BeanFactoryPostProcessor

BeanFactoryPostProcessor 是实现spring容器功能扩展的重要接口，例如修改bean属性值，实现bean动态代理等。很多框架都是通过此接口实现对spring容器的扩展，例如mybatis

```
public class MyBeanFactoryPostProcessor  implements  BeanFactoryPostProcessor  {

    @Override
    public  void  postProcessBeanFactory (ConfigurableListableBeanFactory   beanFactory)  throws
BeansException  {
        System.out.println("调用MyBeanFactoryPostProcessor 的postProcessBeanFactory" );
        BeanDefinition  bd  =  beanFactory.getBeanDefinition( "myJavaBean" );
        System.out.println("属性值============" + bd.getPropertyValues().toString()) ;
        MutablePropertyValues  pv  =  bd.getPropertyValues() ;
        if (pv.contains( "remark" )) {
            pv.addPropertyValue( "remark",  "把备注信息修改一下");
        }
        bd.setScope(BeanDefinition. SCOPE_PROTOTYPE );
    }
}
```

or

```
@Override
public  void  postProcessBeanFactory (ConfigurableListableBeanFactory   beanFactory)  throws
BeansException  {

    BeanDefinitionRegistry  bdr  =  (BeanDefinitionRegistry)beanFactory ;
    GenericBeanDefinition  gbd  =  new  GenericBeanDefinition() ;
    gbd.setBeanClass(EngineFactory. class);
    gbd.setScope(BeanDefinition. SCOPE_SINGLETON );
    gbd.setAutowireCandidate( true);
    bdr.registerBeanDefinition( "engine01-gbd" ,  gbd);
}
```

BeanDefinitionRegistryPostProcessor（触发时机：bean定义注册之前）

BeanDefinitionRegistryPostProcessor的实现类一共要实现以下两个方法：

void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException：该方法的实现中，主要用来对bean定义做一些改变。

void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException：

```
public  class  MyBeanDefinitionRegistryPostProcessor
implements  BeanDefinitionRegistryPostProcessor   {

    @Override
    public  void  postProcessBeanDefinitionRegistry
(BeanDefinitionRegistry  beanDefinitionRegistry)  throws  BeansException  {
        // 创建一个bean的定义类的对象
        RootBeanDefinition  rootBeanDefinition  =  new  RootBeanDefinition(MyService. class);
        // 将Bean 的定义注册到Spring环境
        beanDefinitionRegistry.registerBeanDefinition(  "testService" ,  rootBeanDefinition) ;
    }

    @Override
    public  void  postProcessBeanFactory
(ConfigurableListableBeanFactory  configurableListableBeanFactory)  throws  BeansException  {
        // bean的名字为key, bean的实例为value
                    Map<String,  Object>  beanMap  =
configurableListableBeanFactory.getBeansWithAnnotation(  MyComponent .class);
    }
}
```

BeanDefinitionRegistry

BeanDefinition注册中心

它的默认实现类，主要有三个：SimpleBeanDefinitionRegistry、DefaultListableBeanFactory、GenericApplicationContext

```
// 它继承自 AliasRegistry
public interface BeanDefinitionRegistry  extends AliasRegistry {

    // 关键 -> 往注册表中注册一个新的 BeanDefinition 实例
     void registerBeanDefinition (String  beanName , BeanDefinition  beanDefinition)  throws
BeanDefinitionStoreException ;
    // 移除注册表中已注册的 BeanDefinition 实例
    void removeBeanDefinition(String beanName) throws NoSuchBeanDefinitionException ;
    // 从注册中心取得指定的 BeanDefinition 实例
    BeanDefinition getBeanDefinition(String beanName) throws NoSuchBeanDefinitionException ;
    // 判断 BeanDefinition 实例是否在注册表中（是否注册）
    boolean containsBeanDefinition(String beanName);

    // 取得注册表中所有 BeanDefinition 实例的 beanName（标识）
    String[] getBeanDefinitionNames();
    // 返回注册表中 BeanDefinition 实例的数量
    int getBeanDefinitionCount();
    // beanName（标识）是否被占用
    boolean isBeanNameInUse(String beanName);
}
```

BeanNameAware

```
@Component
public class MyBeanNameAware implements BeanNameAware {

    public String name;

    @Override
    public void setBeanName(String name) {
        System.out.println("bean name is: " + name);
        this.name = name;
    }
}
```

