Spring源代码解析(一）：IOC容器

原创作者: jiwenke 阅读:9369次 评论:4条 更新时间:2007-06-03

在认真学习Rod.Johnson的三部曲之一：<<Professional Java Development with the spring framework>>,顺便也看了看源代码想知道个究竟，抛砖引玉，有兴趣的同志一起讨论研究吧！
以下内容引自博客：http://jiwenke-spring.blogspot.com/,欢迎指导：）
在Spring中，IOC容器的重要地位我们就不多说了，对于Spring的使用者而言，IOC容器实际上是什么呢？我们可以说BeanFactory就是我们看到的IoC容器，当然了Spring为我们准备了许多种IoC容器来使用，这样可以方便我们从不同的层面，不同的资源位置，不同的形式的定义信息来建立我们需要的IoC容器。
在Spring中，最基本的IOC容器接口是BeanFactory - 这个接口为具体的IOC容器的实现作了最基本的功能规定 - 不管怎么着，作为IOC容器，这些接口你必须要满足应用程序的最基本要求：

Java代码

```
1.  public interface BeanFactory {
2.
3.      //这里是对FactoryBean的转义定义，因为如果使用bean的名字检索FactoryBean得到的对象是工厂生成的对象
    ，
4.      //如果需要得到工厂本身，需要转义
5.      String FACTORY_BEAN_PREFIX = "&";
6.
7.
8.      //这里根据bean的名字，在IOC容器中得到bean实例，这个IOC容器就是一个大的抽象工厂。
9.      Object getBean(String name) throws BeansException;
10.
11.     //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。
12.     Object getBean(String name, Class requiredType) throws BeansException;
13.
```

搜索本专栏

文章信息

专栏: Spring源码分析

- 由jiwenke在2007-06-03创建
- 由jiwenke在2007-06-03更新

相关新闻

- Java EE 6最终草案暗示了平台的未来发展方向
- Java EE 6依赖注入提供了统一的EJB与JSF编程模型
- Java EE 6：EJB 3.1的变化引人注目

相关讨论

- Spring源代码解析(一）：IOC容器
- Spring技术内幕——深入解析Spring架构与设计原理（一）引子
- Spring技术内幕——深入解析Spring架构与设计原理（一）IOC实现原理
- 从源代码解读spring IOC容器
- Spring技术内幕——深入解

```
    //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
    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);

}
```

在BeanFactory里只对IOC容器的基本行为作了定义，根本不关心你的bean是怎样定义怎样加载的 - 就像我们只关心从这个工厂里我们得到到什么产品对象，至于工厂是怎么生产这些对象的，这个基本的接口不关心这些。如果要关心工厂是怎样产生对象的，应用程序需要使用具体的IOC容器实现- 当然你可以自己根据这个BeanFactory来实现自己的IOC容器，但这个没有必要，因为Spring已经为我们准备好了一系列工厂来让我们使用。比如XmlBeanFactory就是针对最基础的BeanFactory的IOC容器的实现 - 这个实现使用xml来定义IOC容器中的bean。
Spring提供了一个BeanFactory的基本实现，XmlBeanFactory同样的通过使用模板模式来得到对IOC容器的抽象-AbstractBeanFactory,DefaultListableBeanFactory这些抽象类为其提供模板服务。其中通过resource 接口来抽象bean定义数据，对Xml定义文件的解析通过委托给XmlBeanDefinitionReader来完成。下面我们根据书上的例子，简单的演示IOC容器的创建过程：

Java代码

```
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
3. 把读取配置信息的BeanDefinitionReader,这里是XmlBeanDefinitionReader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bean定义的过程。

我们的IoC容器就建立起来了。在BeanFactory的源代码中我们可以看到：

Java代码

