Spring IOC 过程源码解析

作者：valarchie

原文链接：https://ld246.com/article/1568056058099

来源网站：链滴

废话不多说，我们先做一个傻瓜版的IOC demo作为例子

自定义的Bean定义

```
class MyBeanDefinition{

    public String id;
    public String className;
    public String value;

    public MyBeanDefinition(String id, String className, String value) {
        this.id = id;
        this.className = className;
        this.value = value;
    }

}
```

自定义的Bean工厂

```
class MyBeanFactory {

    Map<String, Object> beanMap = new HashMap<>();

    public MyBeanFactory(MyBeanDefinition beanDefinition) throws ClassNotFoundException,
            IllegalAccessException, InstantiationException {

        Class<?> beanClass = Class.forName(beanDefinition.className);
        Object bean = beanClass.newInstance();
        ((UserService) bean).setName(beanDefinition.value);
        beanMap.put(beanDefinition.id, bean);

    }

    public Object getBean(String id) {
        return beanMap.get(id);
    }


}
```

测试傻瓜版IOC容器

```
public class EasyIOC {

    public static void main(String[] args) throws IllegalAccessException, InstantiationException,
ClassNotFoundException {

        MyBeanDefinition beanDefinition = new MyBeanDefinition("userService",
                "com.valarchie.UserService", "archie");
```

```
        MyBeanFactory beanFactory = new MyBeanFactory(beanDefinition);
        UserService userService = (UserService) beanFactory.getBean("userService");

        System.out.println(userService.getName());

    }


}
```

看完以上这个傻瓜版的例子我们可以思考一下？让我们自己实现IOC容器的关键是什么呢？

按照我的理解，我总结为以下三步

- 读取xml文件形成DOM对象
- 读取DOM文档对象里的Bean定义并装载进BeanFactory中
- 根据bean定义生成实例放进容器，以供使用

所以，接下来我们不会通盘分析整个IOC的流程，因为旁枝细节太多读者看完也云里雾里抓不到重点。我们通过分析最重要的这条代码主干线来理解IOC的过程。

开始分析：

首先我们从xml的配置方式开始分析，因为Spring最初的配置方式就是利用xml来进行配置，所以大分人对xml的配置形式较为熟悉，也比较方便理解。

从ClassPathXmlApplicationContext的构造器开始讲起。

```
public class TestSpring {
    public static void main(String[] args) {
        // IOC容器的启动就从ClassPathXmlApplicationContext的构造方法开始
        ApplicationContext context = new ClassPathXmlApplicationContext("classpath:applicatio
.xml");
        UserService userService = (UserService) context.getBean("userService");
        System.out.println(userService.getName());

    }
}
```

进入到构造方法中，调用重载的另一个构造方法。

```
// 创建ClassPathXmlApplicationContext，加载给定的位置的xml文件，并自动刷新context
public ClassPathXmlApplicationContext(String configLocation) throws BeansException {
        this(new String[] {configLocation}, true, null);
}
```

重载的构造方法中，由于刚才parrent参数传为null，所以不设置父容器。refresh刚才设置为true，程就会进入refresh()方法中

```
public ClassPathXmlApplicationContext(String[] configLocations, boolean refresh, Application
ontext parent)
```

```
            throws BeansException {
        // 由于之前的方法调用将parent设置为null，所以我们就不分析了
        super(parent);
        // 设置路径数组，并依次对配置路径进行简单占位符替换处理，比较简单，我们也不进入分析了
        setConfigLocations(configLocations);
        if (refresh) {
            refresh();
        }
}
```

整个refresh()方法中就是IOC容器启动的主干脉络了，Spring采用了模板方法设计模式进行refresh()法的设计，先规定好整个IOC容器的具体步骤，然后将每一个小步骤由各种不同的子类自己实现。

所有重要的操作都是围绕着BeanFactory在进行。

在注释当中，我们详细的列出了每一步方法所完成的事情。ApplicationContext内部持有了FactoryBan的实例。其实ApplicationContext本身最上层的父接口也是BeanFactory，他拓展了BeanFactory外的功能（提供国际化的消息访问、资源访问，如URL和文件、事件传播、载入多个（有继承关系）下文）

