day41_Spring 第一天

学习目标

- [ ] 掌握Spring是什么
- [ ] 了解工厂解耦
- [ ] 掌握IOC - 控制反转： 把对象的创建工作交给Spring来做。
- [ ] 掌握DI 依赖注入（属性赋值）
- [ ] 掌握Spring整合Junit

一、简介

1. 什么是Spring

- Spring 是分层的Java SE/EE应用 full-stack(服务端的全栈)轻量级（跟EJB比）开源框架，以 IoC(Inversion of Control控制反转，目的是解耦)和AOP(面向切面编程，本质是动态代理，目的是增强)为内核
  - Spring家族有很多的框架，涉及到所有层（web |service |dao）
  - 今天学的Spring仅仅是Spring家族里面的其中一个框架 Spring Framework (IOC + AOP)
- 提供了:
  - 表现层Spring MVC
  - 持久层Spring JDBCTemplate， Spring Data JPA
  - 业务层事务管理等
- 能整合开源世界众多著名的第三方框架和类库，逐渐成为使用最多的Java EE企业应用开源框架。

2. Spring的发展历程

- 1997年，IBM提出了EJB的思想
- 1998年，SUN制定开发标准规范EJB1.0
- 1999年，EJB1.1发布
  2001年，EJB2.0发布
  2003年，EJB2.1发布
  2006年，EJB3.0发布
- Rod Johnson（spring之父）
  - Expert One-to-One J2EE Design and Development(2002)
    阐述了J2EE使用EJB开发设计的优点及解决方案
  - Expert One-to-One J2EE Development without EJB(2004)
    阐述了J2EE开发不使用EJB的解决方式（Spring雏形）
- 2017年9月份发布了spring的最新版本spring 5.0通用版（GA）

3. Spring的优势

- 方便解耦，简化开发（IoC思想，第1、2天）

  通过Spring提供的IoC容器，可以将对象间的依赖关系交由Spring进行控制，避免硬编码所造成的过度程序耦合。用户也不必再为单例模式类、属性文件解析等这些很底层的需求编写代码，可以更专注于上层的应用。 创建对象更简单!

- AOP编程的支持（第3天）

  通过Spring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。

  比如：要求面面项目里，每个方法被调用时，都输出日志到控制台“2020-03-20 11:20:31执行了xxx.xx方法”

- 声明式事务的支持（第3天）

  可以将我们从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活的进行事务的管理，提高开发效率和质量。@Transactional

- 方便程序的测试（第1天）

  可以用非容器依赖的编程方式进行几乎所有的测试工作，测试不再是昂贵的操作，而是随手可做的事情。

  例如：Spring整合了Junit

- 方便集成各种优秀框架（SSM整合，SpringMVC第2天）

  Spring可以降低各种框架的使用难度，提供了对各种优秀框架（Struts、Hibernate、Hessian、Quartz等）的直接支持。

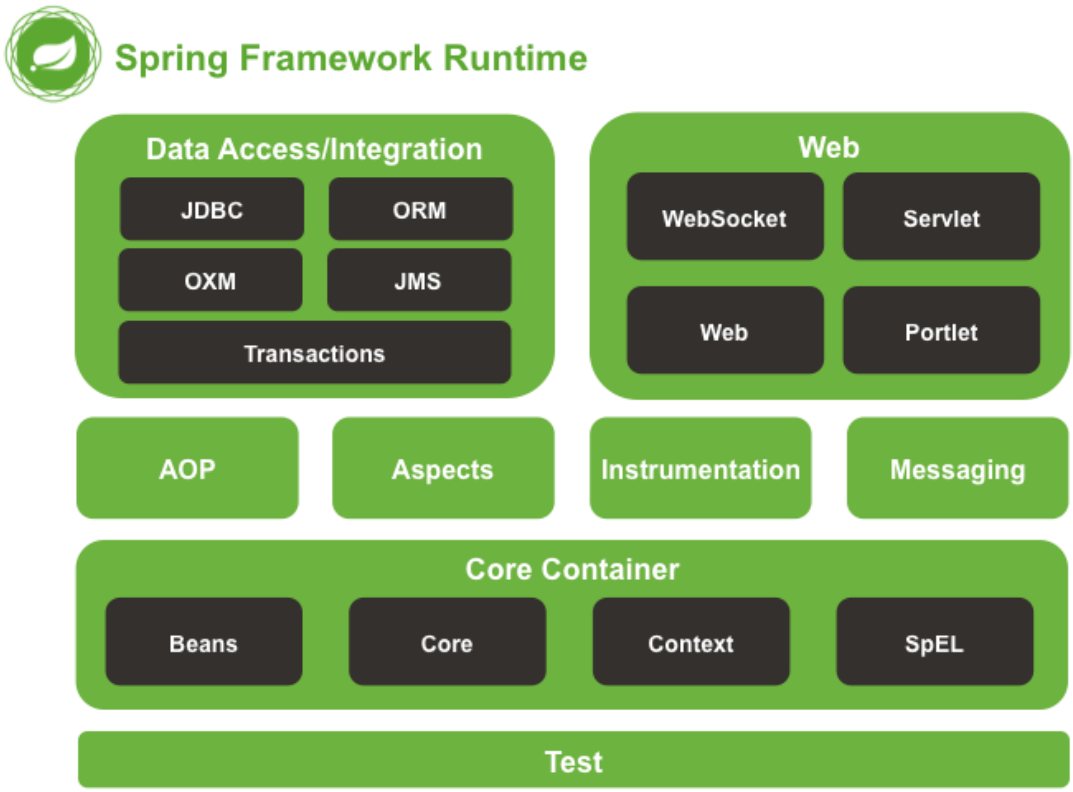
- 降低JavaEE API的使用难度(第3天)

  Spring对JavaEE API（如JDBC、JavaMail、远程调用等）进行了薄薄的封装层，使这些API的使用难度大为降低。

- Java源码是经典学习范例

  Spring的源代码设计精妙、结构清晰、匠心独用，处处体现着大师对Java设计模式灵活运用以及对Java技术的高深造诣。它的源代码无疑是Java技术的最佳实践的范例。

4. Spring的体系结构



二、工厂模式解耦（理解）

1. 耦合性问题

- 耦合性：程序之间（代码间）的依赖性。
  - 编译期依赖：**编译时必须提供依赖的类，否则编译不通过。**
    - Dao | Service：UserService us = new UserService(); // 只有单一独立的类
    - Dao | Service： UserService us02 = new UserServiceImpl(); // 有接口，也有实现类。
  - 运行期依赖：运行时必须提供依赖的类，否则不能运行。
    - 接口和实现的写法：

      UserService us02 =
      Class.forName("com.itheima.service.impl.UserServiceImpl").newInstance();
  - 应当减少编译期依赖，使用运行期依赖
- 耦合性越强，维护成本（时间成本&精力成本）就越高
- 开发时要求：高内聚，低耦合
  - 低耦合： 耦合度很低，代码与代码之间耦合度很低。
  - 高内聚： 把具有一样功能的代码，尽可能靠拢起来。 一个业务有很多的方法，这些方法要尽可能靠在一块。方便管理，维护。

    类与类之间的内聚
    - 注册 ---- RegisterServlet
    - 登录 --- LoginServlet
    - 更新用户 --- UpdateUserServlet
    - 用户 ----- UserServlet

