day42_Spring第二天

今日学习目标

- [ ] 掌握IOC的注解开发
- [ ] 掌握DI的注解开发
- [ ] 掌握纯注解配置
- [ ] 了解注解背后的处理逻辑
- [ ] 掌握动态代理

一、基于注解的IOC和DI

1. 快速入门（重点）

需求描述

- 有dao层：`UserDao`和`UserDaoImpl`
- 有service层：`UserService`和`UserServiceImpl`
- 使用注解配置bean，并注入依赖

需求分析

1. 准备工作：创建Maven项目，导入依赖坐标
2. 编写代码并注解配置：

   编写dao层、service层代码，使用注解`@Component`配置bean：代替xml里的`bean`标签

   使用注解`@Autowired`依赖注入：代替xml里的`property`和`constructor-arg`标签
3. 在配置文件中开启组件扫描
4. 测试

需求实现

1) 准备工作

- 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) 编写代码，并注解配置

- `UserDao` 接口

```
package com.itheima.dao;

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

- `UserDaoImpl` 实现类

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

import com.itheima.dao.UserDao;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;

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

- `UserService` 接口

```
package com.itheima.service;

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

- `UserServiceImpl` 实现类

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

import com.itheima.dao.UserDao;
import com.itheima.service.UserService;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Controller;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;


@Component
public class UserServiceImpl implements UserService {

    //告诉spring，要把UserDaoImpl的对象给注入进来
    @Autowired
    private UserDao userDao;
```

```
    public void add() {
        System.out.println("userServiceImpl..add...");
        userDao.add();
    }
}
```

3) 开启组件扫描

- 创建`applicationContext.xml`，注意引入的`context`名称空间

```
<?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">

    <!--开启组件扫描-->
    <context:component-scan base-package="com.itheima"/>
</beans>
```

4) 功能测试

- 创建一个测试类，调用Service

```
public class TestUserServiceImpl {


    @Test
    public void testAdd(){

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

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

        //3. 调用方法
        us.add();

    }
}
```

步骤小结

1. 导入依赖
2. 定义接口和实现类（dao 和 service）
3. 在实现类上面打上注解 @Component
4. 在属性上面打上注解@AutoWired

5. 在applicationContext.xml里面打开扫描的开关

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

2. 注解使用详解

2.1 开启组件扫描

- 在Spring中，如果要使用注解开发，就需要在`applicationContext.xml`中开启组件扫描，配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       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. 打开注解的扫描开关
            1.1 因为我们创建spring工厂的时候，解读的是applicationContext.xml 这个文件。
            1.2 但是这个文件里面已经不再写 <bean>标签，那么此时spring就不知道要创建哪个类的对象
            1.3 虽然在具体的类身上打上了注解 @Component ，但是spring的工厂不知道这些类
            1.4 所以需要指定让spring去扫描具体的包，以便它识别这些包里面类身上的注解，然后托管这些类。
            1.5 如果想指定具体的包，可以写多个，然后使用空格来间隔， 也可以只写到父亲的包。
    -->

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

</beans>
```

2.2 声明bean的注解【IOC】

简介

注解	说明
`@Component`	用在类上，相当于bean标签
`@Controller`	用在web层类上，配置一个bean（是`@Component`的衍生注解）
`@Service`	用在service层类上，配置一个bean（是`@Component`的衍生注解）
`@Repository`	用在dao层类上，配置一个bean（是`@Component`的衍生注解）

- `@Component`：类级别的一个注解，用于声明一个bean，使用不多
  - `value`属性：bean的唯一标识 （id值）。如果不配置，默认以首字母小写的类名为id
- `@Controller, @Service, @Repository`，作用和`@Component`完全一样，但更加的语义化，使用更多
  - `@Controller`：用于web层的bean
  - `@Service`：用于Service层的bean

- `@Repository`：用于dao层的bean

示例

- `UserDaoImpl`类上使用注解`@Repository`

```
@Repository("userDao")
public class UserDaoImpl implements UserDao{
}
```

- `UserServiceImpl`类上使用注解`@Service`

```
@Service("userService")
public class UserServiceImpl implements UserService{
}
```

- `UserController`类上使用注解`@Controller`

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

2.3 配置bean的注解 【IOC】

注解	说明
`@Scope`	相当于bean标签的`scope`属性
`@PostConstruct`	相当于bean标签的`init-method`属性
`@PreDestroy`	相当于bean标签的`destory-method`属性

配置bean的作用范围：

- `@Scope`：配置bean的作用范围，相当于bean标签的scope属性。加在bean对象上
- `@Scope`的常用值有：
- `singleton`：单例的，容器中只有一个该bean对象
  - 何时创建：容器初始化时
  - 何时销毁：容器关闭时
- `prototype`：多例的，每次获取该bean时，都会创建一个bean对象
  - 何时创建：获取bean对象时
  - 何时销毁：长时间不使用，垃圾回收

```
@Scope("prototype")
@Service("userService")
public class UserServiceImpl implements UserService{
    //...
}
```

配置bean生命周期的方法

- `@PostConstruct`是方法级别的注解，用于指定bean的初始化方法
- `@PreDestroy`是方法级别的注解，用于指定bean的销毁方法

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

    @PostConstruct
    public void init(){
        System.out.println("UserServiceImpl对象已经创建了......");
    }

    @PreDestroy
    public void destroy(){
        System.out.println("UserServiceImpl对象将要销毁了......");
    }

    //......
}
```

2.4 依赖注入的注解【DI】

注解	说明
`@Autowired`	相当于property标签的ref 注入对象
`@Qualifier`	结合`@Autowired`使用，用于根据名称(标识符)注入依赖
`@Resource`	相当于`@Autowired + @Qualifier`
`@Value`	相当于property标签的value ， 注入普通 的属性

注入bean对象

- `@Autowired`：用于byType注入bean对象，按照依赖的类型，从Spring容器中查找要注入的bean对象
    - 1. 如果找到一个，直接注入
    - 2. 如果找到多个，则以变量名为id，查找bean对象并注入
        1. 如果找不到，抛异常
- `@Qualifier`：是按id注入，但需要和`@Autowired`配合使用。
- `@Resource`：(是jdk提供的)用于注入bean对象(byName注入)，相当于`@Autowired + @Qualifier`

> **绝大多数情况下，只要使用`@Autowired`注解注入即可**
>
> **使用注解注入时，不需要set方法了**

- UserDao

```
package com.itehima.dao;

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

- UserDao实现

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

import com.itheima.dao.UserDao;
import org.springframework.stereotype.Component;

//组件  告诉spring，我们打算把这个类交给spring托管（创建对象）
@Component("ud02")
public class UserDaoImpl02 implements UserDao {
    public void add() {
        System.out.println("调用了UserdaoImpl02的add方法~! ~");
    }
}
```

- UserService

```
package com.itehima.service;

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

- UserService实现

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

import com.itheima.dao.UserDao;
import com.itheima.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.annotation.Resource;


/*
    一、 声明Bean的注解：
        @Component ：
            1. 这是通用的注解 ， spring只要看到这个注解，就会把这个类的对象创建出来，然后管理起来。
            2. spring针对三层结构，设计出来每一层特有的注解
                web ---- @Controller
                service ---- @Service
                dao  ---- @Repository
            3. id属性的设置
                3.1 可以使用注解里面的value属性来设置id值
                3.2 如果不设置的话，默认一类的名字(首字母小写) ， 作为id值。

      二、配置Bean的注解
```

```
            @Scope：
                1. 用来配置单例或者多例，默认情况下，spring创建的对象都是单例的。
                2. 如果期望做成多例，就应该写成这样： @Scope("prototype")
                3. 如果期望明码标价成单例： @Scope("singleton")
            @PostConstruct
                当对象创建的时候调用打上这个注解的方法
            @PreDestroy
                 当对象销毁的时候调用打上这个注解的方法
 */

//@Component
@Service("us")
@Scope("singleton")
public class UserServiceImpl implements UserService {


    /*
        @Autowired :
            作用： 注入对象到属性身上，自动注入，是根据类型去匹配对象的。
            用法：
                1. 在spring容器里面查找有没有哪个对象是属于（属性）这种类型的，如果有就直接注入。
                2. 如果不巧，在容器里面属于（属性）这种类型的对象有多个，那么还会挣扎一下
                    2.1 拿属性（变量）的名字，当成id的名字去找对象。如果能找到就注入进来
                    2.2 如果还没有找到匹配的，就直接报异常。
         @Qualifier
            作用： 配合@Autowired 来用的。
            用法：
                1. 当我们spring容器里面存在多个对象的时候 ， @Autowired 难以抉择。
                2. 此时就可以使用@Qualifier 来指定id，告诉 @Autowired 要注入谁！
         @Resource
            作用： 用来注入对象，当出现多个对象的时候，可以使用@Resource来注入
            用法：
                1. 它等价 @Autowired + @Qualifier
                2. 它是按照id的名字去找对象，然后注入进来。

