2020 面试题总结——Spring 篇

1、讲讲 Spring 的加载流程。

这个有点长，，，建议大体读一遍 spring 源码。可以参考这篇文章：Spring 初始化加载流程分析

2、Spring AOP 的实现原理。

Spring AOP 的面向切面编程，是面向对象编程的一种补充，用于处理系统中分布的各个模块的横切关注点，比如说事务管理、日志、缓存等。它是使用动态代理实现的，在内存中临时为方法生成一个 AOP 对象，这个对象包含目标对象的所有方法，在特定的切点做了增强处理，并回调原来的方法。

Spring AOP 的动态代理主要有两种方式实现，JDK 动态代理和 cglib 动态代理。JDK 动态代理通过反射来接收被代理的类，但是被代理的类必须实现接口，核心是 InvocationHandler 和 Proxy 类。cglib 动态代理的类一般是没有实现接口的类，cglib 是一个代码生成的类库，可以在运行时动态生成某个类的子类，所以，CGLIB 是通过继承的方式做的动态代理，因此如果某个类被标记为 final，那么它是无法使用 CGLIB 做动态代理的。

3、讲讲 Spring 事务的传播属性。

事务就是对一系列的数据库操作（比如插入多条数据）进行统一的提交或回滚操作。如果插入成功，那么一起成功，如果中间有一条出现异常，那么回滚之前的所有操作。这样可以防止出现脏数据，防止数据库数据出现问题。

事务的传播行为，指的是当前带有事务配置的方法，需要怎么处理事务。

例如，方法可以继续在当前事务中运行，也可能开启一个新的事务，并在自己的事务中运行。

有一点需要注意，事务的传播级别，并不是数据库事务规范中的名词，而是 Spring 自身所定义的。通过事务的传播级别，Spring 才知道如何处理事务，是创建一个新的事务，还是继续使用当前的事务。

在 TransactionDefinition 接口中，定义了三类七种传播级别。如下：

```
// TransactionDefinition.java
// ========== 支持当前事务的情况 ==========
/**
* 如果当前存在事务，则使用该事务。
* 如果当前没有事务，则创建一个新的事务。
*/
int PROPAGATION_REQUIRED = 0;
/**
* 如果当前存在事务，则使用该事务。
* 如果当前没有事务，则以非事务的方式继续运行。
```

```
*/
int PROPAGATION_SUPPORTS = 1;
/**
* 如果当前存在事务，则使用该事务。
* 如果当前没有事务，则抛出异常。
*/
int PROPAGATION_MANDATORY = 2;
// ========== 不支持当前事务的情况 ==========
/**
* 创建一个新的事务。
* 如果当前存在事务，则把当前事务挂起。
*/
int PROPAGATION_REQUIRES_NEW = 3;
/**
* 以非事务方式运行。
* 如果当前存在事务，则把当前事务挂起。
*/
int PROPAGATION_NOT_SUPPORTED = 4;
/**
* 以非事务方式运行。
* 如果当前存在事务，则抛出异常。
*/
int PROPAGATION_NEVER = 5;
// ========== 其他情况 ==========
/**
* 如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行。
* 如果当前没有事务，则等价于 {@link
TransactionDefinition#PROPAGATION_REQUIRED}
*/
int PROPAGATION_NESTED = 6;
```

当然，绝大数场景下，我们是只用到 PROPAGATION_REQUIRED 传播级别的。

需要注意的是，以 PROPAGATION_NESTED 启动的事务内嵌于外部事物（如果存在外部事务的话），此时，内嵌事务并不是一个独立的事务，它依赖于外部事务的存在，只有通过外部的事务提交，才能引起内部事务的提交，嵌套的子事务不能单独提交。这点类似于 JDBC 中的保存点（SavePoint）概念，嵌套的子事务就相当于保存点的一个应用，一个事务中可以包括多个保存点，每一个嵌套子事务。另外，外部事务的回滚也会导致嵌套子事务的回滚。

