# 课程目标

1、通过分析Spring 源码，深刻掌握核心原理和设计思想。
2、通过本课的学习，完全掌握Spring IOC容器的初始化细节，并手绘时序图。
3、掌握看源码不晕车的要诀。

# 内容定位

1、通过对Spring 源码的分析，掌握IOC容器的初始化流程。
2、动手绘制时序图，帮助梳理源码设计思路。看源码依旧晕车的你，必须要掌握阅读要领。

# 1. Spring 核心之IOC容器初体验

## 1.1. 再谈 IOC与DI

IOC（Inversion of Control）控制反转：所谓控制反转，就是把原先我们代码里面需要实现的对象创建、依赖的代码，反转给容器来帮忙实现。那么必然的我们需要创建一个容器，同时需要一种描述来让容器知道需要创建的对象与对象的关系。这个描述最具体表现就是我们所看到的配置文件。

DI（Dependency Injection）依赖注入：就是指对象是被动接受依赖类而不是自己主动去找，换句话说就是指对象不是从容器中查找它依赖的类，而是在容器实例化对象的时候主动将它依赖的类注入给它。

先从我们自己设计这样一个视角来考虑：
1、对象和对象的关系怎么表示？
可以用xml，properties文件等语义化配置文件表示。

2、描述对象关系的文件存放在哪里？
可能是classpath，filesystem，或者是URL网络资源，servletContext等。

回到正题，有了配置文件，还需要对配置文件解析。

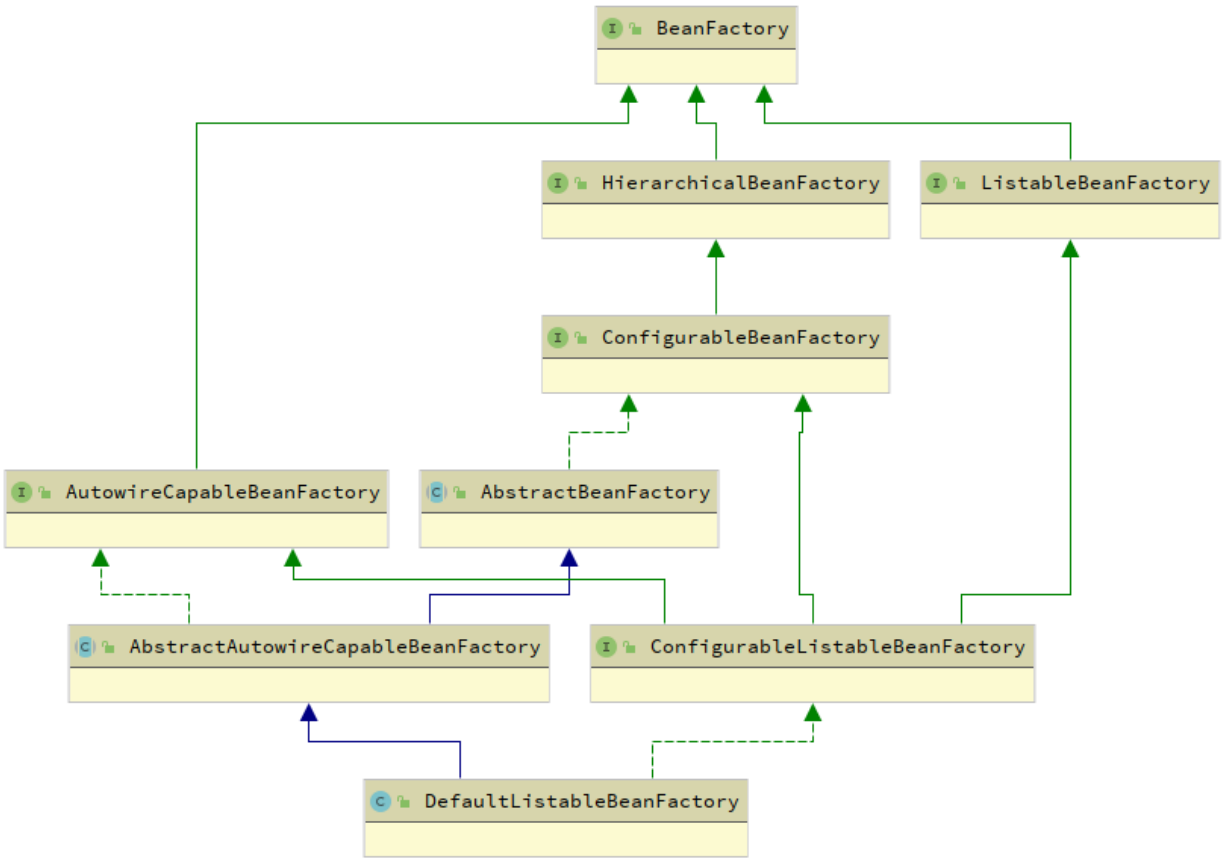
3、不同的配置文件对对象的描述不一样，如标准的，自定义声明式的，如何统一？
在内部需要有一个统一的关于对象的定义，所有外部的描述都必须转化成统一的描述定义。

4、如何对不同的配置文件进行解析？
需要对不同的配置文件语法，采用不同的解析器。

# 1.2. Spring核心容器类图

## 1.2.1. BeanFactory

Spring Bean的创建是典型的工厂模式，这一系列的Bean工厂，也即IOC容器为开发者管理对象间的依赖关系提供了很多便利和基础服务，在Spring中有许多的IOC容器的实现供用户选择和使用，其相互关系如下：




其中BeanFactory作为最顶层的一个接口类，它定义了IOC容器的基本功能规范，BeanFactory有三个重要的子类：ListableBeanFactory、HierarchicalBeanFactory 和 AutowireCapableBeanFactory。

但是从类图中我们可以发现最终的默认实现类是DefaultListableBeanFactory，它实现了所有的接口。

那为何要定义这么多层次的接口呢？查阅这些接口的源码和说明发现，每个接口都有它使用的场合，它主要是为了区分在Spring内部在操作过程中对象的传递和转化过程时，对对象的数据访问所做的限制。

例如ListableBeanFactory 接口表示这些Bean 是可列表化的，而HierarchicalBeanFactory表示的是这些Bean是有继承关系的，也就是每个Bean有可能有父Bean。AutowireCapableBeanFactory接口定义Bean的自动装配规则。这三个接口共同定义了Bean的集合、Bean之间的关系、以及Bean行为。最基本的IOC容器接口BeanFactory，来看一下它的源码：

```java
public interface BeanFactory {

    //对FactoryBean的转义定义，因为如果使用bean的名字检索FactoryBean得到的对象是工厂生成
    //如果需要得到工厂本身，需要转义
    String FACTORY_BEAN_PREFIX = "&";

    //根据bean的名字，获取在IOC容器中得到bean实例
    Object getBean(String name) throws BeansException;

    //根据bean的名字和Class类型来得到bean实例，增加了类型安全验证机制。
    <T> T getBean(String name, @Nullable Class<T> requiredType) throws BeansException;

    Object getBean(String name, Object... args) throws BeansException;

    <T> T getBean(Class<T> requiredType) throws BeansException;

    <T> T getBean(Class<T> requiredType, Object... args) throws BeansException;

    //提供对bean的检索，看看是否在IOC容器有这个名字的bean
    boolean containsBean(String name);

    //根据bean名字得到bean实例，并同时判断这个bean是不是单例
    boolean isSingleton(String name) throws NoSuchBeanDefinitionException;

    boolean isPrototype(String name) throws NoSuchBeanDefinitionException;

    boolean isTypeMatch(String name, ResolvableType typeToMatch) throws NoSuchBeanDefin

    boolean isTypeMatch(String name, @Nullable Class<?> typeToMatch) throws NoSuchBeanD

    //得到bean实例的Class类型
    @Nullable
    Class<?> getType(String name) throws NoSuchBeanDefinitionException;

    //得到bean的别名，如果根据别名检索，那么其原名也会被检索出来
    String[] getAliases(String name);

}
```

在BeanFactory里只对IOC容器的基本行为作了定义，根本不关心你的Bean是如何定义怎样加载的。

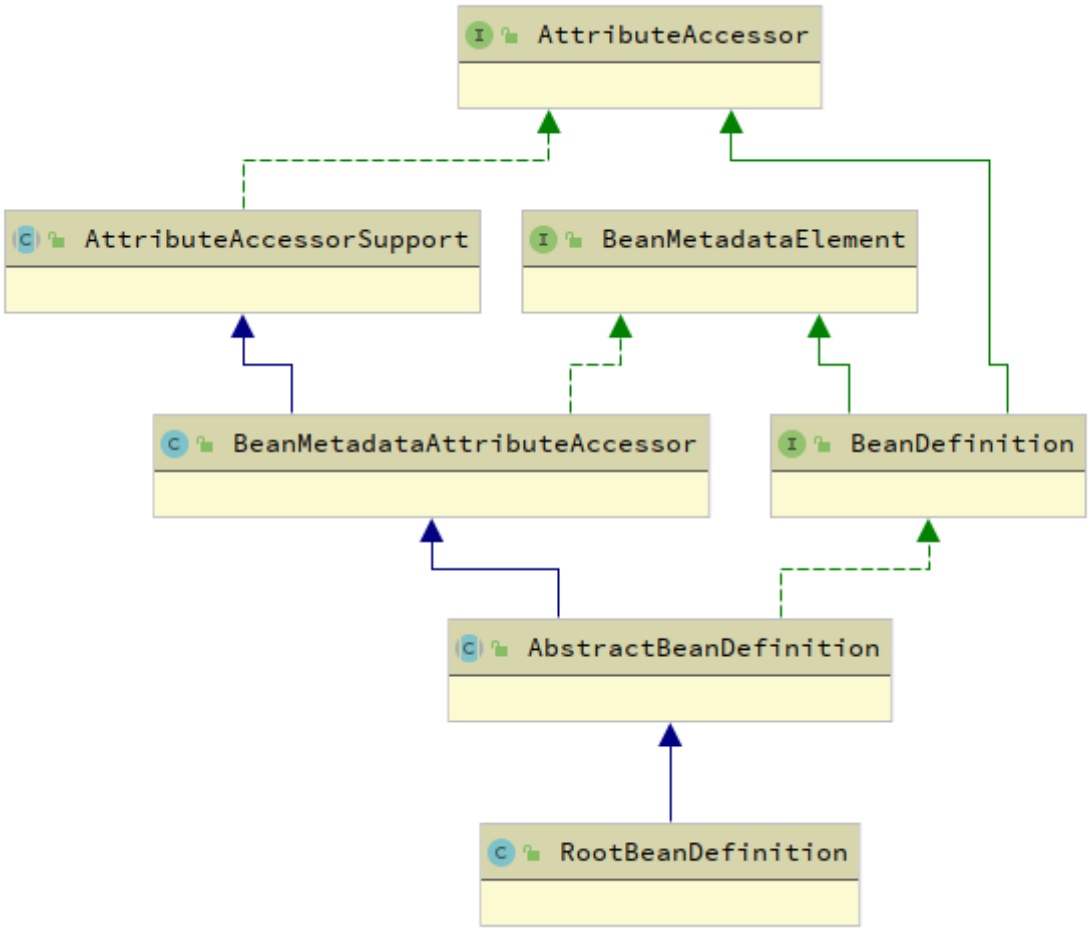
正如我们只关心工厂里得到什么的产品对象，至于工厂是怎么生产这些对象的，这个基本的接口不关心。

而要知道工厂是如何产生对象的，我们需要看具体的IOC容器实现，Spring 提供了许多IOC容器的实现。比如GenericApplicationContext，ClasspathXmlApplicationContext等。

ApplicationContext是Spring 提供的一个高级的IOC容器，它除了能够提供IOC容器的基本功能外，还为用户提供了以下的附加服务。从ApplicationContext接口的实现，我们看出其特点：
1、支持信息源，可以实现国际化。（实现MessageSource接口）
2、访问资源。（实现ResourcePatternResolver接口，后面章节会讲到）
3、支持应用事件。（实现ApplicationEventPublisher接口）

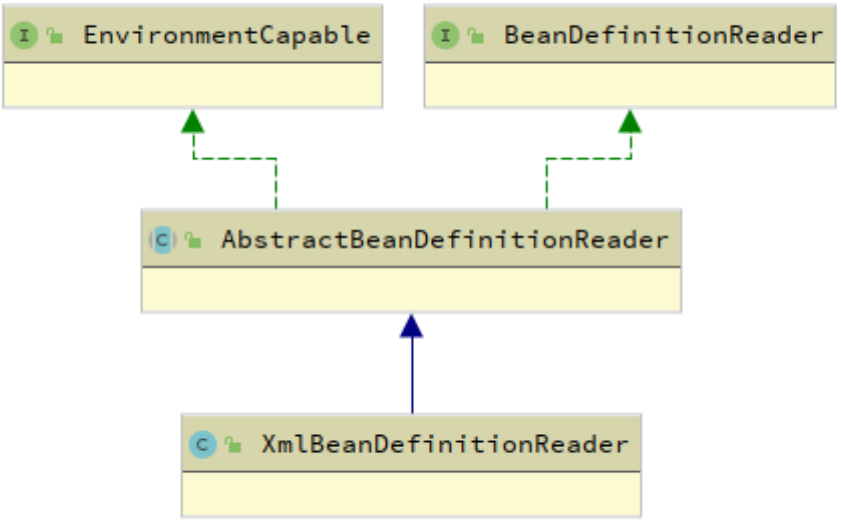
## 1.2.2. BeanDefinition

SpringIOC容器管理了我们定义的各种Bean对象及其相互的关系，Bean对象在Spring实现中是以BeanDefinition来描述的，其继承体系如下：




## 1.2.3. BeanDefinitionReader

Bean的解析过程非常复杂，功能被分的很细，因为这里需要被扩展的地方很多，必须保证有足够的灵活性，以应对可能的变化。Bean的解析主要就是对Spring配置文件的解析。这个解析过程主要通过BeanDefintionReader 来完成，最后看看Spring 中BeanDefintionReader的类结构图：




通过本章内容的分析，我们对Spring框架体系有了一个基本的宏观了解，希望小伙伴们好好理解，最好在脑海中形成画面，为以后的学习打下良好的铺垫。

## 1.3. Web IOC容器初体验

我们还是从大家最熟悉的DispatcherServlet开始，我们最先想到的还是DispatcherServlet的init()方法。我们发现在DispatherServlet中并没有找到init()方法。但是经过探索，往上追索在其父类HttpServletBean中找到了我们想要的init()方法，如下：

```
public abstract class HttpServletBean extends HttpServlet implements EnvironmentCapable
    ...

    /**
     * Map config parameters onto bean properties of this servlet, and
     * invoke subclass initialization.
     * @throws ServletException if bean properties are invalid (or required
     * properties are missing), or if subclass initialization fails.
     */
    @Override
    public final void init() throws ServletException {
        if (logger.isDebugEnabled()) {
            logger.debug("Initializing servlet '" + getServletName() + "'");
        }

        // Set bean properties from init parameters.
        PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(), this.r
        if (!pvs.isEmpty()) {
```

```
            try {
                //定位资源
                BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);
                //加载配置信息
                ResourceLoader resourceLoader = new ServletContextResourceLoader(getSer
                bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoad
                initBeanWrapper(bw);
                bw.setPropertyValues(pvs, true);
            }
            catch (BeansException ex) {
                if (logger.isErrorEnabled()) {
                    logger.error("Failed to set bean properties on servlet '" + getServ
                }
                throw ex;
            }
        }

        // Let subclasses do whatever initialization they like.
        initServletBean();

        if (logger.isDebugEnabled()) {
            logger.debug("Servlet '" + getServletName() + "' configured successfully");
        }
    }
    ...
```

在init()方法中，真正完成初始化容器动作的逻辑其实在initServletBean()方法中，我们继续跟进initServletBean()中的代码在FrameworkServlet类中：

```
public abstract class FrameworkServlet extends HttpServletBean implements ApplicationCo
    ...
    /**
     * Overridden method of {@link HttpServletBean}, invoked after any bean properties
     * have been set. Creates this servlet's WebApplicationContext.
     */
    @Override
    protected final void initServletBean() throws ServletException {
        getServletContext().log("Initializing Spring FrameworkServlet '" + getServletNa
        if (this.logger.isInfoEnabled()) {
            this.logger.info("FrameworkServlet '" + getServletName() + "': initializati
        }
```

```
        long startTime = System.currentTimeMillis();

        try {

            this.webApplicationContext = initWebApplicationContext();
            initFrameworkServlet();
        }
        catch (ServletException ex) {
            this.logger.error("Context initialization failed", ex);
            throw ex;
        }
        catch (RuntimeException ex) {
            this.logger.error("Context initialization failed", ex);
            throw ex;
        }

        if (this.logger.isInfoEnabled()) {
            long elapsedTime = System.currentTimeMillis() - startTime;
            this.logger.info("FrameworkServlet '" + getServletName() + "': initializati
                    elapsedTime + " ms");
        }
    }
    ...
```