            @Value
                作用： 用来注入普通数据，一般是用来注入外部配置文件（如：properties）的内容
                用法： @Value("${key的名字}")

     */

    /*@Autowired
    @Qualifier("userDaoImpl03")
    private UserDao userDao;*/


    @Resource(name = "userDaoImpl")
    private UserDao userDao;


    @Value("北京")
    private String address ;

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

```
    //=========================
    // 当对象创建的时候，调用这个方法
    @PostConstruct
    public void init(){
        System.out.println("调用了UserServiceImpl...init...~");
    }

    //当对象销毁的时候，调用这个方法
    @PreDestroy
    public void destroy(){
        System.out.println("调用了UserServiceImpl...destroy...~");
    }
}
```

- 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 TestUserServiceImpl {

    @Test
    public void testAdd(){

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

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

        //3. 调用方法
        us.add();


        //4. 关闭工厂
        context.close();

    }
}
```

注入普通值

- `@Value`：注入简单类型值，例如：基本数据类型和String

```java
@Service("userService")
public class UserServiceImpl implements UserService{

    @Value("zhangsan")//直接注入字符串值
    private String name;

    //从properties文件中找到key的值，注入进来
    //注意：必须在applicationContext.xml中加载了properties文件才可以使用
    @Value("${properties中的key}")
    private String abc;

    //...
}
```

小结

- 在xml里要开启组件扫描

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

- 声明bean的注解（注册bean的注解） | IOC的注解
  - `@Component("bean的名称")`， 括号里面bean的名称就是id 值， 可以用在任何类上，注册bean对象
  - `@Controller("bean名称")`, `@Service("bean名称")`, `@Repository("bean名称")`，分别用于web层、service层、dao层的类上
- 配置bean的注解
  - 如果要给一个bean设置作用范围：在bean上加注解`@Scope("singleton/prototype")`
  - 如果要给一个bean设置一个初始化方法：就在方法上加注解`@PostConstruct`
  - 如果要给一个bean设置一个销毁方法：就在方法上加注解`@PreDestroy`
- 依赖注入的注解
  - `@Autowired`：byType注入，直接加在依赖项那个成员变量上
    - Spring会根据类型，从容器里查找bean并注入进来
    - 如果只找到一个：直接注入
    - 如果找到了多个：根据属性的名字，把它当成id去找对象注入
    - 如果找不到适合的就会报错
  - `@Autowired + @Qualifier("要注入的bean的名称")`： 这种组合一般不怎么用，因为比较麻烦。
  - `@Resource(name="要注入的bean的名称")`：byName注入
  - `@Value("要注入的简单值")`：用于注入简单值
    - `@Value("${properties里的key}")`：把properties里指定key的值注入进来。**前提条件是必须已经引入了properties文件**

二、注解方式CURD练习

需求描述

- 使用注解开发帐号信息的CURD功能

需求分析

- 使用注解代替某些XML配置，能够代替的有：service层和dao层里面的类可以使用注解来托管。
  - dao层bean对象，可以在类上增加注解`@Repository`
  - service层bean对象，可以在类上增加注解`@Service`
  - Service层依赖于dao层，可以使用注解注入依赖`@AutoWired`
- 不能使用注解代替，仍然要使用XML配置的的有：
  - QueryRunner的bean对象，是DBUtils工具包里提供的类，我们不能给它的源码上增加注解
  - 连接池的bean对象，是c3p0工具包里提供的类，我们不能修改源码增加注解

需求实现

- 导入依赖

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

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

JavaBean

```
package com.itheima.bean;

import lombok.Data;

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

dao层代码

- `AccountDao`接口

```
package com.itheima.dao;

import com.itheima.bean.Account;

import java.util.List;

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

- `AccountDaoImpl`实现类

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

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 org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
```

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

@Repository
public class AccountDaoImpl implements AccountDao {

    @Autowired
    private QueryRunner runner ;

    public int add(Account account) throws SQLException {
        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));
    }
}
```

service层代码

- `AccountService`接口

```
    package com.itheima.service;

    import com.itheima.bean.Account;

    import java.util.List;

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

- `AccountServiceImpl`接口

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

import com.itheima.bean.Account;
import com.itheima.dao.AccountDao;
import com.itheima.service.AccountService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

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

@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao dao;

    public int add(Account account) throws SQLException {
        return dao.add(account);
    }

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

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

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

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

提供配置

- db.properties

```properties
driverClass=com.mysql.jdbc.Driver
jdbcUrl=jdbc:mysql://localhost:3306/day41_spring
user=root
password=root
```

- 创建`applicationContext.xml`

```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. 打开组件的扫描开关-->
    <context:component-scan base-package="com.itheima"/>


    <!--
        2.让spring管理QueryRunner
            以前创建QueryRunner：  QueryRunner runner = new
QueryRunner(C3P0Utils.getDataSource())
    -->
    <bean id="runner" class="org.apache.commons.dbutils.QueryRunner">
        <constructor-arg name="ds" ref="ds"/>
    </bean>


    <!--3. 让spring管理DataSource-->
    <context:property-placeholder location="db.properties"/>
    <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>


</beans>
```

功能测试

```
package com.itheima.test;

import com.itheima.bean.Account;
import com.itheima.service.AccountService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.sql.SQLException;

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

    @Autowired
    private AccountService as;

    @Test
    public void testAdd() throws SQLException {

        Account a = new Account();
```

```
        a.setName("卡特琳娜");
        a.setMoney(100);

        as.add(a);
    }
    @Test
    public void testDelete() throws SQLException {
        as.delete(5);
    }

    @Test
    public void testUpdate() throws SQLException {

        Account a = as.findById(3);
        a.setMoney(1);


        as.update (a);
    }


    @Test
    public void testFindAll() throws SQLException {
        System.out.println(as.findAll());
    }
}
```

小结

1. 导入依赖
2. 定义dao接口和service 接口 各自的实现
3. 使用注解托管dao的实现类和service的实现类 @Repository@Service
4. 不要忘记了在applicationContext.xml 打开开关
5. 在applicationContext.xml里面托管QueryRunner，这样才能给Dao里面注入
6. 在applicationContext.xml里面托管ComboPooledDataSource 这样才能给QueryRunner注入

三、纯注解开发IOC和DI

在上边的CURD练习中，仍然有部分配置需要使用`applicationContext.xml`，那么能不能使用注解替换掉所有的xml呢？Spring提供了一些新注解，可以达到这个目标。

> **请注意：Spring提供的这部分新注解，并非为了完全代替掉XML，只是提供了另外一种配置方案**

注解简介

注解	说明
`@Configuration`	被此注解标记的类，是配置类 等同于applicationContext.xml
`@ComponentScan`	用在配置类上，开启注解扫描。使用basePackage属性指定扫描的包
`@PropertySource`	用在配置类上，加载properties文件。使用value属性指定properties文件路径
`@Import`	用在配置类上，引入子配置类。用value属性指定子配置类的Class
`@Bean`	用在配置类的方法上，把返回值声明为一个bean。用name/value属性指定bean的id

注解详解

1 `@Configuration`配置类

- `@Configuration`把一个Java类声明为核心配置类
  - 加上Java类上，这个Java类就成为了Spring的核心配置类，用于代替`applicationContext.xml`
  - 是`@Component`的衍生注解，所以：核心配置类也是bean，也会被spring管理起来，当然里边也可以注入依赖

```
/*
1. 让这个类先成为核心配置类
            打上注解 @Configuration ，它是从@Component注解衍生出来的，所以这个核心配置类
                也会被spring管理起来。那么即表示在这个核心配置类里面，我们同样可以让spring
                注入进来其他的对象，或者properties的内容
 */
@Configuration
public class AppConfig {
}
```

2 配置类上的注解

- `@ComponentScan`配置组件注解扫描
  - `basePackages`或者`value`属性：指定扫描的基本包
  - 等同于替代了applicationContext.xml里面的这句话

```
<!--1. 打开包的扫描开关-->
 <context:component-scan base-package="com.itheima"/>
```

- `@PropertySource`用于加载properties文件
  - `value`属性：指定propertis文件的路径，从类加载路径里加载
  - 等同于替代了applicationContext.xml里面的这句话

```
<!--导入properties文件-->
<context:property-placeholder location="classpath:db.properties"/>
```

