任务一：IOC控制反转

任务一课程主要内容：

```
* spring概念介绍
* IOC
* spring快速入门
* spring相关API介绍
* Spring配置文件
* DBUtils
* spring注解开发
* spring整合Junit
```

一 Spring概述

1.1 Spring是什么

Spring是分层的 Java SE/EE应用 **full-stack(全栈式) 轻量级开源框架。**

提供了**表现层** SpringMVC和**持久层** Spring JDBC Template以及 **业务层** 事务管理等众多的企业级应用技术，还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。

两大核心：以 **IOC**（Inverse Of Control：控制反转）和 **AOP**（Aspect Oriented Programming：面向切面编程）为内核。

1.2 Spring发展历程

```
* EJB
    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

* Spring
    Rod Johnson（ Spring 之父）
        改变Java世界的大师级人物

    2002年编著《Expert one on one J2EE design and development》
        指出了JavaEE和EJB组件框架中的存在的一些主要缺陷；提出普通java类依赖注入更为简单的解决方案。

    2004年编著《Expert one-on-one J2EE Development without EJB》
        阐述了JavaEE开发时不使用EJB的解决方式（Spring 雏形）
        同年4月spring1.0诞生
```

2006年10月，发布 Spring2.0
2009年12月，发布 Spring3.0
2013年12月，发布 Spring4.0
2017年9月， 发布最新 Spring5.0 通用版（GA）

1.3 Spring优势

1）方便解耦，简化开发
Spring就是一个容器，可以将所有对象创建和关系维护交给Spring管理
什么是耦合度？对象之间的关系，通常说当一个模块(对象)更改时也需要更改其他模块(对象)，这就是耦合，耦合度过高会使代码的维护成本增加。要尽量解耦

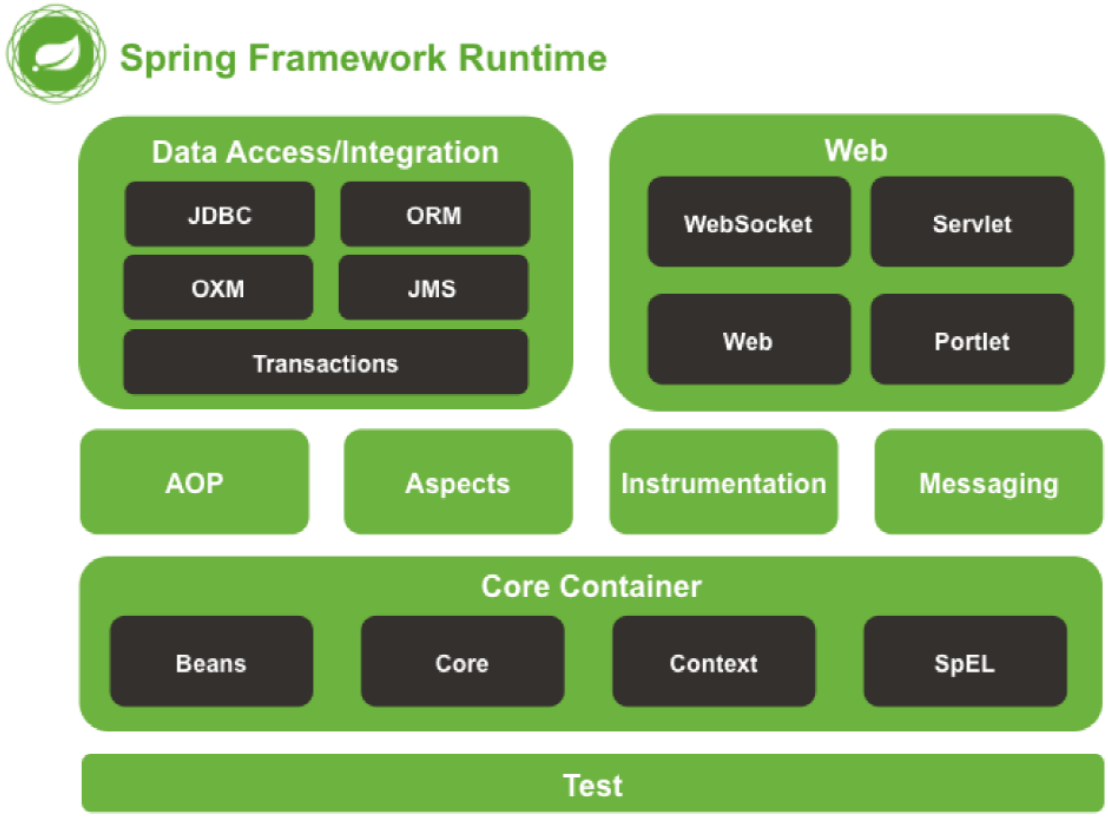
2）AOP编程的支持
Spring提供面向切面编程，方便实现程序进行权限拦截，运行监控等功能。

