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容器的初始化是由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 -> postProcessAfterInitialization()**

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。

**Pointcut**（切点）决定Advice通知应该作用于哪个连接点，也就是说**通过Pointcut来定义需要增强的方法的集合**，这些集合的选取可以按照一定的规则来完成。在这种情况下，Pointcut通常意味着标识方法，例如，这些需要增强的地方可以由某个正则表达式进行标识，或根据某个方法名进行匹配等。

完成对目标方法的切面增强设计（Advice）和关注点的设计（Pointcut）以后，需要一个对象把它们结合起来，完成这个作用的就是**Advisor**（通知器）。通过Advisor，可以定义应该使用哪个通知并在哪个关注点使用它，也就是说通过Advisor，**把Advice和Pointcut结合起来**，这个结合为使用IoC容器配置AOP应用，或者说即开即用地使用AOP基础设施，提供了便利。

**通过使用AopProxy对象封装target目标对象之后，ProxyFactoryBean的getObject()方法得到的对象就不是一个普通的Java对象了，而是一个AopProxy代理对象**。在ProxyFactoryBean中配置的target目标对象，这时已经不会让应用直接调用其方法实现，而是作为AOP实现的一部分。对target目标对象的方法调用会首先被AopProxy代理对象拦截，对于不同的AopProxy代理对象生成方式，会使用不同的拦截回调入口。例如，**对于JDK的AopProxy代理对象，使用的是InvocationHandler的invoke回调入口**；而**对于CGLIB的AopProxy代理对象，使用的是设置好的callback回调，这是由对CGLIB的使用来决定的**。在这些callback回调中，对于AOP实现，是通过DynamicAdvisedInterceptor来完成的，而DynamicAdvisedInterceptor的回调入口是intercept()方法。通过这一系列的准备，已经为实现AOP的横切机制奠定了基础，在这个基础上，AOP的Advisor已经可以通过AopProxy代理对象的拦截机制，对需要它进行增强的target目标对象发挥切面的强大威力了。

可以把AOP的实现部分看成是由**基础设施准备**和**AOP运行辅助**这两个部分组成，这里的**AopProxy代理对象的生成，可以看成是一个静态的AOP基础设施的建立过程**。通过这个准备过程，把代理对象、拦截器这些待调用的部分都准备好，等待着AOP运行过程中对这些基础设施的使用。**对于应用触发的AOP应用，会涉及AOP框架的运行和对AOP基础设施的使用**。这些动态的运行部分，是从前面提到的拦截器回调入口开始的，这些拦截器调用的实现原理，和AopProxy代理对象生成一样，也是AOP实现的重要组成部分。

对于IoC容器的使用，如果需要回调容器，前提是当前的Bean需要实现BeanFactoryAware接口，这个接口只需要实现一个接口方法setBeanFactory，同时设置一个属性来持有BeanFactory的IoC容器，就可以在Bean中取得IoC容器进行回调了。在IoC容器对Bean进行初始化的时候，会对Bean的类型进行判断，如果这是一个BeanFactoryAware的Bean类型，那么IoC容器会调用这个Bean的setBeanFactory方法，完成对这个BeanFactory在Bean中的设置。

Spring IoC是一个独立的模块，它并不是直接在Web容器中发挥作用的，如果要在Web环境中使用IoC容器，需要Spring为IoC设计一个启动过程，把IoC容器导入，并在Web容器中建立起来。具体说来，**这个启动过程是和Web容器的启动过程集成在一起的**。在这个过程中，**一方面处理Web容器的启动，另一方面通过设计特定的Web容器拦截器，将IoC容器载入到Web环境中来，并将其初始化**。在这个过程建立完成以后，IoC容器才能正常工作，**而Spring MVC是建立在IoC容器的基础上的**，这样才能建立起MVC框架的运行机制，从而响应从Web容器传递的HTTP请求。

<servlet>

<servlet-name>sample</servlet-name>

<servlet-class>org.springframework.web.servlet.**DispatcherServlet**</servlet-class>

<load-on-startup>2</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>sample</servlet-name>

<url-pattern>/\*</url-pattern>

</servlet-mapping>

<**context-param**>

<param-name>**contextConfigLocation**</param-name>

<param-value>/WEB-INF/applicationContext.xml</param-value>

<**/context-param**>

<listener>

<listener-class>org.springframework.web.context.**ContextLoaderListener**</listener-class>

</listener>

在这个部署描述文件中，首先定义了一个Servlet对象，它是Spring MVC的DispatcherServlet。这个DispatcherServlet是MVC中很重要的一个类，起着**分发请求**的作用。

