基于xml的Spring应用

1. bean标签的属性

- id和class，Bean的id和全限定名配置
- name，通过name设置Bean的别名，通过别名也可以直接获取到Bean实例
- scope，Bean的作用范围，BeanFactory作为容器时取值singleton和prototype
- lazy-init，Bean的实例化机制，是否延迟加载，BeanFactory作为容器时无效
- init-method，Bean实例化后自动执行的初始化方法，method指定方法名
- detroy-method，Bean实例销毁前的方法，method指定方法名
- autowire，设置自动注入模式，常用的有，按照类型-byType，按照名字-byName
- factory-bean和factory-method，指定哪个工厂，Bean的哪个方法完成创建Bean
- getBean方法传入值是BeanFactory维护beanName-Bean的map中的beanName，默认beanName取id的值，如果id没有设置，beanName取全限定名
- name与id的映射关系，维护在BeanFactory的aliasMap中，name-beanName；如果只设置了name，没有设置id，beanName取name中的第一个值，其他的name作为第一个值的映射
- scope，默认值是singleton；prototype，原型，Spring容器初始化时不会创建Bean实例，调用getBean时才会实例化，每次getBean都会创建一个新的Bean实例，并且不会存储到单例池中（也就是上文中的beanName-Bean的map）
- destroy-method，只有在显示关闭容器时，才会调用
- 除了在bean标签中声明init，还可以在Bean类中实现InitializingBean接口，达到同样的效果（Spring先调用InitializingBean，再调用配置的init方法）

2. Bean的实例化配置

Spring 的实例化主要有以下两种方式（指的时BeanFactory中实例化Bean的方式）：

- 构造方式实例化，底层通过构造方法对Bean进行实例化
- 工厂方式实例化，底层通过调用自定义的工厂方法对Bean进行实例化

> 有参构造，可通过constructor-arg 标签定义，并传输值
>
> 注意：constructor-arg不仅仅可以用于向有参构造方法传输值

工厂方式实例化Bean，分为如下三种：

- 静态工厂方法实例化，自定义方法可以在Bean创建前提供其他业务逻辑操作，或调用第三方jar中的Bean
- 实例工厂方法实例化
- 实现FactoryBean规范延迟实例化，实现接口FactoryBean，存储在单例池中的并不是Bean实例，而是实现FactoryBean的实例，而真正的Bean实例，在getBean后，存储在BeanFactory的factoryBeanObjectCache中；在调用getBean时，才调用getObject将Bean存储在cache中

> 自定义的实例化Bean方法，如果需要传入参数，可以在bean标签中添加 constructor-arg标签传输参数

3. Bean的依赖注入

两种方式。其一，通过Bean的set方法注入，配置方式为

```
<property name="userDao" ref="userDao"/>
<property name="userDao" value="Tom"/>
```

其二，通过构造Bean的方法注入，配置方式为

```
<constructor-arg name="name" ref="userDao"/>
<contructor-arg name="name" value="Tom"/>
```

注入的数据类型有三种，

- 普通数据类型，如String、int、boolean等，通过value指定
- 引用数据类型，通过ref指定
- 集合数据类型，如List、Map、Properties等

集合数据类型的注入，如下：

```
<bean id="userService" class="com.example.service.impl.UserServiceImpl">
    <property name="userDao" ref="userDao"></property>
    <property name="stringList">
        <list>
           <value>aaa</value>
```

```
            <value>bbb</value>
            <value>ccc</value>
          </list>
      </property>
  </bean>
```

另外，Bean还有自动装配的方式。如果，被注入的属性类型时Bean引用的话，那么，可以在bean标签中使用autowire属性去配置自动注入方式，属性值有两个，

- byName，通过属性名称自动装配，即，匹配setXxx与id="xxx"是否一致
- byType，通过Bean的类型从容器中匹配，匹配出多个相同Bean类型时报错

