1. Spring框架概述

## 1.1 概述

Spring是一款为了解决企业级web应用开发遇到的困难而出现的框架，经过多年的发展，Spring已经扩展到了方方面面，但我们目前只需要学习**控制反转(IOC)**和**面向切面(AOP)**即可， Spring的其他部分，如果有需要，再去学习，IOC和AOP是Spring的核心，当然还有事务管理

框架实际上是提供了一套通用功能的半成品项目，它为我们搭起了一个架子，就像所有房子都有一个骨架，但是我们建房子的时候都要重新建立一遍骨架，然后再填砖，那么如果有现成的骨架了，只填砖是不是很快？我们每个项目的具体业务可能不同，但是有大部分基础性的东西是相同的，比如文件上传，数据库查询等，这些是一般的项目都会有的，框架就是把这些东西已经做好了，拿来直接用就好了，我们只需要写具体的业务部分，免去了很多基础性的工作，加快开发速度，这就是框架。框架和类库有什么区别呢？如果把框架比作房子的骨架，那么类库就像起重器，挖掘机等工具

## 1.2 环境搭建

首先需要下载Spring框架 **spring-framework-4.0.4.RELEASE-dist**，4.0.4版官方地址为 <http://repo.springsource.org/libs-release-local/org/springframework/spring/4.0.4.RELEASE/>

此外，还需要下载一个必要组件——日志包commons-logging-1.1.3-bin ，可以在<http://commons.apache.org/> 官网下载也可以在国内网站下载。

其他诸如log4j之类的并不是必须下载的。

下载好以上两个压缩包。在workspace下新建一个lib目录作为user library。将Spring压缩包解压之后，将libs目录下的 文件都添加到你的user library中；将commons包也解压，将根目录的\*.1.3.jar（共3个）都添加到你的user library中。

之后在Eclipse中添加一个user library即可

## 1.3 三层架构

三层架构不是MVC，它和MVC有本质的区别，在使用javaEE做企业级应用开发时，经常采用三层架构分层：

**表示层(View level): 负责接收用户请求、转发请求、显示数据**

**业务层(Service level): 业务层负责组织业务逻辑**

**持久层(Dao level): 持久层负责直接对数据库进行操作**

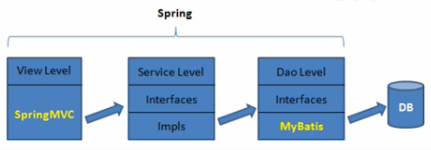
而MVC是表示层常用的架构模式，由数据model、视图view和控制器controller组成，如下图，MVC(Model、View、Controller)属于表示层(View level)，MVC中的model(数据)，是持久层(Dao level)查询的数据经由业务层(Service level)处理后得到的数据

也就是说持久层(Dao level)负责和数据库交互，对数据库直接进行增删改查等操作，持久层(Dao level)将操作的结果交付业务层(Service level)进行相应的处理，业务层(Service level)再将被处理后的、符合我们要求的数据交付给表示层(View level)，由表示层(View level)将数据渲染到视图，层层解耦



SSM和三层架构又有什么关系呢？

SSM即Spring、SpringMVC和MyBatis三个框架，MyBatis负责**持久层(Dao level)** ，SpringMVC负责**表示层(View level)** ，Spring作为整个项目的” 大管家”统管全局，整个企业级web中的所有的对象及对象间的依赖关系都由Spring进行管理



1. 依赖注入，控制反转(IOC)

## 2.1 什么是依赖

依赖就是有联系，就是必然用的到的，就是 “我若依赖你，我就不能离开你”。eg：

**一个超人类:**

**class** SuperMan {}

**超人必然有超能力，因此还需要一个超能力类：**

**class** Power {

/\*\*

\* 能力值

\*/

**protected** String ability=**null**;

/\*\*

\* 能力范围或距离

\*/

**protected** **double** range=0;

**public** Power(String ability,**double** range)

{

**this**.ability = ability;

**this**.range = range;

}

}

**此时，转回头将我们写好的超能力赋予超人**

**class** SuperMan {

**protected** Power power=**null**;

**public** SuperMan()

{

**this**.power = **new** Power(999, 100);

}

}

**此时，“超人”和“超能力”之间不可避免的产生了一个依赖。**

在日常的项目中，依赖是难以完全消除的，但是当依赖达到一定的数量时，会出现一些噩梦般的场景

想象一下，现在我们处于一个怪物横行的世界中，面对不同的怪物我们需要拥有不同超能力的超人，因此我们制造了三个超能力

/\*\*

\* 飞行能力

\*/

**class** Fly

{

//飞行速度

**protected** **int** speed;

**public** Fly(**int** speed) {}

}

/\*\*

\* 蛮力

\*/

**class** Force

{

**protected** **int** force;

**public** Force(**int** force) {}

}

/\*\*

\* 能量枪

\*/

**class** Shot

{

**protected** **int** atk;//伤害值

**protected** **double** range;//攻击范围

**public** Shot(**int** atk,**double** range) {}

}

这时候出现了一个麻烦，由于一共有三个功能，不同功能的超人能解决不同的怪兽，而我们的原则是尽量精简，即能用两个能力解决的怪兽，绝不用三个能力，这造成了一个很大的麻烦，每次我们需要一个超人时，我们都要重新改造我们的**class** SuperMan，给他换上不同的能力，再new出来。也许有人会问，那我们多造几个**class** SuperMan，用到哪个new哪个不就行了吗？呵呵，这才有三个超能力，就要造7个**class** SuperMan了，那如果30个超能力呢？300个呢？

这时，灵机一动的人想到：为什么不可以这样呢？超人的能力可以被随时更换，只需要添加/更换一个芯片(依赖)就能够添加/更换功能。这样的话就不要整个改造我们的**class** SuperMan了，只需要将我们的超能力类(依赖)注入超人即可。

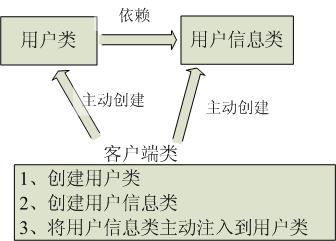
## 2.2控制反转(IOC)和依赖注入(DI)

控制反转(IOC)，即对依赖控制权的转移，程序、组件或类本身不再负责依赖对象的创建和维护，而是将依赖的控制权交由外部容器

以上面的超人为例，超能力(依赖)不再由**class** SuperMan(程序、组件或类)进行创造和维护，而是由外部对芯片(依赖)进行添加/更换，放入什么样的芯片(依赖)不再由**class** SuperMan类的声明决定，而是由外部通过超人的某个接口注入其中，依赖的控制权从超人反转到了外部的超人制造工厂，**这种由外部负责其依赖需求的行为，我们可以称其为控制反转（IoC）**

**在IOC的帮助下，原本紧密耦合的两个类解耦，变为了松散耦合。在原来的开发中，应用程序原本是老大，要获取什么资源都是主动出击，但是在IoC/DI思想中，应用程序就变成被动的了，被动的等待IoC容器来创建并注入它所需要的资源了。**

如：用户类依赖用户信息



**控制反转是一种概念，而依赖注入实现了控制反转(实现控制反转的方法很多，依赖注入是最常用的方法之一)**

在使用依赖注入以前，每段程序在使用依赖时，都是new一个依赖对象出来，自己需要哪个依赖，就去主动创建，属于高耦合，A要使用B，那么A就对B产生了依赖，也就是A和B之间存在一种耦合关系，并且是紧密耦合在一起，而使用了Spring之后就不一样了，创建合作对象B的工作是由Spring来做的，Spring创建好B对象，然后存储到一个容器里面，当A对象需要使用B对象时，Spring就从存放对象的那个容器里面取出A要使用的那个B对象，然后交给A对象使用，至于Spring是如何创建那个对象，以及什么时候创建好对象的，A对象不需要关心这些细节问题，只需要使用即可

**IOC本质就是要抛弃new的方法取得对象，通过配置来取得对象。**

## 2.3 Spring的bean

从前面我们知道Spring其实就是一个大型的工厂，而Spring容器中的Bean就是该工厂的产品.对于Spring容器能够生产哪些产品，则取决于配置文件中配置。

对于我们而言，我们使用Spring框架所做的就是两件事：开发Bean、配置Bean。对于Spring框架来说，它要做的就是根据配置文件来创建Bean实例，并调用Bean实例的方法完成“依赖注入”。如果在前面的超人的例子中，bean就是超人的超能力，我们将bean的控制权交由Spring进行管理，而我们只需要编写超能力，配置超能力即可

在Spring的配置文件（applicationContext）中，<beans>元素节点是Spring配置文件的根元素，而<bean>元素是<beans>元素的子元素，每个<bean>元素代表一个bean实例，定义Bean时至少需要指定两个属性。

**1.Id：**确定该Bean的唯一标识符，容器对Bean管理、访问、以及该Bean的依赖关系，都通过该属性完成。Bean的id属性在Spring容器中是唯一的，他就像一个人的身份证一样，唯一标识一个人。

**2.Class：**指定该Bean的具体实现类。注意这里不能使接口，必须是类。通常情况下，Spring会直接使用new关键字创建该Bean的实例，因此，这里必须提供Bean要实例化的类的类名。

eg：

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

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:context=*"*[*http://www.springframework.org/schema/context*](http://www.springframework.org/schema/context)*"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

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.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

<!-- 定义第一个Bean实例：bean1 -->

<bean id=*"bean1"* class=*"com.Bean1"* />

<!-- 定义第二个Bean实例：bean2 -->

<bean id=*"bean2"* class=*"com.Bean2"* />

</beans>

Spring中bean的配置有三种方式，分别是：

**基于XML的配置**

**基于注解的配置**

**基于Java类的配置**

上面的例子是基于XML的配置

在有了spring的IOC后，我们在写代码的过程中再也不需要new创建对象了，这一切都有IOC帮我们完成了

## 2.4 单元测试

### 2.4.1 SE的单元测试

单元测试是指对软件中的最小可测试单元进行检查和验证。这样可以减少后期维护的精力和费用

让我们先看看以往在javaSE中我们是怎么做的：

单元测试需要Junit包，Junit是一个java测试框架，只需要将其导入库中即可，右键要测试的java文件，new-> other->输入Junit Test Case->勾选setUp和tearDown->next->勾选需要测试的函数->finish自动生成一个测试类，里面包含对应函数的空的测试用例，现在仅需将想要进行的测试内容代码填写进去即可

**public** **class** testTest\_1 {

@Before

**public** **void** setUp() **throws** Exception {

}

@After

**public** **void** tearDown() **throws** Exception {

}

@Test

**public** **void** test() {

*fail*("Not yet implemented");

}

}

这里有三个注解 @Before @After @Test

**在被@Test注解标记的方法上右键->run as->1 JUnit Test即可运行该方法(甚至不需要main)**，junit窗口出现绿色代表运行成功，红色代表运行失败

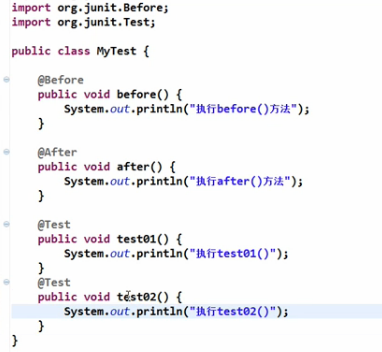
**被@Before注解标记的方法是每一个@Test注解标记的方法执行前要执行的方法，setUp中一般放的是初始化的相关代码**

**被@After注解标记的方法是每一个@Test注解标记的方法执行后要执行的方法，tearDown中一般放的是清理垃圾、资源释放的一些方法**

请注意：

**1.测试类中的方法必须都是public void**

**2.一般情况下我们不会在测试类(如testTest\_1)里面运行测试方法(如test())，我们一般在自己项目的类中导入org.junit.\*，自定义setUp()和tearDown()，只要在对应方法前加上对应的注解即可**



**3.在maven项目中，Junit一般声明为**

<dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>3.8.1</version>  
 <scope>test</scope>  
</dependency>

**这就意味着必须在src/test/java下，maven在编译的时候，src/main/java下是不引用<scope>test</scope>的jar，而编译src/test/java下的测试这会引用<scope>test</scope>的jar**

**简而言之，在src/main/java下无法使用junit框架**

### 2.4.2 Spring的单元测试

请注意：

**需要通过 @RunWith(SpringJUnit4ClassRunner.class) 来启动 Spring 对测试类的支持**

**需要通过 @ContextConfiguration 注释标签来指定 Spring 配置文件或者配置类的位置**

**需要通过 @Transactional 来启用自动的事务管理**

**可以使用 @Autowired 自动织入 Spring 的 bean 用来测试**

eg：

以下是以一个servlet的测试

@RunWith(BlockJUnit4ClassRunner.**class**)

**public** **class** TestTest {

@Before

**public** **void** setUp() **throws** Exception {

}

@After

**public** **void** tearDown() **throws** Exception {

}

@Test

**public** **void** testDoGetHttpServletRequestHttpServletResponse() {

*fail*("Not yet implemented");

}

@Test

**public** **void** testServiceServletRequestServletResponse() {

*fail*("Not yet implemented");

}

@Test

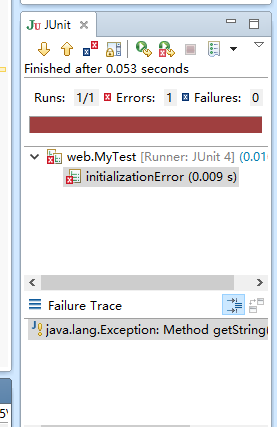
**public** **void** testHttpServlet() {

*fail*("Not yet implemented");

}

}

运行结果：红色代表出错



参考<https://www.ibm.com/developerworks/cn/java/j-lo-springunitest/>

## 2.4 传统方法和Spring方式的比较

以一个helloDemo为例：

**传统方式：**

**public** **class** Demo {

**private** String myName=**null**;

**public** String getMyName(){

**return** myName;

}

**public** **void** setMyName(String name){

**this**. myName =name;

}

**public** String say(){

**return** "hello!"+ myName;

}

**public** **static** **void** main(String[] args) {

Demo demo=**new** Demo();

demo.setName("Lily");

System.*out*.println(demo.say());;

}

}

**Spring方式：**

**第一步**，搭建环境，引入Spring包和日志包

**第二步**，创建Spring的核心文件applicationContex.xml(不一定要这么命名该文件)，它是我们的bean容器文件，Spring中的bean配置都在这个文件中进行，该文件一般放在src目录下即可

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

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:context=[*http://www.springframework.org/schema/context*](http://www.springframework.org/schema/context) xmlns:tx=*"http://www.springframework.org/schema/tx"*

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.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

<!-- 在beans容器中对bean进行配置，在框架加载时就会自动创建该bean类到内存中去 -->

<bean id=*"demo"* class=*"com.demo.Demo"*>

<property name=*"*myName*"*>

<value>丁健</value>

</property>

</bean>

<!-- 上面的配置相当于Demo demo=new Demo();demo.setName("丁健"); -->

</beans>

**第三步**，配置好bean容器后，在执行时需要获取bean容器(applicationContex)，并将依赖注入，这里的applicationContext.xml是相对路径，因为它在src下，因此这么写，如果在某个包下，还要加上包名，如：容器在包com.study.myDemo下就是com/study/myDemo/applicationContext.xml

**public** **static** **void** main(String[] args) {

//获取bean容器

ApplicationContext ac = **new** ClassPathXmlApplicationContext("applicationContext.xml");

//将Demo类的依赖name(丁健)注入

Demo demo=(Demo)ac.getBean("demo");

System.*out*.println(demo.say());;

}

**请注意！bena(Demo)类中必须提供set(setMyName)和get(getMyName)方法才能正常注入依赖，set和get的方法名请按照规范编写**

1. bean

## 3.1 获取bean

在Spring实现IOC的过程中，有2个最基本最重要的包，即**org.springframework.beans** 和**org.springframework.context**，这两个包中大量引用了java的反射机制，通过动态调用的方式避免了硬编码，为spring的反向控制特性提供了基础。在这2个包中，最重要的类是BeanFactory和ApplicationContext。

**BeanFactory ：bean的生产工厂，它可以加载并初始化bean**

**ApplicationContext ：应用上下文，bean的容器，保存bean的对象**

以上两种方式，bean工厂和bean容器都可以获取bean对象，但是**BeanFactory**在初始化时不创建bean对象，而是在facory.getBean("beanName")时创建bean对象。而**ApplicationContext**在初始化时就会将所有设为singleton单例模式的bean对象都创建，然后将自己作为bean的容器保存所有单例的bean，在ac. getBean("beanName")时获取已创建的单例bean对象

**ApplicationContext**是比**BeanFactory**更高级的容器，功能更强大。但beanFactory并非一无是处，相比于**ApplicationContext**，它更节约内存，因此在诸如移动设备等内存较小的设备上编程时，应使用bean工厂。但由于**ApplicationContext**提前加载了bean，**ApplicationContext**运行起来更快，因此它仍然是使用最多的容器

### 3.1.1 BeanFactory

eg：

BeanFactory factory=**new** XmlBeanFactory(**new** ClassPathResource("applicationContext.xml"));

Demo demo=(Demo)factory.getBean("demo");

### 3.1.2 ApplicationContext

获取applicationContext容器对象有三种方法：

ClassPathXmlApplicationContext() 从类路径中加载(相对src)

FileSystemXmlApplicationContext() 从文件路径中加载(必须写绝对路径)

XmlWebApplicationContext() 从web系统中加载(tomcat启动后自动加载)

//获取bean容器

ApplicationContext ac = **new** ClassPathXmlApplicationContext("applicationContext.xml");

**//**FileSystemXmlApplicationContext("E: \\Study\\JAVA\\J2EE\\src\\applicationContext.xml");

**//**FileSystemXmlApplicationContext("E: \\Study\\JAVA\\J2EE\\src\\applicationContext.xml");

//将Demo类的依赖name(丁健)注入

Demo demo=(Demo)ac.getBean("demo");

### 3.1.3 ContextLoaderListener

在实际开发中，并不需要我们手动装配spring的配置文件，我们只需要在web.xml文件中，使用ContextLoaderListener即可实现spring配置文件的自动装配

ContextLoaderListener用来创建Spring容器；而<context-param>标签，用来指定spring配置文件的位置，该两个标签共同起作用，也就可以在web项目启动后，成功的创建了Spring容器了。

<context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath\*:spring/spring-\*.xml</param-value>  
</context-param>  
<listener>  
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
</listener>

上述配置将会加载符合spring/spring-\*\*\*\*.xml的所有xml文件

## 3.2装配bean

### 3.2.1 bean的属性和子节点(3.2.1节可以当做3.2的字典使用)

在xml文件中，一个<beans>节点代表整个容器，<beans>节点可以有很多<bean>节点，每个<bean>节点都代表一个bean

在配置一个bean之前，我们首先需要了解它有哪些常用的属性。

在2.3中有提过，每个bean都有两个必须的属性——id和class，这两个属性分别是bean的唯一标识和bean所用的java类，除此之外，bean还有相当多的属性和子节点

#### 3.2.1.1常用属性：

##### 1.scope

bean最重要的属性之一，指明了本bean的作用域(可访问性)，一般有五个属性值，其中三个只能用于web开发

这里我们用一个例子帮助理解：

eg：

**bean类：**

**public** **class** Demo {

**private** **int** i = 0;

**public** **void** add() {

System.*out*.println(i++);

}

}

**xml文件：**

<bean id=*"Add"* scope=*"singleton"* class=*"com.demo.Demo"*></bean>

**使用bean：**

ApplicationContext ac = **new** ClassPathXmlApplicationContext(

"applicationContext.xml");

BeanFactory factory = **new** XmlBeanFactory(**new** ClassPathResource(

"applicationContext.xml"));

ApplicationContext ac1 = **new** ClassPathXmlApplicationContext(

"applicationContext.xml");

Demo bean1=(Demo) ac.getBean("Add");

bean1.add();

Demo bean2=(Demo) ac1.getBean("Add");

bean2.add();

Demo bean3=(Demo) ac.getBean("Add");

bean3.add();

Demo bean4=(Demo) ac1.getBean("Add");

bean4.add();

**(1)singleton：**单例，默认值，如果没有配置scope的值，默认为单例，这里需要注意的是：相对于每个ApplicationContext或BeanFactory来说是单例的，即同一个ApplicationContext或BeanFactory下只会有一个该bean对象(两个容器的话可以各有一个单例对象)

**结果为：**

0

0

1

1

**(2)prototype：**每次请求都是一个新的对象，一旦这个新的对象给了请求方,那么容器就不在持有对这个对象。就像嫁出去的姑娘,泼出去的水,娘家以后不管了。

**需要注意的是：由于每次都是创建新的bean，因此ApplicationContext在初始化容器时并不会像singleton一样预先创建好一个bean放在容器内**

将scope换为**prototype**后上例的结果为：

0

0

0

0

**(3)request：**只在web应用中使用，和prototype没什么区别,在web情况下每来一个请求,分配一个新实例。

**(4)session：**只在web应用中使用，对于放到session中的信息,可以将scope设置为session,它除了比request更长的存活时间外,其他方面没什么区别。也是为每一个会话提供一个新实例

**(5)global session：**global session只有应用在基于porlet的web应用程序中才有意义，它映射到porlet的global范围的session，如果普通的servlet的web 应用中使用了这个scope，容器会把它作为普通的session的scope对待。

##### 2.init-method和destroy-method

有时候，我们在配置bean的时候，需要额外做一些操作，比如创建bean前获取一些必要的资源，销毁时释放掉资源，这种情况下，就需要我们自己定制初始化方法和销毁方法

eg：

**xml配置：**

<bean id=*"Add"* init-method=*"myInit"* destroy-method=*"myDestroy"* class=*"com.demo.Demo"*></bean>

**bean类：**

**public** **class** Demo {

**private** String name = **null**;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **void** myInit(){

System.*out*.println("初始化");

}

**public** **void** myDestroy(){

System.*out*.println("销毁");

}

}

**我们也可以通过注解的方式进行配置，这样就不需要在xml的bean节点中加这两个属性了，但请注意,使用spring注解需要在<beans>节点下配置**<context:annotation-config/>

**eg：**

**xml配置：**

<context:annotation-config/>

<bean id=*"Add"* class=*"com.demo.Demo"*></bean>

**bean类：**

**public** **class** Demo {

**private** String name = **null**;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**@PostConstruct**

**public** **void** myInit(){

System.*out*.println("初始化");

}

**@PreDestroy**

**public** **void** myDestroy(){

System.*out*.println("销毁");

}

}

##### 3.autowire自动装配

该属性帮助本bean自动装配依赖bean，常用有6个常用属性，请对照下面的属性来看本例：

例子：

**首先定义一个员工类**

**public** **class** Staff {

**private** String name;

**private** Dog dog;

**public** Staff(){}

**public** Staff(String name){

**this**.name=name;

}

//不提供set方法也可以

**public** String getName() {

**return** name;

}

**public** Dog getDog() {

**return** dog;

}

**public** **void** setDog(Dog dog) {

**this**.dog = dog;

}

}

**然后定义一个狗类**

**public** **class** Dog {

**private** String dogName;

**private** String breed;

**public** Dog(){}

**public** Dog(String breed){

**this**.breed=breed;

}

**public** String getDogName() {

**return** dogName;

}

**public** **void** setDogName(String dogName) {

**this**.dogName = dogName;

}

**public** String getBreed() {

**return** breed;

}

}

**最后通过xml配置依赖关系(具体使用步骤省略)**

<bean id=*"staff"* class=*"com.company.Staff"*>

<constructor-arg index=*"0"* type=*"java.lang.String"* value=*"丁健"*></constructor-arg>

**<property name=*"dog"*>**

**<ref bean=*"dog"* />**

**</property>**

</bean>

<bean id=*"dog"* class=*"com.company.Dog"*>

<constructor-arg index=*"0"* type=*"java.lang.String"* value=*"金毛"*></constructor-arg>

<property name=*"dogName"*>

<value>cookie</value>

</property>

</bean>

1. **byName**

从Spring环境中获取目标对象时，目标对象中的属性会根据bean类中需要注入的依赖的属性名在整个Spring环境中查找<bean>标签的id属性值。如果有bean的依赖属性名和某个<bean>的id属性值相同的，那么获取这个对象，实现关联。

**整个Spring环境：表示所有的spring配置文件中查找**，那么id不能有重复的。

**在上例中, class** Staff 中有一个 Dog dog的依赖，其dog与bean<bean id=*"dog"* class=*"com.company.Dog"*>的id相同，因此可以通过byName自动装配

那么xml配置应该为：

<bean id=*"staff"* class=*"com.company.Staff"* autowire=*"byName"*>

<constructor-arg index=*"0"* type=*"java.lang.String"*

value=*"丁健"*></constructor-arg>

<!—注释掉

<property name="dog">

<ref bean="dog" />

</property>

-->

</bean>

<bean id=*"dog"* class=*"com.company.Dog"*>

<constructor-arg index=*"0"* type=*"java.lang.String"*

value=*"金毛"*></constructor-arg>

<property name=*"dogName"*>

<value>cookie</value>

</property>

</bean>

1. **byType**

从Spring环境中获取目标对象时，目标对象中的属性会根据类型在整个spring环境中查找<bean>标签的class属性值。如果有相同的，那么获取这个对象，实现关联。

**这是一个极少用的属性**

**缺点：如果存在多个相同类型的bean对象，会出错。**

**如果属性为单一类型的数据，那么查找到多个关联对象会发生错误。**

**如果属性为数组或集合(泛型)类型，那么查找到多个关联对象不会发生异常。**

<bean id=*"staff"* class=*"com.company.Staff"* autowire=*"byType"*>

<constructor-arg index=*"0"* type=*"java.lang.String"*

value=*"丁健"*></constructor-arg>

<!—注释掉

<property name="dog">

<ref bean="dog" />

</property>

-->

</bean>

<bean id=*"dog"* class=*"com.company.Dog"*>

<constructor-arg index=*"0"* type=*"java.lang.String"*

value=*"金毛"*></constructor-arg>

<property name=*"dogName"*>

<value>cookie</value>

</property>

</bean>

1. **constructor**

使用构造方法完成对象注入，其实也是根据构造方法的参数类型进行对象查找，相当于采用byType的方式。

1. **autodetect**

自动选择：如果对象没有无参数的构造方法，那么自动选择constructor的自动装配方式进行构造注入。如果对象含有无参数的构造方法，那么自动选择byType的自动装配方式进行setter注入。

