Spring

1.1 简介

- 春天 —>给软件行业带来了春天
- 2002年，Rod Jahnson首次推出了Spring框架雏形interface21框架。
- 2004年3月24日，Spring框架以interface21框架为基础，经过重新设计，发布了1.0正式版。
- Rod Johnson的学历，他是悉尼大学的博士，然而他的专业不是计算机，而是音乐学。
- Spring理念：使现有技术更加实用 . 本身就是一个大杂烩，整合现有的框架技术

官网： https://spring.io/projects/spring-framework#overview

官方下载： https://repo.spring.io/release/org/springframework/spring/

GitHub： https://github.com/spring-projects/spring-framework

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>
```

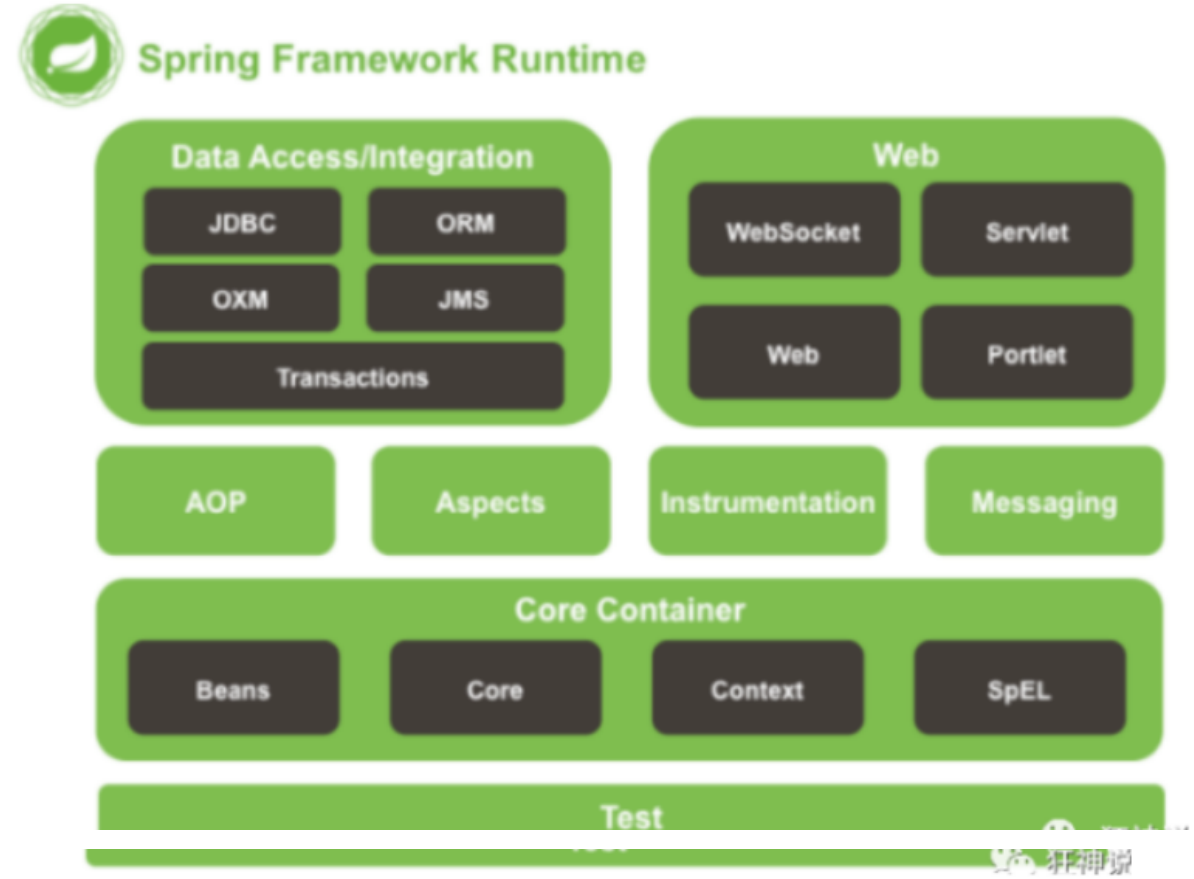
1.2 优点

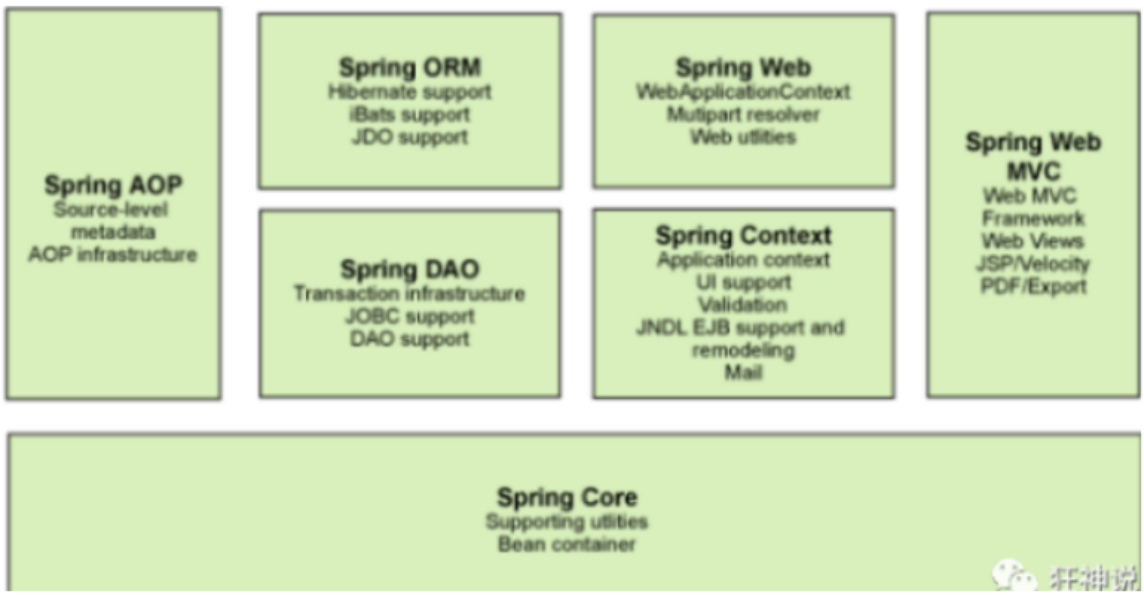
- Spring是一个开源免费的框架 (容器)！
- Spring是一个轻量级的框架，非侵入式的
- **控制反转 IoC，面向切面 Aop**
- 对事务的支持，对框架整合的支持

Spring是一个轻量级的控制反转(IoC)和面向切面(AOP)的容器（框架）。

1.3 组成

Spring 框架是一个分层架构，由 7 个定义良好的模块组成。Spring 模块构建在核心容器之上，核心容器定义了创建、配置和管理 bean 的方式 .





- **核心容器**：核心容器提供 Spring 框架的基本功能。核心容器的主要组件是 BeanFactory，它是工厂模式的实现。BeanFactory 使用*控制反转*（IOC）模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。
- **Spring 上下文**：Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。
- **Spring AOP**：通过配置管理特性，Spring AOP 模块直接将面向切面的编程功能，集成到了 Spring 框架中。所以，可以很容易地使 Spring 框架管理任何支持 AOP的对象。Spring AOP 模块为基于 Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，不用依赖组件，就可以将声明性事务管理集成到应用程序中。
- **Spring DAO**：JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（例如打开和

关闭连接）。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。

- **Spring ORM**：Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具，其中包括 JDO、Hibernate 和 iBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。
- **Spring Web 模块**：Web 上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。所以，Spring 框架支持与 Jakarta Struts 的集成。Web 模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。
- **Spring MVC 框架**：MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成为高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText 和 POI。

1.4 拓展

- Spring Boot
  - 一个快速开发的脚手架
  - 基于SpringBoot可以快速的开发单个微服务
  - 约定大于配置
- Spring Cloud
  - Spring Cloud是基于SpringBoot实现的

2、IOC理论推导

1. UserDao接口

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

2. UserDaoImpl 实现类

```
public class UserDaoImpl implements UserDao {
    public void getUser() {
        System.out.println("查询数据库 获取用户");
    }
}
```

3. UserService 业务接口

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

4. UserServiceImpl 业务实现类

```
public class UserServiceImpl implements UserService {
   private UserDao userDao = new UserDaoImpl();

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

5. 测试

```
@Test
public void test(){
    UserService userService = new UserServiceImpl();
    userService.getUser();
}
```

在我们之前的业务中，用户的需求可能会影响我们原来的代码，我们需要根据用户的需求去修改源代码，如果程序的代码量十分大，修改一次的成本十分昂贵。

我们使用Set接口实现

```
public class UserServiceImpl implements UserService {
    private UserDao userDao ;
    //利用set动态实现值得注入
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }
    public void getUser() {
        userDao.getUser();
    }
}
```

- 之前，程序是主动创建对象，控制权在程序员手上！
- 使用了set注入后，程序不再具有主动性，而是变成了被动的接受对象！

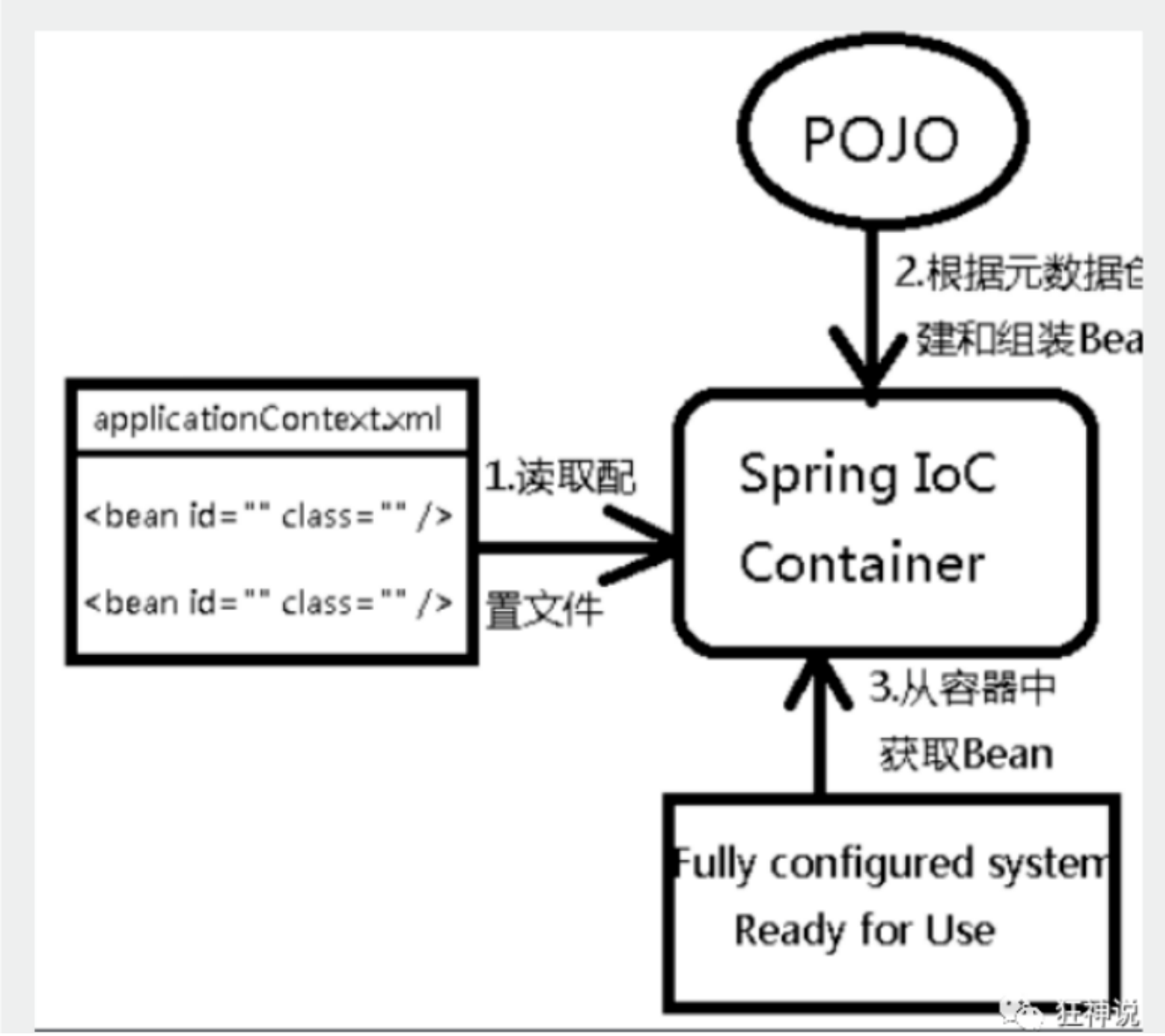
这种思想，从本质上解决了问题，我们程序员不用再去管对象的创建了。系统的耦合性大大降低，可以专注在业务的实现上！这是IOC的原型！

