beanName);
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String
beanName) throws BeansException {
        //此方法在bean的生命周期初始化之后执行
```

```
        System.out.println("MyBeanPostProcessor-->后置处理器
postProcessAfterInitialization");
        return BeanPostProcessor.super.postProcessAfterInitialization(bean,
beanName);
    }
}
```

然后在IOC容器中配置后置处理器：

```
<bean id="myBeanPostProcessor"
class="com.tianna.spring.process.MyBeanPostProcessor"></bean>
```

再次执行测试代码：

```
@Test
public void test(){
    //ConfigurableApplicationContext是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.xml");
    User user = ioc.getBean(User.class);
    System.out.println(user);
    ioc.close();
}
```

结果如下：

```
生命周期1：实例化
生命周期2：依赖注入
MyBeanPostProcessor-->后置处理器postProcessBeforeInitialization
生命周期3：初始化
MyBeanPostProcessor-->后置处理器postProcessAfterInitialization
User{id=1, username='admin', password='123456', age=23}
生命周期4：销毁
```

即为bean的完整的生命周期。

2.2.13、实验十三：FactoryBean

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。

FactoryBean是一个接口，需要创建一个类实现该接口

其中由三个方法：

- 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。

下面通过代码演示一下：

首先创建一个类`UserFactoryBean`来实现接口`FactoryBean`。并重写接口的`getObject()`和`getObjectType()`方法，其中`getObject()`返回一个User对象，`getObjectType()`返回User的类型，代码如下：

```
public class UserFactoryBean implements FactoryBean<User> {
    @Override
    public User getObject() throws Exception {
        return new User();
    }

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

然后再spring配置文件中配置bean

```
<bean id="user" class="com.tianna.spring.factory.UserFactoryBean"></bean>
```

进行测试，会返回user对象：

```
@Test
public void testFactoryBean(){
    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-
factory.xml");
    User user = ioc.getBean(User.class);
    System.out.println(user);
}
```

结果如下：

```
生命周期1：实例化
User{id=null, username='null', password='null', age=null}
```

2.2.14、实验十四：基于xml的自动装配

自动装配：

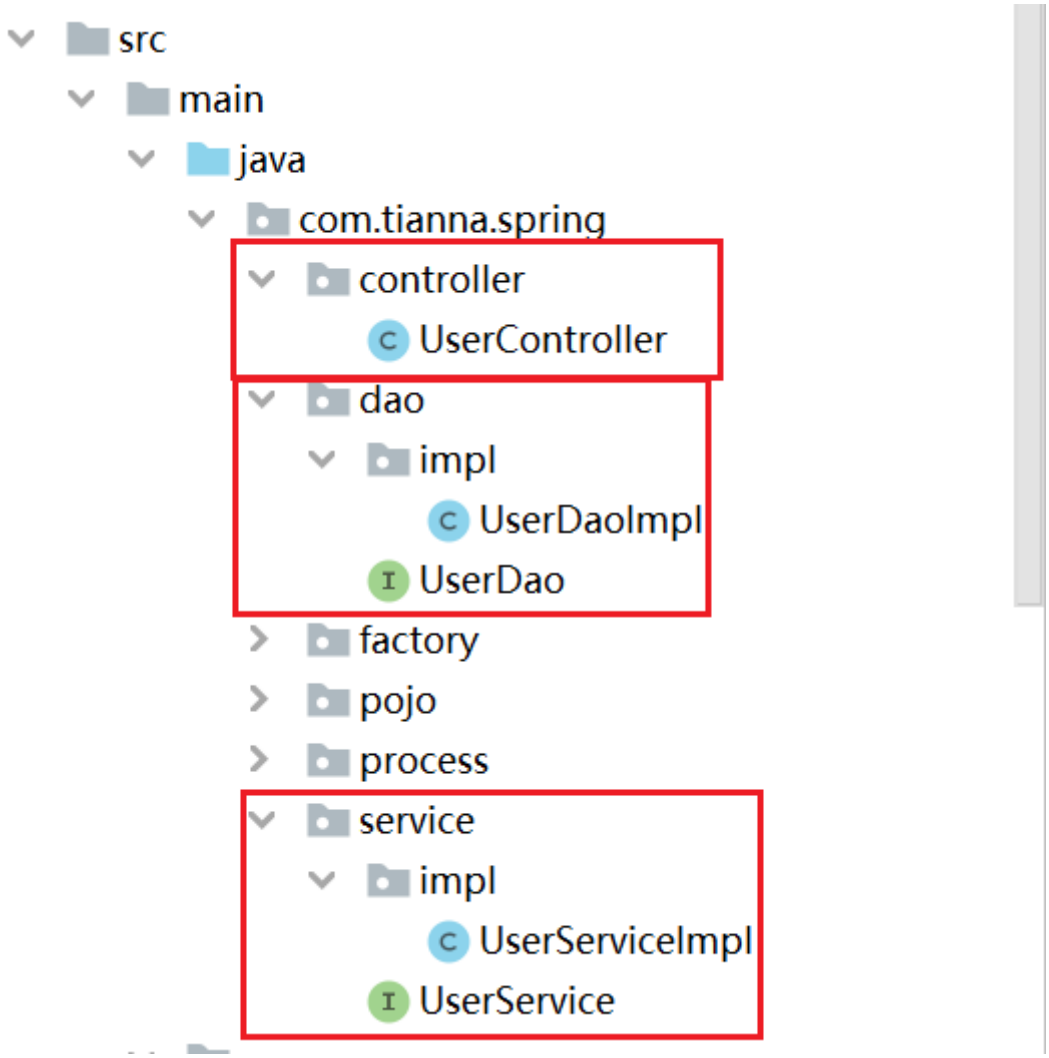
根据指定的策略，在IOC容器中匹配某一个bean，自动为指定的bean中所依赖的**类类型**或接口类型属性赋值

可以通过bean标签中的`autowire`属性设置自动装配的策略:

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：根据要赋值的类型的属性，在IOC容器中匹配某个bean，为属性赋值。
3. byName：将要赋值的属性的属性名 作为bean的id 在容器中匹配某个bean，然后为属性赋值。

1、场景模拟（不使用自动装配）

以三层架构为例，由控制层controller、业务层service、持久层dao组成。业务层service、持久层dao首先需要创建相对应的接口，然后实现接口中的方法。结构组成如下所示：



每个类和接口中的代码如下：

UserController类：

```
public class UserController {
    private UserService userService;
    public UserService getUserService() {
        return userService;
    }
    public void setUserService(UserService userService) {
        this.userService = userService;
    }
    public void saveUser(){
        userService.saveUser();
    }
}
```

创建接口UserService

```
public interface UserService {
    /**
     * 保存用户信息
     */
    void saveUser();
}
```

创建类UserServiceImpl实现接口UserService

```
public class UserServiceImpl implements UserService {
    private UserDao userDao;

    public UserDao getUserDao() {
        return userDao;
    }
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }
    @Override
    public void saveUser() {
        userDao.saveUser();
    }
}
```

创建接口UserDao

```
public interface UserDao {
    /**
     * 保存用户信息
     */
    void saveUser();
}
```

创建类UserDaoImpl实现接口UserDao

```
public class UserDaoImpl implements UserDao {
    @Override
    public void saveUser() {
        System.out.println("保存成功");
    }
}
```