3）声明式事务的支持
通过配置完成事务的管理，无需手动编程

4）方便测试，降低JavaEE API的使用
Spring对Junit4支持，可以使用注解测试

5）方便集成各种优秀框架
不排除各种优秀的开源框架，内部提供了对各种优秀框架的直接支持

1.4 Spring体系结构



二 初识IOC

2.1 概述

控制反转（Inverse Of Control）不是什么技术，而是一种设计思想。它的目的是指导我们设计出更加松耦合的程序。

> 控制：在java中指的是对象的控制权限（创建、销毁）
>
> 反转：指的是对象控制权由原来 **由开发者在类中手动控制** 反转到 **由Spring容器控制**

举个栗子

```
* 传统方式
    之前我们需要一个userDao实例，需要开发者自己手动创建 new UserDao();

* IOC方式
    现在我们需要一个userDao实例，直接从spring的IOC容器获得，对象的创建权交给了spring控制
```

2.2 自定义IOC容器

2.2.1 介绍

需求

实现service层与dao层代码解耦合

步骤分析

1. 创建java项目，导入自定义IOC相关坐标
2. 编写Dao接口和实现类
3. 编写Service接口和实现类
4. 编写测试代码

2.2.2 实现

1）创建java项目，导入自定义IOC相关坐标

```
<dependencies>
    <dependency>
        <groupId>dom4j</groupId>
        <artifactId>dom4j</artifactId>
        <version>1.6.1</version>
    </dependency>
    <dependency>
        <groupId>jaxen</groupId>
        <artifactId>jaxen</artifactId>
        <version>1.1.6</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
```

```
        <version>4.12</version>
    </dependency>
</dependencies
```

2）编写Dao接口和实现类

```
public interface UserDao {

    public void save();
}
```

```
public class UserDaoImpl implements UserDao {

    public void save() {
        System.out.println("保存成功了...");
    }
}
```

3）编写Service接口和实现类

```
public interface UserService {

    public void save();
}
```

```
public class UserServiceImpl implements UserService {

    private UserDao userDao;

    public void save(){
        userDao = new UserDaoImpl();
        userDao.save();
    }
}
```

4）编写测试代码

```
public class UserTest {

    @Test
    public void testSave() throws Exception {
        UserService userService = new UserServiceImpl();
        userService.save();
    }
}
```

5）问题

当前service对象和dao对象耦合度太高，而且每次new的都是一个新的对象，导致服务器压力过大。

> 解耦合的原则是编译期不依赖，而运行期依赖就行了。

6）编写beans.xml

把所有需要创建对象的信息定义在配置文件中

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans>
    <bean id="userDao" class="com.lagou.dao.impl.UserDaoImpl"></bean>
</beans>
```

7）编写BeanFactory工具类

```
public class BeanFactory {

    private static Map<String, Object> ioc = new HashMap<>();

    // 程序启动时，初始化对象实例
    static {
        try {
            // 1.读取配置文件
            InputStream in =
BeanFactory.class.getClassLoader().getResourceAsStream("beans.xml");
            // 2.解析xml
            SAXReader saxReader = new SAXReader();
            Document document = saxReader.read(in);
            // 3.编写xpath表达式
            String xpath = "//bean";
            // 4.获取所有的bean标签
            List<Element> list = document.selectNodes(xpath);
            // 5.遍历并创建对象实例，设置到map集合中
            for (Element element : list) {
                String id = element.attributeValue("id");
                String className = element.attributeValue("class");
                Object object = Class.forName(className).newInstance();
                ioc.put(id, object);
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // 获取指定id的对象实例
    public static Object getBean(String beandId) {
        return ioc.get(beandId);
    }
}
```

8）修改UserServiceImpl实现类

```
public class UserServiceImpl implements UserService {

    private UserDao userDao;

    public void save() throws Exception
        userDao = (UserDao) BeanFactory.getBean("userDao");
        userDao.save();
    }
}
```

2.2.3 知识小结

```
* 其实升级后的BeanFactory就是一个简单的Spring的IOC容器所具备的功能。

* 之前我们需要一个userDao实例，需要开发者自己手动创建 new UserDao();

* 现在我们需要一个userdao实例，直接从spring的IOC容器获得，对象的创建权交给了spring控制

* 最终目标：代码解耦合
```

三 Spring快速入门

3.1 介绍

需求：借助spring的IOC实现service层与dao层代码解耦合

步骤分析

```
1. 创建java项目，导入spring开发基本坐标
2. 编写Dao接口和实现类
3. 创建spring核心配置文件
4. 在spring配置文件中配置 UserDaoImpl
5. 使用spring相关API获得Bean实例
```

3.2 实现

1）创建java项目，导入spring开发基本坐标

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>
```

2）编写Dao接口和实现类

```java
public interface UserDao {

    public void save();
}
```

```java
public class UserDaoImpl implements UserDao {

    public void save() {
        System.out.println("保存成功了...");
    }
}
```

3）创建spring核心配置文件

```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">

</beans>
```

4）在spring配置文件中配置 UserDaoImpl

```xml
<beans ...>
    <bean id="userDao" class="com.lagou.dao.impl.UserDaoImpl"></bean>
</beans>
```

5）使用spring相关API获得Bean实例

```
public class UserTest {

    @Test
    public void testSave() throws Exception {
        ApplicationContext applicationContext =
                        new
ClassPathXmlApplicationContext("applicationContext.xml");
        UserDao userDao = (UserDao) applicationContext.getBean("userDao");
        userDao.save();
    }
}
```

