Spring源码

- Spring源码解析
  - 容器
    - IOC
      - 控制反转（Inversion of Control，缩写为IoC）：是面向对象编程中的一种设计原则，可以用来降低代码之间的耦合度
      - 依赖注入（Dependency Injection，简称DI）：是IOC常见的实现方式
      - 为什么要使用springIOC
        - 在日常程序开发过程当中，我们推荐面向抽象编程，面向抽象编程会产生类的依赖，当我们有了一个管理对象的容器之后，类的产生过程也交给了容器，至于我们自己则可以不需要去关心这些对象的产生了。
      - spring实现IOC的思路和方法
        - 应用程序中提供类，提供依赖关系（属性或者构造方法）
        - 把需要交给容器管理的对象通过配置信息告诉容器（xml、annotation，javaconfig三种方式互不干扰可搭配使用）
          - XML：
            - XML文件头添加约束
              -
                ```
                <beans xmlns="http://www.springframework.org/schema/beans"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xmlns:p="http://www.springframework.org/schema/p"
                    xsi:schemaLocation="http://www.springframework.org/schema/beans
                    https://www.springframework.org/schema/beans/spring-beans.xsd">
                ```
            - 两种方式
              - set注入
                - 属性preperty

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- setter injection using the nested ref element -->
    <property name="beanOne">
        <ref bean="anotherExampleBean"/>
    </property>

    <!-- setter injection using the neater ref attribute -->
    <property name="beanTwo" ref="yetAnotherBean"/>
    <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

following example shows the corresponding `ExampleBean` class:

```java
public class ExampleBean {

    private AnotherBean beanOne;

    private YetAnotherBean beanTwo;

    private int i;

    public void setBeanOne(AnotherBean beanOne) {
        this.beanOne = beanOne;
    }

    public void setBeanTwo(YetAnotherBean beanTwo) {
        this.beanTwo = beanTwo;
    }

    public void setIntegerProperty(int i) {
        this.i = i;
    }
```

- 构造器注入
  - constructor-args

```
package x.y;

public class ThingOne {

    public ThingOne(ThingTwo thingTwo, ThingThree thingThree) {
        // ...
    }
}
```

Assuming that `ThingTwo` and `ThingThree` classes are not related by inheritance, following configuration works fine, and you do not need to specify the construct `<constructor-arg/>` element.

```
<beans>
    <bean id="thingOne" class="x.y.ThingOne">
        <constructor-arg ref="thingTwo"/>
        <constructor-arg ref="thingThree"/>
    </bean>

    <bean id="thingTwo" class="x.y.ThingTwo"/>

    <bean id="thingThree" class="x.y.ThingThree"/>
</beans>
```

- annotation：在XML中开启注解（context：annotation-cofig），打开注解扫描（context：component-scan）
  - XML文件头添加约束
    -
    ```
    <?xml version="1.0" encoding="UTF-8"?>
    <beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
            https://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
            https://www.springframework.org/schema/context/spring-context.xsd">
    ```
  - 在XML中开启注解（context：annotation-cofig），打开注解扫描（context：component-scan），并在需要IOC容器管理的类上添加@component注解
- javaconfig：通过一个类在其上添加@Configuration表明这是个配置类，并开启注解扫描@ComponentScan
  - @ ComponentScan参数
    - includeFilters包含，excludeFilters不包含

```
@Configuration
@ComponentScan(basePackages = "org.example",
        includeFilters = @Filter(type = FilterType.REGEX, pattern = ".*Stub.*Repository"),
        excludeFilters = @Filter(Repository.class))
```

- 四种方式注册bean置容器
  - 包扫描+@Component、@Service、@Repository、@controller
  - @ Bean：return 一个Bean对象
    - @ Conditional：按照一定条件给bean容器注册bean，传入的是一个实现Condition接口的数组，满足条件返回ture则注入，否则不注入
  - import
    - 直接import一个类：容器会自动注册这个组件，id默认是全类名
    - importSelector：返回需要导入的组件的全类名数组
      - 实现ImportSelector接口返回参数是一个String数组，放入import的类名

```
public class MySelector implements ImportSelector {
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        return new String[]{UserDaoImp3.class.getName()};
    }
}
```

    - ImportBeanDefinitionRegistrar:手动注册bean到容器中
      - 实现 ImportBeanDefinitionRegistrar接口，由于提供了BeanDefinitionRegistry注册器，可以通过这个注册器的registerBeanDefinition方法手动注册

```
public class MyBeanDefinitionRegistrar implements ImportBeanDefinitionRegistrar {
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {
        RootBeanDefinition bd=new RootBeanDefinition(color.class);
        registry.registerBeanDefinition(color.class.getName(),bd);
    }
}
```

  - FactoryBean
    - 默认获取的是getObject中返回的对象
    - 在Bean前面加上&时获取的是FactoryBean对象

- 把各个类之间的依赖关系通过配置信息告诉容器（通过自动装配可以省略）

- 自动装配（Spring IOC的精髓）
  - 不使用自动装配
    - 除了在类的定义中提供依赖关系，还需要在spring的配置中需要去描述，实际中如果采用此种方式则Bean的配置会非常的繁杂

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- setter injection using the nested ref element -->
    <property name="beanOne">
        <ref bean="anotherExampleBean"/>
    </property>

    <!-- setter injection using the neater ref attribute -->
    <property name="beanTwo" ref="yetAnotherBean"/>
    <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

  - 使用自动装配：只要在类的定义中提供依赖关系，Spring根据配置类型自动装配
    - XML：在约束头末尾加上default-autowire="ByType或ByName或no或constructor "或者在bean中单独指定autowire
      - ByType:根据类型自动装配
        - 如果容器中存在多个同类型的对象，则会抛出异常
      - ByName：根据"setAbc"（XML方式必须使用构造器或set方法）的"Abc"寻找bean id="abc"的对象自动装配
    - 注解：@ Autowired与@ Resource

- @ Autowired：默认按照ByType匹配，会遇到匹配到相同类型多个Bean的问题
  - @ Qualifier("bean的名字")
  - @ primary 指定主数据源
  - 可以放在三个位置
    - 参数位置，构造器位置（如果容器中只有一个则可以省略），方法位置（@ Bean+方法参数，参数从容器中获取）
- @ Resource：可以通过 @Resource(name="beanName") 指定被注入的bean的名称, 要是未指定name属性, 默认使用成员属性的变量名