参考链接：可能是最漂亮的 Spring 事务管理详解

4、Spring 如何管理事务的，怎么配置事务。

所谓事务管理，其实就是“**按照给定的事务规则来执行提交或者回滚操作**”。

Spring 提供两种类型的事务管理：

- 声明式事务，通过使用注解或基于 XML 的配置事务，从而事务管理与业务代码分离。

- 编程式事务，通过编码的方式实现事务管理，需要在代码中显示的调用事务的获得、提交、回滚。它提供了极大的灵活性，但维护起来非常困难。

实际场景下，我们一般使用 SpringBoot+注解（@Transactional）的声明式事务。Spring 的声明式事务管理建立在 AOP 基础上，其本质是在目标方法执行前进行拦截，在方法开始前创建一个事务，在执行完方法后根据执行情况提交或回滚事务。声明式事务最大的优点就是不需要通过编程的方式管理事务，这样就不用侵入业务代码，只需要在配置文件中做相关的事物声明就可将业务规则应用到业务逻辑中。和编程式事务相比，声明式事务唯一的不足是只能作用到方法级别，无法做到像编程式事务那样到代码块级别。

具体可以去看 Spring 事务管理－编程式事务、声明式事务。

5、说说你对 Spring 的理解，非单例注入的原理？它的生命周期？循环注入的原理，aop 的实现原理，说说 aop 中的几个术语，它们是怎么相互工作的。

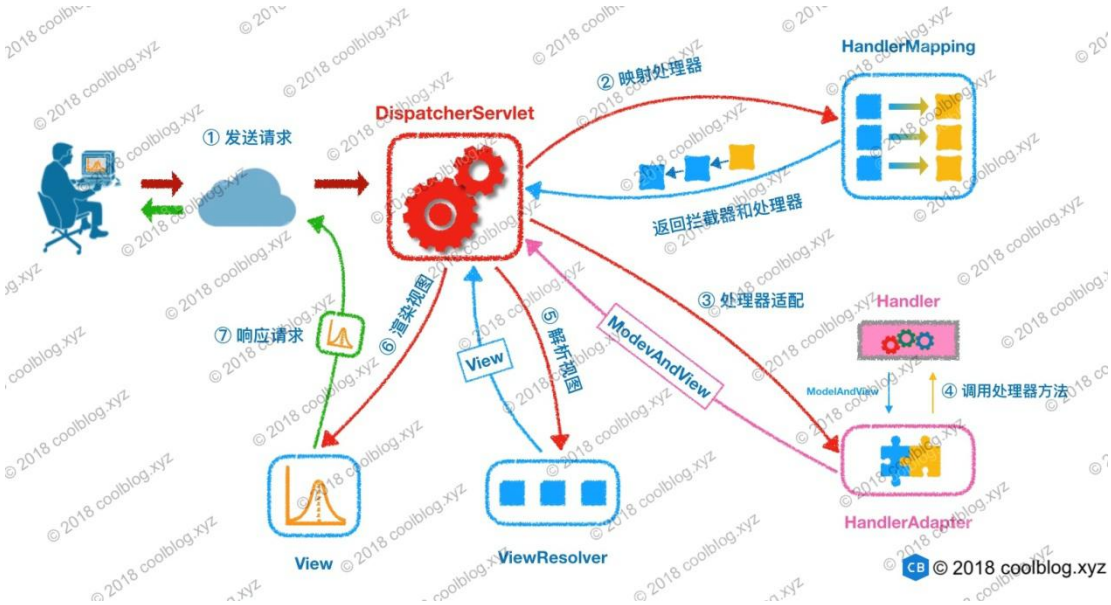
在 bean 的 scope 属性中，当值是 singleton 表示单例,当值是 prototype 表示多例。

