Spring框架

2021年1月11日 20:07

1.简介介绍

- Spring：春天---->Java开发领域带来了春天
- 2002年，首次推出Spring框架雏形：Interface21框架
- Spring框架即以Iterface21框架为基础经过重新设计，并不断丰富其内涵，于2004年3月24日，发布了1.0正式版
- Rod Johnson :Spring Framework创始人，音乐学博士，提出了轮子理论。
- Spring理念：

  When you learn about a framework, it's important to know not only what it does but what principles it follows.
  Here are the guiding principles of the Spring Framework:
  Provide choice at every level. Spring lets you defer design decisions as late as possible. For example, you can switch persistence providers through configuration without changing your code. The same is true for many other infrastructure concerns and integration with third-party APIs.
  Accommodate diverse perspectives. Spring embraces flexibility and is not opinionated about how things should be done. It supports a wide range of application needs with different perspectives.
  Maintain strong backward compatibility. Spring's evolution has been carefully managed to force few breaking changes between versions. Spring supports a carefully chosen range of JDK versions and third-party libraries to facilitate maintenance of applications and libraries that depend on Spring.
  Care about API design. The Spring team puts a lot of thought and time into making APIs that are intuitive and that hold up across many versions and many years.
  Set high standards for code quality. The Spring Framework puts a strong emphasis on meaningful, current, and accurate javadoc. It is one of very few projects that can claim clean code structure with no circular dependencies between packages.

  说明：

  使用现有的技术更加容易使用，本身是一个大型集成框架（大杂烩），整合了现有的技术框架。
  - SHH：Struct2 + Spring + Hibernate
  - SSM：SpringMVC + Spring + Mybatis

- 官网：https://spring.io/
- 官方下载地址：
  https://repo.spring.io/release/org/springframework/spring/
  https://repo.spring.io/release/org/springframework/spring/5.3.2/
  GitHub流下载地址：
  https://github.com/spring-projects/spring-framework/releases/tag/v5.3.2
  Spring Maven：

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.3.2</version>
</dependency>
```

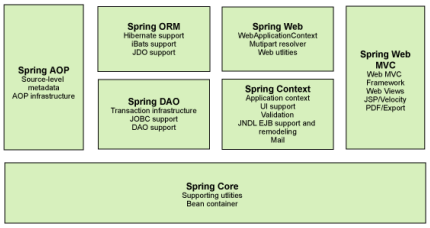
  JDBC：

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>5.3.2</version>
</dependency>
```

- Spring的优点：
  - Spring是一个开源的免费的框架
  - Spring是一个轻量级，非入侵式的框架
  - 控制反转（IOC），面向切面编程（AOP）
  - 支持事务的处理，对框架整合的支持

总结：Spring就是一个轻量级的控制反转（ICO）和面向切面编程（AOP）的框架

2.Spring的组成



七大核心模块：AOP、PRM、Web、DAO、Context、MVC、Core

- **核心容器（Spring Core）**

  核心容器提供Spring框架的基本功能。Spring以bean的方式组织和管理Java应用中的各个组件及其关系。Spring使用BeanFactory来产生和管理Bean，它是工厂模式的实现。BeanFactory使用控制反转(IoC)模式将应用的配置和依赖性规范与实际的应用程序代码分开。

- **应用上下文（Spring Context）**

  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，如JNDI、EJB、电子邮件、国际化、校验和调度功能。

- **Spring面向切面编程（Spring AOP）**

  通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到了 Spring框架中。所以，可以很容易地使 Spring框架管理的任何对象支持 AOP。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 EJB 组件，就可以将声明性事务管理集成到应用程序中。

- **JDBC和DAO模块（Spring DAO）**

  JDBC、DAO的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所抛出的错误信息。异常层次结构简化了错误处理，并且极大的降低了需要编写的代码数量，比如打开和关闭链接。

- **对象实体映射（Spring ORM）**

  Spring框架插入了若干个ORM框架，从而提供了ORM对象的关系工具，其中包括了Hibernate、JDO和 IBatis SQL Map等，所有这些都遵从Spring的通用事物和DAO异常层次结构。

- **Web模块（Spring Web）**

  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Struts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。

- **MVC模块（Spring Web MVC）**

  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、POI等，模型来有JavaBean来构成，存放于m当中，而视图是一个街口，负责实现模型，控制器表示逻辑代码，由c的事情。Spring框架的功能可以用在任何J2EE服务器当中，大多数功能也适用于不受管理的环境。Spring的核心要点就是支持不绑定到特定J2EE服务的可重用业务和数据的访问的对象，毫无疑问这样的对象可以在不同的J2EE环境，独立应用程序和测试环境之间重用。

现代化Java开发，就是基于Spring的开发

Spring Boot 构建一切 + Spring Cloud 协调一切 + Spring Cloud Data Flow 连接一切

- Spring Boot 构建一切
  - 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 协调一切
  - SpringCloud是基于SpringBoot实现的

因为现在大多数公司都在使用SpringBoot进行开发，学习SpringBoot的前提，需要完全掌握Spring及SpringMVC承上启下的作用

弊端

发展太久之后，违背了原来的理念，配置十分繁琐，人称：“配置地狱”

3.IOC理论推导

普通：

1.UserDao 接口

```java
package com.Demo.dao;

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

2.UserDaoImpl 实现类

```java
package com.Demo.dao;

public class UserDaoImpl implements UserDao{
    public void getUser(){
        System.out.println("默认获取用户的数据");
    }
}
```

3.UserService 业务接口

```java
package com.Demo.service;

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

4.UserServiceImpl 业务实现类

```java
package com.Demo.service;

import com.Demo.dao.UserDao;
import com.Demo.dao.UserDaoImpl;
import com.Demo.dao.UserDaoMysqlImpl;
import com.Demo.dao.UserOracleImpl;

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

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

需要不断变化

5.Main函数

```java
import com.Demo.dao.UserDaoImpl;
import com.Demo.service.UserService;
import com.Demo.service.UserServiceImpl;

public class MyText {
    public static void main(String[] args){

        //用户实际调用的是业务层，Dao层他们不需要接触
        UserService userService = new UserServiceImpl();

        userService.getUser();
    }
}
```

解决方案：set方式：在之前的业务中，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原代码如果程序代码量十分大，修改一次的成本代价十分昂贵。我们使用一个set接口实现（**革命性变化，之前程序是主动创建对象，控制权在程序员手里。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收对象。这种思想从本质上解决了问题，程序员不用再去管理对象的创建了，系统的耦合性大大降低，可以更加专注的在业务的实现上，这是IOC的原型**）

4.ICO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，使用了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容器在初始化时要先读取配置文件，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，其实现方式是依赖注入（Dependency Injection，DI）

Hello Spring程序

```java
package com.Deom.pojo;

public class Hello {
    private String name;

    public String getNaem(){
        return name;
    }

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

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

}
```

```java
package com.Deom.pojo;

public class Hello {
    private String str;

    public String getStr() {
        return str;
    }

    public void setStr(String str) {
        this.str = str;
    }

    @Override
    public String toString() {
        return "Hello{" +
                "str='" + str + '\'' +
                '}';
    }
}
```

```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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--使用Spring来创建我们的对象，在Spring这些都称为Bean-->
    <bean id="hello" class="com.Deom.pojo.Hello">
        <property name="str" value="Spring"/>
    </bean>

</beans>
```

```java
import com.Deom.pojo.Hello;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    public static void main(String[] args){
        //获取Spring的上下文对象
        ApplicationContext context = new ClassPathXmlApplicationContext( configLocation: "beans.xml");
        //我们的对象现在都在Spring中管理了，我们要使用，直接去里面取出来就可以了
        Hello hello = (Hello) context.getBean( s: "hello");
        System.out.println(hello.toString());
    }
}
```

类型 变量名 = new 类型（）

Hello hello = new hello

Id = 变量名 class = new 的对象 property 相当于给对象中的属性设置一个值

Bean = 对象 new hello();

```
<beanid="hello"class="com.Deom.pojo.Hello">
<propertyname="str"value="Spring"/>
</bean>
```

总结：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方式来进行注入的

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new ClassPathXmlApplicationContext去浏览一下底层源码

我们彻底不用再程序中去改动，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就是对象由Spring来创建，管理，装配

演示

```
bean.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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="MysqlImpl" class="com.Demo.dao.UserDaoMysqlImpl"/>
    <bean id="OracleImpl" class="com.Demo.dao.UserOracleImpl"/>
    <bean id="UserServiceImpl" class="com.Demo.service.UserServiceImpl">
        <!--
        ref: 引用Spring容器中已经创建好的对象
        value: 具体的值，基本数据类型
        -->
        <property name="userDao" ref="SqlserverImpl"/>
    </bean>

