
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
2019-05-04
Spring

【死磕 Spring】—— IoC 之加载 Bean：从单例缓存中获取单例 Bean

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，「老艿艿」略作修改，记录在理解过程中，参考的资料。

从这篇博客开始我们开始加载 Bean 的第一个步骤，从缓存中获取 Bean 。代码片段如下：

```
// AbstractBeanFactory.java

// 从缓存中或者实例工厂中获取 Bean 对象
// Eagerly check singleton cache for manually registered singletons.
Object sharedInstance = getSingleton(beanName);
if (sharedInstance != null && args == null) {
 if (logger.isTraceEnabled()) {
     if (isSingletonCurrentlyInCreation(beanName)) {
            logger.trace("Returning eagerly cached instance of singleton bean '" + beanName +
                "' that is not fully initialized yet - a consequence of a circular reference");
        } else {
            logger.trace("Returning cached instance of singleton bean '" + beanName + "'");
        }
    }
 // 完成 FactoryBean 的相关处理，并用来获取 FactoryBean 的处理结果
    bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
}
```

调用 `#getSingleton(String beanName)` 方法，从缓存中获取 Bean 。

1. getSingleton

在上篇博客 《【死磕 Spring】—— IoC 之开启 Bean 的加载》 中提到过，Spring 对单例模式的 bean 只会创建一次。后续，如果再获取该 Bean ，则是直接从单例缓存中获取，该过程就体现在 `#getSingleton(String beanName)` 方法中。代码如下：

```
// DefaultSingletonBeanRegistry.java

@Override
@Nullable
public Object getSingleton(String beanName) {
    return getSingleton(beanName, true);
}

@Nullable
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    // 从单例缓冲中加载 bean
    Object singletonObject = this.singletonObjects.get(beanName);
    // 缓存中的 bean 为空，且当前 bean 正在创建
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
        // 加锁
        synchronized (this.singletonObjects) {
            // 从 earlySingletonObjects 获取
            singletonObject = this.earlySingletonObjects.get(beanName);
            // earlySingletonObjects 中没有，且允许提前创建
            if (singletonObject == null && allowEarlyReference) {
                // 从 singletonFactories 中获取对应的 ObjectFactory
                ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
                if (singletonFactory != null) {
                    // 获得 bean
                    singletonObject = singletonFactory.getObject();
                    // 添加 bean 到 earlySingletonObjects 中
                    this.earlySingletonObjects.put(beanName, singletonObject);
                    // 从 singletonFactories 中移除对应的 ObjectFactory
                    this.singletonFactories.remove(beanName);
                }
            }
        }
    }
    return singletonObject;
}
```

这段代码非常简单，保持淡定，过程如下：

第一步，从 singletonObjects 中，获取 Bean 对象。
第二步，若为空且当前 bean 正在创建中，则从 earlySingletonObjects 中获取 Bean 对象。
第三步，若为空且允许提前创建，则从 singletonFactories 中获取相应的 ObjectFactory 对象。若不为空，则调用其 ObjectFactory#getObject(String name) 方法，创建 Bean 对象，然后将其加入到 earlySingletonObjects ，然后从 singletonFactories 删除。

总体逻辑，就是根据 beanName 依次检测这三个 Map，若为空，从下一个，否则返回。这三个 Map 存放的都有各自的功能，代码如下：

```
// DefaultSingletonBeanRegistry.java

/**
 * Cache of singleton objects: bean name to bean instance.
 *
 * 存放的是单例 bean 的映射。
 *
 * 对应关系为 bean name --> bean instance
 */
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);
```

```
/**
 * Cache of singleton factories: bean name to ObjectFactory.
 *
 * 存放的是 ObjectFactory，可以理解为创建单例 bean 的 factory 。
 *
 * 对应关系是 bean name --> ObjectFactory
 **/
private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);

/**
 * Cache of early singleton objects: bean name to bean instance.
 *
 * 存放的是早期的 bean，对应关系也是 bean name --> bean instance。
 *
 * 它与 {@link #singletonFactories} 区别在于 earlySingletonObjects 中存放的 bean 不一定是完整。
 *
 * 从 {@link #getSingleton(String)} 方法中，我们可以了解，bean 在创建过程中就已经加入到 earlySingletonObjects 中了
 * 所以当在 bean 的创建过程中，就可以通过 getBean() 方法获取。
 *
 * 这个 Map 也是【循环依赖】的关键所在。
 */
private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);
```

