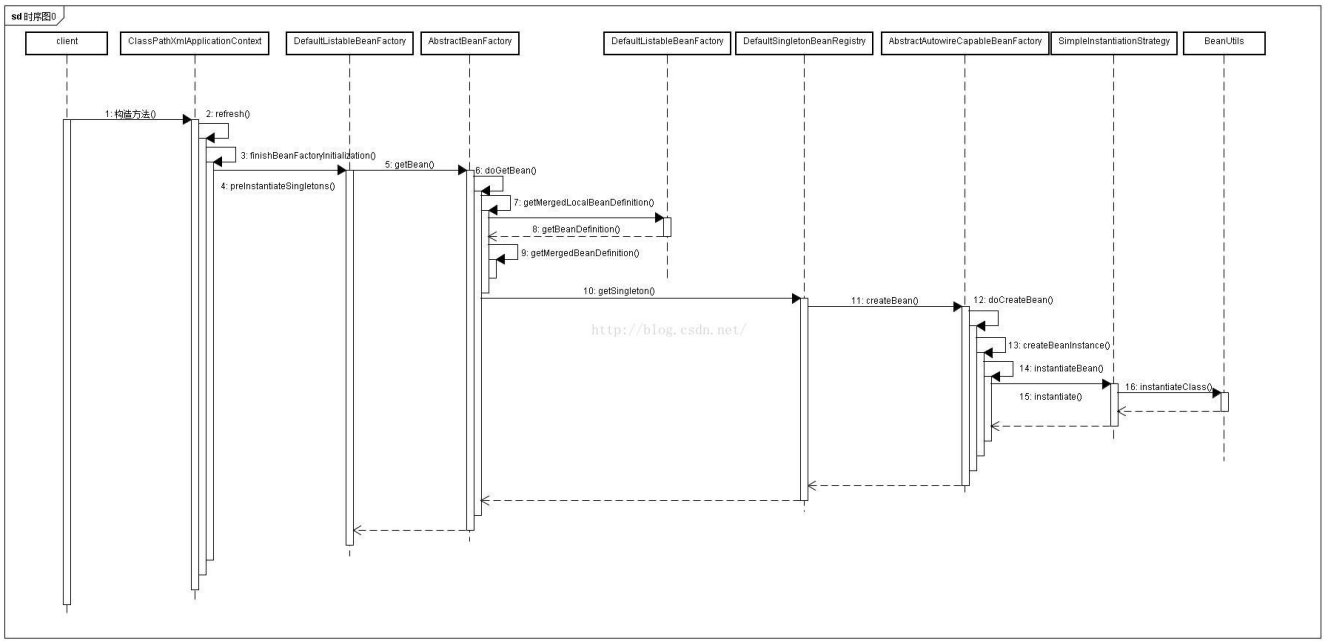
1 bean创建

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
Student student = (Student) context.getBean("student");
```

bean的创建过程如下：



2 refresh()

```
public void refresh() throws BeansException, IllegalStateException {
        synchronized (this.startupShutdownMonitor) {// 来个锁，不然 refresh() 还没结束，你又来个启动或销毁容器的操作，那不就乱套了嘛
            // 准备工作，记录下容器的启动时间、标记“已启动”状态、处理配置文件中的占位符
            prepareRefresh();
         // 这步比较关键，这步完成后，配置文件就会解析成一个个 Bean 定义，注册到 BeanFactory 中，
      // 当然，这里说的 Bean 还没有初始化，只是配置信息都提取出来了，
      // 注册也只是将这些信息都保存到了注册中心(说到底核心是一个 beanName-> beanDefinition 的 map)
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
// 设置 BeanFactory 的类加载器，添加几个 BeanPostProcessor，手动注册几个特殊的 bean
    prepareBeanFactory(beanFactory);
```

```
    try {
        // 【这里需要知道 BeanFactoryPostProcessor 这个知识点，Bean 如果实现了此接口，
     // 那么在容器初始化以后，Spring 会负责调用里面的 postProcessBeanFactory 方法。】

     // 这里是提供给子类的扩展点，到这里的时候，所有的 Bean 都加载、注册完成了，但是都还没有初始化
     // 具体的子类可以在这步的时候添加一些特殊的 BeanFactoryPostProcessor 的实现类或做点什么事
        //可以实现ClassPathXmlApplicationContext类并重写postProcessBeanFactory即可
        postProcessBeanFactory(beanFactory);

        // Invoke factory processors registered as beans in the context.
        invokeBeanFactoryPostProcessors(beanFactory);

        // Register bean processors that intercept bean creation.
        registerBeanPostProcessors(beanFactory);

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

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

        // 从方法名就可以知道，典型的模板方法(钩子方法)，
     // 具体的子类可以在这里初始化一些特殊的 Bean (在初始化 singleton beans 之前)
        onRefresh();

         // 注册事件监听器，监听器需要实现 ApplicationListener 接口
        registerListeners();

        // Instantiate all remaining (non-lazy-init) singletons.
        finishBeanFactoryInitialization(beanFactory);

        // Last step: publish corresponding event.
        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();
        }
    }
    }
```

validateRequiredProperties()

用于验证系统环境中是否有RequiredProperties参数值，如果前面设置了环境需要的属性值，但是系统环境中没有这里会报错

```
public class Test {
    public static void main(String[] args) {
        MyApplicationContext context=new MyApplicationContext();
    }
}
```

```
class MyApplicationContext extends AbstractApplicationContext{
    public MyApplicationContext() {
        getEnvironment().setRequiredProperties("11");
        refresh();
    }
    @Override
    protected void refreshBeanFactory() throws BeansException, IllegalStateException {
    }
    @Override
    protected void closeBeanFactory() {
    }
    @Override
    public ConfigurableListableBeanFactory getBeanFactory() throws
IllegalStateException {
        return null;
    }

 }
```

就会报

```
Exception in thread "main"
org.springframework.core.env.MissingRequiredPropertiesException:
    The following properties were declared as required but could not be resolved: [11]
    at
org.springframework.core.env.AbstractPropertyResolver.validateRequiredProperties(Abstra
ctPropertyResolver.java:123)
    at
org.springframework.core.env.AbstractEnvironment.validateRequiredProperties(AbstractEnv
ironment.java:515)
    at
org.springframework.context.support.AbstractApplicationContext.prepareRefresh(AbstractA
pplicationContext.java:589)
    at
org.springframework.context.support.AbstractApplicationContext.refresh(AbstractApplicat
ionContext.java:510)
    at com.pactera.test.MyApplicationContext.<init>(Test.java:22)
```

3 obtainFreshBeanFactory()

核心方法，加载xml文件并解析，注册bean

```
protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
   refreshBeanFactory(); //刷新bean工厂
   ConfigurableListableBeanFactory beanFactory = getBeanFactory(); //获取bean工厂
   if (logger.isDebugEnabled()) {
      logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
   }
   return beanFactory;
}
```

3.1 refreshBeanFactory

```
protected final void refreshBeanFactory() throws BeansException {
   //避免重复加载bean工厂
   if (hasBeanFactory()) {
      destroyBeans();
      closeBeanFactory();
   }
   try {
      DefaultListableBeanFactory beanFactory = createBeanFactory();
      beanFactory.setSerializationId(getId());
      customizeBeanFactory(beanFactory);
      loadBeanDefinitions(beanFactory);
      synchronized (this.beanFactoryMonitor) {
         this.beanFactory = beanFactory;
      }
   }
```

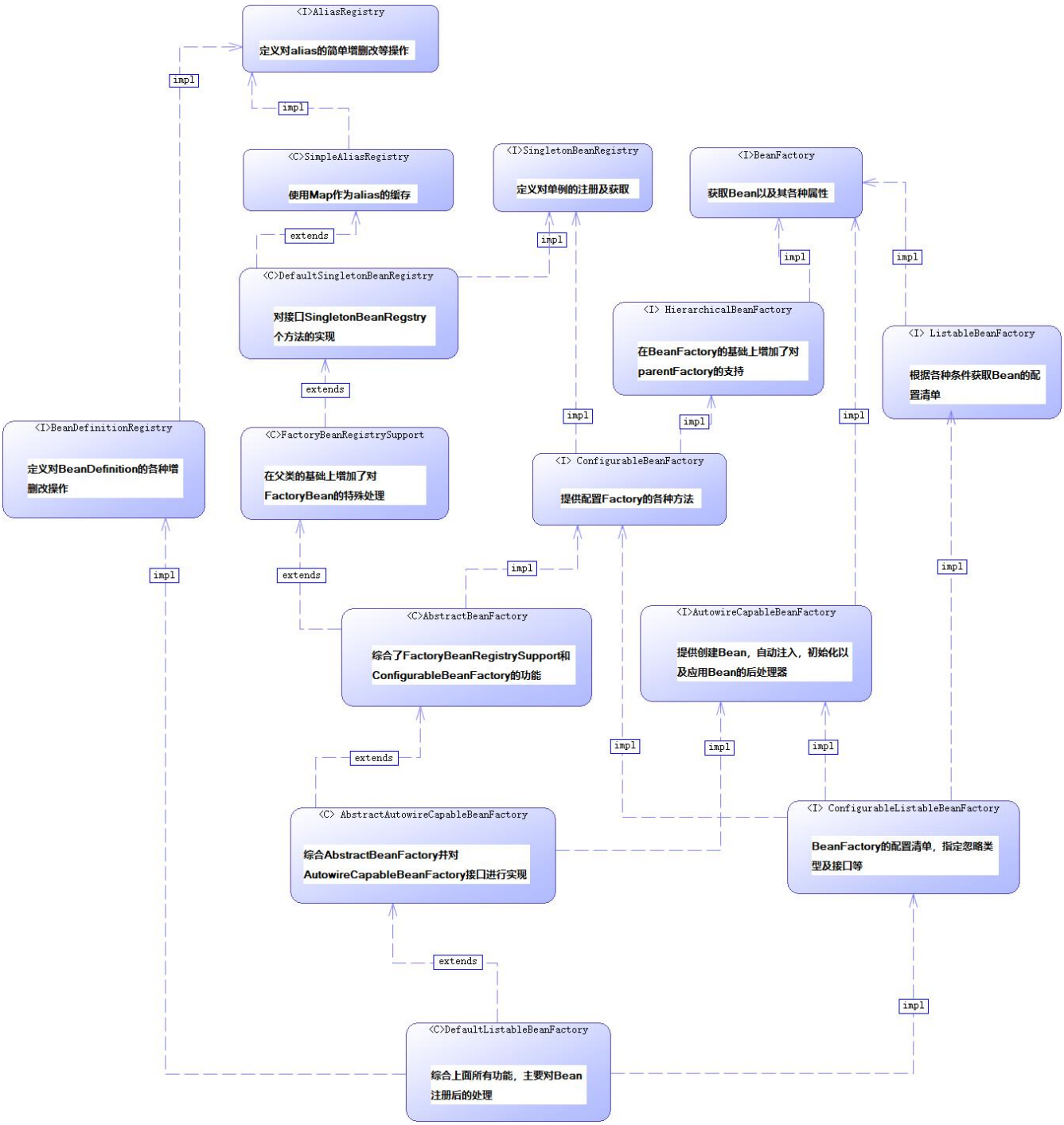
```
   catch (IOException ex) {
      throw new ApplicationContextException("I/O error parsing bean definition source
for " + getDisplayName(), ex);
   }
}
```

```
//为 context 创建一个内部的 bean factory。
//每个refresh方法都会尝试调用该方法
//默认的实现是创建一个DefaultListableBeanFactory对象，利用getInternalParentBeanFactory方法
//将 context 的 parent 的 bean factory ，作为该 context 内部 bean factory 的 parent bean
factory
 //该方法可以被子类覆盖，例如自定义DefaultListableBeanFactory的设定

protected DefaultListableBeanFactory createBeanFactory() {
        return new DefaultListableBeanFactory(getInternalParentBeanFactory());
    }
//创建beanfactory,设置父factory
```

```
/**
* 如果 parent context 实现了ConfigurableApplicationContext接口，
* 那么返回 parent context 内部的 bean factory
* 否则返回 parent context 自身
*/
protected BeanFactory getInternalParentBeanFactory() {
        return (getParent() instanceof ConfigurableApplicationContext) ?
                ((ConfigurableApplicationContext) getParent()).getBeanFactory() :
getParent();
    }
```

如果实现了ConfigurableApplicationContext接口则返回这个BeanFactory，否则返回自己的父BeanFactory

为什么要创建DefaultListableBeanFactory类型的beanFactory?



上图中beanFactory的功能 https://blog.csdn.net/Jatham/article/details/77895555

```
protected void customizeBeanFactory(DefaultListableBeanFactory beanFactory) {
        if (this.allowBeanDefinitionOverriding != null) {

beanFactory.setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);
        }
    //设置是否通过注册名称相同但是定义不同的bean来自动替代前面的bean
        if (this.allowCircularReferences != null) {
            beanFactory.setAllowCircularReferences(this.allowCircularReferences);
        }
    //是否允许bean之间循环引用
    }
```

BeanDefinition 的覆盖，就是在配置文件中定义 bean 时使用了相同的 id 或 name，默认情况下，allowBeanDefinitionOverriding 属性为 null，如果在同一配置文件中重复了，会抛错，但是如果不是同一配置文件中，会发生覆盖。

循环引用：A 依赖 B，而 B 依赖 A。或 A 依赖 B，B 依赖 C，而 C 依赖 A。

默认情况下，Spring 允许循环依赖，当然如果你在 A 的构造方法中依赖 B，在 B 的构造方法中依赖 A 是不行的

BeanDefinition：

```
//用于描述一个具体bean实例
public interface BeanDefinition extends AttributeAccessor, BeanMetadataElement {

    //scope值，单例
    String SCOPE_SINGLETON = ConfigurableBeanFactory.SCOPE_SINGLETON;

    //scope值，非单例
    String SCOPE_PROTOTYPE = ConfigurableBeanFactory.SCOPE_PROTOTYPE;


    //Bean角色：
    //用户
    int ROLE_APPLICATION = 0;
    //某些复杂的配置
    int ROLE_SUPPORT = 1;
    //完全内部使用
    int ROLE_INFRASTRUCTURE = 2;


    //返回此bean定义的父bean定义的名称，如果有的话 <bean parent="">
    String getParentName();
    void setParentName(String parentName);

    //获取bean对象className <bean class="">
    String getBeanClassName();
```

```
    void setBeanClassName(String beanClassName);

    //定义创建该Bean对象的工厂类  <bean factory-bean="">
    String getFactoryBeanName();
    void setFactoryBeanName(String factoryBeanName);


    //定义创建该Bean对象的工厂方法 <bean factory-method="">
    String getFactoryMethodName();
    void setFactoryMethodName(String factoryMethodName);


    //<bean scope="singleton/prototype">
    String getScope();
    void setScope(String scope);


    //懒加载 <bean lazy-init="true/false">
    boolean isLazyInit();
    void setLazyInit(boolean lazyInit);

    //依赖对象  <bean depends-on="">
    String[] getDependsOn();
    void setDependsOn(String[] dependsOn);


    //是否为被自动装配 <bean autowire-candidate="true/false">
    boolean isAutowireCandidate();
    void setAutowireCandidate(boolean autowireCandidate);

    //是否为主候选bean    使用注解：@Primary
    boolean isPrimary();
    void setPrimary(boolean primary);


    //返回此bean的构造函数参数值。
    ConstructorArgumentValues getConstructorArgumentValues();

    //获取普通属性集合
    MutablePropertyValues getPropertyValues();
    //是否为单例
    boolean isSingleton();
    //是否为原型
    boolean isPrototype();
    //是否为抽象类
    boolean isAbstract();

    //获取这个bean的应用
```

```
    int getRole();

    //返回对bean定义的可读描述。
    String getDescription();

    //返回该bean定义来自的资源的描述（用于在出现错误时显示上下文）
    String getResourceDescription();

    BeanDefinition getOriginatingBeanDefinition();
}
```

可以看到上面的很多属性和方法都很熟悉，例如类名、scope、属性、构造函数参数列表、依赖的bean、是否是单例类、是否是懒加载等，其实就是将Bean的定义信息存储到这个BeanDefinition相应的属性中，后面对Bean的操作就直接对BeanDefinition进行，例如拿到这个BeanDefinition后，可以根据里面的类名、构造函数、构造函数参数，使用反射进行对象创建。

BeanDefinition是一个接口，是一个抽象的定义，实际使用的是其实现类，如 ChildBeanDefinition、RootBeanDefinition、GenericBeanDefinition等。

https://www.jianshu.com/p/75ecc099cab1

3 .1.2 loadBeanDefinitions(beanFactory)

用于将BeanDefinition装入BeanFactory中

