Spring

一、Spring 概述

1 Spring 框架是什么

Spring 是于 2003 年兴起的一个轻量级的 Java 开发框架，它是为了解决企业应用开发的复杂性而创建的。Spring 的核心是**控制反转**（IOC）和**面向切面编程**（AOP）。Spring 是可以在 Java SE/EE 中使用的轻量级开源框架。

Spring 的主要作用就是为代码"解耦"，降低代码间的耦合度。就是让对象和对象（模块和模块）之间关系不是使用代码关联，而是通过配置来说明。即在 Spring 中说明对象（模块）的关系。

Spring 根据代码的功能特点，使用 IOC 降低业务对象之间耦合度。IOC 使得主业务在相互调用过程中，不用再自己维护关系了，即不用再自己创建要使用的对象了。而是由 Spring 容器统一管理，自动"注入"，注入即赋值。 而 AOP 使得系统级服务得到了最大复用，且不用再由程序员手工将系统级服务"混杂"到主业务逻辑中了，而是由 Spring 容器统一完成 "注入"。

Spring官网：https://spring.io/

2 Spring 优点

Spring 是一个框架，是一个半成品的软件。由 20 个模块组成。它是一个容器管理对象，容器是装东西的，Spring 容器不装文本，数字。装的是对象。Spring 是存储对象的容器。

1. **轻量**

   Spring 框架使用的 jar 都比较小，一般在 1M 以下或者几百 kb。Spring 核心功能的所需的 jar 总共在 3M 左右。 Spring 框架运行占用的资源少，运行效率高。不依赖其他 jar。

2. **针对接口编程，解耦合**

   Spring 提供了 IOC 控制反转，由容器管理对象，对象的依赖关系。原来在程序代码中的对象创建方式，现在由容器完成。对象之间的依赖解耦合。

3. **AOP 编程的支持**

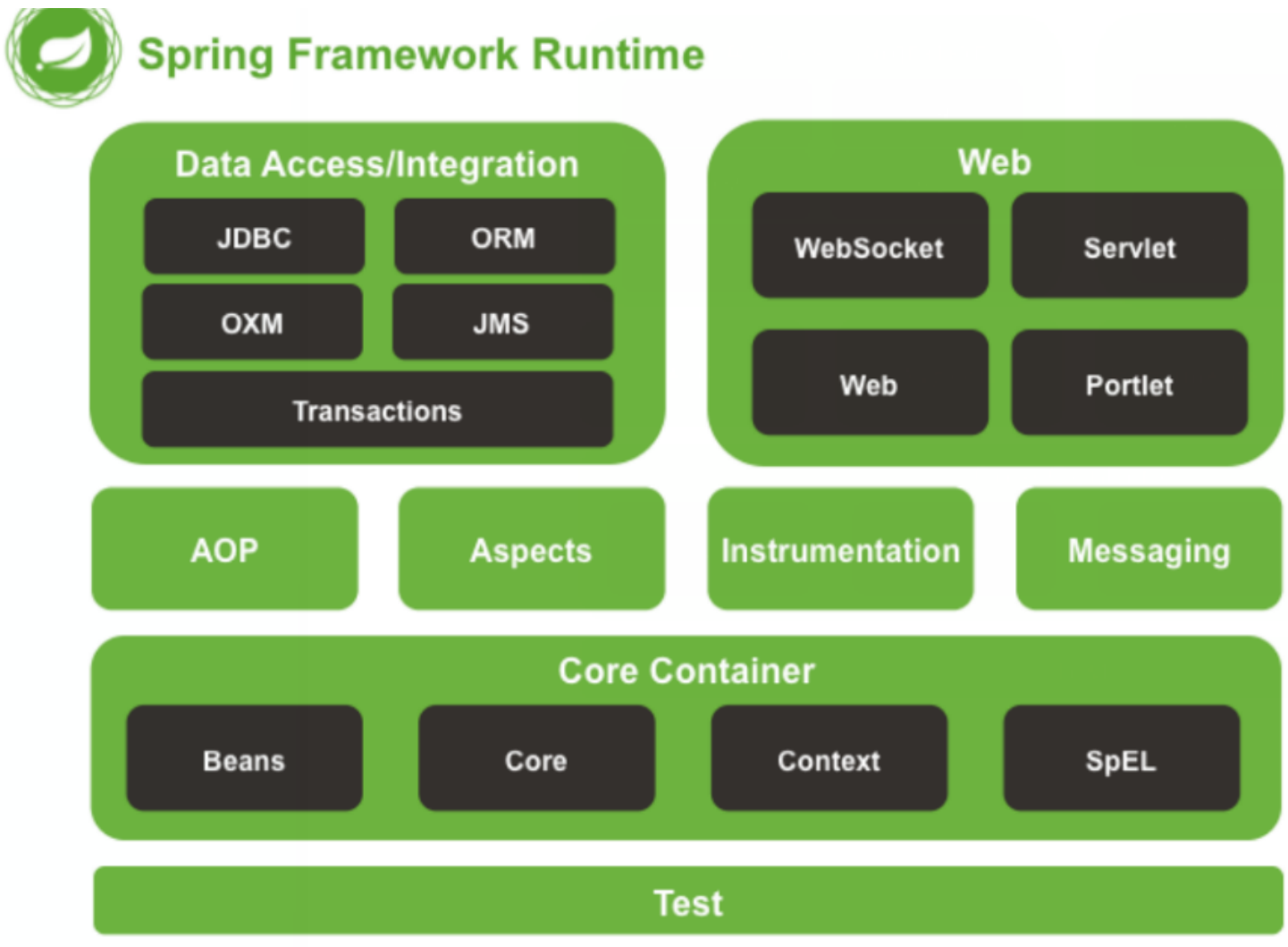
   通过 Spring 提供的 AOP 功能，方便进行面向切面的编程，许多不容易用传统 OOP 实现的功能可以通过 AOP 轻松应付；

   在 Spring 中，开发人员可以从繁杂的事务管理代码中解脱出来，通过声明式方式灵活地进行事务的管理，提高开发效率和质量。

4. **方便集成各种优秀框架**

   Spring 不排斥各种优秀的开源框架，相反 Spring 可以降低各种框架的使用难度，Spring 提供了对各种优秀框架（如 Struts、Hibernate、MyBatis）等的直接支持。简化框架的使用。Spring 像插线板一样，其他框架是插头，可以容易的组合到一起。需要使用哪个框架，就把这个插头放入插线板。不需要可以轻易的移除。

3 Spring 体系结构



Spring 由 20 多个模块组成，它们可以分为数据访问/集成（Data Access/Integration）、 Web、面向切面编程（AOP, Aspects）、提供JVM的代理（Instrumentation）、消息发送（Messaging）、核心容器（Core Container）和测试（Test）。

4 框架怎么学

框架是一个软件，其它人写好的软件。

- 知道框架能做什么，mybatis --> 访问数据库，对表中的数据执行增删改查;
- 框架的语法，框架要完成一个功能，需要一定的步骤支持的;
- 框架的内部实现，框架内部怎么做；原理是什么;
- 通过学习，可以实现一个框架。

5 注意

什么样的对象放入容器中:

- dao类，service类，controller类，工具类
- spring的对象默认都是单例的，在容器中叫这个名称的对象只有一个

不放入到spring容器中的对象:

- 实体类对象，实体类数据来自数据库
- servlet、listener、filter

二、IOC 控制反转

1 IOC 概述

控制反转（IOC，Inversion of Control），是一个概念，是一种思想。指将传统上由程序代码直接操控的对象调用权交给容器，通过容器来实现对象的装配和管理。控制反转就是对对象控制权的转移，从程序代码本身反转到了外部容器。通过容器实现对象的创建，属性赋值，依赖的管理。

IOC 是一个概念，是一种思想，其实现方式多种多样。当前比较流行的实现方式是依赖注入。应用广泛。

依赖：classA 类中含有 classB 的实例，在 classA 中调用 classB 的方法完成功能，即 classA 对 classB 有依赖。

IOC的技术实现:

依赖注入：DI (Dependency Injection)，程序代码不做定位查询，这些工作由容器自行完成。

依赖注入 DI 是指程序运行过程中，若需要调用另一个对象协助时，无须在代码中创建被调用者，而是依赖于外部容器，由外部容器创建后传递给程序。

Spring 的依赖注入对调用者与被调用者几乎没有任何要求，完全支持对象之间依赖关系的管理。

Spring 框架使用依赖注入（DI）实现 IOC。

Spring 容器是一个超级大工厂，负责创建、管理所有的 Java 对象，这些 Java 对象被称为 Bean。Spring 容器管理着容器中 Bean 之间的依赖关系，Spring 使用“依赖注入”的方式来管理 Bean 之间的依赖关系。使用 IOC 实现对象之间的解耦和。

Spring 底层创建对象，使用的是反射机制。

```
控制：创建对象，对象的属性赋值，对象之间的关系管理。
反转：把原来的开发人员管理，创建对象的权限转移给代码之外的容器实现。 由容器代替开发人员管理对象。创建对象，
      给属性赋值。
正转：由开发人员在代码中，使用 new 构造方法创建对象， 开发人员主动管理对象。
      public static void main(String args[]){
         Student student = new Student(); // 在代码中， 创建对象。--正转。
      }
容器：是一个服务器软件，一个框架（spring）
为什么要使用 IOC：目的就是减少对代码的改动，也能实现不同的功能；实现解耦合。

java中创建对象有哪些方式：
  1.构造方法，new Student（）
  2.反射
  3.序列化
  4.克隆
  5.IOC：容器创建对象
  6.动态代理
```

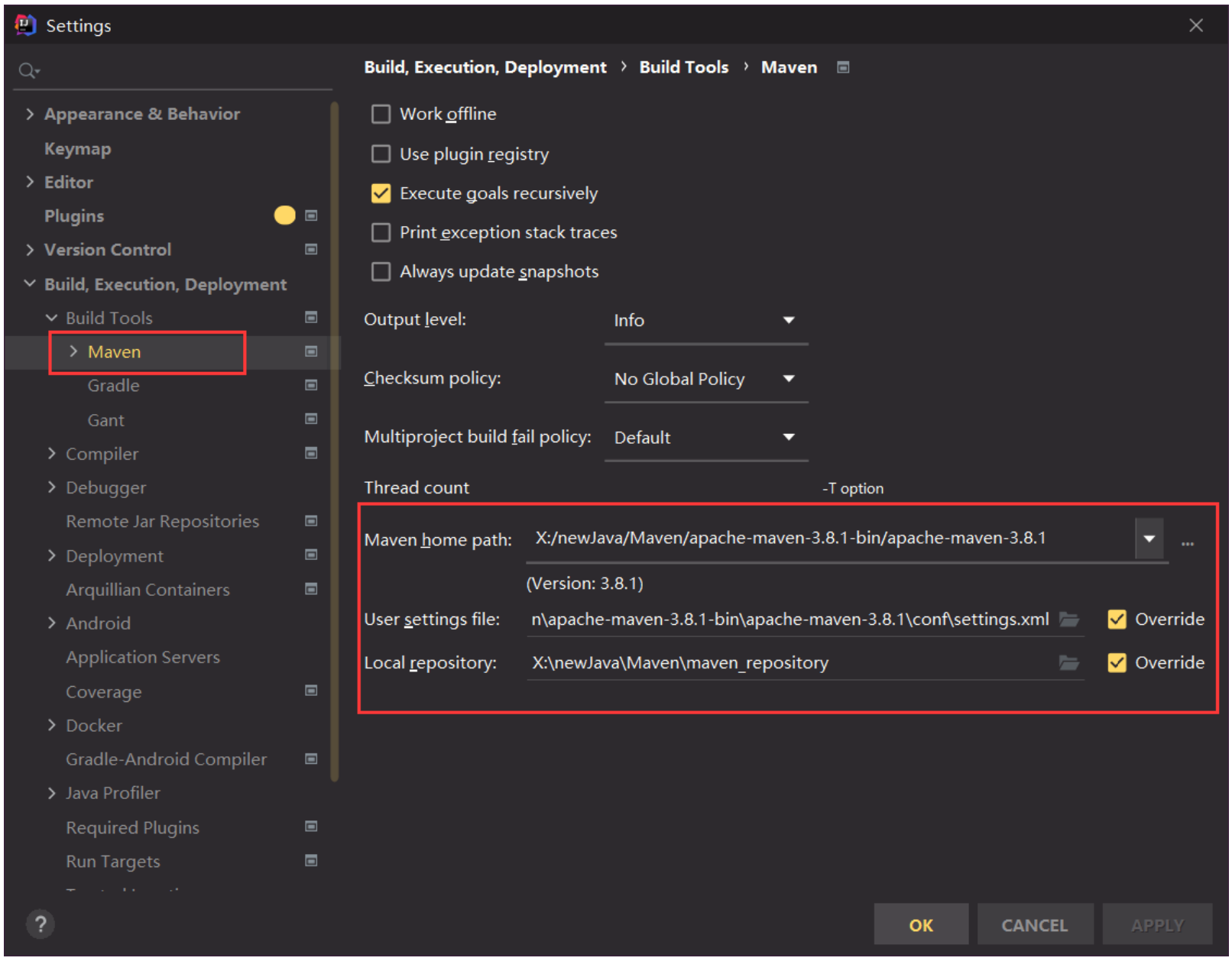
2 开发工具准备

开发工具：idea

依赖管理：maven

jdk：1.8

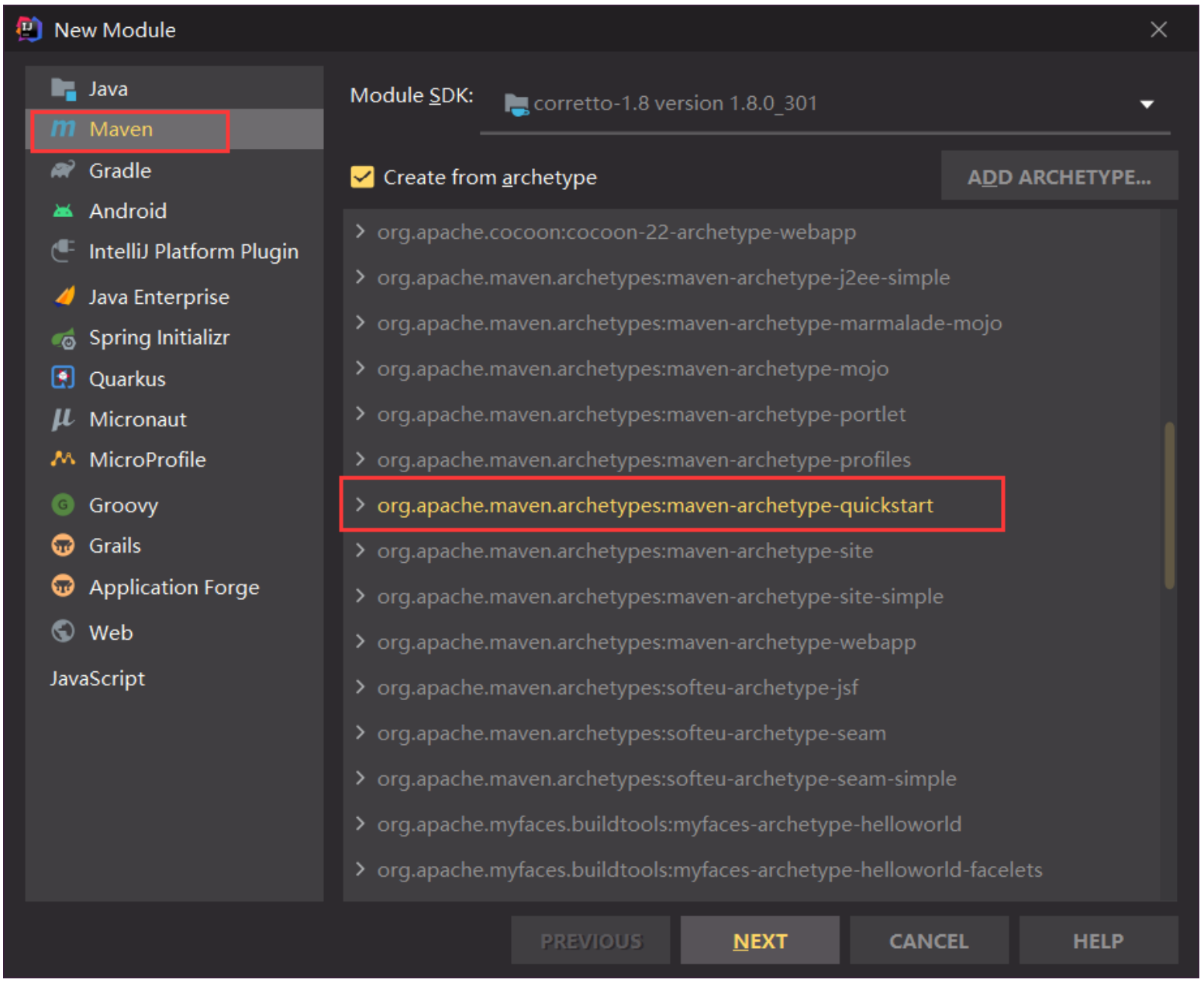
需要设置 maven 本机仓库:

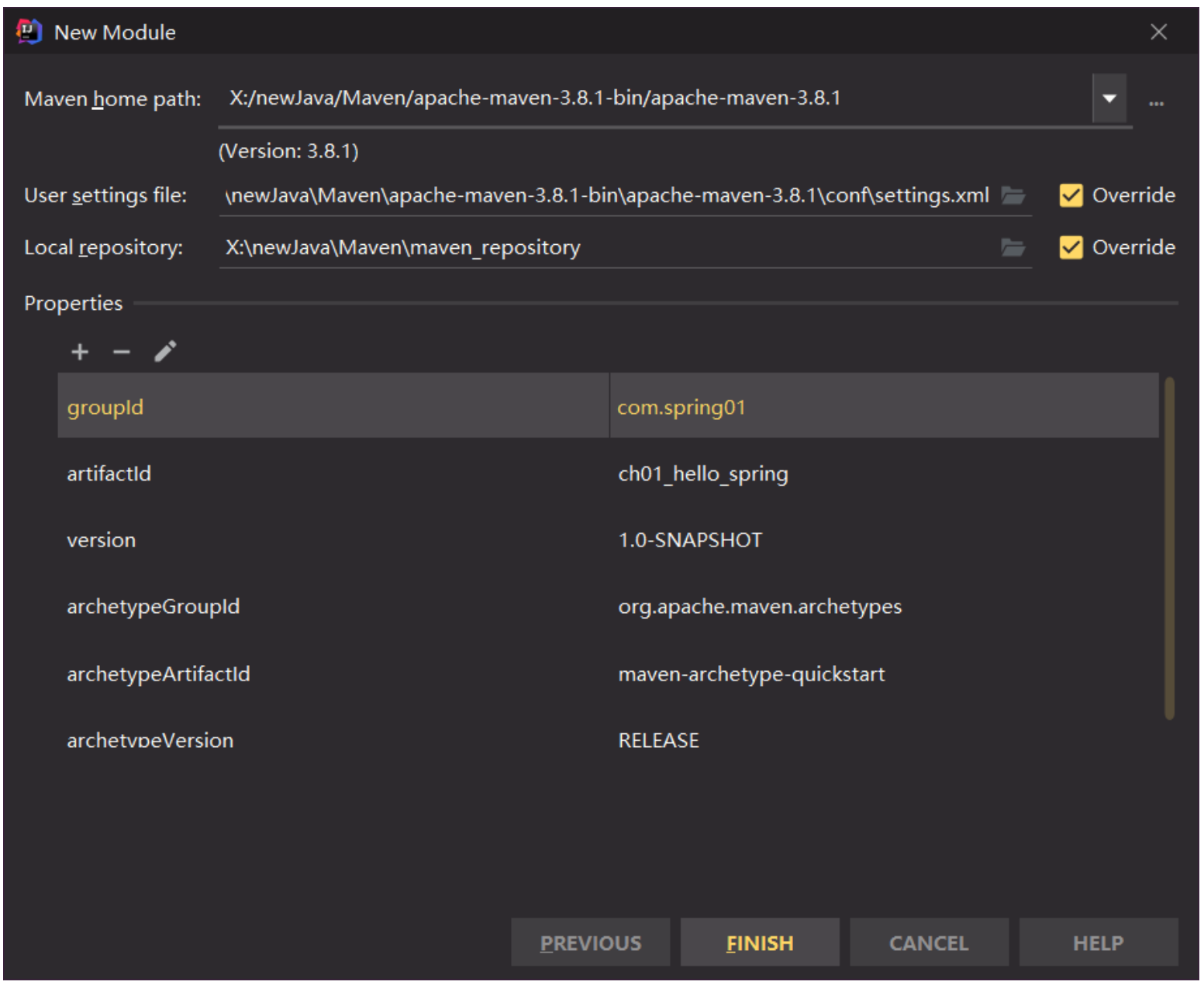


3 Spring 的第一个程序

```
实现步骤:
1.创建maven项目
2.加入maven的依赖
  spring的依赖，版本5.3.15版本
  junit依赖
3.创建类（接口和他的实现类）
  和没有使用框架一样，就是普通的类。
4.创建spring需要使用的配置文件
  声明类的信息，这些类由spring创建和管理
5.测试spring创建的。
```

