Spring注解驱动第一讲--Spring环境搭建

第一步:

创建一个mvn工程,并在pom文件中引入如下版本的Spring-context

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.3.12.RELEASE</version>
</dependency>
```

第二步:在所创建的项目中创建配置类,使用@Configuration注解标识,即为配置类.在配置类中,填写获取bean的方法,方法的返回值为获取的bean类型,方法名即为bean在容器中的名字.代码如下:

```
@Configuration
public class MainConfig {
    @Bean//注解@Bean标注的方法可理解为,以往配置文件中的bean标签,Person类即可在Spring的容器中存在
    public Person person01() {
        return new Person("张三",18);
    }
}
```

Person类:

```
public class Person {
    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 "Person [name=" + name + ", age=" + age + "]";
    }
    public Person(String name, int age) {
        super();
```

```
        this.name = name;
        this.age = age;
    }
    public Person() {
        super();
    }
}
```

测试代码如下:

```
public class MainTest {
    public static void main(String[] args) {
        ApplicationContext applicationContext =  new
AnnotationConfigApplicationContext(MainConfig.class);
        Person obj = applicationContext.getBean(Person.class) ;
        System.out.println(obj);
    }
}
```

注:在获取bean对象的时候,也可以写成applicationContext.getBean("person") ;注意类型的转换即可,也可以通过@Bean("自定义容器中bean的名称"),为容器中的bean改名

Spring注解驱动第二讲--@ComponentScan扫描介绍

关于包扫描的介绍.(@ComponentScan)

在以往采用配置文件,配置扫描包时,会使用<context:component-scan base-package="需要扫描的报名前缀">,即可,之后Spring会自动加载类上带有@Controller,@Service,@Repository,@Component的类;

在Spring注解开发中,只需要在主配置类上添加@ComponentScan(value="需要扫描的报名前缀")注解即可;示例代码如下

```
@Configuration//标记此类为配置类 @ComponentScan(value="com.cnblogs")    public class
MainConfig {    }
```

如果想排除或者只包含指定类型的文件,可使用incloudFiltes和excludeFilters属性进行配置.

1:excludeFilters属性可以接收数组类型,数组里面存放的是@Filter注解入:

```
@Configuration//标记此类为配置类 @ComponentScan(value="com.cnblogs",
excludeFilters= { @Filter(type=FilterType.ANNOTATION,classes= {Controller.class}) }) public
class MainConfig { }
```

以上代码,指定容器中排除加载带有@Controller注解的类.@Filter注解的type属性,注明按照什么方式进行过滤.常用的为FilterType.ANNOTATION(按照注解的方式进行过滤).class属性中存放需要排除的bean.这里只是初步的对@Filter注解进行介绍.下一篇中会有详细的介绍;

2,includeFilters属性,让容器实现只加指定的类.这里需要说一下@ComponentScan注解默认会加载所有带有@Controller,@Service,@Repository,@Component的类到容器中,所以要想实现加载指定的类,需要把@ComponentScan注解的默认加载所有类关掉.即useDefaultFilters = false. 代码示例如下:

```
@Configuration//标记此类为配置类 @ComponentScan(value="com.cnblogs",includeFilters= {
@Filter(type=FilterType.ANNOTATION,classes= {Controller.class})    },useDefaultFilters =
```

```
false)     public class MainConfig {   }
```

以上代码为只加载带有@Controller注解的类

注:如果使用的是jdk1.8,@ComponentScan可以在一个类上配置多个:

```
@Configuration//标记此类为配置类 @ComponentScan(...)   @ComponentScan(...)   public class
MainConfig {    }
```

如果不是jdk1.8可使用@ComponentScans注解里面配置多个@ComponentScan,代码如下:

```
@Configuration//标记此类为配置类 @ComponentScans(    value={
@ComponentScan(...),@ComponentScan(...)         })    public class MainConfig {   }
```

Spring注解驱动第三讲--@Filter介绍

上一讲主要针对@ComponentScan注解做了一些说明,本文主要对@Filter的扫描条件,再做一些详细的介绍

1,FilterType.ANNOTATION 按照注解的方式进行扫描.后面classes属性,为注解的类型,如:

```
@Configuration//标记此类为配置类 @ComponentScan(value="com.wxj",excludeFilters= {
@Filter(type=FilterType.ANNOTATION,classes= {Controller.class})   })   public class
MainConfig {        //将标有@Controller注解的类排除在外不会加载到容器中来    }
```

2,type=FilterType.ASSIGNABLE_TYPE,按照指定的类,进行过滤,后面的classes属性的值为"类名.class".如:

```
@Configuration//标记此类为配置类 @ComponentScan(value="com.wxj",  includeFilters= {
@Filter(type=FilterType.ASSIGNABLE_TYPE,classes= {BookService.class})
},useDefaultFilters=false)  public class MainConfig {       //只会加载BookService,以及
BookService的子类或者其实现类    }
```

以上两种方式为常用的过滤方式.

3,FilterType.CUSTOM,按照自己自定义的方式来进行过滤和筛选(使用此过滤类型,虽然比较繁琐,但是使用起来完全可以由自己来定义扫描的规则)

首先定义@Filter注解的类型

```
@Configuration//标记此类为配置类 @ComponentScan(value="com.wxj",  includeFilters= {
@Filter(type=FilterType.CUSTOM,classes{MyTypeFilter.class})  },useDefaultFilters=false)
public class MainConfig {   }
```

MyTypeFilter即为自己定义的匹配方法,其中MyTypeFilter类中的match方法的返回值为true时,为符合过滤条件,如果返回为false,则不符合过滤条件,代码如下:

```
public class MyTypeFilter implements TypeFilter {    /**         * MetadataReader
metadataReader:获取当前正在扫描的类的信息         * MetadataReaderFactory
metadataReaderFactory,获取带其他任何类的信息          *          */        public boolean
match(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory) throws
IOException {           AnnotationMetadata annotationMetadata =
metadataReader.getAnnotationMetadata();//获取当前类的注解信息             ClassMetadata
classMetadata = metadataReader.getClassMetadata();//获取当前扫描的类信息
Resource resource = metadataReader.getResource();//获取当前扫描的资源信息             String
name = classMetadata.getClassName();//获取类的名字             if(name.contains("er")) {
return true;//如果类的名字中带有"er",则符合过滤的要求          }          return false;
}    }
```

注:TypeFilter中的ASPECTJ和REGEX(正则方式),没有介绍,使用较少,有兴趣可以自行研究.

Spring注解驱动第四讲--@Scope注解

在Spring容器加载扫描的类时,可以通过@Scope注解来控制加载到容器中的Bean是单例还是多例

@Scope注解中的属性值有以下几种

singleton:单例,容器启动时创建Bean对象,且容器中只有一个实例.

prototype:多例,容器启动时不去创建Bean对象,每当获取的Bean的时候容器才去创建一个新的Bean对象.

request:同一次请求创建一个实例(web阶段时使用,本文不介绍)

session:同一个session创建一个实例(web阶段时使用,本文不介绍)

实例代码如下:

1,创建配置类:

```
@Configuration   public class MainConfig2 {      @Scope("singleton")//单例模式
@Bean     public Person getPerson() {          System.out.println("容器开始创建
bean.........");         return new Person("张三",23);      }}
```

2,创建测试类

```
public class IOCTest {          @Test      public void test2() {
ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(MainConfig2.class);
          syso("加载容器完成");          Person person1 =
applicationContext.getBean(Person.class) ;          Person person2 =
applicationContext.getBean(Person.class) ;          System.out.println(person1 == person2);
}     }
```

运行结果: 容器开始创建bean......... 加载容器完成 True

如果将@Scope注解里面的的singleton改为prototype,再次运行。运行结果: 加载容器完成 容器开始创建bean......... 容器开始创建bean......... false

通过两次运行结果,可以观察到prototype为容器加载完成后,每次调用时在创建,singleton为创建容器时就已经创建完成

附加:

在使用singleton的属性时(默认情况下).如果想在容器创建时不先创建Bean,而是在第一次获取的时候才创建,即赖加载.可以使用@Lazy注解.代码实现如下:

```
@Configuration   public class MainConfig2 {      //默认情况下为单例模式      @Lazy//启用懒加
载       @Bean      public Person getPerson() {          System.out.println("容器开始创建
bean.........");         return new Person("张三",23);      }}
```

运行结果: 加载容器完成 容器开始创建bean......... true

可以发现,在创建完容器之后,获取Bean的时候才去创建Bean对象,而且只会在第一次获取的时候才会创建。

Spring注解驱动第五讲--@Conditional注解

在Spring容器创建bean的时候也可以按照自定义的条件来决定创建哪些bean对象.@Conditional注解可以帮助实现此需求场景.

本篇模拟一下:在不同操作系统环境下创建不同的bean到容器中.

@Conditional注解的源码如下:

```
@Target({ElementType.TYPE, ElementType.METHOD})   @Retention(RetentionPolicy.RUNTIME)
@Documented      public @interface Conditional {      Class<? extends Condition>[] value();
}
```

可以发现此注解可以加在类上和方法上,注解中的值为class类型,且实现Condition接口.

创建两个自定义条件类:

```
public class MacConditional implements Condition {          public boolean
matches(ConditionContext context, AnnotatedTypeMetadata metadata) {              //获取运行
```

```
环境信息            Environment environment = context.getEnvironment();          String
osName = environment.getProperty("os.name");              if(osName.contains("Mac")) {//如
果在Mac系统下,返回true,即满足条件                  return true ;          }          return
false;          }}
```

```
public class WindowsConditional implements Condition{      /**          * ConditionContext,
判断条件使用的上下文        * AnnotatedTypeMetadata,注释信息       */        public boolean
matches(ConditionContext context, AnnotatedTypeMetadata metadata) {            //获取当前
ioc的运行环境,以及环境配置          Environment environment = context.getEnvironment();
String osName = environment.getProperty("os.name");
if(osName.contains("Windows")) {//如果运行环境为Win,则满足条件                  return true ;
}            return false;      }}
```

自定义的条件类定义完以后,在配置类中的方法上添加@Conditional注解

```
@Configuration   public class MainConfig2 {      //如果为Windows环境则创建windows对象
@Conditional({WindowsConditional.class})        @Bean("windows")       public Person
getBill() {              return new Person("windows操作系统",40) ;      }      //如果为Mac环境
则创建mac对象        @Conditional({MacConditional.class})        @Bean("mac")      public
Person getlinux() {           return new Person("mac",45) ;      }}
```

测试类:

```
public class IOCTest {              @Test      public void test4() {
ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(MainConfig2.class);          String[] names =
applicationContext.getBeanNamesForType(Person.class);//返回容器中所有bean的名字
for (String name : names) {             System.out.println(name);          }
Map<String, Person> map = applicationContext.getBeansOfType(Person.class) ;//返回容器中的所
有bean的对象             System.out.println(map);                }}
```

```
运行结果:   mac   {mac=Person [name=mac, age=45]}
```

扩展;

1,@Conditional注解还可以加在类上面.代表当满足条件是该配置类下的所有bean才会加载.

2,ConditionContext可以获取到很多信息,如类的注册信息,ioc工厂,获取类加载器等等

Spring注解驱动第六讲--@Import注解

在以前的博客中介绍了两种让容器自动装配bean的方法:

1,使用@Bean注解进行装配

2,使用@ComponentScan进行包扫描,扫描带有

@Controller,@Service,@Repository,@Component注解的类即可装载bean

还有一种方式,使用@Import注解进行对引用第三方类时使用

用法:

一,直接在主配置类上添加@Import注解,注解的值赋值上要创建的类型即可

首先创建一个要加载进来的外部类

```
public class Color {      ...     }
```

可以看到此类没有添加任何可以让包扫描到的注解,在容器创建也是不会加载到容器中

然后在主配置类上添加@Import注解

```
@Configuration   @Import({Color.class})//注解里面的值为需要IOC加载的类的类型  public class
MainConfig2 {    }
```

测试类和返回结果:

```
public class IOCTest {          @Test      public void test4() {
```

```
ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(MainConfig2.class);          String[] names =
applicationContext.getBeanDefinitionNames();//返回容器中所有bean的名字          for (String
name : names) {              System.out.println(name);          }      }}
```

运行结果: mainConfig2   com.wxj.bean.Color

观察可得,使用@Import导入的组件在ioc容器中默认的id为全类名

二,实现ImportSelector接口方式

1,创建一个方法来实现ImportSelector接口

```
public class MyImportSelector implements ImportSelector {        /**         *
AnnotationMetadata类中可以获得注解名字,等注解信息      *          * 返回的String数组为要让
@Import注解加载的类的全类名      */        public String[] selectImports(AnnotationMetadata
importingClassMetadata) {                    return new String[]
{"com.wxj.bean.Color"};//需要ioc容器管理的全类名,多个类可以由,分隔          }}
```

2,更改主配置类的@Import注解中的value值

