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

2018-10-18 分类：死磕 Spring (http://cmsblogs.com/?cat=206) 阅读(8332) 评论(0)

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

继上篇博客 【死磕 Spring】----- 加载 bean 之 缓存中获取单例 bean (http://cmsblogs.com/?p=2808),如果从单例缓存中没有获取到单例 bean，则说明两种情况：

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。

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

        // 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 {
                // No args -> delegate to standard getBean method.
                return parentBeanFactory.getBean(nameToLookup, requiredType);
            }
        }
        if (!typeCheckOnly) {
            markBeanAsCreated(beanName);
        }

        try {
            final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
            checkMergedBeanDefinition(mbd, beanName, args);

            // Guarantee initialization of beans that the current bean depends on.
            String[] dependsOn = mbd.getDependsOn();
            if (dependsOn != null) {
                for (String dep : dependsOn) {
                    if (isDependent(beanName, dep)) {
                        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                                "Circular depends-on relationship between '" + beanName + "' and '" +
dep + "'");
                    }
                    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：

1. 检测。若当前 bean 在创建，则抛出 BeanCurrentlyInCreationException 异常。
2. 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。
3. 判断是否为类型检查。
4. 从 mergedBeanDefinitions 中获取 beanName 对应的 RootBeanDefinition，如果这个 BeanDefinition 是子 Bean 的话，则会合并父类的相关属性。
5. 依赖处理。

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：

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

调用 `isPrototypeCurrentlyInCreation()` 判断当前 bean 是否正在创建，如下：

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

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：

```
    private final ThreadLocal<Object> prototypesCurrentlyInCreation = new NamedThreadLocal<>("Prototype b
eans currently in creation");
```

检查父类 BeanFactory 若 containsBeanDefinition 中不存在 beanName 相对应的 BeanDefinition，则从 parentBeanFactory 中获取。

```
// 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()` 进行处理，只不过在获取之前对 name 进行简单的处理，主要是想获取原始的 beanName，如下：

```
protected String originalBeanName(String name) {
    String beanName = transformedBeanName(name);
    if (name.startsWith(FACTORY_BEAN_PREFIX)) {
        beanName = FACTORY_BEAN_PREFIX + beanName;
    }
    return beanName;
}
```

`transformedBeanName()` 是对 name 进行转换，获取真正的 beanName，因为我们传递的可能是 aliasName （这个过程在博客 【死磕 Spring】----- 加载 bean 之 开启 bean 的加载 (http://cmsblogs.com/?p=2806) 中分析 `transformedBeanName()` 有详细说明），如果 name 是以 “&” 开头的，则加上 “&”，因为在 `transformedBeanName()` 将 “&” 去掉了，这里补上。 **类型检查** 参数 typeCheckOnly 是用来判断调用 `getBean()` 是否为类型检查获取 bean。如果不是仅仅做类型检查则是创建 bean，则需要调用 `markBeanAsCreated()` 记录：

```
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)) {
                // 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);
            }
        }
    }
}
```

获取 RootBeanDefinition

```
final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
```

调用 `getMergedLocalBeanDefinition()` 获取相对应的 BeanDefinition，如下：

```
protected RootBeanDefinition getMergedLocalBeanDefinition(String beanName) throws BeansException {
    // 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，如果存在则直接返回否则调用 `getMergedBeanDefinition()` 获取 RootBeanDefinition，若获取的 BeanDefinition 为子 BeanDefinition，则需要合并父类的相关属性。 **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。

```
        // 获取依赖。
        // 在初始化 bean 时解析 depends-on 标签时设置
        String[] dependsOn = mbd.getDependsOn();
        if (dependsOn != null) {
            // 迭代依赖
        for (String dep : dependsOn) {
            // 检验依赖的bean 是否已经注册给当前 bean 获取其他传递依赖bean
            if (isDependent(beanName, dep)) {
                throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                        "Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");
            }
            // 注册到依赖bean中
            registerDependentBean(dep, beanName);
            try {
                // 调用 getBean 初始化依赖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()` 实例化依赖 bean。 `isDependent()` 是校验该依赖是否已经注册给当前 bean。

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

同步加锁给 dependentBeanMap 对象，然后调用 `isDependent()` 校验。dependentBeanMap 对象保存的是依赖 beanName 之间的映射关系：beanName - > 依赖 beanName 的集合

```
    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);
        // 不存在依赖，返回false
        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<>();
            }
            alreadySeen.add(beanName);
            if (isDependent(transitiveDependency, dependentBeanName, alreadySeen)) {
                return true;
            }
        }
        return false;
    }
```

如果校验成功，则调用 `registerDependentBean()` 将该依赖进行注册，便于在销毁 bean 之前对其进行销毁。

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

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

        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。 最后调用 `getBean()` 实例化依赖 bean。 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

更多阅读

- 【死磕 Spring】----- IOC 之 IOC 初始化总结 (http://cmsblogs.com/?p=2790)
- 【死磕 Spring】----- 加载 bean 之 开启 bean 的加载 (http://cmsblogs.com/?p=2806)
- 【死磕 Spring】----- 加载 bean 之 缓存中获取单例 bean (http://cmsblogs.com/?p=2808)

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 之从单例缓存中获取单例 bean (http://cmsblogs.com/?p=2808)

下一篇
MySQL的索引是什么？怎么优化？ (http://cmsblogs.com/?p=2818)

- 【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)