    方法与方法之间的内聚

    每个方法里面都有乱码解决..两句话 ---- 过滤器。。

    req.setCharacterEncoding();

    resp.setContentType();

1.1 耦合性问题现象

- 在web开发中，服务端通常分为三层：web层、service层、dao层
  - web层调用service层完成功能：需要new一个Service对象
    - 以前的写法，直接new对象
      - UserService userService = new UserService();
    - 真正开发的时候是面向接口编程。
      - UserService userService = new UserServiceImpl();
  - service层调用dao层操作数据库：需要new一个dao对象
    - 以前的写法， 直接new对象
    - 真正开发的时候是面向接口编程。
      - UerDao userDao = new UserDaoImpl02();
- 三层之间的耦合性比较强：存在编译期依赖
  - service里写死了创建某一个dao对象：一旦dao对象换了，就需要修改service的源码
  - web里写死了创建某一个service对象：一旦service对象换了，就需要修改web的源码

1.2 解耦的思路

- 可以使用反射技术，代替`new`创建对象，**避免编译期依赖**

```
Class clazz = Class.forName("全限定类名");
Object obj = clazz.newInstance();
```

- 再把全限定类名提取到配置文件中（xml或properties），读取配置文件信息来反射创建对象

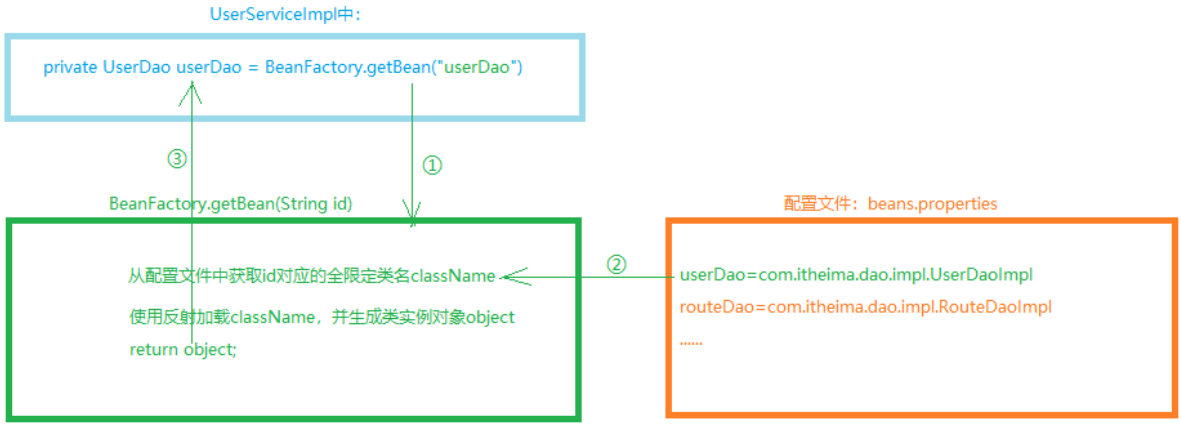
```
//读取配置文件，得到要创建对象的全限定类名；再通过反射技术创建对象
String className = ...;
Class clazz = Class.forName(className);
Object obj = clazz.newInstance();
```

2. 使用工厂模式解耦

需求描述

- UserService调用UserDao，在每一层的方法论里面都打印一句日志即可
- 使用工厂模式+配置文件的方式，降低它们之间的耦合性

需求分析



需求实现

1. 创建项目，导入依赖

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

2. dao层代码

```java
package com.itheima.dao;

public interface UserDao {
    void add();
}
```

```java
package com.itheima.dao.impl;

import com.itheima.dao.UserDao;

public class UserDaoImpl implements UserDao {
    public void add() {
        System.out.println("调用了UserDaoImpl的add方法~！~");
    }
}
```

3. service层代码

```java
package com.itheima.service;

public interface UserService {
    void add() throws Exception;
}
```

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

import com.itheima.dao.UserDao;
import com.itheima.dao.impl.UserDaoImpl;
import com.itheima.factory.BeanFactory;
import com.itheima.service.UserService;

public class UserServiceImpl implements UserService {
    public void add() throws Exception {
        System.out.println("执行了UserServiceImpl的add方法！~");

        //调用dao
        //以前：   UserDao userDao = new UserDao();  userDao.add();

        //现在有接口：
        //UserDao  userDao = new UserDaoImpl();
        //userDao.add();

        //使用工厂解耦
        UserDao userDao = (UserDao) BeanFactory.getBean("ud");
        userDao.add();

    }
}
```

4. 配置文件`beans.properties`

```
# 在这个文件里面，登记要工厂创建的对象的别名和类的全路径
us=com.itheima.service.impl.UserServiceImpl
ud=com.itheima.dao.impl.UserDaoImpl
```

5. 工厂类`BeanFactory`

```
package com.itheima.factory;

import java.util.Enumeration;
import java.util.HashMap;
import java.util.Map;
import java.util.ResourceBundle;

/*
    这是一个用来创建对象的工厂，要想让这个工厂帮忙创建对象，只需要在beans.properties里面登记别名和类的全路径名即可。
    工厂的代码一旦写就，以后可能不会修改了。

    步骤：
        1. 提供静态代码块
            1.1 读取resources/beans.properties
            1.2 把里面的key和value 读取出来之后，存到一个map集合里面去

        2. 提供一个静态方法，供外面的人来获取对象
            2.1 方法需要传递进来别名，这样工厂就能够从map集合中取出对应的全路径，然后创建对象返回了。
            2.2 要注意判空。
 */
public class BeanFactory {

    //0. 定义一个map集合
    static Map<String , String> map = new HashMap<String , String>();

    static{
        //1. 读取beans.properties  这个方法是java工具包提供的，允许我们从resources文件夹下读取文件，只要写文件的名字即可
        ResourceBundle beans = ResourceBundle.getBundle("beans");

        //2. 得到所有的key，返回的是要给集合，这个集合类似迭代器
        Enumeration<String> keys = beans.getKeys();

        //3. 遍历集合，得到每一个key
        while(keys.hasMoreElements()){

            //4. 取出来key和value
            String key = keys.nextElement();
            String value = beans.getString(key);

            //5. 把这些key和value装到map集合中
            map.put(key , value);
        }
    }
    /**
     * 用于问工厂要对象
     * @param name  别名
     * @return 对象
```

```
     */
    public static Object getBean(String name) throws Exception {
        System.out.println("来工厂的getBean方法要对象了~! ~! ~! " + name);

        //1. 得到map集合中保存的类的全路径 ， 也要注意，可能map集合里面没有这个别名的全路径
        String value = map.get(name);

        //2. 如果取出来不是null，则表示map集合中有这个全路径
        if(value !=null ){
            return Class.forName(value).newInstance();
        }

        //3. 如果没有就返回null
        return null;
    }
}
```

- 测试

```
package com.itheima.test;

import com.itheima.factory.BeanFactory;
import com.itheima.service.UserService;
import com.itheima.service.impl.UserServiceImpl;
import org.junit.Test;

public class TestUserServiceImpl {

    @Test
    public void testAdd() throws Exception {

        //创建对象:
        //UserService us = new UserServiceImpl();
        //us.add();

