# Spring 容器的初始化和创建过程

- **以注解驱动为例**
  - 通过 AnnotationConfigApplicationContext 的构造函数来创建 ApplicationContext
    - ApplicationContext applicationContext = new AnnotationConfigApplicationContext(ExtConfig.class)
  - 查看 AnnotationConfigApplicationContext 构造函数的源码：

```
public AnnotationConfigApplicationContext(Class<?>... annotatedClasses) {
   this();
   register(annotatedClasses);
   refresh();
}
```

  - 其中最为重要的是，在构造函数中调用了 refresh() 方法：

```
@Override
public void refresh() throws BeansException, IllegalStateException {
   synchronized (this.startupShutdownMonitor) {
      // Prepare this context for refreshing.
      prepareRefresh();

      // Tell the subclass to refresh the internal bean factory.
      ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

      // Prepare the bean factory for use in this context.
      prepareBeanFactory(beanFactory);

      try {
         // Allows post-processing of the bean factory in context subclasses.
         postProcessBeanFactory(beanFactory);

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

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

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

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

```

```
            // Initialize other special beans in specific context subclasses.
            onRefresh();

            // Check for listener beans and register them.
            registerListeners();

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

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

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

            // Destroy already created singletons to avoid dangling resources.
            destroyBeans();

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

            // Propagate exception to caller.
            throw ex;
        }

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

- # Spring IOC 容器的 refresh
  - 即：容器的创建和刷新，refresh方法调用结束后整个容器就都创建并初始化完毕了
- ## 1. prepareRefresh()
  - Prepare this context for refreshing. 刷新前的预处理工作
    - initPropertySources()： 初始化一些属性设置，允许子类(子容器)自定义个性化的属性设置方法
    - getEnvironment().validateRequiredProperties()：对上一步设置的属性进行校验；上一步没有设置属性就直接跳过了
    - earlyApplicationEvents= new LinkedHashSet：保存容器中的一些早期的事件；一旦派发器(multicaster)可以使用了，就通过派发器将这些早期的事件发布(published)
- ## 2. obtainFreshBeanFactory()
  - Tell the subclass to refresh the internal bean factory. 获得新的BeanFactory
    - refreshBeanFactory()：刷新(创建)BeanFactory
      - 在GenericApplicationContext类的构造函数中：this.beanFactory = newDefaultListableBeanFactory();
      - 并为this.beanFactory设置一个ID：this.beanFactory.setSerializationId(getId());

- getBeanFactory()：返回在GenericApplicationContext类创建的beanFactory
- 将创建的BeanFactory返回，其类型为：DefaultListableBeanFactory

- **3. prepareBeanFactory(beanFactory)**
  - Prepare the bean factory for use in this context. BeanFactory的预准备工作，对BeanFactory进行一些设置
    - 设置BeanFactory的类加载器、支持表达式解析器、xxx
    - 添加部分BeanPostProcessor --- [ApplicationContextAwareProcessor]
    - 设置忽略的自动装配的接口EnvironmentAware、EmbeddedValueResolverAware、xxx
    - 注册可以解析的自动装配，能直接在任何组件中自动注入： BeanFactory、ResourceLoader、ApplicationEventPublisher、ApplicationContext
    - 添加BeanPostProcessor --- [ApplicationListenerDetector]
    - 如果可以找到的话，添加编译时的AspectJ支持 (AspectJ 用于支持AOP)
    - 向BeanFactory中注册一些默认的组件：
      - environment [ConfigurableEnvironment]
      - systemProperties [Map<String, Object>]
      - systemEnvironment[Map<String, Object>]
- **4. postProcessBeanFactory(beanFactory)**
  - Allows post-processing of the bean factory in context subclasses. BeanFactory准备工作完成后，进行的后置处理工作
    - 子类通过重写这个方法来在BeanFactory创建并预准备完成后做进一步的设置
    - 如果没有子类重写这个方法，就直接跳过
- **--- 以上是BeanFactory的创建及预准备工作 ---**
- **5. invokeBeanFactoryPostProcessors(beanFactory)**
  - Invoke factory processors registered as beans in the context. 执行BeanFactoryPostProcessors
    - BeanFactoryPostProcessors：BeanFactory的后置处理器，在BeanFactory标准初始化之后执行的，包含了2个接口：
      - BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor(它是BeanFactoryPostProcessor的一个子接口，但是在调用执行过程中，它比BeanFactoryPostProcessor更早执行)
        - 注意：这些接口或接口的子类根据优先级可以分为三大类： Separate between BeanFactoryPostProcessor that implement PriorityOrdered , Ordered , andthe rest .
