Spring的IOC容器实现原理

IOC的基础 下面我们从IOC/AOP开始，它们是Spring平台实现的核心部分；虽然，我们一开始大多只是在这个层面上，做一些配置和外部特性的使用工作，但对这两个核心模块工作原理和运作机制的理解，对深入理解Spring平台，却是至关重要的；因为，它们同时也是Spring其他模块实现的基础。从Spring要做到的目标，也就是从简化Java EE开发的出发点来看，简单的来说，它是通过对POJO开发的支持，来具体实现的；具体的说，Spring通过为应用开发提供基于POJO的开发模式，把应用开发和复杂的Java EE服务，实现解耦，并通过提高单元测试的覆盖率，从而有效的提高整个应用的开发质量。这样一来，实际上，就需要把为POJO提供支持的，各种Java EE服务支持抽象到应用平台中去，去封装起来；而这种封装功能的实现，在Spring中，就是由IOC容器以及AOP来具体提供的，这两个模块，在很大程度上，体现了Spring作为应用开发平台的核心价值。它们的实现，是Rod.Johnson在他的另一本著作《Expert One-on-One J2EE Development without EJB》 中，所提到Without EJB设计思想的体现；同时也深刻的体现了Spring背后的设计理念。

从更深一点的技术层面上来看，因为Spring是一个基于Java语言的应用平台，如果我们能够对Java计算模型，比如像JVM虚拟机实现技术的基本原理有一些了解，会让我们对Spring实现的理解，更加的深入，这些JVM虚拟机的特性使用，包括像反射机制，代理类，字节码技术等等。它们都是在Spring实现中，涉及到的一些Java计算环境的底层技术；尽管对应用开发人员来说，可能不会直接去涉及这些JVM虚拟机底层实现的工作，但是了解这些背景知识，或多或少，对我们了解整个Spring平台的应用背景有很大的帮助；打个比方来说，就像我们在大学中，学习的那些关于计算机组织和系统方面的基本知识，比如像数字电路，计算机组成原理，汇编语言，操作系统等等这些基本课程的学习。虽然，坦率的来说，对我们这些大多数课程的学习者，在以后的工作中，可能并没有太多的机会，直接从事这么如此底层的技术开发工作；但具备这些知识背景，为我们深入理解基于这些基础技术构架起来的应用系统，毫无疑问，是不可缺少的。随着JVM虚拟机技术的发展，可以设想到的是，更多虚拟机级别的基本特性，将会持续的被应用平台开发者所关注和采用，这也是我们在学习平台实现的过程中，非常值得注意的一点，因为这些底层技术实现，毫无疑问，会对Spring应用平台的开发路线，产品策略产生重大的影响。同时，在使用Spring作为应用平台的时候，如果需要更深层次的开发和性能调优，这些底层的知识，也是我们知识库中不可缺少的部分。有了这些底层知识，理解整个系统，想来就应该障碍不大了。

IOC的一点认识 对Spring IOC的理解离不开对依赖反转模式的理解，我们知道，关于如何反转对依赖的控制，把控制权从具体业务对象手中转交到平台或者框架中，是解决面向对象系统设计复杂性和提高面向对象系统可测试性的一个有效的解决方案。这个问题触发了IoC设计模式的发展，是IoC容器要解决的核心问题。同时，也是产品化的IoC容器出现的推动力。而我觉得Spring的IoC容器，就是一个开源的实现依赖反转模式的产品。

那具体什么是IoC容器呢？它在Spring框架中到底长什么样？说了这么多，其实对IoC容器的使用者来说，我们常常接触到的BeanFactory和ApplicationContext都可以看成是容器的具体表现形式。这些就是IoC容器，或者说在Spring中提IoC容器，从实现来说，指的是一个容器系列。这也就是说，我们通常所说的IoC容器，如果深入到Spring的实现去看，会发现IoC容器实际上代表着一系列功能各异的容器产品。只是容器的功能有大有小，有各自的特点。打个比方来说，就像是百货商店里出售的商品，我们举水桶为例子，在商店中出售的水桶有大有小；制作材料也各不相同，有金属的，有塑料的等等，总之是各式各样，但只要能装水，具备水桶的基本特性，那就可以作为水桶来出售来让用户使用。这在Spring中也是一样，它有各式各样的IoC容器的实现供用户选择和使用；使用什么样的容器完全取决于用户的