</beans>
```

```
MyText.java
import com.Demo.dao.UserDaoImpl;
import com.Demo.dao.UserDaoMysqlImpl;
import com.Demo.dao.UserDaoSqlserverImpl;
import com.Demo.dao.UserOracleImpl;
import com.Demo.service.UserService;
import com.Demo.service.UserServiceImpl;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyText {
    public static void main(String[] args){
        //获取ApplicationContext，拿到Spring的容器
        ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
        //需要什么直接get。
        UserServiceImpl userServiceImpl = (UserServiceImpl) context.getBean("UserServiceImpl");
        userServiceImpl.getUser();

        //用户实际调用的是业务层，Dao层他们不需要接触
        /* UserService userService = new UserServiceImpl();
        ((UserServiceImpl) userService).setUserDao(new UserDaoSqlserverImpl());
        userService.getUser();
        我们不再需要new 一个对象
        */
    }
}
```

5.IOC创建对象的方式

普通：

```
User.java
package com.Demo.pojo;

public class User {
    private String name;

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

    public String getName() {
        return name;
    }

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

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

```java
import com.Demo.pojo.User;

public class MyTest {
    public static void main(String[] args){
        User user = new User();
    }
}
```

- 使用无参构造创建对象，默认！
- 假设我们要使用有参创造对象。
  - 三种方式：
    - 第一种：下标赋值

      Constructor argument index

      You can use the `index` attribute to specify explicitly the index of constructor arguments, as the following example shows:

      ```xml
      <bean id="exampleBean" class="examples.ExampleBean">
          <constructor-arg index="0" value="7500000"/>
          <constructor-arg index="1" value="42"/>
      </bean>
      ```

      User.java | MyTest.java | 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
              https://www.springframework.org/schema/beans/spring-beans.xsd">

          <!--第一种、下标赋值-->
          <bean id="user" class="com.Demo.pojo.User">
              <constructor-arg index="0" value="SongYuChen"/>
          </bean>

      </beans>
      ```

    - 第二种：通过类型创建 **不推荐**

      Constructor argument type matching

      In the preceding scenario, the container can use type matching with simple types if you explicitly specify the type of the constructor argument by using the `type` attribute. as the following example shows:

      ```xml
      <bean id="exampleBean" class="examples.ExampleBean">
          <constructor-arg type="int" value="7500000"/>
          <constructor-arg type="java.lang.String" value="42"/>
      </bean>
      ```

      User.java | MyTest.java | 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
              https://www.springframework.org/schema/beans/spring-beans.xsd">

          <!--第一种、下标赋值-->
          <!--<bean id="user" class="com.Demo.pojo.User">
              <constructor-arg index="0" value="SongYuChen"/>
          </bean>-->

          <!--第二种、通过类型创建 不推荐-->
          <bean id="user" class="com.Demo.pojo.User">
              <constructor-arg type="java.lang.String" value="SongyuChen"/>
          </bean>

      </beans>
      ```

    - 第三种：通过参数名创建

      ```xml
      <beans>
          <bean id="beanOne" class="x.y.ThingOne">
              <constructor-arg ref="beanTwo"/>
              <constructor-arg ref="beanThree"/>
          </bean>

          <bean id="beanTwo" class="x.y.ThingTwo"/>

          <bean id="beanThree" class="x.y.ThingThree"/>
      </beans>
      ```

      User.java | MyTest.java | beans.xml

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

          <!--第一种、下标赋值-->
          <!--<bean id="user" class="com.Demo.pojo.User">
              <constructor-arg index="0" value="SongYuChen"/>
          </bean>-->

          <!--第二种、通过类型创建 不推荐-->
          <!--<bean id="user" class="com.Demo.pojo.User">
              <constructor-arg type="java.lang.String" value="SongyuChen"/>
          </bean>-->

          <!--第三种：直接通过参数名称来设置-->
          <bean id="user" class="com.Demo.pojo.User">
              <constructor-arg name="name" value="songYuChen"/>
          </bean>

      </beans>
      ```

- 我们再进行无参有参共存：**Spring就类似于婚介网站**

```java
package com.Demo.pojo;

public class UserT {
    private String name;

    public UserT(){
        System.out.println("UserT被创建了");
    }

    public String getName() {
        return name;
    }

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

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

}
```

```xml
    <!--第一种，下标赋值-->
    <!--<bean id="user" class="com.Demo.pojo.User">
        <constructor-arg index="0" value="SongYuChen"/>
    </bean>-->

    <!--第二种，通过类型创建，不推荐-->
    <!--<bean id="user" class="com.Demo.pojo.User">
        <constructor-arg type="java.lang.String" value="Songyu
    </bean>-->

    <!--第三种：直接通过参数名称来设置-->
    <bean id="user" class="com.Demo.pojo.User">
        <constructor-arg name="name" value="songYuChen"/>
    </bean>
    <bean id="userT" class="com.Demo.pojo.UserT"/>

</beans>
```

```
MyTest (1)
D:\BX\Java1.8\bin\java.exe ...
UserT被创建了
name=songYuChen

进程已结束,退出代码0
```

```java
import com.Demo.pojo.User;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    public static void main(String[] args){
        //User user = new User();

        //Spring容器
        ApplicationContext context = new ClassPathXmlApplicationContext( configLocation: "beans.xml");

        //两个User是同一个，说明其内存中只要一份实例。
        User user = (User) context.getBean( s: "user");
        User user2 =(User) context.getBean( s: "user");
        System.out.println(user == user2);
        user.show();
    }
}
```

总结：**在配置文件加载的时候，容器中管理的对象就已经初始化了**

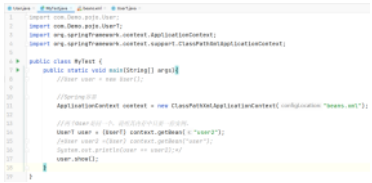
6.Spring的配置

- 别名：其功能就是为对象修改新增一个名称：`<alias name="对象名称" alias="新增名称"/>`

```xml
        <constructor-arg index="0" value="SongYuChen"/>
    </bean>-->

    <!--第二种，通过类型创建 不推荐-->
    <!--<bean id="user" class="com.Demo.pojo.User">
        <constructor-arg type="java.lang.String" value="SongyuChen"/>
    </bean>-->

    <!--第三种，直接通过参数名称来设置-->
    <bean id="user" class="com.Demo.pojo.User">
        <constructor-arg name="name" value="songYuChen"/>
    </bean>
    <bean id="userT" class="com.Demo.pojo.UserT"/>

    <!--别名-->
    <alias name="user" alias="Syc"/>

</beans>
```

```java
import com.Demo.pojo.User;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    public static void main(String[] args){
        //User user = new User();

        //Spring容器
        ApplicationContext context = new ClassPathXmlApplicationContext( configLocation: "beans.xml");

        //两个User是同一个，说明其内存中只要一份实例。
        User user = (User) context.getBean( s: "Syc");
        /*User user2 =(User) context.getBean("user");
        System.out.println(user == user2);*/
        user.show();
    }
}
```

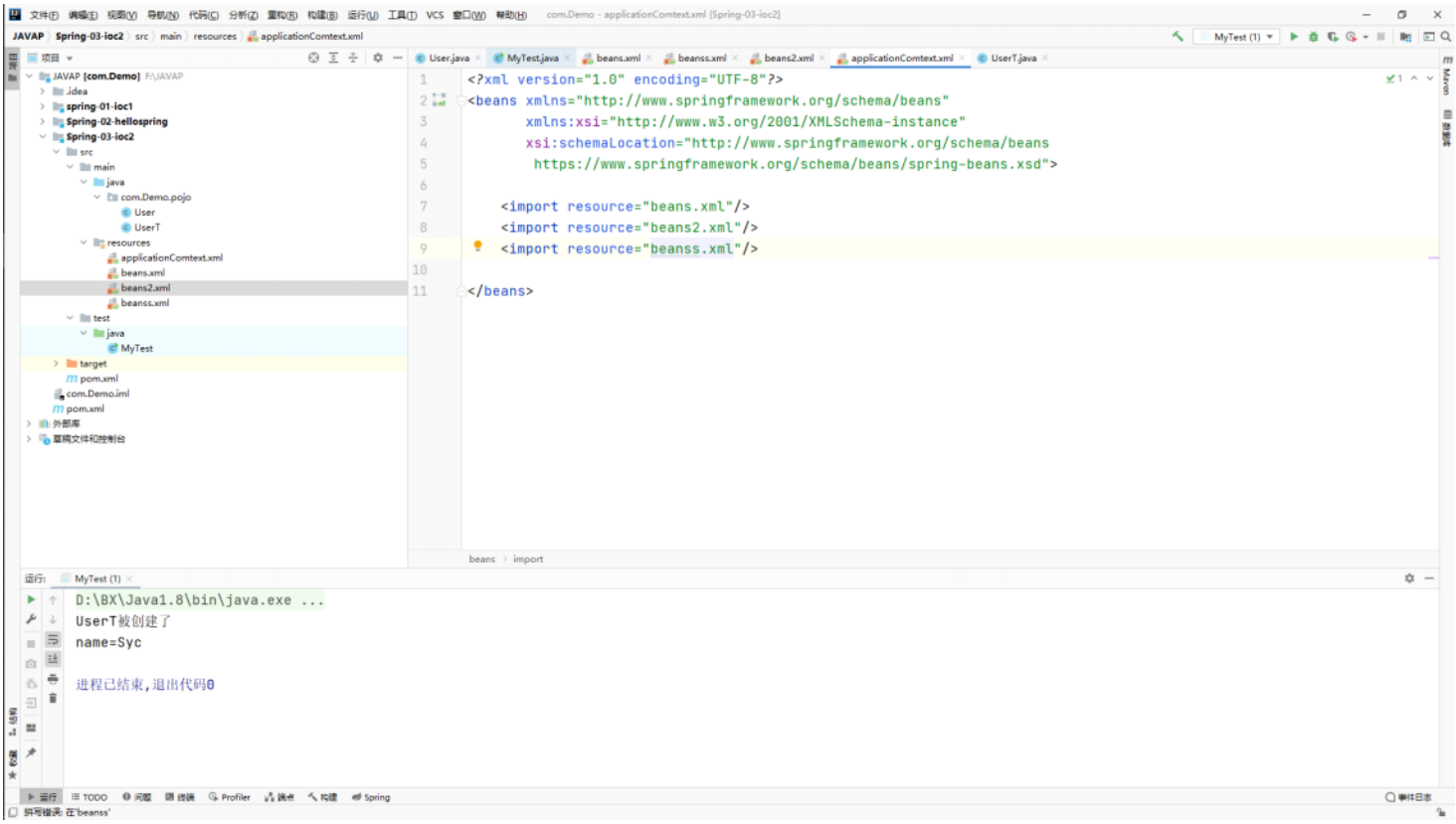
- Bean的配置：**id:bean的唯一标识符，也就是我们的变量名 Class就是bean对象所对应的全限定名（包名+类名）name也是别名，而且name可以同时取多个别名**