- springbean的作用域@ scope
  - singleton：单例模式为springIOC默认配置，容器中只会存在一个共享的Bean实例，并且所有对Bean的请求，只要id与该Bean定义相匹配，则只会返回该Bean的 同一实例，无状态的bean应该使用 singleton作用域。
  - prototype：原型模式，在每次对该Bean请求时都会创建一个新的Bean实例，有状态的bean使用Prototype作用域
    - 当singleton中注入prototype的bean时，默认注入的Bean也是单例的，因为在创建Bean的时候，singleton实例只有一次机会创建
    - 那么如何从一个single的对象中拿出prototype呢?
      - 使用@Lookup注解
      - 通过实现ApplicationContextAware接口，调用setApplicationContext

- Spring Bean生命周期回调方法
  - singleTon创建bean时初始化方法执行、容器关闭时销毁方法执行，prototype则是在getBean时才会执行，且不会执行销毁方法，原因是每次生成的Bean对象需要自己销毁

- 第一种方式：@PostConstruct和@PreDestroy是当前Bean声明周期的初始化回调和销毁时回调
  - 基于InitDestoryAnnotationBeanPostProcessor
- 第二种方式：当前类实现InitializingBean和 DisposableBean回调接口；问题是会与Spring的API产生较多耦合
- 第三种方式：通过@Bean指定init-method和destory-method方法
  - 具体对象中实现init()和destory()方法，并通过@Bean（init-method=“init”和destory-method=destory）
- 第四种方式：BeanPostProcessor，在对象初始化前后调用
  - postProcessBeforeInitialization
  - postProcessAfterInitialization

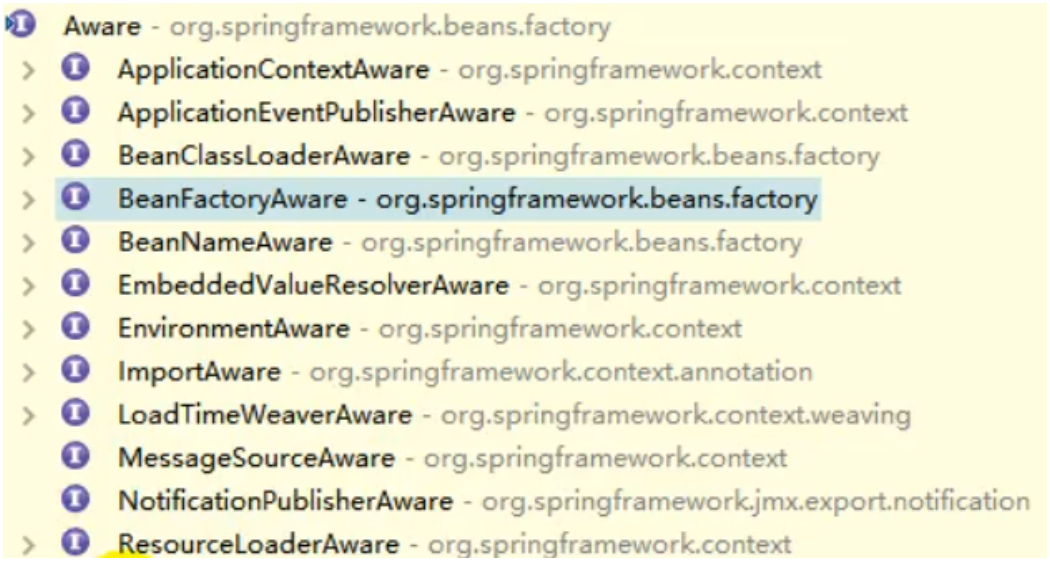
- @ profile用法：是spring提供的一个用来标明当前运行环境的注解，只有当满足profile规定的环境，Bean才会注册到容器中，默认是default
  - 可以对单个Bean使用@ profile注解也可以对整个Beans使用
  - 使@ profile相应的bean环境生效，有两种一种是通过参数设置getEnvironment().setActiveProfiles("abc")，第二种是通过命令行-Dspring.profiles.active=abc
- BeanPostProcessor：bean的后置处理器，程序员可以通过BeanPostProcessor插手bean的实例化过程
  - AbstractAutowireCapableBeanFactory的initializeBean方法是在populateBean（对Bean属性的赋值比如set方法之类）之后执行的，即赋值完再初始化
    - initializeBean()方法中

```
Object wrappedBean = bean;
if (mbd == null || !mbd.isSynthetic()) {
    // 执行后置处理器的 before
    wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
}

try {
    // 执行bean的生命周期回调中的init方法
    // 这里如果是加了@PostConstruct注解，或者实现了InitializingBean这里就是执行afterPropertiesSet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()) {
    //执行后置处理器的 after
    wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
}
```

- ApplicationContextAware：可以在实现该接口的对象中注入ApplicationContext
- InitDestoryAnnotationBeanPostProcessor：生命周期回调方法的@PostConstruct和@PreDestory
- AutoWiredAnnotationBeanPostProcessor：处理@AutoWired注解