在上面的代码中终于看到了我们似曾相识的代码initWebAppplicationContext()，继续跟进：

```
public abstract class FrameworkServlet extends HttpServletBean implements ApplicationCo
    ...
    /**
     * Initialize and publish the WebApplicationContext for this servlet.
     * <p>Delegates to {@link #createWebApplicationContext} for actual creation
     * of the context. Can be overridden in subclasses.
     * @return the WebApplicationContext instance
     * @see #FrameworkServlet(WebApplicationContext)
     * @see #setContextClass
     * @see #setContextConfigLocation
     */
    protected WebApplicationContext initWebApplicationContext() {

        //先从ServletContext中获得父容器WebAppliationContext
        WebApplicationContext rootContext =
```

```
16                  WebApplicationContextUtils.getWebApplicationContext(getServletContext()
17          //声明子容器
18          WebApplicationContext wac = null;
19
20          //建立父、子容器之间的关联关系
21          if (this.webApplicationContext != null) {
22              // A context instance was injected at construction time -> use it
23              wac = this.webApplicationContext;
24              if (wac instanceof ConfigurableWebApplicationContext) {
25                  ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationCor
26                  if (!cwac.isActive()) {
27                      // The context has not yet been refreshed -> provide services such
28                      // setting the parent context, setting the application context id,
29                      if (cwac.getParent() == null) {
30                          // The context instance was injected without an explicit parent
31                          // the root application context (if any; may be null) as the pa
32                          cwac.setParent(rootContext);
33                      }
34                      //这个方法里面调用了AbatractApplication的refresh()方法
35                      //模板方法，规定IOC初始化基本流程
36                      configureAndRefreshWebApplicationContext(cwac);
37                  }
38              }
39          }
40          //先去ServletContext中查找Web容器的引用是否存在，并创建好默认的空IOC容器
41          if (wac == null) {
42              // No context instance was injected at construction time -> see if one
43              // has been registered in the servlet context. If one exists, it is assumed
44              // that the parent context (if any) has already been set and that the
45              // user has performed any initialization such as setting the context id
46              wac = findWebApplicationContext();
47          }
48          //给上一步创建好的IOC容器赋值
49          if (wac == null) {
50              // No context instance is defined for this servlet -> create a local one
51              wac = createWebApplicationContext(rootContext);
52          }
53
54          //触发onRefresh方法
55          if (!this.refreshEventReceived) {
56              // Either the context is not a ConfigurableApplicationContext with refresh
57              // support or the context injected at construction time had already been
58              // refreshed -> trigger initial onRefresh manually here.
```

```
59              onRefresh(wac);
60          }
61
62          if (this.publishContext) {
63              // Publish the context as a servlet context attribute.
64              String attrName = getServletContextAttributeName();
65              getServletContext().setAttribute(attrName, wac);
66              if (this.logger.isDebugEnabled()) {
67                  this.logger.debug("Published WebApplicationContext of servlet '" + getS
68                          "' as ServletContext attribute with name [" + attrName + "]");
69              }
70          }
71
72          return wac;
73      }
74
75      /**
76       * Retrieve a {@code WebApplicationContext} from the {@code ServletContext}
77       * attribute with the {@link #setContextAttribute configured name}. The
78       * {@code WebApplicationContext} must have already been loaded and stored in the
79       * {@code ServletContext} before this servlet gets initialized (or invoked).
80       * <p>Subclasses may override this method to provide a different
81       * {@code WebApplicationContext} retrieval strategy.
82       * @return the WebApplicationContext for this servlet, or {@code null} if not found
83       * @see #getContextAttribute()
84       */
85      @Nullable
86      protected WebApplicationContext findWebApplicationContext() {
87          String attrName = getContextAttribute();
88          if (attrName == null) {
89              return null;
90          }
91          WebApplicationContext wac =
92                  WebApplicationContextUtils.getWebApplicationContext(getServletContext()
93          if (wac == null) {
94              throw new IllegalStateException("No WebApplicationContext found: initialize
95          }
96          return wac;
97      }
98
99      /**
100      * Instantiate the WebApplicationContext for this servlet, either a default
101      * {@link org.springframework.web.context.support.XmlWebApplicationContext}
```

```
     * or a {@link #setContextClass custom context class}, if set.
     * <p>This implementation expects custom contexts to implement the
     * {@link org.springframework.web.context.ConfigurableWebApplicationContext}
     * interface. Can be overridden in subclasses.
     * <p>Do not forget to register this servlet instance as application listener on th
     * created context (for triggering its {@link #onRefresh callback}, and to call
     * {@link org.springframework.context.ConfigurableApplicationContext#refresh()}
     * before returning the context instance.
     * @param parent the parent ApplicationContext to use, or {@code null} if none
     * @return the WebApplicationContext for this servlet
     * @see org.springframework.web.context.support.XmlWebApplicationContext
     */
    protected WebApplicationContext createWebApplicationContext(@Nullable ApplicationCo
        Class<?> contextClass = getContextClass();
        if (this.logger.isDebugEnabled()) {
            this.logger.debug("Servlet with name '" + getServletName() +
                    "' will try to create custom WebApplicationContext context of class
                    contextClass.getName() + "'" + ", using parent context [" + parent
        }
        if (!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass)) {
            throw new ApplicationContextException(
                    "Fatal initialization error in servlet with name '" + getServletNam
                    "': custom WebApplicationContext class [" + contextClass.getName()
                    "] is not of type ConfigurableWebApplicationContext");
        }
        ConfigurableWebApplicationContext wac =
                (ConfigurableWebApplicationContext) BeanUtils.instantiateClass(contextC

        wac.setEnvironment(getEnvironment());
        wac.setParent(parent);
        String configLocation = getContextConfigLocation();
        if (configLocation != null) {
            wac.setConfigLocation(configLocation);
        }
        configureAndRefreshWebApplicationContext(wac);

        return wac;
    }

    protected void configureAndRefreshWebApplicationContext(ConfigurableWebApplicationC
        if (ObjectUtils.identityToString(wac).equals(wac.getId())) {
            // The application context id is still set to its original default value
            // -> assign a more useful id based on available information
```

```
            if (this.contextId != null) {
                wac.setId(this.contextId);
            }
            else {
                // Generate default id...
                wac.setId(ConfigurableWebApplicationContext.APPLICATION_CONTEXT_ID_PREF
                        ObjectUtils.getDisplayString(getServletContext().getContextPath
            }
        }

        wac.setServletContext(getServletContext());
        wac.setServletConfig(getServletConfig());
        wac.setNamespace(getNamespace());
        wac.addApplicationListener(new SourceFilteringListener(wac, new ContextRefreshL

        // The wac environment's #initPropertySources will be called in any case when t
        // is refreshed; do it eagerly here to ensure servlet property sources are in p
        // use in any post-processing or initialization that occurs below prior to #ref
        ConfigurableEnvironment env = wac.getEnvironment();
        if (env instanceof ConfigurableWebEnvironment) {
            ((ConfigurableWebEnvironment) env).initPropertySources(getServletContext(),
        }

        postProcessWebApplicationContext(wac);
        applyInitializers(wac);
        wac.refresh();
    }
    ...
```

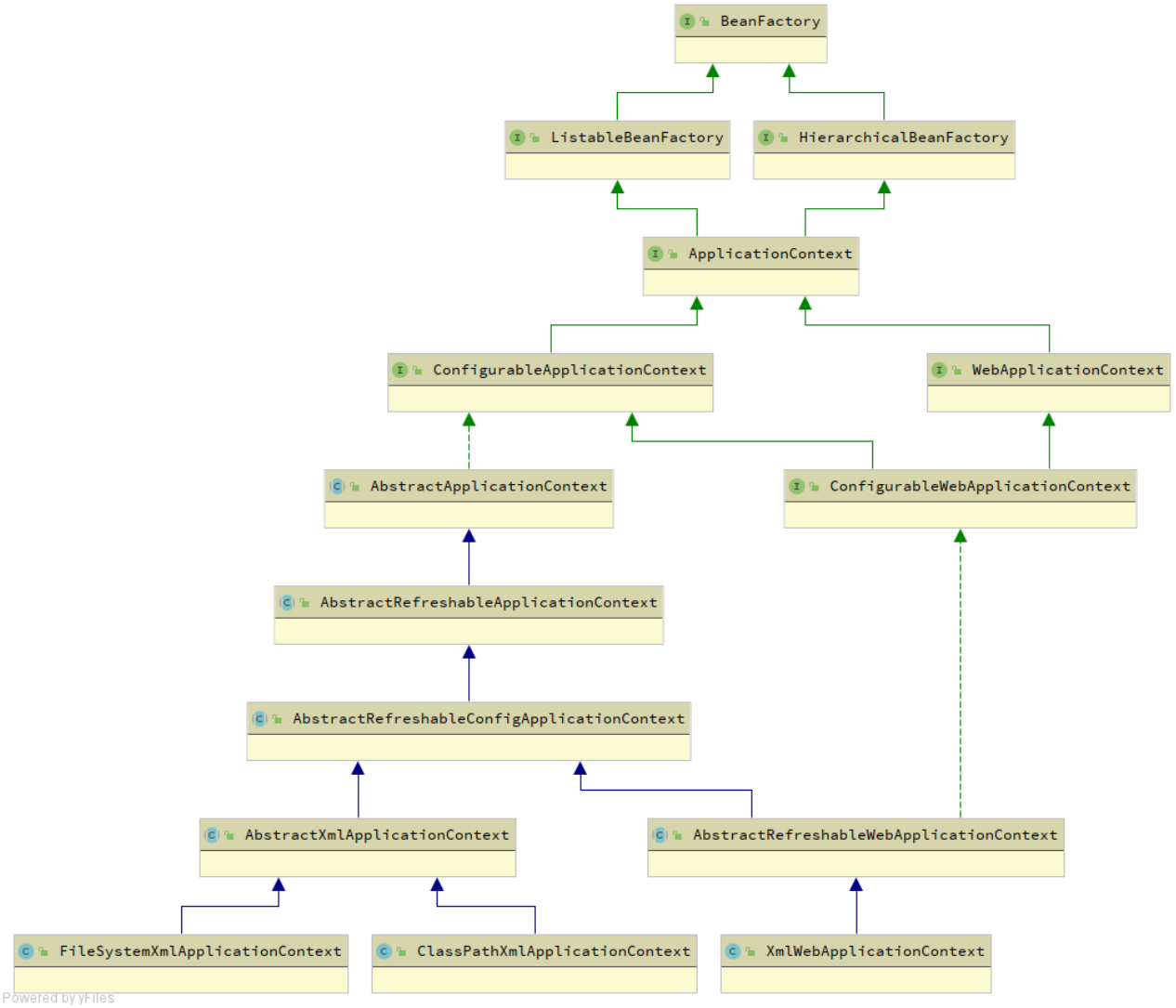
从上面的代码中可以看出，在configAndRefreshWebApplicationContext()方法中，调用refresh()方法，这个是真正启动IOC容器的入口，后面会详细介绍。IOC容器初始化以后，最后调用了DispatcherServlet的 onRefresh()方法，在onRefresh()方法中又是直接调用initStrategies()方法初始化SpringMVC的九大组件：

```
public class DispatcherServlet extends FrameworkServlet {
    ...
    @Override
    protected void onRefresh(ApplicationContext context) {
        initStrategies(context);
    }

```

```
    /**
     * Initialize the strategy objects that this servlet uses.
     * <p>May be overridden in subclasses in order to initialize further strategy objec
     */
    //初始化策略
    protected void initStrategies(ApplicationContext context) {
        //多文件上传的组件
        initMultipartResolver(context);
        //初始化本地语言环境
        initLocaleResolver(context);
        //初始化模板处理器
        initThemeResolver(context);
        //handlerMapping
        initHandlerMappings(context);
        //初始化参数适配器
        initHandlerAdapters(context);
        //初始化异常拦截器
        initHandlerExceptionResolvers(context);
        //初始化视图预处理器
        initRequestToViewNameTranslator(context);
        //初始化视图转换器
        initViewResolvers(context);
        //
        initFlashMapManager(context);
    }
    ...
```

# 2. 基于XML的IOC容器的初始化

IOC容器的初始化包括BeanDefinition的Resource定位、加载和注册这三个基本的过程。我们以ApplicationContext 为例讲解，ApplicationContext系列容器也许是我们最熟悉的，因为Web项目中使用的XmlWebApplicationContext就属于这个继承体系还有ClasspathXmlApplicationContext等，其继承体系如下图所示：




ApplicationContext允许上下文嵌套，通过保持父上下文可以维持一个上下文体系。对于Bean的查找可以在这个上下文体系中发生，首先检查当前上下文，其次是父上下文，逐级向上，这样为不同的Spring应用提供了一个共享的Bean定义环境。

## 2.1. 寻找入口

还有一个我们用的比较多的ClassPathXmlApplicationContext，通过main()方法启动：

```
ApplicationContext app = new ClassPathXmlApplicationContext("application. xml");
```

先看其构造函数的调用：

```
    /**
     * Create a new ClassPathXmlApplicationContext, loading the definitions
     * from the given XML file and automatically refreshing the context.
     * @param configLocation resource location
```

```
     * @throws BeansException if context creation failed
     */
    public ClassPathXmlApplicationContext(String configLocation) throws BeansException
        this(new String[] {configLocation}, true, null);
    }
```

其实际调用的构造函数为：

```
    /**
     * Create a new ClassPathXmlApplicationContext with the given parent,
     * loading the definitions from the given XML files.
     * @param configLocations array of resource locations
     * @param refresh whether to automatically refresh the context,
     * loading all bean definitions and creating all singletons.
     * Alternatively, call refresh manually after further configuring the context.
     * @param parent the parent context
     * @throws BeansException if context creation failed
     * @see #refresh()
     */
    public ClassPathXmlApplicationContext(
            String[] configLocations, boolean refresh, @Nullable ApplicationContext par
            throws BeansException {

        super(parent);
        setConfigLocations(configLocations);
        if (refresh) {
            refresh();
        }
    }
```

还有像AnnotationConfigApplicationContext、FileSystemXmlApplicationContext、XmlWebApplicationContext等都继承自父容器AbstractApplicationContext主要用到了装饰器模式和策略模式，最终都是调用refresh()方法。

## 2.2. 获得配置路径

通过分析ClassPathXmlApplicationContext的源代码可以知道，在创建ClassPathXmlApplicationContext 容器时，构造方法做以下两项重要工作：首先，调用父类容器的构造方法（super（parent）方法）为容器设置好Bean 资源加载器。

然后，再调用父类AbstractRefreshableConfigApplicationContext 的setConfigLocations（configLocations）方法设置Bean配置信息的定位路径。

通过追踪ClassPathXmlApplicationContext的继承体系，发现其父类的父类AbstractApplicationContext中初始化IOC容器所做的主要源码如下：

```
public abstract class AbstractApplicationContext extends DefaultResourceLoader
        implements ConfigurableApplicationContext {
    ...
    //静态初始化块，在整个容器创建过程中只执行一次
    static {
        // Eagerly load the ContextClosedEvent class to avoid weird classloader issues
        // on application shutdown in WebLogic 8.1. (Reported by Dustin Woods.)
        //为了避免应用程序在Weblogic8.1关闭时出现类加载异常加载问题，加载IoC容
        //器关闭事件(ContextClosedEvent)类
        ContextClosedEvent.class.getName();
    }

    /**
     * Create a new AbstractApplicationContext with no parent.
     */
    public AbstractApplicationContext() {
        this.resourcePatternResolver = getResourcePatternResolver();
    }

    /**
     * Create a new AbstractApplicationContext with the given parent context.
     * @param parent the parent context
     */
    public AbstractApplicationContext(@Nullable ApplicationContext parent) {
        this();
        setParent(parent);
    }

    /**
     * Return the ResourcePatternResolver to use for resolving location patterns
     * into Resource instances. Default is a
     * {@link org.springframework.core.io.support.PathMatchingResourcePatternResolver},
     * supporting Ant-style location patterns.
     * <p>Can be overridden in subclasses, for extended resolution strategies,
     * for example in a web environment.
     * <p><b>Do not call this when needing to resolve a location pattern.</b>
     * Call the context's {@code getResources} method instead, which
```

```
     * will delegate to the ResourcePatternResolver.
     * @return the ResourcePatternResolver for this context
     * @see #getResources
     * @see org.springframework.core.io.support.PathMatchingResourcePatternResolver
     */
    //获取一个Spring Source的加载器用于读入Spring Bean定义资源文件
    protected ResourcePatternResolver getResourcePatternResolver() {
        //AbstractApplicationContext继承DefaultResourceLoader，因此也是一个资源加载器
        //Spring资源加载器，其getResource(String location)方法用于载入资源
        return new PathMatchingResourcePatternResolver(this);
    }
    ...
```

AbstractApplicationContext的默认构造方法中有调用PathMatchingResourcePatternResolver的构造方法创建 Spring资源加载器：

```
    public PathMatchingResourcePatternResolver(ResourceLoader resourceLoader) {
        Assert.notNull(resourceLoader, "ResourceLoader must not be null");
        //设置Spring的资源加载器
        this.resourceLoader = resourceLoader;
    }
```