1. 创建 maven 项目





2. 引入 maven 依赖 pom.xml

```xml
<dependencies>
    <!--单元测试-->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <!--Spring 依赖-->
    <dependency>
      <groupId>org.springframework</groupId>
```

```
      <artifactId>spring-context</artifactId>
      <version>5.2.5.RELEASE</version>
    </dependency>
</dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

3. 定义接口与实体类

```
//定义接口
package com.spring01.service;
public interface SomeService {
    void doSome();
}
```

```
//定义接口实现类
package com.spring01.service.impl;
import com.spring01.service.SomeService;
public class SomeServiceImpl implements SomeService {
    @Override
    public void doSome() {
        System.out.println("执行了SomeServiceImpl的doSome()方法");
    }
}
```

```
//测试方法 正转
@Test
public void test01(){
    SomeService service = new SomeServiceImpl();
    service.doSome();
    //执行了SomeServiceImpl的doSome()方法
}
```

4. 创建 Spring 配置文件

在 src/main/resources/ 目录现创建一个 xml 文件，文件名可以随意，但 Spring 建议的名称 applicationContext.xml。

spring 配置中需要加入约束文件才能正常使用，约束文件是 xsd 扩展名。

```
<?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">
    <!--
        spring的配置文件
        1.beans : 是根标签，spring把java对象成为bean。
        2.spring-beans.xsd 是约束文件，和 mybatis 指定 dtd是一样的。
    -->

    <!--
        告诉spring创建对象
        声明bean，就是告诉spring要创建某个类的对象
        id:对象的自定义名称，唯一值。spring 通过这个名称找到对象
        class:类的全限定名称（不能是接口，因为spring是反射机制创建对象，必须使用类）

        spring就完成 SomeService someService = new SomeServiceImpl();
        spring是把创建好的对象放入到 map中，spring框架有一个 map存放对象的。
           springMap.put(id的值，对象）；
           例如 springMap.put("someService", new SomeServiceImpl());

        注意：一个bean标签声明一个对象。
    -->
    <bean id="someService" class="com.spring01.service.impl.SomeServiceImpl" />
    <bean id="someService1" class="com.spring01.service.impl.SomeServiceImpl" scope="prototype"/>

    <!--
       spring能创建一个非自定义类的对象吗？创建一个存在的某个类的对象。
    -->
    <bean id="myDate" class="java.util.Date" />
</beans>
```

bean 标签：用于定义一个实例对象。一个实例对应一个 bean 元素。

id：该属性是 Bean 实例的唯一标识，程序通过 id 属性访问 Bean，Bean 与 Bean 间的依赖关系也是通过 id 属性关联的。

class：指定该 Bean 所属的类，注意这里只能是类，不能是接口。

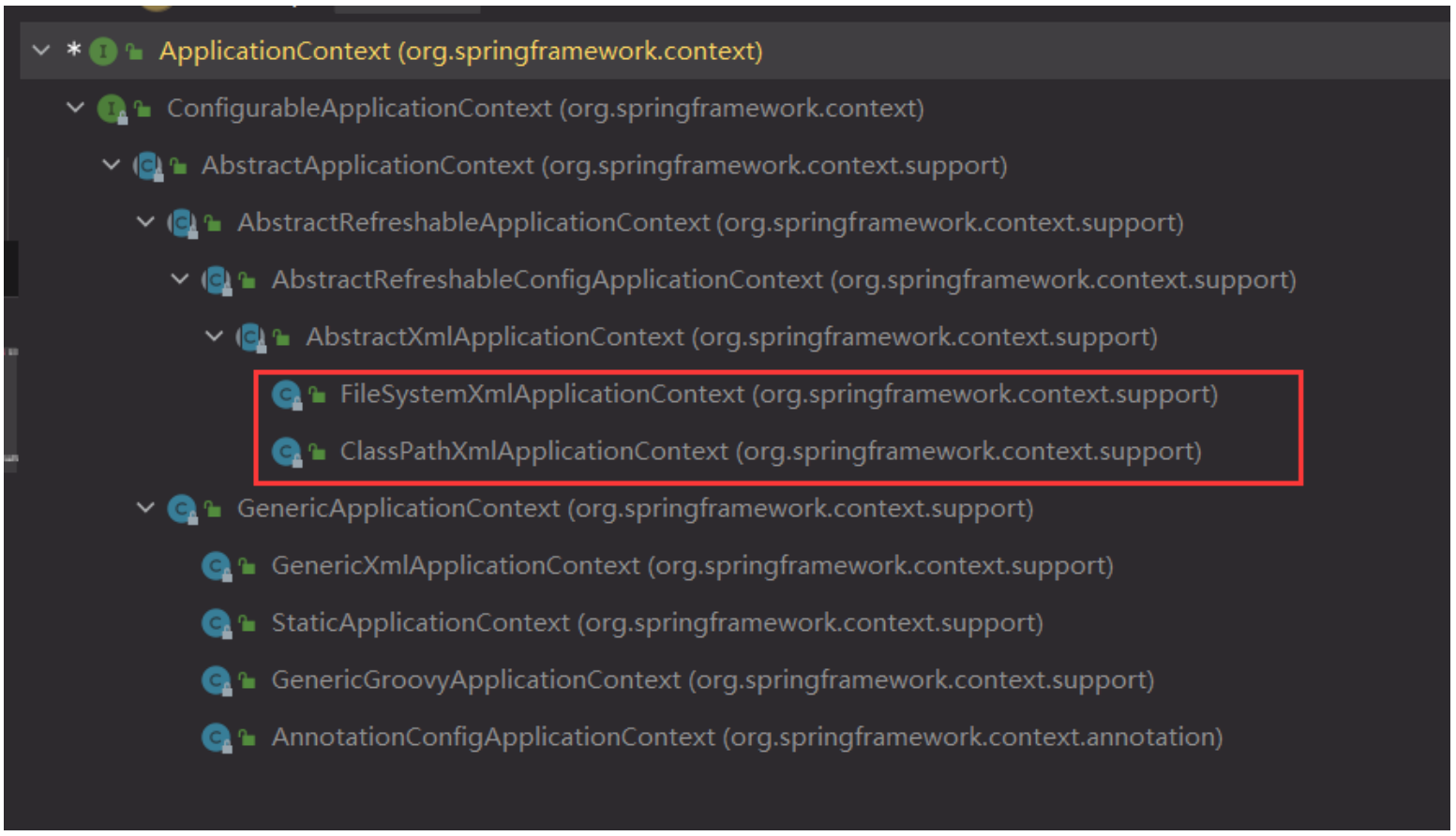
5. 编写测试方法

```java
/**
 * spring默认创建对象的时间：在创建spring的容器时，会创建配置文件中的所有的对象。
 * spring创建对象：默认调用的是无参数构造方法
 */
@Test
public void test02(){
    //使用 Spring 容器创建对象
    //1.指定Spring配置文件的名称
    String cpnfig = "applicationContext.xml";

    //2.创建表示Spring容器的对象，ApplicationContext
    //ApplicationContext 就是表示Spring容器，通过这个容器获取对象
    //ClassPathXmlApplicationContext: 表示从类路径中加载Spring的配置文件
    ApplicationContext ac = new ClassPathXmlApplicationContext(cpnfig);

    //3.从Spring中获取对象，使用 id
    SomeService someService = (SomeService) ac.getBean("someService");
    //4.使用对象的业务方法
    someService.doSome();
}
```

6. 容器接口和实现类

ApplicationContext 接口（容器）

ApplicationContext 用于加载 Spring 的配置文件，在程序中充当“容器”的角色。其实现类有两个。



- 配置文件在类路径下

  若 Spring 配置文件存放在项目的类路径下，则使用 ClassPathXmlApplicationContext 实现类进行加载。

```java
//指定Spring配置文件的名称
String cpnfig = "applicationContext.xml";
```

- ApplicationContext 容器中对象的装配时机

  ApplicationContext 容器，会在容器对象初始化时，将其中的所有对象一次性全部装配好。以后代码中若要使用到这些对象，只需从内存中直接获取即可。执行效率较高。但占用内存。

```java
//创建表示Spring容器的对象，ApplicationContext
//ApplicationContext 就是表示Spring容器，通过这个容器获取对象
//ClassPathXmlApplicationContext: 表示从类路径中加载Spring的配置文件
ApplicationContext ac = new ClassPathXmlApplicationContext(cpnfig);
//从Spring中获取对象，使用 id
SomeService someService = (SomeService) ac.getBean("someService");
```

7. 获取Spring容器中Java对象的信息

```xml
<bean id="someService" class="com.spring01.service.impl.SomeServiceImpl" />
<bean id="someService2" class="com.spring01.service.impl.SomeServiceImpl" />
<bean id="someService1" class="com.spring01.service.impl.SomeServiceImpl" scope="prototype"/>
<!--非自定义类-->
<bean id="myDate" class="java.util.Date" />
```

```java
public class SomeServiceImpl implements SomeService {
    public SomeServiceImpl(){
        System.out.println("SomeServiceImpl的无参数构造方法");
    }
    @Override
    public void doSome() {
        System.out.println("执行了SomeServiceImpl的doSome()方法");
    }
}
```

```java
    /**
     *  获取Spring容器中 java 对象的信息
     */
    @Test
    public void test03(){
        String cpnfig = "applicationContext.xml";
        ApplicationContext ac = new ClassPathXmlApplicationContext(cpnfig);
        //使用 Spring 提供的方法，获取容器中定义的对象的数量
        int nums = ac.getBeanDefinitionCount();
        System.out.println("容器中定义的对象数量: " + nums);
        //获取容器中定义的每个对象的名字
        String[] names = ac.getBeanDefinitionNames();
        for (String name : names) {
            System.out.println(name);
        }
    }
/*
SomeServiceImpl的无参数构造方法
SomeServiceImpl的无参数构造方法
容器中定义的对象数量: 4
someService
someService2
someService1
myDate
*/
```

8. junit 单元测试

```
junit : 单元测试，一个工具类库，做测试方法使用的。
单元：指定的是方法，一个类中有很多方法，一个方法称为单元。

  使用单元测试
   1.需要加入junit依赖。
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
   2.创建测试作用的类：叫做测试类
     src/test/java目录中创建类
   3.创建测试方法
      1）public 方法
      2）没有返回值 void
      3）方法名称自定义，建议名称是test + 你要测试方法名称
      4）方法没有参数
      5）方法的上面加入 @Test ,这样的方法是可以单独执行的。 不用使用main方法。
```

```
在spring的配置文件中，给java对象的属性赋值。

di：依赖注入，表示创建对象，给属性赋值。

di的实现有两种：
1.在spring的配置文件中，使用标签和属性完成，叫做基于XML的di实现
2.使用spring中的注解，完成属性赋值，叫做基于注解的id实现

di的语法分类：
 1. set注入（设置注入）：spring调用类的set方法，在set方法可以实现属性的赋值。
    80%左右都是使用的set注入
 2. 构造注入，spring调用类的有参数构造方法，创建对象。在构造方法中完成赋值。
```

```
实现步骤：
1.创建maven项目
2.加入maven的依赖
  spring的依赖，版本5.2.5版本
  junit依赖
3.创建类（接口和他的实现类）
  和没有使用框架一样， 就是普通的类。
4.创建spring需要使用的配置文件
  声明类的信息，这些类由spring创建和管理
  通过spring的语法，完成属性的赋值
5.测试spring创建的。
```

4 基于 XML(配置文件) 的 DI (依赖注入)

bean 实例在调用无参构造器创建对象后，就要对 bean 对象的属性进行初始化。初始化是由容器自动完成的，称为注入。

根据注入方式的不同，常用的有两类：set 注入、构造注入。

4.1 set 注入

set 注入也叫设值注入，是指通过 set 方法传入被调用者的实例。这种注入方式简单、 直观，因而在 Spring 的依赖注入中大量使用。

- 简单类型

```java
//实体类
package com.spring01.bao1;
public class Student {
    private String name;
    private int age;
    // set get toString
    //注意：必须有 set 方法，没有 set 方法是报错的。
    public Student() {
        System.out.println("spring会调用类的无参数构造方法创建对象");
    }
    public void setEmail(String email){
        System.out.println("setEmail="+email);
    }
   public void setName(String name) {
        System.out.println("setName:"+name);
        this.name = name.toUpperCase();
    }
    public void setAge(int age) {
        System.out.println("setAge:"+age);
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

```xml
<!--src/main/resources/bao1/applicationContext.xml-->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--
        声明student对象
        注入：就是赋值的意思
        简单类型： spring中规定java的基本数据类型和String都是简单类型。
        di:给属性赋值

        set注入（设值注入） ：spring调用类的set方法， 你可以在set方法中完成属性赋值
            简单类型的set注入
            <bean id="xx" class="yyy">
               <property name="属性名字" value="此属性的值"/>
               一个property只能给一个属性赋值
               <property....>
            </bean>
    -->
    <bean id="myStudent" class="com.spring01.bao1.Student">
```

```
        <property name="name" value="李四lisi" /><!--setName("李四")-->
        <property name="age" value="22" /><!--setAge(21)-->
        <!--set注入只是使用set方法-->
        <property name="email" value="lisi@qq.com" /><!--setEmail("lisi@qq.com")-->
    </bean>
</beans>
```

```
    //测试方法
    @Test
    public void test01(){
        String cpnfig = "bao1/applicationContext.xml";
        ApplicationContext ac = new ClassPathXmlApplicationContext(cpnfig);
        Student student = (Student) ac.getBean("myStudent");
        System.out.println("student对象=" + student);
    }
/*
spring会调用类的无参数构造方法创建对象
setEmail=lisi@qq.com
student对象=Student{name='李四lisi', age=22}

先调用构造方法，再调用set方法
*/
```

- 引用类型

```
package com.spring01.bao2;
public class School {
    private String name;
    private String address;
    public void setName(String name) {
        this.name = name;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    @Override
    public String toString() {
        return "School{" +
                "name='" + name + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}
```

```
package com.spring01.bao2;

public class Student {
    private String name;
    private int age;
    //声明一个引用类型
    private School school;

    public Student(){
        System.out.println("spring会调用类的无参数构造方法创建对象");
    }
    public void setName(String name) {
        System.out.println("setName:" + name);
        this.name = name;
    }
    public void setAge(int age) {
        System.out.println("setAge:" + age);
        this.age = age;
    }
    public void setSchool(School school) {
        System.out.println("setSchool:" + school);
        this.school = school;
    }
    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", school=" + school +
                '}';
    }
}
```

```
<!--src/main/resources/bao1/applicationContext.xml-->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--
       引用类型的set注入：spring调用类的set方法
        <bean id="xxx" class="yyy">
           <property name="属性名称" ref="bean的id(对象的名称)" />
        </bean>
    -->
    <bean id="myStudent" class="com.spring01.bao2.Student" >
        <property name="name" value="李四lisi" /><!--setName("李四")-->
        <property name="age" value="22" /><!--setAge(21)-->
        <!--引用类型-->
        <property name="school" ref="mySchool" />
    </bean>
    <!--声明School对象-->
    <bean id="mySchool" class="com.spring01.bao2.School">
        <property name="name" value="北京大学"/>
        <property name="address" value="北京的海淀区" />
    </bean>
</beans>
```

```
public class Test02 {
    @Test
    public void test01(){
        String cpnfig = "bao2/applicationContext.xml";
        ApplicationContext ac = new ClassPathXmlApplicationContext(cpnfig);
        Student student = (Student) ac.getBean("myStudent");
        System.out.println("student对象=" + student);
    }
}
/*
spring会调用类的无参数构造方法创建对象
setName:李四lisi
setAge:22
setSchool:School{name='北京大学', address='北京的海淀区'}
student对象=Student{name='李四lisi', age=22, school=School{name='北京大学', address='北京的海淀区'}}
*/
```

4.2 构造注入

构造注入是指，在构造调用者实例的同时，完成被调用者的实例化。即使用构造器设置依赖关系。

```
package com.spring01.bao3;

public class Student {
    private String name;
    private int age;
    //声明一个引用类型
    private School school;

    public Student(){
        System.out.println("spring会调用类的无参数构造方法创建对象");
    }
    //创建有参构造方法
    public Student(String name, int age, School school) {
        this.name = name;
        this.age = age;
        this.school = school;
        System.out.println("====Student类有参构造方法====");
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setSchool(School school) {
        this.school = school;
    }
    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", school=" + school +
                '}';
    }
}
```

```
package com.spring01.bao3;
public class School {
    private String name;
```

```
    private String address;
    public void setName(String name) {
        this.name = name;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    @Override
    public String toString() {
        return "School{" +
                "name='" + name + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}
```

```
<?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">
    <!--
          构造注入：spring调用类有参数构造方法，在创建对象的同时，在构造方法中给属性赋值。
          构造注入使用 <constructor-arg> 标签
          <constructor-arg> 标签：一个<constructor-arg>表示构造方法一个参数。
          <constructor-arg> 标签属性：
             name:表示构造方法的形参名
             index:表示构造方法的参数的位置，参数从左往右位置是 0 ， 1 ，2的顺序
             value：构造方法的形参类型是简单类型的，使用value
             ref：构造方法的形参类型是引用类型的，使用ref
    -->
    <!--使用name属性实现构造注入-->
    <bean id="myStudent" class="com.spring01.bao3.Student" >
        <constructor-arg name="name" value="张三" />
        <constructor-arg name="age" value="20" />
        <constructor-arg name="school" ref="mySchool" />
    </bean>

    <!--使用 index 属性实现构造注入-->
    <bean id="myStudent1" class="com.spring01.bao3.Student" >
        <constructor-arg index="1" value="18" />
        <constructor-arg index="0" value="李四" />
        <constructor-arg index="2" ref="mySchool" />
    </bean>

    <!--省略index-->
    <bean id="myStudent2" class="com.spring01.bao3.Student">
        <constructor-arg  value="张强强" />
        <constructor-arg  value="22" />
        <constructor-arg  ref="mySchool" />
    </bean>

    <!--声明School对象-->
    <bean id="mySchool" class="com.spring01.bao3.School">
        <property name="name" value="北京大学"/>
        <property name="address" value="北京的海淀区" />
    </bean>

    <!--创建File,使用构造注入-->
    <bean id="myFile" class="java.io.File">
        <constructor-arg name="parent" value="X:\Markdown笔记\Java生态" />
        <constructor-arg name="child" value="学习路线.md" />
    </bean>
</beans>
```

```
public class Test03 {
    @Test
    public void test02(){
        String cpnfig = "bao3/applicationContext.xml";
        ApplicationContext ac = new ClassPathXmlApplicationContext(cpnfig);
        Student student = (Student) ac.getBean("myStudent");
        System.out.println("student对象=" + student);

        Student student1 = (Student) ac.getBean("myStudent1");
        System.out.println("student1对象=" + student1);

        Student student2 = (Student) ac.getBean("myStudent2");
        System.out.println("student2对象=" + student2);

        File myFile = (File) ac.getBean("myFile");
        System.out.println("myFile: " + myFile.getName());
    }
```

```
}
/*
====Student类有参构造方法====
====Student类有参构造方法====
====Student类有参构造方法====
student对象=Student{name='张三', age=20, school=School{name='北京大学', address='北京的海淀区'}}
student1对象=Student{name='李四', age=18, school=School{name='北京大学', address='北京的海淀区'}}
student2对象=Student{name='张强强', age=22, school=School{name='北京大学', address='北京的海淀区'}}
myFile: 学习路线.md
*/
```

