# IOC

## IOC/DI

* 依赖注入（Dependecy Injection)和控制反转（Inversion of Control)是同一个概念，具体的讲：当某个角色 需要另外一个角色协助的时候，在传统的程序设计过程中，通常由调用者来创建被调用者的实例。但在spring中 创建被调用者的工作不再由调用者来完成，因此称为控制反转。创建被调用者的工作由spring来完成，然后注入调用者，因此也称为依赖注入。 IOC侧重于原理，DI 侧重于实现。
* 依赖倒置原则/面向接口编程：高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。

## Spring介绍

* Spring 是轻量级的JavaEE的开源框架，具有IOC/DI，AOP，MVC，TX，ORM等功能。
* 优点是：
* DI有效的降低了耦合度。
* AOP提供了通用的任务的集中管理。
* ORM简化了对数据库的访问。
* 低侵入式设计，代码污染极低。

## BeanFactory

* BeanFactory 只是对IOC容器最基本行为作了定义，而不关心 Bean 是怎样定义和加载的。
* 定义了一些getBean、containsBean、isSingleton、isPrototype等基本容器方法。

## XmlBeanFactory

* xmlBeanFactory是BeanFactory的一个实现类。xmlBeanFactory 的功能是建立在 DefaultListablexmlBeanFactory 这个基本容器的基础上的，并在这个基本容器的基础上添加了其他诸如 xml 读取的附加功能。

## Resource

* 仅仅使用 java 标准 java.net.URL 和针对不同 URL 前缀的标准处理器并不能满足我们对各种底层资源的访问，比如：我们就不能通过 URL 的标准实现来访问相对类路径或者相对 ServletContext 的各种资源。虽然我们可以针对特定的 url 前缀来注册一个新的 URLStreamHandler（和现有的针对各种特定前缀的处理器类似，比如 http：)，然而这往往会是一件比较麻烦的事情(要求了解 url 的实现机制等)，而且 url 接口也缺少了部分基本的方法，如检查当前资源是否存在的方法。
* 相对标准 url 访问机制，Spring 的 Resource 接口对抽象底层资源的访问提供了一套更好的机制。
* Resource 是 Spring 中对外部资源的抽象，最常见的是文件的抽象，特别是 xml 文件，而且 Resource 里面通常是保存了 Spring 使用者的 Bean 定义。
* 其实现类有：ByteArrayResouece,BeanDefinitionResource,InputStreamResource, ClassPathResource等
* ResourceLoader接口用于返回Resource对象；其实现可以看作是一个生产Resource的工厂类。Spring提供了一个适用于所有环境的DefaultResourceLoader实现，可以返回ClassPathResource、UrlResource。
* ResourceLoader在进行加载资源时需要使用前缀来指定需要加载：“classpath:path”表示返回ClasspathResource，“http://path”和“file:path”表示返回UrlResource资源，如果不加前缀则需要根据当前上下文来决定，DefaultResourceLoader默认实现可以加载classpath资源。
* 对于目前所有ApplicationContext都实现了ResourceLoader，因此可以使用其来加载资源。
* ClassPathXmlApplicationContext：不指定前缀将返回默认的ClassPathResource资源，否则将根据前缀来加载资源；
* WebApplicationContext：不指定前缀将返回ServletContextResource，否则将根据前缀来加载资源；

## ApplicationContext

* ApplicationContext是spring中较高级的容器。和BeanFactory类似，它可以加载配置文件中定义的bean，将所有的bean集中在一起，当有请求的时候分配bean。 另外，它增加了企业所需要的功能。

public interface ApplicationContext extends EnvironmentCapable, ListableBeanFactory, HierarchicalBeanFactory,

* MessageSource, ApplicationEventPublisher, ResourcePatternResolver
* 1.支持不同的信息源。继承了MessageSource接口，这个接口为ApplicationContext提供了很多信息源的扩展功能，比如：国际化的实现为多语言版本的应用提供服务。
* 2.访问资源。这一特性主要体现在ResourcePatternResolver接口上，对Resource和ResourceLoader的支持，这样我们可以从不同地方得到Bean定义资源。这种抽象使用户程序可以灵活地定义Bean定义信息，尤其是从不同的IO途径得到Bean定义信息。这在接口上看不出来，不过一般来说，具体ApplicationContext都是继承了DefaultResourceLoader的子类。因为DefaultResourceLoader是AbstractApplicationContext的基类。
* 3.支持应用事件。继承了接口ApplicationEventPublisher，为应用环境引入了事件机制，这些事件和Bean的生命周期的结合为Bean的管理提供了便利。
* 4.附件服务。EnvironmentCapable里的服务让基本的Ioc功能更加丰富。
* 5.ListableBeanFactory和HierarchicalBeanFactory是继承的主要容器。
* 最常被使用的ApplicationContext接口实现类：
* 1，FileSystemXmlApplicationContext：该容器从XML文件中加载已被定义的bean。在这里，你需要提供给构造器XML文件的完整路径。
* 2，ClassPathXmlApplicationContext：该容器从XML文件中加载已被定义的bean。在这里，你不需要提供XML文件的完整路径，只需正确配置CLASSPATH环境变量即可，因为，容器会从CLASSPATH中搜索bean配置文件。
* 3，WebXmlApplicationContext：该容器会在一个 web 应用程序的范围内加载在XML文件中

## ClassPathXmlApplicationContext

## 容器初始化过程

### Bean的注册

* 以上的这些过程都发生在new ClassPathXmlApplicationContext中调用的AbstractApplicationContext#refresh方法中。
* AbstractApplicationContext#refresh
* 逻辑：
  + 1)初始化前的准备工作，比如对系统属性或者环境变量进行准备及验证
  + 2)初始化BeanFactory，并进行XML文件读取（component-scan->包括class文件)
  + 3)对BeanFactory进行各种功能填充，比如@Qualifier和@Autowired
  + 4)子类覆盖方法做额外的处理
  + 5)激活各种BeanFactory处理器
  + 6)注册拦截bean创建的bean处理器，这里只是注册，真正的调用是在getBean的时候
  + 7)为上下文初始化Message源，即为不同语言的消息体进行国际化处理
  + 8)初始化应用消息广播器，并放入applicationEventMulticaster bean中
  + 9)留给子类来初始化其他的bean
* 10)在所有注册的bean中查找listener bean，注册到消息广播器中
  + 11)初始化剩下的代理实例（非lazy-init)（bean的加载)
  + 12)完成刷新过程，通知生命周期处理器lifecycleProcessor刷新过程，同时发出ContextRefreshEvent通知别人。

public void refresh() throws BeansException, IllegalStateException {  
 synchronized (this.startupShutdownMonitor) {  
 // 准备 刷新的上下文环境  
 prepareRefresh();  
  