1. **no**

不支持自动装配功能

1. **default**

表示默认采用上一级标签的自动装配的取值。如果存在多个配置文件的话，那么每一个配置文件的自动装配方式都是独立的。

##### 4.lazy-init懒加载

spring容器在初始化的时候不会初始化这个bean，当配置上lazy-init=true之后，表示该bean是懒加载模式，什么时候用到了该bean才会进行初始化。

<bean name="role1" class="com.fz.entity.Role" lazy-init="true">

#### 3.2.1.2常用子节点：

##### 1. property

**1.装配普通依赖(基本类型、对象和bean)**

<bean>节点的<property>子节点指明了使用bean类提供的set方法来给bean注入依赖，可以通过该子节点注入任何东西，从基本数据类型到类对象，甚至是应用系统的bean

<property>有一个<value>子节点，用来提供被注入的值，还有两个属性name和ref，name指明了依赖要被注入到bean的哪个属性中去，ref是当我们要注入的依赖也是一个bean时，指明这个依赖bean的url，ref可以作为property的属性，也可以作为其子节点，最好作为子节点使用，ref子元素还有三种属性：

**bean**：bean属性可以从其它bean配置文件中(跨xml文件)寻找需要加载的目标bean。bean属性的值可以同目标bean的id属性相同，也可以同它的name属性中任意的一个值相同。

**local：**只能本bean配置文件中(不能跨xml文件)寻找需要加载的目标bean，而且local属性值必须同目标bean的id属性相同。

**parent：**使用了parent指定另一个bean，意味着本bean是继承于指定的bean，可以使用父bean的属性和方法

**请注意：**

**1.被注入依赖的bean必须提供对应的setter(**setMyName**)方法！当然，也可以不使用set方法，而使用构造函数进行注入**

**2.我们还可以给<property>设置空值，eg：**

<property name=*"MyName"*>

<null/>

</property>

下面是一个装配bean的简单案例

eg：

**xml配置：**

<bean id=*"Add"* class=*"com.demo.Demo"*>

<property name=*"myName"*>

<value>Ricardo</value>

<ref bean=*"helloBean"*/> //该bean不在本xml文件中

</property>

</bean>

**bean类：**

**public** **class** Demo {

**private** String myName = **null**;

**public** String getMyName() {

**return** myName;

}

**public** **void** setMyName(String name) {

**this**.myName = name;

}

**public** **void** say(){

System.*out*.println("hello!"+myName);

}

}

**2.装配集合类型**

有时候，我们常常会需要给bean装入集合类型的依赖，eg：

**public** **class** Company {

**private** String companyName;//公司名

**private** String[] staff;//雇员列表

**public** String getCompanyName() {

**return** companyName;

}

**public** **void** setCompanyName(String companyName) {

**this**.companyName = companyName;

}

**public** String[] getStaff() {

**return** staff;

}

**public** **void** setStaff(String[] staff) {

**this**.staff = staff;

}

}

我们可以通过<property>节点的<list>子节点进行集合类型的注入。请注意：

list类型和普通数组类型使用<list>子节点

set集合使用<set>子节点，请注意：由于set集合不可重复的特性，如果在<value>子节点中多次给set集合注入同一个依赖，之前注入的会被后注入的覆盖

map集合相对较复杂一些，当然也使用<map>节点，和set、list不同的是，map使用<entry>子节点，entry子节点有key、key-ref、value、value-ref四个属性，后缀有-ref的属性可以指向其他bean，没有的只能给基本数据类型，在下例中，假设还加入了一组 **姓名—员工** 的键值对

<bean id=*"company"* class=*"com.company.Company"*>

<property name=*"companyName"*>

<value>阿里巴巴</value>

</property>

<property name=*"staff"*>

<list>

<value>马云</value>

<value>张三</value>

<value>李四</value>

<value>王五</value>

</list>

</property>

<property name=*"staffId"*>

<map>

<entry key=*"马云"* value-ref=*"staff\_1"*></entry>

<entry key=*"张三"* value-ref=*"staff\_2"*></entry>

<entry key=*"李四"* value-ref=*"staff\_3"*></entry>

</map>

</property>

</bean>

//此处省略*staff\_2和staff\_3*

<bean id=*"staff\_1"* class=*"com.company.Staff"*>

<property name=*"id"*>

<value>1000</value>

</property>

</bean>

使用：

ApplicationContext appContext=**new** ClassPathXmlApplicationContext("com/company/company.xml");

Company company=(Company)appContext.getBean("company");

System.*out*.println("公司职员有:");

**for** (String staffName : company.getStaff()) {

System.*out*.println(staffName);

}

//获取姓名—员工键值对

Map<String, Staff> allStaff=company.getNameStaff();

**for** (String key : allStaff.keySet()) {

System.*out*.println("name:"+key+" id:"+allStaff.get(key).getId());

}

**当然，如果需要注入bean的集合而不是基本类型的bean，完全没问题，xml只要按如下配置即可：**

<bean id=*"staff\_1"* class=*"com.company.Staff"*>

<bean id=*"staff\_2"* class=*"com.company.Staff "*>

<bean id=*"staff\_3"* class=*"com.company.Staff "*>

<bean id=*"staff\_4"* class=*"com.company.Staff "*>

<bean id=*"company"* class=*"com.company.Company"*>

<property name=*"companyName"*>

<value>阿里巴巴</value>

</property>

<property name=*"staff"*>

<list>

这里的staff不再是基本类型集合String[]，而是bean集合,由于在同一个xml配置文件中，使用local和bean属性中的任意一个都可以，当然，如果只需要注入基本数据类型，一般不用<ref>，只用<value>就行了

<ref bean=*"staff\_1"* />

<ref local=*"staff\_2"* />

<ref bean=*"staff\_3"* />

<ref bean=*"staff\_4"* />

</list>

</property>

</bean>

##### 2. constructor-arg

该子节点可以通过构造函数给bean注入依赖，这样一来，即使没有定义set方法，也能自动注入依赖，当然，我们需要在bean类定义时提供有参的构造函数给spring，让框架自动注入依赖(最好再提供一个无参构造函数)

index属性是构造方法参数列表中的参数位置

type是本参数的类型

value是参数值

**ps：不要问为什么不在定义bean时用默认参数这种很蠢的问题**

**xml配置：**

<bean id=*"staff"* class=*"com.company.Staff"*>

<constructor-arg index=*"1"* type=*"int"* value=*"20"*></constructor-arg>

<constructor-arg index=*"0"* type=*"java.lang.String"* value=*"张三"*></constructor-arg>

</bean>

**bean定义：**

**public** **class** Staff {

**private** **int** age;

**private** String name;

**public** Staff(){}

**public** Staff(String name,**int** age){

**this**.name=name;

**this**.age=age;

}

//通过构造函数注入依赖，不提供set方法也可以

**public** String getName() {

**return** name;

}

**public** **int** getAge() {

**return** age;

}

}

**请注意：spring会尽量使用最符合<constructor-arg>的构造函数，也即是如果有三个构造函数，函数1有一个参数、函数2两个、函数3三个参数，如果<bean>配置中提供了三个<constructor-arg>,spring会使用三个参数的构造方法，如果只提供两个，spring会使用两个参数的构造方法，以此类推。当然，如果你提供了四个<constructor-arg>，只会报错**

### 3.2.2 bean的生命周期

使用bean的流程，无非是先获取bean容器，再通过容器获取bean

1. **实例化**

当bean被载入到容器中时，其生命周期就开始了(对于声明为singleton的bean来说，BeanFactory和ApplicationContext的载入时间是不同的)

首先加载xml文件，然后将bean(scope=singleton)实例化到内存

1. **注入依赖属性**

通过读取xml中bean的配置，将预先配置好的依赖注入bean中，前提是bean中的要注入的依赖属性有get和set方法，bean容器会自动调用set方法进行注入

1. **BeanNameAware(可选阶段)**

如果Bean类实现了BeanNameAware接口(实现该接口需要在bean类中重写setBeanName(String arg0)方法)，Spring就将Bean的ID通过参数传递给setBeanName()方法，即arg0就是BeanID，在过程在一般的开发中使用较少

eg：

**public** **class** Demo **implements** BeanNameAware{

**private** **String** name = **null**;

@Override

**public** **void** setBeanName(String arg0) {

System.*out*.println("这个bean的bean id是："+arg0);

}

**public** **String** getName() {

**return** name;

}

**public** **void** setName (**String** name) {

**this**.name = name;

}

}

1. **BeanFactoryAware (可选阶段)**

如果Bean类实现了BeanFactoryAware接口(实现该接口需要在bean类中重写setBeanDactory (BeanFactory bf)方法)，Spring将调用setBeanDactory(BeanFactory arg0)方法并把BeanFactory容器实例作为参数传入。

eg：

**public** **class** Demo **implements** BeanFactoryAware {

**private** String name = **null**;

**public** Demo(){

System.*out*.println("bean被创建："+i+++"次");

}

@Override

**public** **void** setBeanFactory(BeanFactory arg0) **throws** BeansException {

//defining beans[myDemo,test];

System.*out*.println("bean工厂:"+arg0);

arg0.getBean("test");

arg0.getBean("test");

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

**请注意！请不要在bean中用通过**setBeanFactory**获取的beanFactory再次获取该bean，这样会被编译器检测到无限循环创建bean而报错**

1. **ApplicationContextAwaer (可选阶段)**

如果Bean实现了ApplicationContextAwaer接口(实现该接口需要在bean类中重写setApplicationContext(ApplicationContext arg0)方法)，Spring容器将调用方法，把应用上下文对象(即ApplicationContext容器)作为参数传入.

eg：略

1. **BeanPostProcessor后置处理器 (可选阶段)**

这里和前面不同，虽然BeanPostProcessor仍是接口，但我们一般不直接让bean类实现它，而是另外创建一个实现它的类，然后将该类作为一个bean配置到xml中去，这个实现了BeanPostProcessor的特殊bean对象会对本容器的所有创建的bean生效，相当于一个过滤器，这里面有两个方法：

eg：

**public** **class** MyBeanPostProcessor **implements** BeanPostProcessor{

@Override

**public** Object postProcessAfterInitialization(Object arg0, String arg1)

**throws** BeansException {

// **TODO** Auto-generated method stub

**return** arg0;

}

@Override

**public** Object postProcessBeforeInitialization(Object arg0, String arg1)

**throws** BeansException {

// **TODO** Auto-generated method stub

**return** arg0;

}

}

**postProcessBeforeInitialization(Object arg0, String arg1)**是在本容器的所有bean对象**(arg0)**创建前执行的方法

参数**arg0**是本次容器调用getBean要实例化的bean对象，参数**arg1**是这个被实例化的bean的id，返回值是抛给下一个处理的bean对象，如**postProcessBeforeInitialization**会将自己的返回值抛给**afterPropertiesSet方法**，经过几个环节后，**postProcessAfterInitialization**会将自己的返回值抛给getBean()，因此这两个方法的返回值不能为null，否则，getBean()获取的就是null

**注意！只需要将实现了BeanPostProcessor接口的特殊bean配置到xml中即可绑定容器和后置处理器，不需要其他操作**

**通过这一步骤我们可以实现AOP编程(面向切面编程)，AOP编程可以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术。**

1. **InitializingBean (可选阶段)**

如果Bean类实现了InitializingBean接口(实现该接口需要在bean类中重写afterPropertiesSet ()无参方法)，如果bean实现了该接口，则**postProcessBeforeInitialization**后会调用该方法

1. **定制的初始化方法(可选阶段)**

如果我们的bean有一些特殊的配置，不方便使用后置处理器给所有的bean都进行处理，就可以使用自己设置的初始化方法，只需要在xml文件的<bean>中指定属性init-method为bean中已有的方法即可，然后在该方法中配置bean

eg：

<bean id=*"xx"* init-method=*"configMethod"* class=*"web.MyTest"*></bean>

1. **BeanPostProcessor后置处理器 (可选阶段)**

如果容器绑定了特殊的bean——实现**BeanPostProcessor**接口的bean类，则此阶段会执行**postProcessAfterInitialization**方法

**postProcessAfterInitialization (Object arg0, String arg1)** 是在本容器的所有bean对象**(arg0)**创建后执行的方法，该方法会将自己的返回值抛给getBean()方法作为getBean的返回值，然后就可以使用bean对象了

1. **销毁bean**

以下方法可以销毁容器

ApplicationContext ac = **new** ClassPathXmlApplicationContext(

"applicationContext.xml");

((AbstractApplicationContext) ac).close();

但我们也可以给某些bean实现DisposableBean接口(实现该接口需要在bean类中重写destroy ()方法)，我们可以在destroy()中为实现该接口的bean做特定的清理操作

eg：

**public** **class** Demo **implements** DisposableBean{

**private** String name = **null**;

**public** **void** destroy() **throws** Exception {

// **TODO** Auto-generated method stub

System.*out*.println("我被清理了");

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

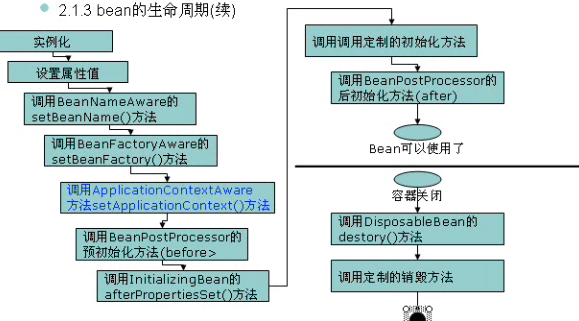
}

}

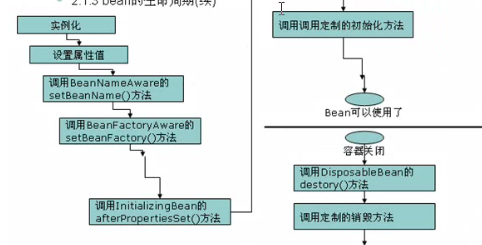
1. **通过定制方法销毁bean**

下一步还可以通过定制的destroy-method方法指定bean中的某个方法作为清理方法

<bean id=*"xx"* destroy-method=*"myDestroyMethod"* class=*"web.MyTest"*></bean>



**以上是applicationContext的生命周期，BeanFactory则会简单一些(没有ApplicationContextAwaer和后置处理器BeanPostProcessor)**



**在实际开发中，我们用的更多的是1,2,6,9,11这几步**

### 3.2.3装配bean

#### 3.2.3.1如何装配：

装配bean就是告诉容器，它拥有哪些bean，以及bean之间的依赖关系是如何的

在2.3节我们提到过，Spring中bean的配置有三种方式，分别是：

**基于XML的配置**

**基于注解的配置**

**基于Java类的配置**

下面我们先来讲讲如何通过xml配置bean

通过读取xml文件获取bean容器的方式有四种，前三种是获取ApplicationContext的，最后一种是获取BeanFactory的：

**ClassPathXmlApplicationContext()**  从类路径中加载(相对src)

**FileSystemXmlApplicationContext()**  从文件路径中加载(必须写绝对路径)

**XmlWebApplicationContext()**  从web系统中加载(tomcat启动后自动加载)

eg:

**ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");**

**//**FileSystemXmlApplicationContext("E: \\Study\\JAVA\\J2EE\\src\\applicationContext.xml");

**XmlBeanFactory()**  通过创建ClassPathResource获取xml文件

eg:

BeanFactory factory=**new** XmlBeanFactory(**new**

ClassPathResource("applicationContext.xml"));

#### 3.2.3.2 set还是构造函数：

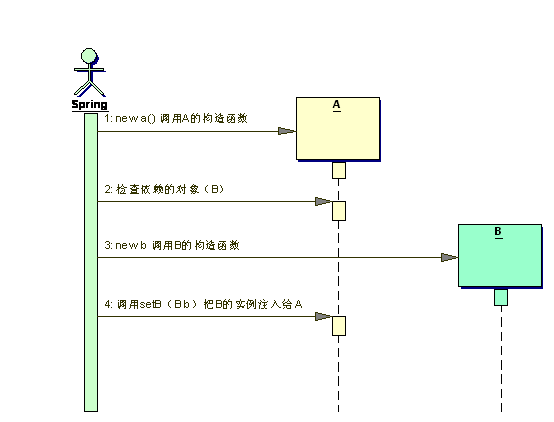
之前，我们都是在bean中定义set方法，通过set方法给bean的属性注入依赖，其实，给一个bean注入依赖，不止可以通过set方法，还可以通过构造函数

**ps：具体用法参考3.2.1.2节的constructor-arg**

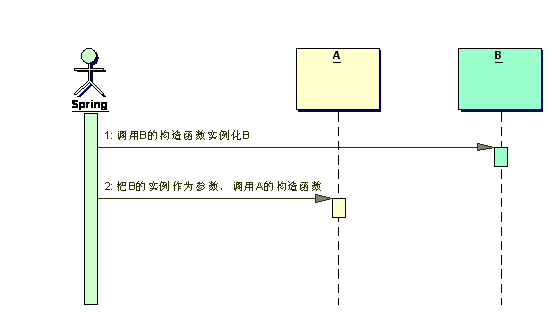
首先，我们从构造bean的流程上看一看二者的区别：

如果我们需要获取一个bean A，获取bean A需要注入bean B

1. set方法
2. 首先调用A的构造函数
3. 发现A有一个依赖叫B，找到B
4. 调用B的构造函数
5. 调用set方法将B注入A中



1. 构造器
2. 准备调用A的构造函数，却发现需要B的实例，找到B
3. 调用B的构造函数实例化B
4. 调用A的构造函数注入依赖



说了这么多，那么我们到底用哪个好呢？

**通过set方法注入的依赖是非强制性的，即不注入也能生成bean实例，而通过构造函数注入的依赖是强制性的，即必须先注入才能生成bean实例**

因此，在我们构造一个bean时，如果某些依赖是bean所必须的，则通过构造函数注入，相对的，如果某些依赖在bean中不一定用得着的，则可以考虑通过set方法注入

#### 3.2.3.3自动装配(autowire)：

set注入和构造注入有时在做配置时比较麻烦。所以框架为了提高开发效率，提供自动装配功能，简化配置。Spring框架式默认不使用自动装配的，要想使用自动装配需要修改spring配置文件中<bean>标签的autowire属性，这样这个bean就使用自动装配进行装配

**具体使用请参见3.2.1.1第3节**

#### 3.2.3.4通过注解装配：

之前我们提到过，Spring中bean的配置有三种方式，分别是：

**基于XML的配置**

**基于注解的配置**

**基于Java类的配置**

本节将讲述如何通过注解装配bean

从spring2.5版本起，spring提供了<context:annotation-config/>，该配置可以向Spring容器注册以下四个BeanPostProcessor：

AutowiredAnnotationBeanPostProcessor

CommonAnnotationBeanPostProcessor

PersistenceAnnotationBeanPostProcessor

RequiredAnnotationBeanPostProcessor

那么，为什么要注册这四个BeanPostProcessor呢？简单地来说，这样可以激活注解功能

#### 3.2.3.5特殊的bean：

在Spring中，有一些特殊的bean来帮助我们处理一些特殊的需求

1. 实现了**BeanPostProcessor后置处理器** 接口的bean(详见3.2.2节第六个周期和第九个周期)，它在创建bean的时候给程序员提供修改bean的能力
2. 分散配置

在开发中，由于项目日益庞大，经常会出现多个bean使用一样的属性值的情况，如果我们在每个bean中都分别配置，一旦属性值发生改变，将会修改多处，在一个大型工程中，这种情况相当致命。而分散配置，可以读取外部的**.properties**文件的内容，然后通过占位符的方式向bean.xml中注入信息，也就是配置文件的配置文件，有点儿类似于php的yii框架中的bootstrap.php文件

这是一个非常常用的功能呢

eg：

**myConfig.properties文件：**

master=\u4E01\u5065

dogName=cookie

dogBreed=\u91D1\u6BDB

**company.xml文件：**

**<context:property-placeholder location=*"com/company/myConfig.properties"*/>**

<bean id=*"staff"* class=*"com.company.Staff"*>

<constructor-arg index=*"0"* type=*"java.lang.String"*

value=*"${master}"*></constructor-arg>

</bean>

<bean id=*"dog"* class=*"com.company.Dog"*>

<constructor-arg index=*"0"* type=*"java.lang.String"*

value=*"${Breed}"*></constructor-arg>

<property name=*"dogName"*>

<value>${dogName}</value>

</property>

</bean>

灰色部分是重点，**<context:property-placeholder location=*"com/company/myConfig.properties"*/>**的location指定了分散配置文件相对于根目录的路径

如果需要引入多个分散配置文件，只需要用逗号分隔，eg：**<context:property-placeholder location=*"com/company/myConfig.properties, com/company/myConfig2.properties "*/>**

1. 其他

都在bean声明周期中，自己看

1. AOP面向切面编程

## 4.1 AOP概述

什么是面向切面编程？简单地来说，就是对某一批(多个)对象统一进行编程，在不增加代码的情况下还可以增加新功能。**这种在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。**

我们知道，面向对象的特点是继承、多态和封装。而封装就要求将功能分散到不同的对象中去，这在软件设计中往往称为职责分配。实际上也就是说，让不同的类设计不同的方法。这样代码就分散到一个个的类中去了。这样做的好处是降低了代码的复杂程度，使类可重用。

但是人们也发现，在分散代码的同时，也增加了代码的重复性。什么意思呢？比如说，我们在两个类中，可能都需要在每个方法中做日志。按面向对象的设计方法，我们就必须在两个类的方法中都加入日志的内容。也许他们是完全相同的，但就是因为面向对象的设计让类与类之间无法联系，而不能将这些重复的代码统一起来。

我们可以将这段代码写在一个独立的类独立的方法里，然后再在这两个类中调用。但是，这样一来，这两个类跟我们上面提到的独立的类就有耦合了，它的改变会影响这两个类。那么，有没有什么办法，能让我们在需要的时候，随意地加入代码呢？这种在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。

AOP在开发新框架的时候用很多，但实际项目中用的不是很多，最典型的是Junit测试框架，其中大量使用了aop编程

## 4.2 一个简单的案例

假设现有一个服务类接口和两个服务类：

**public** **interface** ServiceInterface {

**private** String name;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

@Override

**public** **void** sayHello() {

// **TODO** Auto-generated method stub

}

}

**public** **class** Service\_1 **implements** ServiceInterface{

**private** String name;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

@Override

**public** **void** sayHello() {

// **TODO** Auto-generated method stub

System.*out*.println("hello"+name);

}

}

**public** **class** Service\_2 **implements** ServiceInterface {

@Override

**public** **void** sayHello() {

// **TODO** Auto-generated method stub

System.*out*.println("hello"+name);

}

}

<bean id=*"service\_1"* class=*"com.aop.Service\_1"*>

<property name=*"name"* value=*"DJ"*></property>

</bean>

<bean id=*"service\_2"* class=*"com.aop.Service\_2"*>

<property name=*"name"* value=*"Ricardo"*></property>

</bean>

现有需求:在任何一个实现了ServiceInterface的类调用sayHello方法前，都将这一操作写入日志文件

传统的方法是:

ApplicationContext appContext=**new** ClassPathXmlApplicationContext("com/aop/aop.xml");

Service\_1 service\_1=(Service\_1)appContext.getBean("service\_1");

Service\_2 service\_2=(Service\_2)appContext.getBean("service\_2");

//service\_1执行写入日志操作

service\_1.sayHello();

//service\_2执行写入日志操作

service\_2.sayHello();

这样操作虽然没什么问题，但是非常麻烦，做不到自动化的程度，从上述代码可以看到，写入日志这一步操作，是可以作为公共部分提取出来的

aop通知表示在方法执行前后需要执行的动作。实际上它是Spring AOP框架在程序执行过程中触发的一些代码，spring中一共有五种通知：

**· before(前置通知)：在一个方法之前执行的通知。**

**· after(最终通知)：当某连接点(切面)退出的时候执行的通知(不论是正常返回还是异常退出)。**

**· after-returning(后置通知)：在某连接点正常完成后执行的通知。**

**· after-throwing(异常通知)：在方法抛出异常退出时执行的通知。**

**· around(环绕通知)：在方法调用前后触发的通知。**

|  |  |  |
| --- | --- | --- |
| 通知类型 | 实现的接口 | 描述 |
| Around 环绕通知 | org.aopalliance.intercept.MethodInterceptor | 在方法调用前后触发的通知。 |
| Before 前置通知 | org.springframework.aop.MethodBeforeAdvice | 在目标方法调用前调用 |
| After  后置通知 | org.springframework.aop.AfterReturningAdvice | 在目标方法调用后调用 |
| Throws 异常通知 | org.springframework.aop.ThrowsAdvice | 当目标方法抛出异常时调用 |

## 4.3 AOP详解

### 4.3.1 代理对象

代理对象和真实对象(即被代理的对象)实现一样的接口，真实对象负责实现我们的业务逻辑，如上例中service\_1和service\_2这两个对象就是真实对象，他们负责实现业务逻辑——sayHello。

代理对象内部含有对真实对象的引用，从而可以操作真实对象，同时代理对象实现与真实对象相同的接口以便在任何时刻都能代替真实对象。同时，代理对象可以在执行真实对象操作时，附加其他的操作(又叫**通知**，如上例的写入日志)，相当于对真实对象进行封装。这样，真实对象就不需要关心非业务逻辑以外的操作了，一切交由代理对象完成

spring提供了一个bean——ProxyFactoryBean来帮助我们完成代理，在xml中我们需要配置三个bean：

1. **被代理对象(如service\_1和service\_2)**
2. **通知(spring提供了五种通知，如专门用于写入日志的类)，即业务逻辑以外的操作**
3. **代理对象(实现了ProxyFactoryBean的对象)**