4. Spring的常见默认标签

- beans，一般作为xml配置的根标签，其他标签都是该标签的子标签
- bean，Bean的配置标签
- import，外部资源导入标签
- alias，指定Bean的别名标签，较少使用

通过beans配置不同环境Bean，

```
<beans profile="dev">
   <bean id="userService1" class="com.example.service.impl.UserServiceImpl">
</bean>
</beans>

<beans profile="test">
   <bean id="userDao1" class="com.example.dao.impl.UserDaoImpl"></bean>
</beans>
```

可以使用以下两种方式指定被激活的环境，

- 使用命令行动态参数，虚拟机参数设置 `-Dspring.profiles.active=test`
- 使用代码的方式设置环境变量 `System.setProperty("spring.profiles.active", "test")`

可以给每个模块配置xml文件，在主xml文件中使用import引入其他配置文件，

```
<import resource="classpath:applicationContext-user.xml"/>
<import resource="classpath:applicationContext-orders.xml"/>
```

5. Spring配置非自定义Bean

如果需要将第三方jar中的Bean配置到Spring中，应该如何操作?

需要考虑两个问题，

- 被配置的Bean的实例化方式是什么？无参、有参、静态工厂还是实例工厂？
- 被配置的Bean是否需要注入必要的属性？

例子1：配置DruidDataSource交由Spring管理

观察DruidDataSource，有无参构造，并且需要配置四个参数，分别是驱动名称、Url、用户名和密码，有如下配置

```
<!-- 配置数据源信息 -->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="url" value="jdbc:mysql://192.168.224.100:3306/test"/>
  <property name="username" value="study"/>
  <property name="password" value="123456"/>
</bean>
```

例子2：定义日期字符串交由Spring生成日期对象

代码的方式：

```
String currentTimeStr = "2023-08-27 07:20:00";

DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");
LocalDateTime currentTime = LocalDateTime.parse(currentTimeStr,
dateTimeFormatter);
System.out.println(currentTime);
```

通过Spring配置Bean的方式：

1. 定义dateTimeFormatter的Bean，使用静态工厂的方式

   ```
   <bean name="dateTimeFormatter" class="java.time.format.DateTimeFormatter"
   factory-method="ofPattern">
     <constructor-arg name="pattern" value="yyyy-MM-dd HH:mm:ss"/>
   </bean>
   ```

2. 定义结果currentDateTime，它需要有LocalDateTime的静态方法生成，也是以静态工厂的方式

   ```
   <bean name="currentDateTime" class="java.time.LocalDateTime" factory-
   method="parse">
     <constructor-arg name="text" value="2023-08-27 07:20:00"/>
     <constructor-arg name="formatter" ref="dateTimeFormatter"/>
   </bean>
   ```

例子3：配置MyBatis的SqlSessionFactory交由Spring管理

代码实现：

```
InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory sqlSessionFactory = builder.build(in);
```

通过Spring配置的方式实现：

1. 定义InputStream的Bean，以静态工厂的方式

```
<bean id="in" class="org.apache.ibatis.io.Resources" factory-
method="getResourceAsStream">
  <constructor-arg name="resource" value="mybatis-config.xml"/>
</bean>
```

2. 定义SqlSessionFactoryBuilder的Bean，常规Bean，无参构造的方式

```
<bean id="builder"
class="org.apache.ibatis.session.SqlSessionFactoryBuilder"></bean>
```

