Spring IOC容器实现分析

准备工作

我们都知道, IOC容器和AOP是Spring框架的核心, "To the developer, for the developer and by the developer" - 简化JAVA企业应用的的开发是Spring框架的目标, 为更好的使用IOC容器, 我们结合Spring IOC的源代码对它的实现作一个分析。在了解IOC容器实现的基础上, Spring的使用者可以跟好的使用IOC容器和Spring框架, 同时如果需要对Spring框架作自己的扩展, 这些方面的了解也是很有必要的。我们在这里假设读者已经具备对Spring IOC容器使用的基本知识 - 关于对Spring IOC容器的使用, 可以参考以下的参考资料, 这里就不对一些使用和配置的问题多做讲解了。

- Spring Framework Reference Guide
- Spring In Action
- Expert One-on-one J2EE Development without EJB
- Professional Java Development with the Spring Framework

还需要准备好Spring的源代码, 我们这里用的代码是Spring2.0, 当然了一个可以查看源代码的编辑器也是需要的, 这里使用的是Eclipse3.2 - 很多说明性的图例都是直接从屏幕拷贝下来的。下面是一些文章中用到的专有词汇:
上下文:ApplicationContext
Bean定义信息:BeanDefinition
Bean工厂:BeanFactory
工厂Bean:FactoryBean
单件:Singleton

概述:基本IOC容器和上下文

因为IOC容器为应用开发者管理对象之间的依赖关系提供了很多便利和基础服务, 所以业界有许多IOC容器供开发者选择, Spring Framework就是其中的一个。对Spring IOC容器的使用来说, 我们常常接触到的Bean工厂和上下文就是IOC容器的表现形式, 在这些Spring提供的基本IOC容器的接口定义和实现的基础上, 我们通过定义Bean定义信息来管理应用中的对象依赖关系。
在使用Spring IOC容器的时候, 了解Bean工厂和上下文之间的区别对我们了解Spring IOC容器是比较重要的。从实现上来看, IOC容器定义的基本接口是在Bean工厂定义的, 也就是说Bean工厂是Spring IOC容器的最基本的形式, 很显然, BeanFactory只是一个接口类, 没有给出IOC容器的实现, 只是对IOC容器需要提供的最基本的服务做了定义, 象我们下面看到的DefaultListableBeanFactory,XmlBeanFactory, ApplicationContext这些都可以看成是IOC容器的某种具体实现。看看Bean工厂是怎样定义IOC容器的基本服务的:

```
public interface BeanFactory {
    //这里是对工厂Bean的转义定义, 因为如果使用bean的名字检索IOC容器得到的对象是工厂Bean生成的对象,
    //如果需要得到工厂Bean本身, 需要使用转义的名字来向IOC容器检索
    String FACTORY_BEAN_PREFIX = "&";

    //这里根据bean的名字, 在IOC容器中得到bean实例, 这个IOC容器就象一个大的抽象工厂, 用户可以根据名字得到需要的bean
    //在Spring中, Bean和普通的JAVA对象不同在于:
    //Bean已经包含了我们在Bean定义信息中的依赖关系的处理, 同时Bean是已经被放到IOC容器中进行管理了, 有它自己的生命周期
    Object getBean(String name) throws BeansException;

    //这里根据bean的名字和Class类型来得到bean实例, 和上面的方法不同在于它会抛出异常:如果根据名字取得的bean实例的Class类型和需要的不同的话。
```

```
    Object getBean(String name, Class requiredType) throws BeansException;

    //这里提供对bean的检索，看看是否在IOC容器有这个名字的bean
    boolean containsBean(String name);

    //这里根据bean名字得到bean实例，并同时判断这个bean是不是单件，在配置的时候，默认的
Bean被配置成单件形式,如果不需要单件形式，需要用户在Bean定义信息中标注出来，这样IOC容器在
每次接受到用户的getBean要求的时候，会生成一个新的Bean返回给客户使用 - 这就是Prototype形
式
    boolean isSingleton(String name) throws NoSuchBeanDefinitionException;

    //这里对得到bean实例的Class类型
    Class getType(String name) throws NoSuchBeanDefinitionException;

    //这里得到bean的别名，如果根据别名检索，那么其原名也会被检索出来
    String[] getAliases(String name);
}
```

这个BeanFactory接口为IOC容器的使用提供了使用规范，在这个基础上，Spring还提供了它符合这个IOC容器接口的实现供开发人员使用，比如XmlBeanFactory和各种常见的上下文，我们先看一下XmlBeanFactory这个IOC容器的实现，和那些上下文相比，它提供了基本的IOC容器的基本功能;我们可以认为直接的BeanFactory的实现是IOC容器的基本形式，而各种上下文的实现是IOC容器的高级表现形式。XmlBeanFactory的实现是这样的：

```
public class XmlBeanFactory extends DefaultListableBeanFactory {
    //这里为容器定义了一个默认使用的bean定义读取器，在Spring的使用中，Bean定义信息的读取是
容器初始化的一部分，但是在实现上是和容器的注册以及依赖的注入是分开的，这样可以使用灵活的
bean定义读取机制。
    private final XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(this);
    //这里需要一个Resource类型的Bean定义信息，实际上的定位过程是由Resource的构建过程来完
成的。
    public XmlBeanFactory(Resource resource) throws BeansException {
        this(resource, null);
    }
    //在初始化函数中使用读取器来对资源进行读取，得到bean定义信息。这里完成整个IOC容器对
Bean定义信息的载入和注册过程
    public XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory) throws
BeansException {
        super(parentBeanFactory);
        this.reader.loadBeanDefinitions(resource);
    }
```

我们看到XmlBeanFactory使用了DefaultListableBeanFactory作为它持有的IOC容器实现，在这个基础上，添加了XML形式的Bean定义信息的读取功能。从这个角度看，这个DefaultListableBeanFactory是很重要的一个Spring IOC实现。下面我们可以看到上下文也和XmlBeanFactory一样，通过持有这个DefaultListableBeanFactory来获得基本的IOC容器的功能。通过编程式的使用DefaultListableBeanFactory我们可以看到IOC容器使用的一些基本过程：

```
    ClassPathResource res = new ClassPathResource("beans.xml");
    DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
    XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
    reader.loadBeanDefinitions(res);
```

这些代码演示了以下几个步骤：

1. 创建IOC配置文件的抽象资源
2. 创建一个BeanFactory，这里我们使用DefaultListableBeanFactory
3. 创建一个载入bean定义信息的读取器，这里使用XmlBeanDefinitionReader来载入XML形式的bean定义信息，配置给BeanFactory
4. 从定义好的资源位置读入配置信息，具体的解析过程由XmlBeanDefinitionReader来完成，这样完成整个载入和注册bean定义的过程。我们的IoC容器就建立起来

这个基本过程我们可以看到，IOC容器建立的基本步骤，这些我们可以编程式的完成这些配置，但在Spring中，它提供的上下文已经为我们作了这些事情，所以从这个角度说上下文是一个高级形态上的IOC容器。更方便了用户的使用，相比于那些基本的IOC容器的BeanFactory实现，上下文除了提供基本的上面看到的容器的基本功能外，还为用户提供了以下的附加服务更方便的让客户使用容器：

- 可以支持不同的信息源，我们看到ApplicationContext扩展了MessageSource
- 访问资源，体现在对ResourceLoader和Resource的支持上面，这样我们可以从不同地方得到bean定义资源，这样用户程序可以灵活的定义Bean定义信息
- 支持应用事件，继承了接口ApplicationEventPublisher,这样在上下文中引入了事件机制而BeanFactory没有

在上下文环境中，这些上下文提供的基础服务更丰富了基本IOC容器的功能。所以一般我们建议客户使用上下文作为IOC容器来使用 - 和XmlBeanFactory一样，上下文是通过持有DefaultListableBeanFactory这个基本的IOC容器实现来提供IOC容器的基本功能的，这一点可以在下面我们分析IOC容器的初始化过程中看得很清楚。

IOC容器和上下文的初始化

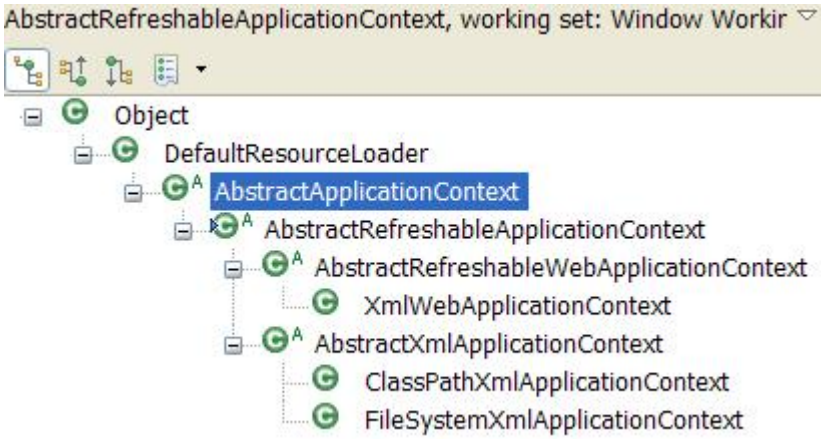
简单来说，IOC容器和上下文的初始化包括Bean定义信息的资源定位，载入和注册过程。在上面编程式的使用DefaultListableBeanFactory中我们可以大致的看到上述过程的实现。值得注意的是，Spring把这三个过程的完成分开并让不同的模块来完成，这样可以让用户更加灵活的对这三个过程来进行剪裁，定义出自己最合适的IOC容器的初始化。比如Bean定义信息的资源定位由ResourceLoader通过统一的Resource接口来完成，这个Resource接口对各种形式的资源信息的使用提供了统一的接口，比如在文件系统中的Bean定义信息可以使用FileSystemResource，在类路径中可以使用上面看到的ClassPathResource等等。第二个关键的部分是Bean定义信息的载入，这个载入过程就是把用户定义好的Bean表示成IOC容器内部的数据结构的过程，在下面我们可以看到这个数据结构就是BeanDefintion，下面我们会对这个载入的过程做一个详细的分析；第三个过程是向IOC容器注册这些Bean定义信息的过程，这个过程是通过调用BeanDefinitionRegistry接口的实现来完成的，这个注册过程把载入过程中解析得到的Bean定义信息向IOC容器中进行注册 - 在IOC容器内部往往使用一个象HashMap这样的容器来持有这些Bean定义。
值得注意的是，IOC容器和上下文的初始化一般不包含Bean的依赖注入的实现，关于依赖注入实现的过程在下面也会进行详细的分析。好了，下面我们详细的看一看IOC容器和上下文的Bean定义信息的资源定位，载入和注册过程是怎么实现的。

