【死磕 Spring】—— IOC 之加载 bean：总结 (http://cmsblogs.com/?p=2905)

2018-11-12 分类：死磕 Spring (http://cmsblogs.com/?cat=206) 阅读(9652) 评论(4)

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

在 【死磕 Spring】 Spring bean 解析篇深入分析了一个配置文件经历了哪些过程转变成了 BeanDefinition，但是这个 BeanDefinition 并不是我们真正想要的想要的 bean，因为它还仅仅只是承载了我们需要的目标 bean 的信息，从 BeanDefinition 到我们需要的目标还需要一个漫长的 bean 的初始化阶段，在 【死磕 Spring】 Spring bean 加载阶段已经详细分析了初始化 bean 的过程，所以这里做一个概括性的总结。 bean 的初始化节点由第一次调用 `getBean()` (显式或者隐式)开启，所以我们从这个方法开始。

```
    public Object getBean(String name) throws BeansException {
        return doGetBean(name, null, null, false);
    }

protected <T> T doGetBean(final String name, @Nullable final Class<T> requiredType,
                          @Nullable final Object[] args, boolean typeCheckOnly) throws BeansException
{

        // 获取 beanName，这里是一个转换动作，将 name 转换Wie beanName
        final String beanName = transformedBeanName(name);
        Object bean;

        // 从缓存中或者实例工厂中获取 bean
        // *** 这里会涉及到解决循环依赖 bean 的问题
        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);
        }

        else {

            // 因为 Spring 只解决单例模式下得循环依赖，在原型模式下如果存在循环依赖则会抛出异常
            // **关于循环依赖后续会单独出文详细说明**
            if (isPrototypeCurrentlyInCreation(beanName)) {
                throw new BeanCurrentlyInCreationException(beanName);
            }

            // 如果容器中没有找到，则从父类容器中加载
            BeanFactory parentBeanFactory = getParentBeanFactory();
            if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
                String nameToLookup = originalBeanName(name);
                if (parentBeanFactory instanceof AbstractBeanFactory) {
                    return ((AbstractBeanFactory) parentBeanFactory).doGetBean(
                            nameToLookup, requiredType, args, typeCheckOnly);
                }
                else if (args != null) {
                    return (T) parentBeanFactory.getBean(nameToLookup, args);
                }
                else {
                    return parentBeanFactory.getBean(nameToLookup, requiredType);
                }
```

```
        }

        // 如果不是仅仅做类型检查则是创建bean，这里需要记录
        if (!typeCheckOnly) {
            markBeanAsCreated(beanName);
        }

        try {
            // 从容器中获取 beanName 相应的 GenericBeanDefinition，并将其转换为 RootBeanDefinition
            final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);

            // 检查给定的合并的 BeanDefinition
            checkMergedBeanDefinition(mbd, beanName, args);

            // 处理所依赖的 bean
            String[] dependsOn = mbd.getDependsOn();
            if (dependsOn != null) {
                for (String dep : dependsOn) {
                    // 若给定的依赖 bean 已经注册为依赖给定的b ean
                    // 循环依赖的情况
                    if (isDependent(beanName, dep)) {
                        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                                "Circular depends-on relationship between '" + beanName + "' and '" +
dep + "'");
                    }
                    // 缓存依赖调用
                    registerDependentBean(dep, beanName);
                    try {
                        getBean(dep);
                    }
                    catch (NoSuchBeanDefinitionException ex) {
                        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                                "'" + beanName + "' depends on missing bean '" + dep + "'", ex);
                    }
                }
            }

            // bean 实例化
            // 单例模式
            if (mbd.isSingleton()) {
                sharedInstance = getSingleton(beanName, () -> {
                    try {
                        return createBean(beanName, mbd, args);
                    }
                    catch (BeansException ex) {
                        // 显示从单利缓存中删除 bean 实例
                        // 因为单例模式下为了解决循环依赖，可能他已经存在了，所以销毁它
                        destroySingleton(beanName);
                        throw ex;
                    }
                });
                bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
```

```
                }

                // 原型模式
                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);
                }

                else {
                    // 从指定的 scope 下创建 bean
                    String scopeName = mbd.getScope();
                    final Scope scope = this.scopes.get(scopeName);
                    if (scope == null) {
                        throw new IllegalStateException("No Scope registered for scope name '" + scopeNam
e + "'");
                    }
                    try {
                        Object scopedInstance = scope.get(beanName, () -> {
                            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 singleton",
                                ex);
                    }
                }
            }
            catch (BeansException ex) {
                cleanupAfterBeanCreationFailure(beanName);
                throw ex;
            }
        }
```

```
        // 检查需要的类型是否符合 bean 的实际类型
        if (requiredType != null && !requiredType.isInstance(bean)) {
            try {
                T convertedBean = getTypeConverter().convertIfNecessary(bean, requiredType);
                if (convertedBean == null) {
                    throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
                }
                return convertedBean;
            }
            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;
    }
```

