

咕泡学院 JavaVIP 高级课程教案

深入分析 Spring 源码（第二阶段）

关于本文档

主题	咕泡学院 Java VIP 高级课程教案--深入分析 Spring 源码（第二阶段）
主讲	Tom 老师
适用对象	咕泡学院 Java 高级 VIP 学员及 VIP 授课老师
源码版本	v3.2.6.RELEASE

五、Spring 源码解读

5.1、什么是 IOC/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可配置的文件。

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：

对象和对象关系怎么表示？

可以用 `xml` ， `properties` 文件等语义化配置文件表示。

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。

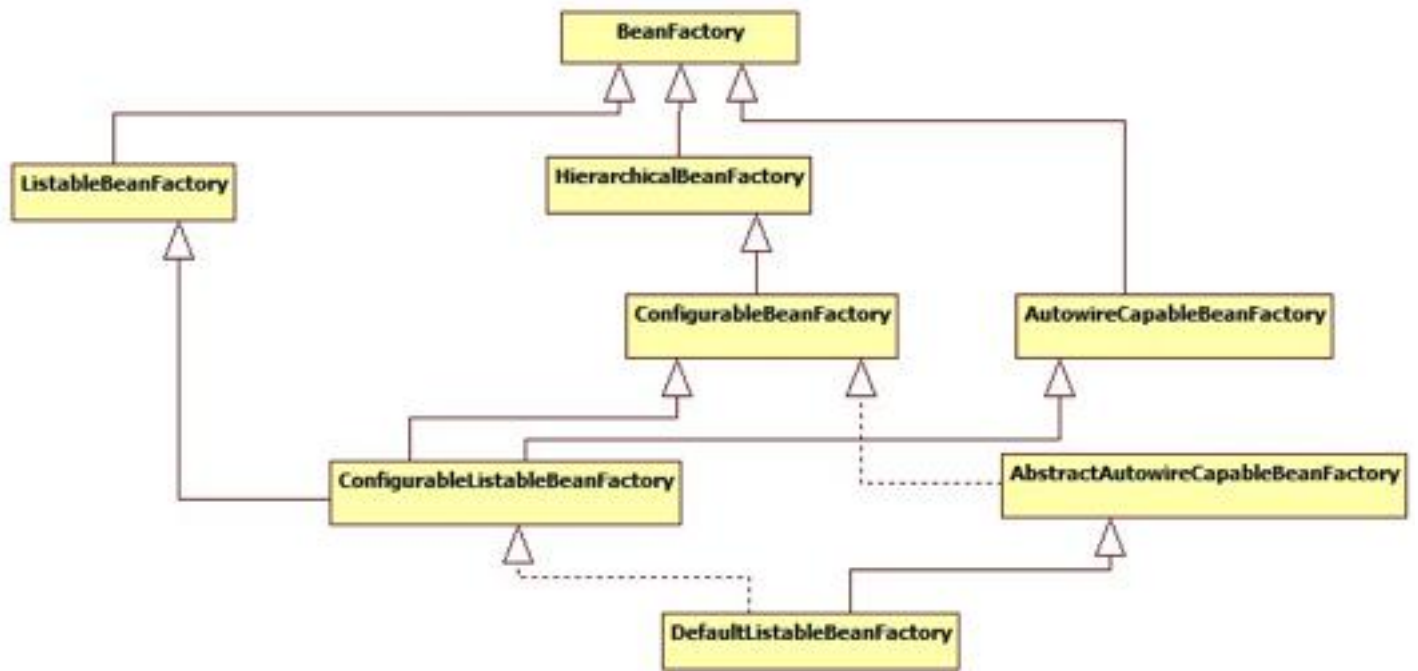
不同的配置文件对对象的描述不一样，如标准的，自定义声明式的，如何统一？ 在内部需要有一个统一的关于对象的定义，所有外部的描述都必须转化成统一的描述定义。

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

5.2、Spring IOC 体系结构

(1) BeanFactory

Spring Bean 的创建是典型的工厂模式，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** 有三个子类：**ListableBeanFactory**、**HierarchicalBeanFactory** 和 **AutowireCapableBeanFactory**。但是从上图中我们可以发现最终的默认实现类是 **DefaultListableBeanFactory**，他实现了所有的接口。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，每个接口都有他使用的场合，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中，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。这四个接口共同定义了 **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**

```

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的对象，
    //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。
    Object getBean(String name, Class requiredType) throws BeansException;
    //提供对 bean 的检索，看看是否在 IOC 容器有这个名称的 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;

    //得到 bean 实例的 Class 类型
    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 XmlBeanFactory，ClasspathXmlApplicationContext 等。其中 XmlBeanFactory 就是针对最基本的 IOC 容器的实现，这个 IOC 容器可以读取 XML 文件定义的 BeanDefinition（XML 文件中对 bean 的描述），如果说 XmlBeanFactory 是容器中的屌丝，ApplicationContext 应该算容器中的高帅富。

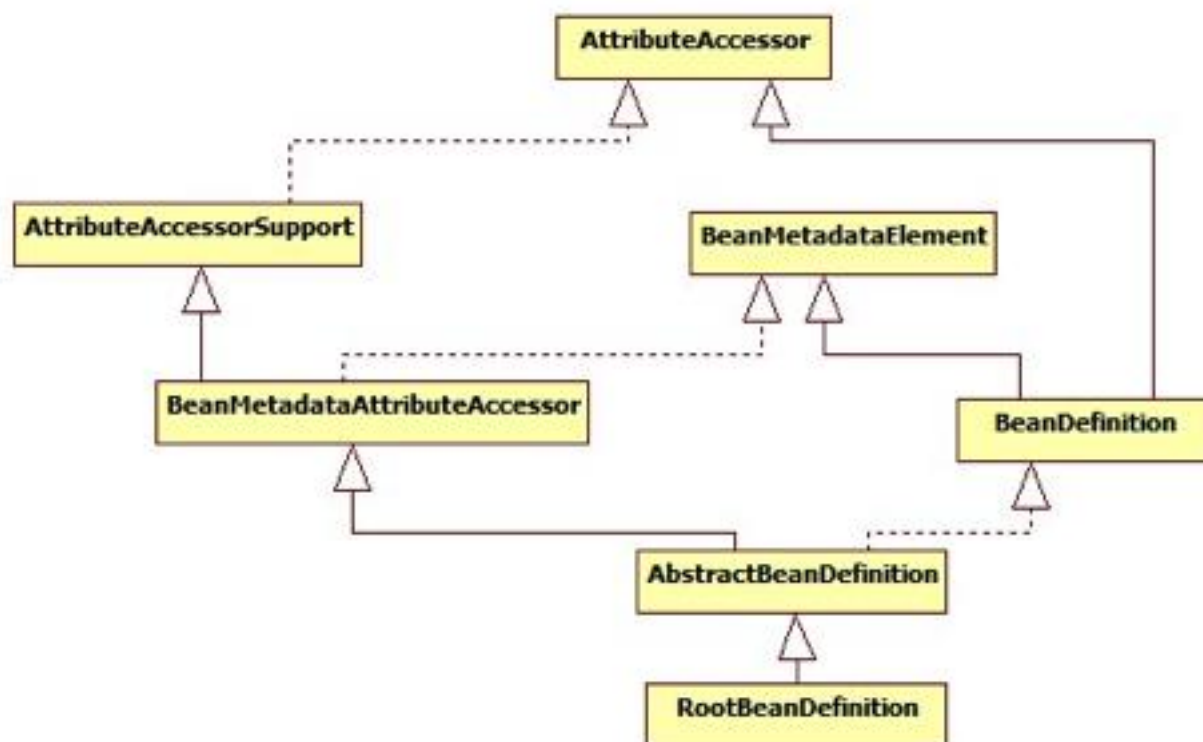
ApplicationContext 是 Spring 提供的一个高级的 IOC 容器，它除了能够提供 IOC 容器的基本功能外，还为用户提供了以下的附加服务。

从 ApplicationContext 接口的实现，我们看出其特点：

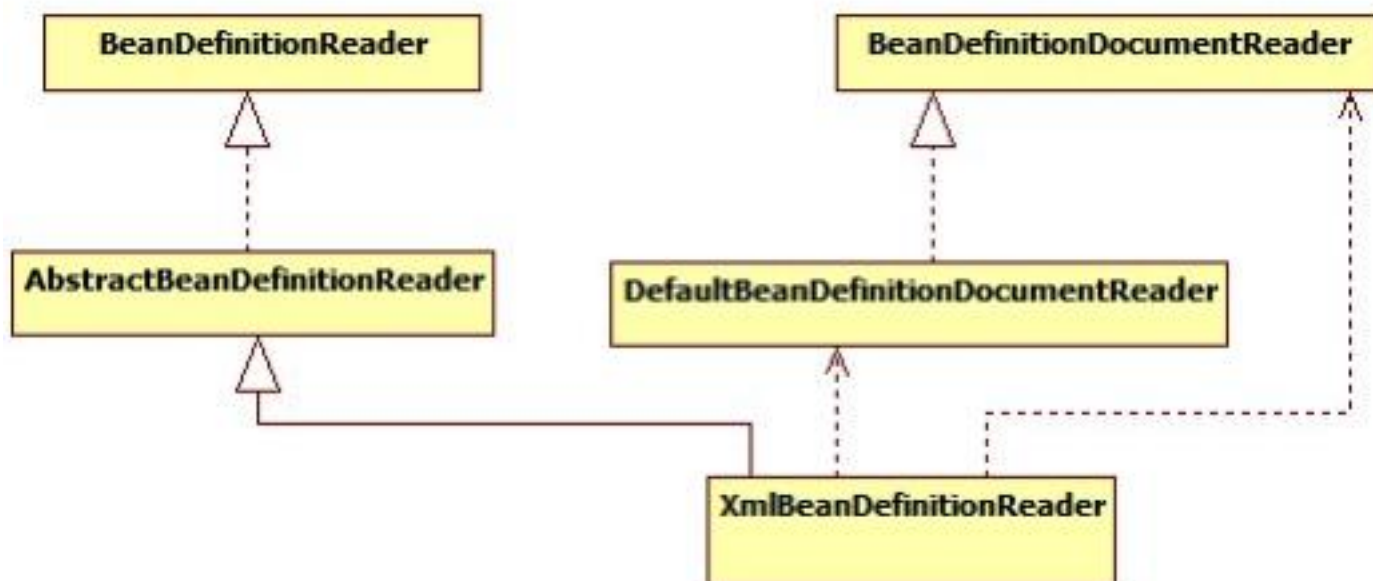
1. 支持信息源，可以实现国际化。（实现 MessageSource 接口）
2. 访问资源。（实现 ResourcePatternResolver 接口，这个后面要讲）
3. 支持应用事件。（实现 ApplicationEventPublisher 接口）

(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：

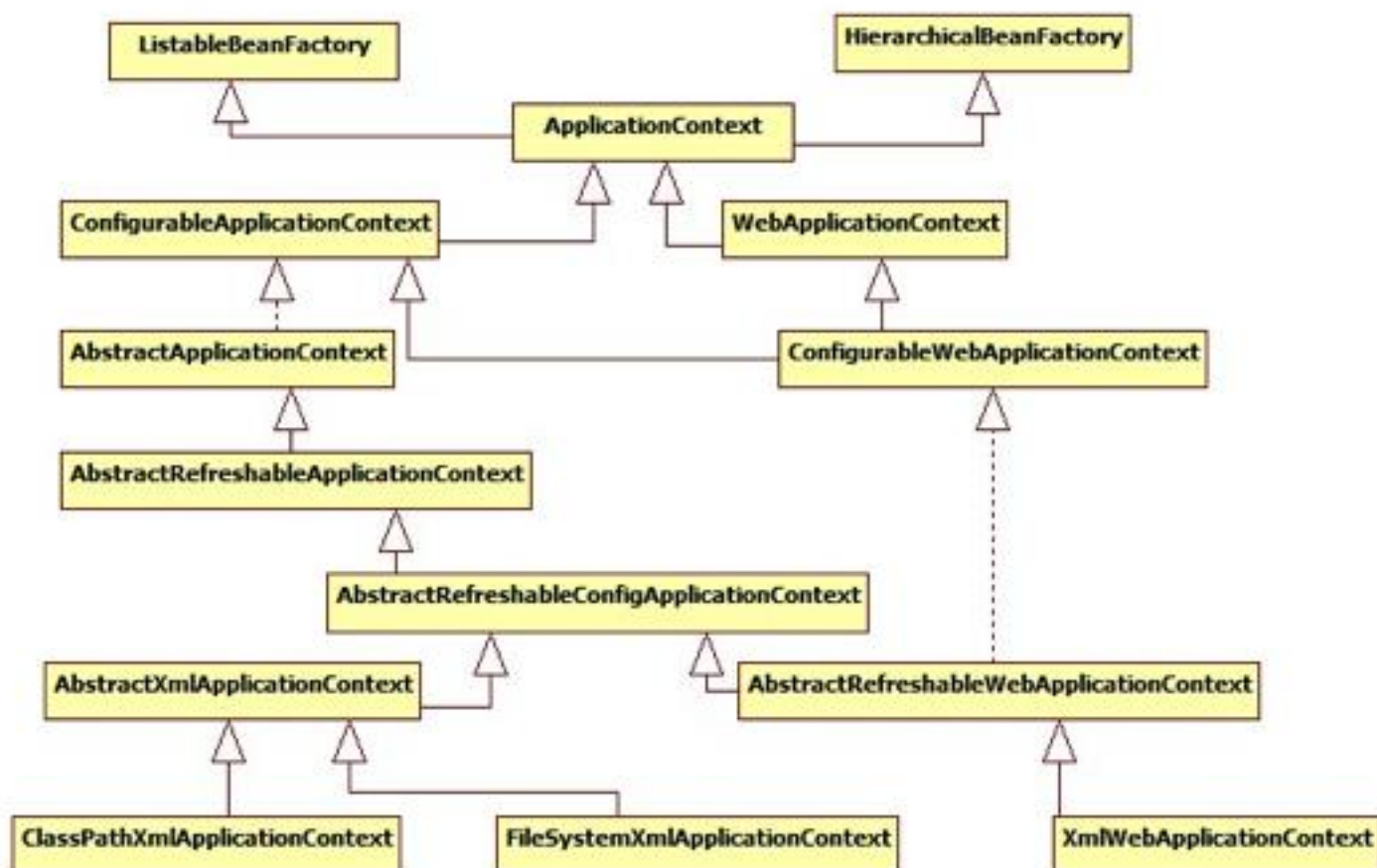


Bean 的解析过程非常复杂，功能被分的很细，因为这里需要被扩展的地方很多，必须保证有足够的灵活性，以应对可能的变化。Bean 的解析主要就是对 Spring 配置文件的解析。这个解析过程主要通过下图中的类完成：



5.3、IOC 容器的初始化

IOC 容器的初始化包括 BeanDefinition 的 Resource 定位、载入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。

下面我们分别简单地演示一下两种 ioc 容器的创建过程

1、XmlBeanFactory(屌丝 IOC)的整个流程

通过 XmlBeanFactory 的源码，我们可以发现：

```
public class XmlBeanFactory extends DefaultListableBeanFactory{
    private final XmlBeanDefinitionReader reader;

    public XmlBeanFactory(Resource resource)throws BeansException{
        this(resource, null);
    }

    public XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory)
        throws BeansException{
        super(parentBeanFactory);
        this.reader = new XmlBeanDefinitionReader(this);
        this.reader.loadBeanDefinitions(resource);
    }
}
```

调用全过程还原，定位、载入、注册

```
//根据 Xml 配置文件创建 Resource 资源对象，该对象中包含了 BeanDefinition 的信息
ClassPathResource resource =new ClassPathResource("application-context.xml");
//创建 DefaultListableBeanFactory
DefaultListableBeanFactory factory =new DefaultListableBeanFactory();
//创建 XmlBeanDefinitionReader 读取器，用于载入 BeanDefinition。之所以需要 BeanFactory 作为参数，是因为会将读取的信息回调配置给 factory
XmlBeanDefinitionReader reader =new XmlBeanDefinitionReader(factory);
//XmlBeanDefinitionReader 执行载入 BeanDefinition 的方法，最后会完成 Bean 的载入和注册。完成后 Bean 就成功的放置到 IOC 容器当中，以后我们就可以从中取得 Bean 来使用

reader.loadBeanDefinitions(resource);
```

通过前面的源码，this.reader = new XmlBeanDefinitionReader(this); 中其中 this 传的是 factory 对象

2、FileSystemXmlApplicationContext 的 IOC 容器流程

1、高富帅 IOC 解剖

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

先看其构造函数：

调用构造函数：