在不使用自动装配时，将上面创建的各个类的关系在spring配置文件中进行配置：

```
<bean id="userController"
class="com.tianna.spring.controller.UserController">
    <property name="userService" ref="userService"></property>
</bean>
<bean id="userService"
class="com.tianna.spring.service.impl.UserServiceImpl">
    <property name="userDao" ref="userDao"></property>
</bean>
<bean id="userDao" class="com.tianna.spring.dao.impl.UserDaoImpl"></bean>
```

测试代码如下：

```
@Test
public void testAutowire(){
    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-
autowire-xml.xml");
    UserController userController = ioc.getBean(UserController.class);
    userController.saveUser();
}
```

结果如下，业务场景模拟成功：

```
保存成功
```

2、自动装配方式：byType(常用)

根据要赋值的类型的属性，在IOC容器中匹配某个bean，为属性赋值。Spring配置文件中的代码如下：

```
<bean id="userController"
class="com.tianna.spring.controller.UserController" autowire="byType">
</bean>
<bean id="userService"
class="com.tianna.spring.service.impl.UserServiceImpl" autowire="byType">
</bean>
<bean id="userDao" class="com.tianna.spring.dao.impl.UserDaoImpl"></bean>
```

注意：

> - 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

> - 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、自动装配方式：byName

将要赋值的属性的属性名 作为bean的id 在容器中匹配某个bean，然后为属性赋值。Spring配置文件中的代码如下：

```
<bean id="userController"
class="com.tianna.spring.controller.UserController" autowire="byName">
</bean>
<bean id="userService"
class="com.tianna.spring.service.impl.UserServiceImpl" autowire="byName">
</bean>
<bean id="userDao" class="com.tianna.spring.dao.impl.UserDaoImpl"></bean>
```

总结：

> 当类型匹配的bean有多个时，此时可以使用byName实现自动装配。

2.3、基于注解管理bean

2.3.1、实验一：标记与扫描

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。

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。

3、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。

以上四个组件的关系：

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。

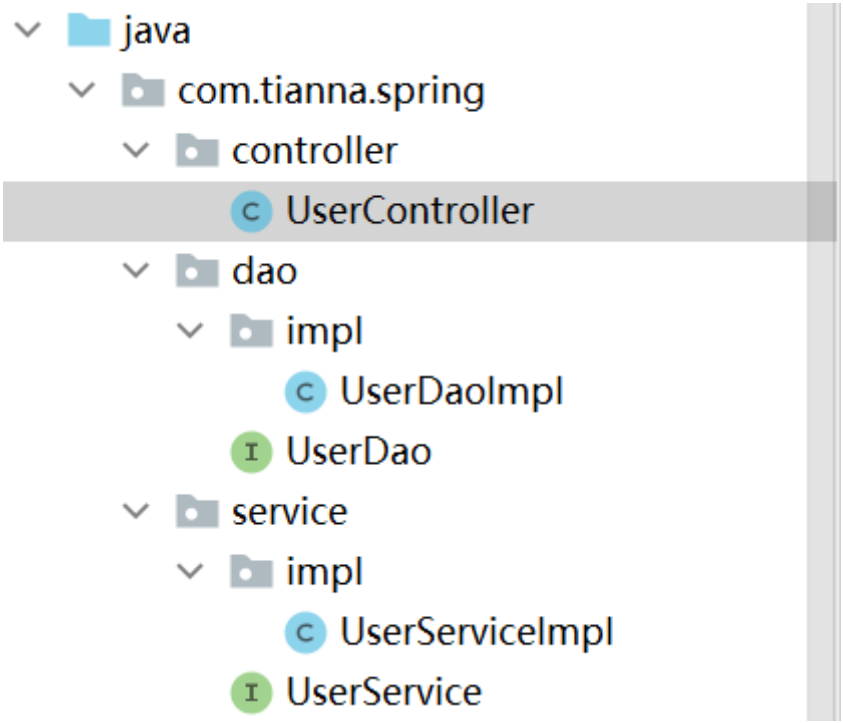
对于Spring使用IOC容器管理这些组件来说没有区别。所以@Controller、@Service、@Repository这三个注解只是给开发人员看的，让我们能够便于分辨组件的作用。

> 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。

下面演示一下如何使用注解标记和扫描组件：

4、创建组件：

创建三层架构：控制层、业务层和持久层，以及业务层和持久层的接口，三层架构创建后的结构如下：



为每个组件上添加合适的注解：

控制层：

创建控制层组件：

```
@Controller
public class UserController {
}
```

业务层：

创建接口UserService

```
public interface UserService {
}
```

创建业务层组件UserServiceImpl

```
@Service
public class UserServiceImpl implements UserService {
}
```

持久层：

创建接口UserDao

```
public interface UserDao {
}
```

创建持久层组件UserDaoImpl

```
@Repository
public class UserDaoImpl implements UserDao {
}
```

5、扫描组件

扫描组件需要在spring配置文件中进行配置，需要使用`context:component-scan`标签，使用`context:component-scan`需要引入`context`命名空间：

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

扫描组件的方式如下：

情况一：最基本的扫描方式，`base-package`属性的内容为扫描的包。

```
<context:component-scan base-package="com.tianna.spring">
</context:component-scan>
```

情况二：指定要排除的组件，需要使用`context:exclude-filter`标签。

context:exclude-filter：排除扫描
type：设置排除扫描的方式，可取`annotation`和`assignable`两个值：

- 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

如下代码为排除`com.tianna.spring`包下带有controller注解的组件。

```
<context:component-scan base-package="com.tianna.spring">
    <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
```

情况三：仅扫描指定组件，需要使用`context:include-filter`标签。

> 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

下面代码为只包含`com.tianna.spring`包下带有controller注解的组件。

```
<context:component-scan base-package="com.tianna.spring" use-default-filters="false">
    <context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
```

6、测试

测试使用情况一：最基本的扫描方式，即扫描`com.tianna.spring`包下所有的组件。

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

结果如下：

```
com.tianna.spring.controller.UserController@3cfdd820
com.tianna.spring.service.impl.UserServiceImpl@928763c
com.tianna.spring.dao.impl.UserDaoImpl@e25951c
```

7、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。以控制层为例，默认情况下为控制层类添加注解如下：

```
@Controller
public class UserController {
}
```

在测试时获取bean则可以通过其默认的id(`userController`)，即类名首字母小写就是来获取bean：

```
 @Test
 public void test(){
     ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-
ioc-annotation.xml");
     UserController userController =
ioc.getBean("userController",UserController.class);
     System.out.println(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。

以UserController为例，将其bean的id设置为`controller`：

```
@Controller("controller")
public class UserController {
}
```

在测试时获取bean则可以通过自定义的id(`controller`)：

```
@Test
public void test(){
    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-
ioc-annotation.xml");
    UserController userController =
ioc.getBean("controller",UserController.class);
    System.out.println(userController);
}
```

2.3.2、实验二：基于注解的自动装配

1、场景模拟

还是以三层结构为例，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、@Autowired注解

@Autowired注解能够实现自动装配，其可以在如下三种位置进行标识：

- 标识在成员变量上，此时不需要设置成员变量的set方法
- 标识在set方法上
- 为当前成员变量赋值的有参构造上。

常用的为将其标识在成员变量上，下面就以标识在成员变量上为例进行演示：

控制层：

