抄书规范

字体3px大小(不设置)，默认无颜色，重点内容红色，不加粗

Spring源码深度解析

第1章 Spring整体架构和环境搭

Spring是于2003年兴起的一个轻量级的Java开源框架由Rod Johnson在其著作《Expert One-On-One J2EE Development and Design》中阐述的部分理念和原型衍生而米。Spring 是为解决企业应用开发的复杂性而创建的，它使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring 的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从 Spring 中受益。

1.1 Spring的整体架构

Spring框架是一个分层架构,它包含一系列的功能要素，并被分为大约20个模块，如图1-1所示。

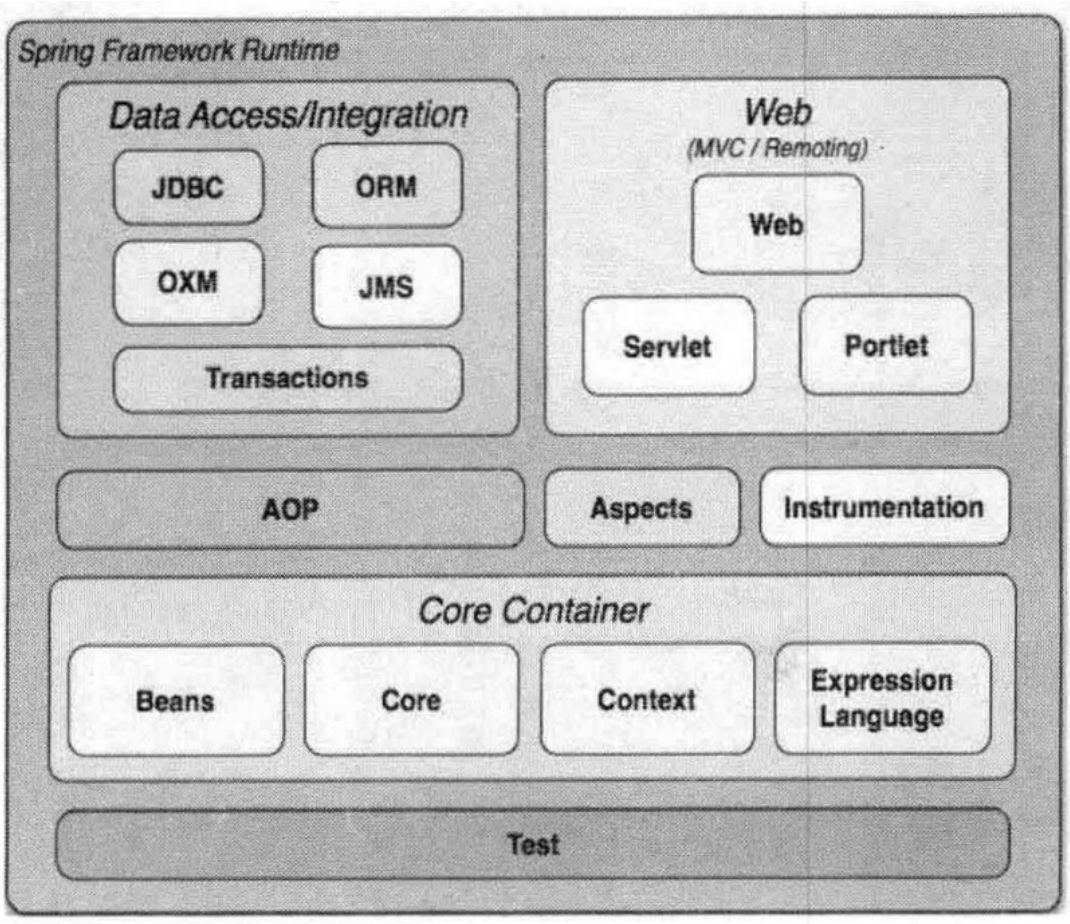


图 1-1　Spring 整体架构图

这些模块被总结为以下几部分。

(1) Core Container.

Core Container(核心容器）包含有Core、Beans、Context 和 Expression Language模块。

Core和Beans模块是框架的基础部分，提供 IoC(反转控制）和依赖注入特性。这里的基础概念是BeanFactory.,它提供对Factory模式的经典实现来消除对程序性单例模式的需要，并真正地允许你从程序逻辑中分离出依赖关系和配置。

- Core模块主要包含Spring框架基本的核心工具类，Spring 的其他组件要都要使用到这个包里的类，Core模块是其他组件的基本核心。当然你也可以在自己的应用系统中使用这些工具类。
- Beans模块是所有应用都要用到的，它包含访问配置文件、创建和管理bean以及进行Inversion of Control/ Dependency Injection (loC/DI)操作相关的所有类。
- Context模块构建于Core和Beans模块基础之上,提供了一种类似于JNDI注册器的框架式的对象访问方法。Context模块继承了Beans的特性，为Spring核心提供了大量扩展，添加了对国际化(例如资源绑定)、事件传播资源加载和对Context 的透明创建的支持。Context模块同时也支持J2EE的一些特性，例如EJB、JMX和基础的远程处理。ApplicationContext接口是 Context模块的关键。
- Expression Language模块提供了一个强大的表达式语言用于在运行时查询和操纵对象。它是JSP 2.1规范中定义的unifed expression language的一个扩展。该语言支持设置/获取属性的值，属性的分配，方法的调用，访问数组上下文(accessiong the contextof arrays)、容器和索引器、逻辑和算术运算符、命名变量以及从.Spring 的 loC容器中根据名称检索对象。它也支持list投影、选择和一般的list 聚合。

(2) Data Access/Integration。

Data Access/Integration层包含有JDBC、ORM、OXM、JMS 和 Transaction模块，其中:

- JDBC模块提供了一个JDBC抽象层，它可以消除冗长的JDBC编码和解析数据库厂商特有的错误代码。这个模块包含了Spring对JDBC 数据访问进行封装的所有类。
- ORM模块为流行的对象-关系映射API，如JPA、JDO、Hibernate、iBatis等，提供了一个交互层。利用ORM封装包，可以混合使用所有Spring 提供的特性进行O/R映射。如前边提到的简单声明性事物管理。

Spring框架插入了若干个ORM框架，从而提供了ORM的对象关系工具，其中包括JDO、Hibernate和 iBatisSQL Map。所有这些都遵从Spring的通用事务和DAO异常层次结构

- OXM模块提供了一个对Object/XML映射实现的抽象层Object/XML映射实现包括JAXB、Castor、XMLBeans、JiBX和XStream。
- JMS (Java Messaging Service)模块主要包含了一些制造和消费消息的特性。
- Transaction模块支持编程和声明性的事物管理，这些事物类必须实现特定的接口，并且对所有的POJO都适用。

(3 ） Web。

Web上下文模块建立在应用程序上下文模块之上,为基于Web的应用程序提供了上下文。所以，Spring框架支持与Jakarta Struts的集成。Web模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。Web层包含了Web、Web-Servlet、Web-Struts和 Web-Porlet模块，具体说明如下。

- Web模块:提供了基础的面向Web的集成特性。例如,多文件上传、使用servlet listeners初始化 IoC 容器以及一个面向Web的应用上下文。它还包含Spring 远程支持中 Web的相关部分。
- Web-Servlet模块web.servlet.jar:该模块包含Spring 的model-view-controlle(MVC)实现。Spring的MVC框架使得模型范围内的代码和web forms之间能够清楚地分离开来，并与Spring框架的其他特性集成在一起。
- Web-Struts模块:该模块提供了对Struts的支持，使得类在Spring应用中能够与一个典型的Struts Web 层集成在一起。注意，该支持在Spring 3.0中是deprecated的。
- Web-Porlet模块:提供了用于Portlet环境和 Web-Servlet模块的MVC 的实现。

(4) AOP。

AOP模块提供了一个符合AOP联盟标准的面向切面编程的实现，它让你可以定义例如方法拦截器和切点，从而将逻辑代码分开，降低它们之间的耦合性。利用 source-level 的元数据功能，还可以将各种行为信息合并到你的代码中，这有点像.Net技术中的 attribute概念。
通过配置管理特性，Spring AOP模块直接将面向切面的编程功能集成到了Spring框架中,所以可以很容易地使Spring框架管理的任何对象支持AOP。Spring AOP模块为基于Spring 的应用程序中的对象提供了事务管理服务。通过使用Spring AOP，不用依赖EJB组件就可以将声明性事务管理集成到应用程序中。

- Aspects模块提供了对AspectJ的集成支持。

Instrumentation模块提供了class instrumentation支持和classloader实现,使得可以在特定的应用服务器上使用。
(5)Test
Test模块支持使用JUnit'和 TestNG对Spring 组件进行测试。

1.2 环境搭建

要构建Spring源码环境首先要安装GitHub以及Gradle。

1.2.1 安装GitHub

首先读者需要到GitHub官网去下载安装包。

1.2.2 安装 Gradle

Gradle是一个基于Groovy的构建工具，它使用Groovy来编写构建脚本，支持依赖管理和多项目创建，类似 Maven，但比其更加简单轻便。Gradle为 Ivy 提供了一个 layer，提供了build-by-convention集成，而且它还让你获得许多类似 Maven的功能。你可以从 http:/www.gradle.org/downloads页面下载 Gradle，下载后将文件解压放到指定目录中（笔者放在了C:\Program Files目录下），然后开始进行环境变量的配置。
(1根据对应目录创建GRADLE_HOME系统变量。

