

[首页](#) / [专栏](#) / [java](#) / 文章详情

# spring源码导读：别怕，文章里没有贴代码

简相杰 发布于 2020-05-20

🔥🔥🔥 SegmentFault D-Day Online 开源开放与新技术创新，快来报名 >>>

1. 本文是一篇spring源码相关的文章。众所周知，读源码是一件令人恐惧的事情，看源码相关的博客、书和文章亦是如此。（笔者曾经狂啃mybatis的设计与实现这本书，结果是看着超级痛苦，看完以后没留下啥印象）。鉴于此，本文的源码解析就不按照传统的去贴代码的方式去讲解spring源码了，本文的源码解析以流程的方式来讲解spring在每一步都干了什么，所以谓之曰spring源码导读。。。。
2. 在正式开始spring源码导读之前，读者总得知道spring里的各个标签是干啥的吧，因此文中前一部分罗列了spring常见的注解用法。并搞了点SpringAOP和spring事务源码的解析作为后面正式开始的导读的开胃菜
3. 介绍完了，让我们开始吧！！！！。

## spring注解

- @Configuration 用于标注配置类
- @Bean 结合@Configuration (full mode) 使用或结合@Component (light mode) 使用。可以导入第三方组件,入方法有参数默认从IOC容器中获取，可以指定initMethod和destroyMethod 指定初始化和销毁方法,多实例对象不会调用销毁方法。
- 包扫描@ComponentScan (@ComponentScans可以配置多个扫描,@TypeFilter:指定过滤规则,自己实现TypeFilter类)  
组件(@Service、@Controller、@Repository):包扫描+组件注解导入注解。
- @Scope:设置组件作用域 1.prototype:多例的2.singleton:单例的（默认值）
- @Lazy 懒加载
- @Conditional({Condition}):按照一定的条件进行判断,满足条件给容器中注册Bean,传入Condition数组,, 使用时需自己创建类继承Condition然后重写match方法。
- @Import[快速给容器中导入一个组件]
  1. Import(类名),容器中就会自动注册这个组件，id默认是组件的全名
  2. ImportSelector: 返回需要导入的组件的全类名的数组
  3. ImportBeanDefinitionRegistrar: 手动注册bean
- FactoryBean:工厂Bean,交给spring用来生产Bean到spring容器中.可以通过前缀&来获取工厂Bean本身。
- @Value:给属性赋值,也可以使用SpEL和外部文件的值

- @PropertySource:读取外部配置文件中的k/v保存到运行环境中,结合@value使用,或使用ConfigurableEnvironment获取
- @Profile:结合@Bean使用,默认为default环境,可以通过命令行参数来切换环境
- 自定义组件使用Spring容器底层的组件:需要让自定义组件实现xxxAware, (例如:ApplicationContextAware),spring在创建对象的时候,会帮我们自动注入。spring通过BeanPostProcessor机制来实现XXXXAware的自动注入。

ApplicationContextProcessor.java

```
private void invokeAwareInterfaces(Object bean) {
    if (bean instanceof Aware) {

        if (bean instanceof ResourceLoaderAware) {
            ((ResourceLoaderAware)bean).setResourceLoader(this.applicationCc
        }

        if (bean instanceof ApplicationContextAware) {
            ((ApplicationContextAware)bean).setApplicationContext(this.appli
        }
    }
}
```

- @Autowried 装配优先级如下:
  1. 使用按照类型去容器中找到对应的组件
  2. 按照属性名称去作为组件id去找对应的组件
- @Qualifier:指定默认的组件,结合@Autowried使用
  - 标注在构造器:spring创建对象调用构造器创建对象
  - 标注在方法上:
- @Primary:spring自动装配的时候,默认首先bean,配合@Bean使用
- @Resource(JSR250):jsr规范:按照组件名称进行装配
- @Inject(JSR330):jsr规范和@Autowired功能一致,不支持require=false;

## Bean生命周期:

### 初始化和销毁

1. 通过@Bean 指定init-method和destroy-method
2. 实现InitializingBean定义初始化逻辑,实现DisposableBean定义销毁方法
3. 实现BeanPostProcessor接口的后置拦截器放入容器中,可以拦截bean初始化,并可以在被拦截的Bean的初始化前后进行一些处理工作。

spring底层常用的BeanPostProcessor:

- \* BeanValidationPostProcessor用来实现数据校验
- \* AutowireAnnotationBeanPostProcessor, @Autowire实现
- \* ApplicationContextProcessor实现XXXAware的自动注入。

## 执行时机

doCreateBean

-populateBean () : 给bean的各种属性赋值

-initializeBean () : 初始化bean

-处理Aware方法

-applyBeanPostProcessorsBeforeInitialization: 后置处理器的实例化前拦截

-invokeInitMethods: 执行@Bean指定的initMethod

-applyBeanPostProcessorsAfterInitialization: 后置处理器的实例化后拦截

## SpringAOP实现原理

### 使用步骤

1. @EnableAspectJAutoProxy 开启基于注解的aop模式
2. @Aspect: 定义切面类，切面类里定义通知
3. @PointCut 切入点，可以写切入点表达式，指定在哪个方法切入
4. 通知方法
  - @Before(前置通知)
  - @After(后置通知)
  - @AfterReturning(返回通知)
  - @AfterThrowing(异常通知)@Around(环绕通知)
5. JoinPoint: 连接点,是一个类，配合通知使用，用于获取切入的点的信息

### SpringAop原理

1. @EnableAspectJAutoProxy
  - @EnableAspectJAutoProxy 通过@Import(AsspectJAutoProxyRegistrar.class)给spring容器中导入了一个AnnotationAwareAspectJAutoProxyCreator。
  - AnnotationAwareAspectJAutoProxyCreator实现了InstantiationAwareBeanPostProcessor, InstantiationAwareBeanPostProcessor是一个BeanPostProcessor。它可以拦截spring的Bean初始化(Initialization)前后和实例化(Initialization)前后。
2. AnnotationAwareAspectJAutoProxyCreator的postProcessBeforeInstantiation(bean实例化前): 会通过调用isInfrastructureClass(beanClass)来判断 被拦截的类是否是基础类型的Advice、PointCut、Advisor、AopInfrastructureBean，或者是否是切面 (@Aspect)，若是则放入adviseBean集合。这里主要是用来处理我们的切面类。

3. AnnotationAwareAspectJAutoProxyCreator的BeanPostProcessorsAfterInitialization (bean初始化后)：
  1. 首先找到被拦截的Bean的匹配的增强器（通知方法），这里有切入点表达式匹配的逻辑
  2. 将增强器保存到proxyFactory中，
  3. 根据被拦截的Bean是否实现了接口，spring自动决定使用JdkDynamicAopProxy还是ObjenesisCglibAopProxy
  4. 最后返回被拦截的Bean的代理对象，注册到spring容器中
4. 代理Bean的目标方法执行过程：CglibAopProxy.intercept();
  1. 保存所有的增强器，并处理转换为一个拦截器链
  2. 如果没有拦截器链，就直接执行目标方法
  3. 如果有拦截器链，就将目标方法，拦截器链等信息传入并创建CglibMethodInvocation对象，并调用proceed()方法获取返回值。proceed方法内部会依次执行拦截器链。

## spring 声明式事务

### 基本步骤

1. 配置数据源：DataSource
2. 配置事务管理器来控制事务：PlatformTransactionManager
3. @EnableTransactionManagement开启基于注解的事务管理功能
4. 给方法上面标注@Transactional标识当前方法是一个事务方法

### 声明式事务实现原理

1. @EnableTransactionManagement利用TransactionManagementConfigurationSelector给spring容器中导入两个组件：AutoProxyRegistrar和ProxyTransactionManagementConfiguration
2. AutoProxyRegistrar给spring容器中注册一个InfrastructureAdvisorAutoProxyCreator, InfrastructureAdvisorAutoProxyCreator实现了InstantiationAwareBeanPostProcessor, InstantiationAwareBeanPostProcessor是一个BeanPostProcessor。它可以拦截spring的Bean初始化(Initialization)前后和实例化(Initialization)前后。利用后置处理器机制在被拦截的bean创建以后包装该bean并返回一个代理对象代理对象执行方法利用拦截器链进行调用（同springAop的原理）
3. ProxyTransactionManagementConfiguration：是一个spring的配置类,它为spring容器注册了一个BeanFactoryTransactionAttributeSourceAdvisor,是一个事务事务增强器。它有两个重要的字段：AnnotationTransactionAttributeSource和TransactionInterceptor。
  1. AnnotationTransactionAttributeSource：用于解析事务注解的相关信息
  2. TransactionInterceptor：事务拦截器，在事务方法执行时，都会调用TransactionInterceptor的invoke->invokeWithinTransaction方法，这里面通过配置的PlatformTransactionManager控制着事务的提交和回滚。

