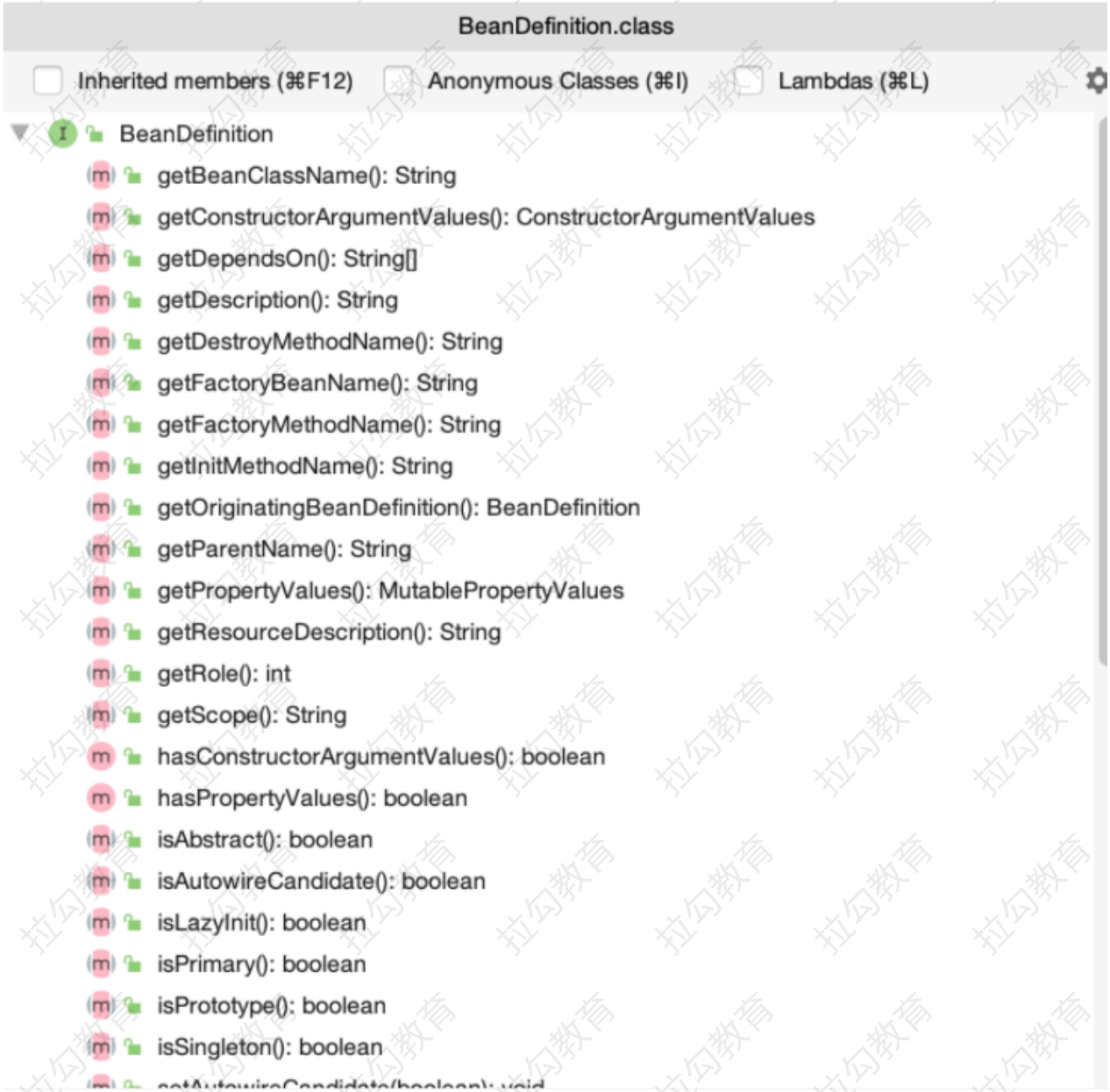


方法名字类似我们bean标签的属性，setBeanClassName对应bean标签中的class属性，所以当我们拿到BeanDefinition对象时，我们可以手动修改bean标签中所定义的属性值。

BeanDefinition对象：我们在 XML 中定义的 bean标签，Spring 解析 bean 标签成为一个 JavaBean，这个JavaBean 就是 BeanDefinition

注意：调用 BeanFactoryPostProcessor 方法时，这时候bean还没有实例化，此时 bean 刚被解析成 BeanDefinition对象

第五部分 Spring IOC源码深度剖析

- 好处：提高培养代码架构思维、深入理解框架
- 原则
  - 定焦原则：抓主线
  - 宏观原则：站在上帝视角，关注源码结构和业务流程（淡化具体某行代码的编写细节）
- 读源码的方法和技巧
  - 断点（观察调用栈）

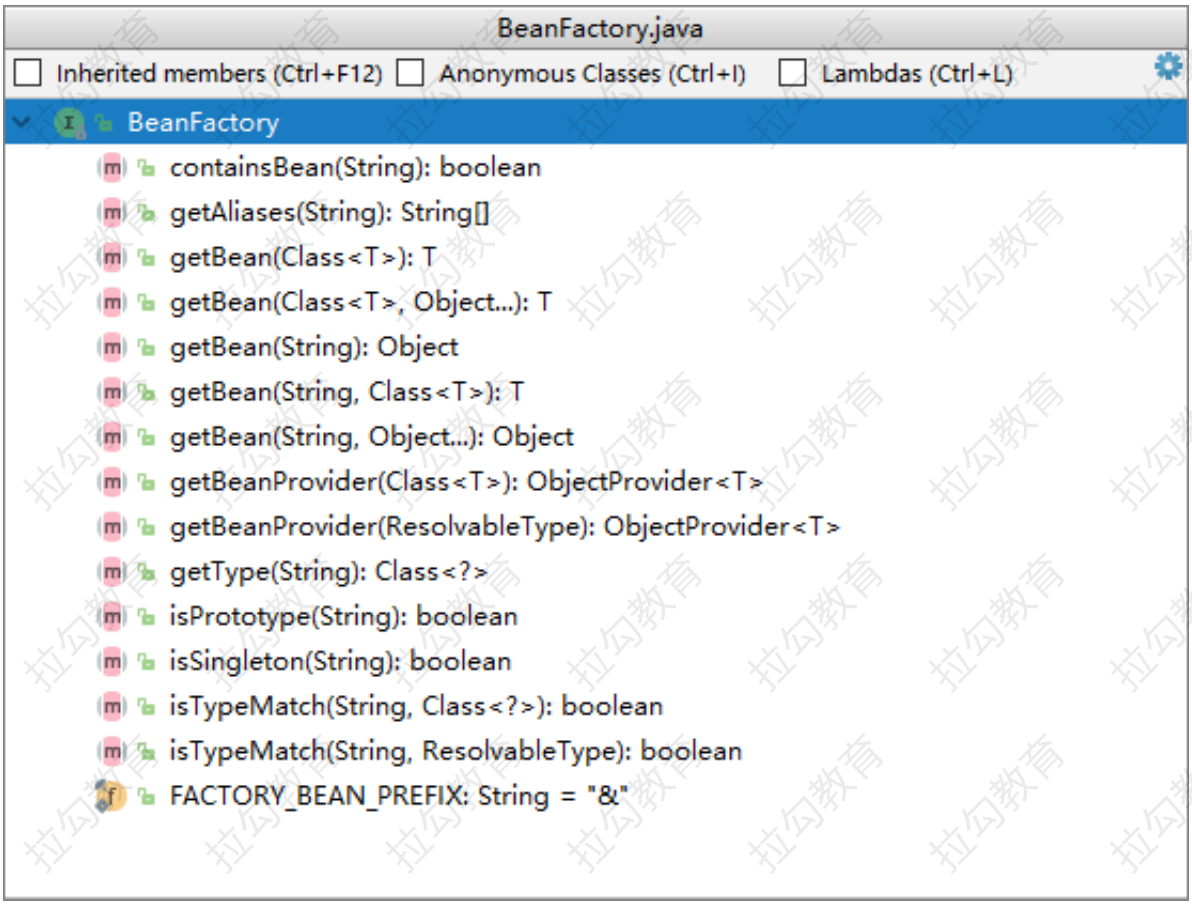
    - 反调（Find Usages）
    - 经验（spring框架中doXXX，做具体处理的地方）
- Spring源码构建
    - 下载源码（github）
    - 安装gradle 5.6.3（类似于maven） Idea 2019.1 Jdk 11.0.5
    - 导入（耗费一定时间）
    - 编译工程（顺序：core-oxm-context-beans-aspects-aop）
        - 工程—>tasks—>compileTestJava

