【死磕 Spring】—— IOC 之 IOC 初始化总结 (http://cmsblogs.com/?p=2790)

2018-10-09 分类：死磕 Spring (http://cmsblogs.com/?cat=206) 阅读(12867) 评论(5)

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

前面 13 篇博文从源码层次分析了 IOC 整个初始化过程，这篇就这些内容做一个总结将其连贯起来。

在前文提过，IOC 容器的初始化过程分为三步骤：Resource 定位、BeanDefinition 的载入和解析，BeanDefinition 注册。

(https://gitee.com/chenssy/blog-home/raw/master/image/201811/spring-201805281001.png)

- **Resource 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。
- **BeanDefinition 的载入和解析。**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都对应着一个 BeanDefinition对象。
- **BeanDefinition 注册。**向 IOC 容器注册在第二步解析好的 BeanDefinition，这个过程是通过 BeanDefinitionRegistery 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。

还记得在博客【死磕 Spring】----- IOC 之 加载 Bean (http://cmsblogs.com/?p=2658) 中提供的一段代码吗？这里我们同样也以这段代码作为我们研究 IOC 初始化过程的开端，如下：

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

刚刚开始的时候可能对上面这几行代码不知道什么意思，现在应该就一目了然了。

- `ClassPathResource resource = new ClassPathResource("bean.xml");`：根据 Xml 配置文件创建 Resource 资源对象。ClassPathResource 是 Resource 接口的子类，bean.xml 文件中的内容是我们定义的 Bean 信息。
- `DefaultListableBeanFactory factory = new DefaultListableBeanFactory();` 创建一个 BeanFactory。DefaultListableBeanFactory 是 BeanFactory 的一个子类，BeanFactory 作为一个接口，其实它本身是不具有独立使用的功能的，而 DefaultListableBeanFactory 则是真正可以独立使用的 IOC 容器，它是整个 Spring IOC 的始祖，在后续会有专门的文章来分析它。
- `XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);`：创建 XmlBeanDefinitionReader 读取器，用于载入 BeanDefinition 。
- `reader.loadBeanDefinitions(resource);`：开启 Bean 的载入和注册进程，完成后的 Bean 放置在 IOC 容器中。

Resource 定位

Spring 为了解决资源定位的问题，提供了两个接口：Resource、ResourceLoader，其中 Resource 接口是 Spring 统一资源的抽象接口，ResourceLoader 则是 Spring 资源加载的统一抽象。关于Resource、ResourceLoader 的更多知识请关注【死磕 Spring】----- IOC 之 Spring 统一资源加载策略 (http://cmsblogs.com/?p=2656)

Resource 资源的定位需要 Resource 和 ResourceLoader 两个接口互相配合，在上面那段代码中 `new ClassPathResource("bean.xml")` 为我们定义了资源，那么 ResourceLoader 则是在什么时候初始化的呢？看 XmlBeanDefinitionReader 构造方法：

```
public XmlBeanDefinitionReader(BeanDefinitionRegistry registry) {
    super(registry);
}
```

直接调用父类 AbstractBeanDefinitionReader：

```
protected AbstractBeanDefinitionReader(BeanDefinitionRegistry registry) {
    Assert.notNull(registry, "BeanDefinitionRegistry must not be null");
    this.registry = registry;

    // Determine ResourceLoader to use.
    if (this.registry instanceof ResourceLoader) {
        this.resourceLoader = (ResourceLoader) this.registry;
    }
    else {
        this.resourceLoader = new PathMatchingResourcePatternResolver();
    }

    // Inherit Environment if possible
    if (this.registry instanceof EnvironmentCapable) {
        this.environment = ((EnvironmentCapable) this.registry).getEnvironment();
    }
    else {
        this.environment = new StandardEnvironment();
    }
}
```

核心在于设置 resourceLoader 这段，如果设置了 ResourceLoader 则用设置的，否则使用 PathMatchingResourcePatternResolver，该类是一个集大成者的 ResourceLoader。

