Spring

一、Spring 基础

1. 课程目标

- 掌握 Spring 核心

2. 知识架构树

- Spring 概述
- Spring IoC
- Spring Aop

3. 理论知识

3.1 Spring 概述

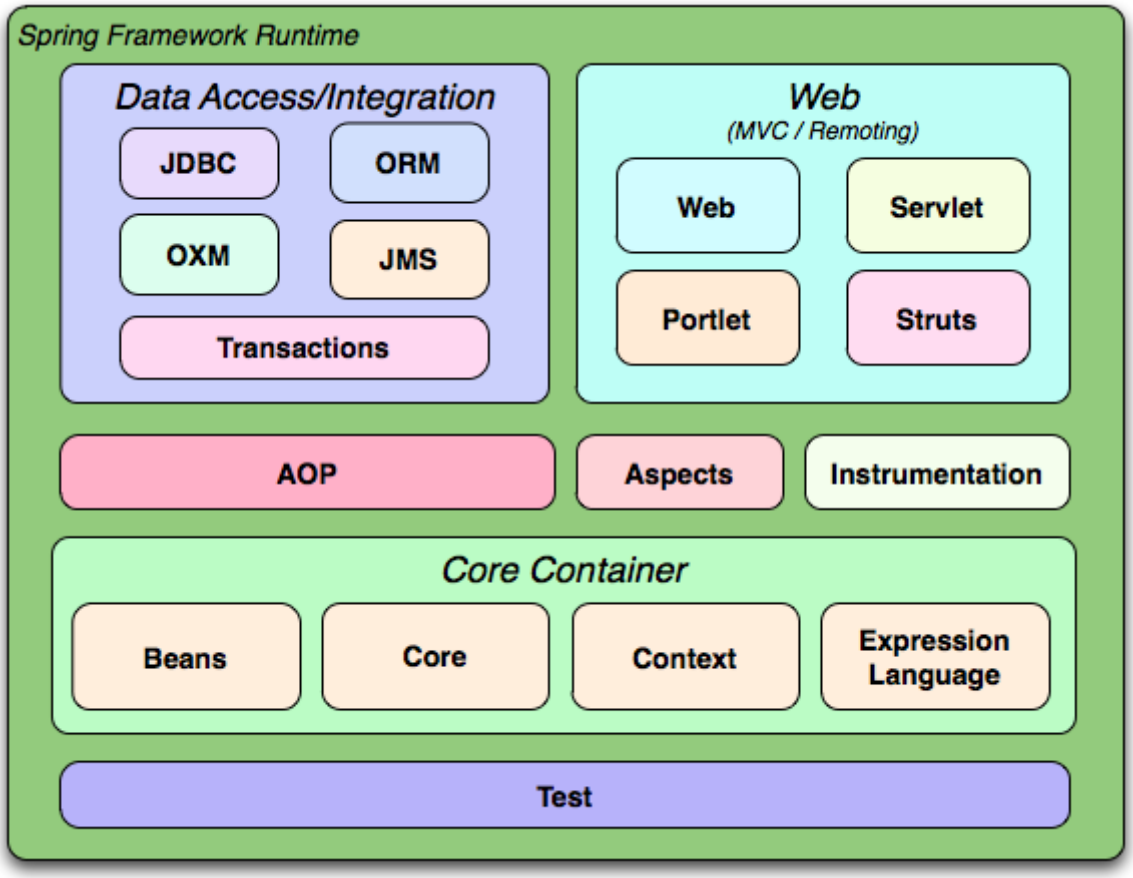
概述

> 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。
>
> Spring 官网：https://spring.io

特性

- 非侵入式：基于 Spring 开发的应用中的对象可以不依赖于 Spring 的API
- **控制反转**：IoC——Inversion of Control，指的是将对象的创建权力交给 Spring 容器去创建。使用 Spring 之前，对象的创建都是由我们自己在代码中通过 new 关键字创建。而使用 Spring 之后。对象的创建都是由给了 Spring 框架。还能去维护对象之间的关系。
- **依赖注入**：DI——Dependency Injection，是指依赖的对象不需要手动调用 setXX 方法去设置，而是通过**配置赋值**。
- **面向切面编程**：Aspect Oriented Programming——AOP，主要用于功能拓展。
- 容器：Spring 是一个容器，因为它包含并且管理应用对象（自己+三方）的生命周期
- 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。
- 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（实际上 Spring 自身也提供了表述层的 SpringMVC 和持久层的 Spring JDBC）

组件



- Data Access/Integration：数据访问/一致性；主要操作持久层，包括 JDBC、ORM 映射关系、Transactions 事务等；
- Web：Web 组件；用于 B/S 架构开发，包括 Web 组件，Servlet 组件等；
- AOP：AOP 组件；
- Core Container：核心组件；主要用于容器对象构建。包括 Beans 组件、Core 核心、Context 上下文、EL 表达式；
- Test：测试组件

3.2 Spring Ioc

> Inversion of Control（IoC）控制反转；IoC不是技术，是一种设计思想。把创建对象以及管理对象之间关系的权力，交给第三方容器（Spring容器），当容器加载时，自动依靠配置创建对象，完善对象之间的关系；
>
> IoC 核心的操作就是创建对象和管理对象的关系，帮助实现核心需求；

3.2.1 Spring 原理

Java中创建对象的方式：

```
1、new关键字（对已知对象的创建）
    Student stu = new Student();

2、克隆，继承自 Object（对已知对象的创建）
    Student stu1 = (Student)stu.clone();

3、反射(重点)
    // 类的全限定名称，包名+类名（逻辑名字，不一定存在）
    Student stu2 =
(Student)Class.forName("com.soft.entity.Student").newInstance();

4、反序列化
```

原理分析：

```
分析：通过反射获取对象
    反射获取对象需要什么？
        Class.forName("动态的类的路径").newInstance();
        类的全路径（全限定路径：包名+类名）：com.soft.xxx.xx
            com.soft.entity.Student
            com.soft.entity.Teacher
            com.soft.servlet.IndexServlet
            com.soft.controller.IndexController

    类的全路径怎么去管理？
        通过xml的形式管理，是标签的形式编写的，层级逻辑非常清晰

        1、类和类之间的关系
        2、对于类可以给定单独的属性

在Spring中，把对象称为 bean，每一个 bean 就是一个对象
    bean：对象的路径、对象的属性、属性对应的值
    beans：对象的集合
```

解析 xml 的方式：dom4j

xml头信息：schema/dtd 1）用于描述该文档的基本信息。 2）规定标签的种类，不能写自定义标签 3）规定标签的书写个数 4）规定标签的书写顺序 5）提醒功能

```java
package com.soft.entity;

public class Student {
    private String no;
    private String name;
    private String sex;
    private String age;

    // 忽略set/get方法，也可以采用 lombok 帮助生成 set/get 方法
}
```

```
<!--
    管理的对象集合
    <beans>：根标签，跟又称为root
```

```
-->
<beans>
    <!--通过配置文件管理Student对象-->
    <bean id="zs" class="com.soft.entity.Student">
        <!--描述学号-->
        <property name="no" value="1001"></property>
        <!--描述姓名-->
        <property name="name" value="zs"></property>
        <!--描述性别-->
        <property name="sex" value="男"></property>
        <!--描述年龄-->
        <property name="age" value="18"></property>
    </bean>

    <bean id="ls" class="com.soft.entity.Student">
        <!--描述学号-->
        <property name="no" value="1002"></property>
        <!--描述姓名-->
        <property name="name" value="ls"></property>
        <!--描述性别-->
        <property name="sex" value="男"></property>
        <!--描述年龄-->
        <property name="age" value="19"></property>
    </bean>
</beans>
```

导入jar包：

```
<dependency>
  <groupId>org.dom4j</groupId>
  <artifactId>dom4j</artifactId>
  <version>2.1.3</version>
</dependency>
```