 // 初始化BeanFactory，并进行XML文件的读取

* ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
* // 对BeanFactory进行各种功能填充
* prepareBeanFactory(beanFactory);
* try {  
   // 子类覆盖方法做额外的处理  
   postProcessBeanFactory(beanFactory);
* // 激活各种BeanFactory处理器  
   invokeBeanFactoryPostProcessors(beanFactory);
* // 注册拦截Bean创建的Bean处理器，这里只是注册，真正的调用是在getBean的时候  
   registerBeanPostProcessors(beanFactory);
* // 为上下文初始化Message源，即不同语言的消息体，国际化处理  
   initMessageSource();
* // 初始化应用消息广播器，并放入applicationEventMulticaster bean中  
   initApplicationEventMulticaster();
* // 留给子类来初始化其他bean  
   onRefresh();
* // 在所有注册的bean中查找Listener bean，注册到消息广播器中  
   registerListeners();
* // 初始化剩下的单例实例（除了lazy-init)  
   finishBeanFactoryInitialization(beanFactory);
* // 完成刷新过程，通知生命周期处理器lifecycleProcessor刷新过程，同时发出ContextRefreshEvent通知别人  
   finishRefresh();  
  }
* catch (BeansException ex) {  
   if (logger.isWarnEnabled()) {  
   logger.warn("Exception encountered during context initialization - " +  
   "cancelling refresh attempt: " + ex);  
   }
* // Destroy already created singletons to avoid dangling resources.  
   destroyBeans();
* // Reset 'active' flag.  
   cancelRefresh(ex);
* // Propagate exception to caller.  
   throw ex;  
  }
* finally {  
   // Reset common introspection caches in Spring's core, since we  
   // might not ever need metadata for singleton beans anymore...  
   resetCommonCaches();  
  }  
  }  
  }

#### obtainFreshBeanFactory（创建beanFactory，解析XML)

* 在后面会调用到loadBeanDefinitions(beanFactory)方法，该方法会创建XmlBeanDefinitionReader，后面会调用reader#loadBeanDefinitions方法。
* 该方法会：
  + 1)Resource[] resources = ((ResourcePatternResolver) resourceLoader).getResources(location);
* 将applicationContext.xml转为Resource。
  + 2)然后会调用Resource#getInputStream，然后将该资源转为XML的Document对象。
* 最终是使用jaxp的dom方式读取的XML配置文件。（JAXP是一种标准，SUN API对其提供了实现)
  + 3)使用BeanDefinitionDocumentReader#registerBeanDefinitions将XML对应的Document对象转为BeanDefinitions。
* 最终会调用parseBeanDefinitions（解析标签)，它会分开解析默认标签和自定义标签。
* 默认标签是import、alias、bean和beans四种。
* 自定义标签会先找到其对应的parser，比如component-scan标签对应着ComponentScanBeanDefinitionParser。该Parser会调用
* PathMatchingResourcePatternResolver#getResources从包名得到Resource封装了的class文件（底层是调用了File API)。MetadataReader会读入class文件，如果有@Component注解，那么返回其对应的BeanDefinition。
  + 4)标签解析完毕后会将beanName和beanDefinition作为key和value放入beanfactory的beanDefinitionMap中。

### Bean的加载

* 在Bean的注册refresh中除了调用了obtainFreshBeanFactory完成了XML2BeanDefinition，还调用了finishBeanFactoryInitialization（初始化非lazy-load且singleton的bean)（BeanDefinition2BeanInstance)，该方法会获取所有beanName，逐个调用getBean方法。

#### FactoryBean（工厂Bean，用户定制)

* Spring通过反射机制利用bean的class属性指定实现类来实例化bean。
* Spring提供了一个FactoryBean的工厂类接口，用户可以通过实现该接口定制实例化bean的逻辑。
* FactoryBean

public interface FactoryBean<T> {

* // 返回bean实例，如果isSingleton()返回true，那么该实例会放到Spring容器中单例缓存池中  
  T getObject() throws Exception;  
  Class<?> getObjectType();  
  boolean isSingleton();  
  }
* FactoryBean：
* 这个接口使你可以提供一个复杂的逻辑来生成Bean。它本质是一个Bean，但这个Bean不是用来注入到其它地方像Service、Dao一样使用的，它是用来生成其它Bean使用的。实现了这个接口后，Spring在容器初始化时，把实现这个接口的Bean取出来，使用接口的getObject()方法来生成我们要想的Bean。当然，那些生成Bean的业务逻辑也要写getObject()方法中。
* 其返回的对象不是指定类的一个实例，其返回的是该工厂Bean的getObject方法所返回的对象。创建出来的对象是否属于单例由isSingleton中的返回决定。
* 使用场景：1、通过外部对类是否是单例进行控制，该类自己无法感知 2、在创建Object对象之前进行初始化的操作，在afterPropertiesSet()中完成。（一次性的初始化，保存在成员变量中，并不是每次getObject都会调用afterPropertiesSet，afterPropertiesSet只会被调用一次)

#### ObjectFactory（Spring使用)

public interface ObjectFactory<T> {  
 T getObject() throws BeansException;  
}

* ObjectFactory：
* 它的目的也是作为一个工厂，来生成Object（这个接口只有一个方法getObject())。这个接口一般被用来，包装一个factory，通过个这工厂来返回一个新实例（prototype类型)。这个接口和FactoryBean有点像，但FactoryBean的实现是被当做一个SPI（Service Provider Interface)实例来使用在BeanFactory里面；ObjectFactory的实现一般被用来注入到其它Bean中，作为API来使用。就像ObjectFactoryCreatingFactoryBean的例子，它的返回值就是一个ObjectFactory，这个ObjectFactory被注入到了Bean中，在Bean通过这个接口的实例，来取得我们想要的Bean。
* 总的来说，FactoryBean和ObjectFactory都是用来取得Bean，但使用的方法和地方不同，FactoryBean被配置好后，Spring调用getObject()方法来取得Bean，ObjectFactory配置好后，在Bean里面可以取得ObjectFactory实例，需要我们手动来调用getObject()来取得Bean。

#### InitializingBean

* InitializingBean接口为bean提供了初始化方法的方式，它只包括afterPropertiesSet方法，凡是继承该接口的类，在初始化bean的时候会执行该方法。

#### doGetBean

* 中有三个方法非常关键：getSingleton，createBean和getObjectForBeanInstance。
* 逻辑梳理：
  + 1)getSingleton(beanName)
* 直接尝试从缓存获取或者从singletonFactories中的ObjectFactory中获取sharedInstance。
* 如果已经创建过了，那么调用getObjectForBeanInstance返回对应的实例（从缓存中只得到了bean的原始状态，还需要对bean进行实例化)。
  + 2)没有创建过，那么检查BeanFactory中是否包含该beanName对应的beanDefinition，没有找到则从父BeanFactory中继续查找，直至找到为止（递归)。
  + 3)如果存在依赖的bean，那么递归实例化所依赖的bean
  + 4)根据scope分别处理：
  + 4.1) 如果是singleton，那么调用getSingleton(beanName,objectFactory)（记录加载状态，通过this.singletonsCurrentlyInCreation.add(beanName)将当前正在创建的bean记录在缓存中，这样便可以对循环依赖进行检测)得到sharedInstance，其中调用了createBean、之后调用getObjectForBeanInstance返回对应的实例。
  + 4.2)如果是prototype，那么调用createBean，之后调用getObjectForBeanInstance。
  + 4.3) 如果是其他的scope，那么调用scope#get(beanName,objectFactory)得到scopedInstance，其中调用了createBean，之后调用getObjectForBeanInstance。
  + 5)createBean方法（创建单例或多例的bean)：
* AbstractBeanDefinition#prepareMethodOverrides（验证及准备覆盖的方法，决定实例化策略->反射 or CGLIB)
* 如果beanDefinition#getMethodOverrides()为空，即用户没有使用replace或lookup的配置方法，那么直接使用反射的方式；如果使用了，需要将这两个配置通过的功能切入进去，所以就必须要使用CGLIB动态代理的方式将包含两个特性所对应的逻辑的拦截增强器设置进去，这样才可以保证在调用方法的时候会被相应的拦截器增强，返回值为包含拦截器的代理实例。
* 给BeanPostProcessor一个机会来返回代理来替代真正的实例，与AOP有关。如果该方法返回bean不为空，则跳过后续实际创建bean的过程，直接返回代理后的bean。
* doCreateBean
  + 1)如果是单例，则需要首先清除缓存factoryBeanInstanceCache
  + 2)实例化bean，将BeanDefinition转换为BeanWrapper。
* 转换过程：
* ①如果存在工厂方法（factory-method)，则使用工厂方法进行初始化（instantiateUsingFactoryMethod)
* ②一个类有多个构造函数，每个构造函数都有不同的参数，所以需要根据参数锁定构造函数并进行初始化（autowireConstructor)
* ③如果既不存在工厂方法也不存在带有参数的构造函数，则使用默认的构造函数进行bean的初始化（instantiateBean)。
  + 3)MergedBeanDefinitionPostProcessor的应用
* bean合并后的处理，Autowired注解正是通过此方法实现诸如类型的预解析。
  + 4)更新singletonFactories
  + 5)属性填充populateBean
* 属性填充时，会autowireByName（按名获取待注入的属性)或autowireByType（按类型获取待注入的属性)
* 填充完毕后会调用客户定制的初始化方法。除了使用配置init-method外，还可以使自定义的bean实现InitializingBean接口，并在afterPropertiesSet中实现自己的初始化业务逻辑。
* init-method和afterPropertiesSet都是在ininitalBean时执行，执行顺序是afterPropertiesSet先执行，init-method后执行，在它们前后分别执行BeanPostProcesser的两个方法。
  + 6)代理bean循环依赖检查
  + 7)注册DisposableBean
* 如果配置了destroy-method，这里需要注册以便于在销毁时候调用。
  + 6)getObjectForBeanInstance
* 无论是从缓存中获取到的bean还是通过不同的scope策略加载的bean都只是最原始的bean状态，并不一定是我们最终想要的bean。
* 比如，我们需要对FactoryBean进行处理，那么这里得到的其实是FactoryBean的初始状态，但是我们真正需要的是FactoryBean中定义的factory-method（getObject方法)方法中返回的bean，而getObjectForBeanInstance就是完成这个工作的。