Bean定义信息的资源定位

在上面编程式使用DefaultListableBeanFactory的时候，我们可以看到首先定义一个Resource来定位容器使用的Bean定信息：

```
ClassPathResource res = new ClassPathResource("beans.xml");
```

这个定义的Resource并不是让DefaultListableBeanFactory直接使用，而是让BeanDefinitionReader来使用，这里我们也可以看到使用上下文对于直接使用DefaultListableBeanFactory的好处，因为在上下文中的使用中，Spring已经为我们提供了一系列具备Resource功能的实现，比如我们常看到的FileSystemXmlApplicationContext,ClassPathXmlApplicationContext,XmlWebApplicationContext,我们下面就看看FileSystemXmlApplicationContext是怎样完成这个资源定位过程的,先看看这些类的继承体系：

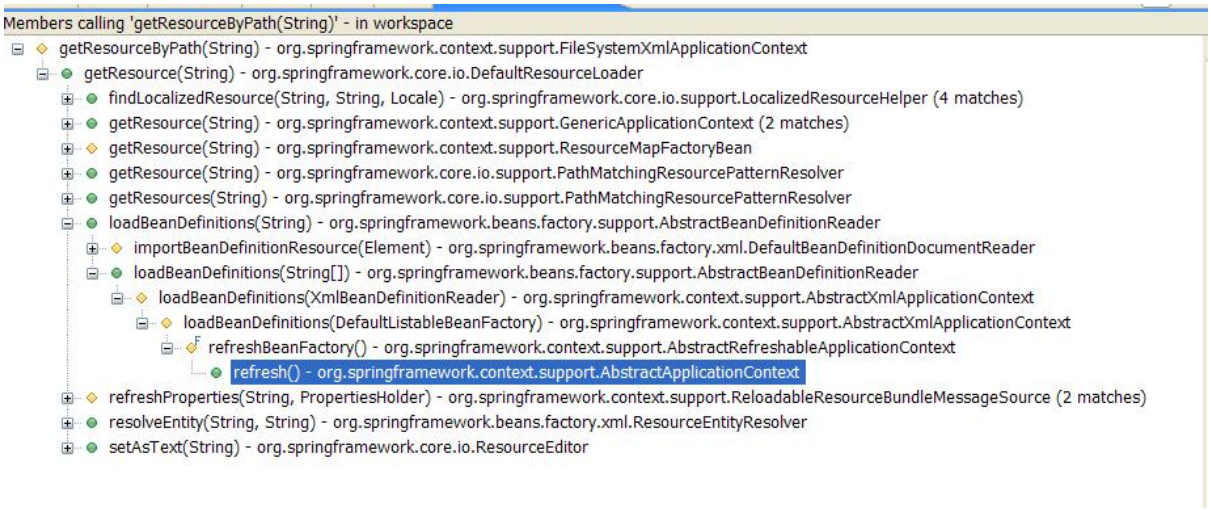


可以看到，这个FileSystemXmlApplicationContext已经通过继承具备了ResourceLoader：

```
public class FileSystemXmlApplicationContext extends AbstractXmlApplicationContext {
    //通过这个字符串数组可以持有多个资源位置
    private String[] configLocations;


    public FileSystemXmlApplicationContext(String configLocation) throws BeansException
{
        this(new String[] {configLocation});
    }

    //这里是一系列初始化函数，得到Resource在文件系统中的位置，并通过refresh来初始化整个IOC
容器
    //这个refresh调用时容器的初始化调用入口
    public FileSystemXmlApplicationContext(String[] configLocations) throws
BeansException {
        this(configLocations, null);
    }

    public FileSystemXmlApplicationContext(String[] configLocations, ApplicationContext
parent)
            throws BeansException {

        super(parent);
        this.configLocations = configLocations;
        refresh();
    }


    public FileSystemXmlApplicationContext(String[] configLocations, boolean refresh)
throws BeansException {
        this(configLocations, refresh, null);
    }

    public FileSystemXmlApplicationContext(String[] configLocations, boolean refresh,
ApplicationContext parent)
            throws BeansException {

        super(parent);
        this.configLocations = configLocations;
        if (refresh) {
```

```
            refresh();
        }
    }


    protected String[] getConfigLocations() {
        return this.configLocations;
    }

   //这里是具体的关于在文件系统中定义Bean定义信息的实现
   //通过构造一个FileSystemResource来得到一个在文件系统中定义的Bean定义信息
   //这个getResourceByPath是在BeanDefinitionReader的loadBeanDefintion中被调用的。
    protected Resource getResourceByPath(String path) {
        if (path != null && path.startsWith("/")) {
            path = path.substring(1);
        }
        return new FileSystemResource(path);
    }
}
```

从下面的调用关系就可以很清楚的看到在初始化调用的refresh中怎样会触发实际的对资源位置的定位过程:



大家会比较奇怪，这个FileSystemXmlApplicationContext在什么地方定义了需要的BeanDefinitionReader呢?我们看看它的基类AbstractRefreshableApplicationContext:

```
public abstract class AbstractRefreshableApplicationContext extends
AbstractApplicationContext {

    /** 这里定义的beanFactory就是ApplicationContext使用的Bean工厂
    private DefaultListableBeanFactory beanFactory;
    .....
    //这个refreshBeanFactory是refresh的一个过程，主要是完成对上下文中IOC容器的初始化
    protected final void refreshBeanFactory() throws BeansException {
        // Shut down previous bean factory, if any.
        synchronized (this.beanFactoryMonitor) {
            if (this.beanFactory != null) {
                this.beanFactory.destroySingletons();
                this.beanFactory = null;
            }
        }
```

```
        // 这里初始化IOC容器
        try {
            //这里创建一个DefaultListableBeanFactory作为上下文使用哪个的IOC容器
            DefaultListableBeanFactory beanFactory = createBeanFactory();
            //这里调用BeanDefinitionReader来载入Bean定义信息
            loadBeanDefinitions(beanFactory);

            synchronized (this.beanFactoryMonitor) {
                this.beanFactory = beanFactory;
            }
            if (logger.isInfoEnabled()) {
                logger.info("Bean factory for application context [" + getDisplayName() + "]:
" + beanFactory);
            }
        }
        catch (IOException ex) {
            throw new ApplicationContextException(
                    "I/O error parsing XML document for application context [" +
getDisplayName() + "]", ex);
        }
    }

    //这就是在上下文中创建DefaultListableBeanFactory的地方
    protected DefaultListableBeanFactory createBeanFactory() {
        return new DefaultListableBeanFactory(getInternalParentBeanFactory());
    }

   //这里是使用BeanDefinitionReader载入Bean定义的地方, 因为允许有多种载入的方式, 虽然用得
最多的是XML定义的形式, 这里委托给子类完成
    protected abstract void loadBeanDefinitions(DefaultListableBeanFactory beanFactory)
            throws IOException, BeansException;
}
这个loadBeanDefinitios在子类AbstractXmlApplicationContext中的实现:
    protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws
IOException {
        // 这里创建XmlBeanDefinitionReader作为读入器
        XmlBeanDefinitionReader beanDefinitionReader = new
XmlBeanDefinitionReader(beanFactory);

        // 这里配置ResourceLoader, 因为DefaultResourceLoader是父类, 所以this可以直接被使用
        beanDefinitionReader.setResourceLoader(this);
        beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));

        // 这是启动Bean定义信息载入的过程
        initBeanDefinitionReader(beanDefinitionReader);
        loadBeanDefinitions(beanDefinitionReader);
    }
从上面的代码可以看到, 在初始化FileSystmXmlApplicationContext的过程中, 启动了IOC容器的初
始化 - refresh ， 这个初始化时通过定义的XmlBeanDefinitionReader来完成的, 使用的IOC容器是
DefualtListableBeanFactory,具体的资源载入在XmlBeanDefinitionReader读入Bean定义的时候
实现- 在AbstractBeanDefinitionReader中:
    public int loadBeanDefinitions(String location) throws BeanDefinitionStoreException {
        //这里取得置入的ResourceLoader,使用的是DefaultResourceLoader
        ResourceLoader resourceLoader = getResourceLoader();
        if (resourceLoader == null) {
            throw new BeanDefinitionStoreException(
```

```
                "Cannot import bean definitions from location [" + location + "]: no
ResourceLoader available");
        }

        if (resourceLoader instanceof ResourcePatternResolver) {
            // 这里对Resource进行解析
            try {
                Resource[] resources = ((ResourcePatternResolver)
resourceLoader).getResources(location);
                int loadCount = loadBeanDefinitions(resources);
                if (logger.isDebugEnabled()) {
                    logger.debug("Loaded " + loadCount + " bean definitions from location
pattern [" + location + "]");
                }
                return loadCount;
            }
            catch (IOException ex) {
                throw new BeanDefinitionStoreException(
                        "Could not resolve bean definition resource pattern [" + location + "]",
ex);
            }
        }
        else {
            // 这里调用DefaultResourceLoader去取得Resource
            Resource resource = resourceLoader.getResource(location);
            int loadCount = loadBeanDefinitions(resource);
            if (logger.isDebugEnabled()) {
                logger.debug("Loaded " + loadCount + " bean definitions from location [" +
location + "]");
            }
            return loadCount;
        }
    }
```

具体的取得Resource实现我们可以看看DefaultResourceLoader是怎样完成的:

```
   public Resource getResource(String location) {
        Assert.notNull(location, "Location must not be null");
        //这里处理带classpath: 前缀的资源定义, 直接返回一个ClassPathResource
        if (location.startsWith(CLASSPATH_URL_PREFIX)) {
            return new
ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.length()),
getClassLoader());
        }
        else {
            try {
                // 然后使用URLResource
                URL url = new URL(location);
                return new UrlResource(url);
            }
            catch (MalformedURLException ex) {
                // 如果都不能处理交给子类的getResourceByPath,比如我们前面看到的
FileSystemXmlApplicationContext的实现
                return getResourceByPath(location);
            }
        }
    }
```

以上的代码对定位资源的过程做了一个基本的描述, 其中涉及到的基本的类有DefaultResourceLoader这是定位Resource的基本类, 由于这个定位过程是在IOC容器的初始化中完

成的, 所以我们可以看到一个上下文初始化的基本过程和定义IOC容器, Bean定义读入器的大概过程 - 在FileSystemXmlApplicationContext这一类的XML上下文中使用的是DefaultListableBeanFactory和XmlBeanDefinitionReader来完成上下文的初始化。