```
/**
 * Create a new FileSystemXmlApplicationContext, loading the definitions
 * from the given XML files and automatically refreshing the context.
 * @param configLocations array of file paths
 * @throws BeansException if context creation failed
 */
public FileSystemXmlApplicationContext(String... configLocations) throws BeansException {
    this(configLocations, true, null);
}
```

实际调用

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

2、设置资源加载器和资源定位

通过分析 `FileSystemXmlApplicationContext` 的源代码可以知道，在创建 `FileSystemXmlApplicationContext` 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 定义资源文件的定位路径。

通过追踪 `FileSystemXmlApplicationContext` 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, DisposableBean {
    //静态初始化块，在整个容器创建过程中只执行一次
```



```

static {
    //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();
}
public AbstractApplicationContext() {
    this.resourcePatternResolver = getResourcePatternResolver();
}
//FileSystemXmlApplicationContext 调用父类构造方法调用的就是该方法
public AbstractApplicationContext(ApplicationContext parent) {
    this();
    setParent(parent);
}
//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 构造方法中调用 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 FileSystemXmlApplicationContet 执行 setConfigLocations 方法通过调用其父类 AbstractRefreshableConfigApplicationContext 的方法进行对 Bean 定义资源文件的定位，该方法的源码如下：

```

//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_DELIMITERS));
}

//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(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 定义资源文件，也可以使用字符串数组，即下面两种方式都是可以的：

A. ClasspathResource res = new ClasspathResource(“a.xml,b.xml,……”);

多个资源文件路径之间可以用“ ,; /t/n” 等分隔。

B. ClasspathResource res = new ClasspathResource(newString[]{“a.xml”, “b.xml”, ……});

至此，Spring IOC 容器在初始化时将配置的 Bean 定义资源文件定位为 Spring 封装的 Resource。

3、AbstractApplicationContext 的 refresh 函数载入 Bean 定义过程：

Spring IOC 容器对 Bean 定义资源的载入是从 refresh() 函数开始的，refresh() 是一个模板方法，refresh() 方法的作用是：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。refresh 的作用类似于对 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 定义资源进行载入

FileSystemXmlApplicationContext 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：

```

public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        //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();
        //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();
        //为 BeanFactory 配置容器特性，例如类加载器、事件处理器等
        prepareBeanFactory(beanFactory);
    }
}

```

```

try {
    //为容器的某些子类指定特殊的 BeanPost 事件处理器
    postProcessBeanFactory(beanFactory);
    //调用所有注册的 BeanFactoryPostProcessor 的 Bean
    invokeBeanFactoryPostProcessors(beanFactory);
    //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);
    //初始化信息源，和国际化相关。
    initMessageSource();
    //初始化容器事件传播器。
    initApplicationEventMulticaster();
    //调用子类的某些特殊 Bean 初始化方法
    onRefresh();
    //为事件传播器注册事件监听器。
    registerListeners();
    //初始化所有剩余的单例 Bean。
    finishBeanFactoryInitialization(beanFactory);
    //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) {
    //销毁以创建的单态 Bean
    destroyBeans();
    //取消 refresh 操作，重置容器的同步标识。
    cancelRefresh(ex);
    throw ex;
}
}
}

```

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 定义资源文件从其子类容器的 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，载入过程就是从这句代码启动。

refresh() 方法的作用是：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。refresh 的作用类似于对 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 定义资源进行载入

AbstractApplicationContext 的 obtainFreshBeanFactory() 方法调用子类容器的 refreshBeanFactory() 方法，启动容器载入 Bean 定义资源文件的过程，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，具体实现调用子类容器的

```

refreshBeanFactory()方法

```
refreshBeanFactory();
ConfigurableListableBeanFactory beanFactory = getBeanFactory();
if (logger.isDebugEnabled()) {
    logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
}
return beanFactory;
}
```

AbstractApplicationContext 子类的 refreshBeanFactory() 方法:

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:

```
protected final void refreshBeanFactory() throws BeansException {
    if (hasBeanFactory()) { // 如果已经有容器, 销毁容器中的 bean, 关闭容器
        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象的 loadBeanDefinitions
        // 方法, 具体的实现调用子类容器
        loadBeanDefinitions(beanFactory);
        synchronized (this.beanFactoryMonitor) {
            this.beanFactory = beanFactory;
        }
    }
    catch (IOException ex) {
        throw new ApplicationContextException("I/O error parsing bean definition source for " +
            getDisplayName(), ex);
    }
}
```

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。

5、AbstractRefreshableApplicationContext 子类的 loadBeanDefinitions 方法:

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
AbstractRefreshableConfigApplicationContext {
    .....
    //实现父类抽象的载入 Bean 定义方法
    @Override
    protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansException,
IOException {
        //创建 XmlBeanDefinitionReader，即创建 Bean 读取器，并通过回调设置到容器中去，容器使用该读取器读
        取 Bean 定义资源
        XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);
        //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，容器本身也是一个资源加载
        器

        beanDefinitionReader.setEnvironment(this.getEnvironment());
        beanDefinitionReader.setResourceLoader(this);
        //为 Bean 读取器设置 SAX xml 解析器
        beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));
        //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
        initBeanDefinitionReader(beanDefinitionReader);
        //Bean 读取器真正实现加载的方法
        loadBeanDefinitions(beanDefinitionReader);
    }
    //Xml Bean 读取器加载 Bean 定义资源
    protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException,
IOException {
        //获取 Bean 定义资源的定位
        Resource[] configResources = getConfigResources();
        if (configResources != null) {
            //Xml Bean 读取器调用其父类 AbstractBeanDefinitionReader 读取定位
            //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 null，则获取 FileSystemXmlApplicationContext 构造方法中
        setConfigLocations 方法设置的资源
        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);
    }
}
//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
protected Resource[] getConfigResources() {
    return null;
}
.....
}
}

```

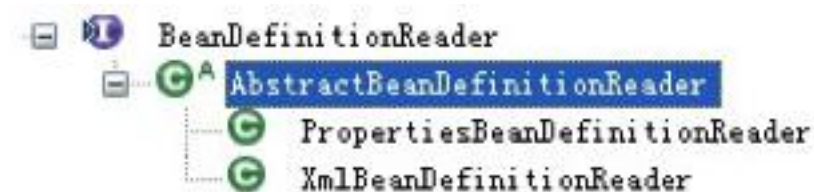
Xml Bean 读取器(XmlBeanDefinitionReader)调用其父类 AbstractBeanDefinitionReader 的 reader.loadBeanDefinitions 方法读取 Bean 定义资源。

由于我们使用 FileSystemXmlApplicationContext 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。

6、AbstractBeanDefinitionReader 读取 Bean 定义资源：

AbstractBeanDefinitionReader 的 loadBeanDefinitions 方法源码如下：

可以到 org.springframework.beans.factory.support 看一下 BeanDefinitionReader 的结构



在其抽象父类 AbstractBeanDefinitionReader 中定义了载入过程

```

//重载方法，调用下面的 loadBeanDefinitions(String, Set<Resource>);方法
public int loadBeanDefinitions(String location) throws BeanDefinitionStoreException {
    return loadBeanDefinitions(location, null);
}
public int loadBeanDefinitions(String location, Set<Resource> actualResources) throws
BeanDefinitionStoreException {
    //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 + "]: no ResourceLoader
available");
    }
    if (resourceLoader instanceof ResourcePatternResolver) {
        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).getResources(location);
            //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 location pattern [" +
location + "]);");
            }
            return loadCount;
        }
        catch (IOException ex) {
            throw new BeanDefinitionStoreException(
                "Could not resolve bean definition resource pattern [" + location + "]", ex);
        }
    }
    else {
        //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 [" + location +
"]);");
        }
        return loadCount;
    }
}
}

```

```

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

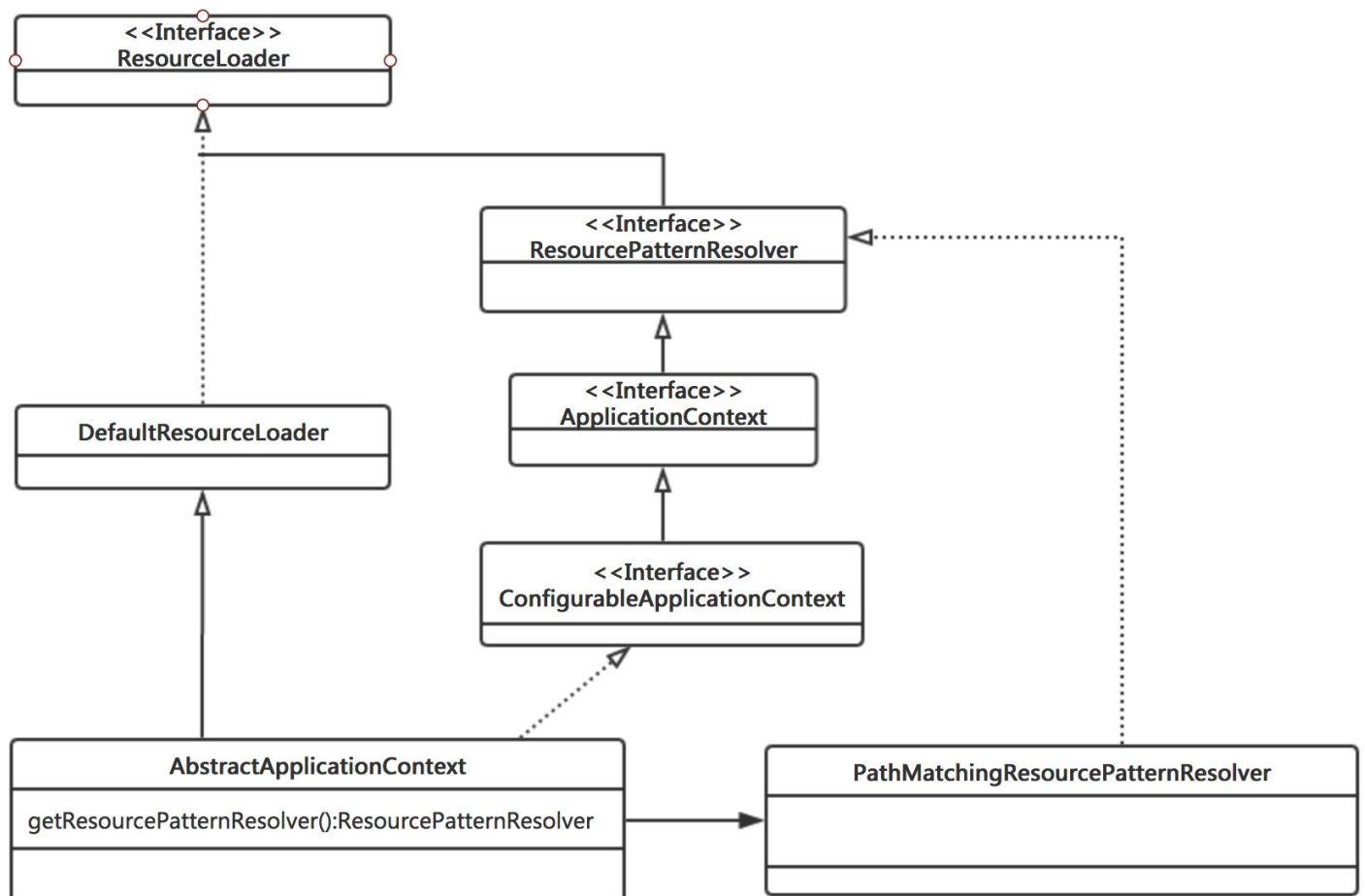
```

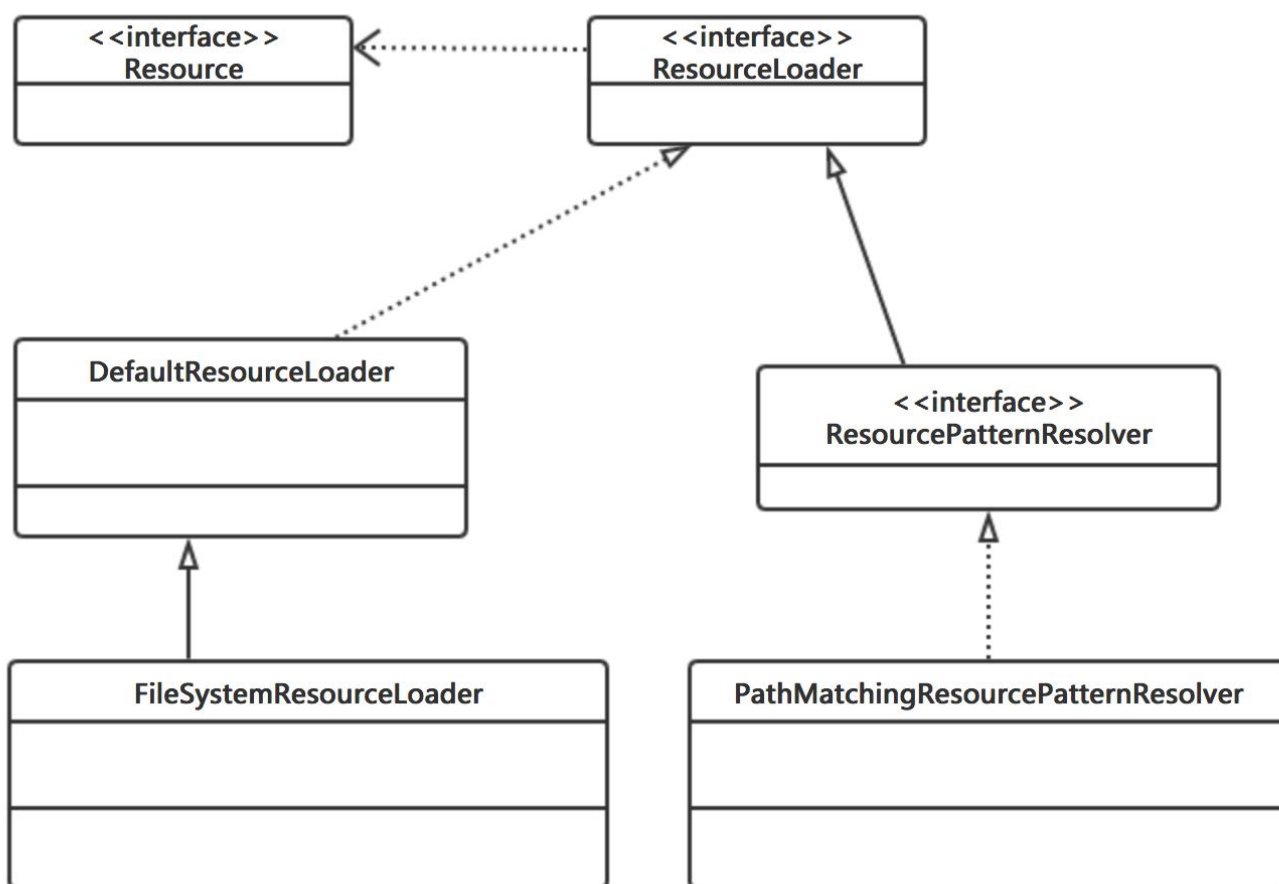
loadBeanDefinitions(Resource... resources)方法和上面分析的3个方法类似，同样也是调用XmlBeanDefinitionReader的loadBeanDefinitions方法。

从对AbstractBeanDefinitionReader的loadBeanDefinitions方法源码分析可以看出该方法做了以下两件事：

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。





看到第 8、16 行，结合上面的 ResourceLoader 与 ApplicationContext 的继承系图，可以知道此时调用的是 DefaultResourceLoader 中的 getSource() 方法定位 Resource，因为 FileSystemXmlApplicationContext 本身就是 DefaultResourceLoader 的实现类，所以此时又回到了 FileSystemXmlApplicationContext 中来。

7、资源加载器获取要读入的资源：

XmlBeanDefinitionReader 通过调用其父类 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
public Resource getResource(String location) {
    Assert.notNull(location, "Location must not be null");
    //如果是类路径的方式，那需要使用 ClassPathResource 来得到 bean 文件的资源对象
    if (location.startsWith(CLASSPATH_URL_PREFIX)) {
        return new ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.length()),
            getClassLoader());
    }
}

```

```

    try {
        // 如果是 URL 方式，使用 UrlResource 作为 bean 文件的资源对象
        URL url = new URL(location);
        return new UrlResource(url);
    }
    catch (MalformedURLException ex) {
    }
    //如果既不是 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);
}

```

`FileSystemXmlApplicationContext` 容器提供了 `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) {
    if (path != null && path.startsWith("/")) {
        path = path.substring(1);
    }
    //这里使用文件系统资源对象来定义 bean 文件
    return new FileSystemResource(path);
}

```

这样代码就回到了 `FileSystemXmlApplicationContext` 中来，他提供了 `FileSystemResource` 来完成从文件系统得到配置文件的资源定义。

这样，就可以从文件系统路径上对 IOC 配置文件进行加载 - 当然我们可以按照这个逻辑从任何地方加载，在 Spring 中我们看到它提供的各种资源抽象，比如 `ClassPathResource`, `UrlResource`, `FileSystemResource` 等来供我们使用。上面我们看到的是定位 `Resource` 的一个过程，而这只是加载过程的一部分。

8、`XmlBeanDefinitionReader` 加载 Bean 定义资源：

Bean 定义的 `Resource` 得到了

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
public int loadBeanDefinitions(Resource resource) throws BeanDefinitionStoreException {
    //将读入的 XML 资源进行特殊编码处理
    return loadBeanDefinitions(new EncodedResource(resource));
}

```

```

//这里是载入 XML 形式 Bean 定义资源文件方法
public int loadBeanDefinitions(EncodedResource encodedResource) throws
BeanDefinitionStoreException {
    .....
    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());
        }
        finally {
            //关闭从 Resource 中得到的 IO 流
            inputStream.close();
        }
    }
    .....
}