需要，但在使用之前如果能够了解容器的基本情况，那会对容器的使用是非常有帮助的；就像我们在购买商品时进行的对商品的考察和挑选那样。

我们从最基本的XmlBeanFactory看起，它是容器系列的最底层实现，这个容器的实现与我们在Spring应用中用到的那些上下文相比，有一个非常明显的特点，它只提供了最基本的IoC容器的功能。从它的名字中可以看出，这个IoC容器可以读取以XML形式定义的BeanDefinition。理解这一点有助于我们理解ApplicationContext与基本的BeanFactory之间的区别和联系。我们可以认为直接的BeanFactory实现是IoC容器的基本形式，而各种ApplicationContext的实现是IoC容器的高级表现形式。

仔细阅读XmlBeanFactory的源码，在一开始的注释里面已经对 XmlBeanFactory的功能做了简要的说明，从代码的注释还可以看到，这是Rod Johnson在2001年就写下的代码，可见这个类应该是Spring的元老类了。它是继承DefaultListableBeanFactory这个类的，这个DefaultListableBeanFactory就是一个很值得注意的容器！

```
public class XmlBeanFactory extends DefaultListableBeanFactory {
    private final XmlBeanDefinitionReader reader = new
XmlBeanDefinitionReader(this);
    public XmlBeanFactory(Resource resource) throws BeansException {
        this(resource, null);
    }
    public XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory)
throws BeansException {
        super(parentBeanFactory);
        this.reader.loadBeanDefinitions(resource);
    }
}
```

BeanFactory测试：

```
public class BeanFactoryTest {

    public static void main(String[] args) {
        ResourcePatternResolver resolver = new
PathMatchingResourcePatternResolver();
        Resource res = resolver.getResource("classpath:spring-test.xml");
        BeanFactory bf = new XmlBeanFactory(res);
        System.out.println("init BeanFactory");

        Mars mars = bf.getBean("mars", Mars.class);
        System.out.println(mars.getCnName() + ":" + mars.getAge());
    }
}
```

XmlBeanFactory的功能是建立在DefaultListableBeanFactory这个基本容器的基础上的，在这个基本容器的基础上实现了其他诸如XML读取的附加功能。对于这些功能的实现原理，看一看XmlBeanFactory的代码实现就能很容易地理解。在如下的代码中可以看到，在XmlBeanFactory构造方法中需要得到Resource对象。对XmlBeanDefinitionReader对象的初始化，以及使用这个这个对象来完成loadBeanDefinitions的调用，就是这个调用启动了从Resource中载入BeanDefinitions的过程，这个loadBeanDefinitions同时也是IoC容器初始化的重要组成部分。

简单来说，IoC容器的初始化包括BeanDefinition的Resouce定位、载入和注册这三个基本的过程。我觉得重点是在载入和对BeanDefinition做解析的这个过程。可以从DefaultListableBeanFactory来入手看看IoC容器是怎样完成BeanDefinition载入的。在refresh调用完成以后，可以看到loadDefinition的调用：

```
public abstract class AbstractXmlApplicationContext extends
AbstractRefreshableConfigApplicationContext {
    public AbstractXmlApplicationContext() {
    }
    public AbstractXmlApplicationContext(ApplicationContext parent) {
        super(parent);
    }
    //这里是实现loadBeanDefinitions的地方
    protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory)
throws IOException {
        // Create a new XmlBeanDefinitionReader for the given BeanFactory.
        // 创建 XmlBeanDefinitionReader，并通过回调设置到 BeanFactory中去，创建
BeanFactory的使用的也是 DefaultListableBeanFactory。
        XmlBeanDefinitionReader beanDefinitionReader = new
XmlBeanDefinitionReader(beanFactory);

        // Configure the bean definition reader with this context's
        // resource loading environment.
        // 这里设置 XmlBeanDefinitionReader， 为XmlBeanDefinitionReader 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));

        // Allow a subclass to provide custom initialization of the reader,
        // then proceed with actually loading the bean definitions.
    // 这是启动Bean定义信息载入的过程
        initBeanDefinitionReader(beanDefinitionReader);
        loadBeanDefinitions(beanDefinitionReader);
    }

    protected void initBeanDefinitionReader(XmlBeanDefinitionReader
beanDefinitionReader) {
    }
```

