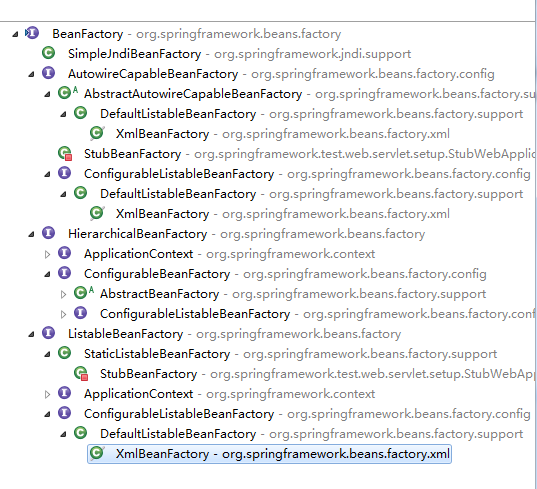
## Spring中的context

### Spring容器理解

spring容器可以理解为生产对象（OBJECT）的地方，在这里容器不只是帮我们创建了对象那么简单，它负责了对象的整个生命周期--创建、装配、销毁。而这里对象的创建管理的控制权都交给了Spring容器，所以这是一种控制权的反转，称为IOC容器，而这里IOC容器不只是Spring才有，很多框架也都有该技术。