        //使用工厂来解耦
        UserService us = (UserService) BeanFactory.getBean("us");
        us.add();
    }
}
```

小结

1. 首先得有接口和实现类： UserDao 和 UserDaoImpl ， UserService 和 UserServiceImpl
2. 使用properties配置文件来记录，别名和实现类的全路径
3. 定义一个工厂类
    1. 在静态代码块里面读取配置文件，使用map集合来存映射关系
    2. 定义一个静态方法，只要有人来获取实例，那么就从map集合里面取出来全路径
    3. 使用反射技术来构建实例返回。

三、控制反转IOC【重点】

- 什么是IOC

> 控制反转，把对象的创建工作交给框架（工厂 Spring），我们不需要自己去new这个对象，只管问工厂要。由原来的主动创建对象，变成自己被动接收 框架创建的对象。

- IOC的作用

> IOC是Spring的核心之一，作用就是为了解耦，降低程序，代码间的耦合度。

1. 快速入门【重点】

需求描述

- 有`UserDao`接口和`UserDaoImpl`实现类
- 通过Spring容器（工厂）得到`UserDaoImpl`的实例对象（IoC方式）

开发步骤

1. 创建Maven项目，导入依赖坐标：Spring的依赖坐标
2. 编写dao接口`UserDao`及实现`UserDaoImpl`
3. 创建Spring核心配置文件，并配置`UserDaoImpl` （作用类似bean.properties）
4. 测试：使用Spring的API，获取Bean实例对象

需求实现

1. 创建Maven项目，导入依赖坐标

```
<dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.2.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
```

2. 编写dao接口`UserDao`及实现`UserDaoImpl`

- 接口`UserDao`

```
package com.itheima.dao;

public interface UserDao {
    void add();
}
```

- 实现类`UserDaoImpl`

```
package com.itheima.dao.impl;

import com.itheima.dao.UserDao;

public class UserDaoImpl implements UserDao {

    public void add() {
        System.out.println("执行了UserDaoImpl的add方法~! ~");
    }

}
```

3. 创建Spring核心配置文件，并配置`UserDaoImpl`

> 这个步骤的作用就是告诉spring，要创建哪个类的对象，并且给这个类起一个别名，方便以后我们问spring要对象。

- 配置文件名称，通常叫`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">

    <!--托管UserDaoImpl，其实就是告诉spring，要创建这个类的对象，
            并且给这个类起别名，以便以后来找到这个对象-->
    <!--
    bean:
        作用： 把具体的一个类交给spring来托管，默认会执行这个类的无参构造函数
        属性：
            id : 给这个类起别名 ，是唯一标识。
            class :  类的全路径地址
        -->
     <bean id="ud" class="com.itheima.dao.impl.UserDaoImpl"></bean>
</beans>
```

4. 使用Spring的API，获取Bean实例对象

- 编写测试类

```
package com.itheima.test;

import com.itheima.dao.UserDao;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;



public class TestUserServiceImpl {

    @Test
    public void testAdd(){
        /*
```

```
            抓住关键字，顺藤摸瓜，自己推导出来代码。
         */
        //1. 创建工厂
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

        //2. 问工厂要对象
        UserDao userDao = (UserDao) context.getBean("ud");


        //3. 调用方法
        userDao.add();


        //销毁工厂(一般不销毁工厂)
        context.close();
    }
}
```

小结

1. 首先编写UserDao 和 UserDaoImpl
2. 在pom.xml里面添加依赖
3. 在resources下面，创建一个xml文件，名字随意。不要手动创建文件的方式。要选择xml配置文件的方式
4. 在xml文件里面登记|注册实现类
5. 问工厂要实例

2. 配置文件详解【了解】

1. `bean`标签的基本配置

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

1. 介绍
    - 用于配置：把对象交给Spring进行控制
    - 默认情况下，Spring是调用类的无参构造来创建对象的；如果没有无参构造，则不能创建成功
2. 基本属性
    - `id`：唯一标识
    - `class`：bean的全限定类名

> 了解：bean的id和name的区别
>
> 1. 一个bean只能有一个id；一个bean可以有多个name
> 2. bean的name值：多个name之间以`,` `;` `空格` 隔开，第1个name作为id，其它作为别名

2. `bean`标签的作用范围配置

```
<bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl" scope="singleton">
</bean>
```

- scope属性取值如下：

取值	说明
`singleton`	默认，表示单例的，一个Spring容器里，只有一个该bean对象
`prototype`	多例的，一个Spring容器里，有多个该bean对象
`request`	web项目里，Spring创建的bean对象将放到`request`域中：一次请求期间有效
`session`	web项目里，Spring创建的bean对象将放到`session`域中：一次会话期间有效
`globalSession`	web项目里，应用在Portlet环境/集群环境；如果没有Portlet/集群环境，那么globalSession相当于session（新版本中已删除）

- 不同scope的bean，生命周期：
  - singleton：bean的生命周期和Spring容器的生命周期相同
    - 整个Spring容器中，只有一个bean对象
    - 何时创建：加载Spring配置文件，初始化Spring容器时，bean对象创建
    - 何时销毁：Spring容器销毁时，bean对象销毁
  - prototype：bean的生命周期和Spring容器无关。Spring创建bean对象之后，交给JVM管理了
    - 整个Spring容器中，会创建多个bean对象，创建之后由JVM管理
    - 何时创建：调用`getBean`方法获取bean对象时，bean对象创建
    - 何时销毁：对象长时间不用时，垃圾回收

3. `bean`生命周期相关方法的配置【了解】

```
<bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"
      init-method="" destroy-method=""></bean>
```

- `init-method`：指定类中初始化方法名称，该方法将在bean对象被创建时执行
- `destroy-method`：指定类中销毁方法名称，该方法将在bean对象被销毁时执行

> 注意：
>
> - prototype类型的bean：Spring容器销毁时，也不会执行销毁方法，因为Spring不负责它的销毁
> - singleton类型的bean：在Spring容器显式关闭时，会执行destroy-method指定的方法

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

UserDao userDao = context.getBean("userDao", UserDao.class);
//显式的关闭Spring容器
((ClassPathXmlApplicationContext)context).close();
```

- service

```
package com.itheima.dao.impl;

import com.itheima.dao.UserDao;

public class UserDaoImpl implements UserDao {

    public UserDaoImpl(){
        System.out.println("调用了UserDaoImpl的无参构造~! ");
    }

    public void add() {
        System.out.println("执行了UserDaoImpl的add方法~! ~");
    }

    public void init(){
        System.out.println("调用了UserDaoImpl的init方法~! ");
    }
    public void destroy(){
        System.out.println("调用了UserDaoImpl的destroy方法~! ");
    }
}
```

- xml配置