IOC本质

控制反转IoC(Inversion of Control)，是一种设计思想，DI(依赖注入)是实现IoC的一种方法，也有人认为DI只是IoC的另一种说法。没有IoC的程序中，我们使用面向对象编程，对象的创建与对象间的依赖关系完全硬编码在程序中，对象的创建由程序自己控制，控制反转后将对象的创建转移给第三方，个人认为所谓控制反转就是：获得依赖对象的方式反转了。

IoC是Spring框架的核心内容，使用多种方式完美的实现了IoC，可以使用XML配置，也可以使用注解，新版本的Spring也可以零配置实现IoC。

Spring容器在初始化时先读取配置文件，根据配置文件或元数据创建与组织对象存入容器中，程序使用时再从Ioc容器中取出需要的对象。



采用XML方式配置Bean的时候，Bean的定义信息是和实现分离的，而采用注解的方式可以把两者合为一体，Bean的定义信息直接以注解的形式定义在实现类中，从而达到了零配置的目的。

控制反转是一种通过描述（XML或注解）并通过第三方去生产或获取特定对象的方式。在Spring中实现控制反转的是IoC容器，其实现方法是依赖注入（Dependency Injection,DI）。

3、HelloSpring

3.1、导入Jar包

注 : spring 需要导入commons-logging进行日志记录 . 我们利用maven , 他会自动下载对应的依赖项

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.7.RELEASE</version>
</dependency>
```

3.2、编写代码

1、编写一个Hello实体类

```
public class Hello {
    private String userName;
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
    public void show(){
        System.out.println("Hello,"+ userName );
    }
}
```

2、编写我们的spring文件，这里我们命名为beans.xml

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

       <!--
       类型 变量名 = new 类型
       bean = 对象 new Hello()
       id  = 变量名
       class = new 的对象
       property 相当于给对象中的属性设置值
        bean就是java对象 ， 由Spring创建和管理
    -->
    <bean id="hello" class="com.oldwang.pojo.Hello">
        <property name="userName" value="张三"></property>
    </bean>
</beans>
```

3、我们可以去进行测试了 .

```
@Test
public void test(){
    //获取spring的上下文对象 解析bean.xml文件 生成管理相应的Bean对象
   ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
   //getBean : 参数即为spring配置文件中bean的id .
   Hello hello = (Hello) context.getBean("hello");
   hello.show();
}
```

3.3、思考

- Hello 对象是谁创建的？【hello 对象是由Spring创建的
- Hello 对象的属性是怎么设置的？hello 对象的属性是由Spring容器设置的

这个过程就叫控制反转：

- 控制：谁来控制对象的创建，传统应用程序的对象是由程序本身控制创建的，使用Spring后，对象是由Spring来创建的
- 反转：程序本身不创建对象，而变成被动的接收对象.

依赖注入：就是利用set方法来进行注入的.

IOC是一种编程思想，由主动的编程变成被动的接收

可以通过newClassPathXmlApplicationContext去浏览一下底层源码.

3.4、修改案例一

我们在案例一中， 新增一个Spring配置文件beans.xml

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

   <bean id="MysqlImpl" class="com.kuang.dao.impl.UserDaoMySqlImpl"/>
   <bean id="OracleImpl" class="com.kuang.dao.impl.UserDaoOracleImpl"/>

   <bean id="ServiceImpl" class="com.kuang.service.impl.UserServiceImpl">
       <!--注意: 这里的name并不是属性 ， 而是set方法后面的那部分 ， 首字母小写-->
       <!--引用另外一个bean ， 不是用value 而是用 ref-->
       <property name="userDao" ref="OracleImpl"/><!--具体使用哪个接口这里可以直接配置-->
   </bean>

</beans>
```

```
@Test
public void test2(){
   ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
   UserServiceImpl serviceImpl = (UserServiceImpl) context.getBean("ServiceImpl");//这里相当于将原来的Service层也IOC了，不需要再在代码中写出调用哪个接口，只需要在配置文件中指明调用的接口即可。
   serviceImpl.getUser();
    //原来的步骤
    //UserService userService = new UserServiceImpl();
    //userService.setUserDao(new UserDaoMysqlImpl());//原先需要在代码中调用特定的方法
    //userService.getUser();
}
```

OK，到了现在，我们彻底不用再程序中去改动了，要实现不同的操作，只需要在xml配置文件中进行修改，所谓的IoC，一句话搞定：对象由Spring 来创建，管理，装配！

4、IOC创建对象的方式

4.1、通过无参构造方法来创建

1、User.java

```
public class User {

    private String userName;

    public User() {
        System.out.println("User 的无参构造");
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public void show(){
        System.out.println(userName);
    }
}
```

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

    <bean id="user" class="com.oldwang.pojo.User">
        <property name="userName" value="oldwang"></property>
    </bean>
</beans>
```

3、测试类

```java
@Test
public void test(){
   ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
   //在执行getBean的时候，user已经创建好了 ，通过无参构造
   User user = (User) context.getBean("user");
   //调用对象的方法 .
   user.show();
}
```

结果可以发现，在调用show方法之前，User对象已经通过无参构造初始化了！

4.2、通过有参构造方法来创建

1、UserT . java

```java
public class User {

    private String userName;

    public User(String userName) {
        this.userName = userName;
        System.out.println("User 的有参构造");
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public void show(){
        System.out.println(userName);
    }
}
```

2、beans.xml 有三种方式编写

```
<?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">

    <!--第一种通过下标赋值-->
    <bean id="user" class="com.oldwang.pojo.User">
        <constructor-arg index="0" value="oldwang"></constructor-arg>
    </bean>

    <!-- 第二种根据参数名字设置 -->
    <bean id="user1" class="com.oldwang.pojo.User">
        <!-- name指参数名 -->
        <constructor-arg name="userName" value="oldwang"></constructor-arg>
    </bean>
    <!-- 第三种根据参数类型设置(不推荐使用) -->
    <bean id="user2" class="com.oldwang.pojo.User">
        <constructor-arg type="java.lang.String" value="oldwang"></constructor-arg>
    </bean>
</beans>
```

3、测试

```
@Test
public void testT(){
   ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
   User user = (User) context.getBean("user");
   user.show();
}
```

结论：在配置文件加载的时候。其中管理的对象都已经初始化了！

5、Spring配置

5.1、别名

alias 设置别名，为bean设置别名，可以设置多个别名

```
<!--设置别名：在获取Bean的时候可以使用别名获取-->
<alias name="userT" alias="userNew"/>
```

5.2、Bean的配置

```
<!--bean就是java对象,由Spring创建和管理-->

<!--
   id 是bean的标识符,要唯一,如果没有配置id,name就是默认标识符
   如果配置id,又配置了name,那么name是别名
   name可以设置多个别名,可以用逗号,分号,空格隔开
   如果不配置id和name,可以根据applicationContext.getBean(.class)获取对象;

class是bean的全限定名=包名+类名
-->
<bean id="hello" name="hello2 h2,h3;h4" class="com.oldwang.pojo.Hello">
   <property name="name" value="Spring"/>
</bean>
```

5.3、import

团队的合作通过import来实现 .

```
<import resource="{path}/beans.xml"/>
```

6、DI依赖注入

6.1、构造器注入

参考IOC创建方式

6.2、set方式注入【重点】

要求被注入的属性 , 必须有set方法 , set方法的方法名由set + 属性首字母大写 , 如果属性是boolean类型 , 没有set方法 , 是 is .

测试pojo类 :

Address.java

```
public class Address {

     private String address;

     public String getAddress() {
         return address;
    }

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

Student.java

```java
public class Student {

     private String name;
     private Address address;
     private String[] books;
     private List<String> hobbys;
     private Map<String,String> card;
     private Set<String> games;
     private String wife;
     private Properties info;

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

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

     public void setBooks(String[] books) {
         this.books = books;
    }

     public void setHobbys(List<String> hobbys) {
         this.hobbys = hobbys;
    }

     public void setCard(Map<String, String> card) {
         this.card = card;
    }

     public void setGames(Set<String> games) {
         this.games = games;
    }

     public void setWife(String wife) {
         this.wife = wife;
    }

     public void setInfo(Properties info) {
         this.info = info;
    }

     public void show(){
         System.out.println("name="+ name
                 + ",address="+ address.getAddress()
                 + ",books="
        );
         for (String book:books){
             System.out.print("<<"+book+">>\t");
        }
         System.out.println("\n爱好:"+hobbys);
```

```
        System.out.println("card:"+card);

        System.out.println("games:"+games);

        System.out.println("wife:"+wife);

        System.out.println("info:"+info);

    }
}
```

1、**常量注入**

```
<bean id="student" class="com.oldwang.pojo.Student">
    <property name="name" value="小明"/>
</bean>
```

测试：

```
@Test
public void test01(){
    ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

    Student student = (Student) context.getBean("student");

    System.out.println(student.getName());

}
```

2、**Bean注入**

注意点：这里的值是一个引用，ref

```
<bean id="addr" class="com.oldwang.pojo.Address">
    <property name="address" value="西安"/>
</bean>

<bean id="student" class="com.oldwang.pojo.Student">
    <property name="name" value="小明"/>
    <property name="address" ref="addr"/>
</bean>
```

3、**数组注入**

```
<bean id="student" class="com.oldwang.pojo.Student">
     <property name="name" value="小明"/>
     <property name="address" ref="addr"/>
     <property name="books">
         <array>
             <value>西游记</value>
             <value>红楼梦</value>
             <value>水浒传</value>
         </array>
     </property>
 </bean>
```

4、**List注入**

```
 <property name="hobbys">
     <list>
         <value>听歌</value>
         <value>看电影</value>
         <value>爬山</value>
     </list>
 </property>
```

5、**Map注入**

```
<property name="card">
     <map>
         <entry key="中国邮政" value="456456456465456"/>
         <entry key="建设" value="1456682255511"/>
     </map>
 </property>
```

6、**set注入**

```
<property name="games">
     <set>
         <value>LOL</value>
         <value>BOB</value>
         <value>COC</value>
     </set>
 </property>
```

7、**Null注入**

```
 <property name="wife"><null/></property>
```

8、**Properties注入**

```
<property name="info">
     <props>
         <prop key="学号">20190604</prop>
         <prop key="性别">男</prop>
         <prop key="姓名">小明</prop>
     </props>
 </property>
```

测试结果：

```
D:\soft\Java\jdk\bin\java.exe ...
name = oldwang  address = Address{address='西安'}
<<oldwang>> <<oldli>>
爱好:[听歌, 看电影, 爬山]
card:{oldwang=特别厉害, oldli=特别厉害}
games:[LOL, BOB, COC]
wife:null
info:{学号=20190604, 性别=男, 姓名=小明}
```

6.3、其他方式注入

p命名空间注入

User.java ：【注意：这里没有有参构造器！】

1、P命名空间注入：需要在头文件中加入约束文件

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

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="user" class="com.oldwang.pojo.User"  p:name="oldwang" p:age="18"></bean>
</beans>
```

c命名空间注入

2、c 命名空间注入：需要在头文件中加入约束文件

```
 导入约束 : xmlns:c="http://www.springframework.org/schema/c"

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

 <!--C(构造: Constructor)命名空间 , 属性依然要设置set方法-->
 <bean id="user" class="com.kuang.pojo.User" c:name="oldwang" c:age="18"/>
```

发现问题：爆红了，刚才我们没有写有参构造！

解决：把有参构造器加上，这里也能知道，c 就是所谓的构造器注入！

测试代码：

```
@Test
 public void test02(){
     ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
     User user = (User) context.getBean("user");
     System.out.println(user);
 }
```

6.4、Bean作用域

Table 3. Bean scopes

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring `ApplicationContext`.
session	Scopes a single bean definition to the lifecycle of an HTTP `Session`. Only valid in the context of a web-aware Spring `ApplicationContext`.
application	Scopes a single bean definition to the lifecycle of a `ServletContext`. Only valid in the context of a web-aware Spring `ApplicationContext`.
websocket	Scopes a single bean definition to the lifecycle of a `WebSocket`. Only valid in the context of a web-aware Spring `ApplicationContext`.