```
@Controller
public class UserController {
    @Autowired
    private UserService userService;

    public void saveUser(){
        userService.saveUser();
    }
}
```

业务层：

```
@Service
public class UserServiceImpl implements UserService {
    @Autowired
    private UserDao userDao;

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

持久层：

```
@Repository
public class UserDaoImpl implements UserDao {
    @Override
    public void saveUser() {
        System.out.println("保存成功");
    }
}
```

spring配置文件代码：

```
<context:component-scan base-package="com.tianna.spring">
</context:component-scan>
```

测试代码：

```
@Test
public void test(){
    ApplicationContext ioc = new ClassPathXmlApplicationContext("spring-
ioc-annotation.xml");
    UserController userController = ioc.getBean(UserController.class);
    userController.saveUser();
}
```

输出结果：

```
保存成功
```

3、@Autowired注解原理

- 默认通过byType方式，在IOC容器中通过类型匹配某个bean为属性赋值。
- 若有多个类型匹配的bean，此时会自动转换为byName的方式实现自动装配的效果，即将要赋值的属性的属性名作为bean的id匹配某个bean为属性赋值。

  在spring配置文件中，分别创建一个UserDaoImpl和UserServiceImpl的bean，其id与赋值的属性名相同。

  ```
  <context:component-scan base-package="com.tianna.spring">
  </context:component-scan>
  <bean id="userDao" class="com.tianna.spring.dao.impl.UserDaoImpl">
  </bean>
  <bean id="userService"
  class="com.tianna.spring.service.impl.UserServiceImpl"></bean>
  ```

  此时就会通过byName的方式实现自动装配的效果。
- 若byType和byName的方式都无法实现自动装配，即IOC容器中有多个类型匹配的bean，且这些bean的id和要赋值的属性的属性名都不一致，此时抛异常。bean的id和要赋值的属性的属性名都不一致情况时spring配置文件内容如下：

  ```
  <context:component-scan base-package="com.tianna.spring">
  </context:component-scan>
  <bean id="dao" class="com.tianna.spring.dao.impl.UserDaoImpl"></bean>
  <bean id="service"
  class="com.tianna.spring.service.impl.UserServiceImpl"></bean>
  ```

  此时可以在要赋值的属性上，添加一个注解@Qualifier，通过该注解的value属性值，指定某个bean的id，将这个bean为属性赋值。@Qualifier的value属性值与上面spring配置文件中bean的id相同。

  控制层：

  ```
  @Controller
  public class UserController {
      @Autowired
      @Qualifier("service")
      private UserService userService;

      public void saveUser(){
          userService.saveUser();
      }
  }
  ```

  业务逻辑层：

```
@Service
public class UserServiceImpl implements UserService {
    @Autowired
    @Qualifier("dao")
    private UserDao userDao;

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

- 若IOC容器中没有任何一个类型匹配的bean，此时抛出异常

  在@Autowired注解中有个属性required，默认值为true，要求必须完成自动装配。

  可以将required设置为false，此时能装配则装配，无法装配则使用属性的默认值

  演示代码如下：

  控制层，将required属性设置为false：

```
@Controller
public class UserController {
    @Autowired(required = false)
    /*@Qualifier("service")*/
    private UserService userService;

    public void saveUser(){
        userService.saveUser();
    }
}
```

  业务逻辑层：去掉@Service注解，并且在spring配置文件中删除该类的bean。

```
/*@Service*/
public class UserServiceImpl implements UserService {
    @Autowired
    /*@Qualifier("dao")*/
    private UserDao userDao;

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

  此时会报空指针异常错误。

3、AOP

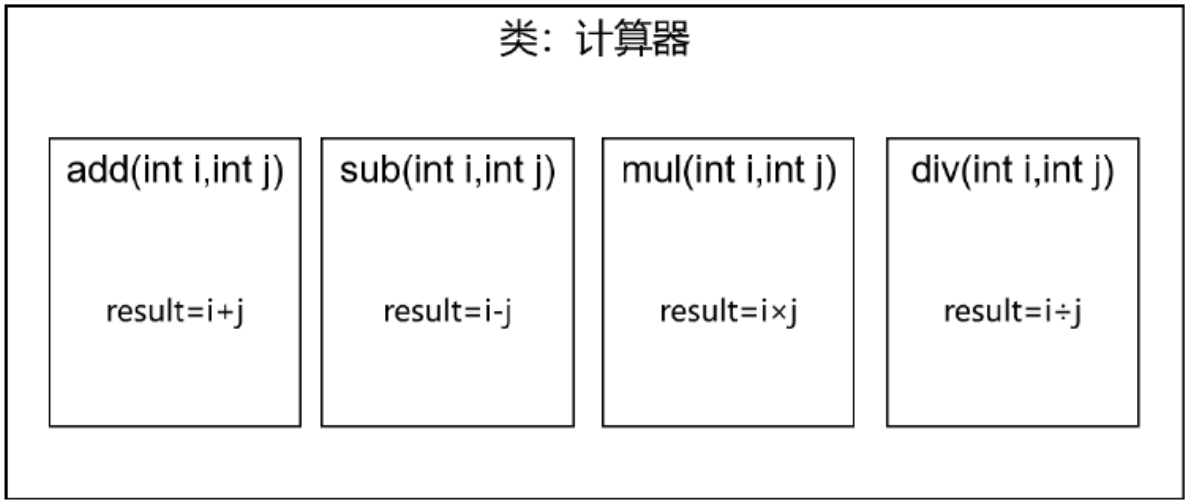
3.1、场景模拟

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：

```
public interface Calculator {

    int add(int i, int j);

    int sub(int i, int j);

    int mul(int i, int j);

    int div(int i, int j);
}
```

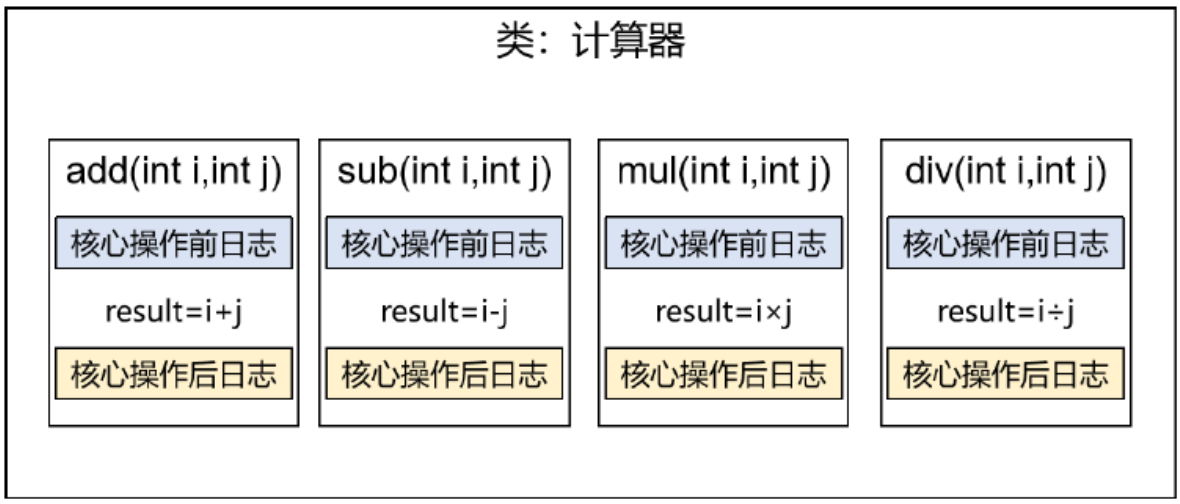
3.1.2、创建实现类




```
public class CalculatorImpl implements Calculator{
    @Override
    public int add(int i, int j) {
        int result = i + j;
        System.out.println("方法内部，result：  "+result);
        return result;
    }

    @Override
    public int sub(int i, int j) {
        int result = i - j;
        System.out.println("方法内部，result：  "+result);
        return result;
    }

    @Override
    public int mul(int i, int j) {
        int result = i * j;
```

```
        System.out.println("方法内部，result：" +result);
        return result;
    }

    @Override
    public int div(int i, int j) {
        int result = i / j;
        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