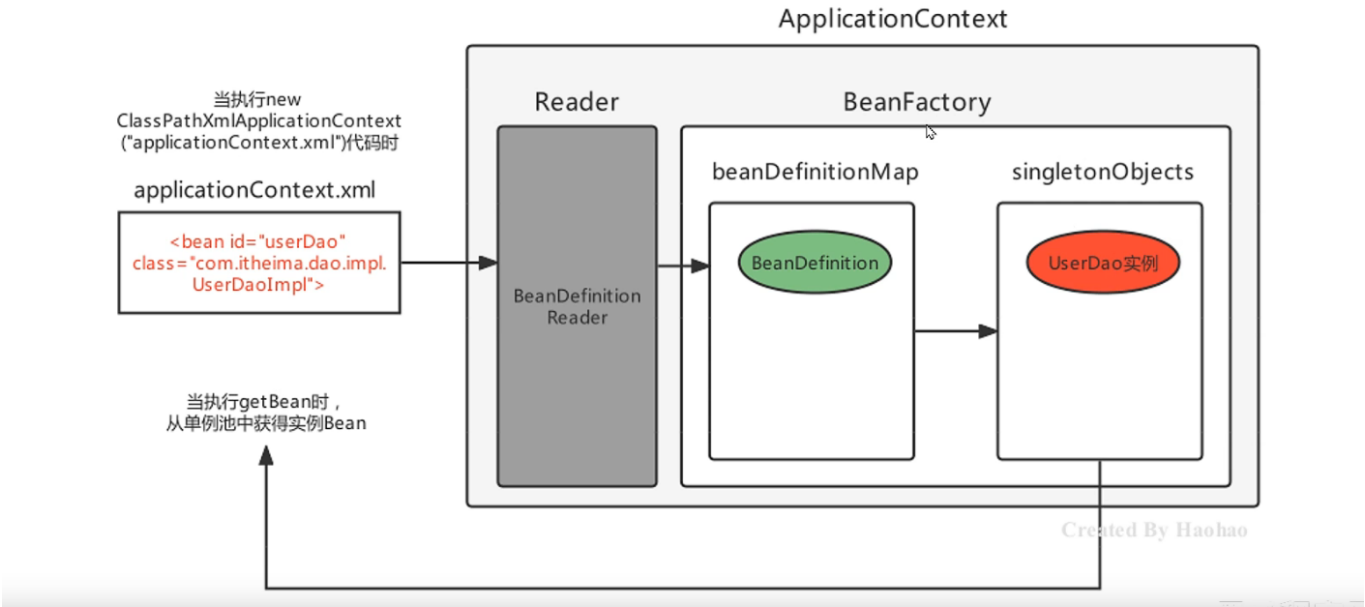
3. 定义结果SqlSessionFactory的Bean，以实例工厂的方式

```
<bean id="sqlSessionFactory" factory-bean="builder" factory-method="build">
  <constructor-arg name="inputStream" ref="in"/>
</bean>
```

6. Bean实例化的基本流程

- Spring容器在进行初始化时，会将xml配置的<bean>的信息封装成一个BeanDefinition对象，所有BeanDefinition存储到一个名为beanDefinitionMap的Map集合中；
- Spring框架对该Map进行遍历，使用反射创建Bean实例对象；
- 创建好的Bean对象存储在一个名为singletonObjects的Map集合中，当调用getBean方法时，从该Map集合中取出Bean实例对象返回。

- Bean 实例化的基本流程



7. Spring的后处理器

Spring的后处理器是Spring对外开发的重要扩展点，允许我们介入到Bean的整个实例化流程中来，以达到动态注册BeanDefinition（就是向beanDefinitionMap中添加BeanDefinition），动态修改BeanDefinition，以及动态修改Bean的作用。Spring主要有两种后处理器：

- BeanFactoryPostProcessor：Bean工厂后处理器，在beanDefinitionMap填充完毕，Bean实例化之前执行；
- BeanPostProcessor：Bean后处理器，一版在Bean实例化之后，填充到单例池singletonObjects之前执行

7.1 BeanFactoryPostProcessor

BeanFactoryPostProcessor是一个接口规范，实现了该接口的类只要交由Spring容器管理的话，那么Spring就会回调该接口的方法，用于对BeanDefinition注册和修改的功能。

