

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Iteface21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。
- SHM: 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>

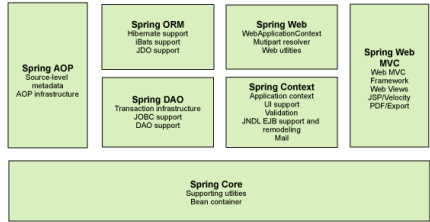
```
Spring Maven:
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.3.2</version>
</dependency>

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。SpringIbbean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，所有这些都被通过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。模型类有JavaBean来构成，存放于w当中，而视图是一个接口，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懈到特定J2EE服务的可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

```

1 package com.Demo.dao;
2
3 public interface UserDao {
4     void getUser();
5 }
6
2.UserDaoImpl 实现类
1 package com.Demo.dao;
2
3 public class UserDaoImpl implements UserDao{
4     public void getUser(){
5         System.out.println("默认获取用户的数据");
6     }
7 }
3.UserService 业务接口
1 package com.Demo.service;
2
3 public interface UserService {
4     void getUser();
5 }
4.UserServiceImpl 业务实现类
1 package com.Demo.service;
2
3 import com.Demo.dao.UserDao;
4 import com.Demo.dao.UserDaoImpl;
5 import com.Demo.dao.UserDaoMysqlImpl;
6 import com.Demo.dao.UserOracleImpl;
7
8 public class UserServiceImpl implements UserService{
9     private UserDao userDao = new UserOracleImpl();
10
11     public void getUser() {
12         userDao.getUser();
13     }
14 }
5.Main函数
1 import com.Demo.dao.UserDaoImpl;
2 import com.Demo.service.UserService;
3 import com.Demo.service.UserServiceImpl;
4
5 public class MyText {
6     public static void main(String[] args){
7
8         // 用户实际调用的是业务层, Dao层他们不需要接触
9         UserService userService = new UserServiceImpl();
10
11         userService.getUser();
12     }
13 }

```

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, 之前程序是主动的创建对象, 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, 程序员不再需要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的原理)

```

1 package com.Demo.service;
2
3 import com.Demo.dao.UserDao;
4 import com.Demo.dao.UserDaoImpl;
5 import com.Demo.dao.UserDaoMysqlImpl;
6 import com.Demo.dao.UserOracleImpl;
7
8 public class UserServiceImpl implements UserService{
9     private UserDao userDao;
10
11     // 通过set方法设置依赖对象
12     public void setUserDao(UserDao userDao) { this.userDao = userDao; }
13
14     public void getUser() { userDao.getUser(); }
15 }

```

```

1 import com.Demo.dao.UserDaoImpl;
2 import com.Demo.dao.UserDaoMysqlImpl;
3 import com.Demo.dao.UserOracleImpl;
4 import com.Demo.service.UserService;
5 import com.Demo.service.UserServiceImpl;
6
7 public class MyText {
8     public static void main(String[] args){
9
10         // 用户实际调用的是业务层, Dao层他们不需要接触
11         UserService userService = new UserServiceImpl();
12         ((UserServiceImpl) userService).setUserDao(new UserOracleImpl());
13         userService.getUser();
14     }
15 }

```

4.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容器在初始化时更先读取配置文件, 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

```

1 package com.Deom.pojo;
2
3 public class Hello {
4     private String name;
5
6     public String getNaem(){
7         return name;
8     }
9
10    public void setName(String name){
11        this.name = name;
12    }
13
14    public void show(){
15        System.out.println("Hello"+name);
16    }
17
18 }
19

```

```

1 package com.Deom.pojo;
2
3 public class Hello {
4     private String str;
5
6     public String getStr() {
7         return str;
8     }
9
10    public void setStr(String str) {
11        this.str = str;
12    }
13
14    @Override
15    public String toString() {
16        return "Hello{" +
17            "str='" + str + '\'' +
18        }';
19    }
20 }
21

```

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://www.springframework.org/schema/beans
5         http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <!-- 使用Spring来创建我们的对象，在Spring这些都称为Bean -->
8     <bean id="hello" class="com.Deom.pojo.Hello">
9         <property name="str" value="Spring"/>
10    </bean>
11
12 </beans>

```

```

1 import com.Deom.pojo.Hello;
2 import org.springframework.context.ApplicationContext;
3 import org.springframework.context.support.ClassPathXmlApplicationContext;
4
5 public class MyTest {
6     public static void main(String[] args){
7         // 获取Spring的上下文对象
8         ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
9         // 我们的对象现在都在Spring中管理了，我们便使用，直接去里面取出来就可以了
10        Hello hello = (Hello) context.getBean("hello");
11        System.out.println(hello.toString());
12    }
13 }
14

```

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();

```
<bean id="hello" class="com.Deom.pojo.Hello">
```

```
<property name="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

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="http://www.springframework.org/schema/beans
5       http://www.springframework.org/schema/beans/spring-beans.xml">
6
7     <bean id="MySQLImpl" class="com.Demo.dao.UserDaoMySQLImpl"/>
8     <bean id="OracleImpl" class="com.Demo.dao.UserDaoOracleImpl"/>
9     <bean id="UserServiceImpl" class="com.Demo.service.UserServiceImpl">
10         <!--
11         ref: 在Spring容器中, 通过id来查找对象
12         value: 从容器中, 取出对象实例
13         -->
14         <property name="userDao" ref="OracleImpl"/>
15     </bean>
16
17 </beans>

