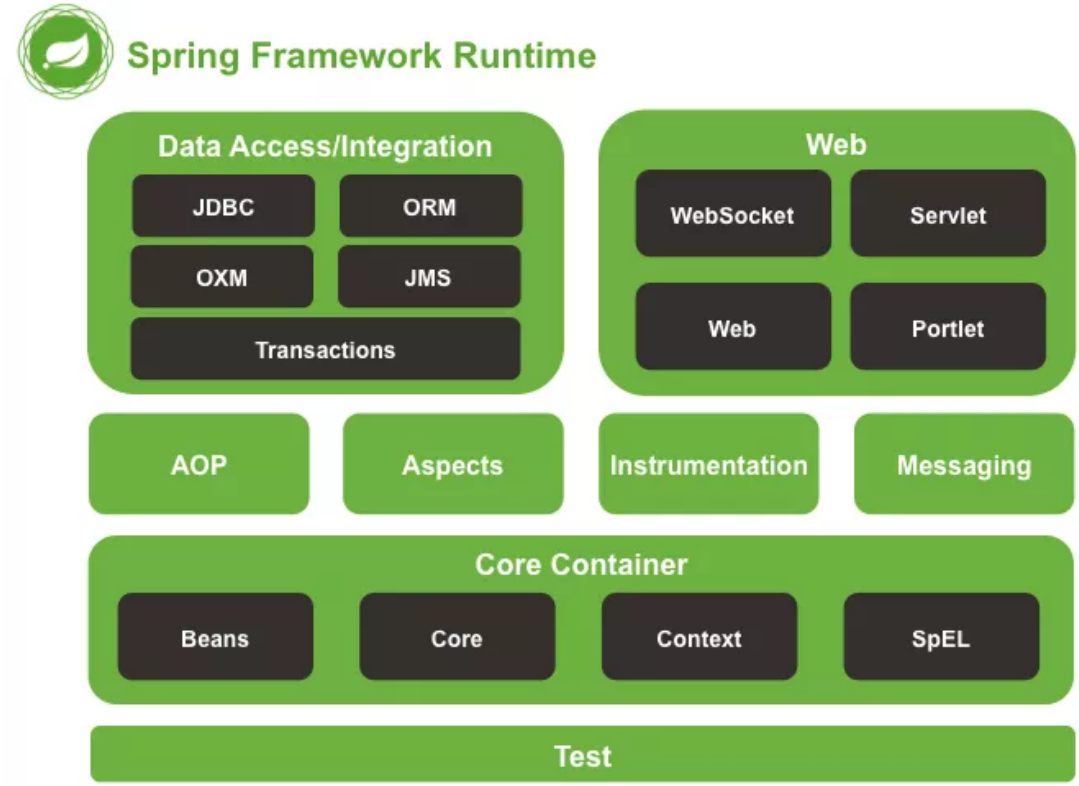
什么是 Spring 框架?

Spring 是一种轻量级开发框架，旨在提高开发人员的开发效率以及系统的可维护性。

我们一般说 Spring 框架指的都是 Spring Framework，它是很多模块的集合，使用这些模块可以很方便地协助我们进行开发。这些模块是：核心容器、数据访问/集成,、Web、AOP（面向切面编程）、工具、消息和测试模块。比如：Core Container 中的 Core 组件是Spring 所有组件的核心，Beans 组件和 Context 组件是实现IOC和依赖注入的基础，AOP组件用来实现面向切面编程。

列举一些重要的Spring模块?

下图对应的是 Spring4.x 版本。目前最新的5.x版本中 Web 模块的 Portlet 组件已经被废弃掉，同时增加了用于异步响应式处理的 WebFlux 组件。



Spring主要模块

- **Spring Core：** 基础,可以说 Spring 其他所有的功能都需要依赖于该类库。主要提供 IOC 依赖注入功能。
- **Spring Aspects** ： 该模块为与AspectJ的集成提供支持。
- **Spring AOP** ： 提供了面向方面的编程实现。
- **Spring JDBC** : Java数据库连接。
- **Spring JMS** ： Java消息服务。
- **Spring ORM** : 用于支持Hibernate等ORM工具。
- **Spring Web** : 为创建Web应用程序提供支持。
- **Spring Test** : 提供了对 JUnit 和 TestNG 测试的支持。

谈谈自己对于 Spring IoC 和 AOP 的理解

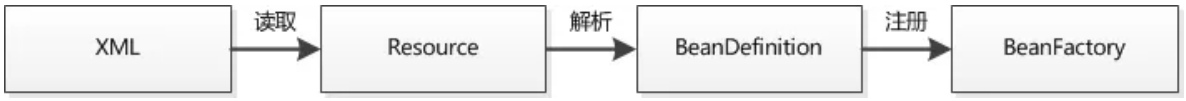
IoC

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 Spirng 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。** 在实际项目中一个 Service 类可能有几百甚至上千个类作为它的底层，假如我们需要实例化这个 Service，你可能要每次都要搞清这个 Service 所有底层类的构造函数，这可能会把人逼疯。如果利用 IOC 的话，你只需要配置好，然后在需要的地方引用/自动注入就行了，这大大增加了项目的可维护性且降低了开发难度。

Spring 时代我们一般通过 XML 文件来配置 Bean，后来开发人员觉得 XML 文件来配置不太好，于是 SpringBoot 注解配置就慢慢开始流行起来。

Spring IOC的初始化过程：



Spring IOC的初始化过程

Spring容器的创建过程：

Spring容器的refresh()【创建刷新】；

1、prepareRefresh()刷新前的预处理;

1）、initPropertySources()初始化一些属性设置;子类自定义个性化的属性设置方法； 2）、getEnvironment().validateRequiredProperties();检验属性的合法等 3）、earlyApplicationEvents= new LinkedHashSet();保存容器中的一些早期的事件；

2、obtainFreshBeanFactory();获取BeanFactory；

1）、refreshBeanFactory();刷新【创建】BeanFactory； 创建了一个this.beanFactory = new DefaultListableBeanFactory(); 设置id； 2）、getBeanFactory();返回刚才GenericApplicationContext创建的BeanFactory对象； 3）、将创建的BeanFactory【DefaultListableBeanFactory】返回；

3、prepareBeanFactory(beanFactory);BeanFactory的预准备工作（BeanFactory进行一些设置）；

1）、设置BeanFactory的类加载器、支持表达式解析器... 2）、添加部分BeanPostProcessor【ApplicationContextAwareProcessor】 3）、设置忽略的自动装配的接口EnvironmentAware、EmbeddedValueResolverAware、xxx； 4）、注册可以解析的自动装配；我们能直接在任何组件中自动注入： BeanFactory、ResourceLoader、ApplicationEventPublisher、ApplicationContext 5）、添加BeanPostProcessor【ApplicationListenerDetector】 6）、添加编译时的AspectJ； 7）、给BeanFactory中注册一些能用的组件；environment【ConfigurableEnvironment】、 systemProperties【Map<String, Object>】、systemEnvironment【Map<String, Object>】

