IOC组成：

BeanFactory、单例池、BeanPostProcessor、Map等的集合，很多博文说IOC是Map集合，这个观点是错的，他只是IOC一个成员。

多例Bean的使用场景

之所以用多例，是为了**防止并发问题**；即一个请求改变了对象的状态，此时对象又处理另一个请求，而之前请求对对象状态的改变导致了对象对另一个请求做了错误的处理；

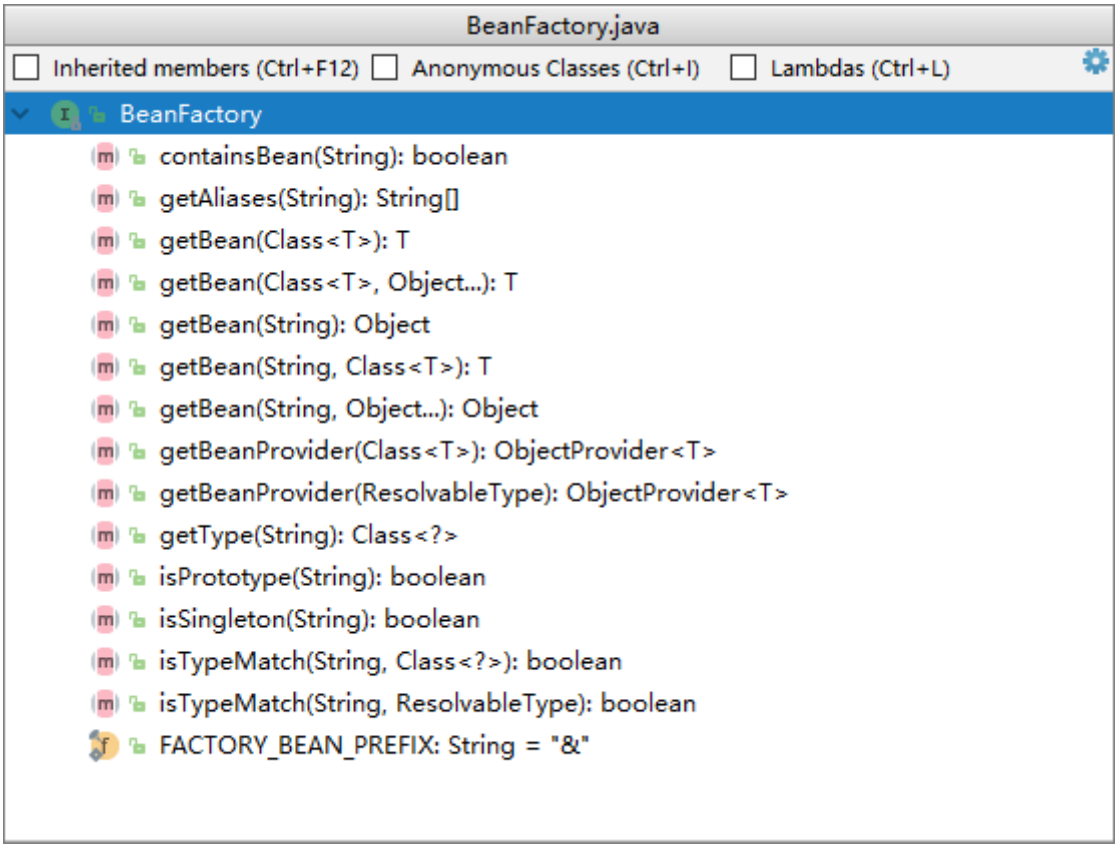
ApplicationContext和BeanFactory：

ApplicationContext是容器的高级接口，BeanFacotry（顶级容器/根容器，规范了/定义了容器的基础行为）（去了解这几个接口的关系）

