一、IOC 容器

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 b
    ean 实例的 Class 类型和需要的不同的话。
12.     Object getBean(String name, Class requiredType) throws BeansException;
13.
14.     //这里提供对 bean 的检索，看看是否在 IOC 容器有这个名字的 bean
15.     boolean containsBean(String name);
16.
17.     //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
18.     boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
19.
20.     //这里对得到 bean 实例的 Class 类型
21.     Class getType(String name) throws NoSuchBeanDefinitionException;
22.
23.     //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
24.     String[] getAliases(String name);
25.
26. }
```

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

```
1.  ClassPathResource res = new ClassPathResource("beans.xml");
2.  DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
3.  XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
4.  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

```
1.  public class XmlBeanFactory extends DefaultListableBeanFactory {
2.      //这里为容器定义了一个默认使用的 bean 定义读取器
3.      private final XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(this);
4.      public XmlBeanFactory(Resource resource) throws BeansException {
5.          this(resource, null);
6.      }
7.      //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。
8.      public XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory) throws BeansExceptio
    n {
9.          super(parentBeanFactory);
10.         this.reader.loadBeanDefinitions(resource);
11.     }
```

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

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

调用的是它初始化代码：
Java 代码

```
1.  public FileSystemXmlApplicationContext(String[] configLocations, boolean refresh, ApplicationContex
    t parent)
2.          throws BeansException {
3.      super(parent);
4.      this.configLocations = configLocations;
5.      if (refresh) {
6.          //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
7.          refresh();
8.      }
9.  }
```

refresh 的模板在 AbstractApplicationContext:
Java 代码

```
1.  public void refresh() throws BeansException, IllegalStateException {
2.      synchronized (this.startupShutdownMonitor) {
3.          synchronized (this.activeMonitor) {
4.              this.active = true;
5.          }
6.
7.          // 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
8.          refreshBeanFactory();
9.          ............
10.  }
```

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

```
1.  protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws IOException {
2.      // 这里使用 XMLBeanDefinitionReader 来载入 bean 定义信息的 XML 文件
3.      XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);
4.
5.      //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
```

```
6.      ///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 XmlBeanDefi
    nitionReader
7.      beanDefinitionReader.setResourceLoader(this);
8.      beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));
9.
10.     initBeanDefinitionReader(beanDefinitionReader);
11.     //这里转到定义好的 XmlBeanDefinitionReader 中对载入 bean 信息进行处理
12.     loadBeanDefinitions(beanDefinitionReader);
13. }
```

转到 beanDefinitionReader 中进行处理：

Java 代码

```
1.  protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException, IOExcepti
    on {
2.      Resource[] configResources = getConfigResources();
3.      if (configResources != null) {
4.          //调用 XmlBeanDefinitionReader 来载入 bean 定义信息。
5.          reader.loadBeanDefinitions(configResources);
6.      }
7.      String[] configLocations = getConfigLocations();
8.      if (configLocations != null) {
9.          reader.loadBeanDefinitions(configLocations);
10.     }
11. }
```

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

Java 代码

```
1.  public int loadBeanDefinitions(String location) throws BeanDefinitionStoreException {
2.   //这里得到当前定义的 ResourceLoader,默认的我们使用 DefaultResourceLoader
3.   ResourceLoader resourceLoader = getResourceLoader();
4.   ...........//如果没有找到我们需要的 ResourceLoader，直接抛出异常
5.      if (resourceLoader instanceof ResourcePatternResolver) {
6.          // 这里处理我们在定义位置时使用的各种 pattern,需要 ResourcePatternResolver 来完成
7.          try {
8.              Resource[] resources = ((ResourcePatternResolver) resourceLoader).getResources(locatio
    n);
9.              int loadCount = loadBeanDefinitions(resources);
10.             return loadCount;
11.         }
12.       .........
13.     }
14.     else {
15.         // 这里通过 ResourceLoader 来完成位置定位
16.         Resource resource = resourceLoader.getResource(location);
17.         // 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
18.         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

```
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

```
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

```
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 BeanDefinitionStoreExceptio
n {
    .......
    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.namespace
Aware);
        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

```
1.  public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreExcep
    tion {
2.      // 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
3.      if (this.parserClass != null) {
4.          XmlBeanDefinitionParser parser =
5.                  (XmlBeanDefinitionParser) BeanUtils.instantiateClass(this.parserClass);
6.          return parser.registerBeanDefinitions(this, doc, resource);
7.      }
8.      // 具体的注册过程,首先得到 XmlBeanDefinitionReader,来处理 xml 的 bean 定义文件
9.      BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();
10.     int countBefore = getBeanFactory().getBeanDefinitionCount();
11.     documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
12.     return getBeanFactory().getBeanDefinitionCount() - countBefore;
13. }
```

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

```
1.  public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {
2.      this.readerContext = readerContext;
3.
4.      logger.debug("Loading bean definitions");
5.      Element root = doc.getDocumentElement();
6.
7.      BeanDefinitionParserDelegate delegate = createHelper(readerContext, root);
8.
9.      preProcessXml(root);
10.     parseBeanDefinitions(root, delegate);
11.     postProcessXml(root);
12. }
13.
14. protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
15.     if (delegate.isDefaultNamespace(root.getNamespaceURI())) {
16.         //这里得到 xml 文件的子节点，比如各个 bean 节点
17.         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;
```

```
                String namespaceUri = ele.getNamespaceURI();
                if (delegate.isDefaultNamespace(namespaceUri)) {
                    //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
                    parseDefaultElement(ele, delegate);
                }
                else {
                    delegate.parseCustomElement(ele);
                }
            }
        }
    } else {
        delegate.parseCustomElement(root);
    }
}

private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {
    //这里对元素 Import 进行处理
    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));
    }
    //这里对我们最熟悉的 bean 元素进行处理
    else if (DomUtils.nodeNameEquals(ele, BEAN_ELEMENT)) {
        //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。
        // 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。
        BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
        if (bdHolder != null) {
            bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
            // 这里是向 IOC 容器注册，实际上是放到 IOC 容器的一个 map 里
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext().getRegist
ry());

            // 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

```
1.  public static void registerBeanDefinition(
2.          BeanDefinitionHolder bdHolder, BeanDefinitionRegistry beanFactory) throws BeansExceptio
    n {
3.
4.      // 这里得到需要注册 bean 的名字;
5.      String beanName = bdHolder.getBeanName();
6.      //这是调用 IOC 来注册的 bean 的过程，需要得到 BeanDefinition
7.      beanFactory.registerBeanDefinition(beanName, bdHolder.getBeanDefinition());
8.
9.      // 别名也是可以通过 IOC 容器和 bean 联系起来的进行注册
10.     String[] aliases = bdHolder.getAliases();
11.     if (aliases != null) {
12.         for (int i = 0; i < aliases.length; i++) {
13.             beanFactory.registerAlias(beanName, aliases[i]);
14.         }
15.     }
16. }
```

我们看看 XmlBeanFactory 中的注册实现:

Java 代码

```
1.  //---------------------------------------------------------------------
2.  // 这里是 IOC 容器对 BeanDefinitionRegistry 接口的实现
3.  //---------------------------------------------------------------------
4.
5.  public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
6.          throws BeanDefinitionStoreException {
7.
8.      ...../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：）

二、IoC 容器在 Web 容器中的启动

以下引用自博客：http://jiwenke-spring.blogspot.com/

上面我们分析了IOC容器本身的实现，下面我们看看在典型的web环境中，Spring IOC容器是怎样被载入和起作用的。

简单的说，在web容器中，通过ServletContext为Spring的IOC容器提供宿主环境，对应的建立起一个IOC容器的体系。其中，首先需要建立的是根上下文，这个上下文持有的对象可以有业务对象，数据存取对象，资源，事物管理器等各种中间层对象。在这个上下文的基础上，和web MVC相关还会有一个上下文来保存控制器之类的MVC对象，这样就构成了一个层次化的上下文结构。在web容器中启动Spring应用程序就是一个建立这个上下文体系的过程。Spring为web应用提供了上下文的扩展接口WebApplicationContext:

Java 代码

```
1.  public interface WebApplicationContext extends ApplicationContext {
2.      //这里定义的常量用于在 ServletContext 中存取根上下文
3.      String ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE = WebApplicationContext.class.getName() + ".ROOT
    ";
4.      ......
5.      //对 WebApplicationContext 来说，需要得到 Web 容器的 ServletContext
6.      ServletContext getServletContext();
7.  }
```

而一般的启动过程，Spring 会使用一个默认的实现，XmlWebApplicationContext - 这个上下文实现作为在 web 容器中的根上下文容器被建立起来，具体的建立过程在下面我们会详细分析。

Java 代码

```
1.  public class XmlWebApplicationContext extends AbstractRefreshableWebApplicationContext {
2.
3.      /** 这是和 web 部署相关的位置信息，用来作为默认的根上下文 bean 定义信息的存放位置*/
4.      public static final String DEFAULT_CONFIG_LOCATION = "/WEB-INF/applicationContext.xml";
5.      public static final String DEFAULT_CONFIG_LOCATION_PREFIX = "/WEB-INF/";
6.      public static final String DEFAULT_CONFIG_LOCATION_SUFFIX = ".xml";
7.
8.      //我们又看到了熟悉的 loadBeanDefinition,就像我们前面对 IOC 容器的分析中一样，这个加载工程在容器的 refres
    h()的时候启动。
9.      protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws IOExceptio
    n {
10.         //对于 XmlWebApplicationContext,当然使用的是 XmlBeanDefinitionReader 来对 bean 定义信息来进行解
    析
11.         XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);
12.
13.         beanDefinitionReader.setResourceLoader(this);
14.         beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));
15.
16.         initBeanDefinitionReader(beanDefinitionReader);
17.         loadBeanDefinitions(beanDefinitionReader);
18.     }
19.
20.     protected void initBeanDefinitionReader(XmlBeanDefinitionReader beanDefinitionReader) {
```

```
21.     }
22.     //使用 XmlBeanDefinitionReader 来读入 bean 定义信息
23.     protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException, IOExc
    eption {
24.         String[] configLocations = getConfigLocations();
25.         if (configLocations != null) {
26.             for (int i = 0; i < configLocations.length; i++) {
27.                 reader.loadBeanDefinitions(configLocations[i]);
28.             }
29.         }
30.     }
31.     //这里取得 bean 定义信息位置，默认的地方是/WEB-INF/applicationContext.xml
32.     protected String[] getDefaultConfigLocations() {
33.         if (getNamespace() != null) {
34.             return new String[] {DEFAULT_CONFIG_LOCATION_PREFIX + getNamespace() + DEFAULT_CONFIG_L
    OCATION_SUFFIX};
35.         }
36.         else {
37.             return new String[] {DEFAULT_CONFIG_LOCATION};
38.         }
39.     }
40. }
```

对于一个 Spring 激活的 web 应用程序，可以通过使用 Spring 代码声明式的指定在 web 应用程序启动时载入应用程序上下文（WebApplicationContext),Spring 的 ContextLoader 是提供这样性能的类，我们可以使用 ContextLoaderServlet 或者 ContextLoaderListener 的启动时载入的 Servlet 来实例化 Spring IOC 容器 - 为什么会有两个不同的类来装载它呢，这是因为它们的使用需要区别不同的 Servlet 容器支持的 Serlvet 版本。但不管是 ContextLoaderSevlet 还是 ContextLoaderListener 都使用 ContextLoader 来完成实际的 WebApplicationContext 的初始化工作。这个 ContextLoder 就像是 Spring Web 应用程序在 Web 容器中的加载器 booter。当然这些 Servlet 的具体使用我们都要借助 web 容器中的部署描述符来进行相关的定义。

下面我们使用 ContextLoaderListener 作为载入器作一个详细的分析，这个 Servlet 的监听器是根上下文被载入的地方，也是整个 Spring web 应用加载上下文的第一个地方；从加载过程我们可以看到，首先从 Servlet 事件中得到 ServletContext，然后可以读到配置好的在 web.xml 的中的各个属性值，然后 ContextLoder 实例化 WebApplicationContext 并完成其载入和初始化作为根上下文。当这个根上下文被载入后，它被绑定到 web 应用程序的 ServletContext 上。任何需要访问该 ApplicationContext 的应用程序代码都可以从 WebApplicationContextUtils 类的静态方法来得到：

Java 代码

```
1.  WebApplicationContext getWebApplicationContext(ServletContext sc)
```

以 Tomcat 作为 Servlet 容器为例，下面是具体的步骤：

1.Tomcat 启动时需要从 web.xml 中读取启动参数，在 web.xml 中我们需要对 ContextLoaderListener 进行配置，对于在 web 应用启动入口是在 ContextLoaderListener 中的初始化部分；从 Spring MVC 上看，实际上在 web 容器中维护了一系列的 IOC 容器，其中在 ContextLoader 中载入的 IOC 容器作为根上下文而存在于 ServletContext 中。

Java 代码

```
1.  //这里对根上下文进行初始化。
2.  public void contextInitialized(ServletContextEvent event) {
3.      //这里创建需要的 ContextLoader
4.      this.contextLoader = createContextLoader();
```

```
    //这里使用 ContextLoader 对根上下文进行载入和初始化
    this.contextLoader.initWebApplicationContext(event.getServletContext());
}
```

通过 ContextLoader 建立起根上下文的过程,我们可以在 ContextLoader 中看到：

Java 代码

```
public WebApplicationContext initWebApplicationContext(ServletContext servletContext)
        throws IllegalStateException, BeansException {
    //这里先看看是不是已经在 ServletContext 中存在上下文,如果有说明前面已经被载入过，或者是配置文件有错误。
    if (servletContext.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE) !
= null) {
    //直接抛出异常
    .........
    }

    ..............
    try {
        // 这里载入根上下文的父上下文
        ApplicationContext parent = loadParentContext(servletContext);

        //这里创建根上下文作为整个应用的上下文同时把它存到 ServletContext 中去,注意这里使用的 ServletContex
t 的属性值是
        //ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE,以后的应用都是根据这个属性值来取得根上下文的 - 往往作为
自己上下文的父上下文
        this.context = createWebApplicationContext(servletContext, parent);
        servletContext.setAttribute(
                WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, this.context);
        ..........

        return this.context;
    }
       ............
}
```

建立根上下文的父上下文使用的是下面的代码，取决于在 web.xml 中定义的参数：locatorFactorySelector，这是一个可选参数：

Java 代码

```
protected ApplicationContext loadParentContext(ServletContext servletContext)
        throws BeansException {

    ApplicationContext parentContext = null;

    String locatorFactorySelector = servletContext.getInitParameter(LOCATOR_FACTORY_SELECTOR_PARA
M);
    String parentContextKey = servletContext.getInitParameter(LOCATOR_FACTORY_KEY_PARAM);

    if (locatorFactorySelector != null) {
```

```
        BeanFactoryLocator locator = ContextSingletonBeanFactoryLocator.getInstance(locatorFactoryS
elector);
        ........
        //得到根上下文的父上下文的引用
        this.parentContextRef = locator.useBeanFactory(parentContextKey);
        //这里建立得到根上下文的父上下文
        parentContext = (ApplicationContext) this.parentContextRef.getFactory();
    }

    return parentContext;
}
```

得到根上下文的父上下文以后，就是根上下文的创建过程：

Java 代码

```
protected WebApplicationContext createWebApplicationContext(
        ServletContext servletContext, ApplicationContext parent) throws BeansException {
    //这里需要确定我们载入的根 WebApplication 的类型，由在 web.xml 中配置的 contextClass 中配置的参数可以决定
我们需要载入什么样的 ApplicationContext,
    //如果没有使用默认的。
    Class contextClass = determineContextClass(servletContext);
    .........
    //这里就是上下文的创建过程
    ConfigurableWebApplicationContext wac =
            (ConfigurableWebApplicationContext) BeanUtils.instantiateClass(contextClass);
    //这里保持对父上下文和 ServletContext 的引用到根上下文中
    wac.setParent(parent);
    wac.setServletContext(servletContext);

    //这里从 web.xml 中取得相关的初始化参数
    String configLocation = servletContext.getInitParameter(CONFIG_LOCATION_PARAM);
    if (configLocation != null) {
        wac.setConfigLocations(StringUtils.tokenizeToStringArray(configLocation,
                ConfigurableWebApplicationContext.CONFIG_LOCATION_DELIMITERS));
    }
   //这里对 WebApplicationContext 进行初始化，我们又看到了熟悉的 refresh 调用。
    wac.refresh();
    return wac;
}
```

初始化根 ApplicationContext 后将其存储到 SevletContext 中去以后,这样就建立了一个全局的关于整个应用的上下文。这个根上下文会被以后的 DispatcherServlet 初始化自己的时候作为自己 ApplicationContext 的父上下文。这个在对 DispatcherServlet 做分析的时候我们可以看看到。

3.完成对 ContextLoaderListener 的初始化以后， Tomcat 开始初始化 DispatchServlet，- 还记得我们在 web.xml 中队载入次序进行了定义。DispatcherServlet 会建立自己的 ApplicationContext,同时建立这个自己的上下文的时候会从 ServletContext 中得到根上下文作为父上下文,然后再对自己的上下文进行初始化，并最后存到 ServletContext 中去供以后检索和使用。

可以从 DispatchServlet 的父类 FrameworkServlet 的代码中看到大致的初始化过程，整个 ApplicationContext 的创建过程和 ContextLoder 创建的过程相类似：

Java 代码

```
protected final void initServletBean() throws ServletException, BeansException {
    .........
    try {
        //这里是对上下文的初始化过程。
        this.webApplicationContext = initWebApplicationContext();
        //在完成对上下文的初始化过程结束后，根据 bean 配置信息建立 MVC 框架的各个主要元素
        initFrameworkServlet();
    }
   ........
}
```

对 initWebApplicationContext()调用的代码如下：

Java 代码

```
protected WebApplicationContext initWebApplicationContext() throws BeansException {
    //这里调用 WebApplicationContextUtils 静态类来得到根上下文
    WebApplicationContext parent = WebApplicationContextUtils.getWebApplicationContext(getServletCo
ntext());

    //创建当前 DispatcherServlet 的上下文，其上下文种类使用默认的在 FrameworkServlet 定义好的：DEFAULT_CON
TEXT_CLASS = XmlWebApplicationContext.class;
    WebApplicationContext wac = createWebApplicationContext(parent);
    ........
    if (isPublishContext()) {
        //把当前建立的上下文存到 ServletContext 中去，注意使用的属性名是和当前 Servlet 名相关的。
        String attrName = getServletContextAttributeName();
        getServletContext().setAttribute(attrName, wac);
    }
    return wac;
}
```

其中我们看到调用了 WebApplicationContextUtils 的静态方法得到根 ApplicationContext:

Java 代码

```
    public static WebApplicationContext getWebApplicationContext(ServletContext sc) {
        //很简单，直接从 ServletContext 中通过属性名得到根上下文
        Object attr = sc.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUT
E);
        .......
        return (WebApplicationContext) attr;
    }
然后创建 DispatcherServlet 自己的 WebApplicationContext:
    protected WebApplicationContext createWebApplicationContext(WebApplicationContext parent)
            throws BeansException {
        .......
```

```
        //这里使用了 BeanUtils 直接得到 WebApplicationContext,ContextClass 是前面定义好的 DEFAULT_CONTEXT_CLASS =
        //XmlWebApplicationContext.class;
        ConfigurableWebApplicationContext wac =
                (ConfigurableWebApplicationContext) BeanUtils.instantiateClass(getContextClass());

        //这里配置父上下文，就是在 ContextLoader 中建立的根上下文
        wac.setParent(parent);

        //保留 ServletContext 的引用和相关的配置信息。
        wac.setServletContext(getServletContext());
        wac.setServletConfig(getServletConfig());
        wac.setNamespace(getNamespace());

        //这里得到 ApplicationContext 配置文件的位置
        if (getContextConfigLocation() != null) {
            wac.setConfigLocations(
                StringUtils.tokenizeToStringArray(
                            getContextConfigLocation(), ConfigurableWebApplicationContext.CONFIG_LOCATION_DELIMITERS));
        }

        //这里调用 ApplicationContext 的初始化过程，同样需要使用 refresh()
        wac.refresh();
        return wac;
    }
```

4. 然后就是 DispatchServlet 中对 Spring MVC 的配置过程，首先对配置文件中的定义元素进行配置 - 请注意这个时候我们的 WebApplicationContext 已经建立起来了，也意味着 DispatcherServlet 有自己的定义资源，可以需要从 web.xml 中读取 bean 的配置信息，通常我们会使用单独的 xml 文件来配置 MVC 中各个要素定义，这里和 web 容器相关的加载过程实际上已经完成了，下面的处理和普通的 Spring 应用程序的编写没有什么太大的差别，我们先看看 MVC 的初始化过程：

Java 代码

```
protected void initFrameworkServlet() throws ServletException, BeansException {
    initMultipartResolver();
    initLocaleResolver();
    initThemeResolver();
    initHandlerMappings();
    initHandlerAdapters();
    initHandlerExceptionResolvers();
    initRequestToViewNameTranslator();
    initViewResolvers();
}
```

5. 这样 MVC 的框架就建立起来了，DispatchServlet 对接受到的 HTTP Request 进行分发处理由 doService()完成，具体的 MVC 处理过

程我们在 doDispatch()中完成，其中包括使用 Command 模式建立执行链，显示模型数据等，这些处理我们都可以在 DispatcherServlet 的代码中看到：

Java 代码

```
protected void doService(HttpServletRequest request, HttpServletResponse response) throws Exception {
    ......
    try {
        doDispatch(request, response);
    }
   .......
}
```

实际的请求分发由 doDispatch(request,response)来完成：

Java 代码

```
protected void doDispatch(final HttpServletRequest request, HttpServletResponse response) throws Exception {
     .......
     // 这是 Spring 定义的执行链，里面放了映射关系对应的 handler 和定义的相关拦截器。
     HandlerExecutionChain mappedHandler = null;

      ......
      try {
          //我们熟悉的 ModelAndView 在这里出现了。
          ModelAndView mv = null;
          try {
              processedRequest = checkMultipart(request);

              //这里更具 request 中的参数和映射关系定义决定使用的 handler
              mappedHandler = getHandler(processedRequest, false);

              ......
              //这里是 handler 的调用过程，类似于 Command 模式中的 execute.
              HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
              mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

              .......
          //这里将模型数据通过视图进行展现
          if (mv != null && !mv.wasCleared()) {
              render(mv, processedRequest, response);
          }
            ........
  }
```

这样具体的 MVC 模型的实现就由 bean 配置文件里定义好的 view resolver,handler 这些类来实现用户代码的功能。

总结上面的过程，我们看到在 web 容器中，ServletContext 可以持有一系列的 web 上下文，而在整个 web 上下文中存在一个根上下文来作为其它 Servlet 上下文的父上下文。这个根上下文是由 ContextLoader 载入并进行初始化的，对于我们的 web 应用，

DispatcherSerlvet 载入并初始化自己的上下文,这个上下文的父上下文是根上下文，并且我们也能从 ServletContext 中根据 Servlet 的名字来检索到我们需要的对应于这个 Servlet 的上下文，但是根上下文的名字是由 Spring 唯一确定的。这个 DispactcherServlet 建立的上下文就是我们开发 Spring MVC 应用的 IOC 容器。

具体的 web 请求处理在上下文体系建立完成以后由 DispactcherServlet 来完成，上面对 MVC 的运作做了一个大致的描述，下面我们会具体就 SpringMVC 的框架实现作一个详细的分析。

三、Spring JDBC

引用自博客：http://jiwenke-spring.blogspot.com/

下面我们看看Spring JDBC相关的实现，

在Spring中，JdbcTemplate是经常被使用的类来帮助用户程序操作数据库，在JdbcTemplate为用户程序提供了许多便利的数据库操作方法，比如查询，更新等，而且在Spring中，有许多类似 JdbcTemplate的模板,比如HibernateTemplate等等 - 看来这是Rod.Johnson的惯用手法，一般而言这种Template中都是通过回调函数CallBack类的使用来完成功能的，客户需要在回调接口中实现自己需要的定制行为，比如使用客户想要用的SQL语句等。不过往往Spring通过这种回调函数的实现已经为我们提供了许多现成的方法供客户使用。一般来说回调函数的用法采用匿名类的方式来实现，比如：

Java 代码

```
1.  JdbcTemplate = new JdbcTemplate(datasource);
2.  jdbcTemplate.execute(new CallBack(){
3.              public CallbackInterfacedoInAction(){
4.                 ......
5.                 //用户定义的代码或者说 Spring 替我们实现的代码
6.              }
7.  }
```

在模板中嵌入的是需要客户化的代码，由 Spring 来作或者需要客户程序亲自动手完成。下面让我们具体看看在 JdbcTemplate 中的代码是怎样完成使命的，我们举 JdbcTemplate.execute()为例，这个方法是在 JdbcTemplate 中被其他方法调用的基本方法之一，客户程序往往用这个方法来执行基本的 SQL 语句：

Java 代码

```
1.  public Object execute(ConnectionCallback action) throws DataAccessException {
2.      //这里得到数据库联接
3.      Connection con = DataSourceUtils.getConnection(getDataSource());
4.      try {
5.          Connection conToUse = con;
6.          //有些特殊的数据库，需要我们使用特别的方法取得 datasource
7.          if (this.nativeJdbcExtractor != null) {
8.              // Extract native JDBC Connection, castable to OracleConnection or the like.
9.              conToUse = this.nativeJdbcExtractor.getNativeConnection(con);
10.         }
11.         else {
12.             // Create close-suppressing Connection proxy, also preparing returned Statements.
13.             conToUse = createConnectionProxy(con);
14.         }
15.     //这里调用的是传递进来的匿名类的方法，也就是用户程序需要实现 CallBack 接口的地方。
16.         return action.doInConnection(conToUse);
17.     }
18.     catch (SQLException ex) {
19.         //如果捕捉到数据库异常，把数据库联接释放，同时抛出一个经过 Spring 转换过的 Spring 数据库异常，
20.         //我们知道，Spring 做了一个有意义的工作是把这些数据库异常统一到自己的异常体系里了。
21.         DataSourceUtils.releaseConnection(con, getDataSource());
22.         con = null;
23.         throw getExceptionTranslator().translate("ConnectionCallback", getSql(action), ex);
```

```
    }
    finally {
        //最后不管怎样都会把数据库连接释放
        DataSourceUtils.releaseConnection(con, getDataSource());
    }
}
```

对于 JdbcTemplate 中给出的其他方法，比如 query,update,execute 等的实现，我们看看 query()：

Java 代码

```
public Object query(PreparedStatementCreator psc, final PreparedStatementSetter pss, final ResultSetExtractor rse)
        throws DataAccessException {
    ..........
    //这里调用了我们上面看到的 execute()基本方法,然而这里的回调实现是 Spring 为我们完成的查询过程
    return execute(psc, new PreparedStatementCallback() {
        public Object doInPreparedStatement(PreparedStatement ps) throws SQLException {
            //准备查询结果集
            ResultSet rs = null;
            try {
            //这里配置 SQL 参数
                if (pss != null) {
                    pss.setValues(ps);
                }
          //这里执行的 SQL 查询
                rs = ps.executeQuery();
                ResultSet rsToUse = rs;
                if (nativeJdbcExtractor != null) {
                    rsToUse = nativeJdbcExtractor.getNativeResultSet(rs);
                }
         //返回需要的记录集合
                return rse.extractData(rsToUse);
            }
            finally {
        //最后关闭查询的纪录集，对数据库连接的释放在 execute()中释放，就像我们在上面分析的看到那样。
                JdbcUtils.closeResultSet(rs);
                if (pss instanceof ParameterDisposer) {
                    ((ParameterDisposer) pss).cleanupParameters();
                }
            }
        }
    });
}
```

辅助类 DataSourceUtils 来用来对数据库连接进行管理的主要工具，比如打开和关闭数据库连接等基本操作：

Java 代码

```
public static Connection doGetConnection(DataSource dataSource) throws SQLException {
```

```
    //把对数据库连接放到事务管理里面进行管理
     ConnectionHolder conHolder = (ConnectionHolder) TransactionSynchronizationManager.getResource(dataSource);
     if (conHolder != null && (conHolder.hasConnection() || conHolder.isSynchronizedWithTransaction())) {
         conHolder.requested();
         if (!conHolder.hasConnection()) {
             logger.debug("Fetching resumed JDBC Connection from DataSource");
             conHolder.setConnection(dataSource.getConnection());
         }
         return conHolder.getConnection();
     }
     // 这里得到需要的数据库连接，在配置文件中定义好的。
     logger.debug("Fetching JDBC Connection from DataSource");
     Connection con = dataSource.getConnection();

     if (TransactionSynchronizationManager.isSynchronizationActive()) {
         logger.debug("Registering transaction synchronization for JDBC Connection");
         // Use same Connection for further JDBC actions within the transaction.
         // Thread-bound object will get removed by synchronization at transaction completion.
         ConnectionHolder holderToUse = conHolder;
         if (holderToUse == null) {
             holderToUse = new ConnectionHolder(con);
         }
         else {
             holderToUse.setConnection(con);
         }
         holderToUse.requested();
         TransactionSynchronizationManager.registerSynchronization(
                 new ConnectionSynchronization(holderToUse, dataSource));
         holderToUse.setSynchronizedWithTransaction(true);
         if (holderToUse != conHolder) {
             TransactionSynchronizationManager.bindResource(dataSource, holderToUse);
         }
     }

     return con;
 }
```

那我们实际的 DataSource 对象是怎样得到的？很清楚我们需要在上下文中进行配置：它作为 JdbcTemplate 父类 JdbcAccessor 的属性存在：

Java 代码

```
 public abstract class JdbcAccessor implements InitializingBean {

     /** 这里是我们依赖注入数据库数据源的地方。 */
     private DataSource dataSource;
```

```
5.
6.      /** Helper to translate SQL exceptions to DataAccessExceptions */
7.      private SQLExceptionTranslator exceptionTranslator;
8.
9.      private boolean lazyInit = true;
10.
11.     ........
12. }
```

而对于 DataSource 的缓冲池实现，我们通过定义 Apache Jakarta Commons DBCP 或者 C3P0 提供的 DataSource 来完成，然后只要在上下文中配置好就可以使用了。从上面我们看到 JdbcTemplate 提供了许多简单查询和更新功能，但是如果需要更高层次的抽象，以及更面向对象的方法来访问数据库。Spring 为我们提供了 org.springframework.jdbc.object 包，这里面包含了 SqlQuery,SqlMappingQuery, SqlUpdate 和 StoredProcedure 等类，这些类都是 Spring JDBC 应用程序可以使用的主要类，但我们要注意使用这些类的时候，用户需要为他们配置好一个 JdbcTemplate 作为其基本的操作的实现。
比如说我们使用 MappingSqlQuery 来将表数据直接映射到一个对象集合 - 具体可以参考书中的例子
1.我们需要建立 DataSource 和 sql 语句并建立持有这些对象的 MappingSqlQuery 对象
2.然后我们需要定义传递的 SqlParameter,具体的实现我们在 MappingSqlQuery 的父类 RdbmsOperation 中可以找到：
Java 代码

```
1.  public void declareParameter(SqlParameter param) throws InvalidDataAccessApiUsageException {
2.      //如果声明已经被编译过，则该声明无效
3.      if (isCompiled()) {
4.          throw new InvalidDataAccessApiUsageException("Cannot add parameters once query is compiled
    ");
5.      }
6.      //这里对参数值进行声明定义
7.      this.declaredParameters.add(param);
```

而这个 declareParameters 维护的是一个列表：
Java 代码

```
1.  /** List of SqlParameter objects */
2.  private List declaredParameters = new LinkedList();
```

这个列表在以后 compile 的过程中会被使用。
3.然后用户程序需要实现 MappingSqlQuery 的 mapRow 接口，将具体的 ResultSet 数据生成我们需要的对象，这是我们迭代使用的方法。
1，2，3 步实际上为我们定义好了一个迭代的基本单元作为操作模板。
4.在应用程序，我们直接调用 execute()方法得到我们需要的对象列表，列表中的每一个对象的数据来自于执行 SQL 语句得到记录集的每一条记录，事实上执行的 execute 在父类 SqlQuery 中起作用：
Java 代码

```
1.  public List executeByNamedParam(Map paramMap, Map context) throws DataAccessException {
2.      validateNamedParameters(paramMap);
3.      Object[] parameters = NamedParameterUtils.buildValueArray(getSql(), paramMap);
4.      RowMapper rowMapper = newRowMapper(parameters, context);
5.      String sqlToUse = NamedParameterUtils.substituteNamedParameters(getSql(), new MapSqlParameterSo
    urce(paramMap));
6.      //我们又看到了 JdbcTemplate,这里使用 JdbcTemplate 来完成对数据库的查询操作，所以我们说 JdbcTemplate 是基
    本的操作类。
```

```
7.      return getJdbcTemplate().query(newPreparedStatementCreator(sqlToUse, parameters), rowMappe
    r);
8.  }
```

在这里我们可以看到 template 模式的精彩应用和对 JdbcTemplate 的灵活使用。通过使用它，我们免去了手工迭代 ResultSet 并将其中的数据转化为对象列表的重复过程。在这里我们只需要定义 SQL 语句和 SqlParameter - 如果需要的话，往往 SQL 语句就常常能够满足我们的要求了。这是灵活使用 JdbcTemplate 的一个很好的例子。

Spring 还为其他数据库操作提供了许多服务，比如使用 SqlUpdate 插入和更新数据库，使用 UpdatableSqlQuery 更新 ResultSet，生成主键，调用存储过程等。

书中还给出了对 BLOB 数据和 CLOB 数据进行数据库操作的例子：

对 BLOB 数据的操作通过 LobHander 来完成，通过调用 JdbcTemplate 和 RDBMS 都可以进行操作：

在 JdbcTemplate 中，具体的调用可以参考书中的例子 - 是通过以下调用起作用的：

Java 代码

```
1.  public Object execute(String sql, PreparedStatementCallback action) throws DataAccessException {
2.      return execute(new SimplePreparedStatementCreator(sql), action);
3.  }
```

然后通过对实现 PreparedStatementCallback 接口的 AbstractLobCreatingPreparedStatementCallback 的回调函数来完成：

Java 代码

```
1.  public final Object doInPreparedStatement(PreparedStatement ps) throws SQLException, DataAccessExce
    ption {
2.      LobCreator lobCreator = this.lobHandler.getLobCreator();
3.      try {
4.          //这是一个模板方法，具体需要由客户程序实现
5.          setValues(ps, lobCreator);
6.          return new Integer(ps.executeUpdate());
7.      }
8.      finally {
9.          lobCreator.close();
10.     }
11. }
12. //定义的需要客户程序实现的虚函数
13. protected abstract void setValues(PreparedStatement ps, LobCreator lobCreator)
14.         throws SQLException, DataAccessException;
```

而我们注意到 setValues()是一个需要实现的抽象方法，应用程序通过实现 setValues 来定义自己的操作 - 在 setValues 中调用 lobCreator.setBlobAsBinaryStrem()。让我们看看具体的 BLOB 操作在 LobCreator 是怎样完成的，我们一般使用 DefaultLobCreator 作为 BLOB 操作的驱动：

Java 代码

```
1.  public void setBlobAsBinaryStream(
2.          PreparedStatement ps, int paramIndex, InputStream binaryStream, int contentLength)
3.          throws SQLException {
4.      //通过 JDBC 来完成对 BLOB 数据的操作，对 Oracle,Spring 提供了 OracleLobHandler 来支持 BLOB 操作。
5.      ps.setBinaryStream(paramIndex, binaryStream, contentLength);
6.      ........
```

```
7.  }
```

上面提到的是零零碎碎的 Spring JDBC 使用的例子，可以看到使用 Spring JDBC 可以帮助我们完成许多数据库的操作。Spring 对数据库操作最基本的服务是通过 JdbcTeamplate 和他常用的回调函数来实现的，在此之上，又提供了许多 RMDB 的操作来帮助我们更便利的对数据库的数据进行操作 - 注意这里没有引入向 Hibernate 这样的 O/R 方案。对这些 O/R 方案的支持，Spring 由其他包来完成服务。书中还提到关于 execute 和 update 方法之间的区别，update 方法返回的是受影响的记录数目的一个计数，并且如果传入参数的话，使用的是 java.sql.PreparedStatement,而 execute 方法总是使用 java.sql.Statement,不接受参数，而且他不返回受影响记录的计数，更适合于创建和丢弃表的语句，而 update 方法更适合于插入，更新和删除操作，这也是我们在使用时需要注意的。

四、Spring MVC

下面我们对Spring MVC框架代码进行分析,对于webApplicationContext的相关分析可以参见以前的文档,我们这里着重分析Spring Web MVC 框架的实现.我们从分析 DispatcherServlet 入手：

Java 代码

```
1.  //这里是对 DispatcherServlet 的初始化方法，根据名字我们很方面的看到对各个 Spring MVC 主要元素的初始化
2.  protected void initFrameworkServlet() throws ServletException, BeansException {
3.      initMultipartResolver();
4.      initLocaleResolver();
5.      initThemeResolver();
6.      initHandlerMappings();
7.      initHandlerAdapters();
8.      initHandlerExceptionResolvers();
9.      initRequestToViewNameTranslator();
10.     initViewResolvers();
11. }
```

看到注解我们知道，这是 DispatcherSerlvet 的初始化过程，它是在 WebApplicationContext 已经存在的情况下进行的，也就意味着在初始化它的时候，IOC 容器应该已经工作了，这也是我们在 web.xml 中配置 Spring 的时候，需要把 DispatcherServlet 的 load-on-startup 的属性配置为 2 的原因。