(2)将系统变量加人到path中。

(3)测试

```
gradle -version //安装成功
```

1.2.3 下载源码

```
git clone git://github.com/SpringSource/Spring-framework.git
```

第2章 容器的基本实现

2.1 容器基本用法

bean是Spring 中最核心的东西，spring就像个大水桶，bean是里面的水，水桶脱离了水便也没什么用处了，那么我们先看看bean 的定义。

```
public class MyTestBean {
    private string teststr = "teststr";
    public string getTeststr() {
        return testStr;
        public void setTeststr(String teststr){
            this.testStr = testStr;
        }
    }
}
```

很普通，bean没有任何特别之处，的确，Spring 的目的就是让我们的bean能成粹的POJO，这也是Spring所追求的。接下来看看配置文件:

```
<?xml version=" 1.0" encoding="UTF-8"?>
<beans xmins="http://www.Springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/xMLSchema-instance"
xsi:schemaLocation="http://ww.Springframework.org/schema/beans/http://www.Spring
framework.org/schema/beans/Spring-beans.xsd">
    <bean id="myTestBean" class="bean.MyTestBean"/ >
</beans>
```

在上面的配置中我们看到了bean的声明方式,尽管Spring 中 bean的元素定义着N种属性来支撑我们业务的各种应用，但是我们只要声明成这样，基本上就已经可以满足我们的大多数应用了。好了，你可能觉得还有什么，但是，真没了，Spring的入门示例到这里已经结束，我们可以写测试代码测试了。

```
@suppresswarnings("deprecation")
public class BeanFactoryTest {
    @Test
    public void testsimpleLoad() {
        BeanFactorybf = new XmlBeanFactory(new
ClassPathResource("beanFactoryTest.xml");
        MyTestBean bean = (MyTestBean) bf.getBean("myTestBean");
        assertEquals("testStr", bean.getTestStr(O));
    }
}
```

相信聪明的读者会很快看到我们期望的结果:在 Eclipse中显示了 Green Baro直接使用BeanFactory作为容器对于Spring的使用来说并不多见,甚至是甚少使用,因为
在企业级的应用中大多数都会使用的是ApplicationContext，后续章节我们会介绍它们之间的区别，这里只是用于测试，让读者更快更好地分析Spring 的内部原理。

2.2 功能分析

现在我们可以来好好分析一下上面测试代码的功能,来探索上面的测试代码中Spring究竟帮助我们完成了什么工作?不管之前你是否使用过Spring,当然，你应该使用过的，毕竟本书面用的是对Spring有一定使用经验的读者，你都应该能猜出来，这段测试代码完成的功能无非就是以下几点

(1）读取配置文件beanFactoryTest.xml。
(2）根据beanFactoryTest.xml中的配置找到对应的类的配置,并实例化。
(3）调用实例化后的实例。
为了更清楚地描述,笔者临时画了设计类图，如图2-1所示,如果想完成我们预想的功能,至少需要3个类。

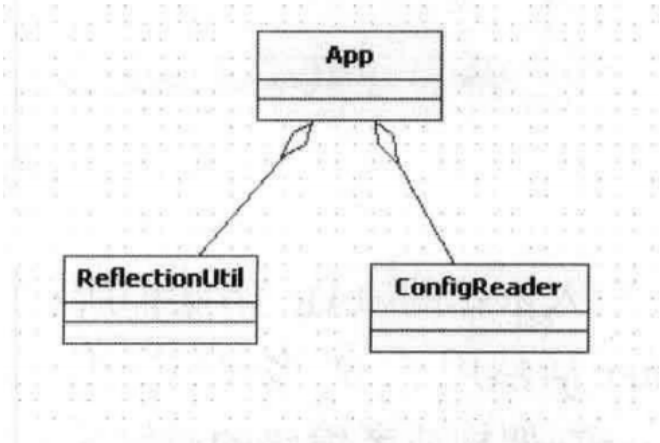


图 2-1　最简单的 Spring 功能架构

- ConfigReader: 用于读取及验证配置文件。我们要用配置文件里面的东西，当然首先要做的就是读取，然后放置在内存中。
- ReflectionUtil:用于根据配置文件中的配置进行反射实例化。比如在上例中beanFactoryTest.xml出现的，我们就可以根据bean.MyTestBean进行实例化。
- App:用于完成整个逻辑的串联。
  按照最原始的思维方式，整个过程无非如此，但是作为一个风靡世界的优秀源码真的就这么简单吗?

2.3 工程搭建

不如我们首先大致看看Spring 的源码。在 Spring源码中，用于实现上面功能的是org.Springframework.beans.jar，我们看源码的时候要打开这个
能，那就没有必要引入Spring 的其他更多的包，当然Core 是必须的，还有些依赖包如下。

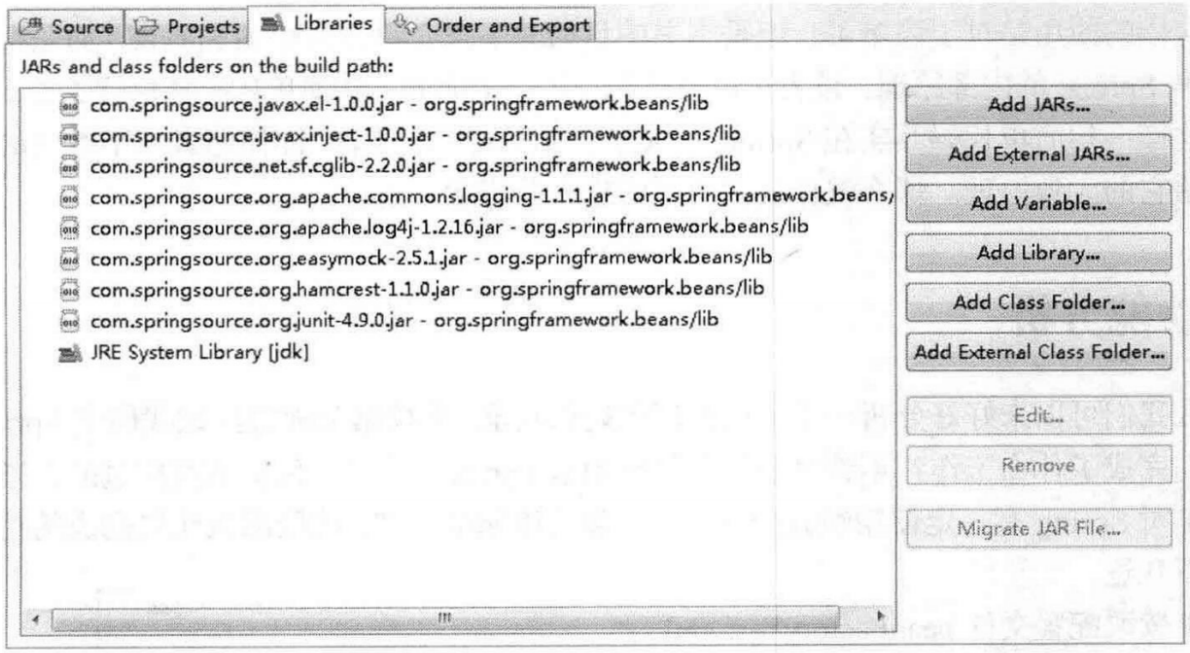


图 2-2　Spring 测试类依赖的 JAR

引人依赖的 JAR 消除掉所有编译错误后，终于可以看源码了。或许你已经知道了答案，Spring居然用了N多代码实现了这个看似很简单的功能，那么这些代码都是做什么用的呢?Spring在架构或者编码的时候又是如何考虑的呢?带着疑问我们踏上了研读 Spring源码的征程。

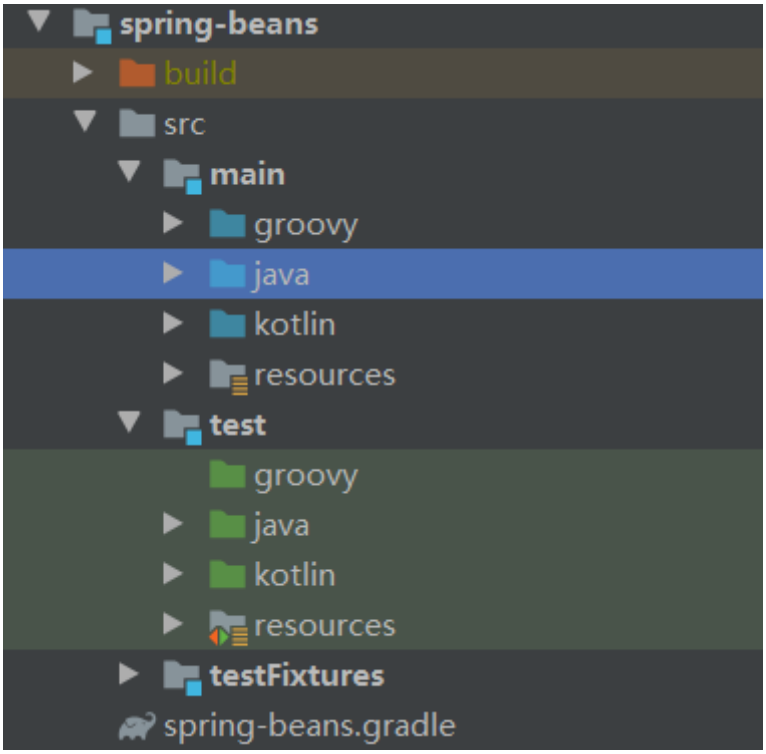
2.4 Spring的结构组成

我们首先尝试梳理一下Spring 的框架结构，从全局的角度了解一下Spring 的结构组成。

2.4.1 beans包的层级结构

笔者认为阅读源码的最好方法是通过示例跟着操作一遍，虽然有时候或者说大多数时候会被复杂的代码绕来绕去，绕到最后已经不知道自己身在何处了，但是,如果配以UML还是可以搞定的。笔者就是按照自己的思路进行分析,并配合必要的UML,希望读者同样可以跟得上思路。我们先看看整个beans工程的源码结构，如图2-3所示。
beans包中的各个源码包的功能如下。