多个别名：可以使用（空格，逗号，分隔符）

```xml
    <!--第一种，下标赋值-->
    <!--<bean id="user" class="com.Demo.pojo.User">
        <constructor-arg index="0" value="SongYuChen"/>
    </bean>-->

    <!--第二种，通过类型创建 不推荐-->
    <!--<bean id="user" class="com.Demo.pojo.User">
        <constructor-arg type="java.lang.String" value="SongyuChen"/>
    </bean>-->

    <!--第三种，直接通过参数名称来设置-->
    <bean id="user" class="com.Demo.pojo.User">
        <constructor-arg name="name" value="songYuChen"/>
    </bean>

    <!--Bean-->
    <bean id="userT" class="com.Demo.pojo.UserT" name="user2 u2,u3;u4">
        <property name="name" value="Syc"/>
    </bean>

    <!--别名-->
    <alias name="user" alias="Syc"/>
```

- Import：**一般用于团队开发使用，它可以将多个配置文件，导入合并为一个**
  - 假设：现在项目中有多个人开发，这三个人复制不同的类开发，不同的类需要注册在不同的bean中，我们可以利用import将所有人的beans.xml合并为一个总的使用的时候直接使用总的配置就可以了。（applicationContext.xml）



7.依赖注入（重点是Set注入）

- 构造器注入
- Set方式注入
  - 依赖注入：Set注入
    - 依赖：bean对象的创建依赖于Spring容器
    - 注入：bean对象中的所有属性，有Spring容器来注入
  - 环境搭建:
    - 复杂类型
      - 就是下列Address.java的代码
    - 真实测试对象
      - 就是下列Studebt.java的代码
    - 测试类
      - 就是下列MyText.java的代码

普通值注入（Value）

Studebt.java代码：

```
packagecom.Demo.pojo;

importjava.util.*;

publicclassStudent{

    privateStringname;
    privateAddressaddress;
    privateString[]books;
    privateList<String>hobbys;
    privateMap<String,String>card;
    privateSet<String>games;
    privateStringwife;
    privatePropertiesinfo;

    publicStringgetName(){
        returnname;
    }

    publicvoidsetName(Stringname){
        this.name=name;
    }

    publicAddressgetAddress(){
        returnaddress;
    }

    publicvoidsetAddress(Addressaddress){
        this.address=address;
    }

    publicString[]getBooks(){
        returnbooks;
    }

    publicvoidsetBooks(String[]books){
        this.books=books;
    }

    publicList<String>getHobbys(){
        returnhobbys;
    }

    publicvoidsetHobbys(List<String>hobbys){
        this.hobbys=hobbys;
    }

    publicMap<String,String>getCard(){
        returncard;
    }

    publicvoidsetCard(Map<String,String>card){
        this.card=card;
    }

    publicSet<String>getGames(){
        returngames;
    }

    publicvoidsetGames(Set<String>games){
        this.games=games;
    }

    publicStringgetWife(){
```

```
        returnwife;
    }

    publicvoidsetWife(Stringwife){
        this.wife=wife;
    }

    publicPropertiesgetInfo(){
        returninfo;
    }

    publicvoidsetInfo(Propertiesinfo){
        this.info=info;
    }

    @Override
    publicStringtoString(){
        return"Student{"+
            "name='"+name+'\''+
            ",address="+address+
            ",books="+Arrays.toString(books)+
            ",hobbys="+hobbys+
            ",card="+card+
            ",games="+games+
            ",wife='"+wife+'\''+
            ",info="+info+
        '}';
    }
}
```

Address.java代码

```
packagecom.Demo.pojo;

publicclassAddress{
    privateStringaddress;

    publicStringgetAddress(){
        returnaddress;
    }

    publicvoidsetAddress(Stringaddress){
        this.address=address;
    }
}
```

pom.xml (Spring-04-di) | bean.xml | MyTest.java | Student.java | Address.java

```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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="student" class="com.Demo.pojo.Student">
        <property name="name" value="Arvin"/>
    </bean>

</beans>
```

bean.xml | MyTest.java | Student.java | Address.java

```java
import com.Demo.pojo.Student;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    public static void main(String[] args){
        ApplicationContext context = new ClassPathXmlApplicationContext( configLocation: "bean.xml");
        Student student = (Student) context.getBean( s: "student");
        System.out.println(student.getName());
    }
}
```

第一种：其他代码在上面（一下基于上面的环境）

```xml
    <bean id="student" class="com.Demo.pojo.Student">
        <!--第一种，普通值注入，  Value-->
        <property name="name" value="Arvin"/>
    </bean>
```

第二种：Bean注入

```xml
<!--第二种，address的定义-->
<bean id="address" class="com.Demo.pojo.Address"/>

<bean id="student" class="com.Demo.pojo.Student">
    <!--第二种，注入，使用的bean注入方法-->
    <property name="address" ref="address"/>
</bean>
```

第三种：数组注入

```xml
<!--第三种，数组注入-->
<property name="books">
    <array>
        <value>西游记</value>
        <value>水浒传</value>
        <value>三国演义</value>
    </array>
</property>
</bean>
```

第四种：List注入

```xml
<!--第四种，List注入-->
<property name="hobbys">
    <list>
        <value>写代码</value>
        <value>听音乐</value>
        <value>看电影</value>
    </list>
</property>
</bean>
```

第五种：Map注入

```
    <!--第五种，Map注入-->
    <property name="card">
        <map>
            <entry key="身份证" value="610121199801080475"/>
            <entry key="银行卡" value="6189754582X"/>
        </map>
    </property>
</bean>
```

第六种：Set注入（重点）

```
        <!--第六种，Set注入-->
        <property name="games">
            <set>
                <value>魔兽世界</value>
                <value>巫师3：狂猎</value>
                <value>王者荣耀</value>
            </set>
        </property>
    </bean>
```

第七种：Null注入（空值注入）

空值注入：

<propertyname="info"value=""/>

Null注入：

```
        <!--第七种，null注入-->
        <property name="wife">
            <null/>
        </property>
```

第八种：Properties注入（其他注入）

```
        <!--第八种，Properties注入-->
        <property name="info">
            <props>
                <prop key="学号">2021</prop>
                <prop key="性别">男性</prop>
                <prop key="年龄">19</prop>
                <prop key="username">Arvin</prop>
            </props>
        </property>
    </bean>
```

输出所有信息：

MyTest.java | Student.java | bean.xml | Address.java

```
package com.Demo.pojo;

public class Address {
    private String address;

    public String getAddress() {
        return address;
    }

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

    @Override
    public String toString() {
        return "Address{" +
                "address='" + address + '\'' +
                '}';
    }
}
```

```
    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", address=" + address.toString() +
                ", books=" + Arrays.toString(books) +
                ", hobbys=" + hobbys +
                ", card=" + card +
                ", games=" + games +
                ", wife='" + wife + '\'' +
                ", info=" + info +
                '}';
    }
}
```

```java
import com.Demo.pojo.Student;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    public static void main(String[] args){
        ApplicationContext context = new ClassPathXmlApplicationContext( configLocation: "bean.xml");
        Student student = (Student) context.getBean( s: "student");
        System.out.println(student.toString());
    }
}
```

```
D:\BX\Java1.8\bin\java.exe ...
Student{name='Arvin', address=com.Demo.pojo.Address@6321e813, books=[西游记, 水浒传, 三国演义], hobbys=[写代码, 听音乐, 看电影], card={身份证=610121199801080475, 银行卡
=6189754582X}, games=[魔兽世界, 巫师3：狂猎, 王者荣耀], wife='null', info={学号=2021, 性别=男性, username=Arvin, 年龄=19}}
```