1.1 isSingletonCurrentlyInCreation

在上面代码中，还有一个非常重要的检测方法 `#isSingletonCurrentlyInCreation(String beanName)` 方法，该方法用于判断该 `beanName` 对应的 Bean 是否在创建过程中，注意这个过程讲的是整个工厂中。代码如下：

```
// DefaultSingletonBeanRegistry.java

/**
 * Names of beans that are currently in creation.
 *
 * 正在创建中的单例 Bean 的名字的集合
 */
private final Set<String> singletonsCurrentlyInCreation =
        Collections.newSetFromMap(new ConcurrentHashMap<>(16));
```

从这段代码中，我们可以预测，在 Bean 创建过程中都会将其加入到 `singletonsCurrentlyInCreation` 集合中。具体是在什么时候加的，我们后面分析。

2. getObjectForBeanInstance

到这里从缓存中获取 bean 的过程已经分析完毕了，我们再看开篇的代码段，从缓存中获取 Bean 后，若其不为 `null` 且 `args` 为空，则会调用 `#getObjectForBeanInstance(Object beanInstance, String name, String beanName,RootBeanDefinition mbd)` 方法，进行处理。

> 为什么会有这么一段呢？因为我们从缓存中获取的 bean 是最原始的 Bean ，并不一定使我们最终想要的 Bean 。
> 怎么办呢？调用 `#getObjectForBeanInstance(...)` 方法，进行处理，该方法的定义为获取给定 Bean 实例的对象，该对象要么是 bean 实例本身，要么就是 FactoryBean 创建的 Bean 对象。

代码如下：

```
// AbstractBeanFactory.java

protected Object getObjectForBeanInstance(
        Object beanInstance, String name, String beanName, @Nullable RootBeanDefinition mbd) {
    // <1> 若为工厂类引用（name 以 & 开头）
    // Don't let calling code try to dereference the factory if the bean isn't a factory.
    if (BeanFactoryUtils.isFactoryDereference(name)) {
        // 如果是 NullBean，则直接返回
        if (beanInstance instanceof NullBean) {
            return beanInstance;
        }
        // 如果 beanInstance 不是 FactoryBean 类型，则抛出异常
        if (!(beanInstance instanceof FactoryBean)) {
            throw new BeanIsNotAFactoryException(transformedBeanName(name), beanInstance.getClass());
        }
    }

    // 到这里我们就有了一个 Bean 实例，当然该实例可能是会是是一个正常的 bean 又或者是一个 FactoryBean
    // 如果是 FactoryBean，我我们则创建该 Bean
    // Now we have the bean instance, which may be a normal bean or a FactoryBean.
    // If it's a FactoryBean, we use it to create a bean instance, unless the
    // caller actually wants a reference to the factory.
    if (!(beanInstance instanceof FactoryBean) || BeanFactoryUtils.isFactoryDereference(name)) {
        return beanInstance;
    }

    Object object = null;
    // <3> 若 BeanDefinition 为 null，则从缓存中加载 Bean 对象
    if (mbd == null) {
        object = getCachedObjectForFactoryBean(beanName);
    }
    // 若 object 依然为空，则可以确认，beanInstance 一定是 FactoryBean 。从而，使用 FactoryBean 获得 Bean 对象
    if (object == null) {
        // Return bean instance from factory.
        FactoryBean<?> factory = (FactoryBean<?>) beanInstance;
        // containsBeanDefinition 检测 beanDefinitionMap 中也就是在所有已经加载的类中
        // 检测是否定义 beanName
        // Caches object obtained from FactoryBean if it is a singleton.
        if (mbd == null && containsBeanDefinition(beanName)) {
            // 将存储 XML 配置文件的 GenericBeanDefinition 转换为 RootBeanDefinition，
            // 如果指定 BeanName 是子 Bean 的话同时会合并父类的相关属性
            mbd = getMergedLocalBeanDefinition(beanName);
        }
        // 是否是用户定义的，而不是应用程序本身定义的
        boolean synthetic = (mbd != null && mbd.isSynthetic());
        // 核心处理方法，使用 FactoryBean 获得 Bean 对象
        object = getObjectFromFactoryBean(factory, beanName, !synthetic);
    }
    return object;
}
```