springioc容器组成：map是ioc容器的一个成员，称之为单例池（signletonObjects）；

ioc容器是一组组件和过程的集合，包括BeanFactory、单例池、BeanPostProcessor等以及之间的协作流程

BeanFactory：



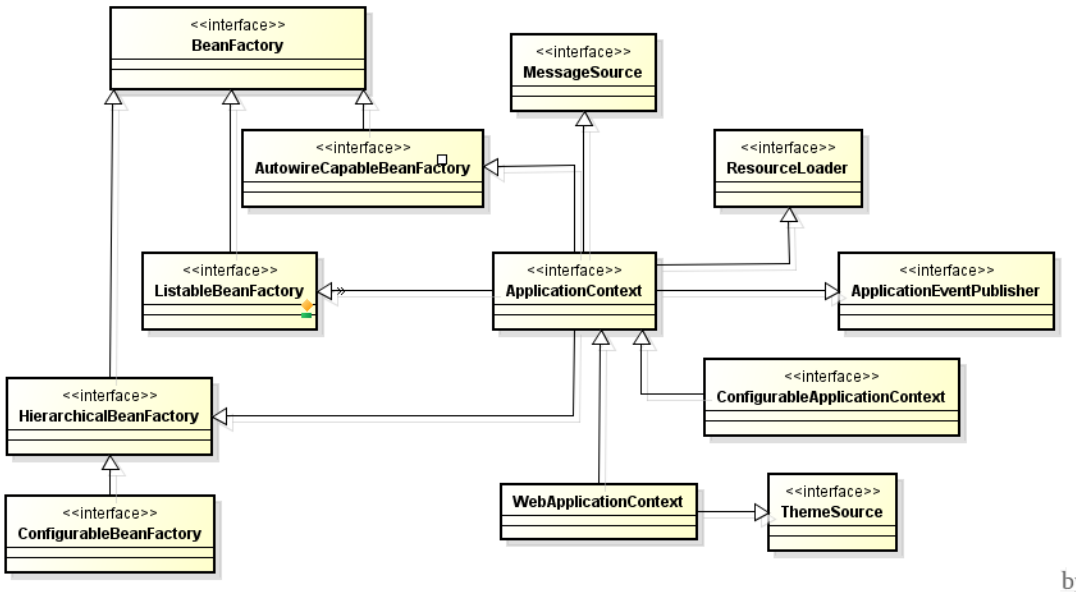
图中getbean有几种：

根据id（String）或根据id+类型（Class）获取

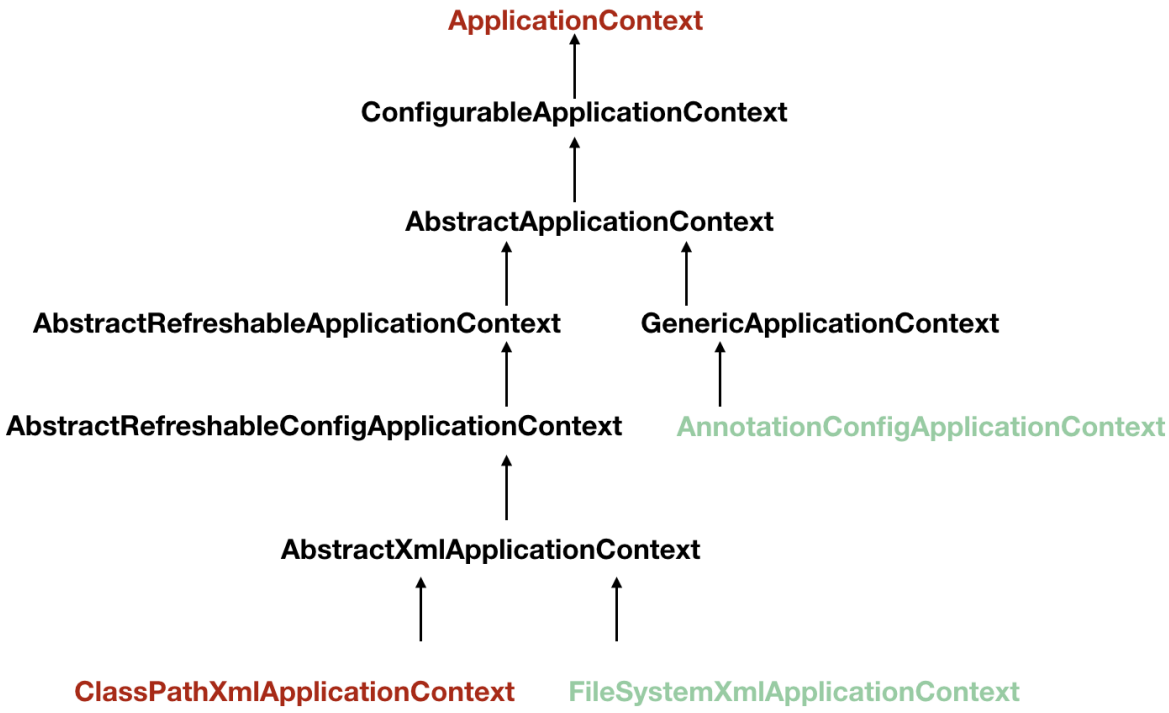
getBeanProvider：获取对象是哪个ObjectFactory产生的。

