说明：spring版本是5.2.7

1 Spring常用类 1.1 BeanDefinition class lazy scope primary PropertyValues ConstructorArgumentValues FactoryBeanName InitMethodName 1.2 BeanDefinitionHolder beanDefinition beanName aliases 1.3 BeanDefinitionRegister registerBeanDefinition removeBeanDefinition getBeanDefinition 1.4 BeanDefinitionReader和 AnnotatedBeanDefinitionReader getRegistry getResourceLoader getBeanClassLoader loadBeanDefinitions 1.5 AbstractApplicationContext 1.6 DefaultListableBeanFactory 1.7 BeanPostProcessor postProcessBeforeInitialization postProcessAfterInitialization 1.8 InstantiationAwareBeanPostProcessor postProcessBeforeInstantiation postProcessAfterInstantiation postProcessProperties 1.9 AutowiredAnnotationBeanPostProcessor 1.10 BeanFactoryPostProcessor postProcessBeanFactory 1.11 BeanDefinitionRegistryPostProcessor postProcessBeanDefinitionRegistry 1.12 ConfigurationClassPostProcessor 1.13 Listener 1.14 Event 1.15 Environment 1.16 Aware

2 Spring是如何解决循环依赖的 循环依赖分为两种情况： 1 构造器依赖 2 注入依赖

分为两种模式： 1 单例模式 2 原形模式

归纳起来共四种情况 1 构造器的单例模式 2 构造器的原形模式 3 依赖注入的单例模式 4 依赖注入的原形模式

Spring只能处理3情况的依赖注入，其他不能的原因如下： 如果是单例模式，在getSingleton时：beforeSingletonCreation会检查Set singletonsCurrentlyInCreation这个集合中是否有要创建的对象，如果有直接报BeanCurrentlyInCreationException错误。如果没有创建bean，this.singletonsCurrentlyInCreation.remove(beanName)之后删除beanName 如果是原形模式，参考7，需要注意此时的ThreadLocal prototypesCurrentlyInCreation是个ThreadLoacl变量

3 Spring启动容器的流程 构造方法{ this();//会在此处把SPRING内部的BEAN加入到beanDefinitionMap中，其中最重要的是ConfigurationClassPostProcessor和AutowiredAnnotationBeanPostProcessor这两个BEAN register(componentClasses);//把CONFIG类加入到beanDefinitionMap中 refresh(); }

5 Spring事务的原理，AOP MethodInterceptor invoke(MethodInvocation invocation)->Interceptor->Advice ExposeInvocationInterceptor(是MethodInterceptor.invoke()) AspectJAroundAdvice(是MethodInterceptor.invoke()) MethodBeforeAdviceInterceptor-AspectJMethodBeforeAdvice AspectJAfterAdvice是MethodInterceptor.invoke() AfterReturningAdviceInterceptor-AspectJAfterReturningAdvice AspectJAfterThrowingAdvice是MethodInterceptor.invoke() 非MethodInterceptor的增强器是在CglibAopProxy.intercept()这个执行时获取责任链时，如果非MethodInterceptor则需要对Advisor加强。通过AdvisorAdapter进行转换。 @PointCut表达式的写法 execute execution(* com.xyz.service...(..)) within within(com.xyz.service.*..) 表达式格式：包名.* 或者包名..* this this(com.xyz.service.AccountService) 目标对象使用aop之后生成的代理对象必须是指定的类型才会被拦截，注意是目标对象被代理之后生成的代理对象和指定的类型匹配才会被拦截 target target(com.xyz.service.AccountService) 目标对象为指定的类型被拦截 args args(com.ms.aop.args.demo1.UserModel,..) 匹配第一个参数类型为com.ms.aop.args.demo1.UserModel的所有方法, .. 表示任意个参数 @target @target(com.ms.aop.jtarget.Annotation1) 目标对象中包含com.ms.aop.jtarget.Annotation1注解，调用该目标对象的任意方法都会被拦截 @within @within(com.ms.aop.jwithin.Annotation1) 声明有com.ms.aop.jwithin.Annotation1注解的类中的所有方法都会被拦截 @annotation @annotation(com.ms.aop.jannotation.demo2.Annotation1) 匹配有指定注解的方法（注解作用在方法上面） @args 方法参数所属的类型上有指定的注解，被匹配 @EnableAspectJAutoProxy->@Import(AspectJAutoProxyRegistrar.class->ImportBeanDefinitionRegistry)