4.3 引用类型属性自动注入

对于引用类型属性的注入，也可不在配置文件中显示的注入。可以通过为标签设置 autowire 属性值，为引用类型属性进行隐式自动注入(默认是不自动注入引用类型属 性）。根据自动注入判断标准的不同，可以分为两种：

- byName：根据名称自动注入
- byType： 根据类型自动注入

1. byName 方式自动注入

   当配置文件中被调用者 bean 的 id 值与代码中调用者 bean 类的属性名相同时，可使用 byName 方式，让容器自动将被调用者 bean 注入给调用者 bean。容器是通过调用者的 bean 类的属性名与配置文件的被调用者 bean 的 id 进行比较而实现自动注入的。

```java
public class School {
    private String name;
    private String address;
    public void setName(String name) {
        this.name = name;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    @Override
    public String toString() {
        return "School{" +
                "name='" + name + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}
```

```java
public class Student {
    private String name;
    private int age;
    //声明一个引用类型
    private School school;
    public Student(){
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setSchool(School school) {
        System.out.println("setSchool:" + school);
        this.school = school;
    }
    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", school=" + school +
                '}';
    }
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--
        引用类型的自动注入：spring框架根据某些规则可以给引用类型赋值。不用你在给引用类型赋值了
       使用的规则常用的是byName，byType
       byName(按名称注入)：java类中引用类型的属性名和spring容器中（配置文件）<bean>的id名称一样，
                           且数据类型是一致的，这样的容器中的bean，spring能够赋值给引用类型。
```

```
          语法：
          <bean id="xx" class="yyy" autowire="byName">
             简单类型属性赋值
          </bean>
     -->
     <!--byName-->
     <bean id="myStudent" class="com.spring01.bao4.Student" autowire="byName" >
         <property name="name" value="李四lisi" />
         <property name="age" value="22" />
     </bean>
     <!--声明School对象-->
     <bean id="school" class="com.spring01.bao4.School">
         <property name="name" value="北京大学"/>
         <property name="address" value="北京的海淀区" />
     </bean>
</beans>
```

```
public class Test04 {
    @Test
    public void test02(){
        String cpnfig = "bao4/applicationContext.xml";
        ApplicationContext ac = new ClassPathXmlApplicationContext(cpnfig);
        Student student = (Student) ac.getBean("myStudent");
        System.out.println("student对象=" + student);
    }
}
```

2. byType 方式自动注入

使用 byType 方式自动注入，要求：配置文件中被调用者 bean 的 class 属性指定的类，要与代码中调用者 bean 类的某引用类型属性类型同源。即要么相同，要么有 is-a 关系（子类或是实现类）。但这样的同源的被调用 bean 只能有一个。多于一个，容器就不知该匹配哪一个了。

```
//子类
public class PrimarySchool extends School {
}
```

```
<?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">
    <!--
         引用类型的自动注入：spring框架根据某些规则可以给引用类型赋值。不用你在给引用类型赋值了
        使用的规则常用的是byName, byType

        byType(按类型注入)：java类中引用类型的数据类型和spring容器中（配置文件）<bean>的class属性
                            是同源关系的，这样的bean能够赋值给引用类型
          同源就是一类的意思：
           1.java类中引用类型的数据类型和bean的class的值是一样的。
           2.java类中引用类型的数据类型和bean的class的值父子类关系的。
           3.java类中引用类型的数据类型和bean的class的值接口和实现类关系的
          语法：
          <bean id="xx" class="yyy" autowire="byType">
             简单类型属性赋值
          </bean>

          注意：在byType中， 在xml配置文件中声明bean只能有一个符合条件的，
                多余一个是错误的
    -->
    <!--byType-->
    <bean id="myStudent" class="com.spring01.bao5.Student"  autowire="byType">
        <property name="name" value="张飒" />
        <property name="age" value="26" />
        <!--引用类型-->
        <!--<property name="school" ref="mySchool" />-->
    </bean>

    <!--声明School对象-->
    <bean id="mySchool" class="com.spring01.bao5.School">
        <property name="name" value="人民大学"/>
        <property name="address" value="北京的海淀区" />
    </bean>

    <!--声明School的子类-->
    <!--<bean id="primarySchool" class="com.spring01.bao5.PrimarySchool">
        <property name="name" value="北京小学" />
        <property name="address" value="北京的大兴区" />
    </bean>-->
</beans>
```

```java
public class Test05 {
    @Test
    public void test05(){
        String cpnfig = "bao5/applicationContext.xml";
        ApplicationContext ac = new ClassPathXmlApplicationContext(cpnfig);
        Student student = (Student) ac.getBean("myStudent");
        System.out.println("student对象=" + student);
    }
}
/*
setSchool:School{name='北京小学', address='北京的大兴区'}
student对象=Student{name='张飒', age=26, school=School{name='北京小学', address='北京的大兴区'}}
*/
```

4.4 为应用指定多个 Spring 配置文件

在实际应用里，随着应用规模的增加，系统中 Bean 数量也大量增加，导致配置文件变得非常庞大、臃肿。为了避免这种情况的产生，提高配置文件的可读性与可维护性，可以将 Spring 配置文件分解成多个配置文件。

```
多个配置优势
  1.每个文件的大小比一个文件要小很多。效率高
  2.避免多人竞争带来的冲突。

  如果你的项目有多个模块（相关的功能在一起） ，一个模块一个配置文件。
  学生考勤模块一个配置文件，    张三
  学生成绩一个配置文件，        李四

多文件的分配方式：
  1. 按功能模块，一个模块一个配置文件
  2. 按类的功能，数据库相关的配置一个文件配置文件，做事务的功能一个配置文件，做service功能的一个配置文件等
```

```java
public class School {
    private String name;
    private String address;
    //set toString
}
```

```java
public class Student {
    private String name;
    private int age;
    //声明一个引用类型
    private School school;
    public Student(){
        System.out.println("spring会调用类的无参数构造方法创建对象");
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setSchool(School school) {
        System.out.println("setSchool:" + school);
        this.school = school;
    }
    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", school=" + school +
                '}';
    }
}
```

```xml
<!--spring-student.xml-->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--student模块所有bean的声明-->
    <!--byType-->
    <bean id="myStudent" class="com.spring01.bao6.Student"  autowire="byType">
        <property name="name" value="张飒" />
        <property name="age" value="30" />
    </bean>
</beans>
```

```xml
<!--spring-school.xml-->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--School模块所有bean的声明， School模块的配置文件-->
    <!--声明School对象-->
    <bean id="mySchool" class="com.spring01.bao6.School">
        <property name="name" value="航空大学"/>
        <property name="address" value="北京的海淀区" />
    </bean>
</beans>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--
         包含关系的配置文件：
         spring-total表示主配置文件 ：  包含其他的配置文件的，主配置文件一般是不定义对象的。
         语法：<import resource="其他配置文件的路径" />
         关键字："classpath:" 表示类路径（class文件所在的目录），
               在spring的配置文件中要指定其他文件的位置， 需要使用classpath，告诉spring到哪去加载读取文件。
    -->

    <!--加载的是文件列表-->
    <!--
    <import resource="classpath:ba06/spring-school.xml" />
    <import resource="classpath:ba06/spring-student.xml" />
    -->

    <!--
       在包含关系的配置文件中，可以通配符（*：表示任意字符）
       注意： 主的配置文件名称不能包含在通配符的范围内（不能叫做spring-total.xml）
    -->
    <import resource="classpath:bao6/spring-*.xml" />
</beans>
```

```java
public class TestO6 {
    @Test
    public void testO6(){
        String cpnfig = "bao6/total.xml";
        ApplicationContext ac = new ClassPathXmlApplicationContext(cpnfig);
        Student student = (Student) ac.getBean("myStudent");
        System.out.println("student对象=" + student);
    }
}
```

5 基于注解的 DI (依赖注入)

5.1 注解概述

使用注解的步骤：

1.加入maven的依赖 spring-context，在你加入spring-context 的同时，间接加入spring-aop的依赖。使用注解必须使用spring-aop依赖；

2.在类中加入spring的注解（多个不同功能的注解）

3.在spring的配置文件中，加入一个组件扫描器的标签，说明注解在你的项目中的位置

4.使用注解创建对象，创建容器ApplicationContext

```
学习的注解：
     @Component
     @Respotory
     @Service
     @Controller
     @value
     @Autowired
     @Resource
```

5.2 定义 Bean 的注解 @Component

```java
package com.spring01.bao1;
import org.springframework.stereotype.Component;
/**
 * @Component: 创建对象的，等同于<bean>的功能
 *      属性：value 就是对象的名称，也就是bean的id值，
 *           value的值是唯一的，创建的对象在整个spring容器中就一个
 *      位置：在类的上面
 *
 *  @Component(value = "myStudent")等同于
 *   <bean id="myStudent" class="com.spring01.bao1.Student" />
 *
 *  spring中和@Component功能一致，创建对象的注解还有：
 *  1.@Repository（用在持久层类的上面） ： 放在dao的实现类上面，
 *               表示创建dao对象，dao对象是能访问数据库的。
 *  2.@Service(用在业务层类的上面)：放在service的实现类上面，
 *              创建service对象，service对象是做业务处理，可以有事务等功能的。
 *  3.@Controller(用在控制器的上面)：放在控制器（处理器）类的上面，创建控制器对象的，
 *              控制器对象，能够接受用户提交的参数，显示请求的处理结果。
 *  以上三个注解的使用语法和@Component一样的。 都能创建对象，但是这三个注解还有额外的功能。
 *  @Repository，@Service，@Controller是给项目的对象分层的。
 */

//使用value属性，指定对象名称
//@Component(value = "myStudent")

//省略value
@Component("myStudent")

//不指定对象名称，由spring提供默认名称: 类名的首字母小写: student
//@Component
public class Student {
    String name;
    Integer age;

    public Student(){
        System.out.println("Student的无参构造方法");
    }

    public void setName(String name) {
        this.name = name;
    }

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

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

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

    <!--
        声明组件扫描器(component-scan),组件就是java对象
        base-package：指定注解在你的项目中的包名。
        component-scan工作方式： spring会扫描遍历base-package指定的包，
            把包中和子包中的所有类，找到类中的注解，按照注解的功能创建对象，或给属性赋值。

        加入了component-scan标签，配置文件的变化：
        1.加入一个新的约束文件spring-context.xsd
        2.给这个新的约束文件起个命名空间的名称
    -->
    <context:component-scan base-package="com.spring01.bao1" />

    <!--指定多个包的三种方式-->
    <!--第一种方式：使用多次组件扫描器，指定不同的包-->
    <context:component-scan base-package="com.spring01.bao1"/>
```

```
    <context:component-scan base-package="com.spring01.bao2"/>

    <!--第二种方式：使用分隔符（;或,）分隔多个包名-->
    <context:component-scan base-package="com.spring01.bao1;com.spring01.bao2" />

    <!--第三种方式：指定父包-->
    <context:component-scan base-package="com.spring01" />
</beans>
```

```
public class MyTest01 {
    @Test
    public void test01(){
        String config = "applicatuionContext.xml";
        ApplicationContext context = new ClassPathXmlApplicationContext(config);
        //从容器中获取对象
        Student student = (Student) context.getBean("myStudent");
        System.out.println("student=" + student);
    }
}
/*
Student的无参构造方法
student=Student{name='null', age=null}
*/
```

5.3 简单类型属性注入@Value

需要在属性上使用注解@Value，该注解的 value 属性用于指定要注入的值。

使用该注解完成属性注入时，类中无需 setter。当然，若属性有 setter，则也可将其加到 setter 上。

```
package com.spring01.bao2;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("myStudent")
public class Student {
    /**
     * @Value: 简单类型的属性赋值
     *    属性：  value 是 String 类型的，表示简单类型的属性值
     *    位置：  1.在属性定义的上面，无需set方法，推荐使用。
     *           2.在set方法的上面
     */
    @Value("张三")
    private String name;
    @Value("16")
    private Integer age;

    public Student(){
        System.out.println("Student的无参构造方法");
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

```
public class MyTest02 {
    @Test
    public void test01(){
        String config = "applicationContext.xml";
        ApplicationContext context = new ClassPathXmlApplicationContext(config);
        //从容器中获取对象
        Student student = (Student) context.getBean("myStudent");
        System.out.println("student=" + student);
    }
}
/*
Student的无参构造方法
student=Student{name='张三', age=16}
*/
```

5.4 byType 自动注入@Autowired

```java
package com.spring01.bao3;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("myStudent")
public class Student {
    @Value("李四")
    private String name;
    @Value("20")
    private int age;

    /**
     * 引用类型
     * @Autowired: spring框架提供的注解，实现引用类型的赋值。
     * spring中通过注解给引用类型赋值，使用的是自动注入原理 ，支持byName, byType
     *
     * @Autowired:默认使用的是byType自动注入。
     *  位置：1）在属性定义的上面，无需set方法， 推荐使用
     *       2）在set方法的上面
     */
    //声明一个引用类型
    @Autowired
    private School school;

    public Student(){
        System.out.println("spring会调用类的无参数构造方法创建对象");
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setSchool(School school) {
        this.school = school;
    }
    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", school=" + school +
                '}';
    }
}
```

```java
package com.spring01.bao3;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("mySchool")
public class School {
    @Value("北京大学")
    private String name;
    @Value("北京的海淀区")
    private String address;
    public void setName(String name) {
        this.name = name;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    @Override
    public String toString() {
        return "School{" +
                "name='" + name + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}
```

```java
public class MyTest03 {
    @Test
    public void test01(){
        String config = "applicationContext.xml";
        ApplicationContext context = new ClassPathXmlApplicationContext(config);
        //从容器中获取对象
```

```
        Student student = (Student) context.getBean("myStudent");
        System.out.println("student=" + student);
    }
}
/*
spring会调用类的无参数构造方法创建对象
student=Student{name='李四', age=20, school=School{name='北京大学', address='北京的海淀区'}}
*/
```

5.5 byName 自动注入@Autowired 与@Qualifier

```
package com.spring01.bao4;

@Component("myStudent")
public class Student {
    @Value("李四")
    private String name;
    @Value("20")
    private int age;

    /**
     * 引用类型
     * @Autowired: spring框架提供的注解，实现引用类型的赋值。
     * spring中通过注解给引用类型赋值，使用的是自动注入原理 ，支持byName, byType
     *
     *      @Autowired属性：required ，是一个boolean类型的，默认true
     *              required=true：表示引用类型赋值失败，程序报错，并终止执行。(推荐使用)
     *              required=false：引用类型如果赋值失败，程序正常执行，引用类型是null
     *
     * @Autowired:默认使用的是byType自动注入。
     *  位置：
     *       1）在属性定义的上面，无需set方法， 推荐使用
     *       2）在set方法的上面
     *
     * 如果要使用byName方式，需要做的是：
     *    1.在属性上面加入@Autowired
     *    2.在属性上面加入@Qualifier(value="bean的id") ：表示使用指定名称的bean完成赋值。
     */
    //byName 自动注入
    @Autowired(required = false)
    @Qualifier(value = "mySchool")
    private School school;

    public Student(){
        System.out.println("spring会调用类的无参数构造方法创建对象");
    }
    //创建有参构造方法
    public Student(String name, int age, School school) {
        this.name = name;
        this.age = age;
        this.school = school;
        System.out.println("====Student类有参构造方法====");
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setSchool(School school) {
        this.school = school;
    }
    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", school=" + school +
                '}';
    }
}
```

```
package com.spring01.bao4;

@Component("mySchool")
public class School {
    @Value("北京大学")
    private String name;
    @Value("北京的海淀区")
    private String address;
```

```
    public void setName(String name) {
        this.name = name;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    @Override
    public String toString() {
        return "School{" +
                "name='" + name + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}

```

5.6 JDK 注解@Resource 自动注入

```
package com.spring01.bao6;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import javax.annotation.Resource;
@Component("myStudent")
public class Student {
    @Value("李四")
    private String name;
    @Value("20")
    private int age;

    /**
     * 引用类型
     * @Resource: 来自jdk中的注解，spring框架提供了对这个注解的功能支持，可以使用它给引用类型赋值
     *              使用的也是自动注入原理，支持byName,byType; 默认是byName
     *       位置：  1.在属性定义的上面，无需set方法，推荐使用。
     *             2.在set方法的上面
     */
    //默认是byName：先使用byName自动注入，如果byName赋值失败，再使用byType
    @Resource
    private School school;

    public Student(){System.out.println("spring会调用类的无参数构造方法创建对象");}
    //创建有参构造方法
    public void setName(String name) {this.name = name;}
    public void setAge(int age) {this.age = age;}
    public void setSchool(School school) {this.school = school;}
    @Override
    public String toString() {
        return "Student{" +"name='" + name + '\'' +", age=" + age +", school=" + school +'}';
    }
}
```

```
package com.spring01.bao6;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
@Component("mySchool")
public class School {
    @Value("北京大学")
    private String name;
    @Value("北京的海淀区")
    private String address;

    public void setName(String name) {this.name = name;}
    public void setAddress(String address) {this.address = address;}
    @Override
    public String toString() {
        return "School{" +"name='" + name + '\'' + ", address='" + address + '\'' + '}';
    }
}
```

```
public class MyTest06 {
    @Test
    public void test01(){
        String config = "applicationContext.xml";
        ApplicationContext context = new ClassPathXmlApplicationContext(config);
        //从容器中获取对象
        Student student = (Student) context.getBean("myStudent");
        System.out.println("student=" + student);
    }
}
```

```
/*
spring会调用类的无参数构造方法创建对象
student=Student{name='李四', age=20, school=School{name='北京大学', address='北京的海淀区'}}
*/
```

```
//只用byName
package com.spring01.bao7;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
import javax.annotation.Resource;

@Component("myStudent")
public class Student {
    @Value("李四")
    private String name;
    @Value("20")
    private int age;

    /**
     * 引用类型
     * @Resource: 来自jdk中的注解，spring框架提供了对这个注解的功能支持，可以使用它给引用类型赋值
     *            使用的也是自动注入原理，支持byName， byType .默认是byName
     *  位置： 1.在属性定义的上面，无需set方法，推荐使用。
     *        2.在set方法的上面
     *
     * @Resource 只使用byName方式，需要增加一个属性 name
     * name的值是bean的id（名称）
     */
    //只使用byName
    @Resource(name = "mySchool")
    private School school;

    public Student(){System.out.println("spring会调用类的无参数构造方法创建对象");}
    //创建有参构造方法
    public void setName(String name) {this.name = name;}
    public void setAge(int age) {this.age = age;}
    public void setSchool(School school) {this.school = school;}
    @Override
    public String toString() {
        return "Student{" +"name='" + name + '\'' +", age=" + age +", school=" + school +'}';
    }
}
```

6 注解与 XML 的对比

注解的优点：

- 方便
- 直观
- 高效（代码少，没有配置文件的书写那么复杂）
- 其弊端也显而易见：以硬编码的方式写入到 Java 代码中，修改是需要重新编译代码的。

XML方式的优点：

- 配置和代码是分离的
- 在 xml 中做修改，无需编译代码，只需重启服务器即可将新的配置加载
- xml 的缺点是：编写麻烦，效率低，大型项目过于复杂

三、AOP 面向切面编程

1 不使用 AOP 的开发方式