```
    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);
        //配置schemas或者dtd的资源解析器,加载xml验证文件
https://www.jianshu.com/p/4259ad820d2f
        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); //设置是否对xml进行验证，默认是
        loadBeanDefinitions(beanDefinitionReader);
}
```

3.1.2.1 AbstractBeanDefinitionReader

XmlBeanDefinitionReader(beanFactory)方法中最终会调用该方法对beanFactory做一些配置

```
/**
* 通过给定的 bean factory 创建一个新的 AbstractBeanDefinitionReader。
* 如果传入的 bean factory 不仅仅实现了 BeanDefinitionRegistry 接口，
* 还实现了ResourceLoader接口，那么 bean factory 还会被当做ResourceLoader资源加载器。
* 通常这种情况的 bean factory 它往往是ApplicationContext的实现。
* 如果给定一个简单的BeanDefinitionRegistry实现，那么默认的ResourceLoader采用
* PathMatchingResourcePatternResolver。
* 如果传入的 bean factory 还实现了EnvironmentCapable接口，
* 那么这个 bean factory 的environment会被这个reader所使用。
* 否者这个reader初始化使用默认StandardEnvironment。
*/
protected AbstractBeanDefinitionReader(BeanDefinitionRegistry registry) {
   Assert.notNull(registry, "BeanDefinitionRegistry must not be null");
   this.registry = registry;

   // Determine ResourceLoader to use.
   if (this.registry instanceof ResourceLoader) {
      this.resourceLoader = (ResourceLoader) this.registry;
   }
   else {
       // DefaultListableBeanFactory未实现ResourceLoader，所以走这个
      this.resourceLoader = new PathMatchingResourcePatternResolver();
   }

   // Inherit Environment if possible
   if (this.registry instanceof EnvironmentCapable) {
      this.environment = ((EnvironmentCapable) this.registry).getEnvironment();
   }
   else {
       // DefaultListableBeanFactory未实现EnvironmentCapable，所以走这个
        // StandardEnvironment初始化的时候就会将systemEnvironment和systemProperties加载进来
      this.environment = new StandardEnvironment();
   }
}
```

3.1.2.2 loadBeanDefinitions

```
protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws
BeansException, IOException {
    //进行扩展 如果有一天我们的applicationContext.xml不是在calsspath下了，我们只有Resource，那怎么办，直接传递进来即可，继承ClassPathXmlApplicationContext类重写getConfigResources方法，返回Resource即可。
        Resource[] configResources = getConfigResources();
        if (configResources != null) {
            reader.loadBeanDefinitions(configResources);
        }
    //xml文件
        String[] configLocations = getConfigLocations();
        if (configLocations != null) {
            reader.loadBeanDefinitions(configLocations);
        }
    }
```

```
//从指定的资源路径加载bean definition
//
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 {
          //将给定的xml解析成resource对象
         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;
   }
}
```

3.1.2.2.1 getResources(location)

将xml文件解析成resource对象

```
public Resource[] getResources(String locationPattern) throws IOException {
        Assert.notNull(locationPattern, "Location pattern must not be null");
        //是不是以classpath*:开始的
        if (locationPattern.startsWith(CLASSPATH_ALL_URL_PREFIX)) {
            //以classpath*:开始的
            if
(getPathMatcher().isPattern(locationPattern.substring(CLASSPATH_ALL_URL_PREFIX.length()
))) {
                // 路径中有? 或*号通配符 如classpath*:applicationContext-*.xml
                return findPathMatchingResources(locationPattern);
            }
            else {
                // 路径没有? 或*号通配符 如classpath:applicationContext.xml
                return
findAllClassPathResources(locationPattern.substring(CLASSPATH_ALL_URL_PREFIX.length()))
;
            }
        }
        else {
            //不以classpath*:开始的
            int prefixEnd = locationPattern.indexOf(":") + 1;
            if (getPathMatcher().isPattern(locationPattern.substring(prefixEnd))) {
```

```
                // 路径中有? 或*号通配符
                return findPathMatchingResources(locationPattern);
            }
            else {
                // 路径没有? 或*号通配符
                return new Resource[]
{getResourceLoader().getResource(locationPattern)};
            }
        }
    }
```

```
protected Resource[] findPathMatchingResources(String locationPattern) throws
IOException {
        //获取根目录路径这里是classpath*:/APP/
        String rootDirPath = determineRootDir(locationPattern);
        //匹配模式这里是applicationContext-*.xml
        String subPattern = locationPattern.substring(rootDirPath.length());
        //然后继续调用getResources，这时候传进去的是classpath*:/APP/，这里返回的是
classpath*:/APP/的URLResources
        Resource[] rootDirResources = getResources(rootDirPath);
        Set<Resource> result = new LinkedHashSet<Resource>(16);
        for (Resource rootDirResource : rootDirResources) {
            //解析，目前是原样返回
            rootDirResource = resolveRootDirResource(rootDirResource);
            //vfs(略)
            if
(rootDirResource.getURL().getProtocol().startsWith(ResourceUtils.URL_PROTOCOL_VFS)) {

result.addAll(VfsResourceMatchingDelegate.findMatchingResources(rootDirResource,
subPattern, getPathMatcher()));
            }
            //jar(略)
            else if (isJarResource(rootDirResource)) {
                result.addAll(doFindPathMatchingJarResources(rootDirResource,
subPattern));
            }
            else {
                //这里开始做匹配
                result.addAll(doFindPathMatchingFileResources(rootDirResource,
subPattern));
            }
        }
        if (logger.isDebugEnabled()) {
```

```
            logger.debug("Resolved location pattern [" + locationPattern + "] to
resources " + result);
        }
        return result.toArray(new Resource[result.size()]);

    }
```

如果路径中没有通配符：

```
protected Set<Resource> doFindAllClassPathResources(String path) throws IOException {
        Set<Resource> result = new LinkedHashSet<Resource>(16);
        //获取类加载器
        ClassLoader cl = getClassLoader();
        //根据类加载器加载资源，默认是classpath下
        Enumeration<URL> resourceUrls = (cl != null ? cl.getResources(path) :
ClassLoader.getSystemResources(path));
        //遍历
        while (resourceUrls.hasMoreElements()) {
            URL url = resourceUrls.nextElement();
            //转变为URLResource
            result.add(convertClassLoaderURL(url));
        }
        //这里是全jar查找，比如说设定是classpath*:applicationContext-*.xml,就会走这里的方法
        if ("".equals(path)) {
            // The above result is likely to be incomplete, i.e. only containing file
system references.
            // We need to have pointers to each of the jar files on the classpath as
well...
            addAllClassLoaderJarRoots(cl, result);
        }
        return result;

##  }
```

当路径中不是以classpath开始并且也没有通配符时：

```
public Resource getResource(String location) {
        Assert.notNull(location, "Location must not be null");
        //以“/”开始，返回的ClassPathContextResource是ClassPathResource的子类，处理了“/”开始的
情况
        if (location.startsWith("/")) {
            return getResourceByPath(location);
        }
```

```
        //以"classpath:"开始
        else if (location.startsWith(CLASSPATH_URL_PREFIX)) {
            //返回的是ClassPathResource
            return new
ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.length()), getClassLoader());
        }
        else {
            //都不是的情况下，是不是URL,以http://开始，返回UrlResource
            try {
                // Try to parse the location as a URL...
                URL url = new URL(location);
                return new UrlResource(url);
            }
            catch (MalformedURLException ex) {
                // No URL -> resolve as resource path.
                //按照path解析，返回的ClassPathContextResource是ClassPathResource的子类
                return getResourceByPath(location);
            }
        }

}
```

spring将给定路径中xml文件匹配并转换成了resource对象

https://blog.csdn.net/u014252478/article/details/84024475

3.1.2.2.2 loadBeanDefinitions()

根据resource加载bean

```
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());
        }
        //resourcesCurrentlyBeingLoaded是个ThreadLocal<Set<EncodedResource>>变量
        //得到变量中存放的值，注意每个线程得到的都是一个拷贝
        Set<EncodedResource> currentResources =
this.resourcesCurrentlyBeingLoaded.get();
        //第一次加载是null
        if (currentResources == null) {
            currentResources = new HashSet<EncodedResource>(4);
            this.resourcesCurrentlyBeingLoaded.set(currentResources);
        }
        //把资源放进set中，纪录已加载内容，不能加载重复内容
```

```
        if (!currentResources.add(encodedResource)) {
            throw new BeanDefinitionStoreException(
                    "Detected cyclic loading of " + encodedResource + " - check your
import definitions!");
        }
        try {
            //得到该Resource的InputStream
            InputStream inputStream = encodedResource.getResource().getInputStream();
            try {
                //用InputSource封装InputStream，此类是org.xml.sax包下的
                InputSource inputSource = new InputSource(inputStream);
                //如果encodedResource的编码格式不为空，设置InputSource编码格式
                if (encodedResource.getEncoding() != null) {
                    inputSource.setEncoding(encodedResource.getEncoding());
                }
                //do..开始真正的解析
                return doLoadBeanDefinitions(inputSource,
encodedResource.getResource());
            }
            finally {
                inputStream.close();
            }
        }
        catch (IOException ex) {
            throw new BeanDefinitionStoreException(
                    "IOException parsing XML document from " +
encodedResource.getResource(), ex);
        }
        finally {
            //加载完成后清空缓存
            currentResources.remove(encodedResource);
            if (currentResources.isEmpty()) {
                this.resourcesCurrentlyBeingLoaded.remove();
            }
        }

##  }
```

首先通过 `resourcesCurrentlyBeingLoaded.get()` 来获取已经加载过的资源，然后将 encodedResource 加入其中，如果 resourcesCurrentlyBeingLoaded 中已经存在该资源，则抛出 BeanDefinitionStoreException 异常。完成后从 encodedResource 获取封装的 Resource 资源并从 Resource 中获取相应的 InputStream ， 最后将 InputStream 封装为 InputSource 调用 `doLoadBeanDefinitions()`

3.1.2.2.2.1 doLoadBeanDefinitions(inputSource, encodedResource.getResource())

将xml文件解析成Document并根据document注册beanDefinition

```
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
            throws BeanDefinitionStoreException {
        try {
            Document doc = doLoadDocument(inputSource, resource);
            return registerBeanDefinitions(doc, resource);
        }
        catch (BeanDefinitionStoreException ex) {
            throw ex;
        }
        catch (SAXParseException ex) {
            throw new XmlBeanDefinitionStoreException(resource.getDescription(),
                    "Line " + ex.getLineNumber() + " in XML document from " + resource
+ " is invalid", ex);
        }
        catch (SAXException ex) {
            throw new XmlBeanDefinitionStoreException(resource.getDescription(),
                    "XML document from " + resource + " is invalid", ex);
        }
        catch (ParserConfigurationException ex) {
            throw new BeanDefinitionStoreException(resource.getDescription(),
                    "Parser configuration exception parsing XML from " + resource, ex);
        }
        catch (IOException ex) {
            throw new BeanDefinitionStoreException(resource.getDescription(),
                    "IOException parsing XML document from " + resource, ex);
        }
        catch (Throwable ex) {
            throw new BeanDefinitionStoreException(resource.getDescription(),
                    "Unexpected exception parsing XML document from " + resource, ex);
        }
    }
```

上面的方法中最主要的就是做了一下两件事：

1. 调用 `doLoadDocument()` 方法，根据 xml 文件获取 Document 实例。
2. 根据获取的 Document 实例注册 Bean 信息

第一件事：doLoadDocument(inputSource, resource);将xml转成document

```
protected Document doLoadDocument(InputSource inputSource, Resource resource) throws
Exception {
        return this.documentLoader.loadDocument(inputSource, getEntityResolver(),
this.errorHandler,
                getValidationModeForResource(resource), isNamespaceAware());

    }
```

这里有两个比较重要的方法getEntityResolver与getValidationModeForResource getEntityResolver返回的是EntityResolver。那什么是EntityResolver呢？作用是什么呢？

首先说下xml解析，一般情况下，SAX先去读取XML上的文档说明做一个验证，一般情况都是从网上下载，但是如果没有网会报错，这就是为什么在没有网的时候我们自己配置xsd或dtd文件。

EntityResolver做了一个本地化，就是先在本地找约束文件，找不到再在网上去找。

先说一下大体验证流程。

以xsd文件为例，首先为什么要找xsd文件，要知道我们要解析XML要按照一定模式去解析，xsd就是模式，他告诉你有哪些标签可以用，标签的属性是什么，子标签是什么等等。。这样才可以去解析。但是如果你自己写的标签不属于xsd定义的，则验证失败，造成无法解析的后果。

spring所有的xsd约束文件都在META-INF/spring.schemas下，加载DTD类型的beansDtdResolver的resolveEntity是直接截取systemId最后的xx.dtd然后去当前路径下寻找

getValidationModeForResource是获取验证模式，如果只有XSD那也就不分什么模式了，但是XSD之前还有个DTD约束。我们要保证约束的匹配性，我不能用XSD约束去验证DTD标签是吧

```
protected EntityResolver getEntityResolver() {
        if (this.entityResolver == null) {
            ResourceLoader resourceLoader = getResourceLoader();
            if (resourceLoader != null) {
                this.entityResolver = new ResourceEntityResolver(resourceLoader);
            }
            else {
                this.entityResolver = new
DelegatingEntityResolver(getBeanClassLoader());
            }
        }
        return this.entityResolver;

    }
```

如果 ResourceLoader 不为 null，则根据指定的 ResourceLoader 创建一个 ResourceEntityResolver。如果 ResourceLoader 为null，则创建 一个 DelegatingEntityResolver，该 Resolver 委托给默认的 BeansDtdResolver 和 PluggableSchemaResolver 。

ResourceEntityResolver：继承自 EntityResolver ，通过 ResourceLoader 来解析实体的引用。
DelegatingEntityResolver：EntityResolver 的实现，分别代理了 dtd 的 BeansDtdResolver 和 xml schemas 的 PluggableSchemaResolver。 BeansDtdResolver： spring bean dtd 解析器。EntityResolver 的实现，用来从 classpath 或者 jar 文件加载 dtd。

PluggableSchemaResolver：使用一系列 Map 文件将 schema url 解析到本地 classpath 资源

EntityResolver：如果 SAX 应用程序需要实现自定义处理外部实体，则必须实现此接口并使用 `setEntityResolver()` 向 SAX 驱动器注册一个实例。

getValidationModeForResource方法来获取对应资源的验证模式

```
protected int getValidationModeForResource(Resource resource) {
        //得到指定的验证模式，当然这里没有指定默认是VALIDATION_AUTO
        int validationModeToUse = getValidationMode();
        if (validationModeToUse != VALIDATION_AUTO) {
            //如果指定了就用指定的
            return validationModeToUse;
        }
        //检查验证模式，根据xml文档头进行检验
        int detectedMode = detectValidationMode(resource);
        if (detectedMode != VALIDATION_AUTO) {
            //如果不等于VALIDATION_AUTO，就用检验后的
            return detectedMode;
        }
        // Hmm, we didn't get a clear indication... Let's assume XSD,
        // since apparently no DTD declaration has been found up until
        // detection stopped (before finding the document's root tag).
        //直接用VALIDATION_XSD
        return VALIDATION_XSD;

##  }
```

自动检测验证模式的功能是在函数detectValidationMode方法中实现的，在detectValidationMode函数中又将自动检测验证模式的工作委托给了专门处理类XmlValidationModeDetector的detectValidationMode方法

```
protected int detectValidationMode(Resource resource) {
   if (resource.isOpen()) {
      throw new BeanDefinitionStoreException(
            "Passed-in Resource [" + resource + "] contains an open stream: " +
            "cannot determine validation mode automatically. Either pass in a Resource
" +
            "that is able to create fresh streams, or explicitly specify the
validationMode " +
            "on your XmlBeanDefinitionReader instance.");
   }

   InputStream inputStream;
   try {
      inputStream = resource.getInputStream();
   }
   catch (IOException ex) {
      throw new BeanDefinitionStoreException(
            "Unable to determine validation mode for [" + resource + "]: cannot open
InputStream. " +
```