```

```

1 import com.Demo.dao.UserDao;
2 import com.Demo.dao.UserDaoMySQLImpl;
3 import com.Demo.dao.UserDaoOracleImpl;
4 import com.Demo.dao.UserDaoSqlserverImpl;
5 import com.Demo.service.UserService;
6 import com.Demo.service.UserServiceImpl;
7 import org.springframework.context.ApplicationContext;
8 import org.springframework.context.support.ClassPathXmlApplicationContext;
9
10 public class MyText {
11     public static void main(String[] args) {
12         // 获取ApplicationContext, 拿到Spring容器
13         ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
14         // 需要什么直接get
15         UserServiceImpl userServiceImpl = (UserServiceImpl) context.getBean("UserServiceImpl");
16         userServiceImpl.getUser();
17
18         // 用户实际调用的是业务层, Dao层他们不需要接触
19         /* UserService userService = new UserServiceImpl();
20         ((UserServiceImpl) userService).setUserDao(new UserDaoSqlserverImpl());
21         userService.getUser();
22         我们不再需要new 一个对象
23         */
24     }
25 }

```

演示

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="http://www.springframework.org/schema/beans
5       http://www.springframework.org/schema/beans/spring-beans.xml">
6
7     <bean id="MySQLImpl" class="com.Demo.dao.UserDaoMySQLImpl"/>
8     <bean id="OracleImpl" class="com.Demo.dao.UserDaoOracleImpl"/>
9     <bean id="SqlserverImpl" class="com.Demo.dao.UserDaoSqlserverImpl"/>
10     <bean id="UserServiceImpl" class="com.Demo.service.UserServiceImpl">
11         <!--
12         ref: 在Spring容器中, 通过id来查找对象
13         value: 从容器中, 取出对象实例
14         -->
15         <property name="userDao" ref="SqlserverImpl"/>
16     </bean>
17
18 </beans>

```

```

1 import com.Demo.dao.UserDaoImpl;
2 import com.Demo.dao.UserDaoMySQLImpl;
3 import com.Demo.dao.UserDaoSqlserverImpl;
4 import com.Demo.dao.UserOracleImpl;
5 import com.Demo.service.UserService;
6 import com.Demo.service.UserServiceImpl;
7 import org.springframework.context.ApplicationContext;
8 import org.springframework.context.support.ClassPathXmlApplicationContext;
9
10 public class MyText {
11     public static void main(String[] args){
12         // 获取ApplicationContext, 拿到Spring容器
13         ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
14         // 需要什么直接get
15         UserServiceImpl userServiceImpl = (UserServiceImpl) context.getBean("UserServiceImpl");
16         userServiceImpl.getUser();
17
18         // 用户实际调用的是业务层, Dao层他们不需要接触
19         /* UserService userService = new UserServiceImpl();
20         ((UserServiceImpl) userService).setUserDao(new UserDaoSqlserverImpl());
21         userService.getUser();
22         我们不再需要new 一个对象
23         */
24     }
25 }

```

5.IOC创建对象的方式

普通:

```

1 package com.Demo.pojo;
2
3 public class User {
4     private String name;
5
6     public User(){
7         System.out.println("User的无参构造");
8     }
9
10    public String getName() {
11        return name;
12    }
13
14    public void setName(String name) {
15        this.name = name;
16    }
17
18    public void show(){
19        System.out.println("name="+name);
20    }
21 }
22

```

```

1 import com.Demo.pojo.User;
2
3 public class MyTest {
4     public static void main(String[] args){
5         User user = new User();
6     }
7 }
8

```

- 使用无参构造创建对象，默认！
- 假设我们要使用有参构造对象。

- 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:

```

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

```

```

User.java  MyTest.java  beans.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://www.springframework.org/schema/beans
5         https://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <!-- 第一种、下标赋值 -->
8     <bean id="user" class="com.Demo.pojo.User">
9         <constructor-arg index="0" value="SongYuChen"/>
10    </bean>
11
12
13 </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:

```

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

```

```

User.java  MyTest.java  beans.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://www.springframework.org/schema/beans
5         https://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <!-- 第一种、下标赋值 -->
8     <!--<bean id="user" class="com.Demo.pojo.User">
9         <constructor-arg index="0" value="SongYuChen"/>
10    </bean>-->
11
12     <!-- 第二种、通过类型创建 不推荐 -->
13     <bean id="user" class="com.Demo.pojo.User">
14         <constructor-arg type="java.lang.String" value="SongyuChen"/>
15     </bean>
16
17 </beans>

```

- 第三种：通过参数名创建

```

<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>
XML

```

```

User.java  MyTest.java  beans.xml
5 https://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <!-- 第一种、下标赋值 -->
8     <!--<bean id="user" class="com.Demo.pojo.User">
9         <constructor-arg index="0" value="SongYuChen"/>
10    </bean>-->
11
12     <!-- 第二种、通过类型创建 不推荐 -->
13     <!--<bean id="user" class="com.Demo.pojo.User">
14         <constructor-arg type="java.lang.String" value="SongyuChen"/>
15     </bean>-->
16
17     <!-- 第三种：直接通过参数名称来设置 -->
18     <bean id="user" class="com.Demo.pojo.User">
19         <constructor-arg name="name" value="songYuChen"/>
20     </bean>
21
22 </beans>

```

- 我们再进行无参有参共存：Spring就类似于婚恋网站

```

1 package com.Demo.pojo;
2
3 public class UserT {
4     private String name;
5
6     public UserT(){
7         System.out.println("UserT被创建了");
8     }
9
10    public String getName() {
11        return name;
12    }
13
14    public void setName(String name) {
15        this.name = name;
16    }
17
18    public void show(){
19        System.out.println("name="+name);
20    }
21 }
22
23

```

```

6
7 <!-- 第一种、下标赋值-->
8 <!--<bean id="user" class="com.Demo.pojo.User">
9     <constructor-arg index="0" value="SongYuChen"/>
10 </bean-->
11
12 <!-- 第二种、通过类型创建 不推荐-->
13 <!--<bean id="user" class="com.Demo.pojo.User">
14     <constructor-arg type="java.lang.String" value="Songyu"
15 </bean-->
16
17 <!-- 第三种、直接通过参数名称来设置-->
18 <bean id="user" class="com.Demo.pojo.User">
19     <constructor-arg name="name" value="songYuChen"/>
20 </bean>
21 <bean id="userT" class="com.Demo.pojo.UserT"/>
22
23 </beans>

