框架知识Anki

简述下自己对于 Spring IoC 和 AOP 理解

- IoC（Inverse of Control:控制反转）是一种**设计思想**，就是 **将原本在程序中手动创建对象的控制权，交由Spring框架来管理。** IoC 在其他语言中也有应用，并非 Spring 特有。 **IoC 容器是 Spring 用来实现 IoC 的载体， IoC 容器实际上就是个Map（key，value）,Map 中存放的是各种对象。**
- 将对象之间的相互依赖关系交给 IoC 容器来管理，并由 IoC 容器完成对象的注入。这样可以很大程度上简化应用的开发，把应用从复杂的依赖关系中解放出来。 **IoC 容器就像是一个工厂一样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的。**
- 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**，如果要代理的对象，实现了某个接口，那么Spring AOP会使用**JDK Proxy**，去创建代理对象，而对于没有实现接口的对象，就无法使用 JDK Proxy 去进行代理了，这时候Spring AOP会使用**Cglib** ，这时候Spring AOP会使用 **Cglib** 生成一个被代理对象的子类来作为代理

Spring 中的 bean 的作用域有哪些

- singleton : 唯一 bean 实例，Spring 中的 bean 默认都是单例的。
- prototype : 每次请求都会创建一个新的 bean 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。
- global-session： 全局session作用域，仅仅在基于portlet的web应用中才有意义，Spring5已经没有了。Portlet是能够生成语义代码(例如：HTML)片段的小型Java Web插件。它们基于portlet容器，可以像servlet一样处理HTTP请求。但是，与 servlet 不同，每个 portlet 都有不同的会话

Spring 中的单例 bean 的线程安全问题了解吗

当多个线程操作同一个对象的时候，**对这个对象的成员变量的写操作会存在线程安全问题。**

但是，一般情况下，我们常用的 `Controller`、`Service`、`Dao` 这些 Bean 是无状态的。无状态的 Bean 不能保存数据，因此是线程安全的。

常见的有 2 种解决办法：

1. **在类中定义一个 `ThreadLocal` 成员变量**，将需要的可变成员变量保存在 `ThreadLocal` 中（推荐的一种方式）。
2. **改变 Bean 的作用域为 “prototype”**：每次请求都会创建一个新的 bean 实例，自然不会存在线程安全问题。

`@Bean` 注解比 `Component` 注解的自定义性更强

下面这个例子是通过 `@Component` 无法实现的。

```
@Bean
public OneService getService(status) {
    case (status)  {
        when 1:
                return new serviceImpl1();
        when 2:
                return new serviceImpl2();
        when 3:
                return new serviceImpl3();
    }
}
```

Spring bean对象的生命周期

- 单例对象
  - 出生:**当容器创建时对象出生**
  - 活着:只要容器还在,对象一直活着
  - 死亡:容器销毁，对象消亡
  - 总结:**单例对象的生命周期和容器相同**
- 多例对象
  - 出生:当我们使用对象时spring框架为我们创建
  - 活着:对象只要是在使用过程中就一直活着。
  - 死亡:**当对象长时间不用，且没有别的对象引用时，由Java的垃圾回收器回收**
- Bean 的基本类型分为 **singleton（单例） 和 prototype（原型/多例）** 两种，在容器创建过程中，单例 Bean 默认跟随容器一起实例化，而当我们指定 Bean节点的 lazy-init=”true” 时，只有在第一次获取 Bean 的时候才会初始化 Bean。当然，如果想让所有单例 Bean 都延迟加载，可以在根节点设置此属性。

  当 scope="prototype" 时，容器也会延迟初始化 Bean，并不会立刻创建对象，而是在第一次请求该 bean 时才初始化（如调用 getBean 方法时）。和单例不同的情况是：在对象销毁时，容器不会帮我们调用任何方法。

  Spring不能对一个 prototype bean 的整个生命周期负责：容器在初始化、配置、装饰或者是装配完一个 prototype 实例后，将它交给客户端，随后就对该prototype 实例不闻不问了。
- 有深入的分析如下，找时间研究下

Bean 的生命周期

- Bean容器/BeanFactory 通过对象的构造器或工厂方法先实例化 Bean；
- 再根据 Resource 中的信息再通过设定好的方法（典型的有setter，统称为BeanWrapper）对 Bean 设置属性值，得到 BeanDefintion 对象，然后 put 到 beanDefinitionMap 中，调用 getBean 的时候，从 beanDefinitionMap 里拿出 Class 对象进行注入（**使用了反射**），同时如果有依赖关系，将递归调用 getBean 方法，即依赖注入的过程。
- 检查 xxxAware 相关接口，比如 BeanNameAware，BeanClassLoaderAware，ApplicationContextAware（ BeanFactoryAware）等等，如果有就调用相应的 setxxx 方法把所需要的xxx传入到 Bean 中。