//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 {
        int validationMode = getValidationModeForResource(resource);
        //将 XML 文件转换为 DOM 对象，解析过程由 documentLoader 实现
        Document doc = this.documentLoader.loadDocument(
            inputSource, this.entityResolver, this.errorHandler, validationMode,
this.namespaceAware);
        //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 定义资源文件的最后一步是将 Bean 定义资源转换为 Document 对象，该过程由 documentLoader 实现

9、DocumentLoader 将 Bean 定义资源转换为 Document 对象：

DocumentLoader 将 Bean 定义资源转换成 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
public Document loadDocument(InputSource inputSource, EntityResolver entityResolver,
    ErrorHandler errorHandler, int validationMode, boolean namespaceAware) throws Exception
{
    //创建文件解析器工厂
    DocumentBuilderFactory factory = createDocumentBuilderFactory(validationMode,
namespaceAware);
    if (logger.isDebugEnabled()) {
        logger.debug("Using JAXP provider [" + factory.getClass().getName() + "]");
    }
    //创建文档解析器
    DocumentBuilder builder = createDocumentBuilder(factory, entityResolver, errorHandler);
    //解析 Spring 的 Bean 定义资源
    return builder.parse(inputSource);
}

protected DocumentBuilderFactory createDocumentBuilderFactory(int validationMode, boolean
namespaceAware)
    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) {
            factory.setNamespaceAware(true);
            try {
                factory.setAttribute(SCHEMA_LANGUAGE_ATTRIBUTE, XSD_SCHEMA_LANGUAGE);
            }
            catch (IllegalArgumentException ex) {
                ParserConfigurationException pcex = new ParserConfigurationException(
                    "Unable to validate using XSD: Your JAXP provider [" + factory +
                    "] does not support XML Schema. Are you running on Java 1.with Apache Crimson?
" +
                    "Upgrade to Apache Xerces (or Java 1.5) for full XSD support.");
                pcex.initCause(ex);
                throw pcex;
            }
        }
    }
    return factory;
}

```

该解析过程调用 JavaEE 标准的 JAXP 标准进行处理。

至此 Spring IOC 容器根据定位的 Bean 定义资源文件，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 定义资源文件转换为 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。

10、XmlBeanDefinitionReader 解析载入的 Bean 定义资源文件：

XmlBeanDefinitionReader 类中的 doLoadBeanDefinitions 方法是从特定 XML 文件中实际载入 Bean 定义资源的方法，该方法在载入 Bean 定义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
BeanDefinitionStoreException {
    //得到 BeanDefinitionDocumentReader 来对 xml 格式的 BeanDefinition 解析
    BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();
    //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;
}
//创建 BeanDefinitionDocumentReader 对象，解析 Document 对象
protected BeanDefinitionDocumentReader createBeanDefinitionDocumentReader() {
    return
    BeanDefinitionDocumentReader.class.cast(BeansUtils.instantiateClass(this.documentReaderClass));
}
```

Bean 定义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 定义资源文件转换得到 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 的 Bean 规则对 Document 对象进行解析。

按照 Spring 的 Bean 规则对 Document 对象解析的过程是在接口 BeanDefinitionDocumentReader 的实现类 DefaultBeanDefinitionDocumentReader 中实现的。

11、DefaultBeanDefinitionDocumentReader 对 Bean 定义的 Document 对象解析：

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
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();
    //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 delegate = createHelper(readerContext, root);
    //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, 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);
}

//创建 BeanDefinitionParserDelegate，用于完成真正的解析过程
protected BeanDefinitionParserDelegate createHelper(XmlReaderContext readerContext, Element root)
{
    BeanDefinitionParserDelegate delegate = new BeanDefinitionParserDelegate(readerContext);
    //BeanDefinitionParserDelegate 初始化 Document 根元素
    delegate.initDefaults(root);
    return delegate;
}

//使用 Spring 的 Bean 规则从 Document 的根元素开始进行 Bean 定义的 Document 对象
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
    //Bean 定义的 Document 对象使用了 Spring 默认的 XML 命名空间
    if (delegate.isDefaultNamespace(root)) {
        //获取 Bean 定义的 Document 对象根元素的所有子节点
        NodeList nl = root.getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            //获得 Document 节点是 XML 元素节点
            if (node instanceof Element) {
                Element ele = (Element) node;
                //Bean 定义的 Document 的元素节点使用的是 Spring 默认的 XML 命名空间
                if (delegate.isDefaultNamespace(ele)) {
                    //使用 Spring 的 Bean 规则解析元素节点
                }
            }
        }
    }
}
```

```

        parseDefaultElement(ele, delegate);
    }
    else {
        //没有使用 Spring 默认的 XML 命名空间，则使用用户自定义的解//析规则解析元素节点
        delegate.parseCustomElement(ele);
    }
}
}
}
else {
    //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
    //解析规则解析 Document 根节点
    delegate.parseCustomElement(root);
}
}
//使用 Spring 的 Bean 规则解析 Document 元素节点
private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {
    //如果元素节点是<Import>导入元素，进行导入解析
    if (delegate.nodeNameEquals(ele, IMPORT_ELEMENT)) {
        importBeanDefinitionResource(ele);
    }
    //如果元素节点是<Alias>别名元素，进行别名解析
    else if (delegate.nodeNameEquals(ele, ALIAS_ELEMENT)) {
        processAliasRegistration(ele);
    }
    //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，
    //按照 Spring 的 Bean 规则解析元素
    else if (delegate.nodeNameEquals(ele, BEAN_ELEMENT)) {
        processBeanDefinition(ele, delegate);
    }
}
//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
protected void importBeanDefinitionResource(Element ele) {
    //获取给定的导入元素的 location 属性
    String location = ele.getAttribute(RESOURCE_ATTRIBUTE);
    //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
    if (!StringUtils.hasText(location)) {
        getReaderContext().error("Resource location must not be empty", ele);
        return;
    }
    //使用系统变量值解析 location 属性值
    location = SystemPropertyUtils.resolvePlaceholders(location);
    Set<Resource> actualResources = new LinkedHashSet<Resource>(4);
    //标识给定的导入元素的 location 是否是绝对路径

```

```
boolean absoluteLocation = false;
try {
    absoluteLocation = ResourcePatternUtils.isUrl(location) ||
ResourceUtils.toURI(location).isAbsolute();
}
catch (URISyntaxException ex) {
    //给定的导入元素的 location 不是绝对路径
}
//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(location,
actualResources);
        if (logger.isDebugEnabled()) {
            logger.debug("Imported " + importCount + " bean definitions from URL location [" +
location + "]");
        }
    }
    catch (BeanDefinitionStoreException ex) {
        getReaderContext().error(
            "Failed to import bean definitions from URL location [" + location + "]", ele, ex);
    }
}
else {
    //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().createRelative(location);
        //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(relativeResource);
            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().toString();
            //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), actualResources);
        }
    }
}
```



```

    }
    if (logger.isDebugEnabled()) {
        logger.debug("Imported " + importCount + " bean definitions from relative location [" +
+ location + "]);
    }
}
catch (IOException ex) {
    getReaderContext().error("Failed to resolve current resource location", ele, ex);
}
catch (BeanDefinitionStoreException ex) {
    getReaderContext().error("Failed to import bean definitions from relative location [" +
location + "]",
        ele, ex);
}
}
Resource[] actResArray = actualResources.toArray(new Resource[actualResources.size()]);
//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));
}
//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));
}
}
//解析 Bean 定义资源 Document 对象的普通元素
protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {
    // 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 {
            //向 Spring IoC 容器注册解析得到的 Bean 定义，这是 Bean 定义向 IOC 容器注册的入口
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
getReaderContext().getRegistry());
        }
        catch (BeanDefinitionStoreException ex) {
            getReaderContext().error("Failed to register bean definition with name '" +
                bdHolder.getBeanName() + "'", ele, ex);
        }
        //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(bdHolder));
    }
}
}

```

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<Alias>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。

12、BeanDefinitionParserDelegate 解析 Bean 定义资源文件中的<Bean>元素：

Bean 定义资源文件中的<Import>和<Alias>元素解析在 DefaultBeanDefinitionDocumentReader 中已经完成，对 Bean 定义资源文件中使用最多的<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
public BeanDefinitionHolder parseBeanDefinitionElement(Element ele) {
    return parseBeanDefinitionElement(ele, null);
}

```

```

//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
//和别名属性
public BeanDefinitionHolder parseBeanDefinitionElement(Element ele, BeanDefinition
containingBean) {
    //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<String>();
    //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, BEAN_NAME_DELIMITERS);
        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, beanName,
containingBean);
    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);
        // 为解析的 Bean 使用别名注册时，为了向后兼容
// Spring 1.2/2.0，给别名添加类名后缀
        String beanClassName = beanDefinition.getBeanClassName();
        if (beanClassName != null &&
            beanName.startsWith(beanClassName) && beanName.length() >
beanClassName.length() &&
            !this.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {
            aliases.add(beanClassName);
        }
    }
    if (logger.isDebugEnabled()) {
        logger.debug("Neither XML 'id' nor 'name' specified - " +
            "using generated bean name [" + beanName + "]");
    }
}
catch (Exception ex) {
    error(ex.getMessage(), ele);
    return null;
}
}
String[] aliasesArray = StringUtils.toStringArray(aliases);
return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);
}
// 当解析出错时，返回 null
return null;
}

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

```

```

        parent = ele.getAttribute(PARENT_ATTRIBUTE);
    }
    //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)属性等
    parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);
    //为<Bean>元素解析的 Bean 设置 description 信息
    bd.setDescription(DomUtils.getChildElementValueByTagName(ele, DESCRIPTION_ELEMENT));
    //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, err);
}
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 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。

13、BeanDefinitionParserDelegate 解析<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 + "'", ele);
            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
public Object parsePropertyValue(Element ele, BeanDefinition bd, String propertyName) {
    String elementName = (propertyName != null) ?
        "<property> element for property '" + propertyName + "'" :
        "<constructor-arg> element";
    //获取<property>的所有子元素，只能是其中一种类型:ref,value,list 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)) {
            if (subElement != null) {
                error(elementName + " must not contain more than one sub-element", ele);
            }
            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' attribute OR
sub-element", ele);
}
//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_ATTRIBUTE));
        //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 {
        //property 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：

- a. ref 被封装为指向依赖对象一个引用。
- b. value 配置都会封装成一个字符串类型的对象。
- c. 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。

14、解析<property>元素的子元素：

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

```
//解析<property>元素中 ref,value 或者集合等子元素
public Object parsePropertySubElement(Element ele, BeanDefinition bd, String defaultValueType) {
    //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)) {
        //获取<property>元素中的 bean 属性值，引用其他解析的 Bean 的名称
        //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)) {
            //获取<property>元素中的 local 属性值，引用同一个 Xml 文件中配置
            //的 Bean 的 id，local 和 ref 不同，local 只能引用同一个配置文件中的 Bean
            refName = ele.getAttribute(LOCAL_REF_ATTRIBUTE);
            if (!StringUtils.hasLength(refName)) {
                //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', 'local' or 'parent' is required for <ref> element", ele);
                    return null;
                }
            }
        }
        //没有配置 ref 的目标属性值
        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)) {
    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 BeanDefinition 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。

15、解析<list>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 parseListElement(Element collectionEle, BeanDefinition bd) {
    //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<Object>(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, BeanDefinition bd, String
defaultElementType) {
    //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, defaultElementType));
        }
    }
}
```

```
    }  
  }  
}
```

经过对 Spring Bean 定义资源文件转换的 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 定义资源文件转换为 Spring IOC 所识别的数据结构——BeanDefinition，它是 Bean 定义资源文件中配置的 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 定义资源的解析后，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

16、解析过后的 BeanDefinition 在 IOC 容器中的注册：

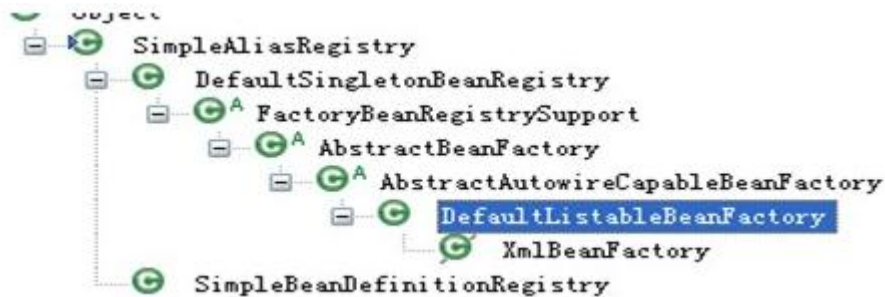
让我们继续跟踪程序的执行顺序，接下来会到我们第3步中分析DefaultBeanDefinitionDocumentReader对 Bean 定义转换的 Document 对象解析的流程中，在其 parseDefaultElement 方法中完成对 Document 对象的解析后得到封装 BeanDefinition 的 BeanDefinitionHolder 对象，然后调用 BeanDefinitionReaderUtils 的 registerBeanDefinition 方法向 IOC 容器注册解析的 Bean，BeanDefinitionReaderUtils 的注册的源码如下：

```
//将解析的 BeanDefinitionHolder 注册到容器中  
public static void registerBeanDefinition(BeansDefinitionHolder definitionHolder,  
BeansDefinitionRegistry registry)  
    throws BeansDefinitionStoreException {  
    //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());  
    //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。

17、DefaultListableBeanFactory 向 IOC 容器注册解析后的 BeanDefinition：

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<String,
BeanDefinition>();
//向 IoC 容器注册解析的 BeanDefiniton
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");
    //校验解析的 BeanDefiniton
    if (beanDefinition instanceof AbstractBeanDefinition) {
        try {
            ((AbstractBeanDefinition) beanDefinition).validate();
        }
        catch (BeanDefinitionValidationException ex) {
            throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription(),
beanName,
                "Validation of bean definition failed", ex);
        }
    }
    //注册的过程中需要线程同步，以保证数据的一致性
    synchronized (this.beanDefinitionMap) {
        Object oldBeanDefinition = this.beanDefinitionMap.get(beanName);
        //检查是否有同名的 BeanDefinition 已经在 IOC 容器中注册，如果已经注册，
        //并且不允许覆盖已注册的 Bean，则抛出注册失败异常
        if (oldBeanDefinition != null) {
            if (!this.allowBeanDefinitionOverriding) {
                throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription(),
beanName,
                    "Cannot register bean definition [" + beanDefinition + "] for bean '" +
beanName +
                    "': There is already [" + oldBeanDefinition + "] bound.");
            }
        }
    }
}
```

```

    }
    else { //如果允许覆盖，则同名的 Bean，后注册的覆盖先注册的
        if (this.logger.isInfoEnabled()) {
            this.logger.info("Overriding bean definition for bean '" + beanName +
                "': replacing [" + oldBeanDefinition + "] with [" + beanDefinition + "]);
        }
    }
}
//IOC 容器中没有已经注册同名的 Bean，按正常注册流程注册
else {
    this.beanDefinitionNames.add(beanName);
    this.frozenBeanDefinitionNames = null;
}
this.beanDefinitionMap.put(beanName, beanDefinition);
//重置所有已经注册过的 BeanDefinition 的缓存
resetBeanDefinition(beanName);
}
}

```

至此，Bean 定义资源文件中配置的 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。

总结：

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-他们都是为处理 BeanDefinitin 服务的，容器解析得到 BeanDefinitionIoC 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。

3. 然后我们就可以通过 BeanFactory 和 ApplicationContext 来享受到 SpringIOC 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。

在使用 SpringIOC 容器的时候我们还需要区别两个概念：

BeanFactory 和 FactoryBean，其中 BeanFactory 指的是 IOC 容器的编程抽象，比如 ApplicationContext, XmlBeanFactory 等，这些都是 IOC 容器的具体表现，需要使用什么样的容器由客户决定, 但 Spring 为我们提供了丰富的选择。FactoryBean 只是一个可以在 IOC 而容器中被管理的一个 bean, 是对各种处理过程和资源使用的抽象, FactoryBean 在需要时产生另一个对象，而不返回 FactoryBean 本身, 我们可以把它看成是一个抽象工厂，对它的调用返回的是工厂生产的产品。所有的 FactoryBean 都实现特殊的 org.springframework.beans.factory.FactoryBean 接口，当使用容器中 FactoryBean 的时候，该容器不会返回 FactoryBean 本身, 而是返回其生成的对象。Spring 包括了大部分的通用资源和服务访问抽象的 FactoryBean 的实现，其中包括: 对 JNDI 查询的处理，对代理对象的处理，对事务性代理的处理，对 RMI 代理的处理等，这些我们都可以看成是具体的工厂, 看成是 Spring 为我们建立好的工厂。也就是说 Spring 通过使用抽象工厂模式为我们准备了一系列工厂来生产一些特定的对象, 免除我们手工重复的工作，我们要使用时只需要在 IOC 容器里配置好就能很方便的使用了

5. 4、IOC 容器的依赖注入

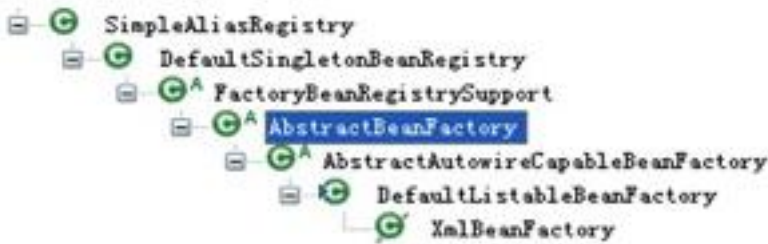
1、依赖注入发生的时间

当 Spring IOC 容器完成了 Bean 定义资源的定位、载入和解析注册以后，IOC 容器中已经管理类 Bean 定义的相关数据，但是此时 IOC 容器还没有对所管理的 Bean 进行依赖注入，依赖注入在以下两种情况发生：

(1). 用户第一次通过 getBean 方法向 IOC 容索要 Bean 时，IOC 容器触发依赖注入。

(2). 当用户在 Bean 定义资源中为<Bean>元素配置了 lazy-init 属性，即让容器在解析注册 Bean 定义时进行预实例化，触发依赖注入。

BeanFactory 接口定义了 Spring IOC 容器的基本功能规范，是 Spring IOC 容器所应遵守的最底层和最基本的编程规范。BeanFactory 接口中定义了几个 getBean 方法，就是用户向 IOC 容器索取管理的 Bean 的方法，我们通过分析其子类的具体实现，理解 Spring IOC 容器在用户索取 Bean 时如何完成依赖注入。



在 BeanFactory 中我们看到 `getBean (String...)` 函数，它的具体实现在 `AbstractBeanFactory` 中

2、`AbstractBeanFactory` 通过 `getBean` 向 IOC 容器获取被管理的 Bean:

`AbstractBeanFactory` 的 `getBean` 相关方法的源码如下:

```
//获取 IOC 容器中指定名称的 Bean
public Object getBean(String name) throws BeansException {
    //doGetBean 才是真正向 IoC 容器获取被管理 Bean 的过程
    return doGetBean(name, null, null, false);
}

//获取 IOC 容器中指定名称和类型的 Bean
public <T> T getBean(String name, Class<T> requiredType) throws BeansException {
    //doGetBean 才是真正向 IoC 容器获取被管理 Bean 的过程
    return doGetBean(name, requiredType, null, false);
}

//获取 IOC 容器中指定名称和参数的 Bean
public Object getBean(String name, Object... args) throws BeansException {
    //doGetBean 才是真正向 IoC 容器获取被管理 Bean 的过程
    return doGetBean(name, null, args, false);
}

//获取 IOC 容器中指定名称、类型和参数的 Bean
public <T> T getBean(String name, Class<T> requiredType, Object... args) throws BeansException {
    //doGetBean 才是真正向 IoC 容器获取被管理 Bean 的过程
    return doGetBean(name, requiredType, args, false);
}

//真正实现向 IOC 容器获取 Bean 的功能，也是触发依赖注入功能的地方
@SuppressWarnings("unchecked")
protected <T> T doGetBean(
    final String name, final Class<T> requiredType, final Object[] args, boolean typeCheckOnly)
    throws BeansException {
    //根据指定的名称获取被管理 Bean 的名称，剥离指定名称中对容器的相关依赖
    //如果指定的是别名，将别名转换为规范的 Bean 名称
    final String beanName = transformedBeanName(name);
    Object bean;
    //先从缓存中取是否已经有被创建过的单态类型的 Bean
    //对于单例模式的 Bean 整个 IOC 容器中只创建一次，不需要重复创建
```



```

Object sharedInstance = getSingleton(beanName);
//IOC 容器创建单例模式 Bean 实例对象
if (sharedInstance != null && args == null) {
    if (logger.isDebugEnabled()) {
        //如果指定名称的 Bean 在容器中已有单例模式的 Bean 被创建
        //直接返回已经创建的 Bean
        if (isSingletonCurrentlyInCreation(beanName)) {
            logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +
                "' that is not fully initialized yet - a consequence of a circular reference");
        }
        else {
            logger.debug("Returning cached instance of singleton bean '" + beanName + "'");
        }
    }
    //获取给定 Bean 的实例对象，主要是完成 FactoryBean 的相关处理
    //注意：BeanFactory 是管理容器中 Bean 的工厂，而 FactoryBean 是
    //创建创建对象的工厂 Bean，两者之间有区别
    bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
}
else {
    //缓存没有正在创建的单例模式 Bean
    //缓存中已经有已经创建的原型模式 Bean
    //但是由于循环引用的问题导致实例化对象失败
    if (isPrototypeCurrentlyInCreation(beanName)) {
        throw new BeanCurrentlyInCreationException(beanName);
    }
    //对 IOC 容器中是否存在指定名称的 BeanDefinition 进行检查，首先检查是否
    //能在当前的 BeanFactory 中获取的所需要的 Bean，如果不能则委托当前容器
    //的父级容器去查找，如果还是找不到则沿着容器的继承体系向父级容器查找
    BeanFactory parentBeanFactory = getParentBeanFactory();
    //当前容器的父级容器存在，且当前容器中不存在指定名称的 Bean
    if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
        //解析指定 Bean 名称的原始名称
        String nameToLookup = originalBeanName(name);
        if (args != null) {
            //委派父级容器根据指定名称和显式的参数查找
            return (T) parentBeanFactory.getBean(nameToLookup, args);
        }
        else {
            //委派父级容器根据指定名称和类型查找
            return parentBeanFactory.getBean(nameToLookup, requiredType);
        }
    }
}
//创建的 Bean 是否需要进行类型验证，一般不需要

```

```

if (!typeCheckOnly) {
    //向容器标记指定的 Bean 已经被创建
    markBeanAsCreated(beanName);
}
//根据指定 Bean 名称获取其父级的 Bean 定义
//主要解决 Bean 继承时子类合并父类公共属性问题
final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
checkMergedBeanDefinition(mbd, beanName, args);
//获取当前 Bean 所有依赖 Bean 的名称
String[] dependsOn = mbd.getDependsOn();
//如果当前 Bean 有依赖 Bean
if (dependsOn != null) {
    for (String dependsOnBean : dependsOn) {
        //递归调用 getBean 方法，获取当前 Bean 的依赖 Bean
        getBean(dependsOnBean);
        //把被依赖 Bean 注册给当前依赖的 Bean
        registerDependentBean(dependsOnBean, beanName);
    }
}
//创建单例模式 Bean 的实例对象
if (mbd.isSingleton()) {
    //这里使用了一个匿名内部类，创建 Bean 实例对象，并且注册给所依赖的对象
    sharedInstance = getSingleton(beanName, new ObjectFactory() {
        public Object getObject() throws BeansException {
            try {
                //创建一个指定 Bean 实例对象，如果有父级继承，则合并子类和父类的定义
                return createBean(beanName, mbd, args);
            }
            catch (BeansException ex) {
                //显式地从容器单例模式 Bean 缓存中清除实例对象
                destroySingleton(beanName);
                throw ex;
            }
        }
    });
    //获取给定 Bean 的实例对象
    bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
}
//IOC 容器创建原型模式 Bean 实例对象
else if (mbd.isPrototype()) {
    //原型模式(Prototype)是每次都会创建一个新的对象
    Object prototypeInstance = null;
    try {
        //回调 beforePrototypeCreation 方法，默认的功能是注册当前创//建的原型对象

```

了

```
        beforePrototypeCreation(beanName);
        //创建指定 Bean 对象实例
        prototypeInstance = createBean(beanName, mbd, args);
    }
    finally {
        //回调 afterPrototypeCreation 方法，默认的功能告诉 IoC 容器指定 Bean 的原型对象不再创建

        afterPrototypeCreation(beanName);
    }
    //获取给定 Bean 的实例对象
    bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
}
//要创建的 Bean 既不是单例模式，也不是原型模式，则根据 Bean 定义资源中
//配置的生命周期范围，选择实例化 Bean 的合适方法，这种在 Web 应用程序中
//比较常用，如：request、session、application 等生命周期
else {
    String scopeName = mbd.getScope();
    final Scope scope = this.scopes.get(scopeName);
    //Bean 定义资源中没有配置生命周期范围，则 Bean 定义不合法
    if (scope == null) {
        throw new IllegalStateException("No Scope registered for scope '" + scopeName + "'");
    }
    try {
        //这里又使用了一个匿名内部类，获取一个指定生命周期范围的实例
        Object scopedInstance = scope.get(beanName, new ObjectFactory() {
            public Object getObject() throws BeansException {
                beforePrototypeCreation(beanName);
                try {
                    return createBean(beanName, mbd, args);
                }
                finally {
                    afterPrototypeCreation(beanName);
                }
            }
        });
        //获取给定 Bean 的实例对象
        bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);
    }
    catch (IllegalStateException ex) {
        throw new BeanCreationException(beanName,
            "Scope '" + scopeName + "' is not active for the current thread; " +
            "consider defining a scoped proxy for this bean if you intend to refer to"
            + "it from a singleton",
            ex);
    }
}
```

```

        }
    }
}
//对创建的 Bean 实例对象进行类型检查
if (requiredType != null && bean != null && !requiredType.isAssignableFrom(bean.getClass()))
{
    throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
}
return (T) bean;
}

```

通过上面对向 IOC 容器获取 Bean 方法的分析，我们可以看到在 Spring 中，如果 Bean 定义的单例模式 (Singleton)，则容器在创建之前先从缓存中查找，以确保整个容器中只存在一个实例对象。如果 Bean 定义的是原型模式 (Prototype)，则容器每次都会创建一个新的实例对象。除此之外，Bean 定义还可以扩展为指定其生命周期范围。

上面的源码只是定义了根据 Bean 定义的模式，采取的不同创建 Bean 实例对象的策略，具体的 Bean 实例对象的创建过程由实现了 ObejctFactory 接口的匿名内部类的 createBean 方法完成，ObejctFactory 使用委派模式，具体的 Bean 实例创建过程交由其实现类 AbstractAutowireCapableBeanFactory 完成，我们继续分析 AbstractAutowireCapableBeanFactory 的 createBean 方法的源码，理解其创建 Bean 实例的具体实现过程。

3、AbstractAutowireCapableBeanFactory 创建 Bean 实例对象：

AbstractAutowireCapableBeanFactory 类实现了 ObejctFactory 接口，创建容器指定的 Bean 实例对象，同时还对创建的 Bean 实例对象进行初始化处理。其创建 Bean 实例对象的方法源码如下：

```

//创建 Bean 实例对象
protected Object createBean(final String beanName, final RootBeanDefinition mbd, final Object[]
args)
    throws BeanCreationException {
    if (logger.isDebugEnabled()) {
        logger.debug("Creating instance of bean '" + beanName + "'");
    }
    //判断需要创建的 Bean 是否可以实例化，即是否可以通过当前的类加载器加载
    resolveBeanClass(mbd, beanName);
    //校验和准备 Bean 中的方法覆盖
    try {
        mbd.prepareMethodOverrides();
    }
    catch (BeanDefinitionValidationException ex) {
        throw new BeanDefinitionStoreException(mbd.getResourceDescription(),

```

```

        beanName, "Validation of method overrides failed", ex);
    }
    try {
        //如果 Bean 配置了初始化前和初始化后的处理器，则试图返回一个需要创建 Bean 的代理对象
        Object bean = resolveBeforeInstantiation(beanName, mbd);
        if (bean != null) {
            return bean;
        }
    }
    catch (Throwable ex) {
        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
            "BeanPostProcessor before instantiation of bean failed", ex);
    }
    //创建 Bean 的入口
    Object beanInstance = doCreateBean(beanName, mbd, args);
    if (logger.isDebugEnabled()) {
        logger.debug("Finished creating instance of bean '" + beanName + "'");
    }
    return beanInstance;
}

//真正创建 Bean 的方法
protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[]
args) {
    //封装被创建的 Bean 对象
    BeanWrapper instanceWrapper = null;
    if (mbd.isSingleton()){//单例模式的 Bean，先从容器中缓存中获取同名 Bean
        instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
    }
    if (instanceWrapper == null) {
        //创建实例对象
        instanceWrapper = createBeanInstance(beanName, mbd, args);
    }
    final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstance() : null);
    //获取实例化对象的类型
    Class beanType = (instanceWrapper != null ? instanceWrapper.getWrappedClass() : null);
    //调用 PostProcessor 后置处理器
    synchronized (mbd.postProcessingLock) {
        if (!mbd.postProcessed) {
            applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
            mbd.postProcessed = true;
        }
    }
}

// Eagerly cache singletons to be able to resolve circular references

```

```

//向容器中缓存单例模式的 Bean 对象，以防循环引用
boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&
    isSingletonCurrentlyInCreation(beanName));
if (earlySingletonExposure) {
    if (logger.isDebugEnabled()) {
        logger.debug("Eagerly caching bean '" + beanName +
            "' to allow for resolving potential circular references");
    }
    //这里是一个匿名内部类，为了防止循环引用，尽早持有对象的引用
    addSingletonFactory(beanName, new ObjectFactory() {
        public Object getObject() throws BeansException {
            return getEarlyBeanReference(beanName, mbd, bean);
        }
    });
}
//Bean 对象的初始化，依赖注入在此触发
//这个 exposedObject 在初始化完成之后返回作为依赖注入完成后的 Bean
Object exposedObject = bean;
try {
    //将 Bean 实例对象封装，并且 Bean 定义中配置的属性值赋值给实例对象
    populateBean(beanName, mbd, instanceWrapper);
    if (exposedObject != null) {
        //初始化 Bean 对象
        exposedObject = initializeBean(beanName, exposedObject, mbd);
    }
}
catch (Throwable ex) {
    if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException)
ex).getBeanName())) {
        throw (BeanCreationException) ex;
    }
    else {
        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
"Initialization of bean failed", ex);
    }
}
if (earlySingletonExposure) {
    //获取指定名称的已注册的单例模式 Bean 对象
    Object earlySingletonReference = getSingleton(beanName, false);
    if (earlySingletonReference != null) {
        //根据名称获取的已注册的 Bean 和正在实例化的 Bean 是同一个
        if (exposedObject == bean) {
            //当前实例化的 Bean 初始化完成
            exposedObject = earlySingletonReference;
        }
    }
}

```

```

    }
    //当前 Bean 依赖其他 Bean，并且当发生循环引用时不允许新创建实例对象
    else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {
        String[] dependentBeans = getDependentBeans(beanName);
        Set<String> actualDependentBeans = new
LinkedHashSet<String>(dependentBeans.length);
        //获取当前 Bean 所依赖的其他 Bean
        for (String dependentBean : dependentBeans) {
            //对依赖 Bean 进行类型检查
            if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
                actualDependentBeans.add(dependentBean);
            }
        }
        if (!actualDependentBeans.isEmpty()) {
            throw new BeanCurrentlyInCreationException(beanName,
                "Bean with name '" + beanName + "' has been injected into other beans
[" +
                StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +
                "] in its raw version as part of a circular reference, but has eventually
been " +
                "wrapped. This means that said other beans do not use the final version
of the " +
                "bean. This is often the result of over-eager type matching - consider
using " +
                "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for
example.");
        }
    }
}
//注册完成依赖注入的 Bean
try {
    registerDisposableBeanIfNecessary(beanName, bean, mbd);
}
catch (BeanDefinitionValidationException ex) {
    throw new BeanCreationException(mbd.getResourceDescription(), beanName, "Invalid
destruction signature", ex);
}
return exposedObject;
}

```

通过对方法源码的分析，我们看到具体的依赖注入实现在以下两个方法中：

(1).createBeanInstance: 生成 Bean 所包含的 java 对象实例。

(2).populateBean : 对 Bean 属性的依赖注入进行处理。

下面继续分析这两个方法的代码实现。

4、createBeanInstance 方法创建 Bean 的 java 实例对象:

在 createBeanInstance 方法中, 根据指定的初始化策略, 使用静态工厂、工厂方法或者容器的自动装配特性生成 java 实例对象, 创建对象的源码如下:

```
//创建 Bean 的实例对象
protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, Object[] args)
{
    //检查确认 Bean 是可实例化的
    Class beanClass = resolveBeanClass(mbd, beanName);
    //使用工厂方法对 Bean 进行实例化
    if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers())
    && !mbd.isNonPublicAccessAllowed()) {
        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
            "Bean class isn't public, and non-public access not allowed: " +
            beanClass.getName());
    }
    if (mbd.getFactoryMethodName() != null) {
        //调用工厂方法实例化
        return instantiateUsingFactoryMethod(beanName, mbd, args);
    }
    //使用容器的自动装配方法进行实例化
    boolean resolved = false;
    boolean autowireNecessary = false;
    if (args == null) {
        synchronized (mbd.constructorArgumentLock) {
            if (mbd.resolvedConstructorOrFactoryMethod != null) {
                resolved = true;
                autowireNecessary = mbd.constructorArgumentsResolved;
            }
        }
    }
    if (resolved) {
        if (autowireNecessary) {
            //配置了自动装配属性, 使用容器的自动装配实例化
            //容器的自动装配是根据参数类型匹配 Bean 的构造方法
            return autowireConstructor(beanName, mbd, null, null);
        }
        else {

```