这里使用 XmlBeanDefinitionReader来载入BeanDefinition到容器中，如以下代码清单所示：

```
    //这里是调用的入口。
    public int loadBeanDefinitions(Resource resource) throws
BeanDefinitionStoreException {
        return loadBeanDefinitions(new EncodedResource(resource));
    }
    //这里是载入XML形式的BeanDefinition的地方。
    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());
        }

        Set<EncodedResource> currentResources =
this.resourcesCurrentlyBeingLoaded.get();
        if (currentResources == null) {
            currentResources = new HashSet<EncodedResource>(4);
            this.resourcesCurrentlyBeingLoaded.set(currentResources);
        }
        if (!currentResources.add(encodedResource)) {
            throw new BeanDefinitionStoreException(
                    "Detected recursive loading of " + encodedResource + " -
check your import definitions!");
        }
        //这里得到XML文件，并得到IO的InputSource准备进行读取。
        try {
            InputStream inputStream =
encodedResource.getResource().getInputStream();
            try {
                InputSource inputSource = new InputSource(inputStream);
                if (encodedResource.getEncoding() != null) {
                    inputSource.setEncoding(encodedResource.getEncoding());
                }
                return doLoadBeanDefinitions(inputSource,
encodedResource.getResource());
            }
            finally {
                inputStream.close();
            }
        }
        catch (IOException ex) {
            throw new BeanDefinitionStoreException(
                    "IOException parsing XML document from " +
encodedResource.getResource(), ex);
        }
        finally {
```

```
            currentResources.remove(encodedResource);
            if (currentResources.isEmpty()) {
                this.resourcesCurrentlyBeingLoaded.set(null);
            }
        }
    }
//具体的读取过程可以在doLoadBeanDefinitions方法中找到：
    //这是从特定的XML文件中实际载入BeanDefinition的地方
    protected int doLoadBeanDefinitions(InputSource inputSource, Resource
resource)
            throws BeanDefinitionStoreException {
        try {
            int validationMode = getValidationModeForResource(resource);
            //这里取得XML文件的Document对象，这个解析过程是由 documentLoader完成的，这
个documentLoader是DefaultDocumentLoader,在定义documentLoader的地方创建
            Document doc = this.documentLoader.loadDocument(
                    inputSource, getEntityResolver(), this.errorHandler,
validationMode, isNamespaceAware());
            //这里启动的是对BeanDefinition解析的详细过程，这个解析会使用到Spring的Bean
配置规则，是我们下面需要详细关注的地方。
            return registerBeanDefinitions(doc, resource);
        }
        catch (BeanDefinitionStoreException ex) {
            throw ex;
        }
        catch (SAXParseException ex) {
            throw new
XmlBeanDefinitionStoreException(resource.getDescription(),
                    "Line " + ex.getLineNumber() + " in XML document from " +
resource + " is invalid", ex);
        }
        catch (SAXException ex) {
            throw new
XmlBeanDefinitionStoreException(resource.getDescription(),
                    "XML document from " + resource + " is invalid", ex);
        }
        catch (ParserConfigurationException ex) {
            throw new BeanDefinitionStoreException(resource.getDescription(),

                    "Parser configuration exception parsing XML from " +
resource, ex);
        }
        catch (IOException ex) {
            throw new BeanDefinitionStoreException(resource.getDescription(),

                    "IOException parsing XML document from " + resource, ex);

        }
        catch (Throwable ex) {
```

```
            throw new BeanDefinitionStoreException(resource.getDescription(),

                    "Unexpected exception parsing XML document from " +
resource, ex);
        }
    }
```

```
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;
    }
```

```
public DelegatingEntityResolver(ClassLoader classLoader) {
        this.dtdResolver = new BeansDtdResolver();
        this.schemaResolver = new PluggableSchemaResolver(classLoader);
    }
```

```
public PluggableSchemaResolver(ClassLoader classLoader) {
        this.classLoader = classLoader;
        this.schemaMappingsLocation = DEFAULT_SCHEMA_MAPPINGS_LOCATION; //
"META-INF/spring.schemas"
    }
```