对于具体的初始化过程，很容易理解，我们拿 initHandlerMappings（）来看看：

Java 代码

```
1.  private void initHandlerMappings() throws BeansException {
2.      if (this.detectAllHandlerMappings) {
3.           // 这里找到所有在上下文中定义的 HandlerMapping,同时把他们排序
4.           // 因为在同一个上下文中可以有不止一个 handlerMapping,所以我们把他们都载入到一个链里进行维护和管
    理
5.          Map matchingBeans = BeanFactoryUtils.beansOfTypeIncludingAncestors(
6.                  getWebApplicationContext(), HandlerMapping.class, true, false);
7.          if (!matchingBeans.isEmpty()) {
8.              this.handlerMappings = new ArrayList(matchingBeans.values());
9.              // 这里通过 order 属性来对 handlerMapping 来在 list 中排序
10.             Collections.sort(this.handlerMappings, new OrderComparator());
11.         }
12.     }
13.     else {
14.         try {
15.             Object hm = getWebApplicationContext().getBean(HANDLER_MAPPING_BEAN_NAME, HandlerMappin
    g.class);
16.             this.handlerMappings = Collections.singletonList(hm);
17.         }
18.         catch (NoSuchBeanDefinitionException ex) {
19.             // Ignore, we'll add a default HandlerMapping later.
20.         }
21.     }
```

```
22.
23.     //如果在上下文中没有定义的话，那么我们使用默认的 BeanNameUrlHandlerMapping
24.     if (this.handlerMappings == null) {
25.         this.handlerMappings = getDefaultStrategies(HandlerMapping.class);
26.     ........
27.     }
28. }
```

怎样获得上下文环境，可以参见我们前面的对 IOC 容器在 web 环境中加载的分析。DispatcherServlet 把定义了的所有 HandlerMapping 都加载了放在一个 List 里待以后进行使用,这个链的每一个元素都是一个 handlerMapping 的配置，而一般每一个 handlerMapping 可以持有一系列从 URL 请求到 Spring Controller 的映射，比如 SimpleUrl
HandlerMaaping 中就定义了一个 map 来持有这一系列的映射关系。
DisptcherServlet 通过 HandlerMapping 使得 Web 应用程序确定一个执行路径，就像我们在 HanderMapping 中看到的那样，
HandlerMapping 只是一个借口：
Java 代码

```
1.  public interface HandlerMapping {
2.    public static final String PATH_WITHIN_HANDLER_MAPPING_ATTRIBUTE =
3.                        Conventions.getQualifiedAttributeName(HandlerMapping.class, "pathWithinHandlerM
    apping");
4.        //实际上维护一个 HandlerExecutionChain,这是典型的 Command 的模式的使用，这个执行链里面维护 handler 和
    拦截器
5.      HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception;
6.  }
```

他的具体实现只需要实现一个接口方法，而这个接口方法返回的是一个 HandlerExecutionChain,实际上就是一个执行链，就像在 Command 模式描述的那样，这个类很简单，就是一个持有一个 Interceptor 链和一个 Controller：
Java 代码

```
1.  public class HandlerExecutionChain {
2.
3.      private Object handler;
4.
5.      private HandlerInterceptor[] interceptors;
6.
7.      ........
8.  }
```

而这些 Handler 和 Interceptor 需要我们定义 HandlerMapping 的时候配置好，比如对具体的 SimpleURLHandlerMapping,他要做的就是根据 URL 映射的方式注册 Handler 和 Interceptor，自己维护一个放映映射的 handlerMap，当需要匹配 Http 请求的时候需要使用这个表里的信息来得到执行链。这个注册的过程在 IOC 容器初始化 SimpleUrlHandlerMapping 的时候就被完成了，这样以后的解析才可以用到 map 里的映射信息，这里的信息和 bean 文件的信息是等价的，下面是具体的注册过程：
Java 代码

```
1.  protected void registerHandlers(Map urlMap) throws BeansException {
2.      if (urlMap.isEmpty()) {
3.          logger.warn("Neither 'urlMap' nor 'mappings' set on SimpleUrlHandlerMapping");
4.      }
5.      else {
```

```
        //这里迭代在 SimpleUrlHandlerMapping 中定义的所有映射元素
        Iterator it = urlMap.keySet().iterator();
        while (it.hasNext()) {
            //这里取得配置的 url
            String url = (String) it.next();
            //这里根据 url 在 bean 定义中取得对应的 handler
            Object handler = urlMap.get(url);
            // Prepend with slash if not already present.
            if (!url.startsWith("/")) {
                url = "/" + url;
            }
            //这里调用 AbstractHandlerMapping 中的注册过程
            registerHandler(url, handler);
        }
    }
}
```

在 AbstractMappingHandler 中的注册代码：

Java 代码

```
protected void registerHandler(String urlPath, Object handler) throws BeansException, IllegalStateE
xception {
    //试图从 handlerMap 中取 handler,看看是否已经存在同样的 Url 映射关系
    Object mappedHandler = this.handlerMap.get(urlPath);
    if (mappedHandler != null) {
    ........
    }

    //如果是直接用 bean 名做映射那就直接从容器中取 handler
    if (!this.lazyInitHandlers && handler instanceof String) {
        String handlerName = (String) handler;
        if (getApplicationContext().isSingleton(handlerName)) {
            handler = getApplicationContext().getBean(handlerName);
        }
    }
    //或者使用默认的 handler.
    if (urlPath.equals("/*")) {
        setDefaultHandler(handler);
    }
    else {
   //把 url 和 handler 的对应关系放到 handlerMap 中去
        this.handlerMap.put(urlPath, handler);
        ........
    }
}
```

handlerMap 是持有的一个 HashMap,里面就保存了具体的映射信息：

Java 代码

```
1.  private final Map handlerMap = new HashMap();
```

而 SimpleUrlHandlerMapping 对接口 HandlerMapping 的实现是这样的，这个 getHandler 根据在初始化的时候就得到的映射表来生成 DispatcherServlet 需要的执行链

Java 代码

```
1.  public final HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
2.      //这里根据 request 中的参数得到其对应的 handler,具体处理在 AbstractUrlHandlerMapping 中
3.      Object handler = getHandlerInternal(request);
4.      //如果找不到对应的，就使用缺省的 handler
5.      if (handler == null) {
6.          handler = this.defaultHandler;
7.      }
8.      //如果缺省的也没有，那就没办法了
9.      if (handler == null) {
10.         return null;
11.     }
12.     // 如果 handler 不是一个具体的 handler,那我们还要到上下文中取
13.     if (handler instanceof String) {
14.         String handlerName = (String) handler;
15.         handler = getApplicationContext().getBean(handlerName);
16.     }
17.     //生成一个 HandlerExecutionChain,其中放了我们匹配上的 handler 和定义好的拦截器，就像我们在 HandlerExec
    utionChain 中看到的那样，它持有一个 handler 和一个拦截器组。
18.     return new HandlerExecutionChain(handler, this.adaptedInterceptors);
19. }
```

我们看看具体的 handler 查找过程：

Java 代码

```
1.  protected Object getHandlerInternal(HttpServletRequest request) throws Exception {
2.      //这里的 HTTP Request 传进来的参数进行分析，得到具体的路径信息。
3.      String lookupPath = this.urlPathHelper.getLookupPathForRequest(request);
4.      .......//下面是根据请求信息的查找
5.      return lookupHandler(lookupPath, request);
6.  }
7.
8.  protected Object lookupHandler(String urlPath, HttpServletRequest request) {
9.      // 如果能够直接能在 SimpleUrlHandlerMapping 的映射表中找到，那最好。
10.     Object handler = this.handlerMap.get(urlPath);
11.     if (handler == null) {
12.         // 这里使用模式来对 map 中的所有 handler 进行匹配，调用了 Jre 中的 Matcher 类来完成匹配处理。
13.         String bestPathMatch = null;
14.         for (Iterator it = this.handlerMap.keySet().iterator(); it.hasNext();) {
15.             String registeredPath = (String) it.next();
16.             if (this.pathMatcher.match(registeredPath, urlPath) &&
```

```
                            (bestPathMatch == null || bestPathMatch.length() <= registeredPath.length())) {
                //这里根据匹配路径找到最象的一个
                handler = this.handlerMap.get(registeredPath);
                bestPathMatch = registeredPath;
            }
        }

        if (handler != null) {
            exposePathWithinMapping(this.pathMatcher.extractPathWithinPattern(bestPathMatch, urlPath), request);
        }
    }
    else {
        exposePathWithinMapping(urlPath, request);
    }
    //
    return handler;
}
```

我们可以看到，总是在 handlerMap 这个 HashMap 中找，当然如果直接找到最好，如果找不到，就看看是不是能通过 Match Pattern 的模式找，我们一定还记得在配置 HnaderMapping 的时候是可以通过 ANT 语法进行配置的，其中的处理就在这里。
这样可以清楚地看到整个 HandlerMapping 的初始化过程 - 同时，我们也看到了一个具体的 handler 映射是怎样被存储和查找的 - 这里生成一个 ExecutionChain 来储存我们找到的 handler 和在定义 bean 的时候定义的 Interceptors.
让我们回到 DispatcherServlet，初始化完成以后，实际的对 web 请求是在 doService()方法中处理的，我们知道 DispatcherServlet 只是一个普通的 Servlet:
Java 代码

```
protected void doService(HttpServletRequest request, HttpServletResponse response) throws Exception {
    .......
    //这里把属性信息进行保存
    Map attributesSnapshot = null;
    if (WebUtils.isIncludeRequest(request)) {
        logger.debug("Taking snapshot of request attributes before include");
        attributesSnapshot = new HashMap();
        Enumeration attrNames = request.getAttributeNames();
        while (attrNames.hasMoreElements()) {
            String attrName = (String) attrNames.nextElement();
            if (this.cleanupAfterInclude || attrName.startsWith(DispatcherServlet.class.getName())) {
                attributesSnapshot.put(attrName, request.getAttribute(attrName));
            }
        }
    }

    // Make framework objects available to handlers and view objects.
```

```
    request.setAttribute(WEB_APPLICATION_CONTEXT_ATTRIBUTE, getWebApplicationContext());
    request.setAttribute(LOCALE_RESOLVER_ATTRIBUTE, this.localeResolver);
    request.setAttribute(THEME_RESOLVER_ATTRIBUTE, this.themeResolver);
    request.setAttribute(THEME_SOURCE_ATTRIBUTE, getThemeSource());

    try {
         //这里使实际的处理入口
        doDispatch(request, response);
    }
    finally {
        // Restore the original attribute snapshot, in case of an include.
        if (attributesSnapshot != null) {
            restoreAttributesAfterInclude(request, attributesSnapshot);
        }
    }
}
```

我们看到，对于请求的处理实际上是让 doDispatch()来完成的 - 这个方法很长，但是过程很简单明了：

Java 代码

```
protected void doDispatch(final HttpServletRequest request, HttpServletResponse response) throws Exception {
    HttpServletRequest processedRequest = request;
    //这是从 handlerMapping 中得到的执行链
    HandlerExecutionChain mappedHandler = null;
    int interceptorIndex = -1;

    ........
    try {
        //我们熟悉的 ModelAndView 开始出现了。
        ModelAndView mv = null;
        try {
            processedRequest = checkMultipart(request);

            // 这是我们得到 handler 的过程
            mappedHandler = getHandler(processedRequest, false);
            if (mappedHandler == null || mappedHandler.getHandler() == null) {
                noHandlerFound(processedRequest, response);
                return;
            }

            // 这里取出执行链中的 Interceptor 进行前处理
            if (mappedHandler.getInterceptors() != null) {
                for (int i = 0; i < mappedHandler.getInterceptors().length; i++) {
                    HandlerInterceptor interceptor = mappedHandler.getInterceptors()[i];
                    if (!interceptor.preHandle(processedRequest, response, mappedHandler.getHandler())) {
```

```
                        triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, null);
                        return;
                    }
                    interceptorIndex = i;
                }
            }

            //在执行 handler 之前，用 HandlerAdapter 先检查一下 handler 的合法性：是不是按 Spring 的要求编写的。
            HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
            mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

            // 这里取出执行链中的 Interceptor 进行后处理
            if (mappedHandler.getInterceptors() != null) {
                for (int i = mappedHandler.getInterceptors().length - 1; i >= 0; i--) {
                    HandlerInterceptor interceptor = mappedHandler.getInterceptors()[i];
                    interceptor.postHandle(processedRequest, response, mappedHandler.getHandler(), mv);
                }
            }
        }

        ........

        // Did the handler return a view to render?
        //这里对视图生成进行处理
        if (mv != null && !mv.wasCleared()) {
            render(mv, processedRequest, response);
        }
        .......
}
```

我们很清楚的看到和 MVC 框架紧密相关的代码,比如如何得到和 http 请求相对应的执行链，怎样执行执行链和怎样把模型数据展现到视图中去。

先看怎样取得 Command 对象，对我们来说就是 Handler - 下面是 getHandler 的代码：

Java 代码

```
protected HandlerExecutionChain getHandler(HttpServletRequest request, boolean cache) throws Exception {
  //在 ServletContext 取得执行链 - 实际上第一次得到它的时候，我们把它放在 ServletContext 进行了缓存。
  HandlerExecutionChain handler =
            (HandlerExecutionChain) request.getAttribute(HANDLER_EXECUTION_CHAIN_ATTRIBUTE);
    if (handler != null) {
        if (!cache) {
            request.removeAttribute(HANDLER_EXECUTION_CHAIN_ATTRIBUTE);
        }
```

```
            return handler;
        }
        //这里的迭代器迭代的时在 initHandlerMapping 中载入的上下文所有的 HandlerMapping
        Iterator it = this.handlerMappings.iterator();
        while (it.hasNext()) {
            HandlerMapping hm = (HandlerMapping) it.next();
            .......
            //这里是实际取得 handler 的过程,在每个 HandlerMapping 中建立的映射表进行检索得到请求对应的 handler
            handler = hm.getHandler(request);

            //然后把 handler 存到 ServletContext 中去进行缓存
            if (handler != null) {
                if (cache) {
                    request.setAttribute(HANDLER_EXECUTION_CHAIN_ATTRIBUTE, handler);
                }
                return handler;
            }
        }
        return null;
    }
```

如果在 ServletContext 中可以取得 handler 则直接返回，实际上这个 handler 是缓冲了上次处理的结果 - 总要有第一次把这个 handler 放到 ServletContext 中去：
如果在 ServletContext 中找不到 handler,那就通过持有的 handlerMapping 生成一个，我们看到它会迭代当前持有的所有的 handlerMapping,因为可以定义不止一个，他们在定义的时候也可以指定顺序，直到找到第一个，然后返回。先找到一个 handlerMapping，然后通过这个 handlerMapping 返回一个执行链，里面包含了最终的 Handler 和我们定义的一连串的 Interceptor。具体的我们可以参考上面的 SimpleUrlHandlerMapping 的代码分析知道 getHandler 是怎样得到一个 HandlerExecutionChain 的。
得到 HandlerExecutionChain 以后，我们通过 HandlerAdapter 对这个 Handler 的合法性进行判断：
Java 代码

```
    protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {
        Iterator it = this.handlerAdapters.iterator();
        while (it.hasNext()) {
            //同样对持有的所有 adapter 进行匹配
            HandlerAdapter ha = (HandlerAdapter) it.next();
            if (ha.supports(handler)) {
                return ha;
            }
        }
        ........
    }
```

通过判断，我们知道这个 handler 是不是一个 Controller 接口的实现，比如对于具体的 HandlerAdapter - SimpleControllerHandlerAdapter:
Java 代码

```
    public class SimpleControllerHandlerAdapter implements HandlerAdapter {

```

```
    public boolean supports(Object handler) {
        return (handler instanceof Controller);
    }
    .......
}
```

简单的判断一下 handler 是不是实现了 Controller 接口。这也体现了一种对配置文件进行验证的机制。
让我们再回到 DispatcherServlet 看到代码：
Java 代码

```
mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
```

这个就是对 handle 的具体调用！相当于 Command 模式里的 Command.execute();理所当然的返回一个 ModelAndView，下面就是一个对 View 进行处理的过程：
Java 代码

```
if (mv != null && !mv.wasCleared()) {
    render(mv, processedRequest, response);
}
```

调用的是 render 方法：
Java 代码

```
protected void render(ModelAndView mv, HttpServletRequest request, HttpServletResponse response)
        throws Exception {response.setLocale(locale);

     View view = null;
     //这里把默认的视图放到 ModelAndView 中去。
     if (!mv.hasView()) {
         mv.setViewName(getDefaultViewName(request));
     }

     if (mv.isReference()) {
         // 这里对视图名字进行解析
         view = resolveViewName(mv.getViewName(), mv.getModelInternal(), locale, request);
     .......
     }
     else {
         // 有可能在 ModelAndView 里已经直接包含了 View 对象，那我们就直接使用。
         view = mv.getView();
     ........
     }

     //得到具体的 View 对象以后，我们用它来生成视图。
     view.render(mv.getModelInternal(), request, response);
}
```

从整个过程我们看到先在 ModelAndView 中寻找视图的逻辑名，如果找不到那就使用缺省的视图，如果能够找到视图的名字，那就对他

进行解析得到实际的需要使用的视图对象。还有一种可能就是在 ModelAndView 中已经包含了实际的视图对象，这个视图对象是可以直接使用的。

不管怎样，得到一个视图对象以后，通过调用视图对象的 render 来完成数据的显示过程，我们可以看看具体的 JstlView 是怎样实现的，我们在 JstlView 的抽象父类 AbstractView 中找到 render 方法：

Java 代码

```
1.  public void render(Map model, HttpServletRequest request, HttpServletResponse response) throws Exce
    ption {
2.      ......
3.      // 这里把所有的相关信息都收集到一个 Map 里
4.      Map mergedModel = new HashMap(this.staticAttributes.size() + (model != null ? model.size
    () : 0));
5.      mergedModel.putAll(this.staticAttributes);
6.      if (model != null) {
7.          mergedModel.putAll(model);
8.      }
9.
10.     // Expose RequestContext?
11.     if (this.requestContextAttribute != null) {
12.         mergedModel.put(this.requestContextAttribute, createRequestContext(request, mergedMode
    l));
13.     }
14.     //这是实际的展现模型数据到视图的调用。
15.     renderMergedOutputModel(mergedModel, request, response);
16. }
```

注解写的很清楚了，先把所有的数据模型进行整合放到一个 Map - mergedModel 里，然后调用 renderMergedOutputModel();这个 renderMergedOutputModel 是一个模板方法，他的实现在 InternalResourceView 也就是 JstlView 的父类：

Java 代码

```
1.  protected void renderMergedOutputModel(
2.          Map model, HttpServletRequest request, HttpServletResponse response) throws Exception {
3.
4.      // Expose the model object as request attributes.
5.      exposeModelAsRequestAttributes(model, request);
6.
7.      // Expose helpers as request attributes, if any.
8.      exposeHelpers(request);
9.
10.     // 这里得到 InternalResource 定义的内部资源路径。
11.     String dispatcherPath = prepareForRendering(request, response);
12.
13.     //这里把请求转发到前面得到的内部资源路径中去。
14.     RequestDispatcher rd = request.getRequestDispatcher(dispatcherPath);
15.     if (rd == null) {
16.         throw new ServletException(
17.                 "Could not get RequestDispatcher for [" + getUrl() + "]: check that this file exist
    s within your WAR");
```

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

首先对模型数据进行处理，exposeModelAsRequestAttributes 是在 AbstractView 中实现的，这个方法把 ModelAndView 中的模型数据和其他 request 数据统统放到 ServletContext 当中去，这样整个模型数据就通过 ServletContext 暴露并得到共享使用了：

Java 代码

```
protected void exposeModelAsRequestAttributes(Map model, HttpServletRequest request) throws Exception {
        Iterator it = model.entrySet().iterator();
        while (it.hasNext()) {
            Map.Entry entry = (Map.Entry) it.next();
            ..........
            String modelName = (String) entry.getKey();
            Object modelValue = entry.getValue();
            if (modelValue != null) {
                request.setAttribute(modelName, modelValue);
            ...........
            }
            else {
                request.removeAttribute(modelName);
                .......
            }
        }
    }
```