- src/main/java 用于展现 Spring 的主要逻辑。
- src/main/resources用于存放系统的配置文件。
- src/test/java用于对主要逻辑进行单元测试。
- src/test/resources用于存放测试用的配置文件。

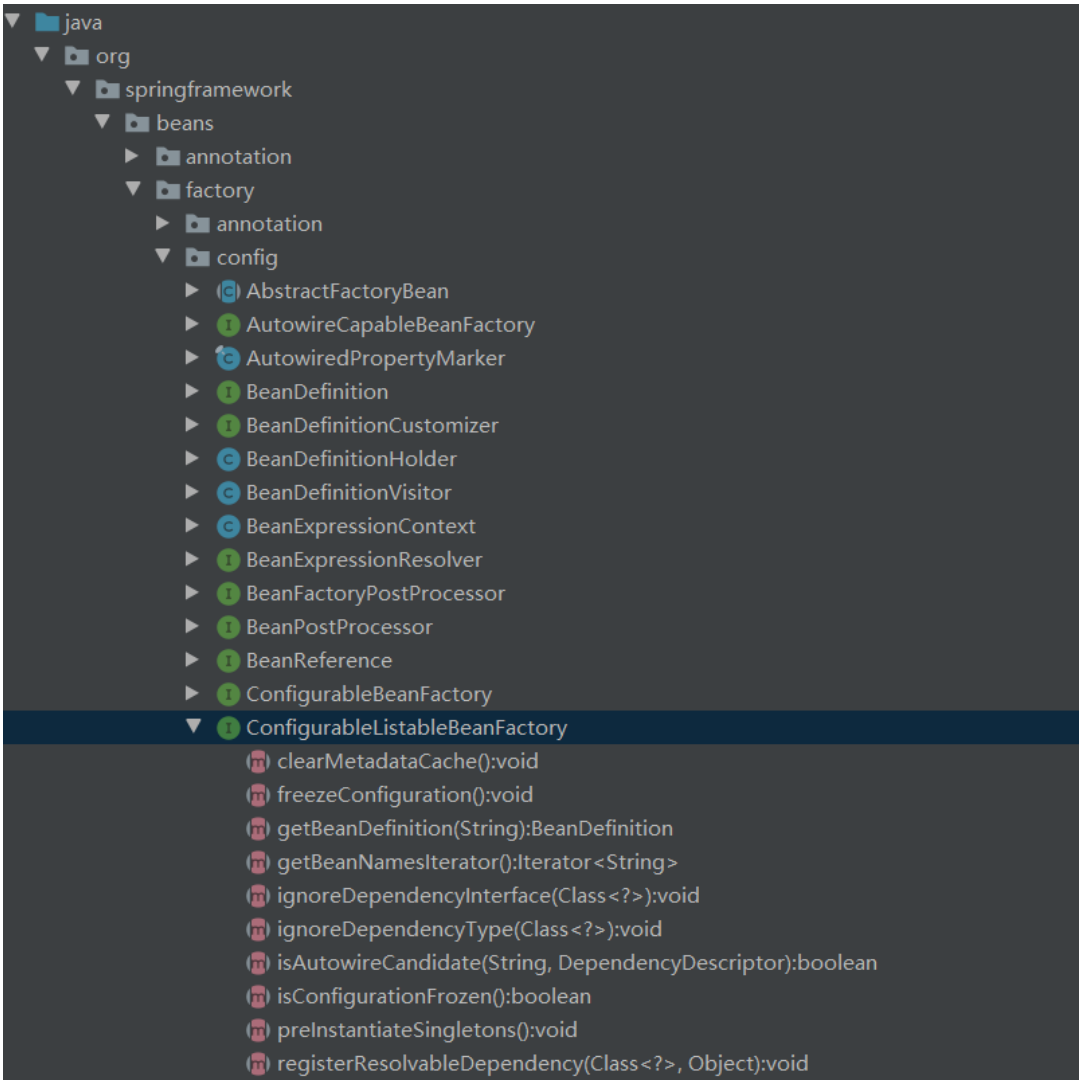


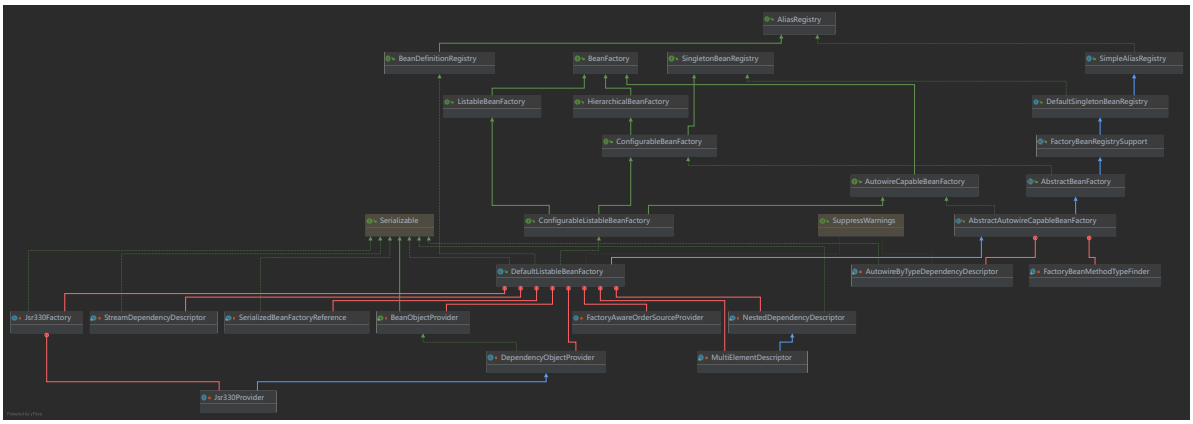
2.4.2 核心类介绍

通过beans工程的结构介绍，我们现在对beans 的工程结构有了初步的认识，但是在正式开始源码分析之前,有必要了解一下Spring 中最核心的两个类。

1.DefaultListableBeanFactory

XmlBeanFactory继承自DefaultListableBeanFactory，而DefaultListableBeanFactory是整个bean加载的核心部分，是Spring注册及加载bean的默认实现，而对于XmlBeanFactory与DefaultListableBeanFactory 不同的地方其实是在XmlBeanFactory 中使用了自定义的XML 读取器XmlBeanDefinitionReader，实现了个性化的 BeanDefinitionReader读取，DefaultListableBeanFactory继承了AbstractAutowireCapableBeanFactory 并实现ConfigurableListableBeanfactory 以从BeanDefinitionRegistry接口。以下是ConfigurableListableBeanFactory 的层次结构图见图（2-4）以及相关类图（见图2-5 ）。





从上面的类图以及层次结构图中，我们可以很清晰地从全局角度了解DefaultListableBeanFactory的脉络。如果读者没有了解过Spring源码可能对上面的类图不是很理解，不过没关系，

通过后续的学习，你会逐渐了解每个类的作用。那么，让我们先简单地了解一下上面类图中的各个类的作用。

- AliasRegistry:定义对alias的简单增删改等操作。
- SimpleAliasRegistry:主要使用map作为alias 的缓存，并对接口AliasRegistry进行实现。
- SingletonBeanRegistry:定义对单例的注册及获取。
- BeanFactory:定义获取bean 及 bean的各种属性。
- DefaultSingletonBeanRegistry:对接口 SingletonBeanRegistry各函数的实现。
- HierarchicalBeanFactory:继承BeanFactory，也就是在BeanFactory定义的功能的基础上增加了对parentFactory 的支持。
- BeanDefinitionRegistry:定义对 BeanDefinition的各种增删改操作。

- FactoryBeanRegistrySupport:在DefaultSingletonBeanRegistry基础上增加了对 FactoryBean的特殊处理功能。
- ConfigurableListableBeanFactory: BeanFactory配置清单，指定忽略类型及接口等
- DefaultListableBeanFactory:综合上面所有功能，主要是对Bean注册后的处理.