功能实现：

```
package com.soft.factory;

import com.soft.entity.Student;
import org.dom4j.Document;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.junit.Before;
import org.junit.Test;

import java.io.InputStream;
import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * 设计模式：工厂模式
 * 模拟Spring的原理
```

```java
 */
public class BeanFactory2 {

    // 所有对象
    Map<String, Object> map = null;

    /**
     * 加载XML文件
     * @return
     * @throws Exception
     */

    public Document loadXml() throws Exception{
        // ----------------------------------------------------------------
--
        // 一、加载配置文件
        // 1、获取bean.xml文件
        SAXReader saxReader = new SAXReader();
        // 要找bean.xml不能找绝对路径，要依据当前类找相对路径
        InputStream inputStream =
BeanFactory2.class.getResourceAsStream("/bean.xml");
        // 读取bean.xml，需要流，返回Document对象
        // Document：文档对象包含了bean.xml的内容
        Document document = saxReader.read(inputStream);
        return document;
        // ----------------------------------------------------------------
--
    }

    /**
     * 解析XML文件并封装Map对象
     * @throws Exception
     */
    @Before
    public void createBeanMap() throws Exception{
        // ----------------------------------------------------------------
--
        // 二、解析文件
        // 2、解析bean.xml（通过Document对象获取内容）

        // 获取文档的跟标签：<beans>
        Element rootElement = loadXml().getRootElement();
        // 获取跟标签下的所有直接子标签：<bean>
        // 这个集合中放的是一个一个对象
        List<Element> beanList = rootElement.elements();

        if(map == null){
            map = new HashMap<>();
        }

        // 遍历bean标签的集合
        for (Element bean : beanList) {
            // 获取bean标签的id，这个id就是对象的唯一表示
            String id = bean.attributeValue("id");
            // 获取bean标签的class，
```

```
            // 也就获取到类的全限定名称
            // 有了全限定名称就可以通过反射创建对象
            String clazz = bean.attributeValue("class");

            // 根据类的权限名称获取类对象，类对象中有属性和方法
            Class<?> aClass = Class.forName(clazz);
            // 通过反射创建对象
            Object o = aClass.newInstance();

            // 把当前类中的所有方法都获取到
            Method[] methods = aClass.getDeclaredMethods();

            // 存放公共方法的集合，key是方法名字小写，value是方法对象
            Map<String, Method> methodMap = new HashMap<>();

            // 遍历方法对象数组
            for (Method method : methods) {
                // 获取方法名
                String name = method.getName();

                // 把名字转成小写，这样有利于通过bean.xml中获取的属性的名字拼接set之后方便比较
                String lowName = name.toLowerCase();

                // 如何把转小写的名字和方法对应起来，还得一一对应
                methodMap.put(lowName, method);
            }

            // ------------------------------------------------
            // 有对象了还得管理对象中的属性，属性属于<bean>标签的子标签<property>
            // 获取<bean>标签的子标签<property>
            List<Element> propList = bean.elements();

            for (Element prop : propList) {
                // 获取标签name属性的值（对象中属性的名字）
                String objPropName = prop.attributeValue("name");
                // 获取标签value属性的值（对象中属性的值）
                String objPropValue = prop.attributeValue("value");

                // 从方法的Map集合中获取指定名称的方法对象
                Method method = methodMap.get("set" + objPropName);

                // 执行方法
                method.invoke(o, objPropValue);
            }
            // ------------------------------------------------

            // 有对象要把对象管理起来，通过map的k-v可以把id和对象关联起来
            map.put(id, o);
        }
        // ----------------------------------------------------------------
    }

    /**
```

```
     * 只获取指定的bean对象
     * @param id
     */
    public Object getBean(String id){
        // ------------------------------------------------------------------
        // 三、从解析后的Map集合中获取指定的对象
         return map.get(id);
        // ------------------------------------------------------------------
    }

    /**
     * 泛型方法
     * @param id
     * @param clazz
     * @param <T>
     * @return
     */
    public <T> T getBean(String id, Class<T> clazz){
        // ------------------------------------------------------------------
        // 三、从解析后的Map集合中获取指定的对象
         return (T)map.get(id);
        // ------------------------------------------------------------------
    }

    @Test
    public void test(){
        Object zs = getBean("zs");
        System.out.println(zs);

        Student stu = (Student)zs;

        stu.getNo();

        Student zs1 = getBean("zs", Student.class);
    }

    /**
     * 这个工厂去分析bean.xml，解析内容创建对象
     */
    // @Test
    public void getBean1() throws Exception {
        // ------------------------------------------------------------------
        // 一、加载配置文件
        // 1、获取bean.xml文件
        SAXReader saxReader = new SAXReader();
        // 要找bean.xml不能找绝对路径，要依据当前类找相对路径
        InputStream inputStream =
BeanFactory2.class.getResourceAsStream("/bean.xml");
        // 读取bean.xml，需要流，返回Document对象
        // Document：文档对象包含了bean.xml的内容
```

```
        Document document = saxReader.read(inputStream);
        // ------------------------------------------------------------------
--


        // ------------------------------------------------------------------
--
        // 二、解析文件
        // 2、解析bean.xml（通过Document对象获取内容）

        // 获取文档的跟标签：<beans>
        Element rootElement = document.getRootElement();
        // 获取跟标签下的所有直接子标签：<bean>
        // 这个集合中放的是一个一个对象
        List<Element> beanList = rootElement.elements();

        // 所有对象
        Map<String, Object> map = new HashMap<>();

        // 遍历bean标签的集合
        for (Element bean : beanList) {
            // 获取bean标签的id，这个id就是对象的唯一表示
            String id = bean.attributeValue("id");
            // 获取bean标签的class，
            // 也就获取到类的全限定名称
            // 有了全限定名称就可以通过反射创建对象
            String clazz = bean.attributeValue("class");

            // 根据类的权限名称获取类对象，类对象中有属性和方法
            Class<?> aClass = Class.forName(clazz);
            // 通过反射创建对象
            Object o = aClass.newInstance();

            // 把当前类中的所有方法都获取到
            Method[] methods = aClass.getDeclaredMethods();

            Map<String, Method> methodMap = new HashMap<>();

            for (Method method : methods) {
                // 获取方法名
                String name = method.getName();

                // 把名字转成小写，这样有利于通过bean.xml中获取的属性的名字拼接set之
后方便比较
                String lowName = name.toLowerCase();

                // 如何把转小写的名字和方法对应起来，还得一一对应
                methodMap.put(lowName, method);
            }

            // ----------------------------------------------------
            // 有对象了还得管理对象中的属性，属性属于<bean>标签的子标签<property>
            // 获取<bean>标签的子标签<property>
            List<Element> propList = bean.elements();

```

```
            for (Element prop : propList) {
                // 获取标签name属性的值（对象中属性的名字）
                String objPropName = prop.attributeValue("name");
                // 获取标签value属性的值（对象中属性的值）
                String objPropValue = prop.attributeValue("value");

                // 从方法的Map集合中获取指定名称的方法对象
                Method method = methodMap.get("set" + objPropName);

                // 执行方法
                method.invoke(o, objPropValue);
            }
            // ------------------------------------------------

            // 有对象要把对象管理起来，通过map的k-v可以把id和对象关联起来
            map.put(id, o);
        }
        // ------------------------------------------------------------------
--


        // ------------------------------------------------------------------
--
        // 三、从解析后的Map集合中获取指定的对象
        System.out.println(map.get("wu"));
        // ------------------------------------------------------------------
--
    }
}
```

3.2.2 Spring Ioc 配置

导入依赖

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.21</version>
</dependency>

会自动导入 spring-core 和 spring-beans 等其他依赖
```

3.2.2.1 基于 XML 配置

编码流程

1. 创建需要被 Spring 管理的类
2. 创建 xml 配置文件，使用 bean 标签配置对象

控制层

```
public class HelloController{}
```

业务层

```
public class HelloService{}
```

持久层

```
public class HelloDao{}
```

实体类

```
public class Student{}
```

配置文件：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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="helloController" class="com.soft.controller.HelloController"/>
    <bean id="helloService" class="com.soft.service.HelloService"/>
    <bean id="helloDao" class="com.soft.dao.HelloDao"/>
    <bean id="student" class="com.soft.entity.Student"/>
</beans>
```

测试

```
public class SpringIocTest {
    @Test
    public void testBean(){
        /*
            ClassPath：代码解析后的文件路径
            Xml：文件类型
            ApplicationContext：应用的上下文环境
        */
        // 根据配置文件获取上下文环境
        ApplicationContext context = new
ClassPathXmlApplicationContext("/bean.xml");

        // 根据上下文获取对象
        HelloController helloController = context.getBean("helloController",
HelloController.class);
        HelloService helloService = context.getBean("helloService",
HelloService.class);
        HelloDao helloDao = context.getBean("helloDao", HelloDao.class);
        Student student = context.getBean("student", Student.class);
    }
}
```

3.2.2.2 基于注解配置

编码流程

1. 创建需要被 Spring 管理的类，在类上使用注解`@Component`。

> 作用是代替 xml 的配置（`<bean id="id名" class="类路径">`）
>
> 其中，id 默认是类名首字母小写，也可以通过注解的 value 属性修改，例如`@Component(value="名字")`或者`@Component("名字")`
>
> 根据分层思想`@Component`注解可以拆拆分为三个注解。控制层：`@Controller`；业务层：`@Service`；持久层：`@Repository`；

2. 创建xml文件，添加注解配置`<context:component-scan base-package="com.soft"/>`。

控制层

```
@Controller
public class HelloController{}
```

业务成

```
@Service
public class HelloService{}
```

持久层

```
@Repository
public class HelloDao{}
```

实体类

```
@Component
public class Student{}
```

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

    <!--目标还是给Student对象赋值，使用注解的形式，要让当前支持注解，要开启注解的开关-->
    <!--
        context：环境
        component：注解名称
        scan：扫描
        base-package：基础包
    -->
    <context:component-scan base-package="com.soft"/>
</beans>
```

测试

```
public class SpringIocTest {
    @Test
    public void testBean(){
        /*
            ClassPath：代码解析后的文件路径
            Xml：文件类型
            ApplicationContext：应用的上下文环境
        */
        // 根据配置文件获取上下文环境
        ApplicationContext context = new
ClassPathXmlApplicationContext("/bean.xml");

