// ClassPathXmlApplicationContext 加载配置文件xml，调用refresh()方法

// AbstractXmlApplicationContext 读取配置文件 将配置转换为bean definitions

// AbstractRefreshableConfigApplicationContext 解析配置文件路径的具体实现类

// AbstractRefreshableApplicationContext

// 提供了refresh()方法中的子方法的实现，这些方法可能是空方法，可以让之类来重写，执行refresh方法是回调这些方法

// AbstractApplicationContext 提供refresh方法来实现spring容器的初始化

Refresh方法

1. prepareRefresh()：

为初始化做准备，设置启动时间；

激活初始化的资源文件，initPropertySources()这个方法是一个空方法，可以让子类来重写这个方法,用来加载占位符属性的具体值

提供了验证属性的空方法，

1. obtainFreshBeanFactory()：初始化bean factory也就是创建DefaultListableBeanFactory实例，读取加载配置文件

DefaultListableBeanFactory是整个bean加载的核心部分，是spring注册和加载bean的默认实现

refreshBeanFactory()->>loadBeanDefinitions()

XmlBeanDefinitionReader读取配置文件，将bean的配置读取到beandefinition中

通过getBean方法初始化bean

读取配置文件过程，创建bean过程，解决循环依赖

obtainFreshBeanFactory()🡪

refreshBeanFactory()AbstractRefreshableApplicationContext类中实现，创建DefaultListableBeanFactory对象，调用loadBeanDefinitions方法来将配置文件加载读取到beandefinition中-》

loadBeanDefinitions的实现在AbstractXmlApplicationContext类中

创建XmlBeanDefinitionReader对象，这个类作为读取器，读取配置文件的工作就是由它来完成的

XmlBeanDefinitionReader继承了AbstractBeanDefinitionReader类，经过一系列方法跳转后，最终在doLoadBeanDefinitions方法中实现解析配置文件，解析的方式是DocumentBuilderFactory创建的解析对象DocumentBuilder，将xml读取到Document中后调用registerBeanDefinitions将内容封装到beandefinition中，方法跳转到doRegisterBeanDefinitions实现，解析bean,import,alias

解析import,读取resource的配置，然后交给工厂解析

解析alias，读取name和alias中的配置，保存到ConcurrentHashMap的实例aliasMap中，key是alias，value是name

解析bean：解析bean委派给BeanDefinitionParserDelegate，在processBeanDefinition方法中解析

读取属性

读取id

id是bean的唯一标识符，在spring容器中不可能同时存在两个相同的id；

读取name,

别名（alias），用法：getBean("name")，支持设置多个别名，之间用英文逗号分割；

读取class

类的全限定名（包名+类名），用“.”号连接；

读取parent

指定bean的父类，class属性失效。

创建GenericBeanDefinition对象

读取scope

bean的作用范围，它包括

singleton：单例，指定该bean在spring容器中只有一个对象，所有通过getBean获得的对象都是同一个对象。

prototype：只要重新获取该bean，都将返回一个不同的对象。

request：在一次http请求中对应一个bean，类似于servlet

session：在一次会话中对应一个bean。

读取abstract

设置bean是否为抽象类，默认abstract="false",如果设为true，将不能被实例化；

读取lazy-init

设置bean对象是否懒加载，如果设为true，则应用第一次用到bean时才实例化对象，否则在初始化spring容器时加载**单例**bean对象。（**非单例不实例化**）

读取autowire：自动装配方式

default（默认）：采用父级标签beans中的default-autowire属性；

byName：通过属性名称来自动装配，即A类中的B对象名称为name，那么将根据id="name"找到该bean进行装配，A类必须提供setName方法；

byType：根据属性类型来找到和配置文件中配置的class类型一致的bean来自动装配，如果找到多个类型一致的bean，则抛异常，如果一个都没有找到，则不执行装配操作，也不抛出异常。

no：不执行自动装配操作，只能用<ref标签进行装配；

constructor：根据构造器中参数类型来自动装配，如果找到多个类型一致的bean，则抛异常，如果一个都没有找到，则不执行装配操作，但是抛出异常（这是和byType不一样的地方）。

“autodetect”（spring3之前有该值，从spring4开始该值被抛弃）:通过Bean类的反省机制（introspection）决定是使用“constructor”还是使用“byType”。

autowire-candidate

默认为true，如果为false，那么该bean不能作为其他bean自动装配的候选者

primary

init-method

它的作用是在创建一个bean**之后**调用该方法，初始化方法必须是一个无参方法。

destroy-method

它的作用是在销毁bean**之前**可以执行指定的方法。注意：必须满足scope="singleton"，并且destroy方法参数个数不能超过1，并且参数类型只能为boolean。

factory-method

设置了factory-method属性后，将指定创建bean的工厂方法；

factory-bean

设置了factory-bean属性后，将指定创建bean的工厂类对象，class属性将失效；

读取子元素

Meta

lookup-method

replaced-method

constructor-arg

property

自动注入javabean的成员变量：

qualifier

指定注入的bean名称，一般不使用这种方法，而使用注解@Qualifier("bean")