下面针对上例来看一看如何配置代理

首先需要一个通知类，由于是调用sayHello前要写入日志，这里选择前置通知

**public** **class** ServiceBeforeAdvice **implements** MethodBeforeAdvice{

@Override

**public** **void** before(Method arg0, Object[] arg1, Object arg2)

**throws** Throwable {

// **TODO** Auto-generated method stub

System.*out*.println("方法： "+arg0.getName()+" 写入日志");

}

}

然后配置我们的xml文件：

<!-- 配置被代理的类 -->

<bean id=*"service\_1"* class=*"com.aop.Service\_1"*>

<property name=*"name"* value=*"DJ"*></property>

</bean>

<bean id=*"service\_2"* class=*"com.aop.Service\_2"*>

<property name=*"name"* value=*"Ricardo"*></property>

</bean>

<!-- 配置前置通知，就是刚才定义的类 -->

<bean id=*"MyPreNotice"* class=*"com.aop.ServiceBeforeAdvice"*></bean>

<!-- 配置代理对象，不需要专门写代理类，因为框架已经帮我们写好，直接在xml中配置即可 -->

<bean id=*"proxy"* class=*"org.springframework.aop.framework.ProxyFactoryBean"*>

<!-- 配置代理对象要代理的接口，因为代理对象必须实现和被代理对象一样的接口，因此必须配置 -->

<property name=*"proxyInterfaces"*>

<!-- 多数情况下被代理的接口不止一个，可能被代理的类实现了多个接口，因此用数组 -->

<list>

<value>com.aop.ServiceInterface</value>

</list>

</property>

<!-- 将通知bean通知织入到代理对象 -->

<property name=*"interceptorNames"*>

<!-- 多数情况下被织入的通知不止一个，因此用数组 -->

<list>

<value>MyPreNotice</value>

</list>

</property>

<!-- 配置想要代理的对象，这里只代理service\_1 -->

<property name=*"target"*>

<ref bean=*"service\_1"* />

</property>

</bean>

### 4.3.2 通知

通知就是切面，是我们编程中共有的部分，将它提取出来作为切面通过代理的方式织入需要使用的地方，就是面向切面编程

一般我们有五种通知

1. **MethodInterceptor(环绕通知)**

又叫拦截器，在方法调用前后触发的通知(实际上是嵌入到方法体最前面和最后面)，该接口需要实现方法：

**public** Object invoke(MethodInvocation invocation){

System.*out*.println("环绕通知，方法调用前");

//执行被代理的方法，这一句是中点，在中点的前后加入切面(通知)

Object result = invocation.proceed();

System.*out*.println("环绕通知，方法调用后");

//将被代理的方法返回，不然这里将会发生中断

**return** result;

}

注意：如果有前置通知和后置通知，执行顺序是：

**前置通知->环绕通知前半部分->环绕通知后半部分->后置通知**

1. **MethodBeforeAdvice(前置通知)**

在目标方法调用前调用，该接口需要实现方法：

**public void before(Method method, Object[] args, Object target)**

**method** 前置通知执行完后要执行的方法

**args** 传递给method方法的参数

**target** 目标对象，即被代理的对象，在上例中是*service\_1*

1. **AfterReturningAdvice(后置通知)**

在目标方法调用后调用，该接口需要实现方法：

**public void afterReturning(Object returnValue, Method method, Object[]args,Object target)**

**returnValue** 被代理方法的返回值，即后置通知前面执行的那个方法的返回值

**method** 被代理方法

**args** 传递给method方法的参数

**target** 目标对象，即被代理的对象，在上例中是*service\_1*

1. **ThrowsAdvice(异常通知)**

当目标方法抛出异常时调用，该接口需要实现方法

必须的实现两个方法中的一个：

**public void afterThrowing(Method method, Object[] args, Object target, RuntimeException throwable)**

**或者**

**public void afterThrowing(RuntimeException throwable)**

**method** 被代理方法

**args** 传递给method方法的参数

**target** 目标对象，即被代理的对象，在上例中是*service\_1*

**throwable** 被代理方法抛出的异常

### 4.3.2 自定义通知切入点

在上例中，我们代理了service\_1并织入了前置通知，因此代理对象在调用service\_1的sayHello方法之前会执行前置通知的相关处理，这里是我们前置通知的切入点，但同时也会引发一个新的问题，如果service\_1还有一个sayBye()方法，不希望在该方法执行前织入前置通知，该如何处理？(织入的通知默认对所有被代理对象的方法生效)

具体方法自行了解。。。AOP是spring的核心概念，但日常开发用的不多，在一些比较高级的地方会用的多

1. 注解

如果一味地只使用xml进行spring配置，那么.xml文件将会十分庞大，难以管理，为此Spring提供了大量的第三方注解供开发者使用，使用这些注解可以大幅提升我们的开发效率，接下来将对spring常用注解进行详细说明

**元注解：元注解是注解的注解，不能直接注解代码**

## 5.1依赖级别

Spring可以使用这类注解进行依赖注入，但需要注意的是，我们对bean的声明定义依然是在xml文件中进行，但是依赖bean的注入过程则通过这些注解帮助我们完成

### 5.1.1 @Autowired

@AutoWired是spring中最常用的注解之一，其作用是”自动装配依赖”，是基于bean的类型进行匹配，比如一个Fly类的依赖属性被@AutoWired注释后，会自动去xml中寻找Fly类型的bean织入依赖属性，如果有多个Fly类型的bean，可以使用@Qualifier来指定哪个bean将被织入，现在我们继续使用超人的例子：

首先我们有一个超能力接口

public interface Ability {  
 void ability();  
}

然后我们提供一个实现了超能力接口的飞行能力

public class Fly implements Ability  
{  
 //飞行速度  
 protected int speed;  
  
 public Fly(int speed) {  
 this.speed=speed;  
 }  
  
 public void ability() {  
 System.*out*.println("超人能以"+this.speed+"/s的速度飞行");  
 }  
}

接下来是我们的超级英雄：

public class SuperMan {  
 private Ability ability;  
  
 public Ability getAbility() {  
 return ability;  
 }  
   
 public void setAbility(Ability ability) {  
 this.ability = ability;  
 }  
  
 public void useAbility(){  
 this.ability.ability();  
 }  
}

#### 5.1.1.1 用于setter方法

@AutoWired用于setter方法，可以摆脱bean的<property>元素，@AutoWired可以自动将指定的bean作为依赖注入到setter方法的参数中，eg：

public class SuperMan {  
 private Ability ability;  
  
 public Ability getAbility() {  
 return ability;  
 }  
 @Autowired  
 public void setAbility(@Qualifier("ability")Ability ability) {  
 this.ability = ability;  
 }  
  
 public void useAbility(){  
 this.ability.ability();  
 }  
}

我们创建了一个超能力bean和一个超人bean，超能力bean被@Autowired注解通过setter方法自动织入超人bean中，只需要加一个@Autowired注解就能达到这种效果，方便高效

<bean id="ability" class="com.po.Fly">  
 <constructor-arg name="speed" value="1000"></constructor-arg>  
</bean>  
<bean id="superMan" class="com.po.SuperMan"></bean>

#### 5.1.1.2 用于构造方法

同样的，对于构造函数，我们也可以这么做，这样可以帮助我们摆脱<constructor-arg>元素

public class SuperMan {  
 private Ability ability;  
  
 @Autowired  
 public SuperMan(Ability ability) {  
 this.ability = ability;  
 }  
  
 public Ability getAbility() {  
 return ability;  
 }  
  
 public void setAbility(Ability ability) {  
 this.ability = ability;  
 }  
  
 public void useAbility(){  
 this.ability.ability();  
 }  
}

<bean id="ability" class="com.po.Fly">  
 <constructor-arg name="speed" value="1000"></constructor-arg>  
</bean>  
<bean id="superMan" class="com.po.SuperMan"></bean>

#### 5.1.1.3 用于依赖属性

当@Autowired注解用于依赖属性时，可以帮助我们摆脱setter方法，即不需要setter方法就可以将依赖注入

public class SuperMan {  
 @Autowired  
 private Ability ability;  
  
 public void useAbility(){  
 this.ability.ability();  
 }  
}

<bean id="ability" class="com.po.Fly">  
 <constructor-arg name="speed" value="1000"></constructor-arg>  
</bean>  
<bean id="superMan" class="com.po.SuperMan"></bean>

#### 5.1.1.4 @Autowired (required=false)

@Autowired注解在使用时默认附带@Required注解的效果，即要织入的bean必须在xml已声明(能找到)，

<!-- <bean id="ability" class="com.po.Fly">  
 <constructor-arg name="speed" value="1000"></constructor-arg>  
 </bean>-->  
 <bean id="superMan" class="com.po.SuperMan">  
 <property name="a" value="1"/>  
 </bean>

由于我们注释了SuperMan的依赖ability，导致报错，但是可以通过关闭required检查来避免报错

public class SuperMan {  
 @Autowired(required=false)  
 private Ability ability;  
  
 public void useAbility(){  
 this.ability.ability();  
 }  
}

### 5.1.2 @Required

@Required注解将会检查类的依赖，是否已经在xml中设置好，如果依赖还未在xml中设置，就会报错

**请注意！该注解只能用在setter方法上！**

public class SuperMan {  
  
 private int a;  
  
 public int getA() {  
 return a;  
 }  
 @Required  
 public void setA(int a) {  
 this.a = a;  
 }  
}

如果bean中未配置依赖a，@Required会使其报错

<bean id="superMan" class="com.po.SuperMan">

<property name="a" value="1"/>  
</bean>

### 5.1.3 @Qualifier

@Autowired是根据类型进行自动装配的。如果当Spring上下文中存在不止一个UserDao类型的bean，并且bean的id并未和依赖的属性名保持一致时，就会抛出异常，@Qualifier可以帮助我们解决这一问题

<bean id="flya" class="com.po.Fly">  
 <constructor-arg name="speed" value="1000"></constructor-arg>  
</bean>

<bean id="flyb" class="com.po.Fly">  
 <constructor-arg name="speed" value="100"></constructor-arg>  
</bean>  
<bean id="superMan" class="com.po.SuperMan"/>

通过@Qualifier指定bean的id

public class SuperMan {  
 @Autowired(required=false)  
 @Qualifier("flyb")  
 private Ability ability;  
   
 public void useAbility(){  
 this.ability.ability();  
 }  
}

### 5.1.4 @Resource

@Resource与@Autowired一样都可以用来装配bean. 都可以写在字段上,或写在setter方法上，然而二者也有不同之处：

@Autowired默认**按类型装配(byType)**，是spring提供的注解

@Resource默认**按name属性装配(byName)**，是jdk 1.6提供的注解，他默认将会按照bean的name属性进行装配。如果没有指定name属性，当注解写在字段上时，默认取字段名，按照名称(byName)查找，如果注解写在setter方法上默认取属性名进行装配。当找不到与name名称匹配的bean时才按照类型(byType)进行装配。

eg：

public class SuperMan {  
 @Resource(name = "bbb")  
 private Ability ability;  
  
 public void useAbility(){  
 this.ability.ability();  
 }  
}

<bean id="flya" class="com.po.Fly" name="aaa">  
 <constructor-arg name="speed" value="1000"></constructor-arg>  
</bean>  
<bean id="flyb" class="com.po.Fly" name="bbb">  
 <constructor-arg name="speed" value="100"></constructor-arg>  
</bean>  
<bean id="superMan" class="com.po.SuperMan"/>

### 5.1.5 @Value

@value注解可以帮助引入配置项，避免硬编码，比如我们希望引入一个已经被加载进spring配置文件中的property文件中的属性时，可以通过@value注解

public class SuperMan {

//引入db.properties文件中的jdbc.url属性  
 @Value("${jdbc.url}")  
 private String url;  
 @Autowired  
 @Qualifier("flya")  
 private Ability ability;  
  
 public void useAbility(){  
 this.ability.ability();  
 }  
 public void out(){  
 System.*out*.println(this.url);  
 }  
}

## 5.2 bean属性级别

Spring可以使用这类注解进行依赖注入，但需要注意的是，我们对bean的声明定义依然是在xml文件中进行，但是依赖bean的注入过程则通过这些注解帮助我们完成

### 5.2.1 @lazy懒加载

主要用于修饰Spring Bean类，用于决定是否要预初始化该Bean。如果配置为true，表示该bean是懒加载模式，什么时候用到了该bean才会进行初始化。

@Lazy(true)  
@Component  
public class Chinese implements Person{  
 //codes here  
}

### 5.2.2 @scope作用域

主要用于修饰Spring Bean类，用于配置bean的作用域，默认是单例模式（singleton），如果需要设置的话可以修改值：

**singleton**：单例模式，在整个Spring IoC容器中，使用singleton定义的Bean将只有一个实例

**prototype**：原型模式，每次通过容器的getBean方法获取prototype定义的Bean时，都将产生一个新的Bean实例

**request**：对于每次HTTP请求，使用request定义的Bean都将产生一个新实例，即每次HTTP请求将会产生不同的Bean实例。只有在Web应用中使用Spring时，该作用域才有效

**session**：对于每次HTTP Session，使用session定义的Bean都将产生一个新实例。同样只有在Web应用中使用Spring时，该作用域才有效

**globalsession**：每个全局的HTTP Session，使用session定义的Bean都将产生一个新实例。典型情况下，仅在使用portlet context的时候有效。同样只有在Web应用中使用Spring时，该作用域才有效

springMVC是基于方法的拦截，而Struts2是基于类的拦截。

这也就意味着每次处理一个请求，struts2就会实例化一个对象；这样就不会有线程安全的问题了。

而spring的controller默认是Singleton(单例)的，这意味着每一个request过来，系统都会用同一个controller实例去处理，这样导致两个结果：一是我们不用每次创建Controller，减少了对象创建和垃圾收集的时间; 二是由于只有一个Controller的instance，当多个线程调用它的时候，它里面的instance变量就不是线程安全的了，会发生窜数据的问题。

当然大多数情况下，我们根本不需要考虑线程安全的问题，比如在controller中织入dao,service等，除非在bean中声明了实例变量。因此，我们在使用spring mvc 的contrller时，应避免在controller中定义实例变量。但是如果使用了实例变量，为了防止实例变量线程不安全，这里有两种方式来处理：

1、在Controller中使用ThreadLocal变量  
2、在Controller类上声明 scope="prototype"，即每次都创建新的controller。

### 5.2.3 @PostConstruct和@PreDestroy

**这两个注解都必须用在方法上**。用于指定构造之后执行的方法 和 销毁之前执行的方法，

public class SuperMan {  
 @Autowired  
 @Qualifier("flya")  
 private Ability ability;  
  
 public SuperMan() {  
 System.*out*.println("构造方法执行");  
 }  
  
 public void useAbility() {  
 this.ability.ability();  
 }  
  
 @PostConstruct  
 public void init() {  
 System.*out*.println("@PostConstrut执行");  
 }  
  
 @PreDestroy  
 public void dostory() {  
 System.*out*.println("@PreDestroy执行");  
 }  
}

以上代码执行的结果为：

构造方法执行

@PostConstrut执行

超人能以1000/s的速度飞行(useAbility执行的结果)

@PreDestroy执行

## 5.3 bean级别

@Component、@Repository、@Service、@Controller都是用来自动注册bean的，之所以定义了这么多的类型是想让你在使用的时候就根据这个bean的定义就知道这个bean的作用。他们的区别仅在于语义不同(作用相同)：

@Service用于标注业务层组件

@Controller用于标注控制层组件（如struts中的action）

@Repository用于标注数据访问组件，即DAO组件

@Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。

### 5.3.1 @Repository

它用于将数据访问层 (DAO 层 ) 的类注册为 Spring Bean。具体只需两步，第一步将该注解标注在 DAO类上，第二步在spring配置文件中添加扫描注解配置，如需扫描多个包可以用逗号分隔

package com.po;

@Repository("flya")  
public class Fly implements Ability  
{  
 //飞行速度  
 protected int speed;  
  
 public Fly() {  
 this.speed=100000;  
 }  
  
 public void ability() {  
 System.*out*.println("超人能以"+this.speed+"/s的速度飞行");  
 }  
}

<context:component-scan base-package="com.po"></context:component-scan>

### 5.3.2 @Service

它用于将业务层 (Service层 ) 的类注册为 Spring Bean。具体只需两步，第一步将该注解标注在 Service类上，第二步在spring配置文件中添加扫描注解配置，如需扫描多个包可以用逗号分隔

package com.po;

@Service("flya")  
public class Fly implements Ability  
{  
 //飞行速度  
 protected int speed;  
  
 public Fly() {  
 this.speed=100000;  
 }  
  
 public void ability() {  
 System.*out*.println("超人能以"+this.speed+"/s的速度飞行");  
 }  
}

<context:component-scan base-package="com.po"></context:component-scan>

### 5.3.3 @Controller

@Controller 用于标记在一个类上，使用它标记的类就是一个SpringMVC Controller 对象。通常与@RequestMapping一起使用

例子同上

### 5.3.4 @Component

泛指组件，当组件不好归类(上面3类)的时候，我们可以使用这个注解进行标注。

## 5.4 容器级别

### 5.3.4 @Configuration和@Bean

这两个注解可以配合使用，用以代替spring的xml配置文件，在@Configuration出现之前，配置信息只能放在xml配置文件中。AnnotationConfigApplicationContext 搭配上 @Configuration 和 @Bean 注解出现后， XML 配置方式不再是 Spring IoC 容器的唯一配置方式。一个被@Configuration标注的类就是一个xml配置文件

@Configuration标注在类上，相当于把该类作为spring的xml配置文件中的<beans>，而@Bean则相当于<bean>标签，

@Configuration  
public class TestConfiguration {  
 public TestConfiguration(){  
 System.*out*.println("spring容器启动初始化。。。");  
 }  
  
 //@Bean注解注册bean,同时可以指定初始化和销毁方法  
 //@Bean(name="testNean",initMethod="start",destroyMethod="cleanUp")  
 @Bean  
 @Scope("prototype")  
 public TestBean testBean() {  
 return new TestBean();  
 }  
}

ApplicationContext context = new AnnotationConfigApplicationContext(TestConfiguration.class);

1. 事务管理

## 6.1 事务的特性

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。

我们都知道，事务应当具有以下特性：

原子性:事务不可分割

一致性:事务执行的前后，数据完整性保持一致.

隔离性:一个事务执行的时候，不应该受到其他事务的打扰

持久性:一旦结束，数据就永久的保存到数据库.

## 6.2 事务隔离级别

如果不考虑事务隔离性，则会产生以下现象：

**(1)丢失更新：**两个事务同时更新一行数据，最后一个事务的更新会覆盖掉第一个事务的更新，从而导致第一个事务更新的数据丢失，这是由于没有加任何锁造成的；

**(2)脏读**:一个事务读到另一个事务未提交数据，后一个事务如果操作失败进行了回滚，前一个事务读取的就是错误数据，这样就造成了脏读。

**(3)不可重复读**: 事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致。**关注点在于修改**，

**(4)虚读(幻读)**:一个事务多次读取一组数据，另一个事务在第一个事务的两次读取操作之间提交了数据（insert），导致一个事务两次查询结果条数不一致。**关注点在于新增和删除**，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样。

数据库提供了四种事务隔离级别：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 事务隔离级别 | 丢失更新 | 脏读 | 不可重复读 | 幻读 |
| 读未提交（read-uncommitted） | 否 | **是** | **是** | **是** |
| 不可重复读（read-committed） | 否 | 否 | **是** | **是** |
| 可重复读（repeatable-read） | 否 | 否 | 否 | **是** |
| 串行化（serializable） | 否 | 否 | 否 | 否 |

1. **读未提交 Read UnCommitted**：

**实现原理：**通过**一级封锁协议**实现

**修改操作：修改前必须加排他写(X)锁，直到事务结束(rollback或commit)。**

**读取操作：不加任何锁。**

由于读取不加锁，读取操作不会与任何操作(包括修改)发生锁冲突，这可能导致**脏读、不可重复读、幻读**现象

**解决了丢失更新的问题。这可能会造成脏读、不可重复读、幻读现象**

1. **读已提交(sqlServer、Oracle默认级别)** **Read Committed**：

**实现原理：**通过**二级封锁协议**实现

**修改操作：修改前必须加排他写(X)锁，直到事务结束(rollback或commit)**

**读取操作：读取前必须加共享读(S)锁，读完立即释放，不必等到读取事务结束。**

因为X锁会和S锁冲突，执行修改操作的事务结束之前，其他事务不能进行读取，解决了**脏读**。

但由于**共享读(S)锁**读取完立即释放，事务A两次读取数据，加了两次S读锁，锁不连续，在两次加锁之间，事务B加了X写锁进行修改操作，可能导致**不可重复读**、**幻读**现象

**解决了丢失更新、脏读，但不可重复读和幻读仍可发生**

1. **可重复读(MySql默认级别) Repeatable Read**：

**实现原理：**通过三**级封锁协议**实现

**修改操作：修改前必须加排他写(X)锁，直到事务结束(rollback或commit)**

**读取操作：读取前必须加共享读(S)锁，直到事务结束(rollback或commit)。**

因为读取操作加S读锁直到事务结束，因此直到读取事务结束之前，修改事务不能进行修改，解决了**不可重复读。**

但由于读取操作没有对整张表加锁，因此在两次读取之间，虽然可能导致**幻读**现象

**解决了丢失更新、脏读、不可重复读，但幻读仍可发生**

1. **序列化 serializable**：

仅仅通过“行级锁”是无法实现事务序列化的，应该对整张表加锁(锁表)

事务隔离的最高级别，提供严格的事务隔离，它要求事务串行执行，事务只能一个接着一个地执行，但不能并发执行。会引发超时和锁竞争，耗资源多、损失并行度、性能低，一般不用

**解决了所有可能出现的事务并发问题**

**如果仅仅把事务级别设为最高，就会带来并发的性能问题**，数据库的事务隔离级别越严格，并发副作用越小，但付出的代价也就越大，因为事务隔离实质上就是使事务在一定程度上“串行化”进行，这显然与“并发”是矛盾的，同时，不同的应用对读一致性和事务隔离程度的要求也是不同的，比如许多应用对“不可重复读”和“幻读”并不敏感，可能更关心数据并发访问的能力。因此，我们应当灵活应用事务隔离的特性来平衡＂隔离＂与＂并发＂的矛盾

## 6.3 spring事务传播

我们在使用事务时，一般是在Service层使用，当我们调用service层的方法时，它可以保证我们对数据库的所有操作都在一个事务中(原子性)，在同一个事务中调用的所有DAO层方法，要么同时成功，要么同时失败，这是我们都知道的，那么如果我们调用一个service层的方法，该方法不止调用了DAO层方法，也调用了其他service方法，也就是**事务方法和事务方法发生嵌套调用**。Spring通过事务传播行为，控制当前的事务如何传播到被嵌套调用的目标服务接口方法中。

需要注意的是：

**事务传播机制只适用于不同bean之间方法的调用，如果一个bean中的两个方法互相调用并不会使用到事务传播。**比如，一个bean的method1的事务传播级别为Required，method2的事务传播级别为Never，我们在method1里面调用method2。首先method1会开启一个事务，而method2也没有报错并正确执行了，而且也在method1开启的事务之中，说明事务传播机制在一个bean自己的方法互相调用中并不起作用，只要一个方法开启了事务，那这个在方法里调用当前bean的其他方法都在这个事务中运行，而不管其他方法的事务传播机制是如何配置的。

spring规定了7种类型的事务传播行为：

假设现在有两个service层的bean：

service1{

方法a

}

service2{

方法b

}

我们试图在方法a中调用方法b

|  |  |  |
| --- | --- | --- |
| 分类 | 事务传播行为类型 | 说明 |
| **如果a方法中有事务，都会使用a中的事务** | **PROPAGATION\_REQUIRED(默认)** | 如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是最常见的选择。  即如果a中有事务，b就使用a中的事务，如果a中没有事务，则B创建一个事务并把a剩下的代码包进去，也就是说即使方法b已经结束，但是a接下来的代码抛出异常要回滚，b也会被回滚 |
| **PROPAGATION\_SUPPORTS** | 支持当前事务，如果当前没有事务，就以非事务方式执行。  即如果a中有事务，b就使用a中的事务，如果a中没有事务，则b不使用事务 |
| **PROPAGATION\_MANDATORY** | 使用当前的事务，如果当前没有事务，就抛出异常。  即如果a中有事务，b就使用a中的事务，如果a中没有事务，则抛出异常 |
| **a方法不会和b方法在同一个事务中执行** | **PROPAGATION\_REQUIRES\_NEW** | 新建事务，如果当前存在事务，把当前事务挂起。  即B总是创建一个新的事务，即使a中有事务，也挂起(暂停)a事务创建一个新的事务。这样一来，如果b已提交，a失败回滚，b是不会回滚的。如果b失败回滚，如果其异常被捕获，a仍可提交 |
| **PROPAGATION\_NOT\_SUPPORTED** | 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。  即B总是以非事务方式执行，即使a中有事务，也挂起(暂停)a事务，B继续非事务执行 |
| **PROPAGATION\_NEVER** | 以非事务方式执行，如果当前存在事务，则抛出异常。  即B总是以非事务方式执行，如果a中有事务则抛出异常 |
| **嵌套事务** | **PROPAGATION\_NESTED** | 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。  即如果a没事务，则新建一个事务，如果a有事务这把b的事务当做a的子事务。如果是a有事务，则是嵌套事务，父事务(a)失败回滚，则子事务(b)也回滚，如果子事务(b)失败回滚，则父事务可以回滚到之前设置好的保存点(savepoint)上，这样父事务还可以继续尝试使用别的事务方法来完成自己的任务 |

## 6.4 spring事务管理

spring的事务管理中，有三个重要的接口TransactionDefinition、PlatformTransactionManager和TransactionStatus