4、postProcessBeanFactory(beanFactory);BeanFactory准备工作完成后进行的后置处理工作；

1）、子类通过重写这个方法来在BeanFactory创建并预准备完成以后做进一步的设置。也可以没有

======================以上是BeanFactory的创建及预准备工作
==================================

5、invokeBeanFactoryPostProcessors(beanFactory);执行BeanFactoryPostProcessor的方法；

BeanFactoryPostProcessor：BeanFactory的后置处理器。在BeanFactory标准初始化之后执行的；
两个接口：BeanFactoryPostProcessor、BeanDefinitionRegistryPostProcessor 1）、执行BeanFactoryPostProcessor的方法； 先执行BeanDefinitionRegistryPostProcessor 1）、获取所有的BeanDefinitionRegistryPostProcessor； 2）、看先执行实现了PriorityOrdered优先级接口的BeanDefinitionRegistryPostProcessor、
postProcessor.postProcessBeanDefinitionRegistry(registry) 3）、在执行实现了Ordered顺序接口的BeanDefinitionRegistryPostProcessor；
postProcessor.postProcessBeanDefinitionRegistry(registry) 4）、最后执行没有实现任何优先级或者是顺序接口的BeanDefinitionRegistryPostProcessors；
postProcessor.postProcessBeanDefinitionRegistry(registry)

6、registerBeanPostProcessors(beanFactory);注册BeanPostProcessor（Bean的后置处理器）【 intercept bean creation】

不同接口类型的BeanPostProcessor；在Bean创建前后的执行时机是不一样的 BeanPostProcessor、DestructionAwareBeanPostProcessor、 InstantiationAwareBeanPostProcessor、SmartInstantiationAwareBeanPostProcessor、
MergedBeanDefinitionPostProcessor【internalPostProcessors】、

1）、获取所有的 BeanPostProcessor;后置处理器都默认可以通过PriorityOrdered、Ordered接口来执行优先级 2）、先注册PriorityOrdered优先级接口的BeanPostProcessor； 把每一个BeanPostProcessor；添加到BeanFactory中 beanFactory.addBeanPostProcessor(postProcessor); 3）、再注册Ordered接口的 4）、最后注册没有实现任何优先级接口的 5）、最终注册MergedBeanDefinitionPostProcessor【internalPostProcessors】； 6）、注册一个监听器ApplicationListenerDetector；来在Bean创建完成后检查是否是ApplicationListener， 如果是applicationContext.addApplicationListener((ApplicationListener<?>) bean);【只注册不执行】

7、initMessageSource();初始化MessageSource组件（做国际化功能；消息绑定，消息解析）；

1）、获取BeanFactory 2）、看容器中是否有id为messageSource的，类型是MessageSource的组件如果有赋值给messageSource，如果没有自己创建一个DelegatingMessageSource；
MessageSource：取出国际化配置文件中的某个key的值；能按照区域信息获取； 3）、把创建好的MessageSource注册在容器中，以后获取国际化配置文件的值的时候，

可以自动注入MessageSource； beanFactory.registerSingleton(MESSAGE_SOURCE_BEAN_NAME, this.messageSource); MessageSource.getMessage(String code, Object[] args, String defaultMessage, Locale locale);

8、initApplicationEventMulticaster();初始化事件派发器；

1）、获取BeanFactory 2）、从BeanFactory中获取applicationEventMulticaster的ApplicationEventMulticaster； 3）、如果上一步没有配置；创建一个SimpleApplicationEventMulticaster 4）、将创建的ApplicationEventMulticaster添加到BeanFactory中，以后其他组件直接自动注入

9、onRefresh();留给子容器（子类）

1、子类重写这个方法，在容器刷新的时候可以自定义逻辑；

10、registerListeners();给容器中将所有项目里面的ApplicationListener监听器注册进来；

1、从容器中拿到所有的ApplicationListener 2、将每个监听器添加到事件派发器中；getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName)；3、派发之前步骤产生的事件；

11、finishBeanFactoryInitialization(beanFactory); 初始化所有剩下的单实例bean；

1、beanFactory.preInstantiateSingletons();初始化后剩下的单实例bean 1）、获取容器中的所有Bean，依次进行初始化和创建对象 2）、获取Bean的定义信息；RootBeanDefinition 3）、Bean不是抽象的，是单实例的，是懒加载； 1）、判断是否是FactoryBean；是否是实现FactoryBean接口的Bean，也就是说，如果是工厂，那就创建一个工厂； 2）、不是工厂Bean。利用getBean(beanName);创建对象 0、getBean(beanName)； ioc.getBean(); 1、doGetBean(name, null, null, false); 2、先获取缓存中保存的单实例Bean。如果能获取到说明这个Bean之前被创建过（所有创建过的单实例Bean都会被缓存起来） 从private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>(256);获取的 3、缓存中获取不到，开始Bean的创建对象流程； 4、标记当前bean已经被创建（防止多线程的情况，出现重复创建） 5、获取Bean的定义信息； 6、【获取当前Bean依赖的其他Bean;如果有按照getBean()把依赖的Bean先创建出来；】 7、启动单实例Bean的创建流程； 1）、createBean(beanName, mbd, args); 2）、Object bean = resolveBeforeInstantiation(beanName, mbdToUse);让BeanPostProcessor先拦截返回代理对象；【InstantiationAwareBeanPostProcessor】：还没有创建bean,提前执行这个类型 的postProcessor；先触发：postProcessBeforeInstantiation()； 如果有返回值：触发postProcessAfterInitialization()；3）、如果前面的InstantiationAwareBeanPostProcessor没有返回代理对象；调用4） 4）、Object beanInstance = doCreateBean(beanName, mbdToUse, args);创建Bean 1）、【创建Bean实例】；createBeanInstance(beanName, mbd, args)；利用工厂方法或者对象的构造器创建出Bean实例；2）、applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName)；调用MergedBeanDefinitionPostProcessor的postProcessMergedBeanDefinition(mbd, beanType, beanName)；3）、【Bean属性赋值】populateBean(beanName, mbd, instanceWrapper)；赋值之前： 1）、拿到InstantiationAwareBeanPostProcessor后置处理器；postProcessAfterInstantiation()； 2）、拿到InstantiationAwareBeanPostProcessor后置处理器；postProcessPropertyValues()； =====赋值之前：=== 3）、应用Bean属性的值；为属性利用setter方法等进行赋值；利用反射调用 applyPropertyValues(beanName, mbd, bw, pvs)；4）、【Bean初始化】initializeBean(beanName, exposedObject, mbd)；1）、【执行Aware接口方法】invokeAwareMethods(beanName, bean);执行xxxAware接口的方法BeanNameAware\BeanClassLoaderAware\BeanFactoryAware 2）、【执行后置处理器初始化之前】applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);BeanPostProcessor.postProcessBeforeInitialization（）；3）、【执行初始化方法】invokeInitMethods(beanName, wrappedBean, mbd)；1）、是否是InitializingBean接口的实现；执行接口规定的初始化； 2）、是否自定义初始化方法； 4）、【执行后置处理器初始化之后】applyBeanPostProcessorsAfterInitialization BeanPostProcessor.postProcessAfterInitialization()；5）、注册Bean的销毁方法；（先注册，等到销毁的时候会调用） 5）、将创建的Bean添加到缓存

中，也就是singletonObjects；【private final Map<String, Object> singletonObjects】 ioc容器就是这些Map；很多的各种Map里面保存了单实例Bean，环境信息。。。。； 所有Bean都利用getBean创建完成以后； 检查所有的Bean是否是SmartInitializingSingleton接口的；如果是；就执行afterSingletonsInstantiated()；