Bean定义信息的载入

上面我们已经看到怎样通过ResourceLoader来定位Bean定义信息的过程, 对使用上下文作为IOC容器的客户来说, 这个过程由上下文替客户完成了, 对使用Bean工厂的客户来说, 需要编程式的为使用的Bean工厂指定Bean定位信息- 而直接的定位过程是与我们的Bean工厂相关的Bean定义读取器(BeanDefinitionReader)在载入过程中完成的 - 也是IOC容器初始化中载入Bean定义信息过程的一部分。容器要载入Bean定义信息, 当然首先要先能够定位到需要的Bean定义信息了。下面我们看看整个Bean定义信息的载入过程。
对IOC容器来说, 这个载入过程相当于把我们定义的Bean定义信息在IOC容器中转化成BeanDefinition的数据结构并建立映射。以后的IOC容器的对Bean的管理功能和依赖注入功能就是通过对BeanDefinition进行操作来完成的。这些BeanDefinition数据在IOC容器里通过一个HashMap来保持和维护 - 这只是一种比较简单的维护方式, 如果你觉得需要提高IOC容器的性能和容量, 可以自己做一些扩展。我们看看上面提到的DefaultListableBeanFactory:

```
public class DefaultListableBeanFactory extends AbstractAutowireCapableBeanFactory
    implements ConfigurableListableBeanFactory, BeanDefinitionRegistry {

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

    /** 这里对是不是预实例化进行控制*/
    private boolean allowEagerClassLoading = true;

    /** 这里就是存放载入Bean定义信息的地方, 以Bean的名字作为key来检索Bean定义信息 */
    private final Map beanDefinitionMap = new HashMap();

    /** 这个列表保存经过排序的Bean的名字 */
    private final List beanDefinitionNames = new ArrayList();

    .......
}
```

我们看看具体的Bean的载入过程, 前面我们看到一个refresh作为载入调用的入口, 这里我们也从这里开始看, 在上下文 - AbstractApplicationContext中的实现给出了一个上下文初始化的基本过程:

```
    public void refresh() throws BeansException, IllegalStateException {
        synchronized (this.startupShutdownMonitor) {
            this.startupTime = System.currentTimeMillis();

            synchronized (this.activeMonitor) {
                this.active = true;
            }

            // 这里初始化IOC容器, 其中包括了对Bean定义信息的载入
            refreshBeanFactory();
            ConfigurableListableBeanFactory beanFactory = getBeanFactory();

            // 下面对使用的Bean工厂进行配置, 这里使用DefaultListableBeanFactory
            beanFactory.setBeanClassLoader(getClassLoader());

            // Populate the bean factory with context-specific resource editors.
            beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this));
```

```
        // Configure the bean factory with context semantics.
        beanFactory.addBeanPostProcessor(new
ApplicationContextAwareProcessor(this));
        beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
        beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
        beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
        beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);

        // Allows post-processing of the bean factory in context subclasses.
        postProcessBeanFactory(beanFactory);

        // 这里对上下文后处理器进行注册
        for (Iterator it = getBeanFactoryPostProcessors().iterator(); it.hasNext();) {
            BeanFactoryPostProcessor factoryProcessor = (BeanFactoryPostProcessor)
it.next();
            factoryProcessor.postProcessBeanFactory(beanFactory);
        }

        if (logger.isInfoEnabled()) {
            if (getBeanDefinitionCount() == 0) {
                logger.info("No beans defined in application context [" + getDisplayName()
+ "]");
            }
            else {
                logger.info(getBeanDefinitionCount() + " beans defined in application
context [" + getDisplayName() + "]");
            }
        }

        try {
            // 这里对工厂后处理器进行触发
            invokeBeanFactoryPostProcessors();

            // 这里注册Bean的后处理器, 因为虽然Bean定义信息被载入了, 但是Bean本身并没有被
创建完成。
            registerBeanPostProcessors();

            // Initialize message source for this context.
            initMessageSource();

            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();

            // Initialize other special beans in specific context subclasses.
            onRefresh();

            // Check for listener beans and register them.
            registerListeners();

            // 这里对定义为单件的Bean进行预实例化, 是否预实例化需要依据Bean定义中的lazy-
init属性的设定
            beanFactory.preInstantiateSingletons();

            // Last step: publish corresponding event.
            publishEvent(new ContextRefreshedEvent(this));
        }
```

```
            catch (BeansException ex) {
                // Destroy already created singletons to avoid dangling resources.
                beanFactory.destroySingletons();
                throw ex;
            }
        }
    }
```

对Bean定义载入过程, 我们可以看看refreshBeanFactory,在AbstractRefreshableApplicationContext中:

```
    protected final void refreshBeanFactory() throws BeansException {

        synchronized (this.beanFactoryMonitor) {
            if (this.beanFactory != null) {
                this.beanFactory.destroySingletons();
                this.beanFactory = null;
            }
        }

        // 这里是Bean工厂的初始化过程
        try {
            //首先创建一个持有的Bean工厂, 这里使用的是DefaultListableBeanFactory
            // 具体的创建过程在上面我们分析过了
            DefaultListableBeanFactory beanFactory = createBeanFactory();

            //这里是调用BeanDefinitionReader载入Bean定义的地方
            loadBeanDefinitions(beanFactory);

            synchronized (this.beanFactoryMonitor) {
                this.beanFactory = beanFactory;
            }
            if (logger.isInfoEnabled()) {
                logger.info("Bean factory for application context [" + getDisplayName() + "]:
" + beanFactory);
            }
        }
        catch (IOException ex) {
            throw new ApplicationContextException(
                    "I/O error parsing XML document for application context [" +
getDisplayName() + "]", ex);
        }
    }
```

具体的载入实现在AbstractXmlApplicationContext中可以看到:

```
    protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws
IOException {
        // 这里创建XmlBeanDefinitionReader作为Bean定义信息的载入器, 同时这个载入器需要一个
Bean工厂的引用, 这样可以最后向它注册载入的Bean定义信息
        XmlBeanDefinitionReader beanDefinitionReader = new
XmlBeanDefinitionReader(beanFactory);

        // Configure the bean definition reader with this context's
        // resource loading environment.
        beanDefinitionReader.setResourceLoader(this);
        beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));

        // 这里对载入器进行初始化, 并载入Bean定义信息
```

```
        initBeanDefinitionReader(beanDefinitionReader);
        loadBeanDefinitions(beanDefinitionReader);
    }
```
我们看看在XmlBeanDefinitionReader是怎样载入Bean定义信息的, 值得注意的是这里已经对定位好的Resource起作用:
```
    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());
        }

        try {
            //这里从Resource中取得Bean定义信息的输入流
            InputStream inputStream = encodedResource.getResource().getInputStream();
            try {
                //这里对输入流进行XML解析
                InputSource inputSource = new InputSource(inputStream);
                if (encodedResource.getEncoding() != null) {
                    inputSource.setEncoding(encodedResource.getEncoding());
                }
                //这里是实际的解析过程
                return doLoadBeanDefinitions(inputSource, encodedResource.getResource());
            }
            finally {
                inputStream.close();
            }
        }
        catch (IOException ex) {
            throw new BeanDefinitionStoreException(
                    "IOException parsing XML document from " +
encodedResource.getResource(), ex);
        }
    }

    protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
            throws BeanDefinitionStoreException {
        try {
            int validationMode = getValidationModeForResource(resource);
            //这里使用DocumentLoader来对XML形式的Bean定义信息进行读入, 转换成DOM数据

            Document doc = this.documentLoader.loadDocument(
                    inputSource, this.entityResolver, this.errorHandler, validationMode,
this.namespaceAware);
            //得到DOM数据后进行的具体解析和注册工作
            return registerBeanDefinitions(doc, resource);
        }
        .............
    }
```
这个DomcumentReader是一个XML的读入器, 被定义为:
```
    private DocumentLoader documentLoader = new DefaultDocumentLoader();
```
作用是解析把XML文档转化为DOM结构信息;我们接着看registerBeanDefinition:
```
    public int registerBeanDefinitions(Document doc, Resource resource) throws
BeanDefinitionStoreException {
        // 这里是为了后向版本兼容使用的XML解析器
        if (this.parserClass != null) {
```

```
            XmlBeanDefinitionParser parser =
                    (XmlBeanDefinitionParser) BeanUtils.instantiateClass(this.parserClass);
            return parser.registerBeanDefinitions(this, doc, resource);
        }
        // 这里定义Bean定义信息的解析器, 这个解析器根据Spring的Bean定义信息的XML格式和规则
对Bean定义进行解析,
        // 得到BeanDefinition向IOC容器进行注册
        BeanDefinitionDocumentReader documentReader =
createBeanDefinitionDocumentReader();
        int countBefore = getBeanFactory().getBeanDefinitionCount();
        documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
        return getBeanFactory().getBeanDefinitionCount() - countBefore;
    }
```
这里看到的BeanDefinitionDocumentReader是一个根据Spring的Bean定义规则解析Bean定义的主要类之一:
```
    //这个函数刻画了主要的解析过程
    public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext)
{
        this.readerContext = readerContext;

        logger.debug("Loading bean definitions");
        //这里得到Bean定义信息DOM结构的根节点
        Element root = doc.getDocumentElement();
        //这个BeanDefinitionParserDelegate是一个重要的辅助类, 它实现了对具体Bean元素在
Bean定义信息的解析
        BeanDefinitionParserDelegate delegate = createHelper(readerContext, root);

        preProcessXml(root);
        //这是主要的解析过程
        parseBeanDefinitions(root, delegate);

        postProcessXml(root);
    }
```
下面的函数对DOM形式的定义信息进行逐个的解析,
```
    protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate
delegate) {
        if (delegate.isDefaultNamespace(root.getNamespaceURI())) {
            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;
                    String namespaceUri = ele.getNamespaceURI();
                    if (delegate.isDefaultNamespace(namespaceUri)) {
                        //这里对Spring定义的默认元素进行解析, 包括Bean, Import等等
                        parseDefaultElement(ele, delegate);
                    }
                    else {
                        delegate.parseCustomElement(ele);
                    }
                }
            }
        } else {
            delegate.parseCustomElement(root);
        }
    }
```

```
   private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate
delegate) {
      //这里对XML的Bean定义信息中的Import元素进行解析
      if (DomUtils.nodeNameEquals(ele, IMPORT_ELEMENT)) {
         importBeanDefinitionResource(ele);
      }
      else if (DomUtils.nodeNameEquals(ele, ALIAS_ELEMENT)) {
         String name = ele.getAttribute(NAME_ATTRIBUTE);
         String alias = ele.getAttribute(ALIAS_ATTRIBUTE);
         getReaderContext().getReader().getBeanFactory().registerAlias(name, alias);
         getReaderContext().fireAliasRegistered(name, alias, extractSource(ele));
      }
      //这里是对Bean元素进行解析的地方, 我们可以看到具体的解析过程是交给
BeanDefinitionParserDelegate来完成的
      else if (DomUtils.nodeNameEquals(ele, BEAN_ELEMENT)) {
         BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
         if (bdHolder != null) {
            bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
            //解析完以后得到BeanDefinition数据结构的数据交给Bean工厂进行注册
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
getReaderContext().getRegistry());
            //在注册完以后, 向上下文发送事件
            getReaderContext().fireComponentRegistered(new
BeanComponentDefinition(bdHolder));
         }
      }
   }
```