```
@Configuration   @Import({MyImportSelector.class})      public class MainConfig2 {       }
```

3,再次运行

```
mainConfig2      com.wxj.bean.Color
```

三,实现ImportBeanDefinitionRegistrar接口(手工的自定义bean)

1,实现ImportBeanDefinitionRegistrar,代码如下:

```
public class MyImportBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {
/**       * importingClassMetadata:当前类的直接信息          * registry:bean定义的注册类,通
过使用它来为容器中注册bean       *          把所有需要加到容器中的bean通过手工注册,调用
BeanDefinitionRegistry的registerBeanDefinition方法进行手工注册          *                */
public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata,
BeanDefinitionRegistry registry) {            //指定bean的定义信息...
RootBeanDefinition beanDefinition = new RootBeanDefinition(Color.class);              //给容
器中的bean自定义id            registry.registerBeanDefinition("Color", beanDefinition);
}    }
```

2,更改主配置类

```
@Configuration   @Import({MyImportBeanDefinitionRegistrar.class})  public class MainConfig2
{    }
```

3,运行结果 mainConfig2   Color

注:在以后阅读SpringBoot源码时,第二种方法应用的极为广泛

Spring注解驱动第七讲--使用FactoryBean注册组件

在实际开发中也可以使用Spring提供的工厂bean来注册组件

首先创建工厂bean的实现

```
/**    * 泛型T即为通过工厂bean获得的bean对象  *     */   public class ColorFactory implements
FactoryBean<Color> {       /**         * 容器通过工厂bean的getObject方法获得要加载的bean
*/        public Color getObject() throws Exception {           return new Color();     }
/**        * 返回要在ioc容器中注册的bean的类型          */         public Class<?>
getObjectType() {             // TODO Auto-generated method stub          return
Color.class;         }     /**         * 设置bean对象在容器里面是否是单例          * 返回为
true:单例的           * 返回为flase:多例的        */         public boolean isSingleton() {
return false;        }}
```

在主配置类中通过@Bean注解将ColorFactory注册到容器中

```
@Configuration  public class MainConfig2 {     //方法名为默认id      @Bean      public
ColorFactory getColorFactory() {         return new ColorFactory();         } }
```

在测试类中我们通过获取容器中组件id的方式获取容器中的bean对象

```
public class IOCTest {          @Test      public void test4() {
ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(MainConfig2.class);          Object colorFactory =
applicationContext.getBean("getColorFactory") ;           System.out.println(colorFactory);
}     }
```

运行结果: com.wxj.bean.Color@57d5872c

我们发现容器中存在的并不是ColorFactory对像,而是Color对象.通过观察可知,容器通过工厂bean的getObject()方法返回的对象来进行创建并注册到容器中.

那么如果非要获取ColorFactory对象,应该怎么获取呢?需要在获取bean的id名称前添加一个"&"符即可.

```
public class IOCTest {          @Test      public void test4() {
ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(MainConfig2.class);          Object colorFactory =
applicationContext.getBean("&getColorFactory") ;//"&"可获得工厂bean对象
System.out.println(colorFactory);                    } }
```

运行结果: com.wxj.ColorFactory@57d5872c

扩展:在使用单实例的时候,getObject方法只会被调用一次,而在多实例的情况下每次获取bean都会调用getObject()方法.

Spring注解驱动第八讲--容器中bean的生命周期

bean的生命周期指的就是bean在容器中的:

创建-->初始化-->销毁;

以上的过程都是由容器来进行管理.

我们可以自定义初始化和销毁方法,的那个进行到当前bean的生命周期的时候,调用我们自己定义的初始化方法和销毁方法.那么自定义初始化和销毁方法有以下四种方式:

1,指定初始化和销毁方法:

在以往使用xml配置文件的时候可以在<bean>标签中加上"init-method"和"destory-method"属性来指定自定义的初始化和销毁方法,本文将不进行详细介绍;

2,使用@Bean注解的initMethod属性和destoryMethod属性来指定初始化方法和销毁方法

创建主配置类