将创建完成后的beandefinition保存到concurrenthashmap中，key是beanname value就是beandefinition

1. prepareBeanFactory()方法设置bean factory的一些特性
2. postProcessBeanFactory()空方法，子类可以重写
3. invokeBeanFactoryPostProcessors

执行实现了接口BeanDefinitionRegistryPostProcessor和BeanFactoryPostProcessor的bean的方法

如果bean实现了这2个接口，那么就会在invokeBeanFactoryPostProcessors方法调用getBean方法来创建实现了这2个接口的bean

1. registerBeanPostProcessors（创建是spring拦截相关的bean，主要是用来协助业务bean创建的）

获取实现接口BeanPostProcessor的对象，将其保存到beanFactory的CopyOnWriteArrayList集合中

BeanPostProcessor bean的后置处理器，可以再前置初始化方法或者后置初始化方法中获取bean，给bean进行指定的操作（辅助bean的创建）

1. initMessageSource 空方法
2. initApplicationEventMulticaster，初始化事件广播
3. onRefresh 空方法
4. registerListeners 注册监听
5. finishBeanFactoryInitialization （创建业务bean）完成上下文的初始化，判断bean是否延迟加载，如果lazy-load为false，并且是单例，那么就调用getBean初始化，在DefaultListableBeanFactory的方法preInstantiateSingletons中实现
6. finishRefresh 结束初始化

getBean方法

AbstractBeanFactory

非单例每次都创建新的对象，单例创建后就将实例保存在ConcurrentHashMap中，下次直接获取就可以了

首次创建实例

AbstractAutowireCapableBeanFactory的createBean方法来创建

获取class的构造函数，通过反射创建对象，将创建后的对象封装到BeanWrapperImpl中

initializeBean 🡪invokeAwareMethods执行aware（BeanNameAware，BeanClassLoaderAware，BeanFactoryAware接口的方法

AbstractAutowireCapableBeanFactory 的applyBeanPostProcessorsBeforeInitialization方法

执行EnvironmentAware

EmbeddedValueResolverAware

ResourceLoaderAware

ApplicationEventPublisherAware

MessageSourceAware

ApplicationContextAware这些接口对应的方法

通过getBeanPostProcessors方法获取ApplicationContextAwareProcessor实例，然后执行postProcessBeforeInitialization方法，在这个方法中完成接口方法的调用，创建ApplicationContextAwareProcessor实例并放到beanFactory中实在refresh方法中的prepareBeanFactory（）方法中完成的

执行BeanPostProcessor postProcessBeforeInitialization方法

在invokeInitMethods方法中首先执行InitializingBean接口的afterPropertiesSet方法，然后执行init-method配置的方法

执行postProcessAfterInitialization方法

TODO

注解的使用component

Context标签

使用component注解，要在配置文件中添加<context:annotation-config/>,

使用了这个标签，解析context的时候会从spring-context包中的META-INF/spring.handlers中加载对应的处理器

org.springframework.context.config.ContextNamespaceHandler就是使用来解析context标签的，通过反射创建实例后调用init方法来初始化解析器，

AnnotationConfigBeanDefinitionParser：解析annotation-config

然后调用parse方法解析

解析context:component-scan标签使用ComponentScanBeanDefinitionParser，根据component-scan获取ComponentScanBeanDefinitionParser实例，调用ComponentScanBeanDefinitionParser的parse方法，方法参数是节点对象和解析器

--

首先获取basepackage的值

创建ClassPathBeanDefinitionScanner对象，执行doScan方法解析basePackage，最终在scanCandidateComponents方法中实现，

创建PathMatchingResourcePatternResolver匹配器，根据basePackage，找到这个路径下的所有class文件，将文件对象封装到Resource中，

解析Resource，判断是否使用了component注解，使用的是ASM字节码操作框架选择class文件是否使用了这个注解

使用了注解那么就将这个类信息封装到beandefinition中（）

然后将beandefinition注册到DefaultListableBeanFactory中（如果没有指定beanname，那么会使用类名首字母小写来作为beanName）