拓展：

```xml
<!--第二种，address的定义-->
<bean id="address" class="com.Demo.pojo.Address">
    <property name="address" value="西安"/>
</bean>
```

```
D:\BX\Java1.8\bin\java.exe ...
Student{name='Arvin', address=Address{address='西安'}, books=[西游记, 水浒传, 三国演义], hobbys=[写代码, 听音乐, 看电影], card={身份证=610121199801080475,
银行卡=6189754582X}, games=[魔兽世界, 巫师3：狂猎, 王者荣耀], wife='null', info={学号=2021, 性别=男性, username=Arvin, 年龄=19}}
```

- **拓展方式**
  - P命名空间注入：这里的P是property的意思

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

    <bean name="classic" class="com.example.ExampleBean">
        <property name="email" value="someone@somewhere.com"/>
    </bean>

    <bean name="p-namespace" class="com.example.ExampleBean"
        p:email="someone@somewhere.com"/>
</beans>
```

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

    <bean name="john-classic" class="com.example.Person">
        <property name="name" value="John Doe"/>
        <property name="spouse" ref="jane"/>
    </bean>

    <bean name="john-modern"
        class="com.example.Person"
        p:name="John Doe"
        p:spouse-ref="jane"/>

    <bean name="jane" class="com.example.Person">
        <property name="name" value="Jane Doe"/>
    </bean>
</beans>
```

实例演示：

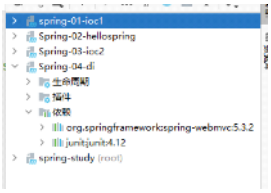
我们需要导入junit包，这个我们经常会用到。我导入到了最外层的pom.xml里，这样我们就不需要不断的导入，调用需要写在该项目的Beans.xml里例如：

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

User.java | userbeans.xml | MyTest.java | pom.xml

```xml
    <modules>
        <module>spring-01-ioc1</module>
        <module>Spring-02-hellospring</module>
        <module>Spring-03-ioc2</module>
        <module>Spring-04-di</module>
    </modules>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.3.2</version>
        </dependency>

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

    </dependencies>
</project>
```

导入后我们如何确认：这里的junit4.12就是我导入的包



User.java userbeans.xml

```java
package com.Demo.pojo;

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

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

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

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

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

    <!--P命名空间注入，可以直接注入属性的值（简单的东西可以采用这个）-->
    <bean id="user" class="com.Demo.pojo.User" p:name="Syc" p:age="18"/>

</beans>
```

User.java userbeans.xml MyTest.java pom.xml (spring-study)

```java
import com.Demo.pojo.Student;
import com.Demo.pojo.User;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    public static void main(String[] args){...}

    @Test
    public void test2(){
        ApplicationContext  context= new ClassPathXmlApplicationContext( configLocation: "userbeans.xml");
        User user = context.getBean( s: "user",User.class);
        System.out.println(user);
    }
}
```

使用这个就可以不用去强制转换类型

```
D:\BX\Java1.8\bin\java.exe ...
User{name='Syc', age=18}

进程已结束,退出代码0
```

- **C命名空间注入和P命名注入类似：通过构造器加入（有参构造，无参构造是为了避免上面的代码报错）**

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

    <bean id="beanTwo" class="x.y.ThingTwo"/>
    <bean id="beanThree" class="x.y.ThingThree"/>

    <!-- traditional declaration with optional argument names -->
    <bean id="beanOne" class="x.y.ThingOne">
        <constructor-arg name="thingTwo" ref="beanTwo"/>
        <constructor-arg name="thingThree" ref="beanThree"/>
        <constructor-arg name="email" value="something@somewhere.com"/>
    </bean>

    <!-- c-namespace declaration with argument names -->
    <bean id="beanOne" class="x.y.ThingOne" c:thingTwo-ref="beanTwo"
        c:thingThree-ref="beanThree" c:email="something@somewhere.com"/>

</beans>
```

实例：

```java
package com.Demo.pojo;

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

    //User的无参构造
    public User() {
    }

    //User的name，age有参构造
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

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

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

    <!--P命名空间注入，可以直接注入属性的值（简单的东西可以采用这个）-->
    <bean id="user" class="com.Demo.pojo.User" p:name="Syc" p:age="18"/>

    <!--C命名空间注入，通过构造器注入就是：constructs-args-->
    <bean id="user2" class="com.Demo.pojo.User" c:name="sYc" c:age="19"/>

</beans>
```

```java
import com.Demo.pojo.Student;
import com.Demo.pojo.User;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    public static void main(String[] args){...}

    @Test
    public void test2(){
        ApplicationContext  context= new ClassPathXmlApplicationContext( configLocation: "userbeans.xml");
        User user = context.getBean( s: "user2",User.class);
        System.out.println(user);
    }
}
```

修改这里就可以执行了

```
D:\BX\Java1.8\bin\java.exe ...
User{name='sYc', age=19}

进程已结束，退出代码0
```

拓展总结：

我们可以使用P命名空间和C命名空间进行注入，官方解释：

XML Shortcut with the p-namespace

The p-namespace lets you use the element's attributes (instead of nested elements) to describe your property values collaborating beans, or both. `bean` `<property/>`

Spring supports extensible configuration formats with namespaces, which are based on an XML Schema definition. The configuration format discussed in this chapter is defined in an XML Schema document. However, the p-namespace is not defined in an XSD file and exists only in the core of Spring. `beans`

The following example shows two XML snippets (the first uses standard XML format and the second uses the p-namespace) that resolve to the same result:

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

    <bean name="classic" class="com.example.ExampleBean">
        <property name="email" value="someone@somewhere.com"/>
    </bean>

    <bean name="p-namespace" class="com.example.ExampleBean"
        p:email="someone@somewhere.com"/>
</beans>
```

XML Shortcut with the c-namespace

Similar to the XML Shortcut with the p-namespace, the c-namespace, introduced in Spring 3.1, allows inlined attributes for configuring the constructor arguments rather then nested elements. `constructor-arg`

The following example uses the namespace to do the same thing as the from Constructor-based Dependency Injection: `c:`

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

    <bean id="beanTwo" class="x.y.ThingTwo"/>
    <bean id="beanThree" class="x.y.ThingThree"/>

    <!-- traditional declaration with optional argument names -->
    <bean id="beanOne" class="x.y.ThingOne">
        <constructor-arg name="thingTwo" ref="beanTwo"/>
        <constructor-arg name="thingThree" ref="beanThree"/>
        <constructor-arg name="email" value="something@somewhere.com"/>
    </bean>

    <!-- c-namespace declaration with argument names -->
    <bean id="beanOne" class="x.y.ThingOne" c:thingTwo-ref="beanTwo"
        c:thingThree-ref="beanThree" c:email="something@somewhere.com"/>

</beans>
```

注意点：P命名和C命名空间不能直接使用，需要导入XML约束：

P命名空间：

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

C命名空间：

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

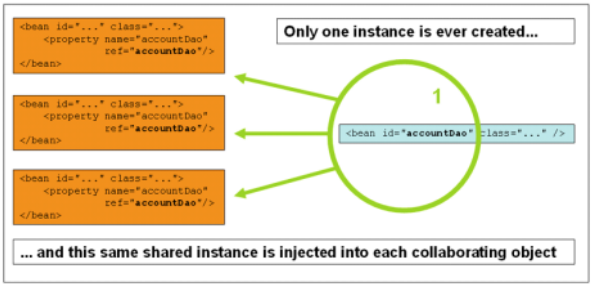
7.Bean的作用域

包含一下六种作用域：

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 . Only valid in the context of a web-aware Spring . `Session` `ApplicationContext`
application	Scopes a single bean definition to the lifecycle of a . Only valid in the context of a web-aware Spring . `ServletContext` `ApplicationContext`
websocket	Scopes a single bean definition to the lifecycle of a . Only valid in the context of a web-aware Spring . `WebSocket` `ApplicationContext`

The Singleton Scope (单例模式)



例如：

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

    <!--P命名空间注入，可以直接注入属性的值（简单的东西可以采用这个）-->
    <bean id="user" class="com.Demo.pojo.User" p:name="Syc" p:age="18"/>

    <!--C命名空间注入，通过构造器注入就是：constructs-args-->
    <bean id="user2" class="com.Demo.pojo.User" c:name="sYc" c:age="19"/>

</beans>
```