单例：多次用 factory.getBean("user",user.class)获取实体类，获得是同一类。

多例：多次用 factory.getBean("user",user.class)获取实体类，获得是多个类。

6、Spring MVC 中 DispatcherServlet 工作流程。

DispatcherServlet 工作流程可以用一幅图来说明。



①发送请求

用户向服务器发送 HTTP 请求，请求被 Spring MVC 的调度控制器 DispatcherServlet 捕获。

②映射处理器

DispatcherServlet 根据请求 URL，调用 HandlerMapping 获得该 Handler 配置的所有相关的对象（包括 Handler 对象以及 Handler 对象对应的拦截器），最后以 HandlerExectuionChain 对象的形式返回。

③处理器适配

DispatcherServlet 根据获得 Handler，选择一个合适的 HandlerAdapter。（如果成功获得 HandlerAdapter 后，此时将开始执行拦截器的 preHandler()方法）

提取请求 Request 中的模型数据，填充 Handler 入参，开始执行 Handler（Controller）。在填充 Handler 的入参过程中，根据你的配置，Spring 将帮你做一些额外的工作：

- HttpMessageConverter：会将请求信息（如 JSON、XML 等数据）转换为一个对象。
- 数据转换：对请求消息进行数据转换，如 String 转换为 Integer、Double 等。
- 数据格式化：对请求消息进行数据格式化，如将字符串转换为格式化数字或格式化日期等。
- 数据验证：验证数据的有效性（长度、格式等），验证结果存储到 BindingResult 或 error 中。（自定义验证机制需要使用注解@InitBinder）

Handler（Controller）执行完成后，向 DispatcherServlet 返回一个 ModelAndView 对象。当然，如果判断方法中有@ResponseBody 注解，则直接将结果写回用户浏览器。

图中没有④。

⑤解析试图

根据返回的 ModelAndView，选择一个合适的 ViewResolver（必须是已经注册到 Spring 容器中的 ViewResolver），解析出 View 对象，然后返回给 DispatcherServlet。

⑥⑦渲染视图+相应请求

ViewResolver 结合 Model 和 View，来渲染视图，并写回给用户浏览器。

7、Spring MVC 用到的注解，作用是什么，原理。

在 Spring MVC 中主要用到以下注解：

- @Controller 注解，它将一个类标记为 Spring Web MVC 控制器 Controller。
- @RestController 注解，在@Controller 注解的基础上，增加了 @ResponseBody 注解，更加适合目前前后端分离的架构下，提供 Restful API，返回例如 JSON 数据格式。当然，返回什么样的数据格式，根据客户端的“ACCEPT”请求头来决定。
- @RequestMapping 注解，用户将特定的 HTTP 请求方法映射到将处理相应请求的控制器的特定类/方法。此注释可应用于两个级别，一是类级别，映射请求的 URL；另一个是方法级别，映射 URL 以及 HTTP 请求方法。
- @GetMapping 注解，相当于是@RequestMapping 的 GET 请求方法的特例，目的是为了提高清晰度。并且仅可注册在方法上。

8、Spring boot 启动机制。

Spring Boot 启动时的关键步骤，主要在两个方面：

- SpringApplication 实例的构建过程，其中主要涉及到了初始化器(Initializer)以及监听器(Listener)这两大概念，它们都通过 META-INF/spring.factories 完成定义。
- SpringApplication 实例 run 方法的执行过程，其中主要有一个 SpringApplicationRunListeners 的概念，它作为 Spring Boot 容器初始化时各阶段事件的中转器，将事件派发给感兴趣的 Listeners(在 SpringApplication 实例的构建过程中得到的)。

强烈建议直接去看这篇文章：[Spring Boot] 1. Spring Boot 启动过程源码分析

注意，这篇文章没有讲关于 IOC 的东西。这块需要自己补充。

9、Spring 中用到的设计模式。

Spring 在设计中用了几种常用的设计模式。

a，工厂模式

在 Spring 中我们一般是将 Bean 的实例化直接交给容器去管理的，实现了使用和创建的分离，这时容器直接管理对象，还有种情况是，bean 的创建过程我们交给一个工厂去实现，而 Spring 容器管理这个工厂。Spring 使用工厂模式可以通过 BeanFactory 或 ApplicationContext 创建 bean 对象。两者对比如下：