## 注入方式

* Spring的IOC有三种注入方式 ：
* 第一是根据属性注入，也叫setter注入；
* 第二种是根据构造方法进行注入；
* 第三种是工厂注入，分为静态工厂注入和动态工厂注入
* <bean id="car1" class="com.lzj.spring.beans.factory.StaticFactory"
* factory-method="getCar">
* 在XML文件中配置bean时，要指定class的属性为工厂的类；factory-method属性指定工厂类中工厂方法，用于创建bean；constrctor-arg用于给工厂方法传递参数。

## 循环依赖

* Spring容器循环依赖包括构造器循环依赖和setter循环依赖。
* 核心就是singletonObjects、singletonFactories、earlySingletonObjects和singletonsCurrentlyInCreation四个集合。
  + 1)singletonObjects是beanName与beanInstance的Map，是真正的缓存，beanInstance是构造完毕的，凡是正常地构造完毕的单例bean都会放入缓存中。
  + 2)earlySingletonObjects也是beanName与beanInstance的Map，beanInstance是已经调用了createBean方法，但是没有清除加载状态和加入至缓存的bean。仅在当前bean创建时存在，用于检测代理bean循环依赖。
  + 3)singleFactories是beanName与ObjectFactory的Map，仅在当前bean创建时存在，是尚未调用createBean的bean。用于setter循环依赖时实现注入。
  + 4)singletonsCurrentlyInCreation是beanName的集合，用于检测构造器循环依赖。
* getBean在循环依赖时所执行的步骤是这样的：
  + 1)检测当前bean是否在singletonObjects中，在则直接返回缓存好的bean；不在则检测是否在singletonFactories中，在，则调用其getObject方法，返回，并从singletonFactories中移除，加入到earlySingletonObjects中。
  + 2)正常创建，beforeSingletonCreation:检测当前bean是否在singletonsCurrentlyInCreation，如果存在，抛出异常。表示存在构造器循环依赖。如果不存在，则将当前bean加入。
  + 3)bean初始化，分为构造方法初始化、工厂方法初始化和简单初始化。如果是构造方法初始化，那么递归地获取参数bean。其他情况不会递归获取bean。
  + 4)addSingletonFactory:如果当前bean不在singletonObjects中，则将当前bean加入到singletonFactories中，并从earlySingletonObjects中移除。
  + 5)填充属性，简单初始化的话会递归创建所依赖的bean。
  + 6)调用用户初始化方法，比如BeanPostProcesser、InitializingBean、init-method，有可能返回代理后的bean。
  + 6) 检测循环依赖，如果当前bean在singletonObjects中，则判断当前bean(current bean)与singletonObjects中的bean(cached bean)是否是同一个，如果不是，那么说明当前bean是被代理过的，由于依赖当前bean的bean持有的是对cached bean的引用，这是不被允许的，所以会抛出BeanCurrentlyInCreationException异常。
  + 7)afterSingletonCreation:将当前bean从singletonsCurrentlyInCreation中删除
  + 8)addSingleton:将当前bean加入到singletonObjects，然后从singletonFactories, earlySingletonObjects中移除，结束

### 构造器循环依赖

* 表示通过构造器注入构成的循环依赖，此依赖是无法解决的，只能抛出BeanCurrentlyInCreationException异常表示循环依赖。
* 1、Spring容器创建单例“A” Bean，首先检测singletonFactories是否包含A，发现没有，于是正常创建，然后检测A是否包含在singletonsCurrentlyInCreation中，没有，则将A放入。构造方法初始化时需要B实例（A尚未放入到singletonFactories中)，于是调用了getBean(B)方法、
* 2、Spring容器创建单例“B” Bean，首先检测singletonFactories是否包含B，发现没有，于是正常创建，然后检测B是否包含在singletonsCurrentlyInCreation中，没有，则将B放入。构造方法初始化时需要C实例（B尚未放入到singletonFactories中)，于是调用了getBean(C)方法、
* 3、Spring容器创建单例“C” Bean，首先检测singletonFactories是否包含C，发现没有，于是正常创建，然后检测C是否包含在singletonsCurrentlyInCreation中，没有，则将C放入。构造方法初始化时需要A实例（C尚未放入到singletonFactories中)，于是调用了getBean(A)方法、
* 4、Spring容器创建单例“A” Bean，首先检测singletonFactories是否包含A，发现没有于是正常创建，然后检测A是否包含在singletonsCurrentlyInCreation中，有，抛出BeanCurrentlyInCreationException异常。

### setter循环依赖

* 表示通过setter注入方式构成的循环依赖。
* 对于setter注入造成的依赖是通过Spring容器提前暴露刚完成构造器注入但未完成其他步骤（如setter注入)的Bean来完成的，而且只能解决单例作用域的Bean循环依赖。
* 1、Spring容器创建单例“A” Bean，首先检测singletonFactories是否包含A，发现没有，于是正常创建，然后检测A是否包含在singletonsCurrentlyInCreation中，没有，则将A放入。将A放入到singletonFactories中。注入属性时需要B实例，于是调用了getBean(B)方法、
* 2、Spring容器创建单例“B” Bean，首先检测singletonFactories是否包含B，发现没有，于是正常创建，然后检测B是否包含在singletonsCurrentlyInCreation中，没有，则将B放入。将B放入到singletonFactories中。注入属性时需要C实例，于是调用了getBean(C)方法、
* 3、Spring容器创建单例“C” Bean，首先检测singletonFactories是否包含C，发现没有，于是正常创建，然后检测C是否包含在singletonsCurrentlyInCreation中，没有，则将C放入。将C放入到singletonFactories中。注入属性时需要A实例，于是调用了getBean(A)方法、
* 4、Spring容器创建单例“A” Bean，首先检测singletonFactories是否包含A，发现有，于是返回缓存了的bean，并将A从singletonFactories删除，返回A实例。
* 5、C得到A实例。set进来，B、A也是这样。结束。
* 对于“prototype”作用域Bean，Spring容器无法完成依赖注入，因为“prototype”作用域的Bean，Spring容器不进行缓存，因此无法提前暴露一个创建中的Bean。