12、finishRefresh(); 完成BeanFactory的初始化创建工作；IOC容器就创建完成；

1）、initLifecycleProcessor();初始化和生命周期有关的后置处理器；LifecycleProcessor 默认从容器中找是否有lifecycleProcessor的组件【LifecycleProcessor】；如果没有new DefaultLifecycleProcessor(); 注册（加入）到容器，这样需要使用的时候，可以自动注入，其他的组件也是这个思想，先加入到容器中，可以自动注入；

写一个LifecycleProcessor的实现类，可以在BeanFactory void onRefresh(); void onClose(); 2）、getLifecycleProcessor().onRefresh(); 拿到前面定义的生命周期处理器（BeanFactory）；回调onRefresh()； 3）、publishEvent(new ContextRefreshedEvent(this));发布容器刷新完成事件；4）、liveBeansView.registerApplicationContext(this);

==========总结==========

1）、Spring容器在启动的时候，先会保存所有注册进来的Bean的定义信息； 1）、xml注册bean；2）、注解注册Bean；@Service、@Component、@Bean、xxx 2）、Spring容器会合适的时机创建这些Bean 1）、用到这个bean的时候；利用getBean创建bean；创建好以后保存在容器中； 2）、统一创建剩下所有的bean的时候；finishBeanFactoryInitialization()； 3）、后置处理器；BeanPostProcessor 1）、每一个bean创建完成，都会使用各种后置处理器进行处理；来增强bean的功能； AutowiredAnnotationBeanPostProcessor:处理自动注入 AnnotationAwareAspectJAutoProxyCreator:来做AOP功能； xxx.... 增强的功能注解：AsyncAnnotationBeanPostProcessor .... 4）、事件驱动模型； ApplicationListener；事件监听；ApplicationEventMulticaster；事件派发：

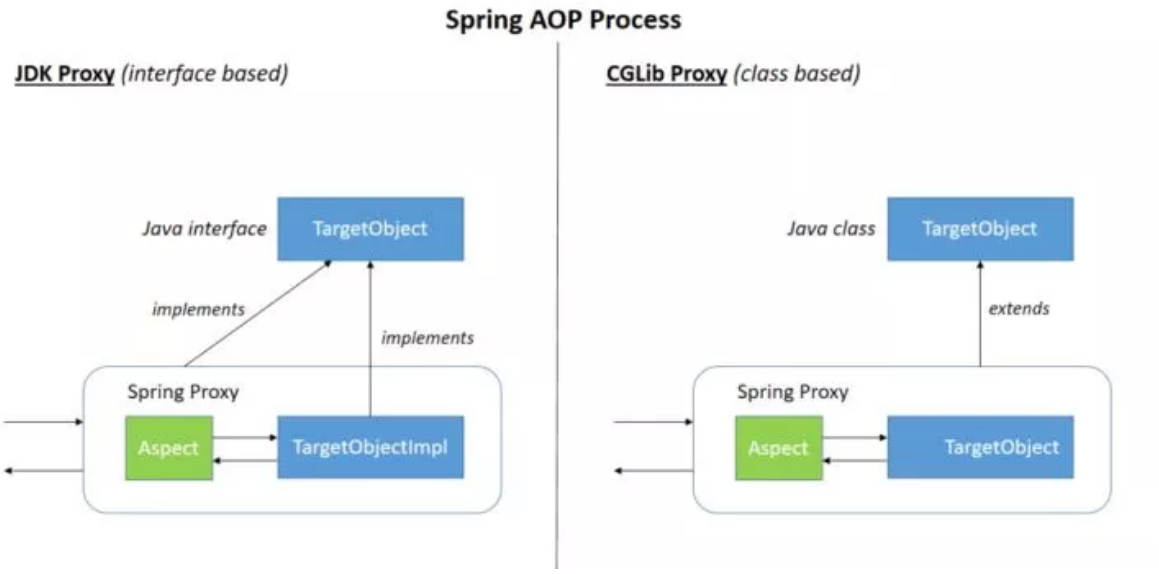
IOC源码阅读

- https://javadoop.com/post/spring-ioc
-

AOP

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，如下图所示：



SpringAOPProcess

当然你也可以使用 AspectJ ,Spring AOP 已经集成了AspectJ ， AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。

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 。

Spring AOP 和 AspectJ AOP 有什么区别?

Spring AOP 属于运行时增强，而 AspectJ 是编译时增强。 Spring AOP 基于代理(Proxying)，而 AspectJ 基于字节码操作(Bytecode Manipulation)。

Spring AOP 已经集成了 AspectJ ， AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。AspectJ 相比于 Spring AOP 功能更加强大，但是 Spring AOP 相对来说更简单，

如果我们的切面比较少，那么两者性能差异不大。但是，当切面太多的话，最好选择 AspectJ ，它比 Spring AOP 快很多。

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?

大部分时候我们并没有在系统中使用多线程，所以很少有人会关注这个问题。单例 bean 存在线程问题，主要是因为当多个线程操作同一个对象的时候，对这个对象的非静态成员变量的写操作会存在线程安全问题。

常见的有两种解决办法：

1. 在Bean对象中尽量避免定义可变的成员变量（不太现实）。

2. 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）。

Spring 中的 bean 生命周期?

这部分网上有很多文章都讲到了，下面的内容整理自：https://yemengying.com/2016/07/14/spring-bean-life-cycle/ ，除了这篇文章，再推荐一篇很不错的文章： https://www.cnblogs.com/zrtqsk/p/3735273.html 。

- Bean 容器找到配置文件中 Spring Bean 的定义。
- Bean 容器利用 Java Reflection API 创建一个Bean的实例。
- 如果涉及到一些属性值 利用 `set()` 方法设置一些属性值。
- 如果 Bean 实现了 `BeanNameAware` 接口，调用 `setBeanName()` 方法，传入Bean的名字。
- 如果 Bean 实现了 `BeanClassLoaderAware` 接口，调用 `setBeanClassLoader()` 方法，传入 `ClassLoader` 对象的实例。
- 如果Bean实现了 `BeanFactoryAware` 接口，调用 `setBeanClassLoader()` 方法，传入 `ClassLoade` r对象的实例。
- 与上面的类似，如果实现了其他 `*.Aware` 接口，就调用相应的方法。
- 如果有和加载这个 Bean 的 Spring 容器相关的 `BeanPostProcessor` 对象，执行 `postProcessBeforeInitialization()` 方法
- 如果Bean实现了 `InitializingBean` 接口，执行 `afterPropertiesSet()` 方法。
- 如果 Bean 在配置文件中的定义包含 init-method 属性，执行指定的方法。
- 如果有和加载这个 Bean的 Spring 容器相关的 `BeanPostProcessor` 对象，执行 `postProcessAfterInitialization()` 方法
- 当要销毁 Bean 的时候，如果 Bean 实现了 `DisposableBean` 接口，执行 `destroy()` 方法。
- 当要销毁 Bean 的时候，如果 Bean 在配置文件中的定义包含 destroy-method 属性，执行指定的方法。