让我们回到数据处理部分的 exposeHelper();这是一个模板方法，其实现在 JstlView 中实现：

Java 代码

```
public class JstlView extends InternalResourceView {

    private MessageSource jstlAwareMessageSource;


    protected void initApplicationContext() {
        super.initApplicationContext();
        this.jstlAwareMessageSource =
                JstlUtils.getJstlAwareMessageSource(getServletContext(), getApplicationContext());
    }

    protected void exposeHelpers(HttpServletRequest request) throws Exception {
        JstlUtils.exposeLocalizationContext(request, this.jstlAwareMessageSource);
    }

}
```

在 JstlUtils 中包含了对于其他而言 jstl 特殊的数据处理和设置。

过程是不是很长？我们现在在哪里了？呵呵，我们刚刚完成的事 MVC 中 View 的 render，对于 InternalResourceView 的 render 过程比较简单只是完成一个资源的重定向处理。需要做的就是得到实际 view 的 internalResource 路径，然后转发到那个资源中去。怎样得到资源的路径呢通过调用：

Java 代码

```
protected String prepareForRendering(HttpServletRequest request, HttpServletResponse response)
        throws Exception {

    return getUrl();
```

那这个 url 在哪里生成呢？我们在 View 相关的代码中没有找到，实际上，他在 ViewRosolve 的时候就生成了，在 UrlBasedViewResolver 中：

Java 代码

```
protected AbstractUrlBasedView buildView(String viewName) throws Exception {
    AbstractUrlBasedView view = (AbstractUrlBasedView) BeanUtils.instantiateClass(getViewClass
());
    view.setUrl(getPrefix() + viewName + getSuffix());
    String contentType = getContentType();
    if (contentType != null) {
        view.setContentType(contentType);
    }
    view.setRequestContextAttribute(getRequestContextAttribute());
    view.setAttributesMap(getAttributesMap());
    return view;
}
```

这里是生成 View 的地方，自然也把生成的 url 和其他一些和 view 相关的属性也配置好了。

那这个 ViewResolve 是什么时候被调用的呢？哈哈，我们这样又要回到 DispatcherServlet 中去看看究竟，在 DispatcherServlet 中：

Java 代码

```
 protected void render(ModelAndView mv, HttpServletRequest request, HttpServletResponse response)
         throws Exception {

     ........
     View view = null;

     // 这里设置视图名为默认的名字
     if (!mv.hasView()) {
         mv.setViewName(getDefaultViewName(request));
     }

     if (mv.isReference()) {
         //这里对视图名进行解析，在解析的过程中根据需要生成实际需要的视图对象。
         view = resolveViewName(mv.getViewName(), mv.getModelInternal(), locale, request);
        ..........
     }
    ......
}
```

下面是对视图名进行解析的具体过程：

Java 代码

```
protected View resolveViewName(String viewName, Map model, Locale locale, HttpServletRequest reques
t)
            throws Exception {
         //我们有可能不止一个视图解析器
        for (Iterator it = this.viewResolvers.iterator(); it.hasNext();) {
            ViewResolver viewResolver = (ViewResolver) it.next();
            //这里是视图解析器进行解析并生成视图的过程。
            View view = viewResolver.resolveViewName(viewName, locale);
            if (view != null) {
                return view;
            }
        }
        return null;
    }
```

这里调用具体的 ViewResolver 对视图的名字进行解析 - 除了单纯的解析之外，它还根据我们的要求生成了我们实际需要的视图对象。具体的 viewResolver 在 bean 定义文件中进行定义同时在 initViewResolver()方法中被初始化到 viewResolver 变量中，我们看看具体的 InternalResourceViewResolver 是怎样对视图名进行处理的并生成 V 视图对象的：对 resolveViewName 的调用模板在 AbstractCachingViewResolver 中，

Java 代码

```
public View resolveViewName(String viewName, Locale locale) throws Exception {
    //如果没有打开缓存设置，那创建需要的视图
    if (!isCache()) {
        logger.warn("View caching is SWITCHED OFF -- DEVELOPMENT SETTING ONLY: This can severely im
pair performance");
        return createView(viewName, locale);
    }
    else {
        Object cacheKey = getCacheKey(viewName, locale);
        // No synchronization, as we can live with occasional double caching.
        synchronized (this.viewCache) {
            //这里查找缓存里的视图对象
            View view = (View) this.viewCache.get(cacheKey);
            if (view == null) {
                //如果在缓存中没有找到，创建一个并把创建的放到缓存中去
                view = createView(viewName, locale);
                this.viewCache.put(cacheKey, view);
            ........
            }
            return view;
        }
    }
}
```

关于这些 createView(),loadView(),buildView()的关系，我们看看 Eclipse 里的 call hiearchy
然后我们回到 view.render 中完成数据的最终对 httpResponse 的写入，比如在 AbstractExcelView 中的实现:
Java 代码

```
protected final void renderMergedOutputModel(
        Map model, HttpServletRequest request, HttpServletResponse response) throws Exception {
    .........
    // response.setContentLength(workbook.getBytes().length);
    response.setContentType(getContentType());
    ServletOutputStream out = response.getOutputStream();
    workbook.write(out);
    out.flush();
}
```

这样就和我们前面的分析一致起来了:DispatcherServlet 在解析视图名的时候就根据要求生成了视图对象,包括在 InternalResourceView 中需要使用的 url 和其他各种和 HTTP response 相关的属性都会写保持在生成的视图对象中，然后就直接调用视图对象的 render 来完成数据的展示。
这就是整个 Spring Web MVC 框架的大致流程，整个 MVC 流程由 DispatcherServlet 来控制。MVC 的关键过程包括:
配置到 handler 的映射关系和怎样根据请求参数得到对应的 handler,在 Spring 中，这是由 handlerMapping 通过执行链来完成的，而具体的映射关系我们在 bean 定义文件中定义并在 HandlerMapping 载入上下文的时候就被配置好了。然后 DispatcherServlet 调用 HandlerMapping 来得到对应的执行链，最后通过视图来展现模型数据，但我们要注意的是视图对象是在解析视图名的时候生成配置好的。这些作为核心类的 HanderMapping,ViewResolver,View,Handler 的紧密协作实现了 MVC 的功能。

声明：JavaEye 文章版权属于作者，受法律保护。没有作者书面许可不得转载。

五、Spring AOP 获取 Proxy

下面我们来看看 Spring 的 AOP 的一些相关代码是怎么得到 Proxy 的，让我们我们先看看 AOP 和 Spring AOP 的一些基本概念：

Advice：

通知，制定在连接点做什么，在 Sping 中，他主要描述 Spring 围绕方法调用注入的额外的行为，Spring 提供的通知类型有：

before advice,AfterReturningAdvice,ThrowAdvice,MethodBeforeAdvice，这些都是 Spring AOP 定义的接口类，具体的动作实现需要用户程序来完成。

Pointcut:

切点，其决定一个 advice 应该应用于哪个连接点，也就是需要插入额外处理的地方的集合，例如，被某个 advice 作为目标的一组方法。Spring pointcut 通常意味着标示方法，可以选择一组方法调用作为 pointcut,Spring 提供了具体的切点来给用户使用，比如正则表达式切点 JdkRegexpMethodPointcut 通过正则表达式对方法名进行匹配，其通过使用 AbstractJdkRegexpMethodPointcut 中的对 MethodMatcher 接口的实现来完成 pointcut 功能：

Java 代码

```
1.  public final boolean matches(Method method, Class targetClass) {
2.      //这里通过反射得到方法的全名
3.      String patt = method.getDeclaringClass().getName() + "." + method.getName();
4.      for (int i = 0; i < this.patterns.length; i++) {
5.          // 这里是判断是否和方法名是否匹配的代码
6.          boolean matched = matches(patt, i);
7.          if (matched) {
8.              for (int j = 0; j < this.excludedPatterns.length; j++) {
9.                  boolean excluded = matchesExclusion(patt, j);
10.                 if(excluded) {
11.                     return false;
12.                 }
13.             }
14.             return true;
15.         }
16.     }
17.     return false;
18. }
```

在 JDKRegexpMethodPointcut 中通过 JDK 中的正则表达式匹配来完成 pointcut 的最终确定：

Java 代码

```
1.  protected boolean matches(String pattern, int patternIndex) {
2.      Matcher matcher = this.compiledPatterns[patternIndex].matcher(pattern);
3.      return matcher.matches();
4.  }
```

Advisor：

当我们完成额外的动作设计（advice)和额外动作插入点的设计（pointcut)以后,我们需要一个对象把他们结合起来，这就是通知器 - advisor,定义应该在哪里应用哪个通知。Advisor 的实现有：DefaultPointcutAdvisor 他有两个属性 advice 和 pointcut 来让我们配置 advice 和 pointcut。

接着我们就可以通过 ProxyFactoryBean 来配置我们的代理对象和方面行为，在 ProxyFactoryBean 中有 interceptorNames 来配置已经定义好的通知器-advisor,虽然这里的名字叫做 interceptNames,但实际上是供我们配置 advisor 的地方，具体的代理实现通过 JDK 的

Proxy 或者 CGLIB 来完成。因为 ProxyFactoryBean 是一个 FactoryBean,在 ProxyFactoryBean 中我们通过 getObject()可以直接得到代理对象：

Java 代码

```
1.  public Object getObject() throws BeansException {
2.      //这里初始化通知器链
3.      initializeAdvisorChain();
4.      if (isSingleton()) {
5.      //根据定义需要生成单件的 Proxy
6.          return getSingletonInstance();
7.      }
8.      else {
9.      .......
10.         //这里根据定义需要生成 Prototype 类型的 Proxy
11.         return newPrototypeInstance();
12.     }
13. }
```

我们看看怎样生成单件的代理对象：

Java 代码

```
1.  private synchronized Object getSingletonInstance() {
2.      if (this.singletonInstance == null) {
3.          this.targetSource = freshTargetSource();
4.          if (this.autodetectInterfaces && getProxiedInterfaces().length == 0 && !isProxyTargetClass
    ()) {
5.              // 这里设置代理对象的接口
6.              setInterfaces(ClassUtils.getAllInterfacesForClass(this.targetSource.getTargetClass
    ()));
7.          }
8.          // Eagerly initialize the shared singleton instance.
9.          super.setFrozen(this.freezeProxy);
10.         // 注意这里的方法会使用 ProxyFactory 来生成我们需要的 Proxy
11.         this.singletonInstance = getProxy(createAopProxy());
12.         // We must listen to superclass advice change events to recache the singleton
13.         // instance if necessary.
14.         addListener(this);
15.     }
16.     return this.singletonInstance;
17. }
18.
19. //使用 createAopProxy 放回的 AopProxy 来得到代理对象。
20. protected Object getProxy(AopProxy aopProxy) {
21.     return aopProxy.getProxy(this.beanClassLoader);
22. }
```

ProxyFactoryBean 的父类是 AdvisedSupport，Spring 使用 AopProxy 接口把 AOP 代理的实现与框架的其他部分分离开来；在

AdvisedSupport 中通过这样的方式来得到 AopProxy,当然这里需要得到 AopProxyFactory 的帮助 - 下面我们看到 Spring 为我们提供的实现，来帮助我们方便的从 JDK 或者 cglib 中得到我们想要的代理对象：

Java 代码

```
protected synchronized AopProxy createAopProxy() {
    if (!this.isActive) {
        activate();
    }
    return getAopProxyFactory().createAopProxy(this);
}
```

而在 ProxyConfig 中对使用的 AopProxyFactory 做了定义：

Java 代码

```
//这个 DefaultAopProxyFactory 是 Spring 用来生成 AopProxy 的地方，
//当然了它包含 JDK 和 Cglib 两种实现方式。
private transient AopProxyFactory aopProxyFactory = new DefaultAopProxyFactory();
```

其中在 DefaultAopProxyFactory 中是这样生成 AopProxy 的：

Java 代码

```
public AopProxy createAopProxy(AdvisedSupport advisedSupport) throws AopConfigException {
    //首先考虑使用 cglib 来实现代理对象，当然如果同时目标对象不是接口的实现类的话
    if (advisedSupport.isOptimize() || advisedSupport.isProxyTargetClass() ||
        advisedSupport.getProxiedInterfaces().length == 0) {
        //这里判断如果不存在 cglib 库，直接抛出异常。
        if (!cglibAvailable) {
            throw new AopConfigException(
                    "Cannot proxy target class because CGLIB2 is not available. " +
                    "Add CGLIB to the class path or specify proxy interfaces.");
        }
        // 这里使用 Cglib 来生成 Proxy，如果 target 不是接口的实现的话，返回 cglib 类型的 AopProxy
        return CglibProxyFactory.createCglibProxy(advisedSupport);
    }
    else {
        // 这里使用 JDK 来生成 Proxy,返回 JDK 类型的 AopProxy
        return new JdkDynamicAopProxy(advisedSupport);
    }
}
```

于是我们就可以看到其中的代理对象可以由 JDK 或者 Cglib 来生成，我们看到 JdkDynamicAopProxy 类和 Cglib2AopProxy 都实现的是 AopProxy 的接口，在 JdkDynamicAopProxy 实现中我们可以看到 Proxy 是怎样生成的：

Java 代码

```
public Object getProxy(ClassLoader classLoader) {
    if (logger.isDebugEnabled()) {
        Class targetClass = this.advised.getTargetSource().getTargetClass();
        logger.debug("Creating JDK dynamic proxy" +
                (targetClass != null ? " for [" + targetClass.getName() + "]" : ""));
    }
```

```
7.      Class[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised);
8.      findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
9.      //这里我们调用 JDK Proxy 来生成需要的 Proxy 实例
10.     return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
11. }
```

这样用 Proxy 包装 target 之后，通过 ProxyFactoryBean 得到对其方法的调用就被 Proxy 拦截了， ProxyFactoryBean 的 getObject()方法得到的实际上是一个 Proxy 了，我们的 target 对象已经被封装了。对 ProxyFactoryBean 这个工厂 bean 而言，其生产出来的对象是封装了目标对象的代理对象。

声明：JavaEye 文章版权属于作者，受法律保护。没有作者书面许可不得转载。

六、Spring 声明式事务处理

我们看看 Spring 中的事务处理的代码，使用 Spring 管理事务有声明式和编程式两种方式，声明式事务处理通过 AOP 的实现把事物管理代码作为方面封装来横向插入到业务代码中，使得事务管理代码和业务代码解藕。在这种方式我们结合 IoC 容器和 Spirng 已有的 FactoryBean 来对事务管理进行属性配置，比如传播行为，隔离级别等。其中最简单的方式就是通过配置 TransactionProxyFactoryBean 来实现声明式事物；
在整个源代码分析中，我们可以大致可以看到 Spring 实现声明式事物管理有这么几个部分：

* 对在上下文中配置的属性的处理，这里涉及的类是 TransactionAttributeSourceAdvisor，这是一个通知器，用它来对属性值进行处理，属性信息放在 TransactionAttribute 中来使用,而这些属性的处理往往是和对切入点的处理是结合起来的。对属性的处理放在类 TransactionAttributeSource 中完成。
* 创建事物的过程，这个过程是委托给具体的事物管理器来创建的，但 Spring 通过 TransactionStatus 来传递相关的信息。
* 对事物的处理通过对相关信息的判断来委托给具体的事物管理器完成。

我们下面看看具体的实现，在 TransactionFactoryBean 中：
Java 代码

```
1.  public class TransactionProxyFactoryBean extends AbstractSingletonProxyFactoryBean
2.          implements FactoryBean, BeanFactoryAware {
3.  //这里是 Spring 事务处理而使用的 AOP 拦截器，中间封装了 Spring 对事务处理的代码来支持声明式事务处理的实现
4.      private final TransactionInterceptor transactionInterceptor = new TransactionInterceptor();
5.
6.      private Pointcut pointcut;
7.
8.  //这里 Spring 把 TransactionManager 注入到 TransactionInterceptor 中去
9.      public void setTransactionManager(PlatformTransactionManager transactionManager) {
10.         this.transactionInterceptor.setTransactionManager(transactionManager);
11.     }
12.
13. //这里把在 bean 配置文件中读到的事务管理的属性信息注入到 TransactionInterceptor 中去
14.     public void setTransactionAttributes(Properties transactionAttributes) {
15.         this.transactionInterceptor.setTransactionAttributes(transactionAttributes);
16.     }
17.
18.     .........中间省略了其他一些方法.......
19.
20.     //这里创建 Spring AOP 对事务处理的 Advisor
21.     protected Object createMainInterceptor() {
22.         this.transactionInterceptor.afterPropertiesSet();
23.         if (this.pointcut != null) {
24.             //这里使用默认的通知器
25.             return new DefaultPointcutAdvisor(this.pointcut, this.transactionInterceptor);
26.         }
27.         else {
28.             // 使用上面定义好的 TransactionInterceptor 作为拦截器，同时使用 TransactionAttributeSourceAd
    visor
```

```
            return new TransactionAttributeSourceAdvisor(this.transactionInterceptor);
        }
    }
}
```

那什么时候 Spring 的 TransactionInterceptor 被注入到 Spring AOP 中成为 Advisor 中的一部分呢？我们看到在 TransactionProxyFactoryBean 中，这个方法在 IOC 初始化 bean 的时候被执行：

Java 代码

```
public void afterPropertiesSet() {
    .......
    //TransactionProxyFactoryBean 实际上使用 ProxyFactory 完成 AOP 的基本功能。
    ProxyFactory proxyFactory = new ProxyFactory();

    if (this.preInterceptors != null) {
        for (int i = 0; i < this.preInterceptors.length; i++) {
            proxyFactory.addAdvisor(this.advisorAdapterRegistry.wrap(this.preInterceptors[i]));
        }
    }

    //这里是 Spring 加入通知器的地方
    //有两种通知器可以被加入 DefaultPointcutAdvisor 或者 TransactionAttributeSourceAdvisor
    //这里把 Spring 处理声明式事务处理的 AOP 代码都放到 ProxyFactory 中去，怎样加入 advisor 我们可以参考 ProxyFactory 的父类 AdvisedSupport()
    //由它来维护一个 advice 的链表，通过这个链表的增删改来抽象我们对整个通知器配置的增删改操作。
    proxyFactory.addAdvisor(this.advisorAdapterRegistry.wrap(createMainInterceptor()));

    if (this.postInterceptors != null) {
        for (int i = 0; i < this.postInterceptors.length; i++) {
            proxyFactory.addAdvisor(this.advisorAdapterRegistry.wrap(this.postInterceptors[i]));
        }
    }

    proxyFactory.copyFrom(this);

    //这里创建 AOP 的目标源
    TargetSource targetSource = createTargetSource(this.target);
    proxyFactory.setTargetSource(targetSource);

    if (this.proxyInterfaces != null) {
        proxyFactory.setInterfaces(this.proxyInterfaces);
    }
    else if (!isProxyTargetClass()) {
        proxyFactory.setInterfaces(ClassUtils.getAllInterfacesForClass(targetSource.getTargetClass()));
    }