在设置容器的资源加载器之后，接下来ClassPathXmlApplicationContext执行setConfigLocations()方法通过调用其父类AbstractRefreshableConfigApplicationContext的方法进行对Bean配置信息的定位，该方法的源码如下：

```
    /**
     * Set the config locations for this application context in init-param style,
     * i.e. with distinct locations separated by commas, semicolons or whitespace.
     * <p>If not set, the implementation may use a default as appropriate.
     */
    //处理单个资源文件路径为一个字符串的情况
    public void setConfigLocation(String location) {
        //String CONFIG_LOCATION_DELIMITERS = ",; /t/n";
        //即多个资源文件路径之间用” ,; \t\n”分隔，解析成数组形式
        setConfigLocations(StringUtils.tokenizeToStringArray(location, CONFIG_LOCATION_
    }

    /**
     * Set the config locations for this application context.
```

```
     * <p>If not set, the implementation may use a default as appropriate.
     */
    //解析Bean定义资源文件的路径，处理多个资源文件字符串数组
    public void setConfigLocations(@Nullable String... locations) {
        if (locations != null) {
            Assert.noNullElements(locations, "Config locations must not be null");
            this.configLocations = new String[locations.length];
            for (int i = 0; i < locations.length; i++) {
                // resolvePath为同一个类中将字符串解析为路径的方法
                this.configLocations[i] = resolvePath(locations[i]).trim();
            }
        }
        else {
            this.configLocations = null;
        }
    }
```

通过这两个方法的源码我们可以看出，我们既可以使用一个字符串来配置多个Spring Bean配置信息，也可以使用字符串数组，即下面两种方式都是可以的：

```
ClassPathResource res = new ClassPathResource("a.xml, b.xml");
```

多个资源文件路径之间可以是用”,; \t\n”等分隔。

```
classPathResource res = new ClassPathResource(new String[]{"a.xml","b.xml"});
```

至此，SpringIOC容器在初始化时将配置的Bean配置信息定位为Spring 封装的Resource。

## 2.3. 开始启动

SpringIOC容器对Bean配置资源的载入是从refresh()函数开始的，refresh()是一个模板方法，规定了IOC容器的启动流程，有些逻辑要交给其子类去实现。它对Bean配置资源进行载入ClassPathXmlApplicationContext 通过调用其父类AbstractApplicationContext的refresh()函数启动整个IOC容器对Bean定义的载入过程，现在我们来详细看看refresh()中的逻辑处理：

```
    @Override
    public void refresh() throws BeansException, IllegalStateException {
        synchronized (this.startupShutdownMonitor) {
```

```
            // Prepare this context for refreshing.
            //1、调用容器准备刷新的方法，获取容器的当时时间，同时给容器设置同步标识
            prepareRefresh();

            // Tell the subclass to refresh the internal bean factory.
            //2、告诉子类启动refreshBeanFactory()方法，Bean定义资源文件的载入从
            //子类的refreshBeanFactory()方法启动
            ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

            // Prepare the bean factory for use in this context.
            //3、为BeanFactory配置容器特性，例如类加载器、事件处理器等
            prepareBeanFactory(beanFactory);

            try {
                // Allows post-processing of the bean factory in context subclasses.
                //4、为容器的某些子类指定特殊的BeanPost事件处理器
                postProcessBeanFactory(beanFactory);

                // Invoke factory processors registered as beans in the context.
                //5、调用所有注册的BeanFactoryPostProcessor的Bean
                invokeBeanFactoryPostProcessors(beanFactory);

                // Register bean processors that intercept bean creation.
                //6、为BeanFactory注册BeanPost事件处理器.
                //BeanPostProcessor是Bean后置处理器，用于监听容器触发的事件
                registerBeanPostProcessors(beanFactory);

                // Initialize message source for this context.
                //7、初始化信息源，和国际化相关.
                initMessageSource();

                // Initialize event multicaster for this context.
                //8、初始化容器事件传播器.
                initApplicationEventMulticaster();

                // Initialize other special beans in specific context subclasses.
                //9、调用子类的某些特殊Bean初始化方法
                onRefresh();

                // Check for listener beans and register them.
                //10、为事件传播器注册事件监听器.
                registerListeners();

```

```
                // Instantiate all remaining (non-lazy-init) singletons.
                //11、初始化所有剩余的单例Bean
                finishBeanFactoryInitialization(beanFactory);

                // Last step: publish corresponding event.
                //12、初始化容器的生命周期事件处理器，并发布容器的生命周期事件
                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.
                //13、销毁已创建的Bean
                destroyBeans();

                // Reset 'active' flag.
                //14、取消refresh操作，重置容器的同步标识。
                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...
                //15、重设公共缓存
                resetCommonCaches();
            }
        }
    }
```

refresh()方法主要为IOC容器Bean的生命周期管理提供条件，Spring IOC容器载入Bean配置信息从其子类容器的refreshBeanFactory()方法启动，所以整个refresh()中 `ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();` 这句以后代码的都是注册容器的信息源和生命周期事件，我们前面说的载入就是从这句代码开始启动。

refresh()方法的主要作用是：在创建IOC容器前，如果已经有容器存在，则需要把已有的容器销毁和关闭，以保证在refresh之后使用的是新建立起来的IOC容器。它类似于对IOC容器的重启，在新建立

好的容器中对容器进行初始化，对Bean 配置资源进行载入。

## 2.4. 创建容器

obtainFreshBeanFactory()方法调用子类容器的refreshBeanFactory()方法，启动容器载入Bean配置信息的过程，代码如下：

```
protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
    //这里使用了委派设计模式，父类定义了抽象的refreshBeanFactory()方法，具体实现调用
    refreshBeanFactory();
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    if (logger.isDebugEnabled()) {
        logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
    }
    return beanFactory;
}
```

AbstractApplicationContext类中只抽象定义了refreshBeanFactory（ ）方法，容器真正调用的是其子类AbstractRefreshableApplicationContext 实现的refreshBeanFactory（ ）方法，方法的源码如下：

```
/**
 * This implementation performs an actual refresh of this context's underlying
 * bean factory, shutting down the previous bean factory (if any) and
 * initializing a fresh bean factory for the next phase of the context's lifecycle.
 */
@Override
protected final void refreshBeanFactory() throws BeansException {
    //如果已经有容器，销毁容器中的bean，关闭容器
    if (hasBeanFactory()) {
        destroyBeans();
        closeBeanFactory();
    }
    try {
        //创建IOC容器
        DefaultListableBeanFactory beanFactory = createBeanFactory();
        beanFactory.setSerializationId(getId());
        //对IOC容器进行定制化，如设置启动参数，开启注解的自动装配等
        customizeBeanFactory(beanFactory);
        //调用载入Bean定义的方法，主要这里又使用了一个委派模式，在当前类中只定义了抽
        loadBeanDefinitions(beanFactory);
```

```
            synchronized (this.beanFactoryMonitor) {
                this.beanFactory = beanFactory;
            }
        }
        catch (IOException ex) {
            throw new ApplicationContextException("I/O error parsing bean definition so
        }
    }
```

在这个方法中，先判断BeanFactory是否存在，如果存在则先销毁beans 并关闭 beanFactory，接着创建 DefaultListableBeanFactory，并调用 loadBeanDefinitions（beanFactory）装载 bean定义。

## 2.5. 载入配置路径

AbstractRefreshableApplicationContext中只定义了抽象的loadBeanDefinitions 方法，容器真正调用的是其子类AbstractXmlApplicationContext 对该方法的实现，
AbstractXmlApplicationContext的主要源码如下：

loadBeanDefinitions()方法同样是抽象方法，是由其子类实现的，也即在AbstractXmlApplicationContext中。

```
public abstract class AbstractXmlApplicationContext extends AbstractRefreshableConfigAp
    ...
    /**
     * Loads the bean definitions via an XmlBeanDefinitionReader.
     * @see org.springframework.beans.factory.xml.XmlBeanDefinitionReader
     * @see #initBeanDefinitionReader
     * @see #loadBeanDefinitions
     */
    //实现父类抽象的载入Bean定义方法
    @Override
    protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws B
        // Create a new XmlBeanDefinitionReader for the given BeanFactory.
        //创建XmlBeanDefinitionReader，即创建Bean读取器，并通过回调设置到容器中去，容
        XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(bean

        // Configure the bean definition reader with this context's
        // resource loading environment.
        //为Bean读取器设置Spring资源加载器，AbstractXmlApplicationContext的
        //祖先父类AbstractApplicationContext继承DefaultResourceLoader，因此，容器本身也
        beanDefinitionReader.setEnvironment(this.getEnvironment());
```

```
21          beanDefinitionReader.setResourceLoader(this);
22          //为Bean读取器设置SAX xml解析器
23          beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));
24
25          // Allow a subclass to provide custom initialization of the reader,
26          // then proceed with actually loading the bean definitions.
27          //当Bean读取器读取Bean定义的Xml资源文件时，启用Xml的校验机制
28          initBeanDefinitionReader(beanDefinitionReader);
29          //Bean读取器真正实现加载的方法
30          loadBeanDefinitions(beanDefinitionReader);
31      }
32
33      /**
34       * Initialize the bean definition reader used for loading the bean
35       * definitions of this context. Default implementation is empty.
36       * <p>Can be overridden in subclasses, e.g. for turning off XML validation
37       * or using a different XmlBeanDefinitionParser implementation.
38       * @param reader the bean definition reader used by this context
39       * @see org.springframework.beans.factory.xml.XmlBeanDefinitionReader#setDocumentRe
40       */
41      protected void initBeanDefinitionReader(XmlBeanDefinitionReader reader) {
42          reader.setValidating(this.validating);
43      }
44
45      /**
46       * Load the bean definitions with the given XmlBeanDefinitionReader.
47       * <p>The lifecycle of the bean factory is handled by the {@link #refreshBeanFactor
48       * method; hence this method is just supposed to load and/or register bean definiti
49       * @param reader the XmlBeanDefinitionReader to use
50       * @throws BeansException in case of bean registration errors
51       * @throws IOException if the required XML document isn't found
52       * @see #refreshBeanFactory
53       * @see #getConfigLocations
54       * @see #getResources
55       * @see #getResourcePatternResolver
56       */
57      //Xml Bean读取器加载Bean定义资源
58      protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansExce
59          //获取Bean定义资源的定位
60          Resource[] configResources = getConfigResources();
61          if (configResources != null) {
62              //Xml Bean读取器调用其父类AbstractBeanDefinitionReader读取定位
63              //的Bean定义资源
```

```
            reader.loadBeanDefinitions(configResources);
        }
        //如果子类中获取的Bean定义资源定位为空，则获取FileSystemXmlApplicationContext构
        String[] configLocations = getConfigLocations();
        if (configLocations != null) {
            //Xml Bean读取器调用其父类AbstractBeanDefinitionReader读取定位
            //的Bean定义资源
            reader.loadBeanDefinitions(configLocations);
        }
    }

    /**
     * Return an array of Resource objects, referring to the XML bean definition
     * files that this context should be built with.
     * <p>The default implementation returns {@code null}. Subclasses can override
     * this to provide pre-built Resource objects rather than location Strings.
     * @return an array of Resource objects, or {@code null} if none
     * @see #getConfigLocations()
     */
    //这里又使用了一个委托模式，调用子类的获取Bean定义资源定位的方法
    //该方法在ClassPathXmlApplicationContext中进行实现，对于我们
    //举例分析源码的FileSystemXmlApplicationContext没有使用该方法
    @Nullable
    protected Resource[] getConfigResources() {
        return null;
    }
```

以XmlBean 读取器的其中一种策略XmlBeanDefinitionReader为例XmlBeanDefinitionReader调用其父类AbstractBeanDefinitionReader的reader.loadBeanDefinitions()方法读取Bean配置资源。

由于我们使用ClassPathXmlApplicationContext作为例子分析，因此getConfigResources的返回值为null，因此程序执行reader.loadBeanDefinitions（configLocations）分支。

## 2.6. 分配路径处理策略

在XmlBeanDefinitionReader的抽象父类AbstractBeanDefinitionReader中定义了载入过程。

AbstractBeanDefinitionReader的loadBeanDefinitions()方法源码如下：

```

public abstract class AbstractBeanDefinitionReader implements EnvironmentCapable, BeanD
    //重载方法，调用下面的loadBeanDefinitions(String, Set<Resource>);方法
    @Override
```

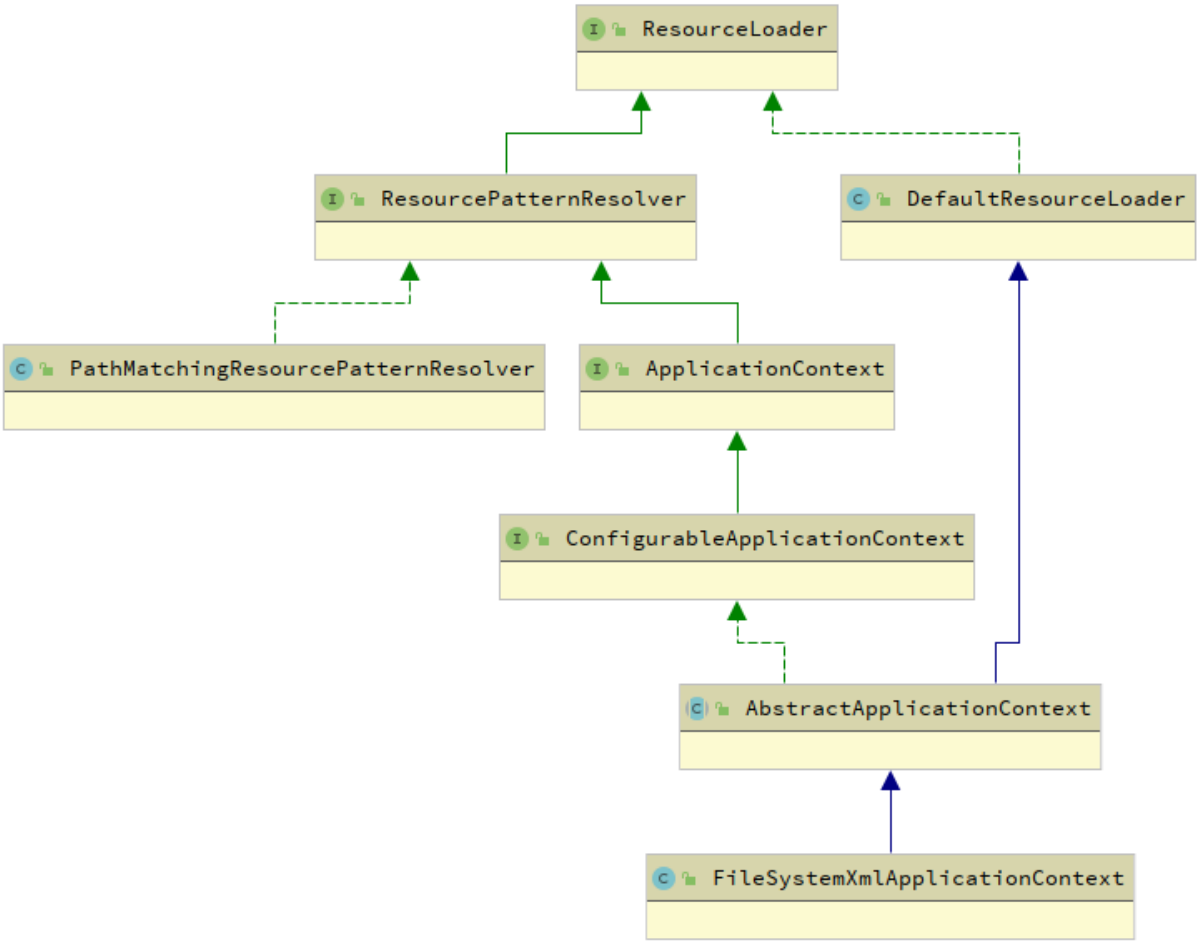
```
    public int loadBeanDefinitions(String location) throws BeanDefinitionStoreException
        return loadBeanDefinitions(location, null);
    }

    public int loadBeanDefinitions(String location, @Nullable Set<Resource> actualResou
        //获取在IoC容器初始化过程中设置的资源加载器
        ResourceLoader resourceLoader = getResourceLoader();
        if (resourceLoader == null) {
            throw new BeanDefinitionStoreException(
                    "Cannot import bean definitions from location [" + location + "]: n
        }

        if (resourceLoader instanceof ResourcePatternResolver) {
            // Resource pattern matching available.
            try {
                //将指定位置的Bean定义资源文件解析为Spring IOC容器封装的资源
                //加载多个指定位置的Bean定义资源文件
                Resource[] resources = ((ResourcePatternResolver) resourceLoader).getRe
                //委派调用其子类XmlBeanDefinitionReader的方法，实现加载功能
                int loadCount = loadBeanDefinitions(resources);
                if (actualResources != null) {
                    for (Resource resource : resources) {
                        actualResources.add(resource);
                    }
                }
                if (logger.isDebugEnabled()) {
                    logger.debug("Loaded " + loadCount + " bean definitions from locati
                }
                return loadCount;
            }
            catch (IOException ex) {
                throw new BeanDefinitionStoreException(
                        "Could not resolve bean definition resource pattern [" + locati
            }
        }
        else {
            // Can only load single resources by absolute URL.
            //将指定位置的Bean定义资源文件解析为Spring IOC容器封装的资源
            //加载单个指定位置的Bean定义资源文件
            Resource resource = resourceLoader.getResource(location);
            //委派调用其子类XmlBeanDefinitionReader的方法，实现加载功能
            int loadCount = loadBeanDefinitions(resource);
            if (actualResources != null) {
```

```
                actualResources.add(resource);
            }
            if (logger.isDebugEnabled()) {
                logger.debug("Loaded " + loadCount + " bean definitions from location [
            }
            return loadCount;
        }
    }


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

AbstractRefreshableConfigApplicationContext的loadBeanDefinitions（Resource..resources）方法实际上是调用AbstractBeanDefinitionReader的loadBeanDefinitions()方法。

从对AbstractBeanDefinitionReader的loadBeanDefinitions()方法源码分析可以看出该方法就做了两件事：

首先，调用资源加载器的获取资源方法resourceLoader.getResource（location）获取到要加载的资源。其次，真正执行加载功能是其子类XmlBeanDefinitionReader的loadBeanDefinitions()方法。在loadBeanDefinitions()方法中调用了AbstractApplicationContext的 getResources()方法，跟进去之后发现getResources()方法其实定义在ResourcePatternResolver中，此时，我们有必要来看一下ResourcePatternResolver的全类图：




从上面可以看到ResourceLoader与ApplicationContext的继承关系，可以看出其实际调用的是DefaultResourceloader 中的getSource()方法定位Resource，因为ClassPathXmlApplicationContext 本身就是DefaultResourceLoader的实现类，所以此时又回到了ClassPathXmlApplicationContext中来。

## 2.7. 解析配置文件路径

XmlBeanDefinitionReader 通过调用ClassPathXmlApplicationContext的父类DefaultResourceLoader的 getResource()方法获取要加载的资源，其源码如下