图示：

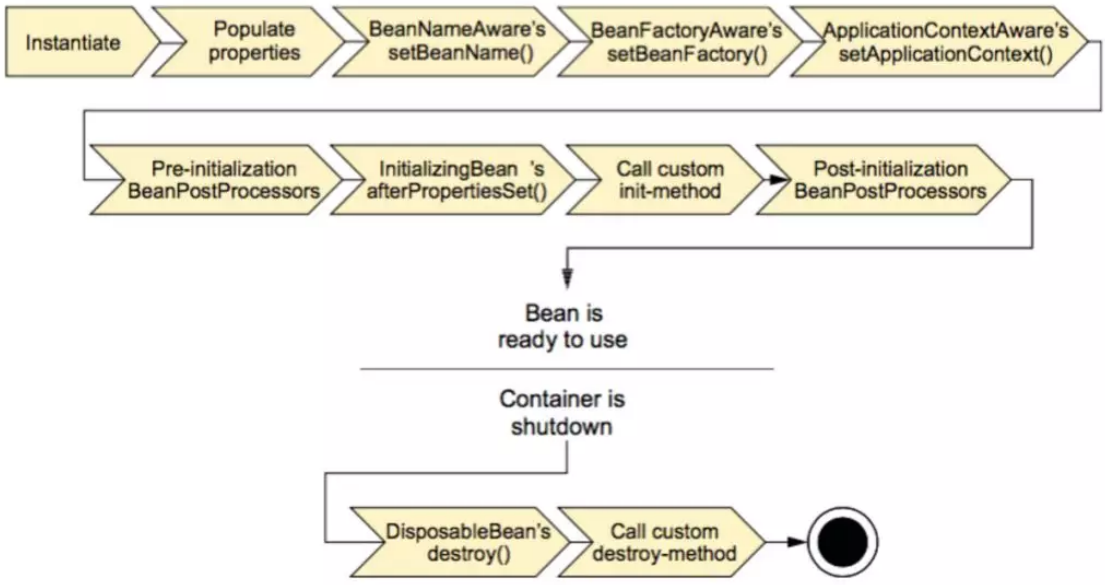
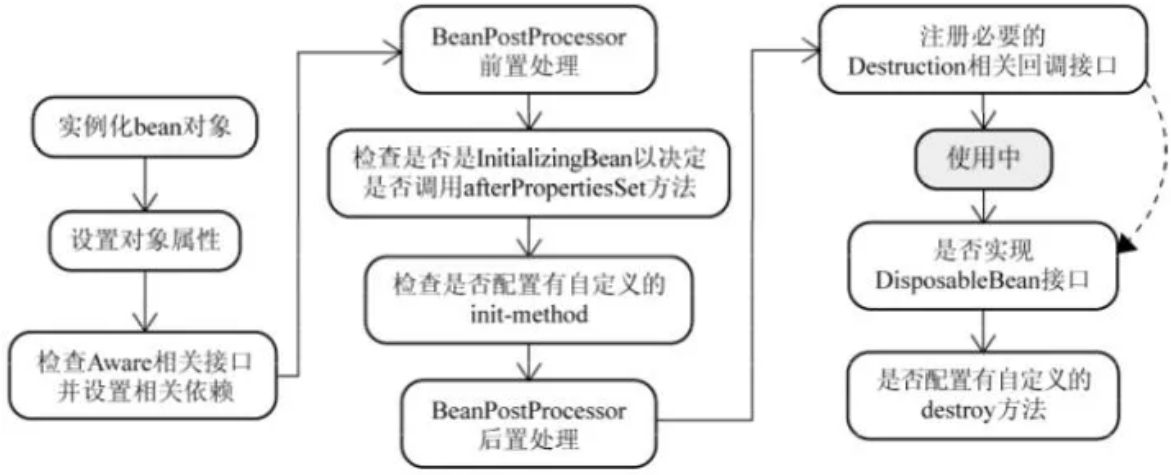


Figure 1.5 A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.

Spring Bean 生命周期

与之比较类似的中文版本:



Spring Bean 生命周期

说说自己对于 Spring MVC 了解?

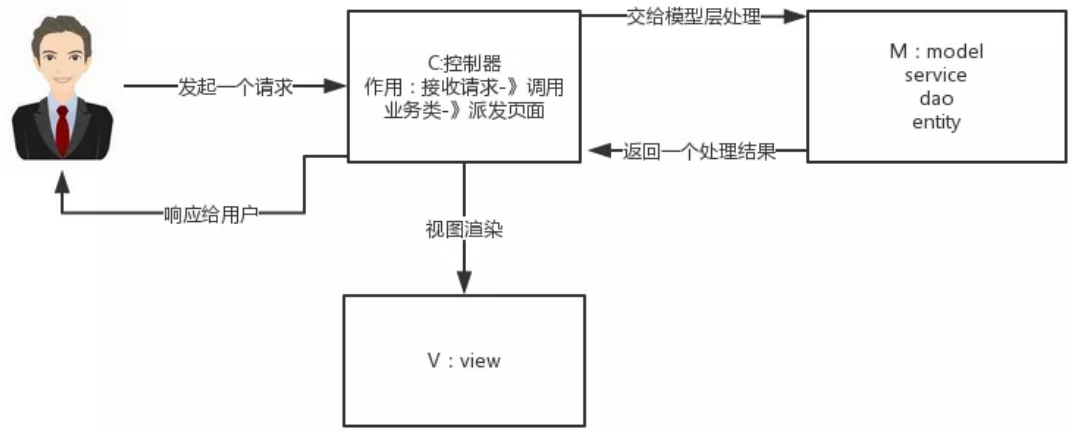
谈到这个问题，我们不得不提提之前 Model1 和 Model2 这两个没有 Spring MVC 的时代。

- **Model1 时代**：很多学 Java 后端比较晚的朋友可能并没有接触过 Model1 模式下的 JavaWeb 应用开发。在 Model1 模式下，整个 Web 应用几乎全部用 JSP 页面组成，只用少量的 JavaBean 来处理数据库连接、访问等操作。这个模式下 JSP 即是控制层又是表现层。显而易见，这种模式存在很多问题。比如①将控制逻辑和表现逻辑混杂在一起，导致代码重用率极低；②前端和后端相互依赖，难以进行测试并且开发效率极低；
- **Model2 时代**： 学过 Servlet 并做过相关 Demo 的朋友应该了解"Java Bean(Model)+ JSP（View,）+Servlet（Controller） "这种开发模式,这就是早期的 JavaWeb MVC 开发模式。Model:系统涉及的数据，也就是 dao 和 bean。View：展示模型中的数据，只是用来展示。Controller：处理用户请求都发送给 ，返回数据给 JSP 并展示给用户。

Model2 模式下还存在很多问题，Model2的抽象和封装程度还远远不够，使用Model2进行开发时不可避免地会重复造轮子，这就大大降低了程序的可维护性和复用性。于是很多JavaWeb开发相关的 MVC 框架营运而生比如Struts2，但是 Struts2 比较笨重。随着 Spring 轻量级开发框架的流行，Spring 生态圈出现了 Spring MVC 框架， Spring MVC 是当前最优秀的 MVC 框架。相比于 Struts2 ， Spring MVC 使用更加简单和方便，开发效率更高，并且 Spring MVC 运行速度更快。

