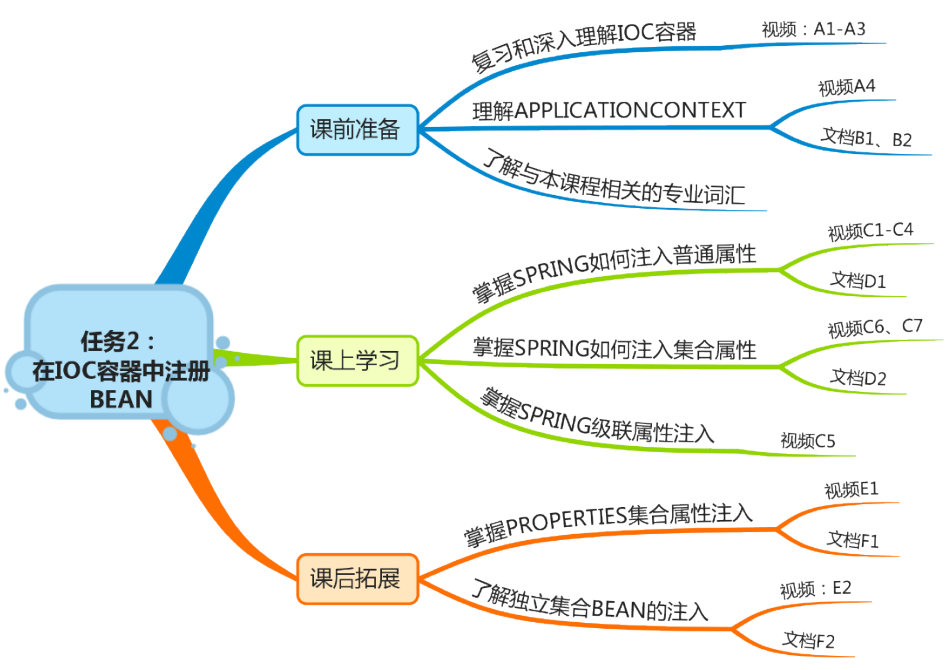
单元 3.3.2-在 IOC 容器中注册和配置 bean

课程导入

同学们，在本单元的学习中我们将接触一个全新的概念，bean，这个 bean 不是 JSP 中所用过的 JavaBean，配置的在 IOC 容器中的 bean。因此要想学好本单元，同学们应该先深入理解 IOC 容器的概念。在本单元的学习中，同学们要着重学习如何配置一个 bean，bean 配置好了以后，如何注入属性，属性注入有很多中方式，细节内容比较多，请同学们注意理解。

知识目标	能力目标	素质目标
1. 掌握 spring 的普通属性注入、依赖对象的注入方式 2. 掌握讲解属性编辑器的原理及作用和编写方法 3. 学会将公共的注入定义描述出来的方法 4．掌握 spring Bean 的作用域	1.能够在 IOC 容器中定义一个 bean 2.能够对 IOC 容器中的 bean 属性进行注入	1.培养学生的团队意识和团队协作精神，锻炼学生的沟通交流能力; 2.通过项目教学，让学生真切的体验项目分析、设计、管理及实施的全过程;
学习任务	**重点难点**	**突破方法**
在 Spring 的 IOC 容器中注册 bean，并注入 bean 的属性。	1.依赖对象的注入方式 2.spring Bean 的作用域	采用翻转课堂、项目导入的教学模式，进行分组讨论、演示动画原理。

学习导航



任务1

在 IOC 容器中注册 bean

Spring IoC 容器完全由实际编写的配置元数据的格式解耦。有下面三个重要的方法把配置元数据提供给 Spring 容器：

基于 XML 的配置文件

基于注解的配置

基于 Java 的配置

提示：对于基于 XML 的配置，Spring 2.0 以后使用 Schema 的格式，使得不同类型的配置拥有了自己的命名空间，是配置文件更具扩展性。

你已经看到了如何把基于 XML 的配置元数据提供给容器，但是让我们看看另一个基于 XML 配置文件的例子，这个配置文件中有不同的 bean 定义，包括延迟初始化，初始化方法和销毁方法的：

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <!-- A simple bean definition -->
   <bean id="..." class="...">
       <!-- collaborators and configuration for this bean go here -->
   </bean>

   <!-- A bean definition with lazy init set on -->
   <bean id="..." class="..." lazy-init="true">
       <!-- collaborators and configuration for this bean go here -->
   </bean>