3.3 知识小结

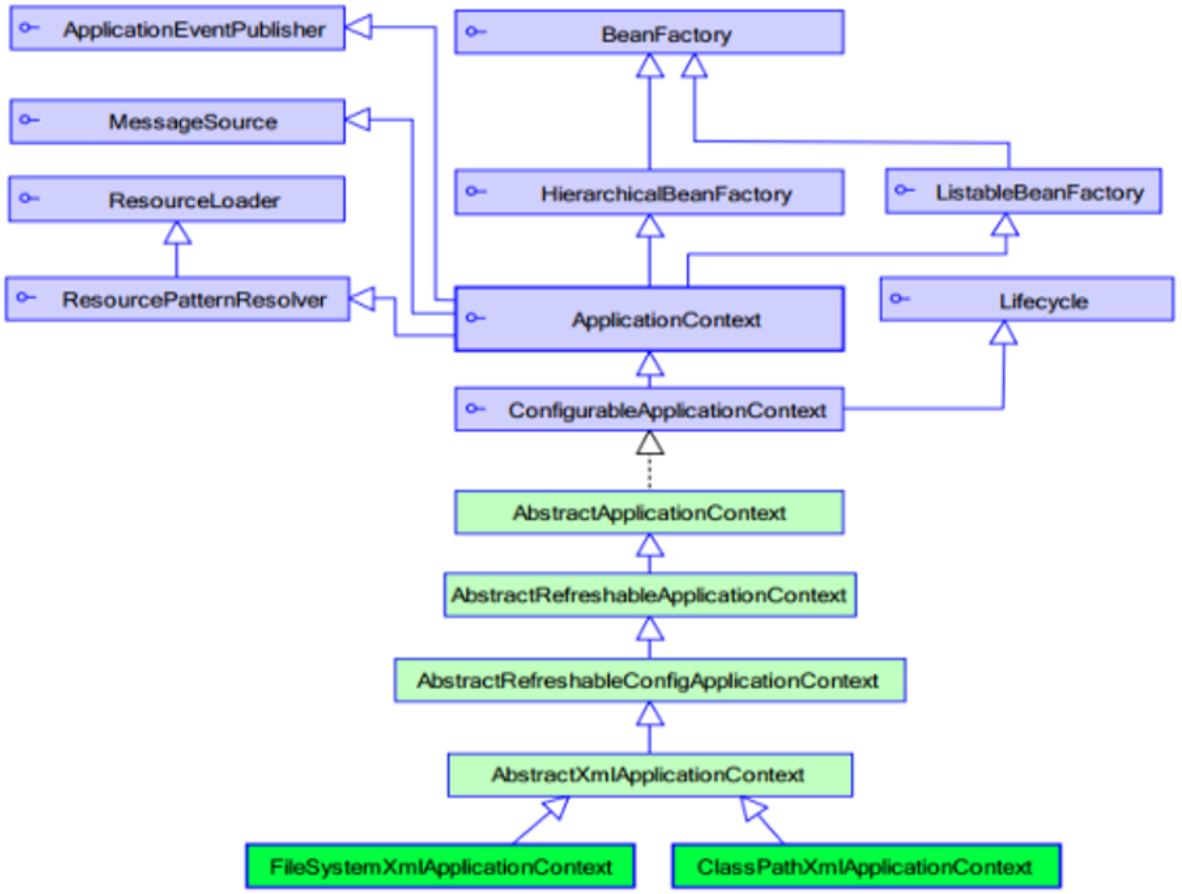
Spring的开发步骤

1. 导入坐标
2. 创建Bean
3. 创建applicationContext.xml
4. 在配置文件中进行Bean配置
5. 创建ApplicationContext对象，执行getBean

四 Spring相关API

4.1 API继承体系介绍

Spring的API体系异常庞大，我们现在只关注两个BeanFactory和ApplicationContext



4.2 BeanFactory

BeanFactory是 IOC 容器的核心接口，它定义了IOC的基本功能。

> 特点：在第一次调用getBean()方法时，创建指定对象的实例

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

4.3 ApplicationContext

代表应用上下文对象，可以获得spring中IOC容器的Bean对象。

> 特点：在spring容器启动时，加载并创建所有对象的实例

常用实现类

```
1. ClassPathXmlApplicationContext
       它是从类的根路径下加载配置文件 推荐使用这种。

2. FileSystemXmlApplicationContext
       它是从磁盘路径上加载配置文件，配置文件可以在磁盘的任意位置。

3. AnnotationConfigApplicationContext
       当使用注解配置容器对象时，需要使用此类来创建 spring 容器。它用来读取注解。
```

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

常用方法

```
1. Object getBean(String name);
     根据Bean的id从容器中获得Bean实例，返回是Object，需要强转。

2. <T> T getBean(Class<T> requiredType);
     根据类型从容器中匹配Bean实例，当容器中相同类型的Bean有多个时，则此方法会报错。

3. <T> T getBean(String name,Class<T> requiredType);
     根据Bean的id和类型获得Bean实例，解决容器中相同类型Bean有多个情况。
```

4.4 知识小结

```
ApplicationContext app = new ClasspathXmlApplicationContext("xml文件");
    app.getBean("id");
    app.getBean(Class);
```