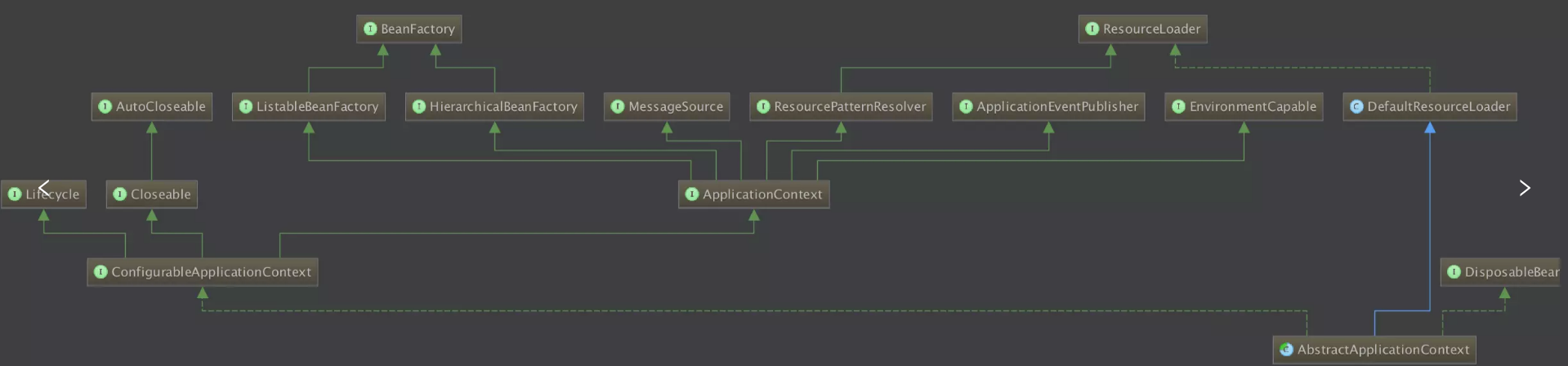


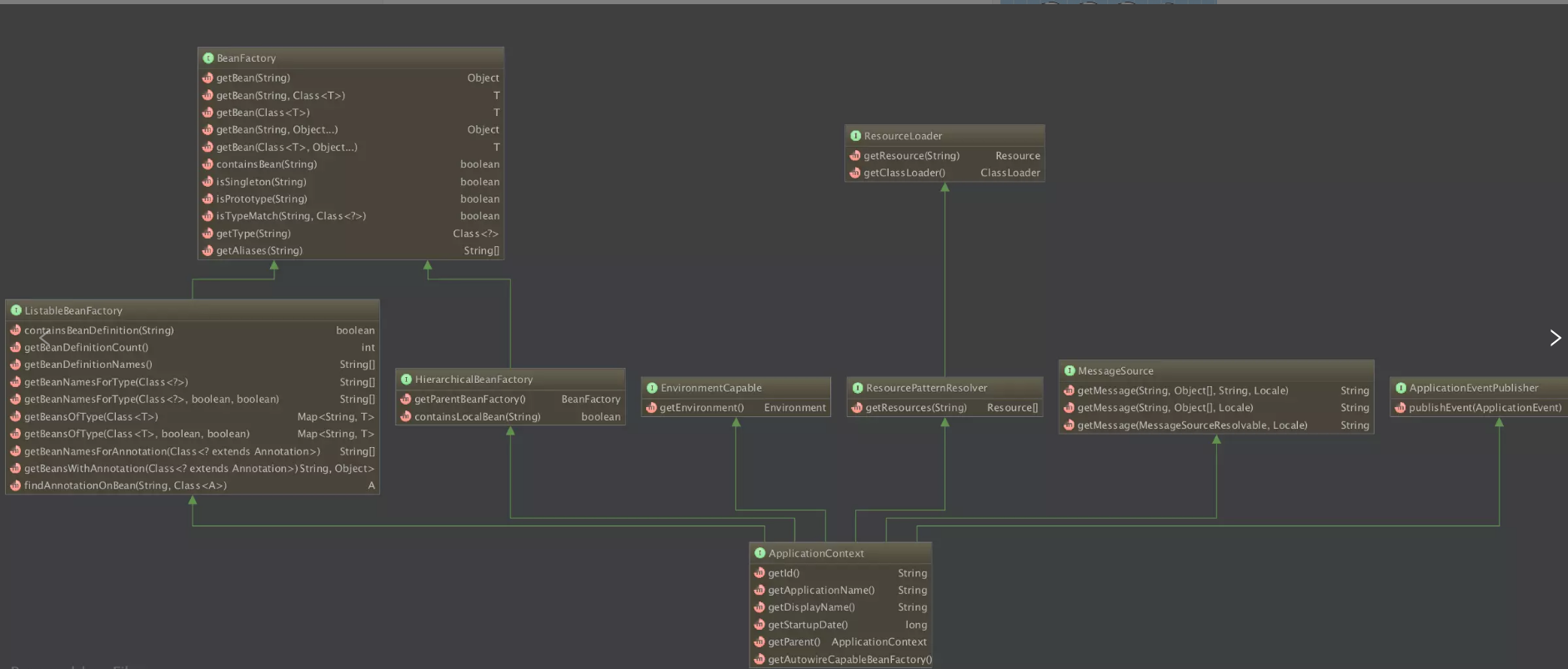
1. **BeanFactory接口：**  
    是Spring bean容器的根接口，提供获取bean，是否包含bean,是否单例与原型，获取bean类型，bean 别名的方法 。它最主要的方法就是getBean(String beanName)。  
   **2.BeanFactory的三个子接口：**  
    \* HierarchicalBeanFactory：提供父容器的访问功能  
    \* ListableBeanFactory：提供了批量获取Bean的方法  
    \* AutowireCapableBeanFactory：在BeanFactory基础上实现对已存在实例的管理  
   **3.ConfigurableBeanFactory：**  
   主要单例bean的注册，生成实例，以及统计单例bean  
   **4.ConfigurableListableBeanFactory：**  
   继承了上述的所有接口，增加了其他功能：比如类加载器,类型转化,属性编辑器,BeanPostProcessor,作用域,bean定义,处理bean依赖关系, bean如何销毁…  
   **5.实现类DefaultListableBeanFactory[详细介绍](https://www.cnblogs.com/sten/p/5758161.html" \t "_blank)：**  
   实现了ConfigurableListableBeanFactory，实现上述BeanFactory所有功能。它还可以注册BeanDefinition  
   接口详细介绍请参考:[揭秘BeanFactory](https://blog.csdn.net/u011179993/article/details/51636742)

**beanFactory启动**

new XmlBeanFactory(new ClassPathResource(“../news-config.xml”));