## 常用注解

- 1)将使用了以下注解的类纳入到Spring的容器管理中。

* @Component
* @Controller
* @Service
* @Repository
* @Scope注解 作用域
  + 2)注入注解
* @Resource 默认按名称装配（autowireByName)，如果没有指定name属性，当注解写在字段上时，默认取字段名。当找不到与名称匹配的bean时才按照类型进行装配。但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配。
* @Autowired 默认按类型装配（autowireByType)，也可以搭配@Qualifier实现按名称装配。

## Bean的作用域

* IOC容器中bean默认是单例的
* 可以使用scope配置作用域
* 作用域有两种：singleton，prototype（每次获取都创建一个新的bean)

## Bean生命周期

* 有三种方式在Bean初始化后和销毁前添加一些操作。
* Bean的方法加上@PostConstruct和@PreDestroy注解（必须有component-scan才有效)
* 在xml中定义init-method和destory-method方法
* Bean实现InitializingBean和DisposableBean接口
* 其执行顺序关系为：
* Bean在实例化的过程中：constructor > @PostConstruct >InitializingBean > init-method
* Bean在销毁的过程中：@PreDestroy > DisposableBean > destroy-method
* 还可以通过BeanPostProcessor在bean初始化前后添加一些操作。
* 初始化执行顺序为：constructor > BeanPostProcessor#postProcessBeforeInitialization > @PostConstructor > InitializingBean > init-method > BeanPostProcessor# postProcessAfterInitialization
* 调用用户自定义初始化方法之前和之后分别会调用BeanPostProcessor的postProcessBeforeInitialization和postProcessAfterInitialization方法。
* 参见getBean中的initializeBean方法。

### 示例

public class Car implements InitializingBean, DisposableBean {  
  
 public Car() {  
 System.out.println("constructor...");  
 }  
  
  
 public void initCar() {  
 System.out.println("init-method...");  
 }  
  
 @PostConstruct  
 public void postConstructMethod() {  
 System.out.println("post construct ..");  
 }  
  
 @PreDestroy  
 public void beforeDestroy() {  
 System.out.println("before destroy...");  
 }  
  
 public void destroyCar() {  
 System.out.println("destroy-method....");  
 }  
  
   
 @Override  
 public void destroy() throws Exception {  
 System.out.println("disposable bean...");  
 }  
  
 @Override  
 public void afterPropertiesSet() throws Exception {  
 System.out.println("initializing bean...");  
 }  
}

public class MyBeanPostProcessor implements BeanPostProcessor{  
  
 @Override  
 public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
 System.out.println("postProcessAfterInitialization "+ bean +" ,"+ beanName);  
 return bean;  
 }  
  
 @Override  
 public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  
 System.out.println("postProcessBeforeInitialization "+ bean +" ,"+ beanName);  
 return bean;  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("beans-cycle.xml");  
 Car car = (Car) ctx.getBean("car");  
 System.out.println(car);  
 ctx.close();  
 }  
}

* <context:component-scan base-package="cycle" ></context:component-scan>
* 输出为：
* constructor...
* postProcessBeforeInitialization cycle.Car@2db7a79b ,car
* post construct ..
* initializing bean...
* init-method...
* postProcessAfterInitialization cycle.Car@2db7a79b ,car
* cycle.Car@2db7a79b
* // 关闭了
* before destroy...
* disposable bean...
* destroy-method....

## 自动装配

* <beans xmlns="http://www.springframework.org/schema/beans"
* xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
* xmlns:p="http://www.springframework.org/schema/p"
* xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">
* <bean id="car" class="autowire.Car"
* p:brand="audi" p:price="300000">
* <bean id="address" class="autowire.Address"
* p:city="Nanjing" p:street="Xianlin">
* <bean id="person" class="autowire.Person"
* p:name="NewSong" p:car-ref="car" p:addr-ref="address">
* 不使用自动装配
* 可以使用p命名空间中的autowire属性，它的值是byName或者byType
* 如果是byName，就会根据当前这个Bean的类的属性名（由setter分隔出来的)在xml文件中找同名（id)的bean，如果是，那么就匹配。
* 因此，我们推荐将id设置为首字母小写的类名（类的属性也是如此)
* 并且这个autowire可以将多个符合的装配，不仅仅是匹配一个；如果没有匹配的就不装配。
* 如果是byType，就会根据Bean的类型和当前Bean的属性的类型进行匹配，前提是每个类只可以对应一个Bean。如果有重名，那么会出错。
* 而对byName而言，可以修改setter的方法名，使得由setter分隔出来的属性名与id一致。
* autowire一旦使用，就会对所有的引用属性使用自动装配；不能同时使用byType和byName.
* 实际上很少使用自动装配
* <bean id="car" class="autowire.Car"
* p:brand="audi" p:price="300000">
* <bean id="address" class="autowire.Address"
* p:city="Nanjing" p:street="Xianlin">
* <bean id="person" class="autowire.Person"
* p:name="NewSong" autowire="byName">
* 注意Person持有了Car和Address，只在Person上加autowire即可

# AOP

## 简介

## AOP术语

* 1、切面（aspect/advisor)
* 类是对物体特征的抽象，切面就是对横切关注点的抽象。组合了Pointcut与Advice，在Spring中有时候也称为Advisor。
* 2、连接点（join point)
* 被拦截到的点，因为Spring只支持方法类型的连接点，所以在Spring中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器。
* 3、切入点（pointcut)
* 描述的一组符合某个条件的join point，通常使用pointcut表达式来限定join point。
* 4、通知（advice)
* 所谓通知指的就是指拦截到连接点之后要执行的代码，通知分为前置、后置、异常、返回、环绕通知五类。
* 5、目标对象
* 代理的目标对象
* 6、织入（weave)
* 将Advice织入join point的这个过程
* 7、引介（introduction)
* 在不修改代码的前提下，引介可以在运行期为类动态地添加一些方法或字段

## Advisor

* 通知Advice是Spring提供的一种切面(Aspect)。但其功能过于简单，只能将切面织入到目标类的所有目标方法中，无法完成将切面织入到指定目标方法中。
* 顾问Advisor是Spring提供的另一种切面。其可以完成更为复杂的切面织入功能。PointcutAdvisor是顾问的一种，可以指定具体的切入点。顾问对通知进行了包装，会根据不同的通知类型，在不同的时间点，将切面织入到不同的切入点。
* Advisor组合了Pointcut与Advice。
* 除了引介Advisor外，几乎所有的advisor都是PointcutAdvisor。

public interface Advisor {  
 Advice getAdvice();  
 /\*\*  
 \* @return whether this advice is associated with a particular target instance  
 \*/  
 boolean isPerInstance();  
}

public interface PointcutAdvisor extends Advisor {  
  
 /\*\*  
 \* Get the Pointcut that drives this advisor.  
 \*/  
 Pointcut getPointcut();  
  