五 Spring配置文件

5.1 Bean标签基本配置

```
<bean id="" class=""></bean>

* 用于配置对象交由Spring来创建。

* 基本属性:
    id：Bean实例在Spring容器中的唯一标识
    class：Bean的全限定名

* 默认情况下它调用的是类中的 无参构造函数，如果没有无参构造函数则不能创建成功。
```

5.2 Bean标签范围配置

```
<bean id="" class="" scope=""></bean>
```

scope属性指对象的作用范围，取值如下：

取值范围	说明
singleton	默认值，单例的
prototype	多例的
request	WEB项目中，Spring创建一个Bean的对象，将对象存入到request域中
session	WEB项目中，Spring创建一个Bean的对象，将对象存入到session域中
global session	WEB项目中，应用在Portlet环境，如果没有Portlet环境那么globalSession 相当于 session

```
1. 当scope的取值为singleton时
      Bean的实例化个数：1个
      Bean的实例化时机：当Spring核心文件被加载时，实例化配置的Bean实例
      Bean的生命周期：
        对象创建：当应用加载，创建容器时，对象就被创建了
        对象运行：只要容器在，对象一直活着
        对象销毁：当应用卸载，销毁容器时，对象就被销毁了

2. 当scope的取值为prototype时
      Bean的实例化个数：多个
      Bean的实例化时机：当调用getBean()方法时实例化Bean
      Bean的生命周期：
        对象创建：当使用对象时，创建新的对象实例
        对象运行：只要对象在使用中，就一直活着
        对象销毁：当对象长时间不用时，被 Java 的垃圾回收器回收了
```

5.3 Bean生命周期配置

```
<bean id="" class="" scope="" init-method="" destroy-method=""></bean>

* init-method：指定类中的初始化方法名称

* destroy-method：指定类中销毁方法名称
```

5.4 Bean实例化三种方式

- 无参**构造**方法实例化
- 工厂**静态**方法实例化
- 工厂**普通**方法实例化

5.4.1 无参构造方法实例化

它会根据默认无参构造方法来创建类对象，如果bean中没有默认无参构造函数，将会创建失败

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

5.4.2 工厂静态方法实例化

应用场景

依赖的jar包中有个A类，A类中有个静态方法m1，m1方法的返回值是一个B对象。如果我们频繁使用B对象，此时我们可以将B对象的创建权交给spring的IOC容器，以后我们在使用B对象时，无需调用A类中的m1方法，直接从IOC容器获得。

```
public class StaticFactoryBean {

    public static UserDao createUserDao(){
        return new UserDaoImpl();
    }
}
```

```
<bean id="userDao" class="com.lagou.factory.StaticFactoryBean"
      factory-method="createUserDao" />
```

5.4.3 工厂普通方法实例化

应用场景

依赖的jar包中有个A类，A类中有个普通方法m1，m1方法的返回值是一个B对象。如果我们频繁使用B对象，

此时我们可以将B对象的创建权交给spring的IOC容器，以后我们在使用B对象时，无需调用A类中的m1方法，直接从IOC容器获得。

```
public class DynamicFactoryBean {

    public UserDao createUserDao(){
        return new UserDaoImpl();
    }
}
```

```
<bean id="dynamicFactoryBean" class="com.lagou.factory.DynamicFactoryBean"/>
<bean id="userDao" factory-bean="dynamicFactoryBean" factory-
method="createUserDao"/>
```

5.5 Bean依赖注入概述

依赖注入 DI（Dependency Injection）：它是 Spring 框架核心 IOC 的具体实现。

在编写程序时，通过控制反转，把对象的创建交给了 Spring，但是代码中不可能出现没有依赖的情况。IOC 解耦只是降低他们的依赖关系，但不会消除。例如：业务层仍会调用持久层的方法。

那这种业务层和持久层的依赖关系，在使用 Spring 之后，就让 Spring 来维护了。简单的说，就是**通过框架把持久层对象传入业务层**，而不用我们自己去获取。

5.6 Bean依赖注入方式

5.6.1 构造方法

在UserServiceImpl中创建有参构造

```
public class UserServiceImpl implements UserService {

    private UserDao userDao;

    public UserServiceImpl(UserDao userDao) {
        this.userDao = userDao;
    }

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

配置Spring容器调用有参构造时进行注入

```
<bean id="userDao" class="com.lagou.dao.impl.UserDaoImpl"/>
<bean id="userService" class="com.lagou.service.impl.UserServiceImpl">
    <!--<constructor-arg index="0" type="com.lagou.dao.UserDao" ref="userDao"/>-
->
    <constructor-arg name="userDao" ref="userDao"/>
</bean>
```

5.6.2 set方法

在UserServiceImpl中创建set方法

```
public class UserServiceImpl implements UserService {

    private UserDao userDao;

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

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

配置Spring容器调用set方法进行注入

```
<bean id="userDao" class="com.lagou.dao.impl.UserDaoImpl"/>
<bean id="userService" class="com.lagou.service.impl.UserServiceImpl">
    <property name="userDao" ref="userDao"/>
</bean>
```