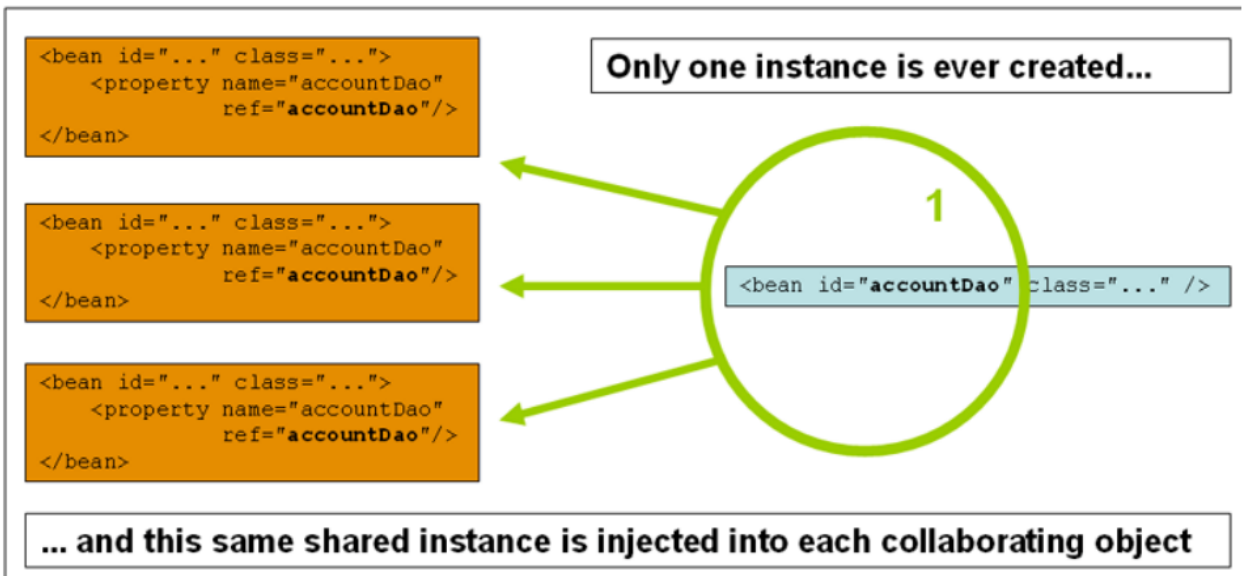
在Spring中，那些组成应用程序的主体及由Spring IoC容器所管理的对象，被称之为bean。简单地讲，**bean就是由IoC容器初始化、装配及管理的对象 .**

几种作用域中，request、session作用域仅在基于web的应用中使用（不必关心你所采用的是什么web应用框架），只能用在基于web的Spring ApplicationContext环境

1. 单例模式（默认）

Singleton

当一个bean的作用域为Singleton，那么Spring IoC容器中只会存在一个共享的bean实例，并且所有对bean的请求，只要id与该bean定义相匹配，则只会返回bean的同一实例。Singleton是单例类型，就是在创建起容器时就同时自动创建了一个bean的对象，不管你是否使用，他都存在了，每次获取到的对象都是同一个对象。注意，Singleton作用域是Spring中的缺省作用域。要在XML中将bean定义成singleton，可以这样配置：

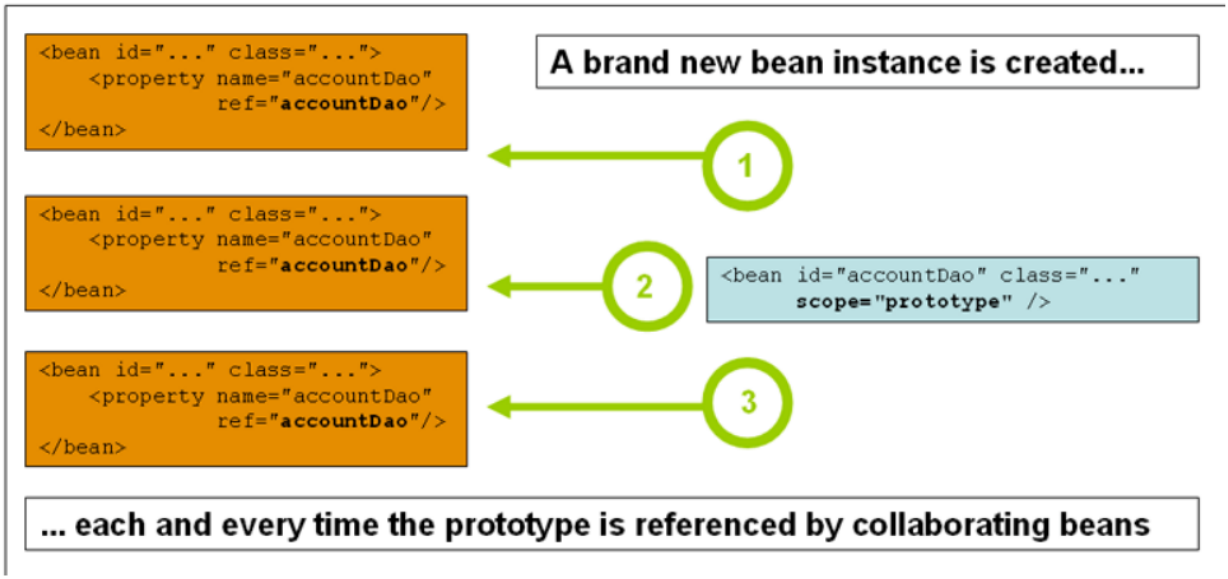



```
<bean id="user2" class="pojo.User" c:name="cxk" c:age="19" scope="singleton"></bean>
```

2. 原型模式: 每次从容器中get的时候，都产生一个新对象！

Prototype

当一个bean的作用域为Prototype，表示一个bean定义对应多个对象实例。Prototype作用域的bean会导致在每次对该bean请求（将其注入到另一个bean中，或者以程序的方式调用容器的getBean()方法）时都会创建一个新的bean实例。Prototype是原型类型，它在我们创建容器的时候并没有实例化，而是当我们获取bean的时候才会去创建一个对象，而且我们每次获取到的对象都不是同一个对象。根据经验，对有状态的bean应该使用prototype作用域，而对无状态的bean则应该使用singleton作用域。在XML中将bean定义成prototype，可以这样配置：




```
<bean id="user2" class="pojo.User" c:name="cxk" c:age="19" scope="prototype"></bean>
```

Request

当一个bean的作用域为Request，表示在一次HTTP请求中，一个bean定义对应一个实例；即每个HTTP请求都会有各自的bean实例，它们依据某个bean定义创建而成。该作用域仅在基于web的Spring ApplicationContext情形下有效。考虑下面bean定义：

```
 <bean id="loginAction" class=cn.csdn.LoginAction" scope="request"/>
```

Spring学习笔记-狂神版

时间:2020-05-30

本文章向大家介绍Spring学习笔记-狂神版，主要包括Spring学习笔记-狂神版使用实例、应用技巧、基本知识点总结和需要注意事项，具有一定的参考价值，需要的朋友可以参考一下。

前言

Mybatis学完开始学Spring

同样先放上参考文档，现在只是需要短时间之内要过一下基础，等考完研再看要不要深入学习吧。

B站 https://www.bilibili.com/video/BV1WE411d7Dv

狂神说Spring01：概述及IOC理论推导 https://mp.weixin.qq.com/s/VM6INdNB_hNfXCMq3UZgTQ

狂神说Spring02：快速上手Spring https://mp.weixin.qq.com/s/Sa39ulmHpNFJ9u48rwCG7A

狂神说Spring03：依赖注入（DI）https://mp.weixin.qq.com/s/Nf-cYENenoZpXqDjv574ig

狂神说Spring04：自动装配 https://mp.weixin.qq.com/s/kvp_3Uva1J2Q5ZVqCUzEsA

狂神说Spring05：使用注解开发 https://mp.weixin.qq.com/s/dCeQwaQ-A97FiUxs7INlHw

狂神说Spring06：静态/动态代理模式 https://mp.weixin.qq.com/s/McxiyucxAQYPSOaJSUCCRQ

狂神说Spring07：AOP就这么简单 https://mp.weixin.qq.com/s/zofgBRRrnEf17MiGZN8IJQ

狂神说Spring08：整合MyBatis https://mp.weixin.qq.com/s/gXFMNU83_7PqTkNZUgvigA

狂神说Spring09：声明式事务 https://mp.weixin.qq.com/s/mYOBJdygHDcXPYBls7cxUA

1、Spring

1.1、简介

Spring : 春天 --->给软件行业带来了春天

2002年，Rod Jahnson首次推出了Spring框架雏形interface21框架。

2004年3月24日，Spring框架以interface21框架为基础，经过重新设计，发布了1.0正式版。

很难想象Rod Johnson的学历，他是悉尼大学的博士，然而他的专业不是计算机，而是音乐学。

Spring理念 : 使现有技术更加实用 . 本身就是一个大杂烩 , 整合现有的框架技术

官网 : http://spring.io/

官方下载地址 : https://repo.spring.io/libs-release-local/org/springframework/spring/

GitHub : https://github.com/spring-projects

1.2、优点

1、Spring是一个开源免费的框架 , 容器 .

2、Spring是一个轻量级的框架 , 非侵入式的 .

3、控制反转 IoC , 面向切面 Aop

4、对事物的支持 , 对框架的支持

.....

一句话概括：

Spring是一个轻量级的控制反转(IoC)和面向切面(AOP)的容器（框架）。

1.3、组成

Spring 框架是一个分层架构，由 7 个定义良好的模块组成。Spring 模块构建在核心容器之上，核心容器定义了创建、配置和管理 bean 的方式 .

组成 Spring 框架的每个模块（或组件）都可以单独存在，或者与其他一个或多个模块联合实现。每个模块的功能如下：

- **核心容器**：核心容器提供 Spring 框架的基本功能。核心容器的主要组件是 BeanFactory，它是工厂模式的实现。BeanFactory 使用*控制反转*（IOC）模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。
- **Spring 上下文**：Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、校验和调度功能。

- **Spring AOP**：通过配置管理特性，Spring AOP 模块直接将面向切面的编程功能，集成到了 Spring 框架中。所以，可以很容易地使 Spring 框架管理任何支持 AOP的对象。Spring AOP 模块为基于 Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，不用依赖组件，就可以将声明性事务管理集成到应用程序中。
- **Spring DAO**：JDBC DAO 抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息。异常层次结构简化了错误处理，并且极大地降低了需要编写的异常代码数量（例如打开和关闭连接）。Spring DAO 的面向 JDBC 的异常遵从通用的 DAO 异常层次结构。
- **Spring ORM**：Spring 框架插入了若干个 ORM 框架，从而提供了 ORM 的对象关系工具，其中包括 JDO、Hibernate 和 iBatis SQL Map。所有这些都遵从 Spring 的通用事务和 DAO 异常层次结构。
- **Spring Web 模块**：Web 上下文模块建立在应用程序上下文模块之上，为基于 Web 的应用程序提供了上下文。所以，Spring 框架支持与 Jakarta Struts 的集成。Web 模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。
- **Spring MVC 框架**：MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成为高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText 和 POI。

1.4、拓展

Spring Boot与Spring Cloud

- Spring Boot 是 Spring 的一套快速配置脚手架，可以基于Spring Boot 快速开发单个微服务;
- Spring Cloud是基于Spring Boot实现的；
- Spring Boot专注于快速、方便集成的单个微服务个体，Spring Cloud关注全局的服务治理框架；
- Spring Boot使用了**约束优于配置的理念**，很多集成方案已经帮你选择好了，能不配置就不配置，Spring Cloud很大的一部分是**基于Spring Boot来实现**，**Spring Boot可以离开Spring Cloud独立使用开发项目**，**但是Spring Cloud离不开Spring Boot，属于依赖的关系。**
- SpringBoot在SpringClound中起到了承上启下的作用，如果你要学习SpringCloud必须要学习SpringBoot。

2、IOC

2.1、IOC基础

新建一个空白的maven项目

2 .1.1、分析实现

我们先用我们原来的方式写一段代码 .

1、先写一个UserDao接口

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

2、再去写Dao的实现类

```
public class UserDaoImpl implements UserDao {
   @Override
   public void getUser() {
       System.out.println("获取用户数据");
  }
}
```

3、然后去写UserService的接口

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

4、最后写Service的实现类

```
public class UserServiceImpl implements UserService {
   private UserDao userDao = new UserDaoImpl();

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

5、测试一下`(这里每次更换不同接口都需要在service层更改new的接口)`

```
@Test
public void test(){
   UserService service = new UserServiceImpl();
   service.getUser();
}
```

这是我们原来的方式，开始大家也都是这么去写的对吧．那我们现在修改一下．

把Userdao的实现类增加一个．

```
public class UserDaoMySqlImpl implements UserDao {
   @Override
   public void getUser() {
       System.out.println("MySql获取用户数据");
  }
}
```

紧接着我们要去使用MySql的话，我们就需要去service实现类里面修改对应的实现

```
public class UserServiceImpl implements UserService {
   private UserDao userDao = new UserDaoMySqlImpl();

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

在假设, 我们再增加一个Userdao的实现类 .

```
public class UserDaoOracleImpl implements UserDao {
   @Override
   public void getUser() {
       System.out.println("Oracle获取用户数据");
  }
}
```

那么我们要使用Oracle , 又需要去service实现类里面修改对应的实现 . 假设我们的这种需求非常大 , 这种方式就根本不适用了, 甚至反人类对吧 , 每次变动 , 都需要修改大量代码 . 这种设计的耦合性太高了, 牵一发而动全身 .