```
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
    private final XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(this);
    public XmlBeanFactory(Resource resource) throws BeansException {

```


析Spring架构与设计原理
（四）Web MVC的实现

相关博客

- Spring源代码解析(一) ： IOC容器
- Spring源代码解析(一) ： IOC容器
- Spring源代码解析(一) ： IOC容器
- Spring源代码解析(一) ： IOC容器
- Spring源代码解析(一) ： IOC容器

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

我们在后面会看到读取器读取资源和注册bean定义信息的整个过程，基本上是和上下文的处理是一样的，从这里我们可以看到上下文和 XmlBeanFactory这两种IOC容器的区别，BeanFactory往往不具备对资源定义的能力，而上下文可以自己完成资源定义，从这个角度上看上下文更好用一些。
仔细分析Spring BeanFactory的结构，我们来看看在BeanFactory基础上扩展出的ApplicationContext - 我们最常使用的上下文。除了具备BeanFactory的全部能力，上下文为应用程序又增添了许多便利：

* 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
* 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是没有的。

ApplicationContext允许上下文嵌套 - 通过保持父上下文可以维持一个上下文体系 - 这个体系我们在以后对Web容器中的上下文环境的分析中可以清楚地看到。对于bean的查找可以在这个上下文体系中发生，首先检查当前上下文，其次是父上下文，逐级向上，这样为不同的Spring应用提供了一个共享的bean定义环境。这个我们在分析Web容器中的上下文环境时也能看到。
ApplicationContext提供IoC容器的主要接口，在其体系中有许多抽象子类比如AbstractApplicationContext为具体的BeanFactory的实现，比如FileSystemXmlApplicationContext和 ClassPathXmlApplicationContext提供上下文的模板，使得他们只需要关心具体的资源定位问题。当应用程序代码实例化 FileSystemXmlApplicationContext的时候，得到IoC容器的一种具体表现 - ApplicationContext，从而应用程序通过ApplicationContext来管理对bean的操作。
BeanFactory 是一个接口，在实际应用中我们一般使用ApplicationContext来使用IOC容器，它们也是IOC容器展现给应用开发者的使用接口。对应用程序开发者来说，可以认为BeanFactory和ApplicationFactory在不同的使用层面上代表了SPRING提供的IOC容器服务。
下面我们具体看看通过FileSystemXmlApplicationContext是怎样建立起IOC容器的，显而易见我们可以通过new来得到IoC容器：

Java代码

```
ApplicationContext = new FileSystemXmlApplicationContext(xmlPath);
```

调用的是它初始化代码：

Java代码

```
public FileSystemXmlApplicationContext(String[] configLocations, boolean refresh, Application
Context parent)
        throws BeansException {
    super(parent);
    this.configLocations = configLocations;
```

```
    if (refresh) {
       //这里是IoC容器的初始化过程，其初始化过程的大致步骤由AbstractApplicationContext来定义
        refresh();
    }
}
```

refresh的模板在AbstractApplicationContext:

Java代码

```
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        synchronized (this.activeMonitor) {
            this.active = true;
        }

        // 这里需要子类来协助完成资源位置定义,bean载入和向IOC容器注册的过程
        refreshBeanFactory();
        ............
 }
```

这个方法包含了整个BeanFactory初始化的过程，对于特定的FileSystemXmlBeanFactory,我们看到定位资源位置由refreshBeanFactory()来实现：
在AbstractXmlApplicationContext中定义了对资源的读取过程，默认由XmlBeanDefinitionReader来读取：

Java代码

```
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws IOException {
    // 这里使用XMLBeanDefinitionReader来载入bean定义信息的XML文件
    XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);


    //这里配置reader的环境，其中ResourceLoader是我们用来定位bean定义信息资源位置的
    ///因为上下文本身实现了ResourceLoader接口，所以可以直接把上下文作为ResourceLoader传递给XmlBeanDefinitionReader
    beanDefinitionReader.setResourceLoader(this);
    beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));

    initBeanDefinitionReader(beanDefinitionReader);
    //这里转到定义好的XmlBeanDefinitionReader中对载入bean信息进行处理
    loadBeanDefinitions(beanDefinitionReader);
}
```

转到beanDefinitionReader中进行处理：

Java代码

