【死磕 Spring】—— IOC 之 加载 Bean (http://cmsblogs.com/?p=2658)

2018-09-07 分类：死磕 Spring (http://cmsblogs.com/?cat=206) 阅读(25988) 评论(6)

> 原文出自：http://cmsblogs.com (http://cmsblogs.com) 先看一段熟悉的代码：

```
ClassPathResource resource = new ClassPathResource("bean.xml");
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
reader.loadBeanDefinitions(resource);
```

这段代码是 Spring 中编程式使用 IOC 容器，通过这四段简单的代码，我们可以初步判断 IOC 容器的使用过程。

- 获取资源
- 获取 BeanFactory
- 根据新建的 BeanFactory 创建一个BeanDefinitionReader对象，该Reader 对象为资源的解析器
- 装载资源 整个过程就分为三个步骤：资源定位、装载、注册，如下：

(https://gitee.com/chenssy/blog-home/raw/master/image/201811/spring-201805281001.png) * **资源定位**。我们一般用外部资源来描述 Bean 对象，所以在初始化 IOC 容器的第一步就是需要定位这个外部资源。在上一篇博客（【死磕 Spring】----- IOC 之 Spring 统一资源加载策略 (http://cmsblogs.com/?p=2656)）已经详细说明了资源加载的过程。 * **装载**。装载就是 BeanDefinition 的载入。BeanDefinitionReader 读取、解析 Resource 资源，也就是将用户定义的 Bean 表示成 IOC 容器的内部数据结构：BeanDefinition。在 IOC 容器内部维护着一个 BeanDefinition Map 的数据结构，在配置文件中每一个 `<bean>` 都对应着一个 BeanDefinition对象。 * **注册**。向IOC容器注册在第二步解析好的 BeanDefinition，这个过程是通过 BeanDefinitionRegistry 接口来实现的。在 IOC 容器内部其实是将第二个过程解析得到的 BeanDefinition 注入到一个 HashMap 容器中，IOC 容器就是通过这个 HashMap 来维护这些 BeanDefinition 的。在这里需要注意的一点是这个过程并没有完成依赖注入，依赖注册是发生在应用第一次调用 `getBean()` 向容器索要 Bean 时。当然我们可以通过设置预处理，即对某个 Bean 设置 lazyinit 属性，那么这个 Bean 的依赖注入就会在容器初始化的时候完成。 资源定位在前面已经分析了，下面我们直接分析加载，上面提过

`reader.loadBeanDefinitions(resource)` 才是加载资源的真正实现，所以我们直接从该方法入手。


```
public int loadBeanDefinitions(Resource resource) throws BeanDefinitionStoreException {
    return loadBeanDefinitions(new EncodedResource(resource));
}
```

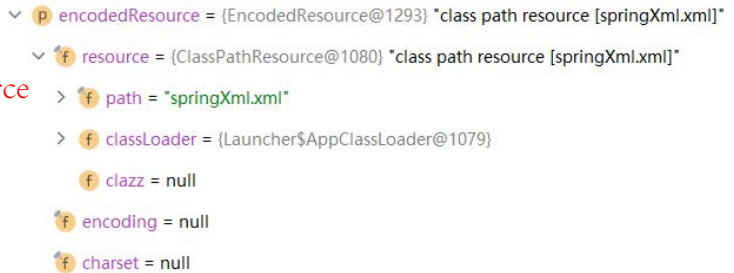
这里resource主要有两个属性1、xxx.xml 2.classloader

从指定的 xml 文件加载 Bean Definition，这里会先对 Resource 资源封装成 EncodedResource。这里为什么需要将 Resource 封装成 EncodedResource呢？主要是为了对 Resource 进行编码，保证内容读取的正确性。封装成 EncodedResource 后，调用