那我们如何去解决呢 ?

我们可以在需要用到他的地方 , 不去实现它 , 而是留出一个接口 , 利用set , 我们去代码里修改下 .

```
public class UserServiceImpl implements UserService {
   private UserDao userDao;
// 利用set实现
   public void setUserDao(UserDao userDao) {
       this.userDao = userDao;
  }

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

现在去我们的测试类里 , 进行测试 ; (有了set方法就可以在调用的时候由用户选择调用的接口)

```
@Test
public void test(){
   UserServiceImpl service = new UserServiceImpl();
   service.setUserDao( new UserDaoMySqlImpl() );
   service.getUser();
   //那我们现在又想用Oracle去实现呢
   service.setUserDao( new UserDaoOracleImpl() );
   service.getUser();
}
```

大家发现了区别没有？可能很多人说没啥区别 . 但是同学们 , 他们已经发生了根本性的变化 , 很多地方都不一样了 . 仔细去思考一下 , **以前所有东西都是由程序去进行控制创建 , 而现在是由我们自行控制创建对象 , 把主动权交给了调用者** . 程序不用去管怎么创建,怎么实现了 . 它只负责提供一个接口 .

这种思想 , 从本质上解决了问题 , 我们程序员不再去管理对象的创建了 , 更多的去关注业务的实现 . 耦合性大大降低 . 这也就是IOC的原型 !

2.1.2IOC本质

控制反转IoC(Inversion of Control)，是一种设计思想，DI(依赖注入)是实现IoC的一种方法，也有人认为DI只是IoC的另一种说法。没有IoC的程序中，我们使用面向对象编程，对象的创建与对象间的依赖关系完全硬编码在程序中，对象的创建由程序自己控制，控制反转后将对象的创建转移给第三方，个人认为所谓控制反转就是：获得依赖对象的方式反转了。

IoC是Spring框架的核心内容，使用多种方式完美的实现了IoC，可以使用XML配置，也可以使用注解，新版本的Spring也可以零配置实现IoC。

Spring容器在初始化时先读取配置文件，根据配置文件或元数据创建与组织对象存入容器中，程序使用时再从Ioc容器中取出需要的对象。

采用XML方式配置Bean的时候，Bean的定义信息是和实现分离的，而采用注解的方式可以把两者合为一体，Bean的定义信息直接以注解的形式定义在实现类中，从而达到了零配置的目的。

控制反转是一种通过描述（XML或注解）并通过第三方去生产或获取特定对象的方式。在Spring中实现控制反转的是IoC容器，其实现方法是依赖注入（Dependency Injection,DI）。

小狂神温馨提示

明白IOC的思想，是理解Spring的核心技巧

3、HelloSpring

3.1、导入Jar包

注 : spring 需要导入commons-logging进行日志记录 . 我们利用maven , 他会自动下载对应的依赖项 .

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-webmvc</artifactId>
   <version>5.2.0.RELEASE</version>
</dependency>
```

3.2、编写代码

1、编写一个Hello实体类

```
public class Hello {
   private String name;

   public String getName() {
```

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

    public void show(){
        System.out.println("Hello,"+ name );
    }
}
```

2、编写我们的spring文件，这里我们命名为beans.xml

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

    <!--bean就是java对象 ， 由Spring创建和管理-->
    <bean id="hello" class="com.kuang.pojo.Hello">
        <property name="name" value="Spring"/>
    </bean>

</beans>
```

3、我们可以去进行测试了 .

```
@Test
public void test(){
    //解析beans.xml文件 ， 生成管理相应的Bean对象
    ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
    //getBean : 参数即为spring配置文件中bean的id .
    Hello hello = (Hello) context.getBean("hello");
    hello.show();
}
```

3.3、思考

- Hello 对象是谁创建的？【hello 对象是由Spring创建的
- Hello 对象的属性是怎么设置的？hello 对象的属性是由Spring容器设置的

这个过程就叫控制反转 :

- 控制 : 谁来控制对象的创建 , 传统应用程序的对象是由程序本身控制创建的 , 使用Spring后 , 对象是由Spring来创建的
- 反转 : 程序本身不创建对象 , 而变成被动的接收对象 .

依赖注入 : 就是利用set方法来进行注入的.

IOC是一种编程思想 , 由主动的编程变成被动的接收

可以通过newClassPathXmlApplicationContext去浏览一下底层源码 .

3.4、修改案例一

我们在案例一中， 新增一个Spring配置文件beans.xml

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

    <bean id="MysqlImpl" class="com.kuang.dao.impl.UserDaoMySqlImpl"/>
    <bean id="OracleImpl" class="com.kuang.dao.impl.UserDaoOracleImpl"/>

    <bean id="ServiceImpl" class="com.kuang.service.impl.UserServiceImpl">
        <!--注意: 这里的name并不是属性 ， 而是set方法后面的那部分 ， 首字母小写-->
        <!--引用另外一个bean ， 不是用value 而是用 ref-->
        <property name="userDao" ref="OracleImpl"/><!--具体使用哪个接口这里可以直接配置-->
    </bean>

</beans>
```

测试！

```
@Test
public void test2(){
    ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
    UserServiceImpl serviceImpl = (UserServiceImpl) context.getBean("ServiceImpl");//这里相当于将原来的Service层也IOC了，不需要再在代码中写出调用哪个接口，只需要在配置文件中指明调用的接口即可。
    serviceImpl.getUser();
     //原来的步骤
     //UserService userService = new UserServiceImpl();
     //userService.setUserDao(new UserDaoMysqlImpl());//原先需要在代码中调用特定的方法
     //userService.getUser();
}
```

OK，到了现在，我们彻底不用再程序中去改动了，要实现不同的操作，只需要在xml配置文件中进行修改，所谓的IoC,一句话搞定：对象由Spring 来创建，管理，装配！

4、IOC创建对象的方式

4.1、通过无参构造方法来创建

1、User.java

```
public class User {

    private String name;
```

```
    public User() {
        System.out.println("user无参构造方法");
    }

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

    public void show(){
        System.out.println("name="+ name );
    }

}
```

2、beans.xml

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

    <bean id="user" class="com.kuang.pojo.User">
        <property name="name" value="kuangshen"/>
    </bean>

</beans>
```

3、测试类

```
@Test
public void test(){
    ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
    //在执行getBean的时候, user已经创建好了 , 通过无参构造
    User user = (User) context.getBean("user");
    //调用对象的方法 .
    user.show();
}
```

结果可以发现，在调用show方法之前，User对象已经通过无参构造初始化了！

4.2、通过有参构造方法来创建

1、UserT . java

```
public class UserT {

    private String name;

    public UserT(String name) {
        this.name = name;
```

```
    }

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

    public void show(){
        System.out.println("name="+ name );
    }

}
```

2、beans.xml 有三种方式编写

```
<!-- 第一种根据index参数下标设置 -->
<bean id="userT" class="com.kuang.pojo.UserT">
    <!-- index指构造方法 , 下标从0开始 -->
    <constructor-arg index="0" value="kuangshen2"/>
</bean>
<!-- 第二种根据参数名字设置 -->
<bean id="userT" class="com.kuang.pojo.UserT">
    <!-- name指参数名 -->
    <constructor-arg name="name" value="kuangshen2"/>
</bean>
<!-- 第三种根据参数类型设置(不推荐使用) -->
<bean id="userT" class="com.kuang.pojo.UserT">
    <constructor-arg type="java.lang.String" value="kuangshen2"/>
</bean>
```

3、测试

```
@Test
public void testT(){
    ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
    UserT user = (UserT) context.getBean("userT");
    user.show();
}
```

结论：在配置文件加载的时候。其中管理的对象都已经初始化了！

5、Spring配置

5.1、别名

alias 设置别名，为bean设置别名，可以设置多个别名

```
<!--设置别名：在获取Bean的时候可以使用别名获取-->
<alias name="userT" alias="userNew"/>
```

5.2、Bean的配置

```
<!--bean就是java对象,由Spring创建和管理-->

<!--
   id 是bean的标识符,要唯一,如果没有配置id,name就是默认标识符
   如果配置id,又配置了name,那么name是别名
   name可以设置多个别名,可以用逗号,分号,空格隔开
   如果不配置id和name,可以根据applicationContext.getBean(.class)获取对象;

class是bean的全限定名=包名+类名
-->
<bean id="hello" name="hello2 h2,h3;h4" class="com.kuang.pojo.Hello">
   <property name="name" value="Spring"/>
</bean>
```

5.3、import

团队的合作通过import来实现 .

```
<import resource="{path}/beans.xml"/>
```

6、DI依赖注入

6.1、构造器注入

4、已经说过

6.2、set方式注入【重点】

要求被注入的属性 , 必须有set方法 , set方法的方法名由set + 属性首字母大写 , 如果属性是boolean类型 , 没有set方法 , 是 is .

测试pojo类 :

Address.java

```java
public class Address {

    private String address;

    public String getAddress() {
        return address;
    }

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

Student.java

```java
package com.kuang.pojo;

import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

public class Student {

    private String name;
    private Address address;
    private String[] books;
    private List<String> hobbys;
    private Map<String,String> card;
    private Set<String> games;
    private String wife;
    private Properties info;

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

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

    public void setBooks(String[] books) {
        this.books = books;
    }

    public void setHobbys(List<String> hobbys) {
        this.hobbys = hobbys;
    }

    public void setCard(Map<String, String> card) {
        this.card = card;
    }
```

```
    public void setGames(Set<String> games) {
        this.games = games;
    }

    public void setWife(String wife) {
        this.wife = wife;
    }

    public void setInfo(Properties info) {
        this.info = info;
    }

    public void show(){
        System.out.println("name="+ name
                + ",address="+ address.getAddress()
                + ",books="
        );
        for (String book:books){
            System.out.print("<<"+book+">>\t");
        }
        System.out.println("\n爱好:"+hobbys);

        System.out.println("card:"+card);

        System.out.println("games:"+games);

        System.out.println("wife:"+wife);

        System.out.println("info:"+info);

    }
}
```

1、**常量注入**

```
<bean id="student" class="com.kuang.pojo.Student">
    <property name="name" value="小明"/>
</bean>
```

测试：

```
@Test
public void test01(){
    ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

    Student student = (Student) context.getBean("student");

    System.out.println(student.getName());

}
```

2、**Bean注入**

注意点：这里的值是一个引用，ref

```
<bean id="addr" class="com.kuang.pojo.Address">
    <property name="address" value="重庆"/>
</bean>

<bean id="student" class="com.kuang.pojo.Student">
    <property name="name" value="小明"/>
    <property name="address" ref="addr"/>
</bean>
```

3、**数组注入**

```
<bean id="student" class="com.kuang.pojo.Student">
    <property name="name" value="小明"/>
    <property name="address" ref="addr"/>
    <property name="books">
        <array>
            <value>西游记</value>
            <value>红楼梦</value>
            <value>水浒传</value>
        </array>
    </property>
</bean>
```

4、**List注入**

```
<property name="hobbys">
    <list>
        <value>听歌</value>
        <value>看电影</value>
        <value>爬山</value>
    </list>
</property>
```

5、**Map注入**

```
<property name="card">
    <map>
        <entry key="中国邮政" value="456456456465456"/>
        <entry key="建设" value="1456682255511"/>
    </map>
</property>
```

6、**set注入**

```
<property name="games">
    <set>
        <value>LOL</value>
        <value>BOB</value>
        <value>COC</value>
    </set>
</property>
```

7、**Null注入**

```
<property name="wife"><null/></property>
```

8、**Properties注入**

```
<property name="info">
    <props>
        <prop key="学号">20190604</prop>
        <prop key="性别">男</prop>
        <prop key="姓名">小明</prop>
    </props>
</property>
```

测试结果：

6.3、其他方式注入

p命名空间注入

User.java ：【注意：这里没有有参构造器！】

```
public class User {
    private String name;
    private int age;