- `@Import`用于导入其它配置类
  - Spring允许提供多个配置类（模块化配置），在核心配置类里加载其它配置类

    - 相当于`xml`中的`<import resource="模块化xml文件路径"/>`标签
- 核心配置类

```
/*
    1. 这是一个核心配置类，等同于applicationContext.xml
    2. 在applicationCotnext.xml里面能做的配置，在这个类里面也一样可以实现。
    3. 注解解释：
        @Configuration : 标记这这个类是核心配置类，创建工厂的时候，要来找这个类。
        @ComponentScan :  用于扫描包，这些包下的类都有注解的。
        @PropertySource : 用于导入外部的properties文件，注解的括号里面，只要写properties的名字即可
                          spring会在类路径底下找文件
        @Import :  用于引入其他的配置类。因为spring运行分模块开发，配置文件可以分为多个。
 */
@Configuration
@ComponentScan("com.itheima")
@Import(AppConfig01.class)
public class AppConfig {
}
```

- 子配置类

```
@PropertySource("classpath:db.properties")
public class AppConfig01 {
}
```

3 `@Bean`声明bean

1) `@Bean`定义bean

- `@Bean`注解：方法级别的注解
    - 作用： 把方法返回值声明成为一个bean，作用相当于`<bean>`标签，这个方法的返回值将会被Spring管理起来。

```
//如果我们想使用全注解的开发方式，那么比如QueryRunner这些类怎么处理呢？
//1. 这些类是jar包里面的源码类，咱们是无法打注解的，但是又想让spring来管理QueryRunner。
// 1.1 我们可以手动创建QueryRunner，
// 1.2 可以让Spring管理我们创建QueryRunner对象

@Bean
public QueryRunner aa(){
    return new QueryRunner();
}
```

    - @Bean注解可以写在方法上，这些方法可以写在核心配置类里面，也可以写在其他的组件类里面，但是一般会写在核心配置类里面。
- `@Bean`注解的属性：
    - `value`属性：**bean的id。如果不设置，那么方法名就是bean的id**

```
@Configuration
@ComponentScan("com.itheima")
public class AppConfig {

        //1. 把一个方法的返回交给spring管理，方法要返回一个对象。
    @Bean("stu")
    public Student a(){
        Student s = new Student(1 , "张三" , 18);
        return s;
    }

}
```

2) `@Bean`的依赖注入

- `@Bean`注解的方法可以有任意参数，这些参数即是bean所需要的依赖，默认采用byType方式注入
- 可以在方法参数上增加注解`@Qualifier`，用于byName注入

3) 完整代码

```
package com.itheima.config;

/*
    这是一个核心配置类，它的作用就是顶替掉 applicationContext.xml
        1. 让这个类先成为核心配置类
            打上注解 @Configuration , 它是从@Component注解衍生出来的，所以这个核心配置类
                也会被spring管理起来。那么即表示在这个核心配置类里面，我们同样可以让spring
                注入进来其他的对象，或者properties的内容
        2. 扫描具体的包
            @ComponentScan("com.itheima")
        3. 导入外部的properties文件
            3.1 @PropertySource("classpath:aa.properties")  value属性，指定配置文件
的名字即可，也可以加上 classpath:前缀
            3.2 需要先把properties里面的内容，注入到属性身上，然后才能使用它们 :
                    @Value("${password}")
                    private String password;
        4. 引入外部的子配置类
            4.1 当我们的配置的内容有点多的时候，可以考虑拆分配置类，分成多个子配置类的写法
            4.2 那么需要在核心配置类里面引入其他的子配置类
                只有一个子配置类
                @Import(AppConfig01.class)

                有多个子配置类
                @Import(value = {AppConfig01.class,AppConfig02.class })

        5. 把方法的返回值声明成一个Bean
            5.1 只要在方法上打上一个注解@Bean，那么这个方法的返回值就会被Spring管理起来
            5.2 默认情况下，如果不指定id值，那么管理的对象的id值就是方法的名字
            5.3 当然我们也可以设置value属性，来设置id值

            5.4 其实方法也可以让spring注入进来其他的对象，只要方法的声明参数即可。
                默认spring是按照类型来自动注入的，等同于参数的前面有一个 @AutoWired 一样。
                如果遇到多个对象，就无法自动注入了，那么可以使用
@Qualifier("createDataSource02")
                来指定注入具体的对象
 */
```

```java
import com.itheima.bean.Student;
import com.mchange.v2.c3p0.ComboPooledDataSource;
import org.apache.commons.dbutils.QueryRunner;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.*;

import javax.sql.DataSource;
import java.beans.PropertyVetoException;

@Configuration
//@ComponentScan("com.itheima")
//@PropertySource("aa.properties")

@Import(value = {AppConfig01.class,AppConfig02.class })
public class AppConfig {

    @Value("${driverClass}")
    private String driverClass;

    @Value("${jdbcUrl}")
    private String jdbcUrl;

    @Value("${user}")
    private String user;

    @Value("${password}")
    private String password;

    /**
     * 打印以上的四个属性
     */
    public void show(){
        System.out.println(driverClass);
        System.out.println(jdbcUrl);
        System.out.println(user);
        System.out.println(password);
    }

    //----------------------下面演示@Bean------------------

    //1. 把一个方法的返回交给spring管理，方法要返回一个对象。
    @Bean("stu")
    public Student a(){
        Student s = new Student(1 , "张三" , 18);
        return s;
    }

    //2. 让spring托管DataSource
    @Bean
    public DataSource createDataSource() throws PropertyVetoException {
        ComboPooledDataSource ds = new ComboPooledDataSource();
        ds.setDriverClass(driverClass);
        ds.setJdbcUrl(jdbcUrl);
        ds.setUser(user);
        ds.setPassword(password);
        System.out.println("ds=" + ds);
```

```
        return ds;
    }

    @Bean
    public DataSource createDataSource02() throws PropertyVetoException {
        ComboPooledDataSource ds = new ComboPooledDataSource();
        ds.setDriverClass(driverClass);
        ds.setJdbcUrl(jdbcUrl);
        ds.setUser(user);
        ds.setPassword(password);
        System.out.println("ds=" + ds);
        return ds;
    }


    //3. 让spring托管QueryRunner
    @Bean
    public QueryRunner createQueryRunner(@Qualifier("createDataSource02")
DataSource ds){
        System.out.println("ds2=" + ds);
        QueryRunner runner = new QueryRunner(ds);
        return runner;
    }
}
```

小结

- 配置类上要加注解`@Configuration`变成核心配置类，主要是用来替代applicationContext.xml
- 要开启组件扫描，在配置类上`@ComponentScan("com.itheima")`
- 如果要把jar包里的类注册成bean：
  - 在配置类里加方法，方法上加@Bean，会把方法返回值注册bean对象
- 如果要引入外部的properties文件，在配置类上加`@PropertySource("classpath:xxx.properties")`
- 引入模块配置类，在配置类上使用`@Import(子配置类.class)`
- @Bean ，如果期望让spring来管理某个方法的返回值（注意： 这个返回值必须得是一个对象，不能是一个普通的数据，比如：数字、字符串...）

四、纯注解方式CURD练习

需求描述

- 使用Spring的新注解，代替CURD练习里，`applicationContext.xml`的所有配置

需求实现

- account

```java
package com.itheima.bean;

import lombok.Data;

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

- dao接口

```java
package com.itheima.dao;

import com.itheima.bean.Account;

import java.util.List;

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

- dao实现

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

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 org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

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

@Repository
public class AccountDaoImpl implements AccountDao {

    @Autowired
    private QueryRunner runner;

    public int add(Account account) throws SQLException {
        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));
    }
}
```

- service接口

```
package com.itheima.service;

import com.itheima.bean.Account;

import java.util.List;

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

- service实现

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

import com.itheima.bean.Account;
import com.itheima.dao.AccountDao;
import com.itheima.service.AccountService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

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

@Service
```

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao dao;

    public int add(Account account) throws SQLException {
        return dao.add(account);
    }

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

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

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

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

- 提供jdbc配置文件：db.properties

```
driverClass=com.mysql.jdbc.Driver
jdbcUrl=jdbc:mysql:///day41_spring
user=root
password=root
```

- 提供核心配置类：AppConfig

```
package com.itheima.config;

import com.mchange.v2.c3p0.ComboPooledDataSource;
import org.apache.commons.dbutils.QueryRunner;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;

import javax.sql.DataSource;
import java.beans.PropertyVetoException;

@Configuration //1. 这是一个核心配置类
@ComponentScan("com.itheima") //2. 表示扫描扫描包
@PropertySource("classpath:db.properties") //3. 引入properties
public class AppConfig {


    @Value("${driverClass}")
```

```
    private String driverClass;

    @Value("${jdbcUrl}")
    private String jdbcUrl;

    @Value("${user}")
    private String user;

    @Value("${password}")
    private String password;


    //4. 让spring托管QueryRunner
    @Bean
    public QueryRunner createRunner(DataSource ds){
        return new QueryRunner(ds);
    }

    //5. 让spring托管DataSource
    @Bean
    public DataSource createDataSource() throws PropertyVetoException {
        ComboPooledDataSource ds = new ComboPooledDataSource();
        ds.setDriverClass(driverClass);
        ds.setJdbcUrl(jdbcUrl);
        ds.setUser(user);
        ds.setPassword(password);
        return ds;
    }