   <!-- A bean definition with initialization method -->
   <bean id="..." class="..." init-method="...">
       <!-- collaborators and configuration for this bean go here -->
   </bean>
```

```
    <!-- A bean definition with destruction method -->
    <bean id="..." class="..." destroy-method="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions go here -->

</beans>
```

在上述示例中：

①xmlns="http://www.springframework.org/schema/beans"，默认命名空间：它没有空间名，用于 Spring Bean 的定义；

②xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"，xsi 命名空间：这个命名空间用于为每个文档中命名空间指定相应的 Schema 样式文件，是标准组织定义的标准命名空间。

你可以查看 **Spring Hello World 实例** 来详细理解如何定义，配置和创建 Spring Beans。关于基于注解的配置将在一个单独的章节中进行讨论。刻意把它保留在一个单独的章节，是因为我想让你在开始使用注解和 Spring 依赖注入编程之前，能掌握一些其他重要的 Spring 概念。

任务 2

XML 方式对 bean 的配置

XmlBeanFactory 为例，最简单的取 bean 方式是：

Java 代码

```
1. BeanFactory factory = new XmlBeanFactory(new FileSystemResource("D:\\workspace\\JavaApplication2\\src\\javaapplication2\\spring\\beans.xml"));
2.             Car obj = (Car)factory.getBean("car");
```

Bean 的配置文件内容也很简单：

Xml 代码

```
1. <bean id="vehicle"   abstract="true">
2.     <property name="wheel" value="Four wheeler"/>
3. </bean>
```

```
4.  <bean id="car" class="javaapplication2.spring.Car" parent="vehicle">
5.      <property name="dicky" value="Flat dicky"/>
6.  </bean>
```

先看起始点，载入先走 AbstractBeanFactory

Java 代码

```
1.  public Object getBean(String name) throws BeansException {
2.      return doGetBean(name, null, null, false);
3.  }
```

doGetBean 方法中：

Java 代码

```
1.  // Create bean instance.
2.  if (mbd.isSingleton()) {
3.           //传入一个内联类 ObjectFactory 并实现了 getObject 方法。
4.           sharedInstance = getSingleton(beanName, new ObjectFactory() {
5.      public Object getObject() throws BeansException {
6.      try {
7.          return createBean(beanName, mbd, args);
8.      }
9.      catch (BeansException ex) {
10.     // Explicitly remove instance from singleton cache: It might have been put there
11.     // eagerly by the creation process, to allow for circular reference resolution.
12.     // Also remove any beans that received a temporary reference to the bean.
13.     destroySingleton(beanName); //有异常则销毁 bean
14.             throw ex;
15.     }
16. }
17. });
18.     bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
19.         //此处开始实例化 bean
```

}

通过 new ObjectFactory()的回调方法，回调当前类继承的 createBean 方法，该方法在父类 AbstractAutowireCapableBeanFactory 中：

AbstractAutowireCapableBeanFactory->

Java 代码

```
1.  protected Object createBean(final String beanName, final RootBeanDefinition mbd, final
    Object[] args)
2.              throws BeanCreationException {
3.
```

```
4.          // Make sure bean class is actually resolved at this point.
5.          resolveBeanClass(mbd, beanName);  //载入该 bean 的 class，并放置到 mbd 里面,bean 的
    生成不在这里。
6.
7.          // Prepare method overrides.
8.          try {
9.              mbd.prepareMethodOverrides();
10.         }
11.         catch (BeanDefinitionValidationException ex) {
12.             throw new BeanDefinitionStoreException(mbd.getResourceDescription(),
13.                     beanName, "Validation of method overrides failed", ex);
14.         }
15.
16.         try {
17.             // Give BeanPostProcessors a chance to return a proxy instead of the target
     bean instance.
18.             Object bean = resolveBeforeInstantiation(beanName, mbd);  //尝试获取一个 prox
    y，普通 bean 这里一般是空的返回
19.
20.             if (bean != null) {
21.                 return bean;
22.             }
23.         }
24.         catch (Throwable ex) {
25.             throw new BeanCreationException(mbd.getResourceDescription(), beanName,
26.                     "BeanPostProcessor before instantiation of bean failed", ex);
27.         }
28.
29.         Object beanInstance = doCreateBean(beanName, mbd, args);  //开始 create bean 的实
    例，mbd 中包括了需要的 class
30.         if (logger.isDebugEnabled()) {
31.             logger.debug("Finished creating instance of bean '" + beanName + "'");
32.         }
33.         return beanInstance;
34.     }
```

进入 **AbstractBeanFactory 中的 protected Class resolveBeanClass 方法：**

Java 代码

