▶ **关于BeanFactory和FactoryBean的区别**

BeanFactory是个Factory，也就是IOC容器或对象⼯⼚，FactoryBean是个Bean。在Spring中，所有的 Bean都是由BeanFactory(也就是IOC容器)来进⾏管理的。但对FactoryBean⽽⾔，这个Bean不是简单 的Bean，⽽是⼀个能⽣产或者修饰对象⽣成的⼯⼚Bean,它的实现与设计模式中的⼯⼚模式和修饰器模式类似

▶ **7种事务的传播机制（可通过spring配置或注解来设置）**

1. REQUIRED（默认）：支持使用当前事务，如果当前事务不存在，创建一个新事务。
2. SUPPORTS：支持使用当前事务，如果当前事务不存在，则不使用事务。
3. MANDATORY：中文翻译为强制，支持使用当前事务，如果当前事务不存在，则抛出Exception。
4. REQUIRES\_NEW：创建一个新事务，如果当前事务存在，把当前事务挂起。
5. NOT\_SUPPORTED：无事务执行，如果当前事务存在，把当前事务挂起。
6. NEVER：无事务执行，如果当前有事务则抛出Exception。
7. NESTED：嵌套事务，如果当前事务存在，那么在嵌套的事务中执行。如果当前事务不存在，则表现跟REQUIRED一样。

注解配置时如：@Transactional(propagation=Propagation.REQUIRED)

▶ **Spring事务四种隔离级别**

注解配置时如：@Transactional(isolation = Isolation.READ\_UNCOMMITTED)

|  |  |  |  |
| --- | --- | --- | --- |
| 事务隔离级别 | 脏读 | 不可重复读 | 幻读 |
| 读未提交（read uncommited） | 是 | 是 | 是 |
| 读已提交（read commited） | 否 | 是 | 是 |
| 可重复读（repeatable read） | 否 | 否 | 是 |
| 串行化（serializable） | 否 | 否 | 否 |

MySQL默认：可重复读

Oracle默认：读已提交

其中：可重复读(repeatable-read)表示：在开始读取数据（事务开启）时，不再允许修改操作

**mysql默认的事务隔离级别为repeatable-read**

**事务的并发问题：**

1、脏读：事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据

　　2、不可重复读：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致。

　　3、幻读：系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

小结：不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表

▶**Spring中bean的作用域与生命周期**

|  |  |
| --- | --- |
| **类别** | **说明** |
| **singleton** | 在Spring IOC容器中仅存在一个Bean实例，Bean以单例方式存在，默认值 |
| **prototype** | 每次从容器中调用Bean时，都返回一个新的实例，即每次调用getBean()时，相当于执行 new XxxBean() |
| **request** | 每次HTTP请求都会创建一个新的Bean，该作用域仅适用于WebApplicationContext环境 |
| **session** | 同一个HTTP Session共享一个Bean，不同Session使用不同Bean，仅适用于WebApplicationContext环境 |
| **globalSession** | 一般用于Portlet应用环境，该作用域适用于WebApplicationContext环境 |

五种作用域中，request、session 和 global session 三种作用域仅在基于web的应用中使用（不必关 心你所采用的是什么web应用框架），只能用在基于 web 的 Spring ApplicationContext 环境。

1. singleton——唯一 bean 实例

当一个 bean 的作用域为 singleton，那么Spring IoC容器中只会存在一个共享的 bean 实例，并且所有对 bean 的请求，只要 id 与该 bean 定义相匹配，则只会返回bean的同一实例

1. prototype——每次请求都会创建一个新的 bean 实例
2. request——每一次HTTP请求都会产生一个新的bean，该bean 仅在当前HTTP request内有效
3. session——每一次HTTP请求都会产生一个新的 bean，该bean 仅在当前 HTTP session 内有效
4. globalsession 作用域中定义的bean被限定于全局portletSession的生命周期范围内