```java
protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException, IOException {
    Resource[] configResources = getConfigResources();
    if (configResources != null) {
        //调用XmlBeanDefinitionReader来载入bean定义信息。
        reader.loadBeanDefinitions(configResources);
    }
    String[] configLocations = getConfigLocations();
    if (configLocations != null) {
        reader.loadBeanDefinitions(configLocations);
    }
}
```

而在作为其抽象父类的AbstractBeanDefinitionReader中来定义载入过程：

Java代码

```java
public int loadBeanDefinitions(String location) throws BeanDefinitionStoreException {
 //这里得到当前定义的ResourceLoader,默认的我们使用DefaultResourceLoader
 ResourceLoader resourceLoader = getResourceLoader();
 .........//如果没有找到我们需要的ResourceLoader，直接抛出异常
    if (resourceLoader instanceof ResourcePatternResolver) {
        // 这里处理我们在定义位置时使用的各种pattern,需要ResourcePatternResolver来完成
        try {
            Resource[] resources = ((ResourcePatternResolver) resourceLoader).getResources(location);
            int loadCount = loadBeanDefinitions(resources);
            return loadCount;
        }
      ........
    }
    else {
        // 这里通过ResourceLoader来完成位置定位
        Resource resource = resourceLoader.getResource(location);
        // 这里已经把一个位置定义转化为Resource接口，可以供XmlBeanDefinitionReader来使用了
        int loadCount = loadBeanDefinitions(resource);
        return loadCount;
    }
}
```

当我们通过ResourceLoader来载入资源，别忘了了我们的GenericApplicationContext也实现了ResourceLoader接口：

Java代码

```java
public class GenericApplicationContext extends AbstractApplicationContext implements BeanDefinitionRegistry {
    public Resource getResource(String location) {
        //这里调用当前的loader也就是DefaultResourceLoader来完成载入
        if (this.resourceLoader != null) {
            return this.resourceLoader.getResource(location);
        }
        return super.getResource(location);
    }
.......
}
```

而我们的FileSystemXmlApplicationContext就是一个DefaultResourceLoader - GenericApplicationContext()通过DefaultResourceLoader:

Java代码

```java
public Resource getResource(String location) {
    //如果是类路径的方式，那需要使用ClassPathResource来得到bean文件的资源对象
    if (location.startsWith(CLASSPATH_URL_PREFIX)) {
        return new ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.length()), getClassLoader());
    }
    else {
        try {
            // 如果是URL方式，使用UrlResource作为bean文件的资源对象
            URL url = new URL(location);
            return new UrlResource(url);
        }
        catch (MalformedURLException ex) {
            // 如果都不是，那我们只能委托给子类由子类来决定使用什么样的资源对象了
            return getResourceByPath(location);
        }
    }
}
```

我们的FileSystemXmlApplicationContext本身就是是DefaultResourceLoader的实现类，他实现了以下的接口：

Java代码

```java
protected Resource getResourceByPath(String path) {
    if (path != null && path.startsWith("/")) {
        path = path.substring(1);
    }
    //这里使用文件系统资源对象来定义bean文件
```

```
    return new FileSystemResource(path);
}
```

这样代码就回到了FileSystemXmlApplicationContext中来，他提供了FileSystemResource来完成从文件系统得到配置文件的资源定义。这样，就可以从文件系统路径上对IOC配置文件进行加载 - 当然我们可以按照这个逻辑从任何地方加载，在Spring中我们看到它提供的各种资源抽象，比如ClassPathResource, URLResource,FileSystemResource等来供我们使用。上面我们看到的是定位Resource的一个过程，而这只是加载过程的一部分 - 我们回到AbstractBeanDefinitionReaderz中的loadDefinitions(resource)来看看得到代表bean文件的资源定义以后的载入过程,默认的我们使用XmlBeanDefinitionReader：

Java代码