}

## Advice

public interface Advice {  
  
}

* 增强（advice)主要包括如下五种类型
  1. 前置增强(BeforeAdvice)：在目标方法执行前实施增强
  2. 后置增强(AfterAdvice)：在目标方法执行后（无论是否抛出遗产)实施增强
  3. 环绕增强(MethodInterceptor)：在目标方法执行前后实施增强
  4. 异常抛出增强(ThrowsAdvice)：在目标方法抛出异常后实施增强
  5. 返回增强（AfterReturningAdvice)：在目标方法正常返回后实施增强
  6. 引介增强(IntroductionIntercrptor)：在目标类中添加一些新的方法和属性

## JDK动态代理与CGLIB代理

* JDK动态代理：
* 其代理对象必须是某个接口的实现，它是通过在运行时创建一个接口的实现类来完成对目标对象的代理
* CGLIB代理：在运行时生成的代理对象是针对目标类扩展的子类。
* CGLIB是高效的代码生产包，底层是依靠ASM操作字节码实现的，性能比JDK强。
* 相关标签
* <aop:aspectj-autoproxy proxy-target-class=”true”/>
* true表示使用CGLIB代理。

## 源码

### 解析AOP标签

* <aop:aspectj-autoproxy />
* 解析配置文件时，一旦遇到aspectj-autoproxy注解时就会使用解析器
* AspectJAutoProxyBeanDefinitionParser进行解析。
* 解析结果是注册了一个bean：AnnotationAwareAspectJAutoProxyCreator。

### 创建AOP代理

* 针对于实现了InstantiationAwareBeanPostProcessor接口的BeanPostProcessor，在doCreateBean之前调用，如果需要进行代理，则直接返回代理后的bean。
* 前面解析AOP标签时注册了AnnotationAwareAspectJAutoProxyCreator，它继承了AbstractAdvisorAutoProxyCreator，是一个InstantiationAwareBeanPostProcessor。

### wrapIfNecessary

* 逻辑：
  + 1)获取可以应用到该bean的所有advisor
  + 2)创建代理
* AbstractAutoProxyCreator.createProxy
* 在获取了所有对应的bean的增强后，便可以进行代理的创建了。
* 逻辑：
  + 1)获取当前类中的属性
  + 2)添加代理接口
  + 3)封装Advisor并加入到ProxyFactory中
  + 4)设置proxyFactory的target要代理的类
  + 5)进行获取代理操作 ProxyFactory#getProxy
* createAopProxy().getProxy(classLoader);
* 如果目标对象实现了接口，默认情况下会使用JDK的动态代理
* 如果目标对象实现了接口，可以强制使用CGLIB（proxy-target-class=false)
* 如果目标对象没有实现接口，必须采用CGLIB

# Transaction

## 事务

* 编程式事务：所谓编程式事务指的是通过编码方式实现事务，即类似于JDBC编程实现事务管理。管理使用TransactionTemplate或者直接使用底层的PlatformTransactionManager。对于编程式事务管理，Spring推荐使用TransactionTemplate。
* 声明式事务：管理建立在AOP之上的。其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，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通过编程的方式管理事务，这样就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(或通过基于@Transactional注解的方式)，便可以将事务规则应用到业务逻辑中。
* 显然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。
* 声明式事务管理使业务代码不受污染，一个普通的POJO对象，只要加上注解就可以获得完全的事务支持。和编程式事务相比，声明式事务唯一不足地方是，后者的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。但是即便有这样的需求，也存在很多变通的方法，比如，可以将需要进行事务管理的代码块独立为方法等等。
* AOP可以帮助实现声明式事务，将事务管理从业务代码中分离
* • Spring 从不同的事务管理 API 中抽象了一整套的事务机制. 开发人员不必了解底层的事务 API, 就可以利用这些事务机制. 有了这些事务机制, 事务管理代码就能独立于特定的事务技术了.
* • Spring 的核心事务管理抽象是PlatformTransactionManager，它为事务管理封装了一组独立于技术的方法. 无论使用 Spring 的哪种事务管理策略(编程式或声明式), 事务管理器都是必须的。

## 事务管理器

* 这些都是PlatformTransactionManager

## 使用注解声明式管理事务

•- 除了在带有切入点, 通知和增强器的 Bean 配置文件中声明事务外, Spring 还允许简单地用 @Transactional 注解来标注事务方法.   
•- 为了将方法定义为支持事务处理的, 可以为方法添加 @Transactional 注解. 根据 Spring AOP 基于代理机制, 只能标注公有方法.  
•- 可以在方法或者类级别上添加 @Transactional 注解. 当把这个注解应用到类上时, 这个类中的所有公共方法都会被定义成支持事务处理的.   
•- 在 Bean 配置文件中只需要启用 <tx:annotation-driven> 元素, 并为之指定事务管理器就可以了.   
•- 如果事务处理器的名称是 transactionManager, 就可以在<tx:annotation-driven> 元素中省略 transaction-manager 属性. 这个元素会自动检测该名称的事务处理器.

## 事务隔离级别

* 设置回滚事务属性
* 默认情况下只对RuntimeException和Error回滚
* 继承自Exception的不会回滚
* rollbackFor和noRollbackFor可以设置对某些特定异常是否回滚
* 值是一个Class对象的数组

## 事务传播行为

* 当事务方法被另一个事务方法调用时, 必须指定事务应该如何传播. 例如: 方法可能继续在现有事务中运行, 也可能开启一个新事务, 并在自己的事务中运行.
* 事务的传播行为可以由传播属性指定. Spring 定义了 7种类传播行为.

### PROPAGATION\_REQUIRED（重要)

* 如果当前存在一个事务，则加入当前事务；如果不存在任何事务，则创建一个新的事务。总之，要至少保证在一个事务中运行。PROPAGATION\_REQUIRED通常作为默认的事务传播行为。
* REQUIRED（内层事务newTransaction为false，内层事务回滚时仅设置回滚标记，外层事务进行外层回滚)

### PROPAGATION\_SUPPORTS

* 如果当前存在一个事务，则加入当前事务；如果当前不存在事务，则直接执行。 对于一些查询方法来说，PROPAGATION*SUPPORTS通常是比较合适的传播行为选择。 如果当前方法直接执行，那么不需要事务的支持；如果当前方法被其他方法调用，而其他方法启动了一个事务的时候，使用PROPAGATION*SUPPORTS可以保证当前方法能够加入当前事务并洞察当前事务对数据资源所做的更新。 比如说，A.service()会首先更新数据库，然后调用B.service()进行查询，那么，B.service()如果是PROPAGATION*SUPPORTS的传播行为， 就可以读取A.service()之前所做的最新更新结果，而如果使用稍后所提到的PROPAGATION*NOT\_SUPPORTED，则B.service()将无法读取最新的更新结果，因为A.service()的事务在这个时候还没有提交(除非隔离级别是read uncommitted)。

### PROPAGATION\_MANDATORY

* PROPAGATION\_MANDATORY强制要求当前存在一个事务，如果不存在，则抛出异常。 如果某个方法需要事务支持，但自身又不管理事务提交或者回滚的时候，比较适合使用
* PROPAGATION\_MANDATORY。

### PROPAGATION*REQUIRES*NEW（重要，两事务完全独立，独立提交、独立回滚)

* 不管当前是否存在事务，都会创建新的事务。如果当前存在事务的话，会将当前的事务挂起(suspend)。 如果某个业务对象所做的事情不想影响到外层事务的话，PROPAGATION*REQUIRES*NEW应该是合适的选择，比如，假设当前的业务方法需要向数据库中更新某些日志信息， 但即使这些日志信息更新失败，我们也不想因为该业务方法的事务回滚而影响到外层事务的成功提交，因为这种情况下，当前业务方法的事务成功与否对外层事务来说是无关紧要的。
* REQUIRES\_NEW（内外层事务平级，内层事务newTransaction为true，suspend外层事务，抛出异常后内层事务进行内层回滚，resume外层事务，外层事务捕获到内层抛出的异常后进行外层回滚)