```
            "Did you attempt to load directly from a SAX InputSource without specifying
the " +
            "validationMode on your XmlBeanDefinitionReader instance?", ex);
   }

   try {
      return this.validationModeDetector.detectValidationMode(inputStream);
   }
   catch (IOException ex) {
      throw new BeanDefinitionStoreException("Unable to determine validation mode for
[" +
            resource + "]: an error occurred whilst reading from the InputStream.",
ex);
   }
}
```

通过是否有DOCTYPE来判断是DTD还是XSD校验

```
public int detectValidationMode(InputStream inputStream) throws IOException {
   // Peek into the file to look for DOCTYPE.
   BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
   try {
      boolean isDtdValidated = false;
      String content;
      while ((content = reader.readLine()) != null) {
         content = consumeCommentTokens(content);
         if (this.inComment || !StringUtils.hasText(content)) {
            continue;
         }
         if (hasDoctype(content)) {
            isDtdValidated = true;
            break;
         }
         if (hasOpeningTag(content)) {
            // End of meaningful data...
            break;
         }
      }
      return (isDtdValidated ? VALIDATION_DTD : VALIDATION_XSD);
   }
   catch (CharConversionException ex) {
      // Choked on some character encoding...
      // Leave the decision up to the caller.
      return VALIDATION_AUTO;
   }
   finally {
```

```
        reader.close();
    }
}
```

DTD格式文档

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">
```

XSD格式文档

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
```

对于文档的读取委托给了DocumentLoader去执行，这里的DocumentLoader是个接口，而真正调用的是DefaultDocumentLoader，首选创建DocumentBuildFactory，再通过DocumentBuilderFactory创建DocumentBuilder，进而解析InputSource来返回Document对象

```
public Document loadDocument(InputSource inputSource, EntityResolver entityResolver,
            ErrorHandler errorHandler, int validationMode, boolean namespaceAware)
throws Exception {
    DocumentBuilderFactory factory = createDocumentBuilderFactory(validationMode,
namespaceAware);
        if (logger.isDebugEnabled()) {
            logger.debug("Using JAXP provider [" + factory.getClass().getName() + "]");
        }
        DocumentBuilder builder = createDocumentBuilder(factory, entityResolver,
errorHandler);
        return builder.parse(inputSource);
}
```

第二件事：registerBeanDefinitions(Document doc, Resource resource)注册bean

```
public int registerBeanDefinitions(Document doc, Resource resource) throws
BeanDefinitionStoreException {
        //实例化BeanDefinitionDocumentReader的实现类DefaultBeanDefinitionDocumentReader
        BeanDefinitionDocumentReader documentReader =
createBeanDefinitionDocumentReader();
        //记录注册之前的数值
        int countBefore = getRegistry().getBeanDefinitionCount();
        //执行注册
        documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
        //注册的数值
        return getRegistry().getBeanDefinitionCount() - countBefore;

    }
```

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

解析工作分为三步：1、解析默认标签；2、解析默认标签下的自定义标签；3、注册解析后的 BeanDefinition。经过前面两个步骤的解析，这时的 BeanDefinition 已经可以满足后续的使用要求了，那么接下来的工作就是将这些 BeanDefinition 进行注册，也就是完成第三步。

3.1.2.2.2.1.1 doRegisterBeanDefinitions()

配置中注册每一个bean，如果有嵌套的beans，那么递归执行这个方法。

```
    protected void doRegisterBeanDefinitions(Element root) {
        // 这里为什么要定义一个 parent? 看到后面就知道了，是递归问题，
   // 因为 <beans /> 内部是可以定义 <beans /> 的，所以这个方法的 root 其实不一定就是 xml 的根节
点，也可以是嵌套在里面的 <beans /> 节点
        BeanDefinitionParserDelegate parent = this.delegate;
        this.delegate = createDelegate(getReaderContext(), root, parent);
        //判断root的命名空间是否是默认的 http://www.springframework.org/schema/beans
    if (this.delegate.isDefaultNamespace(root)) { //处理profile属性
        //处理profile属性
        String profileSpec = root.getAttribute(PROFILE_ATTRIBUTE);
        if (StringUtils.hasText(profileSpec)) {//判断是否存在
            //profile可以指定多个
            String[] specifiedProfiles = StringUtils.tokenizeToStringArray(
                    profileSpec,
BeanDefinitionParserDelegate.MULTI_VALUE_ATTRIBUTE_DELIMITERS);
            //去系统环境中去找是否匹配设置，不存跳过解析，存在就解析
```

```
            if
(!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles)) {
                return;
            }
        }
    }

       //扩展，解析前处理 没用到
    preProcessXml(root);

    parseBeanDefinitions(root, this.delegate);
        //扩展，解析后处理 没用到
    postProcessXml(root);

    this.delegate = parent;
}
```

profile指定不同的环境加载那些bean

```
<beans xmlns="http://www.springframework.org/schema/beans"

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

   xmlns:jdbc="http://www.springframework.org/schema/jdbc"

    xmlns:jee="http://www.springframework.org/schema/jee"

    xsi:schemaLocation="...">

   <beans profile="development">

       <jdbc:embedded-database id="dataSource">

           <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>

           <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>

       </jdbc:embedded-database>

    </beans>

    <beans profile="production">

        <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>

    </beans>
```

```
</beans>
```

接下来的问题是，怎么使用特定的 profile 呢？Spring 在启动的过程中，会去寻找 “spring.profiles.active” 的属性值，根据这个属性值来的。那怎么配置这个值呢？

Spring 会在这几个地方寻找 spring.profiles.active 的属性值：操作系统环境变量、JVM 系统变量、web.xml 中定义的参数、JNDI。

最简单的方式莫过于在程序启动的时候指定：-Dspring.profiles.active="profile1,profile2"

```
public boolean acceptsProfiles(String... profiles) {
        Assert.notEmpty(profiles, "Must specify at least one profile");
        for (String profile : profiles) {
            if (StringUtils.hasLength(profile) && profile.charAt(0) == '!') {
                if (!isProfileActive(profile.substring(1))) {
                    return true;
                }
            }
            //如果系统包含profile属性返回true
            else if (isProfileActive(profile)) {
                return true;
            }
        }
        return false;

}
```

```
protected boolean isProfileActive(String profile) {
        validateProfile(profile);
        //获取系统环境下配置的profile属性
        Set<String> currentActiveProfiles = doGetActiveProfiles();
        return (currentActiveProfiles.contains(profile) ||
                (currentActiveProfiles.isEmpty() &&
doGetDefaultProfiles().contains(profile)));

}
```

```
protected Set<String> doGetActiveProfiles() {
        synchronized (this.activeProfiles) {
            //只获取一次即可，系统配置的属性profile加载一次即可
            if (this.activeProfiles.isEmpty()) {
                //从系统中获取ACTIVE_PROFILES_PROPERTY_NAME==spring.profiles.active
```

```
                String profiles = getProperty(ACTIVE_PROFILES_PROPERTY_NAME);
                if (StringUtils.hasText(profiles)) {
                    setActiveProfiles(StringUtils.commaDelimitedListToStringArray(
                            StringUtils.trimAllWhitespace(profiles)));
                }
            }
            return this.activeProfiles;
        }

}
```

3.1.2.2.2.1.1.1parseBeanDefinitions(root, this.delegate)

解析xml

```
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate
delegate) {
    //
        if (delegate.isDefaultNamespace(root)) {//获取root命名空间是否是
http://www.springframework.org/schema/beans
            NodeList nl = root.getChildNodes();//获取所有子节点
            for (int i = 0; i < nl.getLength(); i++) {
                Node node = nl.item(i);
                if (node instanceof Element) {//获取根元素下的子Node，注意，Node不一定是子标
签，可能是回车，可能是注释
                    Element ele = (Element) node;
                    if (delegate.isDefaultNamespace(ele)) {
                        parseDefaultElement(ele, delegate);//解析节点
                    }
                    else {
                        delegate.parseCustomElement(ele);
                    }
                }
            }
        }
        else {
            delegate.parseCustomElement(root);
        }
    }
```

parseDefaultElement(ele, delegate) 代表解析的节点是 `<import />`、`<alias />`、`<bean />`、`<beans />`

而对于其他的标签，将进入到 delegate.parseCustomElement(element) 这个分支。如我们经常会使用到的 `<mvc />`、`<task />`、`<context />`、`<aop />`等。

这些属于扩展，如果需要使用上面这些 ”非 default“ 标签，那么上面的 xml 头部的地方也要引入相应的 namespace 和 .xsd 文件的路径，如下所示。同时代码中需要提供相应的 parser 来解析，如 MvcNamespaceHandler、TaskNamespaceHandler、ContextNamespaceHandler、AopNamespaceHandler 等。

假如想分析 `<context:property-placeholder location="classpath:xx.properties" />` 的实现原理，就应该到 ContextNamespaceHandler 中找答案。

```
private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {
        if (delegate.nodeNameEquals(ele, IMPORT_ELEMENT)) {// 处理 <import /> 标签
            importBeanDefinitionResource(ele);
        }
        else if (delegate.nodeNameEquals(ele, ALIAS_ELEMENT)) {// 处理 <alias /> 标签定义
            processAliasRegistration(ele);
        }
        else if (delegate.nodeNameEquals(ele, BEAN_ELEMENT)) {// 处理 <bean /> 标签定义
            processBeanDefinition(ele, delegate);
        }
        else if (delegate.nodeNameEquals(ele, NESTED_BEANS_ELEMENT)) {// 如果碰到的是嵌套
的 <beans /> 标签，需要递归
            // recurse
            doRegisterBeanDefinitions(ele);
        }
    }
```

上面这个方法中最重要的就是对bean标签的解析

processBeanDefinition(ele, delegate)

常用的是bean标签，使用processBeanDefinition(ele, delegate);处理

```
protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate
delegate) {
    //将 <bean /> 节点中的例如class、name、id、alias之类的属性信息提取出来，然后封装到一个
BeanDefinitionHolder 中
        BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
        if (bdHolder != null) {
            //bean标签下还有自定义标签再次解析，一般没用过
            /**
            <bean id="animal" class="test.constructor.Animal">
                <myTag:my   value="p1"/>
            </bean>
            **/
            bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
            try {
                //向SpringIOC容器注册解析得到的bean定义
                //这是bean定义向IOC容器注册的入口，实际上是放到一个map里面
```

```
                BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
getReaderContext().getRegistry());
            }
            catch (BeanDefinitionStoreException ex) {
                getReaderContext().error("Failed to register bean definition with name
'" +
                        bdHolder.getBeanName() + "'", ele, ex);
            }
            // Send registration event.
            getReaderContext().fireComponentRegistered(new
BeanComponentDefinition(bdHolder));
        }
    }
```

首先委托 `BeanDefinitionDelegate` 类的 `parseBeanDefinitionElement` 方法进行元素解析，返回 `BeanDefinitionHolder` 类型的实例 `bdHolder`，经过这个方法后， `bdHolder` 实例已经包含我们的配置文件中配置的各个属性了，例如 `class、name、id、alias` 之类的属性。

当返回的 `bdHolder` 不为空的情况下若存在默认标签的子节点下再有自定义属性，还需要再次对自定义标签进行解析。

```
<bean id="test" class="test.MyClass">
    <mybean:user username="aaa"/>
</bean>
```

解析完成之后，需要对解析后的 `bdHolder` 进行注册，同样，注册操作委托给了 `BeanDefinitionReaderUtils` 的 `registerBeanDefinition` 方法。

最后发出响应事件，通知想关的监听器，这个 `bean` 已经加载完成了

parseBeanDefinitionElement

代理解析Bean的流程，如果解析的过程发生错误，返回null

```
public BeanDefinitionHolder parseBeanDefinitionElement(Element ele, BeanDefinition
containingBean) {
        String id = ele.getAttribute(ID_ATTRIBUTE);//获取id
        String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);//获取名称属性
List<String> aliases = new ArrayList<String>();
    // 将 name 属性的定义按照 ”逗号、分号、空格“ 切分，形成一个别名列表数组
    if (StringUtils.hasLength(nameAttr)) {
        String[] nameArr = StringUtils.tokenizeToStringArray(nameAttr,
MULTI_VALUE_ATTRIBUTE_DELIMITERS);
```

```
            aliases.addAll(Arrays.asList(nameArr));
        }

        String beanName = id;
        // 如果没有指定id，那么用别名列表的第一个名字作为beanName
        if (!StringUtils.hasText(beanName) && !aliases.isEmpty()) {
            beanName = aliases.remove(0);
            if (logger.isDebugEnabled()) {
                logger.debug("No XML 'id' specified - using '" + beanName +
                        "' as bean name and " + aliases + " as aliases");
            }
        }

        //检查beanName与别名是否已经被别的bean引用过，即不能定义两个bean的id相同或别名相同
        if (containingBean == null) {
            checkNameUniqueness(beanName, aliases, ele);
        }
//解析bean属性，并封装为BeanDefinition 根据 <bean ...>...</bean> 中的配置创建 BeanDefinition，
然后把配置中的信息都设置到实例中，
        // 真正解析，内部处理<bean />的所有相关的配
置"parent", "class", "abstract", "scope", "singleton", "lazy-init",等，然后存储在
BeanDifinition中，GenericBeanDefinition。
        AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele, beanName,
containingBean);
        // 到这里，整个 <bean /> 标签就算解析结束了，一个 BeanDefinition 就形成了。
        if (beanDefinition != null) {
            if (!StringUtils.hasText(beanName)) {// 如果都没有设置 id 和 name，那么此时的
beanName 就会为 null //如果beanName为空，Spring自动生成beanName
                try {
                    if (containingBean != null) {
                        beanName = BeanDefinitionReaderUtils.generateBeanName(
                                beanDefinition, this.readerContext.getRegistry(), true);
                    }
                    else {
                        //此方法containingBean为null
                        //beanName为class+#+数字，比如注册两个类型相同的bean，后面数字是0和1
                        beanName = this.readerContext.generateBeanName(beanDefinition);
                        // Register an alias for the plain bean class name, if still
possible,
                        // if the generator returned the class name plus a suffix.
                        // This is expected for Spring 1.2/2.0 backwards compatibility.
                        //beanClassName设置为别名
                        String beanClassName = beanDefinition.getBeanClassName();
                        if (beanClassName != null &&
                                beanName.startsWith(beanClassName) && beanName.length() >
beanClassName.length() &&

!this.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {
```

```
                        aliases.add(beanClassName);
                    }
                }
                if (logger.isDebugEnabled()) {
                    logger.debug("Neither XML 'id' nor 'name' specified - " +
                            "using generated bean name [" + beanName + "]");
                }
            }
            catch (Exception ex) {
                error(ex.getMessage(), ele);
                return null;
            }
        }
        String[] aliasesArray = StringUtils.toStringArray(aliases);
        //根据beanDefinition, beanName, aliasesArray创建BeanDefinitionHolder实例
        return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);
    }

    return null;
}
```

在当前层完成的主要工作包括如下内容：

（1）提取元素中的id以及name属性 （2）进一步解析其他所有属性并统一封装至GenericBeanDefinition类型的实例中 （3）如果检测到bean没有指定beanName，那么使用默认规则为此Bean生成beanName

（4）将获取到的信息封装到BeanDefinitionHolder的实例中

parseBeanDefinitionElement(ele, beanName, containingBean)

不关注名称和别名，只解析bean definition自身

```
public AbstractBeanDefinition parseBeanDefinitionElement(
            Element ele, String beanName, BeanDefinition containingBean) {
    this.parseState.push(new BeanEntry(beanName));

    String className = null;
    //解析class属性
    if (ele.hasAttribute(CLASS_ATTRIBUTE)) {
        className = ele.getAttribute(CLASS_ATTRIBUTE).trim();
    }

    try {
        String parent = null;
        if (ele.hasAttribute(PARENT_ATTRIBUTE)) {
            parent = ele.getAttribute(PARENT_ATTRIBUTE);
        }
```

```
        //根据className, parent创建GenericBeanDefinition对象用来存储解析出的各种属性
        AbstractBeanDefinition bd = createBeanDefinition(className, parent);

