* Spring Source
* 他人总结
  + Spring1)ioc 容器——BeanFactory 是最原始的 ioc 容器，有以下方法 1.getBean2.判断是否有 Bean，containsBean3.判断是否单例 isSingleton。BeanFactory 只是对 ioc 容器最基本行为作了定义，而不关心 Bean 是怎样定义和加载的。如果我们想要知道一个工厂具体产生对象的过程，则要看这个接口的实现类。在 spring 中实现这个接口有很多类，其中一个 是xmlBeanFactory。
* xmlBeanFactory 的功能是建立在 DefaultListablexmlBeanFactory 这个基本容器的基础上的，并在这个基本容器的基础上实行了其他诸如 xml 读取的附加功能。 xmlBeanFactory（Resource resource)构造函数，resource 是 spring 中对与外部资源的抽象，最常见的是文件的抽象，特别是 xml 文件，而且 resource 里面通常是保存了 spring 使用者的 Bean 定义，eg.applicationContext.xml 在被加载时，就会被抽象为 resource 处理。
* [我自己理解, resource 就是定义 Bean 的 xml 文件]。
  + loc 容器建立过程：1)创建 ioc 配置文件的抽象资源，这个抽象资源包含了 BeanDefinition 的定义信息。2)创建一个 BeanFactory，这里使用的是 DefaultListablexmlBeanFactory。3)创建一个载入 BeanDefinition 的读取器，这里使用 xmlBeanDefinitionReader 来载入 xml 文件形式的 BeanDefinition。4)然后将上面定义好的 resource 通过一个回调配置给 BeanFactory。
  + 5)从资源里读入配置信息，具体解析过程由 xmlBeanDefinitionReader 完成。6)ioc 容器建立起来。
* BeanDefinition 类似于 resource 接口的功能，起到的作用就是对所有的 Bean 进行一层抽象的统一，把形式各样的对象统一封装为一个便于 spring 内部进行协调管理和调度的数据结构。BeanDefinition 屏蔽了不同对象对于 spring 框架的差异。
* Resource 里有 inputStream。
* 解析 xml，获得 document 对象，接下来只要再对 document 结构进行分析便可知道 Bean 在 xml 中是怎么定义的，也就可以将其转化为 BeanDefinition 对象。我们配置的 Bean 的信息经过解析，在 spring 内部已经转换为 BeanDefinition 这种统一的结构，但这些数据还不能供 ioc 容器直接使用，需要在 ioc 容器中对这些 BeanDefinition 数据进行注册，注册完成的 BeanDefinition，都会以 BeanName 为 Key，BeanDefinition 为 value，交由 map 管理。注册完之后，一个 ioc 容器就可以用了。
* 自己理解的，xml 文件抽象为 resource 对象，Bean 抽象为 BeanDefinition 对象。
  + 2) 依赖注入——依赖注入发生在 getBean 方法中，getBean 又调用 dogetBean 方法。 getBean 是依赖注入的起点，之后调用 createBean 方法，创建过程又委托给了 docreateBean 方法。在 docreateBean 中有两个方法：1)createBeanInstance，生成 Bean 包含的 java 对象 2)populateBean 完成注入。在创建 Bean 的实例中，getInstantiationstrategy 方法挺重要，该方法作用是获得实例化策略对象，也就是指通过哪种方案进行实例化的过程。spring 当中提供两种方案进行实例化：BeanUtils 和 cglib。BeanUtils 实现机制是 java 反射，cglib 是一个第三方类库，采用的是一种字节码加强方式。Spring 中默认实例化策略为 cglib。populateBean 进行依赖注入，获得 BeanDefinition 中设置的 property 信息，简单理解依赖注入的过程就是对这些 property 进行赋值的过程，在配置 Bean 的属性时，属性可能有多种类型，我们在进行注入的时候，不同类型的属性我们不能一概而论地进行处理。集合类型属性和非集合类型属性差别很大，对不同的类型应该有不同的处理过程。所以要先判断 value 类型，再调用具体方法。
  + 3) aop——将那些与业务无关，却为业务模块所公共调用的逻辑或责任封装起来，称其为 aspect，便于减少系统的重复代码。使用模块技术，aop 把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。实现 aop 的两大技术：1)采用动态代理，利用截取消息的方式，对该消息进行装饰，以获取原有对象行为的执行。2)采用静态织入，引入特定的语法创建切面，从而可以使编译器可在编译期间织入有关切面的代码。
* spring 提供两种方式生成代理对象，jdkProxy 和 cglib。默认的策略是，若目标类是接口则用 jdk 动态代理技术，否则使用 cglib 生成代理。在 jdk 动态代理中使用 Proxy.newProxyInstance()生成代理对象（ JdkDynamicAopProxy 类的 getProxy 方法)， JdkDynamicAopProxy 也实现了 invocationhandler 接口，有 invoke 方法，就是在该方法中实现了切片织入。主流程可以简述为：获取可应用到此方法上的通知链（Interceptor chain)，若有，则应用通知，并执行 joinpoint，若没有，则直接反射执行 joinpoint。
* Introduction 是指给一个已有类添加方法或字段属性，Introduction 还可以在不改变现有类代码的情况下，让现有 java 类实现新的接口，或为其指定一个父类实现多继承，相对于 advice 可以动态改变程序的功能或流程来说，Introduction 用来改变类的静态结构。
* 拦截器，是对连接点进行拦截，从而在连接点前后加入自定义的切面模块功能。作用于同一个连接点的多个拦截器组成一个拦截器链，拦截器链上的每一个拦截器，通常会调用下一个拦截器。
* 连接点，程序执行过程中的行为，比如方法调用或特定异常被抛出。
* 切入点，指定一个 advice 将被引发的一系列的连接点的集合。aop 框架必须允许开发者指定切入点。
* 通知（advice)：在特定的连接点，aop 框架执行的动作。Spring 以拦截器作通知模型，维护一个围绕连接点的拦截器链。
* 拦截器（advisor)，自己理解，在 invoke 前后加入的方法就是通知。使用 spring 的 PointCutadvisor，只拦截特定的方法，一个 advisor 定义订一个 PointCut 和一个 advice，满足 PointCut（指定哪些方面需要拦截)，则执行相应的 advice（定义了增强的功能)。
* PointCutadvisor 有 两 个 常 用 实 现 类 ： NameMatchMethodPointCutadvisor 和 regexMethodPointCutadvisor。前者需要注入 mappedname 和 advice 属性，后者需要注入 pattern 和 advice 属性。mappedname 指明要拦截的方法名，pattern 按照正则表达式的方法指明了要拦截的方法名，advice 定义一个增强，即要加入的操作（需要自己实现 MethodBeforeAdvice、MethodafterAdvice、throwAdvice、Methodinterceptor 接口之一)，然后在 ProxyBeanFactory 的拦截器中注入这个 PointCutadvisor。注：一个 ProxyFactoryBean 只能指定一个代理目标。
* 在 spring 中配置 aop 很麻烦，首先需要编写 xxxadvice 类（需要自己实现 MethodBeforeAdvice、MethodafterAdvice、throwAdvice、Methodinterceptor 接口之一)，然后在 xml 配置 advisor。还要在 advisor 中注入 advice，然后将 advisor 加入 ProxyFactoryBean 中。而在 spring2.x 以后引入了 aspect 注解，只需要定义一个 aspect 类，在 aspect 中声明 advice 类（可同时声明多个)，然后在 xml 配置这个 aspect 类，最后添加一行<aop: aspect j-auto proxy>就可以搞定。
* 通知类型 接口 描述
* 前置通知 MethodBeforeAdvice 在目标方法调用前调用
* 后置通知 MethodafterAdvice 在目标方法调用后调用
* 异常通知 throwAdvice 在目标方法抛出异常时调用
* 环绕通知 Methodinterceptor 拦截对目标方法调用
* 还有一类是引入通知，用来定义切入点的。
* Spring IOC
* IOC=ConfigReader+ReflectionUtil
* 容器继承体系
* 1、从接口BeanFactory到HierarchicalBeanFactory，再到ConfigurableBeanFactory,这是一条主要的BeanFactory设计路径。在这条接口设计路径中，BeanFactory，是一条主要的BeanFactory设计路径。在这条接口设计路径中，BeanFactory接口定义了基本的Ioc容器的规范。在这个接口定义中，包括了getBean()这样的Ioc容器的基本方法（通过这个方法可以从容器中取得Bean)。而HierarchicalBeanFactory接口在继承了BeanFactory的基本接口后，增加了getParentBeanFactory()的接口功能，使BeanFactory具备了双亲Ioc容器的管理功能。在接下来的ConfigurableBeanFactory接口中，主要定义了一些对BeanFactory的配置功能，比如通过setParentBeanFactory()设置双亲Ioc容器，通过addBeanPostProcessor()配置Bean后置处理器，等等。通过这些接口设计的叠加，定义了BeanFactory就是最简单的Ioc容器的基本功能。
* 2、第二条接口设计主线是，以ApplicationContext作为核心的接口设计，这里涉及的主要接口设计有，从BeanFactory到ListableBeanFactory，再到ApplicationContext，再到我们常用的WebApplicationContext或者ConfigurableApplicationContext接口。我们常用的应用基本都是org.framework.context 包里的WebApplicationContext或者ConfigurableApplicationContext实现。在这个接口体系中，ListableBeanFactory和HierarchicalBeanFactory两个接口，连接BeanFactory接口定义和ApplicationContext应用的接口定义。在ListableBeanFactory接口中，细化了许多BeanFactory的接口功能，比如定义了getBeanDefinitionNames()接口方法；对于ApplicationContext接口，它通过继承MessageSource、ResourceLoader、ApplicationEventPublisher接口，在BeanFactory简单Ioc容器的基础上添加了许多对高级容器的特性支持。
* 3、.这个接口系统是以BeanFactory和ApplicationContext为核心设计的，而BeanFactory是Ioc容器中最基本的接口，在ApplicationContext的设计中，一方面，可以看到它继承了BeanFactory接口体系中的ListableBeanFactory、AutowireCapableBeanFactory、HierarchicalBeanFactory等BeanFactory的接口，具备了BeanFactory Ioc容器的基本功能；另一方面，通过继承MessageSource、ResourceLoader、ApplicationEventPublisher这些接口，BeanFactory为ApplicationContext赋予了更高级的Ioc容器特性。对于ApplicationContext而言，为了在Web环境中使用它，还设计了WebApplicationContext接口，而这个接口通过继承ThemeSource接口来扩充功能。
* BeanFactory（容器接口)

public interface BeanFactory {

* //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
* String FACTORY*BEAN*PREFIX = "&";
* //这里根据bean的名字，在IOC容器中得到bean实例，这个IOC容器就是一个大的抽象工厂。
* Object getBean(String name) throws BeansException;
* //这里根据bean的名字和Class类型来得到bean实例，和上面的方法不同在于它会抛出异常：如果根据名字取得的bean实例的Class类型和需要的不同的话。
* T getBean(String name, Class requiredType);
* T getBean(Class requiredType) throws BeansException;
* Object getBean(String name, Object... args) throws BeansException;
* //这里提供对bean的检索，看看是否在IOC容器有这个名字的bean
* boolean containsBean(String name);
* //这里根据bean名字得到bean实例，并同时判断这个bean是不是单件
* boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
* //这里根据bean名字得到bean实例，并同时判断这个bean是不是原型
* boolean isPrototype(String name) throws NoSuchBeanDefinitionException;
* //这里对得到bean实例的Class类型
* Class<?> getType(String name) throws NoSuchBeanDefinitionException;
* //这里得到bean的别名，如果根据别名检索，那么其原名也会被检索出来
* String[] getAliases(String name);
* }
* XmlBeanFactory（基础容器实现)

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

* 作为简单ioc容器系列最底层的实现XmlBeanFactory是建立在DefaultListableBeanFactory容器的基础之上的，并在这个基本容器的基础上实现了其他诸如xml读取的附加功能。
* Resource接口体系
* 仅仅使用 java 标准 java.net.URL 和针对不同 URL 前缀的标准处理器并不能满足我们对各种底层资源的访问，比如：我们就不能通过 URL 的标准实现来访问相对类路径或者相对 ServletContext 的各种资源。虽然我们可以针对特定的 url 前缀来注册一个新的 URLStreamHandler（和现有的针对各种特定前缀的处理器类似，比如 http：)，然而这往往会是一件比较麻烦的事情(要求了解 url 的实现机制等)，而且 url 接口也缺少了部分基本的方法，如检查当前资源是否存在的方法。
* 相对标准 url 访问机制，Spring 的 Resource 接口对抽象底层资源的访问提供了一套更好的机制。
* Resource 是 Spring 中对外部资源的抽象，最常见的是文件的抽象，特别是 xml 文件，而且 Resource 里面通常是保存了 Spring 使用者的 Bean 定义。
* 其实现类有：ByteArrayResouece,BeanDefinitionResource,InputStreamResource, ClassPathResource等
* ResourceLoader接口用于返回Resource对象；其实现可以看作是一个生产Resource的工厂类。Spring提供了一个适用于所有环境的DefaultResourceLoader实现，可以返回ClassPathResource、UrlResource。
* ResourceLoader在进行加载资源时需要使用前缀来指定需要加载：“classpath:path”表示返回ClasspathResource，“http://path”和“file:path”表示返回UrlResource资源，如果不加前缀则需要根据当前上下文来决定，DefaultResourceLoader默认实现可以加载classpath资源。
* 对于目前所有ApplicationContext都实现了ResourceLoader，因此可以使用其来加载资源。
* ClassPathXmlApplicationContext：不指定前缀将返回默认的ClassPathResource资源，否则将根据前缀来加载资源；
* WebApplicationContext：不指定前缀将返回ServletContextResource，否则将根据前缀来加载资源；
* ApplicationContext接口（高级容器接口)
* ApplicationContext是spring中较高级的容器。和BeanFactory类似，它可以加载配置文件中定义的bean，将所有的bean集中在一起，当有请求的时候分配bean。 另外，它增加了企业所需要的功能，比如，从属性文件中解析文本信息和将事件传递给所指定的监听器。这个容器在org.springframework.context.ApplicationContext接口中定义。ApplicationContext包含BeanFactory所有的功能，一般情况下，相对于BeanFactory，ApplicationContext会被推荐使用。
* 特点
* 1.支持不同的信息源。继承了MessageSource接口，这个接口为ApplicationContext提供了很多信息源的扩展功能，比如：国际化的实现为多语言版本的应用提供服务。
* 2.访问资源。这一特性主要体现在ResourcePatternResolver接口上，对Resource和ResourceLoader的支持，这样我们可以从不同地方得到Bean定义资源。这种抽象使用户程序可以灵活地定义Bean定义信息，尤其是从不同的IO途径得到Bean定义信息。这在接口上看不出来，不过一般来说，具体ApplicationContext都是继承了DefaultResourceLoader的子类。因为DefaultResourceLoader是AbstractApplicationContext的基类。
* 3.支持应用事件。继承了接口ApplicationEventPublisher，为应用环境引入了事件机制，这些事件和Bean的生命周期的结合为Bean的管理提供了便利。
* 4.附件服务。EnvironmentCapable里的服务让基本的Ioc功能更加丰富。
* 5.ListableBeanFactory和HierarchicalBeanFactory是继承的主要容器。
* 实现
* 最常被使用的ApplicationContext接口实现类：
* 1，FileSystemXmlApplicationContext：该容器从XML文件中加载已被定义的bean。在这里，你需要提供给构造器XML文件的完整路径。
* 2，ClassPathXmlApplicationContext：该容器从XML文件中加载已被定义的bean。在这里，你不需要提供XML文件的完整路径，只需正确配置CLASSPATH环境变量即可，因为，容器会从CLASSPATH中搜索bean配置文件。
* 3，WebXmlApplicationContext：该容器会在一个 web 应用程序的范围内加载在XML文件中
* ClassPathXmlApplicationContext（高级容器实现)

public ClassPathXmlApplicationContext(String[] configLocations, boolean refresh, ApplicationContext parent)  
 throws BeansException {  
  
 super(parent);  
 setConfigLocations(configLocations);  
 if (refresh) {  
 refresh();  
 }  
}

* Bean的注册
* AbstractApplicationContext#refresh（bean注册)
* refresh是AbstractApplicationContext中方法。
* 逻辑：
  + 1)初始化前的准备工作，比如对系统属性或者环境变量进行准备及验证
  + 2)初始化BeanFactory，并进行XML文件读取（component-scan->包括class文件)
  + 3)对BeanFactory进行各种功能填充，比如@Qualifier和@Autowired
  + 4)子类覆盖方法做额外的处理
  + 5)激活各种BeanFactory处理器
  + 6)注册拦截bean创建的bean处理器，这里只是注册，真正的调用是在getBean的时候
  + 7)为上下文初始化Message源，即为不同语言的消息体进行国际化处理
  + 8)初始化应用消息广播器，并放入applicationEventMulticaster bean中
  + 9)留给子类来初始化其他的bean
* 10)在所有注册的bean中查找listener bean，注册到消息广播器中
  + 11)初始化剩下的代理实例（非lazy-init)
  + 12)完成刷新过程，通知生命周期处理器lifecycleProcessor刷新过程，同时发出ContextRefreshEvent通知别人。

public void refresh() throws BeansException, IllegalStateException {  
 synchronized (this.startupShutdownMonitor) {  
 // 准备 刷新的上下文环境  
 prepareRefresh();  
  
 // 初始化BeanFactory，并进行XML文件的读取

* ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
* // 对BeanFactory进行各种功能填充
* prepareBeanFactory(beanFactory);
* try {  
   // 子类覆盖方法做额外的处理  
   postProcessBeanFactory(beanFactory);
* // 激活各种BeanFactory处理器  
   invokeBeanFactoryPostProcessors(beanFactory);
* // 注册拦截Bean创建的Bean处理器，这里只是注册，真正的调用是在getBean的时候  
   registerBeanPostProcessors(beanFactory);
* // 为上下文初始化Message源，即不同语言的消息体，国际化处理  
   initMessageSource();
* // 初始化应用消息广播器，并放入applicationEventMulticaster bean中  
   initApplicationEventMulticaster();
* // 留给子类来初始化其他bean  
   onRefresh();
* // 在所有注册的bean中查找Listener bean，注册到消息广播器中  
   registerListeners();
* // 初始化剩下的单例实例（除了lazy-init)  
   finishBeanFactoryInitialization(beanFactory);
* // 完成刷新过程，通知生命周期处理器lifecycleProcessor刷新过程，同时发出ContextRefreshEvent通知别人  
   finishRefresh();  
  }
* catch (BeansException ex) {  
   if (logger.isWarnEnabled()) {  
   logger.warn("Exception encountered during context initialization - " +  
   "cancelling refresh attempt: " + ex);  
   }
* // Destroy already created singletons to avoid dangling resources.  
   destroyBeans();
* // Reset 'active' flag.  
   cancelRefresh(ex);
* // Propagate exception to caller.  
   throw ex;  
  }
* finally {  
   // Reset common introspection caches in Spring's core, since we  
   // might not ever need metadata for singleton beans anymore...  
   resetCommonCaches();  
  }  
  }  
  }
  + 1) obtainFreshBeanFactory（创建beanFactory，解析XML)
* 看obtainFreshBeanFactory这个方法：
* protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {  
  refreshBeanFactory();  
  ConfigurableListableBeanFactory beanFactory = getBeanFactory();  
  if (logger.isDebugEnabled()) {  
   logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);  
  }  
  return beanFactory;  
  }
  + 1.1) refreshBeanFactory
* 看refreshBeanFactory这个方法，这个方法是抽象方法，有一种是在AbstractRefreshableApplicationContext这个类中实现的：
* @Override  
  protected final void refreshBeanFactory() throws BeansException {  
   if (hasBeanFactory()) {  
   destroyBeans();  
   closeBeanFactory();  
   }  
   try {  
   DefaultListableBeanFactory beanFactory = createBeanFactory();  
   beanFactory.setSerializationId(getId());  
   customizeBeanFactory(beanFactory);
* //该方法最终调用XmlBeanDefinitionReader类中loadBeanDefinitions(EncodedResource)  
   loadBeanDefinitions(beanFactory);  
   synchronized (this.beanFactoryMonitor) {  
   this.beanFactory = beanFactory;  
   }  
  }  
  catch (IOException ex) {  
   throw new ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);  
  }  
  }
  + 1.1.1) loadBeanDefinitions(beanFactory)（创建reader)
* 看loadBeanDefinitions(beanFactory)这个方法：
* @Override  
  protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansException, IOException {  
   // Create a new XmlBeanDefinitionReader for the given BeanFactory.  
   XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);
* // Configure the bean definition reader with this context's  
   // resource loading environment.  
   beanDefinitionReader.setEnvironment(this.getEnvironment());  
   beanDefinitionReader.setResourceLoader(this);  
   beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));
* // Allow a subclass to provide custom initialization of the reader,  
   // then proceed with actually loading the bean definitions.  
   initBeanDefinitionReader(beanDefinitionReader);  
   loadBeanDefinitions(beanDefinitionReader);  
  }
  + 1.1.1.1) loadBeanDefinitions(beanDefinitionReader)（调用reader的load)
* 看loadBeanDefinitions(beanDefinitionReader)的另一个重载：
* protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException, IOException {  
  Resource[] configResources = getConfigResources();  
  if (configResources != null) {  
   reader.loadBeanDefinitions(configResources);  
  }  
  String[] configLocations = getConfigLocations();  
  if (configLocations != null) {  
   reader.loadBeanDefinitions(configLocations);  
  }  
  }
  + 1.1.1.1.1) XmlBeanDefinitionReader.loadBeanDefinitions（解析标签)
* 看reader.loadBeanDefinitions方法：

@Override  
public int loadBeanDefinitions(String... locations) throws BeanDefinitionStoreException {  
 Assert.notNull(locations, "Location array must not be null");  
 int counter = 0;  
 for (String location : locations) {  
 counter += loadBeanDefinitions(location);  
 }  
 return counter;  
}

* loadBeanDefinitions(location,null)
* 它最终调用了loadBeanDefinitions(location,null)方法

public int loadBeanDefinitions(String location, Set<Resource> actualResources) throws BeanDefinitionStoreException {  
 ResourceLoader resourceLoader = getResourceLoader();  
 if (resourceLoader == null) {  
 throw new BeanDefinitionStoreException(  
 "Cannot import bean definitions from location [" + location + "]: no ResourceLoader available");  
 }  
  
 if (resourceLoader instanceof ResourcePatternResolver) {  
 // Resource pattern matching available.  
 try {  
 Resource[] resources = ((ResourcePatternResolver) resourceLoader).getResources(location);  
 int loadCount = loadBeanDefinitions(resources);  
 if (actualResources != null) {  
 for (Resource resource : resources) {  
 actualResources.add(resource);  
 }  
 }  
 if (logger.isDebugEnabled()) {  
 logger.debug("Loaded " + loadCount + " bean definitions from location pattern [" + location + "]");  
 }  
 return loadCount;  
 }  
 catch (IOException ex) {  
 throw new BeanDefinitionStoreException(  
 "Could not resolve bean definition resource pattern [" + location + "]", ex);  
 }  
 }  
 else {  
 // Can only load single resources by absolute URL.  
 Resource resource = resourceLoader.getResource(location);  
 int loadCount = loadBeanDefinitions(resource);  
 if (actualResources != null) {  
 actualResources.add(resource);  
 }  
 if (logger.isDebugEnabled()) {  
 logger.debug("Loaded " + loadCount + " bean definitions from location [" + location + "]");  
 }  
 return loadCount;  
 }  
}

public int loadBeanDefinitions(Resource... resources) throws BeanDefinitionStoreException {  
 Assert.notNull(resources, "Resource array must not be null");  
 int counter = 0;  
 for (Resource resource : resources) {  
 counter += loadBeanDefinitions(resource);  
 }  
 return counter;  
}

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

public int loadBeanDefinitions(EncodedResource encodedResource) throws BeanDefinitionStoreException {  
 Assert.notNull(encodedResource, "EncodedResource must not be null");  
 if (logger.isInfoEnabled()) {  
 logger.info("Loading XML bean definitions from " + encodedResource.getResource());  
 }  
  
 Set<EncodedResource> currentResources = this.resourcesCurrentlyBeingLoaded.get();  
 if (currentResources == null) {  
 currentResources = new HashSet<EncodedResource>(4);  
 this.resourcesCurrentlyBeingLoaded.set(currentResources);  
 }  
 if (!currentResources.add(encodedResource)) {  
 throw new BeanDefinitionStoreException(  
 "Detected cyclic loading of " + encodedResource + " - check your import definitions!");  
 }  
 try {  
 InputStream inputStream = encodedResource.getResource().getInputStream();  
 try {  
 InputSource inputSource = new InputSource(inputStream);  
 if (encodedResource.getEncoding() != null) {  
 inputSource.setEncoding(encodedResource.getEncoding());  
 }  
 return doLoadBeanDefinitions(inputSource, encodedResource.getResource());  
 }  
 finally {  
 inputStream.close();  
 }  
 }  
 catch (IOException ex) {  
 throw new BeanDefinitionStoreException(  
 "IOException parsing XML document from " + encodedResource.getResource(), ex);  
 }  
 finally {  
 currentResources.remove(encodedResource);  
 if (currentResources.isEmpty()) {  
 this.resourcesCurrentlyBeingLoaded.remove();  
 }  
 }  
}

* doLoadBeanDefinitions
* 继续向下找，最终调用了reader的doLoadBeanDefinitions方法：
* protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)  
   throws BeanDefinitionStoreException {  
   try {  
   Document doc = doLoadDocument(inputSource, resource);  
   return registerBeanDefinitions(doc, resource);  
   }  
   catch (BeanDefinitionStoreException ex) {  
   throw ex;  
   }
* ...
* }
* doLoadDocument方法中用到了documentLoader对象加载document，它最后又会用到domParser解析xml文件。
* 最终是使用jaxp的dom方式读取的XML配置文件。（JAXP是一种标准，SUN公司对其提供了实现)
* registerBeanDefinitions
* 将XML对应的Document对象转为BeanDefinitions：

public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreException {  
 BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();  
 int countBefore = getRegistry().getBeanDefinitionCount();  
 documentReader.registerBeanDefinitions(doc, createReaderContext(resource));  
 return getRegistry().getBeanDefinitionCount() - countBefore;  
}

* documentReader#registerBeanDefinitions
* 这是一个抽象方法，在DefaultBeanDefinitionDocumentReader类中得到实现：

public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {  
 this.readerContext = readerContext;  
 logger.debug("Loading bean definitions");  
 Element root = doc.getDocumentElement();  
 doRegisterBeanDefinitions(root);  
}

* doRegisterBeanDefinitions
* 实际解析文档的是doRegisterBeanDefinitions方法：
* protected void doRegisterBeanDefinitions(Element root) {  
  // Any nested elements will cause recursion in this method. In  
  // order to propagate and preserve default-\* attributes correctly,  
  // keep track of the current (parent) delegate, which may be null. Create  
  // the new (child) delegate with a reference to the parent for fallback purposes,  
  // then ultimately reset this.delegate back to its original (parent) reference.  
  // this behavior emulates a stack of delegates without actually necessitating one.  
  BeanDefinitionParserDelegate parent = this.delegate;  
  this.delegate = createDelegate(getReaderContext(), root, parent);
* if (this.delegate.isDefaultNamespace(root)) {  
   String profileSpec = root.getAttribute(PROFILE*ATTRIBUTE);*  
   *if (StringUtils.hasText(profileSpec)) {*  
   *String[] specifiedProfiles = StringUtils.tokenizeToStringArray(*  
   *profileSpec, BeanDefinitionParserDelegate.MULTI*VALUE*ATTRIBUTE*DELIMITERS);  
   if (!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles)) {  
   if (logger.isInfoEnabled()) {  
   logger.info("Skipped XML bean definition file due to specified profiles [" + profileSpec +  
   "] not matching: " + getReaderContext().getResource());  
   }  
   return;  
   }  
   }  
  }
* preProcessXml(root);  
  parseBeanDefinitions(root, this.delegate);  
  postProcessXml(root);
* this.delegate = parent;  
  }
* parseBeanDefinitions（解析标签)
* 实际解析方法parseBeanDefinitions：
* protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {  
  if (delegate.isDefaultNamespace(root)) {  
   NodeList nl = root.getChildNodes();  
   for (int i = 0; i < nl.getLength(); i++) {  
   Node node = nl.item(i);  
   if (node instanceof Element) {  
   Element ele = (Element) node;  
   if (delegate.isDefaultNamespace(ele)) {  
   parseDefaultElement(ele, delegate);  
   }  
   else {  
   delegate.parseCustomElement(ele);  
   }  
   }  
   }  
  }  
  else {  
   delegate.parseCustomElement(root);  
  }  
  }
* 分开解析默认标签和自定义标签。
* parseDefaultElement（解析默认标签)
* 就import、alias、bean和beans四种默认标签进行解析

private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {  
 if (delegate.nodeNameEquals(ele, IMPORT\_ELEMENT)) {  
 importBeanDefinitionResource(ele);  
 }  
 else if (delegate.nodeNameEquals(ele, ALIAS\_ELEMENT)) {  
 processAliasRegistration(ele);  
 }  
 else if (delegate.nodeNameEquals(ele, BEAN\_ELEMENT)) {  
 processBeanDefinition(ele, delegate);  
 }  
 else if (delegate.nodeNameEquals(ele, NESTED\_BEANS\_ELEMENT)) {  
 // recurse  
 doRegisterBeanDefinitions(ele);  
 }  
}

* parseCustomElement（解析自定义标签)
* 调用的是parseCustomElement(ele,null)

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

* NamespaceHandlerSupport#parse

public BeanDefinition parse(Element element, ParserContext parserContext) {  
 return findParserForElement(element, parserContext).parse(element, parserContext);  
}

* a) findParserForElement

private BeanDefinitionParser findParserForElement(Element element, ParserContext parserContext) {  
 String localName = parserContext.getDelegate().getLocalName(element);  
 BeanDefinitionParser parser = this.parsers.get(localName);  
 if (parser == null) {  
 parserContext.getReaderContext().fatal(  
 "Cannot locate BeanDefinitionParser for element [" + localName + "]", element);  
 }  
 return parser;  
}

* parser是ComponentScanBeanDefinitionParser类型。
* b) ComponentScanBeanDefinitionParser#parse

public BeanDefinition parse(Element element, ParserContext parserContext) {  
 String basePackage = element.getAttribute(BASE\_PACKAGE\_ATTRIBUTE);  
 basePackage = parserContext.getReaderContext().getEnvironment().resolvePlaceholders(basePackage);  
 String[] basePackages = StringUtils.tokenizeToStringArray(basePackage,  
 ConfigurableApplicationContext.CONFIG\_LOCATION\_DELIMITERS);  
  
 // Actually scan for bean definitions and register them.  
 ClassPathBeanDefinitionScanner scanner = configureScanner(parserContext, element);  
 Set<BeanDefinitionHolder> beanDefinitions = scanner.doScan(basePackages);  
 registerComponents(parserContext.getReaderContext(), beanDefinitions, element);  
  
 return null;  
}

- 1) ClassPathBeanDefinitionScanner.doScan（解析注解定义的bean)

* 根据自定义标签component-scan来扫描包，批量注册bean。
* protected Set doScan(String... basePackages) {  
  Assert.notEmpty(basePackages, "At least one base package must be specified");  
  Set beanDefinitions = new LinkedHashSet();  
  for (String basePackage : basePackages) {  
   Set candidates = findCandidateComponents(basePackage);  
   for (BeanDefinition candidate : candidates) {  
   ScopeMetadata scopeMetadata = this.scopeMetadataResolver.resolveScopeMetadata(candidate);  
   candidate.setScope(scopeMetadata.getScopeName());  
   String beanName = this.beanNameGenerator.generateBeanName(candidate, this.registry);  
   if (candidate instanceof AbstractBeanDefinition) {  
   postProcessBeanDefinition((AbstractBeanDefinition) candidate, beanName);  
   }  
   if (candidate instanceof AnnotatedBeanDefinition) {  
   AnnotationConfigUtils.processCommonDefinitionAnnotations((AnnotatedBeanDefinition) candidate);  
   }  
   if (checkCandidate(beanName, candidate)) {  
   BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(candidate, beanName);  
   definitionHolder =  
   AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder, this.registry);  
   beanDefinitions.add(definitionHolder);  
   registerBeanDefinition(definitionHolder, this.registry);  
   }  
   }  
  }  
  return beanDefinitions;  
  }
  + 1.1) findCandidateComponents

public Set<BeanDefinition> findCandidateComponents(String basePackage) {  
 Set<BeanDefinition> candidates = new LinkedHashSet<BeanDefinition>();  
 try {  
 String packageSearchPath = ResourcePatternResolver.CLASSPATH\_ALL\_URL\_PREFIX +  
 resolveBasePackage(basePackage) + '/' + this.resourcePattern;

* // 这里是未加筛选的拿到了所有class文件  
   Resource[] resources = this.resourcePatternResolver.getResources(packageSearchPath);  
   boolean traceEnabled = logger.isTraceEnabled();  
   boolean debugEnabled = logger.isDebugEnabled();  
   for (Resource resource : resources) {  
   if (traceEnabled) {  
   logger.trace("Scanning " + resource);  
   }  
   if (resource.isReadable()) {  
   try {
* // 这里会把class文件读进来  
   MetadataReader metadataReader = this.metadataReaderFactory.getMetadataReader(resource);
* // 判断class文件是否是注册在Spring中的bean类型  
   if (isCandidateComponent(metadataReader)) {  
   ScannedGenericBeanDefinition sbd = new ScannedGenericBeanDefinition(metadataReader);  
   sbd.setResource(resource);  
   sbd.setSource(resource);  
   if (isCandidateComponent(sbd)) {  
   if (debugEnabled) {  
   logger.debug("Identified candidate component class: " + resource);  
   }  
   candidates.add(sbd);  
   }  
   else {  
   if (debugEnabled) {  
   logger.debug("Ignored because not a concrete top-level class: " + resource);  
   }  
   }  
   }  
   else {  
   if (traceEnabled) {  
   logger.trace("Ignored because not matching any filter: " + resource);  
   }  
   }  
   }  
   catch (Throwable ex) {  
   throw new BeanDefinitionStoreException(  
   "Failed to read candidate component class: " + resource, ex);  
   }  
  }  
  else {  
   if (traceEnabled) {  
   logger.trace("Ignored because not readable: " + resource);  
   }  
  }  
  }  
  }  
  catch (IOException ex) {  
   throw new BeanDefinitionStoreException("I/O failure during classpath scanning", ex);  
  }  
  return candidates;  
  }
  + 1.1.1) PathMatchingResourcePatternResolver.getResources

public Resource[] getResources(String locationPattern) throws IOException {  
 Assert.notNull(locationPattern, "Location pattern must not be null");  
 if (locationPattern.startsWith(CLASSPATH\_ALL\_URL\_PREFIX)) {  
 // a class path resource (multiple resources for same name possible)  
 if (getPathMatcher().isPattern(locationPattern.substring(CLASSPATH\_ALL\_URL\_PREFIX.length()))) {  
 // a class path resource pattern  
 return findPathMatchingResources(locationPattern);  
 }  
 else {  
 // all class path resources with the given name  
 return findAllClassPathResources(locationPattern.substring(CLASSPATH\_ALL\_URL\_PREFIX.length()));  
 }  
 }  
 else {  
 // Only look for a pattern after a prefix here  
 // (to not get fooled by a pattern symbol in a strange prefix).  
 int prefixEnd = locationPattern.indexOf(":") + 1;  
 if (getPathMatcher().isPattern(locationPattern.substring(prefixEnd))) {  
 // a file pattern  
 return findPathMatchingResources(locationPattern);  
 }  
 else {  
 // a single resource with the given name  
 return new Resource[] {getResourceLoader().getResource(locationPattern)};  
 }  
 }  
}

* + 1.1.1.1) findPathMatchingResources
* protected Resource[] findPathMatchingResources(String locationPattern) throws IOException {  
  String rootDirPath = determineRootDir(locationPattern);  
  String subPattern = locationPattern.substring(rootDirPath.length());
  + // 根路径，component-scan中配置的包名  
    Resource[] rootDirResources = getResources(rootDirPath);  
    Set result = new LinkedHashSet(16);  
    for (Resource rootDirResource : rootDirResources) {  
    rootDirResource = resolveRootDirResource(rootDirResource);  
    URL rootDirURL = rootDirResource.getURL();  
    if (equinoxResolveMethod != null) {  
     if (rootDirURL.getProtocol().startsWith("bundle")) {  
     rootDirURL = (URL) ReflectionUtils.invokeMethod(equinoxResolveMethod, null, rootDirURL);  
     rootDirResource = new UrlResource(rootDirURL);  
     }  
    }  
    if (rootDirURL.getProtocol().startsWith(ResourceUtils.URL*PROTOCOL*VFS)) {  
     result.addAll(VfsResourceMatchingDelegate.findMatchingResources(rootDirURL, subPattern, getPathMatcher()));  
    }  
    else if (ResourceUtils.isJarURL(rootDirURL) || isJarResource(rootDirResource)) {  
     result.addAll(doFindPathMatchingJarResources(rootDirResource, rootDirURL, subPattern));  
    }  
    else {  
     result.addAll(doFindPathMatchingFileResources(rootDirResource, subPattern));  
    }  
    }  
    if (logger.isDebugEnabled()) {  
    logger.debug("Resolved location pattern [" + locationPattern + "] to resources " + result);  
    }  
    return result.toArray(new Resource[result.size()]);  
    }
  + 1.1.1.1.1) doFindPathMatchingFileResources
* protected Set doFindPathMatchingFileResources(Resource rootDirResource, String subPattern)  
   throws IOException {
* File rootDir;  
   try {  
   rootDir = rootDirResource.getFile().getAbsoluteFile();  
   }  
   catch (IOException ex) {  
   if (logger.isWarnEnabled()) {  
   logger.warn("Cannot search for matching files underneath " + rootDirResource +  
   " because it does not correspond to a directory in the file system", ex);  
   }  
   return Collections.emptySet();  
   }  
   return doFindMatchingFileSystemResources(rootDir, subPattern);  
  }
* doFindMatchingFileSystemResources
* protected Set doFindMatchingFileSystemResources(File rootDir, String subPattern) throws IOException {  
  if (logger.isDebugEnabled()) {  
   logger.debug("Looking for matching resources in directory tree [" + rootDir.getPath() + "]");  
  }  
  Set matchingFiles = retrieveMatchingFiles(rootDir, subPattern);  
  Set result = new LinkedHashSet(matchingFiles.size());  
  for (File file : matchingFiles) {  
   result.add(new FileSystemResource(file));  
  }  
  return result;  
  }
  + 1.1.1.1.1.1) retrieveMatchingFiles
  + protected Set retrieveMatchingFiles(File rootDir, String pattern) throws IOException {  
    if (!rootDir.exists()) {  
    // Silently skip non-existing directories.  
    if (logger.isDebugEnabled()) {  
     logger.debug("Skipping [" + rootDir.getAbsolutePath() + "] because it does not exist");  
    }  
    return Collections.emptySet();  
    }  
    if (!rootDir.isDirectory()) {  
    // Complain louder if it exists but is no directory.  
    if (logger.isWarnEnabled()) {  
     logger.warn("Skipping [" + rootDir.getAbsolutePath() + "] because it does not denote a directory");  
    }  
    return Collections.emptySet();  
    }  
    if (!rootDir.canRead()) {  
    if (logger.isWarnEnabled()) {  
     logger.warn("Cannot search for matching files underneath directory [" + rootDir.getAbsolutePath() +  
     "] because the application is not allowed to read the directory");  
    }  
    return Collections.emptySet();  
    }  
    String fullPattern = StringUtils.replace(rootDir.getAbsolutePath(), File.separator, "/");  
    if (!pattern.startsWith("/")) {  
    fullPattern += "/";  
    }  
    fullPattern = fullPattern + StringUtils.replace(pattern, File.separator, "/");  
    Set result = new LinkedHashSet(8);  
    doRetrieveMatchingFiles(fullPattern, rootDir, result);  
    return result;  
    }
  + 1.1.1.1.1.1.1) doRetrieveMatchingFiles（递归方法)
* protected void doRetrieveMatchingFiles(String fullPattern, File dir, Set result) throws IOException {  
  if (logger.isDebugEnabled()) {  
   logger.debug("Searching directory [" + dir.getAbsolutePath() +  
   "] for files matching pattern [" + fullPattern + "]");  
  }
* // 拿到component-scan目录下的所有class文件  
  File[] dirContents = dir.listFiles();  
  if (dirContents == null) {  
   if (logger.isWarnEnabled()) {  
   logger.warn("Could not retrieve contents of directory [" + dir.getAbsolutePath() + "]");  
   }  
   return;  
  }  
  Arrays.sort(dirContents);  
  for (File content : dirContents) {  
   String currPath = StringUtils.replace(content.getAbsolutePath(), File.separator, "/");  
   if (content.isDirectory() && getPathMatcher().matchStart(fullPattern, currPath + "/")) {  
   if (!content.canRead()) {  
   if (logger.isDebugEnabled()) {  
   logger.debug("Skipping subdirectory [" + dir.getAbsolutePath() +  
   "] because the application is not allowed to read the directory");  
   }  
   }  
   else {  
   doRetrieveMatchingFiles(fullPattern, content, result);  
   }  
   }  
   if (getPathMatcher().match(fullPattern, currPath)) {  
   result.add(content);  
   }  
  }  
  }
  + 1.1.2) CachingMetadataResourceFactory.getMetadataReader（读取class文件)

public MetadataReader getMetadataReader(Resource resource) throws IOException {  
 if (getCacheLimit() <= 0) {  
 return super.getMetadataReader(resource);  
 }  
 synchronized (this.metadataReaderCache) {  
 MetadataReader metadataReader = this.metadataReaderCache.get(resource);  
 if (metadataReader == null) {  
 metadataReader = super.getMetadataReader(resource);  
 this.metadataReaderCache.put(resource, metadataReader);  
 }  
 return metadataReader;  
 }  
}

- 1.1.2.1) SimpleMetadataReader.getMetadataReader

* 该类封装了Class文件中的各种信息，保存在ClassMatadata和AnnotationMetadata中。

final class SimpleMetadataReader implements MetadataReader {  
  
 private final Resource resource;  
  
 private final ClassMetadata classMetadata;  
  
 private final AnnotationMetadata annotationMetadata;

* }

public MetadataReader getMetadataReader(Resource resource) throws IOException {  
 return new SimpleMetadataReader(resource, this.resourceLoader.getClassLoader());  
}

* SimpleMetadataReader(Resource resource, ClassLoader classLoader) throws IOException {  
  InputStream is = new BufferedInputStream(resource.getInputStream());  
  ClassReader classReader;  
  try {  
   classReader = new ClassReader(is);  
  }  
  catch (IllegalArgumentException ex) {  
   throw new NestedIOException("ASM ClassReader failed to parse class file - " +  
   "probably due to a new Java class file version that isn't supported yet: " + resource, ex);  
  }  
  finally {  
   is.close();  
  }
* AnnotationMetadataReadingVisitor visitor = new AnnotationMetadataReadingVisitor(classLoader);  
  classReader.accept(visitor, ClassReader.SKIP\_DEBUG);
* this.annotationMetadata = visitor;  
  // (since AnnotationMetadataReadingVisitor extends ClassMetadataReadingVisitor)  
  this.classMetadata = visitor;  
  this.resource = resource;  
  }
  + 1.1.2.1.1) ClassReader.accept
* 源码比较奇怪，大概是按照class文件的格式解析，并将结果封装到visitor里面。
  + 1.1.3) isCandidateComponent
* protected boolean isCandidateComponent(MetadataReader metadataReader) throws IOException {  
  for (TypeFilter tf : this.excludeFilters) {  
   if (tf.match(metadataReader, this.metadataReaderFactory)) {  
   return false;  
   }  
  }  
  for (TypeFilter tf : this.includeFilters) {  
   if (tf.match(metadataReader, this.metadataReaderFactory)) {  
   return isConditionMatch(metadataReader);  
   }  
  }  
  return false;  
  }
* 这里的includeFilters有一个AnnotationTypeFilter。
  + 1.1.3.1) AbstractTypeHierarchyTraversingFilter.match

public boolean match(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory)  
 throws IOException {  
  
 // This method optimizes avoiding unnecessary creation of ClassReaders  
 // as well as visiting over those readers.  
 if (matchSelf(metadataReader)) {  
 return true;  
 }  
 ClassMetadata metadata = metadataReader.getClassMetadata();  
 if (matchClassName(metadata.getClassName())) {  
 return true;  
 }  
  
 if (this.considerInherited) {  
 if (metadata.hasSuperClass()) {  
 // Optimization to avoid creating ClassReader for super class.  
 Boolean superClassMatch = matchSuperClass(metadata.getSuperClassName());  
 if (superClassMatch != null) {  
 if (superClassMatch.booleanValue()) {  
 return true;  
 }  
 }  
 else {  
 // Need to read super class to determine a match...  
 try {  
 if (match(metadata.getSuperClassName(), metadataReaderFactory)) {  
 return true;  
 }  
 }  
 catch (IOException ex) {  
 logger.debug("Could not read super class [" + metadata.getSuperClassName() +  
 "] of type-filtered class [" + metadata.getClassName() + "]");  
 }  
 }  
 }  
 }  
  
 if (this.considerInterfaces) {  
 for (String ifc : metadata.getInterfaceNames()) {  
 // Optimization to avoid creating ClassReader for super class  
 Boolean interfaceMatch = matchInterface(ifc);  
 if (interfaceMatch != null) {  
 if (interfaceMatch.booleanValue()) {  
 return true;  
 }  
 }  
 else {  
 // Need to read interface to determine a match...  
 try {  
 if (match(ifc, metadataReaderFactory)) {  
 return true;  
 }  
 }  
 catch (IOException ex) {  
 logger.debug("Could not read interface [" + ifc + "] for type-filtered class [" +  
 metadata.getClassName() + "]");  
 }  
 }  
 }  
 }  
  
 return false;  
}

* protected boolean matchSelf(MetadataReader metadataReader) {  
  AnnotationMetadata metadata = metadataReader.getAnnotationMetadata();  
  return metadata.hasAnnotation(this.annotationType.getName()) ||  
   (this.considerMetaAnnotations && metadata.hasMetaAnnotation(this.annotationType.getName()));  
  }
* this.annotationType是@Component类型，所以
* metadata.hasAnnotation(this.annotationType.getName())当类上注解了@Component时为true。
* 这里因为@Service等也注解了@Component了，所以@Service、@Controller等在这里都被视为@Component。

public boolean hasMetaAnnotation(String metaAnnotationType) {  
 Collection<Set<String>> allMetaTypes = this.metaAnnotationMap.values();  
 for (Set<String> metaTypes : allMetaTypes) {  
 if (metaTypes.contains(metaAnnotationType)) {  
 return true;  
 }  
 }  
 return false;  
}

- 1.2) registerBeanDefinition（将beanDefinition记录到BeanFactory)

* protected void registerBeanDefinition(BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry registry) {  
  BeanDefinitionReaderUtils.registerBeanDefinition(definitionHolder, registry);  
  }
  + 1.2.1) DefaultListableBeanFactory.registerBeanDefinition（保存beanDefinition)

public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)  
 throws BeanDefinitionStoreException {  
  
 Assert.hasText(beanName, "Bean name must not be empty");  
 Assert.notNull(beanDefinition, "BeanDefinition must not be null");  
  
 if (beanDefinition instanceof AbstractBeanDefinition) {  
 try {  
 ((AbstractBeanDefinition) beanDefinition).validate();  
 }  
 catch (BeanDefinitionValidationException ex) {  
 throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,  
 "Validation of bean definition failed", ex);  
 }  
 }  
  
 BeanDefinition oldBeanDefinition;  
  
 oldBeanDefinition = this.beanDefinitionMap.get(beanName);  
 if (oldBeanDefinition != null) {  
 if (!isAllowBeanDefinitionOverriding()) {  
 throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,  
 "Cannot register bean definition [" + beanDefinition + "] for bean '" + beanName +  
 "': There is already [" + oldBeanDefinition + "] bound.");  
 }  
 else if (oldBeanDefinition.getRole() < beanDefinition.getRole()) {  
 // e.g. was ROLE\_APPLICATION, now overriding with ROLE\_SUPPORT or ROLE\_INFRASTRUCTURE  
 if (this.logger.isWarnEnabled()) {  
 this.logger.warn("Overriding user-defined bean definition for bean '" + beanName +  
 "' with a framework-generated bean definition: replacing [" +  
 oldBeanDefinition + "] with [" + beanDefinition + "]");  
 }  
 }  
 else if (!beanDefinition.equals(oldBeanDefinition)) {  
 if (this.logger.isInfoEnabled()) {  
 this.logger.info("Overriding bean definition for bean '" + beanName +  
 "' with a different definition: replacing [" + oldBeanDefinition +  
 "] with [" + beanDefinition + "]");  
 }  
 }  
 else {  
 if (this.logger.isDebugEnabled()) {  
 this.logger.debug("Overriding bean definition for bean '" + beanName +  
 "' with an equivalent definition: replacing [" + oldBeanDefinition +  
 "] with [" + beanDefinition + "]");  
 }  
 }  
 this.beanDefinitionMap.put(beanName, beanDefinition);  
 }  
 else {  
 if (hasBeanCreationStarted()) {  
 // Cannot modify startup-time collection elements anymore (for stable iteration)  
 synchronized (this.beanDefinitionMap) {  
 this.beanDefinitionMap.put(beanName, beanDefinition);  
 List<String> updatedDefinitions = new ArrayList<String>(this.beanDefinitionNames.size() + 1);  
 updatedDefinitions.addAll(this.beanDefinitionNames);  
 updatedDefinitions.add(beanName);  
 this.beanDefinitionNames = updatedDefinitions;  
 if (this.manualSingletonNames.contains(beanName)) {  
 Set<String> updatedSingletons = new LinkedHashSet<String>(this.manualSingletonNames);  
 updatedSingletons.remove(beanName);  
 this.manualSingletonNames = updatedSingletons;  
 }  
 }  
 }  
 else {  
 // Still in startup registration phase  
 this.beanDefinitionMap.put(beanName, beanDefinition);  
 this.beanDefinitionNames.add(beanName);  
 this.manualSingletonNames.remove(beanName);  
 }  
 this.frozenBeanDefinitionNames = null;  
 }  
  
 if (oldBeanDefinition != null || containsSingleton(beanName)) {  
 resetBeanDefinition(beanName);  
 }  
}

