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。

（2） 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。

（4） 应用可以通过 IoC 容器使用 Bean。

（5） 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。