}
```

功能测试

```
package com.itheima.test;

import com.itheima.bean.Account;
import com.itheima.config.AppConfig;
import com.itheima.service.AccountService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.sql.SQLException;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = AppConfig.class)
public class TestAccountServiceImpl {

    @Autowired
    private AccountService as;

    @Test
    public void testAdd() throws SQLException {

        Account a = new Account();
```

```
        a.setName("德玛");
        a.setMoney(2);

        as.add(a);
    }

    @Test
    public void testDelete() throws SQLException {
        as.delete(6);
    }


    @Test
    public void testUpdate() throws SQLException {

        //先查
        Account a = as.findById(3);
        a.setMoney(70);

        //再改
        as.update(a);
    }

    @Test
    public void testFindAll() throws SQLException {
        System.out.println(as.findAll());
    }
}
```

五、注解深入【拓展了解】

准备环境

1. 创建Module，引入依赖

```
    <dependencies>
        <!-- Spring -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.2.RELEASE</version>
        </dependency>
        <!-- Spring整合Junit -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>5.1.2.RELEASE</version>
        </dependency>
        <!-- Junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
        <!-- snakeyaml，用于解析yaml文件的工具包 -->
        <dependency>
```

```
            <groupId>org.yaml</groupId>
            <artifactId>snakeyaml</artifactId>
            <version>1.25</version>
        </dependency>

        <!-- mysql驱动 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.47</version>
        </dependency>
        <!-- c3p0连接池 -->
        <dependency>
            <groupId>c3p0</groupId>
            <artifactId>c3p0</artifactId>
            <version>0.9.1.2</version>
        </dependency>
    </dependencies>
```

2. 创建核心配置类

- 在`com.itheima`包里创建核心配置类`AppConfig`

```
@Configuration
@ComponentScan("com.itheima")
public class AppConfig {

}
```

3.创建Service

- 接口

```
package com.itheima.service;

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

- 实现

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

import com.itheima.service.UserService;
import org.springframework.stereotype.Service;

@Service
public class UserServiceImpl implements UserService {
    public void add() {
        System.out.println("执行了UserServiceImpl的add方法~");
    }
}
```

4.创建单元测试类

```
package com.itheima.test;

import com.itheima.config.AppConfig;
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.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = AppConfig.class)
public class TestUserServiceImpl {

   /* @Autowired
    private UserService us;
*/
    @Autowired
    private ApplicationContext context;

    @Test
    public void testAdd(){

      /*  us.add();*/

        System.out.println("--------------工厂里面保存：-----------------");
        String[] names = context.getBeanDefinitionNames();
        for (String name : names) {
            System.out.println("name=" + name);
        }
    }
}
```

@ComponentScan

> 这个注解的作用就是用来扫描指定包下的所有类。如果哪个类身上打了注解（@Controller | @Service | @Repository | @Component），就被spring给管理起来。默认情况下Spring管理这些对象的时候，**他们的id名字就是类的名字,但是第一个字母小写。**我们是否可用修改这种命名策略呢?

BeanName生成策略

说明

- 默认的BeanName生成策略：
  - 如果注册bean时指定了id/name，以配置的id/name作为bean的名称
  - 如果没有指定id/name，则以类名首字母小写作为bean的名称
- 在模块化开发中，多个模块共同组成一个工程。
  - 可能多个模块中，有同名称的类，按照默认的BeanName生成策略，会导致名称冲突。
  - 这个时候可以自定义beanname生成策略解决问题
- `@ComponentScan`的`nameGenerator`属性，可以配置自定义的BeanName生成策略，步骤：

1. 创建Java类，实现`BeanNameGenerator`接口，定义BeanName生成策略
2. 在注解`@ComponentScan`中，使用`nameGenerator属性`指定生成策略即可

示例

1. 创建Java类，实现`BeanNameGenerator`接口，定义BeanName生成策略

```
package com.itheima.demo1_componentscan;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionRegistry;
import org.springframework.beans.factory.support.BeanNameGenerator;

/*
    这是我们自己的id 的命名策略 | 策略生成器
 */
public class MyBeanNameGenerator implements BeanNameGenerator {

    /**
     * 主要是用来生成对象的id名字
     * @param beanDefinition 要生成id名字的bean对象
     * @param beanDefinitionRegistry  注册中心。
     * @return
     */
    public String generateBeanName(BeanDefinition beanDefinition,
BeanDefinitionRegistry beanDefinitionRegistry) {

        // 得到我们的管理的类的全路径名。
        String beanClassName = beanDefinition.getBeanClassName();
        System.out.println("beanClassName=" + beanClassName);

        // 把类的全路径名当成它的id名
        return beanClassName;
    }
}
```

2. 在注解`@ComponentScan`中，使用`nameGenerator`指定生成策略

```
@Configuration // 这是核心配置类
@ComponentScan(
        value = "com.itheima" ,  //扫描具体的包
        nameGenerator = MyBeanNameGenerator.class,  //指定我们自己的id命名策略
)
public class AppConfig {
}
```

3. 执行上面的单元测试

扫描规则过滤器

说明

- `@ComponentScan`默认的扫描规则：
    - 扫描指定包里的`@Component`及衍生注解（`@Controller`,`@Service`,`@Repository`）配置的bean
- `@ComponentScan`注解也可以自定义扫描规则，来包含或排除指定的bean。步骤：

1. 创建Java类，实现`TypeFilter`接口，重写`match`方法
    - 方法返回boolean。true表示匹配过滤规则；false表示不匹配过滤规则
2. 使用`@ComponentScan`注解的属性，配置过滤规则：
    - `includeFilter`：用于包含指定TypeFilter过滤的类，符合过滤规则的类将被扫描
    - `excludeFilter`：用于排除指定TypeFilter过滤的类，符合过滤规则的类将被排除

示例1-根据注解过滤

> 哪个类身上有指定的注解，那么就忽略它。这是按照注解的名字来忽略的。

```
@Configuration // 这是核心配置类
@ComponentScan(
        value = "com.itheima" ,  //扫描具体的包
        nameGenerator = MyBeanNameGenerator.class,  //指定我们自己的id命名策略
        excludeFilters =@ComponentScan.Filter(type = FilterType.ANNOTATION,
classes = Service.class) //按照注解来排除类

)
public class AppConfig {
}
```

示例2-根据指定类过滤

```
@Configuration // 这是核心配置类
@ComponentScan(
        value = "com.itheima" ,  //扫描具体的包
        nameGenerator = MyBeanNameGenerator.class,  //指定我们自己的id命名策略
        excludeFilters = @ComponentScan.Filter(type= FilterType.ASSIGNABLE_TYPE
, classes = UserServiceImpl.class) //按照指定的类来排除
)
public class AppConfig {

}
```

示例3-自定义过滤

1. 编写过滤器，实现`TypeFilter`接口，重写`match`方法

```
package com.itheima.demo1_componentscan;

import org.springframework.core.type.ClassMetadata;
import org.springframework.core.type.classreading.MetadataReader;
import org.springframework.core.type.classreading.MetadataReaderFactory;
import org.springframework.core.type.filter.TypeFilter;

import java.io.IOException;

/*
    自定义的忽略|包含的过滤器
 */
public class MyTypeFilter  implements TypeFilter {

    /**
     * 用于控制到底什么规则即满足忽略 或者 包含。
     * @param metadataReader  里面包含了我们正要验证的类的信息
```

```
     * @param metadataReaderFactory 工厂
     * @return
     * @throws IOException
     */
    public boolean match(MetadataReader metadataReader, MetadataReaderFactory
metadataReaderFactory) throws IOException {

        //需求： 哪个类身上包含了User这样的字符串，我们就排除它。

        // 得到现在正在扫描的这个类的元数据对象
        ClassMetadata classMetadata = metadataReader.getClassMetadata();

        //得到这个类的全路径名
        String className = classMetadata.getClassName();


        //判断这个类的名字是否包含User，如果包含，返回true， 否则返回false.
        //如果我们在核心配置类里面使用的是排除的规则，那么true即表示要排除这个类，false表示不
排除这个类
        //如果我们在核心配置类里面使用的是包含的规则，那么true即表示要扫描这个类，false表示不
扫描这个类。
        return className.contains("User");
    }
}
```

2. 使用注解`@ComponentScan`，配置过滤规则

```
@Configuration // 这是核心配置类
@ComponentScan(
        value = "com.itheima" ,  //扫描具体的包
        nameGenerator = MyBeanNameGenerator.class,  //指定我们自己的id命名策略
        excludeFilters = @ComponentScan.Filter(type = FilterType.CUSTOM ,
classes = MyTypeFilter.class) // 按照自定义的规则来排除类
)
public class AppConfig {