XmlBeanFactory对 DefaultListableBeanFactory类进行了扩展，主要用于从XML文档中读取BeanDefinition，对于注册及获取 Bean都是使用从父类DefaultListableBeanFactory继承的方法去实现，而唯独与父类不同的个性化实现就是增加了XmlBeanDefinitionReader类型的reader属性。在XmlBeanFactory 中主要使用reader属性对资源文件进行读取和注册。

2.XmlBeanDefinitionReader

XML配置文件的读取是Spring中重要的功能，因为Spring 的大部分功能都是以配置作为切人点的,那么我们可以从XmlBeanDefinitionReader中梳理一下资源文件读取、解析及注册的大致脉络，首先我们看看各个类的功能。

- ResourceLoader:定义资源加载器，主要应用于根据给定的资源文件地址返回对应的Resource.
- BeanDefinitionReader:主要定义资源文件读取并转换为BeanDefinition的各个功能。
  EnvironmentCapable:定义获取Environment方法。
- DocumentLoader:定义从资源文件加载到转换为Document的功能。
- AbstractBeanDefinitionReader:对 EnvironmentCapable、BeanDefinitionReader类定义的功能进行实现。
- BeanDefinitionDocumentReader:定义读取Docuemnt 并注册BeanDefinition 功能。
  BeanDefinitionParserDelegate:定义解析Element 的各种方法。

经过以上分析，我们可以梳理出整个XML 配置文件读取的大致流程，如图2-6所示在XmlBeanDifinitionReader中主要包含以下几步的处理。

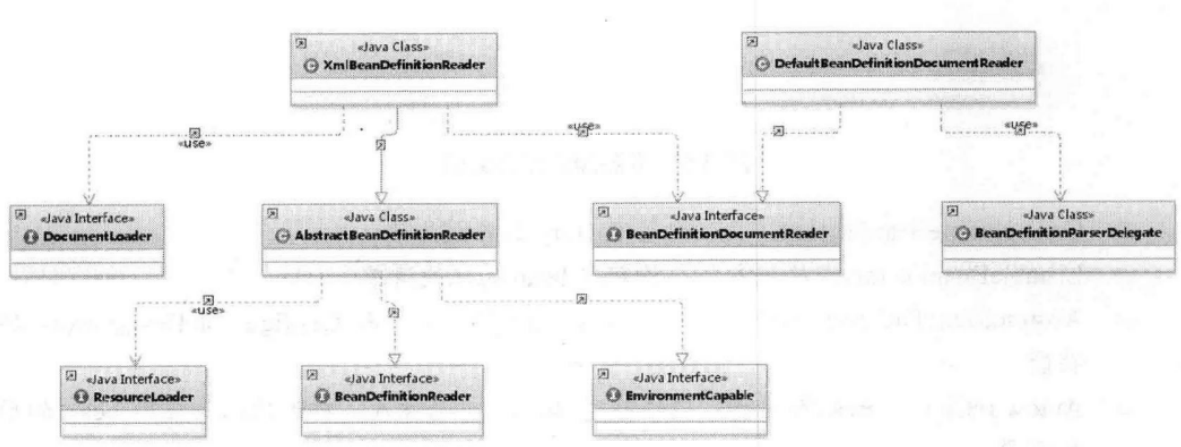


图 2-6　配置文件读取相关类图

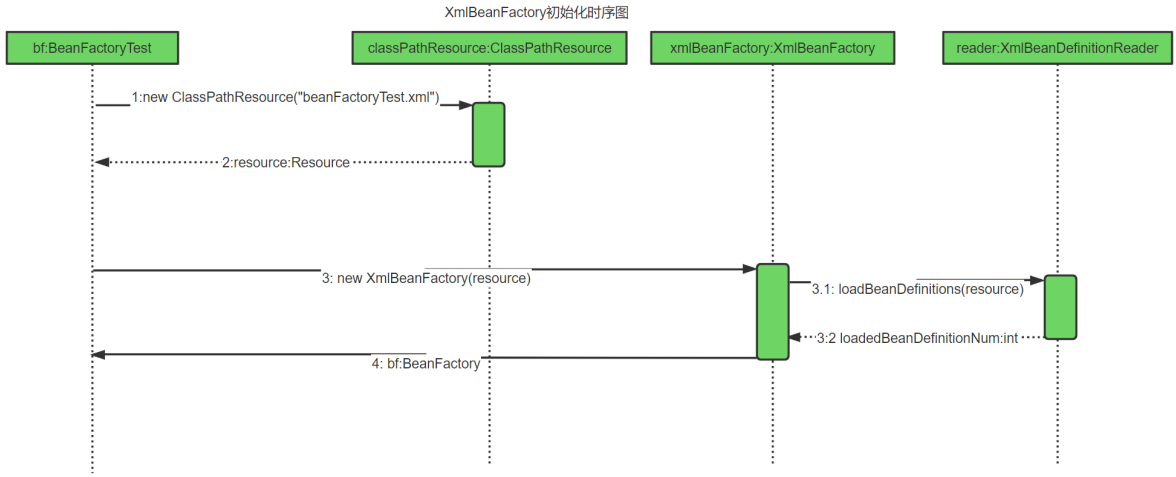
(1)通过继承自AbstractBeanDefinitionReader中的方法，来使用ResourLoader将资源文件路径转换为对应的 Resource文件。
(2)通过DocumentLoader对 Resource文件进行转换，将Resource文件转换为Document文件。
(3)通过实现接口BeanDefinitionDocumentReader 的DefaultBeanDefinitionDocumentReader类对Document进行解析，并使用BeanDefinitionParserDelegate对 Element进行解析。

2.5容器的基础 XmlBeanFactory

好了，到这里我们已经对Spring的容器功能有了一个大致的了解，尽管你可能还很迷糊,但是不要紧，接下来我们会详细探索每个步骤的实现。再次重申一下代码，我们接下来要深入分析以下功能的代码实现:

```
BeanFactory bf = new XmlBeanFactory(new ClassPathResource ("beanFactoryTest.xml"
));
```

通过XmlBeanFactory初始化时序图（如图2-7所示）我们来看一看上面代码的执行逻辑。



时序图从 BeanFactoryTest测试类开始,通过时序图我们可以一目了然地看到整个逻辑处理顺序。在测试的 BeanFactoryTest 中首先调用ClassPathResource的构造函数来构造Resource资源文件的实例对象，这样后续的资源处理就可以用Resource提供的各种服务来操作了，当我们有了Resource后就可以进行XmlBeanFactory 的初始化了。那么Resource资源是如何封装的呢?

2.5.1配置文件封装

Spring 的配置文件读取是通过ClassPathResource 进行封装的，如new ClassPathResource("beanFactoryTest.xml")，那么ClassPathResource完成了什么功能呢?

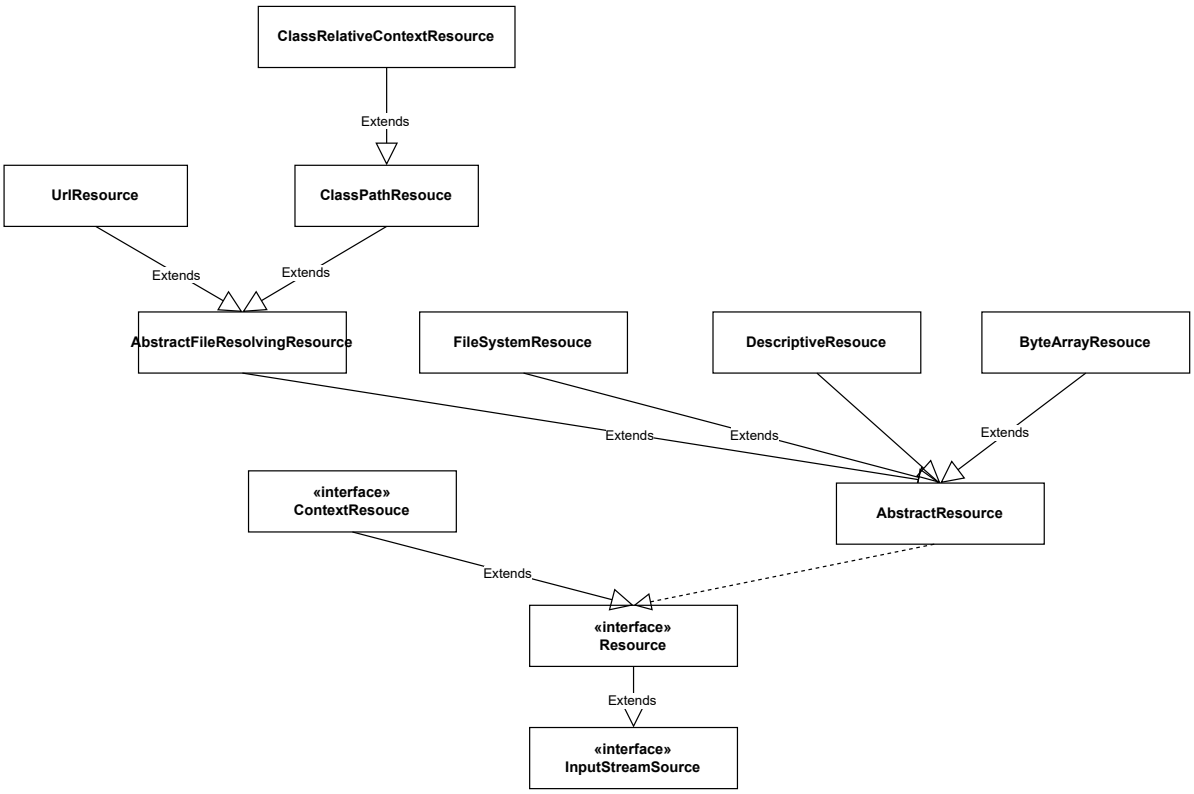
在Java中，将不同来源的资源抽象成URL，通过注册不同的handler (URLStreamHandler)来处理不同来源的资源的读取逻辑，一般handler 的类型使用不同前缀（协议，Protocol)来识别，如“file:”、“http:”、“jar:”等，然而 URL没有默认定义相对Classpath或ServletContext等资源的handler，虽然可以注册自己的URLStreamHandler来解析特定的URL前缀（协议),比如“classpath:”，然而这需要了解URL的实现机制，而且 URL也没有提供一些基本的方法，如检查当前资源是否存在、检查当前资源是否可读等方法。因而Spring 对其内部使用到的资源实现了自己的抽象结构: Resource接口来封装底层资源。

```
public interface InputstreamSource {
    Inputstream getInputStream() throws IOException;
}
public interface Resource extends InputstreamSource {
    boolean exists();
    boolean isReadable();
    boolean isopen();
    URL getURL() throws IOException;
    URI getURI() throws IOException;
    File getFile() throws IOException;
    long lastModified() throws IOException;
    Resource createRelative(string relativePath) throws IOException;
    string getFilename();
    string getDescription();
}
```