我们先通过阅读代码中的注释来了解大概的脉络。

```
public void refresh() throws BeansException, IllegalStateException {
        // 先加锁防止启动、结束冲突
        synchronized (this.startupShutdownMonitor) {
            // 在刷新之前做一些准备工作
            // 设置启动的时间、相关状态的标志位（活动、关闭）、初始化占位符属性源，并确认
            // 每个标记为必须的属性都是可解析的。
            prepareRefresh();

            // 获取一个已刷新的BeanFactory实例。
            ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

            // 定义好Bean工厂的环境特性，例如类加载器，或者后置处理器
            prepareBeanFactory(beanFactory);

            try {
                // 设置在BeanFactory完成初始化之后做一些后置操作，spring留给子类的扩展。
                postProcessBeanFactory(beanFactory);

                // 启动之前已设置的BeanFactory后置处理器
                invokeBeanFactoryPostProcessors(beanFactory);

                // 注册Bean处理器
                registerBeanPostProcessors(beanFactory);

                // 为我们的应用上下文设置消息源（i18n）
                initMessageSource();

                // 初始化事件广播器
                initApplicationEventMulticaster();

                // 初始化特殊的Bean在特殊的Context中，默认实现为空，交给各个具体子类实现
                onRefresh();
```

```
            // 检查监听器并注册
            registerListeners();

            // 实例化所有非懒加载的Bean
            finishBeanFactoryInitialization(beanFactory);

            // 最后一步发布相应的事件
            finishRefresh();
        }

        catch (BeansException ex) {
            if (logger.isWarnEnabled()) {
                logger.warn("Exception encountered during context initialization - " +
                        "cancelling refresh attempt: " + ex);
            }

            // 如果启动失败的话，要销毁之前创建的Beans。
            destroyBeans();

            // 重置ApplicationContext内部active的标志位
            cancelRefresh(ex);

            // 向调用者抛出异常
            throw ex;
        }

        finally {
            // 重置Spring核心内的缓存，因为我们可能不再需要单例bean相关的元数据
            resetCommonCaches();
        }
    }
}
```

阅读完之后我们重点关注obtainFreshBeanFactory()、finishBeanFactoryInitialization(beanFactory这两个方法，因为实质上整个IOC的流程都在这两个方法当中，其他的方法一部分是Spring预留给用的自定义操作如BeanFactory的后置处理器和Bean后置处理器，一部分是关键启动事件的发布和监听作，一部分是关于AOP的操作。

首先，先从obtainFreshBeanFactory()开始说起。

第一步：读取xml文件形成DOM对象

在getBeanFactory()方法之前，先调用refreshBeanFactory()方法进行刷新。我们先说明一下，getBenFactory()非常简单，默认实现只是将上一步刷新成功好构建好的Bean工厂进行返回。返回出去的Bn工厂已经加载好Bean定义了。所以在refreshBeanFactory()这个方法中已经包含了第一步读取xml件构建DOM对象和第二步解析DOM中的元素生成Bean定义进行保存。记住，这里仅仅是保存好Bea定义，此时并未涉及Bean的实例化。

```
protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
    refreshBeanFactory();
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    if (logger.isDebugEnabled()) {
        logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
    }
```

```
        return beanFactory;
    }
```

进入refreshBeanFactory()方法中

```
protected final void refreshBeanFactory() throws BeansException {
        // 如果当前ApplicationContext中已存在FactoryBean的话进行销毁
        if (hasBeanFactory()) {
            destroyBeans();
            closeBeanFactory();
        }
        try {
            // 先生成一个BeanFactory
            DefaultListableBeanFactory beanFactory = createBeanFactory();
            // 设置序列化
            beanFactory.setSerializationId(getId());
            // 设置是否可以覆盖Bean定义和是否可以循环依赖，具体我就不解释了
            customizeBeanFactory(beanFactory);
            // 加载Bean定义到Factory当中去
            // 重点!
            loadBeanDefinitions(beanFactory);
            synchronized (this.beanFactoryMonitor) {
                this.beanFactory = beanFactory;
            }
        }
        catch (IOException ex) {
            throw new ApplicationContextException("I/O error parsing bean definition source for "
+ getDisplayName(), ex);
        }
    }
```