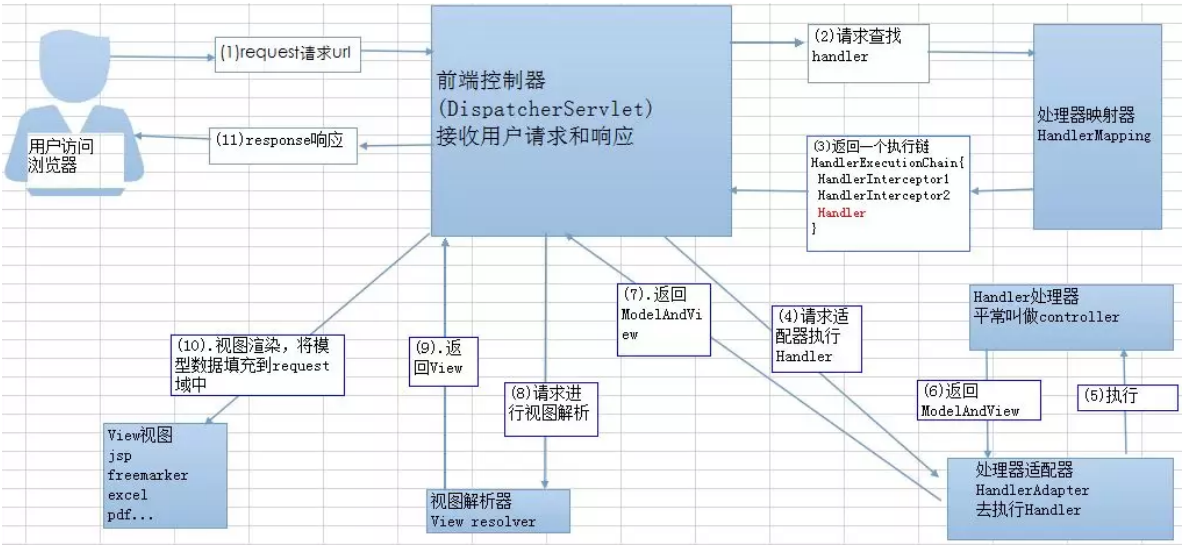
MVC 是一种设计模式,Spring MVC 是一款很优秀的 MVC 框架。Spring MVC 可以帮助我们进行更简洁的Web层的开发，并且它天生与 Spring 框架集成。Spring MVC 下我们一般把后端项目分为 Service层（处理业务）、Dao层（数据库操作）、Entity层（实体类）、Controller层(控制层，返回数据给前台页面)。

Spring MVC 的简单原理图如下：



SpringMVC 工作原理了解吗?

原理如下图所示：



SpringMVC运行原理

上图的一个笔误的小问题：Spring MVC 的入口函数也就是前端控制器 `DispatcherServlet` 的作用是接收请求，响应结果。

流程说明（重要）：

1. 客户端（浏览器）发送请求，直接请求到 `DispatcherServlet`。
2. `DispatcherServlet` 根据请求信息调用 `HandlerMapping`，解析请求对应的 `Handler`。
3. 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。
4. `HandlerAdapter` 会根据 `Handler` 来调用真正的处理器开处理请求，并处理相应的业务逻辑。
5. 处理器处理完业务后，会返回一个 `ModelAndView` 对象，`Model` 是返回的数据对象，`View` 是个逻辑上的 `View`。
6. `ViewResolver` 会根据逻辑 `View` 查找实际的 `View`。
7. `DispaterServlet` 把返回的 `Model` 传给 `View`（视图渲染）。
8. 把 `View` 返回给请求者（浏览器）

Spring 框架中用到了哪些设计模式?

关于下面一些设计模式的详细介绍，可以看笔主前段时间的原创文章《面试官:“谈谈Spring中都用到了那些设计模式?”。》。

- **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`。
- ……

@Component 和 @Bean 的区别是什么?

1. 作用对象不同: `@Component` 注解作用于类，而`@Bean`注解作用于方法。
2. `@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（我们可以使用 `@ComponentScan` 注解定义要扫描的路径从中找出标识了需要装配的类自动装配到 Spring 的 bean 容器中）。`@Bean` 注解通常是我们在标有该注解的方法中定义产生这个 bean,`@Bean`告诉了Spring这是某个类的示例，当我需要用它的时候还给我。
3. `@Bean` 注解比 `Component` 注解的自定义性更强，而且很多地方我们只能通过 `@Bean` 注解来注册bean。比如当我们引用第三方库中的类需要装配到 `Spring`容器时，则只能通过 `@Bean`来实现。

`@Bean`注解使用示例:

```
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }

}
```

上面的代码相当于下面的 xml 配置

```
<beans>
    <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

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();
    }
}
```

将一个类声明为Spring的 bean 的注解有哪些?

我们一般使用 `@Autowired` 注解自动装配 bean，要想把类标识成可用于 `@Autowired` 注解自动装配的 bean 的类,采用以下注解可实现：

- `@Component` ： 通用的注解，可标注任意类为 `Spring` 组件。如果一个Bean不知道属于拿个层，可以使用 `@Component` 注解标注。
- `@Repository` : 对应持久层即 Dao 层，主要用于数据库相关操作。
- `@Service` : 对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao层。
- `@Controller` : 对应 Spring MVC 控制层，主要用户接受用户请求并调用 Service 层返回数据给前端页面。

AOP：【动态代理】

- 指在程序运行期间动态的将某段代码切入到指定方法指定位置进行运行的编程方式；
- 1、导入aop模块；Spring AOP：(spring-aspects)
- 2、定义一个业务逻辑类（MathCalculator）；在业务逻辑运行的时候将日志进行打印（方法之前、方法运行结束、方法出现异常，xxx）
- 3、定义一个日志切面类（LogAspects）：切面类里面的方法需要动态感知MathCalculator.div运行到哪里然后执行；
- 通知方法：
- 前置通知(@Before)：logStart：在目标方法(div)运行之前运行
- 后置通知(@After)：logEnd：在目标方法(div)运行结束之后运行（无论方法正常结束还是异常结束）
- 返回通知(@AfterReturning)：logReturn：在目标方法(div)正常返回之后运行
- 异常通知(@AfterThrowing)：logException：在目标方法(div)出现异常以后运行
- 环绕通知(@Around)：动态代理，手动推进目标方法运行（joinPoint.procced()）
- 4、给切面类的目标方法标注何时何地运行（通知注解）；
- 5、将切面类和业务逻辑类（目标方法所在类）都加入到容器中;
- 6、必须告诉Spring哪个类是切面类(给切面类上加一个注解：@Aspect)
- [7]、给配置类中加 @EnableAspectJAutoProxy 【开启基于注解的aop模式】
- 在Spring中很多的 @EnableXXX;
-
- 三步：
- 1）、将业务逻辑组件和切面类都加入到容器中；告诉Spring哪个是切面类（@Aspect）
- 2）、在切面类上的每一个通知方法上标注通知注解，告诉Spring何时何地运行（切入点表达式）
- 3）、开启基于注解的aop模式；@EnableAspectJAutoProxy
-
- AOP原理：【看给容器中注册了什么组件，这个组件什么时候工作，这个组件的功能是什么？】