获取当前bean名称

BeanFactoryAware

获取当前bean所在的beanFactory

```
public class MyBeanFactoryAware implements BeanFactoryAware {

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {

    }
}
```

ApplicationContextAware

Bean就获得了自己所在的ApplicationContext

```
public class MyApplicationContextAware implements ApplicationContextAware {

    @Override
     public  void  setApplicationContext (ApplicationContext  applicationContext)  throws
BeansException {
    }
}
```

BeanFactory:

是Spring里面最低层的接口，提供了最简单的容器的功能，只提供了实例化对象和拿对象的功能；
ApplicationContext:
应用上下文，继承BeanFactory接口，它是Spring的一各更高级的容器，提供了更多的有用的功能；
1) 国际化（MessageSource）
2) 访问资源，如URL和文件（ResourceLoader）
3) 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层
4) 消息发送、响应机制（ApplicationEventPublisher）
5) AOP（拦截器）

BeanPostProcessor

```
public class MyBeanPostProcessor implements BeanPostProcessor {

    @Override
     public Object postProcessBeforeInitialization (Object bean, String beanName) throws
BeansException {
        return null;//返回null的话后继BeanPostProcessor 将不会被调用
    }

    @Override
     public Object postProcessAfterInitialization (Object bean, String beanName) throws
BeansException {
        return null;
    }
}
```

InitializingBean
bean提供了初始化方法的方式，它只包括afterPropertiesSet方法，凡是继承该接口的类，在初始化bean的时候都会执行该方法

```
@Component
public class MyInitializingBean implements InitializingBean {

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("afterPropertiesSet");
    }
}
```

init-method方法

```
@PostConstruct
public void postTest(){

}
```

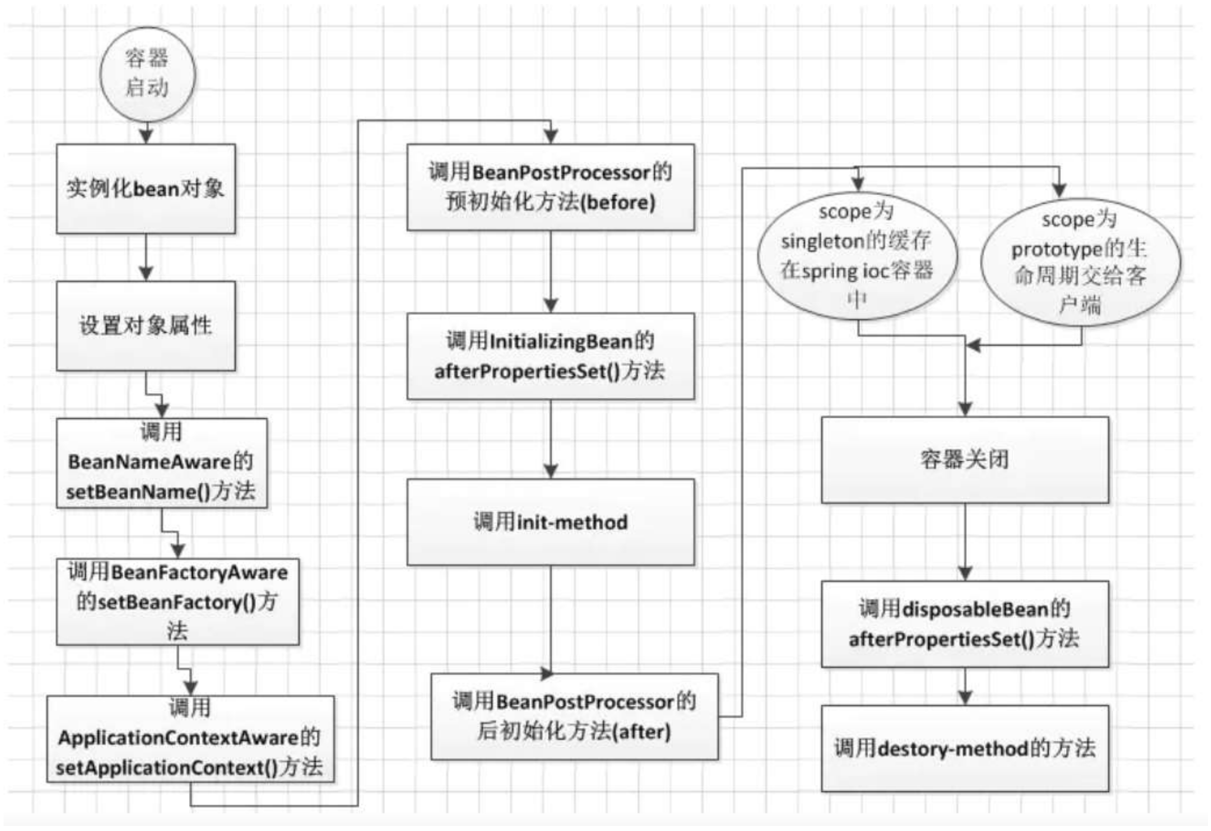
```
public @interface Bean {
        @AliasFor("name")
        String[] value() default {};

        @AliasFor("value")
        String[] name() default {};
        @Deprecated
        Autowire autowire() default Autowire.NO;

        boolean autowireCandidate() default true;

        String initMethod() default "";

        String destroyMethod() default AbstractBeanDefinition.INFER_METHOD;

}
```