        // 根据上下文获取对象
        HelloController helloController = context.getBean("helloController",
HelloController.class);
        HelloService helloService = context.getBean("helloService",
HelloService.class);
        HelloDao helloDao = context.getBean("helloDao", HelloDao.class);
        Student student = context.getBean("student", Student.class);
    }
}
```

3.2.2.3 基于配置类配置

> Spring 3.x 之后推出配置类编码方式，用于取代 xml，使用全注解的形式开发。同时也为 SpringBoot 无配置文件自动配置打下基础。

编码流程

1. 创建需要被 Spring 管理的类。
2. 创建配置类，添加 `@Configuration` 注解。

> `@Configuration` 注解标记配置类，同时该注解继承了 `@Component` 注解，也具备了创建对象的功能，毕竟配置类也是需要实例才可以执行。

3. 配置类中编写带返回值的方法，方法的返回值为需要被 Spring 管理的类型。
4. 方法上添加 `@Bean` 注解

> `@Bean` 注解添加在方法上，表示将方法的返回值注入到 Spring 容器中，同时方法名字是对象在容器中的名字（也可以使用 value 和 那name 属性命名）。

5. 创建xml文件，添加注解配置 `<context:component-scan base-package="com.soft"/>`。

控制层

```
public class HelloController{}
```

业务成

```
public class HelloService{}
```

持久层

```java
public class HelloDao{}
```

实体类

```java
public class Student{}
```

配置类

```java
@Configuration
public class HelloConfig {

    /**
     * 使用 @Bean 注解向容器中注入 HelloController（返回值）对象。
     * 对象在容器中的名字是 controller（方法名）
     */
    @Bean
    public HelloController helloController(){
        // 创建对象并返回
        return new HelloController();
    }

    @Bean
    public HelloService helloService(){
        return new HelloService();
    }

    @Bean
    public HelloDao helloDao(){
        return new HelloDao();
    }

    @Bean
    public Student student1(){
        return new Student("1001", "Tom", "男", 25);
    }

    @Bean("s2")
    public Student student2(){
        return new Student("1001", "Lily", "女", 18);
    }
}
```

测试

```java
public class SpringIocTest {
    @Test
    public void testBean(){
        // 根据配置文件获取上下文环境
        ApplicationContext context = new
AnnotationConfigApplicationContext(HelloConfig.class);

        // 根据上下文获取对象
```

```
        HelloController helloController = context.getBean("helloController", HelloController.class);
        HelloService helloService = context.getBean("helloService", HelloService.class);
        HelloDao helloDao = context.getBean("helloDao", HelloDao.class);

        System.out.println(helloController);
        System.out.println(helloService);
        System.out.println(helloDao);
    }
}
```

3.2.3 Spring DI

> DI（Dependency Injection）即依赖注入，对象之间的依赖由容器在运行期决定，即容器动态的将某个依赖注入到对象之中。

3.2.3.1 Setter注入（属性注入）

1）给注入对象生成set方法 2）创建xml文件，描述所有需要管理的对象，配置对象之间的依赖关系

Controller（控制层）

```
/**
 * 控制层，调用业务成（service）
 */
public class HelloController {

    // 多态写法，使用接口引用对象，便于后期拓展
    private HelloService helloService;

    /**
     * 属性 indexService 的 set 方法
     */
    public void setHelloService(HelloService helloService) {
        this.helloService = helloService;
    }

    /**
     * 查询单个学生的信息
     */
    public void queryStudentInfo(){
        System.out.println(helloService.queryStudentInfo());
    }
}
```

Service（业务层）

```java
/**
 * 业务层的接口
 */
public interface HelloService {
    /**
     * 查询单个学生的信息
     */
    public Student queryStudentInfo();
}
```

```java
/**
 * 业务层的实现类，要调用持久层（dao）
 */
public class HelloServiceImpl implements HelloService {

    // 多态写法，使用接口引用对象，便于后期拓展
    private HelloDao helloDao;

    /**
     * 属性 indexDao 的 set 方法
     */
    public void setHelloDao(HelloDao helloDao) {
        this.helloDao = helloDao;
    }

    /**
     * 查询单个学生的信息
     */
    @Override
    public Student queryStudentInfo() {
        return helloDao.queryStudent();
    }
}
```

Dao（持久层）

```java
/**
 * 持久层接口
 */
public interface HelloDao {
    /**
     * 查询单个学生的信息
     */
    public Student queryStudent();
}
```

```java
/**
 * 持久层的实现类
 */
public class HelloDaoImpl implements HelloDao {

    private Student student;

```

```
    /**
     * 属性 student 的 set 方法
     */
    public void setStudent(Student student) {
        this.student = student;
    }

    /**
     * 查询单个学生的信息
     */
    @Override
    public Student queryStudent() {
        // 通过数据查询学生信息，还没有学过通过Spring连接数据库
        // 直接返回对象，模拟从数据库中查询

        return student;
    }
}
```

Entity

```
public class Student {
    /*
        目标是给类中的属性赋值，使用DI的Setter注入方式，配置文件使用<property>标签
        类中的属性一定要生成Set方法
     */
    private String no;
    private String name;
    private String sex;
    private String age;

    // 省略 set/get/toString 方法，有参和无参构造
}
```

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

    <!-- 将学生类交给 Spring 管理，创建学生对象 -->
    <bean class="setter.com.soft.entity.Student"/>

    <!-- 将学生类交给 Spring 管理，创建张三学生的对象 -->
    <!--
        bean标签的含义就是将配置的对象交给Spring管理
    -->
    <bean id="zs" class="setter.com.soft.entity.Student">
        <!--
            Setter注入，使用<property>标签
            name：
                类中set方法去掉set之后的内容首字母小写。
```

```
                如果去掉set之后，前两个字母是大写的，直接照抄
                    setJSon -> JSon
                    setJson -> json
            value：属性的值，是字面量；
        -->
        <property name="no" value="2001"/>
        <property name="name" value="zs"/>
        <property name="sex" value="男"/>
        <property name="age" value="18"/>
    </bean>

    <!-- 将 Dao 的实现类交给 Spring 管理，创建对象 -->
    <bean id="helloDao" class="setter.com.soft.dao.impl.HelloDaoImpl">
        <!--
            student 是一个对象
            ref：对象引用，使用 bean 标签的 id。
        -->
        <property name="student" ref="zs"/>
    </bean>

    <!-- 将 Service 的实现类交给 Spring 管理，创建对象 -->
    <bean id="helloService"
class="setter.com.soft.service.impl.HelloServiceImpl">
        <property name="helloDao" ref="helloDao"/>
    </bean>

    <!-- 将 Controller 类交给 Spring 管理，创建对象 -->
    <bean id="controller"
class="setter.com.soft.controller.HelloController">
        <property name="helloService" ref="helloService"/>
    </bean>
</beans>
```

测试：

```
public class TestSetter {
    @Test
    public void testStudent(){
        /*
            ClassPath：代码解析后的文件路径
            Xml：文件类型
            ApplicationContext：应用的上下文环境
        */
        // 根据配置文件获取上下文环境
        ApplicationContext context = new
ClassPathXmlApplicationContext("/bean-setter.xml");

        // 根据上下文获取对象
        Student zs = context.getBean("zs", Student.class);

        System.out.println(zs);
    }

    @Test
    public void testController(){
```

```
        /*
            ClassPath：代码解析后的文件路径
            Xml：文件类型
            ApplicationContext：应用的上下文环境
        */
        // 根据配置文件获取上下文环境
        ApplicationContext context = new
ClassPathXmlApplicationContext("/bean-setter.xml");

        // 根据上下文获取对象
        HelloController controller = context.getBean("controller",
HelloController.class);
        controller.queryStudentInfo();
    }
}
```

3.2.3.2 构造器注入

1）给注入对象生成构造方法 2）创建xml文件，描述所有需要管理的对象，配置对象之间的依赖关系

Controller（控制层）

```
public class HelloController {

    HelloService helloService;

    /**
     * 属性 service 的 构造 方法
     */
    public HelloController(HelloService helloService) {
        this.helloService = helloService;
    }

    /**
     * 查询学生信息，要调用Service
     */
    public void queryStudent(){
        System.out.println(indexService.queryStudent());
    }
}
```

Service（业务层）

```
public interface HelloService {
    Student queryStudent();
}
```

```
public class HelloServiceImpl implements HelloService {

    HelloDao helloDao;

    /**
     * 属性 indexDao 的 构造 方法
     */
    public IndexServiceImpl(HelloDao helloDao) {
```

```
        this.helloDao = helloDao;
    }

    @Override
    public Student queryStudent() {
        return helloDao.queryStudent();
    }
}
```

Dao（持久层）

```
public interface HelloDao {
    Student queryStudent();
}
```

```
public class HelloDaoImpl implements HelloDao {

    Student student;

    /**
     * 属性 student 的 构造 方法
     */
    public HelloDaoImpl(Student student) {
        this.student = student;
    }

    @Override
    public Student queryStudent() {
        return student;
    }
}
```

Entity

```
public class Student {
    private String no;
    private String name;
    private String sex;
    private String age;

    // 省略 set/get/toString 方法，有参和无参构造
}
```