```

```

MyTest (1) x
D:\BX\Java1.8\bin\java.exe ...
UserT被创建了
name=songYuChen
进程已结束,退出代码0

```

```

1 import com.Demo.pojo.User;
2 import org.springframework.context.ApplicationContext;
3 import org.springframework.context.support.ClassPathXmlApplicationContext;
4
5 public class MyTest {
6     public static void main(String[] args){
7         //User user = new User();
8
9         //Spring 容器
10        ApplicationContext context = new ClassPathXmlApplicationContext( configLocation: "beans.xml");
11
12        //两个User是同一个,说明其内存中只要一份实例。
13        User user = (User) context.getBean( s: "user");
14        User user2 =(User) context.getBean( s: "user");
15        System.out.println(user == user2);
16        user.show();
17    }
18 }
19

```

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"/>


```

9      <constructor-arg index="0" value="SongYuChen"/>
10    </bean>-->
11
12    <!-- 第二种、通过类型创建 不推荐-->
13    <!--<bean id="user" class="com.Demo.pojo.User">
14      <constructor-arg type="java.lang.String" value="SongyuChen"/>
15    </bean>-->
16
17    <!-- 第三种、直接通过参数名称来设置-->
18    <bean id="user" class="com.Demo.pojo.User">
19      <constructor-arg name="name" value="songYuChen"/>
20    </bean>
21    <bean id="userT" class="com.Demo.pojo.UserT"/>
22
23    <!-- 别名-->
24    <alias name="user" alias="Syc"/>
25  </beans>

```

```

1  import com.Demo.pojo.User;
2  import org.springframework.context.ApplicationContext;
3  import org.springframework.context.support.ClassPathXmlApplicationContext;
4
5  public class MyTest {
6    public static void main(String[] args){
7      //User user = new User();
8
9      //Spring 容器
10     ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
11
12     //两个User是同一个，说明其内存中只要一份实例。
13     User user = (User) context.getBean("Syc");
14     /*User user2 = (User) context.getBean("user");
15     System.out.println(user == user2);*/
16     user.show();
17   }
18 }

```

- 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

```

11    <!-- 第二种、通过类型创建 不推荐-->
12    <!--<bean id="user" class="com.Demo.pojo.User">
13      <constructor-arg type="java.lang.String" value="SongyuChen"/>
14    </bean>-->
15
16    <!-- 第三种、直接通过参数名称来设置-->
17    <bean id="user" class="com.Demo.pojo.User">
18      <constructor-arg name="name" value="songYuChen"/>
19    </bean>
20
21    <!-- Bean-->
22    <bean id="userT" class="com.Demo.pojo.UserT" name="user2"/>
23
24    <!-- 别名-->
25    <alias name="user" alias="Syc"/>
26  </beans>

```

```

1  import com.Demo.pojo.User;
2  import org.springframework.context.ApplicationContext;
3  import org.springframework.context.support.ClassPathXmlApplicationContext;
4
5  public class MyTest {
6    public static void main(String[] args){
7      //User user = new User();
8
9      //Spring 容器
10     ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
11
12     //两个User是同一个，说明其内存中只要一份实例。
13     User user = (User) context.getBean("Syc");
14     /*User user2 = (User) context.getBean("user");
15     System.out.println(user == user2);*/
16     user.show();
17   }
18 }