```
@Configuration  public class MyconfigOfLifeCycle {
@Bean(initMethod="init",destroyMethod="destory")//指定销毁和初始化方法,初始化方法是在创建对
象之后执行,销毁方法是在容器关闭时执行      public Mouse mouse() {          return new
Mouse() ;      } }
```

创建Mouse的类

```
public class Mouse {      public Mouse() {          System.out.println("构造器创建
Mouse.....");       }     public void init() {          System.out.println("初始化
Mouse.....");       }     public void destory() {         System.out.println("销毁
Mouse.....");       } }
```

测试类:

```
public class IOCTest_lifeCycle {     @Test      public void test01(){
AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext(MyconfigOfLifeCycle.class);
System.out.println("容器创建完成.........");              applicationContext.close();//关闭
容器,关闭容器时,才会调用destory方法        }}
```

运行结果:

```
七月 23, 2019 9:15:52 下午
org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh信
息: Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@311d617d: startup
date [Tue Jul 23 21:15:52 CST 2019]; root of context hierarchy     构造器创建
Mouse.......... 初始化Mouse..........       容器创建完成..... 七月 23, 2019 9:15:52 下
午 org.springframework.context.annotation.AnnotationConfigApplicationContext doClose     信
息: Closing
org.springframework.context.annotation.AnnotationConfigApplicationContext@311d617d: startup
date [Tue Jul 23 21:15:52 CST 2019]; root of context hierarchy     销毁Mouse..........
```

可以观察到,容器先去创建bean实体,然后进行初始化方法,当容器销毁时,对象也随之销毁.

注:在多实例的情况下,容器不会去创建bean,只有在调用时才会执行,创建方法和初始化方法,而关闭容器时容器也不会销毁对象实例.

2,将Bean的类实现InitializingBean和DisposableBean接口.

```
/**    * InitializingBean接口中含有afterPropertiesSet方法,即在bean进行创建之后开始执行    *
DisposableBean接口中含有destroy方法,即在BeanFatory销毁的时候开始执行销毁方法   *     */
@Component//让容器可以扫描到      public class Cat implements
InitializingBean,DisposableBean{      public Cat() {             System.out.println("创建
Cat.............");         }     public void destroy() throws Exception {
System.out.println("初始化Cat.............");          }     public void afterPropertiesSet()
throws Exception {             System.out.println("销毁Cat.............");          }}
```

在主配置类中通过@ComponentScan注解扫描包下的bean

再一次执行测试类,运行结果如下:

```
创建Cat.............    销毁Cat.............    构造器创建Mouse..........  初始化
Mouse.......... 容器创建完成..... 七月 23, 2019 9:31:27 下午
org.springframework.context.annotation.AnnotationConfigApplicationContext doClose  信息:
Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@311d617d:
startup date [Tue Jul 23 21:31:26 CST 2019]; root of context hierarchy  销毁
Mouse.......... 初始化Cat.............
```

细心的读者可以发现,我竟然把吧打印的语句写反了^~^!!!!

3,可以使用JSR250提供的@PostConstruct在bean创建完成并属性值赋值完成后执行;@PreDestory在容器销毁bean之前执行.(两个注解都是加在方法上)

创建bean

```
@Component public class Dog {              public Dog() {
System.out.println("Dog创建.......");      }     @PostConstruct        public void init() {
System.out.println("Dog初始化.....@PostConstruct..");       }     @PreDestroy      public
void destory() {         System.out.println("Dog销毁.....@PreDestroy..");         }}
```

运行结果:

```
创建Cat.............    销毁Cat.............    Dog创建.......    Dog初始化.....@PostConstruct..
构造器创建Mouse..........  初始化Mouse..........       容器创建完成..... 七月 23, 2019
9:58:30 下午 org.springframework.context.annotation.AnnotationConfigApplicationContext
```

```
doClose    信息: Closing
org.springframework.context.annotation.AnnotationConfigApplicationContext@311d617d: startup
date [Tue Jul 23 21:58:30 CST 2019]; root of context hierarchy    销毁Mouse.....  Dog
销毁.....@PreDestroy..      初始化Cat.....
```

4,BeanPostProcessor:bean的后置处理器,在bean的初始化前后进行一些处理工作

首先创建自定义的类实现BeanPostProcessor接口