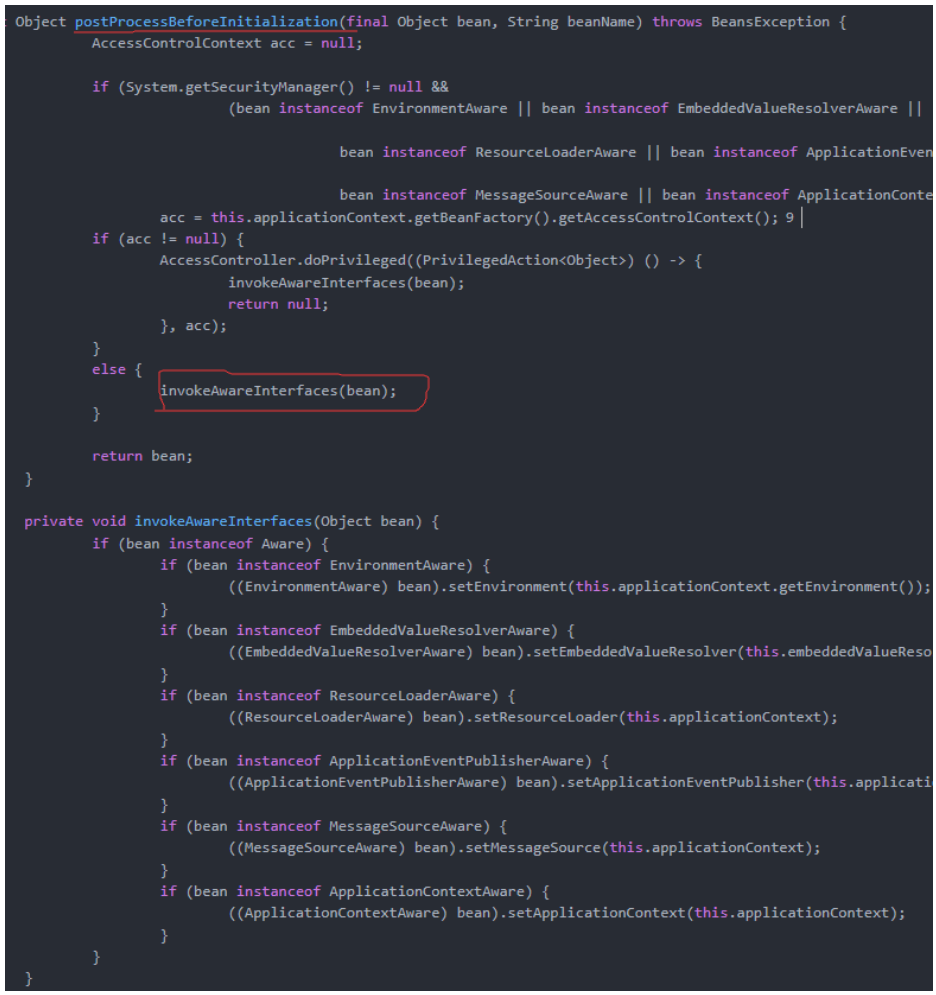
- Aware：在创建对象时，调用规定方法可以注入Spring底层的组件，都通过相关的processor来处理
  - 如ApplicationContext、BeanFactory、ClassLoader等



  - 原理
    - 在AbstractAutowireCapableBeanFactory（实例化Bean时）的initializeBean方法里面，在调用InvokeInitMethods的方法之前。先执行了applyBeanPostProcessBeforeInitialization的方法。这个方法里面就是将所有的XXXAwareProcessor的的

postProcessBeforeInitialization()都执行一遍，会根据invokeAwareInterfaces()代码判断属于哪种类型的Aware再根据类型注入

-



```
Object postProcessBeforeInitialization(final Object bean, String beanName) throws BeansException {
        AccessControlContext acc = null;

        if (System.getSecurityManager() != null &&
                (bean instanceof EnvironmentAware || bean instanceof EmbeddedValueResolverAware ||

                        bean instanceof ResourceLoaderAware || bean instanceof ApplicationEven

                        bean instanceof MessageSourceAware || bean instanceof ApplicationConte
                acc = this.applicationContext.getBeanFactory().getAccessControlContext(); 9
        if (acc != null) {
                AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
                        invokeAwareInterfaces(bean);
                        return null;
                }, acc);
        }
        else {
                invokeAwareInterfaces(bean);
        }

        return bean;
}

private void invokeAwareInterfaces(Object bean) {
        if (bean instanceof Aware) {
                if (bean instanceof EnvironmentAware) {
                        ((EnvironmentAware) bean).setEnvironment(this.applicationContext.getEnvironment());
                }
                if (bean instanceof EmbeddedValueResolverAware) {
                        ((EmbeddedValueResolverAware) bean).setEmbeddedValueResolver(this.embeddedValueReso
                }
                if (bean instanceof ResourceLoaderAware) {
                        ((ResourceLoaderAware) bean).setResourceLoader(this.applicationContext);
                }
                if (bean instanceof ApplicationEventPublisherAware) {
                        ((ApplicationEventPublisherAware) bean).setApplicationEventPublisher(this.applicati
                }
                if (bean instanceof MessageSourceAware) {
                        ((MessageSourceAware) bean).setMessageSource(this.applicationContext);
                }
                if (bean instanceof ApplicationContextAware) {
                        ((ApplicationContextAware) bean).setApplicationContext(this.applicationContext);
                }
        }
}
```

- AOP
  - AOP 概念
    - 与OOP对比，面向切面，传统的OOP开发中的代码逻辑是自上而下的，而这些过程会产生一些横切性问题，这些横切性的问题和我们的主业务逻辑关系不大，这些横切性问题不会影响到主逻辑实现的，但是会散落到代码的各个部分，难以维护。AOP是处理一些横切性问题，AOP的编程思想就是把这些问题和主业务逻辑分开，达到与主业务逻辑解耦的目的。使代码的重用性和开发效率更高
    - 使用场景
      - 日志记录、权限验证、效率检查事务管理、exception等等

- 概念
  - target目标对象也称原始对象中的方法被称为jion point连接点，连接点的集合被称为point cut切点，连接点经过AOP增强的过程叫weaving织入，什么时候织入，织入到哪里被称为advice通知，生成新的对象叫Proxy代理对象
- AOP 实现
  - 过程
    - 开启AOP支持
      - @EnableAspectJAutoProxy（proxyTargetClass=false），默认false表示采用的是JDK的动态代理，true为cglib动态代理

```
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {

}
```

      - XML

```
<aop:aspectj-autoproxy/>
```

    - 声明一个切面Aspect
      - 基于AspectJ注解的切面是一个类
        - @ Componet （第一步先创建Bean并交给IOC）、@ Aspect（第二步添加注解）

```
@Component
@Aspect
public class UserAspect {
}
```