这里我们可以看看具体的Bean元素是怎样根据Spring定义的规则得到BeanDefinition数据结构的, 代码实现在BeanDefinitionParserDelegate中,这个辅助类对B
ean定义信息, 以及其中对属性的定义, 包括List, Map,Set, ref bean等Spring支持的Bean属性定义都进行了处理,最后生成一个BeanDefiniton对象来记录这些解析出来的信息, 同时对需要管理的依赖关系也进行了基本的解析 - 通过BeanDefiniton对这些依赖关系进行了记录, 在IOC容器进行依赖注入的时候, 需要使用到这些依赖信息, 这些我们在后面分析IOC容器怎样实现依赖注入的时候可以看到。这里我们简单看看一个Bean的BeanDefinition是怎样生成的:

```
   public BeanDefinitionHolder parseBeanDefinitionElement(Element ele, BeanDefinition
containingBean) {
      //这里从DOM中取得定义的id, name属性值
      String id = ele.getAttribute(ID_ATTRIBUTE);
      String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);

      //这里对建立Bean的别名集合
      List aliases = new ArrayList();
      if (StringUtils.hasLength(nameAttr)) {
         String[] nameArr = StringUtils.tokenizeToStringArray(nameAttr,
BEAN_NAME_DELIMITERS);
         aliases.addAll(Arrays.asList(nameArr));
      }
      //对重名的Bean进行检查
      String beanName = id;
      if (!StringUtils.hasText(beanName) && !aliases.isEmpty()) {
         beanName = (String) aliases.remove(0);
         if (logger.isDebugEnabled()) {
            logger.debug("No XML 'id' specified - using '" + beanName +
                  "' as bean name and " + aliases + " as aliases");
```

```
            }
        }

        if (containingBean == null) {
            checkNameUniqueness(beanName, aliases, ele);
        }
        //这里对Bean定义进行符合Spring定义规则的解析
        AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele,
beanName, containingBean);
        if (beanDefinition != null) {
            if (!StringUtils.hasText(beanName)) {
                beanName = BeanDefinitionReaderUtils.generateBeanName(
                        beanDefinition, getReaderContext().getReader().getBeanFactory(),
(containingBean != null));
                if (logger.isDebugEnabled()) {
                    logger.debug("Neither XML 'id' nor 'name' specified - " +
                            "using generated bean name [" + beanName + "]");
                }
            }
            //返回的BeanDefintion会被注册到IOC容器里去
            String[] aliasesArray = StringUtils.toStringArray(aliases);
            return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);
        }

        return null;
    }
```

下面这个方法比较长，包含了对Spring定义Bean的各种属性值得解析处理，比如class,scope等等。

```
    public AbstractBeanDefinition parseBeanDefinitionElement(
            Element ele, String beanName, BeanDefinition containingBean) {

        //这里取得Bean定义中的CLASS属性
        String className = null;
        if (ele.hasAttribute(CLASS_ATTRIBUTE)) {
            className = ele.getAttribute(CLASS_ATTRIBUTE);
        }
        //这里取得parent属性
        String parent = null;
        if (ele.hasAttribute(PARENT_ATTRIBUTE)) {
            parent = ele.getAttribute(PARENT_ATTRIBUTE);
        }

        try {
            this.parseState.push(new BeanEntry(beanName));
             //注意这个BeanDefinition的创建过程，会使用当前的Classloader创建一个class属性定义
的JAVA对象实例
             //这个JAVA对象是Bean定义的一部分，是一个纯粹的JAVA对象，还不包含依赖注入的其他
对象
            AbstractBeanDefinition bd = BeanDefinitionReaderUtils.createBeanDefinition(
                    parent, className,
getReaderContext().getReader().getBeanClassLoader());

            //这里在BeanDefintion中设置scope属性，比如singleton和prototype属性的设置
            if (ele.hasAttribute(SCOPE_ATTRIBUTE)) {
                // Spring 2.0 "scope" attribute
                bd.setScope(ele.getAttribute(SCOPE_ATTRIBUTE));
                if (ele.hasAttribute(SINGLETON_ATTRIBUTE)) {
```

```
                error("Specify either 'scope' or 'singleton', not both", ele);
            }
        }
        else if (ele.hasAttribute(SINGLETON_ATTRIBUTE)) {
            // Spring 1.x "singleton" attribute

bd.setSingleton(TRUE_VALUE.equals(ele.getAttribute(SINGLETON_ATTRIBUTE)));
        }
        else if (containingBean != null) {
            // Take default from containing bean in case of an inner bean definition.
            bd.setSingleton(containingBean.isSingleton());
        }

        if (ele.hasAttribute(ABSTRACT_ATTRIBUTE)) {

bd.setAbstract(TRUE_VALUE.equals(ele.getAttribute(ABSTRACT_ATTRIBUTE)));
        }
        //这里设置lazy-init属性, 这个属性对预实例化进行控制
        String lazyInit = ele.getAttribute(LAZY_INIT_ATTRIBUTE);
        if (DEFAULT_VALUE.equals(lazyInit) && bd.isSingleton()) {
            // Just apply default to singletons, as lazy-init has no meaning for prototypes.
            lazyInit = getDefaultLazyInit();
        }
        bd.setLazyInit(TRUE_VALUE.equals(lazyInit));

        if (ele.hasAttribute(AUTOWIRE_CANDIDATE_ATTRIBUTE)) {

bd.setAutowireCandidate(TRUE_VALUE.equals(ele.getAttribute(AUTOWIRE_CANDIDATE_ATTRIBUTE)));
        }
        //这里对auto-wire属性进行设置
        String autowire = ele.getAttribute(AUTOWIRE_ATTRIBUTE);
        if (DEFAULT_VALUE.equals(autowire)) {
            autowire = getDefaultAutowire();
        }
        bd.setAutowireMode(getAutowireMode(autowire));

        String dependencyCheck = ele.getAttribute(DEPENDENCY_CHECK_ATTRIBUTE);
        if (DEFAULT_VALUE.equals(dependencyCheck)) {
            dependencyCheck = getDefaultDependencyCheck();
        }
        bd.setDependencyCheck(getDependencyCheck(dependencyCheck));

        if (ele.hasAttribute(DEPENDS_ON_ATTRIBUTE)) {
            String dependsOn = ele.getAttribute(DEPENDS_ON_ATTRIBUTE);
            bd.setDependsOn(StringUtils.tokenizeToStringArray(dependsOn,
BEAN_NAME_DELIMITERS));
        }
        //是否使用工厂方法来生成bean的JAVA对象
        if (ele.hasAttribute(FACTORY_METHOD_ATTRIBUTE)) {

bd.setFactoryMethodName(ele.getAttribute(FACTORY_METHOD_ATTRIBUTE));
        }
        if (ele.hasAttribute(FACTORY_BEAN_ATTRIBUTE)) {
            bd.setFactoryBeanName(ele.getAttribute(FACTORY_BEAN_ATTRIBUTE));
        }
        //bean初始化方法的设置
        if (ele.hasAttribute(INIT_METHOD_ATTRIBUTE)) {
```

```
            String initMethodName = ele.getAttribute(INIT_METHOD_ATTRIBUTE);
            if (!"".equals(initMethodName)) {
                bd.setInitMethodName(initMethodName);
            }
        }
        else {
            if (getDefaultInitMethod() != null) {
                bd.setInitMethodName(getDefaultInitMethod());
                bd.setEnforceInitMethod(false);
            }
        }

        if (ele.hasAttribute(DESTROY_METHOD_ATTRIBUTE)) {
            String destroyMethodName =
ele.getAttribute(DESTROY_METHOD_ATTRIBUTE);
            if (!"".equals(destroyMethodName)) {
                bd.setDestroyMethodName(destroyMethodName);
            }
        }
        else {
            if (getDefaultDestroyMethod() != null) {
                bd.setDestroyMethodName(getDefaultDestroyMethod());
                bd.setEnforceDestroyMethod(false);
            }
        }

        //这些属性设置在BeanDefinition完成以后，具体的处理在下面完成
        parseMetaElements(ele, bd);
        parseLookupOverrideSubElements(ele, bd.getMethodOverrides());
        parseReplacedMethodSubElements(ele, bd.getMethodOverrides());

        parseConstructorArgElements(ele, bd);
        //这里对property元素的设定进行处理，为依赖注入做好准备
        parsePropertyElements(ele, bd);

        bd.setResourceDescription(getReaderContext().getResource().getDescription());
        bd.setSource(extractSource(ele));

        return bd;
      }
      catch (ClassNotFoundException ex) {
        error("Bean class [" + className + "] not found", ele, ex);
      }
      catch (NoClassDefFoundError err) {
        error("Class that bean class [" + className + "] depends on not found", ele,
err);
      }
      catch (Throwable ex) {
        error("Unexpected failure during bean definition parsing", ele, ex);
      }
      finally {
        this.parseState.pop();
      }

      return null;
    }
```

这里已经生成了BeanDefinition并把在bean定义信息中的设置反映到这个数据结构里了，我们下面看

看和IOC容器管理依赖关系紧密相关的property属性是怎么被处理的:

```
   public void parsePropertyElements(Element beanEle, BeanDefinition bd) {
      //从Bean定义信息中得到property的设置, 设定一个循环对所有的property设置进行处理
      NodeList nl = beanEle.getChildNodes();
      for (int i = 0; i < nl.getLength(); i++) {
         Node node = nl.item(i);
         if (node instanceof Element && DomUtils.nodeNameEquals(node,
PROPERTY_ELEMENT)) {
            //这里是对单个property进行处理的地方
            parsePropertyElement((Element) node, bd);
         }
      }
   }

   public void parsePropertyElement(Element ele, BeanDefinition bd) {
      //得到property的名字
      String propertyName = ele.getAttribute(NAME_ATTRIBUTE);
      if (!StringUtils.hasLength(propertyName)) {
         error("Tag 'property' must have a 'name' attribute", ele);
         return;
      }
      this.parseState.push(new PropertyEntry(propertyName));
      try {
         //在BeanDefinition对所有已经处理过的property都保存在propertyValues这个集合中, 这
里先看看是不是已经处理过同名的property值
         if (bd.getPropertyValues().contains(propertyName)) {
            error("Multiple 'property' definitions for property '" + propertyName + "'",
ele);
            return;
         }
         //这里对各种property的定义进行处理, 比如ref bean,List,Map,Set等等
         //然后构造一个PropertyValue对象放到BeanDefinition的PropertyValues集合中去, 这些
信息在依赖注入的时候会被使用到
         Object val = parsePropertyValue(ele, bd, propertyName);
         PropertyValue pv = new PropertyValue(propertyName, val);
         parseMetaElements(ele, pv);
         pv.setSource(extractSource(ele));
         bd.getPropertyValues().addPropertyValue(pv);
      }
      finally {
         this.parseState.pop();
      }
   }
```