isTypeMatch：是否匹配类型

String FACTORY_BEAN_PREFIX = "&"：跟之前的id加“&”就能得到根对象是一样的。



by 应癫



Spring根据不同功能分配不同层级，使得Bean的操作不会全部堆积在BeanFactory里面，这样需要使用哪些功能就可以实现相应接口即可，设计十分优雅。

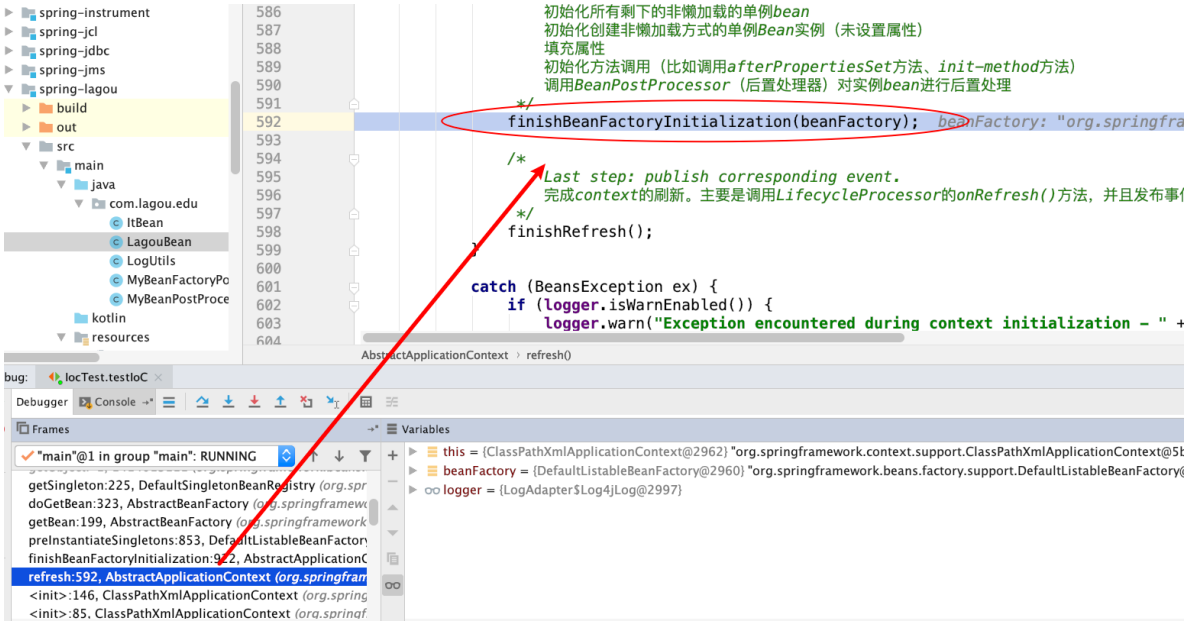
1. ApplicationContext 继承了 ListableBeanFactory，这个 Listable 的意思就是，**通过这个接口，我们可以获取多个 Bean，最顶层 BeanFactory 接口的方法都是获取单个 Bean 的。**
2. ApplicationContext 继承了 HierarchicalBeanFactory，Hierarchical 单词本身已经能说明问题了，也就是说**我们可以在应用中起多个 BeanFactory，然后可以将各个 BeanFactory 设置为父子关系。**
3. AutowireCapableBeanFactory 这个名字中的 Autowire 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。