```
    <bean id="ud" class="com.itheima.dao.impl.UserDaoImpl" scope="singleton"
          init-method="init" destroy-method="destroy"/>
```

4. bean实例化的三种方式【了解】

> 我们通常都是问Spring要对象，那么Spring怎么整出来对象的呢？有三种方式。算起来就只有两种办法创建对象：
>
> 1. 由Spring来创建对象
> 2. 由我们自己来创建对象，然后spring来拿我们的对象给需要的人。

- 无参构造方法实例化，默认的：让Spring调用bean的无参构造，生成bean实例对象给我们 【由Spring创建】
- 工厂静态方法实例化：让Spring调用一个我们自己写好的工厂类的静态方法，得到一个bean实例对象 【由咱们自己创建】
- 工厂非静态方法实例化（实例化方法）：让Spring调用一个工厂对象的非静态方法，得到一个bean实例对象 【由咱们自己创建】

1. 无参构造方法实例化 【spring创建对象】

> UserDaoImpl 是由Spring创建的。

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

2. 工厂静态方法实例化

> UserDaoImpl的由我们写好的StaticFactory的类来创建， Spring工厂没干活，只是问我们的工厂要对象而已。

- 工厂类如下：`com.itheima.factory.StaticFactory`

```
package com.itheima.factory;

import com.itheima.dao.UserDao;
import com.itheima.dao.impl.UserDaoImpl;

public class StaticFactory {

    /**
     * 使用静态方法来创建UserDaoImpl的对象。
     * @return
     */
    public static UserDao getBean(){
        System.out.println("来问StaticFactory的getBean方法要对象了~");
        return new UserDaoImpl();
    }
}
```

- 配置如下：

```
    <!--
        使用静态工厂的方式来创建对象
            运行过程：
                当我们拿着ud03 问spring要对象的时候， spring的工厂直接调用了
StaticFactory
                里面的getBean方法得到了对象，然后把这个对象返回给我们。那么从这看来，
                spring的工厂根本没有创建对象。
     -->
    <bean id="ud03" scope="prototype" class="com.itheima.factory.StaticFactory"
factory-method="getBean"/>
```

3. 工厂非静态方法实例化

> UserDaoImpl的由我们写好的InstanceFactory的类来创建，， Spring工厂没干活，只是问我们的工厂要对象而已。

- 工厂类如下：`com.itheima.factory.InstanceFactory`

```
package com.itheima.factory;

import com.itheima.dao.UserDao;
import com.itheima.dao.impl.UserDaoImpl;
```

```
public class InstanceFactory {

    /**
     * 使用非静态方法来创建UserDaoImpl的对象。
     * @return
     */
    public  UserDao getBean(){
        System.out.println("来问InstanceFactory的getBean方法要对象了~");
        return new UserDaoImpl();
    }
}
```

- 配置如下：

```
   <!--使用非静态工厂的方式来创建对象
        运行工程：
            1. spring会先创建InstanceFactory工厂类的对象，然后标记的别名是 factoryBean
            2. 当我们拿着ud04来问spring要对象时候，spring就会找到factoryBean这个名字对应
的那个对象
            3. 然后调用对象里面的getBean方法得到对象，然后返回给我们-->
    <bean id="factoryBean" class="com.itheima.factory.InstanceFactory"/>
    <bean id="ud04" factory-bean="factoryBean" factory-method="getBean"/>
```

- 小结

1. Spring工厂创建实例有三种方式： 默认的无参构造方式 | 静态工厂方式 | 实例工厂方式
2. 只有无参构造的那种方式是spring创建对象，其他两种都是由我们自己来创建对象
3. 我们使用spring的IOC ，目的就是为了把对象的创建工作交给Spring，后面这种工厂的方式，反而是我们来创建对象，所以一般不用这两种。
4. 既然如此，都不怎么用后面得的两种方式了，为什么spring还要提供这两种入口呢?

   这个其实就是为了兼容, 就是为了兼容以前的旧项目，有的旧项目50年前的旧项目，那个没有spring，但是那个时候已经使用了工厂来创建实例了。

3. IOC小结

1. IOC是什么？ 控制反转，把对象的创建工作交给spring的工厂完成，只管问spring要对象即可
2. 在applicationContext.xml里面注册。
3. 默认创建的实例是单例，如果想要多例，需要配合scope属性，设置成prototype

四、依赖注入DI【重点】

- 依赖注入：Dependency Injection，是Spring的Ioc核心的具体实现。
  - 只有存在了IOC， 才能使用DI 。只有把对象的创建工作交给Spring来管理，那么才能使用DI。
  - 类里依赖什么，由Spring注入（提供）什么

```
class  UserServiceImpl{
    private UserDao userDao;  //属性
}

class User{
    private List <Account> accountList;
}
```

- 什么是依赖注入呢?

> 托管类里面有什么属性需要完成赋值工作，把这个赋值的工作交给spring来做。由spring把属性需要用到的值赋值（注入）进来就称之为依赖注入。

我们通过Ioc把bean对象交给了Spring容器进行管理，降低了耦合性。

但是耦合性不能彻底消除，bean之间还是有一些依赖关系。比如：业务层userService要依赖于持久层userDao。

这样的依赖关系，可以交给Spring帮我们进行依赖的注入，而不用我们自己注入依赖

1. 快速入门 【重点】

需求描述

- 有dao层：`UserDao`和`UserDaoImpl`
- 有service层：`UserService`和`UserServiceImpl`
- `UserServiceImpl`中的方法依赖于`UserDaoImpl`
- 使用Spring，把`UserDaoImpl`注入给`UserServiceImpl`

开发步骤

1. 创建Maven项目，导入依赖坐标
2. 编写dao层`UserDao`及`UserDaoImpl`、service层`UserService`和`UserServiceImpl`
3. 创建Spring核心配置文件，并配置bean和依赖注入
4. 使用Spring的API，测试

需求实现

1. 创建Maven项目，导入依赖

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.1.2.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>
```

2. 编写dao层和service层代码

- dao层接口`UserDao`

```
package com.itheima.dao;

public interface UserDao {
    void add();
}
```

- dao层实现类`UserDaoImpl`

```
package com.itheima.dao.impl;

import com.itheima.dao.UserDao;

public class UserDaoImpl implements UserDao {
    public void add() {
        System.out.println("调用了UserDaoImpl的add方法~! ");
    }
}
```

- service层接口`UserService`

```
package com.itheima.service;

public interface UserService {

    void add();
}
```

- service层实现类`UserServiceImpl`

```
package com.itheima.service.impl;

import com.itheima.dao.UserDao;
import com.itheima.dao.impl.UserDaoImpl;
import com.itheima.service.UserService;

public class UserServiceImpl implements UserService {

    //创建对象
    //UserDao userDao = new UserDaoImpl();

    /*
        让spring注入进来
            1. 声明一个属性
            2. 提供这个属性的set方法
            3. 在配置文件里面，托管这个UserServiceImpl的时候，还要加上property标签
     */
    private UserDao userDao;

    public void setUserDao(UserDao userDao) {
        System.out.println("调用了setUserdao方法~");
```

```
        this.userDao = userDao;
    }

    public void add() {
        System.out.println("调用了UserServiceImpl的add方法~! ~");

        //调用方法
        userDao.add();
    }
}
```

3. 创建Spring核心配置文件，并配置bean和依赖注入

```
<?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">


    <!--1. 把UserServiceImpl交给spring管理-->
    <bean id="us" class="com.itheima.service.impl.UserServiceImpl">
        <property name="userDao" ref="userDao"/>
    </bean>


    <!--2. 把UserDaoImpl交给spring管理-->
    <bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"/>

</beans>
```