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

    <bean id="controller"
class="constructor.com.soft.controller.HelloController">
```

```
        <constructor-arg name="service" ref="service"/>
    </bean>

    <bean id="service"
class="constructor.com.soft.service.impl.HelloServiceImpl">
        <constructor-arg name="dao" ref="dao"/>
    </bean>

    <bean id="dao" class="constructor.com.soft.dao.impl.HelloDaoImpl">
        <constructor-arg name="student" ref="zs"/>
    </bean>

    <!-- 创建张三学生的对象 -->
    <bean id="zs" class="constructor.com.soft.entity.Student">
        <!--
            constructor：构造器
            arg：参数
            标签属性：
                定位属性：
                    name：构造器参数的名字
                    index：构造器参数的下标位置，从0开始
                    type：构造器参数的类型，类型的全限定名称

                赋值属性：
                    value：字面量
                    ref：对象
        -->
        <constructor-arg name="no" value="2002"/>
        <constructor-arg index="1" value="zs"/>
        <constructor-arg name="sex" value="男"/>
        <constructor-arg index="3" value="22"/>
    </bean>
</beans>
```

测试：

```
public class TestConstructor {
    @Test
    public void testStudent(){
        /*
            ClassPath：代码解析后的文件路径
            Xml：文件类型
            ApplicationContext：应用的上下文环境
        */
        // 根据配置文件获取上下文环境
        ApplicationContext context = new
ClassPathXmlApplicationContext("/bean-constructor.xml");

        // 根据上下文获取对象
        Student zs = context.getBean("zs", Student.class);

        System.out.println(zs);
    }
    @Test
    public void testController(){
```

```
        /*
            ClassPath：代码解析后的文件路径
            Xml：文件类型
            ApplicationContext：应用的上下文环境
        */
        // 根据配置文件获取上下文环境
        ApplicationContext context = new
ClassPathXmlApplicationContext("/bean-constructor.xml");

        // 根据上下文获取对象
        IndexController controller = context.getBean("controller",
IndexController.class);

        controller.queryStudent();
    }
}
```

3.2.2.3 基于注解的注入

1）在需要对象管理的类上添加注解`@Component`

2）在注入的属性上添加注解：`@Autowired + @Qualifier`或者`@Resource`

3）创建xml文件，添加注解配置`<context:component-scan base-package="com.soft"/>`。

```
类管理（创建对象）：
    @Component：通用注解
        @Controller：控制层
        @Service：业务层
        @Repository：持久层

属性管理（值注入）：
    Spring:
        @Autowired：注入对象
        @Qualifier：根据名字注入对象，搭配 @Autowired 使用
        @Value：注入字面量

    JDK:
        @Resource：注入对象，可以根据名字，也可以根据类型
```

Controller（控制层）

```
// @Component
@Controller
public class HelloController {

    @Autowired
    HelloService helloService;

    public void queryStudent(){
        System.out.println(helloService.queryStudent());
    }
}
```

Service（业务层）

```java
public interface HelloService {
    Student queryStudent();
}
```

```java
// @Component
@Service
public class HelloServiceImpl implements HelloService {

    /*@Autowired
    @Qualifier("dao2")*/
    /*
        @Resource：它不是Spring的注解
            默认根据名称查找，如果配置的有名称，则必须按照名称找
            如果没有名称，则按照类型找
    */
    @Resource(name = "dao2")
    HelloDao helloDao;

    @Override
    public Student queryStudent() {
        return helloDao.queryStudent();
    }
}
```

Dao（持久层）

```java
public interface HelloDao {
    Student queryStudent();
}
```

```java
// @Component("dao1")
@Repository("dao1")
public class HelloDaoImpl1 implements HelloDao {
    /*
        对对象属性进行赋值，默认按照类型从整个Spring容器中找
     */
    @Autowired
    Student student;

    @Override
    public Student queryStudent() {
        student.setName("Jack");
        return student;
    }
}
```

```java
@Component("dao2")
public class HelloDaoImpl2 implements HelloDao {
    /*
        对对象属性进行赋值，默认按照类型从整个Spring容器中找
     */
    @Autowired
```

```
    Student student;

    @Override
    public Student queryStudent() {
        student.setName("Tom");
        return student;
    }
}
```

Entity（实体类）

```
package annotation.com.soft.entity;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

/*
    @Component：将当前类交给Spring管理
        value：就是当前类的唯一表示名字，默认值是空串，
            当前类的唯一标识名字就是当前类名或者类名首字母小写
            Student
            student

    注解代替了原来XML中的如下代码：
        <bean id="zs" class="xml.com.soft.entity.Student">
 */
public class Student {
    // 目标还是给Student对象赋值
    @Value("3001")
    private String no;
    @Value("Tom")
    private String name;
    @Value("女")
    private String sex;
    @Value("23")
    private String age;

    // 省略 set/get/toString 方法，有参和无参构造
}
```

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

    <!--目标还是给Student对象赋值，使用注解的形式，要让当前支持注解，要开启注解的开关-->
    <!--
        context：环境
```

```
        component：注解名称
        scan：扫描
        base-package：基础包
    -->
    <context:component-scan base-package="annotation.com.soft"/>
</beans>
```

测试：

```
public class TestAnnotation {
    @Test
    public void test {
        ApplicationContext context = new
ClassPathXmlApplicationContext("bean-annotation.xml");

        StudentController studentController = context.getBean("controller",
StudentController.class);
        studentController.login();
        studentController.regist();
    }
}
```

@Autowired和@Resource的区别：

1、`@Autowired` 默认根据类型注入对象，也可以搭配 `@Qualifier` 实现根据名称注入对象； 2、`@Resource` 默认根据名称注入对象，找不到对应名称时，根据类型注入对象；一旦 `@Resource` 指定了名称，则必须按照名称匹配。 3、`@Autowired` 是 Spring 的注解，`@Resource` 是 JDK 的注解（推荐使用，可以解决耦合度的问题。）；

3.2.2.4 基于配置类注入

1. 给方法添加参数（需要注入的类型）
2. 配置类加载时，会将容器中的对象设置到对应的方法的参数中。默认按照类型注入，也可以搭配 `@Qualifier` 注解实现名称注入。

配置类：

```
@Configuration
public class HelloConfig {

    /**
     * 方法的参数是需要注入的内容。
     * 配置类加载时从容器中找 HelloService 类型给方法的参数赋值
     */
    @Bean
    public HelloController helloController(HelloService helloService){
        System.out.println("配置类：" + helloService);
        // 创建对象并返回
        return new HelloController();
    }

    @Bean
    public HelloService helloService(HelloDao helloDao){
        System.out.println("配置类：" + helloDao);
        return new HelloService();
```

```
    }

    /**
     * 根据名字注入
     */
    @Bean
    public HelloDao helloDao(@Qualifier("s1") Student student){
        System.out.println("配置类：" + student);
        return new HelloDao();
    }

    @Bean("s1")
    public Student student1(){
        return new Student("1001", "Tom", "男", 25);
    }

    @Bean("s2")
    public Student student2(){
        return new Student("1001", "Lily", "女", 18);
    }
}
```

3.2.4 对象作用域

> 默认情况下，Spring 容器对每个 bean 元素只会创建一个实例对象，这个实例对象中的内容被容器中的其他对象共享，即单例模式：singleton；
>
> 如果将作用域设置为 prototype，则每次调用就会在容器中创建一个新的对象，实例对象的内容不能被其他对象共享，即多例模式：prototype；

（1）基于XML配置

Controller

```
public class StudentController{
    String msg;

    public String getMsg() {
        System.out.println(this.msg);
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }
}
```

xml 配置

```xml
<!--单例-->
<bean id="sc" class="com.soft.controller.StudentController"
scope="singleton"/>

<!--多例-->
<bean id="sc" class="com.soft.controller.StudentController"
scope="prototype"/>
```

（2）基于注解配置

```java
@Controller
@Scope("singleton")
public class StudentController{
    String msg;

    public String getMsg() {
        System.out.println(this.msg);
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }
}
```

```java
@Controller
@Scope("prototype")
public class StudentController{
    String msg;

    public String getMsg() {
        System.out.println(this.msg);
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }
}
```

测试

```
public static void main(String[] args) {
    ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("bean.xml");

    StudentController controller1 = context.getBean("studentController",
StudentController.class);
    controller1.setMsg("对象");
    controller1.getMsg();

    StudentController controller2 = context.getBean("studentController",
StudentController.class);
    controller2.getMsg();
}
```

3.3 Spring AOP

> IOC 的本质就是保证**基础的功能(CRUD)能正常完成**，正常的功能在完成过程中可能会需要其他的需求来加持基础功能，让基础功能更加的牢靠。基础代码已经完成，想要添加一些功能来加持基础功能必定要修改原有的代码，同时也会给已经完成的功能造成一些影响，这种影响是不想发生的。由于加持的功能并不严格影响基础功能，同时这个加持的功能有是可以单独运行的。如何把两个功能融合起来，就需要将加持的功能插入到需要的地方，不需要是也可去掉。添加和去掉的过程就行形成了一个可插拔式的效果。