@Service@Repository@Controller这3个注解也是用了component注解，所以作用和component一样，

(解析META-INF/spring.handlers，使用ClassLoader.getResource方法获取url,然后获取文件流，然后使用Properties加载将数据读取到Properties对象中

ClassLoader.getResource(String name)：在根目录下查找该资源文件，即"/"或classpath目录，使用反射对性能的开销比较大，性能不是很好)

--》aop

Aop有aspect和adVicor2中方式

Aspect的实现（统一用xml配置来说明）

前置通知

后置通知

返回通知

异常通知

环绕通知

定义一个类来定义上面这个通知，每个通知对应一个方法，然后在配置文件中定义这个类的bean,

在aop:config中添加子元素aop：aspect，然后配置aop:pointcut，

aop:before,

aop:after,

aop:after-returning,

aop:after-throwing,

aop:around

<aop:config>

<aop:aspect id=*"aspect"* ref=*"aspectDemo"*>

<aop:pointcut expression=*"execution(\* com.tom.model.\*.\*(..))"* id=*"pointcut"*/>

<aop:before method=*"before"* pointcut-ref=*"pointcut"*/>

<aop:after method=*"after"* pointcut-ref=*"pointcut"* />

<aop:after-returning method=*"afterReturn"* pointcut-ref=*"pointcut"* returning=*"result"*/>

<aop:after-throwing method=*"ex"* pointcut-ref=*"pointcut"* throwing=*"ex"*/>

</aop:aspect>

</aop:config>

Advicor的实现

前置通知：需要实现org.springframework.aop.MethodBeforeAdvice接口来定义通知

返回通知：需要实现org.springframework.aop.AfterReturningAdvice接口来定义通知

异常通知：实现org.springframework.aop.ThrowsAdvice接口，这个接口没有提供方法，需要自定义方法，方法名为afterThrowing，参数为Exception

环绕通知：实现org.aopalliance.intercept.MethodInterceptor

<bean id=*"adviceDemo"* class=*"com.tom.aop.AdviceDemo"*/>

<aop:config>

<aop:pointcut expression=*"execution(\* com.tom.model.\*.\*(..))"* id=*"pointcut"*/>

<aop:advisor advice-ref=*"adviceDemo"* pointcut-ref=*"pointcut"*/>

</aop:config>

<aop:advisor>大多用于事务管理。

<aop:aspect>大多用于日志、缓存。

< aop:aspect>定义切面时，只需要定义一般的bean就行，  
而定义< aop:advisor>中引用的通知时，通知必须实现Advice接口。

Aop标签解析器配置在spring-aop包的META-INFO/spring.handler文件中（AopNamespaceHandler）

解析aop标签是加载spring.Handler,然后创建解析器实例，调用init方法初始化标签的解析器，用ConfigBeanDefinitionParser解析config标签，调用parse解析（解析spring命名空间的标签的都实现BeanDefinitionParser）

解析pointcut

获取pointcut的id和expression属性，创建RootBeanDefinition来封装pointcut的属性，beanclass为AspectJExpressionPointcut的class对象，然后将beandefinition注册到DefaultListableBeanFactory中

解析aspect

获取aspect的属性id和ref，

获取子标签，判断是否有declare-parents（还没有研究过这个是来做什么的），

解析before，after，after-returning，after-throwing，after-throwing,around

1. before

解析before标签，before对应的beandefinition的class就是AspectJMethodBeforeAdvice，创建beandefinition来封装before标签的信息

创建AspectJPointcutAdvisor的beandefinition，class为AspectJPointcutAdvisor，将before的beandefinition封装到这个beandefinition中，将advisor注册到bean factory中

1. after

解析after标签，after对应的beandefinition就是AspectJAfterAdvice，创建beandefinition来封装after标签的信息

创建AspectJPointcutAdvisor的beandefinition，将after的beandefinition封装到这个beandefinition中，将advisor注册到bean factory中

解析after-returning，after-throwing,around

都是如上

after-returning对应的beandefinition就是AspectJAfterReturningAdvice

after-throwing对应的beandefinition就是AspectJAfterThrowingAdvice

around对应的beandefinition就是AspectJAroundAdvice

解析完后，创建对象时就会创建相应实例的代理对象

切面标签对应的AspectJPointcutAdvisor类注册到spring容器中，id为全量类名加上#数字 ，数字用来区分id 比如：org.springframework.aop.aspectj.AspectJPointcutAdvisor#0

创建aop的对象前（也就是在实例化之前applyBeanPostProcessorsBeforeInstantiation），执行后置处理器AspectJAwareAdvisorAutoProxyCreator前置方法，创建标签对应的AspectJPointcutAdvisor对象，执行的代码在org.springframework.aop.framework.autoproxy.BeanFactoryAdvisorRetrievalHelper.findAdvisorBeans()方法中，