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

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

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

```
}
```

1、P命名空间注入：需要在头文件中加入约束文件

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

<!--P(属性: properties)命名空间 ， 属性依然要设置set方法-->
<bean id="user" class="com.kuang.pojo.User" p:name="狂神" p:age="18"/>
```

c命名空间注入

2、c 命名空间注入：需要在头文件中加入约束文件

```
导入约束 ： xmlns:c="http://www.springframework.org/schema/c"
<!--C(构造: Constructor)命名空间 ， 属性依然要设置set方法-->
<bean id="user" class="com.kuang.pojo.User" c:name="狂神" c:age="18"/>
```

发现问题：爆红了，刚才我们没有写有参构造！

解决：把有参构造器加上，这里也能知道，c 就是所谓的构造器注入！

测试代码：

```
@Test
public void test02(){
    ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
    User user = (User) context.getBean("user");
    System.out.println(user);
}
```

6.4、Bean的作用域

在Spring中，那些组成应用程序的主体及由Spring IoC容器所管理的对象，被称之为bean。简单地讲，**bean就是由IoC容器初始化、装配及管理的对象 .**

几种作用域中，request、session作用域仅在基于web的应用中使用（不必关心你所采用的是什么web应用框架），只能用在基于web的Spring ApplicationContext环境。

Singleton

当一个bean的作用域为Singleton，那么Spring IoC容器中只会存在一个共享的bean实例，并且所有对bean的请求，只要id与该bean定义相匹配，则只会返回bean的同一实例。Singleton是单例类型，就是在创建起容器时就同时自动创建了一个bean的对象，不管你是否使用，他都存在了，每次获取到的对象都是同一个对象。注意，Singleton作用域是Spring中的缺省作用域。要在XML中将bean定义成singleton，可以这样配置：

```
<bean id="ServiceImpl" class="cn.csdn.service.ServiceImpl" scope="singleton">
```

单例模式也就是只new一次对象，之后getBean的都直接获取第一次new的对象

测试：

```
@Test
public void test03(){
    ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
    User user = (User) context.getBean("user");
    User user2 = (User) context.getBean("user2");//第二次getBean
    System.out.println(user==user2);
}
```

Prototype

当一个bean的作用域为Prototype，表示一个bean定义对应多个对象实例。Prototype作用域的bean会导致在每次对该bean请求（将其注入到另一个bean中，或者以程序的方式调用容器的getBean()方法）时都会创建一个新的bean实例。Prototype是原型类型，它在我们创建容器的时候并没有实例化，而是当我们获取bean的时候才会去创建一个对象，而且我们每次获取到的对象都不是同一个对象。根据经验，对有状态的bean应该使用prototype作用域，而对无状态的bean则应该使用singleton作用域。在XML中将bean定义成prototype，可以这样配置：

```
<bean id="account" class="com.foo.DefaultAccount" scope="prototype"/>
 或者
<bean id="account" class="com.foo.DefaultAccount" singleton="false"/>
```

原型模式也就是在之后的getBean时重新new一个对象

Request

当一个bean的作用域为Request，表示在一次HTTP请求中，一个bean定义对应一个实例；即每个HTTP请求都会有各自的bean实例，它们依据某个bean定义创建而成。该作用域仅在基于web的Spring ApplicationContext情形下有效。考虑下面bean定义：

```
<bean id="loginAction" class=cn.csdn.LoginAction" scope="request"/>
```

针对每次HTTP请求，Spring容器会根据loginAction bean的定义创建一个全新的LoginAction bean实例，且该loginAction bean实例仅在当前HTTP request内有效，因此可以根据需要放心的更改所建实例的内部状态，而其他请求中根据loginAction bean定义创建的实例，将不会看到这些特定于某个请求的状态变化。当处理请求结束，request作用域的bean实例将被销毁。

Session

当一个bean的作用域为Session，表示在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。考虑下面bean定义：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

针对某个HTTP Session，Spring容器会根据userPreferences bean定义创建一个全新的userPreferences bean实例，且该userPreferences bean仅在当前HTTP Session内有效。与request作用域一样，可以根据需要放心的更改所创建实例的内部状态，而别的HTTP Session中根据userPreferences创建的实例，将不会看到这些特定于某个HTTP Session的状态变化。当HTTP Session最终被废弃的时候，在该HTTP Session作用域内的bean

也会被废弃掉。

7、Bean的自动装配

- 自动装配是Spring满足bean依赖的一种方式
- Spring会在上下文自动寻找，并自动给bean装配属性

在Spring中有三种装配的方式

1. 在xml中显示配置
2. 在java中显示配置
3. 隐式的自动装配bean 【重要】
4. 这里我们主要讲第三种：自动化的装配bean。

   Spring的自动装配需要从两个角度来实现，或者说是两个操作：

   1. 组件扫描(component scanning)：spring会自动发现应用上下文中所创建的bean；
   2. 自动装配(autowiring)：spring自动满足bean之间的依赖，也就是我们说的IoC/DI；

   组件扫描和自动装配组合发挥巨大威力，使得显示的配置降低到最少。

7.1 测试

1. 环境搭建：一个人有两个宠物

```
public class People {

    private Cat cat;
    private Dog dog;
    private String name;
}
```

```
public class Cat {

    public void shot(){
        System.out.println("miao....");
    }
}
```

```
public class Dog {

    public void shot(){
        System.out.println("wang....");
    }
}
```

配置文件

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

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

    <bean id="cat" class="com.oldwang.pojo.Cat"></bean>

    <bean id="dog" class="com.oldwang.pojo.Dog"></bean>

    <bean id="people" class="com.oldwang.pojo.People">
        <property name="name" value="oldwang"></property>
        <property name="cat" ref="cat"></property>
        <property name="dog" ref="dog"></property>

    </bean>
</beans>
```

2. byType自动装配：byType会自动查找，和自己对象set方法参数的类型相同的bean

   保证所有的class唯一(类为全局唯一)

```xml
<!--
      byType ： 会自动在容器上下文中查找，和自己对象属性类型相同的beanId
-->
<bean id="people" class="com.oldwang.pojo.People" autowire="byType">
    <property name="name" value="oldwang"></property>
</bean>
```

3. byName自动装配：byName会自动查找，和自己对象set对应的值对应的id

   保证所有id唯一，并且和set注入的值一致

```xml
<!--
      byName ： 会自动在容器上下文中查找，和自己对象set方法后面的值对应的beanId
 -->
<bean id="people" class="com.oldwang.pojo.People" autowire="byName">
    <property name="name" value="oldwang"></property>
</bean>
```

7.2 注解实现自动装配

jdk1.5支持的注解，spring2.5支持的注解

The introduction of annotation-based configuration raised the question of whether this approach is “better” than XML.

使用注解须知

1. 导入context约束 context约束
2. 配置注解的支持 context:annotation-config/

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

</beans>
```

@Autowired

直接在属性上使用，也可以再set方法上

使用@Autowired 我们可以不用编写set方法了 前提是你这个自动装配

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <!--开启注解支持-->
    <context:annotation-config/>

    <bean id="cat" class="com.oldwang.pojo.Cat"/>
    <bean id="dog" class="com.oldwang.pojo.Dog"/>
    <bean id="people" class="com.oldwang.pojo.People"/>
</beans>
```

```java
public class People {

    @Autowired
    private Cat cat;
    @Autowired
    private Dog dog;
    private String name;
}
```

科普

```
@Nullable 字段标志的注解，说明这个字段可以为null
```

如果@Autowired自动装配环境比较复杂。自动装配无法通过一个注解完成的时候

我们可以使用@Qualifier(value = "dog")去配合使用，指定一个唯一的id对象

```
public class People {

    @Autowired
    private Cat cat;
    @Autowired
    @Qualifier(value = "dog")
    private Dog dog;
    private String name;
}
```

@Autowired

- @Autowired是按类型自动转配的，不支持id匹配。
- 需要导入 spring-aop的包！
- @Autowired(required=false) 说明：false，对象可以为null；true，对象必须存对象，不能为null。

@Qualifier

- @Autowired是根据类型自动装配的，加上@Qualifier则可以根据byName的方式自动装配
- @Qualifier不能单独使用。

@Resource

- @Resource如有指定的name属性，先按该属性进行byName方式查找装配；
- 其次再进行默认的byName方式进行装配；
- 如果以上都不成功，则按byType的方式自动装配。
- 都不成功，则报异常。

小结

@Autowired与@Resource异同：

1、@Autowired与@Resource都可以用来装配bean。都可以写在字段上，或写在setter方法上。

2、@Autowired默认**按类型装配**（属于spring规范），默认情况下必须要求依赖对象必须存在，如果要允许null 值，可以设置它的required属性为false，如：@Autowired(required=false) ，如果我们想使用名称装配可以结合@Qualifier注解进行使用

3、@Resource（属于J2EE复返），默认**按照名称进行装配**，名称可以通过name属性进行指定。如果没有指定name属性，当注解写在字段上时，默认取字段名进行按照名称查找，如果注解写在setter方法上默认取属性名进行装配。当找不到与名称匹配的bean时才按照类型进行装配。但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配。

它们的作用相同都是用注解方式注入对象，但执行顺序不同。@Autowired先byType，@Resource先byName。

8、使用注解开发

8.1、说明

在spring4之后，必须要保证aop的包导入

使用注解需要导入contex的约束

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

</beans>
```

8.2、Bean的实现

我们之前都是使用 bean 的标签进行bean注入，但是实际开发中，我们一般都会使用注解！

1、配置扫描哪些包下的注解

```
<!--指定注解扫描包-->
<context:component-scan base-package="com.oldwang.pojo"/>
```

2、在指定包下编写类，增加注解

```
@Component("user")
// 相当于配置文件中 <bean id="user" class="当前注解的类"/>
public class User {
   public String name = "oldwang";
}
```

3、测试

```
@Test
public void test(){
   ApplicationContext applicationContext =
       new ClassPathXmlApplicationContext("beans.xml");
   User user = (User) applicationContext.getBean("user");
   System.out.println(user.name);
}
```

8.3、属性注入

使用注解注入属性

1、可以不用提供set方法，直接在直接名上添加@value("值")

```
@Component("user")
// 相当于配置文件中 <bean id="user" class="当前注解的类"/>
public class User {
   @Value("oldwang")
   // 相当于配置文件中 <property name="name" value="oldwang"/>
   public String name;
}
```

2、如果提供了set方法，在set方法上添加@value("值");

```
@Component("user")
public class User {

   public String name;

   @Value("oldwang")
   public void setName(String name) {
       this.name = name;
  }
}
```

8.4、衍生注解

我们这些注解，就是替代了在配置文件当中配置步骤而已！更加的方便快捷！

@Component三个衍生注解

为了更好的进行分层，Spring可以使用其它三个注解，功能一样，目前使用哪一个功能都一样。

- @Controller：web层
- @Service：service层
- @Repository：dao层

写上这些注解，就相当于将这个类交给Spring管理装配了！

8.5、自动装配注解

在Bean的自动装配参考bean的自动装配

8.6、作用域

@scope

- singleton：默认的，Spring会采用单例模式创建这个对象。关闭工厂 ，所有的对象都会销毁。
- prototype：多例模式。关闭工厂 ，所有的对象不会销毁。内部的垃圾回收机制会回收

```java
@Controller("user")
@Scope("prototype")
public class User {
   @Value("oldwang")
   public String name;
}
```

8.7、小结

XML与注解比较

- XML可以适用任何场景 ，结构清晰，维护方便
- 注解不是自己提供的类使用不了，开发简单方便

xml与注解整合开发 ：推荐最佳实践

- xml管理Bean
- 注解完成属性注入
- 使用过程中， 只需要注意一个问题 必须让注解生效，就需要开启注解的支持

```xml
<!--开启注解支持-->
<context:annotation-config/>
<!--指定要扫描的包 这个包下的注解就会生效-->
<context:component-scan base-package="com.oldwang"></context:component-scan>
```

作用：

- 进行注解驱动注册，从而使注解生效
- 用于激活那些已经在spring容器里注册过的bean上面的注解，也就是显示的向Spring注册
- 如果不扫描包，就需要手动配置bean
- 如果不加注解驱动，则注入的值为null！

9、使用Java的方式配置Spring

完全不使用xml配置

JavaConfig 原来是 Spring 的一个子项目，它通过 Java 类的方式提供 Bean 的定义信息，在 Spring4 的版本，JavaConfig 已正式成为 Spring4 的核心功能 。

测试：

1、编写一个实体类，Dog

```java
@Component  //将这个类标注为Spring的一个组件，放到容器中！
public class Dog {
   public String name = "dog";
}
```

2、新建一个config配置包，编写一个MyConfig配置类

```
@Configuration  //代表这是一个配置类
public class MyConfig {

   @Bean //通过方法注册一个bean，这里的返回值就Bean的类型，方法名就是bean的id！
   public Dog dog(){
       return new Dog();
  }