### 6.4.1 TransactionDefinition接口

在spring中，事务是通过 TransactionDefinition 接口来定义的。该接口包含与事务属性有关的方法。

package org.springframework.transaction;

import org.springframework.lang.Nullable;

public interface TransactionDefinition {  
 int PROPAGATION\_REQUIRED = 0;  
 int PROPAGATION\_SUPPORTS = 1;  
 int PROPAGATION\_MANDATORY = 2;  
 int PROPAGATION\_REQUIRES\_NEW = 3;  
 int PROPAGATION\_NOT\_SUPPORTED = 4;  
 int PROPAGATION\_NEVER = 5;  
 int PROPAGATION\_NESTED = 6;  
 int ISOLATION\_DEFAULT = -1;  
 int ISOLATION\_READ\_UNCOMMITTED = 1;  
 int ISOLATION\_READ\_COMMITTED = 2;  
 int ISOLATION\_REPEATABLE\_READ = 4;  
 int ISOLATION\_SERIALIZABLE = 8;  
 int TIMEOUT\_DEFAULT = -1;  
  
 int getPropagationBehavior();  
 int getIsolationLevel();  
 int getTimeout();  
 boolean isReadOnly();  
  
 @Nullable  
 String getName();  
}

该接口没有提供任何设置属性的方法，只有获取属性的方法，这是因为设置属性的方法可以由我们任意定义，并且保存属性的字段也没有任何要求，而读取属性的方法必须固定，Spring 进行事务操作的时候，必须通过调用以上接口提供的读取方法返回事务相关的属性值。

上述方法有4个属性读取方法

1. **getPropagationBehavior** 获取事务传播行为属性

2. **getIsolationLevel** 获取事务隔离级别属性，常量**ISOLATION\_DEFAULT**意味着将使用数据库中默认的隔离级别

3. **getTimeout** 获取事务超时属性,所谓事务超时，就是指一个事务所允许执行的最长时间，如果超过该时间限制但事务还没有完成，则自动回滚事务。在 TransactionDefinition 中以 int 的值来表示超时时间，其单位是秒。

4.**isReadOnly** 是否为只读事务，如果一次事务操作由多个读操作组成，多条查询SQL必须保证整体读一致性。否则，在前条SQL查询之后，后条SQL查询之前，数据被其他用户改变，则该次整体的查询将会出现读取的数据不一致的状态。此时，应该使用只读事务，由于只读事务不存在数据的修改，因此数据库将会为只读事务提供一些优化手段。一旦尝试在只读事务中进行写操作，就会抛出异常

spring为我们提供了一个该接口的实现类**DefaultTransactionDefinition**，该类适用于大多数情况。如果该类不能满足需求，可以通过实现 TransactionDefinition 接口来实现自己的事务定义。

### 6.4.2 PlatformTransactionManager接口

PlatformTransactionManager 用于执行具体的事务操作

public interface PlatformTransactionManager {  
 TransactionStatus getTransaction(@Nullable TransactionDefinition var1) throws TransactionException;  
 void commit(TransactionStatus var1) throws TransactionException;  
 void rollback(TransactionStatus var1) throws TransactionException;  
}

根据所使用的持久层框架不同，PlatformTransactionManager接口提供了不同的实现类：

* **DataSourceTransactionManager**：适用于使用JDBC和MyBatis进行数据持久化操作的情况。
* **HibernateTransactionManager**：适用于使用Hibernate进行数据持久化操作的情况。
* **JpaTransactionManager**：适用于使用JPA进行数据持久化操作的情况。
* 另外还有**JtaTransactionManager** 、**JdoTransactionManager**、**JmsTransactionManager**等等。

### 6.4.3 TransactionStatus接口

通过TransactionStatus接口可以简单的控制事务执行和查询事务状态，上一个接口PlatformTransactionManager的**getTransaction**方法将会返回一个TransactionStatus接口对象

package org.springframework.transaction;  
import java.io.Flushable;  
  
public interface TransactionStatus extends SavepointManager, Flushable {  
 boolean isNewTransaction();  
 boolean hasSavepoint();  
 void setRollbackOnly();  
 boolean isRollbackOnly();  
 void flush();  
 boolean isCompleted();  
}

### 6.4.4编程式事务管理

编程式事务管理需要我们在程序中手动调用beginTransaction()、commit()、rollback()等方法来进行事务管理，**实际应用较少**

下面将以转账为案例讲解编程式事务管理

FundMapper fundMapper=(FundMapper)ap.getBean("fundMapper");  
//转出  
int result1=fundMapper.outAccount(money,outerName);  
int a=2/0;  
//转入  
int result2=fundMapper.inAccount(money,inerName);

outAccount将从outerName的账户中转出money元，同时inAccount将为inerName的账户中转入money元，这是一个典型的事务操作，在多数情况下，这两个操作都会成功，但并不是绝对同时成功，我们人为在中间加入一个RuntimeException，这会导致前一个操作成功，后一个操作由于抛出异常而无法执行

接下来，我们将会用编程式事务管理对这段代码进行修改

在spring配置文件中配置：

<!--配置PlatformTransactionManager -->  
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dataSource"></property>  
</bean>

FundMapper fundMapper=(FundMapper)ap.getBean("fundMapper");  
//编程式事务管理开启  
DefaultTransactionDefinition tsDenfine=new DefaultTransactionDefinition();  
//可重复读  
tsDenfine.setIsolationLevel(TransactionDefinition.*ISOLATION\_REPEATABLE\_READ*);  
//设置事务传播行为  
tsDenfine.setPropagationBehavior(TransactionDefinition.*PROPAGATION\_REQUIRED*);  
//获取事务管理器  
PlatformTransactionManager dstManager=(PlatformTransactionManager)ap.getBean("transactionManager");  
//获取事务状态,之后，Spring将根据传播行为来决定如何开启事务  
TransactionStatus transactionStatus=dstManager.getTransaction(tsDenfine);  
//-------------------事务开始-------------------  
try {  
 //转出  
 fundMapper.outAccount(money,outerName);  
 int a=2/0;  
 //转入  
 fundMapper.inAccount(money,inerName);  
 //成功执行，提交事务  
 dstManager.commit(transactionStatus);  
 return true;  
//编程式事务管理极其灵活，可以在这里指定某种异常出现时才回滚  
}catch (RuntimeException e){  
 dstManager.rollback(transactionStatus);  
 return false;  
}  
//-------------------事务结束-------------------

PS:请注意数据库引擎不要用MyISAM…MyISAM不支持事务管理….

但是这样也有相当大的弊端，每一个业务方法都包含了类似的启动事务、提交/回滚事务的样板代码，加大了代码的冗余度。为了解决这个问题，spring提供了TransactionTemplate，可以通过回调方式来进行编程式事务管理，这里可以使用默认的事务提交和回滚规则，这样在业务代码中就不需要显式调用任何事务管理的 API。我们可以在方法的任何位置调用该参数的 setRollbackOnly() 方法将事务标识为回滚的，以执行事务回滚。

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dataSource"></property>  
</bean>  
<!-- 配置transactionTemplate -->  
<bean id="transactionTemplate" class="org.springframework.transaction.support.TransactionTemplate">  
 <property name="transactionManager" ref="transactionManager"></property>  
 <!--定义事务隔离级别,-1表示使用数据库默认级别-->  
 <property name="readOnly" value="false"></property>  
 <property name="isolationLevelName" value="ISOLATION\_DEFAULT"></property>  
 <property name="propagationBehaviorName" value="PROPAGATION\_REQUIRED"></property>  
</bean>

TransactionTemplate transactionTemplate=(TransactionTemplate)ap.getBean("transactionTemplate");  
return (Boolean) transactionTemplate.execute(new TransactionCallback() {  
 public Object doInTransaction(TransactionStatus status) {  
 FundMapper fundMapper=(FundMapper)ap.getBean("fundMapper");  
 Object result;  
 try {  
 //转出  
 fundMapper.outAccount(money,outerName);  
 int a=2/0;  
 //转入  
 fundMapper.inAccount(money,inerName);  
 //成功执行，提交事务  
 return true;  
 }catch (RuntimeException e){  
 status.setRollbackOnly();  
 return false;  
 }

#### 6.1.4.5声明式事务管理

声明式事务管理通过xml配置进行事务管理，原理也是通过aop实现的，本质上是对目标方法的执行前和执行后进行拦截，然后在目标方法开始之前创建或者加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。其代码侵入性较小，方便后期维护，推荐使用

和编程式事务相比，声明式事务唯一不足地方是，后者的最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。但是即便有这样的需求，也存在很多变通的方法，比如，可以将需要进行事务管理的代码块独立为方法等等。

**注意：声明式事务管理中不要使用catch块将异常捕获！否则不会触发自动回滚**

##### 6.4.5.1基于tx和aop名字空间的xml配置文件

这是一种大型web应用中常见的声明式事务控制方式，tx标签可以配置事务增强，配置事务的隔离级别、传播行为、回滚异常、切入方法等。aop标签将配置事务的切入点

<tx:advice id="transferAdvice" transaction-manager="transactionManager">  
 <tx:attributes>  
 <!-- propagation:事务传播行为  
 isolation:事务隔离级别  
 read-only:只读  
 rollback-for：发生哪些异常回滚  
 no-rollback-for:发生哪些异常不会滚  
 timeout:过期信息  
 -->  
 <tx:method name="transfer\*" propagation="REQUIRED"  
 rollback-for="java.lang.Exception,java.lang.RuntimeException"/>  
 </tx:attributes>  
</tx:advice>  
<!-- 配置切面&ndash;&gt;-->  
<aop:config>  
 <!--&lt;!&ndash; 配置切入点，指定哪些方法需要被执行AOP &ndash;&gt;-->  
 <aop:pointcut id="transferPoint"  
 expression="execution(\* com.service.\*.\*(..))"></aop:pointcut>  
 <!-- &lt;!&ndash;配置切面&ndash;&gt;-->  
 <aop:advisor advice-ref="transferAdvice" pointcut-ref="transferPoint"/>  
</aop:config>

expression="execution(\* com.service.\*.\*(..))"需要重点注意：

第一个\*代表任意的返回类型

第二个\*代表service包下的所有的类

第三个\*代表service包下类的所有方法

..则代表0个或多个参数

常见切入点表达式的例子：

任意公共方法的执行：

execution(public \* \*(..))

任何一个以“set”开始的方法的执行：

execution(\* set\*(..))

AccountService 接口的任意方法的执行：

execution(\* com.xyz.service.AccountService.\*(..))

定义在service包里的任意方法的执行：

execution(\* com.xyz.service.\*.\*(..))

定义在service包或者子包里的任意类的任意方法的执行：

execution(\* com.xyz.service..\*.\*(..))

### 6.4.5基于@Transactional注解的事务管理

基于注解的配置就相当简单了：

<!--配置PlatformTransactionManager -->  
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dataSource"></property>  
</bean>  
  
<tx:annotation-driven transaction-manager="transactionManager"/>

然后在业务层类前面加上注解(不推荐在方法级别上加上注解)

@Transactional(propagation = Propagation.*REQUIRED*)  
public class TransferAccountService {  
@Autowired  
private FundMapper fundMapper;  
 public TransferAccountService() {

}  
 public boolean transferMoney(final String outerName, final String inerName, final int money) {  
 //转出  
 fundMapper.outAccount(money,outerName);  
 int a=2/0;  
 //转入  
 fundMapper.inAccount(money,inerName);  
 //成功执行，提交事务  
 return true;

}

1. Spring定时任务

定时任务在日常的项目中不可或缺，而现有三种实现定时任务的技术：

1.Java自带的java.util.Timer类，这个类允许你调度一个java.util.TimerTask任务。使用这种方式可以让你的程序按照某一个频度执行，但不能在指定时间运行。一般用的较少，不做详细介绍。

2.使用Quartz，这是一个功能比较强大的的调度器，可以让你的程序在指定时间执行，也可以按照某一个频度执行，配置起来稍显复杂，稍后会详细介绍。

3.Spring3.0以后自带的task，可以将它看成一个轻量级的Quartz，而且使用起来比Quartz简单许多，稍后会介绍。

触发定时任务的方式有两种，一是每隔指定时间则触发一次，另一种是每到指定时间则触发一次

下面将结合spring讲解如何使用spring定时任务

## 7.1 任务调度工具Quartz

Quartz支持两种声明方式，一种是继承方式，一种是非继承方式，这里推荐非继承方式，也只讨论非继承方式

添加依赖

<dependency>  
 <groupId>org.quartz-scheduler</groupId>  
 <artifactId>quartz</artifactId>  
 <version>${quartzVersion}</version>  
</dependency>

1. 编写非继承的定时任务类

package com.util;

public class MyQuartz {  
 //执行定时任务  
 public void doTimeTask() {  
 System.*out*.println("正在执行定时任务！");  
 }  
}

1. 配置定时任务类

<bean id="timeTask" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">

<property name="targetObject">  
 <bean class="com.util.MyQuartz"></bean>  
 </property>  
 <property name="targetMethod" value="doTimeTask"></property>  
 <!-- 定时作业不并发调度 -->  
 <property name="concurrent" value="false" />  
</bean>

关键在于指出自定义的定时任务类(targetObject)和方法(targetMethod)

1. 配置触发方式

上面有说到，触发器有两种触发方式，分别是：

**org.springframework.scheduling.quartz.SimpleTriggerBean 每隔指定时间则触发一次**

**org.springframework.scheduling.quartz.CronTriggerBean 每到指定时间则触发一次**

<bean id="simpleTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerFactoryBean">

<property name="jobDetail" ref="timeTask"/>  
 <!-- 调度工厂实例化后，经过5秒开始执行调度 -->  
 <property name="startDelay" value="5000"/>  
 <!-- 每5秒调度一次 -->  
 <property name="repeatInterval" value="5000"/>  
</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean">

<property name="jobDetail" ref="timeTask" />  
 <!--每天12:00运行一次 -->  
 <property name="cronExpression" value="0 0 12 \* \* ?" />  
</bean>

1. 配置调度工厂

<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">

<property name="triggers">  
 <list>  
 <ref bean="simpleTrigger" />  
 </list>  
 </property>  
</bean>

这里以SimpleTriggerBean为例

1. 启动tomcat

## 7.2 轻量级Spring-Task

上述的Quartz配置较为麻烦，是重量级工具，在spring3.0以后，提供了自主开发的定时任务工具，spring-task，可以将它比作一个轻量级的Quartz，而且使用起来很简单，除spring相关的包外不需要额外的包，而且支持注解和配置文件两种

### 配置文件方式配置

* 1. 编写定时任务类

@Service

public class MyQuartz {  
 //执行定时任务  
 public void doTimeTask() {  
 System.*out*.println("正在执行定时任务！");  
 }  
}

* 1. 添加spring配置文件的命名空间和描述

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

* 1. 注册定时任务

<!--扫描并注册自定义定时任务类-->  
<context:component-scan base-package="com.util"/>  
<task:scheduled-tasks>

<!--"0 \* \* \* \* ?"这里表示的是每次0秒也就是一分钟执行一次-->  
 <task:scheduled ref="myQuartz" method="doTimeTask" cron="0 \* \* \* \* ?"/>  
</task:scheduled-tasks>

接下来只需要运行tomcat即可

### 注解方式配置

@Scheduled注解是spring-task配置定时任务的注解，有三个属性：

1. cron：指定cron表达式
2. fixedDelay：从上一个任务完成开始到下一个任务开始的间隔，单位是毫秒。
3. fixedRate：从上一个任务开始到下一个任务开始的间隔，单位是毫秒。
4. 编写定时任务类

@Component  
public class MyQuartz {  
 //执行定时任务  
 @Scheduled(cron = "0 0 3 \* \* ?")  
 public void doTimeTask() {  
 System.*out*.println("正在执行定时任务！");  
 }  
}

1. Spring配置

<!--spring扫描定时任务类的配置 -->  
<context:component-scan base-package="com.util" />  
<task:scheduler id="qbScheduler" pool-size="10"/>  
<!--开启这个配置，spring才能识别@Scheduled注解 -->  
<task:annotation-driven scheduler="qbScheduler" mode="proxy"/>

接下来启动tomcat即可

## 7.3 cron表达式的配置规则

cron表达式的配置示例如下：

|  |
| --- |
| 字段 允许值 允许的特殊字符 |
| 秒 0-59 , - \* / |
| 分 0-59 , - \* / |
| 小时 0-23 , - \* / |
| 日期 1-31 , - \* ? / L W C |
| 月份 1-12 或者 JAN-DEC , - \* / |
| 星期 1-7 或者 SUN-SAT , - \* ? / L C # |
| 年（可选） 留空, 1970-2099 , - \* / |

其中：

-代表区间

\*代表通配符

?代表不想设置该字段

/代表递增(\*/10意味着从启动任务开始每10个时间单位执行一次)

下面只例出几个式子

CRON表达式 含义

"0 0 12 \* \* ?" 每天中午十二点触发

"0 15 10 ? \* \*" 每天早上10：15触发

"0 15 10 \* \* ?" 每天早上10：15触发

"0 15 10 \* \* ? \*" 每天早上10：15触发

"0 15 10 \* \* ? 2005" 2005年的每天早上10：15触发

"0 \* 14 \* \* ?" 每天从下午2点开始到2点59分每分钟一次触发

"0 0/5 14 \* \* ?" 每天从下午2点开始到2：55分结束每5分钟一次触发

"0 0/5 14,18 \* \* ?" 每天的下午2点至2：55和6点至6点55分两个时间段内每5分钟一次触发

"0 0-5 14 \* \* ?" 每天14:00至14:05每分钟一次触发

"0 10,44 14 ? 3 WED" ·· 三月的每周三的14：10和14：44触发

"0 15 10 ? \* MON-FRI" 每个周一、周二、周三、周四、周五的10：15触发

## 7.4 Quartz和spring-task的比较

Quartz是一款重量级的工具，而spring-task相对轻量，使用简单，当然，功能也更

### 7.4.1执行时间对比

**Spring Quartz 和 Spring Task执行时间对比：**

1. Quartz**如果设置同步模式时**：一个任务的两次执行的时间间隔是：“执行时间”和“trigger的设定间隔”的最大值，也就是说，第一个任务开始就进行trigger计时，如果设置的trigger间隔时间到了，任务仍没结束，下一次任务必须等待上一次任务结束才会执行

2. Task**默认就是同步模式**：一个任务的两次执行的时间间隔是：“执行时间”+“trigger的设定间隔”，即一个任务完成执行后，才开始trigger计时

### 7.4.2特点对比

**Spring Quartz 特点：**

1. 默认多线程异步执行

2. 一个任务在上一次调度未完成执行，下一次调度时间到时，会另起一个线程开始新的调度。在业务繁忙时，一个任务或许会有多个线程在执行，导致数据处理异常。(开启并发模式)

3. 单任务同步：配置属性，可以使一个任务的一次调度在未完成时，而不会开启下一次调度(关闭并发模式)

4. 多个任务同时运行，任务之间没有直接的影响，多任务执行的快慢取决于CPU的性能

5. SchedulerFactoryBean不能使用注解来配置

**Spring Task特点：**

1. 默认单线程同步执行

2. 一个任务执行完上一次之后，才会执行下一次调度

3. 多任务之间按顺序执行，一个任务执行完成之后才会执行另一个任务

4. 多任务并行执行需要设置线程池

5. 全程可以通过注解配置

**总结：**

1. **Task注解实现方式，比较简单。Quartz需要手动配置Jobs。**
2. **Task默认单线程串行执行任务，多任务时若某个任务执行时间过长，后续任务会无法及时执行。Quartz采用多线程，无这个问题。**
3. **调度，Task采用顺序执行，若当前调度占用时间过长，下一个调度无法及时执行；**
4. **Quartz采用异步，下一个调度时间到达时，会另一个线程执行调度，不会发生阻塞问题，但调度过多时可能导致数据处理异常**
5. **Quartz可以采用集群方式，分布式部署到多台机器，分配执行定时任务**

## 7.5 幂等性问题

幂等性：就是用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了了副作用。那么我们在部署项目时，一般会将项目分布式部署在多台服务器上，如果不考虑幂等性问题，多台服务器都会同时执行定时任务，导致定时任务执行结果异常，因此，在分布式环境下，幂等性问题相当重要

### 7.5.1通过机器ip限制定时任务的执行

假设我们部署的每台服务器的ip是已知且固定的，那么我们就可以通过限制ip的方式执行定时任务

@Component  
public class RegularTask {  
 @Lazy  
 @Scheduled(cron = "")  
 public void send() {  
 String ip = IPUtil.getLocalIP(); //获取本台服务器ip  
 String allowIp = PropertiesUtil.get("RUN\_TASK\_IP");**//允许的跑定时任务的ip放在properties文件中或者配置在数据库中** if (allowIp.indexOf(ip) == -1) { //ip不匹配直接return  
 return;  
 }  
 //*TODO do task* }  
}

### 7.5.2通过redis分布式锁实现

我们可以通过redis实现一个分布式锁，让多台服务器上的定时任务去竞争分布式锁，只有获取到锁的定时任务才能执行，实现如下：

//设定将值在redis中保存30s，所有的定时任务都来竞争"lock\_ip\_key"  
public static boolean lock(JedisCluster jedisCluster) {  
 return *tryLock*(jedisCluster, "lock\_ip\_key", 30, TimeUnit.SECONDS);  
}  
  
public static boolean tryLock(JedisCluster jedisCluster, String lockKey, long time, TimeUnit unit) {  
 //强制要求使用lock方法时设定时间不能少于1s

if (time < 1 || null == unit || null == jedisCluster || StringUtils.isBlank(lockKey)) {  
 throw new IllegalArgumentException("参数异常");  
 }  
 final long millis = unit.toMillis(time);

String ip = IPUtil.getLocalIP();  
 //如果"lock\_ip\_key"已存在，设置失败,PX意味毫秒，将ip地址作为value，用于删除时检测是否为锁的拥有者  
 final String existKey = jedisCluster.set(lockKey, ip, "NX", "PX", millis);  
 //如果设置成功，代表这个定时任务获取到了锁  
 return "OK".equalsIgnoreCase(existKey);  
}

请注意！这里不能使用**jedisCluster. setnx()**和**jedisCluster.expire()**方法，一旦使用了这两个方法，redis操作就不能保证原子性了，一旦在setnx之后，expire之前发生服务器宕机，则永远无法释放分布式锁资源，而**jedisCluster. set()**方法则可以保证原子性

并且在释放锁时，为了保证只有锁的拥有者可以释放锁，我们将加锁的值设为服务器的ip地址

public static void unLock(JedisCluster jedisCluster, String lockKey, String ip){

if (null == jedisCluster || StringUtils.isAnyBlank(lockKey, ip)) {  
 throw new IllegalArgumentException("参数异常");  
 }  
 if(ip.equals(jedisCluster.get(lockKey))){  
 jedis.del(lockKey);  
 }  
}

1. 其他

## 8.1 监听器、过滤器和拦截器的执行顺序

监听器和过滤器属于servlet部分的内容，而拦截器属于MVC框架提供的。监听器、过滤器一般配置在web.xml中，拦截器一般配置在springmvc.xml配置文件中

**他们的执行顺序是监听器 > 过滤器 > 拦截器，配置时要注意顺序**

首先容器启动，然后注册监听器，其中的一种名为ContextLoaderListener的监听器会注册spring配置文件，接着过滤前 – 拦截前 – Action处理 – 拦截后 – 过滤后，访问结束

接下来，我们将分别对其进行分析比较

### 8.1.1过滤器

过滤器依赖于servlet容器，是通过函数回调实现的。它和拦截器最容易混淆。过滤器可以对几乎所有请求进行过滤，但是缺点是一个过滤器实例只能在容器初始化时调用一次。常用于：在过滤器中修改字符编码，保证response不会乱码；在过滤器中修改HttpServletRequest的一些参数，包括：过滤低俗文字、危险字符等

具体过滤器的使用请参见servlet学习笔记

过滤器和拦截器的区别如下：

**①拦截器是基于Java的反射机制的，而过滤器是基于函数回调。**  
**②拦截器依赖于mvc框架，过滤器依赖于servlet容器。**  
**③拦截器只能对action请求起作用，而过滤器则可以对几乎所有的请求起作用(包括静态资源请求)。**  
**④拦截器可以访问action上下文、值栈里的对象，而过滤器不能访问。**  
**⑤在action的生命周期中，拦截器可以多次被调用，而过滤器只能在容器初始化时被调用一次。**  
**⑥拦截器可以获取IOC容器中的各个bean，而过滤器就不行，这点很重要，在拦截器类里注入一个service，就可以调用业务逻辑。这是因为过滤器**

### 8.1.2拦截器

拦截器依赖于web框架，通过java反射机制实现，是一种面向切面编程，一个拦截器可以在一次访问的生命周期内反复被调用，但缺点是拦截器只能拦截controller请求，对于如静态资源等请求则无能为力

拦截的具体使用方式请参见springMVC学习笔记3.9节



### 8.1.3监听器

监听器Listener依赖于servlet容器，它是实现了javax.servlet.ServletContextListener 接口的服务器端程序，它也是随web应用的启动而启动，只初始化一次，随web应用的停止而销毁。主要作用是：做一些初始化的内容添加工作、设置一些基本的内容、比如一些参数或者是一些固定的对象等。

1. 源码（aop,ioc的实现，bean生命周期,循环依赖,事务，设计模式）
   1. 什么是BeanDefinition

BeanDefinition是对bean的描述，spring将根据BeanDefinition创建bean对象

为什么不用class来描述bean呢？因为Class无法完成bean的抽象，比如bean的作用域，bean的注入模型，bean是否是懒加载等等信息，Class是无法抽象出来的，故而需要一个BeanDefinition类来抽象这些信息，以便于spring能够完美的实例化一个bean

beanDefinition中存放了很多class对象中没有的信息，如：

public abstract class AbstractBeanDefinition extends BeanMetadataAttributeAccessor  
 implements BeanDefinition, Cloneable {

