【死磕 Spring】—— IOC 之从单例缓存中获取单例 bean (http://cmsblogs.com/?p=2808)

2018-10-18 分类：死磕 Spring (http://cmsblogs.com/?cat=206) 阅读(9516) 评论(4)

> 原文出自：http://cmsblogs.com (http://cmsblogs.com)

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：

```
        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 + "'");
                }
            }
            bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
        }
```

首先调用 `getSingleton()` 从缓存中获取 bean，在上篇博客 【死磕 Spring】----- 加载 bean 之 开启 bean 的加载 (http://cmsblogs.com/?p=2806) 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()` 中。如下：

beanFactory.getBean("name")首先从bean工厂内部的单例hashmap，key=beanName，value=单例对象。中获取

```
public Object getSingleton(String beanName) {
    return getSingleton(beanName, true);
}

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)) { //好像没啥用，运行这个判断直接return了
        // 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);
                // ObjectFactory 不为空，则创建 bean
                if (singletonFactory != null) {
                    singletonObject = singletonFactory.getObject();
                    this.earlySingletonObjects.put(beanName, singletonObject);
                    this.singletonFactories.remove(beanName);
                }
            }
        }
    }
    return singletonObject;
}
```

第一次调用某个beanid,getBean("beanId")时，BeanFactory内部的单例hashmap没有单例对象，get结果为null。下一步需要创建该bean对应的单例实例对象。

三级缓存解决spring的循环依赖问题

这段代码非常简单，首先从 singletonObjects 中获取，若为空且当前 bean 正在创建中，则从 earlySingletonObjects 中获取，若为空且允许提前创建则从 singletonFactories 中获取相应的 ObjectFactory，若不为空，则调用其 `getObject()` 创建 bean，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，如下：

- singletonObjects：存放的是单例 bean，对应关系为 `bean name --> bean instance`
- earlySingletonObjects：存放的是早期的 bean，对应关系也是 `bean name --> bean instance` 。它与 singletonObjects 区别在于 earlySingletonObjects 中存放的 bean 不一定是完整的，从上面过程中我们可以了解，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 也是解决循环依赖的关键所在。
- singletonFactories：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`

在上面代码中还有一个非常重要的检测方法 `isSingletonCurrentlyInCreation(beanName)` ，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。如下：

```
public boolean isSingletonCurrentlyInCreation(String beanName) {
    return this.singletonsCurrentlyInCreation.contains(beanName);
}
```

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。 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()` 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()` 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 创建的对象，如下：

https://www.cnblogs.com/vickylinj/p/9474597.html

```
public class CarFactory {
  //非静态方法
  public Car createCar(){
     Car car = new Car();
     car.setBrand("BMW");
     return car;
  }

  //静态方法
  public static Car createStaticCar(){
     Car car = new Car();
     return car;
  }
}
```

非静态方法·

```
  <!-- 工厂方法-->
  <bean id="carFactory" class="com.baobaotao.ditype.CarFactory" />
  <bean id="car5" factory-bean="carFactory" factory-method="createCar">
  </bean>
```

静态方法

```
 <bean id="car6" class="com.baobaotao.ditype.CarFactory"
      factory-method="createStaticCar"></bean>
```

```
protected Object getObjectForBeanInstance(
        Object beanInstance, String name, String beanName, @Nullable RootBeanDefinition mbd) {

    // 若为工厂类引用（name 以 & 开头）
    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
    if (!(beanInstance instanceof FactoryBean) || BeanFactoryUtils.isFactoryDereference(name)) {
        return beanInstance;
    }

    // 加载 FactoryBean
    Object object = null;
    // 若 BeanDefinition 为 null，则从缓存中加载
    if (mbd == null) {
        object = getCachedObjectForFactoryBean(beanName);
    }
    // 若 object 依然为空，则可以确认，beanInstance 一定是 FactoryBean
    if (object == null) {
        FactoryBean<?> factory = (FactoryBean<?>) beanInstance;
        //
        if (mbd == null && containsBeanDefinition(beanName)) {
            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());
        // 核心处理类
        object = getObjectFromFactoryBean(factory, beanName, !synthetic);
    }
    return object;
}
```

该方法主要是进行检测工作的，主要如下：

- 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。
- 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。这里主要是对非 FactoryBean 类型处理。

- 如果 BeanDefinition 为空，则从 factoryBeanObjectCache 中加载，如果还是空，则可以断定 beanInstance 一定是 FactoryBean 类型，则委托 `getObjectFromFactoryBean()` 方法处理

从上面可以看出 `getObjectForBeanInstance()` 主要是返回给定的 bean 实例对象，当然该实例对象为非 FactoryBean 类型，对于 FactoryBean 类型的 bean，则是委托 `getObjectFromFactoryBean()` 从 FactoryBean 获取 bean 实例对象。

```
protected Object getObjectFromFactoryBean(FactoryBean<?> factory, String beanName, boolean shouldPostProcess) {
        // 为单例模式且缓存中存在
        if (factory.isSingleton() && containsSingleton(beanName)) {

            synchronized (getSingletonMutex()) {
                // 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
                    Object alreadyThere = this.factoryBeanObjectCache.get(beanName);
                    // **我实在是不明白这里这么做的原因，这里是干嘛？？？**
                    if (alreadyThere != null) {
                        object = alreadyThere;
                    }
                    else {
                        // 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)) {
                                return object;
                            }
                            // 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 {
                                // 后置处理
                                afterSingletonCreation(beanName);
                            }
                        }
                        // 缓存
                        if (containsSingleton(beanName)) {
                            this.factoryBeanObjectCache.put(beanName, object);
                        }
                    }
                }
                return object;
            }
        }
```

```
        else {
            // 非单例
            Object object = doGetObjectFromFactoryBean(factory, beanName);
            if (shouldPostProcess) {
                try {
                    object = postProcessObjectFromFactoryBean(object, beanName);
                }
                catch (Throwable ex) {
                    throw new BeanCreationException(beanName, "Post-processing of FactoryBean's object fa
iled", ex);
                }
            }
            return object;
        }
    }
```