同时，在部署描述中，为这个DispatcherServlet定义了对应的URL映射，这些URL映射为这个Servlet指定了需要处理的HTTP请求。context-param参数的配置用来指定Spring IoC容器读取Bean定义的XML文件的路径，在这里，这个配置文件被定义为/WEB-INF/applicationContext.xml。在这个文件中，可以看到Spring应用的Bean配置。最后，作为Spring MVC的启动类，**ContextLoaderListener被定义为一个监听器，这个监听器是与Web服务器的生命周期相关联的，由ContextLoaderListener监听器负责完成IoC容器在Web环境中的启动过程**。

**DispatcherServlet和ContextLoaderListener提供了在Web容器中对Spring的接口**，也就是说，**这些接口与Web容器耦合是通过ServletContext来实现的**。**这个ServletContext为Spring的IoC容器提供了一个宿主环境，在宿主环境中，Spring MVC建立起一个IoC容器的体系**。**这个IoC容器体系是通过ContextLoaderListener的初始化来建立的，在建立IoC容器体系后，把DIspatcherServlet作为Spring MVC处理Web请求的转发器建立起来，从而完成响应HTTP请求的准备**。

**IoC容器的启动过程就是建立上下文的过程，该上下文是与ServletContext相伴而生的，同时也是IoC容器在Web应用环境中的具体表现之一**。由**ContextLoaderListener**启动的上下文为**根上下文**。在根上下文的基础上，还有一个**与Web MVC相关的上下文**用来保存控制器（DispatcherServlet）需要的MVC对象，**作为根上下文的子上下文，构成一个层次化的上下文体系**。在Web容器中启动Spring应用程序时，首先建立根上下文，然后建立这个上下文体系的，这个上下文体系的建立是由**ContextLoader**来完成的。

在web.xml中，已经配置了ContextLoaderListener，这个ContextLoaderListener是Spring提供的类，是为在Web容器中建立IoC容器服务的，它实现了ServletContextListener接口。这个接口是在Servlet API中定义的，提供了与Servlet生命周期结合的回调，比如contextInitialized()方法和contextDestroyed()方法。而在Web容器中，建立WebApplicationContext的过程，是在contextInitialized()的接口实现中完成的。具体的载入IoC容器的过程是由ContextLoaderListener交由ContextLoader来完成的，而ContextLoader本身就是ContextLoaderListener的基类。

在ContextLoader中，完成了两个IoC容器建立的基本过程，一个是**在Web容器中建立起双亲IoC容器**，另一个是**生成相应的WebApplicationContext并将其初始化**。

**对于Spring 承载的Web应用而言，可以指定在Web应用程序启动时载入IoC容器**（或者称为WebApplicationContext）。**这个功能是由ContextLoaderListener这样的类来完成的，它是在Web容器中配置的监听器。这个ContextLoaderListener通过使用ContextLoader来完成实际的WebApplicationContext，也就是IoC容器的初始化工作**。**这个ContextLoader就像Spring应用程序在Web容器中的启动器**。这个启动过程是在Web容器中发生的，所以需要根据Web容器部署的要求来定义ContextLoader。

为了了解IoC容器在Web容器中的启动原理，这里对启动器ContextLoaderListener的实现进行分析。**这个监听器是启动根IoC容器并把它载入到Web容器的主要功能模块，也是整个Spring Web应用加载IoC的第一个地方**。从加载过程可以看到，**首先从Servlet事件中得到ServletContext，然后可以读取配置在web.xml中的各个相关的属性值，接着ContextLoader会实例化WebApplicationContext，并完成其载入和初始化过程**。**这个被初始化的第一个上下文作为根上下文而存在，这个根上下文载入后，被绑定到Web应用程序的ServletContext上**。任何需要访问根上下文的应用程序代码都可以从WebApplicationContextUtils类的静态方法中取到。

**在ContextLoaderListener中，实现的是ServletContextListener接口，这个接口里的函数会结合Web容器的生命周期被调用**。**因为ServletContextListener是ServletContext的监听者，如果ServletContext发生变化，会触发出相应的事件，而监听器一直在对这些事件进行监听，如果接收到来监听的事件，就会做出预先设计好的响应动作**。由于ServletContext的变化而触发的监听器的响应具体包括：在服务器启动时，ServletContext被创建的时候；服务器关闭时，ServletContext将被消耗的时候等。对应这些事件及Web容器状态的变化，在监听器中定义了对应的事件响应的回调方法。比如在服务器启动时，ServletContextListener的contextInitialized()方法被调用，服务器关闭时，ServletContextListener的contextDestroyed()方法被调用。

在完成对ContextLoaderListener的初始化后，Web容器开始初始化DispatcherServlet，这个初始化的启动与在web.xml中对载入次序的定义有关。**DispatcherServlet会建立自己的上下文来持有Spring MVC的Bean对象，在建立这个自己持有的IoC容器时，会从ServletContext中得到根上下文作为DIspatcherServlet持有上下文的双亲上下文**。有了这个根上下文，再对自己持有的上下文进行初始化，最后把自己持有的这个上下文保存到ServletContext中，供以后检索和使用。