接下来，进入核心方法loadBeanDefinitions(beanFactory)中，参数是刚创建的beanFactory

```
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansEx
eption, IOException {
        // 根据传入的beanfactory创建一个xml读取器
        XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanF
ctory);


        // 设置bean定义读取器的相关资源加载环境
        beanDefinitionReader.setEnvironment(this.getEnvironment());
        beanDefinitionReader.setResourceLoader(this);
        beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));

        // 这个方法让子类自定义读取器Reader的初始化
        initBeanDefinitionReader(beanDefinitionReader);
        // 接着开始实际加载Bean定义
        loadBeanDefinitions(beanDefinitionReader);
    }
```

进入loadBeanDefinitions(beanDefinitionReader)方法中，参数是刚刚创建好的Reader读取器。

```
protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException,
IOException {
```

```
        // 如果有已经生成好的Resouce实例的话就直接进行解析。
        // 默认的实现是返回null，由子类自行实现。
        Resource[] configResources = getConfigResources();
        if (configResources != null) {
            reader.loadBeanDefinitions(configResources);
        }
        // 没有Resouces的话就进行路径解析。
        String[] configLocations = getConfigLocations();
        if (configLocations != null) {
            reader.loadBeanDefinitions(configLocations);
        }
}
```

我们进入reader.loadBeanDefinitions(configLocations)方法中，这里面方法调用有点绕，我这边只单地描述一下

该方法会根据多个不同位置的xml文件依次进行处理。

接着会对路径的不同写法进行不同处理，例如classpath或者WEB-INF的前缀路径。

根据传入的locations变量生成对应的Resouces。

紧接着进入reader.loadBeanDefinitions(resource)此时参数是Resource。

在经过一层进入loadBeanDefinitions(new EncodedResource(resource))的方法调用中。

```
public int loadBeanDefinitions(EncodedResource encodedResource) throws BeanDefinitionSto
eException {
    Assert.notNull(encodedResource, "EncodedResource must not be null");
    if (logger.isInfoEnabled()) {
        logger.info("Loading XML bean definitions from " + encodedResource.getResource());
    }
    // 通过ThreadLocal实现的当前currentResource
    Set<EncodedResource> currentResources = this.resourcesCurrentlyBeingLoaded.get();
    if (currentResources == null) {
        currentResources = new HashSet<EncodedResource>(4);
        this.resourcesCurrentlyBeingLoaded.set(currentResources);
    }
    if (!currentResources.add(encodedResource)) {
        throw new BeanDefinitionStoreException(
                "Detected cyclic loading of " + encodedResource + " - check your import definitions
");
    }
    try {

        // 最主要的方法在这段
        InputStream inputStream = encodedResource.getResource().getInputStream();
        try {
            // 传入流对象，并设置好编码
            InputSource inputSource = new InputSource(inputStream);
            if (encodedResource.getEncoding() != null) {
                inputSource.setEncoding(encodedResource.getEncoding());
            }
            return doLoadBeanDefinitions(inputSource, encodedResource.getResource());
        }
```

```
        finally {
            inputStream.close();
        }
    }
    catch (IOException ex) {
        throw new BeanDefinitionStoreException(
                "IOException parsing XML document from " + encodedResource.getResource(), ex);
    }
    finally {
        currentResources.remove(encodedResource);
        if (currentResources.isEmpty()) {
            this.resourcesCurrentlyBeingLoaded.remove();
        }
    }
}
```

该方法最主要是创建了对应的输入流，并设置好编码。

然后开始调用doLoadBeanDefinitions()方法。

```
// 内部核心代码就这两句
Document doc = doLoadDocument(inputSource, resource);
return registerBeanDefinitions(doc, resource);
```