补充：关于 Aware ，Aware 就是感知的意思， Aware 的目的是为了让Bean获得Spring容器的服务。 实现了这类接口的 bean 会存在“意识感”，从而让容器调用 setxxx 方法把所需要的 xxx 传到 Bean 中。

- 此时检查是否存在有于 Bean 关联的任何 BeanPostProcessors， 执行 postProcessBeforeInitialization() 方法（前置处理器）。
- 如果 Bean 实现了InitializingBean接口（正在初始化的 Bean），执行 afterPropertiesSet() 方法。
- 检查是否配置了自定义的 init-method 方法，如果有就调用。
- 此时检查是否存在有于 Bean 关联的任何 BeanPostProcessors， 执行 postProcessAfterInitialization() 方法（后置处理器）。返回 wrapperBean（包装后的 Bean）。
- 这时就可以开始使用 Bean 了，当容器关闭时，会检查 Bean 是否实现了 DisposableBean 接口，如果有就调用 destory() 方法。
- 如果 Bean 配置文件中的定义包含 destroy-method 属性，执行指定的方法。

ApplicationContext和BeanFactory的区别

```
public interface ApplicationContext extends EnvironmentCapable,
ListableBeanFactory, HierarchicalBeanFactory,
    MessageSource, ApplicationEventPublisher, ResourcePatternResolver {}
```

- BeanFactory 粗暴简单，可以理解为就是个 HashMap，Key 是 BeanName，Value 是 Bean 实例。通常只提供实例化对象 和获取这两个功能。BeanFactory在启动的时候不会去实例化Bean，只有从容器中拿Bean的时候才会去实例化。
- ApplicationContext（**应用上下文**） 可以称之为 “高级容器”。因为**他比 BeanFactory 多了更多的功能，继承了多个接口，因此具备了更多的功能。** 如国际化，访问资源，载入多个（有继承关系）上下文 ，消息发送、响应机制，AOP等。

循环依赖是什么

bean A依赖于另一个bean B时，bean B依赖于bean A：

- 当Spring上下文加载所有bean时，它会尝试按照它们完全工作所需的顺序创建bean。例如，如果我们没有循环依赖，如下例所示：

豆A→豆B→豆C.

Spring将创建bean C，然后创建bean B（并将bean注入其中），然后创建bean A（并将bean B注入其中）。

- 但是，当具有循环依赖时，Spring无法决定应该首先创建哪个bean，因为它们彼此依赖。

以setter方式构成的循环依赖，spring可以帮你解决，以构造器方式构成的循环依赖，spring无法搞定。

setter 注入和构造器注入的区别就在于创建bean的过程中，setter注入可以先用无参数构造方法返回bean实例，再注入依赖的属性，使用到了 Spring 的三级缓存，而constructor方式**无法先返回bean的实例，必须先实例化它所依赖的属性，这样一来就会造成死循环所以会失败。**

我们使用比较多的在属性上@Autowired的方式，在spring内部的处理中，与setter方法不太一样，但用此种方式循环依赖也可以解决，原理同上，只要能先实例化出来，提前暴露出来，就可以解决循环依赖的问题。

Spring 如何解决循环依赖以及循环依赖的场景有哪些

Spring使用了三级缓存解决了循环依赖的问题。在populateBean()给属性赋值阶段里面Spring会解析你的属性，并且赋值，当发现，A对象里面依赖了B，此时又会走getBean方法，但这个时候，你去缓存中是可以拿的到的。因为我们在对createBeanInstance对象创建完成以后已经放入了缓存当中，所以创建B的时候发现依赖A，直接就从缓存中去拿，此时B创建完，A也创建完。至此Bean的创建完成，最后将创建好的Bean放入单例缓存池中。

Bean 的创建最为核心三个方法解释如下：

- `createBeanInstance`：例化，其实也就是调用对象的**构造方法**实例化对象
- `populateBean`：填充属性，这一步主要是对bean的依赖属性进行注入(`@Autowired`)
- `initializeBean`：回到一些形如`initMethod`、`InitializingBean`等方法

