Spring

依赖注入

依赖注入的方式

构造器方式注入

<bean id="text" class="com.maven.Text" />
<bean id="hello" class="com.maven.Hello"><constructor-arg ref="text" /></bean>

构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。

使用构造器注入的好处

1.保证依赖不可变（final关键字）

2.保证依赖不为空

实例化Controller的时候，由于自己实现了有参数的构造函数，所以不会调用默认构造函数，那么就需要Spring容器传入所需要的参数，这样保证了依赖不为空

3.避免循环依赖

在构造器注入传入参数时，比如在A中注入B，在B中注入A。先初始化A，那么需要传入B，这个时候发现需要B没有初始化，那么就要初始化B，这个时候就出现了问题，会排除循环依赖错误，而使用filed注入方式则只能在运行时才会遇到这个问题

setter方法注入

例如：<bean id="hello" class="com.maven.Hello"><property name="text" ref="text" /></bean>

之所以叫setter方法注入，因为这是通过找到类的对应的setter方法，再进行相应的注入

Setter方法注入是容器通过调用无参构造器或无参static工厂方法实例化bean之后，调用该bean的setter方法，即实现了基于setter的依赖注入。

最好的解决方案是用构造器参数实现强制依赖，setter方法实现可选依赖。

循环依赖

因为加入singletonFactories三级缓存的前提是执行了构造器(因为要先构建出对象)，所以构造器的循环依赖没法解决。 构造器方式无法解决，只能抛出异常

因为Spring容器不缓存"prototype"作用域的bean，因此无法提前暴露一个创建中的bean。 多例方式无法解决，只能抛出异常

单例模式可以解决

通过三级缓存解决

在createBeanInstance()之后会调用addSingleton()方法将bean注册到singletonFactories中

通过提前暴露一个单例工厂方法，从而使其他bean能够引用到该bean/提前暴露一个正在创建中的bean

举例

1. “A的某个field或者setter依赖了B的实例对象，同时B的某个field或者setter依赖了A的实例对象

2.A首先完成了初始化的第一步，**并且将自己提前曝光到singletonFactories中（这步是关键）**

3.A发现自己依赖对象B，此时就尝试去get(B)，发现B还没有被create，所以走create流程

4.B在初始化第一步的时候发现自己依赖了对象A，于是尝试get(A)，尝试一级缓存singletonObjects(肯定没有，因为A还没初始化完全)，尝试二级缓存earlySingletonObjects（也没有），尝试三级缓存singletonFactories，由于A通过ObjectFactory将自己提前曝光了，所以B能够通过ObjectFactory.getObject拿到A对象

5.B拿到A对象后顺利完成了初始化阶段1、2、3，完全初始化之后将自己放入到一级缓存singletonObjects中

6.返回A中，A此时能拿到B的对象顺利完成自己的初始化阶段2、3，最终A也完成了初始化，进去了一级缓存singletonObjects中

7.由于B拿到了A的对象引用，所以B类中的A对象完成了初始化。

不用创建对象，而只需要描述它如何被创建,不在代码里直接组装组件和服务，但是要在配置文件里描述哪些组件需要哪些服务，之后一个容器（IOC容器）负责把他们组装起来。

IOC

容器的初始化过程

1.)Resource定位：指对BeanDefinition的资源定位过程。通俗地讲，就是找到定义Javabean信息的XML文件，并将其封装成Resource对象。

2.)BeanDefinition的载入：把用户定义好的Javabean表示为IoC容器内部的数据结构，这个容器内部的数据结构就是BeanDefinition。

3.)向IoC容器注册这些BeanDefinition。

bean的生命周期

1.实例化Bean

2.设置Bean的属性

3.检查Aware相关接口并设置相关依赖

BeanNameAware spring将bean的id传给setBeanName()方法

BeanFactoryAware 会调用它实现的setBeanFactory(setBeanFactory(BeanFactory)传递的是Spring工厂自身（可以用这个方式来获取其它Bean，只需在Spring配置文件中配置一个普通的Bean就可以）