5.6.3 P命名空间注入

P命名空间注入本质也是set方法注入，但比起上述的set方法注入更加方便，主要体现在配置文件中，如下：

首先，需要引入P命名空间：

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

其次，需要修改注入方式：

```
<bean id="userDao" class="com.lagou.dao.impl.UserDaoImpl"/>
<bean id="userService" class="com.lagou.service.impl.UserServiceImpl"
      p:userDao-ref="userDao"/>
```

5.7 Bean依赖注入的数据类型

上面操作，都是注入Bean对象，除了对象的引用可以注入，普通数据类型和集合都可以在容器中进行注入。

注入数据的三种数据类型

1. **普通数据类型**
2. **引用数据类型**
3. **集合数据类型**

其中引用数据类型，此处就不再赘述了，之前的操作都是对UserDao对象的引用进行注入的。下面将以set方法注入为例，演示普通数据类型和集合数据类型的注入。

5.7.1 注入普通数据类型

```
public class User {
    private String username;
    private String age;

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

    public void setAge(String age) {
        this.age = age;

    }

}
```

```
<bean id="user" class="com.lagou.domain.User">
    <property name="username" value="jack"/>
    <property name="age" value="18"/>
</bean>
```

5.7.2 注入集合数据类型

1）List集合注入

```
public class UserDaoImpl implements UserDao {

    private List<Object> list;

    public void save() {
        System.out.println(list);
        System.out.println("保存成功了...");
    }

}
```

```
<bean id="user" class="com.lagou.domain.User">
    <property name="username" value="jack"/>
    <property name="age" value="18"/>
</bean>
<bean id="userDao" class="com.lagou.dao.impl.UserDaoImpl">
    <property name="list">
        <list>
            <value>aaa</value>
            <ref bean="user"></ref>
        </list>
    </property>
</bean>
```

2）Set集合注入

```
public class UserDaoImpl implements UserDao {

    private Set<Object> set;

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

    public void save() {
        System.out.println(set);
        System.out.println("保存成功了...");
    }
}
```

```
<bean id="user" class="com.lagou.domain.User">
    <property name="username" value="jack"/>
    <property name="age" value="18"/>
</bean>
<bean id="userDao" class="com.lagou.dao.impl.UserDaoImpl">
    <property name="set">
        <list>
            <value>bbb</value>
            <ref bean="user"></ref>
        </list>
    </property>
</bean>
```

3）Array数组注入

```
public class UserDaoImpl implements UserDao {

    private Object[] array;

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

    public void save() {
        System.out.println(Arrays.toString(array));
        System.out.println("保存成功了...");
    }
}
```

```
<bean id="user" class="com.lagou.domain.User">
    <property name="username" value="jack"/>
    <property name="age" value="18"/>
</bean>
<bean id="userDao" class="com.lagou.dao.impl.UserDaoImpl">
    <property name="array">
        <array>
            <value>ccc</value>
            <ref bean="user"></ref>
        </array>
    </property>
</bean>
```

4）Map集合注入

```
public class UserDaoImpl implements UserDao {

    private Map<String, Object> map;

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

    public void save() {
        System.out.println(map);
        System.out.println("保存成功了...");
    }
}
```

```
<bean id="user" class="com.lagou.domain.User">
    <property name="username" value="jack"/>
    <property name="age" value="18"/>
</bean>
<bean id="userDao" class="com.lagou.dao.impl.UserDaoImpl">
    <property name="map">
        <map>
            <entry key="k1" value="ddd"/>
            <entry key="k2" value-ref="user"></entry>
        </map>
    </property>
</bean>
```

5）Properties配置注入

```
public class UserDaoImpl implements UserDao {

    private Properties properties;

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

    public void save() {
        System.out.println(properties);
        System.out.println("保存成功了...");
    }
}
```

```
<bean id="userDao" class="com.lagou.dao.impl.UserDaoImpl">
    <property name="properties">
        <props>
            <prop key="k1">v1</prop>
            <prop key="k2">v2</prop>
            <prop key="k3">v3</prop>
        </props>
    </property>
</bean>
```

5.8 配置文件模块化

实际开发中，Spring的配置内容非常多，这就导致Spring配置很繁杂且体积很大，所以，可以将部分配置拆解到其他配置文件中，也就是所谓的**配置文件模块化**。

1）并列的多个配置文件

```
ApplicationContext act =
                new
ClassPathXmlApplicationContext("beans1.xml","beans2.xml","...");
```

2）主从配置文件

```
<import resource="applicationContext-xxx.xml"/>
```

注意：

- 同一个xml中不能出现相同名称的bean,如果出现会报错
- 多个xml如果出现相同名称的bean，不会报错，但是后加载的会覆盖前加载的bean

5.9 知识小结

Spring的重点配置

```
<bean>标签：创建对象并放到spring的IOC容器
    id属性:在容器中Bean实例的唯一标识，不允许重复
    class属性:要实例化的Bean的全限定名
    scope属性:Bean的作用范围，常用是Singleton(默认)和prototype

<constructor-arg>标签：属性注入
    name属性：属性名称
    value属性：注入的普通属性值
    ref属性：注入的对象引用值

<property>标签：属性注入
    name属性：属性名称
    value属性：注入的普通属性值
    ref属性：注入的对象引用值
    <list>
    <set>
    <array>
    <map>
    <props>

<import>标签:导入其他的Spring的分文件
```