        //解析bean上的属性
        parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);
        //Description
        bd.setDescription(DomUtils.getChildElementValueByTagName(ele,
DESCRIPTION_ELEMENT));
        // 解析 <meta />
        parseMetaElements(ele, bd);
        // 解析 <lookup-method />
        parseLookupOverrideSubElements(ele, bd.getMethodOverrides());
        // 解析 <replaced-method />
        parseReplacedMethodSubElements(ele, bd.getMethodOverrides());

        // 解析 <constructor-arg />解析构造方法
        parseConstructorArgElements(ele, bd);
        // 解析 <property />
        parsePropertyElements(ele, bd);
        // 解析 <qualifier />
        parseQualifierElements(ele, bd);

        bd.setResource(this.readerContext.getResource());
        bd.setSource(extractSource(ele));

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

    return null;
}
```

GenericBeanDefinition实现BeanDefinition，BeanDefinition就是配置文件信息在容器中的表示

parseBeanDefinitionAttributes

方法是对element所有元素属性进行解析

```
    public AbstractBeanDefinition parseBeanDefinitionAttributes(Element ele, String
beanName,
            BeanDefinition containingBean, AbstractBeanDefinition bd) {
        if (ele.hasAttribute(SINGLETON_ATTRIBUTE)) {
        error("Old 1.x 'singleton' attribute in use - upgrade to 'scope' declaration",
ele);
    }
        //解析scope
    else if (ele.hasAttribute(SCOPE_ATTRIBUTE)) {
        bd.setScope(ele.getAttribute(SCOPE_ATTRIBUTE));
    }
        //如果父containingBean存在，就用其定义的scope
    else if (containingBean != null) {
        // Take default from containing bean in case of an inner bean definition.
        bd.setScope(containingBean.getScope());
    }
    //abstract
    if (ele.hasAttribute(ABSTRACT_ATTRIBUTE)) {
        bd.setAbstract(TRUE_VALUE.equals(ele.getAttribute(ABSTRACT_ATTRIBUTE)));
    }

        //lazy-init
    String lazyInit = ele.getAttribute(LAZY_INIT_ATTRIBUTE);
    if (DEFAULT_VALUE.equals(lazyInit)) {
        lazyInit = this.defaults.getLazyInit();
    }
    bd.setLazyInit(TRUE_VALUE.equals(lazyInit));
    //autowire
    String autowire = ele.getAttribute(AUTOWIRE_ATTRIBUTE);
    bd.setAutowireMode(getAutowireMode(autowire));

        //dependency-check
    String dependencyCheck = ele.getAttribute(DEPENDENCY_CHECK_ATTRIBUTE);
    bd.setDependencyCheck(getDependencyCheck(dependencyCheck));
        //depends-on
    if (ele.hasAttribute(DEPENDS_ON_ATTRIBUTE)) {
        String dependsOn = ele.getAttribute(DEPENDS_ON_ATTRIBUTE);
        bd.setDependsOn(StringUtils.tokenizeToStringArray(dependsOn,
MULTI_VALUE_ATTRIBUTE_DELIMITERS));
    }
//autowire-candidate
    String autowireCandidate = ele.getAttribute(AUTOWIRE_CANDIDATE_ATTRIBUTE);
    if ("".equals(autowireCandidate) || DEFAULT_VALUE.equals(autowireCandidate)) {
        String candidatePattern = this.defaults.getAutowireCandidates();
```

```
        if (candidatePattern != null) {
            String[] patterns =
StringUtils.commaDelimitedListToStringArray(candidatePattern);
            bd.setAutowireCandidate(PatternMatchUtils.simpleMatch(patterns, beanName));
        }
    }
    else {
        bd.setAutowireCandidate(TRUE_VALUE.equals(autowireCandidate));
    }
//primary
    if (ele.hasAttribute(PRIMARY_ATTRIBUTE)) {
        bd.setPrimary(TRUE_VALUE.equals(ele.getAttribute(PRIMARY_ATTRIBUTE)));
    }
//init-method
    if (ele.hasAttribute(INIT_METHOD_ATTRIBUTE)) {
        String initMethodName = ele.getAttribute(INIT_METHOD_ATTRIBUTE);
        if (!"".equals(initMethodName)) {
            bd.setInitMethodName(initMethodName);
        }
    }
    else {
        if (this.defaults.getInitMethod() != null) {
            bd.setInitMethodName(this.defaults.getInitMethod());
            bd.setEnforceInitMethod(false);
        }
    }
//解析destroy-method属性
    if (ele.hasAttribute(DESTROY_METHOD_ATTRIBUTE)) {
        String destroyMethodName = ele.getAttribute(DESTROY_METHOD_ATTRIBUTE);
        bd.setDestroyMethodName(destroyMethodName);
    }
    else {
        if (this.defaults.getDestroyMethod() != null) {
            bd.setDestroyMethodName(this.defaults.getDestroyMethod());
            bd.setEnforceDestroyMethod(false);
        }
    }
//解析factory-method属性
    if (ele.hasAttribute(FACTORY_METHOD_ATTRIBUTE)) {
        bd.setFactoryMethodName(ele.getAttribute(FACTORY_METHOD_ATTRIBUTE));
    }
        //解析factory-bean属性
    if (ele.hasAttribute(FACTORY_BEAN_ATTRIBUTE)) {
        bd.setFactoryBeanName(ele.getAttribute(FACTORY_BEAN_ATTRIBUTE));
    }

    return bd;
}
```

parseConstructorArgElements(ele, bd)

解析构造方法

```
<bean id="student" class="org.springframework.StudentBean">
    <constructor-arg index="0">
        <value>12</value>
    </constructor-arg>
    <property name="name" value="zhangsan"/>
</bean>
```

遍历所有的子元素获取constructor-arg进行解析

```
public void parseConstructorArgElements(Element beanEle, BeanDefinition bd) {
   NodeList nl = beanEle.getChildNodes();
   for (int i = 0; i < nl.getLength(); i++) {
      Node node = nl.item(i);
      if (isCandidateElement(node) && nodeNameEquals(node, CONSTRUCTOR_ARG_ELEMENT)) {
         parseConstructorArgElement((Element) node, bd);
      }
   }
}
```

//开始解析

```
public void parseConstructorArgElement(Element ele, BeanDefinition bd) {
   String indexAttr = ele.getAttribute(INDEX_ATTRIBUTE);//解析index
   String typeAttr = ele.getAttribute(TYPE_ATTRIBUTE);
   String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);
   if (StringUtils.hasLength(indexAttr)) { //存在index属性
      try {
         int index = Integer.parseInt(indexAttr);
         if (index < 0) {
            error("'index' cannot be lower than 0", ele);
         }
         else {
            try {
               this.parseState.push(new ConstructorArgumentEntry(index));
               Object value = parsePropertyValue(ele, bd, null);
               ConstructorArgumentValues.ValueHolder valueHolder = new
ConstructorArgumentValues.ValueHolder(value);
               if (StringUtils.hasLength(typeAttr)) {
```

```
                valueHolder.setType(typeAttr);
             }
             if (StringUtils.hasLength(nameAttr)) {
                valueHolder.setName(nameAttr);
             }
             valueHolder.setSource(extractSource(ele));
             if (bd.getConstructorArgumentValues().hasIndexedArgumentValue(index))
{//不允许重复指定相同的参数
                error("Ambiguous constructor-arg entries for index " + index, ele);
             }
             else {
                bd.getConstructorArgumentValues().addIndexedArgumentValue(index,
valueHolder);
             }
          }
          finally {
             this.parseState.pop();
          }
       }
    }
    catch (NumberFormatException ex) {
       error("Attribute 'index' of tag 'constructor-arg' must be an integer", ele);
    }
 }
 else {//不存在index
    try {
       this.parseState.push(new ConstructorArgumentEntry());
       Object value = parsePropertyValue(ele, bd, null);
       ConstructorArgumentValues.ValueHolder valueHolder = new
ConstructorArgumentValues.ValueHolder(value);
       if (StringUtils.hasLength(typeAttr)) {
          valueHolder.setType(typeAttr);
       }
       if (StringUtils.hasLength(nameAttr)) {
          valueHolder.setName(nameAttr);
       }
       valueHolder.setSource(extractSource(ele));
       bd.getConstructorArgumentValues().addGenericArgumentValue(valueHolder);
    }
    finally {
       this.parseState.pop();
    }
 }
}
```

如果配置中指定了index属性，那么操作步骤如下。

（1） 解析constructor-arg的子元素。

（2） 使用ConstructorArgumentValues.ValueHolder类型来封装解析出来的元素。

（3） 将type、name和index属性一并封装在ConstructorArgumentValues.ValueHolder类型中并添加至当前BeanDefinition的constructorArgumentValues的IndexedArgumentValues属性中。

如果没有指定index属性，那么操作步骤如下：

（1） 解析constructor-arg的子元素。

（2） 使用ConstructorArgumentValues.ValueHolder类型来封装解析出来的元素。

（3） 将type、name和index属性一并封装在ConstructorArgumentValues.ValueHolder类型中并添加至当前BeanDefinition的constructorArgumentValues的genericArgumentValues属性中。

可以看到，对于是否指定index属性来讲，Spring的处理流程是不同的，关键在于属性信息被保存的位置。

解析构造函数中子元素的过程parsePropertyValue(ele, bd, null);

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

   // Should only have one child element: ref, value, list, etc.
   NodeList nl = ele.getChildNodes();
   Element subElement = null;
    //找到子元素
   for (int i = 0; i < nl.getLength(); i++) {
      Node node = nl.item(i);
      if (node instanceof Element && !nodeNameEquals(node, DESCRIPTION_ELEMENT) &&
            !nodeNameEquals(node, META_ELEMENT)) {//description, meta不解析
         // Child element is what we're looking for.
         if (subElement != null) {
            error(elementName + " must not contain more than one sub-element", ele);
         }
         else {
            subElement = (Element) node;
         }
      }
   }

    //获取constructor-arg上的ref属性
   boolean hasRefAttribute = ele.hasAttribute(REF_ATTRIBUTE);
    //获取constructor-arg上的value属性
   boolean hasValueAttribute = ele.hasAttribute(VALUE_ATTRIBUTE);
   if ((hasRefAttribute && hasValueAttribute) ||
         ((hasRefAttribute || hasValueAttribute) && subElement != null)) {
```

```
            error(elementName +
                  " is only allowed to contain either 'ref' attribute OR 'value' attribute OR
sub-element", ele);
        }

        if (hasRefAttribute) {
            String refName = ele.getAttribute(REF_ATTRIBUTE);
            if (!StringUtils.hasText(refName)) {
                error(elementName + " contains empty 'ref' attribute", ele);
            }
             //使用RuntimeBeanReference封装对应的ref名称
            RuntimeBeanReference ref = new RuntimeBeanReference(refName);
            ref.setSource(extractSource(ele));
            return ref;
        }
        else if (hasValueAttribute) {
              // value属性的处理，使用TypedStringValue封装。
            TypedStringValue valueHolder = new
TypedStringValue(ele.getAttribute(VALUE_ATTRIBUTE));
            valueHolder.setSource(extractSource(ele));
            return valueHolder;
        }
        else if (subElement != null) {
             //处理子元素
            return parsePropertySubElement(subElement, bd);
        }
        else {
            // Neither child element nor "ref" or "value" attribute found.
            error(elementName + " must specify a ref or value", ele);
            return null;
        }
}
```

（1） 略过description或者meta。

（2） 提取constructor-arg上的ref和value属性，以便于根据规则验证正确性，其规则为在constructor-arg上不存在以下情况。

同时既有ref属性又有value属性。

存在ref属性或者value属性且又有子元素

（3） Ref属性的处理。使用RuntimeBeanReference封装对应的ref名称。如：

（4） value属性的处理。使用TypedStringValue封装，例如：

（5） 子元素的处理。例如：

```
<constructor-arg>

    <map>

     <entry key="key" value="value" />

    </map>

</ constructor-arg>
```

parsePropertyElements(Element beanEle, BeanDefinition bd)

解析property

```
public void parsePropertyElement(Element ele, BeanDefinition bd) {
        String propertyName = ele.getAttribute(NAME_ATTRIBUTE); //获取name值
        if (!StringUtils.hasLength(propertyName)) {
            error("Tag 'property' must have a 'name' attribute", ele);
            return;
        }
        this.parseState.push(new PropertyEntry(propertyName));
        try {
            if (bd.getPropertyValues().contains(propertyName)) {//不允许对同一个name重复配
置保证property唯一
                error("Multiple 'property' definitions for property '" + propertyName +
"'", ele);
                return;
            }
            //解析property值
            Object val = parsePropertyValue(ele, bd, propertyName);
            PropertyValue pv = new PropertyValue(propertyName, val);
            parseMetaElements(ele, pv);
            pv.setSource(extractSource(ele));
            //将property值放入BeanDefinition
            bd.getPropertyValues().addPropertyValue(pv);
        }
        finally {
            this.parseState.pop();
        }
    }
```

注册bean

registerBeanDefinition(bdHolder, getReaderContext().getRegistry())

对解析完成的bean进行注册

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

   // 注册bean.
   String beanName = definitionHolder.getBeanName();
   registry.registerBeanDefinition(beanName, definitionHolder.getBeanDefinition());

    // 如果还有别名的话，也要根据别名统统注册一遍，不然根据别名就找不到 Bean 了.
   String[] aliases = definitionHolder.getAliases();
   if (aliases != null) {
      for (String alias : aliases) {
          // alias -> beanName 保存它们的别名信息，这个很简单，用一个 map 保存一下就可以了，
         // 获取的时候，会先将 alias 转换为 beanName，然后再查找
         registry.registerAlias(beanName, alias);
      }
   }
}
```

真正的注册bean其实是委托给DefaultListableBeanFactory进行的

```
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 {//对methodOverrides进行校验，其实是Spring配置文件中的lookup-method和replace-
method
            ((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);
    //判断相同名称的Bean是否已经注册过
    if (oldBeanDefinition != null) {
```

```
        if (!isAllowBeanDefinitionOverriding()) {// 如果不允许覆盖的话，抛异常
            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()) {// 检查下容器是否进入了Bean的创建阶段，即是否同时创建了任
何bean
            // 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 {
            // // 注册BeanDefinition
            this.beanDefinitionMap.put(beanName, beanDefinition);
            this.beanDefinitionNames.add(beanName);
            this.manualSingletonNames.remove(beanName);
        }
        this.frozenBeanDefinitionNames = null;
    }

    if (oldBeanDefinition != null || containsSingleton(beanName)) {
        resetBeanDefinition(beanName);
  // 重置所有已经注册过的BeanDefinition或单例模式的BeanDefinition的缓存
    }
}
```

- 首先 BeanDefinition 进行校验，该校验也是注册过程中的最后一次校验了，主要是对 AbstractBeanDefinition 的 methodOverrides 属性进行校验
- 根据 beanName 从缓存中获取 BeanDefinition，如果缓存中存在，则根据 allowBeanDefinitionOverriding 标志来判断是否允许覆盖，如果允许则直接覆盖，否则抛出 BeanDefinitionStoreException 异常
- 若缓存中没有指定 beanName 的 BeanDefinition，则判断当前阶段是否已经开始了 Bean 的创建阶段（），如果是，则需要对 beanDefinitionMap 进行加锁控制并发问题，否则直接设置即可。
- 若缓存中存在该 beanName 或者 单利 bean 集合中存在该 beanName，则调用 **resetBeanDefinition()** 重置 BeanDefinition 缓存。

registerAlias(String name, String alias)对别名进行注册

```
public void registerAlias(String name, String alias) {
   Assert.hasText(name, "'name' must not be empty");
   Assert.hasText(alias, "'alias' must not be empty");
   if (alias.equals(name)) { //如果beanName和别名相同则不注册
      this.aliasMap.remove(alias);
   }
   else {
      String registeredName = this.aliasMap.get(alias);
      if (registeredName != null) {
         if (registeredName.equals(name)) {
            // An existing alias - no need to re-register
            return;
         }
         if (!allowAliasOverriding()) {
```

```
            throw new IllegalStateException("Cannot register alias '" + alias + "' for
name '" +
                    name + "': It is already registered for name '" + registeredName +
"'.");
          }
      }
       //若A->B存在，再次出现A->C->B时抛出异常
      checkForAliasCircle(name, alias);
      this.aliasMap.put(alias, name);
   }
}
```

getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder))

通知监听器bean解析、注册完成。这里实现是为了进行扩展。spring没有做监听处理

4 prepareBeanFactory(ConfigurableListableBeanFactory beanFactory)

配置bean工厂