```
package com.spring01.service;
public interface SomeService {
    void doSome();
    void doOther();
}
```

```
package com.spring01.service.impl;
import com.spring01.service.SomeService;
import java.util.Date;
public class SomeServiceImpl implements SomeService {
    @Override
    public void doSome() {
        System.out.println("方法执行时间：" + new Date());
        System.out.println("执行业务方法doSome");
        System.out.println("方法执行结束，提交事务");
```

```
    }

    @Override
    public void doOther() {
        System.out.println("方法执行时间：" + new Date());
        System.out.println("执行业务方法doOther");
        System.out.println("方法执行结束，提交事务");
    }
}
```

```
package com.spring01;
import com.spring01.service.SomeService;
import com.spring01.service.impl.SomeServiceImpl;
public class MyApp {
    public static void main(String[] args) {
        //调用doSome, doOther
        SomeService service = new SomeServiceImpl();
        service.doSome();
        System.out.println("================================");
        service.doOther();
    }
}
/*
方法执行时间：Sat Feb 19 15:01:44 CST 2022
执行业务方法doSome
方法执行结束，提交事务
================================
方法执行时间：Sat Feb 19 15:01:44 CST 2022
执行业务方法doOther
方法执行结束，提交事务
*/
```

```
动态代理：可以在程序的执行过程中，创建代理对象。
通过代理对象执行方法，给目标类的方法增加额外的功能（功能增强）

jdk动态代理实现步骤：
1.创建目标类，SomeServiceImpl目标类，给它的doSome, doOther增加 输出时间，事务。
2.创建InvocationHandler接口的实现类，在这个类实现给目标方法增加功能。
3.使用jdk中类Proxy，创建代理对象。实现创建对象的能力。
```

```
//工具类
public class ServiceTools {
    public static void doLog(){ System.out.println("方法执行时间：" + new Date()); }
    public static void doTrans(){ System.out.println("方法执行结束，提交事务"); }
}
```

```
package com.spring01.service.impl;
import com.spring01.service.SomeService;
public class SomeServiceImpl implements SomeService {
    @Override
    public void doSome() { System.out.println("执行业务方法doSome"); }
    @Override
    public void doOther() { System.out.println("执行业务方法doOther"); }
}
```

```
package com.spring01.handler;
import com.spring01.util.ServiceTools;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class MyIncationHandler implements InvocationHandler {
    //目标对象
    private Object target; //SomeServiceImpl类
    public MyIncationHandler(Object target) {
        this.target = target;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        //通过代理对象执行方法时，会调用执行这个invoke（）
        System.out.println("执行MyIncationHandler中的invoke()");
        System.out.println("method名称："+method.getName());
        String methodName = method.getName();
        Object res = null;

        if("doSome".equals(methodName)){ //JoinPoint Pointcut
            ServiceTools.doLog(); //在目标方法之前，输出时间
            //执行目标类的方法，通过Method类实现
            res  = method.invoke(target, args); //SomeServiceImpl.doSome()
```

```
            ServiceTools.doTrans(); //在目标方法执行之后，提交事务
        } else {
            res  = method.invoke(target, args); //SomeServiceImpl.doOther()
        }

        //目标方法的执行结果
        return res;
    }
}
```

```
public class MyApp {
    public static void main(String[] args) {
        //使用jdk的Proxy创建代理对象
        //创建目标对象
        SomeService target = new SomeServiceImpl();
        //创建InvocationHandler对象
        InvocationHandler handler = new MyIncationHandler(target);
        //使用Proxy创建代理
        SomeService proxy = (SomeService) Proxy.newProxyInstance(
                target.getClass().getClassLoader(),
                target.getClass().getInterfaces(), handler);
        System.out.println("proxy======"+proxy.getClass().getName());//com.sun.proxy.$Proxy0
        //通过代理执行方法，会调用handler中的invoke（）
        proxy.doSome();
        System.out.println("================================================");
        proxy.doOther();
    }
}
```

2 AOP 概述

- AOP（Aspect Orient Programming），面向切面编程。面向切面编程是从动态角度考虑程序运行过程。
- AOP 底层，就是采用动态代理模式实现的。采用了两种代理：JDK 的动态代理与 CGLIB 的动态代理。
- AOP 为 Aspect Oriented Programming 的缩写，意为：面向切面编程，可通过运行期动态代理实现程序功能的统一维护的一种技术。AOP 是 Spring 框架中的一个重要内容。利用 AOP 可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。
- 面向切面编程，就是将交叉业务逻辑封装成切面，利用 AOP 容器的功能将切面织入到主业务逻辑中。所谓交叉业务逻辑是指，通用的、与主业务逻辑无关的代码，如安全检查、 事务、日志、缓存等。
- 若不使用 AOP，则会出现代码纠缠，即交叉业务逻辑与主业务逻辑混合在一起。这样， 会使主业务逻辑变的混杂不清。
- 例如，转账，在真正转账业务逻辑前后，需要权限控制、日志记录、加载事务、结束事务等交叉业务逻辑，而这些业务逻辑与主业务逻辑间并无直接关系。但，它们的代码量所占比重能达到总代码量的一半甚至还多。它们的存在，不仅产生了大量的“冗余”代码，还大大干扰了主业务逻辑 --- 转账。

```
1.动态代理
  实现方式：jdk动态代理，使用jdk中的Proxy，Method，InvocaitonHanderl创建代理对象。
            jdk动态代理要求目标类必须实现接口
  cglib动态代理：第三方的工具库，创建代理对象，原理是继承。 通过继承目标类，创建子类。
               子类就是代理对象。 要求目标类不能是final的， 方法也不能是final的

2.动态代理的作用：
    1）在目标类源代码不改变的情况下，增加功能。
    2）减少代码的重复
    3）专注业务逻辑代码
    4）解耦合，让你的业务功能和日志，事务非业务功能分离。

3.Aop:面向切面编程，基于动态代理的，可以使用jdk，cglib两种代理方式。
  Aop就是动态代理的规范化，把动态代理的实现步骤，方式都定义好了，
  让开发人员用一种统一的方式，使用动态代理。

4.AOP（Aspect Orient Programming）面向切面编程
  Aspect: 切面，给你的目标类增加的功能，就是切面。像上面用的日志，事务都是切面。
          切面的特点：一般都是非业务方法，独立使用的。
  Orient：面向，对着。
  Programming：编程

  oop: 面向对象编程

怎么理解面向切面编程 ？
    1）需要在分析项目功能时，找出切面。
    2）合理的安排切面的执行时间（在目标方法前， 还是目标方法后）
    3）合理的安全切面执行的位置，在哪个类，哪个方法增加增强功能

面向切面编程对有什么好处？
    1.减少重复
    2.专注业务
    注意：面向切面编程只是面向对象编程的一种补充。

什么时候考虑使用aop技术？
    1.当你要给一个系统中存在的类修改功能，但是原有类的功能不完善，但是你还有源代码，使用aop增加功能
```

```
    2.你要给项目中的多个类，增加一个相同的功能，使用aop
    3.给业务方法增加事务，日志输出
```

3 AOP 编程术语

1. 切面（Aspect）：切面泛指交叉业务逻辑。上例中的事务处理、日志处理就可以理解为切面。常用的切面是通知（Advice）、日志、事务、参数检查、权限验证、统计信息。实际就是对主业务逻辑的一种增强。
2. 连接点（JoinPoint）：连接点指可以被切面织入的具体方法。通常业务接口中的方法均为连接点。
3. 切入点（PointCut）：切入点指声明的一个或多个连接点的集合。通过切入点指定一组方法。被标记为 final 的方法是不能作为连接点与切入点的。因为最终的是不能被修改的，不能被增强的。
4. 目标对象（Target）：目标对象指将要被增强的对象。即包含主业务逻辑的类的对象。上例中的 StudentServiceImpl 的对象若被增强，则该类称为目标类，该类对象称为目标对象。当然，不被增强，也就无所谓目标不目标了。
5. 通知（Advice）：通知表示切面的执行时间，Advice 也叫增强。上例中的 MyInvocationHandler 就可以理解为是一种通知。换个角度来说，通知定义了增强代码切入到目标代码的时间点，是目标方法执行之前执行，还是之后执行等。通知类型不同，切入时间不同。切入点定义切入的位置，通知定义切入的时间。

说一个切面有三个关键的要素：

1. 切面的功能代码，切面干什么
2. 切面的执行位置，使用 Pointcut 表示切面执行的位置
3. 切面的执行时间，使用 Advice 表示时间，在目标方法之前，还是目标方法之后。

4 AspectJ 对 AOP 的实现

```
aop的实现
aop是一个规范，是动态的一个规范化，一个标准
aop的技术实现框架：
1.spring：spring在内部实现了aop规范，能做aop的工作。
          spring主要在事务处理时使用aop。
          我们项目开发中很少使用spring的aop实现。 因为spring的aop比较笨重。

2.aspectJ: 一个开源的专门做aop的框架。spring框架中集成了aspectj框架，通过spring就能使用aspectj的功能。
aspectJ框架实现aop有两种方式：
    1.使用xml的配置文件 ： 配置全局事务
    2.使用注解，我们在项目中要做aop功能，一般都使用注解， aspectj有5个注解。
```

4.1 AspectJ 简介

AspectJ 是一个优秀面向切面的框架，它扩展了 Java 语言，提供了强大的切面实现

官网地址：http://www.eclipse.org/aspectj/

AspetJ 是 Eclipse 的开源项目，官网介绍如下：

- a seamless aspect-oriented extension to the Javatm programming language（一种基于 Java 平台的面向切面编程的语言）
- Java platform compatible（兼容 Java 平台，可以无缝扩展）
- easy to learn and use（易学易用）

4.2 AspectJ 的通知类型

AspectJ 中常用的通知(切面的执行时间)有五种类型：在aspectj框架中使用注解表示的。也可以使用xml配置文件中的标签

1. 前置通知 @Before
2. 后置通知　@AfterReturning
3. 环绕通知　@Around
4. 异常通知　@AfterThrowing
5. 最终通知　@After

4.3 AspectJ 的切入点表达式

表示切面执行的位置，使用的是切入点表达式。

AspectJ 定义了专门的表达式用于指定切入点。表达式的原型是：

```
execution(modifiers-pattern? ret-type-pattern
          declaring-type-pattern?name-pattern(param-pattern)
           throws-pattern?)
```

解释：

modifiers-pattern	访问权限类型
ret-type-pattern	返回值类型
declaring-type-pattern	包名类名

name-pattern(param-pattern)　　方法名(参数类型和参数个数)

throws-pattern　　抛出异常类型

?　　表示可选的部分

以上表达式共 4 个部分：execution(访问权限 **方法返回值 方法声明(参数)** 异常类型)

切入点表达式要匹配的对象就是目标方法的方法名。所以，execution 表达式中明显就是方法的签名。注意，表达式中不加粗文字表示可省略部分，各部分间用空格分开。在其中可以使用以下符号：

符号	意义
*	0至多个任意字符
..	用在方法参数中，表示任意多个参数 用在包名后，表示当前包及其子包路径
+	用在类名后，表示当前类及其子类 用在接口后，表示当前接口及其实现类

```
举例：
execution(public * *(..))
指定切入点为：任意公共方法。

execution(* set*(..))
指定切入点为：任何一个以“set”开始的方法。

execution(* com.xyz.service.*.*(..))
指定切入点为：定义在 service 包里的任意类的任意方法。

execution(* com.xyz.service..*.*(..))
指定切入点为：定义在 service 包或者子包里的任意类的任意方法。“..”出现在类名中时，后面必须跟“*”，表示包、子包下的所有类。

execution(* *..service.*.*(..))
指定所有包下的 serivce 子包下所有类（接口）中所有方法为切入点

execution(* *.service.*.*(..))
指定只有一级包下的 serivce 子包下所有类（接口）中所有方法为切入点

execution(* *.ISomeService.*(..))
指定只有一级包下的 ISomeSerivce 接口中所有方法为切入点

execution(* *..ISomeService.*(..))
指定所有包下的 ISomeSerivce 接口中所有方法为切入点

execution(* com.xyz.service.IAccountService.*(..))
指定切入点为：IAccountService 接口中的任意方法。

execution(* com.xyz.service.IAccountService+.*(..))
指定切入点为：IAccountService 若为接口，则为接口中的任意方法及其所有实现类中的任意方法；若为类，则为该类及其子类中的任意方法。

execution(* joke(String,int))
指定切入点为：所有的 joke(String,int)方法，且 joke()方法的第一个参数是 String，第二个参数是 int。如果方法中的参数类型是 java.lang 包下的类，可以直接使用类名，否则必须使用全限定类名，如 joke( java.util.List, int)。

execution(* joke(String,*))
指定切入点为：所有的 joke()方法，该方法第一个参数为 String，第二个参数可以是任意类型，如joke(String s1,String s2)和joke(String s1,double d2)都是，但joke(String s1,double d2,String s3)不是。

execution(* joke(String,..))
指定切入点为：所有的 joke()方法，该方法第一个参数为 String，后面可以有任意个参数且参数类型不限，如 joke(String s1)、joke(String s1,String s2)和 joke(String s1,double d2,String s3)都是。

execution(* joke(Object))
指定切入点为：所有的 joke()方法，方法拥有一个参数，且参数是 Object 类型。joke(Object ob)是，但，joke(String s)与 joke(User u)均不是。

execution(* joke(Object+))
指定切入点为：所有的 joke()方法，方法拥有一个参数，且参数是 Object 类型或该类的子类。不仅 joke(Object ob)是，joke(String s)和 joke(User u)也是。
```

4.4 AspectJ 框架实现 AOP

```
使用aspectj框架实现aop。
使用aop：目的是给已经存在的一些类和方法，增加额外的功能。 前提是不改变原来的类的代码。

使用aspectj实现aop的基本步骤：
1.新建maven项目
2.加入依赖
  1）spring依赖
  2）aspectj依赖
  3）junit单元测试
3.创建目标类：接口和他的实现类。
  要做的是给类中的方法增加功能
4.创建切面类：普通类
  1）在类的上面加入 @Aspect
  2）在类中定义方法，方法就是切面要执行的功能代码
     在方法的上面加入aspectj中的通知注解，例如@Before
     有需要指定切入点表达式execution()
5.创建spring的配置文件：声明对象，把对象交给容器统一管理
  声明对象你可以使用注解或者xml配置文件<bean>
  1) 声明目标对象
  2）声明切面类对象
  3）声明aspectj框架中的自动代理生成器标签。
      自动代理生成器：用来完成代理对象的自动创建功能的。
6.创建测试类，从spring容器中获取目标对象（实际就是代理对象）。
  通过代理执行方法，实现aop的功能增强。
```

1. 新建Maven

   骨架：maven-archetype-quickstart

2. 加入依赖

```xml
<!--pom.xml-->
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <!--Spring 依赖-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.2.5.RELEASE</version>
    </dependency>
    <!--aspectj 依赖-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-aspects</artifactId>
      <version>5.2.5.RELEASE</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

3. 创建目标类：接口和他的实现类。

```java
//接口
package org.example.bao1;
public interface SomeService {
    void doSome(String name, Integer age);
}
```

```java
//接口实现类
package org.example.bao1;
public class SomeServiceImpl implements SomeService {
    @Override
    public void doSome(String name, Integer age) {
        //给目标方法增加一个功能，给doSome()执行之前，输出方法的执行时间
        System.out.println("====目标方法doSome====");
    }
}
```

4. 创建切面类：普通类

```java
package org.example.bao1;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

import java.util.Date;
/**
 *  @Aspect ： 是aspectj框架中的注解。
 *      作用：表示当前类是切面类。
 *      切面类：是用来给业务方法增加功能的类，在这个类中有切面的功能代码
 *      位置：在类定义的上面
 */
@Aspect
public class MyAspectj {
    /*
    * 定义方法，方法是实现切面功能的。
    * 方法的定义要求：
    *       1.公共方法 public
    *       2.方法没有返回值
    *       3.方法名称自定义
    *       4.方法可以有参数，也可以没有参数。
    *          如果有参数，参数不是自定义的，有几个参数类型可以使用。
    */

    /*
     * @Before: 前置通知注解
     *    属性：value，是切入点表达式，表示切面的功能执行的位置。
     *    位置：在方法的上面
     * 特点：
     *  1.在目标方法之前先执行的
     *  2.不会改变目标方法的执行结果
     *  3.不会影响目标方法的执行。
     */
    @Before(value = "execution(public void org.example.bao1.SomeServiceImpl.doSome(String,Integer))")
    public void myBefore(){
        //就是你切面要执行的功能代码
        System.out.println("前置通知，切面功能：在目标方法之前输出执行时间："+ new Date());
    }

    /*
    切入点表达式的多种写法：
    @Before(value = "execution(void org.example.bao1.SomeServiceImpl.doSome(String,Integer))")
    @Before(value = "execution(void *..SomeServiceImpl.doSome(String,Integer))")
    @Before(value = "execution(void *..SomeServiceImpl.do*(String,Integer))")
    @Before(value = "execution(* *..SomeServiceImpl.*(..))")
    @Before(value = "execution(* do*(..))")
    @Before(value = "execution(* org.example.bao1.*ServiceImpl.*(..))")
    */
}
```

5. 创建spring的配置文件：声明对象，把对象交给容器统一管理

```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: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/aop
       https://www.springframework.org/schema/aop/spring-aop.xsd">
    <!--把对象交给spring容器，由spring容器统一创建，管理对象-->
    <!--声明目标对象-->
    <bean id="someService" class="org.example.bao1.SomeServiceImpl" />
    <!--声明切面类对象-->
    <bean id="myAspect" class="org.example.bao1.MyAspectj" />

    <!--声明自动代理生成器：使用aspectj框架内部的功能，创建目标对象的代理对象。
        创建代理对象是在内存中实现的，修改目标对象的内存中的结构。创建为代理对象
        所以目标对象就是被修改后的代理对象.
        aspectj-autoproxy:会把spring容器中的所有的目标对象，一次性都生成代理对象。
```

```
    -->
    <!--<aop:aspectj-autoproxy />-->
    <aop:aspectj-autoproxy />
</beans>
```

6. 创建测试类，从spring容器中获取目标对象（实际就是代理对象）。

```
public class MyTest {
    @Test
    public void test01(){
        String config = "applicationContext.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(config);
        //从容器中获取目标对象
        SomeService proxy = (SomeService) ctx.getBean("someService");
        //proxy: com.sun.proxy.$Proxy8: JDK动态代理
        System.out.println("proxy: " + proxy.getClass().getName());
        //通过代理的对象执行方法，实现目标方法执行时，增强了功能
        proxy.doSome("lisi", 20);
    }
}
/*
proxy: com.sun.proxy.$Proxy8
前置通知，切面功能：在目标方法之前输出执行时间：Sat Feb 19 16:57:08 CST 2022
====目标方法doSome====
*/
```