AspectJAutoProxyRegistrar->注册 internalAutoProxyCreator:AnnotationAwareAspectJAutoProxyCreator.class AnnotationAwareAspectJAutoProxyCreator extends InstantiationAwareBeanPostProcessor,BeanFactoryAware 所以在registerBeanPostProcessors的时候会把AnnotationAwareAspectJAutoProxyCreator这个对象放到容器中, 在创建目标对象时，在后置处理器中通过proxyFactory生成代理对象 CGLIB 执行postProcessAfterInitialization的流程：wrapIfNecessary-> 1 getAdvicesAndAdvisorsForBean 1.1 findCandidateAdvisors 获取所有增强器-类中所有的方法 1.1.1 super.findCandidateAdvisors() 获取所有Advisor接口的类 1.1.2 aspectJAdvisorsBuilder.buildAspectJAdvisors() 获取所有@AspcetJ的类 --@Befor @After注解增加declarationOrder的排序是通过获取Methods之后，通过method排序（通过解析method上的注解，通过指定的顺序排序） 1.2 findAdvisorsThatCanApply 找出所有增强器中可以使用在该类中的增强器，先for循环所有的增强器，里面for循环class中所有的方法，看是否匹配。 1.3 extendAdvisors(eligibleAdvisors); 在集合首位加入集合首位加入ExposeInvocationInterceptor增强 ExposeInvocationInterceptor的作用是可以将当前的MethodInvocation暴露为一个thread-local对象 1.3 sortAdvisors 根据ORDER排序-只支持切面类级别的排序，不支持方法级别的排序,@Befor @After注解，是通过declarationorder排序的，SortObjet->T\smallerObjects\biggerObjects ->两层循环，第一层是确定位置，第二层是找出增强器，增强器的标准是，后置增强器大的先执行，然后前置增强器小的先执行 首先判断当前Advisor所在的切面类是否实现org.springframework.core.Ordered接口，是的话调用getOrder方法获取 否则判断当前Advisor所在的切面类是否包含org.springframework.core.annotation.Order注解，是的话从注解获取 2 createProxy 2.1 ProxyFactory proxyFactory = new ProxyFactory(); 2.2 Advisor[] advisors = buildAdvisors(beanName, specificInterceptors); proxyFactory.addAdvisors(advisors); 2.3 proxyFactory.getProxy(getProxyClassLoader()); 2.3.1 JdkDynamicAopProxy CglibAopProxy 执行目标方法时 CglibAopProxy.intercept(); Expose->throw->return->after->around->before List chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass); new CglibMethodInvocation(proxy, target, method, args, targetClass, chain, methodProxy).proceed();-->ReflectiveMethodInvocation.proceed(); ReflectiveMethodInvocation.proceed(); if (index == size-1) {//index默认=-1 执行目标方法 } 从增强器List中找出第++index个增强器 if (instanceof InterceptorAndDynamicMethodMatcher) {//before,after,afterreturning,afterthrowing .invoke(); } else {//expose .invoke(); } ExposeInvocationInterceptor.invoke(mi){ old = invocation.get(); invocation.set(mi); try{ return mi.proceed(); } finally { invocation.set(old); } } MethodBeforeAdviceInterceptor.invoke(mi){ AspectJMethodBeforeAdvice.before();->invokeAdviceMethod(getJoinPointMatch(), null, null); return mi.proceed(); } AspectJAfterAdvice.invoke() { try { return mi.proceed(); } finally { invokeAdviceMethod(getJoinPointMatch(), null, null); } } AfterReturningAdviceInterceptor.invoke(mi){ Object retVal = mi.proceed(); AspectJAfterReturningAdvice.afterReturning();->invokeAdviceMethod(getJoinPointMatch(), retValue, null); return retVal; } AspectJAfterThrowingAdvice.invoke(mi) { try { return mi.proceed(); } catch (Throwable ex) { invokeAdviceMethod(getJoinPointMatch(), null, ex); } } invokeAdviceMethod会从当前线程中获取mi AspectJAroundAdvice.invoke(im) { invokeAdviceMethod() -> proceedingJoinPoint.proceed(); }

@EnableTransactionManagement

->@Import(TransactionManagementConfigurationSelector.class) extends ImportSelector

->AutoProxyRegistrar ProxyTransactionManagementConfiguration AutoProxyRegistrar implements ImportBeanDefinitionRegistrar -> 注册了一个 internalAutoProxyCreator=InfrastructureAdvisorAutoProxyCreator ProxyTransactionManagementConfiguration是一个@Configuration -> BeanFactoryTransactionAttributeSourceAdvisor\TransactionAttributeSource(此处的构造器会注入一个SpringTransactionAnnotationParser)\TransactionInterceptor三个bean InfrastructureAdvisorAutoProxyCreator extends InstantiationAwareBeanPostProcessor 执行