在loadDocument()方法中会生成一个DocumentBuilderImpl对象，这个对象会调用parse方法，在parse方法中使用SAX进行解析刚才的输入流包装的InputSource，生成DOM对象返回。

```
public Document parse(InputSource is) throws SAXException, IOException {
        if (is == null) {
            throw new IllegalArgumentException(
                DOMMessageFormatter.formatMessage(DOMMessageFormatter.DOM_DOMAIN,
                "jaxp-null-input-source", null));
        }
        if (fSchemaValidator != null) {
            if (fSchemaValidationManager != null) {
                fSchemaValidationManager.reset();
                fUnparsedEntityHandler.reset();
            }
            resetSchemaValidator();
        }
        // 解析xml
        domParser.parse(is);
        // 获取刚才解析好的dom
        Document doc = domParser.getDocument();
        domParser.dropDocumentReferences();
        return doc;
}
```

此时我们的xml文件已经加载并解析成DOM结构对象了，第一步已经完成了。

第二步：读取DOM文档对象里的Bean定义并装载进BeanFactory中

```
// 内部核心代码就这两句
Document doc = doLoadDocument(inputSource, resource);
return registerBeanDefinitions(doc, resource);
```

我们再回到刚刚讲到的这两句核心代码，第一句获取DOM对象后，紧接着第二句registerBeanDefinitions(doc, resource)开始了bean定义的注册工作。

进入registerBeanDefinitions(doc, resource)方法中

```
public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreException {
        // 生成DOM读取器，这个和刚才的读取器不一样，之前的读取器是xml读取器。
        BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();
        // 获取之前的bean定义数量
        int countBefore = getRegistry().getBeanDefinitionCount();

        // 进入重点
        documentReader.registerBeanDefinitions(doc, createReaderContext(resource));

        // 用刚刚又创建的bean定义数量 - 之前的bean定义数量 = 刚刚一共创建的bean定义
        return getRegistry().getBeanDefinitionCount() - countBefore;
}
```

进入documentReader.registerBeanDefinitions(doc, createReaderContext(resource))方法。

方法内读取文档的root元素。

```
protected void doRegisterBeanDefinitions(Element root) {
        // Any nested <beans> elements will cause recursion in this method. In
        // order to propagate and preserve <beans> default-* attributes correctly,
        // keep track of the current (parent) delegate, which may be null. Create
        // the new (child) delegate with a reference to the parent for fallback purposes,
        // then ultimately reset this.delegate back to its original (parent) reference.
        // this behavior emulates a stack of delegates without actually necessitating one.
        BeanDefinitionParserDelegate parent = this.delegate;

        // 生成Bean定义解析类
        this.delegate = createDelegate(getReaderContext(), root, parent);

        // 如果是xml文档中的namespace，进行相应处理
        if (this.delegate.isDefaultNamespace(root)) {
            String profileSpec = root.getAttribute(PROFILE_ATTRIBUTE);
            if (StringUtils.hasText(profileSpec)) {
                String[] specifiedProfiles = StringUtils.tokenizeToStringArray(
                        profileSpec, BeanDefinitionParserDelegate.MULTI_VALUE_ATTRIBUTE_DELIMITERS);
                if (!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles)) {
                    if (logger.isInfoEnabled()) {
                        logger.info("Skipped XML bean definition file due to specified profiles [" + profileSpec +
                                "] not matching: " + getReaderContext().getResource());
                    }
```

```
                return;
            }
        }
    }

    // spring预留给子类的拓展性方法
    preProcessXml(root);

    // 重点
    // 开始解析Bean定义
    parseBeanDefinitions(root, this.delegate);

   // spring预留给子类的拓展性方法
    postProcessXml(root);

    this.delegate = parent;

}
```