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

Bean实例化和初始化的区别：

实例化一般是由类创建的对象，在构造一个实例的时候需要在内存中开辟空间，即Student s = new Student();

初始化在实例化的基础上，并且对对象中的值进行赋一下初始值




```
// ApplicationContext是容器的高级接口，BeanFacotry（顶级容器/根容器，规范了/定义了容器的基础行为）
// Spring应用上下文，官方称之为 IoC容器（错误的认识：容器就是map而已；准确来说，map是ioc容器的一个成员，
// 叫做单例池，singletonObjects,容器是一组组件和过程的集合，包括BeanFactory、单例池、BeanPostProcessor等以及之间的协作流程）
/**
  * Ioc容器创建管理Bean对象的，Spring Bean是有生命周期的
  * 构造器执行、初始化方法执行、Bean后置处理器的before/after方法、：AbstractApplicationContext#refresh#finishBeanFactoryInitialization
  * Bean工厂后置处理器初始化、方法执行：AbstractApplicationContext#refresh#invokeBeanFactoryPostProcessors
  * Bean后置处理器初始化：AbstractApplicationContext#refresh#registerBeanPostProcessors
  */
        ApplicationContext applicationContext = new ClassPathXmlApplicationContext("classpath:applicationContext.xml");
        LagouBean lagouBean = applicationContext.getBean(LagouBean.class);
        System.out.println(lagouBean);
```

Refresh方法讲解：（完成Spring容器的初始化）

refresh方法内部方法列表：有加锁

```
public abstract class AbstractApplicationContext{
@Override
    public void refresh() throws BeansException, IllegalStateException {
        // 对象锁加锁
        synchronized (this.startupShutdownMonitor) {
            /*
                 Prepare this context for refreshing.
                 刷新前的预处理
                 表示在真正做refresh操作之前需要准备做的事情：
                 设置Spring容器的启动时间，
                 开启活跃状态，撤销关闭状态
                 验证环境信息里一些必须存在的属性等
```

```
13              */
14             prepareRefresh();
15 
16         // 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，
17       // 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，
18       //
19 
20             ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();//重要步骤，下面有讲解。
21 
22             /*
23                 BeanFactory的预准备工作（BeanFactory进行一些设置，比如context的类加载
器等）
24              */
25             prepareBeanFactory(beanFactory);
26 
27             try {
28                 /*
29                     BeanFactory准备工作完成后进行的后置处理工作
30                  */
31                 postProcessBeanFactory(beanFactory);
32 
33                 /*
34                     实例化实现了BeanFactoryPostProcessor接口的Bean，并调用接口方法
35                  */
36                 invokeBeanFactoryPostProcessors(beanFactory);
37 
38                 /*
39                     注册BeanPostProcessor（Bean的后置处理器），在创建bean的前后等执行
40                  */
41                 registerBeanPostProcessors(beanFactory);
42 
43                 /*
44                     初始化MessageSource组件（做国际化功能；消息绑定，消息解析）；
45                  */
46                 initMessageSource();
47 
48                 /*
49                     初始化事件派发器
50                  */
51                 initApplicationEventMulticaster();
52 
53                 /*
54                     子类重写这个方法，在容器刷新的时候可以自定义逻辑；如创建Tomcat，
Jetty等WEB服务器
55                  */
56                 onRefresh();
57 
58                 /*
59                     注册应用的监听器。就是注册实现了ApplicationListener接口的监听器
bean
60                  */
61                 registerListeners();
62 
63                 /*
64                     初始化所有剩下的非懒加载的单例bean
65                     初始化创建非懒加载方式的单例Bean实例（未设置属性）
66                     填充属性
67                     初始化方法调用（比如调用afterPropertiesSet方法、init-method方法）
68                     调用BeanPostProcessor（后置处理器）对实例bean进行后置处理
```

```
                 */
                  //重要步骤
                finishBeanFactoryInitialization(beanFactory);

                /*
                     完成context的刷新。主要是调用LifecycleProcessor的onRefresh()方法，并且发布事件（ContextRefreshedEvent）
                 */
                finishRefresh();
            }
            catch (BeansException ex) {
                if (logger.isWarnEnabled()) {
                    logger.warn("Exception encountered during context initialization - " +"cancelling refresh attempt: " + ex);
                }
                destroyBeans();
                cancelRefresh(ex);
                throw ex;
            }
            finally {
                resetCommonCaches();
            }
        }
    }
}
```