定义实现BeanFactoryPostProcessor的类，并注册到Spring容器中，动态定义Bean的示例如下，

```
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {
    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory
beanFactory) throws BeansException {
        System.out.println("beanDefinitionMap填充完毕后回调该方法");
        // 动态注册PersonDao
        BeanDefinition beanDefinition = new RootBeanDefinition();
        beanDefinition.setBeanClassName("com.example.dao.impl.PersonDaoImpl");
        // 强转成DefaultListableBeanFactory
        DefaultListableBeanFactory defaultListableBeanFactory =
(DefaultListableBeanFactory) beanFactory;
        defaultListableBeanFactory.registerBeanDefinition("personDao",
beanDefinition);
    }
}
```

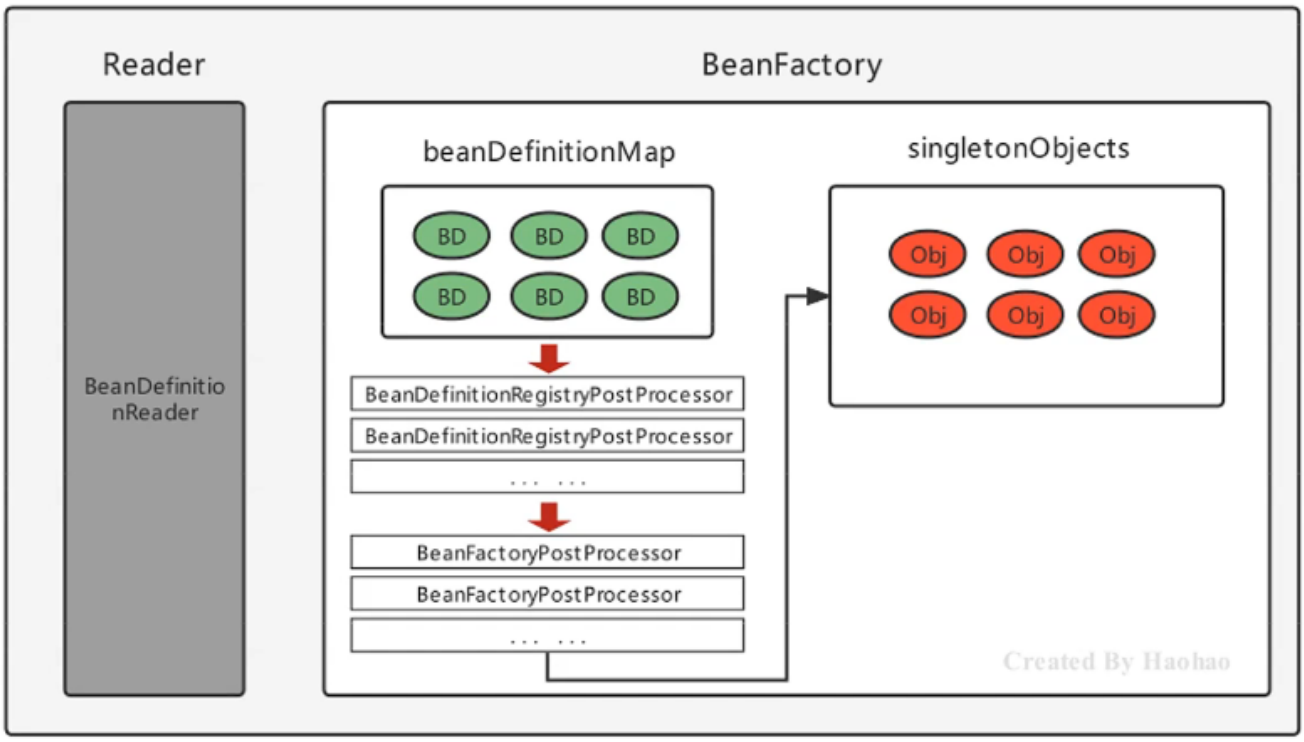
Spring提供了一个BeanFactoryPostProcessor的子接口BeanDefinitionRegistryPostProcessor，专门用于注册BeanDefinition操作，示例如下

```
public class MyBeanDefinitionRegistryPostProcessor implements
BeanDefinitionRegistryPostProcessor {
    @Override
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry
beanDefinitionRegistry) throws BeansException {
        System.out.println("postProcessBeanDefinitionRegistry执行");
        // 注册BeanDefinition
        BeanDefinition beanDefinition = new RootBeanDefinition();
        beanDefinition.setBeanClassName("com.example.dao.impl.PersonDaoImpl");
        beanDefinitionRegistry.registerBeanDefinition("personDao",
beanDefinition);
    }

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory
configurableListableBeanFactory) throws BeansException {
        System.out.println("postProcessBeanFactory执行");
    }
}
```

