Spring 最重要的概念是 IOC 和 AOP，本篇文章其实就是要带领大家来分析下 Spring 的 IOC 容器。既然大家平时都要用到 Spring，怎么可以不好好了解 Spring 呢？阅读本文并不能让你成为 Spring 专家，不过一定有助于大家理解 Spring 的很多概念，帮助大家排查应用中和 Spring 相关的一些问题。

本文采用的源码版本是 4.3.11.RELEASE，算是 5.0.x 前比较新的版本了。为了降低难度，本文所说的所有的内容都是基于 xml 的配置的方式，实际使用已经很少人这么做了，至少不是纯 xml 配置，不过从理解源码的角度来看用这种方式来说无疑是最合适的。

阅读建议：读者至少需要知道怎么配置 Spring，了解 Spring 中的各种概念，少部分内容我还假设读者使用过 SpringMVC。本文要说的 IOC 总体来说有两处地方最重要，一个是创建 Bean 容器，一个是初始化 Bean，如果读者觉得一次性看完本文压力有点大，那么可以按这个思路分两次消化。读者不一定对 Spring 容器的源码感兴趣，也许附录部分介绍的知识对读者有些许作用。

希望通过本文可以让读者不惧怕阅读 Spring 源码，也希望大家能反馈表述错误或不合理的地方。

引言

先看下最基本的启动 Spring 容器的例子：

```
public static void main(String[] args) {
    ApplicationContext context = new
ClassPathXmlApplicationContext("classpath:applicationfile.xml");
}
```

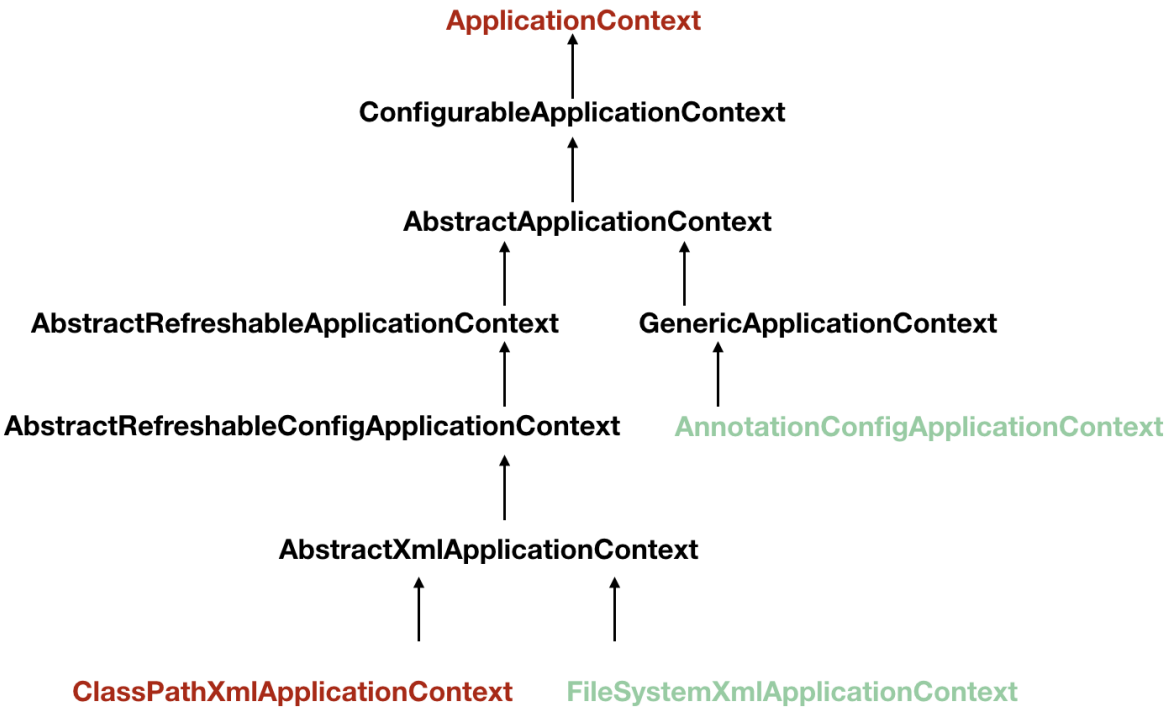
以上代码就可以利用配置文件来启动一个 Spring 容器了，请使用 maven 的小伙伴直接在 dependencies 中加上以下依赖即可，个人比较反对那些不知道要添加什么依赖，然后把 Spring 的所有相关的东西都加进来的方式。

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>4.3.11.RELEASE</version>
</dependency>
```

> spring-context 会自动将 spring-core、spring-beans、spring-aop、spring-expression 这几个基础 jar 包带进来。

多说一句，很多开发者入门就直接接触的 SpringMVC，对 Spring 其实不是很了解，Spring 是渐进式的工具，并不具有很强的侵入性，它的模块也划分得很合理，即使你的应用不是 web 应用，或者之前完全没有使用到 Spring，而你就想用 Spring 的依赖注入这个功能，其实完全是可以的，它的引入不会对其他的组件产生冲突。

废话说完，我们继续。`ApplicationContext context = new ClassPathXmlApplicationContext(...)` 其实很好理解，从名字上就可以猜出一二，就是在 ClassPath 中寻找 xml 配置文件，根据 xml 文件内容来构建 ApplicationContext。当然，除了 ClassPathXmlApplicationContext 以外，我们也还有其他构建 ApplicationContext 的方案可供选择，我们先来看看大体的继承结构是怎么样的：



> 读者可以大致看一下类名，源码分析的时候不至于找不着看哪个类，因为 Spring 为了适应各种使用场景，提供的各个接口都可能有很多的实现类。对于我们来说，就是揪着一个完整的分支看完。
>
> 当然，读本文的时候读者也不必太担心，每个代码块分析的时候，我都会告诉读者我们在说哪个类第几行。

我们可以看到，ClassPathXmlApplicationContext 兜兜转转了好久才到 ApplicationContext 接口，同样的，我们也可以使用绿颜色的 **FileSystemXmlApplicationContext** 和 **AnnotationConfigApplicationContext** 这两个类。

1、FileSystemXmlApplicationContext 的构造函数需要一个 xml 配置文件在系统中的路径，其他和 ClassPathXmlApplicationContext 基本上一样。

2、AnnotationConfigApplicationContext 是基于注解来使用的，它不需要配置文件，采用 java 配置类和各种注解来配置，是比较简单的方式，也是大势所趋吧。

不过本文旨在帮助大家理解整个构建流程，所以决定使用 ClassPathXmlApplicationContext 进行分析。

我们先来一个简单的例子来看看怎么实例化 ApplicationContext。

首先，定义一个接口：

```
public interface MessageService {
    String getMessage();
}
```

定义接口实现类：

```
public class MessageServiceImpl implements MessageService {

    public String getMessage() {
        return "hello world";
    }
}
```

接下来，我们在 **resources** 目录新建一个配置文件，文件名随意，通常叫 application.xml 或 application-xxx.xml 就可以了：

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

    <bean id="messageService" class="com.javadoop.example.MessageServiceImpl"/>
</beans>
```

这样，我们就可以跑起来了：

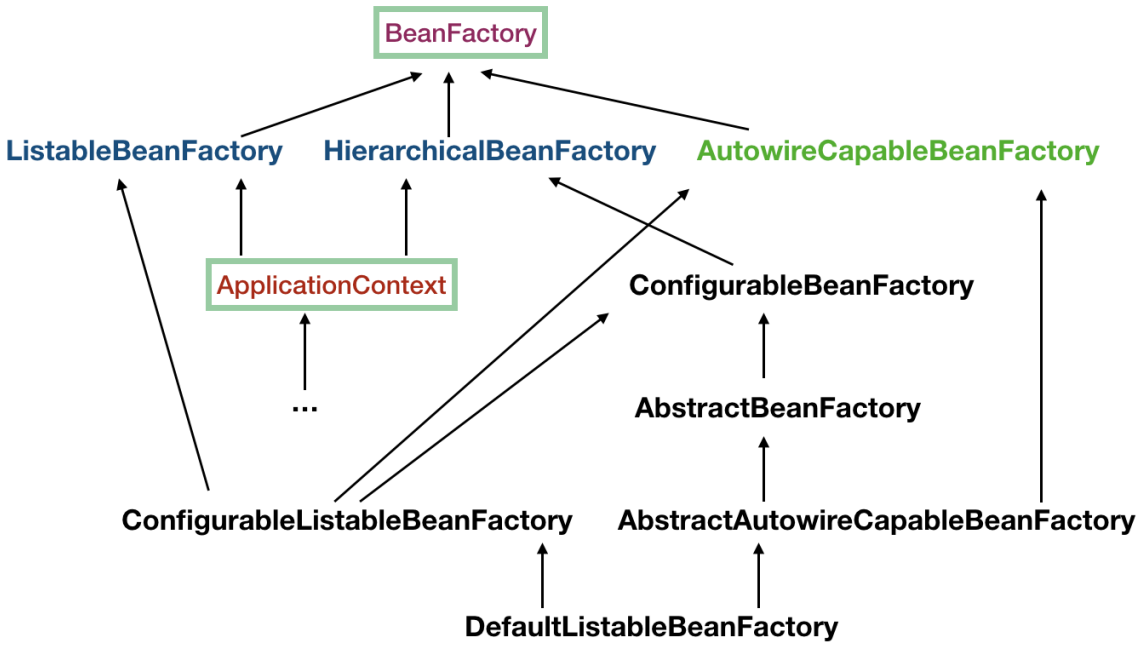
```
public class App {
    public static void main(String[] args) {
        // 用我们的配置文件来启动一个 ApplicationContext
        ApplicationContext context = new
ClassPathXmlApplicationContext("classpath:application.xml");

        System.out.println("context 启动成功");

        // 从 context 中取出我们的 Bean，而不是用 new MessageServiceImpl() 这种方式
        MessageService messageService = context.getBean(MessageService.class);
        // 这句将输出: hello world
        System.out.println(messageService.getMessage());
    }
}
```

以上例子很简单，不过也够引出本文的主题了，就是怎么样通过配置文件来启动 Spring 的 ApplicationContext？也就是我们今天要分析的 IOC 的核心了。ApplicationContext 启动过程中，会负责创建实例 Bean，往各个 Bean 中注入依赖等。

BeanFactory 简介

BeanFactory，从名字上也很好理解，生产 bean 的工厂，它负责生产和管理各个 bean 实例。

初学者可别以为我之前说那么多和 BeanFactory 无关，前面说的 ApplicationContext 其实就是一个 BeanFactory。我们来看下和 BeanFactory 接口相关的主要的继承结构：



我想，大家看完这个图以后，可能就不是很开心了。ApplicationContext 往下的继承结构前面一张图说过了，这里就不重复了。这张图呢，背下来肯定是不需要的，有几个重点和大家说明下就好。

1. ApplicationContext 继承了 ListableBeanFactory，这个 Listable 的意思就是，通过这个接口，我们可以获取多个 Bean，大家看源码会发现，最顶层 BeanFactory 接口的方法都是获取单个 Bean 的。
2. ApplicationContext 继承了 HierarchicalBeanFactory，Hierarchical 单词本身已经能说明问题了，也就是说我们可以在应用中起多个 BeanFactory，然后可以将各个 BeanFactory 设置为父子关系。
3. AutowireCapableBeanFactory 这个名字中的 Autowire 大家都非常熟悉，它就是用来自动装配 Bean 用的，但是仔细看上图，ApplicationContext 并没有继承它，不过不用担心，不使用继承，不代表不可以使用组合，如果你看到 ApplicationContext 接口定义中的最后一个方法 getAutowireCapableBeanFactory() 就知道了。
4. ConfigurableListableBeanFactory 也是一个特殊的接口，看图，特殊之处在于它继承了第二层所有的三个接口，而 ApplicationContext 没有。这点之后会用到。
5. 请先不用花时间在其他的接口和类上，先理解我说的这几点就可以了。

然后，请读者打开编辑器，翻一下 BeanFactory、ListableBeanFactory、HierarchicalBeanFactory、AutowireCapableBeanFactory、ApplicationContext 这几个接口的代码，大概看一下各个接口中的方法，大家心里要有底，限于篇幅，我就不贴代码介绍了。

启动过程分析

下面将会是冗长的代码分析，记住，一定要自己打开源码来看，不然纯看是很累的。

第一步，我们肯定要从 ClassPathXmlApplicationContext 的构造方法说起。

```
public class ClassPathXmlApplicationContext extends AbstractXmlApplicationContext
{
  private Resource[] configResources;

  // 如果已经有 ApplicationContext 并需要配置成父子关系，那么调用这个构造方法
  public ClassPathXmlApplicationContext(ApplicationContext parent) {
    super(parent);
```

```
  }
  ...
  public ClassPathXmlApplicationContext(String[] configLocations, boolean
refresh, ApplicationContext parent)
      throws BeansException {

    super(parent);
    // 根据提供的路径，处理成配置文件数组(以分号、逗号、空格、tab、换行符分割)
    setConfigLocations(configLocations);
    if (refresh) {
      refresh(); // 核心方法
    }
  }
    ...
}
```

接下来，就是 `refresh()`，这里简单说下为什么是 refresh()，而不是 init() 这种名字的方法。因为 ApplicationContext 建立起来以后，其实我们是可以通过调用 refresh() 这个方法重建的，refresh() 会将原来的 ApplicationContext 销毁，然后再重新执行一次初始化操作。