InputStreamSource封装任何能返回InputStream 的类，比如File、Classpath下的资源和Byte Array等。它只有一个方法定义: getInputStream()，该方法返回一个新的InputStream对象。

Resource接口抽象了所有Spring 内部使用到的底层资源:File、URL、Classpath等。首先，它定义了3个判断当前资源状态的方法:存在性(exists)、可读性(isReadable)、是否处于打开状态(isOpen)。另外，Resource接口还提供了不同资源到URL、URI、File类型的转换，以及获取lastModified属性、文件名(不带路径信息的文件名，getFilename())的方法。为了便于操作，Resource还提供了基于当前资源创建一个相对资源的方法:createRelative()。在错误处理中需要详细地打印出错的资源文件，因而Resource还提供了getDescription()方法用于在错误处理中的打印信息。

对不同来源的资源文件都有相应的Resource实现:文件(FileSystemResource)、Classpath资源(ClassPathResource)、URL资源(UrlResource)、InputStream资源(InputStreamResource)、Byte数组(ByteArrayResource)等。相关类图如2-8所示。



在日常的开发工作中，资源文件的加载也是经常用到的，可以直接使用Spring提供的类，比如在希望加载文件时可以使用以下代码:

```
Resource resource=new ClassPathResource ( "beanFactoryTest.xml");
InputStream inputstream=resource.getInputstream();
```

得到inputStream后,我们就可以按照以前的开发方式进行实现了。并且我们已经可以利用Resource及其子类为我们提供好的诸多特性。有了Resource接口便可以对所有资源文件进行统一处理。至于实现其实是非常简单的，以getInputStream为例, ClassPathResource中的实现方式便是通过class或者classLoader提供的底层方法进行调用，而对于FileSystemResource 的实现其实更简单，直接使用FileInputStream对文件进行实例化。

ClassPathResouce.java

```
@Override
public InputStream getInputStream() throws IOException {
   InputStream is;
   if (this.clazz != null) {
      is = this.clazz.getResourceAsStream(this.path);
```

```
    }
    else if (this.classLoader != null) {
       is = this.classLoader.getResourceAsStream(this.path);
    }
    else {
       is = ClassLoader.getSystemResourceAsStream(this.path);
    }
    if (is == null) {
       throw new FileNotFoundException(getDescription() + " cannot be opened
because it does not exist");
    }
    return is;
}
```

FileSystemResource.java

```
@Override
public InputStream getInputStream() throws IOException {
    try {
       return Files.newInputStream(this.filePath);
    }
    catch (NoSuchFileException ex) {
       throw new FileNotFoundException(ex.getMessage());
    }
}
```

当通过Resource相关类完成了对配置文件进行封装后配置文件的读取工作就全权交给XmlBeanDefinitionReader来处理了。

了解了Spring 中将配置文件封装为Resource类型的实例方法后，我们就可以继续探寻XmlBeanFactory的初始化过程了，XmlBeanFactory的初始化有若干办法，Spring 中提供了很多的构造函数，在这里分析的是使用Resource实例作为构造函数参数的办法，代码如下:

@Deprecated

XmlBeanFactory.java

```
public class XmlBeanFactory extends DefaultListableBeanFactory {
    private final XmlBeanDefinitionReader reader = new
XmlBeanDefinitionReader(this);
    /**
     * Create a new XmlBeanFactory with the given resource,
     * which must be parsable using DOM.
     * @param resource the XML resource to load bean definitions from
     * @throws BeansException in case of loading or parsing errors
     */
    public XmlBeanFactory(Resource resource) throws BeansException {
       this(resource, null);//调用底下的构造方法
    }
    /**
     * Create a new XmlBeanFactory with the given input stream,
     * which must be parsable using DOM.
     * @param resource the XML resource to load bean definitions from
     * @param parentBeanFactory parent bean factory
     * @throws BeansException in case of loading or parsing errors
     */
```

```
    public XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory)
throws BeansException {
        super(parentBeanFactory);
        this.reader.loadBeanDefinitions(resource);
    }
}
```

上面函数中的代码 this.reader.loadBeanDefinitions(resource)才是资源加载的真正实现，也是我们分析的重点之一。我们可以看到时序图中提到的XmlBeanDefinitionReader加载数据就是在这里完成的,但是在XmlBeanDefinitionReader加载数据前还有一个调用父类构造函数初始化的过程:super(parentBeanFactory)，跟踪代码到父类AbstractAutowireCapableBeanFactory 的构造函数中:

AbstractAutowireCapableBeanFactory.java

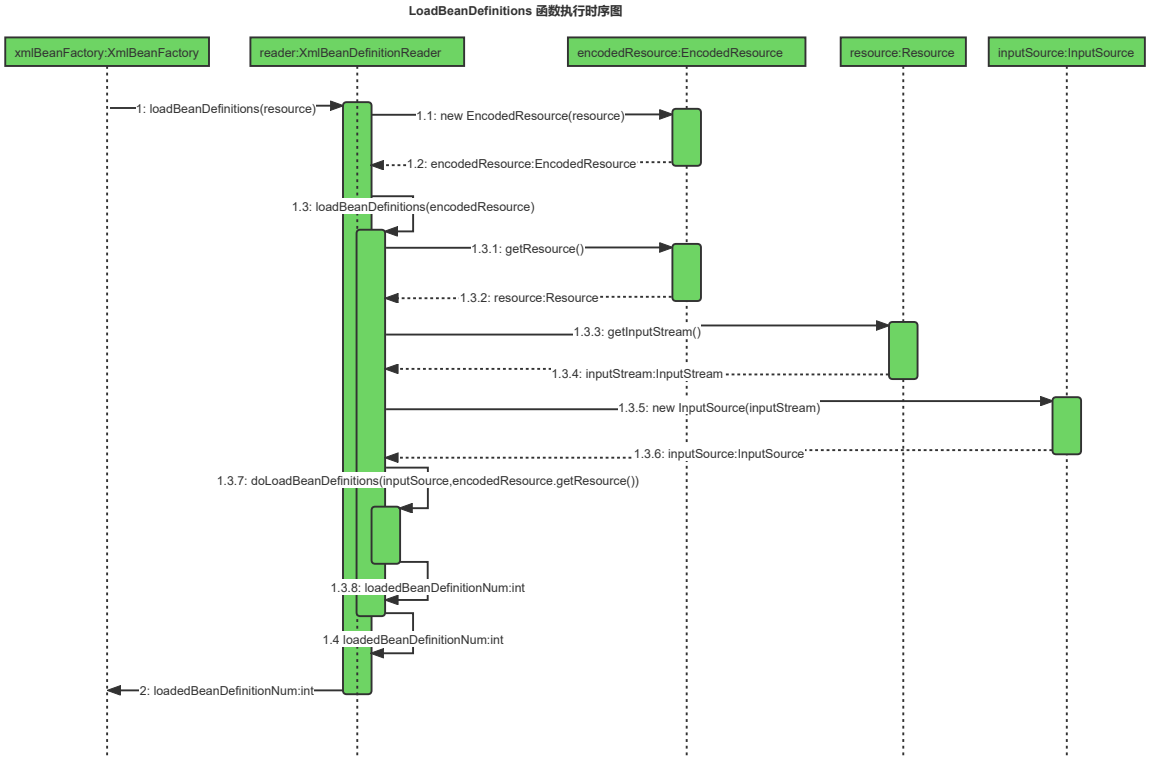
```
/**
 * Create a new AbstractAutowireCapableBeanFactory.
 */
public AbstractAutowireCapableBeanFactory() {
    super();
    ignoreDependencyInterface(BeanNameAware.class);
    ignoreDependencyInterface(BeanFactoryAware.class);
    ignoreDependencyInterface(BeanClassLoaderAware.class);
}
```

这里有必要提及一下ignoreDependencyInterface方法。ignoreDependencyInterface的主要功能是忽略给定接口的自动装配功能，那么，这样做的目的是什么呢?会产生什么样的效果呢?