内部调用 `doGetBean()` 方法， `doGetBean()` 的代码量比较多，从这里就可以看出 bean 的加载过程是一个非常复杂的过程，会涉及到各种各样的情况处理。 `doGetBean()` 可以分为以下几个过程。

1. 转换 beanName。因为我们调用 `getBean()` 方法传入的 name 并不一定就是 beanName，可以传入 aliasName，FactoryBean，所以这里需要进行简单的转换过程。
2. 尝试从缓存中加载单例 bean。
3. bean 的实例化。
4. 原型模式的依赖检查。因为 Spring 只会解决单例模式的循环依赖，对于原型模式的循环依赖都是直接抛出 BeanCurrentlyInCreationException 异常。
5. 尝试从 parentBeanFactory 获取 bean 实例。如果 `parentBeanFactory != null && !containsBeanDefinition(beanName)` 则尝试从 parentBeanFactory 中获取 bean 实例对象，因为 `!containsBeanDefinition(beanName)` 就意味着定义的 xml 文件中没有 beanName 相应的配置，这个时候就只能从 parentBeanFactory 中获取。
6. 获取 RootBeanDefinition，并对其进行合并检查。从缓存中获取已经解析的 RootBeanDefinition，同时如果父类不为 null 的话，则会合并父类的属性。
7. 依赖检查。某个 bean 依赖其他 bean ，则需要先加载依赖的 bean。
8. 对不同的 scope 进行处理。
9. 类型转换处理。如果传递的 requiredType 不为 null，则需要检测所得到 bean 的类型是否与该 requiredType 一致，如果不一致则尝试转换，当然也要能够转换成功，否则抛出 BeanNotOfRequiredTypeException 异常。

下面就以下几个方面进行阐述，说明 Spring bean 的加载过程。

1. 从缓存中获取 bean
2. 创建 bean 实例对象
3. 从 bean 实例中获取对象

从缓存中获取 bean

Java技术驿站

Spring 中根据 scope 可以将 bean 分为以下几类：singleton、prototype 和 其他，这样分的原因在于 Spring 在对不同 scope 处理的时候是这么处理的。

- singleton：在 Spring 的 IoC 容器中只存在一个对象实例，所有该对象的引用都共享这个实例。Spring 容器只会创建该 bean 定义的唯一实例，这个实例会被保存到缓存中，并且对该bean的所有后续请求和引用都将返回该缓存中的对象实例。
- prototype：每次对该bean的请求都会创建一个新的实例
- 其他：其他包括 request、session、global session：
  - request：每次 http 请求将会有各自的 bean 实例。
  - session：在一个 http session 中，一个 bean 定义对应一个 bean 实例。
  - global session：在一个全局的 http session 中，一个 bean 定义对应一个 bean 实例。

所以从缓存中获取的 bean 一定是 singleton bean，这也是 Spring 为何只解决 singleton bean 的循环依赖。调用 `getSingleton()` 从缓存中获取 singleton bean。