往下看，refresh() 方法里面调用了那么多方法，就知道肯定不简单了，请读者先看个大概，细节之后会详细说。

```
@Override
public void refresh() throws BeansException, IllegalStateException {
   // 来个锁，不然 refresh() 还没结束，你又来个启动或销毁容器的操作，那不就乱套了嘛
   synchronized (this.startupShutdownMonitor) {

      // 准备工作，记录下容器的启动时间、标记“已启动”状态、处理配置文件中的占位符
      prepareRefresh();

      // 这步比较关键，这步完成后，配置文件就会解析成一个个 Bean 定义，注册到 BeanFactory
中，
      // 当然，这里说的 Bean 还没有初始化，只是配置信息都提取出来了，
      // 注册也只是将这些信息都保存到了注册中心(说到底核心是一个 beanName-> beanDefinition
的 map)
      ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

      // 设置 BeanFactory 的类加载器，添加几个 BeanPostProcessor，手动注册几个特殊的
bean
      // 这块待会会展开说
      prepareBeanFactory(beanFactory);

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

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

```
            // 调用 BeanFactoryPostProcessor 各个实现类的
postProcessBeanFactory(factory) 方法
            invokeBeanFactoryPostProcessors(beanFactory);

            // 注册 BeanPostProcessor 的实现类，注意看和 BeanFactoryPostProcessor 的区别
            // 此接口两个方法: postProcessBeforeInitialization 和
postProcessAfterInitialization
            // 两个方法分别在 Bean 初始化之前和初始化之后得到执行。注意，到这里 Bean 还没初始化
            registerBeanPostProcessors(beanFactory);

            // 初始化当前 ApplicationContext 的 MessageSource，国际化这里就不展开说了，不然
没完没了了
            initMessageSource();

            // 初始化当前 ApplicationContext 的事件广播器，这里也不展开了
            initApplicationEventMulticaster();

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

            // 注册事件监听器，监听器需要实现 ApplicationListener 接口。这也不是我们的重点，过
            registerListeners();

            // 重点，重点，重点
            // 初始化所有的 singleton beans
            //（lazy-init 的除外）
            finishBeanFactoryInitialization(beanFactory);

            // 最后，广播事件，ApplicationContext 初始化完成
            finishRefresh();
        }

        catch (BeansException ex) {
            if (logger.isWarnEnabled()) {
                logger.warn("Exception encountered during context initialization - "
+
                        "cancelling refresh attempt: " + ex);
            }

            // Destroy already created singletons to avoid dangling resources.
            // 销毁已经初始化的 singleton 的 Beans，以免有些 bean 会一直占用资源
            destroyBeans();

            // Reset 'active' flag.
            cancelRefresh(ex);

            // 把异常往外抛
            throw ex;
        }

        finally {
```

```
            // Reset common introspection caches in Spring's core, since we
            // might not ever need metadata for singleton beans anymore...
            resetCommonCaches();
        }
    }
}
```

下面，我们开始一步步来肢解这个 refresh() 方法。

创建 Bean 容器前的准备工作

这个比较简单，直接看代码中的几个注释即可。

```
protected void prepareRefresh() {
    // 记录启动时间，
    // 将 active 属性设置为 true，closed 属性设置为 false，它们都是 AtomicBoolean 类型
    this.startupDate = System.currentTimeMillis();
    this.closed.set(false);
    this.active.set(true);

    if (logger.isInfoEnabled()) {
        logger.info("Refreshing " + this);
    }

    // Initialize any placeholder property sources in the context environment
    initPropertySources();

    // 校验 xml 配置文件
    getEnvironment().validateRequiredProperties();

    this.earlyApplicationEvents = new LinkedHashSet<ApplicationEvent>();
}
```

创建 Bean 容器，加载并注册 Bean

我们回到 refresh() 方法中的下一行 obtainFreshBeanFactory()。

注意，**这个方法是全文最重要的部分之一**，这里将会初始化 BeanFactory、加载 Bean、注册 Bean 等等。

当然，这步结束后，Bean 并没有完成初始化。这里指的是 Bean 实例并未在这一步生成。

// AbstractApplicationContext.java

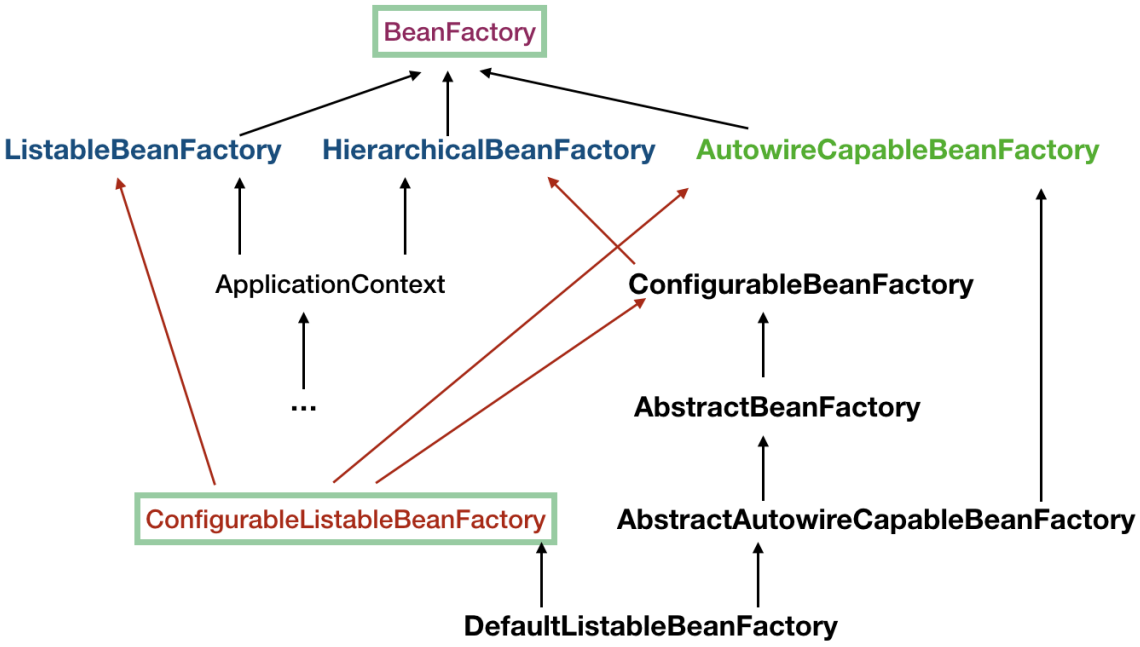
```
protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
   // 关闭旧的 BeanFactory（如果有），创建新的 BeanFactory，加载 Bean 定义、注册 Bean 等
等
   refreshBeanFactory();

   // 返回刚刚创建的 BeanFactory
   ConfigurableListableBeanFactory beanFactory = getBeanFactory();
   if (logger.isDebugEnabled()) {
      logger.debug("Bean factory for " + getDisplayName() + ": " + beanFactory);
   }
   return beanFactory;
}
```

// AbstractRefreshableApplicationContext.java 120

```
@Override
protected final void refreshBeanFactory() throws BeansException {
   // 如果 ApplicationContext 中已经加载过 BeanFactory 了，销毁所有 Bean，关闭
BeanFactory
   // 注意，应用中 BeanFactory 本来就是可以多个的，这里可不是说应用全局是否有 BeanFactory，
而是当前
   // ApplicationContext 是否有 BeanFactory
   if (hasBeanFactory()) {
      destroyBeans();
      closeBeanFactory();
   }
   try {
      // 初始化一个 DefaultListableBeanFactory，为什么用这个，我们马上说。
      DefaultListableBeanFactory beanFactory = createBeanFactory();
      // 用于 BeanFactory 的序列化，我想不部分人应该都用不到
      beanFactory.setSerializationId(getId());

      // 下面这两个方法很重要，别跟丢了，具体细节之后说
      // 设置 BeanFactory 的两个配置属性：是否允许 Bean 覆盖、是否允许循环引用
      customizeBeanFactory(beanFactory);

      // 加载 Bean 到 BeanFactory 中
      loadBeanDefinitions(beanFactory);
      synchronized (this.beanFactoryMonitor) {
         this.beanFactory = beanFactory;
      }
   }
   catch (IOException ex) {
      throw new ApplicationContextException("I/O error parsing bean definition
source for " + getDisplayName(), ex);
   }
}
```

> 看到这里的时候，我觉得读者就应该站在高处看 ApplicationContext 了，ApplicationContext 继承自 BeanFactory，但是它不应该被理解为 BeanFactory 的实现类，而是说其内部持有一个实例化的 BeanFactory（DefaultListableBeanFactory）。以后所有的 BeanFactory 相关的操作其实是委托给这个实例来处理的。

我们说说为什么选择实例化 **DefaultListableBeanFactory** ？前面我们说了有个很重要的接口 ConfigurableListableBeanFactory，它实现了 BeanFactory 下面一层的所有三个接口，我把之前的继承图再拿过来大家再仔细看一下：



我们可以看到 ConfigurableListableBeanFactory 只有一个实现类 DefaultListableBeanFactory，而且实现类 DefaultListableBeanFactory 还通过实现右边的 AbstractAutowireCapableBeanFactory 通吃了右路。所以结论就是，最底下这个家伙 DefaultListableBeanFactory 基本上是最牛的 BeanFactory 了，这也是为什么这边会使用这个类来实例化的原因。

> 如果你想要在程序运行的时候动态往 Spring IOC 容器注册新的 bean，就会使用到这个类。那我们怎么在运行时获得这个实例呢?
>
> 之前我们说过 ApplicationContext 接口能获取到 AutowireCapableBeanFactory，就是最右上角那个，然后它向下转型就能得到 DefaultListableBeanFactory 了。
>
> 那怎么拿到 ApplicationContext 实例呢？如果你不会，说明你没用过 Spring。

在继续往下之前，我们需要先了解 BeanDefinition。**我们说 BeanFactory 是 Bean 容器，那么 Bean 又是什么呢?**

这里的 BeanDefinition 就是我们所说的 Spring 的 Bean，我们自己定义的各个 Bean 其实会转换成一个个 BeanDefinition 存在于 Spring 的 BeanFactory 中。

所以，如果有人问你 Bean 是什么的时候，你要知道 Bean 在代码层面上可以简单认为是 BeanDefinition 的实例。

> BeanDefinition 中保存了我们的 Bean 信息，比如这个 Bean 指向的是哪个类、是否是单例的、是否懒加载、这个 Bean 依赖了哪些 Bean 等等。

BeanDefinition 接口定义

我们来看下 BeanDefinition 的接口定义：

```
public interface BeanDefinition extends AttributeAccessor, BeanMetadataElement {
```

```
    // 我们可以看到，默认只提供 sington 和 prototype 两种，
    // 很多读者可能知道还有 request, session, globalSession, application, websocket
这几种，
    // 不过，它们属于基于 web 的扩展。
    String SCOPE_SINGLETON = ConfigurableBeanFactory.SCOPE_SINGLETON;
    String SCOPE_PROTOTYPE = ConfigurableBeanFactory.SCOPE_PROTOTYPE;

    // 比较不重要，直接跳过吧
    int ROLE_APPLICATION = 0;
    int ROLE_SUPPORT = 1;
    int ROLE_INFRASTRUCTURE = 2;

    // 设置父 Bean，这里涉及到 bean 继承，不是 java 继承。请参见附录的详细介绍
    // 一句话就是：继承父 Bean 的配置信息而已
    void setParentName(String parentName);

    // 获取父 Bean
    String getParentName();

    // 设置 Bean 的类名称，将来是要通过反射来生成实例的
    void setBeanClassName(String beanClassName);

    // 获取 Bean 的类名称
    String getBeanClassName();


    // 设置 bean 的 scope
    void setScope(String scope);

    String getScope();

    // 设置是否懒加载
    void setLazyInit(boolean lazyInit);

    boolean isLazyInit();

    // 设置该 Bean 依赖的所有的 Bean，注意，这里的依赖不是指属性依赖(如 @Autowire 标记的)，
    // 是 depends-on="" 属性设置的值。
    void setDependsOn(String... dependsOn);

    // 返回该 Bean 的所有依赖
    String[] getDependsOn();

    // 设置该 Bean 是否可以注入到其他 Bean 中，只对根据类型注入有效，
    // 如果根据名称注入，即使这边设置了 false，也是可以的
    void setAutowireCandidate(boolean autowireCandidate);

    // 该 Bean 是否可以注入到其他 Bean 中
    boolean isAutowireCandidate();

    // 主要的。同一接口的多个实现，如果不指定名字的话，Spring 会优先选择设置 primary 为 true
的 bean
```

```
    void setPrimary(boolean primary);

    // 是否是 primary 的
    boolean isPrimary();

    // 如果该 Bean 采用工厂方法生成，指定工厂名称。对工厂不熟悉的读者，请参加附录
    // 一句话就是：有些实例不是用反射生成的，而是用工厂模式生成的
    void setFactoryBeanName(String factoryBeanName);
    // 获取工厂名称
    String getFactoryBeanName();
    // 指定工厂类中的 工厂方法名称
    void setFactoryMethodName(String factoryMethodName);
    // 获取工厂类中的 工厂方法名称
    String getFactoryMethodName();

