
回到首页

芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2019-06-01
Spring

【死磕 Spring】—— IoC 之加载 Bean：创建 Bean（一）之主流程

本文主要基于 Spring 5.0.6.RELEASE

摘要：原创出处 http://cmsblogs.com/?p=todo 「小明哥」，谢谢！

作为「小明哥」的忠实读者，「老芳芳」略作修改，记录在理解过程中，参考的资料。

---

1. createBean 抽象方法

在上篇博客 《【死磕 Spring】—— IoC 之加载 Bean：分析各 scope 的 Bean 创建》 中，有一个核心方法没有讲到， `#createBean(String beanName, RootBeanDefinition mbd, Object[] args)` 方法，代码如下：

```
// AbstractBeanFactory.java

protected abstract Object createBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args)
    throws BeanCreationException;
```

该方法定义在 AbstractBeanFactory 中，其含义是根据给定的 BeanDefinition 和 `args` 实例化一个 Bean 对象。
如果该 BeanDefinition 存在父类，则该 BeanDefinition 已经合并了父类的属性。
所有 Bean 实例的创建，都会委托给该方法实现。
该方法接受三个方法参数：

- `beanName` ：bean 的名字。
- `mbd` ：已经合并了父类属性的（如果有的话）BeanDefinition 对象。
- `args` ：用于构造函数或者工厂方法创建 Bean 实例对象的参数。

2. createBean 默认实现

该抽象方法的默认实现是在类 AbstractAutowireCapableBeanFactory 中实现，代码如下：

```
// AbstractAutowireCapableBeanFactory.java

@Override
protected Object createBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args)
        throws BeanCreationException {

    if (logger.isTraceEnabled()) {
        logger.trace("Creating instance of bean '" + beanName + "'");
    }
    RootBeanDefinition mbdToUse = mbd;

    // Make sure bean class is actually resolved at this point, and
    // clone the bean definition in case of a dynamically resolved Class
    // which cannot be stored in the shared merged bean definition.
    // <1> 确保此时的 bean 已经被解析了
    // 如果获取的class 属性不为null，则克隆该 BeanDefinition
    // 主要是因为该动态解析的 class 无法保存到到共享的 BeanDefinition
    Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
    if (resolvedClass != null && !mbd.hasBeanClass() && mbd.getBeanClassName() != null) {
        mbdToUse = new RootBeanDefinition(mbd);
        mbdToUse.setBeanClass(resolvedClass);
    }

    // Prepare method overrides.
    try {
        // <2> 验证和准备覆盖方法
        mbdToUse.prepareMethodOverrides();
    } catch (BeanDefinitionValidationException ex) {
        throw new BeanDefinitionStoreException(mbdToUse.getResourceDescription(),
                beanName, "Validation of method overrides failed", ex);
    }

    try {
        // Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.
        // <3> 实例化的前置处理
        // 给 BeanPostProcessors 一个机会用来返回一个代理类而不是真正的类实例
        // AOP 的功能就是基于这个地方
        Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
        if (bean != null) {
            return bean;
        }
    } catch (Throwable ex) {
        throw new BeanCreationException(mbdToUse.getResourceDescription(), beanName,
                "BeanPostProcessor before instantiation of bean failed", ex);
    }

    try {
        // <4> 创建 Bean 对象
        Object beanInstance = doCreateBean(beanName, mbdToUse, args);
        if (logger.isTraceEnabled()) {
            logger.trace("Finished creating instance of bean '" + beanName + "'");
        }
        return beanInstance;
    } catch (BeanCreationException | ImplicitlyAppearedSingletonException ex) {
        // A previously detected exception with proper bean creation context already,
        // or illegal singleton state to be communicated up to DefaultSingletonBeanRegistry.
        throw ex;
    } catch (Throwable ex) {
        throw new BeanCreationException(
                mbdToUse.getResourceDescription(), beanName, "Unexpected exception during bean creation", ex);
```

```
        }
    }
```

过程如下：

<1> 处，解析指定 BeanDefinition 的 class 属性。
<2> 处，处理 override 属性。
<3> 处，实例化的前置处理。
<4> 处，创建 Bean 对象。