该方法主要是进行检测工作的，主要如下：

<1> 处，若 name 为工厂相关的（以 & 开头），且 beanInstance 为 NullBean 类型则直接返回，如果 beanInstance 不为 FactoryBean 类型则抛出 BeanIsNotAFactoryException 异常。这里主要是校验 beanInstance 的正确性。
<2> 处，如果 beanInstance 不为 FactoryBean 类型或者 name 也不是与工厂相关的，则直接返

回 beanInstance 这个 Bean 对象。这里主要是对非 FactoryBean 类型处理。

<3> 处，如果 BeanDefinition 为空，则从 factoryBeanObjectCache 中加载 Bean 对象。如果还是空，则可以断定 beanInstance 一定是 FactoryBean 类型，则委托 #getObjectFromFactoryBean(FactoryBean<?> factory, String beanName, boolean shouldPostProcess) 方法，进行处理，使用 FactoryBean 获得 Bean 对象。

> 老艿艿：所以实际上，#getObjectForBeanInstance(...) 方法的重心，就是使用 FactoryBean 对象，获得(或者创建)其 Bean 对象，即调用 #getObjectFromFactoryBean(...) 方法。

2.1 getObjectFromFactoryBean

从上面可以看出， #getObjectForBeanInstance(Object beanInstance, String name, String beanName,RootBeanDefinition mbd) 方法，分成两种情况：

第一种，当该实例对象为非 FactoryBean 类型，直接返回给定的 Bean 实例对象 beanInstance 。
第二种，当该实例对象为FactoryBean 类型，从 FactoryBean (beanInstance) 中，获取 Bean 实例对象。

第二种，通过 #getObjectFromFactoryBean(FactoryBean<?> factory, String beanName, boolean shouldPostProcess) 方法来实现。代码如下：

```
// FactoryBeanRegistrySupport.java

/**
 * Cache of singleton objects created by FactoryBeans: FactoryBean name to object.
 *
 * 缓存 FactoryBean 创建的单例 Bean 对象的映射
 * beanName ===> Bean 对象
 */
private final Map<String, Object> factoryBeanObjectCache = new ConcurrentHashMap<>(16);

protected Object getObjectFromFactoryBean(FactoryBean<?> factory, String beanName, boolean shouldPostProcess) {
    // <1> 为单例模式且缓存中存在
    if (factory.isSingleton() && containsSingleton(beanName)) {
        synchronized (getSingletonMutex()) { // <1.1> 单例锁
            // <1.2> 从缓存中获取指定的 factoryBean
            Object object = this.factoryBeanObjectCache.get(beanName);
            if (object == null) {
                // 为空，则从 FactoryBean 中获取对象
                object = doGetObjectFromFactoryBean(factory, beanName);
                // 从缓存中获取
                // TODO 芋艿，具体原因
                // Only post-process and store if not put there already during getObject() call above
                // (e.g. because of circular reference processing triggered by custom getBean calls)
                Object alreadyThere = this.factoryBeanObjectCache.get(beanName);
                if (alreadyThere != null) {
                    object = alreadyThere;
                } else {
                    // <1.3> 需要后续处理
                    if (shouldPostProcess) {
                        // 若该 Bean 处于创建中，则返回非处理对象，而不是存储它
                        if (isSingletonCurrentlyInCreation(beanName)) {
                            // Temporarily return non-post-processed object, not storing it yet..
```

```
                        return object;
                    }
                    // 单例 Bean 的前置处理
                    beforeSingletonCreation(beanName);
                    try {
                        // 对从 FactoryBean 获取的对象进行后处理
                        // 生成的对象将暴露给 bean 引用
                        object = postProcessObjectFromFactoryBean(object, beanName);
                    } catch (Throwable ex) {
                        throw new BeanCreationException(beanName,
                                "Post-processing of FactoryBean's singleton object failed", ex);
                    } finally {
                        // 单例 Bean 的后置处理
                        afterSingletonCreation(beanName);
                    }
                }
                // <1.4> 添加到 factoryBeanObjectCache 中，进行缓存
                if (containsSingleton(beanName)) {
                    this.factoryBeanObjectCache.put(beanName, object);
                }
            }
        }
        return object;
    }
// <2>
} else {
    // 为空，则从 FactoryBean 中获取对象
    Object object = doGetObjectFromFactoryBean(factory, beanName);
    // 需要后续处理
    if (shouldPostProcess) {
        try {
            // 对从 FactoryBean 获取的对象进行后处理
            // 生成的对象将暴露给 bean 引用
            object = postProcessObjectFromFactoryBean(object, beanName);
        }
        catch (Throwable ex) {
            throw new BeanCreationException(beanName, "Post-processing of FactoryBean's object failed", ex);
        }
    }
    return object;
}
}
```