postProcessAfterInitialization的流程： wrapIfNecessary-> 1 getAdvicesAndAdvisorsForBean 1.1 findCandidateAdvisors 获取所有增强器-类中所有的方法 1.1.1 super.findCandidateAdvisors() 获取所有Advisor接口的类 在此可以获取BeanFactoryTransactionAttributeSourceAdvisor增强器 1.2 findAdvisorsThatCanApply 同AOP一样，通过二层循环解决，通过TransactionAttributeSource找出所有的匹配的@Transactional注解-先从目标方法上获取、再从目标类上获取、再从接口方法中获取，再从接口类上获取 2 createProxy 2.1 ProxyFactory proxyFactory = new ProxyFactory(); 2.2 Advisor[] advisors = buildAdvisors(beanName, specificInterceptors); proxyFactory.addAdvisors(advisors); 2.3 proxyFactory.getProxy(getProxyClassLoader()); 2.3.1 JdkDynamicAopProxy CglibAopProxy JdkDynamicAopProxy List chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);//此处会获取BeanFactoryTransactionAttributeSourceAdvisor中的Advice，即TransactionInterceptor MethodInvocation invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain); invocation.proceed(); 和AOP的执行流程是一致的 TransactionInterceptor.invoke(mi); invokeWithinTransaction() { //此处获取的TransactionAttributeSource就是注册的TransactionAttributeSource的单例对象 TransactionAttributeSource tas = getTransactionAttributeSource(); //如果非事务方法，则txAttr为NULL，会从方法、类、接口类、接口方法上寻找TransactionAttribute TransactionAttribute txAttr = (tas != null ? tas.getTransactionAttribute(method, targetClass) : null); //如果注解上指定了transactionManager则使用该tm,如果拦截器有tm，如果没有则取TransactionManager类型的tm; TransactionManager tm = determineTransactionManager(txAttr); TransactionInfo txInfo = createTransactionIfNecessary(ptm, txAttr, joinpointIdentification); try { // This is an around advice: Invoke the next interceptor in the chain. // This will normally result in a target object being invoked. retVal = invocation.proceedWithInvocation(); } catch (Throwable ex) { // target invocation exception completeTransactionAfterThrowing(txInfo, ex); throw ex; } finally { cleanupTransactionInfo(txInfo); } if (vavrPresent && VavrDelegate.isVavrTry(retVal)) { // Set rollback-only in case of Vavr failure matching our rollback rules... TransactionStatus status = txInfo.getTransactionStatus(); if (status != null && txAttr != null) { retVal = VavrDelegate.evaluateTryFailure(retVal, txAttr, status); } } commitTransactionAfterReturning(txInfo); return retVal; }

```
            createTransactionIfNecessary() {
                TransactionStatus status = tm.getTransaction(txAttr);
                return prepareTransactionInfo(tm, txAttr,
joinpointIdentification, status);
            }

            getTransaction() {
                Object transaction = doGetTransaction();
                if (isExistingTransaction(transaction)) {
                    // Existing transaction found -> check propagation behavior
to find out how to behave.
                    return handleExistingTransaction(def, transaction,
debugEnabled);
                }
                if (def.getTimeout() < TransactionDefinition.TIMEOUT_DEFAULT) {
                    throw new InvalidTimeoutException("Invalid transaction
timeout", def.getTimeout());
                }
                if (def.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_MANDATORY) {
                    throw new IllegalTransactionStateException(
                            "No existing transaction found for transaction marked
with propagation 'mandatory'");
                }
```

```
                else if (def.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_REQUIRED ||
                        def.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_REQUIRES_NEW ||
                        def.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_NESTED) {
                    SuspendedResourcesHolder suspendedResources = suspend(null);
                    if (debugEnabled) {
                        logger.debug("Creating new transaction with name [" +
def.getName() + "]: " + def);
                    }
                    try {
                        return startTransaction(def, transaction, debugEnabled,
suspendedResources);
                    }
                    catch (RuntimeException | Error ex) {
                        resume(null, suspendedResources);
                        throw ex;
                    }
                }
                else {
                    // Create "empty" transaction: no actual transaction, but
potentially synchronization.
                    if (def.getIsolationLevel() !=
TransactionDefinition.ISOLATION_DEFAULT && logger.isWarnEnabled()) {
                        logger.warn("Custom isolation level specified but no
actual transaction initiated; " +
                                "isolation level will effectively be ignored: " +
def);
                    }
                    boolean newSynchronization = (getTransactionSynchronization()
== SYNCHRONIZATION_ALWAYS);
                    return prepareTransactionStatus(def, null, true,
newSynchronization, debugEnabled, null);
                }
            }

            doGetTransaction() {
                DataSourceTransactionObject txObject = new
DataSourceTransactionObject();
                txObject.setSavepointAllowed(isNestedTransactionAllowed());
                //从ThreadLocal<Map<Object, Object>> resources获取当前线程的数据库链
接
                ConnectionHolder conHolder =
                        (ConnectionHolder)
TransactionSynchronizationManager.getResource(obtainDataSource());
                txObject.setConnectionHolder(conHolder, false);
                return txObject;
            }

            handexlistingTransacti{
                if(never) {
                    throw exception
                }
                if(not_support) {
                    Object suspendedResources = suspend(transaction);
                    return prepareTransactionStatus(definition, null, false,
newSynchronization, debugEnabled, suspendedResources);
```

```
                }
                if(new){
                    SuspendedResourcesHolder suspendedResources =
suspend(transaction);
                    return startTransaction(definition, transaction,
debugEnabled, suspendedResources)
                }
                if(nested) {
                    DefaultTransactionStatus status =
prepareTransactionStatus(definition, transaction, false, false, debugEnabled,
null);
                    status.createAndHoldSavepoint();
                    return status;
                }

//mandantory\never\support\not_support\require\require_new\nested
                if(mandantory || support || require) {
                    return prepareTransactionStatus(definition, transaction,
false, newSynchronization, debugEnabled, null);
                }
            }

            startTransaction {
                boolean newSynchronization = (getTransactionSynchronization() !=
SYNCHRONIZATION_NEVER);
                DefaultTransactionStatus status = newTransactionStatus(
                        definition, transaction, true, newSynchronization,
debugEnabled, suspendedResources);
                doBegin(transaction, definition);
                prepareSynchronization(status, definition);
                return status;
            }

            doBegin{
                if (!txObject.hasConnectionHolder() ||

txObject.getConnectionHolder().isSynchronizedWithTransaction()) {
                    Connection newCon = obtainDataSource().getConnection();
                    txObject.setConnectionHolder(new ConnectionHolder(newCon),
true);
                }

txObject.getConnectionHolder().setSynchronizedWithTransaction(true);
                con = txObject.getConnectionHolder().getConnection();
                Integer previousIsolationLevel =
DataSourceUtils.prepareConnectionForTransaction(con, definition);
                if (con.getAutoCommit()) {
                    txObject.setMustRestoreAutoCommit(true);
                    if (logger.isDebugEnabled()) {
                        logger.debug("Switching JDBC Connection [" + con + "] to
manual commit");
                    }
                    con.setAutoCommit(false);
                }
                txObject.getConnectionHolder().setTransactionActive(true);
                if (txObject.isNewConnectionHolder()) {
```

```
TransactionSynchronizationManager.bindResource(obtainDataSource(),
txObject.getConnectionHolder());
                }
            }

            completeTransactionAfterThrowing{
                if (txInfo.transactionAttribute != null &&
txInfo.transactionAttribute.rollbackOn(ex)) {

txInfo.getTransactionManager().rollback(txInfo.getTransactionStatus());
                } else {

txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());
                }

            }

            rollback{
                if (status.hasSavepoint()) {
                    if (status.isDebug()) {
                        logger.debug("Rolling back transaction to savepoint");
                    }
                    status.rollbackToHeldSavepoint();//回滚之后会释放掉保存点
                }
                else if (status.isNewTransaction()) {
                    if (status.isDebug()) {
                        logger.debug("Initiating transaction rollback");
                    }
                    doRollback(status);
                }else {
                    if (status.hasTransaction()) {
                        doSetRollbackOnly(status);
                    }
                }
                finally {
                    cleanupAfterCompletion(status);
                }
            }

            private void cleanupAfterCompletion(DefaultTransactionStatus status)
{
                    status.setCompleted();
                    if (status.isNewSynchronization()) {
                        TransactionSynchronizationManager.clear();
                    }
                    if (status.isNewTransaction()) {
                        doCleanupAfterCompletion(status.getTransaction());
                    }
                    if (status.getSuspendedResources() != null) {
                        if (status.isDebug()) {
                            logger.debug("Resuming suspended transaction after
completion of inner transaction");
                        }
                        Object transaction = (status.hasTransaction() ?
status.getTransaction() : null);
                        resume(transaction, (SuspendedResourcesHolder)
status.getSuspendedResources());
```

```
                    }
                }
```

```
            protected void cleanupTransactionInfo(@Nullable TransactionInfo
txInfo) {
                    if (txInfo != null) {
                        txInfo.restoreThreadLocalStatus();
                    }
                }

            commit {
                if (status.hasSavepoint()) {
                    status.releaseHeldSavepoint();
                }
                else if (status.isNewTransaction()) {
                    doCommit(status);
                }
                finally {
                            cleanupAfterCompletion(status);
                        }
            }

        CglibAopProxy
```

```
        TransactionManager
         TransactionStatus getTransaction(@Nullable TransactionDefinition
definition)
         void commit(TransactionStatus status)
         void rollback(TransactionStatus status)

        AnnotationTransactionAttributeSource
        TransactionAnnotationParser   TransactionAttribute
parseTransactionAnnotation(AnnotatedElement element)
        TransactionInterceptor
        TransactionStatus       savepoint
        TransactionDefinition   7大传播属性 4种隔离级别
```