refresh（）两个重要方法：

ObtainFreshBeanFactory()：获取BeanFactory、将解析放入BeanDefinition，放入BeanDefinition方法在BeanFactory中

获取BeanFactory细节：

判断是否已有BeanFactory

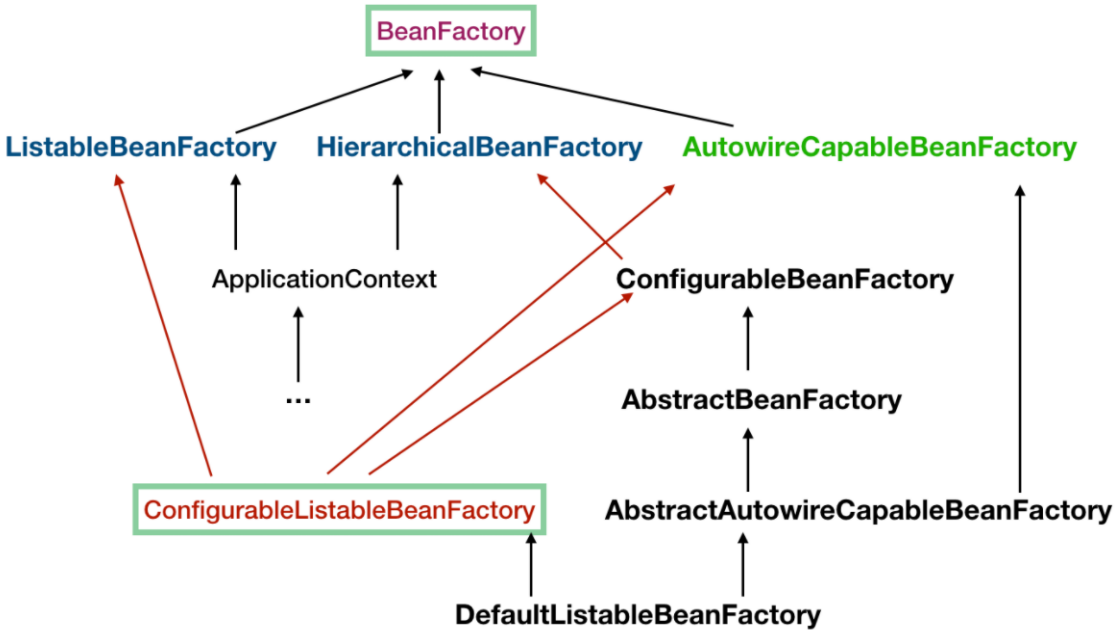
- 如果有BeanFactory
  1）如果有先销毁 beans。
  2）再关闭 BeanFactory。
- 如果没有BeanFactory
  1）实例化BeanFactory（默认是DefaultListableBeanFactory）
  2）BeanFactory设置序列化id
  3）自定义BeanFactory的一些属性（是否覆盖（例：多个xml的id重复是否覆盖）、是否允许循环依赖）。

```
//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，我想大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);
```

```
15
16        // 加载 Bean 到 BeanFactory 中
17        loadBeanDefinitions(beanFactory);
18        synchronized (this.beanFactoryMonitor) {
19           this.beanFactory = beanFactory;
20        }
21     }
```

```
1  protected void customizeBeanFactory(DefaultListableBeanFactory beanFactory) {
2     if (this.allowBeanDefinitionOverriding != null) {
3        // 是否允许 Bean 定义覆盖
4   beanFactory.setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverridin
    g);
5     }
6     if (this.allowCircularReferences != null) {
7        // 是否允许 Bean 间的循环依赖
8        beanFactory.setAllowCircularReferences(this.allowCircularReferences);
9     }
10 }
```