### PROPAGATION*NOT*SUPPORTED

* 不支持当前事务，而是在没有事务的情况下执行。如果当前存在事务的话，当前事务原则上将被挂起(suspend)，但要依赖于对应的PlatformTransactionManager实现类是否支持事务的挂起(suspend)，更多情况请参照TransactionDefinition的javadoc文档。 PROPAGATION*NOT*SUPPORTED与PROPAGATION*SUPPORTS之间的区别，可以参照PROPAGATION*SUPPORTS部分的实例内容。

### PROPAGATION\_NEVER

* 永远不需要当前存在事务，如果存在当前事务，则抛出异常。

### PROPAGATION\_NESTED（重要，内部事务是外部事务的子事务。子事务出现异常会回滚到savepoint，外部事务提交/回滚时子事务也进行提交/回滚)

* 如果存在当前事务，则在当前事务的一个嵌套事务中执行，否则与PROPAGATION*REQUIRED的行为类似，即创建新的事务，在新创建的事务中执行。 PROPAGATION*NESTED粗看起来好像与PROPAGATION*REQUIRES*NEW的行为类似，实际上二者是有差别的。 PROPAGATION*REQUIRES*NEW创建的新事务与外层事务属于同一个“档次”，即二者的地位是相同的，当新创建的事务运行的时候，外层事务将被暂时挂起(suspend)； 而PROPAGATION\_NESTED创建的嵌套事务则不然，它是寄生于当前外层事务的，它的地位比当前外层事务的地位要小一号，当内部嵌套事务运行的时候，外层事务也是处于active状态。是已经存在事务的一个真正的子事务. 嵌套事务开始执行时, 它将取得一个 savepoint. 如果这个嵌套事务失败, 我们将回滚到此 savepoint. 嵌套事务是外部事务的一部分, 只有外部事务结束后它才会被提交，外部事务回滚，它也会被回滚。
* NESTED（内外层事务嵌套，内层事务newTransaction为false，并创建还原点，抛出异常后rollback至还原点，外层事务捕获到内层抛出的异常后进行外层回滚)

## 超时和只读属性

•- 由于事务可以在行和表上获得锁, 因此长事务会占用资源, 并对整体性能产生影响.   
•- 如果一个事物只读取数据但不做修改, 数据库引擎可以对这个事务进行优化.  
•- 超时事务属性: 事务在强制回滚之前可以保持多久. 这样可以防止长期运行的事务占用资源.  
•- 只读事务属性: 表示这个事务只读取数据但不更新数据, 这样可以帮助数据库引擎优化事务.

* 如果你一次执行单条查询语句，则没有必要启用事务支持，数据库默认支持SQL执行期间的读一致性；
* 如果你一次执行多条查询语句，例如统计查询，报表查询，在这种场景下，多条查询SQL必须保证整体的读一致性，否则，在前条SQL查询之后，后条SQL查询之前，数据被其他用户改变，则该次整体的统计查询将会出现读数据不一致的状态，此时，应该启用事务支持
* read-only="true"表示该事务为只读事务，比如上面说的多条查询的这种情况可以使用只读事务，由于只读事务不存在数据的修改，因此数据库将会为只读事务提供一些优化手段，例如Oracle对于只读事务，不启动回滚段，不记录回滚log。
  + （1)在JDBC中，指定只读事务的办法为： connection.setReadOnly(true);
  + （2)在Hibernate中，指定只读事务的办法为： session.setFlushMode(FlushMode.NEVER);
* 此时，Hibernate也会为只读事务提供Session方面的一些优化手段
  + （3)在Spring的Hibernate封装中，指定只读事务的办法为： bean配置文件中，prop属性增加“read-Only”或者用注解方式@Transactional(readOnly=true)
* Spring中设置只读事务是利用上面两种方式（根据实际情况)
* 在将事务设置成只读后，相当于将数据库设置成只读数据库，此时若要进行写的操作，会出现错误。

## 源码

### 解析事务标签

* <tx:annotation-driven />
* 同AOP标签，需要一个对应的BeanDefinitionParser，AnnotationDrivenBeanDefinitionParser。
* InfrastructureAdvisorAutoProxyCreator作为一个AbstractAutoProxyCreator，会在getBean时调用其postProcessAfterInstantiation方法，该方法会创建事务代理。
* 我们之前注册了BeanFactoryTransactionAttributeSourceAdvisor这个类，这个类实现了Advisor接口。BeanFactoryTransactionAttributeSourceAdvisor作为一个advisor，用于对事务方法进行增强。只要类或方法实现了@Transactional接口，该Advisor一定会被加到拦截器链中，对原方法进行事务增强。
* 返回的Advisor类型是BeanFactoryTransactionAttributeSourceAdvisor，而其beanName是TransactionInterceptor。
* 如果方法中存在事务属性（TransactionAttribute)，则使用方法上的属性，否则使用方法所在的类上的属性。如果方法所在类的属性上还是没有搜寻到对应的事务属性，那么再搜寻接口中的方法，再没有的话，最后尝试搜寻接口的类上面的声明。
* 解析@Transactional中的各个属性，并封装到TransactionAttribute中返回。

### 创建事务代理

* 对于标记了@Transactional的方法而言，会被代理，增强事务功能。
* 这些方法的Advisor增强中包括了TransactionInterceptor
* （BeanFactoryTransactionAttributeSourceAdvisor对应的bean)。
* TransactionInterceptor支撑着整个事务功能的架构，它继承了MethodInterceptor。
* TransactionAspectSupport#invokeWithinTransaction（最主要的方法)
* 逻辑：
  + 1)获取事务属性 TransactionAttribute
  + 2)加载配置中的TransactionManager
  + 3)不同的事务处理方式使用不同的逻辑，就声明式事务而言，会获取方法信息并创建事务信息TransactionInfo（此时已经创建了事务)
* 事务信息（TransactionInfo)与事务属性（TransactionAttribute)并不相同。
* 前者包含了后者，且包含了其他事务信息，比如PlatformTransactionManager以及TransactionStatus相关信息。
  + 4)try:执行原始方法
  + 5)catch:异常，回滚事务，再次抛出异常，7)及以后的不会执行
  + 6)finally:清除事务信息
  + 7)提交事务
  + 8)返回原始方法的返回值

#### 开启事务

* 逻辑：
  + 1)获取事务，创建对应的事务实例
  + 2)如果当前线程存在事务，那么根据传播行为进行相应处理，处理完成后返回
* 值得注意的有两点：
  + 1)REQUIRES\_NEW表示当前方法必须在它自己的事务里运行，一个新的事务将被启动。而如果有一个事务正在运行的话，则在这个方法运行期间被挂起（suspend)。
  + 2)NESTED表示如果当前正在有一个事务在运行中，则该方法应该运行在一个嵌套的事务中，被嵌套的事务可以独立于封装事务进行提交或者回滚。如果封装事务不存在，行为就像REQUIRES\_NEW。
* Spring主要有两种处理NESTED的方式：
* 首选设置保存点的方式作为异常处理的回滚
* JTA无法使用保存点，那么处理方式和REQUIRES\_NEW相同，而一旦出现异常，则由Spring的事务异常处理机制去完成后续操作。
  + 3)事务超时的验证
  + 4)事务传播行为的验证
  + 5)构建DefaultTransactionStatus，创建当前事务的状态
  + 6)完善transaction，包括设置ConnectionHolder、隔离级别、timeout，如果是新连接，则绑定到当前线程
  + 7)将事务信息记录在当前线程中。