第1节 Spring IoC容器初始化主体流程

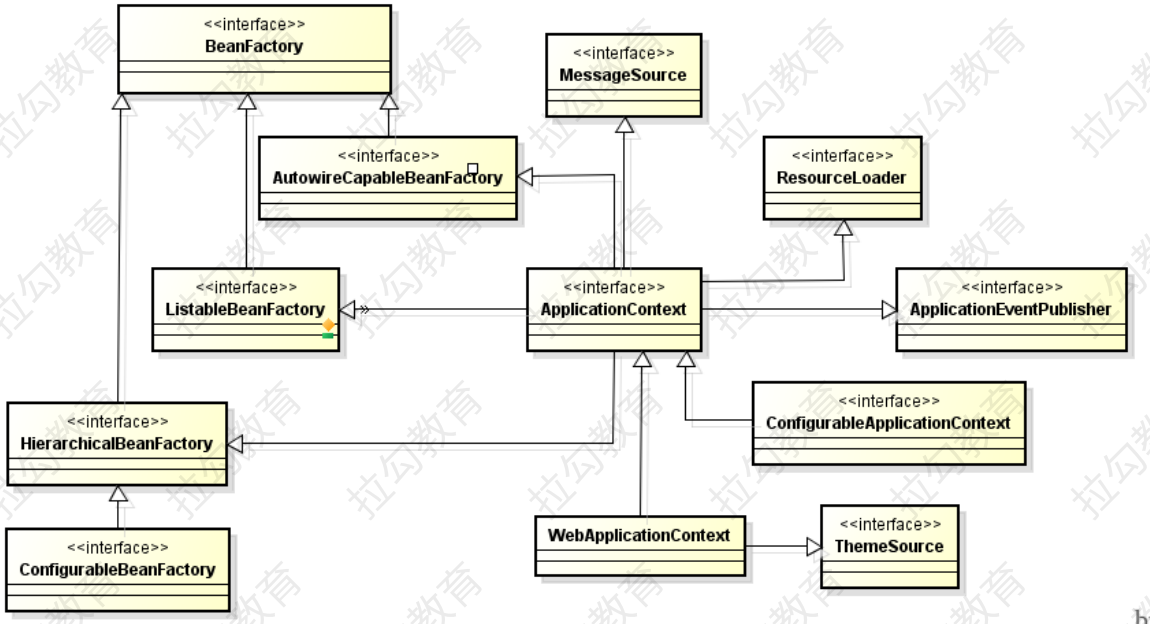
1.1 Spring IoC的容器体系

IoC容器是Spring的核心模块，是抽象了对象管理、依赖关系管理的框架解决方案。Spring 提供了很多的容器，其中 BeanFactory 是顶层容器（根容器），不能被实例化，它定义了所有 IoC 容器 必须遵从的一套原则，具体的容器实现可以增加额外的功能，比如我们常用到的ApplicationContext，其下更具体的实现如 ClassPathXmlApplicationContext 包含了解析 xml 等一系列的内容，AnnotationConfigApplicationContext 则是包含了注解解析等一系列的内容。Spring IoC 容器继承体系非常聪明，需要使用哪个层次用哪个层次即可，不必使用功能大而全的。

BeanFactory 顶级接口方法栈如下



BeanFactory 容器继承体系



by 应癫

通过其接口设计，我们可以看到我们一贯使用的 ApplicationContext 除了继承BeanFactory的子接口，还继承了ResourceLoader、MessageSource等接口，因此其提供的功能也就更丰富了。

下面我们以 ClasspathXmlApplicationContext 为例，深入源码说明 IoC 容器的初始化流程。

1.2 Bean生命周期关键时机点

思路：创建一个类 LagouBean ，让其实现几个特殊的接口，并分别在接口实现的构造器、接口方法中断点，观察线程调用栈，分析出 Bean 对象创建和管理关键点的触发时机。

LagouBean类

```
package com.lagou;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
import org.springframework.beans.factory.config.BeanPostProcessor;
import
org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.stereotype.Component;

/**
 * @Author 应癫
 * @create 2019/12/3 11:46
 */
public class LagouBean implements InitializingBean{

  /**
   * 构造函数
   */
  public LagouBean(){
```

```
    System.out.println("LagouBean 构造器...");
  }

  /**
   * InitializingBean 接口实现
   */
  public void afterPropertiesSet() throws Exception {
    System.out.println("LagouBean afterPropertiesSet...");
  }
}
```

BeanPostProcessor 接口实现类

```
package com.lagou;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.stereotype.Component;

/**
 * @Author 应癫
 * @create 2019/12/3 16:59
 */
public class MyBeanPostProcessor implements BeanPostProcessor {

  public MyBeanPostProcessor() {
    System.out.println("BeanPostProcessor 实现类构造函数...");
  }

  @Override
  public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
    if("lagouBean".equals(beanName)) {
      System.out.println("BeanPostProcessor 实现类
postProcessBeforeInitialization 方法被调用中.....");
    }
    return bean;
  }

  @Override
  public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
    if("lagouBean".equals(beanName)) {
      System.out.println("BeanPostProcessor 实现类
postProcessAfterInitialization 方法被调用中.....");
    }
    return bean;
  }
}
```