        - @ Aspect
          - 默认是单例的

        - @Aspect("perthis(this(com.chenss.dao.IndexDaoImpl))")+@scope("prototype")表示代理对象是com.chenss.dao.IndexDaoImpl时候应用原型模式，以防止其他切点都采用原型模式
    - 继续XML的切面是一个标签
        -
```
<aop:config>
    <aop:aspect id="myAspect" ref="aBean">
        ...
    </aop:aspect>
</aop:config>
```
- 声明一个切点pointCut
    - 切入点表达式由@Pointcut注释表示
    - 切入点确定感兴趣的 join points（连接点），从而使我们能够控制何时执行通知。Spring AOP只支持Spring bean的方法执行 join points（连接点）
- 声明一个通知Advice
    - advice通知与pointcut切入点表达式相关联，并在切入点匹配的方法执行@Before之前、@After之后或前后运行。
        - @ Before（"pointCut1()&&!pointCut2()"）:满足pointCut1()切点不满足pointCut2()切点的连接点（方法）之前执行
        - @ After（"pointCut1()"）:满足pointCut1()切点的连接点（方法）之后执行
        - @ AfterReturning：正常返回后运行
        - @ AfterThrowing：抛出异常时运行
        - @ around：围绕连接点执行，例如方法调用。这是最有用的切面方式。around通知可以在方法调用之前和之后执行自定义行为。它还负责选择是

继续加入点还是通过返回自己的返回值或抛出异常来快速建议的方法执行

- Proceedingjoinpoint 和JoinPoint的区别:
  - 通过JoinPonit拿到返回方法名，返回值，异常值，需要注意的是JoinPoint要写在参数的最前面

```
@After("com.atguigu.aop.LogAspects.pointCut()")
public void logEnd(JoinPoint joinPoint){
    System.out.println(""+joinPoint.getSignature().getName()+"结束。。。
}

@AfterReturning(value="pointCut()",returning="result")
public void logReturn(Object result){
    System.out.println("除法正常返回。。。@AfterReturning:运行结果：{"+result
}

@AfterThrowing(value="pointCut()",throwing="exception")
public void logException(Exception exception){
    System.out.println("除法异常。。。异常信息：{}");
}
```

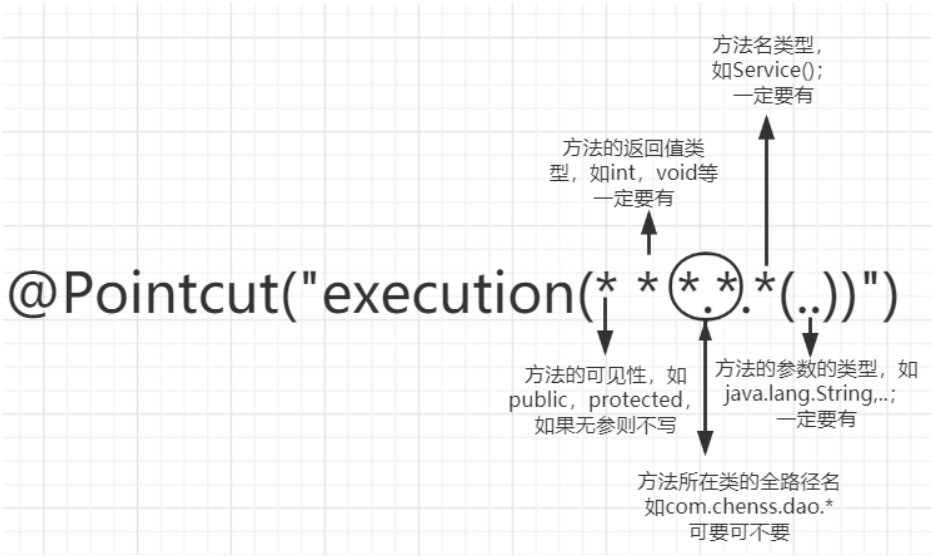
  - JoinPoint的方法
    - 1.java.lang.Object[] getArgs()：获取连接点方法运行时的入参列表；
    - 2.Signature getSignature()：获取连接点的方法签名对象；
    - 3.java.lang.Object getTarget()：获取连接点所在的目标对象；
    - 4.java.lang.Object getThis()：获取代理对象本身；
  - JoinPoint仅能获取相关参数，无法执行连接点。Proceedingjoinpoint 继承了JoinPoint,并扩充实现了proceed()方法，用于执行连接点。proceed()有重载,有个带参数的方法,可以修改目标方法的的参数，通过proceed()这个方法的上下来声明before与after通知

- AspectJ AOP表达式
  - execution：用于匹配方法执行 join points连接点，最小粒度方法，在aop中主要使用
    - execution(方法的可见性? /方法的返回值类型/方法所在类的全路径名?/方法名类型(方法的参数类型) 方

法抛出的异常类型?)这里问号表示当前项可以有也可以没有，其中各项的语义如下

- 方法的可见性，如public，protected；
- 方法的返回值类型，如int，void等；
- 方法所在类的全路径名，如com.spring.Aspect；
- 方法名类型，如buisinessService()；
- 方法的参数类型，如java.lang.String；
- 方法抛出的异常类型，如java.lang.Exception；

- 如图所示



- 详细实例
  - @Pointcut("execution(* com.chenss.dao.*.*(..))")//匹配com.chenss.dao包下的任意接口和类的任意方法
  - @Pointcut("execution(public * com.chenss.dao.*.*(..))")//匹配com.chenss.dao包下的任意接口和类的public方法
  - @Pointcut("execution(public * com.chenss.dao.*.*())")//匹配com.chenss.dao包下的任意接口和类的public 无方法参数的方法
  - @Pointcut("execution(* com.chenss.dao.*.*(java.lang.String, ..))")//匹配com.chenss.dao包下的任意接口和类的第一个参数为String类型的方法

        - @Pointcut("execution(* com.chenss.dao.*.*(java.lang.String))")//匹配com.chenss.dao包下的任意接口和类的只有一个参数，且参数为String类型的方法
        - @Pointcut("execution(public * *(..))")//匹配任意的public方法
        - @Pointcut("execution(* te*(..))")//匹配任意的以te开头的方法
        - @Pointcut("execution(* com.chenss.dao.IndexDao.*(..))")//匹配com.chenss.dao.IndexDao接口中任意的方法
        - @Pointcut("execution(* com.chenss.dao..*.*(..))")//匹配com.chenss.dao包及其子包中任意的方法
    - within：与execution相比，粒度更大，仅能实现到包和接口、类级别。而execution可以精确到方法的返回值，参数个数、修饰符、参数类型等
        - @Pointcut("within(com.chenss.dao.*)")//匹配com.chenss.dao包中的任意方法
        - @Pointcut("within(com.chenss.dao..*)")//匹配com.chenss.dao包及其子包中的任意方法
    - args：args表达式的作用是匹配指定参数类型和指定参数数量的方法,与包名和类名无关
        - @ Pointcut("args(java.lang.String)")//匹配运行时传递的参数类型为指定类型的、且参数个数和顺序匹配
    - target：指向目标接口和子类
        - @ Pointcut("target(com.chenss.dao.IndexDaoImpl)")//目标对象，也就是被代理的对象。限制目标对象为com.chenss.dao.IndexDaoImpl类
            - 由于指向的是目标对象，因此无论使用JDK动态代理还是CGlib都能够是的对象被增强