* 标签解析完毕后会将beanName和beanDefinition作为key和value放入beanfactory的beanDefinitionMap中。
  + 2) finishBeanFactoryInitialization（初始化非lazy-load且singleton的bean)

protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {  
 // Initialize conversion service for this context.  
 if (beanFactory.containsBean(CONVERSION\_SERVICE\_BEAN\_NAME) &&  
 beanFactory.isTypeMatch(CONVERSION\_SERVICE\_BEAN\_NAME, ConversionService.class)) {  
 beanFactory.setConversionService(  
 beanFactory.getBean(CONVERSION\_SERVICE\_BEAN\_NAME, ConversionService.class));  
 }  
  
 // Register a default embedded value resolver if no bean post-processor  
 // (such as a PropertyPlaceholderConfigurer bean) registered any before:  
 // at this point, primarily for resolution in annotation attribute values.  
 if (!beanFactory.hasEmbeddedValueResolver()) {  
 beanFactory.addEmbeddedValueResolver(new StringValueResolver() {  
 @Override  
 public String resolveStringValue(String strVal) {  
 return getEnvironment().resolvePlaceholders(strVal);  
 }  
 });  
 }  
  
 // Initialize LoadTimeWeaverAware beans early to allow for registering their transformers early.  
 String[] weaverAwareNames = beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, false);  
 for (String weaverAwareName : weaverAwareNames) {  
 getBean(weaverAwareName);  
 }  
  
 // Stop using the temporary ClassLoader for type matching.  
 beanFactory.setTempClassLoader(null);  
  
 // Allow for caching all bean definition metadata, not expecting further changes.  
 beanFactory.freezeConfiguration();  
  
 // Instantiate all remaining (non-lazy-init) singletons.  
 beanFactory.preInstantiateSingletons();  
}

- 2.1) ConfigurableListableBeanFactory#preInstantiateSingletons

* DefaultListableBeanFactory.preInstantiateSingletons

public void preInstantiateSingletons() throws BeansException {  
 if (this.logger.isDebugEnabled()) {  
 this.logger.debug("Pre-instantiating singletons in " + this);  
 }  
  
 // Iterate over a copy to allow for init methods which in turn register new bean definitions.  
 // While this may not be part of the regular factory bootstrap, it does otherwise work fine.  
 List<String> beanNames = new ArrayList<String>(this.beanDefinitionNames);  
  
 // Trigger initialization of all non-lazy singleton beans...  
 for (String beanName : beanNames) {  
 RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);  
 if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {  
 if (isFactoryBean(beanName)) {  
 final FactoryBean<?> factory = (FactoryBean<?>) getBean(FACTORY\_BEAN\_PREFIX + beanName);  
 boolean isEagerInit;  
 if (System.getSecurityManager() != null && factory instanceof SmartFactoryBean) {  
 isEagerInit = AccessController.doPrivileged(new PrivilegedAction<Boolean>() {  
 @Override  
 public Boolean run() {  
 return ((SmartFactoryBean<?>) factory).isEagerInit();  
 }  
 }, getAccessControlContext());  
 }  
 else {  
 isEagerInit = (factory instanceof SmartFactoryBean &&  
 ((SmartFactoryBean<?>) factory).isEagerInit());  
 }  
 if (isEagerInit) {  
 getBean(beanName);  
 }  
 }  
 else {  
 getBean(beanName);  
 }  
 }  
 }  
  
 // Trigger post-initialization callback for all applicable beans...  
 for (String beanName : beanNames) {  
 Object singletonInstance = getSingleton(beanName);  
 if (singletonInstance instanceof SmartInitializingSingleton) {  
 final SmartInitializingSingleton smartSingleton = (SmartInitializingSingleton) singletonInstance;  
 if (System.getSecurityManager() != null) {  
 AccessController.doPrivileged(new PrivilegedAction<Object>() {  
 @Override  
 public Object run() {  
 smartSingleton.afterSingletonsInstantiated();  
 return null;  
 }  
 }, getAccessControlContext());  
 }  
 else {  
 smartSingleton.afterSingletonsInstantiated();  
 }  
 }  
 }  
}

* Bean的加载
* FactoryBean（工厂Bean，用户定制)
* Spring通过反射机制利用bean的class属性指定实现类来实例化bean。
* Spring提供了一个FactoryBean的工厂类接口，用户可以通过实现该接口定制实例化bean的逻辑。
* FactoryBean

public interface FactoryBean<T> {

* // 返回bean实例，如果isSingleton()返回true，那么该实例会放到Spring容器中单例缓存池中  
  T getObject() throws Exception;  
  Class<?> getObjectType();  
  boolean isSingleton();  
  }
* FactoryBean：
* 这个接口使你可以提供一个复杂的逻辑来生成Bean。它本质是一个Bean，但这个Bean不是用来注入到其它地方像Service、Dao一样使用的，它是用来生成其它Bean使用的。实现了这个接口后，Spring在容器初始化时，把实现这个接口的Bean取出来，使用接口的getObject()方法来生成我们要想的Bean。当然，那些生成Bean的业务逻辑也要写getObject()方法中。
* 其返回的对象不是指定类的一个实例，其返回的是该工厂Bean的getObject方法所返回的对象。创建出来的对象是否属于单例由isSingleton中的返回决定。
* 使用场景：1、通过外部对类是否是单例进行控制，该类自己无法感知 2、在创建Object对象之前进行初始化的操作，在afterPropertiesSet()中完成。（一次性的初始化，保存在成员变量中，并不是每次getObject都会调用afterPropertiesSet，afterPropertiesSet只会被调用一次)
* 实例：

public class CarFactoryBean implements FactoryBean<Car> {

private String brand;

private double price;

* @Override

public Car getObject() throws Exception {

* return new Car(brand,price);
* }
* @Override

public Class<?> getObjectType() {

* return Car.class;
* }
* @Override

public boolean isSingleton() {

* return true;
* }

public String getBrand() {

* return brand;
* }

public void setBrand(String brand) {

* this.brand = brand;
* }

public double getPrice() {

* return price;
* }

public void setPrice(double price) {

* this.price = price;
* }
* }

public class Main {

public static void main(String[] args) {

* ApplicationContext ctx = new ClassPathXmlApplicationContext("beans-factoryBean.xml");
* Car car = (Car) ctx.getBean("car");
* System.out.println(car);
* }
* }
* ObjectFactory（Spring使用)

public interface ObjectFactory<T> {  
 T getObject() throws BeansException;  
}

* ObjectFactory：
* 它的目的也是作为一个工厂，来生成Object（这个接口只有一个方法getObject())。这个接口一般被用来，包装一个factory，通过个这工厂来返回一个新实例（prototype类型)。这个接口和FactoryBean有点像，但FactoryBean的实现是被当做一个SPI（Service Provider Interface)实例来使用在BeanFactory里面；ObjectFactory的实现一般被用来注入到其它Bean中，作为API来使用。就像ObjectFactoryCreatingFactoryBean的例子，它的返回值就是一个ObjectFactory，这个ObjectFactory被注入到了Bean中，在Bean通过这个接口的实例，来取得我们想要的Bean。
* 总的来说，FactoryBean和ObjectFactory都是用来取得Bean，但使用的方法和地方不同，FactoryBean被配置好后，Spring调用getObject()方法来取得Bean，ObjectFactory配置好后，在Bean里面可以取得ObjectFactory实例，需要我们手动来调用getObject()来取得Bean。

## InitializingBean

* InitializingBean接口为bean提供了初始化方法的方式，它只包括afterPropertiesSet方法，凡是继承该接口的类，在初始化bean的时候会执行该方法。
* AbstractBeanFactory#getBean

public Object getBean(String name) throws BeansException {  
 return doGetBean(name, null, null, false);  
}

* doGetBean
* 有三个方法非常关键：getSingleton，createBean和getObjectForBeanInstance。
* protected T doGetBean(  
   final String name, final Class requiredType, final Object[] args, boolean typeCheckOnly)  
   throws BeansException {
* final String beanName = transformedBeanName(name);  
   Object bean;
* // Eagerly check singleton cache for manually registered singletons.
* // 检查缓存中或者实例工厂中是否有对应的实例（解决循环依赖的问题)
* // Spring创建bean的原则是不等bean创建完成就会将创建bean的ObjectFactory提早曝光，也就是将ObjectFactory加入到缓存中，一旦下个bean创建时需要上个bean则直接使用ObjectFactory。
* // 直接尝试从缓存获取或者从singletonFactories中的ObjectFactory中获取  
  Object sharedInstance = getSingleton(beanName);  
  if (sharedInstance != null && args == null) {
* // 已经创建过了  
   if (logger.isDebugEnabled()) {  
   if (isSingletonCurrentlyInCreation(beanName)) {  
   logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +  
   "' that is not fully initialized yet - a consequence of a circular reference");  
   }  
   else {  
   logger.debug("Returning cached instance of singleton bean '" + beanName + "'");  
   }  
   }
* // 返回对应的实例（从缓存中只得到了bean的原始状态，还需要对bean进行实例化)  
   bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);  
  }  
  else {
* // 没有创建，需要创建
* // Fail if we're already creating this bean instance:  
   // We're assumably within a circular reference.  
   if (isPrototypeCurrentlyInCreation(beanName)) {  
   throw new BeanCurrentlyInCreationException(beanName);  
   }
* // Check if bean definition exists in this factory.  
  BeanFactory parentBeanFactory = getParentBeanFactory();
* // 如果beanDefinitionMap（已经加载了的类)中不包含beanName，则尝试从parentBeanFactory处理  
   if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {  
   // Not found -> check parent.  
   String nameToLookup = originalBeanName(name);  
   if (args != null) {  
   // Delegation to parent with explicit args.
* // 递归  
   return (T) parentBeanFactory.getBean(nameToLookup, args);  
   }  
   else {  
   // No args -> delegate to standard getBean method.  
   return parentBeanFactory.getBean(nameToLookup, requiredType);  
   }  
  }  
  // 从这里开始创建bean，先进行记录  
  if (!typeCheckOnly) {  
   markBeanAsCreated(beanName);  
  }
* try {
* // 将存储XML配置文件的GenericBeanDefinition转换为RootBeanDefinition；转换的时候如果父类bean不为空的话，那么会合并父类的属性。  
   final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);  
   checkMergedBeanDefinition(mbd, beanName, args);
* // Guarantee initialization of beans that the current bean depends on. // 若存在依赖则需要递归实例化依赖的bean  
   String[] dependsOn = mbd.getDependsOn();  
   if (dependsOn != null) {  
   for (String dep : dependsOn) {  
   if (isDependent(beanName, dep)) {  
   throw new BeanCreationException(mbd.getResourceDescription(), beanName,  
   "Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");  
   }
* // 缓存依赖调用  
   registerDependentBean(dep, beanName);  
   getBean(dep);  
   }  
  }
* // Create bean instance.
* // 真正的创建bean  
   if (mbd.isSingleton()) {

// 单例  
 sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {  
 @Override  
 public Object getObject() throws BeansException {  
 try {  
 return createBean(beanName, mbd, args);  
 }  
 catch (BeansException ex) {  
 // Explicitly remove instance from singleton cache: It might have been put there  
 // eagerly by the creation process, to allow for circular reference resolution.  
 // Also remove any beans that received a temporary reference to the bean.  
 destroySingleton(beanName);  
 throw ex;  
 }  
 }  
 });  
 bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);  
 }  
  
 else if (mbd.isPrototype()) {  
 // It's a prototype -> create a new instance.  
 Object prototypeInstance = null;  
 try {  
 beforePrototypeCreation(beanName);  
 prototypeInstance = createBean(beanName, mbd, args);  
 }  
 finally {  
 afterPrototypeCreation(beanName);  
 }  
 bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);  
 }  
 else {

// 在指定的scope上实例化bean  
 String scopeName = mbd.getScope();  
 final Scope scope = this.scopes.get(scopeName);  
 if (scope == null) {  
 throw new IllegalStateException("No Scope registered for scope name '" + scopeName + "'");  
 }  
 try {  
 Object scopedInstance = scope.get(beanName, new ObjectFactory<Object>() {  
 @Override  
 public Object getObject() throws BeansException {  
 beforePrototypeCreation(beanName);  
 try {  
 return createBean(beanName, mbd, args);  
 }  
 finally {  
 afterPrototypeCreation(beanName);  
 }  
 }  
 });  
 bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);  
 }  
 catch (IllegalStateException ex) {  
 throw new BeanCreationException(beanName,  
 "Scope '" + scopeName + "' is not active for the current thread; consider " +  
 "defining a scoped proxy for this bean if you intend to refer to it from a singleton",  
 ex);  
 }  
 }  
 }  
 catch (BeansException ex) {  
 cleanupAfterBeanCreationFailure(beanName);  
 throw ex;  
 }  
 }  
  
 // Check if required type matches the type of the actual bean instance.  
 if (requiredType != null && bean != null && !requiredType.isAssignableFrom(bean.getClass())) {  
 try {  
 return getTypeConverter().convertIfNecessary(bean, requiredType);  
 }  
 catch (TypeMismatchException ex) {  
 if (logger.isDebugEnabled()) {  
 logger.debug("Failed to convert bean '" + name + "' to required type '" +  
 ClassUtils.getQualifiedName(requiredType) + "'", ex);  
 }  
 throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());  
 }  
 }  
 return (T) bean;  
}

- 1) getSingleton(beanName)（借助缓存或singletonFactories)

* getSingleton(beanName,true) - > true表示允许早期依赖
* 逻辑：
  + 1)从singletonObjects中获取，它是一个真正的缓存，有就直接返回
  + 2)获取不到再从earlySingletonObjects里面获取
  + 3)还是获取不到，再尝试从singletonFactories里面获取beanName对应的ObjectFactory，然后调用这个ObjectFactory的getObject来获取之前创建的bean，并放到earlySingletonObjects里面去，并且从singletonFactories中remove掉这个ObjectFactory。
* 成员变量Map：
  + 1)singletonObjects是beanName与beanInstance的Map，是真正的缓存，beanInstance是构造完毕的，凡是正常地构造完毕的单例bean都会放入缓存中。
  + 2)earlySingletonObjects也是beanName与beanInstance的Map，beanInstance是已经调用了createBean方法，但是没有清除加载状态和加入至缓存的bean。仅在当前bean创建时存在，用于检测代理bean循环依赖。
  + 3)singleFactories是beanName与ObjectFactory的Map，仅在当前bean创建时存在，是尚未调用createBean的bean。用于setter循环依赖时实现注入。
  + 4)registeredSingletons：用来保存当前所有已注册的bean。
* singletonFactories和earlySingletonObjects都是一个临时工。在所有的对象创建完毕之后，此两个对象的size都为0。
* protected Object getSingleton(String beanName, boolean allowEarlyReference) {  
  Object singletonObject = this.singletonObjects.get(beanName);  
  if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {  
   synchronized (this.singletonObjects) {  
   singletonObject = this.earlySingletonObjects.get(beanName);  
   if (singletonObject == null && allowEarlyReference) {  
   ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);  
   if (singletonFactory != null) {  
   singletonObject = singletonFactory.getObject();  
   this.earlySingletonObjects.put(beanName, singletonObject);  
   this.singletonFactories.remove(beanName);  
   }  
   }  
   }  
  }  
  return (singletonObject != NULL\_OBJECT ? singletonObject : null);  
  }

- 2) getSingleton(beanName,ObjectFactory)（从头创建单例bean)

* 从头创建一个单例的bean需要经过getSingleton(beanName,ObjectFactory)和createBean两个关键方法。
* 逻辑：
  + 1)检查缓存是否已经加载过
  + 2)若没有加载，则记录beanName的正在加载状态
  + 3)加载单例前 记录加载状态
  + 4)通过ObjectFactory的getObject方法实例化bean
  + 5)加载单例后 清除加载状态
  + 6)将结果记录至缓存并删除加载bean过程中所记录的各种辅助状态
  + 7)返回处理结果

public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {  
 Assert.notNull(beanName, "'beanName' must not be null");  
 synchronized (this.singletonObjects) {

* // 检查对应的bean是否已经被加载过  
   Object singletonObject = this.singletonObjects.get(beanName);  
   if (singletonObject == null) {
* // 没有被加载过  
   if (this.singletonsCurrentlyInDestruction) {  
   throw new BeanCreationNotAllowedException(beanName,  
   "Singleton bean creation not allowed while singletons of this factory are in destruction " +  
   "(Do not request a bean from a BeanFactory in a destroy method implementation!)");  
   }  
   if (logger.isDebugEnabled()) {  
   logger.debug("Creating shared instance of singleton bean '" + beanName + "'");  
   }
* // 记录加载状态  
   beforeSingletonCreation(beanName);  
   boolean newSingleton = false;  
   boolean recordSuppressedExceptions = (this.suppressedExceptions == null);  
   if (recordSuppressedExceptions) {  
   this.suppressedExceptions = new LinkedHashSet();  
   }  
   try {
* // 初始化bean，在这里调用了createBean方法  
   singletonObject = singletonFactory.getObject();  
   newSingleton = true;  
   }  
   catch (IllegalStateException ex) {  
   // Has the singleton object implicitly appeared in the meantime ->  
   // if yes, proceed with it since the exception indicates that state.  
   singletonObject = this.singletonObjects.get(beanName);  
   if (singletonObject == null) {  
   throw ex;  
   }  
   }  
   catch (BeanCreationException ex) {  
   if (recordSuppressedExceptions) {  
   for (Exception suppressedException : this.suppressedExceptions) {  
   ex.addRelatedCause(suppressedException);  
   }  
   }  
   throw ex;  
   }  
   finally {  
   if (recordSuppressedExceptions) {  
   this.suppressedExceptions = null;  
   }
* // 清除加载状态  
   afterSingletonCreation(beanName);  
   }  
   if (newSingleton) {
* // 加入缓存  
   addSingleton(beanName, singletonObject);  
   }  
   }  
   return (singletonObject != NULL\_OBJECT ? singletonObject : null);  
  }  
  }
  + 2.1) beforeSingletonCreation （记录加载状态)
* 记录加载状态，通过this.singletonsCurrentlyInCreation.add(beanName)将当前正在创建的bean记录在缓存中，这样便可以对循环依赖进行检测。
* protected void beforeSingletonCreation(String beanName) {  
  if (!this.inCreationCheckExclusions.contains(beanName) && !this.singletonsCurrentlyInCreation.add(beanName)) {  
   throw new BeanCurrentlyInCreationException(beanName);  
  }  
  }
  + 2.2) afterSingletonCreation（清除加载状态)
* 当bean加载结束后需要移除缓存中对该bean的正在加载状态的记录。
* protected void afterSingletonCreation(String beanName) {  
  if (!this.inCreationCheckExclusions.contains(beanName) && !this.singletonsCurrentlyInCreation.remove(beanName)) {  
   throw new IllegalStateException("Singleton '" + beanName + "' isn't currently in creation");  
  }  
  }
  + 2.3) addSingleton（结果记录至缓存)
* 将结果记录至缓存中并删除加载bean过程中所记录的各种辅助状态
* protected void addSingleton(String beanName, Object singletonObject) {  
  synchronized (this.singletonObjects) {  
   this.singletonObjects.put(beanName, (singletonObject != null ? singletonObject : NULL\_OBJECT));  
   this.singletonFactories.remove(beanName);  
   this.earlySingletonObjects.remove(beanName);  
   this.registeredSingletons.add(beanName);  
  }  
  }
  + 3) createBean（创建单例或多例的bean，在3中有被调用)
* protected Object createBean(String beanName, RootBeanDefinition mbd, Object[] args) throws BeanCreationException {  
  if (logger.isDebugEnabled()) {  
   logger.debug("Creating instance of bean '" + beanName + "'");  
  }  
  RootBeanDefinition mbdToUse = mbd;
* // Make sure bean class is actually resolved at this point, and  
  // clone the bean definition in case of a dynamically resolved Class  
  // which cannot be stored in the shared merged bean definition.
* // 锁定class，根据设置的class属性或者根据className来解析Class  
  Class<?> resolvedClass = resolveBeanClass(mbd, beanName);  
  if (resolvedClass != null && !mbd.hasBeanClass() && mbd.getBeanClassName() != null) {  
   mbdToUse = new RootBeanDefinition(mbd);  
   mbdToUse.setBeanClass(resolvedClass);  
  }
* // Prepare method overrides.  
  try {
* // 验证及准备覆盖的方法   
   mbdToUse.prepareMethodOverrides();  
   }  
   catch (BeanDefinitionValidationException ex) {  
   throw new BeanDefinitionStoreException(mbdToUse.getResourceDescription(),  
   beanName, "Validation of method overrides failed", ex);  
   }
* try {  
   // Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.
* // 给BeanPostProcessor一个机会来返回代理来替代真正的实例  
   Object bean = resolveBeforeInstantiation(beanName, mbdToUse);  
   if (bean != null) {  
   return bean;  
   }  
   }  
   catch (Throwable ex) {  
   throw new BeanCreationException(mbdToUse.getResourceDescription(), beanName,  
   "BeanPostProcessor before instantiation of bean failed", ex);  
   }  
   // 实际创建bean  
   Object beanInstance = doCreateBean(beanName, mbdToUse, args);  
   if (logger.isDebugEnabled()) {  
   logger.debug("Finished creating instance of bean '" + beanName + "'");  
   }  
   return beanInstance;  
  }
  + 3.1) AbstractBeanDefinition#prepareMethodOverrides（决定实例化策略->反射 or CGLIB)
* 验证及准备覆盖的方法
* 在Spring配置中存在lookup-method和replace-method两个配置功能，而这两个配置的加载其实就是将配置统一存放在BeanDefinition中的methodOverrides属性里，这两个功能实现原理其实是在bean实例化的时候如果检测到存在methodOverrides属性，会动态地为当前bean生成代理并使用对应的拦截器为bean做增强处理，相关逻辑实现在bean的实例化部分详细介绍。

public void prepareMethodOverrides() throws BeanDefinitionValidationException {  
 // Check that lookup methods exists.  
 MethodOverrides methodOverrides = getMethodOverrides();  
 if (!methodOverrides.isEmpty()) {  
 Set<MethodOverride> overrides = methodOverrides.getOverrides();  
 synchronized (overrides) {  
 for (MethodOverride mo : overrides) {  
 prepareMethodOverride(mo);  
 }  
 }  
 }  
}

- 3.1.1) prepareMethodOverride

* protected void prepareMethodOverride(MethodOverride mo) throws BeanDefinitionValidationException {
  + //获取对应类中对应方法名的个数  
    int count = ClassUtils.getMethodCountForName(getBeanClass(), mo.getMethodName());  
    if (count == 0) {  
     throw new BeanDefinitionValidationException(  
     "Invalid method override: no method with name '" + mo.getMethodName() +  
     "' on class [" + getBeanClassName() + "]");  
    }  
    else if (count == 1) {  
     // Mark override as not overloaded, to avoid the overhead of arg type checking.
* // 标记MethodOverride暂未被覆盖，避免参数类型检查的开销  
   mo.setOverloaded(false);  
   }  
  }
  + 3.2) resolveBeforeInstantiation（可能会创建代理过的bean)
* 如果该方法返回bean不为空，则跳过后续实际创建bean的过程，直接返回代理后的bean。
* 与AOP有关！
* protected Object resolveBeforeInstantiation(String beanName, RootBeanDefinition mbd) {  
  Object bean = null;  
  if (!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {  
   // Make sure bean class is actually resolved at this point.  
   if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {  
   Class<?> targetType = determineTargetType(beanName, mbd);  
   if (targetType != null) {  
   bean = applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);  
   if (bean != null) {  
   bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);  
   }  
   }  
   }  
   mbd.beforeInstantiationResolved = (bean != null);  
  }  
  return bean;  
  }
  + 3.2.1) applyBeanPostProcessorsBeforeInstantiation（实例化前的后处理器应用)
* protected Object applyBeanPostProcessorsBeforeInstantiation(Class<?> beanClass, String beanName) {  
  for (BeanPostProcessor bp : getBeanPostProcessors()) {  
   if (bp instanceof InstantiationAwareBeanPostProcessor) {  
   InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;  
   Object result = ibp.postProcessBeforeInstantiation(beanClass, beanName);  
   if (result != null) {  
   return result;  
   }  
   }  
  }  
  return null;  
  }
  + 3.2.2) applyBeanPostProcessorsAfterInitialization（实例化后的后处理器应用)
* Spring中的规则是在bean的初始化后尽可能保证将注册的后处理器的postProcessAfterInitialization方法应用到该bean中，因为如果返回的bean不为空，那么便不会再次经历普通bean的创建过程。

public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)  
 throws BeansException {  
  
 Object result = existingBean;  
 for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {  
 result = beanProcessor.postProcessAfterInitialization(result, beanName);  
 if (result == null) {  
 return result;  
 }  
 }  
 return result;  
}

* + 3.3) doCreateBean（创建常规bean)
* 逻辑：
  + 1)如果是单例，则需要首先清除缓存factoryBeanInstanceCache
  + 2)实例化bean，将BeanDefinition转换为BeanWrapper。
* 转换过程：
* 如果存在工厂方法，则使用工厂方法进行初始化
* 一个类有多个构造函数，每个构造函数都有不同的参数，所以需要根据参数锁定构造函数并进行初始化
* 如果既不存在工厂方法也不存在带有参数的构造函数，则使用默认的构造函数进行bean的初始化。
  + 3)MergedBeanDefinitionPostProcessor的应用
* bean合并后的处理，Autowired注解正是通过此方法实现诸如类型的预解析。
  + 4)添加singletonFactories缓存
  + 5)属性填充
  + 6)代理bean循环依赖检查，对于已加载的bean，检测是否已经出现了循环依赖，并判断是否需要抛出异常。（仅针对于当前bean在initializeBean中被代理过的情况，正常的循环依赖在此之前就已经被检测出来的，只有代理后的bean的循环依赖是在这里检查的)
  + 7)注册DisposableBean
* 如果配置了destroy-method，这里需要注册以便于在销毁时候调用。
  + 8)完成创建并返回
* protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[] args)  
   throws BeanCreationException {
* // Instantiate the bean.  
   BeanWrapper instanceWrapper = null;  
   if (mbd.isSingleton()) {  
   instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);  
   }
* // 根据指定bean使用对应的策略创建新的实例，如：工厂方法；构造器注入；简单初始化  
  if (instanceWrapper == null) {  
   instanceWrapper = createBeanInstance(beanName, mbd, args);  
  }  
  final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstance() : null);  
  Class<?> beanType = (instanceWrapper != null ? instanceWrapper.getWrappedClass() : null);  
  mbd.resolvedTargetType = beanType;
* // Allow post-processors to modify the merged bean definition.  
  synchronized (mbd.postProcessingLock) {  
   if (!mbd.postProcessed) {  
   try {  
   applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);  
   }  
   catch (Throwable ex) {  
   throw new BeanCreationException(mbd.getResourceDescription(), beanName,  
   "Post-processing of merged bean definition failed", ex);  
   }  
   mbd.postProcessed = true;  
   }  
  }
* // Eagerly cache singletons to be able to resolve circular references  
  // even when triggered by lifecycle interfaces like BeanFactoryAware.
* // 是否需要提早曝光：单例&&允许循环依赖&&当前bean正在创建中
* // 检测循环依赖  
  boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&  
   isSingletonCurrentlyInCreation(beanName));  
  if (earlySingletonExposure) {  
   if (logger.isDebugEnabled()) {  
   logger.debug("Eagerly caching bean '" + beanName +  
   "' to allow for resolving potential circular references");  
   }
* // 放入singletonFactories，下次重新获取当前bean时可以直接返回

addSingletonFactory(beanName, new ObjectFactory<Object>() {  
 @Override  
 public Object getObject() throws BeansException {

* // 对bean再一次依赖引用，主要应用SmartInstantiationAware BeanPostProcessor ，其中AOP就是在这里将advice动态织入bean中，若没有则直接返回bean，不做任何处理  
   return getEarlyBeanReference(beanName, mbd, bean);  
   }  
   });  
  }  
  // 填充bean，将属性值注入  
  // Initialize the bean instance.  
  Object exposedObject = bean;  
  try {
* // 3.3.3  
   populateBean(beanName, mbd, instanceWrapper);  
   if (exposedObject != null) {
* // 3.3.4
* // 调用初始化方法，比如init-method
* // 注意这里是重新赋值，因为初始化会调用BeanPostProcessor，可能会返回代理对象  
   exposedObject = initializeBean(beanName, exposedObject, mbd);  
   }  
  }  
  catch (Throwable ex) {  
   if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {  
   throw (BeanCreationException) ex;  
   }  
   else {  
   throw new BeanCreationException(  
   mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);  
   }  
  }  
  // 循环依赖检查  
  if (earlySingletonExposure) {  
   Object earlySingletonReference = getSingleton(beanName, false);
* // 如果singletonObjects（缓存)中存在当前bean，earlySingletonReference不为空，它指向的是缓存中的当前bean，但未必和当前bean是同一个，因为上面initializeBean时可能会返回代理后的bean  
   if (earlySingletonReference != null) {  
   if (exposedObject == bean) {  
   exposedObject = earlySingletonReference;  
   }
* // bean不同，说明使用动态代理对其进行了增强，这是不被允许的
* // 因为其他bean依赖于当前bean，注入的是从singletonFactories中取出来的，并不是当前bean（代理后的)  
   else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {  
   String[] dependentBeans = getDependentBeans(beanName);  
   Set actualDependentBeans = new LinkedHashSet(dependentBeans.length);  
   for (String dependentBean : dependentBeans) {  
   if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {  
   actualDependentBeans.add(dependentBean);  
   }  
   }
* // 因为bean创建后它所依赖的bean一定是已经创建的，actualDependentBeans不为空表示当前bean创建后它依赖的bean却没有全部创建完，也就是说存在循环依赖  
   if (!actualDependentBeans.isEmpty()) {  
   throw new BeanCurrentlyInCreationException(beanName,  
   "Bean with name '" + beanName + "' has been injected into other beans [" +  
   StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +  
   "] in its raw version as part of a circular reference, but has eventually been " +  
   "wrapped. This means that said other beans do not use the final version of the " +  
   "bean. This is often the result of over-eager type matching - consider using " +  
   "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.");  
   }  
   }  
  }  
  }  
  // 注册DisposableBean  
  // Register bean as disposable.  
  try {  
   registerDisposableBeanIfNecessary(beanName, bean, mbd);  
  }  
  catch (BeanDefinitionValidationException ex) {  
   throw new BeanCreationException(  
   mbd.getResourceDescription(), beanName, "Invalid destruction signature", ex);  
  }
* return exposedObject;  
  }
  + 3.3.1) createBeanInstance（实例化bean)
* 逻辑：
  + 1)如果在RootBeanDefinition中存在factoryMethodName属性，或者说在配置文件中配置了factory-method，那么Spring会尝试使用instantiateUsingFactoryMethod方法根据RootBeanDefinition中的配置生成bean的实例。
  + 2)解析构造方法并进行构造方法的实例化。因为一个bean对应的类中可能会有多个构造方法，而每个构造方法的参数不同，Spring再根据参数及类型去判断最终会使用哪个构造方法进行实例化。但是，判断的过程是个比较消耗性能的步骤，所以采用缓存机制，如果已经解析过，则不需要重复解析而是直接从RootBeanDefinition中的属性resolvedConstructorOrFactoryMethod缓存的值去取，否则需要再次解析，并将解析的结果添加至RootBeanDefinition中的属性resolvedConstructorOrFactoryMethod。
* protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, Object[] args) {  
  // Make sure bean class is actually resolved at this point.

// 解析class  
 Class<?> beanClass = resolveBeanClass(mbd, beanName);  
  
 if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers()) && !mbd.isNonPublicAccessAllowed()) {  
 throw new BeanCreationException(mbd.getResourceDescription(), beanName,  
 "Bean class isn't public, and non-public access not allowed: " + beanClass.getName());  
 }  
 // 如果工厂方法不为空，则使用工厂方法初始化策略  
 if (mbd.getFactoryMethodName() != null) {  
 return instantiateUsingFactoryMethod(beanName, mbd, args);  
 }  
  
 // Shortcut when re-creating the same bean...  
 boolean resolved = false;  
 boolean autowireNecessary = false;  
 if (args == null) {  
 synchronized (mbd.constructorArgumentLock) {  
 if (mbd.resolvedConstructorOrFactoryMethod != null) {  
 resolved = true;  
 autowireNecessary = mbd.constructorArgumentsResolved;  
 }  
 }  
 }

* // 如果解析过，那么直接创建；否则要获取构造方法  
  if (resolved) {  
   if (autowireNecessary) {  
   return autowireConstructor(beanName, mbd, null, null);  
   }  
   else {  
   return instantiateBean(beanName, mbd);  
   }  
  }  
  // 需要根据参数解析构造方法  
  // Need to determine the constructor...  
  Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);  
  if (ctors != null ||  
   mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_CONSTRUCTOR ||  
   mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args)) {
* // 构造方法自动注入  
   return autowireConstructor(beanName, mbd, ctors, args);  
   }  
   // 使用默认构造方法  
   // No special handling: simply use no-arg constructor.  
   return instantiateBean(beanName, mbd);  
  }

- 3.3.1.1) instantiateUsingFactoryMethod（工厂方法的实例化)

* AbstractAutowireCapableBeanFactory. instantiateUsingFactoryMethod
* protected BeanWrapper instantiateUsingFactoryMethod(  
   String beanName, RootBeanDefinition mbd, Object[] explicitArgs) {
* return new ConstructorResolver(this).instantiateUsingFactoryMethod(beanName, mbd, explicitArgs);  
  }
  + 3.3.1.2) autowireConstructor（有参数的构造方法的实例化)
* protected BeanWrapper autowireConstructor(  
   String beanName, RootBeanDefinition mbd, Constructor<?>[] ctors, Object[] explicitArgs) {
* return new ConstructorResolver(this).autowireConstructor(beanName, mbd, ctors, explicitArgs);  
  }
* ConstructorResolver.autowireConstructor
* 逻辑：
  + 1)确定 构造方法的参数
* 根据explicitArgs参数判断
* 如果传入的参数explicitArgs不为空，那么可以直接确定参数，因为explicitArgs参数是在getBean的时候用户指定的。——getBean(String name,Object ...args)
* 在获取bean的时候，用户不但可以指定bean的名称，还可以指定bean所对应类的构造函数或者工厂方法的方法参数，主要用于静态工厂方法的调用，而这里是需要给定完全匹配的参数的。
* 所以，如果传入参数explicitArgs不为空，则可以确定构造方法参数就是它。
* 从缓存中获取
* 如果确定参数的办法之前已经分析过，即构造方法参数已经记录在缓存中，那么可以直接拿来使用。而且，在缓存中缓存的可能是参数的最终类型，也可能是参数的初始类型。
* 从配置文件获取
* 即从头开始分析。
* 分析从获取配置文件中配置的构造方法信息开始，可以调用BeanDefinition.getConstructorArgumentValues()来获取配置的构造方法信息。
* 有了配置中的信息便可以获取对应的参数值信息了，获取参数值的信息包括直接指定值，而这一处理委托给resolveConstructorArguments方法，并返回能解析到的参数的个数。
  + 2)确定 构造方法

根据构造方法参数在所有构造方法中锁定对应的构造方法，匹配的方法就是根据参数个数匹配，所以在匹配之前需要先对构造方法按照public构造方法优先参数数量降序、非public构造方法参数数量降序。这样可以在遍历的情况下迅速判断排在后面的构造方法参数个数是否符合条件。

* 由于在配置文件中并不是唯一限制使用参数位置索引的方法去创建，同样还支持指定参数名称进行设定参数值的情况，如，那么这种情况就需要首先确定构造方法中的参数名称。
* 获取参数名称可以有两种方式，一种是通过注解的方式直接获取，另一种就是使用Spring中提供的工具类ParameterNameDiscoverer来获取。构造方法、参数名称、参数类型、参数值都确定后就可以锁定构造方法以及转换对应的参数类型了。
  + 3)根据确定的构造方法转换对应的参数类型
* 主要是使用Spring中提供的类型转换器或者用户提供的自定义类型转换器进行转换
  + 4)构造方法不确定性的验证
* 有时候即使构造方法、参数名称、参数类型、参数值都确定后也不一定会直接锁定构造方法，所以Spring在最后又做了一次验证。
  + 5)根据实例化策略以及得到的构造方法及构造方法参数实例化bean。

public BeanWrapper autowireConstructor(final String beanName, final RootBeanDefinition mbd,  
 Constructor<?>[] chosenCtors, final Object[] explicitArgs) {  
  
 BeanWrapperImpl bw = new BeanWrapperImpl();  
 this.beanFactory.initBeanWrapper(bw);  
  
 Constructor<?> constructorToUse = null;  
 ArgumentsHolder argsHolderToUse = null;  
 Object[] argsToUse = null;  
 // explicitArgs通过getBean方法传入

* // 如果getBean方法调用的时候指定方法参数，那么直接使用  
  if (explicitArgs != null) {  
   argsToUse = explicitArgs;  
  }  
  else {
* // 如果在getBean方法时候没有指定则尝试从配置文件中解析  
   Object[] argsToResolve = null;
* // 尝试从缓存中获取  
   synchronized (mbd.constructorArgumentLock) {  
   constructorToUse = (Constructor<?>) mbd.resolvedConstructorOrFactoryMethod;  
   if (constructorToUse != null && mbd.constructorArgumentsResolved) {  
   // Found a cached constructor...
* // 从缓存中取  
   argsToUse = mbd.resolvedConstructorArguments;  
   if (argsToUse == null) {
* // 配置的构造方法参数  
   argsToResolve = mbd.preparedConstructorArguments;  
   }  
  }  
  }
* // 如果缓存中存在  
   if (argsToResolve != null) {
  + // 解析参数类型，如给定方法的构造方法A(int,int)，则通过此方法后就会把配置中的(“1”,”1”)转换为(1,1)
* // 缓存中的值可能是原始值，也可能是最终值  
   argsToUse = resolvePreparedArguments(beanName, mbd, bw, constructorToUse, argsToResolve);  
   }  
  }  
  //没有被缓存  
  if (constructorToUse == null) {  
   // Need to resolve the constructor.  
   boolean autowiring = (chosenCtors != null ||  
   mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE\_CONSTRUCTOR);  
   ConstructorArgumentValues resolvedValues = null;
* int minNrOfArgs;  
   if (explicitArgs != null) {  
   minNrOfArgs = explicitArgs.length;  
   }  
   else {
* // 提取配置文件中的配置的构造方法参数  
   ConstructorArgumentValues cargs = mbd.getConstructorArgumentValues();
* // 用于承载解析后的构造方法参数的值  
   resolvedValues = new ConstructorArgumentValues();
* // 能解析到的参数个数  
   minNrOfArgs = resolveConstructorArguments(beanName, mbd, bw, cargs, resolvedValues);  
   }
* // Take specified constructors, if any.  
   Constructor<?>[] candidates = chosenCtors;  
   if (candidates == null) {  
   Class<?> beanClass = mbd.getBeanClass();  
   try {  
   candidates = (mbd.isNonPublicAccessAllowed() ?  
   beanClass.getDeclaredConstructors() : beanClass.getConstructors());  
   }  
   catch (Throwable ex) {  
   throw new BeanCreationException(mbd.getResourceDescription(), beanName,  
   "Resolution of declared constructors on bean Class [" + beanClass.getName() +  
   "] from ClassLoader [" + beanClass.getClassLoader() + "] failed", ex);  
   }  
   }

// 排序给定的构造方法，public构造方法优先参数数量降序、非public构造方法参数数量降序  
 AutowireUtils.sortConstructors(candidates);  
 int minTypeDiffWeight = Integer.MAX\_VALUE;  
 Set<Constructor<?>> ambiguousConstructors = null;  
 LinkedList<UnsatisfiedDependencyException> causes = null;  
  
 for (Constructor<?> candidate : candidates) {  
 Class<?>[] paramTypes = candidate.getParameterTypes();  
  
 if (constructorToUse != null && argsToUse.length > paramTypes.length) {

* // 如果已经找到选用的构造方法或者需要的参数个数小于当前的构造方法参数个数，则终止  
   // Already found greedy constructor that can be satisfied ->  
   // do not look any further, there are only less greedy constructors left.  
   break;  
   }  
   if (paramTypes.length < minNrOfArgs) {
* // 参数个数不相等  
   continue;  
   }  
   // 参数个数相等  
   ArgumentsHolder argsHolder;
* // 如果构造方法有参数，则根据值构造对应参数类型的参数  
   if (resolvedValues != null) {  
   try {
* // 从注解上获取参数名称  
   String[] paramNames = ConstructorPropertiesChecker.evaluate(candidate, paramTypes.length);  
   if (paramNames == null) {
* // 获取参数名称探索器  
   ParameterNameDiscoverer pnd = this.beanFactory.getParameterNameDiscoverer();  
   if (pnd != null) {
* // 获取指定构造方法的参数名称  
   paramNames = pnd.getParameterNames(candidate);  
   }  
  }
* // 根据名称和数据类型 创建参数持有者  
   argsHolder = createArgumentArray(beanName, mbd, resolvedValues, bw, paramTypes, paramNames,  
   getUserDeclaredConstructor(candidate), autowiring);  
   }  
   catch (UnsatisfiedDependencyException ex) {  
   if (this.beanFactory.logger.isTraceEnabled()) {  
   this.beanFactory.logger.trace(  
   "Ignoring constructor [" + candidate + "] of bean '" + beanName + "': " + ex);  
   }  
   // Swallow and try next constructor.  
   if (causes == null) {  
   causes = new LinkedList();  
   }  
   causes.add(ex);  
   continue;  
   }  
   }
* // 如果构造方法没有参数  
   else {  
   // Explicit arguments given -> arguments length must match exactly.  
   if (paramTypes.length != explicitArgs.length) {  
   continue;  
   }  
   argsHolder = new ArgumentsHolder(explicitArgs);  
   }  
   // 探测是否有不确定性的构造方法存在，例如不同构造方法的参数为父子关系  
   int typeDiffWeight = (mbd.isLenientConstructorResolution() ?  
   argsHolder.getTypeDifferenceWeight(paramTypes) : argsHolder.getAssignabilityWeight(paramTypes));  
   // Choose this constructor if it represents the closest match.
* // 如果它代表着当前最接近的匹配则选择作为构造方法  
   if (typeDiffWeight < minTypeDiffWeight) {  
   constructorToUse = candidate;  
   argsHolderToUse = argsHolder;  
   argsToUse = argsHolder.arguments;  
   minTypeDiffWeight = typeDiffWeight;  
   ambiguousConstructors = null;  
   }  
   else if (constructorToUse != null && typeDiffWeight == minTypeDiffWeight) {  
   if (ambiguousConstructors == null) {  
   ambiguousConstructors = new LinkedHashSet<Constructor<?>>();  
   ambiguousConstructors.add(constructorToUse);  
   }  
   ambiguousConstructors.add(candidate);  
   }  
   }
* if (constructorToUse == null) {  
   if (causes != null) {  
   UnsatisfiedDependencyException ex = causes.removeLast();  
   for (Exception cause : causes) {  
   this.beanFactory.onSuppressedException(cause);  
   }  
   throw ex;  
   }  
   throw new BeanCreationException(mbd.getResourceDescription(), beanName,  
   "Could not resolve matching constructor " +  
   "(hint: specify index/type/name arguments for simple parameters to avoid type ambiguities)");  
   }  
   else if (ambiguousConstructors != null && !mbd.isLenientConstructorResolution()) {  
   throw new BeanCreationException(mbd.getResourceDescription(), beanName,  
   "Ambiguous constructor matches found in bean '" + beanName + "' " +  
   "(hint: specify index/type/name arguments for simple parameters to avoid type ambiguities): " +  
   ambiguousConstructors);  
   }
* if (explicitArgs == null) {

// 将解析的构造方法加入缓存  
 argsHolderToUse.storeCache(mbd, constructorToUse);  
 }  
 }  
  
 try {  
 Object beanInstance;  
  
 if (System.getSecurityManager() != null) {  
 final Constructor<?> ctorToUse = constructorToUse;  
 final Object[] argumentsToUse = argsToUse;  
 beanInstance = AccessController.doPrivileged(new PrivilegedAction<Object>() {  
 @Override  
 public Object run() {  
 return beanFactory.getInstantiationStrategy().instantiate(  
 mbd, beanName, beanFactory, ctorToUse, argumentsToUse);  
 }  
 }, beanFactory.getAccessControlContext());  
 }  
 else {  
 beanInstance = this.beanFactory.getInstantiationStrategy().instantiate(  
 mbd, beanName, this.beanFactory, constructorToUse, argsToUse);  
 }  
 // 将构建的实例加入BeanWrapper中  
 bw.setBeanInstance(beanInstance);  
 return bw;  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(mbd.getResourceDescription(), beanName,  
 "Bean instantiation via constructor failed", ex);  
 }  
}

- 3.3.1.2.1) createArgumentArray（递归获取参数的bean)

private ArgumentsHolder createArgumentArray(  
 String beanName, RootBeanDefinition mbd, ConstructorArgumentValues resolvedValues,  
 BeanWrapper bw, Class<?>[] paramTypes, String[] paramNames, Object methodOrCtor,  
 boolean autowiring) throws UnsatisfiedDependencyException {  
  
 String methodType = (methodOrCtor instanceof Constructor ? "constructor" : "factory method");  
 TypeConverter converter = (this.beanFactory.getCustomTypeConverter() != null ?  
 this.beanFactory.getCustomTypeConverter() : bw);  
  
 ArgumentsHolder args = new ArgumentsHolder(paramTypes.length);  
 Set<ConstructorArgumentValues.ValueHolder> usedValueHolders =  
 new HashSet<ConstructorArgumentValues.ValueHolder>(paramTypes.length);  
 Set<String> autowiredBeanNames = new LinkedHashSet<String>(4);  
  
 for (int paramIndex = 0; paramIndex < paramTypes.length; paramIndex++) {  
 Class<?> paramType = paramTypes[paramIndex];  
 String paramName = (paramNames != null ? paramNames[paramIndex] : null);  
 // Try to find matching constructor argument value, either indexed or generic.  
 ConstructorArgumentValues.ValueHolder valueHolder =  
 resolvedValues.getArgumentValue(paramIndex, paramType, paramName, usedValueHolders);  
 // If we couldn't find a direct match and are not supposed to autowire,  
 // let's try the next generic, untyped argument value as fallback:  
 // it could match after type conversion (for example, String -> int).  
 if (valueHolder == null && !autowiring) {  
 valueHolder = resolvedValues.getGenericArgumentValue(null, null, usedValueHolders);  
 }  
 if (valueHolder != null) {  
 // We found a potential match - let's give it a try.  
 // Do not consider the same value definition multiple times!  
 usedValueHolders.add(valueHolder);  
 Object originalValue = valueHolder.getValue();  
 Object convertedValue;  
 if (valueHolder.isConverted()) {  
 convertedValue = valueHolder.getConvertedValue();  
 args.preparedArguments[paramIndex] = convertedValue;  
 }  
 else {  
 ConstructorArgumentValues.ValueHolder sourceHolder =  
 (ConstructorArgumentValues.ValueHolder) valueHolder.getSource();  
 Object sourceValue = sourceHolder.getValue();  
 try {  
 convertedValue = converter.convertIfNecessary(originalValue, paramType,  
 MethodParameter.forMethodOrConstructor(methodOrCtor, paramIndex));  
 // TODO re-enable once race condition has been found (SPR-7423)  
 /\*  
 if (originalValue == sourceValue || sourceValue instanceof TypedStringValue) {  
 // Either a converted value or still the original one: store converted value.  
 sourceHolder.setConvertedValue(convertedValue);  
 args.preparedArguments[paramIndex] = convertedValue;  
 }  
 else {  
 \*/  
 args.resolveNecessary = true;  
 args.preparedArguments[paramIndex] = sourceValue;  
 // }  
 }  
 catch (TypeMismatchException ex) {  
 throw new UnsatisfiedDependencyException(  
 mbd.getResourceDescription(), beanName, paramIndex, paramType,  
 "Could not convert " + methodType + " argument value of type [" +  
 ObjectUtils.nullSafeClassName(valueHolder.getValue()) +  
 "] to required type [" + paramType.getName() + "]: " + ex.getMessage());  
 }  
 }  
 args.arguments[paramIndex] = convertedValue;  
 args.rawArguments[paramIndex] = originalValue;  
 }  
 else {  
 // No explicit match found: we're either supposed to autowire or  
 // have to fail creating an argument array for the given constructor.  
 if (!autowiring) {  
 throw new UnsatisfiedDependencyException(  
 mbd.getResourceDescription(), beanName, paramIndex, paramType,  
 "Ambiguous " + methodType + " argument types - " +  
 "did you specify the correct bean references as " + methodType + " arguments?");  
 }  
 try {  
 MethodParameter param = MethodParameter.forMethodOrConstructor(methodOrCtor, paramIndex);  
 Object autowiredArgument = resolveAutowiredArgument(param, beanName, autowiredBeanNames, converter);  
 args.rawArguments[paramIndex] = autowiredArgument;  
 args.arguments[paramIndex] = autowiredArgument;  
 args.preparedArguments[paramIndex] = new AutowiredArgumentMarker();  
 args.resolveNecessary = true;  
 }  
 catch (BeansException ex) {  
 throw new UnsatisfiedDependencyException(  
 mbd.getResourceDescription(), beanName, paramIndex, paramType, ex);  
 }  
 }  
 }  
  
 for (String autowiredBeanName : autowiredBeanNames) {  
 this.beanFactory.registerDependentBean(autowiredBeanName, beanName);  
 if (this.beanFactory.logger.isDebugEnabled()) {  
 this.beanFactory.logger.debug("Autowiring by type from bean name '" + beanName +  
 "' via " + methodType + " to bean named '" + autowiredBeanName + "'");  
 }  
 }  
  
 return args;  
}

- 3.3.1.2.1.1) resolveAutowiredArgument

* protected Object resolveAutowiredArgument(  
   MethodParameter param, String beanName, Set autowiredBeanNames, TypeConverter typeConverter) {
* return this.beanFactory.resolveDependency(  
   new DependencyDescriptor(param, true), beanName, autowiredBeanNames, typeConverter);  
  }
  + 3.3.1.2.2) InstantiationStrategy.instantiate
* 实例化的时候可以采用不同的策略进行实例化。
* 如果beanDefinition.getMethodOverrides()为空，即用户没有使用replace或lookup的配置方法，那么直接使用反射的方式；如果使用了，需要将这两个配置通过的功能切入进去，所以就必须要使用动态代理的方式将包含两个特性所对应的逻辑的拦截增强器设置进去，这样才可以保证在调用方法的时候会被相应的拦截器增强，返回值为包含拦截器的代理实例。
* 以SimpleInstantiationStrategy为例：

public Object instantiate(RootBeanDefinition bd, String beanName, BeanFactory owner) {  
 // Don't override the class with CGLIB if no overrides.