六 DbUtils（IOC实战）

6.1 DbUtils是什么?

DbUtils是Apache的一款用于简化Dao代码的工具类，它底层封装了JDBC技术。

核心对象

```
QueryRunner queryRunner = new QueryRunner(DataSource dataSource);
```

核心方法

```
int update(); 执行增、删、改语句

T query(); 执行查询语句
    ResultSetHandler<T> 这是一个接口，主要作用是将数据库返回的记录封装到实体对象
```

举个栗子

查询数据库所有账户信息到Account实体中

```
public class DbUtilsTest {

    @Test
    public void findAllTest() throws Exception {
        // 创建DBUtils工具类，传入连接池
        QueryRunner queryRunner = new QueryRunner(JdbcUtils.getDataSource());
        // 编写sql
        String sql = "select * from account";
        // 执行sql
        List<Account> list = queryRunner.query(sql, new BeanListHandler<Account>
(Account.class));
        // 打印结果
        for (Account account : list) {
            System.out.println(account);
        }
    }
}
```

6.2 Spring的xml整合DbUtils

6.1 介绍

需求

基于Spring的xml配置实现账户的CRUD案例

步骤分析

1. 准备数据库环境
2. 创建java项目，导入坐标
3. 编写Account实体类
4. 编写AccountDao接口和实现类
5. 编写AccountService接口和实现类
6. 编写spring核心配置文件
7. 编写测试代码

6.2 实现

1）准备数据库环境

```
CREATE DATABASE `spring_db`;

USE `spring_db`;

CREATE TABLE `account` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(32) DEFAULT NULL,
  `money` double DEFAULT NULL,
  PRIMARY KEY (`id`)
) ;

insert  into `account`(`id`,`name`,`money`) values (1,'tom',1000),
(2,'jerry',1000);
```

2）创建java项目，导入坐标

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.9</version>
    </dependency>
    <dependency>
        <groupId>commons-dbutils</groupId>
        <artifactId>commons-dbutils</artifactId>
        <version>1.6</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
```

```
        <version>4.12</version>
    </dependency>
</dependencies>
```

3）编写Account实体类

```
public class Account {

    private Integer id;
    private String name;
    private Double money;

}
```

4）编写AccountDao接口和实现类

```
public interface AccountDao {

    public List<Account> findAll();

    public Account findById(Integer id);

    public void save(Account account);

    public void update(Account account);

    public void delete(Integer id);
}
```

```
public class AccountDaoImpl implements AccountDao {

    private QueryRunner queryRunner;

    public void setQueryRunner(QueryRunner queryRunner) {
        this.queryRunner = queryRunner;
    }

    @Override
    public List<Account> findAll() {
        List<Account> list = null;
        // 编写sql
        String sql = "select * from account";
        try {
            // 执行sql
            list = queryRunner.query(sql, new BeanListHandler<Account>
(Account.class));
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return list;
    }
```

```
    @Override
    public Account findById(Integer id) {
        Account account = null;
        // 编写sql
        String sql = "select * from account where id = ?";
        try {
            // 执行sql
            account = queryRunner.query(sql, new BeanHandler<Account>
(Account.class), id);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return account;
    }

    @Override
    public void save(Account account) {
        // 编写sql
        String sql = "insert into account values(null,?,?)";
        // 执行sql
        try {
            queryRunner.update(sql, account.getName(), account.getMoney());
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void update(Account account) {
        // 编写sql
        String sql = "update account set name = ?,money = ? where id = ?";
        // 执行sql
        try {
            queryRunner.update(sql, account.getName(),
account.getMoney(),account.getId());
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void delete(Integer id) {
        // 编写sql
        String sql = "delete from account where id = ?";
        // 执行sql
        try {
            queryRunner.update(sql, id);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

5）编写AccountService接口和实现类

```
public interface AccountService {

    public List<Account> findAll();

    public Account findById(Integer id);

    public void save(Account account);

    public void update(Account account);

    public void delete(Integer id);
}
```

```
public class AccountServiceImpl implements AccountService {

    private AccountDao accountDao;

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

    @Override
    public List<Account> findAll() {
        return accountDao.findAll();
    }

    @Override
    public Account findById(Integer id) {
        return accountDao.findById(id);
    }

    @Override
    public void save(Account account) {
        accountDao.save(account);
    }

    @Override
    public void update(Account account) {
        accountDao.update(account);
    }

    @Override
    public void delete(Integer id) {
        accountDao.delete(id);
    }
}
```

6）编写spring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"
```

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


    <!--把数据库连接池交给IOC容器-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver">
</property>
        <property name="url" value="jdbc:mysql://localhost:3306/spring_db">
</property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>
    </bean>
    <!--把QueryRunner交给IOC容器-->
    <bean id="queryRunner" class="org.apache.commons.dbutils.QueryRunner">
        <constructor-arg name="ds" ref="dataSource"></constructor-arg>
    </bean>
    <!--把AccountDao交给IOC容器-->
    <bean id="accountDao" class="com.lagou.dao.impl.AccountDaoImpl">
        <property name="queryRunner" ref="queryRunner"></property>
    </bean>
    <!--把AccountService交给IOC容器-->
    <bean id="accountService" class="com.lagou.service.impl.AccountServiceImpl">
        <property name="accountDao" ref="accountDao"></property>
    </bean>
</beans>
```