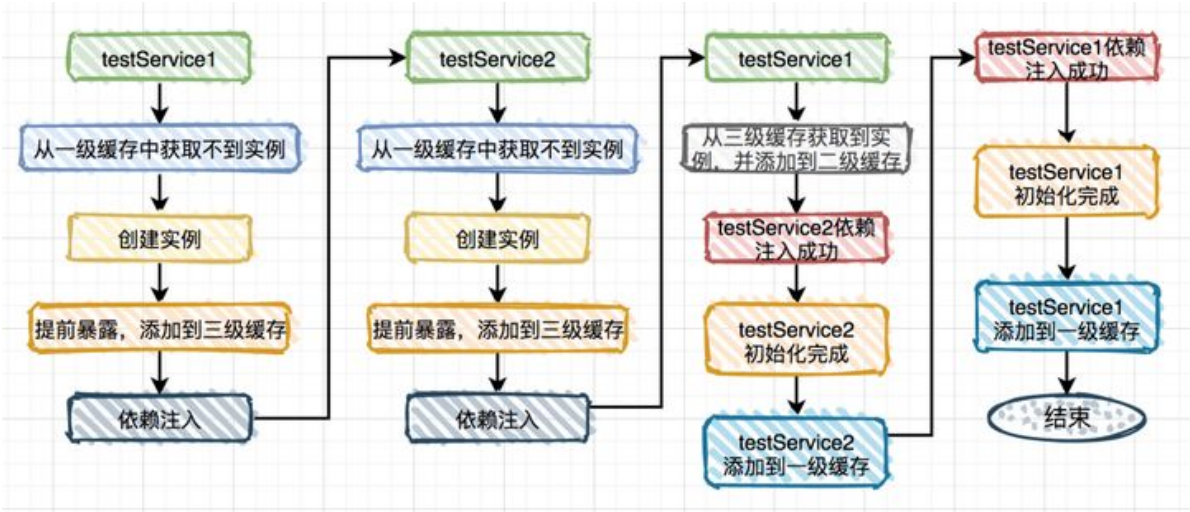
现在先了解一下三级缓存：

1. `singletonObjects`：第一级，单例缓存池。用于存放完全初始化好的 bean，**从该缓存中取出的 bean 可以直接使用**
2. `earlySingletonObjects`：第二级。提前曝光的单例对象的cache，存放原始的 bean 对象（尚未填充属性的 bean）
3. `singletonFactories`：第三级，单例对象工厂缓存 。单例对象工厂的cache，存放 bean 工厂对象

了解完缓存就可以开始了解单例 Bean 的创建过程：

1. 先从一级缓存singletonObjects中去获取。（如果获取到就直接return）
2. 如果获取不到或者对象正在创建中（isSingletonCurrentlyInCreation()），那就再从二级缓存earlySingletonObjects中获取。（如果获取到就直接return）
3. 如果还是获取不到，且允许singletonFactories（allowEarlyReference=true）通过getObject()获取。就从三级缓存singletonFactory.getObject()获取。**（如果获取到了就把这个 bean 从 singletonFactories 中移除，并且放进 earlySingletonObjects。其实也就是从三级缓存移动（是剪切）到了二级缓存）**

下面用一张图告诉你，spring是如何解决循环依赖的：



图来自：https://www.zhihu.com/question/438247718/answer/1730527725（这篇循环依赖写得非常好，以后要复习循环依赖看这篇）