详细解析，每 <x> 处，对应下面的 「2.x」 。

2.1 解析指定 BeanDefinition 的 class

```
// AbstractAutowireCapableBeanFactory.java

Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
if (resolvedClass != null && !mbd.hasBeanClass() && mbd.getBeanClassName() != null) {
    mbdToUse = new RootBeanDefinition(mbd);
    mbdToUse.setBeanClass(resolvedClass);
}
```

#resolveBeanClass(final RootBeanDefinition mbd, String beanName, final Class<?>... typesToMatch) 方法，主要是解析 bean definition 的 class 类，并将已经解析的 Class 存储在 bean definition 中以供后面使用。
如果解析的 class 不为空，则会将该 BeanDefinition 进行设置到 mbdToUse 中。这样做的主要目的是，以为动态解析的 class 是无法保存到共享的 BeanDefinition 中。

详细解析，见 TODO

2.2 处理 override 属性

大家还记得 lookup-method 和 replace-method 这两个配置功能？在博客 《【死磕 Spring】—— IoC 之解析 标签：meta、lookup-method、replace-method》 中，已经详细分析了这两个标签的用法和解析过程，知道解析过程其实就是讲这两个配置存放在 BeanDefinition 中的 methodOverrides 属性中。

我们知道在 bean 实例化的过程中如果检测到存在 methodOverrides ，则会动态地位为当前 bean 生成代理并使用对应的拦截器为 bean 做增强处理。具体的实现我们后续分析，现在先看 mbdToUse.prepareMethodOverrides() 代码块，都干了些什么事，代码如下：

```
// AbstractBeanDefinition.java

public void prepareMethodOverrides() throws BeanDefinitionValidationException {
    // Check that lookup methods exists.
    if (hasMethodOverrides()) {
        Set<MethodOverride> overrides = getMethodOverrides().getOverrides();
        synchronized (overrides) { // 同步
            // 循环，执行 prepareMethodOverride
            for (MethodOverride mo : overrides) {
                prepareMethodOverride(mo);
```

```
            }
        }
    }
}
```

如果存在 methodOverrides ，则获取所有的 override method ，然后通过迭代的方法一次调用 #prepareMethodOverride(MethodOverride mo) 方法。代码如下：

```
// AbstractBeanDefinition.java

protected void prepareMethodOverride(MethodOverride mo) throws BeanDefinitionValidationException {
 int count = ClassUtils.getMethodCountForName(getBeanClass(), mo.getMethodName());
 if (count == 0) {
     throw new BeanDefinitionValidationException(
             "Invalid method override: no method with name '" + mo.getMethodName() +
             "' on class [" + getBeanClassName() + "]");
    } else if (count == 1) {
     // Mark override as not overloaded, to avoid the overhead of arg type checking.
        mo.setOverloaded(false);
    }
}
```

- 根据方法名称，从 class 中获取该方法名的个数：
  - 如果个数为 0 ，则抛出 BeanDefinitionValidationException 异常。
  - 如果个数为 1 ，则设置该重载方法没有被重载。
- 若一个类中存在多个重载方法，则在方法调用的时候还需要根据参数类型来判断到底重载的是哪个方法。在设置重载的时候其实这里做了一个小小优化，那就是当 count == 1 时，设置 overloaded = false ，这样表示该方法没有重载。这样，在后续调用的时候，便可以直接找到方法而不需要进行方法参数的校验。

诚然，其实 mbdToUse.prepareMethodOverrides() 代码块，并没有做什么实质性的工作，只是对 methodOverrides 属性做了一些简单的校验而已。

2.3 实例化的前置处理

#resolveBeforeInstantiation(String beanName, RootBeanDefinition mbd) 方法的作用，是给 BeanPostProcessors 后置处理器返回一个代理对象的机会。其，实在调用该方法之前 Spring 一直都没有创建 bean ，那么这里返回一个 bean 的代理类有什么作用呢？作用体现在后面的 if 判断，代码如下：

```
// AbstractBeanDefinition.java

Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
// ↓ ↓ ↓
if (bean != null) {
 return bean;
}
```