```

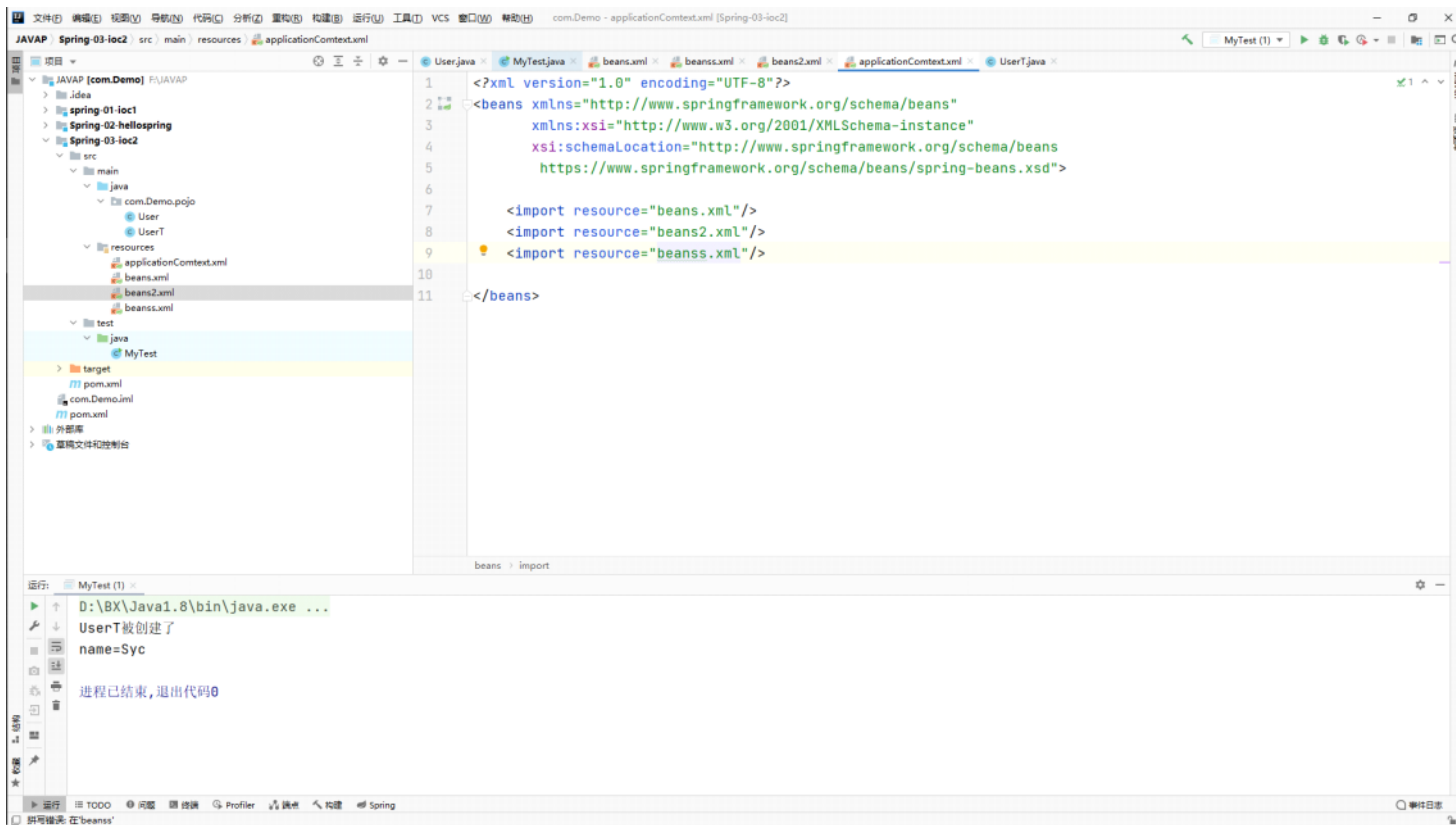
多个别名：可以使用（空格，逗号，分隔符）

```

7    <!-- 第一种、下标赋值-->
8    <!--<bean id="user" class="com.Demo.pojo.User">
9      <constructor-arg index="0" value="SongYuChen"/>
10    </bean>-->
11
12    <!-- 第二种、通过类型创建 不推荐-->
13    <!--<bean id="user" class="com.Demo.pojo.User">
14      <constructor-arg type="java.lang.String" value="SongyuChen"/>
15    </bean>-->
16
17    <!-- 第三种、直接通过参数名称来设置-->
18    <bean id="user" class="com.Demo.pojo.User">
19      <constructor-arg name="name" value="songYuChen"/>
20    </bean>
21
22    <!-- Bean-->
23    <bean id="userT" class="com.Demo.pojo.UserT" name="user2 u2,u3,u4">
24      <property name="name" value="Syc"/>
25    </bean>
26
27    <!-- 别名-->
28    <alias name="user" alias="Syc"/>
29

```

- Import：一般用于团队开发使用，它可以多个配置文件，导入合并为一个
 - 假设：现在项目中有多个开发，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:

```
package com.Demo.pojo;
import java.util.*;

public class Student {
    private String name;
    private Address address;
    private String[] books;
    private List<String> hobbies;
    private Map<String, String> card;
    private Set<String> games;
    private String wife;
    private Properties info;

    public String getName(){
        return name;
    }

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

    public Address getAddress(){
        return address;
    }

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

    public String[] getBooks(){
        return books;
    }

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

    public List<String> getHobbies(){
        return hobbies;
    }

    public void setHobbies(List<String> hobbies){
        this.hobbies = hobbies;
    }

    public Map<String, String> getCard(){
        return card;
    }

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

    public Set<String> getGames(){
        return games;
    }

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

    public String getWife(){
```



```

        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-d)  bean.xml  MyTest.java  Student.java  Address.java
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://www.springframework.org/schema/beans
5          http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7      <bean id="student" class="com.Demo.pojo.Student">
8          <property name="name" value="Arvin"/>
9      </bean>
10
11 </beans>

```

```

bean.xml  MyTest.java  Student.java  Address.java
1  import com.Demo.pojo.Student;
2  import org.springframework.context.ApplicationContext;
3  import org.springframework.context.support.ClassPathXmlApplicationContext;
4
5  public class MyTest {
6      public static void main(String[] args){
7          ApplicationContext context = new ClassPathXmlApplicationContext( configLocation: "bean.xml");
8          Student student = (Student) context.getBean( s: "student");
9          System.out.println(student.getName());
10     }
11 }
12

```

第一种：其他代码在上面（一下基于上面的环境）

```

6
7  <bean id="student" class="com.Demo.pojo.Student">
8      <!-- 第一种，普通值注入，Value-->
9      <property name="name" value="Arvin"/>
10 </bean>
11

```

第二种：Bean注入

```

<!-- 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

```

<!-- 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

```

<!-- 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:
<property name="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:

```
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) +
        ", hobbies=" + hobbies +
        ", card=" + card +
        ", games=" + games +
        ", wife='" + wife + '\'' +
        ", info=" + info +
        '}';
}
```

```

1 import com.Demo.pojo.Student;
2 import org.springframework.context.ApplicationContext;
3 import org.springframework.context.support.ClassPathXmlApplicationContext;
4
5 public class MyTest {
6     public static void main(String[] args){
7         ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
8         Student student = (Student) context.getBean("student");
9         System.out.println(student.toString());
10    }
11 }
12

```

```

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], hobbies=[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:

```

<!-- 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], hobbies=[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

- o P命名空间注入: 这里的P是property的意思

```

<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 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>

```

```

<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 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:

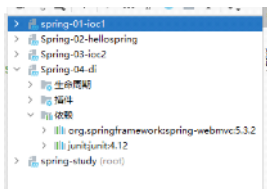
xmlns:p="http://www.springframework.org/schema/p"

```

User.java  userbeans.xml  MyTest.java  pom.xml
11 <modules>
12     <module>spring-01-ioc1</module>
13     <module>Spring-02-hellospring</module>
14     <module>Spring-03-ioc2</module>
15     <module>Spring-04-di</module>
16 </modules>
17
18 <properties>
19     <maven.compiler.source>8</maven.compiler.source>
20     <maven.compiler.target>8</maven.compiler.target>
21 </properties>
22
23 <dependencies>
24     <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
25     <dependency>
26         <groupId>org.springframework</groupId>
27         <artifactId>spring-webmvc</artifactId>
28         <version>5.3.2</version>
29     </dependency>
30
31     <dependency>
32         <groupId>junit</groupId>
33         <artifactId>junit</artifactId>
34         <version>4.12</version>
35     </dependency>
36
37 </dependencies>
38 </project>

```

导入后我们如何确认: 这里的junit4.12就是我导入的包



```
1 package com.Demo.pojo;
2
3 public class User {
4     private String name;
5     private int age;
6
7     public String getName() {
8         return name;
9     }
10
11     public void setName(String name) {
12         this.name = name;
13     }
14
15     public int getAge() {
16         return age;
17     }
18
19     public void setAge(int age) {
20         this.age = age;
21     }
22
23     @Override
24     public String toString() {
25         return "User{" +
26             "name=" + name + '\'' +
27             ", age=" + age +
28             '\'';
29     }
30 }
31
```

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:p="http://www.springframework.org/schema/p"
5       xsi:schemaLocation="http://www.springframework.org/schema/beans
6                           https://www.springframework.org/schema/beans/spring-beans.xsd">
7
8     <!--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）-->
9     <bean id="user" class="com.Demo.pojo.User" p:name="Syc" p:age="18"/>
10
11 </beans>
```

```
1 import com.Demo.pojo.Student;
2 import com.Demo.pojo.User;
3 import org.junit.Test;
4 import org.springframework.context.ApplicationContext;
5 import org.springframework.context.support.ClassPathXmlApplicationContext;
6
7 public class MyTest {
8     public static void main(String[] args){...}
9
10
11     @Test
12     public void test2(){
13         ApplicationContext context= new ClassPathXmlApplicationContext( configLocation: "userbeans.xml");
14         User user = context.getBean( s: "user", User.class);
15         System.out.println(user);
16     }
17 }
18
19
20
21
```

```
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）

```
<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：

```

1 package com.Demo.pojo;
2
3 public class User {
4     private String name;
5     private int age;
6
7     //User的无参构造
8     public User() {
9     }
10
11     //User的name, age 有参构造
12     public User(String name, int age) {
13         this.name = name;
14         this.age = age;
15     }
16
17     public String getName() {
18         return name;
19     }
20
21     public void setName(String name) {
22         this.name = name;
23     }
24 }

```

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:p="http://www.springframework.org/schema/p"
5       xmlns:c="http://www.springframework.org/schema/c"
6       xsi:schemaLocation="http://www.springframework.org/schema/beans
7       https://www.springframework.org/schema/beans/spring-beans.xsd">
8
9     <!--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）-->
10    <bean id="user" class="com.Demo.pojo.User" p:name="Syc" p:age="18"/>
11
12    <!--C命名空间注入，通过构造器注入就是：constructs-args-->
13    <bean id="user2" class="com.Demo.pojo.User" c:name="sYc" c:age="19"/>
14
15 </beans>

```

```

1 import com.Demo.pojo.Student;
2 import com.Demo.pojo.User;
3 import org.junit.Test;
4 import org.springframework.context.ApplicationContext;
5 import org.springframework.context.support.ClassPathXmlApplicationContext;
6
7 public class MyTest {
8     public static void main(String[] args){...}
9
10
11     @Test
12     public void test2(){
13         ApplicationContext context = new ClassPathXmlApplicationContext("userbeans.xml");
14         User user = context.getBean("user2", User.class);
15         System.out.println(user);
16     }
17 }

```

修改这里就可以执行了

```

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

```

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-namespaces

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:

```

<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 than nested elements. constructor-arg

The following example uses the namespace to do the same thing as the from [Constructor-based Dependency Injection](#): c:

```
<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="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：

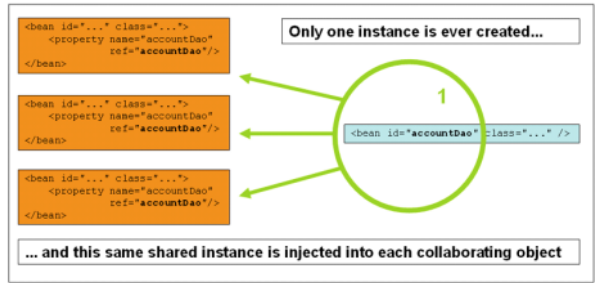
xmlns:c="http://www.springframework.org/schema/c"

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：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:p="http://www.springframework.org/schema/p"
5       xmlns:c="http://www.springframework.org/schema/c"
6       xsi:schemaLocation="http://www.springframework.org/schema/beans
7         https://www.springframework.org/schema/beans/spring-beans.xsd">
8
9     <!-- 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）-->
10    <bean id="user" class="com.Demo.pojo.User" p:name="Syc" p:age="18"/>
11
12    <!-- C命名空间注入，通过构造器注入就是，constructs-args-->
13    <bean id="user2" class="com.Demo.pojo.User" c:name="sYc" c:age="19"/>
14
15 </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）


```

1 import com.Demo.pojo.Student;
2 import com.Demo.pojo.User;
3 import org.junit.Test;
4 import org.springframework.context.ApplicationContext;
5 import org.springframework.context.support.ClassPathXmlApplicationContext;
6
7 public class MyTest {
8     public static void main(String[] args){...}
9
10
11
12
13
14     @Test
15     public void test2(){
16         ApplicationContext context= new ClassPathXmlApplicationContext( configLocation: "userbeans.xml");
17         User user = context.getBean( s: "user",User.class);
18         User user2 = context.getBean( s: "user",User.class);
19         System.out.println(user==user2);
20     }
21 }
22