          - 实现了PriorityOrdered优先级接口
          - 实现了Ordered顺序接口
          - 没有实现任何优先级或顺序的接口
    - 开始调用invokeBeanFactoryPostProcessors()方法：
      - 先执行BeanDefinitionRegistryPostProcessor的方法：
        - 获取所有的BeanDefinitionRegistryPostProcessor
        - 先执行实现了PriorityOrdered优先级接口的BeanDefinitionRegistryPostProcessor
          - postProcessor.postProcessBeanDefinitionRegistry(registry)

        - 再执行实现了Ordered顺序接口的BeanDefinitionRegistryPostProcessor
            - postProcessor.postProcessBeanDefinitionRegistry(registry)
        - 最后执行没有实现任何优先级或者是顺序接口的BeanDefinitionRegistryPostProcessors
            - postProcessor.postProcessBeanDefinitionRegistry(registry)
    - 再执行BeanFactoryPostProcessor的方法：
        - 获取所有的BeanFactoryPostProcessor
        - 获取所有的BeanFactoryPostProcessor
            - postProcessor.postProcessBeanFactory(beanFactory)
        - 再执行实现了Ordered顺序接口的BeanFactoryPostProcessor
            - postProcessor.postProcessBeanFactory(beanFactory)
        - 最后执行没有实现任何优先级或者是顺序接口的BeanFactoryPostProcessor
            - postProcessor.postProcessBeanFactory(beanFactory)

- **6. registerBeanPostProcessors(beanFactory)**
    - registerBeanPostProcessors(beanFactory) 注册BeanPostProcessor(Bean的后置处理器)，这些后置处理器用于拦截bean的创建过程
    - BeanPostProcessor是一个顶级接口，它有很多的子接口，不同类型的BeanPostProcessor有不同的功能，它们的执行时机是不一样的，有的是在Bean的实例化前后执行，有的是在Bean的初始化前后执行。包含了一下几个接口：
        - BeanPostProcessor
        - DestructionAwareBeanPostProcessor
        - InstantiationAwareBeanPostProcessor
        - SmartInstantiationAwareBeanPostProcessor
        - MergedBeanDefinitionPostProcessor 会被注册到 [internalPostProcessors]
        - 注意：这些接口或接口的子类根据优先级可以分为三大类： Separate between BeanFactoryPostProcessor that implement PriorityOrdered , Ordered , andthe rest .