▶**Spring中使用了哪些设计模式?**

1. 工厂模式 ：Spring使用工厂模式通过 BeanFactory、ApplicationContext 创建 bean 对象。
2. 单例模式 ：在 spring 配置文件中定义的 bean 默认为单例模式。
3. 适配器模式 ：Spring AOP 的增强或通知(Advice)使用到了适配器模式、spring MVC 中也是用到了适 配器模式适配Controller
4. 包装器模式
5. 代理模式 :在 AOP 和 remoting 中被用的比较多
6. 观察者模式 ：Spring 事件驱动模型就是观察者模式很经典的一个应用。
7. 策略模式
8. 模板方法模式 ：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。

▶**JDK动态代理和CGLib动态代理区别**

|  |  |  |
| --- | --- | --- |
|  | Cglib | JDK |
| 是否提供子类代理 | 是 | 否 |
| 是否提供接口代理 | 是（可强制） | 是 |
| 区别 | 利用ASM,必须依赖于CGLib的类库，但是它需要类来实现任何接口代理的是指定的类生成一个子类，覆盖其中的方法 | 实现InvocationHandler  在调用具体方法前调用InvocationHandler的invoke方法来处理  被代理的对象必须要实现接口 |

1、如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP  
2、如果目标对象实现了接口，可以强制使用CGLIB实现AOP

3、如果目标对象没有实现了接口，必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换

注：JDK动态代理要比cglib代理执行速度快，但性能不如cglib好。一般单例模式用cglib比较好。

如何强制使用CGLIB实现AOP？  
 （1）添加CGLIB库，SPRING\_HOME/cglib/\*.jar  
 （2）在spring配置文件中加入<aop:aspectj-autoproxy proxy-target-class="true"/>

▶**Spring bean的生命周期**

①. 通过构造器或工厂方法创建 Bean 实例

②. 为 Bean 的属性设置值和对其他 Bean 的引用

③ . 将 Bean 实 例 传 递 给 Bean 前 置 处 理 器 的 postProcessBeforeInitialization 方 法

④. 调用 Bean 的初始化方法(init-method)

⑤ . 将 Bean 实 例 传 递 给 Bean 后 置 处 理 器 的 postProcessAfterInitialization 方法

⑦. Bean 可以使用了

⑧. 当容器关闭时, 调用 Bean 的销毁方法(destroy-method)

