spring-5.3.20

主代码
ApplicationContext context = new ClassPathXmlApplicationContext("spring/sourcecode/beans.xml");
Company company = (Company) context.getBean("company");
System.out.println(company);

核心流程
容器启动过程

开始

创建ClassPathXmlApplicationContext

调用构造方法

1、调用父类的构造方法
初始化一些属性
创建PathMatchingResourcePatternResolver
用来解析配置文件

2、设置配置文件路径到当前应用程序中

3、开始进入容器的创建和刷新环节
refresh()

生成对象的方式
FactoryBean
Supplier
FactoryMethod
反射创建对象
生成代理对象

1、prepareRefresh()
做容器刷新前的准备工作

1、设置容器的启动时间
2、设置活跃状态为true
3、设置关闭状态为false
4、初始化属性值，此方法需在子类自定义实现
可以在这里设置必需的属性值，启动后做校验不通过导致容器无法启动
5、获取Environment对象，并加载系统属性值到Environment对象中
6、校验必需的属性值
7、准备监听器和事件的集合对象，默认为空的集合

1、如果有容器了，销毁所有bean实例和容器
2、创建容器 createBeanFactory() = DefaultListableBeanFactory
3、加载解析配置文件 loadBeanDefinitions()，具体流程看《加载解析配置文件》

如果开启了component-scan，解析时候会注册以下几个BeanDefinition
ConfigurationClassPostProcessor
AutowiredAnnotationBeanPostProcessor
CommonAnnotationBeanPostProcessor

2、obtainFreshBeanFactory
创建容器，并且完成配置文件的加载

3、prepareBeanFactory
给容器对象完成属性的赋值操作

1、设置标准SPEL表达式解析器
2、注册一系列属性编辑器的注册器
3、注册一系列Bean后置处理器
4、注册一系列自动装配时忽略的接口
5、设置自动装配的类出现规则，如果一个类型有多个实现，使用特定的对象注入
6、将系统环境Bean注册到容器中

4、postProcessBeanFactory
默认没有实现，留给子类进行实现操作

5、invokeBeanFactoryPostProcessor
执行容器后置处理器中的方法
查看《容器后置处理器调用流程》

在这个步骤中

ConfigurationClassPostProcessor通过postProcessBeanDefinitionRegistry方法完成了实例化并解析相关注解
@Component（有内部类就循环解析）
@PropertySource 加载外部配置文件添加到环境变量中
@ComponentScan（扫描）
@Import（导入外部BeanDefinition）
@ImportResource
@Bean

ConfigurationClassPostProcessor通过postProcessBeanFactory为所有的配置类生成了代理类
为什么代理？为了让对象内部本次调用@Bean修饰的方法生成同一个对象

```
@Configuration
public class MyConfiguration {
    @Bean
    public TestA a(){
        return new TestA();
    }
    @Bean
    public TestB b(){
        TestA a = a();
        TestA b = a();
        System.out.println(a == b);
        return new TestB();
    }
}
```

6、registerBeanPostProcessor
排序注册BeanS后置处理器

7、initMessageSource
进行国际化相关的操作

8、initApplicationEventMulticaster
初始化事件多播器

通过观察者模式实现，逻辑执行过程如下
1、发布事件之后会调用多播器来进行事件广播，能够触发具体的监听器去执行操作
2、监听器接收到具体的事件之后，可以验证匹配是否能处理当前事件，如果可以直接处理，如果不行不做任何操作
在spring MVC有运用

9、onRefresh
在spring中默认没有任何实现，模板方法
但是在springboot中启动了web容器

10、registerListeners
注册监听器，接受广播的事件

11、finishBeanFactoryInitialization
完成容器初始化
完成所有非懒加载的单例对象的实例化操作

这里开始创建对象
包括实例化、初始化、循环依赖、AOP等

setConversionService()
设置类型转换服务

addEmbeddedValueResolver()
设置内置的值处理器

freezeConfigurations()
冻结BeanDefinition

preInstantiateSingletons
开始对象的实例化

遍历所有Bean定义的每一个元素来创建实例

这里有缓存

getMergedLocalBeanDefinition
获得合并后的RootBeanDefinition
spring对有父子关系的bean定义进行合并

getBean
实例化

是FactoryBean?

getBean得到这个对象
如果这个对象是FactoryBean类型且需要立即加载
就再次调用getBean实例化真正的对象

是

否

是

否

抛拿类?
不是单例?
懒加载?