* // 如果有需要覆盖或者动态替换的方法，则使用cglib进行动态代理，因为可以在创建代理的同时将动态方法织入类中，但是如果没有需要动态改变的方法，为了方便直接反射就可以了  
  if (bd.getMethodOverrides().isEmpty()) {

// 没有需要覆盖的方法  
 Constructor<?> constructorToUse;  
 synchronized (bd.constructorArgumentLock) {  
 constructorToUse = (Constructor<?>) bd.resolvedConstructorOrFactoryMethod;  
 if (constructorToUse == null) {  
 final Class<?> clazz = bd.getBeanClass();  
 if (clazz.isInterface()) {  
 throw new BeanInstantiationException(clazz, "Specified class is an interface");  
 }  
 try {  
 if (System.getSecurityManager() != null) {  
 constructorToUse = AccessController.doPrivileged(new PrivilegedExceptionAction<Constructor<?>>() {  
 @Override  
 public Constructor<?> run() throws Exception {  
 return clazz.getDeclaredConstructor((Class[]) null);  
 }  
 });  
 }  
 else {  
 constructorToUse = clazz.getDeclaredConstructor((Class[]) null);  
 }  
 bd.resolvedConstructorOrFactoryMethod = constructorToUse;  
 }  
 catch (Throwable ex) {  
 throw new BeanInstantiationException(clazz, "No default constructor found", ex);  
 }  
 }  
 }

* // 反射实例化  
   return BeanUtils.instantiateClass(constructorToUse);  
   }  
   else {  
   // Must generate CGLIB subclass.  
   return instantiateWithMethodInjection(bd, beanName, owner);  
   }  
  }
* 以CglibSubclassingInstantiationStrategy为例：

public Object instantiate(Constructor<?> ctor, Object... args) {  
 Class<?> subclass = createEnhancedSubclass(this.beanDefinition);  
 Object instance;  
 if (ctor == null) {  
 instance = BeanUtils.instantiateClass(subclass);  
 }  
 else {  
 try {  
 Constructor<?> enhancedSubclassConstructor = subclass.getConstructor(ctor.getParameterTypes());  
 instance = enhancedSubclassConstructor.newInstance(args);  
 }  
 catch (Exception ex) {  
 throw new BeanInstantiationException(this.beanDefinition.getBeanClass(),  
 "Failed to invoke constructor for CGLIB enhanced subclass [" + subclass.getName() + "]", ex);  
 }  
 }  
 // SPR-10785: set callbacks directly on the instance instead of in the  
 // enhanced class (via the Enhancer) in order to avoid memory leaks.  
 Factory factory = (Factory) instance;  
 factory.setCallbacks(new Callback[] {NoOp.INSTANCE,  
 new LookupOverrideMethodInterceptor(this.beanDefinition, this.owner),  
 new ReplaceOverrideMethodInterceptor(this.beanDefinition, this.owner)});  
 return instance;  
}

* createEnhancedSubclass:

private Class<?> createEnhancedSubclass(RootBeanDefinition beanDefinition) {  
 Enhancer enhancer = new Enhancer();  
 enhancer.setSuperclass(beanDefinition.getBeanClass());  
 enhancer.setNamingPolicy(SpringNamingPolicy.INSTANCE);  
 if (this.owner instanceof ConfigurableBeanFactory) {  
 ClassLoader cl = ((ConfigurableBeanFactory) this.owner).getBeanClassLoader();  
 enhancer.setStrategy(new ClassLoaderAwareGeneratorStrategy(cl));  
 }  
 enhancer.setCallbackFilter(new MethodOverrideCallbackFilter(beanDefinition));  
 enhancer.setCallbackTypes(CALLBACK\_TYPES);  
 return enhancer.createClass();  
}

* + 3.3.1.3) instantiateBean（无参数的构造方法的实例化)

protected BeanWrapper instantiateBean(final String beanName, final RootBeanDefinition mbd) {  
 try {  
 Object beanInstance;  
 final BeanFactory parent = this;  
 if (System.getSecurityManager() != null) {  
 beanInstance = AccessController.doPrivileged(new PrivilegedAction<Object>() {  
 @Override  
 public Object run() {  
 return getInstantiationStrategy().instantiate(mbd, beanName, parent);  
 }  
 }, getAccessControlContext());  
 }  
 else {

* // 实例化即可，见3.3.1.2.1  
   beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);  
   }  
   BeanWrapper bw = new BeanWrapperImpl(beanInstance);  
   initBeanWrapper(bw);  
   return bw;  
  }  
  catch (Throwable ex) {  
   throw new BeanCreationException(  
   mbd.getResourceDescription(), beanName, "Instantiation of bean failed", ex);  
  }  
  }
  + 3.3.2) addSingletonFactory（singletonFactories缓存)
* 执行到这里时可以肯定当前bean已经构造完成，只是尚未填充属性，但是内存地址已经确定了。将当前bean加入到singletonFactories中，下次getBean时会先检测当前bean是否已经被加入到singletonFactories，如果已经存在，则返回缓存，否则就正常创建。
* protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {  
  Assert.notNull(singletonFactory, "Singleton factory must not be null");  
  synchronized (this.singletonObjects) {  
   if (!this.singletonObjects.containsKey(beanName)) {  
   this.singletonFactories.put(beanName, singletonFactory);  
   this.earlySingletonObjects.remove(beanName);  
   this.registeredSingletons.add(beanName);  
   }  
  }  
  }
  + 3.3.3) getEarlyBeanReference（从singletonFactories取出来时调用的getObject)
* protected Object getEarlyBeanReference(String beanName, RootBeanDefinition mbd, Object bean) {  
  Object exposedObject = bean;  
  if (bean != null && !mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {  
   for (BeanPostProcessor bp : getBeanPostProcessors()) {  
   if (bp instanceof SmartInstantiationAwareBeanPostProcessor) {  
   SmartInstantiationAwareBeanPostProcessor ibp = (SmartInstantiationAwareBeanPostProcessor) bp;  
   exposedObject = ibp.getEarlyBeanReference(exposedObject, beanName);  
   if (exposedObject == null) {  
   return null;  
   }  
   }  
   }  
  }  
  return exposedObject;  
  }

- 3.3.4) polulateBean（属性值注入)

* 逻辑：
  + 1)InstantiationAwareBeanPostProcessor处理器的postProcessAfterInstantiation函数的应用，此函数可以控制程序是否继续进行属性填充。
  + 2)根据注入类型（byName/byType)，提取依赖的bean，并统一存入PropertyValues中。
  + 3)应用InstantiationAwareBeanPostProcessor处理器的postProcessPropertyValues方法，对属性获取完毕填充前 对属性的再次处理，典型应用是RequiredAnnotationBeanPostProcessor类中对属性的验证
  + 4)将所有PropertyValues中的属性填充至BeanWrapper中。
* protected void populateBean(String beanName, RootBeanDefinition mbd, BeanWrapper bw) {  
  PropertyValues pvs = mbd.getPropertyValues();
* if (bw == null) {  
   if (!pvs.isEmpty()) {  
   throw new BeanCreationException(  
   mbd.getResourceDescription(), beanName, "Cannot apply property values to null instance");  
   }  
   else {
* // 没有可填充的属性  
   // Skip property population phase for null instance.  
   return;  
   }  
  }  
  // 给InstantiationAwareBeanPostProcessor最后一次机会在属性设置前来改变bean
* // 如：可以用来支持属性注入的类型  
  // Give any InstantiationAwareBeanPostProcessors the opportunity to modify the  
  // state of the bean before properties are set. This can be used, for example,  
  // to support styles of field injection.  
  boolean continueWithPropertyPopulation = true;
* if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {  
   for (BeanPostProcessor bp : getBeanPostProcessors()) {  
   if (bp instanceof InstantiationAwareBeanPostProcessor) {  
   InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
* // 返回值为是否继续填充bean  
   if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {  
   continueWithPropertyPopulation = false;  
   break;  
   }  
   }  
   }  
  }  
  // 如果后处理器发出停止填充命令则终止后续的执行  
  if (!continueWithPropertyPopulation) {  
   return;  
  }
* if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE*BY*NAME ||  
   mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE*BY*TYPE) {  
   MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
* // Add property values based on autowire by name if applicable.  
   if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE*BY*NAME) {
* // 根据名称自动注入，存入PropertyValues  
   autowireByName(beanName, mbd, bw, newPvs);  
   }
* // Add property values based on autowire by type if applicable.  
   if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE*BY*TYPE) {
* // 根据类型自动注入，存入PropertyValues  
   autowireByType(beanName, mbd, bw, newPvs);  
   }
* pvs = newPvs;  
  }  
  // 后处理器已经初始化  
  boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
* // 需要依赖检查  
  boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.DEPENDENCY*CHECK*NONE);
* if (hasInstAwareBpps || needsDepCheck) {  
   PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);  
   if (hasInstAwareBpps) {
* // 对所有需要依赖检查的属性进行后处理  
   for (BeanPostProcessor bp : getBeanPostProcessors()) {  
   if (bp instanceof InstantiationAwareBeanPostProcessor) {  
   InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;  
   pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);  
   if (pvs == null) {  
   return;  
   }  
   }  
   }  
   }  
   if (needsDepCheck) {  
   checkDependencies(beanName, mbd, filteredPds, pvs);  
   }  
  }  
  // 将属性应用到bean中  
  applyPropertyValues(beanName, mbd, bw, pvs);  
  }
  + 3.3.4.1) autowireByName（按名获取待注入的属性)
* protected void autowireByName(  
   String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs) {  
   // 寻找bw中需要依赖注入的属性  
   String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);  
   for (String propertyName : propertyNames) {  
   if (containsBean(propertyName)) {
* // 递归初始化相关的bean  
   Object bean = getBean(propertyName);
* pvs.add(propertyName, bean);
* // 注册依赖  
   registerDependentBean(propertyName, beanName);  
   if (logger.isDebugEnabled()) {  
   logger.debug("Added autowiring by name from bean name '" + beanName +  
   "' via property '" + propertyName + "' to bean named '" + propertyName + "'");  
   }  
   }  
   else {  
   if (logger.isTraceEnabled()) {  
   logger.trace("Not autowiring property '" + propertyName + "' of bean '" + beanName +  
   "' by name: no matching bean found");  
   }  
   }  
  }  
  }
  + 3.3.4.2) autowireByType（按类型获取待注入的属性)
  + protected void autowireByType(  
    String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs) {
* TypeConverter converter = getCustomTypeConverter();  
  if (converter == null) {  
   converter = bw;  
  }
* Set autowiredBeanNames = new LinkedHashSet(4);
* // 寻找bw中需要依赖注入的属性  
  String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);  
  for (String propertyName : propertyNames) {  
   try {  
   PropertyDescriptor pd = bw.getPropertyDescriptor(propertyName);  
   // Don't try autowiring by type for type Object: never makes sense,  
   // even if it technically is a unsatisfied, non-simple property.
* if (Object.class != pd.getPropertyType()) {
* // 探测指定属性的set方法  
   MethodParameter methodParam = BeanUtils.getWriteMethodParameter(pd);  
   // Do not allow eager init for type matching in case of a prioritized post-processor.  
   boolean eager = !PriorityOrdered.class.isAssignableFrom(bw.getWrappedClass());  
   DependencyDescriptor desc = new AutowireByTypeDependencyDescriptor(methodParam, eager);
* // 解析指定beanName的属性所匹配的值，并把解析到的属性名称存储在autowireBeanNames中  
   Object autowiredArgument = resolveDependency(desc, beanName, autowiredBeanNames, converter);  
   if (autowiredArgument != null) {  
   pvs.add(propertyName, autowiredArgument);  
   }

// 如@Autowired private List<A> list;

* // 这时候会找到所有匹配A类型的bean并将其注入，所以每个属性可能会对应多个bean  
   for (String autowiredBeanName : autowiredBeanNames) {
* // 注册依赖  
   registerDependentBean(autowiredBeanName, beanName);  
   if (logger.isDebugEnabled()) {  
   logger.debug("Autowiring by type from bean name '" + beanName + "' via property '" +  
   propertyName + "' to bean named '" + autowiredBeanName + "'");  
   }  
   }  
   autowiredBeanNames.clear();  
  }  
  }  
  catch (BeansException ex) {  
   throw new UnsatisfiedDependencyException(mbd.getResourceDescription(), beanName, propertyName, ex);  
  }  
  }  
  }
  + 3.3.4.2.1) DefaultListableBeanFactory#resolveDependency（寻找类型匹配)

public Object resolveDependency(DependencyDescriptor descriptor, String requestingBeanName,  
 Set<String> autowiredBeanNames, TypeConverter typeConverter) throws BeansException {  
  
 descriptor.initParameterNameDiscovery(getParameterNameDiscoverer());

* // 处理特定的类的注入  
  if (javaUtilOptionalClass == descriptor.getDependencyType()) {  
   return new OptionalDependencyFactory().createOptionalDependency(descriptor, requestingBeanName);  
  }  
  else if (ObjectFactory.class == descriptor.getDependencyType() ||  
   ObjectProvider.class == descriptor.getDependencyType()) {  
   return new DependencyObjectProvider(descriptor, requestingBeanName);  
  }  
  else if (javaxInjectProviderClass == descriptor.getDependencyType()) {  
   return new Jsr330ProviderFactory().createDependencyProvider(descriptor, requestingBeanName);  
  }  
  else {  
   Object result = getAutowireCandidateResolver().getLazyResolutionProxyIfNecessary(  
   descriptor, requestingBeanName);  
   if (result == null) {
* // 处理通用逻辑  
   result = doResolveDependency(descriptor, requestingBeanName, autowiredBeanNames, typeConverter);  
   }  
   return result;  
  }  
  }
  + 3.3.4.2.1.1) doResolveDependency

public Object doResolveDependency(DependencyDescriptor descriptor, String beanName,  
 Set<String> autowiredBeanNames, TypeConverter typeConverter) throws BeansException {  
  
 InjectionPoint previousInjectionPoint = ConstructorResolver.setCurrentInjectionPoint(descriptor);  
 try {  
 Object shortcut = descriptor.resolveShortcut(this);  
 if (shortcut != null) {  
 return shortcut;  
 }  
  
 Class<?> type = descriptor.getDependencyType();  
 Object value = getAutowireCandidateResolver().getSuggestedValue(descriptor);  
 if (value != null) {  
 if (value instanceof String) {  
 String strVal = resolveEmbeddedValue((String) value);  
 BeanDefinition bd = (beanName != null && containsBean(beanName) ? getMergedBeanDefinition(beanName) : null);  
 value = evaluateBeanDefinitionString(strVal, bd);  
 }  
 TypeConverter converter = (typeConverter != null ? typeConverter : getTypeConverter());  
 return (descriptor.getField() != null ?  
 converter.convertIfNecessary(value, type, descriptor.getField()) :  
 converter.convertIfNecessary(value, type, descriptor.getMethodParameter()));  
 }  
  
 Object multipleBeans = resolveMultipleBeans(descriptor, beanName, autowiredBeanNames, typeConverter);  
 if (multipleBeans != null) {  
 return multipleBeans;  
 }  
  
 Map<String, Object> matchingBeans = findAutowireCandidates(beanName, type, descriptor);  
 if (matchingBeans.isEmpty()) {  
 if (descriptor.isRequired()) {  
 raiseNoMatchingBeanFound(type, descriptor.getResolvableType(), descriptor);  
 }  
 return null;  
 }  
  
 String autowiredBeanName;  
 Object instanceCandidate;  
  
 if (matchingBeans.size() > 1) {  
 autowiredBeanName = determineAutowireCandidate(matchingBeans, descriptor);  
 if (autowiredBeanName == null) {  
 if (descriptor.isRequired() || !indicatesMultipleBeans(type)) {  
 return descriptor.resolveNotUnique(type, matchingBeans);  
 }  
 else {  
 // In case of an optional Collection/Map, silently ignore a non-unique case:  
 // possibly it was meant to be an empty collection of multiple regular beans  
 // (before 4.3 in particular when we didn't even look for collection beans).  
 return null;  
 }  
 }  
 instanceCandidate = matchingBeans.get(autowiredBeanName);  
 }  
 else {  
 // We have exactly one match.  
 Map.Entry<String, Object> entry = matchingBeans.entrySet().iterator().next();  
 autowiredBeanName = entry.getKey();  
 instanceCandidate = entry.getValue();  
 }  
  
 if (autowiredBeanNames != null) {  
 autowiredBeanNames.add(autowiredBeanName);  
 }  
 return (instanceCandidate instanceof Class ?  
 descriptor.resolveCandidate(autowiredBeanName, type, this) : instanceCandidate);  
 }  
 finally {  
 ConstructorResolver.setCurrentInjectionPoint(previousInjectionPoint);  
 }  
}

- 3.3.4.2.1.1.1) findAutowireCandidates（获取setter注入的bean)

* protected Map<String, Object> findAutowireCandidates(  
   String beanName, Class<?> requiredType, DependencyDescriptor descriptor) {
* String[] candidateNames = BeanFactoryUtils.beanNamesForTypeIncludingAncestors(  
   this, requiredType, true, descriptor.isEager());  
   Map<String, Object> result = new LinkedHashMap<String, Object>(candidateNames.length);  
   for (Class<?> autowiringType : this.resolvableDependencies.keySet()) {  
   if (autowiringType.isAssignableFrom(requiredType)) {  
   Object autowiringValue = this.resolvableDependencies.get(autowiringType);  
   autowiringValue = AutowireUtils.resolveAutowiringValue(autowiringValue, requiredType);  
   if (requiredType.isInstance(autowiringValue)) {  
   result.put(ObjectUtils.identityToString(autowiringValue), autowiringValue);  
   break;  
   }  
   }  
   }  
   for (String candidateName : candidateNames) {  
   if (!isSelfReference(beanName, candidateName) && isAutowireCandidate(candidateName, descriptor)) {  
   result.put(candidateName, getBean(candidateName));  
   }  
   }  
   if (result.isEmpty()) {  
   DependencyDescriptor fallbackDescriptor = descriptor.forFallbackMatch();  
   for (String candidateName : candidateNames) {  
   if (!candidateName.equals(beanName) && isAutowireCandidate(candidateName, fallbackDescriptor)) {  
   result.put(candidateName, getBean(candidateName));  
   }  
   }  
   }  
   return result;  
  }
  + 3.3.4.3) AutowiredAnnotationBeanPostProcesser#postProcessPropertyValues（@Autowired依赖注入)

public PropertyValues postProcessPropertyValues(  
 PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String beanName) throws BeansException {  
  
 InjectionMetadata metadata = findAutowiringMetadata(beanName, bean.getClass(), pvs);  
 try {  
 metadata.inject(bean, beanName, pvs);  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(beanName, "Injection of autowired dependencies failed", ex);  
 }  
 return pvs;  
}

- 3.3.4.3.1) InjectionMetadata#inject

public void inject(Object target, String beanName, PropertyValues pvs) throws Throwable {  
 Collection<InjectedElement> elementsToIterate =  
 (this.checkedElements != null ? this.checkedElements : this.injectedElements);  
 if (!elementsToIterate.isEmpty()) {  
 boolean debug = logger.isDebugEnabled();  
 for (InjectedElement element : elementsToIterate) {  
 if (debug) {  
 logger.debug("Processing injected element of bean '" + beanName + "': " + element);  
 }  
 element.inject(target, beanName, pvs);  
 }  
 }  
}

- 3.3.4.3.1.1) AutowiredFieldElement#inject  
- protected void inject(Object bean, String beanName, PropertyValues pvs) throws Throwable {

Field field = (Field) this.member;  
 try {  
 Object value;  
 if (this.cached) {  
 value = resolvedCachedArgument(beanName, this.cachedFieldValue);  
 }  
 else {  
 DependencyDescriptor desc = new DependencyDescriptor(field, this.required);  
 desc.setContainingClass(bean.getClass());  
 Set autowiredBeanNames = new LinkedHashSet(1);  
 TypeConverter typeConverter = beanFactory.getTypeConverter();  
 - // 3.3.4.2.1  
 value = beanFactory.resolveDependency(desc, beanName, autowiredBeanNames, typeConverter);  
 synchronized (this) {  
 if (!this.cached) {  
 if (value != null || this.required) {  
 this.cachedFieldValue = desc;  
 registerDependentBeans(beanName, autowiredBeanNames);  
 if (autowiredBeanNames.size() == 1) {  
 String autowiredBeanName = autowiredBeanNames.iterator().next();  
 if (beanFactory.containsBean(autowiredBeanName)) {  
 if (beanFactory.isTypeMatch(autowiredBeanName, field.getType())) {  
 this.cachedFieldValue = new RuntimeBeanReference(autowiredBeanName);  
 }  
 }  
 }  
 }  
 else {  
 this.cachedFieldValue = null;  
 }  
 this.cached = true;  
 }  
 }  
 }  
 if (value != null) {  
 ReflectionUtils.makeAccessible(field);  
 field.set(bean, value);  
 }  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException("Could not autowire field: " + field, ex);  
 }  
}

- 3.3.4.4) applyPropertyValues（注入属性值)

* AbstractAutowireCapableBeanFactory
* protected void applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper bw, PropertyValues pvs) {  
  if (pvs == null || pvs.isEmpty()) {  
   return;  
  }
* MutablePropertyValues mpvs = null;  
  List original;
* if (System.getSecurityManager() != null) {  
   if (bw instanceof BeanWrapperImpl) {  
   ((BeanWrapperImpl) bw).setSecurityContext(getAccessControlContext());  
   }  
  }
* if (pvs instanceof MutablePropertyValues) {  
   mpvs = (MutablePropertyValues) pvs;
* // 如果mpvs中的值已经被转换为对应的类型，则可以直接设置到beanwrapper中  
   if (mpvs.isConverted()) {  
   // Shortcut: use the pre-converted values as-is.  
   try {  
   bw.setPropertyValues(mpvs);  
   return;  
   }  
   catch (BeansException ex) {  
   throw new BeanCreationException(  
   mbd.getResourceDescription(), beanName, "Error setting property values", ex);  
   }  
   }  
   original = mpvs.getPropertyValueList();  
   }  
   else {
* // 如果pvs并不是使用MutablePropertyValues封装的类型，那么直接使用原始的属性获取方法  
   original = Arrays.asList(pvs.getPropertyValues());  
   }
* TypeConverter converter = getCustomTypeConverter();  
   if (converter == null) {  
   converter = bw;  
   }
* // 获取对应的解析器  
  BeanDefinitionValueResolver valueResolver = new BeanDefinitionValueResolver(this, beanName, mbd, converter);
* // Create a deep copy, resolving any references for values.  
  List deepCopy = new ArrayList(original.size());  
  boolean resolveNecessary = false;
* // 遍历属性，将属性转换为对应类的对应属性的类型  
  for (PropertyValue pv : original) {  
   if (pv.isConverted()) {  
   deepCopy.add(pv);  
   }  
   else {  
   String propertyName = pv.getName();  
   Object originalValue = pv.getValue();  
   Object resolvedValue = valueResolver.resolveValueIfNecessary(pv, originalValue);  
   Object convertedValue = resolvedValue;  
   boolean convertible = bw.isWritableProperty(propertyName) &&  
   !PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);  
   if (convertible) {  
   convertedValue = convertForProperty(resolvedValue, propertyName, bw, converter);  
   }  
   // Possibly store converted value in merged bean definition,  
   // in order to avoid re-conversion for every created bean instance.  
   if (resolvedValue == originalValue) {  
   if (convertible) {  
   pv.setConvertedValue(convertedValue);  
   }  
   deepCopy.add(pv);  
   }  
   else if (convertible && originalValue instanceof TypedStringValue &&  
   !((TypedStringValue) originalValue).isDynamic() &&  
   !(convertedValue instanceof Collection || ObjectUtils.isArray(convertedValue))) {  
   pv.setConvertedValue(convertedValue);  
   deepCopy.add(pv);  
   }  
   else {  
   resolveNecessary = true;  
   deepCopy.add(new PropertyValue(pv, convertedValue));  
   }  
   }  
  }  
  if (mpvs != null && !resolveNecessary) {  
   mpvs.setConverted();  
  }
* // Set our (possibly massaged) deep copy.  
  try {  
   bw.setPropertyValues(new MutablePropertyValues(deepCopy));  
  }  
  catch (BeansException ex) {  
   throw new BeanCreationException(  
   mbd.getResourceDescription(), beanName, "Error setting property values", ex);  
  }  
  }
  + 3.3.5) initializeBean（调用init-method方法)
* AbstractAutowireCapableBeanFactory
* 主要是调用用户设定的初始化方法，还有一些其他工作

protected Object initializeBean(final String beanName, final Object bean, RootBeanDefinition mbd) {  
 if (System.getSecurityManager() != null) {  
 AccessController.doPrivileged(new PrivilegedAction<Object>() {  
 @Override  
 public Object run() {

* // 激活Aware方法  
   invokeAwareMethods(beanName, bean);  
   return null;  
   }  
  }, getAccessControlContext());  
  }  
  else {  
   invokeAwareMethods(beanName, bean);  
  }
* Object wrappedBean = bean;  
  if (mbd == null || !mbd.isSynthetic()) {
* // 应用后处理器  
   wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);  
   }
* try {
* // 激活用户自定义的init方法  
   invokeInitMethods(beanName, wrappedBean, mbd);  
   }  
   catch (Throwable ex) {  
   throw new BeanCreationException(  
   (mbd != null ? mbd.getResourceDescription() : null),  
   beanName, "Invocation of init method failed", ex);  
   }
* if (mbd == null || !mbd.isSynthetic()) {  
   wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);  
   }  
   return wrappedBean;  
  }
  + 3.3.5.1) invokeAwareMethods
* Aware
* Spring中提供一些Aware相关接口，比如BeanFactoryAware、ApplicationContextAware等，实现这些Aware接口的bean被初始化后，可以取得一些相对应的资源。
* 如实现BeanFactoryAware的bean在初始化后，Spring容器将会注入BeanFactory的实例。

private void invokeAwareMethods(final String beanName, final Object bean) {  
 if (bean instanceof Aware) {  
 if (bean instanceof BeanNameAware) {  
 ((BeanNameAware) bean).setBeanName(beanName);  
 }  
 if (bean instanceof BeanClassLoaderAware) {  
 ((BeanClassLoaderAware) bean).setBeanClassLoader(getBeanClassLoader());  
 }  
 if (bean instanceof BeanFactoryAware) {  
 ((BeanFactoryAware) bean).setBeanFactory(AbstractAutowireCapableBeanFactory.this);  
 }  
 }  
}

- 3.3.5.2) BeanPostProcessor

* 调用用户自定义初始化方法之前和之后分别会调用BeanPostProcessor的postProcessBeforeInitialization和postProcessAfterInitialization方法，使 用户可以根据自己的业务需求进行响应的处理。

public Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, String beanName)  
 throws BeansException {  
  
 Object result = existingBean;  
 for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {  
 result = beanProcessor.postProcessBeforeInitialization(result, beanName);  
 if (result == null) {  
 return result;  
 }  
 }  
 return result;  
}

public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)  
 throws BeansException {  
  
 Object result = existingBean;  
 for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {  
 result = beanProcessor.postProcessAfterInitialization(result, beanName);  
 if (result == null) {  
 return result;  
 }  
 }  
 return result;  
}

- 3.3.5.2.1) @PostConstructor

* 当使用该注解时，Spring会去注册一个BeanPostProcessor：InitDestroyAnnotationBeanPostProcessor。该bean会同自定义的BeanPostProcessor一样，在自定义初始化方法之前和之后被调用（当然@PostConstrcut只会在之前被调用)。
* InitDestroyAnnotationBeanPostProcessor#postProcessBeforeInitialization

public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  
 LifecycleMetadata metadata = findLifecycleMetadata(bean.getClass());  
 try {  
 metadata.invokeInitMethods(bean, beanName);  
 }  
 catch (InvocationTargetException ex) {  
 throw new BeanCreationException(beanName, "Invocation of init method failed", ex.getTargetException());  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(beanName, "Couldn't invoke init method", ex);  
 }  
 return bean;  
}

- 3.3.5.2.1.1) findLifecycleMetadata

private LifecycleMetadata findLifecycleMetadata(Class<?> clazz) {  
 if (this.lifecycleMetadataCache == null) {  
 // Happens after deserialization, during destruction...  
 return buildLifecycleMetadata(clazz);  
 }  
 // Quick check on the concurrent map first, with minimal locking.  
 LifecycleMetadata metadata = this.lifecycleMetadataCache.get(clazz);  
 if (metadata == null) {  
 synchronized (this.lifecycleMetadataCache) {  
 metadata = this.lifecycleMetadataCache.get(clazz);  
 if (metadata == null) {  
 metadata = buildLifecycleMetadata(clazz);  
 this.lifecycleMetadataCache.put(clazz, metadata);  
 }  
 return metadata;  
 }  
 }  
 return metadata;  
}

private LifecycleMetadata buildLifecycleMetadata(final Class<?> clazz) {  
 final boolean debug = logger.isDebugEnabled();  
 LinkedList<LifecycleElement> initMethods = new LinkedList<LifecycleElement>();  
 LinkedList<LifecycleElement> destroyMethods = new LinkedList<LifecycleElement>();  
 Class<?> targetClass = clazz;  
  
 do {  
 final LinkedList<LifecycleElement> currInitMethods = new LinkedList<LifecycleElement>();  
 final LinkedList<LifecycleElement> currDestroyMethods = new LinkedList<LifecycleElement>();  
  
 ReflectionUtils.doWithLocalMethods(targetClass, new ReflectionUtils.MethodCallback() {  
 @Override  
 public void doWith(Method method) throws IllegalArgumentException, IllegalAccessException {  
 if (initAnnotationType != null) {  
 if (method.getAnnotation(initAnnotationType) != null) {  
 LifecycleElement element = new LifecycleElement(method);  
 currInitMethods.add(element);  
 if (debug) {  
 logger.debug("Found init method on class [" + clazz.getName() + "]: " + method);  
 }  
 }  
 }  
 if (destroyAnnotationType != null) {  
 if (method.getAnnotation(destroyAnnotationType) != null) {  
 currDestroyMethods.add(new LifecycleElement(method));  
 if (debug) {  
 logger.debug("Found destroy method on class [" + clazz.getName() + "]: " + method);  
 }  
 }  
 }  
 }  
 });  
  
 initMethods.addAll(0, currInitMethods);  
 destroyMethods.addAll(currDestroyMethods);  
 targetClass = targetClass.getSuperclass();  
 }  
 while (targetClass != null && targetClass != Object.class);  
  
 return new LifecycleMetadata(clazz, initMethods, destroyMethods);  
}

- 3.3.5.2.1.2) invokeInitMethods

public void invokeInitMethods(Object target, String beanName) throws Throwable {  
 Collection<LifecycleElement> initMethodsToIterate =  
 (this.checkedInitMethods != null ? this.checkedInitMethods : this.initMethods);  
 if (!initMethodsToIterate.isEmpty()) {  
 boolean debug = logger.isDebugEnabled();  
 for (LifecycleElement element : initMethodsToIterate) {  
 if (debug) {  
 logger.debug("Invoking init method on bean '" + beanName + "': " + element.getMethod());  
 }  
 element.invoke(target);  
 }  
 }  
}

##### 3.3.5.3) invokeInitMethods（激活自定义的init方法)

* 客户定制的初始化方法除了使用配置init-method外，还可以使自定义的bean实现InitializingBean接口,并在afterPropertiesSet中实现自己的初始化业务逻辑。
* init-method和afterPropertiesSet都是在初始化bean时执行，执行顺序是afterPropertiesSet先执行，init-method后执行。
* 该方法中实现了这两个步骤的初始化方法调用。

protected void invokeInitMethods(String beanName, final Object bean, RootBeanDefinition mbd)  
 throws Throwable {  
 // 首先检查是否是InitializingBean，如果是的话需要调用afterPropertiesSet方法  
 boolean isInitializingBean = (bean instanceof InitializingBean);  
 if (isInitializingBean && (mbd == null || !mbd.isExternallyManagedInitMethod("afterPropertiesSet"))) {  
 if (logger.isDebugEnabled()) {  
 logger.debug("Invoking afterPropertiesSet() on bean with name '" + beanName + "'");  
 }  
 if (System.getSecurityManager() != null) {  
 try {  
 AccessController.doPrivileged(new PrivilegedExceptionAction<Object>() {  
 @Override  
 public Object run() throws Exception {  
 ((InitializingBean) bean).afterPropertiesSet();  
 return null;  
 }  
 }, getAccessControlContext());  
 }  
 catch (PrivilegedActionException pae) {  
 throw pae.getException();  
 }  
 }  
 else {  
 ((InitializingBean) bean).afterPropertiesSet();  
 }  
 }  
  
 if (mbd != null) {  
 String initMethodName = mbd.getInitMethodName();  
 if (initMethodName != null && !(isInitializingBean && "afterPropertiesSet".equals(initMethodName)) &&  
 !mbd.isExternallyManagedInitMethod(initMethodName)) {  
 invokeCustomInitMethod(beanName, bean, mbd);  
 }  
 }  
}

* + 3.3.5.3.1) invokeCustomInitMethod

protected void invokeCustomInitMethod(String beanName, final Object bean, RootBeanDefinition mbd)  
 throws Throwable {  
  
 String initMethodName = mbd.getInitMethodName();  
 final Method initMethod = (mbd.isNonPublicAccessAllowed() ?  
 BeanUtils.findMethod(bean.getClass(), initMethodName) :  
 ClassUtils.getMethodIfAvailable(bean.getClass(), initMethodName));  
 if (initMethod == null) {  
 if (mbd.isEnforceInitMethod()) {  
 throw new BeanDefinitionValidationException("Couldn't find an init method named '" +  
 initMethodName + "' on bean with name '" + beanName + "'");  
 }  
 else {  
 if (logger.isDebugEnabled()) {  
 logger.debug("No default init method named '" + initMethodName +  
 "' found on bean with name '" + beanName + "'");  
 }  
 // Ignore non-existent default lifecycle methods.  
 return;  
 }  
 }  
  
 if (logger.isDebugEnabled()) {  
 logger.debug("Invoking init method '" + initMethodName + "' on bean with name '" + beanName + "'");  
 }  
  
 if (System.getSecurityManager() != null) {  
 AccessController.doPrivileged(new PrivilegedExceptionAction<Object>() {  
 @Override  
 public Object run() throws Exception {  
 ReflectionUtils.makeAccessible(initMethod);  
 return null;  
 }  
 });  
 try {  
 AccessController.doPrivileged(new PrivilegedExceptionAction<Object>() {  
 @Override  
 public Object run() throws Exception {  
 initMethod.invoke(bean);  
 return null;  
 }  
 }, getAccessControlContext());  
 }  
 catch (PrivilegedActionException pae) {  
 InvocationTargetException ex = (InvocationTargetException) pae.getException();  
 throw ex.getTargetException();  
 }  
 }  
 else {  
 try {  
 ReflectionUtils.makeAccessible(initMethod);  
 initMethod.invoke(bean);  
 }  
 catch (InvocationTargetException ex) {  
 throw ex.getTargetException();  
 }  
 }  
}

* + 3.3.6) getSingleton(beanName,allowEarlyReference)
* protected Object getSingleton(String beanName, boolean allowEarlyReference) {  
  Object singletonObject = this.singletonObjects.get(beanName);  
  if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {  
   synchronized (this.singletonObjects) {  
   singletonObject = this.earlySingletonObjects.get(beanName);  
   if (singletonObject == null && allowEarlyReference) {  
   ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);  
   if (singletonFactory != null) {  
   singletonObject = singletonFactory.getObject();  
   this.earlySingletonObjects.put(beanName, singletonObject);  
   this.singletonFactories.remove(beanName);  
   }  
   }  
   }  
  }  
  return (singletonObject != NULL\_OBJECT ? singletonObject : null);  
  }
  + 3.3.7) registerDisposableBeanIfNecessary（注册DisposableBean)
* 对于销毁方法的扩展，除了配置属性destroy-method，用户还可以注册后处理器DestructionAwareBeanPostProcessor来统一处理bean的销毁方法。
* protected void registerDisposableBeanIfNecessary(String beanName, Object bean, RootBeanDefinition mbd) {  
  AccessControlContext acc = (System.getSecurityManager() != null ? getAccessControlContext() : null);  
  if (!mbd.isPrototype() && requiresDestruction(bean, mbd)) {  
   if (mbd.isSingleton()) {
* // 单例模式下，注册需要销毁的bean，此方法中会处理实现DisposableBean的bean，并且对所有的bean使用DestructionAwareBeanPostProcessors处理  
   // Register a DisposableBean implementation that performs all destruction  
   // work for the given bean: DestructionAwareBeanPostProcessors,  
   // DisposableBean interface, custom destroy method.  
   registerDisposableBean(beanName,  
   new DisposableBeanAdapter(bean, beanName, mbd, getBeanPostProcessors(), acc));  
   }  
   else {
* // 自定义scope的处理  
   // A bean with a custom scope...  
   Scope scope = this.scopes.get(mbd.getScope());  
   if (scope == null) {  
   throw new IllegalStateException("No Scope registered for scope name '" + mbd.getScope() + "'");  
   }  
   scope.registerDestructionCallback(beanName,  
   new DisposableBeanAdapter(bean, beanName, mbd, getBeanPostProcessors(), acc));  
   }  
  }  
  }
  + 4) getObjectForBeanInstance（从bean 的实例中获取对象)
* 无论是从缓存中获取到的bean还是通过不同的scope策略加载的bean都只是最原始的bean状态，并不一定是我们最终想要的bean。
* 比如，我们需要对FactoryBean进行处理，那么这里得到的其实是FactoryBean的初始状态，但是我们真正需要的是FactoryBean中定义的factory-method（getObject方法)方法中返回的bean，而getObjectForBeanInstance就是完成这个工作的。
* 下面这个方法完成了以下任务：
  + 1)对FactoryBean正确性的验证
  + 2)对非FactoryBean不做任何处理
  + 3)对bean进行转换
  + 4)将从Factory解析bean的工作委托给getObjectFromFactoryBean。
* protected Object getObjectForBeanInstance(  
   Object beanInstance, String name, String beanName, RootBeanDefinition mbd) {
* // Don't let calling code try to dereference the factory if the bean isn't a factory.  
   if (BeanFactoryUtils.isFactoryDereference(name) && !(beanInstance instanceof FactoryBean)) {  
   throw new BeanIsNotAFactoryException(transformedBeanName(name), beanInstance.getClass());  
   }
* // Now we have the bean instance, which may be a normal bean or a FactoryBean.  
   // If it's a FactoryBean, we use it to create a bean instance, unless the  
   // caller actually wants a reference to the factory.  
   if (!(beanInstance instanceof FactoryBean) || BeanFactoryUtils.isFactoryDereference(name)) {  
   return beanInstance;  
   }
* Object object = null;  
   if (mbd == null) {  
   object = getCachedObjectForFactoryBean(beanName);  
   }  
   if (object == null) {  
   // Return bean instance from factory.  
   FactoryBean) beanInstance;  
   // Caches object obtained from FactoryBean if it is a singleton.  
   if (mbd == null && containsBeanDefinition(beanName)) {  
   mbd = getMergedLocalBeanDefinition(beanName);  
   }  
   boolean synthetic = (mbd != null && mbd.isSynthetic());  
   object = getObjectFromFactoryBean(factory, beanName, !synthetic);  
   }  
   return object;  
  }
  + 4.1) getObjectFromFactoryBean（从FactoryBean中解析bean)
* 返回的bean如果是单例的，那就必须要保证全局唯一，同时，也因为是单例的，所以不必重复创建，可以使用缓存来提高性能。
* protected Object getObjectFromFactoryBean(FactoryBean<?> factory, String beanName, boolean shouldPostProcess) {  
  if (factory.isSingleton() && containsSingleton(beanName)) {  
   synchronized (getSingletonMutex()) {  
   Object object = this.factoryBeanObjectCache.get(beanName);  
   if (object == null) {  
   object = doGetObjectFromFactoryBean(factory, beanName);  
   // Only post-process and store if not put there already during getObject() call above  
   // (e.g. because of circular reference processing triggered by custom getBean calls)  
   Object alreadyThere = this.factoryBeanObjectCache.get(beanName);  
   if (alreadyThere != null) {  
   object = alreadyThere;  
   }  
   else {  
   if (object != null && shouldPostProcess) {  
   try {  
   object = postProcessObjectFromFactoryBean(object, beanName);  
   }  
   catch (Throwable ex) {  
   throw new BeanCreationException(beanName,  
   "Post-processing of FactoryBean's singleton object failed", ex);  
   }  
   }  
   this.factoryBeanObjectCache.put(beanName, (object != null ? object : NULL*OBJECT));*  
   *}*  
   *}*  
   *return (object != NULL*OBJECT ? object : null);  
   }  
  }  
  else {  
   Object object = doGetObjectFromFactoryBean(factory, beanName);  
   if (object != null && shouldPostProcess) {  
   try {  
   object = postProcessObjectFromFactoryBean(object, beanName);  
   }  
   catch (Throwable ex) {  
   throw new BeanCreationException(beanName, "Post-processing of FactoryBean's object failed", ex);  
   }  
   }  
   return object;  
  }  
  }
  + 4.1.1) doGetObjectFromFactoryBean

private Object doGetObjectFromFactoryBean(final FactoryBean<?> factory, final String beanName)  
 throws BeanCreationException {  
  
 Object object;  
 try {  
 if (System.getSecurityManager() != null) {  
 AccessControlContext acc = getAccessControlContext();  
 try {  
 object = AccessController.doPrivileged(new PrivilegedExceptionAction<Object>() {  
 @Override  
 public Object run() throws Exception {  
 return factory.getObject();  
 }  
 }, acc);  
 }  
 catch (PrivilegedActionException pae) {  
 throw pae.getException();  
 }  
 }  
 else {  
 object = factory.getObject();  
 }  
 }  
 catch (FactoryBeanNotInitializedException ex) {  
 throw new BeanCurrentlyInCreationException(beanName, ex.toString());  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(beanName, "FactoryBean threw exception on object creation", ex);  
 }  
  
 // Do not accept a null value for a FactoryBean that's not fully  
 // initialized yet: Many FactoryBeans just return null then.  
 if (object == null && isSingletonCurrentlyInCreation(beanName)) {  
 throw new BeanCurrentlyInCreationException(  
 beanName, "FactoryBean which is currently in creation returned null from getObject");  
 }  
 return object;  
}

* 循环依赖
* Spring容器循环依赖包括构造器循环依赖和setter循环依赖。
* 核心就是singletonObjects、singletonFactories、earlySingletonObjects和singletonsCurrentlyInCreation四个集合。
  + 1)singletonObjects是beanName与beanInstance的Map，是真正的缓存，beanInstance是构造完毕的，凡是正常地构造完毕的单例bean都会放入缓存中。
  + 2)earlySingletonObjects也是beanName与beanInstance的Map，beanInstance是已经调用了createBean方法，但是没有清除加载状态和加入至缓存的bean。仅在当前bean创建时存在，用于检测代理bean循环依赖。
  + 3)singleFactories是beanName与ObjectFactory的Map，仅在当前bean创建时存在，是尚未调用createBean的bean。用于setter循环依赖时实现注入。
  + 4)singletonsCurrentlyInCreation是beanName的集合，用于检测构造器循环依赖。
* getBean在循环依赖时所执行的步骤是这样的：
  + 1)检测当前bean是否在singletonObjects中，在则直接返回缓存好的bean；不在则检测是否在singletonFactories中，在，则调用其getObject方法，返回，并从singletonFactories中移除，加入到earlySingletonObjects中。
  + 2)正常创建，beforeSingletonCreation:检测当前bean是否在singletonsCurrentlyInCreation，如果存在，抛出异常。表示存在构造器循环依赖。如果不存在，则将当前bean加入。
  + 3)bean初始化，分为构造方法初始化、工厂方法初始化和简单初始化。如果是构造方法初始化，那么递归地获取参数bean。其他情况不会递归获取bean。
  + 4)addSingletonFactory:如果当前bean不在singletonObjects中，则将当前bean加入到singletonFactories中（getObject方法是getEarlyBeanReference)，并从earlySingletonObjects中移除。
  + 5)填充属性，简单初始化的话会递归创建所依赖的bean。
  + 6)调用用户初始化方法，比如BeanPostProcesser、InitializingBean、init-method，有可能返回代理后的bean。
  + 6) 检测循环依赖，如果当前bean在singletonObjects中，则判断当前bean(current bean)与singletonObjects中的bean(cached bean)是否是同一个，如果不是，那么说明当前bean是被代理过的，由于依赖当前bean的bean持有的是对cached bean的引用，这是不被允许的，所以会抛出BeanCurrentlyInCreationException异常。
  + 7)afterSingletonCreation:将当前bean从singletonsCurrentlyInCreation中删除
  + 8)addSingleton:将当前bean加入到singletonObjects，然后从singletonFactories, earlySingletonObjects中移除，结束
* 构造器循环依赖
* 表示通过构造器注入构成的循环依赖，此依赖是无法解决的，只能抛出BeanCurrentlyInCreationException异常表示循环依赖。
* 1、Spring容器创建单例“A” Bean，首先检测singletonFactories是否包含A，发现没有，于是正常创建，然后检测A是否包含在singletonsCurrentlyInCreation中，没有，则将A放入。构造方法初始化时需要B实例（A尚未放入到singletonFactories中)，于是调用了getBean(B)方法、
* 2、Spring容器创建单例“B” Bean，首先检测singletonFactories是否包含B，发现没有，于是正常创建，然后检测B是否包含在singletonsCurrentlyInCreation中，没有，则将B放入。构造方法初始化时需要C实例（B尚未放入到singletonFactories中)，于是调用了getBean(C)方法、
* 3、Spring容器创建单例“C” Bean，首先检测singletonFactories是否包含C，发现没有，于是正常创建，然后检测C是否包含在singletonsCurrentlyInCreation中，没有，则将C放入。构造方法初始化时需要A实例（C尚未放入到singletonFactories中)，于是调用了getBean(A)方法、
* 4、Spring容器创建单例“A” Bean，首先检测singletonFactories是否包含A，发现没有于是正常创建，然后检测A是否包含在singletonsCurrentlyInCreation中，有，抛出BeanCurrentlyInCreationException异常。
* setter循环依赖
* 表示通过setter注入方式构成的循环依赖。
* 对于setter注入造成的依赖是通过Spring容器提前暴露刚完成构造器注入但未完成其他步骤（如setter注入)的Bean来完成的，而且只能解决单例作用域的Bean循环依赖。
* 1、Spring容器创建单例“A” Bean，首先检测singletonFactories是否包含A，发现没有，于是正常创建，然后检测A是否包含在singletonsCurrentlyInCreation中，没有，则将A放入。注入属性时需要B实例，于是调用了getBean(B)方法、
* 2、Spring容器创建单例“B” Bean，首先检测singletonFactories是否包含B，发现没有，于是正常创建，然后检测B是否包含在singletonsCurrentlyInCreation中，没有，则将B放入。注入属性时需要C实例，于是调用了getBean(C)方法、
* 3、Spring容器创建单例“C” Bean，首先检测singletonFactories是否包含C，发现没有，于是正常创建，然后检测C是否包含在singletonsCurrentlyInCreation中，没有，则将C放入。注入属性时需要A实例，于是调用了getBean(A)方法、
* 4、Spring容器创建单例“A” Bean，首先检测singletonFactories是否包含A，发现有，于是返回缓存了的bean，并将A从singletonFactories删除，返回A实例。
* 5、C得到A实例。set进来，B、A也是这样。结束。
* 对于“prototype”作用域Bean，Spring容器无法完成依赖注入，因为“prototype”作用域的Bean，Spring容器不进行缓存，因此无法提前暴露一个创建中的Bean。
* 实例

public class Main {  
 public static void main(String[] args) {  
 ApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");  
 UserService userService = (UserService) applicationContext.getBean("userService");  
 userService.login();  
 }  
}

# 实例——循环依赖

## 注解注入

public class A {  
 @Autowired  
 private B b;  
}

public class B {  
 @Autowired  
 private C c;  
}

public class C {  
 @Autowired  
 private A a;  
}

- 1)创建A时调用beforeSingletonCreation  
  
- 2)调用doCreateBean时earlySingletonExposure为true，调用了addSingletonFactory

* 注意，并没有递归去初始化B，返回它所依赖的bean时得到的是null。

- 3)然后调用populateBean，在postProcesser处理时调用了  
- AutowiredAnnotationBeanPostProcesser#postProcessPropertyValues，最终调用了3.3.4.2.1.1.1) findAutowireCandidates，它调用了A所依赖的B的getBean方法。  
- 4)B的getBean时调用了beforeSingletonCreation，  
  