    // 构造器参数
    ConstructorArgumentValues getConstructorArgumentValues();

    // Bean 中的属性值，后面给 bean 注入属性值的时候会说到
    MutablePropertyValues getPropertyValues();

    // 是否 singleton
    boolean isSingleton();

    // 是否 prototype
    boolean isPrototype();

    // 如果这个 Bean 是被设置为 abstract，那么不能实例化，
    // 常用于作为 父bean 用于继承，其实也很少用......
    boolean isAbstract();

    int getRole();
    String getDescription();
    String getResourceDescription();
    BeanDefinition getOriginatingBeanDefinition();
}
```

> 这个 BeanDefinition 其实已经包含很多的信息了，暂时不清楚所有的方法对应什么东西没关系，希望看完本文后读者可以彻底搞清楚里面的所有东西。
>
> 这里接口虽然那么多，但是没有类似 getInstance() 这种方法来获取我们定义的类的实例，真正的我们定义的类生成的实例到哪里去了呢？别着急，这个要很后面才能讲到。

有了 BeanDefinition 的概念以后，我们再往下看 refreshBeanFactory() 方法中的剩余部分：

```
customizeBeanFactory(beanFactory);
loadBeanDefinitions(beanFactory);
```

虽然只有两个方法，但路还很长啊。。。

customizeBeanFactory

customizeBeanFactory(beanFactory) 比较简单，就是配置是否允许 BeanDefinition 覆盖、是否允许循环引用。

```
protected void customizeBeanFactory(DefaultListableBeanFactory beanFactory) {
   if (this.allowBeanDefinitionOverriding != null) {
      // 是否允许 Bean 定义覆盖

 beanFactory.setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding)
;
   }
   if (this.allowCircularReferences != null) {
      // 是否允许 Bean 间的循环依赖
      beanFactory.setAllowCircularReferences(this.allowCircularReferences);
   }
}
```

BeanDefinition 的覆盖问题可能会有开发者碰到这个坑，就是在配置文件中定义 bean 时使用了相同的 id 或 name，默认情况下，allowBeanDefinitionOverriding 属性为 null，如果在同一配置文件中重复了，会抛错，但是如果不是同一配置文件中，会发生覆盖。

循环引用也很好理解：A 依赖 B，而 B 依赖 A。或 A 依赖 B，B 依赖 C，而 C 依赖 A。

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

至于这两个属性怎么配置？我在附录中进行了介绍，尤其对于覆盖问题，很多人都希望禁止出现 Bean 覆盖，可是 Spring 默认是不同文件的时候可以覆盖的。

之后的源码中还会出现这两个属性，读者有个印象就可以了，它们不是非常重要。

加载 Bean: loadBeanDefinitions

接下来是最重要的 loadBeanDefinitions(beanFactory) 方法了，这个方法将根据配置，加载各个 Bean，然后放到 BeanFactory 中。

读取配置的操作在 XmlBeanDefinitionReader 中，其负责加载配置、解析。

// AbstractXmlApplicationContext.java 80

```
/** 我们可以看到，此方法将通过一个 XmlBeanDefinitionReader 实例来加载各个 Bean。*/
@Override
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws
BeansException, IOException {
   // 给这个 BeanFactory 实例化一个 XmlBeanDefinitionReader
   XmlBeanDefinitionReader beanDefinitionReader = new
XmlBeanDefinitionReader(beanFactory);

   // Configure the bean definition reader with this context's
   // resource loading environment.
   beanDefinitionReader.setEnvironment(this.getEnvironment());
   beanDefinitionReader.setResourceLoader(this);
   beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));

   // 初始化 BeanDefinitionReader，其实这个是提供给子类覆写的，
   // 我看了一下，没有类覆写这个方法，我们姑且当做不重要吧
   initBeanDefinitionReader(beanDefinitionReader);
```

```
    // 重点来了，继续往下
    loadBeanDefinitions(beanDefinitionReader);
}
```

现在还在这个类中，接下来用刚刚初始化的 Reader 开始来加载 xml 配置，这块代码读者可以选择性跳过，不是很重要。也就是说，下面这个代码块，读者可以很轻松地略过。

// AbstractXmlApplicationContext.java 120

```
protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws
BeansException, IOException {
    Resource[] configResources = getConfigResources();
    if (configResources != null) {
        // 往下看
        reader.loadBeanDefinitions(configResources);
    }
    String[] configLocations = getConfigLocations();
    if (configLocations != null) {
        // 2
        reader.loadBeanDefinitions(configLocations);
    }
}

// 上面虽然有两个分支，不过第二个分支很快通过解析路径转换为 Resource 以后也会进到这里
@Override
public int loadBeanDefinitions(Resource... resources) throws
BeanDefinitionStoreException {
    Assert.notNull(resources, "Resource array must not be null");
    int counter = 0;
    // 注意这里是个 for 循环，也就是每个文件是一个 resource
    for (Resource resource : resources) {
        // 继续往下看
        counter += loadBeanDefinitions(resource);
    }
    // 最后返回 counter，表示总共加载了多少的 BeanDefinition
    return counter;
}

// XmlBeanDefinitionReader 303
@Override
public int loadBeanDefinitions(Resource resource) throws
BeanDefinitionStoreException {
    return loadBeanDefinitions(new EncodedResource(resource));
}

// XmlBeanDefinitionReader 314
public int loadBeanDefinitions(EncodedResource encodedResource) throws
BeanDefinitionStoreException {
    Assert.notNull(encodedResource, "EncodedResource must not be null");
    if (logger.isInfoEnabled()) {
        logger.info("Loading XML bean definitions from " +
encodedResource.getResource());
    }
```

```
    // 用一个 ThreadLocal 来存放配置文件资源
    Set<EncodedResource> currentResources =
this.resourcesCurrentlyBeingLoaded.get();
    if (currentResources == null) {
        currentResources = new HashSet<EncodedResource>(4);
        this.resourcesCurrentlyBeingLoaded.set(currentResources);
    }
    if (!currentResources.add(encodedResource)) {
        throw new BeanDefinitionStoreException(
                "Detected cyclic loading of " + encodedResource + " - check your
import definitions!");
    }
    try {
        InputStream inputStream = encodedResource.getResource().getInputStream();
        try {
            InputSource inputSource = new InputSource(inputStream);
            if (encodedResource.getEncoding() != null) {
                inputSource.setEncoding(encodedResource.getEncoding());
            }
            // 核心部分是这里，往下面看
            return doLoadBeanDefinitions(inputSource,
encodedResource.getResource());
        }
        finally {
            inputStream.close();
        }
    }
    catch (IOException ex) {
        throw new BeanDefinitionStoreException(
                "IOException parsing XML document from " +
encodedResource.getResource(), ex);
    }
    finally {
        currentResources.remove(encodedResource);
        if (currentResources.isEmpty()) {
            this.resourcesCurrentlyBeingLoaded.remove();
        }
    }
}

// 还在这个文件中，第 388 行
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
        throws BeanDefinitionStoreException {
    try {
        // 这里就不看了，将 xml 文件转换为 Document 对象
        Document doc = doLoadDocument(inputSource, resource);
        // 继续
        return registerBeanDefinitions(doc, resource);
    }
    catch (...
}
// 还在这个文件中，第 505 行
```

```
// 返回值：返回从当前配置文件加载了多少数量的 Bean
public int registerBeanDefinitions(Document doc, Resource resource) throws
BeanDefinitionStoreException {
   BeanDefinitionDocumentReader documentReader =
createBeanDefinitionDocumentReader();
   int countBefore = getRegistry().getBeanDefinitionCount();
   // 这里
   documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
   return getRegistry().getBeanDefinitionCount() - countBefore;
}
// DefaultBeanDefinitionDocumentReader 90
@Override
public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext)
{
   this.readerContext = readerContext;
   logger.debug("Loading bean definitions");
   Element root = doc.getDocumentElement();
   // 从 xml 根节点开始解析文件
   doRegisterBeanDefinitions(root);
}
```

经过漫长的链路，一个配置文件终于转换为一颗 DOM 树了，注意，这里指的是其中一个配置文件，不是所有的，读者可以看到上面有个 for 循环的。下面开始从根节点开始解析：

doRegisterBeanDefinitions：

```
// DefaultBeanDefinitionDocumentReader 116
protected void doRegisterBeanDefinitions(Element root) {
   // 我们看名字就知道，BeanDefinitionParserDelegate 必定是一个重要的类，它负责解析 Bean
定义，
   // 这里为什么要定义一个 parent? 看到后面就知道了，是递归问题，
   // 因为 <beans /> 内部是可以定义 <beans /> 的，所以这个方法的 root 其实不一定就是 xml
的根节点，也可以是嵌套在里面的 <beans /> 节点，从源码分析的角度，我们当做根节点就好了
   BeanDefinitionParserDelegate parent = this.delegate;
   this.delegate = createDelegate(getReaderContext(), root, parent);

   if (this.delegate.isDefaultNamespace(root)) {
      // 这块说的是根节点 <beans ... profile="dev" /> 中的 profile 是否是当前环境需要
的，
      // 如果当前环境配置的 profile 不包含此 profile，那就直接 return 了，不对此 <beans
/> 解析
      // 不熟悉 profile 为何物，不熟悉怎么配置 profile 读者的请移步附录区
      String profileSpec = root.getAttribute(PROFILE_ATTRIBUTE);
      if (StringUtils.hasText(profileSpec)) {
         String[] specifiedProfiles = StringUtils.tokenizeToStringArray(
               profileSpec,
BeanDefinitionParserDelegate.MULTI_VALUE_ATTRIBUTE_DELIMITERS);
         if
(!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles)) {
            if (logger.isInfoEnabled()) {
               logger.info("Skipped XML bean definition file due to specified
profiles [" + profileSpec +
                     "] not matching: " + getReaderContext().getResource());
```

```
            }
            return;
        }
    }
}

preProcessXml(root); // 钩子
// 往下看
parseBeanDefinitions(root, this.delegate);
postProcessXml(root); // 钩子

this.delegate = parent;
}
```

preProcessXml(root) 和 postProcessXml(root) 是给子类用的钩子方法，鉴于没有被使用到，也不是我们的重点，我们直接跳过。

这里涉及到了 profile 的问题，对于不了解的读者，我在附录中对 profile 做了简单的解释，读者可以参考一下。

接下来，看核心解析方法 parseBeanDefinitions(root, this.delegate) :

```
// default namespace 涉及到的就四个标签 <import />、<alias />、<bean /> 和 <beans />,
// 其他的属于 custom 的
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
    if (delegate.isDefaultNamespace(root)) {
        NodeList nl = root.getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            if (node instanceof Element) {
                Element ele = (Element) node;
                if (delegate.isDefaultNamespace(ele)) {
                    // 解析 default namespace 下面的几个元素
                    parseDefaultElement(ele, delegate);
                }
                else {
                    // 解析其他 namespace 的元素
                    delegate.parseCustomElement(ele);
                }
            }
        }
    }
    else {
        delegate.parseCustomElement(root);
    }
}
```

从上面的代码，我们可以看到，对于每个配置来说，分别进入到 parseDefaultElement(ele, delegate); 和 delegate.parseCustomElement(ele); 这两个分支了。

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

这里的四个标签之所以是 **default** 的，是因为它们是处于这个 namespace 下定义的：

```
http://www.springframework.org/schema/beans
```

又到初学者科普时间，不熟悉 namespace 的读者请看下面贴出来的 xml，这里的第二行 **xmlns** 就是咯。

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="
         http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd"
    default-autowire="byName">
```

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

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

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

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://www.springframework.org/schema/beans"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:mvc="http://www.springframework.org/schema/mvc"
   xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
    "
   default-autowire="byName">
```

同理，以后你要是碰到 `<dubbo />` 这种标签，那么就应该搜一搜是不是有 DubboNamespaceHandler 这个处理类。

回过神来，看看处理 default 标签的方法：

```
private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate
delegate) {
   if (delegate.nodeNameEquals(ele, IMPORT_ELEMENT)) {
      // 处理 <import /> 标签
      importBeanDefinitionResource(ele);
   }
   else if (delegate.nodeNameEquals(ele, ALIAS_ELEMENT)) {
      // 处理 <alias /> 标签定义
      // <alias name="fromName" alias="toName"/>
      processAliasRegistration(ele);
```

```
    }
    else if (delegate.nodeNameEquals(ele, BEAN_ELEMENT)) {
        // 处理 <bean /> 标签定义，这也算是我们的重点吧
        processBeanDefinition(ele, delegate);
    }
    else if (delegate.nodeNameEquals(ele, NESTED_BEANS_ELEMENT)) {
        // 如果碰到的是嵌套的 <beans /> 标签，需要递归
        doRegisterBeanDefinitions(ele);
    }
}
```

如果每个标签都说，那我不吐血，你们都要吐血了。我们挑我们的重点 `<bean />` 标签出来说。

processBeanDefinition 解析 bean 标签

下面是 processBeanDefinition 解析 `<bean />` 标签：

// DefaultBeanDefinitionDocumentReader 298

```
protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate
delegate) {
    // 将 <bean /> 节点中的信息提取出来，然后封装到一个 BeanDefinitionHolder 中，细节往下看
    BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);