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

protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    Object singletonObject = this.singletonObjects.get(beanName);
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
        synchronized (this.singletonObjects) {
            singletonObject = this.earlySingletonObjects.get(beanName);
            if (singletonObject == null && allowEarlyReference) {
                ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName); //如果对象实例化时候，提前暴露到了三级缓存，ObjectFactory
                if (singletonFactory != null) {
                    singletonObject = singletonFactory.getObject(); 通过三级缓存，得到提前暴露的实例对象
                    this.earlySingletonObjects.put(beanName, singletonObject);  暴露的半成品对象放到二级缓存
                    this.singletonFactories.remove(beanName); 删除三级缓存的ObjectFactory
                }
            }
        }
    }
    return singletonObject;
}
```

`getSingleton()` 就是从 singletonObjects、earlySingletonObjects、 singletonFactories 三个缓存中获取，这里也是 Spring 解决 bean 循环依赖的关键之处。详细内容请查看如下内容：

- 【死磕 Spring】----- IOC 之加载 bean：从单例缓存中获取单例 bean ()
- 【死磕 Spring】----- IOC 之加载 bean：创建 bean（五） ()

创建 bean 实例对象

如果缓存中没有，也没有 parentBeanFactory ，则会调用 `createBean()` 创建 bean 实例，该方法主要是在处理不同 scope 的 bean 的时候进行调用。

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

该方法是定义在 AbstractBeanFactory 中的虚拟方法，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。 方法接受三个参数：

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

该抽象方法的默认实现是在类 AbstractAutowireCapableBeanFactory 中实现，该方法其实只是做一些检查和验证工作，真正的初始化工作是由 `doCreateBean()` 实现，如下：

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

        // 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;
        // 单例模型，则从未完成的 FactoryBean 缓存中删除
        if (mbd.isSingleton()) {anceWrapper = this.factoryBeanInstanceCache.remove(beanName);
        }

        // 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 甚至cglib
        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;
        }

        // 检测是否有后置处理
        // 如果有后置处理，则允许后置处理修改 BeanDefinition
        synchronized (mbd.postProcessingLock) {
            if (!mbd.postProcessed) {
                try {
                    // applyMergedBeanDefinitionPostProcessors
                    // 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;
            }
        }

        // 解决单例模式的循环依赖
        // 单例模式 & 运行循环依赖&当前单例 bean 是否正在被创建
        boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&
                isSingletonCurrentlyInCreation(beanName));
        if (earlySingletonExposure) {
            if (logger.isDebugEnabled()) {
                logger.debug("Eagerly caching bean '" + beanName +
                        "' to allow for resolving potential circular references");
            }
            // 提前将创建的 bean 实例加入到ectFactory 中
```

```
            // 这里是为了后期避免循环依赖。实例化阶段后就把对象暴露到三级缓存中ObjectFactory
            addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));
        }

        /*
         * 开始初始化 bean 实例对象
         */
        Object exposedObject = bean;
        try {
            // 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); 填充实例化对象的属性
            // 调用初始化方法
            exposedObject = initializeBean(beanName, exposedObject, mbd); 执行实例化对象的初始化方法init-method
        }
        catch (Throwable ex) {
            if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBe
anName())) {
                throw (BeanCreationException) ex;
            }
            else {
                throw new BeanCreationException(
                        mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);
            }
        }

        /**
         * 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 ev
        entually been " +
                                                "wrapped. This means that said other beans do not use the final v
        ersion of the " +
                                                "bean. This is often the result of over-eager type matching - con
        sider using " +
                                                 "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off,
         for example.");
                            }
                        }
                    }
                }
                try {
                    // 注册 bean
                    registerDisposableBeanIfNecessary(beanName, bean, mbd);
                }
                catch (BeanDefinitionValidationException ex) {
                    throw new BeanCreationException(
                            mbd.getResourceDescription(), beanName, "Invalid destruction signature", ex);
                }

                return exposedObject;
            }
```

`doCreateBean()` 是创建 bean 实例的核心方法，它的整体思路是：

1. 如果是单例模式，则清除 factoryBeanInstanceCache 缓存，同时返回 BeanWrapper 实例对象，当然如果存在。
2. 如果缓存中没有 BeanWrapper 或者不是单例模式，则调用 `createBeanInstance()` 实例化 bean，主要是将 BeanDefinition 转换为 BeanWrapper
3. MergedBeanDefinitionPostProcessor 的应用
4. 单例模式的循环依赖处理
5. 调用 `populateBean()` 进行属性填充。将所有属性填充至 bean 的实例中
6. 调用 `initializeBean()` 初始化 bean
7. 依赖检查
8. 注册 DisposableBean