@Async注解--同AOP

6 ApplicationContext和BeanFactory的区别 ApplicationContext extends EnvironmentCapable, MessageSource, ApplicationEventPublisher, ResourceLoader

7 Spring的设计模式

1 单例模式

2 工厂模式

3 观察者模式-listener 事件触发者 事件 事件监听者

4 责任链模式

5 代理模式

```
6 适配器模式-aop(非MethodInterceptor转换成)
```

8 CGLIIB 无法为final方法或者类创建代理，无法为static方法创建代理，无法为private方法创建代理 可以使用System.setProperty(DebuggingClassWriter.DEBUG_LOCATION_PROPERTY, "target/cglib");设置CGLIB生成的字节码类生成位置 使用步骤 1 Enhancer enhancer = new Enhancer();创建一个增强器 2 enhancer.setSuperclass(NormalClass.class); 设置它的父类为目标类 3 enhancer.setCallback(new MethodInterceptor() {});设置一个方法拦截器，增强方法 4 NormalClass normalClass = (NormalClass) enhancer.create(); 创建目标代理类 5 normalClass.publicMethod(); 执行代理类的方法 9 Spring-boot启动流程（监听机制） SpringApplication.run(xxx.class, args); new SpringApplication(xxx.class).run(args); 1 new流程： 1.1 确定容器类型 NONE,REACTIVE,SERVLET 1.2 设置初始化器 ApplicationContextInitializer.class getSpringFactoriesInstances 1.3 设置监听器 ApplicationListener.class 2 run流程 2.1 new StopWatch 2.2 stopWatch.start 2.3 配置headless属性 2.4 获取SpringApplicationRunListeners-EventPublishingRunListener（在该对象的构造方法中会添加1.3中的监听器）（此处的对象内部包含一个SimpleApplicationEventMulticaster） 2.5 2.4的的监听器starting方法--调用initMulticaster.multicastEvent(new ApplicationStratingEvent()); 2.6 准备环境变量 prepareEnvironment-把参数赋给环境 调用2.4的对象发布ApplicationEnvironmentPreparedEvent事件-此处会调用监听器完成application-x.properties文件的读取工作 2.7 打印banner printBanner() 2.8 创建容器上下文 createApplicationContext()--根据不同的容器类型生成对应的applicationContext对象 AnnotationConfigApplicationContext AnnotationConfigServletWebServerApplicationContext(生成此对象时，在构造方法中会注册5个常用的beandefinition,跟SPRING容器启动流程中的this()构造方法一致) 2.9 创建异常报告器 SpringBootExceptionReporter.class 2.10 准备容器 prepareContext() 给applicationContext设置属性，environment\resourceLoader等 此处会调用1.2获取的初始化器进行初始化 调用2.4的对象发布ApplicationContextInitializedEvent事件 获取一个ConfigurableListableBeanFactory对象 加载SpringBoot的启动类---通过AnnotatedBeanDefinitionReader把启动类的定义信息加入BeanFactory 调用2.4的对象发布ApplicationPreparedEvent事件 2.11 刷新容器 refreshContext() 2.12 刷新后处理 afterRefresh()--空方法 2.13 stopWatch.stop 计时器停止 2.14 2.4的监听器started方法 容器发布ApplicationStartedEvent事件，此处的applicationEventMulticaster是否和2.4中的一致？与2.4不是同一个对象，但是都是同一个类 同时发布AvailabilityChangeEvent事件 2.15 callRunners 获取容器中的ApplicationRunner对象和CommandLineRunner.class对象，调用他们的run方法--可以进行数据的初始化操作 如果有异常，通过异常报告器处理异常 2.16 2.4的监听器running方法 和2.14流程一样，发布ApplicationReadyEvent方法，发布AvailabilityChangeEvent事件