```

        //使用默认的空参构造方法实例化
        return instantiateBean(beanName, mbd);
    }
}

//使用 Bean 的构造方法进行实例化
Constructor[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);
if (ctors != null ||
    mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_CONSTRUCTOR ||
    mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args)) {
    //使用容器的自动装配特性，调用匹配的构造方法实例化
    return autowireConstructor(beanName, mbd, ctors, args);
}

//使用默认的空参构造方法实例化
return instantiateBean(beanName, mbd);
}

//使用默认的空参构造方法实例化 Bean 对象
protected BeanWrapper instantiateBean(final String beanName, final RootBeanDefinition mbd) {
    try {
        Object beanInstance;
        final BeanFactory parent = this;
        //获取系统的安全管理接口，JDK 标准的安全管理 API
        if (System.getSecurityManager() != null) {
            //这里是一个匿名内置类，根据实例化策略创建实例对象
            beanInstance = AccessController.doPrivileged(new PrivilegedAction<Object>() {
                public Object run() {
                    return getInstantiationStrategy().instantiate(mbd, beanName, parent);
                }
            }, getAccessControlContext());
        }
        else {
            //将实例化的对象封装起来
            beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);
        }
        BeanWrapper bw = new BeanWrapperImpl(beanInstance);
        initBeanWrapper(bw);
        return bw;
    }
    catch (Throwable ex) {
        throw new BeanCreationException(mbd.getResourceDescription(), beanName, "Instantiation of
bean failed", ex);
    }
}
}

```

经过对上面的代码分析，我们可以看出，对使用工厂方法和自动装配特性的 Bean 的实例化相当比较清楚，调用相应的工厂方法或者参数匹配的构造方法即可完成实例化对象的工作，但是对于我们最常使用的默认无参构造方法就需要使用相应的初始化策略(JDK 的反射机制或者 CGLIB)来进行初始化了，在方法 `getInstantiationStrategy().instantiate` 中就具体实现类使用初始策略实例化对象。

5、SimpleInstantiationStrategy 类使用默认的无参构造方法创建 Bean 实例化对象：

在使用默认的无参构造方法创建 Bean 的实例化对象时，方法 `getInstantiationStrategy().instantiate` 调用了 `SimpleInstantiationStrategy` 类中的实例化 Bean 的方法，其源码如下：

```
//使用初始化策略实例化 Bean 对象
public Object instantiate(RootBeanDefinition beanDefinition, String beanName, BeanFactory owner)
{
    //如果 Bean 定义中没有方法覆盖，则就不需要 CGLIB 父类类的方法
    if (beanDefinition.getMethodOverrides().isEmpty()) {
        Constructor<?> constructorToUse;
        synchronized (beanDefinition.constructorArgumentLock) {
            //获取对象的构造方法或工厂方法
            constructorToUse = (Constructor<?>)
beanDefinition.resolvedConstructorOrFactoryMethod;
            //如果没有构造方法且没有工厂方法
            if (constructorToUse == null) {
                //使用 JDK 的反射机制，判断要实例化的 Bean 是否是接口
                final Class clazz = beanDefinition.getBeanClass();
                if (clazz.isInterface()) {
                    throw new BeanInstantiationException(clazz, "Specified class is an interface");
                }
                try {
                    if (System.getSecurityManager() != null) {
                        //这里是一个匿名内置类，使用反射机制获取 Bean 的构造方法
                        constructorToUse = AccessController.doPrivileged(new
PrivilegedExceptionAction<Constructor>() {
                            public Constructor run() throws Exception {
                                return clazz.getDeclaredConstructor((Class[]) null);
                            }
                        });
                    }
                    else {
                        constructorToUse = clazz.getDeclaredConstructor((Class[]) null);
                    }
                    beanDefinition.resolvedConstructorOrFactoryMethod = constructorToUse;
                }
                catch (Exception ex) {
                    throw new BeanInstantiationException(clazz, "No default constructor found",
```

```

ex);
        }
    }
}
//使用 BeanUtils 实例化，通过反射机制调用“构造方法.newInstance(arg)”来进行实例化
return BeanUtils.instantiateClass(constructorToUse);
}
else {
    //使用 CGLIB 来实例化对象
    return instantiateWithMethodInjection(beanDefinition, beanName, owner);
}
}
}

```

通过上面的代码分析，我们看到了如果 Bean 有方法被覆盖了，则使用 JDK 的反射机制进行实例化，否则，使用 CGLIB 进行实例化。

instantiateWithMethodInjection 方法调用 SimpleInstantiationStrategy 的子类 CglibSubclassingInstantiationStrategy 使用 CGLIB 来进行初始化，其源码如下：

```

//使用 CGLIB 进行 Bean 对象实例化
public Object instantiate(Constructor ctor, Object[] args) {
    //CGLIB 中的类
    Enhancer enhancer = new Enhancer();
    //将 Bean 本身作为其基类
    enhancer.setSuperclass(this.beanDefinition.getBeanClass());
    enhancer.setCallbackFilter(new CallbackFilterImpl());
    enhancer.setCallbacks(new Callback[] {
        NoOp.INSTANCE,
        new LookupOverrideMethodInterceptor(),
        new ReplaceOverrideMethodInterceptor()
    });
    //使用 CGLIB 的 create 方法生成实例对象
    return (ctor == null) ?
        enhancer.create() :
        enhancer.create(ctor.getParameterTypes(), args);
}

```

CGLIB 是一个常用的字节码生成器的类库，它提供了一系列 API 实现 java 字节码的生成和转换功能。我们在学习 JDK 的动态代理时都知道，JDK 的动态代理只能针对接口，如果一个类没有实现任何接口，要对其进行动态代理只能使用 CGLIB。

6、populateBean 方法对 Bean 属性的依赖注入：

在第 3 步的分析中我们已经了解到 Bean 的依赖注入分为以下两个过程：

(1).createBeanInstance: 生成 Bean 所包含的 java 对象实例。

(2).populateBean : 对 Bean 属性的依赖注入进行处理。

第 4、5 步中我们已经分析了容器初始化生成 Bean 所包含的 Java 实例对象的过程，现在我们继续分析生成对象后，Spring IOC 容器是如何将 Bean 的属性依赖关系注入 Bean 实例对象中并设置好的，属性依赖注入的代码如下：

```
//将 Bean 属性设置到生成的实例对象上
protected void populateBean(String beanName, AbstractBeanDefinition mbd, BeanWrapper bw) {
    //获取容器在解析 Bean 定义资源时为 BeanDefinition 中设置的属性值
    PropertyValues pvs = mbd.getPropertyValues();
    //实例对象为 null
    if (bw == null) {
        //属性值不为空
        if (!pvs.isEmpty()) {
            throw new BeanCreationException(
                mbd.getResourceDescription(), beanName, "Cannot apply property values to null
instance");
        }
    }
    else {
        //实例对象为 null，属性值也为空，不需要设置属性值，直接返回
        return;
    }
}

//在设置属性之前调用 Bean 的 PostProcessor 后置处理器
boolean continueWithPropertyPopulation = true;
if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
    for (BeanPostProcessor bp : getBeanPostProcessors()) {
        if (bp instanceof InstantiationAwareBeanPostProcessor) {
            InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor)
bp;

            if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
                continueWithPropertyPopulation = false;
                break;
            }
        }
    }
}

if (!continueWithPropertyPopulation) {
    return;
}

//依赖注入开始，首先处理 autowire 自动装配的注入
if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME ||
```

```

        mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
//对 autowire 自动装配的处理，根据 Bean 名称自动装配注入
if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME) {
    autowireByName(beanName, mbd, bw, newPvs);
}
//根据 Bean 类型自动装配注入
if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
    autowireByType(beanName, mbd, bw, newPvs);
}
pvs = newPvs;
}
//检查容器是否持有用于处理单例模式 Bean 关闭时的后置处理器
boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
//Bean 实例对象没有依赖，即没有继承基类
boolean needsDepCheck = (mbd.getDependencyCheck() !=
RootBeanDefinition.DEPENDENCY_CHECK_NONE);
if (hasInstAwareBpps || needsDepCheck) {
    //从实例对象中提取属性描述符
    PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw);
    if (hasInstAwareBpps) {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof InstantiationAwareBeanPostProcessor) {
                InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor) bp;
                //使用 BeanPostProcessor 处理器处理属性值
                pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(),
beanName);
                if (pvs == null) {
                    return;
                }
            }
        }
    }
    if (needsDepCheck) {
        //为要设置的属性进行依赖检查
        checkDependencies(beanName, mbd, filteredPds, pvs);
    }
}
//对属性进行注入
applyPropertyValues(beanName, mbd, bw, pvs);
}
//解析并注入依赖属性的过程
protected void applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper bw,

```

```

PropertyValues pvs) {
    if (pvs == null || pvs.isEmpty()) {
        return;
    }
    //封装属性值
    MutablePropertyValues mpvs = null;
    List<PropertyValue> original;
    if (System.getSecurityManager() != null) {
        if (bw instanceof BeanWrapperImpl) {
            //设置安全上下文，JDK 安全机制
            ((BeanWrapperImpl) bw).setSecurityContext(getAccessControlContext());
        }
    }
    if (pvs instanceof MutablePropertyValues) {
        mpvs = (MutablePropertyValues) pvs;
        //属性值已经转换
        if (mpvs.isConverted()) {
            try {
                //为实例化对象设置属性值
                bw.setPropertyValues(mpvs);
                return;
            }
            catch (BeansException ex) {
                throw new BeanCreationException(
                    mbd.getResourceDescription(), beanName, "Error setting property values",
ex);
            }
        }
        //获取属性值对象的原始类型值
        original = mpvs.getPropertyValueList();
    }
    else {
        original = Arrays.asList(pvs.getPropertyValues());
    }
    //获取用户自定义的类型转换
    TypeConverter converter = getCustomTypeConverter();
    if (converter == null) {
        converter = bw;
    }
    //创建一个 Bean 定义属性值解析器，将 Bean 定义中的属性值解析为 Bean 实例对象的实际值
    BeanDefinitionValueResolver valueResolver = new BeanDefinitionValueResolver(this, beanName,
mbd, converter);
    //为属性的解析值创建一个拷贝，将拷贝的数据注入到实例对象中
    List<PropertyValue> deepCopy = new ArrayList<PropertyValue>(original.size());

```

```

boolean resolveNecessary = false;
for (PropertyValue pv : original) {
    //属性值不需要转换
    if (pv.isConverted()) {
        deepCopy.add(pv);
    }
    //属性值需要转换
    else {
        String propertyName = pv.getName();
        //原始的属性值，即转换之前的属性值
        Object originalValue = pv.getValue();
        //转换属性值，例如将引用转换为 IOC 容器中实例化对象引用
        Object resolvedValue = valueResolver.resolveValueIfNecessary(pv, originalValue);
        //转换之后的属性值
        Object convertedValue = resolvedValue;
        //属性值是否可以转换
        boolean convertible = bw.isWritableProperty(propertyName) &&
            !PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);
        if (convertible) {
            //使用用户自定义的类型转换器转换属性值
            convertedValue = convertForProperty(resolvedValue, propertyName, bw, converter);
        }
        //存储转换后的属性值，避免每次属性注入时的转换工作
        if (resolvedValue == originalValue) {
            if (convertible) {
                //设置属性转换之后的值
                pv.setConvertedValue(convertedValue);
            }
            deepCopy.add(pv);
        }
        //属性是可转换的，且属性原始值是字符串类型，且属性的原始类型值不是
        //动态生成的字符串，且属性的原始值不是集合或者数组类型
        else if (convertible && originalValue instanceof TypedStringValue &&
            !((TypedStringValue) originalValue).isDynamic() &&
            !(convertedValue instanceof Collection ||
ObjectUtils.isArray(convertedValue))) {
            pv.setConvertedValue(convertedValue);
            deepCopy.add(pv);
        }
        else {
            resolveNecessary = true;
            //重新封装属性的值
            deepCopy.add(new PropertyValue(pv, convertedValue));
        }
    }
}

```

```

    }
}
if (mpvs != null && !resolveNecessary) {
    //标记属性值已经转换过
    mpvs.setConverted();
}
//进行属性依赖注入
try {
    bw.setPropertyValues(new MutablePropertyValues(deepCopy));
}
catch (BeansException ex) {
    throw new BeanCreationException(
        mbd.getResourceDescription(), beanName, "Error setting property values", ex);
}
}
}

```

分析上述代码，我们可以看出，对属性的注入过程分以下两种情况：

- (1). 属性值类型不需要转换时，不需要解析属性值，直接准备进行依赖注入。
- (2). 属性值需要进行类型转换时，如对其他对象的引用等，首先需要解析属性值，然后对解析后的属性值进行依赖注入。

对属性值的解析是在 BeanDefinitionValueResolver 类中的 resolveValueIfNecessary 方法中进行的，对属性值的依赖注入是通过 bw.setPropertyValues 方法实现的，在分析属性值的依赖注入之前，我们先分析一下对属性值的解析过程。

7、BeanDefinitionValueResolver 解析属性值：

当容器在对属性进行依赖注入时，如果发现属性值需要进行类型转换，如属性值是容器中另一个 Bean 实例对象的引用，则容器首先需要根据属性值解析出所引用的对象，然后才能将该引用对象注入到目标实例对象的属性上去，对属性进行解析的由 resolveValueIfNecessary 方法实现，其源码如下：

```

//解析属性值，对注入类型进行转换
public Object resolveValueIfNecessary(Object argName, Object value) {
    //对引用类型的属性进行解析
    if (value instanceof RuntimeBeanReference) {
        RuntimeBeanReference ref = (RuntimeBeanReference) value;
        //调用引用类型属性的解析方法
        return resolveReference(argName, ref);
    }
    //对属性值是引用容器中另一个 Bean 名称的解析
    else if (value instanceof RuntimeBeanNameReference) {
        String refName = ((RuntimeBeanNameReference) value).getBeanName();
    }
}

```



```

        refName = String.valueOf(evaluate(refName));
        //从容器中获取指定名称的 Bean
        if (!this.beanFactory.containsBean(refName)) {
            throw new BeanDefinitionStoreException(
                "Invalid bean name '" + refName + "' in bean reference for " + argName);
        }
        return refName;
    }
    //对 Bean 类型属性的解析，主要是 Bean 中的内部类
    else if (value instanceof BeanDefinitionHolder) {
        BeanDefinitionHolder bdHolder = (BeanDefinitionHolder) value;
        return resolveInnerBean(argName, bdHolder.getBeanName(), bdHolder.getBeanDefinition());
    }
    else if (value instanceof BeanDefinition) {
        BeanDefinition bd = (BeanDefinition) value;
        return resolveInnerBean(argName, "(inner bean)", bd);
    }
    //对集合数组类型的属性解析
    else if (value instanceof ManagedArray) {
        ManagedArray array = (ManagedArray) value;
        //获取数组的类型
        Class elementType = array.resolvedElementType;
        if (elementType == null) {
            //获取数组元素的类型
            String elementTypeName = array.getElementTypeName();
            if (StringUtils.hasText(elementTypeName)) {
                try {
                    //使用反射机制创建指定类型的对象
                    elementType = ClassUtils.forName(elementTypeName,
this.beanFactory.getBeanClassLoader());
                    array.resolvedElementType = elementType;
                }
                catch (Throwable ex) {
                    throw new BeanCreationException(
                        this.beanDefinition.getResourceDescription(), this.beanName,
                        "Error resolving array type for " + argName, ex);
                }
            }
        }
        //没有获取到数组的类型，也没有获取到数组元素的类型
        //则直接设置数组的类型为 Object
        else {
            elementType = Object.class;
        }
    }
}

```

```

        //创建指定类型的数组
        return resolveManagedArray(argName, (List<?>) value, elementType);
    }
    //解析 list 类型的属性值
    else if (value instanceof ManagedList) {
        return resolveManagedList(argName, (List<?>) value);
    }
    //解析 set 类型的属性值
    else if (value instanceof ManagedSet) {
        return resolveManagedSet(argName, (Set<?>) value);
    }
    //解析 map 类型的属性值
    else if (value instanceof ManagedMap) {
        return resolveManagedMap(argName, (Map<?, ?>) value);
    }
    //解析 props 类型的属性值, props 其实就是 key 和 value 均为字符串的 map
    else if (value instanceof ManagedProperties) {
        Properties original = (Properties) value;
        //创建一个拷贝, 用于作为解析后的返回值
        Properties copy = new Properties();
        for (Map.Entry propEntry : original.entrySet()) {
            Object propKey = propEntry.getKey();
            Object propValue = propEntry.getValue();
            if (propKey instanceof TypedStringValue) {
                propKey = evaluate((TypedStringValue) propKey);
            }
            if (propValue instanceof TypedStringValue) {
                propValue = evaluate((TypedStringValue) propValue);
            }
            copy.put(propKey, propValue);
        }
        return copy;
    }
    //解析字符串类型的属性值
    else if (value instanceof TypedStringValue) {
        TypedStringValue typedStringValue = (TypedStringValue) value;
        Object valueObject = evaluate(typedStringValue);
        try {
            //获取属性的目标类型
            Class<?> resolvedTargetType = resolveTargetType(typedStringValue);
            if (resolvedTargetType != null) {
                //对目标类型的属性进行解析, 递归调用
                return this.typeConverter.convertIfNecessary(valueObject, resolvedTargetType);
            }
        }
    }

```