```

```
37.     this.proxy = getProxy(proxyFactory);
38. }
```

Spring 已经定义了一个 transctionInterceptor 作为拦截器或者 AOP advice 的实现，在 IOC 容器中定义的其他属性比如 transactionManager 和事务管理的属性都会传到已经定义好的 TransactionInterceptor 那里去进行处理。以上反映了基本的 Spring AOP 的定义过程，其中 pointcut 和 advice 都已经定义好，同时也通过通知器配置到 ProxyFactory 中去了。
下面让我们回到 TransactionProxyFactoryBean 中看看 TransactionAttributeSourceAdvisor 是怎样定义的，这样我们可以理解具体的属性是怎样起作用，这里我们分析一下类 TransactionAttributeSourceAdvisor:
Java 代码

```
1.  public class TransactionAttributeSourceAdvisor extends AbstractPointcutAdvisor {
2.      //和其他 Advisor 一样，同样需要定义 AOP 中的用到的 Interceptor 和 Pointcut
3.      //Interceptor 使用传进来的 TransactionInterceptor
4.      //而对于 pointcut,这里定义了一个内部类，参见下面的代码
5.      private TransactionInterceptor transactionInterceptor;
6.
7.      private final TransactionAttributeSourcePointcut pointcut = new TransactionAttributeSourcePoint
    cut();
8.
9.      .........
10.     //定义的 PointCut 内部类
11.         private class TransactionAttributeSourcePointcut extends StaticMethodMatcherPointcut implem
    ents Serializable {
12.         .......
13.       //方法匹配的实现，使用了 TransactionAttributeSource 类
14.         public boolean matches(Method method, Class targetClass) {
15.             TransactionAttributeSource tas = getTransactionAttributeSource();
16.             //这里使用 TransactionAttributeSource 来对配置属性进行处理
17.             return (tas != null && tas.getTransactionAttribute(method, targetClass) != null);
18.         }
19.     ........省略了 equal,hashcode,tostring 的代码
20.     }
```

这里我们看看属性值是怎样被读入的：AbstractFallbackTransactionAttributeSource 负责具体的属性读入任务，我们可以有两种读入方式，比如 annotation 和直接配置.我们下面看看直接配置的读入方式，在 Spring 中同时对读入的属性值进行了缓存处理，这是一个 decorator 模式：
Java 代码

```
1.  public final TransactionAttribute getTransactionAttribute(Method method, Class targetClass) {
2.      //这里先查一下缓存里有没有事务管理的属性配置，如果有从缓存中取得 TransactionAttribute
3.      Object cacheKey = getCacheKey(method, targetClass);
4.      Object cached = this.cache.get(cacheKey);
5.      if (cached != null) {
6.          if (cached == NULL_TRANSACTION_ATTRIBUTE) {
7.              return null;
8.          }
9.          else {
10.             return (TransactionAttribute) cached;
```

```
        }
    }
    else {
        // 这里通过对方法和目标对象的信息来计算事务缓存属性
        TransactionAttribute txAtt = computeTransactionAttribute(method, targetClass);
        //把得到的事务缓存属性存到缓存中，下次可以直接从缓存中取得。
        if (txAtt == null) {
            this.cache.put(cacheKey, NULL_TRANSACTION_ATTRIBUTE);
        }
        else {
            ..........
            this.cache.put(cacheKey, txAtt);
        }
        return txAtt;
    }
}
```

别急，基本的处理在 computeTransactionAttribute（）中：

Java 代码

```
private TransactionAttribute computeTransactionAttribute(Method method, Class targetClass) {
    //这里检测是不是 public 方法
    if(allowPublicMethodsOnly() && !Modifier.isPublic(method.getModifiers())) {
        return null;
    }

    Method specificMethod = AopUtils.getMostSpecificMethod(method, targetClass);

    // First try is the method in the target class.
    TransactionAttribute txAtt = findTransactionAttribute(findAllAttributes(specificMethod));
    if (txAtt != null) {
        return txAtt;
    }

    // Second try is the transaction attribute on the target class.
    txAtt = findTransactionAttribute(findAllAttributes(specificMethod.getDeclaringClass()));
    if (txAtt != null) {
        return txAtt;
    }

    if (specificMethod != method) {
        // Fallback is to look at the original method.
        txAtt = findTransactionAttribute(findAllAttributes(method));
        if (txAtt != null) {
            return txAtt;
        }
        // Last fallback is the class of the original method.
```

```
28.         return findTransactionAttribute(findAllAttributes(method.getDeclaringClass()));
29.     }
30.     return null;
31. }
```

经过一系列的尝试我们可以通过 findTransactionAttribute（）通过调用 findAllAttribute()得到 TransactionAttribute 的对象，如果返回的是 null,这说明该方法不是我们需要事务处理的方法。

在完成把需要的通知器加到 ProxyFactory 中去的基础上，我们看看具体的看事务处理代码怎样起作用，在 TransactionInterceptor 中：

Java 代码

```
1.  public Object invoke(final MethodInvocation invocation) throws Throwable {
2.      //这里得到目标对象
3.      Class targetClass = (invocation.getThis() != null ? invocation.getThis().getClass() : null);
4.
5.      //这里同样的通过判断是否能够得到 TransactionAttribute 来决定是否对当前方法进行事务处理，有可能该属性已经被缓存，
6.      //具体可以参考上面对 getTransactionAttribute 的分析，同样是通过 TransactionAttributeSource
7.      final TransactionAttribute txAttr =
8.              getTransactionAttributeSource().getTransactionAttribute(invocation.getMethod(), targetClass);
9.      final String joinpointIdentification = methodIdentification(invocation.getMethod());
10.
11.     //这里判断我们使用了什么 TransactionManager
12.     if (txAttr == null || !(getTransactionManager() instanceof CallbackPreferringPlatformTransactionManager)) {
13.         // 这里创建事务，同时把创建事务过程中得到的信息放到 TransactionInfo 中去
14.         TransactionInfo txInfo = createTransactionIfNecessary(txAttr, joinpointIdentification);
15.         Object retVal = null;
16.         try {
17.               retVal = invocation.proceed();
18.         }
19.         catch (Throwable ex) {
20.             // target invocation exception
21.             completeTransactionAfterThrowing(txInfo, ex);
22.             throw ex;
23.         }
24.         finally {
25.             cleanupTransactionInfo(txInfo);
26.         }
27.         commitTransactionAfterReturning(txInfo);
28.         return retVal;
29.     }
30.
31.     else {
32.         // 使用的是 Spring 定义的 PlatformTransactionManager 同时实现了回调接口,我们通过其回调函数完成事务处理，就像我们使用编程式事务处理一样。
33.         try {
```

```
34.             Object result = ((CallbackPreferringPlatformTransactionManager) getTransactionManager
    ()).execute(txAttr,
35.                     new TransactionCallback() {
36.                         public Object doInTransaction(TransactionStatus status) {
37.                             //同样的需要一个 TransactonInfo
38.                             TransactionInfo txInfo = prepareTransactionInfo(txAttr, joinpointIdenti
    fication, status);
39.                             try {
40.                                 return invocation.proceed();
41.                             }
42.                          .....这里省去了异常处理和事务信息的清理代码
43.                     });
44.          ...........
45.     }
46. }
```

这里面涉及到事务的创建，我们可以在 TransactionAspectSupport 实现的事务管理代码：

Java 代码

```
1.  protected TransactionInfo createTransactionIfNecessary(
2.          TransactionAttribute txAttr, final String joinpointIdentification) {
3.
4.      // If no name specified, apply method identification as transaction name.
5.      if (txAttr != null && txAttr.getName() == null) {
6.          txAttr = new DelegatingTransactionAttribute(txAttr) {
7.              public String getName() {
8.                  return joinpointIdentification;
9.              }
10.         };
11.     }
12.
13.     TransactionStatus status = null;
14.     if (txAttr != null) {
15.     //这里使用了我们定义好的事务配置信息,有事务管理器来创建事务，同时返回 TransactionInfo
16.         status = getTransactionManager().getTransaction(txAttr);
17.     }
18.     return prepareTransactionInfo(txAttr, joinpointIdentification, status);
19. }
```

首先通过 TransactionManager 得到需要的事务，事务的创建根据我们定义的事务配置决定，在 AbstractTransactionManager 中给出一个标准的创建过程，当然创建什么样的事务还是需要具体的 PlatformTransactionManager 来决定，但这里给出了创建事务的模板：

Java 代码

```
1.  public final TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionE
    xception {
2.      Object transaction = doGetTransaction();
3.      ......
4.
```

```
    if (definition == null) {
        //如果事务信息没有被配置，我们使用 Spring 默认的配置方式
        definition = new DefaultTransactionDefinition();
    }

    if (isExistingTransaction(transaction)) {
        // Existing transaction found -> check propagation behavior to find out how to behave.
        return handleExistingTransaction(definition, transaction, debugEnabled);
    }

    // Check definition settings for new transaction.
    //下面就是使用配置信息来创建我们需要的事务;比如传播属性和同步属性等
    //最后把创建过程中的信息收集起来放到 TransactionStatus 中返回;
    if (definition.getTimeout() < TransactionDefinition.TIMEOUT_DEFAULT) {
        throw new InvalidTimeoutException("Invalid transaction timeout", definition.getTimeout
());
    }

    // No existing transaction found -> check propagation behavior to find out how to behave.
    if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_MANDATORY) {
        throw new IllegalTransactionStateException(
                "Transaction propagation 'mandatory' but no existing transaction found");
    }
    else if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_REQUIRED ||
            definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_REQUIRES_NEW |
|
        definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NESTED) {
        //这里是事务管理器创建事务的地方，并将创建过程中得到的信息放到 TransactionStatus 中去，包括创建出来
的事务
        doBegin(transaction, definition);
        boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
        return newTransactionStatus(definition, transaction, true, newSynchronization, debugEnable
d, null);
    }
    else {
        boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION_ALWAYS);
        return newTransactionStatus(definition, null, false, newSynchronization, debugEnabled, nul
l);
    }
}
```

接着通过调用 prepareTransactionInfo 完成事务创建的准备，创建过程中得到的信息存储在 TransactionInfo 对象中进行传递同时把信息和当前线程绑定；

Java 代码

```
protected TransactionInfo prepareTransactionInfo(
        TransactionAttribute txAttr, String joinpointIdentification, TransactionStatus status) {
```

```
3.
4.      TransactionInfo txInfo = new TransactionInfo(txAttr, joinpointIdentification);
5.      if (txAttr != null) {
6.      .....
7.          // 同样的需要把在 getTransaction 中得到的 TransactionStatus 放到 TransactionInfo 中来。
8.          txInfo.newTransactionStatus(status);
9.      }
10.     else {
11.     .......
12.    }
13.
14.     // 绑定事务创建信息到当前线程
15.     txInfo.bindToThread();
16.     return txInfo;
17. }
```

将创建事务的信息返回，然后看到其他的事务管理代码：

Java 代码

```
1.  protected void commitTransactionAfterReturning(TransactionInfo txInfo) {
2.      if (txInfo != null && txInfo.hasTransaction()) {
3.          if (logger.isDebugEnabled()) {
4.              logger.debug("Invoking commit for transaction on " + txInfo.getJoinpointIdentification
    ());
5.          }
6.          this.transactionManager.commit(txInfo.getTransactionStatus());
7.      }
8.  }
```

通过 transactionManager 对事务进行处理，包括异常抛出和正常的提交事务，具体的事务管理器由用户程序设定。

Java 代码

```
1.  protected void completeTransactionAfterThrowing(TransactionInfo txInfo, Throwable ex) {
2.      if (txInfo != null && txInfo.hasTransaction()) {
3.          if (txInfo.transactionAttribute.rollbackOn(ex)) {
4.              ......
5.              try {
6.                  this.transactionManager.rollback(txInfo.getTransactionStatus());
7.              }
8.              ..........
9.     }
10.         else {
11.             .........
12.             try {
13.                 this.transactionManager.commit(txInfo.getTransactionStatus());
14.             }
15.     ...........
16. }
```

```
17.
18. protected void commitTransactionAfterReturning(TransactionInfo txInfo) {
19.     if (txInfo != null && txInfo.hasTransaction()) {
20.         ......
21.         this.transactionManager.commit(txInfo.getTransactionStatus());
22.     }
23. }
```

Spring 通过以上代码对 transactionManager 进行事务处理的过程进行了 AOP 包装，到这里我们看到为了方便客户实现声明式的事务处理，Spring 还是做了许多工作的。如果说使用编程式事务处理，过程其实比较清楚，我们可以参考书中的例子：

Java 代码

```
1.  TransactionDefinition td = new DefaultTransactionDefinition();
2.  TransactionStatus status = transactionManager.getTransaction(td);
3.  try{
4.        ......//这里是我们的业务方法
5.  }catch (ApplicationException e) {
6.     transactionManager.rollback(status);
7.     throw e
8.  }
9.  transactionManager.commit(status);
10.  ........
```

我们看到这里选取了默认的事务配置 DefaultTransactionDefinition，同时在创建事物的过程中得到 TransactionStatus,然后通过直接调用事务管理器的相关方法就能完成事务处理。

声明式事务处理也同样实现了类似的过程，只是因为采用了声明的方法，需要增加对属性的读取处理，并且需要把整个过程整合到 Spring AOP 框架中和 IoC 容器中去的过程。

下面我们选取一个具体的 transactionManager - DataSourceTransactionManager 来看看其中事务处理的实现：

同样的通过使用 AbstractPlatformTransactionManager 使用模板方法，这些都体现了对具体平台相关的事务管理器操作的封装，比如 commit：

Java 代码

```
1.  public final void commit(TransactionStatus status) throws TransactionException {
2.      ......
3.      DefaultTransactionStatus defStatus = (DefaultTransactionStatus) status;
4.      if (defStatus.isLocalRollbackOnly()) {
5.          ......
6.          processRollback(defStatus);
7.          return;
8.      }
9.           .......
10.         processRollback(defStatus);
11.     ......
12.     }
13.
14.     processCommit(defStatus);
15. }
```

通过对 TransactionStatus 的具体状态的判断，来决定具体的事务处理：

Java 代码

```
1.  private void processCommit(DefaultTransactionStatus status) throws TransactionException {
2.      try {
3.          boolean beforeCompletionInvoked = false;
4.          try {
5.              triggerBeforeCommit(status);
6.              triggerBeforeCompletion(status);
7.              beforeCompletionInvoked = true;
8.              boolean globalRollbackOnly = false;
9.              if (status.isNewTransaction() || isFailEarlyOnGlobalRollbackOnly()) {
10.                 globalRollbackOnly = status.isGlobalRollbackOnly();
11.             }
12.             if (status.hasSavepoint()) {
13.             ........
14.                status.releaseHeldSavepoint();
15.             }
16.             else if (status.isNewTransaction()) {
17.             ......
18.                 doCommit(status);
19.             }
20.         .........
21. }
```

这些模板方法的实现由具体的 transactionManager 来实现，比如在 DataSourceTransactionManager:

Java 代码

```
1.  protected void doCommit(DefaultTransactionStatus status) {
2.      //这里得到存在 TransactionInfo 中已经创建好的事务
3.      DataSourceTransactionObject txObject = (DataSourceTransactionObject) status.getTransaction();
4.
5.      //这里得到和事务绑定的数据库连接
6.      Connection con = txObject.getConnectionHolder().getConnection();
7.      ........
8.      try {
9.      //这里通过数据库连接来提交事务
10.         con.commit();
11.     }
12.    .......
13. }
14.
15. protected void doRollback(DefaultTransactionStatus status) {
16.     DataSourceTransactionObject txObject = (DataSourceTransactionObject) status.getTransaction();
17.     Connection con = txObject.getConnectionHolder().getConnection();
18.     if (status.isDebug()) {
19.         logger.debug("Rolling back JDBC transaction on Connection [" + con + "]");
```

```
20.     }
21.     try {
22.     //这里通过数据库连接来回滚事务
23.         con.rollback();
24.     }
25.     catch (SQLException ex) {
26.         throw new TransactionSystemException("Could not roll back JDBC transaction", ex);
27.     }
28. }
```

我们看到在 DataSourceTransactionManager 中最后还是交给 connection 来实现事务的提交和 rollback。整个声明式事务处理是事务处理在 Spring AOP 中的应用，我们看到了一个很好的使用 Spring AOP 的例子，在 Spring 声明式事务处理的源代码中我们可以看到：

1.怎样封装各种不同平台下的事务处理代码

2.怎样读取属性值和结合事务处理代码来完成既定的事务处理策略

3.怎样灵活的使用 SpringAOP 框架。

如果能够结合前面的 Spring AOP 的源代码来学习，理解可能会更深刻些。

声明：JavaEye 文章版权属于作者，受法律保护。没有作者书面许可不得转载。

七、Spring AOP 中对拦截器调用的实现

前面我们分析了 Spring AOP 实现中得到 Proxy 对象的过程，下面我们看看在 Spring AOP 中拦截器链是怎样被调用的，也就是 Proxy 模式是怎样起作用的，或者说 Spring 是怎样为我们提供 AOP 功能的；
在 JdkDynamicAopProxy 中生成 Proxy 对象的时候：
Java 代码

```
1.  return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
```

这里的 this 参数对应的是 InvocationHandler 对象,这里我们的 JdkDynamicAopProxy 实现了这个接口，也就是说当 Proxy 对象的函数被调用的时候，这个 InvocationHandler 的 invoke 方法会被作为回调函数调用，下面我们看看这个方法的实现：
Java 代码

```
1.  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
2.      MethodInvocation invocation = null;
3.      Object oldProxy = null;
4.      boolean setProxyContext = false;
5.
6.      TargetSource targetSource = this.advised.targetSource;
7.      Class targetClass = null;
8.      Object target = null;
9.
10.     try {
11.         // Try special rules for equals() method and implementation of the
12.         // Advised AOP configuration interface.
13.
14.         if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {
15.             // What if equals throws exception!?
16.             // This class implements the equals(Object) method itself.
17.             return equals(args[0]) ? Boolean.TRUE : Boolean.FALSE;
18.         }
19.         if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {
20.             // This class implements the hashCode() method itself.
21.             return new Integer(hashCode());
22.         }
23.         if (Advised.class == method.getDeclaringClass()) {
24.             // service invocations on ProxyConfig with the proxy config
25.             return AopUtils.invokeJoinpointUsingReflection(this.advised, method, args);
26.         }
27.
28.         Object retVal = null;
29.
30.         if (this.advised.exposeProxy) {
31.             // make invocation available if necessary
32.             oldProxy = AopContext.setCurrentProxy(proxy);
33.             setProxyContext = true;
34.         }
```

```
35.
36.         // May be <code>null</code>. Get as late as possible to minimize the time we "own" the targ
    et,
37.         // in case it comes from a pool.
38.         // 这里是得到目标对象的地方，当然这个目标对象可能来自于一个实例池或者是一个简单的 JAVA 对象
39.         target = targetSource.getTarget();
40.         if (target != null) {
41.             targetClass = target.getClass();
42.         }
43.
44.         // get the interception chain for this method
45.         // 这里获得定义好的拦截器链
46.         List chain = this.advised.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice
    (
47.                 this.advised, proxy, method, targetClass);
48.
49.         // Check whether we have any advice. If we don't, we can fallback on direct
50.         // reflective invocation of the target, and avoid creating a MethodInvocation.
51.         // 如果没有设定拦截器，那么我们就直接调用目标的对应方法
52.         if (chain.isEmpty()) {
53.             // We can skip creating a MethodInvocation: just invoke the target directly
54.             // Note that the final invoker must be an InvokerInterceptor so we know it does
55.             // nothing but a reflective operation on the target, and no hot swapping or fancy proxy
    ing
56.             retVal = AopUtils.invokeJoinpointUsingReflection(target, method, args);
57.         }
58.         else {
59.             // We need to create a method invocation...
60.             // invocation = advised.getMethodInvocationFactory().getMethodInvocation(
61.             //         proxy, method, targetClass, target, args, chain, advised);
62.             // 如果有拦截器的设定，那么需要调用拦截器之后才调用目标对象的相应方法
63.             // 这里通过构造一个 ReflectiveMethodInvocation 来实现，下面我们会看这个 ReflectiveMethodInvo
    cation 类
64.             invocation = new ReflectiveMethodInvocation(
65.                     proxy, target, method, args, targetClass, chain);
66.
67.             // proceed to the joinpoint through the interceptor chain
68.             // 这里通过 ReflectiveMethodInvocation 来调用拦截器链和相应的目标方法
69.             retVal = invocation.proceed();
70.         }
71.
72.         // massage return value if necessary
73.         if (retVal != null && retVal == target && method.getReturnType().isInstance(proxy)) {
74.             // Special case: it returned "this" and the return type of the method is type-compatibl
    e
75.             // Note that we can't help if the target sets
76.             // a reference to itself in another returned object.
```

```
            retVal = proxy;
        }
        return retVal;
    }
    finally {
        if (target != null && !targetSource.isStatic()) {
            // must have come from TargetSource
            targetSource.releaseTarget(target);
        }

        if (setProxyContext) {
            // restore old proxy
            AopContext.setCurrentProxy(oldProxy);
        }
    }
}
```