ApplicationContextAware 会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文

4.检查BeanPostProcessor接口并进行前置处理 调用postProcessBeforeInitialization(Object obj, String s)方法

5.检查Bean在Spring配置文件中配置的init-method属性并自动调用其配置的初始化方法。 由于这个是在Bean初始化结束时调用那个的方法，也可以被应用于内存或缓存技术；

6.检查BeanPostProcessor接口并进行后置处理 调用postProcessAfterInitialization(Object obj, String s)方法

7.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，会调用那个其实现的destroy()方法；

8. 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。

bean知识

bean的作用域

singleton

Spring IoC容器中只会存在一个共享的Bean实例，无论有多少个Bean引用它，始终指向同一对象。

singleton是Bean的默认作用域

默认情况下是容器初始化的时候创建，但也可设定运行时再初始化bean

DefaultSingletonBeanRegistry类里的singletonObjects哈希表保存了单例对象。

Spring容器可以管理singleton作用域下bean的生命周期，在此作用域下，Spring能够精确地知道bean何时被创建，何时初始化完成，以及何时被销毁

prototype

每次通过Spring容器获取prototype定义的bean时，容器都将创建一个新的Bean实例，每个Bean实例都有自己的属性和状态

当容器创建了bean的实例后，bean的实例就交给了客户端的代码管理，Spring容器将不再跟踪其生命周期,并且不会管理那些被配置成prototype作用域的bean的生命周期。

对有状态的bean使用prototype作用域，而对无状态的bean使用singleton作用域。

request **在一次Http请求中，容器会返回该Bean的同一实例。而对不同的Http请求则会产生新的Bean，而且该bean仅在当前Http Request内有效。**

session **在一次Http Session中，容器会返回该Bean的同一实例。而对不同的Session请求则会创建新的实例，该bean实例仅在当前Session内有效。**

global Session 在一个全局的Http Session中，容器会返回该Bean的同一个实例，仅在使用portlet context时有效。

bean的创建过程

1.进入getBean()方法

2.判断当前bean的作用域是否是单例，如果是，则去对应缓存中查找，没有查找到的话则新建实例并保存。如果不是单例，则直接新建实例（createBeanInstance）

3.新建实例后再将注入属性(populateBean)，并处理回调

创建bean

找到@Autowired的对象

创建注入对象，并赋值

大致流程

1.首先根据配置文件找到对应的包，读取包中的类，，找到所有含有@bean，@service等注解的类，利用反射解析它们，包括**解析构造器，方法，属性**等等，然后封装成各种信息类放到container(其实是一个map)里（**ioc容器初始化**）

2.获取类时，首先从container中查找是否有这个类，如果没有，则报错，如果有，则通过构造器信息将这个类new出来

3.**如果这个类含有其他需要注入的属性，则进行依赖注入，如果有则还是从container找对应的解析类，new出对象，并通过之前解析出来的信息类找到setter方法(setter方法注入)，然后用该方法注入对象（这就是依赖注入）。如果其中有一个类container里没找到，则抛出异常**

4.如果有嵌套bean的情况，则通过递归解析

5.如果bean的scope是singleton，则会重用这个bean不再重新创建，将这个bean放到一个map里，每次用都先从这个map里面找。如果scope是session，则该bean会放到session里面。

总结：通过解析 xml 文件，获取到bean的属性(id,name,class,scope,属性等等)里面的内容，利用反射原理创建配置文件里类的实例对象，存入到 Spring 的 bean 容器中