}
```

3、测试

```
@Test
public void test2(){
   ApplicationContext applicationContext =
           new AnnotationConfigApplicationContext(MyConfig.class);
   Dog dog = (Dog) applicationContext.getBean("dog");
   System.out.println(dog.name);
}
```

导入其他配置如何做呢？

1、我们再编写一个配置类！

```
@Configuration  //代表这是一个配置类
public class MyConfig2 {
}
```

2、在之前的配置类中我们来选择导入这个配置类

```
@Configuration
@Import(MyConfig2.class)  //导入合并其他配置类，类似于配置文件中的 inculde 标签
public class MyConfig {

   @Bean
   public Dog dog(){
       return new Dog();
  }

}
```

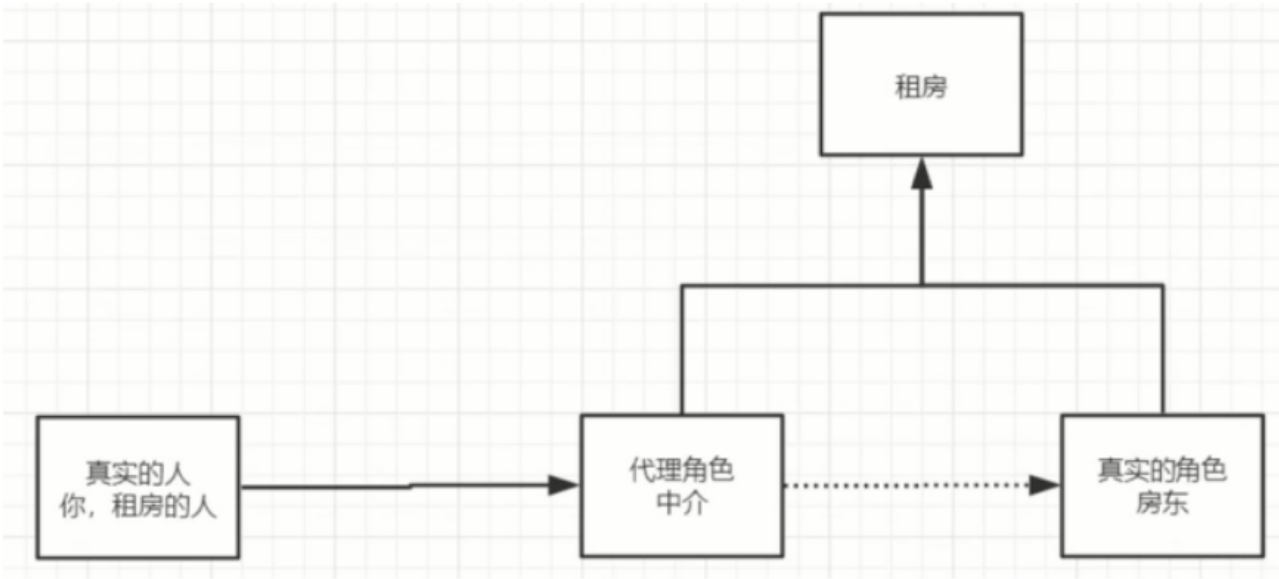
关于这种Java类的配置方式，我们在之后的SpringBoot 和 SpringCloud中还会大量看到，我们需要知道这些注解的作用即可！

10、代理模式

为什么要学习代理模式？ 因为这就是springAOP的底层

代理模式的分类：

- 静态代理
- 动态代理



10.1、静态代理

角色分析：

- 抽象角色： 一般会使用接口或者抽象类来解决
- 真实角色： 被代理的角色
- 代理角色： 代理真实角色， 代理真实角色后，我们一般会做一些附属操作
- 客户： 访问代理对象的人！

代码步骤：

- 接口

```
/**
 * 静态代理模式
 */
public interface Host {
    public void rent();
}
```

- 真是角色

```
public class HostMaster implements Host {
    public void rent() {
        System.out.println("房东要出租房子");
    }
}
```

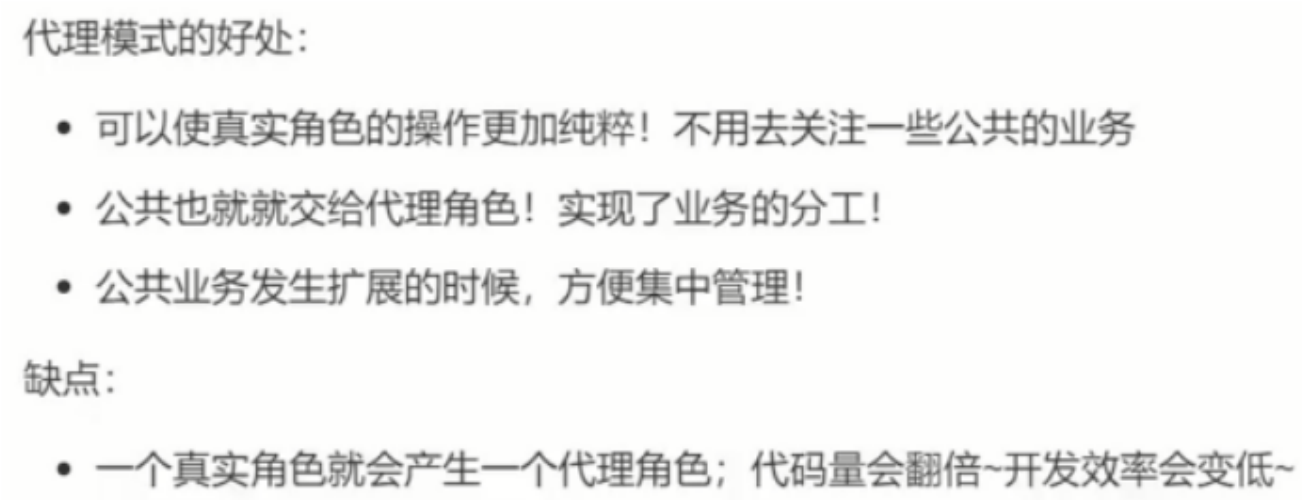
- 代理角色

```
package pojo;
public class Proxy {

    public Host host;

    public Proxy() {

    }

    public Proxy(Host host) {
        super();
        this.host = host;
    }

    public void rent() {
        seeHouse();
        host.rent();
        fee();
        sign();
    }
    //看房
    public void seeHouse() {
        System.out.println("看房子");
    }
    //收费
    public void fee() {
        System.out.println("收中介费");
    }
    //合同
    public void sign() {
        System.out.println("签合同");
    }
}
```

- 客户端访问代理角色

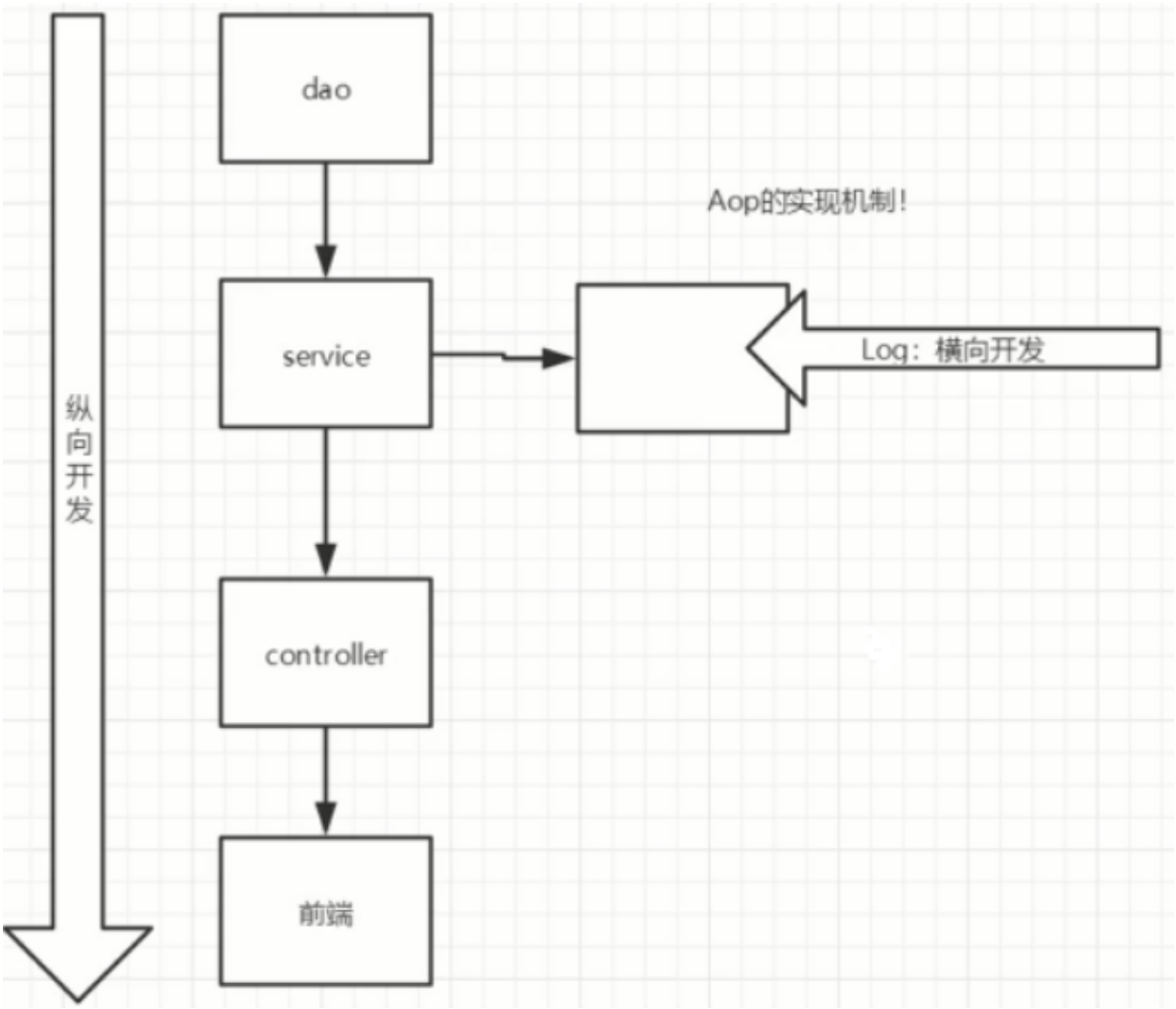
```
package holle4_proxy;

import pojo.Host;
import pojo.HostMaster;
import pojo.Proxy;

public class My {

    public static void main(String[] args) {
        //房东要出租房子
        Host host = new HostMaster();
        //中介帮房东出租房子，但也收取一定费用（增加一些房东不做的操作）
        Proxy proxy = new Proxy(host);
```

```
        //看不到房东，但通过代理，还是租到了房子
        proxy.rent();

    }
}
```



代码翻倍：几十个真实角色就得写几十个代理

AOP横向开发



10.2、动态代理

- 动态代理和静态代理角色一样
- 动态代理类是动态生成的 不是我们直接写好的
- 动态代理也分为两大类：基于接口动态代理，基于类的动态代理
- 基于接口 ---- JDK 动态代理
- 基于类 ---- CGLib

需要了解两个类 Proxy：代理 InvocationHandler：调用处理程序

- 接口

```
public interface Rent {
    public void rent();
}
```

- 真实角色

```
//房东
public class Host implements Rent {
    public void rent() {
        System.out.println("房东要出租房子");
    }
}
```

- 动态代理类

```
//我们会用这个类自动生成代理类
public class ProxyInvocationHandler implements InvocationHandler {

    //被代理的接口
    private Rent rent;

    public void setRent(Rent rent) {
        this.rent = rent;
    }

    //生成得到代理对象
    public Object getProxy(){
        Object instance = Proxy.newProxyInstance(this.getClass().getClassLoader(),
rent.getClass().getInterfaces(), this);
        return instance;
    }

    //处理代理实例 并返回结果
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
{
        //动态代理的本质就是反射
        seeHouse();
        Object invoke = method.invoke(rent, args);
        fare();
        return invoke;
    }

    public void seeHouse(){
        System.out.println("中介在看房子");
    }

    public void fare(){
        System.out.println("收中介费");
    }
}
```

- 测试

```
public class Client {
    public static void main(String[] args) {
        //真是角色
        Host host = new Host();
        //代理角色 现在没有
        ProxyInvocationHandler proxyInvocationHandler = new ProxyInvocationHandler();
        //通过调用程序处理角色 来处理我们要调用的接口对象
        proxyInvocationHandler.setRent(host);
        Rent proxy = (Rent) proxyInvocationHandler.getProxy(); //这里的proxy就是动态生成
的，我们并没有写
        proxy.rent();