```

        //没有获取到属性的目标对象，则按 Object 类型返回
        else {
            return valueObject;
        }
    }
    catch (Throwable ex) {
        throw new BeanCreationException(
            this.beanDefinition.getResourceDescription(), this.beanName,
            "Error converting typed String value for " + argName, ex);
    }
}
else {
    return evaluate(value);
}
}

//解析引用类型的属性值
private Object resolveReference(Object argName, RuntimeBeanReference ref) {
    try {
        //获取引用的 Bean 名称
        String refName = ref.getBeanName();
        refName = String.valueOf(evaluate(refName));
        //如果引用的对象在父类容器中，则从父类容器中获取指定的引用对象
        if (ref.isToParent()) {
            if (this.beanFactory.getParentBeanFactory() == null) {
                throw new BeanCreationException(
                    this.beanDefinition.getResourceDescription(), this.beanName,
                    "Can't resolve reference to bean '" + refName +
                    "' in parent factory: no parent factory available");
            }
            return this.beanFactory.getParentBeanFactory().getBean(refName);
        }
        //从当前的容器中获取指定的引用 Bean 对象，如果指定的 Bean 没有被实例化
        //则会递归触发引用 Bean 的初始化和依赖注入
        else {
            Object bean = this.beanFactory.getBean(refName);
            //将当前实例化对象的依赖引用对象
            this.beanFactory.registerDependentBean(refName, this.beanName);
            return bean;
        }
    }
    catch (BeansException ex) {
        throw new BeanCreationException(
            this.beanDefinition.getResourceDescription(), this.beanName,
            "Cannot resolve reference to bean '" + ref.getBeanName() + "' while setting " +

```

```

argName, ex);
    }
}

//解析 array 类型的属性
private Object resolveManagedArray(Object argName, List<?> ml, Class elementType) {
    //创建一个指定类型的数组，用于存放和返回解析后的数组
    Object resolved = Array.newInstance(elementType, ml.size());
    for (int i = 0; i < ml.size(); i++) {
        //递归解析 array 的每一个元素，并将解析后的值设置到 resolved 数组中，索引为 i
        Array.set(resolved, i,
            resolveValueIfNecessary(new KeyedArgName(argName, i), ml.get(i)));
    }
    return resolved;
}

//解析 list 类型的属性
private List resolveManagedList(Object argName, List<?> ml) {
    List<Object> resolved = new ArrayList<Object>(ml.size());
    for (int i = 0; i < ml.size(); i++) {
        //递归解析 list 的每一个元素
        resolved.add(
            resolveValueIfNecessary(new KeyedArgName(argName, i), ml.get(i)));
    }
    return resolved;
}

//解析 set 类型的属性
private Set resolveManagedSet(Object argName, Set<?> ms) {
    Set<Object> resolved = new LinkedHashSet<Object>(ms.size());
    int i = 0;
    //递归解析 set 的每一个元素
    for (Object m : ms) {
        resolved.add(resolveValueIfNecessary(new KeyedArgName(argName, i), m));
        i++;
    }
    return resolved;
}

//解析 map 类型的属性
private Map resolveManagedMap(Object argName, Map<?, ?> mm) {
    Map<Object, Object> resolved = new LinkedHashMap<Object, Object>(mm.size());
    //递归解析 map 中每一个元素的 key 和 value
    for (Map.Entry entry : mm.entrySet()) {
        Object resolvedKey = resolveValueIfNecessary(argName, entry.getKey());
        Object resolvedValue = resolveValueIfNecessary(
            new KeyedArgName(argName, entry.getKey()), entry.getValue());
        resolved.put(resolvedKey, resolvedValue);
    }
}

```

```
}  
    return resolved;  
  
}
```

通过上面的代码分析，我们明白了 Spring 是如何将引用类型，内部类以及集合类型等属性进行解析的，属性值解析完成后就可以进行依赖注入了，依赖注入的过程就是 Bean 对象实例设置到它所依赖的 Bean 对象属性上去，在第 7 步中我们已经说过，依赖注入是通过 `bw.setPropertyValues` 方法实现的，该方法也使用了委托模式，在 `BeanWrapper` 接口中至少定义了方法声明，依赖注入的具体实现交由其实现类 `BeanWrapperImpl` 来完成，下面我们就分析 `BeanWrapperImpl` 中依赖注入相关的源码。

8、BeanWrapperImpl 对 Bean 属性的依赖注入：

`BeanWrapperImpl` 类主要是对容器中完成初始化的 Bean 实例对象进行属性的依赖注入，即把 Bean 对象设置到它所依赖的另一个 Bean 的属性中去，依赖注入的相关源码如下：

```
//实现属性依赖注入功能  
private void setPropertyValue(PropertyTokenHolder tokens, PropertyValue pv) throws BeansException  
{  
    //PropertyTokenHolder 主要保存属性的名称、路径，以及集合的 size 等信息  
    String propertyName = tokens.canonicalName;  
    String actualName = tokens.actualName;  
    //keys 是用来保存集合类型属性的 size  
    if (tokens.keys != null) {  
        //将属性信息拷贝  
        PropertyTokenHolder getterTokens = new PropertyTokenHolder();  
        getterTokens.canonicalName = tokens.canonicalName;  
        getterTokens.actualName = tokens.actualName;  
        getterTokens.keys = new String[tokens.keys.length - 1];  
        System.arraycopy(tokens.keys, 0, getterTokens.keys, 0, tokens.keys.length - 1);  
        Object propValue;  
        try {  
            //获取属性值，该方法内部使用 JDK 的内省( Introspector)机制  
            //调用属性的 getter(readerMethod)方法，获取属性的值  
            propValue = getPropertyValue(getterTokens);  
        }  
        catch (NotReadablePropertyException ex) {  
            throw new NotWritablePropertyException(getRootClass(), this.nestedPath + propertyName,  
                "Cannot access indexed value in property referenced " +  
                "in indexed property path '" + propertyName + "'", ex);  
        }  
        //获取集合类型属性的长度  
        String key = tokens.keys[tokens.keys.length - 1];  
        if (propValue == null) {  
            throw new NullValueInNestedPathException(getRootClass(), this.nestedPath +
```

```

propertyName,
        "Cannot access indexed value in property referenced " +
        "in indexed property path '" + propertyName + "': returned null");
    }
    //注入 array 类型的属性值
    else if (propValue.getClass().isArray()) {
        //获取属性的描述符
        PropertyDescriptor pd =
getCachedIntrospectionResults().getPropertyDescriptor(actualName);
        //获取数组的类型
        Class requiredType = propValue.getClass().getComponentType();
        //获取数组的长度
        int arrayIndex = Integer.parseInt(key);
        Object oldValue = null;
        try {
            //获取数组以前初始化的值
            if (isExtractOldValueForEditor()) {
                oldValue = Array.get(propValue, arrayIndex);
            }
            //将属性的值赋值给数组中的元素
            Object convertedValue = convertIfNecessary(propertyName, oldValue, pv.getValue(),
requiredType,
                new PropertyTypeDescriptor(pd, new MethodParameter(pd.getReadMethod(),
-1), requiredType));
            Array.set(propValue, arrayIndex, convertedValue);
        }
        catch (IndexOutOfBoundsException ex) {
            throw new InvalidPropertyException(getRootClass(), this.nestedPath + propertyName,
                "Invalid array index in property path '" + propertyName + "'", ex);
        }
    }
    //注入 list 类型的属性值
    else if (propValue instanceof List) {
        PropertyDescriptor pd =
getCachedIntrospectionResults().getPropertyDescriptor(actualName);
        //获取 list 集合的类型
        Class requiredType = GenericCollectionTypeResolver.getCollectionReturnTypes(
            pd.getReadMethod(), tokens.keys.length);
        List list = (List) propValue;
        //获取 list 集合的 size
        int index = Integer.parseInt(key);
        Object oldValue = null;
        if (isExtractOldValueForEditor() && index < list.size()) {
            oldValue = list.get(index);
        }
    }
}

```

```

    }
    //获取 list 解析后的属性值
    Object convertedValue = convertIfNecessary(propertyName, oldValue, pv.getValue(),
requiredType,
        new PropertyTypeDescriptor(pd, new MethodParameter(pd.getReadMethod(), -1),
requiredType));
    if (index < list.size()) {
        //为 list 属性赋值
        list.set(index, convertedValue);
    }
    //如果 list 的长度大于属性值的长度, 则多余的元素赋值为 null
    else if (index >= list.size()) {
        for (int i = list.size(); i < index; i++) {
            try {
                list.add(null);
            }
            catch (NullPointerException ex) {
                throw new InvalidPropertyException(getRootClass(), this.nestedPath +
propertyName,
                    "Cannot set element with index " + index + " in List of size " +
list.size() + ", accessed using property path '" + propertyName +
"': List does not support filling up gaps with null elements");
            }
        }
        list.add(convertedValue);
    }
}
//注入 map 类型的属性值
else if (propValue instanceof Map) {
    PropertyDescriptor pd =
getCachedIntrospectionResults().getPropertyDescriptor(actualName);
    //获取 map 集合 key 的类型
    Class mapKeyType = GenericCollectionTypeResolver.getMapKeyReturnTypes(
        pd.getReadMethod(), tokens.keys.length);
    //获取 map 集合 value 的类型
    Class mapValueType = GenericCollectionTypeResolver.getMapValueReturnTypes(
        pd.getReadMethod(), tokens.keys.length);
    Map map = (Map) propValue;
    //解析 map 类型属性 key 值
    Object convertedMapKey = convertIfNecessary(null, null, key, mapKeyType,
        new PropertyTypeDescriptor(pd, new MethodParameter(pd.getReadMethod(), -1),
mapKeyType));
    Object oldValue = null;
    if (isExtractOldValueForEditor()) {

```

```

        oldValue = map.get(convertedMapKey);
    }
    //解析 map 类型属性 value 值
    Object convertedMapValue = convertIfNecessary(
        propertyName, oldValue, pv.getValue(), mapValueType,
        new TypeDescriptor(new MethodParameter(pd.getReadMethod(), -1,
tokens.keys.length + 1)));
    //将解析后的 key 和 value 值赋值给 map 集合属性
    map.put(convertedMapKey, convertedMapValue);
}
else {
    throw new InvalidPropertyException(getRootClass(), this.nestedPath + propertyName,
        "Property referenced in indexed property path '" + propertyName +
        "' is neither an array nor a List nor a Map; returned value was [" + pv.getValue()
+ "]);
}
}
//对非集合类型的属性注入
else {
    PropertyDescriptor pd = pv.resolvedDescriptor;
    if (pd == null || !pd.getWriteMethod().getDeclaringClass().isInstance(this.object)) {
        pd = getCachedIntrospectionResults().getPropertyDescriptor(actualName);
        //无法获取到属性名或者属性没有提供 setter(写方法)方法
        if (pd == null || pd.getWriteMethod() == null) {
            //如果属性值是可选的，即不是必须的，则忽略该属性值
            if (pv.isOptional()) {
                logger.debug("Ignoring optional value for property '" + actualName +
                    "' - property not found on bean class [" + getRootClass().getName() +
+ "]);

                return;
            }
            //如果属性值是必须的，则抛出无法给属性赋值，因为没提供 setter 方法异常
            else {
                PropertyMatches matches = PropertyMatches.forProperty(propertyName,
getRootClass());

                throw new NotWritablePropertyException(
                    getRootClass(), this.nestedPath + propertyName,
                    matches.buildErrorMessage(), matches.getPossibleMatches());
            }
        }
        pv.getOriginalPropertyValue().resolvedDescriptor = pd;
    }
    Object oldValue = null;
    try {

```



```

Object originalValue = pv.getValue();
Object valueToApply = originalValue;
if (!Boolean.FALSE.equals(pv.conversionNecessary)) {
    if (pv.isConverted()) {
        valueToApply = pv.getConvertedValue();
    }
    else {
        if (isExtractOldValueForEditor() && pd.getReadMethod() != null) {
            //获取属性的 getter 方法(读方法), JDK 内省机制
            final Method readMethod = pd.getReadMethod();
            //如果属性的 getter 方法不是 public 访问控制权限的, 即访问控制权限比较严格,
            //则使用 JDK 的反射机制强行访问非 public 的方法(暴力读取属性值)
            if (!Modifier.isPublic(readMethod.getDeclaringClass().getModifiers()) &&
                !readMethod.isAccessible()) {
                if (System.getSecurityManager() != null) {
                    //匿名内部类, 根据权限修改属性的读取控制限制
                    AccessController.doPrivileged(new PrivilegedAction<Object>() {
                        public Object run() {
                            readMethod.setAccessible(true);
                            return null;
                        }
                    });
                }
                else {
                    readMethod.setAccessible(true);
                }
            }
            try {
                //属性没有提供 getter 方法时, 调用潜在的读取属性值的方法, 获取属性值
                if (System.getSecurityManager() != null) {
                    oldValue = AccessController.doPrivileged(new
PrivilegedExceptionAction<Object>() {
                        public Object run() throws Exception {
                            return readMethod.invoke(object);
                        }
                    }, acc);
                }
                else {
                    oldValue = readMethod.invoke(object);
                }
            }
            catch (Exception ex) {
                if (ex instanceof PrivilegedActionException) {
                    ex = ((PrivilegedActionException) ex).getException();
                }
            }
        }
    }
}

```

```

        }
        if (logger.isDebugEnabled()) {
            logger.debug("Could not read previous value of property '" +
                this.nestedPath + propertyName + "'", ex);
        }
    }
}
//设置属性的注入值
valueToApply = convertForProperty(propertyName, oldValue, originalValue, pd);
}
pv.getOriginalPropertyValue().conversionNecessary = (valueToApply !=
originalValue);
}
//根据 JDK 的内省机制，获取属性的 setter(写方法)方法
final Method writeMethod = (pd instanceof GenericTypeAwarePropertyDescriptor ?
    ((GenericTypeAwarePropertyDescriptor) pd).getWriteMethodForActualAccess() :
    pd.getWriteMethod());
//如果属性的 setter 方法是非 public，即访问控制权限比较严格，则使用 JDK 的反射机制，
//强行设置 setter 方法可访问(暴力为属性赋值)
if (!Modifier.isPublic(writeMethod.getDeclaringClass().getModifiers())
&& !writeMethod.isAccessible()) {
    //如果使用了 JDK 的安全机制，则需要权限验证
    if (System.getSecurityManager() != null) {
        AccessController.doPrivileged(new PrivilegedAction<Object>() {
            public Object run() {
                writeMethod.setAccessible(true);
                return null;
            }
        });
    }
    else {
        writeMethod.setAccessible(true);
    }
}
final Object value = valueToApply;
if (System.getSecurityManager() != null) {
    try {
        //将属性值设置到属性上去
        AccessController.doPrivileged(new PrivilegedExceptionAction<Object>() {
            public Object run() throws Exception {
                writeMethod.invoke(object, value);
                return null;
            }
        }, acc);
    }
}

```

```

        }
        catch (PrivilegedActionException ex) {
            throw ex.getException();
        }
    }
    else {
        writeMethod.invoke(this.object, value);
    }
}
catch (TypeMismatchException ex) {
    throw ex;
}
catch (InvocationTargetException ex) {
    PropertyChangeEvent propertyChangeEvent =
        new PropertyChangeEvent(this.rootObject, this.nestedPath + propertyName,
oldValue, pv.getValue());
    if (ex.getTargetException() instanceof ClassCastException) {
        throw new TypeMismatchException(propertyChangeEvent, pd.getPropertyType(),
ex.getTargetException());
    }
    else {
        throw new MethodInvocationException(propertyChangeEvent,
ex.getTargetException());
    }
}
catch (Exception ex) {
    PropertyChangeEvent pce =
        new PropertyChangeEvent(this.rootObject, this.nestedPath + propertyName,
oldValue, pv.getValue());
    throw new MethodInvocationException(pce, ex);
}
}
}

```

通过对上面注入依赖代码的分析，我们已经明白了 Spring IOC 容器是如何将属性的值注入到 Bean 实例对象中去的：

- (1). 对于集合类型的属性，将其属性值解析为目标类型的集合后直接赋值给属性。
- (2). 对于非集合类型的属性，大量使用了 JDK 的反射和内省机制，通过属性的 getter 方法 (reader method) 获取指定属性注入以前的值，同时调用属性的 setter 方法 (writer method) 为属性设置注入后的值。看到这里相信很多人都明白了 Spring 的 setter 注入原理。

至此 Spring IOC 容器对 Bean 定义资源文件的定位，载入、解析和依赖注入已经全部分析完毕，现在 Spring IOC 容器中管理了一系列靠依赖关系联系起来的 Bean，程序不需要应用自己手动创建所需的对象，Spring IOC 容器会在我们使用的时候自动为我们创建，并且为我们注入好相关的依赖，这就是 Spring 核心功能的控制反转和依赖注入的相关功能。

5.5、IOC 容器的高级特性

1、介绍

通过前面 4 节对 Spring IOC 容器的源码分析，我们已经基本上了解了 Spring IOC 容器对 Bean 定义资源的定位、读入和解析过程，同时也清楚了当用户通过 `getBean` 方法向 IOC 容器获取被管理的 Bean 时，IOC 容器对 Bean 进行的初始化和依赖注入过程，这些是 Spring IOC 容器的基本功能特性。Spring IOC 容器还有一些高级特性，如使用 `lazy-init` 属性对 Bean 预初始化、`FactoryBean` 产生或者修饰 Bean 对象的生成、IOC 容器初始化 Bean 过程中使用 `BeanPostProcessor` 后置处理器对 Bean 声明周期事件管理和 IOC 容器的 `autowiring` 自动装配功能等。

2、Spring IOC 容器的 `lazy-init` 属性实现预实例化：

通过前面我们对 IOC 容器的实现和工作原理分析，我们知道 IOC 容器的初始化过程就是对 Bean 定义资源的定位、载入和注册，此时容器对 Bean 的依赖注入并没有发生，依赖注入主要是在应用程序第一次向容器索取 Bean 时，通过 `getBean` 方法的调用完成。

当 Bean 定义资源的 `<Bean>` 元素中配置了 `lazy-init` 属性时，容器将会在初始化的时候对所配置的 Bean 进行预实例化，Bean 的依赖注入在容器初始化的时候就已经完成。这样，当应用程序第一次向容器索取被管理的 Bean 时，就不用再初始化和对 Bean 进行依赖注入了，直接从容器中获取已经完成依赖注入的现成 Bean，可以提高应用第一次向容器获取 Bean 的性能。

下面我们通过代码分析容器预实例化的实现过程：

(1).`refresh()`

先从 IOC 容器的初始会过程开始，通过前面文章分析，我们知道 IOC 容器读入已经定位的 Bean 定义资源是从 `refresh` 方法开始的，我们首先从 `AbstractApplicationContext` 类的 `refresh` 方法入手分析，源码如下：