```
    //获取Resource的具体实现方法
    @Override
    public Resource getResource(String location) {
        Assert.notNull(location, "Location must not be null");

        for (ProtocolResolver protocolResolver : this.protocolResolvers) {
            Resource resource = protocolResolver.resolve(location, this);
            if (resource != null) {
                return resource;
            }
```

```
        }
        //如果是类路径的方式，那需要使用ClassPathResource 来得到bean 文件的资源对象
        if (location.startsWith("/")) {
            return getResourceByPath(location);
        }
        else if (location.startsWith(CLASSPATH_URL_PREFIX)) {
            return new ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.length
        }
        else {
            try {
                // Try to parse the location as a URL...
                // 如果是URL 方式，使用UrlResource 作为bean 文件的资源对象
                URL url = new URL(location);
                return (ResourceUtils.isFileURL(url) ? new FileUrlResource(url) : new U
            }
            catch (MalformedURLException ex) {
                // No URL -> resolve as resource path.
                //如果既不是classpath标识，又不是URL标识的Resource定位，则调用
                //容器本身的getResourceByPath方法获取Resource
                return getResourceByPath(location);
            }
        }
    }
```

DefaultResourceloader 提供了 getResourceByPath()方法的实现，就是为了处理既不是classpath标识，又不是URL标识的Resource定位这种情况。

```
    protected Resource getResourceByPath(String path) {
        return new ClassPathContextResource(path, getClassLoader());
    }
```

在ClassPathResource中完成了对整个路径的解析。这样，就可以从类路径上对IOC配置文件进行加载，当然我们可以按照这个逻辑从任何地方加载，在Spring中我们看到它提供的各种资源抽象，比如ClassPathResource、URLResource、FileSystemResource 等来供我们使用。上面我们看到的是定位Resource的一个过程，而这只是加载过程的一部分。例如FileSystemXmlApplication 容器就重写了getResourceByPath()方法：

```
    @Override
    protected Resource getResourceByPath(String path) {
        if (path.startsWith("/")) {
```

```
            path = path.substring(1);
        }
        //这里使用文件系统资源对象来定义bean 文件
        return new FileSystemResource(path);
    }
```

通过子类的覆盖，巧妙地完成了将类路径变为文件路径的转换。

## 2.8. 开始读取配置内容

继续回到XmlBeanDefinitionReader的loadBeanDefinitions（Resource..）方法看到代表bean文件的资源定义以后的载入过程。

```
    //XmlBeanDefinitionReader加载资源的入口方法
    @Override
    public int loadBeanDefinitions(Resource resource) throws BeanDefinitionStoreExcepti
        //将读入的XML资源进行特殊编码处理
        return loadBeanDefinitions(new EncodedResource(resource));
    }

    /**
     * Load bean definitions from the specified XML file.
     * @param encodedResource the resource descriptor for the XML file,
     * allowing to specify an encoding to use for parsing the file
     * @return the number of bean definitions found
     * @throws BeanDefinitionStoreException in case of loading or parsing errors
     */
    //这里是载入XML形式Bean定义资源文件方法
    public int loadBeanDefinitions(EncodedResource encodedResource) throws BeanDefiniti
        ...
        try {
            //将资源文件转为InputStream的IO流
            InputStream inputStream = encodedResource.getResource().getInputStream();
            try {
                //从InputStream中得到XML的解析源
                InputSource inputSource = new InputSource(inputStream);
                if (encodedResource.getEncoding() != null) {
                    inputSource.setEncoding(encodedResource.getEncoding());
                }
                //这里是具体的读取过程
                return doLoadBeanDefinitions(inputSource, encodedResource.getResource()
            }
```

```
            ...
    }

    //从特定XML文件中实际载入Bean定义资源的方法
    protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
            throws BeanDefinitionStoreException {
        try {
            //将XML文件转换为DOM对象，解析过程由documentLoader实现
            Document doc = doLoadDocument(inputSource, resource);
            //这里是启动对Bean定义解析的详细过程，该解析过程会用到Spring的Bean配置规则
            return registerBeanDefinitions(doc, resource);
        }
        ...
    }
```

通过源码分析，载入Bean配置信息的最后一步是将Bean配置信息转换为Document对象，该过程由documentloader()方法实现。

## 2.9. 准备文档对象

DocumentLoader 将Bean配置资源转换成Document对象的源码如下：

```
    //使用标准的JAXP将载入的Bean定义资源转换成document对象
    @Override
    public Document loadDocument(InputSource inputSource, EntityResolver entityResolver
            ErrorHandler errorHandler, int validationMode, boolean namespaceAware) thro

        //创建文件解析器工厂
        DocumentBuilderFactory factory = createDocumentBuilderFactory(validationMode, n
        if (logger.isDebugEnabled()) {
            logger.debug("Using JAXP provider [" + factory.getClass().getName() + "]");
        }
        //创建文档解析器
        DocumentBuilder builder = createDocumentBuilder(factory, entityResolver, errorH
        //解析Spring的Bean定义资源
        return builder.parse(inputSource);
    }

    /**
     * Create the {@link DocumentBuilderFactory} instance.
     * @param validationMode the type of validation: {@link XmlValidationModeDetector#V
     * or {@link XmlValidationModeDetector#VALIDATION_XSD XSD})
```

```
     * @param namespaceAware whether the returned factory is to provide support for XML
     * @return the JAXP DocumentBuilderFactory
     * @throws ParserConfigurationException if we failed to build a proper DocumentBuil
     */
    protected DocumentBuilderFactory createDocumentBuilderFactory(int validationMode, b
            throws ParserConfigurationException {

        //创建文档解析工厂
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setNamespaceAware(namespaceAware);

        //设置解析XML的校验
        if (validationMode != XmlValidationModeDetector.VALIDATION_NONE) {
            factory.setValidating(true);
            if (validationMode == XmlValidationModeDetector.VALIDATION_XSD) {
                // Enforce namespace aware for XSD...
                factory.setNamespaceAware(true);
                try {
                    factory.setAttribute(SCHEMA_LANGUAGE_ATTRIBUTE, XSD_SCHEMA_LANGUAGE
                }
                catch (IllegalArgumentException ex) {
                    ParserConfigurationException pcex = new ParserConfigurationExceptio
                            "Unable to validate using XSD: Your JAXP provider [" + fact
                            "] does not support XML Schema. Are you running on Java 1.4
                            "Upgrade to Apache Xerces (or Java 1.5) for full XSD suppor
                    pcex.initCause(ex);
                    throw pcex;
                }
            }
        }

        return factory;
    }
```

上面的解析过程是调用JavaEE标准的JAXP标准进行处理。至此Spring IoC容器根据定位的Bean配置信息，将其加载读入并转换成为Document对象过程完成。接下来我们要继续分析Spring IOC容器将载入的Bean配置信息转换为Document对象之后，是如何将其解析为Spring IOC管理的Bean对象并将其注册到容器中的。

## 2.10. 分配解析策略

XmlBeanDefinitionReader类中的doLoadBeanDefinition（ ）方法是从特定XML文件中实际载入Bean配置资源的方法，该方法在载入Bean配置资源之后将其转换为Document对象，接下来调用

registerBeanDefinitions（）启动Spring IOC容器对Bean定义的解析过程，
registerBeanDefinitions（）方法源码如下：

```
    //按照Spring的Bean语义要求将Bean定义资源解析并转换为容器内部数据结构
    public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefi
        //得到BeanDefinitionDocumentReader来对xml格式的BeanDefinition解析
        BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReade
        //获得容器中注册的Bean数量
        int countBefore = getRegistry().getBeanDefinitionCount();
        //解析过程入口，这里使用了委派模式，BeanDefinitionDocumentReader只是个接口，
        //具体的解析实现过程有实现类DefaultBeanDefinitionDocumentReader完成
        documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
        //统计解析的Bean数量
        return getRegistry().getBeanDefinitionCount() - countBefore;
    }
```

Bean配置资源的载入解析分为以下两个过程：

首先，通过调用XML解析器将Bean配置信息转换得到Document对象，但是这些Document对象并没有按照Spring的Bean 规则进行解析。这一步是载入的过程

其次，在完成通用的XML解析之后，按照Spring Bean的定义规则对Document 对象进行解析，其解析过程是在接口BeanDefinitionDocumentReader的实现类DefaultBeanDefinitionDocumentReader中实现。

## 2.11. 将配置载入内存

BeanDefinitionDocumentReader 接口通过registerBeanDefinitions()方法调用其实现类DefaultBeanDefinitionDocumentReader对Document对象进行解析，解析的代码如下：

```
    //根据Spring DTD对Bean的定义规则解析Bean定义Document对象
    @Override
    public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {
        //获得XML描述符
        this.readerContext = readerContext;
        logger.debug("Loading bean definitions");
        //获得Document的根元素
        Element root = doc.getDocumentElement();
        doRegisterBeanDefinitions(root);
    }

    /**
```

```
     * Register each bean definition within the given root {@code <beans/>} element.
     */
    protected void doRegisterBeanDefinitions(Element root) {
        // Any nested <beans> elements will cause recursion in this method. In
        // order to propagate and preserve <beans> default-* attributes correctly,
        // keep track of the current (parent) delegate, which may be null. Create
        // the new (child) delegate with a reference to the parent for fallback purpose
        // then ultimately reset this.delegate back to its original (parent) reference.
        // this behavior emulates a stack of delegates without actually necessitating c

        //具体的解析过程由BeanDefinitionParserDelegate实现，
        //BeanDefinitionParserDelegate中定义了Spring Bean定义XML文件的各种元素
        BeanDefinitionParserDelegate parent = this.delegate;
        this.delegate = createDelegate(getReaderContext(), root, parent);

        if (this.delegate.isDefaultNamespace(root)) {
            String profileSpec = root.getAttribute(PROFILE_ATTRIBUTE);
            if (StringUtils.hasText(profileSpec)) {
                String[] specifiedProfiles = StringUtils.tokenizeToStringArray(
                        profileSpec, BeanDefinitionParserDelegate.MULTI_VALUE_ATTRIBUTE
                if (!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfi
                    if (logger.isInfoEnabled()) {
                        logger.info("Skipped XML bean definition file due to specified
                                "] not matching: " + getReaderContext().getResource());
                    }
                    return;
                }
            }
        }

        //在解析Bean定义之前，进行自定义的解析，增强解析过程的可扩展性
        preProcessXml(root);
        //从Document的根元素开始进行Bean定义的Document对象
        parseBeanDefinitions(root, this.delegate);
        //在解析Bean定义之后，进行自定义的解析，增加解析过程的可扩展性
        postProcessXml(root);

        this.delegate = parent;
    }

    //创建BeanDefinitionParserDelegate，用于完成真正的解析过程
    protected BeanDefinitionParserDelegate createDelegate(
            XmlReaderContext readerContext, Element root, @Nullable BeanDefinitionParse
```

```
56
57          BeanDefinitionParserDelegate delegate = new BeanDefinitionParserDelegate(reader
58          //BeanDefinitionParserDelegate初始化Document根元素
59          delegate.initDefaults(root, parentDelegate);
60          return delegate;
61      }
62      //使用Spring的Bean规则从Document的根元素开始进行Bean定义的Document对象
63      protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate dele
64          //Bean定义的Document对象使用了Spring默认的XML命名空间
65          if (delegate.isDefaultNamespace(root)) {
66              //获取Bean定义的Document对象根元素的所有子节点
67              NodeList nl = root.getChildNodes();
68              for (int i = 0; i < nl.getLength(); i++) {
69                  Node node = nl.item(i);
70                  //获得Document节点是XML元素节点
71                  if (node instanceof Element) {
72                      Element ele = (Element) node;
73                      //Bean定义的Document的元素节点使用的是Spring默认的XML命名空间
74                      if (delegate.isDefaultNamespace(ele)) {
75                          //使用Spring的Bean规则解析元素节点
76                          parseDefaultElement(ele, delegate);
77                      }
78                      else {
79                          //没有使用Spring默认的XML命名空间，则使用用户自定义的解//析规则
80                          delegate.parseCustomElement(ele);
81                      }
82                  }
83              }
84          }
85          else {
86              //Document的根节点没有使用Spring默认的命名空间，则使用用户自定义的
87              //解析规则解析Document根节点
88              delegate.parseCustomElement(root);
89          }
90      }
91
92      //使用Spring的Bean规则解析Document元素节点
93      private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate
94          //如果元素节点是<Import>导入元素，进行导入解析
95          if (delegate.nodeNameEquals(ele, IMPORT_ELEMENT)) {
96              importBeanDefinitionResource(ele);
97          }
98          //如果元素节点是<Alias>别名元素，进行别名解析
```

```
99          else if (delegate.nodeNameEquals(ele, ALIAS_ELEMENT)) {
100             processAliasRegistration(ele);
101         }
102         //元素节点既不是导入元素，也不是别名元素，即普通的<Bean>元素，
103         //按照Spring的Bean规则解析元素
104         else if (delegate.nodeNameEquals(ele, BEAN_ELEMENT)) {
105             processBeanDefinition(ele, delegate);
106         }
107         else if (delegate.nodeNameEquals(ele, NESTED_BEANS_ELEMENT)) {
108             // recurse
109             doRegisterBeanDefinitions(ele);
110         }
111     }
112
113     /**
114      * Parse an "import" element and load the bean definitions
115      * from the given resource into the bean factory.
116      */
117     //解析<Import>导入元素，从给定的导入路径加载Bean定义资源到Spring IoC容器中
118     protected void importBeanDefinitionResource(Element ele) {
119         //获取给定的导入元素的location属性
120         String location = ele.getAttribute(RESOURCE_ATTRIBUTE);
121         //如果导入元素的location属性值为空，则没有导入任何资源，直接返回
122         if (!StringUtils.hasText(location)) {
123             getReaderContext().error("Resource location must not be empty", ele);
124             return;
125         }
126
127         // Resolve system properties: e.g. "${user.dir}"
128         //使用系统变量值解析location属性值
129         location = getReaderContext().getEnvironment().resolveRequiredPlaceholders(loca
130
131         Set<Resource> actualResources = new LinkedHashSet<>(4);
132
133         // Discover whether the location is an absolute or relative URI
134         //标识给定的导入元素的location是否是绝对路径
135         boolean absoluteLocation = false;
136         try {
137             absoluteLocation = ResourcePatternUtils.isUrl(location) || ResourceUtils.tc
138         }
139         catch (URISyntaxException ex) {
140             // cannot convert to an URI, considering the location relative
141             // unless it is the well-known Spring prefix "classpath*:"
```

```
            //给定的导入元素的location不是绝对路径
        }

        // Absolute or relative?
        //给定的导入元素的location是绝对路径
        if (absoluteLocation) {
            try {
                //使用资源读入器加载给定路径的Bean定义资源
                int importCount = getReaderContext().getReader().loadBeanDefinitions(lo
                if (logger.isDebugEnabled()) {
                    logger.debug("Imported " + importCount + " bean definitions from UR
                }
            }
            catch (BeanDefinitionStoreException ex) {
                getReaderContext().error(
                        "Failed to import bean definitions from URL location [" + locat
            }
        }
        else {
            // No URL -> considering resource location as relative to the current file.
            //给定的导入元素的location是相对路径
            try {
                int importCount;
                //将给定导入元素的location封装为相对路径资源
                Resource relativeResource = getReaderContext().getResource().createRela
                //封装的相对路径资源存在
                if (relativeResource.exists()) {
                    //使用资源读入器加载Bean定义资源
                    importCount = getReaderContext().getReader().loadBeanDefinitions(re
                    actualResources.add(relativeResource);
                }
                //封装的相对路径资源不存在
                else {
                    //获取Spring IOC容器资源读入器的基本路径
                    String baseLocation = getReaderContext().getResource().getURL().toS
                    //根据Spring IOC容器资源读入器的基本路径加载给定导入路径的资源
                    importCount = getReaderContext().getReader().loadBeanDefinitions(
                            StringUtils.applyRelativePath(baseLocation, location), actu
                }
                if (logger.isDebugEnabled()) {
                    logger.debug("Imported " + importCount + " bean definitions from re
                }
            }
```