- BeanFactory：延迟注入（使用到某个 bean 的时候才会注入），相比于 ApplicationContext 来说会占用更少的内存，程序启动速度更快。
- ApplicationContext：容器启动的时候，不管 bean 是否用到，一次性创建所有的 bean。BeanFactory 仅提供了最基本的依赖注入支持，

ApplicationContext 扩展了 BeanFactory ,除了有 BeanFactory 的功能还有额外更多功能，所以一般开发人员使用 ApplicationContext 会更多。

b，单例设计模式

在系统中，有一些对象其实我们只需要一个，比如说：线程池、缓存、对话框、注册表、日志对象等。事实上，这一类对象只能有一个实例，如果制造出多个实例就可能导致一些问题的产生，比如：程序的行为异常、资源使用量、或者不一致性的结果。

使用单例模式的好处：

- 对于频繁使用的对象，可以省略创建对象所花费的时间，这对于那些重量级对象而言，是非常可观的一笔系统开销。
- 由于 new 操作的次数减少，因而对系统内存的使用频率也会降低，这将减轻 GC 压力，缩短 GC 停顿时间。

Spring 中 bean 默认作用域是 singleton（单例），除了 singleton 作用域外，Spring 中 bean 还有下面几种作用域。

- prototype：每次请求都会创建一个新的实例。
- 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。
- 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。

Spring boot 实现单例注册表的核心代码（DefaultSingletonBeanRegistry 类）如下：

```
//通过 ConcurrentMap（线程安全）实现单例注册表 注意这行！！！
private final Map<String, Object> singletonObjects = new
ConcurrentHashMap<>(256);

public Object getSingleton(String beanName, ObjectFactory<?>
singletonFactory) {
   Assert.notNull(beanName, "Bean name must not be null");
   synchronized (this.singletonObjects) {

      //检查缓存中是否存在实例
      Object singletonObject = this.singletonObjects.get(beanName);
      if (singletonObject == null) {
         //省略部分代码
         try {
            singletonObject = singletonFactory.getObject();
            newSingleton = true;
         }
         // 省略部分代码
         // 如果实例对象在不存在，我们注册到单例注册表中
         if (newSingleton) {
```

```
                addSingleton(beanName, singletonObject);
            }
        }
        return singletonObject;
    }
}
//将对象添加到单例注册表中
protected void addSingleton(String beanName, Object singletonObject) {
    synchronized (this.singletonObjects) {
        this.singletonObjects.put(beanName, singletonObject);
        this.singletonFactories.remove(beanName);
        this.earlySingletonObjects.remove(beanName);
        this.registeredSingletons.add(beanName);
    }
}
```

c，代理模式

AOP(Aspect-Oriented Programming:面向切面编程)能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。

Spring AOP 就是基于动态代理的，其中有两种不同的代理方法：JDK 代理和 Cglib 代理。具体可以看之前文章。

当然，也可以使用 AspectJ，Spring AOP 已经集成了 AspectJ。使用 AOP 之后我们可以把一些通用功能抽象出来，在需要用到的地方直接使用即可，这样大大简化了代码量。我们需要增加新功能时也方便，这样也提高了系统扩展性。日志功能、事务管理等等场景都用到了 AOP 。

另外，Spring AOP 属于运行时增强，而 Aspect J 是编译时增强。Spring AOP 基于代理来实现，而 Aspect J 是基于字节码操作。

d，模板方法

模板方法是一种行为设计模式，它定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。 模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤的实现方式。Spring 中 jdbcTemplate、hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。一般情况下，我们都是使用继承的方式来实现模板模式，但是 Spring 并没有使用这种方式，而是使用 Callback 模式与模板方法模式配合，既达到了代码复用的效果，同时增加了灵活性。

e，观察者模式

观察者模式是一种对象行为型模式。它表示的是一种对象与对象之间具有依赖关系，当一个对象发生改变的时候，这个对象所依赖的对象也会做出反应。Spring 事件驱动模型就是观察者模式很经典的一个应用。Spring 事件驱动模型