3.3.1 AOP 原理（代理模式）

```
需求：购买一台手机（核心功能）
途径：通过以下两种方式购买到的产品都是一样的，不一样的地方在于，代理商可能会比官网要有更多的优惠
    1、官方购买
        手机（基础功能）
    2、代理商购买
        手机（基础功能）、赠送话费、钢化膜、手机壳（增加的功能）

代理对象（代理商）：增加额外功能之后的对象
被代理对象（目标对象）：核心业务所在对象
```

静态代理： 1、要求有一个接口 2、目标对象、代理对象都要实现同一个接口

```
// 接口
public interface Sell {
    void sellPhone();
}
```

```
/**
 * 目标对象：官方旗舰店
 */
public class FlagshipShop implements Sell {
    @Override
    public void sellPhone() {
        System.out.println("出售的手机");
    }
}
```

```java
/**
 * 代理对象：第三方商店
 */
public class ThirdShop implements Sell {
    @Override
    public void sellPhone() {
        // 第三方活动
        System.out.println("10大庆，入场即可领取福利！");

        // 原有业务逻辑
        FlagshipShop fs = new FlagshipShop();
        fs.sellPhone();

        System.out.println("活动还剩下最后一天！");
    }
}
```

```java
// 测试
public class StaticProxyTest {
    public static void main(String[] args) {
        /*FlagshipShop flagshipShop = new FlagshipShop();
        flagshipShop.sellPhone();*/

        ThirdShop ts = new ThirdShop();
        ts.sellPhone();
    }
}
```

缺点：因为代理对象，需要与目标对象实现一样的接口。所以会有很多代理类，类太多。一旦接口增加方法，目标对象与代理对象都要维护。

动态代理（JDK代理）：JDK 1.5 增加的功能 1、要求有一个接口 2、创建目标对象（实现接口，可以理解原件） 3、创建 “创建代理对象的工厂”

```java
public interface Sell {
    void sellPhone();
}
```

```java
/**
 * 目标对象：官方旗舰店
 */
public class FlagshipShop implements Sell {
    @Override
    public void sellPhone() {
        System.out.println("出售的手机");
    }
}
```

```java
/**
 * 代理工厂，根据模板孵化代理商
 */
public class ProxyFactory {
    /**
```

```
     * 根据目标对象生成代理对象
     *
     * @param target 目标对象
     * @return 代理对象
     */
    public static Object getProxyBean(Object target) {
        /*
            ClassLoader loader
                模板对象的类加载器

            Class<?>[] interfaces,
                模板对象的的所有接口

            InvocationHandler h
                具体要实现的处理
        */
        return Proxy.newProxyInstance(
                target.getClass().getClassLoader(),
                target.getClass().getInterfaces(),

                (proxy, method, args) -> {
                    // 第三方活动
                    System.out.println("10大庆，入场即可领取福利！");
                    // 执行原有的业务逻辑
                    /*
                        target：模板对象
                        args：模板对象中方法的参数
                        obj：模板对象方法的返回值
                     */
                    Object obj = method.invoke(target, args);

                    System.out.println("活动还剩下最后一天！");

                    return obj;
                });
    }
}
```

```
public class DynamicProxyTest {
    public static void main(String[] args) {
        // 代理对象的工厂
        ProxyFactory pf = new ProxyFactory();
        // 目标对象（原有对象）
        Sell fs = new FlagshipShop();
        // 把原有对象放到工厂中，生成一个代理对象
        Object proxyBean = pf.getProxyBean(fs);

        Sell sell = (Sell)proxyBean;
        sell.sellPhone();
    }
}
```

CGLIB代理

3.3.2 AOP的作用

> AOP **面向切面**的编程思想，把核心业务与增强功能分开，**降低模块之间的耦合度**，形成了可插拔式的结构，在保证核心业务执行不受影响的条件下，可以选择性地插入/去掉增强功能，达到扩展功能的目的。

3.3.3 AOP术语

```
切点（pointcut）：目标对象中的每一个方法都称为切点（切入点的集合）
    目标对象中有：登录、注册、删除、修改等方法

切入点：具体的哪一个切点

连接点（joinpoint）：什么时候添加额外的功能，方法开始前，方法执行后，方法报错了……

增强/通知（advice）：额外添加的那部分功能
    before：前置通知，当方法开始执行时执行。
    afterReturning：后置通知，当方法正常return时执行。
    afterThrowing：异常通知，当方法发生异常时执行。
    after：最终通知，当方法执行完毕之后执行，多用于释放资源。
    around：环绕通知。

目标对象（target）：被代理对象，具有核心的业务的对象
代理（proxy）：代理对象

切面（aspect）：增强和切点表达式组成的内容
织入（weaver）：把额外功能添加到目标对象的过程
```

```
切点表达式：哪个包下的哪个类中的哪个方法需要被增加功能，起的是一个引路人的功能！
    函数（关键字）和表达式组成

语法结构：函数(表达式)

函数：
    execution：用于匹配方法执行的连接点；
    within：用于匹配指定类型内的方法执行；
    this：用于匹配当前AOP代理对象类型的执行方法；注意是AOP代理对象的类型匹配，这样就可能包括引入接口也类型匹配；
    target：用于匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配；
    args：用于匹配当前执行的方法传入的参数为指定类型的执行方法；

    @within：用于匹配所以持有指定注解类型内的方法；
    @target：用于匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；
    @args：用于匹配当前执行的方法传入的参数持有指定注解的执行；
    @annotation：用于匹配当前执行方法持有指定注解的方法；

表达式：目的是告诉spring容器，哪些包/类/方法需要被增强，方法是有路径和返回值的
    *：任意
    ..：0或者多个，在包路径中表示当前包以及子包，在方法路径中表示0个或者多个参数

列子：
    execution(返回值 包路径.类名.方法名(参数))

    execution(* com.soft.service.*.*(..))
```

```
        com.soft.service 包下所有的类的方法都要被增强

    execution(java.lang.String com.soft.dao.*.login*(..))
        com.soft.dao 包下，所有类中，以login开头的所有方法中，返回值为String的方法需要增强

    execution(* com.soft.service.*.insert*(com.soft.entity.Student))
        com.soft.service 包下，以 inset 开头的，且方法的参数是 Student 类型的方法需要给增强。
```

```
aop依赖的jar包：
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
  <version>5.3.20</version>
</dependency>
```

3.3.4 基于XML开发AOP

1、创建通知/增强类，编写增强（before、after、afterReturning、around、afterThrowing） 2、通过XML配置增强类，把增强类交给Spring管理 3、通过XML配置切面

通知类：

```
public class AspectXML {
    /**
     * 前置通知，在方法执行之前执行
     */
    public void before(){
        System.out.println("XML before 前置通知");
    }

    /**
     * 后置通知，在方法return之后执行
     * 通过入参Object接收方法的返回值
     * @param o
     */
    public void afterReturning(Object o){
        System.out.println("XML afterReturning 后置通知" + o);
    }

    /**
     * 环绕通知，在方法执行前执行和方法执行后执行
     * 通过参数ProceedingJoinPoint去执行原有的业务逻辑
     * @param proceedingJoinPoint
     * @return
     * @throws Throwable
     */
    public Object around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
        System.out.println("XML around 环绕通知 开始");
        // 执行原有的业务逻辑
        Object obj = proceedingJoinPoint.proceed();

        System.out.println("XML around 环绕通知 结束");
```

```
        return obj;
    }

    /**
     * 异常时通知
     * 通过参数来指定什么样的异常可以执行该方法
     * @param e
     */
    public void afterThrowing(Exception e){
        System.out.println("XML afterThrowing 异常时通知" + e);
    }

    /**
     * 最终通知，在方法执行之后执行
     */
    public void after(){
        System.out.println("XML after 最终通知");
    }
}
```

xml配置：

```
<?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:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!--增强的业务逻辑类-->
    <bean id="aspectXML" class="com.soft.aspect.AspectXML"/>

    <!--AOP配置-->
    <aop:config>
        <!--
            切面
            ref：依赖增强类的id
            order：执行顺序
        -->
        <aop:aspect ref="aspectXML" order="2">
            <!--把切点表达式抽出来为公共的-->
            <aop:pointcut id="logPointcut" expression="execution(*
com.soft.service.*.*(..))"/>
            <!--前置通知-->
            <aop:before method="before" pointcut-ref="logPointcut"/>
            <!--后置通知-->
            <aop:after-returning method="afterReturning" pointcut-
ref="logPointcut" returning="o"/>
            <!--环绕通知-->
            <aop:around method="around" pointcut-ref="logPointcut"/>
            <!--异常通知-->
            <aop:after-throwing method="afterThrowing"  pointcut-
ref="logPointcut" throwing="e"/>
```

```
            <!--最终通知-->
            <aop:after method="after" pointcut-ref="logPointcut"/>
        </aop:aspect>
    </aop:config>
</beans>
```