```
            catch (IOException ex) {
                getReaderContext().error("Failed to resolve current resource location",
            }
            catch (BeanDefinitionStoreException ex) {
                getReaderContext().error("Failed to import bean definitions from relati
                        ele, ex);
            }
        }
        Resource[] actResArray = actualResources.toArray(new Resource[actualResources.s
        //在解析完<Import>元素之后，发送容器导入其他资源处理完成事件
        getReaderContext().fireImportProcessed(location, actResArray, extractSource(ele
    }

    /**
     * Process the given alias element, registering the alias with the registry.
     */
    //解析<Alias>别名元素，为Bean向Spring IoC容器注册别名
    protected void processAliasRegistration(Element ele) {
        //获取<Alias>别名元素中name的属性值
        String name = ele.getAttribute(NAME_ATTRIBUTE);
        //获取<Alias>别名元素中alias的属性值
        String alias = ele.getAttribute(ALIAS_ATTRIBUTE);
        boolean valid = true;
        //<alias>别名元素的name属性值为空
        if (!StringUtils.hasText(name)) {
            getReaderContext().error("Name must not be empty", ele);
            valid = false;
        }
        //<alias>别名元素的alias属性值为空
        if (!StringUtils.hasText(alias)) {
            getReaderContext().error("Alias must not be empty", ele);
            valid = false;
        }
        if (valid) {
            try {
                //向容器的资源读入器注册别名
                getReaderContext().getRegistry().registerAlias(name, alias);
            }
            catch (Exception ex) {
                getReaderContext().error("Failed to register alias '" + alias +
                        "' for bean with name '" + name + "'", ele, ex);
            }
            //在解析完<Alias>元素之后，发送容器别名处理完成事件
```

```
            getReaderContext().fireAliasRegistered(name, alias, extractSource(ele));
        }
    }

    /**
     * Process the given bean element, parsing the bean definition
     * and registering it with the registry.
     */
    //解析Bean定义资源Document对象的普通元素
    protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate dele
        BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
        // BeanDefinitionHolder是对BeanDefinition的封装，即Bean定义的封装类
        //对Document对象中<Bean>元素的解析由BeanDefinitionParserDelegate实现
        // BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
        if (bdHolder != null) {
            bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
            try {
                // Register the final decorated instance.
                //向Spring IOC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
                BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderCor
            }
            catch (BeanDefinitionStoreException ex) {
                getReaderContext().error("Failed to register bean definition with name
                        bdHolder.getBeanName() + "'", ele, ex);
            }
            // Send registration event.
            //在完成向Spring IOC容器注册解析得到的Bean定义之后，发送注册事件
            getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHo
        }
    }
```

通过上述Spring IOC容器对载入的Bean定义Document解析可以看出，我们使用Spring时，在Spring 配置文件中可以使用 `<import>` 元素来导入IOC容器所需要的其他资源，Spring IOC容器在解析时会首先将指定导入的资源加载进容器中。使用 `<ailas>` 别名时，Spring IOC容器首先将别名元素所定义的别名注册到容器中。

对于既不是 `<import>` 元素，又不是 `<alias>` 元素的元素，即Spring 配置文件中普通的 `<bean>` 元素的解析由 BeanDefinitionParserDelegate 类的parseBeanDefinitionElement()方法来实现。这个解析的过程非常复杂，我们在mini版本的时候，就用properties文件代替了。

## 2.12. 载入 `<bean>` 元素

Bean配置信息中的 `<import>` 和 `<alias>` 元素解析在DefaultBeanDefinitionDocumentReader中已经完成，对Bean 配置信息中使用最多的 `<bean>` 元素交由BeanDefinitionParserDelegate来解析，其解析实现的源码如下：

```
    //解析<Bean>元素的入口
    @Nullable
    public BeanDefinitionHolder parseBeanDefinitionElement(Element ele) {
        return parseBeanDefinitionElement(ele, null);
    }

    /**
     * Parses the supplied {@code <bean>} element. May return {@code null}
     * if there were errors during parse. Errors are reported to the
     * {@link org.springframework.beans.factory.parsing.ProblemReporter}.
     */
    //解析Bean定义资源文件中的<Bean>元素，这个方法中主要处理<Bean>元素的id，name和别名属
    @Nullable
    public BeanDefinitionHolder parseBeanDefinitionElement(Element ele, @Nullable BeanD
        //获取<Bean>元素中的id属性值
        String id = ele.getAttribute(ID_ATTRIBUTE);
        //获取<Bean>元素中的name属性值
        String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);

        //获取<Bean>元素中的alias属性值
        List<String> aliases = new ArrayList<>();

        //将<Bean>元素中的所有name属性值存放到别名中
        if (StringUtils.hasLength(nameAttr)) {
            String[] nameArr = StringUtils.tokenizeToStringArray(nameAttr, MULTI_VALUE_
            aliases.addAll(Arrays.asList(nameArr));
        }

        String beanName = id;
        //如果<Bean>元素中没有配置id属性时，将别名中的第一个值赋值给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<Bean>元素所配置的id或者name的唯一性， containingBean标识<Bean>
        //元素中是否包含子<Bean>元素
        if (containingBean == null) {
            //检查<Bean>元素所配置的id、name或者别名是否重复
            checkNameUniqueness(beanName, aliases, ele);
        }

        //详细对<Bean>元素中配置的Bean定义进行解析的地方
        AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele, beanNam
        if (beanDefinition != null) {
            if (!StringUtils.hasText(beanName)) {
                try {
                    if (containingBean != null) {
                        //如果<Bean>元素中没有配置id、别名或者name，且没有包含子元素
                        //<Bean>元素，为解析的Bean生成一个唯一beanName并注册
                        beanName = BeanDefinitionReaderUtils.generateBeanName(
                                beanDefinition, this.readerContext.getRegistry(), true)
                    }
                    else {
                        //如果<Bean>元素中没有配置id、别名或者name，且包含了子元素
                        //<Bean>元素，为解析的Bean使用别名向IOC容器注册
                        beanName = this.readerContext.generateBeanName(beanDefinition);
                        // Register an alias for the plain bean class name, if still pc
                        // if the generator returned the class name plus a suffix.
                        // This is expected for Spring 1.2/2.0 backwards compatibility.
                        //为解析的Bean使用别名注册时，为了向后兼容
                        //Spring1.2/2.0，给别名添加类名后缀
                        String beanClassName = beanDefinition.getBeanClassName();
                        if (beanClassName != null &&
                                beanName.startsWith(beanClassName) && beanName.length()
                                !this.readerContext.getRegistry().isBeanNameInUse(beanC
                            aliases.add(beanClassName);
                        }
                    }
                    if (logger.isDebugEnabled()) {
                        logger.debug("Neither XML 'id' nor 'name' specified - " +
                                "using generated bean name [" + beanName + "]");
                    }
                }
                catch (Exception ex) {
                    error(ex.getMessage(), ele);
                    return null;
                }
```

```
            }
            String[] aliasesArray = StringUtils.toStringArray(aliases);
            return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);
        }
        //当解析出错时，返回null
        return null;
    }

    /**
     * Validate that the specified bean name and aliases have not been used already
     * within the current level of beans element nesting.
     */
    protected void checkNameUniqueness(String beanName, List<String> aliases, Element b
        String foundName = null;

        if (StringUtils.hasText(beanName) && this.usedNames.contains(beanName)) {
            foundName = beanName;
        }
        if (foundName == null) {
            foundName = CollectionUtils.findFirstMatch(this.usedNames, aliases);
        }
        if (foundName != null) {
            error("Bean name '" + foundName + "' is already used in this <beans> elemer
        }

        this.usedNames.add(beanName);
        this.usedNames.addAll(aliases);
    }

    /**
     * Parse the bean definition itself, without regard to name or aliases. May return
     * {@code null} if problems occurred during the parsing of the bean definition.
     */
    //详细对<Bean>元素中配置的Bean定义其他属性进行解析
    //由于上面的方法中已经对Bean的id、name和别名等属性进行了处理
    //该方法中主要处理除这三个以外的其他属性数据
    @Nullable
    public AbstractBeanDefinition parseBeanDefinitionElement(
            Element ele, String beanName, @Nullable BeanDefinition containingBean) {
        //记录解析的<Bean>
        this.parseState.push(new BeanEntry(beanName));

        //这里只读取<Bean>元素中配置的class名字，然后载入到BeanDefinition中去
```

```
        //只是记录配置的class名字，不做实例化，对象的实例化在依赖注入时完成
        String className = null;

        //如果<Bean>元素中配置了parent属性，则获取parent属性的值
        if (ele.hasAttribute(CLASS_ATTRIBUTE)) {
            className = ele.getAttribute(CLASS_ATTRIBUTE).trim();
        }
        String parent = null;
        if (ele.hasAttribute(PARENT_ATTRIBUTE)) {
            parent = ele.getAttribute(PARENT_ATTRIBUTE);
        }

        try {
            //根据<Bean>元素配置的class名称和parent属性值创建BeanDefinition
            //为载入Bean定义信息做准备
            AbstractBeanDefinition bd = createBeanDefinition(className, parent);

            //对当前的<Bean>元素中配置的一些属性进行解析和设置，如配置的单态(singleton)
            parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);
            //为<Bean>元素解析的Bean设置description信息
            bd.setDescription(DomUtils.getChildElementValueByTagName(ele, DESCRIPTION_E

            //对<Bean>元素的meta(元信息)属性解析
            parseMetaElements(ele, bd);
            //对<Bean>元素的lookup-method属性解析
            parseLookupOverrideSubElements(ele, bd.getMethodOverrides());
            //对<Bean>元素的replaced-method属性解析
            parseReplacedMethodSubElements(ele, bd.getMethodOverrides());

            //解析<Bean>元素的构造方法设置
            parseConstructorArgElements(ele, bd);
            //解析<Bean>元素的<property>设置
            parsePropertyElements(ele, bd);
            //解析<Bean>元素的qualifier属性
            parseQualifierElements(ele, bd);

            //为当前解析的Bean设置所需的资源和依赖对象
            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
        }
        catch (Throwable ex) {
            error("Unexpected failure during bean definition parsing", ele, ex);
        }
        finally {
            this.parseState.pop();
        }

        //解析<Bean>元素出错时，返回null
        return null;
    }
```

只要使用过Spring对Spring 配置文件比较熟悉的人通过对上述源码的分析就会明白我们在Spring配置文件中 `<Bean>` 元素的中配置的属性就是通过该方法解析和设置到Bean中去的。

注意：在解析 `<Bean>` 元素过程中没有创建和实例化Bean对象，只是创建了Bean对象的定义类BeanDefinition，将 `<Bean>` 元素中的配置信息设置到BeanDefinition中作为记录，当依赖注入时才使用这些记录信息创建和实例化具体的Bean对象。
上面方法中一些对一些配置如元信息（meta）、qualifier等的解析，我们在Spring中配置时使用的也不多，我们在使用Spring的 `<Bean>` 元素时，配置最多的是 `<property>` 属性，因此我们下面继续分析源码，了解Bean的属性在解析时是如何设置的。

## 2.13. 载入 `<property>` 元素

BeanDefinitionParserDelegate 在解析 `<Bean>` 调用parsePropertyElements()方法解析 `<Bean>` 元素中的 `<property>` 属性子元素，解析源码如下：

```
    //解析<Bean>元素中的<property>子元素
    public void parsePropertyElements(Element beanEle, BeanDefinition bd) {
        //获取<Bean>元素中所有的子元素
        NodeList nl = beanEle.getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            //如果子元素是<property>子元素，则调用解析<property>子元素方法解析
            if (isCandidateElement(node) && nodeNameEquals(node, PROPERTY_ELEMENT)) {
                parsePropertyElement((Element) node, bd);
            }
```

```
        }
    }

    //解析<property>元素
    public void parsePropertyElement(Element ele, BeanDefinition bd) {
        //获取<property>元素的名字
        String propertyName = ele.getAttribute(NAME_ATTRIBUTE);
        if (!StringUtils.hasLength(propertyName)) {
            error("Tag 'property' must have a 'name' attribute", ele);
            return;
        }
        this.parseState.push(new PropertyEntry(propertyName));
        try {
            //如果一个Bean中已经有同名的property存在，则不进行解析，直接返回。
            //即如果在同一个Bean中配置同名的property，则只有第一个起作用
            if (bd.getPropertyValues().contains(propertyName)) {
                error("Multiple 'property' definitions for property '" + propertyName +
                return;
            }
            //解析获取property的值
            Object val = parsePropertyValue(ele, bd, propertyName);
            //根据property的名字和值创建property实例
            PropertyValue pv = new PropertyValue(propertyName, val);
            //解析<property>元素中的属性
            parseMetaElements(ele, pv);
            pv.setSource(extractSource(ele));
            bd.getPropertyValues().addPropertyValue(pv);
        }
        finally {
            this.parseState.pop();
        }
    }

    //解析获取property值
    @Nullable
    public Object parsePropertyValue(Element ele, BeanDefinition bd, @Nullable String p
        String elementName = (propertyName != null) ?
                        "<property> element for property '" + propertyName + "'" :
                        "<constructor-arg> element";

        // Should only have one child element: ref, value, list, etc.
        //获取<property>的所有子元素，只能是其中一种类型:ref,value,list,etc等
        NodeList nl = ele.getChildNodes();
```

```
        Element subElement = null;
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            //子元素不是description和meta属性
            if (node instanceof Element && !nodeNameEquals(node, DESCRIPTION_ELEMENT) &
                    !nodeNameEquals(node, META_ELEMENT)) {
                // Child element is what we're looking for.
                if (subElement != null) {
                    error(elementName + " must not contain more than one sub-element",
                }
                else {
                    //当前<property>元素包含有子元素
                    subElement = (Element) node;
                }
            }
        }

        //判断property的属性值是ref还是value，不允许既是ref又是value
        boolean hasRefAttribute = ele.hasAttribute(REF_ATTRIBUTE);
        boolean hasValueAttribute = ele.hasAttribute(VALUE_ATTRIBUTE);
        if ((hasRefAttribute && hasValueAttribute) ||
                ((hasRefAttribute || hasValueAttribute) && subElement != null)) {
            error(elementName +
                    " is only allowed to contain either 'ref' attribute OR 'value' attr
        }

        //如果属性是ref，创建一个ref的数据对象RuntimeBeanReference
        //这个对象封装了ref信息
        if (hasRefAttribute) {
            String refName = ele.getAttribute(REF_ATTRIBUTE);
            if (!StringUtils.hasText(refName)) {
                error(elementName + " contains empty 'ref' attribute", ele);
            }
            //一个指向运行时所依赖对象的引用
            RuntimeBeanReference ref = new RuntimeBeanReference(refName);
            //设置这个ref的数据对象是被当前的property对象所引用
            ref.setSource(extractSource(ele));
            return ref;
        }
        //如果属性是value，创建一个value的数据对象TypedStringValue
        //这个对象封装了value信息
        else if (hasValueAttribute) {
            //一个持有String类型值的对象
```

```
            TypedStringValue valueHolder = new TypedStringValue(ele.getAttribute(VALUE_
            //设置这个value数据对象是被当前的property对象所引用
            valueHolder.setSource(extractSource(ele));
            return valueHolder;
        }
        //如果当前<property>元素还有子元素
        else if (subElement != null) {
            //解析<property>的子元素
            return parsePropertySubElement(subElement, bd);
        }
        else {
            // Neither child element nor "ref" or "value" attribute found.
            //propery属性中既不是ref，也不是value属性，解析出错返回null
            error(elementName + " must specify a ref or value", ele);
            return null;
        }
    }
```

通过对上述源码的分析，我们可以了解在Spring 配置文件中，`<Bean>`元素中`<property>`元素的相关配置是如何处理的：
1、ref被封装为指向依赖对象一个引用。
2、value配置都会封装成一个字符串类型的对象。
3、ref 和value 都通过“解析的数据类型属性值.setSource（extractSource（ele））；"方法将属性值引用与所引用的属性关联起来。

在方法的最后对于`<property>`元素的子元素通过 parsePropertySubElement()方法解析我们继续分析该方法的源码，了解其解析过程。

## 2.14. 载入`<property>`的子元素

在BeanDefinitionParserDelegate类中的parsePropertySubElement()方法对`<property>`中的子元素解析，源码如下：

```
    @Nullable
    public Object parsePropertySubElement(Element ele, @Nullable BeanDefinition bd) {
        return parsePropertySubElement(ele, bd, null);
    }

    /**
     * Parse a value, ref or collection sub-element of a property or
     * constructor-arg element.
     * @param ele subelement of property element; we don't know which yet
```