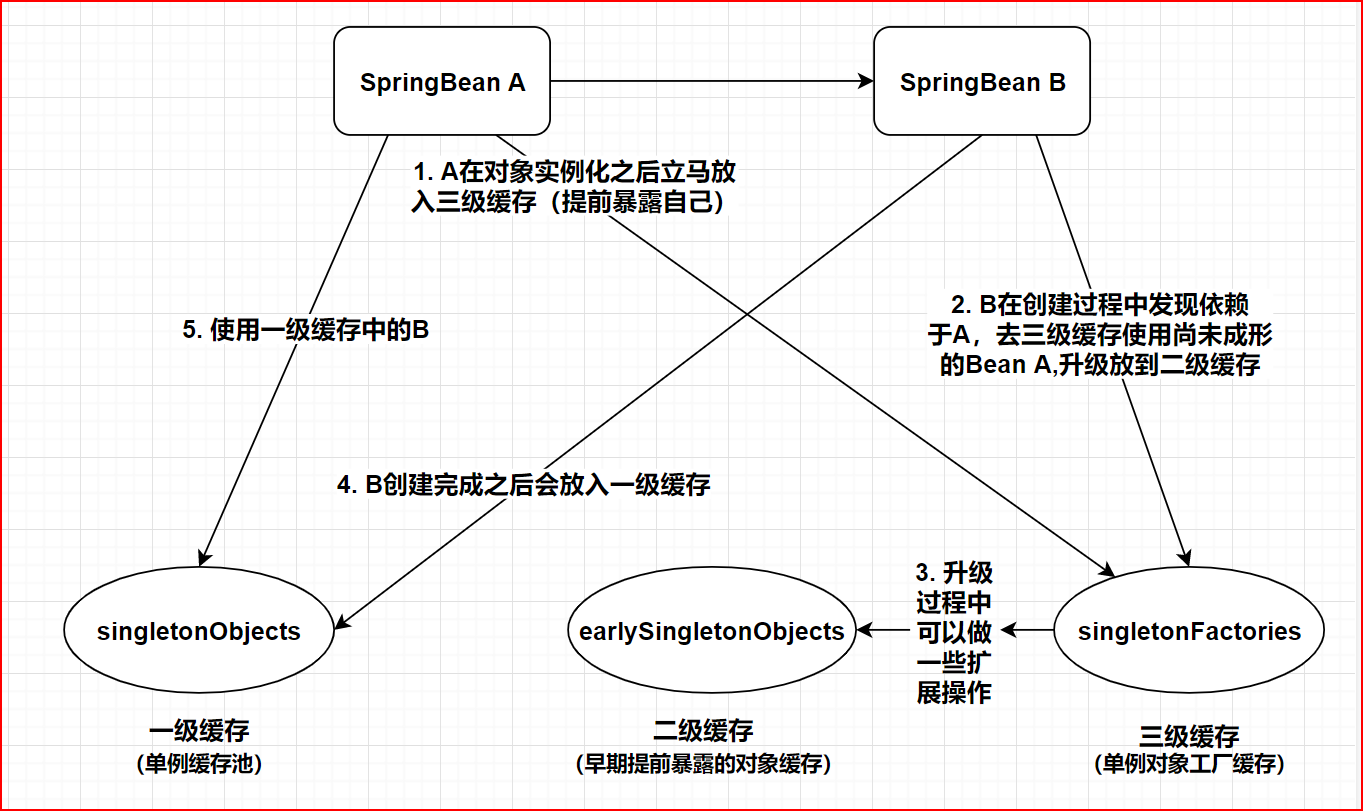
▶**Spring循环依赖及解决方式**

Spring中循环依赖场景有：

构造器的循环依赖（构造器注⼊）

Field 属性的循环依赖（set注⼊）

其中，构造器的循环依赖问题⽆法解决，只能拋出 BeanCurrentlyInCreationException 异常，在解决 属性循环依赖时，spring采⽤的是提前暴露对象的⽅法。



•singletonFactories ： 存的是Bean工厂对象，用来生成半成品的Bean并放入到二级缓存中。用以解决循环依赖。

•earlySingletonObjects ：存放半成品的Bean，半成品的Bean是已创建对象，但是未注入属性和初始化。用以解决循环依赖。

•singletonObjects：存放可用的成品Bean。

比如：

Spring 通过三级缓存解决了循环依赖，其中一级缓存为单例池（singletonObjects）,二级缓存为早期曝光对象 earlySingletonObjects，三级缓存为早期曝光对象工厂（singletonFactories）。

当 A、B 两个类发生循环引用时，在 A 完成实例化后，就使用实例化后的对象去创建一个对象工厂，并添加到三级缓存中，如果 A 被 AOP 代理，那么通过这个工厂获取到的就是 A 代理后的对象，如果 A 没有被 AOP 代理，那么这个工厂获取到的就是 A 实例化的对象。

当 A 进行属性注入时，会去创建 B，同时 B 又依赖了 A，所以创建 B 的同时又会去调用 getBean(a)来获取需要的依赖，此时的 getBean(a)会从缓存中获取：

第一步，先获取到三级缓存中的工厂；

第二步，调用对象工工厂的 getObject 方法来获取到对应的对象，得到这个对象后将其注入到 B 中。紧接着 B 会走完它的生命周期流程，包括初始化、后置处理器等。

当 B 创建完后，会将 B 再注入到 A 中，此时 A 再完成它的整个生命周期。

▶为什么要使用三级缓存呢：

如果要使用二级缓存解决循环依赖，意味着需要代理的Bean在实例化后不确定是否产生循环依赖，所以需要提前完成 AOP 代理，这样违背了 Spring 设计的原则

Spring 在设计之初就是通过 AnnotationAwareAspectJAutoProxyCreator 这个后置处理器来在 Bean 生命周期的最后一步来完成 AOP 代理，而不是在实例化后就立马进行 AOP 代理。