我们先看看目标对象方法的调用，这里是通过 AopUtils 的方法调用 - 使用反射机制来对目标对象的方法进行调用：

Java 代码

```
public static Object invokeJoinpointUsingReflection(Object target, Method method, Object[] args)
    throws Throwable {

    // Use reflection to invoke the method.
    // 利用放射机制得到相应的方法，并且调用 invoke
    try {
        if (!Modifier.isPublic(method.getModifiers()) ||
                !Modifier.isPublic(method.getDeclaringClass().getModifiers())) {
            method.setAccessible(true);
        }
        return method.invoke(target, args);
    }
    catch (InvocationTargetException ex) {
        // Invoked method threw a checked exception.
        // We must rethrow it. The client won't see the interceptor.
        throw ex.getTargetException();
    }
    catch (IllegalArgumentException ex) {
        throw new AopInvocationException("AOP configuration seems to be invalid: tried calling meth
od [" +
                method + "] on target [" + target + "]", ex);
    }
    catch (IllegalAccessException ex) {
        throw new AopInvocationException("Couldn't access method: " + method, ex);
    }
}
```

对拦截器链的调用处理是在 ReflectiveMethodInvocation 里实现的：

Java 代码

```
public Object proceed() throws Throwable {
    //    We start with an index of -1 and increment early.
    // 这里直接调用目标对象的方法，没有拦截器的调用或者拦截器已经调用完了，这个 currentInterceptorIndex 的初始值是 0
    if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size()) {
        return invokeJoinpoint();
    }

    Object interceptorOrInterceptionAdvice =
        this.interceptorsAndDynamicMethodMatchers.get(this.currentInterceptorIndex);
    if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher) {
        // Evaluate dynamic method matcher here: static part will already have
        // been evaluated and found to match.
        // 这里获得相应的拦截器，如果拦截器可以匹配的上的话，那就调用拦截器的 invoke 方法
        InterceptorAndDynamicMethodMatcher dm =
            (InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvice;
        if (dm.methodMatcher.matches(this.method, this.targetClass, this.arguments)) {
            return dm.interceptor.invoke(nextInvocation());
        }
        else {
            // Dynamic matching failed.
            // Skip this interceptor and invoke the next in the chain.
            // 如果拦截器匹配不上，那就调用下一个拦截器，这个时候拦截器链的位置指示后移并迭代调用当前的 proceed 方法
            this.currentInterceptorIndex++;
            return proceed();
        }
    }
    else {
        // It's an interceptor, so we just invoke it: The pointcut will have
        // been evaluated statically before this object was constructed.
        return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(nextInvocation());
    }
}
```

这里把当前的拦截器链以及在拦截器链的位置标志都 clone 到一个 MethodInvocation 对象了，作用是当前的拦截器执行完之后，会继续沿着得到这个拦截器链执行下面的拦截行为，也就是会迭代的调用上面这个 proceed:

Java 代码

```
private ReflectiveMethodInvocation nextInvocation() throws CloneNotSupportedException {
    ReflectiveMethodInvocation invocation = (ReflectiveMethodInvocation) clone();
    invocation.currentInterceptorIndex = this.currentInterceptorIndex + 1;
    invocation.parent = this;
    return invocation;
```

```
6.  }
```

这里的 nextInvocation 就已经包含了当前的拦截链的基本信息，我们看到在 Interceptor 中的实现比如 TransactionInterceptor 的实现中：

Java 代码

```
1.  public Object invoke(final MethodInvocation invocation) throws Throwable {
2.     ......//这里是 TransactionInterceptor 插入的事务处理代码，我们会在后面分析事务处理实现的时候进行分析
3.          try {
4.              //这里是对配置的拦截器链进行迭代处理的调用
5.              retVal = invocation.proceed();
6.          }
7.     ......//省略了和事务处理的异常处理代码 ， 也是 TransactionInterceptor 插入的处理
8.        else {
9.          try {
10.             Object result = ((CallbackPreferringPlatformTransactionManager) getTransactionManager
    ()).execute(txAttr,
11.                     new TransactionCallback() {
12.                         public Object doInTransaction(TransactionStatus status) {
13.                              //这里是 TransactionInterceptor 插入对事务处理的代码
14.                             TransactionInfo txInfo = prepareTransactionInfo(txAttr, joinpointIdenti
    fication, status);
15.                             //这里是对配置的拦截器链进行迭代处理的调用，接着顺着拦截器进行处理
16.                             try {
17.                                 return invocation.proceed();
18.                             }
19.    ......//省略了和事务处理的异常处理代码 ， 也是 TransactionInterceptor 插入的处理
20.    }
```

从上面的分析我们看到了 Spring AOP 的基本实现，比如 Spring 怎样得到 Proxy,怎样利用 JAVA Proxy 以及反射机制对用户定义的拦截器链进行处理。

声明：JavaEye 文章版权属于作者，受法律保护。没有作者书面许可不得转载。

八、Spring 驱动 Hibernate 的实现

O/R 工具出现之后，简化了许多复杂的信息持久化的开发。Spring 应用开发者可以通过 Spring 提供的 O/R 方案更方便的使用各种持久化工具，比如 Hibernate；下面我们就 Spring+Hibernate 中的 Spring 实现做一个简单的剖析。

Spring 对 Hinberanate 的配置是通过 LocalSessionFactoryBean 来完成的，这是一个工厂 Bean 的实现，在基类 AbstractSessionFactoryBean 中：

Java 代码

```
/**
 * 这是 FactoryBean 需要实现的接口方法，直接取得当前的 sessionFactory 的值
 */
public Object getObject() {
    return this.sessionFactory;
}
```

这个值在 afterPropertySet 中定义：

Java 代码

```
public void afterPropertiesSet() throws Exception {
    //这个 buildSessionFactory 是通过配置信息得到 SessionFactory 的地方
    SessionFactory rawSf = buildSessionFactory();
    //这里使用了 Proxy 方法插入对 getCurrentSession 的拦截，得到和事务相关的 session
    this.sessionFactory = wrapSessionFactoryIfNecessary(rawSf);
}
```

我们先看看 SessionFactory 是怎样创建的，这个方法很长，包含了创建 Hibernate 的 SessionFactory 的详尽步骤：

Java 代码

```
protected SessionFactory buildSessionFactory() throws Exception {
    SessionFactory sf = null;

    // Create Configuration instance.
    Configuration config = newConfiguration();

    //这里配置数据源，事务管理器，LobHander 到 Holder 中，这个 Holder 是一个 ThreadLocal 变量,这样这些资源就和线程绑定了
    if (this.dataSource != null) {
        // Make given DataSource available for SessionFactory configuration.
        configTimeDataSourceHolder.set(this.dataSource);
    }

    if (this.jtaTransactionManager != null) {
        // Make Spring-provided JTA TransactionManager available.
        configTimeTransactionManagerHolder.set(this.jtaTransactionManager);
    }

    if (this.lobHandler != null) {
```

```
19.         // Make given LobHandler available for SessionFactory configuration.
20.         // Do early because because mapping resource might refer to custom types.
21.         configTimeLobHandlerHolder.set(this.lobHandler);
22.     }
23.
24.     //这里是使用 Hibernate 的各个属性的配置，这里使用了 Configuration 类来抽象这些数据
25.     try {
26.         // Set connection release mode "on_close" as default.
27.         // This was the case for Hibernate 3.0; Hibernate 3.1 changed
28.         // it to "auto" (i.e. "after_statement" or "after_transaction").
29.         // However, for Spring's resource management (in particular for
30.         // HibernateTransactionManager), "on_close" is the better default.
31.         config.setProperty(Environment.RELEASE_CONNECTIONS, ConnectionReleaseMode.ON_CLOSE.toString
    ());
32.
33.         if (!isExposeTransactionAwareSessionFactory()) {
34.             // Not exposing a SessionFactory proxy with transaction-aware
35.             // getCurrentSession() method -> set Hibernate 3.1 CurrentSessionContext
36.             // implementation instead, providing the Spring-managed Session that way.
37.             // Can be overridden by a custom value for corresponding Hibernate property.
38.             config.setProperty(Environment.CURRENT_SESSION_CONTEXT_CLASS,
39.                     "org.springframework.orm.hibernate3.SpringSessionContext");
40.         }
41.
42.         if (this.entityInterceptor != null) {
43.             // Set given entity interceptor at SessionFactory level.
44.             config.setInterceptor(this.entityInterceptor);
45.         }
46.
47.         if (this.namingStrategy != null) {
48.             // Pass given naming strategy to Hibernate Configuration.
49.             config.setNamingStrategy(this.namingStrategy);
50.         }
51.
52.         if (this.typeDefinitions != null) {
53.             // Register specified Hibernate type definitions.
54.             Mappings mappings = config.createMappings();
55.             for (int i = 0; i < this.typeDefinitions.length; i++) {
56.                 TypeDefinitionBean typeDef = this.typeDefinitions[i];
57.                 mappings.addTypeDef(typeDef.getTypeName(), typeDef.getTypeClass(), typeDef.getParam
    eters());
58.             }
59.         }
60.
61.         if (this.filterDefinitions != null) {
62.             // Register specified Hibernate FilterDefinitions.
63.             for (int i = 0; i < this.filterDefinitions.length; i++) {
```

```
64.                 config.addFilterDefinition(this.filterDefinitions[i]);
65.             }
66.         }
67.
68.         if (this.configLocations != null) {
69.             for (int i = 0; i < this.configLocations.length; i++) {
70.                 // Load Hibernate configuration from given location.
71.                 config.configure(this.configLocations[i].getURL());
72.             }
73.         }
74.
75.         if (this.hibernateProperties != null) {
76.             // Add given Hibernate properties to Configuration.
77.             config.addProperties(this.hibernateProperties);
78.         }
79.
80.         if (this.dataSource != null) {
81.             boolean actuallyTransactionAware =
82.                     (this.useTransactionAwareDataSource || this.dataSource instanceof TransactionAw
    areDataSourceProxy);
83.             // Set Spring-provided DataSource as Hibernate ConnectionProvider.
84.             config.setProperty(Environment.CONNECTION_PROVIDER,
85.                     actuallyTransactionAware ?
86.                     TransactionAwareDataSourceConnectionProvider.class.getName() :
87.                     LocalDataSourceConnectionProvider.class.getName());
88.         }
89.
90.         if (this.jtaTransactionManager != null) {
91.             // Set Spring-provided JTA TransactionManager as Hibernate property.
92.             config.setProperty(
93.                     Environment.TRANSACTION_MANAGER_STRATEGY, LocalTransactionManagerLookup.class.g
    etName());
94.         }
95.
96.         if (this.mappingLocations != null) {
97.             // Register given Hibernate mapping definitions, contained in resource files.
98.             for (int i = 0; i < this.mappingLocations.length; i++) {
99.                 config.addInputStream(this.mappingLocations[i].getInputStream());
100.             }
101.         }
102.
103.         if (this.cacheableMappingLocations != null) {
104.             // Register given cacheable Hibernate mapping definitions, read from the file syste
    m.
105.             for (int i = 0; i < this.cacheableMappingLocations.length; i++) {
106.                 config.addCacheableFile(this.cacheableMappingLocations[i].getFile());
107.             }
```

```
108.         }
109.
110.         if (this.mappingJarLocations != null) {
111.             // Register given Hibernate mapping definitions, contained in jar files.
112.             for (int i = 0; i < this.mappingJarLocations.length; i++) {
113.                 Resource resource = this.mappingJarLocations[i];
114.                 config.addJar(resource.getFile());
115.             }
116.         }
117.
118.         if (this.mappingDirectoryLocations != null) {
119.             // Register all Hibernate mapping definitions in the given directories.
120.             for (int i = 0; i < this.mappingDirectoryLocations.length; i++) {
121.                 File file = this.mappingDirectoryLocations[i].getFile();
122.                 if (!file.isDirectory()) {
123.                     throw new IllegalArgumentException(
124.                             "Mapping directory location [" + this.mappingDirectoryLocations
    [i] +
125.                             "] does not denote a directory");
126.                 }
127.                 config.addDirectory(file);
128.             }
129.         }
130.
131.         if (this.entityCacheStrategies != null) {
132.             // Register cache strategies for mapped entities.
133.             for (Enumeration classNames = this.entityCacheStrategies.propertyNames(); classNames.
    hasMoreElements();) {
134.                 String className = (String) classNames.nextElement();
135.                 String[] strategyAndRegion =
136.                         StringUtils.commaDelimitedListToStringArray(this.entityCacheStrategies.ge
    tProperty(className));
137.                 if (strategyAndRegion.length > 1) {
138.                     config.setCacheConcurrencyStrategy(className, strategyAndRegion[0], strategyA
    ndRegion[1]);
139.                 }
140.                 else if (strategyAndRegion.length > 0) {
141.                     config.setCacheConcurrencyStrategy(className, strategyAndRegion[0]);
142.                 }
143.             }
144.         }
145.
146.         if (this.collectionCacheStrategies != null) {
147.             // Register cache strategies for mapped collections.
148.             for (Enumeration collRoles = this.collectionCacheStrategies.propertyNames(); collRole
    s.hasMoreElements();) {
149.                 String collRole = (String) collRoles.nextElement();
```

```
150.                 String[] strategyAndRegion =
151.                         StringUtils.commaDelimitedListToStringArray(this.collectionCacheStrategie
    s.getProperty(collRole));
152.                 if (strategyAndRegion.length > 1) {
153.                     config.setCollectionCacheConcurrencyStrategy(collRole, strategyAndRegion
    [0], strategyAndRegion[1]);
154.                 }
155.                 else if (strategyAndRegion.length > 0) {
156.                     config.setCollectionCacheConcurrencyStrategy(collRole, strategyAndRegion
    [0]);
157.                 }
158.             }
159.         }
160.
161.         if (this.eventListeners != null) {
162.             // Register specified Hibernate event listeners.
163.             for (Iterator it = this.eventListeners.entrySet().iterator(); it.hasNext();) {
164.                 Map.Entry entry = (Map.Entry) it.next();
165.                 Assert.isTrue(entry.getKey() instanceof String, "Event listener key needs to be o
    f type String");
166.                 String listenerType = (String) entry.getKey();
167.                 Object listenerObject = entry.getValue();
168.                 if (listenerObject instanceof Collection) {
169.                     Collection listeners = (Collection) listenerObject;
170.                     EventListeners listenerRegistry = config.getEventListeners();
171.                     Object[] listenerArray =
172.                             (Object[]) Array.newInstance(listenerRegistry.getListenerClassFor(lis
    tenerType), listeners.size());
173.                     listenerArray = listeners.toArray(listenerArray);
174.                     config.setListeners(listenerType, listenerArray);
175.                 }
176.                 else {
177.                     config.setListener(listenerType, listenerObject);
178.                 }
179.             }
180.         }
181.
182.         // Perform custom post-processing in subclasses.
183.         postProcessConfiguration(config);
184.
185.         // 这里是根据 Configuration 配置创建 SessionFactory 的地方
186.         logger.info("Building new Hibernate SessionFactory");
187.         this.configuration = config;
188.         sf = newSessionFactory(config);
189.     }
190.     //最后把和线程绑定的资源清空
191.     finally {
```

```
192.         if (this.dataSource != null) {
193.             // Reset DataSource holder.
194.             configTimeDataSourceHolder.set(null);
195.         }
196.
197.         if (this.jtaTransactionManager != null) {
198.             // Reset TransactionManager holder.
199.             configTimeTransactionManagerHolder.set(null);
200.         }
201.
202.         if (this.lobHandler != null) {
203.             // Reset LobHandler holder.
204.             configTimeLobHandlerHolder.set(null);
205.         }
206.     }
207.
208.     // Execute schema update if requested.
209.     if (this.schemaUpdate) {
210.         updateDatabaseSchema();
211.     }
212.
213.     return sf;
214. }
```

而直接调用 org.hibernate.cfg.Configuration 来得到需要的 SessionFactory:

Java 代码

```
1.  protected SessionFactory newSessionFactory(Configuration config) throws HibernateException {
2.      return config.buildSessionFactory();
3.  }
```

所以我们这里看到LocalSessionFactory大致起到的一个读取资源配置然后生成SessionFactory的作用；当然这里在得到 SessionFactory之后，还需要对 session 的事务管理作一些处理 - 使用了一个 Proxy 模式对 getCurrentSession 方法进行了拦截；

Java 代码

```
1.  //这里先根据当前的 SessionFactory 的类型得到 Proxy，然后插入 Spring 定义好的 getCurrentSession 拦截器
2.      protected SessionFactory getTransactionAwareSessionFactoryProxy(SessionFactory target) {
3.          Class sfInterface = SessionFactory.class;
4.          if (target instanceof SessionFactoryImplementor) {
5.              sfInterface = SessionFactoryImplementor.class;
6.          }
7.          return (SessionFactory) Proxy.newProxyInstance(sfInterface.getClassLoader(),
8.                  new Class[] {sfInterface}, new TransactionAwareInvocationHandler(target));
9.      }
```