    // 下面的几行先不要看，跳过先，跳过先，跳过先，后面会继续说的

    if (bdHolder != null) {
        bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
        try {
            // Register the final decorated instance.
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
getReaderContext().getRegistry());
        }
        catch (BeanDefinitionStoreException ex) {
            getReaderContext().error("Failed to register bean definition with name
'" +
                    bdHolder.getBeanName() + "'", ele, ex);
        }
        // Send registration event.
        getReaderContext().fireComponentRegistered(new
BeanComponentDefinition(bdHolder));
    }
}
```

继续往下看怎么解析之前，我们先看下 `<bean />` 标签中可以定义哪些属性：

Property	
class	类的全限定名
name	可指定 id、name(用逗号、分号、空格分隔)
scope	作用域
constructor arguments	指定构造参数
properties	设置属性的值
autowiring mode	no(默认值)、byName、byType、 constructor
lazy-initialization mode	是否懒加载(如果被非懒加载的bean依赖了那么其实也就不能懒加载了)
initialization method	bean 属性设置完成后，会调用这个方法
destruction method	bean 销毁后的回调方法

上面表格中的内容我想大家都非常熟悉吧，如果不熟悉，那就是你不够了解 Spring 的配置了。

简单地说就是像下面这样子：

```
<bean id="exampleBean" name="name1, name2, name3"
class="com.javadoop.ExampleBean"
      scope="singleton" lazy-init="true" init-method="init" destroy-
method="cleanup">

    <!-- 可以用下面三种形式指定构造参数 -->
    <constructor-arg type="int" value="7500000"/>
    <constructor-arg name="years" value="7500000"/>
    <constructor-arg index="0" value="7500000"/>

    <!-- property 的几种情况 -->
    <property name="beanOne">
        <ref bean="anotherExampleBean"/>
    </property>
    <property name="beanTwo" ref="yetAnotherBean"/>
    <property name="integerProperty" value="1"/>
</bean>
```

当然，除了上面举例出来的这些，还有 factory-bean、factory-method、`<lockup-method />`、`<replaced-method />`、`<meta />`、`<qualifier />` 这几个，大家是不是熟悉呢？自己检验一下自己对 Spring 中 bean 的了解程度。

有了以上这些知识以后，我们再继续往里看怎么解析 bean 元素，是怎么转换到 BeanDefinitionHolder 的。

// BeanDefinitionParserDelegate 428

```
public BeanDefinitionHolder parseBeanDefinitionElement(Element ele) {
    return parseBeanDefinitionElement(ele, null);
}
```

```
public BeanDefinitionHolder parseBeanDefinitionElement(Element ele,
BeanDefinition containingBean) {
   String id = ele.getAttribute(ID_ATTRIBUTE);
   String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);

   List<String> aliases = new ArrayList<String>();

   // 将 name 属性的定义按照 “逗号、分号、空格” 切分，形成一个 别名列表数组，
   // 当然，如果你不定义 name 属性的话，就是空的了
   // 我在附录中简单介绍了一下 id 和 name 的配置，大家可以看一眼，有个20秒就可以了
   if (StringUtils.hasLength(nameAttr)) {
      String[] nameArr = StringUtils.tokenizeToStringArray(nameAttr,
MULTI_VALUE_ATTRIBUTE_DELIMITERS);
      aliases.addAll(Arrays.asList(nameArr));
   }

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

   if (containingBean == null) {
      checkNameUniqueness(beanName, aliases, ele);
   }

   // 根据 <bean ...>...</bean> 中的配置创建 BeanDefinition，然后把配置中的信息都设置到实
例中，
   // 细节后面细说，先知道下面这行结束后，一个 BeanDefinition 实例就出来了。
   AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele,
beanName, containingBean);

   // 到这里，整个 <bean /> 标签就算解析结束了，一个 BeanDefinition 就形成了。
   if (beanDefinition != null) {
      // 如果都没有设置 id 和 name，那么此时的 beanName 就会为 null，进入下面这块代码产生
      // 如果读者不感兴趣的话，我觉得不需要关心这块代码，对本文源码分析来说，这些东西不重要
      if (!StringUtils.hasText(beanName)) {
         try {
            if (containingBean != null) {// 按照我们的思路，这里 containingBean 是
null 的
               beanName = BeanDefinitionReaderUtils.generateBeanName(
                     beanDefinition, this.readerContext.getRegistry(), true);
            }
            else {
               // 如果我们不定义 id 和 name，那么我们引言里的那个例子：
               //   1. beanName 为：com.javadoop.example.MessageServiceImpl#0
               //   2. beanClassName 为：com.javadoop.example.MessageServiceImpl
```

```
                beanName = this.readerContext.generateBeanName(beanDefinition);

                String beanClassName = beanDefinition.getBeanClassName();
                if (beanClassName != null &&
                      beanName.startsWith(beanClassName) && beanName.length() >
beanClassName.length() &&

!this.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {
                   // 把 beanClassName 设置为 Bean 的别名
                   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);
       // 返回 BeanDefinitionHolder
       return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);
    }

    return null;
}
```

然后，我们再看看怎么根据配置创建 BeanDefinition 实例的：

```
public AbstractBeanDefinition parseBeanDefinitionElement(
       Element ele, String beanName, BeanDefinition containingBean) {

    this.parseState.push(new BeanEntry(beanName));

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

    try {
       String parent = null;
       if (ele.hasAttribute(PARENT_ATTRIBUTE)) {
          parent = ele.getAttribute(PARENT_ATTRIBUTE);
       }
       // 创建 BeanDefinition，然后设置类信息而已，很简单，就不贴代码了
       AbstractBeanDefinition bd = createBeanDefinition(className, parent);

       // 设置 BeanDefinition 的一堆属性，这些属性定义在 AbstractBeanDefinition 中
```

```
            parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);
            bd.setDescription(DomUtils.getChildElementValueByTagName(ele,
DESCRIPTION_ELEMENT));

            /**
             * 下面的一堆是解析 <bean>......</bean> 内部的子元素，
             * 解析出来以后的信息都放到 bd 的属性中
             */

            // 解析 <meta />
            parseMetaElements(ele, bd);
            // 解析 <lookup-method />
            parseLookupOverrideSubElements(ele, bd.getMethodOverrides());
            // 解析 <replaced-method />
            parseReplacedMethodSubElements(ele, bd.getMethodOverrides());
        // 解析 <constructor-arg />
            parseConstructorArgElements(ele, bd);
            // 解析 <property />
            parsePropertyElements(ele, bd);
            // 解析 <qualifier />
            parseQualifierElements(ele, bd);

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

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

        return null;
    }
```

到这里，我们已经完成了根据 `<bean />` 配置创建了一个 BeanDefinitionHolder 实例。注意，是一个。

我们回到解析 `<bean />` 的入口方法:

```
protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate
delegate) {
    // 将 <bean /> 节点转换为 BeanDefinitionHolder，就是上面说的一堆
    BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
```

```
    if (bdHolder != null) {
        // 如果有自定义属性的话，进行相应的解析，先忽略
        bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
        try {
            // 我们把这步叫做 注册Bean 吧
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
getReaderContext().getRegistry());
        }
        catch (BeanDefinitionStoreException ex) {
            getReaderContext().error("Failed to register bean definition with name
'" +
                    bdHolder.getBeanName() + "'", ele, ex);
        }
        // 注册完成后，发送事件，本文不展开说这个
        getReaderContext().fireComponentRegistered(new
BeanComponentDefinition(bdHolder));
    }
}
```

大家再仔细看一下这块吧，我们后面就不回来说这个了。这里已经根据一个 `<bean />` 标签产生了一个 BeanDefinitionHolder 的实例，这个实例里面也就是一个 BeanDefinition 的实例和它的 beanName、aliases 这三个信息，注意，我们的关注点始终在 BeanDefinition 上：

```
public class BeanDefinitionHolder implements BeanMetadataElement {

  private final BeanDefinition beanDefinition;

  private final String beanName;

  private final String[] aliases;
...
```

然后我们准备注册这个 BeanDefinition，最后，把这个注册事件发送出去。

下面，我们开始说说注册 Bean 吧。

注册 Bean

// BeanDefinitionReaderUtils 143

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

    String beanName = definitionHolder.getBeanName();
    // 注册这个 Bean
    registry.registerBeanDefinition(beanName,
definitionHolder.getBeanDefinition());

    // 如果还有别名的话，也要根据别名全部注册一遍，不然根据别名就会找不到 Bean 了
    String[] aliases = definitionHolder.getAliases();
    if (aliases != null) {
        for (String alias : aliases) {
```

```
            // alias -> beanName 保存它们的别名信息，这个很简单，用一个 map 保存一下就可以
了，
            // 获取的时候，会先将 alias 转换为 beanName，然后再查找
            registry.registerAlias(beanName, alias);
        }
    }
}
```

别名注册的放一边，毕竟它很简单，我们看看怎么注册 Bean。

// DefaultListableBeanFactory 793

```
@Override
public void registerBeanDefinition(String beanName, BeanDefinition
beanDefinition)
        throws BeanDefinitionStoreException {

    Assert.hasText(beanName, "Bean name must not be empty");
    Assert.notNull(beanDefinition, "BeanDefinition must not be null");

    if (beanDefinition instanceof AbstractBeanDefinition) {
        try {
            ((AbstractBeanDefinition) beanDefinition).validate();
        }
        catch (BeanDefinitionValidationException ex) {
            throw new BeanDefinitionStoreException(...);
        }
    }

    // old? 还记得“允许 bean 覆盖”这个配置吗？allowBeanDefinitionOverriding
    BeanDefinition oldBeanDefinition;

    // 之后会看到，所有的 Bean 注册后会放入这个 beanDefinitionMap 中
    oldBeanDefinition = this.beanDefinitionMap.get(beanName);

    // 处理重复名称的 Bean 定义的情况
    if (oldBeanDefinition != null) {
        if (!isAllowBeanDefinitionOverriding()) {
            // 如果不允许覆盖的话，抛异常
            throw new
BeanDefinitionStoreException(beanDefinition.getResourceDescription()...
        }
        else if (oldBeanDefinition.getRole() < beanDefinition.getRole()) {
            // log...用框架定义的 Bean 覆盖用户自定义的 Bean
        }
        else if (!beanDefinition.equals(oldBeanDefinition)) {
            // log...用新的 Bean 覆盖旧的 Bean
        }
        else {
            // log...用同等的 Bean 覆盖旧的 Bean，这里指的是 equals 方法返回 true 的 Bean
        }
        // 覆盖
        this.beanDefinitionMap.put(beanName, beanDefinition);
```

```
    }
    else {
        // 判断是否已经有其他的 Bean 开始初始化了.
        // 注意，"注册Bean" 这个动作结束，Bean 依然还没有初始化，我们后面会有大篇幅说初始化过
程，
        // 在 Spring 容器启动的最后，会 预初始化 所有的 singleton beans
        if (hasBeanCreationStarted()) {
            // Cannot modify startup-time collection elements anymore (for stable
iteration)
            synchronized (this.beanDefinitionMap) {
                this.beanDefinitionMap.put(beanName, beanDefinition);
                List<String> updatedDefinitions = new ArrayList<String>
(this.beanDefinitionNames.size() + 1);
                updatedDefinitions.addAll(this.beanDefinitionNames);
                updatedDefinitions.add(beanName);
                this.beanDefinitionNames = updatedDefinitions;
                if (this.manualSingletonNames.contains(beanName)) {
                    Set<String> updatedSingletons = new LinkedHashSet<String>
(this.manualSingletonNames);
                    updatedSingletons.remove(beanName);
                    this.manualSingletonNames = updatedSingletons;
                }
            }
        }
        else {
            // 最正常的应该是进到这个分支。

            // 将 BeanDefinition 放到这个 map 中，这个 map 保存了所有的 BeanDefinition
            this.beanDefinitionMap.put(beanName, beanDefinition);
            // 这是个 ArrayList，所以会按照 bean 配置的顺序保存每一个注册的 Bean 的名字
            this.beanDefinitionNames.add(beanName);
            // 这是个 LinkedHashSet，代表的是手动注册的 singleton bean，
            // 注意这里是 remove 方法，到这里的 Bean 当然不是手动注册的
            // 手动指的是通过调用以下方法注册的 bean ：
            //     registerSingleton(String beanName, Object singletonObject)
            // 这不是重点，解释只是为了不让大家疑惑。Spring 会在后面"手动"注册一些 Bean，
            // 如 "environment"、"systemProperties" 等 bean，我们自己也可以在运行时注册
Bean 到容器中的
            this.manualSingletonNames.remove(beanName);
        }
        // 这个不重要，在预初始化的时候会用到，不必管它。
        this.frozenBeanDefinitionNames = null;
    }

    if (oldBeanDefinition != null || containsSingleton(beanName)) {
        resetBeanDefinition(beanName);
    }
}
```

总结一下，到这里已经初始化了 Bean 容器，`<bean />` 配置也相应的转换为了一个个 BeanDefinition，然后注册了各个 BeanDefinition 到注册中心，并且发送了注册事件。