执行发现，先执行postProcessBeanDefinitionRegistry，再执行postProcessBeanFactory，并且MyBeanDefinitionRegistryPostProcessor和MyBeanFactoryPostProcessor都存在时，先执行前者，即先执行子类方法，再执行父类方法。

执行过程可如下图所示



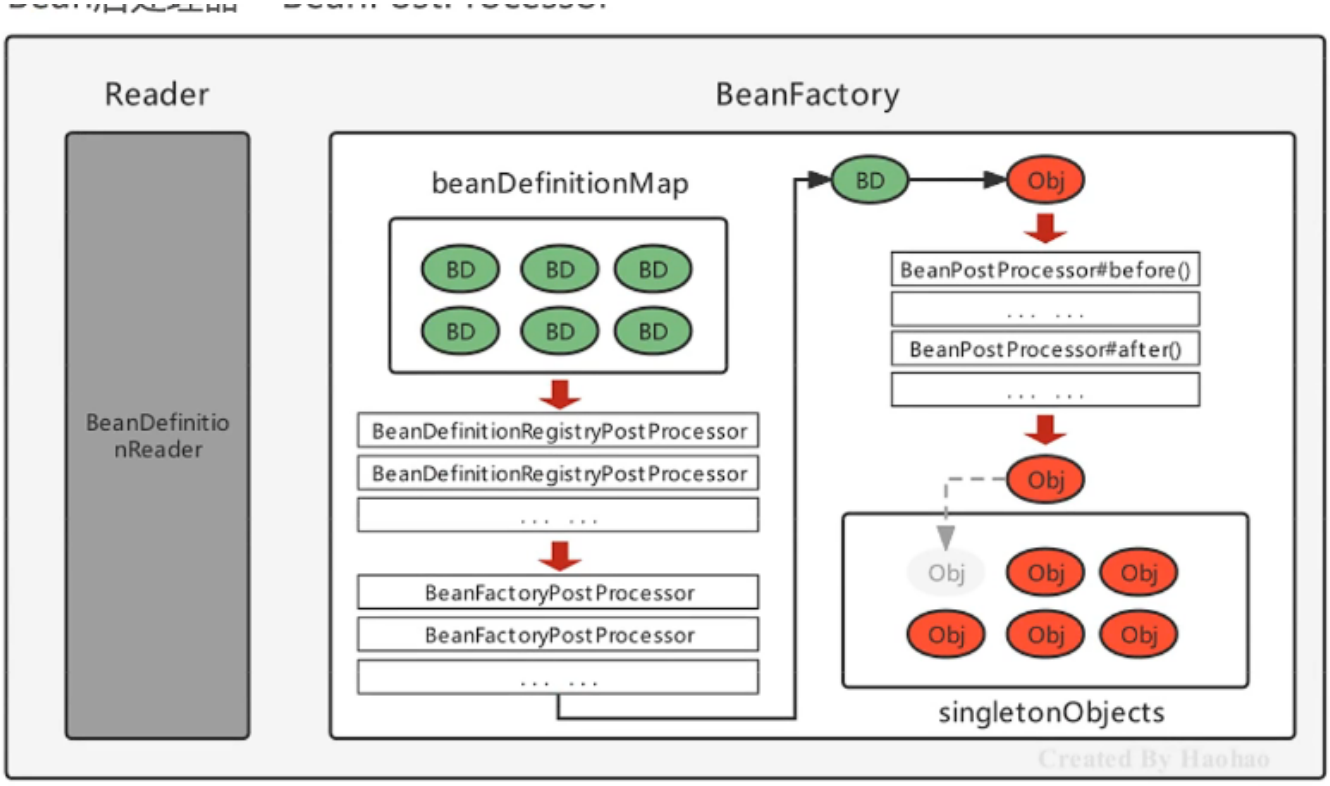
案例：

- 自定义@MyComponent注解，使用在类上；
- 使用包扫描工具BeanClassScanUtils完成指定包的类扫描；
- 自定义BeanFactoryPostProcessor完成注解@MyComponent的解析，解析后最终被Spring管理

7.2 BeanPostProcessor

Bean被实例化后，到最终缓存到名为singletonObjects单例池之前，中间会经过Bean的初始化过程，例如：属性的填充、处事方法init的执行等，其中，有一个对外进行扩展的点BeanPostProcessor，称之为Bean后处理器。与BeanFactoryPostProcessor类似，它是一个接口，实现该接口并被容器管理后，会在流程节点上被Spring自动调用。

- BeanFactoryPostProcessor是对Bean定义的操作，BeanPostProcessor是对Bean的操作
- 定义BeanPostProcessor后发现，每个Bean的实例化都会执行定义的BeanPostProcessor
- 先执行Bean的实例化，也就是调用Bean的构造方法，再执行beforeInitialization，再执行afterInitialization
- 在before和after中间，执行Bean的init-method和接口InitializingBean
- init-method可以自定义init方法，设置在配置的init-method属性中，或者，Bean实现接口InitializingBean，需要注意的是，先执行InitializingBean的方法，再执行init-method