▶**谈谈对Spring AOP的理解**

•**Aspect（切面）**： Aspect 声明类似于 Java 中的类声明，在 Aspect 中会包含着一些 Pointcut 以及相应的 Advice。

•**Joint point（连接点）**：表示在程序中明确定义的点，典型的包括方法调用，对类成员的访问以及异常处理程序块的执行等等，它自身还可以嵌套其它 joint point。

•**Pointcut（切点）**：表示一组 joint point，这些 joint point 或是通过逻辑关系组合起来，或是通过通配、正则表达式等方式集中起来，它定义了相应的 Advice 将要发生的地方。

•**Advice（增强）**：Advice 定义了在 Pointcut 里面定义的程序点具体要做的操作，它通过 before、after 和 around 来区别是在每个 joint point 之前、之后还是代替执行的代码。

•**Target（目标对象）**：织入 Advice 的目标对象.。

•**Weaving（织入）**：将 Aspect 和其他对象连接起来, 并创建 Adviced object 的过程

****Advice 的类型****

●before：前置通知，在一个方法执行前被调用。 

●after:在方法执行之后调用的通知，无论方法执行是否成功。 

●after-returning:仅当方法成功完成后执行的通知。 

●after-throwing:在方法抛出异常退出时执行的通知。 

●around:在方法执行之前和之后调用的通知。

▶**@Component 和 @Bean 的区别**

●作用对象不同，@Component 注解作用于类，而@Bean注解作用于方法。

●@Component通常是通过类路径扫描来自动侦测以及自动装配到Spring容器中（ @ComponentScan注解）。@Bean 注解通常是指有该注解的方法中定义产生这个 bean，通常和@Configuration注解使用。

▶**Spring mvc执行流程**

1. 客户端发出一个http请求给web服务器，web服务器对http请求进行解析，如果匹配DispatcherServlet的请求映射路径（在web.xml中指定），web容器将请求转交给DispatcherServlet.  
    2、DipatcherServlet接收到这个请求之后将根据请求的信息（包括URL、Http方法、请求报文头和请求参数Cookie等）以及HandlerMapping的配置找到处理请求的处理器（Handler）。  
    3-4、DispatcherServlet根据HandlerMapping找到对应的Handler,将处理权交给Handler（Handler将具体的处理进行封装），再由具体的HandlerAdapter对Handler进行具体的调用。  
    5、Handler对数据处理完成以后将返回一个ModelAndView()对象给DispatcherServlet。  
    6、Handler返回的ModelAndView()只是一个逻辑视图并不是一个正式的视图，DispatcherSevlet通过ViewResolver将逻辑视图转化为真正的视图View。  
    7、Dispatcher通过model解析出ModelAndView()中的参数进行解析最终展现出完整的view并返回给客户端。

#### ▶**spring bean 大多数是单例，在多线程下安全的原因**

在Spring中，绝大部分Bean都可以声明为singleton作用域。就是因为Spring对一些Bean（如RequestContextHolder、TransactionSynchronizationManager、LocaleContextHolder等）中非线程安全状态采用ThreadLocal进行处理，让有状态的Bean就可以在多线程中共享了。

最浅显的解决办法就是将多态 bean 的作用域由“singleton”变更为“prototype”。

▶**在 Java 中依赖注入有以下三种实现方式：**

1. 构造器注入

2. Setter 方法注入

3. 接口注入

▶**Spring 有几种配置方式？**

1. 基于 XML 的配置
2. 基于注解的配置
3. 基于 Java 的配置

▶**spring事务的传播属性**

事务传播行为（propagation behavior）指的就是当一个事务方法被另一个事务方法调用时，这个事务方法应该如何进行。

例如：methodA事务方法调用methodB事务方法时，methodB是继续在调用者methodA的事务中运行呢，还是为自己开启一个新事务运行，这就是由methodB的事务传播行为决定的。

|  |  |
| --- | --- |
| **事务传播行为类型** | **说明** |
| **PROPAGATION\_REQUIRED** | **如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是最常见的选择。** |
| **PROPAGATION\_SUPPORTS** | **支持当前事务，如果当前没有事务，就以非事务方式执行。** |
| **PROPAGATION\_MANDATORY** | **使用当前的事务，如果当前没有事务，就抛出异常。** |
| **PROPAGATION\_REQUIRES\_NEW** | **新建事务，如果当前存在事务，把当前事务挂起。** |
| **PROPAGATION\_NOT\_SUPPORTED** | **以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。** |
| **PROPAGATION\_NEVER** | **以非事务方式执行，如果当前存在事务，则抛出异常。** |
| **PROPAGATION\_NESTED** | **如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。** |