*//表示bean的类型，如UserService.class，spring在创建bean的过程中根据此属性通过反射来获取对象*  
 private volatile Object beanClass;  
 *//表示bean的作用域，值为singleton代表是单例，为prototype代表是原型，这里SCOPE\_DEFAULT* *等同于单例*  
 private String scope = *SCOPE\_DEFAULT*;  
 *//表示bean是否需要懒加载，默认不需要，原型bean的lazyInit属性无效。*

*// 1.声明为true时，懒加载的单例bean会在第一次getBean()时生成该bean；*

*// 2.声明为false时，非懒加载的单例bean会在spring容器创建时直接生成*  
 private boolean lazyInit = false;  
 *//声明当前bean依赖于别的bean。所依赖的bean会被容器确保在当前bean实例化之前被实例化*  
 private String[] dependsOn;  
 *//为true表示该bean是主bean，spring中一个接口可能有多个bean实现，在进行依赖注入时，* *如果找到了多个bean对象，会优先使用主bean进行注入*  
 private boolean primary = false;  
 *//指定bean的init-method，即初始化方法*  
 private String initMethodName;

……  
}

* 1. FactoryBean接口和ObjectFactory接口

### 9.2.1 FactoryBean接口

在spring中，大量的对象都是通过beanFactory工厂生产的。一般情况下，Spring通过反射机制读取配置文件或注解为指定实现类实例化Bean。但在某些情况下，实例化Bean过程比较复杂，如果按照传统的方式，则需要在<bean>中提供大量的配置信息。配置方式的灵活性是受限的，这时采用编码的方式可能会得到一个简单的方案。Spring为此提供了一个org.springframework.bean.factory.FactoryBean的工厂类接口，用户可以通过实现该接口定制实例化Bean的逻辑。

如果使用FactoryBean创建bean对象，实际上会在ioc容器中创建两个bean对象，一个是我们希望创建的类型为T的bean对象，其id为类型名（首字母小写），另一个对象是FactoryBean对象本身，其beanName为&符加上id。如**MyFactoryBean extends FactoryBean<MyBean>**，会创建myFactoryBean和&myFactoryBean两个bean对象

public interface FactoryBean<T> {  
  
 */\*\*  
 \* 实例化bean，将其创建完成后返回  
 \*/* T getObject() throws Exception;  
  
 */\*\*  
 \* 获取bean的class类型  
 \*/* Class<?> getObjectType();  
  
 */\*\*  
 \* 判断是否为单例模式  
 \*/* boolean isSingleton();

}

只要一个类实现了FactoryBean<T>，并且对其通过注解或者xml等方式生成了FactoryBean<T>实现类的bean(即FactoryBean实现类需要加@Componet注解)，那么T就会被创建成为bean对象，并被spring容器管理

### 9.2.2 ObjectFactory接口

@FunctionalInterface  
public interface ObjectFactory<T> {  
  
 */\*\*  
 \* Return an instance (possibly shared or independent)  
 \* of the object managed by this factory.  
 \** ***@return*** *the resulting instance  
 \** ***@throws*** *BeansException in case of creation errors  
 \*/* T getObject() throws BeansException;  
  
}

这个接口类似于FactoryBean接口。ObjectFactory仅被当作一个普通工厂使用，他的T不会被ioc容器加载为bean对象，想要使用他的T对象就必须通过调用 getObject()方法获取，这是和FactoryBean最大的区别

* 1. spring ioc原理
     1. ioc的入口
        1. 入口一：ClassPathXmlApplicationContext

通过xml配置方式初始化bean，则入口为ClassPathXmlApplicationContext

这种方式下Spring 初始化的入口在 ContextLoaderListener，如果项目用了 Spring，通常可以在 web.xml 中找到下面这行代码。

<context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath\*:spring/spring-\*.xml</param-value>  
</context-param>

<listener>  
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
</listener>

ContextLoaderListener 是 Spring 的入口，而 contextConfigLocation 是 Spring 配置文件的路径

ContextLoaderListener 是实现了 javax.servlet.ServletContextListener 接口的服务器端程序，ContextLoaderListener随 web 应用的启动而启动，只初始化一次，随 web 应用的停止而销毁。在web应用启动的时候会调用 contextInitialized 方法，停止的时候会调用 contextDestroyed 方法

public class ContextLoaderListener extends ContextLoader implements ServletContextListener{

*/\*\*  
 \* 在web应用启动的时候会调用 contextInitialized 方法  
 \*/*

@Override  
 public void contextInitialized(ServletContextEvent event) {  
 initWebApplicationContext(event.getServletContext());  
 }  
  
  
 */\*\*  
 \* Close the root web application context.  
 \*/* @Override  
 public void contextDestroyed(ServletContextEvent event) {  
 closeWebApplicationContext(event.getServletContext());  
 ContextCleanupListener.*cleanupAttributes*(event.getServletContext());  
 }  
  
}

进入**initWebApplicationContext**方法，可以看到如下代码

if (this.context == null) {

*// 创建一个WebApplicationContext并保存到context属性*  
 this.context = createWebApplicationContext(servletContext);  
}  
if (this.context instanceof ConfigurableWebApplicationContext) {  
 ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) this.context;  
 if (!cwac.isActive()) {  
 // The context has not yet been refreshed -> provide services such as  
 // setting the parent context, setting the application context id, etc  
 if (cwac.getParent() == null) {  
 // The context instance was injected without an explicit parent ->  
 // determine parent for root web application context, if any.  
 ApplicationContext parent = loadParentContext(servletContext);  
 cwac.setParent(parent);  
 }

*//配置并刷新应用上下文，在这个方法里会调用refresh()方法刷新容器*  
 configureAndRefreshWebApplicationContext(cwac, servletContext);  
 }  
}

protected void configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac, ServletContext sc) {  
 *// 如果应用上下文id是原始默认值，则根据相关信息生成一个更有用的*

if (ObjectUtils.*identityToString*(wac).equals(wac.getId())) {  
 // The application context id is still set to its original default value  
 // -> assign a more useful id based on available information

*// 从servletContext中解析初始化参数contextId(可以在web.xml中配置)*  
 String idParam = sc.getInitParameter(*CONTEXT\_ID\_PARAM*);  
 if (idParam != null) {  
 wac.setId(idParam);  
 }  
 else {  
 *// 如果idParam为空, 则生成默认的id*  
 wac.setId(ConfigurableWebApplicationContext.*APPLICATION\_CONTEXT\_ID\_PREFIX* +  
 ObjectUtils.*getDisplayString*(sc.getContextPath()));  
 }  
 }  
  
 wac.setServletContext(sc);  
 String configLocationParam = sc.getInitParameter(*CONFIG\_LOCATION\_PARAM*);  
 if (configLocationParam != null) {  
 wac.setConfigLocation(configLocationParam);  
 }  
  
 // The wac environment's #initPropertySources will be called in any case when the context  
 // is refreshed; do it eagerly here to ensure servlet property sources are in place for  
 // use in any post-processing or initialization that occurs below prior to #refresh  
 ConfigurableEnvironment env = wac.getEnvironment();  
 if (env instanceof ConfigurableWebEnvironment) {  
 ((ConfigurableWebEnvironment) env).initPropertySources(sc, null);  
 }  
  
 customizeContext(sc, wac);

*//应用上下文的刷新*  
 wac.refresh();  
}

还有一种ioc的入口，是通过new直接创建applicationContext对象，在创建applicationContext容器的构造函数中，spring也会调用refresh方法刷新spring上下文容器

public ClassPathXmlApplicationContext(String[] configLocations, boolean refresh, ApplicationContext parent)  
 throws BeansException {  
  
 super(parent);  
 setConfigLocations(configLocations);  
 if (refresh) {  
 refresh();  
 }  
}

* + - 1. 入口二：AnnotationConfigApplicationContext

通过注解方式初始化bean，则入口为AnnotationConfigApplicationContext

public class AnnotationConfigApplicationContext extends GenericApplicationContext implements AnnotationConfigRegistry {

public AnnotationConfigApplicationContext(Class<?>... annotatedClasses) {  
 *//主要做了3件事：*

*//1.调用父类GenericApplicationContext的无参构造函数，用于创建beanFactory工厂，即DefaultListableBeanFactory*

*//2.指定注解配置读取器AnnotatedBeanDefinitionReader*

*//3.指定包路径扫描器ClassPathBeanDefinitionScanner*

this();

*//将传入的【springConfig】配置类注册到容器中，这里只是注册成BeanDefinition，并未实例化*  
 register(annotatedClasses);

*//应用上下文的刷新*  
 refresh();  
 }

}

* + 1. refresh方法

那让我们来看看AbstractApplicationContext.java#refresh()刷新方法做了什么

public void refresh() throws BeansException, IllegalStateException {

// 加锁，避免refresh()操作还没结束又开始执行启动或销毁容器的操作  
 synchronized (this.startupShutdownMonitor) {  
 // 准备工作，记录下容器的启动时间、标记“已启动”状态、处理配置文件中的占位符,不需要特别关注  
 prepareRefresh();  
   
 // 创建beanFactory，并且将配置文件解析成一个个 BeanDefinition，存放在BeanFactory的beanDefinitionMap中  
 ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();  
  
 // prepareBeanFactory是为了初始化beanFactory，给beanFactory的各项属性赋值，例如上下文的 ClassLoader、后置处理器等。这个方法会注册3个默认环境 bean：environment、systemProperties 和 systemEnvironment，注册 2 个 bean 后置处理器：ApplicationContextAwareProcessor 和 ApplicationListenerDetector  
 prepareBeanFactory(beanFactory);  
  
 try {  
 // ApplicationContext的子类可以重写该方法，在所有的beanDenifition加载完成之后，bean实例化之前执行，用以处理beanFactory。该方法默认实现为空，留给子类去实现  
 postProcessBeanFactory(beanFactory);  
  
 //调用并执行所有的已定义的 实现BeanFactoryPostProcessor接口的类 的postProcessBeanFactory，可以对beanFactory和beanDefinition进行后续处理  
 invokeBeanFactoryPostProcessors(beanFactory);  
  
 // 注册所有的 BeanPostProcessor，将所有实现了 BeanPostProcessor 接口的类加载到 BeanFactory 中。这一步仅注册，生成具体的bean时才会执行  
 registerBeanPostProcessors(beanFactory);  
  
 //做网站国际化时会用得到，比如点击切换英文，切换中文等，暂时可以忽略.  
 initMessageSource();  
  
 // 初始化当前 ApplicationContext 的事件广播器，这里也不展开了  
 initApplicationEventMulticaster();  
  
 //这也是一个空方法，具体的子类可以重写该方法，在这里初始化一些特殊的 Bean（在初始化 singleton beans 之前）.  
 onRefresh();  
  
 //注册事件监听器，监听器需要实现 ApplicationListener 接口.  
 registerListeners();  
  
 // 实例化所有的非懒加载的单例bean.  
 finishBeanFactoryInitialization(beanFactory);  
  
 //完成此上下文的刷新，主要是推送上下文刷新完毕事件（ContextRefreshedEvent）到监听器。  
 finishRefresh();  
 }  
  
 catch (BeansException ex) {  
 if (logger.isWarnEnabled()) {  
 logger.warn("Exception encountered during context initialization - " +  
 "cancelling refresh attempt: " + ex);  
 }  
  
 // Destroy already created singletons to avoid dangling resources.  
 destroyBeans();  
  
 // Reset 'active' flag.  
 cancelRefresh(ex);  
  
 // Propagate exception to caller.  
 throw ex;  
 }  
  
 finally {  
 // Reset common introspection caches in Spring's core, since we  
 // might not ever need metadata for singleton beans anymore...  
 resetCommonCaches();  
 }  
 }  
}

* + - 1. prepareRefresh方法

**prepareRefresh**主要负责做一些准备工作，比如记录下容器的启动时间、标记“已启动”状态、处理配置文件中的占位符

protected void prepareRefresh() {  
 this.startupDate = System.*currentTimeMillis*();  
 this.active.set(true);  
  
 if (logger.isInfoEnabled()) {  
 logger.info("Refreshing " + this);  
 }  
  
 // 处理占位符  
 initPropertySources();  
  
 // Validate that all properties marked as required are resolvable  
 // see ConfigurablePropertyResolver#setRequiredProperties  
 getEnvironment().validateRequiredProperties();  
  
 // Allow for the collection of early ApplicationEvents,  
 // to be published once the multicaster is available...  
 this.earlyApplicationEvents = new LinkedHashSet<ApplicationEvent>();  
}

* + - 1. obtainFreshBeanFactory方法（重要）

**obtainFreshBeanFactory**方法帮助我们创建beanFactory，并将要注册的bean信息读取成一个个beanDefinition对象，放到beanFactory的beanDefinitionMap中去

protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {

//刷新 BeanFactory,即创建一个新的BeanFactory, 由 AbstractRefreshableApplicationContext 的refreshBeanFactory()实现  
 refreshBeanFactory();

//拿到刷新后的 BeanFactory  
 ConfigurableListableBeanFactory beanFactory = getBeanFactory();  
 if (logger.isDebugEnabled()) {  
 logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);  
 }  
 return beanFactory;  
}

protected final void refreshBeanFactory() throws BeansException {  
 if (hasBeanFactory()) {  
 destroyBeans();  
 closeBeanFactory();  
 }  
 try {

//创建一个新的BeanFactory  
 DefaultListableBeanFactory beanFactory = createBeanFactory();  
 beanFactory.setSerializationId(getId());  
 customizeBeanFactory(beanFactory);

//扫描bean，并加载beanDefinition到beanDefinitionMap中  
 loadBeanDefinitions(beanFactory);  
 synchronized (this.beanFactoryMonitor) {  
 this.beanFactory = beanFactory;  
 }  
 }  
 catch (IOException ex) {  
 throw new ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);  
 }  
}

* + - 1. prepareBeanFactory方法

prepareBeanFactory是为了初始化beanFactory，给beanFactory的各项属性赋值

* + - 1. postProcessBeanFactory方法

这是一个空方法，ApplicationContext的子类可以重写该方法，在所有的beanDenifition加载完成之后，bean实例化之前执行，用以处理beanFactory，

* + - 1. invokeBeanFactoryPostProcessors方法（重要）

调用并执行所有的已定义实现BeanFactoryPostProcessor接口的类的postProcessBeanFactory，可以对beanFactory和beanDefinition进行后续处理

BeanFactoryPostProcessor 接口是 Spring 初始化 BeanFactory 时对外暴露的扩展点，Spring IoC 容器允许 BeanFactoryPostProcessor 在容器实例化任何 bean 之前读取 bean 的定义，并可以修改它。

protected void invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory beanFactory) {  
 // 1.getBeanFactoryPostProcessors():拿到当前应用上下文中已经注册的 BeanFactoryPostProcessor列表

// 2.invokeBeanFactoryPostProcessors: 实例化并调用所有已注册的BeanFactoryPostProcessor

PostProcessorRegistrationDelegate.*invokeBeanFactoryPostProcessors*(beanFactory, getBeanFactoryPostProcessors());  
  
 // Detect a LoadTimeWeaver and prepare for weaving, if found in the meantime  
 // (e.g. through an @Bean method registered by ConfigurationClassPostProcessor)  
 if (beanFactory.getTempClassLoader() == null && beanFactory.containsBean(*LOAD\_TIME\_WEAVER\_BEAN\_NAME*)) {  
 beanFactory.addBeanPostProcessor(new LoadTimeWeaverAwareProcessor(beanFactory));  
 beanFactory.setTempClassLoader(new ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));  
 }  
}

BeanFactoryPostProcessor有一个子接口，名叫BeanDefinitionRegistryPostProcessor。父接口会将ConfigurableListableBeanFactory暴露给我们使用，而子接口BeanDefinitionRegistryPostProcessor会将BeanDefinitionRegistry暴露给我们使用。通过子接口，我们可以向 BeanDefinitionMap中添加新的BeanDefinition

public interface BeanDefinitionRegistryPostProcessor extends BeanFactoryPostProcessor {void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException;  
}

BeanFactoryPostProcessor的执行顺序规则为：

1. 先执行子接口BeanDefinitionRegistryPostProcessor的实现类，然后再执行父接口BeanFactoryPostProcessor的实现类
2. 在满足第一条的前提下，优先执行实现了PriorityOrdered接口的实现类，然后再执行实现了Ordered接口的实现类，最后是常规的实现类
   * + 1. registerBeanPostProcessors方法（重要）

注册所有的 BeanPostProcessor，将所有实现了 BeanPostProcessor 接口的类加载到 BeanFactory 中。这一步仅注册，生成具体的bean时才会执行

BeanPostProcessor 接口是 Spring 初始化 bean 时对外暴露的扩展点，Spring IoC 容器允许 BeanPostProcessor 在容器初始化 bean 的前后，添加自己的逻辑处理。在这边只是注册到 BeanFactory 中，具体调用是在 bean 初始化的时候。

protected void registerBeanPostProcessors(ConfigurableListableBeanFactory beanFactory) {

//注册BeanPostProcessor  
 PostProcessorRegistrationDelegate.*registerBeanPostProcessors*(beanFactory, this);  
}

public static void registerBeanPostProcessors(ConfigurableListableBeanFactory beanFactory, AbstractApplicationContext applicationContext) {  
 //找出所有实现BeanPostProcessor接口的类  
 String[] postProcessorNames = beanFactory.getBeanNamesForType(BeanPostProcessor.class, true, false);  
  
 // Register BeanPostProcessorChecker that logs an info message when  
 // a bean is created during BeanPostProcessor instantiation, i.e. when  
 // a bean is not eligible for getting processed by all BeanPostProcessors.  
 int beanProcessorTargetCount = beanFactory.getBeanPostProcessorCount() + 1 + postProcessorNames.length;  
 beanFactory.addBeanPostProcessor(new BeanPostProcessorChecker(beanFactory, beanProcessorTargetCount));  
  
 // Separate between BeanPostProcessors that implement PriorityOrdered,  
 // Ordered, and the rest.

// 3.定义不同的变量用于区分: 实现PriorityOrdered接口的BeanPostProcessor、实现Ordered接口的BeanPostProcessor、普通BeanPostProcessor

// 3.1 priorityOrderedPostProcessors: 用于存放实现PriorityOrdered接口的BeanPostProcessor  
 List<BeanPostProcessor> priorityOrderedPostProcessors = new ArrayList<>();

// 3.2 internalPostProcessors: 用于存放Spring内部的BeanPostProcessor  
 List<BeanPostProcessor> internalPostProcessors = new ArrayList<>();

// 3.3 orderedPostProcessorNames: 用于存放实现Ordered接口的BeanPostProcessor的beanName  
 List<String> orderedPostProcessorNames = new ArrayList<>();

// 3.4 nonOrderedPostProcessorNames: 用于存放普通BeanPostProcessor的beanName  
 List<String> nonOrderedPostProcessorNames = new ArrayList<>();  
 for (String ppName : postProcessorNames) {

// 4.遍历postProcessorNames, 将BeanPostProcessors按3.1 - 3.4定义的变量区分开  
 if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {  
 BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);  
 priorityOrderedPostProcessors.add(pp);  
 if (pp instanceof MergedBeanDefinitionPostProcessor) {  
 internalPostProcessors.add(pp);  
 }  
 }  
 else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {  
 orderedPostProcessorNames.add(ppName);  
 }  
 else {  
 nonOrderedPostProcessorNames.add(ppName);  
 }  
 }  
  
 // First, register the BeanPostProcessors that implement PriorityOrdered.

// 5 对priorityOrderedPostProcessors进行排序并注册  
 *sortPostProcessors*(priorityOrderedPostProcessors, beanFactory);  
 *registerBeanPostProcessors*(beanFactory, priorityOrderedPostProcessors);  
  
 // Next, register the BeanPostProcessors that implement Ordered.

//6.1 注册实现Ordered接口的BeanPostProcessors  
 List<BeanPostProcessor> orderedPostProcessors = new ArrayList<>();  
 for (String ppName : orderedPostProcessorNames) {  
 BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);  
 orderedPostProcessors.add(pp);  
 if (pp instanceof MergedBeanDefinitionPostProcessor) {  
 internalPostProcessors.add(pp);  
 }  
 }

// 6.2 对orderedPostProcessors进行排序  
 *sortPostProcessors*(orderedPostProcessors, beanFactory);

//6.3 注册orderedPostProcessors  
 *registerBeanPostProcessors*(beanFactory, orderedPostProcessors);  
  
 // Now, register all regular BeanPostProcessors.

// 7.注册所有常规的BeanPostProcessors（过程与6类似）  
 List<BeanPostProcessor> nonOrderedPostProcessors = new ArrayList<>();  
 for (String ppName : nonOrderedPostProcessorNames) {  
 BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);  
 nonOrderedPostProcessors.add(pp);  
 if (pp instanceof MergedBeanDefinitionPostProcessor) {  
 internalPostProcessors.add(pp);  
 }  
 }  
 *registerBeanPostProcessors*(beanFactory, nonOrderedPostProcessors);  
  
 // Finally, re-register all internal BeanPostProcessors.

//8.最后, 重新注册所有内部BeanPostProcessors（相当于内部的BeanPostProcessor会被移到处理器链的末尾）  
 *sortPostProcessors*(internalPostProcessors, beanFactory);  
 *registerBeanPostProcessors*(beanFactory, internalPostProcessors);  
  
 // Re-register post-processor for detecting inner beans as ApplicationListeners,  
 // moving it to the end of the processor chain (for picking up proxies etc).

// 9.重新注册ApplicationListenerDetector（跟8类似，主要是为了移动到处理器链的末尾）  
 beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(applicationContext));  
}

这里执行顺序同样和beanFactoryPostProcessor的逻辑一样，都是优先执行实现了PriorityOrdered接口的BeanPostProcessor实现类，然后再执行实现了Ordered接口的BeanPostProcessor实现类，最后是常规的BeanPostProcessor实现类

* + - 1. initMessageSource方法

做网站国际化时会用得到，比如点击切换英文，切换中文等，暂时可以忽略

* + - 1. initApplicationEventMulticaster方法

初始化当前 ApplicationContext 的事件广播器

在容器创建的过程中，spring会发布很多容器事件，如容器启动、刷新、关闭等，这个功能将通过事件广播器来派发事件通知

* + - 1. onRefresh方法

这也是一个空方法，具体的子类可以重写该方法，在这里初始化一些特殊的 Bean（在初始化 singleton beans 之前）

* + - 1. registerListeners方法

注册事件监听器，监听器需要实现 ApplicationListener 接口

如果用户想监听容器事件，就需要一个实现了ApplicationListener 接口的监听器

protected void registerListeners() {  
 *// 注册之前保存的监听器*  
 for (ApplicationListener<?> listener : getApplicationListeners()) {  
 getApplicationEventMulticaster().addApplicationListener(listener);  
 }  
  
 // Do not initialize FactoryBeans here: We need to leave all regular beans  
 // uninitialized to let post-processors apply to them!

*// 从容器中取出所有监听器并注册*

String[] listenerBeanNames = getBeanNamesForType(ApplicationListener.class, true, false);  
 for (String listenerBeanName : listenerBeanNames) {  
 getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);  
 }  
  
 *// 发布之前步骤产生的容器事件*

Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;  
 this.earlyApplicationEvents = null;  
 if (earlyEventsToProcess != null) {  
 for (ApplicationEvent earlyEvent : earlyEventsToProcess) {  
 getApplicationEventMulticaster().multicastEvent(earlyEvent);  
 }  
 }  
}

* + - 1. finishBeanFactoryInitialization方法（重要）

实例化所有的非懒加载的单例bean，具体创建bean的过程可以参照下一节bean的生命周期

protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {  
 // 1.初始化此上下文的转换服务  
 if (beanFactory.containsBean(*CONVERSION\_SERVICE\_BEAN\_NAME*) && beanFactory.isTypeMatch(*CONVERSION\_SERVICE\_BEAN\_NAME*, ConversionService.class)) {  
 beanFactory.setConversionService(beanFactory.getBean(*CONVERSION\_SERVICE\_BEAN\_NAME*, ConversionService.class));  
 }  
  
 // 2.初始化LoadTimeWeaverAware Bean实例对象  
 String[] weaverAwareNames = beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, false);  
 for (String weaverAwareName : weaverAwareNames) {  
 getBean(weaverAwareName);  
 }  
  
 // Stop using the temporary ClassLoader for type matching.  
 beanFactory.setTempClassLoader(null);  
  
 //3.冻结所有bean定义，注册的bean定义不会被修改或进一步后处理，因为马上要创建 Bean 实例对象了  
 beanFactory.freezeConfiguration();  
  
 //4. 实例化所有非懒加载的单例bean对象.  
 beanFactory.preInstantiateSingletons();  
}

* + - * 1. preInstantiateSingletons

DefaultListableBeanFactory.class#preInstantiateSingletons()

public void preInstantiateSingletons() throws BeansException {  
 if (this.logger.isDebugEnabled()) {  
 this.logger.debug("Pre-instantiating singletons in " + this);  
 }  
  
 // Iterate over a copy to allow for init methods which in turn register new bean definitions.  
 // While this may not be part of the regular factory bootstrap, it does otherwise work fine.

// 1.创建beanDefinitionNames的副本beanNames用于后续的遍历，以允许init等方法注册新的bean定义  
 List<String> beanNames = new ArrayList<String>(this.beanDefinitionNames);  
  
 // Trigger initialization of all non-lazy singleton beans...