我们下面看看对各种property定义是怎样处理的, 比如ref bean,List, Map等, 这里分别用了parsePropertyValue,parseSubElement来处理对ref bean和子属性的处理, 对ref bean是使用了一个RuntimeBeanReference的对象作为一个占位符来代表需要依赖的Bean,而这个Bean的数据在另外一个BeanDefinition中, 这里只是对依赖关系作了一个记录, 具体的依赖注入在向容器请求bean的时候发生, 对其他List,Map对象, 这里直接根据设置来生成:

```
   public Object parsePropertyValue(Element ele, BeanDefinition bd, String
propertyName) {
      String elementName = (propertyName != null) ?
                  "<property> element for property '" + propertyName + "'" :
                  "<constructor-arg> element";

      // Should only have one child element: ref, value, list, etc.
      NodeList nl = ele.getChildNodes();
      Element subElement = null;
```

```
        for (int i = 0; i < nl.getLength(); i++) {
            if (nl.item(i) instanceof Element) {
                Element candidateEle = (Element) nl.item(i);
                if (DESCRIPTION_ELEMENT.equals(candidateEle.getTagName())) {
                    // Keep going: we don't use this value for now.
                }
                else {
                    // Child element is what we're looking for.
                    if (subElement != null &&
!META_ELEMENT.equals(subElement.getTagName())) {
                        error(elementName + " must not contain more than one sub-element",
ele);
                    }
                    subElement = candidateEle;
                }
            }
        }
        //这里判读是不是一个ref bean的属性
        boolean hasRefAttribute = ele.hasAttribute(REF_ATTRIBUTE);
        boolean hasValueAttribute = ele.hasAttribute(VALUE_ATTRIBUTE);
        if ((hasRefAttribute && hasValueAttribute) ||
                ((hasRefAttribute || hasValueAttribute)) && subElement != null) {
            error(elementName +
                    " is only allowed to contain either 'ref' attribute OR 'value' attribute OR
sub-element", ele);
        }
        if (hasRefAttribute) {
            String refName = ele.getAttribute(REF_ATTRIBUTE);
            if (!StringUtils.hasText(refName)) {
                error(elementName + " contains empty 'ref' attribute", ele);
            }
            //如果属性是ref bean属性, 构造一个RuntimeBeanReference来代表这两个bean之间的
依赖关系
            RuntimeBeanReference ref = new RuntimeBeanReference(refName);
            ref.setSource(extractSource(ele));
            return ref;
        }
        else if (hasValueAttribute) {
            return ele.getAttribute(VALUE_ATTRIBUTE);
        }

        if (subElement == null) {
            // Neither child element nor "ref" or "value" attribute found.
            error(elementName + " must specify a ref or value", ele);
        }
        //这里是对子属性的处理
        return parsePropertySubElement(subElement, bd);
    }
下面对子属性的处理, 包含了各种属性类型:
    public Object parsePropertySubElement(Element ele, BeanDefinition bd, String
defaultTypeClassName) {
        if (!isDefaultNamespace(ele.getNamespaceURI())) {
            return parseNestedCustomElement(ele, bd);
        }//如果是一个bean属性, 需要调用对BeanDefinition的解析过程
        else if (DomUtils.nodeNameEquals(ele, BEAN_ELEMENT)) {
            return parseBeanDefinitionElement(ele, bd);
        }//如果是一个ref属性, 和上面一样构造一个RuntimeBeanReference来代表持有的bean的依
```

赖关系

```
        else if (DomUtils.nodeNameEquals(ele, REF_ELEMENT)) {
            // A generic reference to any name of any bean.
            String refName = ele.getAttribute(BEAN_REF_ATTRIBUTE);
            boolean toParent = false;
            if (!StringUtils.hasLength(refName)) {
                // A reference to the id of another bean in the same XML file.
                refName = ele.getAttribute(LOCAL_REF_ATTRIBUTE);
                if (!StringUtils.hasLength(refName)) {
                    // A reference to the id of another bean in a parent context.
                    refName = ele.getAttribute(PARENT_REF_ATTRIBUTE);
                    toParent = true;
                    if (!StringUtils.hasLength(refName)) {
                        error("'bean', 'local' or 'parent' is required for <ref> element", ele);
                    }
                }
            }
            if (!StringUtils.hasText(refName)) {
                error("<ref> element contains empty target attribute", ele);
            }
            RuntimeBeanReference ref = new RuntimeBeanReference(refName, toParent);
            ref.setSource(extractSource(ele));
            return ref;
        }
        else if (DomUtils.nodeNameEquals(ele, IDREF_ELEMENT)) {
            // A generic reference to any name of any bean.
            String beanRef = ele.getAttribute(BEAN_REF_ATTRIBUTE);
            if (!StringUtils.hasLength(beanRef)) {
                // A reference to the id of another bean in the same XML file.
                beanRef = ele.getAttribute(LOCAL_REF_ATTRIBUTE);
                if (!StringUtils.hasLength(beanRef)) {
                    error("Either 'bean' or 'local' is required for <idref> element", ele);
                }
            }
            RuntimeBeanNameReference ref = new RuntimeBeanNameReference(beanRef);
            ref.setSource(extractSource(ele));
            return ref;
        }//如果子属性是value,比如各种List里的持有的字符串数据, 在这里处理, 返回的是String类型
的字符串
        else if (DomUtils.nodeNameEquals(ele, VALUE_ELEMENT)) {
            // It's a literal value.
            String value = DomUtils.getTextValue(ele);
            String typeClassName = ele.getAttribute(TYPE_ATTRIBUTE);
            if (!StringUtils.hasText(typeClassName)) {
                typeClassName = defaultTypeClassName;
            }
            if (StringUtils.hasText(typeClassName)) {
                try {
                    return buildTypedStringValue(value, typeClassName);
                }
                catch (ClassNotFoundException ex) {
                    error("Type class [" + typeClassName + "] not found for <value>
element", ele, ex);
                }
            }
            return value;
        }
```

```
        else if (DomUtils.nodeNameEquals(ele, NULL_ELEMENT)) {
            // It's a distinguished null value.
            return null;
        }//这里是对List子属性的处理
        else if (DomUtils.nodeNameEquals(ele, LIST_ELEMENT)) {
            return parseListElement(ele, bd);
        }//这里对Set子属性的处理
        else if (DomUtils.nodeNameEquals(ele, SET_ELEMENT)) {
            return parseSetElement(ele, bd);
        }//这里对Map子属性的处理
        else if (DomUtils.nodeNameEquals(ele, MAP_ELEMENT)) {
            return parseMapElement(ele, bd);
        }//这里对Prop子属性的处理
        else if (DomUtils.nodeNameEquals(ele, PROPS_ELEMENT)) {
            return parsePropsElement(ele);
        }
        error("Unknown property sub-element: [" + ele.getTagName() + "]", ele);
        return null;
    }
```

我们看看在一个List属性的处理是怎样的:

```
    public List parseListElement(Element collectionEle, BeanDefinition bd) {
        String defaultTypeClassName =
collectionEle.getAttribute(VALUE_TYPE_ATTRIBUTE);
        NodeList nl = collectionEle.getChildNodes();
        //这里先构建一个List作为这个属性对应的JAVA对象
        ManagedList list = new ManagedList(nl.getLength());
        list.setSource(extractSource(collectionEle));
        list.setMergeEnabled(parseMergeAttribute(collectionEle));
        //这里对这个List含有的元素进行处理, 因为有可能含有各种各样的元素类型, 这里使用了
parsePropertySubElement来处理
        //在上面对parsePropertySubElement的分支中我们可以看到各种子属性的处理, 比如ref, 各
种value,还可能有List, Set等 - 总之可以迭代到底
        for (int i = 0; i < nl.getLength(); i++) {
            if (nl.item(i) instanceof Element) {
                Element ele = (Element) nl.item(i);
                list.add(parsePropertySubElement(ele, bd, defaultTypeClassName));
            }
        }
        return list;
    }
```

以上就是整个对Bean定义信息进行处理的过程, 在Bean定义信息的载入过程中, Bean定义信息被抽象到BeanDefinition这个数据结构中, 同时bean定义信息中的各种元素都得到了有效的解析, 比如ref bean会使用一个RuntimeBeanReference对象来代表相应的依赖关系, List属性会有一个相应的填充好定义值得List对象来代表等等。这些属性值在BeanDefinition中被一个propertyValues的集合来持有。在依赖注入的时候, 这个BeanDefinition中已经包含所有IOC容器需要的用户在Bean定义信息中定义的数据。也标志着整个载入过程的完成。

Bean定义信息的注册

在IOC容器里建立好Bean定义信息对应的BeanDefinition数据结构以后, 持有这个数据结构的时BeanDefinitionReader, 因为读入器和IOC容器是分开实现的, 所以在初始化IOC容器的过程中, 读入器需要完成一个向DefaultListableBeanFactory的BeanDefinitionRegistry接口的回调完成BeanDefinition在IOC容器的注册。这样整个持有Bean定义信息的IOC容器就建立起来了, 让我们回到

代码XmlBeanDefinitionReader的parseDefaultElement:

```
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
getReaderContext().getRegistry());
```

我们看看BeanDefinitionReaderUtils里面的实现:

```
    public static void registerBeanDefinition(
            BeanDefinitionHolder bdHolder, BeanDefinitionRegistry beanFactory) throws
BeansException {

        // 在注册的时候, 使用的key是BeanDefinition的BeanName
        String beanName = bdHolder.getBeanName();
        //这里调用BeanFactory来向IOC容器自己注册, 在初始化XmlBeanDefinitionReader的时候,
需要给Reader指定一个B
eanFactory,这个BeanFactory就是在注册里面用到, 这样可以把Bean定义和注册两个过程分开, 为
Reader和容器的使用提供了灵活性
        beanFactory.registerBeanDefinition(beanName, bdHolder.getBeanDefinition());

        // 这里向IOC容器注册Bean的别名集合
        String[] aliases = bdHolder.getAliases();
        if (aliases != null) {
            for (int i = 0; i < aliases.length; i++) {
                beanFactory.registerAlias(beanName, aliases[i]);
            }
        }
    }
```