```
public int loadBeanDefinitions(EncodedResource encodedResource) throws BeanDefinitionStoreExce
ption {
    .......
    try {
        //这里通过Resource得到InputStream的IO流
        InputStream inputStream = encodedResource.getResource().getInputStream();
        try {
            //从InputStream中得到XML的解析源
            InputSource inputSource = new InputSource(inputStream);
            if (encodedResource.getEncoding() != null) {
                inputSource.setEncoding(encodedResource.getEncoding());
            }
            //这里是具体的解析和注册过程
            return doLoadBeanDefinitions(inputSource, encodedResource.getResource());
        }
        finally {
            //关闭从Resource中得到的IO流
            inputStream.close();
        }
    }
       .........
}

protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
        throws BeanDefinitionStoreException {
    try {
        int validationMode = getValidationModeForResource(resource);
        //通过解析得到DOM，然后完成bean在IOC容器中的注册
        Document doc = this.documentLoader.loadDocument(
                inputSource, this.entityResolver, this.errorHandler, validationMode, this.na
mespaceAware);
        return registerBeanDefinitions(doc, resource);
    }
```

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

我们看到先把定义文件解析为DOM对象，然后进行具体的注册过程：

Java代码

```java
public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreException {
    // 这里定义解析器，使用XmlBeanDefinitionParser来解析xml方式的bean定义文件 - 现在的版本不用这个解析器了，使用的是XmlBeanDefinitionReader
    if (this.parserClass != null) {
        XmlBeanDefinitionParser parser =
                (XmlBeanDefinitionParser) BeanUtils.instantiateClass(this.parserClass);
        return parser.registerBeanDefinitions(this, doc, resource);
    }
    // 具体的注册过程,首先得到XmlBeanDefinitionReader,来处理xml的bean定义文件
    BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();
    int countBefore = getBeanFactory().getBeanDefinitionCount();
    documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
    return getBeanFactory().getBeanDefinitionCount() - countBefore;
}
```

具体的在BeanDefinitionDocumentReader中完成对，下面是一个简要的注册过程来完成bean定义文件的解析和IOC容器中bean的初始化

Java代码

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

    logger.debug("Loading bean definitions");
    Element root = doc.getDocumentElement();

    BeanDefinitionParserDelegate delegate = createHelper(readerContext, root);

    preProcessXml(root);
    parseBeanDefinitions(root, delegate);
    postProcessXml(root);
}

protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
    if (delegate.isDefaultNamespace(root.getNamespaceURI())) {
        //这里得到xml文件的子节点，比如各个bean节点
        NodeList nl = root.getChildNodes();
```

```
18.
19.         //这里对每个节点进行分析处理
20.         for (int i = 0; i < nl.getLength(); i++) {
21.             Node node = nl.item(i);
22.             if (node instanceof Element) {
23.                 Element ele = (Element) node;
24.                 String namespaceUri = ele.getNamespaceURI();
25.                 if (delegate.isDefaultNamespace(namespaceUri)) {
26.                     //这里是解析过程的调用，对缺省的元素进行分析比如bean元素
27.                     parseDefaultElement(ele, delegate);
28.                 }
29.                 else {
30.                     delegate.parseCustomElement(ele);
31.                 }
32.             }
33.         }
34.     } else {
35.         delegate.parseCustomElement(root);
36.     }
37. }
38.
39. private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {
40.     //这里对元素Import进行处理
41.     if (DomUtils.nodeNameEquals(ele, IMPORT_ELEMENT)) {
42.         importBeanDefinitionResource(ele);
43.     }
44.     else if (DomUtils.nodeNameEquals(ele, ALIAS_ELEMENT)) {
45.         String name = ele.getAttribute(NAME_ATTRIBUTE);
46.         String alias = ele.getAttribute(ALIAS_ATTRIBUTE);
47.         getReaderContext().getReader().getBeanFactory().registerAlias(name, alias);
48.         getReaderContext().fireAliasRegistered(name, alias, extractSource(ele));
49.     }
50.     //这里对我们最熟悉的bean元素进行处理
51.     else if (DomUtils.nodeNameEquals(ele, BEAN_ELEMENT)) {
52.         //委托给BeanDefinitionParserDelegate来完成对bean元素的处理，这个类包含了具体的bean解析的过
    程。
53.         // 把解析bean文件得到的信息放到BeanDefinition里，他是bean信息的主要载体，也是IOC容器的管理
    对象。
54.         BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
55.         if (bdHolder != null) {
56.             bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
57.             // 这里是向IOC容器注册，实际上是放到IOC容器的一个map里
58.             BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext().get
```

```
        Registry());

                // 这里向IOC容器发送事件，表示解析和注册完成。
                getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder))
        ;
            }
        }
    }
```

