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
2019-06-19
Spring

【死磕 Spring】—— IoC 之加载 Bean：总结

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。

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(显式或者隐式)调用 `#getBean(...)` 方法来开启，所以我们从这个方法开始。代码如下：

```
// AbstractBeanFactory.java

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 {
    // <1> 返回 bean 名称，剥离工厂引用前缀。
    // 如果 name 是 alias ，则获取对应映射的 beanName 。
    final String beanName = transformedBeanName(name);
    Object bean;

    // 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 + "'");
            }
        }
        // <2> 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);
    } else {
        // Fail if we're already creating this bean instance:
        // We're assumably within a circular reference.
        // <3> 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。
        if (isPrototypeCurrentlyInCreation(beanName)) {
            throw new BeanCurrentlyInCreationException(beanName);
        }

        // <4> 如果容器中没有找到，则从父类容器中加载
        // Check if bean definition exists in this factory.
        BeanFactory parentBeanFactory = getParentBeanFactory();
        if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
            // Not found -> check parent.
            String nameToLookup = originalBeanName(name);
            if (parentBeanFactory instanceof AbstractBeanFactory) {
                return ((AbstractBeanFactory) parentBeanFactory).doGetBean(
                        nameToLookup, requiredType, args, typeCheckOnly);
            } else if (args != null) {
                // Delegation to parent with explicit args.
                return (T) parentBeanFactory.getBean(nameToLookup, args);
            } else if (requiredType != null) {
                // No args -> delegate to standard getBean method.
                return parentBeanFactory.getBean(nameToLookup, requiredType);
            } else {
                return (T) parentBeanFactory.getBean(nameToLookup);
            }
        }

        // <5> 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 {
            // <6> 从容器中获取 beanName 相应的 GenericBeanDefinition 对象，并将其转换为 RootBeanDefinition 对象
            final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
            // 检查给定的合并的 BeanDefinition
            checkMergedBeanDefinition(mbd, beanName, args);

            // Guarantee initialization of beans that the current bean depends on.
            // <7> 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 bean
                    // 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 TODO 芋艿
                    registerDependentBean(dep, beanName);
                    try {
                        getBean(dep);
```

```
                } catch (NoSuchBeanDefinitionException ex) {
                    throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                            "'" + beanName + "' depends on missing bean '" + dep + "'", ex);
                }
            }
        }

        // <8> bean 实例化
        // Create bean instance.
        if (mbd.isSingleton()) { // 单例模式
            sharedInstance = getSingleton(beanName, () -> {
                try {
                    return createBean(beanName, mbd, args);
                }
                catch (BeansException ex) {
                    // Explicitly remove instance from singleton cache: It might have been put there
                    // eagerly by the creation process, to allow for circular reference resolution.
                    // Also remove any beans that received a temporary reference to the bean.
                    // 显式从单例缓存中删除 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。 TODO 芋艿
                    destroySingleton(beanName);
                    throw ex;
                }
            });
            bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
        } else if (mbd.isPrototype()) { // 原型模式
            // It's a prototype -> create a new instance.
            Object prototypeInstance;
            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 '" + scopeName + "'");
            }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;
            }
        }

        // <9> 检查需要的类型是否符合 bean 的实际类型
        // Check if required type matches the type of the actual bean instance.
        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.isTraceEnabled()) {
                    logger.trace("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(...)` 方法的代码量比较多，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(...)` 方法，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

1. 从缓存中获取 bean

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 ：
- 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(String beanName) 方法，从缓存中获取 singleton bean。代码如下：

```
// DefaultSingletonBeanRegistry.java

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;
}
```

该方法就是从 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）之循环依赖处理

2. 创建 bean 实例对象

如果缓存中没有，也没有 parentBeanFactory ，则会调用 #createBean(String beanName, RootBeanDefinition mbd, Object[] args) 方法，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。代码如下：

```
// AbstractBeanFactory.java

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

该方法是定义在 AbstractBeanFactory 中的抽象方法，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(final String beanName, final RootBeanDefinition mbd, final 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;
}
```

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

<1> 处，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> 处，如果缓存中没有 BeanWrapper 或者不是单例模式，则调用 #createBeanInstance(...) 方法，实例化 bean，主要是将 BeanDefinition 转换为 BeanWrapper 。
<3> 处，MergedBeanDefinitionPostProcessor 的应用。
<4> 处，单例模式的循环依赖处理。
<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。
<6> 处，调用 #initializeBean(final String beanName, final Object bean, RootBeanDefinition mbd) 方法，初始化 bean 。
<7> 处，依赖检查。
<8> 处，注册 DisposableBean 。

2.1 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(...) 方法。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，代码如下：