非常有用，在很多场景都可以解耦我们的代码。比如我们每次添加商品的时候都需要重新更新商品索引，这个时候就可以利用观察者模式来解决这个问题。

Spring 中 Observer 模式常用的地方是 listener 的实现。如 ApplicationListener。

f，适配器模式

适配器模式(Adapter Pattern) 将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为包装器(Wrapper)。

我们知道 Spring AOP 的实现是基于代理模式，但是 Spring AOP 的增强或通知(Advice)使用到了适配器模式，与之相关的接口是 AdvisorAdapter 。Advice 常用的类型有：BeforeAdvice（目标方法调用前,前置通知）、AfterAdvice（目标方法调用后,后置通知）、AfterReturningAdvice(目标方法执行结束后，return 之前)等等。每个类型 Advice（通知）都有对应的拦截器:MethodBeforeAdviceInterceptor、AfterReturningAdviceAdapter、AfterReturningAdviceInterceptor。Spring 预定义的通知要通过对应的适配器，适配成 MethodInterceptor 接口(方法拦截器)类型的对象（如：MethodBeforeAdviceInterceptor 负责适配 MethodBeforeAdvice）。

在 Spring MVC 中，DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。解析到对应的 Handler（也就是我们平常说的 Controller 控制器）后，开始由 HandlerAdapter 适配器处理。HandlerAdapter 作为期望接口，具体的适配器实现类用于对目标类进行适配，Controller 作为需要适配的类。

为什么要在 Spring MVC 中使用适配器模式？ Spring MVC 中的 Controller 种类众多，不同类型的 Controller 通过不同的方法来对请求进行处理。如果不利用适配器模式的话，DispatcherServlet 直接获取对应类型的 Controller，需要的自行来判断，像下面这段代码一样：

```
if(mappedHandler.getHandler() instanceof MultiActionController){
   ((MultiActionController)mappedHandler.getHandler()).xxx
}else if(mappedHandler.getHandler() instanceof XXX){
    ...
}else if(...){
   ...
}
```

假如我们再增加一个 Controller 类型就要在上面代码中再加入一行 判断语句，这种形式就使得程序难以维护，也违反了设计模式中的开闭原则 – 对扩展开放，对修改关闭。具体可以参考 SpringMVC 中的适配器（适配者模式）

g，策略模式

策略模式对应于解决某一个问题的一个算法族，允许用户从该算法族中任选一个算法解决某一问题，同时可以方便的更换算法或者增加新的算法。并且由客户端决定调用哪个算法，spring 中在实例化对象的时候用到 Strategy 模式。

总结如下：

Spring 框架中用到了哪些设计模式？

- 工厂设计模式 : Spring 使用工厂模式通过 BeanFactory、ApplicationContext 创建 bean 对象。
- 代理设计模式 : Spring AOP 功能的实现。
- 单例设计模式 : Spring 中的 Bean 默认都是单例的。
- 模板方法模式 : Spring 中 jdbcTemplate、hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。
- 包装器设计模式 : 我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。
- 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。
- 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。
- ……

参考文章：Spring 用到的设计模式，你都知道吗？ 和 面试官:“谈谈 Spring 中都用到了那些设计模式?”