如果代理对象不为空，则直接返回代理对象，这一步骤有非常重要的作用，Spring 后续实现 AOP 就是基于这个地方判断的。

#resolveBeforeInstantiation(String beanName, RootBeanDefinition mbd) 方法，代码如下：

```
// AbstractAutowireCapableBeanFactory.java

@Nullable
protected Object resolveBeforeInstantiation(String beanName, RootBeanDefinition mbd) {
    Object bean = null;
    if (!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {
        // Make sure bean class is actually resolved at this point.
        if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
            Class<?> targetType = determineTargetType(beanName, mbd);
            if (targetType != null) {
                // 前置
                bean = applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);
                if (bean != null) {
                    // 后置
                    bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);
                }
            }
        }
        mbd.beforeInstantiationResolved = (bean != null);
    }
    return bean;
}
```

- 这个方法核心就在于 `applyBeanPostProcessorsBeforeInstantiation()` 和 `applyBeanPostProcessorsAfterInitialization()` 两个方法，before 为实例化前的后处理器应用，after 为实例化后的后处理器应用。
- 由于本文的主题是创建 bean ，关于 Bean 的增强处理后续 LZ 会单独出博文来做详细说明。

详细解析，见 TODO

2.4 创建 Bean

如果没有代理对象，就只能走常规的路线进行 bean 的创建了，该过程有 `#doCreateBean(final String beanName, final RootBeanDefinition mbd, final @Nullable Object[] args)` 方法来实现。代码如下：

```
// AbstractAutowireCapableBeanFactory.java

protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final @Nullable Object[] args)
        throws BeanCreationException {

    // Instantiate the bean.
    // BeanWrapper 是对 Bean 的包装，其接口中所定义的功能很简单包括设置获取被包装的对象，获取被包装 bean 的属性描述器
    BeanWrapper instanceWrapper = null;
    // <1> 单例模型，则从未完成的 FactoryBean 缓存中删除
    if (mbd.isSingleton()) {
        instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
    }
    // <2> 使用合适的实例化策略来创建新的实例：工厂方法、构造函数自动注入、简单初始化
    if (instanceWrapper == null) {
        instanceWrapper = createBeanInstance(beanName, mbd, args);
    }
    // 包装的实例对象
    final Object bean = instanceWrapper.getWrappedInstance();
    // 包装的实例对象的类型
    Class<?> beanType = instanceWrapper.getWrappedClass();
```

```
if (beanType != NullBean.class) {
    mbd.resolvedTargetType = beanType;
}

// Allow post-processors to modify the merged bean definition.
// <3> 判断是否有后置处理
// 如果有后置处理，则允许后置处理修改 BeanDefinition
synchronized (mbd.postProcessingLock) {
    if (!mbd.postProcessed) {
        try {
            // 后置处理修改 BeanDefinition
            applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
        } catch (Throwable ex) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                    "Post-processing of merged bean definition failed", ex);
        }
        mbd.postProcessed = true;
    }
}

// Eagerly cache singletons to be able to resolve circular references
// even when triggered by lifecycle interfaces like BeanFactoryAware.
// <4> 解决单例模式的循环依赖
boolean earlySingletonExposure = (mbd.isSingleton() // 单例模式
        && this.allowCircularReferences // 运行循环依赖
        && isSingletonCurrentlyInCreation(beanName)); // 当前单例 bean 是否正在被创建
if (earlySingletonExposure) {
    if (logger.isTraceEnabled()) {
        logger.trace("Eagerly caching bean '" + beanName +
                "' to allow for resolving potential circular references");
    }
    // 提前将创建的 bean 实例加入到 singletonFactories 中
    // 这里是为了后期避免循环依赖
    addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));
}

// Initialize the bean instance.
// 开始初始化 bean 实例对象
Object exposedObject = bean;
try {
    // <5> 对 bean 进行填充，将各个属性值注入，其中，可能存在依赖于其他 bean 的属性
    // 则会递归初始依赖 bean
    populateBean(beanName, mbd, instanceWrapper);
    // <6> 调用初始化方法
    exposedObject = initializeBean(beanName, exposedObject, mbd);
} catch (Throwable ex) {
    if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {
        throw (BeanCreationException) ex;
    } else {
        throw new BeanCreationException(
                mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);
    }
}

// <7> 循环依赖处理
if (earlySingletonExposure) {
    // 获取 earlySingletonReference
    Object earlySingletonReference = getSingleton(beanName, false);
    // 只有在存在循环依赖的情况下，earlySingletonReference 才不会为空
    if (earlySingletonReference != null) {
```

```
                // 如果 exposedObject 没有在初始化方法中被改变，也就是没有被增强
                if (exposedObject == bean) {
                    exposedObject = earlySingletonReference;
                // 处理依赖
                } else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {
                    String[] dependentBeans = getDependentBeans(beanName);
                    Set<String> actualDependentBeans = new LinkedHashSet<>(dependentBeans.length);
                    for (String dependentBean : dependentBeans) {
                        if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
                            actualDependentBeans.add(dependentBean);
                        }
                    }
                    if (!actualDependentBeans.isEmpty()) {
                        throw new BeanCurrentlyInCreationException(beanName,
                                "Bean with name '" + beanName + "' has been injected into other beans [" +
                                StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +
                                "] in its raw version as part of a circular reference, but has eventually been " +
                                "wrapped. This means that said other beans do not use the final version of the " +
                                "bean. This is often the result of over-eager type matching - consider using " +
                                "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.");
                    }
                }
            }
        }

        // Register bean as disposable.
        // <8> 注册 bean
        try {
            registerDisposableBeanIfNecessary(beanName, bean, mbd);
        } catch (BeanDefinitionValidationException ex) {
            throw new BeanCreationException(
                    mbd.getResourceDescription(), beanName, "Invalid destruction signature", ex);
        }

        return exposedObject;
    }
```