从配置文件看user只在P命名空间注入了一次。（这里搞错了user2，代码部分自行进行修改）

```java
import com.Demo.pojo.Student;
import com.Demo.pojo.User;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    public static void main(String[] args){...}

    @Test
    public void test2(){
        ApplicationContext  context= new ClassPathXmlApplicationContext( configLocation: "userbeans.xml");
        User user = context.getBean( s: "user",User.class);
        User user2 = context.getBean( s: "user",User.class);
        System.out.println(user==user2);
    }
}
```

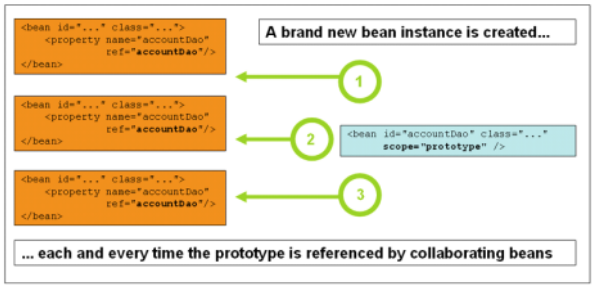
```
D:\BX\Java1.8\bin\java.exe ...
true

进程已结束,退出代码0
```

虽然我们拿出来的对象是多个但是都是被这一个单例出来的，**这是它的默认实现**

```
<!--C命名空间注入，通过构造器注入就是：constructs-args-->
<bean id="user2" class="com.Demo.pojo.User" c:name="sYc" c:age="19" scope="singleton"/>
```

这里我们可以进行设置（singleton就是单例，它默认也是单例模式）

The Prototype Scope（原型模式）



原型模式：每一个Bean它都是一个单独的对象

User.java　userbeans.xml　MyTest.java

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

    <!--P命名空间注入，可以直接注入属性的值（简单的东西可以采用这个）-->
    <bean id="user" class="com.Demo.pojo.User" p:name="Syc" p:age="18"/>

    <!--C命名空间注入，通过构造器注入就是：constructs-args-->
    <bean id="user2" class="com.Demo.pojo.User" c:name="sYc" c:age="19" scope="prototype"/>

</beans>
```

```java
import com.Demo.pojo.Student;
import com.Demo.pojo.User;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    public static void main(String[] args){...}

    @Test
    public void test2(){
        ApplicationContext  context= new ClassPathXmlApplicationContext( configLocation: "userbeans.xml");
        User user = context.getBean( s: "user2",User.class);
        User user2 = context.getBean( s: "user2",User.class);

        System.out.println(user.hashCode());
        System.out.println(user2.hashCode());
        System.out.println(user==user2);
    }
}
```

```
D:\BX\Java1.8\bin\java.exe ...
275310919
2109874862
false

进程已结束,退出代码0
```

总结：

- 单例模式（Spring默认机制）

```
<!--C命名空间注入，通过构造器注入就是：constructs-args-->
<bean id="user2" class="com.Demo.pojo.User" c:name="sYc" c:age="19" scope="singleton"/>
```

- 原型模式：每次从容器中get的时候，都会产生一个新对象。

```
<!--C命名空间注入，通过构造器注入就是：constructs-args-->
<bean id="user2" class="com.Demo.pojo.User" c:name="sYc" c:age="19" scope="prototype"/>
```

- 其余的request、session、application这些只能在web开发中使用到

8.Bean的自动装配

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

在Spring中有三种自动装配的方法

- 在XML中显示配置（我们之前一直使用的就是这个）
- 在Java中显示配置
- 隐式自动装配bean **（重点）**

测试：

- 环境搭建：一个人有两个宠物

```java
package com.Demo.pojo;

public class Cat {
    public void shout(){
        System.out.println("喵喵喵~" );
    }
}
```

```java
package com.Demo.pojo;

public class Dog {
    public void shout(){
        System.out.println("汪汪汪~");
    }
}
```

People.java代码：

```java
packagecom.Demo.pojo;

publicclassPeople{
    privateCatcat;
    privateDogdog;
    privateStringname;

    publicCatgetCat(){
        returncat;
    }

    publicvoidsetCat(Catcat){
        this.cat=cat;
    }

    publicDoggetDog(){
        returndog;
    }
```

```
    public void setDog(Dog dog) {
        this.dog = dog;
    }

    public String getName() {
        return name;
    }

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

    @Override
    public String toString() {
        return "People{" +
        "cat=" + cat +
        ", dog=" + dog +
        ", name='" + name + '\'' +
        '}';
    }
}
```

Dog.java Cat.java People.java beans.xml MyTest.java

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="cat" class="com.Demo.pojo.Cat"/>
    <bean id="dog" class="com.Demo.pojo.Dog"/>

    <bean id="people" class="com.Demo.pojo.People" p:name="张山">
        <property name="dog" ref="dog"/>
        <property name="cat" ref="cat"/>
    </bean>
</beans>
```

Dog.java Cat.java People.java beans.xml MyTest.java

```
import com.Demo.pojo.People;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyTest {
    @Test
    public void test(){
        ApplicationContext context= new ClassPathXmlApplicationContext( configLocation: "beans.xml");

        People people = context.getBean( s: "people", People.class);
        people.getDog().shout();
        people.getCat().shout();
    }
}
```

测试 已通过: 1共 1 个测试 – 284ms

```
284ms
D:\BX\Java1.8\bin\java.exe ...
汪汪汪~
喵喵喵~
```

- ByName自动装配：

Dog.java Cat.java People.java beans.xml MyTest.java

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="cat" class="com.Demo.pojo.Cat"/>
    <bean id="dog" class="com.Demo.pojo.Dog"/>

    <bean id="people" class="com.Demo.pojo.People" p:name="张山" autowire="byName"/>
</beans>
```

ByName：会自动在容器上下文中自动查找和自己对象set方法后面的值对应的bean ID

- ByType自动装配：

ByType：会自动在容器上下文中查找，和自己属性类型相同的bean

总结：

Byname的时候，需要保证所有Bean的ID唯一，并且这个Bean需要和自动注入的属性的Set方法的值一致。

Bytype的时候，需要保证所有Bean的Class唯一，并且这个Bean需要和自动注入属性的类型一致。（Bytype可以不用声明ID进行使用）

使用注解实现自动装配（**重点**）：

jdk1.5支持的注解，Spring2.5就支持注解了。

> **Are annotations better than XML for configuring Spring?**
>
> The introduction of annotation-based configuration raised the question of whether this approach is "better" than XML. The short answer is "it depends." The long answer is that each approach has its pros and cons, and, usually, it is up to the developer to decide which strategy suits them better. Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without touching their source code or recompiling them. Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.
>
> No matter the choice, Spring can accommodate both styles and even mix them together. It is worth pointing out that through its JavaConfig option, Spring lets annotations be used in a non-invasive way, without touching the target components source code and that, in terms of tooling, all configuration styles are supported by the Spring Tools for Eclipse.

说明：

大多数情况下我们会使用注解来进行开发，因为XML是比较麻烦的。

要使用注解我们需要知道：

- 导入约束（context约束）
  - `xmlns:context="http://www.springframework.org/schema/context"`
  - 导入约束的支持

    http://www.springframework.org/schema/context
    https://www.springframework.org/schema/context/spring-context.xsd
- 配置注解的支持（这个是注解驱动的支持）
  - `<context:annotation-config/>`

```
<?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>
```

演示：

```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"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:p="http://www.springframework.org/schema/p"
       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
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean id="cat" class="com.Demo.pojo.Cat"/>
    <bean id="dog11" class="com.Demo.pojo.Dog"/>

    <bean id="people" class="com.Demo.pojo.People" p:name="张山" autowire="byType"/>
</beans>
```

红色框我导入的是context，黄色框我导入的是aop

```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"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:p="http://www.springframework.org/schema/p"
       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
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

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

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

- **@Autowired：**

```java
package com.Demo.pojo;

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

public class People {

    //@Autowired自动配置
    @Autowired
    private Cat cat;
    @Autowired
    private Dog dog;

    private String name;

    public Cat getCat() {
        return cat;
    }

    public void setCat(Cat cat) {
        this.cat = cat;
    }

    public Dog getDog() {
        return dog;
```

转入bean.xml的配置中

```
测试 已通过: 1共 1 个测试 – 373 ms
373 ms  D:\BX\Java1.8\bin\java.exe ...
汪汪汪~
喵喵喵~
```

当然它还可以在set方法使用：

去掉set方法演示：People.java

```
package com.Demo.pojo;

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

public class People {

    //@Autowired自动配置
    @Autowired
    private Cat cat;
    @Autowired
    private Dog dog;

    private String name;

    public Cat getCat() {
        return cat;
    }

    public Dog getDog() {
        return dog;
    }

    public String getName() {
        return name;
    }

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

    @Override
    public String toString() {
        return "People{" +
        "cat=" + cat +
        ", dog=" + dog +
        ", name='" + name + '\'' +
        '}';
    }
}
```

Autowired扩展用法：

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Autowired(required = false)
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