关于具体的Spring BeanDefinition的解析，是在BeanDefinitionParserDelegate中完成的。这个类里包含了各种Spring Bean定义规则的处理，感兴趣的同学可以仔细研究。我们举一个例子来分析这个处理过程，比如我们最熟悉的对Bean元素的处理是怎样完成的，也就是我们在XML定义文件中出现的这个最常见的元素信息是怎样被处理的。在这里，我们会看到那些熟悉的BeanDefinition定义的处理，比如id、name、aliase等属性元素。把这些元素的值从XML文件相应的元素的属性中读取出来以后，会被设置到生成的BeanDefinitionHolder中去。这些属性的解析还是比较简单的。对于其他元素配置的解析，

比如各种Bean的属性配置，通过一个较为复杂的解析过程，这个过程是由parseBeanDefinitionElement来完成的。解析完成以后，会把解析结果放到BeanDefinition对象中并设置到BeanDefinitionHolder中去，如以下清单所示： #DefaultBeanDefinitionDocumentReader

```
/**
     * Process the given bean element, parsing the bean definition
     * and registering it with the registry.
     */
    protected void processBeanDefinition(Element ele,
BeanDefinitionParserDelegate delegate) {
        BeanDefinitionHolder bdHolder =
delegate.parseBeanDefinitionElement(ele);
        if (bdHolder != null) {
            bdHolder = delegate.decorateBeanDefinitionIfRequired(ele,
bdHolder);
            try {
                // Register the final decorated instance.
                BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
getReaderContext().getRegistry());
            }
            catch (BeanDefinitionStoreException ex) {
                getReaderContext().error("Failed to register bean definition
with name '" +
                        bdHolder.getBeanName() + "'", ele, ex);
            }
            // Send registration event.
            getReaderContext().fireComponentRegistered(new
BeanComponentDefinition(bdHolder));
        }
    }
```

```
public static void registerBeanDefinition(
            BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry
registry)
            throws BeanDefinitionStoreException {

        // Register bean definition under primary name.
        String beanName = definitionHolder.getBeanName();
        registry.registerBeanDefinition(beanName,
definitionHolder.getBeanDefinition());

        // Register aliases for bean name, if any.
        String[] aliases = definitionHolder.getAliases();
        if (aliases != null) {
            for (String aliase : aliases) {
                registry.registerAlias(beanName, aliase);
            }
        }
```

```
    }
```

#BeanDefinitionReaderUtils

```
public static void registerBeanDefinition(
            BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry
registry)
            throws BeanDefinitionStoreException {

        // Register bean definition under primary name.
        String beanName = definitionHolder.getBeanName();
        registry.registerBeanDefinition(beanName,
definitionHolder.getBeanDefinition());

        // Register aliases for bean name, if any.
        String[] aliases = definitionHolder.getAliases();
        if (aliases != null) {
            for (String aliase : aliases) {
                registry.registerAlias(beanName, aliase);
            }
        }
    }
```

上面的registry正是DefaultListableBeanFactory!

#BeanDefinitionParserDelegate

```
public BeanDefinitionHolder parseBeanDefinitionElement(Element ele,
BeanDefinition containingBean) {
        //这里取得在<bean>元素中定义的id、name和aliase属性的值
        String id = ele.getAttribute(ID_ATTRIBUTE);
        String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);

        List<String> aliases = new ArrayList<String>();
        if (StringUtils.hasLength(nameAttr)) {
            String[] nameArr = StringUtils.tokenizeToStringArray(nameAttr,
BEAN_NAME_DELIMITERS);
            aliases.addAll(Arrays.asList(nameArr));
        }

        String beanName = id;
        if (!StringUtils.hasText(beanName) && !aliases.isEmpty()) {
            beanName = 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);
        }

        //这个方法会引发对bean元素的详细解析
AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele,
beanName, containingBean);
        if (beanDefinition != null) {
            if (!StringUtils.hasText(beanName)) {
                try {
                    if (containingBean != null) {
                        beanName = BeanDefinitionReaderUtils.generateBeanName(

                                beanDefinition,
this.readerContext.getRegistry(), true);
                    }
                    else {
                        beanName =
this.readerContext.generateBeanName(beanDefinition);
                        // Register an alias for the plain bean class name, if
still possible,
                        // if the generator returned the class name plus a
suffix.
                        // This is expected for Spring 1.2/2.0 backwards
compatibility.
                        String beanClassName =
beanDefinition.getBeanClassName();
                        if (beanClassName != null &&
                                beanName.startsWith(beanClassName) &&
beanName.length() > beanClassName.length() &&

 !this.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {
                            aliases.add(beanClassName);
                        }
                    }
                    if (logger.isDebugEnabled()) {
                        logger.debug("Neither XML 'id' nor 'name' specified -
" +
                                "using generated bean name [" + beanName +
"]");
                    }
                }
                catch (Exception ex) {
                    error(ex.getMessage(), ele);
                    return null;
                }
            }
```

```
            String[] aliasesArray = StringUtils.toStringArray(aliases);
            return new BeanDefinitionHolder(beanDefinition, beanName,
aliasesArray);
        }

        return null;
    }
```

