在Spring中，Spring IOC提供了一个基本的JavaBean容器，通过**IoC模式管理依赖关系**，并通过**依赖注入**和**AOP切面**增强了为JavaBean这样的POJO对象赋予**事务管理**、**生命周期管理**等基本功能。

在Spring IoC容器的设计中，我们可以看到两个主要的容器系列，一个是实现**BeanFactory接口**的简单容器系列，这系列容器只实现了容器的最基本功能；另一个是**ApplicationContext**应用上下文，它作为容器的高级形态而存在。

在Spring提供的基本IoC容器的接口定义和实现的基础上，**Spring通过定义BeanDefinition来管理基于Spring应用中的各种对象以及它们之间的相互依赖关系**。**BeanDefinition抽象了我们对Bean的定义**，是让容器起作用的主要数据类型。IoC容器是用来管理对象依赖关系的，对IoC容器来说，BeanDefinition就是对依赖反转模式中管理的对象依赖关系的数据抽象，也是容器实现依赖反转功能的核心数据结构，依赖反转功能都是围绕对这个BeanDefinition的处理来完成的。

BeanFactory是一个Factory，也就是IoC容器。FactoryBean是一个接口，里面有一个方法：getObject()，当某个实现了FactoryBean接口的类交给IoC容器管理时，如果调用getBean()方法，返回的是FactoryBean中getObject()方法返回的实例，只有在调用getBean()方法的时候在Bean名称前加上“$”，返回的才是IoC容器原来返回的对象。

ClassPathResource res = new ClassPathResource("beans.xml");  
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();  
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);  
reader.loadBeanDefinitions(res);

这样，我们就可以通过factory对象来使用DefaultListableBeanFactory这个IoC容器了。在使用IoC容器时，需要如下几个步骤：

（1）创建IoC配置文件的抽象资源，这个抽象资源包含了BeanDefinition的定义信息。

1. 创建一个BeanFactory，这里使用DefaultListableBeanFactory。

（3）创建一个载入BeanDefinition的读取器，这里使用XmlBeanDefinitionReader来载入XML文件形式的BeanDefinition，通过一个**回调**配置给BeanFactory。

（4）从定义好的资源位置读入配置信息，具体的解析过程由XMLBeanDefinitionReader来完成。完成整个载入和注册Bean定义之后，需要的IoC容器就建立起来了。

IoC容器的初始化时由refresh()方法来启动的，这个方法标志着IoC容器的正式启动。这个启动包括**BeanDefinition**的**Resource定位**、**载入**和**注册**三个基本过程。Spring把这三个过程分开，并使用不同的模块来完成，如使用相应的ResourceLoader、BeanDefinitionReader等模块，通过这样的设计方式，可以**让用户更加灵活地对这三个过程进行剪裁或扩展，定义出最适合自己的IoC容器的初始化过程**。

第一个过程是**Resource定位过程**。这个Resource定位指的是BeanDefinition的资源定位，它是由ResourceLoader通过统一的Resource接口来完成，这个Resource对各种形式的BeanDefinition的使用都提供了统一接口。

第二个过程是**BeanDefinition的载入**。这个载入过程是把用户定义好的Bean表示成IoC容器内部的数据结构，而这个容器内部的数据结构就是BeanDefinition。这个BeanDefinition实际上就是POJO对象在IoC容器中的抽象，通过这个BeanDefinition定义的数据结构，使IoC容器能够方便地对POJO对象也就是Bean进行管理。

第三个过程是**向IoC容器注册这些BeanDefinition的过程**。这个过程是调用BeanDefinitionRegistry接口的实现来完成的。这个注册过程把载入过程中解析得到的BeanDefinition向IoC容器进行注册。在IoC容器内部将BeanDefinition注入到一个HashMap中去，IoC容器就是通过这个HashMap来持有这些BeanDefinition数据的。

在IoC容器初始化过程中，一般不包含Bean依赖注入的实现。在Spring IoC的设计中，**Bean定义的载入和依赖注入是两个独立的过程**。依赖注入一般发生在应用第一次通过getBean()向容器索取Bean的时候。

在Bean的创建和对象依赖注入的过程中，需要依据BeanDefinition中的信息来**递归**地完成依赖注入，这些递归都是以**getBean()**为入口的。一个递归是**在上下文体系中查找需要的Bean和创建Bean的递归调用**；另一个递归是**在依赖注入时，通过递归调用容器的getBean()方法，得到当前Bean的依赖Bean，同时也触发对依赖Bean的创建和注入**。在对Bean的属性进行依赖注入时，解析的过程也是一个递归的过程。这样，根据依赖关系，一层一层地完成Bean的创建和注入，直到最后完成当前Bean的创建。**有了这个顶层Bean的创建和对它的属性依赖注入的完成，意味着和当前Bean相关的整个依赖链的注入也完成了**。

**在Bean创建和依赖注入完成以后，在IoC容器中建立起一系列依靠依赖关系联系起来的Bean，这个Bean已经不是简单的Java对象了。该Bean系列以及Bean之间的依赖关系建立完成以后，通过IoC容器的相关接口方法，就可以非常方便地供上层应用使用了**。