- @EnableAspectJAutoProxy；
- 1、@EnableAspectJAutoProxy是什么?
- @Import(AspectJAutoProxyRegistrar.class)：给容器中导入AspectJAutoProxyRegistrar
- 利用AspectJAutoProxyRegistrar自定义给容器中注册bean；BeanDefinetion
- internalAutoProxyCreator=AnnotationAwareAspectJAutoProxyCreator
- 给容器中注册一个AnnotationAwareAspectJAutoProxyCreator；
-
- 2、 AnnotationAwareAspectJAutoProxyCreator：
- AnnotationAwareAspectJAutoProxyCreator
- ->AspectJAwareAdvisorAutoProxyCreator
- ->AbstractAdvisorAutoProxyCreator
- ->AbstractAutoProxyCreator
- implements SmartInstantiationAwareBeanPostProcessor, BeanFactoryAware
-
- 关注后置处理器（在bean初始化完成前后做事情）、自动装配BeanFactory
- AbstractAutoProxyCreator.setBeanFactory()
- AbstractAutoProxyCreator.有后置处理器的逻辑；
- AbstractAdvisorAutoProxyCreator.setBeanFactory()-》initBeanFactory()
- AnnotationAwareAspectJAutoProxyCreator.initBeanFactory()
-
- 流程：
- 1）、传入配置类，创建ioc容器
- 2）、注册配置类，调用refresh（）刷新容器；
- 3）、registerBeanPostProcessors(beanFactory);注册bean的后置处理器来方便拦截bean的创建；
- 1）、先获取ioc容器已经定义了的需要创建对象的所有BeanPostProcessor
- 2）、给容器中加别的BeanPostProcessor
- 3）、优先注册实现了PriorityOrdered接口的BeanPostProcessor；
- 4）、再给容器中注册实现了Ordered接口的BeanPostProcessor；
- 5）、注册没实现优先级接口的BeanPostProcessor；
- 6）、注册BeanPostProcessor，实际上就是创建BeanPostProcessor对象，保存在容器中；
- 创建internalAutoProxyCreator的BeanPostProcessor【AnnotationAwareAspectJAutoProxyCreator】
- 1）、创建Bean的实例
- 2）、populateBean；给bean的各种属性赋值
- 3）、initializeBean：初始化bean；
- 1）、invokeAwareMethods()：处理Aware接口的方法回调
- 2）、applyBeanPostProcessorsBeforeInitialization()：应用后置处理器的postProcessBeforeInitialization（）
- 3）、invokeInitMethods()；执行自定义的初始化方法
- 4）、applyBeanPostProcessorsAfterInitialization()；执行后置处理器的postProcessAfterInitialization（）；
- 4）、BeanPostProcessor(AnnotationAwareAspectJAutoProxyCreator)创建成功；--》aspectJAdvisorsBuilder
- 7）、把BeanPostProcessor注册到BeanFactory中；
- beanFactory.addBeanPostProcessor(postProcessor);
- =======以上是创建和注册AnnotationAwareAspectJAutoProxyCreator的过程========
-
- AnnotationAwareAspectJAutoProxyCreator => InstantiationAwareBeanPostProcessor
- 4）、finishBeanFactoryInitialization(beanFactory);完成BeanFactory初始化工作；创建剩下的单实例bean
- 1）、遍历获取容器中所有的Bean，依次创建对象getBean(beanName);

- getBean->doGetBean()->getSingleton()->
- 2）、创建bean
- 【AnnotationAwareAspectJAutoProxyCreator在所有bean创建之前会有一个拦截，InstantiationAwareBeanPostProcessor，会调用postProcessBeforeInstantiation()】
- 1）、先从缓存中获取当前bean，如果能获取到，说明bean是之前被创建过的，直接使用，否则再创建；
- 只要创建好的Bean都会被缓存起来
- 2）、createBean（）;创建bean；
- AnnotationAwareAspectJAutoProxyCreator 会在任何bean创建之前先尝试返回bean的实例
- 【BeanPostProcessor是在Bean对象创建完成初始化前后调用的】
- 【InstantiationAwareBeanPostProcessor是在创建Bean实例之前先尝试用后置处理器返回对象的】
- 1）、resolveBeforeInstantiation(beanName, mbdToUse);解析BeforeInstantiation
- 希望后置处理器在此能返回一个代理对象；如果能返回代理对象就使用，如果不能就继续
- 1）、后置处理器先尝试返回对象；
- bean = applyBeanPostProcessorsBeforeInstantiation（）：
- 拿到所有后置处理器，如果是InstantiationAwareBeanPostProcessor;
- 就执行postProcessBeforeInstantiation
- if (bean != null)
- {
- bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);
- }
- 2）、doCreateBean(beanName, mbdToUse, args);真正的去创建一个bean实例；和3.6流程一样；
- 3）、
- AnnotationAwareAspectJAutoProxyCreator【InstantiationAwareBeanPostProcessor】 的作用：
- 1）、每一个bean创建之前，调用postProcessBeforeInstantiation()；
- 关心MathCalculator和LogAspect的创建
- 1）、判断当前bean是否在advisedBeans中（保存了所有需要增强bean）
- 2）、判断当前bean是否是基础类型的Advice、Pointcut、Advisor、AopInfrastructureBean，
- 或者是否是切面（@Aspect）
- 3）、是否需要跳过
- 1）、获取候选的增强器（切面里面的通知方法）【List candidateAdvisors】
- 每一个封装的通知方法的增强器是 InstantiationModelAwarePointcutAdvisor；
- 判断每一个增强器是否是 AspectJPointcutAdvisor 类型的；返回true
- 2）、永远返回false
- 2）、创建对象
- postProcessAfterInitialization；
- return wrapIfNecessary(bean, beanName, cacheKey);//包装如果需要的情况下
- 1）、获取当前bean的所有增强器（通知方法） Object[] specificInterceptors
- 1、找到候选的所有的增强器（找哪些通知方法是需要切入当前bean方法的）
- 2、获取到能在bean使用的增强器。
- 3、给增强器排序
- 2）、保存当前bean在advisedBeans中；
- 3）、如果当前bean需要增强，创建当前bean的代理对象；
- 1）、获取所有增强器（通知方法）
- 2）、保存到proxyFactory
- 3）、创建代理对象：Spring自动决定
- JdkDynamicAopProxy(config);jdk动态代理；
- ObjenesisCglibAopProxy(config);cglib的动态代理；
- 4）、给容器中返回当前组件使用cglib增强了的代理对象；

- 5）、以后容器中获取到的就是这个组件的代理对象，执行目标方法的时候，代理对象就会执行通知方法的流程；
- 3）、目标方法执行 ；
- 容器中保存了组件的代理对象（cglib增强后的对象），这个对象里面保存了详细信息（比如增强器，目标对象，xxx）；
- 1）、CglibAopProxy.intercept();拦截目标方法的执行
- 2）、根据ProxyFactory对象获取将要执行的目标方法拦截器链；
- List chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);
- 1）、List interceptorList保存所有拦截器 初始化长度5
- 一个默认的ExposeInvocationInterceptor 和 4个增强器；
- 2）、遍历所有的增强器，将其转为Interceptor；
- registry.getInterceptors(advisor);
- 3）、将增强器转为List；
- 如果是MethodInterceptor，直接加入到集合中
- 如果不是，使用AdvisorAdapter将增强器转为MethodInterceptor；
- 转换完成返回MethodInterceptor数组；
- 3）、如果没有拦截器链，直接执行目标方法;
- 拦截器链（其实就是每一个通知方法又被包装为方法拦截器，利用MethodInterceptor机制）
- 4）、如果有拦截器链，把需要执行的目标对象，目标方法，
- 拦截器链等信息传入创建一个 CglibMethodInvocation 对象，
- 并调用 Object retVal = mi.proceed();
- 5）、拦截器链的触发过程;
- 1)、如果没有拦截器执行目标方法，或者拦截器的索引和拦截器数组-1大小一样（指定到了最后一个拦截器）执行目标方法；
- 2)、链式获取每一个拦截器，拦截器执行invoke方法，每一个拦截器等待下一个拦截器执行完成返回以后再来执行；
- 拦截器链的机制，保证通知方法与目标方法的执行顺序；
- 总结：
- 1）、 @EnableAspectJAutoProxy 开启AOP功能
- 2）、 @EnableAspectJAutoProxy 会给容器中注册一个组件 AnnotationAwareAspectJAutoProxyCreator
- 3）、AnnotationAwareAspectJAutoProxyCreator是一个后置处理器；
- 4）、容器的创建流程：
- 1）、registerBeanPostProcessors（）注册后置处理器；创建 AnnotationAwareAspectJAutoProxyCreator对象
- 2）、finishBeanFactoryInitialization（）初始化剩下的单实例bean
- 1）、创建业务逻辑组件和切面组件
- 2）、AnnotationAwareAspectJAutoProxyCreator拦截组件的创建过程
- 3）、组件创建完之后，判断组件是否需要增强
- 是：切面的通知方法，包装成增强器（Advisor）;给业务逻辑组件创建一个代理对象（cglib）；
- 5）、执行目标方法：
- 1）、代理对象执行目标方法
- 2）、CglibAopProxy.intercept()；
- 1）、得到目标方法的拦截器链（增强器包装成拦截器MethodInterceptor）
- 2）、利用拦截器的链式机制，依次进入每一个拦截器进行执行；
- 3）、效果：
- 正常执行：前置通知-》目标方法-》后置通知-》返回通知
- 出现异常：前置通知-》目标方法-》后置通知-》异常通知
-

