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
2019-04-01
Spring

【死磕 Spring】—— IoC 之注册解析的 BeanDefinitions

本文主要基于 Spring 5.0.6.RELEASE

摘要: 原创出处 http://cmsblogs.com/?p=2763 「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。

DefaultBeanDefinitionDocumentReader 的 #processBeanDefinition() 方法，完成 Bean 标签解析的核心工作。代码如下：

```
// DefaultBeanDefinitionDocumentReader.java

protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {
    // 进行 bean 元素解析。
    // 如果解析成功，则返回 BeanDefinitionHolder 对象。而 BeanDefinitionHolder 为 name 和 alias 的 BeanDefinition 对象
    // 如果解析失败，则返回 null 。
    BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
    if (bdHolder != null) {
        // 进行自定义标签处理
        bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
        try {
            // 进行 BeanDefinition 的注册
            // Register the final decorated instance.
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext().getRegistry());
        } catch (BeanDefinitionStoreException ex) {
            getReaderContext().error("Failed to register bean definition with name '" +
                    bdHolder.getBeanName() + "'", ele, ex);
        }
        // 发出响应事件，通知相关的监听器，已完成该 Bean 标签的解析。
        // Send registration event.
        getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));
    }
}
```

解析工作分为三步：
- 1、解析默认标签。
- 2、解析默认标签后下得自定义标签。

- 3、注册解析后的 BeanDefinition 。

经过前面两个步骤的解析，这时的 BeanDefinition 已经可以满足后续的使用要求了，那么接下来的工作就是将这些 BeanDefinition 进行注册，也就是完成第三步。

1. BeanDefinitionReaderUtils

注册 BeanDefinition ，由 BeanDefinitionReaderUtils.registerBeanDefinition() 完成。代码如下：

```
// BeanDefinitionReaderUtils.java

public static void registerBeanDefinition(
        BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry registry)
        throws BeanDefinitionStoreException {

    // 注册 beanName
    // Register bean definition under primary name.
    String beanName = definitionHolder.getBeanName();
    registry.registerBeanDefinition(beanName, definitionHolder.getBeanDefinition());

    // 注册 alias
    // Register aliases for bean name, if any.
    String[] aliases = definitionHolder.getAliases();
    if (aliases != null) {
        for (String alias : aliases) {
            registry.registerAlias(beanName, alias);
        }
    }
}
```

首先，通过 beanName 注册 BeanDefinition 。详细解析，见 2.1 通过 beanName 注册 。然后，再通过注册别名 alias 和 beanName 的映射。详细解析，见 2.2 注册 alias 和 beanName 的映射 。

2. BeanDefinitionRegistry

BeanDefinition 的注册，由接口 org.springframework.beans.factory.support.BeanDefinitionRegistry 定义。