```
public class CalculatorPureImpl implements Calculator{
    @Override
    public int add(int i, int j) {
        System.out.println("[日志] 方法：add，参数：" + i + "," + j);
        int result = i + j;
        System.out.println("方法内部，result：" +result);
        System.out.println("[日志] 方法：add，结果：" + result);
        return result;
    }

    @Override
    public int sub(int i, int j) {
        System.out.println("[日志] 方法：sub，参数：" + i + "," + j);
        int result = i - j;
        System.out.println("方法内部，result：" +result);
        System.out.println("[日志] 方法：sub，结果：" + result);
        return result;
    }

    @Override
    public int mul(int i, int j) {
        System.out.println("[日志] 方法：mul，参数：" + i + "," + j);
        int result = i * j;
        System.out.println("方法内部，result：" +result);
```

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

    @Override
    public int div(int i, int j) {
        System.out.println("[日志] 方法：div，参数：" + i + "," + j);
        int result = i / j;
        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

1、现有代码的缺陷：

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

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。

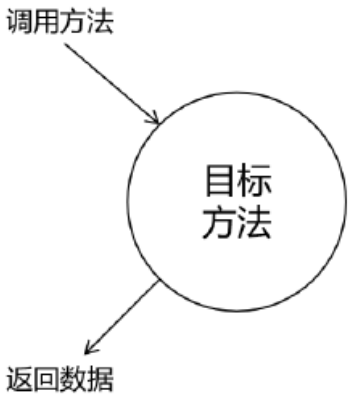
3、困难

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。

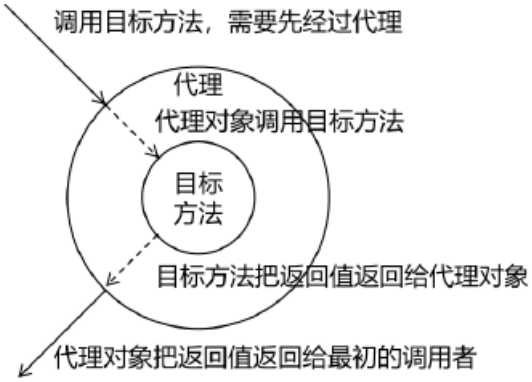
3.2、代理模式

3.2.1、概念

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：



就是为目标对象创建一个代理对象，每一次访问目标对象都通过代理对象进行访问，在代理对象里控制目标对象的方法的执行，在代理对象中加入加入一些额外的操作。

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

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

目标类：

```
public class CalculatorImpl implements Calculator{
    @Override
    public int add(int i, int j) {
        int result = i + j;
        System.out.println("方法内部，result：+result);
        return result;
    }

    @Override
    public int sub(int i, int j) {
        int result = i - j;
        System.out.println("方法内部，result：+result);
        return result;
    }

    @Override
    public int mul(int i, int j) {
        int result = i * j;
        System.out.println("方法内部，result：+result);
        return result;
    }

    @Override
    public int div(int i, int j) {
```

```
        int result = i / j;
        System.out.println("方法内部，result：  "+result);
        return result;
    }
}
```

静态代理类：

```
public class CalculatorStaticProxy implements Calculator{
    private CalculatorImpl target;

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

    @Override
    public int add(int i, int j) {
        System.out.println("[日志] 方法：add，参数：  " + i + "," + j);
        int result = target.add(i,j);
        System.out.println("[日志] 方法：add，结果：  " + result);
        return result;
    }

    @Override
    public int sub(int i, int j) {
        System.out.println("[日志] 方法：sub，参数：  " + i + "," + j);
        int result = target.add(i,j);
        System.out.println("[日志] 方法：sub，结果：  " + result);
        return result;
    }

    @Override
    public int mul(int i, int j) {
        System.out.println("[日志] 方法：mul，参数：  " + i + "," + j);
        int result = target.add(i,j);
        System.out.println("[日志] 方法：mul，结果：  " + result);
        return result;
    }

    @Override
    public int div(int i, int j) {
        System.out.println("[日志] 方法：div，参数：  " + i + "," + j);
        int result = target.div(i,j);
        System.out.println("[日志] 方法：div，结果：  " + result);
        return result;
    }
}
```

测试方法及结果：

```
@Test
public void testProxy(){
    CalculatorStaticProxy proxy = new CalculatorStaticProxy(new
CalculatorImpl());
    proxy.add(1,2);
}
```

```
[日志] 方法：add，参数：1,2
方法内部，result：3
[日志] 方法：add，结果：3
```

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

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。

生产代理对象的工厂类：

```
public class ProxyFactory {
    //目标对象
    private 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 h = new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                //proxy: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
                Object result = null;
                try {
```

```
                    System.out.println("[日志] 方法：" + method.getName() +
"参数：" + Arrays.toString(args));
                    //执行目标对象的方法
                    result = method.invoke(target, args);
                    System.out.println("[日志] 方法：" + method.getName() +
" 结果：" + result);
                } catch (Exception e) {
                    e.printStackTrace();
                    System.out.println("[日志] 方法：" + method.getName() +
" 异常：" + e);
                }  finally {
                    System.out.println("[日志] 方法：" + method.getName() +
"方法执行完毕");
                }
                return result;
            }
        };
        return Proxy.newProxyInstance(classLoader,interfaces,h);
    }
}
```

测试方法及结果如下：

```
@Test
public void testProxy(){
    ProxyFactory proxyFactory = new ProxyFactory(new CalculatorImpl());
    Calculator proxy = (Calculator)proxyFactory.getProxy();
    proxy.add(1,2);
}
```