▶**Spring事务三要素**

数据源：表示具体的事务性资源，是事务的真正处理者，如MySQL等。

事务管理器：像一个大管家，从整体上管理事务的处理过程，如打开、提交、回滚等。

事务应用和属性配置：像一个标识符，表明哪些方法要参与事务，如何参与事务，以及一些相关属性如隔离级别、超时时间等。

▶**Spring事务的注解配置**

把一个DataSource（如DruidDataSource）作为一个@Bean注册到Spring容器中，配置好事务性资源。

把一个@EnableTransactionManagement注解放到一个@Configuration类上，配置好事务管理器，并启用事务管理。

把一个@Transactional注解放到类上或方法上，可以设置注解的属性，表明该方法按配置好的属性参与到事务中。

▶**Spring声明式事务实现原理**

\* 配置DataSource

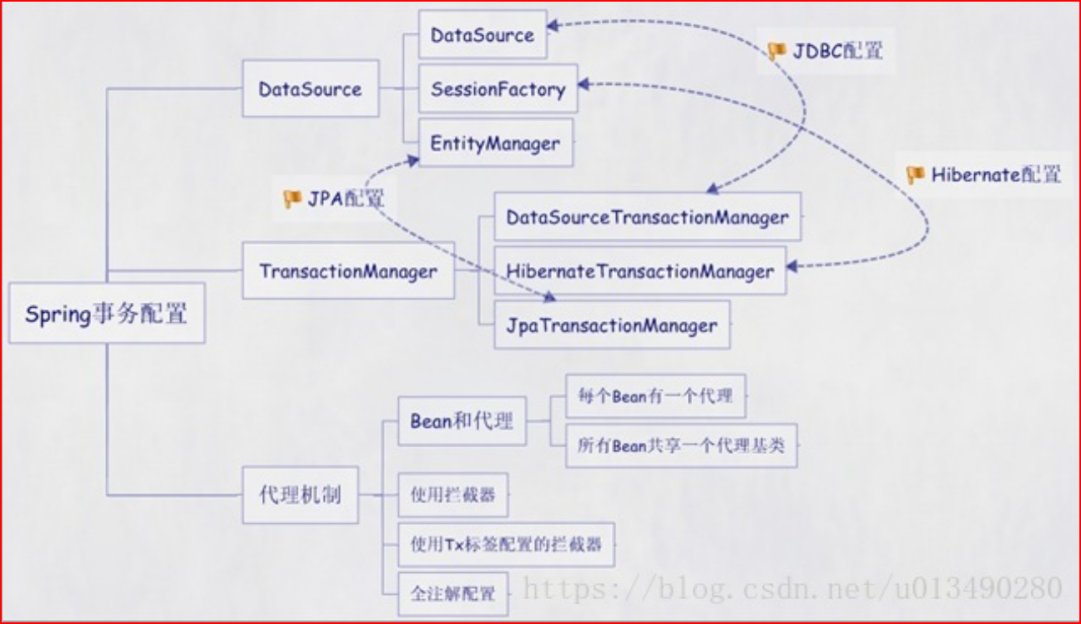
\* 配置事务管理器

\* 事务的传播特性

\* 那些类那些方法使用事务

Spring配置文件中关于事务配置总是由三个组成部分，分别是DataSource、TransactionManager和代理机制这三部分，无论哪种配置方式，一般变化的只是代理机制这部分。

    DataSource、TransactionManager这两部分只是会根据数据访问方式有所变化，比如使用Hibernate进行数据访问时，DataSource实际为SessionFactory，TransactionManager的实现为 HibernateTransactionManager。