进入parseBeanDefinitions(root, this.delegate)。将之前的文档对象和bean定义解析类作为参数传。

```
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
    if (delegate.isDefaultNamespace(root)) {
        NodeList nl = root.getChildNodes();
        // 遍历去解析根节点的每个子节点元素
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            // 如果是标签元素的话
            if (node instanceof Element) {
                Element ele = (Element) node;
                if (delegate.isDefaultNamespace(ele)) {

                    // 解析默认的元素
                    // 重点
                    parseDefaultElement(ele, delegate);
                }
                else {
                    // 解析指定自定义元素
                    delegate.parseCustomElement(ele);
                }
            }
        }
    }
    else {
        // 非默认命名空间的，进行自定义解析，命名空间就是xml文档头内的xmlns，用来定义标签

        delegate.parseCustomElement(root);
    }
}
```

进入到parseDefaultElement(ele, delegate)当中，会发现其实对四种标签进行分别的解析。

```
private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {
```

```
        if (delegate.nodeNameEquals(ele, IMPORT_ELEMENT)) {
            importBeanDefinitionResource(ele);
        }
        else if (delegate.nodeNameEquals(ele, ALIAS_ELEMENT)) {
            processAliasRegistration(ele);
        }
        else if (delegate.nodeNameEquals(ele, BEAN_ELEMENT)) {
            // 分析Bean标签
            processBeanDefinition(ele, delegate);
        }
        else if (delegate.nodeNameEquals(ele, NESTED_BEANS_ELEMENT)) {
            // recurse
            doRegisterBeanDefinitions(ele);
        }
    }
```

我们主要分析Bean元素标签的解析，进入processBeanDefinition(ele, delegate)方法中最内层。

```
public BeanDefinitionHolder parseBeanDefinitionElement(Element ele, BeanDefinition contain
ngBean) {
        // 获取bean标签内的id
        String id = ele.getAttribute(ID_ATTRIBUTE);
        // 获取bean标签内的name
        String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);

        // 设置多别名
        List<String> aliases = new ArrayList<String>();
        if (StringUtils.hasLength(nameAttr)) {
            String[] nameArr = StringUtils.tokenizeToStringArray(nameAttr, MULTI_VALUE_ATTRI
UTE_DELIMITERS);
            aliases.addAll(Arrays.asList(nameArr));
        }

        // 当没有设置id的时候
        String beanName = id;
        if (!StringUtils.hasText(beanName) && !aliases.isEmpty()) {
            beanName = aliases.remove(0);
            if (logger.isDebugEnabled()) {
                logger.debug("No XML 'id' specified - using '" + beanName +
                        "' as bean name and " + aliases + " as aliases");
            }
        }

        // 检查beanName是否唯一
        if (containingBean == null) {
            checkNameUniqueness(beanName, aliases, ele);
        }
        // 内部做了Bean标签的解析工作
        AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele, beanName, c
ntainingBean);
        if (beanDefinition != null) {
            if (!StringUtils.hasText(beanName)) {
                try {
                    if (containingBean != null) {
```

```
                beanName = BeanDefinitionReaderUtils.generateBeanName(
                        beanDefinition, this.readerContext.getRegistry(), true);
            }
            else {
                beanName = this.readerContext.generateBeanName(beanDefinition);
                // Register an alias for the plain bean class name, if still possible,
                // if the generator returned the class name plus a suffix.
                // This is expected for Spring 1.2/2.0 backwards compatibility.
                String beanClassName = beanDefinition.getBeanClassName();
                if (beanClassName != null &&
                        beanName.startsWith(beanClassName) && beanName.length() > beanCla
sName.length() &&
                        !this.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {
                    aliases.add(beanClassName);
                }
            }
            if (logger.isDebugEnabled()) {
                logger.debug("Neither XML 'id' nor 'name' specified - " +
                        "using generated bean name [" + beanName + "]");
            }
        }
        catch (Exception ex) {
            error(ex.getMessage(), ele);
            return null;
        }
    }
    String[] aliasesArray = StringUtils.toStringArray(aliases);
    return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);
}

return null;
}
```