BeanDefinition 的载入和解析

`reader.loadBeanDefinitions(resource);` 开启 BeanDefinition 的解析过程。如下：

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

在这个方法会将资源 resource 包装成一个 EncodedResource 实例对象，然后调用 `loadBeanDefinitions()` 方法，而将 Resource 封装成 EncodedResource 主要是为了对 Resource 进行编码，保证内容读取的正确性。

```
public int loadBeanDefinitions(EncodedResource encodedResource) throws BeanDefinitionStoreException {
    // 省略一些代码
    try {
        // 将资源文件转为 InputStream 的 IO 流
        InputStream inputStream = encodedResource.getResource().getInputStream();
        try {
            // 从 InputStream 中得到 XML 的解析源
            InputSource inputSource = new InputSource(inputStream);
            if (encodedResource.getEncoding() != null) {
                inputSource.setEncoding(encodedResource.getEncoding());
            }

            // 具体的读取过程
            return doLoadBeanDefinitions(inputSource, encodedResource.getResource());
        }
        finally {
            inputStream.close();
        }
    }
    // 省略一些代码
}
```

从 encodedResource 源中获取 xml 的解析源，调用 `doLoadBeanDefinitions()` 执行具体的解析过程。

```
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
        throws BeanDefinitionStoreException {
    try {
        Document doc = doLoadDocument(inputSource, resource);
        return registerBeanDefinitions(doc, resource);
    }
    // 省略很多catch代码
```

在该方法中主要做两件事：1、根据 xml 解析源获取相应的 Document 对象，2、调用 `registerBeanDefinitions()` 开启 BeanDefinition 的解析注册过程。

转换为 Document 对象

调用 `doLoadDocument()` 会将 Bean 定义的资源转换为 Document 对象。

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

`loadDocument()` 方法接受五个参数：

- **inputSource**：加载 Document 的 Resource 源
- **entityResolver**：解析文件的解析器

- **errorHandler**：处理加载 Document 对象的过程的错误
- **validationMode**：验证模式
- **namespaceAware**：命名空间支持。如果要提供对 XML 名称空间的支持，则为true

对于这五个参数，有两个参数需要重点关注下：entityResolver、validationMode。这两个参数分别在 【死磕Spring】----- IOC 之 获取 Document 对象 (http://cmsblogs.com/?p=2695)、【死磕Spring】----- IOC 之 获取验证模型 (http://cmsblogs.com/?p=2688) 中有详细的讲述。

`loadDocument()` 在类 DefaultDocumentLoader 中提供了实现，如下：

```
public Document loadDocument(InputSource inputSource, EntityResolver entityResolver,
        ErrorHandler errorHandler, int validationMode, boolean namespaceAware) throws Exception {
  // 创建文件解析工厂
    DocumentBuilderFactory factory = createDocumentBuilderFactory(validationMode, namespaceAware);
    if (logger.isDebugEnabled()) {
        logger.debug("Using JAXP provider [" + factory.getClass().getName() + "]");
    }
    // 创建文档解析器
    DocumentBuilder builder = createDocumentBuilder(factory, entityResolver, errorHandler);
    // 解析 Spring 的 Bean 定义资源
    return builder.parse(inputSource);
}
```

这到这里，就已经将定义的 Bean 资源文件，载入并转换为 Document 对象了，那么下一步就是如何将其解析为 Spring IOC 管理的 Bean 对象并将其注册到容器中。这个过程有方法 `registerBeanDefinitions()` 实现。如下：

```
public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreExcepti
on {
    // 创建 BeanDefinitionDocumentReader 来对 xml 格式的BeanDefinition 解析
    BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();
    // 获得容器中注册的Bean数量
    int countBefore = getRegistry().getBeanDefinitionCount();

    // 解析过程入口，这里使用了委派模式，BeanDefinitionDocumentReader只是个接口，
    // 具体的解析实现过程有实现类DefaultBeanDefinitionDocumentReader完成
    documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
    return getRegistry().getBeanDefinitionCount() - countBefore;
}
```

