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
2019-05-07
Spring

【死磕 Spring】—— IoC 之加载 Bean：parentBeanFactory 与依赖处理

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。

如果从单例缓存中没有获取到单例 Bean 对象，则说明两种两种情况：

1. 该 Bean 的 scope 不是 singleton
2. 该 Bean 的 scope 是 singleton ，但是没有初始化完成。

针对这两种情况，Spring 是如何处理的呢？统一加载并完成初始化！这部分内容的篇幅较长，拆分为两部分：

> 第一部分，主要是一些检测、parentBeanFactory 以及依赖处理。
> 第二部分则是各个 scope 的初始化。

代码如下：

```
// AbstractBeanFactory.java
//protected <T> T doGetBean(final String name, final Class<T> requiredType, final Object[] args, boolean typeCheckOnl

// ... 省略很多代码

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

    // ... 省略很多代码
```

这段代码主要处理如下几个部分：

<3> 处，检测。若当前 Bean 在创建，则抛出 BeanCurrentlyInCreationException 异常。

- 详细解析，见 「1. 检测」 。

<4> 处，如果 beanDefinitionMap 中不存在 beanName 的 BeanDefinition（即在 Spring bean 初始化过程中没有加载），则尝试从 parentBeanFactory 中加载。

- 详细解析，见 「2. 检查父类 BeanFactory」 。

<5> 处，判断是否为类型检查。
- 详细解析，见 「3. 类型检查」 。

<6> 处，从 mergedBeanDefinitions 中获取 beanName 对应的 RootBeanDefinition 对象。如果这个 BeanDefinition 是子 Bean 的话，则会合并父类的相关属性。
- 详细解析，见 「4. 获取 RootBeanDefinition」 。

<7> 处，依赖处理。
- 详细解析，见 「5. 处理依赖」 。

1. 检测

在前面就提过，Spring 只解决单例模式下的循环依赖，对于原型模式的循环依赖则是抛出 BeanCurrentlyInCreationException 异常，所以首先检查该 beanName 是否处于原型模式下的循环依赖。如下：

```
// AbstractBeanFactory.java

if (isPrototypeCurrentlyInCreation(beanName)) {
    throw new BeanCurrentlyInCreationException(beanName);
}
```

调用 #isPrototypeCurrentlyInCreation(String beanName) 方法，判断当前 Bean 是否正在创建。代码如下：

```
// AbstractBeanFactory.java

protected boolean isPrototypeCurrentlyInCreation(String beanName) {
    Object curVal = this.prototypesCurrentlyInCreation.get();
 return (curVal != null &&
            (curVal.equals(beanName)  // 相等
                    || (curVal instanceof Set && ((Set<?>) curVal).contains(beanName)))); // 包含
}
```

- 其实检测逻辑和单例模式一样，一个“集合”存放着正在创建的 Bean ，从该集合中进行判断即可，只不过单例模式的“集合”为 Set ，而原型模式的则是 ThreadLocal 。prototypesCurrentlyInCreation 定义如下：

```
// AbstractBeanFactory.java

/** Names of beans that are currently in creation. */
private final ThreadLocal<Object> prototypesCurrentlyInCreation =
     new NamedThreadLocal<>("Prototype beans currently in creation");
```

2. 检查父类 BeanFactory

若 #containsBeanDefinition(String beanName) 方法中不存在 beanName 相对应的 BeanDefinition 对象时，则从 parentBeanFactory 中获取。代码如下：

```
// AbstractBeanFactory.java
```

```
// 获取 parentBeanFactory
BeanFactory parentBeanFactory = getParentBeanFactory();
// parentBeanFactory 不为空且 beanDefinitionMap 中不存该 name 的 BeanDefinition
if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
    // 确定原始 beanName
    String nameToLookup = originalBeanName(name);
    // 若为 AbstractBeanFactory 类型，委托父类处理
    if (parentBeanFactory instanceof AbstractBeanFactory) {
        return ((AbstractBeanFactory) parentBeanFactory).doGetBean(
                nameToLookup, requiredType, args, typeCheckOnly);
    } else if (args != null) {
        // 委托给构造函数 getBean() 处理
        return (T) parentBeanFactory.getBean(nameToLookup, args);
    } else {
        // 没有 args，委托给标准的 getBean() 处理
        return parentBeanFactory.getBean(nameToLookup, requiredType);
    }
}
```

整个过程较为简单，都是委托 parentBeanFactory 的 #getBean(...) 方法来进行处理，只不过在获取之前对 breanName 进行简单的处理，主要是想获取原始的 beanName 。代码如下：

```
// AbstractBeanFactory.java

protected String originalBeanName(String name) {
    String beanName = transformedBeanName(name); // <1>
 if (name.startsWith(FACTORY_BEAN_PREFIX)) { // <2>
        beanName = FACTORY_BEAN_PREFIX + beanName;
    }
 return beanName;
}
```