- 5)调用doCreateBean时earlySingletonExposure为true，调用了addSingletonFactory

* 注意，并没有递归去初始化C，返回它所依赖的bean时得到的是null。
  + 6)然后调用populateBean，在postProcesser处理时调用了
  + AutowiredAnnotationBeanPostProcesser#postProcessPropertyValues，最终调用了3.3.4.2.1.1.1) findAutowireCandidates，它调用了B所依赖的C的getBean方法。
  + 7)B的getBean时调用了beforeSingletonCreation，
  + 8)调用doCreateBean时earlySingletonExposure为true，调用了addSingletonFactory
* 注意，并没有递归去初始化A，返回它所依赖的bean时得到的是null。
  + 9)然后调用populateBean，在postProcesser处理时调用了
  + AutowiredAnnotationBeanPostProcesser#postProcessPropertyValues，最终调用了3.3.4.2.1.1.1) findAutowireCandidates，它调用了C所依赖的A的getBean方法。
* 10)A在调用getSingleton时发现singletonFactories存在CircleA，然后调用其getObject方法（调用了getEarlyReference)，之后将CircleA放入earlySingletonObjects，然后从singletonFactories中移除。

- 11)从getBean(CircleA)中返回，回到CircleC的findAutowireCandidates，然后带着CircleA实例回到了inject方法，将CircleA实例设置给了CircleC。  
- 12)CircleC去检测循环依赖，没有循环依赖，返回bean。  
- 13)在调用getSingleton时发现singletonFactories存在CircleA，然后调用其getObject方法（调用了getEarlyReference)，之后将CircleA放入earlySingletonObjects，然后从singletonFactories中移除。  
- 13)从getBean(CircleC)中返回，回到CircleB的findAutowireCandidates，然后带着CircleC实例回到了inject方法，将CircleC实例设置给了CircleB。  
- 14)CircleB去检测循环依赖，没有循环依赖，返回bean。  
- 15)从getBean(CircleB)中返回，回到CircleA的findAutowireCandidates，然后带着CircleB实例回到了inject方法，将CircleB实例设置给了CircleA。  
- 16)CircleA去检测循环依赖，没有循环依赖，返回bean。结束。

## setter注入

public class A {  
 private B b;  
   
 @Autowired  
 public void setB(B b) {  
 this.b = b;  
 }  
}

public class B {  
 private C c;  
 @Autowired  
 public C getC() {  
 return c;  
 }  
}

public class C {  
 private A a;  
  
 @Autowired  
 public void setA(A a) {  
 this.a = a;  
 }  
}

* 调用栈与注解注入一致

## 构造器注入（抛出BeanCurrentlyInCreationException异常)

public class A {  
 private B b;  
 @Autowired  
 public A(B b) {  
 this.b = b;  
 }  
}

* 步骤类似，只有两个地方不同。
  + 1)调用doCreateBean时，会采用构造方法初始化的方式，此时会递归地初始化构造方法参数bean。因为是构造方法初始化，所以递归获取参数bean是在将自己放入singletonFactories之前。
  + 2)正因为没有将自己放入singletonFactories，所以不会在getBean从singletonFactories返回已经创建过的bean。

## 存在代理时的循环依赖（抛出BeanCurrentlyInCreationException异常)

@Component("CircleA")  
public class A {  
 @Autowired  
 private B b;  
}

@Component("CircleB")  
public class B {  
 @Autowired  
 private A a;  
}

@Component  
public class ABeanPostProcessor implements BeanPostProcessor {  
 @Override  
 public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  
 if(beanName.equals("CircleA")) {  
 System.out.println("proxy-A");  
 return new Object();  
 }  
 return bean;  
 }  
  
 @Override  
 public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
 return bean;  
 }  
}

* <context:component-scan base-package="proxycircle" ></context:component-scan>
* Exception in thread "main"
* org.springframework.beans.factory.BeanCurrentlyInCreationException: Error creating bean with name 'CircleA': Bean with name 'CircleA' has been injected into other beans [CircleB] in its raw version as part of a circular reference, but has eventually been wrapped. This means that said other beans do not use the final version of the bean. This is often the result of over-eager type matching - consider using 'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.
* 此时就与earlySIngletonObjects有关系了。
  + 1)getBean(A)时，将A加入了singletonFactories，注入属性时setter注入B，调用getBean(B)
  + 2)调用getBean(B)时，将B加入了singletonFactories，注入属性时setter注入A，调用getBean(A)
  + 3)因为A已经存在在singletonFactories，于是取出，调用getObject，然后将A加入到earlySIngletonObjects，返回A。
  + 4)B注入属性A完毕后，B构造完毕，将B加入singletonObjects，从 earlySIngletonObjects和singletonFactories中移除B
  + 5)A注入属性B完毕后，执行BeanPostProcessor，此时A变为了Object（CurrentA)。检测代理bean循环依赖，发现singletonObjects中存在Cached A，于是取出，将CachedA 与 CurrentA比较，发现不同，然后发现有B依赖着Cached A，数据发生不一致，抛出异常。
* https://www.iflym.com/index.php/code/201208280003.html
* Spring AOP（AspectJ)
* AOP术语
* 1、切面（aspect/advisor)
* 类是对物体特征的抽象，切面就是对横切关注点的抽象。组合了Pointcut与Advice，在Spring中有时候也称为Advisor。
* 2、连接点（join point)
* 被拦截到的点，因为Spring只支持方法类型的连接点，所以在Spring中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器。
* 3、切入点（pointcut)
* 描述的一组符合某个条件的join point，通常使用pointcut表达式来限定join point。
* 4、通知（advice)
* 所谓通知指的就是指拦截到连接点之后要执行的代码，通知分为前置、后置、异常、返回、环绕通知五类。
* 5、目标对象
* 代理的目标对象
* 6、织入（weave)
* 将Advice织入join point的这个过程
* 7、引介（introduction)
* 在不修改代码的前提下，引介可以在运行期为类动态地添加一些方法或字段
* Advisor
* 通知Advice是Spring提供的一种切面(Aspect)。但其功能过于简单，只能将切面织入到目标类的所有目标方法中，无法完成将切面织入到指定目标方法中。
* 顾问Advisor是Spring提供的另一种切面。其可以完成更为复杂的切面织入功能。PointcutAdvisor是顾问的一种，可以指定具体的切入点。顾问对通知进行了包装，会根据不同的通知类型，在不同的时间点，将切面织入到不同的切入点。
* Advisor组合了Pointcut与Advice。
* 除了引介Advisor外，几乎所有的advisor都是PointcutAdvisor。

public interface Advisor {  
 Advice getAdvice();  
 /\*\*  
 \* @return whether this advice is associated with a particular target instance  
 \*/  
 boolean isPerInstance();  
}

public interface PointcutAdvisor extends Advisor {  
  
 /\*\*  
 \* Get the Pointcut that drives this advisor.  
 \*/  
 Pointcut getPointcut();  
  
}

* Advice

public interface Advice {  
  
}

* 增强（advice)主要包括如下五种类型
  1. 前置增强(BeforeAdvice)：在目标方法执行前实施增强
  2. 后置增强(AfterAdvice)：在目标方法执行后（无论是否抛出遗产)实施增强
  3. 环绕增强(MethodInterceptor)：在目标方法执行前后实施增强
  4. 异常抛出增强(ThrowsAdvice)：在目标方法抛出异常后实施增强
  5. 返回增强（AfterReturningAdvice)：在目标方法正常返回后实施增强
  6. 引介增强(IntroductionIntercrptor)：在目标类中添加一些新的方法和属性
* BeanPostProcessor

public interface BeanPostProcessor {  
 Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;  
 Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException;  
}

* JDK动态代理与CGLIB代理
* JDK动态代理：
* 其代理对象必须是某个接口的实现，它是通过在运行时创建一个接口的实现类来完成对目标对象的代理
* CGLIB代理：在运行时生成的代理对象是针对目标类扩展的子类。
* CGLIB是高效的代码生产包，底层是依靠ASM操作字节码实现的，性能比JDK强。
* 相关标签
* <aop:aspectj-autoproxy proxy-target-class=”true”/>
* true表示使用CGLIB代理。
* 解析AOP标签
* <aop:aspectj-autoproxy />
* 解析配置文件时，一旦遇到aspectj-autoproxy注解时就会使用解析器
* AspectJAutoProxyBeanDefinitionParser进行解析。
* 解析结果是注册了一个bean：AnnotationAwareAspectJAutoProxyCreator。
* 与IOC的衔接
* 处理自定义标签

public BeanDefinition parseCustomElement(Element ele, BeanDefinition containingBd) {  
 String namespaceUri = getNamespaceURI(ele);

* // AopNamespaceHandler  
  NamespaceHandler handler = this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri);  
  if (handler == null) {  
   error("Unable to locate Spring NamespaceHandler for XML schema namespace [" + namespaceUri + "]", ele);  
   return null;  
  }  
  return handler.parse(ele, new ParserContext(this.readerContext, this, containingBd));  
  }
* NamespaceHandler#parse
* NamespaceHandler是一个接口，它有一个实现是NamespaceHandlerSupport，实现了它的parse方法，而AopNamespaceHandler直接继承了parse方法。
* 它继承自NamespaceHandlerSupport，实现了NamespaceHandler接口的parse方法，而AopNamespaceHandler直接继承该parse方法。
* NamespaceHandlerSupport#parse

public BeanDefinition parse(Element element, ParserContext parserContext) {  
 return findParserForElement(element, parserContext).parse(element, parserContext);  
}

private BeanDefinitionParser findParserForElement(Element element, ParserContext parserContext) {  
 String localName = parserContext.getDelegate().getLocalName(element);  
 BeanDefinitionParser parser = this.parsers.get(localName);  
 if (parser == null) {  
 parserContext.getReaderContext().fatal(  
 "Cannot locate BeanDefinitionParser for element [" + localName + "]", element);  
 }  
 return parser;  
}

* 这里返回的parser即为AspectJAutoProxyBeanDefinitionParser。
* AspectJAutoProxyBeanDefinitionParser#parse

public BeanDefinition parse(Element element, ParserContext parserContext) {  
 AopNamespaceUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(parserContext, element);  
 extendBeanDefinition(element, parserContext);  
 return null;  
}

* AopNamespaceUtils#registerAspectJAnnotationAutoProxyCreatorIfNecessary
* 注册这个creator

public static void registerAspectJAnnotationAutoProxyCreatorIfNecessary(  
 ParserContext parserContext, Element sourceElement) {  
 // 注册或升级AutoProxyCreator定义beanName为internalAutoProxyCreator的BeanDefinition  
 BeanDefinition beanDefinition = AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(  
 parserContext.getRegistry(), parserContext.extractSource(sourceElement));

* // 对于proxy-target-class以及expose-proxy属性的增强  
  useClassProxyingIfNecessary(parserContext.getRegistry(), sourceElement);
* // 注册组件并通知，便于监听器做进一步处理
* // 其中beanDefinition的className为AnnotationAwareAspectJAutoProxyCreator  
  registerComponentIfNecessary(beanDefinition, parserContext);  
  }
  + 1) AopConfigUtils#registerAspectJAnnotationAutoProxyCreatorIfNecessary

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

* registerOrEscalateApcAsRequired

private static BeanDefinition registerOrEscalateApcAsRequired(Class<?> cls, BeanDefinitionRegistry registry, Object source) {  
 Assert.notNull(registry, "BeanDefinitionRegistry must not be null");

* // 如果已经存在自动代理创建器，且存在的在自动代理创建器与现在的不一致，那么需要根据优先级来判断到底需要使用哪个  
  if (registry.containsBeanDefinition(AUTO*PROXY*CREATOR*BEAN*NAME)) {  
   BeanDefinition apcDefinition = registry.getBeanDefinition(AUTO*PROXY*CREATOR*BEAN*NAME);  
   if (!cls.getName().equals(apcDefinition.getBeanClassName())) {  
   int currentPriority = findPriorityForClass(apcDefinition.getBeanClassName());  
   int requiredPriority = findPriorityForClass(cls);  
   if (currentPriority < requiredPriority) {
* // 改变bean最重要的是改变bean所对应的className属性  
   apcDefinition.setBeanClassName(cls.getName());  
   }  
  }
* // 如果已经存在自动代理创建器，且与将要创建的一致，那么无需再次创建  
   return null;  
   }  
   RootBeanDefinition beanDefinition = new RootBeanDefinition(cls);  
   beanDefinition.setSource(source);  
   beanDefinition.getPropertyValues().add("order", Ordered.HIGHEST*PRECEDENCE);*  
   *beanDefinition.setRole(BeanDefinition.ROLE*INFRASTRUCTURE);  
   registry.registerBeanDefinition(AUTO*PROXY*CREATOR*BEAN*NAME, beanDefinition);  
   return beanDefinition;  
  }

public static final String AUTO\_PROXY\_CREATOR\_BEAN\_NAME =  
 "org.springframework.aop.config.internalAutoProxyCreator";

* 这里的registery.registerBeanDefinition即为
* DefaultListableBeanFactory.registerBeanDefinition。
  + 2) useClassProxyingIfNecessary
* 处理proxy-target-class属性和expose-proxy属性

private static void useClassProxyingIfNecessary(BeanDefinitionRegistry registry, Element sourceElement) {  
 if (sourceElement != null) {  
 boolean proxyTargetClass = Boolean.valueOf(sourceElement.getAttribute(PROXY\_TARGET\_CLASS\_ATTRIBUTE));  
 if (proxyTargetClass) {  
 AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);  
 }  
 boolean exposeProxy = Boolean.valueOf(sourceElement.getAttribute(EXPOSE\_PROXY\_ATTRIBUTE));  
 if (exposeProxy) {  
 AopConfigUtils.forceAutoProxyCreatorToExposeProxy(registry);  
 }  
 }  
}

public static void forceAutoProxyCreatorToUseClassProxying(BeanDefinitionRegistry registry) {  
 if (registry.containsBeanDefinition(AUTO\_PROXY\_CREATOR\_BEAN\_NAME)) {  
 BeanDefinition definition = registry.getBeanDefinition(AUTO\_PROXY\_CREATOR\_BEAN\_NAME);  
 definition.getPropertyValues().add("proxyTargetClass", Boolean.TRUE);  
 }  
}

public static void forceAutoProxyCreatorToExposeProxy(BeanDefinitionRegistry registry) {  
 if (registry.containsBeanDefinition(AUTO\_PROXY\_CREATOR\_BEAN\_NAME)) {  
 BeanDefinition definition = registry.getBeanDefinition(AUTO\_PROXY\_CREATOR\_BEAN\_NAME);  
 definition.getPropertyValues().add("exposeProxy", Boolean.TRUE);  
 }  
}

* 创建AOP代理
* 上文是通过自定义配置完成了读AnnotationAwareAspectJAutoProxyCreator类的自动注册。
* 可见这个类实现了BeanPostProcessor接口，而实现该接口后，当Spring加载这个Bean时会在实例化前 调用其postProcessAfterInitialization方法。
* 与IOC的衔接
* beanPostProcessor在两个地方被调用，一个是
* 另一个是：
* 前一个地方是针对于实现了InstantiationAwareBeanPostProcessor接口的BeanPostProcessor，前一个地方创建代理成功后会直接返回。其他的BeanPostProcessor会在第二个地方被调用。

public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)  
 throws BeansException {  
  
 Object result = existingBean;  
 for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {  
 result = beanProcessor.postProcessAfterInitialization(result, beanName);  
 if (result == null) {  
 return result;  
 }  
 }  
 return result;  
}

* 在解析AOP标签中注册的AnnotationAwareAspectJAutoProxyCreator实现了BeanPostProcessor接口，所以在这里会被调用其postProcessAfterInstantiation方法。
* AbstractAutoProxyCreator#postProcessAfterInitialization

public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
 if (bean != null) {  
 Object cacheKey = getCacheKey(bean.getClass(), beanName);  
 if (!this.earlyProxyReferences.contains(cacheKey)) {  
 return wrapIfNecessary(bean, beanName, cacheKey);  
 }  
 }  
 return bean;  
}

* wrapIfNecessary
* 逻辑：
  + 1)获取可以应用到该bean的所有advisor
  + 2)创建代理
* protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {
* // 如果已经处理过  
  if (beanName != null && this.targetSourcedBeans.contains(beanName)) {  
   return bean;  
  }
* // 无需增强  
  if (Boolean.FALSE.equals(this.advisedBeans.get(cacheKey))) {  
   return bean;  
  }
* // 给定的bean类是否代表一个基础设施类，基础设施类不应代理，或者配置了指定bean不需要自动代理  
  if (isInfrastructureClass(bean.getClass()) || shouldSkip(bean.getClass(), beanName)) {  
   this.advisedBeans.put(cacheKey, Boolean.FALSE);  
   return bean;  
  }  
  // 获取适合应用到该bean的所有advisor  
  // Create proxy if we have advice.  
  Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, null);
* // 如果获取到了增强则需要增强创建代理  
  if (specificInterceptors != DO*NOT*PROXY) {  
   this.advisedBeans.put(cacheKey, Boolean.TRUE);
* // 创建代理  
   Object proxy = createProxy(  
   bean.getClass(), beanName, specificInterceptors, new SingletonTargetSource(bean));  
   this.proxyTypes.put(cacheKey, proxy.getClass());  
   return proxy;  
   }
* this.advisedBeans.put(cacheKey, Boolean.FALSE);  
   return bean;  
  }
* 创建代理主要包含了两个步骤：
  + 1)获取增强方法或者增强器
  + 2)根据获取的增强进行代理
  + 1) getAdvicesAndAdvisorsForBean（获取可以应用到该bean的所有advisor)
* AbstractAdvisorAutoProxyCreator.getAdvicesAndAdvisorsForBean
* protected Object[] getAdvicesAndAdvisorsForBean(Class<?> beanClass, String beanName, TargetSource targetSource) {  
  List advisors = findEligibleAdvisors(beanClass, beanName);  
  if (advisors.isEmpty()) {  
   return DO*NOT*PROXY;  
  }  
  return advisors.toArray();  
  }
* findEligibleAdvisors（合格的)
* 对于指定bean的增强方法的获取包含两个步骤：
  + 1)获取所有的增强
  + 2)寻找所有的增强中适用于bean的增强并应用
* protected List findEligibleAdvisors(Class<?> beanClass, String beanName) {  
  List candidateAdvisors = findCandidateAdvisors();  
  List eligibleAdvisors = findAdvisorsThatCanApply(candidateAdvisors, beanClass, beanName);  
  extendAdvisors(eligibleAdvisors);  
  if (!eligibleAdvisors.isEmpty()) {  
   eligibleAdvisors = sortAdvisors(eligibleAdvisors);  
  }  
  return eligibleAdvisors;  
  }
  + 1.1) findCandidateAdvisors（获取所有的增强)
* AnnotationAwareAspectJAutoProxyCreator.findCandidateAdvisors
* protected List findCandidateAdvisors() {  
  // Add all the Spring advisors found according to superclass rules.
* // 当使用注解方式配置AOP的方式的时候，并不是丢弃了对XML配置的支持
* // 在这里调用父类方法加载配置文件中的AOP声明  
  List advisors = super.findCandidateAdvisors();  
  // Build Advisors for all AspectJ aspects in the bean factory.  
  advisors.addAll(this.aspectJAdvisorsBuilder.buildAspectJAdvisors());  
  return advisors;  
  }
  + 1.1.1) AbstractAdvisorAutoProxyCreator#findCandidateAdvisors（获取配置文件中的增强)
* protected List findCandidateAdvisors() {  
  return this.advisorRetrievalHelper.findAdvisorBeans();  
  }
  + 1.1.1.1) BeanFactoryAdvisorRetrievalHelper#findAdvisorBeans

public List<Advisor> findAdvisorBeans() {  
 // Determine list of advisor bean names, if not cached already.  
 String[] advisorNames = null;  
 synchronized (this) {  
 advisorNames = this.cachedAdvisorBeanNames;  
 if (advisorNames == null) {  
 // Do not initialize FactoryBeans here: We need to leave all regular beans  
 // uninitialized to let the auto-proxy creator apply to them!

* // 从BeanFactory中获取所有对应Advisor的类
* advisorNames = BeanFactoryUtils.beanNamesForTypeIncludingAncestors(  
   this.beanFactory, Advisor.class, true, false);  
  this.cachedAdvisorBeanNames = advisorNames;  
  }  
  }  
  if (advisorNames.length == 0) {  
   return new LinkedList();  
  }
* List advisors = new LinkedList();  
  for (String name : advisorNames) {  
   if (isEligibleBean(name)) {  
   if (this.beanFactory.isCurrentlyInCreation(name)) {  
   if (logger.isDebugEnabled()) {  
   logger.debug("Skipping currently created advisor '" + name + "'");  
   }  
   }  
   else {  
   try {
* // getBean方法可以获取Advisor
* advisors.add(this.beanFactory.getBean(name, Advisor.class));  
  }  
  catch (BeanCreationException ex) {  
   Throwable rootCause = ex.getMostSpecificCause();  
   if (rootCause instanceof BeanCurrentlyInCreationException) {  
   BeanCreationException bce = (BeanCreationException) rootCause;  
   if (this.beanFactory.isCurrentlyInCreation(bce.getBeanName())) {  
   if (logger.isDebugEnabled()) {  
   logger.debug("Skipping advisor '" + name +  
   "' with dependency on currently created bean: " + ex.getMessage());  
   }  
   // Ignore: indicates a reference back to the bean we're trying to advise.  
   // We want to find advisors other than the currently created bean itself.  
   continue;  
   }  
   }  
   throw ex;  
  }  
  }  
  }  
  }  
  return advisors;  
  }
* advisorNames = BeanFactoryUtils.beanNamesForTypeIncludingAncestors(
* this.beanFactory, Advisor.class, true, false);
  + 1.1.1.1.1) BeanFactoryUtils.beanNamesForTypeIncludingAncestors

public static String[] beanNamesForTypeIncludingAncestors(  
 ListableBeanFactory lbf, Class<?> type, boolean includeNonSingletons, boolean allowEagerInit) {  
  
 Assert.notNull(lbf, "ListableBeanFactory must not be null");

* //获取Class为Advisor的所有bean的名字  
  String[] result = lbf.getBeanNamesForType(type, includeNonSingletons, allowEagerInit);  
  if (lbf instanceof HierarchicalBeanFactory) {  
   HierarchicalBeanFactory hbf = (HierarchicalBeanFactory) lbf;  
   if (hbf.getParentBeanFactory() instanceof ListableBeanFactory) {  
   String[] parentResult = beanNamesForTypeIncludingAncestors(  
   (ListableBeanFactory) hbf.getParentBeanFactory(), type, includeNonSingletons, allowEagerInit);  
   List resultList = new ArrayList();  
   resultList.addAll(Arrays.asList(result));  
   for (String beanName : parentResult) {  
   if (!resultList.contains(beanName) && !hbf.containsLocalBean(beanName)) {  
   resultList.add(beanName);  
   }  
   }  
   result = StringUtils.toStringArray(resultList);  
   }  
  }  
  return result;  
  }

- 1.1.2) BeanFactoryAspectJAdvisorsBuilder#buildAspectJAdvisors（获取标记@Aspect注解的类中的增强)

* 逻辑：
  + 1)遍历所有beanName，所有在beanFactory中注册的bean都会被提取出来
  + 2)遍历所有beanName，找出声明@Aspect注解的类，进行进一步的处理
  + 3)对标记为AspectJ注解的类进行增强的提取
  + 4)将提取结果加入缓存

public List<Advisor> buildAspectJAdvisors() {  
 List<String> aspectNames = this.aspectBeanNames;  
  
 if (aspectNames == null) {  
 synchronized (this) {  
 aspectNames = this.aspectBeanNames;  
 if (aspectNames == null) {  
 List<Advisor> advisors = new LinkedList<Advisor>();  
 aspectNames = new LinkedList<String>();

* // 获取所有的beanName  
   String[] beanNames = BeanFactoryUtils.beanNamesForTypeIncludingAncestors(  
   this.beanFactory, Object.class, true, false);
* // 遍历所有的beanName找出对应的增强方法  
   for (String beanName : beanNames) {
* // 不合法的bean则略过  
   if (!isEligibleBean(beanName)) {  
   continue;  
   }  
   // We must be careful not to instantiate beans eagerly as in this case they  
   // would be cached by the Spring container but would not have been weaved.
* // 获取对应的bean的类型  
   Class<?> beanType = this.beanFactory.getType(beanName);  
   if (beanType == null) {  
   continue;  
   }
* // 如果存在@Aspect注解  
   if (this.advisorFactory.isAspect(beanType)) {  
   aspectNames.add(beanName);  
   AspectMetadata amd = new AspectMetadata(beanType, beanName);  
   if (amd.getAjType().getPerClause().getKind() == PerClauseKind.SINGLETON) {  
   MetadataAwareAspectInstanceFactory factory =  
   new BeanFactoryAspectInstanceFactory(this.beanFactory, beanName);
* // 解析标记@Aspect中的增强方法  
   List classAdvisors = this.advisorFactory.getAdvisors(factory);  
   if (this.beanFactory.isSingleton(beanName)) {  
   this.advisorsCache.put(beanName, classAdvisors);  
   }  
   else {  
   this.aspectFactoryCache.put(beanName, factory);  
   }  
   advisors.addAll(classAdvisors);  
   }  
   else {  
   // Per target or per this.  
   if (this.beanFactory.isSingleton(beanName)) {  
   throw new IllegalArgumentException("Bean with name '" + beanName +  
   "' is a singleton, but aspect instantiation model is not singleton");  
   }  
   MetadataAwareAspectInstanceFactory factory =  
   new PrototypeAspectInstanceFactory(this.beanFactory, beanName);  
   this.aspectFactoryCache.put(beanName, factory); advisors.addAll(this.advisorFactory.getAdvisors(factory));  
   }  
   }  
   }  
   this.aspectBeanNames = aspectNames;  
   return advisors;  
   }  
   }  
  }
* if (aspectNames.isEmpty()) {  
   return Collections.emptyList();  
  }  
  // 记录在缓存中
* List advisors = new LinkedList();  
  for (String aspectName : aspectNames) {  
   List cachedAdvisors = this.advisorsCache.get(aspectName);  
   if (cachedAdvisors != null) {  
   advisors.addAll(cachedAdvisors);  
   }  
   else {  
   MetadataAwareAspectInstanceFactory factory = this.aspectFactoryCache.get(aspectName);  
   advisors.addAll(this.advisorFactory.getAdvisors(factory));  
   }  
  }  
  return advisors;  
  }
* 获取
* this.advisorFactory.getAdvisors(factory)
  + 1.1.2.1) getAdvisors（增强器的获取)
* ReflectiveAspectJAdvisorFactory.getAdvisors
* 逻辑：
  + 1)对增强器的获取
  + 2)加入同步实例化增强器以保证增强使用前的实例化
  + 3)对DeclareParents注解的获取

public List<Advisor> getAdvisors(MetadataAwareAspectInstanceFactory aspectInstanceFactory) {

* // 获取标记为@Aspect的类  
  Class<?> aspectClass = aspectInstanceFactory.getAspectMetadata().getAspectClass();  
  String aspectName = aspectInstanceFactory.getAspectMetadata().getAspectName();
* // 验证  
  validate(aspectClass);
* // We need to wrap the MetadataAwareAspectInstanceFactory with a decorator  
  // so that it will only instantiate once.  
  MetadataAwareAspectInstanceFactory lazySingletonAspectInstanceFactory =  
   new LazySingletonAspectInstanceFactoryDecorator(aspectInstanceFactory);
* List advisors = new LinkedList();  
  for (Method method : getAdvisorMethods(aspectClass)) {
* // 获取普通的advisor  
   Advisor advisor = getAdvisor(method, lazySingletonAspectInstanceFactory, advisors.size(), aspectName);  
   if (advisor != null) {  
   advisors.add(advisor);  
   }  
   }
* // If it's a per target aspect, emit the dummy instantiating aspect.  
   if (!advisors.isEmpty() && lazySingletonAspectInstanceFactory.getAspectMetadata().isLazilyInstantiated()) {
* // 如果寻找的增强器不为空，而且又配置了增强延迟初始化，那么需要在advisors开头加入同步实例化增强器  
   Advisor instantiationAdvisor = new SyntheticInstantiationAdvisor(lazySingletonAspectInstanceFactory);  
   advisors.add(0, instantiationAdvisor);  
   }  
   // 获取DeclareParents注解（引介增强IntroductionAdvisor)  
   // Find introduction fields.  
   for (Field field : aspectClass.getDeclaredFields()) {  
   Advisor advisor = getDeclareParentsAdvisor(field);  
   if (advisor != null) {  
   advisors.add(advisor);  
   }  
   }
* return advisors;  
  }
  + 1.1.2.1.1) getAdvisor（普通增强器的获取)
* 逻辑：
  + 1)切点信息的获取
  + 2)根据切点信息生成增强

public Advisor getAdvisor(Method candidateAdviceMethod, MetadataAwareAspectInstanceFactory aspectInstanceFactory,  
 int declarationOrderInAspect, String aspectName) {  
  
 validate(aspectInstanceFactory.getAspectMetadata().getAspectClass());  
 //切点信息的获取  
 AspectJExpressionPointcut expressionPointcut = getPointcut(  
 candidateAdviceMethod, aspectInstanceFactory.getAspectMetadata().getAspectClass());  
 if (expressionPointcut == null) {  
 return null;  
 }  
 // 根据切点信息生成增强器  
 return new InstantiationModelAwarePointcutAdvisorImpl(expressionPointcut, candidateAdviceMethod,  
 this, aspectInstanceFactory, declarationOrderInAspect, aspectName);  
}

* + 1.1.2.1.1.1) getPointcut（方法上切点信息的获取)

private AspectJExpressionPointcut getPointcut(Method candidateAdviceMethod, Class<?> candidateAspectClass) {

* // 获取方法上的注解  
  AspectJAnnotation<?> aspectJAnnotation =  
  AbstractAspectJAdvisorFactory.findAspectJAnnotationOnMethod(candidateAdviceMethod);  
  if (aspectJAnnotation == null) {  
   return null;  
  }  
  // 使用AspectJExpressionPointcut实例封装获取的信息
* AspectJExpressionPointcut ajexp =  
   new AspectJExpressionPointcut(candidateAspectClass, new String[0], new Class<?>[0]);
* // 提取得到的注解中的表达式，如@Pointcut(“execution(\* \*....)”)  
  ajexp.setExpression(aspectJAnnotation.getPointcutExpression());  
  ajexp.setBeanFactory(this.beanFactory);  
  return ajexp;  
  }
  + 1.1.2.1.1.1.1) findAspectJAnnotationOnMethod
* protected static AspectJAnnotation<?> findAspectJAnnotationOnMethod(Method method) {
* // 寻找特定的注解类  
  Class[] {  
   Before.class, Around.class, After.class, AfterReturning.class, AfterThrowing.class, Pointcut.class};  
  for (Class<?> c : classesToLookFor) {  
   AspectJAnnotation<?> foundAnnotation = findAnnotation(method, (Class) c);  
   if (foundAnnotation != null) {  
   return foundAnnotation;  
   }  
  }  
  return null;  
  }
* 获取指定方法上的注解并使用AspectJAnnotation封装

private static <A extends Annotation> AspectJAnnotation<A> findAnnotation(Method method, Class<A> toLookFor) {  
 A result = AnnotationUtils.findAnnotation(method, toLookFor);  
 if (result != null) {  
 return new AspectJAnnotation<A>(result);  
 }  
 else {  
 return null;  
 }  
}

* + 1.1.2.1.1.2) InstantiationModelAwarePointcutAdvisorImpl（根据切点信息生成增强器)
* 所有的增强都由Advisor的实现类InstantiationModelAwarePointcutAdvisorImpl统一封装的。
* 在实例初始化的过程中还完成了对于增强器的初始化，根据注解中的信息初始化对应的增强器是在instantiateAdvice函数中实现的。

public InstantiationModelAwarePointcutAdvisorImpl(AspectJExpressionPointcut declaredPointcut,  
 Method aspectJAdviceMethod, AspectJAdvisorFactory aspectJAdvisorFactory,  
 MetadataAwareAspectInstanceFactory aspectInstanceFactory, int declarationOrder, String aspectName) {  
  
 this.declaredPointcut = declaredPointcut;  
 this.declaringClass = aspectJAdviceMethod.getDeclaringClass();  
 this.methodName = aspectJAdviceMethod.getName();  
 this.parameterTypes = aspectJAdviceMethod.getParameterTypes();  
 this.aspectJAdviceMethod = aspectJAdviceMethod;  
 this.aspectJAdvisorFactory = aspectJAdvisorFactory;  
 this.aspectInstanceFactory = aspectInstanceFactory;  
 this.declarationOrder = declarationOrder;  
 this.aspectName = aspectName;  
  
 if (aspectInstanceFactory.getAspectMetadata().isLazilyInstantiated()) {  
 // Static part of the pointcut is a lazy type.  
 Pointcut preInstantiationPointcut = Pointcuts.union(  
 aspectInstanceFactory.getAspectMetadata().getPerClausePointcut(), this.declaredPointcut);  
  
 // Make it dynamic: must mutate from pre-instantiation to post-instantiation state.  
 // If it's not a dynamic pointcut, it may be optimized out  
 // by the Spring AOP infrastructure after the first evaluation.  
 this.pointcut = new PerTargetInstantiationModelPointcut(  
 this.declaredPointcut, preInstantiationPointcut, aspectInstanceFactory);  
 this.lazy = true;  
 }  
 else {  
 // A singleton aspect.  
 this.pointcut = this.declaredPointcut;  
 this.lazy = false;  
 this.instantiatedAdvice = instantiateAdvice(this.declaredPointcut);  
 }  
}

private Advice instantiateAdvice(AspectJExpressionPointcut pcut) {  
 return this.aspectJAdvisorFactory.getAdvice(this.aspectJAdviceMethod, pcut,  
 this.aspectInstanceFactory, this.declarationOrder, this.aspectName);  
}

public Advice getAdvice(Method candidateAdviceMethod, AspectJExpressionPointcut expressionPointcut,  
 MetadataAwareAspectInstanceFactory aspectInstanceFactory, int declarationOrder, String aspectName) {  
  
 Class<?> candidateAspectClass = aspectInstanceFactory.getAspectMetadata().getAspectClass();  
 validate(candidateAspectClass);  
  
 AspectJAnnotation<?> aspectJAnnotation =  
 AbstractAspectJAdvisorFactory.findAspectJAnnotationOnMethod(candidateAdviceMethod);  
 if (aspectJAnnotation == null) {  
 return null;  
 }  
  
 // If we get here, we know we have an AspectJ method.  
 // Check that it's an AspectJ-annotated class  
 if (!isAspect(candidateAspectClass)) {  
 throw new AopConfigException("Advice must be declared inside an aspect type: " +  
 "Offending method '" + candidateAdviceMethod + "' in class [" +  
 candidateAspectClass.getName() + "]");  
 }  
  
 if (logger.isDebugEnabled()) {  
 logger.debug("Found AspectJ method: " + candidateAdviceMethod);  
 }  
  
 AbstractAspectJAdvice springAdvice;  
 // 根据不同的注解类型封装不同的增强器  
 switch (aspectJAnnotation.getAnnotationType()) {  
 case AtBefore:  
 springAdvice = new AspectJMethodBeforeAdvice(  
 candidateAdviceMethod, expressionPointcut, aspectInstanceFactory);  
 break;  
 case AtAfter:  
 springAdvice = new AspectJAfterAdvice(  
 candidateAdviceMethod, expressionPointcut, aspectInstanceFactory);  
 break;  
 case AtAfterReturning:  
 springAdvice = new AspectJAfterReturningAdvice(  
 candidateAdviceMethod, expressionPointcut, aspectInstanceFactory);  
 AfterReturning afterReturningAnnotation = (AfterReturning) aspectJAnnotation.getAnnotation();  
 if (StringUtils.hasText(afterReturningAnnotation.returning())) {  
 springAdvice.setReturningName(afterReturningAnnotation.returning());  
 }  
 break;  
 case AtAfterThrowing:  
 springAdvice = new AspectJAfterThrowingAdvice(  
 candidateAdviceMethod, expressionPointcut, aspectInstanceFactory);  
 AfterThrowing afterThrowingAnnotation = (AfterThrowing) aspectJAnnotation.getAnnotation();  
 if (StringUtils.hasText(afterThrowingAnnotation.throwing())) {  
 springAdvice.setThrowingName(afterThrowingAnnotation.throwing());  
 }  
 break;  
 case AtAround:  
 springAdvice = new AspectJAroundAdvice(  
 candidateAdviceMethod, expressionPointcut, aspectInstanceFactory);  
 break;  
 case AtPointcut:  
 if (logger.isDebugEnabled()) {  
 logger.debug("Processing pointcut '" + candidateAdviceMethod.getName() + "'");  
 }  
 return null;  
 default:  
 throw new UnsupportedOperationException(  
 "Unsupported advice type on method: " + candidateAdviceMethod);  
 }  
  
 // Now to configure the advice...  
 springAdvice.setAspectName(aspectName);  
 springAdvice.setDeclarationOrder(declarationOrder);  
 String[] argNames = this.parameterNameDiscoverer.getParameterNames(candidateAdviceMethod);  
 if (argNames != null) {  
 springAdvice.setArgumentNamesFromStringArray(argNames);  
 }  
 springAdvice.calculateArgumentBindings();  
 return springAdvice;  
}

* 以AspectJMethodBeforeAdvice 为例，这个类是被MethodBeforeAdviceInterceptor持有的。
* MethodBeforeAdviceInterceptor会在createProxy中被织入到代理对象中。

public class MethodBeforeAdviceInterceptor implements MethodInterceptor, Serializable {  
  
 private MethodBeforeAdvice advice;  
  
  
 /\*\*  
 \* Create a new MethodBeforeAdviceInterceptor for the given advice.  
 \* @param advice the MethodBeforeAdvice to wrap  
 \*/  
 public MethodBeforeAdviceInterceptor(MethodBeforeAdvice advice) {  
 Assert.notNull(advice, "Advice must not be null");  
 this.advice = advice;  
 }  
  
 @Override  
 public Object invoke(MethodInvocation mi) throws Throwable {  
 this.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis() );  
 return mi.proceed();  
 }  
  
}

* AspectJMethodBeforeAdvice

public class AspectJMethodBeforeAdvice extends AbstractAspectJAdvice implements MethodBeforeAdvice, Serializable {  
  
 public AspectJMethodBeforeAdvice(  
 Method aspectJBeforeAdviceMethod, AspectJExpressionPointcut pointcut, AspectInstanceFactory aif) {  
  
 super(aspectJBeforeAdviceMethod, pointcut, aif);  
 }  
  
  
 @Override  
 public void before(Method method, Object[] args, Object target) throws Throwable {  
 invokeAdviceMethod(getJoinPointMatch(), null, null);  
 }  
  
 @Override  
 public boolean isBeforeAdvice() {  
 return true;  
 }  
  
 @Override  
 public boolean isAfterAdvice() {  
 return false;  
 }  
  
}

* invokeAdviceMethod
* protected Object invokeAdviceMethod(JoinPointMatch jpMatch, Object returnValue, Throwable ex) throws Throwable {  
  return invokeAdviceMethodWithGivenArgs(argBinding(getJoinPoint(), jpMatch, returnValue, ex));  
  }
* invokeAdviceMethodWithGivenArgs
* protected Object invokeAdviceMethodWithGivenArgs(Object[] args) throws Throwable {  
  Object[] actualArgs = args;  
  if (this.aspectJAdviceMethod.getParameterTypes().length == 0) {  
   actualArgs = null;  
  }  
  try {  
   ReflectionUtils.makeAccessible(this.aspectJAdviceMethod);  
   // TODO AopUtils.invokeJoinpointUsingReflection
* // 激活增强方法 method.invoke()  
   return this.aspectJAdviceMethod.invoke(this.aspectInstanceFactory.getAspectInstance(), actualArgs);  
   }  
   catch (IllegalArgumentException ex) {  
   throw new AopInvocationException("Mismatch on arguments to advice method [" +  
   this.aspectJAdviceMethod + "]; pointcut expression [" +  
   this.pointcut.getPointcutExpression() + "]", ex);  
   }  
   catch (InvocationTargetException ex) {  
   throw ex.getTargetException();  
   }  
  }
  + 1.1.2.1.2) SyntheticInstantiationAdvisor（同步实例化增强器)
* Advisor instantiationAdvisor = new
* SyntheticInstantiationAdvisor(lazySingletonAspectInstanceFactory);  
  advisors.add(0, instantiationAdvisor);

protected static class SyntheticInstantiationAdvisor extends DefaultPointcutAdvisor {  
  
 public SyntheticInstantiationAdvisor(final MetadataAwareAspectInstanceFactory aif) {  
 super(aif.getAspectMetadata().getPerClausePointcut(), new MethodBeforeAdvice() {

// 目标方法前调用，类似@Before  
 @Override  
 public void before(Method method, Object[] args, Object target) {  
 // Simply instantiate the aspect

* // 简单初始化aspect  
   aif.getAspectInstance();  
   }  
  });  
  }  
  }
  + 1.1.2.1.3) getDeclareParentsAdvisor（获取DeclareParents)
* DeclareParents主要用于引介增强的注解形式的实现，而其实现方式和普通增强很类似，只不过只用DeclareParentsAdvisor对功能进行封装。

private Advisor getDeclareParentsAdvisor(Field introductionField) {  
 DeclareParents declareParents = introductionField.getAnnotation(DeclareParents.class);  
 if (declareParents == null) {  
 // Not an introduction field  
 return null;  
 }  
  
 if (DeclareParents.class == declareParents.defaultImpl()) {  
 throw new IllegalStateException("'defaultImpl' attribute must be set on DeclareParents");  
 }  
  
 return new DeclareParentsAdvisor(  
 introductionField.getType(), declareParents.value(), declareParents.defaultImpl());  
}

* + 1.2) findAdvisorsThatCanApply（获取匹配的增强并应用)
* protected List findAdvisorsThatCanApply(  
   List candidateAdvisors, Class<?> beanClass, String beanName) {
* ProxyCreationContext.setCurrentProxiedBeanName(beanName);  
   try {
* // 过滤已经得到的advisor  
   return AopUtils.findAdvisorsThatCanApply(candidateAdvisors, beanClass);  
   }  
   finally {  
   ProxyCreationContext.setCurrentProxiedBeanName(null);  
   }  
  }
* AopUtils.findAdvisorsThatCanApply

public static List<Advisor> findAdvisorsThatCanApply(List<Advisor> candidateAdvisors, Class<?> clazz) {  
 if (candidateAdvisors.isEmpty()) {  
 return candidateAdvisors;  
 }  
 List<Advisor> eligibleAdvisors = new LinkedList<Advisor>();  
 for (Advisor candidate : candidateAdvisors) {

* // 首先处理引介增强  
   if (candidate instanceof IntroductionAdvisor && canApply(candidate, clazz)) {  
   eligibleAdvisors.add(candidate);  
   }  
   }  
   boolean hasIntroductions = !eligibleAdvisors.isEmpty();  
   for (Advisor candidate : candidateAdvisors) {  
   if (candidate instanceof IntroductionAdvisor) {  
   // already processed  
   continue;  
   }
* // 对于普通bean的处理  
   if (canApply(candidate, clazz, hasIntroductions)) {  
   eligibleAdvisors.add(candidate);  
   }  
   }  
   return eligibleAdvisors;  
  }
  + 1.2.1) canApply（真正的匹配)

public static boolean canApply(Advisor advisor, Class<?> targetClass, boolean hasIntroductions) {

* // 处理引入增强  
  if (advisor instanceof IntroductionAdvisor) {  
   return ((IntroductionAdvisor) advisor).getClassFilter().matches(targetClass);  
  }
* // 处理PointcutAdvisor，是指有切入点的Advisor  
  else if (advisor instanceof PointcutAdvisor) {  
   PointcutAdvisor pca = (PointcutAdvisor) advisor;  
   return canApply(pca.getPointcut(), targetClass, hasIntroductions);  
  }  
  else {
* // 没有切入点的始终匹配  
   // It doesn't have a pointcut so we assume it applies.  
   return true;  
   }  
  }

public static boolean canApply(Pointcut pc, Class<?> targetClass, boolean hasIntroductions) {  
 Assert.notNull(pc, "Pointcut must not be null");  
 if (!pc.getClassFilter().matches(targetClass)) {  
 return false;  
 }  
  
 MethodMatcher methodMatcher = pc.getMethodMatcher();  
 if (methodMatcher == MethodMatcher.TRUE) {  
 // No need to iterate the methods if we're matching any method anyway...  
 return true;  
 }  
  
 IntroductionAwareMethodMatcher introductionAwareMethodMatcher = null;  
 if (methodMatcher instanceof IntroductionAwareMethodMatcher) {  
 introductionAwareMethodMatcher = (IntroductionAwareMethodMatcher) methodMatcher;  
 }  
 // 获取bean目标类和所有接口，放到集合中  
 Set<Class<?>> classes = new LinkedHashSet<Class<?>>(ClassUtils.getAllInterfacesForClassAsSet(targetClass));  
 classes.add(targetClass);

* // 遍历集合，获取每个类/接口的所有方法，并对方法进行逐个匹配  
  for (Class<?> clazz : classes) {  
   Method[] methods = ReflectionUtils.getAllDeclaredMethods(clazz);  
   for (Method method : methods) {  
   if ((introductionAwareMethodMatcher != null &&  
   introductionAwareMethodMatcher.matches(method, targetClass, hasIntroductions)) ||  
   methodMatcher.matches(method, targetClass)) {  
   return true;  
   }  
   }  
  }  
  return false;  
  }
  + 2) createProxy（创建代理)
* AbstractAutoProxyCreator#createProxy
* 在获取了所有对应的bean的增强后，便可以进行代理的创建了。
* 逻辑：
  + 1)获取当前类中的属性
  + 2)添加代理接口
  + 3)封装Advisor并加入到ProxyFactory中
  + 4)设置要代理的类
  + 5)子类可以在此函数中进行对ProxyFactory的进一步封装
  + 6)进行获取代理操作
* specificInterceptors就是增强们（advisors)
* targetSource 是new SingletonTargetSource(bean)
* protected Object createProxy(  
   Class<?> beanClass, String beanName, Object[] specificInterceptors, TargetSource targetSource) {
* if (this.beanFactory instanceof ConfigurableListableBeanFactory) {  
   AutoProxyUtils.exposeTargetClass((ConfigurableListableBeanFactory) this.beanFactory, beanName, beanClass);  
   }
* ProxyFactory proxyFactory = new ProxyFactory();
* // 获取当前类中的相关属性  
  proxyFactory.copyFrom(this);
* if (!proxyFactory.isProxyTargetClass()) {  
   if (shouldProxyTargetClass(beanClass, beanName)) {  
   proxyFactory.setProxyTargetClass(true);  
   }  
   else {
* // 添加代理接口  
   evaluateProxyInterfaces(beanClass, proxyFactory);  
   }  
  }   
     
  Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);  
  for (Advisor advisor : advisors) {
* // 加入增强  
   proxyFactory.addAdvisor(advisor);  
   }  
   // 设置要代理的类  
   proxyFactory.setTargetSource(targetSource);
* // 定制代理  
  customizeProxyFactory(proxyFactory);  
  // 用来控制proxyFactory被配置后，是否还允许修改增强，缺省值为false  
  proxyFactory.setFrozen(this.freezeProxy);  
  if (advisorsPreFiltered()) {  
   proxyFactory.setPreFiltered(true);  
  }
* return proxyFactory.getProxy(getProxyClassLoader());  
  }
  + 2.1) buildAdvisors（封装拦截器为Advisor)
* protected Advisor[] buildAdvisors(String beanName, Object[] specificInterceptors) {  
  // Handle prototypes correctly...  
  Advisor[] commonInterceptors = resolveInterceptorNames();
* List allInterceptors = new ArrayList();  
  if (specificInterceptors != null) {  
   allInterceptors.addAll(Arrays.asList(specificInterceptors));  
   if (commonInterceptors.length > 0) {  
   if (this.applyCommonInterceptorsFirst) {  
   allInterceptors.addAll(0, Arrays.asList(commonInterceptors));  
   }  
   else {  
   allInterceptors.addAll(Arrays.asList(commonInterceptors));  
   }  
   }  
  }  
  if (logger.isDebugEnabled()) {  
   int nrOfCommonInterceptors = commonInterceptors.length;  
   int nrOfSpecificInterceptors = (specificInterceptors != null ? specificInterceptors.length : 0);  
   logger.debug("Creating implicit proxy for bean '" + beanName + "' with " + nrOfCommonInterceptors +  
   " common interceptors and " + nrOfSpecificInterceptors + " specific interceptors");  
  }
* Advisor[] advisors = new Advisor[allInterceptors.size()];  
  for (int i = 0; i < allInterceptors.size(); i++) {
* // 拦截器进行封装转化为advisor  
   advisors[i] = this.advisorAdapterRegistry.wrap(allInterceptors.get(i));  
   }  
   return advisors;  
  }
  + 2.1.1) DefaultAdvisorAdapterRegistery#wrap

public Advisor wrap(Object adviceObject) throws UnknownAdviceTypeException {

* // 如果要封装的对象本身就是Advisor类型的，那么直接返回  
  if (adviceObject instanceof Advisor) {  
   return (Advisor) adviceObject;  
  }
* // 因为此封装方法只对Advisor和Advice有效，如果不是则不能封装  
  if (!(adviceObject instanceof Advice)) {  
   throw new UnknownAdviceTypeException(adviceObject);  
  }  
  Advice advice = (Advice) adviceObject;  
  if (advice instanceof MethodInterceptor) {  
   // So well-known it doesn't even need an adapter.
* // 如果是MethodInterceptor类型则使用DefaultPointcutAdvisor封装  
   return new DefaultPointcutAdvisor(advice);  
   }
* // 如果存在Advisor的Adapter，那么也同样需要进行封装  
  for (AdvisorAdapter adapter : this.adapters) {  
   // Check that it is supported.  
   if (adapter.supportsAdvice(advice)) {  
   return new DefaultPointcutAdvisor(advice);  
   }  
  }  
  throw new UnknownAdviceTypeException(advice);  
  }
  + 2.2) getProxy

public Object getProxy(ClassLoader classLoader) {  
 return createAopProxy().getProxy(classLoader);  
}

- 2.2.1) createAopProxy（创建代理)