1、Spring是什么?

Spring是一个轻量级的IoC和AOP容器框架。是为Java应用程序提供基础性服务的一套框架，目的是用于简化企业应用程序的开发，它使得开发者只需要关心业务需求。常见的配置方式有三种：基于XML的配置、基于注解的配置、基于Java的配置。

主要由以下几个模块组成：

Spring Core：核心类库，提供IOC服务；

Spring Context：提供框架式的Bean访问方式，以及企业级功能（JNDI、定时任务等）；

Spring AOP：AOP服务；

Spring DAO：对JDBC的抽象，简化了数据访问异常的处理；

Spring ORM：对现有的ORM框架的支持；

Spring Web：提供了基本的面向Web的综合特性，例如多方文件上传；

Spring MVC：提供面向Web应用的Model-View-Controller实现。

2、Spring 的优点?

（1）spring属于低侵入式设计，代码的污染极低；

（2）spring的DI机制将对象之间的依赖关系交由框架处理，减低组件的耦合性；

（3）Spring提供了AOP技术，支持将一些通用任务，如安全、事务、日志、权限等进行集中式管理，从而提供更好的复用。

（4）spring对于主流的应用框架提供了集成支持。

3、Spring的AOP理解：

OOP面向对象，允许开发者定义纵向的关系，但并适用于定义横向的关系，导致了大量代码的重复，而不利于各个模块的重用。

AOP，一般称为面向切面，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理。

AOP实现的关键在于 代理模式，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。

（1）AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强，他会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。

（2）Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时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。

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：

①JDK动态代理只提供接口的代理，不支持类的代理。核心InvocationHandler接口和Proxy类，InvocationHandler 通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例, 生成目标类的代理对象。

②如果代理类没有实现 InvocationHandler 接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。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）静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

InvocationHandler 的 invoke(Object proxy,Method method,Object[] args)：proxy是最终生成的代理实例; method 是被代理目标实例的某个具体方法; args 是被代理目标实例某个方法的具体入参, 在方法反射调用时使用。

4、Spring的IoC理解：

（1）IOC就是控制反转，是指创建对象的控制权的转移，以前创建对象的主动权和时机是由自己把控的，而现在这种权力转移到Spring容器中，并由容器根据配置文件去创建实例和管理各个实例之间的依赖关系，对象与对象之间松散耦合，也利于功能的复用。DI依赖注入，和控制反转是同一个概念的不同角度的描述，即 应用程序在运行时依赖IoC容器来动态注入对象需要的外部资源。

（2）最直观的表达就是，IOC让对象的创建不用去new了，可以由spring自动生产，使用java的反射机制，根据配置文件在运行时动态的去创建对象以及管理对象，并调用对象的方法的。

（3）Spring的IOC有三种注入方式：构造器注入、setter方法注入、根据注解注入。

IoC让相互协作的组件保持松散的耦合，而AOP编程允许你把遍布于应用各层的功能分离出来形成可重用的功能组件。

5、BeanFactory和ApplicationContext有什么区别?

BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。其中ApplicationContext是BeanFactory的子接口。

（1）BeanFactory：是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系。ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：

①继承MessageSource，因此支持国际化。

②统一的资源文件访问方式。

③提供在监听器中注册bean的事件。

④同时加载多个配置文件。

⑤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，比如应用的web层。

（2）

①BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

②ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。

③相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

（3）BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。

（4）BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

6、请解释Spring Bean的生命周期?

首先说一下Servlet的生命周期：实例化，初始init，接收请求service，销毁destroy；

Spring上下文中的Bean生命周期也类似，如下：

（1）实例化Bean：

对于BeanFactory容器，当客户向容器请求一个尚未初始化的bean时，或初始化bean的时候需要注入另一个尚未初始化的依赖时，容器就会调用createBean进行实例化。对于ApplicationContext容器，当容器启动结束后，通过获取BeanDefinition对象中的信息，实例化所有的bean。

（2）设置对象属性（依赖注入）：

实例化后的对象被封装在BeanWrapper对象中，紧接着，Spring根据BeanDefinition中的信息 以及 通过BeanWrapper提供的设置属性的接口完成依赖注入。

（3）处理Aware接口：

接着，Spring会检测该对象是否实现了xxxAware接口，并将相关的xxxAware实例注入给Bean：

①如果这个Bean已经实现了BeanNameAware接口，会调用它实现的setBeanName(String beanId)方法，此处传递的就是Spring配置文件中Bean的id值；

②如果这个Bean已经实现了BeanFactoryAware接口，会调用它实现的setBeanFactory()方法，传递的是Spring工厂自身。

③如果这个Bean已经实现了ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文；

（4）BeanPostProcessor：

如果想对Bean进行一些自定义的处理，那么可以让Bean实现了BeanPostProcessor接口，那将会调用postProcessBeforeInitialization(Object obj, String s)方法。由于这个方法是在Bean初始化结束时调用的，所以可以被应用于内存或缓存技术；

（5）InitializingBean 与 init-method：

如果Bean在Spring配置文件中配置了 init-method 属性，则会自动调用其配置的初始化方法。

（6）如果这个Bean实现了BeanPostProcessor接口，将会调用postProcessAfterInitialization(Object obj, String s)方法；

以上几个步骤完成后，Bean就已经被正确创建了，之后就可以使用这个Bean了。

（7）DisposableBean：

当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean这个接口，会调用其实现的destroy()方法；

（8）destroy-method：

最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法。

7、 解释Spring支持的几种bean的作用域。

Spring容器中的bean可以分为5个范围：

（1）singleton：默认，每个容器中只有一个bean的实例，单例的模式由BeanFactory自身来维护。

（2）prototype：为每一个bean请求提供一个实例。

（3）request：为每一个网络请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。

（4）session：与request范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。

（5）global-session：全局作用域，global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。全局作用域与Servlet中的session作用域效果相同。

8、Spring框架中的单例Beans是线程安全的么?

Spring框架并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全和并发问题需要开发者自行去搞定。但实际上，大部分的Spring bean并没有可变的状态(比如Serview类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。如果你的bean有多种状态的话（比如 View Model 对象），就需要自行保证线程安全。最浅显的解决办法就是将多态bean的作用域由“singleton”变更为“prototype”。

9、Spring如何处理线程并发问题?