10、一个 HTTP 请求就是一个线程吗。

一个 HTTP 请求就是一个线程。tomcat 会维护一个线程池，每一个 http 请求，会从线程池中取出一个空闲线程。

在 Tomcat 中 server.xml 中连接器设置如下：

```
<Connector port="8080"
                   maxThreads="150" minSpareThreads="25"
maxSpareThreads="75"
                   enableLookups="false" redirectPort="8443"
acceptCount="100"
                   debug="0" connectionTimeout="20000"
                   disableUploadTimeout="true" />
```

相关参数表示为：

- minProcessors：最小空闲连接线程数，用于提高系统处理性能，默认值为 10，服务器启动时创建的处理请求的线程数。
- maxProcessors：最大连接线程数，即：并发处理的最大请求数，默认值为 75。
- acceptCount：指定当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理。

- connectionTimeout：一个请求最多等待时间，超过则报错。举个例子，现在这个值设置为 20000，则代表请求建立一个 socket 连接后，如果一直没有收到客户端的 FIN，也没有数据过来，那么此连接也必须等到 20s 后，才能被超时释放。

这三个值的具体使用场景如下：

- 情况 1：Tomcat 启动时，会按照 minProcessors 数目创建线程。
- 情况 2：接受一个请求，此时 tomcat 启动的线程数目没有到达 maxThreads，tomcat 会启动一个线程来处理此请求。
- 情况 3：接受一个请求，此时 tomcat 启动的线程数已经到达 maxThreads，tomcat 会把此请求放入等待队列中，等待空闲线程。
- 情况 4：接受一个请求，此时 tomcat 启动的线程数已经到达 maxThreads，等待队列中的请求个数也达到了 acceptCount，此时 tomcat 会直接拒绝此次请求，返回 connect refused。

参考链接：一个 http 请求就是一个线程吗，java 的服务是每收到一个请求就新开一个线程来处理吗

11、SpringMVC 和 Structs2 的区别。

- Structs2 是类级别的拦截，一个类对应一个 request 上下文；而 Spring MVC 是方法级别的拦截，一个方法对应一个 request 上下文，而方法同时又跟一个 url 对应。
- 在拦截机制上，Structs2 有自己定义的 Interceptor 机制，而 Spring MVC 是独立的 AOP 方式。
- SpringMCV 几乎实现了零配置，在配置了基本的东西之后，再编写 controller 类和方式时，只需要加上注解即可，无需频繁地配置文件，而 Structs2 的机制是无法使用注解开发，那就需要重新编写一个 action 类配置一遍，非常繁琐。
- 在于 Ajax 的集成方面，Spring MVC 只需要一个注解@ResponseBody 就可在返回之中返回数据，SpringMVC 会自动将返回值数据转换为 json 数据，非常方便；而 Structs2 则需要自己手动将返回值数据转换成 json 格式，再手动写回浏览器，比较麻烦。

12、Mybaits 框架的优缺点。

优点如下：

- 与 JDBC 相比，减少了 50%以上的代码量。
- MyBatis 是最简单的持久化框架，小巧并且简单易学。

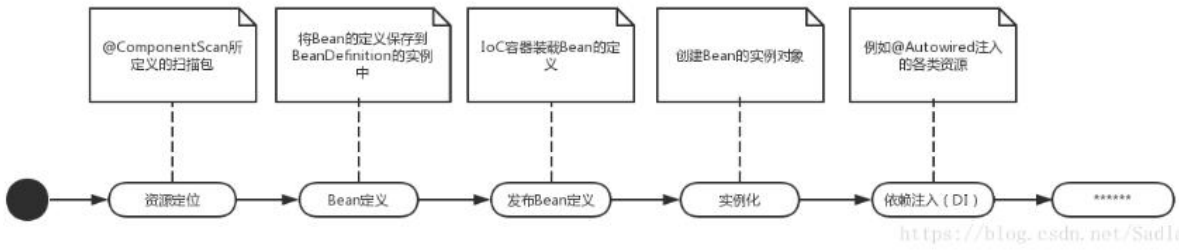
- MyBatis 灵活，不会对应用程序或者数据库的现有设计强加任何影响，SQL 写在 XML 里，从程序代码中彻底分离，降低了耦合度，便于统一管理和优化，可以重用。
- 提供 XML 标签，支持编写动态 SQL 语句（XML 中使用 if，else）。
- 提供映射标签，支持对象与数据库的 ORM 字段关系映射（在 XML 中配置映射关系，也可以使用注解）。

