



[回到首页](#)

## [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>

[2019-02-07](#)

[Spring](#)

# 【死磕 Spring】—— IoC 之加载 BeanDefinition

本文主要基于 Spring 5.0.6.RELEASE

摘要：原创出处 <http://cmsblogs.com/?p=2658> 「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。

先看一段熟悉的代码：

```
ClassPathResource resource = new ClassPathResource("bean.xml"); // <1>
DefaultListableBeanFactory factory = new DefaultListableBeanFactory(); // <2>
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory); // <3>
reader.loadBeanDefinitions(resource); // <4>
```

这段代码是 Spring 中程式使用 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。

1. 获取资源
2. 获取 BeanFactory
3. 根据新建的 BeanFactory 创建一个 BeanDefinitionReader 对象，该 Reader 对象为资源的解析器
4. 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：



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》](#)）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。
- 本文，我们分享的就是装载这个步骤。

FROM 《Spring 源码深度解析》P16 页  
BeanDefinitionReader，主要定义资源文件读取并转换为 BeanDefinition 的各个功能。

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（Bean 创建），Bean 创建是发生在应用第一次调用 #getBean(...) 方法，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 = false 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。

FROM 老茈茈

简单的说，上面步骤的结果是，XML Resource => XML Document => Bean Definition。

## 1. loadBeanDefinitions

资源定位在前面已经分析了，下面我们直接分析加载，上面看到的 reader.loadBeanDefinitions(resource) 代码，才是加载资源的真正实现，所以我们直接从该方法入手。代码如下：

```
// XmlBeanDefinitionReader.java
@Override
public int loadBeanDefinitions(Resource resource) throws BeanDefinitionStoreException {
    return loadBeanDefinitions(new EncodedResource(resource));
}
```

从指定的 xml 文件加载 Bean Definition，这里会先对 Resource 资源封装成 org.springframework.core.io.support.EncodedResource 对象。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。然后，再调用 #loadBeanDefinitions(EncodedResource encodedResource) 方法，执行真正的逻辑实现。

```
/**
 * 当前线程，正在加载的 EncodedResource 集合。
 */
private final ThreadLocal<Set<EncodedResource>> resourcesCurrentlyBeingLoaded = new NamedThreadLocal<>("XML bean defini
```

```

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

    // <1> 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);
    }
    if (!currentResources.add(encodedResource)) { // 将当前资源加入记录中。如果已存在，抛出异常
        throw new BeanDefinitionStoreException("Detected cyclic loading of " + encodedResource + " - check your import d
    }
    try {
        // <2> 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);
            if (encodedResource.getEncoding() != null) { // 设置编码
                inputSource.setEncoding(encodedResource.getEncoding());
            }
            // 核心逻辑部分，执行加载 BeanDefinition
            return doLoadBeanDefinitions(inputSource, encodedResource.getResource());
        } finally {
            inputStream.close();
        }
    } catch (IOException ex) {
        throw new BeanDefinitionStoreException("IOException parsing XML document from " + encodedResource.getResource());
    } finally {
        // 从缓存中剔除该资源 <3>
        currentResources.remove(encodedResource);
        if (currentResources.isEmpty()) {
            this.resourcesCurrentlyBeingLoaded.remove();
        }
    }
}

```

<1> 处，通过 `resourcesCurrentlyBeingLoaded.get()` 代码，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。

- 为什么需要这么做呢？答案在 “Detected cyclic loading”，避免一个 `EncodedResource` 在加载时，还没加载完成，又加载自身，从而导致死循环。
- 也因此，在 <3> 处，当一个 `EncodedResource` 加载完成后，需要从缓存中剔除。

<2> 处理，从 `encodedResource` 获取封装的 `Resource` 资源，并从 `Resource` 中获取相应的 `InputStream`，然后将 `InputStream` 封装为 `InputSource`，最后调用

`#doLoadBeanDefinitions(InputSource inputSource, Resource resource)` 方法，执行加载 `Bean Definition` 的真正逻辑。

## 2. doLoadBeanDefinitions

```

/**
 * Actually load bean definitions from the specified XML file.
 * @param inputSource the SAX InputSource to read from

```

```

* @param resource the resource descriptor for the XML file
* @return the number of bean definitions found
* @throws BeanDefinitionStoreException in case of loading or parsing errors
* @see #doLoadDocument
* @see #registerBeanDefinitions
*/
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
    throws BeanDefinitionStoreException {
try {
    // <1> 获取 XML Document 实例
    Document doc = doLoadDocument(inputSource, resource);
    // <2> 根据 Document 实例, 注册 Bean 信息
    int count = registerBeanDefinitions(doc, resource);
    if (logger.isDebugEnabled()) {
        logger.debug("Loaded " + count + " bean definitions from " + resource);
    }
    return count;
} catch (BeanDefinitionStoreException ex) {
    throw ex;
} catch (SAXParseException ex) {
    throw new XmlBeanDefinitionStoreException(resource.getDescription(),
        "Line " + ex.getLineNumber() + " in XML document from " + resource + " is invalid", ex);
} 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);
}
}

```

在 <1> 处, 调用 `#doLoadDocument(InputSource inputSource, Resource resource)` 方法, 根据 xml 文件, 获取 Document 实例。

在 <2> 处, 调用 `#registerBeanDefinitions(Document doc, Resource resource)` 方法, 根据获取的 Document 实例, 注册 Bean 信息。

## 2.1 doLoadDocument

```

/**
* 获取 XML Document 实例
*
* Actually load the specified document using the configured DocumentLoader.
* @param inputSource the SAX InputSource to read from
* @param resource the resource descriptor for the XML file
* @return the DOM Document
* @throws Exception when thrown from the DocumentLoader
* @see #setDocumentLoader
* @see DocumentLoader#loadDocument
*/
protected Document doLoadDocument(InputSource inputSource, Resource resource) throws Exception {

```

```
return this.documentLoader.loadDocument(inputSource, getEntityResolver(), this.errorHandler,
    getValidationModeForResource(resource), isNamespaceAware());
}
```

1. 调用 `#getValidationModeForResource(Resource resource)` 方法，获取指定资源（xml）的验证模式。详细解析，见 [《【死磕 Spring】—— IoC 之获取验证模型》](#)。
2. 调用 `DocumentLoader#loadDocument(InputSource inputSource, EntityResolver entityResolver, ErrorHandler errorHandler, int validationMode, boolean namespaceAware)` 方法，获取 XML Document 实例。详细解析，见 [《【死磕 Spring】—— IoC 之获取 Document 对象》](#)。

## 2.2 registerBeanDefinitions

该方法的详细解析，见 [《【死磕 Spring】—— IoC 之注册 BeanDefinition》](#)。

## 666. 彩蛋

本文未完，需要继续阅读后续几篇文章。

文章目录

1. [1. 1. loadBeanDefinitions](#)
2. [2. 2. doLoadBeanDefinitions](#)
  1. [2.1. 2.1 doLoadDocument](#)
  2. [2.2. 2.2 registerBeanDefinitions](#)
3. [3. 666. 彩蛋](#)

2014 - 2023 芋道源码 |  
总访客数 次 && 总访问量 次  
[回到首页](#)