Spring注解分类 @Component @Controller @Service @Repository @Configuration @Bean @Import @ComponentScan

```
        @Profile
        @ImportResource
        @ComponentScans
        @Primary
        @Lazy
        @PropertySource
```

```
        @ConfigurationProperties--springboot注解
```

```
    @Value
    @Autowired
    @Resource
    @Inject
    @Qualifier
```

@SpringBootApplication @SpringBootConfiguration @Configuration @EnableAutoConfiguration @AutoConfigurationPackage @Import(AutoConfigurationPackages.Registrar.class) @Import(AutoConfigurationImportSelector.class) @ComponentScan SpringApplication.run(PracticeApplication.class, args); ConfigurableApplicationContext applicationContext = new SpringApplication(primarySources).run(args); 1 new SpringApplication 1 设置SpringApplication的primarySources属性为启动类 2 根据classPath中是否有reactive.dispatcher,servlet.dispatcher等类信息来判定容器类型是NONE,SERVLET,REACTIVE（其实是给SpringApplication对象的某个属性赋值） 3 从Spring.Factory文件中读取key=ApplicationContextInitializer的类，并采用反射机制初始化后，放入List集合中（该List集合也是SpringApplication对象的某个属性赋值） 4 从Spring.Factory文件中读取key=ApplicationListener的类，并采用反射机制初始化后，放入List集合中（该List集合也是SpringApplication对象的某个属性赋值） 2 run 1 new StopWatch().start() 2 从Spring.Factory文件中读取key=SpringApplicationRunListener的类 此对象中会实例化EventPublishingRunListener（在该对象的构造方法中会传入上述的SpringApplication对象，还会创建一个SimpleApplicationEventMulticaster对象，并且给Multicaster对象添加1.3中的ApplicationListener） 3 上一步获取的监听器starting方法--调用initMulticaster.multicastEvent(new ApplicationStratingEvent()); 4 prepareEnvironment准备环境 1 根据不同的容器类型，生成不同的Environment，此处的StandardServletEnvironment会加载系统变量、JVM变量、context-param、servlet-init-param、args 2 发布一个initialMulticaster.multicastEvent(new ApplicationEnvironmentPreparedEvent()) 发布的事件会被ConfigFileApplicationListener监听到，该监听器会找spring.profile.active属性，并找/config和classpath下application.yml文件，并把它解析后加入到Environment变量中 5 printBanner 6 createApplicationContext(); 根据不同的容器类型生成对应的applicationContext对象 AnnotationConfigServletWebServerApplicationContext(生成此对象时，在构造方法中会注册5个常用的beandefinition,参考AnnotationConfigApplicationContext的this()构造方法一致) ConfigurationClassPostProcessor AutowiredAnnotationBeanPostProcessor CommonAnnotationBeanPostProcessor EventListenerMethodProcessor DefaultEventListenerFactory 7 准备容器 prepareContext() 给applicationContext设置属性，environment\resourceLoader等 此处会调用1.2获取的初始化器进行初始器，调用initialize方法 调用2.4的对象发布ApplicationContextInitializedEvent事件 加载SpringBoot的启动类---通过AnnotatedBeanDefinitionReader把启动类的定义信息加入BeanFactory 调用2.4的对象发布ApplicationPreparedEvent事件 8 refreshContext() 1 prepareRefresh() 在1中会记录一些状态值，比如closed active 还会初始化earlyApplicationListeners和earlyApplicationEvents两个SET,SET集合是ApplicationContext的属性 2 obtainFreshBeanFactory() 给工厂启个名字，然后返回一个BEAN工厂,如果是XML文件的工厂，此处会解析XML中的内容加入beanDefinitionMap 3 prepareBeanFactory() 给容器添加一些特性比如：classloader\environment\beanPostProcessor 4 postProcessBeanFactory() 留给子类实现的方法，可以在此处在beanFactory中添加特性（添加一些BeanPostProcessor） 5 invokeBeanFactoryPostProcessors() 1 for (容器初始化包含的List) { if (BeanDefinitionRegistryPostProcessor) { registryProcessor.postProcessBeanDefinitionRegistry(registry) } else { List regularPostProcessors.add() } } 2 String[] postProcessorNames =beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class) 3 for (postProcessorNames) { if (PriorityOrdered) { 1 构建单例对象 beanFactory.getBean(ppName) 2 for