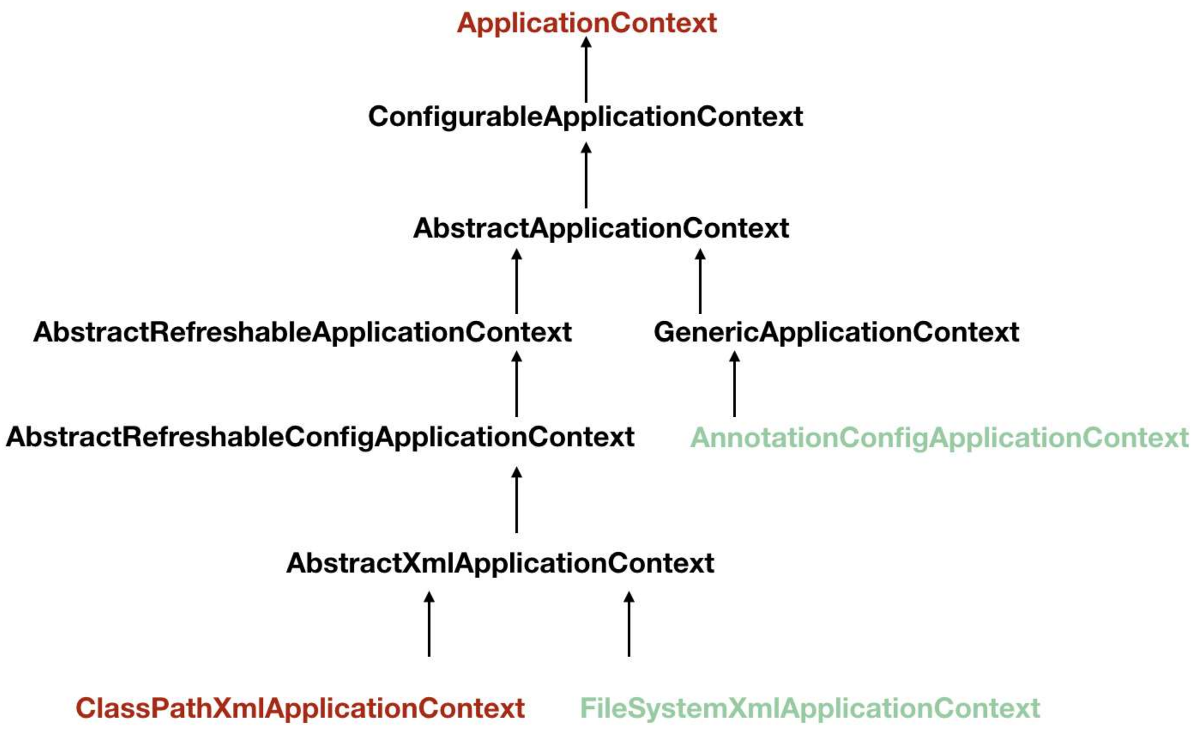
DisposableBean

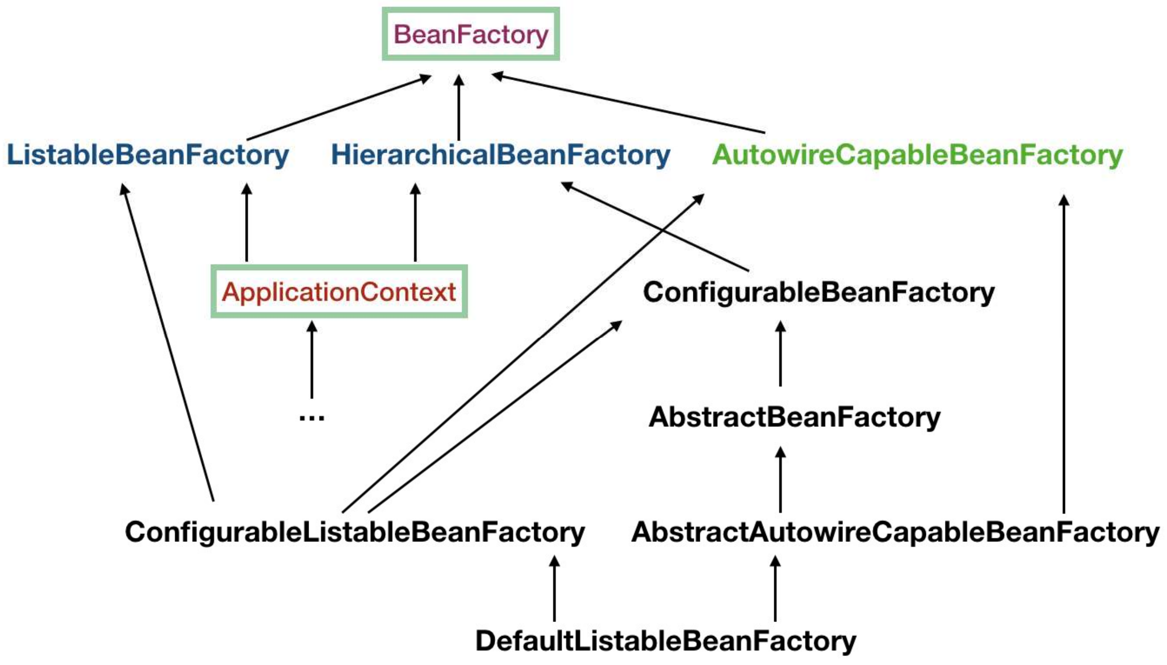
Destroy-method
@PreDestroy

spring中，有内置的一些BeanPostProcessor实现类，例如：
CommonAnnotationBeanPostProcessor：支持@Resource注解的注入
RequiredAnnotationBeanPostProcessor：支持@Required注解的注入
AutowiredAnnotationBeanPostProcessor：支持@Autowired注解的注入
PersistenceAnnotationBeanPostProcessor：支持@PersistenceUnit和@PersistenceContext注解的注入
ApplicationContextAwareProcessor：用来为bean注入ApplicationContext等容器对象

AutowireCapableBeanFactory
ConfigurableListableBeanFactory


容器启动
实例化bean对象
设置对象属性
调用BeanNameAware的setBeanName()方法
调用BeanFactoryAware的setBeanFactory()方法
调用ApplicationContextAware的setApplicationContext()方法
调用BeanPostProcessor的预初始化方法(before)
调用InitializingBean的afterPropertiesSet()方法
调用init-method
调用BeanPostProcessor的后初始化方法(after)
scope为singleton的缓存在spring ioc容器中
scope为prototype的生命周期交给客户端
容器关闭
调用disposableBean的afterPropertiesSet()方法
调用destory-method的方法


ApplicationContext
ConfigurableApplicationContext
AbstractApplicationContext
AbstractRefreshableApplicationContext
GenericApplicationContext
AbstractRefreshableConfigApplicationContext
AnnotationConfigApplicationContext
AbstractXmlApplicationContext
ClassPathXmlApplicationContext
FileSystemXmlApplicationContext


BeanFactory
ListableBeanFactory
HierarchicalBeanFactory
AutowireCapableBeanFactory
ApplicationContext
ConfigurableBeanFactory
...
AbstractBeanFactory
ConfigurableListableBeanFactory
AbstractAutowireCapableBeanFactory
DefaultListableBeanFactory

```
ClassPathResource resource = new ClassPathResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);
```