```
@Component public class MyBeanPostProcessor implements BeanPostProcessor{        /**
* 该方法在初始化之前执行         * beanName:初始化当前bean对象的名字,         * bean:初始化
当前的bean对象        *          * 返回的Object为初始化前在该方法中经过处理之后的对象(可原
样返回)         */        public Object postProcessBeforeInitialization(Object bean, String
beanName) throws BeansException {
System.out.println("postProcessBeforeInitialization...." +beanName + "=>" + bean);
return bean;         }     /**         * 该方法在初始化之后执行          * beanName:初始化当
前bean对象的名字,          * bean:初始化当前的bean对象       *          * 返回的Object为初始
化之后在该方法中经过处理之后的对象(可原样返回)          */        public Object
postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
System.out.println("postProcessAfterInitialization...." +beanName + "=>" + bean);
return bean;         } }
```

之后在执行测试类,观察运行结果

```
创建Cat.....    postProcessBeforeInitialization....cat=>com.wxj.bean.Cat@731f8236  初
始化Cat.....    postProcessAfterInitialization....cat=>com.wxj.bean.Cat@731f8236   Dog
创建.....       postProcessBeforeInitialization....dog=>com.wxj.bean.Dog@4f9a3314  Dog初始
化.....@PostConstruct..
postProcessAfterInitialization....dog=>com.wxj.bean.Dog@4f9a3314   构造器创建
Mouse..... postProcessBeforeInitialization....mouse=>com.wxj.bean.Mouse@75f9eccc    初
始化Mouse.....   postProcessAfterInitialization....mouse=>com.wxj.bean.Mouse@75f9eccc
容器创建完成..... 七月 23, 2019 10:57:35 下午
org.springframework.context.annotation.AnnotationConfigApplicationContext doClose  信息:
Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@311d617d:
startup date [Tue Jul 23 22:57:35 CST 2019]; root of context hierarchy  销毁
Mouse..... Dog销毁.....@PreDestroy..    销毁Cat.....
```

通过观察结果可以看到,在每个bean初始化之前,都会先执行postProcessBeforeInitialization方法,而在初始化完成之后,会执行postProcessAfterInitialization方法.

注意:如果后置器处理中返回的为null,那么容器不会把使用@Bean注解的bean,加载到容器中

Spring注解驱动第九讲--属性赋值

在使用只用Spring注解开发的时候,可以使用@Value和@PropertySource注解进行给对象的属性进行赋值.

1,创建一个简单的实体类

```
public class Person {     /**         * @Value注解中的值可以有以下几种        * 1,直接将值写
在Value的属性当中(String,int,boolean.....)例如name属性        * 2,写spEL表达式即#{}的形式,例
如age属性        * 3,在主配置类中通过@PropertySource注解加载配置文件,然后通过${}的形式取配置
文件中的值        */        @Value("张三")        private String name ;       @Value("#{31-
9}")      private int age ;      @Value("${person.nikname}")       private String nikname ;
public String getNikname() {             return nikname;         }     public void
setNikname(String nikname) {             this.nikname = nikname;       }     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;         }     public Person(String
name, int age) {         super();         this.name = name;           this.age = age;
```

```
}        public Person() {          super();     }     @Override        public String toString() {             return "Person [name=" + name + ", age=" + age + ", nikname=" + nikname + "]";       }      }
```

2,在类目录下添加配置文件person.properties

```
person.nikname=小张
```

3,创建测试类

```
    @Test      public void test6() {          AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext(MainPropertiesConfig.class);
Object object = applicationContext.getBean("person") ;          System.out.println(object);
}
```

运行结果: Person [name=张三, age=22, nikname=小张]

Spring注解驱动第十讲--@Autowired使用

概念理解:

自动装配:Spring利用依赖注入（DI），完成对IOC容器中中各个组件的依赖关系赋值;

一.可以利用@Autowired注解实现自动注入,这里省去了一些简单bean的创建,示例代码如下:

BookService.java

```
@Service   public class BookService {      @Autowired       private BookDao bookdao;
@Override       public String toString() {            return "BookService [bookdao=" + bookdao + "]";       }}
```

BookDao.java

```
@Repository      public class BookDao {      }
```

测试方法:

```
@Test     public void test02() {          AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext(MainConfigAutoware.class);
BookService bookService = applicationContext.getBean(BookService.class) ;
System.out.println(bookService);//打印出BookService中的BookDao对象                BookDao bookDao = applicationContext.getBean(BookDao.class);
System.out.println(bookDao);//打印出容器中的BookDao对象


          applicationContext.close(); //关闭容器 }
```

运行结果

```
BookService [bookdao=com.dao.BookDao@75f9eccc]     com.dao.BookDao@75f9eccc
```

从上面的运行结果可以看出:容器中的bean和自动注入到BookService中的bean是统一个BookDao@75f9eccc.@AutoWired注解的工作原理就是,如果某个类中引用容器中的bean,可以在该类的该属性上加@Autowired注解即可.

那么@AutoWired默认优先按照类型去容器中找对应的组件.

此种情况是容器中只有一个bean对象的时候,如果找到多个相同类型的组件,那么Spring将会抛出一个错误

```
expected single matching bean but found 2: bookDao,bookDao2
```

对于这种找到多个Bean对象的时候,可以使用@Qualifier("指定的id名字"),来指定具体使用哪个bean实例代码如下:

```
@Service   public class BookService {      @Qualifier("bookDao")//指定加载"bookDao"
@Autowired      private BookDao bookdao;          @Override       public String toString() {
return "BookService [bookdao=" + bookdao + "]";       }}
```

此时,错误将会消失;容器中会加载bookDao对象;

另一种解决办法就是使用@Primary注解,该注解加载配置类的@Bean注解下,意为当存在两个相同类型的bean时,取被@Primary标注的bean对象

配置类代码如下:

```
@Configuration  @ComponentScan({"com.wxj.service","com.wxj.dao"}) public class MainConfigAutoware {      @Primary          @Bean("bookDao2")      public BookDao bookDao() {  BookDao bookDao = new BookDao() ;              bookDao.setLable("2");           return bookDao ;      } }
```

更改BookDao类

```
@Repository      public class BookDao {          private String lable = "1";      public String getLable() {           return lable;      }     public void setLable(String lable) {  this.lable = lable;       }     @Override       public String toString() {           return "BookDao [lable=" + lable + "]";     } }
```

运行测试结果:

```
BookService [bookdao=BookDao [lable=2]]
```

容器中加载的是被@Primary标注的bean.

注意:如果@Qualifier和@Primary注解同时使用,首选@Qualifier指定的bean;

还有一种情况就是,容器中没有该类型的bean,运行测试类将会报错:

expected at least 1 bean which qualifies as autowire candidate. Dependency annotations: {@org.springframework.beans.factory.annotation.Autowired(required=true)}

根据Spring容器的提示可以看到,类中若使用@AutoWired注解实现自动装配,那么在容器中就至少有一个该类型的bean.那么若何解决该问题呢?可以通过指定@Autowired的required属性来说明该对象有就加载,没有就不加载

@Autowired(required=true):默认的情况,说明容器中必须存在被@Autowired注解的bean,

@Autowired(required=false):说明容器中有指定的bean就加载,没有就不加载.

扩展:@Autowired还可以加在其他位置;

1,方法上:Spring容器创建当前对象,就会调用方法完成赋值.方法使用的参数就是从ioc容器中进行获取

2,有参构造器:在容器创建bean对象时,也会从容器中进行选取.如果容器中只有一个有残构造器,注解还可以省略掉.

3,参数的位置:也是从容器中获取,@Bean标注的对象创建对象时方法的参数值也是从容器中获取.

二,@Resource(JSR250):可以和@Autowired一样实现自动装配.默认是按照组件的id进行装备,不支持@Primary和required属性.也不能和@Qualifier组合使用

三,@Inject(JSR330):需要导入javax.inject的包,和@Autowired一样的使用方法但是没有任何属性

扩展,以上注解的工作原理主要是依赖AutowiredAnnotationBeanPostProcessor类进行实现

Spring注解驱动第十一讲--引用Spring底层组件

在日常开发过程中,自定义组件想要使用Spring容器底层的一些组件.那么自定义组件实现***Aware即可;

在创建对象的时候, 会调用接口规定的方法注入相关组件;例如:

实现ApplicationContextAware接口,可以在自定义组件中获得Spring的ioc容器,代码如下:

```
public class Color implements ApplicationContextAware,BeanNameAware,EmbeddedValueResolverAware{          private ApplicationContext applicationContext;     public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {  //实现ApplicationContextAware接口之后,可以使用本方法将Spring容器加再进来,保存起来使用.  this.applicationContext=applicationContext ;          }     public void setBeanName(String name) {           //实现BeanNameAware接口之后,该方法可以获得到当前bean在ioc容器中的名字;
```