是

所有实例化的单例
如果继承了SmartInitializingSingleton
执行afterSingletonsInstantiated方法

详细的实例化步骤

doGetBean

transformedBeanName
转换bean的名称

三级缓存

getSingleton
从缓存中获取

参数不为空
说明需要创建对象而不是从缓存中获取

对象不为空且参数为空

是

否，即对象为空 or 参数不为空

抛出异常
因为原型模式不支持循环依赖

是

如果是原型bean
且对象正在创建中?

否

当前容器有父容器且
beanDefinitionMap中没有这个bean

是，调用父容器的getBean

否

标记这个对象已经被创建过

获得合并后的BeanDefinition

先实例化其依赖的bean
再实例化当前bean

原型作用域对象
Prototype

单例对象

其他作用域的对象
没见过就不做计划了

没有缓存
每次都是实例化新对象

getObjectForBeanInstance
有必要的话需要转换类型

如果这个对象是一个FactoryBean实例就利用这个实例创建真正的对象
如果不是，就直接使用这个对象

adaptBeanInstance

将这个bean对象加入一级缓存
并从二级缓存和三级缓存中删除

afterSingletonCreation
取消正在创建的标记

createBean
实例化

beforeSingletonCreation
标记这个bean正在创建

getSingleton
一级缓存中有的话直接用
没有的话调用ObjectFactory#getObject

详细步骤

结束

否

对象为空?

是

doCreateBean

详细步骤

resolveBeforeInstantiation
1、如果bean有自定义的TargetSource，就不用再执行后续的实例化了，而是直接创建代理对象（目前spring的代码中没有这样的例子）；否则什么都不做
2、shouldSkip方法解析切面类，实例化切面、通知、切点

prepareMethodOverrides
处理lookup methods

解析得到Class对象

详细步骤

createBeanInstance
1、得到beanB的Class对象
2、访问修饰符检查，不满足条件就抛出异常
3、如果这个bean有Supplier，通过Supplier来实例化对象，返回其包装对象
4、如果有factoryMethod，通过factoryMethod实例化对象，返回其包装对象
5、反射使用构造方法生成对象，使用自动注入的构造器或默认的构造器，返回其包装对象

AutowiredAnnotationBeanPostProcessor
@Autowired,@Value
CommonAnnotationBeanPostProcessor
InitDestroyAnnotationBeanPostProcessor
@PostConstruct,@PreDestroy,@Resource

调用MergedBeanDefinitionPostProcessor的后置处理方法修改BeanDefinition
其实就是解析这几个注解

applyMergedBeanDefinitionPostProcessors

单例
循环引用
且对象创建中

是

否

向三级缓存中
添加beanName和lambda表达式
（提前曝光）

populateBean
属性填充

属性填充

创建依赖的bean对象

initializeBean
完成初始化

初始化完成后
就是完整的对象了

registerDisposableBeanIfNecessary
回调，设置需要销毁的对象

1、invokeAwareMethods调用aware接口子类相关的方法，设置容器属性
BeanNameAware
BeanClassLoaderAware
BeanFactoryAware
前面忽略了看三个
2、applyBeanPostProcessorsBeforeInitialization循环调用BeanPostProcessor来完成Before方法的调用
ApplicationContextAwareProcessor完成对Aware接口子类的设置
EnvironmentAware
EmbeddedValueResolverAware
ResourceLoaderAware
ApplicationEventPublisherAware
MessageSourceAware
ApplicationStartupAware
ApplicationContextAware
这几个也是前面忽略的
3、invokeInitMethods执行初始化方法
bean如果实现了InitializingBean接口就调用afterPropertiesSet方法
有自定义init-method方法调用init-method来执行
4、applyBeanPostProcessorsAfterInitialization循环调用BeanPostProcessor来完成after方法的调用
AbstractAutoProxyCreator重载wrapIfNecessary方法生成AOP代理对象，将被短路或责任链
PS：前面的步骤shouldSkip方法已经实例化了切面，切点配通知
PS：其两个子类注解（@EnableAspectJAutoProxy）使用AnnotationAwareAspectJAutoProxyCreator
XML（<aop:aspectj-autoproxy/>）使用AspectJAwareAdvisorAutoProxyCreator

楚河汉界

加载解析配置文件

XML → XmlBeanDefinitionReader

经过一系列的loadBeanDefinitions方法
最终为了完成当前配置文件的加载工作

doLoadBeanDefinitions
实际干活的方法，完成配置文件的解析工作

doLoadDocument
将配置文件解析成文档对象

registerBeanDefinitions
解析标签元素且完成注册功能

doRegisterBeanDefinitions
实际完成注册的操作

parseBeanDefinitions
解析beanDefinition

是否是默认标签

是 → parseDefaultElement
使用默认方式进行解析

否 → parseCustomElement
使用额外的方式进行解析

spring.handlers
根据不同的标签
查找对应的handler处理类

handler.parse

将解析完成的BeanDefinition对象注册到容器的
BeanDefinitionMap和BeanDefinitionNames集合中

容器后置处理器调用流程

开始处理

1、创建一个空的集合，用来存储**已经执行过的容器后置处理器**
Set<String> processedBeans = new HashSet<>()