将解析好的Bean定义并附加别名数组填入new BeanDefinitionHolder(beanDefinition, beanName, liasesArray)中进行返回。然后调用以下这个方法。

```
BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext().getRegistry())
```

最主要的操作就是将刚才解析好的Bean定义放入beanDefinitionMap中去。

解析成功后将Bean定义进行保存。第二步也已经完成。

第三步：使用创建好的Bean定义，开始实例化Bean。

我们回到最开始的refresh方法中，在finishBeanFactoryInitialization(beanFactory)方法中，开始实化非懒加载的Bean对象。我们跟着调用链进入到preInstantiateSingletons()方法中

```
@Override
public void preInstantiateSingletons() throws BeansException {
    if (this.logger.isDebugEnabled()) {
        this.logger.debug("Pre-instantiating singletons in " + this);
    }
```

```
    // 将之前做好的bean定义名列表拷贝放进beanNames中，然后开始遍历
    List<String> beanNames = new ArrayList<String>(this.beanDefinitionNames);

    // 触发所有非懒加载的单例Bean实例化
    for (String beanName : beanNames) {
        RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);

        // 如果非抽象并且是单例和非懒加载的话
        if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
            // 检测是否是工厂方法Bean。 创建Bean的不同方式，读者可自行百度。
            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 {
                    isEagerInit = (factory instanceof SmartFactoryBean &&
                            ((SmartFactoryBean<?>) factory).isEagerInit());
                }
                if (isEagerInit) {
                    getBean(beanName);
                }
            }
            else {
                getBean(beanName);
            }
        }
    }

    // 关于实例化之后做的自定义操作代码省略....
}
```

在该方法中根据Bean实例是通过工厂方法实例还是普通实例化，最主要的方法还是getBean(beanNae)方法。我们继续分析普通实例化的过程。进入getBean()方法当中doGetBean()方法，发现方法参数oGetBean(name, null, null, false)后三个参数全部为null,它就是整个IOC中的核心代码。

