使用spring配置类代替xml配置文件注册bean类

spring配置类，即在类上加@Configuration注解，使用这种配置类来注册bean，效果与xml文件是完全一样的，只是创建springIOC容器的方式不同：

```
//通过xml文件创建springIOC容器
ApplicationContext applicationContext = new ClassPathXmlApplicationContext("/spring-beans.xml");    //通过配置类创建springIOC容器    ApplicationContext applicationContext = new AnnotationConfigApplicationContext(BeanConfig.class);
```

0. 传统的xml配置文件方式注册bean类

单个注册

```
<!--单实例模式-->    <bean id="person" class="cn.monolog.entity.Person" scope="singleton" />    <!--多实例模式-->    <bean id="book" class="cn.monolog.entity.Book" scope="prototype" />
```

扫描批量注册，只能注册加了组件注解(@Repository、@Service、@Controller、@Component)的类

```
<!--通过扫描批量注册加了组件注解的bean-->    <context:component-scan base-package="cn.monolog.entity" />
```

1. 使用配置类单个注册

①在配置类上加@Configuration注解

②在方法上加@Bean注解，bean的id默认为方法名，如果需要自定义id，则在该注解上加value属性

③如果需要指定bean的作用域，则还要在方法上加@Scope注解，如果不加该注解，则默认为单例模式

该注解的value属性有如下几种常用取值(均为字符串格式)

单例模式(singleton)：创建IOC容器时即创建bean对象，以后每次取值都是取的这个bean对象

原型模式(protortype)：创建IOC容器时不创建bean对象，以后每次取值时再分别创建

另外还有request、session、global session

④对于单例模式，还可以加@Lazy注解，即懒加载，表示在创建IOC容器时并不创建bean对象，而是在第一次获取bean对象时才创建，之后再获取bean对象时不再创建，因此仍然是单实例

懒加载的作用：加快SpringIOC容器的启动速度、解决循环依赖的问题

```
/**    * springIOC配置容器类        * 用于代替spring-beans.xml，生成bean对象    */    @Configuration    public class BeanConfig {        //生成Person实例        @Bean(value = "person")        public Person person() {            String name = "张三";            int age = 29;            return new Person(name, age);        }    /**         * 使用单实例模式生成Book实例         * 这时，是在springIOC容器启动时就创建了bean实例，然后每次获取实例时，直接从容器中拿         */        @Bean(value = "singleBook")        @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)        public Book singleBook() {            String
```

```
name = "hello world";          double price = 29.00D;          return new Book(name,
price);        }     /**          * 使用多实例模式生成Book实例          * 这时，启动
springIOC容器时并未创建bean实例，而是每次获取bean实例时再调用该方法去新创建        */
@Bean(value = "multipleBook")        @Scope(value =
ConfigurableBeanFactory.SCOPE_PROTOTYPE)        public Book multipleBook() {          String
name = "hello world";          double price = 29.00D;          return new Book(name,
price);        }   /**      * 使用懒加载模式生成bean实例          * 懒加载只对单实例模式有
效          * 本来，单实例模式，是在启动springIOC容器时创建bean实例       * 使用懒加载后，在启
动springIOC容器时并不创建bean实例，而是在首次获取bean时才创建          */
@Bean(value = "person")        @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
@Lazy     public Person person() {          return new Person("刘能", 53);     }      }
```

2. 使用配置类扫描批量注册，只能注册加了组件注解(@Repository、@Service、@Controller、@Component)的类

①在配置类上加@Configuration注解

②在配置类上加@ComponentScan注解，并在其basePackages属性(字符串数组类型)中写明要扫描的包，可以写1个或多个包，如果想排除某些类，可以写在excludeFilter属性中

```
/**    * 通过扫描批量注册加了组件注解的bean    * 并排除Dog类    */   @Configuration
@ComponentScan(basePackages = {"cn.monolog.entity", "cn.monolog.service",
"cn.monolog.dao"},
excludeFilters = {@ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE, classes =
{Dog.class})})   public class BeanConfig {   }
```

3. 按条件注册

①在配置类上加@Configuration注解

②在方法上加@Bean注解，bean的id默认为方法名，如果需要自定义id，则在该注解上加value属性

③在方法上加@Conditional注解，该注解的参数是字节码数组，什么样的类的字节码呢？必须是实现了Condition接口的。使用时，需要自定义1个或多个Condition的实现类，并在其实现方法中定义生效条件，当满足生效条件时，才会去注册该bean类

```
import org.springframework.context.annotation.Condition;     /**    * springIOC条件注册中的
条件类      */   public class JuventusCondition implements Condition {       @Override
public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
//获取bean注册器          BeanDefinitionRegistry registry = context.getRegistry();
//判断该IOC容器中是否含有id为juventus的组件，如果是，则生效          if
(registry.containsBeanDefinition("juventus")) {              return true;          }
return false;        } }
```

```
/**    * 按条件注册bean类    */   @Configuration   public class BeanConfig {          /**
* 按条件生成名字为ronaldo的Person对象       * 条件为写在JuventusCondition类中      */
@Bean(value = "ronaldo")        @Scope(value = SCOPE_SINGLETON)
@Conditional({JuventusCondition.class})          public Person ronaldo() {          return
new Person("ronaldo", 34);      } }
```