--------- 分割线 ---------

到这里是一个分水岭，前面的内容都还算比较简单，不过应该也比较繁琐，大家要清楚地知道前面都做了哪些事情。

Bean 容器实例化完成后

说到这里，我们回到 refresh() 方法，我重新贴了一遍代码，看看我们说到哪了。是的，我们才说完 obtainFreshBeanFactory() 方法。

考虑到篇幅，这里开始大幅缩减掉没必要详细介绍的部分，大家直接看下面的代码中的注释就好了。

```
@Override
public void refresh() throws BeansException, IllegalStateException {
   // 来个锁，不然 refresh() 还没结束，你又来个启动或销毁容器的操作，那不就乱套了嘛
   synchronized (this.startupShutdownMonitor) {

      // 准备工作，记录下容器的启动时间、标记“已启动”状态、处理配置文件中的占位符
      prepareRefresh();

      // 这步比较关键，这步完成后，配置文件就会解析成一个个 Bean 定义，注册到 BeanFactory
中，
      // 当然，这里说的 Bean 还没有初始化，只是配置信息都提取出来了，
      // 注册也只是将这些信息都保存到了注册中心(说到底核心是一个 beanName-> beanDefinition
的 map)
      ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

      // 设置 BeanFactory 的类加载器，添加几个 BeanPostProcessor，手动注册几个特殊的
bean
      // 这块待会会展开说
      prepareBeanFactory(beanFactory);

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

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



         // 注册 BeanPostProcessor 的实现类，注意看和 BeanFactoryPostProcessor 的区别
         // 此接口两个方法: postProcessBeforeInitialization 和
postProcessAfterInitialization
         // 两个方法分别在 Bean 初始化之前和初始化之后得到执行。这里仅仅是注册，之后会看到回调
这两方法的时机
         registerBeanPostProcessors(beanFactory);
```

```
            // 初始化当前 ApplicationContext 的 MessageSource，国际化这里就不展开说了，不然
没完没了了
            initMessageSource();

            // 初始化当前 ApplicationContext 的事件广播器，这里也不展开了
            initApplicationEventMulticaster();

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

            // 注册事件监听器，监听器需要实现 ApplicationListener 接口。这也不是我们的重点，过
            registerListeners();

            // 重点，重点，重点
            // 初始化所有的 singleton beans
            //（lazy-init 的除外）
            finishBeanFactoryInitialization(beanFactory);

            // 最后，广播事件，ApplicationContext 初始化完成，不展开
            finishRefresh();
        }

        catch (BeansException ex) {
            if (logger.isWarnEnabled()) {
                logger.warn("Exception encountered during context initialization - "
+
                        "cancelling refresh attempt: " + ex);
            }

            // Destroy already created singletons to avoid dangling resources.
            // 销毁已经初始化的 singleton 的 Beans，以免有些 bean 会一直占用资源
            destroyBeans();

            // Reset 'active' flag.
            cancelRefresh(ex);

            // 把异常往外抛
            throw ex;
        }

        finally {
            // Reset common introspection caches in Spring's core, since we
            // might not ever need metadata for singleton beans anymore...
            resetCommonCaches();
        }
    }
}
```

准备 Bean 容器: prepareBeanFactory

之前我们说过，Spring 把我们在 xml 配置的 bean 都注册以后，会"手动"注册一些特殊的 bean。

这里简单介绍下 prepareBeanFactory(factory) 方法：

```
/**
 * Configure the factory's standard context characteristics,
 * such as the context's ClassLoader and post-processors.
 * @param beanFactory the BeanFactory to configure
 */
protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
   // 设置 BeanFactory 的类加载器，我们知道 BeanFactory 需要加载类，也就需要类加载器，
   // 这里设置为加载当前 ApplicationContext 类的类加载器
   beanFactory.setBeanClassLoader(getClassLoader());

   // 设置 BeanExpressionResolver
   beanFactory.setBeanExpressionResolver(new
StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));
   //
   beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this,
getEnvironment()));

   // 添加一个 BeanPostProcessor，这个 processor 比较简单：
   // 实现了 Aware 接口的 beans 在初始化的时候，这个 processor 负责回调，
   // 这个我们很常用，如我们会为了获取 ApplicationContext 而 implement
ApplicationContextAware
   // 注意：它不仅仅回调 ApplicationContextAware，
   //    还会负责回调 EnvironmentAware、ResourceLoaderAware 等，看下源码就清楚了
   beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));

   // 下面几行的意思就是，如果某个 bean 依赖于以下几个接口的实现类，在自动装配的时候忽略它们，
   // Spring 会通过其他方式来处理这些依赖。
   beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
   beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
   beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
   beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
   beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
   beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);

   /**
    * 下面几行就是为特殊的几个 bean 赋值，如果有 bean 依赖了以下几个，会注入这边相应的值，
    * 之前我们说过，"当前 ApplicationContext 持有一个 BeanFactory"，这里解释了第一行。
    * ApplicationContext 还继承了 ResourceLoader、ApplicationEventPublisher、
MessageSource
    * 所以对于这几个依赖，可以赋值为 this，注意 this 是一个 ApplicationContext
    * 那这里怎么没看到为 MessageSource 赋值呢？那是因为 MessageSource 被注册成为了一个普通
的 bean
    */
   beanFactory.registerResolvableDependency(BeanFactory.class, beanFactory);
   beanFactory.registerResolvableDependency(ResourceLoader.class, this);
   beanFactory.registerResolvableDependency(ApplicationEventPublisher.class,
this);
   beanFactory.registerResolvableDependency(ApplicationContext.class, this);
```

```
    // 这个 BeanPostProcessor 也很简单，在 bean 实例化后，如果是 ApplicationListener 的
子类，
    // 那么将其添加到 listener 列表中，可以理解成：注册 事件监听器
    beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(this));

    // 这里涉及到特殊的 bean，名为：loadTimeWeaver，这不是我们的重点，忽略它
    // tips: ltw 是 AspectJ 的概念，指的是在运行期进行织入，这个和 Spring AOP 不一样，
    //     感兴趣的读者请参考我写的关于 AspectJ 的另一篇文章
https://www.javadoop.com/post/aspectj
    if (beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
        beanFactory.addBeanPostProcessor(new
LoadTimeWeaverAwareProcessor(beanFactory));
        // Set a temporary ClassLoader for type matching.
        beanFactory.setTempClassLoader(new
ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));
    }

    /**
     * 从下面几行代码我们可以知道，Spring 往往很 "智能" 就是因为它会帮我们默认注册一些有用的
bean，
     * 我们也可以选择覆盖
     */

    // 如果没有定义 "environment" 这个 bean，那么 Spring 会 "手动" 注册一个
    if (!beanFactory.containsLocalBean(ENVIRONMENT_BEAN_NAME)) {
        beanFactory.registerSingleton(ENVIRONMENT_BEAN_NAME, getEnvironment());
    }
    // 如果没有定义 "systemProperties" 这个 bean，那么 Spring 会 "手动" 注册一个
    if (!beanFactory.containsLocalBean(SYSTEM_PROPERTIES_BEAN_NAME)) {
        beanFactory.registerSingleton(SYSTEM_PROPERTIES_BEAN_NAME,
getEnvironment().getSystemProperties());
    }
    // 如果没有定义 "systemEnvironment" 这个 bean，那么 Spring 会 "手动" 注册一个
    if (!beanFactory.containsLocalBean(SYSTEM_ENVIRONMENT_BEAN_NAME)) {
        beanFactory.registerSingleton(SYSTEM_ENVIRONMENT_BEAN_NAME,
getEnvironment().getSystemEnvironment());
    }
}
```

在上面这块代码中，Spring 对一些特殊的 bean 进行了处理，读者如果暂时还不能消化它们也没有关系，慢慢往下看。

初始化所有的 singleton beans

我们的重点当然是 `finishBeanFactoryInitialization(beanFactory);` 这个巨头了，这里会负责初始化所有的 singleton beans。

注意，后面的描述中，我都会使用**初始化**或**预初始化**来代表这个阶段，Spring 会在这个阶段完成所有的 singleton beans 的实例化。

我们来总结一下，到目前为止，应该说 BeanFactory 已经创建完成，并且所有的实现了 BeanFactoryPostProcessor 接口的 Bean 都已经初始化并且其中的 postProcessBeanFactory(factory) 方法已经得到回调执行了。而且 Spring 已经“手动”注册了一些特殊的 Bean，如 `environment`、`systemProperties` 等。

剩下的就是初始化 singleton beans 了，我们知道它们是单例的，如果没有设置懒加载，那么 Spring 会在接下来初始化所有的 singleton beans。

// AbstractApplicationContext.java 834

```
// 初始化剩余的 singleton beans
protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory
beanFactory) {

   // 首先，初始化名字为 conversionService 的 Bean。本着送佛送到西的精神，我在附录中简单介
绍了一下 ConversionService，因为这实在太实用了
   // 什么，看代码这里没有初始化 Bean 啊！
   // 注意了，初始化的动作包装在 beanFactory.getBean(...) 中，这里先不说细节，先往下看吧
   if (beanFactory.containsBean(CONVERSION_SERVICE_BEAN_NAME) &&
         beanFactory.isTypeMatch(CONVERSION_SERVICE_BEAN_NAME,
ConversionService.class)) {
      beanFactory.setConversionService(
            beanFactory.getBean(CONVERSION_SERVICE_BEAN_NAME,
ConversionService.class));
   }

   // Register a default embedded value resolver if no bean post-processor
   // (such as a PropertyPlaceholderConfigurer bean) registered any before:
   // at this point, primarily for resolution in annotation attribute values.
   if (!beanFactory.hasEmbeddedValueResolver()) {
      beanFactory.addEmbeddedValueResolver(new StringValueResolver() {
         @Override
         public String resolveStringValue(String strVal) {
            return getEnvironment().resolvePlaceholders(strVal);
         }
      });
   }

   // 先初始化 LoadTimeWeaverAware 类型的 Bean
   // 之前也说过，这是 AspectJ 相关的内容，放心跳过吧
   String[] weaverAwareNames =
beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, false);
   for (String weaverAwareName : weaverAwareNames) {
      getBean(weaverAwareName);
   }

   // Stop using the temporary ClassLoader for type matching.
   beanFactory.setTempClassLoader(null);

   // 没什么别的目的，因为到这一步的时候，Spring 已经开始预初始化 singleton beans 了，
   // 肯定不希望这个时候还出现 bean 定义解析、加载、注册。
   beanFactory.freezeConfiguration();
```

```
    // 开始初始化
    beanFactory.preInstantiateSingletons();
}
```

从上面最后一行往里看，我们就又回到 DefaultListableBeanFactory 这个类了，这个类大家应该都不陌生了吧。

preInstantiateSingletons

// DefaultListableBeanFactory 728

```
@Override
public void preInstantiateSingletons() throws BeansException {
    if (this.logger.isDebugEnabled()) {
        this.logger.debug("Pre-instantiating singletons in " + this);
    }
    // this.beanDefinitionNames 保存了所有的 beanNames
    List<String> beanNames = new ArrayList<String>(this.beanDefinitionNames);

    // 下面这个循环，触发所有的非懒加载的 singleton beans 的初始化操作
    for (String beanName : beanNames) {

        // 合并父 Bean 中的配置，注意 <bean id="" class="" parent="" /> 中的 parent，用的不多吧，
        // 考虑到这可能会影响大家的理解，我在附录中解释了一下 "Bean 继承"，不了解的请到附录中看一下
        RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);

        // 非抽象、非懒加载的 singletons。如果配置了 'abstract = true'，那是不需要初始化的
        if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
            // 处理 FactoryBean(读者如果不熟悉 FactoryBean，请移步附录区了解)
            if (isFactoryBean(beanName)) {
                // FactoryBean 的话，在 beanName 前面加上 ‘&’ 符号。再调用 getBean，getBean 方法别急
                final FactoryBean<?> factory = (FactoryBean<?>) getBean(FACTORY_BEAN_PREFIX + beanName);
                // 判断当前 FactoryBean 是否是 SmartFactoryBean 的实现，此处忽略，直接跳过
                boolean isEagerInit;
                if (System.getSecurityManager() != null && factory instanceof SmartFactoryBean) {
                    isEagerInit = AccessController.doPrivileged(new PrivilegedAction<Boolean>() {
                        @Override
                        public Boolean run() {
                            return ((SmartFactoryBean<?>) factory).isEagerInit();
                        }
                    }, getAccessControlContext());
                }
                else {
                    isEagerInit = (factory instanceof SmartFactoryBean &&
                            ((SmartFactoryBean<?>) factory).isEagerInit());
                }
                if (isEagerInit) {
```

```
                getBean(beanName);
            }
         }
         else {
            // 对于普通的 Bean，只要调用 getBean(beanName) 这个方法就可以进行初始化了
            getBean(beanName);
         }
      }
   }

   // 到这里说明所有的非懒加载的 singleton beans 已经完成了初始化
   // 如果我们定义的 bean 是实现了 SmartInitializingSingleton 接口的，那么在这里得到回
调，忽略
   for (String beanName : beanNames) {
      Object singletonInstance = getSingleton(beanName);
      if (singletonInstance instanceof SmartInitializingSingleton) {
         final SmartInitializingSingleton smartSingleton =
(SmartInitializingSingleton) singletonInstance;
         if (System.getSecurityManager() != null) {
            AccessController.doPrivileged(new PrivilegedAction<Object>() {
               @Override
               public Object run() {
                  smartSingleton.afterSingletonsInstantiated();
                  return null;
               }
            }, getAccessControlContext());
         }
         else {
            smartSingleton.afterSingletonsInstantiated();
         }
      }
   }
}
```