案例：对Bean方法进行执行时间日志增强

- Bean的方法执行之前控制台打印当前时间
- Bean的方法执行之后控制台打印当前时间

分析：

- 对方法进行增强主要就是代理设计模式和包装设计模式
- 由于Bean方法不确定，所以使用动态代理在运行期间执行增强操作
- 在Bean实例创建完毕后，进入到单例池之前，使用Proxy代替真实的目标Bean

8. Spring Bean的生命周期

Spring Bean的生命周期是从Bean实例化之后，即，通过反射创建出对象之后，到Bean成为一个完整对象，最终存储到单例池中，这个过程被称为Spring Bean的生命周期。Spring Bean的生命周期答题分为三个阶段：

- Bean的实例化阶段：Spring框架会取出BeanDefinition的信息进行判断，当前Bean的范围是否是singleton的，是否是延迟加载的，是否是FactoryBean的，最终，将一个普通的singleton的Bean通过反射进行实例化。
- Bean的初始化阶段：Bean创建之后还只是一个“半成品”，还需要对Bean实例的属性进行填充、执行一些Aware接口方法、执行BeanPostProcessor方法、执行Initializing接口的初始化方法、执行自定义初始化init方法等。该阶段是Spring最具技术含量和复杂度的阶段，AOP增强、Spring注解功能以及高频面试题Bean的循环引用问题都是在这个阶段体现的。
- Bean的完成阶段：经过初始化，Bean就成为了一个完整的Spring Bean，被存储到单例池singletonObjects中，完成了整个生命周期。

8.1 Bean实例的属性注入

分以下集中情况：

- 注入普通属性，String、int或存储基本类型的集合，直接通过set方法反射设置
- 注入单向对象引用属性，从容器中getBean后通过set方法反射设置，如果容器中没有，则先创建被注入对象Bean实例（完成整个声明周期）后，再进行注入操作
- 注入双向对象引用属性时，涉及到循环引用问题。