缺点如下：

- SQL 语句的编写工作量大，尤其是字段多、关联表多时，更是如此，对开发人员编写 SQL 语句的功底有一定要求。
- SQL 语句依赖数据库，导致数据库移植性差，不能随意更换数据库。

13、Spring 中 Bean 的生命周期。

首先来看 bean 初始化之前的准备工作，如下图。



主要看前三步：

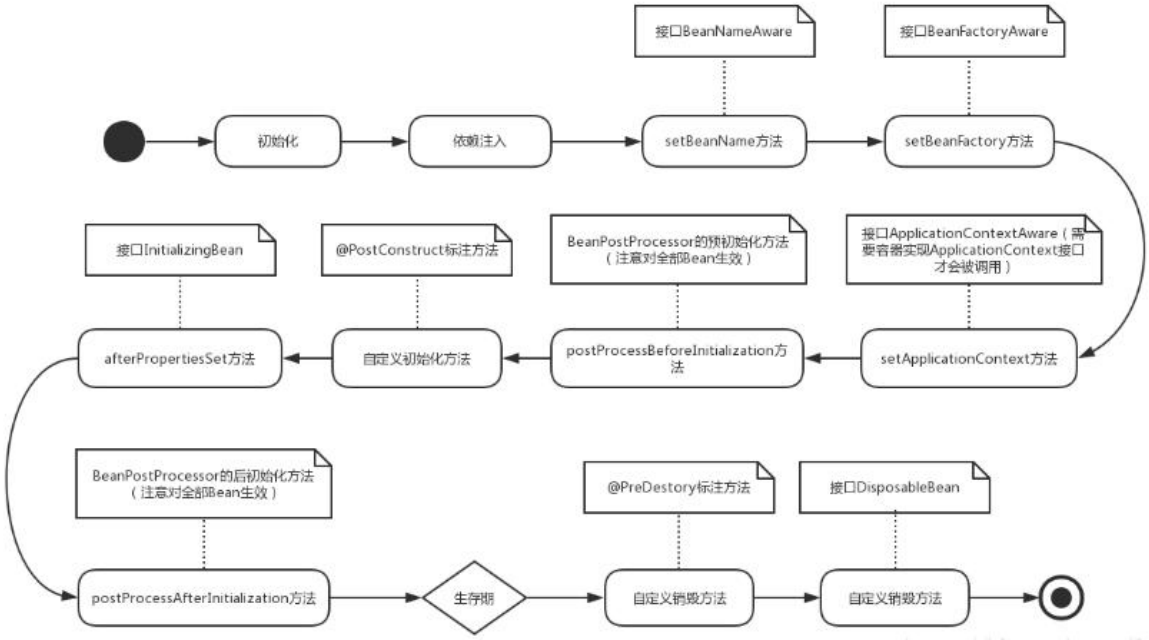
spring 通过我们的配置，比如@ComponentScan 定义的扫描路径去找到带有@Component 的类，这个过程就是一个资源定位的过程。

一旦找到资源，那么它就开始解析，并将定义的信息保存到 beanDefinition 中。注意，此时还没有初始化 bean，也没有 bean 实例，它有的仅仅是 bean 的定义。

然后把 bean 定义发布到 Spring IOC 容器中给，此时，IOC 容器也只有 bean 的定义，还是没有 bean 的实例生成。

对于 IOC 容器 BeanFactory 来说，它遵循延迟注入原则，只有当用到某个 bean 的时候才会初始化（就是只有当 getBean()调用的时候才会触发 bean 实例化）。而对于 ApplicationContext 容器来说，它是在系统启动的时候就完成了所有 bean 的初始化。