```
protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {

    // Tell the internal bean factory to use the context's class loader etc.
    //告知内部的bean工厂，使用上下文的类加载器
    beanFactory.setBeanClassLoader(getClassLoader());

    //设置bean表达式解析器，
    //StandardBeanExpressionResolver内部expressionParser属性默认SpelExpressionParser类型
    //spel = spring el表达式
    beanFactory.setBeanExpressionResolver(new
StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));

    //将ResourceEditorRegistrar实例添加到工厂的propertyEditorRegistrars属性中，
    //propertyEditorRegistrars是一个LinkedHashSet，里面的元素将会应用到工厂bean中
    //ResourceEditorRegistrar持有上下文和environment的引用
    beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this,
getEnvironment()));

    // Configure the bean factory with context callbacks.
    // 使用上下文回调配置bean 工厂
    //在工厂的beanPostProcessor属性中添加处理器，beanPostProcessor是一个ArrayList
    beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));

    //在工厂的ignoredDependencyInterfaces属性中添加Aware系列接口，
    //ignoredDependencyInterfaces是一个HashSet
    //忽略自动装配
```

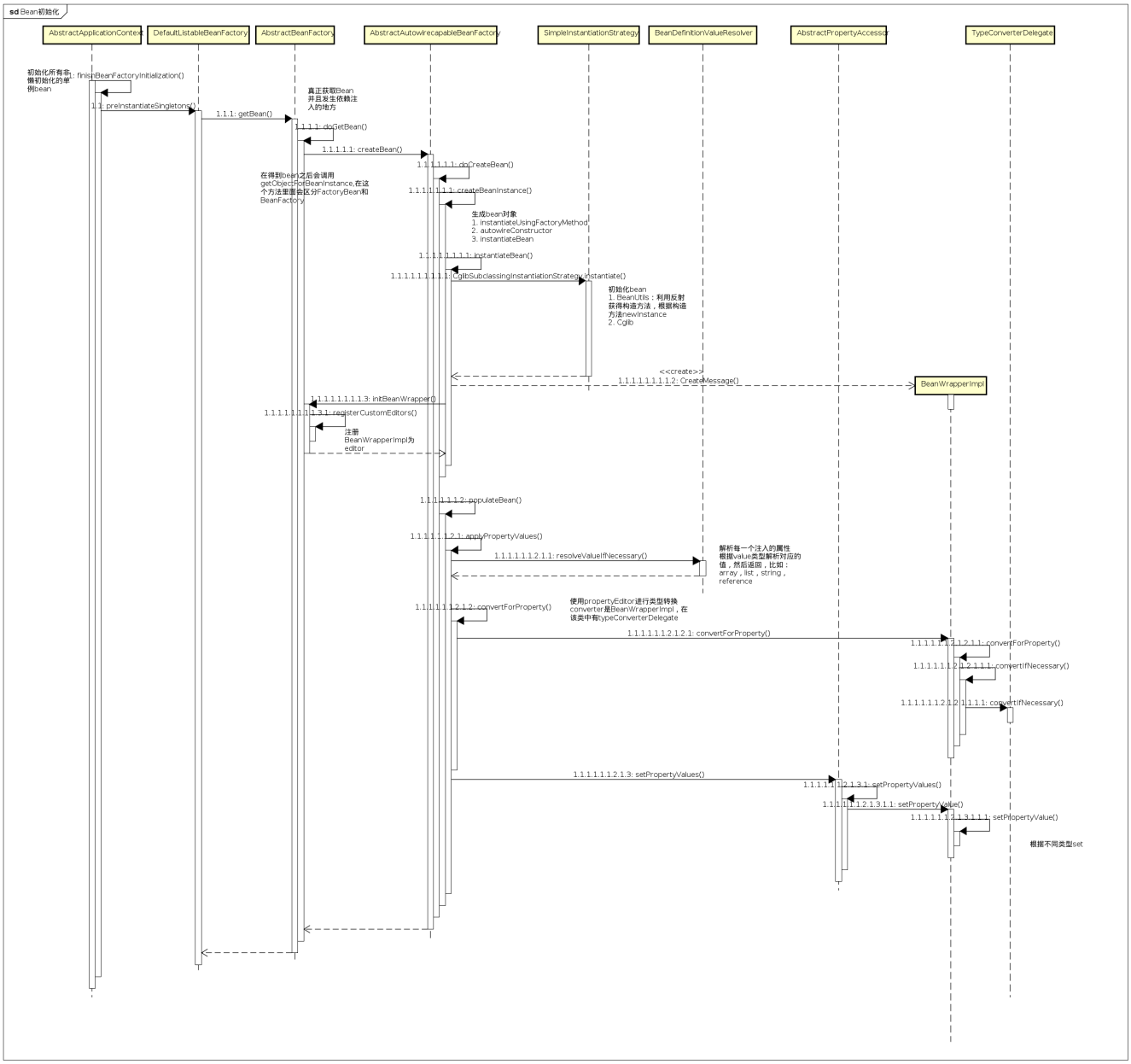
```
    beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
    beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
    beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
    beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);
    beanFactory.ignoreDependencyInterface(EnvironmentAware.class);

    // BeanFactory interface not registered as resolvable type in a plain factory.
    // MessageSource registered (and found for autowiring) as a bean.

    // 在普通的工厂中，BeanFactory接口并没有按照resolvable类型进行注册
    // MessageSource被注册成一个Bean（并被自动注入）

    //BeanFactory.class为key，beanFactory为value放入到了beanFactory的
resolvableDependencies属性中
    //resolvableDependencies是一个ConcurrentHashMap,映射依赖类型和对应的被注入的value
    //这样的话BeanFactory/ApplicationContext虽然没有以bean的方式被定义在工厂中，
    //但是也能够支持自动注入，因为他处于resolvableDependencies属性中
    beanFactory.registerResolvableDependency(BeanFactory.class, beanFactory);

    //再将上下文的一些接口与上下文本身做映射，一一放入到resolvableDependencies中
    beanFactory.registerResolvableDependency(ResourceLoader.class, this);
    beanFactory.registerResolvableDependency(ApplicationEventPublisher.class, this);
    beanFactory.registerResolvableDependency(ApplicationContext.class, this);

    // Detect a LoadTimeWeaver and prepare for weaving, if found.
    // 检测LoadTimeWeaver，如果有就准备织入
    if (beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {

        //如果有LoadTimeWeaver，加入bean后处理器
        beanFactory.addBeanPostProcessor(new
LoadTimeWeaverAwareProcessor(beanFactory));

        // Set a temporary ClassLoader for type matching.
        // 为匹配类型设置一个临时的ClassLoader
        beanFactory.setTempClassLoader(new
ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));
    }

    // Register default environment beans.
    // 注册默认的environment beans

    // 判断目前这个bean工厂中是否包含指定name的bean，忽略父工厂
    if (!beanFactory.containsLocalBean(ENVIRONMENT_BEAN_NAME)) {


        //2.注册environment单例
        beanFactory.registerSingleton(ENVIRONMENT_BEAN_NAME, getEnvironment());
    }
```

```
    if (!beanFactory.containsLocalBean(SYSTEM_PROPERTIES_BEAN_NAME)) {

        //注册systemProperties单例
        beanFactory.registerSingleton(SYSTEM_PROPERTIES_BEAN_NAME,
getEnvironment().getSystemProperties());
    }
    if (!beanFactory.containsLocalBean(SYSTEM_ENVIRONMENT_BEAN_NAME)) {

        //注册systemEnvironment单例
        beanFactory.registerSingleton(SYSTEM_ENVIRONMENT_BEAN_NAME,
getEnvironment().getSystemEnvironment());
    }
}
```

- 为工厂设置类的加载器、表达式解析器、属性编辑器注册器等
- 为工厂添加后处理器、要忽略的依赖接口
- 在工厂中注册可解析的依赖
- 在工厂中提前注册一些单例Bean

5.finishBeanFactoryInitialization(beanFactory)

通过上面的步骤之后只是注册了beanDefinition，bean的初始化就是从这开始的，这个方法的功能是初始化非懒加载的单例bean, 实例化所有剩余的 bean 实例（非懒加载）。除了一些内部的bean、实现了 BeanFactoryPostProcessor 的 bean、实现了 BeanPostProcessor 的 bean，其他的 bean 都会在这个 finishBeanFactoryInitialization 方法中被实例化。




```
    protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory
beanFactory) {
        // 初始化名字为 conversionService 的 Bean.
        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));
        }
    // //初始化LoadTimeWeaverAware类型的Bean  AspectJ部分.
    String[] weaverAwareNames =
beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, false);
    for (String weaverAwareName : weaverAwareNames) {
```

```
            getBean(weaverAwareName);
        }

        // 停止使用临时的类加载器，LoadTimeWeaver使用.
        beanFactory.setTempClassLoader(null);

        // // 冻结上下文，不允许再进行修改配置
        beanFactory.freezeConfiguration();

        // 实例化所有剩余（非懒加载）单例对象
        beanFactory.preInstantiateSingletons();
    }
```

5.1 preInstantiateSingletons()

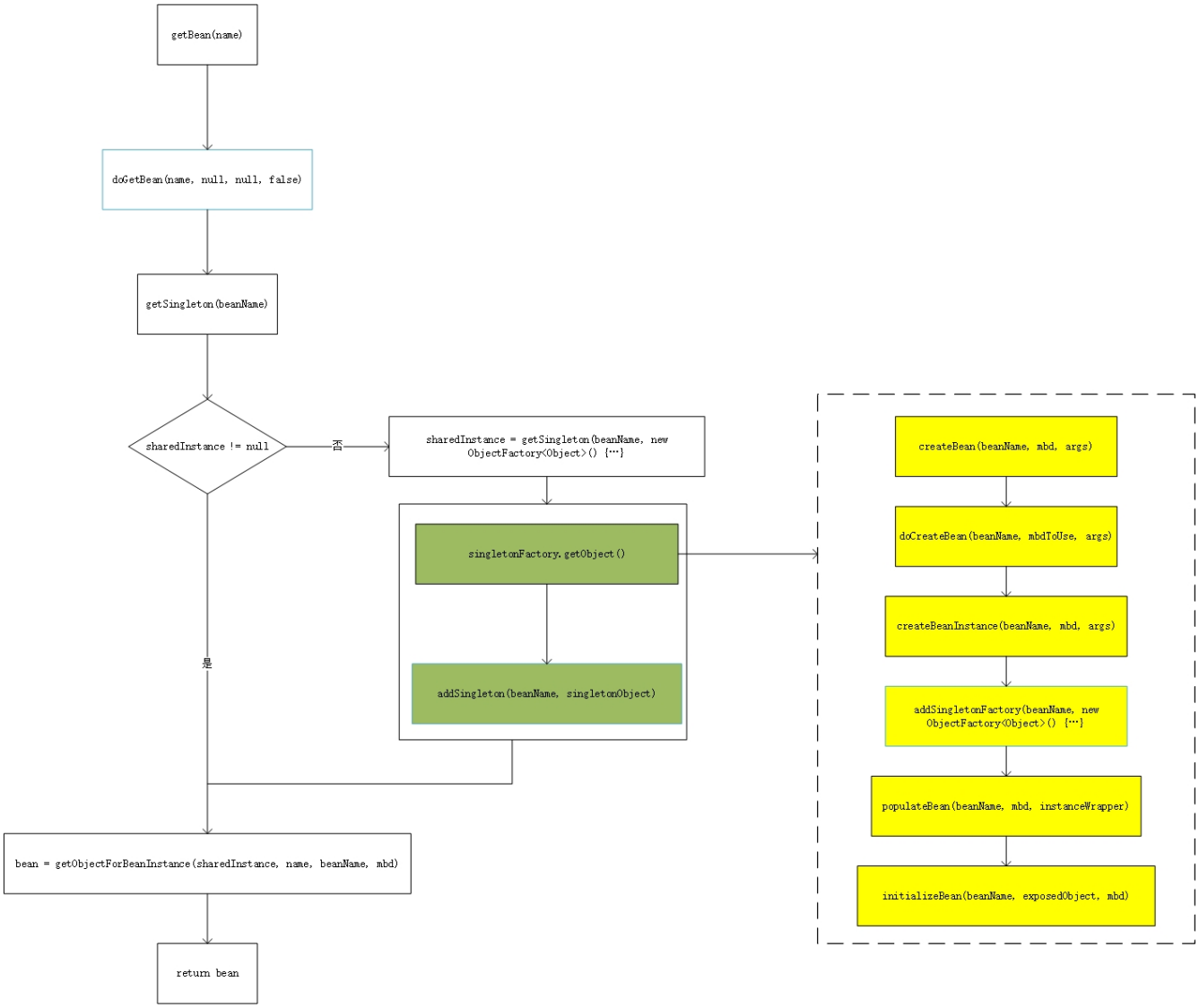
初始化

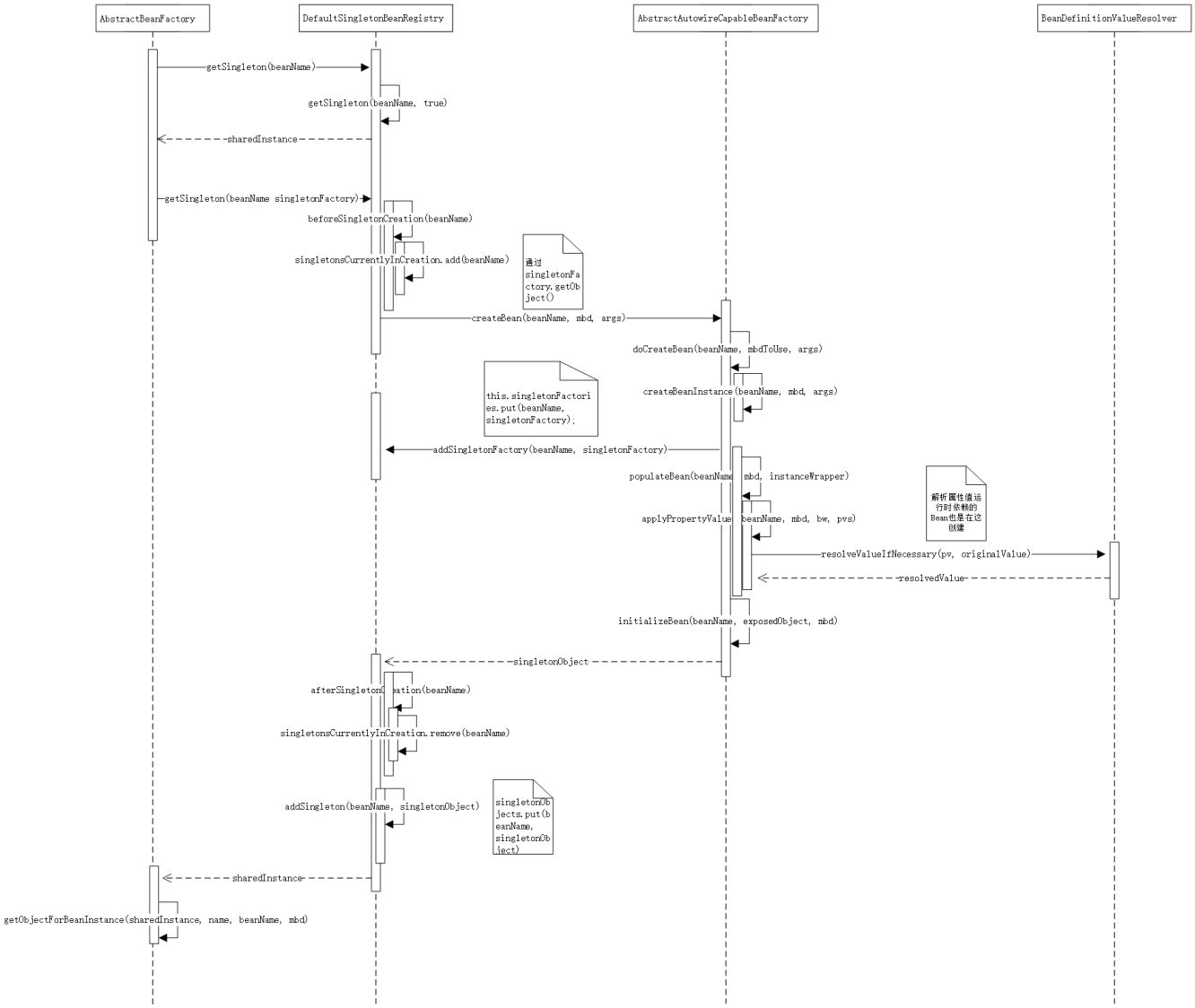
```
public void preInstantiateSingletons() throws BeansException {
    if (this.logger.isDebugEnabled()) {
        this.logger.debug("Pre-instantiating singletons in " + this);
    }