我们看到在parseBeanDefinition中对具体bean元素的解析式交给BeanDefinitionParserDelegate来完成的，下面我们看看解析完的bean是怎样在IOC容器中注册的：
在BeanDefinitionReaderUtils调用的是：

Java代码

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

    // 这里得到需要注册bean的名字;
    String beanName = bdHolder.getBeanName();
    //这是调用IOC来注册的bean的过程，需要得到BeanDefinition
    beanFactory.registerBeanDefinition(beanName, bdHolder.getBeanDefinition());

    // 别名也是可以通过IOC容器和bean联系起来的进行注册
    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：

Java代码

```
//---------------------------------------------------------------------
// 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
```

```
9.      //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。
10.     Object oldBeanDefinition = this.beanDefinitionMap.get(beanName);
11.     if (oldBeanDefinition != null) {
12.         if (!this.allowBeanDefinitionOverriding) {
13.         ..........
14.     }
15.     else {
16.         //把bean的名字加到IOC容器中去
17.         this.beanDefinitionNames.add(beanName);
18.     }
19.     //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。
20.     this.beanDefinitionMap.put(beanName, beanDefinition);
21.     removeSingleton(beanName);
22. }
```

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。
从上面的代码来看，我们总结一下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.

在使用IOC容器的时候，我们注意到除了少量粘合代码，绝大多数以正确IoC风格编写的应用程序代码完全不用关心如何到达工厂，因为容器将把这些对象与容器管理的其他对象钩在一起。基本的策略是把工厂放到已知的地方，最好是放在对预期使用的上下文有意义的地方，以及代码将实际需要访问工厂的地方。 Spring本身提供了对声明式载入web应用程序用法的应用程序上下文，并将其存储在ServletContext中的框架实现。具体可以参见以后的文章。
在使用Spring IOC容器的时候我们还需要区别两个概念:
Beanfactory 和Factory bean，其中BeanFactory指的是IOC容器的编程抽象，比如ApplicationContext， XmlBeanFactory等，这些都是IOC容器的具体表现，需要使用什么样的容器由客户决定但Spring为我们提供了丰富的选择。而 FactoryBean只是一个可以在IOC容器中被管理的一个bean,是对各种处理过程和资源使用的抽象,Factory bean在需要时产生另一个对象，而不返回FactoryBean本省，我们可以把它看成是一个抽象工厂，对它的调用返回的是工厂生产的产品。所有的 Factory bean都实现特殊的org.springframework.beans.factory.FactoryBean接口，当使用容器中factory bean的时候，该容器不会返回factory bean本身，而是返回其生成的对象。Spring包括了大部分的通用资源和服务访问抽象的Factory bean的实现，其中包括:

对JNDI查询的处理，对代理对象的处理，对事务性代理的处理，对RMI代理的处理等，这些我们都可以看成是具体的工厂，看成是SPRING为我们建立好的工厂。也就是说Spring通过使用抽象工厂模式为我们准备了一系列工厂来生产一些特定的对象，免除我们手工重复的工作，我们要使用时只需要在IOC容器里配置好就能很方便的使用了。
现在我们来看看在Spring的事件机制，Spring中有3个标准事件，ContextRefreshEvent,ContextCloseEvent,RequestHandledEvent他们通过ApplicationEvent接口，同样的如果需要自定义时间也只需要实现ApplicationEvent接口，参照ContextCloseEvent的实现可以定制自己的事件实现：