}
```

3. 执行单元测试，看看打印的结果。

@PropertySource

`yml`配置文件介绍 耶某

- 大家以前学习过的常用配置文件有`xml`和`properties`两种格式，但是这两种都有一些不足：
  - `properties`：
    - 优点：键值对的格式，简单易读
    - 缺点：不方便表示复杂的层级
  - `xml`：
    - 优点：层次结构清晰
    - 缺点：配置和解析语法复杂
- springboot采用了一种新的配置文件：`yaml`（或`yml`），它综合了`xml`和`properties`的优点。

    - `yaml are't markup language` => yaml
    - 使用空格表示层次关系：相同空格的配置项属于同一级
    - 配置格式是`key:空格value`，键值对之间用`:空格`表示
- `yaml`文件示例：

```
jdbc:
    driver: com.mysql.jdbc.Driver # 注意：英文冒号后边必须有一个空格
    url: jdbc:mysql:///spring
    username: root
    password: root

jedis:
    host: localhost
    port: 6379
```

使用`@PropertySource`加载`yml`

说明

- `@PropertySource`可以使用`factory`属性，配置`PropertySourceFactory`，用于自定义配置文件的解析
- 步骤：
    1. 创建`yaml`文件：`application.yml`
    2. 导入依赖`snakeyaml`，它提供了解析yml文件的功能
    3. 创建Java类，实现`PropertySourceFactory`接口，重写`createPropertySource`方法
    4. 使用`@PropertySource`注解，配置工厂类

示例

1. 在`resources`目录里创建`yaml`文件：`application.yml`

```
jdbc:
    driver: com.mysql.jdbc.Driver # 注意：英文冒号后边必须有一个空格
    url: jdbc:mysql:///spring
    username: root
    password: root

jedis:
    host: localhost
    port: 6379
```

2. 在`pom.xml`增加导入依赖`snakeyaml`，它提供了解析yml文件的功能

```
<dependency>
    <groupId>org.yaml</groupId>
    <artifactId>snakeyaml</artifactId>
    <version>1.25</version>
</dependency>
```

3. 创建Java类，实现`PropertySourceFactory`接口，重写`createPropertySource`方法

```
public class YamlSourceFactory implements PropertySourceFactory {
    /**
     * 解析yaml配置文件
```

```
     * @param name 名称
     * @param resource 配置文件EncodedResource对象
     * @return PropertySource
     */
    @Override
    public PropertySource<?> createPropertySource(String name, EncodedResource
resource) throws IOException {
        //1. 创建yaml解析的工厂
        YamlPropertiesFactoryBean factoryBean = new YamlPropertiesFactoryBean();
        //2. 设置要解析的资源内容
        factoryBean.setResources(resource.getResource());
        //3. 把资源文件解析成Properties对象
        Properties properties = factoryBean.getObject();
        //4. 把properties封装成PropertySource对象并返回
        return new PropertiesPropertySource("application", properties);
    }
}
```

4. 使用`@PropertySource`注解，配置工厂类

```
@Configuration("appConfig")
@PropertySource(
    value = "classpath:application.yml",
    factory = YamlSourceFactory.class
)
public class AppConfig {

    @Value("${jdbc.driver}")
    private String driverClass;

    @Value("${jdbc.url}")
    private String jdbcUrl;

    @Value("${jdbc.username}")
    private String user;

    @Value("${jdbc.password}")
    private String password;

    public void show(){
        System.out.println(driverClass);
        System.out.println(jdbcUrl);
        System.out.println(user);
        System.out.println(password);
    }
}
```

5. 测试

```
package com.itheima.test;

import com.itheima.config.AppConfig;
import com.itheima.demo4_conditional.Person;
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.test.context.ActiveProfiles;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import javax.sql.DataSource;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = AppConfig.class)
public class TestUserServiceImpl {

    //注入AppConfig
    @Autowired
    private AppConfig config;



    @Test
    public void testShow(){
        config.show();
    }

}
```

@Import

注册bean的方式

如果要注解方式配置一个bean，可以如下方式：

- 在类上使用`@Component`,`@Controller`,`@Service`,`@Repository`：只能用于自己编写的类上，jar包里的类不能使用（比如ComboPooledDataSource）
- 在方法上使用`@Bean`：把方法返回值配置注册成bean到IoC容器，通常用于注册第三方jar里的bean
- 在核心配置类上使用`@Import`：
  - `@Import(类名.class)`，注册的bean的id是全限定类名
  - `@Import(自定义ImportSelector.class)`：把自定义ImportSelector返回的类名数组，全部注册bean
  - `@Import(自定义ImportBeanDefinitionRegister.class)`：在自定义ImportBeanDefinitionRegister里手动注册bean

ImportSelector导入器

示例1-直接导入注册bean

> 使用@import来到一个类

```
@Configuration
@ComponentScan("com.itheima")
@Import(Teacher.class)
public class AppConfig {
}
```

示例2-使用ImportSelector注册bean

- 导入器

```
package com.itheima.demo3_import;

import org.springframework.context.annotation.ImportSelector;
import org.springframework.core.type.AnnotationMetadata;

/*
    导入选择器,
 */
public class MyImportSelector  implements ImportSelector {

    /**
     * 导入具体的类
     * @param importingClassMetadata
     * @return 返回一个数组，数组里面包含谁，就表示导入谁
     */
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        //数组里面包含三个类的全路径。表示要导入这个三个类
        return new String[]{Student.class.getName() , Teacher.class.getName() ,
Worker.class.getName()};
    }
}
```

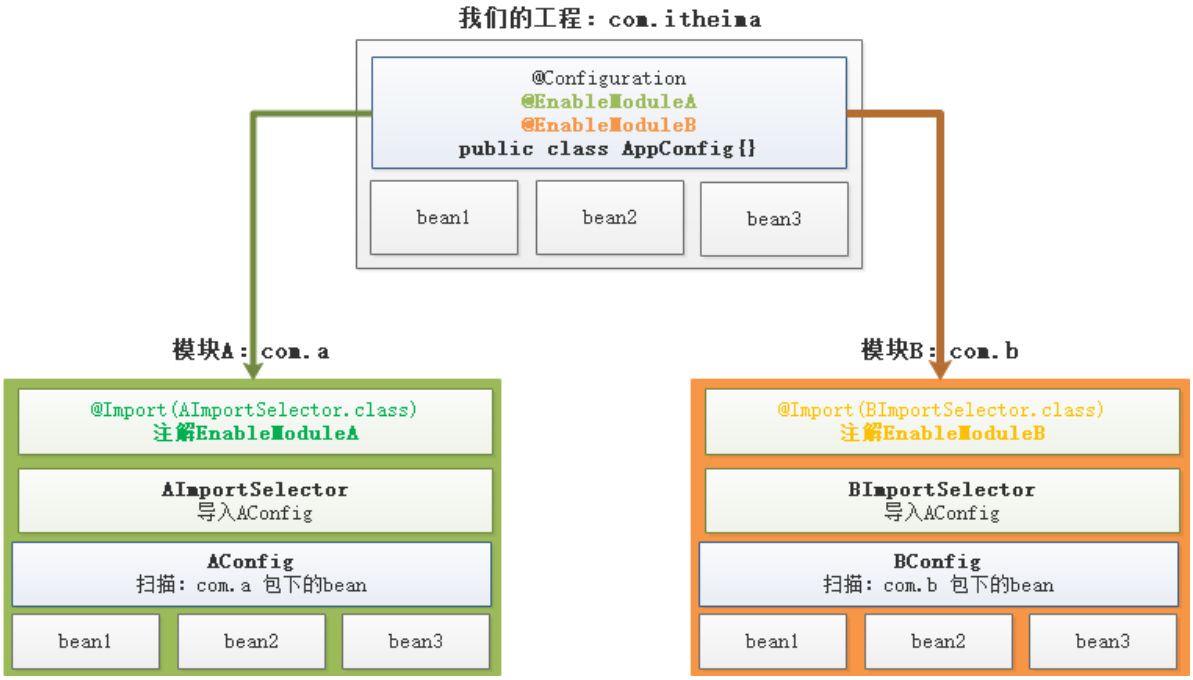
- 示例

```
@Configuration
@ComponentScan("com.itheima")
@Import(MyImportSelector.class)
public class AppConfig {
    @Value("${jdbc.driver}")
    private String jdbcDriver;