7）编写测试代码

```
public class AccountServiceTest {

    ApplicationContext applicationContext =
            new ClassPathXmlApplicationContext("applicationContext.xml");
    AccountService accountService =
applicationContext.getBean(AccountService.class);

    //测试保存
    @Test
    public void testSave() {
        Account account = new Account();
        account.setName("lucy");
        account.setMoney(100d);
        accountService.save(account);
    }

    //测试查询
    @Test
    public void testFindById() {
        Account account = accountService.findById(3);
        System.out.println(account);
    }

    //测试查询所有
    @Test
```

```
    public void testFindAll() {
        List<Account> accountList = accountService.findAll();
        for (Account account : accountList) {
            System.out.println(account);
        }
    }

    //测试修改
    @Test
    public void testUpdate() {
        Account account = new Account();
        account.setId(3);
        account.setName("jack");
        account.setMoney(2000d);
        accountService.update(account);
    }

    //测试删除
    @Test
    public void testDelete() {
        accountService.delete(3);
    }
}
```

8）抽取jdbc配置文件

applicationContext.xml加载jdbc.properties配置文件获得连接信息。

首先，需要引入context命名空间和约束路径：

```
* 命名空间：
    xmlns:context="http://www.springframework.org/schema/context"

* 约束路径：
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd
```

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

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

6.3 知识小结

```
* DataSource的创建权交由Spring容器去完成

* QueryRunner的创建权交由Spring容器去完成，使用构造方法传递DataSource

* Spring容器加载properties文件
    <context:property-placeholder location="xx.properties"/>
    <property name="" value="${key}"/>
```

七 Spring注解开发

Spring是轻代码而重配置的框架，配置比较繁重，影响开发效率，所以注解开发是一种趋势，注解代替xml配置文件可以简化配置，提高开发效率。

7.1 Spring常用注解

7.1.1 介绍

Spring常用注解主要是替代`<bean>`的配置

注解	说明
@Component	使用在类上用于实例化Bean
@Controller	使用在web层类上用于实例化Bean
@Service	使用在service层类上用于实例化Bean
@Repository	使用在dao层类上用于实例化Bean
@Autowired	使用在字段上用于根据类型依赖注入
@Qualifier	结合@Autowired一起使用,根据名称进行依赖注入
@Resource	相当于@Autowired+@Qualifier，按照名称进行注入
@Value	注入普通属性
@Scope	标注Bean的作用范围
@PostConstruct	使用在方法上标注该方法是Bean的初始化方法
@PreDestroy	使用在方法上标注该方法是Bean的销毁方法

说明:

JDK11以后完全移除了javax扩展导致不能使用@resource注解

需要maven引入依赖

```
  <dependency>
       <groupId>javax.annotation</groupId>
        <artifactId>javax.annotation-api</artifactId>
       <version>1.3.2</version>
 </dependency>
```

注意

使用注解进行开发时，需要在applicationContext.xml中配置组件扫描，作用是指定哪个包及其子包下的Bean需要进行扫描以便识别使用注解配置的类、字段和方法。

```
<!--注解的组件扫描-->
<context:component-scan base-package="com.lagou"></context:component-scan>
```

7.1.2 实现

1）Bean实例化（IOC）

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

使用@Compont或@Repository标识UserDaoImpl需要Spring进行实例化。

```
// @Component(value = "userDao")
@Repository // 如果没有写value属性值，Bean的id为：类名首字母小写
public class UserDaoImpl implements UserDao {

}
```

2）属性依赖注入（DI）

```
<bean id="userService" class="com.lagou.service.impl.UserServiceImpl">
    <property name="userDao" ref="userDaoImpl"/>
</bean>
```

使用@Autowired或者@Autowired+@Qulifier或者@Resource进行userDao的注入

```
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDao userDao;

    // <property name="userDao" ref="userDaoImpl"/>
    // @Autowired
    // @Qualifier("userDaoImpl")
    // @Resource(name = "userDaoImpl")
```

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

3）@Value

使用@Value进行字符串的注入，结合SPEL表达式获得配置参数

```
@Service
public class UserServiceImpl implements UserService {

    @Value("注入普通数据")
    private String str;
    @Value("${jdbc.driver}")
    private String driver;

}
```

4）@Scope

```
<bean scope=""/>
```

使用@Scope标注Bean的范围

```
@Service
@Scope("singleton")
public class UserServiceImpl implements UserService {{

}
```

5）Bean生命周期

```
<bean init-method="init" destroy-method="destory" />
```

使用@PostConstruct标注初始化方法，使用@PreDestroy标注销毁方法

```
@PostConstruct
public void init(){
    System.out.println("初始化方法....");
}

@PreDestroy
public void destroy(){
    System.out.println("销毁方法.....");
}
```

7.2 Spring常用注解整合DbUtils

步骤分析

1. 拷贝xml配置项目，改为注解配置项目
2. 修改AccountDaoImpl实现类
3. 修改AccountServiceImpl实现类
4. 修改spring核心配置文件
5. 编写测试代码

1）拷贝xml配置项目，改为常用注解配置项目

过程略....