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。

loadBeanDefinitions()（在ObtainFreshBeanFactory()内部）：

加载应用中的BeanDefinitions：

（1）XML文件的解析：使用XmlBeanDefinitionReader读取XML信息。（解析标签成AbstractBeanDefinition）

解析过程：

下诉过程类似于以下代码：

```
InputStream resourceAsStream =
BeanFactory.class.getClassLoader().getResourceAsStream("beans.xml");
 SAXReader saxReader = new SAXReader();
 Document document = saxReader.read(resourceAsStream);
 Element rootElement = document.getRootElement();
 List<Element> list = rootElement.selectNodes("//bean");
```

1) 找到定义Javabean信息的XML文件，并将其封装成Resource对象，遍历Rescource数组，使用XmlBeanDefinitionReader 遍历解析成Element

2) 解析的标签成AbstractBeanDefinition （BeanDefinitions接口的实现类）

```
//根据标签进行解析
//parseDefaultElement(ele, delegate) 解析节点 <import />、<alias />、<bean />、<beans />
//delegate.parseCustomElement(element) 这个分支。如我们经常会使用到的 <mvc />、<task />、<context />、<aop />等。

   // 解析<bean/> 标签方法讲解，解析返回AbstractBeanDefinition对象,该对象实现BeanDefinition接口
processBeanDefinition(ele, delegate);

   //解析<bean/>标签成AbstractBeanDefinition
   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;
      }
```

（2）将BeanDefinition封装成BeanDefinitionHolder。

BeanDefinitionHolder内部结构：

```
 private final BeanDefinition beanDefinition;
 private final String beanName;
 private final String[] aliases;//bean的别名
```

（3）注册BeanDefinition到 IoC 容器：BeanDefinition 对象之后放入一个Map中，BeanFactory 是以 Map 的结构组织这些 BeanDefinition的。（最终BeanDefinition放入map是在DefaultListableBeanFactory方法里完成的）

```
public DefaultListableBeanFactory {

  public void registerBeanDefinition(String beanName, BeanDefinition
beanDefinition)
    private final Map<String, BeanDefinition> beanDefinitionMap = new
ConcurrentHashMap<>   (256);
    // 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);
    }
}
```

finishBeanFactoryInitialization（）- Bean对象创建流程：

1. 存放所有的beanNames到List
2. 通过BeanNames遍历，初始化所有单例bean（每个BeanNames对应一个Bean）

(1) 合并父子关系的BeanDefinition。

(2) 如果不是抽象&单例&不是延迟加载，往下走。

(3) 实例化bean。

实例化bean步骤:

1) 获取真正的BeanName。

2) 尝试从缓存中获取bean（没有则创建）。

```
// 尝试从缓存中获取 bean（没有自己创建）
Object sharedInstance = getSingleton(beanName);
```

```
//如果该bean是prototype&开启循环依赖，报异常。
// 如果是prototype类型且开启允许循环依赖，则抛出异常
if (isPrototypeCurrentlyInCreation(beanName)) {
    throw new BeanCurrentlyInCreationException(beanName);
}
```

3) 合并父子bean属性。

```
// 合并父子bean 属性
   final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
   checkMergedBeanDefinition(mbd, beanName, args);
```

4) 从单例池获取bean。

```
// 创建单例bean
if (mbd.isSingleton()) {
     sharedInstance = getSingleton(beanName, () -> {//单例池获取bean
     // 单例池为空时，创建 bean
     return createBean(beanName, mbd, args);
}
```

5) 如果单例池拿到的为空（不为空直接返回，没有下面步骤）

1.判断是否正在销毁状态，是的话抛异常。

2.不是销毁的话就创建bean，先标识bean正在被创建（因为bean创建过程步骤多，需标识来知道状态信息）。

```
   public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory)
{
        synchronized (this.singletonObjects) {
               //单例池获取bean
            Object singletonObject = this.singletonObjects.get(beanName);
            if (singletonObject == null) {
                // 是否正在销毁，异常
                if (this.singletonsCurrentlyInDestruction) {
                    throw new BeanCreationNotAllowedException();
                    }
                }
                // 验证完要真正开始创建对象，先标识该bean正在被创建，因为spingbean创建过程复杂，步骤很多，需要标识
                beforeSingletonCreation(beanName);
   }
```