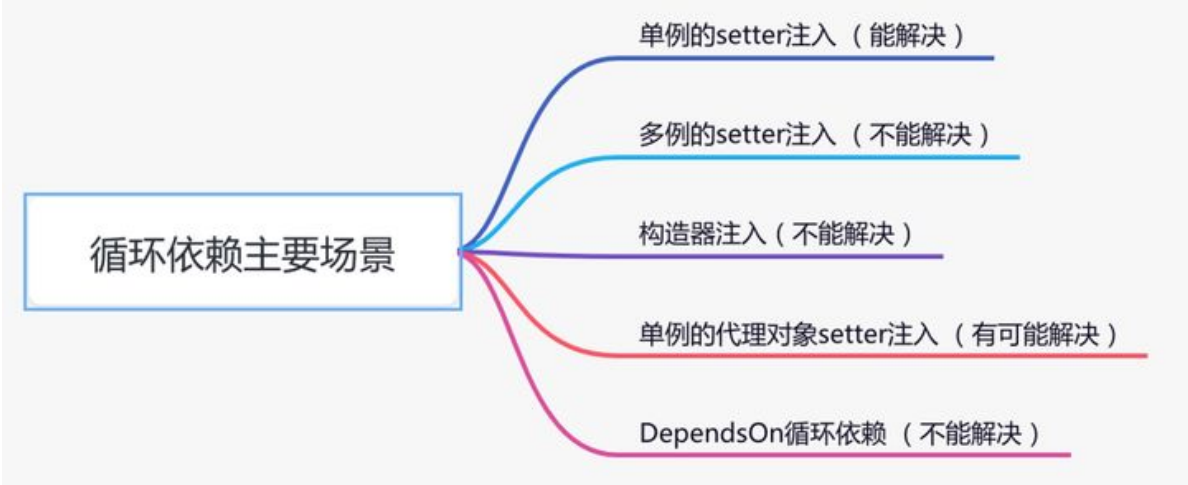
- 关于二级缓存的作用上面这篇文章也写了，想不去来就去看下这个

> TestService1注入到TestService3又需要从第三级缓存中获取实例，而第三级缓存里保存的并非真正的实例对象，而是`ObjectFactory`对象。说白了，**两次从三级缓存中获取都是`ObjectFactory`对象，而通过它创建的实例对象每次可能都不一样的。**

> 这样不是有问题?
>
> **为了解决这个问题，spring引入的第二级缓存。**上面图1其实TestService1对象的实例已经被添加到第二级缓存中了，而在TestService1注入到TestService3时，只用从第二级缓存中获取该对象即可。

循环依赖的N种场景中为什么构造器注入这些不能解决

spring中出现循环依赖主要有以下场景：



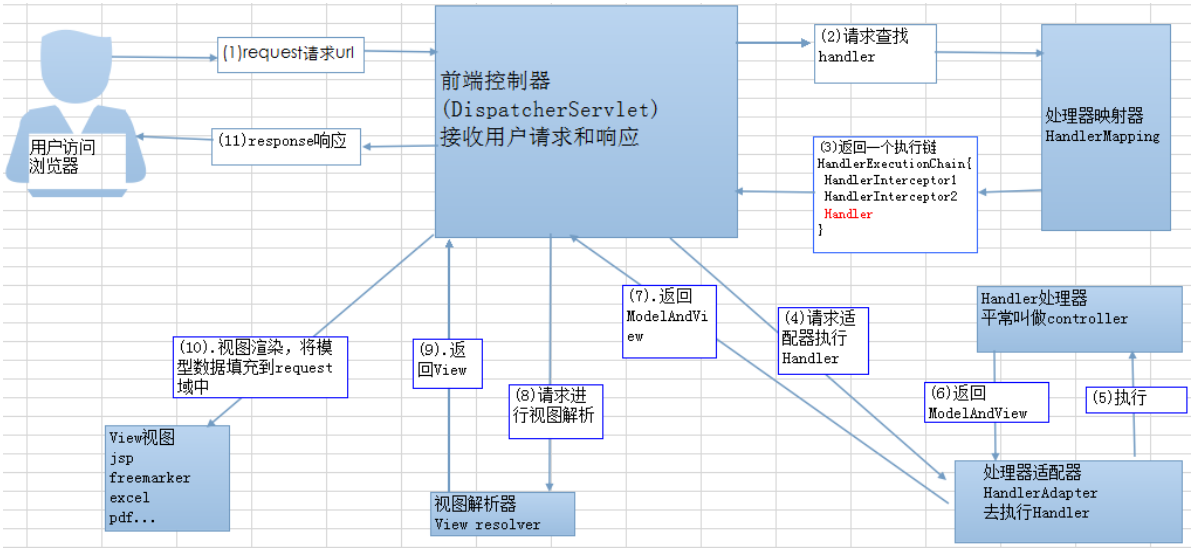
- 以setter方式构成的循环依赖，spring可以帮你解决，以构造器方式构成的循环依赖，spring无法搞定。
- setter 注入和构造器注入的区别就在于创建bean的过程中，**setter注入可以先用无参数构造方法返回bean实例，再注入依赖的属性**，使用到了 Spring 的三级缓存，而constructor方式**无法先返回bean的实例，必须先实例化它所依赖的属性，这样一来就会造成死循环所以会失败。**

JDK 动态代理和 CGLIB 的区别

静态代理与动态代理 静态代理，是编译时增强，AOP 框架会在编译阶段生成 AOP 代理类，在程序运行前代理类的.class文件就已经存在了。常见的实现：JDK静态代理，AspectJ 。 动态代理，是运行时增强，它不修改代理类的字节码，而是在程序运行时，运用反射机制，在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