```
[日志] 方法：add参数：[1, 2]
方法内部，result：3
[日志] 方法：add 结果：3
[日志] 方法：add方法执行完毕
```

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

1、横切关注点

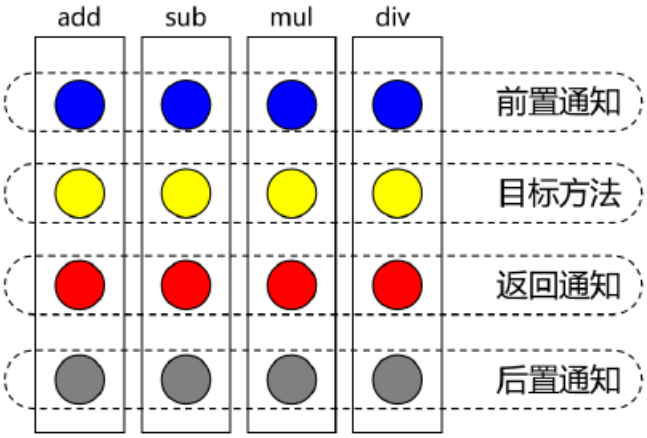
从每个方法中抽取出来的同一类非核心业务。在同一个项目中，我们可以使用多个横切关注点对相关方法进行多个不同方面的增强。

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。

2、通知

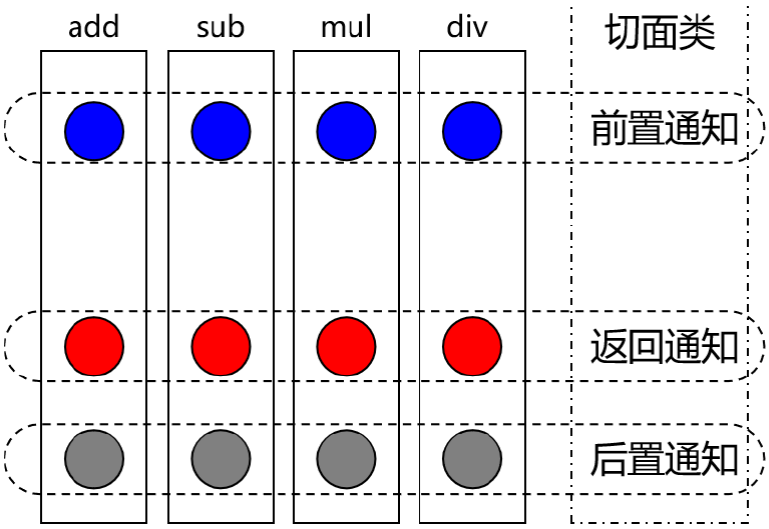
每一个横切关注点上要做的事情都需要写一个方法来实现，这样的方法就叫通知方法。

- 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、切面

封装通知方法的类。



4、目标

被代理的目标对象。

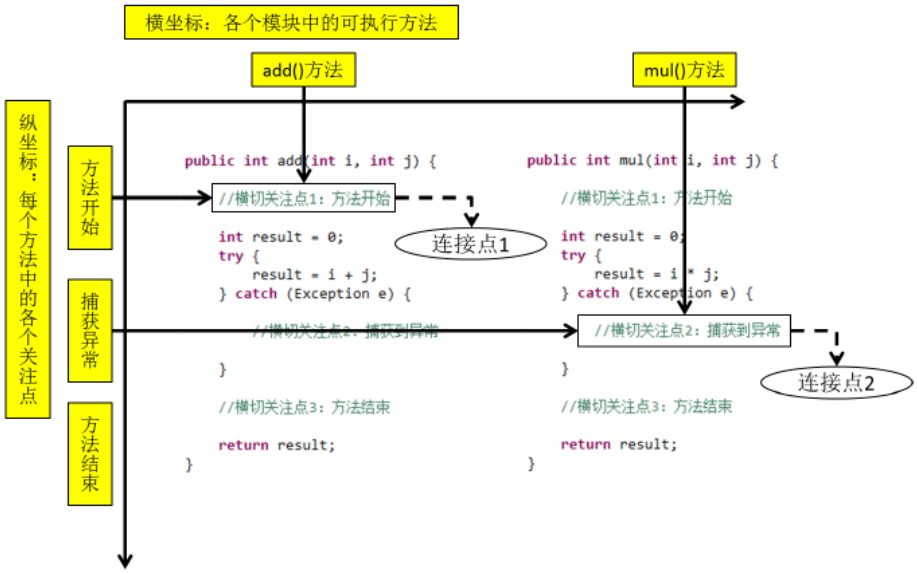
5、代理

向目标对象应用通知之后创建的代理对象。

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。



7、切入点（表达式）

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。

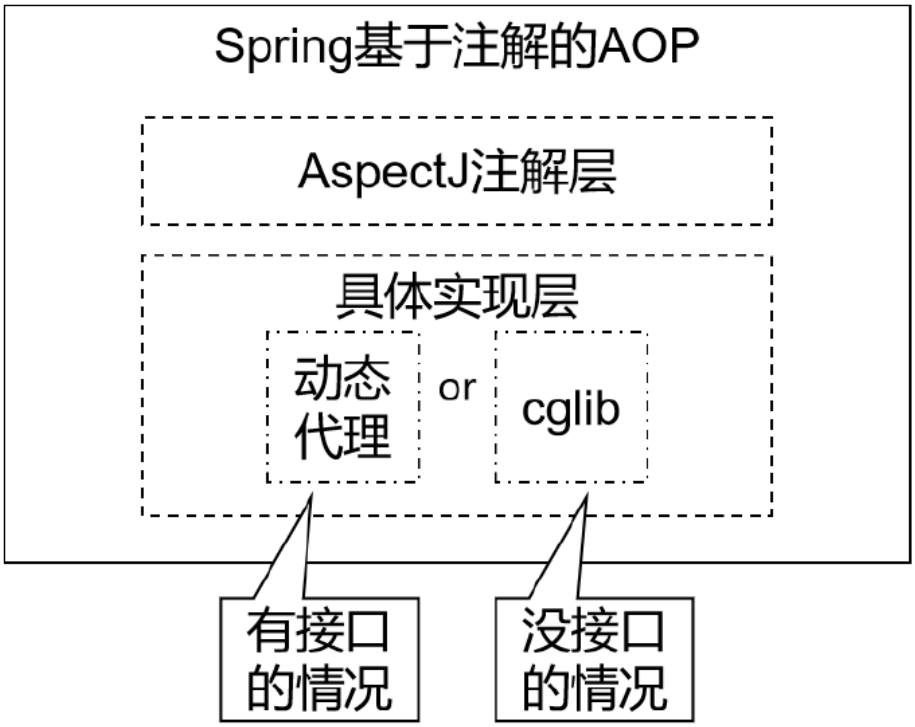
切点通过 org.springframework.aop.Pointcut 接口进行描述，它使用类和方法作为连接点的查询条件。

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

1、添加依赖

在pom.xml文件中加入如下依赖

```
<!--基于Maven依赖特性，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>5.3.1</version>
</dependency>
<!--junit测试-->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</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>5.3.1</version>
</dependency>
```

2、准备被代理的目标资源

接口：

```
public interface Calculator {

    int add(int i, int j);

    int sub(int i, int j);

    int mul(int i, int j);

    int div(int i, int j);
}
```

实现类：

```
@Component
public class CalculatorImpl implements Calculator{
    @Override
    public int add(int i, int j) {
        int result = i + j;
        System.out.println("方法内部，result：" +result);
        return result;
    }

    @Override
    public int sub(int i, int j) {
        int result = i - j;
        System.out.println("方法内部，result：" +result);
        return result;
    }

    @Override
    public int mul(int i, int j) {
        int result = i * j;
        System.out.println("方法内部，result：" +result);
        return result;
    }

    @Override
    public int div(int i, int j) {
        int result = i / j;
        System.out.println("方法内部，result：" +result);
        return result;
    }
}
```

3.4.3、创建切面类并配置

切面类代码如下，详细信息后续介绍：

```
@Component
@Aspect //将当前组件标识为切面
public class LoggerAspect {

    //重用切入点表达式
    @Pointcut("execution(*
com.tianna.spring.aop.annotation.CalculatorImpl.*(..))")
    public void pointCut(){}