我们看看XmlBeanFactory中的注册实现:

```
    //---------------------------------------------------------------------
    // 这里是IOC容器对BeanDefinitionRegistry接口的实现
    //---------------------------------------------------------------------

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

        .....//这里省略了对BeanDefinition的验证过程
        //先看看在容器里是不是已经有了同名的bean,如果有抛出异常。
        Object oldBeanDefinition = this.beanDefinitionMap.get(beanName);
        if (oldBeanDefinition != null) {
            if (!this.allowBeanDefinitionOverriding) {
            ..........
        }
        else {
            //把bean的名字加到IOC容器中去
            this.beanDefinitionNames.add(beanName);
        }
        //这里把bean的名字和Bean定义联系起来放到一个HashMap中去,IOC容器通过这个Map来维
护容器里的Bean定义信息。
        this.beanDefinitionMap.put(beanName, beanDefinition);
        removeSingleton(beanName);
    }
```

这样就完成了Bean定义在IOC容器中的注册, 就可被IOC容器进行管理和使用了。这里我们看到了在DefaultListableBeanFactory里定义的数据结构被使用来保存Bean定义信息的完整过程, 也就是我们上面分析的整个Bean定义信息的载入和注册过程:

```
public class DefaultListableBeanFactory extends AbstractAutowireCapableBeanFactory
    implements ConfigurableListableBeanFactory, BeanDefinitionRegistry {
    ......

    /** 这里就是存放载入Bean定义信息的地方, 以Bean的名字作为key来检索Bean定义信息 */
```

```
    private final Map beanDefinitionMap = new HashMap();

    /** 这个列表保存经过排序的Bean的名字 */
    private final List beanDefinitionNames = new ArrayList();

    .......
}
```

通过上面的分析, 我们可以总结一下IOC容器初始化的基本步骤:

- 初始化的入口在容器实现中的refresh()调用来完成
- 对bean 定义载入IOC容器使用的方法是loadBeanDefinition,其中的大致过程如下:通过ResourceLoader来完成资源文件位置的定 位, DefaultResourceLoader是默认的实现, 同时上下文本身就给出了ResourceLoader的实现, 可以从类路径, 文件系统, URL等方式来定为资源位置。如果是XmlBeanFactory作为IOC容器, 那么需要为它指定bean定义的资源, 也就是说bean定义文件时通过 抽象成Resource来被IOC容器处理的, 容器通过BeanDefinitionReader来完成定义信息的解析和Bean信息的注册,往往使用的 是XmlBeanDefinitionReader来解析bean的xml定义文件 - 实际的处理过程是委托给BeanDefinitionParserDelegate来完成的, 从而得到bean的定义信息, 这些信息在Spring中使用BeanDefinition对象来表示 - 这个名字可以让我们想到loadBeanDefinition,RegisterBeanDefinition这些相关的方法 - 他们都是为处理BeanDefinitin服务的, IoC容器解析得到BeanDefinition以后, 需要把它在IOC容器中注册, 这由IOC实现 BeanDefinitionRegistry接口来实现。注册过程就是在IOC容器内部维护的一个HashMap来保存得到的 BeanDefinition的过程。这个HashMap是IoC容器持有bean信息的场所, 以后对bean的操作都是围绕这个HashMap来实现 的。
- 然后我们就可以通过BeanFactory和ApplicationContext来享受到Spring IOC的服务了.

IOC容器的依赖注入实现

我们在上面对上下文的初始化过程作了一个详细的分析, 这个过程主要完成的是在IOC容器中建立Bean定义信息映射的过程 , 在这个过程中并没有看到IOC容器对Bean依赖关系进行注入的处理 - 我们看到在处理相关的Bean属性的时候, 使用了RuntimeBeanReference对象作为依赖信息的纪录, 下面我们看看IOC容器是怎样对Bean的依赖关系进行注入的。

下面的代码分析假设当前IOC容器已经载入了用户定义的Bean信息, 这个依赖注入的过程是用户第一次向IOC容器索要Bean的时候触发的, 当然也有例外就是我们可以在Bean定义信息中通过控制lazy-init属性来使得容器完成对Bean的预实例化 - 这个预实例化也是一个完成依赖注入的过程, 稍后我们会详细的进行分析。

在用户向IOC容器索要Bean的时候, 代码入口在DefaultListableBeanFactory的基类AbstractBeanFactory中:

```
    public Object getBean(String name, Class requiredType, final Object[] args) throws BeansException {
        final String beanName = transformedBeanName(name);
        Object bean = null;

        // Eagerly check singleton cache for manually registered singletons.
        // 这里先从缓存中去取,处理那些已经被创建过的单件模式的bean, 对这种bean的请求不需要重复的去创建
        Object sharedInstance = getSingleton(beanName);
        if (sharedInstance != null) {
            if (isSingletonCurrentlyInCreation(beanName)) {
            .......
            }
            else {
            .......
```

```
            }
            //这里的getObjectForBeanInstance完成的是FactoryBean的相关处理
            if (containsBeanDefinition(beanName)) {
                RootBeanDefinition mergedBeanDefinition =
getMergedBeanDefinition(beanName, false);
                bean = getObjectForBeanInstance(sharedInstance, name,
mergedBeanDefinition);
            }
            else {
                bean = getObjectForBeanInstance(sharedInstance, name, null);
            }
        }

        else {
            // Fail if we're already creating this singleton instance:
            // We're assumably within a circular reference.
            if (isSingletonCurrentlyInCreation(beanName)) {
                throw new BeanCurrentlyInCreationException(beanName);
            }

            // Check if bean definition exists in this factory.
            //这里检查是否能在当前的工厂中取到我们需要的bean, 如果在当前的工厂中取不到, 则到父
工厂取, 如果一直取不到
            //那就顺着工厂链一直向上查找
            BeanFactory parentBeanFactory = getParentBeanFactory();
            if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
                // Not found -> check parent.
                String nameToLookup = originalBeanName(name);

                // 这里调用父工厂的getbean取需要的bean
                // 这里有一个迭代, 在父工厂中也会重复这么一个getbean的过程。
                if (parentBeanFactory instanceof AbstractBeanFactory) {
                    // Delegation to parent with args only possible for AbstractBeanFactory.

                    return ((AbstractBeanFactory)
parentBeanFactory).getBean(nameToLookup, requiredType, args);
                }
                else if (args == null) {
                    // No args -> delegate to standard getBean method.
                    return parentBeanFactory.getBean(nameToLookup, requiredType);
                }
                else {
                    throw new NoSuchBeanDefinitionException(beanName,
                            "Cannot delegate to parent BeanFactory because it does not
supported passed-in arguments");
                }
            }
            //把这个已经被要求过的bean记录下来, 因为第一次要求bean的时候往往就是依赖被容器对
bean进行注入的时候。
            this.alreadyCreated.add(beanName);

            final RootBeanDefinition mergedBeanDefinition =
getMergedBeanDefinition(beanName, false);
            checkMergedBeanDefinition(mergedBeanDefinition, beanName, args);

            //这里是根据已经载入的beandefinition来创建bean和完成依赖注入的地方。
            if (mergedBeanDefinition.isSingleton()) {
```

```
            sharedInstance = getSingleton(beanName, new ObjectFactory() {
                public Object getObject() throws BeansException {
                    try {
                        //注意这个createBean, 是创建bean同时完成依赖注入的地方。
                        return createBean(beanName, mergedBeanDefinition, args);
                    }
                    catch (BeansException ex) {
                        destroySingleton(beanName);
                        throw ex;
                    }
                }
            });
            bean = getObjectForBeanInstance(sharedInstance, name,
mergedBeanDefinition);
        }
        //这里是处理prototype类型的bean请求的地方
        else if (mergedBeanDefinition.isPrototype()) {
            // It's a prototype -> create a new instance.
            Object prototypeInstance = null;
            try {
                beforePrototypeCreation(beanName);
                //每次请求, 直接通过createBean来创建
                prototypeInstance = createBean(beanName, mergedBeanDefinition, args);
            }
            finally {
                afterPrototypeCreation(beanName);
            }
            bean = getObjectForBeanInstance(prototypeInstance, name,
mergedBeanDefinition);
        }
       ........
       return bean;
   }
```

具体的bean创建过程和依赖关系的注入在createBean中, 这个方法在AbstractAutowireCapableBeanFactory中给出了实现:

```
   protected Object createBean(String beanName, RootBeanDefinition
mergedBeanDefinition, Object[] args)
        throws BeanCreationException {

       // Guarantee initialization of beans that the current one depends on.
       // 这里对取得当前bean的所有依赖bean, 确定能够取得这些已经被确定的bean, 如果没有被创
建, 那么这个createBean会被这些IOC
       // getbean时创建这些bean
       if (mergedBeanDefinition.getDependsOn() != null) {
           for (int i = 0; i < mergedBeanDefinition.getDependsOn().length; i++) {
               getBean(mergedBeanDefinition.getDependsOn()[i]);
           }
       }

           ........

           // 这里是实例化bean对象的地方, 注意这个BeanWrapper类, 是对bean操作的主要封装类
           if (instanceWrapper == null) {
               instanceWrapper = createBeanInstance(beanName, mergedBeanDefinition,
args);
           }
           Object bean = instanceWrapper.getWrappedInstance();
```

```
        ......
        //这个populate方法，是对已经创建的bean实例进行依赖注入的地方，会使用到在
loadBeanDefinition的时候得到的那些propertyValue来对bean进行注入。
        if (continueWithPropertyPopulation) {
            populateBean(beanName, mergedBeanDefinition, instanceWrapper);
        }

        //这里完成客户自定义的对bean的一些初始化动作
        Object originalBean = bean;
        bean = initializeBean(beanName, bean, mergedBeanDefinition);
        // Register bean as disposable, and also as dependent on specified "dependsOn"
beans.
        registerDisposableBeanIfNecessary(beanName, originalBean,
mergedBeanDefinition);

        return bean;
    }

    .........
  }
```

我们看看是用的wrapper类是怎样被创建的，这个对象被创建的时候已经为我们的bean创建了JAVA对象：

```
  protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition
mergedBeanDefinition, Object[] args)
        throws BeansException {

    BeanWrapper instanceWrapper = null;
    //这里使用BeanWrapper的不同创建方法
    if (mergedBeanDefinition.getFactoryMethodName() != null)  {
        instanceWrapper = instantiateUsingFactoryMethod(beanName,
mergedBeanDefinition, args);
    }
    else if (mergedBeanDefinition.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_CONSTRUCTOR ||
            mergedBeanDefinition.hasConstructorArgumentValues() )  {
        instanceWrapper = autowireConstructor(beanName, mergedBeanDefinition);
    }
    else {
        // No special handling: simply use no-arg constructor.
        // 这是最正常的创建，使用Spring默认的BeanWrapper实现BeanWrapperImp
        instanceWrapper = instantiateBean(beanName, mergedBeanDefinition);
    }
    return instanceWrapper;
  }

  protected BeanWrapper instantiateBean(String beanName, RootBeanDefinition
mergedBeanDefinition)
        throws BeansException {
    //这里是创建bean对象的地方，同时把这个bean对象放到BeanWrapper中去
    Object beanInstance =
getInstantiationStrategy().instantiate(mergedBeanDefinition, beanName, this);
    BeanWrapper bw = new BeanWrapperImpl(beanInstance);
    initBeanWrapper(bw);
    return bw;
  }
```