// 2.遍历beanNames，触发所有非懒加载单例bean的初始化  
 for (String beanName : beanNames) {

// 3.获取beanName对应的MergedBeanDefinition  
 RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);

// 4.bd对应的Bean实例：不是抽象类 && 是单例 && 不是懒加载  
 if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {

// 5.判断beanName对应的bean是否为FactoryBean  
 if (isFactoryBean(beanName)) {

// 5.1 通过beanName获取FactoryBean实例

// （重点）通过getBean(&beanName)拿到的是FactoryBean本身；通过getBean(beanName)拿到的是FactoryBean创建的Bean实例  
 final FactoryBean<?> factory = (FactoryBean<?>) getBean(*FACTORY\_BEAN\_PREFIX* + beanName);

// 5.2 判断这个FactoryBean是否希望立即初始化  
 boolean isEagerInit;  
 if (System.*getSecurityManager*() != null && factory instanceof SmartFactoryBean) {  
 isEagerInit = AccessController.*doPrivileged*(new PrivilegedAction<Boolean>() {  
 @Override  
 public Boolean run() {  
 return ((SmartFactoryBean<?>) factory).isEagerInit();  
 }  
 }, getAccessControlContext());  
 }  
 else {  
 isEagerInit = (factory instanceof SmartFactoryBean &&  
 ((SmartFactoryBean<?>) factory).isEagerInit());  
 }

// 5.3 （重点）如果希望立即初始化，则通过beanName获取bean实例  
 if (isEagerInit) {  
 getBean(beanName);  
 }  
 }  
 else {

// 6. （重点）如果beanName对应的bean不是FactoryBean，只是普通Bean，通过beanName获取bean实例  
 getBean(beanName);  
 }  
 }  
 }  
  
 // Trigger post-initialization callback for all applicable beans...

// 7.遍历beanNames，触发所有SmartInitializingSingleton的后初始化回调  
 for (String beanName : beanNames) {

// 7.1 拿到beanName对应的bean实例  
 Object singletonInstance = getSingleton(beanName);

// 7.2 判断singletonInstance是否实现了SmartInitializingSingleton接口  
 if (singletonInstance instanceof SmartInitializingSingleton) {  
 final SmartInitializingSingleton smartSingleton = (SmartInitializingSingleton) singletonInstance;

// 7.3 触发SmartInitializingSingleton实现类的afterSingletonsInstantiated方法  
 if (System.*getSecurityManager*() != null) {  
 AccessController.*doPrivileged*(new PrivilegedAction<Object>() {  
 @Override  
 public Object run() {  
 smartSingleton.afterSingletonsInstantiated();  
 return null;  
 }  
 }, getAccessControlContext());  
 }  
 else {  
 smartSingleton.afterSingletonsInstantiated();  
 }  
 }  
 }  
}

* + - 1. finishRefresh方法

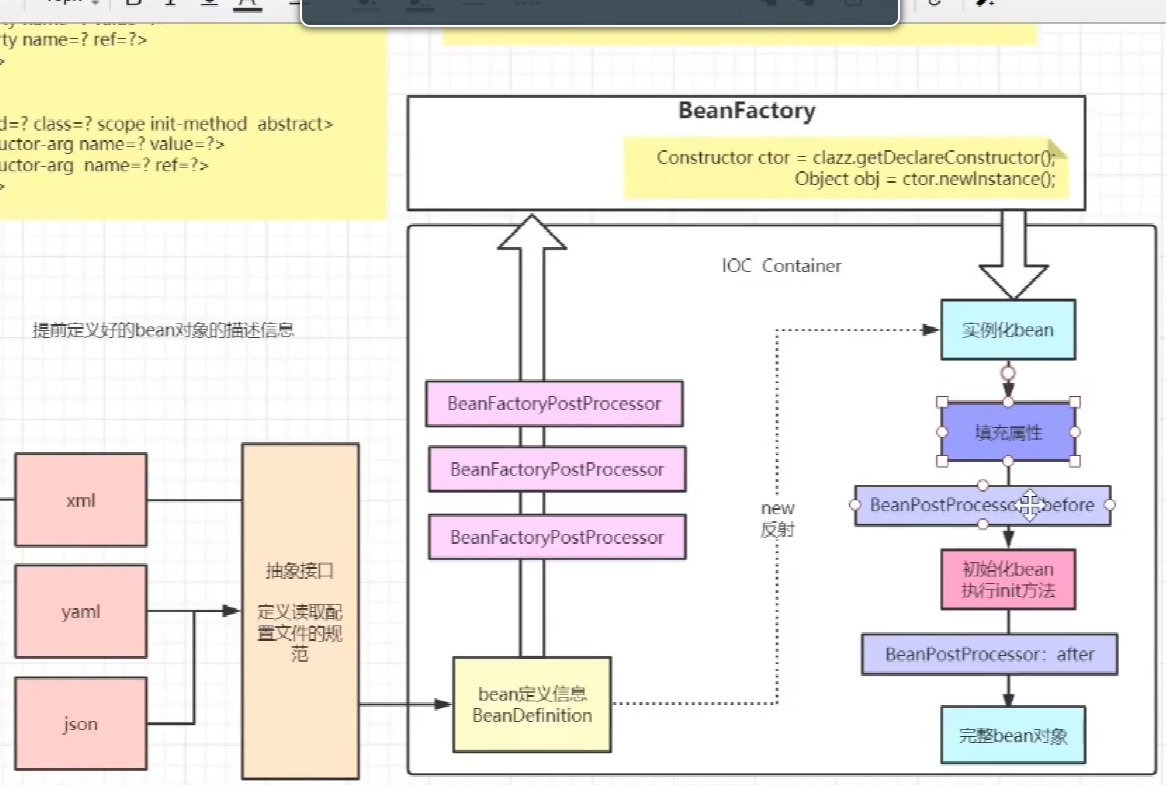
完成此上下文的刷新，主要是推送上下文刷新完毕事件（ContextRefreshedEvent）到监听器。

* 1. 单例bean的生命周期
     1. 创建单例bean的过程

对于多例bean（prototype原型），spring是在getBean时创建的，因此对于bean的生命周期，我们在这里只讨论单例模式下的bean

下面是一个包含bean生命周期的ioc启动过程

1. spring将从配置文件或注解中扫描出要创建的bean对象
2. 给每个bean对象创建beanDefinetion，并放置到beanFactory的beanDefintionMap中
3. 调用postProcessBeanFactory方法
4. 调用beanFactoryPostProcesser的扩展方法扩展beanFactory
5. 注册BeanPostProcesser扩展类，用于后续创建bean时扩展bean对象
6. createBeanInstance()实例化bean
7. populate()填充bean属性
8. 执行bean实现的各种aware接口(BeanNameAware BeanFactoryAware ApplicationContextAware)
9. 执行beanPostProcesser的before方法
10. 初始化bean，执行init方法（对应bean配置中的init-method属性或者被@PostConstruct注解标记的方法）
11. 执行beanPostProcesser的after方法



* + 1. 源码探究

refresh()方法会调用AbstractApplicationContext.java#finishBeanFactoryInitialization()方法，该方法将通过调用DefaultListableBeanFactory.class#beanFactory.preInstantiateSingletons()来实例化non-lazy的bean对象

* + - 1. getBean()

让我们看一下DefaultListableBeanFactory.class#beanFactory.preInstantiateSingletons()中使用到的AbstractBeanFactory.java#getBean()方法

@Override  
public Object getBean(String name) throws BeansException {

// （重点）获取name对应的bean实例，如果不存在，则创建一个  
 return doGetBean(name, null, null, false);  
}

* + - 1. doGetBean()

AbstractBeanFactory.java#doGetBean()方法

protected <T> T doGetBean(final String name, @Nullable final Class<T> requiredType,  
 @Nullable final Object[] args, boolean typeCheckOnly) throws BeansException {  
 // 1.解析beanName，主要是解析别名、去掉FactoryBean的前缀“&”  
 final String beanName = transformedBeanName(name);  
 Object bean;  
  
 // Eagerly check singleton cache for manually registered singletons.

// （重点）2.尝试从一级、二级、三级缓存中获取beanName对应的实例  
 Object sharedInstance = getSingleton(beanName);  
 if (sharedInstance != null && args == null) {

// 3.1如果beanName的实例存在于缓存中  
 if (logger.isDebugEnabled()) {  
 if (isSingletonCurrentlyInCreation(beanName)) {  
 logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +  
 "' that is not fully initialized yet - a consequence of a circular reference");  
 }  
 else {  
 logger.debug("Returning cached instance of singleton bean '" + beanName + "'");  
 }  
 }

// （重点）3.2 返回beanName对应的实例对象（主要用于FactoryBean的特殊处理，普通Bean会直接返回sharedInstance本身）  
 bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);  
 }  
 else {

// else，多级缓存中没有beanName的实例对象  
 // Spring只能解决单例模式下的循环依赖，在原型模式下如果存在循环依赖则抛出异常

// 这里检测原型模式下，该bean是否在加载，如果在加载则说明存在原型模式下的循环依赖，那么抛出异常  
 if (isPrototypeCurrentlyInCreation(beanName)) {  
 throw new BeanCurrentlyInCreationException(beanName);  
 }  
  
 // 5.获取parentBeanFactory  
 BeanFactory parentBeanFactory = getParentBeanFactory();

//5.1如果存在父beanFactory并且当前beanFactory中没有该bean的BeanDefinition  
 if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {  
 // Not found -> check parent.

// 5.2 将别名解析成真正的beanName  
 String nameToLookup = originalBeanName(name);

// 5.3 尝试在parentBeanFactory中获取bean对象实例  
 if (parentBeanFactory instanceof AbstractBeanFactory) {  
 return ((AbstractBeanFactory) parentBeanFactory).doGetBean(  
 nameToLookup, requiredType, args, typeCheckOnly);  
 }  
 else if (args != null) {  
 // Delegation to parent with explicit args.  
 return (T) parentBeanFactory.getBean(nameToLookup, args);  
 }  
 else {  
 // No args -> delegate to standard getBean method.  
 return parentBeanFactory.getBean(nameToLookup, requiredType);  
 }  
 }  
  
 if (!typeCheckOnly) {

// 6.如果不是仅仅做类型检测，而是创建bean实例，这里要将beanName放到alreadyCreated缓存  
 markBeanAsCreated(beanName);  
 }  
  
 try {

// 7.根据beanName重新获取MergedBeanDefinition（步骤6将MergedBeanDefinition删除了，这边获取一个新的）  
 final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);

// 7.1 检查MergedBeanDefinition   
 checkMergedBeanDefinition(mbd, beanName, args);  
  
 // Guarantee initialization of beans that the current bean depends on.

// 8.拿到当前bean依赖的bean名称集合，在实例化自己之前，需要先实例化自己依赖的bean  
 String[] dependsOn = mbd.getDependsOn();  
 if (dependsOn != null) {   
 for (String dep : dependsOn) {

// 8.1 检查dep是否依赖于beanName，即检查是否存在循环依赖（因为beanName已经依赖于dep了）  
 if (isDependent(beanName, dep)) {  
 throw new BeanCreationException(mbd.getResourceDescription(), beanName,  
 "Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");  
 }

// 8.2 将dep和beanName的依赖关系注册到缓存中  
 registerDependentBean(dep, beanName);

// 8.3 获取dep对应的bean实例，如果dep还没有创建bean实例，则创建dep的bean实例  
 getBean(dep);  
 }  
 }  
  
 // Create bean instance

// 9.针对不同的scope进行bean的创建  
 if (mbd.isSingleton()) {

//（重点） 9.1 scope为singleton的bean创建（新建了一个ObjectFactory，并且重写了getObject方法），参见getSingleton的重载二  
 sharedInstance = getSingleton(beanName, () -> {  
 try {

// 9.1.1 （重点）创建Bean实例  
 return createBean(beanName, mbd, args);  
 }  
 catch (BeansException ex) {  
 destroySingleton(beanName);  
 throw ex;  
 }  
 });

// （重点）9.1.2 返回beanName对应的实例对象  
 bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);  
 }  
  
 else if (mbd.isPrototype()) {  
 // It's a prototype -> create a new instance.

// 9.2 scope为prototype的bean创建  
 Object prototypeInstance = null;  
 try {

// 9.2.1 创建实例前的操作（将beanName保存到prototypesCurrentlyInCreation缓存中）  
 beforePrototypeCreation(beanName);

// 9.2.2 创建Bean实例  
 prototypeInstance = createBean(beanName, mbd, args);  
 }  
 finally {

// 9.2.3 创建实例后的操作（将创建完的beanName从prototypesCurrentlyInCreation缓存中移除，即beforePrototypeCreation()方法的反操作）  
 afterPrototypeCreation(beanName);  
 }

// 9.2.4 返回beanName对应的实例对象  
 bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);  
 }  
  
 else {

// 9.3 其他scope的bean创建，可能是request之类的

// 9.3.1 根据scopeName，从缓存拿到scope实例  
 String scopeName = mbd.getScope();  
 final Scope scope = this.scopes.get(scopeName);  
 if (scope == null) {  
 throw new IllegalStateException("No Scope registered for scope name '" + scopeName + "'");  
 }  
 try {

// 9.3.2 从指定的scope下创建bean  
 Object scopedInstance = scope.get(beanName, () -> {

// 9.3.3 创建实例前的操作（将beanName保存到prototypesCurrentlyInCreation缓存中）  
 beforePrototypeCreation(beanName);  
 try {

// 9.3.4 创建bean实例  
 return createBean(beanName, mbd, args);  
 }  
 finally {

// 9.3.5 创建实例后的操作（将创建完的beanName从prototypesCurrentlyInCreation缓存中移除）  
 afterPrototypeCreation(beanName);  
 }  
 });

// 9.3.6 返回beanName对应的实例对象  
 bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);  
 }  
 catch (IllegalStateException ex) {  
 throw new BeanCreationException(beanName,  
 "Scope '" + scopeName + "' is not active for the current thread; consider " +  
 "defining a scoped proxy for this bean if you intend to refer to it from a singleton",  
 ex);  
 }  
 }  
 }  
 catch (BeansException ex) {  
 cleanupAfterBeanCreationFailure(beanName);  
 throw ex;  
 }  
 }  
  
 // 10.检查所需类型是否与实际的bean对象的类型匹配  
 if (requiredType != null && !requiredType.isInstance(bean)) {  
 try {

// 10.1 类型不对，则尝试转换bean类型  
 T convertedBean = getTypeConverter().convertIfNecessary(bean, requiredType);  
 if (convertedBean == null) {  
 throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());  
 }  
 return convertedBean;  
 }  
 catch (TypeMismatchException ex) {  
 if (logger.isDebugEnabled()) {  
 logger.debug("Failed to convert bean '" + name + "' to required type '" +  
 ClassUtils.*getQualifiedName*(requiredType) + "'", ex);  
 }  
 throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());  
 }  
 }

// 11.返回创建出来的bean实例对象  
 return (T) bean;  
}

* + - * 1. getSingleton()重载一

DefaultSingletonBeanRegistry.java#getSingleton()重载方法一将尝试从一级、二级、三级缓存中获取beanName对应的实例，三级缓存的概念可以参见下一节循环依赖。**这个方法会将三级缓存工厂创建的bean放入到二级缓存中**

@Override  
@Nullable  
public Object getSingleton(String beanName) {  
 return getSingleton(beanName, true);  
}

@Nullable  
protected Object getSingleton(String beanName, boolean allowEarlyReference) {

//（重点）尝试从一级缓存中获取bean对象，即尝试获取成品bean

Object singletonObject = this.singletonObjects.get(beanName);

//如果一级缓存中没有，并且bean已经处于创建过程中  
 if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {  
 synchronized (this.singletonObjects) {

//（重点）尝试从二级缓存中获取bean对象，即半成品bean  
 singletonObject = this.earlySingletonObjects.get(beanName);

//如果二级缓存也没有，同时允许使用三级缓存，就从三级缓存获取ObjectFactory对象  
 if (singletonObject == null && allowEarlyReference) {  
 ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);  
 if (singletonFactory != null) {

//（重点）通过ObjectFactory对象获取bean，实际上执行的是getEarlyBeanReference()方法，因为添加三级缓存时执行的是addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean))  
 singletonObject = singletonFactory.getObject();

//将获取到的bean放到二级缓存中  
 this.earlySingletonObjects.put(beanName, singletonObject);

//从三级缓存中移除bean的ObjectFactory对象  
 this.singletonFactories.remove(beanName);  
 }  
 }  
 }  
 }  
 return singletonObject;  
}

我们可以看到这里是通过三级缓存来解决循环依赖的：

1. 首先是从一级缓存查找，如果一级缓存有，则返回
2. 如果一级缓存没有，并且bean已经在创建过程中了，则去二级缓存查找，如果有则返回
3. 如果二级缓存没有，并且允许使用三级缓存，则从三级缓存中获取ObjectFactory对象，并通过它获取AOP增强后的代理对象作为bean，然后将获取到的bean对象从三级缓存中移动到二级缓存中
   * + - 1. getObjectForBeanInstance()对factoryBean进行处理

如果通过getSingleton()获取到了bean，即sharedInstance变量，那么在返回之前，还需要通过AbstractBeanFactory.java#getObjectForBeanInstance()方法进行处理

getObjectForBeanInstance()方法被doGetBean()调用，用于对factoryBean类型的bean进行特殊处理，让我们来看看源码

protected Object getObjectForBeanInstance(  
 Object beanInstance, String name, String beanName, @Nullable RootBeanDefinition mbd) {  
  
 // 1.如果name以“&”为前缀，但是beanInstance不是FactoryBean，则抛异常  
 if (BeanFactoryUtils.*isFactoryDereference*(name) && !(beanInstance instanceof FactoryBean)) {  
 throw new BeanIsNotAFactoryException(transformedBeanName(name), beanInstance.getClass());  
 }  
  
 // 2.1 如果beanInstance不是FactoryBean（也就是普通bean），则直接返回beanInstance

// 2.2 如果beanInstance是FactoryBean，并且name以“&”为前缀，则直接返回beanInstance（以“&”为前缀代表想获取的是FactoryBean本身）  
 if (!(beanInstance instanceof FactoryBean) || BeanFactoryUtils.*isFactoryDereference*(name)) {  
 return beanInstance;  
 }  
 // 3 走到这代表beanInstance是FactoryBean，但name不带有“&”前缀，表示想要获取的是FactoryBean创建的bean对象实例  
 Object object = null;  
 if (mbd == null) {

// 4.如果mbd为空，则尝试从factoryBeanObjectCache缓存中获取该FactoryBean创建的对象实例  
 object = getCachedObjectForFactoryBean(beanName);  
 }  
 if (object == null) {  
 // 5.只有beanInstance是FactoryBean才能走到这边，因此直接强转  
 FactoryBean<?> factory = (FactoryBean<?>) beanInstance;  
 // Caches object obtained from FactoryBean if it is a singleton.  
 if (mbd == null && containsBeanDefinition(beanName)) {

// 6.mbd为空，但是该bean的BeanDefinition在缓存中存在，则获取该bean的MergedBeanDefinition  
 mbd = getMergedLocalBeanDefinition(beanName);  
 }  
 boolean synthetic = (mbd != null && mbd.isSynthetic());

// 8.从FactoryBean获取对象实例  
 object = getObjectFromFactoryBean(factory, beanName, !synthetic);  
 }  
 return object;  
}

getObjectFromFactoryBean()

核心是第8步，源码如下：

protected Object getObjectFromFactoryBean(FactoryBean<?> factory, String beanName, boolean shouldPostProcess) {  
 //8.1判断是否未单例，如果是单例，就不必重复创建

if (factory.isSingleton() && containsSingleton(beanName)) {  
 synchronized (getSingletonMutex()) {

//8.1.1尝试从缓存获取  
 Object object = this.factoryBeanObjectCache.get(beanName);  
 if (object == null) {

//8.1.2缓存没有，尝试调用doGetObjectFromFactoryBean获取  
 object = doGetObjectFromFactoryBean(factory, beanName);

//8.1.3再次从缓存获取，看看有没有其他线程创建好的   
 Object alreadyThere = this.factoryBeanObjectCache.get(beanName);  
 if (alreadyThere != null) {  
 //8.1.4如果缓存中有，那么直接用缓存的就好了

object = alreadyThere;  
 } else {

//8.1.5如果应该对bean进行后置处理，则进行后置处理  
 if (shouldPostProcess) {  
 if (isSingletonCurrentlyInCreation(beanName)) {  
 // Temporarily return non-post-processed object, not storing it yet..  
 return object;  
 }  
 beforeSingletonCreation(beanName);  
 try {  
 object = postProcessObjectFromFactoryBean(object, beanName);  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(beanName,  
 "Post-processing of FactoryBean's singleton object failed", ex);  
 }  
 finally {  
 afterSingletonCreation(beanName);  
 }  
 }  
 if (containsSingleton(beanName)) {

//8.1.6将object缓存起来  
 this.factoryBeanObjectCache.put(beanName, object);  
 }  
 }  
 }  
 return object;  
 }  
 }  
 else {

//8.2如果不是单例则直接调用doGetObjectFromFactoryBean获取  
 Object object = doGetObjectFromFactoryBean(factory, beanName);  
 if (shouldPostProcess) {  
 try {  
 object = postProcessObjectFromFactoryBean(object, beanName);  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(beanName, "Post-processing of FactoryBean's object failed", ex);  
 }  
 }  
 return object;  
 }  
}

doGetObjectFromFactoryBean()

接下来看doGetObjectFromFactoryBean

private Object doGetObjectFromFactoryBean(final FactoryBean<?> factory, final String beanName)  
 throws BeanCreationException {  
  
 Object object;  
 try {  
 if (System.*getSecurityManager*() != null) {  
 AccessControlContext acc = getAccessControlContext();  
 try {  
 object = AccessController.*doPrivileged*((PrivilegedExceptionAction<Object>) factory::getObject, acc);  
 }  
 catch (PrivilegedActionException pae) {  
 throw pae.getException();  
 }  
 }  
 else {

//调用用户实现的getObject()方法 创建bean对象  
 object = factory.getObject();  
 }  
 }  
 catch (FactoryBeanNotInitializedException ex) {  
 throw new BeanCurrentlyInCreationException(beanName, ex.toString());  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(beanName, "FactoryBean threw exception on object creation", ex);  
 }  
  
 if (object == null) {  
 if (isSingletonCurrentlyInCreation(beanName)) {  
 throw new BeanCurrentlyInCreationException(  
 beanName, "FactoryBean which is currently in creation returned null from getObject");  
 }  
 object = new NullBean();  
 }  
 return object;  
}

* + - * 1. getSingleton()重载二

DefaultSingletonBeanRegistry.java# getSingleton()

doGetBean()方法的9.1节调用了这个重载方法，这个方法是用来创建bean的，它将调用createBean()方法，这个我们后面会讲到。同时，这个方法还会将新创建的bean(已初始化完成)放到一级缓存中

public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {  
 Assert.*notNull*(beanName, "Bean name must not be null");  
 synchronized (this.singletonObjects) {

//尝试从单例池(一级缓存)中获取bean对象  
 Object singletonObject = this.singletonObjects.get(beanName);  
 if (singletonObject == null) {

//如果当前beanFactory正在被销毁则抛出异常  
 if (this.singletonsCurrentlyInDestruction) {  
 throw new BeanCreationNotAllowedException(beanName,  
 "Singleton bean creation not allowed while singletons of this factory are in destruction " +  
 "(Do not request a bean from a BeanFactory in a destroy method implementation!)");  
 }  
 if (logger.isDebugEnabled()) {  
 logger.debug("Creating shared instance of singleton bean '" + beanName + "'");  
 }

//bean创建前的准备工作，这一步很重要，和解决循环引用有关，它会将beanName放到一个叫singletonsCurrentlyInCreation的set集合中去，表示这个bean正在被创建（可以看一看 重载一的isSingletonCurrentlyInCreation()方法）  
 beforeSingletonCreation(beanName);  
 boolean newSingleton = false;  
 boolean recordSuppressedExceptions = (this.suppressedExceptions == null);  
 if (recordSuppressedExceptions) {  
 this.suppressedExceptions = new LinkedHashSet<>();  
 }  
 try {

//（重点），通过方法参数传入的匿名内部类的getObject()方法（即creatBean()方法）来获取单例bean对象  
 singletonObject = singletonFactory.getObject();  
 newSingleton = true;  
 }catch (IllegalStateException ex) {  
 // Has the singleton object implicitly appeared in the meantime ->  
 // if yes, proceed with it since the exception indicates that state.  
 singletonObject = this.singletonObjects.get(beanName);  
 if (singletonObject == null) {  
 throw ex;  
 }  
 } catch (BeanCreationException ex) {  
 if (recordSuppressedExceptions) {  
 for (Exception suppressedException : this.suppressedExceptions) {  
 ex.addRelatedCause(suppressedException);  
 }  
 }  
 throw ex;  
 }finally {  
 if (recordSuppressedExceptions) {  
 this.suppressedExceptions = null;  
 }

//创建bean的后置操作，将beanName从singletonsCurrentlyInCreation中移除，表示bean的创建已经结束了，和beforeSingletonCreation()方法呼应  
 afterSingletonCreation(beanName);  
 }

if (newSingleton) {

//（重点）如果是新创建的bean，则加入到一级缓存中去，同时从二级缓存和三级缓存中删除对应缓存  
 addSingleton(beanName, singletonObject);  
 }  
 }  
 return singletonObject;  
 }  
}

createBean()方法

在doGetBean()的9.1.1步骤中，getSingleton()重载二方法是通过调用AbstractAutowireCapableBeanFactory.java #createBean()来创建bean的

@Override  
protected Object createBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args)  
 throws BeanCreationException {  
  
 if (logger.isDebugEnabled()) {  
 logger.debug("Creating instance of bean '" + beanName + "'");  
 }  
 RootBeanDefinition mbdToUse = mbd;  
  
 Class<?> resolvedClass = resolveBeanClass(mbd, beanName);  
 if (resolvedClass != null && !mbd.hasBeanClass() && mbd.getBeanClassName() != null) {  
 mbdToUse = new RootBeanDefinition(mbd);  
 mbdToUse.setBeanClass(resolvedClass);  
 }  
  
 // Prepare method overrides.  
 try {  
 mbdToUse.prepareMethodOverrides();  
 }  
 catch (BeanDefinitionValidationException ex) {  
 throw new BeanDefinitionStoreException(mbdToUse.getResourceDescription(),  
 beanName, "Validation of method overrides failed", ex);  
 }  
  
 try {  
 //（重点）(具体实现可以看aop实现原理)，获取所有的InstantiationAwareBeanPostProcessor，并执行他们的postProcessBeforeInstantiation生成代理bean对象  
 Object bean = resolveBeforeInstantiation(beanName, mbdToUse);