* protected final synchronized AopProxy createAopProxy() {  
  if (!this.active) {  
   activate();  
  }  
  return getAopProxyFactory().createAopProxy(this);  
  }
  + 2.2.1.1) DefaultAopProxyFactory#createAopProxy
* 逻辑：
* 如果目标对象实现了接口，默认情况下会使用JDK的动态代理
* 如果目标对象实现了接口，可以强制使用CGLIB（proxy-target-class=false)
* 如果目标对象没有实现接口，必须采用CGLIB

public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {  
 if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {  
 Class<?> targetClass = config.getTargetClass();  
 if (targetClass == null) {  
 throw new AopConfigException("TargetSource cannot determine target class: " +  
 "Either an interface or a target is required for proxy creation.");  
 }  
 if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {  
 return new JdkDynamicAopProxy(config);  
 }  
 return new ObjenesisCglibAopProxy(config);  
 }  
 else {  
 return new JdkDynamicAopProxy(config);  
 }  
}

- 2.2.2) getProxy（获取代理)

* 1>JdkDynamicAopProxy#getProxy
* 该AopProxy实现了InvocationHandler接口，重写了invoke方法。

public Object getProxy(ClassLoader classLoader) {  
 if (logger.isDebugEnabled()) {  
 logger.debug("Creating JDK dynamic proxy: target source is " + this.advised.getTargetSource());  
 }  
 Class<?>[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised, true);  
 findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);  
 return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);  
}

* invoke方法：

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 MethodInvocation invocation;  
 Object oldProxy = null;  
 boolean setProxyContext = false;  
  
 TargetSource targetSource = this.advised.targetSource;  
 Class<?> targetClass = null;  
 Object target = null;  
  
 try {

* // 处理equals  
   if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {  
   // The target does not implement the equals(Object) method itself.  
   return equals(args[0]);  
   }
* // 处理hashcode  
   else if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {  
   // The target does not implement the hashCode() method itself.  
   return hashCode();  
   }  
   else if (method.getDeclaringClass() == DecoratingProxy.class) {  
   // There is only getDecoratedClass() declared -> dispatch to proxy config.  
   return AopProxyUtils.ultimateTargetClass(this.advised);  
   }  
   else if (!this.advised.opaque && method.getDeclaringClass().isInterface() &&  
   method.getDeclaringClass().isAssignableFrom(Advised.class)) {  
   // Service invocations on ProxyConfig with the proxy config...  
   return AopUtils.invokeJoinpointUsingReflection(this.advised, method, args);  
   }
* Object retVal;  
  // 有时候目标对象内部的自我调用将无法实施切面中的增强，则需要通过此属性暴露代理  
  if (this.advised.exposeProxy) {  
   // Make invocation available if necessary.  
   oldProxy = AopContext.setCurrentProxy(proxy);  
   setProxyContext = true;  
  }  
    
  // May be null. Get as late as possible to minimize the time we "own" the target,  
  // in case it comes from a pool.  
  target = targetSource.getTarget();  
  if (target != null) {  
   targetClass = target.getClass();  
  }  
  // 获取当前方法的拦截器链  
  // Get the interception chain for this method.  
  List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);  
    
  // Check whether we have any advice. If we don't, we can fallback on direct  
  // reflective invocation of the target, and avoid creating a MethodInvocation.  
  if (chain.isEmpty()) {  
   // We can skip creating a MethodInvocation: just invoke the target directly  
   // Note that the final invoker must be an InvokerInterceptor so we know it does  
   // nothing but a reflective operation on the target, and no hot swapping or fancy proxying.  
   Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);
* // 如果没有发现任何拦截器，那么直接调用切点方法（method.invoke())  
   retVal = AopUtils.invokeJoinpointUsingReflection(target, method, argsToUse);  
   }  
   else {
* // 将拦截器封装在ReflectiveMethodInvocation  
   // We need to create a method invocation...  
   invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);  
   // Proceed to the joinpoint through the interceptor chain.
* // proceed中实现了拦截器方法的逐一调用  
   retVal = invocation.proceed();  
   }
* // Massage return value if necessary.  
   Class<?> returnType = method.getReturnType();
* // 返回结果  
   if (retVal != null && retVal == target &&  
   returnType != Object.class && returnType.isInstance(proxy) &&  
   !RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {  
   // Special case: it returned "this" and the return type of the method  
   // is type-compatible. Note that we can't help if the target sets  
   // a reference to itself in another returned object.  
   retVal = proxy;  
   }  
   else if (retVal == null && returnType != Void.TYPE && returnType.isPrimitive()) {  
   throw new AopInvocationException(  
   "Null return value from advice does not match primitive return type for: " + method);  
   }  
   return retVal;  
   }  
   finally {  
   if (target != null && !targetSource.isStatic()) {  
   // Must have come from TargetSource.  
   targetSource.releaseTarget(target);  
   }  
   if (setProxyContext) {  
   // Restore old proxy.  
   AopContext.setCurrentProxy(oldProxy);  
   }  
   }  
  }
  + 2.2.2.1) ReflectiveMethodInvocation#proceed（执行拦截器链的方法)

public Object proceed() throws Throwable {

- // 执行完所有增强后，执行切点方法(method.invoke())

// We start with an index of -1 and increment early.  
 if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {  
 return invokeJoinpoint();  
 }  
 // 获取下一个要执行的拦截器  
 Object interceptorOrInterceptionAdvice = this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);  
 if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher) {

* // 动态匹配  
   // Evaluate dynamic method matcher here: static part will already have  
   // been evaluated and found to match.  
   InterceptorAndDynamicMethodMatcher dm =  
   (InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvice;  
   if (dm.methodMatcher.matches(this.method, this.targetClass, this.arguments)) {  
   return dm.interceptor.invoke(this);  
   }  
   else {
* // 不匹配则不执行拦截器，递归调用自己，执行下一个拦截器  
   // Dynamic matching failed.  
   // Skip this interceptor and invoke the next in the chain.  
   return proceed();  
   }  
  }  
  else {
* // 若为普通拦截器则直接调用拦截器  
   // 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(this);  
   }  
  }
  + 2.2.2.1.1) invokeJoinpoint（执行切点方法)
* protected Object invokeJoinpoint() throws Throwable {  
  return AopUtils.invokeJoinpointUsingReflection(this.target, this.method, this.arguments);  
  }

public static Object invokeJoinpointUsingReflection(Object target, Method method, Object[] args)  
 throws Throwable {  
  
 // Use reflection to invoke the method.  
 try {  
 ReflectionUtils.makeAccessible(method);  
 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 method [" +  
 method + "] on target [" + target + "]", ex);  
 }  
 catch (IllegalAccessException ex) {  
 throw new AopInvocationException("Could not access method [" + method + "]", ex);  
 }  
}

- 2.2.2.1.2) invoke （执行拦截器方法)

public interface MethodInterceptor extends Interceptor {  
 Object invoke(MethodInvocation invocation) throws Throwable;  
}

* 2>CglibAopProxy#getProxy
* 虽然返回的Proxy是ObjenesisCglibAopProxy，但它继承了CglibAopProxy 的getProxy方法。
* 实现了Enhancer的创建及接口封装。

public Object getProxy(ClassLoader classLoader) {  
 if (logger.isDebugEnabled()) {  
 logger.debug("Creating CGLIB proxy: target source is " + this.advised.getTargetSource());  
 }  
  
 try {  
 Class<?> rootClass = this.advised.getTargetClass();  
 Assert.state(rootClass != null, "Target class must be available for creating a CGLIB proxy");  
  
 Class<?> proxySuperClass = rootClass;  
 if (ClassUtils.isCglibProxyClass(rootClass)) {  
 proxySuperClass = rootClass.getSuperclass();  
 Class<?>[] additionalInterfaces = rootClass.getInterfaces();  
 for (Class<?> additionalInterface : additionalInterfaces) {  
 this.advised.addInterface(additionalInterface);  
 }  
 }  
 // 验证class  
 // Validate the class, writing log messages as necessary.  
 validateClassIfNecessary(proxySuperClass, classLoader);  
 // 创建及配置Enhancer  
 // Configure CGLIB Enhancer...  
 Enhancer enhancer = createEnhancer();  
 if (classLoader != null) {  
 enhancer.setClassLoader(classLoader);  
 if (classLoader instanceof SmartClassLoader &&  
 ((SmartClassLoader) classLoader).isClassReloadable(proxySuperClass)) {  
 enhancer.setUseCache(false);  
 }  
 }  
 enhancer.setSuperclass(proxySuperClass);  
 enhancer.setInterfaces(AopProxyUtils.completeProxiedInterfaces(this.advised));  
 enhancer.setNamingPolicy(SpringNamingPolicy.INSTANCE);  
 enhancer.setStrategy(new ClassLoaderAwareUndeclaredThrowableStrategy(classLoader));  
 // 设置拦截器  
 Callback[] callbacks = getCallbacks(rootClass);  
 Class<?>[] types = new Class<?>[callbacks.length];  
 for (int x = 0; x < types.length; x++) {  
 types[x] = callbacks[x].getClass();  
 }  
 // fixedInterceptorMap only populated at this point, after getCallbacks call above  
 enhancer.setCallbackFilter(new ProxyCallbackFilter(  
 this.advised.getConfigurationOnlyCopy(), this.fixedInterceptorMap, this.fixedInterceptorOffset));  
 enhancer.setCallbackTypes(types);  
 // 生成代理类以及创建代理  
 // Generate the proxy class and create a proxy instance.  
 return createProxyClassAndInstance(enhancer, callbacks);  
 }  
 catch (CodeGenerationException ex) {  
 throw new AopConfigException("Could not generate CGLIB subclass of class [" +  
 this.advised.getTargetClass() + "]: " +  
 "Common causes of this problem include using a final class or a non-visible class",  
 ex);  
 }  
 catch (IllegalArgumentException ex) {  
 throw new AopConfigException("Could not generate CGLIB subclass of class [" +  
 this.advised.getTargetClass() + "]: " +  
 "Common causes of this problem include using a final class or a non-visible class",  
 ex);  
 }  
 catch (Throwable ex) {  
 // TargetSource.getTarget() failed  
 throw new AopConfigException("Unexpected AOP exception", ex);  
 }  
}

* + 2.2.2.1) getCallbacks

private Callback[] getCallbacks(Class<?> rootClass) throws Exception {  
 // Parameters used for optimisation choices...  
 boolean exposeProxy = this.advised.isExposeProxy();  
 boolean isFrozen = this.advised.isFrozen();  
 boolean isStatic = this.advised.getTargetSource().isStatic();  
  
 // Choose an "aop" interceptor (used for AOP calls).

* // 将拦截器封装在DynamicAdvisedInterceptor中  
  Callback aopInterceptor = new DynamicAdvisedInterceptor(this.advised);
* // Choose a "straight to target" interceptor. (used for calls that are  
  // unadvised but can return this). May be required to expose the proxy.  
  Callback targetInterceptor;  
  if (exposeProxy) {  
   targetInterceptor = isStatic ?  
   new StaticUnadvisedExposedInterceptor(this.advised.getTargetSource().getTarget()) :  
   new DynamicUnadvisedExposedInterceptor(this.advised.getTargetSource());  
  }  
  else {  
   targetInterceptor = isStatic ?  
   new StaticUnadvisedInterceptor(this.advised.getTargetSource().getTarget()) :  
   new DynamicUnadvisedInterceptor(this.advised.getTargetSource());  
  }
* // Choose a "direct to target" dispatcher (used for  
  // unadvised calls to static targets that cannot return this).  
  Callback targetDispatcher = isStatic ?  
   new StaticDispatcher(this.advised.getTargetSource().getTarget()) : new SerializableNoOp();
* Callback[] mainCallbacks = new Callback[] {
* // 将拦截器链加入到Callback中  
   aopInterceptor, // for normal advice  
   targetInterceptor, // invoke target without considering advice, if optimized  
   new SerializableNoOp(), // no override for methods mapped to this  
   targetDispatcher, this.advisedDispatcher,  
   new EqualsInterceptor(this.advised),  
   new HashCodeInterceptor(this.advised)  
  };
* Callback[] callbacks;
* // If the target is a static one and the advice chain is frozen,  
  // then we can make some optimisations by sending the AOP calls  
  // direct to the target using the fixed chain for that method.  
  if (isStatic && isFrozen) {  
   Method[] methods = rootClass.getMethods();  
   Callback[] fixedCallbacks = new Callback[methods.length];  
   this.fixedInterceptorMap = new HashMap<String, Integer>(methods.length);
* // TODO: small memory optimisation here (can skip creation for methods with no advice)  
   for (int x = 0; x < methods.length; x++) {  
   List chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(methods[x], rootClass);  
   fixedCallbacks[x] = new FixedChainStaticTargetInterceptor(  
   chain, this.advised.getTargetSource().getTarget(), this.advised.getTargetClass());  
   this.fixedInterceptorMap.put(methods[x].toString(), x);  
   }
* // Now copy both the callbacks from mainCallbacks  
   // and fixedCallbacks into the callbacks array.  
   callbacks = new Callback[mainCallbacks.length + fixedCallbacks.length];  
   System.arraycopy(mainCallbacks, 0, callbacks, 0, mainCallbacks.length);  
   System.arraycopy(fixedCallbacks, 0, callbacks, mainCallbacks.length, fixedCallbacks.length);  
   this.fixedInterceptorOffset = mainCallbacks.length;  
  }  
  else {  
   callbacks = mainCallbacks;  
  }  
  return callbacks;  
  }
* CGLIB对于方法的拦截是通过将自定义的拦截器（实现了MethodInterceptor接口)加入Callback中并在调用代理时直接激活拦截器的intercept方法实现的，那么在getCallback中实现了这样一个目的：DynamicAdvisedInterceptor继承自MethodInterceptor，加入到Callback中后，在再次调用代理时会直接调用DynamicAdvisedInterceptor中的intercept方法。
* CGLIB方式实现的代理，其核心逻辑在DynamicAdvisedInterceptor中的intercept方法中（JDK动态代理的核心逻辑是在invoke方法中)。

private static class DynamicAdvisedInterceptor implements MethodInterceptor, Serializable {  
  
 private final AdvisedSupport advised;  
  
 public DynamicAdvisedInterceptor(AdvisedSupport advised) {  
 this.advised = advised;  
 }  
  
 @Override  
 public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {  
 Object oldProxy = null;  
 boolean setProxyContext = false;  
 Class<?> targetClass = null;  
 Object target = null;  
 try {  
 if (this.advised.exposeProxy) {  
 // Make invocation available if necessary.  
 oldProxy = AopContext.setCurrentProxy(proxy);  
 setProxyContext = true;  
 }  
 // May be null. Get as late as possible to minimize the time we  
 // "own" the target, in case it comes from a pool...  
 target = getTarget();  
 if (target != null) {  
 targetClass = target.getClass();  
 }

* // 获取拦截器链  
   List chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);  
   Object retVal;  
   // Check whether we only have one InvokerInterceptor: that is,  
   // no real advice, but just reflective invocation of the target.  
   if (chain.isEmpty() && Modifier.isPublic(method.getModifiers())) {  
   // We can skip creating a MethodInvocation: just invoke the target directly.  
   // Note that the final invoker must be an InvokerInterceptor, so we know  
   // it does nothing but a reflective operation on the target, and no hot  
   // swapping or fancy proxying.  
   Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);
* // 如果拦截器为空，则直接激活原方法  
   retVal = methodProxy.invoke(target, argsToUse);  
   }  
   else {

// 进入拦截器链  
 // We need to create a method invocation...  
 retVal = new CglibMethodInvocation(proxy, target, method, args, targetClass, chain, methodProxy).proceed();  
 }  
 retVal = processReturnType(proxy, target, method, retVal);  
 return retVal;  
 }  
 finally {  
 if (target != null) {  
 releaseTarget(target);  
 }  
 if (setProxyContext) {  
 // Restore old proxy.  
 AopContext.setCurrentProxy(oldProxy);  
 }  
 }  
 }  
  
 @Override  
 public boolean equals(Object other) {  
 return (this == other ||  
 (other instanceof DynamicAdvisedInterceptor &&  
 this.advised.equals(((DynamicAdvisedInterceptor) other).advised)));  
 }  
  
 /\*\*  
 \* CGLIB uses this to drive proxy creation.  
 \*/  
 @Override  
 public int hashCode() {  
 return this.advised.hashCode();  
 }  
  
 protected Object getTarget() throws Exception {  
 return this.advised.getTargetSource().getTarget();  
 }  
  
 protected void releaseTarget(Object target) throws Exception {  
 this.advised.getTargetSource().releaseTarget(target);  
 }  
}

- 2.2.2.2) createProxyClassAndInstance

* protected Object createProxyClassAndInstance(Enhancer enhancer, Callback[] callbacks) {  
  Class<?> proxyClass = enhancer.createClass();  
  Object proxyInstance = null;
* if (objenesis.isWorthTrying()) {  
   try {  
   proxyInstance = objenesis.newInstance(proxyClass, enhancer.getUseCache());  
   }  
   catch (Throwable ex) {  
   logger.debug("Unable to instantiate proxy using Objenesis, " +  
   "falling back to regular proxy construction", ex);  
   }  
  }
* if (proxyInstance == null) {  
   // Regular instantiation via default constructor...  
   try {  
   proxyInstance = (this.constructorArgs != null ?  
   proxyClass.getConstructor(this.constructorArgTypes).newInstance(this.constructorArgs) :  
   proxyClass.newInstance());  
   }  
   catch (Throwable ex) {  
   throw new AopConfigException("Unable to instantiate proxy using Objenesis, " +  
   "and regular proxy instantiation via default constructor fails as well", ex);  
   }  
  }
* ((Factory) proxyInstance).setCallbacks(callbacks);  
  return proxyInstance;  
  }
* 实例

public class LoggingAspect {  
 @Pointcut("@annotation(org.springframework.web.bind.annotation.RequestMapping)")  
 public void declareJoinPointExpression() {  
 }  
  
 @Before("declareJoinPointExpression()")  
 public void beforeMethod(JoinPoint joinPoint) {// 连接点  
 Object[] args = joinPoint.getArgs();// 取得方法参数  
 log.info("The method [ {} ] begins with Parameters: {}", joinPoint.getSignature(), Arrays.toString(args));  
 }  
  
 @AfterReturning(value = "declareJoinPointExpression()", returning = "result")  
 public void afterMethodReturn(JoinPoint joinPoint, Object result) {  
 log.info("The method [ {} ] ends with Result: {}", joinPoint.getSignature(), result);  
 }  
  
 @AfterThrowing(value = "declareJoinPointExpression()", throwing = "e")  
 public void doAfterThrowing(JoinPoint joinPoint, Exception e) {  
 log.error("Error happened in method: [ {} ]", joinPoint.getSignature());  
 log.error("Parameters: {}", Arrays.toString(joinPoint.getArgs()));  
 log.error("Exception StackTrace: {}", e);  
 }  
}

@Controller  
public class HelloController {  
   
 @RequestMapping("/hello")  
 public ModelAndView hello(ModelAndView modelAndView){  
 modelAndView.addObject("user",new RegisterDTO("admin"));  
 modelAndView.setViewName("hello");  
 return modelAndView;  
 }  
}

* 来看一下这个HelloController是怎么被AOP代理的。
* 从AbstractAutoProxyCreator.postProcessAfterInitialization开始看起。
* 在findCandidateAdvisors中
* super.findCandidateAdvisors()返回了三个Advisor
* 在buildAdvisors中，又添加了LoggingAspect对应的三个Advisor，类型是InstantiationModelAwarePointcutAdvisorImpl。
* 它getPointcut返回的是AspectJExpressionPointcut
* 然后下面会找到所有Controller，得到Controller的所有方法，查看是否匹配。
* 最后返回的interceptors有5个，
* 之后调用createProxy，最后调用了DefaultAopProxyFactory.createAopProxy方法，返回了CglibAopProxy。
* 它再调用getProxy方法，将5个advisor封装为Callback。
* 最后生成cglib代理实例。
* Spring Transaction（声明式事务)
* 事务介绍（两类事务)
* 编程式事务：所谓编程式事务指的是通过编码方式实现事务，即类似于JDBC编程实现事务管理。管理使用TransactionTemplate或者直接使用底层的PlatformTransactionManager。对于编程式事务管理，Spring推荐使用TransactionTemplate。
* 声明式事务：管理建立在AOP之上的。其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。声明式事务最大的优点就是不需要通过编程的方式管理事务，这样就不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明(或通过基于@Transactional注解的方式)，便可以将事务规则应用到业务逻辑中。
* 显然声明式事务管理要优于编程式事务管理，这正是spring倡导的非侵入式的开发方式。
* 声明式事务管理使业务代码不受污染，一个普通的POJO对象，只要加上注解就可以获得完全的事务支持。和编程式事务相比，声明式事务唯一不足地方是，后者的最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。但是即便有这样的需求，也存在很多变通的方法，比如，可以将需要进行事务管理的代码块独立为方法等等。
* 事务传播行为
* PROPAGATION\_REQUIRED
* 如果当前存在一个事务，则加入当前事务；如果不存在任何事务，则创建一个新的事务。总之，要至少保证在一个事务中运行。PROPAGATION\_REQUIRED通常作为默认的事务传播行为。
* PROPAGATION\_SUPPORTS
* 如果当前存在一个事务，则加入当前事务；如果当前不存在事务，则直接执行。 对于一些查询方法来说，PROPAGATION*SUPPORTS通常是比较合适的传播行为选择。 如果当前方法直接执行，那么不需要事务的支持；如果当前方法被其他方法调用，而其他方法启动了一个事务的时候，使用PROPAGATION*SUPPORTS可以保证当前方法能够加入当前事务并洞察当前事务对数据资源所做的更新。 比如说，A.service()会首先更新数据库，然后调用B.service()进行查询，那么，B.service()如果是PROPAGATION*SUPPORTS的传播行为， 就可以读取A.service()之前所做的最新更新结果，而如果使用稍后所提到的PROPAGATION*NOT\_SUPPORTED，则B.service()将无法读取最新的更新结果，因为A.service()的事务在这个时候还没有提交(除非隔离级别是read uncommitted)。
* PROPAGATION\_MANDATORY
* PROPAGATION\_MANDATORY强制要求当前存在一个事务，如果不存在，则抛出异常。 如果某个方法需要事务支持，但自身又不管理事务提交或者回滚的时候，比较适合使用
* PROPAGATION\_MANDATORY。
* PROPAGATION*REQUIRES*NEW
* 不管当前是否存在事务，都会创建新的事务。如果当前存在事务的话，会将当前的事务挂起(suspend)。 如果某个业务对象所做的事情不想影响到外层事务的话，PROPAGATION*REQUIRES*NEW应该是合适的选择，比如，假设当前的业务方法需要向数据库中更新某些日志信息， 但即使这些日志信息更新失败，我们也不想因为该业务方法的事务回滚而影响到外层事务的成功提交，因为这种情况下，当前业务方法的事务成功与否对外层事务来说是无关紧要的。
* PROPAGATION*NOT*SUPPORTED
* 不支持当前事务，而是在没有事务的情况下执行。如果当前存在事务的话，当前事务原则上将被挂起(suspend)，但要依赖于对应的PlatformTransactionManager实现类是否支持事务的挂起(suspend)，更多情况请参照TransactionDefinition的javadoc文档。 PROPAGATION*NOT*SUPPORTED与PROPAGATION*SUPPORTS之间的区别，可以参照PROPAGATION*SUPPORTS部分的实例内容。
* PROPAGATION\_NEVER
* 永远不需要当前存在事务，如果存在当前事务，则抛出异常。
* PROPAGATION\_NESTED
* 如果存在当前事务，则在当前事务的一个嵌套事务中执行，否则与PROPAGATION*REQUIRED的行为类似，即创建新的事务，在新创建的事务中执行。 PROPAGATION*NESTED粗看起来好像与PROPAGATION*REQUIRES*NEW的行为类似，实际上二者是有差别的。 PROPAGATION*REQUIRES*NEW创建的新事务与外层事务属于同一个“档次”，即二者的地位是相同的，当新创建的事务运行的时候，外层事务将被暂时挂起(suspend)； 而PROPAGATION\_NESTED创建的嵌套事务则不然，它是寄生于当前外层事务的，它的地位比当前外层事务的地位要小一号，当内部嵌套事务运行的时候，外层事务也是处于active状态。是已经存在事务的一个真正的子事务. 嵌套事务开始执行时, 它将取得一个 savepoint. 如果这个嵌套事务失败, 我们将回滚到此 savepoint. 嵌套事务是外部事务的一部分, 只有外部事务结束后它才会被提交，外部事务回滚，它也会被回滚。
* 解析事务标签
* <tx:annotation-driven />
* 同AOP标签，需要一个对应的BeanDefinitionParser。
* AnnotationDrivenBeanDefinitionParser#parse

public BeanDefinition parse(Element element, ParserContext parserContext) {  
 registerTransactionalEventListenerFactory(parserContext);  
 String mode = element.getAttribute("mode");  
 if ("aspectj".equals(mode)) {  
 // mode="aspectj"  
 registerTransactionAspect(element, parserContext);  
 }  
 else {  
 // mode="proxy"  
 AopAutoProxyConfigurer.configureAutoProxyCreator(element, parserContext);  
 }  
 return null;  
}

* AopAutoProxyConfigurer#configureAutoProxyCreator
* 逻辑：
* 注册了一个creator和三个bean。这三个bean支撑起了整个的事务功能。
* 其中的两个bean（TransactionInterceptor和AnnotationTransactionAttributeSource)被注入到了一个名为BeanFactoryTransactionAttributeSourceAdvisor这个bean中。

public static void configureAutoProxyCreator(Element element, ParserContext parserContext) {

* // 注册InfrastructureAdvisorAutoProxyCreator这个类  
  AopNamespaceUtils.registerAutoProxyCreatorIfNecessary(parserContext, element);
* String txAdvisorBeanName = TransactionManagementConfigUtils.TRANSACTION*ADVISOR*BEAN\_NAME;  
  if (!parserContext.getRegistry().containsBeanDefinition(txAdvisorBeanName)) {  
   Object eleSource = parserContext.extractSource(element);  
   // Create the TransactionAttributeSource definition.
* //创建TransactionAttributeSource的bean
* RootBeanDefinition sourceDef = new RootBeanDefinition(  
   "org.springframework.transaction.annotation.AnnotationTransactionAttributeSource");  
  sourceDef.setSource(eleSource);  
  sourceDef.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);
* // 注册bean，并使用Spring中的定义规则生成beanName  
   String sourceName = parserContext.getReaderContext().registerWithGeneratedName(sourceDef);  
   // 创建TransactionInterceptor的bean  
   RootBeanDefinition interceptorDef = new RootBeanDefinition(TransactionInterceptor.class);  
   interceptorDef.setSource(eleSource);  
   interceptorDef.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);  
   registerTransactionManager(element, interceptorDef);  
   interceptorDef.getPropertyValues().add("transactionAttributeSource", new RuntimeBeanReference(sourceName));
* // 注册bean，并使用Spring中的定义规则生成beanName  
   String interceptorName = parserContext.getReaderContext().registerWithGeneratedName(interceptorDef);
* // Create the TransactionAttributeSourceAdvisor definition.  
  // 创建TransactionAttributeSourceAdvisor的bean
* RootBeanDefinition advisorDef = new RootBeanDefinition(BeanFactoryTransactionAttributeSourceAdvisor.class);  
   advisorDef.setSource(eleSource);  
   advisorDef.setRole(BeanDefinition.ROLE\_INFRASTRUCTURE);
* // 将sourceName的bean注入advisorDef的transactionAttributeSource  
   advisorDef.getPropertyValues().add("transactionAttributeSource", new RuntimeBeanReference(sourceName));
* // 将interceptorName的bean注入advisorDef的adviceBeanName  
   advisorDef.getPropertyValues().add("adviceBeanName", interceptorName);  
   if (element.hasAttribute("order")) {  
   advisorDef.getPropertyValues().add("order", element.getAttribute("order"));  
   }  
   parserContext.getRegistry().registerBeanDefinition(txAdvisorBeanName, advisorDef);
* CompositeComponentDefinition compositeDef = new CompositeComponentDefinition(element.getTagName(), eleSource);  
  compositeDef.addNestedComponent(new BeanComponentDefinition(sourceDef, sourceName));  
  compositeDef.addNestedComponent(new BeanComponentDefinition(interceptorDef, interceptorName));  
  compositeDef.addNestedComponent(new BeanComponentDefinition(advisorDef, txAdvisorBeanName));  
  parserContext.registerComponent(compositeDef);
* }  
  }
* AopNamespaceUtils#registerAutoProxyCreatorIfNecessary

public static void registerAutoProxyCreatorIfNecessary(  
 ParserContext parserContext, Element sourceElement) {  
 BeanDefinition beanDefinition = AopConfigUtils.registerAutoProxyCreatorIfNecessary(  
 parserContext.getRegistry(), parserContext.extractSource(sourceElement));  
 useClassProxyingIfNecessary(parserContext.getRegistry(), sourceElement);  
 registerComponentIfNecessary(beanDefinition, parserContext);  
}

* AopConfigUtils#registerAutoProxyCreatorIfNecessary

public static BeanDefinition registerAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry, Object source) {  
 return registerOrEscalateApcAsRequired(InfrastructureAdvisorAutoProxyCreator.class, registry, source);  
}

* 这里注册了InfrastructureAdvisorAutoProxyCreator（基础设施)这个类。
* BeanFactoryTransactionAttributeSourceAdvisor（用于对事务方法进行增强)
* 在AOP的BeanFactoryAdvisorRetrievalHelper.findAdvisorBeans中获取了所有类型为Advisor的bean，包括BeanFactoryTransactionAttributeSourceAdvisor这个类，并随着其他的Advisor一起在后续的步骤中被织入代理。

public class BeanFactoryTransactionAttributeSourceAdvisor extends AbstractBeanFactoryPointcutAdvisor {  
  
 private TransactionAttributeSource transactionAttributeSource;  
  
 private final TransactionAttributeSourcePointcut pointcut = new TransactionAttributeSourcePointcut() {  
 @Override  
 protected TransactionAttributeSource getTransactionAttributeSource() {  
 return transactionAttributeSource;  
 }  
 };  
  
  
 /\*\*  
 \* Set the transaction attribute source which is used to find transaction  
 \* attributes. This should usually be identical to the source reference  
 \* set on the transaction interceptor itself.  
 \* @see TransactionInterceptor#setTransactionAttributeSource  
 \*/  
 public void setTransactionAttributeSource(TransactionAttributeSource transactionAttributeSource) {  
 this.transactionAttributeSource = transactionAttributeSource;  
 }  
  
 /\*\*  
 \* Set the {@link ClassFilter} to use for this pointcut.  
 \* Default is {@link ClassFilter#TRUE}.  
 \*/  
 public void setClassFilter(ClassFilter classFilter) {  
 this.pointcut.setClassFilter(classFilter);  
 }  
  
 @Override  
 public Pointcut getPointcut() {  
 return this.pointcut;  
 }  
  
}

* 与IOC的衔接
* InfrastructureAdvisorAutoProxyCreator作为一个AbstractAutoProxyCreator，会在getBean时调用其postProcessAfterInstantiation方法，该方法会创建事务代理。
* InfrastructureAdvisorAutoProxyCreator#postProcessAfterInstantiation

public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws BeansException {  
 Object cacheKey = getCacheKey(beanClass, beanName);  
  
 if (beanName == null || !this.targetSourcedBeans.contains(beanName)) {  
 if (this.advisedBeans.containsKey(cacheKey)) {  
 return null;  
 }  
 if (isInfrastructureClass(beanClass) || shouldSkip(beanClass, beanName)) {  
 this.advisedBeans.put(cacheKey, Boolean.FALSE);  
 return null;  
 }  
 }  
  
 // Create proxy here if we have a custom TargetSource.  
 // Suppresses unnecessary default instantiation of the target bean:  
 // The TargetSource will handle target instances in a custom fashion.  
 if (beanName != null) {  
 TargetSource targetSource = getCustomTargetSource(beanClass, beanName);  
 if (targetSource != null) {  
 this.targetSourcedBeans.add(beanName);  
 Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(beanClass, beanName, targetSource);  
 Object proxy = createProxy(beanClass, beanName, specificInterceptors, targetSource);  
 this.proxyTypes.put(cacheKey, proxy.getClass());  
 return proxy;  
 }  
 }  
  
 return null;  
}

* getAdvicesAndAdvisorsForBean会寻找所有实现Advisor接口的类。
* 我们之前注册了BeanFactoryTransactionAttributeSourceAdvisor这个类，这个类实现了Advisor接口。BeanFactoryTransactionAttributeSourceAdvisor作为一个advisor，用于对事务方法进行增强。只要类或方法实现了@Transactional接口，该Advisor一定会被加到拦截器链中，对原方法进行事务增强。
* 返回的Advisor类型是BeanFactoryTransactionAttributeSourceAdvisor，而其beanName是TransactionInterceptor。
* 之后调用了findAdvisorsThatCanApply 方法，又调用canApply方法。
  + 1) canApply（判断bean是否需要添加事务增强)
* canApply(BeanFactoryTransactionAttributeSourceAdvisor,targetClass)
* 关键在于是否从指定的类或类中的方法找到对应的事务属性

public static boolean canApply(Advisor advisor, Class<?> targetClass, boolean hasIntroductions) {

* // 处理引入增强  
  if (advisor instanceof IntroductionAdvisor) {  
   return ((IntroductionAdvisor) advisor).getClassFilter().matches(targetClass);  
  }
* // 处理PointcutAdvisor，是指有切入点的Advisor  
  else if (advisor instanceof PointcutAdvisor) {  
   PointcutAdvisor pca = (PointcutAdvisor) advisor;  
   return canApply(pca.getPointcut(), targetClass, hasIntroductions);  
  }  
  else {
* // 没有切入点的始终匹配  
   // It doesn't have a pointcut so we assume it applies.  
   return true;  
   }  
  }
* pca.getPointcut()对于BeanFactoryTransactionAttributeSourceAdvisor而言，是TransactionAttributeSourcePointcut。

private final TransactionAttributeSourcePointcut pointcut = new TransactionAttributeSourcePointcut() {  
 @Override  
 protected TransactionAttributeSource getTransactionAttributeSource() {  
 return transactionAttributeSource;  
 }  
};

* pc.getMethodMatcher()对于TransactionAttributeSourcePointcut而言，就是this。

public final MethodMatcher getMethodMatcher() {  
 return this;  
}

public static boolean canApply(Pointcut pc, Class<?> targetClass, boolean hasIntroductions) {  
 Assert.notNull(pc, "Pointcut must not be null");  
 if (!pc.getClassFilter().matches(targetClass)) {  
 return false;  
 }  
 MethodMatcher methodMatcher = pc.getMethodMatcher();  
 if (methodMatcher == MethodMatcher.TRUE) {  
 // No need to iterate the methods if we're matching any method anyway...  
 return true;  
 }  
  
 IntroductionAwareMethodMatcher introductionAwareMethodMatcher = null;  
 if (methodMatcher instanceof IntroductionAwareMethodMatcher) {  
 introductionAwareMethodMatcher = (IntroductionAwareMethodMatcher) methodMatcher;  
 }  
 // 获取bean目标类和所有接口，放到集合中  
 Set<Class<?>> classes = new LinkedHashSet<Class<?>>(ClassUtils.getAllInterfacesForClassAsSet(targetClass));  
 classes.add(targetClass);

* // 遍历集合，获取每个类/接口的所有方法，并对方法进行逐个匹配  
  for (Class<?> clazz : classes) {  
   Method[] methods = ReflectionUtils.getAllDeclaredMethods(clazz);  
   for (Method method : methods) {  
   if ((introductionAwareMethodMatcher != null &&  
   introductionAwareMethodMatcher.matches(method, targetClass, hasIntroductions)) ||
* // 实际的匹配方法  
   methodMatcher.matches(method, targetClass)) {  
   return true;  
   }  
  }  
  }  
  return false;  
  }
* methodMatcher.matches(method, targetClass)会使用
* TransactionAttributeSourcePointcut类的matches方法。
  + 1.1) matches（匹配方法)
* 这里的tas就是TransactionAttributeSource。
* 如果该bean的该方法中存在事务属性，那么该类/方法需要继续事务增强。

public boolean matches(Method method, Class<?> targetClass) {  
 if (TransactionalProxy.class.isAssignableFrom(targetClass)) {  
 return false;  
 }  
 TransactionAttributeSource tas = getTransactionAttributeSource();  
 return (tas == null || tas.getTransactionAttribute(method, targetClass) != null);  
}

- 1.1.1) AnnotationTransactionAttributeSource#getTransactionAttribute（获取事务属性，封装了@Transactional中的配置信息)

* 先尝试从缓存加载，如果对应信息没有被缓存的话，工作又委托给了computeTransactionAttribute方法。

public TransactionAttribute getTransactionAttribute(Method method, Class<?> targetClass) {  
 if (method.getDeclaringClass() == Object.class) {  
 return null;  
 }  
  
 // First, see if we have a cached value.  
 Object cacheKey = getCacheKey(method, targetClass);  
 Object cached = this.attributeCache.get(cacheKey);  
 if (cached != null) {  
 // Value will either be canonical value indicating there is no transaction attribute,  
 // or an actual transaction attribute.  
 if (cached == NULL\_TRANSACTION\_ATTRIBUTE) {  
 return null;  
 }  
 else {  
 return (TransactionAttribute) cached;  
 }  
 }  
 else {  
 // We need to work it out.  
 TransactionAttribute txAttr = computeTransactionAttribute(method, targetClass);  
 // Put it in the cache.  
 if (txAttr == null) {  
 this.attributeCache.put(cacheKey, NULL\_TRANSACTION\_ATTRIBUTE);  
 }  
 else {  
 String methodIdentification = ClassUtils.getQualifiedMethodName(method, targetClass);  
 if (txAttr instanceof DefaultTransactionAttribute) {  
 ((DefaultTransactionAttribute) txAttr).setDescriptor(methodIdentification);  
 }  
 if (logger.isDebugEnabled()) {  
 logger.debug("Adding transactional method '" + methodIdentification + "' with attribute: " + txAttr);  
 }  
 this.attributeCache.put(cacheKey, txAttr);  
 }  
 return txAttr;  
 }  
}

- 1.1.1.1) computeTransactionAttribute（提取事务注解)

* 逻辑：如果方法中存在事务属性，则使用方法上的属性，否则使用方法所在的类上的属性。如果方法所在类的属性上还是没有搜寻到对应的事务属性，那么再搜寻接口中的方法，再没有的话，最后尝试搜寻接口的类上面的声明。

protected TransactionAttribute computeTransactionAttribute(Method method, Class<?> targetClass) {  
 // Don't allow no-public methods as required.  
 if (allowPublicMethodsOnly() && !Modifier.isPublic(method.getModifiers())) {  
 return null;  
 }  
  
 // Ignore CGLIB subclasses - introspect the actual user class.  
 Class<?> userClass = ClassUtils.getUserClass(targetClass);  
 // The method may be on an interface, but we need attributes from the target class.  
 // If the target class is null, the method will be unchanged.

* // method 代表接口中的方法，specificMethod代表实现类中的方法  
  Method specificMethod = ClassUtils.getMostSpecificMethod(method, userClass);  
  // If we are dealing with method with generic parameters, find the original method.  
  specificMethod = BridgeMethodResolver.findBridgedMethod(specificMethod);  
  // 查看方法中是否存在事务声明  
  // First try is the method in the target class.  
  TransactionAttribute txAttr = findTransactionAttribute(specificMethod);  
  if (txAttr != null) {  
   return txAttr;  
  }  
  // 查看方法所在类中是否存在事务声明  
  // Second try is the transaction attribute on the target class.  
  txAttr = findTransactionAttribute(specificMethod.getDeclaringClass());  
  if (txAttr != null && ClassUtils.isUserLevelMethod(method)) {  
   return txAttr;  
  }  
  // 如果存在接口，则到接口中寻找  
  if (specificMethod != method) {  
   // Fallback is to look at the original method.
* // 查看接口方法  
   txAttr = findTransactionAttribute(method);  
   if (txAttr != null) {  
   return txAttr;  
   }  
   // Last fallback is the class of the original method.
* // 到接口中的类中去寻找  
   txAttr = findTransactionAttribute(method.getDeclaringClass());  
   if (txAttr != null && ClassUtils.isUserLevelMethod(method)) {  
   return txAttr;  
   }  
   }
* return null;  
  }
* protected TransactionAttribute findTransactionAttribute(Method method) {  
  return determineTransactionAttribute(method);  
  }
* protected TransactionAttribute determineTransactionAttribute(AnnotatedElement ae) {  
  if (ae.getAnnotations().length > 0) {  
   for (TransactionAnnotationParser annotationParser : this.annotationParsers) {  
   TransactionAttribute attr = annotationParser.parseTransactionAnnotation(ae);  
   if (attr != null) {  
   return attr;  
   }  
   }  
  }  
  return null;  
  }
  + 1.1.1.1.1) TransactionAnnotationParser#parseTransactionAnnotation（解析注解)
* 以SpringTransactionAnnotationParser为例：

public TransactionAttribute parseTransactionAnnotation(AnnotatedElement ae) {

* //寻找@Transactional注解，有则解析该注解  
  AnnotationAttributes attributes = AnnotatedElementUtils.getMergedAnnotationAttributes(ae, Transactional.class);  
  if (attributes != null) {  
   return parseTransactionAnnotation(attributes);  
  }  
  else {  
   return null;  
  }  
  }
* 解析@Transactional中的各个属性，并封装到TransactionAttribute中返回。
* protected TransactionAttribute parseTransactionAnnotation(AnnotationAttributes attributes) {  
  RuleBasedTransactionAttribute rbta = new RuleBasedTransactionAttribute();  
  Propagation propagation = attributes.getEnum("propagation");  
  rbta.setPropagationBehavior(propagation.value());  
  Isolation isolation = attributes.getEnum("isolation");  
  rbta.setIsolationLevel(isolation.value());  
  rbta.setTimeout(attributes.getNumber("timeout").intValue());  
  rbta.setReadOnly(attributes.getBoolean("readOnly"));  
  rbta.setQualifier(attributes.getString("value"));  
  ArrayList rollBackRules = new ArrayList();  
  Class<?>[] rbf = attributes.getClassArray("rollbackFor");  
  for (Class<?> rbRule : rbf) {  
   RollbackRuleAttribute rule = new RollbackRuleAttribute(rbRule);  
   rollBackRules.add(rule);  
  }  
  String[] rbfc = attributes.getStringArray("rollbackForClassName");  
  for (String rbRule : rbfc) {  
   RollbackRuleAttribute rule = new RollbackRuleAttribute(rbRule);  
   rollBackRules.add(rule);  
  }  
  Class<?>[] nrbf = attributes.getClassArray("noRollbackFor");  
  for (Class<?> rbRule : nrbf) {  
   NoRollbackRuleAttribute rule = new NoRollbackRuleAttribute(rbRule);  
   rollBackRules.add(rule);  
  }  
  String[] nrbfc = attributes.getStringArray("noRollbackForClassName");  
  for (String rbRule : nrbfc) {  
   NoRollbackRuleAttribute rule = new NoRollbackRuleAttribute(rbRule);  
   rollBackRules.add(rule);  
  }  
  rbta.getRollbackRules().addAll(rollBackRules);  
  return rbta;  
  }
* 创建事务代理
* 重要类/接口介绍
* PlatformTransactionManager

public interface PlatformTransactionManager {  
 TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;  
 void commit(TransactionStatus status) throws TransactionException;  
 void rollback(TransactionStatus status) throws TransactionException;  
  
}

* getTransaction()：返回一个已经激活的事务或创建一个新的事务（根据给定的TransactionDefinition类型参数定义的事务属性)，返回的是TransactionStatus对象代表了当前事务的状态，其中该方法抛出TransactionException（未检查异常)表示事务由于某种原因失败。
* commit()：用于提交TransactionStatus参数代表的事务
* rollback()：用于回滚TransactionStatus参数代表的事务
* TransactionSynchronizationManager（持有一系列事务相关的ThreadLocal对象)

public abstract class TransactionSynchronizationManager {  
  
 private static final ThreadLocal<Map<Object, Object>> resources =  
 new NamedThreadLocal<Map<Object, Object>>("Transactional resources");  
  
 private static final ThreadLocal<Set<TransactionSynchronization>> synchronizations =  
 new NamedThreadLocal<Set<TransactionSynchronization>>("Transaction synchronizations");  
  
 private static final ThreadLocal<String> currentTransactionName =  
 new NamedThreadLocal<String>("Current transaction name");  
  
 private static final ThreadLocal<Boolean> currentTransactionReadOnly =  
 new NamedThreadLocal<Boolean>("Current transaction read-only status");  
  
 private static final ThreadLocal<Integer> currentTransactionIsolationLevel =  
 new NamedThreadLocal<Integer>("Current transaction isolation level");  
  
 private static final ThreadLocal<Boolean> actualTransactionActive =  
 new NamedThreadLocal<Boolean>("Actual transaction active");

* }
* getResource（获取当前线程绑定的连接)

public static Object getResource(Object key) {  
 Object actualKey = TransactionSynchronizationUtils.unwrapResourceIfNecessary(key);  
 Object value = doGetResource(actualKey);  
 if (value != null && logger.isTraceEnabled()) {  
 logger.trace("Retrieved value [" + value + "] for key [" + actualKey + "] bound to thread [" +  
 Thread.currentThread().getName() + "]");  
 }  
 return value;  
}

private static Object doGetResource(Object actualKey) {  
 Map<Object, Object> map = resources.get();  
 if (map == null) {  
 return null;  
 }  
 Object value = map.get(actualKey);  
 // Transparently remove ResourceHolder that was marked as void...  
 if (value instanceof ResourceHolder && ((ResourceHolder) value).isVoid()) {  
 map.remove(actualKey);  
 // Remove entire ThreadLocal if empty...  
 if (map.isEmpty()) {  
 resources.remove();  
 }  
 value = null;  
 }  
 return value;  
}

* bindResouce（将新连接绑定到当前线程)

public static void bindResource(Object key, Object value) throws IllegalStateException {  
 Object actualKey = TransactionSynchronizationUtils.unwrapResourceIfNecessary(key);  
 Assert.notNull(value, "Value must not be null");  
 Map<Object, Object> map = resources.get();  
 // set ThreadLocal Map if none found  
 if (map == null) {  
 map = new HashMap<Object, Object>();  
 resources.set(map);  
 }  
 Object oldValue = map.put(actualKey, value);  
 // Transparently suppress a ResourceHolder that was marked as void...  
 if (oldValue instanceof ResourceHolder && ((ResourceHolder) oldValue).isVoid()) {  
 oldValue = null;  
 }  
 if (oldValue != null) {  
 throw new IllegalStateException("Already value [" + oldValue + "] for key [" +  
 actualKey + "] bound to thread [" + Thread.currentThread().getName() + "]");  
 }  
 if (logger.isTraceEnabled()) {  
 logger.trace("Bound value [" + value + "] for key [" + actualKey + "] to thread [" +  
 Thread.currentThread().getName() + "]");  
 }  
}

* unbindResource（释放当前线程绑定的连接)

public static Object unbindResource(Object key) throws IllegalStateException {  
 Object actualKey = TransactionSynchronizationUtils.unwrapResourceIfNecessary(key);  
 Object value = doUnbindResource(actualKey);  
 if (value == null) {  
 throw new IllegalStateException(  
 "No value for key [" + actualKey + "] bound to thread [" + Thread.currentThread().getName() + "]");  
 }  
 return value;  
}

* setActualTransactionActive（设置当前线程是否存在事务)

public static void setActualTransactionActive(boolean active) {  
 actualTransactionActive.set(active ? Boolean.TRUE : null);  
}

* setCurrentTransactionIsolationLevel（设置当前线程对应事务的隔离级别)

public static void setCurrentTransactionIsolationLevel(Integer isolationLevel) {  
 currentTransactionIsolationLevel.set(isolationLevel);  
}

* isSynchronizationActive（当前线程对应的事务synchronization不为空)

public static boolean isSynchronizationActive() {  
 return (synchronizations.get() != null);  
}

* initSynchronization（初始化当前线程的synchronization)

public static void initSynchronization() throws IllegalStateException {  
 if (isSynchronizationActive()) {  
 throw new IllegalStateException("Cannot activate transaction synchronization - already active");  
 }  
 logger.trace("Initializing transaction synchronization");  
 synchronizations.set(new LinkedHashSet<TransactionSynchronization>());  
}

* clearSynchronization（清空当前线程的synchronization)

public static void clearSynchronization() throws IllegalStateException {  
 if (!isSynchronizationActive()) {  
 throw new IllegalStateException("Cannot deactivate transaction synchronization - not active");  
 }  
 logger.trace("Clearing transaction synchronization");  
 synchronizations.remove();  
}

* TransactionSynchronization（自定义触发器)
* TransactionStatus（事务状态)

public interface TransactionStatus extends SavepointManager, Flushable {  
  
 boolean isNewTransaction();  
  
 boolean hasSavepoint();  
  
 void setRollbackOnly();  
  
 boolean isRollbackOnly();  
  
 @Override  
 void flush();  
 boolean isCompleted();  
  
}

* DefaultTransactionStatus是其实现类
* isNewTransaction（是否是新事务)

public boolean isNewTransaction() {  
 return (hasTransaction() && this.newTransaction);  
}

* hasTransaction（是否有事务)

public boolean hasTransaction() {  
 return (this.transaction != null);  
}

* isReadOnly（是否是只读事务)

public boolean isReadOnly() {  
 return this.readOnly;  
}

* isGlobalRollbackOnly（是否是rollback-only)

public boolean isGlobalRollbackOnly() {  
 return ((this.transaction instanceof SmartTransactionObject) &&  
 ((SmartTransactionObject) this.transaction).isRollbackOnly());  
}

* 与AOP的衔接
* ReflectiveMethodInvocation#proceed（执行拦截器链的方法)
* 其中调用了MethodInterceptor.invoke()拦截器方法。
* 对于标记了@Transactional的方法而言，会被代理，增强事务功能。
* 这些方法的Advisor增强中包括了TransactionInterceptor
* （BeanFactoryTransactionAttributeSourceAdvisor对应的bean)。
* TransactionInterceptor支撑着整个事务功能的架构，它继承了MethodInterceptor。

public Object proceed() throws Throwable {

- // 执行完所有增强后，执行切点方法(method.invoke())

// We start with an index of -1 and increment early.  
 if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {  
 return invokeJoinpoint();  
 }  
 // 获取下一个要执行的拦截器  
 Object interceptorOrInterceptionAdvice = this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);  
 if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher) {

* // 动态匹配  
   // Evaluate dynamic method matcher here: static part will already have  
   // been evaluated and found to match.  
   InterceptorAndDynamicMethodMatcher dm =  
   (InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvice;  
   if (dm.methodMatcher.matches(this.method, this.targetClass, this.arguments)) {  
   return dm.interceptor.invoke(this);  
   }  
   else {
* // 不匹配则不执行拦截器，递归调用自己，执行下一个拦截器  
   // Dynamic matching failed.  
   // Skip this interceptor and invoke the next in the chain.  
   return proceed();  
   }  
  }  
  else {
* // 若为普通拦截器则直接调用拦截器  
   // 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(this);  
   }  
  }
* TransactionInterceptor#invoke（事务增强)

public Object invoke(final MethodInvocation invocation) throws Throwable {  
 // Work out the target class: may be {@code null}.  
 // The TransactionAttributeSource should be passed the target class  
 // as well as the method, which may be from an interface.  
 Class<?> targetClass = (invocation.getThis() != null ? AopUtils.getTargetClass(invocation.getThis()) : null);  
  
 // Adapt to TransactionAspectSupport's invokeWithinTransaction...  
 return invokeWithinTransaction(invocation.getMethod(), targetClass, new InvocationCallback() {  
 @Override  
 public Object proceedWithInvocation() throws Throwable {  
 return invocation.proceed();  
 }  
 });  
}

* TransactionAspectSupport#invokeWithinTransaction
* 逻辑：
  + 1)获取事务属性 TransactionAttribute
  + 2)加载配置中的TransactionManager
  + 3)不同的事务处理方式使用不同的逻辑，就声明式事务而言，会获取方法信息并创建事务信息TransactionInfo（此时已经创建了事务)
* 事务信息（TransactionInfo)与事务属性（TransactionAttribute)并不相同。
* 前者包含了后者，且包含了其他事务信息，比如PlatformTransactionManager以及TransactionStatus相关信息。
  + 4)try:执行原始方法
  + 5)catch:异常，回滚事务，再次抛出异常，7)及以后的不会执行
  + 6)finally:清除事务信息
  + 7)提交事务
  + 8)返回原始方法的返回值
* protected Object invokeWithinTransaction(Method method, Class<?> targetClass, final InvocationCallback invocation)  
   throws Throwable {
* // If the transaction attribute is null, the method is non-transactional.
* // 获取对应的事务属性  
  final TransactionAttribute txAttr = getTransactionAttributeSource().getTransactionAttribute(method, targetClass);
* // 获取BeanFactory中的transactionManager  
  final PlatformTransactionManager tm = determineTransactionManager(txAttr);
* // 获取方法唯一标识（类、方法)  
  final String joinpointIdentification = methodIdentification(method, targetClass, txAttr);  
  // 声明式事务处理  
  if (txAttr == null || !(tm instanceof CallbackPreferringPlatformTransactionManager)) {  
   // Standard transaction demarcation with getTransaction and commit/rollback calls.
* // 创建TransactionInfo（创建事务)  
   TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);  
   Object retVal = null;  
   try {  
   // This is an around advice: Invoke the next interceptor in the chain.  
   // This will normally result in a target object being invoked.
* // 执行原始方法  
   retVal = invocation.proceedWithInvocation();  
   }  
   catch (Throwable ex) {  
   // target invocation exception
* // 异常回滚  
   completeTransactionAfterThrowing(txInfo, ex);
* // 回滚后又将异常抛了出来  
   throw ex;  
   }  
   finally {
* // 清除消息  
   cleanupTransactionInfo(txInfo);  
   }
* // 提交事务  
   commitTransactionAfterReturning(txInfo);  
   return retVal;  
   }
* else {

// 编程式事务，略过  
 // It's a CallbackPreferringPlatformTransactionManager: pass a TransactionCallback in.  
 try {  
 Object result = ((CallbackPreferringPlatformTransactionManager) tm).execute(txAttr,  
 new TransactionCallback<Object>() {  
 @Override  
 public Object doInTransaction(TransactionStatus status) {  
 TransactionInfo txInfo = prepareTransactionInfo(tm, txAttr, joinpointIdentification, status);  
 try {  
 return invocation.proceedWithInvocation();  
 }  
 catch (Throwable ex) {  
 if (txAttr.rollbackOn(ex)) {  
 // A RuntimeException: will lead to a rollback.  
 if (ex instanceof RuntimeException) {  
 throw (RuntimeException) ex;  
 }  
 else {  
 throw new ThrowableHolderException(ex);  
 }  
 }  
 else {  
 // A normal return value: will lead to a commit.  
 return new ThrowableHolder(ex);  
 }  
 }  
 finally {  
 cleanupTransactionInfo(txInfo);  
 }  
 }  
 });  
  
 // Check result: It might indicate a Throwable to rethrow.  
 if (result instanceof ThrowableHolder) {  
 throw ((ThrowableHolder) result).getThrowable();  
 }  
 else {  
 return result;  
 }  
 }  
 catch (ThrowableHolderException ex) {  
 throw ex.getCause();  
 }  
 }  
}

- 1)createTransactionIfNecessary（创建事务)

* 逻辑：
  + 1)使用DelegatingTransactionAttribute封装传入的TransactionAttribute。
* TransactionAttribute在这里的实际类型是RuleBasedTransactionAttribute，是由获取事务属性时生成，主要用于数据承载，使用DelegatingTransactionAttribute承载可以提供更多的功能。
  + 2)获取事务
  + 3)构建事务信息
* protected TransactionInfo createTransactionIfNecessary(  
   PlatformTransactionManager tm, TransactionAttribute txAttr, final String joinpointIdentification) {
* // If no name specified, apply method identification as transaction name.