    }
}
```

万能代理

```
//我们会用这个类自动生成代理类
public class ProxyInvocationHandler implements InvocationHandler {

    //被代理的接口
    private Object target;

    public void setTarget(Object target) {
        this.target = target;
    }

    //生成得到代理对象
    public Object getProxy(){
        Object instance = Proxy.newProxyInstance(this.getClass().getClassLoader(),
target.getClass().getInterfaces(), this);
        return instance;
    }

    //处理代理实例 并返回结果
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        //动态代理的本质就是反射
        sysoLog(method.getName());
        Object invoke = method.invoke(target, args);
        return invoke;
    }

    public void sysoLog(String message){
        System.out.println("执行了"+message+"方法");
    }
}
```

测试

```
public class Client {
    public static void main(String[] args) {
        UserService userService = new UserServiceImpl(); //真实角色

        //代理角色
        ProxyInvocationHandler proxyInvocationHandler = new ProxyInvocationHandler();
        //设置要代理的对象
        proxyInvocationHandler.setTarget(userService);
        //动态生成代理类
        UserService proxy = (UserService) proxyInvocationHandler.getProxy();
        proxy.add();
    }
}
```

动态代理的好处

- 可以使真实的角色操作更加纯粹，不用关注一些公共的业务
- 公共也就可以交给代理角色，实现了业务分工
- 公共业务发生扩展的时候方便集中管理
- 一个动态代理类就是一个接口，一般就是对应的一类业务
- 一个动态代理类可以代理多个类，只要实现了同一个接口

11、AOP

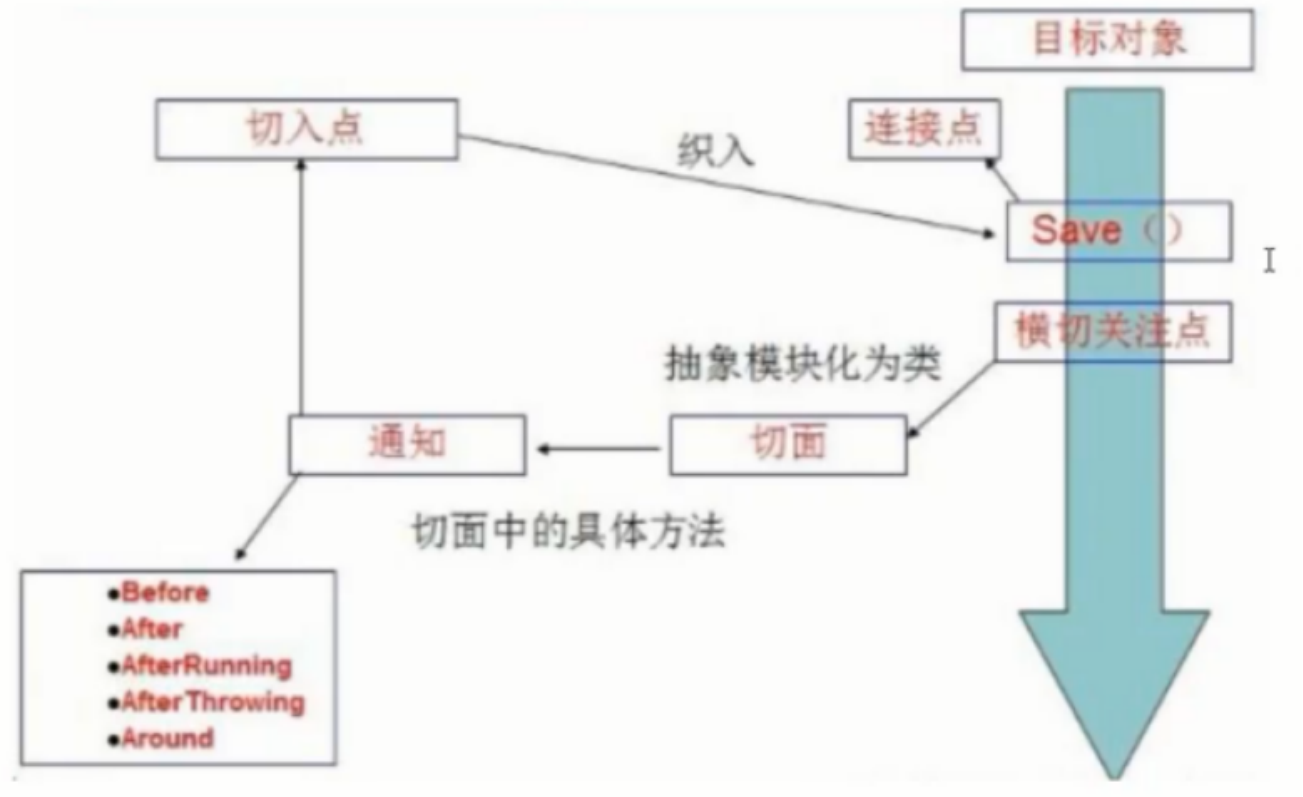
11.1、什么是AOP

AOP（Aspect Oriented Programming）意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

11.2、AOP在Spring中的使用

提供声明式事务，允许用户自定义切面

- 横切关注点：跨越应用程序多个模块的方法或功能。即是，与我们业务逻辑无关的，但是我们需要关注的部分，就是横切关注点。如日志，安全，缓存，事务等等...
- 切面(Aspect)：横切关注点 被模块化的特殊对象。即，它是一个类。（Log类）
- 通知(Advice)：切面必须要完成的工作。即，它是类中的一个方法。（Log类中的方法）
- 目标(Target)：被通知对象。（生成的代理类)
- 代理(Proxy)：向目标对象应用通知之后创建的对象。（生成的代理类）
- 切入点(PointCut)：切面通知执行的"地点"的定义。（最后两点：在哪个地方执行，比如：method.invoke()）
- 连接点(JointPoint)：与切入点匹配的执行点。



SpringAOP中，通过Advice定义横切逻辑，Spring中支持5种类型的Advice:

通知类型	连接点	实现接口
前置通知	方法前	org.springframework.aop.MethodBeforeAdvice
后置通知	方法后	org.springframework.aop.AfterReturningAdvice
环绕通知	方法前后	org.springframework.aop.MethodInterceptor
异常抛出通知	方法抛出异常	org.springframework.aop.ThrowsAdvice
引介通知	类中增加新方法属性	org.springframework.aop.IntroductionOnterceptor

即AOP在不改变原有代码的情况下，去增加新的功能。（代理）

11.3、使用Spring实现AOP

导入jar包

```
<!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.4</version>
</dependency>
```

11.3.1、方法一：使用原生spring接口

springAPI接口实现

applicationContext.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
       xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop = "http://www.springframework.org/schema/aop"
       xsi:schemaLocation = "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

    <!-- bean definition & AOP specific configuration -->
    <bean id="userService" class="com.oldwang.aop.service.UserServiceImpl"></bean>
    <bean id="log" class="com.oldwang.aop.log.SysLog"></bean>
    <bean id="after" class="com.oldwang.aop.log.AfterLog"></bean>
    <!--方式一，使用原生的spring API接口-->
    <!--注册aop-->
    <!--配置aop需要导入aop的约束-->
    <aop:config>
        <!--aop的切入点:expression表达式,execution:要执行的位置-->
        <aop:pointcut id="pointcut" expression="execution(*
com.oldwang.aop.service.UserServiceImpl.*(..))"></aop:pointcut>
        <!--执行环绕增强-->
        <aop:advisor advice-ref="log" pointcut-ref="pointcut"></aop:advisor>
        <aop:advisor advice-ref="after" pointcut-ref="pointcut"></aop:advisor>
    </aop:config>

</beans>
```

接口

```
public interface UserService {

    void add();
    void delete();
    void update();
    void query();
}

public class UserServiceImpl implements UserService {

    public void add() {
        System.out.println("增加一个用户");
    }

    public void delete() {
        System.out.println("删除一个用户");
    }
```

```
    public void update() {
        System.out.println("修改一个用户");
    }

    public void query() {
        System.out.println("查询一个用户");
    }
}
```

日志类

```
public class AfterLog implements AfterReturningAdvice {

    //returnVaule: 返回值
    public void afterReturning(Object returnValue, Method method, Object[] objects, Object
o1) throws Throwable {
        System.out.println("执行了"+method.getName()+"方法，返回结果为"+returnValue);
    }
}


public class SysLog implements MethodBeforeAdvice {
    /**
     *
     * @param method  要执行的方法
     * @param objects 参数
     * @param target 目标对象
     * @throws Throwable
     */
    public void before(Method method, Object[] objects, Object target) throws Throwable {
        System.out.println(target.getClass().getName()+"的"+method.getName()+"被执行了");
    }
}
```

测试类

```
public class SpringAopTest {

    @Test
    public void test(){
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("bean.xml");
        UserService userService = applicationContext.getBean("userService",
UserService.class);
        userService.add();
    }
}
```

11.3.2、方法二：自定义类实现AOP

application.xml

```
    <!--方式二,自定义切面-->
    <bean id="diy" class="com.oldwang.aop.diy.DIYPoint"></bean>
    <aop:config>
        <!--自定义切面，ref:要引用的类-->
        <aop:aspect ref="diy">
            <aop:pointcut id="point" expression="execution(*
com.oldwang.aop.service.UserServiceImpl.*(..))"></aop:pointcut>
            <!--通知-->
            <aop:before method="before" pointcut-ref="point"></aop:before>
            <aop:after method="after" pointcut-ref="point"></aop:after>
        </aop:aspect>
    </aop:config>
```

切面类

```
package com.kuang.diy;

public class DIYPoint {
    public void before(){
        System.out.println("====方法执行前======");
    }
    public void after(){
        System.out.println("====方法执行后======");
    }
}
```

11.3.3、方法三：使用注解实现

application.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- 注册 -->
    <bean id="userservice" class="service.UserServiceImpl"/>
    <!--方式三，使用注解实现-->
    <bean id="diyAnnotation" class="diy.DiyAnnotation"></bean>

    <!-- 开启自动代理
        实现方式：默认JDK (proxy-targer-class="fasle")
                  cgbin (proxy-targer-class="true")-->
    <aop:aspectj-autoproxy/>

</beans>
```

DiyAnnotation.java

```
//使用注解aop
@Aspect //表示这是一个切面类
public class DiyAnnotation {

    @Before("execution(* com.oldwang.aop.service.UserServiceImpl.*(..))")
    public void before(){
        System.out.println("======方法执行前=======");
    }
    @After("execution(* com.oldwang.aop.service.UserServiceImpl.*(..))")
    public void after(){
        System.out.println("======方法执行后=======");
    }
    //在环绕增强中，我们可以给地暖管一个参数，代表我们要获取切入的点
    @Around("execution(* com.oldwang.aop.service.UserServiceImpl.*(..))")
    public void arrount(ProceedingJoinPoint jp) throws Throwable{
        System.out.println("环绕前");
        Object proceed = jp.proceed();
        System.out.println("环绕后" + proceed);
    }
}
```

项目中使用

```
package com.naver.dict.api.aop;

import com.alibaba.fastjson.JSON;
import com.naver.dict.api.dto.common.Entry;
import com.naver.dict.api.util.MailSenderUtil;
import com.naver.dict.common.util.JedisUtil;
import com.naver.dict.openpro.batch.validation.utils.ValidationUtil;
import com.nhncorp.lucy.spring.core.data.DataHandlerFactory;
import org.apache.commons.lang3.StringUtils;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.EnableAspectJAutoProxy;
import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.stereotype.Component;
import java.util.ArrayList;
import java.util.List;

/**
 * @Auther: wangheng
```

```
 * @Date: 2020-07-13
 * @Description: 整合DB Entry Validation切面类
 */
@Aspect
@Component
@EnableAspectJAutoProxy
@EnableAsync
public class EntryValidationAop {

    private Logger logger = LoggerFactory.getLogger(getClass());

    @Pointcut("execution(* com.naver.dict.common.bo.CommonEntryBOImpl.getEntrys(..))")
    public void getEntrys() {
    }

    /**
     * 针对批量查询单词详情
     *
     * @param joinPoint joinPoint
     * @param entrys    多个词条返回结果
     */
    @AfterReturning(value = "getEntrys()", returning = "entrys")
    @Async("sendMailExector")
    public void getEntrys(JoinPoint joinPoint, List<Entry> entrys) {
        try {
            if (entrys != null && entrys.size() > 0) {
                boolean check;
                List message = new ArrayList();
                check = ValidationUtil.check(entrys, message,
CommonBoVerificationRules.values());
                if (!check) {
                    Object[] args = joinPoint.getArgs();
                    List<String> list = null;
                    for (Object ars : args) {
                        if (ars instanceof List) {
                            list = (List) ars;
                            break;
                        }
                    }
                    logger.error("common DB EntryValidationAop batch getEntrys error! {}",
JSON.toJSONString(entrys));
                    sendMessage(message, entrys.get(0), list);
                }
            }
        } catch (Exception e) {
            logger.error("common DB EntryValidationAop batch getEntrys error", e);
        }
    }