- <1> 处，#transformedBeanName(String name) 方法，是对 name 进行转换，获取真正的 beanName 。在 《【死磕 Spring】—— IoC 之开启 Bean 的加载》 中，已经有详细解析。
- <2> 处，如果 name 是以 “&” 开头的，则加上 “&” ，因为在 #transformedBeanName(String name) 方法，将 “&” 去掉了，这里补上。

3. 类型检查

方法参数 typeCheckOnly ，是用来判断调用 #getBean(...) 方法时，表示是否为仅仅进行类型检查获取 Bean 对象。如果不是仅仅做类型检查，而是创建 Bean 对象，则需要调用 #markBeanAsCreated(String beanName) 方法，进行记录。代码如下：

```
// AbstractBeanFactory.java

/**
 *  Names of beans that have already been created at least once.
 *
 *  已创建 Bean 的名字集合
 */
private final Set<String> alreadyCreated = Collections.newSetFromMap(new ConcurrentHashMap<>(256));
```

```
protected void markBeanAsCreated(String beanName) {
    // 没有创建
    if (!this.alreadyCreated.contains(beanName)) {
        // 加上全局锁
        synchronized (this.mergedBeanDefinitions) {
            // 再次检查一次：DCL 双检查模式
            if (!this.alreadyCreated.contains(beanName)) {
                // Let the bean definition get re-merged now that we're actually creating
                // the bean... just in case some of its metadata changed in the meantime.
                // 从 mergedBeanDefinitions 中删除 beanName，并在下次访问时重新创建它。
                clearMergedBeanDefinition(beanName);
                // 添加到已创建 bean 集合中
                this.alreadyCreated.add(beanName);
            }
        }
    }
}

protected void clearMergedBeanDefinition(String beanName) {
    this.mergedBeanDefinitions.remove(beanName);
}
```

4. 获取 RootBeanDefinition

```
// AbstractBeanFactory.java

// 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);
```

调用 `#getMergedLocalBeanDefinition(String beanName)` 方法，获取相对应的 BeanDefinition 对象。代码如下：

```
// AbstractBeanFactory.java

/** Map from bean name to merged RootBeanDefinition. */
private final Map<String, RootBeanDefinition> mergedBeanDefinitions = new ConcurrentHashMap<>(256);

protected RootBeanDefinition getMergedLocalBeanDefinition(String beanName) throws BeansException {
    // Quick check on the concurrent map first, with minimal locking.
    // 快速从缓存中获取，如果不为空，则直接返回
    RootBeanDefinition mbd = this.mergedBeanDefinitions.get(beanName);
    if (mbd != null) {
        return mbd;
    }
    // 获取 RootBeanDefinition，
    // 如果返回的 BeanDefinition 是子类 bean 的话，则合并父类相关属性
    return getMergedBeanDefinition(beanName, getBeanDefinition(beanName));
}
```

- 首先，直接从 `mergedBeanDefinitions` 缓存中获取相应的 RootBeanDefinition 对象，如果存在则直接返回。

- 否则，调用 #getMergedBeanDefinition(String beanName, BeanDefinition bd) 方法，获取 RootBeanDefinition 对象。若获取的 BeanDefinition 为子 BeanDefinition，则需要合并父类的相关属性。关于该方法的源码，本文不做详细解析。感兴趣的胖友，可以自己研究。

调用 #checkMergedBeanDefinition() 方法，检查给定的合并的 BeanDefinition 对象。代码如下：

```
// AbstractBeanFactory.java

protected void checkMergedBeanDefinition(RootBeanDefinition mbd, String beanName, @Nullable Object[] args)
     throws BeanDefinitionStoreException {
 if (mbd.isAbstract()) {
     throw new BeanIsAbstractException(beanName);
    }
}
```

5. 处理依赖

如果一个 Bean 有依赖 Bean 的话，那么在初始化该 Bean 时是需要先初始化它所依赖的 Bean 。代码如下：

```
// AbstractBeanFactory.java

// Guarantee initialization of beans that the current bean depends on.
// 处理所依赖的 bean
String[] dependsOn = mbd.getDependsOn();
if (dependsOn != null) {
    for (String dep : dependsOn) {
        // <1> 若给定的依赖 bean 已经注册为依赖给定的 bean
        // 即循环依赖的情况，抛出 BeanCreationException 异常
        if (isDependent(beanName, dep)) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                    "Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");
        }
        // <2> 缓存依赖调用 TODO 芋艿
        registerDependentBean(dep, beanName);
        try {
            // <3> 递归处理依赖 Bean
            getBean(dep);
        } catch (NoSuchBeanDefinitionException ex) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                    "'" + beanName + "' depends on missing bean '" + dep + "'", ex);
        }
    }
}
```