用户传进来的容器后置处理器是调用的这个方法
AbstractApplicationContext#addBeanFactoryPostProcessor

2、判断当前容器是否是
BeanDefinitionRegistry

否

是

创建两个空的集合，分别用来存储**常规容器后置处理器**和**Bean定义注册表后置处理器**
List<BeanFactoryPostProcessor> regularPostProcessors = new ArrayList<>();
List<BeanDefinitionRegistryPostProcessor> registryProcessors = new ArrayList<>();

遍历处理所有
用户传进来的容器后置处理器
执行后置处理器方法

3、遍历处理所有
用户传进来的容器后置处理器

判断是否是一个
Bean定义注册表后置处理器

是

否

类型转换
执行特定方法postProcessBeanDefinitionRegistry
执行后的对**Bean定义注册后置处理器集合**中

放到**常规容器后置处理器集合**中

4、创建集合对象
用来**暂存Bean定义注册后置处理器**
List<BeanDefinitionRegistryPostProcessor>
currentRegistryProcessors = new ArrayList<>();

5、获取当前容器中所有的Bean定义注册后置处理器

判断是否实现了
PriorityOrdered接口

是

将所有符合条件的对象
对象加入**创建的集合**中
对象的名字加入**创建的集合**中

否

6、同5，并且判断是否实现了Ordered接口

跟上述流程一样

1、将4创建的集合排序

7、同5，如果两个接口都没实现

跟上述流程一样

2、将4创建的集合合并到**Bean定义注册后置处理器集合**中

8、执行2创建的两个集合中
所有容器后置处理器的后置处理方法

567就是按顺序
将容器中的Bean定义注册后置处理器
合并到目标集合并执行特定方法

3、遍历4创建的集合
执行特定方法postProcessBeanDefinitionRegistry

4、清空4创建的集合

总结：这些步骤处理了容器中所有Bean定义注册后置处理器
以及用户传进来的所有容器后置处理器
接下来处理常规的容器后置处理器
1中创建的集合保证了已经执行过后置处理方法的容器后置处理器将跳过

接下来处理常规的容器后置处理器，调用其后置处理方法
优先处理实现了PriorityOrdered接口的
然后是Ordered接口
最后是两者都没有实现的对象

三级缓存与循环依赖

如何解决循环依赖问题？
提前暴露对象，实例化完成的对象就能暴露了，可以被其他对象引用
对象是否完整无所谓，不完整的后续会启动初始化完成
单例模式可以解决循环依赖；原型模式无法解决因为没有利用缓存
构造器注入不能解决循环依赖因为无法实例化，setter注入可以

二级缓存
一级缓存（singletonObjects）：缓存成品，即初始化完成的对象
二级缓存（earlySingletonObjects）：缓存半成品，即实例化完成未完成初始化的对象
三级缓存（singletonFactories）：存的beanName和lambda表达式，用于提前暴露和可能创建的代理对象；
循环引用时执行lambda表达式，将对象或创建的代理对象移动到二级缓存

如果单单是为了解决循环依赖的问题，二级缓存就够用了，那为什么有三级缓存呢？三级缓存是为了代理对象做的
有了二级缓存之后，二级缓存还有必要吗？有必要，三级缓存如果创建了代理对象，必须入二级缓存，否则可能多次创建出不同的代理对象

创建对象过程中，所有对象实例化之后都是先入三级缓存（为了解决循环依赖问题和代理对象问题）
不存在循环依赖的情况下，对象初始化完成之后就从三级缓存中删除，入一级缓存
存在循环依赖的情况下，A依赖B，B已经在三级缓存中，此时B从三级缓存移动到二级缓存，如果需要被代理则入二级缓存的是代理对象

对象查找
三级缓存查找对象，先查一级缓存，有的话直接用，没有的话查二级缓存
二级缓存有的话直接用（初始化后会自动移动到一级缓存），没有的话查三级缓存
三级缓存有的话，返回对象本身或其代理对象，入二级缓存

什么时候生成代理对象
1、不存在循环依赖的情况：
实例化之后暴露，入二级缓存
初始化完成后，在初始中的after方法中生成代理对象，移动到一级缓存
2、存在循环依赖的情况，三级缓存中的元素被当作属性注入，执行lambda表达式创建出代理对象，入二级缓存