BeanFactoryPostProcessor 接口实现类

```
package com.lagou;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
import
org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.stereotype.Component;

/**
 * @Author 应癫
 * @create 2019/12/3 16:56
 */
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {

  public MyBeanFactoryPostProcessor() {
    System.out.println("BeanFactoryPostProcessor的实现类构造函数...");
  }

  @Override
  public void postProcessBeanFactory(ConfigurableListableBeanFactory
beanFactory) throws BeansException {
    System.out.println("BeanFactoryPostProcessor的实现方法调用中......");
  }
}
```

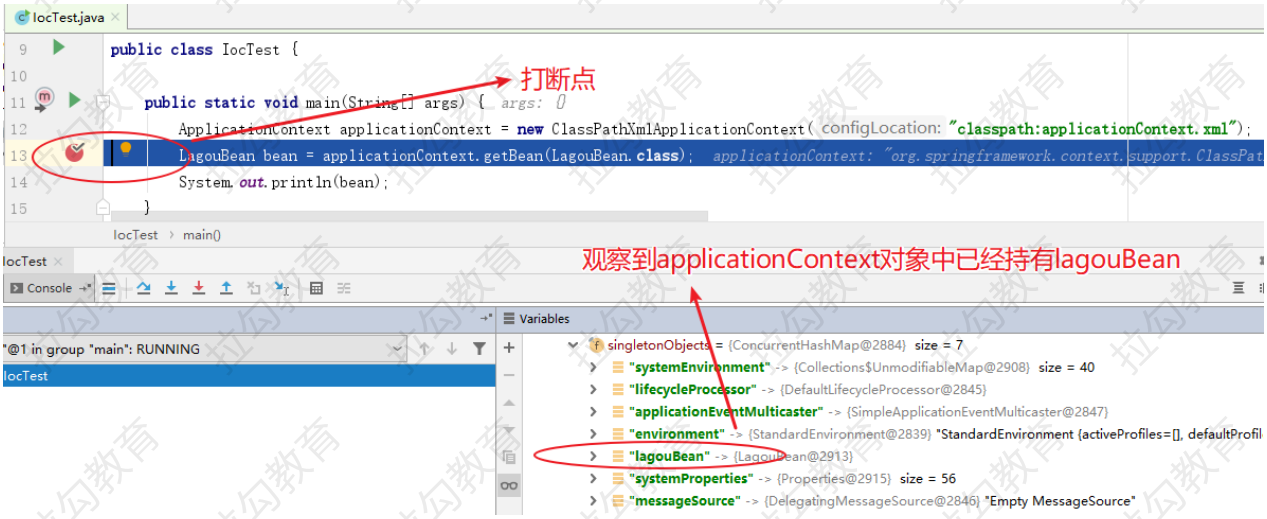
applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="
      http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
">
  <bean id="lagouBean" class="com.lagou.LagouBean"/>
  <bean id="myBeanFactoryPostProcessor"
class="com.lagou.MyBeanFactoryPostProcessor"/>
  <bean id="myBeanPostProcessor" class="com.lagou.MyBeanPostProcessor"/>
</beans>
```

IoC 容器源码分析用例