如果说BeanFactory是Sping的心脏，那么ApplicationContext就是完整的身躯了





**除了提供BeanFactory所支持的所有功能外ApplicationContext还有额外的功能**

* 默认初始化所有的Singleton，也可以通过配置取消预初始化。
* 继承MessageSource，因此支持国际化。
* 资源访问，比如访问URL和文件。
* 事件机制。
* 同时加载多个配置文件。
* 以声明式方式启动并创建Spring容器。

| **ApplicationContext常用实现类** | **作用** |
| --- | --- |
| AnnotationConfigApplicationContext | 从一个或多个基于java的配置类中加载上下文定义，适用于java注解的方式。 |
| ClassPathXmlApplicationContext | 从类路径下的一个或多个xml配置文件中加载上下文定义，适用于xml配置的方式。 |
| FileSystemXmlApplicationContext | 从文件系统下的一个或多个xml配置文件中加载上下文定义，也就是说系统盘符中加载xml配置文件。 |
| AnnotationConfigWebApplicationContext | 专门为web应用准备的，适用于注解方式。 |
| XmlWebApplicationContext | 从web应用下的一个或多个xml配置文件加载上下文定义，适用于xml配置方式。 |

BeanFactory是Spring容器依赖注入的基础。位于类结构树的顶端，接口中最重要的方法就是getBean(String)。得到特定名称的Bean对象。  
ListableBeanFactory是用来访问容器内bean的相关信息，根据方法名可以看出，就相当于对一个容器中的内容相关信息的获取。  
HierarchicalBeanFactory是容器层次化。获取父级BeanFactory。但是在ApplicationContext接口中没有setParent()方法，在ConfigurableApplicationContext接口中出现了set方法。这个在后面说到的时候再说。通过bean名称判断当前容器是否包含这个bean实例。  
ResourceLoader是资源加载接口，用于对不同的Resource进行加载。Resource getResource(String location)通过一个标识加载资源信息。  
ResourcePatternResolver是ResourceLoader的扩展，将一个标识拆分成多个资源。  
MessageSource接口，�提供了消息处理的功能。在web项目上用于国际化。  
EnvironmentCapable运行环境接口。通过这个接口可以获取到Environment。[spring的Environment](https://link.jianshu.com/?t=http://my.oschina.net/u/867830/blog/407191?fromerr=y5GpJr0O" \t "_blank)  
ApplicationEventPublisher提供了发布event组件的接口。[spring的事件体系](https://link.jianshu.com/?t=http://blog.csdn.net/xiejx618/article/details/43268307) 和[Spring内置事件处理](https://link.jianshu.com/?t=http://wiki.jikexueyuan.com/project/spring/event-handling-in-spring.html" \t "_blank)

**ApplicationContext中的主要实现类解读**

ConfigurableApplicationContext接口方法完成了对ApplicationContext进一步的扩展。ApplicationContext以及他继承的部分接口都是获取相关Properties。ConfigurableApplicationContext中提供了set方法。  
setId对应ApplicationContext中的get方法。  
setParent对应HierarchicalBeanFactory中的get方法。  
setEnvironment对应EnvironmentCapable中的get方法。  
addApplicationListener对应ApplicationEventPublisher。扩展之后可以发布事件，也有了监听能力。  
同时重新定义了getEnvironment()。  
isActive(),refresh()和close()方法的定义让实现这个接口的类具有了状态标识。  
addBeanFactoryPostProcessor为操作容器内的bean提供了接口。BeanFactoryPostProcessor接口，允许修改容器中的定义的bean。  
以上就是ConfigurableApplicationContext接口对ApplicationContext的扩展内容。

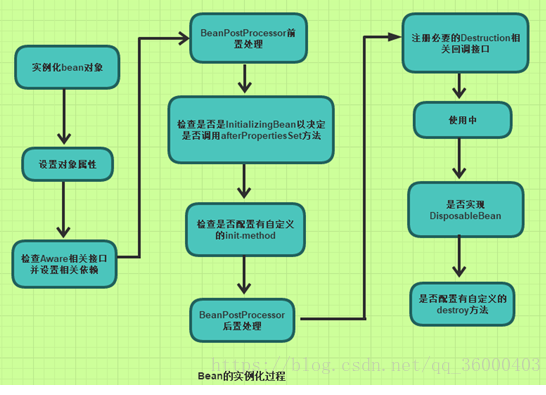
### 实例化过程

容器启动之后，并不会马上就实例化相应的bean定义，我们知道，容器现在仅仅拥有所有对象的BeanDefinition来保存实例化阶段将要用的必要信息。只有当请求通过BeanFactory的getBean()方法来请求某个对象实例的时候，才有可能触发Bean实例化阶段的活动。BeanFactory的getBean方法可以被客户端对象显示调用，也可以在容器内部隐式地被调用。隐式调用有如下两种情况