这段代码逻辑是：通过迭代的方式依次对依赖 bean 进行检测、校验。如果通过，则调用 #getBean(String beanName) 方法，实例化依赖的 Bean 对象。

5.1 isDependent

<1> 处，调用 #isDependent(String beanName, String dependentBeanName) 方法，是校验该依赖是否已经注册给当前 Bean 。代码如下：

```
// DefaultSingletonBeanRegistry.java

/**
 * Map between dependent bean names: bean name to Set of dependent bean names.
 *
 * 保存的是依赖 beanName 之间的映射关系：beanName - > 依赖 beanName 的集合
 */
private final Map<String, Set<String>> dependentBeanMap = new ConcurrentHashMap<>(64);

protected boolean isDependent(String beanName, String dependentBeanName) {
 synchronized (this.dependentBeanMap) {
     return isDependent(beanName, dependentBeanName, null);
    }
}
```

dependentBeanMap 对象保存的是依赖 beanName 之间的映射关系：beanName - > 依赖 beanName 的集合。

同步加锁给 dependentBeanMap 对象，然后调用 #isDependent(String beanName, String dependentBeanName, Set<String> alreadySeen) 方法，进行校验。代码如下：

```
// DefaultSingletonBeanRegistry.java

private boolean isDependent(String beanName, String dependentBeanName, @Nullable Set<String> alreadySeen) {
    // alreadySeen 已经检测的依赖 bean
    if (alreadySeen != null && alreadySeen.contains(beanName)) {
        return false;
    }
    // 获取原始 beanName
    String canonicalName = canonicalName(beanName);
    // 获取当前 beanName 的依赖集合
    Set<String> dependentBeans = this.dependentBeanMap.get(canonicalName);
    if (dependentBeans == null) {
        return false;
    }
    // 存在，则证明存在已经注册的依赖
    if (dependentBeans.contains(dependentBeanName)) {
        return true;
    }
    // 递归检测依赖
    for (String transitiveDependency : dependentBeans) {
        if (alreadySeen == null) {
            alreadySeen = new HashSet<>();
        }
        // 添加到 alreadySeen 中
        alreadySeen.add(beanName);
        // 递推
        if (isDependent(transitiveDependency, dependentBeanName, alreadySeen)) {
            return true;
        }
    }
    return false;
}
```

- 代码比较长，当然也有点绕。感兴趣的胖友，可以调试下。

5.2 registerDependentBean

<2> 处，如果校验成功，则调用 #registerDependentBean(String beanName, String dependentBeanName) 方法，将该依赖进行注册，便于在销毁 Bean 之前对其进行销毁。代码如下：

```
// DefaultSingletonBeanRegistry.java

/**
 * Map between dependent bean names: bean name to Set of dependent bean names.
 *
 * 保存的是依赖 beanName 之间的映射关系：beanName - > 依赖 beanName 的集合
 */
private final Map<String, Set<String>> dependentBeanMap = new ConcurrentHashMap<>(64);

/**
 * Map between depending bean names: bean name to Set of bean names for the bean's dependencies.
 *
 * 保存的是依赖 beanName 之间的映射关系：依赖 beanName - > beanName 的集合
 */
private final Map<String, Set<String>> dependenciesForBeanMap = new ConcurrentHashMap<>(64);

public void registerDependentBean(String beanName, String dependentBeanName) {
    // 获取 beanName
    String canonicalName = canonicalName(beanName);

    // 添加 <canonicalName, <dependentBeanName>> 到 dependentBeanMap 中
    synchronized (this.dependentBeanMap) {
        Set<String> dependentBeans =
                this.dependentBeanMap.computeIfAbsent(canonicalName, k -> new LinkedHashSet<>(8));
        if (!dependentBeans.add(dependentBeanName)) {
            return;
        }
    }

    // 添加 <dependentBeanName, <canonicalName>> 到 dependenciesForBeanMap 中
    synchronized (this.dependenciesForBeanMap) {
        Set<String> dependenciesForBean =
                this.dependenciesForBeanMap.computeIfAbsent(dependentBeanName, k -> new LinkedHashSet<>(8));
        dependenciesForBean.add(canonicalName);
    }
}
```

其实将就是该映射关系保存到两个集合中：dependentBeanMap、dependenciesForBeanMap 。

5.3 getBean

<3> 处，最后调用 #getBean(String beanName) 方法，实例化依赖 Bean 对象。

6. 小结

至此，加载 bean 的第二个部分也分析完毕了，下篇开始分析第三个部分：各大作用域 bean 的处理。

文章目录

1. 1. 检测
2. 2. 检查父类 BeanFactory
3. 3. 类型检查
4. 4. 获取 RootBeanDefinition
5. 5. 处理依赖
    1. 5.1. 5.1 isDependent
    2. 5.2. 5.2 registerDependentBean
    3. 5.3. 5.3 getBean
6. 6. 小结