```
1.  try {
2.      if (mbd.hasBeanClass()) {
3.          return mbd.getBeanClass();
4.      }
5.      if (System.getSecurityManager() != null) {
6.          return AccessController.doPrivileged(new PrivilegedExceptionAction<Class>() {
7.              public Class run() throws Exception {
```

```
                return doResolveBeanClass(mbd, typesToMatch);
            }
        }, getAccessControlContext());
    }
    else {
        return doResolveBeanClass(mbd, typesToMatch);   <---还要继续进去看生成方法。
    }
}
```

转入 doResolveBeanClass：

Java 代码

```
private Class doResolveBeanClass(RootBeanDefinition mbd, Class... typesToMatch) throws ClassNotFoundException {
    if (!ObjectUtils.isEmpty(typesToMatch)) {
        ClassLoader tempClassLoader = getTempClassLoader();
        if (tempClassLoader != null) {
            if (tempClassLoader instanceof DecoratingClassLoader) {
                DecoratingClassLoader dcl = (DecoratingClassLoader) tempClassLoader;
            for (Class<?> typeToMatch : typesToMatch) {
                dcl.excludeClass(typeToMatch.getName())
            }
        }
        String className = mbd.getBeanClassName();
        return (className != null ? ClassUtils.forName(className, tempClassLoader) : null); //通过自己的 ClassUtils 的 forName 方法来实例化 class
        }
    }
    return mbd.resolveBeanClass(getBeanClassLoader());  <----这里传入了 bean 的 classloader，下面继续看这里
}
```

AbstractBeanDefinition->resolveBeanClass

Java 代码

```
public Class resolveBeanClass(ClassLoader classLoader) throws ClassNotFoundException {

        String className = getBeanClassName();
        if (className == null) {
            return null;
        }
        Class resolvedClass = ClassUtils.forName(className, classLoader);//classloader 传入后，仍然是用 forName 方法加载 class
        this.beanClass = resolvedClass;
        return resolvedClass;
```

```
9.      }
```

再来看 forName 做了些什么

ClassUtils ->

Java 代码

```
1.  ClassLoader classLoaderToUse = classLoader;
2.  if (classLoaderToUse == null) {
3.      classLoaderToUse = getDefaultClassLoader();
4.  }
5.  try {
6.      return classLoaderToUse.loadClass(name); //也比较简单，直接调用 loadClass 方法加载
7.  }
```

最终将 class load 进来。

Bean 实例化过程：

AbstractAutowireCapableBeanFactory->createBeanInstance

Java 代码

```
1.  protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, Objec
    t[] args)
2.      // Need to determine the constructor...
3.      //提取构造函数，如果没有就是空
4.      Constructor[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanNa
    me);
5.      if (ctors != null ||mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_CO
    NSTRUCTOR ||mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args))  {
6.              return autowireConstructor(beanName, mbd, ctors, args);
7.      }
8.
9.      // No special handling: simply use no-arg constructor.
10.     return instantiateBean(beanName, mbd); //这里实例化
```

进入

AbstractAutowireCapableBeanFactory->instantiateBean

Java 代码

```
1.  protected BeanWrapper instantiateBean(final String beanName, final RootBeanDefinition m
    bd)
2.  ...这里省略没用的
3.      else {
4.          beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);//
    下面看这里的实例化
5.      }
6.
7.      BeanWrapper bw = new BeanWrapperImpl(beanInstance);   //返回一个包装类对象
8.      initBeanWrapper(bw);
9.      return bw;
```

SimpleInstantiationStrategy->instantiate

Java 代码

```
1.  public Object instantiate(RootBeanDefinition beanDefinition, String beanName, BeanFacto
    ry owner)
2.      synchronized (beanDefinition.constructorArgumentLock) {
3.              constructorToUse = (Constructor<?>) beanDefinition.resolvedConstructorOrFac
    toryMethod;
4.          ...
5.      }
6.      return BeanUtils.instantiateClass(constructorToUse);     //BeanUtils 来初始化实例 ，
    给出了实例化需要的构造函数
```

再来看 **BeanUtils** 的实例化方法，比较简单，直接用反射的构造函数来 **newInstance**。

BeanUtils->

Java 代码

```
1.  public static <T> T instantiateClass(Constructor<T> ctor, Object... args) throws BeanIn
    stantiationException {
2.      try {
3.          ReflectionUtils.makeAccessible(ctor);
4.          return ctor.newInstance(args);
5.      }
```

AbstractAutowireCapableBeanFactory->

Java 代码