4. 使用Spring的API，测试

```
package com.itheima.test;

import com.itheima.service.UserService;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestServiceImpl {

    @Test
    public void testAdd(){

        //1. 创建工厂
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

        //2. 问工厂要对象
        UserService userService = (UserService) context.getBean("us");

        //3. 调用方法
        userService.add();
    }
}
```

小结

2. 三种常见注入方式

1. set方法注入（最常用）

1) 介绍

在类中提供需要注入的成员（依赖项）的set方法，在配置文件中注入属性的值

```
<bean id="" class="">
    <property name="属性名" value="属性值"></property>
    <property name="属性名" ref="bean的id"></property>
</bean>
```

- `property`标签：用在bean标签内部，表示要给某一属性注入数据
    - `name`：属性名称
    - `value`：要注入的属性值，注入简单类型值
    - `ref`：要注入的属性值，注入其它bean对象

2) 示例

- service

```
package com.itheima.service.impl;

import com.itheima.dao.UserDao;
import com.itheima.dao.impl.UserDaoImpl;
import com.itheima.service.UserService;

/*
    使用set方法完成依赖注入
 */
public class UserServiceImpl02 implements UserService {

    private String address;

    public void setAddress(String address) {
        this.address = address;
    }

    private UserDao userDao;

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

    public void add() {
        System.out.println("调用了UserServiceImpl的add方法~! ~"+address);
        userDao.add();
    }
}
```

- 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">


    <!--
        1. 把UserServiceImpl交给spring管理
            property标签：
                作用： 用来完整依赖注入，其实就是完成属性的赋值工作
                属性：
                    name :    UserServiceImpl02里面的属性名字
                    value :  只匹配基本数据类型和String 给属性注入的值
                    ref :  匹配注入对象类型，写进来的是bean的id名字 | 别名
     -->
    <bean id="us02" class="com.itheima.service.impl.UserServiceImpl02">
        <property name="address" value="深圳湾1号"/>
        <property name="userDao" ref="userDao"/>
    </bean>

    <!--2. 把UserDaoImpl交给spring管理-->
    <bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"/>

</beans>
```

2. 构造方法注入

1) 介绍

在类中提供构造方法，构造方法的每个参数就是一个依赖项，通过构造方法给依赖项注入值。

```
<bean id="" class="">
    <constructor-arg name="构造参数名称" value="构造参数的值"></constructor-arg>
    <constructor-arg name="构造参数名称" ref="bean的id"></constructor-arg>
</bean>
```

- `name`：构造参数的名称
- `type`：构造参数的类型
- `index`：构造参数的索引
- `value`：要注入的值，注入简单类型值
- `ref`：要注入的值，注入其它bean对象

2) 示例

- service

```
package com.itheima.service.impl;

import com.itheima.dao.UserDao;
import com.itheima.service.UserService;

/*
    使用有参构造的方式，完成属性的注入。
```

```
 */
public class UserServiceImpl03 implements UserService {

    private String address;
    private UserDao userDao;


    public UserServiceImpl03(String address, UserDao userDao) {
        this.address = address;
        this.userDao = userDao;
    }

    public void add() {
        System.out.println("调用了UserServiceImpl的add方法~! ~" +  address);

        //调用方法
        userDao.add();
    }
}
```

- 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">


    <!--
        1. 把UserServiceImpl交给spring管理
            1.1 默认spring创建对象走的是无参构造方法，如果期望走有参构造，需要指定标签
constructor-arg
            1.2 constructor-arg:
                作用：  用于指定构造的参数
                属性：
                    name :构造参数的名字
                    value : 给构造参数赋值 ，匹配的是基本数据类型和字符串
                    ref :给构造参数赋值 ，匹配的是对象
     -->
    <bean id="us03" class="com.itheima.service.impl.UserServiceImpl03">
        <constructor-arg name="address" value="北京"/>
        <constructor-arg name="userDao" ref="userDao"/>
    </bean>

    <!--2. 把UserDaoImpl交给spring管理-->
    <bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"/>

</beans>
```

3. p名称空间注入

1) 介绍

> p名称空间注入，本质仍然是set方法注入

在xml中引入p名称空间的约束

然后通过`p:属性名称=""`来注入简单数据、使用`p:属性名称-ref=""`注入其它bean对象，它的本质仍然是set方法注入

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="" class="" p:属性名="简单值" p:属性名-ref="bean的id"></bean>

</beans>
```

2) 示例

- service

```
package com.itheima.service.impl;

import com.itheima.dao.UserDao;
import com.itheima.service.UserService;

/*
    使用p名称空间完成依赖注入
 */
public class UserServiceImpl04 implements UserService {

    private String address;
    private UserDao userDao;

    public void setAddress(String address) {
        this.address = address;
    }

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

    public void add() {
        System.out.println("调用了UserServiceImpl的add方法~！~");

        //调用方法
        userDao.add();
    }
}
```

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


    <!--
        1. 把UserServiceImpl交给spring管理
            使用p名称空间完成依赖注入：
                1. 它背后还是走set方法
                2. 注入普通的属性：   p:属性="值"
                3. 注入对象属性：   p:属性-ref="id的名字"
    -->
    <bean id="us04" class="com.itheima.service.impl.UserServiceImpl04"
p:address="深圳" p:userDao-ref="userDao"/>


    <!--2. 把UserDaoImpl交给spring管理-->
    <bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"/>

</beans>
```

小结

1. 注入方式有三种，set方法 ，构造方法注入，p名称空间
2. 最常用的是set方法。
3. 以后如果使用注解了，方法也不需要写了。

3. 注入集合数据

介绍

- 前边我们介绍了如何注入简单数据类型和bean对象，但是在实际开发中，可能会需要给集合属性注入数据，比如：给数组、List、Set、Map等注入数据

示例

代码

```
package com.itheima.service.impl;

import com.itheima.dao.UserDao;
import com.itheima.service.UserService;

import java.util.*;

/*
    注入集合数据，使用set方法来注入
 */
public class UserServiceImpl05 implements UserService {

    // 注入集合数据 ：   数组 、 list 、 set 、 map、 properties

    private String [] array;
    private List<String> list;
```

```
    private Set<String> set;
    private Map<String ,String> map;
    private Properties properties;

    public void setArray(String[] array) {
        this.array = array;
    }

    public void setList(List<String> list) {
        this.list = list;
    }

    public void setSet(Set<String> set) {
        this.set = set;
    }

    public void setMap(Map<String, String> map) {
        this.map = map;
    }

    public void setProperties(Properties properties) {
        this.properties = properties;
    }

    public void add() {
        System.out.println("调用了UserServiceImpl05的add方法~! ~");
        System.out.println("array=" + Arrays.toString(array));
        System.out.println("list=" + list);
        System.out.println("set=" + set);
        System.out.println("map=" + map);
        System.out.println("properties=" + properties);
    }
}
```

配置注入数据

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

    <!--注入集合数据-->

    <bean id="us05" class="com.itheima.service.impl.UserServiceImpl05">

       <!-- 1. 注入数组-->
        <property name="array">
            <array>
                <value>aa</value>
                <value>bb</value>
                <value>cc</value>
            </array>
        </property>
```

```
        <!--2. 注入集合list-->
        <property name="list">
            <list>
                <value>list01</value>
                <value>list02</value>
                <value>list03</value>
            </list>
        </property>