    - 获取所有的BeanPostProcessor，后置处理器都默认可以通过PriorityOrdered、Ordered接口来指定优先级
    - 先注册实现了PriorityOrdered优先级接口的BeanPostProcessor
        - 把每一个BeanPostProcessor添加到BeanFactory中
        - beanFactory.addBeanPostProcessor(postProcessor);
    - 再注册实现了Ordered顺序接口的BeanPostProcessor
    - 然后注册没有实现任何优先级接口的BeanPostProcessor
    - 最终注册MergedBeanDefinitionPostProcessor [ internalPostProcessors ]
    - 最最后再注册一个ApplicationListenerDetector [ 监听器探测器 ]
        - 用来在bean创建完成后检查是不是ApplicationListener，如果是：
        - applicationContext.addApplicationListener((ApplicationListener<?>) bean);

- **7. initMessageSource()**

- Initialize message source for this context. 初始化MessageSource组件 (国际化功能；消息绑定，消息解析)
  - 获取BeanFactory
  - 检查容器中是否有id为messageSource，类型是MessageSource的组件
    - 如果有赋值给this.messageSource，如果没有Spring自动创建一个DelegatingMessageSource
    - MessageSource的作用：取出国际化配置文件中的某个key的值，能按照区域信息获取
  - 把创建好的MessageSource注册在容器中，以后获取国际化配置文件的值的时候，可以自动注入MessageSource
    - beanFactory.registerSingleton(MESSAGE_SOURCE_BEAN_NAME, this.messageSource)；
    - MessageSource.getMessage(String code, Object[] args, String defaultMessage, Locale locale)；

- **8. initApplicationEventMulticaster()**
  - Initialize event multicaster for this context. 初始化事件派发器 (基于事件驱动模型开发)
    - 获取BeanFactory
    - 从BeanFactory中获取id为applicationEventMulticaster的ApplicationEventMulticaster
    - 如果上一步没有配置，Spring自动创建一个SimpleApplicationEventMulticaster
    - 将创建好的ApplicationEventMulticaster添加到BeanFactory中，以后其他组件需要时就可以直接自动注入

- **9. onRefresh()**
  - Initialize other special beans in specific context subclasses. 留给子类(子容器)去做一些事情
    - 如果子类重写这个方法，在容器刷新的时候可以自定义逻辑

- **10. registerListeners()**
  - Check for listener beans and register them. 向容器中注册所有项目里面的ApplicationListener
    - 从容器中拿到所有的ApplicationListener
    - 将每个监听器添加到事件派发器中：getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);
    - 派发之前步骤产生的事件：getApplicationEventMulticaster().multicastEvent(earlyEvent);

- **11. finishBeanFactoryInitialization(beanFactory)**
  - Instantiate all remaining (non-lazy-init) singletons. 初始化所有剩下的单实例bean
  - beanFactory.preInstantiateSingletons();
    - 1. 获取容器中的所有Bean，依次进行初始化和创建对象
    - 2. 获取Bean的定义信息：RootBeanDefinition
    - 3. 如果Bean不是抽象的，是单实例的，不是懒加载：

- 判断是不是FactoryBean：是不是实现FactoryBean接口的Bean；
- 如果不是工厂Bean，利用getBean(beanName)创建对象
  - 1) getBean(beanName)，IOC容器中获取bean：ioc.getBean()也是走的这个流程
  - 2) doGetBean(name, null, null, false);
  - 3) 先获取缓存中保存的单实例Bean，如果能获取到，说明这个bean之前已经被创建过(所有创建过的单实例bean都会缓存起来)
    - 从 private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>(256) 这个集合中获取
  - 4) 缓存中获取不到，开始Bean的创建对象流程
  - 5) 先标记当前bean已经被创建(小细节，为了防止多线程并发导致单实例bean被多次创建)
  - 6) 获取Bean的定义信息
  - 7) 获取当前Bean依赖的其他Bean；如果有依赖于其他的bean，按照getBean()的方式把依赖的Bean先创建出来
    - 这点保证了，this Bean 的依赖不会出现问题。这种先将依赖Bean创建的过程，就是递归调用的机制
  - 8) 在进行了上面的一切保障措施，启动单实例Bean的创建流程：
    - createBean(beanName, mbd, args)
    - Object bean = resolveBeforeInstantiation(beanName, mbdToUse)
      - 尝试让BeanPostProcessor先拦截返回代理对象，[ InstantiationAwareBeanPostProcessor ]类型的后置处理器提前执行：
        - 先触发：postProcessBeforeInstantiation()