接下来，我们就进入到 getBean(beanName) 方法了，这个方法我们经常用来从 BeanFactory 中获取一个 Bean，而初始化的过程也封装到了这个方法里。

getBean

在继续前进之前，读者应该具备 FactoryBean 的知识，如果读者还不熟悉，请移步附录部分了解 FactoryBean。

// AbstractBeanFactory 196

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

// 我们在剖析初始化 Bean 的过程，但是 getBean 方法我们经常是用来从容器中获取 Bean 用的，注意
切换思路，
// 已经初始化过了就从容器中直接返回，否则就先初始化再返回
```

```
@SuppressWarnings("unchecked")
protected <T> T doGetBean(
      final String name, final Class<T> requiredType, final Object[] args,
boolean typeCheckOnly)
      throws BeansException {
   // 获取一个 “正统的” beanName，处理两种情况，一个是前面说的 FactoryBean(前面带 ‘&’)，
   // 一个是别名问题，因为这个方法是 getBean，获取 Bean 用的，你要是传一个别名进来，是完全可
以的
   final String beanName = transformedBeanName(name);

   // 注意跟着这个，这个是返回值
   Object bean;

   // 检查下是不是已经创建过了
   Object sharedInstance = getSingleton(beanName);

   // 这里说下 args 呗，虽然看上去一点不重要。前面我们一路进来的时候都是
getBean(beanName)，
   // 所以 args 传参其实是 null 的，但是如果 args 不为空的时候，那么意味着调用方不是希望获取
Bean，而是创建 Bean
   if (sharedInstance != null && args == null) {
      if (logger.isDebugEnabled()) {
         if (isSingletonCurrentlyInCreation(beanName)) {
            logger.debug("...");
         }
         else {
            logger.debug("Returning cached instance of singleton bean '" +
beanName + "'");
         }
      }
      // 下面这个方法：如果是普通 Bean 的话，直接返回 sharedInstance，
      // 如果是 FactoryBean 的话，返回它创建的那个实例对象
      // (FactoryBean 知识，读者若不清楚请移步附录)
      bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
   }

   else {
      if (isPrototypeCurrentlyInCreation(beanName)) {
         // 创建过了此 beanName 的 prototype 类型的 bean，那么抛异常，
         // 往往是因为陷入了循环引用
         throw new BeanCurrentlyInCreationException(beanName);
      }

      // 检查一下这个 BeanDefinition 在容器中是否存在
      BeanFactory parentBeanFactory = getParentBeanFactory();
      if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
         // 如果当前容器不存在这个 BeanDefinition，试试父容器中有没有
         String nameToLookup = originalBeanName(name);
         if (args != null) {
            // 返回父容器的查询结果
            return (T) parentBeanFactory.getBean(nameToLookup, args);
         }
```

```
            else {
                // No args -> delegate to standard getBean method.
                return parentBeanFactory.getBean(nameToLookup, requiredType);
            }
        }

        if (!typeCheckOnly) {
            // typeCheckOnly 为 false，将当前 beanName 放入一个 alreadyCreated 的 Set 集合中。
            markBeanAsCreated(beanName);
        }

        /*
         * 稍稍总结一下：
         * 到这里的话，要准备创建 Bean 了，对于 singleton 的 Bean 来说，容器中还没创建过此 Bean；
         * 对于 prototype 的 Bean 来说，本来就是要创建一个新的 Bean。
         */
        try {
            final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
            checkMergedBeanDefinition(mbd, beanName, args);

            // 先初始化依赖的所有 Bean，这个很好理解。
            // 注意，这里的依赖指的是 depends-on 中定义的依赖
            String[] dependsOn = mbd.getDependsOn();
            if (dependsOn != null) {
                for (String dep : dependsOn) {
                    // 检查是不是有循环依赖，这里的循环依赖和我们前面说的循环依赖又不一样，这里肯定是不允许出现的，不然要乱套了，读者想一下就知道了
                    if (isDependent(beanName, dep)) {
                        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                                "Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");
                    }
                    // 注册一下依赖关系
                    registerDependentBean(dep, beanName);
                    // 先初始化被依赖项
                    getBean(dep);
                }
            }

            // 如果是 singleton scope 的，创建 singleton 的实例
            if (mbd.isSingleton()) {
                sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {
                    @Override
                    public Object getObject() throws BeansException {
                        try {
                            // 执行创建 Bean，详情后面再说
                            return createBean(beanName, mbd, args);
                        }
                        catch (BeansException ex) {
```

```
                    destroySingleton(beanName);
                    throw ex;
                }
            }
        });
        bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
    }

    // 如果是 prototype scope 的，创建 prototype 的实例
    else if (mbd.isPrototype()) {
        // It's a prototype -> create a new instance.
        Object prototypeInstance = null;
        try {
            beforePrototypeCreation(beanName);
            // 执行创建 Bean
            prototypeInstance = createBean(beanName, mbd, args);
        }
        finally {
            afterPrototypeCreation(beanName);
        }
        bean = getObjectForBeanInstance(prototypeInstance, name, beanName,
mbd);
    }

    // 如果不是 singleton 和 prototype 的话，需要委托给相应的实现类来处理
    else {
        String scopeName = mbd.getScope();
        final Scope scope = this.scopes.get(scopeName);
        if (scope == null) {
            throw new IllegalStateException("No Scope registered for scope
name '" + scopeName + "'");
        }
        try {
            Object scopedInstance = scope.get(beanName, new
ObjectFactory<Object>() {
                @Override
                public Object getObject() throws BeansException {
                    beforePrototypeCreation(beanName);
                    try {
                        // 执行创建 Bean
                        return createBean(beanName, mbd, args);
                    }
                    finally {
                        afterPrototypeCreation(beanName);
                    }
                }
            });
            bean = getObjectForBeanInstance(scopedInstance, name, beanName,
mbd);
        }
        catch (IllegalStateException ex) {
            throw new BeanCreationException(beanName,
```

```
                    "Scope '" + scopeName + "' is not active for the current
thread; consider " +
                    "defining a scoped proxy for this bean if you intend to
refer to it from a singleton",
                    ex);
            }
         }
      }
      catch (BeansException ex) {
         cleanupAfterBeanCreationFailure(beanName);
         throw ex;
      }
   }

   // 最后，检查一下类型对不对，不对的话就抛异常，对的话就返回了
   if (requiredType != null && bean != null && !requiredType.isInstance(bean)) {
      try {
         return getTypeConverter().convertIfNecessary(bean, requiredType);
      }
      catch (TypeMismatchException ex) {
         if (logger.isDebugEnabled()) {
            logger.debug("Failed to convert bean '" + name + "' to required type
'" +
                  ClassUtils.getQualifiedName(requiredType) + "'", ex);
         }
         throw new BeanNotOfRequiredTypeException(name, requiredType,
bean.getClass());
      }
   }
   return (T) bean;
}
```

大家应该也猜到了，接下来当然是分析 createBean 方法：

```
protected abstract Object createBean(String beanName, RootBeanDefinition mbd,
Object[] args) throws BeanCreationException;
```

第三个参数 args 数组代表创建实例需要的参数，不就是给构造方法用的参数，或者是工厂 Bean 的参数嘛，不过要注意，在我们的初始化阶段，args 是 null。

这回我们要到一个新的类了 AbstractAutowireCapableBeanFactory，看类名，AutowireCapable？类名是不是也说明了点问题了。

主要是为了以下场景，采用 @Autowired 注解注入属性值：

```
public class MessageServiceImpl implements MessageService {
    @Autowired
    private UserService userService;

    public String getMessage() {
        return userService.getMessage();
    }
}
```

```
<bean id="messageService" class="com.javadoop.example.MessageServiceImpl" />
```

以上这种属于混用了 xml 和 注解 两种方式的配置方式，Spring 会处理这种情况。

好了，读者要知道这么回事就可以了，继续向前。

// AbstractAutowireCapableBeanFactory 447

```
/**
 * Central method of this class: creates a bean instance,
 * populates the bean instance, applies post-processors, etc.
 * @see #doCreateBean
 */
@Override
protected Object createBean(String beanName, RootBeanDefinition mbd, Object[]
args) throws BeanCreationException {
   if (logger.isDebugEnabled()) {
      logger.debug("Creating instance of bean '" + beanName + "'");
   }
   RootBeanDefinition mbdToUse = mbd;

   // 确保 BeanDefinition 中的 Class 被加载
   Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
   if (resolvedClass != null && !mbd.hasBeanClass() && mbd.getBeanClassName() !=
null) {
      mbdToUse = new RootBeanDefinition(mbd);
      mbdToUse.setBeanClass(resolvedClass);
   }

   // 准备方法覆写，这里又涉及到一个概念：MethodOverrides，它来自于 bean 定义中的 <lookup-
method />
   // 和 <replaced-method />，如果读者感兴趣，回到 bean 解析的地方看看对这两个标签的解析。
   // 我在附录中也对这两个标签的相关知识点进行了介绍，读者可以移步去看看
   try {
      mbdToUse.prepareMethodOverrides();
   }
   catch (BeanDefinitionValidationException ex) {
      throw new BeanDefinitionStoreException(mbdToUse.getResourceDescription(),
            beanName, "Validation of method overrides failed", ex);
   }

   try {
      // 让 InstantiationAwareBeanPostProcessor 在这一步有机会返回代理，
      // 在 《Spring AOP 源码分析》那篇文章中有解释，这里先跳过
      Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
      if (bean != null) {
         return bean;
      }
   }
   catch (Throwable ex) {
      throw new BeanCreationException(mbdToUse.getResourceDescription(),
beanName,
            "BeanPostProcessor before instantiation of bean failed", ex);
```

```
    }
    // 重头戏，创建 bean
    Object beanInstance = doCreateBean(beanName, mbdToUse, args);
    if (logger.isDebugEnabled()) {
        logger.debug("Finished creating instance of bean '" + beanName + "'");
    }
    return beanInstance;
}
```

创建 Bean

我们继续往里看 doCreateBean 这个方法：

```
/**
 * Actually create the specified bean. Pre-creation processing has already
happened
 * at this point, e.g. checking {@code postProcessBeforeInstantiation} callbacks.
 * <p>Differentiates between default bean instantiation, use of a
 * factory method, and autowiring a constructor.
 * @param beanName the name of the bean
 * @param mbd the merged bean definition for the bean
 * @param args explicit arguments to use for constructor or factory method
invocation
 * @return a new instance of the bean
 * @throws BeanCreationException if the bean could not be created
 * @see #instantiateBean
 * @see #instantiateUsingFactoryMethod
 * @see #autowireConstructor
 */
protected Object doCreateBean(final String beanName, final RootBeanDefinition
mbd, final Object[] args)
      throws BeanCreationException {

    // Instantiate the bean.
    BeanWrapper instanceWrapper = null;
    if (mbd.isSingleton()) {
        instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
    }
    if (instanceWrapper == null) {
        // 说明不是 FactoryBean，这里实例化 Bean，这里非常关键，细节之后再说
        instanceWrapper = createBeanInstance(beanName, mbd, args);
    }
    // 这个就是 Bean 里面的 我们定义的类 的实例，很多地方我直接描述成 "bean 实例"
    final Object bean = (instanceWrapper != null ?
instanceWrapper.getWrappedInstance() : null);
    // 类型
    Class<?> beanType = (instanceWrapper != null ?
instanceWrapper.getWrappedClass() : null);
    mbd.resolvedTargetType = beanType;

    // 建议跳过吧，涉及接口：MergedBeanDefinitionPostProcessor
    synchronized (mbd.postProcessingLock) {
```

```
      if (!mbd.postProcessed) {
         try {
            // MergedBeanDefinitionPostProcessor，这个我真不展开说了，直接跳过吧，很少用
的
            applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
         }
         catch (Throwable ex) {
            throw new BeanCreationException(mbd.getResourceDescription(),
beanName,
                  "Post-processing of merged bean definition failed", ex);
         }
         mbd.postProcessed = true;
      }
   }

   // Eagerly cache singletons to be able to resolve circular references
   // even when triggered by lifecycle interfaces like BeanFactoryAware.
   // 下面这块代码是为了解决循环依赖的问题，以后有时间，我再对循环依赖这个问题进行解析吧
   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");
      }
      addSingletonFactory(beanName, new ObjectFactory<Object>() {
         @Override
         public Object getObject() throws BeansException {
            return getEarlyBeanReference(beanName, mbd, bean);
         }
      });
   }

   // Initialize the bean instance.
   Object exposedObject = bean;
   try {
      // 这一步也是非常关键的，这一步负责属性装配，因为前面的实例只是实例化了，并没有设值，这里
就是设值
      populateBean(beanName, mbd, instanceWrapper);
      if (exposedObject != null) {
         // 还记得 init-method 吗？还有 InitializingBean 接口？还有 BeanPostProcessor
接口？
         // 这里就是处理 bean 初始化完成后的各种回调
         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) {
      //
      Object earlySingletonReference = getSingleton(beanName, false);
      if (earlySingletonReference != null) {
         if (exposedObject == bean) {
            exposedObject = earlySingletonReference;
         }
         else if (!this.allowRawInjectionDespiteWrapping &&
hasDependentBean(beanName)) {
            String[] dependentBeans = getDependentBeans(beanName);
            Set<String> actualDependentBeans = new LinkedHashSet<String>
(dependentBeans.length);
            for (String dependentBean : dependentBeans) {
               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.");
            }
         }
      }
   }

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

```
    return exposedObject;
}
```