5 其他 AOP 通知注解的实现

5.1 指定通知方法中的参数：JoinPoint

```
package org.example.bao1;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import java.util.Date;
@Aspect
public class MyAspectj {
    /*
     * 指定通知方法中的参数 ：  JoinPoint
     * JoinPoint:业务方法，要加入切面功能的业务方法
     *    作用是：可以在通知方法中获取方法执行时的信息，例如方法名称，方法的实参。
     *    如果你的切面功能中需要用到方法的信息，就加入JoinPoint.
     *    这个JoinPoint参数的值是由框架赋予，必须是第一个位置的参数
     */
    @Before(value = "execution(void *..SomeServiceImpl.doSome(String,Integer))")
    public void myBefore(JoinPoint jp){
        //获取方法的完整定义
        System.out.println("方法的签名（定义）=" + jp.getSignature());
        System.out.println("方法的名称=" + jp.getSignature().getName());
        //获取方法的实参
        Object args [] = jp.getArgs();
        for (Object arg:args){
            System.out.println("参数="+arg);
        }
        //就是你切面要执行的功能代码
        System.out.println("2=====前置通知，切面功能：在目标方法之前输出执行时间："+ new Date());
    }
}
/*
proxy: com.sun.proxy.$Proxy8
方法的签名（定义）=void org.example.bao1.SomeService.doSome(String,Integer)
方法的名称=doSome
参数=lisi
参数=20
2=====前置通知，切面功能：在目标方法之前输出执行时间：Sat Feb 19 19:21:54 CST 2022
====目标方法doSome====
*/
```

5.2 @AfterReturning 后置通知 - 注解有 returning 属性

```
//接口
package org.example.bao2;
public interface SomeService {
    String doOther(String name, Integer age);
    Student doAdd(String name, Integer age);
}
```

```
//接口实现类
package org.example.bao2;
```

```java
public class SomeServiceImpl implements SomeService {
    @Override
    public String doOther(String name, Integer age) {
        System.out.println("====目标方法doOther====");
        return "spring-aop";
    }

    @Override
    public Student doAdd(String name, Integer age) {
        Student student = new Student();
        student.setName(name);
        student.setAge(age);
        System.out.println("student = " + student);
        return student;
    }
}
```

```java
//切面类
package org.example.bao2;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
/**
 *  @Aspect : 是aspectj框架中的注解。
 *      作用：表示当前类是切面类。
 *      切面类：是用来给业务方法增加功能的类，在这个类中有切面的功能代码
 *      位置：在类定义的上面
 */
@Aspect
public class MyAspectj {
    /*
     * 后置通知定义方法，方法是实现切面功能的。
     * 方法的定义要求：
     *      1.公共方法 public
     *      2.方法没有返回值
     *      3.方法名称自定义
     *      4.方法有参数的，推荐是Object，参数名自定义
     */

    /*
     * @AfterReturning:后置通知
     *    属性：1.value 切入点表达式
     *          2.returning 自定义的变量，表示目标方法的返回值的。
     *           自定义变量名必须和通知方法的形参名一样。
     *    位置：在方法定义的上面
     *
     * 特点：
     *  1。在目标方法之后执行的。
     *  2. 能够获取到目标方法的返回值，可以根据这个返回值做不同的处理功能
     *     相当于 Object res = doOther();
     *  3. 可以修改这个返回值
     *
     *  后置通知的执行
     *    Object res = doOther();
     *    参数传递： 传值， 传引用
     *    myAfterReturing(res);
     *    System.out.println("res="+res)
     */
    @AfterReturning(value = "execution(* *..SomeServiceImpl.doOther(..))", returning = "res")
    public void myAfterReturing(JoinPoint jp , Object res){
        // Object res:是目标方法执行后的返回值，根据返回值做你的切面的功能处理
        System.out.println("后置通知：方法的定义"+ jp.getSignature());
        System.out.println("后置通知：在目标方法之后执行的，获取的返回值是：" + res);
        if(res.equals("spring-aop")){
            //做一些功能
        } else{
            //做其它功能
        }
        //修改目标方法的返回值，看一下是否会影响 最后的方法调用结果
        if( res != null){
            res = "Hello Aspectj";
        }
    }

    @AfterReturning(value = "execution(* *..SomeServiceImpl.doAdd(..))", returning = "res")
    public void myAfterReturing2(JoinPoint jp , Object res){
        // Object res:是目标方法执行后的返回值，根据返回值做你的切面的功能处理
        System.out.println("后置通知：方法的定义"+ jp.getSignature());
        System.out.println("后置通知：在目标方法之后执行的，获取的返回值是：" + res);

        //修改目标方法的返回值，看一下是否会影响 最后的方法调用结果
        if(res != null && res instanceof Student){
            Student res1 = (Student) res;
```

```
            res1.setAge(33);
            res1.setName("张三丰");
        }
    }
}
```

```
<!--applicationContext.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: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/aop
       https://www.springframework.org/schema/aop/spring-aop.xsd">
    <!--把对象交给spring容器，由spring容器统一创建，管理对象-->
    <!--声明目标对象-->
    <bean id="someService" class="org.example.bao2.SomeServiceImpl" />
    <!--声明切面类对象-->
    <bean id="myAspect" class="org.example.bao2.MyAspectj" />

    <!--声明自动代理生成器：使用aspectj框架内部的功能，创建目标对象的代理对象。
        创建代理对象是在内存中实现的， 修改目标对象的内存中的结构。 创建为代理对象
        所以目标对象就是被修改后的代理对象.
        aspectj-autoproxy:会把spring容器中的所有的目标对象，一次性都生成代理对象。
    -->
    <!--<aop:aspectj-autoproxy />-->
    <aop:aspectj-autoproxy />
</beans>
```

```
package org.example;

import org.example.bao2.SomeService;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    @Test
    public void test01(){
        String config = "applicationContext.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(config);
        //从容器中获取目标对象
        SomeService proxy = (SomeService) ctx.getBean("someService");
        //通过代理的对象执行方法，实现目标方法执行时，增强了功能
        String str = proxy.doOther("张三", 28);
        System.out.println("str = " + str);
        System.out.println("=======================================");
        Student s = proxy.doAdd("张三", 28);
        System.out.println("student = " + s);
    }
}
/*
====目标方法doOther====
后置通知：方法的定义String org.example.bao2.SomeService.doOther(String,Integer)
后置通知：在目标方法之后执行的，获取的返回值是：spring-aop
str = spring-aop
=======================================
student = Student{name='张三', age=28}
后置通知：方法的定义Student org.example.bao2.SomeService.doAdd(String,Integer)
后置通知：在目标方法之后执行的，获取的返回值是：Student{name='张三', age=28}
student = Student{name='张三丰', age=33}
*/
```

5.3 @Around 环绕通知-增强方法有 ProceedingJoinPoint 参数

在目标方法执行之前之后执行。被注解为环绕增强的方法要有返回值，Object 类型。并且方法可以包含一个 ProceedingJoinPoint 类型的参数。接口 ProceedingJoinPoint 其有一个 proceed() 方法，用于执行目标方法。若目标方法有返回值，则该方法的返回值就是目标方法的返回值。最后，环绕增强方法将其返回值返回。该增强方法实际是拦截了目标方法的执行。

```
package org.example.bao3;
public interface SomeService {
    String doFirst(String name, Integer age);
}
```

```java
package org.example.bao3;
import org.example.bao1.Student;
public class SomeServiceImpl implements SomeService {
    @Override
    public String doFirst(String name, Integer age) {
        System.out.println("====业务方法doFirst()====");
        return "doFirst";
    }
}
```

```java
package org.example.bao3;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.example.bao1.Student;
import java.util.Date;
@Aspect
public class MyAspectj {
    /**
    * 环绕通知方法的定义格式
    *       1.public
    *       2.必须有一个返回值，推荐使用Object
    *       3.方法名称自定义
    *       4.方法有参数，固定的参数 ProceedingJoinPoint
    */

    /*
     * @Around: 环绕通知
     *      属性：value 切入点表达式
     *      位置：在方法的定义上面
     * 特点：
     *   1.它是功能最强的通知
     *   2.在目标方法的前和后都能增强功能。
     *   3.控制目标方法是否被调用执行
     *   4.修改原来的目标方法的执行结果。影响最后的调用结果
     *
     *  环绕通知，等同于jdk动态代理的，InvocationHandler接口
     *
     *  参数：ProceedingJoinPoint 就等同于 Method
     *       作用：执行目标方法的
     *  返回值：就是目标方法的执行结果，可以被修改。
     *
     *  环绕通知：经常做事务，在目标方法之前开启事务，执行目标方法，在目标方法之后提交事务
     */
    @Around(value = "execution(* *..SomeServiceImpl.doFirst(..))")
    public Object myAround(ProceedingJoinPoint pjp) throws Throwable {
        String name = "";
        //获取第一个参数值
        Object args [] = pjp.getArgs();
        if( args!= null && args.length > 1){
            Object arg = args[0];
            name = (String) arg;
        }

        //实现环绕通知
        Object result = null;
        System.out.println("环绕通知：在目标方法之前，输出时间："+ new Date());
        //1.目标方法调用
        if( "张三".equals(name)){
            //符合条件，调用目标方法
            result = pjp.proceed(); //method.invoke(); Object result = doFirst();
        }
        //2.在目标方法后加入功能
        System.out.println("环绕通知：在目标方法之后，提交事务");
        //修改目标方法的执行结果， 影响方法最后的调用结果
        if( result != null){
            result = "Hello AspectJ AOP";
        }
        //返回目标方法的执行结果
        return result;
    }
}
```

```java
public class MyTest03 {
    @Test
    public void test01(){
        String config = "applicationContext.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(config);
        //从容器中获取目标对象
        SomeService proxy = (SomeService) ctx.getBean("someService");
```

```
        //通过代理的对象执行方法，实现目标方法执行时，增强了功能
        String str = proxy.doFirst("张三", 30);
        System.out.println("str = " + str);
    }
}
/*
环绕通知：在目标方法之前，输出时间：Sat Feb 19 20:15:43 CST 2022
====业务方法doFirst()====
环绕通知：在目标方法之后，提交事务
str = Hello AspectJ AOP
*/
```

5.4 @AfterThrowing 异常通知-注解中有 throwing 属性

在目标方法抛出异常后执行。该注解的 throwing 属性用于指定所发生的异常类对象。当然，被注解为异常通知的方法可以包含一个参数 Throwable，参数名称为 throwing 指定的名称，表示发生的异常对象。

```
package org.example.bao4;
public interface SomeService {
    void doSecond();
}
```

```
package org.example.bao4;
public class SomeServiceImpl implements SomeService {
    @Override
    public void doSecond() {
        System.out.println("====业务方法doSecond()====" + (10/0));
    }
}
```

```
package org.example.bao4;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import java.util.Date;
@Aspect
public class MyAspectj {
    /**
    * 异常通知方法的定义格式
    *    1.public
    *    2.没有返回值
    *    3.方法名称自定义
    *    4.方法有一个参数 Exception，如果还有是 JoinPoint,
    */

    /*
     * @AfterThrowing: 异常通知
     *     属性：1. value 切入点表达式
     *          2. throwinng 自定义的变量，表示目标方法抛出的异常对象。
     *             变量名必须和方法的参数名一样
     * 特点：
     *   1. 在目标方法抛出异常时执行的
     *   2. 可以做异常的监控程序， 监控目标方法执行时是不是有异常。
     *      如果有异常，可以发送邮件，短信进行通知
     *
     *  执行就是：
     *   try{
     *       SomeServiceImpl.doSecond(..)
     *   }catch(Exception e){
     *       myAfterThrowing(e);
     *   }
     */
    @AfterThrowing(value = "execution(* *..SomeServiceImpl.doSecond(..))", throwing = "ex")
    public void myAfterThrowing(Exception ex) {
        System.out.println("异常通知：方法发生异常时，执行：" + ex.getMessage());
        //发送邮件，短信，通知开发人员
    }
}
```

```
public class MyTest04 {
    @Test
    public void test01(){
        String config = "applicationContext.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(config);
        //从容器中获取目标对象
        SomeService proxy = (SomeService) ctx.getBean("someService");
        proxy.doSecond();
    }
}
/*
```

```
异常通知：方法发生异常时，执行：/ by zero

java.lang.ArithmeticException: / by zero
    at org.example.bao4.SomeServiceImpl.doSecond(SomeServiceImpl.java:6)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    ......
*/
```

5.5 @After 最终通知

无论目标方法是否抛出异常，该增强均会被执行。

```
public interface SomeService {
    void doThird();
}
```

```
public class SomeServiceImpl implements SomeService {
    @Override
    public void doThird() {
        System.out.println("====业务方法doThird()====" + (10/0));
    }
}
```

```
@Aspect
public class MyAspectj {
    /**
     * 最终通知方法的定义格式
     *      1.public
     *      2.没有返回值
     *      3.方法名称自定义
     *      4.方法没有参数，如果还有是JoinPoint,
     */

    /*
     * @After :最终通知
     *      属性：value 切入点表达式
     *      位置：在方法的上面
     * 特点：
     *  1.总是会执行
     *  2.在目标方法之后执行的
     *
     *  try{
     *      SomeServiceImpl.doThird(..)
     *  }catch(Exception e){
     *
     *  }finally{
     *      myAfter()
     *  }
     */
    @After(value = "execution(* *..SomeServiceImpl.doThird(..))")
    public void myAfter(){
        System.out.println("执行最终通知，总是会被执行的代码");
        //一般做资源清除工作的。
    }
}
```

```
public class MyTest05 {
    @Test
    public void test01(){
        String config = "applicationContext.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(config);
        //从容器中获取目标对象
        SomeService proxy = (SomeService) ctx.getBean("someService");
        //通过代理的对象执行方法，实现目标方法执行时，增强了功能
        proxy.doThird();
    }
}
/*
执行最终通知，总是会被执行的代码

java.lang.ArithmeticException: / by zero
    at org.example.bao5.SomeServiceImpl.doThird(SomeServiceImpl.java:6)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    ......
*/
```

5.6 @Pointcut 定义切入点

当较多的通知增强方法使用相同的 execution 切入点表达式时，编写、维护均较为麻烦。AspectJ 提供了@Pointcut 注解，用于定义 execution 切入点表达式。

其用法是，将 @Pointcut 注解在一个方法之上，以后所有的 execution 的 value 属性值均可使用该方法名作为切入点。代表的就是 @Pointcut 定义的切入点。这个使用 @Pointcut 注解的方法一般使用 private 的标识方法，即没有实际作用的方法

```
@Aspect
public class MyAspectj {
    @After(value = "mypt()")
    public void myAfter(){
        System.out.println("执行最终通知，总是会被执行的代码");
        //一般做资源清除工作的。
    }

    @Before(value = "mypt()")
    public void myBefore(){
        System.out.println("前置通知，在目标方法之前先执行的");
    }

    /*
     * @Pointcut: 定义和管理切入点，如果你的项目中有多个切入点表达式是重复的，可以复用的。可以使用@Pointcut
     *    属性：value 切入点表达式
     *    位置：在自定义的方法上面
     * 特点：
     *   当使用 @Pointcut 定义在一个方法的上面，此时这个方法的名称就是切入点表达式的别名。
     *   其它的通知中，value属性就可以使用这个方法名称，代替切入点表达式了
     */
    @Pointcut(value = "execution(* *..SomeServiceImpl.doThird(..))" )
    private void mypt(){ /*无需代码*/ }
}
```

5.7 cglib 代理

```
public class SomeServiceImpl {
    public void doThird() {
        System.out.println("====业务方法doThird()====");
    }
}
```

```
@Aspect
public class MyAspectj {
    @After(value = "mypt()")
    public void myAfter(){
        System.out.println("执行最终通知，总是会被执行的代码");
        //一般做资源清除工作的。
    }

    @Before(value = "mypt()")
    public void myBefore(){
        System.out.println("前置通知，在目标方法之前先执行的");
    }

    /*
     * @Pointcut: 定义和管理切入点，如果你的项目中有多个切入点表达式是重复的，可以复用的。可以使用@Pointcut
     *    属性：value 切入点表达式
     *    位置：在自定义的方法上面
     * 特点：
     *   当使用 @Pointcut 定义在一个方法的上面，此时这个方法的名称就是切入点表达式的别名。
     *   其它的通知中，value属性就可以使用这个方法名称，代替切入点表达式了
     */
    @Pointcut(value = "execution(* *..SomeServiceImpl.doThird(..))" )
    private void mypt(){ /*无需代码*/ }
}
```

```
    <!--声明目标对象-->
    <bean id="someService" class="org.example.bao7.SomeServiceImpl" />
    <!--声明切面类对象-->
    <bean id="myAspect" class="org.example.bao7.MyAspectj" />

    <!--声明自动代理生成器：使用aspectj框架内部的功能，创建目标对象的代理对象。
        创建代理对象是在内存中实现的， 修改目标对象的内存中的结构。 创建为代理对象
        所以目标对象就是被修改后的代理对象.
        aspectj-autoproxy:会把spring容器中的所有的目标对象，一次性都生成代理对象。
    -->
    <!--<aop:aspectj-autoproxy />-->
    <aop:aspectj-autoproxy />
```

```
public class MyTest07 {
    @Test
```

```
    public void test01(){
        String config = "applicationContext.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(config);
        //从容器中获取目标对象
        SomeServiceImpl proxy = (SomeServiceImpl) ctx.getBean("someService");
        /*
        * 目标类没有接口，使用cglib动态接口，spring框架会自动使用cglib
        * proxy: org.example.bao7.SomeServiceImpl$$EnhancerBySpringCGLIB$$2b7b679f
        * */
        System.out.println("proxy: " + proxy.getClass().getName());
        //通过代理的对象执行方法，实现目标方法执行时，增强了功能
        proxy.doThird();
    }
}
/*
proxy: org.example.bao7.SomeServiceImpl$$EnhancerBySpringCGLIB$$2b7b679f
前置通知，在目标方法之前先执行的
====业务方法doThird()====
执行最终通知，总是会被执行的代码
*/
```

有接口也可以使用cglib代理

```
    <!--声明自动代理生成器：使用aspectj框架内部的功能，创建目标对象的代理对象。
        创建代理对象是在内存中实现的， 修改目标对象的内存中的结构。 创建为代理对象
        所以目标对象就是被修改后的代理对象.

        aspectj-autoproxy:会把spring容器中的所有的目标对象，一次性都生成代理对象。
    -->
    <!--<aop:aspectj-autoproxy />-->


    <!--
       如果你期望目标类有接口，使用cglib代理
       proxy-target-class="true":告诉框架，要使用cglib动态代理
    -->
    <aop:aspectj-autoproxy proxy-target-class="true"/>
```

四、Spring 集成 MyBatis

将 MyBatis 与 Spring 进行整合，主要解决的问题就是将 SqlSessionFactory 对象交由 Spring 来管理。所以，该整合只需要将 SqlSessionFactory 的对象生成器 SqlSessionFactoryBean 注册在 Spring 容器中，再将其注入给 Dao 的实现类即可完成整合。