    //前置通知
    //@Before("execution(public int
com.tianna.spring.aop.annotation.CalculatorImpl.add(int,int))")
    @Before("execution(* com.tianna.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();
        //获取连接点所对应的参数
        Object[] args = joinPoint.getArgs();
        System.out.println("LoggerAspect，方法：" + signature.getName() +
"参数：" + Arrays.toString(args));
    }
    @After("pointCut()")
    public void afterAdviceMethod(JoinPoint joinPoint){
        //获取连接点所对应方法的签名信息
        Signature signature = joinPoint.getSignature();
        System.out.println("LoggerAspect，方法：" + signature.getName() +
"，执行完毕");
    }
    //再返回通知中若要获取目标对象方法的返回值，
    //只需要通过@AfterReturning注解的returning属性
    //就可以将通知方法的某个参数指定为接受目标对象方法的返回值参数
    @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();
        System.out.println("LoggerAspect，方法：" + signature.getName() +
"，结果：" + result);
    }
    //再异常通知中若要获取目标对象方法的异常，
    //只需要通过@AfterThrowing注解的throwing属性
    //就可以将通知方法的某个参数指定为接受目标对象方法出现的异常的参数
    @AfterThrowing(value = "pointCut()",throwing = "ex")
    public void afterThrowingAdviceMethod(JoinPoint joinPoint,Throwable ex)
{
        //获取连接点所对应方法的签名信息
```

```
        Signature signature = joinPoint.getSignature();
        System.out.println("LoggerAspect，方法：" + signature.getName() +
"，异常：" + ex);
    }

    //环绕通知的方法一定要和目标对象方法的返回值一致
    @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");
            throw new RuntimeException(e);
        }finally {
            System.out.println("环绕通知-->后置通知");
        }
        return result;

    }

}
```

在Spring的配置文件中配置：

```
<!--
    基于注解的AOP的实现：
    切面类和目标类都需要交给IOC容器管理
    切面类必须通过@Aspect注解标识为一个切面
    在Spring配置文件中设置<aop:aspectj-autoproxy/>开启基于注解的AOP
-->
<context:component-scan base-package="com.tianna.spring.aop.annotation">
</context:component-scan>
<!--开启基于注解的AOP-->
<aop:aspectj-autoproxy/>
```

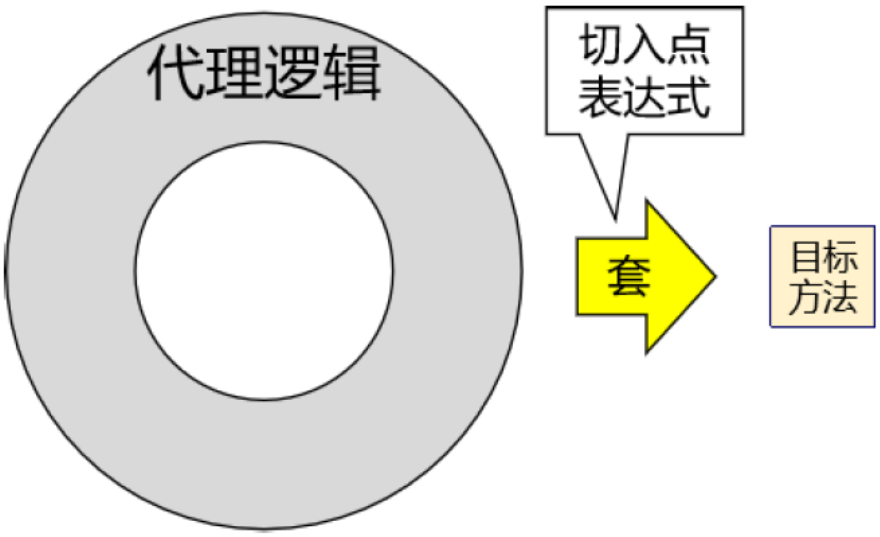
3.4.4、各种通知

在切面中，需要通过指定的注解将方法标识为通知方法。

- @Before：前置通知，在目标对象方法执行之前执行
- @After：后置通知，在目标对象方法的finally子句中执行
- @AfterReturning：返回通知，在目标对象方法返回值之后执行
- @AfterThrowing：异常通知，在目标对象方法的catch子句中执行
- @Around：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。环绕通知的方法一定要和目标对象方法的返回值一致

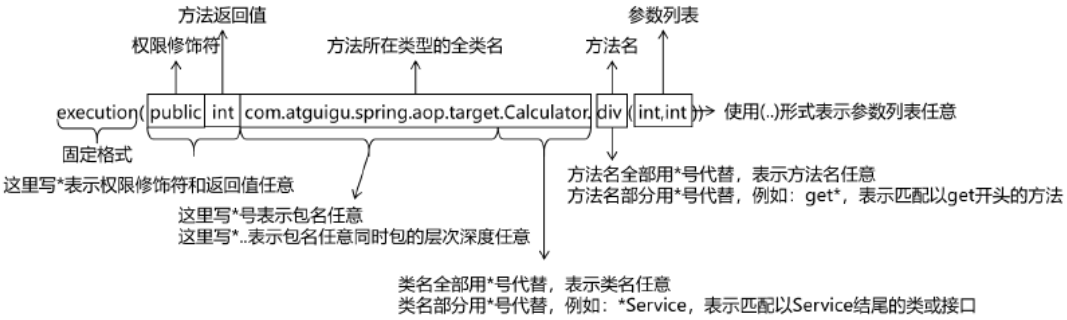
3.4.5、切入点表达式语法

作用：



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.atguigu.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

声明

```
@Pointcut("execution(* com.tianna.spring.aop.annotation.CalculatorImpl.*(..))")
public void pointCut(){}
```

在切面中使用

```
@After("pointCut()")
public void afterAdviceMethod(JoinPoint joinPoint){
    //获取连接点所对应方法的签名信息
    Signature signature = joinPoint.getSignature();
    System.out.println("LoggerAspect，方法：" + signature.getName() + "，执行完毕");
}
```

3.4.7、获取通知的相关信息

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(* com.tianna.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();
    //获取连接点所对应的参数
    Object[] args = joinPoint.getArgs();
    System.out.println("LoggerAspect，方法：" + signature.getName() + "参数：" + Arrays.toString(args));
}
```

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

```
//再返回通知中若要获取目标对象方法的返回值，
//只需要通过@AfterReturning注解的returning属性
//就可以将通知方法的某个参数指定为接受目标对象方法的返回值参数
@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();
    System.out.println("LoggerAspect，方法：" + signature.getName() + "，结果：" + result);
}
```

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

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

3.4.8、环绕通知

环绕通知的方法一定要和目标对象方法的返回值一致

```
//环绕通知的方法一定要和目标对象方法的返回值一致
@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");
        throw new RuntimeException(e);
    }finally {
        System.out.println("环绕通知-->后置通知");
    }
    return result;
}
```

3.4.9、切面的优先级

可以通过@Order注解的value属性设置优先级，默认值Integer的最大值

@Order注解的value属性值越小，优先级越高

如下为创建一个ValidateAspect切面类，并设置其优先级为1：

```
@Component
@Aspect
@Order(1)
public class ValidateAspect {

    @Before("execution(* com.tianna.spring.aop.annotation.CalculatorImpl.*(..))")
    public void beforeMethod(){
        System.out.println("ValidateAspect-->前置通知");
    }
}
```