(BeanDefinitionRegistryPostProcessor postProcessor : postProcessors) { postProcessor.postProcessBeanDefinitionRegistry(registry);--ConfigurationClassPostProcessor在此执行 } } } 4 for (postProcessorNames) { if (Ordered) { 1 构建单例对象 beanFactory.getBean(ppName) 2 for (BeanDefinitionRegistryPostProcessor postProcessor : postProcessors) { postProcessor.postProcessBeanDefinitionRegistry(registry); } } } 5 for (postProcessorNames) { if (Normal) { 1 构建单例对象 beanFactory.getBean(ppName) 2 for (BeanDefinitionRegistryPostProcessor postProcessor : postProcessors) { postProcessor.postProcessBeanDefinitionRegistry(registry); } } } 6 执行所有BeanDefinitionRegistryPostProcessor的postProcessBeanFactory方法 7 执行上面regularPostProcessors的postProcessBeanFactory方法 8 按照同样的流程执行2-3-4-5的BeanFactoryPostProcessor类型 6 registerBeanPostProcessors() String[] postProcessorNames = beanFactory.getBeanNamesForType(BeanPostProcessor.clas) for (String ppName : postProcessorNames) { if (PriorityOrdered.class) { BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class); List beanPostProcessors.add(pp);--beanPostProcessors是BeanFactory的属性 } else if (beanFactory.isTypeMatch(ppName, Ordered.class)) { BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class); List beanPostProcessors.add(pp);--beanPostProcessors是BeanFactory的属性 } else { BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class); List beanPostProcessors.add(pp);--beanPostProcessors是BeanFactory的属性 } } 7 initMessageSource() 8 initApplicationEventMulticaster() if(存在多播器) { 则使用该多播器 } else { new SimpleApplicationEventMulticaster() } 9 onRefresh() spring.factory文件中 EnableAutoConfiguration=DispatcherServletAutoConfiguration ServletWebServerFactoryAutoConfiguration DispatcherServletAutoConfiguration内部有两个加了@Configuration静态类，这两个类里面有@Bean分别生成了DispatcherServletRegistrationBean（ServletContextInitializer）和DispatcherServlet ServletWebServerFactoryAutoConfiguration中 @Import（EmbeddedTomcat.class,EmbeddedJetty.class,EmbeddedUndertow.class） EmbeddedTomcat中@ConditionalOnClass({ Servlet.class, Tomcat.class, UpgradeProtocol.class })，会创建一个TomcatServletWebServerFactory tomcat启动过程 ServletWebServerApplicationContext.onRefresh(); createWebServer(); 1 根据容器类型获取容器工厂-tomcat\jetty ServletWebServerFactory factory = getWebServerFactory(); 2 通过工厂方法获取相应的容器 this.webServer = factory.getWebServer(getSelfInitializer()); 3 注册优雅关闭对象 getBeanFactory().registerSingleton("webServerGracefulShutdown", new WebServerGracefulShutdownLifecycle(this.webServer)); 4 注册开启关闭对象 getBeanFactory().registerSingleton("webServerStartStop", new WebServerStartStopLifecycle(this, this.webServer)); 5 初始化参数配置 initPropertySources();

```
                getWebServerFactory()流程
                    String[] beanNames =
getBeanFactory().getBeanNamesForType(ServletWebServerFactory.class);
                    return getBeanFactory().getBean(beanNames[0],
ServletWebServerFactory.class);
                factory.getWebServer(getSelfInitializer())
                    其中getSelfInitializer()会返回一个匿名内部类，该类会实现
ServletContextInitializer接口，
                    调用该匿名类的onStartup(ServletContext servletContext)时候，会调
用
                    for (ServletContextInitializer beans :
getServletContextInitializerBeans()) {
                                beans.onStartup(servletContext);
                    }循环IOC容器中所有的ServletContextInitializer对象，调用onStartup
方法
```

```
                    getWebServer流程
                        1 Tomcat tomcat = new Tomcat();
                        2 设置baseDir
                        3 创建Context容器-engine\host\context\wrapper
                        4 context容器中Map<ServletContainerInitializer,Set<Class<?
>>>集合中添加对象-tomcatStarter，此处的TomcatStarter starter = new
TomcatStarter(initializers); TomcatStarter中有个属性ServletContextInitializer[]
                        5 调用tomcat.start
                            startInternal()->StandardContext容器的startInternal()
方法中会循环getSelfInitializer()会返回一个匿名内部类，调用onstartup方法
                            此时会先调用TomcatStarter的onStartup方法，在该方法中调用
DispatcherServletRegistrationBean的onStartup方法，该方法中会把DispatcherServlet加入到
ServletContext容器中
        10 registerListeners()
            1 如果有加载好的ApplicationListener，则加入到ApplicationEventMulticaster
            2 获取所有的ApplicationListener类，加入到ApplicationEventMulticaster
            3 如果earlyApplicationEvents非空，则发布
ApplicationEventMulticaster().multicastEvent()相应的事件
        11 finishBeanFactoryInitialization()
            1 for(beanDefinitionNames) {
                    if(!abstract && isSingleton && !lazy) {
                        if(isFactoryBean) {
                            getBean(&beanName)
                        } else {
                            getBean(beanName)
                        }
                    }
                }
                2 for(beanDefinitionNames) {
                      Object singletonInstance = getSingleton(beanName);
                      if(SmartInitializingSingleton) {
                        smartSingleton.afterSingletonsInstantiated();----
EventListenerMethodProcessor.afterSingletonsInstantiated处理@EventListener注解，内
部是把有注解的方法转换成ApplicationListener对象
                      }
                   }
             getBean流程:
              AbstractBeanFactory.doGetBean()
               1 transformedBeanName->处理factorybean类型对象名字
               2 getSingleton->获取单例对象,此处主要是从缓存中获
取,singletonObjects,earlySingletonObjects,singletonFactorie，如果singletonFactorie
中存在，则把对象从singletonFactorie删除，并加入到earlySingletonObjects对象中
               3 如果单例对象存在则getObjectForBeanInstance->获取真正的BEAN，处理
factorybean的对象
               4 如果单例对象不存在
isPrototypeCurrentlyInCreation(ThreadLocal<Object>
prototypesCurrentlyInCreation)->检查对象是否在创建，如果在创建则报
BeanCurrentlyInCreationException
               5 markBeanAsCreated->把对象标记为创建中 Set<String> alreadyCreated
               6 如果是单例对象
                 6.1 getSingleton(beanName, ObjectFactory)
                     6.1.1 beforeSingletonCreation(beanName);--检查该对象是否在创建
中，singletonsCurrentlyInCreation放入到这个Set集合中
                     6.1.2 singletonObject = singletonFactory.getObject();//调用工
厂方法生成对象
                        AbstractAutowireCapableBeanFactory.createBean()的调用流程:
                            6.1.2.1 Object bean =
resolveBeforeInstantiation(beanName, mbdToUse);
```