- this：指向代理对象和接口
  - @ Pointcut("this(com.chenss.dao.IndexDaoImpl)")//当前对象，也就是代理对象，代理对象时通过代理目标对象的方式获取新的对象，与原值并非一个
    - @ EnableAspectJAutoProxy（proxyTargetClass=false），此时目标对象实现类不能够被增强，原因是当前生成代理的对象并非为this(com.chenss.dao.IndexDaoImpl)中的indexDaoImpl。
    - @ EnableAspectJAutoProxy（proxyTargetClass=true），此时目标对象实现类能够被增强，原因是通过cglib生成的代理的对象是this(com.chenss.dao.IndexDaoImpl)中的indexDaoImpl。
  - 关于实体类通过JDK动态代理getBean不能够获得的原因
    - indexDao implements Dao，@ EnableAspectJAutoProxy（proxyTargetClass=false）此时，indexDao indexdao=（indexDao）annotationConfigApplicationContext.getBean("indexDao")报类型不能转换错误
    - 原因是由于JDK动态代理的对象继承了Proxy类，由于单继承原则，不能继续继承其他类，因此只能代理接口，因此在拿不到具体的实现类
- @ args：接受一个参数，并且传递的参数的运行时持有@Chenss注解
  - @ Pointcut("@ args(com.chenss.anno.Chenss")：即目标类的连接点方法传入的参数含有@Chenss注解，如目标类中有test(AOP aop),AOP被@Chenss注解。则该方法就会被增强
- @ annotation：匹配带有@Chenss注解的方法

- @ Pointcut("@annotation(com.chenss.anno.Chenss)")//匹配带有@Chenss注解的方法

- AOP原理
  - @ EnableAspectJAutoProxy
    - @ Import({AspectJAutoProxyRegistrar.class})：利用AspectJAutoProxyRegistrar继承ImportBeanDefinitionRegistrar，从而获得BeanDefinitionRegistry往容器中注入AnnotationAwareAspectJAutoProxyCreator
      - 从InternalAutoProxyCreator=AnnotationAwareAspectJAutoProxyCreator的继承链关系，可以发现实现了SmartInstantiationAwareBeanPostProcessor,和BeanFactoryAware这两个接口
        - SmartInstantiationAwareBeanPostProcessor: bean初始化前后
        - BeanFactoryAware：拿到BeanFactory
    - 创建流程
      - 1、传入配置类，创建IOC容器
      - 2、注册配置类，调用refresh()方法初始化容器
      - 3、refresh()方法中的registerBeanPostProcessors(beanFactory)：向beanFactory中注册BeanPostProcessor，并创建BeanPostProcessor对象保存在容器中
      - 4、创建AnnotationAwareAspectJAutoProxyCreator过程：AbstractAutoWireCapableBeanFactory类中的creatBean()
        - creatBean()【创建Bean到容器】:先通过resolveBeforeInstantiation()判断是否能返回一个代理对象，如果不能则继续

- doCreatBean：createBeanInstance()【创建Bean实例】-->populateBean()【给Bean属性赋值】-->initializeBean()【初始化Bean】
  - initializeBean：初始化Bean
    - invokeAwareMethods()：判断是否属于Aware接口，调用父类AbstractAdvisorAutoProxyCreator的setBeanFactory()传入BeanFactory
    - applyBeanPostProcessorsBeforeInitialization()：应用所有后置处理器的PostProcessorsBeforeInitialization()的方法
    - invokeInitMethods():执行初始化方法：@ PostConstruct等
    - applyBeanPostProcessorsAfterInitialization：应用所有后置处理器的PostProcessorsAfterInitialization()的方法
- 5、AnnotationAwareAspectJAutoProxyCreator：initbeanFactory，并向其中传入Reflec1AspectJAdvisorFactory与BeanFactoryAspectJAdvisorsBuilderAdapter，自此AnnotationAwareAspectJAutoProxyCreator对象创建完毕
- 执行流程
  - AnnotationAwareAspectJAutoProxyCreator本质是InstantiationAwareBeanPostProcessor
    - InstantiationAwareBeanPostProcessor区别于BeanPostProcessor
      - BeanPostProcessor：是在对象创建完成后初始化前后调用的

        - InstantiationAwareBeanPostProcessor是在对象创建前，先通过后置处理器尝试拿到代理对象
    - refresh()方法中的finishBeanFactoryInitialization(beanFactory)：创建剩下的非懒加载的单实例Bean
        - getBean()-->doGetBean()-->getSingleton():能获取则获取不能获取再创建creatBean()之后的步骤和上述创建Bean的过程一样，而通过resolveBeforeInstantiation()能返回一个代理对象，就不走后面的doCreateBean的步骤了
            - 后置处理器此时尝试获取对象

```
bean = applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);
if (bean != null) {
    bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);
}
```

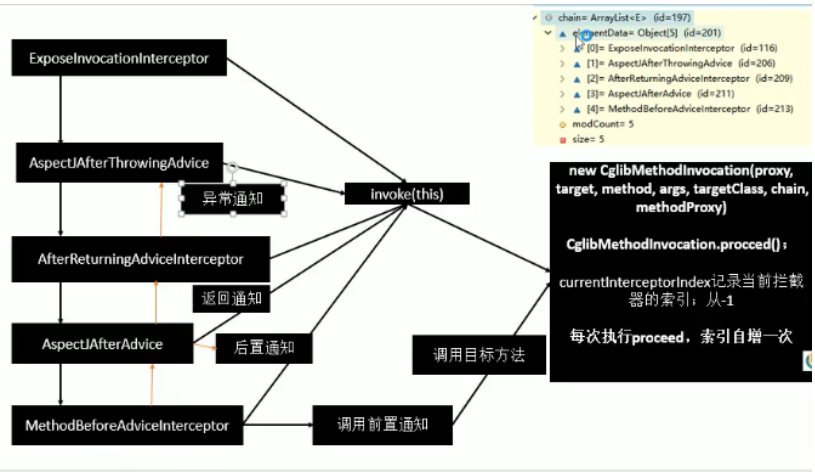
            - 再判断是否属于InstantiationAwareBeanPostProcessor，执行其中的PostProcessBeforeInstantiation，即AnnotationAwareAspectJAutoProxyCreator中的PostProcessBeforeInstantiation方法

```
protected Object applyBeanPostProcessorsBeforeInstantiation(Class<?> beanClass, String beanName) {
    for (BeanPostProcessor bp : getBeanPostProcessors()) {
        if (bp instanceof InstantiationAwareBeanPostProcessor) {
            InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
            Object result = ibp.postProcessBeforeInstantiation(beanClass, beanName);
            if (result != null) {
                return result;
            }
        }
    }
    return null;
}
```