在具体生成BeanDefinition以后。我们举一个对property进行解析的例子来完成对整个BeanDefinition载入过程的分析，还是在类BeanDefinitionParserDelegate的代码中，它对BeanDefinition中的定义一层一层地进行解析，比如从属性元素集合到具体的每一个属性元素，然后才是对具体的属性值的处理。根据解析结果，对这些属性值的处理会封装成PropertyValue对象并设置到BeanDefinition对象中去，如以下代码清单所示。

```
/**
 * 这里对指定bean元素的property子元素集合进行解析。
 */
public void parsePropertyElements(Element beanEle, BeanDefinition bd) {
    //遍历所有bean元素下定义的property元素
    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)) {
            //在判断是property元素后对该property元素进行解析的过程
            parsePropertyElement((Element) node, bd);
        }
    }
}
public void parsePropertyElement(Element ele, BeanDefinition bd) {
    //这里取得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 {
        //如果同一个bean中已经有同名的存在，则不进行解析，直接返回。也就是说，如果在同一个
bean中有同名的property设置，那么起作用的只是第一个。
        if (bd.getPropertyValues().contains(propertyName)) {
            error("Multiple 'property' definitions for property '" +
propertyName + "'", ele);
            return;
        }
        //这里是解析property值的地方，返回的对象对应对Bean定义的property属性设置的解析结
果，这个解析结果会封装到PropertyValue对象中，然后设置到BeanDefinitionHolder中去。
```

```
        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();
    }
}
/**
 * 这里取得property元素的值，也许是一个list或其他。
 */
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++) {
        Node node = nl.item(i);
        if (node instanceof Element && !DomUtils.nodeNameEquals(node,
DESCRIPTION_ELEMENT) &&
                !DomUtils.nodeNameEquals(node, META_ELEMENT)) {
            // Child element is what we're looking for.
            if (subElement != null) {
                error(elementName + " must not contain more than one sub-
element", ele);
            }
            else {
                subElement = (Element) node;
            }
        }
    }
    //这里判断property的属性，是ref还是value,不允许同时是ref和value。
    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);
    }
    //如果是ref，创建一个ref的数据对象RuntimeBeanReference,这个对象封装了ref的信息。
```

```
    if (hasRefAttribute) {
        String refName = ele.getAttribute(REF_ATTRIBUTE);
        if (!StringUtils.hasText(refName)) {
            error(elementName + " contains empty 'ref' attribute", ele);
        }
        RuntimeBeanReference ref = new RuntimeBeanReference(refName);
        ref.setSource(extractSource(ele));
        return ref;
    } //如果是value，创建一个value的数据对象TypedStringValue ,这个对象封装了value的信息。
    else if (hasValueAttribute) {
        TypedStringValue valueHolder = new TypedStringValue(ele.getAttribute(VALUE_ATTRIBUTE));
        valueHolder.setSource(extractSource(ele));
        return valueHolder;
    } //如果还有子元素，触发对子元素的解析
    else if (subElement != null) {
        return parsePropertySubElement(subElement, bd);
    }
    else {
        // Neither child element nor "ref" or "value" attribute found.
        error(elementName + " must specify a ref or value", ele);
        return null;
    }
}
```

