

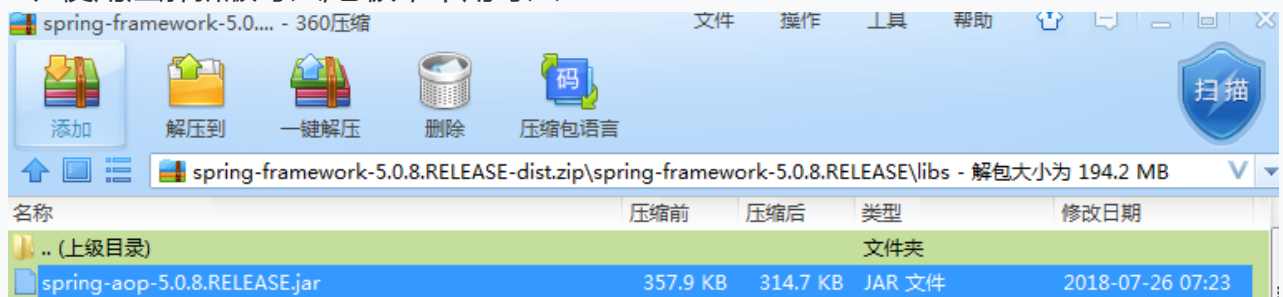
09 Spring_配置详解_注解代替xml配置

Spring

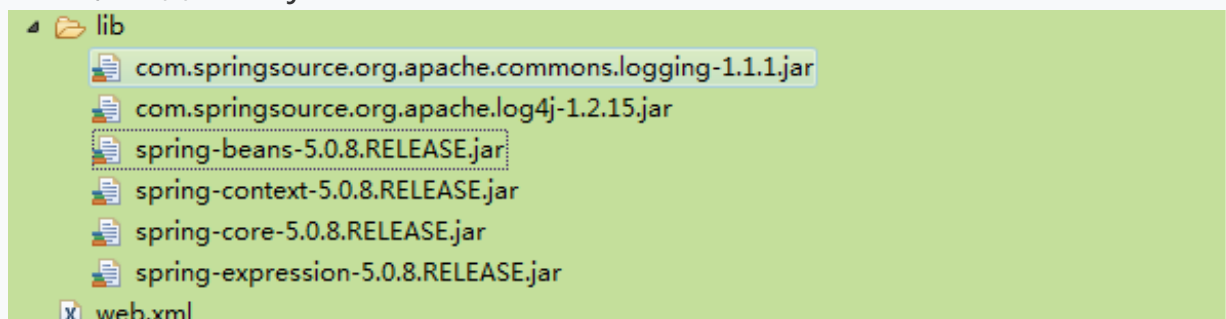
一:导入冥名 && 开始使用注解 && 注解 @Component

1、导入jar包

1、使用注解新版导入;老板本不用导入



2、之前必备导入的jar包



2、导入冥名空间

1、context冥名空间

注意:jar包的版本是否与使用的一致

```
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
http://www.springframework.org/schema/beans/spring-beans.xsd
6.     http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-5.0.xsd">
7. </beans>

```

3、开始使用注解

- 1、指定扫描包下的所有的类中的注解
- 2、注意:扫描指定包下所有的子孙包

```

1. <!--
2.     注意:扫描指定包下所有的子孙包
3. -->
4. <context:component-scan base-package="com.spring.bean">
</context:component-scan>

```

4、注解 @Component

- 1、相对于配置中的< bean name="users" class="com.spring.bean.Users">

```

1. import org.springframework.stereotype.Component;
2. //相对于配置中的<bean name="users" class="com.spring.bean.Users">
3. @Component("users")
4. public class Users {
5.     private Integer id;
6.     private String name;
7.     private Car car;
8.     //对应的get set 方法
9. }

```

5、测试

```

1. @org.junit.Test
2. public void getVoid02(){
3.     //1.创建容器对象
4.     ClassPathXmlApplicationContext classPath= new

```

```

5.      ClassPathXmlApplicationContext("applicationContext.xml");
6.      //2.找容器要对象
7.      Users bean = (Users)classPath.getBean("users");
8.      //3.打印对象单列为true
9.      System.out.println("bean:"+bean);
10.     }

```

```

<terminated> ITest.getVoid02 (1) [JUnit] D:\JDK\jdk-8u101-windows-x64\bin\javaw.exe (2018年8月14日 下午12:51:45)
log4j:WARN No appenders could be found for logger (org.springframework)
log4j:WARN Please initialize the log4j system properly.
bean:Users [id=null, name=null, car=null]