```
// AbstractAutowireCapableBeanFactory.java

protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) {
    // Make sure bean class is actually resolved at this point.
    // 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.isNonPublicAccessAllowed()) { // 校
    throw new BeanCreationException(mbd.getResourceDescription(), beanName,
            "Bean class isn't public, and non-public access not allowed: " + beanClass.getName());
}

// <1> 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);
}

// <2> 使用 FactoryBean 的 factory-method 来创建，支持静态工厂和实例工厂
if (mbd.getFactoryMethodName() != null)  {
    return instantiateUsingFactoryMethod(beanName, mbd, args);
}

// <3> Shortcut when re-creating the same bean...
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) {
    // <3.1> autowire 自动注入，调用构造函数自动注入
    if (autowireNecessary) {
        return autowireConstructor(beanName, mbd, null, null);
    } else {
        // <3.2> 使用默认构造函数构造
        return instantiateBean(beanName, mbd);
    }
}

// Candidate constructors for autowiring?
// <4> 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);
// <4.1> 有参数情况时，创建 Bean 。先利用参数个数，类型等，确定最精确匹配的构造方法。
if (ctors != null || mbd.getResolvedAutowireMode() == AUTOWIRE_CONSTRUCTOR ||
        mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args))  {
    return autowireConstructor(beanName, mbd, ctors, args);
}

// Preferred constructors for default construction?
// <4.1> 选择构造方法，创建 Bean 。
ctors = mbd.getPreferredConstructors();
if (ctors != null) {
    return autowireConstructor(beanName, mbd, ctors, null); // args = null
}

// No special handling: simply use no-arg constructor.
```

```
    // <4.2> 有参数时，又没获取到构造方法，则只能调用无参构造方法来创建实例了(兜底方法)
    return instantiateBean(beanName, mbd);
}
```

实例化 Bean 对象，是一个复杂的过程，其主要的逻辑为：

<1> 处，如果存在 Supplier 回调，则调用 #obtainFromSupplier(Supplier<?> instanceSupplier, String beanName) 方法，进行初始化。
<2> 处，如果存在工厂方法，则使用工厂方法进行初始化。
<3> 处，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：
- <3.1> 处，是使用构造函数自动注入，即调用 #autowireConstructor(String beanName, RootBeanDefinition mbd, Constructor<?>[] ctors, Object[] explicitArgs) 方法。
- <3.2> 处，还是默认构造函数，即调用 #instantiateBean(final String beanName, final RootBeanDefinition mbd) 方法。

<4> 处，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。
- <4.1> 处，如果存在参数，则使用相应的带有参数的构造函数，即调用 #autowireConstructor(String beanName, RootBeanDefinition mbd, Constructor<?>[] ctors, Object[] explicitArgs) 方法。
- <4.2> 处，否则，使用默认构造函数，即调用 #instantiateBean(final String beanName, final RootBeanDefinition mbd) 方法。

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(...) 方法，工厂方法初始化
3. #autowireConstructor(...) 方法，构造函数自动注入初始化
4. #instantiateBean(...) 方法，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。

2.2 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。代码如下：

```
// AbstractAutowireCapableBeanFactory.java

protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper bw) {
    // 没有实例化对象
    if (bw == null) {
        // 有属性，则抛出 BeanCreationException 异常
        if (mbd.hasPropertyValues()) {
            throw new BeanCreationException(
```

```
                mbd.getResourceDescription(), beanName, "Cannot apply property values to null instance");
        // 没有属性，直接 return 返回
    } else {
        // Skip property population phase for null instance.
        return;
    }
}

// <1> 在设置属性之前给 InstantiationAwareBeanPostProcessors 最后一次改变 bean 的机会
// Give any InstantiationAwareBeanPostProcessors the opportunity to modify the
// state of the bean before properties are set. This can be used, for example,
// to support styles of field injection.
boolean continueWithPropertyPopulation = true;
if (!mbd.isSynthetic()  // bean 不是"合成"的，即未由应用程序本身定义
        && hasInstantiationAwareBeanPostProcessors()) { // 是否持有 InstantiationAwareBeanPostProcessor
    // 迭代所有的 BeanPostProcessors
    for (BeanPostProcessor bp : getBeanPostProcessors()) {
        if (bp instanceof InstantiationAwareBeanPostProcessor) { // 如果为 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);