```
/**
*  Ioc 容器源码分析基础案例
*/
@Test
public void testIoC() {
  ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");
  LagouBean lagouBean = applicationContext.getBean(LagouBean.class);
  System.out.println(lagouBean);
}
```

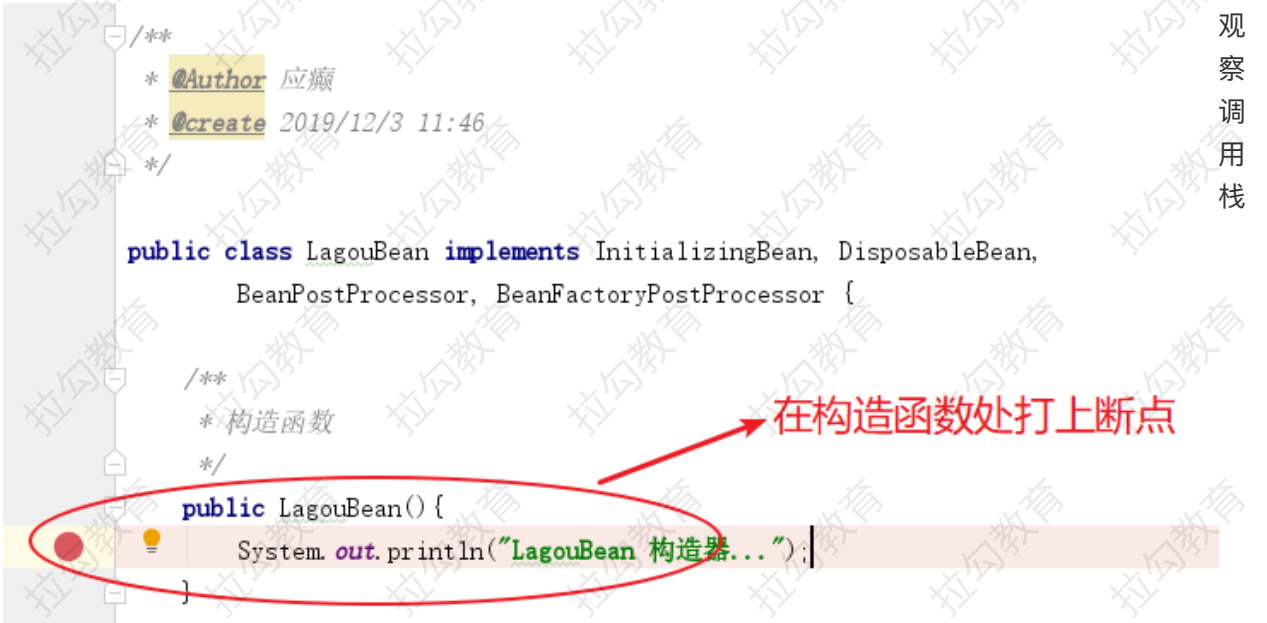
（1）分析 Bean 的创建是在容器初始化时还是在 getBean 时

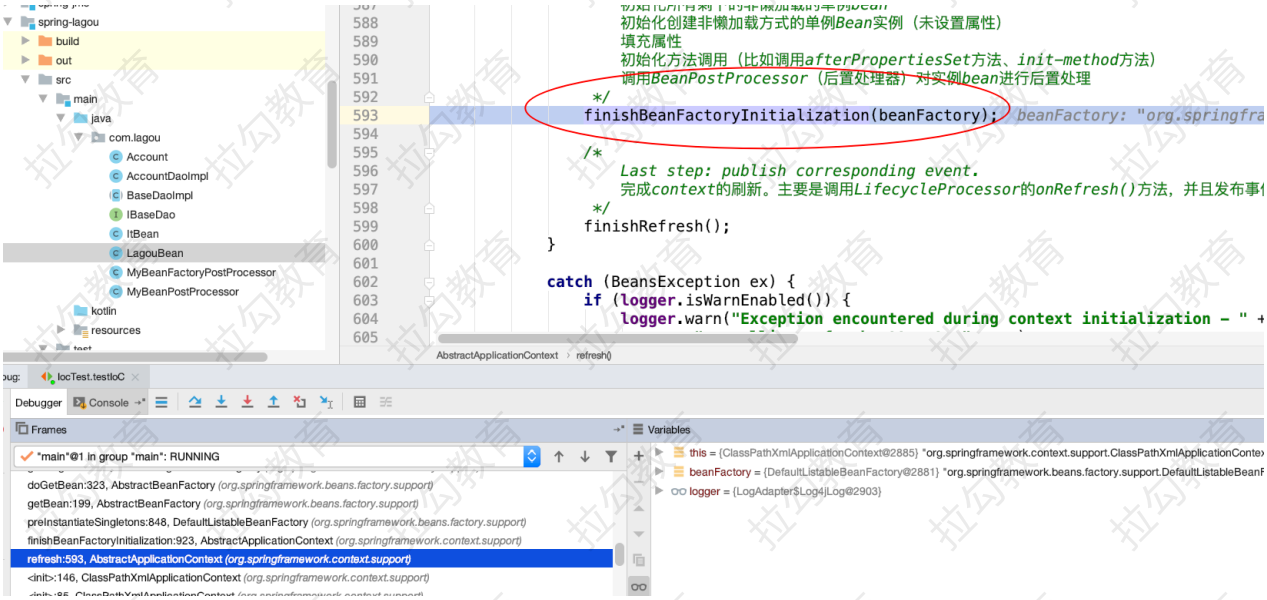


根据断点调试，我们发现，在未设置延迟加载的前提下，Bean 的创建是在容器初始化过程中完成的。

（2）分析构造函数调用情况

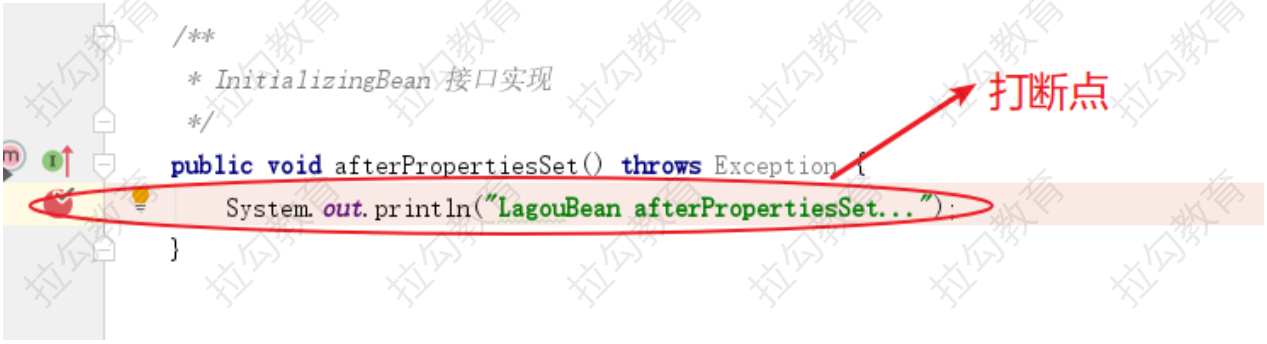
观察调用栈



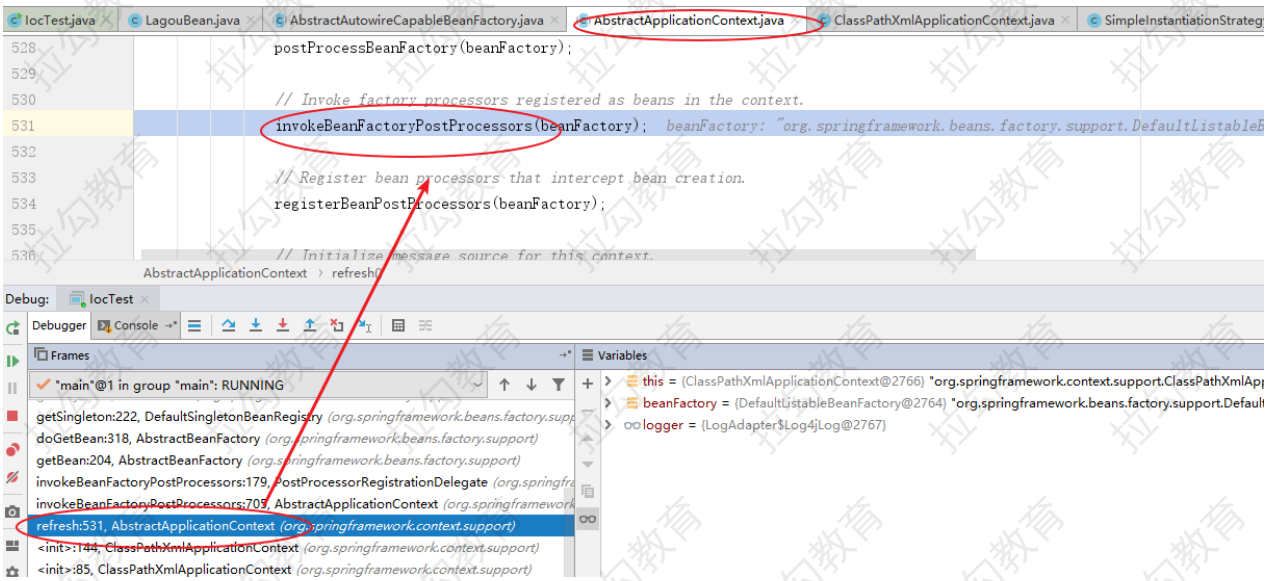


通过如上观察，我们发现构造函数的调用时机在AbstractApplicationContext类refresh方法的finishBeanFactoryInitialization(beanFactory)处;

（3）分析 InitializingBean 之 afterPropertiesSet 初始化方法调用情况



观察调用栈



通过如上观察，我们发现 InitializingBean中afterPropertiesSet 方法的调用时机也是在AbstractApplicationContext类refresh方法的invokeBeanFactoryPostProcessors(beanFactory);

（4）分析BeanFactoryPostProcessor 初始化和调用情况

分别在构造函数、postProcessBeanFactory 方法处打断点，观察调用栈，发现

BeanFactoryPostProcessor 初始化在AbstractApplicationContext类refresh方法的invokeBeanFactoryPostProcessors(beanFactory);

postProcessBeanFactory 调用在AbstractApplicationContext类refresh方法的invokeBeanFactoryPostProcessors(beanFactory);

（5）分析 BeanPostProcessor 初始化和调用情况

分别在构造函数、postProcessBeanFactory 方法处打断点，观察调用栈，发现

BeanPostProcessor 初始化在AbstractApplicationContext类refresh方法的registerBeanPostProcessors(beanFactory);

postProcessBeforeInitialization 调用在AbstractApplicationContext类refresh方法的finishBeanFactoryInitialization(beanFactory);

postProcessAfterInitialization 调用在AbstractApplicationContext类refresh方法的finishBeanFactoryInitialization(beanFactory);

（6）总结

根据上面的调试分析，我们发现 Bean对象创建的几个关键时机点代码层级的调用都在 AbstractApplicationContext 类 的 refresh 方法中，可见这个方法对于Spring IoC 容器初始化来说相当关键，汇总如下：

关键点	触发代码
构造器	refresh#finishBeanFactoryInitialization(beanFactory)(beanFactory)
BeanFactoryPostProcessor 初始化	refresh#invokeBeanFactoryPostProcessors(beanFactory)
BeanFactoryPostProcessor 方法调用	refresh#invokeBeanFactoryPostProcessors(beanFactory)
BeanPostProcessor 初始化	registerBeanPostProcessors(beanFactory)
BeanPostProcessor 方法调用	refresh#finishBeanFactoryInitialization(beanFactory)

对象的实例化、Bean工厂后置处理器初始化以及调用是在refresh#invokeBeanFactoryPostProcessors(beanFactory) 中触发的，但是 Bean 后置处理器初始化是在registerBeanPostProcessors(beanFactory)中完成，调用是在refresh#finishBeanFactoryInitialization(beanFactory) 中触发。

1.3 Spring IoC容器初始化主流程

由上分析可知，Spring IoC 容器初始化的关键环节就在 AbstractApplicationContext#refresh() 方法中，我们查看 refresh 方法来俯瞰容器创建的主体流程，主体流程下的具体子流程我们后面再来讨论。