首先创建 BeanDefinition 的解析器 BeanDefinitionDocumentReader，然后调用 `documentReader.registerBeanDefinitions()` 开启解析过程，这里使用的是委派模式，具体的实现由子类 DefaultBeanDefinitionDocumentReader 完成。

```
public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {
   // 获得XML描述符
    this.readerContext = readerContext;
    logger.debug("Loading bean definitions");

    // 获得Document的根元素
    Element root = doc.getDocumentElement();

    // 解析根元素
    doRegisterBeanDefinitions(root);
}
```

对 Document 对象的解析

从 Document 对象中获取根元素 root，然后调用 `doRegisterBeanDefinitions()` 开启真正的解析过程。

```
protected void doRegisterBeanDefinitions(Element root) {
    BeanDefinitionParserDelegate parent = this.delegate;
    this.delegate = createDelegate(getReaderContext(), root, parent);

   // 省略部分代码

    preProcessXml(root);
    parseBeanDefinitions(root, this.delegate);
    postProcessXml(root);

    this.delegate = parent;
}
```

`preProcessXml()` 、 `postProcessXml()` 为前置、后置增强处理，目前 Spring 中都是空实现，`parseBeanDefinitions()` 是对根元素 root 的解析注册过程。

```
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
   // Bean定义的Document对象使用了Spring默认的XML命名空间
    if (delegate.isDefaultNamespace(root)) {
      // 获取Bean定义的Document对象根元素的所有子节点
        NodeList nl = root.getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            // 获得Document节点是XML元素节点
            if (node instanceof Element) {
                Element ele = (Element) node;
                // Bean定义的Document的元素节点使用的是Spring默认的XML命名空间
                if (delegate.isDefaultNamespace(ele)) {
                   // 使用Spring的Bean规则解析元素节点（默认解析规则）
                    parseDefaultElement(ele, delegate);
                }
                else {
                   // 没有使用Spring默认的XML命名空间，则使用用户自定义的解析规则解析元素节点
                    delegate.parseCustomElement(ele);
                }
            }
        }
    }
    else {
      // Document 的根节点没有使用Spring默认的命名空间，则使用用户自定义的解析规则解析
        delegate.parseCustomElement(root);
    }
}
```

迭代 root 元素的所有子节点，对其进行判断，若节点为默认命名空间，则ID调用 `parseDefaultElement()` 开启默认标签的解析注册过程，否则调用 `parseCustomElement()` 开启自定义标签的解析注册过程。

标签解析

若定义的元素节点使用的是 Spring 默认命名空间，则调用 `parseDefaultElement()` 进行默认标签解析，如下：

```
private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {
   // 如果元素节点是<Import>导入元素，进行导入解析
    if (delegate.nodeNameEquals(ele, IMPORT_ELEMENT)) {
        importBeanDefinitionResource(ele);
    }
    // 如果元素节点是<Alias>别名元素，进行别名解析
    else if (delegate.nodeNameEquals(ele, ALIAS_ELEMENT)) {
        processAliasRegistration(ele);
    }
    // 如果元素节点<Bean>元素，则进行Bean解析注册
    else if (delegate.nodeNameEquals(ele, BEAN_ELEMENT)) {
        processBeanDefinition(ele, delegate);
    }

    // // 如果元素节点<Beans>元素，则进行Beans解析
    else if (delegate.nodeNameEquals(ele, NESTED_BEANS_ELEMENT)) {
        // recurse
        doRegisterBeanDefinitions(ele);
    }
}
```

对四大标签：import、alias、bean、beans 进行解析，其中 bean 标签的解析为核心工作。关于各个标签的解析过程见如下文章：

- 【死磕Spring】----- IOC 之解析Bean：解析 import 标签 (http://cmsblogs.com/?p=2724)
- 【死磕 Spring】—— IOC 之解析 bean 标签：开启解析进程 (http://cmsblogs.com/?p=2731)
- 【死磕 Spring】—— IOC 之解析 bean 标签：BeanDefinition） (http://cmsblogs.com/?p=2734)
- 【死磕 Spring】—— IOC 之解析 bean 标签：meta、lookup-method、replace-method） (http://cmsblogs.com/?p=2736)
- 【死磕 Spring】—— IOC 之解析 bean 标签：constructor-arg、property 子元素 (http://cmsblogs.com/?p=2754)
- 【死磕 Spring】—— IOC 之解析 bean 标签：解析自定义标签 (http://cmsblogs.com/?p=2756)