实现 Spring 与 MyBatis 的整合常用的方式：扫描的 Mapper 动态代理；

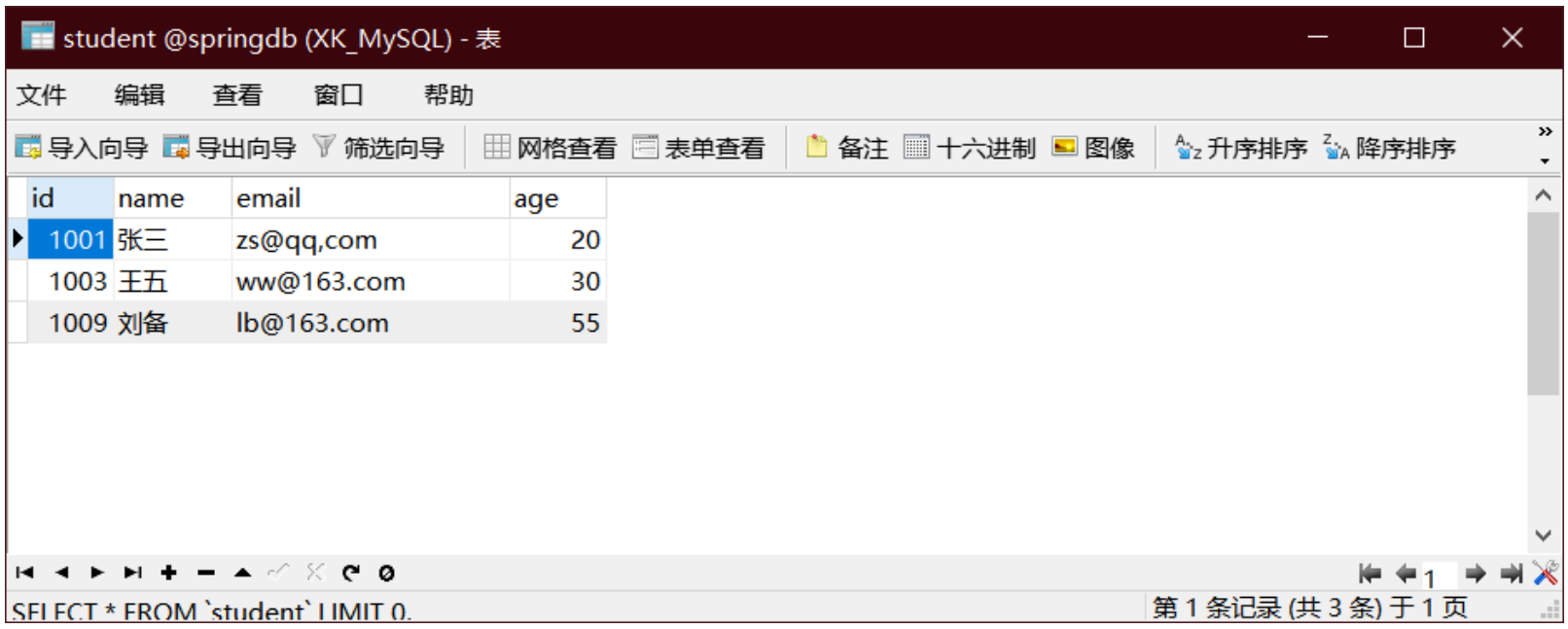
Spring 像插线板一样，mybatis 框架是插头，可以容易的组合到一起。插线板 spring 插上 mybatis，两个框架就是一个整体。

```
把 mybatis框架和 spring 集成在一起，像一个框架一样使用。

用的技术是：ioc
为什么是ioc：能把mybatis和spring集成在一起，像一个框架，是因为ioc能创建对象。
    可以把mybatis框架中的对象交给spring统一创建，开发人员从spring中获取对象。
    开发人员就不用同时面对两个或多个框架了，就面对一个spring

mybatis使用步骤，对象
    1.定义dao接口 ，StudentDao
    2.定义mapper文件 StudentDao.xml
    3.定义mybatis的主配置文件 mybatis.xml
    4.创建dao的代理对象， StudentDao dao = SqlSession.getMapper(StudentDao.class);
        List<Student> students  = dao.selectStudents();

要使用dao对象，需要使用getMapper()方法，
怎么能使用getMapper()方法，需要哪些条件
    1.获取SqlSession对象， 需要使用SqlSessionFactory的openSession()方法。
    2.创建SqlSessionFactory对象。通过读取mybatis的主配置文件，能创建SqlSessionFactory对象

需要SqlSessionFactory对象，使用Factory能获取SqlSession，有了SqlSession就能有dao，目的就是获取dao对象
Factory创建需要读取主配置文件

我们会使用独立的连接池类替换mybatis默认自己带的，把连接池类也交给spring创建。

主配置文件：
 1.数据库信息
        <environment id="mydev">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/springdb"/>
                <property name="username" value="root"/>
```

```
                <property name="password" value="123456"/>
            </dataSource>
         </environment>
2. mapper文件的位置
   <mappers>
        <mapper resource="com/bjpowernode/dao/StudentDao.xml"/>
        <!--<mapper resource="com/bjpowernode/dao/SchoolDao.xml" />-->
   </mappers>

================================================================
通过以上的说明，我们需要让spring创建以下对象
1.独立的连接池类的对象，使用阿里的druid连接池
2.SqlSessionFactory对象
3.创建出dao对象

需要学习就是上面三个对象的创建语法，使用xml的bean标签。

连接池：多个连接Connection对象的集合，List<Connection> connlist : connList就是连接池

通常使用Connection访问数据库
Connection conn = DriverManger.getConnection(url,username,password);
Statemenet stmt = conn.createStatement(sql);
stmt.executeQuery();
conn.close();

使用连接池
在程序启动的时候，先创建一些Connection
Connection c1 = ...
Connection c2 = ...
Connection c3 = ...
List<Connection>  connlist = new ArrayLits();
connList.add(c1);
connList.add(c2);
connList.add(c3);

Connection conn = connList.get(0);
Statemenet stmt = conn.createStatement(sql);
stmt.executeQuery();
把使用过的connection放回到连接池
connList.add(conn);

Connection conn1 = connList.get(1);
Statemenet stmt = conn1.createStatement(sql);
stmt.executeQuery();
把使用过的connection放回到连接池
connList.add(conn1);
```

```
spring和mybatis的集成
步骤：
1.新建maven项目
2.加入maven的依赖
  1）spring依赖
  2）mybatis依赖
  3）mysql驱动
  4）spring的事务的依赖
  5）mybatis和spring集成的依赖：mybatis官方体用的，用来在spring项目中创建mybatis
     的SqlSesissonFactory，dao对象的
3.创建实体类
4.创建dao接口和mapper文件
5.创建mybatis主配置文件
6.创建Service接口和实现类，属性是dao。
7.创建spring的配置文件：声明mybatis的对象交给spring创建
 1）数据源DataSource
 2）SqlSessionFactory
 3) Dao对象
 4）声明自定义的service
8.创建测试类，获取Service对象，通过service调用dao完成数据库的访问
```

1 MySQL 创建数据库 springdb，新建表 Student




```sql
CREATE TABLE `student` (
 `id` int(11) NOT NULL ,
 `name` varchar(255) DEFAULT NULL,
 `email` varchar(255) DEFAULT NULL,
 `age` int(11) DEFAULT NULL,
 PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

2 maven 依赖 pom.xml

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

  <groupId>org.example</groupId>
  <artifactId>ch07-spring-mybatis</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <!--单元测试-->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <!--Spring核心IOC-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.2.5.RELEASE</version>
    </dependency>
    <!--做Spring事务用到的-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-tx</artifactId>
      <version>5.2.5.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>5.2.5.RELEASE</version>
    </dependency>
    <!--mybatis依赖-->
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis</artifactId>
      <version>3.5.1</version>
    </dependency>
    <!--mybatis和spring集成的依赖-->
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis-spring</artifactId>
```

```
      <version>1.3.1</version>
    </dependency>
    <!--mysql驱动-->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.9</version>
    </dependency>
    <!--阿里公司的数据库连接池-->
    <dependency>
      <groupId>com.alibaba</groupId>
      <artifactId>druid</artifactId>
      <version>1.1.12</version>
    </dependency>
  </dependencies>

  <build>
    <resources>
      <!--目的是把src/main/java目录中的xml文件包含到输出结果中，输出到classes目录中-->
      <resource>
        <directory>src/main/java</directory><!--所在的目录-->
        <includes><!--包括目录下的.properties,.xml 文件都会扫描到-->
          <include>**/*.properties</include>
          <include>**/*.xml</include>
        </includes>
        <filtering>false</filtering>
      </resource>
    </resources>
    <plugins>
      <!--指定JDK的版本-->
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

3 定义实体类 Student

```
package org.example.domain;

public class Student {
    //属性名和列名一样
    private Integer id;
    private String name;
    private String email;
    private Integer age;

    public Student() {}
    public Student(Integer id, String name, String email, Integer age) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.age = age;
    }

    public Integer getId() {return id; }
    public void setId(Integer id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
    public Integer getAge() { return age; }
    public void setAge(Integer age) { this.age = age; }

    @Override
    public String toString() {
        return "Student{" +  "id=" + id + ", name='" + name + '\'' + ", email='" + email + '\'' + ",
age=" + age + '}';
    }
}
```

4 定义 StudentDao 接口

```java
package org.example.dao;
import org.example.domain.Student;
import java.util.List;
public interface StudentDao {
    int insertStudent(Student student);
    List<Student> selectStudents();
}
```

5 定义映射文件 mapper

```xml
<!--package org.example.dao.StudentDao.xml-->
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="org.example.dao.StudentDao">
    <select id="selectStudents" resultType="org.example.domain.Student">
        select id, name, email, age from student order by id desc
    </select>

    <insert id="insertStudent">
        insert into student values(#{id}, #{name}, #{email}, #{age})
    </insert>
</mapper>
```

6 定义 MyBatis 主配置文件

```xml
<!--src/main/resources/mybatis_config.xml-->

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
   <settings>
       <!--设置mybatis输出日志-->
       <setting name="logImpl" value="STDOUT_LOGGING"/>
   </settings>

    <!--设置别名-->
    <typeAliases>
        <!--name：实体类所在的包名-->
        <package name="org.example.domain"/>
    </typeAliases>

    <!-- sql mapper（sql映射文件）的位置-->
    <mappers>
        <!-- name：是包名，这个包中的所有 mapper.xml 一次都能加载 -->
        <package name="org.example.dao"/>
    </mappers>
</configuration>
```

7 定义 Service 接口和实现类

```java
package org.example.service;
import org.example.domain.Student;
import java.util.List;
public interface StudentService {
    int addStudent(Student student);
    List<Student> queryStudents();
}
```

```java
package org.example.service.impl;

import org.example.dao.StudentDao;
import org.example.domain.Student;
import org.example.service.StudentService;

import java.util.List;

public class StudentServiceImpl implements StudentService {
    //引用类型
    private StudentDao studentDao;
    //使用set注入，赋值
    public void setStudentDao(StudentDao studentDao) {
        this.studentDao = studentDao;
    }

    @Override
```

```
    public int addStudent(Student student) {
        int nums = studentDao.insertStudent(student);
        return nums;
    }

    @Override
    public List<Student> queryStudents() {
        List<Student> students = studentDao.selectStudents();
        return students;
    }
}
```

8 修改 Spring 配置文件

8.1 数据源的配置(掌握)

Druid 数据源 DruidDataSource：

Druid 是阿里的开源数据库连接池。是 Java 语言中最好的数据库连接池。Druid 能够提供强大的监控和扩展功能。

官网：https://github.com/alibaba/druid

使用地址：https://github.com/alibaba/druid/wiki/FAQ

```
    <!--声明数据源DataSource,作用是连接数据库-->
    <bean id="myDataSource" class="com.alibaba.druid.pool.DruidDataSource"
          init-method="init" destroy-method="close">
        <!--set注入给DuridDataSource提供连接数据库的信息-->
        <property name="url" value="jdbc:mysql://localhost:3306/springdb" />
        <property name="username" value="root" />
        <property name="password" value="111" />
        <property name="maxActive" value="20" /> <!--最大连接数-->
    </bean>
```

8.2 从属性文件读取数据库连接信息

```
#src/main/resources/jdbc.properties
jdbc.url=jdbc:mysql://localhost:3306/springdb
jdbc.username=root
jdbc.password=111
jdbc.maxActive=20
```

```
    <!--
        把数据库的配置信息，写在一个独立的文件，编译修改数据库的配置内容
        spring知道jdbc.properties文件的位置
    -->
    <context:property-placeholder location="classpath:jdbc.properties" />

    <!--声明数据源DataSource,作用是连接数据库-->
    <bean id="myDataSource" class="com.alibaba.druid.pool.DruidDataSource"
          init-method="init" destroy-method="close">
        <!--set注入给DuridDataSource提供连接数据库的信息-->
        <property name="url" value="${jdbc.url}" />
        <property name="username" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
        <property name="maxActive" value="${jdbc.maxActive}" /> <!--最大连接数-->
    </bean>
```

8.3 注册 SqlSessionFactoryBean

```
    <!--声明的是mybatis中提供的SqlSessionFactoryBean类，这个类内部创建SqlSessionFactory的
         SqlSessionFactory sqlSessionFactory = new ..
     -->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <!--set注入，把数据库连接池付给了dataSource属性-->
        <property name="dataSource" ref="myDataSource" />
        <!--
            mybatis主配置文件的位置
            configLocation属性是Resource类型，读取配置文件
            它的赋值，使用value，指定文件的路径，使用 classpath:表示文件的位置
        -->
        <property name="configLocation" value="classpath:mybatis_config.xml" />
    </bean>
```

8.4 定义 Mapper 扫描配置器 MapperScannerConfigurer

```
    <!--创建dao对象，使用SqlSession的getMapper(StudentDao.class)
        MapperScannerConfigurer:在内部调用getMapper()生成每个dao接口的代理对象
    -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer" >
        <!--指定SqlSessionFactory对象的id-->
        <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />
        <!--指定包名，包名是dao接口所在的包名。
            MapperScannerConfigurer会扫描这个包中的所有接口，把每个接口都执行
            一次getMapper()方法，得到每个接口的dao对象。
            创建好的dao对象放入到spring的容器中的。 dao对象的默认名称是 接口名首字母小写
        -->
        <property name="basePackage" value="org.example.dao"/>
    </bean>
```

8.5 向 Service 注入接口名

向 Service 注入 Mapper 代理对象时需要注意，由于通过 Mapper 扫描配置器 MapperScannerConfigurer 生成的 Mapper 代理对象没有名称，所以在向 Service 注入 Mapper 代理时，无法通过名称注入。但可通过接口的简单类名注入，因为生成的是这个 Dao 接口的对象。

```
    <!--声明service-->
    <bean id="studentService" class="org.example.service.impl.StudentServiceImpl">
        <property name="studentDao" ref="studentDao" />
    </bean>
```

9 Spring 配置文件全部配置

```
<!--src/main/resources/applicationcontext.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
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       https://www.springframework.org/schema/context/spring-context.xsd">

    <!--
       把数据库的配置信息，写在一个独立的文件，编译修改数据库的配置内容
       spring知道jdbc.properties文件的位置
    -->
    <context:property-placeholder location="classpath:jdbc.properties" />

    <!--声明数据源DataSource,作用是连接数据库-->
    <bean id="myDataSource" class="com.alibaba.druid.pool.DruidDataSource"
          init-method="init" destroy-method="close">
        <!--set注入给DuridDataSource提供连接数据库的信息-->
        <property name="url" value="${jdbc.url}" />
        <property name="username" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
        <property name="maxActive" value="${jdbc.maxActive}" /> <!--最大连接数-->
     </bean>

     <!--声明的是mybatis中提供的SqlSessionFactoryBean类，这个类内部创建SqlSessionFactory的
         SqlSessionFactory  sqlSessionFactory = new ..
     -->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <!--set注入，把数据库连接池付给了dataSource属性-->
        <property name="dataSource" ref="myDataSource" />
        <!--
           mybatis主配置文件的位置
           configLocation属性是Resource类型，读取配置文件
           它的赋值，使用value，指定文件的路径，使用 classpath:表示文件的位置
        -->
        <property name="configLocation" value="classpath:mybatis_config.xml" />
    </bean>

    <!--创建dao对象，使用SqlSession的getMapper(StudentDao.class)
        MapperScannerConfigurer:在内部调用getMapper()生成每个dao接口的代理对象
    -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer" >
        <!--指定SqlSessionFactory对象的id-->
        <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />
        <!--指定包名，包名是dao接口所在的包名。
            MapperScannerConfigurer会扫描这个包中的所有接口，把每个接口都执行
            一次getMapper()方法，得到每个接口的dao对象。
            创建好的dao对象放入到spring的容器中的。 dao对象的默认名称是 接口名首字母小写
        -->
        <property name="basePackage" value="org.example.dao"/>
    </bean>

    <!--声明service-->
    <bean id="studentService" class="org.example.service.impl.StudentServiceImpl">
```

```
        <property name="studentDao" ref="studentDao" />
    </bean>
</beans>
```

10 测试

```
package org.example;

import org.example.dao.StudentDao;
import org.example.domain.Student;
import org.example.service.StudentService;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.List;

public class MyTest01 {
    @Test
    public void testInsert(){
        String config = "applicationcontext.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(config);
        StudentService service = (StudentService) ctx.getBean("studentService");
        Student student = new Student(1004, "王五", "ww@gmail.com", 25);
        int nums = service.addStudent(student);
        System.out.println("nums = " + nums);
    }

    @Test
    public void testSelect(){
        String config = "applicationcontext.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(config);
        StudentService service = (StudentService) ctx.getBean("studentService");
        List<Student> students = service.queryStudents();
        students.forEach(s -> System.out.println("student = " + s));
    }
}

```

五、Spring 事务

```
spring的事务处理
回答问题
1.什么是事务
  讲mysql的时候，提出了事务。事务是指一组sql语句的集合，集合中有多条sql语句
  可能是insert，update，select，delete，我们希望这些多个sql语句都能成功，
  或者都失败，这些sql语句的执行是一致的，作为一个整体执行。

2.在什么时候想到使用事务
  当我的操作，涉及得到多个表，或者是多个sql语句的insert，update，delete。需要保证
  这些语句都是成功才能完成我的功能，或者都失败，保证操作是符合要求的。

  在java代码中写程序，控制事务，此时事务应该放在那里呢？
     service类的业务方法上，因为业务方法会调用多个dao方法，执行多个sql语句

3.通常使用JDBC访问数据库，  还是mybatis访问数据库怎么处理事务
    jdbc访问数据库，处理事务   Connection conn ; conn.commit(); conn.rollback();
    mybatis访问数据库，处理事务，  SqlSession.commit();  SqlSession.rollback();
    hibernate访问数据库，处理事务，  Session.commit(); Session.rollback();

4. 3问题中事务的处理方式，有什么不足
  1) 不同的数据库访问技术，处理事务的对象，方法不同，
     需要了解不同数据库访问技术使用事务的原理
  2) 掌握多种数据库中事务的处理逻辑。什么时候提交事务，什么时候回滚事务
  3) 处理事务的多种方法。
  总结：就是多种数据库的访问技术，有不同的事务处理的机制，对象，方法。

5.怎么解决不足
  spring提供一种处理事务的统一模型，  能使用统一步骤，方式完成多种不同数据库访问技术的事务处理。
  使用spring的事务处理机制，可以完成mybatis访问数据库的事务处理
  使用spring的事务处理机制，可以完成hibernate访问数据库的事务处理。

6.处理事务，需要怎么做，做什么
  spring处理事务的模型，使用的步骤都是固定的。把事务使用的信息提供给spring就可以了
  1）事务内部提交，回滚事务，使用的事务管理器对象，代替你完成commit，rollback
     事务管理器是一个接口和他的众多实现类。
      接口：PlatformTransactionManager，定义了事务重要方法 commit，rollback
      实现类：spring把每一种数据库访问技术对应的事务处理类都创建好了。
             mybatis访问数据库---spring创建好的是 DataSourceTransactionManager
            hibernate访问数据库----spring创建的是 HibernateTransactionManager
```

```
    怎么使用：你需要告诉spring 你用是那种数据库的访问技术，怎么告诉spring呢？
    声明数据库访问技术对于的事务管理器实现类，在spring的配置文件中使用<bean>声明就可以了
    例如，你要使用mybatis访问数据库，你应该在xml配置文件中
    <bean id="xxx" class="...DataSourceTransactionManager">

2）你的业务方法需要什么样的事务，说明需要事务的类型。
   说明方法需要的事务：
      1. 事务的隔离级别：有4个值。
      DEFAULT：采用 DB 默认的事务隔离级别。MySql 的默认为 REPEATABLE_READ； Oracle默认为 READ_COMMITTED。
      ➢ READ_UNCOMMITTED：读未提交。未解决任何并发问题。
      ➢ READ_COMMITTED：读已提交。解决脏读，存在不可重复读与幻读。
      ➢ REPEATABLE_READ：可重复读。解决脏读、不可重复读，存在幻读
      ➢ SERIALIZABLE：串行化。不存在并发问题。

      2. 事务的超时时间： 表示一个方法最长的执行时间，如果方法执行时超过了时间，事务就回滚。
        单位是秒，整数值，默认是-1.