1.对于 Beanfactory来说，对象实例化默认采用延迟初始化。通常情况下，当对象A被请求而需要第一次实例化的时候，如果它所依赖的对象B之前同样没有被实例化，那么容器会先实例化对象A所依赖的对象。这时容器内部就会首先实例化对象B，以及对象A依赖的其他还没有被实例化的对象。这种情况是容器内部调用getBean（），对于本次请求的请求方是隐式的。

2.Applicationcontext启动之后会实例化所有的bean定义。但 ApplicationContext在实现的过程中依然循Sping容器实现流程的两个阶段，只不过它会在启动阶段的活动完成之后，紧接着调用注册到该容器的所有bean定义的实例化方法getBean()。这就是为什么当你得到 ApplicationContext类型的容器引用时，容器内所有对象已经被全部实例化完成。不信你查一下类org.springframework.context.support.AbstractApplicationContext的 refresh（）方法

之所以说 getBean（）方法是有可能触发Bean实例化阶段的活动，是因为只有当对应某个bean定义的 getBean（）方法第一次波调用时，不管是显式的还是隐式的，Bean实例化阶段的活动才会被触发，第二次被调用则会直接返回容器缓存的第一次实例化完的对象实例（ prototype类型bean除外）。当getBean()方法内部发现该bean定义之前还没有被实例化之后，会通过 createBean（）方法来进行具体的对象实例化，实例化过程如下图所示



### 生命周期

#### Bean的实例化与BeanWrapper

容器在内部实现的时候,采用策略模式决定如何初始化bean实例,通常,可以通过反射或者CGLIB动态字节码.

org.springframework.beans.factory.support.InstantiationStrategy定义是实例化策略的抽象接口，其直接子类SimpleInstantiationStrategy实现了简单的对象实例化功能，可以通过反射来实例化对象实例，但不支持方法注入方式的对象实例化。CglibSubclassingInstantiationStrategy继承了SimpleInstantiationStrategy的以反射方式实例化对象的功能，并且通过CGLIB的动态字节码生成功能，该策略实现类可以动态生成某个类的子类，进而满足了方法注入所需的对象实例化需求。默认情况下，容器内部采用的是CglibSubclassingInstantiationStrategy

容器只要根据相应bean定义的BeanDefinition取得实例化信息结合CglibSubclassingInstantiationStratery以及不同的bean定义类型，就可以返回实例化完成的对象实例。但是，返回方式上有些”点缀”.不是直接返回构造完成的对象实例，而是以BeanWrapper对构造完成的对象实例进行包裹，返回相应的BeanWrapper实例。

BeanWrapper接口通常在Spring框架内部使用，它有一个实现类org.springframework.beans.BeanWrapperImpl. 其作用是对某个bean进行”包裹”，然后对这个”包裹”的bean进行操作，比如设置或者获取bean的相应属性值。而在第一步结束后返回BeanWrapper实例而不是原先的对象实例，就是为了第二步”设置对象属性”。

BeanWrapper定义继承了org.springframework.beans.PropertyAccessor接口，可以以统一的方式对对象属性进行访问；BeanWrapper定义同时又直接或者间接继承了PropertyEditorRegistry和TypeConverter接口。不知你是否还记得CustomEditorConfigurer？当把各种PropertyEditor注册给容器时，BeanWrapper用到这些PropertyEditor，在第一步构造完成对象之后，Spring会根据对象实例构造一个BeanWrapperImpl实例，然后将之前CustomEditorConfigurer注册的PropertyEditor复制一份给BeanWrapperImpl实例(这就是BeanWrapper同时又是PropertyEditorRegistry的原因)，这样，当BeanWrapper转换类型，设置对象属性时，就不会无从下手了。

#### 各色的Aware接口

当对象实例化完成并且相关属性以及依赖设置完成之后，Spring容器会检查当前对象实例是否实现了一系列的以Aware命名结尾的接口定义，如果是，则将这些Aware接口定义中规定的依赖注入给当前对象实

#### BeanPostProcessor

BeanpostProcessor的概念容易与 BeanfactoryPostProcessor的概念混淆。但只要记住BeanPostProcessor是存在于对象实例化阶段，而BeanFactoryPostProcessor则是存在于容器启动阶段，这两个概念就比较容易区分了。