        <!--3. 注入set-->
        <property name="set">
            <set>
                <value>set01</value>
                <value>set02</value>
                <value>set03</value>
            </set>
        </property>

        <!--4. 注入map-->
        <property name="map">
            <map>
                <entry key="username" value="admin"/>
                <entry key="password" value="123456"/>
            </map>
        </property>

        <!--5. 注入properties-->
        <property name="properties">
            <props>
                <prop key="driverClass">com.mysql.jdbc.Driver</prop>
                <prop key="jdbcUrl">jdbc:mysql:///day40_spring</prop>
                <prop key="username">root</prop>
                <prop key="password">root</prop>
            </props>
        </property>
    </bean>
</beans>
```

> 所有单列结构的数据集合，标签可以互换使用。例如：List、Set、数组等
>
> 所有键值对结构的数据集合，标签可以互换使用。例如：Map、Properties等

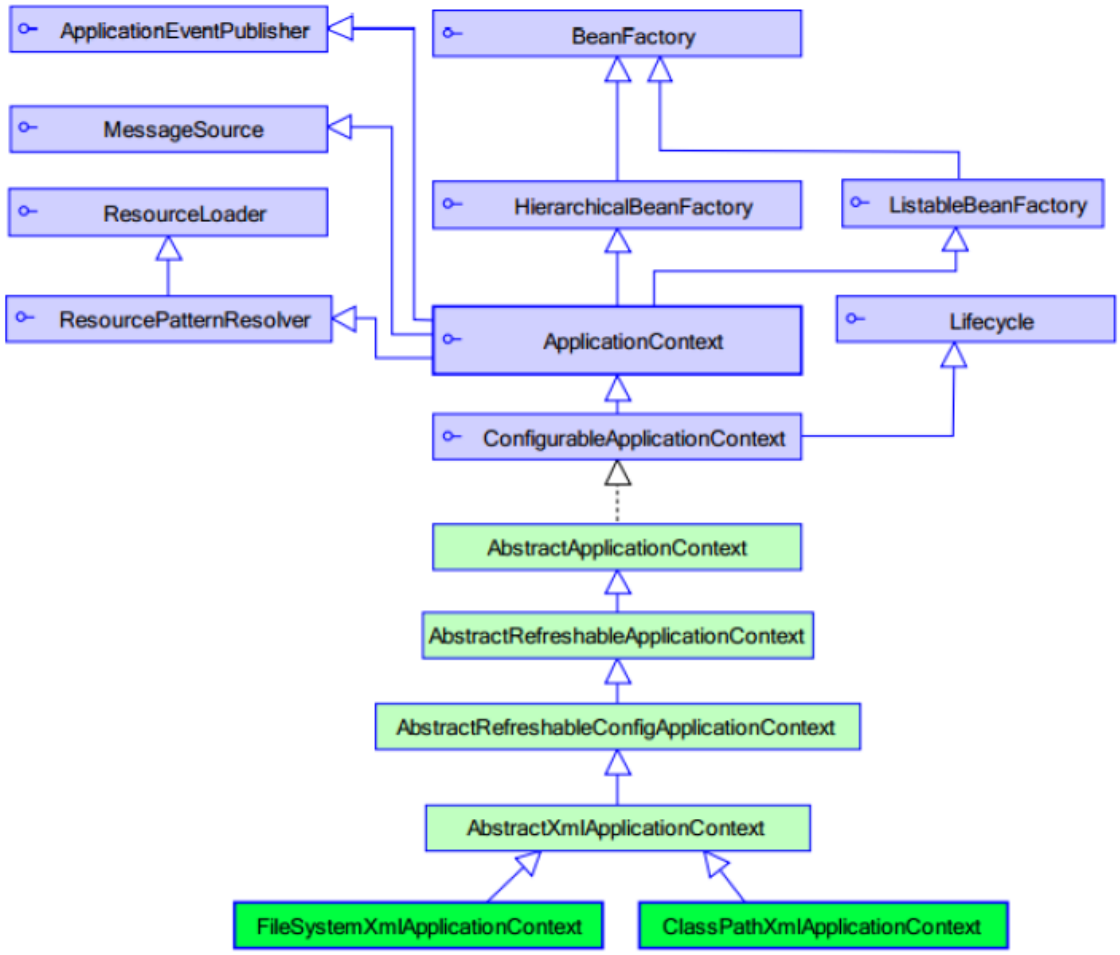
小结

1. 所有的DI数据类型里面，最常用的是对象数据。
2. 最常用的方式 set方法。
3. 数组 、 list 、set写法基本一样， map 和 properties基本一样。

五、相关API介绍（了解）

1. `ApplicationContext`的继承体系

- `ApplicationContext`：接口，代表应用上下文，可以通过其实例对象获取Spring容器中的bean对象



2. `ApplicationContext`

2.1 XmlBeanFactory 和 `ApplicationContext`的区别

- ApplicationContext 是现在使用的工厂

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

- XmlBeanFactory 是老版本使用的工厂,目前已经被废弃【了解】

```
BeanFactory beanFactory =
    new XmlBeanFactory(new ClassPathResource("applicationContext.xml"));
```

- 两者的区别:
  - ApplicationContext加载方式是框架启动时就开始创建所有单例的bean,存到了容器里面
  - XmlBeanFactory加载方式是用到bean时再加载(目前已经被废弃)

2.2 `ApplicationContext`的实现类

2.2.1 `ClassPathXmlApplicationContext`

- 是从类加载路径里，加载xml配置文件的
- 什么是类加载路径：代码编译之后的那个classes文件夹，
    - 开发中可以认为Maven项目的：**Java文件夹、resources文件夹，都是类加载路径**

2.2.2 `FileSystemXmlApplicationContext`

- 从磁盘路径里，加载xml配置文件的

2.2.3 `AnnotationConfigApplicationContext`

- 用注解配置Spring时，通过此类加载配置类创建Spring容器，它用于读取类上的注解配置

2.3 `getBean()`方法

- ApplicationContext提供了多种getBean方法的重载，常用的如下：

方法	参数	返回值
`getBean(String beanId)`	bean的id	`Object`，bean对象
`getBean(String beanId,Class beanType)`	bean的Class类型	bean对象
`getBean(Class beanType)`		bean对象
getBeanDefinitionNames		String[] 获取工厂管理的对象的名字

六、CURD练习【了解】

需求描述

- 完成帐户信息的增、删、改、查操作，要求使用Spring对service层和dao层解耦
    - 把service和dao交给spring管理
    - 如果有依赖的话，需要让spring去注入内容

需求分析

1. 准备工作：
    - 创建Maven的Java项目，配置坐标，引入依赖
    - 创建数据库
    - 创建JavaBean
2. 编写代码：
    - 创建service和dao的接口和实现类，并添加上：查询全部、添加帐号、修改帐号、删除帐号的功能
3. 配置文件：
    - 创建Spring核心配置文件，配置所有的bean
4. 测试
    - 创建单元测试类，测试功能是否正常

需求实现

3.1 准备工作

0. 准备数据库

> 需要先手动创建一个数据库，然后再运行下面的这段sql语句