DIspatcherServlet通过集成FrameworkServlet和HttpServletBean而继承了HttpServlet，**通过使用Servlet API来对HTTP请求进行响应**，成为Spring MVC的前端处理器，同时成为MVC模块与Web容器集成的处理前端。

DIspatcherServlet的工作大致可以分为两个部分：一是初始化部分，由initServletBean()启动，通过initWebApplicationContext()方法最终调用DIspatcherServlet的initStrategies()方法，在这个方法里，DIspatcherServlet对MVC模块的其他部分进行了初始化，比如handlerMapping、ViewResolver等；另一个是对HTTP请求进行响应，作为一个Servlet，Web容器会调用Servlet的doGet()和doPost()方法，在经过FrameworkServlet的processRequest()简单处理后，会调用DIspatcherServlet的doService()方法，在这个方法调用中封装了doDispatch()，这个doDispatch()是Dispatcher实现MVC模式的主要部分。

**根上下文是和Web应用相对应的一个上下文，而DispatcherServlet持有的上下文是和Servlet对应的一个上下文**。**在根上下文中定义的Bean是可以被各个Servlet持有的上下文得到和共享的**。

Spring MVC的实现大致由以下几个步骤完成：

（1）需要建立Controller控制器和HTTP请求之间的映射关系，即在Spring MVC实现中是如何根据请求得到对应的Controller的？在Spring MVC中，这个工作是由在HandlerMapping中封装的HandlerExecutionChain对象来完成的，而对Controller控制其和HTTP请求的映射关系的配置是在Bean定义中描述，并在IoC容器初始化时，通过初始化HandlerMapping来完成的，这些定义的映射关系会被载入到一个handlerMap中使用。

（2）在初始化过程中，Controller和HTTP请求之间的映射关系建立好以后，为Spring MVC接收HTTP请求并完成响应处理做好了准备。**在MVC框架接收到HTTP请求的时候，DispatcherServlet会根据具体的URL请求信息，在HandlerMapping中进行查询，从而得到对应的HanderExecutionChain。在这个HandlerExecutionChain中封装了配置的Controller，这个请求对应的Controller会完成请求的响应动作，生成需要的ModelAndView对象，可以从该对象中获得Model模型数据和视图对象**。

（3）**得到ModelAndView以后，DispatcherServlet把获得的模型数据交给特定的视图对象，从而完成这些数据的视图呈现工作。这个视图呈现由视图对象的render方法来完成。对应于不同的视图对象，render方法会完成不同的视图呈现处理，为用户提供丰富的Web UI表现**。

在Spring事务处理中，可以通过设计一个TransactionProxyFactoryBean来使用AOP功能，通过这个TransactionProxyFactoryBean可以生成Proxy代理对象，在这个代理对象中，通过TranscationInterceptor来完成对代理方法的拦截，正是这些AOP的拦截功能，将事务处理的功能编织进来。

Spring作为应用平台或框架的设计出发点是支持POJO的开发，这点在实现事务处理的时候也不例外。在Spring中，它既支持编程式事务管理方式，又支持声明式事务管理方式，在使用Spring处理事务的时候，声明式事务处理通常比编程式事务管理更方便一些。

Spring对应用的支持，一方面，通过声明式事务处理，将事务处理的过程和业务代码分离出来。这种声明方式实际上是通过AOP的方式来完成。显然，Spring已经把那些通用的事务处理过程抽象出来，并通过AOP的方式进行封装，然后用声明式的使用方式交付给客户使用。这样，应用程序可以更简单地管理事务，并且只需要关注事务的处理策略。另一方面，应用在选择数据源时可能会采取不同的方案，当以Spring作为平台时，Spring在应用和具体的数据源之间，搭建一个中间平台，通过这个中间平台，解耦应用和具体数据源之间的绑定，并且，Spring为常用的数据源的事务处理支持提供了一系列的TranscationManager。这些Spring封装好的TranscationManager为应用提供了很大的方便，因为在这些具体事务处理中，已经根据底层的实现，封装好了事务处理的设置以及与特定数据源相关的特定事务处理过程，这样应用在使用不同的数据源时，可以做到事务处理的即开即用。这样的另外一个好处是，如果应用有其他的数据源事务处理需要，Spring也提供了一种一致的方式，这种有机的事务过程抽象和具体的事务处理相结合的设计，仔细体会一下会发现它是非常精妙的。

在使用Spring声明式事务处理的时候，一种常用的方法是结合IoC容器和Spring已有的**TransactionProxyFactoryBean**对事务管理进行配置，比如，可以在这个TransactionProxyFactoryBean中为事务方法配置传播行为、并发事务隔离级别等事务处理属性，从而对声明式事务的处理提供指导。具体来说，在以下的内容中，在对声明式事务处理的原理分析中，声明式事务的实现大致可以分为以下几个部分：