    - AnnotationAwareAspectJAutoProxyCreator会在所有Bean创建之前有一个拦截，那么到底做了什么呢？
        - PostProcessorBeforeInstantiation：先创建切面对象
            - adviseBeans():判断当前Bean是否在adviseBeans（保存了所有需要增强的Bean）中

        - 判断当前Bean是否是基础类型的Adive、pointCut、Advisor、AopInfrastructureBean或者是否为切面
    - PostProcessorAfterInitialization：warpIfNecessary()
        - 获取当前所有增强方法（通知方法）：找到能在当前Bean使用的增强器（找到哪些通知方法是需要切入当前Bean方法的）：通过canApply()切入点表达式进行匹配
        - 对适用的增强器进行排序，并保存在当前Bean的adviseBeans中
        - 如果当前Bean需要增强，通过ProxyFactory创建当前Bean的代理对象：通过是否实现接口等自动决定
            - JDK动态代理
            - cglib动态代理
    - 通过warpIfNecessary()给容器返回一个代理对象，当执行目标方法的时候，代理对象就会执行通知方法的流程，getBean取出的也是代理对象
- 目标方法执行：容器中保存了代理对象的详细信息，比如目标对象，增强器等、会生成一个拦截器链拦截目标方法的执行
    - CglibAopProxy.intercept():拦击目标方法的执行，通过methodInterceptor包装成拦截器
    - 根据ProxyFactory对象获取将要执行的目标方法的拦截器链interceptList
        - 如：ExposeInvocationInterceptor、AspectJAfterThrowingAdvice、AfterReturningAdviceInterceptor、AspectJAfterAdvice、MethodBeforeAdviceInterceptor

        - 传入拦截器链创建一个CglibMethodInvocation对象，并调用proceed()方法执行拦截器链
    - 拦截链的触发过程CglibMethodInvocation
        - currentInterceptorIndex，记录当前拦截器的索引，每次调用proceed()索引增加一次，如果没有拦截器执行目标方法或者索引与拦截器个数-1相等时，执行目标方法
        - 拦截链获取每一个拦截器，并执行invoke方法，等下个拦截器执行完毕再回来执行，由此保证了通知方法与目标方法的执行顺序



- 声明式事务
    - 步骤
        - 开启事务管理功能@EnableTransactionManagement
        - 并在需要事务控制的对象上添加@ Transactional
        - 配置事务管理器PlatformTransactionManager并交给容器
    - 原理
        - @ EnableTransactionManagement
            - 利用import(TransactionManagementConfigurationSelector)给容器中导入组件：AutoProxyRegistrar、ProxyTransactionManagementConfiguration
                - AutoProxyRegistrar

- 给容器注册一个InfrastructureAdvisorAutoProxyCreator组件：利用后置处理器在对象创建以后，包装对象返回一个代理对象，代理对象执行方法时利用拦截器进行增强
- ProxyTransactionManagementConfiguration
  - 给容器中注册事务增强器，和事务拦截器ApplicationEventMulticaster：先获取事务相关属性，再获取PlatformTransactionManager，最后执行目标方法，异常则通过事务管理器rollback(),正常则commit()

- 扩展
  - BeanFactoryPostProcessor
    - 与BeanPostProcessor区别
      - BeanPostProcessor：是Bean的后置处理器，bean创建对象初始化前后工作
      - BeanFactoryPostProcessor：是BeanFactory的后置处理器，在BeanFactory标准初始化之后调用，所有的Bean定义已经保存加载到BeanFactory，但是Bean的实例仍未创建，即可以通过beanFactory取到BeanDefinition，但是不能注册BeanDefinition
    - BeanDefinitionRegistryPostProcessor （extends BeanFactoryPostProcessor）
      - 由于比BeanFactoryPostProcessor多了postProcessBeanDefinitionRegistry(BeanDefinitionRegistry beanDefinitionRegistry) :提供了beanDefinitionRegistry使得实现了BeanDefinitionRegistryPostProcessor可以向容器中注入BeanDefinition
    - 执行过程refresh()-->invokeBeanFactoryPostProcessors()-->postProcessBeanDefinitionRegistry(我们自己定义的BeanDefinitionRegistry)-->invokeBeanDefinitionRegistryPostProcessors(

currentRegistryProcessors系统定义的BeanDefinitionRegistry)--> invokeBeanFactoryPostProcessors(我们自己定义的BeanFactoryPostProcessor)