我们注意到在这里定义的实例化的策略是

```
  private InstantiationStrategy instantiationStrategy = new
CglibSubclassingInstantiationStrategy();
```
一般而言可以直接实例化也可以通过cglib来完成bean对象的重新实例化，在CglibSubclassingInstantiationStrategy中：
```
   public Object instantiate(
         RootBeanDefinition beanDefinition, String beanName, BeanFactory owner) {

      // Don't override the class with CGLIB if no overrides.
      // 这里是重新实例化bean对象的地方，返回后放到BeanWrapper对象当中去
      if (beanDefinition.getMethodOverrides().isEmpty()) {
         return BeanUtils.instantiateClass(beanDefinition.getBeanClass());
      }
      else {
         // Must generate CGLIB subclass.
         return instantiateWithMethodInjection(beanDefinition, beanName, owner);
      }
   }
```
这里我们看到对bean的JAVA对象的创建过程，如果没有什么依赖关系的话，那主要的bean创建过程已经完成了，但是如果存在依赖关系的话，这些依赖关系还要进行注入，
回到AbstractAutowireCapableBeanFactory中的populate方法，这里是处理bean的依赖注入的地方：
```
   protected void populateBean(String beanName, RootBeanDefinition
mergedBeanDefinition, BeanWrapper bw)
         throws BeansException {
      //首先取得我们在loadBeanDefinition中取得的依赖定义propertyValues
      PropertyValues pvs = mergedBeanDefinition.getPropertyValues();
      ......

      checkDependencies(beanName, mergedBeanDefinition, filteredPds, pvs);
      //主要地依赖注入处理在这里
      applyPropertyValues(beanName, mergedBeanDefinition, bw, pvs);
   }

   private void applyPropertyValues(
         String beanName, RootBeanDefinition mergedBeanDefinition, BeanWrapper bw,
PropertyValues pvs)
         throws BeansException {

      if (pvs == null) {
         return;
      }

      BeanDefinitionValueResolver valueResolver =
            new BeanDefinitionValueResolver(this, beanName, mergedBeanDefinition);

      // Create a deep copy, resolving any references for values.
      // 这里把那些相关的有依赖关系的property内容copy过来
      MutablePropertyValues deepCopy = new MutablePropertyValues();
      PropertyValue[] pvArray = pvs.getPropertyValues();
      for (int i = 0; i < pvArray.length; i++) {
         PropertyValue pv = pvArray[i];
         //这个队property的resolve过程包含了一个对依赖bean的迭代解析和创建
         Object resolvedValue =
               valueResolver.resolveValueIfNecessary("bean property '" + pv.getName()
+ "'", pv.getValue());
         deepCopy.addPropertyValue(pvArray[i].getName(), resolvedValue);
      }
```

```
        // 这里把copy过来的propertyValue置入到BeanWrapper中去, 这个set其实并不简单, 它通过
wrapper完成了实际的依赖注入
        try {
            // Synchronize if custom editors are registered.
            // Necessary because PropertyEditors are not thread-safe.
            if (!getCustomEditors().isEmpty()) {
                synchronized (this) {
                    bw.setPropertyValues(deepCopy);
                }
            }
            else {
                bw.setPropertyValues(deepCopy);
            }
        }
        .........
    }
```

这里有一个迭代的解析和bean依赖的创建, 注入:

```
         Object resolvedValue =    valueResolver.resolveValueIfNecessary("bean property '"
+ pv.getName() + "'", pv.getValue());
```

在BeanDefinitionValueResolver中是这样实现这个resolve的:

```
    public Object resolveValueIfNecessary(String argName, Object value) throws
BeansException {

        if (value instanceof BeanDefinitionHolder) {
            // Resolve BeanDefinitionHolder: contains BeanDefinition with name and aliases.
            BeanDefinitionHolder bdHolder = (BeanDefinitionHolder) value;
            return resolveInnerBeanDefinition(argName, bdHolder.getBeanName(),
bdHolder.getBeanDefinition());
        }
        else if (value instanceof BeanDefinition) {
            // Resolve plain BeanDefinition, without contained name: use dummy name.
            BeanDefinition bd = (BeanDefinition) value;
            return resolveInnerBeanDefinition(argName, "(inner bean)", bd);
        }
        else if (value instanceof RuntimeBeanNameReference) {
            String ref = ((RuntimeBeanNameReference) value).getBeanName();
            if (!this.beanFactory.containsBean(ref)) {
                throw new BeanDefinitionStoreException(
                        "Invalid bean name '" + ref + "' in bean reference for " + argName);
            }
            return ref;
        }
        else if (value instanceof RuntimeBeanReference) {
            RuntimeBeanReference ref = (RuntimeBeanReference) value;
            return resolveReference(argName, ref);
        }
        else if (value instanceof ManagedList) {
            // May need to resolve contained runtime references.
            return resolveManagedList(argName, (List) value);
        }
        else if (value instanceof ManagedSet) {
            // May need to resolve contained runtime references.
            return resolveManagedSet(argName, (Set) value);
        }
        else if (value instanceof ManagedMap) {
            // May need to resolve contained runtime references.
```

```
            return resolveManagedMap(argName, (Map) value);
        }
        else if (value instanceof ManagedProperties) {
            Properties copy = new Properties();
            copy.putAll((Properties) value);
            return copy;
        }
        else if (value instanceof TypedStringValue) {
            // Convert value to target type here.
            TypedStringValue typedStringValue = (TypedStringValue) value;
            try {
                Class resolvedTargetType = resolveTargetType(typedStringValue);
                return this.beanFactory.doTypeConversionIfNecessary(
                        this.typeConverter, typedStringValue.getValue(), resolvedTargetType,
null);
            }
            catch (Throwable ex) {
                // Improve the message by showing the context.
                throw new BeanCreationException(
                        this.beanDefinition.getResourceDescription(), this.beanName,
                        "Error converting typed String value for " + argName, ex);
            }
        }
        else {
            // No need to resolve value...
            return value;
        }
    }
```

这里可以看到对各种依赖类型的resolve,我们看看怎样解析reference bean的, 非常清楚的向IOC容器去请求 - 也许会触发下一层依赖的bean的创建和依赖注入过程:

```
    private Object resolveReference(String argName, RuntimeBeanReference ref) throws
BeansException {
        ..........
        try {
                //向父工厂请求bean
                return
this.beanFactory.getParentBeanFactory().getBean(ref.getBeanName());
            }
            else {
                //向自己所在的工厂请求bean
                Object bean = this.beanFactory.getBean(ref.getBeanName());
                if (this.beanDefinition.isSingleton()) {
                    this.beanFactory.registerDependentBean(ref.getBeanName(),
this.beanName);
                }
                return bean;
            }
        }
     .........
    }
```

假 设我们经过穷推, 已经到最后一层的bean的依赖创建和注入, 这个具体的注入过程, 也就是依赖注入过程要依靠BeanWrapperImp的实现我们回到 applyPropertyValues中来, 这里是已经迭代对依赖进行完解析的地方, 也就是需要对依赖进行注入的地方 - 注意这个token是在wrapper中已经对属性做过处理了:

```
    private void setPropertyValue(PropertyTokenHolder tokens, Object newValue) throws
BeansException {
        String propertyName = tokens.canonicalName;
```

```
    if (tokens.keys != null) {
        // Apply indexes and map keys: fetch value for all keys but the last one.
        PropertyTokenHolder getterTokens = new PropertyTokenHolder();
        getterTokens.canonicalName = tokens.canonicalName;
        getterTokens.actualName = tokens.actualName;
        getterTokens.keys = new String[tokens.keys.length - 1];
        System.arraycopy(tokens.keys, 0, getterTokens.keys, 0, tokens.keys.length -
1);

        //这里取得需要的propertyValue, 这个getPropertyValue同样不简单, 这个getProperty
实际上已经取出了bean对象中的属性引用
        //所以下面可以直接把依赖对象注入过去
        Object propValue = null;
        try {
            propValue = getPropertyValue(getterTokens);
        }
        ........

        // 如果根据token取不到propertyValue, 直接抛出异常
        String key = tokens.keys[tokens.keys.length - 1];
        if (propValue == null) {
            throw new NullValueInNestedPathException(getRootClass(), this.nestedPath
+ propertyName,
                    "Cannot access indexed value in property referenced " +
                    "in indexed property path '" + propertyName + "': returned null");
        } //这里处理属性是List的注入
        else if (propValue.getClass().isArray()) {
              .....
              Array.set(propValue, Integer.parseInt(key), convertedValue);
              .......
        } //这里处理对List的注入
        else if (propValue instanceof List) {
            ........
            List list = (List) propValue;
             ........
              if (index < list.size()) {
                  list.set(index, convertedValue);
              }
              else if (index >= list.size()) {
                  for (int i = list.size(); i < index; i++) {
                      try {
                          list.add(null);
                      }
                  }
                  list.add(convertedValue);
          .......
        } //这里处理对Map的注入
        else if (propValue instanceof Map) {
            ......
            Map map = (Map) propValue;
            ......
            map.put(convertedMapKey, convertedMapValue);
        }
      ........
    }
    //这里是通过对一般属性进行注入的地方
```

```
      else {
         PropertyDescriptor pd = getPropertyDescriptorInternal(propertyName);
         if (pd == null || pd.getWriteMethod() == null) {
            PropertyMatches matches = PropertyMatches.forProperty(propertyName,
getRootClass());
            throw new NotWritablePropertyException(
                  getRootClass(), this.nestedPath + propertyName,
                  matches.buildErrorMessage(), matches.getPossibleMatches());
         }
         //得到需要的set/get方法
         Method readMethod = pd.getReadMethod();
         Method writeMethod = pd.getWriteMethod();
         Object oldValue = null;
         ........

         try {
            Object convertedValue =
this.typeConverterDelegate.convertIfNecessary(oldValue, newValue, pd);

            //千辛万苦, 这里是通过set方法对bean对象的依赖属性进行注入
            if (!Modifier.isPublic(writeMethod.getDeclaringClass().getModifiers())) {
               writeMethod.setAccessible(true);
            }
            writeMethod.invoke(this.object, new Object[] {convertedValue});
        ...........
   }
```