拦截器的实现如下：

Java 代码

```
1.  private static class TransactionAwareInvocationHandler implements InvocationHandler {
```

```
2.
3.      private final SessionFactory target;
4.
5.      public TransactionAwareInvocationHandler(SessionFactory target) {
6.          this.target = target;
7.      }
8.
9.      public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
10.         // Invocation on SessionFactory/SessionFactoryImplementor interface coming in...
11.         // 这里对 getCurrentSession 方法进行拦截，得到一个和当前事务绑定的 session 交给用户
12.         if (method.getName().equals("getCurrentSession")) {
13.             // Handle getCurrentSession method: return transactional Session, if any.
14.             try {
15.                 return SessionFactoryUtils.doGetSession((SessionFactory) proxy, false);
16.             }
17.             catch (IllegalStateException ex) {
18.                 throw new HibernateException(ex.getMessage());
19.             }
20.         }
21.         else if (method.getName().equals("equals")) {
22.             // Only consider equal when proxies are identical.
23.             return (proxy == args[0] ? Boolean.TRUE : Boolean.FALSE);
24.         }
25.         else if (method.getName().equals("hashCode")) {
26.             // Use hashCode of SessionFactory proxy.
27.             return new Integer(hashCode());
28.         }
29.
30.         // 这里是需要运行的 SessionFactory 的目标方法
31.         try {
32.             return method.invoke(this.target, args);
33.         }
34.         catch (InvocationTargetException ex) {
35.             throw ex.getTargetException();
36.         }
37.     }
38. }
```

我们看看 getCurrentSession 的实现，在 SessionFactoryUtils 中：

Java 代码

```
1.  private static Session doGetSession(
2.              SessionFactory sessionFactory, Interceptor entityInterceptor,
3.              SQLExceptionTranslator jdbcExceptionTranslator, boolean allowCreate)
4.              throws HibernateException, IllegalStateException {
5.
6.          Assert.notNull(sessionFactory, "No SessionFactory specified");
```

```
7.
8.          //这个 TransactionSynchronizationManager 的 Resource 是一个 ThreadLocal 变量,sessionFactory 是一个单例，但 ThreadLocal 是和线程绑定的
9.          //这样就实现了 Hiberante 中常用的通过 ThreadLocal 的 session 管理机制
10.         SessionHolder sessionHolder = (SessionHolder) TransactionSynchronizationManager.getResource(sessionFactory);
11.         if (sessionHolder != null && !sessionHolder.isEmpty()) {
12.             // pre-bound Hibernate Session
13.             Session session = null;
14.             if (TransactionSynchronizationManager.isSynchronizationActive() &&
15.                     sessionHolder.doesNotHoldNonDefaultSession()) {
16.                 // Spring transaction management is active ->
17.                 // register pre-bound Session with it for transactional flushing.
18.                 session = sessionHolder.getValidatedSession();
19.                 if (session != null && !sessionHolder.isSynchronizedWithTransaction()) {
20.                     logger.debug("Registering Spring transaction synchronization for existing Hibernate Session");
21.                     TransactionSynchronizationManager.registerSynchronization(
22.                             new SpringSessionSynchronization(sessionHolder, sessionFactory, jdbcExceptionTranslator, false));
23.                     sessionHolder.setSynchronizedWithTransaction(true);
24.                     // Switch to FlushMode.AUTO, as we have to assume a thread-bound Session
25.                     // with FlushMode.NEVER, which needs to allow flushing within the transaction.
26.                     FlushMode flushMode = session.getFlushMode();
27.                     if (flushMode.lessThan(FlushMode.COMMIT) &&
28.                             !TransactionSynchronizationManager.isCurrentTransactionReadOnly()) {
29.                         session.setFlushMode(FlushMode.AUTO);
30.                         sessionHolder.setPreviousFlushMode(flushMode);
31.                     }
32.                 }
33.             }
34.             else {
35.                 // No Spring transaction management active -> try JTA transaction synchronization.
36.                 session = getJtaSynchronizedSession(sessionHolder, sessionFactory, jdbcExceptionTranslator);
37.             }
38.             if (session != null) {
39.                 return session;
40.             }
41.         }
42.         //这里直接打开一个 Session
43.         logger.debug("Opening Hibernate Session");
44.         Session session = (entityInterceptor != null ?
45.                 sessionFactory.openSession(entityInterceptor) : sessionFactory.openSession());
46.
```

```
47.         // Use same Session for further Hibernate actions within the transaction.
48.         // Thread object will get removed by synchronization at transaction completion.
49.         // 把新打开的 Session 放到 SessionHolder,然后放到 ThreadLocal 里面去和线程绑定起来，这个 ThreadLocal 是在 TransactionSynchronizationManager 中配置好的，可以根据 sessionFactory 来索取
50.         // 同时根据事务处理的状态来配置 session 的属性,比如把 FlushMode 设置为 Never，同时把 session 和事务处理关联起来
51.         if (TransactionSynchronizationManager.isSynchronizationActive()) {
52.             // We're within a Spring-managed transaction, possibly from JtaTransactionManager.
53.             logger.debug("Registering Spring transaction synchronization for new Hibernate Session");
54.             SessionHolder holderToUse = sessionHolder;
55.             if (holderToUse == null) {
56.                 holderToUse = new SessionHolder(session);
57.             }
58.             else {
59.                 holderToUse.addSession(session);
60.             }
61.             if (TransactionSynchronizationManager.isCurrentTransactionReadOnly()) {
62.                 session.setFlushMode(FlushMode.NEVER);
63.             }
64.             TransactionSynchronizationManager.registerSynchronization(
65.                     new SpringSessionSynchronization(holderToUse, sessionFactory, jdbcExceptionTranslator, true));
66.             holderToUse.setSynchronizedWithTransaction(true);
67.             if (holderToUse != sessionHolder) {
68.                 TransactionSynchronizationManager.bindResource(sessionFactory, holderToUse);
69.             }
70.         }
71.         else {
72.             // No Spring transaction management active -> try JTA transaction synchronization.
73.             registerJtaSynchronization(session, sessionFactory, jdbcExceptionTranslator, sessionHolder);
74.         }
75.
76.         // Check whether we are allowed to return the Session.
77.         if (!allowCreate && !isSessionTransactional(session, sessionFactory)) {
78.             closeSession(session);
79.             throw new IllegalStateException("No Hibernate Session bound to thread, " +
80.                 "and configuration does not allow creation of non-transactional one here");
81.         }
82.
83.         return session;
84.     }
```

这里就是在 Spring 中为使用 Hiberante 的 SessionFactory 以及 Session 做的准备工作，在这个基础上，用户可以通过使用 HibernateTemplate 来使用 Hibernate 的 O/R 功能，和以前看到的一样这是一个 execute 的回调：

Java 代码

```
1.  public Object execute(HibernateCallback action, boolean exposeNativeSession) throws DataAccessExcep
    tion {
2.      Assert.notNull(action, "Callback object must not be null");
3.      //这里得到配置好的 Hibernate 的 Session
4.      Session session = getSession();
5.      boolean existingTransaction = SessionFactoryUtils.isSessionTransactional(session, getSessionFac
    tory());
6.      if (existingTransaction) {
7.          logger.debug("Found thread-bound Session for HibernateTemplate");
8.      }
9.
10.     FlushMode previousFlushMode = null;
11.     try {
12.         previousFlushMode = applyFlushMode(session, existingTransaction);
13.         enableFilters(session);
14.         Session sessionToExpose = (exposeNativeSession ? session : createSessionProxy(session));
15.         //这里是回调的入口
16.         Object result = action.doInHibernate(sessionToExpose);
17.         flushIfNecessary(session, existingTransaction);
18.         return result;
19.     }
20.     catch (HibernateException ex) {
21.         throw convertHibernateAccessException(ex);
22.     }
23.     catch (SQLException ex) {
24.         throw convertJdbcAccessException(ex);
25.     }
26.     catch (RuntimeException ex) {
27.         // Callback code threw application exception...
28.         throw ex;
29.     }
30.     finally {
31.         //如果这个调用的方法在一个事务当中，
32.         if (existingTransaction) {
33.             logger.debug("Not closing pre-bound Hibernate Session after HibernateTemplate");
34.             disableFilters(session);
35.             if (previousFlushMode != null) {
36.                 session.setFlushMode(previousFlushMode);
37.             }
38.         } //否则把 Session 关闭
39.         else {
40.             // Never use deferred close for an explicitly new Session.
41.             if (isAlwaysUseNewSession()) {
42.                 SessionFactoryUtils.closeSession(session);
43.             }
44.             else {
```

```
45.                 SessionFactoryUtils.closeSessionOrRegisterDeferredClose(session, getSessionFactory
    ());
46.             }
47.         }
48.     }
49. }
```

我们看看怎样得到对应的 Session 的，仍然使用了 SessionFactoryUtils 的方法 doGetSession:

Java 代码

```
1.  protected Session getSession() {
2.      if (isAlwaysUseNewSession()) {
3.          return SessionFactoryUtils.getNewSession(getSessionFactory(), getEntityInterceptor());
4.      }
5.      else if (!isAllowCreate()) {
6.          return SessionFactoryUtils.getSession(getSessionFactory(), false);
7.      }
8.      else {
9.          return SessionFactoryUtils.getSession(
10.                 getSessionFactory(), getEntityInterceptor(), getJdbcExceptionTranslator());
11.     }
12. }
```

这样我们就可以和其他的 Template 那样使用 Hibernate 的基本功能了，使用的时候 Spring 已经为我们对 Session 的获取和关闭，事务处理的绑定做好了封装 - 从这个角度看也大大方便了用户的使用。

九、Spring Acegi 框架鉴权的实现

简单分析一下 Spring Acegi 的源代码实现：

Servlet.Filter 的实现 AuthenticationProcessingFilter 启动 Web 页面的验证过程 - 在 AbstractProcessingFilter 定义了整个验证过程的模板：

Java 代码

```
1.  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
2.      throws IOException, ServletException {
3.      //这里检验是不是符合 ServletRequest/SevletResponse 的要求
4.      if (!(request instanceof HttpServletRequest)) {
5.          throw new ServletException("Can only process HttpServletRequest");
6.      }
7.
8.      if (!(response instanceof HttpServletResponse)) {
9.          throw new ServletException("Can only process HttpServletResponse");
10.     }
11.
12.     HttpServletRequest httpRequest = (HttpServletRequest) request;
13.     HttpServletResponse httpResponse = (HttpServletResponse) response;
14.     //根据 HttpServletRequest 和 HttpServletResponse 来进行验证
15.     if (requiresAuthentication(httpRequest, httpResponse)) {
16.         if (logger.isDebugEnabled()) {
17.             logger.debug("Request is to process authentication");
18.         }
19.         //这里定义 Acegi 中的 Authentication 对象来持有相关的用户验证信息
20.         Authentication authResult;
21.
22.         try {
23.             onPreAuthentication(httpRequest, httpResponse);
24.             //这里的具体验证过程委托给子类完成，比如 AuthenticationProcessingFilter 来完成基于 Web 页面的用
    户验证
25.             authResult = attemptAuthentication(httpRequest);
26.         } catch (AuthenticationException failed) {
27.             // Authentication failed
28.             unsuccessfulAuthentication(httpRequest, httpResponse, failed);
29.
30.             return;
31.         }
32.
33.         // Authentication success
34.         if (continueChainBeforeSuccessfulAuthentication) {
35.             chain.doFilter(request, response);
36.         }
37.         //完成验证后的后续工作，比如跳转到相应的页面
38.         successfulAuthentication(httpRequest, httpResponse, authResult);
```

```
39.
40.         return;
41.     }
42.
43.     chain.doFilter(request, response);
44. }
```

在 AuthenticationProcessingFilter 中的具体验证过程是这样的：

Java 代码

```
1.  public Authentication attemptAuthentication(HttpServletRequest request)
2.      throws AuthenticationException {
3.      //这里从 HttpServletRequest 中得到用户验证的用户名和密码
4.      String username = obtainUsername(request);
5.      String password = obtainPassword(request);
6.
7.      if (username == null) {
8.          username = "";
9.      }
10.
11.     if (password == null) {
12.         password = "";
13.     }
14.     //这里根据得到的用户名和密码去构造一个 Authentication 对象提供给 AuthenticationManager 进行验证，里面包含了用户的用户名和密码信息
15.     UsernamePasswordAuthenticationToken authRequest = new UsernamePasswordAuthenticationToken(username, password);
16.
17.     // Place the last username attempted into HttpSession for views
18.     request.getSession().setAttribute(ACEGI_SECURITY_LAST_USERNAME_KEY, username);
19.
20.     // Allow subclasses to set the "details" property
21.     setDetails(request, authRequest);
22.     //这里启动 AuthenticationManager 进行验证过程
23.     return this.getAuthenticationManager().authenticate(authRequest);
24. }
```

在 Acegi 框架中，进行验证管理的主要类是 AuthenticationManager,我们看看它是怎样进行验证管理的 - 验证的调用入口是 authenticate 在 AbstractAuthenticationManager 的实现中：

//这是进行验证的函数，返回一个 Authentication 对象来记录验证的结果，其中包含了用户的验证信息，权限配置等，同时这个 Authentication 会以后被授权模块使用

Java 代码

```
1.  //如果验证失败，那么在验证过程中会直接抛出异常
2.      public final Authentication authenticate(Authentication authRequest)
3.          throws AuthenticationException {
4.          try {//这里是实际的验证处理，我们下面使用 ProviderManager 来说明具体的验证过程，传入的参数 authRequest 里面已经包含了从 HttpServletRequest 中得到的用户输入的用户名和密码
```

```
            Authentication authResult = doAuthentication(authRequest);
            copyDetails(authRequest, authResult);

            return authResult;
        } catch (AuthenticationException e) {
            e.setAuthentication(authRequest);
            throw e;
        }
    }
```

在 ProviderManager 中进行实际的验证工作，假设这里使用数据库来存取用户信息：

Java 代码

```
public Authentication doAuthentication(Authentication authentication)
    throws AuthenticationException {
    //这里取得配置好的 provider 链的迭代器，在配置的时候可以配置多个 provider,这里我们配置的是 DaoAuthentica
tionProvider 来说明，它使用数据库来保存用户的用户名和密码信息。
    Iterator iter = providers.iterator();

    Class toTest = authentication.getClass();

    AuthenticationException lastException = null;

    while (iter.hasNext()) {
        AuthenticationProvider provider = (AuthenticationProvider) iter.next();

        if (provider.supports(toTest)) {
            logger.debug("Authentication attempt using " + provider.getClass().getName());
            //这个 result 包含了验证中得到的结果信息
            Authentication result = null;

            try {//这里是 provider 进行验证处理的过程
                result = provider.authenticate(authentication);
                sessionController.checkAuthenticationAllowed(result);
            } catch (AuthenticationException ae) {
                lastException = ae;
                result = null;
            }

            if (result != null) {
                sessionController.registerSuccessfulAuthentication(result);
                publishEvent(new AuthenticationSuccessEvent(result));

                return result;
            }
        }
    }
```

```
34.
35.     if (lastException == null) {
36.         lastException = new ProviderNotFoundException(messages.getMessage("ProviderManager.provider
    NotFound",
37.                     new Object[] {toTest.getName()}, "No AuthenticationProvider found for {0}"));
38.     }
39.
40.     // 这里发布事件来通知上下文的监听器
41.     String className = exceptionMappings.getProperty(lastException.getClass().getName());
42.     AbstractAuthenticationEvent event = null;
43.
44.     if (className != null) {
45.         try {
46.             Class clazz = getClass().getClassLoader().loadClass(className);
47.             Constructor constructor = clazz.getConstructor(new Class[] {
48.                         Authentication.class, AuthenticationException.class
49.                     });
50.             Object obj = constructor.newInstance(new Object[] {authentication, lastException});
51.             Assert.isInstanceOf(AbstractAuthenticationEvent.class, obj, "Must be an AbstractAuthent
    icationEvent");
52.             event = (AbstractAuthenticationEvent) obj;
53.         } catch (ClassNotFoundException ignored) {}
54.         catch (NoSuchMethodException ignored) {}
55.         catch (IllegalAccessException ignored) {}
56.         catch (InstantiationException ignored) {}
57.         catch (InvocationTargetException ignored) {}
58.     }
59.
60.     if (event != null) {
61.         publishEvent(event);
62.     } else {
63.         if (logger.isDebugEnabled()) {
64.             logger.debug("No event was found for the exception " + lastException.getClass().getName
    ());
65.         }
66.     }
67.
68.     // Throw the exception
69.     throw lastException;
70. }
```

我们下面看看在 DaoAuthenticationProvider 是怎样从数据库中取出对应的验证信息进行用户验证的，在它的基类 AbstractUserDetailsAuthenticationProvider 定义了验证的处理模板：

Java 代码

```
1.  public Authentication authenticate(Authentication authentication)
2.      throws AuthenticationException {
```

```
3.      Assert.isInstanceOf(UsernamePasswordAuthenticationToken.class, authentication,
4.          messages.getMessage("AbstractUserDetailsAuthenticationProvider.onlySupports",
5.              "Only UsernamePasswordAuthenticationToken is supported"));
6.
7.      // 这里取得用户输入的用户名
8.      String username = (authentication.getPrincipal() == null) ? "NONE_PROVIDED" : authentication.ge
    tName();
9.      // 如果配置了缓存，从缓存中去取以前存入的用户验证信息 - 这里是 UserDetail，是服务器端存在数据库里的用户
    信息，这样就不用每次都去数据库中取了
10.     boolean cacheWasUsed = true;
11.     UserDetails user = this.userCache.getUserFromCache(username);
12.     //没有取到，设置标志位，下面会把这次取到的服务器端用户信息存入缓存中去
13.     if (user == null) {
14.         cacheWasUsed = false;
15.
16.         try {//这里是调用 UserDetailService 去取用户数据库里信息的地方
17.             user = retrieveUser(username, (UsernamePasswordAuthenticationToken) authentication);
18.         } catch (UsernameNotFoundException notFound) {
19.             if (hideUserNotFoundExceptions) {
20.                 throw new BadCredentialsException(messages.getMessage(
21.                         "AbstractUserDetailsAuthenticationProvider.badCredentials", "Bad credential
    s"));
22.             } else {
23.                 throw notFound;
24.             }
25.         }
26.
27.         Assert.notNull(user, "retrieveUser returned null - a violation of the interface contract
    ");
28.     }
29.
30.     if (!user.isAccountNonLocked()) {
31.         throw new LockedException(messages.getMessage("AbstractUserDetailsAuthenticationProvider.lo
    cked",
32.                 "User account is locked"));
33.     }
34.
35.     if (!user.isEnabled()) {
36.         throw new DisabledException(messages.getMessage("AbstractUserDetailsAuthenticationProvider.
    disabled",
37.                 "User is disabled"));
38.     }
39.
40.     if (!user.isAccountNonExpired()) {
41.         throw new AccountExpiredException(messages.getMessage("AbstractUserDetailsAuthenticationPro
    vider.expired",
42.                 "User account has expired"));
```

```
43.     }
44.
45.     // This check must come here, as we don't want to tell users
46.     // about account status unless they presented the correct credentials
47.     try {//这里是验证过程，在 retrieveUser 中从数据库中得到用户的信息，在 additionalAuthenticationChecks 中进行对比用户输入和服务器端的用户信息
48.           //如果验证通过，那么构造一个 Authentication 对象来让以后的授权使用，如果验证不通过，直接抛出异常结束鉴权过程
49.         additionalAuthenticationChecks(user, (UsernamePasswordAuthenticationToken) authentication);
50.     } catch (AuthenticationException exception) {
51.         if (cacheWasUsed) {
52.             // There was a problem, so try again after checking
53.             // we're using latest data (ie not from the cache)
54.             cacheWasUsed = false;
55.             user = retrieveUser(username, (UsernamePasswordAuthenticationToken) authentication);
56.             additionalAuthenticationChecks(user, (UsernamePasswordAuthenticationToken) authentication);
57.         } else {
58.             throw exception;
59.         }
60.     }
61.
62.     if (!user.isCredentialsNonExpired()) {
63.         throw new CredentialsExpiredException(messages.getMessage(
64.                 "AbstractUserDetailsAuthenticationProvider.credentialsExpired", "User credentials have expired"));
65.     }
66.     //根据前面的缓存结果决定是不是要把当前的用户信息存入缓存以供下次验证使用
67.     if (!cacheWasUsed) {
68.         this.userCache.putUserInCache(user);
69.     }
70.
71.     Object principalToReturn = user;
72.
73.     if (forcePrincipalAsString) {
74.         principalToReturn = user.getUsername();
75.     }
76.     //最后返回 Authentication 记录了验证结果供以后的授权使用
77.     return createSuccessAuthentication(principalToReturn, authentication, user);
78. }
79. //这是是调用 UserDetailService 去加载服务器端用户信息的地方，从什么地方加载要看设置，这里我们假设由 JdbcDaoImp 来从数据中进行加载
80. protected final UserDetails retrieveUser(String username, UsernamePasswordAuthenticationToken authentication)
81.     throws AuthenticationException {
82.     UserDetails loadedUser;
```

```
83.     //这里调用 UserDetailService 去从数据库中加载用户验证信息，同时返回从数据库中返回的信息，这些信息放到了 U
    serDetails 对象中去了
84.     try {
85.         loadedUser = this.getUserDetailsService().loadUserByUsername(username);
86.     } catch (DataAccessException repositoryProblem) {
87.         throw new AuthenticationServiceException(repositoryProblem.getMessage(), repositoryProble
    m);
88.     }
89.
90.     if (loadedUser == null) {
91.         throw new AuthenticationServiceException(
92.             "UserDetailsService returned null, which is an interface contract violation");
93.     }
94.     return loadedUser;
95. }
```