代码中先通过实例化Bean,实例化好之后再判断该Bean所需的依赖，并递归调用进行实例化bean，成后整个IOC的核心流程也就完成了。

```
protected <T> T doGetBean(
        final String name, final Class<T> requiredType, final Object[] args, boolean typeCheckOn
y)
        throws BeansException {

    final String beanName = transformedBeanName(name);
    Object bean;
```

```
    // 从缓存中获取beanName对应的单例
    Object sharedInstance = getSingleton(beanName);
    if (sharedInstance != null && args == null) {
        if (logger.isDebugEnabled()) {
            if (isSingletonCurrentlyInCreation(beanName)) {
                logger.debug("Returning eagerly cached instance of singleton bean '" + beanName
+
                        "' that is not fully initialized yet - a consequence of a circular reference");
            }
            else {
                logger.debug("Returning cached instance of singleton bean '" + beanName + "'");
            }
        }
        // 返回beanName对应的实例对象（主要用于FactoryBean的特殊处理，
        // 普通Bean会直接返回sharedInstance本身）
        bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
    }

    else {
        // scope为prototype的循环依赖校验：如果beanName已经正在创建Bean实
        // 例中，而此时我们又要再一次创建beanName的实例，则代表出现了循环
        // 依赖，需要抛出异常。
            // 例子：如果存在A中有B的属性，B中有A的属性，那么当依赖注入的时候
            //，就会产生当A还未创建完的时候因为对于B的创建再次返回创建A，造成
            //循环依赖
        if (isPrototypeCurrentlyInCreation(beanName)) {
            throw new BeanCurrentlyInCreationException(beanName);
        }

        //如果parentBeanFactory存在，并且beanName在当前BeanFactory不存在
        //Bean定义，则尝试从parentBeanFactory中获取bean实例
        BeanFactory parentBeanFactory = getParentBeanFactory();
        if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
            // 将别名解析成真正的beanName
            String nameToLookup = originalBeanName(name);
            if (args != null) {
                // Delegation to parent with explicit args.
                return (T) parentBeanFactory.getBean(nameToLookup, args);
            }
            else {
                // No args -> delegate to standard getBean method.
                return parentBeanFactory.getBean(nameToLookup, requiredType);
            }
        }

        // 如果不需要类型检查的话 标记为已创建
        if (!typeCheckOnly) {
            markBeanAsCreated(beanName);
        }

        try {
            // 将子Bean定义与父Bean定义进行整合
            final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
            // 整合后如果发现是抽象类不能实例 抛出异常
```

```
            checkMergedBeanDefinition(mbd, beanName, args);

            // 获取Bean定义所需的依赖并逐一初始化填充
            String[] dependsOn = mbd.getDependsOn();
            if (dependsOn != null) {
                for (String dep : dependsOn) {
                    // 判断是否循环依赖
                    if (isDependent(beanName, dep)) {
                        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                                "Circular depends-on relationship between '" + beanName + "' and '" + d
p + "'");
                    }
                    // 注册依赖的Bean
                    registerDependentBean(dep, beanName);
                    try {
                    // 递归调用生成所需依赖的Bean
                        getBean(dep);
                    }
                    catch (NoSuchBeanDefinitionException ex) {
                        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                                "'" + beanName + "' depends on missing bean '" + dep + "'", ex);
                    }
                }
            }

            // 如果是单例的话
            if (mbd.isSingleton()) {
                sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {
                    @Override
                    public Object getObject() throws BeansException {
                        try {
                            return createBean(beanName, mbd, args);
                        }
                        catch (BeansException ex) {
                            // Explicitly remove instance from singleton cache: It might have been put t
ere
                            // 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);
            }

            // 如果是原型的话
            else if (mbd.isPrototype()) {
                // It's a prototype -> create a new instance.
                Object prototypeInstance = null;
                try {
                    beforePrototypeCreation(beanName);
                    prototypeInstance = createBean(beanName, mbd, args);
                }
```

```
                finally {
                    afterPrototypeCreation(beanName);
                }
                bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
            }
                // 非单例和原型 范围的情况 例如session,request等情况
            else {
                String scopeName = mbd.getScope();
                final Scope scope = this.scopes.get(scopeName);
                if (scope == null) {
                    throw new IllegalStateException("No Scope registered for scope name '" + scop
Name + "'");
                }
                try {
                    Object scopedInstance = scope.get(beanName, new ObjectFactory<Object>() {
                        @Override
                        public Object getObject() throws BeansException {
                            beforePrototypeCreation(beanName);
                            try {
                                return createBean(beanName, mbd, args);
                            }
                            finally {
                                afterPrototypeCreation(beanName);
                            }
                        }
                    });
                    bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);
                }
                catch (IllegalStateException ex) {
                    throw new BeanCreationException(beanName,
                            "Scope '" + scopeName + "' is not active for the current thread; consider " +
                            "defining a scoped proxy for this bean if you intend to refer to it from a sing
eton",
                            ex);
                }
            }
        }
        catch (BeansException ex) {
            cleanupAfterBeanCreationFailure(beanName);
            throw ex;
        }
    }

    // 检测实例Bean的类型和所需类型是否一致
    if (requiredType != null && bean != null && !requiredType.isInstance(bean)) {
        try {
            return getTypeConverter().convertIfNecessary(bean, requiredType);
        }
        catch (TypeMismatchException ex) {
            if (logger.isDebugEnabled()) {
                logger.debug("Failed to convert bean '" + name + "' to required type '" +
                        ClassUtils.getQualifiedName(requiredType) + "'", ex);
            }
            throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
```

```
        }
    }
    return (T) bean;
}
```

根据Bean定义去实例化Bean。第三步也已经完成。

文章篇幅有限，IOC整个的创建过程还是比较冗长的，希望读者看完文章对IOC的创建过程有一个主脉络的思路之后还是需要翻开源码进行解读，其实阅读源码并不难，因为Spring的代码注释都挺健全如果遇到不清楚的稍微google一下就知道了。建议读者自己试着一步一步的分析IOC过程的源码。