// <2> 自动注入
if (mbd.getResolvedAutowireMode() == AUTOWIRE_BY_NAME || mbd.getResolvedAutowireMode() == 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);
    // Add property values based on autowire by name if applicable.
    // 根据名称自动注入
    if (mbd.getResolvedAutowireMode() == AUTOWIRE_BY_NAME) {
        autowireByName(beanName, mbd, bw, newPvs);
    }
    // Add property values based on autowire by type if applicable.
    // 根据类型自动注入
    if (mbd.getResolvedAutowireMode() == 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() != AbstractBeanDefinition.DEPENDENCY_CHECK_NONE);
```

```
        // <3> BeanPostProcessor 处理
        PropertyDescriptor[] filteredPds = null;
        if (hasInstAwareBpps) {
            if (pvs == null) {
                pvs = mbd.getPropertyValues();
            }
            // 遍历 BeanPostProcessor 数组
            for (BeanPostProcessor bp : getBeanPostProcessors()) {
                if (bp instanceof InstantiationAwareBeanPostProcessor) {
                    InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
                    // 对所有需要依赖检查的属性进行后处理
                    PropertyValues pvsToUse = ibp.postProcessProperties(pvs, bw.getWrappedInstance(), beanName);
                    if (pvsToUse == null) {
                        // 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
                        if (filteredPds == null) {
                            filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
                        }
                        pvsToUse = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);
                        if (pvsToUse == null) {
                            return;
                        }
                    }
                    pvs = pvsToUse;
                }
            }
        }

        // <4> 依赖检查
        if (needsDepCheck) {
            if (filteredPds == null) {
                filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
            }
            // 依赖检查，对应 depends-on 属性
            checkDependencies(beanName, mbd, filteredPds, pvs);
        }

        // <5> 将属性应用到 bean 中
        if (pvs != null) {
            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。
统一存入到 PropertyValues 中，PropertyValues 用于描述 bean 的属性。

- <2> ，根据注入类型(AbstractBeanDefinition#getResolvedAutowireMode() 方法的返回值)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(...)）
- <3> ，进行 BeanPostProcessor 处理。
- <4> ，依赖检测。

<5> ，将所有 PropertyValues 中的属性，填充到 BeanWrapper 中。

2.3 初始化 bean

初始化 bean 为 #createBean(...) 方法的最后一个过程，代码如下：

```
// AbstractAutowireCapableBeanFactory.java

protected Object initializeBean(final String beanName, final Object bean, @Nullable RootBeanDefinition mbd) {
    if (System.getSecurityManager() != null) { // 安全模式
        AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
            // <1> 激活 Aware 方法，对特殊的 bean 处理：Aware、BeanClassLoaderAware、BeanFactoryAware
            invokeAwareMethods(beanName, bean);
            return null;
        }, getAccessControlContext());
    } else {
        // <1> 激活 Aware 方法，对特殊的 bean 处理：Aware、BeanClassLoaderAware、BeanFactoryAware
        invokeAwareMethods(beanName, bean);
    }

    // <2> 后处理器，before
    Object wrappedBean = bean;
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
    }

    // <3> 激活用户自定义的 init 方法
    try {
        invokeInitMethods(beanName, wrappedBean, mbd);
    } catch (Throwable ex) {
        throw new BeanCreationException(
                (mbd != null ? mbd.getResourceDescription() : null),
                beanName, "Invocation of init method failed", ex);
    }

    // <2> 后处理器，after
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
    }

    return wrappedBean;
}
```

初始化 bean 的方法其实就是三个步骤的处理，而这三个步骤主要还是根据用户设定的来进行初始化，这三个过程为：

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

3. 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(...) 方法来创建的 bean 实例，最终都会调用 #getObjectForBeanInstance(...) 方法来根据传入的 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(...) 方法来实现。代码如下：

```
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。
<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。
<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。
- 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。

3. 小结

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
2. 【死磕 Spring】—— IoC 之加载 bean：从单例缓存中获取单例 bean
3. 【死磕 Spring】—— IoC 之加载 bean：parentBeanFactory 与依赖处理
4. 【死磕 Spring】—— IoC 之加载 bean：分析各 scope 的 bean 创建
5. 【死磕 Spring】—— IoC 之加载 bean：创建 bean（一）之主流程
6. 【死磕 Spring】—— IoC 之加载 bean：创建 bean（二）之实例化 Bean 对象(1)
7. 【死磕 Spring】—— IoC 之加载 bean：创建 bean（三）之实例化 Bean 对象(2)
8. 【死磕 Spring】—— IoC 之加载 bean：创建 bean（四）之属性填充
9. 【死磕 Spring】—— IoC 之加载 bean：创建 bean（五）之循环依赖处理
10. 【死磕 Spring】—— IoC 之加载 bean：创建 bean（六）之初始化 Bean 对象

文章目录

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