    @Value("${jedis.host}")
    private String jedisHost;

    public void print(){
        System.out.println(host);
        System.out.println(port);
    }
}
```

示例3-ImportSelector的高级使用

说明

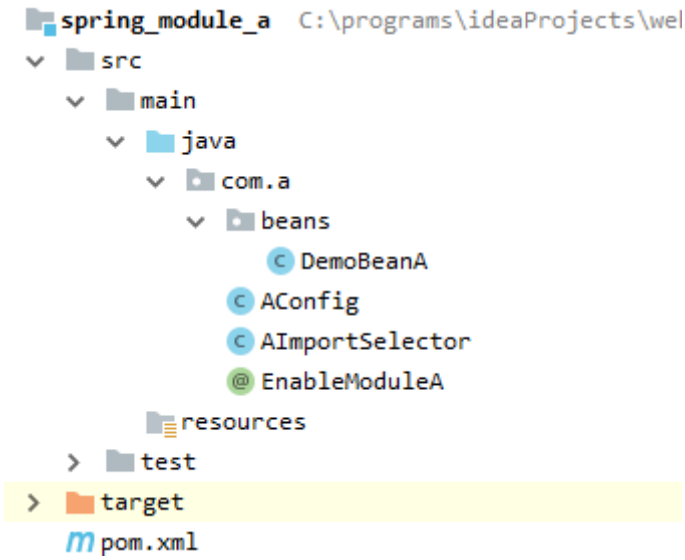
- springboot框架里有大量的`@EnableXXX`注解，底层就是使用了ImportSelector解决了模块化开发中，如何启动某一模块功能的问题
- 例如：
    - 我们开发了一个工程，要引入其它的模块，并启动这个模块的功能：把这个模块的bean进行扫描装载



- 步骤：
    1. 创建一个Module：module_a
        1. 在包`com.a`里创建几个Bean
        2. 创建核心配置文件AConfig，扫描`com.a`
        3. 定义一个ImportSelector：`AImportSelector`，导入`AConfig`类
        4. 定义一个注解`@EnableModuleA`
    2. 在我们的Module的核心配置文件`AppConfig`上增加注解：`@EnableModuleA`
        1. 引入module_a的坐标
        2. 测试能否获取到Module a里的bean

第一步：创建新Module：`module_a`

```
<!-- module_a的坐标 -->
<groupId>com.itheima</groupId>
    <artifactId>spring_module_a</artifactId>
    <version>1.0-SNAPSHOT</version>
```



1. 在包`com.a.beans`里创建类`DemoBeanA`

```
@Component
public class DemoBeanA {
}
```

2. 创建核心配置类`AConfig`

```
@Configuration
@ComponentScan("com.a")
public class AConfig {
}
```

3. 创建导入器：创建Java类，实现`ImportSelector`接口，重写`selectImports`方法

```
public class AImportSelector implements ImportSelector {
    /**
     * @param importingClassMetadata。被@Import注解标注的类上所有注解的信息
     */
    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        return new String[]{AConfig.class.getName()};
    }
}
```

4. 定义注解`@EnableModuleA`

```
@Import(AImportSelector.class)
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface EnableModuleA {
}
```

第二步：引入module_a，启动模块a

1. 在我们自己的工程pom.xml里增加依赖

```
<dependency>
    <groupId>com.itheima</groupId>
    <artifactId>spring_module_a</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

2. 在我们自己的工程核心配置类上，使用注解@EnableModuleA启动模块a

```
@Configuration
@EnableModuleA
public class AppConfig {

}
```

3. 在我们自己的工程里测试

```
    @Test
    public void testBean(){
        //在我们自己的工程里，可以获取到module_a里的bean
        DemoBeanA demoBeanA = app.getBean(DemoBeanA.class);
        System.out.println(demoBeanA);
    }
```

ImportBeanDefinitionRegister注册器

说明

- `ImportBeanDefinitionRegister`提供了更灵活的注册bean的方式
- AOP里的`@EnableAspectJAutoProxy`就使用了这种注册器，用于注册不同类型的代理对象
- 步骤：
    1. 创建注册器：

       创建Java类，实现`ImportBeanDefinitionRegister`接口，重写`registerBeanDefinitions`方法
    2. 在核心配置类上，使用`@Import`配置注册器

示例

1. 创建类`com.other.Other`

```
public class Other {
    public void show(){
        System.out.println("Other.show....");
    }
}
```

2. 创建注册器

```
public class CustomImportBeanDefinitionRegister implements
ImportBeanDefinitionRegistrar {
    /**
     * @param importingClassMetadata 当前类的注解信息
     * @param registry 用于注册bean的注册器
     */
    @Override
    public void registerBeanDefinitions(AnnotationMetadata
importingClassMetadata, BeanDefinitionRegistry registry) {
        //获取bean定义信息
        BeanDefinition beanDefinition =
BeanDefinitionBuilder.rootBeanDefinition("com.other.Other").getBeanDefinition();
        //注册bean，方法参数：bean的id，bean的定义信息
        registry.registerBeanDefinition("other", beanDefinition);
    }
}
```

2. 在核心配置类上，使用`@Import`配置注册器

```
@Configuration
@Import({CustomImportBeanDefinitionRegister.class})
public class AppConfig {

}
```

3. 测试

```
@Test
public void testImportRegister(){
    //获取扫描范围外，使用ImportBeanDefinitionRegister注册的bean
    Other other = app.getBean("other",Other.class);
    other.showOther();
}
```

@Conditional （条件）

说明

- `@Conditional` 加在bean上，用于选择性的注册bean：
  - 符合Condition条件的，Spring会生成bean对象 存储容器中
  - 不符合Condition条件的，不会生成bean对象
- 示例：
  - 有一个类Person(姓名name，年龄age)
  - 如果当前操作系统是Linux：就创建Person(linus, 62)对象，并注册bean
  - 如果当前操作系统是Windows：就创建Person(BillGates, 67)对象，并注册bean
- 步骤
  1. 创建Person类
  2. 创建两个Java类，都实现 `Condition` 接口：
     - WindowsCondition：如果当前操作系统是Windows，就返回true
     - LinuxCondition：如果当前操作系统是Linux，就返回true
  3. 在核心配置类里创建两个bean
     - 一个bean名称为bill，加上 `@Conditional(WindowsCondition.class)`
     - 一个bean名称为linus，加上 `@Conditional(LinuxCondition.class)`

示例

1. 创建Person类

```
package com.itheima.demo4_conditional;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Teacher {
    private String name;
    private int age;
}
```

2. 创建两个Java类，实现 `Condition` 接口

```
package com.itheima.demo4_conditional;

import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;

public class WindowsCondition implements Condition {

    /**
     * 用于判定当前的操作系统是否是windows
     * @param context
     * @param metadata
     * @return
     */
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata
metadata) {

        //得到系统的名字
        String name = context.getEnvironment().getProperty("os.name");

        return name.contains("Windows");
    }
}
```

```
package com.itheima.demo4_conditional;

import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;

public class LinuxCondition implements Condition {

    /**
     * 用于判定当前的操作系统是否是Linux
     * @param context
     * @param metadata
     * @return
     */
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata
metadata) {

        //得到系统的名字
        String name = context.getEnvironment().getProperty("os.name");

        return name.contains("Linux");
    }
}
```

3. 核心配置类

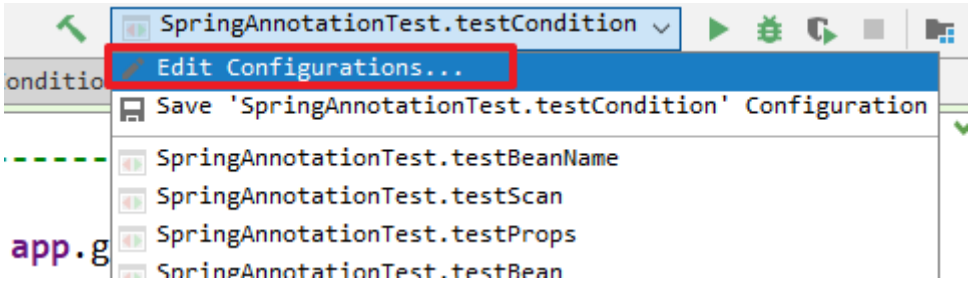
```
@Configuration
public class AppConfig {

    //生成两个Person对象
    @Bean
    @Conditional(WindowsCondition.class) //如果当前的操作系统是windows ，就创建对象，管理起来
    public Person billGates(){
        return  new Person("比尔", 67);
    }

    @Bean
    @Conditional(LinuxCondition.class) //如果当前的操作系统是linux ，就创建对象，管理起来
    public Person linus(){
        return  new Person("林纳斯", 62);
    }


}
```