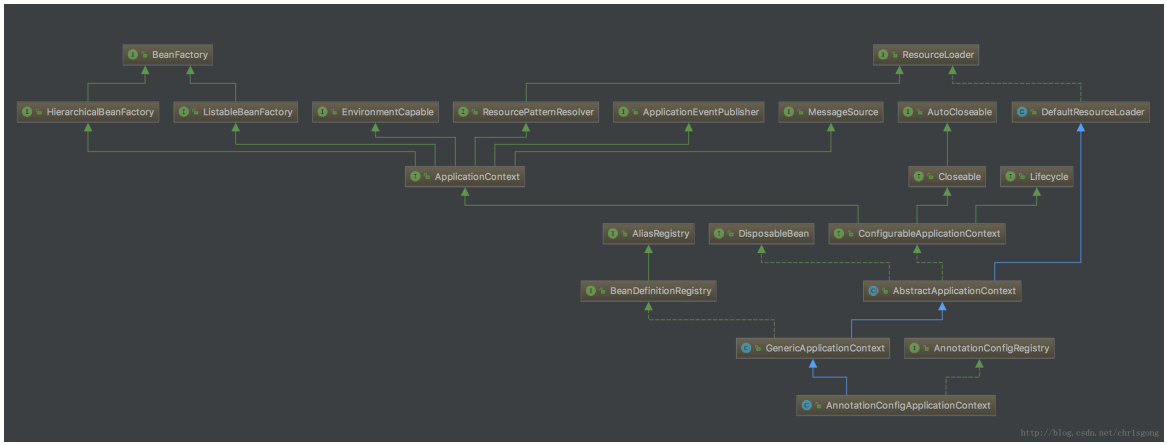
- 1、getBeanFactoryPostProcessors()得到手动获取的（就是程序员自己写的，并且没有交给spring管理，就是没有加上@Component，而是通过AnnotationConfigApplicationContext.addBeanFactoryPostProcessor(new PostProcesspor)手动添加的），并处理自己定义的BeanDefinitionRegistryPostProcessor
  - 内部定义了两个List一个是BeanFactoryPostProcessor，另一个是BeanDefinitionRegistryPostProcessor，如果是BeanDefinitionRegistryPostProcessor，则执行postProcessBeanDefinitionRegistry()方法，根据类型分别添加到两个list中
- 2、List<BeanDefinitionRegistryPostProcessor> currentRegistryProcessors = new ArrayList<>()：这个List主要是维护spring自己实现了BeanDefinitionRegistryPostProcessor接口的对象
  - 其实这里实现BeanDefinitionRegistryPostProcessor接口的只有一个，就是那个最重要的ConfigurationClassPostProcessor，名字叫做internalConfigurationAnnotationProcessor
  - registryProcessors.addAll(currentRegistryProcessors)：这里是把spring的和我们定义的（这里是指加了注解交给spring管理的，不包括我们自己通过实现接口不加注解的）合并
  - invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry)：注意这里是调用方法,注意这里的参数currentRegistryProcessors，它传过来的是spring自己的BeanFactoryPostProcessor， 为什么没有把我们程序员自己定义的传进去？因为我们程序员自己定义的上面已经处理过了

        - invokeBeanFactoryPostProcessors(regularPostProcessors, beanFactory):处理程序员自己定义的实现了BeanFactoryPostProcessor类的后置处理器类
- ApplicationListener：监听容器中发布的事件，完成事件驱动模型开发
    - 步骤
        - 实现ApplicationEvent接口
            - 写一个监听器来监听某个事件（ApplicationEvent及其子类)
            - 把监听器加入到容器
            - 发布一个事件：applicationContext.publishEvent(new ApplicationEvent())
        - 使用@ EventListener注解

```
@EventListener(classes={ApplicationEvent.class})
public void listen(ApplicationEvent event){
    System.out.println("UserService。。监听到的事件："+event);
}
```

            - 使用EventListenerMethodProcessor处理器实现了**SmartInitializingSingleton**接口实现了**afterSingletonInstantiated()**方法
    - 原理
        - getApplicationEventMulticaster()-->getApplicationListeners-->invokeListener(Listener Event)：获取到事件的派发器，再获取到所有的ApplicationListener，最后执行监听器
            - ApplicationEventMulticaster：refresh()-->initApplicationEventMulticaster():初始化ApplicationEventMulticaster，先去容器中找有没有ID=“ApplicationEventMulticaster”的组件，有则直接获取，没有则new SimpleApplicationEventMulticaster();
            - refresh()-->registerListeners()-->getApplicationEventMulticaster().addApplicationListenerBean:从容器中拿到所有的监听器，把他们注册到applicationEventMulticaster中

        - ContextRefreshedEvent： refresh()-->finishRefresh()-->publishEvent(new ContextRefreshedEvent(this))-->上面的步骤
        - ContextClosedEvent：close()-->doClose()-->publishEvent(new ContextClosedEvent(this));-->上面的步骤
- AnnotationConfigApplicationContext()：容器的创建过程
    - 类继承结构



    - 具体实现
        - 构造方法
            - 在这个构造器里初始化了一个Bean定义读取器和扫描器（这个扫描器是后来主动扫描才会用得到）

```
public AnnotationConfigApplicationContext() {
    this.reader = new AnnotatedBeanDefinitionReader(this);
    this.scanner = new ClassPathBeanDefinitionScanner(this);
}
```

            - 主要工作是通过父类的构造函数初始化了一个beanFactory，父类GenericApplicationContext中new了一个DefaultListableBeanFactory，之后初始化了6个BeanDefinition对象放入BeanDefinitionMap中去

```
0 = {ConcurrentHashMap$MapEntry@925} "org.springframework.context.annotation.internalConfigurationAnnotationProcessor"
1 = {ConcurrentHashMap$MapEntry@926} "org.springframework.context.event.internalEventListenerFactory"
2 = {ConcurrentHashMap$MapEntry@927} "org.springframework.context.event.internalEventListenerProcessor"
3 = {ConcurrentHashMap$MapEntry@928} "org.springframework.context.annotation.internalAutowiredAnnotationProcessor"
4 = {ConcurrentHashMap$MapEntry@929} "org.springframework.context.annotation.internalCommonAnnotationProcessor"
5 = {ConcurrentHashMap$MapEntry@930} "org.springframework.context.annotation.internalRequiredAnnotationProcessor"
```

        - refresh()初始化

- prepareRefresh()：初始化工厂类准备工作：包括设置启动时间，是否激活标志位，初始化属性源（property source）配置，但并不属于Bean的生命周期
  - getEnvironment().validateRequiredProperties()
    - ConfigurableEnvironment getEnvironment()
- ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory()：得到一个beanFactory,实际上就是返回了一个DefaultListableBeanFactory
  - refreshBeanFactory();
  - ConfigurableListableBeanFactory beanFactory = getBeanFactory();return beanFactory;
- prepareBeanFactory(beanFactory)：非常重要的一个类，准备一个工厂
  - beanFactory.setBeanClassLoader(getClassLoader())：设置一个类加载器，因为后续我们要根据bd去产生类，产生类就必不可少的需要一个类加载器
  - beanFactory.setBeanExpressionResolver(new StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));：bean表达式解释器
  - beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));：添加一个Spring的后置处理器
- postProcessBeanFactory(beanFactory);留给子类实现
- invokeBeanFactoryPostProcessors(beanFactory)：在BeanFactory标准初始化之后执行的，所谓标准初始化指的就是前面几步执行完成后
  - PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors(beanFactory, getBeanFactoryPostProcessors())# getBeanFactoryPostProcessors()：拿到**自己定义的**BeanFactoryPostProcessor，**即未添加@Component注解的。**