// 如果没有指定名称，则使用方法唯一标识，并使用DelegatingTransactionAttribute封装TransactionAttribute  
 if (txAttr != null && txAttr.getName() == null) {  
 txAttr = new DelegatingTransactionAttribute(txAttr) {  
 @Override  
 public String getName() {  
 return joinpointIdentification;  
 }  
 };  
 }  
  
 TransactionStatus status = null;  
 if (txAttr != null) {  
 if (tm != null) {

* // 创建事务，返回TransactionStatus  
   status = tm.getTransaction(txAttr);  
   }  
   else {  
   if (logger.isDebugEnabled()) {  
   logger.debug("Skipping transactional joinpoint [" + joinpointIdentification +  
   "] because no transaction manager has been configured");  
   }  
   }  
  }
* // 根据指定的属性与status准备一个TransactionInfo  
  return prepareTransactionInfo(tm, txAttr, joinpointIdentification, status);  
  }
  + 1.1)getTransaction（开启事务，返回TransactionStatus)
* 逻辑：
  + 1)获取事务，创建对应的事务实例
  + 2)如果当前线程存在事务，那么根据传播行为进行相应处理
  + 3)事务超时的验证
  + 4)事务传播行为的验证
  + 5)构建DefaultTransactionStatus，创建当前事务的状态
  + 6)完善transaction，包括设置ConnectionHolder、隔离级别、timeout，如果是新连接，则绑定到当前线程
  + 7)将事务信息记录在当前线程中。

public final TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException {  
 Object transaction = doGetTransaction();  
  
 // Cache debug flag to avoid repeated checks.  
 boolean debugEnabled = logger.isDebugEnabled();  
  
 if (definition == null) {  
 // Use defaults if no transaction definition given.  
 definition = new DefaultTransactionDefinition();  
 }  
 // 判断当前线程是否存在事务，依据（DataSourceTransactionManager)是当前线程记录的连接connectionHolder不为空，且connectionHolder中的transactionActive属性为true  
 if (isExistingTransaction(transaction)) {  
 // Existing transaction found -> check propagation behavior to find out how to behave.

* // 当前线程已存在事务，根据传播行为进行相应的处理，直接返回  
   return handleExistingTransaction(definition, transaction, debugEnabled);  
   }
* // 当前线程不存在事务  
  // 事务超时的验证  
  // Check definition settings for new transaction.  
  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 proceed.
* // 事务传播行为的验证
* // 当前线程不存在事务，但是传播行为却被声明为PROPAGATION*MANDATORY（支持当前事务，如果当前没有事务，就抛出异常)，则抛出异常*  
  *if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION*MANDATORY) {  
   throw new IllegalTransactionStateException(  
   "No existing transaction found for transaction marked with propagation 'mandatory'");  
  }  
  else if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION*REQUIRED ||*  
   *definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION*REQUIRES*NEW ||*  
   *definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION*NESTED) {
* // 当前线程没有事务，且传播行为是以上传播行为，那么空挂起
* // 考虑到如果有注册的Synchronization的话，需要暂时将这些与将要开启的新事务无关的Synchronization先放一边。
* // 剩下的其他情况，则返回不包含任何transaction object的TransactionStatus并返回
* // 这种情况下虽然是空的事务，但有可能需要处理在事务过程中相关的Synchronization。  
   SuspendedResourcesHolder suspendedResources = suspend(null);  
   if (debugEnabled) {  
   logger.debug("Creating new transaction with name [" + definition.getName() + "]: " + definition);  
   }
* try {  
   boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION\_NEVER);
* // 创建当前事务的状态  
   DefaultTransactionStatus status = newTransactionStatus(  
   definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);
* // 完善事务，包括设置ConnectionHolder、隔离级别、timeout
* // 另外如果是新连接，绑定到当前线程  
   doBegin(transaction, definition);
* // 将事务信息记录在当前线程中  
   prepareSynchronization(status, definition);  
   return status;  
   }  
   catch (RuntimeException ex) {  
   resume(null, suspendedResources);  
   throw ex;  
   }  
   catch (Error err) {  
   resume(null, suspendedResources);  
   throw err;  
   }  
  }  
  else {  
   // Create "empty" transaction: no actual transaction, but potentially synchronization.  
   if (definition.getIsolationLevel() != TransactionDefinition.ISOLATION*DEFAULT && logger.isWarnEnabled()) {*  
   *logger.warn("Custom isolation level specified but no actual transaction initiated; " +*  
   *"isolation level will effectively be ignored: " + definition);*  
   *}*  
   *boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION*ALWAYS);  
   return prepareTransactionStatus(definition, null, true, newSynchronization, debugEnabled, null);  
  }  
  }
  + 1.1.1) doGetTransaction（创建事务实例)
* DataSourceTransactionManager.doGetTransaction
* protected Object doGetTransaction() {  
  DataSourceTransactionObject txObject = new DataSourceTransactionObject();  
  txObject.setSavepointAllowed(isNestedTransactionAllowed());
* // 如果当前线程已经记录数据库连接，则使用原有连接  
  ConnectionHolder conHolder =  
   (ConnectionHolder) TransactionSynchronizationManager.getResource(this.dataSource);
* txObject.setConnectionHolder(conHolder, false);  
  return txObject;  
  }
  + 1.1.1.1) DataSourceTransactionObject.setConnectionHolder

public void setConnectionHolder(ConnectionHolder connectionHolder, boolean newConnectionHolder) {  
 super.setConnectionHolder(connectionHolder);  
 this.newConnectionHolder = newConnectionHolder;  
}

- 1.1.2) handleExistingTransaction （处理已存在的事务)

* 值得注意的有两点：
  + 1)REQUIRES\_NEW表示当前方法必须在它自己的事务里运行，一个新的事务将被启动。而如果有一个事务正在运行的话，则在这个方法运行期间被挂起（suspend)。
  + 2)NESTED表示如果当前正在有一个事务在运行中，则该方法应该运行在一个嵌套的事务中，被嵌套的事务可以独立于封装事务进行提交或者回滚。如果封装事务不存在，行为就像REQUIRES\_NEW。
* Spring主要有两种处理NESTED的方式：
* 首选设置保存点的方式作为异常处理的回滚
* JTA无法使用保存点，那么处理方式和REQUIRES\_NEW相同，而一旦出现异常，则由Spring的事务异常处理机制去完成后续操作。

private TransactionStatus handleExistingTransaction(  
 TransactionDefinition definition, Object transaction, boolean debugEnabled)  
 throws TransactionException {  
   
 if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_NEVER) {  
 throw new IllegalTransactionStateException(  
 "Existing transaction found for transaction marked with propagation 'never'");  
 }  
  
 if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED) {  
 if (debugEnabled) {  
 logger.debug("Suspending current transaction");  
 }  
 Object suspendedResources = suspend(transaction);  
 boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION\_ALWAYS);  
 return prepareTransactionStatus(  
 definition, null, false, newSynchronization, debugEnabled, suspendedResources);  
 }  
  
 if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_REQUIRES\_NEW) {  
 if (debugEnabled) {  
 logger.debug("Suspending current transaction, creating new transaction with name [" +  
 definition.getName() + "]");  
 }  
 SuspendedResourcesHolder suspendedResources = suspend(transaction);  
 try {  
 boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION\_NEVER);  
 DefaultTransactionStatus status = newTransactionStatus(  
 definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);  
 doBegin(transaction, definition);  
 prepareSynchronization(status, definition);  
 return status;  
 }  
 catch (RuntimeException beginEx) {  
 resumeAfterBeginException(transaction, suspendedResources, beginEx);  
 throw beginEx;  
 }  
 catch (Error beginErr) {  
 resumeAfterBeginException(transaction, suspendedResources, beginErr);  
 throw beginErr;  
 }  
 }  
  
 if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_NESTED) {  
 if (!isNestedTransactionAllowed()) {  
 throw new NestedTransactionNotSupportedException(  
 "Transaction manager does not allow nested transactions by default - " +  
 "specify 'nestedTransactionAllowed' property with value 'true'");  
 }  
 if (debugEnabled) {  
 logger.debug("Creating nested transaction with name [" + definition.getName() + "]");  
 }  
 if (useSavepointForNestedTransaction()) {  
 // Create savepoint within existing Spring-managed transaction,  
 // through the SavepointManager API implemented by TransactionStatus.  
 // Usually uses JDBC 3.0 savepoints. Never activates Spring synchronization.  
 DefaultTransactionStatus status =  
 prepareTransactionStatus(definition, transaction, false, false, debugEnabled, null);  
 status.createAndHoldSavepoint();  
 return status;  
 }  
 else {  
 // Nested transaction through nested begin and commit/rollback calls.  
 // Usually only for JTA: Spring synchronization might get activated here  
 // in case of a pre-existing JTA transaction.  
 boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION\_NEVER);  
 DefaultTransactionStatus status = newTransactionStatus(  
 definition, transaction, true, newSynchronization, debugEnabled, null);  
 doBegin(transaction, definition);  
 prepareSynchronization(status, definition);  
 return status;  
 }  
 }  
 // 处理SUPPORTS和REQUIRED  
 // Assumably PROPAGATION\_SUPPORTS or PROPAGATION\_REQUIRED.  
 if (debugEnabled) {  
 logger.debug("Participating in existing transaction");  
 }  
 if (isValidateExistingTransaction()) {  
 if (definition.getIsolationLevel() != TransactionDefinition.ISOLATION\_DEFAULT) {  
 Integer currentIsolationLevel = TransactionSynchronizationManager.getCurrentTransactionIsolationLevel();  
 if (currentIsolationLevel == null || currentIsolationLevel != definition.getIsolationLevel()) {  
 Constants isoConstants = DefaultTransactionDefinition.constants;  
 throw new IllegalTransactionStateException("Participating transaction with definition [" +  
 definition + "] specifies isolation level which is incompatible with existing transaction: " +  
 (currentIsolationLevel != null ?  
 isoConstants.toCode(currentIsolationLevel, DefaultTransactionDefinition.PREFIX\_ISOLATION) :  
 "(unknown)"));  
 }  
 }  
 if (!definition.isReadOnly()) {  
 if (TransactionSynchronizationManager.isCurrentTransactionReadOnly()) {  
 throw new IllegalTransactionStateException("Participating transaction with definition [" +  
 definition + "] is not marked as read-only but existing transaction is");  
 }  
 }  
 }  
 boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION\_NEVER);  
 return prepareTransactionStatus(definition, transaction, false, newSynchronization, debugEnabled, null);  
}

- 1.1.3) suspend（null->doNothing)

* 如果当前线程不存在事务，并且事务传播行为是Required、Required\_New、Nested，那么执行suspend(null)。
* 考虑到如果有注册的synchronization的话，需要暂时将这些与将要开启的新事务无关的synchronization先放一边。
* 正常的suspend是记录原有事务的状态，以便后续操作对事务的恢复
* （TransactionSynchronizationManager.unBindResource)
* protected final SuspendedResourcesHolder suspend(Object transaction) throws TransactionException {  
  if (TransactionSynchronizationManager.isSynchronizationActive()) {  
   List suspendedSynchronizations = doSuspendSynchronization();  
   try {  
   Object suspendedResources = null;  
   if (transaction != null) {  
   suspendedResources = doSuspend(transaction);  
   }  
   String name = TransactionSynchronizationManager.getCurrentTransactionName();  
   TransactionSynchronizationManager.setCurrentTransactionName(null);  
   boolean readOnly = TransactionSynchronizationManager.isCurrentTransactionReadOnly();  
   TransactionSynchronizationManager.setCurrentTransactionReadOnly(false);  
   Integer isolationLevel = TransactionSynchronizationManager.getCurrentTransactionIsolationLevel();  
   TransactionSynchronizationManager.setCurrentTransactionIsolationLevel(null);  
   boolean wasActive = TransactionSynchronizationManager.isActualTransactionActive();  
   TransactionSynchronizationManager.setActualTransactionActive(false);  
   return new SuspendedResourcesHolder(  
   suspendedResources, suspendedSynchronizations, name, readOnly, isolationLevel, wasActive);  
   }  
   catch (RuntimeException ex) {  
   // doSuspend failed - original transaction is still active...  
   doResumeSynchronization(suspendedSynchronizations);  
   throw ex;  
   }  
   catch (Error err) {  
   // doSuspend failed - original transaction is still active...  
   doResumeSynchronization(suspendedSynchronizations);  
   throw err;  
   }  
  }  
  else if (transaction != null) {  
   // Transaction active but no synchronization active.  
   Object suspendedResources = doSuspend(transaction);  
   return new SuspendedResourcesHolder(suspendedResources);  
  }  
  else {  
   // Neither transaction nor synchronization active.  
   return null;  
  }  
  }
  + 1.1.3.1) doSuspendSynchronization

private List<TransactionSynchronization> doSuspendSynchronization() {  
 List<TransactionSynchronization> suspendedSynchronizations =  
 TransactionSynchronizationManager.getSynchronizations();  
 for (TransactionSynchronization synchronization : suspendedSynchronizations) {  
 synchronization.suspend();  
 }

* // 清空synchronization  
  TransactionSynchronizationManager.clearSynchronization();  
  return suspendedSynchronizations;  
  }
  + 1.1.4) newTransactionStatus （创建当前事务的状态)
* boolean newSynchronization = (getTransactionSynchronization() !=
* SYNCHRONIZATION\_NEVER);
* DefaultTransactionStatus status = newTransactionStatus(definition, transaction, true, newSynchronization, debugEnabled, null);
* 新事务则传入的newTransaction为true，否则为false
* definition.isReadOnly()返回的是@Transactional中的属性，默认为false。
* protected DefaultTransactionStatus newTransactionStatus(  
   TransactionDefinition definition, Object transaction, boolean newTransaction,  
   boolean newSynchronization, boolean debug, Object suspendedResources) {
* boolean actualNewSynchronization = newSynchronization &&  
   !TransactionSynchronizationManager.isSynchronizationActive();  
   return new DefaultTransactionStatus(  
   transaction, newTransaction, actualNewSynchronization,  
   definition.isReadOnly(), debug, suspendedResources);  
  }
* 理解newTransaction
* newTransaction标识该切面方法是否新建了事务，后续切面方法执行完毕时，通过该字段判断是否 需要提交事务或者回滚事务。
  + 1.1.5) doBegin（完善事务实例)
* 逻辑：
  + 1)尝试获取连接。如果当前线程中的connectionHolder已经存在，则没有必要再次获取；对于事务同步设置为true的需要重新获取连接
  + 2)设置隔离级别以及只读标识
  + 3)更改默认的提交设置，将提交操作委托给Spring来处理
  + 4)设置标志位，标识当前连接已经被事务激活
  + 5)设置超时时间
  + 6)如果是新连接，则将connectionHolder绑定到当前线程
* protected void doBegin(Object transaction, TransactionDefinition definition) {  
  DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;  
  Connection con = null;
* try {  
   if (txObject.getConnectionHolder() == null ||  
   txObject.getConnectionHolder().isSynchronizedWithTransaction()) {  
   Connection newCon = this.dataSource.getConnection();  
   if (logger.isDebugEnabled()) {  
   logger.debug("Acquired Connection [" + newCon + "] for JDBC transaction");  
   }
* // 设置ConnectionHolder  
   txObject.setConnectionHolder(new ConnectionHolder(newCon), true);  
   }
* txObject.getConnectionHolder().setSynchronizedWithTransaction(true);  
   con = txObject.getConnectionHolder().getConnection();
* // 设置隔离级别  
   Integer previousIsolationLevel = DataSourceUtils.prepareConnectionForTransaction(con, definition);
* txObject.setPreviousIsolationLevel(previousIsolationLevel);
* // Switch to manual commit if necessary. This is very expensive in some JDBC drivers,  
  // so we don't want to do it unnecessarily (for example if we've explicitly  
  // configured the connection pool to set it already).
* // 更改自动提交设置，由Spring控制提交  
   if (con.getAutoCommit()) {  
   txObject.setMustRestoreAutoCommit(true);  
   if (logger.isDebugEnabled()) {  
   logger.debug("Switching JDBC Connection [" + con + "] to manual commit");  
   }  
   con.setAutoCommit(false);  
   }
* prepareTransactionalConnection(con, definition);
* // 设置判断当前线程是否存在事务的依据，即transactionActive  
   txObject.getConnectionHolder().setTransactionActive(true);
* int timeout = determineTimeout(definition);  
  if (timeout != TransactionDefinition.TIMEOUT\_DEFAULT) {
* //设置timeout属性  
   txObject.getConnectionHolder().setTimeoutInSeconds(timeout);  
   }
* // Bind the connection holder to the thread.
* // 如果是新的连接，则将当前获取到的连接绑定到当前线程  
   if (txObject.isNewConnectionHolder()) {  
   TransactionSynchronizationManager.bindResource(getDataSource(), txObject.getConnectionHolder());  
   }  
   }
* catch (Throwable ex) {  
   if (txObject.isNewConnectionHolder()) {  
   DataSourceUtils.releaseConnection(con, this.dataSource);  
   txObject.setConnectionHolder(null, false);  
   }  
   throw new CannotCreateTransactionException("Could not open JDBC Connection for transaction", ex);  
   }  
  }
  + 1.1.6) prepareSynchronization（记录事务信息至当前线程)
* protected void prepareSynchronization(DefaultTransactionStatus status, TransactionDefinition definition) {  
  if (status.isNewSynchronization()) {  
   TransactionSynchronizationManager.setActualTransactionActive(status.hasTransaction());  
   TransactionSynchronizationManager.setCurrentTransactionIsolationLevel(  
   definition.getIsolationLevel() != TransactionDefinition.ISOLATION\_DEFAULT ?  
   definition.getIsolationLevel() : null);  
   TransactionSynchronizationManager.setCurrentTransactionReadOnly(definition.isReadOnly());  
   TransactionSynchronizationManager.setCurrentTransactionName(definition.getName());  
   TransactionSynchronizationManager.initSynchronization();  
  }  
  }
  + 1.2) prepareTransactionInfo（构建事务信息)
* 当已经建立事务连接并完成了事务信息的提取后，需要将所有的事务信息统一记录在TransactionInfo实例中，这个实例包含了目标方法开始前的所有状态信息。一旦事务执行失败，Spring会通过TransactionInfo实例来进行回滚等后续工作。
* protected TransactionInfo prepareTransactionInfo(PlatformTransactionManager tm,  
   TransactionAttribute txAttr, String joinpointIdentification, TransactionStatus status) {
* TransactionInfo txInfo = new TransactionInfo(tm, txAttr, joinpointIdentification);  
   if (txAttr != null) {  
   // We need a transaction for this method...  
   if (logger.isTraceEnabled()) {  
   logger.trace("Getting transaction for [" + txInfo.getJoinpointIdentification() + "]");  
   }  
   // The transaction manager will flag an error if an incompatible tx already exists.
* // 记录事务状态  
   txInfo.newTransactionStatus(status);  
   }  
   else {  
   // The TransactionInfo.hasTransaction() method will return false. We created it only  
   // to preserve the integrity of the ThreadLocal stack maintained in this class.  
   if (logger.isTraceEnabled())  
   logger.trace("Don't need to create transaction for [" + joinpointIdentification +  
   "]: This method isn't transactional.");  
   }
* // We always bind the TransactionInfo to the thread, even if we didn't create  
   // a new transaction here. This guarantees that the TransactionInfo stack  
   // will be managed correctly even if no transaction was created by this aspect.  
   txInfo.bindToThread();  
   return txInfo;  
  }
  + 2)completeTransactionAfterThrowing （回滚事务)
* protected void completeTransactionAfterThrowing(TransactionInfo txInfo, Throwable ex) {
* // 当抛出异常时首先判断当前是否存在事务，这是基础依据  
  if (txInfo != null && txInfo.hasTransaction()) {  
   if (logger.isTraceEnabled()) {  
   logger.trace("Completing transaction for [" + txInfo.getJoinpointIdentification() +  
   "] after exception: " + ex);  
   }
* // 判断是否回滚 默认的依据是 抛出的异常是否是RuntimeException或者是Error的类型  
   if (txInfo.transactionAttribute.rollbackOn(ex)) {  
   try {  
   // 根据TransactionStatus信息进行回滚处理
  + txInfo.getTransactionManager().rollback(txInfo.getTransactionStatus());  
    }  
    catch (TransactionSystemException ex2) {  
     logger.error("Application exception overridden by rollback exception", ex);  
     ex2.initApplicationException(ex);  
     throw ex2;  
    }  
    catch (RuntimeException ex2) {  
     logger.error("Application exception overridden by rollback exception", ex);  
     throw ex2;  
    }  
    catch (Error err) {  
     logger.error("Application exception overridden by rollback error", ex);  
     throw err;  
    }  
    }  
    else {
  + // 如果不满足回滚条件，即使抛出异常也会提交  
    // We don't roll back on this exception.  
    // Will still roll back if TransactionStatus.isRollbackOnly() is true.  
    try {  
     txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());  
    }  
    catch (TransactionSystemException ex2) {  
     logger.error("Application exception overridden by commit exception", ex);  
     ex2.initApplicationException(ex);  
     throw ex2;  
    }  
    catch (RuntimeException ex2) {  
     logger.error("Application exception overridden by commit exception", ex);  
     throw ex2;  
    }  
    catch (Error err) {  
     logger.error("Application exception overridden by commit error", ex);  
     throw err;  
    }  
    }  
     }  
    }
  + 2.1) TransactionAttribute.rollbackOn（判断是否需要回滚)
* DefaultTransactionAttribute的实现是

public boolean rollbackOn(Throwable ex) {  
 return (ex instanceof RuntimeException || ex instanceof Error);  
}

* RuleBasedTransactionAttribute的实现是

public boolean rollbackOn(Throwable ex) {  
 if (logger.isTraceEnabled()) {  
 logger.trace("Applying rules to determine whether transaction should rollback on " + ex);  
 }  
  
 RollbackRuleAttribute winner = null;  
 int deepest = Integer.MAX\_VALUE;  
  
 if (this.rollbackRules != null) {  
 for (RollbackRuleAttribute rule : this.rollbackRules) {  
 int depth = rule.getDepth(ex);  
 if (depth >= 0 && depth < deepest) {  
 deepest = depth;  
 winner = rule;  
 }  
 }  
 }  
  
 if (logger.isTraceEnabled()) {  
 logger.trace("Winning rollback rule is: " + winner);  
 }  
  
 // User superclass behavior (rollback on unchecked) if no rule matches.  
 if (winner == null) {  
 logger.trace("No relevant rollback rule found: applying default rules");  
 return super.rollbackOn(ex);  
 }  
  
 return !(winner instanceof NoRollbackRuleAttribute);  
}

- 2.2) rollback（回滚)

public final void rollback(TransactionStatus status) throws TransactionException {  
 if (status.isCompleted()) {  
 throw new IllegalTransactionStateException(  
 "Transaction is already completed - do not call commit or rollback more than once per transaction");  
 }  
  
 DefaultTransactionStatus defStatus = (DefaultTransactionStatus) status;  
 processRollback(defStatus);  
}

* processRollback
  + 1)首先是自定义触发器的调用，包括在回滚前、完成回滚后的调用，当然完成回滚包括正常回滚与回滚过程中出现异常，自定义的触发器会根据这些信息做进一步处理，而对于触发器的注册，常见是在回调过程过程中提供TransactionSynchronizationManager类中的静态方法直接注册。

public static void registerSynchronization(TransactionSynchronization synchronization)

- 2)除了触发监听函数外，就是真正的回滚逻辑处理了。有保存点则回滚到保存点，是新事务则回滚整个事务；存在事务又不是新事务，则做回滚标记。  
- 3)回滚后进行信息清除

private void processRollback(DefaultTransactionStatus status) {  
 try {  
 try {

* // 激活所有TransactionSynchronization中对应的beforeCompletion方法  
   triggerBeforeCompletion(status);  
   if (status.hasSavepoint()) {  
   if (status.isDebug()) {  
   logger.debug("Rolling back transaction to savepoint");  
   }
* // 如果有保存点，也就是当前事务为单独的线程，则会退到保存点  
   status.rollbackToHeldSavepoint();  
   }  
   else if (status.isNewTransaction()) {  
   if (status.isDebug()) {  
   logger.debug("Initiating transaction rollback");  
   }
* // 如果当前事务为独立的新事务，则直接回滚  
   doRollback(status);  
   }  
   else if (status.hasTransaction()) {  
   if (status.isLocalRollbackOnly() || isGlobalRollbackOnParticipationFailure()) {  
   if (status.isDebug()) {  
   logger.debug("Participating transaction failed - marking existing transaction as rollback-only");  
   }
* // 如果当前事务不是独立的事务，那么只能标记状态，等到事务链执行完毕后统一回滚  
   doSetRollbackOnly(status);  
   }  
   else {  
   if (status.isDebug()) {  
   logger.debug("Participating transaction failed - letting transaction originator decide on rollback");  
   }  
   }  
  }  
  else {  
   logger.debug("Should roll back transaction but cannot - no transaction available");  
  }  
  }  
  catch (RuntimeException ex) {  
   triggerAfterCompletion(status, TransactionSynchronization.STATUS*UNKNOWN);*  
   *throw ex;*  
  *}*  
  *catch (Error err) {*  
   *triggerAfterCompletion(status, TransactionSynchronization.STATUS*UNKNOWN);  
   throw err;  
  }
* // 激活所有TransactionSynchronization中对应的方法  
   triggerAfterCompletion(status, TransactionSynchronization.STATUS*ROLLED*BACK);  
  }  
  finally {
* // 清空记录的资源并将挂起的资源恢复  
   cleanupAfterCompletion(status);  
  }  
  }
  + 2.2.1) triggerBeforeCompletion（调用触发器)
* protected final void triggerBeforeCompletion(DefaultTransactionStatus status) {  
  if (status.isNewSynchronization()) {  
   if (status.isDebug()) {  
   logger.trace("Triggering beforeCompletion synchronization");  
   }  
   TransactionSynchronizationUtils.triggerBeforeCompletion();  
  }  
  }
* TransactionSynchronizationUtils.triggerBeforeCompletion()

public static void triggerBeforeCompletion() {  
 for (TransactionSynchronization synchronization : TransactionSynchronizationManager.getSynchronizations()) {  
 try {  
 synchronization.beforeCompletion();  
 }  
 catch (Throwable tsex) {  
 logger.error("TransactionSynchronization.beforeCompletion threw exception", tsex);  
 }  
 }  
}

- 2.2.2) rollbackToHeldSavepoint（回滚至保存点)

public void rollbackToHeldSavepoint() throws TransactionException {  
 if (!hasSavepoint()) {  
 throw new TransactionUsageException(  
 "Cannot roll back to savepoint - no savepoint associated with current transaction");  
 }  
 getSavepointManager().rollbackToSavepoint(getSavepoint());  
 getSavepointManager().releaseSavepoint(getSavepoint());  
 setSavepoint(null);  
}

* JdbcTransactionObjectSupport.rollbackToSavepoint

public void rollbackToSavepoint(Object savepoint) throws TransactionException {  
 ConnectionHolder conHolder = getConnectionHolderForSavepoint();  
 try {  
 conHolder.getConnection().rollback((Savepoint) savepoint);  
 }  
 catch (Throwable ex) {  
 throw new TransactionSystemException("Could not roll back to JDBC savepoint", ex);  
 }  
}

- 2.2.3) doRollback（回滚整个事务)

* protected void doRollback(DefaultTransactionStatus status) {  
  DataSourceTransactionObject txObject = (DataSourceTransactionObject) status.getTransaction();  
  Connection con = txObject.getConnectionHolder().getConnection();  
  if (status.isDebug()) {  
   logger.debug("Rolling back JDBC transaction on Connection [" + con + "]");  
  }  
  try {  
   con.rollback();  
  }  
  catch (SQLException ex) {  
   throw new TransactionSystemException("Could not roll back JDBC transaction", ex);  
  }  
  }
  + 2.2.4) doSetRollbackOnly（设置回滚标记)
* protected void doSetRollbackOnly(DefaultTransactionStatus status) {  
  DataSourceTransactionObject txObject = (DataSourceTransactionObject) status.getTransaction();  
  if (status.isDebug()) {  
   logger.debug("Setting JDBC transaction [" + txObject.getConnectionHolder().getConnection() +  
   "] rollback-only");  
  }  
  txObject.setRollbackOnly();  
  }

public void setRollbackOnly() {  
 this.rollbackOnly = true;  
}

- 2.2.5) triggerAfterCompletion（调用触发器)

private void triggerAfterCompletion(DefaultTransactionStatus status, int completionStatus) {  
 if (status.isNewSynchronization()) {  
 List<TransactionSynchronization> synchronizations = TransactionSynchronizationManager.getSynchronizations();  
 TransactionSynchronizationManager.clearSynchronization();  
 if (!status.hasTransaction() || status.isNewTransaction()) {  
 if (status.isDebug()) {  
 logger.trace("Triggering afterCompletion synchronization");  
 }  
 // No transaction or new transaction for the current scope ->  
 // invoke the afterCompletion callbacks immediately  
 invokeAfterCompletion(synchronizations, completionStatus);  
 }  
 else if (!synchronizations.isEmpty()) {  
 // Existing transaction that we participate in, controlled outside  
 // of the scope of this Spring transaction manager -> try to register  
 // an afterCompletion callback with the existing (JTA) transaction.  
 registerAfterCompletionWithExistingTransaction(status.getTransaction(), synchronizations);  
 }  
 }  
}

- 2.2.6) cleanupAfterCompletion（回滚后清除信息)

* 逻辑：
  + 1)设置状态是对事务信息做完成标识以避免重复调用
  + 2)如果当前事务是新的同步状态，需要将绑定到当前线程的事务信息清除
  + 3)如果是新事务需要做些清除资源的工作

private void cleanupAfterCompletion(DefaultTransactionStatus status) {

* // 设置完成状态  
  status.setCompleted();  
  if (status.isNewSynchronization()) {  
   TransactionSynchronizationManager.clear();  
  }  
  if (status.isNewTransaction()) {
* // 清除资源  
   doCleanupAfterCompletion(status.getTransaction());  
   }  
   if (status.getSuspendedResources() != null) {  
   if (status.isDebug()) {  
   logger.debug("Resuming suspended transaction after completion of inner transaction");  
   }
* // 结束之前事务的挂起状态  
   resume(status.getTransaction(), (SuspendedResourcesHolder) status.getSuspendedResources());  
   }  
  }
  + 2.2.6.1) doCleanupAfterCompletion（新事务则释放资源)
* protected void doCleanupAfterCompletion(Object transaction) {  
  DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;
* // 将连接从当前线程中解除绑定  
  // Remove the connection holder from the thread, if exposed.  
  if (txObject.isNewConnectionHolder()) {  
   TransactionSynchronizationManager.unbindResource(this.dataSource);  
  }
* // Reset connection.  
  Connection con = txObject.getConnectionHolder().getConnection();  
  try {  
   if (txObject.isMustRestoreAutoCommit()) {   
   con.setAutoCommit(true);  
   }
* // 重置数据库连接  
   DataSourceUtils.resetConnectionAfterTransaction(con, txObject.getPreviousIsolationLevel());  
   }  
   catch (Throwable ex) {  
   logger.debug("Could not reset JDBC Connection after transaction", ex);  
   }
* if (txObject.isNewConnectionHolder()) {  
   if (logger.isDebugEnabled()) {  
   logger.debug("Releasing JDBC Connection [" + con + "] after transaction");  
   }
* // 如果当前事务是独立的新创建的事务，则在事务完成时释放数据库连接  
   DataSourceUtils.releaseConnection(con, this.dataSource);  
   }
* txObject.getConnectionHolder().clear();  
  }
  + 2.2.6.2) resume（将挂起事务恢复)
* 如果在事务执行前有事务挂起，那么当前事务执行结束后需要将挂起事务恢复
* protected final void resume(Object transaction, SuspendedResourcesHolder resourcesHolder)  
   throws TransactionException {
* if (resourcesHolder != null) {  
   Object suspendedResources = resourcesHolder.suspendedResources;  
   if (suspendedResources != null) {  
   doResume(transaction, suspendedResources);  
   }  
   List suspendedSynchronizations = resourcesHolder.suspendedSynchronizations;  
   if (suspendedSynchronizations != null) {  
   TransactionSynchronizationManager.setActualTransactionActive(resourcesHolder.wasActive);  
   TransactionSynchronizationManager.setCurrentTransactionIsolationLevel(resourcesHolder.isolationLevel);  
   TransactionSynchronizationManager.setCurrentTransactionReadOnly(resourcesHolder.readOnly);  
   TransactionSynchronizationManager.setCurrentTransactionName(resourcesHolder.name);  
   doResumeSynchronization(suspendedSynchronizations);  
   }  
   }  
  }
  + 3)commitTransactionAfterReturning（提交事务)
* protected void commitTransactionAfterReturning(TransactionInfo txInfo) {  
  if (txInfo != null && txInfo.hasTransaction()) {  
   if (logger.isTraceEnabled()) {  
   logger.trace("Completing transaction for [" + txInfo.getJoinpointIdentification() + "]");  
   }  
   txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());  
  }  
  }
* commit
* 某个事务是另一个事务的嵌入事务，但是这些事务又不在Spring的管理范围之内，或者无法设置保存点，那么Spring会通过设置回滚标识的方式来禁止提交。首先当某个嵌入事务发生回滚的时候会设置回滚标识，而等到外部事务提交时，一旦判断出当前事务流被设置了回滚标识，则由外部事务来统一进行整体事务的回滚。

public final void commit(TransactionStatus status) throws TransactionException {  
 if (status.isCompleted()) {  
 throw new IllegalTransactionStateException(  
 "Transaction is already completed - do not call commit or rollback more than once per transaction");  
 }  
  
 DefaultTransactionStatus defStatus = (DefaultTransactionStatus) status;

* // 如果在事务链中已经被标记回滚，那么不会尝试提交事务，直接回滚  
  if (defStatus.isLocalRollbackOnly()) {  
   if (defStatus.isDebug()) {  
   logger.debug("Transactional code has requested rollback");  
   }  
   processRollback(defStatus);  
   return;  
  }  
  if (!shouldCommitOnGlobalRollbackOnly() && defStatus.isGlobalRollbackOnly()) {  
   if (defStatus.isDebug()) {  
   logger.debug("Global transaction is marked as rollback-only but transactional code requested commit");  
   }  
   processRollback(defStatus);  
   // Throw UnexpectedRollbackException only at outermost transaction boundary  
   // or if explicitly asked to.  
   if (status.isNewTransaction() || isFailEarlyOnGlobalRollbackOnly()) {  
   throw new UnexpectedRollbackException(  
   "Transaction rolled back because it has been marked as rollback-only");  
   }  
   return;  
  }  
  // 处理事务提交  
  processCommit(defStatus);  
  }
* processCommit
* 在提交过程中也不是直接提交的，而是考虑了诸多方面。
* 符合提交的条件如下：
* 当事务状态中有保存点信息的话便不会提交事务；
* 当事务不是新事务的时候也不会提交事务
* 原因是：
* 对于内嵌事务，在Spring中会将其在开始之前设置保存点，一旦内嵌事务出现异常便根据保存点信息进行回滚，但是如果没有出现异常，内嵌事务并不会单独提交，而是根据事务流由最外层事务负责提交，所以如果当前存在保存点信息便不是最外层事务，不做提交操作。

private void processCommit(DefaultTransactionStatus status) throws TransactionException {  
 try {  
 boolean beforeCompletionInvoked = false;  
 try {

* // 预留  
   prepareForCommit(status);
* // 添加的TransactionSynchronization中对应方法的调用  
   triggerBeforeCommit(status);
* // 添加的TransactionSynchronization中对应方法的调用  
   triggerBeforeCompletion(status);  
   beforeCompletionInvoked = true;  
   boolean globalRollbackOnly = false;  
   if (status.isNewTransaction() || isFailEarlyOnGlobalRollbackOnly()) {  
   globalRollbackOnly = status.isGlobalRollbackOnly();  
   }  
   if (status.hasSavepoint()) {  
   if (status.isDebug()) {  
   logger.debug("Releasing transaction savepoint");  
   }
* // 如果存在保存点，则清除保存点信息  
   status.releaseHeldSavepoint();  
   }  
   else if (status.isNewTransaction()) {  
   if (status.isDebug()) {  
   logger.debug("Initiating transaction commit");  
   }
* // 如果是新事务，则提交  
   doCommit(status);  
   }  
   // Throw UnexpectedRollbackException if we have a global rollback-only  
   // marker but still didn't get a corresponding exception from commit.  
   if (globalRollbackOnly) {  
   throw new UnexpectedRollbackException(  
   "Transaction silently rolled back because it has been marked as rollback-only");  
   }  
  }  
  catch (UnexpectedRollbackException ex) {  
   // can only be caused by doCommit  
   triggerAfterCompletion(status, TransactionSynchronization.STATUS*ROLLED*BACK);  
   throw ex;  
  }  
  catch (TransactionException ex) {  
   // can only be caused by doCommit  
   if (isRollbackOnCommitFailure()) {  
   doRollbackOnCommitException(status, ex);  
   }  
   else {  
   triggerAfterCompletion(status, TransactionSynchronization.STATUS\_UNKNOWN);  
   }  
   throw ex;  
  }  
  catch (RuntimeException ex) {  
   if (!beforeCompletionInvoked) {
* // 添加的TransactionSynchronization中对应方法的调用  
   triggerBeforeCompletion(status);  
   }
* // 提交过程中出现异常则回滚  
   doRollbackOnCommitException(status, ex);  
   throw ex;  
   }  
   catch (Error err) {  
   if (!beforeCompletionInvoked) {  
   triggerBeforeCompletion(status);  
   }  
   doRollbackOnCommitException(status, err);  
   throw err;  
   }
* // Trigger afterCommit callbacks, with an exception thrown there  
   // propagated to callers but the transaction still considered as committed.  
   try {
* // 添加的TransactionSynchronization中对应方法的调用  
   triggerAfterCommit(status);  
   }  
   finally {  
   triggerAfterCompletion(status, TransactionSynchronization.STATUS\_COMMITTED);  
   }
* }  
  finally {  
   cleanupAfterCompletion(status);  
  }  
  }
* 实例

public interface LoginService {  
 void login(RegisterDTO dto);  
}

@Service  
public class LoginServiceImpl implements LoginService{  
 @Override  
 @Transactional(rollbackFor = RuntimeException.class,propagation = Propagation.NESTED)  
 public void login(RegisterDTO dto) {  
 System.out.println("login...");  
 throw new RuntimeException("exception in loginservice");  
 }  
}

public interface UserService {  
 void addUser(RegisterDTO dto);  
}

@Service  
public class UserServiceImpl implements UserService{  
   
 @Transactional(rollbackFor = RuntimeException.class,propagation = Propagation.NESTED)  
 @Override  
 public void addUser(RegisterDTO dto) {  
 System.out.println("addUser...");  
 }  
}

public interface RegisterService {  
 void register(RegisterDTO dto);  
}

@Service  
public class RegisterServiceImpl implements RegisterService{  
 @Autowired  
 private LoginService loginService;  
 @Autowired  
 private UserService userService;  
   
 @Transactional(rollbackFor = RuntimeException.class,propagation = Propagation.NESTED)  
 @Override  
 public void register(RegisterDTO dto) {  
 System.out.println("registering...");  
 loginService.login(dto);  
 System.out.println("invoke other methods...");  
 userService.addUser(dto);  
 }  
}

@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration("classpath\*:spring/spring-\*.xml")  
@WebAppConfiguration  
public class TransactionTest {  
 @Autowired  
 private RegisterService registerService;  
   
 @Test  
 public void test(){  
 registerService.register(new RegisterDTO());  
 }  
}

* REQUIRED（内层事务newTransaction为false，内层事务回滚时仅设置回滚标记，外层事务进行外层回滚)
* 如果几个不同的service都是共享同一个事务（也就是service对象嵌套传播机制为Propagation.REQUIRED)，那么它们会一起提交，一起回滚。
* 同一个事务，如果一个service已经提交了，在另外service中rollback自然对第一个service提交的代码回滚不了的。所以spring处理嵌套事务，就是在TransactionInterceptor方法中，根据一系列开关（事务传播行为)，来处理事务是同一个还是重新获取，如果是同一个事务,不同service的commit与rollback的时机。
* 这里有一个外层切面register，使用了事务，里面调用了两个service，它们也是要求使用事务的。
* 因为传播行为是REQUIRED，所以共用同一个事务。
* 当调用register时，在getTransaction时会将TransactionStatus中的newTransaction设置为true，并且将连接绑定到当前线程，设置当前线程存在事务。
* 当后续调用login和addUser时，它们的TransactionStatus中的newTransaction设置为false。
* 如果register方法中抛出运行时异常，那么直接rollback整个事务，因为它是一个新事务。
* 如果login方法中抛出运行时异常，只能将rollbackOnly设置为true。因为login中没有catch该异常（回滚后又抛出该异常)，所以异常被register捕获（跳过了addUser)，所以register又要执行rollback方法，整个事务进行回滚。
* REQUIRES\_NEW（内外层事务平级，内层事务newTransaction为true，suspend外层事务，抛出异常后内层事务进行内层回滚，resume外层事务，外层事务捕获到内层抛出的异常后进行外层回滚)
* register创建一个事务，newTransaction为true。
* 调用login方法时检测到已存在事务，则将已存在事务suspend，并且创建的新事务，newTransaction为true。
* 当login方法抛出异常时，因为newTransaction为true，则回滚该事务。异常被抛出到外层register，register捕获该异常，并回滚。
* NESTED（内外层事务嵌套，内层事务newTransaction为false，并创建还原点，抛出异常后rollback至还原点，外层事务捕获到内层抛出的异常后进行外层回滚)
* 如果PlatformTransactionManager支持还原点，便如上执行；如果不支持，那么行为与REQUIRES\_NEW相同。
* 外层事务register的newTransaction为true，进入内层事务login。
* 内层事务login的newTransaction为false，并在获取Transaction时创建了一个还原点。（JTA不支持还原点，此时行为与REQUIRES\_NEW相同)。抛出异常后在rollback时直接rollback至之前创建的还原点，并删除了该还原点。
* 外层事务捕获到该异常，进入rollback，因为是新事务，执行外层事务的回滚。
* Spring MVC
* 配置文件示例
* web.xml
* contextConfigLocation  
  classpath*:spring/spring-*.xml

org.springframework.web.context.ContextLoaderListener

springDispatcherServlet  
 org.springframework.web.servlet.DispatcherServlet  
   
 contextConfigLocation  
 classpath:spring/spring-web.xml

springDispatcherServlet  
 /

* 在这里配置了contextConfigLocation，指定了配置文件的位置；
* 并配置了DispatcherServlet，Spring使用该类拦截Web请求并进行转发；
* 还配置了ContextLoaderListener，用于初始化Spring。
* 在Spring配置文件中，需要配置viewResolver。
* 运行流程
* DispatcherServlet： SpringMVC总的拦截器
* HandlerMapping：请求和处理器之间的映射，用于获取HandlerExecutionChain
* HandlerExecutionChain：持有一组Interceptor和实际请求处理器HandlerAdapter，负责执行Interceptor的各个方法和处理方法。
* HandlerAdapter：实际的请求处理器，处理后返回ModelAndView
* HandlerExceptionResolver：异常处理器，当拦截器的postHandle方法调用后检查异常。
* ViewResolver：视图解析器,解析视图名，得到View，由逻辑视图变为物理视图。
* View ：有render方法，渲染视图
* 渲染完毕后调用转发
* 初始化ApplicationContext
* ContextLoaderListener（入口)
* ContextLoaderListener的作用就是启动Web容器时，自动装配ApplicationContext的配置信息。
* 因为它实现了ServletContextListener这个接口，在web.xml配置这个监听器，启动容器时，就会默认执行它实现的方法，使用ServletContextListener接口，开发者能够在为客户端请求提供服务之前向ServletContext中添加任意的对象。
* 在ServletContextListener中的核心逻辑是初始化WebApplicationContext实例并存放在ServletContext中。

public void contextInitialized(ServletContextEvent event) {  
 initWebApplicationContext(event.getServletContext());  
}

* ContextLoader#initWebApplicationContext

public WebApplicationContext initWebApplicationContext(ServletContext servletContext) {  
 if (servletContext.getAttribute(WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE) != null) {  
 throw new IllegalStateException(  
 "Cannot initialize context because there is already a root application context present - " +  
 "check whether you have multiple ContextLoader\* definitions in your web.xml!");  
 }  
  