```
     * @param defaultValueType the default type (class name) for any
     * {@code <value>} tag that might be created
     */
    //解析<property>元素中ref,value或者集合等子元素
    @Nullable
    public Object parsePropertySubElement(Element ele, @Nullable BeanDefinition bd, @Nu
        //如果<property>没有使用Spring默认的命名空间，则使用用户自定义的规则解析内嵌元素
        if (!isDefaultNamespace(ele)) {
            return parseNestedCustomElement(ele, bd);
        }
        //如果子元素是bean，则使用解析<Bean>元素的方法解析
        else if (nodeNameEquals(ele, BEAN_ELEMENT)) {
            BeanDefinitionHolder nestedBd = parseBeanDefinitionElement(ele, bd);
            if (nestedBd != null) {
                nestedBd = decorateBeanDefinitionIfRequired(ele, nestedBd, bd);
            }
            return nestedBd;
        }
        //如果子元素是ref，ref中只能有以下3个属性：bean、local、parent
        else if (nodeNameEquals(ele, REF_ELEMENT)) {
            // A generic reference to any name of any bean.
            //可以不再同一个Spring配置文件中，具体请参考Spring对ref的配置规则
            String refName = ele.getAttribute(BEAN_REF_ATTRIBUTE);
            boolean toParent = false;
            if (!StringUtils.hasLength(refName)) {
                // A reference to the id of another bean in a parent context.
                //获取<property>元素中parent属性值，引用父级容器中的Bean
                refName = ele.getAttribute(PARENT_REF_ATTRIBUTE);
                toParent = true;
                if (!StringUtils.hasLength(refName)) {
                    error("'bean' or 'parent' is required for <ref> element", ele);
                    return null;
                }
            }
            if (!StringUtils.hasText(refName)) {
                error("<ref> element contains empty target attribute", ele);
                return null;
            }
            //创建ref类型数据，指向被引用的对象
            RuntimeBeanReference ref = new RuntimeBeanReference(refName, toParent);
            //设置引用类型值是被当前子元素所引用
            ref.setSource(extractSource(ele));
            return ref;
```

```
        }
        //如果子元素是<idref>，使用解析ref元素的方法解析
        else if (nodeNameEquals(ele, IDREF_ELEMENT)) {
            return parseIdRefElement(ele);
        }
        //如果子元素是<value>，使用解析value元素的方法解析
        else if (nodeNameEquals(ele, VALUE_ELEMENT)) {
            return parseValueElement(ele, defaultValueType);
        }
        //如果子元素是null，为<property>设置一个封装null值的字符串数据
        else if (nodeNameEquals(ele, NULL_ELEMENT)) {
            // It's a distinguished null value. Let's wrap it in a TypedStringValue
            // object in order to preserve the source location.
            TypedStringValue nullHolder = new TypedStringValue(null);
            nullHolder.setSource(extractSource(ele));
            return nullHolder;
        }
        //如果子元素是<array>，使用解析array集合子元素的方法解析
        else if (nodeNameEquals(ele, ARRAY_ELEMENT)) {
            return parseArrayElement(ele, bd);
        }
        //如果子元素是<list>，使用解析list集合子元素的方法解析
        else if (nodeNameEquals(ele, LIST_ELEMENT)) {
            return parseListElement(ele, bd);
        }
        //如果子元素是<set>，使用解析set集合子元素的方法解析
        else if (nodeNameEquals(ele, SET_ELEMENT)) {
            return parseSetElement(ele, bd);
        }
        //如果子元素是<map>，使用解析map集合子元素的方法解析
        else if (nodeNameEquals(ele, MAP_ELEMENT)) {
            return parseMapElement(ele, bd);
        }
        //如果子元素是<props>，使用解析props集合子元素的方法解析
        else if (nodeNameEquals(ele, PROPS_ELEMENT)) {
            return parsePropsElement(ele);
        }
        //既不是ref，又不是value，也不是集合，则子元素配置错误，返回null
        else {
            error("Unknown property sub-element: [" + ele.getNodeName() + "]", ele);
            return null;
        }
    }
```

通过上述源码分析，我们明白了在Spring配置文件中，对 `<property>` 元素中配置的array、list、set、map、prop 等各种集合子元素的都通过上述方法解析，生成对应的数据对象，比如ManagedList、ManagedArray、ManagedSet等，这些Managed 类是Spring对象BeanDefiniton的数据封装，对集合数据类型的具体解析有各自的解析方法实现，解析方法的命名非常规范，一目了然，我们对 `<list>` 集合元素的解析方法进行源码分析，了解其实现过程。

## 2.15. 载入 `<list>` 的子元素

在BeanDefinitionParserDelegate 类中的parselistElement()方法就是具体实现解析 `<property>` 元素中的 `<list>` 集合子元素，源码如下：

```
    //解析<list>集合子元素
    public List<Object> parseListElement(Element collectionEle, @Nullable BeanDefinitio
        //获取<list>元素中的value-type属性，即获取集合元素的数据类型
        String defaultElementType = collectionEle.getAttribute(VALUE_TYPE_ATTRIBUTE);
        //获取<list>集合元素中的所有子节点
        NodeList nl = collectionEle.getChildNodes();
        //Spring中将List封装为ManagedList
        ManagedList<Object> target = new ManagedList<>(nl.getLength());
        target.setSource(extractSource(collectionEle));
        //设置集合目标数据类型
        target.setElementTypeName(defaultElementType);
        target.setMergeEnabled(parseMergeAttribute(collectionEle));
        //具体的<list>元素解析
        parseCollectionElements(nl, target, bd, defaultElementType);
        return target;
    }

    //具体解析<list>集合元素，<array>、<list>和<set>都使用该方法解析
    protected void parseCollectionElements(
            NodeList elementNodes, Collection<Object> target, @Nullable BeanDefinition
        //遍历集合所有节点
        for (int i = 0; i < elementNodes.getLength(); i++) {
            Node node = elementNodes.item(i);
            //节点不是description节点
            if (node instanceof Element && !nodeNameEquals(node, DESCRIPTION_ELEMENT))
                //将解析的元素加入集合中，递归调用下一个子元素
                target.add(parsePropertySubElement((Element) node, bd, defaultElementTy
            }
        }
    }
```

经过对Spring Bean配置信息转换的Document 对象中的元素层层解析Spring IOC现在已经将XML形式定义的Bean 配置信息转换为Spring IOC所识别的数据结构——BeanDefinition，它是Bean配置信息中配置的POJO对象在Spring IOC容器中的映射，我们可以通过AbstractBeanDefinition为入口，看到了IOC容器进行索引、查询和操作。

通过Spring IOC容器对Bean配置资源的解析后，IOC容器大致完成了管理Bean对象的准备工作，即初始化过程，但是最为重要的依赖注入还没有发生，现在在IOC容器中BeanDefinition 存储的只是一些静态信息，接下来需要向容器注册Bean定义信息才能全部完成IOC容器的初始化过程

## 2.16. 分配注册策略

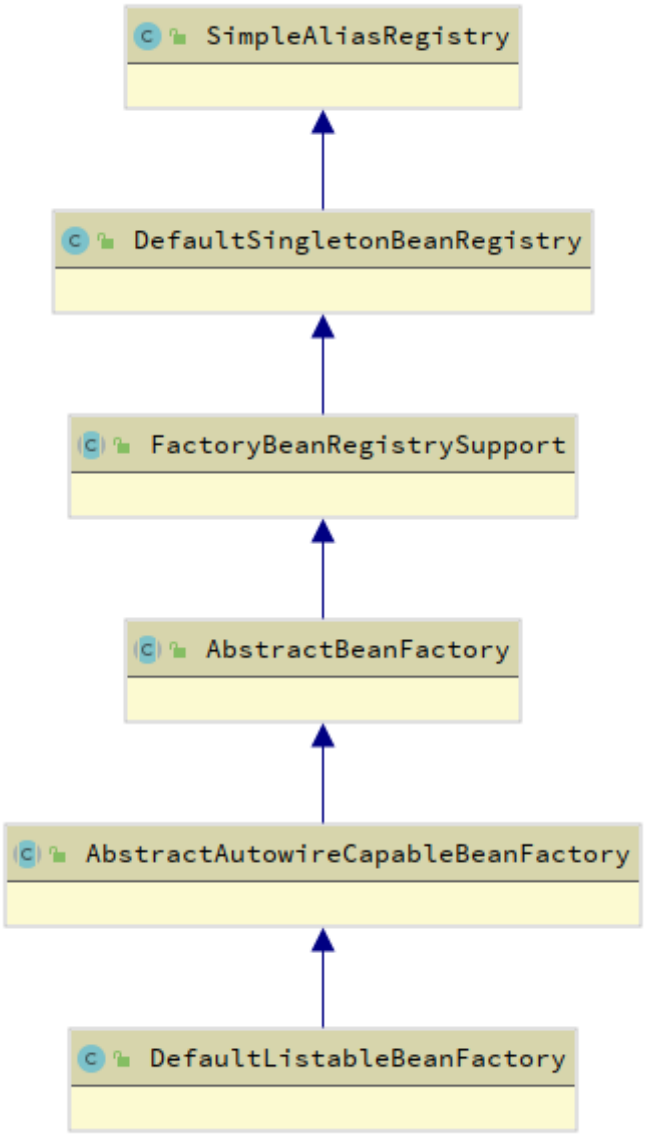
让我们继续跟踪程序的执行顺序，接下来我们来分析DefaultBeanDefinitionDocumentReader对Bean 定义转换的Document对象解析的流程中，在其parseDefaultElement()方法中完成对Document 对象的解析后得到封装BeanDefinition的BeanDefinitionHold对象，然后调用BeanDefinitionReaderUtils 的registerBeanDefinition()方法向IOC容器注册解析的Bean，BeanDefinitionReaderUtils的注册的源码如下：

```
    //将解析的BeanDefinitionHold注册到容器中
    public static void registerBeanDefinition(
            BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry registry)
            throws BeanDefinitionStoreException {

        // Register bean definition under primary name.
        //获取解析的BeanDefinition的名称
        String beanName = definitionHolder.getBeanName();
        //向IOC容器注册BeanDefinition
        registry.registerBeanDefinition(beanName, definitionHolder.getBeanDefinition())

        // Register aliases for bean name, if any.
        //如果解析的BeanDefinition有别名，向容器为其注册别名
        String[] aliases = definitionHolder.getAliases();
        if (aliases != null) {
            for (String alias : aliases) {
                registry.registerAlias(beanName, alias);
            }
        }
    }
```

当调用BeanDefinitionReaderUtils 向IOC 容器注册解析的BeanDefinition时，真正完成注册功能的是DefaultListableBeanFactory。

## 2.17. 向容器注册

DefaultListableBeanFactory中使用一个HashMap的集合对象存放IOC容器中注册解析的BeanDefinition，向IOC容器注册的主要源码如下：




```
    //存储注册信息的BeanDefinition
    private final Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap

    //向IOC容器注册解析的BeanDefiniton
    @Override
    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");
```

```
11
12          //校验解析的BeanDefiniton
13          if (beanDefinition instanceof AbstractBeanDefinition) {
14              try {
15                  ((AbstractBeanDefinition) beanDefinition).validate();
16              }
17              catch (BeanDefinitionValidationException ex) {
18                  throw new BeanDefinitionStoreException(beanDefinition.getResourceDescri
19                          "Validation of bean definition failed", ex);
20              }
21          }
22
23          BeanDefinition oldBeanDefinition;
24
25          oldBeanDefinition = this.beanDefinitionMap.get(beanName);
26
27          if (oldBeanDefinition != null) {
28              if (!isAllowBeanDefinitionOverriding()) {
29                  throw new BeanDefinitionStoreException(beanDefinition.getResourceDescri
30                          "Cannot register bean definition [" + beanDefinition + "] for b
31                          "': There is already [" + oldBeanDefinition + "] bound.");
32              }
33              else if (oldBeanDefinition.getRole() < beanDefinition.getRole()) {
34                  // e.g. was ROLE_APPLICATION, now overriding with ROLE_SUPPORT or ROLE_
35                  if (this.logger.isWarnEnabled()) {
36                      this.logger.warn("Overriding user-defined bean definition for bean
37                              "' with a framework-generated bean definition: replacing ["
38                              oldBeanDefinition + "] with [" + beanDefinition + "]");
39                  }
40              }
41              else if (!beanDefinition.equals(oldBeanDefinition)) {
42                  if (this.logger.isInfoEnabled()) {
43                      this.logger.info("Overriding bean definition for bean '" + beanName
44                              "' with a different definition: replacing [" + oldBeanDefin
45                              "] with [" + beanDefinition + "]");
46                  }
47              }
48              else {
49                  if (this.logger.isDebugEnabled()) {
50                      this.logger.debug("Overriding bean definition for bean '" + beanNam
51                              "' with an equivalent definition: replacing [" + oldBeanDef
52                              "] with [" + beanDefinition + "]");
53                  }
```

```
            }
            this.beanDefinitionMap.put(beanName, beanDefinition);
        }
        else {
            if (hasBeanCreationStarted()) {
                // Cannot modify startup-time collection elements anymore (for stable i
                //注册的过程中需要线程同步，以保证数据的一致性
                synchronized (this.beanDefinitionMap) {
                    this.beanDefinitionMap.put(beanName, beanDefinition);
                    List<String> updatedDefinitions = new ArrayList<>(this.beanDefiniti
                    updatedDefinitions.addAll(this.beanDefinitionNames);
                    updatedDefinitions.add(beanName);
                    this.beanDefinitionNames = updatedDefinitions;
                    if (this.manualSingletonNames.contains(beanName)) {
                        Set<String> updatedSingletons = new LinkedHashSet<>(this.manual
                        updatedSingletons.remove(beanName);
                        this.manualSingletonNames = updatedSingletons;
                    }
                }
            }
            else {
                // Still in startup registration phase
                this.beanDefinitionMap.put(beanName, beanDefinition);
                this.beanDefinitionNames.add(beanName);
                this.manualSingletonNames.remove(beanName);
            }
            this.frozenBeanDefinitionNames = null;
        }

        //检查是否有同名的BeanDefinition已经在IOC容器中注册
        if (oldBeanDefinition != null || containsSingleton(beanName)) {
            //重置所有已经注册过的BeanDefinition的缓存
            resetBeanDefinition(beanName);
        }
    }
```

至此，Bean 配置信息中配置的Bean 被解析过后，已经注册到IOC容器中，被容器管理起来，真正完成了IOC容器初始化所做的全部工作。现在IOC 容器中已经建立了整个Bean的配置信息，这些 BeanDefinition 信息已经可以使用，并且可以被检索，IOC容器的作用就是对这些注册的Bean定义信息进行处理和维护。这些的注册的Bean定义信息是IOC容器控制反转的基础，正是有了这些注册的数据，容器才可以进行依赖注入。

# 3. 基于Annotation的IOC初始化

## 3.1. Annotation的前世今生

从Spring2.0以后的版本中，Spring 也引入了基于注解（Annotation）方式的配置，注解（Annotation）是JDK1.5中引入的一个新特性，用于简化Bean的配置，可以取代XML配置文件。开发人员对注解（Annotation）的态度也是萝卜青菜各有所爱，个人认为注解可以大大简化配置，提高开发速度，但也给后期维护增加了难度。目前来说XML方式发展的相对成熟，方便于统一管理。随着Spring Boot的兴起，基于注解的开发甚至实现了零配置。但作为个人的习惯而言，还是倾向于XML配置文件和注解（Annotation）相互配合使用。Spring IOC容器对于类级别的注解和类内部的注解分以下两种处理策略：

1）、类级别的注解：如@Component、@Repository、@Controller、@Service以及JavaEE6的@ManagedBean 和@Named 注解，都是添加在类上面的类级别注解，Spring容器根据注解的过滤规则扫描读取注解 Bean定义类，并将其注册到Spring IOC容器中。

2）、类内部的注解：如@Autowire、@Value、@Resource以及EJB和WebService相关的注解等，都是添加在类内部的字段或者方法上的类内部注解，SpringIOC容器通过Bean后置注解处理器解析Bean 内部的注解。下面将根据这两种处理策略，分别分析Spring处理注解相关的源码。

## 3.2. 定位Bean扫描路径

在Spring中管理注解Bean定义的容器有两个：AnnotationConfigApplicationContext和AnnotationConfigWebApplicationContex。这两个类是专门处理Spring 注解方式配置的容器，直接依赖于注解作为容器配置信息来源的IOC容器。AnnotationConfigWebApplicationContext是AnnotationConfigApplicationContext的Web版本，两者的用法以及对注解的处理方式几乎没有差别。现在我们以AnnotationConfigApplicationContext为例看看它的源码：

```
public class AnnotationConfigApplicationContext extends GenericApplicationContext imple

    //保存一个读取注解的Bean定义读取器，并将其设置到容器中
    private final AnnotatedBeanDefinitionReader reader;

    //保存一个扫描指定类路径中注解Bean定义的扫描器，并将其设置到容器中
    private final ClassPathBeanDefinitionScanner scanner;


    /**
     * Create a new AnnotationConfigApplicationContext that needs to be populated
     * through {@link #register} calls and then manually {@linkplain #refresh refreshed
     */
    //默认构造函数，初始化一个空容器，容器不包含任何 Bean 信息，需要在稍后通过调用其reg
    //方法注册配置类，并调用refresh()方法刷新容器，触发容器对注解Bean的载入、解析和注册
```