4. 使用导入方式注册

①在配置类上加@Configuration注解

②在配置类上加@Import注解，该注解的value属性为字节码数组，可以接收以下三种类的字节码

要注册的类；

实现了ImportSelector接口的自定义类，其实现方法的返回值为要注册的类的全限定名数组；

实现了ImportBeanDefinitionRegistrar的自定义类，在其实现方法中，使用BeanDefinitionRegistry手动注册，前面两种方式注册的bean的id只能是类的全限定名，只有这种方式可以自定义bean的id；

```
import org.springframework.context.annotation.ImportSelector;  /**  * 使用import注解注册bean的选择器  * 注意该方法可以返回空数组，但不能返回null，因为在后续源码中会获取返回值的长度  * 而且这里只能用类的全限定名  */  public class ColorImportSelector implements ImportSelector {  @Override  public String[] selectImports(AnnotationMetadata importingClassMetadata) {  String[] results = {"cn.monolog.entity.Green", "cn.monolog.entity.Pink"};  return results;  } }
```

```
import org.springframework.context.annotation.ImportBeanDefinitionRegistrar; /**  * 使用import注解注册bean时，自定义bean选择器 * 在该类的实现方法中，使用bean注册器手动注册  * 注意，如果多次为同一个id注册bean，后面的会覆盖前面的  */  public class ColorDefinitionRegistrar implements ImportBeanDefinitionRegistrar {  @Override  public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {  //判断该容器中是否已经存在id为yellow的组件，如果不存在，则手动注册  String id = "yellow";  if (!registry.containsBeanDefinition(id)) {  //bean的id  String beanName = "yellow";  //bean的类型  BeanDefinition beanDefinition = new RootBeanDefinition(Yellow.class);  registry.registerBeanDefinition(beanName, beanDefinition);  }  } }
```

```
/**  * springIOC容器配置类  * 使用import注解注册bean  * 该注解可以接收三种参数  * 1.要注册的类的字节码  * 2.ImportSelector的自定义实现类的字节码，在其实现方法中，将要注册的类的全限定名写入返回数组  * 3.ImportBeanDefinitionRegistrar的自定义实现类的字节码，在其实现方法中，使用bean注册器手动注册  * 其中，前两种方式注册的bean的id只能是该类的全限定名  * 第三种方式注册的bean类可以自定义id  */  @Configuration  @Import({Red.class, ColorImportSelector.class, ColorDefinitionRegistrar.class})  public class BeanConfig {  }
```

5. 使用工厂模式注册

工厂模式与第1条(单个注册)的方法完全相同，唯一的特别之处在于，注册的是一个"工厂类"，即实现了FactoryBean<T>接口的自定义类，这种类的特点是，在IOC容器中直接按id获取的并不是它本身的对象，而是它内部注册的bean类的对象。要想获取它本身的对象，要在id前面拼接"&"符号。

FactoryBean<T>接口有三个方法：

getObject——用于注册真正的bean类

getObjectType——用于获取bean类的类型，即泛型T

isSingleton——注册的bean类是否为单例模式

```
import org.springframework.beans.factory.FactoryBean;  /**  * 工厂类  */  public class ColorFactoryBean implements FactoryBean<Color> {  //注册bean  @Override  public Color getObject() throws Exception {  return new Color("#FFF");  }  //获取bean的类型  @Override  public Class<?> getObjectType() {  return Color.class;  }  //是否单实例模式  @Override  public boolean isSingleton() {  return true;  } }
```

bean是由IOC容器初始化、装配以及管理的对象，除此之外，bean就跟普通的对象一样。然而bean的定义以及bean之间的相互依赖关系是通过配置元数据来描述。

创建一个bean的定义，其实就是什么时候创建一个bean，在spring框架中支持五种作用域，分别如下：

- Singleton：在Spring IOC 容器仅存在一个Bean实例，Bean以单例方式存在，这个是默认值。
- prototype：每次从容器调用bean时，都会返回一个新的实例，也就是每次调用getBean()时都会实例化一个新的bean。
- request：每次HTTP请求都会创建一个新的Bean，该作用于仅适用于web环境
- session：每个HTTP Session共享一个Bean，不同的Session使用不同的Bean，同样只适用于web环境。
- Global Session：一般作用于Portlet应用环境，只作用于Web环境。

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