- List<BeanFactoryPostProcessor> regularPostProcessors = new ArrayList<>()与List<BeanDefinitionRegistryPostProcessor> registryProcessors = new ArrayList<>()：遍历自己定义的BeanFactoryPostProcessor，前者放入BeanFactoryPostProcessor，后者放入BeanDefinitionRegistryPostProcessor
- registryProcessor.postProcessBeanDefinitionRegistry(registry)：执行我们自己定义的BeanDefinitionRegistryPostProcessor，有优先级的先执行（判断逻辑与下面BeanPostProcessor一样的）
- List<BeanDefinitionRegistryPostProcessor> **currentRegistryProcessors** = new ArrayList<>()：主要是维护spring自己实现了BeanDefinitionRegistryPostProcessor接口的对象
- String[] postProcessorNames =beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class）在前面提到的构造方法中会给我们在BeanDefinitionMap中默认初始化6个BD对象，但是**只有ConfigurationClassProcessor是BeanDefinitionRegistryPostProcessor类型的**，因此**currentRegistryProcessors** 的**大小只有1**
- invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);接着处理ConfigurationClassProcessor
- invokeBeanFactoryPostProcessors：最后处理剩下的BeanDefinitionRegistryPostProcessor
- ==========上面处理完了BeanDefinitionRegistryPostProcessor==========
- String[] postProcessorNames =beanFactory.getBeanNamesForType(BeanFactoryPo

stProcessor.class):拿到所有BeanFactoryPostProcessor的名字

- List<BeanFactoryPostProcessor> priorityOrderedPostProcessors = new ArrayList<>();List<String> orderedPostProcessorNames = new ArrayList<>();List<String> nonOrderedPostProcessorNames = new ArrayList<>();：三个优先级的BeanFactoryPostProcessor集合
- invokeBeanFactoryPostProcessors(priorityOrderedPostProcessors)--->invokeBeanFactoryPostProcessors(orderedPostProcessors);--->invokeBeanFactoryPostProcessors(nonOrderedPostProcessors);:再按照顺序依次执行

- registerBeanPostProcessors(beanFactory)：向Bean工厂注册BeanPostProcessor
  - List<BeanPostProcessor> priorityOrderedPostProcessors = new ArrayList<>();List<BeanPostProcessor> internalPostProcessors = new ArrayList<>()：分别定义两个BeanPostProcessor集合，分别用来存放BeanPostProcessors，和PriorityOrdered--->如何放的？String[] postProcessorNames = beanFactory.getBeanNamesForType先拿到所有BeanPostPorcessor的名字，再通过beanFactory.isTypeMatch根据类型匹配放入
- initApplicationEventMulticaster()：初始化事件派发器
  - ConfigurableListableBeanFactory beanFactory = getBeanFactory();先拿到BeanFactory，如果容器中存在定义好了的ApplicationEventMulticaster，则取出
  - else new SimpleApplicationEventMulticaster(beanFactory);否则new一个SimpleApplicationEventMulticaster
- onRefresh():子类重写，在刷新的时候重新定义逻辑

- registerListeners();注册ApplicationListener
  - getApplicationEventMulticaster().addApplicationListener(listener);将容器中的listener添加到时间派发器中
- finishBeanFactoryInitialization(beanFactory);初始化所有剩下的单实例Bean
  - beanFactory.preInstantiateSingletons()--->DefaultListableBeanFactory#preInstantiateSingletons
    - List<String> beanNames = new ArrayList<>(this.beanDefinitionNames);先拿到所有的beanName
    - if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()）：当对象满足非抽象、单例、非懒加载，满足再继续下面的步骤
    - getBean()-->doGetBean()-->getSingleton(从singletonObject（是一个ConcurrentHashMap，保存创建好的单实例Bean对象）获取以前缓存中保存的单实例Bean)，如果拿不到则进行bean的创建过程
      - 先标记当前Bean已经被创建，获取Bean的定义信息，判断是否依赖其他的bean，如果依赖则通过getBean拿到依赖的Bean，执行CreateBean
      - Object bean = resolveBeforeInstantiation();：先让InstantiationAwareBeanPostProcessor尝试通过代理的方式拿到对象，如果获取到则触发postProcessorBeforeInstantiation，（**InstantiationAwareBeanPostProcessor的执行时机就在这里**）
      - 如果获取不到，则执行doCreateBean()--->createBeanInstance( )创建Bean实例;--->populateBean()为Bean设置属性;--->拿到InstantiationAwareBeanPostProcessor触发postProcessorAfterInstantiation-->再触发postProcessPropertyValues()拿到一些属性值-->applyPropertyValues()最后将值赋给Bean（**利用**

反射为属性利用setter赋值的执行时机就在这里）；

- initializeBean();-->invokeAwareMethods(BeanNameAware或者BeanClassLoaderAware或者BeanFactoryAware);执行某些Aware接口的方法--->执行后置处理器的applyBeanPostProcessorsBeforeInitialization()方法--->invokeInitMethods();执行bean的生命周期回调中的init方法--->执行后置处理器的applyBeanPostProcessorsAfterInitialization(**AOP的增强就是在AnnotationAwareAspectJAutoProxyCreator的PostProcessorAfterInitialization中通过wrapIfNecessary()方法实现的**);-->registerDisposableBeanIfNecessary注册bean生命周期回调的destory方法
- 自此doCreateBean返回了一个BeanInstance实例,添加到SingletonObjects这个map中

- 检查通过getBean得到的singletonInstance是不是SmartInitializingSingleton接口的，是则执行afterSingletonsInstantiated();（**@ EventListener注解的执行时机就在这里**）

- finishRefresh();完成容器的初始化操作
  - initLifecycleProcessor();初始化LifecycleProcessor处理器，处理实现LifecycleProcessor的bean对象
  - getLifecycleProcessor().onRefresh();调用上述定义的onRefresh()方法
  - publishEvent()容器发布刷新完成事件

- （版权出处：xulilei，转载联系作者）