```
@Override
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // 第一步：刷新前的预处理
        prepareRefresh();

        /*
       第二步：
       获取BeanFactory；默认实现是DefaultListableBeanFactory
```

```
            加载BeanDefition 并注册到 BeanDefitionRegistry
        */
        ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();

        // 第三步：BeanFactory的预准备工作（BeanFactory进行一些设置，比如context的类加
载器等）
        prepareBeanFactory(beanFactory);

        try {
            // 第四步：BeanFactory准备工作完成后进行的后置处理工作
            postProcessBeanFactory(beanFactory);

            // 第五步：实例化并调用实现了BeanFactoryPostProcessor接口的Bean
            invokeBeanFactoryPostProcessors(beanFactory);

            // 第六步：注册BeanPostProcessor（Bean的后置处理器），在创建bean的前后等执
行
            registerBeanPostProcessors(beanFactory);

            // 第七步：初始化MessageSource组件（做国际化功能；消息绑定，消息解析）；
            initMessageSource();

            // 第八步：初始化事件派发器
            initApplicationEventMulticaster();

            // 第九步：子类重写这个方法，在容器刷新的时候可以自定义逻辑
            onRefresh();

            // 第十步：注册应用的监听器。就是注册实现了ApplicationListener接口的监听器
bean
            registerListeners();

            /*
        第十一步：
        初始化所有剩下的非懒加载的单例bean
        初始化创建非懒加载方式的单例Bean实例（未设置属性）
                填充属性
                初始化方法调用（比如调用afterPropertiesSet方法、init-method方法）
                调用BeanPostProcessor（后置处理器）对实例bean进行后置处
      */
            finishBeanFactoryInitialization(beanFactory);

            /*
        第十二步：
        完成context的刷新。主要是调用LifecycleProcessor的onRefresh()方法，并且发布事
件                (ContextRefreshedEvent)
      */
            finishRefresh();
```

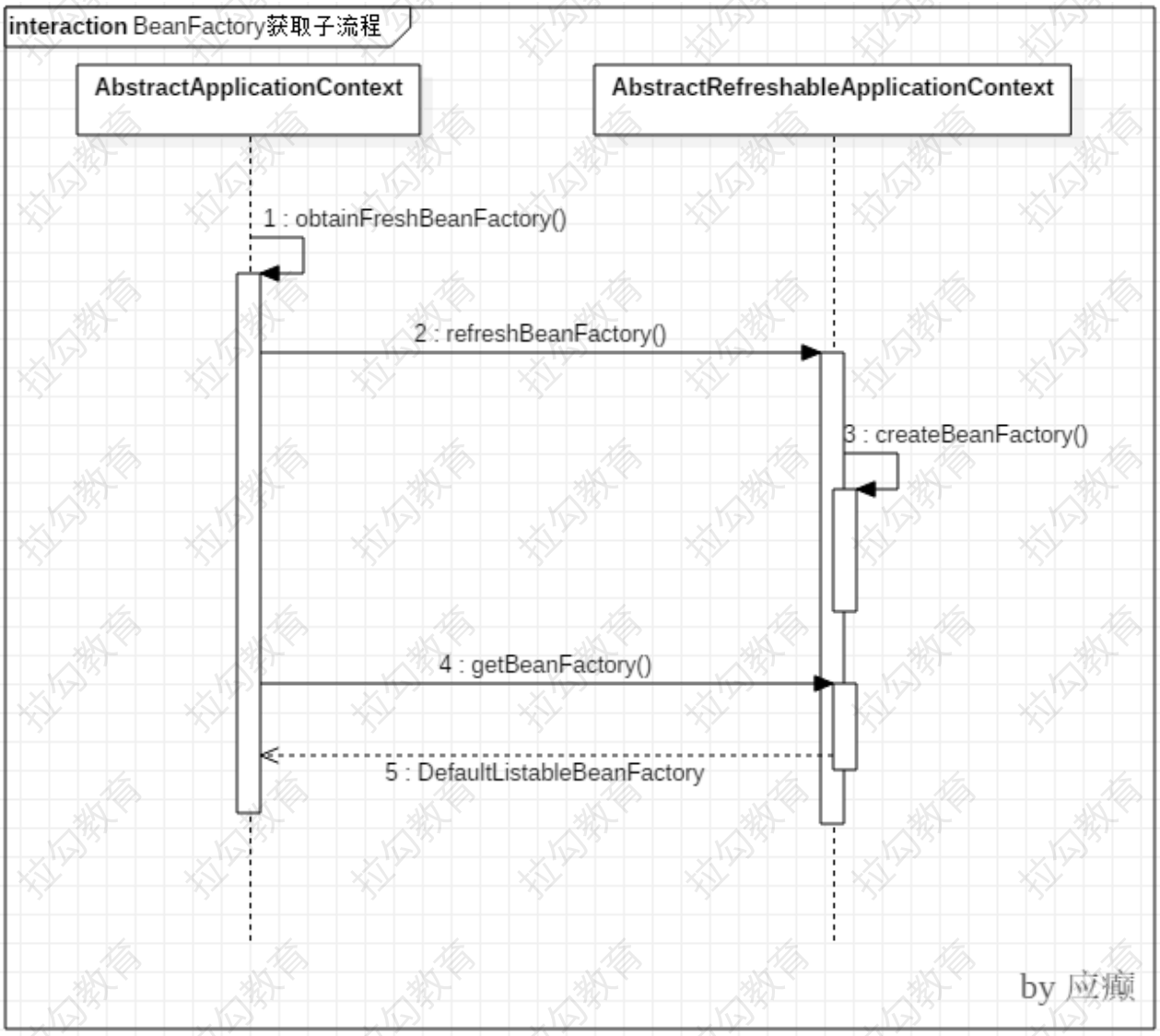
```
        }