#### 回滚事务

* 一旦事务执行失败，Spring会通过TransactionInfo实例来进行回滚等后续工作。
  + 1)默认情况下判断是否回滚默认的依据是 抛出的异常是否是RuntimeException或者是Error的类型
  + 2)如果不满足回滚条件，即使抛出异常也会提交。
  + 3)有保存点则回滚到保存点，是新事务则回滚整个事务；存在事务又不是新事务，则做回滚标记，等到事务链执行完毕后统一回滚。
  + 4)将挂起事务恢复。

#### 提交事务

* 某个事务是另一个事务的嵌入事务，但是这些事务又不在Spring的管理范围之内，或者无法设置保存点，那么Spring会通过设置回滚标识的方式来禁止提交。首先当某个嵌入事务发生回滚的时候会设置回滚标识，而等到外部事务提交时，一旦判断出当前事务流被设置了回滚标识，则由外部事务来统一进行整体事务的回滚。
* 在提交过程中也不是直接提交的，而是考虑了诸多方面。
* 符合提交的条件如下：
* 当事务状态中有保存点信息的话便不会提交事务；
* 当事务不是新事务的时候也不会提交事务
* 原因是：
* 对于内嵌事务，在Spring中会将其在开始之前设置保存点，一旦内嵌事务出现异常便根据保存点信息进行回滚，但是如果没有出现异常，内嵌事务并不会单独提交，而是根据事务流由最外层事务负责提交，所以如果当前存在保存点信息便不是最外层事务，不做提交操作。

# MVC

## 面试题

### 与Struts的区别

* SpringMVC的入口是一个Servlet即前端控制器，而Struts2入口是一个Filter过虑器。
* SpringMVC是基于方法开发(一个url对应一个方法)，请求参数传递到方法的形参，可以设计为单例或多例(建议单例)，struts2是基于类开发，传递参数是通过类的属性，只能设计为多例。

### SpringMVC的控制器是不是单例模式,如果是,有什么问题,怎么解决

* 是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方案是在控制器里面不能写字段

### SpingMVC中的控制器的注解一般用那个,有没有别的注解可以替代

* 一般用@Conntroller注解,表示是表现层,不能用用别的注解代替.

### @RequestMapping注解用在类上面有什么作用

* 用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。

### 怎么样把某个请求映射到特定的方法上面

* 直接在方法上面加上注解@RequestMapping,并且在这个注解里面写上要拦截的路径

### 如果在拦截请求中,我想拦截get方式提交的方法,怎么配置

* 可以在@RequestMapping注解里面加上method=RequestMethod.GET

### 如果在拦截请求中,我想拦截提交参数中包含"type=test"字符串,怎么配置

* 可以在@RequestMapping注解里面加上params="type=test"

### 我想在拦截的方法里面得到从前台传入的参数,怎么得到

* 直接在形参里面声明这个参数就可以,但必须名字和传过来的参数一样

### 如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象

* 直接在方法中声明这个对象,SpringMVC就自动会把属性赋值到这个对象里面

### 怎么样在方法里面得到Request,或者Session

* 直接在方法的形参中声明request,SpringMVC就自动把request对象传入

### SpringMVC中函数的返回值是什么.

* 返回值可以有很多类型,有String, ModelAndView,当一般用String比较好

### SpringMVC怎么处理返回值的

* SpringMVC根据配置文件中InternalResourceViewResolver的前缀和后缀,用前缀+返回值+后缀组成完整的返回值

### SpringMVC怎么样设定重定向和转发的

* 在返回值前面加"forward:"就可以让结果转发,譬如"forward:user.do?name=method4" 在返回值前面加"redirect:"就可以让返回值重定向,譬如"redirect:http://www.baidu.com"
* forward/redirect:除了跳转至某个页面，也可以跳转至某个controller处理方法。
* 并且使用时需要以/开头, 默认是相对于项目的根目录的，不加就是相对于当前controller的@RequestMapping值的路径（没有这个注解就是相对于项目根目录)。

### SpringMVC用什么对象从后台向前台传递数据的

* 通过ModelMap对象,可以在这个对象里面用put方法,把对象加到里面,前台就可以通过el表达式拿到

### SpringMVC中有个类把视图和数据都合并的一起的,叫什么

* 叫ModelAndView

### 怎么样把ModelMap里面的数据放入Session里面

* 可以在类上面加上@SessionAttributes注解,里面包含的字符串就是要放入session里面的key

### SpringMVC怎么和AJAX相互调用的

* 通过Jackson框架就可以把Java里面的对象直接转化成Js可以识别的Json对象
* 具体步骤如下
* 1.加入Jackson.jar
* 2.在配置文件中配置json的映射
* 3.在接受Ajax方法里面可以直接返回Object,List等,但方法前面要加上@ResponseBody注解

### 当一个方法向AJAX返回特殊对象,譬如Object,List等,需要做什么处理

* 要加上@ResponseBody注解

## Mapping

### @RequestMapping（映射)

* RequestMapping是根据该访问路径来决定调用哪个方法来处理，访问路径是由类上的RequestMapping+方法上的RequestMapping决定的。
* 返回值是转发到哪个路径下，转发路径是由 prefix+返回值+suffix决定的。

### @PathVariable（注入)

### @RequestParam（注入)

### @RequestHeader（注入)

### @CookieValue（注入)

### 使用POJO对象绑定请求参数值（注入)

### 使用Servlet原生API作为参数（注入)

### 数据绑定(注入)/类型转换/校验

#### WebDataBinder

#### 数据转换

#### 数据格式化

* 格式化和类型转换是同时进行的

#### 数据校验

#### HttpMessageConverter

* HttpMessageConverter使用

## Model

### ModelAndView

* 在handle方法中获取视图和模型，然后将域对象存放到视图和模型（ModelAndView)中。
* 默认是放到request域中。

### Map&Model

* 参数的真实类型为ExtendedModelMap对象的子类，既实现了Model接口，又继承了ModelMap类
* 该参数的类型也可以写为Model或者ModelMap，不过一般使用Map类型就足够了

## View

* 解析流程
* 无论返回值是一个ModelAndView、String、View，SpringMVC都会将其转为一个ModelAndView对象。
* 通过Controller实现URL映射得到ModelAndView，然后ViewResolver实例化View对象，得到物理视图，然后调用View的render方法，将Model数据渲染到View中，然后转发至View。

### 视图

* 每个请求都会创建一个新的视图对象

### 视图解析器

### InternalResourceViewResolver

* InternalResourceView
* InternalResourceViewResolver
* 这两个接口实现和JSP有关的渲染和转发功能。

### annotation-driven

* 前两个bean是与RequestMapping有关，第三个bean与异常处理有关。
* 还可以处理数据类型转换、数据格式化、数据校验、AJAX。
* 以下是DispatcherServlet的持有的handlerAdapters的数据情况：

### default-servlet-handler

## 异常处理

### HandlerExceptionResolver

### ExceptionHandlerExceptionResolver

* 可以把处理异常的代码放到一个类中，比如ExceptionControllerAdvice
* 为这个类加一个注解@ControllerAdvice
* 当在当前类找不到对应的注解了@ExceptionHandler的方法，那么将会在注解了@ControllerAdvice的类中注解了@ExceptionHandler方法去处理。