SpringMVC

执行流程

1、 用户发送请求至前端控制器DispatcherServlet

2、 DispatcherServlet收到请求调用HandlerMapping处理器映射器。

3、 处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。

4、 DispatcherServlet通过HandlerAdapter处理器适配器调用处理器

5、 执行处理器(Controller，也叫后端控制器)。

6、 Controller执行完成返回ModelAndView

7、 HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet

8、 DispatcherServlet将ModelAndView传给ViewReslover视图解析器

9、 ViewReslover解析后返回具体View

10、DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。

11、DispatcherServlet响应用户

常用注解

@Compoent

@Controller

@Service

@Repository

在类定义之前添加@Component注解，它会被spring容器识别，并转为bean。

@RequestMapping

@RequestParam

@Autowired

@Resource

servlet生命周期

1.加载和实例化

2.初始化

3.请求处理

4.服务终止

Spring类

ApplicationContext

FileSystemXmlApplicationContext

ClassPathXmlApplicationContext

WebXmlApplicationContext

基本概念

Spring优点

轻量级，非侵入式 对现有的类结构没有影响

可以提供众多服务，如事务管理，WS等

AOP的很好支持，方便面向切面编程，使得业务逻辑和系统服务分开

对主流的框架提供了很好的集成支持，如hibernate,Struts2,JPA等，像一个胶水一样，把一些好的框架粘合在一起方便实用

使用Spring的IOC容器，将对象之间的依赖关系交给Spring，降低组件之间的耦合性，让我们更专注于应用逻辑

Spring DI机制降低了业务对象替换的复杂性。

Spring的高度可开放性，并不强制依赖于Spring，开发者可以自由选择Spring部分或全部

Spring缺点

缺少一个公用控制器

没有SpringBoot好用

Spring像一个胶水，将框架黏在一起，后面拆分的话就不容易拆分了

AOP

实现原理

jdk动态代理

实现过程

主要通过Proxy.newProxyInstance()和InvocationHandler这两个类和方法实现

创建代理类proxy实现Invocation接口，重写invoke()方法 调用被代理类方法时默认调用此方法

将被代理类作为构造函数的参数传入代理类proxy

调用Proxy.newProxyInsatnce(classloader,interfaces,handler)方法生成代理类

生成的代理类 $Proxy0 extends Proxy implements Person

类型为$Proxy0

因为Proxy已经继承了Proxy,所以java动态代理只能对接口进行代理

代理对象会实现用户提供的这组接口，因此可以将这个代理对象强制类型转化为这组接口中的任意一个

通过反射生成对象

总结：**代理类调用自己方法时，通过自身持有的中介类对象来调用中介类对象的invoke方法，从而达到代理执行被代理对象的方法。**

cglib

生成对象类型为Enhancer

实现原理类似于 jdk 动态代理，只是他在运行期间生成的代理对象是针对目标类扩展的子类

Spring在运行期间通过CGlib继承要被动态代理的类，重写父类的方法，实现AOP面向切面编程。

静态代理

缺点

如果要代理一个接口的多个实现的话需要定义不同的代理类

代理类 和 被代理类 必须实现同样的接口，万一接口有变动，代理、被代理类都得修改,难以维护

在编译的时候就直接生成代理类

JDK动态代理和cglib的对比

CGLib所创建的动态代理对象在实际运行时候的性能要比JDK动态代理高

1.6和1.7的时候，CGLib更快

1.8的时候，jdk更快

CGLib在创建对象的时候所花费的时间却比JDK动态代理多

singleton的代理对象或者具有实例池的代理，因为**无需频繁的创建代理对象**，所以比较适合采用CGLib动态代理，反之，则适合用JDK动态代理

JDK动态代理是面向接口的，CGLib动态代理是通过字节码底层继承代理类来实现（如果被代理类被final关键字所修饰，那么会失败）

JDK生成的代理类类型是Proxy(因为继承的是Proxy)，CGLIB生成的代理类类型是Enhancer类型

如果要被代理的对象是个实现类，那么Spring会使用JDK动态代理来完成操作（Spirng默认采用JDK动态代理实现机制）；
如果要被代理的对象不是实现类，那么Spring会强制使用CGLib来实现动态代理。