主要流程如下：

若为单例且单例 Bean 缓存中存在 beanName ，则 <1> 进行后续处理（跳转到下一步），否则，则 <2> 从 FactoryBean 中获取 Bean 实例对象。

<1.1> 首先，获取锁。其实我们在前面篇幅中发现了大量的同步锁，锁住的对象都是 this.singletonObjects，主要是因为在单例模式中必须要保证全局唯一。代码如下：

```
// DefaultSingletonBeanRegistry.java

/**
 * Cache of singleton objects: bean name to bean instance.
 *
 * 存放的是单例 bean 的映射。
```

```
     *
     * 对应关系为 bean name --> bean instance
     */
    private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);

    public final Object getSingletonMutex() {
     return this.singletonObjects;
    }
```

<1.2> 然后，从 factoryBeanObjectCache 缓存中获取实例对象 object 。若 object 为空，则调用 #doGetObjectFromFactoryBean(FactoryBean<?> factory, String beanName) 方法，从 FactoryBean 获取对象，其实内部就是调用 FactoryBean#getObject() 方法。代码如下：

```
    private Object doGetObjectFromFactoryBean(final FactoryBean<?> factory, final String beanName)
        throws BeanCreationException {
        Object object;
        try {
            // 需要权限验证
            if (System.getSecurityManager() != null) {
                AccessControlContext acc = getAccessControlContext();
                try {
                    // <x> 从 FactoryBean 中，获得 Bean 对象
                    object = AccessController.doPrivileged((PrivilegedExceptionAction<Object>) factory::getObject,
                } catch (PrivilegedActionException pae) {
                    throw pae.getException();
                }
            } else {
                // <x> 从 FactoryBean 中，获得 Bean 对象
                object = factory.getObject();
            }
        } catch (FactoryBeanNotInitializedException ex) {
            throw new BeanCurrentlyInCreationException(beanName, ex.toString());
        } catch (Throwable ex) {
            throw new BeanCreationException(beanName, "FactoryBean threw exception on object creation", ex);
        }
        // Do not accept a null value for a FactoryBean that's not fully
        // initialized yet: Many FactoryBeans just return null then.
        if (object == null) {
            if (isSingletonCurrentlyInCreation(beanName)) {
                throw new BeanCurrentlyInCreationException(
                        beanName, "FactoryBean which is currently in creation returned null from getObject");
            }
            object = new NullBean();
        }
        return object;
    }
```

- 在 <x> 处，可以看到，调用 FactoryBean#getObject() 方法，获取 Bean 对象。

<1.3> 如果需要后续处理(shouldPostProcess = true)，则进行进一步处理，步骤如下：

- 若该 Bean 处于创建中（#isSingletonCurrentlyInCreation(String beanName) 方法返回 true ），则返回非处理的 Bean 对象，而不是存储它。
- 调用 #beforeSingletonCreation(String beanName) 方法，进行创建之前的处理。默认实现将该 Bean 标志为当前创建的。
- 调用 #postProcessObjectFromFactoryBean(Object object, String beanName) 方法，对从 FactoryBean

获取的 Bean 实例对象进行后置处理。详细解析，见 「2.3 postProcessObjectFromFactoryBean」 。
    - 调用 #afterSingletonCreation(String beanName) 方法，进行创建 Bean 之后的处理，默认实现是将该 bean 标记为不再在创建中。

<1.4> 最后，加入到 factoryBeanObjectCache 缓存中。

该方法应该就是创建 Bean 实例对象中的核心方法之一了。这里我们关注三个方法：

```
#beforeSingletonCreation(String beanName)
#afterSingletonCreation(String beanName)
#postProcessObjectFromFactoryBean(Object object, String beanName)
```