@Nullable 字段标记了这个注解说明这个字段可以为null。

@Nullable 字段实例1：

```java
package com.Demo.pojo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.lang.Nullable;

public class People {

    //@Autowired自动配置
    @Autowired
    private Cat cat;
    @Autowired
    private Dog dog;
    private String name;

    //People的构造器
    public People(@Nullable String name) {
        this.name = name;
    }

    public Cat getCat() {
        return cat;
    }

    public Dog getDog() {
```

加上@Nullable后里面的属性值可以为null

```
D:\BX\Java1.8\bin\java.exe ...
汪汪汪~
喵喵喵~

进程已结束,退出代码0
```

@Nullable 字段实例2：

@Autowired内代码：

```
Public @interface Autowired{
    boolean required() default true;
}
```

```java
package com.Demo.pojo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.lang.Nullable;

public class People {

    //如果显示的定义了Autowired的required属性为false，说明这个对象可以为null，否则不允许为null
    @Autowired(required = false)
    private Cat cat;
    @Autowired
    private Dog dog;
    private String name;

    public Cat getCat() {
        return cat;
    }

    public Dog getDog() {
        return dog;
    }

    public String getName() {
        return name;
```

当然我们也可以和@Qualifier配合使用

```java
package com.Demo.pojo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.lang.Nullable;

public class People {

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

    public Cat getCat() {
        return cat;
    }

    public Dog getDog() {
        return dog;
    }
```

```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"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:p="http://www.springframework.org/schema/p"
       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
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

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

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

如果@Autowired自动装配的环境比较复杂（多属性），自动装配无法通过一个注解（@Autowired）完成的时候，我们可以使用@Qualifier(value = "ID")去配合@Autowired的使用，指定一个唯一的bean对象注入。

@Resource注解

@Resource是Java的原生注解方式，它会通过ID进行判断，当ID不符合时它会判断Class

Dog.java Cat.java People.java beans.xml MyTest.java

```java
package com.Demo.pojo;

import javax.annotation.Resource;

public class People {

    @Resource
    private Cat cat;

    @Resource
    private Dog dog;
    private String name;

    public Cat getCat() {
        return cat;
    }

    public Dog getDog() {
        return dog;
    }

    public String getName() {
        return name;
    }
}
```

测试 已通过: 1共 1 个测试 – 333 ms

```
D:\BX\Java1.8\bin\java.exe ...
汪汪汪~
喵喵喵~
```

Dog.java Cat.java People.java beans.xml MyTest.java

```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"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:p="http://www.springframework.org/schema/p"
       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
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

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

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

beans › bean

测试 已通过: 1共 1 个测试 – 322 ms

```
D:\BX\Java1.8\bin\java.exe ...
汪汪汪~
喵喵喵~
```

当然它也可以像@Qualifier一样进行指配装订

```java
package com.Demo.pojo;

import javax.annotation.Resource;

public class People {

    @Resource(name = "catNoe")
    private Cat cat;

    @Resource
    private Dog dog;
    private String name;

    public Cat getCat() {
        return cat;
    }

    public Dog getDog() {
        return dog;
    }

    public String getName() {
        return name;
    }
```

@Resource和@Autowired的区别：

@Autowired通过Bytype的方式实现，而且必须要求这个对象存在

@Resource默认是通过Byname的方式实现，如果找不到ID则通过Bytype进行实现，如果以上两种都找不到则报错@Resource就像是@Autowired和@Qualifier的集合体。