# Spring 扩展(钩子)

1. BeanFactoryPostProcessor: beanFactory后置处理器, 的拦截时机: 所有Bean的定义信息已经加载到容器, 但还没有被实例化。可以对beanFactory进行一些操作。
2. BeanPostProcessor: bean后置处理器, 拦截时机: bean创建对象初始化前后进行拦截工作。可以对每一个Bean进行一些操作。
3. BeanDefinitionRegistryPostProcessor: 是BeanFactoryPostProcessor的子接口, 拦截时机: 所有Bean的定义信息已经加载到容器, 但还没有被实例化, 可以对每一个Bean的BeanDefinition进行一些操作。
4. ApplicationListener,自定义ApplicationListener实现类并加入到容器中,可以监听spring容器中发布的事件。spring在创建容器的时候 (finishRefresh () 方法) 会发布ContextRefreshedEvent事件, 关闭的时候 (doClose()) 会发布ContextClosedEvent事件。也可以通过spring容器的publishEvent发布自己的事件。
  1. 事件发布流程: publishEvent方法
    1. 获取事件的多播器, getApplicationEventMulticaster()。
    2. 调用multicastEvent(applicationEvent, eventType)派发事件。获取到所有的ApplicationListener,即getApplicationListeners(), 然后同步或者异步的方式执行监听器的onApplicationEvent。
  2. 事件的多播器的初始化中 (initApplicationEventMulticaster () ) , 如果容器中没有配置applicationEventMulticaster, 就使用SimpleApplicationEventMulticaster。然后获取所有的监听器, 并把它们注册到SimpleApplicationEventMulticaster中。
5. @EventListener(class={}): 在普通的业务逻辑的方法上监听事件特定的事件。原理: EventListenerMethodProcessor是一个SmartInitializingSingleton, 当所有的单例bean都初始化完以后, 容器会回调该接口的方法afterSingletonsInstantiated(),该方法里会遍历容器中所有的bean, 并判断每一个bean里是否带有@EventListener注解的Method, 然后创建ApplicationListenerMethodAdapter存储并包装该Method, 最后将ApplicationListenerMethodAdapter添加到spring容器中。

## Spring源代码分析

spring核心逻辑AbstractApplicationContext的refresh()方法如下

```
public void refresh() {
    synchronized (this.startupShutdownMonitor) {
        // 刷新前的预准备工作
        prepareRefresh();
        // 提取bean的配置信息并封装成BeanDefinition实例, 然后将其添加到注册中心。注册中心
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
        //对beanFactory进行一些配置, 注册一些BeanPostProcessor和一些特殊的Bean。
        prepareBeanFactory(beanFactory);
```

```
//留给子类在BeanFactory准备工作完成后处理一些工作。
postProcessBeanFactory(beanFactory);
//调用 BeanFactory的后置处理器。
invokeBeanFactoryPostProcessors(beanFactory);
//注册Bean的后置处理器。
registerBeanPostProcessors(beanFactory);
//国际化相关功能
initMessageSource();
//初始化事件派发器；
initApplicationEventMulticaster();
// 提供给子容器类，供子容器去实例化其他的特殊的Bean
onRefresh();
// 处理容器中已有的ApplicationListener
registerListeners();
//初始化容器中剩余的单实例bean
finishBeanFactoryInitialization(beanFactory);
```

## prepareRefresh()

1. 记录启动时间，设置容器的active和close状态。
2. initPropertySources():提供给子容器类，子容器类可覆盖该方法进行一些自定义的属性设置。
3. getEnvironment().validateRequiredProperties(): 检验属性的合法性
4. this.earlyApplicationEvents = new LinkedHashSet<ApplicationEvent>(): 保存容器

## obtainFreshBeanFactory()

提取bean的配置信息并封装成BeanDefinition实例，然后将其添加到注册中心。注册中心是一个ConcurrentHashMap<String,BeanDefinition>类型，key为Bean的名字，value为BeanDefinition实例。