举例来说:当A中有属性B，那么当Spring在获取A的 Bean 的时候如果其属性B还没有初始化,那么Spring 会自动初始化B，这也是Spring中提供的一个重要特性。但是，某些情况下，B不会被初始化，其中的一种情况就是B实现了BeanNameAware接口，spring中是这样介绍的:自动装配时忽略给定的依赖接口，典型应用是通过其他方式解析Application 上下文注册依赖，类似于 BeanFactory 通过BeanFactoryAware进行注入或者ApplicationContext通过ApplicationContextAware进行注入。

2.5.2加载Bean

之前提到的在XmlBeanFactory构造函数中调用了XmlBeanDefinitionReader类型的reader属性提供的方法 this.reader.loadBeanDefinitions(resource)，而这句代码则是整个资源加载的切人点,我们先来看看这个方法的时序图，如图2-9所示。



看到图2-9我们才知道什么叫山路十八弯，绕了这么半天还没有真正地切人正题，比如加载XML文档和解析注册 Bean，一直还在做准备工作。我们根据上面的时序图来分析一下这里究竟在准备什么?从上面的时序图中我们尝试梳理整个的处理过程如下。

（1）封装资源文件。当进入XmlBeanDefinitionReader后，首先对参数Resource使用EncodedResource类进行封装。

（2）获取输入流。从 Resource中获取对应的 InputStream并构造 InputSource。

（3）通过构造的 InputSource实例和 Resource实例继续调用函数doLoadBeanDefinitionso我们来看一下loadBeanDefinitions函数具体的实现过程:

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

那么EncodedResource的作用是什么呢?通过名称，我们可以大致推断这个类主要是用于对资源文件的编码进行处理的。其中的主要逻辑体现在getReader()方法中，当设置了编码属性的时候Spring会使用相应的编码作为输入流的编码。

```
public Reader getReader() throws IOException {
    if (this.charset != null) {
        return new InputStreamReader(this.resource.getInputStream(),
this.charset);
    }
    else if (this.encoding != null) {
        return new InputStreamReader(this.resource.getInputStream(),
this.encoding);
    }
    else {
        return new InputStreamReader(this.resource.getInputStream());
    }
}
```

上面代码构造了一个有编码encoding的InputStreamReader。当构造好encodedResource对象后，再次转入了可复用方法 loadBeanDefinitions(new EncodedResource(resource))。

这个方法内部才是真正的数据准备阶段，也就是时序图所描述的逻辑:

```
/**
 * Load bean definitions from the specified XML file.
 * @param encodedResource the resource descriptor for the XML file,
 * allowing to specify an encoding to use for parsing the file
 * @return the number of bean definitions found
 * @throws BeanDefinitionStoreException in case of loading or parsing errors
 */
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);
    }
    Set<EncodedResource> currentResources =
this.resourcesCurrentlyBeingLoaded.get();
    //通过属性记录已经加载过的资源
    if (!currentResources.add(encodedResource)) {
        throw new BeanDefinitionStoreException(
            "Detected cyclic loading of " + encodedResource + " - check your
import definitions!");
    }
    //从encodedResource中获取已经封装的Resource对象并再次从Resource中获取其中的
inpPutStream
    try (InputStream inputStream =
encodedResource.getResource().getInputStream()) {
        //InputSource这个类并不来自于Spring，它的全路径是org.xml.sax.InputSource
        InputSource inputSource = new InputSource(inputStream);
        if (encodedResource.getEncoding() != null) {
            inputSource.setEncoding(encodedResource.getEncoding());
        }
        //真正进入了逻辑核心部分
        return doLoadBeanDefinitions(inputSource,
encodedResource.getResource());
    }
    catch (IOException ex) {
        throw new BeanDefinitionStoreException(
            "IOException parsing XML document from " +
encodedResource.getResource(), ex);
    }
    finally {
        currentResources.remove(encodedResource);
        if (currentResources.isEmpty()) {
            this.resourcesCurrentlyBeingLoaded.remove();
        }
    }
}
```

我们再次整理一下数据准备阶段的逻辑，首先对传入的resource参数做封装，目的是考虑到Resource可能存在编码要求的情况,其次,通过SAX读取XML文件的方式来准备InputSourc对象,最后将准备的数据通过参数传人真正的核心处理部分doLoadBeanDefinitions(inputSourceencodedResource.getResource())。

```
/**
 * 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 {
      Document doc = doLoadDocument(inputSource, resource); //2 加载XML获取doc
      int count = registerBeanDefinitions(doc, resource);//3 根据doc 注册bean
      if (logger.isDebugEnabled()) {
         logger.debug("Loaded " + count + " bean definitions from " + resource);
      }
      return count;
   }
   catch ...
}
protected Document doLoadDocument(InputSource inputSource, Resource resource)
throws Exception {// 1 获取xml验证模式
    return this.documentLoader.loadDocument(inputSource, getEntityResolver(),
this.errorHandler,

 getValidationModeForResource(resource), isNamespaceAware());
}
protected int getValidationModeForResource(Resource resource) {
    int validationModeToUse = getValidationMode();
    if (validationModeToUse != VALIDATION_AUTO) {
        return validationModeToUse;
    }
    int detectedMode = detectValidationMode(resource);
    if (detectedMode != VALIDATION_AUTO) {
        return detectedMode;
    }
    // Hmm, we didn't get a clear indication... Let's assume XSD,
    // since apparently no DTD declaration has been found up until
    // detection stopped (before finding the document's root tag).
    return VALIDATION_XSD;
}
```

在上面冗长的代码中假如不考虑异常类的代码，其实只做了三件事,这三件事的每一件都必不可少。
(1)获取对XML文件的验证模式。

(2)加载XML文件，并得到对应的Document。

(3)根据返回的Document注册 Bean信息。
这3个步骤支撑着整个Spring 容器部分的实现基础，尤其是第3步对配置文件的解析,逻辑非常的复杂,那么我们先从获取XML文件的验证模式开始讲起。

2.6获取XML的验证模式

了解XML文件的读者都应该知道XML文件的验证模式保证了XML文件的正确性,而比较常用的验证模式有两种:DTD和XSD。它们之间什么区别呢?

2.6.1 DTD与XSD区别

DTD (Document Type Definition)即文档类型定义，是一种 XML约束模式语言，是XML文件的验证机制，属于XML文件组成的一部分。DTD是一种保证XML文档格式正确的有效方法，可以通过比较XML文档和DTD文件来看文档是否符合规范，元素和标签使用是否正确。一个 DTD文档包含:元素的定义规则，元素间关系的定义规则，元素可使用的属性，可使用的实体或符号规则。
要使用DTD验证模式的时候需要在XML文件的头部声明，以下是在Spring 中使用DTD声明方式的代码:

spring-beans.dtd

```
<!ELEMENT beans (
    description?,
    (import | alias | bean)*
)>

<!--
    Default values for all bean definitions. Can be overridden at
    the "bean" level. See those attribute definitions for details.
-->
<!ATTLIST beans default-lazy-init (true | false) "false">
<!ATTLIST beans default-merge (true | false) "false">
<!ATTLIST beans default-autowire (no | byName | byType | constructor |
autodetect) "no">
<!ATTLIST beans default-init-method CDATA #IMPLIED>
<!ATTLIST beans default-destroy-method CDATA #IMPLIED>

<!--
    Element containing informative text describing the purpose of the enclosing
    element. Always optional.
    Used primarily for user documentation of XML bean definition documents.
-->
<!ELEMENT description (#PCDATA)>
```

XML Schema语言就是XSD (XML Schemas Definition)。XML Schema描述了XML文档的结构。可以用一个指定的XML Schema来验证某个XML文档，以检查该XML文档是否符合其要求。文档设计者可以通过XML Schema指定一个XML文档所允许的结构和内容，并可据此检查一个XML文档是否是有效的。XML Schema本身是一个XML文档，它符合XML语法结构。可以用通用的XML解析器解析它。

在使用XML Schema文档对 XML实例文档进行检验，除了要声明名称空间外(xmlns=http://www.Springframework.org/schema/beans)，还必须指定该名称空间所对应的XML Schema文档的存储位置。通过schemaLocation属性来指定名称空间所对应的XML Schema文档的存储位置，它包含两个部分，一部分是名称空间的URI，另一部分就是该名称空间所标识的 XML Schema文件位置或URL地址(xsi:schemaLocation="http://www.Springframework.org/schema/beans http://www.Springframework.org/schema/beans/Spring-beans.xsd)。

```
<?xml version="1.0"encoding="UTF-8"?>
<beans xmlns="http:/ /www.Springframework.org/schema/beans"
xmlns:xsi="http: //www .w3.org/2001/xMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beanshttp://www.spring
framework.org/schema/beans/Spring-beans.xsd">
</beans>
```

Spring-beans-3.0.xsd

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.springframework.org/schema/beans">
    <xsd:import namespace="http://www.w3.org/XML/1998/namespace"/>
    <xsd:annotation>
        <xsd:documentation>
        </xsd:documentation>
    </xsd:annotation>
    ...
    ...
    <!-- simple internal types -->
    <xsd:simpleType name="defaultable-boolean">
        <xsd:restriction base="xsd:NMTOKEN">
            <xsd:enumeration value="default"/>
            <xsd:enumeration value="true"/>
            <xsd:enumeration value="false"/>
        </xsd:restriction>
    </xsd:simpleType>

</xsd:schema>
```

我们只是简单地介绍一下XML文件的验证模式的相关知识，目的在于让读者对后续知识的理解能有连续性，如果对XML有兴趣的读者可以进一步查阅相关资料。

2.6.2 验证模式的读取

了解了DTD与XSD的区别后我们再去分析Spring 中对于验证模式的提取就更容易理解了。通过之前的分析我们锁定了Spring通过getValidationModeForResource方法来获取对应资源的的验证模式。

XmlDefinitionReader.java