3.3.5 基于注解开发AOP

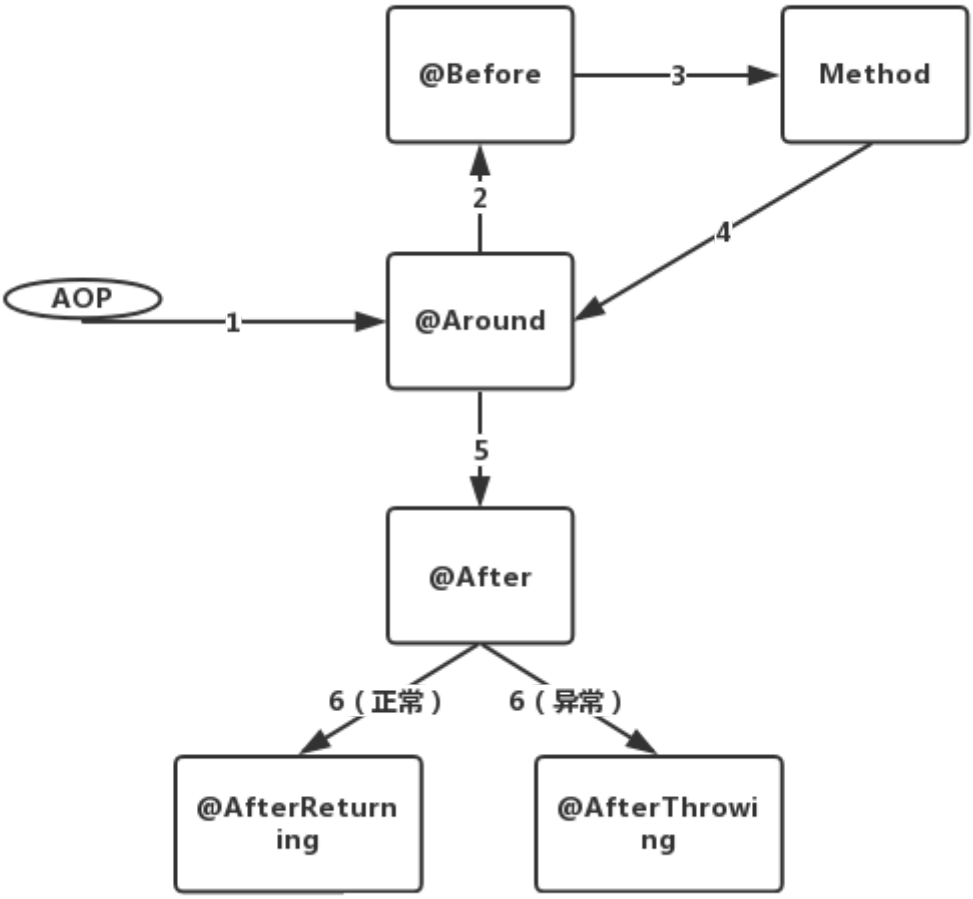
1、创建通知/增强类，编写增强（before、after、afterReturning、around、afterThrowing） 2、通过XML配置注解开关 3、在增强类中添加对应的注解

```
package com.soft.aspect;

@Component // 把当前类交给Spring管理
@Aspect // 当前类是一个增强类/通知类
@Order(1) // 执行顺序，数值越小，优先级越高
public class AspectAnnotation {

    @Pointcut("execution(* com.soft.dao.impl.*.*(..))")
    public void pointCut(){
    }

    /**
     * 前置通知
     * 在方法执行之前添加通知
     */
    @Before("pointCut()")
    public void before(){
        System.out.println("注解方式：前置通知");
    }

    /**
     * 最终通知
     * 在方法执行之后添加通知
     */
    @After("pointCut()")
    public void after(){
        System.out.println("注解方式：最终通知");
    }

    /**
     * 环绕通知
     * 在方法执行之前，return时添加通知
     * @param proceedingJoinPoint
     * @return
     * @throws Throwable
     */
    @Around("pointCut()")
    public Object around(ProceedingJoinPoint proceedingJoinPoint) throws
Throwable {
        System.out.println("注解方式：环绕通知----开始");

        // 执行目标方法
        Object proceed = proceedingJoinPoint.proceed();

```

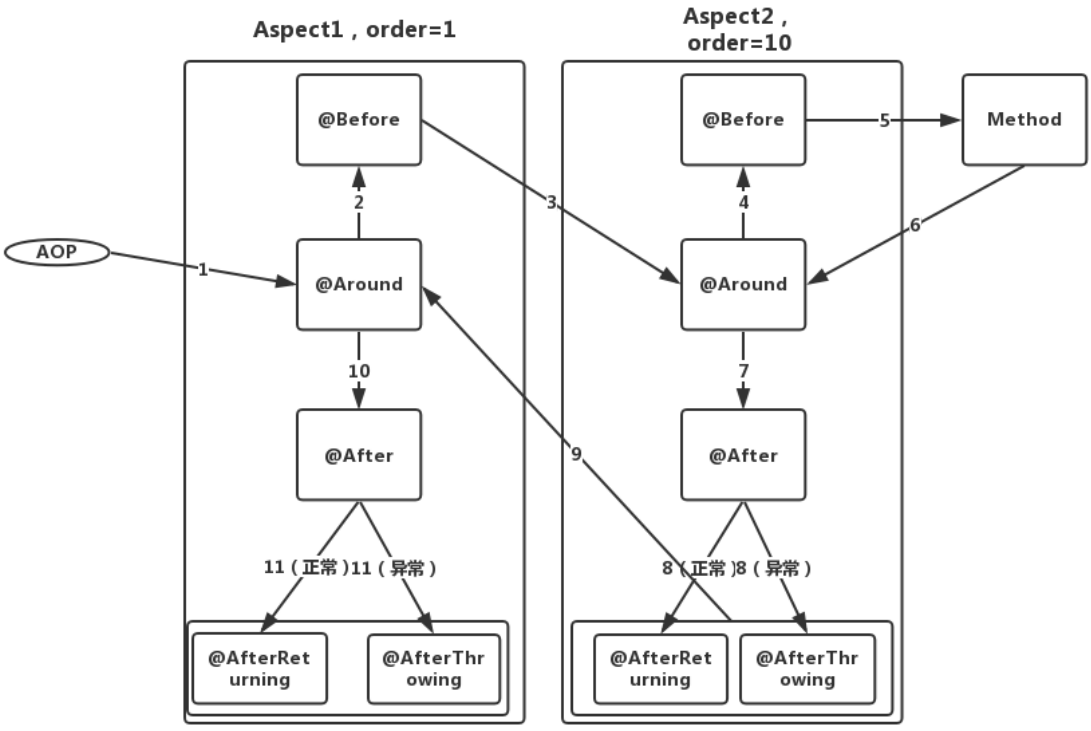
```
        System.out.println("注解方式：环绕通知----方法结束");
        return proceed;
    }

    /**
     * 后置通知
     * 在方法return之后添加通知
     * @param o
     */
    @AfterReturning(value = "pointCut()", returning = "o")
    public void afterReturning(Object o){
        System.out.println("注解方式：后置通知");
    }

    /**
     * 异常通知
     * 方法发生异常时添加通知，可以指定异常类型（触发这个方法的条件）
     */
    @AfterThrowing(value = "pointCut()", throwing = "e")
    public void afterThrowing(Throwable e){
        System.out.println("注解方式：异常通知");
    }
}
```

```
<?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:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!--AOP注解的开关-->
    <aop:aspectj-autoproxy/>
</beans>
```

3.3.6 单个切面中通知的执行顺序



3.3.7 多个切面通知的执行顺序



切面执行顺序总结：

```
在不指定order排序的情况下：
    1、xml配置按照配置的先后顺序执行
    2、注解配置按照切面类类名的字典排序执行
    3、xml和注解同时存在，xml优先执行

在指定order排序的情况下：
    按照order给定的数值从小到大执行，数值越小，执行顺序越优先；
```

3.3.8 AOP的应用场景

事务、日志、权限控制等

简单日志框架实现：

```
package com.soft.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;

/**
 * 实现日志的需求
 * 1、方法的执行开始
 * 2、方法的执行结束
 * 3、方法的执行结果
 * 4、方法的执行参数
 * 5、方法的执行异常
 */
@Component
@Aspect
@Order(0)
public class LogAspect {
    @Around("execution(* com.soft.service.*.*(..))")
    public Object around(ProceedingJoinPoint proceedingJoinPoint) throws
Throwable {
        // 日期格式化
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String format = sdf.format(new Date());
        // 核心的业务对象（方法）
        Object target = proceedingJoinPoint.getTarget();
        // 截取字符串去掉地址符
        String s = target.toString();
        int index = s.lastIndexOf("@");
        String substring = s.substring(0, index);

        System.out.println(red("[日志] " + format + "\t" + substring + " 开始
执行...."));
        // 方法执行的参数
```

```
        Object[] args = proceedingJoinPoint.getArgs();
        // 借助数组的工具类
        System.out.println(red("[日志] " + format + "\t" + "参数：" +
Arrays.toString(args)));
        // 方法执行的结果
        Object proceed = proceedingJoinPoint.proceed();
        System.out.println(red("[日志] " + format + "\t" + substring + " 结束
执行，结果：" + proceed));
        return proceed;
    }

    @AfterThrowing(value = "execution(* com.soft.service.*.*(..))", throwing
= "e")
    public void afterThrowing(Exception e){
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String format = sdf.format(new Date());
        System.out.println(red("[日志] " + format + "\t" + "异常：" + e));
    }

    public String red(String old){
        return "\033[1;31m" + old + "\033[0m";
    }
}
```