到这里，我们已经分析完了 doCreateBean 方法，总的来说，我们已经说完了整个初始化流程。

接下来我们挑 doCreateBean 中的三个细节出来说说。一个是创建 Bean 实例的 createBeanInstance 方法，一个是依赖注入的 populateBean 方法，还有就是回调方法 initializeBean。

注意了，接下来的这三个方法要认真说那也是极其复杂的，很多地方我就点到为止了，感兴趣的读者可以自己往里看，最好就是碰到不懂的，自己写代码去调试它。

创建 Bean 实例

我们先看看 createBeanInstance 方法。需要说明的是，这个方法如果每个分支都分析下去，必然也是极其复杂冗长的，我们挑重点说。此方法的目的就是实例化我们指定的类。

```
protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd,
Object[] args) {
    // 确保已经加载了此 class
    Class<?> beanClass = resolveBeanClass(mbd, beanName);

    // 校验一下这个类的访问权限
    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)  {
        // 采用工厂方法实例化，不熟悉这个概念的读者请看附录，注意，不是 FactoryBean
        return instantiateUsingFactoryMethod(beanName, mbd, args);
    }

    // 如果不是第一次创建，比如第二次创建 prototype bean。
    // 这种情况下，我们可以从第一次创建知道，采用无参构造函数，还是构造函数依赖注入 来完成实例化
    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) {
            // 构造函数依赖注入
            return autowireConstructor(beanName, mbd, null, null);
        }
        else {
            // 无参构造函数
```

```
            return instantiateBean(beanName, mbd);
        }
    }

    // 判断是否采用有参构造函数
    Constructor<?>[] ctors =
determineConstructorsFromBeanPostProcessors(beanClass, beanName);
    if (ctors != null ||
            mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_CONSTRUCTOR
||
            mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args))  {
        // 构造函数依赖注入
        return autowireConstructor(beanName, mbd, ctors, args);
    }

    // 调用无参构造函数
    return instantiateBean(beanName, mbd);
}
```

挑个简单的**无参构造函数**构造实例来看看：

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

                    return getInstantiationStrategy().instantiate(mbd, beanName,
parent);
                }
            }, getAccessControlContext());
        }
        else {
            // 实例化
            beanInstance = getInstantiationStrategy().instantiate(mbd, beanName,
parent);
        }
        // 包装一下，返回
        BeanWrapper bw = new BeanWrapperImpl(beanInstance);
        initBeanWrapper(bw);
        return bw;
    }
    catch (Throwable ex) {
        throw new BeanCreationException(
                mbd.getResourceDescription(), beanName, "Instantiation of bean
failed", ex);
    }
```

```
}
```

我们可以看到，关键的地方在于：

```
beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);
```

这里会进行实际的实例化过程，我们进去看看：

// SimpleInstantiationStrategy 59

```
@Override
public Object instantiate(RootBeanDefinition bd, String beanName, BeanFactory
owner) {

   // 如果不存在方法覆写，那就使用 java 反射进行实例化，否则使用 CGLIB，
   // 方法覆写 请参见附录"方法注入"中对 lookup-method 和 replaced-method 的介绍
   if (bd.getMethodOverrides().isEmpty()) {
      Constructor<?> constructorToUse;
      synchronized (bd.constructorArgumentLock) {
         constructorToUse = (Constructor<?>)
bd.resolvedConstructorOrFactoryMethod;
         if (constructorToUse == null) {
            final Class<?> clazz = bd.getBeanClass();
            if (clazz.isInterface()) {
               throw new BeanInstantiationException(clazz, "Specified class is an
interface");
            }
            try {
               if (System.getSecurityManager() != null) {
                  constructorToUse = AccessController.doPrivileged(new
PrivilegedExceptionAction<Constructor<?>>() {
                     @Override
                     public Constructor<?> run() throws Exception {
                        return clazz.getDeclaredConstructor((Class[]) null);
                     }
                  });
               }
               else {
                  constructorToUse = clazz.getDeclaredConstructor((Class[])
null);
               }
               bd.resolvedConstructorOrFactoryMethod = constructorToUse;
            }
            catch (Throwable ex) {
               throw new BeanInstantiationException(clazz, "No default
constructor found", ex);
            }
         }
      }
      // 利用构造方法进行实例化
      return BeanUtils.instantiateClass(constructorToUse);
   }
   else {
```

```
        // 存在方法覆写，利用 CGLIB 来完成实例化，需要依赖于 CGLIB 生成子类，这里就不展开了。
        // tips: 因为如果不使用 CGLIB 的话，存在 override 的情况 JDK 并没有提供相应的实例化
支持
        return instantiateWithMethodInjection(bd, beanName, owner);
    }
}
```

到这里，我们就算实例化完成了。我们开始说怎么进行属性注入。

bean 属性注入

看完了 createBeanInstance(...) 方法，我们来看看 populateBean(...) 方法，该方法负责进行属性设值，处理依赖。

// AbstractAutowireCapableBeanFactory 1203

```
protected void populateBean(String beanName, RootBeanDefinition mbd, BeanWrapper
bw) {
    // bean 实例的所有属性都在这里了
    PropertyValues pvs = mbd.getPropertyValues();

    if (bw == null) {
        if (!pvs.isEmpty()) {
            throw new BeanCreationException(
                    mbd.getResourceDescription(), beanName, "Cannot apply property
values to null instance");
        }
        else {
            // Skip property population phase for null instance.
            return;
        }
    }

    // 到这步的时候，bean 实例化完成（通过工厂方法或构造方法），但是还没开始属性设值，
    // InstantiationAwareBeanPostProcessor 的实现类可以在这里对 bean 进行状态修改，
    // 我也没找到有实际的使用，所以我们暂且忽略这块吧
    boolean continueWithPropertyPopulation = true;
    if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof InstantiationAwareBeanPostProcessor) {
                InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor) bp;
                // 如果返回 false，代表不需要进行后续的属性设值，也不需要再经过其他的
BeanPostProcessor 的处理
                if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(),
beanName)) {
                    continueWithPropertyPopulation = false;
                    break;
                }
            }
        }
    }
```

```
    if (!continueWithPropertyPopulation) {
        return;
    }

    if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME ||
            mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
        MutablePropertyValues newPvs = new MutablePropertyValues(pvs);

        // 通过名字找到所有属性值，如果是 bean 依赖，先初始化依赖的 bean。记录依赖关系
        if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME) {
            autowireByName(beanName, mbd, bw, newPvs);
        }

        // 通过类型装配。复杂一些
        if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
            autowireByType(beanName, mbd, bw, newPvs);
        }

        pvs = newPvs;
    }

    boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
    boolean needsDepCheck = (mbd.getDependencyCheck() !=
RootBeanDefinition.DEPENDENCY_CHECK_NONE);

    if (hasInstAwareBpps || needsDepCheck) {
        PropertyDescriptor[] filteredPds =
filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
        if (hasInstAwareBpps) {
            for (BeanPostProcessor bp : getBeanPostProcessors()) {
                if (bp instanceof InstantiationAwareBeanPostProcessor) {
                    InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor) bp;
                    // 这里有个非常有用的 BeanPostProcessor 进到这里:
AutowiredAnnotationBeanPostProcessor
                    // 对采用 @Autowired、@Value 注解的依赖进行设值，这里的内容也是非常丰富的，
不过本文不会展开说了，感兴趣的读者请自行研究
                    pvs = ibp.postProcessPropertyValues(pvs, filteredPds,
bw.getWrappedInstance(), beanName);
                    if (pvs == null) {
                        return;
                    }
                }
            }
        }
        if (needsDepCheck) {
            checkDependencies(beanName, mbd, filteredPds, pvs);
        }
    }
    // 设置 bean 实例的属性值
    applyPropertyValues(beanName, mbd, bw, pvs);
}
```

initializeBean

属性注入完成后，这一步其实就是处理各种回调了，这块代码比较简单。

```
protected Object initializeBean(final String beanName, final Object bean,
RootBeanDefinition mbd) {
   if (System.getSecurityManager() != null) {
      AccessController.doPrivileged(new PrivilegedAction<Object>() {
         @Override
         public Object run() {
            invokeAwareMethods(beanName, bean);
            return null;
         }
      }, getAccessControlContext());
   }
   else {
      // 如果 bean 实现了 BeanNameAware、BeanClassLoaderAware 或 BeanFactoryAware
接口，回调
      invokeAwareMethods(beanName, bean);
   }

   Object wrappedBean = bean;
   if (mbd == null || !mbd.isSynthetic()) {
      // BeanPostProcessor 的 postProcessBeforeInitialization 回调
      wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean,
beanName);
   }

   try {
      // 处理 bean 中定义的 init-method，
      // 或者如果 bean 实现了 InitializingBean 接口，调用 afterPropertiesSet() 方法
      invokeInitMethods(beanName, wrappedBean, mbd);
   }
   catch (Throwable ex) {
      throw new BeanCreationException(
            (mbd != null ? mbd.getResourceDescription() : null),
            beanName, "Invocation of init method failed", ex);
   }

   if (mbd == null || !mbd.isSynthetic()) {
      // BeanPostProcessor 的 postProcessAfterInitialization 回调
      wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean,
beanName);
   }
   return wrappedBean;
}
```

大家发现没有，BeanPostProcessor 的两个回调都发生在这边，只不过中间处理了 init-method，是不是和读者原来的认知有点不一样了？

附录

id 和 name

每个 Bean 在 Spring 容器中都有一个唯一的名字（beanName）和 0 个或多个别名（aliases）。

我们从 Spring 容器中获取 Bean 的时候，可以根据 beanName，也可以通过别名。

```
beanFactory.getBean("beanName or alias");
```

在配置 `<bean />` 的过程中，我们可以配置 id 和 name，看几个例子就知道是怎么回事了。

```
<bean id="messageService" name="m1, m2, m3"
class="com.javadoop.example.MessageServiceImpl">
```

以上配置的结果就是：beanName 为 messageService，别名有 3 个，分别为 m1、m2、m3。

```
<bean name="m1, m2, m3" class="com.javadoop.example.MessageServiceImpl" />
```

以上配置的结果就是：beanName 为 m1，别名有 2 个，分别为 m2、m3。

```
<bean class="com.javadoop.example.MessageServiceImpl">
```

beanName 为：com.javadoop.example.MessageServiceImpl#0，

别名 1 个，为： com.javadoop.example.MessageServiceImpl

```
<bean id="messageService" class="com.javadoop.example.MessageServiceImpl">
```

以上配置的结果就是：beanName 为 messageService，没有别名。

配置是否允许 Bean 覆盖、是否允许循环依赖

我们说过，默认情况下，allowBeanDefinitionOverriding 属性为 null。如果在同一配置文件中 Bean id 或 name 重复了，会抛错，但是如果不是同一配置文件中，会发生覆盖。

可是有些时候我们希望在系统启动的过程中就严格杜绝发生 Bean 覆盖，因为万一出现这种情况，会增加我们排查问题的成本。

循环依赖说的是 A 依赖 B，而 B 又依赖 A。或者是 A 依赖 B，B 依赖 C，而 C 却依赖 A。默认 allowCircularReferences 也是 null。

它们两个属性是一起出现的，必然可以在同一个地方一起进行配置。

添加这两个属性的作者 Juergen Hoeller 在这个 jira 的讨论中说明了怎么配置这两个属性。

```
public class NoBeanOverridingContextLoader extends ContextLoader {

  @Override
  protected void customizeContext(ServletContext servletContext,
ConfigurableWebApplicationContext applicationContext) {
    super.customizeContext(servletContext, applicationContext);
    AbstractRefreshableApplicationContext arac =
(AbstractRefreshableApplicationContext) applicationContext;
    arac.setAllowBeanDefinitionOverriding(false);
  }
}
```

```
public class MyContextLoaderListener extends
org.springframework.web.context.ContextLoaderListener {

  @Override
  protected ContextLoader createContextLoader() {
    return new NoBeanOverridingContextLoader();
  }

}
```

```
<listener>
    <listener-class>com.javadoop.MyContextLoaderListener</listener-class>
</listener>
```

如果以上方式不能满足你的需求，请参考这个链接：解决spring中不同配置文件中存在name或者id相同的bean可能引起的问题

profile

我们可以把不同环境的配置分别配置到单独的文件中，举个例子：

```
<beans profile="development"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="...">

    <jdbc:embedded-database id="dataSource">
        <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
        <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
    </jdbc:embedded-database>
</beans>
```

```
<beans profile="production"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

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

应该不必做过多解释了吧，看每个文件第一行的 profile=""。

当然，我们也可以在一个配置文件中使用：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

    <beans profile="development">
```

```
        <jdbc:embedded-database id="dataSource">
            <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
            <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
        </jdbc:embedded-database>
    </beans>

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

理解起来也很简单吧。

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

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

最简单的方式莫过于在程序启动的时候指定：

```
-Dspring.profiles.active="profile1,profile2"
```

> profile 可以激活多个

当然，我们也可以通过代码的形式从 Environment 中设置 profile：

```
AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext();
ctx.getEnvironment().setActiveProfiles("development");
ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);
ctx.refresh(); // 重启
```

如果是 Spring Boot 的话更简单，我们一般会创建 application.properties、application-dev.properties、application-prod.properties 等文件，其中 application.properties 配置各个环境通用的配置，application-{profile}.properties 中配置特定环境的配置，然后在启动的时候指定 profile：

```
java -Dspring.profiles.active=prod -jar JavaDoop.jar
```