根据代理机制的不同，Spring事务的配置又有几种不同的方式：

第一种方式：每个Bean都有一个代理

第二种方式：所有Bean共享一个代理基类

第三种方式：使用拦截器

第四种方式：使用tx标签配置的拦截器

第五种方式：全注解

▶**事务注解在类/方法上**

@Transactional注解既可以标注在类上，也可以标注在方法上。当在类上时，默认应用到类里的所有方法。如果此时方法上也标注了，则方法上的优先级高。

▶**事务注解在类上的继承性**

@Transactional注解的作用可以传播到子类，即如果父类标了子类就不用标了。但倒过来就不行了。

子类标了，并不会传到父类，所以父类方法不会有事务。父类方法需要在子类中重新声明而参与到子类上的注解，这样才会有事务。

▶**事务注解在接口/类上**

@Transactional注解可以用在接口上，也可以在类上。在接口上时，必须使用基于接口的代理才行，即JDK动态代理。

事实是Java的注解不能从接口继承，如果你使用基于类的代理，即CGLIB，或基于织入方面，即AspectJ，事务设置不会被代理和织入基础设施认出来，目标对象不会被包装到一个事务代理中。

Spring团队建议注解标注在类上而非接口上。

▶**只在public方法上生效？**

当采用代理来实现事务时，（注意是代理），@Transactional注解只能应用在public方法上。当标记在protected、private、package-visible方法上时，不会产生错误，但也不会表现出为它指定的事务配置。可以认为它作为一个普通的方法参与到一个public方法的事务中。

如果想在非public方法上生效，考虑使用AspectJ（织入方式）。

▶**目标类里的自我调用没有事务？**

在代理模式中（这是默认的），只有从外部的方法调用进入通过代理会被拦截，这意味着自我调用（实际就是，目标对象中的一个方法调用目标对象的另一个方法）在运行时不会导致一个实际的事务，即使被调用的方法标有注解。

如果你希望自我调用也使用事务来包装，考虑使用AspectJ的方式。在这种情况下，首先是没有代理。相反，目标类被织入（即它的字节码被修改）来把@Transactional加入到运行时行为，在任何种类的方法上都可以。

▶**事务与线程**

和JavaEE事务上下文一样，Spring事务和一个线程的执行相关联，底层是一个ThreadLocal<Map<Object, Object>>，就是每个线程一个map，key是DataSource，value是Connection。

▶**逻辑事务与物理事务**

事务性资源实际打开的事务就是物理事务，如数据库的Connection打开的事务。Spring会为每个@Transactional方法创建一个事务范围，可以理解为是逻辑事务。

在逻辑事务中，大范围的事务称为外围事务，小范围的事务称为内部事务，外围事务可以包含内部事务，但在逻辑上是互相独立的。每一个这样的逻辑事务范围，都能够单独地决定rollback-only状态。

那么如何处理逻辑事务和物理事务之间的关联关系呢，这就是传播特性解决的问题。

#### ▶**BeanFactory 和 ApplicationContext 有什么区别**

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。如国际化，访问资源，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，消息发送、响应机制，AOP等。  
BeanFactory在启动的时候不会去实例化Bean，中有从容器中拿Bean的时候才会去实例化。ApplicationContext在启动的时候就把所有的Bean全部实例化了。它还可以为Bean配置lazy-init=true来让Bean延迟实例化

是三种较常见的 ApplicationContext 实现方式：

1.ClassPathXmlApplicationContext：从 classpath 的 XML 配置文件中读取上下文，并生成上 下文定义

2.FileSystemXmlApplicationContext ：由文件系统中的 XML 配置文件读取上下文。

3.XmlWebApplicationContext：由 Web 应用的 XML 文件读取上下文。

4.AnnotationConfigApplicationContext(基于 Java 配置启动容器)

▶**请解释下 Spring 框架中的 IOC**

Spring 中的 org.springframework.beans 包和 org.springframework.context 包 构成了 Spring 框架 IoC 容器的基础。 BeanFactory 接口提供了一个先进的配置机制，使得任何类型的对象的配置成为可能。 ApplicationContex 接口对 BeanFactory（是一个子接口）进行了扩展，在 BeanFactory 的基础上添加了其他功能，比如与 Spring 的 AOP 更容易集成，也提供了处理 message resource 的机制（用于国际化）、事件传播以及应用层的特别配置，比如针对 Web 应用的 WebApplicationContext。 org.springframework.beans.factory.BeanFactory 是 Spring IoC 容器的具体实现， 用来包装和管理前面提到的各种 bean。BeanFactory 接口是 Spring IoC 容器的核心接口。