实例化 bean

如果缓存中没有 BeanWrapper 实例对象或者该 bean 不是 singleton，则调用 `createBeanInstance()` 创建 bean 实例。该方法主要是根据参数 BeanDefinition、args[] 来调用构造函数实例化 bean 对象。过程较为复杂，如下：

```
    protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, @Nullable Object[] a
rgs) {
        // 解析 bean，将 bean 类名解析为 class 引用
        Class<?> beanClass = resolveBeanClass(mbd, beanName);

        if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers()) && !mbd.isNonPublicAccessAl
lowed()) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                    "Bean class isn't public, and non-public access not allowed: " + beanClass.getName
());
        }

        // 如果存在 Supplier 回调，则使用给定的回调方法初始化策略
        Supplier<?> instanceSupplier = mbd.getInstanceSupplier();
        if (instanceSupplier != null) {
            return obtainFromSupplier(instanceSupplier, beanName);
        }

        // 如果工厂方法不为空，则使用工厂方法初始化策略
        if (mbd.getFactoryMethodName() != null)  {
            return instantiateUsingFactoryMethod(beanName, mbd, args);
        }

        boolean resolved = false;
        boolean autowireNecessary = false;
        if (args == null) {
            // constructorArgumentLock 构造函数的常用锁
            synchronized (mbd.constructorArgumentLock) {
                // 如果已缓存的解析的构造函数或者工厂方法不为空，则可以利用构造函数解析
                // 因为需要根据参数确认到底使用哪个构造函数，该过程比较消耗性能，所有采用缓存机制
                if (mbd.resolvedConstructorOrFactoryMethod != null) {
                    resolved = true;
                    autowireNecessary = mbd.constructorArgumentsResolved;
                }
            }
        }
        // 已经解析好了，直接注入即可
        if (resolved) {
            // 自动注入，调用构造函数自动注入
            if (autowireNecessary) {
                return autowireConstructor(beanName, mbd, null, null);
            }
            else {
                // 使用默认构造函数构造
                return instantiateBean(beanName, mbd);
            }
        }

        // 确定解析的构造函数
        // 主要是检查已经注册的 SmartInstantiationAwareBeanPostProcessor
        Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);
```

```
        if (ctors != null ||
                mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_CONSTRUCTOR ||
                mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args))  {
            // 构造函数自动注入
            return autowireConstructor(beanName, mbd, ctors, args);
        }

        //使用默认构造函数注入
        return instantiateBean(beanName, mbd);
    }
```

实例化 bean 是一个复杂的过程，其主要的逻辑为：

- 如果存在 Supplier 回调，则调用 `obtainFromSupplier()` 进行初始化
- 如果存在工厂方法，则使用工厂方法进行初始化
- 首先判断缓存，如果缓存中存在，即已经解析过了，则直接使用已经解析了的，根据 constructorArgumentsResolved 参数来判断是使用构造函数自动注入还是默认构造函数
- 如果缓存中没有，则需要先确定到底使用哪个构造函数来完成解析工作，因为一个类有多个构造函数，每个构造函数都有不同的构造参数，所以需要根据参数来锁定构造函数并完成初始化，如果存在参数则使用相应的带有参数的构造函数，否则使用默认构造函数。

其实核心思想还是在于根据不同的情况执行不同的实例化策略，主要是包括如下四种策略：

1. Supplier 回调
2. `instantiateUsingFactoryMethod()` 工厂方法初始化
3. `autowireConstructor()`，构造函数自动注入初始化
4. `instantiateBean()`，默认构造函数注入