    // 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.
    // 1.创建beanDefinitionNames的副本beanNames用于后续的遍历，以允许init等方法注册新的bean定义
    List<String> beanNames = new ArrayList<String>(this.beanDefinitionNames);

    // Trigger initialization of all non-lazy singleton beans...
    // 2.遍历beanNames，触发所有非懒加载单例bean的初始化
    for (String beanName : beanNames) {
        // 3.获取beanName对应的合并的BeanDefinition对象
        RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
        // 4.bd对应的Bean实例：不是抽象类 && 是单例 && 不是懒加载
        if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
            // 5.如果beanName对应的bean是FactoryBean
            if (isFactoryBean(beanName)) {
                // 5.1 通过getBean(&beanName)拿到的是FactoryBean本身，（getBean(beanName)拿
到的是FactoryBean创建的Bean实例）
                final FactoryBean<?> factory = (FactoryBean<?>)
getBean(FACTORY_BEAN_PREFIX + beanName);
                // 5.2 判断这个FactoryBean是否希望急切的初始化
                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) {
                    // 5.3 通过beanName获取bean实例
                    getBean(beanName);
                }
            } else {
                // 6.如果beanName对应的bean不是FactoryBean，只是普通Bean，通过beanName获取
bean实例
                getBean(beanName);
            }
        }
    }

    // Trigger post-initialization callback for all applicable beans...
    // 7.遍历beanNames，触发所有SmartInitializingSingleton的后初始化回调
    for (String beanName : beanNames) {
        // 8.拿到beanName对应的bean实例
        Object singletonInstance = getSingleton(beanName);
        // 9.判断singletonInstance是否实现了SmartInitializingSingleton接口
        if (singletonInstance instanceof SmartInitializingSingleton) {
            final SmartInitializingSingleton smartSingleton =
(SmartInitializingSingleton) singletonInstance;
            // 10.调用SmartInitializingSingleton实现类的afterSingletonsInstantiated方法
            if (System.getSecurityManager() != null) {
                AccessController.doPrivileged(new PrivilegedAction<Object>() {
                    @Override
                    public Object run() {
                        smartSingleton.afterSingletonsInstantiated();
                        return null;
                    }
                }, getAccessControlContext());
            } else {
                smartSingleton.afterSingletonsInstantiated();
            }
        }
    }
}
```

5.2 getBean(beanName)

这个方法用来创建一个bean第一次调用此方法创建一个单例的bean的流程图大致如下






```
protected <T> T doGetBean(
            final String name, final Class<T> requiredType, final Object[] args,
boolean typeCheckOnly)
            throws BeansException {
//把当前传入的beanName转化为标准beanName,例如把去除&引用(如果带&表示要获取FactoryBean)、alias转化为标准的beanName等
        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
         *为什么首先会使用这段代码呢，因为在创建单例bean的时候会存在依赖注入的情况，而在创建依赖的时候为了避免循环依赖，
         *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创建时候需要依赖上个bean则直接使用ObjectFactory
```

```
         */
        //从缓存或者实例工厂中获取bean
        Object sharedInstance = getSingleton(beanName);
        if (sharedInstance != null && args == null) {
            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 + "'");
                }
            }
            /*获得给定bean实例的对象。这里的处理过程是这样的:
            *(1)如果sharedInstance不是FactoryBean实例，而是个普通bean，那么直接返回;
            *(2)如果sharedInstance是FactoryBean实例，且name要求返回FactoryBean实例，那么直接返
回;
            *(3)否则从本地缓存获得合并BeanDefinition,如果本地缓存不存在合并BeanDefinition，则从
本地缓存取得BeanDefinition，构造合并     BeanDefinition，然后通过factoryBean获得bean*/
            bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
        }

        else {
            //检查beanName所代表的bean是否处于正在创建阶段，主要是防止循环引用
            //A中有B的属性，B中有A的属性，那么当依赖注入的时候，就会产生当A还未创建完的时候因为
    //对于B的创建再次返回创建A，造成循环依赖，
            if (isPrototypeCurrentlyInCreation(beanName)) {
                throw new BeanCurrentlyInCreationException(beanName);
            }

            //对容器中的BeanDefinition是否存在进行检查，检查是否能在当前的Bean Factory中获取的需
要的Bean。如果在当前的工厂中取不到就到双亲的beanFactory中取。如果父工厂中存在当前工厂将委托父工厂完成
getBean的任务。如果当前的双亲工长取不到就顺着双亲的BeanFactory链一直向上找
            BeanFactory parentBeanFactory = getParentBeanFactory();
            if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
                // Not found -> check parent.
                String nameToLookup = originalBeanName(name);
                if (args != null) {
                    // Delegation to parent with explicit args.
                    return (T) parentBeanFactory.getBean(nameToLookup, args);
                }
                else {
                    // No args -> delegate to standard getBean method.
                    return parentBeanFactory.getBean(nameToLookup, requiredType);
                }
```

```
            }
            //是否要类型检查，如果不检查，直接把beanName标记为已创建
            if (!typeCheckOnly) {
                markBeanAsCreated(beanName);
            }

            try {
                //根据beanName获取beanDefiniton并合并父bean的属性
                final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
                checkMergedBeanDefinition(mbd, beanName, args);

                 //如果bean存在依赖bean，先构建依赖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 + "'");
                        }
                        registerDependentBean(dep, beanName);
                        getBean(dep);
                    }
                }

                // Create bean instance.
                if (mbd.isSingleton()) {//单例时
                    sharedInstance = getSingleton(beanName, new ObjectFactory<Object>()
{
                        @Override
                        public Object getObject() throws BeansException {
                            try {
                                /*所有生成Bean都有BeanWrapper封装，bean的生成采用策略模式，
CglibSubclassingInstantiationStrategy实现                                    是默认的策略
                                1.如果配置为工厂方法创建(配置了factory-method的)，由java反射
Mehthod的invoke完成bean的创建。
                                2.如果采用构造子配置创建(配置了constructor-arg的)，如果bean对
应的类包含多个构造子，采用cglib动态字节码构                                    造；如果只有唯一的构
造子，那么采用java反射Constructor的newInstance方法
                                3.如果是普通bean配置,直接通过反射Class默认的Constructor，然后
调用newInstance获得bean.*/
                                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.
                                // 因为单例模式下为了解决循环依赖，可能他已经存在了，所以销毁它
                                destroySingleton(beanName);
                                throw ex;
                            }
                        }
                    });
                    /*由于当前构造的bean可能是FactoryBean，所以要通过以下方法获得真实需要的bean
                    (1)如果sharedInstance不是FactoryBean实例，而是个普通bean，那么直接返回；
                    (2)如果sharedInstance是FactoryBean实例，且name要求返回FactoryBean实例，
那么直接返回；
                    (3)否则从本地缓存获得合并BeanDefinition,如果本地缓存不存在合并
BeanDefinition，则从本地缓存取得BeanDefinition，构造合                    并BeanDefinition，然
后通过factoryBean获得bean*/
                    bean = getObjectForBeanInstance(sharedInstance, name, beanName,
mbd);
                }

                else if (mbd.isPrototype()) {
                    // It's a prototype -> create a new instance.
                    Object prototypeInstance = null;
                    try {
                        //维护prototypesCurrentlyInCreation，主要把beanName加入到其中
                        beforePrototypeCreation(beanName);
                        prototypeInstance = createBean(beanName, mbd, args);
                    }
                    finally {
                        afterPrototypeCreation(beanName);
                    }
                    bean = getObjectForBeanInstance(prototypeInstance, name, beanName,
mbd);
                }

                else {
                    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;
            }
        }

         //检查需要的类型是否符合bean的实际类型
        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;
    }
```

对于Spring加载bean的过程，大致分为以下几步：

1.转换对应beanName

这里传入的参数name不一定就是beanName，有可能是别名或FactoryBean，所以需要进行一系列的解析，这些解析内容包括如下内容

去除FactoryBean的修饰符，也就是如果name=”&aa”，那么会首先去除&而使name=”aa” 。关于FactoryBean的用法 https://blog.csdn.net/w_linux/article/details/80063062

取指定alias所表示的最终beanName，例如别名A指向名称为B的bean则返回B；若别名A指向别名B，别名B又指向名称为C的bean则返回C

2.尝试从缓存中加载单例

单例在Spring的同一个容器内只会被创建一次，后续再获取bean，就直接从单例缓存中获取了。这里只是尝试加载，首先尝试从缓存中加载，如果加载不成功则再次尝试从singletonFactories中加载，因为在创建单例bean的时候会存在依赖注入的情况，而在创建依赖的时候为了避免循环依赖，在Spring中创建bean的原则是不等bean创建完成就会将创建bean的ObjectFactory提早曝光加入到缓存中，一旦下一个bean创建时候需要依赖上一个bean则直接使用ObjectFactory

3.bean的实例化

如果从缓存中得到了bean的原始状态，则需要对bean进行实例化，这里有必要强调一下，在缓存中记录的只是最原始的bean状态，并不一定是我们最终想要的bean

4.原型模式的依赖检查

只有在单例情况下才会尝试解决循环依赖，如果存在A中有B的属性，B中有A的属性，那么当依赖注入的时候，就会产生当A还未创建完的时候因为对于B的创建再次返回创建A，造成循环依赖，也就是情况：isPrototypeCurrentlyInCreation(beanName)判断true

5.检测parentBeanFactory

从代码上来看，如果缓存没有数据的话直接转到父类工厂上去加载，!this.containsBeanDefinition(beanName检测如果当前加载的XML配置文件中不包含beanName所对应的配置，就只能到parentBeanFactory去尝试，然后再去递归的调用getBean方法

6.将存储XML配置文件的GernericBeanDefinition转换为RootBeanDefinition

因为从XML配置文件中读取到的Bean信息是存储在GernericBeanDefinition中的，但是所有的Bean后续处理都是针对于RootBeanDefinition的，所以这里需要进行一个转换，转换的同时如果父类bean不为空的话，则会一并合并父类属性。

在容器初始化阶段，对解释资源获得的BeanDefinition是由GenericBeanDefinition来定义，其只能刻画bean自身的一些特性。但bean与bean之间还存在着继承、方法覆盖、包含等一些复杂的关系。而这些关系的刻画是由RootBeanDefinition来负责的。RootBeanDefinition也就是我们下面所说的mergedBeanDefinition，mergedBeanDefinition是指对具有继承关系的bean所对应的BeanDefinition的合成，以达到完成表达一个bean的目的

7.寻找依赖

因为bean的初始化过程很可能会用到某些属性，而某些属性很可能是动态配置的，并且配置成依赖于其他的bean，那么这个时候就有必要先加载依赖的bean，所以，在Spring的加载顺寻中，在初始化某一个bean的时候首先会初始化这个bean所对应的依赖

8.针对不同的scope进行bean的创建

在Spring中存在着不同的scope，其中默认的是singleton，但是还有些其他的配置诸如prototype、request之类的，在这个步骤中，Spring会根据不同的配置进行不同的初始化策略

9.类型转换

程序到这里返回bean后已经基本结束了，通常对该方法的调用参数requiredType是为空的，但是可能会存在这样的情况，返回的bean其实是个Spring，但是requiredType却传入Integer类型，那么这时候本步骤就会起作用了，它的功能是将返回的bean转换为requiredType所指定的类型，当然，Spring转换为Integer是最简单的一种转换，在Spring中提供了各种各样的转换器，用户也可以自己扩展转换器来满足需求

5.2.1 getSingleton

```
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    // 1.从单例对象缓存中获取beanName对应的单例对象
    Object singletonObject = this.singletonObjects.get(beanName);
    // 2.如果单例对象缓存中没有，并且该beanName对应的单例bean正在创建中
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
        // 3.加锁进行操作
        synchronized (this.singletonObjects) {
            // 4.从早期单例对象缓存中获取单例对象（之所以成为早期单例对象，是因为
earlySingletonObjects里
            // 的对象的都是通过提前曝光的ObjectFactory创建出来的，还未进行属性填充等操作）
            singletonObject = this.earlySingletonObjects.get(beanName);
            // 5.如果在早期单例对象缓存中也没有，并且允许创建早期单例对象引用
            if (singletonObject == null && allowEarlyReference) {
                // 6.从单例工厂缓存中获取beanName的单例工厂
                ObjectFactory<?> singletonFactory =
this.singletonFactories.get(beanName);
                if (singletonFactory != null) {
                    // 7.如果存在单例对象工厂，则通过工厂创建一个单例对象
                    singletonObject = singletonFactory.getObject();
                    // 8.将通过单例对象工厂创建的单例对象，放到早期单例对象缓存中
                    this.earlySingletonObjects.put(beanName, singletonObject);
                    // 9.移除该beanName对应的单例对象工厂，因为该单例工厂已经创建了一个实例对象，并
且放到earlySingletonObjects缓存了，
                    // 因此，后续获取beanName的单例对象，可以通过earlySingletonObjects缓存拿
到，不需要在用到该单例工厂
                    this.singletonFactories.remove(beanName);
                }
            }
        }
    }
```

```
    // 10.返回单例对象
    return (singletonObject != NULL_OBJECT ? singletonObject : null);
 }
```

这个方法首先尝试从singletonObjects里面获取实例，如果获取不到再从earlySingletonObjects里面获取，如果还获取不到，再尝试从singletonFactories里面获取beanName对应的ObjectFactory，然后调用这个ObjectFactory的getObject来创建bean，并放到earlySingletonObjects里面去，并且从singletonFactories里面remove掉这个ObjectFactory，而对于后续的所有内存操作都只为了循环依赖检测时候使用，也就是在allowEarlyReference为true的情况下才会使用

这里涉及用于存在bean的不同map，说明如下：

singletonObjects：用于保存BeanName和创建bean实例之间的关系，bean name->bean instance
singletonFactories：用于保存BeanName和创建bean的工厂之间的关系，bean name->ObjectFactory
earlySingletonObjects：也是保存BeanName和创建bean实例之间的关系，与singletonObjects的不同之处在于，当一个单例bean被放到这里面后，那么当bean还在创建过程中，就可以通过getBean方法获取到了，其目的是用来检测循环引用

registeredSingletons：用来保存当前所有已注册的bean

这段代码很重要，在正常情况下，该代码很普通，只是正常的检查下我们要拿的 bean 实例是否存在于缓存中，如果有就返回缓存中的 bean 实例，否则就返回 null。

这段代码之所以重要，是因为该段代码是 Spring 解决循环引用的核心代码。

解决逻辑：我们先用构造函数创建一个 “不完整” 的 bean 实例（之所以说不完整，是因为此时该 bean 实例还未初始化），并且提前曝光该 bean 实例的 ObjectFactory（提前曝光就是将 ObjectFactory 放到 singletonFactories 缓存），通过 ObjectFactory 我们可以拿到该 bean 实例，如果出现循环引用，我们可以通过缓存中的 ObjectFactory 来拿到 bean 实例，从而避免出现循环引用导致的死循环。这边通过缓存中的 ObjectFactory 拿到的 bean 实例虽然拿到的是 “不完整” 的 bean 实例，但是由于是单例，所以后续初始化完成后，该 bean 实例的引用地址并不会变，所以最终我们看到的还是完整 bean 实例。

5.2.2 getObjectForBeanInstance

该方法从名称上就能看出，其实现的功能是从一个给定的bean实例，获取客户端调用所真实想要的bean。由于Spring能够管理各式的bean，当bean是FactoryBean实例的时候，客户端调用可以通过&+beanName的方式获得FactoryBean实例，而如果bean实例是普通的bean，那么该方法直接返回。

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

/*如果bean是工厂解除参照(即name带有&)，且不是FactoryBean实例，那么抛出异常。这里的意思就是说如果name带有&，而传进来的又不是FactoryBean实例*/
        if (BeanFactoryUtils.isFactoryDereference(name) && !(beanInstance instanceof
FactoryBean)) {
            throw new BeanIsNotAFactoryException(transformedBeanName(name),
beanInstance.getClass());
```

```
        }

/*现在我们拥有了bean实例，它可能是一个普通bean或者一个FactoryBean.如果是一个工厂bean，我们使用它创建
一个bean实例，除非调用者确实想要工厂的引用。
如果bean实例未实现FactoryBean，或者要求返回BeanFactory自身，那么直接返回。*/
        if (!(beanInstance instanceof FactoryBean) ||
BeanFactoryUtils.isFactoryDereference(name)) {
            return beanInstance;
        }

    /*以下的代码意味着处理非普通bean，而是FactoryBean之类的*/
        Object object = null;
        if (mbd == null) {//
/*这里主要是从一个FactoryBean到object映射的缓存中，取出由FactoryBean创建的object.这种映射关系是由
factoryBeanObjectCache(映射关系：FactoryBean name --> object ，例如'myJndi'---
>myJndiObject)字段维护，该字段位于FactoryBeanRegistrySupport中*/
            object = getCachedObjectForFactoryBean(beanName);
        }

        if (object == null) {

            FactoryBean factory = (FactoryBean) beanInstance;
/*containsBeanDefinition方法是由默认内部工厂DefaultListableBeanFactory实现，因为
BeanDefinition是缓存在它之下的*/
            if (mbd == null && containsBeanDefinition(beanName)) {
/*从本地获得指定beanName的合并BeanDefinition，所谓的合并BeanDefinition就是指子类的
BeanDefinition与父类BeanDefinition的合并。*/
                mbd = getMergedLocalBeanDefinition(beanName);
            }
/*判断beanDefinition是否是合成的，即是否与其他beanDefinition进行了合并*/
            boolean synthetic = (mbd != null && mbd.isSynthetic());
            //从factoryBean获得bean
            object = getObjectFromFactoryBean(factory, beanName, !synthetic);
        }
        return object;
    }
```