这里比较重要的是propValue的取得, 我们看看getPropertyValue的实现:

```
   private Object getPropertyValue(PropertyTokenHolder tokens) throws BeansException
{
      ....
      //这里取得对property的读取方法, 然后取得在bean对象中的属性引用
      Method readMethod = pd.getReadMethod();
       try {
         if (!Modifier.isPublic(readMethod.getDeclaringClass().getModifiers())) {
            readMethod.setAccessible(true);
         }
         Object value = readMethod.invoke(this.object, (Object[]) null);
         if (tokens.keys != null) {
               ......
               }//这里处理Array属性
                else if (value.getClass().isArray()) {
                   value = Array.get(value, Integer.parseInt(key));
                }//这里处理List属性
                else if (value instanceof List) {
                   List list = (List) value;
                   value = list.get(Integer.parseInt(key));
                }//这里处理Set属性
                else if (value instanceof Set) {
                   // Apply index to Iterator in case of a Set.
                   Set set = (Set) value;
                   int index = Integer.parseInt(key);
                   }
                ......
                }//这里处理Map属性
                else if (value instanceof Map) {
                   Map map = (Map) value;
                   Class mapKeyType = null;
```

```
                    if (JdkVersion.isAtLeastJava15()) {
                        mapKeyType =
GenericCollectionTypeResolver.getMapKeyReturnType(
                                pd.getReadMethod(), tokens.keys.length);
                    }
                ....
                    Object convertedMapKey =
this.typeConverterDelegate.convertIfNecessary(
                            null, null, key, mapKeyType);
                    // Pass full property name and old value in here, since we want full
                    // conversion ability for map values.
                    value = map.get(convertedMapKey);
                }
            ........
            return value;
}
```

这就是整个依赖注入的处理过程, 在这个过程中起主要作用的是WrapperImp , 这个Wrapper不是一个简单的对bean对象的封装, 因为它需要处理在beanDefinition中的信息来迭代的处理依赖注入。从 上面可以看到两个明显的迭代过程, 一个是迭代的在上下文体系中查找需要的bean和创建没有被创建的bean - 根据依赖关系为线索, 另一个迭代实在依赖注入的时候, 如果依赖没有创建, 因为是一个向容器取得bean的过程 - 其中的IOC工厂的getbean方法被迭代的调用, 中间又迭代的对需要创建的bean进行了创建和依赖注入, 这样根据依赖关系, 一层一层的创建和注入直 至顶层被要求的bean创建和依赖注入完成 - 这样最后得到一个依赖创建和注入完成的最顶层bean被用来交给客户程序使用。这个就是整个依赖注入的实现过程, 在这里我们需要注意的是这个依赖注入的触发是客户第一次向容器请求Bean的时候触发的, 但是也有例外就是在我们通过设置Bean的lazy-init属性来控制预实例化的过程, 这个预实例化在初始化上下文的时候就完成了Bean的依赖注入:

我们回头看看在上下文初始化的调用, 也就是refresh中的代码实现:

```
            beanFactory.preInstantiateSingletons();
```

在DefaultListableBeanFactory中的实现是这样的:

```
    public void preInstantiateSingletons() throws BeansException {
        if (logger.isInfoEnabled()) {
            logger.info("Pre-instantiating singletons in factory [" + this + "]");
        }
        //这里迭代所有的bean定义, 看看是否有需要预实例化的
        for (Iterator it = this.beanDefinitionNames.iterator(); it.hasNext();) {
            String beanName = (String) it.next();
            if (!containsSingleton(beanName) && containsBeanDefinition(beanName)) {
                RootBeanDefinition bd = getMergedBeanDefinition(beanName, false);

                //预实例化只对singleton和lazy-init设为false的bean起作用
                //而实际的预实例化就是在容器启动的过程就把依赖注入, 而不是等到用户要求的时候
                //调用getBean,和用户第一次要求的时候处理是一样的
                if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
                    Class beanClass = resolveBeanClass(bd, beanName);
                    if (beanClass != null && FactoryBean.class.isAssignableFrom(beanClass)) {
                        getBean(FACTORY_BEAN_PREFIX + beanName);
                    }
                    else {
                        getBean(beanName);
                    }
                }
            }
        }
    }
```

这里实际上就在初始化的时候通过getBean调用完成了依赖注入，这个依赖注入的触发不是客户主动触发的，而是上下文自己触发的。这个lazy-init的属性的设置在BeanDefinitionParserDelegate中可以看到：

```
   public AbstractBeanDefinition parseBeanDefinitionElement(
         Element ele, String beanName, BeanDefinition containingBean) {
         ........
         //这里取得属性值，在bean定义文件中
         String lazyInit = ele.getAttribute(LAZY_INIT_ATTRIBUTE);
         //如果是默认的值并且这个bean是单例，设为false
         if (DEFAULT_VALUE.equals(lazyInit) && bd.isSingleton()) {
            // Just apply default to singletons, as lazy-init has no meaning for prototypes.
            lazyInit = getDefaultLazyInit();
         }
         //否则设为true
         bd.setLazyInit(TRUE_VALUE.equals(lazyInit));

         ........
   }
```

这个预实例化也算是对依赖注入处理的一个小小的控制吧。另外我们也可以看看我们常见的工厂Bean是怎样实现的，在getBean中，我们常常看到下面的调用：

```
            bean = getObjectForBeanInstance(sharedInstance, name,
mergedBeanDefinition);
```

这就是处理工厂Bean的地方：

```
   protected Object getObjectForBeanInstance(Object beanInstance, String name,
RootBeanDefinition mbd)
         throws BeansException {

      String beanName = transformedBeanName(name);
      //这里判断是不是工厂Bean,如果不是就抛出异常
      if (BeanFactoryUtils.isFactoryDereference(name) && !(beanInstance instanceof
FactoryBean)) {
         throw new BeanIsNotAFactoryException(beanName, beanInstance.getClass());
      }

      boolean shared = (mbd == null || mbd.isSingleton());
      Object object = beanInstance;
      //如果是一个工厂Bean，最后得到的是通过这个工厂bean的工厂方法生成的产品对象
      // 这里对beanInstance转型成工厂Bean
      if (beanInstance instanceof FactoryBean) {
         if (!BeanFactoryUtils.isFactoryDereference(name)) {
            // Return bean instance from factory.
            FactoryBean factory = (FactoryBean) beanInstance;
            if (logger.isDebugEnabled()) {
               logger.debug("Bean with name '" + beanName + "' is a factory bean");
            }
            // 如果是单件的话，从工厂的缓存中去取，如果缓存中还没有，那就让工厂生产一个并放到
缓存中去
            if (shared && factory.isSingleton()) {
               synchronized (this.factoryBeanObjectCache) {
                  object = this.factoryBeanObjectCache.get(beanName);
                  if (object == null) {
                     object = getObjectFromFactoryBean(factory, beanName, mbd);
                     this.factoryBeanObjectCache.put(beanName, object);
                  }
               }
            }
            else {
```

```
                //这是让工厂生产对象的地方，如果需要的时prototype类型的话，每次都从工厂生产
                object = getObjectFromFactoryBean(factory, beanName, mbd);
            }
        }
        else {
            // The user wants the factory itself.
            if (logger.isDebugEnabled()) {
                logger.debug("Calling code asked for FactoryBean instance for name '" +
beanName + "'");
            }
        }
    }

    return object;
  }
```

具体的工厂过程是这样的，因为工厂bean需要实现getObject方法，这就是工厂的生产方法：

```
  private Object getObjectFromFactoryBean(FactoryBean factory, String beanName,
RootBeanDefinition mbd)
        throws BeanCreationException {

    Object object;

    try {
        object = factory.getObject();
    }
    catch (FactoryBeanNotInitializedException ex) {
        throw new BeanCurrentlyInCreationException(beanName, ex.toString());
    }
    catch (Throwable ex) {
        throw new BeanCreationException(beanName, "FactoryBean threw exception on
object creation", ex);
    }

    // Do not accept a null value for a FactoryBean that's not fully
    // initialized yet: Many FactoryBeans just return null then.
    if (object == null && isSingletonCurrentlyInCreation(beanName)) {
        throw new BeanCurrentlyInCreationException(
                beanName, "FactoryBean which is currently in creation returned null from
getObject");
    }

    if (object != null && (mbd == null || !mbd.isSynthetic())) {
        object = postProcessObjectFromFactoryBean(object, beanName);
    }

    return object;
  }
```

这里返回的已经是工厂生产的产品，并不是工厂本身。通过这种Bean工厂的机制，可以为我们提供一个很好的封装机制，比如封装Proxy,RMI,JNDI等等。

总结

在这里，我们结合Spring的源代码对Spring的核心 - IOC容器的基本实现作了一个分析，其中包括IOC容器和上下文的基本工作原理，上下文的初始化过程，IOC容器对依赖注入的实现等等；总的来说，对上下文的基本工作原理我们可以大致的分为以下几个方面：

- Bean定义信息的定位:对IOC容器来说, 它为我们管理POJO之间的依赖关系提供了帮助, 但客户也需要依据Spring的Bean定义规则提供Bean定义信息, 我们可以使用各种形式的Bean定义信息, 当然最常用的是XML的。这里Spring为客户提供了了很大的灵活性, 同时为了初始化Spring IOC容器, 首先就需要定位到这些有效的Bean定义信息, 这里Spring使用Resource这个接口来统一这些Bean定义信息, 而这个定位由ResourceLoader来完成。如果我们使用上下文的话, 上下文本身就为客户提供了定位的功能 - 因为上下文本身在Spring里就是DefaultResourceLoader的子类。如果使用基本的Bean工厂作为IOC容器, 客户需要自己为它指定相应的Resource来完成Bean信息的定位。
- 上下文的初始化:我们在使用上下文的时候, 需要一个对它进行初始化的过程, 完成初始化以后, 这个IOC容器才是可用的。这个过程的入口是在上下文的refresh中实现的, 其中初始化过程中比较重要的部分是对Bean定义信息的载入和注册工作。相当于在IOC容器中需要建立一个Bean定义的映像, Spring为了达到载入的灵活性, 把载入的功能从IOC容器中分离出来, 有BeanDefinitionReader来完成Bean定义信息的读取, 解析和IOC容器内部数据结构的建立(BeanDefinition),然后向BeanFactory进行回调来在IOC容器中建立起Bean定义的映像, 在DefaultListableBeanFactory中, 这些BeanDefinition被维护在一个hashmap中, 以后的IOC容器对Bean的管理和操作就是通过这些建立起来的BeanDefinition来完成的。
- 在上下文初始化完成以后, IOC容器的使用就准备好了, 这个时候只是在IOC容器内部建立了BeanDefinition,具体的依赖关系还没有注入。在客户第一次向IOC容器请求Bean的时候, IOC容器对相关的Bean依赖关系进行注入 - 如果需要提前注入, 需要客户通过lazy-init属性来对预实例化进行控制, 这个预实例化是上下文初始化的一部分。
- 在依赖注入完成以后, IOC容器就会保持这些具备依赖关系的Bean让客户来使用。

以上这些过程结合在一起, 为Spring IOC容器的强大能力提供了基本的保证也也希望为您更好的使用和扩展Spring框架提供参考。