        ......
    }
}
```

第2节 BeanFactory创建流程

2.1 获取BeanFactory子流程

时序图如下



2.2 BeanDefinition加载解析及注册子流程

（1）该子流程涉及到如下几个关键步骤

Resource定位：指对BeanDefinition的资源定位过程。通俗讲就是找到定义Javabean信息的XML文件，并将其封装成Resource对象。

BeanDefinition载入：把用户定义好的Javabean表示为IoC容器内部的数据结构，这个容器内部的数据结构就是BeanDefinition。

注册BeanDefinition到 IoC 容器

（2）过程分析

Step 1：子流程入口在 AbstractRefreshableApplicationContext#refreshBeanFactory 方法中

```
@Override
protected final void refreshBeanFactory() throws BeansException {
    // 判断是否已有bean factory
    if (hasBeanFactory()) {
        // 销毁 beans
        destroyBeans();
        // 关闭 bean factory
        closeBeanFactory();
    }
    try {
        // 实例化 DefaultListableBeanFactory
        DefaultListableBeanFactory beanFactory = createBeanFactory();  beanFactory: "o
        // 设置序列化id
        beanFactory.setSerializationId(getId());
        // 自定义bean工厂的一些属性（是否覆盖、是否允许循环依赖）
        customizeBeanFactory(beanFactory);
        // 加载应用中的BeanDefinitions
        loadBeanDefinitions(beanFactory);  beanFactory: "org.springframework.beans.fac
        synchronized (this.beanFactoryMonitor) {
            // 赋值当前bean facotry
            this.beanFactory = beanFactory;
```

入口

Step 2：依次调用多个类的 loadBeanDefinitions 方法 —> AbstractXmlApplicationContext —> AbstractBeanDefinitionReader —> XmlBeanDefinitionReader 一直执行到 XmlBeanDefinitionReader 的 doLoadBeanDefinitions 方法

```
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)  inputSource:
        throws BeanDefinitionStoreException {

    try {
        // 读取xml信息，将xml中信息保存到Document对象
        Document doc = doLoadDocument(inputSource, resource);  doc: "[#document: null]"  inpu
        // 解析document对象，封装BeanDefinition对象并进行注册
        int count = registerBeanDefinitions(doc, resource);  doc: "[#document: null]"  resour
        if (logger.isDebugEnabled()) {
            logger.debug("Loaded " + count + " bean definitions from " + resource);
        }
        return count;
    }
```

Step 3：我们重点观察XmlBeanDefinitionReader 类的 registerBeanDefinitions 方法，期间产生了多次重载调用，我们定位到最后一个

```
public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreException {  doc: "[#d
    BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();  documentReader: DefaultBeanI
    // 获取已有BeanDefinition的数量
    int countBefore = getRegistry().getBeanDefinitionCount();  countBefore: 0
    // 注册BeanDefinition
    documentReader.registerBeanDefinitions(doc, createReaderContext(resource));  documentReader: DefaultBeanDefiniti
    // 返回新注册的BeanDefinition数量
    return getRegistry().getBeanDefinitionCount() - countBefore;
}
```

关注这两个方法

此处我们关注两个地方：一个createReaderContext方法，一个是DefaultBeanDefinitionDocumentReader类的registerBeanDefinitions方法，先进入createReaderContext 方法看看

```
public XmlReaderContext createReaderContext(Resource resource) {
    return new XmlReaderContext(resource, this.problemReporter, this.eventListener,
            this.sourceExtractor, reader: this, getNamespaceHandlerResolver());
}

public NamespaceHandlerResolver getNamespaceHandlerResolver() {
    if (this.namespaceHandlerResolver == null) {
        this.namespaceHandlerResolver = createDefaultNamespaceHandlerResolver();
    }
    return this.namespaceHandlerResolver;
}

protected NamespaceHandlerResolver createDefaultNamespaceHandlerResolver() {
    ClassLoader cl = (getResourceLoader() != null ? getResourceLoader().getClassLoader() : getBeanClassLoader());
    return new DefaultNamespaceHandlerResolver(cl);
}
```

我们可以看到，此处 Spring 首先完成了 NamespaceHandlerResolver 的初始化。

我们再进入 registerBeanDefinitions 方法中追踪，调用了DefaultBeanDefinitionDocumentReader#registerBeanDefinitions 方法

```
@Override
public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {  doc: "[#document: null]
    this.readerContext = readerContext;  readerContext: XmlReaderContext@2900  readerContext: XmlReaderConte
    doRegisterBeanDefinitions(doc.getDocumentElement());  doc: "[#document: null]"
}
```

进入 doRegisterBeanDefinitions 方法

```
                profileSpec, BeanDefinitionParserDelegate.MULTI_VALUE_ATTRIBUTE_DELIMITERS);
            if (!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles)) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Skipped XML bean definition file due to specified profiles [" + profileSpec +  lo
                            "] not matching: " + getReaderContext().getResource());
                }
                return;
            }
        }
    }
    preProcessXml(root);
    parseBeanDefinitions(root, this.delegate);  root: "[beans: null]"  delegate: BeanDefinitionParserDelegate@2865
    postProcessXml(root);

    this.delegate = parent;
}
```

重点是parseBeanDefinitions方法
preProcessXml 和 postProcessXml 都是钩子方法

进入 parseBeanDefinitions 方法

```
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {  root: "[beans: r
    if (delegate.isDefaultNamespace(root)) {
        NodeList nl = root.getChildNodes();  nl: "[beans: null]"  root: "[beans: null]"
        for (int i = 0; i < nl.getLength(); i++) {  i: 1
            Node node = nl.item(i);  node: "[bean: null]"  nl: "[beans: null]"  i: 1
            if (node instanceof Element) {
                Element ele = (Element) node;  ele: "[bean: null]"  node: "[bean: null]"
                if (delegate.is
                    // 解析默认标签元素
                    parseDefaultElement(ele, delegate);  ele: "[bean: null]"  delegate: BeanDefinitionParser
                }
                else {
                    // 解析自定义标签元素
                    delegate.parseCustomElement(ele);
                }
            }
        }
```

Element ele = (Element) node

解析bean元素

进入 parseDefaultElement 方法