```

二:@Component | @Controller | @Repository | @Service

- 1、这些注解注释在类上，都是告诉spring，此类是需要被管理的bean，其中 `@Component` 功能最单一通用，仅是表明需要被spring管理的bean，其他三个除此之外，还代表了此类在应用中的角色。
- 2、@Component
|-需要被spring管理的bean，没有具体的角色。
- 3、@Controller
|-是一个前端控制器对象(三层架构中的表现层)
- 4、@Service
|-是一个业务层对象(三层架构中的业务层)
- 5、@Repository
|-是一个Dao对象(三层架构中的数据访问层)
- 6、在这里无论使用哪个注解都可以获取到Users的对象
- 7、四个注解都可以指定bean在容器中名字，如果未指定，则默认把类名首字母小写然后作为bean的名字

```

1.      @Component("users")
2.      @Service("users")//表示Service(业务逻辑)层的注解
3.      @Controller("users")//表示Controller(web)层的注解
4.      @Repository("users")//表示Repository(dao)层的注解

```

```

5.  public class Users {
6.      .....
7.  }

```

三:@Scope() 指定对象的作用域

1、实体类

scope属性(常用):

- | -singleton(默认值) : 单列对象,在Spring容器中只会有一个对象
- | -prototype : 多列原型,在Spring容器中每次都会方式新的对象
- | -session(了解) : web环境中,对象与session生命周期一致
- | -request(了解) : web环境中,对象与request生命周期一致

2、总结:在一般情况下使用默认值

3、注意 : 在Strtus和Spring整合是Strtus的actionBean必须为配置多列的

```

1.  @Repository("users")//表示Repository(dao)层的注解
2.  @Scope(scopeName="singleton")
3.  public class Users {
4.      private Integer id;
5.      private String name;
6.      private Car car;
7.      //对应的get set 方法
8.  }

```

2、测试

```

1.  @org.junit.Test
2.  public void getVoid02() {
3.      //1.创建容器对象
4.      ClassPathXmlApplicationContext classPath= new
ClassPathXmlApplicationContext("applicationContext.xml");
5.      //2.找容器要对象
6.      Users bean = (Users)classPath.getBean("users");
7.      Users bean1 = (Users)classPath.getBean("users");
8.      //3.打印对象单列为true
9.      System.out.println("bean:"+(bean==bean1));

```

```
10.     }
```

四:@Value("值")属性注入

1、方式一

1、直接在属性名上加入注解

```
1.     @Repository("users")
2.     @Scope(scopeName="singleton")
3.     public class Users {
4.         @Value("123")
5.         private Integer id;
6.         private String name;
7.         private Car car;
8.         //对应的get set 方法
9.     }
```

2、方式二

1、在set方法上加入注解

```
1.     @Repository("users")//表示Repository(dao)层的注解
2.     @Scope(scopeName="singleton")
3.     public class Users {
4.         private Integer id;
5.         private String name;
6.         private Car car;
7.         @Value("陈老师")
8.         public void setName(String name) {
9.             this.name = name;
10.        }
11.        //对应的get set 方法
12.    }
```

3、区别

- 1、加在属性名上是字段上通过反射Filed赋值(破坏了封装的特性)
- 2、加在set方法上是通过set方法赋值(推荐)

4、EL表达式和sqEL表达式(开发中很少使用)

4-1、EL表达式

- 1、@Value注解的功能不仅如此，还可以使用"\${值}"
- 2、配置

```
1. <context:property-placeholder location="db.porperties"/>
2. <context:component-scan base-package="com.spring.bean01">
    </context:component-scan>
```

3、db.porperties

```
1. jdbc.driverClass=com.mysql.jdbc.Driver
2. jdbc.jdbcUrl=jdbc:mysql://172.16.10.164:3306/hibernates?
   useUnicode=true&characterEncoding=utf-8
3. jdbc.user=root
4. jdbc.password=root
```

4、实体类

```
1. @Repository("users")
2. public class Users {
3.     @Value("${jdbc.driverClass}")
4.     private String name;
5.     public void getVoid(){
6.         System.out.println(name);
7.     }
8. }
```

5、测试

```

1.  @RunWith(SpringJUnit4ClassRunner.class)
2.  @ContextConfiguration("classpath:applicationContext.xml")
3.  public class Test {
4.      @Resource(name="users")
5.      private Users users;
6.      @org.junit.Test
7.      public void getVoid02() throws Exception{
8.          users.getVoid();
9.      }
10. }

```

```

<terminated> Test.getVoid02 (2) [JUnit] D:\JDK\jdk-8u101-windows-x64\jdk\bin\javaw.exe (2018年8月18日 上午1:28:39)
log4j:WARN No appenders could be found for logger (org.sp
log4j:WARN Please initialize the log4j system properly.
com.mysql.jdbc.Driver

```

4-2、sqEL表达式(开发中很少使用)