（1）**读取和处理在IoC容器中配置的事务处理属性，并转化为Spring事务处理需要的内部数据结构**。具体来说，这里涉及的类是TransactionAttributeSourceAdvisor，从名字可以看出，它是一个AOP通知器，Spring使用这个通知器来完成对事务处理属性值的处理。处理的结果是，在IoC容器中配置的事务处理属性信息，会被读入并转化成TranscationAttribute表示的数据对象，这个数据对象是Spring对事务处理属性值的数据抽象，对这些属性的处理是和TransactionProxyFactoryBean拦截下来的事务方法的处理结合起来的。

（2）**Spring事务处理模块实现统一的事务处理过程**。这个通用的事务处理过程包含处理事务配置属性，以及与线程绑定完成事务处理的过程，Spring通过TransactionInfo和TransactionStatus这两个数据对象，在事务处理过程中记录和传递相关执行场景。

（3）**底层的事务处理实现**。对于底层的事务操作，Spring委托给具体的事务处理器来完成，这些具体的事务处理器，就是在IoC容器中配置声明式事务处理时，配置的PlatformTransactionManager的具体实现，比如DataSourceTransactionManager和HibernateTransactionManager等。

从Spring实现的角度来看，声明式事务处理的大致实现过程是这样的：在**为事务处理配置好AOP的基础设施**（比如，对应的Proxy代理对象和事务处理Interceptor拦截器对象）之后，首先需要完成**对这些事务属性配置的读取**，这些属性的读取处理是在TransactionInterceptor中实现的；在完成这些事务处理属性的读取之后，Spring为事务处理的具体实现做好了准备。可以看到，**Spring声明式事务处理的过程同时也是一个整合事务处理实现到Spring AOP和IoC容器中去的过程**。我们在整个过程中可以看到下面一些要点，在这些要点中，体现了对Spring框架的基本特性的灵活使用。

（1）如何封装各种不同事务处理环境下的事务处理，具体来说，作为应用平台的Spring它没法对应用使用什么样的事务处理环境做出限制，这样，对应用户使用的不同的事务处理器，Spring事务处理平台都需要为用户提供服务。这些事务处理实现包括在应用中常见的DataSource的Connection、Hibernate的Transaction等，Spring事务处理通过一种统一的方式把它们封装起来，从而实现一个通用的事务处理过程，实现这部分事务处理对应用透明，使应用即开即用。

（2）如何读取事务处理属性值，在事务处理属性正确读取的基础上，结合事务处理代码，从而完成在既定的事务处理配置下，事务处理方法的实现。

（3）如何灵活地使用Spring AOP框架，对事务处理进行封装，提供给应用即开即用的声明式事务处理功能。

在这个过程中，有几个Spring事务处理的核心类是我们需要关注的。其中包括**TransactionInterceptor**，它是使用AOP实现声明式事务处理的拦截器，封装了Spring对声明式事务处理实现的基本过程；还包括**TransactionAttributeSource**和**TransactionAttribute**这两个类，它们封装了对声明式事务处理属性的识别，以及信息的读入和配置。我们看到的TransactionAttribute对象，可以视为对事务处理属性的数据抽象，如果在使用声明式事务处理的时候，应用没有配置这些属性，Spring将会用户提供DefaultTransactionAttribute对象，在这个DefaultTransactionAttribute对象中，提供了默认的事务处理属性设置。

在事务处理过程中，可以看到**TransactionInfo**和**TransactionStatus**这两个对象，它们是存放事务处理信息的主要数据对象，它们通过与线程的绑定来实现事务的隔离性。具体来说，**TransactionInfo对象本身就像是一个栈，对应着每一次事务方法的调用，它会保存每一次事务方法调用的事务处理信息**。值得注意的是，在TransactionInfo对象中，它持有TransactionStatus对象，这个TransactionStatus是非常重要的，由**这个TransactionStatus来掌管事务执行的详细信息，包括具体的事务对象、事务执行状态、事务设置状态等**。在事务的创建、启动、提交和回滚的过程中，都需要与这个TransactionStatus对象中的数据打交道。在准备完这些与事务管理有关的数据之后，具体的事务处理是由事务处理器TransactionManager来完成的。**在事务处理器完成事务处理的过程中，与具体事务处理器无关的操作都被封装到AbstractPlatformTransactionManager中实现了。这个抽象的事务处理器为不同的具体事务处理器提供了通用的事务处理模板，它封装了在事务处理过程中，与具体事务处理器无关的公共的事务处理部分**。我们在具体的事务处理器（比如DataSourceTransactionManager和HibernateTransactonManager）的实现中可以看到，最为底层的事务创建、挂起、提交、回滚操作。