其实无论哪种策略，他们的实现逻辑都差不多：确定构造函数和构造方法，然后实例化。只不过相对于 Supplier 回调和默认构造函数注入而言，工厂方法初始化和构造函数自动注入初始化会比较复杂，因为他们构造函数和构造参数的不确定性，Spring 需要花大量的精力来确定构造函数和构造参数，如果确定了则好办，直接选择实例化策略即可。当然在实例化的时候会根据是否有需要覆盖或者动态替换掉的方法，因为存在覆盖或者织入的话需要创建动态代理将方法织入，这个时候就只能选择 CGLIB 的方式来实例化，否则直接利用反射的方式即可。

属性填充

属性填充其实就是将 BeanDefinition 的属性值赋值给 BeanWrapper 实例对象的过程。在填充的过程需要根据注入的类型不同来区分是根据类型注入还是名字注入，当然在这个过程还会涉及循环依赖的问题的。

```
protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper bw) {
        // 没有实例化对象
        if (bw == null) {
            // 有属性抛出异常
            if (mbd.hasPropertyValues()) {
                throw new BeanCreationException(
                        mbd.getResourceDescription(), beanName, "Cannot apply property values to null ins
tance");
            }
            else {
                // 没有属性直接返回
                return;
            }
        }

        // 在设置属性之前给 InstantiationAwareBeanPostProcessors 最后一次改变 bean 的机会
        boolean continueWithPropertyPopulation = true;

        // bena 不是"合成"的，即未由应用程序本身定义
        // 是否持有 InstantiationAwareBeanPostProcessor
        if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
            // 迭代所有的 BeanPostProcessors
            for (BeanPostProcessor bp : getBeanPostProcessors()) {
                // 如果为 InstantiationAwareBeanPostProcessor
                if (bp instanceof InstantiationAwareBeanPostProcessor) {
                    InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
                    // 返回值为是否继续填充 bean
                    // postProcessAfterInstantiation：如果应该在 bean上面设置属性则返回true，否则返回false
                    // 一般情况下，应该是返回true，返回 false 的话，
                    // 将会阻止在此 Bean 实例上调用任何后续的 InstantiationAwareBeanPostProcessor 实例。
                    if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
                        continueWithPropertyPopulation = false;
                        break;
                    }
                }
            }
        }

        // 如果后续处理器发出停止填充命令，则终止后续操作
        if (!continueWithPropertyPopulation) {
            return;
        }

        // bean 的属性值
        PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyValues() : null);

        if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME ||
                mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {

            // 将 PropertyValues 封装成 MutablePropertyValues 对象
            // MutablePropertyValues 允许对属性进行简单的操作，
```

```
            // 并提供构造函数以支持Map的深度复制和构造。
            MutablePropertyValues newPvs = new MutablePropertyValues(pvs);

            // 根据名称自动注入
            if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME) {
                autowireByName(beanName, mbd, bw, newPvs);
            }

            // 根据类型自动注入
            if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
                autowireByType(beanName, mbd, bw, newPvs);
            }

            pvs = newPvs;
        }

        // 是否已经注册了 InstantiationAwareBeanPostProcessors
        boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
        // 是否需要进行依赖检查
        boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.DEPENDENCY_CHECK_NONE);

        if (hasInstAwareBpps || needsDepCheck) {
            if (pvs == null) {
                pvs = mbd.getPropertyValues();
            }

            // 从 bw 对象中提取 PropertyDescriptor 结果集
            // PropertyDescriptor：可以通过一对存取方法提取一个属性
            PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowC
aching);
            if (hasInstAwareBpps) {
                for (BeanPostProcessor bp : getBeanPostProcessors()) {
                    if (bp instanceof InstantiationAwareBeanPostProcessor) {
                        InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) b
p;
                        // 对所有需要依赖检查的属性进行后处理
                        pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), be
anName);
                        if (pvs == null) {
                            return;
                        }
                    }
                }
            }
            if (needsDepCheck) {
                // 依赖检查，对应 depends-on 属性
                checkDependencies(beanName, mbd, filteredPds, pvs);
            }
        }

        if (pvs != null) {
            // 将属性应用到 bean 中
```

```
            applyPropertyValues(beanName, mbd, bw, pvs);
        }
    }
```