下面来看 IOC 容器初始化 bean 的过程，如下图。



简单来说就以下几步：

- 如果 Bean 实现了 BeanNameAware 接口，工厂调用 Bean 的 setBeanName()方法传递 Bean 的 ID。
- 如果 Bean 实现了 BeanFactoryAware 接口，工厂调用 setBeanFactory()方法传入工厂自身。
- 如果 Bean 实现了 ApplicationContextAware 接口，则可以调用 setApplicationContext 获取 ApplicationContext。
- 将 Bean 实例传递给 Bean 的前置处理器的 postProcessBeforeInitialization(Object bean, String beanname)方法。调用 Bean 的初始化方法。
- 将 Bean 实例传递给 Bean 的后置处理器的 postProcessAfterInitialization(Object bean, String beanname)方法。Bean 使用过程。
- 如果 Bean 实现了 DisposableBean 接口，则可以调用其 destory()方法，也或者可以通过重写@PreDestroy 方法在关闭时释放一些资源。

14、Spring 中事务失效的几种原因。

Spring 中通过注解@Transactional 来实现事务，但在以下几种情况时，事务会失效。

- Spring 中事务自调用会失效，如果 A 方法调用 B 方法，B 方法上有事务注解，AB 方法在同一个类中，那么 B 方法的事务就会失效，这是动态代理的原因。

- Spring 的事务注解@Transactional 只能作用在 public 修饰的方法上才起作用，如果放在非 public（如 private、protected）方法上，虽然不报错，但是事务不起作用。
- 如果 MySQL 用的引擎是 MyISAM，则事务会不起作用，原因是 MyISAM 不支持事务，可以改成 InnoDB 引擎。
- Spring 建议在具体的类（或类的方法）上使用@Transactional 注解，而不要使用在类所实现的任何接口上。在接口上使用@Transactional 注解，只能当你设置了基于接口的代理时他才会生效，因为注解是不能继承的，这就意味着如果正在使用基于类的代理时，那么事务的设置将不能被基于类的代理所识别，而且对象也将不会被事务代理所包装。
- 如果在业务代码中抛出 RuntimeException 异常，事务回滚；但是抛出 Exception，事务不回滚。

需要注意的是，@Transactional 也可以作用于类上，放在类级别上等同于该类的每个公有方法都放上了@Transactional。

15、注解继承问题。

对于使用元注解@Inherited 修饰的自定义注解，作用在父类上的自定义注解可以被继承下来。作用在接口上自定义注解不能被实现它的类继承下来。

需要注意的是：使用 Inherited 声明出来的注解，只有在类上使用时才会有效，对方法，属性等其他无效，并且也对于接口无效。简单一点来说吧，就是写在父类的类上的注解可以被继承，卸载父类方法上的注解无法被继承！

所以一般来说，我们应当在实现方法上打注解。

回到上一题说的事务注解@Transactional，它也有@Inherited 修饰，所以对它来说，如果父类级别使用@Transactional，那子类会继承；如果父类方法级别使用@Transactional，那么子类方法不会继承；如果接口实现@Transactional，子类不会继承。

16、MyBatis 的分页。

MyBatis 实现分页操作时，有逻辑分页和物理分页这两个区别。

- 逻辑分页：将数据一次性从数据库查出到内存中，在内存中进行逻辑上的分页。在实际应用方面，Mybaits 自带分页 RowBounds 可以实现逻辑分页。
- 物理分页：直接特定的 SQL 语句，只从数据库中查询出需要的数据。在实际应用方面，Mybatis 可以通过自写 SQL 或者通过分页插件 PageHelper 来实现物理分页。

逻辑分页内存开销比较大，物理分页内存开销比较小。在数据量比较小的情况下，逻辑分页效率比物理分页高；在数据量很大的情况下，建议使用物理分页，因为物理内存开销太大，容易造成内存溢出。

我习惯在项目中使用 PageHelper 来实现分页。