- 1、测试同上
- 2、获取当前系统的语言

```

1.  @Repository("person")
2.  public class Person {
3.      //获取当前系统的语言
4.      @Value("#{systemProperties['user.language']}")
5.      private String name;
6.      public String getName() {
7.          System.out.println(name);
8.          return name;
9.      }
10. }

```

- 3、获取其他Bean对象中的方法

```

1.  @Repository("person")
2.  public class Person {
3.      //获取其他对象的中方法
4.      @Value("#{users.getVoid()}")
5.      private String name;
6.      public String getName() {
7.          System.out.println("person对象:"+name);
8.          return name;
9.      }
10. }

```

4、配置语法相同

五: 对象引用注入

1、方式一 :@Autowired("对象名")引用注入 | @Qualifier

- 1、自动装配：通过类型进行扫描在进行装配
- 2、@Autowired(required=true) | @Autowired()
|-两种写法含义是相同的，如果容器无法提供bean，则异常。
- 3、@Autowired(required=false)
|-含义是需要注入，如果没有呢也无所谓
- 4、如匹配多个,无法主动装配就使用 @Qualifier("对象名称")

1)、实体类

```

1.  @Component("car")
2.  public class Car {
3.      @Value("蓝博基尼")
4.      private String name;
5.      private String color;
6.      //对应的get set方法
7.  }

```

```

1.  @Repository("users")//表示Repository(dao)层的注解
2.  @Scope(scopeName="singleton")
3.  public class Users {

```



```

4.     private Integer id;
5.     @Value("陈老师")
6.     private String name;
7.     @Autowired
8.     @Qualifier("car")//如出现多个同对象,告诉Spring自动装配那个对象
9.     private Car car;
10.    //对应的get set方法
11.    }

```

2)、其寻找依赖的顺序

1、类型(byType)

|-注释在setter和构造器上，选择参数类型

|-注释在属性上，选择属性类型

|-注:如果容器中相同的类型的bean超过1个，则使用byName方式

2、指定名称(byName)

|-通过注解@Qualifier来指定名称,则仅去容器中寻找指定名称的bean

3)、@Required

1、仅用于在setter方法上使用，用于验证此依赖是否已注入，如果未注入，则spring在构建bean时抛出异常

4)、@Qualifier

1、提供注入限制，必须按照名字注入

2、方式二 :JSR-250标准(了解)_@Resource(name="对象名")

1、JSR-330标准中提供一些用于注入的公共注解，放在javax.annotation包下

2、Spring支持其中的@Resource和@ManagedBean

3、@ManagedBean 对应@Component

4、@Resource 对应@Autowired，但注入顺序为先byName，然后bytype,最后按照@Qualifier指定名称注入如果有的话

1)、实体类

```

1.  @Component("car")
2.  public class Car {
3.      @Value("蓝博基尼")
4.      private String name;
5.      private String color;
6.      //对应的get set方法
7.  }

```


```

1.  import javax.annotation.Resource;
2.  @Repository("users")//表示Repository(dao)层的注解
3.  @Scope(scopeName="singleton")
4.  public class Users {
5.      private Integer id;
6.      @Value("陈老师")
7.      private String name;
8.      @Resource(name="users")//手动注入
9.      private Car car;
10.     //对应的get set方法
11. }

```

3、方式三:JSR-330标准(了解)

1、JSR-330标准中提供一些用于注入的公共注解，放在javax.inject包中(需额外导包，不在JRE中)。

 javax.inject-2.5.0-b61.jar

2、包含如下注解，Spring在3.0后开始支持。功能和spring提供的类似

|-@Inject 对应@Autowired

|-@Singleton 对应@Scope("singleton")

|-@Named 对应@Qualifier

1)、实体类

```

1.  @Component("car")
2.  public class Car {
3.      @Value("蓝博基尼")
4.      private String name;
5.      private String color;
6.      //对应get set 方法
7.  }

```

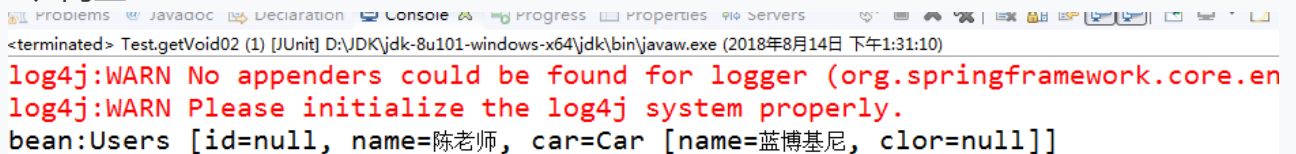
```

1.  @Repository("users")//表示Repository(dao)层的注解
2.  @Singleton//javax.inject包中
3.  public class Users {
4.      private Integer id;
5.      @Value("陈老师01")
6.      private String name;
7.      @Inject()//javax.inject包中
8.      @Named(value="car")//javax.inject包中
9.      private Car car;
10.     //对应的get set 方法
11. }