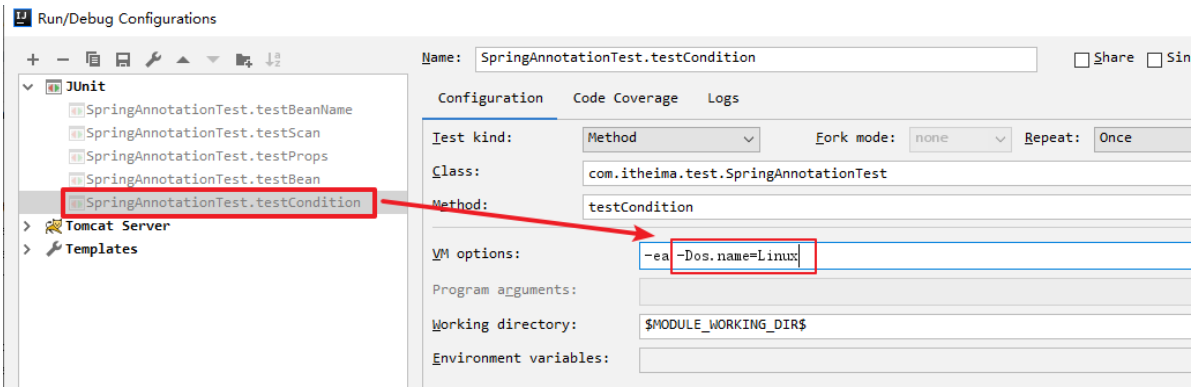
5. 测试

```
package com.itheima.test;

import com.itheima.config.AppConfig;
import com.itheima.demo4_conditional.Teacher;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import javax.sql.DataSource;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = AppConfig.class)
public class TestDemo {

    //注入工厂
    @Autowired
    private ApplicationContext context;


    @Test
    public void test01(){
        //得到工厂里面管理的bean的名字
        String[] definitionNames = context.getBeanDefinitionNames();
        for (String name : definitionNames) {
            System.out.println("name=" + name);
        }
    }


}
```

执行一次单元测试方法之后，按照以下方式，可以通过JVM参数的方式，设置os.name的参数值。

设置之后，再次执行单元测试方法





Conditional的扩展注解

`@Conditional`在springboot里应用非常多，以下列出了一些`@Conditional`的扩展注解：

- @ConditionalOnBean：当容器中有指定Bean的条件下进行实例化。
- @ConditionalOnMissingBean：当容器里没有指定Bean的条件下进行实例化。
- @ConditionalOnClass：当classpath类路径下有指定类的条件下进行实例化。
- @ConditionalOnMissingClass：当类路径下没有指定类的条件下进行实例化。
- @ConditionalOnWebApplication：当项目是一个Web项目时进行实例化。
- @ConditionalOnNotWebApplication：当项目不是一个Web项目时进行实例化。
- @ConditionalOnProperty：当指定的属性有指定的值时进行实例化。
- @ConditionalOnExpression：基于SpEL表达式的条件判断。
- @ConditionalOnJava：当JVM版本为指定的版本范围时触发实例化。
- @ConditionalOnResource：当类路径下有指定的资源时触发实例化。
- @ConditionalOnJndi：在JNDI存在的条件下触发实例化。
- @ConditionalOnSingleCandidate：当指定的Bean在容器中只有一个，或者有多个但是指定了首选的Bean时触发实例化。

@Profile

说明

- 在开发中，我们编写的工程通常要部署不同的环境，比如：开发环境、测试环境、生产环境。不同环境的配置信息是不同的，比如：数据库配置信息；如果每次切换环境，都重新修改配置的话，会非常麻烦，且容易出错
- 针对这种情况，Spring提供了`@Profile`注解：可以根据不同环境配置不同的bean，激活不同的配置
    - `@Profile`注解的底层就是`@Conditional`

- 例如：
  - 定义三个数据源：
    - 开发环境一个`DataSource`，使用`@Profile`配置环境名称为`dev`
    - 测试环境一个`DataSource`，使用`@Profile`配置环境名称为`test`
    - 生产环境一个`DataSource`，使用`@Profile`配置环境名称`pro`
  - 在测试类上，使用`@ActiveProfiles`激活哪个环境，哪个环境的数据源会生效
    - 实际开发中有多种方式可以进行激活，这里演示一个单元测试类里是怎样激活的

示例

1. 在`pom.xml`中增加导入依赖`mysql驱动`，`c3p0`

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>
    <dependency>
        <groupId>c3p0</groupId>
        <artifactId>c3p0</artifactId>
        <version>0.9.1.2</version>
    </dependency>


    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.5.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.2.5.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>
```

2. 在配置类里创建三个数据源，并配置`@Profile`

```
@Configuration
public class AppConfig {

     //生成三个数据源
    @Bean
    @Profile("dev")
    public DataSource devDataSource() throws PropertyVetoException {
        System.out.println("dev  开发环境的");
        ComboPooledDataSource dataSource = new ComboPooledDataSource();
        dataSource.setDriverClass("com.mysql.jdbc.Driver");
        dataSource.setJdbcUrl("jdbc:mysql:///devdb");
```

```
        dataSource.setUser("root");
        dataSource.setPassword("root");
        dataSource.setMaxPoolSize(20);
        return dataSource;
    }

    @Bean
    @Profile("test")
    public DataSource testDataSource() throws PropertyVetoException {
        System.out.println("test  测试环境的");
        ComboPooledDataSource dataSource = new ComboPooledDataSource();
        dataSource.setDriverClass("com.mysql.jdbc.Driver");
        dataSource.setJdbcUrl("jdbc:mysql:///testdb");
        dataSource.setUser("root");
        dataSource.setPassword("root");
        dataSource.setMaxPoolSize(20);
        return dataSource;
    }


    @Bean
    @Profile("release")
    public DataSource releaseDataSource() throws PropertyVetoException {
        System.out.println("release  生产环境的");
        ComboPooledDataSource dataSource = new ComboPooledDataSource();
        dataSource.setDriverClass("com.mysql.jdbc.Driver");
        dataSource.setJdbcUrl("jdbc:mysql:///releasedb");
        dataSource.setUser("root");
        dataSource.setPassword("root");
        dataSource.setMaxPoolSize(20);
        return dataSource;
    }
}
```

3. 在测试类上，使用`@ActiveProfiles`激活哪个环境，哪个环境的数据源会生效

> 或者使用JVM参数`-Dspring.profiles.active=dev`

```
@ActiveProfiles("dev") //激活dev环境
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = AppConfig.class)
public class SpringTest {

    @Autowired
    private ApplicationContext app;

    @Test
    public void testProfile(){
        DataSource dataSource = app.getBean(DataSource.class);
        System.out.println(dataSource);
    }
}
```

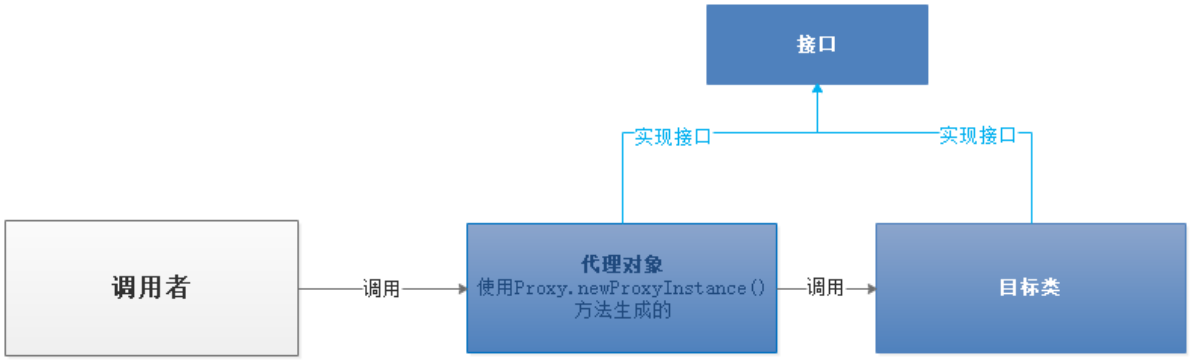
六、动态代理复习

- 客户.租房() ---> 房东.租房()

- 客户.租房() ---> 中介.租房() ---> 房东.租房()
- 客户.唱歌() ---> 明星.唱歌()
- 客户.唱歌() ---> 经纪人.唱歌() ---> 明星.唱歌()
- 代理：当你不能直接调用目标对象，或者不方便直接调用目标对象，可以通过代理间接调用
  - 代理可以对目标对象进行增强
  - 代理可以对目标对象进行控制
- 静态代理：装饰者模式
- 对原有的功能进行扩展升级 | 增强有哪些手段?

> 代理模式， 装饰者模式

1. JDK的基于接口的动态代理



API介绍

`Proxy`类

- 使用JDK的动态代理的要求：目标对象必须实现了接口。
- 相关类：JDK的类`java.lang.reflect.Proxy`，提供了生成代理对象的方法
- 生成代理对象的方法：
  `Proxy.newProxyInstance(ClassLoader loader,Class[] interfaces, InvocationHandler h)`
  - `loader`：类加载器
  - `interfaces`：目标对象所实现的接口 字节码数组
  - `h`：用于写代理对象要做的事情，通常写成`InvocationHandler`接口的匿名内部类，实现其`invoke`方法

`InvocationHandler`接口

- 接口只有一个方法：每次当代理对象被调用时，这个方法都会执行。在方法里通常写代理对象的行为
  `invoke(Object proxy, Method method, Object[] args)`
- 方法的参数：
  - `Object proxy`：最终生成的代理对象
  - `Method method`：用户在调用代理对象时，所执行的方法对象
  - `Object[] args`：用户在调用代理对象，执行方法时，所传递的实参
- 方法的返回值：
  - 当用户调用的代理对象的方法后，得到的返回值

使用示例

有目标类（待增强的类）

- 接口