```
/**
 * Determine the validation mode for the specified {@link Resource}.
 * If no explicit validation mode has been configured, then the validation
 * mode gets {@link #detectValidationMode detected} from the given resource.
 * <p>Override this method if you would like full control over the validation//
可以自定义
 * mode, even when something other than {@link #VALIDATION_AUTO} was set.
 * @see #detectValidationMode
 */
protected int getvalidationModeForResource(Resource resource){
    int validationModeToUse = getValidationMode ();
    //如果手动指定了验证模式则使用指定的验证模式
```

```
    if (validationModeToUse != VALIDATION_AUTO){
        return validationModeToUse;
    }
    //如果未指定则使用自动检测
    int detectedMode = detectvalidationMode(resource);// 会调用
XmlValidationModeDetector的detectvalidationMode方法
    if(detectedMode != VALIDATION_AOTO) {
        return detectedMode;
    }
    return VALIDATION_XSD;
}
```

方法的实现其实还是很简单的，无非是如果设定了验证模式则使用设定的验证模式(可以通过对调用XmlBeanDefinitionReader 中的setValidationMode方法进行设定)，否则使用自动检测的方式。而自动检测验证模式的功能是在函数detectValidationMode方法中实现的，在detectValidationMode函数中又将自动检测验证模式的工作委托给了专门处理类XmlValidationModeDetector，调用了XmlValidationModeDetector的 detectValidationMode方法，具体代码如下:

```
public int detectValidationMode(InputStream inputStream) throws IOException {
    // Peek into the file to look for DOCTYPE.
    BufferedReader reader = new BufferedReader(new
InputStreamReader(inputStream));
    try {
        boolean isDtdValidated = false;
        String content;
        while ((content = reader.readLine()) != null) {
            content = consumeCommentTokens(content);
            if (this.inComment || !StringUtils.hasText(content)) {
                continue;
            }
            if (hasDoctype(content)) {// 调用底下的方法
                isDtdValidated = true;
                break;
            }
            if (hasOpeningTag(content)) {
                // End of meaningful data...
                break;
            }
        }
        return (isDtdValidated ? VALIDATION_DTD : VALIDATION_XSD);
    }
    catch (CharConversionException ex) {
        // Choked on some character encoding...
        // Leave the decision up to the caller.
        return VALIDATION_AUTO;
    }
    finally {
        reader.close();
    }
}

/**
 * Does the content contain the DTD DOCTYPE declaration?
 */
```

```
private boolean hasDoctype(String content) {
    return content.contains(DOCTYPE);
}
```

只要我们理解了XSD与 DTD的使用方法，理解上面的代码应该不会太难，Spring用来检测验证模式的办法就是判断是否包含DOCTYPE，如果包含就是DTD，否则就是XSD。

2.7 获取Document

经过了验证模式准备的步骤就可以进行Document加载了，同样XmlBeanFactoryReader类对于文档读取并没有亲力亲为,而是委托给了DocumentLoader去执行,这里的DocumentLoader是个接口，而真正调用的是DefaultDocumentLoader，解析代码如下:

DefaultDocumentLoader.java

```
@Override
public Document loadDocument(InputSource inputSource, EntityResolver
entityResolver,
      ErrorHandler errorHandler, int validationMode, boolean namespaceAware)
throws Exception {

   DocumentBuilderFactory factory = createDocumentBuilderFactory(validationMode,
namespaceAware);
   if (logger.isTraceEnabled()) {
      logger.trace("Using JAXP provider [" + factory.getClass().getName() +
"]");
   }
   DocumentBuilder builder = createDocumentBuilder(factory, entityResolver,
errorHandler);
   return builder.parse(inputSource);
}
```

对于这部分代码其实并没有太多可以描述的,因为通过SAX解析XML文档的套路大致都差不多，Spring在这里并没有什么特殊的地方，同样首先创建DocumentBuilderFactory，再通过DocumentBuilderFactory创建 DocumentBuilder，进而解析inputSource来返回Document对象。对此感兴趣的读者可以在网上获取更多的资料。这里有必要提及一下 EntityResolver，对于参数entityResolver，传入的是通过getEntityResolver()函数获取的返回值，如下代码:

XmlBeanDefinitionReader.java

```
protected Document doLoadDocument(InputSource inputSource, Resource resource)
throws Exception {
   return this.documentLoader.loadDocument(inputSource, getEntityResolver(),
this.errorHandler,
         getValidationModeForResource(resource), isNamespaceAware());
}
/**
 * Return the EntityResolver to use, building a default resolver
 * if none specified.
 */
protected EntityResolver getEntityResolver() {
    if (this.entityResolver == null) {
        // Determine default EntityResolver to use.
        ResourceLoader resourceLoader = getResourceLoader();
```

```
        if (resourceLoader != null) {
            this.entityResolver = new ResourceEntityResolver(resourceLoader);
        }
        else {
            this.entityResolver = new
DelegatingEntityResolver(getBeanClassLoader());
        }
    }
    return this.entityResolver;
}
```

那么，EntityResolver 到底是做什么用的呢?

2.7.1 EntityResolver 用法

在loadDocument方法中涉及一个参数EntityResolver,何为EntityResolver?官网这样解释:如果SAX应用程序需要实现自定义处理外部实体，则必须实现此接口并使用setEntityResolver方法向SAX驱动器注册一个实例。也就是说，对于解析一个XML，SAX首先读取该XML文档上的声明，根据声明去寻找相应的DTD定义，以便对文档进行一个验证。默认的寻找规则,即通过网络（实现上就是声明的 DTD 的URI地址）来下载相应的 DTD声明，并进行认证。下载的过程是一个漫长的过程,而且当网络中断或不可用时,这里会报错,就是因为相应的DTD声明没有被找到的原因。

EntityResolver 的作用是项目本身就可以提供一个如何寻找DTD声明的方法，即由程序来实现寻找DTD声明的过程，比如我们将DTD文件放到项目中某处，在实现时直接将此文档读取并返回给SAX即可。这样就避免了通过网络来寻找相应的声明。

首先看entityResolver的接口方法声明:

```
Inputsource resolveEntity (string publicId, string systemId)
```

这里，它接收两个参数publicId和 systemId，并返回一个inputSource对象。这里我们以特定配置文件来进行讲解。

(1）如果我们在解析验证模式为XSD的配置文件，代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://vww.Springframework.org/schema/beans/Spring-     beans.xsd">
</beans>
```

读取到以下两个参数。

- publicId: null
- systemId: http://www.Springframework.org/schema/beans/Spring-beans.xsd

(2）如果我们在解析验证模式为DTD的配置文件，代码如下:

```
<?xml version="1.0"encoding=UTF一8"?>
<! DOCTYPE beans PUBLIC "-//Spring//DTD BEAN 2.0//ENM"http://ww
.Springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
</beans>
```

读取到以下两个参数。

- publicId:-l/Spring//DTD BEAN 2.0//EN
- systemId: http://www.Springframework.org/dtd/Spring-beans-2.0.dtd

之前已经提到过，验证文件默认的加载方式是通过URL进行网络下载获取，这样会造成延迟，用户体验也不好，一般的做法都是将验证文件放置在自己的工程里，那么怎么做才能将这个URL转换为自己工程里对应的地址文件呢?我们以加载DTD文件为例来看看Spring 中是如何实现的。根据之前Spring 中通过getEntityResolver()方法对EntityResolver 的获取，我们知道，Spring 中使用DelegatingEntityResolver类为EntityResolver的实现类，resolveEntity实现方法如下:

DelegatingEntityResolver.java

```
@Override
@Nullable
public InputSource resolveEntity(@Nullable String publicId, @Nullable String
systemId)
      throws SAXException, IOException {
   if (systemId != null) {
      if (systemId.endsWith(DTD_SUFFIX)) {
         return this.dtdResolver.resolveEntity(publicId, systemId);
      }
      else if (systemId.endsWith(XSD_SUFFIX)) {
         return this.schemaResolver.resolveEntity(publicId, systemId);
      }
   }
   // Fall back to the parser's default behavior.
   return null;
}
```

我们可以看到，对不同的验证模式，Spring使用了不同的解析器解析。这里简单描述一下原理,比如加载DTD类型的BeansDtdResolver的resolveEntity是直接截取systemId最后的xx.dtd然后去当前路径下寻找，而加载XSD类型的PluggableSchemaResolver类的resolveEntity是默认到META-INF/Spring.schemas文件中找到systemid所对应的XSD文件并加载。

BeansDtdResolver.java

```
private static final String DTD_EXTENSION = ".dtd";
private static final String DTD_NAME = "spring-beans";

@Override
@Nullable
public InputSource resolveEntity(@Nullable String publicId, @Nullable String
systemId) throws IOException {
    if (logger.isTraceEnabled()) {
        logger.trace("Trying to resolve XML entity with public ID [" + publicId
+
                    "] and system ID [" + systemId + "]");
    }
    if (systemId != null && systemId.endsWith(DTD_EXTENSION)) {
        int lastPathSeparator = systemId.lastIndexOf('/');
        int dtdNameStart = systemId.indexOf(DTD_NAME, lastPathSeparator);
        if (dtdNameStart != -1) {
            String dtdFile = DTD_NAME + DTD_EXTENSION;// spring-beans.dtd
            if (logger.isTraceEnabled()) {
```

```
                logger.trace("Trying to locate [" + dtdFile + "] in Spring jar
on classpath");
            }
            try {
                Resource resource = new ClassPathResource(dtdFile, getClass());
                InputSource source = new InputSource(resource.getInputStream());
                source.setPublicId(publicId);
                source.setSystemId(systemId);
                if (logger.isTraceEnabled()) {
                    logger.trace("Found beans DTD [" + systemId + "] in
classpath: " + dtdFile);
                }
                return source;
            }
            catch (FileNotFoundException ex) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Could not resolve beans DTD [" + systemId +
"]: not found in classpath", ex);
                }
            }
        }
    }
    // Fall back to the parser's default behavior.
    return null;
}
```