```
//容器初始化的过程，读入 Bean 定义资源，并解析注册
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        //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();
        //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();
        //为 BeanFactory 配置容器特性，例如类加载器、事件处理器等
        prepareBeanFactory(beanFactory);
```

```

try {
    //为容器的某些子类指定特殊的 BeanPost 事件处理器
    postProcessBeanFactory(beanFactory);
    //调用所有注册的 BeanFactoryPostProcessor 的 Bean
    invokeBeanFactoryPostProcessors(beanFactory);
    //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);
    //初始化信息源，和国际化相关。
    initMessageSource();
    //初始化容器事件传播器。
    initApplicationEventMulticaster();
    //调用子类的某些特殊 Bean 初始化方法
    onRefresh();
    //为事件传播器注册事件监听器。
    registerListeners();
    //这里是对容器 lazy-init 属性进行处理的入口方法
    finishBeanFactoryInitialization(beanFactory);
    //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) {
    //销毁以创建的单态 Bean
    destroyBeans();
    //取消 refresh 操作，重置容器的同步标识。
    cancelRefresh(ex);
    throw ex;
}
}
}

```

在 refresh 方法中 `ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();` 启动了 Bean 定义资源的载入、注册过程，而 `finishBeanFactoryInitialization` 方法是对注册后的 Bean 定义中的预实例化 (`lazy-init=false`，Spring 默认就是预实例化，即为 `true`) 的 Bean 进行处理的地方。

(2).finishBeanFactoryInitialization 处理预实例化 Bean:

当 Bean 定义资源被载入 IOC 容器之后，容器将 Bean 定义资源解析为容器内部的数据结构 `BeanDefinition` 注册到容器中，`AbstractApplicationContext` 类中的 `finishBeanFactoryInitialization` 方法对配置了预实例化属性的 Bean 进行预初始化过程，源码如下：

```

//对配置了 lazy-init 属性的 Bean 进行预实例化处理
protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {
    //这是 Spring3 以后新加的代码，为容器指定一个转换服务(ConversionService)

```

```

//在对某些 Bean 属性进行转换时使用
if (beanFactory.containsBean(CONVERSION_SERVICE_BEAN_NAME) &&
    beanFactory.isTypeMatch(CONVERSION_SERVICE_BEAN_NAME, ConversionService.class)) {
    beanFactory.setConversionService(
        beanFactory.getBean(CONVERSION_SERVICE_BEAN_NAME, ConversionService.class));
}
//为了类型匹配，停止使用临时的类加载器
beanFactory.setTempClassLoader(null);
//缓存容器中所有注册的 BeanDefinition 元数据，以防被修改
beanFactory.freezeConfiguration();
//对配置了 lazy-init 属性的单态模式 Bean 进行预实例化处理
beanFactory.preInstantiateSingletons();
}

```

ConfigurableListableBeanFactory 是一个接口，其 preInstantiateSingletons 方法由其子类 DefaultListableBeanFactory 提供。

(3)、DefaultListableBeanFactory 对配置 lazy-init 属性单态 Bean 的预实例化：

```

//对配置 lazy-init 属性单态 Bean 的预实例化
public void preInstantiateSingletons() throws BeansException {
    if (this.logger.isInfoEnabled()) {
        this.logger.info("Pre-instantiating singletons in " + this);
    }
    //在对配置 lazy-init 属性单态 Bean 的预实例化过程中，必须多线程同步，以确保数据一致性
    synchronized (this.beanDefinitionMap) {
        for (String beanName : this.beanDefinitionNames) {
            //获取指定名称的 Bean 定义
            RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
            //Bean 不是抽象的，是单态模式的，且 lazy-init 属性配置为 false
            if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
                //如果指定名称的 bean 是创建容器的 Bean
                if (isFactoryBean(beanName)) {
                    //FACTORY_BEAN_PREFIX="&", 当 Bean 名称前面加"&"符号
                    //时，获取的是产生容器对象本身，而不是容器产生的 Bean。
                    //调用 getBean 方法，触发容器对 Bean 实例化和依赖注入过程
                    final FactoryBean factory = (FactoryBean) getBean(FACTORY_BEAN_PREFIX +
beanName);

                    //标识是否需要预实例化
                    boolean isEagerInit;
                    if (System.getSecurityManager() != null && factory instanceof SmartFactoryBean)
                    {
                        //一个匿名内部类
                        isEagerInit = AccessController.doPrivileged(new

```

```
PrivilegedAction<Boolean>() {
    public Boolean run() {
        return ((SmartFactoryBean) factory).isEagerInit();
    }
}, getAccessControlContext());
}
else {
    isEagerInit = factory instanceof SmartFactoryBean && ((SmartFactoryBean)
factory).isEagerInit();
}
if (isEagerInit) {
    //调用 getBean 方法，触发容器对 Bean 实例化和依赖注入过程
    getBean(beanName);
}
}
else {
    //调用 getBean 方法，触发容器对 Bean 实例化和依赖注入过程
    getBean(beanName);
}
}
}
}
```

通过对 lazy-init 处理源码的分析，我们可以看出，如果设置了 lazy-init 属性，则容器在完成 Bean 定义的注册之后，会通过 getBean 方法，触发对指定 Bean 的初始化和依赖注入过程，这样当应用第一次向容器索取所需的 Bean 时，容器不再需要对 Bean 进行初始化和依赖注入，直接从已经完成实例化和依赖注入的 Bean 中取一个现成的 Bean，这样就提高了第一次获取 Bean 的性能。

3、FactoryBean 的实现:

在 Spring 中，有两个很容易混淆的类：BeanFactory 和 FactoryBean。

BeanFactory: Bean 工厂，是一个工厂(Factory)，我们 Spring IOC 容器的最顶层接口就是这个 BeanFactory，它的作用是管理 Bean，即实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。

FactoryBean: 工厂 Bean，是一个 Bean，作用是产生其他 bean 实例。通常情况下，这种 bean 没有什么特别的要求，仅需要提供一个工厂方法，该方法用来返回其他 bean 实例。通常情况下，bean 无须自己实现工厂模式，Spring 容器担任工厂角色；但少数情况下，容器中的 bean 本身就是工厂，其作用是产生其它 bean 实例。

当用户使用容器本身时，可以使用转义字符”&”来得到 FactoryBean 本身，以区别通过 FactoryBean 产生的实例对象和 FactoryBean 对象本身。在 BeanFactory 中通过如下代码定义了该转义字符：

StringFactory BEAN PREFIX = "&";

如果 myJndiObject 是一个 FactoryBean，则使用&myJndiObject 得到的是 myJndiObject 对象，而不是 myJndiObject 产生出来的对象。

(1). FactoryBean 的源码如下：

```
//工厂 Bean，用于产生其他对象
public interface FactoryBean<T> {
    //获取容器管理的对象实例
    T getObject() throws Exception;
    //获取 Bean 工厂创建的对象类型
    Class<?> getObjectType();
    //Bean 工厂创建的对象是否是单态模式，如果是单态模式，则整个容器中只有一个实例
    //对象，每次请求都返回同一个实例对象
    boolean isSingleton();
}
```

(2). AbstractBeanFactory 的 getBean 方法调用 FactoryBean：

在前面我们分析 Spring IOC 容器实例化 Bean 并进行依赖注入过程的源码时，提到在 getBean 方法触发容器实例化 Bean 的时候会调用 AbstractBeanFactory 的 doGetBean 方法来进行实例化的过程，源码如下：

```
//真正实现向 IOC 容器获取 Bean 的功能，也是触发依赖注入功能的地方
@SuppressWarnings("unchecked")
protected <T> T doGetBean(
    final String name, final Class<T> requiredType, final Object[] args, boolean
typeCheckOnly)
    throws BeansException {
    //根据指定的名称获取被管理 Bean 的名称，剥离指定名称中对容器的相关依赖
    //如果指定的是别名，将别名转换为规范的 Bean 名称
    final String beanName = transformedBeanName(name);
    Object bean;
    //先从缓存中取是否已经有被创建过的单态类型的 Bean，对于单态模式的 Bean 整
    //个 IoC 容器中只创建一次，不需要重复创建
    Object sharedInstance = getSingleton(beanName);
    //IoC 容器创建单态模式 Bean 实例对象
    if (sharedInstance != null && args == null) {
        if (logger.isDebugEnabled()) {
            //如果指定名称的 Bean 在容器中已有单态模式的 Bean 被创建，直接返回
            //已经创建的 Bean
            if (isSingletonCurrentlyInCreation(beanName)) {
                logger.debug("Returning eagerly cached instance of singleton bean '" + beanName
+
                    "' that is not fully initialized yet - a consequence of a circular
```



```

reference");
    }
    else {
        logger.debug("Returning cached instance of singleton bean '" + beanName + "'");
    }
}
//获取给定 Bean 的实例对象，主要是完成 FactoryBean 的相关处理
bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
}
.....
}
//获取给定 Bean 的实例对象，主要是完成 FactoryBean 的相关处理
protected Object getObjectForBeanInstance(
    Object beanInstance, String name, String beanName, RootBeanDefinition mbd) {
    //容器已经得到了 Bean 实例对象，这个实例对象可能是一个普通的 Bean，
    //也可能是一个工厂 Bean，如果是一个工厂 Bean，则使用它创建一个 Bean 实例对象，
    //如果调用本身就想获得一个容器的引用，则指定返回这个工厂 Bean 实例对象
    //如果指定的名称是容器的解引用(dereference，即是对象本身而非内存地址)，
    //且 Bean 实例也不是创建 Bean 实例对象的工厂 Bean
    if (BeanFactoryUtils.isFactoryDereference(name) && !(beanInstance instanceof FactoryBean)) {
        throw new BeanIsNotAFactoryException(transformedBeanName(name),
beanInstance.getClass());
    }
    //如果 Bean 实例不是工厂 Bean，或者指定名称是容器的解引用，
    //调用者向获取对容器的引用，则直接返回当前的 Bean 实例
    if (!(beanInstance instanceof FactoryBean) || BeanFactoryUtils.isFactoryDereference(name)) {
        return beanInstance;
    }
    //处理指定名称不是容器的解引用，或者根据名称获取的 Bean 实例对象是一个工厂 Bean
    //使用工厂 Bean 创建一个 Bean 的实例对象
    Object object = null;
    if (mbd == null) {
        //从 Bean 工厂缓存中获取给定名称的 Bean 实例对象
        object = getCacheableObjectForFactoryBean(beanName);
    }
    //让 Bean 工厂生产给定名称的 Bean 对象实例
    if (object == null) {
        FactoryBean factory = (FactoryBean) beanInstance;
        //如果从 Bean 工厂生产的 Bean 是单态模式的，则缓存
        if (mbd == null && containsBeanDefinition(beanName)) {
            //从容器中获取指定名称的 Bean 定义，如果继承基类，则合并基类相关属性
            mbd = getMergedLocalBeanDefinition(beanName);
        }
        //如果从容器得到 Bean 定义信息，并且 Bean 定义信息不是虚构的，

```

```

        //则让工厂 Bean 生产 Bean 实例对象
        boolean synthetic = (mbd != null && mbd.isSynthetic());
        //调用 FactoryBeanRegistrySupport 类的 getObjectFromFactoryBean 方法,
        //实现工厂 Bean 生产 Bean 对象实例的过程
        object = getObjectFromFactoryBean(factory, beanName, !synthetic);
    }
    return object;
}

```

在上面获取给定 Bean 的实例对象的 getObjectForBeanInstance 方法中，会调用 FactoryBeanRegistrySupport 类的 getObjectFromFactoryBean 方法,该方法实现了 Bean 工厂生产 Bean 实例对象。

Dereference(解引用)：一个在 C/C++中应用比较多的术语，在 C++中，“*”是解引用符号，而“&”是引用符号，解引用是指变量指向的是所引用对象的本身数据，而不是引用对象的内存地址。

(3)、AbstractBeanFactory 生产 Bean 实例对象：

AbstractBeanFactory 类中生产 Bean 实例对象的主要源码如下：

```

//Bean 工厂生产 Bean 实例对象
protected Object getObjectFromFactoryBean(FactoryBean factory, String beanName, boolean
shouldPostProcess) {
    //Bean 工厂是单态模式，并且 Bean 工厂缓存中存在指定名称的 Bean 实例对象
    if (factory.isSingleton() && containsSingleton(beanName)) {
        //多线程同步，以防止数据不一致
        synchronized (getSingletonMutex()) {
            //直接从 Bean 工厂缓存中获取指定名称的 Bean 实例对象
            Object object = this.factoryBeanObjectCache.get(beanName);
            //Bean 工厂缓存中没有指定名称的实例对象，则生产该实例对象
            if (object == null) {
                //调用 Bean 工厂的 getObject 方法生产指定 Bean 的实例对象
                object = doGetObjectFromFactoryBean(factory, beanName, shouldPostProcess);
                //将生产的实例对象添加到 Bean 工厂缓存中
                this.factoryBeanObjectCache.put(beanName, (object != null ? object : NULL_OBJECT));
            }
            return (object != NULL_OBJECT ? object : null);
        }
    }
    //调用 Bean 工厂的 getObject 方法生产指定 Bean 的实例对象
    else {
        return doGetObjectFromFactoryBean(factory, beanName, shouldPostProcess);
    }
}

```

```

//调用 Bean 工厂的 getObject 方法生产指定 Bean 的实例对象
private Object doGetObjectFromFactoryBean(
    final FactoryBean factory, final String beanName, final boolean shouldPostProcess)
    throws BeanCreationException {
    Object object;
    try {
        if (System.getSecurityManager() != null) {
            AccessControlContext acc = getAccessControlContext();
            try {
                //实现 PrivilegedExceptionAction 接口的匿名内置类
                //根据 JVM 检查权限，然后决定 BeanFactory 创建实例对象
                object = AccessController.doPrivileged(new PrivilegedExceptionAction<Object>() {
                    public Object run() throws Exception {
                        //调用 BeanFactory 接口实现类的创建对象方法
                        return factory.getObject();
                    }
                }, acc);
            }
            catch (PrivilegedActionException pae) {
                throw pae.getException();
            }
        }
        else {
            //调用 BeanFactory 接口实现类的创建对象方法
            object = factory.getObject();
        }
    }
    catch (FactoryBeanNotInitializedException ex) {
        throw new BeanCurrentlyInCreationException(beanName, ex.toString());
    }
    catch (Throwable ex) {
        throw new BeanCreationException(beanName, "FactoryBean threw exception on object creation",
ex);
    }
    //创建出来的实例对象为 null，或者因为单态对象正在创建而返回 null
    if (object == null && isSingletonCurrentlyInCreation(beanName)) {
        throw new BeanCurrentlyInCreationException(
            beanName, "FactoryBean which is currently in creation returned null from getObject");
    }
    //为创建出来的 Bean 实例对象添加 BeanPostProcessor 后置处理器
    if (object != null && shouldPostProcess) {
        try {
            object = postProcessObjectFromFactoryBean(object, beanName);
        }
    }
}

```

```

        catch (Throwable ex) {
            throw new BeanCreationException(beanName, "Post-processing of the FactoryBean's object
failed", ex);
        }
    }
    return object;
}

```

从上面的源码分析中,我们可以看出,BeanFactory 接口调用其实现类的 getObject 方法来实现创建 Bean 实例对象的功能。

(4). 工厂 Bean 的实现类 getObject 方法创建 Bean 实例对象:

FactoryBean 的实现类有非常多,比如:Proxy、RMI、JNDI、ServletContextFactoryBean 等等,FactoryBean 接口为 Spring 容器提供了一个很好的封装机制,具体的 getObject 有不同的实现类根据不同的实现策略来具体提供,我们分析一个最简单的 AnnotationTestFactoryBean 的实现源码:

```

public class AnnotationTestBeanFactory implements FactoryBean<IJmxTestBean> {
    private final FactoryCreatedAnnotationTestBean instance = new
FactoryCreatedAnnotationTestBean();
    public AnnotationTestBeanFactory() {
        this.instance.setName("FACTORY");
    }
    //AnnotationTestBeanFactory 产生 Bean 实例对象的实现
    public IJmxTestBean getObject() throws Exception {
        return this.instance;
    }
    public Class<? extends IJmxTestBean> getObjectType() {
        return FactoryCreatedAnnotationTestBean.class;
    }
    public boolean isSingleton() {
        return true;
    }
}

```

其他的 Proxy, RMI, JNDI 等等,都是根据相应的策略提供 getObject 的实现。这里不做一一分析,这已经不是 Spring 的核心功能,有需要的时候再去深入研究。

4. BeanPostProcessor 后置处理器的实现:

BeanPostProcessor 后置处理器是 Spring IOC 容器经常使用到的一个特性,这个 Bean 后置处理器是一个监听器,可以监听容器触发的 Bean 声明周期事件。后置处理器向容器注册以后,容器中管理的 Bean 就具备了接收 IOC 容器事件回调的能力。

BeanPostProcessor 的使用非常简单，只需要提供一个实现接口 BeanPostProcessor 的实现类，然后在 Bean 的配置文件中设置即可。

(1). BeanPostProcessor 的源码如下：

```
package org.springframework.beans.factory.config;
import org.springframework.beans.BeansException;
public interface BeanPostProcessor {
    //为在 Bean 的初始化前提供回调入口
    Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;
    //为在 Bean 的初始化之后提供回调入口
    Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException;
}
```

这两个回调的入口都是和容器管理的 Bean 的生命周期事件紧密相关，可以为用户提供在 Spring IOC 容器初始化 Bean 过程中自定义的处理操作。

(2). AbstractAutowireCapableBeanFactory 类对容器生成的 Bean 添加后置处理器：

BeanPostProcessor 后置处理器的调用发生在 Spring IOC 容器完成对 Bean 实例对象的创建和属性的依赖注入完成之后，在对 Spring 依赖注入的源码分析过程中我们知道，当应用程序第一次调用 getBean 方法(lazy-init 预实例化除外)向 Spring IOC 容器索取指定 Bean 时触发 Spring IOC 容器创建 Bean 实例对象并进行依赖注入的过程，其中真正实现创建 Bean 对象并进行依赖注入的方法是 AbstractAutowireCapableBeanFactory 类的 doCreateBean 方法，主要源码如下：

```
//真正创建 Bean 的方法
protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[]
args) {
    //创建 Bean 实例对象
    .....
    try {
        //对 Bean 属性进行依赖注入
        populateBean(beanName, mbd, instanceWrapper);
        if (exposedObject != null) {
            //在对 Bean 实例对象生成和依赖注入完成以后，开始对 Bean 实例对象
            //进行初始化，为 Bean 实例对象应用 BeanPostProcessor 后置处理器
            exposedObject = initializeBean(beanName, exposedObject, mbd);
        }
    }
    catch (Throwable ex) {
        if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException)
ex).getBeanName())) {
            throw (BeanCreationException) ex;
        }
    }
}
```

```
.....  
//为应用返回所需要的实例对象  
return exposedObject;  
  
}
```

从上面的代码中我们知道，为 Bean 实例对象添加 BeanPostProcessor 后置处理器的入口的是 initializeBean 方法。

(3). initializeBean 方法为容器产生的 Bean 实例对象添加 BeanPostProcessor 后置处理器：

同样在 AbstractAutowireCapableBeanFactory 类中，initializeBean 方法实现为容器创建的 Bean 实例对象添加 BeanPostProcessor 后置处理器，源码如下：

```
//初始容器创建的 Bean 实例对象，为其添加 BeanPostProcessor 后置处理器  
protected Object initializeBean(final String beanName, final Object bean, RootBeanDefinition mbd)  
{  
    //JDK 的安全机制验证权限  
    if (System.getSecurityManager() != null) {  
        //实现 PrivilegedAction 接口的匿名内部类  
        AccessController.doPrivileged(new PrivilegedAction<Object>() {  
            public Object run() {  
                invokeAwareMethods(beanName, bean);  
                return null;  
            }  
        }, getAccessControlContext());  
    }  
    else {  
        //为 Bean 实例对象包装相关属性，如名称，类加载器，所属容器等信息  
        invokeAwareMethods(beanName, bean);  
    }  
    Object wrappedBean = bean;  
    //对 BeanPostProcessor 后置处理器的 postProcessBeforeInitialization  
    //回调方法的调用，为 Bean 实例初始化前做一些处理  
    if (mbd == null || !mbd.isSynthetic()) {  
        wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);  
    }  
    //调用 Bean 实例对象初始化的方法，这个初始化方法是在 Spring Bean 定义配置  
    //文件中通过 init-method 属性指定的  
    try {  
        invokeInitMethods(beanName, wrappedBean, mbd);  
    }  
    catch (Throwable ex) {  
        throw new BeanCreationException(  
            (mbd != null ? mbd.getResourceDescription() : null),
```

```

        beanName, "Invocation of init method failed", ex);
    }
    //对 BeanPostProcessor 后置处理器的 postProcessAfterInitialization
    //回调方法的调用，为 Bean 实例初始化之后做一些处理
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
    }
    return wrappedBean;
}
//调用 BeanPostProcessor 后置处理器实例对象初始化之前的处理方法
public Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, String beanName)
    throws BeansException {
    Object result = existingBean;
    //遍历容器为所创建的 Bean 添加的所有 BeanPostProcessor 后置处理器
    for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
        //调用 Bean 实例所有的后置处理中的初始化前处理方法，为 Bean 实例对象在
        //初始化之前做一些自定义的处理操作
        result = beanProcessor.postProcessBeforeInitialization(result, beanName);
        if (result == null) {
            return result;
        }
    }
    return result;
}
//调用 BeanPostProcessor 后置处理器实例对象初始化之后的处理方法
public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)
    throws BeansException {
    Object result = existingBean;
    //遍历容器为所创建的 Bean 添加的所有 BeanPostProcessor 后置处理器
    for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
        //调用 Bean 实例所有的后置处理中的初始化后处理方法，为 Bean 实例对象在
        //初始化之后做一些自定义的处理操作
        result = beanProcessor.postProcessAfterInitialization(result, beanName);
        if (result == null) {
            return result;
        }
    }
    return result;
}
}

```

BeanPostProcessor 是一个接口，其初始化前的操作方法和初始化后的操作方法均委托其实现子类来实现，在 Spring 中，BeanPostProcessor 的实现子类非常的多，分别完成不同的操作，如：AOP 面向切面

编程的注册通知适配器、Bean 对象的数据校验、Bean 继承属性/方法的合并等等，我们以最简单的 AOP 切面织入来简单了解其主要的功能。

(4).AdvisorAdapterRegistrationManager 在 Bean 对象初始化后注册通知适配器：

AdvisorAdapterRegistrationManager 是 BeanPostProcessor 的一个实现类，其主要的作用为容器中管理的 Bean 注册一个面向切面编程的通知适配器，以便在 Spring 容器为所管理的 Bean 进行面向切面编程时提供方便，其源码如下：

```
//为容器中管理的 Bean 注册一个面向切面编程的通知适配器
public class AdvisorAdapterRegistrationManager implements BeanPostProcessor {
    //容器中负责管理切面通知适配器注册的对象
    private AdvisorAdapterRegistry advisorAdapterRegistry =
GlobalAdvisorAdapterRegistry.getInstance();
    public void setAdvisorAdapterRegistry(AdvisorAdapterRegistry advisorAdapterRegistry) {
        this.advisorAdapterRegistry = advisorAdapterRegistry;
    }
    //BeanPostProcessor 在 Bean 对象初始化前的操作
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException
    {
        //没有做任何操作，直接返回容器创建的 Bean 对象
        return bean;
    }
    //BeanPostProcessor 在 Bean 对象初始化后的操作
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException
    {
        if (bean instanceof AdvisorAdapter){
            //如果容器创建的 Bean 实例对象是一个切面通知适配器，则向容器的注册
            this.advisorAdapterRegistry.registerAdvisorAdapter((AdvisorAdapter) bean);
        }
        return bean;
    }
}
```

其他的 BeanPostProcessor 接口实现类的也类似，都是对 Bean 对象使用到的一些特性进行处理，或者向 IOC 容器中注册，为创建的 Bean 实例对象做一些自定义的功能增加，这些操作是容器初始化 Bean 时自动触发的，不需要人为的干预。

5.Spring IOC 容器 autowiring 实现原理：

Spring IOC 容器提供了两种管理 Bean 依赖关系的方式：

a. 显式管理：通过 BeanDefinition 的属性值和构造方法实现 Bean 依赖关系管理。

b. autowiring: Spring IOC 容器的依赖自动装配功能，不需要对 Bean 属性的依赖关系做显式的声明，只需要在配置好 autowiring 属性，IOC 容器会自动使用反射查找属性的类型和名称，然后基于属性的类型或者名称来自动匹配容器中管理的 Bean，从而自动地完成依赖注入。

通过对 autowiring 自动装配特性的理解，我们知道容器对 Bean 的自动装配发生在容器对 Bean 依赖注入的过程中。在前面对 Spring IOC 容器的依赖注入过程源码分析中，我们已经知道了容器对 Bean 实例对象的属性注入的处理发生在 AbstractAutoWireCapableBeanFactory 类中的 populateBean 方法中，我们通过程序流程分析 autowiring 的实现原理：

(1). AbstractAutoWireCapableBeanFactory 对 Bean 实例进行属性依赖注入：

应用第一次通过 getBean 方法(配置了 lazy-init 预实例化属性的除外)向 IoC 容器索取 Bean 时，容器创建 Bean 实例对象，并且对 Bean 实例对象进行属性依赖注入，AbstractAutoWireCapableBeanFactory 的 populateBean 方法就是实现 Bean 属性依赖注入的功能，其主要源码如下：

```
protected void populateBean(String beanName, AbstractBeanDefinition mbd, BeanWrapper bw) {
    //获取 Bean 定义的属性值，并对属性值进行处理
    PropertyValues pvs = mbd.getPropertyValues();
    .....
    //对依赖注入处理，首先处理 autowiring 自动装配的依赖注入
    if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME ||
        mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
        MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
        //根据 Bean 名称进行 autowiring 自动装配处理
        if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME) {
            autowireByName(beanName, mbd, bw, newPvs);
        }
        //根据 Bean 类型进行 autowiring 自动装配处理
        if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
            autowireByType(beanName, mbd, bw, newPvs);
        }
    }
    //对非 autowiring 的属性进行依赖注入处理
    .....
}
```

(2). Spring IOC 容器根据 Bean 名称或者类型进行 autowiring 自动依赖注入：

```
//根据名称对属性进行自动依赖注入
protected void autowireByName(
    String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs)
{
    //对 Bean 对象中非简单属性(不是简单继承的对象，如 8 中原始类型，字符串，URL 等都是简单属性)进行处理
    String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
```

```

for (String propertyName : propertyNames) {
    //如果 Spring IOC 容器中包含指定名称的 Bean
    if (containsBean(propertyName)) {
        //调用 getBean 方法向 IoC 容器索取指定名称的 Bean 实例，迭代触发属性的初始化和依赖注入
        Object bean = getBean(propertyName);
        //为指定名称的属性赋予属性值
        pvs.add(propertyName, bean);
        //指定名称属性注册依赖 Bean 名称，进行属性依赖注入
        registerDependentBean(propertyName, beanName);
        if (logger.isDebugEnabled()) {
            logger.debug("Added autowiring by name from bean name '" + beanName +
                "' via property '" + propertyName + "' to bean named '" + propertyName + "'");
        }
    }
    else {
        if (logger.isTraceEnabled()) {
            logger.trace("Not autowiring property '" + propertyName + "' of bean '" + beanName
+
                "' by name: no matching bean found");
        }
    }
}
}

//根据类型对属性进行自动依赖注入
protected void autowireByType(
    String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs)
{
    //获取用户定义的类型转换器
    TypeConverter converter = getCustomTypeConverter();
    if (converter == null) {
        converter = bw;
    }
    //存放解析的要注入的属性
    Set<String> autowiredBeanNames = new LinkedHashSet<String>(4);
    //对 Bean 对象中非简单属性(不是简单继承的对象，如 8 中原始类型，字符
    //URL 等都是简单属性)进行处理
    String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
    for (String propertyName : propertyNames) {
        try {
            //获取指定属性名称的属性描述器
            PropertyDescriptor pd = bw.getPropertyDescriptor(propertyName);
            //不对 Object 类型的属性进行 autowiring 自动依赖注入
            if (!Object.class.equals(pd.getPropertyType())) {
                //获取属性的 setter 方法

```

```

MethodParameter methodParam = BeanUtils.getWriteMethodParameter(pd);
//检查指定类型是否可以被转换为目标对象的类型
boolean eager = !PriorityOrdered.class.isAssignableFrom(bw.getWrappedClass());
//创建一个要被注入的依赖描述
DependencyDescriptor desc = new AutowireByTypeDependencyDescriptor(methodParam,
eager);

//根据容器的 Bean 定义解析依赖关系，返回所有要被注入的 Bean 对象
Object autowiredArgument = resolveDependency(desc, beanName, autowiredBeanNames,
converter);

if (autowiredArgument != null) {
    //为属性赋值所引用的对象
    pvs.add(propertyName, autowiredArgument);
}
for (String autowiredBeanName : autowiredBeanNames) {
    //指定名称属性注册依赖 Bean 名称，进行属性依赖注入
    registerDependentBean(autowiredBeanName, beanName);
    if (logger.isDebugEnabled()) {
        logger.debug("Autowiring by type from bean name '" + beanName + "' via
property '" +
                propertyName + "' to bean named '" + autowiredBeanName + "'");
    }
}
//释放已自动注入的属性
autowiredBeanNames.clear();
}
}
catch (BeansException ex) {
    throw new UnsatisfiedDependencyException(mbd.getResourceDescription(), beanName,
propertyName, ex);
}
}
}

```

通过上面的源码分析，我们可以看出来通过属性名进行自动依赖注入的相对比通过属性类型进行自动依赖注入要稍微简单一些，但是真正实现属性注入的是 DefaultSingletonBeanRegistry 类的 registerDependentBean 方法。

(3).DefaultSingletonBeanRegistry 的 registerDependentBean 方法对属性注入：

```

//为指定的 Bean 注入依赖的 Bean
public void registerDependentBean(String beanName, String dependentBeanName) {
    //处理 Bean 名称，将别名转换为规范的 Bean 名称
    String canonicalName = canonicalName(beanName);
    //多线程同步，保证容器内数据的一致性

```

```

//先从容器中: bean 名称-->全部依赖 Bean 名称集合找查找给定名称 Bean 的依赖 Bean
synchronized (this.dependentBeanMap) {
    //获取给定名称 Bean 的所有依赖 Bean 名称
    Set<String> dependentBeans = this.dependentBeanMap.get(canonicalName);
    if (dependentBeans == null) {
        //为 Bean 设置依赖 Bean 信息
        dependentBeans = new LinkedHashSet<String>(8);
        this.dependentBeanMap.put(canonicalName, dependentBeans);
    }
    //向容器中: bean 名称-->全部依赖 Bean 名称集合添加 Bean 的依赖信息
    //即, 将 Bean 所依赖的 Bean 添加到容器的集合中
    dependentBeans.add(dependentBeanName);
}

//从容器中: bean 名称-->指定名称 Bean 的依赖 Bean 集合找查找给定名称 Bean 的依赖 Bean
synchronized (this.dependenciesForBeanMap) {
    Set<String> dependenciesForBean = this.dependenciesForBeanMap.get(dependentBeanName);
    if (dependenciesForBean == null) {
        dependenciesForBean = new LinkedHashSet<String>(8);
        this.dependenciesForBeanMap.put(dependentBeanName, dependenciesForBean);
    }
    //向容器中: bean 名称-->指定 Bean 的依赖 Bean 名称集合添加 Bean 的依赖信息
    //即, 将 Bean 所依赖的 Bean 添加到容器的集合中
    dependenciesForBean.add(canonicalName);
}
}

```

通过对 autowiring 的源码分析, 我们可以看出, autowiring 的实现过程:

- a. 对 Bean 的属性调用 getBean 方法, 完成依赖 Bean 的初始化和依赖注入。
- b. 将依赖 Bean 的属性引用设置到被依赖的 Bean 属性上。
- c. 将依赖 Bean 的名称和被依赖 Bean 的名称存储在 IOC 容器的集合中。

Spring IOC 容器的 autowiring 属性自动依赖注入是一个很方便的特性, 可以简化开发时的配置, 但是凡事都有两面性, 自动属性依赖注入也有不足, 首先, Bean 的依赖关系在配置文件中无法很清楚地看出来, 对于维护造成一定困难。其次, 由于自动依赖注入是 Spring 容器自动执行的, 容器是不会智能判断的, 如果配置不当, 将会带来无法预料的后果, 所以自动依赖注入特性在使用时还是综合考虑。