`loadBeanDefinitions()`，这个方法才是真正的逻辑实现。如下：

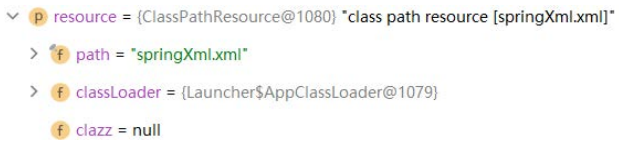
```
public int loadBeanDefinitions(EncodedResource encodedResource) throws BeanDefinitionStoreException {
        Assert.notNull(encodedResource, "EncodedResource must not be null");
        if (logger.isInfoEnabled()) {
            logger.info("Loading XML bean definitions from " + encodedResource.getResource());
        }

        // 获取已经加载过的资源
        Set<EncodedResource> currentResources = this.resourcesCurrentlyBeingLoaded.get();
        if (currentResources == null) {
            currentResources = new HashSet<>(4);
            this.resourcesCurrentlyBeingLoaded.set(currentResources);
        }

        // 将当前资源加入记录中
        if (!currentResources.add(encodedResource)) {
            throw new BeanDefinitionStoreException(
                    "Detected cyclic loading of " + encodedResource + " - check your import definitions!"
);
        }
        try {
            // 从 EncodedResource 获取封装的 Resource 并从 Resource 中获取其中的 InputStream
            InputStream inputStream = encodedResource.getResource().getInputStream();
            try {
                InputSource inputSource = new InputSource(inputStream);
                // 设置编码
                if (encodedResource.getEncoding() != null) {
                    inputSource.setEncoding(encodedResource.getEncoding());
                }
                // 核心逻辑部分
                return doLoadBeanDefinitions(inputSource, encodedResource.getResource());
            }
            finally {
                inputStream.close();
            }
        }
        catch (IOException ex) {
            throw new BeanDefinitionStoreException(
                    "IOException parsing XML document from " + encodedResource.getResource(), ex);
        }
        finally {
            // 从缓存中剔除该资源
            currentResources.remove(encodedResource);
            if (currentResources.isEmpty()) {
                this.resourcesCurrentlyBeingLoaded.remove();
            }
        }
    }
```

资源encodedResource



首先通过

resourcesCurrentlyBeingLoaded.get() 来获取已经加载过的资源，然后将 encodedResource 加入其中，如果 resourcesCurrentlyBeingLoaded 中已经存在该资源，则抛出 BeanDefinitionStoreException 异常。完成后从 encodedResource 获取封装的 Resource 资源并从 Resource 中获取相应的 InputStream ，最后将 InputStream 封装为 InputSource 调用 doLoadBeanDefinitions() 。方法 doLoadBeanDefinitions() 为从 xml 文件中加载 Bean Definition 的真正逻辑，如下: inputSource内部封装了BufferInputStream，以及FileInputStream指向了我们自己的xml文件




```
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
            throws BeanDefinitionStoreException {
        try {
            // 获取 Document 实例
            Document doc = doLoadDocument(inputSource, resource);
            // 根据 Document 实例****注册 Bean信息 xml文件如何读取解析？使用w3c.dom包下的Document对象读取
            return registerBeanDefinitions(doc, resource);
        }
        catch (BeanDefinitionStoreException ex) {
            throw ex;
        }
        catch (SAXParseException ex) {
            throw new XmlBeanDefinitionStoreException(resource.getDescription(),
                    "Line " + ex.getLineNumber() + " in XML document from " + resource + " is invalid", e
x);
        }
        catch (SAXException ex) {
            throw new XmlBeanDefinitionStoreException(resource.getDescription(),
                    "XML document from " + resource + " is invalid", ex);
        }
        catch (ParserConfigurationException ex) {
            throw new BeanDefinitionStoreException(resource.getDescription(),
                    "Parser configuration exception parsing XML from " + resource, ex);
        }
        catch (IOException ex) {
            throw new BeanDefinitionStoreException(resource.getDescription(),
                    "IOException parsing XML document from " + resource, ex);
        }
        catch (Throwable ex) {
            throw new BeanDefinitionStoreException(resource.getDescription(),
                    "Unexpected exception parsing XML document from " + resource, ex);
        }
    }
```

核心部分就是 try 块的两行代码。

1. 调用 doLoadDocument() 方法，根据 xml resource文件获取 Document 实例(w3c.dom包)。
2. 根据获取的 Document 实例注册 Bean 信息。 其实在

doLoadDocument() 方法内部还获取了 xml 文件的验证模式。如下:

```
protected Document doLoadDocument(InputSource inputSource, Resource resource) throws Exception {
    return this.documentLoader.loadDocument(inputSource, getEntityResolver(), this.errorHandler,
            getValidationModeForResource(resource), isNamespaceAware());
}
```

调用

`getValidationModeForResource()` 获取指定资源（xml）的验证模式。所以 `doLoadBeanDefinitions()` 主要就是做了三件事情。 1. 调用 `getValidationModeForResource()` 获取 xml 文件的验证模式 2. 调用 `loadDocument()` 根据 xml 文件获取相应的 Document 实例。 3. 调用 `registerBeanDefinitions()` 注册 Bean 实例。

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 Spring 统一资源加载策略 (http://cmsblogs.com/?p=2656)

下一篇
Refresh your Java skills–面对Java学习过程中的一些迷茫 (http://cmsblogs.com/?p=2660)

- 【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)