4. 综合实验

- 搭建 Spring 项目，实现项目构建。

5. 作业实践

- 搭建 Spring 项目，基于注解实现控制层、业务层、持久层之间的调用

二、Spring 高级

1. 课程目标

- 掌握 Spring 配置数据源
- 掌握 Spring 事务配置
- 了解 Spring 操作 JDBC

2. 知识架构树

- Spring JDBC
- Spring 事务
- Spring 整合 Junit4

3. 理论知识

3.1 Spring JDBC

Oracle

```
-- 建表语句
create table tb_student(
```

```
    no integer primary key not null,
    name varchar2(200),
    sex char(1),
    tel varchar2(11),
    address varchar2(200),
    class varchar2(20),
    del_flg char(1) default 1
);

-- 序列
create sequence seq_student
start with 10000
increment by 1;

-- 导入数据
insert into tb_student values(seq_student.nextval, 'Tom', '1',
'18203908190', '河南郑州', 'Java1', '1');
insert into tb_student values(seq_student.nextval, 'Jack', '1',
'18203908191', '河南郑州', 'Java1',  '1');
insert into tb_student values(seq_student.nextval, 'Peter', '1',
'18203908192', '河南郑州', 'Java1', '1');
insert into tb_student values(seq_student.nextval, 'Lily', '0',
'18203908193', '河南郑州', 'Java1', '1');
insert into tb_student values(seq_student.nextval, 'Marry', '0',
'18203908194', '河南郑州', 'Java1', '1');
commit;
```

MySQL

```
create table tb_student(
    no integer primary key not null auto_increment,
    name varchar(200),
    sex char(1),
    tel varchar(11),
    address varchar(200),
    class varchar(20),
    del_flg char(1) default 1
);


insert into tb_student(name, sex, tel, address, class, del_flg)
values
('Tom', '1', '18203908190', '河南郑州', 'Java1', '1'),
('Jack', '1', '18203908191', '河南郑州', 'Java1',  '1'),
('Peter', '1', '18203908192', '河南郑州', 'Java1', '1'),
('Lily', '0', '18203908193', '河南郑州', 'Java1', '1'),
('Marry', '0', '18203908194', '河南郑州', 'Java1', '1');
commit;
```

3.1.1 JDBC配置

1、导入数据库驱动

```
<!-- Oracle 驱动 -->
<dependency>
    <groupId>ojdbc</groupId>
    <artifactId>ojdbc</artifactId>
    <version>6.0</version>
</dependency>
<!-- MySQL 驱动 -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.28</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.3.20</version>
</dependency>
```

2、在Spring容器中配置数据库连接 3、配置JdbcTemplate，并且和数据库连接关联

```
# Oracle连接
oracle.driver=oracle.jdbc.OracleDriver
oracle.url=jdbc:oracle:thin:@localhost:1521:orcl
oracle.username=user
oracle.password=pass

# MySQL连接
mysql.driver=com.mysql.cj.jdbc.Driver
mysql.url=jdbc:mysql://localhost:3306/riu?serverTimezone=GMT
mysql.username=user
mysql.password=pass
```

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

<!--配置SpringJDBC相关的内容-->
    <!--数据库的连接-->
    <!--
        JDBC连接数据库的流程：
            1、加载驱动（驱动包导入项目）
                Class.forName();
            2、获取连接（url、username、password）
                DriverManager(类，对象)
            3、创建SQL
```

```
            4、创建SQL执行器(SQL执行器是依据数据库连接对象创建的)
                Statement
            5、执行SQL
            6、处理结果
            7、关闭连接
    -->
    <!--通过一个对象来获取数据库的连接，在spring中，一个对象就是一个bean-->
    <!--读取properties配置文件-->
    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!-- 数据源 -->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!--驱动、url、账号、密码-->
        <property name="driverClassName" value="${mysql.driver}"/>
        <property name="url" value="${mysql.url}"/>
        <property name="username" value="${mysql.username}"/>
        <property name="password" value="${mysql.password}"/>
    </bean>

    <!--
        可以理解为构建SQL执行器
        在Spring中叫JdbcTemplate，要通过JdbcTemplate执行SQL
        执行SQL在Dao层，在Dao层有JdbcTemplate的属性
    -->
    <bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"/>
    </bean>
</beans>
```

4、在Dao层注入JdbcTemplate，调用方法

```
@Autowired
JdbcTemplate jdbcTemplate;

// 查询单条记录，返回值为Map
Map<String, Object> queryForMap(String sql, @Nullable Object... args)
// 查询多条记录，返回List<Map>
List<Map<String, Object>> queryForList(String sql, @Nullable Object... args)

RowMapper<T> rowMapper = new BeanPropertyRowMapper(Class<T> clazz);
// 查询返回单条记录
T jdbcTemplate.queryForObject(sql, rowMapper, args...);
// 查询多条记录
List<T> jdbcTemplate.query(sql, rowMapper);
List<T> jdbcTemplate.query(sql, rowMapper,  args...);

// 插入、更新、删除，返回操作成功的记录数
jdbcTemplate.update(sql, args...);
```

3.1.2 CRUD 增删改查

```java
@Repository
public class StudentDaoImpl implements StudentDao {

    @Resource
    JdbcTemplate jdbcTemplate;

    @Override
    public int add() {
        String sql = "insert into tb_student(sno, sname, ssex, sbirthday, class, spass, mno) " +
                "values (seq_student.nextval, ?,?,sysdate,?,?,?)";

        int i = jdbcTemplate.update(sql, new String[]{"半月", "0", "95031", "666666", "9002"});

        return i;
    }

    @Override
    public int queryCount() {
        // 创建SQL
        String sql = "select count(1) count from tb_student";

        // 执行SQL
        SqlRowSet sqlRowSet = jdbcTemplate.queryForRowSet(sql);

        // 处理结果
        while(sqlRowSet.next()){
            int i = sqlRowSet.getInt("count");
            return i;
        }
        return 0;
    }

    @Override
    public Map<String, Object> queryForMap() {
        // 创建SQL
        String sql = "select * from tb_student where sno = 156";

        Map<String, Object> map = jdbcTemplate.queryForMap(sql);
        return map;
    }

    @Override
    public List<Map<String, Object>> queryForListMap() {
        // 创建SQL
        String sql = "select * from tb_student where sname like '%' || ? || '%'";

        List<Map<String, Object>> list = jdbcTemplate.queryForList(sql, "张");
        return list;
    }
```

```
50
51      @Override
52      public Student queryForObject() {
53          // 创建SQL
54          String sql = "select sno, sname, ssex, sbirthday, class clazz,
    spass, mno from tb_student where sno = ?";
55
56          // 将实体类和表的单行做映射
57          RowMapper<Student> rowMapper = new BeanPropertyRowMapper<Student>
    (Student.class);
58
59          Student student = jdbcTemplate.queryForObject(sql, rowMapper,
    "156");
60
61          return student;
62      }
63
64      @Override
65      public List<Student> queryForListObject() {
66          // 创建SQL
67          String sql = "select sno, sname, ssex, sbirthday, class clazz,
    spass, mno " +
68                  "from tb_student where sname like '%' || ? || '%'";
69
70          // 将实体类和表的单行做映射
71          RowMapper<Student> rowMapper = new BeanPropertyRowMapper<Student>
    (Student.class);
72
73          List<Student> list = jdbcTemplate.query(sql, rowMapper, "张");
74
75          return list;
76      }
77  }
```

3.1.3 数据库连接池

> 数据库连接池负责**分配、管理**和**释放**数据库连接，它允许应用程序重复使用一个现有的数据库连接，而不是再重新建立一个；释放空闲时间超过最大空闲时间的数据库连接来避免因为没有释放数据库连接而引起的数据库连接遗漏。**能明显提高对数据库操作的性能。**
>
> 连接池基本的思想是在系统初始化的时候，将数据库连接作为对象存储在内存中，当用户需要访问数据库时，并非建立一个新的连接，而是**从连接池中取出一个已建立的空闲连接对象**。使用完毕后，用户也并非将连接关闭，而是将连接放回连接池中，以供下一个请求访问使用。
>
> 而连接的建立、断开都由连接池自身来管理。同时，还可以通过设置连接池的参数来控制连接池中的初始连接数、连接的上下限数以及每个连接的最大使用次数、最大空闲时间等等。也可以通过其自身的管理机制来监视数据库连接的数量、使用情况等。
>
> C3P0（开源）、DBCP（apache）、Druid（阿里）

```
1  <dependency>
2      <groupId>com.alibaba</groupId>
3      <artifactId>druid</artifactId>
4      <version>1.2.8</version>
5  </dependency>
```

```xml
<!--<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">-->
<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="${jdbc.uname}"/>
    <property name="password" value="${jdbc.pwd}"/>
    <!--连接池功能-->
    <!--连接池中最多支持多少个活动会话-->
    <property name="maxActive" value="100"/>
    <!--程序向连接池中请求连接时,超过maxWait的值后，认为本次请求失败，即连接池没有可用连接，单位毫秒，设置-1时表示无限等待-->
    <property name="maxWait" value="5"/>
</bean>