```

4、测试

1、同上



<terminated> Test.getVoid02 (1) [JUnit] D:\JDK\jdk-8u101-windows-x64\jdk\bin\javaw.exe (2018年8月14日 下午1:31:10)
 log4j:WARN No appenders could be found for logger (org.springframework.core.en
 log4j:WARN Please initialize the log4j system properly.
 bean:Users [id=null, name=陈老师, car=Car [name=蓝博基尼, color=null]]

六: @PostConstruct | @PreDestroy (初始化|销毁)

1、jdk官方提供

1)、实体类

```

1.  @Repository("users")//表示Repository(dao)层的注解
2.  @Scope(scopeName="singleton")
3.  public class Users {
4.      private Integer id;
5.      @Value("陈老师")
6.      private String name;
7.      @Resource(name="car")
8.      private Car car;
9.
10.     @PostConstruct//相对于init-method=""配置初始化
11.     public void init(){
12.         System.out.println("init初始化的方法");

```

```

13.     }
14.
15.     @PreDestroy//相对于destroy-method=""配置销毁
16.     public void destroy() {
17.         System.out.println("destroy初始化的方法");
18.     }
19.
20.     //对应的get set 方法
21. }

```

2)、扩展_实现接口(不推荐)

1、让我们的bean实现InitializingBean 和DisposableBean两个即可，每个接口提供了一个方法

```

1. import org.springframework.beans.factory.DisposableBean;
2. import org.springframework.beans.factory.InitializingBean;
3. public class Person implements InitializingBean, DisposableBean {
4.     //实现的方法
5. }

```

3)、测试

```

1. @org.junit.Test
2. public void getVoid02() {
3.     //1.创建容器对象
4.     ClassPathXmlApplicationContext classPath= new
ClassPathXmlApplicationContext("applicationContext.xml");
5.     //2.找容器要对象
6.     Users bean = (Users)classPath.getBean("users");
7.     //3.打印对象单列为true
8.     System.out.println("bean:"+bean);
9.     classPath.close();
10. }

```

<terminated> Test.getVoid02 (1) [JUnit] D:\JDK\jdk-8u101-windows-x64\jdk\bin\javaw.exe (2018年8月14日 下午1:47:51)

log4j:WARN No appenders could be found for logger (org.springframework.cc
log4j:WARN Please initialize the log4j system properly.

init初始化的方法

bean:Users [id=null, name=陈老师, car=Car [name=蓝博基尼, clor=null]]

destroy初始化的方法

4)、多种方式混合使用执行顺序

- 1、注解
- 2、接口的方式
- 3、xml配置回调方法

如果有出现多种方式使用同一个方法，则此方法仅执行一次

七: Bean 创建先后

- 1、bean A使用ref指向beanB时，已经隐含了B要先于A创建，但有的时候两个bean的创建时不相互依赖，但从逻辑或业务角度具有依赖关系。
- 2、如Bean B 是进行jdbc驱动注册作用的，Bean A是创建Connection的，这两个bean在创建时是没有依赖的。
- 3、但从逻辑角度讲，得先进行jdbc驱动注册，然后才能获取Connection。
- 4、针对于这种隐含的依赖关系，可使用使用depends-on属性来显示指定，而且可以指定多个(使用逗号，分号，空白字符等分割)

1、实体类

```
1. public class Cat {
2.     private String name;
3.     private Dog dog;
4.     //对应的get set 方法
5. }
```

```
1. public class Dog {
2.     public Dog() {
3.         super();
4.         System.out.println("----Dog() -----");
5.     }
6. }
```

2、配置

```
1. <bean id="c1" class="bean.Cat" depends-on="d2" >
2.     <property name="name" value="旺财"></property>
3.     <property name="dog" ref="d2"></property>
4. </bean>
5. <bean id="d2" class="bean.Dog" >
6. </bean>
```

3、测试

```
2018-08-17 04:43:31 [main] DEBUG [org.springframework.context]
2018-08-17 04:43:31 [main] DEBUG [org.springframework.context]
2018-08-17 04:43:31 [main] DEBUG [org.springframework.beans]
2018-08-17 04:43:31 [main] DEBUG [org.springframework.beans]
2018-08-17 04:43:31 [main] DEBUG [org.springframework.beans]
----Dog() -----
2018-08-17 04:43:31 [main] DEBUG [org.springframework.beans]
2018-08-17 04:43:31 [main] DEBUG [org.springframework.beans]
2018-08-17 04:43:31 [main] DEBUG [org.springframework.beans]
2018-08-17 04:43:31 [main] DEBUG [org.springframework.beans]
Cat()
2018-08-17 04:43:31 [main] DEBUG [org.springframework.beans]
```