```
package com.itheima.proxy.jdk;

public interface Start {

    void sing(String name);
    void dance(String name);
}
```

- 实现类：

```
package com.itheima.proxy.jdk;

public class SuperStar implements Star {
    public void sing(String name) {
        System.out.println("明星在唱歌：" + name);
    }

    public void dance(String name) {
        System.out.println("明星在跳舞：" + name);
    }
}
```

有通知类（用于进行功能增强的）

> 主要是用来做增强的。

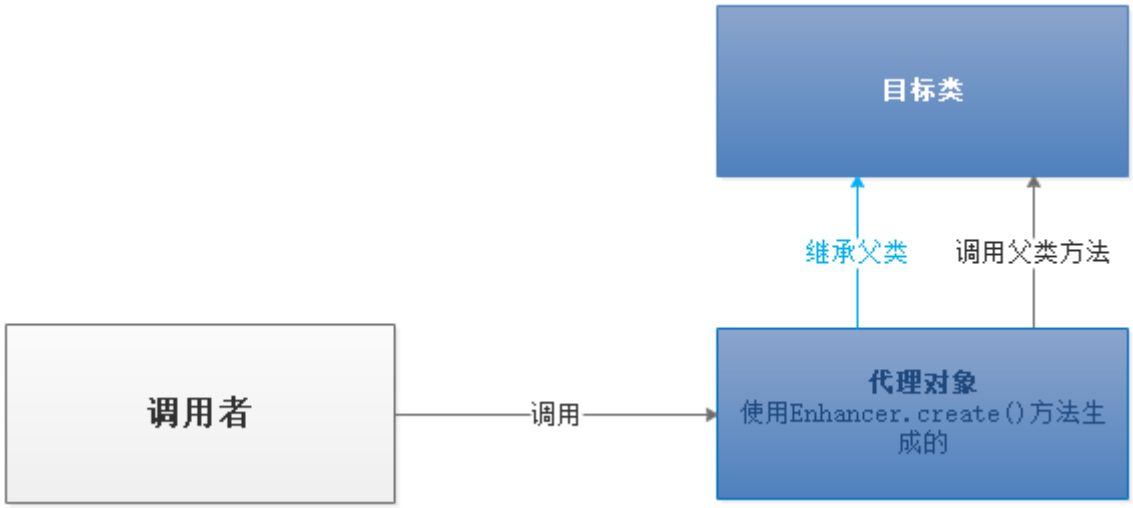
```
package com.itheima.proxy.jdk;

public class StarAdvice {

    public void before(){
        System.out.println("彩排一下~~");
    }
    public void after(){
        System.out.println("收钱~~");
    }
}
```

使用动态代理生成代理对象

```
package com.itheima.test;

import com.itheima.proxy.jdk.Star;
import com.itheima.proxy.jdk.StarAdvice;
import com.itheima.proxy.jdk.SuperStar;
import org.junit.Test;
```

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class TestJDKProxy {

    @Test
    public void testDemo(){

        //1. 真实对象
        final Star star = new SuperStar();

        //2. 创建代理
        Star proxyObj = (Star) Proxy.newProxyInstance(
                star.getClass().getClassLoader(),  // 使用什么类加载器
                star.getClass().getInterfaces(), //实现什么接口
                new InvocationHandler() { //调用处理器

                    //外部的代理对象调用什么方法， 这里的这个invoke方法都会被执行。
                    // proxy :代理对象， method :方法对象 ,args : 方法的参数
                    public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                        System.out.println("调用了invoke...");
                        //让真实对象唱歌
                        //star.sing("忘情水");

                        //增强：  彩排
                        StarAdvice sa = new StarAdvice();
                        sa.before();

                        //使用反射调用
                        Object o = method.invoke(star , args);

                        //增强
                        sa.after();
                        return o;
                    }
                }
        );

        //3. 调用代理的唱歌和跳舞方法
        proxyObj.sing("忘情水");
        //proxyObj.dance("脱衣舞");

    }
}
```

2. cglib的基于子类的动态代理【了解】



API介绍

`Enhancer`类

- 使用cglib的要求：
  - 目标对象不需要有接口
  - 目标类不能是final类
  - 要增强的方法不能是final方法
- 相关类介绍：
  - jar包：Spring框架已经把cglib包含进去了，所以只要导入`spring-context`即可
  - 核心类：`org.springframework.cglib.proxy.Enhancer`，提供了生成代理对象的方法
- 生成代理对象的方法：

  `Enhancer.create(Class superClass, Callback callback)`
  - `superClass`：目标对象的字节码
  - `callback`：回调函数，用于写代理对象要做的事情，通常写成`MethodInterceptor`的匿名内部类对象

    > callback相当于jdk动态代理中的`InvocationHandler`

`MethodInterceptor`接口

- 接口只有一个方法：每次当代理对象被调用时，这个方法都会执行。在方法里通常写代理对象的行为

  `intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy)`
- 方法的参数：
  - `Object proxy`：最终生成的代理对象
  - `Method method`：用户在调用代理对象时，所执行的方法对象
  - `Object[] args`：用户在调用代理对象，执行方法时，所传递的实参
  - `MethodProxy methodProxy`：用户在调用代理对象时，所执行的方法的代理对象
    - `methodProxy.invokeSuper(proxy, args)`：调用目标对象的方法，性能更强
- 方法的返回值：
  - 当用户调用的代理对象的方法后，得到的返回值

使用示例

目标类（待增强的）

```
package com.itheima.proxy.cglib;

import com.itheima.proxy.jdk.Star;

public class SuperStar {

    public void sing(String name) {
        System.out.println("明星在唱歌：" + name);
    }

    public void dance(String name) {
        System.out.println("明星在跳舞：" + name);
    }
}
```

通知类（用于增强的）

```
package com.itheima.proxy.cglib;

public class StarAdvice {

    public void before(){
        System.out.println("彩排一下~~");
    }
    public void after(){
        System.out.println("收钱~~");
    }
}
```

使用cglib生成代理对象

```
package com.itheima.test;

import com.itheima.proxy.cglib.StarAdvice;
import com.itheima.proxy.cglib.SuperStar;
import org.junit.Test;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

public class TestCglibProxy {



    @Test
    public void testDemo(){
        //1. 创建真实对象
```

```
        final SuperStar star = new SuperStar();

        //2.创建代理
        Enhancer enhancer = new Enhancer();

        SuperStar proxyObj = (SuperStar) enhancer.create(SuperStar.class, new
MethodInterceptor() {

            //o : 代理对象， method : 方法对象 ， objects: 方法参数 ， methodProxy:
方法代理
            public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {

                //增强： 彩排
                StarAdvice sa = new StarAdvice();
                sa.before();


                //调用真实对象的方法
                //Object result = method.invoke(star ,objects );

                //等于就是调用父类的方法。
                Object result =  methodProxy.invokeSuper(o , objects);


                //增强： 收钱
                sa.after();

                return result;
            }
        });

        //3 调用方法
        proxyObj.sing("忘情水");
        proxyObj.dance("机械舞");

    }
}
```

总结

- IOC + DI (注解版本)
- IOC的注解:
  - @Component ： 通用的注解
    - @Controller ----- web层
    - @Service ------ service层
    - @Repository --- -dao层
  - 配置Bean
    - 默认创建的对象还是单例的对象，如果想做成多例： @Scope("prototype")
    - 初始化方法： @PostConstrut
    - 销毁方法： @PreDestroy
- DI的注解

    - @AutoWired： 自动注入，按照类型来查找对象注入，如果找到多个对象，那么会把变量名当成id的名字去找对象，如果还找不到就报错了。
    - @Qulifier： 配合@Autowired来使用，表示根据id名字来注入
    - @Resource ： 根据名字来注入
    - @Value：用来注入普通，简单的数据，一般使用来注入外部properties内容
- 只要使用了注解，就一定要记得在xml文件里面打开扫描的开关
    - <context:component-scan base-package="com.itheima"/>
- 纯注解
    - @Configuration：标记一个类成为核心配置类
    - @ComponentScan： 扫描组件，指定包
    - @PropertySource： 导入外部的properties文件
    - @Import: 导入其他的配置类，让spring管理其他类
    - @Bean： 打在方法上，这个方法的返回值将会被spring管理起来。