        - 如果上面的触发有返回值，再触发：postProcessAfterInitialization()
    - 如果前面的 InstantiationAwareBeanPostProcessor 没有返回代理对象，接着往下进行
    - **Object beanInstance = doCreateBean(beanName, mbdToUse, args); 真正的创建bean**
      - a. 创建bean实例：createBeanInstance(beanName, mbd, args)
        - 利用工厂方法或者对象的构造器创建出Bean实例；
      - b. applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName)
        - 调用MergedBeanDefinitionPostProcessor的postProcessMergedBeanDefinition(mbd, beanType, beanName)；
      - c. Bean属性赋值：populateBean(beanName, mbd, instanceWrapper)赋值之前：
        - 获取InstantiationAwareBeanPostProcessor类型的后置处理器；
          - 执行postProcessAfterInstantiation()方法

- 获取InstantiationAwareBeanPostProcessor类型的后置处理器；
  - 执行postProcessPropertyValues()方法
- 开始赋值：
  - 应用Bean属性的值；为属性利用setter方法等进行赋值；
  - applyPropertyValues(beanName, mbd, bw, pvs)

- d. Bean初始化：initializeBean(beanName, exposedObject, mbd)
  - 执行Aware接口的方法：invokeAwareMethods(beanName, bean)，执行xxxAware接口的方法
    - 如果是BeanNameAware\BeanClassLoaderAware\BeanFactoryAware接口，就执行这些接口里面的方法
  - 执行后置处理器初始化之前：
    - applyBeanPostProcessorsBeforeInitialization
    - BeanPostProcessor.postProcessBeforeInitialization()
  - 执行初始化方法：invokeInitMethods(beanName, wrappedBean,mbd)
    - 如果是InitializingBean接口的实现，执行接口规定的初始化；
    - 如果自定义了初始化方法，执行自定义的初始化
  - 执行后置处理器初始化之后的方法：
    - applyBeanPostProcessorsAfterInitialization
    - BeanPostProcessor.postProcessAfterInitialization()
- e. 注册Bean的销毁方法：egisterDisposableBeanIfNecessary(beanName,bean, mbd)

- 将创建的Bean添加到缓存中 [ singletonObjects ]

- 4. 所有Bean都利用getBean创建完成以后：
  - 检查所有的Bean是否是SmartInitializingSingleton接口
  - 如果是，就执行afterSingletonsInstantiated()

- **--- Episode ---**
  - 什么是Spring容器？
    - 本质上就是一些Map对象，这些Map中保存很多信息；有的保存了单实例bean，有的保存了环境信息，有的保存了工厂信息，等等。

- **12. finishRefresh()**
  - Last step: publish corresponding event. 完成BeanFactory的初始化创建工作，IOC容器就创建完成。

- clearResourceCaches()：清理所有的资源缓存
- initLifecycleProcessor()：初始化和生命周期有关的后置处理器（LifecycleProcessor）
  - 默认从容器中找id为lifecycleProcessor的组件，类型为LifecycleProcessor
  - 如果没有，Spring自动创建一个：new DefaultLifecycleProcessor()，并加载到容器中
  - 允许我们自定义一个LifecycleProcessor的实现类，可以在IOC容器的刷新和关闭阶段做一些事情：
    - void onRefresh()
    - void onClose()
  - getLifecycleProcessor().onRefresh()
    - 获取上一步定义的生命周期处理器 (BeanFactory)，回调onRefresh()；
  - publishEvent(new ContextRefreshedEvent(this))
    - 发布容器刷新完成事件
  - liveBeansView.registerApplicationContext(this)

- **--- cut-off rule ---**
- **总结**
  - **1. Spring容器在启动的时候，先会保存所有注册进来的Bean的定义信息**
    - xml注册bean；
    - 注解注册Bean：@Service、@Component、@Bean、等等
  - **2. Spring容器会合适的时机创建这些Bean**
    - 启动过程中如果利用到某个bean的时候，利用getBean创建bean，创建好以后保存在容器中
    - 统一创建剩下所有的bean的时候：finishBeanFactoryInitialization()
  - **3. 后置处理器：BeanPostProcessor**
    - 不同的后置处理器完成不同的功能
    - 每一个bean创建完成，会使用各种后置处理器进行处理，来增强bean的功能，如：
      - AutowiredAnnotationBeanPostProcessor：处理自动注入
      - AnnotationAwareAspectJAutoProxyCreator：完成AOP功能
      - AsyncAnnotationBeanPostProcessor：异步注解支持
      - ……
  - **4. 事件驱动模型**
    - ApplicationListener：事件监听器
    - ApplicationEventMulticaster：事件派发器