整体的思路：

<1> 处，如果是单例模式，则清除缓存。
- 详细解析，见 TODO

<2> 处，调用 `#createBeanInstance(String beanName, RootBeanDefinition mbd, Object[] args)` 方法，实例化 bean ，主要是将 BeanDefinition 转换为 `org.springframework.beans.BeanWrapper` 对象。
- 详细解析，见 《【死磕 Spring】—— IoC 之加载 Bean：创建 Bean（二）》 和 《【死磕 Spring】—— IoC 之加载 Bean：创建 Bean（三）》 中。

<3> 处，MergedBeanDefinitionPostProcessor 的应用。
- 详细解析，见 TODO

<4> 处，单例模式的循环依赖处理。
- 详细解析，见 《【死磕 Spring】—— IoC 之加载 Bean：创建 Bean（五）之循环依赖处理》 。

<5> 处，调用 `#populateBean(String beanName, RootBeanDefinition mbd, BeanWrapper bw)` 方法，进行属性填充。将所有属性填充至 bean 的实例中。
- 详细解析，见 《【死磕 Spring】—— IoC 之加载 bean：创建 Bean（四）之属性填充》 。

<6> 处，调用 `#initializeBean(final String beanName, final Object bean, RootBeanDefinition mbd)` 方法，初始化 bean 。
- 详细解析，见 《死磕 Spring】—— IoC 之加载 Bean：创建 Bean（六）之初始化

Bean 对象》 。

<7> 处，依赖检查。

- 详细解析，见 TODO

<8> 处，注册 DisposableBean 。

- 详细解析，见 TODO

3. 小结

#doCreateBean(...) 方法，完成 bean 的创建和初始化工作，内容太多，这里就只列出整体思路。下文开始，将该方法进行拆分进行详细讲解，分布从以下几个方面进行阐述：

#createBeanInstance(String beanName, RootBeanDefinition mbd, Object[] args) 方法，实例化 bean 。

循环依赖的处理。

#populateBean(String beanName, RootBeanDefinition mbd, BeanWrapper bw) 方法，进行属性填充。

#initializeBean(final String beanName, final Object bean, RootBeanDefinition mbd) 方法，初始化 Bean 。

文章目录