1. refreshBeanFactory: 如果当前容器已经有了BeanFactory就销毁原来的BeanFactory。然后创建
  - \* 对BeanFactory并进行配置，主要配置是否允许BeanDefinition覆盖，是否允许Bean间的循环引
  - \* 加载BeanDefinition，解析XML文件和配置文件，将其转换为BeanDefinition，然后保存到Def
2. getBeanFactory() 简单的返回beanFactory，即DefaultListableBeanFactory。

## prepareBeanFactory ()

1. 设置BeanFactory的类加载器、设置支持SPEL表达式的解析器。
2. 添加ApplicationContextAwareProcessor用于处理XXXAware接口的回调。
3. 设置忽略一些接口。并注册一些类，这些类可以在bean里直接进行自动装配。
4. 添加ApplicationListenerDetector用于识别并保存ApplicationListener的子类。



## postProcessBeanFactory () :

提供给子容器类，子容器类可以覆盖该方法在BeanFactory准备工作完成后处理一些工作。

## invokeBeanFactoryPostProcessors()

执行BeanFactoryPostProcessor类型的监听方法。

- \* BeanFactoryPostProcessor是beanFactory后置处理器，在整个BeanFactory标准初始化完成后进行。
- \* BeanDefinitionRegistryPostProcessor继承了BeanFactoryPostProcessor，在beanFactory

## segmentfault

[注册登录](#)

- \* 系统默认了一些BeanFactoryPostProcessor。例如：ConfigurationClassPostProcessor
- \* 调用顺序
  1. 先调用BeanDefinitionRegistryPostProcessor类型的拦截器，
  2. 然后再依次调用实现了PriorityOrdered, Ordered接口的BeanFactoryPostProcessor
  3. 最后调用普通的BeanFactoryPostProcessor

## registerBeanPostProcessors()

注册Bean的后置处理器。

1. 从beanFactory里获取所有BeanPostProcessor类型的Bean的名称。
2. 调用beanFactory的getBean方法并传入每一个BeanPostProcessor类型的Bean名称，从容器中获取。
3.
  1. 第一步向beanFactory注册实现了PriorityOrdered的BeanPostProcessor类型的Bean实例。
  2. 第二步向beanFactory注册实现了Ordered的BeanPostProcessor类型的Bean实例。
  3. 第三步向beanFactory注册普通的BeanPostProcessor类型的Bean实例。
  4. 最后一步向beanFactory重新注册实现了MergedBeanDefinitionPostProcessor的BeanPostProcessor
4. 向beanFactory注册BeanPostProcessor的过程就是简单的将实例保存到beanFactory的beanPostProcessors中。

## initMessageSource()

国际化相关功能

1. 看容器中是否有id为messageSource的，类型是MessageSource的Bean实例。如果有赋值给messageSource。
2. 把创建好的MessageSource注册在容器中，以后获取国际化配置文件的值的时候，可以自动注入MessageSource。

## initApplicationEventMulticaster()

初始化事件派发器；

1. 看容中是否有名称为applicationEventMulticaster的，类型是ApplicationEventMulticaster
2. 把创建好的ApplicationEventMulticaster添加到BeanFactory中。

## onRefresh():

提供给子容器类，供子容器去实例化其他的特殊的Bean。

## registerListeners():

处理容器中已有的ApplicationListener。

1. 从容器中获得所有的ApplicationListener
2. 将每个监听器添加到事件派发器（ApplicationEventMulticaster）中；
3. 处理之前步骤产生的事件；

## finishBeanFactoryInitialization():

初始化容器中剩余的单实例bean：拿到剩余的所有的BeanDefinition，依次调用getBean方法（详情看beanFactory.getBean的执行流程）

## finishRefresh():

最后一步。

1. 初始化和生命周期有关的后置处理器；LifecycleProcessor，如果容器中没有指定处理就创建一个DefaultLifecycleProcessor
2. 获取容器中所有的LifecycleProcessor回调onRefresh()方法。
3. 发布容器刷新完成事件ContextRefreshedEvent。