执行顺序不同：@Resource是先通过ID检测，而@Autowired是先通过Bytype检测

都可以放在属性字段上

都是用来自动装配的

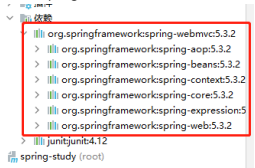
@Autowired总结：

@Autowired：直接在属性上使用即可，也可以在set方式上使用

使用Autowired我们可以不用编写Set方法，前提是自动装配的属性在IOC（Sprinh）容器中存在且符合Byname，因为注解是用反射来实现的。

8.使用注解开发

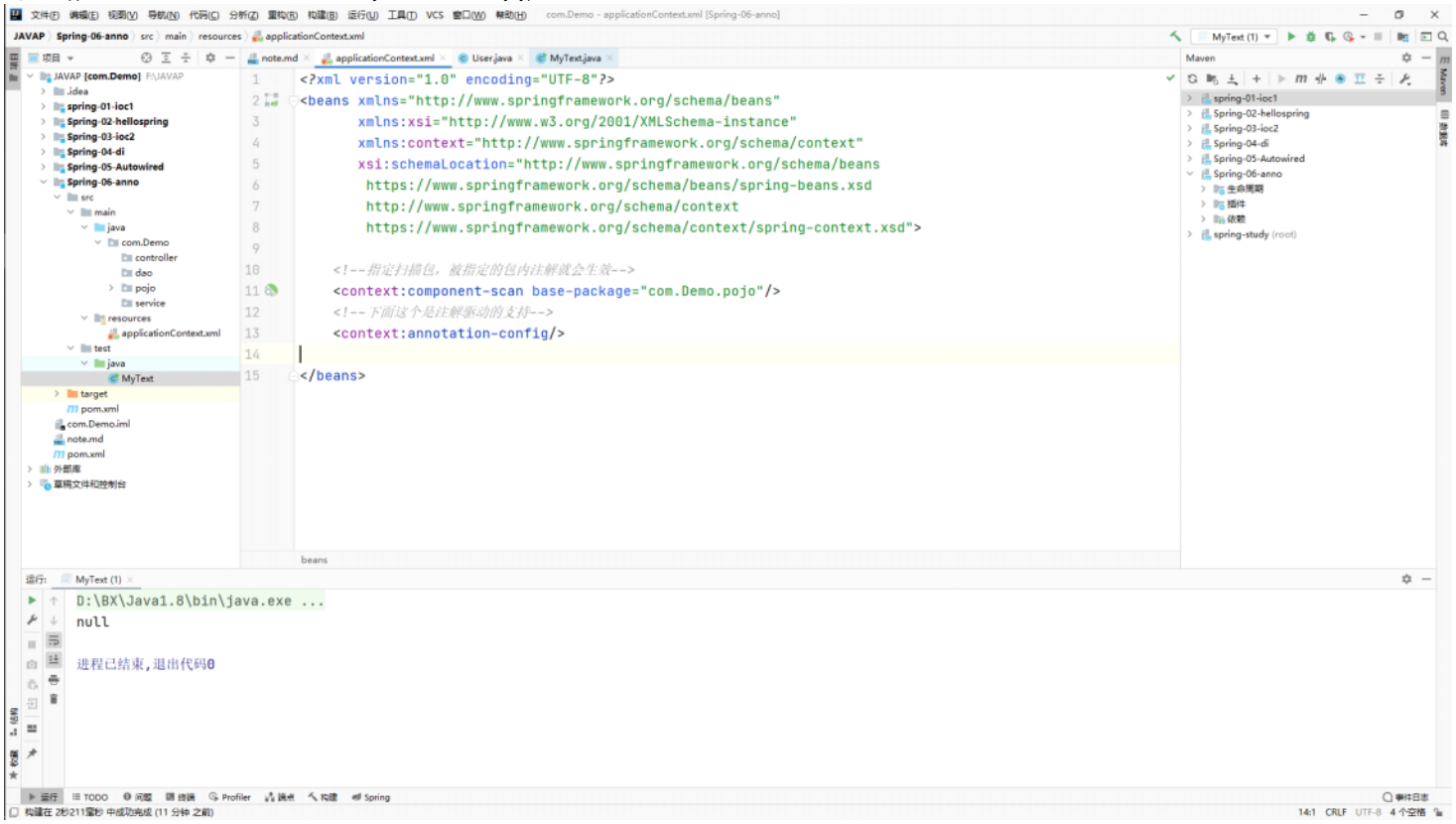
在使用Spring4之后，要使用注解开发，必须要帮助aop的包导入了

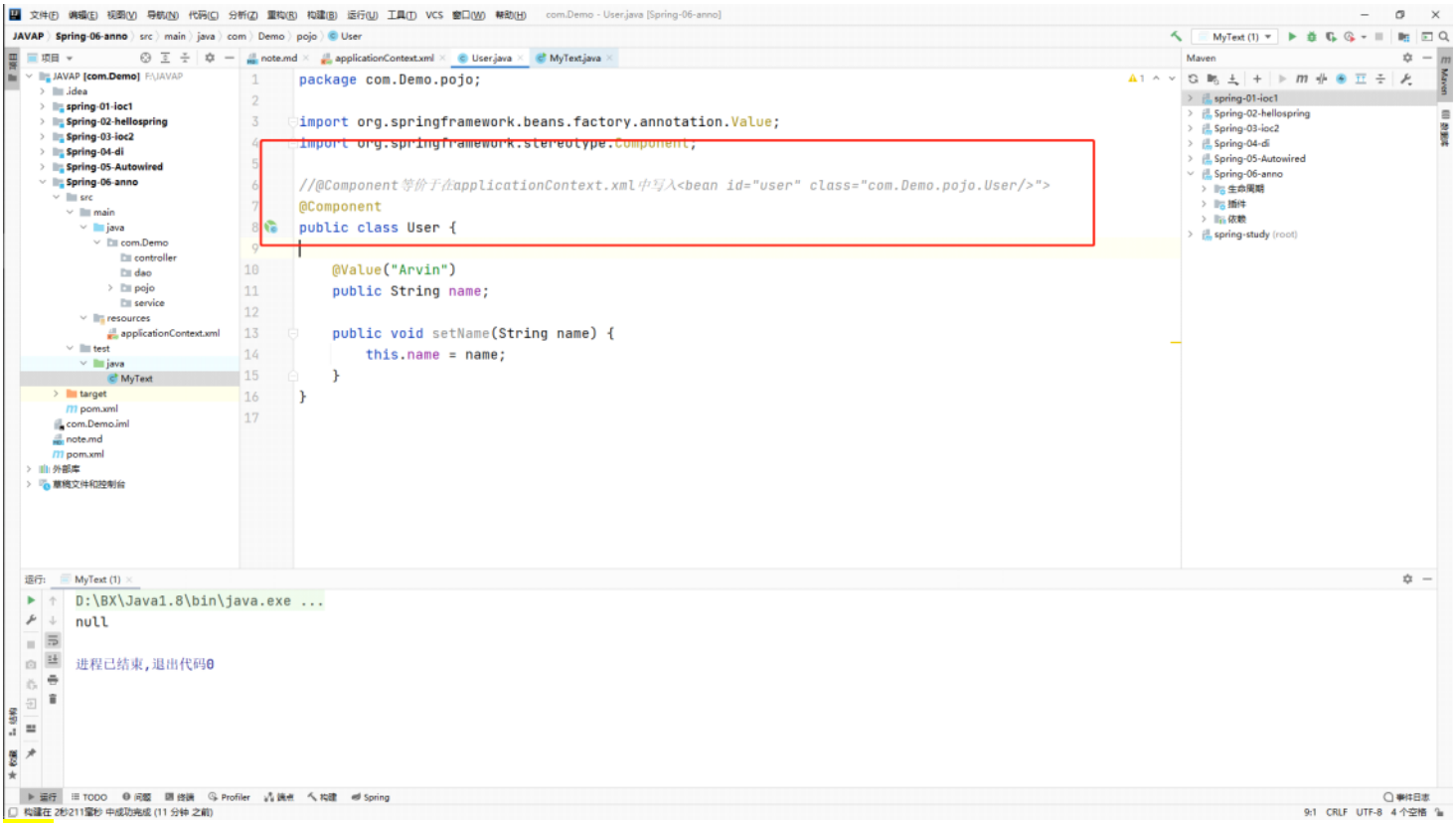


使用注解需要导入context约束，增加注解的支持

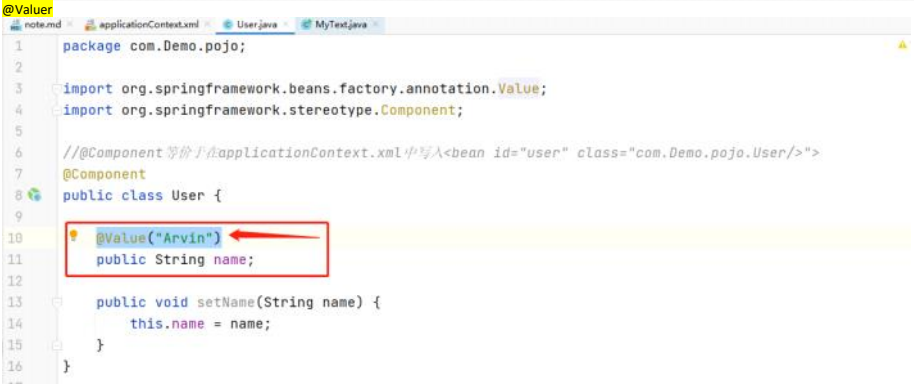
- Bean
- 属性如何注入
  - @Component:

    等价于在applicationContext.xml中写入<bean id = "byname" class = "bytype"/>





- @Valuer



@Value("")，相当于<bean id ="byname" class ="bytype" p:"name" value ="Arvin"/>，它相当于<property name="name" value="Arvin"/>

- 衍生注解
  - @Compoent有几个衍生注解，我们在Web开发中会按MVC三层架构分层
    - dao层：@Repository
    - service层：@Service
    - controller层：@Controller

  这四个注解功能是一样的，都代表将某个类注册到Spring容器中
- 自动装配
  - -@Autowired:自动装配，通过类型(byType)。名字
  - 如果@Autowired不能唯一自动装配上属性，则需要通过@Qualifier(value="ID")
  - -@Nullable:字段标记的注解，说明这个字段可以为null
  - -@Resource:自动装配，通过名字(byName)。类型
- 作用域

  @Scope("singleton")：单例模式

  @Scope("prototype")：原型模式

小结

XML与注解：

- XML更加万能，适用于任何场合，维护也简单方便
- 注解不是自己的类无法进行使用，维护相对复杂

建议：XML用来管理Bean，注解只负责完成属性的注入，我们在使用中只需要注意，必须让注解生效就需要开启注解的支持

```
<!--指定扫描包，被指定的包内注解就会生效-->
<context:component-scanbase-package="com.Demo"/>
<!--下面这个是注解驱动的支持-->
<context:annotation-config/>
```

9.使用Java配置Spring（利用Java原生注解）

我们现在要完全不使用Spring的XML配置，全权交给Java来做

JavaConfig是Spring的子项目，在Spring4之后，它变成了核心功能

实体类

```java
//这个注解的意思就是什么这个类被Spring接管了
@Component
public class User {
    private String name;

    public String getName() {
        return name;
    }

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

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

配置文件（配置类）

```java

import com.Demo.pojo.User;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

//这个会被Spring容器托管，因为它本身就是一个@Component
//@Configuration代表这是一个配置类，就和applicationContext.xml一样
//@ComponentScan功能就是扫描包
@Configuration
@ComponentScan("com.Demo.pojo")
//@Import(类名) 导入配置类
@Import(dConfig2.class)
public class dConfig {

    //注册一个Bean，就相当于我们之前写的一个bean标签
    //这个方法的名字就相当于bean标签的ID属性，方法的返回值就相当于bean标签的Class属性
    @Bean
    public User getUser(){
        return new User();
    }
}
```

note.md dConfig.java User.java MyText.java dConfig2.java

```java
package com.Demo.config;

import org.springframework.context.annotation.Configuration;

@Configuration
public class dConfig2 {

}
```

测试类

note.md dConfig.java User.java MyText.java dConfig2.java

```java
import com.Demo.config.dConfig;
import com.Demo.pojo.User;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MyText {
    public static void main(String[] args) {

        //如果完全使用了配置类方式去做，我们就只能通过AnnotationConfig上下文来获取容器，通过配置类的Class对象加载
        //我们通过注解获取上下文
        ApplicationContext context = new AnnotationConfigApplicationContext(dConfig.class);

        User getUser = (User) context.getBean( s: "getUser");

        System.out.println(getUser.getName());
    }
}
```

这种纯Java的配置方式，在SpringBoot中随处可见。

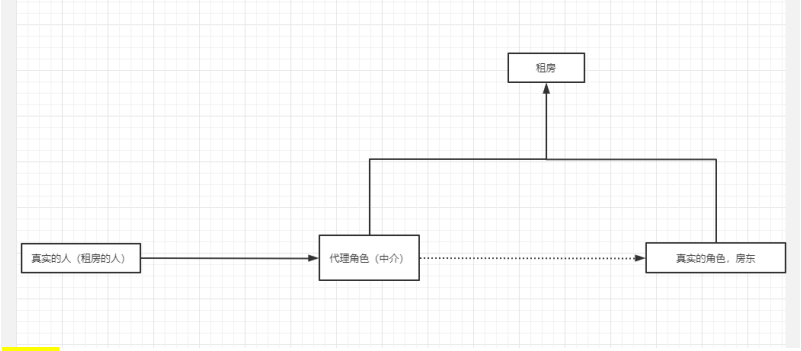
10.代理模式

为什么学习代理模式?

因为这就是SpringAOP的底层（面向切面编程的底层实现）（SpringAOP和SpringMVC是面试必问重点）

代理模式的分类:

- 静态代理
- 动态代理



静态代理

- 抽象的角色：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

代码演示:

接口

Rent.java

```java
package com.Demo.demo01;

//租房的接口
public interface Rent {
    public void rent();

}
```

真实角色

Host.java

```java
package com.Demo.demo01;

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

代理角色

Proxy.java

```java
package com.Demo.demo01;

public class Proxy implements Rent{
    private Host host;

    public Proxy() {
    }

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

    public void rent() {
        host.rent();
        seeHouse();
        hetong();
        fare();
    }

    //看房
    public void seeHouse(){
        System.out.println("中介带你看房");
    }

    //签合同
    public void hetong(){
        System.out.println("签合同");
    }

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

客户端访问代理角色

Rent.java Host.java Client.java Proxy.java