}
```

12、整合mybatis

mybatis-spring官网：https://mybatis.org/spring/zh/

mybatis的配置流程：

1. 编写实体类
2. 编写核心配置文件
3. 编写接口
4. 编写Mapper.xmi
5. 测试

12.1、mybatis-spring-方式一

1. 编写数据源配置
2. sqlSessionFactory
3. sqlSessionTemplate（相当于sqlSession）
4. 需要给接口加实现类【new】
5. 将自己写的实现类，注入到Spring中
6. 测试！

先导入jar包

```
<dependencies>

    <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.2.7.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.9.4</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.2.7.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.2</version>
    </dependency>

    <dependency>
```

```
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>2.0.4</version>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.12</version>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.12</version>
    </dependency>

</dependencies>

<!--在build中配置resources，来防止资源导出失败的问题-->
<!-- Maven解决静态资源过滤问题 -->
<build>
<resources>
    <resource>
        <directory>src/main/java</directory>
        <includes>
            <include>**/*.properties</include>
            <include>**/*.xml</include>
        </includes>
        <filtering>false</filtering>
    </resource>
    <resource>
        <directory>src/main/resources</directory>
        <includes>
            <include>**/*.properties</include>
            <include>**/*.xml</include>
        </includes>
        <filtering>false</filtering>
    </resource>
</resources>
</build>
```

user.java

```
@Data
public class User {
    private int id;
    private String name;
    private String pwd;
}
```

mapper目录下的 UserMapper、UserMapperImpl、UserMapper.xml

UserMapper接口

```
public interface UserMapper {
     List<User> getUser();
}
```

UserMapperImpl

```
public class UserMapperImpl implements UserMapper {

    @Autowired
    private SqlSessionTemplate sqlSessionTemplate;

    public List<User> getUser() {
        UserMapper mapper = sqlSessionTemplate.getMapper(UserMapper.class);
        return mapper.getUser();
    }
}
```

UserMapper.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!--namespace=绑定一个指定的Dao/Mapper接口-->
<mapper namespace="com.oldwang.dao.UserMapper">
    <select id="getUserList" resultType="com.oldwang.pojo.User">
     select * from mybatis.user
  </select>
</mapper>
```

resource目录下的 mybatis-config.xml、spring-dao.xml、applicationContext.xml

mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<!--configuration核心配置文件-->
<configuration>

    <!-- 配置mybatis的log实现为STDOUT_LOGGING -->
    <settings>
        <setting name="logImpl" value="STDOUT_LOGGING"/>
    </settings>

</configuration>
```

spring-dao.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
       xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop = "http://www.springframework.org/schema/aop"
       xsi:schemaLocation = "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
    <!--DataSource:使用Spring的数帮源替换Mybatis的配置 其他数据源：c3p0、dbcp、druid
        这使用Spring提供的JDBC: org.springframework.jdbc.datasource -->
    <!--data source -->
    <bean id="datasource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/mybatis?
useSSL=false&amp;useUnicode=true&amp;characterEncoding=utf-
8&amp;serverTimezone=Asia/Shanghai"/>
        <property name="username" value="root" />
        <property name="password" value="root" />
    </bean>
    <!--sqlSessionFactory-->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="datasource" />
        <!--绑定 mybatis 配置文件-->
        <property name="configLocation" value="classpath:mybatis-config.xml"/>
        <property name="mapperLocations" value="classpath:com/oldwang/dao/*.xml"/>
    </bean>
    <!-- sqlSessionTemplate 就是之前使用的：sqlsession -->
    <bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
        <!-- 只能使用构造器注入sqlSessionFactory 原因：它没有set方法-->
        <constructor-arg index="0" ref="sqlSessionFactory"/>
    </bean>

</beans>
```

applicationContext.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
       xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop = "http://www.springframework.org/schema/aop"
       xsi:schemaLocation = "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

    <!-- 导入spring-dao.xml -->
```

```
    <import resource="spring-dao.xml"/>

    <bean id="userMapper" class="com.oldwang.dao.UserMapperImpl">
        <property name="sqlSessionTemplate" ref="sqlSession"></property>
    </bean>

    <bean id="userMapper2" class="com.oldwang.dao.UserServiceImpl2">
        <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
    </bean>

</beans>
```

测试

```
public class MyTest {
    public static void main(String[] args) {
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        UserMapper userMapper = applicationContext.getBean("userMapper", UserMapper.class);
        List<User> user = userMapper.getUserList();
        user.forEach(System.out::println);
    }
}
```

12.2、mybatis-spring-方式二

UserMapperImpl2

```
package com.oldwang.dao;

public class UserServiceImpl2 extends SqlSessionDaoSupport implements UserMapper {
    @Override
    public List<User> getUserList() {
        SqlSession sqlSession = getSqlSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        return mapper.getUserList();
    }

    public void addUser2(Map<String, Object> map) {

    }

    public List<User> getUserLike(String value) {
        return null;
    }

    public List<User> getUserLimit(Map<String, Integer> map) {
        return null;
    }
}
```

spring-dao.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
       xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop = "http://www.springframework.org/schema/aop"
       xsi:schemaLocation = "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
    <!--DataSource:使用Spring的数帮源替换Mybatis的配置 其他数据源：c3p0、dbcp、druid
        这使用Spring提供的JDBC: org.springframework.jdbc.datasource -->
    <!--data source -->
    <bean id="datasource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/mybatis?
useSSL=false&amp;useUnicode=true&amp;characterEncoding=utf-
8&amp;serverTimezone=Asia/Shanghai"/>
        <property name="username" value="root" />
        <property name="password" value="root" />
    </bean>
    <!--sqlSessionFactory-->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="datasource" />
        <!--绑定 mybatis 配置文件-->
        <property name="configLocation" value="classpath:mybatis-config.xml"/>
        <property name="mapperLocations" value="classpath:com/oldwang/dao/*.xml"/>
    </bean>
</beans>
```

applicationContext.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
       xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop = "http://www.springframework.org/schema/aop"
       xsi:schemaLocation = "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

    <!-- 导入spring-dao.xml -->
    <import resource="spring-dao.xml"/>

    <bean id="userMapper" class="com.oldwang.dao.UserMapperImpl">
        <property name="sqlSessionTemplate" ref="sqlSession"></property>
    </bean>
```

```
    <bean id="userMapper2" class="com.oldwang.dao.UserServiceImpl2">
        <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
    </bean>

</beans>
```

测试

```
public class MyTest {
    public static void main(String[] args) {
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        UserMapper userMapper = applicationContext.getBean("userMapper2",
UserMapper.class);
        List<User> user = userMapper.getUserList();
        user.forEach(System.out::println);
    }
}
```

13. 声明式事务

- 把一组业务当成一个业务来做；要么都成功，要么都失败！
- 事务在项目开发中，十分的重要，涉及到数据的一致性问题
- 确保完整性和一致性

事务的ACID原则： 1、原子性 2、隔离性 3、一致性 4、持久性

ACID参考文章：https://www.cnblogs.com/malaikuangren/archive/2012/04/06/2434760.html

Spring中的事务管理

- 声明式事务：AOP
- 编程式事务：需要再代码中，进行事务管理

声明式事务

先导入jar包

```
<dependencies>

    <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.2.7.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.9.4</version>
    </dependency>
```

```
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.2.7.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.2</version>
    </dependency>

    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>2.0.4</version>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.12</version>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.12</version>
    </dependency>

</dependencies>

<!--在build中配置resources，来防止资源导出失败的问题-->
<!-- Maven解决静态资源过滤问题 -->
<build>
<resources>
    <resource>
        <directory>src/main/java</directory>
        <includes>
            <include>**/*.properties</include>
            <include>**/*.xml</include>
        </includes>
        <filtering>false</filtering>
    </resource>
    <resource>
        <directory>src/main/resources</directory>
        <includes>
            <include>**/*.properties</include>
            <include>**/*.xml</include>
        </includes>
        <filtering>false</filtering>
    </resource>
```

```
</resources>
</build>
```

pojo实体类 User

```
@Data
@ToString
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private int id;
    private String name;
    private String pwd;
}
```

mapper目录下的 UserMapper、UserMapperImpl、UserMapper.xml

接口UserMapper

```
public interface UserMapper {

    public List<User> getUser();

    public int insertUser(User user);

    public int delUser(@Param("id") int id);
}
```

UserMapper.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!--namespace=绑定一个指定的Dao/Mapper接口-->

<!-- 绑定接口 -->
<mapper namespace="com.oldwang.mapper.UserMapper">
    <select id="getUser" resultType="com.oldwang.pojo.User">
        select * from user
    </select>

    <insert id="insertUser"  parameterType="com.oldwang.pojo.User" >
        insert into mybatis.User (id,name,pwd) values (#{id},#{name},#{pwd})
    </insert>

    <delete id="delUser" parameterType="_int">
        <!-- deleteAAAAA是故意写错的 -->
        delete AAAfrom mybatis.user where id = #{id}
    </delete>
```

```
</mapper>
```

UserMapperImpl(extends SqlSessionDaoSupport)

```
package com.oldwang.mapper;

import com.oldwang.pojo.User;
import org.mybatis.spring.support.SqlSessionDaoSupport;

import java.util.List;

public class UserMapperImpl extends SqlSessionDaoSupport implements UserMapper {
    @Override
    public List<User> getUser() {
        UserMapper mapper = getSqlSession().getMapper(UserMapper.class);
        User user = new User(8,"你好","wwww");
        mapper.insertUser(user);
        mapper.delUser(8);
        return mapper.getUser();
    }

    //插入
    @Override
    public int insertUser(User user) {
        return getSqlSession().getMapper(UserMapper.class).insertUser(user);
    }
    //删除
    @Override
    public int delUser(int id) {
        return getSqlSession().getMapper(UserMapper.class).delUser(id);
    }
}
```

spring-dao.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!--DataSource:使用Spring的数帮源替换Mybatis的配置 其他数据源：c3p0、dbcp、druid
        这使用Spring提供的JDBC: org.springframework.jdbc.datasource -->
```

```
    <!--data source -->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/mybatis?
useSSL=false&amp;useUnicode=true&amp;characterEncoding=utf-
8&amp;serverTimezone=Asia/Shanghai"/>
        <property name="username" value="root" />
        <property name="password" value="root" />
    </bean>

    <!--sqlSessionFactory-->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"></property>
        <!--绑定 mybatis 配置文件-->
        <property name="configLocation" value="classpath:mybatis-config.xml"></property>
        <property name="mapperLocations" value="classpath:com/oldwang/mapper/*.xml">
</property>
    </bean>

    <!--声明式事务-->
    <bean id="transaction"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!--结合aop实现事务置入-->
    <tx:advice id="transaction-manager" transaction-manager="transaction">
        <!--给哪些方法配置事务-->
        <!--配置事务的传播特性-->
        <tx:attributes>
            <tx:method name="add" propagation="REQUIRED"/>
            <tx:method name="delete" propagation="REQUIRED"/>
            <tx:method name="update" propagation="REQUIRED"/>
            <tx:method name="*" propagation="REQUIRED"/>
            <tx:method name="query" read-only="true"/>
        </tx:attributes>
    </tx:advice>

    <!--配置事务切入-->
    <aop:config>
        <aop:pointcut id="txpointxut" expression="execution(* com.oldwang.mapper.*.*
(..))"/>
        <aop:advisor advice-ref="transaction-manager" pointcut-ref="txpointxut">
</aop:advisor>
    </aop:config>

</beans>
```

mybatis-config.xml

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

```
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<!--configuration核心配置文件-->
<configuration>

    <!-- 配置mybatis的log实现为STDOUT_LOGGING -->
    <settings>
        <setting name="logImpl" value="STDOUT_LOGGING"/>
    </settings>

</configuration>
```

application.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
       xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop = "http://www.springframework.org/schema/aop"
       xsi:schemaLocation = "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

    <!-- 导入spring-dao.xml -->
    <import resource="spring-dao.xml"/>

    <bean id="userMapper" class="com.oldwang.mapper.UserMapperImpl">
        <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
    </bean>

</beans>
```

测试

```
public class MyTest {
    public static void main(String[] args) {
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        UserMapper userMapper = applicationContext.getBean("userMapper", UserMapper.class);
        List<User> user = userMapper.getUser();
        user.forEach(System.out::println);
    }
}
```