```
private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate)
    // import元素处理
    if (delegate.nodeNameEquals(ele, IMPORT_ELEMENT)) {
        importBeanDefinitionResource(ele);
    }
    // alias 元素处理
    else if (delegate.nodeNameEquals(ele, ALIAS_ELEMENT)) {
        processAliasRegistration(ele);
    }
    // bean 元素处理
    else if (delegate.nodeNameEquals(ele, BEAN_ELEMENT)) {
        processBeanDefinition(ele, delegate);  ele: "[bean: null]"  delegate: BeanDef
    }
    // 嵌套 beans 处理
    else if (delegate.nodeNameEquals(ele, NESTED_BEANS_ELEMENT)) {
        // recurse
        doRegisterBeanDefinitions(ele);
    }
```

解析bean元素

进入 processBeanDefinition 方法

```
protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {
    // 解析bean元素为BeanDefinition，但是此时使用 BeanDefinitionHolder 又包装成了 BeanDefinitionHolder 对象
    BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
    if (bdHolder != null) {
        /*
            <bean id="demo" class="com.chenhao.spring.MyTestBean">
                <property name="beanName" value="bean demo1"/>
                <meta key="demo" value="demo"/>
                <mybean:username="mybean"/>
            </bean>
            如果有自定义标签，则处理自定义标签
         */
        bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
        try {
            // 完成BeanDefinition的注册
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext().getRegistry());
        }
```

首先解析成BeanDefinitionHolder对象

完成注册

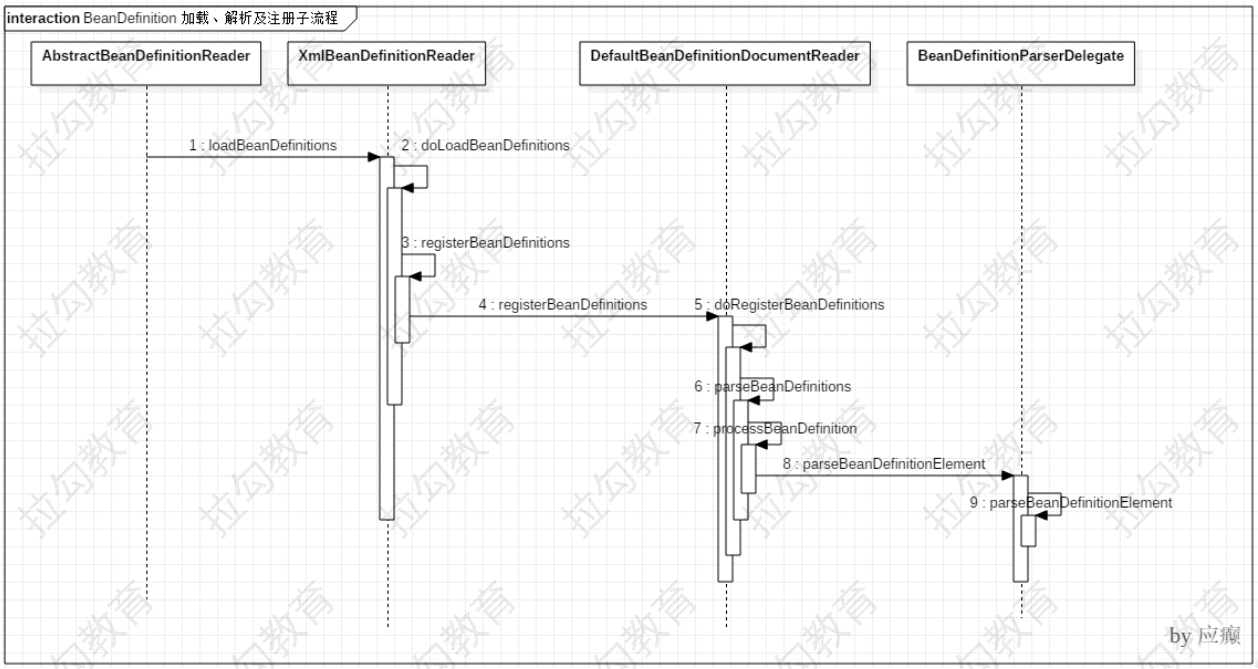
至此，注册流程结束，我们发现，所谓的注册就是把封装的 XML 中定义的 Bean信息封装为 BeanDefinition 对象之后放入一个Map中，BeanFactory 是以 Map 的结构组织这些 BeanDefinition 的。

```
// Still in startup registration phase
this.beanDefinitionMap.put(beanName, beanDefinition);   beanDefinitionMap:
this.beanDefinitionNames.add(beanName);
removeManualSingletonName(beanName);
```

可以在DefaultListableBeanFactory中看到此Map的定义

```
/** Map of bean definition objects, keyed by bean name. */
private final Map<String, BeanDefinition> beanDefinitionMap = new
ConcurrentHashMap<>(256);
```

（3）时序图



第3节 Bean创建流程

- 通过最开始的关键时机点分析，我们知道Bean创建子流程入口在AbstractApplicationContext#refresh()方法的finishBeanFactoryInitialization(beanFactory) 处

```
/*
    Instantiate all remaining (non-lazy-init) singletons.
    初始化所有剩下的非懒加载的单例bean
    初始化创建非懒加载方式的单例Bean实例（未设置属性）
    填充属性
    初始化方法调用（比如调用afterPropertiesSet方法、init-method方法）
    调用BeanPostProcessor（后置处理器）对实例bean进行后置处理
 */
finishBeanFactoryInitialization(beanFactory);  beanFactory: "org.springframework.beans.factory.s
```

bean 创建子流程入口

- 进入finishBeanFactoryInitialization

```
// Stop using the temporary ClassLoader for type matching.
beanFactory.setTempClassLoader(null);

// Allow for caching all bean definition metadata, not expecting further chang
beanFactory.freezeConfiguration();

// Instantiate all remaining (non-lazy-init) singletons.
// 实例化所有立即加载的单例bean
beanFactory.preInstantiateSingletons();
```

- 继续进入DefaultListableBeanFactory类的preInstantiateSingletons方法，我们找到下面部分的代码，看到工厂Bean或者普通Bean，最终都是通过getBean的方法获取实例

```
        if (bean instanceof FactoryBean) {
            final FactoryBean<?> factory = (FactoryBean<?>) bean;
            boolean isEagerInit;
            if (System.getSecurityManager() != null && factory instanceof SmartFactoryBean) {
                isEagerInit = AccessController.doPrivileged((PrivilegedAction<Boolean>)
                                ((SmartFactoryBean<?>) factory)::isEagerInit,
                        getAccessControlContext());
            }
            else {
                isEagerInit = (factory instanceof SmartFactoryBean &&
                        ((SmartFactoryBean<?>) factory).isEagerInit());
            }
            if (isEagerInit) {
                getBean(beanName);
            }
        }
    }
    else {
        // 实例化当前bean
        getBean(beanName);
    }
```

- 继续跟踪下去，我们进入到了AbstractBeanFactory类的doGetBean方法，这个方法中的代码很多，我们直接找到核心部分