## ConfigurationClassPostProcessor处理@Configuration的过程：



1. 先从主从中心取出所有的BeanDefinition。依次判断，若一个BeanDefinition是被@Configuration标注的，spring将其标记为FullMode，否则若一个BeanDefinition没有被@Configuration标注，但有被@Bean标注的方法，spring将其标记为LightMode。筛选出所有候选配置BeanDefinition（FullMode和LightMode）
2. 创建一个ConfigurationClassParser，调用parse方法解析每一个配置类。
  1. 解析@PropertySources,将解析结果设置到Environment
  2. 利用ComponentScanAnnotationParser，将@ComponentScans标签解析成BeanDefinitionHolder。再迭代解析BeanDefinitionHolder
  3. 解析@Import，@ImportResource
  4. 将@Bean解析为MethodMetadata，将结果保存到ConfigurationClass中。最终ConfigurationClass会被保存到ConfigurationClassParser的configurationClasses中。
3. 调用ConfigurationClassParser的loadBeanDefinitions方法，加载解析结果到注册中。
  1. 从利用ComponentScanAnnotationParser的configurationClasses获取所有的ConfigurationClass，依次调用loadBeanDefinitionsForConfigurationClass方法。
  2. loadBeanDefinitionsForConfigurationClass会将每一个BeanMethod转为ConfigurationClassBeanDefinition，最后将其添加到spring的注册中心。

## beanFactory.getBean方法执行的过程

1. 首先将方法传入的beanName进行转换：先去除FactoryBean前缀（&符）如果传递的beanName是别名，则通过别名找到bean的原始名称。
2. 根据名称先从singletonObjects（一个Map类型的容）获取bean实例。如果能获取到就先判断该bean实例是否实现了FactoryBean，如果是FactoryBean类型的bean实例，就通过FactoryBean获取Bean。然后直接返回该bean实例。getBean方法结束。
3. 如果从singletonObjects没有获取到bean实例就开始创建Bean的过程。
  1. 首先标记该Bean处于创建状态。
  2. 根据Bean的名称找到BeanDefinition。查看该Bean是否有前置依赖的Bean。若有则先创建该Bean前置依赖的Bean。
  3. spring调用AbstractAutowireCapableBeanFactory的createBean方法并传入BeanDefinition开始创建对象。先调用resolveBeforeInstantiation给BeanPostProcessor一个机会去返回一个代理对象去替代目标Bean的实例。
  4. 如果BeanPostProcessor没有返回Bean的代理就通过doCreateBean方法创建对象。
    1. 首先确定Bean的构造函数，如果有有参构造器，先自动装配有参构造器，默认使用无参数构造器。
    2. 选择一个实例化策略去实例化bean。默认使用CglibSubclassingInstantiationStrategy。该策略模式中,首先判断bean是否有方法被覆盖,如果没有则直接通过反射的方式来创建,如果有的话则通过CGLIB来实例化bean对象. 把创建好的bean对象包裹在BeanWrapper里。
    3. 调用MergedBeanDefinitionPostProcessor的postProcessMergedBeanDefinition
    4. 判断容器是否允许循环依赖，如果允许循环依赖，就创建一个ObjectFactory类并实现ObjectFactory接口的唯一的一个方法getObject（）用于返回Bean。然后将该

ObjectFactory添加到singletonFactories中。

5. 调用populateBean为bean实例赋值。在赋值之前执行InstantiationAwareBeanPostProcessor的postProcessAfterInstantiation和postProcessPropertyValues方法。
  6. 调用initializeBean初始化bean。如果Bean实现了XXXAware，就先处理对应的Aware方法。然后调用beanProcessor的postProcessBeforeInitialization方法。再以反射的方式调用指定的bean指定的init方法。最后调用beanProcessor的postProcessAfterInitialization方法。
  7. 调用registerDisposableBeanIfNecessary，将该bean保存在一个以beanName为key，以包装了bean引用的DisposableBeanAdapter，为value的map中，在spring容器关闭时，遍历这个map来获取需要调用bean来依次调用Bean的destroyMethod指定的方法。
5. 将新创建出来的Bean保存到singletonObjects中