配置方式

XML方式

注解方式

基于Java类配置 通过 @Configuration 和 @Bean 这两个注解实现的

@Configuration 作用于类上，相当于一个xml配置文件；

@Bean 作用于方法上，相当于xml配置中的<bean>；

基本概念

核心业务功能和切面功能分别独立进行开发 ，然后把切面功能和核心业务功能 "编织" 在一起，这就叫AOP

让关注点代码与业务代码分离

面向切面编程就是指： 对很多功能都有的重复的代码抽取，再在运行的时候往业务方法上动态植入“切面类代码”。

应用场景：日志，事务管理，权限控制

事务管理

基本概念

如果需要某一组操作具有原子性，就用注解的方式开启事务，按照给定的事务规则来执行提交或者回滚操作

事务控制

编程式事务控制 用户通过代码的形式手动控制事务

Conn.setAutoCommite(false); // 设置手动控制事务

粒度较细，比较灵活，但开发起来比较繁琐： 每次都要开启、提交、回滚

声明式事务控制 Spring提供对事务的控制管理

XML方式

注解方式

事务属性

事务传播行为

required_new 如果当前方法有事务了，当前方法事务会挂起，在为加入的方法开启一个新的事务，直到新的事务执行完。当前方法的事务才开始

required(默认方法) 如果当前方法已经有事务了，加入当前方法事务

其余五种方式(拓展学习)

事务传播行为用来描述由某一个事务传播行为修饰的方法被嵌套进另一个方法的时候事务如何传播。

数据库隔离级别

TransactionDefinition.ISOLATION_DEFAULT: 使用后端数据库默认的隔离级别，Mysql 默认采用的 REPEATABLE_READ隔离级别 Oracle 默认采用的 READ_COMMITTED隔离级别.

TransactionDefinition.ISOLATION_READ_UNCOMMITTED: 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读

TransactionDefinition.ISOLATION_READ_COMMITTED: 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生

TransactionDefinition.ISOLATION_REPEATABLE_READ: 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。

TransactionDefinition.ISOLATION_SERIALIZABLE: 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

事务超时属性 指一个事务所允许执行的最长时间，如果超过该时间限制但事务还没有完成，则自动回滚事务。

事务只读属性 对事物资源是否执行只读操作

回滚规则 定义了哪些异常会导致事务回滚而哪些不会。默认情况下，事务只有遇到运行期异常时才会回滚，而在遇到检查型异常时不会回滚，也可以用用户自己定义

Spring事务管理接口

PlatformTransactionManager （平台）事务管理器

Spring并不直接管理事务，而是提供了多种事务管理器，通过PlatformTransactionManager这个接口，Spring为各个平台如JDBC、Hibernate等都提供了对应的事务管理器，但是具体的实现就是各个平台自己的事情了。

TransactionDefinition 事务定义属性(事务隔离级别、传播行为、超时、只读、回滚规则)

TransactionStatus 事务运行状态

事务管理一般在Service层

如果在dao层，回滚的时候只能回滚到当前方法，但一般我们的service层的方法都是由很多dao层的方法组成的

如果在dao层，commit的次数会过多

相关设计模式

1.单例模式 实现方式 Spring依赖注入Bean实例默认是单例的

2.工厂模式 实现方式

静态工厂 beanFactory

工厂模式 factoryBean

3.代理模式 实现方式 AOP中的JDK动态代理和CGLib

4.适配器模式

实现方式 SpringMVC中的适配器HandlerAdatper

实现原理 HandlerAdatper根据Handler规则执行不同的Handler

5.装饰器模式 实现方式 Spring中用到的包装器模式在类名上有两种表现：一种是类名中含有Wrapper，另一种是类名中含有Decorator。

7.观察者模式

实现方式 spring的事件驱动模型使用的是 观察者模式 ，Spring中Observer模式常用的地方是listener的实现

具体实现 事件机制的实现需要三个部分,事件源,事件,事件监听器

8.策略模式 实现方式 Spring框架的资源访问Resource接口 。该接口提供了更强的资源访问能力，Spring 框架本身大量使用了 Resource 接口来访问底层资源

9.模板方法模式

参考博客(写的很详细很棒) https://blog.csdn.net/aa1215018028/article/details/81703900