//（重点）如果InstantiationAwareBeanPostProcessor的postProcessBeforeInstantiation()返回不为null，则不说明创建了代理对象，不再进行后续的实例化和初始化过程  
 if (bean != null) {  
 return bean;  
 }  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(mbdToUse.getResourceDescription(), beanName,  
 "BeanPostProcessor before instantiation of bean failed", ex);  
 }  
  
 try {

//（重点），创建bean的方法   
 Object beanInstance = doCreateBean(beanName, mbdToUse, args);  
 if (logger.isDebugEnabled()) {  
 logger.debug("Finished creating instance of bean '" + beanName + "'");  
 }  
 return beanInstance;  
 }  
 catch (BeanCreationException | ImplicitlyAppearedSingletonException ex) {  
 // A previously detected exception with proper bean creation context already,  
 // or illegal singleton state to be communicated up to DefaultSingletonBeanRegistry.  
 throw ex;  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(  
 mbdToUse.getResourceDescription(), beanName, "Unexpected exception during bean creation", ex);  
 }  
}

doCreateBean()方法

protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final @Nullable Object[] args)  
 throws BeanCreationException {  
  
 // Instantiate the bean.  
 BeanWrapper instanceWrapper = null;  
 if (mbd.isSingleton()) {

//如果是单例，那么就从FactoryBean缓存中清除当前bean的缓存，防止重复创建  
 instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);  
 }  
 if (instanceWrapper == null) {

//（重点），实例化bean，开辟内存空间，创建bean对象  
 instanceWrapper = createBeanInstance(beanName, mbd, args);  
 }  
 final Object bean = instanceWrapper.getWrappedInstance();  
 Class<?> beanType = instanceWrapper.getWrappedClass();  
 if (beanType != NullBean.class) {  
 mbd.resolvedTargetType = beanType;  
 }  
  
 // Allow post-processors to modify the merged bean definition.  
 synchronized (mbd.postProcessingLock) {  
 if (!mbd.postProcessed) {  
 try {

//（重点），扫描@Autowired、@Reference、@Resouce等注解  
 applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(mbd.getResourceDescription(), beanName,  
 "Post-processing of merged bean definition failed", ex);  
 }  
 mbd.postProcessed = true;  
 }  
 }  
  
 // 判断当前bean是否需要提前曝光，如果是单例、允许循环依赖、bean正在创建，则需要提前曝光  
 boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&  
 isSingletonCurrentlyInCreation(beanName));  
 if (earlySingletonExposure) {  
 if (logger.isDebugEnabled()) {  
 logger.debug("Eagerly caching bean '" + beanName +  
 "' to allow for resolving potential circular references");  
 }

// （重点），在bean初始化完成之前，添加三级缓存，同时移除二级缓存  
 addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));  
 }  
  
 // Initialize the bean instance.  
 Object exposedObject = bean;  
 try {

//（重点），填充bean的属性，如果存在循环依赖，则会递归创建所依赖的bean   
 populateBean(beanName, mbd, instanceWrapper);

//（重点）执行初始化逻辑   
 exposedObject = initializeBean(beanName, exposedObject, mbd);  
 }  
 catch (Throwable ex) {  
 if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {  
 throw (BeanCreationException) ex;  
 }  
 else {  
 throw new BeanCreationException(  
 mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);  
 }  
 }  
  
 if (earlySingletonExposure) {  
 Object earlySingletonReference = getSingleton(beanName, false);  
 if (earlySingletonReference != null) {  
 if (exposedObject == bean) {  
 exposedObject = earlySingletonReference;  
 }  
 else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {  
 String[] dependentBeans = getDependentBeans(beanName);  
 Set<String> actualDependentBeans = new LinkedHashSet<>(dependentBeans.length);  
 for (String dependentBean : dependentBeans) {  
 if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {  
 actualDependentBeans.add(dependentBean);  
 }  
 }  
 if (!actualDependentBeans.isEmpty()) {  
 throw new BeanCurrentlyInCreationException(beanName,  
 "Bean with name '" + beanName + "' has been injected into other beans [" +  
 StringUtils.*collectionToCommaDelimitedString*(actualDependentBeans) +  
 "] in its raw version as part of a circular reference, but has eventually been " +  
 "wrapped. This means that said other beans do not use the final version of the " +  
 "bean. This is often the result of over-eager type matching - consider using " +  
 "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.");  
 }  
 }  
 }  
 }  
  
 // Register bean as disposable.  
 try {  
 registerDisposableBeanIfNecessary(beanName, bean, mbd);  
 }  
 catch (BeanDefinitionValidationException ex) {  
 throw new BeanCreationException(  
 mbd.getResourceDescription(), beanName, "Invalid destruction signature", ex);  
 }  
  
 return exposedObject;  
}

createBeanInstance()实例化bean

AbstractAutowireCapableBeanFactory.java #createBeanInstance()负责实例化bean，注意这里只实例化常规bean，aop代理bean由resolveBeforeInstantiation()方法获取

protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) {  
 //确保bean的class此时已被解析  
 Class<?> beanClass = resolveBeanClass(mbd, beanName);  
  
 if (beanClass != null && !Modifier.*isPublic*(beanClass.getModifiers()) && !mbd.isNonPublicAccessAllowed()) {  
 throw new BeanCreationException(mbd.getResourceDescription(), beanName,  
 "Bean class isn't public, and non-public access not allowed: " + beanClass.getName());  
 }  
  
 Supplier<?> instanceSupplier = mbd.getInstanceSupplier();  
 if (instanceSupplier != null) {  
 return obtainFromSupplier(instanceSupplier, beanName);  
 }  
  
 if (mbd.getFactoryMethodName() != null) {  
 return instantiateUsingFactoryMethod(beanName, mbd, args);  
 }  
  
 // Shortcut when re-creating the same bean...  
 boolean resolved = false;  
 boolean autowireNecessary = false;  
 if (args == null) {  
 synchronized (mbd.constructorArgumentLock) {  
 if (mbd.resolvedConstructorOrFactoryMethod != null) {  
 resolved = true;  
 autowireNecessary = mbd.constructorArgumentsResolved;  
 }  
 }  
 }  
 if (resolved) {  
 if (autowireNecessary) {  
 return autowireConstructor(beanName, mbd, null, null);  
 }  
 else {  
 return instantiateBean(beanName, mbd);  
 }  
 }  
  
 // Need to determine the constructor...  
 Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);  
 if (ctors != null ||  
 mbd.getResolvedAutowireMode() == RootBeanDefinition.*AUTOWIRE\_CONSTRUCTOR* ||  
 mbd.hasConstructorArgumentValues() || !ObjectUtils.*isEmpty*(args)) {  
 return autowireConstructor(beanName, mbd, ctors, args);  
 }  
  
 //（重点）使用无参构造函数实例化bean  
 return instantiateBean(beanName, mbd);  
}

protected BeanWrapper instantiateBean(final String beanName, final RootBeanDefinition mbd) {  
 try {  
 Object beanInstance;  
 final BeanFactory parent = this;  
 if (System.*getSecurityManager*() != null) {  
 beanInstance = AccessController.*doPrivileged*((PrivilegedAction<Object>) () ->  
 // 实例化只会分配内存空间，设置默认值

getInstantiationStrategy().instantiate(mbd, beanName, parent),  
 getAccessControlContext());  
 }  
 else {  
 beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);  
 }  
 BeanWrapper bw = new BeanWrapperImpl(beanInstance);  
 initBeanWrapper(bw);  
 return bw;  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(  
 mbd.getResourceDescription(), beanName, "Instantiation of bean failed", ex);  
 }  
}

applyMergedBeanDefinitionPostProcessors()扫描注解

protected void applyMergedBeanDefinitionPostProcessors(RootBeanDefinition mbd, Class<?> beanType, String beanName) {  
 for (MergedBeanDefinitionPostProcessor processor : getBeanPostProcessorCache().mergedDefinition) {  
 processor.postProcessMergedBeanDefinition(mbd, beanType, beanName);  
 }  
}

可以看到，该方法主要是获取所有的MergedBeanDefinitionPostProcessor接口的实现，并调用该接口的postProcessMergedBeanDefinition()方法

也就是说每个bean创建的过程中都会调用postProcessMergedBeanDefinition()方法，执行：

processor.postProcessMergedBeanDefinition(mbd, beanType, beanName);

就是这里的processor就是MergedBeanDefinitionPostProcessor接口的实现类，该接口有很多实现类，分别用于解析不同的注解，这些注解都是标记在bean属性或者方法上的，eg：

1. 对于@Resource注解，其实现类是CommonAnnotationBeanPostProcessor
2. 对于@Autowired注解，其实现类是AutowiredAnnotationBeanPostProcessor
3. 对于dubbo的@Reference，其实现类是ReferenceAnnotationBeanPostProcessor

在MergedBeanDefinitionPostProcessor#postProcessMergedBeanDefinition()方法中，解析出当前bean中所有被指定注解标记的字段和方法

以上实现类，还都会实现InstantiationAwareBeanPostProcessor，等执行到populateBean()时，就会扫描所有InstantiationAwareBeanPostProcessor，并执行他们的postProcessPropertyValues ()方法，[详见populateBean()](#InstantiationAwareBeanPostProcessor的post)

addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean))

在完成bean的实例化以后，在填充属性之前，spring将工厂添加到三级缓存，同时移除二级缓存。通过这个工厂singletonFactory的getObject()方法可以得到bean对象

这个方法属于创建提前代理的动态代理bean，具体可以参见aop原理

DefaultSingletonBeanRegistry.java#addSingletonFactory ()

protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {  
 Assert.*notNull*(singletonFactory, "Singleton factory must not be null");  
 synchronized (this.singletonObjects) {  
 if (!this.singletonObjects.containsKey(beanName)) {  
 this.singletonFactories.put(beanName, singletonFactory);  
 this.earlySingletonObjects.remove(beanName);  
 this.registeredSingletons.add(beanName);  
 }  
 }  
}

AbstractAutowireCapableBeanFactory # getEarlyBeanReference()

而参数singletonFactory的getObject()方法实现为上面lambda表达式中的AbstractAutowireCapableBeanFactory.java # getEarlyBeanReference()方法，其源码如下：

protected Object getEarlyBeanReference(String beanName, RootBeanDefinition mbd, Object bean) {  
 Object exposedObject = bean;  
 if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {  
 for (BeanPostProcessor bp : getBeanPostProcessors()) {

// SmartInstantiationAwareBeanPostProcessor是BeanPostProcessor的一个子接口，它将在aop原理中被提到  
 if (bp instanceof SmartInstantiationAwareBeanPostProcessor) {  
 SmartInstantiationAwareBeanPostProcessor ibp = (SmartInstantiationAwareBeanPostProcessor) bp;

// （重点）调用SmartInstantiationAwareBeanPostProcessor的getEarlyBeanReference()接口，实际上调用的是AbstractAutoProxyCreator类的实现  
 exposedObject = ibp.getEarlyBeanReference(exposedObject, beanName);  
 }  
 }  
 }  
 return exposedObject;  
}

AbstractAutoProxyCreator # getEarlyBeanReference()

关于这个方法可以重点参考aop发生循环依赖时的代理对象创建流程

public Object getEarlyBeanReference(Object bean, String beanName) throws BeansException {  
 Object cacheKey = getCacheKey(bean.getClass(), beanName);

//如果前bean不在earlyProxyReferences中，**即意味着是非提前代理**，就需要加入到earlyProxyReferences

//earlyProxyReferences是提前动态代理集合，如果存在的话，就说明已经代理过了  
 if (!this.earlyProxyReferences.contains(cacheKey)) {  
 this.earlyProxyReferences.add(cacheKey);  
 }

// （重点），按需创建代理对象  
 return wrapIfNecessary(bean, beanName, cacheKey);  
}

再往后具体可以参考aop原理去看

populateBean()填充bean属性

AbstractAutowireCapableBeanFactory.java #populateBean()负责给实例化之后的bean填充bean属性

protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper bw) {

//bw是上一步createBeanInstance实例化的对象，对他进行判空   
 if (bw == null) {  
 if (mbd.hasPropertyValues()) {  
 throw new BeanCreationException(  
 mbd.getResourceDescription(), beanName, "Cannot apply property values to null instance");  
 }  
 else {  
 // Skip property population phase for null instance.  
 return;  
 }  
 }  
  
 // 给InstantiationAwareBeanPostProcessor最后一次机会，在属性注入前修改bean的属性值  
 boolean continueWithPropertyPopulation = true;  
 if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {  
 for (BeanPostProcessor bp : getBeanPostProcessors()) {  
 if (bp instanceof InstantiationAwareBeanPostProcessor) {  
 InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;

//执行InstantiationAwareBeanPostProcessor的postProcessAfterInstantiation()方法  
 if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {  
 continueWithPropertyPopulation = false;  
 break;  
 }  
 }  
 }  
 }  
  
 if (!continueWithPropertyPopulation) {  
 return;  
 }

//获取要注入的属性列表  
 PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyValues() : null);

//根据bean的依赖注入方式，遍历所有属性，依次对属性进行注入  
 if (mbd.getResolvedAutowireMode() == RootBeanDefinition.*AUTOWIRE\_BY\_NAME* ||  
 mbd.getResolvedAutowireMode() == RootBeanDefinition.*AUTOWIRE\_BY\_TYPE*) {

//重点，@Resource、@Autowired等注解装配不会进入这里，而是在后续操作中装配

//想走到这里，有两种方式：

//1.自定义一个BeanFactoryPostProcessor，修改beanDenfintion的*autowireMode*为上面if中的两个之一

//2.采用xml方式声明bean，然后将<bean>标签的autowire属性设置为byName或者byType，详见[autowire属性](#bean标签的autowire属性)  
 MutablePropertyValues newPvs = new MutablePropertyValues(pvs);  
   
 // Add property values based on autowire by name if applicable.  
 if (mbd.getResolvedAutowireMode() == RootBeanDefinition.*AUTOWIRE\_BY\_NAME*) {

//（重点）根据bean的属性名进行依次注入  
 autowireByName(beanName, mbd, bw, newPvs);  
 }  
  
 // Add property values based on autowire by type if applicable.  
 if (mbd.getResolvedAutowireMode() == RootBeanDefinition.*AUTOWIRE\_BY\_TYPE*) {

//（重点）根据bean的类型进行依次注入  
 autowireByType(beanName, mbd, bw, newPvs);  
 }  
  
 pvs = newPvs;  
 }

//判断容器是否注册了InstantiationAwareBeanPostProcessors  
 boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();

//是否进行了依赖检查  
 boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.*DEPENDENCY\_CHECK\_NONE*);  
  
 if (hasInstAwareBpps || needsDepCheck) {  
 if (pvs == null) {  
 pvs = mbd.getPropertyValues();  
 }  
 PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);  
 if (hasInstAwareBpps) {  
 for (BeanPostProcessor bp : getBeanPostProcessors()) {  
 if (bp instanceof InstantiationAwareBeanPostProcessor) {  
 InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;

// 重点，执行InstantiationAwareBeanPostProcessor的postProcessPropertyValues()方法，即使用@Resource、@Autowired等注解进行装配  
 pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);  
 if (pvs == null) {  
 return;  
 }  
 }  
 }  
 }  
 if (needsDepCheck) {

//检查depends-on依赖的bean是否已经完成初始化  
 checkDependencies(beanName, mbd, filteredPds, pvs);  
 }  
 }  
  
 if (pvs != null) {

//（重点），填充bean属性,即将pvs中的所有属性填充到bw中  
 applyPropertyValues(beanName, mbd, bw, pvs);  
 }  
}

autowireByName()根据名称装配属性

protected void autowireByName(String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs) {

//获取所有set方法存在的，不是简单类型(基本数据类型、Date、URI、URL、Number等)的属性名，即过滤出满足装配条件的bean  
 String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);  
 for (String propertyName : propertyNames) {  
 if (containsBean(propertyName)) {

//从ioc中获取bean，如果该bean还未创建，将进入doGetBean()递归初始化bean  
 Object bean = getBean(propertyName);

//将获取到的bean放入pvs中  
 pvs.add(propertyName, bean);

//注册依赖关系  
 registerDependentBean(propertyName, beanName);  
 if (logger.isDebugEnabled()) {  
 logger.debug("Added autowiring by name from bean name '" + beanName +  
 "' via property '" + propertyName + "' to bean named '" + propertyName + "'");  
 }  
 }  
 else {  
 if (logger.isTraceEnabled()) {  
 logger.trace("Not autowiring property '" + propertyName + "' of bean '" + beanName +  
 "' by name: no matching bean found");  
 }  
 }  
 }  
}

autowireByType()根据类型装配属性

protected void autowireByType(String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs) {  
 //获取类型转换器  
 TypeConverter converter = getCustomTypeConverter();  
 if (converter == null) {  
 converter = bw;  
 }  
  
 Set<String> autowiredBeanNames = new LinkedHashSet<>(4);

//获取所有set方法存在的，不是简单类型(基本数据类型、Date、URI、URL、Number等)的属性名，即过滤出满足装配条件的bean  
 String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);  
 for (String propertyName : propertyNames) {  
 try {  
 PropertyDescriptor pd = bw.getPropertyDescriptor(propertyName);  
 // Don't try autowiring by type for type Object: never makes sense,  
 // even if it technically is a unsatisfied, non-simple property.  
 if (Object.class != pd.getPropertyType()) {

//获取对应属性的set方法  
 MethodParameter methodParam = BeanUtils.*getWriteMethodParameter*(pd);  
 // Do not allow eager init for type matching in case of a prioritized post-processor.  
 boolean eager = !PriorityOrdered.class.isInstance(bw.getWrappedInstance());  
 DependencyDescriptor desc = new AutowireByTypeDependencyDescriptor(methodParam, eager);

//解析属性对应的值，  
 Object autowiredArgument = resolveDependency(desc, beanName, autowiredBeanNames, converter);  
 if (autowiredArgument != null) {

//将键值对放入pvs中  
 pvs.add(propertyName, autowiredArgument);  
 }

//注册依赖关系  
 for (String autowiredBeanName : autowiredBeanNames) {  
 registerDependentBean(autowiredBeanName, beanName);  
 if (logger.isDebugEnabled()) {  
 logger.debug("Autowiring by type from bean name '" + beanName + "' via property '" +  
 propertyName + "' to bean named '" + autowiredBeanName + "'");  
 }  
 }  
 autowiredBeanNames.clear();  
 }  
 }  
 catch (BeansException ex) {  
 throw new UnsatisfiedDependencyException(mbd.getResourceDescription(), beanName, propertyName, ex);  
 }  
 }  
}

InstantiationAwareBeanPostProcessor#postProcessPropertyValues()注解的依赖注入

这一步是为@Resource、@Reference、@Autowired等注解标记的属性注入依赖，在doCreateBean()的前面的代码中，spring通过MergedBeanDefinitionPostProcessor接口的postProcessMergedBeanDefinition()方法，解析出当前bean中所有被指定注解标记的字段和方法

到了这一步，就要对之前解析出的字段和方法进行依赖注入。因为上述注解的扫描类，基本都同时实现了MergedBeanDefinitionPostProcessor接口和InstantiationAwareBeanPostProcessor接口，因此解析出的字段和方法的元数据都存放在这两个接口的实现类中，那么通过调用实现类的postProcessPropertyValues()方法即可完成注解的依赖注入

具体的代码就不再看了，感兴趣可以研究一下

1. 对于@Resource注解，其实现类是CommonAnnotationBeanPostProcessor
2. 对于@Autowired注解，其实现类是AutowiredAnnotationBeanPostProcessor
3. 对于dubbo的@Reference，其实现类是ReferenceAnnotationBeanPostProcessor

applyPropertyValues()填充bean属性

该方法负责将pvs中保存的属性和值设置到bean中

protected void applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper bw, PropertyValues pvs) {

if (pvs.isEmpty()) {  
 return;  
 }  
  
 if (System.*getSecurityManager*() != null && bw instanceof BeanWrapperImpl) {  
 ((BeanWrapperImpl) bw).setSecurityContext(getAccessControlContext());  
 }  
  
 MutablePropertyValues mpvs = null;  
 List<PropertyValue> original;  
 //如果pvs是MutablePropertyValues类型的实现  
 if (pvs instanceof MutablePropertyValues) {

//如果pvs中的所有制类型都已经转换完毕，则可以直接设置到bean中  
 mpvs = (MutablePropertyValues) pvs;  
 if (mpvs.isConverted()) {  
 // Shortcut: use the pre-converted values as-is.  
 try {  
 bw.setPropertyValues(mpvs);  
 return;  
 }  
 catch (BeansException ex) {  
 throw new BeanCreationException(  
 mbd.getResourceDescription(), beanName, "Error setting property values", ex);  
 }  
 }

//没有转换完毕，保存原始值，等待类型转换  
 original = mpvs.getPropertyValueList();  
 }  
 else {

// pvs不是MutablePropertyValues类型的实现，保存原始值，等待类型转换  
 original = Arrays.*asList*(pvs.getPropertyValues());  
 }

//获取类型转换器  
 TypeConverter converter = getCustomTypeConverter();  
 if (converter == null) {  
 converter = bw;  
 }  
 BeanDefinitionValueResolver valueResolver = new BeanDefinitionValueResolver(this, beanName, mbd, converter);  
  
 // Create a deep copy, resolving any references for values.

//准备进行深拷贝  
 List<PropertyValue> deepCopy = new ArrayList<>(original.size());  
 boolean resolveNecessary = false;

//（重点），遍历属性原始值，将属性转换为对应类的对应属性类型  
 for (PropertyValue pv : original) {  
 if (pv.isConverted()) {

//如果当前pv已被转换，直接保存  
 deepCopy.add(pv);  
 }  
 else {  
 String propertyName = pv.getName();  
 Object originalValue = pv.getValue();  
 Object resolvedValue = valueResolver.resolveValueIfNecessary(pv, originalValue);  
 Object convertedValue = resolvedValue;  
 boolean convertible = bw.isWritableProperty(propertyName) &&  
 !PropertyAccessorUtils.*isNestedOrIndexedProperty*(propertyName);  
 if (convertible) {  
 convertedValue = convertForProperty(resolvedValue, propertyName, bw, converter);  
 }  
 // Possibly store converted value in merged bean definition,  
 // in order to avoid re-conversion for every created bean instance.  
 if (resolvedValue == originalValue) {  
 if (convertible) {  
 pv.setConvertedValue(convertedValue);  
 }  
 deepCopy.add(pv);  
 }  
 else if (convertible && originalValue instanceof TypedStringValue &&  
 !((TypedStringValue) originalValue).isDynamic() &&  
 !(convertedValue instanceof Collection || ObjectUtils.*isArray*(convertedValue))) {  
 pv.setConvertedValue(convertedValue);  
 deepCopy.add(pv);  
 }  
 else {  
 resolveNecessary = true;  
 deepCopy.add(new PropertyValue(pv, convertedValue));  
 }  
 }  
 }  
 if (mpvs != null && !resolveNecessary) {  
 mpvs.setConverted();  
 }  
  
 // Set our (possibly massaged) deep copy.  
 try {  
 bw.setPropertyValues(new MutablePropertyValues(deepCopy));  
 }  
 catch (BeansException ex) {  
 throw new BeanCreationException(  
 mbd.getResourceDescription(), beanName, "Error setting property values", ex);  
 }  
}

initializeBean()初始化bean

这个方法负责对bean进行初始化操作

protected Object initializeBean(final String beanName, final Object bean, @Nullable RootBeanDefinition mbd) {  
 if (System.*getSecurityManager*() != null) {  
 AccessController.*doPrivileged*((PrivilegedAction<Object>) () -> {  
 invokeAwareMethods(beanName, bean);  
 return null;  
 }, getAccessControlContext());  
 }else {

//（重点），执行一系列aware接口，如BeanNameAware、BeanClassLoaderAware、BeanFactoryAware  
 invokeAwareMethods(beanName, bean);  
 }  
  
 Object wrappedBean = bean;  
 if (mbd == null || !mbd.isSynthetic()) {

//（重点），调用所有BeanPostProcesser的postProcessBeforeInitialization()方法  
 wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);  
 }  
  
 try {

//（重点），执行初始化方法，先判断bean是否实现了InitializingBean，如果实现了就调用afterPropertiesSet()方法

//随后会判断有没有指定init-method，即被@PostConstruct注解标记的方法

//如果指定了，并且 init-method方法名不是afterPropertiesSet，则调用该方法

invokeInitMethods(beanName, wrappedBean, mbd);  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(  
 (mbd != null ? mbd.getResourceDescription() : null),  
 beanName, "Invocation of init method failed", ex);  
 }  
 if (mbd == null || !mbd.isSynthetic()) {

//（重点），调用所有BeanPostProcesser的postProcessAfterInitialization ()方法  
 wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);  
 }  
  
 return wrappedBean;  
}

* 1. spring aop原理

spring是通过动态代理(详见设计模式笔记)来实现aop的，那么具体是怎么操作的呢？

* + 1. @EnableAspectJAutoProxy注解

首先我们都知道，spring开启aop是通过在任意@Configuration类上添加@EnableAspectJAutoProxy注解

@Target(ElementType.*TYPE*)  
@Retention(RetentionPolicy.*RUNTIME*)  
@Documented  
@Import(AspectJAutoProxyRegistrar.class)  
public @interface EnableAspectJAutoProxy {  
  
 //是否使用cglib创建代理boolean proxyTargetClass() default false;  
  
 //代理对象是否暴露在ThreadLocal中，以便通过aopContext获取boolean exposeProxy() default false;  
  