在实例化前，如果InstantiationAwareBeanPostProcessor存在，
则调用postProcessBeforeInstantiation创建对象， --AOP增强器在这个位置添加
再执行所有BeanPostProcessor的postProcessAfterInitialization方法，如果有值，直接返回，流程结束。
6.1.2.2 Object beanInstance = doCreateBean(beanName, mbdToUse, args);
AbstractAutowireCapableBeanFactory.doCreateBean()的调用流程:
6.1.2.2.1 BeanWrapper instanceWrapper = createBeanInstance(beanName, mbd, args);返回BeanWrapper，BEAN的封装对象
6.1.2.2.2 addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean))把BEAN加入到singletonFactories中（SmartInstantiationAwareBeanPostProcessor）
6.1.2.2.3 populateBean(beanName, mbd, instanceWrapper);
1 执行InstantiationAwareBeanPostProcessor.postProcessAfterInstantiation方法
2 InstantiationAwareBeanPostProcessor.postProcessProperties @AutoWired @Resource @Inject
6.1.2.2.4 exposedObject = initializeBean(beanName, exposedObject, mbd);
1 invokeAwareMethods->执行实现了BeanNameAware，BeanClassLoaderAware，BeanFactoryAware接口的属性
2 执行BeanPostProcessor处理器的postProcessBeforeInitialization方法 CommonAnnotationBeanPostProcessor.before方法可以处理@PostConstruct
3 invokeInitMethods->
3.1 执行实现了InitializingBean的方法
3.2 执行指定了init_method()的方法
4 执行BeanPostProcessor处理器的postProcessAfterInitialization ---AOP在此生成代理对象
6.1.2.2.6 registerDisposableBeanIfNecessary(beanName, bean, mbd);
6.1.3 afterSingletonCreation(beanName);--把这个beanName从singletonsCurrentlyInCreation集合中删除
6.1.4 addSingleton(beanName, singletonObject);--把单例对象放入SingletonObject的Map中，并把它从earlySingletonObject的MAP和SingletonFactory的MAP中删除.
6.2 通过6.1返回的对象，调用getObjectForBeanInstance，原理同上3
7 如果是原型模式
7.1 beforePrototypeCreation -> 在prototypesCurrentlyInCreation中加入当前的BEAN
7.2 createBean->创建对象 同6.1
7.3 afterPrototypeCreation-> 把prototypesCurrentlyInCreation中的BEAN删除
7.4 getObjectForBeanInstance->同3
8 如果是其他SCOPE
同7
12 finishRefresh()
1 initLifecycleProcessor().onRefresh()--调用实现了Lifecycle接口的Bean的start()方法
2 publishEvent(ContextRefreshedEvent())
9 刷新后处理 afterRefresh()--空方法

```
    10 stopWatch.stop  计时器停止
    11 发布ApplicationStartedEvent事件
    12 callRunners---获取容器中的ApplicationRunner对象和CommandLineRunner.class对象，调用他们的run方法--可以进行一些事件的初始化操作
    13 发布ApplicationReadyEvent方法
```