3.4.10、测试

对第3.4.3节中的切面类和3.4.9节中的切面类进行测试，测试方法如下：

```
@Test
public void testAOPByAnnotation(){
    ApplicationContext ioc = new ClassPathXmlApplicationContext("aop-annotation.xml");
    Calculator calculator = ioc.getBean(Calculator.class);
    calculator.add(1,0);
}
```

输出结果如下：

```
ValidateAspect-->前置通知
环绕通知-->前置通知
LoggerAspect，方法：add参数：[1, 0]
方法内部，result：1
LoggerAspect，方法：add，结果：1
LoggerAspect，方法：add，执行完毕
环绕通知-->返回通知
环绕通知-->后置通知
```

3.5、基于XML的AOP（了解）

LoggerAspect切面类内容如下：

```
@Component
public class LoggerAspect {
```

```
    public void beforeAdviceMethod(JoinPoint joinPoint){
        //获取连接点所对应方法的签名信息
        Signature signature = joinPoint.getSignature();
        //获取连接点所对应的参数
        Object[] args = joinPoint.getArgs();
        System.out.println("LoggerAspect，方法：" + signature.getName() +
"参数：" + Arrays.toString(args));
    }
    public void afterAdviceMethod(JoinPoint joinPoint){
        //获取连接点所对应方法的签名信息
        Signature signature = joinPoint.getSignature();
        System.out.println("LoggerAspect，方法：" + signature.getName() +
"，执行完毕");
    }

    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();
        System.out.println("LoggerAspect，方法：" + signature.getName() +
"，结果：" + result);
    }

    public void afterThrowingAdviceMethod(JoinPoint joinPoint,Throwable ex)
{
        //获取连接点所对应方法的签名信息
        Signature signature = joinPoint.getSignature();
        System.out.println("LoggerAspect，方法：" + signature.getName() +
"，异常：" + ex);
    }

    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");
            throw new RuntimeException(e);
        }finally {
            System.out.println("环绕通知-->后置通知");
        }
        return result;
    }

}
```

ValidateAspect切面类内容如下：

```
@Component
public class ValidateAspect {

    public void beforeMethod(){
        System.out.println("ValidateAspect-->前置通知");
    }
}
```

spring配置文件的内容如下：

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

<aop:config>
    <!--设置公共的切入点表达式-->
    <aop:pointcut id="pointCut" expression="execution(*
com.tianna.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:before>
        <aop:after method="afterAdviceMethod" pointcut-ref="pointCut">
</aop:after>
        <aop:after-returning method="afterReturningAdviceMethod" pointcut-
ref="pointCut" returning="result"></aop:after-returning>
        <aop:after-throwing method="afterThrowingAdviceMethod" pointcut-
ref="pointCut" throwing="ex"></aop:after-throwing>
        <aop:around method="aroundAdviceMethod" pointcut-ref="pointCut">
</aop:around>
    </aop:aspect>
    <aop:aspect ref="validateAspect" order="1">
        <aop:before method="beforeMethod" pointcut-ref="pointCut">
</aop:before>
    </aop:aspect>
</aop:config>
```

测试方法及结果如下：

```
public void testAOPByXML(){
    ApplicationContext ioc = new ClassPathXmlApplicationContext("aop-
xml.xml");
    Calculator calculator= ioc.getBean(Calculator.class);
    calculator.add(1,1);
}
```

```
ValidateAspect-->前置通知
LoggerAspect，方法：add参数：[1, 1]
环绕通知-->前置通知
方法内部，result：2
环绕通知-->返回通知
环绕通知-->后置通知
LoggerAspect，方法：add，结果：2
LoggerAspect，方法：add，执行完毕
```

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

1、加入依赖

在pom.xml文件添加如下依赖：

```
<!--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>5.3.1</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>5.3.1</version>
</dependency>
<!--spring测试相关-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.3.1</version>
</dependency>
<!--junit测试-->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

```
<!--MySQL驱动-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.29</version>
</dependency>
<!--数据源-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.31</version>
</dependency>
```

2、创建jdbc.properties

其中包含了连接数据库的相关信息

```
jdbc.driver = com.mysql.cj.jdbc.Driver
jdbc.url = jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
jdbc.username = root
jdbc.password = 123456
```

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.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd">

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

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

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

```
</beans>
```

4.1.3、测试

1、在测试类装装配JdbcTemplate

需要在当前类上添加`@RunWith`和`@ContextConfiguration`两个标签，其功能分别如下：

- @RunWith：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。**
- @ContextConfiguration：设置Spring测试环境的配置文件

```
//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;

}
```

2、测试添加功能

在测试类中添加如下代码：

```
@Test
public void testInsert(){
    String sql = "insert into t_user values(null,?,?,?,?,?)";
    jdbcTemplate.update(sql,"root","123",23,"女","123@qq.com");
}
```

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);
}
```

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> list = jdbcTemplate.query(sql, new BeanPropertyRowMapper<>
(User.class));
    list.forEach(System.out::println);
}
```

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);
}
```

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 conn = ...;
try {
    // 开启事务：关闭事务的自动提交
    conn.setAutoCommit(false);
    // 核心操作
    // 提交事务
    conn.commit();
}catch(Exception e){
    // 回滚事务
    conn.rollBack();
}finally{
    // 释放数据库连接
    conn.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

1、在pom.xml文件中添加如下依赖

```
<!--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>5.3.1</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>5.3.1</version>
</dependency>
<!--spring测试相关-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.3.1</version>
</dependency>
<!--junit测试-->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
<!--MySQL驱动-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.29</version>
</dependency>
<!--数据源-->
```

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.31</version>
</dependency>
```

2、创建jdbc.properties，连接数据库的相关信息

```
jdbc.driver = com.mysql.cj.jdbc.Driver
jdbc.url = jdbc:mysql://localhost:3306/ssm?serverTimezone=UTC
jdbc.username = root
jdbc.password = 123456
```

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"
xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd">

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

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

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

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

</beans>
```

4、创建表

创建用户表和书籍表，用于模拟买书的业务。

```
CREATE TABLE `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` int(11) 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;

CREATE TABLE `t_user1` (
`user_id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',
`username` varchar(20) DEFAULT NULL COMMENT '用户名',
`balance` int(10) unsigned DEFAULT NULL COMMENT '余额（无符号）',
PRIMARY KEY (`user_id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;

insert into `t_book`(`book_id`,`book_name`,`price`,`stock`) values (1,'斗破
苍
穹',80,100),(2,'斗罗大陆',50,100);
insert into `t_user1`(`user_id`,`username`,`balance`) values
(1,'admin',50);
```

5、创建组件

创建BookController：

```
@Controller
public class BookController {
    @Autowired
    private BookService bookService;

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

创建业务逻辑层接口BookService：

```
public interface BookService {
    /**
     * 买书
     * @param userId
     * @param bookId
     */
    void buyBook(Integer userId, Integer bookId);
}
```

创建业务逻辑层实现类BookServiceImpl：

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    private BookDao bookDao;
    @Override
    public void buyBook(Integer userId, Integer bookId) {
        //查询图书的价格
        Integer 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);
    }
}
```

创建实体层接口BookDao：

```
public interface BookDao {
    /**
     * 根据图书的id查询图书的价格
     * @param bookId
     * @return
     */
    Integer getPriceByBookId(Integer bookId);