处理流程如下：

1. 根据 hasInstantiationAwareBeanPostProcessors 属性来判断是否需要在注入属性之前给 InstantiationAwareBeanPostProcessors 最后一次改变 bean 的机会，此过程可以控制 Spring 是否继续进行属性填充。
2. 根据注入类型的不同来判断是根据名称来自动注入（`autowireByName()`）还是根据类型来自动注入（`autowireByType()`），统一存入到 PropertyValues 中，PropertyValues 用于描述 bean 的属性。
3. 判断是否需要进行 BeanPostProcessor 和 依赖检测。
4. 将所有 PropertyValues 中的属性填充到 BeanWrapper 中。

初始化 bean

初始化 bean 为 `createBean()` 的最后一个过程，该过程主要做三件事情：

1. 激活 Aware 方法
2. 后置处理器的应用
3. 激活自定义的 init 方法

```
    protected Object initializeBean(final String beanName, final Object bean, @Nullable RootBeanDefinitio
n mbd) {
        if (System.getSecurityManager() != null) {
            AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
                // 激活 Aware 方法
                invokeAwareMethods(beanName, bean);
                return null;
            }, getAccessControlContext());
        }
        else {
            // 对特殊的 bean 处理：Aware、BeanClassLoaderAware、BeanFactoryAware
            invokeAwareMethods(beanName, bean);
        }

        Object wrappedBean = bean;
        if (mbd == null || !mbd.isSynthetic()) {
            // 后处理器
            wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
        }

        try {
            // 激活用户自定义的 init 方法
            invokeInitMethods(beanName, wrappedBean, mbd);
        }
        catch (Throwable ex) {
            throw new BeanCreationException(
                    (mbd != null ? mbd.getResourceDescription() : null),
                    beanName, "Invocation of init method failed", ex);
        }
        if (mbd == null || !mbd.isSynthetic()) {
            // 后处理器
            wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
        }
        return wrappedBean;
    }
```

从 bean 实例中获取对象

无论是从单例缓存中获取的 bean 实例 还是通过 `createBean()` 创建的 bean 实例，最终都会调用 `getObjectForBeanInstance()` ，该方法是根据传入的 bean 实例获取对象，按照 Spring 的传统，该方法也只是做一些检测工作，真正的实现逻辑是委托给 `getObjectFromFactoryBean()` 实现。

```
    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。

End!!! 到这里，Spring 加载 bean 的整体过程都已经分析完毕了，详情请给位移步到以下链接：

1. 【死磕 Spring】----- IOC 之加载 bean：开启 bean 的加载 (http://cmsblogs.com/?p=2806)
2. 【死磕 Spring】----- IOC 之加载 bean：从单例缓存中获取单例 bean (http://cmsblogs.com/?p=2808)
3. 【死磕 Spring】----- IOC 之加载 bean：parentBeanFactory 与依赖处理 (http://cmsblogs.com/?p=2810)
4. 【死磕 Spring】----- IOC 之加载 bean：分析各 scope 的 bean 创建 (http://cmsblogs.com/?p=2839)
5. 【死磕 Spring】—— IOC 之开启 bean 的实例化进程 (http://cmsblogs.com/?p=2846)
6. 【死磕 Spring】—— IOC 之 Factory 实例化 bean (http://cmsblogs.com/?p=2848)
7. 【死磕 Spring】—— IOC 之构造函数实例化 bean (http://cmsblogs.com/?p=2850)

8. 【死磕 Spring】—— IOC 之 属性填充 (http://cmsblogs.com/?cat=206)
9. 【死磕 Spring】—— IOC 之循环依赖处理 (http://cmsblogs.com/?p=2887)
0. 【死磕 Spring】—— IOC 之 bean 的初始化 (http://cmsblogs.com/?p=2890)

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
常用性能监控指南 (http://cmsblogs.com/?p=2903)

下一篇
分库分表技术演进&最佳实践 (http://cmsblogs.com/?p=2909)

- 【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)
- 【死磕 Spring】—— Spring 的环境&属性：PropertySource、Environment、Profile (http://cmsblogs.com/?p=4032)
- 【死磕 Spring】—— IOC 之 BeanDefinition 注册机： BeanDefinitionRegistry (http://cmsblogs.com/?p=4026)