> SpringBean的循环引用问题。Spring提供了**三级缓存**存储**完整Bean实例**和**半成品Bean实例**，用于解决循环引用的问题。
>
> Spring的三级缓存分别是singletonObjects、earlySingletonObjects和singletonFactories。
>
> 1. singletonObjects：这是Spring缓存单例Bean对象的最终缓存，也就是我们通常所说的单例池。当Bean对象创建完成后，Spring会将其放入singletonObjects缓存中，以便后续使用。
> 2. earlySingletonObjects：这是Spring缓存Bean对象的第二级缓存，也称为早期单例对象缓存。当Spring创建Bean对象时，会先创建一个早期单例对象，并将其放入earlySingletonObjects缓存中。在创建完所有Bean对象后，Spring会将earlySingletonObjects缓存中的所有早期单例对象转化为单例对象，并放入singletonObjects缓存中。
> 3. singletonFactories：这是Spring缓存Bean对象的第一级缓存，也称为单例工厂缓存。当Spring创建BeanDefinition对象时，会将其转化为一个单例工厂对象，并将其放入singletonFactories缓存中。当获取Bean对象时，Spring会先从singletonObjects缓存中查找，如果未找到，则从

> earlySingletonObjects缓存中查找，如果仍未找到，则从singletonFactories缓存中查找。如果在singletonFactories缓存中找到了对应的单例工厂对象，则会使用该工厂对象创建Bean对象并放入singletonObjects缓存中。
>
> 通过使用三级缓存，Spring可以提高Bean对象的创建效率，避免重复创建对象和循环依赖等问题。同时，三级缓存也提供了一些扩展机制，如可以使用BeanPostProcessor接口在Bean对象创建前后进行一些处理。需要注意的是，如果Bean对象的作用域不是单例，则不会使用三级缓存，而是每次创建新的Bean对象。

```
public class DefaultSingletonBeanRegistry ... {
    // 1. 最终存储单例Bean成品的容器，即实例化和初始化都完成的Bean，称之为“一级缓存”
    private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>
(256);
    // 2. 早期Bean单例池，缓存半成品对象，且当前对象已经被其他对象引用了，称之为“二级缓存”
    private final Map<String, Object> earlySingletonObjects = new
ConcurrentHashMap<>(16);
    // 3. 单例Bean的工厂池，缓存半成品对象，对象未被引用，使用时通过在工厂创建Bean，称之为“三级缓存”
    private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>
(16);
}
```

以UserService和UserDao循环依赖为例，说明三级缓存的过程：

- UserService实例化对象，但尚未初始化，将UserService存储到三级缓存；
- UserService属性注入，需要UserDao，从缓存中获取，没有找到UserDao；
- UserDao实例化对象，但尚未初始化，将UserDao存储到三级缓存；
- UserDao属性注入，需要UserService，从三级缓存中获取UserService，UserService从三级缓存移入二级缓存；
- UserDao执行其他生命周期过程，最终成为一个完整的Bean，存储的一级缓存，删除二三级缓存；
- UserService注入UserDao；
- UserService执行其他生命周期过程，最终成为一个完整的Bean，存储的一级缓存，删除二三级缓存。

8.2 常用的Aware接口

Aware接口是框架辅助属性注入的一种思想，其他框架中也可以看到类似的接口。框架具备高度封装性，我们接触到的一般是业务代码，一个底层功能API不能轻易获取到，但是，这并不意味着永远用不到这些对象，如果用到了，就可以使用框架提供的类似Aware接口，让框架注入该对象。

Aware接口	回调方法	作用

Aware接口	回调方法	作用
ServletContextAware	setServletContext(Servletcontext context)	Spring框架回调方法注入ServletContext对象，web环境生效
BeanFactoryAware	setBeanFactory(BeanFactory factory)	Spring框架回调方法注入beanFactory对象
BeanNameAware	setBeanName(String beanName)	注入当前bean在容器中的beanName
ApplicationContextAware	setApplicationContext(ApplicationContext applicationContext)	注入applicationContext对象

- Spring IoC 整体流程总结