Java代码

```
1.  public class ContextClosedEvent extends ApplicationEvent {
2.
3.      public ContextClosedEvent(ApplicationContext source) {
4.          super(source);
5.      }
6.
7.      public ApplicationContext getApplicationContext() {
8.          return (ApplicationContext) getSource();
9.      }
10. }
```

可以通过显现ApplicationEventPublishAware接口，将事件发布器耦合到ApplicationContext这样可以使用 ApplicationContext框架来传递和消费消息,然后在ApplicationContext中配置好bean就可以了，在消费消息的过程中，接受者通过实现ApplicationListener接收消息。
比如可以直接使用Spring的ScheduleTimerTask和TimerFactoryBean作为定时器定时产生消息，具体可以参见《Spring框架高级编程》。
TimerFactoryBean是一个工厂bean，对其中的ScheduleTimerTask进行处理后输出，参考ScheduleTimerTask的实现发现它最后调用的是jre的TimerTask：

Java代码

```
1.  public void setRunnable(Runnable timerTask) {
2.      this.timerTask = new DelegatingTimerTask(timerTask);
3.  }
```

在书中给出了一个定时发送消息的例子，当然可以可以通过定时器作其他的动作，有两种方法：
1.定义MethodInvokingTimerTaskFactoryBean定义要执行的特定bean的特定方法，对需要做什么进行封装定义；
2.定义TimerTask类，通过extends TimerTask来得到，同时对需要做什么进行自定义
然后需要定义具体的定时器参数，通过配置ScheduledTimerTask中的参数和timerTask来完成，以下是它需要定义的具体属性，timerTask是在前面已经定义好的bean

Java代码

```
1.  private TimerTask timerTask;
2.
3.  private long delay = 0;
4.
```

```
5.  private long period = 0;
6.
7.  private boolean fixedRate = false;
```

最后，需要在ApplicationContext中注册，需要把ScheduledTimerTask配置到FactoryBean - TimerFactoryBean，这样就由IOC容器来管理定时器了。参照
TimerFactoryBean的属性，可以定制一组定时器。

Java代码

```
1.  public class TimerFactoryBean implements FactoryBean, InitializingBean, DisposableBean {
2.
3.      protected final Log logger = LogFactory.getLog(getClass());
4.
5.      private ScheduledTimerTask[] scheduledTimerTasks;
6.
7.      private boolean daemon = false;
8.
9.      private Timer timer;
10.
11.     ..........
12. }
```

如果要发送时间我们只需要在定义好的ScheduledTimerTasks中publish定义好的事件就可以了。具体可以参考书中例子的实现，这里只是结合FactoryBean的原理做一些解释。如果结合事件和定时器机制，我们可以很方便的实现heartbeat(看门狗），书中给出了这个例子，这个例子实际上结合了Spring事件和定时机制的使用两个方面的知识 - 当然了还有IOC容器的知识（任何Spring应用我想都逃不掉IOC的魔爪：）

Spring源代码解析(二)：IoC容器在Web容 ... | spring源码分析-XmlBeanFactory导读

评论 共 4 条 发表评论

4 楼 apache2008 2009-10-20 10:52 引用

牛人，绝对的牛人

3 楼 chinaboby2008 2009-10-11 16:30 引用

java功底很牢啊。 

2 楼 allanouyang 2009-06-29 15:07 引用

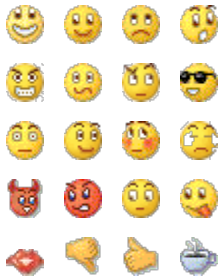
谢谢

1 楼 wanjianfei 2009-05-26 19:42 引用

太有用了，学习中...

发表评论

表情图标



B I U Quote Code List Img URL Flash Table

字体颜色: 标准 字体大小: 标准 对齐: 标准

提示：选择您需要装饰的文字, 按上列按钮即可添加上相应的标签

您还没有登录，请登录后发表评论(快捷键 Alt+S / Ctrl+Enter) 提交