      3. 事务的传播行为 ： 控制业务方法是不是有事务的， 是什么样的事务的。
          7个传播行为，表示你的业务方法调用时，事务在方法之间是如果使用的。

          PROPAGATION_REQUIRED
          PROPAGATION_REQUIRES_NEW
          PROPAGATION_SUPPORTS
          以上三个需要掌握的

          PROPAGATION_MANDATORY
          PROPAGATION_NESTED
          PROPAGATION_NEVER
          PROPAGATION_NOT_SUPPORTED

  3）事务提交事务，回滚事务的时机
        1.当你的业务方法，执行成功，没有异常抛出，当方法执行完毕，spring在方法执行后提交事务。事务管理器commit
        2.当你的业务方法抛出运行时异常或ERROR，spring执行回滚，调用事务管理器的rollback
       运行时异常的定义：RuntimeException和他的子类都是运行时异常，例如NullPointException, NumberFormatException
        3.当你的业务方法抛出非运行时异常，主要是受查异常时，提交事务
        受查异常：在你写代码中，必须处理的异常。例如IOException, SQLException

总结spring的事务
  1.管理事务的是 事务管理和他的实现类
  2.spring的事务是一个统一模型
     1）指定要使用的事务管理器实现类，使用<bean>
     2）指定哪些类，哪些方法需要加入事务的功能
     3）指定方法需要的隔离级别，传播行为，超时

     你需要告诉spring，你的项目中类信息，方法的名称，方法的事务传播行为。
```

1 Spring 的事务管理

事务原本是数据库中的概念，在 Dao 层。但一般情况下，需要将事务提升到业务层， 即 Service 层。这样做是为了能够使用事务的特性来管理具体的业务。

在 Spring 中通常可以通过以下两种方式来实现对事务的管理：

1. 使用 Spring 的事务注解管理事务
2. 使用 AspectJ 的 AOP 配置管理事务

2 Spring 事务管理 API

2.1 事务管理器接口

事务管理器是 PlatformTransactionManager 接口对象。其主要用于完成事务的提交、回滚，及获取事务的状态信息。

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
void	`commit(TransactionStatus status)` Commit the given transaction, with regard to its status.
TransactionStatus	`getTransaction(TransactionDefinition definition)` Return a currently active transaction or create a new one, according to the specified propagation behavior.
void	`rollback(TransactionStatus status)` Perform a rollback of the given transaction.

1. 常用的两个实现类

   PlatformTransactionManager 接口有两个常用的实现类：

   - DataSourceTransactionManager：使用 JDBC 或 MyBatis 进行数据库操作时使用。
   - HibernateTransactionManager：使用 Hibernate 进行持久化数据时使用。
2. Spring 的回滚方式

Spring 事务的默认回滚方式是：**发生运行时异常和 error 时回滚，发生检查时(编译)异常时提交。**不过，对于受查异常，程序员也可以手工设置其回滚方式。

3. 回顾错误与异常
    - Throwable 类是 Java 语言中所有错误或异常的超类。只有当对象是此类(或其子类之一)的实例时，才能通过 Java 虚拟机或者 Java 的 throw 语句抛出。
    - Error 是程序在运行过程中出现的无法处理的错误，比如 OutOfMemoryError、ThreadDeath、NoSuchMethodError 等。当这些错误发生时，程序是无法处理（捕获或抛出）的，JVM 一般会终止线程。
    - 程序在编译和运行时出现的另一类错误称之为异常，它是 JVM 通知程序员的一种方式。通过这种方式，让程序员知道已经或可能出现错误，要求程序员对其进行处理。
    - 运行时异常，是 RuntimeException 类或其子类，即只有在运行时才出现的异常。如，NullPointerException、ArrayIndexOutOfBoundsException、IllegalArgumentException 等均属于运行时异常。这些异常由 JVM 抛出，在编译时不要求必须处理（捕获或抛出）。但，只要代码编写足够仔细，程序足够健壮，运行时异常是可以避免的。
    - 受查异常，也叫编译时异常，即在代码编写时要求必须捕获或抛出的异常，若不处理，则无法通过编译。如 SQLException，ClassNotFoundException，IOException 等都属于受查异常。
    - RuntimeException 及其子类以外的异常，均属于受查异常。当然，用户自定义的 Exception 的子类，即用户自定义的异常也属受查异常。程序员在定义异常时，只要未明确声明定义的为 RuntimeException 的子类，那么定义的就是受查异常。

2.2 事务定义接口

事务定义接口 TransactionDefinition 中定义了事务描述相关的三类常量：事务隔离级别、事务传播行为、事务默认超时时限，及对它们的操作。

Field Summary

Fields

Modifier and Type	Field and Description
static int	ISOLATION_DEFAULT Use the default isolation level of the underlying datastore.
static int	ISOLATION_READ_COMMITTED Indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur.
static int	ISOLATION_READ_UNCOMMITTED Indicates that dirty reads, non-repeatable reads and phantom reads can occur.
static int	ISOLATION_REPEATABLE_READ Indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur.
static int	ISOLATION_SERIALIZABLE Indicates that dirty reads, non-repeatable reads and phantom reads are prevented.
static int	PROPAGATION_MANDATORY Support a current transaction; throw an exception if no current transaction exists.
static int	PROPAGATION_NESTED Execute within a nested transaction if a current transaction exists, behave like PROPAGATION_REQUIRED otherwise.
static int	PROPAGATION_NEVER Do not support a current transaction; throw an exception if a current transaction exists.
static int	PROPAGATION_NOT_SUPPORTED Do not support a current transaction; rather always execute non-transactionally.
static int	PROPAGATION_REQUIRED Support a current transaction; create a new one if none exists.
static int	PROPAGATION_REQUIRES_NEW Create a new transaction, suspending the current transaction if one exists.
static int	PROPAGATION_SUPPORTS Support a current transaction; execute non-transactionally if none exists.
static int	TIMEOUT_DEFAULT Use the default timeout of the underlying transaction system, or none if timeouts are not supported.

1. 定义了四个事务隔离级别常量

    这些常量均是以 ISOLATION_开头。即形如 ISOLATION_XXX。

    - DEFAULT：采用 DB 默认的事务隔离级别。MySql 的默认为 REPEATABLE_READ； Oracle 默认为 READ_COMMITTED。
    - READ_UNCOMMITTED：读未提交。未解决任何并发问题。
    - READ_COMMITTED：读已提交。解决脏读，存在不可重复读与幻读。
    - REPEATABLE_READ：可重复读。解决脏读、不可重复读，存在幻读
    - SERIALIZABLE：串行化。不存在并发问题。
2. 定义了七个事务传播行为常量

    所谓事务传播行为是指，处于不同事务中的方法在相互调用时，执行期间事务的维护情况。如，A 事务中的方法 doSome()调用 B 事务中的方法 doOther()，在调用执行期间事务的维护情况，就称为事务传播行为。

    事务传播行为是加在方法上的。 事务传播行为常量都是以 PROPAGATION_ 开头，形如 PROPAGATION_XXX。

    PROPAGATION_REQUIRED

    PROPAGATION_REQUIRES_NEW

    PROPAGATION_SUPPORTS

    PROPAGATION_MANDATORY

    PROPAGATION_NESTED

    PROPAGATION_NEVER

    PROPAGATION_NOT_SUPPORTED

- PROPAGATION_REQUIRED：

  指定的方法必须在事务内执行。若当前存在事务，就加入到当前事务中；若当前没有事务，则创建一个新事务。这种传播行为是最常见的选择，也是 Spring 默认的事务传播行为。

  如该传播行为加在 doOther()方法上。若 doSome()方法在调用 doOther() 方法时就是在事务内运行的，则 doOther()方法的执行也加入到该事务内执行。若 doSome()方法在调用 doOther() 方法时没有在事务内执行，则 doOther() 方法会创建一个事务，并在其中执行。

- PROPAGATION_SUPPORTS：

  指定的方法支持当前事务，但若当前没有事务，也可以以非事务方式执行。

- PROPAGATION_REQUIRES_NEW：

  总是新建一个事务，若当前存在事务，就将当前事务挂起，直到新事务执行完毕。

3. 定义了默认事务超时时限

   常量 TIMEOUT_DEFAULT 定义了事务底层默认的超时时限，sql 语句的执行时长。

   注意，事务的超时时限起作用的条件比较多，且超时的时间计算点较复杂。所以，该值一般就使用默认值即可。

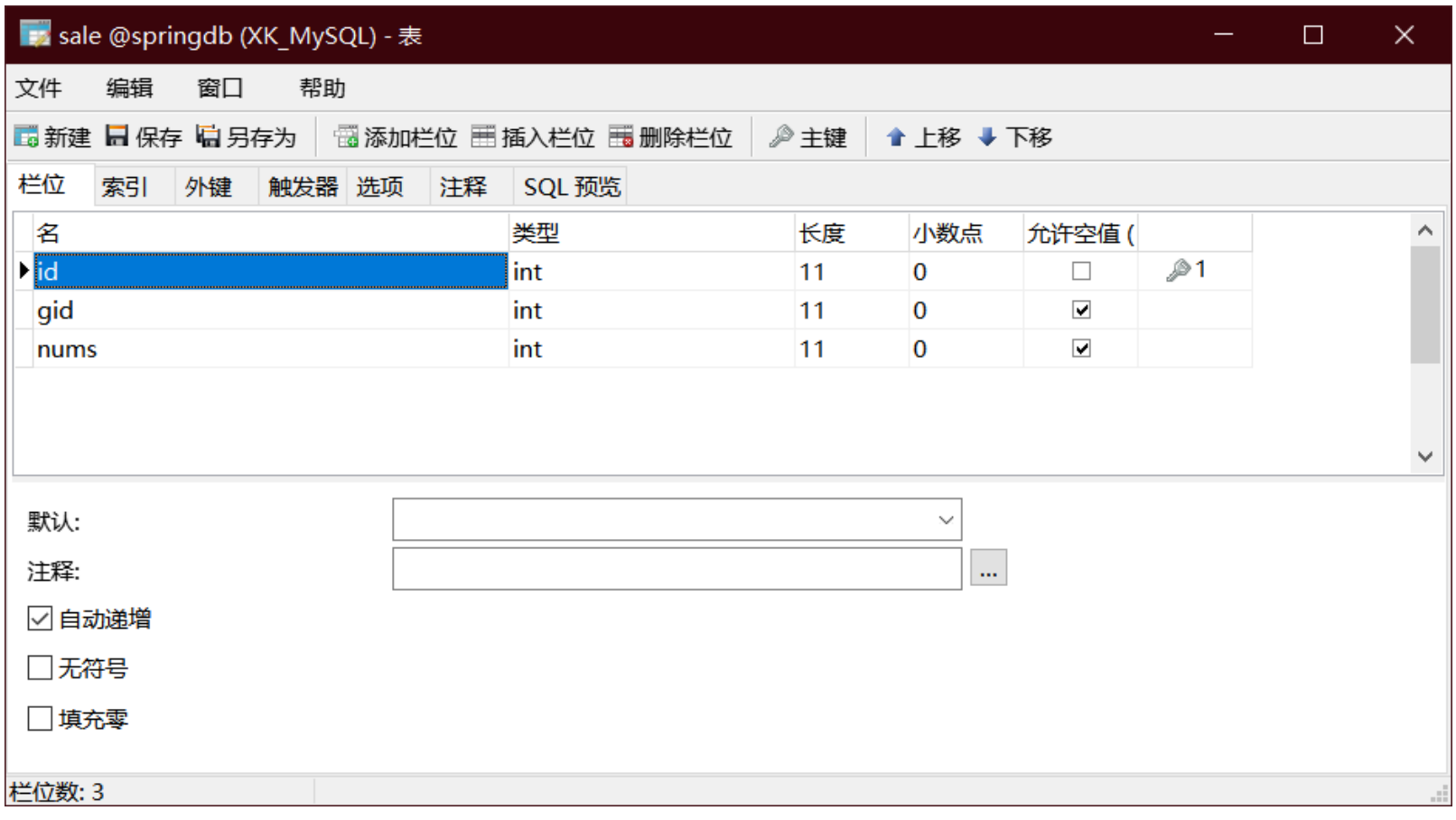
3 程序举例环境搭建

```
ch08-spring-trans: 做事务的环境项目
实现步骤：
1.新建maven项目
2.加入maven的依赖
  1）spring依赖
  2）mybatis依赖
  3）mysql驱动
  4）spring的事务的依赖
  5）mybatis和spring集成的依赖： mybatis官方体用的，用来在spring项目中创建mybatis
     的SqlSesissonFactory，dao对象的
3.创建实体类
  Sale, Goods
4.创建dao接口和mapper文件
  SaleDao接口 ，GoodsDao接口
  SaleDao.xml , GoodsDao.xml
5.创建mybatis主配置文件
6.创建Service接口和实现类，属性是saleDao, goodsDao。
7.创建spring的配置文件：声明mybatis的对象交给spring创建
 1）数据源DataSource
 2）SqlSessionFactory
 3) Dao对象
 4）声明自定义的service

8.创建测试类，获取Service对象，通过service调用dao完成数据库的访问
```

3.1 创建数据库表

创建两个数据库表 sale，goods

sale 销售表



goods 商品表

goods @springdb (XK_MySQL) - 表

文件 编辑 窗口 帮助

新建 保存 另存为 添加栏位 插入栏位 删除栏位 主键 上移 下移

栏位 索引 外键 触发器 选项 注释 SQL 预览

名	类型	长度	小数点	允许空值 (	
id	int	11	0	☐	1
name	varchar	80	0	☑	
amount	int	11	0	☑	
price	float	0	0	☑	

默认:

注释:

☐ 自动递增

☐ 无符号

☐ 填充零

栏位数: 4

goods 表数据

3.2 maven 依赖 pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.xukang</groupId>
  <artifactId>ch08-spring-trans</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <!--单元测试-->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <!--Spring核心IOC-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.2.5.RELEASE</version>
    </dependency>
    <!--做Spring事务用到的-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-tx</artifactId>
      <version>5.2.5.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
```

```
      <artifactId>spring-jdbc</artifactId>
      <version>5.2.5.RELEASE</version>
    </dependency>
    <!--mybatis依赖-->
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis</artifactId>
      <version>3.5.1</version>
    </dependency>
    <!--mybatis和spring集成的依赖-->
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis-spring</artifactId>
      <version>1.3.1</version>
    </dependency>
    <!--mysql驱动-->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.9</version>
    </dependency>
    <!--阿里公司的数据库连接池-->
    <dependency>
      <groupId>com.alibaba</groupId>
      <artifactId>druid</artifactId>
      <version>1.1.12</version>
    </dependency>
  </dependencies>

  <build>
    <resources>
      <!--目的是把src/main/java目录中的xml文件包含到输出结果中，输出到classes目录中-->
      <resource>
        <directory>src/main/java</directory><!--所在的目录-->
        <includes><!--包括目录下的.properties,.xml 文件都会扫描到-->
          <include>**/*.properties</include>
          <include>**/*.xml</include>
        </includes>
        <filtering>false</filtering>
      </resource>
    </resources>
    <plugins>
      <!--指定JDK的版本-->
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

3.3 创建实体类

```java
package com.xukang.entity;
public class Sale {
    private Integer id;
    private Integer gid;
    private Integer nums;
    //无参构造 有参构造
    //get set toString
}
```

```java
package com.xukang.entity;
public class Goods {
    private Integer id;
    private String name;
    private Integer amount;
    private Float price;
    //无参构造 有参构造
    //get set toString
}
```

3.4 定义dao接口

```
package com.xukang.dao;
import com.xukang.entity.Sale;
public interface SaleDao {
    //增加销售记录
    int insertSale(Sale sale);
}
```

```
package com.xukang.dao;
import com.xukang.entity.Goods;
public interface GoodsDao {
    //更新库存
    //goods表示本次用户购买的商品信息，id，购买的数量
    int updateGoods(Goods goods);

    //查询商品的信息
    Goods selectGoods(Integer gid);
}
```

3.5 定义 dao 接口对应的 sql 映射文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.xukang.dao.SaleDao">
    <insert id="insertSale">
        insert into sale(gid, nums) values(#{gid}, #{nums})
    </insert>
</mapper>
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.xukang.dao.GoodsDao">
    <select id="selectGoods" resultType="com.xukang.entity.Goods">
        select id, name, amount, price from goods where id = #{gid}
    </select>

    <update id="updateGoods">
        update set amount = amount - #{amount} where id = #{id}
    </update>
</mapper>
```

3.6 定义异常类

```
package com.xukang.excep;
//自定义的运行时异常
public class NotEnoughException extends RuntimeException{
    public NotEnoughException() {
    }

    public NotEnoughException(String message) {
        super(message);
    }
}
```

3.7 定义 Service 接口

```
package com.xukang.service;
public interface BuyGoodsService {
    /**
     * 购买商品的方法
     * @param goodsId 购买商品的编号
     * @param nums 购买的数量
     */
    void buy(Integer goodsId, Integer nums);
}
```

3.8 定义 service 的实现类

```java
package com.xukang.service.impl;

import com.xukang.dao.GoodsDao;
import com.xukang.dao.SaleDao;
import com.xukang.entity.Goods;
import com.xukang.entity.Sale;
import com.xukang.excep.NotEnoughException;
import com.xukang.service.BuyGoodsService;

public class BuyGoodsServiceImpl implements BuyGoodsService {
    private SaleDao saleDao;
    private GoodsDao goodsDao;
    public void setSaleDao(SaleDao saleDao) {
        this.saleDao = saleDao;
    }
    public void setGoodsDao(GoodsDao goodsDao) {
        this.goodsDao = goodsDao;
    }

    @Override
    public void buy(Integer goodsId, Integer nums) {
        System.out.println("buy方法的开始");
        //记录销售信息，向sale表添加记录
        Sale sale = new Sale(null, goodsId, nums);
        int count = saleDao.insertSale(sale);
        System.out.println("sale表添加记录条数：" + count);

        //更新库存
        Goods goods = goodsDao.selectGoods(goodsId);
        if(goods == null){
            throw new NullPointerException("编号" + goodsId + "的商品不存在。");
        }else if(goods.getAmount() < nums){
            //商品库存不足
            throw new NotEnoughException("编号" + goodsId + "的商品库存不足。");
        }
        //修改库存了
        Goods buyGoods = new Goods(goodsId, null, nums, null);
        count = goodsDao.updateGoods(buyGoods);
        System.out.println("goods表更新数目条数：" + count);
        System.out.println("buy方法的完成");
    }
}
```

3.9 修改 Spring 配置文件内容

```properties
#jdbc.properties
jdbc.url=jdbc:mysql://localhost:3306/springdb
jdbc.username=root
jdbc.password=111
jdbc.maxActive=20
```

```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
       http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd">
    <!--
       把数据库的配置信息，写在一个独立的文件，编译修改数据库的配置内容
       spring知道jdbc.properties文件的位置
    -->
    <context:property-placeholder location="classpath:jdbc.properties" />

    <!--声明数据源DataSource,作用是连接数据库-->
    <bean id="myDataSource" class="com.alibaba.druid.pool.DruidDataSource"
          init-method="init" destroy-method="close">
        <!--set注入给DuridDataSource提供连接数据库的信息-->
        <property name="url" value="${jdbc.url}" />
        <property name="username" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
        <property name="maxActive" value="${jdbc.maxActive}" /> <!--最大连接数-->
     </bean>

     <!--声明的是mybatis中提供的SqlSessionFactoryBean类，这个类内部创建SqlSessionFactory的
         SqlSessionFactory  sqlSessionFactory = new ..
     -->
```

```xml
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <!--set注入，把数据库连接池付给了dataSource属性-->
        <property name="dataSource" ref="myDataSource" />
        <!--
           mybatis主配置文件的位置
           configLocation属性是Resource类型，读取配置文件
           它的赋值，使用value，指定文件的路径，使用 classpath:表示文件的位置
        -->
        <property name="configLocation" value="classpath:mybatis_config.xml" />
    </bean>

    <!--创建dao对象，使用SqlSession的getMapper(StudentDao.class)
        MapperScannerConfigurer:在内部调用getMapper()生成每个dao接口的代理对象
    -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer" >
        <!--指定SqlSessionFactory对象的id-->
        <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />
        <!--指定包名，包名是dao接口所在的包名。
            MapperScannerConfigurer会扫描这个包中的所有接口，把每个接口都执行
            一次getMapper()方法，得到每个接口的dao对象。
            创建好的dao对象放入到spring的容器中的。 dao对象的默认名称是 接口名首字母小写
        -->
        <property name="basePackage" value="com.xukang.dao"/>
    </bean>

    <!--声明service-->
    <bean id="buyService" class="com.xukang.service.impl.BuyGoodsServiceImpl">
        <property name="goodsDao" ref="goodsDao" />
        <property name="saleDao" ref="saleDao" />
    </bean>
</beans>
```