2）修改AccountDaoImpl实现类

```
@Repository
public class AccountDaoImpl implements AccountDao {

    @Autowired
    private QueryRunner queryRunner;

    ....
}
```

3）修改AccountServiceImpl实现类

```
@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;

    ....
}
```

4）修改spring核心配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w1.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">

    <!--注解的组件扫描-->
```

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

    <!--加载jdbc配置文件-->
    <context:property-placeholder location="classpath:jdbc.properties">
</context:property-placeholder>

    <!--把数据库连接池交给IOC容器-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${jdbc.driver}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
    </bean>
    <!--把QueryRunner交给IOC容器-->
    <bean id="queryRunner" class="org.apache.commons.dbutils.QueryRunner">
        <constructor-arg name="ds" ref="dataSource"></constructor-arg>
    </bean>
</beans>
```

5）编写测试代码

```
public class AccountServiceTest {

    ApplicationContext applicationContext =
            new ClassPathXmlApplicationContext("applicationContext.xml");
    AccountService accountService =
applicationContext.getBean(AccountService.class);

    //测试查询
    @Test
    public void findByIdTest() {
        Account account = accountService.findById(3);
        System.out.println(account);
    }
}
```

7.3 Spring新注解

使用上面的注解还不能全部替代xml配置文件，还需要使用注解替代的配置如下：

```
* 非自定义的Bean的配置：<bean>

* 加载properties文件的配置：<context:property-placeholder>

* 组件扫描的配置：<context:component-scan>

* 引入其他文件：<import>
```

注解	说明
@Configuration	用于指定当前类是一个Spring 配置类，当创建容器时会从该类上加载注解
@Bean	使用在方法上，标注将该方法的返回值存储到 Spring 容器中
@PropertySource	用于加载 properties 文件中的配置
@ComponentScan	用于指定 Spring 在初始化容器时要扫描的包
@Import	用于导入其他配置类

7.4 Spring纯注解整合DbUtils

步骤分析

1. 编写Spring核心配置类
2. 编写数据库配置信息类
3. 编写测试代码

1）编写Spring核心配置类

```
@Configuration
@ComponentScan("com.lagou")
@Import(DataSourceConfig.class)
public class SpringConfig {

    @Bean("queryRunner")
    public QueryRunner getQueryRunner(@Autowired DataSource dataSource) {
        return new QueryRunner(dataSource);
    }
}
```

2）编写数据库配置信息类

```
@PropertySource("classpath:jdbc.properties")
public class DataSourceConfig {
    @Value("${jdbc.driver}")
    private String driver;
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
    private String password;

    @Bean("dataSource")
    public DataSource getDataSource() {
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setDriverClassName(driver);
```

```
        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        return dataSource;
    }
}
```

3）编写测试代码

```
public class AccountServiceTest {

    ApplicationContext applicationContext =
                        new
AnnotationConfigApplicationContext(SpringConfig.class);
    AccountService accountService =
applicationContext.getBean(AccountService.class);

    //测试查询
    @Test
    public void testFindById() {
        Account account = accountService.findById(3);
        System.out.println(account);
    }
}
```

八 Spring整合Junit

8.1普通Junit测试问题

在普通的测试类中，需要开发者手动加载配置文件并创建Spring容器，然后通过Spring相关API获得Bean实例；如果不这么做，那么无法从容器中获得对象。

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

AccountService accountService =
applicationContext.getBean(AccountService.class);
```

我们可以让SpringJunit负责创建Spring容器来简化这个操作，开发者可以直接在测试类注入Bean实例；但是需要将配置文件的名称告诉它。

8.2 Spring整合Junit

步骤分析

1. 导入spring集成Junit的坐标
2. 使用@Runwith注解替换原来的运行器
3. 使用@ContextConfiguration指定配置文件或配置类
4. 使用@Autowired注入需要测试的对象
5. 创建测试方法进行测试

1）导入spring集成Junit的坐标

```
<!--此处需要注意的是，spring5 及以上版本要求 junit 的版本必须是 4.12 及以上-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.1.5.RELEASE</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

2）使用@Runwith注解替换原来的运行器

```
@RunWith(SpringJUnit4ClassRunner.class)
public class SpringJunitTest {

}
```

3）使用@ContextConfiguration指定配置文件或配置类

```
@RunWith(SpringJUnit4ClassRunner.class)
//@ContextConfiguration(value = {"classpath:applicationContext.xml"}) 加载spring核心配置文件
@ContextConfiguration(classes = {SpringConfig.class}) // 加载spring核心配置类
public class SpringJunitTest {

}
```

4）使用@Autowired注入需要测试的对象

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {SpringConfig.class})
public class SpringJunitTest {

    @Autowired
    private AccountService accountService;
}
```

5）创建测试方法进行测试

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {SpringConfig.class})
public class SpringJunitTest {

    @Autowired
    private AccountService accountService;

    //测试查询
    @Test
    public void testFindById() {
        Account account = accountService.findById(3);
        System.out.println(account);
    }
}
```