在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域，因为Spring对一些Bean中非线程安全状态采用ThreadLocal进行处理，解决线程安全问题。

ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。同步机制采用了“时间换空间”的方式，仅提供一份变量，不同的线程在访问前需要获取锁，没获得锁的线程则需要排队。而ThreadLocal采用了“空间换时间”的方式。

ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

10-1、Spring基于xml注入bean的几种方式：

（1）Set方法注入；

（2）构造器注入：①通过index设置参数的位置；②通过type设置参数类型；

（3）静态工厂注入；

（4）实例工厂；

详细内容可以阅读：https://blog.csdn.net/a745233700/article/details/89307518

10-2、Spring的自动装配：

在spring中，对象无需自己查找或创建与其关联的其他对象，由容器负责把需要相互协作的对象引用赋予各个对象，使用autowire来配置自动装载模式。

在Spring框架xml配置中共有5种自动装配：

（1）no：默认的方式是不进行自动装配的，通过手工设置ref属性来进行装配bean。

（2）byName：通过bean的名称进行自动装配，如果一个bean的 property 与另一bean 的name 相同，就进行自动装配。

（3）byType：通过参数的数据类型进行自动装配。

（4）constructor：利用构造函数进行装配，并且构造函数的参数通过byType进行装配。

（5）autodetect：自动探测，如果有构造方法，通过 construct的方式自动装配，否则使用byType的方式自动装配。

基于注解的方式：

使用@Autowired注解来自动装配指定的bean。在使用@Autowired注解之前需要在Spring配置文件进行配置，<context:annotation-config />。在启动spring IoC时，容器自动装载了一个AutowiredAnnotationBeanPostProcessor后置处理器，当容器扫描到@Autowied、@Resource或@Inject时，就会在IoC容器自动查找需要的bean，并装配给该对象的属性。在使用@Autowired时，首先在容器中查询对应类型的bean：

如果查询结果刚好为一个，就将该bean装配给@Autowired指定的数据；

如果查询的结果不止一个，那么@Autowired会根据名称来查找；

如果上述查找的结果为空，那么会抛出异常。解决方法时，使用required=false。

@Autowired可用于：构造函数、成员变量、Setter方法

注：@Autowired和@Resource之间的区别

(1) @Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。

(2) @Resource默认是按照名称来装配注入的，只有当找不到与名称匹配的bean才会按照类型来装配注入。

11、Spring 框架中都用到了哪些设计模式?

（1）工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；

（2）单例模式：Bean默认为单例模式。

（3）代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；

（4）模板方法：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。

（5）观察者模式：定义对象键一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被制动更新，如Spring中listener的实现--ApplicationListener。和发布订阅模式的区别

12、Spring事务的实现方式和实现原理：

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过binlog或者redo log实现的。

（1）Spring事务的种类：

spring支持编程式事务管理和声明式事务管理两种方式：

①编程式事务管理使用TransactionTemplate。

②声明式事务管理建立在AOP之上的。其本质是通过AOP功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务最大的优点就是不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明或通过@Transactional注解的方式，便可以将事务规则应用到业务逻辑中。

声明式事务管理要优于编程式事务管理，这正是spring倡导的非侵入式的开发方式，使业务代码不受污染，只要加上注解就可以获得完全的事务支持。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

（2）spring的事务传播行为：

spring事务的传播行为说的是，当多个事务同时存在的时候，spring如何处理这些事务的行为。

① PROPAGATION_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

② PROPAGATION_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。‘

③ PROPAGATION_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

④ PROPAGATION_REQUIRES_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

⑤ 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。

⑥ PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

⑦ 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，则按REQUIRED属性执行。

（3）Spring中的隔离级别：

① ISOLATION_DEFAULT：这是个 PlatfromTransactionManager 默认的隔离级别，使用数据库默认的事务隔离级别。

② ISOLATION_READ_UNCOMMITTED：读未提交，允许另外一个事务可以看到这个事务未提交的数据。

③ ISOLATION_READ_COMMITTED：读已提交，保证一个事务修改的数据提交后才能被另一事务读取，而且能看到该事务对已有记录的更新。

④ ISOLATION_REPEATABLE_READ：可重复读，保证一个事务修改的数据提交后才能被另一事务读取，但是不能看到该事务对已有记录的更新。

⑤ ISOLATION_SERIALIZABLE：一个事务在执行的过程中完全看不到其他事务对数据库所做的更新。

13、Spring框架中有哪些不同类型的事件?

Spring 提供了以下5种标准的事件：

（1）上下文更新事件（ContextRefreshedEvent）：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。

如果一个bean实现了ApplicationListener接口，当一个ApplicationEvent 被发布以后，bean会自动被通知。

14、解释一下Spring AOP里面的几个名词：

（1）切面（Aspect）：被抽取的公共模块，可能会横切多个对象。 在Spring AOP中，切面可以使用通用类（基于模式的风格） 或者在普通类中以 @AspectJ 注解来实现。

（2）连接点（Join point）：指方法，在Spring AOP中，一个连接点 总是 代表一个方法的执行。

（3）通知（Advice）：在切面的某个特定的连接点（Join point）上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。许多AOP框架，包括Spring，都是以拦截器做通知模型， 并维护一个以连接点为中心的拦截器链。

（4）切入点（Pointcut）：切入点是指 我们要对哪些Join point进行拦截的定义。通过切入点表达式，指定拦截的方法，比如指定拦截add、*search*。

（5）引入（Introduction）：（也被称为内部类型声明（inter-type declaration））。声明额外的方法或者某个类型的字段。Spring允许引入新的接口（以及一个对应的实现）到任何被代理的对象。例如，你可以使用一个引入来使bean实现 IsModified 接口，以便简化缓存机制。

（6）目标对象（Target Object）： 被一个或者多个切面（aspect）所通知（advise）的对象。 也有人把它叫做 被通知（adviced） 对象。 既然Spring AOP是通过运行时代理实现的，这个对象永远是一个 被代理（proxied） 对象。

（7）织入（Weaving）：指把增强应用到目标对象来创建新的代理对象的过程。Spring是在运行时完成织入。

切入点（pointcut）和连接点（join point）匹配的概念是AOP的关键，这使得AOP不同于其它仅仅提供拦截功能的旧技术。 切入点使得定位通知（advice）可独立于OO层次。 例如，一个提供声明式事务管理的around通知可以被应用到一组横跨多个对象中的方法上（例如服务层的所有业务操作）。

15、Spring通知有哪些类型?

（1）前置通知（Before advice）：在某连接点（join point）之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。

（2）返回后通知（After returning advice）：在某连接点（join point）正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。

（3）抛出异常后通知（After throwing advice）：在方法抛出异常退出时执行的通知。

（4）后通知（After (finally) advice）：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。

（5）环绕通知（Around Advice）：包围一个连接点（join point）的通知，如方法调用。这是最强大的一种通知类型。 环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。 环绕通知是最常用的一种通知类型。大部分基于拦截的AOP框架，例如Nanning和JBoss4，都只提供环绕通知。

同一个aspect，不同advice的执行顺序：

①没有异常情况下的执行顺序：

around before advice before advice target method 执行 around after advice after advice afterReturning

②有异常情况下的执行顺序：

around before advice before advice target method 执行 around after advice after advice afterThrowing:异常发生 java.lang.RuntimeException: 异常发生