与BeanFactoryPostProcessor通常会处理容器内所有符合条件的 BeanDefinition类似，BeanPostProcessor会处理容器内所有符合条件的实例化后的对象实例。该接口声明了两个方法，分别在两个不同的时机执行，

Public interface BeanPostProcessor{

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

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

}

#### @PostConstruct

@PostConstruct: JSR250规则定义,在Bean创建完成,且属于赋值完成后进行初始化,属于JDK规范的注解

#### InitializingBean和init-method

该接口定义很简单，其作用在于，在对象实例化过程调用过“ BeanPostProcessor的前置处理”之后，会接着检测当前对象是否实现了InitializingBean接口，如果是，则会调用其afterPropertiesSet()方法进一步调整对象实例的状态。比如，在有些情况下，某个业务对象实例化完成后，还不能处于可以使用状态。这个时候就可以让该业务对象实现该接口，并在方法 afterPropertiesSet（）中完成对该业务对象的后续处理。

虽然该接口在Spring容器内部广泛使用，但如果真的让我们的业务对象实现这个接口，则显得Spring容器比较具有侵入性。所以，Spring还提供了另一种方式来指定自定义的对象初始化操作，那就是在XML配置的时候，使用的init-method属性。

通过init-method，系统中业务对象的自定义初始化操作可以以任何方式命名，而不再受制于InitializingBean的 afterPropertiesSet（）。如果系统开发过程中规定：所有业务对象的自定义初始化操作都必须以init（）命名，为了省去挨个的设置init-method这样的烦琐，我们还可以通过最顶层的的 default-init-method统一指定这－init（）方法名。

#### DisposableBean与destroy-method

当所有的一切，该设置的设置，该注入的注入，该调用的调用完成之后，容器将检查 singleton类型的bean实例，看其是否实现了org.spring framework.beans.factory. Disposablebean接口。或者其对应的bean定义是否通过的 destroy-method属性指定了自定义的对象销毁方法。如果是，就会为该实例注册一个用于对象销毁的回调（ Callback），以便在这些 singleton类型的对象实例销毁之前，执行销毁逻辑

与 InitializingBean和init-method用于对象的自定义初始化相对应， DisposableBean和

destroy-method为对象提供了执行自定义销毁逻辑的机会。

最常见到的该功能的使用场景就是在 Spring容器中注册数据库连接池，在系统退出后，连接池应该关闭，以释放相应资源。

————————————————

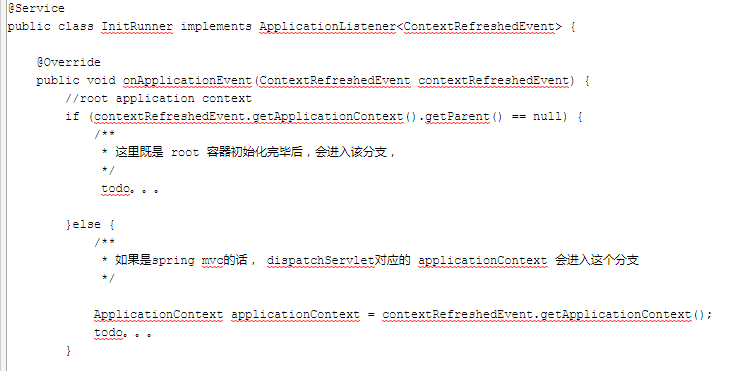
版权声明：本文为CSDN博主「zzf\_\_」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/qq\_36000403/article/details/83189280