比如，再往下看，我们看到像List这样的属性配置是怎样被解析的，依然在BeanDefinitionParserDelegate中：返回的是一个List对象，这个List是Spring定义的ManagedList，作为封装List这类配置定义的数据封装，如以下代码清单所示。

```
public List parseListElement(Element collectionEle, BeanDefinition bd) {
    String defaultElementType = collectionEle.getAttribute(VALUE_TYPE_ATTRIBUTE);
    NodeList nl = collectionEle.getChildNodes();
    ManagedList<Object> target = new ManagedList<Object>(nl.getLength());
    target.setSource(extractSource(collectionEle));
    target.setElementTypeName(defaultElementType);
    target.setMergeEnabled(parseMergeAttribute(collectionEle));
    //具体的List元素的解析过程。
    parseCollectionElements(nl, target, bd, defaultElementType);
    return target;
}
protected void parseCollectionElements(
        NodeList elementNodes, Collection<Object> target, BeanDefinition bd, String defaultElementType) {
    //遍历所有的元素节点，并判断其类型是否为Element。
    for (int i = 0; i < elementNodes.getLength(); i++) {
```

```
        Node node = elementNodes.item(i);
        if (node instanceof Element && !DomUtils.nodeNameEquals(node,
DESCRIPTION_ELEMENT)) {
    //加入到target中去，target是一个ManagedList，同时触发对下一层子元素的解析过程，这是
一个递归的调用。
            target.add(parsePropertySubElement((Element) node, bd,
defaultElementType));
        }
    }
```

经过这样一层一层的解析，我们在XML文件中定义的BeanDefinition就被整个给载入到了IoC容器中，并在容器中建立了数据映射。在IoC容器中建立了对应的数据结构，或者说可以看成是POJO对象在IoC容器中的映像，这些数据结构可以以AbstractBeanDefinition为入口，让IoC容器执行索引、查询和操作。

在我的感觉中，对核心数据结构的定义和处理应该可以看成是一个软件的核心部分了。所以，这里的BeanDefinition的载入可以说是IoC容器的核心，如果说IoC容器是Spring的核心，那么这些BeanDefinition就是Spring的核心的核心了！

呵呵，这部分代码数量不小，但如果掌握这条主线，其他都可以举一反三吧，就像我们掌握了操作系统启动的过程，以及在操作系统设计中的核心数据结构像进程数据结构，文件系统数据结构，网络协议数据结构的设计和处理一样，对整个系统的设计原理，包括移植，驱动开发和应用开发，是非常有帮助的！

让我们回到牛逼的delegate:

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

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

```
public class DefaultNamespaceHandlerResolver implements
NamespaceHandlerResolver {

    /**
     * The location to look for the mapping files. Can be present in multiple
JAR files.
     */
    public static final String DEFAULT_HANDLER_MAPPINGS_LOCATION = "META-
INF/spring.handlers";

        public DefaultNamespaceHandlerResolver() {
        this(null, DEFAULT_HANDLER_MAPPINGS_LOCATION);
    }

/**
     * Locate the {@link NamespaceHandler} for the supplied namespace URI
     * from the configured mappings.
     * @param namespaceUri the relevant namespace URI
     * @return the located {@link NamespaceHandler}, or {@code null} if none
found
     */
    public NamespaceHandler resolve(String namespaceUri) {
        Map<String, Object> handlerMappings = getHandlerMappings();
        Object handlerOrClassName = handlerMappings.get(namespaceUri);
        if (handlerOrClassName == null) {
            return null;
        }
        else if (handlerOrClassName instanceof NamespaceHandler) {
            return (NamespaceHandler) handlerOrClassName;
        }
        else {
            String className = (String) handlerOrClassName;
            try {
                Class<?> handlerClass = ClassUtils.forName(className,
this.classLoader);
                if (!NamespaceHandler.class.isAssignableFrom(handlerClass)) {

                    throw new FatalBeanException("Class [" + className + "]
for namespace [" + namespaceUri +
                            "] does not implement the [" +
NamespaceHandler.class.getName() + "] interface");
                }
                NamespaceHandler namespaceHandler = (NamespaceHandler)
BeanUtils.instantiateClass(handlerClass);
                namespaceHandler.init();
                handlerMappings.put(namespaceUri, namespaceHandler);
                return namespaceHandler;
            }
            catch (ClassNotFoundException ex) {
```

```
                throw new FatalBeanException("NamespaceHandler class [" +
classNameName + "] for namespace [" +
                        namespaceUri + "] not found", ex);
            }
            catch (LinkageError err) {
                throw new FatalBeanException("Invalid NamespaceHandler class
[" + classNameName + "] for namespace [" +
                        namespaceUri + "]: problem with handler class file or
dependent class", err);
            }
        }
    }
}
```