## 拦截器

### 使用

* 第一步：定义一个拦截器，实现HandlerInterceptor接口

public class FirstInterceptor implements HandlerInterceptor {

* @Override

public void afterCompletion(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2, Exception arg3)

* throws Exception {
* System.out.println("[FirstInterceptor] afterCompletion");
* }
* @Override

public void postHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2, ModelAndView arg3)

* throws Exception {
* System.out.println("[FirstInterceptor] postHandle");
* }
* @Override

public boolean preHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2) throws Exception {

* System.out.println("[FirstInterceptor] preHandle");
* return true;
* }
* }
* 第二步：在springmvc配置文件中进行如下配置：
* <mvc:interceptors>
* </mvc:interceptors>
* [FirstInterceptor] preHandle
* //controller处理方法被调用
* [FirstInterceptor] postHandle
* [FirstInterceptor] afterCompletion

### 顺序

* 类似于Filter，当返回true时，继续检测其他拦截器，返回false，直接调到放行过的拦截器的afterComplication方法。
* preHandle是在调用处理方法之前调用
* postHandle是在调用处理方法之后，渲染视图之前被调用。
* afterCompletion是在渲染视图之后被调用
* preHandle：权限；日志；事务等
* handle
* postHandle 修改request域的属性；修改视图
* render
* afterCompltion 释放资源
* 对于preHandle方法，按照拦截器的配置顺序的顺序执行
* 对于postHandle方法，按照拦截器的配置顺序的逆序执行
* afterComplication同postHandle方法

## 源码

* DispatcherServlet： SpringMVC总的拦截器
* HandlerMapping：请求和处理器之间的映射，用于获取HandlerExecutionChain
* HandlerExecutionChain：持有一组Interceptor和实际请求处理器HandlerAdapter，负责执行Interceptor的各个方法和处理方法。
* HandlerAdapter：实际的请求处理器，处理后返回ModelAndView
* HandlerExceptionResolver：异常处理器，当拦截器的postHandle方法调用后检查异常。
* ViewResolver：视图解析器,解析视图名，得到View，由逻辑视图变为物理视图。
* View ：有render方法，渲染视图
* 渲染完毕后调用转发

### 初始化ApplicationContext

* ContextLoaderListener（入口)
* ContextLoaderListener的作用就是启动Web容器时，自动装配ApplicationContext的配置信息。
* 因为它实现了ServletContextListener这个接口，在web.xml配置这个监听器，启动容器时，就会默认执行它实现的方法，使用ServletContextListener接口，开发者能够在为客户端请求提供服务之前向ServletContext中添加任意的对象。
* 在ContextLoaderListener中的核心逻辑是初始化WebApplicationContext实例并存放在ServletContext中。

public void contextInitialized(ServletContextEvent event) {  
 initWebApplicationContext(event.getServletContext());  
}

* 在初始化过程中，程序首先会读取ContextLoader类的同目录下的配置文件ContextLoader.properties，并根据其中的配置提取将要实现WebApplicationContext接口的实现类，并根据这个实现类通过反射的方式进行实例的创建。

private static final String DEFAULT\_STRATEGIES\_PATH = "ContextLoader.properties";

* 在ContextLoader目录下有一个配置文件ContextLoader.properties
* 内容为：
* org.springframework.web.context.WebApplicationContext=org.springframework.web.context.support.XmlWebApplicationContext

### 初始化DispatcherServlet

* 第一次访问网站时，会初始化访问到的servlet。
* 初始化阶段会调用servlet的init方法，在DispatcherServlet中是由其父类HttpServletBean实现的。
* 逻辑：
  + 1)封装及验证初始化参数 解析init-param并封装到PropertyValues中
  + 2)将DispatcherServlet转化为BeanWrapper实例
  + 3)注册相对于Resource的属性编辑器
  + 4)PropertyValues属性注入
  + 5)servletBean的初始化（initServletBean)
* FrameworkServlet#initServletBean调用了FrameworkServlet#initWebApplicationContext
* 创建或刷新WebApplicationContext实例，并对servlet功能所使用的变量进行初始化
* 逻辑：
  + 1)对已经创建的WebApplicationContext实例进行配置和刷新
* （configureAndRefreshWebApplicationContext)
* 最后会调用ApplicationContext中的refresh方法（与IOC的衔接)
  + 2)刷新Spring在Web功能实现中必须使用的全局变量（onRefresh)
* 初始化各个模块：
* // 初始化文件上传模块  
  initMultipartResolver(context);
* // 初始化国际化模块  
  initLocaleResolver(context);
* // 初始化主题模块  
  initThemeResolver(context);
* // 初始化HandlerMappings  
  initHandlerMappings(context);
* // 初始化HandlerAdapters  
  initHandlerAdapters(context);
* // 初始化HandlerExceptionResolvers  
  initHandlerExceptionResolvers(context);  
  initRequestToViewNameTranslator(context);
* // 初始化ViewResolvers  
  initViewResolvers(context);  
  initFlashMapManager(context);
* 重点注意初始化HandlerMappings和HandlerAdapters。
* 初始化HandlerMappings是将Controller的各个请求方法封装为HandlerMethod，注册到mappingRegistry（key是RequestMappingInfo，value是MappingRegistration，它封装了之前创建的handlerMethod)中。
* 初始化HandlerAdapters是获取所有实现了HandlerAdapter接口的类保存到成员变量handlerAdapters中。
* RequestMappingHandlerAdapter（请求映射处理器适配器)作为例子，初始化时会去初始化argumentResolvers、returnValueHandlers等。

### 处理请求

* FrameworkServlet#service（入口)所有请求都会调用processRequest方法。
* 逻辑：
  + 1)为了保证当前线程的LocaleContext和RequestAttributes可以在当前请求处理完毕后还能恢复，提取当前线程的两个属性
  + 2)根据当前request创建对应的LocaleContext和RequestAttributes，并绑定到当前线程
  + 3)委托给doService（DispatcherServlet#doService)方法进一步处理
  + 4)请求处理结束后恢复线程到原始状态
  + 5)请求处理结束后无论成功与否都会发布事件通知
* doService的主体是在doDispatch方法中进行。
* 逻辑：
  + 1)处理文件上传请求
  + 2)将handler实例（即HandlerMethod)和所有匹配的拦截器封装到HandlerExecutionChain mappedHandler中。如果没有找到，那么根据策略抛出异常或直接返回错误码。
  + 3)getHandlerAdapter（根据HandlerExecutionChain获取可以支持它的HandlerAdapter)
* 如果是HandlerMethod，那么返回的是RequestMappingHandlerAdapter。
  + 4)调用HandlerExecutionChain mappedHandler的applyPreHandle
  + 5)执行HandlerAdapter的handle方法，处理请求。
* 比如RequestMappingHandlerAdapter#handleInternal，
* 注入参数 getMethodArgumentValues
* 遍历请求参数，然后委托给HandlerMethodArgumentResolver#resolveArgument进行参数解析，先解析，后使用WebDataBinder进行数据转换（如有必要)
* 执行方法
* 处理返回值 如果返回值是普通的对象(@ResponseBody)，那么handler是
* RequestResponseBodyMethodProcessor。
  + 6)转换视图名称，加上前缀和后缀
  + 7)调用HandlerExecutionChain mappedHandler的applyPostHandle
  + 8)processDispatchResult（处理ModelAndView请求结果)
* 如果返回的是纯数据（@ResponseBody)，mv就是null，该方法基本上是空方法。
* 处理异常：使用handlerExceptionResolver来处理异常
* 如果handler处理结果中返回了view，那么需要对页面进行渲染，调用render方法渲染视图（需要跳转的话，在render中进行跳转)。