创建aop对象,在 org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.initializeBean(String, Object, RootBeanDefinition)方法中创建代理对象；在对象创建之后执行初始化方法之前，再次执行BeanPostProcessor的前置方法（applyBeanPostProcessorsBeforeInitialization）

执行完初始化方法之后执行BeanPostProcessor的后置方法（applyBeanPostProcessorsAfterInitialization），

在AspectJAwareAdvisorAutoProxyCreator 的父类org.springframework.aop.framework.autoproxy.AbstractAutoProxyCreator.postProcessAfterInitialization(Object, String)方法中创建代理对象（目标对象实现了接口就使用JDK动态代理-- JdkDynamicAopProxy，没有实现接口就使用CGLIB-- ObjenesisCglibAopProxy）

CGLIB:使用代理对象调用方法时，在org.springframework.aop.framework.CglibAopProxy.DynamicAdvisedInterceptor. intercept这个方法中生成对应的方法执行chain，依次指向before，目标，after等方法；DynamicAdvisedInterceptor是类CglibAopProxy的内部类，实现了MethodInterceptor接口，在MethodInterceptor接口的intercept方法中实现功能增强,获取代理对象后，调用业务方法，就会执行intercept方法，在这个方法中按照配置执行通知方法和业务方法

Aop方法的执行流程（cglib）

执行业务方法---》执行CGLibAopProxy的内部类DynamicAdvisedInterceptor的intercept方法（DynamicAdvisedInterceptor实现了org.springframework.cglib.proxy.MethodInterceptor接口，代理对象的回调方法就是intercept）

获取目标对象，获取切面的通知方法，创建方法调用器CglibMethodInvocation（继承ReflectiveMethodInvocation方法），业务功能增强就是通过proceed方法实现，获取增强的方法对象

ExposeInvocationInterceptor

AspectJAfterThrowingAdvice—》异常通知

AfterReturningAdviceInterceptor—》返回结果通知

AspectJAfterAdvice—》返回通知

MethodBeforeAdviceInterceptor—》前置通知

[1.@Bean](mailto:1.@Bean),@Autowired,@Resource,@Inject,@Qualifier,

@Primary

@Autowired:方法加载，方法内部的使用参数加载，构造方法加载

[2.@Configuration](mailto:2.@Configuration)

1. AnnotationConfigApplicationContext
2. @ComponentScan(excludeFilters, includeFilters,org.springframework.core.type.filter.TypeFilter—自定义注解过滤器)
3. @Scope(设置bean的作用域，单例，原型，request，session)
4. @Lazy(延迟创建bean，主要是针对单例bean，在启动spring容器的时候不创建bean，在调用getBean方法时才会创建bean实例)
5. @Conditional:实例化之前进行条件判断，参数就是实现org.springframework.context.annotation.Condition接口的类对象，在实现类的matches方法中进行条件判断，返回true就表示可以实例化，返回false就表示不实例化注册到spring容器中
6. @Import:指定导入容器的组件，id为类的全类名
7. ImportSelector：这是一个是接口，返回需要导入到容器的组件的全类名
8. ImportBeanDefinitionRegistrar(模块化管理bean)，实现接口ImportBeanDefinitionRegistrar，重写registerBeanDefinitions方法，方法参数：1 AnnotationMetadata(当前了的注解信息),BeanDefinitionRegistory（beandefinition注册类，把所有需要添加到容器中的bean加入）
9. FactoryBean：bean通过factoryBean注册到spring容器中（bean id前面如果是$,那么获取到的就是factoryBean对象，如果不是用$开始，那么就返回的是getObject中创建的对象）
10. Springbean的生命周期
11. Bean初始化后执行指定方法和bean销毁时执行指定的方法
12. init-method和destroy-method
13. 实现initializingbean接口，重写afterPropertiesSet方法，在初始化后，实现DisposableBean的destroy方法，当bean销毁时执行接口方法destroy方法
14. 使用jsr250规范的注解@PostConstrct（bean创建之后执行）和@PreDestroy（容器关闭时执行）
15. BeanPostProcessor: 有postProcessBeforeInitialization和postProcessAfterInitialization方法，在init-method方法执行之前和执行之后分别执行者2个方法
16. Spring中的aware接口，ApplicationContextAware，BeanNameAware…ApplicationContextAware:获取spring容器
17. Aop(日志，调用方法的参数，方法的返回结果，方法异常)

同时定义了环绕通知和其他的4中通知，先执行环绕通知，然后才执行其他4中方法

1. More