3.10 定义测试类

```java
public class MyTest {
    @Test
    public void test01(){
        String config = "applicationcontext.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(config);
        //从容器中获取service
        BuyGoodsService service = (BuyGoodsService) ctx.getBean("buyService");
        //调用方法
        service.buy(1001, 200);
    }
}
```

4 使用 Spring 的事务注解管理事务

通过 @Transactional 注解方式，可将事务织入到相应 public 方法中，实现事务管理。

@Transactional 的所有可选属性如下所示：

- propagation：用于设置事务传播属性。该属性类型为 Propagation 枚举，默认值为 Propagation.REQUIRED
- isolation：用于设置事务的隔离级别。该属性类型为 Isolation 枚举，默认值为 Isolation.DEFAULT。
- readOnly：用于设置该方法对数据库的操作是否是只读的。该属性为 boolean，默认值为 false。
- timeout：用于设置本操作与数据库连接的超时时限。单位为秒，类型为 int，默认值为 -1，即没有时限。
- rollbackFor：指定需要回滚的异常类。类型为 Class[]，默认值为空数组。当然，若只有一个异常类时，可以不使用数组。
- rollbackForClassName：指定需要回滚的异常类类名。类型为 String[]，默认值为空数组。当然，若只有一个异常类时，可以不使用数组。
- noRollbackFor：指定不需要回滚的异常类。类型为 Class[]，默认值为空数组。当然，若只有一个异常类时，可以不使用数组。

需要注意的是，@Transactional 若用在方法上，只能用于 public 方法上。对于其他非 public 方法，如果加上了注解@Transactional，虽然 Spring 不会报错，但不会将指定事务织入到该方法中。因为 Spring 会忽略掉所有非 public 方法上的@Transaction 注解。

若@Transaction 注解在类上，则表示该类上所有的方法均将在执行时织入事务。

实现注解的事务步骤：

1. 声明事务管理器对象

```xml
    <!--使用Spring的事务管理-->
    <!--1.声明事务管理器-->
    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager" >
        <!--连接的数据库，指定数据源-->
        <property name="dataSource" ref="myDataSource" />
    </bean>
```

2. 开启事务注解驱动

```
告诉spring框架，我要使用注解的方式管理事务。
spring使用aop机制，创建@Transactional所在的类代理对象，给方法加入事务的功能。
spring给业务方法加入事务：
在你的业务方法执行之前，先开启事务，在业务方法之后提交或回滚事务，使用aop的环绕通知
@Around("你要增加的事务功能的业务方法名称")
Object myAround(){
    开启事务，spring给你开启
    try{
        buy(1001,10);
        spring的事务管理器.commit();
    }catch(Exception e){
         spring的事务管理器.rollback();
    }
}
```

```
<!--2.开启事务注解驱动，告诉Spring使用注解管理事务，创建代理对象
        transactionManager: 事务管理器对象的id
-->
<tx:annotation-driven transaction-manager="transactionManager" />
```

3. 业务层 public 方法加入事务属性

```
public class BuyGoodsServiceImpl implements BuyGoodsService {
    private SaleDao saleDao;
    private GoodsDao goodsDao;
    public void setSaleDao(SaleDao saleDao) {
        this.saleDao = saleDao;
    }
    public void setGoodsDao(GoodsDao goodsDao) {
        this.goodsDao = goodsDao;
    }

    /*
     * rollbackFor:表示发生指定的异常一定回滚
     *    处理逻辑是：
     *      1) spring框架会首先检查方法抛出的异常是不是在rollbackFor的属性值中
     *           如果异常在rollbackFor列表中，不管是什么类型的异常，一定回滚。
     *      2) 如果你的抛出的异常不在rollbackFor列表中，spring会判断异常是不是RuntimeException,
     *           如果是一定回滚。
     */
    /*@Transactional(
            propagation = Propagation.REQUIRED,
            isolation = Isolation.DEFAULT,
            readOnly = false,
            rollbackFor = {
                    NullPointerException.class,
                    NotEnoughException.class
            }
    )*/

    //使用的是事务控制的默认值， 默认的传播行为是REQUIRED，默认的隔离级别DEFAULT
    //默认抛出运行时异常，回滚事务。
    @Transactional
    @Override
    public void buy(Integer goodsId, Integer nums) {
        System.out.println("buy方法的开始");
        //记录销售信息，向sale表添加记录
        Sale sale = new Sale(null, goodsId, nums);
        int count = saleDao.insertSale(sale);
        System.out.println("sale表添加记录条数：" + count);

        //更新库存
        Goods goods = goodsDao.selectGoods(goodsId);
        if(goods == null){
            throw new NullPointerException("编号" + goodsId + "的商品不存在。");
        }else if(goods.getAmount() < nums){
            //商品库存不足
            throw new NotEnoughException("编号" + goodsId + "的商品库存不足。");
        }
        //修改库存了
        Goods buyGoods = new Goods(goodsId, null, nums, null);
        count = goodsDao.updateGoods(buyGoods);
        System.out.println("goods表更新数目条数：" + count);
        System.out.println("buy方法的完成");
    }
}
```

5 使用 AspectJ 的 AOP 配置管理事务

```
适合大型项目，有很多的类，方法，需要大量的配置事务，使用aspectj框架功能，在spring配置文件中
声明类，方法需要的事务。这种方式业务方法和事务配置完全分离。
```

使用 XML 配置事务代理的方式的不足是，每个目标类都需要配置事务代理。当目标类较多，配置文件会变得非常臃肿。

使用 XML 配置顾问方式可以自动为每个符合切入点表达式的类生成事务代理。其用法很简单，只需将前面代码中关于事务代理的配置删除，再替换为如下内容即可。

1. maven 依赖 pom.xml

   新加入 aspectj 的依赖坐标

```xml
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>5.2.5.RELEASE</version>
</dependency>
```

2. 在容器中添加事务管理器

```xml
    <!--声明事务管理器-->
    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager" >
        <!--连接的数据库，指定数据源-->
        <property name="dataSource" ref="myDataSource" />
    </bean>
```

3. 配置事务通知

   为事务通知设置相关属性。用于指定要将事务以什么方式织入给哪些方法。

   例如，应用到 buy 方法上的事务要求是必须的，且当 buy 方法发生异常后要回滚业务。

```xml
<!--声明业务方法它的事务属性（隔离级别，传播行为，超时时间）
        id: 自定义名称，表示 <tx:advice> 和 </tx:advice> 之间的配置内容的
        transaction-manager:事务管理器对象的id
    -->
    <tx:advice id="myAdvice" transaction-manager="transactionManager" >
        <!--tx:attributes:表示配置事务的属性-->
        <tx:attributes>
            <!--tx:method:给具体的方法配置事务属性，method可以有多个，分别给不同的方法设置事务属性
                name: 方法名称，1）完整的方法名称，不带有包和类
                                2）方法可以使用通配符，*表示任意字符
                propagation: 传播行为，枚举型
                isolation: 隔离级别
                rollback-for: 你指定的异常类名，全限定名称，发生异常一定回滚
            -->
            <tx:method name="buy" propagation="REQUIRED" isolation="DEFAULT"
                       rollback-
for="java.lang.NullPointerException,com.xukang.excep.NotEnoughException"/>

            <!--使用通配符，指定很多的方法-->
            <tx:method name="add*" propagation="REQUIRES_NEW" />
            <!--指定修改方法-->
            <tx:method name="modify*" />
            <!--删除方法-->
            <tx:method name="remove*" />
            <!--查询方法，query，search，find-->
            <tx:method name="*" propagation="SUPPORTS" read-only="true" />
        </tx:attributes>
    </tx:advice>
```

4. 配置增强器

   指定将配置好的事务通知，织入给谁。

```xml
    <!--配置aop-->
    <aop:config>
        <!--
            配置切入点表达式：指定哪些包中类，要使用事务
            id: 切入点表达式的名称，唯一值
            expression: 切入点表达式，指定哪些类要使用事务，aspectj会创建代理对象
        -->
        <aop:pointcut id="servicePt" expression="execution(* *..service..*.*(..))"/>

        <!--配置增强器：关联 advice 和 pointcut
            advice-ref:通知，上面tx:advice 哪里的配置
            pointcut-ref:切入点表达式的id
        -->
        <aop:advisor advice-ref="myAdvice" pointcut-ref="servicePt" />
    </aop:config>
```

5. 修改测试类

```
public class MyTest {
    @Test
    public void test01(){
        String config = "applicationcontext.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(config);
        //从容器中获取service
        BuyGoodsService service = (BuyGoodsService) ctx.getBean("buyService");
        System.out.println("service是代理: " + service.getClass().getName());
        //调用方法
        service.buy(1001, 10);
    }
}
```

六、Spring 与 Web

```
web项目中怎么使用容器对象。
1.做的是javase项目有main方法的，执行代码是执行main方法的，
  在main里面创建的容器对象
  ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
2.web项目是在tomcat服务器上运行的。tomcat一起动，项目一直运行的。
```

在 Web 项目中使用 Spring 框架，首先要解决在 web 层（这里指 Servlet）中获取到 Spring 容器的问题。只要在 web 层获取到了 Spring 容器，便可从容器中获取到 Service 对象。

1 Web 项目使用 Spring

```
在web项目中使用spring，完成学生注册功能
实现步骤：
1.创建maven，web项目
2.加入依赖
  拷贝ch7-spring-mybatis中依赖
  jsp，servlet依赖
3.拷贝ch7-spring-mybatis的代码和配置文件
4.创建一个jsp发起请求，有参数id,name,email,age
5.创建Servlet，接收请求参数，调用 Service，调用dao完成注册
6.创建一个jsp作为显示结果页面
```

1. 新建一个 Maven Project

   类型 maven-archetype-webapp

2. 复制代码，配置文件，jar

   将 spring-mybatis 项目中以下内容复制到当前项目中：

   - Service 层、Dao 层全部代码
   - 配置文件 applicationContext.xml 及 jdbc.properties，mybatis.xml
   - pom.xml
   - 加入 servlet，jsp 依赖

```
    <!-- servlet依赖 -->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.1.0</version>
      <scope>provided</scope>
    </dependency>
    <!-- jsp依赖 -->
    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>jsp-api</artifactId>
      <version>2.2.1-b03</version>
      <scope>provided</scope>
    </dependency>
```

3. 定义 index 页面

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <p>注册学生</p>
    <form action="reg" method="post" >
        <table>
            <tr>
                <td>id:</td>
```

```
                <td><input type="text" name="id"></td>
            </tr>
            <tr>
                <td>姓名:</td>
                <td><input type="text" name="name"></td>
            </tr>
            <tr>
                <td>email:</td>
                <td><input type="text" name="email"></td>
            </tr>
            <tr>
                <td>年龄:</td>
                <td><input type="text" name="age"></td>
            </tr>
            <tr>
                <td></td>
                <td><input type="submit" value="注册学生"></td>
            </tr>
        </table>
    </form>
</body>
</html>
```

4. 定义 RegisterServlet（重点代码）

```
package org.example.controller;

import org.example.domain.Student;
import org.example.service.StudentService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.IOException;

public class RegisterServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        request.setCharacterEncoding("utf-8");
        String strId = request.getParameter("id");
        String strName = request.getParameter("name");
        String strEmail = request.getParameter("email");
        String strAge = request.getParameter("age");
        //创建spring的容器对象
        String config = "applicationcontext.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(config);
        System.out.println("容器信息===" + ctx);
        //获取service
        StudentService studentService = (StudentService) ctx.getBean("studentService");
        Student student = new Student();
        student.setAge(Integer.valueOf(strId));
        student.setName(strName);
        student.setEmail(strEmail);
        student.setAge(Integer.valueOf(strAge));
        int count = studentService.addStudent(student);

        //给一个结果页面
        if(count == 1){
            request.getRequestDispatcher("/success.jsp").forward(request, response);
        }else{
            request.getRequestDispatcher("/fail.jsp").forward(request, response);
        }
    }
}
```

5. 定义 success / fail 页面

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
result.jsp 注册成功
</body>
</html>
```

```jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
fail.jsp 注册失败
</body>
</html>
```

6. web.xml 注册 Servlet

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">

    <servlet>
        <servlet-name>RegisterServlet</servlet-name>
        <servlet-class>org.example.controller.RegisterServlet</servlet-class>
    </servlet>
    <!--别名-->
    <servlet-mapping>
        <servlet-name>RegisterServlet</servlet-name>
        <url-pattern>/reg</url-pattern>
    </servlet-mapping>
</web-app>
```

7. 运行结果分析

   当表单提交，跳转到 success.jsp 后，多刷新几次页面，查看后台输出，发现每刷新一次页面，就 new 出一个新的 Spring 容器。即，每提交一次请求，就会创建一个新的 Spring 容 器。对于一个应用来说，只需要一个 Spring 容器即可。所以，将 Spring 容器的创建语句放 在 Servlet 的 doGet()或 doPost()方法中是有问题的。

   此时，可以考虑，将 Spring 容器的创建放在 Servlet 进行初始化时进行，即执行 init() 方法时执行。并且，Servlet 还是单例多线程的，即一个业务只有一个 Servlet 实例，所有执行 该业务的用户执行的都是这一个 Servlet 实例。这样，Spring 容器就具有了唯一性了。

   但是，Servlet 是一个业务一个 Servlet 实例，即 LoginServlet 只有一个，但还会有 StudentServlet、TeacherServlet 等。每个业务都会有一个 Servlet，都会执行自己的 init()方法， 也就都会创建一个 Spring 容器了。这样一来，Spring 容器就又不唯一了。

2 使用 Spring 的监听器 ContextLoaderListener

```
需求：
web项目中容器对象只需要创建一次，把容器对象放入到全局作用域ServletContext中。
怎么实现：
    使用监听器 当全局作用域对象被创建时 创建容器 存入ServletContext
     监听器作用：
     1）创建容器对象，执行 ApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
     2）把容器对象放入到ServletContext， ServletContext.setAttribute(key,ctx)
     监听器可以自己创建，也可以使用框架中提供好的ContextLoaderListener

     private WebApplicationContext context;
     public interface WebApplicationContext extends ApplicationContext

     ApplicationContext:javase项目中使用的容器对象
     WebApplicationContext：web项目中的使用的容器对象

    把创建的容器对象，放入到全局作用域
     key： WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE
           WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE
     value：this.context

     servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE,
this.context);
```

对于 Web 应用来说，ServletContext 对象是唯一的，一个 Web 应用，只有一个 ServletContext 对象，该对象是在 Web 应用装载时初始化的。若将 Spring 容器的创建时机， 放在 ServletContext 初始化时，就可以保证 Spring 容器的创建只会执行一次，也就保证了 Spring 容器在整个应用中的唯一性。

当 Spring 容器创建好后，在整个应用的生命周期过程中，Spring 容器应该是随时可以被访问的。即，Spring 容器应具有全局性。而放入 ServletContext 对象的属性，就具有应用的全局性。所以，将创建好的 Spring 容器，以属性的形式放入到 ServletContext 的空间中，就 保证了 Spring 容器的全局性。

上述的这些工作，已经被封装在了如下的 Spring 的 Jar 包的相关 API 中：spring-web-5.2.5.RELEASE

1. maven 依赖 pom.xml

```xml
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.2.5.RELEASE</version>
</dependency>
```

2. 注册监听器 ContextLoaderListener

若要在 ServletContext 初始化时创建 Spring 容 器 ， 就需要使用监听器接口 ServletContextListener 对 ServletContext 进行监听。在 web.xml 中注册该监听器。

```xml
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
```

Spring 为该监听器接口定义了一个实现类 ContextLoaderListener，完成了两个很重要的 工作：创建容器对象，并将容器对象放入到了 ServletContext 的空间中。

打开 ContextLoaderListener 的源码。看到一共四个方法，两个是构造方法，一个初始化方法，一个销毁方法。

```java
/.../

package org.springframework.web.context;

import ...

public class ContextLoaderListener extends ContextLoader implements ServletContextListener {
    public ContextLoaderListener() {
    }

    public ContextLoaderListener(WebApplicationContext context) { super(context); }

    public void contextInitialized(ServletContextEvent event) {
        this.initWebApplicationContext(event.getServletContext());
    }

    public void contextDestroyed(ServletContextEvent event) {
        this.closeWebApplicationContext(event.getServletContext());
        ContextCleanupListener.cleanupAttributes(event.getServletContext());
    }
}
```

所以，在这四个方法中较重要的方法应该就是 contextInitialized()，context 初始化方法。

跟踪 initWebApplicationContext()方法，可以看到，在其中创建了容器对象。

```java
try {
    if (this.context == null) {
        this.context = this.createWebApplicationContext(servletContext);
    }
```

并且，将创建好的容器对象放入到了 ServletContext 的空间中，key 为一个常量：WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE。

```java
servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, this.context);
```

3. 指定 Spring 配置文件的位置 spring-param

ContextLoaderListener 在对 Spring 容器进行创建时，需要加载 Spring 配置文件。其默认 的 Spring 配置文件位置与名称为：WEB-INF/applicationContext.xml。但，一般会将该配置文件放置于项目的 classpath 下，即 src 下，所以需要在 web.xml 中对 Spring 配置文件的位置及名称进行指定。

```xml
    <context-param>
        <!-- contextConfigLocation:表示配置文件的路径  -->
        <param-name>contextConfigLocation</param-name>
        <!--自定义配置文件的路径-->
        <param-value>classpath:applicationcontext.xml</param-value>
    </context-param>
```

从监听器 ContextLoaderListener 的父类 ContextLoader 的源码中可以看到其要读取的配置文件位置参数名称 contextConfigLocation。

```
public class ContextLoader {
    public static final String CONTEXT_ID_PARAM = "contextId";
    public static final String CONFIG_LOCATION_PARAM = "contextConfigLocation";
    public static final String CONTEXT_CLASS_PARAM = "contextClass";
```

4. 获取 Spring 容器对象
   - 直接从 ServletContext 中获取

     从对监听器 ContextLoaderListener 的源码分析可知，容器对象在 ServletContext 的中存 放的 key 为 WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE。所以，可 以直接通过 ServletContext 的 getAttribute()方法，按照指定的 key 将容器对象获取到。

```
    //获取ServletContext中的容器对象，创建好的容器对象，拿来就用
        String key = WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE;
        Object attr = getServletContext().getAttribute(key);
        if(attr != null){
            ctx = (WebApplicationContext) attr;
        }
```

   - 通过 WebApplicationContextUtils 工具类获取

     工具类 WebApplicationContextUtils 有一个方法专门用于从 ServletContext 中获取 Spring 容器对象：getRequiredWebApplicationContext(ServletContext sc)

```
    //使用框架中的方法，获取容器对象
    ServletContext sc = getServletContext();
    WebApplicationContext ctx = WebApplicationContextUtils.getRequiredWebApplicationContext(sc);
```

以上两种方式，无论使用哪种获取容器对象，刷新 success 页面后，可看到代码中使用的 Spring 容器均为同一个对象