如果是单元测试中使用的话，在测试类中使用 @ActiveProfiles 指定，这里就不展开了。

工厂模式生成 Bean

请读者注意 factory-bean 和 FactoryBean 的区别。这节说的是前者，是说静态工厂或实例工厂，而后者是 Spring 中的特殊接口，代表一类特殊的 Bean，附录的下面一节会介绍 FactoryBean。

设计模式里，工厂方法模式分静态工厂和实例工厂，我们分别看看 Spring 中怎么配置这两个，来个代码示例就什么都清楚了。

静态工厂：

```xml
<bean id="clientService"
    class="examples.ClientService"
    factory-method="createInstance"/>
```

```java
public class ClientService {
    private static ClientService clientService = new ClientService();
    private ClientService() {}

    // 静态方法
    public static ClientService createInstance() {
        return clientService;
    }
}
```

实例工厂：

```xml
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- inject any dependencies required by this locator bean -->
</bean>

<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>

<bean id="accountService"
    factory-bean="serviceLocator"
    factory-method="createAccountServiceInstance"/>
```

```java
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();

    private static AccountService accountService = new AccountServiceImpl();

    public ClientService createClientServiceInstance() {
        return clientService;
    }

    public AccountService createAccountServiceInstance() {
        return accountService;
    }
}
```

FactoryBean

FactoryBean 适用于 Bean 的创建过程比较复杂的场景，比如数据库连接池的创建。

```java
public interface FactoryBean<T> {
    T getObject() throws Exception;
    Class<T> getObjectType();
    boolean isSingleton();
}
```

```
public class Person {
    private Car car ;
    private void setCar(Car car){ this.car = car;  }
}
```

我们假设现在需要创建一个 Person 的 Bean，首先我们需要一个 Car 的实例，我们这里假设 Car 的实例创建很麻烦，那么我们可以把创建 Car 的复杂过程包装起来：

```
public class MyCarFactoryBean implements FactoryBean<Car>{
    private String make;
    private int year ;

    public void setMake(String m){ this.make =m ; }

    public void setYear(int y){ this.year = y; }

    public Car getObject(){
      // 这里我们假设 Car 的实例化过程非常复杂，反正就不是几行代码可以写完的那种
      CarBuilder cb = CarBuilder.car();

      if(year!=0) cb.setYear(this.year);
      if(StringUtils.hasText(this.make)) cb.setMake( this.make );
      return cb.factory();
    }

    public Class<Car> getObjectType() { return Car.class ; }

    public boolean isSingleton() { return false; }
}
```

我们看看装配的时候是怎么配置的：

```
<bean class = "com.javadoop.MyCarFactoryBean" id = "car">
  <property name = "make" value ="Honda"/>
  <property name = "year" value ="1984"/>
</bean>
<bean class = "com.javadoop.Person" id = "josh">
  <property name = "car" ref = "car"/>
</bean>
```

看到不一样了吗？id 为 “car” 的 bean 其实指定的是一个 FactoryBean，不过配置的时候，我们直接让配置 Person 的 Bean 直接依赖于这个 FactoryBean 就可以了。中间的过程 Spring 已经封装好了。

说到这里，我们再来点干货。我们知道，现在还用 xml 配置 Bean 依赖的越来越少了，更多时候，我们可能会采用 java config 的方式来配置，这里有什么不一样呢？

```
@Configuration
public class CarConfiguration {

    @Bean
    public MyCarFactoryBean carFactoryBean(){
      MyCarFactoryBean cfb = new MyCarFactoryBean();
```

```
        cfb.setMake("Honda");
        cfb.setYear(1984);
        return cfb;
    }

    @Bean
    public Person aPerson(){
    Person person = new Person();
      // 注意这里的不同
    person.setCar(carFactoryBean().getObject());
    return person;
    }
}
```

这个时候，其实我们的思路也很简单，把 MyCarFactoryBean 看成是一个简单的 Bean 就可以了，不必理会什么 FactoryBean，它是不是 FactoryBean 和我们没关系。

初始化 Bean 的回调

有以下四种方案：

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>
```

```
public class AnotherExampleBean implements InitializingBean {

    public void afterPropertiesSet() {
        // do some initialization work
    }
}
```

```
@Bean(initMethod = "init")
public Foo foo() {
    return new Foo();
}
```

```
@PostConstruct
public void init() {

}
```

销毁 Bean 的回调

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-
method="cleanup"/>
```

```
public class AnotherExampleBean implements DisposableBean {

    public void destroy() {
        // do some destruction work (like releasing pooled connections)
    }
}
```

```
@Bean(destroyMethod = "cleanup")
public Bar bar() {
    return new Bar();
}
```

```
@PreDestroy
public void cleanup() {

}
```

ConversionService

既然文中说到了这个，顺便提一下好了。

最有用的场景就是，它用来将前端传过来的参数和后端的 controller 方法上的参数进行绑定的时候用。

像前端传过来的字符串、整数要转换为后端的 String、Integer 很容易，但是如果 controller 方法需要的是一个枚举值，或者是 Date 这些非基础类型（含基础类型包装类）值的时候，我们就可以考虑采用 ConversionService 来进行转换。

```
<bean id="conversionService"
  class="org.springframework.context.support.ConversionServiceFactoryBean">
  <property name="converters">
    <list>
      <bean class="com.javadoop.learning.utils.StringToEnumConverterFactory"/>
    </list>
  </property>
</bean>
```

ConversionService 接口很简单，所以要自定义一个 convert 的话也很简单。

下面再说一个实现这种转换很简单的方式，那就是实现 Converter 接口。

来看一个很简单的例子，这样比什么都管用。

```
public class StringToDateConverter implements Converter<String, Date> {

    @Override
    public Date convert(String source) {
        try {
            return DateUtils.parseDate(source, "yyyy-MM-dd", "yyyy-MM-dd
HH:mm:ss", "yyyy-MM-dd HH:mm", "HH:mm:ss", "HH:mm");
        } catch (ParseException e) {
            return null;
        }
    }
}
```

只要注册这个 Bean 就可以了。这样，前端往后端传的时间描述字符串就很容易绑定成 Date 类型了，不需要其他任何操作。

Bean 继承

在初始化 Bean 的地方，我们说过了这个：

```
RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
```

这里涉及到的就是 `<bean parent="" />` 中的 parent 属性，我们来看看 Spring 中是用这个来干什么的。

首先，我们要明白，这里的继承和 java 语法中的继承没有任何关系，不过思路是相通的。child bean 会继承 parent bean 的所有配置，也可以覆盖一些配置，当然也可以新增额外的配置。

Spring 中提供了继承自 AbstractBeanDefinition 的 `ChildBeanDefinition` 来表示 child bean。

看如下一个例子：

```
<bean id="inheritedTestBean" abstract="true"
class="org.springframework.beans.TestBean">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>

<bean id="inheritsWithDifferentClass"
class="org.springframework.beans.DerivedTestBean"
        parent="inheritedTestBean" init-method="initialize">

    <property name="name" value="override"/>
</bean>
```

parent bean 设置了 `abstract="true"` 所以它不会被实例化，child bean 继承了 parent bean 的两个属性，但是对 name 属性进行了覆写。

child bean 会继承 scope、构造器参数值、属性值、init-method、destroy-method 等等。

当然，我不是说 parent bean 中的 abstract = true 在这里是必须的，只是说如果加上了以后 Spring 在实例化 singleton beans 的时候会忽略这个 bean。

比如下面这个极端 parent bean，它没有指定 class，所以毫无疑问，这个 bean 的作用就是用来充当模板用的 parent bean，此处就必须加上 abstract = true。

```
<bean id="inheritedTestBeanWithoutClass" abstract="true">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>
```

方法注入

一般来说，我们的应用中大多数的 Bean 都是 singleton 的。singleton 依赖 singleton，或者 prototype 依赖 prototype 都很好解决，直接设置属性依赖就可以了。

但是，如果是 singleton 依赖 prototype 呢？这个时候不能用属性依赖，因为如果用属性依赖的话，我们每次其实拿到的还是第一次初始化时候的 bean。

一种解决方案就是不要用属性依赖，每次获取依赖的 bean 的时候从 BeanFactory 中取。这个也是大家最常用的方式了吧。怎么取，我就不介绍了，大部分 Spring 项目大家都会定义那么个工具类的。

另一种解决方案就是这里要介绍的通过使用 Lookup method。

lookup-method

我们来看一下 Spring Reference 中提供的一个例子：

```
package fiona.apple;

// no more Spring imports!

public abstract class CommandManager {

    public Object process(Object commandState) {
        // grab a new instance of the appropriate Command interface
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

    // okay... but where is the implementation of this method?
    protected abstract Command createCommand();
}
```

xml 配置 `<lookup-method />`：

```
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="myCommand" class="fiona.apple.AsyncCommand" scope="prototype">
    <!-- inject dependencies here as required -->
</bean>

<!-- commandProcessor uses statefulCommandHelper -->
<bean id="commandManager" class="fiona.apple.CommandManager">
    <lookup-method name="createCommand" bean="myCommand"/>
</bean>
```

Spring 采用 **CGLIB 生成字节码** 的方式来生成一个子类。我们定义的类不能定义为 final class，抽象方法上也不能加 final。

lookup-method 上的配置也可以采用注解来完成，这样就可以不用配置 `<lookup-method />` 了，其他不变：

```
public abstract class CommandManager {

    public Object process(Object commandState) {
        MyCommand command = createCommand();
        command.setState(commandState);
        return command.execute();
    }

    @Lookup("myCommand")
    protected abstract Command createCommand();
}
```

> 注意，既然用了注解，要配置注解扫描：`<context:component-scan base-package="com.javadoop" />`

甚至，我们可以像下面这样：

```
public abstract class CommandManager {

    public Object process(Object commandState) {
        MyCommand command = createCommand();
        command.setState(commandState);
        return command.execute();
    }

    @Lookup
    protected abstract MyCommand createCommand();
}
```

> 上面的返回值用了 MyCommand，当然，如果 Command 只有一个实现类，那返回值也可以写 Command。

replaced-method

记住它的功能，就是替换掉 bean 中的一些方法。

```
public class MyValueCalculator {

    public String computeValue(String input) {
        // some real code...
    }

    // some other methods...
}
```

方法覆写，注意要实现 MethodReplacer 接口：

```
public class ReplacementComputeValue implements
org.springframework.beans.factory.support.MethodReplacer {

    public Object reimplement(Object o, Method m, Object[] args) throws Throwable
{
        // get the input value, work with it, and return a computed result
        String input = (String) args[0];
        ...
        return ...;
    }
}
```

配置也很简单：

```
<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">
    <!-- 定义 computeValue 这个方法要被替换掉 -->
    <replaced-method name="computeValue" replacer="replacementComputeValue">
        <arg-type>String</arg-type>
    </replaced-method>
</bean>

<bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>
```

> arg-type 明显不是必须的，除非存在方法重载，这样必须通过参数类型列表来判断这里要覆盖哪个方法。

BeanPostProcessor

应该说 BeanPostProcessor 概念在 Spring 中也是比较重要的。我们看下接口定义：

```
public interface BeanPostProcessor {

   Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException;

   Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException;

}
```

看这个接口中的两个方法名字我们大体上可以猜测 bean 在初始化之前会执行 postProcessBeforeInitialization 这个方法，初始化完成之后会执行 postProcessAfterInitialization 这个方法。但是，这么理解是非常片面的。

首先，我们要明白，除了我们自己定义的 BeanPostProcessor 实现外，Spring 容器在启动时自动给我们也加了几个。如在获取 BeanFactory 的 obtainFactory() 方法结束后的 prepareBeanFactory(factory)，大家仔细看会发现，Spring 往容器中添加了这两个 BeanPostProcessor：ApplicationContextAwareProcessor、ApplicationListenerDetector。

我们回到这个接口本身，读者请看第一个方法，这个方法接受的第一个参数是 bean 实例，第二个参数是 bean 的名字，重点在返回值将会作为新的 bean 实例，所以，没事的话这里不能随便返回个 null。

那意味着什么呢？我们很容易想到的就是，我们这里可以对一些我们想要修饰的 bean 实例做一些事情。但是对于 Spring 框架来说，它会决定是不是要在这个方法中返回 bean 实例的代理，这样就有更大的想象空间了。

最后，我们说说如果我们自己定义一个 bean 实现 BeanPostProcessor 的话，它的执行时机是什么时候？

如果仔细看了代码分析的话，其实很容易知道了，在 bean 实例化完成、属性注入完成之后，会执行回调方法，具体请参见类 AbstractAutowireCapableBeanFactory#initBean 方法。

首先会回调几个实现了 Aware 接口的 bean，然后就开始回调 BeanPostProcessor 的 postProcessBeforeInitialization 方法，之后是回调 init-method，然后再回调 BeanPostProcessor 的 postProcessAfterInitialization 方法。

总结

按理说，总结应该写在附录前面，我就不讲究了。

在花了那么多时间后，这篇文章终于算是基本写完了，大家在惊叹 Spring 给我们做了那么多的事的时候，应该透过现象看本质，去理解 Spring 写得好的地方，去理解它的设计思想。

本文的缺陷在于对 Spring 预初始化 singleton beans 的过程分析不够，主要是代码量真的比较大，分支旁路众多。同时，虽然附录条目不少，但是庞大的 Spring 真的引出了很多的概念，希望日后有精力可以慢慢补充一些。

（全文完）