}

* + 1. AspectJAutoProxyRegistrar类

这里我们可以看到@EnableAspectJAutoProxy注解通过使用@Import注解，将生成AspectJAutoProxyRegistrar类的bean实例，并将该bean放入ioc容器中，交由容器管理

class AspectJAutoProxyRegistrar implements ImportBeanDefinitionRegistrar {@Override  
 public void registerBeanDefinitions(  
 AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {  
 //（重点），向BeanDefinitionRegistry注册AnnotationAwareAspectJAutoProxyCreator  
 AopConfigUtils.*registerAspectJAnnotationAutoProxyCreatorIfNecessary*(registry);  
  
 AnnotationAttributes enableAspectJAutoProxy =  
 AnnotationConfigUtils.*attributesFor*(importingClassMetadata, EnableAspectJAutoProxy.class);  
 if (enableAspectJAutoProxy != null) {  
 if (enableAspectJAutoProxy.getBoolean("proxyTargetClass")) {

//根据@EnableAspectJAutoProxy注解的配置强制使用cglib创建代理  
 AopConfigUtils.*forceAutoProxyCreatorToUseClassProxying*(registry);  
 }  
 if (enableAspectJAutoProxy.getBoolean("exposeProxy")) {

//根据@EnableAspectJAutoProxy注解的配置强制在目标方法执行前将代理对象暴露到ThreadLocal中  
 AopConfigUtils.*forceAutoProxyCreatorToExposeProxy*(registry);  
 }  
 }  
 }  
  
}

* + 1. AnnotationAwareAspectJAutoProxyCreator类

AspectJAutoProxyRegistrar类将会注册AnnotationAwareAspectJAutoProxyCreator到beanDefinition中，这个类的祖先类中，有个叫AbstractAutoProxyCreator的祖先类，它实现了SmartInstantiationAwareBeanPostProcessor，而SmartInstantiationAwareBeanPostProcessor是InstantiationAwareBeanPostProcessor的子接口，有点绕哈。**总而言之，就是类AnnotationAwareAspectJAutoProxyCreator间接实现了InstantiationAwareBeanPostProcessor接口**

**这个AbstractAutoProxyCreator非常重要，spring的aop的代理类的创建，一共有三个创建时机，其中有两个都是在这个类中实现的**

* + 1. Aop代理对象创建的三个时机
       1. 用户自定义TargetSource的动态代理对象创建——InstantiationAwareBeanPostProcessor的postProcessBeforeInstantiation()方法

如果用户自定义了targetSource，那么将会在这个逻辑创建代理对象，后面也不再走bean的创建过程了(即实例化->填充属性->初始化)

* + - * 1. 源码

首先我们先介绍一下bean实例化处理器**InstantiationAwareBeanPostProcessor**

这个InstantiationAwareBeanPostProcessor继承自BeanPostProcessor，但他额外加了两个方法，即postProcessBeforeInstantiation()和postProcessAfterInstantiation()：

**postProcessBeforeInstantiation：**before方法，在调用doCreateBean()之前，由resolveBeforeInstantiation()方法调用。**如果此方法返回一个非空对象，那么bean的创建过程将会”短路”，即createBean()方法将直接返回，而不再进行createBeanInstance()->populateBean()->initializeBean()等操作，具体短路过程可以参见createBean()的resolveBeforeInstantiation()方法**。

**postProcessAfterInstantiation：**after方法，在populate填充属性时被调用

**如果用户自定义了targetSource，就会走InstantiationAwareBeanPostProcessor 的before方法创建代理对象，同时跳过实例化和初始化过程**

**postProcessBeforeInstantiation就是我们真正生成代理类的地方，下一节我们将介绍postProcessBeforeInstantiation()方法的调用时机**

public abstract class AbstractAutoProxyCreator extends ProxyProcessorSupport implements SmartInstantiationAwareBeanPostProcessor, BeanFactoryAware {  
 @Override  
 public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws BeansException {  
 Object cacheKey = getCacheKey(beanClass, beanName);  
  
 if (!StringUtils.hasLength(beanName) || !this.targetSourcedBeans.contains(beanName)) {  
 if (this.advisedBeans.containsKey(cacheKey)) {  
 return null;  
 }  
 if (isInfrastructureClass(beanClass) || shouldSkip(beanClass, beanName)) {  
 this.advisedBeans.put(cacheKey, Boolean.*FALSE*);  
 return null;  
 }  
 }  
  
 //获取用户自定义的targetSource   
 TargetSource targetSource = getCustomTargetSource(beanClass, beanName);

//如果存在用户自定义的targetSource，那么就在用户\***实例化\***之前创建代理对象即可，避免目标对象不必要的实例化，后续常规创建bean的过程就被短路了，不会再执行  
 if (targetSource != null) {  
 if (StringUtils.hasLength(beanName)) {

// targetSource是被代理对象实例的来源，即代理对象代理的就是targetSourceBean对象

this.targetSourcedBeans.add(beanName);  
 }

// 获取所有的advisors/advice/interceptor  
 Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(beanClass, beanName, targetSource);

//（重点），创建动态代理对象  
 Object proxy = createProxy(beanClass, beanName, specificInterceptors, targetSource);  
 this.proxyTypes.put(cacheKey, proxy.getClass());  
 return proxy;  
 }  
  
 return null;  
 }  
  
 @Override  
 public boolean postProcessAfterInstantiation(Object bean, String beanName) {  
 return true;  
 }  
}

* + - * 1. 调用时机——resolveBeforeInstantiation()

那么什么时候调用AbstractAutoProxyCreator**#**postProcessBeforeInstantiation呢？

ioc将会在createBean()方法中，调用resolveBeforeInstantiation()方法，通过该方法将调用所有的InstantiationAwareBeanPostProcessor的postProcessBeforeInstantiation()方法

@Override  
 protected Object createBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args)  
 throws BeanCreationException {  
  **……**  
 try {  
 //（重点），获取所有的InstantiationAwareBeanPostProcessor，并执行他们的postProcessBeforeInstantiation生成代理bean对象  
 Object bean = resolveBeforeInstantiation(beanName, mbdToUse);

//（重点），如果InstantiationAwareBeanPostProcessor的postProcessBeforeInstantiation()返回不为null，则说明已经创建了代理对象，那么不再进行后续的实例化和初始化过程，发生短路，createBean()直接返回  
 if (bean != null) {  
 return bean;  
 }  
 } catch (Throwable ex) {  
 throw new BeanCreationException(mbdToUse.getResourceDescription(), beanName,  
 "BeanPostProcessor before instantiation of bean failed", ex);  
 }  
  
 try {

//执行doCreateBean()  
 Object beanInstance = doCreateBean(beanName, mbdToUse, args);  
 if (logger.isDebugEnabled()) {  
 logger.debug("Finished creating instance of bean '" + beanName + "'");  
 }  
 return beanInstance;  
 }  
  **……**  
 }

@Nullable  
protected Object resolveBeforeInstantiation(String beanName, RootBeanDefinition mbd) {  
 Object bean = null;  
 if (!Boolean.*FALSE*.equals(mbd.beforeInstantiationResolved)) {  
 // mbd不是合成的，并且beanFactory中存在InstantiationAwareBeanPostProcessor  
 if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {  
 Class<?> targetType = determineTargetType(beanName, mbd);  
 if (targetType != null) {

//（重点），获取所有InstantiationAwareBeanPostProcessor，并且执行其前置before方法  
 bean = applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);  
 if (bean != null) {

//（重点），如果前置方法返回不为空，则跳过spring默认的bean初始化过程，直接执行bean处理器(BeanPostProcessors)的after方法

//获取所有BeanPostProcessors，并且执行其后置after方法  
 bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);  
 }  
 }  
 }  
 mbd.beforeInstantiationResolved = (bean != null);  
 }  
 return bean;  
}

* + - 1. 提前代理的动态代理创建(发生循环依赖的流程)——addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean))

**如果发生了循环依赖，则通过这个流程创建动态代理对象**

* + - * 1. 循环依赖发生时三级缓存的变化

我们先来过一下循环依赖发生时，三级缓存的变化

发生循环依赖时，执行getBean(A)，在实例化A后将A放入三级缓存，同时删除二级缓存中的A，接着执行populateBean(A)时，发现需要B

接着会调用getBean(B)，在实例化B后将B放入三级缓存，接着执行populateBean(B)，发现需要A

接着调用getBean(A)，在getSingleton(A)重载一中，在三级缓存中找到了A的工厂，接着调用singletonFactory.getObject()，其实就是getEarlyBeanReference()方法，获取bean的半成品对象

* + - * 1. AbstractAutowireCapableBeanFactory # getEarlyBeanReference()

这个方法在AbstractAutowireCapableBeanFactory.java中，通过该方法可以获取到代理对象

protected Object getEarlyBeanReference(String beanName, RootBeanDefinition mbd, Object bean) {  
 Object exposedObject = bean;  
 if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {  
 for (BeanPostProcessor bp : getBeanPostProcessors()) {

// SmartInstantiationAwareBeanPostProcessor是BeanPostProcessor的一个子接口，它将在aop原理中被提到  
 if (bp instanceof SmartInstantiationAwareBeanPostProcessor) {  
 SmartInstantiationAwareBeanPostProcessor ibp = (SmartInstantiationAwareBeanPostProcessor) bp;

// （重点），调用SmartInstantiationAwareBeanPostProcessor的getEarlyBeanReference()接口，实际上调用的是AbstractAutoProxyCreator类的实现  
 exposedObject = ibp.getEarlyBeanReference(exposedObject, beanName);  
 }  
 }  
 }  
 return exposedObject;  
}

AbstractAutoProxyCreator # getEarlyBeanReference()

这个方法在AbstractAutoProxyCreator.java中，通过该方法可以获取到代理对象

public Object getEarlyBeanReference(Object bean, String beanName) throws BeansException {  
 Object cacheKey = getCacheKey(bean.getClass(), beanName);

//如果前bean不在earlyProxyReferences中，**即意味着是非提前代理**，就需要加入到earlyProxyReferences

//earlyProxyReferences是提前动态代理集合，如果存在的话，就说明已经代理过了  
 if (!this.earlyProxyReferences.contains(cacheKey)) {  
 this.earlyProxyReferences.add(cacheKey);  
 }

// 重点，按需创建代理对象  
 return wrapIfNecessary(bean, beanName, cacheKey);  
}

wrapIfNecessary()

protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {

//如果bean在targetSourceBeans中，代表已经被处理过  
 if (StringUtils.*hasLength*(beanName) && this.targetSourcedBeans.contains(beanName)) {  
 return bean;  
 }

//如果bean不在advisedBeans中，则代表不需要被织入

if (Boolean.*FALSE*.equals(this.advisedBeans.get(cacheKey))) {  
 return bean;  
 }

//是不是基础bean，是不是需要跳过  
 if (isInfrastructureClass(bean.getClass()) || shouldSkip(bean.getClass(), beanName)) {  
 this.advisedBeans.put(cacheKey, Boolean.*FALSE*);  
 return bean;  
 }  
  
 //同postProcessBeforeInstantiation，获取所有的advisors/advice/interceptor  
 Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, null);

//如果存在增强器则进行代理  
 if (specificInterceptors != *DO\_NOT\_PROXY*) {  
 this.advisedBeans.put(cacheKey, Boolean.*TRUE*);

//（重点），创建代理  
 Object proxy = createProxy(bean.getClass(), beanName, specificInterceptors, new SingletonTargetSource(bean));  
 this.proxyTypes.put(cacheKey, proxy.getClass());  
 return proxy;  
 }  
  
 this.advisedBeans.put(cacheKey, Boolean.*FALSE*);  
 return bean;  
}

JDK动态代理 or Cglib动态代理？

后面的代码感兴趣可以继续往下看，后面有一段还需要注意

//根据类加载器获取代理对象

public Object getProxy(@Nullable ClassLoader classLoader) {  
 return createAopProxy().getProxy(classLoader);  
}

public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {  
  
 @Override  
 public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {  
 ......

//如果被代理类有接口，则使用JDK动态代理，否则使用cglib动态代理   
 if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {  
 return new JdkDynamicAopProxy(config);  
 }  
 return new ObjenesisCglibAopProxy(config);  
 ......  
 }s  
}

* + - 1. 非提前代理的动态代理创建(常规流程)——BeanPostProcessor的postProcessBeforeInstantiation()方法

**如果没有用户自定义的TargetSource，也没有发生循环依赖，那么接下来就会走正常的实例化和初始化流程**

经过createInstance()创建实例，populateBean()填充属性，在initializeBean()这一步，将会调用BeanPostProcessor的相关方法

由于InstantiationAwareBeanPostProcessor是继承自BeanPostProcessor，因此也会进行BeanPostProcessor的before和after方法的调用，具体调用时机可以看initializeBean()方法的实现

现在我们来看看AbstractAutoProxyCreator中实现的postProcessAfterInitialization()方法

public abstract class AbstractAutoProxyCreator extends ProxyProcessorSupport  
 implements SmartInstantiationAwareBeanPostProcessor, BeanFactoryAware {  
 @Override  
 public Object postProcessBeforeInitialization(Object bean, String beanName) {  
 return bean;  
 }  
@Override  
 public Object postProcessAfterInitialization(@Nullable Object bean, String beanName) throws BeansException {  
 if (bean != null) {  
 Object cacheKey = getCacheKey(bean.getClass(), beanName);

//如果前bean不在earlyProxyReferences中，**即意味着是非提前代理**，就按需创建代理

//earlyProxyReferences是提前动态代理集合，如果存在的话，就说明已经代理过了  
 if (!this.earlyProxyReferences.contains(cacheKey)) {  
 return wrapIfNecessary(bean, beanName, cacheKey);  
 }  
 }  
 return bean;  
 }  
}

* + - * 1. wrapIfNecessary()按需创建代理

参考上一节中的wrapIfNecessary()

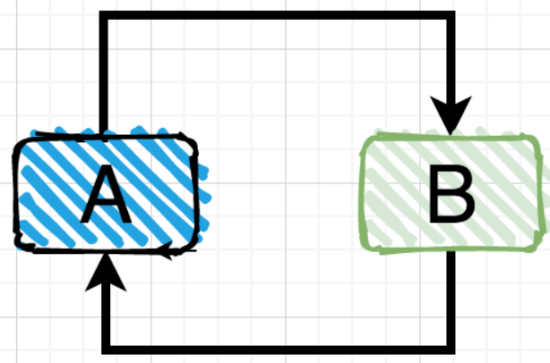
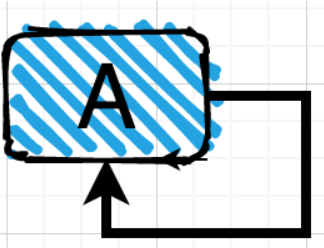
* 1. 循环依赖
     1. 什么是循环依赖

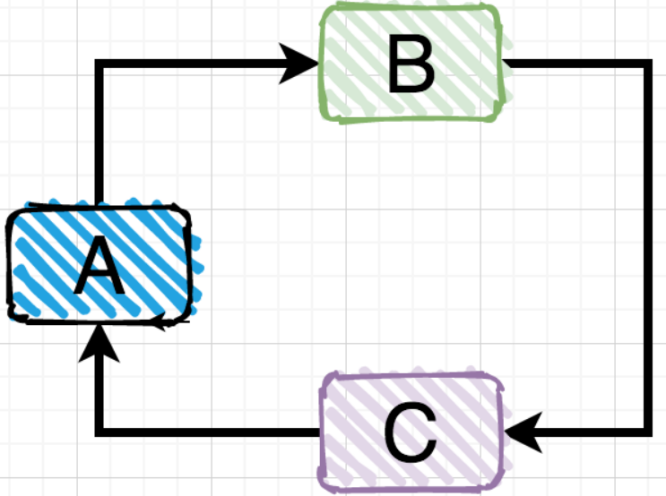
这里举一个简单的例子：

@Service  
public class A {  
 @Autowired  
 private B b;  
}  
@Service  
public class B {  
 @Autowired  
 private A a;  
}

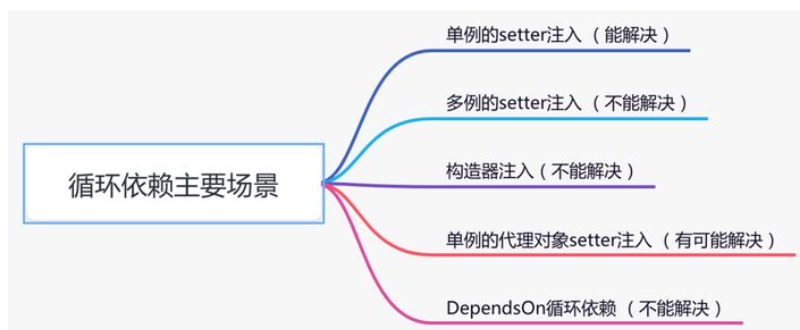
我们声明了a、b两个bean，且a中需要注入一个b，b中需要注入一个a。这就形成了循环依赖，当我们创建A的时候，需要注入B，创建B的时候需要注入A，导致无限循环

常见的循环依赖有如下三种：



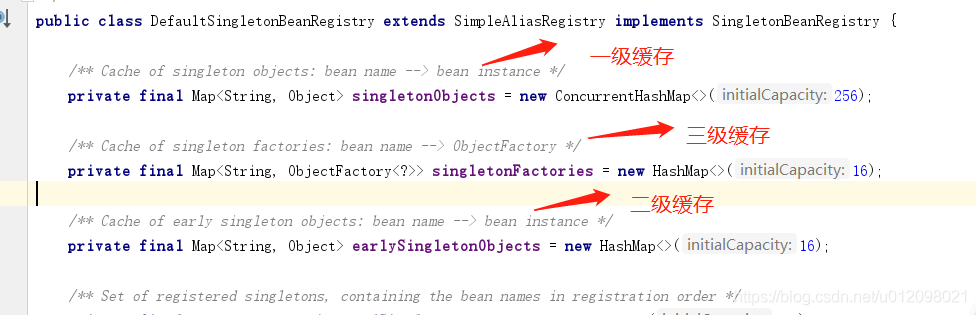


* + 1. 三级缓存机制



Spring支持解决单例情况下通过setter注入的循环依赖问题，但是对于多例模式下，或者构造器注入的循环依赖问题，spring无能为力

为了解决循环依赖问题，spring提供了三级缓存机制，spring在启动过程中，创建了三个map：



1. **singletonObjects 一级缓存，又叫单例池，用于保存实例化、注入、初始化完成的bean实例**
2. **earlySingletonObjects 二级缓存，又叫提前暴露对象，用于保存实例化完成的bean实例(未注入、初始化)**
3. **singletonFactories 三级缓存，用于保存bean创建工厂，以便于后面扩展有机会创建代理对象。**

**关于bean的创建，我们需要明确主要分为两个步骤，一个是实例化，即new的过程，另一个是注入和初始化，即给bean的属性赋值**

**三级缓存本质上是通过允许存在bean的半成品状态(即实例化已完成，但还未初始化)来解决循环依赖的**

* + - 1. 如何通过二级缓存解决循环依赖

下面用一个例子我们看看一级缓存和二级缓存如何解决循环依赖：

1. 首先实例化一个AClass，实例化完成后把自己存到二级缓存「earlySingletonObjects」里面去
2. 初始化AClass，发现需要依赖BClass
3. 试图从一级缓存「singletonObjects」中寻找BClass，没有，然后到二级缓存「earlySingletonObjects」里面寻找发现也没有，于是开始创建BClass
4. 实例化BClass 完成后，把BClass存放到二级缓存「earlySingletonObjects」里面去
5. 尝试初始化BClass，发现需要依赖AClass
6. 试图从一级缓存「singletonObjects」中寻找，显然是没有的，然后到二级缓存「earlySingletonObjects」里面寻找，发现了AClass对象，将AClass注入到BClass中，BClass初始化完成
7. 把BClass放置到一级缓存「singletonObjects」里面，然后将BClass从二级缓存「earlySingletonObjects」删除掉。
8. 重新初始化AClass，从一级缓存「singletonObjects」中找到了BClass，将BClass注入AClass，AClass初始化完成
9. 把AClass放置到一级缓存「singletonObjects」里面，然后把AClass从二级缓存「earlySingletonObjects」删除掉。
   * + 1. 为什么需要三级缓存

从上面的例子可以看出，明明采用二级缓存就能够解决循环依赖，但是Spring为什么使用了三级缓存呢？

如果bean没有使用AOP的话，那么二级缓存是够用的，当时当我们使用了AOP时，对于A\B循环引用，如果A使用了AOP，我们真正希望注入B中的是代理后的对象A，而仅使用二级缓存时，实际注入的是原始的对象A

引入三级缓存之后，第三级缓存中存储的是bean工厂，spring会调用getEarlyBeanReference，判断是否使用了AOP，如果使用了AOP，getEarlyBeanReference将会返回一个代理对象，然后将其放置到二级缓存中(此时就拿到的代理对象，也是未填充属性的，也就是仍然是未初始化完的对象)

所以上面的例子应该是

—————尝试构建AClass—————

1. AClass调用createBeanInstance()方法进行实例化
2. AClass将自己的工厂存放到三级缓存「singletonFactories」中
3. AClass调用populateBean()方法填充属性，发现自己依赖了BClass，然后调用beanFactory.getBean()尝试获取BClass

—————发现AClass依赖BClass，转而构建BClass—————

1. 接着依次调用doGetBean()、getSingleton()重载一(缓存中无BClass)、getSingleton()重载二试图创建BClass、doCreateBean()、createBeanInstance()、将BClass放入三级缓存中同时删除二级缓存的BClass、通过populateBean()方法填充属性填充BClass属性，随后发现BClass需要AClass，然后调用beanFactory.getBean()尝试获取AClass
2. 接着doGetBean(AClass)，getSingleton()重载一，发现三级缓存中有AClass的工厂，接着调用工厂的getObject()方法，即getEarlyBeanReference()方法获取到AClass的bean，接着getSingleton()重载方法一会将三级缓存工厂创建的AClass的bean放到二级缓存中，同时删除三级缓存的AClass工厂对象。
3. 好了，现在我们已经获取到了AClass的bean，那么将它作为doGetBean()的返回值给BClass用于属性注入，接着执行初始化方法，那么BClass就创建好了，这时候createBean()方法完成，退出到getSingleton()重载二中
4. GetSingleton()重载二将BClass放入一级缓存，得到完整的BClass的bean

—————继续构建AClass—————

1. 填充AClass的属性，那么AClass就创建好了，这时候createBean()方法完成，退出到getSingleton()重载二中
2. GetSingleton()重载二将AClass放入一级缓存，得到完整的AClass的bean

如下图:



* + - 1. 缓存移动的时机

好了，我们现在来总结一下多级缓存加入的时机：

1. 通过addSingletonFactory()，将实例化的bean创建工厂加入三级缓存，同时从二级缓存中移除bean；
2. 通过getSingleton()重载一获取三级缓存时，会调用工厂的getObject()方法，即 getEarlyBeanReference()方法，获取提前暴露对象，随后将其从三级缓存移动到二级缓存(如果发生了循环依赖才有这一步，如果没有发生循环依赖，bean将被从三级缓存直接移动到一级缓存)
3. 最后getSingleton()重载二会将实例化完成的bean从二级缓存(或三级缓存中)移动到一级缓存中

下面我们用两个例子来看一看：

* + - * 1. 发生了循环依赖的情况

假设是A依赖于B，B依赖于A：

1. getBean(A)，先实例化A，**然后将A放入三级缓存，同时从二级缓存中移除A**，在A执行populateBean(A)时，发现需要B，则会调用getBean(B)。
2. getBean(B)会先将B实例化，**然后将B放到三级缓存中，同时从二级缓存中移除B**，在B执行populateBean(B)时发现需要A。接着调用getBean(A)
3. getBean(A)会先去getSingleton(A)重载一中尝试从缓存中获取，发现三级缓存中有，那么就会将获取到的**bean A从三级缓存移动到二级缓存中**，得到A的半成品
4. 得到A后回去继续populateBean(B)，接着initializeBean(B)，完成后退出createBean(B)，回到getSingleton(B)重载二中，发现是新创建的bean，则**将B从三级缓存移动到一级缓存中**，此时B的bean创建完成
5. 得到B后回去继续populateBean(A) ，接着initializeBean(A)，完成后退出createBean(A)，回到getSingleton(A)重载二中，发现是新创建的bean，则**将A从二级缓存移动到一级缓存中**，此时A的bean创建完成
   * + - 1. 没有发生循环依赖的情况

如果A依赖于B，而B不依赖于A，即没有发生循环依赖的情况下时：

1. getBean(A)，先实例化A，**然后将A放入三级缓存，同时从二级缓存中移除A**，在A执行populateBean(A)时，发现需要B，则会调用getBean(B)。
2. getBean(B)会先将B实例化，**然后将B放到三级缓存中，同时从二级缓存中移除B**，接着执行populateBean(B)，接着是initializeBean(B)，完成后退出createBean(B)，回到getSingleton(B)重载二中，发现是新创建的bean，**则直接将bean B从三级缓存移动到一级缓存中**，此时B的bean创建完成
3. 得到B后回去继续populateBean(A)，接着initializeBean(A)，完成后退出createBean(A)，回到getSingleton(A)重载二中，发现是新创建的bean，则**将A从三级缓存移动到一级缓存中**，此时A的bean创建完成
   * + 1. spring可以解决的循环依赖

**Spring只能解决单例模式下的setter注入导致的循环依赖**

**多例模式下的循环注入：**多例对象不会放入缓存池，所以必然无法从缓存池中获得对象，而是每次都新建一个对象，会导致无限创建对象，最终导致OOM，无法解决

**单例模式下的构造器注入：**因为加入三级缓存这一步是在实例化Bean后进行的。而实例化A的Bean需要调用构造器，这就要求实例化ClassA的时候，缓存中已经存在ClassB，反过来实例化ClassB的时候，缓存中已经存在ClassA，这是互相矛盾的，所以无法解决单例模式下的构造器循环依赖。

* 1. spring事务
     1. @Transaction注解的实现原理

学过了aop以后，我们大致就能猜出spring是通过aop来实现@Transaction注解的

* 1. 总结spring源码中的设计模式