2.2 isSingletonCurrentlyInCreation

可能有小伙伴觉得前面两个方法不是很重要，LZ 可以肯定告诉你，这两方法是非常重要的操作，因为他们记录着 Bean 的加载状态，是检测当前 Bean 是否处于创建中的关键之处，对解决 Bean 循环依赖起着关键作用。

#beforeSingletonCreation(String beanName) 方法，用于添加标志，当前 bean 正处于创建中
#afterSingletonCreation(String beanName) 方法，用于移除标记，当前 Bean 不处于创建中。

其实在这篇博客刚刚开始就已经提到了， #isSingletonCurrentlyInCreation(String beanName) 方法，是用于检测当前 Bean 是否处于创建之中。代码如下：

```
// DefaultSingletonBeanRegistry.java

/**
 * Names of beans that are currently in creation.
 *
 * 正在创建中的单例 Bean 的名字的集合
 */
private final Set<String> singletonsCurrentlyInCreation =
        Collections.newSetFromMap(new ConcurrentHashMap<>(16));
```

是根据 singletonsCurrentlyInCreation 集合中是否包含了 beanName 。

2.2.1 beforeSingletonCreation

集合的元素，则一定是在 #beforeSingletonCreation(String beanName) 方法中添加的。代码如下：

```
// DefaultSingletonBeanRegistry.java

protected void beforeSingletonCreation(String beanName) {
 if (!this.inCreationCheckExclusions.contains(beanName)
            && !this.singletonsCurrentlyInCreation.add(beanName)) { // 添加
     throw new BeanCurrentlyInCreationException(beanName); // 如果添加失败，则抛出 BeanCurrentlyInCreationException 异
    }
}
```

2.2.2 afterSingletonCreation

#afterSingletonCreation(String beanName) 方法，为移除，则一定就是对 singletonsCurrentlyInCreation 集合

remove 了。代码如下：

```
// DefaultSingletonBeanRegistry.java

protected void afterSingletonCreation(String beanName) {
 if (!this.inCreationCheckExclusions.contains(beanName) &&
            !this.singletonsCurrentlyInCreation.remove(beanName)) { // 移除
        // 如果移除失败，则抛出 IllegalStateException 异常
     throw new IllegalStateException("Singleton '" + beanName + "' isn't currently in creation");
    }
}
```

2.3 postProcessObjectFromFactoryBean

postProcessObjectFromFactoryBean(Object object, String beanName) 方法，对从 FactoryBean 处获取的 Bean 实例对象进行后置处理。其默认实现是直接返回 object 对象，不做任何处理。代码如下：

```
// DefaultSingletonBeanRegistry.java

protected Object postProcessObjectFromFactoryBean(Object object, String beanName) throws BeansException {
 return object;
}
```

2.3.1

当然，子类可以重写，例如应用后处理器。
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory 抽象类，对其提供了实现，代码如下：

```
// AbstractAutowireCapableBeanFactory.java

protected Object postProcessObjectFromFactoryBean(Object object, String beanName) {
 return applyBeanPostProcessorsAfterInitialization(object, beanName);
}
```

该方法的定义为：对所有的 {@code postProcessAfterInitialization} 进行回调注册 BeanPostProcessors ，让他们能够后期处理从 FactoryBean 中获取的对象。下面是具体实现：

```
// AbstractAutowireCapableBeanFactory.java

@Override
public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)
        throws BeansException {
    Object result = existingBean;
    // 遍历 BeanPostProcessor
    for (BeanPostProcessor processor : getBeanPostProcessors()) {
        // 处理
        Object current = processor.postProcessAfterInitialization(result, beanName);
        // 返回空，则返回 result
        if (current == null) {
```

```
                return result;
            }
            // 修改 result
            result = current;
        }
        return result;
    }
```

- 对于后置处理器，这里我们不做过多阐述，后面会专门的博文进行详细介绍，这里我们只需要记住一点：尽可能保证所有 bean 初始化后都会调用注册的 `BeanPostProcessor#postProcessAfterInitialization(Object bean, String beanName)` 方法进行处理，在实际开发过程中大可以针对此特性设计自己的业务逻辑。

3. 小结

至此，从缓存中获取 Bean 对象过程已经分析完毕了。

下面两篇博客分析，如果从单例缓存中没有获取到单例 Bean ，则 Spring 是如何处理的。

文章目录

2014 - 2023 芋道源码 |
总访客数 次 && 总访问量 次
回到首页