我们自定义标签时，需要做的是在META-INF下建两个文件spring.handlers, spring.schemas，然后实现我们自己的NamespaceHandler和BeanDefinitionParser.

```
public class AopNamespaceHandler extends NamespaceHandlerSupport {

    /**
     * Register the {@link BeanDefinitionParser BeanDefinitionParsers} for the
     * '{@code config}', '{@code spring-configured}', '{@code aspectj-
autoproxy}'
     * and '{@code scoped-proxy}' tags.
     */
    public void init() {
        // In 2.0 XSD as well as in 2.1 XSD.
        registerBeanDefinitionParser("config", new
ConfigBeanDefinitionParser());
        registerBeanDefinitionParser("aspectj-autoproxy", new
AspectJAutoProxyBeanDefinitionParser());
        registerBeanDefinitionDecorator("scoped-proxy", new
ScopedProxyBeanDefinitionDecorator());

        // Only in 2.0 XSD: moved to context namespace as of 2.1
        registerBeanDefinitionParser("spring-configured", new
SpringConfiguredBeanDefinitionParser());
    }

}
```

#AspectJAutoProxyBeanDefinitionParser

```
public BeanDefinition parse(Element element, ParserContext parserContext) {

 AopNamespaceUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(parser
Context, element);
        extendBeanDefinition(element, parserContext);
        return null;
    }
```

AopNamespaceUtils

```
public static void registerAspectJAnnotationAutoProxyCreatorIfNecessary(
            ParserContext parserContext, Element sourceElement) {

        BeanDefinition beanDefinition =
AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(
                parserContext.getRegistry(),
parserContext.extractSource(sourceElement));
        useClassProxyingIfNecessary(parserContext.getRegistry(),
sourceElement);
        registerComponentIfNecessary(beanDefinition, parserContext);
    }
```

#AopConfigUtils

```
public static BeanDefinition
registerAspectJAnnotationAutoProxyCreatorIfNecessary(BeanDefinitionRegistry
registry, Object source) {
        return
registerOrEscalateApcAsRequired(AnnotationAwareAspectJAutoProxyCreator.class,
registry, source);
    }

private static BeanDefinition registerOrEscalateApcAsRequired(Class cls,
BeanDefinitionRegistry registry, Object source) {
        Assert.notNull(registry, "BeanDefinitionRegistry must not be null");
        if (registry.containsBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME)) {
            BeanDefinition apcDefinition =
registry.getBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME);
            if (!cls.getName().equals(apcDefinition.getBeanClassName())) {
                int currentPriority =
findPriorityForClass(apcDefinition.getBeanClassName());
                int requiredPriority = findPriorityForClass(cls);
                if (currentPriority < requiredPriority) {
                    apcDefinition.setBeanClassName(cls.getName());
                }
```

```
            }
            return null;
        }
        RootBeanDefinition beanDefinition = new RootBeanDefinition(cls);
        beanDefinition.setSource(source);
        beanDefinition.getPropertyValues().add("order",
Ordered.HIGHEST_PRECEDENCE);
        beanDefinition.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
                // 看这里!
        registry.registerBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME,
beanDefinition);
        return beanDefinition;
    }
```

另外，基于注解的配置，

1. <context:component-scan base-**package**="com.itlong.whatsmars.spring"/>

处理过程参见ComponentScanBeanDefinitionParser，

```
public BeanDefinition parse(Element element, ParserContext parserContext) {
        String[] basePackages =
StringUtils.tokenizeToStringArray(element.getAttribute(BASE_PACKAGE_ATTRIBUTE)
,
                ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS);

        // Actually scan for bean definitions and register them.
        ClassPathBeanDefinitionScanner scanner =
configureScanner(parserContext, element);
        Set<BeanDefinitionHolder> beanDefinitions =
scanner.doScan(basePackages);
        registerComponents(parserContext.getReaderContext(), beanDefinitions,
element);

        return null;
    }
```