2.1 通过 beanName 注册

调用 BeanDefinitionRegistry 的 #registerBeanDefinition(String beanName, BeanDefinition beanDefinition) 方法，实现通过 beanName 注册 BeanDefinition 。代码如下：

```
// DefaultListableBeanFactory.java

/** Whether to allow re-registration of a different definition with the same name. */
private boolean allowBeanDefinitionOverriding = true;

/** Map of bean definition objects, keyed by bean name. */
private final Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap<>(256);
/** List of bean definition names, in registration order. */
private volatile List<String> beanDefinitionNames = new ArrayList<>(256);
```

```
/** List of names of manually registered singletons, in registration order. */
private volatile Set<String> manualSingletonNames = new LinkedHashSet<>(16);
/** Cached array of bean definition names in case of frozen configuration. */
@Nullable
private volatile String[] frozenBeanDefinitionNames;

@Override
public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
        throws BeanDefinitionStoreException {

    // 校验 beanName 与 beanDefinition 非空
    Assert.hasText(beanName, "Bean name must not be empty");
    Assert.notNull(beanDefinition, "BeanDefinition must not be null");

    // <1> 校验 BeanDefinition 。
    // 这是注册前的最后一次校验了，主要是对属性 methodOverrides 进行校验。
    if (beanDefinition instanceof AbstractBeanDefinition) {
        try {
            ((AbstractBeanDefinition) beanDefinition).validate();
        } catch (BeanDefinitionValidationException ex) {
            throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,
                    "Validation of bean definition failed", ex);
        }
    }

    // <2> 从缓存中获取指定 beanName 的 BeanDefinition
    BeanDefinition existingDefinition = this.beanDefinitionMap.get(beanName);
    // <3> 如果已经存在
    if (existingDefinition != null) {
        // 如果存在但是不允许覆盖，抛出异常
        if (!isAllowBeanDefinitionOverriding()) {
            throw new BeanDefinitionOverrideException(beanName, beanDefinition, existingDefinition);
        // 覆盖 beanDefinition 大于 被覆盖的 beanDefinition 的 ROLE ，打印 info 日志
        } else if (existingDefinition.getRole() < beanDefinition.getRole()) {
            // e.g. was ROLE_APPLICATION, now overriding with ROLE_SUPPORT or ROLE_INFRASTRUCTURE
            if (logger.isInfoEnabled()) {
                logger.info("Overriding user-defined bean definition for bean '" + beanName +
                        "' with a framework-generated bean definition: replacing [" +
                        existingDefinition + "] with [" + beanDefinition + "]");
            }
        // 覆盖 beanDefinition 与 被覆盖的 beanDefinition 不相同，打印 debug 日志
        } else if (!beanDefinition.equals(existingDefinition)) {
            if (logger.isDebugEnabled()) {
                logger.debug("Overriding bean definition for bean '" + beanName +
                        "' with a different definition: replacing [" + existingDefinition +
                        "] with [" + beanDefinition + "]");
            }
        // 其它，打印 debug 日志
        } else {
            if (logger.isTraceEnabled()) {
                logger.trace("Overriding bean definition for bean '" + beanName +
                        "' with an equivalent definition: replacing [" + existingDefinition +
                        "] with [" + beanDefinition + "]");
            }
        }
        // 允许覆盖，直接覆盖原有的 BeanDefinition 到 beanDefinitionMap 中。
        this.beanDefinitionMap.put(beanName, beanDefinition);
    // <4> 如果未存在
    } else {
        // 检测创建 Bean 阶段是否已经开启，如果开启了则需要对 beanDefinitionMap 进行并发控制
```

```
            if (hasBeanCreationStarted()) {
                // beanDefinitionMap 为全局变量，避免并发情况
                // Cannot modify startup-time collection elements anymore (for stable iteration)
                synchronized (this.beanDefinitionMap) {
                    // 添加到 BeanDefinition 到 beanDefinitionMap 中。
                    this.beanDefinitionMap.put(beanName, beanDefinition);
                    // 添加 beanName 到 beanDefinitionNames 中
                    List<String> updatedDefinitions = new ArrayList<>(this.beanDefinitionNames.size() + 1);
                    updatedDefinitions.addAll(this.beanDefinitionNames);
                    updatedDefinitions.add(beanName);
                    this.beanDefinitionNames = updatedDefinitions;
                    // 从 manualSingletonNames 移除 beanName
                    if (this.manualSingletonNames.contains(beanName)) {
                        Set<String> updatedSingletons = new LinkedHashSet<>(this.manualSingletonNames);
                        updatedSingletons.remove(beanName);
                        this.manualSingletonNames = updatedSingletons;
                    }
                }
            } else {
                // Still in startup registration phase
                // 添加到 BeanDefinition 到 beanDefinitionMap 中。
                this.beanDefinitionMap.put(beanName, beanDefinition);
                // 添加 beanName 到 beanDefinitionNames 中
                this.beanDefinitionNames.add(beanName);
                // 从 manualSingletonNames 移除 beanName
                this.manualSingletonNames.remove(beanName);
            }

            this.frozenBeanDefinitionNames = null;
        }

        // <5> 重新设置 beanName 对应的缓存
        if (existingDefinition != null || containsSingleton(beanName)) {
            resetBeanDefinition(beanName);
        }
    }
```