 Log logger = LogFactory.getLog(ContextLoader.class);  
 servletContext.log("Initializing Spring root WebApplicationContext");  
 if (logger.isInfoEnabled()) {  
 logger.info("Root WebApplicationContext: initialization started");  
 }  
 long startTime = System.currentTimeMillis();  
  
 try {  
 // Store context in local instance variable, to guarantee that  
 // it is available on ServletContext shutdown.  
 if (this.context == null) {

* // 创建WebApplicationContext  
   this.context = createWebApplicationContext(servletContext);  
   }  
   if (this.context instanceof ConfigurableWebApplicationContext) {  
   ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) this.context;  
   if (!cwac.isActive()) {  
   // The context has not yet been refreshed -> provide services such as  
   // setting the parent context, setting the application context id, etc  
   if (cwac.getParent() == null) {  
   // The context instance was injected without an explicit parent ->  
   // determine parent for root web application context, if any.  
   ApplicationContext parent = loadParentContext(servletContext);  
   cwac.setParent(parent);  
   }  
   configureAndRefreshWebApplicationContext(cwac, servletContext);  
   }  
   }  
   // 记录在ServletContext中 servletContext.setAttribute(WebApplicationContext.ROOT*WEB*APPLICATION*CONTEXT*ATTRIBUTE, this.context);
* ClassLoader ccl = Thread.currentThread().getContextClassLoader();  
   if (ccl == ContextLoader.class.getClassLoader()) {  
   currentContext = this.context;  
   }  
   else if (ccl != null) {  
   currentContextPerThread.put(ccl, this.context);  
   }
* if (logger.isDebugEnabled()) {  
   logger.debug("Published root WebApplicationContext as ServletContext attribute with name [" +  
   WebApplicationContext.ROOT*WEB*APPLICATION*CONTEXT*ATTRIBUTE + "]");  
   }  
   if (logger.isInfoEnabled()) {  
   long elapsedTime = System.currentTimeMillis() - startTime;  
   logger.info("Root WebApplicationContext: initialization completed in " + elapsedTime + " ms");  
   }
* return this.context;  
  }  
  catch (RuntimeException ex) {  
   logger.error("Context initialization failed", ex);  
   servletContext.setAttribute(WebApplicationContext.ROOT*WEB*APPLICATION*CONTEXT*ATTRIBUTE, ex);  
   throw ex;  
  }  
  catch (Error err) {  
   logger.error("Context initialization failed", err);  
   servletContext.setAttribute(WebApplicationContext.ROOT*WEB*APPLICATION*CONTEXT*ATTRIBUTE, err);  
   throw err;  
  }  
  }
* ContextLoader#createWebApplicationContext（创建WebApplicationContext实例)
* protected WebApplicationContext createWebApplicationContext(ServletContext sc) {  
  Class<?> contextClass = determineContextClass(sc);  
  if (!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass)) {  
   throw new ApplicationContextException("Custom context class [" + contextClass.getName() +  
   "] is not of type [" + ConfigurableWebApplicationContext.class.getName() + "]");  
  }  
  return (ConfigurableWebApplicationContext) BeanUtils.instantiateClass(contextClass);  
  }
* determineContextClass
* protected Class<?> determineContextClass(ServletContext servletContext) {  
  String contextClassName = servletContext.getInitParameter(CONTEXT*CLASS*PARAM);
* // 如果在web.xml中配置了contextClass,则直接加载这个类  
  if (contextClassName != null) {  
   try {  
   return ClassUtils.forName(contextClassName, ClassUtils.getDefaultClassLoader());  
   }  
   catch (ClassNotFoundException ex) {  
   throw new ApplicationContextException(  
   "Failed to load custom context class [" + contextClassName + "]", ex);  
   }  
  }  
  else {
* // 否则使用默认值  
   contextClassName = defaultStrategies.getProperty(WebApplicationContext.class.getName());  
   try {  
   return ClassUtils.forName(contextClassName, ContextLoader.class.getClassLoader());  
   }  
   catch (ClassNotFoundException ex) {  
   throw new ApplicationContextException(  
   "Failed to load default context class [" + contextClassName + "]", ex);  
   }  
   }  
  }
* 看defaultStrategies是怎么初始化的
* static {  
  // Load default strategy implementations from properties file.  
  // This is currently strictly internal and not meant to be customized  
  // by application developers.  
  try {  
   ClassPathResource resource = new ClassPathResource(DEFAULT*STRATEGIES*PATH, ContextLoader.class);  
   defaultStrategies = PropertiesLoaderUtils.loadProperties(resource);  
  }  
  catch (IOException ex) {  
   throw new IllegalStateException("Could not load 'ContextLoader.properties': " + ex.getMessage());  
  }  
  }

private static final String DEFAULT\_STRATEGIES\_PATH = "ContextLoader.properties";

* 在ContextLoader目录下有一个配置文件ContextLoader.properties
* 内容为：
* org.springframework.web.context.WebApplicationContext=org.springframework.web.context.support.XmlWebApplicationContext
* 在初始化过程中，程序首先会读取ContextLoader类的同目录下的配置文件ContextLoader.properties，并根据其中的配置提取将要实现WebApplicationContext接口的实现类，并根据这个实现类通过反射的方式进行实例的创建。
* 初始化DispatcherServlet
* 第一次访问网站时，会初始化访问到的servlet。
* 初始化阶段会调用servlet的init方法，在DispatcherServlet中是由其父类HttpServletBean实现的。
* 逻辑：
  + 1)封装及验证初始化参数 解析init-param并封装到PropertyValues中
  + 2)将DispatcherServlet转化为BeanWrapper实例
  + 3)注册相对于Resource的属性编辑器
  + 4)PropertyValues属性注入
  + 5)servletBean的初始化（initServletBean)

public final void init() throws ServletException {  
 if (logger.isDebugEnabled()) {  
 logger.debug("Initializing servlet '" + getServletName() + "'");  
 }  
  
 // Set bean properties from init parameters.  
 try {

* // 解析init-param并封装到PropertyValues中  
   PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(), this.requiredProperties);
* // 把DispatcherServlet转化为一个BeanWrapper，从而能够以Spring的方式来对init-param的值进行注入  
   BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);  
   ResourceLoader resourceLoader = new ServletContextResourceLoader(getServletContext());
* // 设置自定义属性编辑器，如果遇到Resource类型的属性将会使用ResourceEditor进行解析  
   bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoader, getEnvironment()));
* // 空实现，留给子类覆盖  
   initBeanWrapper(bw);
* // 属性注入  
   bw.setPropertyValues(pvs, true);  
   }  
   catch (BeansException ex) {  
   if (logger.isErrorEnabled()) {  
   logger.error("Failed to set bean properties on servlet '" + getServletName() + "'", ex);  
   }  
   throw ex;  
   }
* // Let subclasses do whatever initialization they like.
* // 留给子类扩展  
  initServletBean();
* if (logger.isDebugEnabled()) {  
   logger.debug("Servlet '" + getServletName() + "' configured successfully");  
  }  
  }
  + 1)ServletConfigPropertyValues（封装init-param)

private static class ServletConfigPropertyValues extends MutablePropertyValues {  
  
 /\*\*  
 \* Create new ServletConfigPropertyValues.  
 \* @param config ServletConfig we'll use to take PropertyValues from  
 \* @param requiredProperties set of property names we need, where  
 \* we can't accept default values  
 \* @throws ServletException if any required properties are missing  
 \*/  
 public ServletConfigPropertyValues(ServletConfig config, Set<String> requiredProperties)  
 throws ServletException {  
  
 Set<String> missingProps = (requiredProperties != null && !requiredProperties.isEmpty() ?  
 new HashSet<String>(requiredProperties) : null);  
  
 Enumeration<String> paramNames = config.getInitParameterNames();  
 while (paramNames.hasMoreElements()) {  
 String property = paramNames.nextElement();  
 Object value = config.getInitParameter(property);  
 addPropertyValue(new PropertyValue(property, value));  
 if (missingProps != null) {  
 missingProps.remove(property);  
 }  
 }  
  
 // Fail if we are still missing properties.  
 if (!CollectionUtils.isEmpty(missingProps)) {  
 throw new ServletException(  
 "Initialization from ServletConfig for servlet '" + config.getServletName() +  
 "' failed; the following required properties were missing: " +  
 StringUtils.collectionToDelimitedString(missingProps, ", "));  
 }  
 }  
}

- 2)FrameworkServlet#initServletBean（对WebApplicationContext实例补充初始化)

* protected final void initServletBean() throws ServletException {  
  getServletContext().log("Initializing Spring FrameworkServlet '" + getServletName() + "'");  
  if (this.logger.isInfoEnabled()) {  
   this.logger.info("FrameworkServlet '" + getServletName() + "': initialization started");  
  }  
  long startTime = System.currentTimeMillis();
* try {  
   this.webApplicationContext = initWebApplicationContext();
* // 留给子类覆盖  
   initFrameworkServlet();  
   }  
   catch (ServletException ex) {  
   this.logger.error("Context initialization failed", ex);  
   throw ex;  
   }  
   catch (RuntimeException ex) {  
   this.logger.error("Context initialization failed", ex);  
   throw ex;  
   }
* if (this.logger.isInfoEnabled()) {  
   long elapsedTime = System.currentTimeMillis() - startTime;  
   this.logger.info("FrameworkServlet '" + getServletName() + "': initialization completed in " +  
   elapsedTime + " ms");  
   }  
  }
  + 2.1)FrameworkServlet#initWebApplicationContext
* 创建或刷新WebApplicationContext实例，并对servlet功能所使用的变量进行初始化
* 逻辑：
  + 1)寻找或创建对应的WebApplicationContext实例
* 通过构造函数的注入进行初始化
* 通过contextAttribute进行初始化
* 通过在web.xml中配置的servlet参数contextAttribute来查找ServletContext中对应的属性，默认为WebApplicationContext.class.getName() + “.ROOT”，也就是在ContextLoaderListener加载时会创建WebApplicationContext实例，并将实例以
* WebApplicationContext.class.getName() + “.ROOT”为key放入ServletContext中。
* 重新创建WebApplicationContext实例
  + 2)对已经创建的WebApplicationContext实例进行配置和刷新
  + 3)刷新Spring在Web功能实现中必须使用的全局变量
* protected WebApplicationContext initWebApplicationContext() {  
  WebApplicationContext rootContext =  
   WebApplicationContextUtils.getWebApplicationContext(getServletContext());  
  WebApplicationContext wac = null;
* if (this.webApplicationContext != null) {
* // applicationContext实例在构造函数中被注入  
   // A context instance was injected at construction time -> use it  
   wac = this.webApplicationContext;  
   if (wac instanceof ConfigurableWebApplicationContext) {  
   ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) wac;  
   if (!cwac.isActive()) {  
   // The context has not yet been refreshed -> provide services such as  
   // setting the parent context, setting the application context id, etc  
   if (cwac.getParent() == null) {  
   // The context instance was injected without an explicit parent -> set  
   // the root application context (if any; may be null) as the parent  
   cwac.setParent(rootContext);  
   }
* // 刷新上下文环境  
   configureAndRefreshWebApplicationContext(cwac);  
   }  
  }  
  }  
  if (wac == null) {  
   // No context instance was injected at construction time -> see if one  
   // has been registered in the servlet context. If one exists, it is assumed  
   // that the parent context (if any) has already been set and that the  
   // user has performed any initialization such as setting the context id
* // 根据contextAttribute属性加载WebApplicationContext   
   wac = findWebApplicationContext();  
  }  
  if (wac == null) {  
   // No context instance is defined for this servlet -> create a local one
* // 重新创建WebApplicationContext  
   wac = createWebApplicationContext(rootContext);  
   }
* if (!this.refreshEventReceived) {  
   // Either the context is not a ConfigurableApplicationContext with refresh  
   // support or the context injected at construction time had already been  
   // refreshed -> trigger initial onRefresh manually here.
* // 刷新Spring在web功能实现中必须使用的全局变量  
   onRefresh(wac);  
   }
* if (this.publishContext) {  
   // Publish the context as a servlet context attribute.  
   String attrName = getServletContextAttributeName();  
   getServletContext().setAttribute(attrName, wac);  
   if (this.logger.isDebugEnabled()) {  
   this.logger.debug("Published WebApplicationContext of servlet '" + getServletName() +  
   "' as ServletContext attribute with name [" + attrName + "]");  
   }  
   }
* return wac;  
  }
  + 2.1.1) （与IOC衔接)configureAndRefreshWebApplicationContext
* protected void configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac) {  
  if (ObjectUtils.identityToString(wac).equals(wac.getId())) {  
   // The application context id is still set to its original default value  
   // -> assign a more useful id based on available information  
   if (this.contextId != null) {  
   wac.setId(this.contextId);  
   }  
   else {  
   // Generate default id...  
   wac.setId(ConfigurableWebApplicationContext.APPLICATION*CONTEXT*ID\_PREFIX +  
   ObjectUtils.getDisplayString(getServletContext().getContextPath()) + '/' + getServletName());  
   }  
  }
* wac.setServletContext(getServletContext());  
  wac.setServletConfig(getServletConfig());  
  wac.setNamespace(getNamespace());  
  wac.addApplicationListener(new SourceFilteringListener(wac, new ContextRefreshListener()));
* // The wac environment's #initPropertySources will be called in any case when the context  
  // is refreshed; do it eagerly here to ensure servlet property sources are in place for  
  // use in any post-processing or initialization that occurs below prior to #refresh  
  ConfigurableEnvironment env = wac.getEnvironment();  
  if (env instanceof ConfigurableWebEnvironment) {  
   ((ConfigurableWebEnvironment) env).initPropertySources(getServletContext(), getServletConfig());  
  }
* postProcessWebApplicationContext(wac);  
  applyInitializers(wac);
* // 加载配置文件以及整合parent到wac（就是ApplicationContext中的refresh方法)  
  wac.refresh();  
  }
  + 2.1.2) findWebApplicationContext
* protected WebApplicationContext findWebApplicationContext() {  
  String attrName = getContextAttribute();  
  if (attrName == null) {  
   return null;  
  }  
  WebApplicationContext wac = WebApplicationContextUtils.getWebApplicationContext(getServletContext(), attrName);  
  if (wac == null) {  
   throw new IllegalStateException("No WebApplicationContext found: initializer not registered?");  
  }  
  return wac;  
  }
  + 2.1.3) FrameworkServlet#createWebApplicationContext
* protected WebApplicationContext createWebApplicationContext(ApplicationContext parent) {
* // 获取servlet的初始化参数contextClass，如果没有配置默认为XMLWebApplicationContext.class  
  Class<?> contextClass = getContextClass();  
  if (this.logger.isDebugEnabled()) {  
   this.logger.debug("Servlet with name '" + getServletName() +  
   "' will try to create custom WebApplicationContext context of class '" +  
   contextClass.getName() + "'" + ", using parent context [" + parent + "]");  
  }  
  if (!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass)) {  
   throw new ApplicationContextException(  
   "Fatal initialization error in servlet with name '" + getServletName() +  
   "': custom WebApplicationContext class [" + contextClass.getName() +  
   "] is not of type ConfigurableWebApplicationContext");  
  }
* // 通过反射实例化contextClass  
  ConfigurableWebApplicationContext wac =  
   (ConfigurableWebApplicationContext) BeanUtils.instantiateClass(contextClass);
* wac.setEnvironment(getEnvironment());
* // parent为在ContextLoaderListener中创建的实例  
  wac.setParent(parent);
  + // 获取contextConfigLocation属性，配置在servlet初始化参数中  
    wac.setConfigLocation(getContextConfigLocation());  
    // 初始化Spring环境包括加载配置文件等（即为2.1.1)  
    configureAndRefreshWebApplicationContext(wac);
* return wac;  
  }
  + 2.1.4) DispatcherServlet#onRefresh
* protected void onRefresh(ApplicationContext context) {  
  initStrategies(context);  
  }
* HandlerMapping：请求和处理器之间的映射，用于获取HandlerExecutionChain
* HandlerAdapter：实际的请求处理器，处理后返回ModelAndView
* HandlerExceptionResolver：异常处理器，当拦截器的postHandle方法调用后检查异常。
* ViewResolver：视图解析器,解析视图名，得到View，由逻辑视图变为物理视图。
* protected void initStrategies(ApplicationContext context) {
* // 初始化文件上传模块  
  initMultipartResolver(context);
* // 初始化国际化模块  
  initLocaleResolver(context);
* // 初始化主题模块  
  initThemeResolver(context);
* // 初始化HandlerMappings  
  initHandlerMappings(context);
* // 初始化HandlerAdapters  
  initHandlerAdapters(context);
* // 初始化HandlerExceptionResolvers  
  initHandlerExceptionResolvers(context);  
  initRequestToViewNameTranslator(context);
* // 初始化ViewResolvers  
  initViewResolvers(context);  
  initFlashMapManager(context);  
  }
  + 2.1.4.1) initHandlerMappings
* 当客户端发出Request时DispatcherServlet会将Request提交给HandlerMapping，然后HandlerMapping根据WebApplicationContext的配置，回传给DispatcherServlet相应的Controller。
* 在基于SpringMVC的web应用程序中，我们可以为DispatcherServlet提供多个HandlerMapping供其使用。DispatcherServlet在选用HandlerMapping的过程中，将根据我们所指定的一系列HandlerMapping的优先级进行排序，然后优先使用优先级在前的HandlerMapping。如果当前的HandlerMapping能够返回可用的Handler，则使用当前的Handler进行Web请求的处理，而不再继续询问其他的HandlerMapping。否则，将继续按照各个HandlerMapping的优先级询问，直到获取一个可用的Handler为止。
* 默认情况下，SpringMVC将加载当前系统中所有实现了HandlerMapping接口的bean。如果只期望SpringMVC加载指定的handlerMapping时，可以修改web.xml中的DispatcherServlet的初始参数，将detectAllHandlerMappings的值设置为false。
* 此时，SpingMVC将查找名为handlerMapping的bean，并作为当前系统中唯一的handlerMapping。如果没有定义handlerMapping的话，则SpringMVC将按照DispatcherServlet所在目录下的DispatcherServlet.properties中所定义的内容来加载默认的handlerMapping。

private void initHandlerMappings(ApplicationContext context) {  
 this.handlerMappings = null;  
  
 if (this.detectAllHandlerMappings) {  
 // Find all HandlerMappings in the ApplicationContext, including ancestor contexts.  
 Map<String, HandlerMapping> matchingBeans =  
 BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerMapping.class, true, false);  
 if (!matchingBeans.isEmpty()) {  
 this.handlerMappings = new ArrayList<HandlerMapping>(matchingBeans.values());  
 // We keep HandlerMappings in sorted order.  
 AnnotationAwareOrderComparator.sort(this.handlerMappings);  
 }  
 }  
 else {  
 try {  
 HandlerMapping hm = context.getBean(HANDLER\_MAPPING\_BEAN\_NAME, HandlerMapping.class);  
 this.handlerMappings = Collections.singletonList(hm);  
 }  
 catch (NoSuchBeanDefinitionException ex) {

* // Ignore, we'll add a default HandlerMapping later.  
   }  
   }
* // Ensure we have at least one HandlerMapping, by registering  
   // a default HandlerMapping if no other mappings are found.  
   if (this.handlerMappings == null) {  
   this.handlerMappings = getDefaultStrategies(context, HandlerMapping.class);  
   if (logger.isDebugEnabled()) {  
   logger.debug("No HandlerMappings found in servlet '" + getServletName() + "': using default");  
   }  
   }  
  }
* 默认的handlerMappings有
* org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping, org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping
* getBean时会调用afterPropertiesSet

public void afterPropertiesSet() {  
 initHandlerMethods();  
}

* + 2.1.4.1.1) AbstractHandlerMethodMapping#initHandlerMethods（怎么把Controller里的各个方法封装为HandlerMethod的)
* protected void initHandlerMethods() {  
  if (logger.isDebugEnabled()) {  
   logger.debug("Looking for request mappings in application context: " + getApplicationContext());  
  }  
  String[] beanNames = (this.detectHandlerMethodsInAncestorContexts ?  
   BeanFactoryUtils.beanNamesForTypeIncludingAncestors(getApplicationContext(), Object.class) :  
   getApplicationContext().getBeanNamesForType(Object.class));
* for (String beanName : beanNames) {  
   if (!beanName.startsWith(SCOPED*TARGET*NAME\_PREFIX)) {  
   Class<?> beanType = null;  
   try {  
   beanType = getApplicationContext().getType(beanName);  
   }  
   catch (Throwable ex) {  
   // An unresolvable bean type, probably from a lazy bean - let's ignore it.  
   if (logger.isDebugEnabled()) {  
   logger.debug("Could not resolve target class for bean with name '" + beanName + "'", ex);  
   }  
   }  
   if (beanType != null && isHandler(beanType)) {  
   detectHandlerMethods(beanName);  
   }  
   }  
  }  
  handlerMethodsInitialized(getHandlerMethods());  
  }
* 重点！凡是有Controller或者RequestMapping注解的Class类都被视为HandlerMethod，之后会遍历该类的Method检查是否符合要求。
* protected boolean isHandler(Class<?> beanType) {  
  return (AnnotatedElementUtils.hasAnnotation(beanType, Controller.class) ||  
   AnnotatedElementUtils.hasAnnotation(beanType, RequestMapping.class));  
  }
  + 2.1.4.1.1.1) detectHandlerMethods

protected void detectHandlerMethods(final Object handler) {  
 Class<?> handlerType = (handler instanceof String ?  
 getApplicationContext().getType((String) handler) : handler.getClass());  
 final Class<?> userType = ClassUtils.getUserClass(handlerType);  
  
 Map<Method, T> methods = MethodIntrospector.selectMethods(userType,  
 new MethodIntrospector.MetadataLookup<T>() {  
 @Override  
 public T inspect(Method method) {  
 try {  
 return getMappingForMethod(method, userType);  
 }  
 catch (Throwable ex) {  
 throw new IllegalStateException("Invalid mapping on handler class [" +  
 userType.getName() + "]: " + method, ex);  
 }  
 }  
 });  
  
 if (logger.isDebugEnabled()) {  
 logger.debug(methods.size() + " request handler methods found on " + userType + ": " + methods);  
 }  
 for (Map.Entry<Method, T> entry : methods.entrySet()) {  
 Method invocableMethod = AopUtils.selectInvocableMethod(entry.getKey(), userType);  
 T mapping = entry.getValue();  
 registerHandlerMethod(handler, invocableMethod, mapping);  
 }  
}

- 2.1.4.1.1.1.1) MethodIntrospector#selectMethods

public static <T> Map<Method, T> selectMethods(Class<?> targetType, final MetadataLookup<T> metadataLookup) {  
 final Map<Method, T> methodMap = new LinkedHashMap<Method, T>();  
 Set<Class<?>> handlerTypes = new LinkedHashSet<Class<?>>();  
 Class<?> specificHandlerType = null;  
  
 if (!Proxy.isProxyClass(targetType)) {  
 handlerTypes.add(targetType);  
 specificHandlerType = targetType;  
 }  
 handlerTypes.addAll(Arrays.asList(targetType.getInterfaces()));  
  
 for (Class<?> currentHandlerType : handlerTypes) {  
 final Class<?> targetClass = (specificHandlerType != null ? specificHandlerType : currentHandlerType);  
  
 ReflectionUtils.doWithMethods(currentHandlerType, new ReflectionUtils.MethodCallback() {  
 @Override  
 public void doWith(Method method) {  
 Method specificMethod = ClassUtils.getMostSpecificMethod(method, targetClass);  
 T result = metadataLookup.inspect(specificMethod);  
 if (result != null) {  
 Method bridgedMethod = BridgeMethodResolver.findBridgedMethod(specificMethod);  
 if (bridgedMethod == specificMethod || metadataLookup.inspect(bridgedMethod) == null) {  
 methodMap.put(specificMethod, result);  
 }  
 }  
 }  
 }, ReflectionUtils.USER\_DECLARED\_METHODS);  
 }  
  
 return methodMap;  
}

- 2.1.4.1.1.1.1.1) doWithMethods

public static void doWithMethods(Class<?> clazz, MethodCallback mc, MethodFilter mf) {  
 // Keep backing up the inheritance hierarchy.  
 Method[] methods = getDeclaredMethods(clazz);  
 for (Method method : methods) {  
 if (mf != null && !mf.matches(method)) {  
 continue;  
 }  
 try {  
 mc.doWith(method);  
 }  
 catch (IllegalAccessException ex) {  
 throw new IllegalStateException("Not allowed to access method '" + method.getName() + "': " + ex);  
 }  
 }  
 if (clazz.getSuperclass() != null) {  
 doWithMethods(clazz.getSuperclass(), mc, mf);  
 }  
 else if (clazz.isInterface()) {  
 for (Class<?> superIfc : clazz.getInterfaces()) {  
 doWithMethods(superIfc, mc, mf);  
 }  
 }  
}

- 2.1.4.1.1.1.1.1.1) metadataLookup#inspect（根据Method找到RequestMapping注解信息)

public T inspect(Method method) {  
 try {  
 return getMappingForMethod(method, userType);  
 }  
 catch (Throwable ex) {  
 throw new IllegalStateException("Invalid mapping on handler class [" +  
 userType.getName() + "]: " + method, ex);  
 }  
}

* protected RequestMappingInfo getMappingForMethod(Method method, Class<?> handlerType) {  
  RequestMappingInfo info = createRequestMappingInfo(method);  
  if (info != null) {  
   RequestMappingInfo typeInfo = createRequestMappingInfo(handlerType);  
   if (typeInfo != null) {  
   info = typeInfo.combine(info);  
   }  
  }  
  return info;  
  }

private RequestMappingInfo createRequestMappingInfo(AnnotatedElement element) {  
 RequestMapping requestMapping = AnnotatedElementUtils.findMergedAnnotation(element, RequestMapping.class);  
 RequestCondition<?> condition = (element instanceof Class ?  
 getCustomTypeCondition((Class<?>) element) : getCustomMethodCondition((Method) element));  
 return (requestMapping != null ? createRequestMappingInfo(requestMapping, condition) : null);  
}

* protected RequestMappingInfo createRequestMappingInfo(  
   RequestMapping requestMapping, RequestCondition<?> customCondition) {
* return RequestMappingInfo  
   .paths(resolveEmbeddedValuesInPatterns(requestMapping.path()))  
   .methods(requestMapping.method())  
   .params(requestMapping.params())  
   .headers(requestMapping.headers())  
   .consumes(requestMapping.consumes())  
   .produces(requestMapping.produces())  
   .mappingName(requestMapping.name())  
   .customCondition(customCondition)  
   .options(this.config)  
   .build();  
  }

- 2.1.4.1.1.1.2) registerHandlerMethod

* protected void registerHandlerMethod(Object handler, Method method, T mapping) {  
  this.mappingRegistry.register(mapping, handler, method);  
  }
  + 2.1.4.1.1.1.2.1) MappingRegistry#register

public void register(T mapping, Object handler, Method method) {  
 this.readWriteLock.writeLock().lock();  
 try {  
 HandlerMethod handlerMethod = createHandlerMethod(handler, method);  
 assertUniqueMethodMapping(handlerMethod, mapping);  
  
 if (logger.isInfoEnabled()) {  
 logger.info("Mapped \"" + mapping + "\" onto " + handlerMethod);  
 }  
 this.mappingLookup.put(mapping, handlerMethod);  
  
 List<String> directUrls = getDirectUrls(mapping);  
 for (String url : directUrls) {  
 this.urlLookup.add(url, mapping);  
 }  
  
 String name = null;  
 if (getNamingStrategy() != null) {  
 name = getNamingStrategy().getName(handlerMethod, mapping);  
 addMappingName(name, handlerMethod);  
 }  
  
 CorsConfiguration corsConfig = initCorsConfiguration(handler, method, mapping);  
 if (corsConfig != null) {  
 this.corsLookup.put(handlerMethod, corsConfig);  
 }  
  
 this.registry.put(mapping, new MappingRegistration<T>(mapping, handlerMethod, directUrls, name));  
 }  
 finally {  
 this.readWriteLock.writeLock().unlock();  
 }  
}

- 2.1.4.1.1.1.2.1.1) AbstractHandlerMethodMapping#createHandlerMethod

* protected HandlerMethod createHandlerMethod(Object handler, Method method) {  
  HandlerMethod handlerMethod;  
  if (handler instanceof String) {  
   String beanName = (String) handler;  
   handlerMethod = new HandlerMethod(beanName,  
   getApplicationContext().getAutowireCapableBeanFactory(), method);  
  }  
  else {  
   handlerMethod = new HandlerMethod(handler, method);  
  }  
  return handlerMethod;  
  }

public HandlerMethod(String beanName, BeanFactory beanFactory, Method method) {  
 Assert.hasText(beanName, "Bean name is required");  
 Assert.notNull(beanFactory, "BeanFactory is required");  
 Assert.notNull(method, "Method is required");  
 this.bean = beanName;  
 this.beanFactory = beanFactory;  
 this.beanType = ClassUtils.getUserClass(beanFactory.getType(beanName));  
 this.method = method;  
 this.bridgedMethod = BridgeMethodResolver.findBridgedMethod(method);

* // 初始化方法参数  
  this.parameters = initMethodParameters();  
  this.resolvedFromHandlerMethod = null;  
  }

private MethodParameter[] initMethodParameters() {  
 int count = this.bridgedMethod.getParameterTypes().length;  
 MethodParameter[] result = new MethodParameter[count];  
 for (int i = 0; i < count; i++) {  
 HandlerMethodParameter parameter = new HandlerMethodParameter(i);  
 GenericTypeResolver.resolveParameterType(parameter, this.beanType);  
 result[i] = parameter;  
 }  
 return result;  
}

* + 2.1.4.2) initHandlerAdapters
* detectAllHandlerAdapters这个变量和detectAllHandlerMappings作用差不多。

private void initHandlerAdapters(ApplicationContext context) {  
 this.handlerAdapters = null;  
  
 if (this.detectAllHandlerAdapters) {  
 // Find all HandlerAdapters in the ApplicationContext, including ancestor contexts.  
 Map<String, HandlerAdapter> matchingBeans =  
 BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerAdapter.class, true, false);  
 if (!matchingBeans.isEmpty()) {  
 this.handlerAdapters = new ArrayList<HandlerAdapter>(matchingBeans.values());  
 // We keep HandlerAdapters in sorted order.  
 AnnotationAwareOrderComparator.sort(this.handlerAdapters);  
 }  
 }  
 else {  
 try {  
 HandlerAdapter ha = context.getBean(HANDLER\_ADAPTER\_BEAN\_NAME, HandlerAdapter.class);  
 this.handlerAdapters = Collections.singletonList(ha);  
 }  
 catch (NoSuchBeanDefinitionException ex) {  
 // Ignore, we'll add a default HandlerAdapter later.  
 }  
 }  
  
 // Ensure we have at least some HandlerAdapters, by registering  
 // default HandlerAdapters if no other adapters are found.  
 if (this.handlerAdapters == null) {  
 this.handlerAdapters = getDefaultStrategies(context, HandlerAdapter.class);  
 if (logger.isDebugEnabled()) {  
 logger.debug("No HandlerAdapters found in servlet '" + getServletName() + "': using default");  
 }  
 }  
}

* getDefaultStrategies(context, HandlerAdapter.class)
* defaultStrategies是从配置文件DispatcherServlet.properties中加载得到的。
* 默认有以下HandlerAdapter：
* org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter, org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter, org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter
* protected List getDefaultStrategies(ApplicationContext context, Class strategyInterface) {  
  String key = strategyInterface.getName();  
  String value = defaultStrategies.getProperty(key);  
  if (value != null) {  
   String[] classNames = StringUtils.commaDelimitedListToStringArray(value);  
   List strategies = new ArrayList(classNames.length);  
   for (String className : classNames) {  
   try {  
   Class<?> clazz = ClassUtils.forName(className, DispatcherServlet.class.getClassLoader());  
   Object strategy = createDefaultStrategy(context, clazz);  
   strategies.add((T) strategy);  
   }  
   catch (ClassNotFoundException ex) {  
   throw new BeanInitializationException(  
   "Could not find DispatcherServlet's default strategy class [" + className +  
   "] for interface [" + key + "]", ex);  
   }  
   catch (LinkageError err) {  
   throw new BeanInitializationException(  
   "Error loading DispatcherServlet's default strategy class [" + className +  
   "] for interface [" + key + "]: problem with class file or dependent class", err);  
   }  
   }  
   return strategies;  
  }  
  else {  
   return new LinkedList();  
  }  
  }
* HttpRequestHandlerAdapter（HTTP请求处理器适配器)
* 仅仅支持对HTTP请求处理器的适配，它简单地将HTTP请求和响应对象传递给HTTP请求处理器的实现，并不需要返回值，主要应用在基于HTTP的远程调用的实现上。
* SimpleControllerHandlerAdapter（简单控制器处理器适配器)
* 这个实现类将HTTP请求适配到一个Controller的实现进行处理。这里控制器的实现是一个简单的Controller接口的实现。SimpleControllerHandlerAdapter 被设计成一个框架类的实现，不需要被改写，客户化的业务逻辑通常是在Controller接口的实现类中实现的。
* AnnotationMethodHandlerAdapter（注解方法处理器适配器)
* 这个类的实现是基于注解的实现，它需要结合注解方法映射和注解方法处理器协同工作。它通过解析声明在注解控制器的请求映射信息来解析相应的处理器方法来处理当前的HTTP请求。
* 在处理的过程中，它通过反射来发现探测处理器方法的参数，调用处理器方法，并且映射返回值到模型和控制器对象，最后返回模型和控制器对象给DispatcherServlet。
* RequestMappingHandlerAdapter（请求映射处理器适配器)
* 初始化该Adapter时初始化一些解析器：

public void afterPropertiesSet() {  
 // Do this first, it may add ResponseBody advice beans  
 initControllerAdviceCache();  
  
 if (this.argumentResolvers == null) {  
 List<HandlerMethodArgumentResolver> resolvers = getDefaultArgumentResolvers();  
 this.argumentResolvers = new HandlerMethodArgumentResolverComposite().addResolvers(resolvers);  
 }  
 if (this.initBinderArgumentResolvers == null) {  
 List<HandlerMethodArgumentResolver> resolvers = getDefaultInitBinderArgumentResolvers();  
 this.initBinderArgumentResolvers = new HandlerMethodArgumentResolverComposite().addResolvers(resolvers);  
 }  
 if (this.returnValueHandlers == null) {  
 List<HandlerMethodReturnValueHandler> handlers = getDefaultReturnValueHandlers();  
 this.returnValueHandlers = new HandlerMethodReturnValueHandlerComposite().addHandlers(handlers);  
 }  
}

private List<HandlerMethodArgumentResolver> getDefaultArgumentResolvers() {  
 List<HandlerMethodArgumentResolver> resolvers = new ArrayList<HandlerMethodArgumentResolver>();  
  
 // Annotation-based argument resolution  
 resolvers.add(new RequestParamMethodArgumentResolver(getBeanFactory(), false));  
 resolvers.add(new RequestParamMapMethodArgumentResolver());  
 resolvers.add(new PathVariableMethodArgumentResolver());  
 resolvers.add(new PathVariableMapMethodArgumentResolver());  
 resolvers.add(new MatrixVariableMethodArgumentResolver());  
 resolvers.add(new MatrixVariableMapMethodArgumentResolver());  
 resolvers.add(new ServletModelAttributeMethodProcessor(false));  
 resolvers.add(new RequestResponseBodyMethodProcessor(getMessageConverters(), this.requestResponseBodyAdvice));  
 resolvers.add(new RequestPartMethodArgumentResolver(getMessageConverters(), this.requestResponseBodyAdvice));  
 resolvers.add(new RequestHeaderMethodArgumentResolver(getBeanFactory()));  
 resolvers.add(new RequestHeaderMapMethodArgumentResolver());  
 resolvers.add(new ServletCookieValueMethodArgumentResolver(getBeanFactory()));  
 resolvers.add(new ExpressionValueMethodArgumentResolver(getBeanFactory()));  
 resolvers.add(new SessionAttributeMethodArgumentResolver());  
 resolvers.add(new RequestAttributeMethodArgumentResolver());  
  
 // Type-based argument resolution  
 resolvers.add(new ServletRequestMethodArgumentResolver());  
 resolvers.add(new ServletResponseMethodArgumentResolver());  
 resolvers.add(new HttpEntityMethodProcessor(getMessageConverters(), this.requestResponseBodyAdvice));  
 resolvers.add(new RedirectAttributesMethodArgumentResolver());  
 resolvers.add(new ModelMethodProcessor());  
 resolvers.add(new MapMethodProcessor());  
 resolvers.add(new ErrorsMethodArgumentResolver());  
 resolvers.add(new SessionStatusMethodArgumentResolver());  
 resolvers.add(new UriComponentsBuilderMethodArgumentResolver());  
  
 // Custom arguments  
 if (getCustomArgumentResolvers() != null) {  
 resolvers.addAll(getCustomArgumentResolvers());  
 }  
  
 // Catch-all  
 resolvers.add(new RequestParamMethodArgumentResolver(getBeanFactory(), true));  
 resolvers.add(new ServletModelAttributeMethodProcessor(true));  
  
 return resolvers;  
}

* Spring中所使用的Handler并没有任何特殊的联系，为了统一处理，Spring提供了不同情况下的适配器。
  + 2.1.4.3) initHandlerExceptionResolvers
* HanlderExceptionResolver是可以被用户定制的，只要实现该接口，实现resolveException方法并定义为一个bean即可。
* resolveException方法返回一个ModelAndView对象，在方法内部对异常的类型进行判断，然后尝试生成对应的ModelAndView对象，如果该方法返回null，则Spring会继续寻找其他实现了HanlderExceptionResolver接口的bean。

private void initHandlerExceptionResolvers(ApplicationContext context) {  
 this.handlerExceptionResolvers = null;  
  
 if (this.detectAllHandlerExceptionResolvers) {  
 // Find all HandlerExceptionResolvers in the ApplicationContext, including ancestor contexts.  
 Map<String, HandlerExceptionResolver> matchingBeans = BeanFactoryUtils  
 .beansOfTypeIncludingAncestors(context, HandlerExceptionResolver.class, true, false);  
 if (!matchingBeans.isEmpty()) {  
 this.handlerExceptionResolvers = new ArrayList<HandlerExceptionResolver>(matchingBeans.values());  
 // We keep HandlerExceptionResolvers in sorted order.  
 AnnotationAwareOrderComparator.sort(this.handlerExceptionResolvers);  
 }  
 }  
 else {  
 try {  
 HandlerExceptionResolver her =  
 context.getBean(HANDLER\_EXCEPTION\_RESOLVER\_BEAN\_NAME, HandlerExceptionResolver.class);  
 this.handlerExceptionResolvers = Collections.singletonList(her);  
 }  
 catch (NoSuchBeanDefinitionException ex) {  
 // Ignore, no HandlerExceptionResolver is fine too.  
 }  
 }  
  
 // Ensure we have at least some HandlerExceptionResolvers, by registering  
 // default HandlerExceptionResolvers if no other resolvers are found.  
 if (this.handlerExceptionResolvers == null) {  
 this.handlerExceptionResolvers = getDefaultStrategies(context, HandlerExceptionResolver.class);  
 if (logger.isDebugEnabled()) {  
 logger.debug("No HandlerExceptionResolvers found in servlet '" + getServletName() + "': using default");  
 }  
 }  
}

- 2.1.4.4) initRequestToViewNameTranslator

* 当Controller方法没有返回一个View或者逻辑视图名称，并且在该方法中没有直接往response输出流里面写数据的时候，Spring就会按照约定好的方式提供一个逻辑视图名称（最简答的情况就是加prefix和suffix)。
* 这个逻辑视图名称是通过Spring定义的RequestToViewNameTranslator接口的getViewName方法来实现的，用户也可以自定义自己的RequestToViewNameTranslator。
* Spring为我们提供了一个默认的实现DefaultRequestToViewNameTranslator。

public static final String REQUEST\_TO\_VIEW\_NAME\_TRANSLATOR\_BEAN\_NAME = "viewNameTranslator";

private void initRequestToViewNameTranslator(ApplicationContext context) {  
 try {  
 this.viewNameTranslator =  
 context.getBean(REQUEST\_TO\_VIEW\_NAME\_TRANSLATOR\_BEAN\_NAME, RequestToViewNameTranslator.class);  
 if (logger.isDebugEnabled()) {  
 logger.debug("Using RequestToViewNameTranslator [" + this.viewNameTranslator + "]");  
 }  
 }  
 catch (NoSuchBeanDefinitionException ex) {  
 // We need to use the default.  
 this.viewNameTranslator = getDefaultStrategy(context, RequestToViewNameTranslator.class);  
 if (logger.isDebugEnabled()) {  
 logger.debug("Unable to locate RequestToViewNameTranslator with name '" +  
 REQUEST\_TO\_VIEW\_NAME\_TRANSLATOR\_BEAN\_NAME + "': using default [" + this.viewNameTranslator +  
 "]");  
 }  
 }  
}

- 2.1.4.5) initViewResolvers

* 当Controller将请求处理结果放到ModelAndView中以后，DispatcherServlet会根据ModelAndView选择合适的视图进行渲染。ViewResolver接口定义了resolveViewName方法，根据viewName创建合适类型的View实现。
* 可以在Spring的配置文件中定义viewResolver。

private void initViewResolvers(ApplicationContext context) {  
 this.viewResolvers = null;  
  
 if (this.detectAllViewResolvers) {  
 // Find all ViewResolvers in the ApplicationContext, including ancestor contexts.  
 Map<String, ViewResolver> matchingBeans =  
 BeanFactoryUtils.beansOfTypeIncludingAncestors(context, ViewResolver.class, true, false);  
 if (!matchingBeans.isEmpty()) {  
 this.viewResolvers = new ArrayList<ViewResolver>(matchingBeans.values());  
 // We keep ViewResolvers in sorted order.  
 AnnotationAwareOrderComparator.sort(this.viewResolvers);  
 }  
 }  
 else {  
 try {  
 ViewResolver vr = context.getBean(VIEW\_RESOLVER\_BEAN\_NAME, ViewResolver.class);  
 this.viewResolvers = Collections.singletonList(vr);  
 }  
 catch (NoSuchBeanDefinitionException ex) {  
 // Ignore, we'll add a default ViewResolver later.  
 }  
 }  
  
 // Ensure we have at least one ViewResolver, by registering  
 // a default ViewResolver if no other resolvers are found.  
 if (this.viewResolvers == null) {  
 this.viewResolvers = getDefaultStrategies(context, ViewResolver.class);  
 if (logger.isDebugEnabled()) {  
 logger.debug("No ViewResolvers found in servlet '" + getServletName() + "': using default");  
 }  
 }  
}

- 2.1.4.6) initFlashMapManager

* flash attributes提供了一个请求存储属性，可供其他请求使用。在使用重定向的时候非常必要。
* Spring MVC 有两个主要的抽象来支持 flash attributes。 FlashMap 用于保持 flash attributes 而 FlashMapManager用于存储，检索，管理FlashMap 实例。
* Flash attribute 支持默认开启，并不需要显式启用，它永远不会导致HTTP Session的创建。 每一个请求都有一个 “input”FlashMap 具有从上一个请求（如果有的话)传过来的属性和一个 “output” FlashMap 具有将要在后续请求中保存的属性。 这两个 FlashMap 实例都可以通过静态方法RequestContextUtils从Spring MVC的任何位置访问。

private void initFlashMapManager(ApplicationContext context) {  
 try {  
 this.flashMapManager = context.getBean(FLASH\_MAP\_MANAGER\_BEAN\_NAME, FlashMapManager.class);  
 if (logger.isDebugEnabled()) {  
 logger.debug("Using FlashMapManager [" + this.flashMapManager + "]");  
 }  
 }  
 catch (NoSuchBeanDefinitionException ex) {  
 // We need to use the default.  
 this.flashMapManager = getDefaultStrategy(context, FlashMapManager.class);  
 if (logger.isDebugEnabled()) {  
 logger.debug("Unable to locate FlashMapManager with name '" +  
 FLASH\_MAP\_MANAGER\_BEAN\_NAME + "': using default [" + this.flashMapManager + "]");  
 }  
 }  
}