2.8解析及注册BeanDefinitions

当把文件转换为Document后，接下来的提取及注册bean就是我们的重头戏。继续上面的分析，当程序已经拥有XML文档文件的 Document实例对象时，就会被引入下面这个方法。

XmlBeanDefinitionReader.java

```
public int registerBeanDefinitions(Document doc, Resource resource) throws
BeanDefinitionStoreException {
    //使用DefaultBeanDefinitionDocumentReader实例化BeanDefinitionDocumentReader
    BeanDefinitionDocumentReader documentReader =
createBeanDefinitionDocumentReader();
    //在实例化BeanDefinitionReader时候会将BeanDefinitionRegistry传入，默认使用继承自
DefaultListableBeanFactory的子类
    //记录统计前BeanDefinition的加载个数
    int countBefore = getRegistry().getBeanDefinitionCount();
    //加载及注册bean
    documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
    //记录本次加载的BeanDefinition的个数
    return getRegistry().getBeanDefinitionCount() - countBefore;
}
```

其中的参数 doc是通过上一节 loadDocument 加载转换出来的。在这个方法中很好地应用了面向对象中单一职责的原则，将逻辑处理委托给单一的类进行处理，而这个逻辑处理类就是BeanDefinitionDocumentReader。BeanDefinitionDocumentReader是一个接口，而实例化的工作是在createBeanDefinitionDocumentReader()中完成的，而通过此方法，BeanDefinitionDocumentReader真正的类型其实已经是DefaultBeanDefinitionDocumentReader

了，进入DefaultBeanDefinition DocumentReader后，发现这个方法的重要目的之一就是提取root，以便于再次将root作为参数继续BeanDefinition的注册。

DefaultBeanDefinitionDocumentReader.java

```
@Override
public void registerBeanDefinitions(Document doc, XmlReaderContext
readerContext) {
   this.readerContext = readerContext;

doRegisterBeanDefinitions(doc.getDocumentElement());//root=doc.getDocumentElemen
t()
}
```

经过艰难险阻，磕磕绊绊，我们终于到了核心逻辑的底部doRegisterBeanDefinitions(root)至少我们在这个方法中看到了希望。
如果说以前一直是XML加载解析的准备阶段,那么doRegisterBeanDefinitions算是真正地开始进行解析了，我们期待的核心部分真正开始了。

DefaultBeanDefinitionDocumentReader.java

```
/**
 * Register each bean definition within the given root {@code <beans/>} element.
 */
@SuppressWarnings("deprecation")  // for Environment.acceptsProfiles(String...)
protected void doRegisterBeanDefinitions(Element root) {
   // Any nested <beans> elements will cause recursion in this method. In
   // order to propagate and preserve <beans> default-* attributes correctly,
   // keep track of the current (parent) delegate, which may be null. Create
   // the new (child) delegate with a reference to the parent for fallback
purposes,
   // then ultimately reset this.delegate back to its original (parent)
reference.
   // this behavior emulates a stack of delegates without actually necessitating
one.
   BeanDefinitionParserDelegate parent = this.delegate;
    //专门处理解析
   this.delegate = createDelegate(getReaderContext(), root, parent);

   if (this.delegate.isDefaultNamespace(root)) {
      String profileSpec = root.getAttribute(PROFILE_ATTRIBUTE);
      if (StringUtils.hasText(profileSpec)) {
         String[] specifiedProfiles = StringUtils.tokenizeToStringArray(
               profileSpec,
BeanDefinitionParserDelegate.MULTI_VALUE_ATTRIBUTE_DELIMITERS);
         // We cannot use Profiles.of(...) since profile expressions are not
supported
         // in XML config. See SPR-12458 for details.
         if
(!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles)) {
            if (logger.isDebugEnabled()) {
               logger.debug("Skipped XML bean definition file due to specified
profiles [" + profileSpec +
                     "] not matching: " + getReaderContext().getResource());
            }
```

```
                return;
            }
        }
    }
     //解析前置处理，交给子类实现   模板方法，可以进行自定义
    preProcessXml(root);
    parseBeanDefinitions(root, this.delegate);
     //解析后置处理，交给子类实现   模板方法，可以进行自定义
    postProcessXml(root);
    this.delegate = parent;
}
```

通过上面的代码我们看到了处理流程，首先是对profile的处理，然后开始进行解析，可是当我们跟进 preProcessXml(root)或者postProcessXml(root)发现代码是空的，既然是空的写着还要有什么用呢?就像面向对象设计方法学中常说的一句话，一个类要么是面向继承的设计的，要么就用final修饰。在DefaultBeanDefinitionDocumentReader 中并没有用final修饰，所以它是面向继承而设计的。这两个方法正是为子类而设计的，如果读者有了解过设计模式，可以很快速地反映出这是模版方法模式，如果继承自DefaultBeanDefinitionDocumentReader 的子类需要在 Bean解析前后做一些处理的话，那么只需要重写这两个方法就可以了。

2.8.1 profile属性的使用(区分各种环境)

我们注意到在注册Bean的最开始是对PROFILE_ATTRIBUTE属性的解析，可能对于我们来说,profile属性并不是很常用。让我们先了解一下这个属性。
分析profile前我们先了解下profile 的用法，官方示例代码片段如下:

```
<beans xmlns="http://www.Springframework.org/schema/beans"
    xmlns :xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jdbc="http://www.Springframework.org/ schema/jdbc"
    xmlns:jee="http://www.Springframework.org/schema/jee"
xsi:schemaLocation="...">
    <beans profile="dev" >
    </beans>
    <beans profile="production" >
    </beans>
</beans>
```

集成到Web环境中时,在 web.xml中加入以下代码:

```
<context-param>
    <param-name>Spring.profiles.active</param-name>
    <param-value>dev</param-value>
</context-param>
```

有了这个特性我们就可以同时在配置文件中部署两套配置来适用于生产环境和开发环境,这样可以方便的进行切换开发、部署环境,最常用的就是更换不同的数据库。

了解了profile的使用再来分析代码会清晰得多，首先程序会获取 beans节点是否定义了profile属性，如果定义了则会需要到环境变量中去寻找，所以这里首先断言environment 不可能为空，因为profile是可以同时指定多个的，需要程序对其拆分，并解析每个profile是都符合环境变量中所定义的,不定义则不会浪费性能去解析。

2.8.2解析并注册BeanDefinition

处理了 profile后就可以进行XML的读取了，跟踪代码进入 parseBeanDefinitions(root,this.delegate)。

DefaultBeanDefinitionDocumentReader.java

```
/**
 * Parse the elements at the root level in the document:
 * "import", "alias", "bean".
 * @param root the DOM root element of the document
 */
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate
delegate) {
   if (delegate.isDefaultNamespace(root)) {
      NodeList nl = root.getChildNodes();
      for (int i = 0; i < nl.getLength(); i++) {
         Node node = nl.item(i);
         if (node instanceof Element) {
            Element ele = (Element) node;
            if (delegate.isDefaultNamespace(ele)) {
               parseDefaultElement(ele, delegate);
            }
            else {
               delegate.parseCustomElement(ele);
            }
         }
      }
   }
   else {
      delegate.parseCustomElement(root);
   }
}

private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate
delegate) {
   /**
    * import
    */
   if (delegate.nodeNameEquals(ele, IMPORT_ELEMENT)) {
      importBeanDefinitionResource(ele);
   }
   /**
    * alias
    */
   else if (delegate.nodeNameEquals(ele, ALIAS_ELEMENT)) {
      processAliasRegistration(ele);
   }
   /**
    * bean
    */
   else if (delegate.nodeNameEquals(ele, BEAN_ELEMENT)) {
      processBeanDefinition(ele, delegate);
   }
   /**
    * nested<beans>
```

```
     */
    else if (delegate.nodeNameEquals(ele, NESTED_BEANS_ELEMENT)) {
        // recurse
        doRegisterBeanDefinitions(ele);
    }
}
```

上面的代码看起来逻辑还是蛮清晰的，因为在Spring 的 XML 配置里面有两大类Bean声明，一个是默认的，如:

```
<bean id="test" class="test.TestBean"/>
```

另一类就是自定义的，如:

```
<tx:annotation-driven/>
```

而两种方式的读取及解析差别是非常大的，如果采用Spring默认的配置，Spring当然知道该怎么做，但是如果是自定义的，那么就需要用户实现一些接口及配置了。对于根节点或者子节点如果是默认命名空间的话则采用parseDefaultElement方法进行解析，否则使用delegate.parseCustomElement方法对自定义命名空间进行解析。而判断是否默认命名空间还是自定义命名空间的办法其实是使用node.getNamespaceURI()获取命名空间,并与Spring 中固定的命名空间http://www.Springframework.org/schema/beans进行比对。如果一致则认为是默认，否则就认为是自定义。而对于默认标签解析与自定义标签解析我们将会在下一章中进行讨论。