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


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

创建实体层实现类BookDaoImpl：

```
@Repository
public class BookDaoImpl implements BookDao {
    @Autowired
    private JdbcTemplate jdbcTemplate;
    @Override
    public Integer getPriceByBookId(Integer bookId) {
        String sql = "select price from t_book where book_id = ?";
        return jdbcTemplate.queryForObject(sql,Integer.class,bookId);
```

```
    }

    @Override
    public void updateStock(Integer bookId) {
        String sql = "update t_book set stock = stock - 1 where book_id =
?";
        jdbcTemplate.update(sql,bookId);
    }

    @Override
    public void updateBalance(Integer userId, Integer price) {
        String sql = "update t_user1 set balance = balance - ? where
user_id = ?";
        jdbcTemplate.update(sql,price,userId);

    }
}
```

4.3.2、测试无事务情况

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。

测试方法如下：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:tx-annotation.xml")
public class TxByAnnotationTest {
    @Autowired
    private BookController bookController;

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

由于**默认情况下一个sql独占一个事务，且自动提交。**因此图书的库存更新了，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。所以需要添加事务。

4.3.3、加入事务

声明式事务的配置步骤：

1. 在Spring的配置文件中配置事务管理器
2. 开启事务的注解驱动

在需要被事务管理的方法上，添加@Transactional注解，该方法就会被事务管理
@Transactional注解标识的位置

1. 标识在方法上
2. 标识在类上，则类中所有的方法都会被事务管理

1、添加事务配置

在spring配置文件中需要做两件事：

1. 配置事务管理器
2. 开启事务的注解驱动，需要引入 `xmlns:tx="http://www.springframework.org/schema/tx"` 命名空间。

在Spring配置文件中添加配置：

```
<!--配置事务管理器-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!--装配数据源-->
    <property name="dataSource" ref="dataSource"></property>
</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id为transactionManager，则该属性可以不写
-->
<tx:annotation-driven transaction-manager="transactionManager"/>
```

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。

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    private BookDao bookDao;
    @Override
    @Transactional
    public void buyBook(Integer userId, Integer bookId) {
        //查询图书的价格
        Integer 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);
    }
}
```

3、执行结果

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、事务属性：只读

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。

2、使用方式

默认为`false`

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    private BookDao bookDao;
    @Override
    @Transactional(readOnly = true)
    public void buyBook(Integer userId, Integer bookId) {
        //查询图书的价格
        Integer 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);
    }
}
```

3、执行结果

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

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。

2、测试

将超时时间设置为3s，并在方法中添加一个5s的睡眠，因此会超时回滚。

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    private BookDao bookDao;
    @Override
    @Transactional(timeout = 3)
    public void buyBook(Integer userId, Integer bookId) {
        try {
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        //查询图书的价格
        Integer 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);
    }
}
```

3、执行结果

执行过程中抛出异常：

```
org.springframework.transaction.TransactionTimedOutException: Transaction
timed out: deadline was Tue Aug 16 15:30:16 CST 2022
```

4.3.7、事务属性：回滚策略

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

2、使用方式

将用户金额更改为足够购买一本书，然后在方法中执行1/0的运算，因此抛出异常，此时会发生回滚，即无法完成买书操作。

因此设置当发生数学运算异常（ArithmeticException）时不回滚，就可以完成买书的操作。

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    private BookDao bookDao;
    @Override
```

```
    @Transactional(noRollbackFor = ArithmeticException.class)
    //@Transactional(noRollbackForClassName =
"java.lang.ArithmeticException")
    public void buyBook(Integer userId, Integer bookId) {
        //查询图书的价格
        Integer 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);
    }
}
```

3、执行结果

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

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

隔离级别	脏读	不可重复读	幻读
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	√	√

2、使用方式

```
@Transactional(isolation = Isolation.DEFAULT)//使用数据库默认的隔离级别
@Transactional(isolation = Isolation.READ_UNCOMMITTED)//读未提交
@Transactional(isolation = Isolation.READ_COMMITTED)//读已提交
@Transactional(isolation = Isolation.REPEATABLE_READ)//可重复读
@Transactional(isolation = Isolation.SERIALIZABLE)//串行化
```

4.3.9、事务属性：事务传播行为

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。

2、测试

创建一个结算的业务，在该业务中将执行多次购买图书的任务，进而观察余额不足时时执行哪个事务。

创建接口CheckoutService：

```
public interface CheckoutService {
    /**
     * 结账
     * @param userId
     * @param bookIds
     */
    void checkout(Integer userId, Integer[] bookIds);
}
```

创建实现类CheckoutServiceImpl：

```
@Service
public class CheckoutServiceImpl implements CheckoutService {
    @Autowired
    private BookService bookService;
    @Override
    @Transactional
    public void checkout(Integer userId, Integer[] bookIds) {
        for (Integer bookId : bookIds) {
            bookService.buyBook(userId,bookId);
        }

    }
}
```

在BookController中添加方法：

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

然后将数据库中用户的金额修改为100元，此时该用户只够买第一本书，无法同时购买两本书，因此如果执行checkout的事务的话任何一本都无法购买成功，如果执行buyBook的事务的话就可以购买一本，第二本书无法购买。

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，常用的有如下两个：

- @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。
- @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。

当设置为@Transactional(propagation = Propagation.REQUIRED)时两本都无法购买，当设置为@Transactional(propagation = Propagation.REQUIRES_NEW)时可以购买一本。

```
@Service
public class BookServiceImpl implements BookService {
    @Autowired
    private BookDao bookDao;
    @Override
    @Transactional(
            propagation = Propagation.REQUIRES_NEW
    )
    public void buyBook(Integer userId, Integer bookId) {
        //查询图书的价格
        Integer 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);
    }
}
```

4.4、基于XML的声明式事务

4.3.1、场景模拟

与基于注解的声明式事务一致

基于xml实现的声明式事务，必须引入aspectJ的依赖：

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>5.3.1</version>
</dependency>
```

4.3.2、修改Spring配置文件

将Spring配置文件的内容修改为如下内容：

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

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

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

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

```
    </bean>

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

    <!--配置事务管理器-->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <!--装配数据源-->
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!-- tx:advice标签：配置事务通知 -->
    <!-- id属性：给事务通知标签设置唯一标识，便于引用 -->
    <!-- transaction-manager属性：关联事务管理器 -->
    <tx:advice id="tx" transaction-manager="transactionManager">
        <tx:attributes>
            <!-- tx:method标签：配置具体的事务方法 -->
            <!-- name属性：指定方法名，可以使用星号代表多个字符 -->
            <tx:method name="buyBook"/>
            <!-- read-only属性：设置只读属性 -->
            <!-- rollback-for属性：设置回滚的异常 -->
            <!-- no-rollback-for属性：设置不回滚的异常 -->
            <!-- isolation属性：设置事务的隔离级别 -->
            <!-- timeout属性：设置事务的超时属性 -->
            <!-- propagation属性：设置事务的传播行为 -->
        </tx:attributes>
    </tx:advice>
    <!-- 配置事务通知和切入点表达式 -->
    <aop:config>
        <aop:advisor advice-ref="tx" pointcut="execution(*
com.tianna.spring.service.impl.*.*(..))"></aop:advisor>
    </aop:config>

</beans>
```