```
        // 创建单例bean
        if (mbd.isSingleton()) {
            sharedInstance = getSingleton(beanName, () -> {
                try {
                    // 创建 bean
                    return createBean(beanName, mbd, args);
                }
                catch (BeansException ex) {
                    // Explicitly remove instance from singleton cache: It might have been put there
                    // eagerly by the creation process, to allow for circular reference resolution.
                    // Also remove any beans that received a temporary reference to the bean.
                    destroySingleton(beanName);
                    throw ex;
                }
            });
            bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
        }
```

- 接着进入到AbstractAutowireCapableBeanFactory类的方法，找到以下代码部分

```
try {
    Object beanInstance = doCreateBean(beanName, mbdToUse, args);
    if (logger.isTraceEnabled()) {
        logger.trace("Finished creating instance of bean '" + beanName + "'");
    }
    return beanInstance;
}
```

- 进入doCreateBean方法看看，该方法我们关注两块重点区域
  - 创建Bean实例，此时尚未设置属性

```
if (instanceWrapper == null) {
    // 创建 Bean 实例，但是尚未设置属性
    instanceWrapper = createBeanInstance(beanName, mbd, args);
}
```

  - 给Bean填充属性，调用初始化方法，应用BeanPostProcessor后置处理器

```
// 初始化bean实例
Object exposedObject = bean;
try {
    // Bean属性填充
    populateBean(beanName, mbd, instanceWrapper);
    // 调用初始化方法，应用BeanPostProcessor后置处理器
    exposedObject = initializeBean(beanName, exposedObject, mbd);
}
```

第4节 lazy-init 延迟加载机制原理

- lazy-init 延迟加载机制分析

普通 Bean 的初始化是在容器启动初始化阶段执行的，而被lazy-init=true修饰的 bean 则是在从容器里第一次进行context.getBean() 时进行触发。Spring 启动的时候会把所有bean信息(包括XML和注解)解析转化成Spring能够识别的BeanDefinition并存到Hashmap里供下面的初始化时用，然后对每个 BeanDefinition 进行处理，如果是懒加载的则在容器初始化阶段不处理，其他的则在容器初始化阶段进行初始化并依赖注入。

```
public void preInstantiateSingletons() throws BeansException {
   // 所有beanDefinition集合
   List<String> beanNames = new ArrayList<String>(this.beanDefinitionNames);
   // 触发所有非懒加载单例bean的初始化
   for (String beanName : beanNames) {
       // 获取bean 定义
      RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
      // 判断是否是懒加载单例bean，如果是单例的并且不是懒加载的则在容器创建时初始化
      if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
          // 判断是否是 FactoryBean
         if (isFactoryBean(beanName)) {
                final FactoryBean<?> factory = (FactoryBean<?>)
getBean(FACTORY_BEAN_PREFIX + beanName);
                boolean isEagerInit;
                if (System.getSecurityManager() != null && factory instanceof
SmartFactoryBean) {
                   isEagerInit = AccessController.doPrivileged(new
PrivilegedAction<Boolean>() {
```

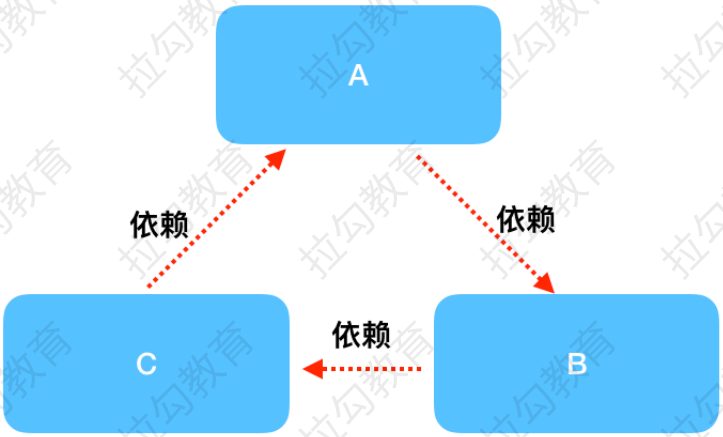
```
                    @Override
                    public Boolean run() {
                        return ((SmartFactoryBean<?>) factory).isEagerInit();
                    }
                }, getAccessControlContext());
            }
        }else {
            /*
             如果是普通bean则进行初始化并依赖注入，此 getBean(beanName)接下来触发的逻辑和
             懒加载时 context.getBean("beanName") 所触发的逻辑是一样的
            */
            getBean(beanName);
        }
    }
  }
}
```

- 总结
  - 对于被修饰为lazy-init的bean Spring 容器初始化阶段不会进行 init 并且依赖注入，当第一次进行getBean时候才进行初始化并依赖注入
  - 对于非懒加载的bean，getBean的时候会从缓存里头获取，因为容器初始化阶段 Bean 已经初始化完成并缓存了起来

第5节 Spring IoC循环依赖问题

5.1 什么是循环依赖

循环依赖其实就是循环引用，也就是两个或者两个以上的 Bean 互相持有对方，最终形成闭环。比如A依赖于B，B依赖于C，C又依赖于A。



注意，这里不是函数的循环调用，是对象的相互依赖关系。循环调用其实就是一个死循环，除非有终结条件。

Spring中循环依赖场景有：

- 构造器的循环依赖（构造器注入）
- Field 属性的循环依赖（set注入）

其中，构造器的循环依赖问题无法解决，只能抛出 BeanCurrentlyInCreationException 异常，在解决属性循环依赖时，spring采用的是提前暴露对象的方法。

5.2 循环依赖处理机制

- 单例 bean 构造器参数循环依赖（无法解决）
- prototype 原型 bean循环依赖（无法解决）

  对于原型bean的初始化过程中不论是通过构造器参数循环依赖还是通过setXxx方法产生循环依赖，Spring都 会直接报错处理。

  AbstractBeanFactory.doGetBean()方法：

```
if (isPrototypeCurrentlyInCreation(beanName)) {
  throw new BeanCurrentlyInCreationException(beanName);
}
```

```
protected boolean isPrototypeCurrentlyInCreation(String beanName) {
    Object curVal = this.prototypesCurrentlyInCreation.get();
    return (curVal != null &&
        (curVal.equals(beanName) || (curVal instanceof Set && ((Set<?>)
curVal).contains(beanName))));
}
```

  在获取bean之前如果这个原型bean正在被创建则直接抛出异常。原型bean在创建之前会进行标记这个beanName正在被创建，等创建结束之后会删除标记

```
try {
    //创建原型bean之前添加标记
    beforePrototypeCreation(beanName);
    //创建原型bean
    prototypeInstance = createBean(beanName, mbd, args);
}
finally {
    //创建原型bean之后删除标记
    afterPrototypeCreation(beanName);
}
```

  总结：Spring 不支持原型 bean 的循环依赖。

- 单例bean通过setXxx或者@Autowired进行循环依赖