5.2.3 getSingleton(String beanName, ObjectFactory<?> singletonFactory)

```
public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {
    Assert.notNull(beanName, "'beanName' must not be null");
    synchronized (this.singletonObjects) {
        // 从缓存中获取单例 bean，若不为空，则直接返回，不用再初始化
        Object singletonObject = this.singletonObjects.get(beanName);
        if (singletonObject == null) {
```

```
            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
+ "'");
            }
            /*
             * 将 beanName 添加到 singletonsCurrentlyInCreation 集合中,
             * 用于表明 beanName 对应的 bean 正在创建中
             */
            beforeSingletonCreation(beanName);
            boolean newSingleton = false;
            boolean recordSuppressedExceptions = (this.suppressedExceptions == null);
            if (recordSuppressedExceptions) {
                this.suppressedExceptions = new LinkedHashSet<Exception>();
            }
            try {
                // 通过 getObject 方法调用 createBean 方法创建 bean 实例
                singletonObject = singletonFactory.getObject();
                newSingleton = true;
            }
            catch (IllegalStateException ex) {
                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;
                }
                // 将 beanName 从 singletonsCurrentlyInCreation 移除
                afterSingletonCreation(beanName);
            }
            if (newSingleton) {
```

```
                /*
                 * 将 <beanName, singletonObject> 键值对添加到 singletonObjects 集合中，
                 * 并从其他集合（比如 earlySingletonObjects）中移除 singletonObject 记录
                 */
                addSingleton(beanName, singletonObject);
            }
        }
        return (singletonObject != NULL_OBJECT ? singletonObject : null);
    }
}
```

上面的方法逻辑不是很复杂，这里简单总结一下。如下：

1. 先从 singletonObjects 集合获取 bean 实例，若不为空，则直接返回
2. 若为空，进入创建 bean 实例阶段。先将 beanName 添加到 singletonsCurrentlyInCreation
3. 通过 getObject 方法调用 createBean 方法创建 bean 实例
4. 将 beanName 从 singletonsCurrentlyInCreation 集合中移除
5. 将 <beanName, singletonObject> 映射缓存到 singletonObjects 集合中

5.2.4 createBean

```
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.
    //根据class属性或className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 {
       //准备方法重写
      mbdToUse.prepareMethodOverrides();
   }
   catch (BeanDefinitionValidationException ex) {
      throw new BeanDefinitionStoreException(mbdToUse.getResourceDescription(),
            beanName, "Validation of method overrides failed", ex);
```

```
    }

    try {
        //  如果Bean配置了PostProcessor，那么返回一个Proxy代替目标实例
       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);
    }
//所有生成Bean都有BeanWrapper封装，bean的生成采用策略模式，
CglibSubclassingInstantiationStrategy实现是默认的策略
//1.如果配置为工厂方法创建(配置了factory-mother的)，由java反射Mehthod的invoke完成bean的创建。
//2.如果采用构造子配置创建(配置了constructor-arg的)，如果bean对应的类包含多个构造子，采用cglib动态
字节码构造；如果只有唯一的构造子，那么采用java反射Constructor的newInstance方法
//3.如果是普通bean配置,直接通过反射Class默认的Constructor，然后调用newInstance获得bean.
    Object beanInstance = doCreateBean(beanName, mbdToUse, args);
    if (logger.isDebugEnabled()) {
       logger.debug("Finished creating instance of bean '" + beanName + "'");
    }
    return beanInstance;
}
```

上面的方法实现了一下几个功能：

1、根据设置的class属性或者className来解析、加载class 2、对override属性进行标记和验证（bean XML配置中的lookup-method和replace-method属性） 3、应用初始化前的后处理器，如果处理器中返回了AOP的代理对象，则直接返回该单例对象，不需要继续创建

5.2.4.1 doCreateBean

```
protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd,
final Object[] args)
      throws BeanCreationException {

   // Instantiate the bean.
   BeanWrapper instanceWrapper = null;
    //单例的情况下从factoryBeanInstanceCache移除 instanceWrapper
   if (mbd.isSingleton()) {
      instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
   }
   if (instanceWrapper == null) {
```

```
        //创建Bean
        //创建bean时根据bean对应的策略创建新的实例，如：工厂方法、构造器自动注入、简单初始化
       instanceWrapper = createBeanInstance(beanName, mbd, args);
    }
    final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstance()
: null);
    Class<?> beanType = (instanceWrapper != null ? instanceWrapper.getWrappedClass() :
null);
    mbd.resolvedTargetType = beanType;

    // bean初始化前调用post-processors，我们可以在bean实例化之前定制一些操作
    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.
     //如果当前bean是单例，且支持循环依赖，且当前bean正在创建，通过往singletonFactories添加一个
objectFactory，
     //这样后期如果有其他bean依赖该bean 可以从singletonFactories获取到bean，
getEarlyBeanReference可以对返回的bean进行修改，这边目前除了可能      会返回动态代理对象 其他的都是直
接返回bean
    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");
       }
        //为了避免循环依赖，在bean初始化完成前，就将创建bean实例的ObjectFactory放入工厂缓存
(singletonFactories)
       addSingletonFactory(beanName, new ObjectFactory<Object>() {
          @Override
          public Object getObject() throws BeansException {
               /*
                  * 获取原始对象的早期引用，在 getEarlyBeanReference 方法中，会执行 AOP
                  * 相关逻辑。若 bean 未被 AOP 拦截，getEarlyBeanReference 原样返回
                  * bean，所以大家可以把
```

```
                 *      return getEarlyBeanReference(beanName, mbd, bean)
                 * 等价于:
                 *      return bean;
                 */
            return getEarlyBeanReference(beanName, mbd, bean);
         }
      });
   }

   // Initialize the bean instance.
   Object exposedObject = bean;
   try {
       //对bean进行填充，将各个属性值注入，其中可能存在依赖其他bean的属性时就会递归初始依赖bean
      populateBean(beanName, mbd, instanceWrapper);
      if (exposedObject != null) {
          //调用初始化方法，比如init-method、注入Aware对象、应用后处理器
         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);
      }
   }

   if (earlySingletonExposure) {
       //从提前曝光的bean缓存中查询bean，目的是验证是否有循环依赖存在
           //如果存在循环依赖，也就是说该bean已经被其他bean递归加载过，放入了提早曝光的bean缓存中
       //只有检测到循环依赖的情况下，earlySingletonReference才不会为null
      Object earlySingletonReference = getSingleton(beanName, false);
      if (earlySingletonReference != null) {
          //如果exposedObject没有在 initializeBean 初始化方法中改变，也就是没有被增强
         if (exposedObject == bean) {
            exposedObject = earlySingletonReference;
         }
         else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName))
{
            String[] dependentBeans = getDependentBeans(beanName);
            Set<String> actualDependentBeans = new LinkedHashSet<String>
(dependentBeans.length);
            for (String dependentBean : dependentBeans) {
                //检测依赖
```

```
                if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
                   actualDependentBeans.add(dependentBean);
                }
             }
              //因为bean创建后，其依赖的bean一定也是已经创建的
             //如果actualDependentBeans不为空，则表示依赖的bean并没有被创建完，即存在循环依赖
             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.");
             }
          }
       }
   }

   // Register bean as disposable.
   try {
      registerDisposableBeanIfNecessary(beanName, bean, mbd);
   }
   catch (BeanDefinitionValidationException ex) {
      throw new BeanCreationException(
            mbd.getResourceDescription(), beanName, "Invalid destruction signature",
ex);
   }

   return exposedObject;
}
```

上面的方法完成了一下工作

1、如果是单例则首先尝试冲缓存中获取并清除缓存 2、实例化bean，并使用BeanWarpper包装 1如果存在工厂方法，则使用工厂方法实例化 2如果有多个构造函数，则根据传入的参数确定构造函数进行初始化 3使用默认的构造函数初始化 3、应用MergedBeanDefinitionPostProcessor，Autowired注解就是在这样完成的解析工作 4、依赖处理。如果A和B存在循环依赖，那么Spring在创建B的时候，需要自动注入A时，并不会直接创建再次创建A，而是通过放入缓存中A的ObjectFactory来创5、建实例，这样就解决了循环依赖的问题 6、属性填充。所有需要的属性都在这一步注入到bean 7、循环依赖检查 8、注册DisposableBean。如果配置了destroy-method，这里需要注册，以便在销毁时调用

完成创建并返回

5.2.4.1 createBeanInstance(beanName, mbd, args)

```
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);

    ////确保class不为空，并且访问权限为public
        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());
        }

        if (mbd.getFactoryMethodName() != null)  {
            //如果存在工厂方法则使用工厂方法进行初始化
            //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(beanName,
mbd, args)
            //方法根据RootBeanDefinition中的配置生成bean的实例。
            return instantiateUsingFactoryMethod(beanName, mbd, args);
        }

        // Shortcut when re-creating the same bean...
        boolean resolved = false;
        boolean autowireNecessary = false;
        if (args == null) {
            //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
            synchronized (mbd.constructorArgumentLock) {
                if (mbd.resolvedConstructorOrFactoryMethod != null) {
                    //已经解析过class的构造器
                    resolved = true;
                    autowireNecessary = mbd.constructorArgumentsResolved;
                }
            }
        }
        //如果已经解析过则使用解析好的构造函数方法不需要再次锁定
        //一个类中有多个构造函数，判断使用哪个构造函数实例化过程比较耗性能，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中。
```

```
        if (resolved) {
            if (autowireNecessary) {
                //构造函数自动注入
                return autowireConstructor(beanName, mbd, null, null);
            }
            else {
                //使用默认构造函数构造
                return instantiateBean(beanName, mbd);
            }
        }

        // 需要根据参数解析、确定构造函数
        Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass,
beanName);
    // 解析的构造器不为空 || 注入类型为构造函数自动注入 || bean定义中有构造器参数 || 传入参数不为空
        if (ctors != null ||
                mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_CONSTRUCTOR ||
                mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args))  {
            //构造函数自动注入
            return autowireConstructor(beanName, mbd, ctors, args);
        }

        // No special handling: simply use no-arg constructor.
        //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实例化。
        return instantiateBean(beanName, mbd);
    }
```

5.2.4.1.1 autowireConstructor

```
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...
                    argsToUse = mbd.resolvedConstructorArguments;
                    if (argsToUse == null) {
                        //配置的构造函数参数
                        argsToResolve = mbd.preparedConstructorArguments;
                    }
                }
            }
            //如果缓存中存在
            if (argsToResolve != null) {
                //解析参数类型，如给定方法的构造函数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 {
                //提取配置文件中配置的构造函数参数
                ConstructorArgumentValues cargs = mbd.getConstructorArgumentValues();
                //用于承载解析后的构造函数参数的值
                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);
                }
            }
            //排序给定的构造函数，public构造函数优先参数数量降序/非public构造函数参数数量降序
            AutowireUtils.sortConstructors(candidates);
            int minTypeDiffWeight = Integer.MAX_VALUE;
            Set<Constructor<?>> ambiguousConstructors = null;
            List<Exception> causes = null;

            for (int i = 0; i < candidates.length; i++) {
                Constructor<?> candidate = candidates[i];
                Class<?>[] paramTypes = candidate.getParameterTypes();

                if (constructorToUse != null && argsToUse.length > paramTypes.length) {
                    //如果已经找到选用的构造函数或者需要的参数个数小于当前的构造函数则终止，
                    //因为已经按照参数个数降序排列
                    break;
                }
                if (paramTypes.length < minNrOfArgs) {
                    //参数个数不对等
                    continue;
                }

                ArgumentsHolder argsHolder;
                if (resolvedValues != null) {
                    //有参数则根据值构造对应参数类型的参数
                    try {
                        String[] paramNames = null;
                        if (constructorPropertiesAnnotationAvailable) {
                            //注解上获取参数名称
                            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) {
                                //获取指定的构造函数的参数名称
                                paramNames = pnd.getParameterNames(candidate);
                            }
                        }
                        //根据名称和根据类型创建参数持有者
                        argsHolder = createArgumentArray(
                                beanName, mbd, resolvedValues, bw, paramTypes,
paramNames, candidate, autowiring);
                    }
                    catch (UnsatisfiedDependencyException ex) {
                        if (this.beanFactory.logger.isTraceEnabled()) {
                            this.beanFactory.logger.trace(
                                    "Ignoring constructor [" + candidate + "] of bean
'" + beanName + "': " + ex);
                        }
                        if (i == candidates.length - 1 && constructorToUse == null) {
                            if (causes != null) {
                                for (Exception cause : causes) {
                                    this.beanFactory.onSuppressedException(cause);
                                }
                            }
                            throw ex;
                        }
                        else {
                            // Swallow and try next constructor.
                            if (causes == null) {
                                causes = new LinkedList<Exception>();
                            }
                            causes.add(ex);
                            continue;
                        }
                    }
                }
                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函数存在，例如不同构造函数的参数为父子关系
                int typeDiffWeight = (mbd.isLenientConstructorResolution() ?
                        argsHolder.getTypeDifferenceWeight(paramTypes) :
argsHolder.getAssignabilityWeight(paramTypes));
                // Choose this constructor if it represents the closest match.
```

```
                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) {
                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) {
                //将解析的构造函数加入缓存
                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>() {
                    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.setWrappedInstance(beanInstance);
            return bw;
        }
        catch (Throwable ex) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
"Instantiation of bean failed", ex);
        }
    }
```

上面代码非常长，总体的功能逻辑如下：

1、确定参数。 如果调用getBean方式时传入的参数不为空，则可以直接使用传入的参数； 再尝试从缓存中获取参数否则，需要解析配置节点时，配置的构造器参数。 2、确定构造函数。根据第一步中确定下来的参数，接下来的任务就是根据参数的个数、类型来确定最终调用的构造函数。首先是根据参数个数匹配，把所有构造函数根据参数个数升序排序，再去筛选参数个数匹配的构造函数；因为配置文件中可以通过参数位置索引，也可以通过参数名称来设定参数值，如,所有还需要解析参数的名称：通过注解的方式获取；通过工具类ParameterNameDiscoverer来获取。最后，根据解析好的参数名称、参数类型、实际参数就可以确定构造函数，并且将参数转换成对应的类型 3、根据确定的构造函数转换成对应的参数类型 4、构造函数不确定性的验证。因为有一些构造函数的参数类型为父子关系，所以Spring会做一次验证 5、如果条件符合（传入参数为空），将解析好的构造函数、参数放入缓存