```
create table t_account (id int primary key auto_increment , name varchar(25) ,
money int);
insert into t_account values(null , 'zs' , 1000);
insert into t_account values(null , 'ls' , 2000);
insert into t_account values(null , 'ww' , 3000);
```

1. 创建Maven的Java项目，项目坐标自定，然后引入依赖如下：

```
<dependencies>
        <!-- 数据库驱动 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.47</version>
        </dependency>

        <!-- c3p0连接池（也可以用其它连接池） -->
        <dependency>
            <groupId>com.mchange</groupId>
            <artifactId>c3p0</artifactId>
            <version>0.9.5.2</version>
        </dependency>

        <!-- DBUtils工具包 -->
        <dependency>
            <groupId>commons-dbutils</groupId>
            <artifactId>commons-dbutils</artifactId>
            <version>1.7</version>
        </dependency>

        <!-- Spring -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.2.RELEASE</version>
        </dependency>

        <!-- 单元测试 -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.8</version>
        </dependency>
```

```
    </dependencies>
```

2. 创建JavaBean：Account类如下：

```
package com.itheima.bean;

import lombok.Data;

@Data
public class Account {
    private int id;
    private String name;
    private int money;
}
```

3.2 编写代码

1) Service层代码如下：

1. Service层接口：`AccountService`

```
package com.itheima.service;

import com.itheima.bean.Account;

import java.sql.SQLException;
import java.util.List;

public interface AccountService {
    int add(Account account) throws SQLException;
    int delete(int id) throws SQLException;
    int update(Account account) throws SQLException;
    Account findById(int id) throws SQLException;
    List<Account> findAll() throws SQLException;
}
```

2. Service实现类：`AccountServiceImpl`

```
package com.itheima.service.impl;

import com.itheima.bean.Account;
import com.itheima.dao.AccountDao;
import com.itheima.dao.impl.AccountDaoImpl;
import com.itheima.service.AccountService;

import java.sql.SQLException;
import java.util.List;

public class AccountServiceImpl implements AccountService {

    //1. 声明属性
    private AccountDao accountDao;

    //2. 提供属性的set方法
```

```
    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    public int add(Account account) throws SQLException {

        //以前的代码
        //AccountDao dao = new AccountDao();

        //有接口之后
        //AccountDao dao = new AccountDaoImpl();
        //dao.add(account);

        return accountDao.add(account);
    }

    public int delete(int id) throws SQLException {
        return accountDao.delete(id);
    }

    public int update(Account account) throws SQLException {
        return accountDao.update(account);
    }

    public Account findById(int id) throws SQLException {
        return accountDao.findById(id);
    }

    public List<Account> findAll() throws SQLException {
        return accountDao.findAll();
    }
}
```

2) dao层代码如下：

1. dao层接口：`AccountDao`

```
package com.itheima.dao;

import com.itheima.bean.Account;

import java.sql.SQLException;
import java.util.List;

public interface AccountDao {
    int add(Account account) throws SQLException;
    int delete(int id) throws SQLException;
    int update(Account account) throws SQLException;
    Account findById(int id) throws SQLException;
    List<Account> findAll() throws SQLException;
}
```

2. dao实现类：`AccountDaoImpl`

```
package com.itheima.dao.impl;
```

```java
import com.itheima.bean.Account;
import com.itheima.dao.AccountDao;
import org.apache.commons.dbutils.QueryRunner;
import org.apache.commons.dbutils.handlers.BeanHandler;
import org.apache.commons.dbutils.handlers.BeanListHandler;

import java.sql.SQLException;
import java.util.List;

public class AccountDaoImpl  implements AccountDao {

    //1. 声明属性
    private QueryRunner runner;

    //2. 提供属性的set方法
    public void setRunner(QueryRunner runner) {
        this.runner = runner;
    }

    public int add(Account account) throws SQLException {
        //以前的代码
        //QueryRunner runner = new QueryRunner(C3P0Utils.getDataSource());

        String sql = "insert into t_account values(null , ? , ? )";
        return runner.update(sql , account.getName() , account.getMoney());
    }

    public int delete(int id) throws SQLException {
        String sql = "delete from t_account where id = ?";
        return runner.update(sql , id);
    }

    public int update(Account account) throws SQLException {
        String sql = "update t_account set name = ? , money = ?  where id =
? ";
        return runner.update(sql , account.getName() ,
account.getMoney(),account.getId());
    }

    public Account findById(int id) throws SQLException {
        String sql = "select * from t_account where id = ?";
        return runner.query(sql , new BeanHandler<Account>(Account.class) ,
id );
    }

    public List<Account> findAll() throws SQLException {
        String sql = "select * from t_account";
        return runner.query(sql , new BeanListHandler<Account>
(Account.class)  );
    }
}
```

3.3 提供配置

1. 创建Spring的核心配置文件：`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
http://www.springframework.org/schema/context/spring-context.xsd">

    <!--1. 把service交给spring管理-->
    <bean id="as" class="com.itheima.service.impl.AccountServiceImpl">
        <property name="accountDao" ref="ad"/>
    </bean>

    <!--2. 把dao交个spring管理-->
    <bean id="ad" class="com.itheima.dao.impl.AccountDaoImpl">
        <property name="runner" ref="runner"/>
    </bean>

    <!--
        3. 把QueryRunner交给spring管理
            3.1 默认情况下，spring创建某一个类的对象的时候，执行的都是这个类的无参构造方法
            3.2 但是我们以前创建QueryRunner对象的时候，都不是走无参构造的方式创建，而是走有参构造方式创建
                如：   QueryRunner runner = new QueryRunner(C3P0Utils.getDataSource());
            3.3 需要让spring管理DataSource，然后给QueryRunner注入进来
    -->
    <bean id="runner" class="org.apache.commons.dbutils.QueryRunner">
        <constructor-arg name="ds" ref="ds"/>
    </bean>


    <!--引入外部的properties文件-->
    <context:property-placeholder location="db.properties"/>

    <!--
        4. 把DataSource交给Spring管理
            4.1 DataSource其实就是连接池对象
            4.2 以前我们使用C3p0的时候，连接池对象是一个叫做ComboPooledDataSouce，
                所以现在也需要把这个类交给spring管理
            4.3 由于是去连接数据库，建立连接池，所以需要告诉这个类，
                连接什么数据库，账号和密码是什么？
    -->
    <bean id="ds" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${driverClass}"/>
        <property name="jdbcUrl" value="${jdbcUrl}"/>
        <property name="user" value="${user}"/>
        <property name="password" value="${password}"/>
    </bean>
```

```
    <!--<bean id="ds" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="com.mysql.jdbc.Driver"/>
        <property name="jdbcUrl"
value="jdbc:mysql://localhost:3306/day41_spring"/>
        <property name="user" value="root"/>
        <property name="password" value="root"/>
    </bean>-->
</beans>
```