```

```

287 ms
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

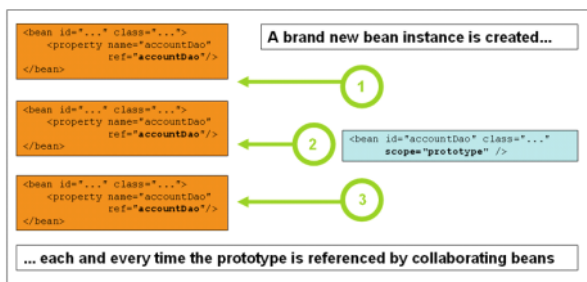
```

<!--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

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:p="http://www.springframework.org/schema/p"
5     xmlns:c="http://www.springframework.org/schema/c"
6     xsi:schemaLocation="http://www.springframework.org/schema/beans
7         https://www.springframework.org/schema/beans/spring-beans.xsd">
8
9     <!--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）-->
10    <bean id="user" class="com.Demo.pojo.User" p:name="Syc" p:age="18"/>
11
12    <!--C命名空间注入，通过构造器注入就是: constructs-args-->
13    <bean id="user2" class="com.Demo.pojo.User" c:name="sYc" c:age="19" scope="prototype"/>
14
15 </beans>

```

```
User.java x userbeans.xml x MyTest.java x
1 import com.Demo.pojo.Student;
2 import com.Demo.pojo.User;
3 import org.junit.Test;
4 import org.springframework.context.ApplicationContext;
5 import org.springframework.context.support.ClassPathXmlApplicationContext;
6
7 public class MyTest {
8     public static void main(String[] args){...}
9
10
11
12
13
14     @Test
15     public void test2(){
16         ApplicationContext context= new ClassPathXmlApplicationContext( configLocation: "userbeans.xml");
17         User user = context.getBean( < "user2",User.class);
18         User user2 = context.getBean( < "user2",User.class);
19
20         System.out.println(user.hashCode());
21         System.out.println(user2.hashCode());
22         System.out.println(user==user2);
23     }
24 }
25
```

测试已失败: 1共 1 个测试 - 291 ms

```
291ms
291 ms
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

```
Dog.java x Cat.java x People.java x beans.xml x MyTest.java x
1 package com.Demo.pojo;
2
3 public class Cat {
4     public void shout(){
5         System.out.println("喵喵喵~" );
6     }
7 }
8
```

```
Dog.java x Cat.java x People.java x beans.xml x MyTest.java x
1 package com.Demo.pojo;
2
3 public class Dog {
4     public void shout(){
5         System.out.println("汪汪汪~");
6     }
7 }
8
```

People.java代码:

```
package com.Demo.pojo;
```

```
public class People {
    private Cat cat;
    private Dog dog;
    private String name;

    public Cat getCat() {
        return cat;
    }

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

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

```

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 + "'\n" +
        '}';
}
}

```

```

Dog.java  Cat.java  People.java  beans.xml  MyTest.java
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xmlns:p="http://www.springframework.org/schema/p"
5      xsi:schemaLocation="http://www.springframework.org/schema/beans
6          https://www.springframework.org/schema/beans/spring-beans.xsd">
7
8      <bean id="cat" class="com.Demo.pojo.Cat"/>
9      <bean id="dog" class="com.Demo.pojo.Dog"/>
10
11     <bean id="people" class="com.Demo.pojo.People" p:name="张山">
12         <property name="dog" ref="dog"/>
13         <property name="cat" ref="cat"/>
14     </bean>
15 </beans>

```

```

Dog.java  Cat.java  People.java  beans.xml  MyTest.java
1  import com.Demo.pojo.People;
2  import org.junit.Test;
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  public class MyTest {
7      @Test
8      public void test(){
9          ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
10
11          People people = context.getBean("people", People.class);
12          people.getDog().shout();
13          people.getCat().shout();
14      }
15  }
16

```

```

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

```

• ByName自动装配:

```

Dog.java  Cat.java  People.java  beans.xml  MyTest.java
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xmlns:p="http://www.springframework.org/schema/p"
5      xsi:schemaLocation="http://www.springframework.org/schema/beans
6          https://www.springframework.org/schema/beans/spring-beans.xsd">
7
8      <bean id="cat" class="com.Demo.pojo.Cat"/>
9      <bean id="dog" class="com.Demo.pojo.Dog"/>
10
11     <bean id="people" class="com.Demo.pojo.People" p:name="张山" autowire="byName"/>
12 </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
 http://www.springframework.org/schema/context/spring-context.xsd">

 <context:annotation-config/>

</beans>
```

演示：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xmlns:context="http://www.springframework.org/schema/context"
5 xmlns:aop="http://www.springframework.org/schema/aop"
6 xmlns:p="http://www.springframework.org/schema/p"
7 xsi:schemaLocation="http://www.springframework.org/schema/beans
8 https://www.springframework.org/schema/beans/spring-beans.xsd
9 http://www.springframework.org/schema/context
10 https://www.springframework.org/schema/context/spring-context.xsd
11 http://www.springframework.org/schema/aop
12 https://www.springframework.org/schema/aop/spring-aop.xsd">
13
14 <bean id="cat" class="com.Demo.pojo.Cat"/>
15 <bean id="dog1" class="com.Demo.pojo.Dog"/>
16
17 <bean id="people" class="com.Demo.pojo.People" p:name="张山" autowire="byType"/>
18 </beans>
```