<!--jdbc模板-相当于DBUtil-->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

配置详解：

配置	缺省值	说明
name		配置这个属性的意义在于，如果存在多个数据源，监控的时候可以通过名字来区分开来。如果没有配置，将会生成一个名字，格式是："DataSource-" + System.identityHashCode(this)
jdbcUrl		连接数据库的url，不同数据库不一样。例如：mysql : jdbc:mysql://10.20.153.104:3306/druid2 oracle : jdbc:oracle:thin:@10.20.149.85:1521:ocnauto
username		连接数据库的用户名
password		连接数据库的密码。如果你不希望密码直接写在配置文件中，可以使用ConfigFilter。详细看这里：https://github.com/alibaba/druid/wiki/%E4%BD%BF%E7%94%A8ConfigFilter
driverClassName	根据url自动识别	这一项可配可不配，如果不配置druid会根据url自动识别dbType，然后选择相应的driverClassName
initialSize	0	初始化时建立物理连接的个数。初始化发生在显示调用init方法，或者第一次getConnection时
maxActive	8	最大连接池数量
maxIdle	8	已经不再使用，配置了也没效果
minIdle		最小连接池数量
maxWait		获取连接时最大等待时间，单位毫秒。配置了maxWait之后，缺省启用公平锁，并发效率会有所下降，如果需要可以通过配置useUnfairLock属性为true使用非公平锁。
poolPreparedStatements	false	是否缓存preparedStatement，也就是PSCache。PSCache对支持游标的数据库性能提升巨大，比如说oracle。在mysql5.5以下的版本中没有PSCache功能，建议关闭掉。5.5及以上版本有PSCache，建议开启。
maxOpenPreparedStatements	-1	要启用PSCache，必须配置大于0，当大于0时，poolPreparedStatements自动触发修改为true。在Druid中，不会存在Oracle下PSCache占用内存过多的问题，可以把这个数值配置大一些，比如说100
validationQuery		用来检测连接是否有效的sql，要求是一个查询语句。如果validationQuery为null，testOnBorrow、testOnReturn、testWhileIdle都不会其作用。在mysql中通常为select 'x'，在oracle中通常为select 1 from dual
testOnBorrow	true	申请连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能。
testOnReturn	false	归还连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能
testWhileIdle	false	建议配置为true，不影响性能，并且保证安全性。申请连接的时候检测，如果空闲时间大于timeBetweenEvictionRunsMillis，执行validationQuery检测连接是否有效。
timeBetweenEvictionRunsMillis		有两个含义：1) Destroy线程会检测连接的间隔时间 2) testWhileIdle的判断依据，详细看testWhileIdle属性的说明
numTestsPerEvictionRun		不再使用，一个DruidDataSource只支持一个EvictionRun
minEvictableIdleTimeMillis		Destory线程中如果检测到当前连接的最后活跃时间和当前时间的差值大于minEvictableIdleTimeMillis，则关闭当前连接。
connectionInitSqls		物理连接初始化的时候执行的sql
exceptionSorter	根据dbType自动识别	当数据库抛出一些不可恢复的异常时，抛弃连接
filters		属性类型是字符串，通过别名的方式配置扩展插件，常用的插件有：监控统计用的filter:stat 日志用的filter:log4j 防御sql注入的filter:wall
proxyFilters		类型是List<com.alibaba.druid.filter.Filter>，如果同时配置了filters和proxyFilters，是组合关系，并非替换关系
removeAbandoned		对于建立时间超过removeAbandonedTimeout的连接强制关闭

配置	缺省值	说明
removeAbandonedTimeout		指定连接建立多长时间就需要被强制关闭
logAbandoned		指定发生removeabandoned的时候，是否记录当前线程的堆栈信息到日志中

3.2 Spring 事务

3.2.1 事务回顾

事务的作用：保证一个操作的完整性，要么全部成功，要么全部失败。

事务的四大特性：A（原子性）C（一致性）I（隔离性）D（持久性）

事务的隔离级别：

```
**读未提交：**读到没有提交的数据
    脏读、不可重复读（修改）、幻读（添加/删除）

**读已提交（Oracle）：**读到已经提交的数据。
    不可重复读、幻读

**可重复读（MySQL）：**
    幻读

**序列化读：**一个一个读
    效率低
```

3.2.2 声明式事务管理

3.2.2.1 事务的传播特性

> 就是多个事务方法相互调用时，事务如何在这些方法间传播。 一般情况下我们会把事务设置在 Service 层，执行 Service 中的方法过程中对数据库的增删改操作保持在同一个事务中；如果在 Service 调用 Dao 的过程中还调用了 Service 其他方法，那么其他方法的事务时如何规定的，必须保证在方法中调用的其他方法与本身的方法处在同一个事务中，否则如何保证事物的一致性。

```
PROPAGATION_REQUIRED：如果当前环境中有事务，添加到事务中；如果当前环境中没有事务，那么开启一个新的事务，添加到事务中；
PROPAGATION_SUPPORTS：如果当前环境中有事务，按照有事务执行；如果当前环境中没有事务，按照没有事务执行；
PROPAGATION_NOT_SUPPORTED：不管有没有事务都按照没有事务执行；
PROPAGATION_MANDATORY：如果当前环境中有事务，按照有事务执行；如果当前环境中没有事务，抛出异常；
PROPAGATION_REQUIRES_NEW：始终创建新的事务执行；
PROPAGATION_NEVER：如果当前环境中有事务，抛出异常；
PROPAGATION_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与PROPAGATION_REQUIRED类似的操作
```

3.2.2.2 基于XML的声明式事务管理

```
<!--不是事务的包，他的作用是将Spring提供的事务通过切面的形式添加到代码中-->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.9</version>
</dependency>
```

（1）配置事务管理器

> 常见的事务管理器：JDBC事务管理器、Hibernate事务管理器、JTA分布式事务管理器
>
> 可以理解为增强类，在核心业务基础上增加的功能；

```
<!-- 将SpringJDBC事务管理器添加到Spring容器中 -->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

（2）配置事务属性

```
<!--设置事务的属性-->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <!--
            name：需要添加方法的方法名称，建议使用通配符表达式 *
            isolation：数据库的默认隔离级别，默认是 default
            propagation：事务的传播特性，默认：REQUIRED
            read-only：只读，针对查询操作
            rollback-for：默认情况下对RuntimeException、Error进行回滚，主要解决的是自定义的异常。配置异常时需要写全限定名称（包名 + 类名）
            no-rollback-for：配置不需要回滚的异常
        -->
        <tx:method name="add*" propagation="REQUIRED" isolation="DEFAULT"
no-rollback-for="java.lang.ArithmeticException"/>
        <tx:method name="edit*" propagation="REQUIRES_NEW"
isolation="DEFAULT"/>
        <tx:method name="del*" propagation="REQUIRED" isolation="DEFAULT"/>
    </tx:attributes>
</tx:advice>
```

（3）配置事务切面

```
<!--配置事务的切面-->
<aop:config>
    <!--切点表达式-->
    <aop:pointcut id="transactionPointCut" expression="execution(*
com.soft.service.*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="transactionPointCut"/>
</aop:config>
```

3.2.2.3 基于注解的声明式事务管理

（1）配置事务管理器

```
<!-- 将SpringJDBC事务管理器添加到Spring容器中 -->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

（2）开启事务注解的开关，关联事务管理器

```
<!--开启事务注解的开关-->
<tx:annotation-driven transaction-manager="transactionManager" />
```

（3）在需要使用事务的类或者方法上添加注解@Transactional

`@Transactional` 写在类上，表示类中所有的方法都使用事务。写到方法上，表示该方法使用事务。如果类上和方法都存在，采用方法上的事务配置（就近原则）；

`@Transactional` 的属性：参照xml配置

```
value和transactionManager ：配置事务管理器的名字
propagation：传播特性配置，默认是 Propagation.REQUIRED
isolation：隔离级别，默认是 Isolation.DEFAULT
readOnly：只读，应用与查询
rollbackFor：配置需要回滚的异常，Xxxx.class
rollbackForClassName：配置需要回滚的异常，全限定名称（包+类）
noRollbackFor：配置不需要回滚的异常，Xxxx.class
noRollbackForClassName：配置不需要回滚的异常，全限定名称（包+类）
```

3.3 Spring 整合 Junit4

1、导包

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>5.3.21</version>
</dependency>

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
</dependency>
```

2、编写测试类

```
@RunWith(SpringJUnit4ClassRunner.class)
// classpath不能少，标识的是编译后的文件路径
@ContextConfiguration({"classpath:spring-jdbc.xml"})
public class SpringTestDemo{
    @Autowired
    QueryController queryController;

    @Test
    public void test() {
        System.out.println(queryController);
    }
}
```

4. 综合实验

- 基于 Spring 实现学生表增删改查

5. 作业实践

- Spring 配置数据源
- Spring JDBC 实现增删改查
- 配置事务