6、根据实例化策略将构造函数、参数实例化bean

5.2.4.1.1.1 instantiate

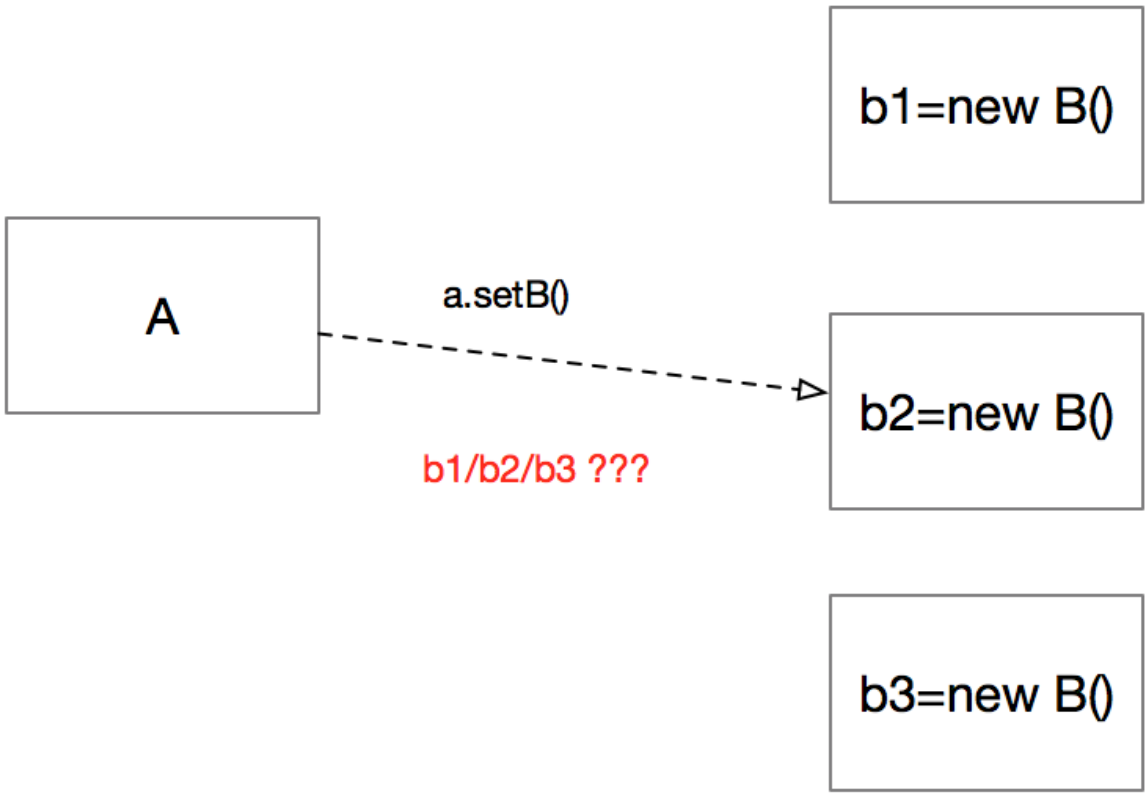
在上面创建bean的最后一步，Spring并没有通过解析好的构造函数和实参来直接创建，而是使用了策略模式来决定是使用反射还是代理的方法创建bean

```
public Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory
owner,
            final Constructor<?> ctor, @Nullable Object... args) {
        //如果不存在 lookup-override或者replace-override属性的话，直接通过构造函数和参数进行实例
化
        if (!bd.hasMethodOverrides()) {
            if (System.getSecurityManager() != null) {
                // use own privileged to change accessibility (when security is on)
                AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
                    ReflectionUtils.makeAccessible(ctor);
                    return null;
```

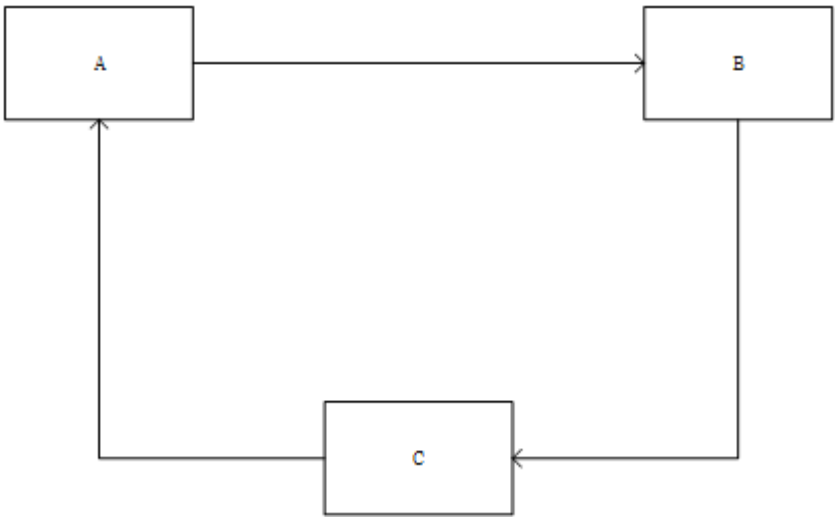
```
                });
            }
            return (args != null ? BeanUtils.instantiateClass(ctor, args) :
BeanUtils.instantiateClass(ctor));
        }
        else {
            //因为使用了lookup-override或者replace-override功能的话，就需要通过动态代理来创建
bean
            return instantiateWithMethodInjection(bd, beanName, owner, ctor, args);
        }
    }
```

5.3 循环依赖

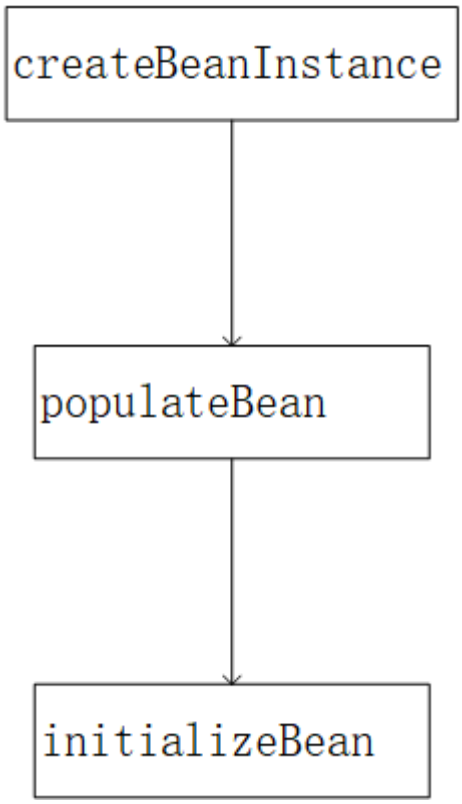
只有单例情况下，spring才会试着去解决循环依赖问题，多例是不会去解决循环依赖的。这个也好理解，如果是多例的话，比如a -> b 并且 b -> a 那么，当A a=new A(); 之后要注入b，b却是多例的，那么究竟该注入哪个B是不确定的



spring解决循环依赖主要靠上面的三级缓存机制因此spring 无法解决构造器循环依赖，spring只能解决setter循环依赖，对于解决setter循环依赖spring是通过提前曝光早期的bean实例。循环依赖其实就是循环引用，也就是两个或则两个以上的bean互相持有对方，最终形成闭环。比如A依赖于B，B依赖于C，C又依赖于A。



检测循环依赖相对比较容易，Bean在创建的时候可以给该Bean打标，如果递归调用回来发现正在创建中的话，即说明了循环依赖了。bean创建的一个简化的过程如下



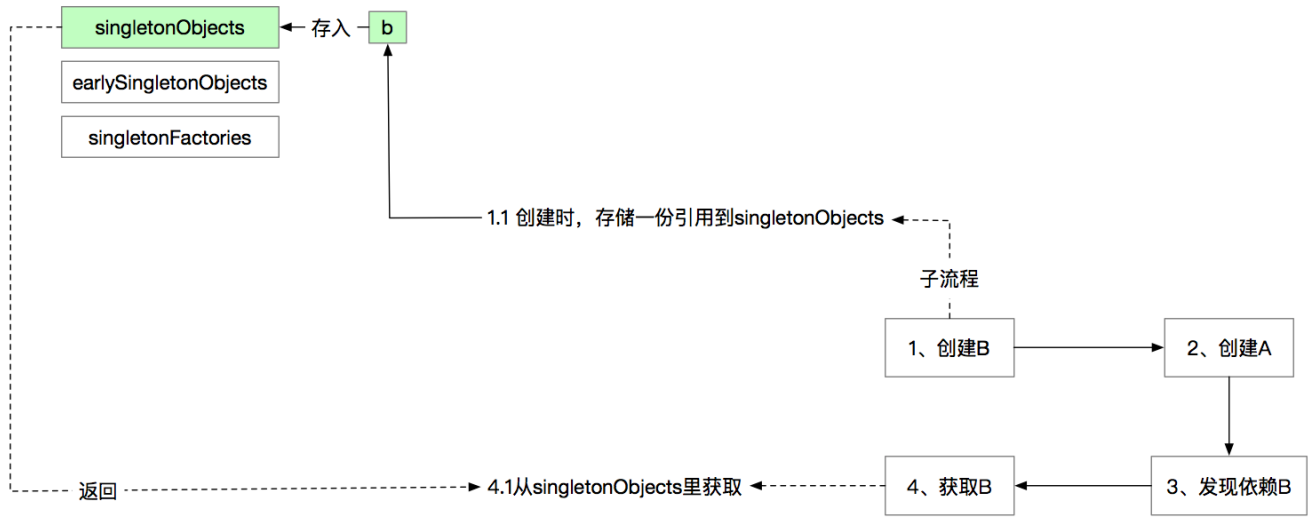
（1）createBeanInstance：实例化，其实也就是调用对象的构造方法实例化对象

（2）populateBean：填充属性，这一步主要是多bean的依赖属性进行填充

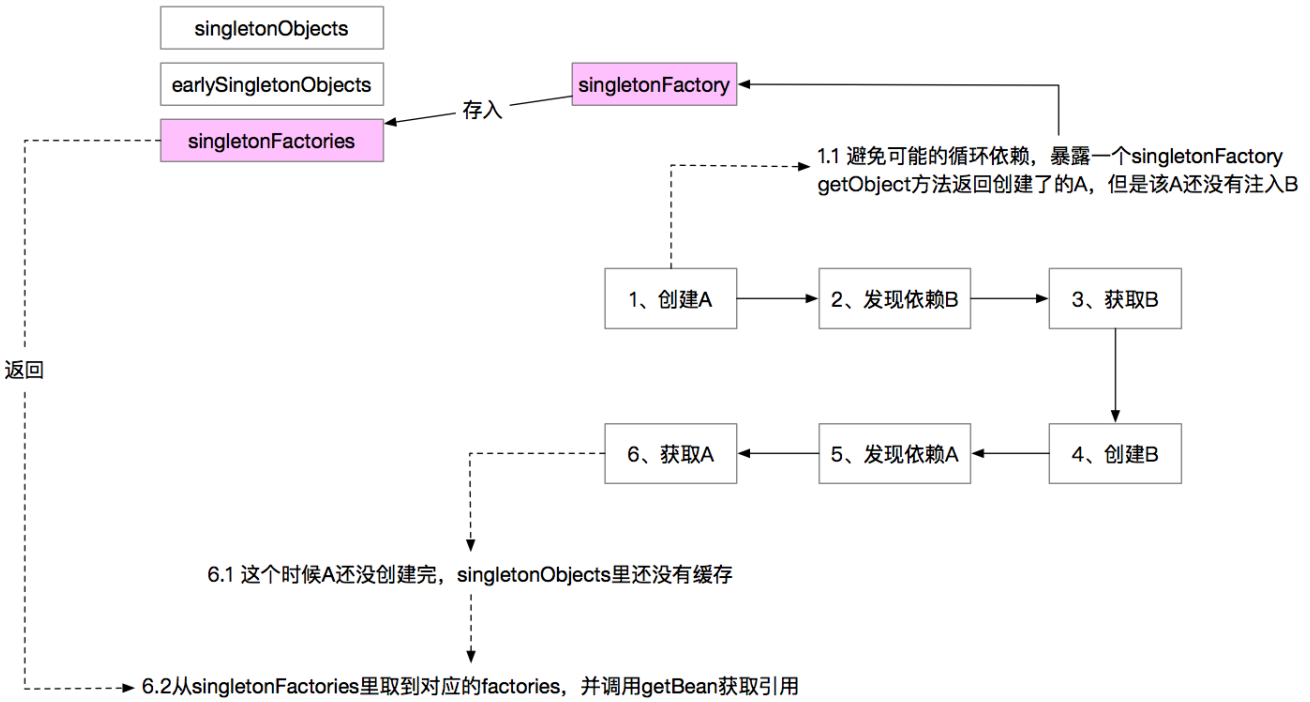
（3）initializeBean：调用spring xml中的init 方法。

从上面讲述的单例bean初始化步骤我们可以知道，循环依赖主要发生在第一、第二部分。bean创建过程中的三级缓存就是为了解决上面第二部分。

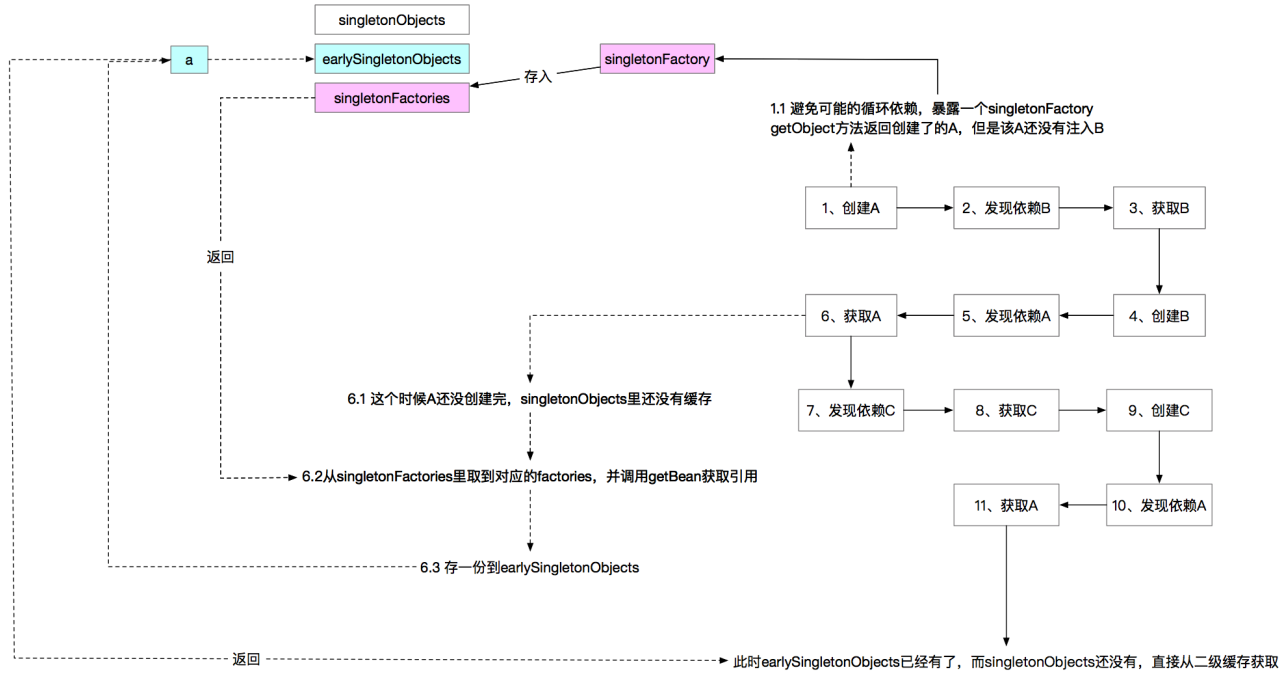
A 依赖 B（B不依赖A）



A 依赖 B && B依赖A



A依赖B && B依赖A + B依赖C && C 依赖 A



https://blog.csdn.net/u010853261/article/details/77940767

https://www.jianshu.com/p/10f94b776e55

https://blog.csdn.net/cgj296645438/article/details/80049130

https://blog.csdn.net/cgj296645438/article/category/7411638

https://blog.csdn.net/benhuo931115/article/details/74611464

https://blog.csdn.net/nuomizhende45/article/details/81158383

https://blog.csdn.net/u014252478/article/category/8339758/1?

https://www.jianshu.com/u/37f55fd33905

仿spring反转实现https://blog.csdn.net/MadCode2222222222222/article/details/78769851