红色框我导入的是context, 黄色框我导入的是aop

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xmlns:context="http://www.springframework.org/schema/context"
5 xmlns:aop="http://www.springframework.org/schema/aop"
6 xmlns:p="http://www.springframework.org/schema/p"
7 xsi:schemaLocation="http://www.springframework.org/schema/beans
8 https://www.springframework.org/schema/beans/spring-beans.xsd
9 http://www.springframework.org/schema/context
10 https://www.springframework.org/schema/context/spring-context.xsd
11 http://www.springframework.org/schema/aop
12 https://www.springframework.org/schema/aop/spring-aop.xsd">
13
14 <!-- 开启注解支持 -->
15 <context:annotation-config/>
16
17 <bean id="cat" class="com.Demo.pojo.Cat"/>
18 <bean id="dog" class="com.Demo.pojo.Dog"/>
19 <bean id="people" class="com.Demo.pojo.People" />
20 </beans>
```

• @Autowired:

```
1 package com.Demo.pojo;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4
5 public class People {
6
7 // @Autowired 自动配置
8 @Autowired
9 private Cat cat;
10 @Autowired
11 private Dog dog;
12
13 private String name;
14
15 public Cat getCat() {
16 return cat;
17 }
18
19 public void setCat(Cat cat) {
20 this.cat = cat;
21 }
22
23 public Dog getDog() {
24 return dog;
25 }
26 }
```

```
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:

```
Dog.java x Cat.java x People.java x beans.xml x MyTest.java x
19 public void setCat(Cat cat) {
20 this.cat = cat;
21 }
22
23 public Dog getDog() {
24 return dog;
25 }
26
27 @Autowired
28 public void setDog(Dog dog) {
29 this.dog = dog;
30 }
31
32 public String getName() {
33 return name;
34 }
35
36 public void setName(String name) {
37 this.name = name;
38 }
39
40 @Override
41 public String toString() {
42 return "People{" +
 测试已通过: 1共 1 个测试 - 373ms
```

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:



```
1 package com.Demo.pojo;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.lang.Nullable;
5
6 public class People {
7
8 // @Autowired 自动配置
9 @Autowired
10 private Cat cat;
11 @Autowired
12 private Dog dog;
13 private String name;
14
15 // People的构造器
16 public People(@Nullable String name) {
17 this.name = name;
18 }
19
20 public Cat getCat() {
21 return cat;
22 }
23
24 public Dog getDog() {
```

测试 已通过: 1共 1 个测试 - 325 ms

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;
}
```

```
1 package com.Demo.pojo;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.lang.Nullable;
5
6 public class People {
7
8 // 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
9 @Autowired(required = false)
10 private Cat cat;
11 @Autowired
12 private Dog dog;
13 private String name;
14
15 public Cat getCat() {
16 return cat;
17 }
18
19 public Dog getDog() {
20 return dog;
21 }
22
23 public String getName() {
24 return name;
25 }
26}
```

测试 已通过: 1共 1 个测试 - 347 ms

当然我们也可以和@Qualifier配合使用

```
1 package com.Demo.pojo;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.beans.factory.annotation.Qualifier;
5 import org.springframework.lang.Nullable;
6
7 public class People {
8
9 @Autowired
10 @Qualifier(value = "cat0")
11 private Cat cat;
12 @Autowired
13 @Qualifier(value = "dog0")
14 private Dog dog;
15 private String name;
16
17 public Cat getCat() {
18 return cat;
19 }
20
21 public Dog getDog() {
22 return dog;
23 }
24}
```

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xmlns:context="http://www.springframework.org/schema/context"
5 xmlns:aop="http://www.springframework.org/schema/aop"
6 xmlns:p="http://www.springframework.org/schema/p"
7 xsi:schemaLocation="http://www.springframework.org/schema/beans
8 https://www.springframework.org/schema/beans/spring-beans.xsd
9 http://www.springframework.org/schema/context
10 https://www.springframework.org/schema/context/spring-context.xsd
11 http://www.springframework.org/schema/aop
12 https://www.springframework.org/schema/aop/spring-aop.xsd">
13
14 <!-- 开启注解支持-->
15 <context:annotation-config/>
16
17 <bean id="cat" class="com.Demo.pojo.Cat"/>
18 <bean id="catD" class="com.Demo.pojo.Cat"/>
19 <bean id="dog" class="com.Demo.pojo.Dog"/>
20 <bean id="dogD" class="com.Demo.pojo.Dog"/>
21 <bean id="people" class="com.Demo.pojo.People" />
22 </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

```

1 package com.Demo.pojo;
2
3 import javax.annotation.Resource;
4
5 public class People {
6
7 @Resource
8 private Cat cat;
9
10 @Resource
11 private Dog dog;
12 private String name;
13
14 public Cat getCat() {
15 return cat;
16 }
17
18 public Dog getDog() {
19 return dog;
20 }
21
22 public String getName() {
23 return name;
24 }
25
26 }

```

测试 已通过: 1共 1 个测试 - 333 ms

333ms D:\BX\Java1.8\bin\java.exe ...

汪汪汪~

喵喵喵~

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xmlns:context="http://www.springframework.org/schema/context"
5 xmlns:aop="http://www.springframework.org/schema/aop"
6 xmlns:p="http://www.springframework.org/schema/p"
7 xsi:schemaLocation="http://www.springframework.org/schema/beans
8 https://www.springframework.org/schema/beans/spring-beans.xsd
9 http://www.springframework.org/schema/context
10 https://www.springframework.org/schema/context/spring-context.xsd
11 http://www.springframework.org/schema/aop
12 https://www.springframework.org/schema/aop/spring-aop.xsd">
13
14 <!-- 开启注解支持-->
15 <context:annotation-config/>
16
17 <bean id="cat" class="com.Demo.pojo.Cat"/>
18 <bean id="catNoe" class="com.Demo.pojo.Cat"/>
19 <bean id="dog" class="com.Demo.pojo.Dog"/>
20 <bean id="dogNoe" class="com.Demo.pojo.Dog"/>
21 <bean id="people" class="com.Demo.pojo.People"/>
22 </beans>

```

beans: 1 bean

测试 已通过: 1共 1 个测试 - 322 ms

322ms D:\BX\Java1.8\bin\java.exe ...