3.4 功能测试

1. 编写单元测试类`AccountTest`如下：

```
package com.itheima.test;

import com.itheima.bean.Account;
import com.itheima.service.AccountService;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.sql.SQLException;

public class TestAccountServiceImpl {

    @Test
    public void testAdd() throws SQLException {
        //1. 创建工厂
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

        //2. 问工厂要对象
        AccountService as = (AccountService) context.getBean("as");

        //3. 调用方法
        Account a = new Account();
        a.setName("赵信");
        a.setMoney(10);

        int row  = as.add(a);
        System.out.println("row=" + row);
    }


    @Test
    public void testDelete() throws SQLException {
        //1. 创建工厂
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

        //2. 问工厂要对象
        AccountService as = (AccountService) context.getBean("as");

        //3. 调用方法
        int row  = as.delete(4);
        System.out.println("row=" + row);
    }
```

```
    @Test
    public void testUpdate() throws SQLException {
        //1. 创建工厂
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

        //2. 问工厂要对象
        AccountService as = (AccountService) context.getBean("as");

        //3. 调用方法

        //3.1 先查询
        Account a = as.findById(3);
        a.setMoney(666);

        //3.2 再修改
        int row  = as.update(a);
        System.out.println("row=" + row);
    }

    @Test
    public void testFindAll() throws SQLException {
        //1. 创建工厂
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

        //2. 问工厂要对象
        AccountService as = (AccountService) context.getBean("as");

        //3. 调用方法

        System.out.println("list=" + as.findAll());
    }
}
```

补充

引入`properties`文件

如果需要在`applicationContext.xml`中引入properties文件：

- 准备一个properties文件放在resources里：`db.properties`

```
driverClass=com.mysql.jdbc.Driver
jdbcUrl=jdbc:mysql://localhost:3306/day40_spring
user=root
password=root
```

- 在`applicationContext.xml`中引入并使用`db.properties`
    - Spring的名称空间（建议使用idea自动生成的，如果idea抽风了，就自己手写）

```
<beans
       xmlns:名称空间="http://www.springframework.org/schema/名称空间"
       xsi:scehmaLocation="
          http://www.springframework.org/schema/名称空间
         http://www.springframework.org/schema/名称空间/spring-名称空间.xsd">
</beans>
```

- 使用context名称空间提供的标签，引入外部的properties文件

> context的标签，硬着头皮写出来就可以了，不要害怕！

```
    <!--1. 引入外部的properties文件-->
    <context:property-placeholder location="db.properties"/>

    <!--2. 使用properties文件-->
    <bean id="ds" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${driverClass}"/>
        <property name="jdbcUrl" value="${jdbcUrl}"/>
        <property name="user" value="${user}"/>
        <property name="password" value="${password}"/>
    </bean>
```

分模块提供配置文件

在大型项目开发中，如果把所有的配置都写在一个配置文件`applicationContext.xml`中，会导致：

- 配置文件过于臃肿
- 不利于分模块开发，不利于模块之间的解耦

Spring提供了分模块配置的方式，即：每个模块|层提供一个配置文件，在核心配置文件中引入模块配置：

- dao模块有一个配置文件：`applicationContext-dao.xml`只配置dao相关的对象
- service模块有一个配置文件：`applicationContext-service.xml`只配置service相关的对象
- 有一个总的核心配置文件：`applicationContext-all.xml`如下

```
<import resource="classpath:applicationContext-service.xml"/>
<import resource="classpath:applicationContext-dao.xml"/>
```

小结

```
1. 把dao和service都交给spring托管
2. 在service里面注入进来dao
3. dao里面要用到QueryRunner ， QueryRunner也可以让Spring注入进来。
4. 但凡是我们需要自己new对象，都交给spring来完成，然后注入进来即可。
```

七、Spring整合Junit【掌握】

在上边的CURD中，单元测试类里还需要我们自己去创建`ApplicationContext`，并自己去获取bean对象。Spring提供了整合Junit的方法，让单元测试更简洁方便。

注解简介

注解	说明
`@RunWith`	用在测试类上，用于声明不再使用Junit，而是使用Spring提供的运行环境
`@ContextConfiguration`	用在测试类上，用于指定Spring配置类、或者Spring的配置文件

> Spring提供了单元测试的运行环境：**SpringJunit4ClassRunner**，配置到`@RunWith`注解上：
>
> `@RunWith(SpringJunit4ClassRunner.class)`

- 要使用以上注解，需要导入jar包依赖：`spring-test` 和 `junit`

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.1.2.RELEASE</version>
</dependency>

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

使用示例

步骤

1. 在pom.xml文件中增加依赖：`spring-test` 和 `junit`
2. 修改单元测试类
    1. 在单元测试类上增加注解：`@RunWith(SpringJunit4ClassRunner.class)`

       目的：使用Spring的单元测试运行器，替换Junit原生的运行器
    2. 在单元测试类上增加注解：`@ContextConfiguration()`

       目的：指定配置文件或配置类
    3. 在测试类里的依赖项上，直接使用`@Autowired`注入依赖

实现

1. 在pom.xml文件中增加依赖：`spring-test` 和 `junit`

```
<dependencies>
        <!-- Spring -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.2.RELEASE</version>
        </dependency>

        <!--spring单元测试的依赖-->
        <dependency>
```

```
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>5.1.2.RELEASE</version>
        </dependency>

        <!--Junit的依赖-->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>

    </dependencies>
```

2. UserService接口

```
package com.itheima.service;

public interface UserService {
    void add();
}
```

3. UserServiceImpl实现类

```
package com.itheima.service.impl;

import com.itheima.service.UserService;

public class UserServiceImpl implements UserService {
    public void add() {
        System.out.println("调用了UserServiceImpl的add方法~！~");
    }
}
```

4. 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">

    <bean id="us" class="com.itheima.service.impl.UserServiceImpl"/>
</beans>
```

5. 修改单元测试类

```
package com.itheima.test;

import com.itheima.service.UserService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
```

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

/*
    spring提供的单元测试支持里面，主要依赖两个注解：

        @RunWith ：
            用来指定单元测试使用的环境是什么，在里面指定SpringJUnit4ClassRunner 表示单元测试环境
            使用spring提供的环境

        @ContextConfiguration ：
            1. 用来设置核心配置文件，以便spring的测试环境在背后能够创建出来spring的工厂，进而进行IOC和DI的工作。
            2. 指定核心配置文件的时候，需要加上classpath: 这是固定写法
 */

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class TestUserSerivceImpl02 {

    /*
        @Autowired 下一次课才讲！ 它的作用就是把对象注入到属性us身上。
     */
    @Autowired
    private UserService us;

    @Test
    public void testAdd(){
        us.add();
    }
}
```

小结

```
1. 导入依赖
2. 在测试类上打注解
3. 直接在测试类里面使用@Autowired注入对象。
```

总结

- IOC + DI
- IOC :
  - 是什么， 是控制反转，就是把对象的创建工作交给spring来完成。
  - 怎么做?
    - 1. 写接口和实现类
      2. 在applicationContext.xml中，写bean标签，把实现类交给spring管理
  - 细节:
    - 1. 默认情况下，spring创建对象是单例的。执行它的无参构造。
      2. 如果期望做成多例，需要配合一个属性 scope="prototype"
- DI

    - 是什么？ 是依赖注入，其实就是让spring创建对象的时候，顺便完成属性的赋值工作。
    - 依赖注入的方式：
        1. set方法注入
        2. 有参构造方法注入
        3. p名称空间注入（背后走的是set方法）
    - 依赖注入的数据类型
        1. 注入普通的数据
        2. 注入对象数据 【最常用】
        3. 注入集合数据
- spring整合Junit
    - @RunWith(SpringJunit4ClassRunner.class)
    - @ContextConfiguration("classpath:applicationContext.xml")