IoC容器中的Bean生命周期：

1. Bean实例的创建。
2. 为Bean实例设置属性。
3. 调用Bean的初始化方法。

在调用Bean的初始化方法之前，会调用一系列的aware接口实现，把相关的**BeanName**、**BeanClassLoader**，以及**BeanFactory**注入到Bean中去。接着会看到对invokeInitMethods()的调用，这时还会看到启动**afterPropertiesSet()**的过程，当然，这需要Bean实现**InitializingBean**的接口，对应的初始化处理可以在InitializingBean接口的afterPropertiesSet()方法中实现，这里同样是对Bean的一个**回调**。最后，还会看到判断Bean是否配置有**init-method**，如果有，那么通过invokeCustomInitMethod()方法来直接调用，最终完成Bean的初始化。

1. 应用可以通过IoC容器使用Bean。
2. 当容器关闭时，调用Bean的销毁方法。

在Bean的销毁过程中，首先对**postProcessBeforeDestruction()**进行调用，然后调用Bean的**destroy()**方法，最后是对Bean的**自定义销毁方法**的调用。

BeanPostProcessor是一个监听器，它可以监听容器触发的事件。将它向IoC容器注册后，容器中管理的Bean具备了接收IoC容器事件回调的能力。BeanPostProcessor的使用非常简单，只需要通过设计一个具体的后置处理器来实现。同时，这个具体的后置处理器需要实现接口类BeanPostProcessor，然后设置到XML的Bean配置文件中。这个BeanPostProcessor是一个接口，它有两个方法：一个是postProcessBeforeInitialization()，在Bean的初始化前提供回调入口；一个是postProcessAfterInitialization()，在Bean的初始化后提供回调入口，这两个回调的触发都是和容器管理Bean的生命周期相关的。

调用顺序：**postProcessBeforeInitialization() -> afterPropertiesSet() -> init-method -> applyProcessAfterInitialization()**

在自动装配中，不需要对Bean属性做显式的依赖关系声明，只需要配置好autowiring属性，IoC容器会根据这个属性的配置，自动查找属性的类型或者名字，然后基于属性的类型或名字来自动匹配IoC容器中的Bean，从而自动地完成依赖注入。

关于容器的基本工作原理，可以大致整理出以下几个方面：

（1）**BeanDefinition的定位**

对于IoC容器来说，它为管理POJO之间的依赖关系提供了帮助，但也要依据Spring的定义规则提供Bean定义信息。我们可以使用各种形式的Bean定义信息，其中比较熟悉和常用的是使用XML的文件格式。在Bean定义方面，Spring为用户提供了很大的灵活性。在初始化IoC容器的过程中，首先需要定位到这些有效的Bean定义信息，这里**Spring使用Resource接口来统一这些Bean定义信息，而这个定位由ResourceLoader来完成**。**如果使用上下文，ApplicationContext本身就为客户提供了定位的功能。因为上下文本身就是DefaultResourceLoader的子类**。**如果使用基本的BeanFactory作为IoC容器，客户需要做的额外工作就是为BeanFactory指定相应的Resource来完成Bean信息的定位**。

（2）**容器的初始化**

在使用上下文时，需要一个对它进行初始化的过程，完成初始化以后，这个IoC容器才是可用的。这个过程的入口是在refresh()中实现的，这个**refresh()相当于容器的初始化函数**。在初始化过程中，比较重要的部分是**对BeanDefinition信息的载入和注册工作**。相当于在IoC容器中需要建立一个BeanDefinition定义的数据映像，**Spring为了达到载入的灵活性，把载入的功能从IoC容器中分离出来，由BeanDefinitionReader来完成Bean定义信息的读取、解析和IoC容器内部BeanDefinition的建立**。**在DefaultListableBeanFactory中，这些BeanDefinition被维护在一个HashMap中，以后的IoC容器对Bean的管理和操作就是通过这些BeanDefinition来完成的**。

在容器初始化完成以后，IoC容器的使用就准备好了，但这时只是在IoC容器内部建立了BeanDefinition，具体的依赖关系还没有注入。在客户第一次向IoC容器请求Bean时，IoC容器对相关的Bean依赖关系进行注入。如果需要提前注入，客户可以通过lazy-init属性进行预实例化，这个预实例化是上下文初始化的一部分，起到提前完成依赖注入的控制作用。在依赖注入完成以后，IoC容器就会保持这些具备依赖关系的Bean供客户直接使用。这时可以通过getBean()来取得Bean，这些Bean不是简单的Java对象，而是已经包含了对象之间依赖关系的Bean，尽管这些依赖注入的过程对用户来说是不可见的。

“**基础**”（base）可以视为**待增强对象**或者说**目标对象**。“**切面**”（aspect）通常包含**对于基础的增强应用**。“**配置**”（configuration）可以看成是一种**编织**，通过在AOP体系中提供这个配置环境，可以把基础和切面结合起来，从而完成切面对目标对象的编织实现。

**Advice**（通知）**定义在连接点做什么，为切面增强提供织入接口**。在Spring AOP中，它主要描述Spring AOP围绕方法调用而注入的切面行为。Advice是AOP联盟定义的一个接口。在Spring AOP的实现中，使用了这个统一接口，并通过这个接口，为AOP切面增强的织入功能做了更多的细化和扩展，比如提供了更具体的通知类型，如BeforeAdvice、AfterAdvice、ThrowsAdvice等。作为Spring AOP定义的接口类，具体的切面增强可以通过这些接口集成到AOP框架中去发挥作用。