汪汪汪~

喵喵喵~

当然它也可以像@Qualifier一样进行指定装配

```

1 package com.Demo.pojo;
2
3 import javax.annotation.Resource;
4
5 public class People {
6
7 @Resource(name = "catNoe")
8 private Cat cat;
9
10 @Resource
11 private Dog dog;
12 private String name;
13
14 public Cat getCat() {
15 return cat;
16 }
17
18 public Dog getDog() {
19 return dog;
20 }
21
22 public String getName() {
23 return name;
24 }
25 }

```

#### @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

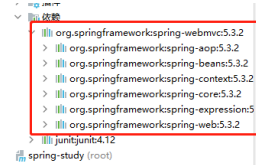
#### @Autowired总结:

@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 (Spring) 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"/>

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xmlns:context="http://www.springframework.org/schema/context"
5 xsi:schemaLocation="http://www.springframework.org/schema/beans
6 http://www.springframework.org/schema/beans/spring-beans.xsd
7 http://www.springframework.org/schema/context
8 http://www.springframework.org/schema/context/spring-context.xsd">
9
10 <!-- 指定扫描包, 被指定的包内注解就会生效-->
11 <context:component-scan base-package="com.Demo.pojo"/>
12 <!-- 下面是注解驱动的支持-->
13 <context:annotation-config/>
14
15 </beans>

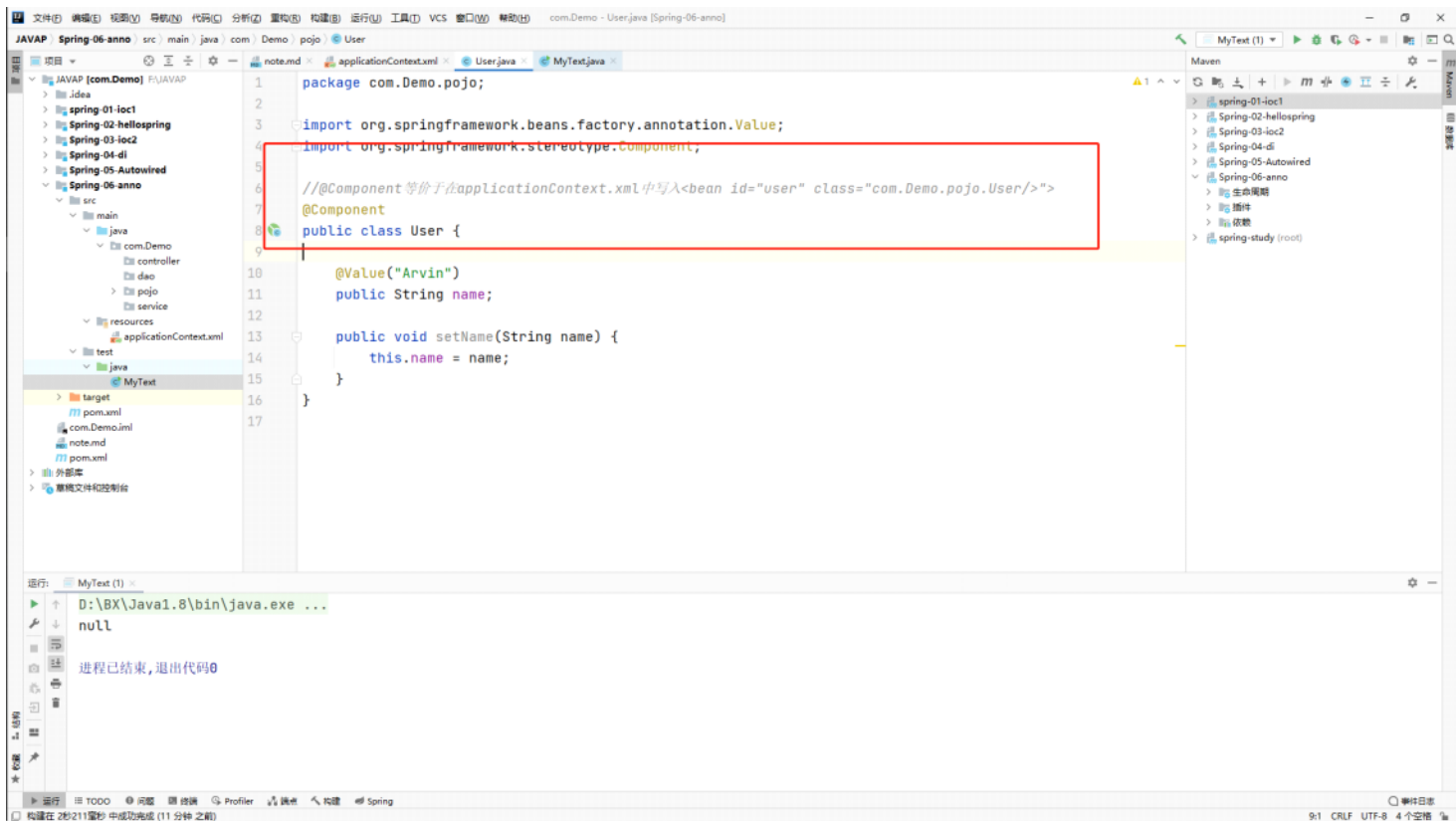
```

运行: MyText (1) x

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

null

进程已结束, 退出代码0



#### • @Value



@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

- @Component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-scan base-package="com.Demo"/>  
<!--下面这个是注解驱动的支持-->  
<context:annotation-config/>