## spring原理补充

### spring解决循环依赖

以类A，B互相依赖注入为例

1. 根据类A的名称先从singletonObjects获取Bean实例，发现获取不到，就通过doGetBean方法开始创建Bean的流程。
2. 根据A的名称找到对应的BeanDefinition，通过doCreateBean（）方法创建对象，先确定类A的构造函数，然后选择一个实例化策略去实例化类A。
3. 判断容器是否允许循环依赖，如果允许循环依赖，就创建一个ObjectFactory类并实现ObjectFactory接口的唯一的一个方法getObject（）用于返回类A。然后将该ObjectFactory添加到singletonFactories中。
4. 调用populateBean（）为类A进行属性赋值，发现需要依赖类B，此时类B尚未创建，启动创建类B的流程。
  1. 根据类B的名称先从singletonObjects获取Bean实例，发现获取不到，就开始通过doGetBean方法开始创建Bean的流程
  2. 找到类B对应的BeanDefinition，确认B的构造函数，然后实例化B。
  3. 判断容器是否允许循环依赖，创建一个ObjectFactory并实现getObject（）方法，用于返回类B，并添加到singletonFactories中。
  4. 调用populateBean（）为类B进行属性赋值，发现需要依赖类A，调用getSingleton方法获取A：A现在已存在于singletonFactories中，getSingleton将A从singletonFactories方法中移除并放入earlySingletonObjects中。
  5. 调用getSingleton（）方法获取B：getSingleton将A从singletonFactories方法中移除并放入earlySingletonObjects中。
  6. 调用initializeBean初始化bean，最后将新创建出来的类B保存到singletonObjects中

5. 调用getSingleton () 方法获取A，这时A已在earlySingletonObjects中了，就直接返回A
6. 调用initializeBean初始化bean，最后将新创建出来的类B保存到singletonObjects中。

## @Autowire 实现原理

上面介绍**beanFactory.getBean**方法执行的过程中提到：populateBean为bean实例赋值。在赋值之前执行InstantiationAwareBeanPostProcessor的postProcessAfterInstantiation和postProcessPropertyValues方法。@Autowire由AutowiredAnnotationBeanPostProcessor完成，它实现了InstantiationAwareBeanPostProcessor。

AutowiredAnnotationBeanPostProcessor执行过程：


1. postProcessAfterInstantiation方法执行，直接return null。
2. postProcessPropertyValues方法执行，主要逻辑在此处理。待补充。。。。。

spring  java 源码分析

阅读 13.7k · 更新于 2020-05-20

 赞 30

 收藏 22

 分享

本作品系原创，采用《署名-非商业性使用-禁止演绎 4.0 国际》许可协议



简相杰

234 声望 196 粉丝

关注作者

4 条评论

得票数

最新



撰写评论 ...



提交评论



**chazen996**：讲的不错，配合雷神的视频一起食用点个赞

👍 2 · 回复 · 2020-07-19



**贾培凶超级凶**：有点乱。。。。

👍 · 回复 · 2020-05-21



**简相杰**（作者）：好吧

👍 · 回复 · 2020-05-21



**\_60be68c12aff3**：BeanDefinitionRegistryPostProcessor拦截时机错了，不是所有Bean的定义信息已经加载到容器，而是将要加载到容器

👍 · 回复 · 6 月 11 日

## 你知道吗？

不要基于想象开发，要基于原型开发。

注册登录

继续阅读

### Spring IOC 容器源码分析系列文章导读

Spring 是一个轻量级的企业级应用开发框架，于 2004 年由 Rod Johnson 发布了 1.0 版本。经...

[coolblog](#) · 阅读 6.2k · 15 赞

### Spring 源码导读

做为Java开源世界的第一框架, Spring已经成为事实上的Java EE开发标准Spring框架最根本的使...

[carl\\_zhao](#) · 阅读 1.1k · 6 赞

### Spring源码阅读——ClassPathXmlApplicationContext（一）

ClassPathXmlApplicationContext继承了AbstractXmlApplicationContext，实现了...

[zhzhd](#) · 阅读 3.5k · 1 赞

### spring学习之源码分析--HierarchicalBeanFactory

HierarchicalBeanFactory HierarchicalBeanFactory继承了BeanFactory，扩展了父类容器的方...

大军 · 阅读 1.6k · 1 赞

## spring源码解读

this()注册一个reader和scanner，用于扫描资源和加载资源。register(annotatedClasses)将配...

李沁春 · 阅读 711

## Spring 后置处理器源码

在《几种自定义Spring生命周期的初始化和销毁方法》最后一段描述了启动 Spring 容器过程中...

林晓邬 · 阅读 88

## SpringBoot源码 - bean的加载（上）

这一行代码就是今天的男主角了 - 它完成了bean的加载。它的实现在...

知九千 · 阅读 136

## Spring 源码学习 10：prepareBeanFactory 和 postProcessBeanFactory

根据 refresh 流程，当 obtainFreshBeanFactory 执行结束后，下一步会执行 prepareBeanFacto...

程序员小航 · 阅读 857