```
1.  Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[]
     args)
2.
3.      // Initialize the bean instance.
4.          Object exposedObject = bean;
5.          try {
6.              populateBean(beanName, mbd, instanceWrapper);
7.              if (exposedObject != null) {
8.                  exposedObject = initializeBean(beanName, exposedObject, mbd);
9.              }
10.         }
11.     return exposedObject;       //返回给 AbstractBeanFactory
```

任务 3	**回顾 spring 注册 bean 的 5 中方法**
	第一种：在初始化时保存 ApplicationContext 对象

```
1 ApplicationContext ac = new
  FileSystemXmlApplicationContext("applicationContext.xml");
2 ac.getBean("beanId");
```

说明：这种方式适用于采用 Spring 框架的独立应用程序，需要程序通过配置文件手工初始化 Spring。

第二种：通过 Spring 提供的工具类获取 ApplicationContext 对象

```
1 import org.springframework.web.context.support.WebApplicationContextUtils;
  ApplicationContext ac1 =
2 WebApplicationContextUtils.getRequiredWebApplicationContext(ServletContext sc);
3 ApplicationContext ac2 =
4 WebApplicationContextUtils.getWebApplicationContext(ServletContext sc);
  ac1.getBean("beanId");
5 ac2.getBean("beanId");
```

说明：

1、这两种方式适合于采用 Spring 框架的 B/S 系统，通过 ServletContext 对象获取 ApplicationContext 对象，然后在通过它获取需要的类实例；

2、第一种方式在获取失败时抛出异常，第二种方式返回 null。

第三种：继承自抽象类 ApplicationObjectSupport

说明：通过抽象类 ApplicationObjectSupport 提供的 getApplicationContext()方法可以方便的获取到 ApplicationContext 实例，进而获取 Spring 容器中的 bean。Spring 初始化时，会通过该抽象类的 setApplicationContext(ApplicationContext context)方法将 ApplicationContext 对象注入。

第四种：继承自抽象类 WebApplicationObjectSupport

说明：和上面方法类似，通过调用 getWebApplicationContext()获取 WebApplicationContext 实例；

第五种：实现接口 ApplicationContextAware

说明：实现该接口的 setApplicationContext(ApplicationContext context)方法，并保存 ApplicationContext 对象。Spring 初始化时，会通过该方法将 ApplicationContext 对象注入。

虽然 Spring 提供了后三种方法可以实现在普通的类中继承或实现相应的类或接口来获取 Spring 的 ApplicationContext 对象，但是在使用时一定要注意继承或实现这些抽象类或接口的普通 java 类一定要在 Spring 的配置文件（即 application-context.xml 文件）中进行配置，否则获取的 ApplicationContext 对象将为 null。

下面通过实现接口 ApplicationContextAware 的方式演示如何获取 Spring 容器中的 bean：

首先自定义一个实现了 ApplicationContextAware 接口的类，实现里面的方法：

```
1  package com.ghj.tool;
2  import org.springframework.beans.BeansException;
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.ApplicationContextAware;
5  public class SpringConfigTool implements ApplicationContextAware {// extends
6  ApplicationObjectSupport{
7  private static ApplicationContext ac = null;
8  private static SpringConfigTool springConfigTool = null;
9  public synchronized static SpringConfigTool init() {
10 if (springConfigTool == null) {
11 springConfigTool = new SpringConfigTool();
}
return springConfigTool;
```

```
12  }
13  public void setApplicationContext(ApplicationContext applicationContext)throws
    BeansException {
14  ac = applicationContext;
15  }
16  public synchronized static Object getBean(String beanName) {
17  return ac.getBean(beanName);
    }
18  }
19
25
26
```

其次在 applicationContext.xml 文件进行配置：

复制代码 代码如下:

```
<bean id="SpringConfigTool" class="com.ghj.tool.SpringConfigTool"/>
```

最后通过如下代码就可以获取到 Spring 容器中相应的 bean 了：

复制代码 代码如下:

```
SpringConfigTool.getBean("beanId");
```

注意一点，在服务器启动 Spring 容器初始化时，不能通过以下方法获取 Spring 容器:

?

```
1  import org.springframework.web.context.ContextLoader;
```

```
2  import org.springframework.web.context.WebApplicationContext;
3  WebApplicationContext wac = ContextLoader.getCurrentWebApplicationContext();
   wac.getBean(beanID);
```