对于默认标签则由 `parseCustomElement()` 负责解析。

```
    public BeanDefinition parseCustomElement(Element ele) {
        return parseCustomElement(ele, null);
    }

    public BeanDefinition parseCustomElement(Element ele, @Nullable BeanDefinition containingBd) {
        String namespaceUri = getNamespaceURI(ele);
        if (namespaceUri == null) {
            return null;
        }
         NamespaceHandler handler = this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri
);
        if (handler == null) {
             error("Unable to locate Spring NamespaceHandler for XML schema namespace [" + namespaceUri +
"]", ele);
            return null;
        }
        return handler.parse(ele, new ParserContext(this.readerContext, this, containingBd));
    }
```

获取节点的 namespaceUri，然后根据该 namespaceuri 获取相对应的 Handler，调用 Handler 的 `parse()` 方法即完成自定义标签的解析和注入。想了解更多参考：【死磕Spring】----- IOC 之解析自定义标签 (http://cmsblogs.com/?p=2756)

注册 BeanDefinition

经过上面的解析，则将 Document 对象里面的 Bean 标签解析成了一个个的 BeanDefinition ，下一步则是将这些 BeanDefinition 注册到 IOC 容器中。动作的触发是在解析 Bean 标签完成后，如下：

DefaultBeanDefinitionDocumentReader类的processBeanDefinition方法

```
    protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {
        BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
        if (bdHolder != null) {
            bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
            try {
                // Register the final decorated instance. //使用DefualtListableBeanFactory具体注册到hashmap中
                 BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext().getRegistry
());
            }
            catch (BeanDefinitionStoreException ex) {
                getReaderContext().error("Failed to register bean definition with name '" +
                        bdHolder.getBeanName() + "'", ele, ex);
            }
            // Send registration event.
            getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));
        }
    }
```

调用 BeanDefinitionReaderUtils.registerBeanDefinition() 注册，其实这里面也是调用 BeanDefinitionRegistry 的 registerBeanDefinition() 来注册 BeanDefinition，不过最终的实现是在 DefaultListableBeanFactory 中实现，如下：

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

  // 省略一堆校验

    BeanDefinition oldBeanDefinition;

    oldBeanDefinition = this.beanDefinitionMap.get(beanName);
      // 省略一堆 if
        this.beanDefinitionMap.put(beanName, beanDefinition);
    }
    else {
        if (hasBeanCreationStarted()) {
            // Cannot modify startup-time collection elements anymore (for stable iteration)
            synchronized (this.beanDefinitionMap) {
                this.beanDefinitionMap.put(beanName, beanDefinition);
                List<String> updatedDefinitions = new ArrayList<>(this.beanDefinitionNames.size() + 1
);
                updatedDefinitions.addAll(this.beanDefinitionNames);
                updatedDefinitions.add(beanName);
                this.beanDefinitionNames = updatedDefinitions;
                if (this.manualSingletonNames.contains(beanName)) {
                    Set<String> updatedSingletons = new LinkedHashSet<>(this.manualSingletonNames);
                    updatedSingletons.remove(beanName);
                    this.manualSingletonNames = updatedSingletons;
                }
            }
        }
        else {
            // Still in startup registration phase
            this.beanDefinitionMap.put(beanName, beanDefinition);
            this.beanDefinitionNames.add(beanName);
            this.manualSingletonNames.remove(beanName);
        }
        this.frozenBeanDefinitionNames = null;
    }

    if (oldBeanDefinition != null || containsSingleton(beanName)) {
        resetBeanDefinition(beanName);
    }
}
```

这段代码最核心的部分是这句 this.beanDefinitionMap.put(beanName, beanDefinition) ，所以注册过程也不是那么的高大上，就是利用一个 Map 的集合对象来存放，key 是 beanName，value 是 BeanDefinition。