IOC:把对象的创建、初始化、销毁交给 spring 来管理，而不是由开发者控制，实现控制反转。

▶**Spring IOC容器的加载过程**

1. 预刷新容器：比如设定容器开启时间，标记容器已启动状态等等
2. 配置解析BeanDefinition，注册到BeanFactory（DefaultListableBeanFactory->beanDefinitionMap）
3. 设置BeanFactory的类加载器，添加几个 BeanPostProcessor
4. 调用BeanFactoryPostProcessor各个实现类的 postProcessBeanFactory方法
5. 注册BeanPostProcessor的实现类
6. 初始化当前ApplicationContext的MessageSource
7. 初始化当前ApplicationContext的事件广播器
8. 初始化一些特殊的 Bean
9. 注册事件监听器
10. 初始化所有的singleton beans（lazy-init 的除外）
11. 广播事件，ApplicationContext 初始化完成

▶**请举例说明如何在 Spring 中注入一个 Java Collection？**

Spring 提供了以下四种集合类的配置元素： 

<list>该标签用来装配可重复的 list 值。 

<set>该标签用来装配没有重复的 set 值。

<Map>: 该标签可用来注入键和值可以为任何类型的键值对。

<props>: 该标签支持注入键和值都是字符串类型的键值对。

示例：



# ▶**spring bean自动装配**

<bean>元素的autowire属性负责自动装配<bean>标签定义Javabean的属性。这样做可以省去很多配置Javabean属性的标签代码，使代码整洁、美观。

示例：



在 Spring 框架中共有 5 种自动装配

1. no：这是 Spring 框架的默认设置，在该设置下自动装配是关闭的，开发者需要自行在 bean 定义中 用标签明确的设置依赖关系。
2. byName：该选项可以根据 bean 名称设置依赖关系。当向一个 bean 中自动装配一个属性时，容器 将根据 bean 的名称自动在在配置文件中查询一个匹配的 bean。如果找到的话，就装配这个属性， 如果没找到的话就报错。
3. byType：该选项可以根据 bean 类型设置依赖关系。当向一个 bean 中自动装配一个属性时，容器 将根据 bean 的类型自动在在配置文件中查询一个匹配的 bean。如果找到的话，就装配这个属性， 如果没找到的话就报错。
4. constructor：造器的自动装配和 byType 模式类似，但是仅仅适用于与有构造器相同参数的 bean， 如果在容器中没有找到与构造器参数类型一致的 bean，那么将会抛出异常。
5. autodetect：该模式自动探测使用构造器自动装配或者 byType 自动装配。首先，首先会尝试找合 适的带参数的构造器，如果找到的话就是用构造器自动装配，如果在 bean 内部没有找到相应的构造 器或者是无参构造器，容器就会自动选择 byTpe 的自动装配方式。

▶**Spring 框架中有哪些不同类型的事件**

Spring 提供了以下 5 中标准的事件：

1.上下文更新事件（ContextRefreshedEvent）：该事件会在 ApplicationContext 被初始化或者更 新时发布。也可以在调用 ConfigurableApplicationContext 接口中的 refresh()方法时被触发。

2.上下文开始事件（ContextStartedEvent）：当容器调用 ConfigurableApplicationContext 的 Start()方法开始/重新开始容器时触发该事件。

3.上下文停止事件（ContextStoppedEvent）：当容器调用 ConfigurableApplicationContext 的 Stop()方法停止容器时触发该事件。

4.上下文关闭事件（ContextClosedEvent）：当 ApplicationContext 被关闭时触发该事件。容器被 关闭时，其管理的所有单例 Bean 都被销毁。

5.请求处理事件（RequestHandledEvent）：在 Web 应用中，当一个 http 请求（request）结束触 发该事件。