JDK 动态代理基于接口，所以只有接口中的方法会被增强，而 CGLIB 基于类继承，需要注意就是如果方法使用了 final 修饰，或者是 private 方法，是不能被增强的。

Springmvc 请求处理的流程



（找时间详细看下，结合其他的自己总结下）

1. 客户端发送url请求，前端控制器（**DispatcherServlet**）接收到这个请求然后转发给处理器映射器（**HandlerMapping**）。
2. 处理器映射器会对url请求进行分析，找到对应的后端控制器（Handler），并且生成处理器对象及处理器拦截器（形成一条执行链）返回给前端控制器。
3. 根据处理器映射器返回的后端控制器(Handler)的名称/索引， 前端控制器 会找合适的处理器适配器（**HandlerAdapter**）
4. 处理器适配器会去执行后端控制器(Handler在开发的时候会被叫成**Controller**）。补充：执行之前会有转换器、数据绑定、校验器等等操作。完成上面这些才会去执行后端控制器。
5. 后端控制器Handler执行完成之后返回一个 **ModelAndView** 对象， Model 是返回的数据对象，View 是个逻辑上的 View。
6. 处理器适配器会将这个 ModelAndView 返回前端控制器。前端控制器会将 ModelAndView 对象交给合适的视图解析器 **ViewResolver** 。
7. 视图解析器（ViewResolver）解析 ModelAndView 对象,返回 **视图对象（view）**。
8. 前端控制器请求视图对视图对象（View）进行渲染(数据填充)之后返回并响应给浏览器/客户端。

Springmvc 有哪些组件

1、前端控制器DispatcherServlet（不需要工程师开发）,由框架提供（重要）

作用：**Spring MVC 的入口函数。接收请求，响应结果，相当于转发器，中央处理器。有了DispatcherServlet 减少了其它组件之间的耦合度。用户请求到达前端控制器，它就相当于mvc模式中的c，DispatcherServlet是整个流程控制的中心，由它调用其它组件处理用户的请求，DispatcherServlet的存在降低了组件之间的耦合性。**

2、处理器映射器HandlerMapping(不需要工程师开发),由框架提供

作用：根据请求的url查找Handler。HandlerMapping负责根据用户请求找到Handler即处理器（Controller），SpringMVC提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

3、处理器适配器HandlerAdapter

作用：按照特定规则（HandlerAdapter要求的规则）去执行Handler 通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

4、处理器Handler(需要工程师开发)

注意：编写Handler时按照HandlerAdapter的要求去做，这样适配器才可以去正确执行Handler
Handler 是继DispatcherServlet前端控制器的后端控制器，在DispatcherServlet的控制下Handler对具体的用户请求进行处理。 由于Handler涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发Handler。

5、视图解析器View resolver(不需要工程师开发),由框架提供

作用：进行视图解析，根据逻辑视图名解析成真正的视图（view） View Resolver负责将处理结果生成View视图，View Resolver首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。 springmvc框架提供了很多的View视图类型，包括：jstlView、freemarkerView、pdfView等。 一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。

6、视图View(需要工程师开发)

View是一个接口，实现类支持不同的View类型（jsp、freemarker、pdf...）

@Transactional(rollbackFor = Exception.class)注解的注意事项

- 使用阿里巴巴编码规范插件，使用`@Transactional`注解，如果不加`rollbackFor`，会提示`需要在Transactional注解指定rollbackFor或者在方法中显式的rollback`

原因：我们知道：Exception分为运行时异常RuntimeException和非运行时异常。事务管理对于企业应用来说是至关重要的，即使出现异常情况，它也可以保证数据的一致性。

当`@Transactional`注解作用于类上时，该类的所有 public 方法将都具有该类型的事务属性，同时，我们也可以在方法级别使用该标注来覆盖类级别的定义。如果类或者方法加了这个注解，那么这个类里面的方法抛出异常，就会回滚，数据库里面的数据也会回滚。

在`@Transactional`注解中如果不配置`rollbackFor`属性,那么事物只会在遇到`RuntimeException`的时候才会回滚,加上`rollbackFor=Exception.class`,可以让事物在遇到非运行时异常时也回滚。

Spring中事务传播行为/机制

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。

Spring中的7个事务传播行为:

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**：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。

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。
- **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`。

Mybatis中#{}和${}的区别是什么?

- #{}是预编译处理，mybatis在处理${}时**会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值**
- ${}是字符串替换，mybatis在处理#{}时会**把${}替换成变量的值。**
- 总结：使用#{}可以有效的防止SQL注入，提高系统安全性。