3.AbstractAutowireCapableBeanFactory#createBeanInstance：通过反射实例化bean（就是生命周期图中的第一步，还未设置属性）。

4.AbstractAutowireCapableBeanFactory#populateBean：对实例化bean进行属性设置。

5.遍历bean的后置处理器进行调用，如下图。

```
@Override
public Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, String beanName)
        throws BeansException {

    Object result = existingBean;
    for (BeanPostProcessor processor : getBeanPostProcessors()) {
        Object current = processor.postProcessBeforeInitialization(result, beanName);
        if (current == null) {
            return result;
        }
        result = current;
    }
    return result;
}

@Override
public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)
        throws BeansException {

    Object result = existingBean;
    for (BeanPostProcessor processor : getBeanPostProcessors()) {
        Object current = processor.postProcessAfterInitialization(result, beanName);
        if (current == null) {
            return result;
        }
        result = current;
    }
    return result;
}
```

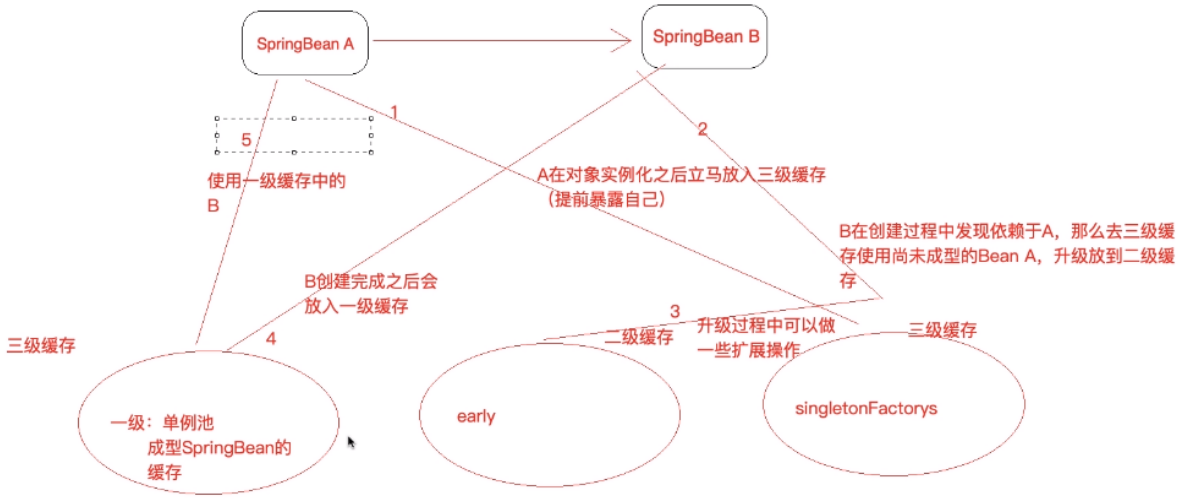
延迟加载源码

之前：如果不是抽象&单例&不是延迟加载，往下走。

当用到bean时（比如调用getBean（））

会进入doGetBean（），后面流程参考finishBeanFactoryInitialization（）2-（3）实例化bean。

循环依赖



1.SpringBean A 实例化放进三级缓存

2.SpringBean B创建过程中发现依赖于A，去三级缓存使用未成型的SpringBean A

3.将SpringBean A升级到二级缓存（升级过程可以做扩展操作），然后就可以使用

4.SpringBean B就可以创建，放入一级缓存（只可以放成型bean）

5.SpringBean A可以在一级缓存中直接拿SpringBean B（已成型）

等于说二、三级缓存作用是暂时放一个不成型的bean（因为不成型bean不能放到一级缓存），一级缓存才是正常拿到的bean。

之所以分二三级缓存是因为升级过程可以做一些扩展操作

spring源码博文：https://blog.csdn.net/nuomizhende45/article/details/81158383