```
16      public AnnotationConfigApplicationContext() {
17          this.reader = new AnnotatedBeanDefinitionReader(this);
18          this.scanner = new ClassPathBeanDefinitionScanner(this);
19      }
20
21      /**
22       * Create a new AnnotationConfigApplicationContext with the given DefaultListableBe
23       * @param beanFactory the DefaultListableBeanFactory instance to use for this conte
24       */
25      public AnnotationConfigApplicationContext(DefaultListableBeanFactory beanFactory) {
26          super(beanFactory);
27          this.reader = new AnnotatedBeanDefinitionReader(this);
28          this.scanner = new ClassPathBeanDefinitionScanner(this);
29      }
30
31      /**
32       * Create a new AnnotationConfigApplicationContext, deriving bean definitions
33       * from the given annotated classes and automatically refreshing the context.
34       * @param annotatedClasses one or more annotated classes,
35       * e.g. {@link Configuration @Configuration} classes
36       */
37      //最常用的构造函数，通过将涉及到的配置类传递给该构造函数，以实现将相应配置类中的Bea
38      public AnnotationConfigApplicationContext(Class<?>... annotatedClasses) {
39          this();
40          register(annotatedClasses);
41          refresh();
42      }
43
44      /**
45       * Create a new AnnotationConfigApplicationContext, scanning for bean definitions
46       * in the given packages and automatically refreshing the context.
47       * @param basePackages the packages to check for annotated classes
48       */
49      //该构造函数会自动扫描以给定的包及其子包下的所有类，并自动识别所有的Spring Bean，将
50      public AnnotationConfigApplicationContext(String... basePackages) {
51          this();
52          scan(basePackages);
53          refresh();
54      }
55
56
57      /**
58       * {@inheritDoc}
```

```
     * <p>Delegates given environment to underlying {@link AnnotatedBeanDefinitionReade
     * and {@link ClassPathBeanDefinitionScanner} members.
     */
    @Override
    public void setEnvironment(ConfigurableEnvironment environment) {
        super.setEnvironment(environment);
        this.reader.setEnvironment(environment);
        this.scanner.setEnvironment(environment);
    }

    /**
     * Provide a custom {@link BeanNameGenerator} for use with {@link AnnotatedBeanDefi
     * and/or {@link ClassPathBeanDefinitionScanner}, if any.
     * <p>Default is {@link org.springframework.context.annotation.AnnotationBeanNameGe
     * <p>Any call to this method must occur prior to calls to {@link #register(Class..
     * and/or {@link #scan(String...)}.
     * @see AnnotatedBeanDefinitionReader#setBeanNameGenerator
     * @see ClassPathBeanDefinitionScanner#setBeanNameGenerator
     */
    //为容器的注解Bean读取器和注解Bean扫描器设置Bean名称产生器
    public void setBeanNameGenerator(BeanNameGenerator beanNameGenerator) {
        this.reader.setBeanNameGenerator(beanNameGenerator);
        this.scanner.setBeanNameGenerator(beanNameGenerator);
        getBeanFactory().registerSingleton(
                AnnotationConfigUtils.CONFIGURATION_BEAN_NAME_GENERATOR, beanNameGenera
    }

    /**
     * Set the {@link ScopeMetadataResolver} to use for detected bean classes.
     * <p>The default is an {@link AnnotationScopeMetadataResolver}.
     * <p>Any call to this method must occur prior to calls to {@link #register(Class..
     * and/or {@link #scan(String...)}.
     */
    //为容器的注解Bean读取器和注解Bean扫描器设置作用范围元信息解析器
    public void setScopeMetadataResolver(ScopeMetadataResolver scopeMetadataResolver) {
        this.reader.setScopeMetadataResolver(scopeMetadataResolver);
        this.scanner.setScopeMetadataResolver(scopeMetadataResolver);
    }


    //---------------------------------------------------------------------
    // Implementation of AnnotationConfigRegistry
    //---------------------------------------------------------------------
```

```
102
103     /**
104      * Register one or more annotated classes to be processed.
105      * <p>Note that {@link #refresh()} must be called in order for the context
106      * to fully process the new classes.
107      * @param annotatedClasses one or more annotated classes,
108      * e.g. {@link Configuration @Configuration} classes
109      * @see #scan(String...)
110      * @see #refresh()
111      */
112     //为容器注册一个要被处理的注解Bean，新注册的Bean，必须手动调用容器的
113     //refresh()方法刷新容器，触发容器对新注册的Bean的处理
114     public void register(Class<?>... annotatedClasses) {
115         Assert.notEmpty(annotatedClasses, "At least one annotated class must be specifi
116         this.reader.register(annotatedClasses);
117     }
118
119     /**
120      * Perform a scan within the specified base packages.
121      * <p>Note that {@link #refresh()} must be called in order for the context
122      * to fully process the new classes.
123      * @param basePackages the packages to check for annotated classes
124      * @see #register(Class...)
125      * @see #refresh()
126      */
127     //扫描指定包路径及其子包下的注解类，为了使新添加的类被处理，必须手动调用
128     //refresh()方法刷新容器
129     public void scan(String... basePackages) {
130         Assert.notEmpty(basePackages, "At least one base package must be specified");
131         this.scanner.scan(basePackages);
132     }
133     ...
```

通过上面的源码分析，我们可以看到Spring 对注解的处理分为两种方式：

1）、直接将注解Bean注册到容器中
可以在初始化容器时注册；也可以在容器创建之后手动调用注册方法向容器注册，然后通过手动刷新容器，使得容器对注册的注解 Bean 进行处理。

2）、通过扫描指定的包及其子包下的所有类
在初始化注解容器时指定要自动扫描的路径，如果容器创建以后向给定路径动态添加了注解 Bean，则需要手动调用容器扫描的方法，然后手动刷新容器，使得容器对所注册的Bean进行处理。

接下来，将会对两种处理方式详细分析其实现过程。

## 3.3. 读取Annotation元数据

当创建注解处理容器时，如果传入的初始参数是具体的注解Bean定义类时，注解容器读取并注册。

### 3.3.1. AnnotationConfigApplicationContext通过调用注解 Bean定义读取器

AnnotatedBeanDefinitionReader的register()方法向容器注册指定的注解Bean，注解 Bean定义读取器向容器注册注解 Bean的源码如下：

```
    //注册多个注解Bean定义类
    public void register(Class<?>... annotatedClasses) {
        for (Class<?> annotatedClass : annotatedClasses) {
            registerBean(annotatedClass);
        }
    }

    /**
     * Register a bean from the given bean class, deriving its metadata from
     * class-declared annotations.
     * @param annotatedClass the class of the bean
     */
    //注册一个注解Bean定义类
    public void registerBean(Class<?> annotatedClass) {
        doRegisterBean(annotatedClass, null, null, null);
    }

    /**
     * Register a bean from the given bean class, deriving its metadata from
     * class-declared annotations, using the given supplier for obtaining a new
     * instance (possibly declared as a lambda expression or method reference).
     * @param annotatedClass the class of the bean
     * @param instanceSupplier a callback for creating an instance of the bean
     * (may be {@code null})
     * @since 5.0
     */
    public <T> void registerBean(Class<T> annotatedClass, @Nullable Supplier<T> instanc
        doRegisterBean(annotatedClass, instanceSupplier, null, null);
    }

    /**
     * Register a bean from the given bean class, deriving its metadata from
```

```
     * class-declared annotations, using the given supplier for obtaining a new
     * instance (possibly declared as a lambda expression or method reference).
     * @param annotatedClass the class of the bean
     * @param name an explicit name for the bean
     * @param instanceSupplier a callback for creating an instance of the bean
     * (may be {@code null})
     * @since 5.0
     */
    public <T> void registerBean(Class<T> annotatedClass, String name, @Nullable Suppli
        doRegisterBean(annotatedClass, instanceSupplier, name, null);
    }

    /**
     * Register a bean from the given bean class, deriving its metadata from
     * class-declared annotations.
     * @param annotatedClass the class of the bean
     * @param qualifiers specific qualifier annotations to consider,
     * in addition to qualifiers at the bean class level
     */
    //Bean定义读取器注册注解Bean定义的入口方法
    @SuppressWarnings("unchecked")
    public void registerBean(Class<?> annotatedClass, Class<? extends Annotation>... qu
        doRegisterBean(annotatedClass, null, null, qualifiers);
    }

    /**
     * Register a bean from the given bean class, deriving its metadata from
     * class-declared annotations.
     * @param annotatedClass the class of the bean
     * @param name an explicit name for the bean
     * @param qualifiers specific qualifier annotations to consider,
     * in addition to qualifiers at the bean class level
     */
    //Bean定义读取器向容器注册注解Bean定义类
    @SuppressWarnings("unchecked")
    public void registerBean(Class<?> annotatedClass, String name, Class<? extends Annc
        doRegisterBean(annotatedClass, null, name, qualifiers);
    }

    /**
     * Register a bean from the given bean class, deriving its metadata from
     * class-declared annotations.
     * @param annotatedClass the class of the bean
```

```
     * @param instanceSupplier a callback for creating an instance of the bean
     * (may be {@code null})
     * @param name an explicit name for the bean
     * @param qualifiers specific qualifier annotations to consider, if any,
     * in addition to qualifiers at the bean class level
     * @param definitionCustomizers one or more callbacks for customizing the
     * factory's {@link BeanDefinition}, e.g. setting a lazy-init or primary flag
     * @since 5.0
     */
    //Bean定义读取器向容器注册注解Bean定义类
    <T> void doRegisterBean(Class<T> annotatedClass, @Nullable Supplier<T> instanceSupp
            @Nullable Class<? extends Annotation>[] qualifiers, BeanDefinitionCustomize

        //根据指定的注解Bean定义类，创建Spring容器中对注解Bean的封装的数据结构
        AnnotatedGenericBeanDefinition abd = new AnnotatedGenericBeanDefinition(annotat
        if (this.conditionEvaluator.shouldSkip(abd.getMetadata())) {
            return;
        }

        abd.setInstanceSupplier(instanceSupplier);
        //解析注解Bean定义的作用域，若@Scope("prototype")，则Bean为原型类型；
        //若@Scope("singleton")，则Bean为单态类型
        ScopeMetadata scopeMetadata = this.scopeMetadataResolver.resolveScopeMetadata(a
        //为注解Bean定义设置作用域
        abd.setScope(scopeMetadata.getScopeName());
        //为注解Bean定义生成Bean名称
        String beanName = (name != null ? name : this.beanNameGenerator.generateBeanNam

        //处理注解Bean定义中的通用注解
        AnnotationConfigUtils.processCommonDefinitionAnnotations(abd);
        //如果在向容器注册注解Bean定义时，使用了额外的限定符注解，则解析限定符注解。
        //主要是配置的关于autowiring自动依赖注入装配的限定条件，即@Qualifier注解
        //Spring自动依赖注入装配默认是按类型装配，如果使用@Qualifier则按名称
        if (qualifiers != null) {
            for (Class<? extends Annotation> qualifier : qualifiers) {
                //如果配置了@Primary注解，设置该Bean为autowiring自动依赖注入装//配时的首
                if (Primary.class == qualifier) {
                    abd.setPrimary(true);
                }
                //如果配置了@Lazy注解，则设置该Bean为非延迟初始化，如果没有配置，
                //则该Bean为预实例化
                else if (Lazy.class == qualifier) {
                    abd.setLazyInit(true);
```

```
                }
                //如果使用了除@Primary和@Lazy以外的其他注解，则为该Bean添加一
                //个autowiring自动依赖注入装配限定符，该Bean在进autowiring
                //自动依赖注入装配时，根据名称装配限定符指定的Bean
                else {
                    abd.addQualifier(new AutowireCandidateQualifier(qualifier));
                }
            }
        }
        for (BeanDefinitionCustomizer customizer : definitionCustomizers) {
            customizer.customize(abd);
        }

        //创建一个指定Bean名称的Bean定义对象，封装注解Bean定义类数据
        BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(abd, beanName)
        //根据注解Bean定义类中配置的作用域，创建相应的代理对象
        definitionHolder = AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, de
        //向IOC容器注册注解Bean类定义对象
        BeanDefinitionReaderUtils.registerBeanDefinition(definitionHolder, this.registr
    }
    ...
```

从上面的源码我们可以看出，注册注解Bean定义类的基本步骤：

a、需要使用注解元数据解析器解析注解Bean中关于作用域的配置。

b、使用AnnotationConfigUtils的processCommonDefinitionAnnotations()方法处理注解 Bean 定义类中通用的注解。

c、使用AnnotationConfigUtils的applyScopedProxyMode()方法创建对于作用域的代理对象。

d、通过BeanDefinitionReaderUtils 向容器注册Bean。

下面我们继续分析这4步的具体实现过程

## 3.3.2. AnnotationScopeMetadataResolver 解析作用域元数据

AnnotationScopeMetadataResolver 通过 resolveScopeMetadata()方法解析注解Bean定义类的作用域元信息，即判断注册的Bean是原生类型（prototype）还是单态（singleton）类型，其源码如下：

```
    //解析注解Bean定义类中的作用域元信息
    @Override
    public ScopeMetadata resolveScopeMetadata(BeanDefinition definition) {
```

```
        ScopeMetadata metadata = new ScopeMetadata();
        if (definition instanceof AnnotatedBeanDefinition) {
            AnnotatedBeanDefinition annDef = (AnnotatedBeanDefinition) definition;
            //从注解Bean定义类的属性中查找属性为”Scope”的值，即@Scope注解的值
            //annDef.getMetadata().getAnnotationAttributes()方法将Bean
            //中所有的注解和注解的值存放在一个map集合中
            AnnotationAttributes attributes = AnnotationConfigUtils.attributesFor(
                    annDef.getMetadata(), this.scopeAnnotationType);
            //将获取到的@Scope注解的值设置到要返回的对象中
            if (attributes != null) {
                metadata.setScopeName(attributes.getString("value"));
                //获取@Scope注解中的proxyMode属性值，在创建代理对象时会用到
                ScopedProxyMode proxyMode = attributes.getEnum("proxyMode");
                //如果@Scope的proxyMode属性为DEFAULT或者NO
                if (proxyMode == ScopedProxyMode.DEFAULT) {
                    //设置proxyMode为NO
                    proxyMode = this.defaultProxyMode;
                }
                //为返回的元数据设置proxyMode
                metadata.setScopedProxyMode(proxyMode);
            }
        }
        //返回解析的作用域元信息对象
        return metadata;
    }
```

上述代码中的annDef.getMetadata().getAnnotationAttributes()方法就是获取对象中指定类型的注解的值。

### 3.3.3. AnnotationConfigUtils 处理注解Bean定义类中的通用注解

AnnotationConfigUtils类的processCommonDefinitionAnnotations()在向容器注册Bean之前，首先对注解 Bean定义类中的通用Spring注解进行处理，源码如下：

```

    //处理Bean定义中通用注解
    static void processCommonDefinitionAnnotations(AnnotatedBeanDefinition abd, Annotat
        AnnotationAttributes lazy = attributesFor(metadata, Lazy.class);
        //如果Bean定义中有@Lazy注解，则将该Bean预实例化属性设置为@lazy注解的值
        if (lazy != null) {
            abd.setLazyInit(lazy.getBoolean("value"));
        }

```

```
        else if (abd.getMetadata() != metadata) {
            lazy = attributesFor(abd.getMetadata(), Lazy.class);
            if (lazy != null) {
                abd.setLazyInit(lazy.getBoolean("value"));
            }
        }
        //如果Bean定义中有@Primary注解，则为该Bean设置为autowiring自动依赖注入装配的首选
        if (metadata.isAnnotated(Primary.class.getName())) {
            abd.setPrimary(true);
        }
        //如果Bean定义中有@ DependsOn注解，则为该Bean设置所依赖的Bean名称，
        //容器将确保在实例化该Bean之前首先实例化所依赖的Bean
        AnnotationAttributes dependsOn = attributesFor(metadata, DependsOn.class);
        if (dependsOn != null) {
            abd.setDependsOn(dependsOn.getStringArray("value"));
        }

        if (abd instanceof AbstractBeanDefinition) {
            AbstractBeanDefinition absBd = (AbstractBeanDefinition) abd;
            AnnotationAttributes role = attributesFor(metadata, Role.class);
            if (role != null) {
                absBd.setRole(role.getNumber("value").intValue());
            }
            AnnotationAttributes description = attributesFor(metadata, Description.clas
            if (description != null) {
                absBd.setDescription(description.getString("value"));
            }
        }
    }
```

### 3.3.4. AnnotationConfigUtils 根据注解 Bean定义类中配置的作用域为其应用相应的代理策略

AnnotationConfigUtils类的applyScopedProxyMode()方法根据注解 Bean定义类中配置的作用域@Scope 注解的值，为Bean 定义应用相应的代理模式，主要是在Spring 面向切面编程（AOP）中使用。

源码如下：

```
    //根据作用域为Bean应用引用的代码模式
    static BeanDefinitionHolder applyScopedProxyMode(
```

```
            ScopeMetadata metadata, BeanDefinitionHolder definition, BeanDefinitionRegi