下面我们重点分析一下 JdbcDaoImp 这个类来看看具体是怎样从数据库中得到用户信息的：

Java 代码

```
1.  public class JdbcDaoImpl extends JdbcDaoSupport implements UserDetailsService {
2.      //~ Static fields/initializers =============================================================
    ======================
3.      //这里是预定义好的对查询语句，对应于默认的数据库表结构，也可以自己定义查询语句对应特定的用户数据库验证表
    的设计
4.      public static final String DEF_USERS_BY_USERNAME_QUERY =
5.              "SELECT username,password,enabled FROM users WHERE username = ?";
6.      public static final String DEF_AUTHORITIES_BY_USERNAME_QUERY =
7.              "SELECT username,authority FROM authorities WHERE username = ?";
8.
9.      //~ Instance fields ========================================================================
    ======================
10.     //这里使用 Spring JDBC 来进行数据库操作
11.     protected MappingSqlQuery authoritiesByUsernameMapping;
12.     protected MappingSqlQuery usersByUsernameMapping;
13.     private String authoritiesByUsernameQuery;
14.     private String rolePrefix = "";
15.     private String usersByUsernameQuery;
16.     private boolean usernameBasedPrimaryKey = true;
17.
18.     //~ Constructors ===========================================================================
    ======================
19.     //在初始化函数中把查询语句设置为预定义的 SQL 语句
20.     public JdbcDaoImpl() {
21.         usersByUsernameQuery = DEF_USERS_BY_USERNAME_QUERY;
22.         authoritiesByUsernameQuery = DEF_AUTHORITIES_BY_USERNAME_QUERY;
23.     }
24.
```

```
25.     //~ Methods ========================================================================================
    ======================
26.
27.     protected void addCustomAuthorities(String username, List authorities) {}
28.
29.     public String getAuthoritiesByUsernameQuery() {
30.         return authoritiesByUsernameQuery;
31.     }
32.
33.     public String getRolePrefix() {
34.         return rolePrefix;
35.     }
36.
37.     public String getUsersByUsernameQuery() {
38.         return usersByUsernameQuery;
39.     }
40.
41.     protected void initDao() throws ApplicationContextException {
42.         initMappingSqlQueries();
43.     }
44.
45.     /**
46.      * Extension point to allow other MappingSqlQuery objects to be substituted in a subclass
47.      */
48.     protected void initMappingSqlQueries() {
49.         this.usersByUsernameMapping = new UsersByUsernameMapping(getDataSource());
50.         this.authoritiesByUsernameMapping = new AuthoritiesByUsernameMapping(getDataSource());
51.     }
52.
53.     public boolean isUsernameBasedPrimaryKey() {
54.         return usernameBasedPrimaryKey;
55.     }
56.     //这里是取得数据库用户信息的具体过程
57.     public UserDetails loadUserByUsername(String username)
58.         throws UsernameNotFoundException, DataAccessException {
59.         //根据用户名在用户表中得到用户信息，包括用户名，密码和用户是否有效的信息
60.         List users = usersByUsernameMapping.execute(username);
61.
62.         if (users.size() == 0) {
63.             throw new UsernameNotFoundException("User not found");
64.         }
65.         //取集合中的第一个作为有效的用户对象
66.         UserDetails user = (UserDetails) users.get(0); // contains no GrantedAuthority[]
67.         //这里在权限表中去取得用户的权限信息，同样的返回一个权限集合对应于这个用户
68.         List dbAuths = authoritiesByUsernameMapping.execute(user.getUsername());
69.
70.         addCustomAuthorities(user.getUsername(), dbAuths);
```

```
71.
72.         if (dbAuths.size() == 0) {
73.             throw new UsernameNotFoundException("User has no GrantedAuthority");
74.         }
75.         //这里根据得到的权限集合来配置返回的 User 对象供以后使用
76.         GrantedAuthority[] arrayAuths = (GrantedAuthority[]) dbAuths.toArray(new GrantedAuthority[d
    bAuths.size()]);
77.
78.         String returnUsername = user.getUsername();
79.
80.         if (!usernameBasedPrimaryKey) {
81.             returnUsername = username;
82.         }
83.
84.         return new User(returnUsername, user.getPassword(), user.isEnabled(), true, true, true, arr
    ayAuths);
85.     }
86.
87.     public void setAuthoritiesByUsernameQuery(String queryString) {
88.         authoritiesByUsernameQuery = queryString;
89.     }
90.
91.     public void setRolePrefix(String rolePrefix) {
92.         this.rolePrefix = rolePrefix;
93.     }
94.
95.     public void setUsernameBasedPrimaryKey(boolean usernameBasedPrimaryKey) {
96.         this.usernameBasedPrimaryKey = usernameBasedPrimaryKey;
97.     }
98.
99.     public void setUsersByUsernameQuery(String usersByUsernameQueryString) {
100.         this.usersByUsernameQuery = usersByUsernameQueryString;
101.     }
102.
103.     //~ Inner Classes ==================================================================================
    ========================
104.
105.     /**
106.      * 这里是调用 Spring JDBC 的数据库操作，具体可以参考对 JDBC 的分析，这个类的作用是把数据库查询得到的记录
    集合转换为对象集合 - 一个很简单的 O/R 实现
107.      */
108.     protected class AuthoritiesByUsernameMapping extends MappingSqlQuery {
109.         protected AuthoritiesByUsernameMapping(DataSource ds) {
110.             super(ds, authoritiesByUsernameQuery);
111.             declareParameter(new SqlParameter(Types.VARCHAR));
112.             compile();
113.         }
```

```
114.
115.         protected Object mapRow(ResultSet rs, int rownum)
116.             throws SQLException {
117.             String roleName = rolePrefix + rs.getString(2);
118.             GrantedAuthorityImpl authority = new GrantedAuthorityImpl(roleName);
119.
120.             return authority;
121.         }
122.     }
123.
124.     /**
125.      * Query object to look up a user.
126.      */
127.     protected class UsersByUsernameMapping extends MappingSqlQuery {
128.         protected UsersByUsernameMapping(DataSource ds) {
129.             super(ds, usersByUsernameQuery);
130.             declareParameter(new SqlParameter(Types.VARCHAR));
131.             compile();
132.         }
133.
134.         protected Object mapRow(ResultSet rs, int rownum)
135.             throws SQLException {
136.             String username = rs.getString(1);
137.             String password = rs.getString(2);
138.             boolean enabled = rs.getBoolean(3);
139.             UserDetails user = new User(username, password, enabled, true, true, true,
140.                     new GrantedAuthority[] {new GrantedAuthorityImpl("HOLDER")});
141.
142.             return user;
143.         }
144.     }
145. }
```

从数据库中得到用户信息后，就是一个比对用户输入的信息和这个数据库用户信息的比对过程，这个比对过程在DaoAuthenticationProvider:

Java 代码

```
1.  //这个 UserDetail 是从数据库中查询到的，这个 authentication 是从用户输入中得到的
2.      protected void additionalAuthenticationChecks(UserDetails userDetails,
3.          UsernamePasswordAuthenticationToken authentication)
4.          throws AuthenticationException {
5.          Object salt = null;
6.
7.          if (this.saltSource != null) {
8.              salt = this.saltSource.getSalt(userDetails);
9.          }
10.         //如果用户没有输入密码，直接抛出异常
```

```
        if (authentication.getCredentials() == null) {
            throw new BadCredentialsException(messages.getMessage(
                    "AbstractUserDetailsAuthenticationProvider.badCredentials", "Bad credentials
"),
                    includeDetailsObject ? userDetails : null);
        }
        //这里取得用户输入的密码
        String presentedPassword = authentication.getCredentials() == null ? "" : authentication.ge
tCredentials().toString();
        //这里判断用户输入的密码是不是和数据库里的密码相同，这里可以使用 passwordEncoder 来对数据库里的密码
加解密
        // 如果不相同，抛出异常，如果相同则鉴权成功
        if (!passwordEncoder.isPasswordValid(
                userDetails.getPassword(), presentedPassword, salt)) {
            throw new BadCredentialsException(messages.getMessage(
                    "AbstractUserDetailsAuthenticationProvider.badCredentials", "Bad credentials
"),
                    includeDetailsObject ? userDetails : null);
        }
    }
```

上面分析了整个 Acegi 进行验证的过程，从 AuthenticationProcessingFilter 中拦截 Http 请求得到用户输入的用户名和密码，这些用户输入的验证信息会被放到 Authentication 对象中持有并传递给 AuthenticatioManager 来对比在服务端的用户信息来完成整个鉴权。这个鉴权完成以后会把有效的用户信息放在一个 Authentication 中供以后的授权模块使用。在具体的鉴权过程中，使用了我们配置好的各种 Provider 以及对应的 UserDetailService 和 Encoder 类来完成相应的获取服务器端用户数据以及与用户输入的验证信息的比对工作。

我们从 FilterSecurityInterceptor 我们从入手看看怎样进行授权的：

Java 代码

```
//这里是拦截器拦截 HTTP 请求的入口
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        FilterInvocation fi = new FilterInvocation(request, response, chain);
        invoke(fi);
    }
//这是具体的拦截调用
    public void invoke(FilterInvocation fi) throws IOException, ServletException {
        if ((fi.getRequest() != null) && (fi.getRequest().getAttribute(FILTER_APPLIED) != null)
            && observeOncePerRequest) {
           //在第一次进行过安全检查之后就不会再做了
            fi.getChain().doFilter(fi.getRequest(), fi.getResponse());
        } else {
            //这是第一次收到相应的请求，需要做安全检测，同时把标志为设置好 -  FILTER_APPLIED，下次就再有请
求就不会作相同的安全检查了
            if (fi.getRequest() != null) {
                fi.getRequest().setAttribute(FILTER_APPLIED, Boolean.TRUE);
            }
```

```
            //这里是做安全检查的地方
            InterceptorStatusToken token = super.beforeInvocation(fi);
            //接着向拦截器链执行
            try {
                fi.getChain().doFilter(fi.getRequest(), fi.getResponse());
            } finally {
                super.afterInvocation(token, null);
            }
        }
    }
```

我们看看在 AbstractSecurityInterceptor 是怎样对 HTTP 请求作安全检测的：

Java 代码

```
protected InterceptorStatusToken beforeInvocation(Object object) {
    Assert.notNull(object, "Object was null");

    if (!getSecureObjectClass().isAssignableFrom(object.getClass())) {
        throw new IllegalArgumentException("Security invocation attempted for object "
            + object.getClass().getName()
            + " but AbstractSecurityInterceptor only configured to support secure objects of type: "
            + getSecureObjectClass());
    }
    //这里读取配置 FilterSecurityInterceptor 的 ObjectDefinitionSource 属性，这些属性配置了资源的安全设置
    ConfigAttributeDefinition attr = this.obtainObjectDefinitionSource().getAttributes(object);

    if (attr == null) {
        if(rejectPublicInvocations) {
            throw new IllegalArgumentException(
                  "No public invocations are allowed via this AbstractSecurityInterceptor. "
                + "This indicates a configuration error because the "
                + "AbstractSecurityInterceptor.rejectPublicInvocations property is set to 'true'");
        }

        if (logger.isDebugEnabled()) {
            logger.debug("Public object - authentication not attempted");
        }

        publishEvent(new PublicInvocationEvent(object));

        return null; // no further work post-invocation
    }


    if (logger.isDebugEnabled()) {
```

```
32.         logger.debug("Secure object: " + object.toString() + "; ConfigAttributes: " + attr.toString
    ());
33.     }
34.     //这里从 SecurityContextHolder 中去取 Authentication 对象，一般在登录时会放到 SecurityContextHolder 中
    去
35.     if (SecurityContextHolder.getContext().getAuthentication() == null) {
36.         credentialsNotFound(messages.getMessage("AbstractSecurityInterceptor.authenticationNotFound
    ",
37.                 "An Authentication object was not found in the SecurityContext"), object, attr);
38.     }
39.
40.     // 如果前面没有处理鉴权，这里需要对鉴权进行处理
41.     Authentication authenticated;
42.
43.     if (!SecurityContextHolder.getContext().getAuthentication().isAuthenticated() || alwaysReauthen
    ticate) {
44.         try {//调用配置好的 AuthenticationManager 处理鉴权，如果鉴权不成功，抛出异常结束处理
45.             authenticated = this.authenticationManager.authenticate(SecurityContextHolder.getContex
    t()
46.                                                                                 .getAuthen
    tication());
47.         } catch (AuthenticationException authenticationException) {
48.             throw authenticationException;
49.         }
50.
51.         // We don't authenticated.setAuthentication(true), because each provider should do that
52.         if (logger.isDebugEnabled()) {
53.             logger.debug("Successfully Authenticated: " + authenticated.toString());
54.         }
55.         //这里把鉴权成功后得到的 Authentication 保存到 SecurityContextHolder 中供下次使用
56.         SecurityContextHolder.getContext().setAuthentication(authenticated);
57.     } else {//这里处理前面已经通过鉴权的请求，先从 SecurityContextHolder 中去取得 Authentication
58.         authenticated = SecurityContextHolder.getContext().getAuthentication();
59.
60.         if (logger.isDebugEnabled()) {
61.             logger.debug("Previously Authenticated: " + authenticated.toString());
62.         }
63.     }
64.
65.     // 这是处理授权的过程
66.     try {
67.         //调用配置好的 AccessDecisionManager 来进行授权
68.         this.accessDecisionManager.decide(authenticated, object, attr);
69.     } catch (AccessDeniedException accessDeniedException) {
70.         //授权不成功向外发布事件
71.         AuthorizationFailureEvent event = new AuthorizationFailureEvent(object, attr, authenticate
    d,
```

```
                accessDeniedException);
        publishEvent(event);

        throw accessDeniedException;
    }

    if (logger.isDebugEnabled()) {
        logger.debug("Authorization successful");
    }

    AuthorizedEvent event = new AuthorizedEvent(object, attr, authenticated);
    publishEvent(event);

    // 这里构建一个 RunAsManager 来替代当前的 Authentication 对象，默认情况下使用的是 NullRunAsManager 会把 SecurityContextHolder 中的 Authentication 对象清空
    Authentication runAs = this.runAsManager.buildRunAs(authenticated, object, attr);

    if (runAs == null) {
        if (logger.isDebugEnabled()) {
            logger.debug("RunAsManager did not change Authentication object");
        }

        // no further work post-invocation
        return new InterceptorStatusToken(authenticated, false, attr, object);
    } else {
        if (logger.isDebugEnabled()) {
            logger.debug("Switching to RunAs Authentication: " + runAs.toString());
        }

         SecurityContextHolder.getContext().setAuthentication(runAs);

         // revert to token.Authenticated post-invocation
         return new InterceptorStatusToken(authenticated, true, attr, object);
     }
}
```

到这里我们假设配置 AffirmativeBased 作为 AccessDecisionManager：

Java 代码

```
//这里定义了决策机制，需要全票才能通过
    public void decide(Authentication authentication, Object object, ConfigAttributeDefinition config)
        throws AccessDeniedException {
        //这里取得配置好的迭代器集合
        Iterator iter = this.getDecisionVoters().iterator();
        int deny = 0;
        //依次使用各个投票器进行投票，并对投票结果进行计票
```

```
        while (iter.hasNext()) {
            AccessDecisionVoter voter = (AccessDecisionVoter) iter.next();
            int result = voter.vote(authentication, object, config);
            //这是对投票结果进行处理，如果遇到其中一票通过，那就授权通过，如果是弃权或者反对，那就继续投票
            switch (result) {
            case AccessDecisionVoter.ACCESS_GRANTED:
                return;

            case AccessDecisionVoter.ACCESS_DENIED:
            //这里对反对票进行计数
                deny++;

                break;

            default:
                break;
            }
        }
        //如果有反对票，抛出异常，整个授权不通过
        if (deny > 0) {
            throw new AccessDeniedException(messages.getMessage("AbstractAccessDecisionManager.accessDenied",
                    "Access is denied"));
        }

        // 这里对弃权票进行处理，看看是全是弃权票的决定情况，默认是不通过，由 allowIfAllAbstainDecisions 变量控制
        checkAllowIfAllAbstainDecisions();
    }
```

具体的投票由投票器进行，我们这里配置了 RoleVoter 来进行投票：

```
    public int vote(Authentication authentication, Object object, ConfigAttributeDefinition config) {
        int result = ACCESS_ABSTAIN;
        //这里取得资源的安全配置
        Iterator iter = config.getConfigAttributes();

        while (iter.hasNext()) {
            ConfigAttribute attribute = (ConfigAttribute) iter.next();

            if (this.supports(attribute)) {
                result = ACCESS_DENIED;

                // 这里对资源配置的安全授权级别进行判断，也就是匹配 ROLE 为前缀的角色配置
                // 遍历每个配置属性，如果其中一个匹配该主体持有的 GrantedAuthority,则访问被允许。
                for (int i = 0; i < authentication.getAuthorities().length; i++) {
                    if (attribute.getAttribute().equals(authentication.getAuthorities()[i].getAuthority())) {
```

```
51.                         return ACCESS_GRANTED;
52.                     }
53.                 }
54.             }
55.         }
56.
57.         return result;
58.     }
```

上面就是对整个授权过程的一个分析，从 FilterSecurityInterceptor 拦截 Http 请求入手，然后读取对资源的安全配置以后，把这些信息交由 AccessDecisionManager 来进行决策，Spring 为我们提供了若干决策器来使用，在决策器中我们可以配置投票器来完成投票，我们在上面具体分析了角色投票器的使用过程。