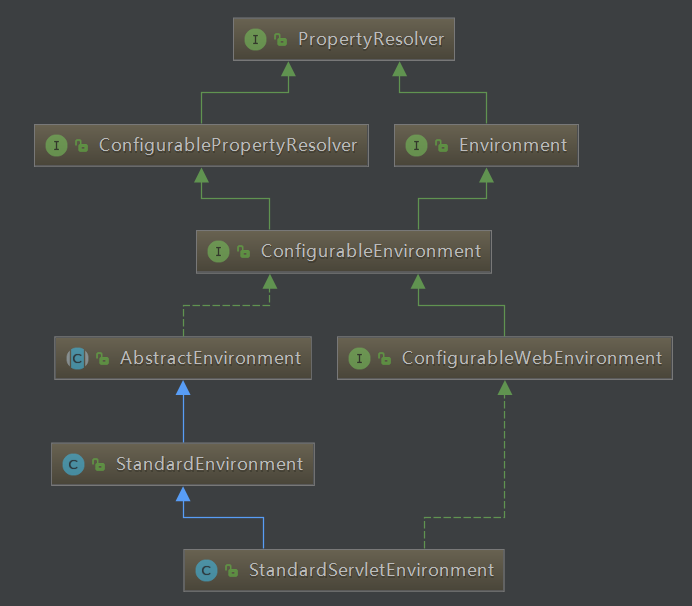
## Spring中的listener

通过 *ApplicationEvent* 类和 *ApplicationListener* 接口来提供在 *ApplicationContext* 中处理事件。如果一个 bean 实现 *ApplicationListener*，那么每次 *ApplicationEvent* 被发布到 ApplicationContext 上，那个 bean 会被通知。

|  |  |
| --- | --- |
| **序号** | **Spring 内置事件 & 描述** |
| 1 | **ContextRefreshedEvent**  *ApplicationContext* 被初始化或刷新时，该事件被发布。这也可以在 *ConfigurableApplicationContext*接口中使用 refresh() 方法来发生。 |
| 2 | **ContextStartedEvent**  当使用 *ConfigurableApplicationContext* 接口中的 start() 方法启动 *ApplicationContext* 时，该事件被发布。你可以调查你的数据库，或者你可以在接受到这个事件后重启任何停止的应用程序。 |
| 3 | **ContextStoppedEvent**  当使用 *ConfigurableApplicationContext* 接口中的 stop() 方法停止 *ApplicationContext* 时，发布这个事件。你可以在接受到这个事件后做必要的清理的工作。 |
| 4 | **ContextClosedEvent**  当使用 *ConfigurableApplicationContext* 接口中的 close() 方法关闭 *ApplicationContext* 时，该事件被发布。一个已关闭的上下文到达生命周期末端；它不能被刷新或重启。 |
| 5 | **RequestHandledEvent**  这是一个 web-specific 事件，告诉所有 bean HTTP 请求已经被服务。 |

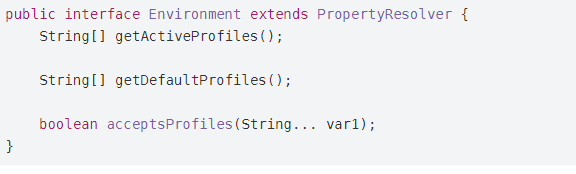


## Spring中的environment



<https://www.cnblogs.com/youzhibing/p/9622441.html>

Environment即环境，也可以叫做上下文，相当于是是我们的应用程序运行时的一个背景信息。在spring中，Environment这个概念是与spring的容器container集成在一起的一个抽象概念。它主要为我们的应用程序环境的两个方面的支持：profiles and properties。在spring库中，Environment也有其接口的代码定义，如下所示：



profile的作用就是符合这个给定profile配置的bean的集合。我们已经很熟悉spring中的bean了，而bean在定义好之后，是可以分配到一个指定的profile的。注意，这与bean定义的方式没有关系，无论是采用xml的方式还是java注解的方式定义bean，我们都可以配置其属于哪一个profile。并且这一点是在运行时确立的。

同时，当有多个profile的时候，我们可以指定哪一个Profile是生效的，如果不指定的话spring则会根据默认的profile去运行。若使用java配置的话，我们可以使用注解@Profile来指定某个bean属于哪一个profile，只有当这个指定的profile被激活处于active状态时，这个bean才会被创建。例如下面这个datasource bean的例子



properties文件顾名思义就是拿来存放属性的。它能帮我们管理各种信息，如：应用程序中的一些属性变量、JVM系统属性、系统的环境变量、servlet上下文参数、等等。properties配置文件在几乎所有的应用程序中都有着重要的应用。

在上面贴出的Environment接口代码当中，我们看但这个接口继承了另一个叫PropertyResolver的接口，这个接口是与property相关的，而Environment当中除了定义了profile的相关方法之后并没有其他和profile相关的类或接口的抽象。这一点其实不难理解，Environment和property是最基本的概念与抽象，而profile是指的生效的properties，因此它只是提供了方法的抽象，从涉及角度出发我们就能准确的理解这两个接口的定义。PropertyResolver接口的定义如下