        //获取注解Bean定义类中@Scope注解的proxyMode属性值
        ScopedProxyMode scopedProxyMode = metadata.getScopedProxyMode();
        //如果配置的@Scope注解的proxyMode属性值为NO，则不应用代理模式
        if (scopedProxyMode.equals(ScopedProxyMode.NO)) {
            return definition;
        }
        //获取配置的@Scope注解的proxyMode属性值，如果为TARGET_CLASS
        //则返回true，如果为INTERFACES，则返回false
        boolean proxyTargetClass = scopedProxyMode.equals(ScopedProxyMode.TARGET_CLASS)
        //为注册的Bean创建相应模式的代理对象
        return ScopedProxyCreator.createScopedProxy(definition, registry, proxyTargetCl
    }
```

这段为Bean引用创建相应模式的代理，这里不做深入的分析。

### 3.3.5. BeanDefinitionReaderUtils 向容器注册 Bean

BeanDefinitionReaderUtils 主要是校验BeanDefinition信息，然后将Bean添加到容器中一个管理BeanDefinition的 HashMap中。

## 3.4. 扫描指定包并解析为BeanDefinition

当创建注解处理容器时，如果传入的初始参数是注解Bean定义类所在的包时，注解容器将扫描给定的包及其子包，将扫描到的注解Bean定义载入并注册。

### 3.4.1. ClassPathBeanDefinitionScanner 扫描给定的包及其子包

AnnotationConfigApplicationContext 通过调用类路径Bean定义扫描器ClassPathBeanDefinitionScanner扫描给定包及其子包下的所有类，主要源码如下：

```
public class ClassPathBeanDefinitionScanner extends ClassPathScanningCandidateComponent

    //创建一个类路径Bean定义扫描器
    public ClassPathBeanDefinitionScanner(BeanDefinitionRegistry registry) {
        this(registry, true);
    }

    //为容器创建一个类路径Bean定义扫描器，并指定是否使用默认的扫描过滤规则。
    //即Spring默认扫描配置：@Component、@Repository、@Service、@Controller
    //注解的Bean，同时也支持JavaEE6的@ManagedBean和JSR-330的@Named注解
```

```
    public ClassPathBeanDefinitionScanner(BeanDefinitionRegistry registry, boolean useD
        this(registry, useDefaultFilters, getOrCreateEnvironment(registry));
    }

    /**
     * Create a new {@code ClassPathBeanDefinitionScanner} for the given bean factory a
     * using the given {@link Environment} when evaluating bean definition profile meta
     * <p>If the passed-in bean factory does not only implement the {@code
     * BeanDefinitionRegistry} interface but also the {@link ResourceLoader} interface,
     * will be used as default {@code ResourceLoader} as well. This will usually be the
     * case for {@link org.springframework.context.ApplicationContext} implementations.
     * <p>If given a plain {@code BeanDefinitionRegistry}, the default {@code ResourceL
     * will be a {@link org.springframework.core.io.support.PathMatchingResourcePatterr
     * @param registry the {@code BeanFactory} to load bean definitions into, in the fc
     * of a {@code BeanDefinitionRegistry}
     * @param useDefaultFilters whether to include the default filters for the
     * {@link org.springframework.stereotype.Component @Component},
     * {@link org.springframework.stereotype.Repository @Repository},
     * {@link org.springframework.stereotype.Service @Service}, and
     * {@link org.springframework.stereotype.Controller @Controller} stereotype annotat
     * @param environment the Spring {@link Environment} to use when evaluating bean
     * definition profile metadata
     * @since 3.1
     * @see #setResourceLoader
     */
    public ClassPathBeanDefinitionScanner(BeanDefinitionRegistry registry, boolean useD
            Environment environment) {

        this(registry, useDefaultFilters, environment,
                (registry instanceof ResourceLoader ? (ResourceLoader) registry : null)
    }

    /**
     * Create a new {@code ClassPathBeanDefinitionScanner} for the given bean factory a
     * using the given {@link Environment} when evaluating bean definition profile meta
     * @param registry the {@code BeanFactory} to load bean definitions into, in the fc
     * of a {@code BeanDefinitionRegistry}
     * @param useDefaultFilters whether to include the default filters for the
     * {@link org.springframework.stereotype.Component @Component},
     * {@link org.springframework.stereotype.Repository @Repository},
     * {@link org.springframework.stereotype.Service @Service}, and
     * {@link org.springframework.stereotype.Controller @Controller} stereotype annotat
     * @param environment the Spring {@link Environment} to use when evaluating bean
```

```
     * definition profile metadata
     * @param resourceLoader the {@link ResourceLoader} to use
     * @since 4.3.6
     */
    public ClassPathBeanDefinitionScanner(BeanDefinitionRegistry registry, boolean useD
            Environment environment, @Nullable ResourceLoader resourceLoader) {

        Assert.notNull(registry, "BeanDefinitionRegistry must not be null");
        //为容器设置加载Bean定义的注册器
        this.registry = registry;

        if (useDefaultFilters) {
            registerDefaultFilters();
        }
        setEnvironment(environment);
        //为容器设置资源加载器
        setResourceLoader(resourceLoader);
    }

    //调用类路径Bean定义扫描器入口方法
    public int scan(String... basePackages) {
        //获取容器中已经注册的Bean个数
        int beanCountAtScanStart = this.registry.getBeanDefinitionCount();

        //启动扫描器扫描给定包
        doScan(basePackages);

        // Register annotation config processors, if necessary.
        //注册注解配置(Annotation config)处理器
        if (this.includeAnnotationConfig) {
            AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
        }

        //返回注册的Bean个数
        return (this.registry.getBeanDefinitionCount() - beanCountAtScanStart);
    }

    /**
     * Perform a scan within the specified base packages,
     * returning the registered bean definitions.
     * <p>This method does <i>not</i> register an annotation config processor
     * but rather leaves this up to the caller.
     * @param basePackages the packages to check for annotated classes
```

```
     * @return set of beans registered if any for tooling registration purposes (never
     */
    //类路径Bean定义扫描器扫描给定包及其子包
    protected Set<BeanDefinitionHolder> doScan(String... basePackages) {
        Assert.notEmpty(basePackages, "At least one base package must be specified");
        //创建一个集合，存放扫描到Bean定义的封装类
        Set<BeanDefinitionHolder> beanDefinitions = new LinkedHashSet<>();
        //遍历扫描所有给定的包
        for (String basePackage : basePackages) {
            //调用父类ClassPathScanningCandidateComponentProvider的方法
            //扫描给定类路径，获取符合条件的Bean定义
            Set<BeanDefinition> candidates = findCandidateComponents(basePackage);
            //遍历扫描到的Bean
            for (BeanDefinition candidate : candidates) {
                //获取Bean定义类中@Scope注解的值，即获取Bean的作用域
                ScopeMetadata scopeMetadata = this.scopeMetadataResolver.resolveScopeMe
                //为Bean设置注解配置的作用域
                candidate.setScope(scopeMetadata.getScopeName());
                //为Bean生成名称
                String beanName = this.beanNameGenerator.generateBeanName(candidate, th
                //如果扫描到的Bean不是Spring的注解Bean，则为Bean设置默认值，
                //设置Bean的自动依赖注入装配属性等
                if (candidate instanceof AbstractBeanDefinition) {
                    postProcessBeanDefinition((AbstractBeanDefinition) candidate, beanN
                }
                //如果扫描到的Bean是Spring的注解Bean，则处理其通用的Spring注解
                if (candidate instanceof AnnotatedBeanDefinition) {
                    //处理注解Bean中通用的注解，在分析注解Bean定义类读取器时已经分析过
                    AnnotationConfigUtils.processCommonDefinitionAnnotations((Annotated
                }
                //根据Bean名称检查指定的Bean是否需要在容器中注册，或者在容器中冲突
                if (checkCandidate(beanName, candidate)) {
                    BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(ca
                    //根据注解中配置的作用域，为Bean应用相应的代理模式
                    definitionHolder =
                            AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, d
                    beanDefinitions.add(definitionHolder);
                    //向容器注册扫描到的Bean
                    registerBeanDefinition(definitionHolder, this.registry);
                }
            }
        }
        return beanDefinitions;
```

```
    }
    ...
```

类路径Bean定义扫描器ClassPathBeanDefinitionScanner 主要通过findCandidateComponents()方法调用其父类ClassPathScanningCandidateComponentProvider类来扫描获取给定包及其子包下

### 3.4.2. ClassPathScanningCandidateComponentProvider 扫描给定包及其子包的类

ClassPathScanningCandidateComponentProvider 类的findCandidateComponents()方法具体实现扫描给定类路径包的功能，主要源码如下：

```
public class ClassPathScanningCandidateComponentProvider implements EnvironmentCapable,

    //保存过滤规则要包含的注解，即Spring默认的@Component、@Repository、@Service、
    //@Controller注解的Bean，以及JavaEE6的@ManagedBean和JSR-330的@Named注解
    private final List<TypeFilter> includeFilters = new LinkedList<>();

    //保存过滤规则要排除的注解
    private final List<TypeFilter> excludeFilters = new LinkedList<>();

    //构造方法，该方法在子类ClassPathBeanDefinitionScanner的构造方法中被调用
    public ClassPathScanningCandidateComponentProvider(boolean useDefaultFilters) {
        this(useDefaultFilters, new StandardEnvironment());
    }

    /**
     * Create a ClassPathScanningCandidateComponentProvider with the given {@link Envir
     * @param useDefaultFilters whether to register the default filters for the
     * {@link Component @Component}, {@link Repository @Repository},
     * {@link Service @Service}, and {@link Controller @Controller}
     * stereotype annotations
     * @param environment the Environment to use
     * @see #registerDefaultFilters()
     */
    public ClassPathScanningCandidateComponentProvider(boolean useDefaultFilters, Envir
        //如果使用Spring默认的过滤规则，则向容器注册过滤规则
        if (useDefaultFilters) {
            registerDefaultFilters();
        }
```

```
        setEnvironment(environment);
        setResourceLoader(null);
    }

    //向容器注册过滤规则
    @SuppressWarnings("unchecked")
    protected void registerDefaultFilters() {
        //向要包含的过滤规则中添加@Component注解类，注意Spring中@Repository
        //@Service和@Controller都是Component，因为这些注解都添加了@Component注解
        this.includeFilters.add(new AnnotationTypeFilter(Component.class));
        //获取当前类的类加载器
        ClassLoader cl = ClassPathScanningCandidateComponentProvider.class.getClassLoad
        try {
            //向要包含的过滤规则添加JavaEE6的@ManagedBean注解
            this.includeFilters.add(new AnnotationTypeFilter(
                    ((Class<? extends Annotation>) ClassUtils.forName("javax.annotation
            logger.debug("JSR-250 'javax.annotation.ManagedBean' found and supported fo
        }
        catch (ClassNotFoundException ex) {
            // JSR-250 1.1 API (as included in Java EE 6) not available - simply skip.
        }
        try {
            //向要包含的过滤规则添加@Named注解
            this.includeFilters.add(new AnnotationTypeFilter(
                    ((Class<? extends Annotation>) ClassUtils.forName("javax.inject.Nam
            logger.debug("JSR-330 'javax.inject.Named' annotation found and supported f
        }
        catch (ClassNotFoundException ex) {
            // JSR-330 API not available - simply skip.
        }
    }

    //扫描给定类路径的包
    public Set<BeanDefinition> findCandidateComponents(String basePackage) {
        if (this.componentsIndex != null && indexSupportsIncludeFilters()) {
            return addCandidateComponentsFromIndex(this.componentsIndex, basePackage);
        }
        else {
            return scanCandidateComponents(basePackage);
        }
    }

    //判断元信息读取器读取的类是否符合容器定义的注解过滤规则
```

```
    protected boolean isCandidateComponent(MetadataReader metadataReader) throws IOExce
        //如果读取的类的注解在排除注解过滤规则中，返回false
        for (TypeFilter tf : this.excludeFilters) {
            if (tf.match(metadataReader, getMetadataReaderFactory())) {
                return false;
            }
        }
        //如果读取的类的注解在包含的注解的过滤规则中，则返回ture
        for (TypeFilter tf : this.includeFilters) {
            if (tf.match(metadataReader, getMetadataReaderFactory())) {
                return isConditionMatch(metadataReader);
            }
        }
        //如果读取的类的注解既不在排除规则，也不在包含规则中，则返回false
        return false;
    }
```

## 3.5. 注册注解BeanDefinition

AnnotationConfigWebApplicationContext是AnnotationConfigApplicationContext的Web版，它们对于注解Bean的注册和扫描是基本相同的，但是AnnotationConfigWebApplicationContext对注解Bean定义的载入稍有不同，AnnotationConfigWebApplicationContext 注入注解 Bean定义源码如下：

```

    //载入注解Bean定义资源
    @Override
    protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) {
        //为容器设置注解Bean定义读取器
        AnnotatedBeanDefinitionReader reader = getAnnotatedBeanDefinitionReader(beanFac
        //为容器设置类路径Bean定义扫描器
        ClassPathBeanDefinitionScanner scanner = getClassPathBeanDefinitionScanner(bean

        //获取容器的Bean名称生成器
        BeanNameGenerator beanNameGenerator = getBeanNameGenerator();
        //为注解Bean定义读取器和类路径扫描器设置Bean名称生成器
        if (beanNameGenerator != null) {
            reader.setBeanNameGenerator(beanNameGenerator);
            scanner.setBeanNameGenerator(beanNameGenerator);
            beanFactory.registerSingleton(AnnotationConfigUtils.CONFIGURATION_BEAN_NAME
        }

```

```
        //获取容器的作用域元信息解析器
        ScopeMetadataResolver scopeMetadataResolver = getScopeMetadataResolver();
        //为注解Bean定义读取器和类路径扫描器设置作用域元信息解析器
        if (scopeMetadataResolver != null) {
            reader.setScopeMetadataResolver(scopeMetadataResolver);
            scanner.setScopeMetadataResolver(scopeMetadataResolver);
        }

        if (!this.annotatedClasses.isEmpty()) {
            if (logger.isInfoEnabled()) {
                logger.info("Registering annotated classes: [" +
                        StringUtils.collectionToCommaDelimitedString(this.annotatedClas
            }
            reader.register(this.annotatedClasses.toArray(new Class<?>[this.annotatedCl
        }

        if (!this.basePackages.isEmpty()) {
            if (logger.isInfoEnabled()) {
                logger.info("Scanning base packages: [" +
                        StringUtils.collectionToCommaDelimitedString(this.basePackages)
            }
            scanner.scan(this.basePackages.toArray(new String[this.basePackages.size()]
        }

        //获取容器定义的Bean定义资源路径
        String[] configLocations = getConfigLocations();
        //如果定位的Bean定义资源路径不为空
        if (configLocations != null) {
            for (String configLocation : configLocations) {
                try {
                    //使用当前容器的类加载器加载定位路径的字节码类文件
                    Class<?> clazz = ClassUtils.forName(configLocation, getClassLoader(
                    if (logger.isInfoEnabled()) {
                        logger.info("Successfully resolved class for [" + configLocatio
                    }
                    reader.register(clazz);
                }
                catch (ClassNotFoundException ex) {
                    if (logger.isDebugEnabled()) {
                        logger.debug("Could not load class for config location [" + con
                                "] - trying package scan. " + ex);
                    }
                    //如果容器类加载器加载定义路径的Bean定义资源失败
```

```
                    //则启用容器类路径扫描器扫描给定路径包及其子包中的类
                    int count = scanner.scan(configLocation);
                    if (logger.isInfoEnabled()) {
                        if (count == 0) {
                            logger.info("No annotated classes found for specified class
                        }
                        else {
                            logger.info("Found " + count + " annotated classes in packa
                        }
                    }
                }
            }
        }
    }
    ...
```

以上就是解析和注入注解配置资源的全过程分析。

# 4. IOC容器初始化小结

现在通过上面的代码，总结一下IOC容器初始化的基本步骤：
1、初始化的入口在容器实现中的refresh()调用来完成。
2、对Bean定义载入IOC容器使用的方法是loadBeanDefinition().

其中的大致过程如下：通过ResourceLoader 来完成资源文件位置的定位，DefaultResourceLoader是默认的实现，同时上下文本身就给出了ResourceLoader的实现，可以从类路径，文件系统，URL等方式来定为资源位置。如果是XmlBeanFactory作为IOC容器，那么需要为它指定Bean定义的资源，也就是说Bean定义文件时通过抽象成Resource来被IOC容器处理的，容器通过BeanDefinitionReader 来完成定义信息的解析和Bean信息的注册，往往使用的是XmlBeanDefinitionReader来解析Bean的XML定义文件-实际的处理过程是委托给BeanDefinitionParserDelegate来完成的，从而得到bean的定义信息，这些信息在Spring中使用BeanDefinition对象来表示-这个名字可以让我们想到loadBeanDefinition(), registerBeanDefinition()这些相关方法。它们都是为处理BeanDefinitin服务的，容器解析得到BeanDefinition以后，需要把它在IOC容器中注册，这由IOC实现BeanDefinitionRegistry 接口来实现。注册过程就是在IOC容器内部维护的一个HashMap来保存得到的BeanDefinition的过程。这个HashMap是IOC 容器持有Bean 信息的场所，以后对Bean的操作都是围绕这个HashMap来实现的。

然后我们就可以通过BeanFactory和ApplicationContext来享受到Spring IOC的服务了，在使用IOC容器的时候，我们注意到除了少量粘合代码，绝大多数以正确IOC风格编写的应用程序代码完全不用关心如何到达工厂，因为容器将把这些对象与容器管理的其他对象钩在一起。基本的策略是把工厂放到已知的地方，最好是放在对预期使用的上下文有意义的地方以及代码将实际需要访问工厂的地方。

Spring本身提供了对声明式载入web应用程序用法的应用程序上下文，并将其存储在ServletContext中的框架实现。

以下是容器初始化全过程的时序图：