主要流程如下：

- 若为单例且单例 bean 缓存中存在 beanName，则进行后续处理（跳转到下一步），否则则从 FactoryBean 中获取 bean 实例对象，如果接受后置处理，则调用 `postProcessObjectFromFactoryBean()` 进行后置处理。
- 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），然后从 factoryBeanObjectCache 缓存中获取实例对象 object，若 object 为空，则调用 `doGetObjectFromFactoryBean()` 方法从 FactoryBean 获取对象，其实内部就是调用 `FactoryBean.getObject()` 。
- 如果需要后续处理，则进行进一步处理，步骤如下：
  - 若该 bean 处于创建中（isSingletonCurrentlyInCreation），则返回非处理对象，而不是存储它
  - 调用 `beforeSingletonCreation()` 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()` 对从 FactoryBean 获取的 bean 实例对象进行后置处理，默认实现是按照原样直接返回，具体实现是在 AbstractAutowireCapableBeanFactory 中实现的，当然子类也可以重写它，比如应用后置处理
  - 调用 `afterSingletonCreation()` 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。
- 最后加入到 FactoryBeans 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()` 、 `afterSingletonCreation()` 、 `postProcessObjectFromFactoryBean()` 。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。**before 方法用于标志当前 bean 处于创建中，after 则是移除。其实在这篇博客刚刚开始就已经提到了 `isSingletonCurrentlyInCreation()` 是用于检测当前 bean 是否处于创建之中，如下：

```
    public boolean isSingletonCurrentlyInCreation(String beanName) {
        return this.singletonsCurrentlyInCreation.contains(beanName);
    }
```

是根据 singletonsCurrentlyInCreation 集合中是否包含了 beanName，集合的元素则一定是在 beforeSingletonCreation() 中添加的，如下：

```
protected void beforeSingletonCreation(String beanName) {
    if (!this.inCreationCheckExclusions.contains(beanName) && !this.singletonsCurrentlyInCreation.add
(beanName)) {
        throw new BeanCurrentlyInCreationException(beanName);
    }
}
```

afterSingletonCreation() 为移除，则一定就是对 singletonsCurrentlyInCreation 集合 remove 了，如下：

```
protected void afterSingletonCreation(String beanName) {
    if (!this.inCreationCheckExclusions.contains(beanName) && !this.singletonsCurrentlyInCreation.rem
ove(beanName)) {
        throw new IllegalStateException("Singleton '" + beanName + "' isn't currently in creation");
    }
}
```

postProcessObjectFromFactoryBean() 是对从 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，子类可以重写，例如应用后处理器。AbstractAutowireCapableBeanFactory 对其提供了实现，如下：

```
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：

```
public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)
        throws BeansException {

    Object result = existingBean;
    for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
        Object current = beanProcessor.postProcessAfterInitialization(result, beanName);
        if (current == null) {
            return 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() 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。 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?

更多阅读

- 【死磕 Spring】----- 加载 bean 之 开启 bean 的加载 (http://cmsblogs.com/?p=2806)
- 【死磕 Spring】----- IOC 之 IOC 初始化总结 (http://cmsblogs.com/?p=2790)

标签： Spring源码解析 (http://cmsblogs.com/?tag=spring%e6%ba%90%e7%a0%81%e8%a7%a3%e6%9e%90)

死磕Java (http://cmsblogs.com/?tag=%e6%ad%bb%e7%a3%95java)

死磕Spring (http://cmsblogs.com/?tag=%e6%ad%bb%e7%a3%95spring)

chenssy (http://cmsblogs.com/?author=1)
不想当厨师的程序员不是好的架构师....

上一篇
【死磕 Spring】—— IOC 之开启 bean 的加载 (http://cmsblogs.com/?p=2806)

下一篇
【死磕 Spring】—— IOC 之parentBeanFactory 与依赖处理 (http://cmsblogs.com/?p=2810)

- 【死磕 Redis】—— 如何排查 Redis 中的慢查询 (http://cmsblogs.com/?p=18352)
- 【死磕 Redis】—— 发布与订阅 (http://cmsblogs.com/?p=18348)
- 【死磕 Redis】—— 布隆过滤器 (http://cmsblogs.com/?p=18346)
- 【死磕 Redis】—— 理解 pipeline 管道 (http://cmsblogs.com/?p=18344)
- 【死磕 Redis】—— 事务 (http://cmsblogs.com/?p=18340)
- 【死磕 Redis】—— Redis 的线程模型 (http://cmsblogs.com/?p=18337)
- 【死磕 Redis】—— Redis 通信协议 RESP (http://cmsblogs.com/?p=18334)
- 【死磕 Redis】—— 开篇 (http://cmsblogs.com/?p=18332)
- 【死磕 Spring】—— IOC 总结 (http://cmsblogs.com/?p=4047)
- 【死磕 Spring】—— 4 张图带你读懂 Spring IOC 的世界 (http://cmsblogs.com/?p=4045)
- 【死磕 Spring】—— 深入分析 ApplicationContext 的 refresh() (http://cmsblogs.com/?p=4043)
- 【死磕 Spring】—— ApplicationContext 相关接口架构分析 (http://cmsblogs.com/?p=4036)
- 【死磕 Spring】—— IOC 之 分析 bean 的生命周期 (http://cmsblogs.com/?p=4034)