```java
package com.Demo.demo01;

public class Client {
    public static void main(String[] args) {
        //房东要租房子
        Host host = new Host();
        //代理，中介帮房东租房子
        Proxy proxy = new Proxy(host);
        //你直接找中介即可
        proxy.rent();
    }
}
```

代理模式的好处：

- 可以使用真实角色的操作更加存粹，不用关注一些公共的业务
- 公共业务代理角色去做，实现了业务的分公
- 公共业务发生扩展的时候方便集中管理

缺点：一个真实角色就会产生一个代理角色，代码量翻倍，开发效率变低（使用动态代理的反射来解决这个问题）

对于业务的分工深层刨析：

UserService.java UserServiceImpl.java Client.java UserServiceProxy.java

```java
package com.Demo.demo02;

public interface UserService {
    public void add();
    public void delete();
    public void update();
    public void query();
}
```

UserService.java UserServiceImpl.java Client.java UserServiceProxy.java

```java
package com.Demo.demo02;

//真实对象
public class UserServiceImpl  implements UserService{
    public void add() {
        System.out.println("增加了一个用户");
    }

    public void delete() {
        System.out.println("删除了一个用户");
    }

    public void update() {
        System.out.println("改正了一个用户");
    }

    public void query() {
        System.out.println("查找了一个用户");
    }
}
```

UserService.java UserServiceImpl.java Client.java UserServiceProxy.java

```java
package com.Demo.demo02;

public class UserServiceProxy implements UserService{
    private UserServiceImpl userService;

    public void setUserService(UserServiceImpl userService) {
        this.userService = userService;
    }

    public void add() {
        log( msg: "add");
        userService.add();
    }

    public void delete() {
        log( msg: "delete");
        userService.delete();
    }

    public void update() {
        log( msg: "update");
        userService.update();
    }

    public void query() {
        log( msg: "query");
        userService.query();
    }

    //日志方法
    public void log(String msg){
        System.out.println("[DeBug] 使用了"+msg+"方法");
    }
}
```

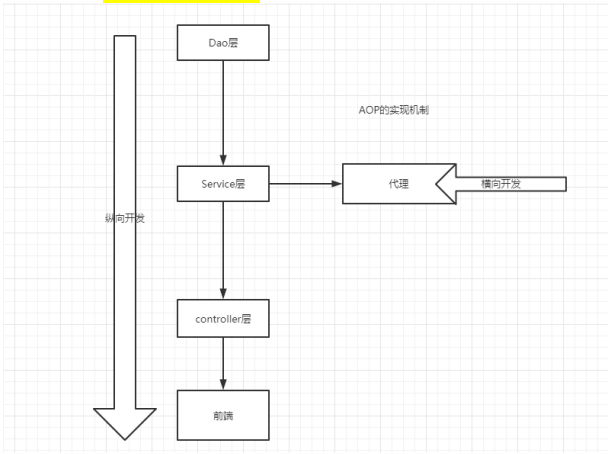
UserService.java × UserServiceImpl.java × Client.java × UserServiceProxy.java ×

```java
package com.Demo.demo02;

public class Client {
    public static void main(String[] args) {
        UserServiceImpl userService = new UserServiceImpl();
        UserServiceProxy proxy = new UserServiceProxy();
        proxy.setUserService(userService);
        proxy.add();
    }
}
```

为什么使用代理：改动原有的业务代码，在公司中是大忌。

关于AOP

实现代码参考：**对于业务的分工深层剖析**



动态代理

- 动态代理和静态代理角色一样
- 动态代理的代理类类是动态生成的，不是我们直接写好的（我们创建一个模板）
- 动态代理分为两大类：
  - 基于接口的动态代理
    - JDK的动态代理最经典（原生的）[我们使用这个]
  - 基于类的动态代理
    - Cglib
  - 现在使用最多的是Java字节码实现
    - Javassist

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

InvocationHandler

InvocationHandler是由代理实例的调用处理程序实现的接口，是反射包（reflect）下的。每个代理实例都有一个关联的调用处理程序，当在代理实例上调用方法时，方法调用将被编码并分配到其调用处理程序的invoke方法（通过反射去执行一个方法）。（自动生成调用处理程序实现的接口）

方法：

- **Proxy：调用该方法的代理实例（你要代理谁？）**
- method：所述方法对应于调用代理实例上的接口方法实例，方法对象的声明类将是该方法声明的接口，它可以是代理类继承该方法的代理接口的超级接口。（你要代理哪一个方法？）
- args：包含的方法调用传递代理实例的参数值的对象阵列，或null如果接口方法没有参数。原始类型的参数包含在适当的原始包装器类的实例中，例如Java.lang.Integer或Java.lang.boolean。（你要给方法传什么参数？）
- 结果：返回它自己生成的一个实例类，也就是我们的代理类

Proxy

proxy提供了创建动态代理类和实例的静态方法，它也是由这些方法创建的所有动态代理类的超类

为某个接口创建代理Foo：

```
InvocationHandler handler = new MyInvocationHandler(...);
   Class<?> proxyClass = Proxy.getProxyClass(Foo.class.getClassLoader(), Foo.class);
   Foo f = (Foo) proxyClass.getConstructor(InvocationHandler.class).
                 newInstance(handler);
```

或更简单地：

```
Foo f = (Foo) Proxy.newProxyInstance(Foo.class.getClassLoader(),
                                     new Class<?>[] { Foo.class },
                                     handler);
```

Modifier and Type	Method and Description
static InvocationHandler	getInvocationHandler(Object proxy) 返回指定代理实例的调用处理程序。
static 类<?>	getProxyClass(ClassLoader loader, 类<?>... interfaces) 给出类加载器和接口数组的代理类的 java.lang.Class对象。
static boolean	isProxyClass(类<?> cl) 如果且仅当使用 getProxyClass方法或 newProxyInstance方法将指定的类动态生成为代理类时，则返回true。
static Object	newProxyInstance(ClassLoader loader, 类<?>[] interfaces, InvocationHandler h) 返回指定接口的代理类的实例，该接口将方法调用分派给指定的调用处理程序。

```
public static Object newProxyInstance(ClassLoader loader,
                                       类<?>[] interfaces,
                                       InvocationHandler h)
                                throws IllegalArgumentException
```

返回指定接口的代理类的实例，该接口将方法调用分派给指定的调用处理程序。

`Proxy.newProxyInstance`因为与`IllegalArgumentException`相同的原因而`Proxy.getProxyClass`。

参数

`loader` - 类加载器来定义代理类

`interfaces` - 代理类实现的接口列表

`h` - 调度方法调用的调用处理函数

结果

具有由指定的类加载器定义并实现指定接口的代理类的指定调用处理程序的代理实例

异常

`IllegalArgumentException` - 如果对可能传递给 `getProxyClass`有任何 `getProxyClass`被违反

`SecurityException` -如果安全管理器，`s`存在任何下列条件得到满足：

- 给定的`loader`是`null` ，并且调用者的类加载器不是`null` ，并且调用`s.checkPermission`与`RuntimePermission("getClassLoader")`权限拒绝访问;
- 对于每个代理接口， `intf` ，呼叫者的类加载器是不一样的或类加载器的祖先`intf`和调用`s.checkPackageAccess()`拒绝访问`intf` ;
- 任何给定的代理接口的是非公和呼叫者类是不在同一`runtime package`作为非公共接口和调用`s.checkPermission`与`ReflectPermission("newProxyInPackage.{package name}")`权限拒绝访问。

`NullPointerException` - 如果 `interfaces`数组参数或其任何元素是 `null` ，或者如果调用处理程序 `h`是 `null`

动态代理演示：

client.java Host.java Rent.java ProxyInvocationHandler.java

```java
package com.Demo.demo03;

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

client.java Host.java Rent.java ProxyInvocationHandler.java

```java
package com.Demo.demo03;

//租房的接口
public interface Rent {
    public void rent();

}
```

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

//我们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;
    }

    //生成得到代理类
    public Object getProxy(){
        return Proxy.newProxyInstance(this.getClass().getClassLoader(),rent.getClass().getInterfaces(), h: this);
    }

    //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使用反射机制实现
        seeHouse();
        Object result = method.invoke(rent, args);
        fare();
        return result;
    }
    public void seeHouse(){
        System.out.println("中介带看房子");
    }
    public void fare(){
        System.out.println("收中介费");
    }

}
```

client.java Host.java Rent.java ProxyInvocationHandler.java

```java
package com.Demo.demo03;

public class client {
    public static void main(String[] args) {
        //真实角色
        Host host = new Host();

        //代理角色：暂时没有
        ProxyInvocationHandler pih = new ProxyInvocationHandler();
        //通过调用程序处理角色来处理我们要用的接口对象
        pih.setRent(host);

        //这里的proxy就是动态生成的，我们并没有写
        Rent proxy = (Rent) pih.getProxy();

        proxy.rent();
    }
}
```