处理过程如下：

<1> 对 BeanDefinition 进行校验，该校验也是注册过程中的最后一次校验了，主要是对 AbstractBeanDefinition 的 methodOverrides 属性进行校验。
<2> 根据 beanName 从缓存中获取 BeanDefinition 对象。
<3> 如果缓存中存在，则根据 allowBeanDefinitionOverriding 标志来判断是否允许覆盖。如果允许则直接覆盖。否则，抛出 BeanDefinitionStoreException 异常。
<4> 若缓存中没有指定 beanName 的 BeanDefinition，则判断当前阶段是否已经开始了 Bean 的创建阶段？如果是，则需要对 beanDefinitionMap 进行加锁控制并发问题，否则直接设置即可。
- 对于 #hasBeanCreationStarted() 方法，后续做详细介绍，这里不过多阐述。

<5> 若缓存中存在该 beanName 或者单例 bean 集合中存在该 beanName ，则调用 #resetBeanDefinition(String beanName) 方法，重置 BeanDefinition 缓存。

其实整段代码的核心就在于 this.beanDefinitionMap.put(beanName, beanDefinition); 代码块。而 BeanDefinition 的缓存也不是神奇的东西，就是定义一个 Map ：

key 为 beanName 。
value 为 BeanDefinition 对象。

2.2 注册 alias 和 beanName 的映射

调用 BeanDefinitionRegistry 的 #registerAlias(String name, String alias) 方法，注册 alias 和 beanName 的映射关系。代码如下：

```
// SimpleAliasRegistry.java

/** Map from alias to canonical name. */
// key: alias
// value: beanName
private final Map<String, String> aliasMap = new ConcurrentHashMap<>(16);

@Override
public void registerAlias(String name, String alias) {
    // 校验 name 、 alias
    Assert.hasText(name, "'name' must not be empty");
    Assert.hasText(alias, "'alias' must not be empty");
    synchronized (this.aliasMap) {
        // name == alias 则去掉alias
        if (alias.equals(name)) {
            this.aliasMap.remove(alias);
            if (logger.isDebugEnabled()) {
                logger.debug("Alias definition '" + alias + "' ignored since it points to same name");
            }
        } else {
            // 获取 alias 已注册的 beanName
            String registeredName = this.aliasMap.get(alias);
            // 已存在
            if (registeredName != null) {
                // 相同，则 return ，无需重复注册
                if (registeredName.equals(name)) {
                    // An existing alias - no need to re-register
                    return;
                }
                // 不允许覆盖，则抛出 IllegalStateException 异常
                if (!allowAliasOverriding()) {
                    throw new IllegalStateException("Cannot define alias '" + alias + "' for name '" +
                            name + "': It is already registered for name '" + registeredName + "'.");
                }
                if (logger.isDebugEnabled()) {
                    logger.debug("Overriding alias '" + alias + "' definition for registered name '" +
                            registeredName + "' with new target name '" + name + "'");
                }
            }
            // 校验，是否存在循环指向
            checkForAliasCircle(name, alias);
            // 注册 alias
            this.aliasMap.put(alias, name);
            if (logger.isTraceEnabled()) {
                logger.trace("Alias definition '" + alias + "' registered for name '" + name + "'");
            }
        }
    }
}
```

注册 alias 和注册 BeanDefinition 的过程差不多。

在最后，调用了 #checkForAliasCircle() 来对别名进行了循环检测。代码如下：

```
protected void checkForAliasCircle(String name, String alias) {
    if (hasAlias(alias, name)) {
        throw new IllegalStateException("Cannot register alias '" + alias +
                "' for name '" + name + "': Circular reference - '" +
                name + "' is a direct or indirect alias for '" + alias + "' already");
    }
}
public boolean hasAlias(String name, String alias) {
    for (Map.Entry<String, String> entry : this.aliasMap.entrySet()) {
        String registeredName = entry.getValue();
        if (registeredName.equals(name)) {
            String registeredAlias = entry.getKey();
            if (registeredAlias.equals(alias) || hasAlias(registeredAlias, alias)) {
                return true;
            }
        }
    }
    return false;
}
```

- 如果 name、alias 分别为 1 和 3 ，则构成 （1,3） 的映射。加入，此时集合中存在 （A,1）、（3,A） 的映射，意味着出现循环指向的情况，则抛出 IllegalStateException 异常。

3. 小结

到这里 BeanDefinition 基于 beanName 和 alias 的维度，都已经注入到缓存中，下一步则是等待初始化使用了。我们，后续的文章，继续搞起来。

文章目录