ConfigurationClassPostProcessor工作流程
ConfigurationClassPostProcessor.postProcessBeanDefinitionRegistry();

```
        1 获取所有的BeanDefinition   registry.getBeanDefinitionNames()

        2 循环所有的BeanDefinition
ConfigurationClassUtils.checkConfigurationClassCandidate 找出加了@Configuration注解的类（只会有默认启动的类和@SpringBootApplication的类，此时只能获取@SpringBootApplication的类）
        checkConfigurationClassCandidate()会判断一个是否是一个配置类,并为BeanDefinition设置属性为lite或者full。
        在这儿为BeanDefinition设置lite和full属性值是为了后面在使用
        如果有@Configuration注解&&proxyBeanMethods=true则设置beanDefinition中的configurationClass的Attribute=full
        如果@Component、@ComponentScan、@Import、@ImportResource、@Bean则设置为lite

        3 ConfigurationClassParser.parse(2中的配置类)
            3.1 doProcessConfigurationClass()
                3.1.1 如果有@Component&&如果有内部类&&内部类是@Configuartion类，则执行3的流程
                3.1.2 是否有@PropertySSource注解
                    如果有则把相应的属性加入到Environment中
                3.1.3 是否有@ComponentScan注解
                    3.1.3.1 ComponentScanAnnotationParser.parse()解析@SpringBootApplication注解的类，获取相关的BeanDefinition信息
                        3.1.3.1.1 此处如果没有配置扫描的包路径，默认取@SpringBootApplication类所在的包路径
                        3.1.3.1.2 ClassPathBeanDefinitionScanner.doScan(),扫描指定的路径
                            3.1.3.1.2.1 获取指定路径下的所有包含@Component的定义信息
                               3.1.3.1.2.1.1 扫描指定路径下的所有类
                               3.1.3.1.2.1.2 获取其中有@Component注解的类
                            3.1.3.1.2.2 注册上一步获取的BeanDefinition
                    3.1.3.2 循环上一步获取的BeanDefinition信息，重复2、3、4、5的流程
                3.1.4 处理@Import注解
                        3.1.4.1 如果是ImportSelector，则调用selectImports，得到相应的Bean,转换成SourceClassses,继续调用3.1.4进行@Import的解析
                        3.1.4.2 如果是ImportBeanDefinitionRegistrar，则把key=注册器 value=注解元信息加入到configclasss的map中。 Map<ImportBeanDefinitionRegistrar, AnnotationMetadata>
                        3.1.4.3 如果不是上述两种，则当成@Configuration类处理重复3的流程
                3.1.5 处理@ImportResource注解
                        如果存在则加入到configClass的Map<String, Class<? extends BeanDefinitionReader>>中
                3.1.6 处理@Bean方法
                        3.1.6.1 找出asm可以处理的@Bean方法
                        3.1.6.2 把它加入到configClass的Set<BeanMethod>中
                3.1.7 处理接口中的默认方法
                        3.1.7.1 获取所有的接口
```

```
                        3.1.7.2 找出接口中所有@Bean注解的方法
                        3.1.7.3 如果@Bean的方法不是abstract方法，则加入到configclass的
Set<BeanMethod>中
                3.1.8 处理父类
                        如果有父类，并且父类不是java开头，并且不是已经知道的父类，则把该父
类当成sourceClass，继续3的流程
                3.1.9 流程处理完，返回null
            3.2 把解析完的configClass加入Map中

        4 parser.getConfigurationClasses()，获取所有3.2中的Map的key值。

        5
ConfigurationClassBeanDefinitionReader.loadBeanDefinitions(configClassess) 加载所
有的配置类
            5.1 如果是@Import          configClass.isImported()
                new BeanDefinition()对象，通过解析@Lazy、@Primary、@DependsOn、
@Role、@Description注解给Beandefinition对象赋属性值。
                然后把该BeanDefinition注册到容器中
            5.2 如果有@Bean注解          configClass.getBeanMethods()---如果一个配置类
上有多个@Bean方法，则生成多个BeanDefinition
                new BeanDefinition()对象（该对象试试@Bean注解所在类的对象定义，不是@Bean
返回对象的对象定义），设置
                beanDef.setFactoryBeanName(configClass.getBeanName());
                beanDef.setUniqueFactoryMethodName(methodName)这两个属性。
                通过解析@Lazy、@Primary、@DependsOn、@Role、@Description注解给
Beandefinition对象赋属性值。
                再解析@Bean注解的属性给BeanDefinition属性赋值。
                解析@Scope给对象赋值
                然后把该BeanDefinition注册到容器中
            5.3 如果有@ImportSource注解  configClass.getImportedResources()
                xml文件解析
            5.4 如果有BeanDefinitionRegistry
configClass.getImportBeanDefinitionRegistrars()
                执行所有的BeanDefinitionRegistry.registryBeanDefinition方法注册
BeanDefinition

        6 如果5解析之后又产生了新的类，则重复执行2、3、4、5的动作

postProcessorBeanFactory
    1 enhanceConfigurationClasses(beanFactory)---增强配置类(为full的类进行CGLIB加强)
      对加了@Configuration注解的配置类进行Cglib代理
            添加了两个MethodInterceptor。(BeanMethodInterceptor和
BeanFactoryAwareMethodInterceptor)
            通过这两个类的名称，可以猜出，前者是对加了@Bean注解的方法进行增强，后者是为代理对
象的beanFactory属性进行增强
            被代理的对象，如何对方法进行增强呢？就是通过MethodInterceptor拦截器实现的
            类似于SpringMVC中的拦截器，每次执行请求时，都会对经过拦截器。
            同样，加了MethodInterceptor，那么在每次代理对象的方法时，都会先经过
MethodInterceptor中的方法
    2 增加ImportAwareBeanPostProcessor
```

这个类是完成Spring自动装配的关键，SpringBoot注解上
@Import(AutoConfigurationImportSelector.class)的注解在这个地方被解析
AutoConfigurationImportSelector implements ImportSelector(selectImports) selectImports() { 1 从META-INF/spring.factories中找出key=EnableAutoConfiguration的所有类 2 获取所有的排除类信息，排除掉，返回所有的自动装配的类 }