* 处理请求
* FrameworkServlet#service（入口)
* DispatcherServlet（FrameworkServlet的子类)无论是doGet、doPost等方法都会调用processRequest方法处理请求。
* FrameworkServlet#processRequest
* 逻辑：
  + 1)为了保证当前线程的LocaleContext和RequestAttributes可以在当前请求处理完毕后还能恢复，提取当前线程的两个属性
  + 2)根据当前request创建对应的LocaleContext和RequestAttributes，并绑定到当前线程
  + 3)委托给doService方法进一步处理
  + 4)请求处理结束后恢复线程到原始状态
  + 5)请求处理结束后无论成功与否都会发布事件通知
* protected final void processRequest(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {
* long startTime = System.currentTimeMillis();  
   Throwable failureCause = null;
* LocaleContext previousLocaleContext = LocaleContextHolder.getLocaleContext();  
   LocaleContext localeContext = buildLocaleContext(request);
* RequestAttributes previousAttributes = RequestContextHolder.getRequestAttributes();  
   ServletRequestAttributes requestAttributes = buildRequestAttributes(request, response, previousAttributes);
* WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);  
   asyncManager.registerCallableInterceptor(FrameworkServlet.class.getName(), new RequestBindingInterceptor());  
   // 将localeContext和requestAttributes绑定到当前线程  
   initContextHolders(request, localeContext, requestAttributes);
* try {  
   doService(request, response);  
   }  
   catch (ServletException ex) {  
   failureCause = ex;  
   throw ex;  
   }  
   catch (IOException ex) {  
   failureCause = ex;  
   throw ex;  
   }  
   catch (Throwable ex) {  
   failureCause = ex;  
   throw new NestedServletException("Request processing failed", ex);  
   }
* finally {
* // 恢复线程到原始状态  
   resetContextHolders(request, previousLocaleContext, previousAttributes);  
   if (requestAttributes != null) {  
   requestAttributes.requestCompleted();  
   }
* if (logger.isDebugEnabled()) {  
   if (failureCause != null) {  
   this.logger.debug("Could not complete request", failureCause);  
   }  
   else {  
   if (asyncManager.isConcurrentHandlingStarted()) {  
   logger.debug("Leaving response open for concurrent processing");  
   }  
   else {  
   this.logger.debug("Successfully completed request");  
   }  
   }  
  }  
  // 发布事件通知  
  publishRequestHandledEvent(request, response, startTime, failureCause);
* }  
  }
* DispatcherServlet#doService
* 将已经初始化的功能辅助工具变量设置在request属性中。
* 主要业务逻辑是在doDispatch方法中处理。
* protected void doService(HttpServletRequest request, HttpServletResponse response) throws Exception {  
  if (logger.isDebugEnabled()) {  
   String resumed = WebAsyncUtils.getAsyncManager(request).hasConcurrentResult() ? " resumed" : "";  
   logger.debug("DispatcherServlet with name '" + getServletName() + "'" + resumed +  
   " processing " + request.getMethod() + " request for [" + getRequestUri(request) + "]");  
  }
* // Keep a snapshot of the request attributes in case of an include,  
  // to be able to restore the original attributes after the include.  
  Map<String, Object> attributesSnapshot = null;  
  if (WebUtils.isIncludeRequest(request)) {  
   attributesSnapshot = new HashMap<String, Object>();  
   Enumeration<?> attrNames = request.getAttributeNames();  
   while (attrNames.hasMoreElements()) {  
   String attrName = (String) attrNames.nextElement();  
   if (this.cleanupAfterInclude || attrName.startsWith("org.springframework.web.servlet")) {  
   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());
* FlashMap inputFlashMap = this.flashMapManager.retrieveAndUpdate(request, response);  
  if (inputFlashMap != null) {  
   request.setAttribute(INPUT*FLASH*MAP*ATTRIBUTE, Collections.unmodifiableMap(inputFlashMap));*  
  *}*  
  *request.setAttribute(OUTPUT*FLASH*MAP*ATTRIBUTE, new FlashMap());  
  request.setAttribute(FLASH*MAP*MANAGER\_ATTRIBUTE, this.flashMapManager);
* try {  
   doDispatch(request, response);  
  }  
  finally {  
   if (!WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {  
   // Restore the original attribute snapshot, in case of an include.  
   if (attributesSnapshot != null) {  
   restoreAttributesAfterInclude(request, attributesSnapshot);  
   }  
   }  
  }  
  }
* doDispatch（主体)
* protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {  
  HttpServletRequest processedRequest = request;  
  HandlerExecutionChain mappedHandler = null;  
  boolean multipartRequestParsed = false;
* WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
* try {  
   ModelAndView mv = null;  
   Exception dispatchException = null;
* try {
* // 如果request是MultiPartContent类型的，则将其转为MultiPartHttpServletRequest  
   processedRequest = checkMultipart(request);  
   multipartRequestParsed = (processedRequest != request);  
   // 根据request信息寻找对应的Handler  
   mappedHandler = getHandler(processedRequest);  
   if (mappedHandler == null || mappedHandler.getHandler() == null) {
* // 没有找到对应的handler，则通过response反馈错误信息  
   noHandlerFound(processedRequest, response);  
   return;  
   }  
   // 根据当前的handler寻找对应的HandlerAdapter  
   HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());  
   // 如果当前handler支持Last-Modified请求头，则对其进行处理  
   // Process last-modified header, if supported by the handler.  
   String method = request.getMethod();  
   boolean isGet = "GET".equals(method);  
   if (isGet || "HEAD".equals(method)) {  
   long lastModified = ha.getLastModified(request, mappedHandler.getHandler());  
   if (logger.isDebugEnabled()) {  
   logger.debug("Last-Modified value for [" + getRequestUri(request) + "] is: " + lastModified);  
   }  
   if (new ServletWebRequest(request, response).checkNotModified(lastModified) && isGet) {  
   return;  
   }  
   }  
   // 调用拦截器的preHandle方法  
   if (!mappedHandler.applyPreHandle(processedRequest, response)) {  
   return;  
   }  
   // 调用handler并返回视图  
   mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
* if (asyncManager.isConcurrentHandlingStarted()) {  
   return;  
   }  
   // 转换视图名称（加prefix和suffix)  
   applyDefaultViewName(processedRequest, mv);
* // 调用拦截器的postHandle方法  
   mappedHandler.applyPostHandle(processedRequest, response, mv);  
   }  
   catch (Exception ex) {  
   dispatchException = ex;  
   }  
   catch (Throwable err) {  
   // As of 4.3, we're processing Errors thrown from handler methods as well,  
   // making them available for @ExceptionHandler methods and other scenarios.  
   dispatchException = new NestedServletException("Handler dispatch failed", err);  
   }  
   // 处理handle的结果  
   processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);  
  }  
  catch (Exception ex) {  
   triggerAfterCompletion(processedRequest, response, mappedHandler, ex);  
  }  
  catch (Throwable err) {  
   triggerAfterCompletion(processedRequest, response, mappedHandler,  
   new NestedServletException("Handler processing failed", err));  
  }  
  finally {  
   if (asyncManager.isConcurrentHandlingStarted()) {  
   // Instead of postHandle and afterCompletion  
   if (mappedHandler != null) {  
   mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);  
   }  
   }  
   else {  
   // Clean up any resources used by a multipart request.  
   if (multipartRequestParsed) {  
   cleanupMultipart(processedRequest);  
   }  
   }  
  }  
  }
  + 1) checkMultiPart（处理文件上传请求)
* protected HttpServletRequest checkMultipart(HttpServletRequest request) throws MultipartException {  
  if (this.multipartResolver != null && this.multipartResolver.isMultipart(request)) {  
   if (WebUtils.getNativeRequest(request, MultipartHttpServletRequest.class) != null) {  
   logger.debug("Request is already a MultipartHttpServletRequest - if not in a forward, " +  
   "this typically results from an additional MultipartFilter in web.xml");  
   }  
   else if (hasMultipartException(request) ) {  
   logger.debug("Multipart resolution failed for current request before - " +  
   "skipping re-resolution for undisturbed error rendering");  
   }  
   else {  
   try {  
   return this.multipartResolver.resolveMultipart(request);  
   }  
   catch (MultipartException ex) {  
   if (request.getAttribute(WebUtils.ERROR*EXCEPTION*ATTRIBUTE) != null) {  
   logger.debug("Multipart resolution failed for error dispatch", ex);  
   // Keep processing error dispatch with regular request handle below  
   }  
   else {  
   throw ex;  
   }  
   }  
   }  
  }  
  // If not returned before: return original request.  
  return request;  
  }
* MultipartResolver.resolveMultipart
* MultipartHttpServletRequest resolveMultipart(HttpServletRequest request) throws MultipartException;
  + 2) getHandler（获取HandlerExecutionChain)
* protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {  
  for (HandlerMapping hm : this.handlerMappings) {  
   if (logger.isTraceEnabled()) {  
   logger.trace(  
   "Testing handler map [" + hm + "] in DispatcherServlet with name '" + getServletName() + "'");  
   }  
   HandlerExecutionChain handler = hm.getHandler(request);  
   if (handler != null) {  
   return handler;  
   }  
  }  
  return null;  
  }
* HandlerMapping.getHandler
* HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception;
  + 2.1) AbstractHandlerMapping#getHandler

public final HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {

* // 根据request获取对应的handler  
  Object handler = getHandlerInternal(request);  
  if (handler == null) {
* // 如果没有对应的request的handler则使用默认的handler  
   handler = getDefaultHandler();  
   }
* // 没有默认的handler则无法继续处理  
  if (handler == null) {  
   return null;  
  }
* // 当查找的Controller为String，就意味着返回的是配置的bean名称，需要根据bean名称查找对应的bean  
  // Bean name or resolved handler?  
  if (handler instanceof String) {  
   String handlerName = (String) handler;  
   handler = getApplicationContext().getBean(handlerName);  
  }
* HandlerExecutionChain executionChain = getHandlerExecutionChain(handler, request);  
  if (CorsUtils.isCorsRequest(request)) {  
   CorsConfiguration globalConfig = this.corsConfigSource.getCorsConfiguration(request);  
   CorsConfiguration handlerConfig = getCorsConfiguration(handler, request);  
   CorsConfiguration config = (globalConfig != null ? globalConfig.combine(handlerConfig) : handlerConfig);  
   executionChain = getCorsHandlerExecutionChain(request, executionChain, config);  
  }  
  return executionChain;  
  }
* 2.1.1-a) AbstractUrlHandlerMapping#getHandlerInternal
* 以AbstractUrlHandlerMapping为例
* protected Object getHandlerInternal(HttpServletRequest request) throws Exception {
* // 截取用于匹配url的有效路径  
  String lookupPath = getUrlPathHelper().getLookupPathForRequest(request);
* // 根据路径寻找handler  
  Object handler = lookupHandler(lookupPath, request);  
  if (handler == null) {  
   // We need to care for the default handler directly, since we need to  
   // expose the PATH*WITHIN*HANDLER*MAPPING*ATTRIBUTE for it as well.  
   Object rawHandler = null;  
   if ("/".equals(lookupPath)) {
* // 如果请求路径是/，那么使用RootHandler进行处理  
   rawHandler = getRootHandler();  
   }  
   if (rawHandler == null) {
* // 如果无法找到handler，则使用默认handler  
   rawHandler = getDefaultHandler();  
   }  
   if (rawHandler != null) {  
   // Bean name or resolved handler?  
   if (rawHandler instanceof String) {
* // 根据beanName寻找对应的beanName  
   String handlerName = (String) rawHandler;  
   rawHandler = getApplicationContext().getBean(handlerName);  
   }  
   validateHandler(rawHandler, request);  
   // 模板方法
* handler = buildPathExposingHandler(rawHandler, lookupPath, lookupPath, null);  
   }  
   }  
   if (handler != null && logger.isDebugEnabled()) {  
   logger.debug("Mapping [" + lookupPath + "] to " + handler);  
   }  
   else if (handler == null && logger.isTraceEnabled()) {  
   logger.trace("No handler mapping found for [" + lookupPath + "]");  
   }  
   return handler;  
  }
  + 2.1.1.1) lookupHandler
* protected Object lookupHandler(String urlPath, HttpServletRequest request) throws Exception {  
  // Direct match?
  + // 直接匹配  
    Object handler = this.handlerMap.get(urlPath);  
    if (handler != null) {  
    // Bean name or resolved handler?  
    if (handler instanceof String) {  
     String handlerName = (String) handler;  
     handler = getApplicationContext().getBean(handlerName);  
    }  
    validateHandler(handler, request);  
    return buildPathExposingHandler(handler, urlPath, urlPath, null);  
    }  
    // 通配符匹配  
    // Pattern match?  
    List matchingPatterns = new ArrayList();  
    for (String registeredPattern : this.handlerMap.keySet()) {  
    if (getPathMatcher().match(registeredPattern, urlPath)) {  
     matchingPatterns.add(registeredPattern);  
    }  
    else if (useTrailingSlashMatch()) {  
     if (!registeredPattern.endsWith("/") && getPathMatcher().match(registeredPattern + "/", urlPath)) {  
     matchingPatterns.add(registeredPattern +"/");  
     }  
    }  
    }
* String bestMatch = null;  
  Comparator patternComparator = getPathMatcher().getPatternComparator(urlPath);  
  if (!matchingPatterns.isEmpty()) {  
   Collections.sort(matchingPatterns, patternComparator);  
   if (logger.isDebugEnabled()) {  
   logger.debug("Matching patterns for request [" + urlPath + "] are " + matchingPatterns);  
   }  
   bestMatch = matchingPatterns.get(0);  
  }  
  if (bestMatch != null) {  
   handler = this.handlerMap.get(bestMatch);  
   if (handler == null) {  
   if (bestMatch.endsWith("/")) {  
   handler = this.handlerMap.get(bestMatch.substring(0, bestMatch.length() - 1));  
   }  
   if (handler == null) {  
   throw new IllegalStateException(  
   "Could not find handler for best pattern match [" + bestMatch + "]");  
   }  
   }  
   // Bean name or resolved handler?  
   if (handler instanceof String) {  
   String handlerName = (String) handler;  
   handler = getApplicationContext().getBean(handlerName);  
   }  
   validateHandler(handler, request);  
   String pathWithinMapping = getPathMatcher().extractPathWithinPattern(bestMatch, urlPath);
* // There might be multiple 'best patterns', let's make sure we have the correct URI template variables  
   // for all of them  
   Map<String, String> uriTemplateVariables = new LinkedHashMap<String, String>();  
   for (String matchingPattern : matchingPatterns) {  
   if (patternComparator.compare(bestMatch, matchingPattern) == 0) {  
   Map<String, String> vars = getPathMatcher().extractUriTemplateVariables(matchingPattern, urlPath);  
   Map<String, String> decodedVars = getUrlPathHelper().decodePathVariables(request, vars);  
   uriTemplateVariables.putAll(decodedVars);  
   }  
   }  
   if (logger.isDebugEnabled()) {  
   logger.debug("URI Template variables for request [" + urlPath + "] are " + uriTemplateVariables);  
   }  
   return buildPathExposingHandler(handler, bestMatch, pathWithinMapping, uriTemplateVariables);  
  }
* // No handler found...  
  return null;  
  }
  + 2.1.1.1.1) buildPathExposingHandler（将handler封装为HandlerExecutionChain，并添加两个拦截器)
* protected Object buildPathExposingHandler(Object rawHandler, String bestMatchingPattern,  
   String pathWithinMapping, Map<String, String> uriTemplateVariables) {
* HandlerExecutionChain chain = new HandlerExecutionChain(rawHandler);  
   chain.addInterceptor(new PathExposingHandlerInterceptor(bestMatchingPattern, pathWithinMapping));  
   if (!CollectionUtils.isEmpty(uriTemplateVariables)) {  
   chain.addInterceptor(new UriTemplateVariablesHandlerInterceptor(uriTemplateVariables));  
   }  
   return chain;  
  }
* 2.1.1-b) AbstractHandlerMethodMapping#getHandlerInternal（加入拦截器)
* protected HandlerMethod getHandlerInternal(HttpServletRequest request) throws Exception {  
  String lookupPath = getUrlPathHelper().getLookupPathForRequest(request);  
  if (logger.isDebugEnabled()) {  
   logger.debug("Looking up handler method for path " + lookupPath);  
  }  
  this.mappingRegistry.acquireReadLock();  
  try {  
   HandlerMethod handlerMethod = lookupHandlerMethod(lookupPath, request);  
   if (logger.isDebugEnabled()) {  
   if (handlerMethod != null) {  
   logger.debug("Returning handler method [" + handlerMethod + "]");  
   }  
   else {  
   logger.debug("Did not find handler method for [" + lookupPath + "]");  
   }  
   }  
   return (handlerMethod != null ? handlerMethod.createWithResolvedBean() : null);  
  }  
  finally {  
   this.mappingRegistry.releaseReadLock();  
  }  
  }
  + 2.1.1.1) lookupHandlerMethod
* protected HandlerMethod lookupHandlerMethod(String lookupPath, HttpServletRequest request) throws Exception {  
  List matches = new ArrayList();
* // 直接匹配  
  List directPathMatches = this.mappingRegistry.getMappingsByUrl(lookupPath);  
  if (directPathMatches != null) {  
   addMatchingMappings(directPathMatches, matches, request);  
  }  
  if (matches.isEmpty()) {  
   // No choice but to go through all mappings...
* // 加入所有的映射  
   addMatchingMappings(this.mappingRegistry.getMappings().keySet(), matches, request);  
   }
* if (!matches.isEmpty()) {  
   Comparator comparator = new MatchComparator(getMappingComparator(request));
* // 按匹配程度排序  
   Collections.sort(matches, comparator);  
   if (logger.isTraceEnabled()) {  
   logger.trace("Found " + matches.size() + " matching mapping(s) for [" +  
   lookupPath + "] : " + matches);  
   }
  + // 得到最符合的匹配结果  
    Match bestMatch = matches.get(0);  
    if (matches.size() > 1) {  
     if (CorsUtils.isPreFlightRequest(request)) {  
     return PREFLIGHT*AMBIGUOUS*MATCH;  
     }  
     Match secondBestMatch = matches.get(1);  
     if (comparator.compare(bestMatch, secondBestMatch) == 0) {  
     Method m1 = bestMatch.handlerMethod.getMethod();  
     Method m2 = secondBestMatch.handlerMethod.getMethod();  
     throw new IllegalStateException("Ambiguous handler methods mapped for HTTP path '" +  
     request.getRequestURL() + "': {" + m1 + ", " + m2 + "}");  
     }  
    }  
    handleMatch(bestMatch.mapping, lookupPath, request);  
    return bestMatch.handlerMethod;  
     }  
     else {  
    return handleNoMatch(this.mappingRegistry.getMappings().keySet(), lookupPath, request);  
     }  
    }
  + 2.1.2) getHandlerExecutionChain（加入拦截器)
* 将handler实例和所有匹配的拦截器封装到HandlerExecutionChain中。
* protected HandlerExecutionChain getHandlerExecutionChain(Object handler, HttpServletRequest request) {  
  HandlerExecutionChain chain = (handler instanceof HandlerExecutionChain ?  
   (HandlerExecutionChain) handler : new HandlerExecutionChain(handler));
* String lookupPath = this.urlPathHelper.getLookupPathForRequest(request);  
  for (HandlerInterceptor interceptor : this.adaptedInterceptors) {  
   if (interceptor instanceof MappedInterceptor) {  
   MappedInterceptor mappedInterceptor = (MappedInterceptor) interceptor;  
   if (mappedInterceptor.matches(lookupPath, this.pathMatcher)) {  
   chain.addInterceptor(mappedInterceptor.getInterceptor());  
   }  
   }  
   else {  
   chain.addInterceptor(interceptor);  
   }  
  }  
  return chain;  
  }
  + 3) noHandlerFound（没有找到HandlerExecutionChain)
* protected void noHandlerFound(HttpServletRequest request, HttpServletResponse response) throws Exception {  
  if (pageNotFoundLogger.isWarnEnabled()) {  
   pageNotFoundLogger.warn("No mapping found for HTTP request with URI [" + getRequestUri(request) +  
   "] in DispatcherServlet with name '" + getServletName() + "'");  
  }
* // 默认为false  
  if (this.throwExceptionIfNoHandlerFound) {  
   throw new NoHandlerFoundException(request.getMethod(), getRequestUri(request),  
   new ServletServerHttpRequest(request).getHeaders());  
  }  
  else {  
   response.sendError(HttpServletResponse.SC*NOT*FOUND);  
  }  
  }
  + 4) getHandlerAdapter（根据HandlerExecutionChain获取HandlerAdapter)
* protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {  
  for (HandlerAdapter ha : this.handlerAdapters) {  
   if (logger.isTraceEnabled()) {  
   logger.trace("Testing handler adapter [" + ha + "]");  
   }  
   if (ha.supports(handler)) {  
   return ha;  
   }  
  }  
  throw new ServletException("No adapter for handler [" + handler +  
   "]: The DispatcherServlet configuration needs to include a HandlerAdapter that supports this handler");  
  }
* 4.1-a) SimpleControllerHandlerAdapter#supports

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

* 4.1-b) AbstractHandlerMethodAdapter#supports

public final boolean supports(Object handler) {  
 return (handler instanceof HandlerMethod && supportsInternal((HandlerMethod) handler));  
}

- 5) HandlerExecutionChain#applyPreHandle（拦截器preHandle)

* boolean applyPreHandle(HttpServletRequest request, HttpServletResponse response) throws Exception {  
  HandlerInterceptor[] interceptors = getInterceptors();  
  if (!ObjectUtils.isEmpty(interceptors)) {  
   for (int i = 0; i < interceptors.length; i++) {  
   HandlerInterceptor interceptor = interceptors[i];  
   if (!interceptor.preHandle(request, response, this.handler)) {  
   triggerAfterCompletion(request, response, null);  
   return false;  
   }  
   this.interceptorIndex = i;  
   }  
  }  
  return true;  
  }
  + 6) HandlerAdapter#handle（处理请求)
* ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception;
* 6.1-a) SimpleControllerHandlerAdapter.handle

public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)  
 throws Exception {  
 return ((Controller) handler).handleRequest(request, response);  
}

- 6.1.1) AbstractController#handleRequest

public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)  
 throws Exception {  
  
 if (HttpMethod.OPTIONS.matches(request.getMethod())) {  
 response.setHeader("Allow", getAllowHeader());  
 return null;  
 }  
  
 // Delegate to WebContentGenerator for checking and preparing.  
 checkRequest(request);  
 prepareResponse(response);  
 // 如果需要session内的同步执行  
 // Execute handleRequestInternal in synchronized block if required.  
 if (this.synchronizeOnSession) {  
 HttpSession session = request.getSession(false);  
 if (session != null) {  
 Object mutex = WebUtils.getSessionMutex(session);  
 synchronized (mutex) {

* // 调用用户的逻辑  
   return handleRequestInternal(request, response);  
   }  
  }  
  }  
  // 调用用户的逻辑  
  return handleRequestInternal(request, response);  
  }
* 6.1-b) AbstractHandlerMethodAdapter#handle

public final ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)  
 throws Exception {  
  
 return handleInternal(request, response, (HandlerMethod) handler);  
}

- 6.1.1) RequestMappingHandlerAdapter#handleInternal

* protected ModelAndView handleInternal(HttpServletRequest request,  
   HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {
* ModelAndView mav;  
   checkRequest(request);
* // Execute invokeHandlerMethod in synchronized block if required.  
   if (this.synchronizeOnSession) {  
   HttpSession session = request.getSession(false);  
   if (session != null) {  
   Object mutex = WebUtils.getSessionMutex(session);  
   synchronized (mutex) {  
   mav = invokeHandlerMethod(request, response, handlerMethod);  
   }  
   }  
   else {  
   // No HttpSession available -> no mutex necessary  
   mav = invokeHandlerMethod(request, response, handlerMethod);  
   }  
   }  
   else {  
   // No synchronization on session demanded at all...  
   mav = invokeHandlerMethod(request, response, handlerMethod);  
   }
* if (!response.containsHeader(HEADER*CACHE*CONTROL)) {  
   if (getSessionAttributesHandler(handlerMethod).hasSessionAttributes()) {  
   applyCacheSeconds(response, this.cacheSecondsForSessionAttributeHandlers);  
   }  
   else {  
   prepareResponse(response);  
   }  
   }
* return mav;  
  }
  + 6.1.1.1) invokeHandlerMethod
* protected ModelAndView invokeHandlerMethod(HttpServletRequest request,  
   HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {
* ServletWebRequest webRequest = new ServletWebRequest(request, response);  
   try {
* WebDataBinderFactory binderFactory = getDataBinderFactory(handlerMethod);  
  ModelFactory modelFactory = getModelFactory(handlerMethod, binderFactory);
* ServletInvocableHandlerMethod invocableMethod = createInvocableHandlerMethod(handlerMethod);  
  invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);  
  invocableMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandlers);  
  invocableMethod.setDataBinderFactory(binderFactory);  
  invocableMethod.setParameterNameDiscoverer(this.parameterNameDiscoverer);
* ModelAndViewContainer mavContainer = new ModelAndViewContainer();  
  mavContainer.addAllAttributes(RequestContextUtils.getInputFlashMap(request));  
  modelFactory.initModel(webRequest, mavContainer, invocableMethod);  
  mavContainer.setIgnoreDefaultModelOnRedirect(this.ignoreDefaultModelOnRedirect);
* AsyncWebRequest asyncWebRequest = WebAsyncUtils.createAsyncWebRequest(request, response);  
  asyncWebRequest.setTimeout(this.asyncRequestTimeout);
* WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);  
  asyncManager.setTaskExecutor(this.taskExecutor);  
  asyncManager.setAsyncWebRequest(asyncWebRequest);  
  asyncManager.registerCallableInterceptors(this.callableInterceptors);  
  asyncManager.registerDeferredResultInterceptors(this.deferredResultInterceptors);
* if (asyncManager.hasConcurrentResult()) {  
   Object result = asyncManager.getConcurrentResult();  
   mavContainer = (ModelAndViewContainer) asyncManager.getConcurrentResultContext()[0];  
   asyncManager.clearConcurrentResult();  
   if (logger.isDebugEnabled()) {  
   logger.debug("Found concurrent result value [" + result + "]");  
   }  
   invocableMethod = invocableMethod.wrapConcurrentResult(result);  
  }
* invocableMethod.invokeAndHandle(webRequest, mavContainer);  
  if (asyncManager.isConcurrentHandlingStarted()) {  
   return null;  
  }
* return getModelAndView(mavContainer, modelFactory, webRequest);  
  }  
  finally {  
   webRequest.requestCompleted();  
  }  
  }
* protected ServletInvocableHandlerMethod createInvocableHandlerMethod(HandlerMethod handlerMethod) {  
  return new ServletInvocableHandlerMethod(handlerMethod);  
  }

private ModelAndView getModelAndView(ModelAndViewContainer mavContainer,  
 ModelFactory modelFactory, NativeWebRequest webRequest) throws Exception {  
  
 modelFactory.updateModel(webRequest, mavContainer);  
 if (mavContainer.isRequestHandled()) {  
 return null;  
 }

* // 将返回值ModelMap转为ModelAndView  
  ModelMap model = mavContainer.getModel();  
  ModelAndView mav = new ModelAndView(mavContainer.getViewName(), model, mavContainer.getStatus());  
  if (!mavContainer.isViewReference()) {  
   mav.setView((View) mavContainer.getView());  
  }  
  if (model instanceof RedirectAttributes) {  
   Map<String, ?> flashAttributes = ((RedirectAttributes) model).getFlashAttributes();  
   HttpServletRequest request = webRequest.getNativeRequest(HttpServletRequest.class);  
   RequestContextUtils.getOutputFlashMap(request).putAll(flashAttributes);  
  }  
  return mav;  
  }
  + 6.1.1.1.1) ServletInvocableHandlerMethod#invokeAndHandle

public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer,  
 Object... providedArgs) throws Exception {  
  
 Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);  
 setResponseStatus(webRequest);  
  
 if (returnValue == null) {  
 if (isRequestNotModified(webRequest) || hasResponseStatus() || mavContainer.isRequestHandled()) {  
 mavContainer.setRequestHandled(true);  
 return;  
 }  
 }  
 else if (StringUtils.hasText(this.responseReason)) {  
 mavContainer.setRequestHandled(true);  
 return;  
 }  
  
 mavContainer.setRequestHandled(false);  
 try {  
 this.returnValueHandlers.handleReturnValue(  
 returnValue, getReturnValueType(returnValue), mavContainer, webRequest);  
 }  
 catch (Exception ex) {  
 if (logger.isTraceEnabled()) {  
 logger.trace(getReturnValueHandlingErrorMessage("Error handling return value", returnValue), ex);  
 }  
 throw ex;  
 }  
}

* + 6.1.1.1.1.1) InvocableHandlerMethod#invokeForRequest

public Object invokeForRequest(NativeWebRequest request, ModelAndViewContainer mavContainer,  
 Object... providedArgs) throws Exception {  
  
 Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);  
 if (logger.isTraceEnabled()) {  
 logger.trace("Invoking '" + ClassUtils.getQualifiedMethodName(getMethod(), getBeanType()) +  
 "' with arguments " + Arrays.toString(args));  
 }  
 Object returnValue = doInvoke(args);  
 if (logger.isTraceEnabled()) {  
 logger.trace("Method [" + ClassUtils.getQualifiedMethodName(getMethod(), getBeanType()) +  
 "] returned [" + returnValue + "]");  
 }  
 return returnValue;  
}

- 6.1.1.1.1.1.1) （处理参数)getMethodArgumentValues（方法参数注入，返回值即为方法参数)

private Object[] getMethodArgumentValues(NativeWebRequest request, ModelAndViewContainer mavContainer,  
 Object... providedArgs) throws Exception {  
 // 遍历请求参数，然后委托给HandlerMethodArgumentResolver#resolveArgument进行参数解析  
 MethodParameter[] parameters = getMethodParameters();  
 Object[] args = new Object[parameters.length];  
 for (int i = 0; i < parameters.length; i++) {  
 MethodParameter parameter = parameters[i];  
 parameter.initParameterNameDiscovery(this.parameterNameDiscoverer);  
 args[i] = resolveProvidedArgument(parameter, providedArgs);  
 if (args[i] != null) {  
 continue;  
 }  
 if (this.argumentResolvers.supportsParameter(parameter)) {  
 try {  
 args[i] = this.argumentResolvers.resolveArgument(  
 parameter, mavContainer, request, this.dataBinderFactory);  
 continue;  
 }  
 catch (Exception ex) {  
 if (logger.isDebugEnabled()) {  
 logger.debug(getArgumentResolutionErrorMessage("Failed to resolve", i), ex);  
 }  
 throw ex;  
 }  
 }  
 if (args[i] == null) {  
 throw new IllegalStateException("Could not resolve method parameter at index " +  
 parameter.getParameterIndex() + " in " + parameter.getMethod().toGenericString() +  
 ": " + getArgumentResolutionErrorMessage("No suitable resolver for", i));  
 }  
 }  
 return args;  
}

- 6.1.1.1.1.1.1.1) HandlerMethodArgumentResolverComposite#resolveArgument（将形参解析为实参)

public Object resolveArgument(MethodParameter parameter, ModelAndViewContainer mavContainer,  
 NativeWebRequest webRequest, WebDataBinderFactory binderFactory) throws Exception {  
  
 HandlerMethodArgumentResolver resolver = getArgumentResolver(parameter);  
 if (resolver == null) {  
 throw new IllegalArgumentException("Unknown parameter type [" + parameter.getParameterType().getName() + "]");  
 }  
 return resolver.resolveArgument(parameter, mavContainer, webRequest, binderFactory);  
}

- 6.1.1.1.1.1.1.1.1) getArgumentResolver

private HandlerMethodArgumentResolver getArgumentResolver(MethodParameter parameter) {  
 HandlerMethodArgumentResolver result = this.argumentResolverCache.get(parameter);  
 if (result == null) {  
 for (HandlerMethodArgumentResolver methodArgumentResolver : this.argumentResolvers) {  
 if (logger.isTraceEnabled()) {  
 logger.trace("Testing if argument resolver [" + methodArgumentResolver + "] supports [" +  
 parameter.getGenericParameterType() + "]");  
 }  
 if (methodArgumentResolver.supportsParameter(parameter)) {  
 result = methodArgumentResolver;  
 this.argumentResolverCache.put(parameter, result);  
 break;  
 }  
 }  
 }  
 return result;  
}

- 6.1.1.1.1.1.1.1.2) （例子)RequestParamMethodArgumentResolver

public boolean supportsParameter(MethodParameter parameter) {  
 if (parameter.hasParameterAnnotation(RequestParam.class)) {  
 if (Map.class.isAssignableFrom(parameter.nestedIfOptional().getNestedParameterType())) {  
 String paramName = parameter.getParameterAnnotation(RequestParam.class).name();  
 return StringUtils.hasText(paramName);  
 }  
 else {  
 return true;  
 }  
 }  
 else {  
 if (parameter.hasParameterAnnotation(RequestPart.class)) {  
 return false;  
 }  
 parameter = parameter.nestedIfOptional();  
 if (MultipartResolutionDelegate.isMultipartArgument(parameter)) {  
 return true;  
 }  
 else if (this.useDefaultResolution) {  
 return BeanUtils.isSimpleProperty(parameter.getNestedParameterType());  
 }  
 else {  
 return false;  
 }  
 }  
}

* AbstractNamedValueMethodArgumentResolver#resolveArgument

public final Object resolveArgument(MethodParameter parameter, ModelAndViewContainer mavContainer,  
 NativeWebRequest webRequest, WebDataBinderFactory binderFactory) throws Exception {  
  
 NamedValueInfo namedValueInfo = getNamedValueInfo(parameter);  
 MethodParameter nestedParameter = parameter.nestedIfOptional();  
  
 Object resolvedName = resolveStringValue(namedValueInfo.name);  
 if (resolvedName == null) {  
 throw new IllegalArgumentException(  
 "Specified name must not resolve to null: [" + namedValueInfo.name + "]");  
 }  
  
 Object arg = resolveName(resolvedName.toString(), nestedParameter, webRequest);  
 if (arg == null) {  
 if (namedValueInfo.defaultValue != null) {  
 arg = resolveStringValue(namedValueInfo.defaultValue);  
 }  
 else if (namedValueInfo.required && !nestedParameter.isOptional()) {  
 handleMissingValue(namedValueInfo.name, nestedParameter, webRequest);  
 }  
 arg = handleNullValue(namedValueInfo.name, arg, nestedParameter.getNestedParameterType());  
 }  
 else if ("".equals(arg) && namedValueInfo.defaultValue != null) {  
 arg = resolveStringValue(namedValueInfo.defaultValue);  
 }  
  
 if (binderFactory != null) {  
 WebDataBinder binder = binderFactory.createBinder(webRequest, null, namedValueInfo.name);  
 try {

* // WebDataBinder进行数据转换  
   arg = binder.convertIfNecessary(arg, parameter.getParameterType(), parameter);  
   }  
   catch (ConversionNotSupportedException ex) {  
   throw new MethodArgumentConversionNotSupportedException(arg, ex.getRequiredType(),  
   namedValueInfo.name, parameter, ex.getCause());  
   }  
   catch (TypeMismatchException ex) {  
   throw new MethodArgumentTypeMismatchException(arg, ex.getRequiredType(),  
   namedValueInfo.name, parameter, ex.getCause());
* }  
  }
* handleResolvedValue(arg, namedValueInfo.name, parameter, mavContainer, webRequest);
* return arg;  
  }
  + 6.1.1.1.1.1.1.1.2.1) resolveName
* protected Object resolveName(String name, MethodParameter parameter, NativeWebRequest request) throws Exception {  
  HttpServletRequest servletRequest = request.getNativeRequest(HttpServletRequest.class);  
  MultipartHttpServletRequest multipartRequest =  
   WebUtils.getNativeRequest(servletRequest, MultipartHttpServletRequest.class);
* Object mpArg = MultipartResolutionDelegate.resolveMultipartArgument(name, parameter, servletRequest);  
  if (mpArg != MultipartResolutionDelegate.UNRESOLVABLE) {  
   return mpArg;  
  }
* Object arg = null;  
  if (multipartRequest != null) {  
   List files = multipartRequest.getFiles(name);  
   if (!files.isEmpty()) {  
   arg = (files.size() == 1 ? files.get(0) : files);  
   }  
  }  
  if (arg == null) {  
   String[] paramValues = request.getParameterValues(name);  
   if (paramValues != null) {  
   arg = (paramValues.length == 1 ? paramValues[0] : paramValues);  
   }  
  }  
  return arg;  
  }
  + 6.1.1.1.1.1.1.1.2.2) WebDataBinder#convertIfNecessary

public <T> T convertIfNecessary(Object value, Class<T> requiredType, MethodParameter methodParam)  
 throws TypeMismatchException {  
  
 return getTypeConverter().convertIfNecessary(value, requiredType, methodParam);  
}

- 6.1.1.1.1.1.2)（处理调用) InvocableHandlerMethod#doInvoke

* protected Object doInvoke(Object... args) throws Exception {  
  ReflectionUtils.makeAccessible(getBridgedMethod());  
  try {
* // Controller中的方法，method.invoke  
   return getBridgedMethod().invoke(getBean(), args);  
   }  
   catch (IllegalArgumentException ex) {  
   assertTargetBean(getBridgedMethod(), getBean(), args);  
   String text = (ex.getMessage() != null ? ex.getMessage() : "Illegal argument");  
   throw new IllegalStateException(getInvocationErrorMessage(text, args), ex);  
   }  
   catch (InvocationTargetException ex) {  
   // Unwrap for HandlerExceptionResolvers ...  
   Throwable targetException = ex.getTargetException();  
   if (targetException instanceof RuntimeException) {  
   throw (RuntimeException) targetException;  
   }  
   else if (targetException instanceof Error) {  
   throw (Error) targetException;  
   }  
   else if (targetException instanceof Exception) {  
   throw (Exception) targetException;  
   }  
   else {  
   String text = getInvocationErrorMessage("Failed to invoke handler method", args);  
   throw new IllegalStateException(text, targetException);  
   }  
   }  
  }
  + 6.1.1.1.1.2) （处理返回值)HandlerMethodReturnValueHandler#handleReturnValue
* 以HandlerMethodReturnValueHandlerComposite为例：

public void handleReturnValue(Object returnValue, MethodParameter returnType,  
 ModelAndViewContainer mavContainer, NativeWebRequest webRequest) throws Exception {  
  
 HandlerMethodReturnValueHandler handler = selectHandler(returnValue, returnType);  
 if (handler == null) {  
 throw new IllegalArgumentException("Unknown return value type: " + returnType.getParameterType().getName());  
 }  
 handler.handleReturnValue(returnValue, returnType, mavContainer, webRequest);  
}

- 6.1.1.1.1.2.1) selectHandler

private HandlerMethodReturnValueHandler selectHandler(Object value, MethodParameter returnType) {  
 boolean isAsyncValue = isAsyncReturnValue(value, returnType);  
 for (HandlerMethodReturnValueHandler handler : this.returnValueHandlers) {  
 if (isAsyncValue && !(handler instanceof AsyncHandlerMethodReturnValueHandler)) {  
 continue;  
 }  
 if (handler.supportsReturnType(returnType)) {  
 return handler;  
 }  
 }  
 return null;  
}

* 如果返回值是ModelAndView，那么handler是ModelAndViewMethodReturnValueHandler。
* 如果返回值是普通的对象(@ResponseBody)，那么handler是
* RequestResponseBodyMethodProcessor。
  + 6.1.1.1.1.2.2) handleReturnValue
* 以RequestResponseBodyMethodProcessor为例：

public void handleReturnValue(Object returnValue, MethodParameter returnType,  
 ModelAndViewContainer mavContainer, NativeWebRequest webRequest)  
 throws IOException, HttpMediaTypeNotAcceptableException, HttpMessageNotWritableException {  
  
 mavContainer.setRequestHandled(true);  
 ServletServerHttpRequest inputMessage = createInputMessage(webRequest);  
 ServletServerHttpResponse outputMessage = createOutputMessage(webRequest);  
  
 // Try even with null return value. ResponseBodyAdvice could get involved.  
 writeWithMessageConverters(returnValue, returnType, inputMessage, outputMessage);  
}

* 以ModelAndViewMethodReturnValueHandler为例：

public void handleReturnValue(Object returnValue, MethodParameter returnType,  
 ModelAndViewContainer mavContainer, NativeWebRequest webRequest) throws Exception {  
  
 if (returnValue == null) {  
 mavContainer.setRequestHandled(true);  
 return;  
 }  
  
 ModelAndView mav = (ModelAndView) returnValue;  
 if (mav.isReference()) {  
 String viewName = mav.getViewName();  
 mavContainer.setViewName(viewName);  
 if (viewName != null && isRedirectViewName(viewName)) {  
 mavContainer.setRedirectModelScenario(true);  
 }  
 }  
 else {  
 View view = mav.getView();  
 mavContainer.setView(view);  
 if (view instanceof SmartView) {  
 if (((SmartView) view).isRedirectView()) {  
 mavContainer.setRedirectModelScenario(true);  
 }  
 }  
 }  
 mavContainer.setStatus(mav.getStatus());  
 mavContainer.addAllAttributes(mav.getModel());  
}

* + 7) applyDefaultViewName（转换视图名称)

private void applyDefaultViewName(HttpServletRequest request, ModelAndView mv) throws Exception {  
 if (mv != null && !mv.hasView()) {  
 mv.setViewName(getDefaultViewName(request));  
 }  
}

* protected String getDefaultViewName(HttpServletRequest request) throws Exception {  
  return this.viewNameTranslator.getViewName(request);  
  }
* DefaultRequestToViewNameTransaltor.getDefaultViewName

public String getViewName(HttpServletRequest request) {  
 String lookupPath = this.urlPathHelper.getLookupPathForRequest(request);  
 return (this.prefix + transformPath(lookupPath) + this.suffix);  
}

- 8) HandlerExecutionChain#applyPostHandle（拦截器postHandle)

* void applyPostHandle(HttpServletRequest request, HttpServletResponse response, ModelAndView mv) throws Exception {  
  HandlerInterceptor[] interceptors = getInterceptors();  
  if (!ObjectUtils.isEmpty(interceptors)) {  
   for (int i = interceptors.length - 1; i >= 0; i--) {  
   HandlerInterceptor interceptor = interceptors[i];  
   interceptor.postHandle(request, response, this.handler, mv);  
   }  
  }  
  }
  + 9) processDispatchResult（处理ModelAndView请求结果)
* 如果返回的是纯数据（@ResponseBody)，mv就是null，该方法基本上是空方法。

private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,  
 HandlerExecutionChain mappedHandler, ModelAndView mv, Exception exception) throws Exception {  
  
 boolean errorView = false;  
  
 if (exception != null) {  
 if (exception instanceof ModelAndViewDefiningException) {  
 logger.debug("ModelAndViewDefiningException encountered", exception);  
 mv = ((ModelAndViewDefiningException) exception).getModelAndView();  
 }  
 else {  
 Object handler = (mappedHandler != null ? mappedHandler.getHandler() : null);

* // 处理异常  
   mv = processHandlerException(request, response, handler, exception);  
   errorView = (mv != null);  
   }  
  }
* // Did the handler return a view to render?
* // 如果handler处理结果中返回了view，那么需要对页面进行渲染  
  if (mv != null && !mv.wasCleared()) {  
   render(mv, request, response);  
   if (errorView) {  
   WebUtils.clearErrorRequestAttributes(request);  
   }  
  }  
  else {  
   if (logger.isDebugEnabled()) {  
   logger.debug("Null ModelAndView returned to DispatcherServlet with name '" + getServletName() +  
   "': assuming HandlerAdapter completed request handling");  
   }  
  }
* if (WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {  
   // Concurrent handling started during a forward  
   return;  
  }
* if (mappedHandler != null) {  
   mappedHandler.triggerAfterCompletion(request, response, null);  
  }  
  }
  + 9.1) processHandlerException（处理异常)
* protected ModelAndView processHandlerException(HttpServletRequest request, HttpServletResponse response,  
   Object handler, Exception ex) throws Exception {
* // Check registered HandlerExceptionResolvers...  
   ModelAndView exMv = null;  
   for (HandlerExceptionResolver handlerExceptionResolver : this.handlerExceptionResolvers) {
* // 使用handlerExceptionResolver来处理异常  
   exMv = handlerExceptionResolver.resolveException(request, response, handler, ex);  
   if (exMv != null) {  
   break;  
   }  
   }  
   if (exMv != null) {  
   if (exMv.isEmpty()) {  
   request.setAttribute(EXCEPTION\_ATTRIBUTE, ex);  
   return null;  
   }  
   // We might still need view name translation for a plain error model...  
   if (!exMv.hasView()) {  
   exMv.setViewName(getDefaultViewName(request));  
   }  
   if (logger.isDebugEnabled()) {  
   logger.debug("Handler execution resulted in exception - forwarding to resolved error view: " + exMv, ex);  
   }  
   WebUtils.exposeErrorRequestAttributes(request, ex, getServletName());  
   return exMv;  
   }
* throw ex;  
  }
  + 9.2) render（渲染视图)
* protected void render(ModelAndView mv, HttpServletRequest request, HttpServletResponse response) throws Exception {  
  // Determine locale for request and apply it to the response.  
  Locale locale = this.localeResolver.resolveLocale(request);  
  response.setLocale(locale);
* View view;  
  if (mv.isReference()) {  
   // We need to resolve the view name.  
   view = resolveViewName(mv.getViewName(), mv.getModelInternal(), locale, request);  
   if (view == null) {  
   throw new ServletException("Could not resolve view with name '" + mv.getViewName() +  
   "' in servlet with name '" + getServletName() + "'");  
   }  
  }  
  else {  
   // No need to lookup: the ModelAndView object contains the actual View object.  
   view = mv.getView();  
   if (view == null) {  
   throw new ServletException("ModelAndView [" + mv + "] neither contains a view name nor a " +  
   "View object in servlet with name '" + getServletName() + "'");  
   }  
  }
* // Delegate to the View object for rendering.  
  if (logger.isDebugEnabled()) {  
   logger.debug("Rendering view [" + view + "] in DispatcherServlet with name '" + getServletName() + "'");  
  }  
  try {  
   if (mv.getStatus() != null) {  
   response.setStatus(mv.getStatus().value());  
   }  
   view.render(mv.getModelInternal(), request, response);  
  }  
  catch (Exception ex) {  
   if (logger.isDebugEnabled()) {  
   logger.debug("Error rendering view [" + view + "] in DispatcherServlet with name '" +  
   getServletName() + "'", ex);  
   }  
   throw ex;  
  }  
  }
  + 9.2.1) resolveViewName（创建视图)
* protected View resolveViewName(String viewName, Map<String, Object> model, Locale locale,  
   HttpServletRequest request) throws Exception {
* for (ViewResolver viewResolver : this.viewResolvers) {
* // 使用viewResolver解析视图名称  
   View view = viewResolver.resolveViewName(viewName, locale);  
   if (view != null) {  
   return view;  
   }  
   }  
   return null;  
  }
  + 9.2.1.1) AbstractCachingViewResolver#resolveViewName

public View resolveViewName(String viewName, Locale locale) throws Exception {  
 if (!isCache()) {

* // 不存在缓存的情况下直接创建视图  
   return createView(viewName, locale);  
   }  
   else {
* // 直接从缓存中获取  
   Object cacheKey = getCacheKey(viewName, locale);  
   View view = this.viewAccessCache.get(cacheKey);  
   if (view == null) {  
   synchronized (this.viewCreationCache) {  
   view = this.viewCreationCache.get(cacheKey);  
   if (view == null) {  
   // Ask the subclass to create the View object.  
   view = createView(viewName, locale);  
   if (view == null && this.cacheUnresolved) {  
   view = UNRESOLVED*VIEW;*  
   *}*  
   *if (view != null) {*  
   *this.viewAccessCache.put(cacheKey, view);*  
   *this.viewCreationCache.put(cacheKey, view);*  
   *if (logger.isTraceEnabled()) {*  
   *logger.trace("Cached view [" + cacheKey + "]");*  
   *}*  
   *}*  
   *}*  
   *}*  
   *}*  
   *return (view != UNRESOLVED*VIEW ? view : null);  
   }  
  }
  + 9.2.1.1.1) UrlBasedViewResolver#createView
* protected View createView(String viewName, Locale locale) throws Exception {  
  // If this resolver is not supposed to handle the given view,  
  // return null to pass on to the next resolver in the chain.
* // 如果当前解析器不支持当前解析器，如viewName为空等情况  
  if (!canHandle(viewName, locale)) {  
   return null;  
  }
* // 处理redirect:xx的情况  
  // Check for special "redirect:" prefix.  
  if (viewName.startsWith(REDIRECT*URL*PREFIX)) {  
   String redirectUrl = viewName.substring(REDIRECT*URL*PREFIX.length());  
   RedirectView view = new RedirectView(redirectUrl, isRedirectContextRelative(), isRedirectHttp10Compatible());  
   view.setHosts(getRedirectHosts());  
   return applyLifecycleMethods(viewName, view);  
  }
* // 处理forward:xx的情况  
  // Check for special "forward:" prefix.  
  if (viewName.startsWith(FORWARD*URL*PREFIX)) {  
   String forwardUrl = viewName.substring(FORWARD*URL*PREFIX.length());  
   return new InternalResourceView(forwardUrl);  
  }  
  // Else fall back to superclass implementation: calling loadView.  
  return super.createView(viewName, locale);  
  }
* AbstractCachingViewResolver.createView
* protected View createView(String viewName, Locale locale) throws Exception {  
  return loadView(viewName, locale);  
  }
* UrlBasedViewResolver.loadView
* protected View loadView(String viewName, Locale locale) throws Exception {  
  AbstractUrlBasedView view = buildView(viewName);  
  View result = applyLifecycleMethods(viewName, view);  
  return (view.checkResource(locale) ? result : null);  
  }
* UrlBasedViewResolver.buildView
* protected AbstractUrlBasedView buildView(String viewName) throws Exception {  
  AbstractUrlBasedView view = (AbstractUrlBasedView) BeanUtils.instantiateClass(getViewClass());
* // 添加前缀和后缀  
  view.setUrl(getPrefix() + viewName + getSuffix());
* String contentType = getContentType();  
  if (contentType != null) {
* // 设置ContentType  
   view.setContentType(contentType);  
   }
* view.setRequestContextAttribute(getRequestContextAttribute());  
   view.setAttributesMap(getAttributesMap());
* Boolean exposePathVariables = getExposePathVariables();  
   if (exposePathVariables != null) {  
   view.setExposePathVariables(exposePathVariables);  
   }  
   Boolean exposeContextBeansAsAttributes = getExposeContextBeansAsAttributes();  
   if (exposeContextBeansAsAttributes != null) {  
   view.setExposeContextBeansAsAttributes(exposeContextBeansAsAttributes);  
   }  
   String[] exposedContextBeanNames = getExposedContextBeanNames();  
   if (exposedContextBeanNames != null) {  
   view.setExposedContextBeanNames(exposedContextBeanNames);  
   }
* return view;  
  }
  + 9.2.2) AbstractView.render（页面跳转)

public void render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response) throws Exception {  
 if (logger.isTraceEnabled()) {  
 logger.trace("Rendering view with name '" + this.beanName + "' with model " + model +  
 " and static attributes " + this.staticAttributes);  
 }  
 // 处理Model  
 Map<String, Object> mergedModel = createMergedOutputModel(model, request, response);  
 prepareResponse(request, response);

* // 处理页面跳转   
  renderMergedOutputModel(mergedModel, getRequestToExpose(request), response);  
  }
  + 9.2.2.1) AbstractView#createMergedOutputModel（处理Model)
* protected Map<String, Object> createMergedOutputModel(Map<String, ?> model, HttpServletRequest request,  
   HttpServletResponse response) {
* @SuppressWarnings("unchecked")  
   Map<String, Object> pathVars = (this.exposePathVariables ?  
   (Map<String, Object>) request.getAttribute(View.PATH\_VARIABLES) : null);
* // Consolidate static and dynamic model attributes.  
   int size = this.staticAttributes.size();  
   size += (model != null ? model.size() : 0);  
   size += (pathVars != null ? pathVars.size() : 0);
* Map<String, Object> mergedModel = new LinkedHashMap<String, Object>(size);  
   mergedModel.putAll(this.staticAttributes);  
   if (pathVars != null) {  
   mergedModel.putAll(pathVars);  
   }  
   if (model != null) {  
   mergedModel.putAll(model);  
   }
* // Expose RequestContext?  
   if (this.requestContextAttribute != null) {  
   mergedModel.put(this.requestContextAttribute, createRequestContext(request, response, mergedModel));  
   }
* return mergedModel;  
  }
  + 9.2.2.2) renderMergedOutputModel#renderMergedOutputModel（处理页面跳转)
* protected void renderMergedOutputModel(  
   Map<String, Object> model, HttpServletRequest request, HttpServletResponse response) throws Exception {  
   // 将model中的数据以属性方式设置到request中  
   // Expose the model object as request attributes.  
   exposeModelAsRequestAttributes(model, request);
* // Expose helpers as request attributes, if any.  
   exposeHelpers(request);
* // Determine the path for the request dispatcher.  
   String dispatcherPath = prepareForRendering(request, response);
* // Obtain a RequestDispatcher for the target resource (typically a JSP).  
   RequestDispatcher rd = getRequestDispatcher(request, dispatcherPath);  
   if (rd == null) {  
   throw new ServletException("Could not get RequestDispatcher for [" + getUrl() +  
   "]: Check that the corresponding file exists within your web application archive!");  
   }
* // If already included or response already committed, perform include, else forward.  
   if (useInclude(request, response)) {  
   response.setContentType(getContentType());  
   if (logger.isDebugEnabled()) {  
   logger.debug("Including resource [" + getUrl() + "] in InternalResourceView '" + getBeanName() + "'");  
   }  
   rd.include(request, response);  
   }
* else {  
   // Note: The forwarded resource is supposed to determine the content type itself.  
   if (logger.isDebugEnabled()) {  
   logger.debug("Forwarding to resource [" + getUrl() + "] in InternalResourceView '" + getBeanName() + "'");  
   }  
   rd.forward(request, response);  
   }  
  }
* 实例-MVC

@Controller  
public class IndexController {  
   
 @RequestMapping("/hello")  
 public ModelAndView index(ModelAndView modelAndView){  
 modelAndView.addObject("user",new RegisterDTO("admin"));  
 modelAndView.setViewName("hello");  
 return modelAndView;  
 }  
}

* hello.jsp
* ${requestScope.user.username}
* 执行路径：
* getHandler中handlerMappings为
* hm是0号，它调用getHandler。
* HandlerExecutionChain handler = hm.getHandler(request);
* 该方法中的getHandlerInternal方法这里用到的类是AbstractHandlerMethodMapping。
* 最终返回的是该handler，类型是HandlerMethod。
* getHandlerExecutionChain执行完后多了三个interceptor。
* getHandlerAdapter中的handlerAdapters是
* 遍历时先用到的是RequestMappingHandlerAdapter，调用的supports是AbstractHandlerMethodAdapter。匹配后，返回的adapter是RequestMappingHandlerAdapter类型的
* 然后调用该adapter的handle方法。
* handle方法最终会调用Controller的对应方法，然后获取ModelAndView类型返回值的ModelAndViewMethodReturnValueHandler，对modelAndView进行处理。
* 然后调用applyDefaultViewName方法.
* 在调用processDispatchResult之前，modelAndView是这样的。
* 然后去调用render方法。
* render
* ->InternalResourceViewResolver.resolveViewName
* ->UriBasedViewResolver.createView
* ->UriBasedViewResolver.loadView
* ->InternalResourceViewResolver.buildView
* 此时得到的view如下：
* 之后又调用了view的render方法，最终调用了requestDispatcher.forward方法结束整个过程。
* 实例-REST

@RequestMapping("/users/{name}")  
@ResponseBody  
public RegisterDTO findUserByName(@PathVariable("name") String name){  
 return new RegisterDTO(name);  
}

* 前面都一样，但是处理请求结果时使用的handler不一样，它使用的是RequestResponseBodyMethodProcessor，将返回值写入输出流。