```
System.out.println("当前bean在容器中的名字为:" + name);          }          public void
setEmbeddedValueResolver(StringValueResolver resolver) {             // 实现
EmbeddedValueResolverAware接口之后,可以获取到String类型的值解析器
System.out.println("你好${os.name},我是#{90*20}");                    }      }
```

运行测试类,运行结果如下:

```
当前bean在容器中的名字为:color    你好Mac OS X,我是1800
```

扩展:如果想注入其他的底层组件,实现***Aware接口即可.

Spring注解开发第十二讲--@Profile注解讲解

@Profile:

Spring为我们提供的可以根据当前环境，动态的激活和切换一系列组件的功能;

我们以数据源为例,例如我们想在开发环境使用A数据源,测试环境使用B数据源,上线以后环境使用C数据源.

那么使用@Profile注解可以帮我们实现这个需求.

首先编写配置文件将我们的数据库配置添加到配置文件中,代码如下;

dbconfig.properties

```
db.user=root      db.password=123456      db.driverClass=com.mysql.jdbc.Driver
```

创建主配置类,代码如下:

```
@PropertySource("classpath:/dbconfig.properties") @Configuration   public class
MainConfigOfProfile implements EmbeddedValueResolverAware{             @Value("${db.user}")
private String user;            private StringValueResolver valueResolver;
private String  driverClass;                    @Bean//任何环境下都能加载         public
Yellow yellow(){         return new Yellow();          }          @Profile("test")//测试环境
才加载          @Bean("testDataSource")      public DataSource
dataSourceTest(@Value("${db.password}")String pwd) throws Exception{
ComboPooledDataSource dataSource = new ComboPooledDataSource();
dataSource.setUser(user);           dataSource.setPassword(pwd);
dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/test");
dataSource.setDriverClass(driverClass);             return dataSource;      }
@Profile("dev")//开发环境标识         @Bean("devDataSource")      public DataSource
dataSourceDev(@Value("${db.password}")String pwd) throws Exception{
ComboPooledDataSource dataSource = new ComboPooledDataSource();
dataSource.setUser(user);           dataSource.setPassword(pwd);
dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/ssm_crud");
dataSource.setDriverClass(driverClass);             return dataSource;      }
@Profile("prod")//生产环境标识         @Bean("prodDataSource")      public DataSource
dataSourceProd(@Value("${db.password}")String pwd) throws Exception{
ComboPooledDataSource dataSource = new ComboPooledDataSource();
dataSource.setUser(user);           dataSource.setPassword(pwd);
dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/scw_0515");
dataSource.setDriverClass(driverClass);             return dataSource;      }     @Override
public void setEmbeddedValueResolver(StringValueResolver resolver) {            // TODO
Auto-generated method stub          this.valueResolver = resolver;          driverClass =
valueResolver.resolveStringValue("${db.driverClass}");      } }
```

@Profile：指定组件在哪个环境的情况下才能被注册到容器中，不指定，任何环境下都能注册这个组件

1）、加了环境标识的bean，只有这个环境被激活的时候才能注册到容器中。默认是default环境

2）、写在配置类上，只有是指定的环境的时候，整个配置类里面的所有配置才能开始生效

3）、没有标注环境标识的bean在，任何环境下都是加载的;

运行测试类并制定环境;切换环境的时候 介绍了两种方法,代码如下:

```
public class IOCTest_Profile {            //1、使用命令行动态参数: 在虚拟机参数位置加载 -
Dspring.profiles.active=test          //2、代码的方式激活某种环境;          @Test
public void test01(){         AnnotationConfigApplicationContext applicationContext =
new AnnotationConfigApplicationContext();          //1、创建一个applicationContext
//2、设置需要激活的环境
applicationContext.getEnvironment().setActiveProfiles("dev");             //3、注册主配置类
applicationContext.register(MainConfigOfProfile.class);             //4、启动刷新容器
applicationContext.refresh();                                    String[] namesForType =
applicationContext.getBeanNamesForType(DataSource.class);            for (String string :
namesForType) {              System.out.println(string);             }
Yellow bean = applicationContext.getBean(Yellow.class);
System.out.println(bean);          applicationContext.close();       }}
```

注意:在使用第二种方式的时候,不能使用获取容器的有参构造器